**1. Введение**

Современная вычислительная машина была изобретена и мыслилась как некоторое устройство, облегчающее и убыстряющее проведение сложных расчетов, требующих на свое выполнение значительного времени. Теперь в большинстве случаев их доминирующим свойством и основной характеристикой считается способность хранить огромные объемы информации, к которым можно просто обратиться. Способность же проводить вычисления, т. е. выполнять арифметические действия, во многих ситуациях уже стала почти несущественной.

Во всех таких случаях значительные объемы информации, подлежащей обработке, в некотором смысле представляют ***абстракцию*** некоторого фрагмента реального мира. Информация, поступающая в машину, состоит из определенного множества данных, относящихся к какой-то проблеме, — это именно те данные, которые считаются относящимися к данной конкретной задаче и из которых, как мы надеемся, можно получить (вывести) желанный ответ. Мы говорим о данных как об абстрактном представлении реалий, поскольку некоторые свойства и характеристики объектов при этом игнорируются. Считается, что для конкретной задачи они не являются существенными, определяющими. Поэтому абстрагирование — это *упрощение* фактов.

В качестве примера можно взять, скажем, список (файл) некоторых служащих. Каждый служащий в этом списке представляется (абстрагируется) как некоторое множество данных, относящихся либо к тому, что он делает, либо к процедурам, "обрабатывающим" его деятельность. Это множество может включать и некоторые идентифицирующие данные вроде его (или ее) фамилии или оклада. Однако маловероятно, что сюда будут включаться такие посторонние сведения, как цвет волос, вес или рост.

Решая любую задачу с помощью машины или без нее, необходимо выбрать уровень абстрагирования, т. е. определить множество данных, представляющих реальную ситуацию. При выборе следует руководствоваться той задачей, которую необходимо решить. Затем надлежит выбрать способ представления этой информации. Здесь уже необходимо ориентироваться на те средства, с помощью которых решается задача, т. е. учитывать возможности, предоставляемые вычислительной машиной. В большинстве случаев эти два этапа не бывают полностью независимыми.

Выбор представления данных часто является довольно трудной проблемой, ибо не определяется однозначно доступными средствами. Всегда нужно принимать во внимание и операции, которые выполняются над этими данными. Вот хороший пример: представление чисел. Они сами по себе уже абстракции некоторых свойств объектов, которые следует как-то охарактеризовать. Если единственной (или доминирующей) операцией будет сложение, то хорошим способом представить число*п* будет просто последовательность из*п* "черточек". При таком представлении правило сложения действительно и очевидно, и просто. В основу "римского" представления положен тот же принцип простоты, и правила сложения для небольших чисел столь же просты. С другой стороны, "арабское" представление чисел требует далеко не очевидных правил (даже для небольших чисел), правила требуют запоминания, выучивания наизусть. Ситуация, однако, изменяется, если речь заходит о сложении больших чисел или введении умножения и деления. Разложение таких операций на более простые оказывается значительно проще в случае арабского представления, что объясняется систематичностью представления, основанного на "позиционном весе" цифр числа.

Хорошо известно, что в основе внутреннего представления чисел в вычислительных машинах лежат двоичные цифры. Для человека такое представление не подходит: уж очень велико число таких цифр. Зато оно очень подходит для электронных схем: значения 0 и 1 можно удобно и надежно кодировать наличием или отсутствием электрического тока или магнитного поля.

На этом примере можно обнаружить, что вопросы представления часто разбиваются на несколько уровней детализации. Представьте себе, что необходимо задать положение некоторого объекта. Первое решение может привести к выбору пары вещественных чисел в прямоугольной или полярной системе координат. Второе решение — к представлению с плавающей запятой, где каждое вещественное число *х* состоит из пары целых чисел, обозначающих дробную часть*f* и показатель степени *е* некоторого основания (так, что *х = f \* 2е*). Третье решение — оно основывает на знании того, как в машине хранятся данные, — может привести к двоичному, позиционному представлению для целых чисел. И последнее, двоичные цифры можно кодировать направлением магнитного поля в "магнитном запоминающем устройстве". Очевидно, что первое из решений в этой цепочке зависит главным образом от постановки задачи, а последующие все больше и больше — от используемого инструмента и методов работы с ним. Так, вряд ли можно требовать, чтобы программист решал какое следует использовать представление чисел или каковым будут характеристики запоминающего устройства. Такие решения низших уровней можно оставлять создателям самих машин, они лучше других информированы о сегодняшней технологии влияющей на выбор, который затем будет распространен на все (или почти все) приложения, связанные с использованием чисел.

В таком контексте становится очевидным значение*языков программирования*. Любому языку программирования соответствует некоторая абстрактная машина, способная интерпретировать понятия, используемые в этом языке. Это тоже некоторый уровень абстрагирования от реальных устройств, использованных в существующих машинах. Таким образом программист, использующий такой язык высокого уровня, освобождается (и ограждается) от вопросов представления чисел, если числа в контекст этого языка — элементарные объекты.

Значение использования любого языка, представляющего подходящее множество основных абстракций, общих для большинства задач обработки данных, заключается главным образом в том, что это приводит к получению надежных программ.

Легче строить программы, основываясь на таких знакомых понятиях, как числа, множества, последовательности и повторения, вместо разрядов, ячеек и переходов. Конечно же, в реальной машине все данные, будь то числа, множества или последовательности, выглядят как огромные массивы разрядов. Но для программиста все это не имеет значения, поскольку он не беспокоится о деталях представления выбранной абстракции, ибо уверен, что на машине (или в трансляторе) выбрано наиболее уместное представление.

Чем ближе абстракция к конкретной машине, тем легче для инженера или реализатора языка выбрать подходящее представление, тем выше вероятность, что это единственное представление подойдет для всех (или почти всех) мыслимых приложений. Этот факт устанавливает определенные пределы на степень абстрагирования от заданной машины. Например, не имеет смысла включать в основные элементы универсального языка геометрические объекты, поскольку их удобное представление из-за присущей им внутренней сложности будет в значительной мере определяться теми операциями, которые над ними совершаются. Природа и частота этих операций создателю универсального языка и его транслятора неизвестны, и какое бы представление он ни выбрал, оно может оказаться неподходящим при некоторых применениях.

Ясно, что нам хотелось использовать привычные математические понятия чисел, множеств, последовательностей и т. п., а не присущие машинам понятия вроде строк разрядов. Столь же очевидно, что мы хотели использовать и понятия, для которых, и это уже известно, существуют эффективные трансляторы. Причем не хотелось бы как употреблять машинно-ориентированный или машинно-зависимый язык, так и описывать программы для машины в абстрактных понятиях, оставляющих открытыми вопросы представления.

**1.2.Алгоритмы и данные**

**Решение задачи с помощью ЭВМ** — это процесс автоматического преобразования исходной информации (исходных данных) в искомый результат в соответствии с заданной последовательностью действий, называемой алгоритмом.

***Алгоритм***— одно из основных понятий математики, которое невозможно строго определить, а можно лишь пояснить с помощью других слов. Можно сказать, что алгоритм — это точное предписание, которое задает вычислительный процесс, начинающийся из некоторой совокупности возможных для этого алгоритма исходных данных и направленный на получение полностью определяемого этими исходными данными результата.

**СВОЙСТВА АЛГОРИТМОВ**

Алгоритмы должны обладать следующими свойствами:

* [Конечность](https://math.gsu.by/wp-content/uploads/courses/structure/m1.html)
* [Определенность](https://math.gsu.by/wp-content/uploads/courses/structure/m2.html)
* [Массовость](https://math.gsu.by/wp-content/uploads/courses/structure/m3.html)
* [Результативность](https://math.gsu.by/wp-content/uploads/courses/structure/m4.html)
* [Эффективность](https://math.gsu.by/wp-content/uploads/courses/structure/m5.html)

**ЭТАПЫ ПОДГОТОВКИ ЗАДАЧИ К РЕШЕНИЮ**

Основными этапами подготовки задачи к решению на ЭВМ, которые связаны с построением алгоритма, являются:

* Постановка задачи.
* Построение модели задачи.
* Разработка алгоритма.
* Реализация алгоритма в виде программы.
* Проверка правильности алгоритма.
* Анализ сложности алгоритма.

**Постановка задачи.** Она позволяет точно сформулировать и понять задачу. В первую очередь необходимо установить, имеет ли исходная задача решение. Далее следует выяснить, нельзя ли данную задачу решить посредством готовых прикладных программ (приложений), таких, как электронные таблицы, системы управления базами данных и т.п.

На этапе постановки задачи мы должны получить ответ на следующие вопросы: что дано? достаточно ли исходных данных? нет ли избыточных? какие результаты должны быть получены? какие сделаны допущения? Поскольку решаемая задача относится к некоторой области реального мира (предметной области), мы должны выделить эту область, игнорируя некоторые ее свойства и характеристики, не свойственные с точки зрения рассматриваемой задачи. Так мы получаем абстракцию некоторого фрагмента реального мира, т.е. упрощаем факты до такой степени, что обработка абстрактных данных позволит получить желаемый ответ (результаты) по рассматриваемой проблеме.

На этом этапе предметную область мы видим глазами пользователя, специалиста по данной предметной области, естественно, и данные представляются в понятиях и обозначениях этой области. Это ***внешний уровень представления данных***. Для решения же задачи на ЭВМ эти данные должны быть представлены в машинном виде, на***физическом уровне*.**

Каждая модель ЭВМ имеет свою систему команд и ограниченный состав базовых данных, а также форматы их представления. В качестве базовых данных обычно используются двоичные и десятичные числа с фиксированной запятой, числа с плавающей запятой, символы и т.д. Формат данных определяет, какой размер памяти занимают отдельные данные, как трактуются отдельные части и данные в целом. Различают форматы данных фиксированной длины (один или несколько байтов) и переменной длины, в любом случае данные размещаются в одной или нескольких смежных ячейках (байтах). Система команд определяет состав операций, выполняемых в ЭВМ над данными. Нас не интересует, на каком уровне — аппаратном или микропрограммном они выполняются. Каждая команда также имеет свой формат длиной в один или несколько байтов.

В ячейках памяти размещаются либо данные, либо команды (коды программы). Таким образом, каждая ячейка имеет свой адрес и некоторое содержимое. По содержимому ячейки невозможно определить, что там хранится: данные, команда или их части. Например, содержимое байта '0011000 Г может трактоваться как символ Т, десятичное число 31, двоичное число, равное 49, и т.д. Из вышеизложенного следует, что в модели ЭВМ на машинном уровне определены только базовые типы данных, которые, естественно, в большинстве случаев не совпадают с типами данных, полученными при абстрагировании. Следовательно, существует разрыв между двумя уровнями представления данных. Этот разрыв постепенно устраняется на последующих этапах подготовки к решению задачи.

**Построение модели задачи**. После того как получена четкая постановка задачи, нужно подобрать или разработать формальную модель задачи, привлекая достижения современных наук. Выбранная модель существенно влияет на все последующие этапы. Выбор модели в высокой степени зависит от подготовленности, знаний, эрудиции, опыта разработчиков. Здесь нужно установить, какие структуры данных больше всего подходят нашей задаче с точки зрения удобства представления и простоты решения. **СТРУКТУРА ДАННЫХ** Под***структурой данных*** понимаем организованную некоторым образом информацию, которая может быть описана, создана и обработана в программах. Теперь на данные решаемой задачи начинаем смотреть с учетом их обработки на машине. Следовательно, в разработке формальной модели задачи при активном взаимодействии должны участвовать специалисты по предметной области и программисты в широком смысле этого слова.

Имеется большое число абстрактных структур, разработанных и детально изученных поколениями ученых. Рассмотрены способы их представления, области рационального применения, основные операции над ними и предложены всевозможные алгоритмы выполнения этих операций. Абстрактные данные, полученные на этапе постановки задачи, теперь конкретизируются и представляются с использованием подходящих структур.

**Разработка алгоритма.** Методы и алгоритмы решения задачи ищутся и разрабатываются в рамках построенной формальной модели задачи. При этом необходимо выяснить, существуют ли решения аналогичных или похожих задач, т.е. руководствоваться накопленным опытом, что позволяет значительно сократить сроки разработки и уменьшить требуемые расходы.

На этом этапе выделяются так называемые ***абстрактные типы данных*(**АТД), которые определяют организацию (структуру) и область значений конкретных данных задачи и операции над этими данными. АТД позволяют сосредоточиться на идеализированных моделях данных и операциях над ними и отвлечься от программной реализации. В то время как над базовыми типами данных, представленными на машинном уровне, выполняются простые машинные операции, операции над АТД являются достаточно сложными и, как правило, реализуются отдельными процедурами (функциями). Разработка алгоритмов таких процедур облегчается, если АТД представлены с использованием абстрактных структур, упомянутых выше.

**Реализация алгоритма в виде программы.** Это довольно трудный этап: сначала нужно выбрать язык или языки программирования. При этом необходимо учесть, что каждый язык поддерживает одну или несколько парадигм программирования.

***Парадигмы программирования*** — это модели разработки и реализации программ. Разные модели приводят к разным приемам программирования, они обычно не противоречат друг другу, а являются взаимодополняющими. Общим для всех моделей является то, что разработка программы базируется на абстракциях, соответствующих элементам решаемой задачи, а реализация представляет собой совокупность модулей. Различие моделей состоит в том, как формировать абстракции и что содержит модуль.

Язык процедурного программирования базируется на модели построения программы как совокупности функций, а абстракция данных базируется на структурах данных. Объектно- ориентированное программирование базируется на модели построения программ как набора объектов абстрактного типа данных. В каждом языке программирования имеются средства представления данных различных типов. Тип данных определяет, какие значения могут принимать данные этого типа и допустимые операции над ними; Абстрактные типы данных, определенные на этапах разработки модели и алгоритма, представляются теперь средствами языка программирования в виде синтаксических конструкций, т.е. получают программную реализацию. Это ***промежуточный уровень*** представления данных. Затем в результате трансляции программы данные представляются в виде последовательностей битов — это ***физический уровень*** представления данных. Таким образом, в процессе последовательных преобразований структуры данных из одного вида представления переводятся в другие (рис. 1.):



**Уровни представления данных**

Окончательная структура данных определяется программистами на промежуточном уровне, поскольку данные, представленные в программе, однозначно переводятся транслятором в двоичные коды и образуют структуры хранения данных.

**Проверка правильности алгоритма.** Она выполняется на различных этапах. На этапе разработки алгоритма доказывается правильность каждого шага алгоритма, затем доказывается конечность алгоритма, проверяются допустимые входные данные и полученные по ним выходные данные. На этапе программной реализации алгоритма необходимо следить за тем, чтобы в процессе преобразования правильного алгоритма получить правильную программу.

***Правильностью (корректностью) программы*** называют свойство программы, характеризующее отсутствие в ней ошибок по отношению к целям разработки [35]. Каждая программа определяется правилами ее представления (синтаксисом языка программирования) и правилами интерпретации ее содержательного значения (семантикой). Корректная программа может быть создана только при строгом соотношении этих правил. Задача определения синтаксической правильности решается транслятором автоматически в ходе трансляции программы. Проблема доказательства семантической правильности оказывается значительно сложнее, так как процедура доказательства существенно зависит от решаемой задачи и исходных данных. Формализованное описание постановки задачи называют ***спецификацией задачи****.* Область науки об информатике, занимающаяся доказательством соответствия между программной реализацией и спецификацией задачи, получила название ***верификации программ.*** Цель верификации программ — демонстрация корректности программ. Определенные успехи, достигнутые в автоматизации верификации программ, пока не дошли до широкого практического применения.

Семантическая правильность программы традиционно проверяется путем ее тестирования. ***Тестирование***— это процесс выполнения программы с целью нахождения ошибок. Однако выполнить всестороннее тестирование сложной программы во всех диапазонах исходных данных и при всех их сочетаниях практически невозможно.

**Анализ сложности алгоритма.** Алгоритму решения задачи предъявляются два противоречивых требования:

1) быть простым для понимания, отладки и сопровождения;

2)быть эффективным в смысле затрат времени и памяти ЭВМ.

Несмотря на то, что объемы памяти и быстродействие ЭВМ выросли колоссально, такой подход к оценке эффективности продолжает оставаться справедливым.

Какому из этих требований отдать предпочтение, определяется характером решаемой задачи. Если программа выполняется несколько раз, а время выполнения для пользователя несущественно, то разумнее отдать предпочтение первому требованию, так как стоимость создания программы намного дороже стоимости машинного времени, затрачиваемого на решение задачи.

Если же задача для своего решения требует значительных вычислительных ресурсов, выполняется многократно либо время решения ограничено внешними событиями как в системах реального времени, то требование эффективности становится более важным. Обычно эффективные алгоритмы реализуются более сложными программами, поэтому желательно, чтобы такие программы по возможности были простыми в сопровождении. С одной стороны, целью анализа алгоритма является получение оценок (границ) для объема памяти или времени работы, которое потребуется алгоритму для обработки конкретных данных. С другой стороны, для решения одной и той же задачи могут быть использованы различные по эффективности алгоритмы. Тогда по результатам анализа выбирается наиболее эффективный из них.

При анализе с задачей связывается некоторое число, называемое ***размером задачи****,* которое выражает меру количества (объема) исходных данных. В качестве размера могут выступать размеры массивов, число вершин или дуг графа, число записей в файле и т.п. Время выполнения программы *Т(п)* зависит от размера *п*, единица измерения *Т(п)* не определена. Традиционно определяют время выполнения *Т(п)* в наихудшем случае как меру сложности алгоритма.

Время, затрачиваемое алгоритмом на его выполнение как функция размера задачи, называется ***временной сложностью*** этого алгоритма. Поведение этой сложности в пределе при увеличении размера задачи и называется *асимптотической временной сложностью*. Аналогично определяются *емкостная сложность* и*асимптотическая емкостная сложность*. Асимптотическая сложность алгоритма в конечном счете и определяет размер задач, которые можно решить с помощью этого алгоритма.

Если алгоритм *Л* имеет время выполнения *ТА(п) = сп1*, а алгоритм *В — Т(п) = kdn,* где *с, d,k* — константы, то в зависимости от их соотношения может оказаться, что для малых *п* более эффективным является алгоритм *В*, в то же время очевидно, что для больших я более эффективным будет алгоритм *А*. Это связано с тем, что функций *ТА(П)* и *ТB(п)* имеют различные скорости роста.

Для описания скорости роста функций используется 0-символика. Так, *Т(п) = О(пА)* означает, что время выполнения программы *Т(п)* имеет порядок *О(п )* и подразумевается, что существуют положительные константы *с* и *n0* такие, что для всех *n*, больших или равных *n0*, выполняется неравенство *T(п)≤ сп2*. Аналогично *T(п)* имеет порядок *O(f(n))*, если существуют константы *с* и *n0*, такие, что для всех *п* = *n0* выполняется неравенство *Т(п)≤cf(n)*, тогда*f(n)* определяет степень роста. Говорят, что *f(n)* является *верхней границей* сложности задачи. Если мы пытаемся улучшить алгоритм с целью снизить время выполнения задачи, то возникает вопрос, где необходимо остановиться, так как дальнейшее улучшение невозможно, т,е. найти ***нижнюю границу***сложности алгоритма. Определение нижней границы вызывает существенные затруднения и потому редко выполняется.

Определим сложность алгоритма сортировки методом пузырька массива *А* с *п* элементами. Основу алгоритма составляет двойной цикл.

**for (i=0; i<n-1; i++)**

**for(j=n-1;j<i;j--)**

**If (А[j]<А[j-1])**

**<перестановка элементов A[j] и А[j-1]>**

Перестановка элементов требует фиксированного времени. Для каждого *i*, устанавливаемого внешним циклом по *i*, внутренний цикл по *j* выполняется *(n-i)* раз, общее количество шагов определяется суммой.

Следовательно, в худшем случае количество перестановок равно этому же значению. Таким образом, алгоритм метода пузырька имеет временную сложность *О(п2)*, если даже в некоторых шагах (или даже всех) перестановок элементов не будет.

Если при некотором *i* не будет ни одной перестановки, то это означает, что массив уже отсортирован и процесс сортировки можно прекратить. Для этого достаточно ввести управляющий переключатель *р*, который устанавливается в нуль" в начале внешнего цикла. Если во внутреннем цикле перестановок не было, его значение не изменяется, и по этому признаку осуществляется выход из внешнего цикла. Очевидно, что временная сложность будет равна *0(l)*, если исходный массив был отсортирован, и *О(n2)*— в худшем случае, когда массив был отсортирован в обратном порядке.

При вычислениях времени выполнения программ с применением О - символики пользуются правилами сумм и произведений.

***Правило сумм*** заключается в следующем.

Пусть время выполнения двух фрагментов программы *Т1(п)* и *Т2(п)* имеет порядок*O(f11(n))* и *O(f2(n))* соответственно. Время последовательного выполнения этих фрагментов вычисляется как *Т1(п) + Т2(п) = O(f1(n)) + О(f2(n))* и имеет порядок *0(max(f1(n), f2(n)))*

По ***правилу произведений***: *Т1(п) • Т2(п) = O(f1(n)\*f2(n))*. Тогда с применением этих правил можно определить время выполнения операторов базовых структур программ.

1)Время, выполнения последовательности операторов определяется по правилу сумм, т.е. равно наибольшему времени выполнения одного оператора в данной последовательности.

2)Время выполнения оператора ветвления определяется наибольшим временем выполнения одной из ветвей оператора.

3)Время выполнения цикла определяется как произведение времени выполнения тела цикла на количество повторений цикла.

При анализе сложности алгоритмов вводится понятие ***детерминированных* и *недетерминированных*** алгоритмов. В каждый момент времени выполнения алгоритм находится в некотором определенном состоянии. Состояние определяется совокупностью значений обрабатываемых данных и выполняемым действием. **Свойство *детерминированности*** означает, что для каждого имеющегося в данный момент состояния либо существует в точности одно следующее состояние, либо не существует никакого следующего — это заключительное состояние.

Пусть теперь для любого состояния алгоритма возможно иметь более одного следующего. Когда детерминированный алгоритм доходит до того места, где требуется выбрать одну из альтернатив, он исследует их последовательно друг за другом и выбирает один единственный. ***Недетерминированный* алгорит**м просматривает все альтернативы не последовательно, а все их одновременно, копируя самого себя. Эти копии, в свою очередь, могут порождать новые копии и т.д. Все копии продолжают выполняться все вместе до тех пор, пока одна или несколько из них не найдут решения, после этого останавливаются все копии.

Все задачи, которые могут быть решены с помощью детерминированного алгоритма за полиномиальное время, образуют, один класс, класс *Р - задач*, и считаются легко разрешимыми. Задача, сложность которой не менее *fn*(где*f* — константа, или полином от *n*), считается экспоненциальной и относится к *классу Е*. При небольших значениях *п* экспоненциальный алгоритм, как отмечалось выше, может быть более «быстрым», чем полиномиальный. Но различие между этими алгоритмами при больших значениях *п* проявляется всегда.

Есть задачи, которые не попадают ни в класс *Р*, ни в класс *Е*. В настоящее время для этих задач не найдено полиномиального алгоритма. Доказано, что они являются взаимно эквивалентными. Это означает, что если бы для одной из них удалось найти полиномиальное решение, то все они также получили бы полиномиальное решение.

Задачи, которые могут быть решены с помощью недетерминированного алгоритма за полиномиальное время, относятся к классу ***NP -задач***. Очевидно, что *P?NP*, т.е. каждая задача, разрешимая по детерминированному алгоритму за полиномиальное время, также разрешима за полиномиальное время по недетерминированному алгоритму. Истинность обратного утверждения не доказана. Она является одним из неразрешенных вопросов современной математики и информатики.

А теперь, кратко рассмотрим вопросы, относящиеся к *NP*-задачам.

*Определение 1.* Задача *Q* сводится к задаче R тогда и только тогда, когда для всякого решения s задачи *R* существует функция *g(s)*, которую можно вычислить полиномиально и которая является решением задачи *Q (Q—>R)*. Это означает, что если мы умеем решать задачу *R*, то мы умеем решать и задачу *Q*.

*Определение 2.* Если одновременно *Q->R* и *R->Q*, то говорят, что *Q* и *R* эквивалентны.

(*Определение 3.* Задача называется *NP-сложной (NP-трудной)* тогда и только тогда, когда к ней сводима любая задача из класса*NP*. Сама *NP-сложная* задача может принадлежать как к классу *Р*, так и к классу *NP*.

*Определение 4.* Задача называется *NP-полной* в том и только в том случае, если она является *NP-сложной*и попадает в класс *NP*. *Задача разрешимости логического выражения* заключается в следующем. Пусть дано выражение Е, которое включает логические операторы ИЛИ, И, НЕ и логические переменные q1, q2,,…qn. Нужно найти такие значения переменных *qi* чтобы при этих значениях выражение E(q1,q2,..., qn) было *истинным*. Эта задача попадает в класс *NP*.

В 1971 г. Куком была доказана теорема: *задача разрешимости логического выражения является NP-сложной.*

Так как задача разрешимости принадлежит к классу *NP*, то она является ***NP-полной*.**

Данная теорема утверждает, что решение любой задачи из класса *NP* может быть получено из решения задачи разрешимости путем некоторой трансформации, которая тоже полиномиальна. Иначе, для решения произвольной задачи класса *NP* достаточно уметь решать задачу разрешимости. Трудность заключается в том, что мы не умеем как следует решать задачу разрешимости.

В настоящее время доказано, что большинство действительно сложных классических задач принадлежит к классу *NP-сложных*, и, более того, они являются *NP-полными*. В этом смысле все они эквивалентны. Поэтому если бы мы умели решать одну из них за полиномиальное время, то мы смогли бы решать все задачи этого класса также за полиномиальное время.

Относительно *NP-полных* задач можно сказать, что, во-первых, мы не знаем, являются ли они не полиномиальными и, во-вторых, не знаем детерминированного полиномиального алгоритма для их решения. Но в конкретных случаях можно использовать принципы моделирования и преобразования условий. Даже если некоторая общая задача *Q* принадлежит к классу *NP-полных задач*, то любая конкретная задача с уточненными данными может быть все-таки решена, возможно, даже за полиномиальное время. Единственное, что вытекает из теории, это то, что для общего случая не существует единого полиномиального алгоритма.

**Простейшие типы данных**

Любой новый простейший тип определяется простым перечислением входящих в него различных значений. Такой тип называется ***перечисляемым*.**

Определение таких типов вводит не только идентификатор нового типа, но и одновременно множество идентификаторов, обозначающих значения этого нового типа. Затем везде в программе и| можно уже использовать как константы, что в значительной степени способствует наглядности.

Если предполагается, что переменные s, d, г и b определяются как целые числа константы в порядке их перечисления отображаются в натуральные числа. Кроме этого, транслятор может проверить состоятельность использования той или иной операции.

Поскольку перечисления задают упорядоченность, то перечисляемые типы будут весьма чувствительными к введению операций, порождающих для заданного аргумента предыдущие или последующие значения.

   **1.1 Простейшие стандартные типы**

К стандартным простейшим типам мы относим типы, имеющиеся (встроенные) на большинстве вычислительных машин. Сюда входят целые числа, логические истинностные значения и множество печатаемых символов. На многих машинах есть и дробные числа вместе с соответствующими стандартными арифметическими операциями. В Си обозначают эти типы следующими идентификаторами:

int, float, char.

Тип int включает в себя некоторое подмножество целых чисел, размер которого варьируется от машины к машине. Если для представления целых чисел в машине используется *п* разрядов, причем используется дополнительный код, то допустимые числа должны удовлетворять условию *–2n-1< х < 2n-1*. Считается, что все операции над данными этого типа выполняются точно и соответствуют обычным правилам арифметики. Если результат выходит за пределы представимого множества, то вычисления будут прерваны. Такое событие называется *переполнением*. Четыре основные арифметические операции считаются стандартными: сложение (+) вычитание (-), умножение (\*) и деление (/). Мы будем делать отличие между ***эйлеровской целой арифметикой*** и модульной; в первой деление обозначается косой чертой, а взятие остатка от деления — %. Пусть частное *q = m/п*, а остаток *r = m % п*. Всегда справедливо отношение *q\*n + r=m* и 0 < |r| < |*n*|.

Знак остатка тот же, что и у делителя (или же остаток равен нулю). Следовательно, эйлерово целое деление несимметрично относительно нуля и для него справедливы равенства:

(-m) /п = m/ (-n) = - (m / п).

Примеры:

31/10=3 31%10=1 -31/10=-3 -31%10=-1

31/-10=-3 31%-10=1 -31/-10=3 -31%-10=-1

В модульной (конгруэнтной) арифметике значение *m* % n фактически есть некоторый класс конгруэнтности, т. е. множество целых, а не единственное число. В это множество входят все числа вида *m - Q \* п*для произвольных*Q*. Очевидно, что такие классы можно представлять одним специфическим элементом, например самым маленьким из неотрицательных элементов. Следовательно, мы определяем *R = m MOD n* и в то же время *Q= m / n* так что удовлетворяются соотношения:

Q\*n+R=m и 0≤R≤n.

Примеры:

31/10=3 31 MOD 10=1

-31/10=-4 -31 MOD 10=9

Необходимо обратить внимание, что деление на 10n можно выполнять, сдвигая десятичное число вправо на *п* позиций, причем "выдвигаемые" цифры просто игнорируются. Этот же прием допустим, когда число представлено не в десятичном, а в двоичном виде. Если мы имеем дело с представлением в дополнительном коде (а на большинстве современных машин именно так и обстоит дело), то сдвиг реализует деление, определенное выше как операция << (а не операция /). Умеренно сложные трансляторы поэтому представляют операции вида *m / 2n*с помощью быстрой oпepaции сдвига, однако это упрощение к выражениям типа *m / 2n*не применимо.

Если про некоторую переменную известно, что она не принимает отрицательных значений (скажем, это какой-то счетчик), то это свойство можно подчеркнуть, сославшись на дополнительный стандартный тип Unsigned. Если на машине для представления целых (без знака) используется *п* разрядов, то переменым такого типа можно присваивать величины, удовлетворяющие условию *0 ≤ х < 2n.* Тип float обозначает подмножество вещественных чисел. Считается, что арифметика с операндами типа int дает точный результат, однако аналогичные действия со значениями типа float могут быть неточными в пределах ошибок округлений, вызванных вычислениями с конечным числом цифр. Это принципиальное различие, и оно привело в большинстве языков программирования к явно различным типам int и float.

Два значения двоичного типа BOOLEAN обозначаются идентификаторами TRUE и FALSE. Булевские операции — это логические конъюнкция, дизъюнкция и отрицание. Их значения (результаты) определены в табл. 1.1. Логическая конъюнкция обозначается через AND (или &), логическая дизъюнкция — OR, а отрицание — NOT (или -). Обращаем внимание, что операции сравнения дают результат типа BOOLEAN. Таким образом, результат некоторого сравнения можно присвоить какой-то переменной или же использовать в качестве логического операнда в булевском выражении. Например, для булевских переменных/) и *q* и целых переменных *х*= 5, г/ = 8,2=10 два присваивания:

*q:=(x<y)&(y<z);*

породят результаты: *р* = FALSE и *q* = TRUE.

Таблица 1.1. Логические операции

|  |
| --- |
|  |
|  |
| р | q | p AND q | p OR q | NOT p |
|  |  |  |  |  |
| TRUE | TRUE | TRUE | TRUE | FALSE |
|  |  |  |  |  |
| TRUE | FALSE | TRUE | FALSE | FALSE |
|  |  |  |  |  |
| FALSE | TRUE | TRUE | FALSE | TRUE |
|  |  |  |  |  |
| FALSE | FALSE | FALSE | FALSE | TRUE |

Логические операции AND и OR в Си Паскале (и в некоторых других языках) обладают дополнительными свойствами, отличающими их от других бинарных операций. В то время когда, например, сумма *х + у* не определена, если*х* или *у* неизвестны, конъюнкция *р & q* определена даже при неизвестном *q*, если известно, что *р* имеет значение FALSE. Такая "условность" является важным и полезным свойством. Поэтому точное определение AND и OR дается такими двумя уравнениями:

р AND *q* = если *р*, тогда *q*, иначе FALSE,

р OR *q* = если *р*, тогда TRUE, иначе *q*.

В стандартный тип char входит множество печатаемых символов. К несчастью, не существует такого стандартного множества символов, которое было бы принято на всех вычислительных машинах. Поэтому использование определения "стандартное во многих случаях может приводить к неверному пониманию: и ситуацию error следует воспринимать как стандартное для той вычислительной машины, на которой следует выполнить некоторую программу.

Наиболее широко распространено множество символов, принятое в качестве стандартного Международной организацией по стандартизации (ISO), и в частно-сти его американская версия ASCII (American Standard Code for Information Inter-change). В частности, это множество состоит из 95 печатаемых (графических) символов и 33 управляющих, последние используются главным образом при передаче данных и для управления печатающими устройствами.

Чтобы иметь возможность составлять алгоритмы, манипулирующие символами (т. е. значениями типа char) и не зависящие от системы, следует все же выделить некоторые из свойств, присущих множеству символов, а именно:

* Тип CHAR содержит 26 прописных латинских букв и 24 строчных, 10 арабских цифр и некоторое число других графических символов, например знаки пунктуации.
* Подмножества букв и цифр упорядочены и "соприкасаются", т. е. («A»≤х)&(х≤»Z») - прописная буква, («а»≤)&(х≤»z») - строчная буква, («0»≤)&(х≤»9») - цифра.
* Тип CHAR содержит некоторый непечатаемый символ — пробел; его можно использовать как разделитель. Когда заходит речь о написании программ в машинно-независимой форме, то особенно важно указать, что существуют две стандартные функции преобразования типов между char и int. Мы будем называть их isalnum(ch) (дает порядковый номер ch во множестве символов) и toascii(j) (дает символ с порядковым номером j).
* **1.2.Ограниченные типы (диапазоны)**
* Часто приходится сталкиваться с положением, когда переменной присваивается значение некоторого типа, лежащее только внутри определенного интервала значений. Такое положение можно подчеркнуть, определив, что указанная переменная относится к ограниченному типу (диапазону). Такой тип задается следующим образом:
* TYPE T=[min..max],
* где min и max выражения, определяющие концы такого диапазона. Необходимо отметить, что операндами этих выражений могут быть только константы, (язык Модула 2).
* Примеры:
* TYPE year =[1900..1999];
* TYPE letter =[`A`..`Z`];
* TYPE digit =[`0`..`9`];
* Если теперь определить переменные
* VAR y: year;
* VAR L: letter;
* то присваивание y:=1984 и L=:`L` допускаются, а y:=1291 и L=:`9` - нет.
* Однако в легальности подобных присваивания можно удостовериться без выполнения программы лишь в тех частях, когда речь идет о присваивании констант. Справедливости же присваивания *у := i* и *L := с*, где*i* — переменная целого типа,*с* - символьного, транслятор не может определить, если он только просматривает текст. Системы, ведущие такие проверки в процессе выполнения самих программ, оказываются очень ценными для их разработки. Использование трансляторами избыточной информации для выделения возможных ошибок еще раз объясняет стремление применять языки высокого уровня.

**1.3 Вектор**

**Вектор** является наиболее простой структурой хранения многоэлементных [структур данных.](https://math.gsu.by/wp-content/uploads/courses/structure/l1.2.html#top2) Вектор размещается в одной сплошной области памяти. Элементы структуры данных в векторе физически размещаются последовательно один за другим. Поэтому в элементе структуры данных имеется только элемент данных и отсутствует элемент отношений, показывающий связи элемента структуры данных с другими элементами.

Такая структура хранения наиболее близко соответствует реальной организации памяти большинства ЭВМ. Все элементы структуры данных имеют одинаковую длину. Доступ к отдельному элементу осуществляется с использованием базового адреса (адреса начала) вектора и номера этого элемента. Обычно в векторной памяти размещаются массивы (одномерные и многомерные), строки, стеки, очереди, таблицы и т.п.

Для манипулирования со структурой данных, хранимой в векторе, требуется управляющая информация, определяющая адрес начала вектора, тип элементов структуры данных и их количество в векторе и т.п. Для хранения управляющей информации создается дескриптор (описатель) вектора, или информационный вектор.

Таким образом, структура вектора состоит из двух частей: дескриптора и непосредственно вектора, в котором последовательно размещаются элементы структуры данных.

**дескриптор вектора**

|  |
| --- |
|  |
|  |
| Адрес начала вектора |
|  |
| Длина элемента |
|  |
| Индекс начального эл-та |
|  |
| Индекс конечного эл-та |

Дескриптор содержит: адрес начала вектора, тип элементов структуры данных или их длину, число элементов или их начальный и конечный индексы. Адрес элемента с индексом*i* определяется выражением *Аi= Am+(i - m) \*L*,

В большинстве случаев отображение структур данных в структуры хранения в виде вектора можно выполнить, используя такой тип данных, как «массив». При этом транслятор создает как дескриптор вектора, так и сам вектор, а обращение к элементам вектора сводится к обращению к элементам массива. Однако транслятор создает статический массив с фиксированными размерами, что может явиться ограничивающим фактором его применения. В прикладной программе можно организовать хранение данных в динамическом векторе. Для этого нужно выполнить ряд операций: определить структуру дескриптора, получить динамическую память под дескриптор и заполнить дескриптор управляющей информацией. Используя данные из дескриптора, определить размер памяти для вектора и получить динамическую память. При необходимости инициализировать лементы структуры данных в векторе. Адреса элементов для доступа к ним вычисляются с использованием управляющей информации из дескриптора. В программе доступ к структуре данных, хранимой в виде вектора, осуществляется посредством указателя на дескриптор.

Указатель на дескриптор —» Дескриптор —>Вектор

Иногда вместо дескриптора используется только указатель на начало вектора. Тогда остальные параметры структуры данных (тип или длина элементов, число элементов и т.п.) должны быть известны каждой из процедур обработки структуры данных, что может усложнить их реализацию.

Приведем пример программы организации хранения структуры данных в векторной памяти и обработки ее элементов.

Элементами структуры данных являются вещественные числа, обработка элемента сводится сначала к занесению в элемент некоторого значения, а затем к выводу его значения на печать. Доступ к элементам структуры данных осуществляется по их индексам, значения которых меняются от*m*до*п*.

Пример 1.1.BR  
/\*Использование векторной памяти. \*/  
#include <stdio.h>  
#include <alloc.h>  
#include <string.h>  
#define DESC struct D\_s       /\* Дескриптор вектора \*/  
DESC  
{float \*ptr\_v;        /\* Указатель начала вектора\*/  
int len;         /\* Длина элемента структуры \*/  
int m;          /\* индекс начального элемента \*/  
int n;         /\* Индекс конечного элемента \*/  
}  
int main()  
{DESC \*d1;         /\*Указатель на дескриптор \*/  
float **ptr\_i**;        /\* Указатель на элемент\*/  
float a;         /\* для выборки эл-та \*/  
int i;  
/\* Получение памяти под дескриптор \*/  
d1 = (DESC\*) calloc(1,sizeof(DESC));  
if (d1==NULL)       /\*память под дескриптор не выделена \*/  
return -1;      /\* неудачное завершение программы \*/  
/\* Занесение в дескриптор управляющей информации \*/  
d1->len = sizeof(float);     /\* Длина элемента \*/  
d1-> m= -3;       /• Индекс начального элемента \*/  
d1->n = 5;        /\* Индекс конечного элемента 7  
/\* Получение памяти под вектор размером (n-m+1)\*len \*/  
d1->ptr\_v = malloc((d1->n — d1->m +1)\*d1->len);  
if (d1->ptr\_v == NULL)         /\* память под вектор не выделена \*/  
{free (d1);          /\* освобождение памяти из-под дескриптора \*/  
return -1;            /\* неудачное завершение программы \*/  
}  
/\* Инициализация элементов структуры \*/  
for (i=d1->m; i≤d1->n; i++)  
{ /\* Адрес i-го элемента структуры \*/  
ptr\_i = d1->ptr\_v + i - d1->m;  
/\* Занесем в i-й элемент число \*/  
\*ptr\_i = 12.34/(i+1);  
}  
/\* Выборка и обработка элементов структуры \*/  
for (i = d1->m; i ≤ d1->n; i++)  
{ ptr\_i = d1->ptr\_v + i -d1->m;      /\* адрес элемента \*/  
a = \*ptr\_i;           /\* Выборка элемента \*/  
printf(«\n %d -й элемент = %f»,i,a); /\* Печать \*/  
}  
getchar();          /\* Ждем нажатия ввода \*/  
}

**1.4 Список**

**Список**является более сложной структурой хранения, в которой элементы имеют явные связи. В списке элементы структур данных размещаются не обязательно в смежных областях памяти. Каждый элемент структуры данных содержит по крайней мере два поля. Одно поле — основное содержит данные этого элемента или указатель на данные, а другое поле предназначено для размещения указателя на следующий элемент. Оконечный элемент структуры данных в поле указателя содержит признак конца списка (NULL). Элемент структуры данных, в свою очередь, тоже может быть структурой, хранимой в виде списка. Различают однонаправленные списки, двунаправленные списки, циклические списки. Число элементов списка может изменяться. Учитывая наличие явных связей между элементами списка, иногда используют термины *цепной список* или ***связный список***. Список, отражающий отношение соседства между элементами (у каждого элемента списка могут быть только один предыдущий и один последующий элементы), называется***линейным***. Любой другой список называется ***нелинейным*.** Однонаправленный список является простейшей формой списка.

   **Массив**



**Достоинства однонаправленного списка** — простота создания, включения и удаления элементов. **Недостаток** — просмотр элементов осуществляется всегда с начала списка. Правда, программа обработки может запоминать указатель на текущий элемент, тогда обработку можно продолжать с него.

**Циклический список** отличается от однонаправленного только тем, что последний элемент в поле указателя содержит не признак конца списка, а ссылку на начало списка, т.е. на первый элемент, что позволяет перейти на другую операцию обработки элементов, начиная с любого элемента списка.

В **двунаправленном списке** каждый элемент содержит два указателя — на последующий и предыдущий элементы, что позволяет вести обработку элементов как в прямом, так и в обратном направлении.

Физически список реализуется как совокупность дескриптора списка и элементов, размещенных в оперативной памяти или на внешних запоминающих устройствах (ВЗУ) и связанных друг с другом в цепочку с помощью указателей. Дескриптор обычно реализуется в виде записи и может содержать следующую информацию о списке;

* тип структуры;
* имя списка;
* указатель (адрес) начала списка;
* текущее число элементов;
* описание элемента (длина, тип и т.п.).

Вместо дескриптора можно использовать только указатель.  
Элементы структуры данных в списке могут быть как одинаковых, так и различных типов. Обработка элементов в первом случае проще, однако иногда это приводит к излишним затратам памяти, если данные существенно различаются по длине. Если данные различаются только по длине, то каждый элемент должен содержать информацию о своей длине. При различии по формату элементы снабжаются дескрипторами, определяющими формат и длину. Однако в этом случае выгоды от экономии памяти могут быть потеряны за счет усложнения процедур обработки элементов.

Элементы структуры данных в списке могут быть упорядочены по возрастанию или убыванию значения некоторого поля данных (по ключу). Обработка *упорядоченных*списков во многих случаях проще обработки неупорядоченных списков. Списки позволяют свободное размещение отдельных элементов структуры данных в различных областях динамической памяти, что обеспечивает более гибкое использование памяти ЭВМ. Тогда для включения в список нового элемента запрашивается динамическая память, а при удалении элемента из списка, занятая им память освобождается. Это обеспечивает возможность гибкого использования памяти, но получение и освобождение памяти сопряжены с обращением к средствам управления памятью операционной системы и приводят к дополнительным затратам времени.

Возможна организация списка и в смежной области памяти. В этом случае все элементы структуры данных имеют один и тот же тип, а число возможных элементов ограничено размерами области памяти. На рис. приведен пример цепного списка, размещенного в одной сплошной области памяти (указатель списка размещен в другой области и содержит адрес списка). Элементами данных являются буквы. Последовательность букв, размещенных в списке, образует слово АЗБУКА. В качестве указателей использованы индексы.

Если обработка списка предусматривает операции как включения, так и удаления элементов, то потребуются создание и ведение двух дополнительных списков: списка свободных и списка занятых элементов. Удаляемый элемент при этом включается в список свободных, а для включения элемента используется элемент из списка свободных, если таковые имеются. В качестве указателей в таких списках могут использоваться либо индексы элементов, либо их адреса.

**1.5 Сеть**

**Сеть** как структура хранения является дальнейшим развитием списковой структуры хранения. Каждый элемент структуры данных в сети, кроме данных, может содержать несколько указателей на другие элементы. На один и тот же элемент могут ссылаться несколько других элементов, образуя, таким образом, сложные переплетения по связи.

В общем случае количество полей в элементах структуры данных в сети может быть различным, тогда элемент должен содержать информацию о своем формате. Вместо данных в элементе сети, как и в списке, может располагаться ссылка на данные.

Сети используются для представления в памяти ЭВМ сложных структур данных, таких, как графы, [деревья,](https://math.gsu.by/wp-content/uploads/courses/structure/l5.2.html#top3)многоуровневые и многосвязные таблицы, структуры сложных программных изделий, организационные структуры предприятий и связи между ними и т.п.

Каждый элемент структуры данных в сети обычно размещается в динамически получаемой памяти, что обеспечивает гибкость представления и обработки данных. Сети могут быть организованы и в смежной области памяти фиксированной длины, если количество элементов и структура данных, представленная сетью, при решении задачи остаются неизменными.

**1.6. Массивы**

**Массивы** являются наиболее широко используемыми структурами данных и предусмотрены во всех языках программирования. Массив состоит из элементов одного типа, называемого базовым, поэтому структура массива однородна. Базовый тип может быть как скалярным, так и структурированным, т.е. элементами массива могут быть числа, символы, строки, структуры, в том числе и массивы. Число элементов массива фиксировано, поэтому объем занимаемой массивом памяти, остается неизменным. С каждым элементом массива связан один или несколько индексов. Они однозначно определяют место элемента в массиве и обеспечивают прямой доступ к нему. Индексы массива относятся к определенному порядковому типу, поэтому индексы можно вычислять. Это обеспечивает, с одной стороны, гибкость обработки элементов массива, с другой стороны, создает опасность выхода за пределы массива, если не предусмотрены соответствующие средства контроля.

В зависимости от числа индексов различают **одномерные и многомерные массивы**. Допустимое число индексов массива (размерность массива) и диапазоны изменения их значений устанавливаются языком программирования, трансляторы с языка программирования могут уточнить эти значения. По стандарту языка Си размерность массива не может превышать 31, а нумерация индексов начинается всегда с нуля, т.е. индекс изменяется от 0 до n-1, где n — количество значений индекса (мощность индекса).

**ПРИМЕНЕНИЕ МАССИВОВ**

Областями применения массивов являются:

* числовые массивы в вычислительных задачах;
* матричная алгебра, экстраполяция, интерполяци;
* указатель (адрес) начала списка;
* таблицы — массивы с элементами типа «запись»;
* управляющие и информационные таблицы в операционных системах, трансляторах, системах управления базами данных (СУБД);
* представление других структур: графов, деревьев.

Массив в памяти хранится в виде вектора, т. е. все элементы размещаются в смежных участках памяти подряд, начиная с адреса, соответствующего началу массива. Элементы одномерного массива размещаются в последовательности А0, А1, А2, …,Аn-1. Элементы двухмерного массива размещаются либо по строкам, когда наиболее быстро меняется последний индекс ( в Си, Паскале, ПЛ/1), либо по столбцам, когда наиболее быстро меняется первый индекс (в Фортране).

Вся информация, необходимая для управления массивом, задается при его описании в программе. Описание содержит имя массива, тип элементов, который однозначно определяет длину элемента, диапазоны изменения индексов или число значений индексов, если нижние границы индексов фиксированы. Таким образом, общее количество элементов массива и размер памяти для массива полностью определяются описанием массива. Транслятор выделяет необходимую память и строит управляющий блок массива — дескриптор, или информационный вектор, например, такого содержания (для одномерного массива):

* тип структуры;
* адрес начала массива Аn;
* тип элемента (длина элемента массива L);
* нижняя граница индекса in;
* верхняя граница индекса ik.

В случае многомерного массива для каждого индекса задаются нижняя и верхняя границы или же для каждой строки (каждого индекса) создается свой дескриптор.

Этой информации достаточно как для доступа к элементам массива, так и для контроля над тем, чтобы значения индексов не выходили за установленные диапазоны. Доступ к любому i - му элементу в одномерном массиве осуществляется по адресу:  
Ai=An+(i-in)\*L=An-in\*L+i\*L= A0+i\*L,  
где A0=An-in\*L - Фиктивное начало массива, т. е. адрес нулевого эл-та.

То, что размеры массива, формируемого транслятором, фиксированы, может явиться ограничивающим фактором применения готовой программы. Действительно, требуется, чтобы память для массива выделялась в размерах, необходимых для решения конкретной задачи, а каковы будут ее потребности, заранее может быть неизвестно. В таких ситуациях массив можно строить в динамической памяти, получаемой с помощью средств управления памятью операционной системы. Управление, доступом к элементам таких массивов осуществляется самой программой по вычисляемым индексам (адресам) элементов с использованием указателя.

Пусть n-мерный массив А, у которого число элементов по I-M измерениям равно mi, а индексация по всем измерениям начинается с нуля, представлен в динамической памяти в виде вектора В. Тогда элементу А[i1,i2, …,in] исходного массива будет соответствовать элемент вектора В[к] с индексом, равным  
k=((…(i1\*m2+i2)\*m3+i3) \*m4+…+in-1)\*mn+in.

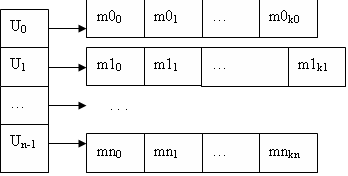
Существуют различные варианты использования динамической памяти, некоторые из них будут рассмотрены позже.

**Свободные массивы**

***Свободными*** называют двухмерные массивы, размер каждой из строк которых может быть различным. Поскольку стандарты языка программирования не допускают такого вида массивов, то создается симбиоз массивов: одномерный массив указателей, число элементов n которого равно числу строк переменной длины, и n одномерных массивов различной длины. Таким образом, массив указателей имеет фиксированную длину, значит, фиксировано и число строк свободного массива. Память под каждую строку свободного массива запрашивается динамически.

Следовательно, при создании такого массива требуется (n+1) обращений к ОС для получения динамической памяти. После того как надобность в свободном массиве отпала, необходимо освободить занимаемую им память, что опять-таки потребует (n+1) обращений к ОС.

На риc. изображена **структура хранения свободного массива с п строками**. Массив указателей, U имеет п элементов, каждый из которых содержит адреса векторов-строк массива.



Для управления свободным массивом может создаваться дескриптор, в простейшем случае — это указатель на массив указателей. Свободные массивы используются для представления других, более сложных структур. С примерами таких массивов мы встретимся ниже. По аналогичной схеме могут быть созданы и двухмерные массивы с одинаковыми длинами строк.

Такое представление массивов имеет свои преимущества и недостатки. К **недостаткам** можно отнести потребность в дополнительной памяти под массив указателей и многократное обращение к ОС для получения памяти. К **преимуществам** можно отнести следующее:

1)при работе с очень большими массивами получение больших сплошных областей может стать невозможным; в то же время получение памяти под каждую строку вполне возможно;  
2)появляется возможность обрабатывать большие массивы данных, хранящихся в файлах, размещая в оперативной памяти только те строки, которые совместно обрабатываются в данный момент;  
3)применение свободных массивов, имеющих строки различной длины, позволяет более эффективно использовать память;  
4)при сортировке массива по строкам (не элементов в строке, а самих строк) вместо перестановок строк в некоторых случаях можно переставлять только элементы массива указателей, в то время как сами строки остаются на месте.

Пример использование свободного массива мы рассмотрим на практических занятиях.

**Треугольные и разреженные матрицы**

**Треугольные и разреженные матрицы**. Иногда при использовании матриц имеет смысл хранить только какую-либо часть матрицы. К таким матрицам можно отнести треугольные и разреженные матрицы.

В **треугольной матрице** Аn\*n все элементы выше или ниже главной диагонали являются нулевыми. Если нулевыми являются элементы выше диагонали (нижняя треугольная матрица), то ненулевыми будут элементы a11; a21, a22;a31,a32n;a33;…;an1, an2,…,ann. Такую матрицу целесообразно хранить в виде одномерного массива (вектора) В с числом элементов m=n\*(n + 1)/2. Тогда индекс ib в массиве В для исходного элемента аij , i = 1,2,...,n, j = 1,2,...,i, определяется по формуле

…

Если же индексация массивов начинается с нуля, i = 0, 1,...,я-1> j = 0, 1 ,...,i, то индексы нижнего треугольного массива вычисляются по формуле



Если в исходной матрице нулевыми являются элементы, расположенные ниже главной диагонали (верхняя треугольная матрица), то в массиве В ненулевые элементы разместятся в последовательности a11,a121,…,a1n; a22,a23,…,a2n;…;ann, а доступ к элементу aij в этом cлучае будет осуществляться по индексу ib, вычисляемому по более сложной формуле



При индексации, начинающейся с нуля, формула преобразуется



**Разреженными** называются матрицы, содержащие большое количество нулевых элементов. Они обычно встречаются в научных приложениях. Представление таких матриц в виде двухмерных массивов приводит к нерациональному использованию памяти и неэффективности выполняемых над ними операций. Во многих работах рассматриваются различные способы представления разреженных матриц. Все они сводятся к сохранению только ненулевых элементов матрицы и их индексов.

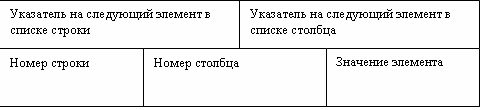
Первый и наиболее простой способ заключается в следующем. Создается двухмерный массив (n\*3), где n — число ненулевых элементов исходной матрицы. Первые два элемента каждой строки содержат индексы строки и столбца ненулевого элемента, а третий — сам ненулевой элемент. Если же значения элементов исходной матрицы нецелого типа, то вместо одного двухмерного массива (n\*3) создаются три одномерных массива с n элементами каждый. Обработка таких массивов особых затруднений не вызывает.

Имеется возможность несколько сократить объем требуемой памяти. Создаются три массива: массив STR по числу строк исходной матрицы; массивы STL и ZN по числу ненулевых элементов исходной разреженной матрицы. В массив STL последовательно заносятся индексы столбцов ненулевых элементов исходной матрицы, сначала для первой строки, затем для второй и т.д., т. е. индексы столбцов каждой i-и строки образуют i-ю группу. Каждый i-й элемент массива STR содержит индекс начала i-й группы в массиве STL. В массив ZN заносятся значения соответствующих ненулевых элементов.

Алгоритмы обработки разреженной матрицы при таком представлении несколько усложняются, так как число ненулевых элементов в строках исходной матрицы различно, а массив STL не содержит признаков конца групп. Поэтому для перехода с одной строки матрицы на другую нужно использовать информацию из массива STR.

Рассмотренные способы представления разреженных матриц массивами удобны только тогда, когда число и расположение ненулевых элементов остаются неизменными. В противном случае более подходящим является способ представления разреженной матрицы с использованием многосвязных циклических списков с динамическим получением памяти для элементов списка.

**Элемент структуры**



Для каждой строки и каждого столбца строятся циклические списки. Ненулевой элемент аij исходной матрицы попадает в два списка — i-й циклический список для i-й строки и j-й циклический список для j-го столбца. Поэтому каждый элемент структуры должен содержать два указателя — указатель на следующий элемент в списке строки и указатель на следующий элемент в списке столбца. Каждый список начинается с головного элемента. Элемент списка, представляющий ненулевой элемент матрицы, использует все пять полей структуры элемента. В головном элементе списка используется только одно поле — указатель по строке в списке строки или указатель по столбцу в списке столбца.

**1.7. Особенности использования массивов в языке Си**

В соответствии с синтаксисом Си в языке допускаются только одномерные массивы. Однако элементами массива также могут быть массивы, следовательно, массив массивов можно рассматривать как двухмерный массив. Аналогично можно использовать и многомерные массивы.

Объявление N-мерного массива имеет вид:  
тип <имя массива> [размер1][размер2]...[размер], где тип — базовый тип элементов массива;  
размер i — число значений i-го индекса массива. Размер задается константой или константным выражением, массивы переменного размера не допускаются. Индекс всегда начинается с нуля. Таким образом, первым элементом массива является элемент с нулевыми индексами. В языке Си в отличие от языков ПЛ/1 и Паскаль нет средств поддержки контроля выхода индексов за установленные пределы.

Для массива транслятор Си всегда выделяет участок непрерывной оперативной памяти, размеры которого определяются числом элементов массива и длиной элемента (т.е. базовым типом). Таким образом, размер выделяемой под массив памяти фиксирован. Так как программы разрабатываются для решения определенного класса задач, то обычно размер массива выбирается по максимуму, в результате при решении конкретной задачи часть памяти может расходоваться впустую. Если же задавать предельно маленький размер массива, то может потребоваться часто редактировать исходную программу, изменяя размеры массивов, и заново транслировать программу.

Выходом из этого положения могут служить динамическое выделение необходимой памяти и последовательное размещение там элементов массива, т.е. хранение массива в виде вектора. В случае одномерного массива доступ к его элементам осуществляется обычным использованием индекса, т.е. индексы массива и вектора совпадают. Доступ к элементам двухмерного массива потребует вычисления индекса вектора по индексам двухмерного массива.

Между массивами и указателями в Си имеется тесная связь, которая существенно используется при разработке программ. При динамическом выделении памяти для массива не обойтись без указателей. Применение указателей во многих случаях позволяет обойти трудности, связанные с фиксированными размерами статических массивов.

При использовании указателей массивов необходимо обратить внимание на следующие моменты:

**ОСОБЕННОСТИ МАССИВА В СИ**  
— Имя массива является указателем константой, поэтому его значение изменить невозможно. Так, если массив определен как int A[5], то А[1] или \*(А+1) указывает на второй элемент массива, но выражение А++ недопустимо.  
— Ответственность за проверку выхода за границы массива полностью лежит на программисте.

В Си все параметры функции, за исключением параметров типа «указатель» и «массив», передаются по значению. Если в качестве параметра функции передается массив, то на самом деле внутрь функции попадает только начальный адрес массива.

Рассмотрим это подробнее. Поскольку при передаче массива в функцию попадает только его начальный адрес, а все элементы как одномерного, так и многомерного массива размещаются в последовательных ячейках памяти (в векторной памяти), то в функции вид обращения к элементам массива полностью определяется прототипом функции. Так, если прототип функции имеет вид:  
fl(int\*a, int\*b,...) или fl(int a[], int b[],...),  
то предполагается, что функция обрабатывает массивы а и b как одномерные. Тогда обращение к элементу массива осуществляется по одному единственному индексу этого элемента. Если исходный массив (фактический параметр) является одномерным, то индексы элементов массива в функции и исходного массива совпадают, т.е. элементу A[i] исходного массива соответствует элемент а[i] в функции (или \*(a+i), что равносильно). Если же исходный массив является многомерным, то значения индексов элементов массива в функции придется вычислять. Так, если исходный массив описан как int A[M][N], ТО его элементу А[i][j] в функции будет соответствовать элемент a[i\*n+j) (или \*(a+i\*n+j)). Рассмотренный подход позволяет использовать функцию для обработки как динамических, так и статических массивов.

Если в функции предполагается обработка многомерного массива с использованием всех его индексов, то размеры всех их, за исключением первого, должны быть указаны в прототипе, например f2(float a[][n][p],...). При этом обращение к элементу массива в функции будет осуществляться по его индексам: a[i][j][k]. Нельзя упускать из виду, что размеры индексов массива задаются константными выражениями.

**1.8. Строки и операции над ними Си**

Строки относятся к фундаментальным типам данных. **Строка** — это последовательность символов, принадлежащих конечному множеству символов (алфавиту). Представление строк в языке программирования определяется стандартами этого языка.

В языке Паскаль предусмотрен строковый тип STRING. Максимальная длина строки равна 255. текущая длина строки S находится в начальном байте строки S[0].

В языке Си строковый тип не предусмотрен, строкой служит массив данных символьного типа char, определяется как char S[n], где n – максимальный размер строки типа unsigned. Заметим, что в прототипах стандартных функций Си используется другое определение этого типа: typedef unsigned size\_t. Конец строки определяется нулевым символом '\0' (байт, все биты которого нулевые), текущая длина строки равна числу символов до символа '\0' исключая его. Массив символов без конечного символа '\0' не можем рассматриваться как строка, это просто массив символьных данных. В языке предусмотрена строковая константа (литерал) — это последовательность символов, заключенная в двойные кавычки, например "Структуры данных". Строковая константа заканчивается нулевым символом '\0'

Для эффективного манипулирования строками современные системы программирования содержат обширные средства и функции.

Рассмотрим алгоритмы выполнения некоторых операций над строками применительно к языку Си и прототипы соответствующих стандартных функций.

Создание и инициализация строки. Строка определяется как массив данных символьного типа, например char S[80]. Транслятор выделяет под массив 80 байт, значит, максимальная длина строки будет 79 символов. В Си не предусмотрен механизм контроля выхода за пределы массива (строки), поэтому ответственность за правильность использования строки несет программист. Если массив определен как автоматический (в пределах функции), то выделенную область памяти транслятор не очищает, статические массивы инициализируются нулевыми байтами, а это означает, что текущая длина строки равна нулю.  
Строка может быть инициализирована при ее определении: char S[40]="ЭTO стро-ка данных"; здесь оставшиеся свободными байты инициализируются нулями. Од-нако нельзя присвоить та¬кой строке новое значение оператором присваивания, например, оператор S="Hовая строка" является ошибочным. Можно определить динамическую строку:  
char\* S1; Sl=(char\*) calloc(40,sizeof(char)):

Только такой строке в любой момент оператором присваивания можно присвоить значение строковой константы: S1="Иванов Иван Иванович".

Определение длины строки S. Осуществляемся последовательным просмотром и подсчётом символов S[i], пока не встретится символ конца строки '\0'. Прототип функции:  
size\_t strlen(const char \*S): результат (возвращаемое значение) - длина строки в байтах.

Заполнение строки заданным символом. Осуществляется только в пределах текущей длины строки, т.е. все символы строки от начала до конечного символа “\0” заменяются заданным символом, при этом длина строки остается прежней. Прототип функции: char\* strset(char \*S, int с), результат — указатель на начале строки S.

Заполнение части строки заданным символом. Отличается тем что заполняется только заданное число n символов, но в пределах текущей длины строки. Прото-тип функции: char\* strnset(char \*S, int c, size\_t n).

Копирование строки S2 в строку S1. Строка S1 заполняетcz символами из строки S2, включая символ конца строки '\0'. Контроль выхода за пределы массива под строку S1 не осуществляется, это возлагается на программиста. Прототип функции: char\* strcpy(char \*S1, char \*S2), результат — указатель на строку S1.

Соединение (конкатенация) строк S1 и S2. К концу строки S1 начиная с символа конца строки '\0' копируется строка S2. Строка S2 сохраняется. Длина строки S1 становится равной сумме длин строк S1 и S2, поэтому память, выделенная под S1, должна быть не меньше этой длины. Прототип функции: char\* strcat(char S1, const char\* S2), возвращаемое значение указатель на строку S1.

Можно предложить другой вариант этой операции, когда соединением двух строк создается новая строка в динамической памяти.

Сравнение двух строк S1 и S2. Осуществляется посимвольным сравнением строк. Так как длины строк L1 и L2 могут отличаться, то число сравниваемых символов ограничивается значением min(L1,L2)+l. В этом случае последними сравниваются символ из более длинной строки и нулевой символ "\0" замыкающий короткую строку. Сравнение строк заканчивается, когда будут исчерпаны все сравниваемые символы либо при неравенстве очередной пары символов. Результат операции определяется результатом последнего сравнения:  
(S1[i] == S2[i]) - равен нулю.  
(S1[i] < S2[i]) - меньше нуля.  
(S1[i] > S2[i]) - больше нуля.  
Прототип функции:  
int strcmp(const char \*S1, const char \*S2). В ранних версиях TCи сравнение проводится в предположении, что символы принадлежат к типу signed char. Это приводит к ошибочным результатам при сравнении строк типа «Иванов» и «Иван», когда начальные символы совпадают, а последний сравниваемый символ более длинной строки является символом из второй половины кодовой таблицы.

Поиск и выделение лексических единиц в строке. Является полезной операцией для выделения отдельных частей строки, разделенных заданными символами- ограничителями. Прототип функции:  
char\* strtok(char \*S1, const char \*S2); в строке S1 осуществляется поиск до первого любого символа-ограничителя, заданного строкой S2. Найденный символ- ограничитель в строке S1 заменяется символом конца строки '\0'. Возвращаемое значение — указатель на начало строки. Таким образом, в виде урезанной начальной строки выделяется первая лексическая единица, а исходная строка оказывается разбитой на две строки по месту первого найденного символа-ограничителя, но этого символа в строке S1 уже нет. Если нужно выделить все лексические единицы в строке S1, то достаточно вызывать функцию strtok с первым аргументом, равным NULL, до тех пор, пока возвращаемое значение не станет равным нулю (NULL). Если в строке S1 встретится несколько подряд идущих символов из строки S2, то первый из них заменился символом '\0' остальные останутся на месте, однако следующий оператор strtok пропустит их и вернет указатель на следующую лексему.

Важнейшими операциями для манипулирования строками являются операции поиска по образцу. Поиск по образцу в строке считается успешным, если в ней найдена подстрока или символ, соответствующие образцу с точки зрения заданного условия или совокупности условий.

Поиск заданного символа в строке. Имеет две разновидности.**Поиск первого вхождения символа** oпределяет позицию, в которой встретился заданный символ при последовательном просмотре символов с начала строки. Поиск последнего вхождения. Отличается тем, что просмотр символов идет от конца строки к ее началу. Прототипы функций:  
char\* strchr(const char \*S, int с) и char\* strrchr(const char \*S, int c), возвращаемое значение указатель на найденный символ или NULL, если символ не найден.

Поиск в основной строке S1 символов, заданных в строке S2. Имеются три разновидности операции. Первая из них — поиск в строке S1 первого вхождения одного из символов, заданных в строке S2. Осуществляется просмотр символов Sl[i], начиная с начала строки и до ее конца. Если очередной символ Sl[i] совпадает с любым из символов строки S2, то поиск прекращается. Возвращаемое значение — указатель на найденный символ или NULL, если в строке не оказалось символов из строки S2. Алгоритм основан на простом переборе: с каждым символом S[i] сравниваются все символы из строки S2, следовательно, временная сложность в худшем случае O(L1\*L2), где L1 и L2 — длины строк S1 и S2 соответственно. Прототип функции:  
char\* strpbrk(const char \*S1, const char \*S2).

Вторая операция отличается от первой только тем, что определяется длина начальной части строки S1, в которой нет символов из строки S2. Прототип функции:  
size\_t strspn(const char \*S1, const char \*S2).

Третья операция отличается от второй тем, что определяется длина начальной части строки S1, в которой содержатся только такие символы, которые принадлежат множеству символов из строки S2. Прототип функции:  
size\_t strcspn(const char \*S1, const char \*S2).

Поиск подстроки (образа) S2 в строке S1. Является наиболее широко используемой операцией. Очевидно, что длина L2 подстроки S2 не может быть больше длины L1 строки S1. Прототип функции:  
char\* strstr(const char \*S1,const char \*S2). Возвращаемое значение указатель на первый символ найденной в S1 подстроки.

Известно несколько алгоритмов поиска подстроки в строке. В алгоритме прямого перебора сначала первые L2 символа строки S1 сравниваются с соответствующими символами образа S2. Если все символы в парах окажутся равными, то на этом поиск заканчивается. В противном случае образ S2 сдвигается в строке S1 на одну позицию вправо и осуществляется сравнение следующих L2 символов и так до тех пор, пока либо все символы не совпадут, либо после (L1-L2) сдвигов не будет достигнут конец строки S1. Результатом является указатель на первый символ в строке S1, с которого начинается искомая подстрока в случае удачи, либо NULL. если подстрока не найдена. Этот алгоритм достаточно эффективен в том случае, когда несовпадение пары символов происходит после всего лишь нескольких сравнений.

Улучшением рассмотренного алгоритма являются **алгоритм Кнута, Мориса и Пратта** (1970 г.) и **алгоритм Боуера и Мура** (1975 г.). Эти алгоритмы основаны на том, что после неудачного поиска на очередном интервале сдвиг образа S2 может осуществляться не на одну, а на несколько позиций вправо в зависимости от некоторых условий.

Алгоритм Кнута, Мориса и Пратта базируется на том, что после частичного совпадения начальной части образа S2 с соответствующими символами строки S1 и при несовпадении очередной пары символов образ S2 сдвигается в строке S1 на пройденное расстояние, если в нем нет начального символа образа S2, либо до позиции с начальным символом. Этот алгоритм дает выигрыш только тогда, когда после очередного сдвига неудаче предшествует некоторое число совпадений символов. Лишь в этом случае образ может быть сдвинут более чем на единицу.

Алгоритм Боуера и Мура отличается от алгоритма Кнута и К\* тем, что сравнение символов начинается с конца образа и идет к его началу. Длина сдвига в случае неудачи, т.е. при несовпадении сравниваемых символов S1j и S2k, может лежать в пределах от 1 до длины образа-строки М. Это зависит от наличия и места расположения символа S1j в образе S2.

**Алгоритм КМП**:

* Основная идея заключается в использовании информации о ранее найденных совпадениях для того, чтобы избежать их повторного поиска. Алгоритм строит так называемую *префикс-функцию* для паттерна (подстроки), которая помогает определить, на сколько символов можно сдвигать паттерн при несоответствии.
* Когда происходит несоответствие, мы не начинаем поиск с начала строки, а сразу сдвигаем паттерн на основании значений префикс-функции, что значительно ускоряет процесс.

**Алгоритм Боуера и Мура**:

* Этот алгоритм использует два основных эвристических принципа для оптимизации поиска: **сдвиг по символам, которые отсутствуют в паттерне**, и **сдвиг с учётом несоответствий на самых правых символах паттерна**.
* Когда происходит несоответствие, алгоритм не просто сдвигает паттерн на один символ, а пытается сдвигать его на несколько позиций сразу, используя информацию о положении символов в паттерне.

Для всех символов, входящих в образ S2, кроме последнего, длина сдвига образа устанавливается так: для предпоследнего символа она равна 1, для следующего от конца - 2, затем --- 3 и т.д., для первого (М-1), где М -— длина образа S2. Если в образе один и тот же символ встречается несколько раз (без учета последнего символа), то для него устанавливается наименьшее значение. Для всех остальных символов алфавита, который используется для представления строк S2 и S1, в том числе и для последнею символа, если он не повторяется в образе, длина сдвига устанавливается равной длине образа S2.

При несовпадении любой пары символов S1j и S2k осуществляется сдвиг на величину, установленную для первого с конца символа сравниваемой части строки S1j. Таким образом, если сравнение началось с символа S1j, а несовпавшим символом является любой сравниваемый символ S1j, то сдвиг будет на одну и ту же величину, установленную для символа S1j.

**1.9. Записи и операции над ними Си**

Запись (**структура**) создается объединением поименованных элементов (полей) произвольных типов. Сами элементы, в свою очередь, могут быть составными. В математике такие составные типы называют декартовым произведением составляющих его типов. Мощность составляющего типа есть произведение мощностей составляющих его типов.

Каждый элемент в записи имеет свое уникальное имя, но такое же имя может использоваться в других записях или для обозначения других объектов программы.

Записи, как и массивы, состоят из фиксированного числа элементов, но между массивами и записями имеются два существенных различия. Во-первых, в отличие от массива, состоящего из однотипных элементов, элементы записи могут быть разных типов. Во-вторых, в то время как доступ к элементам массива осуществляется посредством индексов, доступ к элементам записи — по их именам. Отдельные поля (элементы) записи могут служить в качестве ключей записей.

Записи, как правило, используются в других, более сложных структурах: в таблицах, файлах, базах данных. Отдельная запись используется редко, обычно для извлечения и обработки элементa из более сложной структуры.

Примером записи может служить запись с данными о служащем;

* фамилия;
* имя;
* отчество;
* дата рождения;
* дата поступления на работу;
* специальность;
* семейное положение.

Элементы записи «дата рождения» и «дата поступления на работу», в свою очередь, могут рассматриваться также как запись:

* день;
* месяц:
* год.

Запись хранится в одной сплошной области памяти, причем ее элементы размещаются в памяти последовательно друг за другом в том порядке, в котором они перечислены в записи, например: фамилия, имя, отчество, день, месяц и год рождения, день, месяц и год поступления на работу, специальность, семейное положение.

В языке Си запись получила название структура. Для работы с конкретной структурой в программе вводится структурный тип:  
struct имя\_структурного\_типа  
{ определение элемента 1;  
определение элемента 2;  
…  
определение последнего элемента;  
};

Определим структурный тип о служащем. Поскольку два эле¬мента с датами са-ми являются записями, сначала для них опреде¬ляется структурный тип:  
struct Date  
{ int Den; /\* день \*/  
int Mes; /\* месяц \*/  
int God; /\* год \*/  
};

Этот тип теперь можно использовать для определения струк¬турного типа о слу-жащем:  
struct sluz  
{ char Fam[20];  
char lm[12];  
char Otc[15];  
struct Date Rozd, Post;  
char Spec[15];  
char SemPol;  
}

При определении структурного типа его элементам память не выделяется, поэтому их нельзя инициализировать, т. е. структурный тип не является объектом, он определяет только формат структуры. С использованием описанного структурного типа можно определить конкретные структуры, т. е. ввести в рассмотрение объекты структурного типа, например:  
struct sluz Ivanov, Petrov, Sidorov;

Здесь Ivanov, Petrov, Sidorov являются переменными структурного типа, для каждой из них выделяется необходимая для размещения статическая память. Переменные структурного типа, указатели на них и массивы структур могут быть определены одновременно с определением структурного типа:  
struct sluz  
{ char[20] Fam;  
char SemPol;  
} Ivanov, Petrov, Sidorov, Rab, \*T, Sp[5];

При таком определении имя структурного типа sluz может быть опущено (определяется структурный безымянный тип), однако это делается крайне редко. Как указывалось выше, записи могут являться элементами более сложных структур, тогда имя струк¬турного типа используется при определении отдельного элемента. Например, имя Date применялось при определении элементов Rozd и Post в записи sluz. Имя структурного типа используется также при создании массива структур:  
struct sluz Otd531[36];

Кроме того, ссылка (указатель) на имя структурного типа может использоваться для создания динамического объекта структурного типа:  
struct sluz \*Tmp;  
Tmp=(struct sluz\*) malloc(sizeof(struct sluz));

Переменные структурных типов или указатели на них могут выступать как параметры функций и возвращаемые функциями значения: struct sluz\* Poisk(struct sluz Otd[]):

Функция Роisk осуществляет поиск служащего в массиве, заданном параметром Otd[], и возвращает указатель на запись с найденным служащим или NULL в случае неудачи.

Основными операциями над СТРУКТУРАМИ являются:

* инициализация записи;
* доступ к элементам записи;
* присваивание значения структуре;
* сравнение элементов записей.

Инициализация записи. Похожа на инициализацию массива и осуществляется при ее определении, например:  
Struct sluz Orlov = {"Орлов","Иван","Петрович", {2,8,1980},{ 15,7,1997}, "слесарь", 'х'};

Доступ к элементам записи. Осуществляется с использованием составного имени: имя\_структуры.имя\_элемента, например: Petrov.SemPol=’ж’; или Petrov.Rozd.God = 1981; Доступ возможен также через указатель: Tmp->Post.God = 1999; или (\*Tmp.Post.God) = 1999; Доступ к элементу структуры, являющейся элементом массива структур, осуществляется с использованием индекса элемента массива: Sp[2].Spec= "инженер"; Sp[2].Post.God = 2000;

По стандарту языка Си разрешается присваивание значения структуре:  
Petrov = \*Tmp; \*Tmp=Rab; Rab=Petrov; Sp[l]=Rab;

Сравнение элементов записей. Осуществляется обычным порядком, с учетом типа данных сравниваемых элементов. Сравнение записей в целом в Си не предусмотрено, возможно, только поэлементное сравнение записей.

**1.10.Множества в математике**

С термином*множество* в математике и в языках программирования, при манипулировании со структурами данных связывается разный смысл.

В математике **множество** — это любая совокупность объектов, выбранная из универсального множества. Универсальным при этом считается множество, содержащее сразу все рассматриваемые элементы. Элементами множества в математике могут быть данные различных типов, число элементов не ограничено, одинаковых элементов нет.

Множество определяется как A = {a1,а2,… аn}, аi – элементы множества. Если х элемент множества А, то записывают х∈А. Возможно другое определение множества: А={х| х – день недели}. Если А подмножество В, то пишут А ∊ В, пустое множество обозначают А = Ø или А={}.

Пусть А и В некоторые множества элементов из некоторого класса объектов U, тогда определены следующие операции.  
I.**Дополнение — унарная операция**  
A'= { х| х∈А } содержит все те элементы U, которые не являются элементами множества А.  
2. **Объединение множеств А и В**  
АÙВ={ х| х∈А или х∈В} содержит все элементы U, каждый из которых является либо элементом множества А, либо элементо м множества В, либо одновременно элементом множества А и элементом множества В.  
3. **Пересечение множеств А и В**  
А∩В={ х| х∈А и х∈В} содержит все элементы U, каждый из которых является одновременно элементом множества А и элементом множества В.  
4.**Вычитание множеств А и В**  
А-В={ х| х∈А, но х∉В} содержит все все те элементы U, каждый из которых является элементом множества А, но не является элементом множества В.  
5**. Произведение множест А и В** называется такое множество, каждый элемент которого представляет собой совокупность двух объектов (пару), при этом один из объектов пары является элементом из А, а второй – элементом из В:  
АхВ={ (а,в)| а∈А, но b∈В}. Любое подмножество множества АхВ есть отношение R, при этом множество А называется областью определения, а множество В – областью значений. Отношение R часто имеет смысл =, >, <, и т.д..  
6.**Функция (отношение, преобразование) f**: A∩B или f: A→B есть множество пар элементов (а,в), таких, что а∈А, b∈В и b=f(a).

**1.11. Множества в языках программирования**

В некоторых языках программирования, например в Паскале, предусмотрены структуры типа «множество» (множественные типы). Множество в языке программирования состоит из элементов одного типа (базового типа). В качестве базового типа может выступать любой простой порядковый тип: char, integer или другой, определенный пользователем. Размер множества ограничен некоторым предельно допустимым количеством элементов, например, в Паскале это 256, значения элементов могут изменяться только в пределах от нуля до 255. Стандарт языка определяет операции над множествами, таковыми в Паскале являются: объединение (+), пересечение (\*), разность (-), проверка принадлежности элемента множеству (in). Предусмотрены также операции сравнения множеств =, <>, <=, >=, например, А<=В — операция проверки того, является ли А подмножеством В. В языке Си структура типа «множество» не предусмотрена.

**1.12. Множество как обобщенное понятие структур данных**

С точки зрения структур данных **множество** можно рассматривать как совокупность данных, над которыми выполняется некоторое число операций, образующих функциональную спецификацию структуры этого множества. Пусть определен некоторый тип данных Т. Определим другой тип. элементами которого является множество объектов типа Т. Над данными этого множественного типа допустимы следующие основные операции:

* создать множество – создаётся пустое множество, возвращается его адрес;
* включить элемент - формируемся новое множество добавлением одного элемента к существующему множеству;
* найти элемент - проверить, есть ли элемент в множестве, если есть, определить его адрес;
* удалить элемент — формируется новое множество с удалением одного элемента, если он есть. В противном случае множество остается без изменения;
* пусто — проверить, есть ли элементы во множестве;
* выборка — выдается элемент для обработки, если он есть. В простейшем случае определяется только его адрес, по которому осуществляется обработка;
* выборка с удалением — элемент выбирается для обработки, затем удаляется из множества;
* объединение множеств — над множествами выполняются операции как над математическими множествами, возможны некоторые модификации таких операций.

Многоэлементные структуры, которые мы будем рассматривать, такие, как стеки, очереди, деревья, таблицы, представляют собой частные случаи понятия «множество», а перечисленные выше операции над различными структурами могут иметь другие названия и различные алгоритмы их реализации. Они, эти алгоритмы, в существенной мере зависят от физической структуры представления данных, составляющих множества.

**1. Поиск**

Одно из наиболее часто встречающихся в программировании действий — поиск. Он же представляет собой идеальную задачу, на которой можно испытывать различные структуры данных по мере их появления. Существует несколько основных "вариаций этой темы", и для них создано много различных алгоритмов.

При дальнейшем рассмотрении мы исходим из такого принципиального допущения: группа данных, в которой необходимо отыскать заданный элемент, фиксирована. Будем считать, что множество из N элементов задано, скажем, в виде такого массива:  
int A[n];

   **1.1.Линейный поиск**

Если нет никакой дополнительной информации о разыскиваемых данных, то очевидный подход — простой последовательный просмотр массива с увеличением шаг за шагом той его части, где желаемого элемента не обнаружено. Такой метод называется ***линейным поиском*.** Условия окончания поиска таковы:  
1. Элемент найден, т. е. аi = х;  
2. Весь массив просмотрен и совпадения не обнаружено.

Это дает нам линейный алгоритм:  
i=0;  
while ((i<N)&&(A[i]!=x)) i++;

Обратите внимание, что порядок элементов в логическом выражении имеет существенное значение. Инвариант цикла, т. е. условие, выполняющееся перед каждым увеличением индекса i выглядит так:  
(0≤i<N)&(Ak:0≤k<i : ak!=x).

Он говорит, что для всех значений k, меньших чем i, совпадения не было. Отсюда и из того факта, что поиск заканчивается только в случае ложности условия в заголовке цикла, можно вывести окончательное условие:  
((i=N)OR(ai=x)) & (Ak:0≤k<i : ak!=x).

Это условие не только указывает на желаемый результат, но из него же следует, что если элемент найден, то он найден вместе с минимально возможным индексом, т. е. это первый из таких элементов. Равенство i = N свидетельствует, что совпадения не существует.

Совершенно очевидно, что окончание цикла гарантировано, поскольку на каждом шаге значение i увеличивается, и, следовательно, оно, конечно же, достигнет за конечное число шагов предела N; фактически же, если совпадения не было, это произойдёт после N шагов.

Ясно, что на каждом шаге требуется увеличивать индекс и вычислять логическое выражение. А можно ли эту работу упростить и таким образом убыстрить поиск? Единственная возможность - попытаться упростить само логическое выражение, ведь оно стоит из двух членов. Следовательно, единственный шанс на пути к более простому решению — сформулировать простое условие эквивалентное нашему сложному. Это можно сделать, если мы гарантируем, что совпадение всегда произойдет. Для этого достаточно в конец массива поместить дополнительный элемент со значением х. Назовем такой вспомогательный элемент барьером, ведь он охраняет нас от перехода за пределы массива. Теперь массив нужно описан так: int A[n+1], и алгоритм линейного поиска с барьером выглядит следующим образом:  
a[N] = x; i = 0;  
while (a[i] != х) i++;  
Результирующее условие, выведенное из того же инварианта, что и прежде:  
(ai=x) & (Ak:0≤k<i : ak!=x).

Очевидно, равенство i = N свидетельствует о том, что совпадения (если не счи-тать совпадения с барьером) не было.

**1.2. Поиск делением пополам (двоичный поиск)**

Совершенно очевидно, что других способов убыстрения поиска не существует, если, конечно, нет еще какой-либо информации о данных, среди которых идет поиск. Хорошо известно, что поиск можно сделать значительно более эффективным, если данные будут упорядочены. Поэтому мы приводим алгоритм, основанный на знании того, что массив A упорядочен, т. е. удовлетворяет условию: Ak:1≤k<N:ak-1≤ak.

Основная идея — выбрать случайно некоторый элемент, предположим аm, и сравнить его с аргументом поиска х. Если он равен x, то поиск заканчивается, если он меньше х, то мы заключаем, что все элементы с индексами, меньшими или равными m, можно исключить из дальнейшего поиска; если же он больше х, то исключаются индексы больше и равные m. Это соображение приводит нас к следующему алгоритму (он называется поиском делением пополам). Здесь две индексные переменные L и R отмечают соответственно левый и правый концы секции массива А, где еще может быть обнаружен требуемый элемент.

L=0; R=N-1; ff=0;  
while ((L<=R)&&(ff!=1) {  
/\* m – любое значение между L и R \*/  
if (a[m]=x) ff=1;  
else if (a[m]  
 else R=m-1;  
};  
Инвариант цикла, т. е. условие, выполняющееся перед каждым шагом, таков:  
(L≤R) &(Ak: 0≤k<L: ak<x)  
&(Ak: R<k<N: ak>x),  
из чего выводится результат:  
found OR ((L > R) &(Ak: 0≤k<L: ak<x) &(Ak: R<k<N:ak>x)),  
откуда следует:  
(am= x) OR (Ak: О ≤ k<N: ak!=x).

Выбор m совершенно произволен в том смысле, что корректность алгоритма от него не зависит. Однако на его эффективность влияет выбор m. Ясно, что наша задача — исключить на каждом шаге из дальнейшего поиска, каким бы ни был результат сравнения, как можно больше элементов. Оптимальным решением будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива. В результате максимальное число сравнений равно log N, округленному до ближайшего целого. Таким образом, приведенный алгоритм существенно выигрывает по сравнению с линейным поиском, ведь там ожидаемое число сравнений N/2.

Эффективность можно несколько улучшить, поменяв местами заголовки условных операторов. Проверку на равенство можно выполнять во вторую очередь, так как она удовлетворяется лишь единожды и приводит к окончанию работы. Но более существенно следующее соображение: нельзя ли, как и при линейном поиске, отыскать такое решение, которое опять бы упростило условие окончания. И мы действительно находим такой быстрый алгоритм, как только отказываемся от наивного желания закончить поиск при фиксации совпадения. На первый взгляд это кажется странным, однако, при внимательном рассмотрении обнаруживается, что выигрыш в эффективности на каждом шаге превосходит потери от сравнения с несколькими дополнительными элементами. Напомним, что число шагов в худшем случае — log N. Быстрый алгоритм основан на следующем инварианте: (Ak : 0 < k < =L: аk < х) & (Ak : R ≤ k < N: ak >= х), причем поиск продолжается до тех пор, пока обе секции не "накроют" массив целиком.  
L=0; R=N;  
while (L  
 m=(L+R)/2;  
if (a[m]<x) L=m+1;  
else R=m;  
}; /p>

Условие окончания — L >= R, но достижимо ли оно? Для доказательства этого нам необходимо показать, что при всех обстоятельствах разность R - L на каждом шаге убывает. В начале каждого шага L< R. Для среднего арифметического m справедливо условие L < m < R. Следовательно, разность действительно убывает, ведь либо L увеличивается при присваивании ему значения m + 1, либо R уменьшается при присваивании значения m. При L = R повторение цикла заканчивается. Однако наш инвариант и условие L = R еще не свидетельствуют о совпадении. Конечно, при R = N никаких совпадений нет. В других же случаях мы должны учитывать, что элемент a[R] в сравнениях никогда не участвует. Следовательно, необходима дополнительная проверка на равенство a[R] = х. В отличие от первого нашего решения, приведенный алгоритм, как и в случае линейного поиска, находит совпадающий элемент с наименьшим индексом.

**1.3. Поиск в таблице**

Поиск в массиве иногда называют поиском в таблице, особенно если ключ сам является составным объектом, таким как массив чисел или символов. Часто встречается именно последний случай, когда массивы символов называют *строками* или *словами*.

Строковый тип определяется следующим образом:  
char A[M];  
Соответственно определяется и отношение порядка для строк:  
(x=y)=(Aj: 0≤j<M: xj=yj),  
(x<y)=Ei: 0≤i<N: ((Aj: 0≤y<i: xj=yj)&(xi<yi)).

Для того чтобы установить факт совпадения, мы должны, очевидно, убедиться, что все символы сравниваемых строк соответственно равны один другому. Поэтому сравнение составных операндов сводится к поиску их несовпадающих частей, т. е. к поиску "на неравенство". Если неравных частей не существует, можно говорить о равенстве. Предположим, что размер слов достаточно мал, скажем меньше 30. В этом случае мы будем использовать линейный поиск и поступать таким образом.

Для большинства практических приложений желательно исходить из того, что строки имеют переменный размер. Это предполагает, что размер указывается в каждой отдельной строке. Если исходить из ранее описанного типа, то размер не должен превосходить максимального размера М. Такая схема достаточно гибка и подходит для многих случаев, в то же время она позволяет избежать сложностей динамического распределения памяти. Чаще всего используются два представления размера строк:

1. Размер неявно указывается путем добавления концевого символа, и больше этот символ нигде не употребляется. Обычно для этой цели используется "непечатаемый" символ со значением «0». (И для дальнейшего это очень важно, что это минимальный символ из всего множества символов.)  
2. Размер явно хранится в качестве первого элемента массива, т. е. строка s имеет следующий вид: *s = S0, S1, S2,…, Sn-1*. Здесь *S1, S2,…,Sn-1*— фактические символы строки, a *s0*– количество символов. Такой прием имеет то преимущество, что размер явно доступен, недостаток же в том, что этот размер ограничен размером множества символов (256).

С учётом первой схемы алгоритм сравнение строк выполняется так:

i=0;  
while ((x[i]=y[i]) && (x[i]!=0x00) i++;

Концевой символ работает как барьер,а инвариант цикла таков:  
Aj: 0≤j<i: xj=yj!=0x00

Результирующее же условие выглядит следующим образом:  
((xi!=yi)OR(xi=0x00))&(Aj: 0≤j <i$ xj=yj!=0x00).

При условии xi=yi считается, что х и у совпадают, если же xi<yi то x<y.

Теперь мы готовы вернуться к задаче поиска в таблице. Он требует "вложенных" поисков, а именно: поиска по строчкам таблицы, а для каждой строчки последовательных сравнений — между компонентами. Например, пусть таблица Т и аргумент поиска х определяются таким образом:  
char T[N-1], x;

Допустим, N достаточно велико, а таблица упорядочена в алфавитном порядке. Воспользуемся поиском делением пополам. Используя соответствующие алгоритмы и сравнения строк , о которых шла речь выше, получаем такой фрагмент программы:  
L=0; R=N;  
while (L<N) {  
m=(L+R)/2;  
i=0;  
while ((T[m,i]=x[i])&&(x[i]!=0x00) i++;  
if (T[m,i]<x[i]) L=m+1;  
else R=m;  
}  
if (R  
while ((T[R,i]=x[i])&&(x[i]!=0x00) i++;  
/\* (R<N) & (T[R,i]=x[i]) фиксирует совпадение \*/

**1.4. Прямой поиск строки**

Часто приходится сталкиваться со специфическим поиском, так называемым поиском строки. Его можно определить следующим образом. Пусть задан массив s из N элементов и массив р из М элементов, причем 0 < М ≤ N. Описаны они так:  
char s[N-1], p[M-1];

Поиск строки обнаруживает первое вхождение р в s. Обычно s можно считать некоторым текстом, а р - образом или словом, и мы хотим найти первое вхождение этого слова в указанном тексте. Это действие типично для любых систем обработки текстов, отсюда и очевидная заинтересованность в поиске эффективного алгоритма для этой задачи. Однако прежде, чем обращать внимание на эффективность, давайте сначала разберём некий "прямолинейный" алгоритм поиска. Мы его будем называть *прямым поиском строки*.

Еще до определения алгоритма нужно более точно сформулировать, что же мы от него желаем получить. Будем считать результатом индекс i, указывающий на первое с начала строки совпадение с образом. С этой целью введем предикат Р(i,j):  
Р(i,j)=Ak: 0≤k<j: si+k=pk.

Наш результирующий индекс, очевидно, должен удовлетворять этому предикату P(i, М). Но этого недостаточно; Поскольку поиск должен обнаружить первое вхождение образа, P(k, M) но быть ложным для всех k < i. Обозначим это условие через Q(i):  
Q(i)= Ak: 0≤k<i: ~P(k,M).

Поставленная задача сразу же наводит на мысль оформить поиск как повторяющееся сравнение. Вычисление Р, естественно, вновь сводится к повторяющимся сравнениям отдельных символов. Если применить к Р теорему Моргана, то получается, что итерации должны "искать" несовпадение между образом и строкой символов.  
P(i,j)=( Ak: 0≤k<j: si+k=pk)= (~Ek:0≤k<j: si+k!=pk).

В результате этого уточнения мы приходим к повторению внутри повторения. Предикаты Р и Q включаются в соответствующие места программы как примечания. Они представляют собой инварианты циклов упомянутых итеративных процессов.

*Анализ прямого поиска в строке*. Этот алгоритм работает достаточно эффективно, если мы допускаем, что несовпадение пары символов происходит, по крайней мере, после всего лишь нескольких сравнений во внутреннем цикле. При большой мощности типа это достаточно частый случай. Можно предполагать, что для текстов, составленных из 128 символов, несовпадение будет обнаруживаться после одной или двух проверок. Тем не менее, в худшем случае производительность будет внушать опасение. Возьмем, например, строку, состоящую из N - 1 символов А и единственного В, а образ содержит М - 1 символов А и опять В. Чтобы обнаружить совпадение в конце строки, требуется произвести порядка N\* М сравнений. Однако, есть прием, который существенно улучшает обработку этого неблагоприятного варианта.

**1.5. Поиск в строке. Алгоритм Кнута, Морриса и Пратта**

Приблизительно в 1970 г. Д. Кнут, Дж. Моррис и В. Пратт изобрели алгоритм (КМП-алгоритм), фактически требующий только N сравнений даже в самом плохом случае. Новый алгоритм основывается на том соображении, что, начиная каждый раз сравнение образа с самого начала, мы можем уничтожать ценную информацию. После частичного совпадения начальной части образа с соответствующими символами строки мы фактически знаем пройденную часть строки и можем "вычислить" некоторые сведения (на основе самого образа), с помощью которых потом быстро продвинемся по тексту. Приведенный пример поиска слова *Hooligan*показывает принцип работы такого алгоритма. Символы, подвергшиеся сравнению, здесь подчеркнуты. Обратите внимание: при каждом несовпадении двух символов образ сдвигается на все пройденное расстояние, поскольку меньшие сдвиги не могут привести к полному совпадению.  
Hoola-Hoola girls like Hooligans.  
Hooligan

Hooligan

Hooligan

Hooligan

Hooligan

Hooligan

……

Hooligan

КМП - алгоритм записывается так :  
i=0; j=0;  
while ((j< М) && (i < N)) {  
while ((i ≥ 0) && (s[i] != p[j])) j = D;  
i++; j++;  
};

Однако такая запись умышленно не совсем точная, поскоку в ней есть сдвиг на неопределенную величину D. Вскоре мы к ней вернемся, а теперь сначала отметим, что условия Q(i -j) и P(i –j,j) сохраняются как глобальные инварианты, и к ним можно добавить отношения 0 ≤i < N и 0 ≤ j < М. Это предполагает, что мы должны отказаться от соглашения, что i всегда отмечает в тексте текущее положение первого символа образа. Точнее, выравненное положение образа теперь i -j.

Если алгоритм заканчивает работу по причине j = М, то из составляющей  
P(i - j,j) следует справедливость P(i - М, М), т.е. согласно Р(i,j)=Ak: 0≤k<j: si+k=pk. совпадение начинается с позиции i - М. Если же работа окончена из-за i = N, то, поскольку, j < М, из инварианта Q(i) следует, что совпадения вообще нет.

Теперь мы должны показать, что алгоритм никогда не делает инвариант ложным. Видно, что вначале i =j = 0, и он истинен. Сначала исследуем эффект от двух операторов, увеличивающих i и j на единицу. Они, очевидно, не сдвигают образ вправо и не делают ложным Q(i - j), поскольку разность остается неизменной. Но, может быть, они делают ложным Р(i - j, j) — вторую составляющую инварианта? Обращаем внимание, что в этой точке истинно отрицание выражения в заголовке внутреннего цикла, т. е. либо j < 0, либо si = рj. Последнее увеличивает частичное совпадение и устанавливает P(i - j, j + 1), а первое также постулирует истинность P(i - j, j + 1). Следовательно, увеличение на единицу i и j не может также сделать ложным тот или другой инвариант. Но в алгоритме остается еще только присваивание j = D. Мы просто постулируем, что значение D всегда таково, что замена j на D оставляет инвариант справедливым.

Для того чтобы найти соответствующее выражение для D, мы должны вначале понять смысл этого присваивания. При условии что D < j, присваивание соответствует сдвигу образа вправо на j - D позиций. Естественно, мы хотим, чтобы сдвиг был настолько больше, насколько это возможно, т. е. D должно быть как можно меньше.

Если инвариант P(i – j, j) & Q(i - j) после присваивания j значения D истинен, то перед ним должно быть истинно P(i - D,D) & Q(i - D). Это предусловие и будет поэтому нашим ориентиром при поиске подходящего выражения для D. Основное соображение: благодаря P(i - j, j ) мы знаем, что  
Si-j...Si-1 =P0...pi-1  
(мы только что просматривали первые j символов образа и убедились в их совпадении с текстом). Поэтому условие P( i - D, D) c D≤j, т.е.  
p0…pD-1=si-D…si-1  
превращается в  
p0…pD-1=pi-D…pj-1 (1)  
и предикат ~P(i -k,M) для k=1, ...,j - D (чтобы убедиться в инвариантности Q(i - D)) превращается в  
р0...рk-1!=pj-k…pj-1 для всех k=1,…,J - D. (2)

Важный вывод: очевидно, что значение D определяется одним лишь образом и не зависит от строки текста. Условия 1 и 2 говорят, что для определения D мы должны для каждого j найти наименьшее D, т. е. самую длинную последовательность символов образа, непосредственно предшествующих позиции j, которая совпадает полностью с началом образа. Для каждого j такое D мы будем обозначать через dj. Так как эти значения зависят только от образа, то перед началом фактического поиска можно вычислить вспомогательную таблицу d; эти вычисления сводятся к некоторой предтрансляции образа. Соответствующие усилия будут оправданными, если размер текста значительно превышает размер образа (М<<N). Если нужно искать многие вхождения одного и того же образа, то можно пользоваться одними и теми же d.

**1. Стеки**

Структура, отражающая отношение соседства между своими элементами, называется **линейной.** К линейным структурам можно отнести массивы, очереди, стеки, деки, линейные списки.

Структуры стека. **Стек** — это динамически изменяемый упорядоченный набор элементов. Помещение элементов в стек и выборка их из него производятся с одного и того же конца, который называется вершиной стека, т.е. выборка элементов из стека осуществляется в порядке, обратном их засылке. Это правило формулируется так: «последним пришел, первым ушел» (LIFO: «Last Input — First Output»).

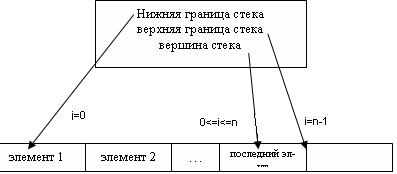
**Области применения стека:**

* передача параметров в функции;
* трансляция (синтаксический и семантический анализы, генерация кодов и т.д.);
* реализация рекурсии в программировании;
* реализация управления динамической памятью и т.п.

Стеки могут представляться в памяти либо в виде вектора, либо в виде цепного списка.

При векторном представлении под стек отводится сплошная область памяти, достаточно большая, чтобы в ней можно было поместить некоторое максимальное число элементов, которое определяется решаемой задачей. Граничные адреса этой области являются параметрами физической структуры стека - вектора. В процессе заполнения стека место последнего элемента (его адрес ) помешается в указатель вершины стека. Если указатель выйдет за верхнюю границу стека, то стек считается переполненным и включение нового элемента становится невозможным. Поэтому для стека надо отводить достаточно большую память, однако если стек в процессе решения задачи заполняется только частично, то память используется неэффективно. Так как под стек отводится фиксированный объем памяти, а количество элементов переменно, то говорят, что стек в векторной памяти - это*полустатическая структура данных*. Обычно в стеке элементы имеют один и гот же тип, поэтому обработка такого стека достаточно проста.

Многие современные ЭВМ содержат в своей конструкций аппаратные стеки или средства работы со стеками. Однако даже в этом случае при разработке программ часто приходится использовать свои программные стеки. Стек, представленный как век гор, имеет вид:



  Структура стека в векторной памяти



    Списковая структура стека

При списковом представлении стека память под дескриптор и под каждый элемент стека получают динамически; включение и выборка элемента осуществляются с начала списка, которое одновременно является вершиной стека. Переполнение стека в этом случае не происходит, однако алгоритмы обработки сложнее, а время обработки удлиняется, так как операции включения и выборки элементов сопряжены с обращением к операционной системе для получения или освобождения памяти.

**2. Операции над стеками**

**Основными операциями над**[**стеками**](https://math.gsu.by/wp-content/uploads/courses/structure/l4.1.html#top4)**являются:**

* создание стека;
* включение элемента в стек;
* выборка элемента из стека;
* извлечение данных;
* уничтожение стека.

**Создание стека**. Для стека в векторной памяти необходимо ностроить дескриптор и отвести память под элементы стека исходя из максимально возможного их количества. Это может быть выполнено либо транслятором, либо путем получения динамической памяти в ходе выполнения программы. В любом случае при инициализации дескриптора в него обязательно заносятся указатели начала и конца стека, а в указателе вершины устанавливается признак «пусто».

Для стека списковой структуры создание стека сводится к построению только дескриптора, в нем достаточно наличия только указателя на начало списка, который одновременно является и указателем вершины списка, так как включение и выборка элемента осуществляются с начала списка. Этот указатель устанавливается в NULL, обозначая, что стек пуст.

**Включение элемента в стек**. В случае стека в векторной памяти, если нет переполнения стека, новый элемент включается в вершину стека, при переполнении отказ oт включения.

В случае стека списковой структуры переполнения, как правило не бывает, по-этому включение сводится к получению динамической памяти под элемент и занесению туда данных. Очевидно, что в указатель в новом элементе заносится значение указателя на начало списка (для первою элемента это NULL), а в указатель начала списка - адрес нового элемента.

**Выборка элемента из стека**. Она возможна только тогда, когда стек не пускт. Она означает, что выбирается значение из элемента, находящегося в вершине стека, а сам элемент исключается из стека. Исключение для векторного стека сводится к смещению указателя вершины на один элемент к началу стека. При выборке из списка списковой структуры указатель на начальный элемент списка заменяется на указатель из начального элемента, а память освобождается из-под элемента списка.

**Извлечение данных** из любого элемента стека без удаления самого элемента. В запросе на выполнение этой операции необходимо указать номер элемента от вершины стека или какой-либо другой уникальный признак элемента; по нему отыскивается нужный элемент, если таковой имеется в стеке.

**Уничтожение стека**. Сводится к освобождению динамической памяти, полученной в процессе создания и обработки стека; если же элементы и дескриптор были построены транслятором, то - к очистке элементов списка и установке признака «пусто» в дескрипторе.

Создадим файл, в котором определены структура дескриптора стека STC и переменная S1 типа STC, а также включены функции, реализующие рассмотренные выше операции над стеками.: Дескриптор построен транслятором, память под элементы стека! получена динамически. Элементы стека имеют значения типа EL, максимальное число элементов m. В дескрипторе определены указатель начала стека в виде адреса начала динамической памяти и указатели вершины и конца стека в виде целых чисел (индексов- элементов стека). Этот файл включается директивой #include в\* исходный файл с программой для работы со стеком. Предварительно должен быть определен тип элемента EL. например define double EL. Допускаются типы EL. только такие, что переменным этого типа можно присваивать значения оператором «=». Таковыми являются скалярные типы (int, float, double, char) и структурный тип struct.

Пример.  
/\* Файл включения для работы со стеком.  
c:\bc\xbs\stack\incl stc.c \*/  
#define STC struct st  
STC              /\* дескриптор стека \*/  
{ EL \*un;           /\* Указатель начала стека \*/  
int uk;          /\* Указатель конца стека \*/  
int uv;            /\* Указатель вершины стека \*/  
int m.            /\* число элементов в стеке \*/  
} S1;             /\* s1 переменная типа STC \*/  
/\* ДОБАВЛЕНИЕ ЭЛЕМЕНТА В СТЕК /  
**int Push\_el(STC s,EL el)  
{if (s->un == NULL)**     /\* стек не создан \*/  
        return -2;  
  if (s->uv == s->uk)  
       return -1;     /\* стек полон \*/  
   \*(s->un + s->uv+1) = el; ++s->uv;  
return 0;  
}

/\* ВЫБОРКА ЭЛЕМЕНТА ИЗ СТЕКА \*/  
int Pop\_el(STC \*s,EL \*el)  
   {if (s->un == NULL)  
       return -2;            /\* стек не создан \*/  
    if (s->uv < 0)  
     return -1;      /\* стек пуст \*/  
   else  
{ \*el = \*(s->un + s->uv);  
    --s->uv;  
      return 0;  
     }  
}  
/\*ИЗВЛЕЧЕНИЕ ЗНАЧЕНИЯ ЭЛЕМЕНТА ИЗ СТЕКА БЕЗ УДАЛЕНИЯ ЭЛЕМЕНТА \*/  
int Peek\_el(STC \*s,EL \*el)  
{if (s->un == NULL)  
      return -2;     /\* стек не создан \*/  
if (s->uv < 0)  
       return -1;     /\* стек пуст \*/  
else  
{ \*el = \*(s->un + s->uv);  
    return 0;  
   }  
}  
/\* ОСВОБОЖДЕНИЕ СТЕКА \*/  
int Destroy(STC \*s)  
{ free(s->un);  
   s->un = NULL;  
    return 0;  
}  
/\* ЗДАНИЕ СТЕКА \*/  
int Crt stc(STC \*s)  
{ int n=10;          /\* число элементов стека можно определить как глобальную переменную \*/  
s->un = (EL\*) calloc(n,sizeof(EL));  
if (s->un == NULL)  
    return -1;               /\* память не выделена \*/  
  else  
       { s->uv = -1; s->uk = n-1;  
         s->m =n; return 0;  
         }  
}  
/\* \*\*\*\* Конец файла включения \*/

Ниже приведен пример программы на Си для работы со стеком в векторной памяти. Элементом стека является переменная типа struct, хотя в структуре содержится единственный элемент — строка. Это вызвано тем ограничением, о котором говорилось выше. Содержанием элемента стека является команда DOS либо имя исполняемого файла. Обработка элемента стека сводится к выполнению этой команды или исполняемого файла.

Пример 2.2.  
/\* РАБОТА СО СТЕКОМ В ВЕКТОРНОЙ ПАМЯТИ  
\stack\dstackg2.c — головной файл \*/  
#include  
#include  
#include  
#include  
#include  
typedef struct          \*Структура элемента стека с именем EL \*/  
{ char Name[80];  
} EL;  
EL e;  
#include"c:\bc\xbs\stack\incl\_stc.c" /\*файл включения \*/  
char \*menu[7][40];  
static int p=1;  
int ln\_el(EL\*);  
int Show\_stc(STC);  
void main\_menu(void);  
/\* ===================== ГЛАВНАЯ ФУНКЦИЯ ====== \*/  
int main()  
{\*menu[0]="1.Создание пустого стека";  
  \*menu[1]="2.Включение элемента в стек";  
   \*menu[2]="3.Выборка элемента из стека";  
   \*menu[3]="4.Освобождение стека";  
   \*menu[4]="5.Вывод содержимого стека на экран";  
   \*menu[5]="6.Конец работы";  
   \*rnenu[6]=" Введите номер строки";  
clrscr();  
printf(" Работа со стеком в векторной памяти\n” );  
while(p)  
{ main\_menu();  
      clrscr();  
}  
printf(" Конец работы со стеком\n");  
return 0;  
}  
/\* ВЫВОД ГЛАВНОГО МЕНЮ \*/  
void main\_menu(void)  
{ char ns; int pp=1,r=0,i;  
flushall();                      /\* чистка буферов \*/  
while (pp)  
     {for(i=0;i<7;i++)  
       printf("\n %s",\*menu[i]);  
           printf("\n");  
           ns=getchar();  
 if (ns<"1" || ns>'6')  
{ clrscr();  
рrintf("\nОшибка в номере!! Будьте внимательны.");  
      continue;  
}  
    else pp=0;  
switch(ns)  
{ case "1":  
  if ( Crt\_stc(&s1) == -1)  
       { printf ("\n Память под стек не выделена");  
          getch();  
}            break;  
case "2":  
    if (ln\_el(&e) == 0)  
      {r=Push\_el(&s1,e);  
            if (r == -2)  
            { printf("\nCтек не создан!!!"); getch();  
             }  
    else  
          if(r==-1)  
{printf("\n Стек полон!!!");  
            getch();  
}  
}            break;  
case "3":  
r=Pop\_el(&s1,&e);  
if(r==-1)  
    printf("\n Стек пуст");  
  else  
    if (r == -2)  
         printf("\n Стек не создан!!");  
       else  
{ printf("\n Элемент выбран\n");  
/\* Обработка элемента \*/  
system(e.Name);  
}  
getch();break;  
case "4": Destroy(&s1);break;  
case "5":  
     if (Show\_stc(s1) == -1)  
{ printf("\n Стек не создан"); getch();  
    }          break;  
case "6": p=0;  
    }  
  }  
}  
/\*==================================================\*/  
int ln\_el(EL \*el)  
{ printf("\n Ввод элемента стека или \*\* для отказа от ввода");  
printf("\n Введите команду DOS или имя исполняемого файла\n=>");  
flushall();  
gets(el->Name);  
    return 0;  
}  
/\*==================================================\*/  
int Show\_stc(STC s)  
{ int i;  
    if (s.un == NULL)  
      return -1;  
for (i=0; i<=s.uv;i++)  
printf("\n %s",s.un[i].Name);  
getch();  
return 0;  
}

**3. Применение стеков при разработке приложений**

Одно из применений [стеков](https://math.gsu.by/wp-content/uploads/courses/structure/l4.1.html#top4) можно продемонстрировать на примере вычисления значения арифметического выражения в калькуляторах. Пусть арифметическое выражение составлено из комбинации чисел, знаков бинарных арифметических операций (операторов) +,-,\*,/,^, круглых скобок (,) и пробелов. Алгоритм вычисления предусматривает представление выражения в определенном формате. Различают представление выражения в инфиксной и постфиксной формах. В инфиксной форме записи каждый бинарный оператор помещается между двумя своими операндами. Для уточнения порядка вычислений могут использоваться круглые скобки. Инфиксный формат записи используется в большинстве языков программирования и калькуляторах и практически совпадает с естественной формой записи выражения. В постфиксной форме записи (обратной польской записи ОПЗ, или Reverse Polish Notation RPN) операнды предшествуют своему оператору. Примеры записи выражений:  
Постфиксное выражение  
5.7 6.8 + =  
15 4\*25 2/3-^ + =  
3 7.5 \* 6е2 5/+=  
Инфиксное выражение  
5.7+6.8=  
15\*4+(25/2-3)^2=  
3\*7.5+6е2/5=

Как видим, выражения представляются в виде символьных строк. Составляющими частями строк (элементами) являются числа, операторы и круглые скобки.

**Постфиксный калькулятор**. Наиболее простым является алгоритм вычисления постфиксного выражения. Исходная строка содержит элементы только двух видов: числа и операторы. Пусть выражение заканчивается символом '='. Алго-ритм использует один стек, элементами которого являются числа вещественного типа. Алгоритм вычисления можно описать следующим образом.  
1. Из исходной строки выделяется очередной элемент.  
2. Если элемент — число, то оно заносится в стек. Переход к п.1.  
3. Если элемент — оператор, то из стека последовательно извлекаются два элемента, сначала правый операнд, затем левый операнд и над ними выполняется операция, определенная оператором. Результат операции (число) заносится в стек. Переход к п.1.  
4. Пункты 1 — 3 выполняются до тех пор, пока в исходной строке не встретится признак конца выражения '='. В этом случае число, находящееся в стеке, является результатом вычисления.

Рассмотрим порядок вычисления выражения  
3 7.5 \* 6е2 5/ + =  
Шаг 1. Из строки выделяется число 3 и помещается в стек. Стек: 3.  
Шаг 2. Из строки выделяется число 7.5 и помещается в стек. Стек: 3 7.5.  
Шаг 3. Из строки выделяется оператор '\*'. Из стека извле¬каются числа 7.5 и 3. Выполняется операция 3\*7.5, результат 22.5 помещается в стек. Стек: 22.5.  
Шаг 4. Из строки выделяется число 6е2 и помещается в стек. Стек: 22.5 600.  
Шаг 5. Из строки выделяется число 5 и помещается в стек. Стек: 22.5 600 5.  
Шаг 6. Из строки выделяется оператор 7'. Из стека извлекаются два числа 5 и 600. Выполняется операция 600/5, и результат 120 помещается в стек. Стек: 22.5 120.  
Шаг 7. Из строки выделяется оператор '+'. Из стека извлекаются два числа 120 и 22.5. Выполняется операция 22.5+120. Ре¬зультат 142.5 засылается в стек.  
Шаг 8. Из строки выделяется символ '=' — признак конца выражения. Из стека извлекается результат вычисления — число 142.5.

**Преобразование выражения из инфиксной формы в постфиксную**. Постфиксная форма представления непривычна для практического применения в калькуляторах. Поэтому рассмотрим алгоритм преобразования выражения из инфиксной формы в постфиксную. Элементами исходной строки-выражения являются числа (операнды), операторы и круглые скобки.

Алгоритм преобразования основан на методе стека с приоритетами. В нем всем операторам и скобкам-разделителям ставятся в соответствие целочисленные приоритеты. Чем старше операция, тем выше ее приоритет. Открывающая скобка имеет низший приоритет, равный 0, закрывающая скобка — равный 1.

В ходе обработки исходной строки операнды переносятся в выходную строку непосредственно, а операторы — через стек в соответствии со своими приоритетами. Элемент стека состоит из двух полей:

* оператор или скобка — символьный тип;
* приоритет — целочисленный тип.

Приоритет пустого стека полагаем равным нулю.

Алгоритм метода состоит в следующем.  
1. Из исходной строки выделяется очередной элемент Si.  
2. Если Si — операнд, то записать его в выходную строку, перейти к п.1; иначе перейти к п.З.  
3. Если приоритет Si равен нулю (т.е. элемент — открывающая скобка) или больше приоритета элемента Sj, находящегося в вершине стека, то добавить Si в вершину стека и перейти к п.4; иначе перейти к п.5.  
4. Если теперь элемент в вершине стека имеет приоритет, равный 1 (т.е. добавленный элемент Si является закрывающей скобкой), то из стека удалить два верхних элемента (закрывающую и открывающую скобки) и затем перейти к п.1; иначе перейти кп.1.  
5. Элемент (оператор) из вершины стека вытолкнуть в выходную строку и перейти к п.З.  
6. Пункты 1 — 5 выполнять до тех пор, пока не встретится признак конца выражеия — символ '='. Тогда все элементы из стека вытолкнуть в выходную строку, затем занести туда символ “=”.

Рассмотрим порядок преобразования на примере строки 15\*4+(25.2-3)^2=.  
Шаг 1. Выделен операнд 15, заносим его в выходную строку. Выходная строка: 15, стек пуст.  
Шаг 2. Выделен оператор '\*', его приоритет больше приоритета пустого стека, помещаем в стек. Стек: \*, выходная строка: 15.  
Шаг 3. Выделен операнд 4, его в выходную строку. Выходная строка: 15 4, стек: \*.  
Шаг 4. Выделен оператор '+', его приоритет меньше приоритета '\*' в вершине стека, поэтому '\*' выталкиваем в выходную строку, а '+' заносим в стек. Выходная строка: 15 4\*, стек: +.  
Шаг 5. Выделена открывающая скобка с нулевым приоритетом, помещаем его в стек. Выходная строка: 15 4\*, стек: + (.  
Шаг 6. Выделен операнд 25, помещаем в выходную строку. Выходная строка: 15 4 \* 25, стек: + (.  
Шаг 7. Выделен оператор “/”, его приоритет выше приоритета '('. помещаем в стек. Строка: 15 4 \* 25, стек: + ( /.  
Шаг 8. Выделен операнд 2, его в строку: 15 4 \* 25 2.  
Шаг 9. Выделен оператор '-', его приоритет меньше приоритета “/” из вершины стека, поэтому ”/” — в строку, теперь приоритет '-' больше приоритета '(' и '-' заносим в стек. Выходная строка 15 4 \* 25 2 /, стек: + ( -.  
Шаг 10. Выделен операнд 3, его в строку: 15 4 + 25 2 / 3.  
Шаг 11. Выделена закрывающая скобка ')', её приоритет меньше приоритета '-' из стека, поэтому '-' — в строку. Теперь приоритет ')' больше приоритета '('> помещаем ')' в стек, но так как его приоритет равен 1, то из стека удаляем два элемента: ')' и '(' без занесения их в выходную строку). Выходная строка: 15 4 \* 25 2/3-, стек: +.  
Шаг 12. Выделен оператор '^', его приоритет больше приоритета '+', '^' засылаем в стек: + ^, строка без изменения.  
Шаг 13. Выделен операнд 2, его — в строку. 15 4\*25 2/3-2.  
Шаг 14. Выделен признак конца выражения '=', из стека выталкиваем в строку '^' и '+', затем в строку заносим '='. Выходная строка сформирована полностью: 15 4\*25 2/3-2А+=, стек пуст.

**Инфиксный калькулятор**. Очевидно, что, используя рассмотренные выше алгоритмы преобразования инфиксного выражения в постфиксное и вычисления постфиксного выражения, легко создать инфиксный калькулятор. Его алгоритм будет основан на применении двух стеков: стека операторов в алгоритме преобразования и стека операндов в алгоритме вычисления. Сам алгоритм будет состоять из двух самостоятельных частей, выполняемых последовательно. Первая часть преобразует инфиксную строку в постфиксную, которая является входом для второй части, выполняющей вычисление постфиксного выражения.

Более подходящим является алгоритм, в котором рассмотренные выше алгоритмы выполняются не последовательно, а совместно. По мере того как часть инфиксной строки преобразуется в постфиксную, осуществляется вычисление преобразованной части выражения. Такой подход облегчает проверку правильности исходного выражения и позволяет прекратить его обработку при выявлении ошибок.

**4. Очереди**

**Очередь** — это динамически изменяемый упорядоченный набор элементов. Добавление элемента в очередь производится с одного конца (хвоста очереди), а выборка — с другого конца (головы очереди) в соответствии с правилом "Первым пришел — первым ушел" (FIFO: First Input — First Output). Такая очередь является простой очередью без приоритетов. Часто используются очереди с приоритетами, в них более приоритетные элементы включаются ближе к голове очереди, выборка осуществляется, как обычно, с головы очереди.

Очереди находят широкое применение в операционных системах (очереди задач, буфера ввода-вывода, буфер ввода с клавиатуры, очереди в сетях ЭВМ и т.п.), при моделировании реальных процессов и т.д. Очереди могут иметь векторную или списковую структуру хранения, в свою очередь векторная структура может занимать статическую либо динамическую память. Очередь векторной структуры из-за ограниченности элементов имеет свойство переполнения, когда хвост очереди достигнет конца вектора. В этом случае добавление элементов становится невозможным, даже если в начальной части вектора будут свободные элементы из-под выбранных элементов. Для устранения такого недостатка образуют кольцевые очереди. При достижении конца вектора новые элементы добавляются в свободные элементы с начала вектора. Здесь так-же возможно переполнение, когда хвост догонит голову. Если же голова догонит хвост, то очередь оказывается пустой.

**Основными операциями над очередью являются;**

* создание и освобождение очереди:
* включение в очередь нового элемента:
* выборка элемента из очереди.

При их выполнении используются такие вспомогательные операции, как проверка наличия элементов, проверка переполнения, организация перехода по кольцу, изменение приоритета: вызванная этим перестройка очереди.

Операции над очередями списковой структуры не вызываю никаких затруднений. Элементы добавляются в начало списка, а выбираются с его конца либо, наоборот, в зависимости от того, какая операция должна выполняться быстрее: включение или выборка элемента.

**Операции создания и освобождения очереди** векторной структуры. Они такие же, как и при работе со стеками. Дескриптор очереди должен содержать адрес начала вектора, указатели головы, хвоста и конца очереди в виде их адресов или индексов.

Рассмотрим возможные состояния и соответствующие действия при выполнении операций включения и выборки элементов.

**Включение в очередь нового элемента**. 1. Очередь заполнена -возврат признака переполнения. 2. Очередь пуста — включение элемента в начало очереди, корректировка указателя хвоста. 3. Очередь заполнена частично - добавление в первый свободный элемент с учетом кольцевой организации, корректировка указателя хвоста.

**Выборка элемента из очереди**. 1. Очередь пуста - возврат соответствующего признака. 2. В очереди единственный элемент — выбрать элемент и установить указатели в начальное состояние. 3. Очередь заполнена частично — выбрать элемент с головы очереди с у том кольцевой организации, скорректировать указатель головы.

Создадим файл, в котором определены структура дескриптора очереди QUE и переменная ql типа QUE: туда же включены функции, реализующие операции над очередями. Дескриптор строится транслятором, память под элементы выделяется динамически по запросу. Этот файл включается директивой #include в исходный файл с программой для работы с очередью. Предварительно должен быть определен тип элемента ЕL, например define int EL. Допускаются типы EL, только такие, что переменным этого типа можно присваивать значения оператором “=". Такими типами являются скалярные типы и структурный тип struct.

Пример разберём на практических занятиях.

**5. Деки**

**Дек** - это разновидность [очереди](https://math.gsu.by/wp-content/uploads/courses/structure/l4.4.html#top5), в которой включение и выборка элементов возможны с обоих концов. Например, может использоваться при управлении памятью, когда распределение памяти производится и сверху, и снизу.

В свою очередь, существуют разновидности дека: дек с ограниченным входом и дек с ограниченный выходом. Дек с ограниченным входом допускает включение элементов только на одном конце, а дек с ограниченным выходом допускает выборку элементов только с одного конца.

Деки могут иметь как векторную, так и списковую структуру хранения. Операции над деками такие же, как и над очередями. При векторном способе хранения программная реализация операций достаточна сложна, она упрощается при представлении очереди в виде двунаправленного списка.

**6. Операции над линейными списками**

Структуры [линейных списков](https://math.gsu.by/wp-content/uploads/courses/structure/l2.4.html#top7) мы уже с вами рассматривали. Теперь остановимся на возможных операциях. Поскольку [списки](https://math.gsu.by/wp-content/uploads/courses/structure/l2.4.html#top6)используются для представления абстрактных данных других типов, рассмотрим те операции, которые могут выполняться над списками независимо от того, какие конкретные данные они представляют. Эти операции можно разбить на две группы: операции над списками в целом и операции над отдельными элементами списка.

**ОПЕРАЦИИ НАД СПИСКАМИ В ЦЕЛОМ:**

* создание списка;
* уничтожение списка;
* печать (вывод) списка;
* копирование списка;
* упорядочение списка;
* объединение списков;
* сохранение и восстановление списка.

**ОПЕРАЦИИ НАД ОТДЕЛЬНЫМИ ЭЛЕМЕНТАМИ СПИСКА**

* поиск элемента в списке;
* включение элемента в список;
* выборка и обработка элементов;
* удаление элемента из списка.

**Поиск элемента в списке**. Поиск элемента является важнейшей операцией, которая используется при выполнении почти всех остальных операций. Поиск осуществляется по значению одного или нескольких полей элемента списка последовательным просмотром элементов. Наше рассмотрение ограничим поиском первого подходящего элемента. Между элементами линейного списка, как мы знаем, существует отношение порядка: для каждого очередного (текущего) элемента, кроме первого и последнего, имеются предыдущий и последующий элементы.

Обозначим указатели на эту тройку элементов через PRED, ТЕК и SLED (смотри рис.), пусть поле указателя в элементе имеет имя Ptr , а поле ключа — key.



Отношение порядка между элементами списка

Переход к последующему элементу осуществляется по указателю LED=TEK.Ptr в текущем элементе. Конец списка определяется нулевым указателем в текущем элементе (TEK.Ptr==NULL).

Возможны удачный и неудачный поиски. Удачный поиск завершается при нахождении искомого элемента, в этом случае возвращается адрес найденного элемента, по которому и осуществляется его обработка. Если элемент не найден при достижении конца списка (неудачный поиск), то возвращается некоторый признак «элемент не найден» (обычно нулевой указатель NULL). Для поиска элемента достаточно одного указателя ТЕК. Алгоритм поиска можно представить в следующем виде:  
 1.  ТЕК = указатель на первый элемент.  
 2.  Если ТЕК == NULL, то конец списка и элемент не найден.  
Переход к п. 5.  
 3.  Если ТЕК.кеу == ключу поиска, то элемент найден, переход к п.5.  
 4.  ТЕК = TEK.Ptr; переход к п. 2.  
 5.  Конец алгоритма, возврат результата.

Поиск в упорядоченном списке имеет свою особенность. Так при поиске в списке, упорядоченном по возрастанию ключей, если ключ поиска меньше ключа текущего элемента, то дальнейший поиск теряет смысл, например, в упорядоченном списке с ключами 5, 7, 9,... осуществляется поиск элемента с ключом 8. Поиск прекращается при достижении элемента с ключом 9 (8ɡ).

**Создание списка.** Создание пустого списка сводится к построению дескриптора с нулевым указателем на начало списка или в простейшем случае к обнулению указателя списка. Построение дескриптора аналогично построению дескриптора вектора, рассмотренному выше, с некоторыми особенностями. Дескриптор может содержать, кроме указателя на начало списка, указатели и на другие элементы, например на текущий, предыдущий, последний. При повторном использовании дескриптора нужно предварительно освободить память, занятую элементами существующего списка, связанного с этим дескриптором.

После создания дескриптора формирование списка сводится к последовательному включению элементов в список.

**Уничтожение списка**. Уничтожение списка сводится к освобождению памяти, занятой элементами и дескриптором списка. Алгоритм уничтожения сводится к следующему:  
 1.  В указатель ТЕК заносим адрес начала списка из дескриптора, а в SLED — адрес следующего элемента: SLED=TEK.Ptr.  
 2.  Если TEK==NULL, то переход на пункт 5, иначе по указателю ТЕК освободим память.  
 3.  Продвинемся по списку на следующий элемент: TEK=SLED; SLED=TEK.Ptr (или SLED=SLED.Ptr, что равносильно).  
 4.  Выполним пункты 2 и 3.  
 5.  Освободим память из-под дескриптора (обнулим указатель на начало списка). Конец алгоритма.

**Печать (вывод) списка**. Печать списка сводится к последовательному просмотру элементов списка до его конца и печати полей каждого элемента в определенном формате. Аналогичным образом производятся и другие виды обработки элементов списка. Для продвижения по списку достаточно одного указателя ТЕК.  
 1.  В ТЕК заносим адрес начала списка.  
 2.  Если TEK==NULL, то переход на 5, иначе печать элемента по указателю ТЕК.  
 3.  Продвижение по списку ТЕК = TEK.Ptr.  
 4.  Выполнение пунктов 2 и 3.  
Конец алгоритма.

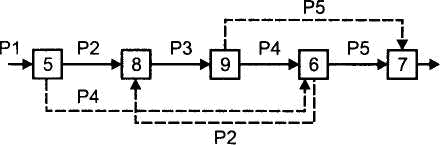
**Копирование списка**. В результате копирования создаются два списка с одинаковыми элементами в различных областях памяти. Алгоритм копирования списка А в список В можно представить в следующем виде.  
 1.  Построить дескриптор нового списка В.  
 2.  Установить указатель текущего элемента ТЕК\_А на очередной элемент списка А, начиная с начала списка.  
 3.  Получить память для элемента В и занести ее адрес в указатель ТЕК\_В.  
 4.  Копировать содержимое текущего элемента списка А в текущий элемент списка В: \*ТЕК\_В = \*ТЕК\_А.  
 5.  Выполнять пункты 2 — 4, пока не будет достигнут конец списка А.

Иногда копирование списка производится для получения; упорядоченного списка из исходного неупорядоченного.

Упорядочение списка. Упорядочение списка (сортировка списка) — это перестановка элементов списка в восходящем или нисходящем порядке в зависимости от конкретных значений одного или нескольких полей внутри элементов списка. Обработка упорядоченных списков облегчается.

Обычно упорядочение списка осуществляется при его создании или копировании. Упорядочение существующего списка «на месте» можно выполнить по методу прямого включения, суть которого заключается в следующем. Начиная со второго элемента, выбираем очередной элемент и включаем его в нужное место с начала списка. Для этого достаточно скорректировать указатели элементов.

Пусть в неупорядоченном списке элементы расположены в следующей последовательности ключей: 5, 8, 9, 6, 7, ...Обозначим указатели на элементы как Р1, Р2, РЗ, Р4, Р5 (рис. ).



Алгоритм включения элемента в нужное место можно описать так. Последовательным просмотром с начала списка и сравнением ключей текущего и следующего элементов по признаку TEK.key > SLED.key устанавливаем, что элемент с ключом 6 должен быть переставлен в списке ближе к его началу. Далее последовательным просмотром с начала списка и сравнением по признаку ТЕК.кеу > 6 находим место вставки — это между элементами с ключами 5 и 8. Процесс перестановки элемента с ключом 6 сводится к обмену указателями между элементами:  
 Р1.Ptr = РЗ.Ptr; (P4);  
 РЗ.Ptr = Р4.Ptr; (P5);  
 Р4.Ptr = PI.Ptr; (Р2);

При обмене указателями нужно следить за тем, чтобы не потерять еще не использованный указатель.

**Объединение списков**. Алгоритмы объединения списков в основном определяются характером решаемой задачи и зависят от многих факторов.  
1. Объединяемые списки могут рассматриваться как множества, тогда возможны схемы логического объединения А∩В, А?В, А-В, и производится соответствующий отбор элементов, включаемых в новый список.  
2. Исходные списки могут быть упорядочены или нет.  
3. Результирующий список создается на новом месте и исходные списки сохраняются либо он создается на месте исходных без дополнительного выделения памяти.

Простое объединение неупорядоченных списков без их сохранения сводится к пересылке указателя второго списка в поле указателя последней записи первого списка, после чего память для Дескриптора второго списка освобождается (или же указатель списка обнуляется).

Процедура объединения упорядоченных списков сложнее. Из второго списка выбирается очередная запись. По ее ключу в первом списке определяется запись, за которой необходимо помесить выбранную запись, и корректируются указатели в первом и втором списках, память не освобождается. Включать или не включать записи с одинаковыми ключами в объединенный список, зависит от характера решаемой задачи. Если запись с дублирующим ключом не включается, то занимаемая ею память во втором списке должна освобождаться. После объединения списков память, занимаемая дескриптором, освобождается.

**Сохранение и восстановление списка**. Реальная работа со списками, при их многократном использовании, вряд ли возможна без их сохранения в файлах и последующего восстановления в оперативной памяти. Очевидно, что в файле нужно сохранять только данные из элементов и нет никакой необходимости сохранять указатели, так как при восстановлении списка память под элементы списка выделяется динамически.

Процедура сохранения списка аналогична процедуре печати списка; отличие лишь в том, что данные выводятся в файл. Процедура восстановления списка подобна процедуре создания, но данные вводятся не с терминала, а считываются из файла.

Возможны два вида хранения данных в файле: в текстовом формате с использованием средств форматного ввода-вывода и в двоичном формате — неформатный ввод-вывод. Хранение элементов списка в текстовом формате предпочтительней, так как позволяет редактировать данные непосредственно в файле. Более того, появляется возможность построения списка по данным из обычного файла с однотипными записями.

**Включение элемента в список**. Алгоритм включения элемента в список существенно зависит от того, как поступать с элементами с одинаковыми (дублирующими) ключами. Возможны три варианта:

* разрешается наличие элементов с дублирующими ключами. Новый элемент включается в список. Если список упорядоченный, то новый элемент включается за элементами с такими же ключами;
* элементы с дублирующими ключами не допускаются, но новый элемент вклю-чается вместо старого (обновление на месте);
* элемент с дублирующим ключом отвергается.

Кроме того, действия по включению элемента зависят от места включения элемента в список.

**Включение элемента в начало списка**. Если список пустой, то:  
 1.  Получение памяти под элемент, адрес элемента помещается в указатель списка.  
 2.  Заполнение полей элемента данными (инициализация элемента).  
 3.  Обнуление указателя в элементе.

Если список не пустой, то:  
  1.  Получение памяти под элемент. В поле указателя нового элемента помещает-ся содержимое указателя списка из дескриптора а адрес элемента помещается в поле указателя списка в дескрипторе.  
  2.  Инициализация нового элемента.

**Включение элемента за определенным (текущим) элементом**. После того как текущий элемент, за которым включается новый, найден:  
 1.  Получаем память под элемент, его адрес помещаем в указатель NEW. В поле указателя его помещаем содержимое поля указателя из текущего элемента (за которым включаем): NEW.Ptr= TEK.Ptr.  
 2.  В поле указателя текущего элемента помещаем адрес нового элемента:  
TEK.Ptr = NEW  
 3.  Инициализируем новый элемент данными.

По этому же алгоритму осуществляется и включение элемента в конец списка.

**Включение элемента перед определенным (текущим) элементом**. При поиске места включения запоминаются два адреса: текущего элемента ТЕК, перед которым включается новый элемент, и предыдущего элемента PRED. После того как нужный элемент найден, новый элемент включается за предыдущим, как было рассмотрено выше.

Возможен другой подход, при котором применяется небольшая хитрость. В ходе поиска запоминаем адрес только текущего элемента. После того как элемент, перед которым нужно включить новый, найден, включим новый элемент не перед ним, а за ним. Затем данные этих элементов поменяем местами.

**Включение элемента в упорядоченный список**. Место включаемого элемента в упорядоченном списке определяется значением его ключа. Если список упорядочен по возрастанию ключей, то осуществляется последовательный поиск элемента, ключ которого больше ключа включаемого элемента. Перед этим элементом включается новый элемент. Если ключ нового элемента больше чем самый большой ключ в списке, то новый элемент включается в конец списка.

**Выборка и обработка элементов**. Различают последовательную и выборочную обработку элементов. При последовательной обработке выбираются все элементы по цепочке указателей и обрабатываются, как правило, по одному и тому же алгоритму.

При выборочной обработке, отбор элемента осуществляется по некоторым признакам, зависящим от значений полей элемента. Обработка элементов, выбранных по различным признакам, может производиться также по различным алгоритмам.

Обработка может заключаться либо только в извлечении и использовании данных, либо в обновлении данных в списке.

**Удаление элемента из списка**. Осуществляется поиск удаляемого элемента. После того как он найден, адрес из поля указателя этого элемента пересылается в поле указателя предыдущего элемента, а память удаляемого элемента освобождается.

**Нелинейные структуры данных**

Нелинейные структуры данных позволяют выражать более сложные отношения между элементами, нежели линейные отношения соседства, как это было в линейных списках. К нелинейным структурам данных относят графы, деревья и леса. Эти структуры находят широкое применение при решении практических задач.

В данной теме мы с Вами достаточно полно рассмотрим способы построения и представления деревьев и основные операции над ними. Вопросы представления графов и алгоритмы на графах рассмотрим позже.

   **1. Рекурсии**

Для представления нелинейных структур и реализации алгоритмов их обработки во многих случаях удобно использование рекурсий.

В программировании рекурсии находят двоякое применение — для определения некоторых объектов и при разработке алгоритмов.

**Рекурсивным** называется объект, частично определяемый помощью самого себя. Мощность такого определения заключается в том, что оно позволяет с помощью конечного высказывания (описания типа данных) определить бесконечное множество объектов, например дерево, список.

Аналогично с помощью конечной процедуры можно описать бесконечное вычисление, причем без наличия явных повторений. **Рекурсивной** называют процедуру, которая прямо или косвенно обращается к самой себе. Если процедура Р содержит явное обращение к самой себе, то ее называют **прямо рекурсивной**. Если она ссылается на другую процедуру Q, которая, в свою очередь явно или косвенно обращается к процедуре Р, то процедура является**косвенно рекурсивной**.

Применение рекурсии позволяет давать более понятные и сжатые описания данных и алгоритмов.

Следуя Вирту, скажем, что рекурсивную процедуру Р можно представить как некоторую композицию Р из множества операторов S, не содержащих Р, и самой Р:  
Р => P[S,P].

Для того чтобы рекурсивная процедура не привела к не заканчивающимся вычислениям, необходимо наличие в процедуре P некоторого условия В обращения к самой себе. Тогда схему рекурсивной процедуры можно представить так:  
Р => if В then P[S,P] или Р => P[S, if В then P].

Использование любой процедуры требует организации связи по управлению и связи по данным между вызывающей процедурой А и вызываемой процедурой В. Связь по управлению обеспечивает передачу управления вызываемой процедуре В и возврат в вызывающую процедуру А после завершения выполнения процедуры В. Вызов процедуры осуществляется посредством оператора вызова по имени вызываемой процедуры.

Связь по данным обеспечивает передачу в вызываемую процедуру фактических параметров и адреса точки возврата в процедуру А, т.е. адреса оператора, следующего за оператором вызова. В качестве фактических параметров могут быть переданы либо значения определенного типа, либо адреса переменных (данных). Адреса переменных передаются тогда, когда в ходе своего выполнения процедура В изменяет значения этих переменных либо когда объем передаваемых данных велик (например, массив данных). Результаты выполнения процедуры В могут быть возвращены в процедуру А несколькими способами:  
 1.  возврат значения определенного типа через имя вызываемой процедуры: тип «имя процедуры» (фактические параметры);  
 2.  возврат по адресам, переданным фактическими параметрами;  
 3.  через глобальные переменные, регистры и т.д.

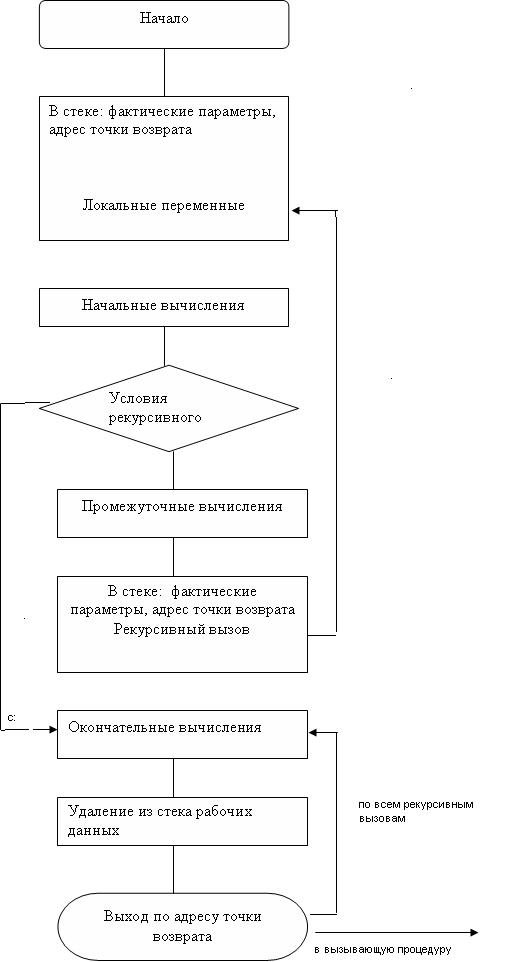
Конкретные механизмы реализации связи по управлению и связи по данным зависят от системы программирования и операционной системы. Эти механизмы скрыты от программиста, ему доступны лишь оператор вызова — <имя(фактические параметры)>, и оператор возврата — . Наиболее широкое применение находит организация связи с использованием стеков. При вызове процедуры В из процедуры А в стек помещаются фактические параметры (значения или адреса) и адрес точки возврата в вызывающую процедуру А. После этого осуществляется передача управления процедуре В. В процедуре В порождаются ее локальные переменные, место для них также отводится в стеке.

Таким образом, к началу выполнения процедуры В в стеке оказываются фактические параметры, адрес точки возврата и локальные переменные, которые сохраняются там до завершения процедуры В. 1

Возврат управления из процедуры В в процедуру А (return значение) осуществляется по адресу точки возврата. Перед этим из стека удаляются данные процедуры В (указатель вершины стека перемещается к концу данных процедуры А). Процедура получает возвращаемое значение, если это предусмотрено.

В рекурсивной процедуре (рис.) каждый рекурсивный вызов, естественно, осуществляется с новыми фактическими параметрами, порождается новое множество локальных перемени а адресом возврата является внутренний адрес в самой процедуре ( точка С: на рис.). Таким образом, если будет выполнено n рекурсивных вызовов, то в стеке образуется (n +1) совокупностей данных, причем еще ни одного возврата из курсивной процедуры не будет. Если условие рекурсивного вызова не будет удовлетворяться, то выполняются окончательные вычисления и происходит возврат, причем n возвратов осуществляются во внутреннюю точку рекурсивной процедуры (точку С:), а последний, (n + 1)-й — в процедуру, откуда 6ыла вызвана рекурсивная процедура.

Итак, выполнение рекурсивной процедуры распадается два периода. Сначала (n + 1) раз выполняется часть процедуры от начала до точки рекурсивного вызова, затем столько же раз от точки возврата (точка С:) до конца процедуры. Важно знать, что вторая часть сначала выполняется с данными последнего рекурсивного вызова, затем — предпоследнего и т.д. Последний, (п + 1)-й раз эта часть выполняется с данными, полученными при начальном вызове рекурсивной процедуры. Возврат осуществляется в вызывающую процедуру, ей же возвращаются и результаты.



С каждым обращением к рекурсивной процедуре ассоциируется номер уровня, начиная с единицы; считается, что вызывающая процедура имеет нулевой уровень. Число рекурсивных обращений к процедуре в процессе вычисления при заданном aргументе образует глубину рекурсии. Поскольку каждая активизация рекурсивной процедуры требует памяти для сохранения данных, то максимальная глубина рекурсии должна быть не только конечна, но и достаточно мала.

Рекурсивные алгоритмы лучше всего использовать тогда, когда сама решаемая задача явно связана с рекурсивными данными или рекурсивными вычислениями. Там же, где есть очевидное итерационное решение, рекурсию следует избегать. В таких функциях единственное обращение к самой себе обычно осуществляется в конце функции.

Пример.

Рассмотрим процедуру вычисления факториала:  
n! = n\*(n - 1)\*(n - 2)\*...\*2\*1. Очевидно, что n! можно представить в виде n!= n\*(n - 1)!, в свою очередь, (n - 1)! можно представить как (n - 1)! = (n-1)\*(n - 2)! и т.д., а 0! = 1. Тогда рекурсией функция вычисления факториала на Си может выглядеть так:  
 double fact(int n)  
{if (n<0) return n; /\* Признак ошибки отрицательное число \*/  
    if (n==0) return 1;  
      else return (n\*fact(n - 1)  
}

Проследим, как же выполняется эта функция при задании конкретного значения n, обеспечив вывод значения фактического параметра n в начале функции и значения n\*fact(n - 1) после курсивного обращения функции к самой себе: double fact(int n)  
  {double f;  
    printf (" %d ",n);  
    if (n<0) return n;  
    if (n==0) return 1;  
       else  
           {f = n\*fact(n- 1);  
              printf(“n=%d f=%lf ",n,f);  
              return f ;  
         }  
}

При вычислении 4!, т.е. при обращении к функции fact(4), будут напечатаны:  
4 3 2 1 0 n = 1 f = 1.000000е+0001 n = 2 f = 2.000000e+0001  
n=3 f = 6.000000е+0001 n = 4 f= 2.400000е+0011

Таким образом, глубина рекурсии равна 5, при каждом обращении к функции с аргументами 4, 3, 2, 1 осуществляется сохранение аргумента, локальной переменной f и адреса точки возврата, после чего происходит рекурсивное обращение к самой себе с аргументом, уменьшенным на единицу. При последнем вызове функции с аргументом 0 значению факториала присваивается 1 и оно возвращается на предыдущий, четвертый уровень. Сохраненное на этом уровне значение n=1 умножается на возвращенное значение 1, полученное значение 1 возвращается на третий уровень с n = 2. Произведение 1\*2 = 2 возвращается на второй уровень, где вычисляется 2\*3 = 6, на первом уровне получается окончательный результат 4\*6 = 24, он и возвращается в основную программу.

Вычисление факториала с использованием не рекурсивной функции гораздо экономнее по времени и по памяти:  
double factn(n)  
{double f=1;  
   if (n<0) return n;     /\* Ошибка в параметре \*/  
   if (n==0) return 1;  
   for (i=1; i<=n; i + +)  
         f = f \* i;  
         return f; }

В рассмотренных функциях возвращаемые значения имеют тип double, использование long int при n᝼ приводит к ошибкам вычислений из-за переполнения.

**2. Общие сведения о деревьях**

Имеется множество подходов к определению дерева как [нелинейной структуры](https://math.gsu.by/wp-content/uploads/courses/structure/l5.1.html#top8).

В частности, **дерево** определяется как ациклический граф G = (V,Е) со специальной вершиной r?V (корень дерева), у которого (графа):  
 1) полустепень захода вершины г равна 0;  
 2) полустепень захода остальных вершин равна 1;  
 3) из вершины r (корня дерева) имеется единственный путь до каждой вершины v?V.

В некоторых источниках дерево определяется как иерархическая структура. **Дерево** представляет собой иерархию элементов называемых узлами (вершинами). На самом верхнем уровне иерархии имеется только один узел — корень дерева. Каждый узел кроме корня, связан только с одним узлом на более высоком уровне, называемым *исходным узлом (предком, отцом)*. Каждый элемент может быть связан посредством ребер (ветвей) с одним или несколькими элементами на следующем, более низком, уровне (с порожденными узлами, дочерними узлами). Элементы, расположенные в конце ветви, т.е. не имеющие порожденных, называются листьями (висящими вершинами). От корня до любого узла существует только один путь. Длина пути (число ветвей) от корня до некоторой вершины называется*уровнем* или *глубиной уровня этой вершины*. Уровень корня — нулевой. Наибольшая длина пути от корня до листьев дерева (максимальный уровень дерева) называется*высотой дерева*.

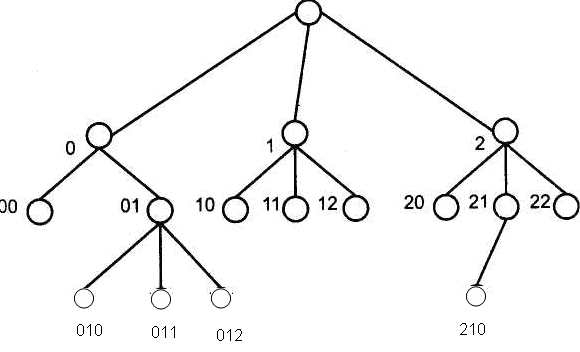
Любой узел дерева с его потомками на всех уровнях также образует дерево, называемое *поддеревом*, дочерние узлы также образуют поддеревья. Следовательно, каждый узел имеет столько поддеревьев, сколько у него дочерних узлов.

Отсюда возникает рекурсивное определение дерева: дерево содержит одну или несколько вершин, причем одну из них называют корнем, а остающиеся объединяют в конечное множество деревьев, называемых поддеревьями. Однако во многих алгоритмах обработки деревьев используется понятие пустого дерева, не имеющего ни одного узла, тогда более подходящим является следующее рекурсивное определение.

Древовидная структура с базовым типом Т— это либо 1) пустая структура, либо 2) узел типа Т, с которым связано конечное число древовидных структур с базовым типом Т, называемых *поддеревьями*. Верхний узел называется *корнем*.

Число поддеревьев данного узла образует степень узла, максимальное значение m степени всех узлов дерева является степенью дерева. Дерево степени m называется m-арным деревом. все узлы дерева, кроме листьев, имеют степень, равную m, то кое дерево называется полным m-арным деревом. Дерево степени 2 называется *бинарным (двоичный) деревом*. Оно также может быть полным или неполным.

Если в дереве на каждом уровне задан порядок следования вершин, то такое дерево называется ***упорядоченным деревом***. Обычно деревья считаются упорядоченными, порядок следования вершин любого узла считается заданным слева направо. Каждая вершина m-арного дерева может быть представлена строкой символов над алфавитом {0,1,..., m- 1}, при этом корень дерева характеризуется пустой строкой. Узлам первого уровня приписываются односимвольные строки 0,1,...,m-1, узлам второго уровня — двухсимвольные строки, узлы i-го уровня характеризуются строками длины i. Любая дочь вершины v характеризуется строкой, префикс которой я вляется строкой узла v, к которой справа приписывается символ 0,1,...или m- 1 в зависимости от ее позиции в нижележащем от v уровне. Строка, приписанная к листу дерева, не является префиксом ни для каких других строк, характеризующих другие вершины дерева, т.е. является уникальной. Множество строк, соответствующих листьям некоторого дерева, образует *префиксный код*этого дерева. На рис. показаны представление тернарного (m = 3) дерева и его префиксный код.



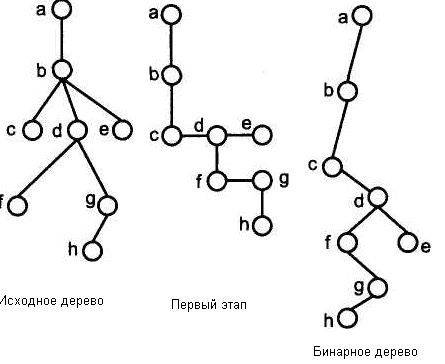
Префиксный код дерева=(00,010,011,012,10,11,12,20,210,22)

**Тернарное дерево и его префиксный код**

Мы видим, что строки, приписанные к листьям, имеют длину. Если листьям дерева поставить в соответствие символ некоторого алфавита, то элементы префиксного кода дерева можно рассматривать как коды соответствующих символов алфавита. Тогда для символов алфавита получаем коды различной длины. Этот подход используется при сжатии данных по алгоритму Хаффмана, когда символы, встречающиеся чаще, имеют коды меньшей длины.

**3. Представление m-apt дерева бинарным деревом**

Любое [m-арное дерево](https://math.gsu.by/wp-content/uploads/courses/structure/l5.2.html#top9) (m Ū 2) может быть представлено **бинарным деревом.** Алгоритм преобразования выполняется в два этапа. На первом этапе для каждого узла самый левый , дочерний узел ставится на один уровень ниже прямо под узлом - отцом, остальные дочерние узлы ставятся на этом же уровне правее первого узла. На втором этапе осуществляется поворот мы на 45 градусов по часовой стрелке, т.е. вертикальные линии становятся левыми ветвями, а горизонтальные — правыми. Рассмотрим на примере схему преобразования тернарного дерева в бинарное.



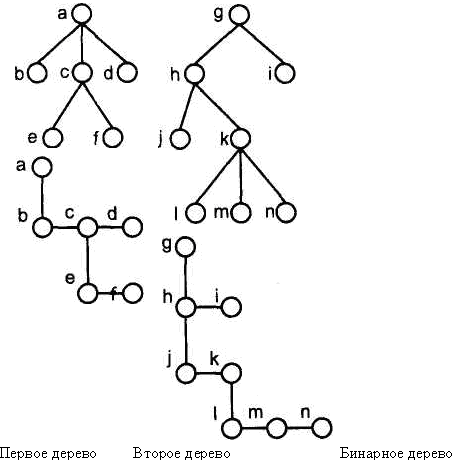
**Преобразование тернарного дерева в бинарное**

**4. Леса**

Множество, состоящее из некоторого числа непересекающихся деревьев, называется **лесом.** Если удалим из дерева корень со всеми его связями с поддеревьями, получим лес, состоящий из поддеревьев корня. Наоборот, добавив к любому лесу с однотипными элементами всего один узел и связав его с корнями деревьев леса, получим дерево, корнем которого станет добавленный узел.

Лес может быть преобразован в бинарное дерево. Алгоритм преобразования леса в бинарное дерево схож с алгоритмом преобразования m-арного дерева. Сначала каждое дерево леса обрабатывается точно так же, как и отдельное дерево на первом этапе. Затем корни деревьев соединяются горизонтальными линиями и осуществляется поворот по часовой стрелке на 45 градусов относительно корня первого (левого) дерева. Таким образом, корнем леса, представленного бинарным деревом, становится корень первого дерева, второе дерево становится правым поддеревом корня первого, третье дерево — правым поддеревом корня второго дерева и т.д.

Приведем схему преобразования леса, состоящего из двух деревьев а и g, в бинарное дерево.



**Преобразование леса графов в бинарное дерево**

Представление m-арного дерева и леса бинарным деревом дает возможность однообразного физического представления деревьев в ЭВМ и облегчает их обработку.

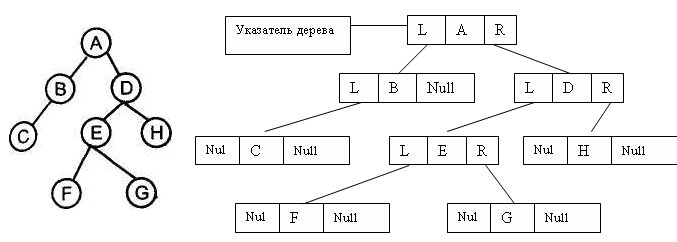
**5. Представление деревьев в памяти ЭВМ**

Элементы древовидной структуры могут быть размещены как в последовательной (векторной) памяти, так и в динамически получаемых областях памяти. Поскольку каждый узел дерева имеет логические связи со своими дочерними узлами, эти связи должны быть отражены и в физической структуре в явном или неявном виде. Пока ограничимся рассмотрением только бинарных деревьев. При явном отражении связей каждый узел дерева имеет структуру вида:  
Данные — Левый указатель LPTR — Правый указатель RPTR.

Здесь поля LPTR и RPTR содержат указатели на левое и правое поддеревья данного узла. Форма указателей может быть различной.

Представление дерева в векторной памяти допустимо только тогда, когда в процессе обработки объем памяти, занимаемой его элементами, не превышает фиксированного объема векторной памяти, т.е. число элементов дерева не может превышать некоторого предельного значения. Элементы дерева занимают последовательные ячейки векторной памяти. При явном задании связей указатели LPTR и RPTR можно задавать двояко: либо в индексов элементов вектора, либо их адресов в памяти. Такое представление дерева мы рассмотрим ниже в разделе о древовидных таблицах поиска.

Неявное задание связей возможно только при работе с бинарными деревьями специального вида, например типа «пирамида», используемого для сортировки данных. Применение такого дерева рассматривается в разделе сортировки пирамидой. Связи между элементами дерева-пирамиды явно не задаются, а вычисляются.



**Схема представления дерева**

Связанное размещение элементов дерева в динамических областях памяти более удобно, обеспечивает возможность легко вставлять в дерево вершины или удалять их из него; кроме того, дерево может разрастаться до произвольного размера. Рассмотрением именно такого представления мы и будем заниматься. Схема связанного представления двоичного дерева приводится на рис. Для управления любым деревом необходимо наличие дескриптора или в простейшем случае указателя на корень дерева (указателя дерева).

**6. Идеально сбалансированное бинарное дерево**

Поскольку максимальный путь до листьев дерева определяется высотой дерева, то при заданном числе n узлов дерева стремятся построить дерево минимальной высоты. Этого можно достичь, если размещать максимально возможное количество узлов на всех уровнях, кроме последнего. В случае двоичного дерева это достигается таким образом, что все поступающие при построении дерева узлы распределяются поровну слева и справа от каждого узла.

Говорят, что бинарное дерево **идеально сбалансировано**, если для каждого его узла количество узлов в левом и правом поддеревьях различается не более чем на 1.

Создание идеально сбалансированного дерева не вызывает затруднений. Если число узлов и известно и дана последовательность значений поля данных вершин а[0], а[1],...,а[п - 1], то можно использовать следующий рекурсивный алгоритм построения идеально сбалансированного дерева:  
1.  Начиная с а[0], берем очередное значение а[i] в качестве значения корня дерева (поддерева).  
2.  Строим левое поддерево с n1 = n/2 узлами тем же cпособом.  
3.  Строим правое поддерево с nr = n - nl - 1 узлами.

Таким образом, значение a[0] окажется в корне дерева, а именно на него будет ссылаться указатель дерева, а[1],...а[nl] значения попадут в левое поддерево, остальные — в правое поддерево. Следовательно, распределение значений по узлам дерева полностью определяется исходной последовательностью данных.

Теперь рассмотрим программу реализации этого алгоритма на Си. Данными в узлах дерева являются целые числа, которые вводятся с клавиатуры в ходе диалога. Для показа результата построения дерева в программу включена функция обхода узлов дерева, которая выводит данные в узлах на экран.

Пример.

/\* ИДЕАЛЬНО СБАЛАНСИРОВАННОЕ БИНАРНОЕ ДЕРЕВО \*/

#include  
#include  
#include  
typedef struct RECORD    /\* Структура элемента дерева \*/  
{ int DATA;          /\* данные узла — здесь только ключ \*/  
RECORD \*LPTR,\*RPTR;    /\* левый и правый указатели на поддерево\*/  
};  
RECORD \*ROOT;    /\* Указатель корня дерева \*/  
static int ndi=0;  
/\*последовательный номер включаемого элемента ,  
удобства работы пользователя и контроля наличия дерева. Перед  
каждым обращением к функциям tree, preorder в вызывающей функции  
переменную ndi нужно обнулять. \*/

RECORD\* tree(int k);         /\* функция создания дерева, к - число узлов\*/  
int preorder(RECORD \*)           ;      /\* функция нисходящего обхода дерева\*/  
void p(RECORD\*);   /\* Печать данных в узле \*/  
void lnData(RECORD\*);   /\* Ввод данных узла\*/  
/\* ===============================================9  
/\* ГЛАВНАЯ ФУНКЦИЯ \*/  
main()  
{int i,n;  
printf("\n\n РАБОТА С ИДЕАЛЬНО СБАЛАНСИРОВАННЫМ БИНАРНЫМ ДЕРЕВОМ");  
printf("\n Создание дерева");  
printf("\n Введите число узлов дерева =>");  
scanf("%d",&n);  
/\* Создание идеально сбалансированного дерева \*/  
ndi=O;  
if( (ROOT=tree(n))==NULL)  
      { printf("\n Дерево не создано");  
         getch();          /\* return -1; \*/  
}  
printf("\n Обход дерева с выводом данных в узлах");  
printf("\n Нисходящий обход дерева:\n");  
ndi=O;    preorder(ROOT); getch();  
printf("\n Конец\n"); getch(); return 0;  
}  
/\* ПОСТРОЕНИЕ БИНАРНОГО ИДЕАЛЬНО СБАЛАНСИРОВАННОГО ДЕРЕ-ВА\*/  
RECORD\* tree(int k)  
{ RECORD \*newnode;  
int nl,nr;  
if (k≤0)                       /\* k<0 - ошибочное задание параметра \*/  
newnode =0;  
   else  
    { nl = k/2;  
       alloc(1,sizeof(RECORD));  
InData(newnode);  
newnode.LPTR=tree(nl);  
newnode.RPTR=tree(nr);

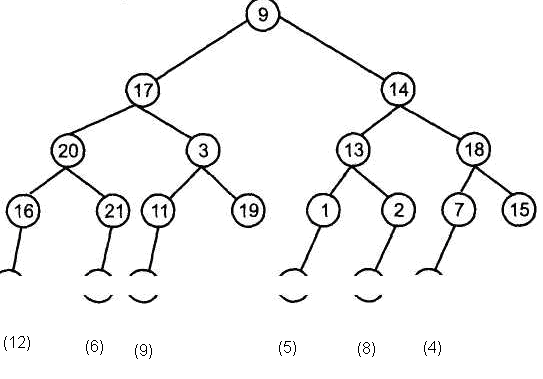
}  
return newnode;  
}

/\* НИСХОДЯЩИЙ ОБХОД ДЕРЕВА \*/  
int preorder(RECORD \*T)  
  { if(T==NULL && ndi==0)  
     { printf("\n Дерево пустое");  
getch(); return -1;  
}  
if (T!=0)  
{ ndi++; p(T);       /\* обработка данных в узле \*/  
preorder(T.LPTR);  
preorder(T.RPTR);  
}  
return 0;  
}

//\* ОБРАБОТКА ДАННЫХ УЗЛА ДЕРЕВА \*/  
void p(RECORD \*B)  
{ printf(" %d ",B.DATA);  
}  
/\* ВВОД ДАННЫХ УЗЛА ДЕРЕВА \*/  
void lnData(RECORD\* d)  
   {         printf(“Введите %d-й ключ =>",++ndi);  
              scanf("%d",&d.DATA);  
}

    Функция tree возвращает указатель на корень построенного дерева, а при рекурсивном обращении — указатель на очередной дочерний узел.

Идеально сбалансированное бинарное дерево, построенное из последовательности ключей: 9,17,20,16,12,21,6,3,11,4,19,14,13,1,5,2,8,18,7,10,15, будет иметь следующий вид:



**Идеально сбалансированное бинарное дерево**

При таком построении, несмотря на то что идеально сбалансированное дерево является упорядоченным в смысле порядка следования вершин на каждом уровне, никакие особые требования относительно значений данных в вершинах не накладываются, т.е. значения данных в вершинах в общем случае не упорядочены. Это ведет к тому, что поиск узла с нужными данными осуществляется последовательным обходом всех n узлов дерева, затраты на поиск будут порядка О(n). Поэтому такие «простые» сбалансированные деревья применяются редко. Обычно используются сбалансированные деревья поиска, которые обеспечивают минимальную длину поиска порядка O(log2n), где n — число узлов дерева.

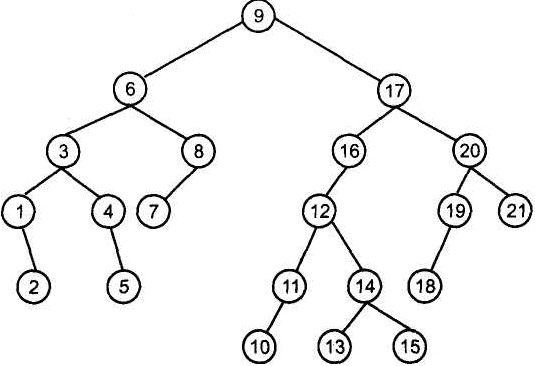
Добавление и удаление узлов дерева вызывают определенные трудности. Чтобы поддерживать сбалансированность дерева при добавлении и удалении узлов, необходимо иметь информацию о сбалансированности каждого поддерева и определенными действиями поддерживать их сбалансированность. В силу этого идеально сбалансированные деревья поиска применяются для работы с данными, состав которых мало изменяется в процессе обработки.

**7. Бинарные (двоичные) деревья поиска**

**Бинарные деревья** чаще всего применяются для представления множеств данных, элементы которых ищутся по уникальному, только им присущему ключу. Если бинарное дерево организовано таким образом, что для каждого узла ti, все ключи в левом поддереве меньше ключа ti, а ключи в правом поддереве больше ключа ti, то это дерево называется ***бинарным деревом поиска*.** В дереве поиска можно найти место каждого ключа, двигаясь начиная от корня и переходя на левое или правое поддерево каждого узла в зависимости от значения ключа. Таким образом, места элементов в дереве определяются как значениями ключей, так и последовательностью их поступления. Определяющим фактором является значение ключа, от последовательности поступления элементов зависит степень сбалансированности дерева. При случайном распределении ключей в исходной последовательности получается почти сбалансированное дерево. Если же исходная последовательность упорядочена по возрастанию или убыванию ключей, то дерево вырождается в последовательный список. Высота такого дерева равна числу элементов дерева, уменьшенному на 1.

Создание дерева поиска заключается в следующем. Первый элемент образует корень дерева. Для последующих элементов осуществляется поиск места включения по ветвям дерева до тех пор, пока не будет найден подходящий узел с нулевым указателем, туда и подключается элемент. Для каждого узла запрашивается динамическая память, ее адрес заносится в указатель узла-предка, данные элемента помещаются в узел, и обнуляются левый и правый указатели нового узла. Дерево поиска, построенное из последовательности ключей  
9,17,20,16,12,21,6,3,11,4,19,14,13,1,5,2,8,18,7,10,15, имеет вид, приведенный на рис.

**Бинарное дерево поиска**



**8. Сбалансированные деревья поиска**

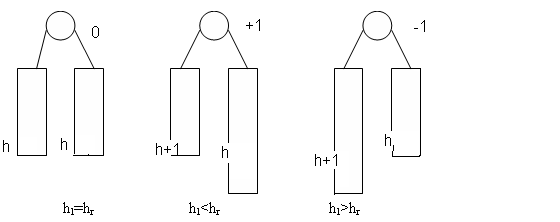
Длина поиска в двоичном дереве поиска определяется высотой самого дерева. Она для заданного числа элементов n зависит от порядка поступления элементов при построении дерева и может колебаться от Iog2n в случае идеальной сбалансированности дерева до (n-1) в случае вырождения дерева в линейный список. Таким образом, затраты на поиск и включение элемента будут порядка от O(log2n) до О(n).

Доказано, что при случайном порядке включения элементов в дерево средние затраты на поиск элемента будут 1.386\*O(log2n). Увеличение затрат на 39% вполне оправдано более простыми средствами поддержания обычного дерева поиска по сравнению с идеально сбалансированным деревом поиска. Однако при работе с большими деревьями свойство случайности распределения поступающих данных редко соблюдается — это скорее исключение, чем правило. Обычно входные последовательности являются частично упорядоченными (по фамилиям, номенклатурным номерам, квалификационным признакам и т.п.). Для таких ситуаций наиболее подходящими являются так называемые ***сбалансированные деревья поиска*.** Рассмотрим два вида таких деревьев: АВЛ- деревья, предложенные в 1962 г. Адельсоном-Вельским и Ландисом, и рандомизированные деревья поиска.  
   **8.1. Сбалансированные АВЛ-деревья поиска**

Требования к сбалансированности в АВЛ - деревьях менее жесткие, чем в идеально сбалансированных деревьях. **АВЛ - деревом** называется такое дерево, у которого высота поддеревьев для каждой вершины различается не более чем на 1. Максимальная высота АВЛ -дерева с n вершинами не превосходит 1,44\*log2(n+ 1) - 1,33, т. е. затраты на поиск не превосходят l,45\*O(log2n).

**Отличие** АВЛ -деревьев от обычных деревьев поиска заключается в том, что при включении и удалении элементов необходимо поддерживать сбалансированность дерева в целом. Для этого в каждый узел дерева добавляется одно вспомогательное поле, содержащее информацию о равновесности поддеревьев (показатель сбалансированности узла). Его значениями могут быть: 0 - высоты правого и левого поддеревьев равны; + 1 — высота правого поддерева больше; - 1 — высота правого поддерева меньше.

 При попытке добавить или удалить элемент в поддереве с показателем сбалансированности, отличным от нуля, дерево может стать несбалансированным, и потребуется операция балансировки.



   **8.2. Рандомизированные деревья поиска**

Создание **рандомизированных деревьев поиска** основа на том, что «случайность» включается в алгоритм вставки элемента в дерево без каких-либо допущений относительно порядка вставки элементов. Идея заключается в том, что при вставке нового узла в дерево, состоящее из N узлов, вероятность появления ново-о узла в корне дерева должна быть 1/(N +1).

**9. Оптимальные деревья поиска**

Существуют достаточно редкие случаи, когда имеется информация о вероятности обращений к отдельным ключам в дереве поиска. Такие вероятности могут быть получены только на основе статистических измерений, когда ключи поиска в дереве остаются неизменными.

Пусть известны вероятности pi обращения к i-м узлам с ключами ki. Мы хотим организовать бинарное дерево поиска таким образом, чтобы минимизировать математическое ожидание числа сравнений при поиске. Для этого припишем каждому узлу в определении длины пути вес, равный вероятности обращения к этому узлу. Тогда взвешенная длина пути дерева есть сумма всех путей от корня к каждому узлу, умноженных на вероятности обращения к этому узлу:



Остается минимизировать эту взвешенную длину пути при данном распределении вероятностей. Очевидно, что узлы с большими вероятностями должны располагаться ближе к корню дерева, но тогда оптимальным может оказаться не сбалансированное, а вырожденное дерево.

Однако такой подход учитывает только удачный поиск, т.е. поиск только существующих в дереве ключей. Если же множество ключей дерева является подмножеством ключей поиска (аргументов поиска), то наличие ключей поиска, приводящих к неудачному поиску, повлияет на структуру оптимального дерева поиска. Такая ситуация присуща, например, трансляторам, когда среди слов исходной программы отыскиваются зарезервированные слова, вероятность появления которых в программе можно установить. В таких случаях нахождение оптимальных деревьев поиска усложняется. (Существует метод динамического программирования для построения оптимального дерева).

**10. Операции над деревьями**

Деревья используются для представления некоторых данных и для манипулирования ими. Конкретное представление данных и выполняемые над ними операции зависят от решаемой прикладной задачи. Мы будем рассматривать общие операции над деревьями, которые могут выполняться при решении любых задач. К этим операциям можно отнести следующие:

**ОПЕРАЦИИ НАД ДЕРЕВЬЯМИ:**

* создание дерева;
* поиск (локализация) элемента в дереве;
* включение элемента в дерево;
* обработка данных в вершинах дерева;
* удаление элемента из дерева;
* сохранение и восстановление бинарного дерева;
* уничтожение бинарного дерева.

Эти операции рассматриваются применительно к двоичным деревьям, размещаемым в динамической памяти.

**Создание дерева**. Для управления деревом как динамической структурой необходимо наличие дескриптора. Если дескриптор построен в динамической памяти, то доступ к дереву осуществляется через цепочку указателей, как показано.



Доступ к дереву посредством дескриптора

В простейшем случае дескриптор не используется, доступ к дереву осуществляется по указателю, объявленному в программе как переменная. Алгоритм включения в дерево отдельных элементов состоит из трех действий: поиск места включения; получение динамической памяти для элемента и образование связи узла с деревом с помощью указателя; занесение данных элемента в узел.

Последние два действия не зависят от вида дерева. Поиск места включения существенно зависит от вида дерева, как это мы видели при создании идеально сбалансированного двоичного дерева. После своего создания одни деревья, как, например, бинарные деревья поиска, легко наращиваются добавлением новых элементов в любой момент времени. Другие, такие, как идеально сбалансированные, для наращивания с сохранением своих свойств требуют значительных затрат.

Поскольку новые элементы можно включать в дерево поиска в любой момент выполнения программы, то создание дерева сводится к управлению вызовом функции включения элемента. Функции включения в обычное и сбалансированное деревья поиска будем рассматривать ниже.

**Поиск (локализация) элемента в дереве**. Поиск элемента может преследовать различные цели. Во-первых, определить, имеется ли искомый элемент в дереве или нет. В этом случае результатом будет возврат признака о наличии или отсутствии элемента. Во-вторых, поиск с целью выборки и обработки данных элемента, тогда результат возвращается в виде адреса вершины. В-третьих, поиск с целью удаления элемента: такой поиск обычно является частью операции удаления, а не самостоятельной операцией.

Алгоритм поиска зависит от вида дерева. В идеально сбалансированном дереве поиск приходится проводить обходом вершин дерева в некоторой последовательности. Минимальная длина поиска равна 1, максимальная — n, а средняя — n/2.

Алгоритм поиска в бинарном дереве поиска, как простом, так и сбалансированном, достаточно прост. Поиск осуществляется целенаправленным движением по ветвям дерева. Если ключ поиска равен ключу в вершине, значит, ключ найден и его адрес возвращается через параметр-указатель. Если ключ поиска меньше ключа в вершине, то осуществляется движение вниз влево, в противном случае — вниз вправо. Если в очередной вершине дальнейшее движение вниз невозможно (указатель равен NULL), то это означает, что искомого ключа нет в дереве. Максимальная длина поиска равна высоте дерева.

**Включение элемента в дерево**. Особенности операций включения зависят от вида дерева.

**Включение элемента в бинарное дерево поиска**. Как указывалось выше, включение элемента состоит из трех действий: поиска места включения; получения динамической памяти для элемента; занесения данных в новый узел.

Сразу же нужно определиться, что делать, если в дереве уже есть элемент с включаемым ключом. В одних случаях такой элемент может просто отвергаться, в других случаях включаться, например, если дерево используется для сортировки по ключа Возможно использование такого элемента также для обновления данных в узле дерева.

Рассмотрим вариант, когда элемент с дублирующим ключе отвергается. Порядок поиска места включения такой же, как и при поиске элемента в дереве. Однако положительным исходом считается отсутствие элемента с заданным ключом, т.е. приход в результате движения к пустому поддереву, обозначенному указателем с нулевым значением NULL. Тогда получаем динамическую память для элемента и его адрес заносим в этот пустой указатель, затем помещаем в новую вершину ключ и данные, а указатели в ней обнуляем.

Если же элемент с дублирующим ключом включается, то в каждой вершине осуществляется сравнение только на «меньше» на «больше или равно». Если ключ включаемого элемента меньше ключа узла, то движение — вниз влево, в противном случае вниз вправо. Применение такого дерева для сортировки приведёт к «устойчивой» сортировке, т.е. элементы с одинаковыми ключами отсортируются в порядке их поступления в дерево.

**Включение элемента в сбалансированное АВЛ -дерево поиска**. Операция включения выполняется в два этапа. Поскольку сбалансированное дерево является частным случаем обычного дерева поиска, то на первом этапе выполняется включение элемента в дерево точно так же, как и при включении в обычное дерево, проходом по пути поиска, получением динамической памяти и формированием в ней новой вершины дерева. Дополнительна новой вершине устанавливается признак её сбалансированности, равный 0, так как новая вершина не имеет поддеревьев.

Второй этап выполняется при обратном движении (из-за рекурсивности функции включения) и заключается в восстановлении сбалансированности в узлах дерева, если она была нарушена. При этом учитывается, откуда (слева или справа) осуществляется возврат в вершину, каков показатель сбалансированности в ней, выросла ли высота поддерева. Возникающие ситуации и особенности восстановления сбалансированности рассмотрим на примерах.

**Пример**. Пусть имеется дерево с вершинами 3 и 2. Высота поддеревьев| вершины 3 hL = 1, hR = 0, признак сбалансированности bal3=-1, у вершины 2 hL =hR = 0, bal2=0. Добавим вершину 1, у неё hL =hR = 0, bal1=0. Возвращаемся в вершину 2, теперь у нее hL = 1, hR= 0. bal2=-1, но критерий сбалансированности поддерева соблюдается. В вершине 3 стало hL = 2, hR = 0, т.е. критерий сбалансированности нарушен, дерево нужно перестраивать. Это достигается так называемым однократным LL-поворотом, в результате получаем сбалансированное дерево с вершиной 2. Необходимые операции балансировки заключаются в обмене указателями ti и T по кругу по часовой стрелке. Кроме этого, необходимо также изменить показатели сбалансированности вершин (см. рис.).



Рис. Балансировка однократным LL-поворотом

Теперь в дерево с вершинами 4 и 5 включим вершину 6. Для балансировки в вершине 4 выполняются однократный RR - поворот, обмен указателями по кругу против часовой стрелки (рис.).

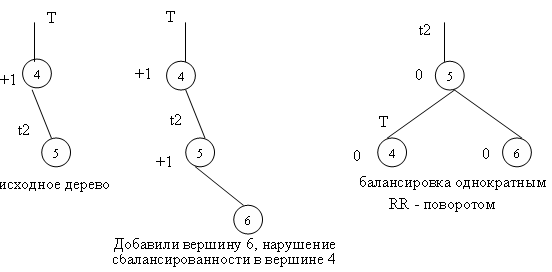


Рис. Балансировка однократным RR-поворотом

Более сложные ситуации приводят к двукратным поворот поворот направо и налево (двукратный .RL-поворот) и noeoj налево и направо (двукратный .LR-поворот). Пример двукраг го RL-noeopoma демонстрируется включением в дерево с Bepi нами 5 и 7 новой вершины 6 (рис. 3.12). Возникла несбалансир ванность в вершине 5. Вначале выполняется правый поворот трёх вершин, затем левый поворот двух вершин (7 и 6). При включении в дерево с вершинами 9 и 3 новой верг ны 8 возникает несбалансированность поддерева с корнем Она устраняется двукратным LR-поворотом: сначала лев! поворот трех вершин, затем правый поворот двух вершин 3'| 8 (рис.3.13).

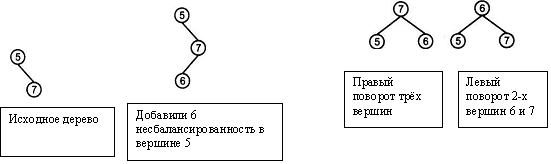


Рис. Балансировка двукратным RL поворотом

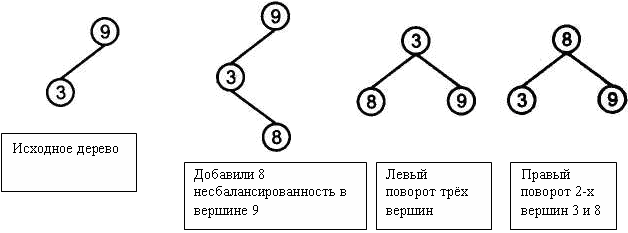
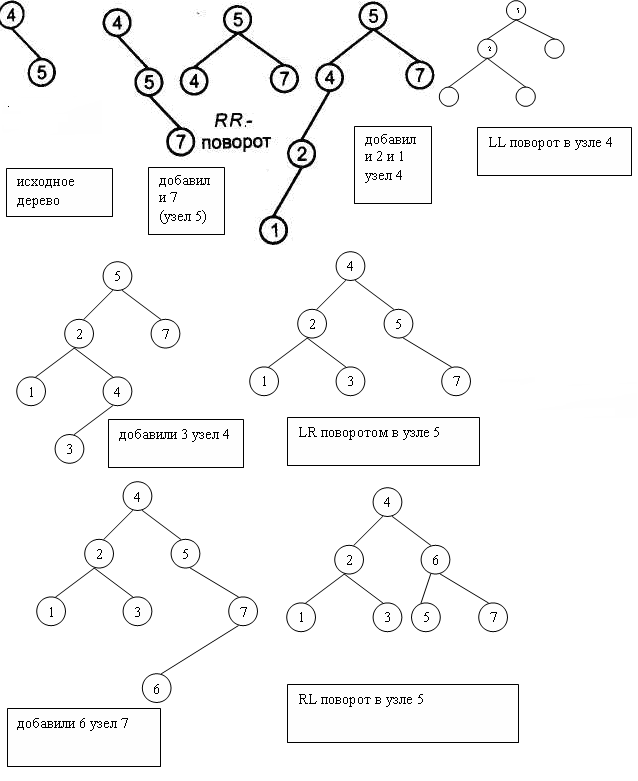


Рис. Балансировка двукратным LR поворотом

Порядок построения сбалансированного дерева из исходной последовательности элементов с ключами [8]: 4, 5, 7, 2, 1, 3, 6, можно проследить по следующему рисунку



На рис. приведено сбалансированное АВЛ - дерево, построенное из последовательности ключей 9,17,20,16,12,21,6,3,11, 4,19,14,13,1,5,2,8,18,7,10,15.

**Включение элемента в рандомизированное дерево поиска.**Напомним, что привставке нового узла в дерево, состоящее из N узлов, вероятность появления нового узла в корне дерева должна быть равна 1/(N + 1). Следовательно, для реализации алгоритма вставки необходимо, во-первых, чтобы каждый узел дерева содержал счетчик узлов в дереве (поддереве), во-вторых, имел генератор случайных чисел и, в-третьих, обеспечил вставку элемента в корень дерева.

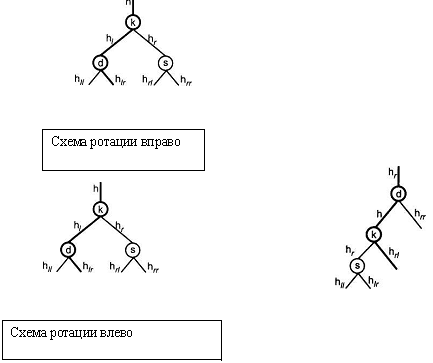
Для поддержания счетчика узлов необходимо в структуре элемента дерева предусмотреть соответствующий элемент целого типа. Для определенности примем, что структура элемента имеет вид:  
define struct RECORD  
{ int DATA;   /\* ключ+данные, здесь только ключ \*/  
RECORD \*LPTR, \*RPTR;   /\* указатели на дочерние узлы \*/  
int N; /\* счетчик узлов \*/  
int bal;     /\* признак сбалансированности в АВЛ-деревьях \*/  
}

Счетчик в заданном узле равен сумме чисел узлов в левом и правом поддеревьях + 1 (сам узел). Тогда счетчик для данного узла Т определится по рекурсивной функции:  
int count(RECORT \*T)  
{if (T==NULL) return 0 ;  
else  
return (count(T->LPTR) + count(T->RPTR) +1);  
}

Для установки счетчиков во всех узлах дерева достаточно выполнить обход всех узлов и при посещении каждого узла выполнить функцию count для этого узла. Установку счетчиков в узлах дерева можно выполнить и в процессе включения элемента в дерево. Это легко достигается в бинарном дереве поиска, но более сложно в [АВЛ](https://math.gsu.by/wp-content/uploads/courses/structure/l5.8.1.html#top10)- и рандомизированном деревьях поиска. В качестве датчика случайных чисел можно использовать стандартную функцию rand().

Рассмотрим особенности вставки элемента в корень дерева поиска. Сначала элемент вставляется в дерево обычным порядком и попадает в нижнюю часть дерева, образуя конечный узел (лист). После этого осуществляется продвижение нового узла вверх с сохранением свойств бинарного дерева поиска. Оно выполняется с использованием так называемых ротаций (ротация вправо и ротация влево). При каждой ротации вставляемый элемент поднимается на один уровень. Таким образом, число ротаций равно начальной глубине уровня вставляемого узла.

В ротации узла участвуют три связи (указателя): в ротации вправо — указатель на корень, на левый дочерний узел и на правый дочерний узел дочернего узла (на правую внучку); в ротации влево — указатель на корень, на правый дочерний узел и на левый дочерний узел дочернего узла. Ротация выполняется за счет обменов значениями указателей. При ротации вправо правая связь левого дочернего узла становится левой связью старого корня. Правая связь нового корня указывает на старый корень, а указатель на старый корень заменяется указателем на левый корень. При ротации влево левая связь правого дочернего узла становится правой связью старого корня. Левая связь нового корня указывает на старый корень, а указатель на старый корень заменяется указателем на правый корень.



При работе с деревьями к одним элементам обращаются чаще к другим реже. Встречаются ситуации, когда необходимо повторно обращаться к элементам, к которым обращались позже всех. Время доступа к таким элементам можно было бы сократить, поместив их ближе к корню дерева. Для этого достаточно поместить в корень дерева узел, к которому обращались последний раз. Тогда чаще используемые элементы и окажутся вблизи корня.

**Вставка элемента в рандомизированное дерево** проходит так. Начиная с корня дерева, осуществляется движение по ветвям дерева, как и при вставке в обычное дерево поиска. При посещении каждого узла по очередному случайному числу, выработанному генератором случайных чисел, определяется вероятность вставки элемента именно в этот узел h (корень поддерева) по условию: rand() < RAND\_MAX / (Nh + 1), где rand() — случайное число, RAND\_MAX — максимальное случайное число, Nh — счетчик узлов в данном узле, ≤ 0.

Если условие выполняется, осуществляется вставка элемента в этот узел по алгоритму вставки в корень, и на месте старого узла оказывается новый элемент. Если же ни в одном из узлов условие не выполняется, то элемент вставляется обычным порядком и становится оконечным узлом (листом) дерева.



**Обработка данных в вершинах дерева**. Различают два вида обработки: выборочную обработку отдельной вершины и последовательную обработку всех вершин. Выборочная обработка включает поиск элемента, и если он найден, то и обработку данных в вершине. Обработка данных зависит от решаемой задачи и может сводиться к извлечению данных без их изменения либо обновлению данных в вершине дерева.

При последовательной обработке осуществляется так называемый обход дерева. Обход дерева — это процесс одноразового посещения и обработки данных каждой вершины дерева. В зависимости от того, в каком направлении осуществляется движение по ветвям дерева, и когда обрабатываются данные в вершине, различают шесть видов обхода.

**Левый нисходящий обход Lpreorder**— движение сверху от корневой вершины к листьям.



Рис. Левый нисходящий обход дерева:

R — вершина, А — левое поддерево, В — правое. Сначала обрабатываются данные в вершине R, затем левый нисходящий обход левого поддерева, наконец, левый нисходящий обход правого поддерева условно можно выразить схемой R, А, В.

**Левый восходящий обход Lpostorder**– снизу вверх, от листьев вверх к корню:

* Левый восходящий обход левого поддерева;
* левый восходящий обход правого поддерева;
* обработка данных в узле.

Схема: А, В, R — обработка данных узла после поддеревьев.

**Левый смешанный обход Linorder**— слева направо, от самого левого листа вверх к корню, затем вниз к самому правому листу:

* левый смешанный обход левого поддерева;
* обработка данных в узле;
* левый смешанный обход правого поддерева.

Схема: A. R. В левое поддерево, обработка узла, правое поддерево.

При всех обходах движение по ветвям дерева продолжается до тех пор, пока указатель на узел дерева не станет нулевым.

Применяются и правые обходы: правый нисходящий Rpreorder (R, В, А), правый восходящий Rpostorder (В, A, R), правый смешанный Rinorder (В, R, А). Их алгоритмы получаются из алго¬ритмов левых обходов, если «левый» заменить на «правый». Поэтому их иногда именуют ***обратными обходами*.**

Кроме рассмотренных нисходящего, восходящего и смешанного обходов иногда применяют ***обход дерева по уровням*.** При таком обходе узлы дерева посещаются сверху вниз и слева направо, т.е. сначала корень дерева, затем узлы первого уровня слева направо, затем узлы второго уровня и т.д.

Функции обхода могут быть как рекурсивными, так и итеративными. Наиболее просто реализуются рекурсивные функции первых шести видов обхода, в которых рекурсивный обход соответствует рекурсивной структуре дерева. В них, естественно, используются системные стеки. В нерекурсивных функциях приходится использовать явный стек, который, как известно, реализует правило LIFO — последним пришел — первым ушел.

Обход по уровням (корень — первый уровень — второй уровень — ...) не соответствует рекурсивной структуре дерева (корень — поддерево — поддерево — ...). В нерекурсивной функции обхода по уровням вместо стека приходится ис-пользовать очередь типа FIFO — первым пришел — первым ушел.

Очевидно, что функция обработки данных в вершине Р(Т) зависит от конкретного применения дерева, в то время как обходы остаются одними и теми же. Бинарные деревья поиска могут быть использованы не только по своему прямому назначению — для поиска данных, но и для их сортировки. Совершая левый смешанный обход бинарного дерева поиска и располагая данные из узлов в том порядке, в котором они встречаются, получим упорядоченную по возрастанию ключей последовательность. Аналогично правый смешанный обход дерева дает последовательность, упорядоченную по убыванию ключей. Поэтому иногда о деревьях поиска говорят, как о деревьях сортировки.

**Удаление элемента из дерева**. Особенности операции удаления зависят от вида дерева. **Удаление элемента из бинарного дерева поиска**. Операция удаления элемента из дерева поиска сложнее, чем операция включения элемента. После удаления элемента дерево поиска должно сохранить свое свойство — обеспечение поиска элементов. Действия по удалению определяются тем, в каком месте дерева находится удаляемый элемент. В процедуре удаления различают следующие ситуации.  
1.   Элемента с искомым ключом нет, дерево не изменяется, возврат с признаком отсутствия ключа.  
2.     Удаляемый элемент — лист, в родительском узле соответствующий указатель на удаляемый элемент обнуляется, память из-под элемента освобождается.  
3.    Удаляемый элемент только с одним поддеревом. Корректируются указатели, память освобождается.  
4.    Удаляемый элемент имеет два поддерева. В этом случае необходимо спуститься вдоль самой правой ветви левого поддерева до конца и заменить удаляемый элемент конечным элементом с корректировкой указателей. Память освобождается. Вместо эго можно спуститься до конца вдоль самой левой ветви правого поддерева и заменить удаляемый элемент конечным элементом, скорректировать указатели и освободить память.

На примере мы рассмотрим результат удалении узла 13, имеющего два поддерева. Узел 13 заменен узлом 10. Если бы спускались по левой ветви правого поддерева, то узел 13 бы заменен узлом 14.



Рис. Удаление узла из бинарного дерева поиска

**Удаление элемента из сбалансированного АВЛ - дерева поиска.** Примечательно тем, что дополнительно выполняется балансировка дерева, если сбалансированность была нарушена в результате удаления элемента. При включении элемента нарушение сбалансированности происходило из-за роста высоты, а при удалении это случается из-за ее снижения.

**Удаление элемента из рандомизированного дерева поиска**. Рандомизированное дерево поиска обладает всеми свойствами бинарного дерева поиска. Поэтому удаление элемента можно осуществлять по тому же алгоритму, что и в обычном бинарном дереве поиска. Тогда удаляемый элемент замещается са-мым правым узлом левого дочернего узла (или самым левым узлом правого дочернего узла). Однако при этом может несколько ослабнуть «случайность» размещения элементов в узлах дерева. Поэтому замещающий узел также можно выбрать как «случайный». Естественно, это усложнит алгоритм удаления элемента. После удаления элемента потребуется корректировка счетчиков узлов.

**Сохранение и восстановление бинарного дерева**. Выполняя левый нисходящий обход дерева, сохраним ключ и данные каждого узла в файле. Указатели сохранять нет необходимости. Последовательно читая из файла сохраненные ключ и данные узлов, строим бинарное дерево. Дерево восстанавливается в прежнем виде, хотя данные в файле могут сохраниться в последовательности, несколько отличной от исходной. После сохранения в случае необходимости дерево уничтожается.

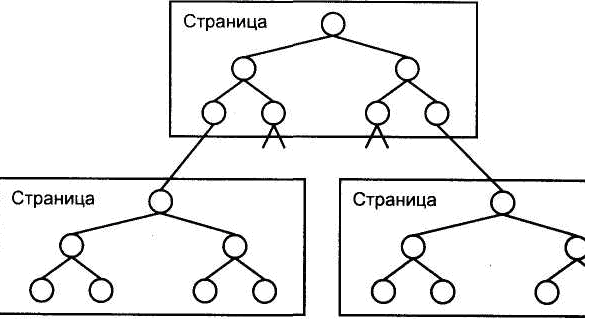
**Уничтожение бинарного дерева**. Для уничтожения бинарного дерева достаточно произвести обход дерева и при посещении каждого узла освободить память из-под него. При этом, однако, нельзя допускать того, что будет уничтожен узел, имеющий поддеревья, так как занятая ими память не будет освобождена и станет недоступной для повторного использования. Это требование удовлетворяется при левом или правом восходящем обходе. После освобождения всей занятой памяти указатель дерева необходимо обнулить.

**11. Особенности крупномасштабных деревьев**

Бинарные деревья находят широкое применение, однако есть области информатики, где невозможно ограничиваться только бинарными деревьями. Одной из таких областей является управление данными, размещенными на ВЗУ. Для этого формируются и используются крупномасштабные деревья поиска, в которых необходимы и включения, и удаления элементов. Если идеально сбалансированное бинарное дерево включает 1млн элементов, то для двоичного поиска элемента потребуется в среднем около 20 шагов (Iog2106) поиска. Поскольку дерево находится на ВЗУ, то каждый шаг включает обращение к ВЗУ для чтения, и желательна такая организация хранения данных, которая потребует меньше обращений.

Эту проблему можно было бы решить следующим образом. Бинарное дерево поиска, размещенное на ВЗУ, логически разделяется на страницы. Каждая страница включает одно поддерево с несколькими узлами (обычно десятки и сотни узлов). На ВЗУ страница размещается в блоке памяти, который считывается с устройства или записывается на него за одно обращение. Следовательно, резко сокращается количество обращений к ВЗУ. Дальнейший поиск нужного узла в пределах страницы осуществляется в оперативной памяти. Таким образом, общее число обращений к уздам дерева в процессе поиска остается прежним, но их большая часть осуществляется в оперативной памяти, поэтому время поиска значи-тельно сокращается. Схема разбиения на страницы рассмотрим на примере.

Если теперь в том же дереве с 1 млн элементов каждая страница будет включать поддерево с 100 узла ми, то средняя длина поиска будет log100106 = 3 обращениям к страницам при условии, что дерево «случайное» и потому почти сбалансировано.



Однако в наихудшем случае, когда дерево сильно перекошено число обращений может стать чрезмерно большим, а многие блоки будут заполнены только частично. Поэтому в случае крупномасштабных деревьев почти всегда требуется управление их ростом, что при большом количестве узлов вызывает затруднения.

Реальное воплощение нашла другая схема реализации крупнономасштабных деревьев, так называемые сильно ветвящиеся деревья, у которых каждый узел содержит несколько (>2) элементов данных (ключей) и имеет несколько потомков. Один узел (страница) такого дерева размещается в одном блоке внешней памяти и считывается в оперативную память за один прием. Очевидно что и в этом случае необходимо поддерживать сбалансированность дерева.

**12. В-деревья**

Для управления ростом дерева применение идеальной сбалансированности потребовало бы слишком больших затрат на балансировку, поэтому правила балансировки смягчают. Одной из распространенных структур такого типа является *В*-дерево. Эта структура была разработана Р.Бэйером и Э.Мак-Крейтом в 1972 г. **В-деревом** порядка п называется сильно ветвящееся дерево сте¬пени 2n + 1 (напомним, что степень узла — это число потомков узла, а степень дерева — наибольшее значение степени всех узлов дерева), обладающее следующими свойствами:  
1)   каждый узел, за исключением корня, содержит не менее n и не более 2n ключей (элементов) при заданном постоянном для дерева n (n ≤ m ≤ 2n). Ключи узлов одного уровня упорядочены по возрастанию слева направо;  
2)   корень содержит не менее одного и не более 2n ключей;  
3)   каждый узел является либо листом, т.е. не имеет потомков, либо имеет (m + 1) потомков, где m — число находящихся в узле ключей;  
4)   структура узла включает два массива. Первый массив из 2n ячеек предназна-чен для хранения элементов узла дерева. Второй массив из 2n + 1 ячеек содержит m + 1 указателей на потомков данного узла, остальные указатели — нулевые. Ес-тественно, в листовом узле массив указателей полностью свободен. Указатели деревьев на ВЗУ представляют собой адреса на диске;  
5)   все листья дерева расположены на одном уровне.

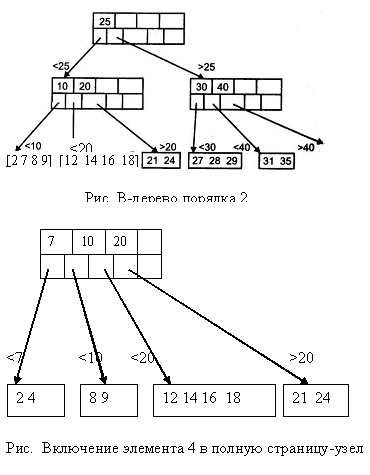
Таким образом, в В - дереве порядка n с N элементами наихудший случай при поиске потребует lognN обращений, на которые, как известно, при работе с ВЗУ тратится основная часть времени. Кроме того, коэффициент использования памяти в B-дереве составляет не менее 50%, так как страницы, содержащие узел, заполнены хотя бы наполовину.

**13. Особенности операций над B -деревьями**

**Создание В-дерева**. Выполняется в два этапа. На первом этапе определяется структура дерева и создается дескриптор дерева или просто указатель дерева. На втором этапе осуществляется заполнение дерева последовательным включением элементов в узлы дерева. Сначала заполняется корневой узел, затем узлы следующих уровней. Особенности роста дерева при включении элементов мы рассмотрим ниже. Поиск элемента в дереве. Начинается с корневого узла. Считывается страница с корневым узлом. Выполняется поиск элемента в упорядоченной последователь-ности. Если элемент найден, вращается его адрес или значение в зависимости от алгоритма обработки узла. Если же элемент не найден, а узел не листовой то определяется направление поиска и считывается следующая страница с узлом-потомком. Поиск продолжается в этом узле и в узлах нижестоящих уровней, пока не будет найден искомый элемент. Если же элемент не будет найден и в листовом узле, то возвращается признак его отсутствия в дереве (неудачный поиск).

**Включение элемента в В-дерево**. Сначала определяется узел в который должен быть включен элемент. Если элемент включается в узел, содержащий m<2n элементов, то он помещается в соответствующее место в упорядоченной последовательности находящихся в узле элементов, как, например, при вставке элемента с ключом 22 или 23 на рисунке. Включение элемента в уже заполненный узел влияет на структуру дерева и вызывает появление новой страницы.

**Теперь рассмотрим на примере**. Включим элемент с ключом 4. Его место в самом левом листовом узле, однако там нет места, так как включении получилось бы пять элементов: 2, 4, 7, 8, 9. В этом случае образуется новая страница. В старой странице остаются первые n элементов (2, 4), последние n элементов (8, 9) переносятся в новую страницу, а средний элемент поднимается в вышестоящий узелпредок с корректировкой указателей в нем



Если в результате образования нового узла и переноса среднего элемента будет переполнен узел-предок, то он, в свою очередь, разбивается на две части, образуется новая страница и т.д. При переполнении корневого узла высота дерева увеличивается на 1. Таким образом, в отличие от обычных деревьев В-дерево растет с листьев, а не с корня.

**Удаление элемента из В-дерева**. Оно несколько сложнее, чем включение, когда возникают случаи несбалансированности, т.е. в узле остается меньше n ключей. В таком случае можно либо позаимствовать один элемент из соседней страницы, если там число элементов больше n, либо слить две страницы в одну.

**Обработка данных.** Различают два вида обработки: выборочную обработку отдельного элемента и последовательную обработку всех элементов или группы элементов с ключами в заданных пределах. Выборочная обработка включает поиск элемента, и если он найден, то выборку или обновление данных в элементе.

При последовательной обработке осуществляются обход либо всех узлов дерева, либо узлов в заданных пределах и обработка элементов в посещаемых узлах. Левый смешанный обход приводит к обработке элементов в порядке возрастания их ключей.

**14.Разновидности В-дерева**

**В+-дерево**. Рассмотренная структура В-дерева предполагает, что в узлах дерева находятся полные (истинные) значения ключей. Связанные с ключом данные мо-гут находиться в том же узле, что и сам ключ, или же в другом месте. В последнем случае в узле для каждого ключа необходимо иметь указатель на данные. Тогда для обработки данных придется считывать блок с данными.

В В+-дереве данные элементов вместе с полными ключами содержатся только в листьях (концевых узлах). Во внутренних узлах В+-дерева содержатся только ключи-разделители, задающие диапазоны изменения ключей для поддеревьев. Во многих случаях ключи-разделители совпадают с истинными ключами или их частью (общий ключ или больший ключ), хотя это не обязательно. Например, если ключами элементов являются фамилии служащих, то строка «Иван» может служить ключом-разделителем для фамилий Ибрагимов, Ивакин, с одной стороны, и фамилий Иванов, Иванченко — с другой.

Поскольку все ключи и данные расположены на одном уровне в листьях, поиск всегда заканчивается в листе, при вставке в случае расщепления страницы-узла средний элемент помещается в начало новой страницы, а ключ-разделитель, образованный по его ключу, переносится на предыдущий, более высокий ypoвень. Удаление ключа всегда производится из листа.

Если все узлы-листья связать указателями слева направо, то обеспечивается последовательный доступ к элементам в порядке возрастания их ключей, что обеспечивает обработку всех элементов или последовательности элементов в за-данных пределах.

Таким образом, В+-деревья во многом превосходят B - деревья, а уступают В-деревьям в том, что требуют несколько больше памяти для представления (ключи-разделители и ключи в листьях).

**В\* -дерево**. Отличается от В-дерева более высоким коэффициентом заполнения. Если в В-дереве каждый узел должен быть заполнен не менее чем наполовину, то для В\*-дерева требуется, чтобы каждый узел был заполнен элементами не менее чем на 2/3. Алгоритмы вставки и удаления элементов сложнее чем в В-дереве.

**(2—3)-дерево**. В-дерево порядка 1 носит название (2-3 дерева (двоичного В-дерева), так как каждый узел в таком дереве может иметь 2 или 3 потомка. Это дерево иногда применяется при работе в основной памяти, применение его для работы на ВЗУ нерационально.

**1.Способы решения задач**

Существуют четыре основных способа решения задач:  
1)   применение формулы;  
2)   использование рекурсий;  
3)   использование алгоритма;  
4)   метод перебора, метод проб и ошибок и др.

Здесь нужны некоторые пояснения. Вам известно, что для решения любой задачи необходимо разработать алгоритм ее решения. Когда говорится, что способом решения задачи является применение формулы или использование рекурсий, это означает, что основные результаты получаются по некоторым известным формулам или с использованием рекурсивных определений и рекурсивных функций.

*Пример 4.1.*  
Пусть требуется вычислить сумму n чисел натурального ряда: ∑*i*=1+2+...+*n*

Эту задачу можно решить первыми тремя способами. Известно, что эта сумма определяется по формуле S(n)=∑*i* =n(n+1)/2  
При ее использовании сложность решения не зависит от n и равна 0(1).

Решение этой же задачи можно определить рекурсивно. S(n) = S(n - 1) + n, S(1) = 1 или по итеративному алгоритму с использованием цикла. Тогда сложность алгоритма равна О(n).

Аналогично можно вычислить и значения:  
∑*i2*=n(n+1)(2n+1)/6; ∑*i3*= (1+2+...+n)2 =(n(n+1)/2)2

Под способом решения задач с использованием алгоритма понимается следующее. Многие задачи решаются по известным алгоритмам, разработанным методами и приемами вычислительной и дискретной математики и других наук. Для решения других задач разрабатываются новые алгоритмы с использованием уже известных алгоритмов решения подобных задач.

Существует множество задач, которые принято называть задачами выбора. **Пример 1**. Пусть известны попарные расстояния между n городами. Требуется найти кратчайший путь между некоторой парой городов.  
**Пример 2.**Имеется n различных деталей и n станков. На каждом станке можно обрабатывать любую деталь, но время обработки одной и той же детали на различных станках может быть различным. Пусть заданы времена обработки каждой детали на каждом станке. Как разместить n деталей по n станкам так, чтобы суммарное время работы было минимальным?  
**Пример 3.** Имеется n предметов, вес каждого из них известен. Можно ли разделить их на две равные по весу части?

Эти задачи — задачи выбора имеют следующие общие свойства:

1) конечность множества вариантов выбора (путей между городами, размещение деталей, распределение предметов между частями);  
2) каждому варианту соответствует количественная характеристика (длина пути, суммарное время обработки деталей, вес предметов в каждой части);  
3) требуется выбрать вариант, числовая характеристика которого удовлетворяет заданному условию, или определить существование такого варианта. Иногда нужен только один вариант или вариант с экстремальной характеристикой (оптимальный вариант), иногда — все варианты, а иногда — не сам вариант, а только его числовая характеристика.

Казалось бы, решить такие задачи просто — поочередно пересмотреть все варианты и выбрать требуемый, т.е. использовать алгоритм прямого перебора (переборный алгоритм). Однако такой подход приемлем только тогда, когда число вариантов небольшое. Применение ЭВМ позволяет методом перебора решать задачи выбора с большим числом вариантов, однако и при этом возможное число интересных для практики задач невелико. В то же время число таких задач увеличивается.

Для решения некоторых задач выбора разработаны точные алгоритмы, приводящие к получению результатов за приемлемое время. Во многих случаях, когда такие алгоритмы невозможно получить или они являются недопустимо сложными, применяют различные методы целенаправленного перебора вариантов, сокращающих их число. Это «жадные» (градиентные) алгоритмы, метод ветвей и границ, алгоритмы с возвратом, динамическое программирование и др. Для решения задач поиска оптимального варианта часто используются эвристические алгоритмы.

**Эвристический алгоритм** обладает двумя свойствами:  
1) он находит хорошие, хотя и не обязательно оптимальные решения;  
2) его можно быстрее и проще реализовать, чем любой точный алгоритм, дающий оптимальное решение.

**2. Применение рекурсий**

Схема рекурсивной функции и особенности ее выполнения были подробно рассмотрены нами ранее.

Рекурсии используются тогда, когда задачу можно решить путем разбиения ее на меньшие подзадачи, выполняемые с помощью одного и того же алгоритма. Процесс разбиения не может продолжаться бесконечно. Он завершается тогда, когда достигается простейшее решение подзадачи (условие останова). Таким образом, рекурсия действует по принципу «разделяй и властвуй». Во многих рекурсивных функциях используются два рекурсивных вызова, каждый из которых работает со своей половиной данных. Это наиболее важный случай метода «разделяй и властвуй». Такие функции использовались при работе с деревьями: левое поддерево — правое поддерево. В дальнейшем мы увидим применение этого метода при бинарном поиске, сортировке слиянием, обеспечивающим оптимальную производительность поиска и сортировки.

Применение рекурсий во многих случаях дает весьма простые и изящные решения задач. В то же время рекурсивный алгоритм может оказаться неэффективным. Примером может служить рекурсивная функция определения числа Фибоначчи. В ней имеются два рекурсивных вызова, причем второй рекурсивный вызов не использует вычисления, выполненные во время первого вызова. Это приводит к повторным вычислениям. Время вычисления FN+1 примерно в 1,618 раза («золотое сечение») больше времени вычисления FN.  
**Пример** . Вычисление чисел Фибоначчи, которые определяются рекуррентными соотношениями: F0 = О, F1 = 1, Fn= Fn-1 + Fn-2. Рекурсивная функция на Си будет иметь вид:

int Fib(int n)  
{if (n < 2) return n;  
else return (Fib(n - 1) + Fib(n - 2)); }

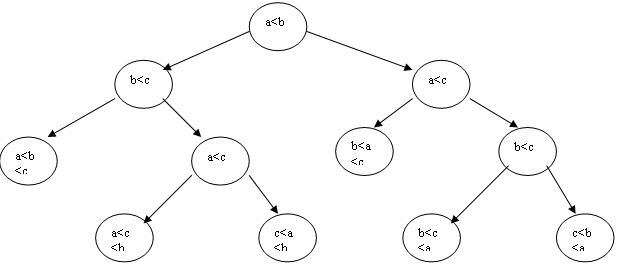
Каждое обращение к функции Fib(n) при n > 1 приводит к двум дальнейшим обращениям, т.е. общее число обращений растет экспоненциально, сложность алгоритма 0(2n). В то же время нерекурсивный вариант функции, приведенный ниже, имеет временную сложность О(n).

int Fib(int n)  
{int i, x, у, z;  
if (n < 2) return n;  
x = 0;                         /\* Fo \*/  
y=1;                        /\* F1 \*/  
for (i =2; i≤n; i++)  
{z = x+y; x = y; y = z;  
}  
return z;  
}

**3. Дерево решений**

Программу, содержащую разветвления, можно представить в виде двоичного дерева, называемого **деревом решений**. Каждый внутренний узел дерева представляет одну из точек разветвления в программе и соответствует условию (правилу), по которому осуществляется движение по правой или левой ветви. Возможные результаты (решения задачи) представляются листьями.

Если для конкретных исходных данных задача имеет одно единственное решение, то оно находится движением по соответствующим ветвям дерева, пока не будет достигнут лист с искомым решением. Длина пути от вершины дерева до отдельных листьев дерева, зависящая от конкретных исходных данных, может быть различна. В наихудшем случае, соответствующем самому длинному пути, временная сложность решения задачи равна высоте дерева решений. Иногда высота дерева определяется как функция от размера задачи. Рассмотрим дерево решений для задачи упорядочения трех чисел a, b и с в порядке возрастания их значений. Условие движения по ветвям дерева определяется сравнением пары чисел. Пусть все числа имеют разные значения. Дерево решений можно представить, как показано на рис.



Дерево решений упорядочения трех чисел

Часто возникают задачи, когда необходимо найти все возможные решения или для нахождения подходящего решения приходится перебирать все или большинство возможных решений. В этом случае требуется посетить все или некоторое количество листьев дерева по некоторому порядку, например левым смешанным обходом дерева (Inorder).

**Пример.**

Пусть имеется кодовый замок с n двоичными переключателями («вкл»-«выкл»). Число возможных комбинаций ключа равно 2n. Попытаемся открыть замок наугад, не зная кода. Построим дерево решений, оно будет иметь высоту n и 2n листьев. Длина пути до всех листьев одна и та же и равна n. Решение задачи сводится к полному перебору всех n-мерных векторов с нуль -единичными элементами. Для открытия замка придется сделать максимум 2n, в среднем 2n/2 попытки. Временная сложность нахождения одного решения О(n), а всех 2n решений не превышает О(n\*2n), т.е. является экспоненциальной. Дерево решений для этой задачи будет иметь следующий вид:

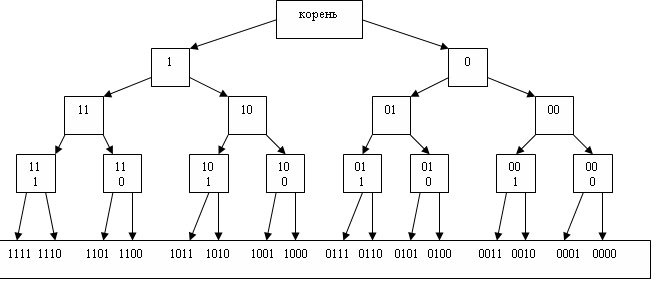


Рис. Дерево решений поиска ключа

Дерево решений строится только для удобства разработки и анализа алгоритма, а в программе, естественно, никакого дерева нет.

В чем различие двух рассмотренных деревьев решений? В задаче сортировки направление движения по ветвям дерева полностью и однозначно определяется исходными данными (заданными значениями а, b, с), поэтому решение находится не больше чем за три шага, т.е. при любых исходных данных за два или три шага мы достигаем того из листьев, который определяет искомое решение.

В задаче с кодовым замком исходные данные не задаются, просто ставится задача нахождения среди 2n возможных решений той комбинации кода, которая является ключом, открывающим замок. Таким образом, движение по ветвям дерева определяется не значениями исходных данных, а некоторыми общими правилами. В приведенном варианте дерева решений движение по левой ветви добавляет в код единичный бит, а движение по правой ветви — нулевой бит. Следовательно, любые 4 ветви, ведущие от корня к одному из листьев, дают 4-мерный вектор-код одного из возможных решений.

**4. Переборные задачи**

Методы перебора применяются при решении многих задач. **Суть переборной задачи** заключается в том, что необходимо найти на конечном множестве X элемент х, удовлетворяющий совокупности условий К(х), в предположении, что пространство поиска X содержит некоторое конечное число различных точек.

1. Алгоритм решения переборной задачи можно представить в следующем виде: Взять из множества X еще не рассмотренный элемент X0.  
2. Проверить совокупность условий К(х0).  
3. Если какое-либо условие не выполнено, то перейти к п.1.  
4. Если выполнены все условия, то х0 является решением задачи. Если же возможно несколько решений и требуется найти все их, то вернуться к п.1, пока не будут перебраны все элементы множества X.

Сложность алгоритма будет определяться во многом тем, как представлены элементы, в каком порядке их перебирать, в чем заключается проверка условий K(X0) И сколько элементов нужно найти.

Рассмотрим задачу поиска элемента с заданным значением в двухмерном массиве, элементами которого являются значения скалярного типа. Для выборки элемента достаточно задать его индексы, организовав вложенные циклы. Обработка элемента сводится к сравнению его значения с заданным числом. При их равенстве поиск прекращается, результатом являются индексы найденного элемента. При неудачном поиске, когда элемента с заданным значением нет в массиве, перебираются все элементы; поиск прекращается после обработки последнего элемента. Результатом является возврат некоторого признака «элемент не найден». Поиск в массиве, упорядоченном по возрастанию значений элементов, несколько упрощается. Если искомого элемента нет, то поиск прекращается тогда, когда заданное значение элемента будет меньше значения выбранного элемента.

В этой задаче множество X включает все n элементов массива, условием К(х) является проверка на равенство значения выбранного элемента и заданного значения. Обработка одного элемента никак не связана со значениями других элементов массива. Поэтому сложность алгоритма равна О(n).

В задаче поиска наибольшего или наименьшего элемента массива приходится перебирать все элементы, результат поиска зависит от соотношения значений всех элементов. Более сложной является задача поиска минимального элемента среди максимальных элементов строк массива или задача поиска седловой точки, когда минимальный элемент в строке одновременно является максимальным элементом в столбце массива.

Задачи подобного типа решаются применением циклов, и их программирование особых затруднений не вызывает. Такие задачи обычно не относят к переборным задачам. Задача усложняется, когда элементами множества X являются некоторые последовательности. Такие задачи можно решить методом перебора.

В теории перебора используются два фундаментальных понятия:*номер хода* i и *номер варианта j*. Для каждого хода i перебираются варианты j, пока не будет найдено «подходящее решение». Особенности решения конкретной задачи перебора вариантов определяются этим «подходящим решением» (совокупностью условий К(х)). Им может быть первое найденное решение; все возможные решения; оптимальное среди всех возможных решений т.д. В любом случае задача должна быть решена, если не за минимально возможное время, то по крайней мере за приемлемое для данной задачи.

**5. Метод ветвей и границ**

**Алгоритм метода ветвей и границ** похож на алгоритм с возвратом, он также использует дерево решений и применим для широкого круга дискретных комбинаторных задач. Если алгоритм с возвратом направлен на нахождение одного или нескольких решений, удовлетворяющих определённым условиям, то алгоритмы ветвей и границ ориентированны преимущественно на оптимизацию.

В дереве решений для каждого узла (вершины) определяется некоторая функция стоимости. Цель – найти конфигурацию, на которой функция стоимости дос-тигает максимального или минимального значения. Алгоритмы ветвей и границ, как правило, довольно сложны. Для сокращения процесса поиска оптимального решения используется метод поиска, основанный на правилах вычёркивания ветвей. Сущность метода ветвей и границ рассмотрим на примере решения задачи бивалентного линейного программирования.  
**Пример.**

Имеются единичные ресурсы х1, х2,…хn. Применение каждого ресурса хi даёт прибыль сj. Необходимо выбрать такие ресурсы хi, которые обеспечивают наибольшую прибыль  
maxP=max∑cjxj при ограничениях  
∑aijxj ≤bi, i=1,2,…m и xj=0 или 1, j=1,2,…n.

Построим двоичное дерево решений следующим образом. Ветви, отходящие от корня (узла нулевого уровня), будут соответствовать выбору ресурса x1: левая ветвь — ресурс выбран (x1 = 1), правая ветвь — ресурс не выбран (x1 = 0). Аналогично левые ветви, исходящие из узлов (к - 1)-го уровня, соответствуют выбору ресурса Хk (хk = 1), к = 1,2,...,n, а правые — (хk = 0). В результате получаем идеально сбалансированное дерево высоты n с 2n листьями.

Поиск оптимального решения осуществляется обходом дерева. Для конкретности рассмотрения примем порядок левого нисходящего обхода Preorder. Это означает, что сначала двигаемся по левым поддеревьям вниз до конца, затем возвращаемся на предыдущий уровень и двигаемся по правым поддеревьям. Каждому узлу к-го уровня (к = 0,1,2,...,n) соответствует некоторый k-мерный вектор с нуль-единичными элементами, который позволяет однозначно вычислить значение прибыли для этого узла  
Pk= ∑cjxj  
и определить, удовлетворяются ли условия  
∑aijxj≤bi , i=1,2,…m.

Корню дерева (к = 0) соответствуют отсутствие ресурсов и нулевая прибыль Р = 0. При обходе по мере посещения узлов дерева, удовлетворяющих нашим ус-ловиям, выбирается максимальное текущее значение прибыли Рi, и запоминается соответствующий ей к-мерный вектор Хk.

Пусть мы находимся в некотором промежуточном узле к-го уровня, в котором удовлетворяются наши условия. Прибыль в этом узле составляет:  
Pk=∑cjxj  
что соответствует выбранной комбинации ресурсов х1,х2,…хk.

Целесообразно ли двигаться дальше вниз от этого узла? Поскольку движение по правой ветви вниз на один уровень не приводит к выбору нового ресурса (хk+1=0), то и значение прибыли не изменяется. Поэтому вопрос касается только движения вниз по левой ветви.

Если бы теперь мы выбрали все оставшиеся ресурсы (хk+1 = 1, …,хn=1), т.е. двигались бы до конца по левым ветвям, то прибыль увеличилась бы и составила:  
Pk+m=∑cjxj+∑cm

Это максимально возможное значение прибыли при движении от рассматриваемого узла. Поэтому движение вниз от этого узла имеет смысл только тогда, когда Рk+m > Рt , в противном случае необходимо возвращаться назад.

Таким образом, текущее максимальное значение прибыли Pt он деляет границу движения вниз от данного узла, и ветви рассматриваемого узла вычеркиваются, т.е. от дальнейшего обхода исключаются все узлы-потомки.

Если же Рk+m> Рt, то обход дерева по левым поддеревьям продолжается. Придя в следующий узел, т.е. выбрав хk+1=1, проверяем, удовлетворяются ли условия  
∑aijxj≤bi , i=1,2,…m.

Если да, то вычисляем новое значение прибыли, при необходимости обновляем текущее максимальное значение прибыли Рt, и продолжаем движение вниз по левым поддеревьям по рассмотренному алгоритму. Если эти условия не удовлетворяются или мы уже обработали конечный узел (лист) поддерева, то возвращаемся назад на один или несколько уровней и начинаем обход правого поддерева. Нужно отметить, что движение по правой ветви вниз на один уровень всегда осуществляется безусловно, так как при этом значение прибыли не изменяется и не нарушаются наши условия.

По завершении обхода как результат получаем искомое максимальное значение прибыли max Р = Рt, и оптимальный вектор X\*, у которого единичные элементы соответствуют выбранным ресурсам.

**6. Метод проб и ошибок**

Рассмотрим это на примере хорошо известной задачи о восьми ферзях. Восемь ферзей нужно расставить на шахматной доске так, чтобы ни один ферзь не угрожал другому: ферзь бьет все фигуры, находящиеся на той же горизонтали или вертикали.

Процесс расстановки ферзей может выглядеть так. Поставим первого ферзя на какую-нибудь клетку. Затем поставим второго ферзя на первую клетку и проверим, что ему не угрожает первый. Если угрожает, то передвинем второго ферзя и снова проверим и так до тех пор, пока второй ферзь не окажется на допустимой клетке. Затем будем двигать третьего ферзя и т.д.

В рассматриваемой задаче номером хода i будет порядковый номер ферзя, а номером варианта j — порядковый номер попытки установить этого ферзя после того, как предыдущие ферзи установлены. Может оказаться, что в ходе расстановки i-го ферзя все варианты будут неудачными, т.е. мы не сумеем поставить его на доску. В таком случае мы должны будем вернуться на ход назад и установить предыдущего (i- 1)-го ферзя по-другому, т.е. перейти к следующему варианту его расстановки. Очевидно, что для этого надо знать последний рассмотренный вариант установки (i - 1)-го ферзя. Затем увеличиваем номер варианта и продолжаем просмотр вариантов установки этого ферзя.

Итак, процесс расстановки ферзей выглядит следующим образом. Мы движемся вперед, увеличивая номер хода. Для каждого хода движемся вбок, подбирая допустимый вариант, и идем вперед к следующему ходу, если вариант подобран. Если невозможно подобрать вариант, то возвращаемся на ход назад и продолжаем движение вбок, начиная со следующего варианта. После установки последнего ферзя записываем полученное решение. В этих движениях вперед и назад по номерам ходов и состоит особенность схемы перебора.

**7.Динамическое программирование**

Многие прикладные задачи сводятся к отысканию минимума (максимума) некоторой функции на некотором множестве. Очень, часто это множество представляет собой последовательность элементов а1, a2,...,an. Тогда получаем задачу поиска минимума в множестве последовательностей. Искомый минимум может быть получен путем полного перебора всех последовательностей. Meтод динамического программирования позволяет сократить перебор.

Пусть определена функция F, которая сопоставляет некоторое число каждой последовательности из некоторого допустимого множества допустимых последовательностей. Требуется найти такую последовательность а1, a2,...,an, которая минимизирует функцию F(a1,...,an). Метод динамического программирования применим только тогда, когда функция F удовлетворяет принципу оптимальности Беллмана. Этот принцип требует, чтобы всякий отрезок ai, ai+1,…,am оптимальной последовательности а1, a2,...,an (1 ≤ i≤m≤n) был бы сам оптимальным среди всех отрезков совпадающих с ним по числу элементов и в крайних элементах. Это свойство позволяет найти оптимальную последовательно, постепенно удлиняя уже найденные от начала (конца) последовательности отрезки. Важным классом функций, удовлетворяющих принципу оптимальности Беллмана, являются аддитивные функции, равные сумме некоторых функций, зависящих только от соседних элементов последовательности: F(a1...,an) =∑G(ai,ai+1)

**Динамическим программированием** называют процесс пошагового решения задач, когда на каждом шаге выбирается одно решение из множества допустимых на этом шаге решений, причём такое, которое оптимизирует заданную функцию.

В качестве примера можно привести задачу поиска кратчайшего, пути на графе. Будем считать, что элементами последовательности являются n вершин графа. Всей последовательности соответствует путь по графу, проходящий по перечисленным в ней вершинам. Функция G сопоставляет каждой дуге графа ее длину: G(i,j) — длина дуги от вершины i к вершине j, а ∑G(i,j) суммарная длина пути, задаваемого последовательностью. Задача поиска оптимальной последовательности здесь и является задачей поиска кратчайшего пути на графе между начальной и конечной вершинами или кратчайших путей, между заданной вершиной и всеми остальными. Метод динамического программирования сводит перебор всех путей к перебору вершин графа, которых обычно меньше, чем путей.

**8.Алгоритмы сжатия данных**

Задачи сжатия данных не относятся к задачам выбора, тем не менее ознакомимся с некоторыми алгоритмами их решения.

Несмотря на наличие запоминающих устройств большой емкости и большие скорости передачи данных, проблема сжатия данных остается актуальной. Решению этой проблемы уделяется достаточно серьезное внимание. Разработано много способов сжатия данных различных видов: текстовых, графических и т.д. Кратко рассмотрим некоторые алгоритмы, применяемые к байтовому представлению данных.

При рассмотрении алгоритма сжатия нас интересуют такие его характеристики как коэффициент сжатия данных, ориентированность на исходные данные, временная сложность алгоритма, потребность в дополнительной памяти.

Методы сжатия можно разбить на два класса:*методы, ориентированные на конкретную структуру или содержимое данных и разрабатываемые в расчете на конкретные приложения*; методы, которые используются во многих приложениях.

Методы сжатия первого класса рассмотрим на примере сжатия данных в телефонном справочнике о жителях населенного пункта.

Пусть информация о жителе представляется в виде записи, имеющей структуру с элементами: фамилия, имя, отчество, дата рождения, место рождения, номер телефона, адрес (улица, номера дома и квартиры).

Для сжатия данных такого вида можно применить несколько способов. Переход от естественных обозначений к более компактным представлениям. Данные, представленные в виде, удобном, чтения человеком, можно сжать за счет другого, более компактного представления. Дату мы привыкли представлять в символьном виде, например 12 апр 1961, 12.04.1961, 12/04/1961. При этом каждая дата будет записываться в строку длиной 11 или 12 байт. Если удалить все разделительные символы, то в самой краткой форме: 12041961 — она займет 9 байт. Если же эту строку преобразовать в число, то для ее хранения в памяти достаточно 4-х байт. Таким образом, строка символов длиной 9 байт сжимается до 4 байт. Аналогичным способом номера телефонов также могут быть сжаты до 4 байт. Имеется другой способ представления как общее число дней, прошедших к данной дате, считая от какой-либо исходной, фиксированной для данного приложения даты. Можно также каждый элемент даты представлять определенным числом бит: день — 5 битами, месяц — 4 битами и год - 11 битами, тогда дату можно упаковать в 20 бит и поместить в 3 байта.

**Кодирование элементов, имеющих одинаковые значения.** Обычно в населенном пункте встречается много однофамильцев, а число различных фамилий и имен ограничено. Наиболее распространенные 256 имен можно закодировать одним байтом, а имена не попавшие в этот перечень, представлять полностью. Таким же способом можно закодировать названия улиц. Фамилии отличаются большим разнообразием, чем имена. Учитывая, что фамилии часто образованы от имен, ту часть фамилии, которая впадает с именем, можно заменить кодом имени.

Этот способ сжатия данных требует наличия кодовой таблицы как и при кодировании, так и при декодировании данных.  
Сжатие упорядоченных данных. Поскольку телефонный справочник лексикографически упорядочивается по фамилиям, то начальные символы фамилий в соседних элементах часто совпадают. Тогда в последующих элементах справочника вместо начальных символов фамилий в первом байте можно указать число совпадающих символов, например:

исходное представление                          сжатое представление

Иванов                                                            Иванов  
Иванов                                                            6  
Ивановский                                                    6ский  
Иванченко                                                      4ченко  
Ивашкин                                                         3шкин  
Ивашов                                                           4ов или 3шов  
Ивлиев                                                            2лиев

При таком кодировании не требуется дополнительной таблицы, но последующие записи оказываются зависимыми от предыдущих, поэтому алгоритмы добавления и удаления элементов в справочнике усложняются. Следующие алгоритмы сжатия применимы во многих приложениях.

**Групповое кодирование**. Это один из самых старых и простых алгоритмов сжатия данных. Сжатие происходит за счет того, что в исходных данных встречаются цепочки одинаковых байтов. Замена их на пары <счетчик, значение> уменьшает избыточность данных. Дополнительной памяти алгоритм не требует, выполняется быстро за один проход. При передаче данных этот способ применяется и для сжатия цепочки одинаковых битов.

**Кодирование одинаковых цепочек байтов**. Сжатие осуществляется за счет повторяющихся одинаковых цепочек байтов. Алгоритмы этого способа различаются методами поиска повторяющихся цепочек (методами поиска подстроки в строке) и значительно сложнее алгоритмов группового кодирования.

**Алгоритм Хаффмана**. Является одним из классических алгоритмов. Алгоритм использует частоту (вероятность) появления одинаковых элементов (символов) в данных. Элементы входного потока, которые встречаются чаще (имеют большую вероятность), кодируются цепочкой битов меньшей длины, а встречающиеся редко — цепочкой большей длины. Алгоритм выполняется за два прохода: сбор статистики и непосредственное кодирование. Алгоритм требует наличия кодовой таблицы. Таблицы могут быть постоянными или создаваться по результатам сбора статистики при первом проходе. Средняя длина кода элемента определяется выражением  
L=∑Li′pi  
где n — число различных элементов в сжимаемых данных; рi — вероятность появления i-го элемента; Li — длина кода i-ro элемента.

**Коды Хаффмана** обладают двумя важными свойствами. Во-первых, коды Хаффмана являются двоичными кодами с минимальной средней длиной. Во-вторых, коды Хаффмана обладают префиксным свойством. Это означает, что ни один код в полученном множестве кодов не имеет префикса другого кода. Поэтому обычным сканированием строки с кодами Хаффмана можно выделить каждое следующее закодированное значение в силу уникальности префикса.  
**Например**, пусть некоторое сообщение включает только символы а, в, г, д, о с вероятностями появления их 0.12, 0.40, 0.15, 0.08, 0.25, и они закодированы последовательностями битов: а — 000, в — 11, г — 01, д — 001, о — 10 соотвественно. Тогда двоичная последовательность битов «1110001000» будет соответствовать исходной последовательности символе «вода». Действительно, сканируя двоичную последовательность выделяем префиксы (коды): 11 (в), 10 (о), 001 (д) и 000(а).

Недостатком кодов Хаффмана является то, что из-за переменности их длины при декодировании, для выделения очередного элемента требуются дополнитель-ные затраты времени. Кроме того, потеря или изменение одного бита в битовой последовательности может привести к печальным последствиям.

Как видим, для реализации алгоритма Хаффмана нeобxoдимо построить кодовую таблицу. Она используется и для кодирования исходного текста, и для декодирования. Элементами таблицы являются структуры вида: <символ>, <код>, <длина кода>. Следует выбирать таблицу, наиболее соответствующую характеристикам конкретных кодируемых данных.

Существует много различных модификаций префиксных кодов переменной длины. Существуют алгоритмы, которые позволяют с помощью бинарного дерева Хаффмана построить кодовую таблицу.

**1. Общие понятия СОРТИРОВКИ**

**Сортировку** следует понимать как процесс перегруппировки однотипных элементов структуры данных в некотором определенном порядке. Цель сортировки - облегчить последующий поиск, обновление, исключение, включение элементов в структуру данных. На отсортированных данных легче определить, имеются ли пропущенные элементы, все ли элементы проверены, легче найти общие элементы двух однотипных структур, слить их воедино. Сортировка является важным средством ускорения работы практически любого алгоритма, в котором требуется частое обращение к определенным элементам структуры данных.

Разработано множество алгоритмов сортировки, однако нет алгоритма, который был бы наилучшим в любом случае. Эффективность алгоритма сортировки может зависеть от ряда факторов, таких, как: число сортируемых элементов; диапазон и распределение значений сортируемых элементов; степень отсортированности элементов; характеристики алгоритма (сложность, требования к памяти и т.п.); место размещения элементов (в оперативной памяти или на ВЗУ).

В зависимости от последнего фактора все методы сортировки разбивают на два класса:*внутренняя сортировка* (сортировка массивов) и *внешняя сортировка* (сортировка файлов, или сортировка последовательностей).

Сортируемые элементы часто представляют собой записи, данных определенной структуры. Каждая запись имеет поле ключа, по значению которого осуществляется сортировка, и поля данных. При рассмотрении алгоритмов сортировки нас интересует только поле ключа, поэтому другие поля опускаются из рассмотрения.

Метод сортировки называется устойчивым, если в процессе сортировки относительное расположение элементов с одинаковыми (равными) ключами не изменяется. Устойчивость сортировки желательна, если речь идет об элементах, уже отсортированных по некоторым другим ключам (свойствам), не влияющим на ключ, по которому сейчас осуществляется сортировка.

**Внутренняя сортировка:**

**Сортировка методом прямого включения, Сортировка методом прямого выбора, Сортировка методом прямого обмена (сортировка методом пузырька), Шейкерная сортировка**

**2.1. Сортировка методом прямого включения (Insertion Sort)**

* Этот метод сортировки работает по принципу вставки элементов в отсортированную часть массива. Сортировка начинается с второго элемента, который вставляется в уже отсортированную часть массива. Каждый новый элемент вставляется на своё место среди уже отсортированных элементов.
* **Время работы**: O(n²) в худшем случае, O(n) в лучшем (если массив уже отсортирован).

**2.2. Сортировка методом прямого выбора (Selection Sort)**

* Алгоритм находит минимальный (или максимальный) элемент в неотсортированной части массива и меняет его местами с первым элементом. Затем этот процесс повторяется для оставшейся части массива, и так до тех пор, пока весь массив не станет отсортированным.
* **Время работы**: O(n²) независимо от состояния массива (практически не зависит от начальной сортировки).

**2.3. Сортировка методом прямого обмена (сортировка методом пузырька) (Bubble Sort)**

* Алгоритм проходит по массиву несколько раз, сравнивая соседние элементы и меняя их местами, если они идут в неправильном порядке. После каждого прохода наибольший элемент "всплывает" на своё правильное место.
* **Время работы**: O(n²) в худшем и среднем случаях. В лучшем случае (если массив уже отсортирован) O(n).

**2.4. Шейкерная сортировка (Cocktail Shaker Sort)**

* Этот алгоритм является улучшенной версией сортировки пузырьком. Он проходит по массиву в обе стороны: сначала слева направо (как пузырёк), затем справа налево, повторяя этот процесс до тех пор, пока весь массив не будет отсортирован.
* **Время работы**: O(n²) в худшем и среднем случаях. В лучшем случае (если массив уже отсортирован) O(n).

Все эти методы являются **внутренними сортировками**, то есть они работают с данными, которые помещаются в память целиком, и их сложность обычно O(n²), что делает их неэффективными для больших массивов.

Основными требованиями к алгоритмам сортировки, как и ко всяким алгоритмам, являются требования по памяти и по времени выполнения. Это предполагает, что сортировка элементов массива выполняется на месте, без передачи их в результирующий массив. Хорошей мерой эффективности алгоритма по времени могут быть число необходимых сравнений ключей С и число пересылок М, которые зависят от числа элементов n.

Самыми простыми методами сортировки являются так называемые *прямые методы*, они требуют порядка О(n2 ) сравнений ключей. *Быстрые (улучшенные) методы* сортировки в самом благоприятном случае требуют порядка O(n\*log2n) сравнений, имеются многочисленные варианты прямых и быстрых методов сортировки.

Прямые методы сортировки, не требующие дополнительной памяти, можно разбить на три категории:

* сортировка методом прямого включения;
* сортировка методом прямого выбора;
* сортировка методом прямого обмена (сортировка методом пузырька).

Быстрые методы сортировки являются улучшением прямых методов сортировки.

**2.1. Сортировка методом прямого включения**

Пусть имеется последовательность элементов а0, а1, a2, ..., аn-1. Эта последовательность мысленно делится на две последовательности: упорядоченную последоваельность а0, а1, ....,ai-1 и исходную, неупорядоченную ai, ai+1, ..., an-1. Очевидно, что первоначально упорядоченная последовательность состоит из единственного элемента а0. При каждом шаге, начиная с i = 1 и увеличивая i каждый раз на единицу, из исходной последовательности извлекается i-й элемент аi, и вставляется в готовую последовательность в нужное место, при необходимости сдвигая элементы готовой последовательности на одну позицию вправо вплоть i-й позиции. Поиск подходящего места осуществляется сравнением аi, с элементами aj, j=i-1, i-2, ..., 0, и одновременным обменом местами аi, и cj, если аi<aj, т.е. сдвигом aj на одну позицию вправо. Процесс поиска заканчивается при выполнении одного из условий:

* ai≥аj,
* достигнут левый конец готовой последовательности.

Этот метод обеспечивает устойчивую сортировку.

Для процедуры сортировки методом прямого включения число сравнений С и число пересылок М таковы: Cm,n= n-1; Ссредн = (n2 + n - 2)/4; Сmах = (n2 - n)/2 - 1; Mmin = 2(n - 1); Мсредн= (n2+ 9n - 10)/4; Мmах = (n2 + Зn - 4)/2.

Рассмотрим пример программы сортировки методом прямого включения.  
Пример.

/\* СОРТИРОВКА МЕТОДОМ ПРЯМОГО ВКЛЮЧЕНИЯ \*/  
#include  
#define n 8  
int A[n]={27,412,71,81,59,14,273,87};  
main()  
{ void Sis(int A[],int nn);  
int j;  
printf("\n Исходный массив: \n\t");  
for (j=0; j<n; j++)  
printf("%d\t",A[j]);  
printf("\n");  
Sis(A.n);  
printf("\n Отсортированный массив :\n\f);  
for (j=0; j<n; j++)  
printf("%d\t",A[j]);  
printf("\n");  
getch(); return 0; }

/\*ФУНКЦИЯ СОРТИРОВКИ ПРЯМЫМ ВКЛЮЧЕНИЕМ\*/  
void Sis(int A[],int nn)  
{ int i,j,k;  
printf("\n Отладочная печать по шагам сортировки");  
for (i=1; i<nn; i++)  
{k=A[i];  
j=i-1;  
while (k<A[j] && j >=0)  
{ A[j+1]=A[j];  
j--;}  
/\* Отладочная печать \*/  
printf("\n i=%d",i);  
for (j=0; j<nn; j++)  
printf("\t%d",A[j]);  
}  
}

Исходный массив:  
27  412   71   81   59   14   273   87  
Отладочная печать по шагам сортировки:  
i=1  27  412  71  81  59  14  273  87  
i=2  27  71  412  81  59  14  273  87  
i=3  27  71  81  412  59  14  273  87  
i=4  27  59  71  81  412  14  273  87  
i=5  14  27  59  71  81  412  273  87  
i=6  14  27  59  71  81  273  412  87  
i=7  14  27  59  71  81  87  273  412

Отсортированный массив:  
14   27   59   71   81   87   273   412

Алгоритм с прямыми включениями можно легко улучшить с учетом того, что готовая последовательность, куда надо вставить новый элемент, уже упорядочена. Тогда место включения ищется методом двоичного (бинарного) поиска. Такой улучшенный алгоритм сортировки называется*методом с двоичным включением (методом прямого включения с делением пополам)*. Однако применение этого алгоритма оправдано только тогда, когда число сортируемых элементов достаточно велико.

Среднее число сравнений здесь С = O(nlog2n), а среднее число сдвигов М = О(n2).

Пример

/\* СОРТИРОВКА МЕТОДОМ ПРЯМОГО ВКЛЮЧЕНИЯ С ДЕЛЕНИЕМ ПОПО-ЛАМ \*/  
«include  
define n 8

int A[n]={27,412,71,81,59,14,273,87};  
main()  
{ void BinIns(int A[],int nn);  
int j;  
printf("\n Исходный массив \n");  
for (j=0; j<n; j++)  
printf("\t%d",A[j]);  
printf("\n");  
BinIns(A,n);  
printf("\n Отсортированный массив \n");  
for (j=0; j<n; j++)  
printf("\t%d",A[j]);  
printf("\n");  
getch(); return 0;  
}  
/\* ФУНКЦИЯ СОРТИРОВКИ ПРЯМЫМ ВКЛЮЧЕНИЕМ С ДЕЛЕНИЕМ ПОПО-ЛАМ \*/  
void Binlns(int A[],int nn)  
{ int i,j,x,m,L,R;  
printf("\ Отладочная печать по шагам сортировки ");  
for (i=1; i<nn; i++)  
{ x=A[i]; L=0; R=i; printf("\n %d",i);  
while (L<R)  
{ m=(L+R)/2;  
if (A[m] ≤x)  
L=m+1;  
else  
R=m;  
}  
for (j=i; j≥R; j--)  
A[R]=x;  
/\* Отладочная печать по шагам сортировки \*/  
for (j=0; j<nn;j++)  
printf("\t %d”,A[j]);  
}  
}

Исходный массив 27   412   71   81   59   14   273   87  
Отладочная печать по шагам сортировки:

j=1   27  412  71  81  59  14  273  87  
j=2   27  71  412  81  59  14  273  87  
j=3   27  71  81  412  59  14  273  87  
j=4   27  59  71  81  412  14  273  87  
j=5   14  27  59  71  81  412  273  87  
j=6   14  27  59  71  81  273  412  87  
j=7   14  27  59  71  81  87  273  412  
Отсортированный массив:  
14   27   59   71   81   87   273   412

**2.2. Сортировка методом прямого выбора**

Алгоритм сортировки заключается в следующем:

1. В исходной последовательности из n элементов отыскива¬ется элемент с наименьшим ключом. 2. Он меняется местами с первым элементом. 3. В оставшейся последовательности из (n - 1) элементов отыскивается минимальный элемент и меняется местами со вторым элементом и т.д., пока не останется один, самый большой элемент.

Число сравнений С = (п2 - n), число перестановок: Мmin=(n-1), Мсредн = n(In n + g), где g = 0.577216 — константа Эйлера, Mmax=n2/4+3(n-1).

Отсюда алгоритм с прямым выбором несколько предпочтительней алгоритма прямого включения. Однако если ключи вначале почти упорядочены, то прямое включение будет несколько быстрее. Приведем пример программы с функцией сортировки посредством прямого выбора и результаты выполнения программы с отладочной печатью по шагам сортировки.

Пример.

/\* СОРТИРОВКА ПОСРЕДСТВОМ ПРЯМОГО ВЫБОРА \*/  
#include  
#include  
#define n 8  
int A[n]={27,412,71,81,59,14,273,87};  
main()  
{ void StrSel(int A[],int nn);  
int j;  
printf("\n Исходный массив \n");  
for (j=0; j<n; j++)  
printf("\t%d",A[j]);  
printf("\n");  
StrSel(A,n);  
printf("\n Отсортированный массив \n");  
for (j=0; j<n; j++)  
printf("\t%d",A[j]);  
printf("\n"); getch();  
return 0;  
}  
/\* ФУНКЦИЯ СОРТИРОВКИ ПРЯМЫМ ВЫБОРОМ \*/  
void StrSel(int A[],int nn)  
{ int i,j,x,k;  
for (i=0; i<nn-1; i++)  
{ x=A[i]; k=i; printf("\n %d",i);  
for (j=j+1; j<nn; j++)  
if (A[j]<x)  
{ k=j; x=A[k]; }  
A[k]=A[i]; A[i]=x;  
/\* Отладочная печать по шагам сортировки \*/  
for (j=0; j<nn;j++)  
printf("\t %d”,A[j]);  
}  
}

Исходный массив:  
27   412   71    81   59    14    273   87  
Отладочная печать по шагам сортировки:  
0   14  412  71  81  59  27  273  87  
1   14  27  71  81  59  412  273  87  
2   14  27  59  81  71  412  273  87  
3   14  27  59  71  81  412  273  87  
4   14  27  59  71  81  412  273  87  
5   14  27  59  71  81  87  273  412  
6   14  27  59  71  81  87  273  412

Отсортированный массив:  
14   27   59   71   81   87   273   412.  
Несмотря на то, что после пятого шага последовательность уже отсортирована, сортировка продолжается.

**2.3. Сортировка методом прямого обмена (сортировка методом пузырька)**

Алгоритм прямого обмена основывается на сравнении и смене мест для пары соседних элементов, в результате одного прохода через последовательность по направлению с ее конца к началу самый меньший (легкий) элемент оказывается в начале последовательности (или самый большой элемент в конце последовательности при обратном направлении). При втором проходе следующий меньший элемент сдвинется к началу последовательности и т.д., пока все элементы не будут упорядочены.

Алгоритм можно улучшить с учетом того, что если в очередном проходе не было обмена, то последовательность упорядочена. Число сравнений в прямом обменном алгоритме Cm;n = n, Сmах = (n2-n)/2, число перестановок Мmin= 0, Мсредн = 3(n2- n)/4, Мmах = 3(n2 - n)/2.  
Пример.

/\* СОРТИРОВКА МЕТОДОМ ПУЗЫРЬКА\*/  
#include  
#include  
#define n 8  
int A[n]={27,412,71,81,59,14,273,87};  
/\* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = \*/  
main()  
{ void BblSort(int A[],int nn);  
int j;  
printf("\n Исходный массив \n");  
for (j=0; j<n; j++)  
printf("\t%d",A[j]);  
printf("\n");  
BblSort(A,n);  
printf("\n Отсортированный массив \n");  
for (j=0; j<n; j++)  
printf("\t%d",A[j]);  
printf("\n");  
getch(); return 0;  
}  
/\* ФУНКЦИЯ СОРТИРОВКИ МЕТОДОМ ПУЗЫРЬКА \*/  
void BblSort(int A[],int nn)  
{ int i,j,k,p;  
for (i=0; i<nn—1; i++)  
{ p=0;                               /\* признак отсутствия обменов \*/  
for (j=nn-1;j>i;j++)  
  if (A[j]<A[j-1])  
   {k=A[j];A[j]=A[j-1];A[j-1]=k;  
      p=1;                                          /\* был обмен \*/  
}  
/\* Если обменов не было, то сортировка выполнена \*/  
if (р== 0)  
    return;  
/\* Отладочная печать по шагам сортировки \*/  
printf (“\ni=%d",i);  
for (j=0; j<nn;j++)  
printf("\t %d”,A[j]);  
}  
}

Результаты выполнения программы:  
Исходный массив:  
27   412   71   81   59   14   273   87  
Отладочная печать по шагам сортировки:  
i=0   14  27  412  71  81  59  87  273  
i=1   14  27  59  412  71  81  87  273  
i=2   14  27  59  71  412  81  87  273  
i=3   14  27  59  71  81  412  87  273  
i=4   14  27  59  71  81  87  412  273  
i=5    14  27  59  71  81  87  273  412

Отсортированный массив  
14   27   59   71   81   87   273   412

**2.4. Шейкерная сортировка**

Особенностью пузырьковой сортировки является то, что легкий пузырек всплывает сразу, за один проход от конца последовательности к ее началу, где бы он ни находился, в то же время тяжелый пузырек тонет очень медленно, на одну позицию за один проход. Если бы изменить направление просмотра, то тяжелый элемент опустился бы вниз за один проход. Это свойство использовано в алгоритме шейкерной сортировки, в котором чередуются направления последовательных просмотров. Шейкерная сортировка хороша тогда, когда элементы последовательности почти упорядочены.

Пример.  
/'ШЕЙКЕРНАЯ СОРТИРОВКА \*/  
include  
#include  
#define n 8  
int A[n]={27,412,71,81,59,14,273,87};  
main()  
{ void ShkrSort(int A[],int nn);  
int j;  
printf("\n Исходный массив: \n");  
for (j=0; j<n; j++)  
printf("%d\t",A[j]);  
printf("\n");  
ShkrSort(A,n);  
printf("\n Отсортированный массив :\n");  
for (j=0; j<n; j++)  
printf("%d\t",A[j]);  
printf("\n");  
getch(); return 0;  
}

/\* ФУНКЦИЯ ШЕЙКЕРНОЙ СОРТИРОВКИ \*/  
void ShkrSort(int A[],int nn)  
{ int i,j,k,x,L,R;  
L=1; R=nn-1; k=nn-1;  
do  
{  
for (j=R; j≥L; j--)  
if (A[j-1] > A[j])  
   { x=A[j-1]; A[j-1]=A[j]; A[j]=x; k=j;}  
L=k+1;  
/\* Отладочнаяечать \*/  
printf(“ \nL=%d",L);  
for (i=0; i<nn; i++)  
printf("\t%d",A[i]);  
for (j=L; j≤R; j++)  
if (A[j-1] > А[j])  
{ x=A[j-1]; A[j-1]=A[j]; A[j]=x; k=j;}  
R=k-1;  
/\* Отладочная печать \*/  
printf(" \nR=%d",R);  
for (i=0; i<nn; i++)  
printf("\t%d",A[i]);  
}  
while (L<R); ассив:  
}  
    Исходный массив:  
27    412   71   81   59   14   273   87  
     Отладочная печать  
L=2   14  27  412  71  81  59  87  273  
R=6   14  27  71  81  59  87  273  412  
L=4   14  27  59  71  81  87  273  412  
R=2   14  27  59  71  81  87  273  412

**Быстрые (улучшенные) методы сортировки:**

**Метод Шелла, Сортировка с помощью дерева (пирамиды), Быстрая сортировка Хоара (сортировка разделением)**  
 **3.1. Метод Шелла (Shell Sort)**

* Метод Шелла — это улучшенная версия сортировки методом прямого включения, которая использует **шаги** для изначального разделения массива на несколько частей. Это позволяет быстрее упорядочить элементы, начиная с более крупных шагов и постепенно их уменьшая. Процесс аналогичен вставке, но элементы могут быть удалены на большие расстояния друг от друга, что ускоряет сортировку.
* **Время работы**: Зависит от последовательности шагов. В худшем случае — O(n²), но с хорошими шагами время может быть значительно улучшено, вплоть до O(n log n).

**3.2. Сортировка с помощью дерева (пирамиды) (Heap Sort)**

* Алгоритм сортировки с использованием структуры данных **двусвязного дерева** — пирамиды. Массив преобразуется в **макс-кучу** (или мин-кучу) — структуру данных, где каждый родительский элемент больше (или меньше) своих детей. Затем, начиная с корня дерева, извлекаются максимальные (или минимальные) элементы и переносятся в конец массива, после чего куча восстанавливается.
* **Время работы**: O(n log n) для всех случаев (лучший, средний и худший).

**3.3. Быстрая сортировка Хоара (Quick Sort)**

* Быстрая сортировка (Quick Sort) — это алгоритм с использованием метода **разделяй и властвуй**. Выбирается опорный элемент (например, средний или случайный), и массив делится на две части: элементы меньше опорного и элементы больше опорного. После чего рекурсивно сортируются обе части. Быстрая сортировка имеет возможность выбора опорного элемента, что влияет на её эффективность.
* **Время работы**: O(n log n) в среднем случае. В худшем случае (если опорный элемент выбран плохо) может быть O(n²), но это редкость при случайном выборе опорного элемента.

Все эти алгоритмы быстрее, чем методы сортировки с квадратичной сложностью, особенно для больших массивов.

Все прямые методы сортировки фактически передвигают каждый элемент на всяком элементарном шаге на одну позицию. Поэтому они требуют порядка О(n2) таких шагов. Отсюда следует, что в основу любых улучшений должен быть положен принцип перемещения элементов на каждом шаге на возможно большие расстояния

**3.1. Метод Шелла**

Метод Шелла является улучшением метода сортировки с помощью прямого включения и основан на сортировке посредством включений с уменьшающимися расстояниями. Сначала отдельно группируются и сортируются методом прямых включений элементы, отстоящие друг от друга на некотором расстоянии h1, затем на расстоянии h2 < h1 и т.д., последнее расстояние должно быть равно 1. Таким образом, если для сортировки будет использовано t расстояний, то ht=1, hi+1<hi. Желательно, чтобы расстояния обеспечивали бы взаимодействие различных цепочек как можно чаще.</h

Учитывая этот факт имеет смысл использовать такие последовательности (записаны в обратном порядке): 1, 4, 13, 40, 121, ...,т.е. hk-1 = 3hk+ 1,ht=1, t = [Iog3n]- 1 или 1,3,7, 15,31, ..., т.е. hk-1 = 2hk + 1,ht= 1, t = [Iog2n] - 1. В последнем случае для сортировки n элементов методом Шелла затраты пропорциональны n1,2, что значительно лучше n2, но хуже n\*log2n. Часто используют h1 = n/2, h2 = n/4, ..., и т.д.

На каждом шаге методом включения сортируются группы a1, а1+h, a1+2h,…,a1+kh, 1+kh≤n; a2,a2+h,… . Так как начальный шаг является наибольшим, то число элементов массива в начальной группе является наименьшим, например, если h = n/2, то группируются только два элемента. При втором проходе шаг уменьшается, а число элементов в группе увеличивается, и при h = 1 группа включает всю последовательность целиком.

Рассмотрим процесс сортировки последовательности из 8 целых чисел при h1 = 4: 5, 6, 2, 4, 8, 3, 1, 7.

При h1 = 4 сортируются четыре группы по два элемента в каждой: 5, 8; 6, 3; 2, 1; 4, 7. В результате перестановок 6—3 и 2—1 получаем частично упорядоченную последовательность 5, 3, 1, 4, 8, 6, 2, 7. При h2= 2 сортируются две группы по четыре элемента: 5, 1, 8, 2 и 3, 4, 6, 7. В первой группе после перестановки 5—1 получим 1, 5, 8, 2, затем включение 2 в упорядоченную часть приведет к сдвигам 8 и 5: 1, 2, 5, 8. Поскольку вторая группа упорядочена, то последовательность примет вид: 1, 3, 2, 4, 5, 6, 8, 7. Сортировка с шагом h3 = 1 приводит к окончательному результату: 1, 2, 3, 4, 5, 6, 7, 8.

Рассмотрим пример.  
Пример.

/\* СОРТИРОВКА МЕТОДОМ ШЕЛЛА \*/  
#include  
#define n 15  
int A[n]={12,3,14,15,6,1,7,8,9,5,11,2,13,4,10};  
int main()  
{ void Shell(int A[],int nn);  
int j;  
printf("\n Исходный массив: \n");  
printf("\t");  
for (j=0; j<n;j++)  
printf("%d”,A[j]);  
printf("\n Отладочная печать по шагам сортировки");  
Shell(A,n);  
printf("\n Отсортированный массив :\n");  
for (j=0; j<n;j++)  
printf("%d”,A[j]);  
printf("\n");  
getch(); return 0;  
}  
/\* ФУНКЦИЯ СОРТИРОВКИ МЕТОДОМ ШЕЛЛА\*/  
void Shell(int A[],int nn)  
{ int i,j,k,x;  
k=(nn+1)/2;  
while (k 𕟳)  
{  
for (i=k; i<nn; i++)  
{ if (A[i-k] > А[i])  
{ x=A[i]; j=i-k;  
M: A[j+k]=A[j];  
if (j>k)  
       { if (A[j-k] > x)  
{j=j-k;  
goto M;  
}  
}  
А[j]=х;  
}  
}  
/\* Отладочная печать \*/  
printf(" \nk=%d ",k);  
for (i=0; i<nn;i++)  
printf(" %d",A[i]);  
if (k>2)  
k=(k+1)/2;  
else  
k=k/2;  
}  
}  
Исходный массив:  
12   3   14   15   6    1   7   8    9   5   11  2   13   4   10.  
Отладочная печать по шагам сортировки:  
k = 8   9  3   11   2  6  1  7  8   12  5  14  15  13  4  10  
k = 4    6  1  7  2  9  3  10  8  12  4  11  15  13  5   14  
k = 2   6  1  7   2  9  3  10  4  11  5   12  8   13  15  14  
k = 1   1  2  3   4  5  6  7   8   9  10  11  12  13  14  15  
Отсортированный массив:  
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15.

**3.2. Сортировка с помощью дерева (пирамиды)**

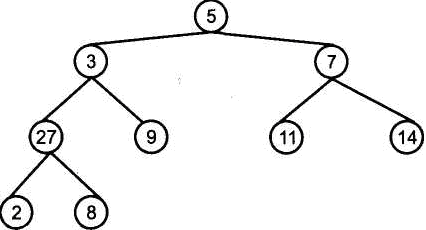
Этот метод является совершенствованием метода сортировки с помощью прямого выбора, основанного на повторяющихся поисках наименьшего элемента среди n элементов, затем среди оставшихся (n - 1) элементов и т.д. Таким образом, за один проход мы находим только один элемент и никакой другой информации не запоминаем. Рассматриваемый ниже алгоритм лишен этого недостатка.

В 1964 г. Вильямсом был предложен алгоритм сортировки с помощью дерева. Дерево имеет специальную структуру, называемую пирамидой. Пирамида состоит из помеченного двоичного дерева высоты h, обладающего следующими тремя свойствами:

1) каждая конечная вершина имеет высоту h или h - 1;  
2) каждая конечная вершина высоты h находится слева от любой конечной вершины высоты h - 1;  
3) ключ любой вершины больше ключа следующей за ней вершины (потомка). Отсюда следует, что ключ корня является наибольшим ключом дерева.

Используя свойства 1) и 2), любой массив из n элементов мысленно можно представить в виде двоичного дерева, которое затем можно преобразовать в пирамиду.

Пусть элементы массива А имеют значения ai, i = 1,2, ..., n: 5, 3, 7, 27, 9, 11, 14, 2, 8. Из них построим дерево следующим образом. Первый элемент (5) образует корень дерева (нулевой уровень), следующие два элемента (3, 7) образуют 1-й уровень, за тем следующие четыре элемента (27, 9, 11, 14) образуют 2-й yровень и т.д. На каждом уровне элементы массива размещаются слева направо. Тогда из исходного массива образуется следующее дерево:



Исходное дерево для построения пирамиды

Из рисунка видно, что для любой вершины дерева аi потомками являются вершины аjи аj+1, j = 2i, и последняя нелистовая вершина образуется из элемента аk, к = n mod 2.

Из полученного дерева пирамида строится с учетом того, ключ любой вершины пирамиды больше ключей потомков (свойство 3 двоичного дерева высоты h). Начиная с самой последней нелистовой вершины i = n mod 2, проверяем, выполняется ли : свойство. Если оно нарушено, то ключи вершины и потомка меняем местами. Затем это же проделываем с вершиной i =i – 1 и т.д., пока не достигнем корня дерева.

В нашем примере i = 9 mod 2 = 4, а4 > а8 и а4> а9 (27 >2 и 27>8) поэтому перестановок не требуется. При i = 3 а3 (7) меньше a6, и a7(11 и 14), a a7 > а6, поэтому а3 и а7 меняем местами. В дальнейшем при i = 2 меняем местами a2 и а4, затем а4 и а9, наконец, при i = 1 последовательно меняем местами а1 и а2, а2 и а5.  
j:   1   2   3   4    5   6   7   8   9  
i=4:5   3   7   27   9   11   14   2   8   перестановок нет  
i=3:5   3   14   27   9   11   7   2   8       a3↔a7, a7 -- лист  
i=2:5   27   14   3   9   11   7   2   8   a2↔a4, a4 -- не лист  
      5   27   14   8   9   11   7   2   3   a4↔a9  
i=1:27   5   14   8   9   11   7   2   3   a1↔a2, a2 -- не лист  
      27   9   14   8   5   11   7   2   3   a2↔a5

В результате получили пирамиду



Таким образом, на первом этапе исходный массив преобразовали в структуру пирамиды, где а1 — наибольший элемент.

На втором этапе, используя эту структуру, упорядочиваем элементы массива. Поменяем местами первый и последний элементы: а1↔ аn, n = 9 (27 ↔ 3). Получаем новое дерево, которое теперь уже не является пирамидой .

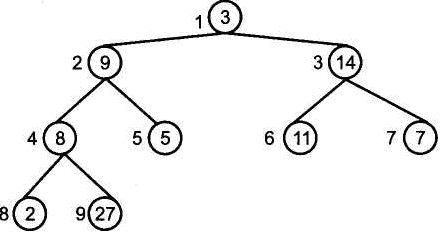


Рис. Перестановка наибольшего элемента из вершины в низ пирамиды

Мысленно исключим из рассмотрения последний элемент дерева a9 = 27, получим новое дерево с (n - 1) элементами. В этом дереве только корневая вершина нарушает свойство 3 пирамиды. Последовательно переставляя ее: а1↔ а3 ↔ a6. получаем новую пирамиду с вершиной а1=14. Поменяв местами а1=14 аn-1 = 2 и исключив аn-1 из рассмотрения, получаем новое дерево с (n - 2) элементами. Повторяем операции по преобразованию, пока не получим упорядоченный по возрастанию массив.

Для упорядочения массива по убыванию необходимо использовать пирамиду, обладающую измененным свойством: ключ любой вершины меньше ключа следующей за ней вершины (потомка), следовательно, ключ корня является наименьшим ключом дерева.

Пример.  
/\* СОРТИРОВКА ПИРАМИДОЙ ПО ВОЗРАСТАНИЮ, ВАРИАНТ 1 \*/  
#include  
#define n 9  
int A[n]={5,3,7,27,9,11,14,2,8};  
main()  
{ void HeapSort(int A[],int nn);  
void Sift(int A[],int L, int R);  
int j;  
printf("\n Сортировка пирамидой, вариант первый");  
printf("\n Исходный массив \n\t");  
for (j=0; j<n;j++)  
printf("%d ",A[j]);  
printf("\n Отладочная печать");  
HeapSort(A,n);  
printf("\n Отсортированный массив \n\t");  
for (j=0; j<nn; j++)  
printf("%d ",A[j]);  
printf("\n");  
getch(); return 0;  
}  
void HeapSort(int A[],int nn)  
{ int L,R,x,i;  
L=nn/2; R=nn-1;  
/\* Построение пирамиды из исходного массива \*/  
while (L>n0)  
{L=L-1;  
Sift(A,L,R);  
/\* Отладочная печать \*/  
printf("\nL=%d: ",L);  
for (i=0;i<nnn;i++)  
printf("%d ",a[i]);  
/\* Сортировка: пирамида=> отсортированный массив \*/  
while (R>0)  
{ x=A[0]; A[0]=A[R]; A[R]=x;  
R--;  
/\* Отладочная печать \*/  
printf("\nR=%d: ",R);  
for (i=0;i<nnn;i++)  
printf("%d ",A[i]);  
Sift(A,L,R);  
}  
}  
/\* ФУНКЦИЯ ПОСТРОЕНИЯ ПИРАМИДЫ \*/  
void Sift(int A[],int L,int R)  
{ int i,j,x,k;  
i=L; j=2\*L+1;  
x=A[L];  
if ((j<nR) && (A[j]<nA[j+1]))  
j++;  
while (j≤nR) && (x<nA[j]))  
{ k=A[i]; A[i]=A[j]; А[j]=к;  
i=j;  
j=2\*j+1;  
if ((j<nR) && (A[j]<nA[j+1]))  
j++;  
}  
}

Сортировка пирамидой, вариант первый.  
Исходный массив:  
5 3 7 27 9 11 14 2 8.  
Отладочная печать:  
L = 3: 5 3 7 27 9 11 14 2 8  
L = 2: 5 3 14 27 9 11 7 2 8  
L = 1: 5 27 14 8 9 11 7 2 3

**3.3. Быстрая сортировка Хоара (сортировка разделением)**

Этот метод является усовершенствованием метода, основанного на обмене. Несмотря на то, что пузырьковая сортировка является наименее эффективной из трех прямых методов сортировки, метод быстрой сортировки оказался самым лучшим известных в настоящее время методов. Он заключается в следующем. Выбираем наугад какой-либо элемент х исходного массива. Будем просматривать слева наш массив до тех пор, пока не встретим элемент аi > х, затем будем просматривать массив справа пока не встретим аj≤ х. Поменяем местами эти два элемента и продолжим наш процесс просмотра и обмена до тех пор, пока оба просмотра не встретятся. В результате исходный массив окажется разбитым на две части, левая часть будет содержать элементы меньше или равные х, а правая часть — элементы больше х. Применив эту процедуру разделения к левой и правой частям массива от точки встречи, получим четыре части и т.д., пока в каждой части окажется только один элемент.

Остается решить вопрос, каким образом выбирать граничный элемент х для разделения. Если при равновероятном распределении элементов массива в качестве разделения каждый раз выбирать медиану, то общее число сравнений будет равно n\*\log2n, а общее число обменов — (n\*log2n)/6. При случайном выборе границы средние затраты увеличатся всего лишь в (2\*ln 2) раза. Однако при крайне неблагоприятном случае, когда каждый раз для разделения выбирается наибольший элемент в обрабатываемой части массива, производительность процедуры будет наихудшая, порядка 0(n2). Обычно в процедурах быстрой сортировки в качестве границы выбирают средний элемент, при этом получаются неплохие показатели производительности.

Рассмотрим пример сортировки посредством рекурсивной функции.  
Пример.  
/'БЫСТРАЯ СОРТИРОВКА ХОАРА \*/  
#include <stdio.h>  
#define n 15  
int A[n]={12,2,13,4,15,6,9,11,3,7,5,10,1,8,14};  
/\* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =\*/  
main()  
{ void QuickSort(int A[],int L,int R);  
int j;  
printf("\n Исходный массив: \n\t\t");  
for (j=0; j<n; j++)  
printf("%d ",A[j]);  
printf("\n Отладочная печать");  
QuickSort(A,0,n-1);  
printf("\n Отсортированный массив :\п");  
for (j=0; j<n; j++)  
printf("%d ",A[j]);  
printf("\n"); getch(); return 0;  
}  
/\* РЕКУРСИВНАЯ ФУНКЦИЯ БЫСТРОЙ СОРТИРОВКИ \*/  
void QuickSort(int A[],int L,int R)  
{ int i,j,k,x,m;  
i=L; j=R;  
x=A[(L+R)/2];  
do {  
while (A[i]<x)  
i++;  
while (x<A[j])  
j--;  
if(i≤j)  
{ k=A[i]; A[i]=A[j]; A[j]=k;  
i++; j--;  
/\* Отладочная печать \*/  
printf("\n i=%d j=%d x=%d: ",i-1,j+1,x);  
for (m=0; m<n; m++)  
printf(" %d”,A[m]);  
}  
}  
while (i<j);  
if (L<j)  
{ printf("\t L=%d j=%d",L,j);  
QuickSort(A,L,j);  
}  
if (i  
{ printf("\t i=%d R=%d",i,R);  
QuickSort(A,i,R);  
}  
}  
Исходный массив:  
12 2 13 4 15 6 9 11 3 7 5 10 1 8 14.  
Отладочная печать:  
I=0 j=13 x=11: 8 2 13 4 15 6 9 11 3 7 5 10 1 12 14  
I=2 j=12 x=11: 8 2 1 4 15 6 9 11 3 7 5 10 13 12 14  
I=4 j=11 x=11: 8 2 1 4 10 6 9 11 3 7 5 15 13 12 14  
I=7 j=10 x=11: 8 2 1 4 10 6 9 5 3 7 11 15 13 12 14 L=0 j=9  
I=4 j=9 x=10: 8 2 1 4 7 6 9 5 3 10 11 15 13 12 14 L=0 j=8  
I=0 j=9 x=7 : 3 2 1 4 7 6 9 5 8 10 11 15 13 12 14  
I=4 j=7 x=7 : 3 2 1 4 5 6 9 7 8 10 11 15 13 12 14 L=0 j=5  
I=0 j=2 x=1 : 1 2 3 4 5 6 9 7 8 10 11 15 13 12 14 L=0 j=1  
I=0 j=0 x=1 : 1 2 3 4 5 6 9 7 8 10 11 15 13 12 14 I=1 R=5  
I=3 j=3 x=4 : 1 2 3 4 5 6 9 7 8 10 11 15 13 12 14 L=1 j=2  
I=1 j=1 x=2 : 1 2 3 4 5 6 9 7 8 10 11 15 13 12 14 I=4 R=5  
I=4 j=4 x=5 : 1 2 3 4 5 6 9 7 8 10 11 15 13 12 14 I=6 R=8  
I=6 j=7 x=7 : 1 2 3 4 5 6 7 9 8 10 11 15 13 12 14 I=7 R=8  
I=7 j=8 x=9 : 1 2 3 4 5 6 7 8 9 10 11 15 13 12 14 I=10 R14  
I=11 j=13 x=13: 1 2 3 4 5 6 7 8 9 10 11 12 13 15 14 L=10 j=12  
I=11 j=11 x=12: 1 2 3 4 5 6 7 8 9 10 11 12 13 15 14 I=12 R14  
I=13 j=14 x=15: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 L=12 j=13  
I=12 j=12 x=13: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Алгоритм быстрой сортировки можно реализовать и в виде нерекурсивной функции. При этом рекурсия заменяется итерацией, естественно, понадобятся дополнительные операции в функции по сохранению нужной информации в стеке. На каждом этапе разделения, как было показано, возникают, в свою очередь, две задачи по разделению. Одну из них мы можем выполнять сразу в очередной операции, а информацию о второй придется запоминать в перечне разделений (в стеке), которые необходимо провести позже. Каждое требование на разделение задается просто левой и правой границами — индексами массива и заносится в стек, организованный в виде массива структур. Требования из стека выполняются в обратном порядке. Максимальный размер стека равен n (размер исходного массива), однако эта граница, как правило, не достигается. Ниже приводится пример использования нерекурсивной функции быстрой сортировки NRQuickSort.  
Пример.  
/'НЕРЕКУРСИВНАЯ ФУНКЦИЯ БЫСТРОЙ СОРТИРОВКИ ХОАРА 7  
#include <stdio.h>  
#define n 15  
int A[n]={12,2,13,4,15,6,9,11,3,7,5,10,1,8,14};  
main()  
{ void NRQuickSort(int A[],int nn);  
int j;  
printf("\n Исходный массив: \n\t\t");  
for (j=0; j<n; j++)  
printf("%d ",A[j]);  
printf(“\n Отладочная печать”);  
for (j=0;j<n;j++)  
printf (“%d”,A[j]);  
printf (“\n”);  
getch(); return 0;  
}  
/\* Нерекурсивная функция быстрой сортировки \*/  
void NRQuickSort(int A[],int nn)  
{  
int I,j,k,x,m,s,L,R;  
#define d 16  
struct stack  
{ int L; int R;} st[d];  
s=1;st[1].L=0;st[1].R=nn-1;  
do {  
L=st[s].L; R=st[s].R;s--;  
do  
{I=L; j=R;  
x=A[(L+R)/2];  
do  
{ while (A[I]<x) I++;  
while (x<a[j])j--;  
if (I≤j)  
{ k=A[I]; A[I]=A[j]; A[j]=k  
I++; j--;  
/\* Отладочная печать \*/  
printf(“\n I=%d j=%d x=%d:”,I-1,j+1,x);  
for (m=0;m<n;m++)  
printf(“%d”,A[m]);  
}  
}  
while (I<j);  
if (I<R)  
{  
s++;st[s].L=I; st[s].R=R;  
printf(“I=%d R=%d s=%d”,I,R,s);  
}  
R=j;  
}  
while (L<R);  
}  
while (a!=R);  
}

**4. Поразрядная («карманная») сортировка**

 **Поразрядная сортировка** (Radix Sort) — это алгоритм сортировки, который сортирует числа или строки по отдельным разрядам (или символам). Он выполняет сортировку начиная с младших разрядов (или символов) и далее до старших, используя стабильные сортировки (например, сортировку подсчётом). Часто используется для сортировки целых чисел или строк с фиксированной длиной.

 **Время работы**: O(n \* k), где n — количество элементов, k — количество разрядов.

Рассмотренные выше методы обеспечивали сортировку на месте, без использования дополнительной памяти для размещения сортируемых массивов. Имеются также методы сортировки, использующие дополнительные массивы. К ним относятся методы поразрядной сортировки и методы сортировки слиянием.

Алгоритмы поразрядной сортировки рассматривают ключи сортируемых элементов как числа с основанием R и работают с отдельными цифрами чисел. При побайтовом рассмотрении ключей, если ключами являются десятичные числа, то R = 10, при сим-ольных ключах R = 128 или R = 256. При побитовом рассмотрении ключей R = 2.

Алгоритм поразрядной сортировки заключается в том, что при очередном проходе элементы исходного массива помещаются в R массивов (раскладываются по R «карманам») в зависимости от значения младшей цифры ключа. При R = 10, ключи начинающиеся с 0, окажутся в первом массиве, начинающие с 1, — во втором массиве и т.д., т.е. потребуется 10 дополнительных массивов «карманов». Далее данные, находящиеся в каждом «кармане», раскладываются по значениям второй цифры и т.д., пока не будут рассмотрены все цифры ключей. Очевидно, что сколько бы проходов ни сделали, общее число сортируемых элементов остается постоянным, а «карманы» будут содержать разное количество элементов. Поэтому одного дополнительного массива того же размера, что и исходный, было бы достаточно, и в нем можно было бы разместить все «карманы». Однако размеры «карманов» определяются только в процессе очередного прохода, поэтом, видимо, наиболее подходящим способом реализации «карманов» будет применение цепных списков, а не массива.

**5. Порядковые статистики**  
  **Порядковая статистика** — это элементы, которые стоят на определённых местах в отсортированном массиве. Например, медиана — это элемент, который находится на позиции, разделяющей отсортированный массив пополам. Задачи, связанные с порядковыми статистиками, могут включать поиск k-го по величине элемента, медианы и другие.

 Метод для нахождения порядковой статистики может быть оптимизирован с помощью алгоритма **поиска медианы** или **быстрого выбора** (Quickselect), который является модификацией быстрой сортировки.

Задача вычисления порядковых статистик заключается в нахождении к-го наименьшего элемента в последовательности из n элементов. Эту задачу иногда называют задачей нахождения к-ой порядковой статистики. При к =1 задача сводится к нахождению наименьшего элемента последовательности, а при к =n наибольшего элемента и решается не более чем за О(n) шагов. Одно из очевидных решений задачи вычисления порядковых s статистик состоит в следующем: упорядочить исходную последовательность в порядке не убывания элементов и взять к-й элемент. При этом, однако, необходимо учитывать требования реальной задачи, в интересах которой вычисляются порядковые статистики.

**Во-первых,** может оказаться, что в упорядоченной последовательности ключи нескольких элементов (к-й, (к + 1)-й, ...)…] имеют одно и то же значение. Тогда необходимо учитывать, должны ли такие элементы следовать друг за другом в том же порядке, как и в исходной последовательности, или это необязательно. В первом случае упорядочение исходной последовательности необходимо выполнить одним из методов устойчивой сортировки, во втором случае можно использовать любой метод сортировки.  
**Во-вторых**, отыскивается ли одна или несколько порядковых статистик. В зависимости от этого может стать выгодным применение того или иного алгоритма.  
**В-третьих**, на выбор алгоритма влияет число элементов n исходной последовательности.

Наконец, если исходная последовательность должна быть сохранена без изменения, то для отыскания порядковых статистик придется создать копию исходной последовательности или массив ключей из элементов исходной последовательности. Использование массива ключей может оказаться выгодным и в том случае, когда элементы исходной последовательности имеют сложную структуру и их перестановки в процессе вычисления порядковых структур требуют значительного времени. Если требование устойчивой сортировки не обязательно, то можно применить рекурсивную процедуру, основанную на быстрой сортировке Хоара (Quicksort), которая обеспечивает самую быструю в среднем сортировку.

Отличие рекурсивной процедуры нахождения к-й порядковой статистики от процедуры быстрой сортировки Хоара заключается в том, что, во-первых, после разбиения на две группы выбирается та группа, в ко¬торой находится к-й элемент, и, во-вторых, в дальнейшем обрабатывается только эта группа, а не обе группы, как это делается в быстрой сортировке Хоара.

Время, затрачиваемое на нахождение к-й порядковой статистики как по одному методу так и по другому, зависит от удачного выбора опорного элемента при разбиении на группы. Время будет минимально, если при каждом проходе группы содержат примерно одинаковое число элементов. Доказано, что в том случае, если каждый проход осуществляется даже на множестве, составляющем 9/10 предыдущего, то время выполнения будет порядка О(n), а в худшем случае — не более О(n2). Линейный метод нахождения порядковых статистик, обеспечивает в самом худшем случае время выполнения порядка О(n). Этот метод основан на поиске «хорошего» опорного элемента и применяется для последовательностей с большим числом элементов.

**Внешняя сортировка (сортировка последовательностей):**

**Прямое слияние, Естественное слияние, Сбалансированное многопутевое слияние, Многофазная сортировка, Формирование и распределение начальных серий**

**6. Внешняя сортировка (сортировка последовательностей)**

* Внешняя сортировка используется, когда данные не помещаются в основную память и хранятся на внешних носителях, таких как диски. Алгоритмы внешней сортировки обычно работают с данными, которые разбиты на блоки и обрабатываются по частям. Основной задачей является минимизация количества операций с внешней памятью.

**6.1. Особенности внешней сортировки**

* Основной особенностью внешней сортировки является необходимость работы с данными, которые не могут быть загружены в память целиком. Алгоритмы внешней сортировки часто используют такие методы, как **сортировка слиянием**, где блоки данных сортируются по частям и затем объединяются.

**6.2. Прямое слияние**

* **Прямое слияние** — это метод внешней сортировки, при котором данные, отсортированные в блоках, сливаются в один отсортированный файл. Алгоритм требует, чтобы блоки данных, прочитанные с внешней памяти, были отсортированы перед слиянием.

**6.3. Естественное слияние**

* **Естественное слияние** — это улучшенная версия прямого слияния, которая использует естественную структуру отсортированных блоков. Когда алгоритм слияния обнаруживает уже отсортированные последовательности в данных, он не выполняет слияние всех блоков, а просто соединяет эти последовательности по мере их обнаружения.

**6.4. Сбалансированное многопутевое слияние**

* Этот метод используется для слияния множества отсортированных последовательностей одновременно. Вместо слияния парных блоков, как в прямом слиянии, сливаются сразу несколько последовательностей, что уменьшает количество этапов слияния и ускоряет процесс.

**6.5. Многофазная сортировка**

* **Многофазная сортировка** — это метод сортировки, который включает несколько фаз обработки данных. Он может сочетать различные подходы, такие как предварительное разбиение данных на части и последующее слияние этих частей с использованием разных методов сортировки в зависимости от доступной памяти и структуры данных.

**6.6. Формирование и распределение начальных серий**

* Внешняя сортировка часто начинается с этапа **формирования начальных серий** — это этап, на котором данные разбиваются на небольшие отсортированные блоки (серии). Эти серии затем распределяются по внешней памяти и используются для дальнейшего слияния, что позволяет эффективно обрабатывать большие объёмы данных.

   **6.1. Особенности внешней сортировки**

Сортировка данных, находящихся в последовательных файлах на ВЗУ, имеет свои особенности. *Во-первых*, поскольку операции обмена с ВЗУ занимают существенную долю времени обработки, стремятся как можно уменьшить число обменов, *во вторых*, из-за ограниченности размеров оперативной памяти, в каждый момент времени сортировке доступна только часть сортируемой последовательности.

Если размеры файла невелики и все записи из него могут быть считаны в оперативную память, то для выполнения сортировки необходимо:

1. Считать все записи из файла в оперативную память.  
2. Произвести сортировку записей файла в оперативной мяти.  
3. Записать отсортированные записи в файл.

Для сортировки файла в оперативной памяти можно использовать один из алгоритмов, рассмотренных выше. Однако, так как в этих алгоритмах для доступа к отдельным элементам массивов использовались индексы, то записи файла необходимо считывать в массив. Элементы массива должны иметь такую же структуру, что и записи файла. В противном случае для доступа к записям файла придется использовать некоторый другой механизм.

Если размеры файла велики и все записи файла одновременно не помещаются в доступной оперативной памяти машины, ТО приходится пользоваться другими методами сортировки. Наиболее важными из них являются методы сортировки с помощью слияния. Слияние означает объединение двух или более последовательностей записей в одну-единственную упорядоченную последовательность с помощью повторяющегося выбора из доступных в данный момент записей. Напомним, что организация последовательностей такова, что в данный момент непосредственно доступен только один-единственный очередной элемент последовательности.

**6.2. Прямое слияние**

*Сортировка, называемая прямым слиянием*, выполняется следующим образом.

1. Последовательность А разбивается на две половины В и С.  
2. Части В и С сливаются, при этом одиночные элементы из В и С образуют упорядоченные пары.  
3. Полученная последовательность А вновь обрабатывается, как указано в двух предыдущих шагах, но сливаются упорядоченные пары. Теперь упорядоченные пары переходят в упорядоченные четверки.  
4. Повторяя предыдущие шаги, сливаем четверки в восьмерки и т.д., пока не будет упорядочена вся последовательность.

Каждый проход (этап) сортировки включает две фазы:

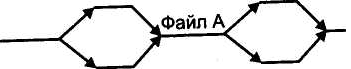


Рис. Двухфазная сортировка прямым слиянием

1. Фаза разделения, когда записи из файла А разбиваются на две равные части (при нечетном числе записей в первой части на 1 больше) и помещаются в файлы В и С соответственно.  
2. Фаза слияния, когда данные из файлов В и С, слитые в упорядоченные пары, четверки, восьмерки и т.д., помещаются в файл А.

Рассмотрим пример.  
Пусть в исходном файле А ключи сортировки будут следующие:  
А: 8 2 13 4 15 6 9 11 3 7 5 10 1 12 14  
На первом этапе получаем:  
В: 8 2 13 4 15 6 9 11  
С: 3 7 5 10 1 12 14  
А: 3 8 2 7 5 13 4 10 1 15 6 12 9 14 11  
Второй этап:  
В: 3 8 2 7 5 13 4 10  
С: 1 15 6 12 9 14 11  
А: 1 3 8 15 2 6 7 12 5 9 13 14 4 10 11  
Третий этап:  
В: 1 3 8 15 2 6 7 12  
С: 5 9 13 14 4 10 11  
А: 1 3 5 8 9 13 14 15 2 4 6 7 10 11 12  
Четвертый этап:  
В: 1 3 5 8 9 13 14 15  
С: 2 4 6 7 10 11 12  
А: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Теперь рассмотрим более детально, какие операции выполняются при каждой фазе при двухфазной сортировке. Фаза разделения:

1. Открыть файл А как входной (для чтения).  
2. Открыть файлы В и С как выходные (для записи).  
3. Считываемые из файла А записи попеременно посылаем в файлы В и С: первая запись в файл В, вторая в С, затем третья в В, четвертая в С и т.д.  
4. Закрыть файлы А, В и С.

Фаза слияния:

1. Открыть файл А как выходной (для записи).  
2. Открыть файлы В и С как входные (для чтения).  
3. Установить размер порции сливаемых записей. Напоминаем, что эти порции рав-ны 1,2,4,... записям для первого и последующих этапов соответственно.  
4. Для каждой порции считываются по одной записи из файлов В и С.  
5. Меньшая из них пересылается в файл А, и считывается очередная запись из того файла, чья запись была переслана в файл А.  
6. Повторяются пункты 4 и 5 до тех пор, пока записи очередной порции одного из файлов не будут исчерпаны.  
7. Оставшиеся записи из порции другого файла пересылаются в файл А.  
8. Пункты 4—7 повторяются до тех пор, пока не будет достигнут конец одного из файлов В или С. Тогда оставшиеся записи из другого файла пересылаются в файл А.  
9. Закрытие файлов А, В и С.

Сортировка завершается тогда, когда длина порции достигнет n. Приведем пример двухфазной сортировки прямым слиянием, записи файла — целые числа (ключи).  
Пример.  
/\* Сортировка файла прямым слиянием, двухфазная сортировка \*/  
#include <stdio.h>  
#include <conio.h>  
#include <stdlib.h>  
int main()  
{int vsort1 (char\* r);  
int n,b; char f[40]="A";  
FILE \*fin,\*fout;  
printf("\n Создание файла А");  
if ((fout=fopen(f,"w"))== NULL)  
{ printf("\n Файл %s не создан",f);  
getch(); return -1;  
}  
n=100;  
printf("\n Исходный файл:\n");  
while (n--)  
{ b=rand();  
/\* заполняем файл случайными целыми числами \*/  
fprintf(fout," %d",b);  
printf("%d\t",b);  
}  
fclose(fout);  
vsort1(f);                                               /\* вызов функции внешней сортировки \*/  
printf("\n Отсортированный файл:\n");  
fin=fopen(f,"r");  
while (fscanf(fin,"%d",&b) !=EOF)  
printf("%d\t",b);  
fclose(fin);  
getch(); return 0;  
}  
/\* ВНЕШНЯЯ ДВУХФАЗНАЯ СОРТИРОВКА ПРЯМЫМ СЛИЯНИЕМ, фаза  
разделения и вызов фазы слияния \*/  
int vsort1 (char \*ff)  
/\* ff — исходный файл, подлежащий сортировке \*/  
{int vsort2(char \*a,int m);                                     /\* функция фазы слияния \*/  
FILE \*A,\*B,\*C;                                                   /\* файловые переменные \*/  
int a;                                                     /\* для чтения из исходного файла \*/  
int nb,nc;                                                     /\* счетчики элементов в формируемых группах \*/  
int р;                                                     /\* р=1 — признак достижения конца исходного файла \*/  
int m=1;                                                     /\* число элементов в группе: 1,2,4,8,...\*/  
int k=0;                                                    /"счетчик числа элементов в исходном файле \*/  
while(1)  
{if ((A=fopen(ff,"r"))== NULL)  
{ printf("\n Файл %s не открывается",ff);  
getch(); return -1; }  
if ((B=fopen("B","w"))== NULL)  
{ printf("\n Файл В1 не открывается");  
getch(); return -1;  
}  
if ((C=fopen("C","w"))== NULL)  
{ printf("\n Файл С1 не открывается");  
getch(); return -1;  
}  
nb=0; nc=0; p=0;  
while (1)  
{while (1)  
{ if(fscanf(A,"%d",&a)== EOF)  
             { p=1; break;}  
             else  
             { if (nb<m)                             /\* формируем группу для файла В \*/  
                { fprintf(B," %d ",a);  
                k++; nb++; continue;}  
               else  
                 {fprintf(C," %d ",a);  
                  nb=0; nc++; break;  
               }  
             }  
}  
if(p)  
break;  
while (1)  
{ if(fscanf(A,"%d",&a)== EOF)  
       { p=1;break;  
}  
else  
     { if (nc<m)      \*="" формируем="" группу="" для="" файла="" С=""   
           {fprintf(C,”%d”,a); k++;  
             nc++; continue;  
              }  
              else  
                  {fprintf(B,”%d”,a); k++;  
                       nc=0; nb++;  
                       break;  
                    }  
          }  
}  
if (p)  
   break;  
}  
fclose(A); fclose(B); fclose(C);  
vsort2(ff,m);          /\* Вызов функции слияния \*/  
if (m>=(k-k/2))          /\* конец сортировки \*/  
{     remove (“B”); remove (“C”);  
      return 0;  
}  
m=m\*2; k=0;  
}  
}  
/\* Фаза слияния \*/</m)     >

vsort2(char \*a,int m)  
/\* a - файл для слияния,  
          m - число элементов в группах сливаемых файлов В и С \*/  
{ FILE \*A,\*B,\*C; /\* файловые переменные \*/  
     int b,c; /\* для считывания данных из файлов В и С \*/  
     int nb,nc;/\*счетчики считанных из групп элементов файлов В и С\*/  
     int rb,rc; /\* коды завершения операции считывания из файлов В и С\*/  
if ((A=fopen(a,"w")) == NULL)  
     { printf("\n Файл %s не открывается",a);  
     getch(); return -1;  
     }  
if ((B=fopen("B","r")) == NULL)  
     { printf("\n Файл B не открывается");  
     getch(); return -1;  
     }  
if ((C=fopen("C","r")) == NULL)  
     { printf("\n Файл C не открывается");  
      getch(); return -1;  
     }  
rb= fscanf(B,"%d",&b); rc=fscanf(C,"%d",&c);  
nb=1;nc=1;  
while(1)  
    { if ((rb> 0) && (rc <= 0)) /\* конец файла С \*/  
                { fprintf(A," %d ",b);  
                while (fscanf(B,"%d",&b) >0)  
                fprintf(A," %d ",b);  
                     fclose(A); fclose(B); fclose(C);  
                return 0;  
                }  
           else  
           { if ((rb ≤ 0) && (rc > 0)) /\* конец файла В \*/  
                    { fprintf(A," %d ",c);  
               while (fscanf(C,"%d",&c)>0)  
                    fprintf(A," %d ",c);  
               fclose(A); fclose(B); fclose(C);  
                    return 0;  
               }  
           else  
                     if ((rb <= 0) && (rc <= 0))  
           /\* конец обоих файлов В и С, случается, если они оба пустые \*/  
                     { fclose(A); fclose(B); fclose(C);  
                          return 0;  
                     }  
                }  
           if (nb≤m && nc≤m)  
               { if (b≤c)  
                     { fprintf(A," %d ",b);  
                         rb=fscanf(B,"%d",&b);  
                         nb++;  
                         continue;  
                     }  
               else  
                    { fprintf(A," %d ",c);  
                         rc=fscanf(C,"%d",&c);  
                    nc++;  
                    continue;  
                    }  
               }  
          if (nb≤m && nc>m)  
               { fprintf(A," %d ",b);  
               rb=fscanf(B,"%d",&b);  
               nb++;  
               continue;  
               }  
           if (nb>m && nc≤m)  
                    { fprintf(A," %d ",c);  
                    rc=fscanf(C,"%d",&c);  
                    nc++;  
                    continue;  
                    }  
                if (nb>m && nc>m)  
                    { nb=1; nc=1;  
                    continue;  
               }  
   }  
}

Для выполнения рассмотренной двухфазной сортировки прямым слиянием нужны 3 файла (3 ленты), поэтому иногда ее называют трехленточным слиянием. Фаза разделения является вспомогательной, в ней записи не переставляются, но она занимает половину всех операций по пересылке записей (чтение из файла А и запись в файлы В и С). Если объединить разделение со слиянием, то от лишних операций можно избавиться. Теперь вместо слияния в одну последовательность результаты слияния сразу же распределяются по двум файлам, которые станут исходными для следующего прохода. Такая сортировка называется однофазной, но она требует наличия 4 файлов, по 2 входных и по 2 выходных при каждом проходе. С учетом этого такую сортировку называют также двухпутевой (балансированной сортировкой. Общее число пересылок при однофазной сортировке М = H\*log2«. Число сравнений особого практического значения не имеет, так как затраты на операции пересылок (обмен данными с файлом) намного превышают затраты на сравнение. Рассмотрим пример.  
/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/  
/\* Сортировка файла прямым слиянием, однофазная сортировка \*/  
/\* фаза слияния-разделения выполняется нерекурсивной функцией \*/  
/\* 3 вспомогательных файла \vsort21.cpp \*/  
#include <stdio.h>  
#include <conio.h>  
#include <stdlib.h>  
/\* ======================================================= \*/  
main()  
{int vsort1(char\* r);  
int n,b; char f[40]="A";  
FILE \*fin,\*fout;  
printf("\n Создание файла А");  
if ((fout=fopen(f,"w")) == NULL)  
     { printf("\n Файл %s не создан",f);  
     getch(); return -1;  
     }  
n=1024;  
printf("\n Исходный файл:\n");  
while (n--)  
     { b=rand(); /\* заполняем файл случайными целыми числами \*/  
     fprintf(fout," %d",b);  
      printf("%d\t",b);  
     }  
fclose(fout);  
vsort1(f); /\* вызов функции внешней сортировки \*/  
printf("\n Отсортированный файл:\n");  
fin=fopen(f,"r");  
while (fscanf(fin,"%d",&b) != EOF)  
     printf("%d\t",b);  
fclose(fin);  
getch(); return 0;  
}  
/\* ===================================================== \*/  
/\* ВНЕШНЯЯ ОДНОФАЗНАЯ СОРТИРОВКА ПРЯМЫМ СЛИЯНИЕМ,  
     фаза начального разделения и вызов фазы слияния-разделения \*/  
int vsort1(char \*ff)  
/\* ff - исходный файл, подлежащий сортировке \*/  
{int vsort2(char \*f,int k); /\*функция фазы слияния-разделения\*/  
FILE \*A,\*B,\*C; /\* файловые переменные \*/  
     /\* файлы "B", "C", "E" в функциях - временные \*/  
int a; /\* для чтения из исходного файла \*/  
int nb,nc; /\* счетчики элементов в формируемых группах \*/  
int p; /\* p=1 - признак достижения конца исходного файла \*/  
     int m=1; /\* число элементов в группе: 1,2,4,8,...\*/  
     int k=0; /\*счетчик числа элементов в исходном файле \*/  
while(1)  
{ if ((A=fopen(ff,"r")) == NULL)  
     { printf("\n Файл %s не открывается",ff);  
     getch(); return -1;  
     }  
  
if ((B=fopen("B","w")) == NULL)  
     { printf("\n Файл B не открывается");  
     getch(); return -1;  
     }  
if ((C=fopen("C","w")) == NULL)  
     { printf("\n Файл C не открывается");  
      getch(); return -1;  
     }  
nb=0; nc=0; p=0;  
while (1)  
    {while (1)  
     { if(fscanf(A,"%d",&a) == EOF)  
          { p=1; break;  
          }  
     else  
           { if (nb<m) /\* формируем группу для файла В \*/  
               { fprintf(B," %d ",a);k++;  
                     nb++; continue;  
               }  
                    else  
                    { fprintf(C," %d ",a); k++;  
                    nb=0; nc++;  
                    break;  
               }  
          }  
     }  
     if (p)  
     break;  
while (1)  
     { if(fscanf(A,"%d",&a) == EOF)  
          { p=1;break;  
           }  
     else  
          { if (nc<m) /\* формируем группу для файла С \*/  
          { fprintf(C," %d ",a); k++;  
               nc++; continue;  
                }  
               else  
               {fprintf(B," %d ",a); k++;  
                    nc=0;nb++;  
                    break;  
               }  
          }  
     }  
     if (p)  
      break;  
}  
     fclose(A); fclose(B); fclose(C);  
     vsort2(ff,k); /\* вызов функции слияния \*/  
     return 0;  
m=m\*2; k=0;  
}  
}  
/\* ====================================================== \*/  
/\* ВНЕШНЯЯ ОДНОФАЗНАЯ СОРТИРОВКА ПРЯМЫМ СЛИЯНИЕМ,  
     фаза слияния-разделения. Функция нерекурсивная \*/  
vsort2(char \*f,int k)  
{ FILE \*B,\*C,\*D,\*E; /\* файловые переменные \*/  
     int b,c; /\* для считывания данных из файлов В и С \*/  
     int nb,nc;/\*счетчики считанных из групп элементов файлов В и С\*/  
     int nd,ne; /\* счетчики сливаемых в группы элементов \*/  
     int rb,rc; /\* коды завершения операции считывания из файлов В и С\*/  
     int sm; /\* число элементов в сливаемой группе: sm=2\*m \*/  
     static m=1; /\* число элементов в исходной группе \*/  
     static int p=1; /\* переключатель \*/  
while(1)  
{if(p%2)  
{  
if ((B=fopen("B","r")) == NULL)  
     { printf("\n Файл B не открывается");  
     getch(); return -1;  
     }  
if ((C=fopen("C","r")) == NULL)  
     { printf("\n Файл C не открывается");  
     getch(); return -1;  
     }  
if ((D=fopen(f,"w")) == NULL)  
     { printf("\n Файл D не открывается");  
      getch(); return -1;  
     }  
if ((E=fopen("E","w")) == NULL)  
     { printf("\n Файл E не открывается");  
     getch(); return -1;  
     }  
}  
else  
{  
if ((B=fopen(f,"r")) == NULL)  
     { printf("\n Файл D не открывается");  
     getch(); return -1;  
     }  
if ((C=fopen("E","r")) == NULL)  
     { printf("\n Файл E не открывается");  
     getch(); return -1;  
     }  
if ((D=fopen("B","w")) == NULL)  
     { printf("\n Файл B не открывается");  
     getch(); return -1;  
     }  
if ((E=fopen("C","w")) == NULL)  
     { printf("\n Файл C не открывается");  
     getch(); return -1;  
     }  
}  
sm=2\*m;  
rb= fscanf(B,"%d",&b); rc=fscanf(C,"%d",&c);  
nb=1;nc=1; nd=0;ne=0;  
while(1)  
{ if ((rb> 0) && (rc ≤ 0)) /\* конец файла С \*/  
          {if (nd<sm) /\* формируем группу для файла D \*/  
               { fprintf(D," %d ",b);  
                nd++;  
               }  
           else  
          { fprintf(E," %d ",b);  
           ne++;  
          }  
           while (fscanf(B,"%d",&b) >0)  
                { if (nd<sm) /\* формируем группу для файла D \*/  
               { fprintf(D," %d ",b);  
                     nd++;  
                }  
               else  
               { fprintf(E," %d ",b);  
                          ne++;  
                     }  
                }  
               fclose(B); fclose(C);  
           break;  
          }  
          else  
          { if ((rb ≤ 0) && (rc > 0)) /\* конец файла В \*/  
               { if (nd<sm) /\* формируем группу для файла D \*/  
               { fprintf(D," %d ",c);  
                nd++;  
               }  
           else  
               { fprintf(E," %d ",c);  
                ne++;  
               }  
          while (fscanf(C,"%d",&c)>0)  
               { if (nd<sm) /\* формируем группу для файла  
D \*/  
                          { fprintf(D," %d ",c);  
                         nd++;  
                     }  
                    else  
                     { fprintf(E," %d ",c);  
                    ne++;  
                }  
               }  
               fclose(B); fclose(C);  
          break;  
          }  
     else  
               if ((rb ≤ 0) && (rc ≤ 0)) /\* конец обоих файлов В и С,  
  
               случается, если они оба пустые \*/  
               { fclose(B); fclose(C);  
                    break;  
               }  
          }  
      if (nb≤m && nc≤m)  
               { if (b≤c)  
               { if (nd==sm && ne==sm)  
                    { nd=0; ne=0;  
                     }  
                    if (nd<sm) /\* формируем группу для файла D \*/  
                    { fprintf(D," %d ",b);  
                          nd++;  
                     }  
                    else  
                    { fprintf(E," %d ",b);  
                          ne++;  
                     }  
                    rb=fscanf(B,"%d",&b);  
                    nb++;  
                    continue;  
                }  
           else  
               { if (nd==sm && ne==sm)  
                    { nd=0; ne=0;  
                     }  
                    if (nd  
                    { fprintf(D," %d ",c);  
                          nd++;  
                     }  
                    else  
                    { fprintf(E," %d ",c);  
                          ne++;  
                    }  
                    rc=fscanf(C,"%d",&c);  
                    nc++;  
                    continue;  
                }  
          }  
      if (nb≤m && nc>m)  
          { if (nd==sm && ne==sm)  
               { nd=0; ne=0;  
                }  
                if (nd<sm) /\* формируем группу для файла D \*/  
                { fprintf(D," %d ",b);  
                     nd++;  
               }  
                    else  
                    { fprintf(E," %d ",b);  
                          ne++;  
                    }  
                rb=fscanf(B,"%d",&b);  
          nb++;  
          continue;  
          }  
     if (nb>m && nc≤m)  
          { if (nd==sm && ne==sm)  
                { nd=0; ne=0;  
                }  
          if (nd<sm) /\* формируем группу для файла D \*/  
                { fprintf(D," %d ",c);  
                     nd++;  
                }  
           else  
                { fprintf(E," %d ",c);  
                     ne++;  
                }  
          rc=fscanf(C,"%d",&c);  
          nc++;  
          continue;  
          }  
      if (nb>m && nc>m)  
          { nb=1; nc=1;  
           continue;  
          }  
} /\* конец цикла \*/  
          fclose(B); fclose(C); fclose(D); fclose(E);  
if (m≥(k-k/2))  
     { if (!(p%2))  
          { remove(f);  
                     rename("B",f);  
          }  
          remove("B"); remove("C");remove("E");  
          return 0;  
     }  
      m=2\*m;  
      p++;  
}  
}

**6.3 Естественное слияние**

Сортировка прямым слиянием не учитывает, были ли данные частично упорядочены или нет, т.е. были ли уже некоторые упорядоченные подпоследовательности или нет. Поэтому размер сливаемых подпоследовательностей на к-м проходе ≤ 2к . В то же время при очередном слиянии две упорядоченные подпоследовательности длиной m и n можно было бы слить в одну длиной m + n. Сортировка, при которой всегда сливаются две очередные упорядоченные подпоследовательности, называется естественным слиянием. Упорядоченные подпоследовательности будем называть сериями.

Представим, что исходный файл разделен на два файла, каждый из которых содержит по n серий (один может содержать (n -1) серий). Тогда при слиянии образуется файл, содержащий также n серий. При каждом проходе число серий уменьша-тся вдвое и общее число пересылок в худшем случае равно nlog2n, а в среднем меньше.

Процесс сортировки заканчивается, если при очередном слиянии в выходной файл будет записана только одна единственная серия. Рассмотрим процесс естественного слияния на примере. Пусть исходный файл А содержит записи с ключами:  
17 31 « 05 59 « 13 41 43 67 «11 23 29 47 « 03 07 71 « 02 19 57 « 37 61.

Как видим, они образуют 7 серий, которые для наглядности разделены знаками «. Разобьем файл А на два файла В и С: первую серию запишем в файл В, вторую в файл С, третью в файл В, четвертую в файл С и т.д., которые затем сольем в файл А:  
Первый проход:  
В: 17 31 « 13 41 43 67 « 03 07 71 « 37 61  
С: 05 59 « 11 23 29 47 «02 19 57  
А: 05 17 31 59 « 11 13 23 29 41 43 47 67 « 02 03 07 19 57 71 «37 61  
Второй проход:  
В: 05 17 31 59 «02 03 07 19 57 71  
С: 11 13 23 29 41 43 47 67 « 37 61  
А: 05 11 13 17 23 29 31 41 43 47 59 67 « 02 03 07 19 37 57 61 71  
После третьего прохода сольются две последовательности и получим:  
А: 02 03 05 07 11 13 17 19 23 29 31 37 41 43 47 57 59 61 67 71

Алгоритм естественного слияния выглядит так:

1. Открыть файл А как входной (для чтения).  
2. Открыть файлы В и С как выходные (для записи).  
3. Считывать записи и посылать их в файл В до тех пор, пока очередная запись больше предыдущей (ri> ri-1 ), то есть до конца серии.  
4. Запись ri ≤ ri-1 записать в файл С. Считывать записи и посылать их в С до тех пор, пока ri> ri-1, то есть до конца следующей серии.  
5. Выполнять пункты 3 и 4, пока не будет достигнут конец файла А.  
6. Выполнить фазу слияния файлов В и С в файл А.  
7. Пункты 1—5 повторять до тех пор, пока в результате слияния в выходной файл не будет записана только одна-единственная серия.

В процессе сортировки может возникнуть такая ситуация, несколько серий из файла А при разделении образуют в выходном файле В или С одну серию, в таком случае число проходов уменьшается.

Пусть в файле А находятся 6 серий:  
1 ... 29 « 21 ...47 « 31 ... 55 » 49 ... 81 « 75 ... 80 « 3 ... 18.  
При разделении в файлы В и С попадут серии в следующем порядке:  
В: 1 ... 29 «31 ... 55 «75 ... 80  
С: 21 ... 47 « 49 ... 81 « 3 ...18

В файле В три серии образовали одну серию, а в файле С первые две серии также объединились в одну. При очёредном слиянии образуется всего две серии: 1 ...81 « 3 ...18, которые сольются в одну при очередном проходе.

Если использовать четыре файла, то фазы слияния и разделения можно объединить в одну фазу. Для этого достаточно полученные при слиянии файлов В и С серии поочередно направлять в файлы D и Е, и, наоборот, если сливаются серии из файлов D и Е, полученные серии поочередно направлять в В и С.

Сортировка заканчивается, если при очередной фазе слияния - разделения образуется только одна серия и один из выходных файлов останется пустым, т.е. туда не будет записана ни одна серия.

**6.4. Сбалансированное многопутевое слияние**

Затраты на сортировку последовательностей пропорциональны числу проходов, так как, при каждом проходе пересылаются все данные, в двухфазных сортировках дважды, в однофазных — один раз. Один из способов сократить число пересылок — распределять серии более чем в два файла. Если в исходном файле имеется n серий и они поровну распределяются в N файлов, то первый проход даст n/N серий, второй проход — n/N2, третий — n/N3 и т.д. Отсюда общее число проходов, необходимых для сортировки n элементов с помощью N-путевого слияния, равно к = logNn, а число пересылок при объединении фаз слияния и разделения в самом худшем случае будет М = nlogNn. При каждом проходе будут участвовать N входных и N выходных файлов, в которые по очереди распределяются последовательные серии. Таким образом, получается однофазное многопутевое сбалансированное слияние - разделение.

Алгоритм многопутевого слияния, естественно, сложнее рассмотренных выше. Из N выходных файлов образуется кольцевая очередь. Первоначально n серий данных из исходного файла распределяются поочередно в N выходных файлов, по n/N серий в каждый файл. Первая серия направляется в первый файл, вторая — во второй и т.д., затем по кольцу опять в первый, второй и т.д. до конца исходного файла. Выходные файлы становятся входными, а остальные N файлов образуют кольцевую очередь выходных файлов.

Так как фазы слияния и разделения объединены в одну то по мере слияния N очередных входных серий (по одной из каждого входного файла) в одну выходную она направляется в очередной выходной файл. Таким образом, управление пересылкой слитых серий в выходные файлы (полуфаза разделения) осуществляется с использованием кольцевой очереди и не вызывает никаких затруднений. Количество файлов в очереди равно N и остается постоянным.

Управление слиянием N очередных входных серий в одну выходную серию представляет собой более сложный процесс. Входным файлам присвоим номера i = 1, 2, ..., N. В каждом i-м файле имеется последовательность серий j = 1, 2,…. В процессе слияния текущая j-я серия каждого i-го файла сливается в одну выходную серию. Входной файл может находиться в одном из трех состояний:

1) файл активен, если не исчерпаны записи текущей j-й серии файла;  
2) файл временно не активен, если исчерпаны записи текущей j-серии, но не достигнут конец файла. Такой файл не участвует в процессе слияния очередной серии, но активизируется при слиянии следующей серии;  
3) файл не активен, если достигнут конец файла, т.е. исчерпаны все серии. В текущем проходе такой файл больше не используется.

Для определения конца текущей серии или конца файла может потребоваться опережающее чтение записи или ведение специальной таблицы.

При формировании очередной выходной серии вначале из каждого активного файла считывается одна запись текущей серии. Среди них выбирается i-я запись с наименьшим ключом и пересылается в активный выходной файл. Активным является тот выходной файл из кольцевой очереди, куда направляются записи формируемой серии. Файл остается активным до конца формирования текущей выходной серии.

Из i-го файла, чья запись была переслана в выходной файл, считывается очередная запись, если только этот входной файл остается активным. Опять выбирается запись с наименьшим ключом и направляется в активный выходной файл. Такой процесс слияния продолжается до исчерпания записей из текущих серий всех N файлов. Затем активизируется очередной выходной файл и начинается процесс формирования следующей выходной серии.

Очередной проход заканчивается, когда будет достигнут конец всех входных файлов. Выходные файлы становятся входными, а входные файлы — выходными, и начинается очередной проход сортировки.

Процесс сортировки заканчивается, когда при очередном проходе будет сформирована одна-единственная серия.

**6.5. Многофазная сортировка**

Метод многофазной сортировки основан на том, что имеется несколько входных файлов с разным числом серий и только один выходной файл. При каждом проходе серии из входных файлов сливаются до тех пор, пока входной файл с наименьшим числом серий не будет исчерпан. Тогда освободившийся файл становится выходным, начинается следующий проход, серии из всех остальных файлов сливаются в этот выходной файл. Процесс сортировки завершается, когда все серии будут объединены в одну серию.  
Рассмотрим пример использования трех файлов fl, f2 и f3. Пусть вначале файл fl содержит 13 серий, файл f2 — 8 серий, а f3 — выходной файл. Рассмотрим сам процесс coртировки.

Сначала 8 серий из файлов f 1 и f2 сливаются и посылаются в f3. Файл f2 оказывается свободным, при очередном проходе туда посылаются 5 серий, полученных от слияния записей из f1 и f3, и т.д. до тех пор, пока все серии не сольются в одну в файле f1.  
  
Файлы  
                                         f1     f2     f3  
Исходное состояние      13     8      0  
Первый проход               5      0     8  
Второй проход               0      5      3  
Третий проход               3      2     0  
Четвертый проход         1     0      2  
Пятый проход               0     1      1  
Шестой проход             1      0      0  
  
Сортировка завершена, отсортированные записи в файле f1.

Многофазная сортировка более эффективна, чем сбалансированная многопутевая, так как при одном и том же общем количестве файлов число сливаемых файлов больше.

Число проходов при многофазной сортировке будет зависеть от начального распределения серий по входным файлам. Доказано на практике, что для хорошей работы многофазной сортировки с тремя файлами необходимо, чтобы числа начальных серий в двух выходных файлах были двумя соседними числами Фибоначчи.

**6.6. Формирование и распределение начальных серий**

Как мы уже знаем, такие методы сортировки, как естественное слияние, многопутевое сбалансированное слияние и многофазная сортировка, имеют дело с сериями. Эти методы практически не требуют никакой дополнительной оперативной памяти, кроме буферов для файлов и самой программы. В то же время при решении задач всегда имеются свободные области оперативной памяти, которые можно ис-пользовать, чтобы улучшить процесс сортировки. Этого можно добиться объединением методов сортировки для массивов (внутренней сортировки) и для последовательностей (внешней сортировки). Очевидно, что если на этапе распределения начальных серий по файлам длины серий будут большими, то число проходов для сортировки исходного файла значительно сократится.

Итак, на начальном этапе сортировки считываем некоторое количество записей исходного массива в доступную область опе¬ративной памяти и, используя подходящую сортировку массивов, получаем серию, длина которой L определяется размером доступной памяти. Полученную серию записываем в один из выходных файлов. Аналогичным образом создаем и распределяем по файлам последующие серии, пока не достигнем конца исходного файла.

Формирование начальной серии из массива записей, считанных в оперативную память, можно выполнить сортировкой с использованием пирамиды, имеющей специальную структуру. Она отличается от пирамиды, используемой для сортировки массива, тем, что ключ каждой вершины меньше ключей своих потомков (свойство №3 двоичного дерева-пирамиды высоты h). Это приводит к тому, что из пирамиды на каждом шаге выталкивается запись с наименьшим ключом. Выталкиваемая запись сразу же пересылается в выходной файл и не записывается в конец массива.

Таким образом, формирование серии происходит за три этапа: чтение из файла и заполнение массива; построение пирамиды; сортировка элементов с выталкиванием их из массива в файл. В результате массив оказывается свободным и происходит формирование следующей серии. Длины всех серий одинаковы и равны длине массива, последняя серия может иметь меньшую длину.

Этот способ может с успехом применяться и для сортировки тех файлов, все записи которой умещаются в доступную область.

Имеется более изощренный способ формирования серии посредством сортировки пирамидой, когда пирамида использует как туннель. Идея заключается в следующем. Сначала массив заполняется L записями из исходного файла и строится пирамида. На этапе сортировки после выталкивания наименьшего элемента в выходной файл из входного файла считывается очередной элемент. Если его ключ больше ключа последнего вытесненного элемента, то новый элемент включается в пирамиду и осуществляется сортировка среди L элементов. Если же ключ нового элемента меньше ключа вытесненного элемента, то последний элемент массива пересылается в начало массива (вершину пирамиды), а на его место помещается новый элемент. Длина сортируемого массива (размер пирамиды) мысленно уменьшается на единицу (L = L - 1), и в сортировке теперь участвует только (L - 1) элементов.

Такой процесс продолжается до тех пор, пока размер сортируемого массива не сократится до нуля (L = 0). Формирование очередной серии на этом заканчивается, а весь массив оказывается заполненным не отсортированными элементами. Начинается формирование новой серии: в массиве строится пирамида, ocуществляется сортировка элементов и их выталкивание в файл. Длина серии при использовании туннеля больше или равна длине массива L, при случайном распределении элементов в исходном файле средняя длина серий равна 2L.

Для формирования начальной серии можно предложить также другой, с нашей точки зрения, более простой и эффективный способ. В оперативной памяти из считываемых записей входного файла строится древовидная таблица поиска. Записи в таблице размещаются в порядке их поступления из файла, никаких перестановок записей в таблице не производится. Дерево же организуется за счет образования соот-ветствующих ссылок. Левый смешанный обход дерева дает возможность извлекать записи из дерева в порядке возрастания ключей.

Извлеченные записи пересылаются в очередной выходной файл. В таблицу считывается очередная порция записей файла для формирования следующей серии.

Начальное распределение серий по файлам в случае естественного слияния и многопутевого сбалансированного слияния не вызывает затруднений, так как все файлы имеют одинаковое число серий.

Многофазовая сортировка требует оптимального распределения по файлам. Для этого, во-первых, надо знать число серий в исходном файле, а оно становится известным только в процессе формирования серий. Во-вторых, числа серий в файлах при оптимальном распределении по (N - 1) файлам должны быть числами Фибоначчи порядка (N - 2), что возможно только при определенных значениях общего числа серий. Недостающее число серий можно дополнить пустыми сериями, причем пустые серии нужно как можно более равномерно распределить по (N- 1) файлам.

**1. Виды таблиц**

**Таблица** — одна из важнейших структур данных, применяемых в задачах как системного, так и прикладного программирования. Таблица состоит из совокупности элементов, снабженных отличительными признаками — ключами. Ключи используются для доступа к элементам таблицы.

Каждый элемент таблицы обычно представляется записью, которая содержит ряд информационных полей. Одно или несколько полей играют особую роль и используются при поиске элемента. Их называют ключами записи. Элементы добавляются и выбираются из таблицы по ключам.

Таблицы можно классифицировать по различным признакам. По месту хранения различают *внутренние*таблицы, размещаемые в оперативной памяти ЭВМ, и *внешние*таблицы (файлы), размещаемые на ВЗУ. По отношениям связи между элементами различают*линейные и нелинейные*таблицы. Таблица, отражающая отношение соседства между элементами, называется линейной. Все остальные таблицы являются нелинейными. К ним можно отнести древовидные таблицы, таблицы с вычисляемыми входами.

Различают также *упорядоченные и неупорядоченные* таблицы. В упорядоченной таблице записи расставлены в определенном последовательном порядке в соответствии с некоторым критерием упорядочения. Алгоритмы поиска в упорядоченных и неупорядоченных таблицах значительно различаются по времени их исполнения. Собственно говоря, упорядочение (сортировка) таблиц производится в основном для облегчения и убыстрения поиска данных.

Широкое применение таблицы находят как: управляющие и информационные таблицы в операционных системах и трансляторах

**2. Условия поиска**

Пусть имеется множество А элементов аi, которые хранятся в таблице. Поиском будем называть процедуру выделения из множества А элементов подмножества В, содержащего один, несколько или все элементы, признаки которых удовлетворяют некоторому заранее поставленному условию, и только эти элементы.

Условие, по которому отбираются элементы из множества А в подмножество В, будем называть условиями поиска. Существует множество типов условий поиска, рассмотрим некоторые из них.

**Поиск по совпадению**. Задан некоторый ключ, требуется выделить один или все элементы (записи), признаки которых совпадают с заданным ключом. Иначе говоря, необходимо найти все записи, для которых pi = q, где рi — ключ i-й записи, a q — значение поискового признака (заданного ключа, или ключа поиска).

**Поиск по интервалу**. В условии поиска задан некоторый числовой интервал а < - q ≤ b, требуется выделить одну или все записи, значения признаков которых заключены между границами интервала. Например, найти всех студентов, имеющих оценки 3, 4 и 5 по некоторому предмету: 3 ≤ q ≤5.

**Поиск по близости**. Найти такую запись, для которой abs(pi -q) минимально среди всех записей, т.е. запись с ближайшим к q значением pi. Сюда же относится поиск записи с максимальным или минимальным значением ключа.

**Поиск по значению функции**. Поиск осуществляется с использованием некоторой функции от значений признаков записей. Выделяются такие записи, у которых значение функции равно заданному ключу или попадает в заданный интервал, или удовлетворяет условиям близости. Например, по таблице успеваемости найти всех студентов, средний балл которых не ниже заданного значения.

**Поиск по текстовому значению (семантике)**. Осуществляется поиск по признакам с нечисловыми значениями, такими, как строки, коды иерархической классификации, дескрипторы в тезаурусах и т.п. Поиск и здесь может осуществляться по совпадению, близости или интервалу. Могут задаваться также общие ключи, например найти сотрудников, фамилии которых начинаются с «Иван» (Иванов, Иванченко, Ивановский). Формулировка критерия, позволяющего судить о смысловом совпадении, интервале или близости, может вылиться в сложную задачу.

**Поиск по нескольким условиям**. Используется некоторое логическое выражение над несколькими логическими значениями, например, найти всех абитуриентов, проживающих в Москве и имеющих средний балл не менее 4.3, или найти всех служащих имеющих специальность «механик» и возраст более 45 лет.

Все рассмотренные выше условия поиска, кроме условия по¬иска по близости, связаны с проверкой некоторых внутренних характеристик каждого отдельного эле-мента, т.е. выполнение или невыполнение условия поиска может быть проверено в каждом элементе в отдельности и не зависит от наличия других элемен¬тов. Выпол-нение условия поиска по близости не может быть про¬верено в отдельном элементе без учета других объектов, т.е. здесь учитываются некоторые отношения между элементами.

Алгоритмы поиска и время поиска во многом зависят от способа организации данных. По содержательному смыслу данные можно разбить на два вида: фактографические данные и документальные данные.

*Фактографические данные об объекте*(элементе множества) включают: объект (фамилия сотрудника, код изделия, название отдела и т.п.); значения признаков (свойств, характеристик) объекта (Иванов, 8к88, бухгалтерия).

Сами признаки (названия признаков) в таблицу не включаются, они определяются структурой записи, а их смысловое содержание (семантика) известно программе.

Фактографические данные имеют полностью формализованный вид, удобный для обработки. Они, как правило, представляются в виде таблицы или логически взаимосвязанных таблиц. Помимо фактографических сведений каждый объект может характеризоваться *документальными данными*, например, автобиография для объекта «сотрудник», учебник по дискретной математике для объекта «математика», справочные пособия и т.п.

Документальные данные могут иметь вид неформализованной или полуформали-зованной информации. К первым можно отнести книги, периодические издания, справочники и т.п.. К полуформализованной можно отнести заказы на билеты, различные бланки и анкеты.

Мы ограничиваемся рассмотрением таблиц, хранящих фактографические данные.

**3. Линейные таблицы**

В **линейной таблице** элементы располагаются друг за другом, т. е. для каждого элемента таблицы существуют отношения порядка. Линейные таблицы в оперативной памяти отображаются в массивы или линейные связанные списки. Здесь рассматриваются таблицы, представленные в программе только как одномерные массивы и хранящиеся в виде вектора.

Записи таблицы могут занимать или все элементы массива, тогда таблица считается заполненной полностью, или только часть элементов массива. Так как размеры массивов фиксированы, то добавлять записи можно только до заполнения таблицы, после чего можно только обновлять записи. Удалять записи можно двумя способами: все записи, размещенные за удаляемой записью, сдвигаются на одну позицию к началу массива, в конце массива образуется свободный элемент либо элемент массива с удаленной записью помечается как свободный.

Естественно, поиск элементов в таблице осуществляется для получения и использования данных из элемента. В каждом конкретном случае эти данные используются по разному. Поэтому нас интересуют только сам поиск и необходимые для поиска данные. Таким образом, нам в каждом элементе достаточно рассмаривать только ключевые данные.

**3.1. Поиск в неупорядоченных таблицах**

Поиск одного элемента в неупорядоченной таблице по заданному условию осуществляется последовательным просмотром элементов или до нахождения искомого элемента, т.е. до выполнения условия поиска, или до конца таблицы, если искомый элемент не найден. Возвращаемыми значениями являются адрес (индекс) элемента или значение элемента либо признак отсутствия элемента.

При поиске всех элементов, отвечающих заданному условию поиска, осуществляется последовательный просмотр до конца таблицы, выбираются и обрабатываются все элементы, удовлетворяющие условию поиска. Оценка времени поиска. Минимальное время (длина) поиска, когда элемент нахо-дится в начале таблицы, O(1), максимальное время О(n), n — число элементов таблицы. Среднее время поиска при равновероятном распределении элементов в таблице О(n/2) т.е. О(n).

Рассмотрим пример, когда элемент таблицы содержит только целочисленный ключ, а поля данных отсутствуют. Таблица задана в виде одномерного массива целых чисел (ключей записей). Функция поиска возвращает индекс элемента либо -1 в случае отсутствия искомого элемента.  
Пример.  
/\* ПОИСК ЭЛЕМЕНТА ПО СОВПАДЕНИЮ \*/  
int poiski (int A[],int n, int k)  
{ int i = 0;  
while (A[i] != k && i < n)  
i++;  
return (A[i] == k) ? i : -1; }  
}

По этому алгоритму на каждом шаге поиска выполняются два сравнения. Для улучшения алгоритма можно использовать заграждающий элемент. В этом случае последняя запись таблицы запоминается, а после завершения поиска восстанавливается в таблице. В последний элемент массива заносится ключ поиска, и образуется так называемый заграждающий элемент.

Теперь на каждом шаге поиска осуществляется только одно сравнение, а сам поиск продолжается до нахождения элемента с заданным ключом. Если искомого элемента в исходной таблице не было, поиск закончится на заграждающем элементе. Таким образом, использование заграждающего элемента в случае числовых ключей существенно сокращает число сравнений. Если ключом поиска является символьная строка, то использование заграждающего элемента вряд ли оправдано.

Приведем пример использования заграждающего элемента.  
Пример.  
/\* ПОИСК ПО СОВПАДЕНИЮ КЛЮЧА — ЗАГРАЖДАЮЩИЙ ЭЛЕМЕНТ\* /  
#include <stdio.h>  
#include <conio.h>  
#define size 10  
int A[size] = {1,7,3,4,8,5,2,6,0,9};  
mainQ  
{ int poisk2(int A[],int n,int key);  
int key ;  
int ind;  
printf("\n Поиск элемента по совпадению");  
printf("\n Введите ключ поиска =>");  
flushall();  
scanf("%d",&key);  
ind= poisk2(A,size,key);  
if (ind == -1)  
printf("\n Элемент %d не найден \n",key);  
else  
printf("\n Элемент %d имеет индекс %d\n",key,ind);  
getch(); return 0;  
}

/\* ФУНКЦИЯ ПОИСКА ЭЛЕМЕНТА ПО СОВПАДЕНИЮ С ЗАГРАЖДАЮЩИМ ЭЛЕМЕНТОМ \*/  
int poisk2(int A[],int n, int key)  
{ int i = 0,r;  
r=A[n-1];  
A[n-1]=key;          /\* заграждающий элемент \*/  
while (A[i] != key )  
i++;  
A[n-1]=r;                /\* восстановили последний элемент \*/  
if ((i==n-1) && (r!=key))  
return -1;  
else return i;  
}

**3.2. Поиск в упорядоченных таблицах**

Таблица может быть упорядочена по возрастанию или убыванию значения ключа или по частоте обращения к записям таблицы. В первом случае для поиска записей обычно применяют Двоичный (бинарный) поиск, а во втором случае применяют последовательный просмотр. Упорядочение по частоте обращения к записям требует больших усилий и применяется редко. Дело в том, что определение частот обращения возможно лишь в статических по времени таблицах при длительной работе с ними и сборе соответствующей информации, например в таблицах ключевых слов трансляторов. Поэтому, говоря об упорядоченных таблицах, имеют в виду упорядочение по возрастанию или убыванию ключа, чаще всего по возрастанию.

Поиск в упорядоченных таблицах можно организовать по разному.  
**Последовательный поиск**. Выполняется так же, как и в неупорядоченной таблице, но поиск прекращается, либо когда найдена запись {pi = q), либо когда ключ очередной записи станет больше заданного ключа (рi> q), либо по достижении конца таблицы. Последние два события соответствуют отсутствию записи с заданным ключом.  
**Двоичный (бинарный, логарифмический ) поиск в упорядоченной таблице**. Является самым экономичным. Он состоит в последовательном делении таблицы пополам и определении, в какой из двух частей находится искомая запись. Последующему делению каждый раз подвергается часть, содержащая искомый ключ. Поскольку таблица упорядочена по возрастанию ключа, то чтобы определить, в какой части находится требуемое значение ключа, достаточно сравнить значение ключа в точке деления с искомым.

Длина двоичного поиска для п элементов имеет порядок O(log2n). Например, для таблицы из 1000 элементов средняя длина последовательного просмотра равна 500, в то время как длина двоичного поиска — только 10, а для 1000000 — лишь 20.

Иногда используются частично упорядоченные таблицы. Таблица состоит из разделов, соответствующих различным интервалам ключа. Разделы упорядочены, а внутри разделов записи не упорядочены. Для поиска записей используется комбинированный метод: раздел отыскивается путем двоичного поиска, а внутри раздела применяют последовательный поиск.

Рассмотрим программу с двумя вариантами функции двоичного поиска. Варианты отличаются только местом проверки того, не является ли элемент искомым в точке деления. В первом варианте такая проверка осуществляется в теле цикла, а во втором варианте — в управляющем операторе цикла while.  
Пример.  
/\* БИНАРНЫЙ ПОИСК В УПОРЯДОЧЕННОЙ ТАБЛИЦЕ 7  
#include <stdio.h>  
#define m 20  
int bisearch(int A[],int n,int key);  
main()  
{ int i.key, A[m];  
printf(" \nБинарный поиск в упорядоченной таблице ");  
printf("\n Введите %d целых чисел в возрастающем порядке»,m);  
for (i=0; i  
scanf("%d",&A[i]);  
printf("\n Введите значение искомого ключа =>");  
scanf("%d",&key);  
i = bisearch1(A,m,key);  
if(i== -1)  
printf(" Ключ %d не найден ",key);  
else  
printf(" %d-u элемент имеет ключ = %d \n",i,A[i]);  
getch(); return 0;  
}  
/\* ФУНКЦИЯ БИНАРНОГО ПОИСКА. Вариант 1 \*/  
int bisearch1(int A[],int n,int key)  
{int li,rj,k;  
li=0; rj=n-1;  
while (li ≤ rj )  
{ к = (li+rj)/2;  
if (key > A[k])  
li = k+1;  
else  
if (key < A[k])  
rj = k-1;  
else return k;  
}  
return -1;  
}  
/\* ФУНКЦИЯ БИНАРНОГО ПОИСКА Вариант 2 \*/  
int bisearch2(int A[],int n.int key)  
{ int li,rj,k;  
li=0; rj=n-1; к = li;  
while ( li ≤ rj && A[k] != key )  
{ к = (li+rj)/2;  
if (key > A[k])  
li = k+1;  
else  
if (key < A[k])  
rj = k-1;  
}  
return (A[k] == key) ? к : -1;

**3.3. Некоторые рекомендации по работе с линейными таблицами**

При решении задач, связанных с обработкой таблиц, число элементов таблицы обычно заранее неизвестно. Кроме того, свойство массовости алгоритма требует возможности применения программы для таблиц с произвольным числом элементов, таблицы же могут содержать многие сотни и тысячи записей. Естественно, при каждом применении программы создавать такие большие таблицы нет необходимости, они должны быть сохранены в файлах до следующей обработки. С течением времени может возникнуть необходимость добавления и удаления отдельных элементов таблицы.

Такие потребности могут быть удовлетворены следующим образом. Первоначально таблица создается не в оперативной памяти, а в файле на ВЗУ. Для обработки таблица считывается в динамически полученную векторную память в виде одномерного массива. Возникает проблема определения размера запрашиваемой динамической памяти, т.е. определения числа элементов таблицы. Казалось бы, что для этого достаточно извлечь информацию о длине файла, хранимую системой управления файлами операционной системы:  
FILE •fin;  
long len;      /\* размер файла в байтах \*/  
char name[ ];    /\* для имени файла \*/  
fin = fopen(name, »r»);  
len = filelength(fileno(fin));

Однако это допустимо только тогда, когда записи таблицы хранятся в файле в двоичном виде и, следовательно, размеры таблицы в оперативной памяти и на ВЗУ совпадают. Если же данные записей хранятся в файле в форматированном виде, то эти размеры не совпадают, так как размер записи в оперативной памяти определяется структурой записи, а размер форматированной записи на ВЗУ будет другим. Действительно, число int a = 31570 занимает в ОП 2 байта, а на ВЗУ — все 5 и то без учета пробелов между отдельными данными. Или char name[20] =»qq.txt» в ОП занимает все 20 байт, а на ВЗУ — только действительную длину строки. Таким образом, при хранении записей таблицы в файле в форматированном виде необходимо знать число записей т. Тогда для размещения таблицы в ОП получают динамическую память, размер которой будет определяться как произведение числа элементов т на длину каждого элемента. Например:  
struct Rec \*T1;  
Т1 = (Rec\*) calloc(m, sizeof(Rec)); или  
Т1 = (Rec\*) malloc(m\*sizeof(Rec));

В приведенном ниже примере в начале файла находится число записей таблицы m, затем следуют т записей таблицы. Это достигается следующим образом. После открытия файла в режиме записи "w+" fout=fopen(name,"w") сразу же записываем в файл число m= 0, резервируя место для записи действительной длины. При этом необходимо учитывать возможности расширения поля при больших значениях т.

Далее ведем подсчет числа i выводимых в файл записей, уста¬навливаем указатель файла в его начало и записываем в файл число i:  
/\* fpostj POS; \*/  
fseek(fout, 0L, 0);  
/\* или fgetpos(fout, &POS); \*/  
fprintf(fout, " %5d ", /);     /\* 5 — размер поля для числа записей i\*/  
fclose(fout);

Теперь для чтения таблицы в оперативную память открываем файл, считываем число записей таблицы и получаем динамическую память. Затем последовательно считываем записи из файла в таблицу.

Расширение таблицы за счет новых элементов осуществляется добавлением записей в конец файла и последующей коррекцией числа записей в начале файла. Удаление записей из таблицы можно выполнить по схеме: чтение таблицы из файла в оперативную память — последовательная запись элементов таблицы в файл, исключая удаляемые записи — коррекция числа элементов в начале файла.

**4. Логически связанные таблицы**

Информация об одних и тех же объектах задачи может храниться в нескольких логически связанных между собой таблицах. Например, в одной таблице хранятся данные об успеваемости студентов, в другой — анкетные данные, в третьей — данные о состоянии здоровья и т.д. При решении некоторых задач может потребоваться совместная обработка данных из нескольких таблиц. В таких случаях между записями одних и тех же объектов в отдельных таблицах должна существовать некая связь. Это не связь по указателям, как в списках. Она может осуществляться либо по одинаковым индексам, когда в таблицах записи об объектах следуют в одном и том же порядке, либо по некоторому общему полю, имеющемуся в связанных таблицах, например по фамилиям.

Рассмотрим пример работы с двумя связанными последовательными таблицами. Первая — таблица анкетных данных, содержит записи с полями: фамилия, учетный номер, семейное положение, рост, вес. Вторая таблица содержит записи об успеваемости студентов с полями: «Учетный номер», «Структуры и алгоритмы обработки данных в ЭВМ», «Операционные системы», «Теория вычислительных процессов и структур», «Метрология программного обеспечения», «Физическая подготовка». Для связи между записями используется поле учетного номера, состоящего из начальной буквы фамилии и порядкового номера по этой букве (А1, А2, Ж1, Ж2, ...). Таблицы создаются и хранятся в файлах подобно тому, как это было в предыдущем примере.

Необходимо выдать списки холостых студентов, имеющих средний балл по успеваемости не ниже 4,5.  
Алгоритм решения задачи заключается в следующем. Считываем таблицы в оперативную память. Последовательно просматриваем записи с анкетными данными и выбираем очередного холостого студента. По его учетному номеру в таблице успеваемости находим соответствующую запись. Вычисляем средний балл по 5 дисциплинам. Если он удовлетворяет заданному условию (≥4.5), то из таблицы анкетных данных извлекаем фамилию, а из таблицы успеваемости - оценки по дисциплинам. Эти Данные плюс средний балл можно выдать на экран или поместить в другую таблицу.

**5. Древовидные таблицы**  
   
   **5.1.Сравнение табличной и древовидной структур**

Табличное представление данных находит самое широкое применение в информационных системах. Для таких систем ocновными операциями являются поиск и выборка нужной информации. Вычислительная сложность алгоритмов поиска, как мы знаем, зависит от того, является ли таблица упорядоченной или . Для неупорядоченных таблиц она порядка О(n), а для упорядоченных — порядка O(log2n). Если число элементов в таблице велико, то в большинстве случаев можно применять неупорядоченные таблицы. Однако большие объемы данных вынуждают создавать упорядоченные таблицы.

Оценим вычислительную сложность алгоритма создания таблицы, упорядоченной по возрастанию ключей. Алгоритм основан на том, что очередная запись таблицы помещается не простор конец таблицы, а сразу же в нужное место. Если ключ новой записи больше ключа последней записи в таблице, то новая запись включается в конец таблицы. В противном случае все записи, ключ которых больше ключа новой записи, сдвигаются на одну позицию к концу таблицы, и новая запись помещается на место последней сдвинутой записи. Таким образом, основными операциями алгоритма создания упорядоченной таблицы являются операции сравнения и перемещения. В наилучшем случае, когда при создании таблицы записи поступают в порядке возрастания ключей, для каждой записи тратится по одной операции сравнения и перемещения. В наихудшем случае, когда исходные записи упорядочены в об-ратном порядке, включение каждой i-й записи (i = 1,2, ....n) требует i операций сравнения и (i + 1) операций перемещения. Общее количество операций сравнения для n записей составит:

C= сумма от I=1 до n-1 от I=0.5(n2-n), а для операций перемещения  
М= сумма от I=1 до n-1 jn (I+1) = 0.5(n2-n)+(n-1)

Общее число операций сравнения и перемещения будет равно С + М = n2 — 1, т.е. порядка О(n2), такого же порядка будет и в среднем.

Теперь видим, что создание упорядоченной таблицы является достаточно сложной операцией и требует значительных затрат машинного времени. Кроме того, каждая операция включения новых и удаления существующих записей в упорядоченной таблице имеет вычислительную сложность порядка О(n).

Вычислительная сложность поиска в упорядоченной таблице O(log2n) такая же, как и в сбалансированном двоичном дереве поиска. Однако создание бинарного дерева поиска имеет сложность порядка O(nlog2n), а включение и удаление элементов — O(log2n), т.е. по этим показателям сбалансированное бинарное дерево поиска имеет значительное преимущество перед, упорядоченными таблицами.

**5.2. Представление древовидной таблицы**

Вышеизложенное наталкивает на мысль о возможности представления двоичного дерева поиска в виде таблицы, то есть создания древовидной таблицы поиска. Напомним, что бинарное дерево поиска организовано таким образом, что для любого узла ti, все ключи в левом поддереве меньше ключа ti, а ключи в правом поддереве больше ключа ti (все ключи уникальные). В дереве поиска можно на место каждого ключа, двигаясь начиная от корня и переходя на левое или правое поддерево каждого узла в зависимости от соотношения искомого ключа и ключа в узле дерева. Каждый узел, кроме ключа и данных, содержит два указателя: на левое и правое поддеревья.

Древовидная таблица поиска, как и обычная, представ собой массив. Каждый элемент массива — строка таблицы является записью, имеющей по крайней мере три поля: поле данных (ключа), указатель на левое поддерево и указатель на правое поддерево. Так как доступ к элементам массива осуществляется по индексам, то указатели являются не чем иным, как индексами элементов массива.

Таким образом, структура записи таблицы имеет вид: Левый указатель — Ключ + данные — Правый указатель

Поле «Ключ + данные» может иметь структуру любой сложности, состоящую из однородных или неоднородных элементов. Однако нас интересуют только ключи записей, по которым строится дерево и осуществляются все операции над записями: поиск, включение, удаление, выборка, обработка и обнов данных.

**5.3. Основные операции и возможная структура древовидной таблицы**

**Основными функциональными операциями над древовидными таблицами являются:**

* инициализация таблицы;
* включение элемента в таблицу;
* поиск элемента по ключу;
* удаление элемента;
* сохранение таблицы в файле;
* восстановление таблицы из файла;
* обходы дерева в таблице.

Эти операции по выполняемым функциям схожи с аналогичными операциями над бинарными деревьями поиска, однако aлгоритмы реализации их существенно различны. Это вызвано тем, каждый узел бинарного дерева поиска размещается в динамической памяти, получаемой при включении узла и освобождаемой при его удалении. Узлы же бинарного дерева в древовидной таблице размещаются в массиве (в векторной памяти). Это вызывает необходимость ведения списков занятых и свободных элементов массива. Список занятых элементов определяется указателями в узлах, начиная с корня, для ведения списка свободных элементов можно использовать один из указателей в свободных элементах. Для каждого списка необходим указатель на его начало.

Так как число элементов в массиве фиксировано, то требуется вести счетчик занятых элементов массива, а для сохранения таблицы в файле и последующего ее чтения из файла нужна информация о размере таблицы. Следовательно, таблица должна содержать некоторую управляющую информацию, включающую: указатель на корень дерева; указатель на начало списка свободных элементов; счетчик числа занятых элементов; число элементов таблицы.

Исходя из этого можно предложить такую структуру таблицы, в которой первые две строки заняты управляющей информацией. Массив имеет n элементов, первые два из которых содержат управляющую информацию, остальные m=n-2 предназначены для элементов двоичного дерева поиска.  
Таблица.

|  |
| --- |
|  |
|  |
| Указатель на корень | Пусто | Указатель списка свободных элементов |
|  |  |  |
| Счетчик занятых | Пусто | Размер таблицы m |
|  |  |  |
| Левый указатель | Ключ + данные | Правый указатель/ Указатель свободного элемента |
|  |  |  |
| ... | ... | ... |
|  |  |  |
| ... | ... | ... |

**Инициализация таблицы.** При инициализации таблицы обнуляются все левые указатели, включая указатель на корень и счетчик занятых элементов, очищаются поля ключей и данных. В указатель списка свободных элементов заносится индекс первой свободной строки, т.е. 2. Размер таблицы m устанавливается равным либо m = n (число строк массива), либо m = n - 2 (максимальное число узлов дерева). Правые указатели, начиная с третьей строки массива (с индексом i = 2), должны показывать на последующие строки (3, 4, ..., n). Правый указатель в последней строке можно обнулять, так как возможность включения нового элемента определяется не концом списка свободных элементов, а счетчиком занятых элементов. В качестве примера рассмотрим табл., в которой поле «Ключ+данные» представлено только ключом, имеющим положительные целочисленные значения, n= 15, m = 13. Таблица 6.2

|  |
| --- |
|  |
| 0 | 0 | 2 |
|  |  |  |
| 0 | 0 | 13 |
|  |  |  |
| 0 | 0 | 3 |
|  |  |  |
| 0 | 0 | 4 |
|  |  |  |
| ... | ... | ... |
|  |  |  |
| 0 | 0 | i+1 |
|  |  |  |
| ... | ... | ... |
|  |  |  |
| 0 | 0 | 14 |
|  |  |  |
| 0 | 0 | 15 |

Пусть элементы таблицы имеют структуру  
struct KR  
{ int L;     /\* Левый указатель \*/  
int key;    /\* Поле ключа \*/  
int R;       /\* Правый указатель \*/  
} Т[15];     /\* Массив под таблицу \*/

**Включение элемента в таблицу**. Примем, что запись с дублирующим ключом в таблицу не включается.

Исходные данные для включения — таблица Т и ключ key >0.  
Состояние таблицы в момент включения:

а) полная;  
б) пустая;  
в) заполнена частично.

*Состояние а) — таблица полная,* следовательно, счетчик занятых элементов равен размеру таблицы: T[1].L=T[1].R. Возврат – 2.  
*Состояние б) — таблица пустая,* т.е. Т[1].L =0.

1. Выбираем индекс j первой свободной строки из указателя списка свободных j = T[0].R и заносим его в указатель корня дерева T[0].L=j.  
2. Корректируем указатель списка свободных элементов, заненеся в него указатель на следующий свободный элемент T[0].R=T[j].R  
3. В свободной строке j формируем узел дерева R[j].key=key; T[j].L=T[j].R=0  
4. Корректируем счетчик занятых элементов, увеличив его на единицу T[l].L=T[l].L+l.

*Состояние в) — таблица заполнена частично.*

1. Начиная с вершины дерева i=T[0].L, ищем ключ или место включения нового элемента. Исходы поиска: ключ найден или не найден.  
2. Ключ найден, возврат — 1.  
3. Ключ не найден: выбираем свободную строку j из списка свободных и переносим индекс j в указатель предыдущей строки.  
4. Выполняем пункты 2), 3), 4) состояния б).

**Поиск элемента по ключу.** Исходные данные: таблица Т и ключ key. Состояния таблицы: а) таблица пустая, б) не пустая.

а) Таблица пустая, т.е. T[1].L = = 0, возврат — 2.  
б) Таблица не пустая.

1. Начиная с вершины дерева j=Т[О].L, ищем ключ. Исход: найден или нет.  
2. Найден — возврат индекса j найденной записи, не найден — возврат — 1.

**Удаление элемента.** Исходные данные: таблица Т и ключ key. При каждом движении по дереву в поисках записи с заданным ключом запоминаем адрес предыдущего указателя. Состояния таблицы: таблица пуста или не пуста. Если табли¬ца пуста, T.e.T[l].L= =0, то возврат — 2. Если таблица не пуста, то осуществляется поиск ключа. Если не найден, то возврат — 1.  
Ключ найден в строке j, возможные положения узла в дереве:

а) Узел не имеет потомков.  
б) Узел дерева с двумя потомками.  
в) Узел дерева с одним потомком,

а) **Узел не имеет потомков.**  
Если узел является корнем дерева, то:

1. Обнуляем указатель на корень дерева (указатель списка занятых элементов) T[0].L=0.  
2. Счетчик занятых T[1].L уменьшаем на 1.  
3. Корректируем список свободных элементов, включив освобожденную строку в начало списка свободных: T[j].R=T[0].R; T[0].R=j. Чистим левый указатель и ключ строки j: T[j].L=0; T[/].key=O.  
5. Возврат j.

Можно было поступить проще. Так как с удалением корня дерево становится пустым, достаточно заново инициализировать таблицу. Если же узел является листом дерева, то обнуляем предыдущий указатель, затем выполняем пункты 2—5.

**б) Узел дерева с двумя потомками.**

1. Шаг по дереву влево вниз на один уровень, затем вправо вниз до конца. При каждом движении запоминаем адрес предыдущего указателя.  
2. Данные+ключ (здесь только ключ) конечного элемента пересылаем в поле данные+ключ удаляемого элемента.  
3. Левый указатель конечного элемента пересылаем в предыдущий указатель.  
4. Выполняем пункты а) 2—5.

**в) Узел дерева с одним потомком.**  
Если удаляемый узел является корнем дерева, то в указатель списка занятых заносим индекс следующего элемента (правого или левого) по дереву, затем выполняем пункты а) 2—5.

Если же удаляемый элемент является промежуточным узлом дерева, то в указатель предыдущего элемента заносим индекс следующего элемента, затем опять-таки выполняем пункты а) 2-

Отметим, что при удалении любого узла выполняются пункты а) 2—5.  
**Сохранение таблицы в файле**. Исходными данными являю таблица Т и имя файла. Естественно, сохраняются только не пустые таблицы.

1. Получение имени файла.  
2. Открытие файла для записи (как выходного), если это существующий файл, то старые записи затираются. Если файл не удастся открыть, то возврат -1.  
3. Извлечение из таблицы размера таблицы m = T[l].L n = m + 2.  
4. Форматированный вывод в файл n записей из таблицы.  
5. Закрытие файла. Возврат 0.

**Восстановление таблицы из файла**. Исходные данные файла. Возвращаемое значение — указатель на динамическую таблицу либо NULL в случае неудачи.

1. Получение имени файла.  
3. Открытие файла, в случае неудачи возврат NULL. Ввод из файла двух первых записей с управляющей информацией таблицы в переменные типа KR.  
4. Извлечение размера таблицы из второй записи и определение общего числа элементов n=m + 2.  
5. Получение динамической памяти для таблицы с n элементами.  
6. Занесение в таблицу первых двух элементов, считанных из файла раньше.  
7. В цикле ввод из файла последующих (n - 2) записей в строки i = 2, 3, ..., n - 1.  
8. Возврат указателя на таблицу. По этому указателю теперь можно выполнять все операции, рассмотренные выше (кроме инициализации таблицы).

**Обходы дерева в таблице**. При рассмотрении деревьев в списковой памяти мы отмечали, что для управления деревом создается дескриптор, содержащий такие данные, как имя дерева, дата создания, информация об узлах дерева и обязательно указатель на корень дерева. Обращение к дереву осуществляется через дескриптор по схеме:  
*Указатель на дескриптор -> Дескриптор —> Корень дерева.*

В простейшем случае дескриптор содержит только указатель на корень дерева, тогда обращение к дереву предельно упрощается:  
*Указатель на корень дерева —> Корень дерева.*

Для управления деревом в древовидной таблице мы создали управляющую информацию в первых двух строках таблицы, корень дерева первоначально находится в третьей строке, а по мере удаления элементов может переместиться в любую строку таблицы, кроме первых двух. Указатель на корень дерева всегда находится в левом указателе первой строки T[0].L.

Для обхода дерева в списковой памяти мы применяли рекурсивные функции. Функция обхода содержала только один параметр: указатель на корень дерева. Применение аналогичной функции обхода в древовидной таблице потребует наличия двух параметров: указателя на таблицу (массив) и указателя на корень, содержащего индекс строки таблицы с корнем, то есть целого числа ≥2. Если таблица находится в массиве Table, то обращение к функции нисходящего обхода имеет вид: Preorder(Table, Table[0].L).

**6. Таблицы с вычисляемыми входами**  
   
   **6.1. Понятие таблицы с вычисляемыми входами**

Метод двоичного поиска, как мы знаем, обеспечивает время поиска порядка O(log2 n), т.е. зависит от размера таблицы. Естественное стремление человека к экономии привело к разработке теории и практике применения***таблиц с вычисляемыми входами*** . Любой элемент такой таблицы имеет свой адрес, известный (доступный) программе обработки таблицы. Время обращения к элементам таблицы с вычисляемыми входами не зависит или почти не зависит от размеров таблицы.

Были разработаны различные способы организации таблиц с вычисляемыми входами и соответствующие алгоритмы работы с такими таблицами. Подход к организации таких таблиц состоит в том, чтобы положение (место, адрес) конкретной записи в таблице определять по значению некоторого ключа этой записи. Существует множество способов отображения числового значения ключа в адрес соответствующего элемента в таблице.

Зависимость между адресом записи в таблице А и ее ключом k называется***функцией расстановки*** *(адресной функцией)* А =f(k). В зависимости от характера функции f методы вычисления адреса по ключу можно разделить на две группы:

а) методы, в которых функция расстановки реализует взаимно однозначное соот-ветствие между адресом А и ключом k;  
б) методы перемешивания (хеширования, рандомизации), в которых для различ-ных ключей k1 ! = k2 функция расстановки может дать одно и то же значение, т.е. один и тот же адрес в таблице: f(k1 )=f(k2 ),k1 ! = k2 .

Такая ситуация носит название коллизии (конфликта), а ключи k1 и k2 являются синонимами функции f. Естественным является стремление к уменьшению числа коллизий. Выбор функции расстановки и нахождение способов разрешения коллизий во многом зависят от решаемой прикладной задачи и вызывают немало трудностей, однако это оправдывается конечным результатом - быстротой обработки таблицы.

Подбор функции расстановки f(k), обеспечивающей взаимную однозначность преобразования ключа записи в адрес ее хранения, в общем случае весьма затруднителен. Этот случай является идеальным, и таблица с вычисляемым входом представляется в виде массива из N элементов T[i], i = 0, 1, ..., N - 1, где N — число возможных значений ключей, т.е. число элементов таблицы совпадает с числом возможных значений ключей. Предполагается, что функция расстановки f(k) ставит в соответствие всякому ключу kjнекоторое целое i, различное для разных kj , и такое, что 0≤ i≤ N-l. Тогда элемент с ключом kj достаточно разместить в T[i], и время поиска становится постоянным 0(1). Наиболее простым является случай, когда значение ключа к равно i или i + const.

Пример 1 . Имеется список студентов учебной группы, хранящийся в виде таблицы. Каждый студент имеет свой фиксированный порядковый номер i в списке (i = 1, 2, ..., N). Тогда доступ к данным i-го студента в таблице можно осуществлять по индексу записи в таблице T[i - 1]. Число элементов в таблице равно числу студентов в учебной группе. Здесь учитывается тот факт, что состав учебной группы остается неизменным. Таким образом, в рассмотренном примере ключом доступа является номер студента в списке, адресом входа в таблицу — индекс строки массива, функция расстановки А = f(k) = к - 1.  
Пример 2 . Учет числа свободных мест в вагонах поездов. Пусть осуществляется продажа билетов за 30 дней до отправления поезда. Всего имеется 50 номеров поездов, каждый поезд имеет не более 15 вагонов. Таблица обновляется каждый день. Составим таблицу, в которой для каждого поезда Р (1 ...50) на каждый день отправления D (1...30) имеется одна запись со структурой: номер поезда, всего свободных мест — в вагоне 1, 2, ..., 15. Размер таблицы будет равен m = P\*D записей. Ключом записи является пара значений Р и D, а функция расстановки будет иметь вид: f(P,D) = (Р- l)\*30 + (D- 1).

В идеальном случае, когда все поезда отправляются каждый день в пятнадцативагонном составе, таблица используется полностью. Однако если некоторые поезда отправляются только в определенные дни или в составе поезда меньше 15 вагонов, то будет только частичное использование таблицы.

Следовательно, если число возможных значений адресов, вычис¬ляемых по функции расстановки, значительно больше числа возможных записей в таблице, память будет использована нерационально. Во многих случаях число возможных значений ключа N настолько вели ко, что нельзя и помышлять о создании массива размером N. Так, если в качестве ключа использовать фамилию длиной до 6 букв русского алфавита, то число возможных значений ключа превысит несколько миллиардов.

С другой стороны, число реальных значений ключа m бывает значительно меньше числа возможных значений N: m « N. Тогда таблица заполняется лишь частично, т.е. коэффициент заполнения к = m/N <<1. Действительно, даже в простейшем случае, когда в качестве ключа использовался 8-битовый символ, число возможных значений N = 256, а число реальных ключей вряд ли превысит 100.

Следующая проблема связана с размещением элементов в таблице при возникновении коллизий. Где разместить элемент, вызвавший коллизию? Внутри самой таблицы или за ее предела. Таким образом, при создании таблиц с вычисляемыми входами возникают две проблемы:

1) как лучше выбрать функцию расстановки f, чтобы уменьшить число коллизий;  
2) где разместить конфликтные записи: внутри самого массива или вне его.

**6.2. Выбор функции расстановки**

При выборе функции расстановки f, отображающей множество ключей k в адреса аi, i = 0, 1, ..., N - 1, где N — размер таблицы, желательно, чтобы она не только сокращала число коллизий, но не допускала *скучивания*ключей в отдельных частях таблицы. Поэтому ее подбирают из условия случайного и возможно более равномерного отображения ключей в адреса хранения. Таблицы, построенные по такому принципу, называют ***таблицами со случайным перемешиванием*** *(хеш-таблицами)*, а вычисление адресов — ***хешированием*.**

Функция расстановки f(k) должна иметь значения между О и N -1. По этой причине ее часто определяют в виде f(k) = q(k) mod N, где q(k) — функция, принимающая целочисленные значения. Тогда существуют **два вида коллизий:**

а) непосредственные коллизии, соответствующие ключам кiи kj, таким, что q(ki) = q(kj), i! = j;  
б) коллизии но модулю, порождаемые ключами кi и kj, такими, что q(ki)! = q(kj), но q(ki)modN = q(kj)modN, i! = j.

Существует правило Люма : доля коллизий по модулю уменьшается, если размером таблицы выбрать целое N, не имеющее делителей, меньших 20. N можно взять, например, простым, но это не обязательно.

Функция q(k) должна обеспечивать хорошее начальное распределение адресов по всей таблице. Для выбора такой функции нет готовых рецептов. Во многом это определяется решаемой задачей. Полезно построить несколько таких функций и моделировать их поведение при решении задачи.

**Метод деления**. В методе деления в качестве значения хеш-функции используется остаток от деления ключа на некоторое целое число М: А = kmodM или а = q(k)modM. Эффективность рассеивания по таблице во многом зависит от значения М. Плохо, если М является степенью основания системы счисления ключа или 2, при М = 28— это значение последнего символа при символьном ключе. Лучше, если М является простым числом. Очевидно, что в этом случае число возможных адресов равно М (0, 1, ..., М - 1), все ключи, равные (к + i\*M), i = 0, 1, ..., будут конфликтными.

Рассмотрим таблицу, в которой в качестве ключей записей используются фами-лии студентов учебной группы. Используем для вычисления адреса входа в таблицу только первую букву фамилии (заглавная буква русского алфавита). В русском алфавите 32 буквы, коды прописных букв имеют значения С, = 128, 129, ..., 159. Если в качестве адреса взять а = Ci-128, то а = 0, 1, ..., 31. Если список учебной группы содержит всего 17 фамилий, то по методу деления адреса могут быть вычислены как ai = (Сi-128)modl7. Фамилии, начинающиеся с А и С, Б и Т, ... будут конфликтными по модулю.

Очевидно, что для больших таблиц вычисление адреса по одной букве, когда число возможных адресов всего 32, мало подходит. Можно предложить такой подход. Рассматривая коды символов символьного ключа как целые без знака, находим сумму значений всех символов ключа и полученную сумму делим на число символов в ключе, затем целую часть частного делим на M и в качестве значения хеш-функции используем остаток от такого деления.

**Метод умножения**. Представим ключ в виде двоичною числа. Возьмем некоторую дробь р, умножим дробь р на k и возьмем дробную часть этого произведения (р\*к). Затем умножим ее на М и возьмем целую часть. Хорошие результаты получаются, если в качестве р взять «золотое сечение», равное 0, 6180339887.

Рассмотренные два метода — метод деления и метод умножения, являются во многих случаях наилучшими.

**Метод середины квадратов**. Значение ключа представляется целым двоичным числом и возводится в квадрат. За адрес записи А принимается m бит средней части результата к2. Этот метод очень близок методу умножения, однако по многим параметрам уступает ему.

**Преобразование системы счисления**. Пусть ключ выражен в некоторой р-ичной системе счисления: к = а0 + а1р + а2р2 + … . Функция t(k), используя те же коэффициенты аi, в q-ичной системе счисления вычисляет значение полинома t(k) = ao + а1q+a2q2+…+as-1qs-1 для 5 младших цифр ключа, р и q — положительные числа, р < q, либо вычисляет t(k) с использованием всех коэффициентов, затем оставляют s младших цифр:  
27130410 ≥ 2\*135 + 7\*134 +1\*133+3\*132+0\*13+4\*130=945221, взяв, младшие 4 цифры, получим адрес 5221.

Метод слияния (метод свертки). Ключ разбивается на не¬сколько отрезков (частей) постоянной длины, затем эти части арифметически или логически склады-ваются. Символьные ключи перед разбивкой могут преобразоваться в двоичный код. Пример: ключ 271304 = > 271 + 304 = 575. Естественно, могут быть предложены другие способы разбивки на части.

**Метод деления многочленов**. Пусть к выражено в двоичной системе счисления к =bn2n + bn-1 2n-1+…+b020.

Используя те же коэффициенты, представим k в виде многочлена r(t) = bntn+bn-1tn-1+…+b0t0. Найдем остаток от деления r(t) на постоянный для всех ключей многочлен c(t): c(t) =cmtm+cm-1tm-1+…+c1t+c0.

Этот остаток, опять рассматриваемый в двоичной системе счисления, и используем в качестве значения A =f(k). Этот метод трудоёмок, поэтому находит реализацию аппаратным или микропрограммным способом.

Методы вычисления хеш-функции и методы вычисления последовательностей псевдослучайных чисел имеют много общего. Отсюда одно из названий функции расстановки —*функция рандомизации*. Получение очередного псевдослучайного числа не что иное, как вычисление значения хеш-функции от последнего (или двух последних) значения последовательности.

**6.3. Разрешение коллизий методом цепочек**

Каким бы ни был выбор хеш-функции, возможны коллизии, первичные и вторичные скучивания. Для разрешения коллизий предложено несколько методов. Для различных операций над таблицами (поиск, вставка, удаление) эти методы несколько различаются. Методы разрешения коллизий, возникающих при вставках ключей, можно разбить на две группы:

* методы цепочек;
* методы открытой адресации.

**Методы цепочек**. Методом цепочек называется метод, в котором из записей, вызвавших коллизию по каждому входу аi, образуется свой список. Для поддержания списка во все записи таблицы добавляются указатели. Списки образуются по мере необходимости по одному на каждый возможный хеш - адрес таблицы. Сами списки могут размещаться как в памяти, принадлежащей хеш-таблице, так и в отдельной памяти. В первом случае цепочки называются внутренними, а во втором случае — внешними.

***Таблицы с внешними цепочками***. При использовании внешних цепочек возможны два способа организации хеш-таблицы.*В первом способе* строка хеш-таблицы содержит запись таблицы с полями данных и полем указателя. Первая запись, поступившая в хеш-таблицу по адресу ai, помещается в строку таблицы, поле указателя обнуляется. Все другие записи, поступившие по этому же адресу а, т.е. записи, вызвавшие коллизии, помещаются в цепной список переполнения. Таким образом, реально создаются списки только для тех строк хеш-таблицы, при входе в которые возникли коллизии. *Во втором способе* строка хеш-таблицы содержит только одно поле, поле указателя, т.е. хеш-таблица является адресной таблицей. Первоначально все указатели обнулены. Любая запись, поступившая в хеш-таблицу по адресу аi, помещается в цепной список переполнения для этого адреса. Данный способ позволяет создавать хеш-таблицы больших объемов, обеспечивая простое разрешение коллизий. Кроме того, когда элементы таблицы имеют переменную длину или когда цепные списки размещаются на ВЗУ, этот способ является наиболее подходящим.

Цепные списки в основной памяти создаются либо в динамической памяти, когда для каждой записи запрашивается память и для каждого хеш-адреса создается свой цепной список, либо в статической памяти — в таблице переполнения, единой для хеш-адресов.

Среднее число проб при удачном поиске равно 1 + к/2, к — коэффициент заполнения таблицы.

***Таблица с внутренними цепочками***. Таблицы с внутренними цепочками могут обеспечить более эффективное использование памяти. Если известно, что таблица в процессе ее использования будет оставаться почти неизменной, то такая таблица обычно заполняется в два этапа. Сначала поступающие записи помещаются в хеш-таблицу и временную таблицу переполнения. Затем цепочки переполнения переносятся из таблицы переполнения в незанятые строки основной хеш-таблицы. Еще один подход это заполнение хеш-таблицы сразу, без применения дополнительной таблицы переполнения. При возникновении коллизий поиска свободного места в таблице можно использовать разные методы, например последовательный просмотр строк таблицы. При вставке новых элементов появляется тенденция объединения хеш-адресов в группы, что проявляется в срастании списков. Причина в том, что свободное пространство принадлежит хеш таблице и коллизия становится возмож-ной не только для ключей-синонимов по хеш-функции, но и для ключей-синонимов списку переполнения. Следовательно, в один список могут попасть ключи, не являющиеся синонимами по хеш-функции, что удлиняет список.

Чтобы избежать этого, можно переместить «чужой» ключ из списка и освободить занятое место, однако для этого требуются дополнительные затраты времени. Итак, при таком подходе решение коллизий происходит за счет «чужих», еще не занятых строк хеш-таблицы. Поэтому этот метод называют еще***методом срастающихся цепочек***.

Если коэффициент заполнения таблицы равен к, то средне число проб для метода внутренних цепочек при удачном поиске равно 1 + к/4+1/(8\*k)(e2\*k-1-2\*k), а при неудачном поиск равно 1 + 1/4\*(е2\*k - 1 - 2\*к).

**6.4. Методы открытой адресации**

Используется только одна хеш-таблица с N элементами, цепочки переполнения не создаются. Суть метода заключается в том, чтобы после входа в таблицу по адресу аi полностью отказаться от указателей и просто по некоторому правилу (алгоритму) просматривать строки таблицы, пока не будут найдены либо свободная строка при вставке нового элемента, либо ключ к при поиске элемента.

Алгоритм открытой адресации в общем виде можно описать следующим образом.

1. Вычисляем адрес точки входа в таблицу по ключу к а = f(k). Установим счетчик i = 0.  
2. Если строка Т[а] свободна, то, значит, записи с ключом к в таблице нет. При вставке в строку помещаются запись и удачный конец алгоритма, при поиске конец алгоритма при отсутствии ключа.  
3. Если ключ в строке Т[а] равен k, тогда при вставке — конец алгоритма по признаку «дублирующий ключ», при поиске — удачный конец и возврат адреса строки ai.  
4. В противном случае — коллизия. Увеличиваем счетчик на 1: i = (i+ 1) и по некоторой функции вычисляем адрес ячейки внутри таблицы аi= g(f(k), i).

Переход на п. 2.

Функции g(f(k), i), 1≤ i ≤ N, определяют последовательность адресов для просмотра элементов таблицы. Их выбор вызывает некоторые трудности и в основном определяется особенностями использования таблицы при решении конкретной задачи. Наиболее распространенными схемами открытой адресации являются:

* линейная открытая адресация;
* квадратичная открытая адресация;
* открытая адресация с двойным хешированием.

*Линейная открытая адресация*. Это простейшая схема открытой адресации, при которой ai= g(f(k), i) = f(k) + с\*i, где с константа. Наиболее простая схема при с = 1.

Чтобы избежать первичного скучивания записей, с и N должны быть взаимно простыми, а с — не очень малым числом. Эта схема работает хорошо, пока таблица не слишком заполнена но по мере заполнения процесс замедляется.

При с = 1 среднее число проб при неудачном поиске равно: 0.5\*(1 + (1 - к)-2), а при удачном 0.5\*(1 + 1/(1 - к)), где к — коэффициент заполнения таблицы.

*Квадратичная открытая адресация*. При квадратичной адресации ai= g(f(k), i) =f(k) + c\*i + d\*i2, где с u d — константы. Число, проб здесь меньше, однако при почти заполненной таблице не удается избежать вторичного скучивания, когда при большом числе ключей синонимы одной группы вступают в коллизию с другими ключами.

*Открытая адресация с двойным хешированием*. По этой схеме аi = g(f(k), i) = f{k) + i\*h(k), где h(k) — хеш-функция, почти такая же, что и f(к), но не эквивалентная ей. Возможны различные варианты f(k) и h(k). Например, если N — простое число и f(k) = kmodN, то можно взять h(k) = 1 + kmod(N - 2), т.е. аi = g(f(k), i) = kmodN + i\*(1 + k\*mod(N - 2)). В этом случае, функции f(к) и h(k) являются «независимыми» в том смысле, что на различных ключах значения обеих функций f и h совпадают с вероятностью O(1/N2), а не O(1/N).

Среднее число проб при неудачном поиске равно 1/(1 - к), а при удачном 1/k\*ln(1/(1 - к)), к — коэффициент заполнения таблицы. Если функция h(k) зависит от f(k), то вторичные скучивания вызывают увеличение средних значений.

**6.5 Особенности алгоритмов удаления записей из таблицы**

Удаление из таблиц с однозначным соответствием между адресом в таблице и ключом не вызывает никаких трудностей, просто строка Т[а] с удаляемым ключом к помечается как свободная (поле ключа очищается).

Для таблиц с внешними цепочками переполнения в динамической памяти удаление является процедурой, обратной вставке, и выполняется аналогично операции удаления из обычного линейного списка. Корректируются значения указателей, и освобождается динамическая память. Однако если первый элемент по адресу а = f(к) хранится в самой хеш-таблице, то процедура удаления усложняется.

Для таблиц с линейной открытой адресацией при с = 1 придется переместить содержимое всех строк, следующих за удаляемой строкой, на одну позицию. Для всех остальных видов таблиц при удалении записи нарушается связь между синонимами. Чтобы этого не произошло, вместо ключа в строке удаляемой записи можно вписать специальный код. Тогда каждая строка хеш-таблицы находится в одном из трех состояний: свободном, занятом или удаленном. Строка из свободного состояния может перейти в занятое состояние, но освободившаяся строка не сможет перейти обратно в свободное состояние, если строка является промежуточным элементом цепочки. Взаимные переходы из занятого состояния в удаленное и обратно допустимы.

Строка с удаленным состоянием в процессе поиска записи по ключу просматривается и пропускается, а в процессе вставки используется для размещения новой записи, если запись не является начальной в своей цепочке. Поэтому, несмотря на постепенное увеличение числа удаленных записей и снижение реального коэффициента заполнения таблицы, время поиска может и не уменьшаться.

**6.6. Переразмещение (рехеширование) таблицы**

Выбор размера таблицы вызывает определенные трудности. При малых коэффициентах заполнения время поиска и вставки минимально, но память используется неэффективно. Если размер таблицы выбран из максимального числа записей, то коэффициент заполнения становится близким к единице и время поиска и вставки заметно возрастает.

Переполнение таблицы — явление редкое, необходимость в увеличении размера таблицы возникает обычно из-за резкого увеличения времени обработки, а не физического переполнения таблицы. Увеличение размера таблицы ведет к изменению хеш функции и необходимости полного переразмещения записей в расширенной таблице и использования новой хеш-функции. Taкой процесс получил название *рехеширования таблицы.*

**Алгоритмы на графах**

Графы как структуры данных широко применяются при решении многих задач. Они могут использоваться для представления данных, моделей процессов, структур программ и т.д. Мы с вами рассмотрим два основных вопроса:

1) представление графов в программах;  
2) алгоритмы на графах.

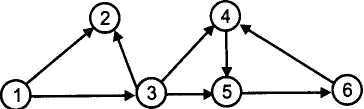
Необходимо отметить, что многие задачи, решаемые с использованием графов, относятся к задачам выбора. Все они могут быть, решены с помощью переборных алгоритмов. Однако сложность таких алгоритмов настолько велика, что многие практические задачи не могут быть решены за приемлемое время. Мы рассматрим некоторые из известных алгоритмов на графах, получивших практическое применение. Большинство из них основано на эвристических методах.  
   **1. Основные определения**

Граф G = (V,E) включает непустое множество V, называемое **множеством вершин (узлов) графа**, множество Е, представляющее собой множество ребер графа и отображение множества ребер на множество пар элементов из множества V.

Множества V и Е конечны. Из определения следует, что каждому ребру графа x&8712E можно поставить в соответствие пару вершин (u,v) ∈V этого графа, тогда говорят, что ребро х соединяет вершины u и v. Любые две вершины u и v, соединен-ные в графе ребром x называются смежными, и говорят, что ребро х инци¬дентно вершинам u и v. Граф, у которого каждая вершина смежна со всеми остальными вершинами, называется **полным графом**.

Ребро x, направленное от одной вершины u к другой v, называется **ориентированным ребром, или дугой.** Граф, содержащий только ориентированные ребра, называется**ориентированным графом(орграфом)**. Ребро, не имеющее определенного направления, называется **неориентированным ребром**. Граф содержащий неориентированные рёбра называется **неориентированным**. Граф, содержащий как ребра, так и дуги, называется **смешанным графом.**

Графическому представлению ориентированного графа с 6 вершинами и 8 дугами, приведенному на рис., соответствует граф G = (V,E), у которого  
V = {1,2,3,4,5,6} и Е ={(1,2),(1,3),(3,2),(3,4),(3,5),(4,5),(5,6),(6,4)}.



Ориентированный граф

Ребро х, соответствующее упорядоченной паре вершин (u,v), начинается в вершине u (исходит из u) и заканчивается в вершине v (входит в v), u — начальная вершина ребра (дуги), v — конечная. Ребро, соединяющее некоторую вершину саму с собой, называется **петлей**, в нашем рассмотрении петли не допускаются. В случае ориентированных ребер между парой вершин возможны два ребра, которые имеют противоположную направленность и считаются разными ребрами х1= (u,v), X2= (v,u). Если между любой парой вершин графа имеется не более одного ребра в неориентированном графе или не более одного ребра данного направления в ориентированном графе, то такой граф называется **простым**, в противном случае — **мультиграфом**. Мы будем рассматривать только простые графы.

Граф, у которого каждому ребру приписан вес, называется **взвешенным графом**. Вершина, у которой нет ни одной смежной вершины, называется **изолированной**. Граф, имеющий только изолированные вершины, называется **нульграфом**. Для ориентированного графа число ребер, исходящих из некоторой начальной вершины v, называется **полустепенью исхода**. Число ребер, для которых вершина v является конечной, называется **полустепенью захода вершины v**, а сумма полустепеней исхода и захода называется **полной степенью этой вершины**. Любая последовательность ребер простого орграфа такая, что конечная вершина любого ребра этой последовательности является начальной вершиной следующего за ним ребра, задает путь в графе. **Путь в графе** обходит все вершины данной последовательности, если начало пути находится в начальной вершине первого ребра, а конец — в конечной вершине последнего ребра последовательности. **Длиной пути** считается число ребер в пути. **Простой путь**(простой относительно ребер) — это путь в графе, все ребра которого различны. **Элементарный путь**(простой относительно вершин) — путь, в котором все вершины различны. Все элементарные пути — простые, но не наоборот. **Цикл (контур)** — путь, начинающийся и заканчивающийся в одной и той же вершине. **Простой цикл**— соответствующий ему путь простой. **Элементарный цикл** проходит через любую вершину не более одного раза. В элементарном цикле начальная вершина появляется два раза, в простом — может более двух раз. Ациклический орграф — граф, не имеющий ни одного цикла. Связный граф — граф, в котором есть пути между любыми двумя вершинами

**2. Представление графов.**

Реализация алгоритмов обработки графов в виде машинных процедур в высокой степени зависит от способа представления графов на логическом и физическом уровнях. А способ представления графа существенно зависит от типов структур данных, допускаемых используемым языком программирования и типом ЭВМ. Выбор наилучшего способа представления графа зависит от природы моделируемых данных (процессов) и операций, выполняемых над ними. На выбор подходящего представления влияют такие факторы, как число вершин, максимальная полустепень исхода, частота изменения числа вершин и ребер, является ли граф ориентированным или нет и т.д. Таким образом, какого-то одного наилучшего способа представления не существует, лучшее представление зависит от решаемой задачи. Выбор конкретного представления графа может существенно повлиять на эффективность его обработки.

Граф G = (V,E) может быть полностью определён простым перечислением двух множеств V и E. Однако в большинстве случаев этот способ представления не позволяет создавать эффективные алгоритмы обработки графов.

Широко распространенным способом представления графов являются матричный способ и соответствующее отображение их в векторной (смежной) памяти в виде двухмерных массивов. Этот способ имеет несомненные достоинства:

а) массивы легко хранить и обрабатывать в ЭВМ;  
б) для получения характеристик графа могут быть использованы операции матричной алгебры;  
в) имеются хорошо известные алгоритмы обработки графов.

Однако матричный способ представления имеет и крупный недостаток, связанный с тем, что массивы являются статическими по размерам структурами.

**2.1. Матрица смежности**

Очень часто граф представляется в виде матрицы смежности Аn\*n, в которой аij=1, если (vi,vj) принадлежит Е, аij=0 в противном случае.

Для простых неориентированных графов матрица смежности симметрична, т.е. Aij=aji. В случае взвешенного графа полагается aij=wij где wij — вес ребра.

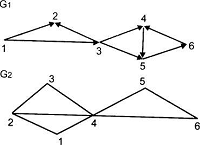
Матрица смежности зависит от упорядочения графа (от порядка нумерации вершин). Для различных упорядочений получаются различные матрицы (изоморфизм), однако любая матрица смежности графа G может быть получена из другой матрицы смежности этого же графа путем перестановки некоторых строк и соответствующих им столбцов.

Используя матрицу смежности, можно определить все пути между вершинами vi и vj и их длины, определить все возможные пути, циклы, простые и элементарные пути и т.п.

Рассмотрим представление орграфа G1 и неориентированного графа G2 в виде матриц смежности. СМ рисунок.

Если в графе ребер (дуг) много, то такое представление достаточно компактное, в противном случае получается разреженная матрица.

При разработке программ, предназначенных для работы с графами, необходимо контролировать правильность задания параметров графа. Естественно, при числе вершин графа, меньших 2, не приходится говорить о матрице смежности. Если m — число вершин графа и n — число дуг, очевидными являются требования: m≥2, n≥1 и n≤m\*(m - 1) для ориентированного или n≤m\*(m - 1)/2 для неориентированного графа. Здесь учитывается тот факт, что в ориентированном графе вершины могут быть соединены прямыми и обратными дугами.



Риc. Ориентированный (G\) и неориентированный (G2) графы -,G2 G1

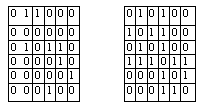


Рис. Матрицы смежности ориентированного (Gi) и неориентированного (G2) графов

**2.2. Векторы смежности**

При представлении графа в виде векторов смежности создается матрица, число строк которой равно числу вершин графа. Каждая i-я строка матрицы содержит номера вершин, смежных с i-той вершиной, петли исключаются. Число столбцов матрицы определяется максимальной полустепенью исхода в орграфе или максимальной степенью в неориентированном графе.

G1             G2  
2 3 0          2 4 0 0 0  
0 0 0          1 3 4 0 0  
2 4 5          2 4 0 0 0  
5 0 0          1 2 3 5 6  
6 0 0          4 6 0 0 0  
4 0 0          4 5 0 0 0  
Представление векторов смежности массивами

Такие массивы занимают меньше памяти, время просмотра их сокращается. Однако и здесь не удается избежать разреженности матрицы, если полустепень исхода какой-либо отдельной вершины близка к числу вершин графа.

**2.3. Списки смежности**

Список смежности для вершины v — это для орграфа список концов дуг, исходящих из вершины v, а для неориентированно графа — список смежных с v вершин. Для орграфа G1 и графа G2 списки смежности выглядят так:  
G1                  G2  
1 => 2→3             1=> 2→4  
2=> NULL            2=> 1→3→4  
3=> 2→4→5        3=> 2-→4  
4=>5                     4=>1→2→3→5→6  
5=>6                     5=>4→6  
6 =>4                    6=>4→5  
Здесь: => — указатель начала списка, → связь вершин в списке смежности.

Списки смежности можно представить в виде связанных последовательных списков, причем для каждой вершины создается свой последовательный список с динамическим получением памяти для каждого элемента списка. Для указателей списков тогда создается массив указателей, число элементов которого равно числу вершин графа. Очевидно, что алгоритмы обработки графов при таком их представлении сильно усложняются.

Проще использовать для представления списков смежности графа двухмерный массив или два одномерных массива. Рассмотрим вариант представления неориентированного графа в виде списков смежности на примере простого графа.



Рис. Представление графа в виде списков смежности

Граф имеет Р=5 нумерованных вершин (1,2...5) и Q=6 рёбер. Зададим граф в виде ребер, указывая (U,V), где u и v — номера вершин, 1=(u,v)≤P: (1,2); (1,3); (2,3); (2,4); (3,5); (5,4). Эти данные являются исходными при представлении графа в виде списков смежности для вершин 1,2,...Р. Списки смежности представим в векторной памяти в виде двух одномерных массивов с индекса 1,2,...N, где N=P+2Q. В массиве NEXT первые Р элементов содержат указатели списков (индексы строк массивов) для каждой из вершин графа 1,2,...Р. Остальные 2Q элементы массива NEXT содержат указатели на следующие элементы в каждом из списков. Первые Р элементов массива ADJ не используются, в остальных 2Q элементах содержатся номера вершин ребер (v,u) в пордке их поступления при формировании массива. Обратите внимание на то, что ребро задается вершинами (u,v), а в массив верп ны заносятся как (v,u).  
Алгоритм построения и формирования массивов ADJ и NEXT достаточно прост.

1. Ввод числа вершин Р и числа ребер Q, Q≤P(P - 1).  
2. Определение размера массивов N=P+2Q, получение динамической памяти для них (ADJ и NEXT), если они не были определены в программе как переменные.  
3. Обнуление первых Р элементов массивов ADJ и NEXT.  
4. В цикле для каждого ребра вводятся номера его вершин (U,V), которые размеща-ются в последовательных ячейках ADJ, начиная с элемента ADJ[P+1] в порядке (V,U), формируются связи (указатели) списков.

**2.4. Матрица инцидентности**

Напомним, что любые две вершины u и v, соединенные в графе ребром а, называются **смежными**, и говорят, что ребро а инцидентно вершинам u и v.

Любой граф можно задать матрицей инцидентности. Матрица инцидентности B[bij] размерностью n\*m (n — число шин, m — число дуг) определяется следующим образом: Ьц=+1 если ui является начальной вершиной дуги aj; bij=-l, если ui является конечной вершиной дуги аj; bij=0 в противном случае.

Если граф неориентированный, то его матрица инцидентности определяется аналогично, за исключением того, что элемент -1 поменяется на +1

**3. Пути в графе**

Во многих задачах требуется определение путей в графах\* К ним относятся задачи определения наличия пути между дву\ вершинами, всех путей между ними, экстремальных путей, пут от одной вершины до всех других, определения дуг, составляю!\*! щих путь, и т.п.

Рассмотрим матрицу смежности. Единица в /-й строке и j-м столбце матрицы А, т.е. ау=1, указывает на наличие ребра (v,,Vj)k т.е. на путь длиной 1 из V,B VJ. Элементами матрицы А2 будут; ay(2)= i(aik\*ak).

Для каждого к условие atk\*akj - 1 выполняется тогда и толькo тогда, когда оба элемента а^ и а^-равны 1, т.е. имеются ребрйц (Vj,Vk) и (vk,vj), следовательно, существует путь из v,-в Vy длиной 2 через v^. Тогда приведенная выше сумма равна числу различш путей длиной 2 из v, в vj через различные вершины vk. Аналогично элемент ау матрицы АТ задает число путей ДОИДь ной 3 из v, в Vj, а по матрице Ат мы можем определить число путей ДЛИНОЙ Г ИЗ V/ В Vj.

Если мы хотим определить, существует ли в графе с п верцш§! нами путь из v, в v,-, то нам необходимо проверить, имеются лЩ элементарные пути длиной, меньшей или равной (л - 1). Такие пути определяются с помощью матрицы 4П-1 Элемент by показывает число существующих путей из v,- в V/ длиной, меньшей или равной (я-1). Если элемент Ьу>0, то вершина v,достижима из v,. Матрица В"=А+А2+А\*+...А" дает возможность определить число элементарных путей и элементарных циклов в графе, представленном матрицей смежности А.

**8.4.Путевая матрица (матрица достижимости)** Если нас интересует, достижима ли вершина v, из вершины v, и не интересует число путей из вершины v, в vj, то достаточно в матрице В" все ненулевые элементы поменять на 1. Тогда получим матрицу достижимости, или путевую матрицу Рп\*„, элементы которой ру = 1, если существует путь из v, в vy, и ру = 0 в противном случае. Матрицу Р называют также транзитивным замыканием матрицы смежности.

Путевая матрица только показывает, имеется или отсутствует путь между парой вершин (или цикл в любой вершине), но не определяет все пути. Рассмотренный способ определения путевой матрицы Рп из матрицы В" громоздок. Более рациональный способ получения путевой матрицы предложен Уоршаллом [8], в котором используются реализуемые просто булевы операции. В этом алгоритме вначале путевая матрица Р устанавливается равной матрице смежности, затем за к проходов, к=1,2,...,п, в циклах по i nj выполняются булевы операции:  
pkj), i,j - 1,2, ..., п. Pk-i[i.k]

Алгоритм выполняется за п проходов. После к-го прохода ру содержит 1 или 0 в зависимости от того, имеются ли пути между вершинами i и j, которые не проходят через вершины с номерами, большими или равными к, т.е. между вершинами / и j могут быть вершины только с номерами, меньшими или равными к. Это представлено на рис. 8.7.

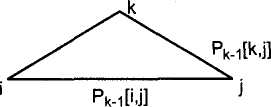


Рис. 8.7. Схема определения элемента матрицы

Таким образом, после &-го прохода путь между вершинами / и j есть тогда и только тогда, если:

1) он уже был или  
2) есть пути между вершинами / и к и одновременно между вершинами к и у, проходящие через вершины с номерами, меньшими к.

Алгоритм реализуется посредством трех вложенных циклов по к, i и j, имеет временную сложность О(п3). Ниже демонстрируется получение путевой матрицы из матрицы смежности. Здесь из-за особенностей языка Си к принимает значения 0, 1, ...,«- 1.  
Пример 8.5.  
/\* ВЫЧИСЛЕНИЕ ПУТЕВОЙ МАТРИЦЫ ПО ЗАДАННОЙ МАТРИЦЕ СМЕЖНОСТИ \*/ #include <stdio.h> void pm(int A[][5],int P[][5],int n); int main() { int  
i,j,n;  
int A[5][5] = {0,0,0,1,0, 1,0,1,0,0, 0,0,0,0,1, 0,0,1,0,1, 0,0,0,0,0}; int P[5][5]; n=5;  
printf("\n Вычисление путевой матрицы по матрице смежности"); printf("\n Матрица смежности:"); for (i=0;i<n;i++) {printf("\n"); for O=0;j  
}  
pm(A,P,n);  
printf("\n Путевая матрица:"); for (i=0;i<n;i++) for (i=0;i<n;i++) {printf("\n"); for (j=O;j  
printf("\n"); getch(); return 0;  
}  
/\* =============================================== \*/  
/\* ФУНКЦИЯ ВЫЧИСЛЕНИЯ ПУТЕВОЙ МАТРИЦЫ ПО ЗАДАННОЙ МАТРИЦЕ СМЕЖНОСТИ 7  
void pm(int A[][5],int P[][5],int n)  
/\* A — матрица смежности, Р — путевая матрица 7  
{ int i,j,k;  
for (i=0;i<n;i++)  
for (j=O;j<n;j++)  
P[i][j] = A(i]0]; /\* копирование А в Р for (k=0;k<n;k++) for (i=0;i<n;i++) for  
(j=O;j<n;j++) P[i]U] =P[i]U] II (P[i][k] && P[k]0]);

**8.5. Минимальная путевая матрица** Если по путевой матрице мы можем определить, имеется ли путь между вершинами v, и vj, то минимальная путевая матрица содержит длины кратчайших путей (количество дуг) между вершинами у,-и vj, i,j = 1,2,...,и.

Алгоритм вычисления минимальной путевой матрицы получается модификацией алгоритма Уоршалла. В матрице смежности все нулевые элементы меняются на бесконечность (число > n), а выражение для вычисления элемента минимальной путевой матрицы принимает вид:  
Pij= min {pij, (ри, +pkj)).

Алгоритм, как и при вычислении путевой матрицы, выполняется за n проходов. После к-ro прохода рц содержит минимальную длину пути (в дугах) между вершинами / и j, который не проходит через вершины с номерами, большими к. После к-то прохода за минимальную длину пути между вершинами / и у принимается более короткая из двух:

1) длина пути, которая была вычислена до этого, т.е. пути, проходящего только через вершины с номерами, меньшими к;

* 2) длина пути, проходящего через вершину к, причем пути (i-k) и (k-j) также проходят только через вершины с номерами, меньшими к.

Иногда необходимо знать не только минимальный путь между вершинами i и j, но и вершины, через которые пролегает этот путь (маршрут). Для этого дополнительно вводится матрица М„\*„, в которой элемент /гг(/ содержит номер вершины к, полу-ченный при нахождении наименьшего значения /?/,- при к-и прохождении. Если pij = 0, то это значит, что вершины i и у смежны.  
printf("\n"); } /\* Вычисление минимальной путевой матрицы \*/  
pm(A,P,M,n);  
printf("\n Минимальная путевая матрица:\п");  
for (i=0;i<n;i++)  
{for (j=O;j<n;j++)  
printf(" %d ",P[i][j]==1OOOO ? 0 : P[i][j]);  
printf("\n"); }  
/\* вывод маршрута между всеми парами вершин 7 for (i=1; i≤n; i++)  
for 0=1; j≤n; j^  
) MinPath(M,P,n,ij);  
return 0;

**8.6. Кратчайшие пути** Ha практике часто требуется найти наилучший маршрут от одной вершины графа до другой. Что означает «наилучший»? Наилучший маршрут могут определять многие факторы:

* расстояние в км;
* время прохождения маршрута;
* вероятность передачи сообщения по сетям связи;
* пункты, которые необходимо посетить, и т.д.

Мы рассматриваем задачу наилучшего маршрута в смысле кратчайшего расстояния. Эта задача моделируется с помощью связного графа G, в котором каждому ребру приписан положительный вес, равный длине ребра. Длина пути в таком графе равна сумме длин ребер, составляющих путь.

Задачи о кратчайших путях относятся к фундаментальным задачам комбинаторной оптимизации, так как многие задачи можно свести к отысканию кратчайшего пути в графе. Существуют различные типы задач о кратчайшем пути между вершинами:

* двумя заданными вершинами;
* данной вершиной и всеми остальными вершинами; каждой парой вершин и т.д.

**8.6.1. Алгоритм Дейкстры** Одним из наилучших алгоритмов отыскания кратчайшего пути между двумя вершинами является алгоритм Дейкстры [8], который по существу совпадает с алгоритмом Прима [8] отыскания остовного дерева минимального веса. Рассмотрим алгоритм Дейкстры отыскания кратчайшего пути от заданной начальной вершины v до конечной вершины w. Граф G=(V,E) имеет т вершин, вершины пронумерованы 1,2,...,т. Граф представлен взвешенной матрицей смежности Ат\*т, где ау \_ длина дуги, соединяющей вершины i и j. Если дуга отсутствует, то принимается ац - бесконечность (число, большее длины наибольшей дуги графа).

В результате вычислений получаем перечень дуг, лежащих на кратчайшем пути и представленных тройками чисел: начальная вершина, конечная вершина и длина пути от заданной вершины v до конечной вершины дуги. Фактически этот перечень можно рассматривать как дерево с корнем v, узлы дерева содержат номера вершин графа, длины означают расстояния от корня (заданной вершины v) до этих вершин. Обратный обход дерева от заданной конечной вершины w до начальной v дает перечень вершин, лежащих на искомом кратчайшем пути.

Алгоритм Дейкстры состоит в следующем.

1. Выбираем начальную вершину v, от которой до другой заданной вершины w отыскивается кратчайший путь. Остальные (т - 1) вершины включаем в перечень невыбранных вершин. Определяем расстояния от заданной начальной вершины v до всех остальных вершин.  
2. По наименьшему расстоянию выбираем новую вершину, исключив ее из перечня невыбранных. Число невыбранных вершин уменьшаем на 1. Дугу перехода, определившую наименьшее расстояние, включаем в перечень дуг кратчайшего пути.  
3. Определяем расстояния от новой выбранной вершины до всех невыбранных с учетом пройденных от начальной точки v расстояний. Если расстояние до некоторой вершины меньше ранее определенного, то заменяем его новым значением. Отбираем наименьшее расстояние и переходим к новой вершине, исключив ее из перечня невыбранных. Число невыбранных вершин уменьшаем на 1. Дугу перехода включаем в перечень дуг кратчайшего пути и т.д., пока не достигнем заданной конечной вершины w при поиске кратчайшего пути между двумя вершинами v и к- или не выберем все n вершин графа при поиске кратчайших путей от вершины v до всех остальных.

Алгоритм Дейкстры относится к «жадным» алгоритмам. **«Жадность»** заключается в том, что на каждом шаге к списку выбранных вершин добавляется та из оставшихся вершин, расстояние до которой от начальной вершины меньше, чем для других оставшихся вершин. В то же время этот алгоритм использует метод динамического программирования, так как на каждом шаге выбирается оптимальное значение, которое затем не пересчитывается.

Процесс выполнения алгоритма можно проследить на примере графа, представленного на рис. 8.8, в качестве начальной и конечной вершин выбраны вершины 1 и 8 соответственно.

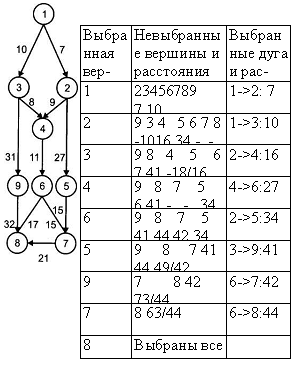


Рис. 8.8. Поиск кратчайших путей в графе

**8.6.2. Алгоритм Флойда** Иногда требуется решать общую задачу нахождения кратчайших путей, т.е. нахождения для каждой пары вершин (u,v) пути от вершины v к вершине iv. длина (расстояние) которого минимальна среди всех возможных путей от г к и\ Эту задачу можно решить, последовательно применяя алюритм Дсйкстры нахождения кратчайших путей для каждой вершины. Тогда временная сложность будет равна О(п) при использовании матрицы смежности и O(n\*e\*log(n)) при использовании списков смежности.

Существует прямой способ решения этой задачи, использующий алгоритм Флойда. Граф G=(V,E) с п вершинами пред¬ставляется взвешенной матрицей смежности Sn\*n. В результате решения задачи формируется матрица А„\*„, в которой элемент fly будет содержать значение кратчайшего пути от вершины i до вершины j.

Сначала исходная матрица S копируется в матрицу А, причем если дуга (ij) отсутст-вует, то ац - °°, а диагональные элементы ПЦ = 0. Алгоритм выполняется за п проходов. После к-то прохода fly-содержит значение наименьшей длины путей из вершины / к j, которые не проходят через вершины с номером, большим к, т.е. между вершинами i и j могут быть вершины только с номерами, меньшими или равными к. При к-м проходе элемент ац вычисляется как Ak[i,j] = mm(Ak.\[i, j], AkA[i, к] + Акл[к,Д) (см. рис. 8.7).

Алгоритм Флойда реализуется посредством трех вложенных циклов и имеет сложность О(пг). Часто необходимо знать не только кратчайшее расстояние между вершинами v и w, но и вершины, через которые пролегает кратчайший путь. Для этого дополнительно используется матрица М„\*„, в которой элемент тц содержит номер вершины к, полученный при нахождении наименьшего значения ац в к-м проходе. Если ау- = 0, то вершины i uj смежны. Ниже приводится программа нахождения кратчайших рассто¬яний по алгоритму Флойда.

**8.7. Остовные деревья графа**

**Остовным деревом** для связного неориентированного графа G = (V,E) с п вершинами называется неориентированное дерево, содержащее все и вершин и (п - 1) ребер графа. Таким образом, остовное дерево связывает все вершины графа и из каждой вершины графа можно попасть в любую другую. В полном графе с п вершинами имеется п"' остовных деревьев. Для одного и того графа можно построить несколько остовных деревьев (рис. 8.9).

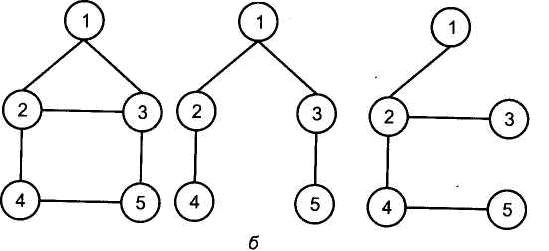


Рис. 8.9. Граф (а) и ею остовные деревья (б) и (в)

Пусть G = (V,E) — связный неориентированный граф и Т = {V,F) — остовное дерево для него. Тогда из [2]:

а) для любых двух вершин v, и у,- путь между ними единствен;  
б) если к остовному дереву Г добавить ребро из {E-F), т.е. из оставшихся, не включенных в дерево ребер графа, то возникнет ровно один цикл и Т перестанет быть деревом.

Под **остовным деревом** ориентированного графа понимают такое дерево, в котором одна из вершин графа связана со всеми остальными. Основными задачами нахождения остовных деревьев графа являются:

* нахождение остовного дерева;
* нахождение минимального и максимального остовного дерева.

**8.8. Обходы графов. Поиск в глубину и поиск в ширину**

При решении многих задач, связанных с графами, необходим систематический обход вершин и дуг (ребер) графа. Широкое применение получили два метода обхода:

* поиск в глубину;
* поиск в ширину.

При поиске в глубину осуществляется регулярный обход графа по правилам.

1. Находясь в вершине v, нужно двигаться в любую другую н>, ранее не пройденную вершину (если таковая найдется), одновременно запоминая дугу z, по которой мы впервые попали в вершину w.  
2. Если из вершины v мы не можем попасть в ранее не пройденную вершину или таковой вообще нет, то мы возвращаемся в вершину дуги z, из которой впервые попали в вершину v, и продолжим поиск в глубину из этой вершины.

При выполнении обхода по этим правилам мы стремимся проникнуть вглубь графа как можно дальше, затем отступаем на шаг и снова стремимся пройти вперед и т.д.

При поиске в глубину в ориентированном графе мы можем попасть в вершину vr из вершины v только в том случае, если в орграфе есть дуга (i',u), то есть двигаемся вперед только в направлении ориентации дуг, а возвращаемся против ориентации. В неориентированном графе таких ограничений нет. В неориентированном графе обход можно начинать с любой вершины. Топология полученного остовного дерева зависит от начальной вершины.

Рассмотрим особенности поиска в глубину в орграфе. Те дуги, которые ведут к новым вершинам, называются дугами дерева, или древесными дугами. Они формируют для данного графа либо остовное дерево (дерево поиска в глубину), либо остовный лес (глбинное остовное дерево, глубинный остовный лес). Для того чтобы после завершения обхода в глубину все вершины оказались пройденными и образовалось остовное дерево, необходимо и достаточно, чтобы обход начинался во входе графа (вершине, имеющей только исходящие дуги) и этот вход был единственным. В противном случае при обходе в глубину образуется глубинный остовный лес. Если при поиске в глубину вершины графа нумеруются в порядке их посещения, то такая нумерация называется глубинной нумерацией графа или М-нумерацией вершин графа [14]. После завершения поиска в глубину все дуги орграфа разбиваются на 4 класса:

1) класс Т (Tree) древесных дуг, порождающих дерево поиска в глубину либо глубинный остовный лес. Для каждой дуги (х,у)е Т имеем М(х)< М(у), М[г\ — номер вершины i;  
2) класс F (Forward) прямых дуг, к которому относятся дуги (х,у), такие, что М(х) < М(у) и вершина х соединена с у путем, состоящим из древесных дуг. Другими словами, прямые дуги идут эт предков к непосредственным потомкам, но не являются древесными дугами;  
3) класс В (Back) обратных дуг, представляющих дуги (х,у), гакие, что М(х) < М(у) и вершина у соединена с х путем, состояцим из древесных дуг. По-другому обратные дуги идут от потомков к предкам;  
4) класс С (Cross) поперечных дуг, представляющих собой дуги х,у), такие, что М{х) > М(у) и вершины х и у не соединены путем, «стоящим из древесных дуг, т.е. поперечные дуги соединяют вер- нины, не являющиеся ни потомками, ни предками друг друга.

Рассмотрим пример (рис. 8.10). Древесные дуги: (а, Ь), (Ь, $#60?), (е, h), (h, у), (е, d), (а, с), (г,./), <\ /), (/,#). Прямые дуги: (b. d), (с, g). Обратные дуги: (g, /), (d, b). 1онеречные дуги: {/', /;), (/', /;).

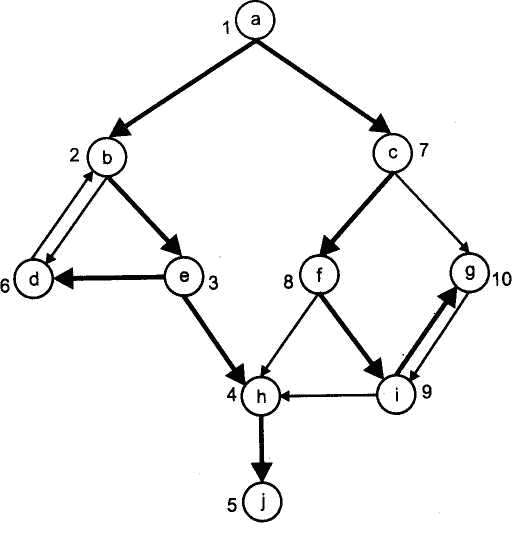


Рис. 8.10. Обход дерева в глубину

Если начать обход в глубину этого же графа, начиная с другой вершины, то образуется глубинный остовный лес. Например, обход, начиная с вершины с, приведет к образованию леса с двумя остовными деревьями. Первое дерево включает ребра (с, /), (f, h), (h,j), if, i), (i, g). Второе дерево с началом обхода в вершине а состоит из ребер (a, b), (b, d), (d, e). Для упражнения определите прямые, обратные и поперечные дуги графа, полученные в результате такого обхода.

При поиске в глубину в связном неориентированном графе образуется одно единственное глубинное дерево. В нем различают только два класса ребер: древесные и обратные. Сложность алгоритма поиска в глубину равна О(е), где е — число дуг (ребер) графа.

Ниже приводится пример вычисления остовного дерева по алгоритму поиска в глубину. Исходный граф представлен списками смежности.

**8.9.1. Алгоритм Прима** Наиболее простым алгоритмом поиска остовного дерева минимального веса является алгоритм Прима, удивительно похожий на алгоритм Дейкстры поиска кратчайшего пути. Алгоритм Прима заключается в следующем.

1. Вначале выбираем некоторую вершину v, остальные (т - 1) вершины графа отмечаются как невыбранные.  
2. Определяются веса между выбранной вершиной v и остальными невыбранными вершинами.  
3. Выбираем вершину с наименьшим весом до нее, фиксируем выбранные ребро и вес.  
4. Выбранную вершину исключаем из перечня невыбранных, число выбранных вершин уменьшаем на 1.  
5. Пункты 1 — 4 повторяем до тех пор, пока не будут выбраны все вершины, т.е. (т - 1) раз.

Процесс поиска остовного дерева наименьшего веса по алгоритму Прима можно проследить на примере графа, приведенного на рис. 8.12. Приведенная программа демонстрирует определение остовного дерева минимального веса для неориентированного графа по алгоритму Прима

**8.9. Остовное дерево наименьшей стоимости (минимального веса)** Пусть G - (V, Е) — связный взвешенный неориентированный граф, для которого задана матрица смежности, отображающая веса ребер в числа (вещественные или це-лые). Стоимость (вес) остовного дерева определяется как сумма стоимостей (весов) его ребер. Цель — найти для графа G остовное дерево наименьшей стоимости (минимального веса). Полный граф с п вершинами содержит п"'2 остовных деревьев. Поиск каждого остовного дерева занимает О(е) времени. В полном дереве е=п\*(п-\)12. Тогда решение задачи прямым поиском имело бы сложность О(п"-2 п(п - \)12)=О(пп).

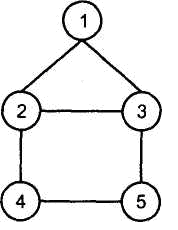


Рис. 8.11. Граф и его остовные деревья: а — исходный граф; 6 -- поиск в глубину; в— поиск в ширину

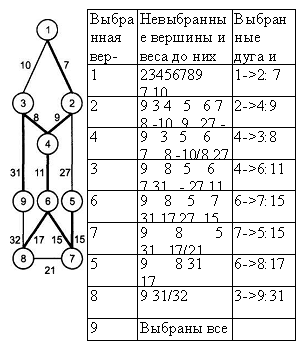


Рис. 8.12. Поиск остовного дерева наименьшего веса по алгоритму Прима

**8.9.2. Алгоритм Крускала** Пусть имеется связный взвешенный граф G( V,E) с п вершинами. Построение остовного дерева минимального веса начинается с графа Т=( V,0), имеющего только п вершин без ребер. Каждая вершина, таким образом, оказывается связанной только с самой собой. Построение дерева сводится к формированию набора связных компонент, постепенным объединением которых и формируется остовное дерево.

Упорядочим ребра множества Е в порядке возрастания их веса (стоимости). Выберем ребро с наименьшим весом С\ и включим в граф Т. Теперь в графе Т (п - 1) компонент содержит только по одной вершине, одна компонента содержит две вершины и одно ребро. Выбираем следующее наименьшее ребро. Если оно связывает две вершины из разных компонент, то это ребро добавляется в граф Т. Если же ребро связывает две вершины из одной компоненты, то такое ребро отбрасывается, так как его добавление в связную компоненту, являющуюся свободным деревом, приведет к образованию цикла. Когда все вершины графа будут принадлежать одной компоненте, построение остовного дерева минимального веса Т с (и - 1) ребрами заканчивается. Процесс построения остовного дерева минимального веса можно проследить на примере графа, приведенного на рис. 8.13. Ниже приводится программа нахождения остовного дерева минимального веса по алгоритму Крускала.

Упорядоченная таблица весов ребер графа

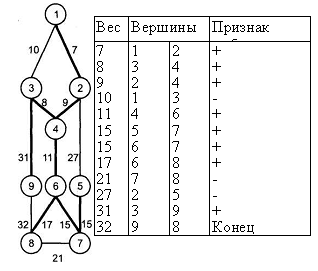


Рис. 8.13. Поиск остовного дерева наименьшего веса по алгоритму Крускала

**8.10. Упорядочение графа (топологическая сортировка)** Часто встречаются ситуации, когда необходимо решать задачу сортировки элементов множества, для которых определен частичный порядок, т.е. упорядочение задано не на всех, а только на некоторых парах элементов. Вот несколько примеров.

В толковом словаре одни слова определяются с помощью других. Топологическая сортировка слов в словаре требует, чтобы все слова, участвующие в определении данного слова, находились в словаре раньше определяемого слова. В учебных программах вузов одни предметы опираются на материал других. Топологическая сортировка означает чтение курсов в таком порядке, чтобы ни один курс не читался раньше того курса, на материале которого он основан. Сетевые методы планирования и управления предполагают, что любая работа может быть выполнена только после того, как будут выполнены опорные работы. Пусть имеется ориентированный граф G-{V,E) с произвольно занумерованными вершинами v,-, i = 1,2,...,и. Граф является упорядоченным, если в отношении каждой его дуги О,,^) справедливо неравенство v,<="" p="">

Рассмотрим граф, представленный на рис. 8.14, а). Из рисунка видно, что вершина 3 имеет ранг R=0, вершина 1 имеет ранг R=l, вершины 5 и 7 имеют ранг R=2. вершина 4 имеет ранг R=3, вершина 2 имеет ранг R=4 и вершина 6 имеет ранг R=5.

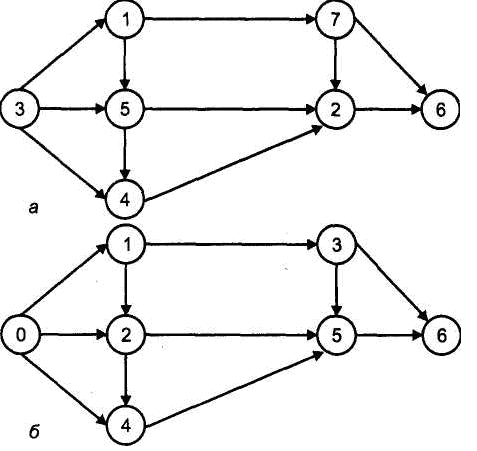


Рис. 8.14. Неупорядоченный (а) и упорядоченный (б) графы

Алгоритм упорядочения графа состоит из двух этапов. На первом этапе определяются ранги вершин, на втором этапе осуществляется перенумерация их в соответствии с рангами. Единственной вершине v,, не имеющей входов, присваивается номер 0, а затем нумерация выполняется в порядке возрастания ранга вершины, причем нумерация вершин одного ранга безразлична. Упорядоченный граф имеет вершины v,- с номерами 0,1,2,...,/! - 1. Рассмотренный выше граф после упорядочения будет иметь нумерацию вершин, как показано на рис. 8.14, б).

Таким образом, исходным номерам вершин 1, 2, 3, 4, 5, 6, 7 после упорядочения соответствуют новые номера 1, 5, 0, 4 ,2, 6, 3. Ниже приводится программа [6], использующая для упорядочения графа метод Форда.