А. С. Третьяков

АЛГОРИТМЫ & СТРУКТУРЫ ДАННЫХ

Оглавление

Предисловие

Данная книга собрана по материалам сайта Kvodo.ru, которые были отредактированы и несколько обобщены. В ней кратко описаны алгоритмы и структуры данных, наиболее часто используемые в программировании, а также конкретные способы их реализации на двух популярных языках программирования: C++ и Pascal. Книга рассчитана на читателя с некоторым базовым набором знаний. Она не претендует называться учебником, и тем более заменить общепризнанный учебный материал.

Изучать главы книги необязательно по порядку. В этом плане она может служить справочником. Большинство тем содержат рисунки, схемы и таблицы, поясняющие работу алгоритмов и структур данных.

В отличие от первоначальной версии (материала сайта), в коде отсутствуют некоторые элементы, необходимые для реального функционирования программы. Сделано это с целью «оторвать» приведенный код от конкретной среды разработки. Читателю придется самому добавлять необходимые директивы и операторы.

В разделе «Алгоритмы» каждая тема подкреплена листингом, написанном как на C++, так и на Pascal. Но в разделе «Структуры данных», в подавляющем большинстве случаев, приведен код только для C++, т. к. некоторые темы требовали значительного обилия кода, что не позволило использовать сразу два языка. Поэтому предполагается, что читатель знаком с языком C++. Изучить его можно, воспользовавшись соответствующей книгой или сетевым ресурсом, например, тем же Kvodo.ru.

Книга, с большой вероятностью, содержит ошибки. Мы будем благодарны, если Вы укажите их, воспользовавшись электронной почтой: Kvodo.ru@inbox.ru

Выражаем благодарность всем тем читателям, что давали полезные советы и находили ошибки.

Раздел I

СТРУКТУРЫ ДАННЫХ

Введение

Необходимым условием хранения информации в памяти компьютера является возможность преобразования этой самой информации в подходящую для компьютера форму. В том случае, если это условие выполнимо, следует определить структуру, пригодную именно для наличествующей информации, ту, которая предоставит требующийся набор возможностей работы с ней. Здесь под структурой понимается способ представления информации, посредством которого совокупность отдельно взятых элементов образует нечто единое, обусловленное их взаимосвязью друг с другом. Скомпонованные по каким-либо правилам и логически связанные межу собой данные могут весьма эффективно обрабатываться, так как общая для них структура предоставляет набор возможностей для работы с ними – одно из того за счет чего достигаются высокие результаты в решениях тех или иных задач.

Говоря о не вычислительной технике, можно показать ни один случай, где у информации видна явная структура. Наглядным примером служат книги самого разного содержания. Они разбиты на страницы, параграфы и главы, имеют, как правило, оглавление, то есть интерфейс пользования ими. В широком смысле, структурой обладает всякое живое существо, без нее органика навряд-ли смогла бы существовать.

Вполне вероятно, читателю приходилось сталкиваться со структурами данных непосредственно в информатике, например, с теми, что встроены в язык программирования. Часто они именуются типами данных. К таковым относятся: массивы, числа, строки, файлы и т. д.

Методы хранения информации, называемые «простыми», т. е. неделимыми на составные части, предпочтительнее изучать вместе с конкретным языком программирования, либо же глубоко углубляться в суть их работы. Поэтому здесь будут рассмотрены лишь «интегрированные» структуры, те которые состоят из простых, а именно: массивы, списки, деревья и графы.

Глава 1. Массивы

1.1 Введение

Массив – это структура данных с фиксированным и упорядоченным набором однотипных элементов (компонентов). Доступ к какому-либо из элементов массива осуществляется по имени и номеру (индексу) этого элемента. Количество индексов определяет *размерность массива*. Так, например, чаще всего встречаются *одномерные* (вектора) и *двумерные* (матрицы) массивы. Первые имеют один индекс, вторые – два. Пусть одномерный массив называется *A*, тогда для получения доступа к его *i*-ому элементу потребуется указать название массива и номер требуемого элемента: *A*[*i*]. Когда *A* – матрица, то она представляема в виде таблицы, доступ к элементам которой осуществляется по имени массива, а также номерам строки и столбца, на пересечении которых расположен элемент: *A*[*i*, *j*], где *i* – номер строки, *j* – номер столбца.

В разных языках программирования работа с массивами может в чем-то различаться, но основные принципы, как правило, везде одни. В языке Pascal, обращение к одномерному и двумерному массиву происходит точно так, как это показано выше, а, например, в C++ двумерный массив следует указывать так: *A*[*i*][*j*]. Элементы массива нумеруются поочередно. На то, с какого значения начинается нумерация, влияет язык программирования. Чаще всего этим значением является 0 или 1.

Массивы, описанного типа называются *статическими*, но существуют также массивы по определенным признакам отличные от них: *динамические* и *гетерогенные*. Динамичность первых характеризуется непостоянностью размера, т. е. по мере выполнения программы размер динамического массива может изменяться. Такая функция делает работу с данными более гибкой, но при этом приходится жертвовать быстродействием, да и сам процесс усложняется. Обязательный критерий статического массива, как было сказано, это однородность данных, единовременно хранящихся в нем. Когда же данное условие не выполняется, то массив является гетерогенным. Его использование обусловлено недостатками, которые имеются в предыдущем виде, но оно оправданно во многих случаях.

Таким образом, даже если Вы определились со структурой, и в качестве нее выбрали массив, то этого все же недостаточно. Ведь массив это только общее обозначение, род для некоторого числа возможных реализаций. Поэтому необходимо определиться с конкретным способом представления, с наиболее подходящим массивом.

1.2 Хеш-функции

Хеш-функция – функция, преобразовывающая входную последовательность данных произвольного размера в выходную последовательность фиксированного размера. Процесс преобразования данных называется *хешированием*. Результат хеширования – *хеш-код* (хеш-сумма, хеш).

При решении класса практических задач выбирается такая хеш-функция, которая является наиболее оптимальной именно для данного класса. В общем случае следует использовать «хорошую» функцию. Когда хеш-функцию называют «хорошей», то подразумевают под этим, что она:

1. вычисляется достаточно быстро;
2. сводит к минимуму число *коллизий*.

Коллизией хеш-функции называется ситуация, при которой два различных элемента имеют один и тот же хеш-код. Коллизий следует избегать, выбирая «хорошую» хеш-функцию, и используя один из методов разрешения конфликтов: *открытое* или *закрытое* хеширование (см. гл. 1.3). Предотвратить коллизии могут далеко не все хеш-функции, но «хорошие» способны минимизировать вероятность их появления. При определенных обстоятельствах (известна некоторая информация о ключах), можно найти идеальную хеш-функцию, т. е. такую, которая полностью исключает возможность появления коллизий.

Использовать хеш-функцию или нет, зависит от того насколько целесообразно применение ее свойств, а также свойств алгоритма, по которому она может быть реализована на языке программирования. В одних ситуациях наиболее важна высокая скорость работы, в других равномерное распределение хеш-кодов и т. п. Далее будут рассмотрены два наиболее известных метода хеширования.

1.2.1 Метод деления

Пусть k – ключ (тот, что необходимо хешировать), а n – максимально возможное число хеш-кодов. Тогда метод хеширования посредством деления будет заключаться во взятии остатка от деления k на n: $h(k)=k \bmod n$, где mod – операция взятия остатка от деления.

Например, на вход подаются следующие ключи:

3, 6, 7, 15, 32, 43, 99, 100, 133, 158.

Определим n равным 10, из чего последует, что возможные значения хешей лежат в диапазоне $0…n$-1. Используя данную функцию, получим следующие значения хеш-кодов:

$h(3)=3$, $h(6)=6$, $h(7)=7$, $h(15)=5$, $h(32)=2$, $h(42)=2$, $h(99)=9$, $h(100)=0$, $h(133)=3$, $h(158)=8$

На C++ программу, выполняющую хеширование методом деления, можно записать так:

```
int HashFunction(int k)
    { return (k%10); }
void main()
    {
        int key;
        cout<<"Ключ > "; cin>>key;
        cout<<"HashFunction("<<key<<")="<<HashFunction(key)<<endl;
    }
```

Во избежание большого числа коллизий рекомендуется выбирать n простым числом, и не рекомендуется степенью с основанием 2 и показателем m (2^m). Вообще, по возможности, следует выбирать n, опираясь на значения входящих ключей. Так, например если все или большинство $k=10^m$ (m – натуральное число), то неудачным выбором будет $n=10*m$ и $n=10^m$.

1.2.2 Метод умножения

Получить из исходной последовательности ключей последовательность хеш-кодов, используя метод умножения (мультипликативный метод), значит воспользоваться хеш-функцией:

$$h(k)=\lfloor n*(\{k*A\})\rfloor$$

Здесь A – рациональное число, по модулю меньшее единицы ($0<A<1$), а k и n обозначают то же, что и в предыдущем методе: ключ и размер хеш-таблицы. Также правая часть функции содержит три пары скобок:

- ( ) – скобки приоритета;
- ⌊ ⌋ – скобки взятия целой части;
- { } – скобки взятия дробной части.

Аргумент хеш-функции k ($k\geq 0$) в результате даст значение хеш-кода $h(k)=x$, лежащие в диапазоне $0...n-1$. Для работы с отрицательными числами можно x взять по модулю.

От выбора A и n зависит то, насколько оптимальным окажется хеширование умножением на определенной последовательности. Не имея сведений о входящих ключах, в качестве n следует выбрать одну из степеней двойки, т. к. умножение на 2^m равносильно сдвигу на m разрядов, что компьютером производиться быстрее. Неплохим значением для A (в общем случае) будет $(\sqrt{5}-1)/2\approx 0{,}6180339887$. Оно основано на свойствах золотого сечения:

Золотое сечение – такое деление величины на две части, при котором отношение большей части к меньшей равно отношению всей величины к ее большей части.

Отношение большей части к меньшей, выраженное квадратичной иррациональностью:

$\varphi=(\sqrt{5}+1)/2\approx1{,}6180339887$

Для мультипликативной хеш-функции было приведено обратное отношение:

$1/\varphi=(\sqrt{5}-1)/2\approx0{,}6180339887$

При таком A, хеш-коды распределяться достаточно равномерно, но многое зависит от начальных значений ключей.

Для демонстрации работы мультипликативного метода, положим n=13, A=0,618033. В качестве ключей возьмем числа: 25, 44 и 97. Подставим их в функцию:

1. $h(k)=\lfloor 13*(\{25*0{,}618033\})\rfloor=\lfloor 13*\{15{,}450825\}\rfloor=\lfloor 13*0{,}450825\rfloor=\lfloor 5{,}860725\rfloor=5$
2. $h(k)=\lfloor 13*(\{44*0{,}618033\})\rfloor=\lfloor 13*\{27{,}193452\}\rfloor=\lfloor 13*0{,}193452\rfloor=\lfloor 2{,}514876\rfloor=2$
3. $h(k)=\lfloor 13*(\{97*0{,}618033\})\rfloor=\lfloor 13*\{59{,}949201\}\rfloor=\lfloor 13*0{,}949201\rfloor=\lfloor 12{,}339613\rfloor=12$

Реализация метода на C++:

```
int HashFunction(int k)
	{
		int n=13; double A=0.618033;
		int h=n*fmod(k*A, 1);
		return h;
	}
void main()
	{
		int key;
		cout<<"Ключ > "; cin>>key;
		cout<<"HashFunction("<<key<<")="<<HashFunction(key)<<endl;
	}
```

1.3 Хеш-таблица

Хеш-таблицей называется структура данных, предназначенная для реализации ассоциативного массива, в котором адресация реализуется посредством хеш-функции. *Хеш-функция* – это функция, преобразующая ключ *key* в некоторый индекс *i* равный *h*(*key*), где *h*(*key*) – хеш-значение (хеш-ключ) *key*. Весь процесс получения индексов хеш-таблицы называется *хешированием*.

Хеш-таблица позволяет организовать массив, специфика которого проявляется в связанности индексов по отношению к хеш-функции. Индексы могут быть не только целого типа данных (как это было в простых массивах), но и любого другого, для которого вычислимы хеш-значения. Данные, хранящиеся в виде такой структуры, удобны в обработке: хеш-таблица позволяет за минимальное время ($O(1)$) выполнять операции поиска, вставки и удаления элементов.

Предположим, имеется товарная накладная с информацией о датах и городах (когда и куда отправляются товары). Определенному товару соответствует его числовой код в диапазоне от 000000000 до 999999999:

Код	Дата	Город доставки
000000000	2016-03-10	Тула
……	……	……
000142426	2016-09-23	Москва
000142427	2018-08-25	Орел
……	……	……
603603003	2017-05-14	Казань
603603004	2019-02-018	Крым
……	……	……
999999999	2026-09-01	Хабаровск

Здесь ключами являются коды товаров. Такая организация данных не оптимальна, поскольку памяти для хранения списка выделяется больше чем нужно. Решением проблемы будет хеширование ключей списка. Хешировать можно те же коды или другие данные, например, даты. У девятизначных кодов ключи имеют целочисленный тип данных, а у дат – символьные (строковые). Для хеширования понадобиться хеш-функция. В качестве последней можно взять, например, следующую хеш-функцию:

```
string hashf(string key)
    {
        string h=key.erase(0, 5);
        return (h);
    }
```

Формальный параметр *key* это строка, состоящая из 10 символов, т. е. дата. Функция *erase*() удаляет из строки *key* 5 символов, после чего результат вида «месяц-число» присваивается переменной *h*, которая и будет индексом нового списка. Например, из ключа 2026-09-01, путем хеширования, получился хеш-ключ 09-01. Таким образом, размер накладной заметно сокращается, исчезают все пустые элементы. Но в некоторых местах возможно возникновения проблем следующего типа: пусть имеется два ключа: 2017-06-06 и 2025-06-06, тогда результат работы функции в двух случаях будет 06-06, следовательно, разные значения в хеш-таблицы попадут под один и тот же индекс, т.е. возникнет коллизия.

1.3.1 Открытое хеширование

Принцип организации хеш-таблицы методом открытого хеширования заключается в реализации связанных цепочек, начинающихся в ячейках хеш-таблицы. Под цепочками подразумеваются связанные списки, указатели на которые хранятся в ячейках хеш-таблицы. Каждый элемент связанного списка – блок данных, и если с некоторым указателем, хранящимся по адресу *i*, связаны *n* таких блоков (*n*>1), то это свидетельствует о том, что *n* ключей получили одно и то же хеш-значение *i*, т. е. имеет место коллизия. Но метод открытого хеширования устраняет конфликт, поскольку данные хранятся не в самой таблице, а в связных списках, которые увеличиваются при возникновении конфликта.

Ниже (рис. 1.1) изображены связанные списки со ссылающейся на них хеш-таблицей размером *m*. Первый столбец таблицы содержит хешированные значения ключей, второй – ссылки на списки. Количество списков ограничено лишь числом элементов исходного массива (он не показан, но предполагается). Состоят списки из трех (последний элемент подсписка – из двух) полей: & - адрес элемента списка, $ - данные, * - указатель на следующий список.

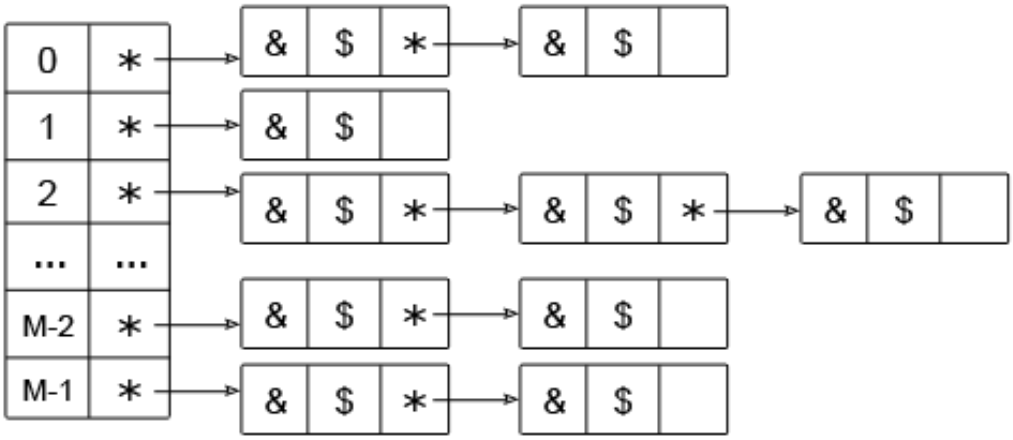


Рисунок 1.1 – Хеш-таблица

Если в исходном массиве было всего n элементов (столько же будут содержать в совокупности и все списки), то средняя длина списков будет равна $\alpha=n/m$, где m – число элементов хеш-таблицы, α – коэффициент заполнения хеш-таблицы. Предположив, например, что в списке на рисунке выше m=5 (заклеив 4-ую по счету строку), получим среднее число списков α=2.

Чтобы увеличить скорость работы операций поиска, вставки и удаления нужно, зная n, подобрать m примерно равное ему, т. к. тогда α будет равняться 1-ому или ≈1, следовательно, можно рассчитывать на оптимальное время, в лучшем случае равное $O(1)$. В худшем случае все n элементов окажутся в одном списке, и тогда, например, операция нахождения элемента (в худшем случае) потребует $O(n)$ времени.

1.3.2 Закрытое хеширование

Первый метод назывался открытым, потому что он позволял хранить сколь угодно много элементов, а при закрытом хешировании их количество ограниченно размером хеш-таблицы. В отличие от открытого хеширования закрытое не требует каких-либо дополнительных структур данных. В ячейках таблицы хранятся не указатели, а элементы исходного массива, доступ к каждому из которых осуществляется по хеш-значению ключа, при этом одна ячейка может содержать только один элемент.

Сам процесс заполнения хеш-таблицы с использованием алгоритма закрытого хеширования осуществляется следующим образом:

1. имеется изначально пустая хеш-таблица T размера m, массив A размера n ($m \geq n$) и хеш-функция $h()$, пригодная для обработки ключей массива A;
2. элемент x_i, ключ которого key_i, помещается в одну из ячеек хеш-таблицы, руководствуясь следующим правилом:
    - если $h(key_i)$ – номер свободной ячейки таблицы T, то в последнюю записывается x_i;

- если $h(key_i)$ – номер уже занятой ячейки таблицы T, то на занятость проверяется другая ячейка, если она свободна, то x_i заноситься в нее, иначе вновь проверяется другая ячейка, и так до тех пор, пока не найдется свободная или окажется, что все m ячеек таблицы заполнены.

Последовательность, в которой просматриваются ячейки хеш-таблицы, называется *последовательностью проб*. Последовательность проб задается специальной функцией, например интервал между просматриваемыми ячейками может вычисляться линейно, или увеличиваться на некоторое изменяющееся значение.

Рассмотрим метод закрытого хеширования на примере построения хеш-таблицы. Положим, имеется целочисленный массив A, состоящий из 9 элементов:

{$A[key]=data$, здесь key – ключ, $data$ – некоторые данные}

$A[13]=8$, $A[56]=4$, $A[79]=1$, $A[37]=5$, $A[41]=2$, $A[76]=9$, $A[51]=3$, $A[93]=9$, $A[30]=1$

Также есть хеш-таблица размера $m=10$, и хеш-функция $h(key)=key \ \% \ m$ (% – операция «остаток от деления»). Заполним хеш-таблицу элементами массив A:

номер элемента	0	1	2	3	4	5	6	7	8	9
key	30	41	51	13	93		56	37	76	79
data	1	2	3	8	9		4	5	9	1
проба	1	1	2	1	2		1	1	3	1

Для расстановки элементов использовалась выбранная формула. Подставив ключ, например первого элемента в нее получим: $h(13) = 13 \ \% \ 10 = 3$, поэтому его номер в хеш-таблице 3. Последовательное добавление элементов приведет к возникновению коллизии при обработке элемента $A[76]$. Хеш-значение его ключа 6, но в хеш-таблице ячейка с таким номером уже занята. Используя формулу *линейного пробирования* (один из типов последовательности проб) $h_i(key)=(h(key) + i) \ \% \ m$ (i – число проверок, после первой проверки $i=0$), продолжим поиск свободной ячейки. Применим функцию при $i=1$: $h_1(76)=7$; убедившись, что ячейка 7 занята, продолжаем поиск, увеличив i на 1: $h_2(76)=8$. Ячейка 8 свободна, помещаем в нее элемент. Этот же метод используем и для всех остальных элементов.

1.4 Ассоциативный массив

Для доступа к элементам индексного массива используются обычные целые числа, называемые индексами. У *ассоциативного массива* (*словаря*) эту функцию выполняют ключи. Они, в отличие от индексов, могут быть заданы не только числовым типом данных, но и, например строковым или булевым. Каждому элементу ассоциативного массива соответствует пара «ключ-значение» (*key*, *value*), и на нем определены четыре базовые операции:

- INSERT – операция добавления пары в массив;
- REASSIGN – операция изменения существующей пары;
- DELETE – операция удаления пары из массива;
- SEARCH – операция поиска пары в массиве.

Данный список является стандартным, но все же он может быть дополнен некоторыми другими операциями. Также стоит отметить, что здесь приведены не общепринятые названия операций: обычно они зависят от языка программирования, на котором реализуются, да и вообще термин, обозначающий ассоциативный массив, варьируется в зависимости все от того же языка. Три наиболее распространенных названия:

- словарь (Objective-C, .NET, Python, …);
- карта (C++, Java, Haskell, …);
- хеш (Perl, Ruby, …).

В одних языках программирования высокого уровня реализована встроенная поддержка ассоциативных массивов (Perl, PHP, Python, Ruby, JavaScript), в других они поставляются со специальными библиотеками (C++). Если же язык не имеет необходимых средств, то работу ассоциативного массива, как правило, можно имитировать при помощи других языковых компонентов.

В качестве примера объекта, организованного по принципу ассоциативного массива, можно привести тетрадь или базу данных, служащую для записи студентов, выписавших книги в университетской библиотеке, и пока не вернувших их. Ключами здесь будут названия книг, значениями – имена студентов. Так одно значение может соответствовать нескольким ключам (при этом ключи не повторяются), т. е. одну книгу может выписать только один студент, но этот же студент вправе выписать и другие книги.

Исходный код интерфейса ассоциативного массива, на тех языках программирования, в которых имеются встроенные средства (или подключаемые библиотеки) работы с этим типом данных, как правило, получается достаточно компактным и легко читаемым. Так реализация консольного приложения, обслуживающего университетскую библиотеку, на C++ сводится примерно к следующему варианту:

```
#include <map>
int main()
	{
		string name, book, number;
		map <string, string> library;
		do
			{
				cout<<"\n\nВыберите операцию:"<<endl;
				cout<<"\n1. Добавить запись\n2. Найти запись\n";
				cout<<"3. Удалить запись\n4. Вывести весь список\n5. Выйти\n\n";
				cout<<"Номер операции > "; cin>>number;
				if (number=="1")
					{
						cin.sync(); // очистка буфера
						cout<<"Название книги > "; getline(cin, book);
						cout<<"Имя студента > ";   getline(cin, name);
						library[book]=name;
						cout<<"Добавлено..."<<endl<<endl;
					}
				else if (number=="2")
					{
						cin.sync(); // очистка буфера
						cout<<"Книга > "; getline(cin, book);
						map <string,
						string>::const_iterator ifind=library.find(book);
						if (ifind!=library.end())
							{
								cout<<"Книга: <"<<ifind->first<<"> | ";
								cout<<"Студент: <"<<ifind->second<<">\n";
							}
						else cout<<"Данные отсутствуют";
					}
				else if (number=="3")
					{
						cin.sync(); // очистка буфера
						cout<<"Книга > "; getline(cin, book);
						library.erase(book);
						cout<<"\nДанные удалены..."<<endl;
					}
				else if (number=="4")
					{
						map <string, string>::const_iterator i;
						for(i=library.begin(); i!=library.end(); ++i)
							{
								cout<<"Книга: <"<<i->first<<"> | ";
								cout<<"Студент: <"<<i->second<<">\n";
							}
					}
				else if (number=="5") return 0;
				else cerr<<"\nНеопознанная команда"<<endl;
```

```
        } while (number!="5");
    }
```

Здесь за весь функционал ассоциативного массива отвечает контейнер <map> библиотеки STL. Для его работы, вначале необходимо подключить файл map к программе посредством директивы #include:

#include <map>

Затем описать сам массив:

map <*тип данных ключей*, *тип данных значений*> *название массива*;

Для работы с массивом, контейнер <map> предоставляет определенное количество специальных функций, но в программе выше, использовались лишь некоторые из них.

Фактически ассоциативный массив базируется на другой структуре данных. В зависимости от языка программирования, он имеет различную реализацию. Например, в языках Ruby и Python ассоциативные массивы используют хеш-таблицу, а контейнер <map> в C++ выполнен на основе красно-чёрного дерева. Другие реализации оперируют сортированными и несортированными связными списками. Также возможен вариант, в котором используется массив фиксированного размера с указателем на последний заполненный элемент этого массива.

1.5 Динамический массив

Массив, размер которого может изменяться во время работы программы, называется динамическим. Иначе говоря, *динамический массив* – это структура данных переменного размера, позволяющая в процессе выполнения программы автоматически добавлять и удалять элементы. Его максимальный размер может быть определен константой или определяться во время выполнения программы. В первом случае, динамический массив строиться на базе статического массива, а во втором – переменного, то есть такого, размер которому назначается во время работы программы. Бывает, что массив переменной длины принимают за динамический массив. Но часто это не так, поскольку размер второго, как было отмечено, может изменяться автоматически, что неверно для первого. Например, объявлен массив *A*, размером *n*, где *n* – неинициализированная переменная, т. е. на момент начала выполнения программы, *n* не присвоена никакая величина. Предполагается, что ее значение, а вместе с тем и размер массива, вводиться пользователем. Данная ситуация говорит о том, что *A* – массив переменной длины, но это не означает, что он обязательно является динамическим, хотя, как уже отмечалось, такое возможно.

Реализация простого динамического массива основывается на следующих манипуляциях. Выделяется массив фиксированного размера. Он делиться на две части: первая хранит элементы динамического массива, а вторая является резервным пространством, которое, по мере необходимости, может быть задействовано. Такая организация, позволяет добавлять элементы в конец динамического массива, а также удалять их, причем делать это за константное время. Если массив заполнен полностью (заполнены обе части исходного массива), то говорят, что исчерпан физический размер; количество элементов, используемых для динамического содержимого массива, называют логическим размером.

Большинство актуальных языков программирования высокого уровня поддерживают динамические массивы. Для работы с последними в них имеются встроенные операторы или функции, а некоторые языки программирования предоставляют несколько вариантов их реализации. Например, C++ позволяет использовать для этого:

- функции malloc, calloc, realloc и free;
  Пример создания и очистки целочисленного динамического массива, состоящего из 33 элементов:

```
int *A = malloc(sizeof(int) *33);      //создание
free(A);                               //удаление
```

- операторы new и delete (с их помощью можно только односторонне изменить размер массива – полностью очистить его);
  Пример создания и очистки вещественного динамического массива, состоящего из 11 элементов:

```
float *A=new float[10];     //создание
delete []A;                 //удаление
```

- класс vector.
  Пример создания и очистки целочисленного динамического массива, состоящего из 22 элементов:

```
vector<int> A(22);  //создание
A.clear();          //удаление
```

Выше, на примерах были продемонстрированы три способа создания и очистки динамического массива в C++. Но это не весь функционал языка. Так, например, класс vector обладает набором методов для доступа к элементам, добавления и удаления их и т. п.

В Pascal динамический массив можно определить двумя способами. Первый предполагает использование процедуры new, а второй процедуры SetLength. В обоих случаях вначале объявляется массив:

var <*имя массива*>: array of <*тип данных*>;

После чего, в основном блоке программы используется одна из процедур:

<*имя массива*>:=new <*имя типа*>[<*значение*>]

или

SetLength(<*имя массива*>, <*значение*>);

Процедуры SetLength и new резервируют место в памяти под динамическую переменную (в нашем случае под динамический массив).

Обычный статический массив выигрывает у динамического в быстродействии, что компенсируется расширенным функционалом последнего. В общем случае, временная сложность основных операций над элементами динамического массива следующая:

- Доступ к элементу по его индексу осуществляется за константное время $O(1)$;
- Вставка или удаление элемента:
  - начало массива: $O(n)$ (линейное время);
  - середина массива: $O(n)$ (линейное время);
  - конец массива: $O(1)$ (константное время).

Глава 2. Списки

2.1 Введение

Список – абстрактный тип данных, реализующий упорядоченный набор значений. Списки отличаются от массивов тем, что доступ к их элементам осуществляется последовательно, в то время как массивы – структура данных произвольного доступа. Список имеет несколько реализаций в виде структур данных, например, таких как: связный список, стек, очередь и дек.

Связный список – это структура данных, представляющая собой конечное множество упорядоченных элементов, связанных друг с другом посредствам указателей. Каждый элемент структуры содержит поле с какой-либо информацией, а также указатель на следующий элемент. В отличие от массива, к элементам списка нет произвольного доступа.

По типу связности выделяют *односвязные*, *двусвязные*, *XOR-связные*, *кольцевые* и некоторые другие виды списков.

В *односвязном списке* (рис. 2.1) начальным элементом является *Head list* (*голова списка* [произвольное наименование]), а все остальное называется *хвостом*. Хвост списка составляют элементы, разделенные на две части: информационную (поле *info*) и указательную (поле *next*). В последнем элементе вместо указателя, содержится признак конца списка – *nil*.

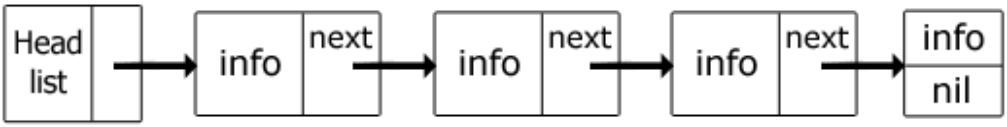


Рисунок 2.1 – Односвязный список

Односвязный список не слишком удобен, т. к. из одной точки есть возможность попасть лишь в следующую точку, двигаясь тем самым в конец списка. Когда кроме указателя на следующий элемент есть указатель и на предыдущий, то такой список называется *двусвязным* (рис. 2.2).

Возможность двигаться как вперед, так и назад полезна для выполнения некоторых операций, но дополнительные указатели требуют задействования большего количества памяти, чем таковой необходимо в эквивалентном односвязном списке.

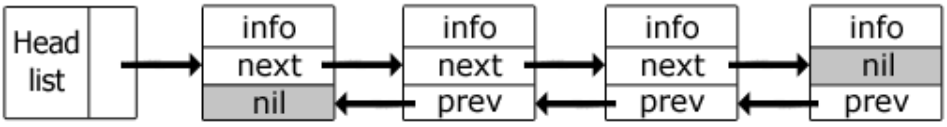


Рисунок 2.2 – Двусвязный список

Для двух видов списков описанных выше существует подвид, называемый *кольцевым* (*циклическим*) *списком* (рис. 2.3). Сделать из односвязного списка кольцевой можно добавив всего лишь один указатель в последний элемент, так чтобы он ссылался на первый. А для двусвязного потребуется два указателя: на первый и последний элементы.

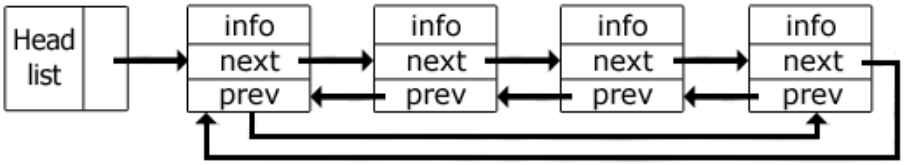


Рисунок 2.3 – Кольцевой связный список

В каждом элементе *XOR-связного списка* хранится только один адрес. Он определяется выполнением над адресами логической операции XOR (исключающее «ИЛИ», строгая дизъюнкция). Например, имеем список элементов: *A*, *B*, *C*, *D*, *E*. Чтобы узнать куда двигаться из элемента C, необходимо выполнить над адресами элементов B и D операцию XOR.

Помимо рассмотренных видов списочных структур есть и другие способы организации данных по типу «список», но они, как правило, во многом схожи с разобранными, поэтому здесь они будут опущены.

Кроме различия по связям, списки делятся по методам работы с данными. О некоторых таких методах сказано далее.

Стек характерен тем, что получить доступ к его элементом можно лишь с одного конца, называемого вершиной стека, иначе говоря: стек – структура данных, функционирующая по принципу LIFO (last in — first out, «последним пришёл — первым вышел»). Изобразить эту структуру данных лучше в виде вертикального списка (рис. 2.4), например, стопки каких-либо вещей, где чтобы воспользоваться одной из них нужно поднять все те вещи, что лежат выше нее, а положить предмет можно лишь на вверх стопки.

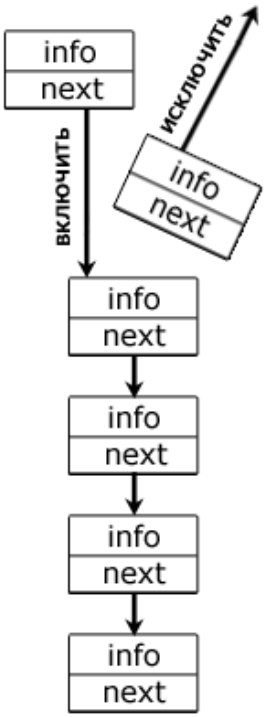


Рисунок 2.4 – Стек

В показанном на рисунке односвязном списке операции над элементами происходят строго с одного конца: для включения нужного элемента в пятую по счету ячейку необходимо исключить тот элемент, который занимает эту позицию. Если бы было, например 6 элементов, а вставить конкретный элемент требовалось также в пятую ячейку, то исключить бы пришлось уже два элемента.

Структура данных *Очередь* (рис. 2.5) использует принцип организации FIFO (First In, First Out — «первым пришёл — первым вышел»). В некотором смысле такой метод более справедлив, чем тот, по которому функционирует стек, ведь простое правило, лежащее в основе привычных очередей в различные магазины, больницы считается вполне справедливым, а именно оно является базисом этой структуры. Пусть данное наблюдение будет примером. Строго говоря, очередь – это список, добавление элементов в который допустимо, лишь в его конец, а их извлечение производиться с другой стороны, называемой началом списка.

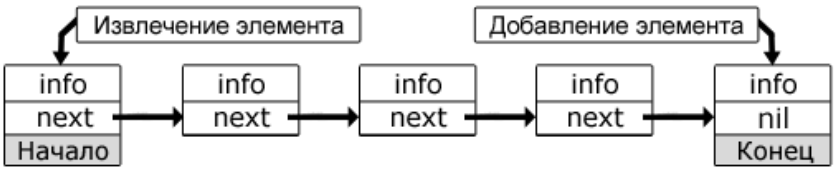


Рисунок 2.5 – Очередь

Элементу необходимо пройти весь список, прежде чем он станет доступным. В этом отношении работа со следующей структурой покажется несколько более удобной.

Дек (deque — double ended queue, «двухсторонняя очередь») – стек с двумя концами (рис. 2.6). Действительно, несмотря конкретный перевод, дек можно определять не только как двухстороннюю очередь, но и как стек, имеющий два конца. Это означает, что данный вид списка позволяет добавлять элементы в начало и в конец, и то же самое справедливо для операции извлечения.

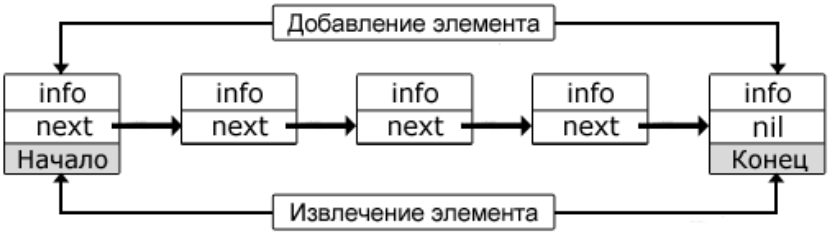


Рисунок 2.6 – Дек

Эта структура одновременно функционирует по двум способам организации данных: FIFO и LIFO. Поэтому ее допустимо отнести к отдельной программной единице, полученной в результате суммирования двух предыдущих видов списка.

2.2 Связный список

Структура данных, представляющая собой конечное множество упорядоченных элементов (узлов), связанных друг с другом посредством указателей, называется *связным списком*. Каждый элемент связного списка содержит поле с данными, а также указатель (ссылку) на следующий и/или предыдущий элемент. Эта структура позволяет эффективно выполнять операции добавления и удаления элементов для любой позиции в последовательности. Причем это не потребует реорганизации структуры, которая бы потребовалась в массиве. Минусом связного списка, как и других структур типа «список», в сравнении его с массивом является отсутствие возможности работать с данными в режиме произвольного доступа, т. е. список – структура последовательно доступа, в то время как массив – произвольного. Последний недостаток снижает эффективность ряда операций.

Каждый узел односвязного (однонаправленного связного) списка содержит указатель на следующий узел (рис. 2.7). Из одной точки можно попасть лишь в следующую точку, двигаясь тем самым в конец. Так получается своеобразный поток, текущий в одном направлении.

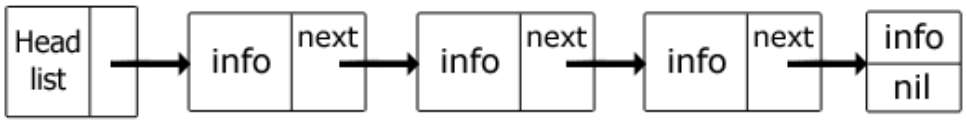


Рисунок 2.7 – Односвязный список

Повторим уже ранее сказанное. На изображении каждый из блоков представляет элемент (узел) списка. *Head list* – заголовочный элемент списка (для него предполагается поле *next*). Он не содержит данные, а только ссылку на следующий элемент. На наличие данных указывает поле *info*, а ссылки – поле *next* (далее за ссылки будет отвечать и поле *prev*). Признаком отсутствия указателя является поле *nil*.

Односвязный список не самый удобный тип связного списка, т. к. из одной точки можно попасть лишь в следующую точку, двигаясь тем самым в конец. Когда кроме указателя на следующий элемент есть указатель на предыдущий, то такой список называется двусвязным.

Та особенность двусвязного списка, что каждый элемент имеет две ссылки: на следующий и на предыдущий элемент, позволяет двигаться как в его конец, так и в начало (рис. 2.8). Операции добавления и удаления здесь наиболее эффективны, чем односвязном списке, поскольку всегда известны адреса тех элементов списка, указатели которых направлены на изменяемый элемент. Но добавление и удаление элемента в двусвязном списке, требует изменения большого количества ссылок, чем этого потребовал бы односвязный список.

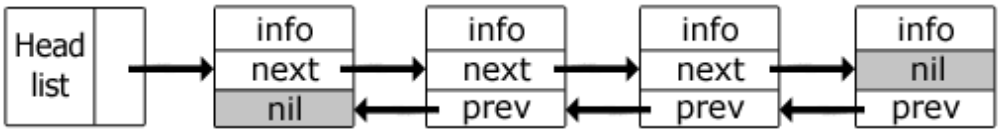


Рисунок 2.8 – Двусвязный список

Возможность двигаться как вперед, так и назад полезна для выполнения некоторых операций, но дополнительные указатели требуют задействования большего количества памяти, чем таковой необходимо в односвязном списке.

Когда количество памяти, по каким-либо причинам, ограничено, тогда альтернативой двусвязному списку может послужить XOR-связный список (рис. 2.9). Последний использует логическую операцию XOR (исключающее «ИЛИ», строгая дизъюнкция), которая для двух переменных возвращает истину лишь при условии, что истинно только одно из них, а второе, соответственно, ложно. Таблица истинности операции:

a	b	a ⊕ b
0	0	0
0	1	1
1	0	1
1	1	0

В качестве переменных *a* и *b* у нас выступают адреса (на следующий и предыдущий элемент), над которыми выполняется операция XOR, возвращающая реальный адрес следующего элемента.

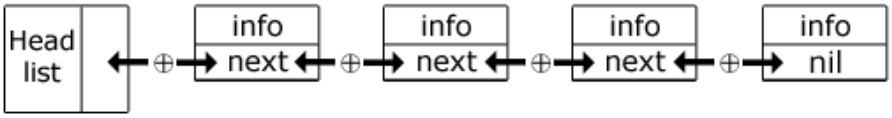


Рисунок 2.9 – XOR-связный список

Еще один вид связного списка – кольцевой список. В кольцевом односвязном списке последний элемент ссылается на первый. В случае двусвязного кольцевого списка, плюс к этому, первый элемент ссылается на последний (рис. 2.10). Таким образом, получается зацикленная структура.

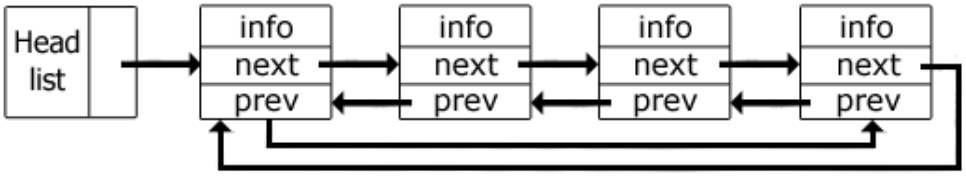


Рисунок 2.10 – Кольцевой связный список

Рассмотрим основные операции над связными списками.

На примере двусвязного списка, разберем принцип работы этой структуры данных. При реализации списка удобно использовать структуры (в Pascal – записи).

Общая форма описания узла двунаправленного связного списка и указателя на первый элемент списка:

```
struct имя_списка
    {
       информационное поле 1;
       информационное поле 2;
       …
       информационное поле n;
       указатель на следующий элемент;
       указатель на предыдущий элемент;
    };
имя_списка *указатель_на_голову_списка
```

Пример:

```
struct DoubleList                  //описание узла списка
    {
          int data;                //информационное поле
          DoubleList *next;        //указатель на следующий элемент
          DoubleList *prev;        //указатель на предыдущий элемент
    };
DoubleList *head;                  //указатель на первый элемент списка
```

Теперь, предполагая, что описан узел и указатель на голову списка (см. пример выше), напишем функции обработки списка.

Опишем функцию *AddList*, которая в качестве параметров принимает значение будущего узла и его адрес, после чего создает в списке заданный узел:

```
void AddList(int value, int position)
    {
          DoubleList *node=new DoubleList;     //создание нового элемента
          node->data=value;                    //присвоение элементу значения
          if (head==NULL)                      //если список пуст
```

```
            {
                node->next=node;            //установка указателя next
                node->prev=node;            //установка указателя prev
                head=node;                  //определяется голова списка
            }
        else
            {
                DoubleList *p=head;
                for(int i=position; i>1; i--) p=p->next;
                p->prev->next=node;
                node->prev=p->prev;
                node->next=p;               //добавление элемента
                p->prev=node;
            }
        cout<<"\nЭлемент добавлен...\n\n";
    }
```

Функция *DeleteList* для удаления элемента использует его адрес:

```
int DeleteList(int position)
    {
        if (head==NULL) { cout<<"\nСписок пуст\n\n"; return 0; }
        if (head==head->next)               //если это последний элемент в списке
            {
                delete head;                //удаление элемента
                head=NULL;
            }
        else
            {
                DoubleList *a=head;
                for (int i=position; i>1; i--) a=a->next;
                if (a==head) head=a->next;
                a->prev->next=a->next;//удаление элемента
                a->next->prev=a->prev;
                delete a;
            }
        cout<<"\nЭлемент удален...\n\n";
    }
```

Если список пуст, то выводится сообщение, извещающее об этом, и функция возвращается в место вызова.

Функция *PrintList* выводит на экран все элементы списка:

```
void PrintList()
    {
        if (head==NULL) cout<<"Список пуст\n";
        else
            {
```

```
                DoubleList *a=head;
                cout<<"\nЭлементы списка: ";
                do
                        {
                                cout<<a->data<<" ";
                                a=a->next;
                        } while(a!=head); cout<<"\n\n";
        }
}
```

Вывести список можно и посредством цикла, то есть итеративно. Теперь соединим эти три функции в одной программе, а также напишем главную функцию, отвечающую за вызов подпрограмм:

```
struct DoubleList                       //описание узла списка
    {
        int data;                       //информационное поле
        DoubleList *next;               //указатель на следующий элемент
        DoubleList *prev;               //указатель на предыдущий элемент
    };
DoubleList *head;                       //указатель на первый элемент списка

void AddList(int value, int position)   //добавление элемента
    {
        DoubleList *node=new DoubleList; //создание нового элемента
        node->data=value;               //присвоение элементу значения
        if (head==NULL)                 //если список пуст
            {
                node->next=node;        //установка указателя next
                node->prev=node;        //установка указателя prev
                head=node;              //определяется голова списка
            }
        else
            {
                DoubleList *p=head;
                for(int i=position; i>1; i--) p=p->next;
                p->prev->next=node;
                node->prev=p->prev;
                node->next=p;           //добавление элемента
                p->prev=node;
            }
        cout<<"\nЭлемент добавлен...\n\n";
    }

int DeleteList(int position)            //удаление элемента
    {
        if (head==NULL) { cout<<"\nСписок пуст\n\n"; return 0; }
        if (head==head->next)           //если это последний элемент в списке
            {
```

```
                delete head;            //удаление элемента
                head=NULL;
        }
    else
        {
            DoubleList *a=head;
            for (int i=position; i>1; i--) a=a->next;
            if (a==head) head=a->next;
            a->prev->next=a->next;//удаление элемента
            a->next->prev=a->prev;
            delete a;
        }
    cout<<"\nЭлемент удален...\n\n";
}

void PrintList()        //вывод списка
{
    if (head==NULL) cout<<"Список пуст\n";
    else
        {
            DoubleList *a=head;
            cout<<"\nЭлементы списка: ";
            do
                {
                    cout<<a->data<<" ";
                    a=a->next;
                } while(a!=head); cout<<"\n\n";
        }
}

void main()     //главная функция
{
    int value, position, x;
    do
        {
            cout<<"1. Добавить элемент"<<endl;
            cout<<"2. Удалить элемент"<<endl;
            cout<<"3. Вывести список"<<endl;
            cout<<"0. Выйти"<<endl;
            cin>>x;
            switch (x)
                {
                    case 1:
                    cout<<"Значение > "; cin>>value;
                    cout<<"Позиция > "; cin>>position;
                    AddList(value, position); break;
                    case 2:
                    cout<<"Позиция > "; cin>>position;
                    DeleteList(position); break;
                    case 3: PrintList(); break;
```

```
                    }
                } while (x!=0);
        }
```

От программы требуется, чтобы она выполнялась до тех пор, пока не выполнен выход из цикла do функции main (для выхода из него пользователь должен ввести 0). По этой причине в конце функции main не был описан оператор, тормозящий программу.

2.3 Стек

Стек характерен тем, что получить доступ к его элементам можно лишь с одного конца, называемого вершиной стека; иначе говоря: стек – структура данных типа "список", функционирующая по принципу LIFO (last in — first out, «последним пришёл — первым вышел»). Графически его удобно изобразить в виде вертикального списка (рис. 2.11), например, стопки книг, где чтобы воспользоваться одной из них, и не нарушить установленный порядок, нужно поднять все те книги, что лежат выше нее, а положить книгу можно лишь поверх всех остальных.

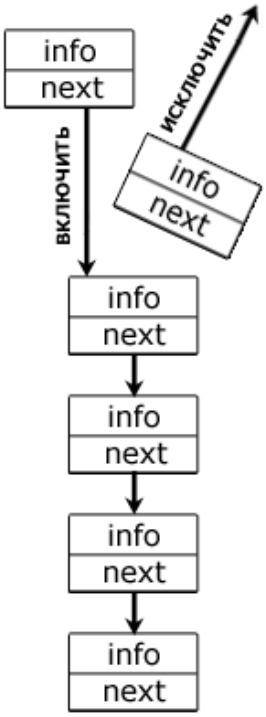


Рисунок 2.11 – Стек

Впервые стек был предложен в 1946 году Аланом Тьюрингом, как средство возвращения из подпрограмм. В 1955 году немцы Клаус Самельсон и Фридрих Бауэр из Технического университета Мюнхена использовали стек для перевода языков программирования и запатентовали идею в 1957 году. Но международное признание пришло к ним лишь в 1988 году.

На рисунке показан стек, операции над элементами которого, происходят строго с одного конца: для включения нужного элемента в *n*-ую ячейку, необходимо сдвинуть *n*-1 элементов, и исключить тот элемент, который занимает *n*-ую позицию.

Стек, чаще всего, реализуется на основе обычных массивов, односвязных и двусвязных списков. В зависимости от конкретных условий, выбирается одна из этих структур данных.

Основными операциями над стеками являются:

- добавление элемента;
- удаление элемента;
- чтение верхнего элемента.

В языках программирования эти три операции, обычно дополняются и некоторыми другими. Вот, например, список функций C++ для работы со стеком:

- push() – добавить элемент;
- pop() – удалить элемент;
- top() – получить верхний элемент;
- size() – размер стека;
- empty() – проверить стек на наличие элементов.

Данные функции входят в стандартную библиотеку шаблонов C++ – STL, а именно в контейнер stack. Далее будет рассмотрен пример работы с контейнером stack, а пока разберем стек, реализованный на основе массива.

Программа, имитирующая интерфейс стека, основанного на базе статического массива, будет состоять из следующих операций, реализованных в виде функций:

- *Creation*() – создание пустого стека;
- *Full*() – проверка стека на пустоту;
- *Add*() – добавление элемента;
- *Delete*() – удаление элемента;
- *Top*() – вывод верхнего элемента;
- *Size*() – вывод размера стека.

Как таковых пользовательских операций тут всего четыре: *Add*, *Delete*, *Top*, *Size*. Они доступны из консоли. Функция *Creation* – создает пустой стек сразу после запуска программы, а *Full* проверяет возможность выполнения пользовательских операций.

```
const int n=3;
struct Stack
    {
        int A[n];
        int count;
    };

void Creation(Stack *p)      //создание стека
    { p->count=0; }

int Full(Stack *p)           //проверка стека на пустоту
    {
        if (p->count==0) return 1;
        else if (p->count==n) return -1;
        else return 0;
    }
```

```
void Add(Stack *p)              //добавление элемента
    {
        int value;
        cout<<"Введите элемент > "; cin>>value;
        p->A[p->count]=value;
        p->count++;
    }

void Delete(Stack *p)           //удаление элемента
    { p->count--; }

int Top(Stack *p)               //вывод элементов
    { return p->A[p->count-1]; }

int Size(Stack *p)              //размер стека
    { return p->count; }

void main()                     //главная функция
    {
        Stack s;
        Creation(&s);
        char number;
        do
            {
                cout<<"1. Добавить элемент"<<endl;
                cout<<"2. Удалить элемент"<<endl;
                cout<<"3. Вывести верхний элемент"<<endl;
                cout<<"4. Узнать размер стека"<<endl;
                cout<<"0. Выйти"<<endl;
                cout<<"Номер команды > "; cin>>number;
                switch (number)
                    {
                        case '1':
                        if (Full(&s)==-1) cout<<endl<<"Стек заполнен\n\n";
                        else
                            {
                                Add(&s);
                                cout<<endl<<"Элемент добавлен в стек\n\n";
                            }
                        break;

                        case '2':
                        if (Full(&s)==1) cout<<endl<<"Стек пуст\n\n";
                        else
                            {
                                Delete(&s);
                                cout<<endl<<"Элемент удален из стека\n\n";
                            }
                        break;
```

```
                    case '3':
                    if (Full(&s)==1) cout<<endl<<"Стек пуст\n\n";
                    else cout<<"\nЭлементы стека: "<<Top(&s)<<"\n\n";
                    break;

                    case '4':
                    if (Full(&s)==1) cout<<endl<<"Стек пуст\n\n";
                    else cout<<"\nРазмер стека: "<<Size(&s)<<"\n\n";
                    break;

                    case '0': break;
                    default: cout<<endl<<" Команда не определена\n\n";
                    break;
                }
            } while(number!='0');
    }
```

Как видно, часто встречающимся элементом программы является поле *count* структуры stack. Оно исполняет роль указателя на «голову» стека. Например, для удаления элемента достаточно сдвинуть указатель на одну ячейку назад. Основная часть программы зациклена, и чтобы выйти из нее, а, следовательно, и из приложения вообще, нужно указать 0 в качестве номера команды.

Многие языки программирования располагают встроенными средствами организации и обработки стеков. Одним из них, как подчеркивалось ранее, является C++. Разберем принципы функционирования контейнера stack стандартной библиотеки шаблонов STL. Во-первых, stack – контейнер, в котором добавление и удаление элементов осуществляется с одного конца, что свойственно непосредственно стеку. Использование стековых операций не требует их описания в программе, т. е. stack здесь предоставляет набор стандартных функций. Для начала работы со стеком, в программе необходимо подключить библиотеку stack:

```
#include <stack>
```

и в функции описать его самого:

stack <*тип данных*> *имя стека*;

Обратите внимание, что скобки, обособляющие тип данных, указываются здесь не как подчеркивающие общую форму записи, а как обязательные при описании стека.

Теперь в программе можно использовать методы контейнера. Следующая программа является аналогом предыдущей, но вместо пользовательских функций в ней используются функции контейнера stack.

```
#include <stack>
void main()                    //главная функция
    {
        stack <int> S;         //создание стека S типа int
```

```
    char number; int value;
    do
        {
            cout<<"1. Добавить элемент"<<endl;
            cout<<"2. Удалить элемент"<<endl;
            cout<<"3. Получить верхний элемент"<<endl;
            cout<<"4. Узнать размер стека"<<endl;
            cout<<"0. Выйти"<<endl;
            cout<<"Номер команды > "; cin>>number;
            switch (number)
            {
                case '1':      //добавление элемента
                cout<<"Значение > "; cin>>value;
                S.push(value);
                cout<<endl<<"Элемент добавлен в стек\n\n";
                break;

                case '2':      //удаление  элемента
                if (S.empty()==true) cout<<"\nСтек пуст\n\n";
                else
                    {
                        S.pop();
                        cout<<endl<<"Элемент удален из стека\n\n";
                    }
                break;

                case '3':              //вывод верхнего элемента
                if (S.empty()==true) cout<<"\nСтек пуст\n\n";
                else cout<<"\nВерхний элемент стека: "<<S.top()<<"\n\n";
                break;

                case '4':              //вывод размера стека
                if (S.empty()==true) cout<<"\nСтек пуст\n\n";
                else cout<<"\nРазмер стека: "<<S.size()<<"\n\n";
                break;

                case '0': break;    //выход
                default: cout<<endl<<"Команда не определенная\n\n";
                break;
            }
        } while(number!='0');
}
```

Использование стандартных средств C++ позволило заметно сократить общий объем программы, и тем самым упростить листинг. По своему функционалу данная программа полностью аналогична предыдущей.

2.4 Очередь

Очередь – структура данных типа «список», позволяющая добавлять элементы лишь в конец списка, и извлекать их из его начала. Она функционирует по принципу FIFO (First In, First Out — «первым пришёл — первым вышел»), для которого характерно, что все элементы a_1, a_2, …, a_{n-1}, a_n, добавленные раньше элемента a_{n+1}, должны быть удалены прежде, чем будет удален элемент a_{n+1}. Также очередь может быть определена как частный случай односвязного списка, который обслуживает элементы в порядке их поступления. Как и в «живой» очереди, здесь первым будет обслужен тот, кто пришел первым.

Стандартный набор операций (часто у разных авторов он не идентичен), выполняемых над очередями, совпадает с тем, что используется при обработке стеков:

- добавление элемента;
- удаление элемента;
- чтение первого элемента.

Только, если в отношении стека в момент добавления или удаления элемента допустимо задействование лишь его вершины, то касательно очереди эти две операции должны быть применены так, как это регламентировано в определении этой структуры данных, т. е. добавление – в конец, удаление – из начала. Далее, при реализации интерфейса очереди, список стандартных операций будет расширен.

Выделяют два способа программной реализации очереди. Первый из них основан на базе массива, а второй на базе указателей (связного списка). Первый способ – статический, т. к. очередь представляется в виде простого статического массива, второй – динамический.

2.4.1 Реализация очереди с помощью массива

Данный способ позволяет организовать и впоследствии обрабатывать очередь, имеющую фиксированный размер. Определим список операций, который будет использоваться как при реализации статической очереди, так и динамической:

- *Creation*(*Q*) – создание очереди *Q*;
- *Full*(*Q*) – проверка очереди *Q* на пустоту;
- *Add*(*Q*) – добавление элемента в очередь *Q* (его значение задается из функции);
- *Delete*(*Q*) – удаление элемента из очереди *Q*;
- *Top*(*Q*) – вывод начального элемента очереди *Q*;
- *Size*(*Q*) – размер очереди *Q*.

В программе каждая из этих операций предстанет в виде отдельной подпрограммы. Помимо того, потребуется описать массив данных *data*[*n*], по сути,

являющийся хранилищем данных вместимостью *n*, а также указатель на конец очереди (на ту позицию, в которую будет добавлен очередной элемент) – *last*. Изначально *last* равен 0.

```
const int n=4;                  //размер очереди
struct Queue
    {
        int data[n]; //массив данных
        int last;               //указатель на начало
    };

void Creation(Queue *Q)     //создание очереди
    { Q->last=0; }

bool Full(Queue *Q)             //проверка очереди на пустоту
    {
        if (Q->last==0) return true;
        else return false;
    }

void Add(Queue *Q)              //добавление элемента
    {
        if (Q->last==n)
            { cout<<"\nОчередь заполнена\n\n"; return; }
        int value;
        cout<<"\nЗначение > "; cin>>value;
        Q->data[Q->last++]=value;
        cout<<endl<<"Элемент добавлен в очередь\n\n";
    }

void Delete(Queue *Q)       //удаление элемента
    {
        for (int i=0; i<Q->last && i<n; i++)    //смещение элементов
            Q->data[i]=Q->data[i+1]; Q->last--;
    }

    int Top(Queue *Q)       //вывод начального элемента
    { return Q->data[0]; }

int Size(Queue *Q)              //размер очереди
    { return Q->last; }

void main()                     //главная функция
    {
        Queue Q;
        Creation(&Q);
        char number;
        do
            {
```

```
                cout<<"1. Добавить элемент"<<endl;
                cout<<"2. Удалить элемент"<<endl;
                cout<<"3. Вывести верхний элемент"<<endl;
                cout<<"4. Узнать размер очереди"<<endl;
                cout<<"0. Выйти\n\n";
                cout<<"Номер команды > "; cin>>number;
                switch (number)
                        {
                                case '1': Add(&Q);
                                break;

                                case '2':
                                if (Full(&Q)) cout<<endl<<"Очередь пуста\n\n";
                                else
                                        {
                                                Delete(&Q);
                                                cout<<endl<<"Элемент удален\n\n";
                                        }
                                break;

                                case '3':
                                if (Full(&Q)) cout<<endl<<"Очередь пуста\n\n";
                                else cout<<"\nНачальный элемент: "<<Top(&Q)<<"\n\n";
                                break;

                                case '4':
                                if (Full(&Q)) cout<<endl<<"Очередь пуста\n\n";
                                else cout<<"\nРазмер очереди: "<<Size(&Q)<<"\n\n";
                                break;

                                case '0': break;
                                default: cout<<endl<<"Команда не определена\n\n";
                                break;
                        }
                } while(number!='0');
        }
```

В функции main, сразу после запуска программы, создается переменная *Q* структурного типа *Queue*, адрес которой будет посылаться в функцию (в зависимости от выбора операции) как фактический параметр. Функция *Creation* создает очередь, обнуляя указатель на последний элемент. Далее выполняется оператор цикла do..while (цикл с постусловием), выход из которого осуществляется только в том случае, если пользователь ввел 0 в качестве номера команды. В остальных случаях вызывается подпрограмма соответствующая команде, либо выводиться сообщение о том, что команда не определена.

Из всех подпрограмм особого внимания заслуживает функция *Delete*. Удаление элемента из очереди осуществляется путем сдвига всех элементов в начало, т. е. значения элементов переписываются: в *data*[0] записывается значение элемента

data[1], в *data*[1] – *data*[2] и т. д.; указатель конца смещается на позицию назад. Получается, что эта операция требует линейного времени $O(n)$, где n – размер очереди, в то время как остальные операции выполняются за константное время. Данная проблема поддается решению. Вместо «мигрирующей» очереди, наиболее приемлемо реализовать очередь на базе циклического массива. Здесь напрашивается аналогия с очередью в магазин: если в первом случае покупатели подходили к продавцу, то теперь продавец будет подходить к покупателям (конечно, такая тактика оказалась бы бесполезной, например, в супермаркетах и т. п.). В приведенной реализации очередь считалась заполненной тогда, когда указатель *last* находился над последней ячейкой, т. е. на расстоянии n элементов от начала. В циклическом варианте расширяется интерпретация определения позиции *last* относительно начала очереди. Пусть на начало указывает переменная *first*. Представим массив в виде круга – замкнутой структуры. После последнего элемента идет первый, и поэтому можно говорить, что очередь заполнила весь массив, тогда когда ячейки с указателями *last* и *first* находятся радом, а именно за *last* следует *first*. Теперь, удаление элемента из очереди осуществляется простым смещением указателя *first* на одну позицию вправо (по часовой); чтобы добавить элемент нужно записать его значение в ячейку *last* массива *data* и сместить указатель *last* на одну позицию правее. Чтобы не выйти за границы массива воспользуемся следующим выражением:

$(A \bmod n) + 1$

Здесь A – один из указателей, n – размер массива, а mod – операция взятия остатка от деления.

В циклической реализации, как и прежде, очередь не содержит элементов тогда, когда *first* и *last* указывают на одну и ту же ячейку. Но в таком случае возникает одно небольшое отличие этой реализации от предшествующей. Рассмотрим случай заполнения очереди, основанной на базе массива, размер которого 5:

Элементы	*first*	*last*
-	1	1
1	1	2
1, 2	1	3
1, 2, 3	1	4
1, 2, 3, 4	1	5

В левом столбце записаны произвольные значения элементов, а в двух других значения указателей при соответствующем состоянии очереди. Необходимо заметить, что в массив размером 5 удалось поместить только 4 элемента. Все дело в том, что еще один элемент требует смещения указателя *last* на позицию 1. Тогда *last=first*. Но

именно эта ситуация является необходимым и достаточным условием отсутствия в очереди элементов. Следовательно, мы не можем хранить в массиве больше *n*-1 элементов.

В следующей программе реализован интерфейс очереди, основанной на базе циклического массива:

```
const int n=6;                  //размер очереди
struct Queue
    {
        int data[n]; //массив данных
        int first;              //указатель на начало
        int last;               //указатель на конец
    };

void Creation(Queue *Q)     //создание очереди
    { Q->first=Q->last=1; }

bool Full(Queue *Q)             //проверка очереди на пустоту
    {
        if (Q->last==Q->first) return true;
        else return false;
    }

void Add(Queue *Q)              //добавление элемента
    {
        int value;
        cout<<"\nЗначение > "; cin>>value;
        if ((Q->last%(n-1))+1==Q->first)
        cout<<"\nОчередь заполнена\n\n";
        else
            {
                Q->data[Q->last]=value;
                Q->last=(Q->last%(n-1))+1;
                cout<<endl<<"Элемент добавлен в очередь\n\n";
            }
    }

void Delete(Queue *Q)       //удаление элемента
    {
        Q->first=(Q->first%(n-1))+1;
        cout<<endl<<"Элемент удален из очереди\n\n";
    }

int Top(Queue *Q)               //вывод начального элемента
    { return Q->data[Q->first]; }

int Size(Queue *Q)              //размер очереди
    {
        if (Q->first>Q->last) return (n-1)-(Q->first-Q->last);
```

```
        else return Q->last-Q->first;
    }

void main()                     //главная функция
    {
        Queue Q;
        Creation(&Q);
        char number;
        do
            {
                cout<<"1. Добавить элемент"<<endl;
                cout<<"2. Удалить элемент"<<endl;
                cout<<"3. Вывести верхний элемент"<<endl;
                cout<<"4. Узнать размер очереди"<<endl;
                cout<<"0. Выйти\n\n";
                cout<<"Номер команды > "; cin>>number;
                switch (number)
                    {
                        case '1': Add(&Q);
                        break;

                        case '2':
                        if (Full(&Q)) cout<<endl<<"Очередь пуста\n\n";
                        else Delete(&Q);
                        break;

                        case '3':
                        if (Full(&Q)) cout<<endl<<"Очередь пуста\n\n";
                        else cout<<"\nНачальный элемент: "<<Top(&Q)<<"\n\n";
                        break;

                        case '4':
                        if (Full(&Q)) cout<<endl<<"Очередь пуста\n\n";
                        else cout<<"\nРазмер очереди: "<<Size(&Q)<<"\n\n";
                        break;

                        case '0': break;
                        default: cout<<endl<<"Команда не определена\n\n";
                        break;
                    }
            } while(number!='0');
    }
```

Таким образом, циклический вариант позволяет оптимизировать операцию *Delete*, которая прежде требовала линейного времени, а теперь выполняется за константное, независимо от длины очереди. Тем не менее, реализация очереди на базе массива имеет один существенный недостаток: размер очереди статичен, поскольку зависит от размера массива. Реализация очереди на базе связного списка позволит обойти эту проблему.

2.4.2 Реализация очереди с помощью указателей

Данный способ предполагает работу с динамической памятью. Для представления очереди используется односвязный список, в конец которого помещаются новые элементы, а старые извлекаются, соответственно, из начала списка. Здесь каждый узел списка имеет два поля: информационное и связующее:

```
struct Node
    {
        int data;
        Node *next;
    };
```

Также понадобиться определить указатели на начало и конец очереди:

```
struct Queue
    {
        Node *first;
        Node *last;
    };
```

Следующее консольное приложение обслуживает очередь, каждый элемент которой – целое число. Весь процесс обуславливают все те же операции: *Creation*, *Full*, *Add*, *Delete*, *Top*, *Size*.

```
struct Node                 //описание узла списка
    {
        int data;           //информационное поле
        Node *next;         //указатель на следующий элемент
    };

struct Queue                //описание очереди
    {
        int size;           //счетчик размера очереди
        Node *first;        //указатель на начало очереди
        Node *last;         //указатель на конец очереди
    };

void Creation(Queue *Q)     //создание очереди
    {
        Q->first=new Node;
        Q->first->next=NULL;
        Q->last=Q->first;
        Q->size=0;
    }

bool Full(Queue *Q)         //проверка очереди на пустоту
```

```
	{
		if (Q->first==Q->last) return true;
		else return false;
	}

int Top(Queue *Q)		//вывод начального элемента
	{ return Q->first->next->data; }

void Add(Queue *Q)		//добавление элемента
	{
		int value;
		cout<<"\nЗначение > "; cin>>value;
		Q->last->next=new Node;
		Q->last=Q->last->next;
		Q->last->data=value;		//добавление элемента в конец
		Q->last->next=NULL;		//обнуление указателя на следующий элемент
		Q->size++;
		cout<<"\nЭлемент добавлен\n\n";
	}

void Delete(Queue *Q)			//удаление элемента
	{
		Q->first=Q->first->next; //смещение указателя
		Q->size--;
		cout<<"\nЭлемент удален\n\n";
	}

int Size(Queue *Q)			//размер очереди
	{ return Q->size; }

void main()				//главная функция
	{
		Queue Q;
		Creation(&Q);
		char number;
		do
			{
				cout<<"1. Добавить элемент"<<endl;
				cout<<"2. Удалить элемент"<<endl;
				cout<<"3. Вывести верхний элемент"<<endl;
				cout<<"4. Узнать размер очереди"<<endl;
				cout<<"0. Выйти\n\n";
				cout<<"Номер команды > "; cin>>number;
				switch (number)
					{
						case '1': Add(&Q);
						break;

						case '2':
						if (Full(&Q)) cout<<endl<<"Очередь пуста\n\n";
```

```
                    else Delete(&Q);
                    break;

                    case '3':
                    if (Full(&Q)) cout<<endl<<"Очередь пуста\n\n";
                    else { cout<<"\nНачальный элемент: "<<Top(&Q)<<"\n\n"; }
                    break;

                    case '4':
                    if (Full(&Q)) cout<<endl<<"Очередь пуста\n\n";
                    else cout<<"\nРазмер очереди: "<<Size(&Q)<<"\n\n";
                    break;

                    case '0': break;
                    default: cout<<endl<<"Команда не определена\n\n";
                    break;
                }
        } while(number!='0');
}
```

Как отмечалось, этот способ позволяет не заботиться о месте, отводимом под рассматриваемую структуру данных – ее объем ограничен лишь памятью компьютера. Тем не менее, к числу недостатков данной реализации, в сравнении с предыдущей, можно отнести: увеличение времени обработки и количества памяти, да и сам код несколько сложнее для понимания.

2.5 Дек

Дек (deque — double ended queue, «двусторонняя очередь») – структура данных типа «список», функционирующая одновременно по двум принцам организации данных: FIFO и LIFO. Определить дек можно как очередь с двумя сторонами, так и стек, имеющий два конца (рис. 2.12). То есть данный подвид списка характерен двухсторонним доступом: выполнение поэлементной операции, определенной над деком, предполагает возможность выбора одной из его сторон в качестве активной.

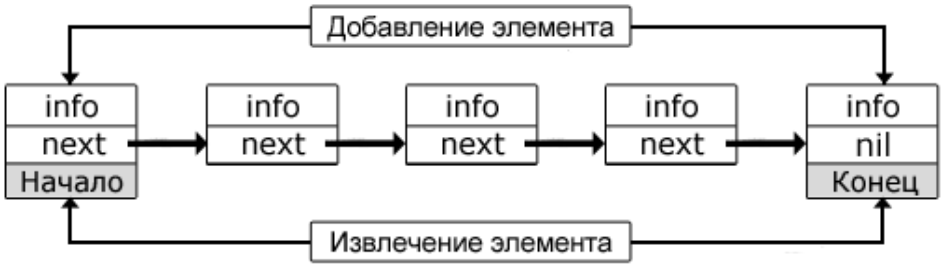


Рисунок 2.12 – Дек

Число основных операций, выполняемых над стеком и очередью, как помнит читатель, равнялось трем: добавление элемента, удаление элемента, чтение элемента. При этом не указывалось место структуры данных, активное в момент их выполнения, поскольку ранее оно однозначно определялось свойствами (определением) самой структуры. Теперь, ввиду дека как обобщенного случая, для приведенных операций следует указать эту область. Разделив каждую из операций на две: одну применительно к «голове» дека, другую – его «хвосту», получим набор из шести операций:

- добавление элемента в начало;
- добавление элемента в конец;
- удаление первого элемента;
- удаление последнего элемента;
- чтение первого элемента;
- чтение последнего элемента.

На практике этот список может быть дополнен проверкой дека на пустоту, получением его размера и некоторыми другими операциями.

В плане реализации двусторонняя очередь очень близка к стеку и обычной очереди: в качестве ее базиса приемлемо использовать как массив, так и список. Далее мы напишем интерфейс обработки дека, представленного обычным индексным массивом. Программа будет состоять из основной части и девяти функций:

- *Creation* – обнуляет указатель на конец, так сказать создает дек;
- *Full* – проверяет дек на наличие элементов;

- *AddL* – добавляет элемент в начало;
- *AddR* – добавляет элемент в конец;
- *DeleteL* – удаляет первый элемент;
- *DeleteR* – удаляет последний элемент;
- *OutputL* – выводит первый элемент;
- *OutputR* – выводит последний элемент;
- *Size* – выводит количество элементов дека.

Помимо самого массива потребуется указатель на последний элемент, назовем его *last*. Указатель на первый элемент не потребуется, поскольку дек будет представлять собой смещающуюся структуру, т. е. при добавлении нового элемента в начало, каждый из старых элементов сместиться на одну позицию вправо, уступив тем самым первому элементу ячейку с индексом 0 (в C++), следовательно, адрес первого элемента – константа.

Программная реализация двусторонней очереди (дека):

```
const int n=5;                  //размер дека
struct Deque
    {
        int data[n];            //массив данных
        int last;               //указатель на конец
    };

void Creation(Deque *D)      //создание дека
    { D->last=0; }

bool Full(Deque *D)             //проверка дека на пустоту
    {
        if (D->last==0) return true;
        else return false;
    }

void AddL(Deque *D)             //добавление элемента в начало
    {
        if (D->last==n)
            { cout<<"\nДек заполнен\n\n"; return; }
        int value;
        cout<<"\n3начение > "; cin>>value;
        for (int i=D->last; i>0; i--) D->data[i]=D->data[i-1];
        D->data[0]=value;
        D->last++;
        cout<<endl<<"Элемент добавлен\n\n";
    }

void AddR(Deque *D) //добавление элемента в конец
    {
        if (D->last==n) { cout<<"\nДек заполнен\n\n"; return; }
        int value;
```

```
        cout<<"\n3начение > "; cin>>value;
        D->data[D->last++]=value;
        cout<<endl<<"Элемент добавлен\n\n";
    }

void DeleteL(Deque *D)                  //удаление первого элемента
    {
        for (int i=0; i<D->last; i++)   //смещение элементов
        D->data[i]=D->data[i+1]; D->last--;
    }

void DeleteR(Deque *D)                  //удаление последнего элемента
    { D->last--; }

int OutputL(Deque *D)                   //вывод первого элемента
    { return D->data[0]; }

int OutputR(Deque *D)                   //вывод последнего элемента
    { return D->data[D->last-1]; }

int Size(Deque *D)                      //размер дека
    { return D->last; }

int main()                              //главная функция
    {
        Deque D;
        Creation(&D);
        char number;
        do
            {
                cout<<"1. Добавить элемент в начало"<<endl;
                cout<<"2. Добавить элемент в конец"<<endl;
                cout<<"3. Удалить первый элемент"<<endl;
                cout<<"4. Удалить последний элемент"<<endl;
                cout<<"5. Вывести первый элемент"<<endl;
                cout<<"6. Вывести последний элемент"<<endl;
                cout<<"7. Узнать размер дека"<<endl;
                cout<<"0. Выйти\n\n";
                cout<<"Номер команды > "; cin>>number;
                switch (number)
                    {
                        case '1': AddL(&D);
                        break;

                        case '2': AddR(&D);
                        break;

                        case '3':
                        if (Full(&D)) cout<<endl<<"Дек пуст\n\n";
                        else
```

```
                {
                    DeleteL(&D);
                    cout<<endl<<"Элемент удален из дека\n\n";
                }
            break;

            case '4':
            if (Full(&D)) cout<<endl<<"Дек пуст\n\n";
            else
                {
                    DeleteR(&D);
                    cout<<endl<<"Элемент удален\n\n";
                }
            break;

            case '5':
            if (Full(&D)) cout<<endl<<"Дек пуст\n\n";
            else cout<<"\nПервый элемент: "<<OutputL(&D)<<"\n\n";
            break;

            case '6':
            if (Full(&D)) cout<<endl<<"Дек пуст\n\n";
            else cout<<"\nПоследний элемент: "<<OutputR(&D)<<"\n\n";
            break;

            case '7':
            if (Full(&D)) cout<<endl<<"Дек пуст\n\n";
            else cout<<"\nРазмер дека: "<<Size(&D)<<"\n\n";
            break;

            case '0': break;
            default: cout<<endl<<"Команда не определена\n\n";
            break;
        }
    } while(number!='0');
}
```

Двусторонняя очередь, реализованная таким способом, имеет два существенных недостатка: ограниченный размер и линейное время. Последнее касается добавления элемента в начало или извлечение его оттуда, а именно операциям *AddL* и *DeleteL* необходимо *n* перестановок, где *n* – число элементов в деке.

Стандартная библиотека C++ предоставляет специальные средства работы с двусторонней очередью. Для этого в ней предусмотрен контейнер deque. Он позволяет за *O*(1) осуществлять вставку и удаление элементов. Методы контейнера deque:

- front – возврат значения первого элемента;
- back – возврат значения последнего элемента;
- push_front – добавление элемента в начало;

- push_back – добавление элемента в конец;
- pop_front – удаление первого элемента;
- pop_back – удаление последнего элемента;
- size – возврат числа элементов дека;
- clear – очистка дека.

Следующая программа полностью повторяет функционал предыдущей, но для обработки дека она использует не пользовательские подпрограммы, а методы контейнера deque.

```
#include <deque>
int main()                          //главная функция
    {
        deque<int>  D;              //создание дека D размером 5
        deque<int>::iterator out;
        int value;
        char number;
        do
            {
                cout<<"1. Добавить элемент в начало"<<endl;
                cout<<"2. Добавить элемент в конец"<<endl;
                cout<<"3. Удалить первый элемент"<<endl;
                cout<<"4. Удалить последний элемент"<<endl;
                cout<<"5. Вывести первый элемент"<<endl;
                cout<<"6. Вывести последний элемент"<<endl;
                cout<<"7. Узнать размер дека"<<endl;
                cout<<"0. Выйти\n\n";
                cout<<"Номер команды > "; cin>>number;
                switch (number)
                    {
                        case '1':
                        cout<<"\nЗначение > "; cin>>value;
                        D.push_front(value);
                        cout<<endl<<"Элемент добавлен\n\n";
                        break;

                        case '2':
                        cout<<"\nЗначение > "; cin>>value;
                        D.push_back(value);
                        cout<<endl<<"Элемент добавлен\n\n";
                        break;

                        case '3': if (D.empty()) cout<<"\nДек пуст\n\n";
                        else
                            {
                                D.erase(D.begin());
                                cout<<endl<<"Элемент удален\n\n";
                            }
                        break;
```

```
                        case '4': if (D.empty()) cout<<"\nДек пуст\n\n";
                        else
                                {
                                        D.erase(D.end()-1);
                                        cout<<endl<<"Элемент удален\n\n";
                                }
                        break;

                        case '5':
                        if (D.empty()) cout<<endl<<"Дек пуст\n\n";
                        else
                                {
                                        out=D.begin();
                                        cout<<"\nПервый элемент: "<<*out<<"\n\n";
                                }
                        break;

                        case '6': if (D.empty()) cout<<"\nДек пуст\n\n";
                        else
                                {
                                        out=D.end()-1;
                                        cout<<"\nПоследний элемент: "<<*out<<"\n\n";
                                }
                        break;

                        case '7':
                        if (D.empty()) cout<<endl<<"Дек пуст\n\n";
                        else cout<<"\nРазмер дека: "<<D.size()<<"\n\n";
                        break;

                        case '0': break;
                        default: cout<<endl<<"Команда не определена\n\n";
                        break;
                }
        } while(number!='0');
}
```

Глава 3. Графы

3.1 Основные понятия и виды графов

Математика оперирует не содержанием вещей, а их структурой, абстрагируя ее из всего того, что дано как целое. Отвлечение от качеств и свойств предмета позволяет выявить у данного предмета его основу, неотъемлемую часть, то, что поставит его в один ряд с непохожими на него, на первый взгляд, предметами. Теория графов – это раздел математики, поэтому и в ней используется принцип отвлечения: не важно, что представляет собой предмет, важно лишь то, является ли он графом, т. е. обладает ли обязательными для графов свойствами.

Прежде чем перейти к изучению способов представления графа, рассмотрим примеры, дадим определение графу и ознакомимся с основными понятиями теории графов. Здесь не будут рассмотрены все аспекты теории графов (этого и не требуется), но, по большой части, те, что понадобятся нам в дальнейшем.

В своей жизни мы, так или иначе, соприкасались с объектами, имеющими структуру графа. К таким объектам относятся разного рода маршруты общественного транспорта: система метрополитена, автобусные маршруту и т.п. В частности, программисту знакома компьютерная сеть, также являющаяся графом (рис. 3.1). Общее здесь это наличие точек, соединенных линиями. Так в компьютерной сети точками являются отдельные серверы, а линиями – различные виды электрических сигналов. В метрополитене первое – станции, второе – туннели, проложенные между ними. В теории графов точки именуется *вершинами,* или *узлами*, а линии – *ребрами,* или *дугами*. Таким образом, граф – это совокупность вершин, соединённых ребрами.

Вернемся к компьютерной сети. Она обладает определенной топологией, и может быть условно изображена в виде некоторого числа компьютеров и путей их соединяющих. На рисунке ниже в качестве примера показана *полносвязная топология*.

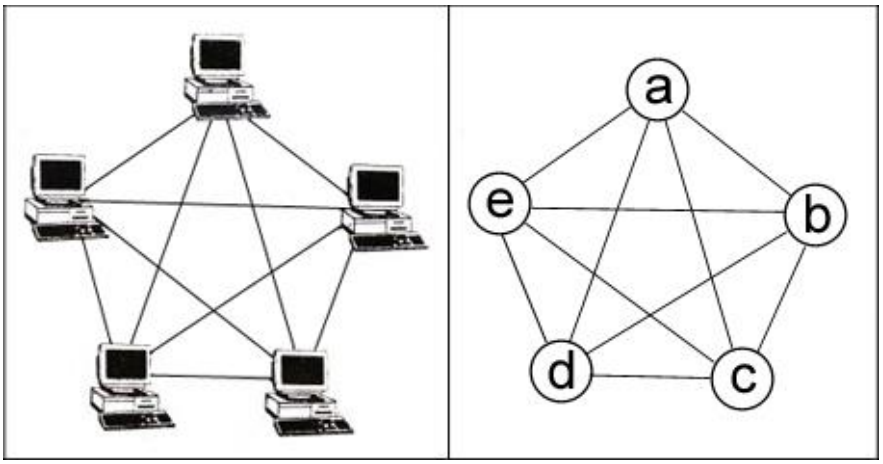


Рисунок 3.1 – Компьютерная сеть

Это, по сути, граф. Пять компьютеров являются вершинами, а соединения (пути передачи сигналов) между ними – ребрами. Заменив компьютеры вершинами, мы получим математический объект – граф, который имеет 10 ребер и 5 вершин. Пронумеровать вершины можно произвольным образом, а не обязательно так, как это сделано на рисунке.

Вот некоторые важные обозначения, используемые в теории графов:

- $G=(V, E)$, здесь G – граф, V – его вершины, а E – ребра;
- $|V|$ – *порядок* (число вершин);
- $|E|$ – *размер графа* (число рёбер).

В нашем случае (рис. 1) $|V|=5$, $|E|=10$;

Когда из любой вершины доступна любая другая вершина, то такой граф называется *неориентированным* связным графом (рис. 3.1). Если же граф связный, но это условие не выполняется, тогда такой граф называется *ориентированным* или *орграфом* (рис. 3.2). Ребра орграфа принято называть *дугами*.

В ориентированных и неориентированных графах имеется понятие *степени вершины*. *Степень вершины* – это количество ребер, соединяющих ее с другими вершинами. *Степень входа* вершины – количество входящих в эту вершину ребер, *степень выхода* – количество исходящих ребер. Сумма всех степеней графа равна удвоенному количеству всех его ребер. Для рисунка 2 сумма всех степеней равна 20.

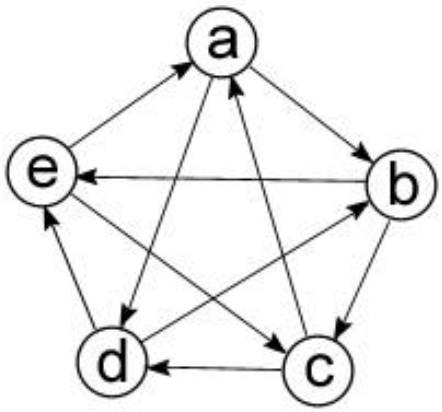


Рисунок 3.2 – Ориентированный граф

В орграфе, в отличие от неориентированного графа, имеется возможность двигаться из вершины h в вершину s без промежуточных вершин, лишь тогда когда дуга выходит из h и входит в s, но не наоборот.

Ориентированные графы имеют следующую форму записи:

$G=(V, A)$, где V – вершины, A – направленные ребра.

Третий тип графов – *смешанные графы* (рис. 3.3). Они имеют как направленные ребра, так и ненаправленные. Формально смешанный граф записывается так: *G*=(*V*, *E*, *A*), где каждая из букв в скобках обозначает тоже, что ей приписывалось ранее.

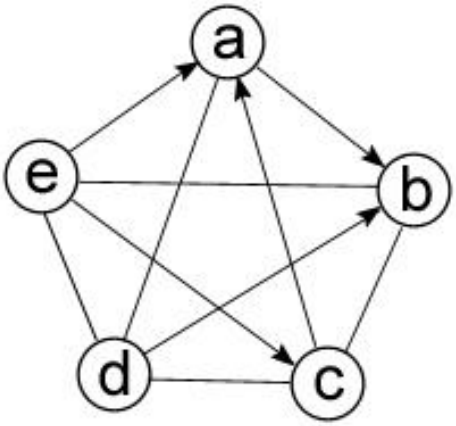


Рисунок 3.3 – Смешанный граф

На рисунке 3 изображен смешанный граф. Одни дуги направленные [(*e*, *a*), (*e*, *c*), (*a*, *b*), (*c*, *a*), (*d*, *b*)], другие – ненаправленные [(*e*, *d*), (*e*, *b*), (*d*, *c*)…].

Когда у ребра оба конца совпадают, т. е. ребро выходит из некоторой вершины *F* и входит в нее, то такое ребро называется *петлей* (рис. 3.4).

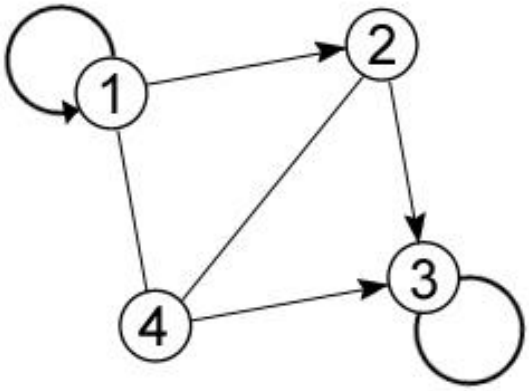


Рисунок 3.4 – Петли графа

Два или более графов на первый взгляд могут показаться разными по своей структуре, что возникает вследствие различного их изображения. Но это не всегда так. Возьмем два графа (рис. 3.5). Они эквивалентны друг другу, ведь не изменяя структуру одного графа можно построить другой. Такие графы называются *изоморфными*, т. е. обладающими тем свойством, что какая-либо вершина с определенным числом ребер в одном графе имеет тождественную вершину в другом.

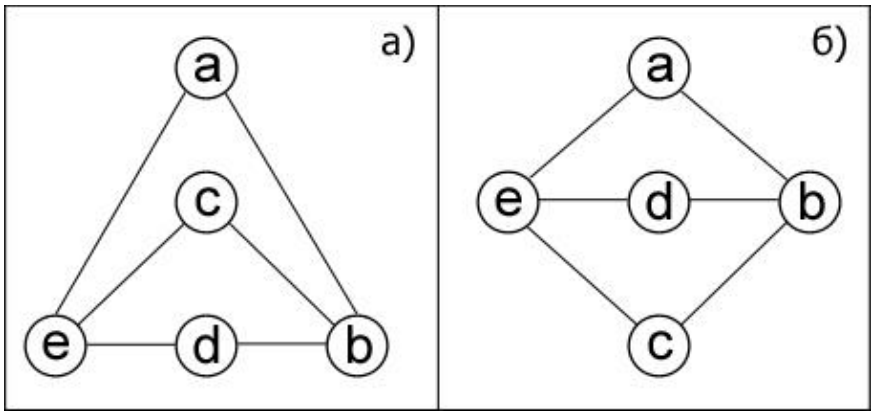


Рисунок 3.5 – Изоморфные графы

Когда каждому ребру графа поставлено в соответствие некоторое значение, называемое *весом ребра*, тогда такой граф взвешенный. В разных задачах в качестве веса могут выступать различные виды измерений, например длины, цены маршруты и т. п. В графическом представлении графа весовые значения указываются, как правило, рядом с ребрами.

В любом из рассмотренных нами графов имеется возможность выделить путь и, причем не один. *Путь* – это последовательность вершин, каждая из которых соединена с соседней вершиной посредством ребра. Если первая и последняя вершины совпадают, то такой путь называется *циклом*. *Длина пути* определяется количеством составляющих его ребер. Например, на рисунке 4.а путем служит последовательность [(*e*), (*a*), (*b*), (*c*)]. Этот путь является подграфом, так как к нему применимо определение последнего, а именно: граф $G'=(V', E')$ является подграфом графа $G=(V, E)$, только тогда когда V' и E' принадлежат V, E.

Граф, как и большинство других математических объектов, может быть представлен на компьютере (сохранен в его памяти). Существуют несколько способов его интерпретации, вот наиболее известные из них:

- матрица смежности;
- матрица инцидентности;
- список смежности;
- список ребер.

Использование двух первых методов предполагает хранение графа в виде двумерного массива (матрицы). Причем размеры этих массивов, зависят от количества вершин и/или ребер в конкретном графе. Так размер матрицы смежности $n \times n$, где n – число вершин, а матрицы инцидентности $n \times m$, n – число вершин, m – число ребер в графе.

3.2 Матрица смежности

Матрица смежности графа — это квадратная матрица, в которой каждый элемент принимает одно из двух значений: 0 или 1. Прежде чем отобразить граф через матрицу смежности, рассмотрим простой пример такой матрицы (рис. 3.6).

	1	2	3	4
1	0	1	0	1
2	0	0	1	1
3	0	1	0	0
4	1	0	1	0

Рисунок 3.6 – Матрица смежности

Это двоичная квадратная матрица, т. к. число строк в ней равно числу столбцов, и любой из ее элементов имеет значение либо 1, либо 0. Первая строка и первый столбец (не входят в состав матрицы, а показаны здесь для легкости ее восприятия) содержат номера, на пересечении которых находится каждый из элементов, и они определяют индексное значение последних. Имея в наличии лишь матрицу такого типа, несложно построить соответствующий ей граф.

Ниже (рис. 3.7) изображена все та же матрица смежности, имеющая размерность 4×4. Числа, выделенные синим, можно рассматривать как вершины смешанного графа, расположенного справа – того, представлением которого является матрица.

	1	2	3	4
1	0	1	0	1
2	0	0	1	1
3	0	1	0	0
4	1	0	1	0

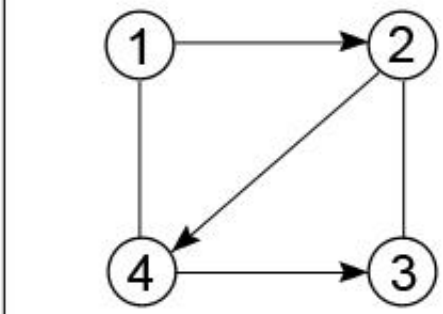


Рисунок 3.7 – Граф и его матрица смежности

Для графического отображения графа необходимо уметь вычислять по матрице смежности количество его вершин, а также обладать знанием следующего правила. Когда из одной вершины в другую проход свободен (имеется ребро), тогда в ячейку заноситься 1, иначе – 0. Немного формализовав только что сказанное, получим: если из i в j существует ребро, то $A[i][j]:=1$, в противном случае $A[i][j]:=0$. Как видно, все элементы на главной диагонали равны нулю, это следствие отсутствия у графа петель. Но ни что не мешает, чтобы некоторые или все элементы главной диагонали были единицами.

В программе матрица смежности задается при помощи обычного двумерного массива, имеющего размерность $n \times n$, где n – число вершин графа. На языке C++, описать ее можно, например, так:

```
int graph[n][n] =
    {
        {0, 1, 0, 1},
        {0, 0, 1, 1},
        {0, 1, 0, 0},
        {1, 0, 1, 0}
    };
```

В Паскале способ описания практически аналогичен:

```
const graph: array[1..n,1..n] of integer =
    (
        (0, 1, 0, 1),
        (0, 0, 1, 1),
        (0, 1, 0, 0),
        (1, 0, 1, 0)
    );
```

В случае если граф неизвестен заранее, а определяется пользователем, то нужно организовать ручной ввод двумерного массива, так как это предусматривает конкретный язык программирования.

Чтобы представить граф в виде матрицы смежности понадобиться $O(|V|^2)$ памяти, поскольку ее размер, как уже отмечалось, равен квадрату числа всех вершин. И если количество ребер графа, в сопоставлении с количеством вершин, невелико, то значения многих элементов матрицы смежности будут нулевыми, следовательно, использование данного метода нецелесообразно, т. к. для подобных графов имеются наиболее оптимальные способы их представления.

3.3 Список смежности

По отношению к памяти списки смежности менее требовательны, чем матрицы смежности, для их хранения нужно $O(|V|+|E|)$ памяти. Такой список графически можно изобразить в виде таблицы, столбцов в которой – два, а строк не больше чем вершин в графе. В качестве примера возьмем смешанный граф:

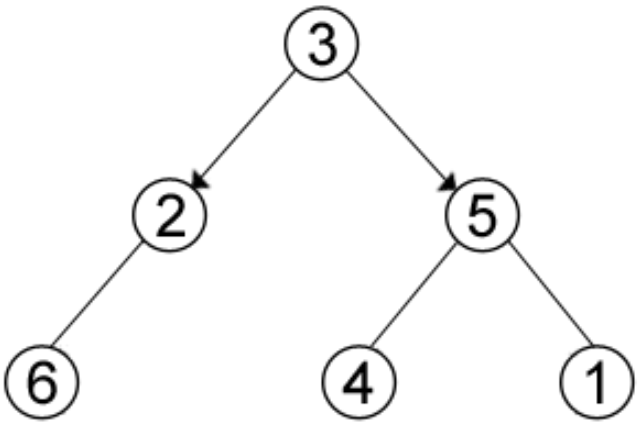


Рисунок 3.8 – Смешанный граф

В нем 6 вершин ($|V|$) и 5 ребер ($|E|$). Из последних 2 ребра направленные и 3 ненаправленные, и, причем из каждой вершины выходит, как минимум одно ребро в другую вершину, из чего следует, что в списке смежности этого графа будет $|V|$ строк.

Вершина выхода	Вершины входа
1	5
2	6
3	2, 5
4	5
5	1, 4
6	2

В *i* строке и 1 столбце указана вершина выхода, а в *i* строке и 2 столбце – вершины входа. Так, к примеру, из вершины 5 выходят два ребра, входящие в вершины 1 и 4.

Теперь перейдем непосредственно к программной реализации списка смежности. Количество вершин и ребер будут задаваться с клавиатуры, поэтому установим ограничения, иначе говоря, определим две константы, одна из которых

отвечает за максимально возможное число вершин (*Vmax*), другая – ребер (*Emax*). Далее, нам понадобятся три одномерных целочисленных массива:

terminal[1..*Emax*] – хранит вершины, в которые входят ребра;

next [1..*Emax*] – содержит указатели на элементы массива *terminal*;

head[1..*Vmax*] – содержит указатели на начала подсписков, т. е. на такие вершины записанные в массив *terminal*, с которых начинается процесс перечисления всех вершин смежных одной *i*-ой вершине.

В программе позволительно выделить две основные части: ввод ребер с последующим добавлением их в список смежности, и вывод получившегося списка смежности на экран.

Код программы на C++:

```
const int Vmax=100, Emax=Vmax*2;
int head[Vmax];
int next_el[Emax];
int terminal[Emax];
int n, m, i, j, k, v, u;
char r;
void Add(int v, int u)   //добавление ребра
    {
        k=k+1;
        terminal[k]=u;
        next_el[k]=head[v];
        head[v]=k;
    }

void main()              //главная функция
    {
        k=0;
        cout<<"Кол. вершин >> "; cin>>n;
        cout<<"Кол. ребер >> "; cin>>m;
        cout<<"Вводите смежные вершины:"<<endl;
        for (i=0; i<m; i++)
            {
                cin>>v>>u;
                cout<<"Ребро ориент.? (y/n) >> "; cin>>r;
                if (r=='y') Add(v, u);
                else
                    {
                        Add(v, u);
                        Add(u, v);
                    }
                cout<<"..."<<endl;
            }
        cout<<"Список смежности графа:";
```

```
        for (i=0; i<n+1; i++)       //вывод списка смежности
                {
                        j=head[i];
                        if (i) cout<<i<<"->";
                        while (j>0)
                                {
                                        if (!next_el[j]) cout<<terminal[j];
                                        else cout<<terminal[j]<<", ";
                                        j=next_el[j];
                                }
                        cout<<endl;
                }
    }
```

Код программы на Pascal:

```
Const
Vmax=100;
Emax=Vmax*2;
Var
head: array [1..Vmax] of integer;
next: array [1..Emax] of integer;
terminal: array [1..Emax] of integer;
n, m, i, j, k, v, u: integer;
r: char;

procedure Add(v, u: integer);       {добавление ребер}
    begin
        k:=k+1;
        terminal[k]:=u;
        next[k]:=head[v];
        head[v]:=k;
    end;

begin       {основной блок программы}
    k:=0;
    write('Кол. вершин >> '); read(n);
    write('Кол. ребер >> '); read(m);
    writeln('Вводите смежные вершины:');
    for i:=1 to m do
        begin
            read(v, u);
            write('Ребро ориент.? (y/n) >> '); read(r);
            if r='y' then add(v, u)
            else begin
                add(v, u);
                add(u, v);
                end;
            writeln('...');
```

```
        end;

    writeln('Список смежности графа:');
    for i:=1 to n do    {вывод списка смежности}
        begin
            j:=head[i];
            write(i, '->');
            while j>0 do
                begin
                    if next[j]=0 then write(terminal[j])
                    else write(terminal[j], ', ');
                    j:=next[j];
                end;
            writeln;
        end;
end.
```

Работа двух приведенных программ идентична, а отличаются они друг от друга, лишь символикой, используемой в языке программирования, поэтому дальнейшие рассуждения будут касаться их обеих. Итак, первое действие, на которое стоит обратить внимание это запрос на ввод суммы вершин *n* и ребер *m* графа (пользователь должен заранее знать эти данные). Далее, запускается цикл ввода ребер (смежных вершин). Условие в этом цикле нужно для того, чтобы узнать, какое введено ребро. Если введено направленное ребро, то функция *add* вызывается 1 раз, иначе – 2, тем самым внося сведения, что есть ребро как из вершины *v* в вершину *u*, так и из *u* в *v*. После того как список смежности сформирован, программа выводит его на экран. Для этого использован цикл от 1 до *n*, где *n* – количество вершин, а также вложенный в него цикл, который прекращает свою работу тогда, когда указатель на следующий элемент для *i*-ой вершины отсутствует, т. е. все смежные ей вершины уже выведены.

Функция *add* производит добавление ребер в изначально пустой список смежности:

```
Add(v, u)
k:=k+1
terminal[k]:=u
next[k]:=head[v]
head[v]:=k
```

Чтобы сделать это, производятся операции с формальными параметрами, вспомогательной переменной *k* и тремя одномерными целочисленными массивами. Значение переменной *k* увеличивается на 1. Затем в *k*-ый элемент массива *terminal* записывается конечная для некоторого ребра вершина *u*. В третий строке *k*-ому элементу массива *next* присваивается адрес следующего элемента массива *terminal*. Ну

и в конце массив *head* заполняется указателями на стартовые элементы, те с которых начинается подсписок смежных вершин с некоторой *i*-ой вершиной.

Так как в ячейке на пересечении *i*-ой строки и 2-ого столбца могут быть записаны несколько элементов (что соответствует нескольким смежным вершинам) назовем каждую строку в списке смежности его подсписком. Таким образом, в выведенном списке смежности, элементы подсписков будут отсортированы в обратном порядке. Но, как правило, порядок вывода смежных вершин (в подсписках) попросту неважен.

3.4 Список ребер

Список, в каждой строке которого записаны две смежные вершины и вес, соединяющего их ребра, называется списком ребер графа. Возьмем связный граф $G=(V, E)$, и множество ребер E разделим на два класса d и k, где d – подмножество, включающее только неориентированные ребра графа G, а k – ориентированные. Предположим, что некоторая величина p соответствует количеству всех ребер, входящих в подмножество d, а s – тоже относительно k. Тогда для графа G высота списка ребер будет равна $s+p*2$. Иными словами, количество строк в списке ребер всегда должно быть равно величине, получающейся в результате сложения ориентированных ребер с неориентированными, увеличенными вдвое. Это утверждение следует из сказанного ранее, а именно, что данный способ представления графа предусматривает хранение в каждой строке двух смежных вершин, а неориентированное ребро, соединяющее вершины v и u, идет как из v в u, так и из u в v.

Рассмотрим смешанный граф (рис. 3.9), в котором 5 вершин, 4 ребра и каждому ребру поставлено в соответствие некоторое целое значение (для наглядности оно составлено из номеров вершин).

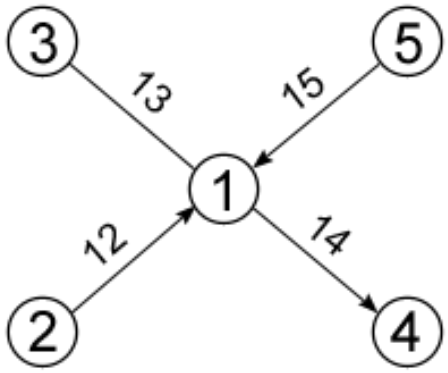


Рисунок 3.9 – Смешанный граф

В нем 3 направленных ребра и 1 ненаправленное. Подставив значения в формулу, получим высоту списка ребер: 3+1*2=5.

1	3	13
1	4	14
2	1	12
3	1	13
5	1	15

Так выглядит список ребер для приведенного графа. Это таблица размером $n \times 3$, где $n=s+p*2=3+1*2=5$. Элементы первого столбца располагаются в порядке возрастания, в то время как порядок элементов второго столбца зависит от первого, а третьего от двух предыдущих.

Программная реализация списка ребер похожа на реализацию списка смежности. Так же как и в последней, в ней изначально необходимо организовать ввод данных, которые при помощи специальной функции будут распределяться по массивам. Второй шаг – вывести получившийся список смежности на экран.

В списке смежности хранились только смежные вершины, а здесь, помимо них, будут храниться веса инцидентных этим вершинам ребер, и для этой цели понадобиться дополнительный массив. К тому же, данный метод требует более строго порядка вывода вершин, т. к. если в списке смежности последовательность нарушалась лишь в строках, то использование аналогичного способа построения приведет к нарушению порядка в столбцах. Поэтому функцию добавления ребер следует организовать иначе.

Код программы на C++:

```
const int Vmax=100,
Emax=Vmax*(Vmax-1)/2; //в случае, если граф полный
int terminal[Emax], weight[Emax], point[Emax];
int head[Vmax], last[Vmax];
int n, m, v, u, w, k=0, i;
char r;
void Add(int v, int u, int w) //добавление ребра
    {
        k=k+1;
        terminal[k]=u; weight[k]=w;
        //если вершина v новая, то
        //первая смежная ей вершина имеет номер k
        if (head[v]==0) head[v]=k;
        //если вершина v уже просматривалась, то
        //следующая смежная с ней вершина имеет номер k
        if (last[v]!=0) point[last[v]]=k;
        last[v]=k;
    }

void main()     //главная функция
    {
        cout<<"Кол. вершин >> "; cin>>n;
        cout<<"Кол. ребер >> "; cin>>m;
        cout<<"Вводите ребра и их вес (v, u, w):\n";
        for (i=0; i<m; i++)
            {
                cin>>v>>u>>w;
```

```
                cout<<"Ребро ориент.? (y/n) >> "; cin>>r;
                if (r=='y') Add(v, u, w);
                else
                        {
                                Add(v, u, w);
                                Add(u, v, w);
                        }
                cout<<"..."<<endl;
        }
    m=m*2;
    cout<<"Список ребер графа:\n";
    for (i=0; i<m; i++) //вывод списка ребер
        {
            k=head[i];
            while (k>0)
                {
                    cout<<"("<<i<<"->"<<terminal[k]<<")$="<<weight[k]<<endl;
                    k=point[k];
                }
        }
}
```

Код программы на Pascal:

```
const
Vmax=100;
Emax=Vmax*(Vmax-1) div 2;       {в случае, если граф полный}
Var
terminal, weight, point: array [1..Emax] of integer;
head, last: array [1..Vmax] of integer;
n, m, v, u, w, k, i: integer;
yn: char;
procedure Add(v, u, w: integer);
    begin
        k:=k+1;
        terminal[k]:=u; weight[k]:=w;
        {если вершина v новая, то
        первая смежная ей вершина имеет номер k}
        if head[v]=0 then head[v]:=k;
        {если вершина v уже просматривалась, то
        следующая смежная с ней вершина имеет номер k}
        if last[v]<>0 then point[last[v]]:=k;
        last[v]:=k;
    end;

begin       {основной блок программы}
    write('Количество вершин > '); read(n);
    write('Количество ребер > '); read(m);
    writeln('Вводите ребра и их вес (v, u, w) > ');
```

```
    for i:=1 to m do                {ввод ребер}
        begin
            read(v, u, w);
            write('Ребро ориентир.? (y/n) > '); read(yn);
            if yn='n' then begin
                Add(v, u, w);
                Add(u, v, w);
                end
            else begin
                Add(v, u, w);
                end;
        end;
    m:=m*2;
    writeln('Список ребер:');
    for i:=1 to m do                {вывод списка ребер}
        begin
            k:=head[i];
            while (k>0) do begin
                writeln('(', i, '->', terminal[k], ')$=', weight[k]);
                k:=point[k];
                end;
        end;
end.
```

Максимально возможное количество вершин в графе задано константой *Vmax*, а ребер – *Emax*. Последняя константа инициализируется выражением *Vmax**(*Vmax*-1)/2, вычисляющим количество ребер в *полном* графе при известном числе вершин. Далее, в программах описываются 5 массивов:

- *terminal* – массив вершин, в которые входят ребра;
- *weight* – массив весов ребер;
- *head*[*i*]=*k* – хранит для *i*-ой вершины номер *k*-ого элемента массивов *terminal* и *weight*, где *terminal*[*k*] – первая вершина смежная с *i*-ой, а *weight*[*k*] – вес инцидентного ей ребра;
- *last*[*i*]=*k* – хранит для *i*-ой вершины номер *k*-ого элемента массивов *terminal* и *weight*, где *terminal*[*k*] – последняя вершина смежная с *i*-ой, а *weight*[*k*] – вес инцидентного ей ребра;
- *point*[*i*]=*k* – хранит для *i*-ой вершины номер *k*-ого элемента массивов *terminal* и *weight*, где *terminal*[*k*] – следующая вершина смежная с *i*-ой, а *weight*[*k*] – вес инцидентного ей ребра.

После ввода количества вершин (*n*) и ребер (*m*) графа, запускается цикл, на каждом шаге которого пользователь вводит с клавиатуры две смежные вершины и вес, лежащего между ними ребра. Если ребро является ориентированным, то функция добавления данных в список ребер (*Add*) вызывается один раз, если неориентированным – два раза. Общее число вызовов функции вычисляется все по тому же выражению *s*+(*p**2), где *s* – ориентированные ребра графа, *p* –

неориентированные. Закончив формирование списка ребер, необходимо вдвое увеличить переменную *m*, т. к. в случае чисто неориентированного графа высота списка будет равна 0+(*m**2).

Осталось вывести на экран получившуюся конструкцию. Вспомним, что на номер первой вершины смежной с *i*-ой вершиной указывает элемент *head*[*i*], следовательно, каждая новая итерация внешнего цикла должна начинаться с взятия этого номера (*k*=*head*[*i*]). Внутренний цикл перестанет выполняться тогда, когда не найдется ни одной смежной с *i*-ой вершины (*k* станет равной нулю), а внешний – по окончанию вывода списка ребер.

3.5 Матрица инцидентности

Говорить о том, что ребро *g* и каждая из вершин *u* и *y* инцидентна *g*, стоит лишь в том случае, если *g* соединяет *u* и *y*. Уяснив это, перейдем к рассмотрению данного метода. Матрица инцидентности строиться по похожему, но не по тому же принципу, что и матрица смежности. Так если последняя имеет размер *n*×*n*, где *n* – число вершин, то матрица инцидентности – *n*×*m*, здесь *n* – число вершин графа, *m* – число ребер. То есть теперь чтобы задать значение какой-либо ячейки, нужно сопоставить не вершину с вершиной, а вершину с ребром.

В каждой ячейки матрицы инцидентности *неориентированного графа* стоит 0 или 1, а в случае *ориентированного графа*, вносятся 1, 0 или -1. То же самое, но наиболее структурировано:

1. неориентированный граф:
   - 1 – вершина инцидентна ребру;
   - 0 – вершина не инцидентна ребру.
2. ориентированный граф:
   - 1 – вершина инцидентна ребру, и является его началом;
   - 0 – вершина не инцидентна ребру;
   - -1 – вершина инцидентна ребру, и является его концом.

Построим матрицу инцидентности сначала для неориентированного графа (рис. 3.10), а затем для орграфа (рис. 3.11). Ребра обозначим буквами от a до e, вершины – цифрами. Все ребра графа не направленны, поэтому матрица инцидентности заполнена положительными значениями.

	a	b	c	d	e
1	1	1	0	0	0
2	1	0	1	1	0
3	0	0	0	1	1
4	0	1	1	0	1

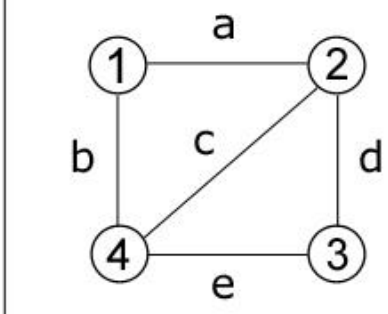


Рисунок 3.10 – Неориентированный граф и его матрица инцидентности

Для орграфа матрица имеет немного другой вид. В каждую из ее ячеек внесено одно из трех значений. Обратите внимание, что нули в двух этих матрицах занимают одинаковые позиции, ведь в обоих случаях структура графа одна. Но некоторые положительные единицы сменились на отрицательные, например, в неориентированном графе ячейка (1, *b*) содержит 1, а в орграфе -1. Дело в том, что в

первом случае ребро *b* не направленное, а во втором – направленное, и, причем вершиной входа для него является вершина «1».

	a	b	c	d	e
1	1	-1	0	0	0
2	-1	0	1	1	0
3	0	0	0	-1	-1
4	0	1	-1	0	1

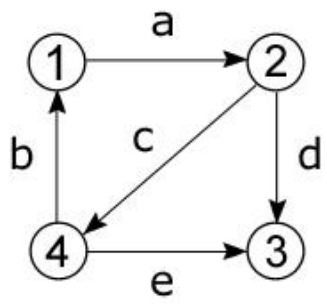


Рисунок 3.11 – Ориентированный граф и его матрица инцидентности

Каждый столбец отвечает за какое-либо одно ребро, поэтому граф, описанный при помощи матрицы инцидентности, всегда будет иметь следующий признак: любой из столбцов матрицы инцидентности содержит две единицы, либо 1 и -1 когда это ориентированное ребро, все остальное в нем – нули.

В программе матрица инцидентности задается, также как и матрица смежности, а именно при помощи двумерного массива. Его элементы могут быть инициализированы при объявлении, либо по мере выполнения программы.

Глава 4. Деревья

4.1 Введение

Дерево, как математический объект, – это абстракция из соименных единиц, встречающихся в природе. Схожесть структуры естественных деревьев с графами определенного вида говорит о допущении установления аналогии между ними. А именно со связанными и вместе с этим *ациклическими* (не имеющими циклов) графами. Последние по своему строению действительно напоминают деревья, но в чем то и имеются различия, например, принято изображать математические деревья с *корнем* расположенным вверху, т. е. все *ветви* растут сверху вниз.

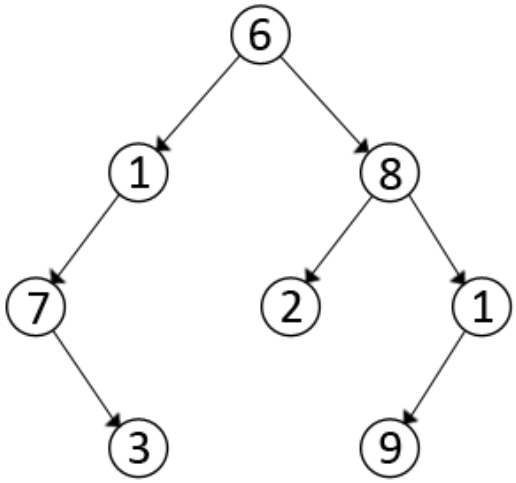


Рисунок 4.1 – Структура данных «Дерево»

Поскольку деревья являются разновидностью графов, у них многие определения совпадают, либо интуитивно схожи. Так *корневой узел* (рис. 4.1, вершина 6) в структуре дерева – это единственная вершина, характерная отсутствием *предков*, т. е. такая, что на нее не ссылается ни какая другая вершина, а из самого корневого узла можно дойти до любой из имеющихся вершин дерева, что следует из свойства связности данной структуры. Узлы, не ссылающиеся ни на какие другие узлы, иначе говоря, ни имеющие потомков называются *листьями* (2, 3, 9), либо *терминальными узлами*. Элементы, расположенные между корневым узлом и листьями – *промежуточные узлы* (1, 1, 7, 8). Каждый узел дерева имеет только одного предка, или если он корневой, то не имеет ни одного.

Поддерево – часть дерева, включающая некоторый корневой узел и все его узлы-потомки. Так, например, на рисунке одно из поддеревьев включает корень 8 и элементы 2, 1, 9.

С деревом можно выполнять многие операции, например, находить элементы, удалять элементы и поддеревья, вставлять поддеревья, находить корневые узлы для некоторых вершин и др. Одной из важнейших операций является *обход дерева*. Выделяются несколько методов обхода. Наиболее популярные из них: *симметричный*, *прямой* и *обратный* обход. При прямом обходе узлы-предки посещаются прежде своих потомков, а в обратном обходе, соответственно, обратная ситуация. В симметричном обходе поочередно просматриваются поддеревья главного дерева.

Представление данных в рассмотренной структуре выгодно в случае наличия у информации явной иерархии. Например, работа с данными о биологических родах и видах, служебных должностях, географических объектах и т. п. требует иерархически выраженной структуры, такой как математические деревья.

4.2 Двоичное дерево

Двоичное дерево – вид связного ациклического (не имеющего циклов) графа, характерный тем, что у каждого из его узлов имеется не более двух потомков (связанных узлов, находящихся иерархически ниже).

Определения следующего абзаца не относятся непосредственно к двоичным деревьям, а скорее к деревьям вообще, поэтому тем, у кого не возникает проблем с понятиями можно перейти к следующему абзацу.

В двоичном дереве есть только один узел, у которого нет предка, он называется *корнем*. Конечные узлы – *листья*. Если у корня отсутствует предок, то у листьев – *потомки*. Все вершины помимо корня и листьев называются *узлами ветвления*. Длина пути от корня до узла определяет *уровень* этого самого узла. Уровень корня дерева всегда равен нулю, а уровень всех его потомков определяется удаленностью от него. Например, узлы *F* и *L* (рис. 4.2) расположены на первом уровне, а *U* и *B* – на третьем.

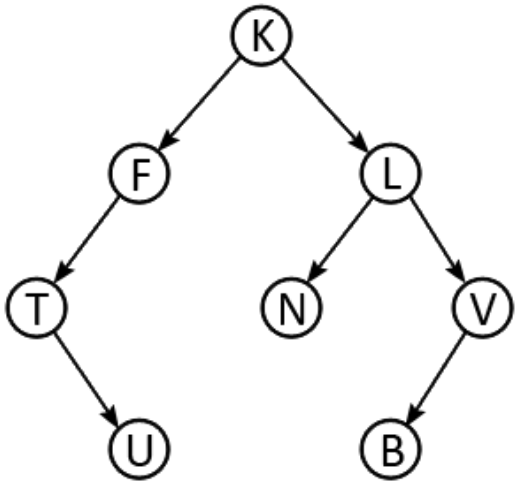


Рисунок 4.2 – Двоичное дерево

Связанный граф является деревом тогда и только тогда, когда *P*-*A*=1, где *P* – количество вершин в графе, а *A* – количество ребер, поскольку в любом дереве с *n* вершинами, должно быть *n*-1 ребро. Это справедливо и для бинарного дерева, так как оно входит в класс деревьев. А увидеть отличительные признаки бинарного дерева, можно просто зная его определение. Достаточно взглянуть на рисунок 1, чтобы понять является ли изображенный на нем граф бинарным деревом. Во-первых, он связный и не имеет ни одного цикла (следовательно, имеем дело с деревом), во-вторых из каждого узла исходит не больше двух ребер (если бы граф был неориентированным, то допускалось три исходящих ребра), что указывает на главный признак двоичного дерева. Но усмотреть в дереве бинарное дерево, можно и немного иным способом, который все же основываются на указанных свойствах. Наше дерево имеет 3 уровня (0,

1, 2, 3), и если выписать количество вершин находящихся на каждом из них, то получиться такой список:

Уровень	Количество вершин
0	1
1	2
2	3
3	2

Если обозначить уровень символом *k*, а количество вершин *n*, то для бинарного дерева будет справедливо равенство $n \leq 2^k$, т. е. количество вершин на *k*-ом уровне не может иметь значение большее, чем степень двойки этого уровня. Для доказательства этого достаточно построить *полное дерево* (рис. 4.3), все уровни которого содержат максимально возможное для двоичного дерева количество вершин.

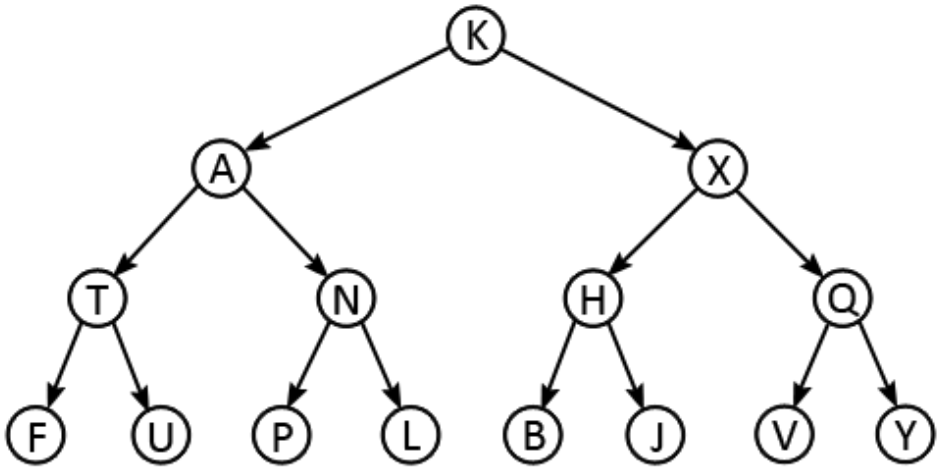


Рисунок 4.3 – Полное двоичное дерево

Продолжая построение, будем получать на каждом новом уровне число вершин равное *k*-ой степени двойки, а их общее количество вычисляется по формуле: $\sum_{i=0}^{n} 2^i$

Для рисунка 2 формула раскрывается так: $2^0+2^1+2^2+2^3=15$.

Операция, при которой вершины дерева поочередно просматриваются и каждая только один раз, называется обходом дерева. Выделяют четыре основных метода обхода:

- обход в ширину;
- прямой обход;
- обратный обход;

- симметричный обход.

Обход в ширину – это поуровневая обработка узлов слева на право. Работа метода заключается в просмотре всех вершин, начиная с некоторой вершины на *n*-ом уровне. Пусть нулевой уровень будет стартовым, тогда, начиная с вершины *K* (рис. 4.3), мы методом обхода в ширину будем поочередно двигаться вниз, просматривая вершины в следующем порядке: K, A, X, T, N, H, Q, F, U, P, L, B, J, V, Y.

Обход в прямом порядке вначале предполагает обработку узлов предков, а потом их потомков, то есть сначала посещается вершина дерева, далее левое и правое поддеревья (именно в таком порядке). Последовательность прямого обхода (рис. 4.3): K, A, T, F, U, N, P, L, X, H, B, J, Q, V, Y.

Обход в обратном порядке противоположен прямому обходу. Первыми осматриваются потомки, а уже затем предки, иначе говоря, первоначально обращение идет к элементам нижних уровней левого поддерева, потом то же самое с элементами правого, и в конце осматривается корень. Порядок обратного обхода все того же дерева: F, U, T, P, L, N, A, B, J, H, V, Y, Q, X, K.

Обход в симметричном порядке заключается в посещении левого узла, перехода в корень и оттуда в правый узел. Симметричный обход: F, T, U, A, P, N, L, K, B, H, J, X, V, Q, Y.

На практике используются не просто бинарные деревья, а их частные случаи, например, такие как двоичное дерево поиска, АВЛ-дерево, двоичная куча и другие.

4.3 Двоичное дерево поиска

Двоичное дерево поиска представляет собой бинарное дерево, обладающее следующим свойством. Значение любого из узлов L_1, L_2, L_3, …, L_n левого поддерева, выходящего из некоторого корня K, всегда меньше значения самого этого корня: $Value(L_i) < Value(K)$, тогда как отношение узлов R_1 , R_2, R_3, …, R_n правого поддерева к корню определяется нестрогим неравенством: $Value(R_i) >= Value(K)$.

Ниже изображено двоичное дерево поиска (рис. 4.4). Из его корня, значение которого 10, выходят два поддерева, причем все левые элементы меньше правых и самого корня, а правые, соответственно, больше. Такое следование продолжается и на следующих уровнях, т. е. если, например, взять поддерево с корнем 16, то значения всех правых элементов будут больше или равны 16, а левых – меньше.

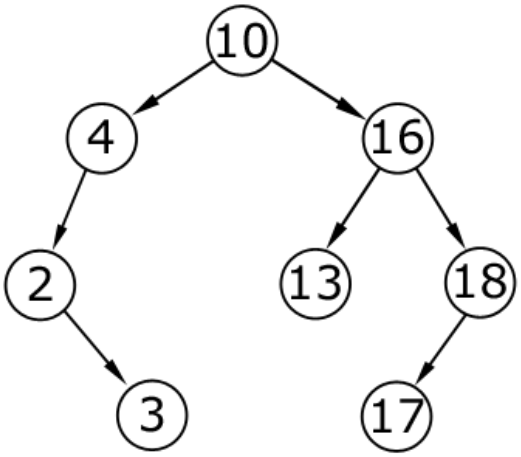


Рисунок 4.4 – Двоичное дерево поиска

Алгоритм поиска по такой структуре данных будет достаточно эффективен, ведь максимальное количество шагов, которое может понадобиться для обнаружения нужного элемента, равно количеству уровней самого бинарного дерева поиска. Поиск элемента в двоичном дереве состоит из следующих шагов:

1. если дерево не содержит в себе элементов, то остановить поиск;
2. иначе сравнить искомый элемент (ключ) с корнем:
    - если значение ключа равно значению корня – поиск завершен;
    - если значение ключа больше значения корня – искать в правой части;
    - если значение ключа меньше значения корня – искать в левой части.

Эти действия выполняются до того момента, пока элемент не будет найден, либо окажется что он вовсе отсутствует.

Рассмотренный метод для изображенного на рисунке выше бинарного дерева поиска, будет работать по такому сценарию. Допустим, значение ключа равно 13, и нужно найти узел с тем же значением. Вначале сравнивается ключ с корнем, значение которого 10. Так как 13 больше 10, далее для поиска следует выбрать правую ветку дерева, поскольку элементы в ней больше 10, что известно из основного свойства бинарного дерева поиска. Далее поиск осуществляется в правом поддереве. Ключ 13 сравнивается с вершиной 16. Первое значение меньше, следовательно, поиск продолжается в левой части, и именно там расположен искомый элемент.

Помимо операции поиска, в рассматриваемую структуру данных эффективно добавлять элементы, а также извлекать их из нее. Во многих ситуациях целесообразным оказывается обычное двоичное дерево перестроить в дерево поиска, отсортировав его элементы одним из алгоритмов сортировки.

4.4 Куча

Имеется список целых чисел 5, 1, 7, 3, 9, 2, 8. Построим два дерева, в которых каждому узлу соответствует одно из значений этого списка, причем первое дерево будет организованно нисходящей иерархией, а второе – восходящей (рис. 4.5).

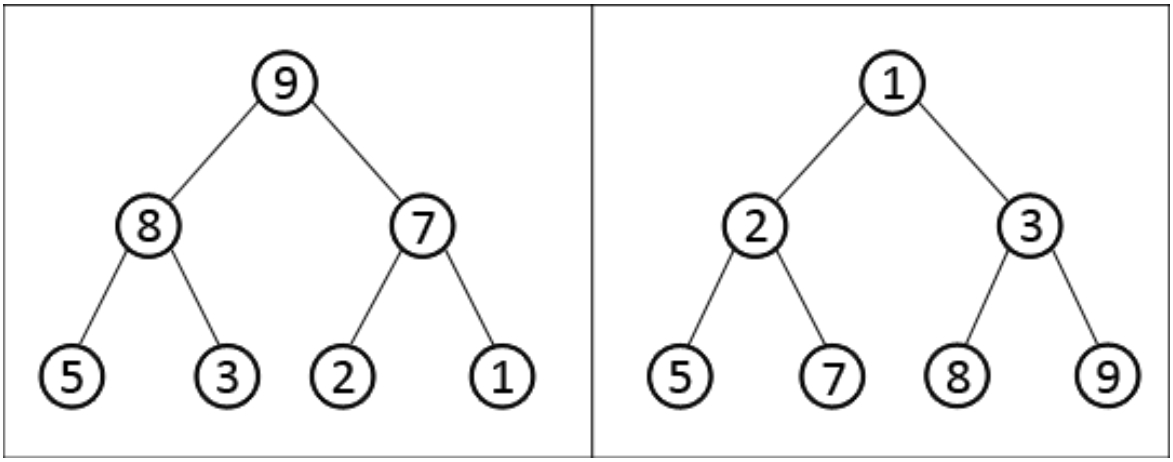


Рисунок 4.5 – Структура данных «Куча»

Когда корневой элемент кучи имеет наибольшее из возможных значений, то ее называют *max-кучей*, а когда наименьшее – *min-кучей*.

Два получившихся дерева полностью удовлетворяют свойству кучи: если n является узлом-предком относительно m, а m соответственно узлом-потомком узла n, то для max-кучи всегда выполняется неравенство $n \geq m$, а для min-кучи $n \leq m$.

Данные, представленные такой структурой, удобны в обработке; некоторые операции с ними оказываются весьма не ресурсоемкими, например, удаление минимального (для min-кучи) или максимального (для max-кучи) элемента, добавление нового элемента и др.

Чтобы лучше понять принципы функционирования такой структуры данных как куча, рассмотрим программу, в которой определяются максимальный и минимальный элементы кучи. В качестве примера будем использовать две кучи, изображенные на рисунке выше, а представим их в программе при помощи простого одномерного массива.

Пусть известно, с каким из видов дерева (min- или max-кучей) имеем дело. Тогда определить максимальный элемент max-кучи, и минимальный элемент min-кучи не составит труда, поскольку в обоих случаях это элемент под одним и тем же номером, а именно первый элемент массива. Составим программу на C++ для решения данной задачи:

```
const int n=7;
int i, j;
string T;
int s;
int min_heap[n]={1, 2, 3, 5, 7, 8, 9};
int max_heap[n]={9, 8, 7, 5, 3, 2, 1};

void Min()
    { cout<<"Корень min-кучи: "<<min_heap[0]; }
void Max()
    { cout<<"Корень max-кучи: "<<max_heap[0]; }

void main ()
    {
        cout<<"min-heap или max-heap? > ";
        cin>>T;
        if (T=="min-heap") Min();
        else if (T=="max-heap") Max();
        else cout<<"Ошибка!";
    }
```

Столкнуться со структурами данных типа кучи в программировании можно не раз. Так, например, их используют для оптимизации некоторых алгоритмов, многие популярные языки оперируют ими. Кроме того существуют различные варианты этой структуры, каждый из которых характерен своими специфическими свойствами.

4.5 АВЛ-дерево

АВЛ-дерево – структура данных, изобретенная в 1968 году двумя советскими математиками: Евгением Михайловичем Ландисом и Георгием Максимовичем Адельсон-Вельским. Прежде чем дать конструктивное определение АВЛ-дереву, сделаем это для сбалансированного двоичного дерева поиска. Сбалансированным называется такое двоичное дерево поиска, в котором высота каждого из поддеревьев, имеющих общий корень, отличается не более чем на некоторую константу k, и при этом выполняются условия характерные для двоичного дерева поиска. АВЛ-дерево – сбалансированное двоичное дерево поиска с k=1. Для его узлов определен *коэффициент сбалансированности* (*balance factor*). Balance factor – это разность высот правого и левого поддеревьев, принимающая одно значение из множества {-1, 0, 1}. Ниже изображен пример АВЛ-дерева, каждому узлу которого поставлен в соответствие его реальный коэффициент сбалансированности.

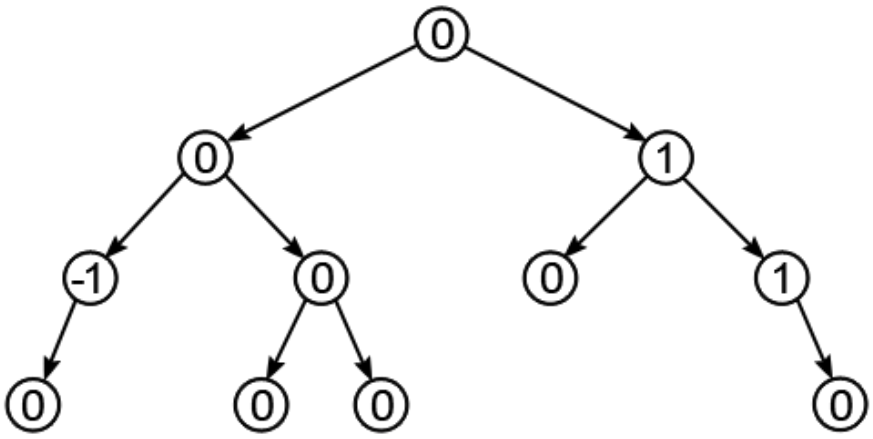


Рисунок 4.6 – АВЛ-дерево

Положим B_i – коэффициент сбалансированности узла T_i (i – номер узла, отсчитываемый сверху вниз от корневого узла по уровням слева направо). Balance factor узла T_i рассчитывается следующим образом. Пусть функция $h()$ с параметрами T_i и L возвращает высоту левого поддерева L узла T_i, а с T_i и R – правого. Тогда $B_i=h(T_i, R)-h(T_i, L)$. Например, B_4=-1, так как $h(T_4, R)-h(T_4, L)$=0-1=-1.

Сбалансированное дерево эффективно в обработке, что следует из следующих рассуждений. Максимальное количество шагов, которое может потребоваться для обнаружения нужного узла, равно количеству уровней самого бинарного дерева поиска. А так как поддеревья сбалансированного дерева, «растущие» из произвольного корня, практически симметричны, то и его листья расположены на сравнительно невысоком уровне, т. е. высота дерева сводиться к оптимальному минимуму. Поэтому критерий баланса положительно сказывается на общей производительности. Но в

процессе обработки АВЛ-дерева, балансировка может нарушиться, тогда потребуется осуществить операцию балансировки. Помимо нее, над АВЛ-деревом определены операции вставки и удаления элемента, выполнение которых может привести к дисбалансу дерева.

Доказано, что высота АВЛ-дерева, имеющего *n* узлов, примерно равна $log_2 n$. Беря в виду это, а также то, то, что время выполнения операций добавления и удаления напрямую зависит от операции поиска, получим временную сложность трех операций для худшего и среднего случая – $O(log\ n)$.

Прежде чем рассматривать основные операции над АВЛ-деревом, определим структуру для представления его узлов, а также три специальные функции:

```
struct Node
    {
        int key;
        char height;
        Node *right;
        Node *left;
        Node(int k) { key=k; height=1; left=right=0; }
    };

char height(Node *p)
    {
        if (p) return p->height;
        else return 0;
    }

int BF(Node *p)
    { return height(p->right)-height(p->left); }

void OverHeight(Node *p)
    {
        char hleft=height(p->left);
        char hright=height(p->right);
        p->height=(hleft>hright ? hleft : hright)+1;
    }
```

Структура *Node* описывает узлы АВЛ-дерева. Ее поля *right* и *left* являются указателями на правое и левое поддеревья. Поле *key* хранит ключ узла, *height* – высоту поддерева. Функция-конструктор создает новый узел. Функции *height* и *BF* вычисляют коэффициент сбалансированности узла, а *OverHeight* – корректирует значение поля *height*, затронутое в процессе балансировки.

4.5.1 Балансировка

Если после выполнения операции добавления или удаления, коэффициент сбалансированности какого-либо узла АВЛ-дерева становиться равен 2, т. е. $|h(T_i, R)-h(T_i, L)|=2$, то необходимо выполнить операцию балансировки. Она осуществляется путем вращения (поворота) узлов – изменения связей в поддереве. Вращения не меняют свойств бинарного дерева поиска, и выполняются за константное время. Всего различают 4 их типа:

- малое правое вращение;
- большое правое вращение;
- малое левое вращение;
- большое левое вращение.

Оба типа больших вращений являются комбинацией малых вращений (право-левым или лево-правым вращением).

Возможны два случая нарушения сбалансированности. Первый из них исправляется 1 и 3 типом, а второй – 2 и 4. Рассмотрим первый случай. Пусть имеется следующее сбалансированное поддерево:

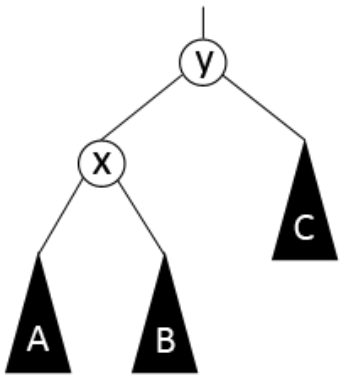


Рисунок 4.7 – Сбалансированное дерево

Здесь *x* и *y* – узлы, а *A*, *B*, *C* – поддеревья. После добавления к поддереву *A* узла *v*, баланс нарушиться, и потребуется балансировка. Она осуществляется правым поворотом (тип 1) узла *y*:

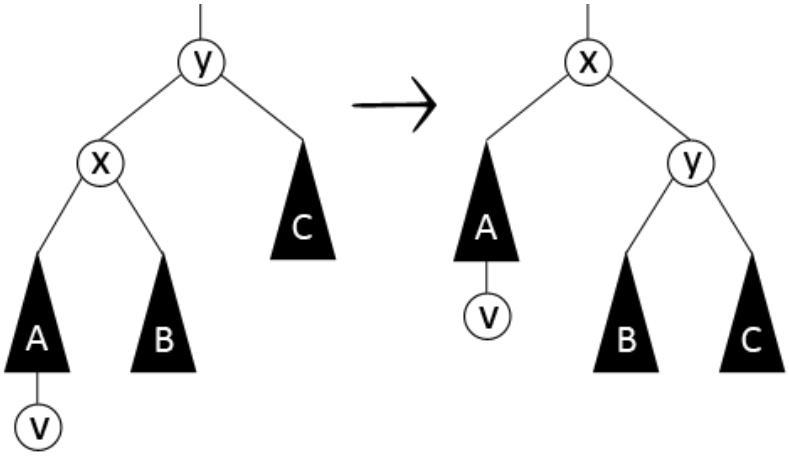


Рисунок 4.8 – Балансировка малым правым поворотом

Малое левое вращение выполняется симметрично малому правому. Следующие две функции выполняют малый правый и малый левый повороты.

```
Node* RightRotation(Node *x)     //малый правый поворот
    {
                Node *y=x->left;
                x->left=y->right;
                y->right=x;
                OverHeight(x);
                OverHeight(y);

                return y;

    }

Node *LeftRotation(Node *y)      //малый левый поворот
    {
                Node *x=y->right;
                y->right=x->left;
                x->left=y;
                OverHeight(y);
                OverHeight(x);
                return x;
    }
```

Второй случай дисбаланса исправляется большим правым или большим левым вращением. Пусть имеется следующее сбалансированное поддерево:

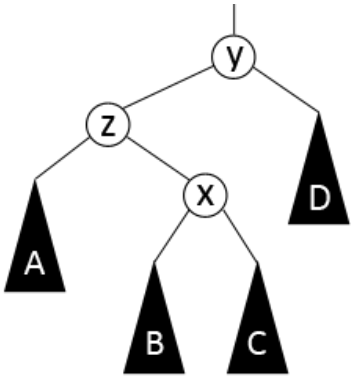


Рисунок 4.9 – Сбалансированное дерево

Вставка узлов в поддерево *A* или *D*, не нарушит сбалансированности, но добавление их в *B* или *C* приведет к необходимости произвести балансировку вращением 2-ого типа (рис. 4.10).

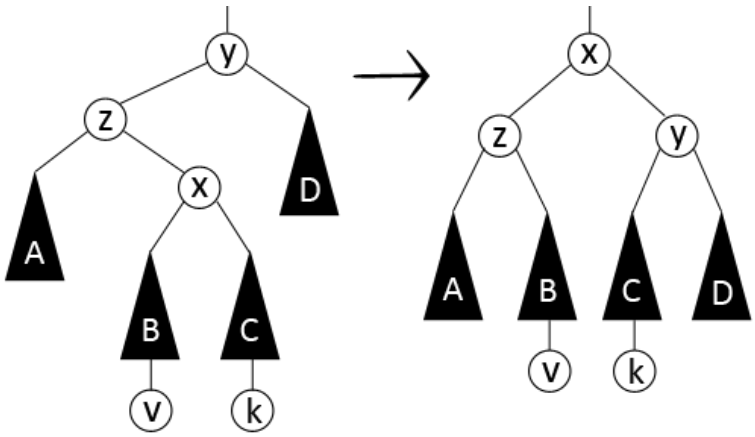


Рисунок 4.10 – Балансировка большим правым поворотом

Большое левое вращение выполняется симметрично большому правому. Функция *Balance* выполняет балансировку узла путем вращения его поддеревьев:

```
Node *Balance(Node *x)
    {
                OverHeight(x);
                if (BF(x)==2)
                    {
                        if (BF(x->right)<0) x->right=RightRotation(x->right);
                        return LeftRotation(x);
                    }
                if (BF(x)==-2)
                {
                    if (BF(x->left)>0) x->left=LeftRotation(x->left);
                    return RightRotation(x);
```

```
        }
        return x;
    }
```

Данная функция проверяет условия, и в зависимости от результата балансирует узел x, применяя один из типов вращения.

4.5.2 Добавление узлов

Операция вставки нового узла в АВЛ-дерево выполняется рекурсивно. По ключу данного узла, производиться поиск места вставки: спускаясь вниз по дереву, алгоритм сравнивает ключ добавляемого узла со встречающимися ключами, далее происходит вставка нового элемента; по возвращению из рекурсии, выполняется проверка всех показателей сбалансированности узлов и, в случае необходимости, выполняется балансировка. Для осуществления балансировки следует знать, с каким из рассмотренных выше случаев дисбаланса имеем дело. Допустим, мы добавили узел x в левое поддерево, для которого выполнялось $h(T_i, R) < h(T_i, L)$, т. е. высота левого поддерева изначально превышала высоту правого. Если левое поддерево этого узла выше правого, то потребуется большое вращение, иначе – малое.

Функция добавления узла:

```
Node *Insert(Node *x, int k)
    {
        if (!x) return new Node(k);
        if (k<x->key) x->left=Insert(x->left, k);
        else x->right=Insert(x->right, k);
        return Balance(x);
    }
```

4.5.3 Удаление узлов

Также как и вставку узла, его удаление удобно задать рекурсивно. Пусть *x* – удаляемый узел, тогда если *x* – лист (терминальный узел), то алгоритм удаления сводиться к простому исключению узла *x*, и подъему к корню с переопределением balance factor’ов узлов. Если же *x* не является листом, то он либо имеет правое поддерево, либо не имеет его. Во втором случае, из свойства АВЛ-дерева, следует, что левое поддерево имеет высоту 1, и здесь алгоритм удаления сводиться к тем же действиям, что и при терминальном узле. Остается ситуация когда у *x* есть правое поддерево. В таком случае нужно в правом поддереве отыскать следующий по значению за *x* узел *y*, заменить *x* на *y*, и рекурсивно вернуться к корню, переопределяя коэффициенты сбалансированности узлов. Из свойства двоичного дерева поиска следует, что узел *y* имеет наименьшее значение среди всех узлов правого поддерева узла *x*.

Для программной реализации операции удаления узла опишем функцию *Delete*:

```
Node *Delete(Node *x, int k)
    {
        if (!x) return 0;
        if (k<x->key) x->left=Delete(x->left, k);
        else if (k>x->key) x->right=Delete(x->right, k);
        else
            {
                Node *y=x->left;
                Node *z=x->right;
                delete x;
                if (!z) return y;
                Node* min=SearchMin(z);
                min->right=DeleteMin(z);
                min->left=y;
                return Balance(min);
            }
        return Balance(x);
    }
```

Из нее вызываются вспомогательные функции: *SearchMin* и *DeleteMin*. Первая ищет минимальный элемент в правом поддереве, вторая удаляет его. Опишем эти вспомогательные функции:

```
Node *SearchMin(Node *x)
    {
        if (x->left) return SearchMin(x->left);
        else return x;
    }

Node *DeleteMin(Node *x)
    {
        if (x->left==0) return x->right;
        x->left=DeleteMin(x->left);
        return Balance(x);
    }
```

Операция удаления реализуется определенно сложнее, чем операция добавления. Да и последствия ее выполнения могут потребовать поворота в каждом узле. Но какова вероятность возникновения ситуации, при которой появится потребность в поворотах? Этим вопросом задается Никлаус Вирт в своей книге «Алгоритмы и структуры», и отвечает на него: «Удивительный результат эмпирических проверок показал, что в то время как один поворот вызывается приблизительно каждыми двумя включениями, тем не менее, при удалении мы имеем дело с одним поворотом на целых пять удалений».

Раздел II

АЛГОРИТМЫ

Введение

Понятие *алгоритма* – одно из базовых в информатике и главное в программировании. Алгоритм – это совокупность инструкций исполнителю, выполнять конечную последовательность действий для достижения решения им задачи за конечное число шагов. Под исполнителем, чаще всего, понимается некоторый механизм (компьютер, станок), способный понимать и выполнять те или иные команды. Но в качестве исполнителя может выступать, например, человек или животное, т. к. алгоритм это не обязательно сложные, понятные лишь компьютеру инструкции. Действие схожее с приготовлением пищи или походом в магазин также является алгоритмом.

Слово «алгоритм» происходит от имени средневекового персидского ученого Абу Абдуллах Мухаммеда ибн Муса аль-Хорезми (алгоритм – аль-Хорезми). Оно впервые упоминается в его сочинении, датированным примерно 825-ым годом.

Алгоритм должен быть описан согласно некоторым правилам. Различают следующие способы описания алгоритма:

- словесное описание (с помощью естественного языка);
- графическое описание (с помощью блок-схем);
- программное описание (с помощью языка программирования);
- псевдокод (словесное описание + программное описание).

Выделяют следующие виды алгоритмов:

- *линейный алгоритм* – набор инструкций, выполняемых последовательно во времени друг за другом;
- *разветвляющийся алгоритм* – алгоритм, в котором направление обработки информации зависит от результата выполнения логического условия;
- *циклический алгоритм* – алгоритм, предусматривающий многократное повторение одного и того же действия над исходными данными.

Алгоритм обладает следующими свойствами:

- *дискретность* – процесс решения задачи алгоритм должен представляться как последовательное выполнение некоторых простых шагов;
- *детерминированность* – выдача алгоритмом одного и того же результата для одних и тех же исходных данных;
- *понятность* – использование только понятных исполнителю команд;
- *конечность* – завершенность работы алгоритма и выдача им результата за конечное число шагов;
- *универсальность* (массовость) – возможность применения алгоритма к разным наборам исходных данных;
- *результативность* – завершение алгоритма определёнными результатами.

Глава 5. Анализ сложности алгоритмов

5.1 Введение

Для решения задачи часто приходится выбирать метод как из числа алгоритмов различных по принципу своей работы, так и из числа возможных реализаций одного алгоритма. За конечное число шагов, при разных исходных данных, все они приведут к правильному решению задачи. Но из всего спектра вариантов, следует выбирать наиболее оптимальные методы. Критерием оптимальности является *сложность алгоритма*. Выделяют *временную* и *пространственную* сложность. Временная сложность определяется количеством элементарных операций (инструкций), совершаемых алгоритмом для решения им поставленной задачи. Пространственная сложность измеряется объемом затраченной алгоритмом памяти. Далее будет рассматриваться только временная сложность.

Выделяют два основных класса алгоритмов: алгоритмы с повторением и рекурсивные алгоритмы. В основе алгоритмов с повторением лежат операторы цикла и условные операторы. Анализ алгоритмов этого класса потребует подсчета всех циклов и операций внутри них. Рекурсивные алгоритмы (рекурсивная функция – функция вызывающая сама себя) разбивают основную задачу на части и решают отдельно каждую из них. Анализ рекурсивного алгоритма, как правило, сложнее. Он требует подсчета числа операций разбиения задачи, выполнения алгоритма на каждой из частей основной задачи, а также их объединения.

Какие инструкции считать? Ответ на этот вопрос не всегда очевиден, но, по крайней мере, мнения сходятся в одном – при подсчете следует учитывать только существенные операции. К ним обычно относят:

- простое присваивание: a ← b;
- индексация элемента массива (нахождение его значения);
- арифметические операции: -, +, *, /;
- операции сравнения: <, >, =, <=, >=, <>;
- логические операции: or, and, not.

На временную сложность алгоритма оказывает значительное влияние объем входных данных. Так, если при обработке небольшого объема данных разница между работой двух различных алгоритмов покажется несущественной, то увеличение этого объема ощутимо скажется на времени выполнения каждого из них. Но временная сложность часто зависит не только от объема, но и от значений данных, а также порядка их поступления. Например, многие алгоритмы сортировки потратят гораздо меньше времени на упорядочивание массива, если он уже отсортирован. Из-за подобных трудностей рассматривают понятие временной сложности в трех случаях: худшем, лучшем и среднем. Худший случай соответствует наименее удачному набору входных данных, когда алгоритму для решения задачи приходится выполнить

максимальное число элементарных операций. В лучшем случае набор входных данных обеспечит минимально возможное число операций. Средний случай определяется гораздо сложнее. Входные данные разбиваются на возможные группы, которыми они могут быть. Далее, определяется вероятность появления каждой группы. После чего досчитывается время работы алгоритма на данных каждой группы. Для нас наибольший интерес представляют наиболее и наименее благоприятные случаи.

Посчитаем число инструкций алгоритма линейного поиска:

```
for (i=0; i<n; i++)
    if (A[i]==key) return i;
return -1;
```

В лучшем случае искомому элементу (ключу) равно значение первого элемента массива. Тогда потребуется всего четыре операции:

- в заголовке цикла: присвоение переменной *i* начального значения и сравнение этого значения со значением *n;*
- в операторе ветвления: индексация *i*-ого элемента и сравнение его с ключом.

В худшем случае искомый элемент окажется в конце массива, либо он вовсе будет отсутствовать. В таком случае цикл выполнит *n* итераций, и число операций возрастет до 2+4*n*:

- в заголовке цикла: начальное присвоение и первое сравнение, а также *n* сравнений и *n* увеличений значения счетчика: 2+2*n*;
- в операторе ветвления: *n* индексаций и *n* сравнений: 2*n*.

Теперь посчитаем количество инструкций алгоритма нахождения максимального элемента:

```
Max=A[0];
for (i=1; i<n; i++)
    {
        if (A[i]>Max)
        Max=A[i];
    }
```

В первой строке выполняются две операции: нахождение в массиве *A* элемента с индексом 0 и присвоение значения этого элемента переменной *Max*. Перед запуском тела цикла выполняются еще две операции: инициализация переменной счетчика *i* и сравнение значения этой переменной с числом элементов *n*. Цикл выполнит *n*-1 итерацию, за это время *i* инкрементируется *n*-1 раз и условие (*i*<*n*) проверится *n*-1 раз. Тело цикло выполнилось *n*-1 раз, следовательно, оператор ветвления сравнил значения элементов *n*-1 раз, выполнив тем самым по две инструкции на каждой итерации:

нахождение *i*-ого элемента и сравнение его с *Max*. Получаем количество операций алгоритма: 2+2+4(*n*-1) = 4+4*n*-4 = 4*n*.

Такой будет сложность алгоритма в лучшем случае, т. е. максимальный элемент соответствовал первому элементу массива, поэтому оператор ветвления в теле цикла не выполнялся. В худшем случае массив будет упорядочен по возрастанию, а значит, на каждой итерации потребуется заменить значение переменной *Max*. Это потребует еще по две инструкции на каждой итерации: нахождения *i*-ого элемента и присвоения его значения переменной *Max*. В итоге получим количество операций для наихудшего случая: 4*n*+2*n*=6*n*.

5.2 Асимптотический анализ

Анализ сравнения затрат времени алгоритмов, выполняемых решение экземпляра некоторой задачи, при больших объемах входных данных, называется *асимптотическим*. Алгоритм, имеющий меньшую асимптотическую сложность, является наиболее эффективным.

В асимптотическом анализе, *сложность алгоритма* – это функция, позволяющая определить, как быстро увеличивается время работы алгоритма с увеличением объёма данных.

Основные оценки роста, встречающиеся в асимптотическом анализе:

- *O* (О-большое) – верхняя асимптотическая оценка роста временной функции;
- Ω (Омега) – нижняя асимптотическая оценка роста временной функции;
- Θ (Тета) – нижняя и верхняя асимптотические оценки роста временной функции.

Пусть n – величина объема данных. Тогда рост функции алгоритма $f(n)$ можно ограничить функций $g(n)$ асимптотически:

Обозначение	Описание
$f(n) \in O(g(n))$	f ограничена сверху функцией g с точностью до постоянного множителя
$f(n) \in \Omega(g(n))$	f ограничена снизу функцией g с точностью до постоянного множителя
$f(n) \in \Theta(g(n))$	f ограничена снизу и сверху функцией g

Например, время уборки помещения линейно зависит от площади этого самого помещения ($\Theta(S)$), т. е. с ростом площади в n раз, время уборки увеличиться также в n раз. Поиск имени в телефонной книге потребует линейного времени $O(n)$, если воспользоваться алгоритмом линейного поиска (см. п. 6.1), либо времени, логарифмически зависящего от числа записей ($O(\log_2(n))$), в случае применения двоичного поиска (см. п. 6.2).

Для нас наибольший интерес представляет *O*-функция. Кроме того, в последующих главах, сложность алгоритмов будет даваться только для верхней асимптотической границы.

Под фразой «сложность алгоритма есть $O(f(n))$» подразумевается, что с увеличением объема входных данных n, время работы алгоритма будет возрастать не быстрее, чем некоторая константа, умноженная на $f(n)$.

Важные правила асимптотического анализа:

1. $O(k*f) = O(f)$ – постоянный множитель k (константа) отбрасывается, поскольку с ростом объема данных, его смысл теряется, например:
$$O(9,1n) = O(n)$$
2. $O(f*g) = O(f)*O(g)$ – оценка сложности произведения двух функций равна произведению их сложностей, например:
$$O(5n*n) = O(5n)*O(n) = O(n)*O(n) = O(n*n) = O(n^2)$$
3. $O(f/g)=O(f)/O(g)$ – оценка сложности частного двух функций равна частному их сложностей, например:
$$O(5n/n) = O(5n)/O(n) = O(n)/O(n) = O(n/n) = O(1)$$
4. $O(f+g)$ равна доминанте $O(f)$ и $O(g)$ – оценка сложности суммы функций определяется как оценка сложности доминанты первого и второго слагаемых, например:
$$O(n^5+n^{10}) = O(n^{10})$$

Подсчет количества операций – дело утомительное и, что важно, совсем не обязательное. Исходя из выше перечисленных правил, чтобы определить сложность алгоритма, не нужно, как мы это делали прежде, считать все операции, достаточно знать какой сложностью обладает та или иная конструкция алгоритма (оператор или группа операторов). Так, алгоритм, не содержащий циклов и рекурсий, имеет константную сложность $O(1)$. Сложность цикла, выполняющего n итераций, равна $O(n)$. Конструкция их двух вложенных циклов, зависящих от одной и той же переменной n, имеет квадратичную сложность $O(n^2)$.

Вот наиболее часто встречающиеся классы сложности:

- $O(1)$ – константная сложность;
- $O(n)$ – линейная сложность;
- $O(n^a)$ – полиномиальная сложность;
- $O(\log(n))$ – логарифмическая сложность;
- O(n*log(n)) – квазилинейная сложность;
- $O(2^n)$ – экспоненциальная сложность;
- $O(n!)$ – факториальная сложность.

Глава 6. Сортировка

6.1 Сортировка пузырьком

Сортировка пузырьком (*обменная сортировка*) – простой в реализации и малоэффективный алгоритм сортировки. Метод изучается одним из первых на курсе теории алгоритмов, в то время как на практике используется очень редко.

Идея алгоритма заключается в следующем. Соседние элементы последовательности сравниваются между собой и, в случае необходимости, меняются местами. В качестве примера (рис. 6.1) рассмотрим упорядочивание методом пузырьковой сортировки массива, количество элементов *n* которого равно 5: 9, 1, 4, 7, 5. В итоге должен получиться массив с элементами, располагающимися в порядке возрастания их значений.

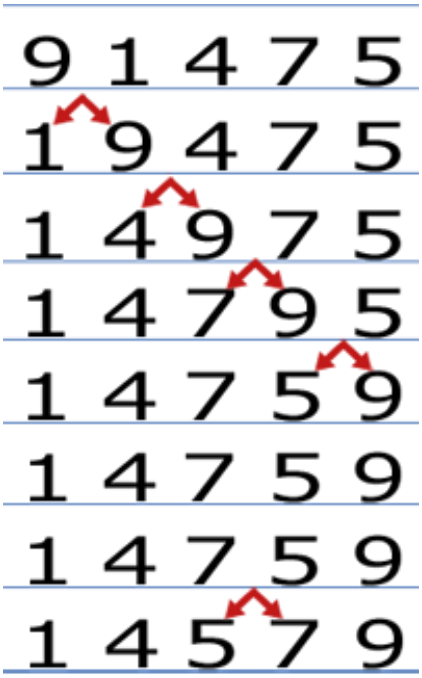


Рисунок 6.1 – Пример пузырьковой сортировки

Вначале сравниваются два первых элемента последовательности: 9 и 1. Так как значение первого элемента больше значения второго, т. е. 9>1, они меняются местами. Далее, сравниваются второй и третий элементы: девятка больше четверки, следовательно, элементы снова обмениваются позициями. Аналогично алгоритм продолжает выполняться до тех пор, пока все элементы массива не окажутся на своих местах. Всего для этого потребуется $n*(n-1)$ сравнений. В частности, на данной последовательности произведено 20 сравнений и только 5 перестановок.

Код программы на C++:

```
int i, j, count, key;
void BubbleSort(int A[], int n) //сортировка пузырьком
	{
		for (i=0; i<n-1; i++)
			{
				for (j=1; j<n; j++)
					{
						key=j;
						count=A[j-1];
						if (A[key-1]>A[j])
							{
								A[key-1]=A[j];
								A[j]=count;
							}
					}
			}
		cout<<"Результирующий массив: ";
		for (i=0; i<n; i++) cout<<A[i]<<" "; //вывод массива
	}
void main ()	//главная функция
	{
	int n; int A[1000];
	cout<<"Количество элементов > "; cin>>n;
	for (i=0; i<n; i++) //ввод массива
		{ cout<<i+1<<" элемент > "; cin>>A[i]; }
	BubbleSort(A, n);
	}
```

Код программы на Pascal:

```
type arr=array[1..1000] of integer;
var A: arr;
n, i, j: integer;
procedure BubbleSort(A: arr; n: integer); {сортировка пузырьком}
	var key, count: integer;
	begin
		for i:=1 to n do
			begin
				for j:=1 to n-1 do
					begin
						key:=j;
						count:=A[j];
						if A[key]>A[j+1] then
							begin
								A[key]:=A[j+1];
								A[j+1]:=count;
							end
```

```
                    end
                end;
            write('Результирующий массив: ');
            for i:=1 to n do write(A[i],' '); {вывод массива}
        end;

begin       {основной блок программы}
    write('Количество элементов > '); read(n);
    for i:=1 to n do {ввод массива}
        begin write(i, ' элемент > '); read(A[i]); end;
    BubbleSort(A, n);
end.
```

Приведенный код можно улучшить, а именно – вдвое уменьшить количество выполняемых сравнений. Для этого достаточно с каждым шагом *i* внешнего цикла на *i* уменьшать количество итераций внутреннего цикла. Ниже показан основной фрагмент алгоритма обменной сортировки для языков C++ и Pascal, его улучшенный вариант.

Код программы на C++:

```
for (i=0; i<n-1; i++)
    {
        for (j=0; j<n-(i+1); j++)
            {
                key=j+1;
                count=A[key];
                if (A[j]>A[key])
                    {
                        A[key]=A[j];
                        A[j]=count;
                    }
            }
    }
```

Код программы на Pascal:

```
for i:=1 to n-1 do
    begin
        for j:=1 to n-i do
            begin
                key:=j+1;
                count:=A[key];
                if A[j]>A[key] then
                    begin
                        A[key]:=A[j];
                        A[j]:=count;
                    end
            end
    end;
```

Как отмечалось, алгоритм редко используется на практике, поскольку имеет низкую производительность. В лучшем случае сортировка пузырьком потребует $O(n)$ времени, а в среднем и худшем – $O(n^2)$.

6.2 Сортировка выбором

Сортировка выбором – возможно, самый простой в реализации алгоритм сортировки. Как и в большинстве других подобных алгоритмов, в его основе лежит операция сравнения. Сравнивая каждый элемент с каждым, и в случае необходимости производя обмен, метод приводит последовательность к необходимому упорядоченному виду.

Идея алгоритма очень проста. Пусть имеется массив A размером n, тогда сортировка выбором сводится к следующему:

1. берем первый элемент последовательности $A[i]$, здесь i – номер элемента, для первого i равен 1;
2. находим минимальный (максимальный) элемент последовательности и запоминаем его номер в переменную *key*;
3. если номер первого элемента и номер найденного элемента не совпадают, т. е. если $key \neq 1$, тогда два этих элемента обмениваются значениями, иначе никаких манипуляций не происходит;
4. увеличиваем i на 1 и продолжаем сортировку оставшейся части массива, а именно с элемента с номером 2 по n, так как элемент $A[1]$ уже занимает свою позицию.

С каждым последующим шагом размер подмассива, с которым работает алгоритм, уменьшается на 1, но на способ сортировки это не влияет, он одинаков для каждого шага.

Рассмотрим работу алгоритма на примере конкретной последовательности целых чисел. Дан массив (рис. 6.2), состоящий из пяти целых чисел 9, 1, 4, 7, 5. Требуется расположить его элементы по возрастанию, используя сортировку выбором. Начнем по порядку сравнивать элементы. Второй элемент меньше первого – запоминаем это ($key=2$). Далее, мы видим, что он также меньше и всех остальных, а так как $key \neq 1$, меняем местами первый и второй элементы. Продолжим упорядочивание оставшейся части, пытаясь найти замену элементу со значением 9. Теперь в *key* будет занесена 3-ка, поскольку элемент с номером 3 имеет наименьшее значение. Как видно, $key \neq 2$, следовательно, меняем местами 2-ой и 3-ий элементы. Продолжаем расставлять на места элементы, пока на очередном шаге размер поддмассива не станет равным 1-ому.

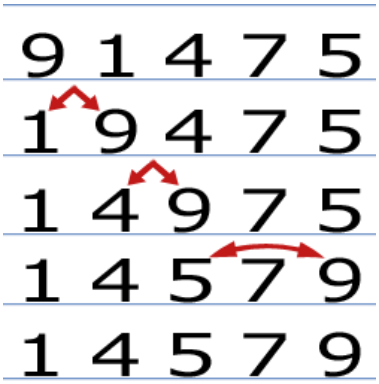


Рисунок 6.2 – Пример сортировки выбором

Код программы на C++:

```
int i, j;
void SelectionSort(int A[], int n) //сортировка выбором
	{
		int count, key;
		for (i=0; i<n-1; i++)
			{
				count=A[i]; key=i;
				for (j=i+1; j<n; j++)
				if (A[j]<A[key]) key=j;
				if (key!=i)
					{
						A[i]=A[key];
						A[key]=count;
					}
			}
		cout<<"Результирующий массив: ";
		for (i=0; i<n; i++) cout<<A[i]<<" "; //вывод массива
	}

void main()	//главная функция
	{
		int n, A[1000];
		cout<<"Количество элементов > "; cin>>n;
		for (i=0; i<n; i++) //ввод массива
			{
				cout<<i+1<<" элемент > ";
				cin>>A[i];
			}
		SelectionSort(A, n);
	}
```

Код программы на Pascal:

```
const max=1000;
type arr=array[1..max] of integer;
var i, j, n: integer;
A: arr;
procedure SelectionSort(A: arr; n: integer); {сортировка выбором}
    var key, count: integer;
    begin
        for i:=1 to n do
            begin
                count:=A[i]; key:=i;
                for j:=i+1 to n do
                    if (A[key]>A[j]) then key:=j;
                if (key<>i) then
                    begin
                        A[i]:=A[key];
                        A[key]:=count;
                    end;
            end;
        write('Результирующий массив: ');
        for i:=1 to n do write(A[i], ' '); {вывод массива}
    end;

begin    {основной блок программы}
    write('Количество элементов > ');
    read(n);
    for i:=1 to n do {ввод массива}
        begin
            write(i,' элемент > ');
            read(A[i]);
        end;
    SelectionSort(A, n);
end.
```

Сортировка выбором проста в реализации, и в некоторых ситуациях стоит предпочесть ее наиболее сложным и совершенным методам. Но в большинстве случаев данный алгоритм уступает в эффективности последним, так как в худшем, лучшем и среднем случае ей потребуется $O(n^2)$ времени.

6.3 Сортировка вставками

Сортировка вставками – простой алгоритм сортировки, преимущественно использующийся в учебном программировании. К положительной стороне метода относится простота реализации, а также его эффективность на частично упорядоченных последовательностях, и/или состоящих из небольшого числа элементов. Тем не менее, высокая вычислительная сложность не позволяет рекомендовать алгоритм в повсеместном использовании.

Рассмотрим алгоритм сортировки вставками на примере колоды игральных карт. Процесс их упорядочивания по возрастанию (в колоде карты расположены в случайном порядке) будет следующим. Обратим внимание на вторую карту, если ее значение меньше первой, то меняем эти карты местами, в противном случае карты сохраняют свои позиции, и алгоритм переходит к шагу 2. На 2-ом шаге смотрим на третью карту, здесь возможны четыре случая отношения значений карт:

1. первая и вторая карта меньше третьей;
2. первая и вторая карта больше третьей;
3. первая карта уступает значением третьей, а вторая превосходит ее;
4. первая карта превосходит значением третью карту, а вторая уступает ей.

В первом случае не происходит никаких перестановок. Во втором – вторая карта смещается на место третьей, первая на место второй, а третья карта занимает позицию первой. В предпоследнем случае первая карта остается на своем месте, в то время как вторая и третья меняются местами. Ну и наконец, последний случай требует рокировки лишь первой и третьей карт. Все последующие шаги полностью аналогичны расписанным выше.

Рассмотрим на примере числовой последовательности процесс сортировки методом вставок (рис. 6.3). Клетка, выделенная темно-серым цветом – активный на данном шаге элемент, ему также соответствует *i*-ый номер. Светло-серые клетки это те элементы, значения которых сравниваются с *i*-ым элементом. Все, что закрашено белым – не затрагиваемая на шаге часть последовательности.

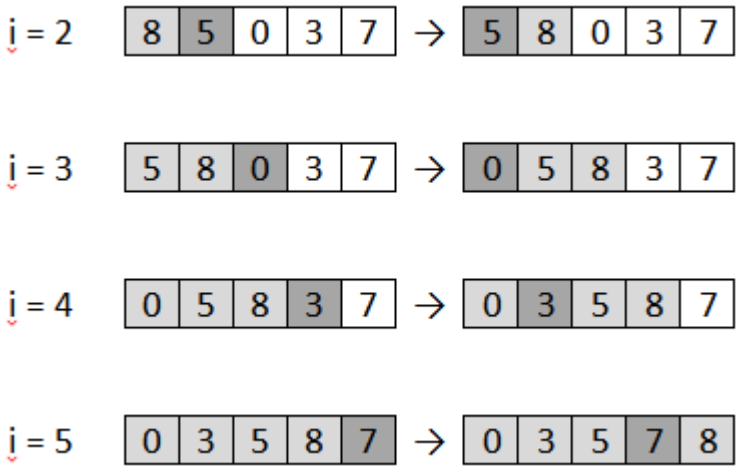


Рисунок 6.3 – Пример сортировки вставками

Таким образом, получается, что на каждом этапе выполнения алгоритма сортируется некоторая часть массива, размер которой с шагом увеличивается, и в конце сортируется весь массив целиком.

Код программы на C++:

```
int i, j, key=0, temp=0;
void InsertSort(int *mas, int n) //сортировка вставками
    {
        for (i=0; i<n-1; i++)
            {
                key=i+1;
                temp=mas[key];
                for (j=i+1; j>0; j--)
                    {
                        if (temp<mas[j-1])
                            {
                                mas[j]=mas[j-1];
                                key=j-1;
                            }
                    }
                mas[key]=temp;
            }
        cout<<endl<<"Результирующий массив: ";
        for (i=0; i<n; i++) //вывод массива
        cout<<mas[i]<<" ";
    }

void main ()    //главная функция
    {
        int n;
        cout<<"Количество элементов в массиве > "; cin>>n;
        int *mas=new int[n];
```

```
        for (i=0; i<n; i++) //ввод массива
            {
                cout<<i+1<<" элемент > ";
                cin>>mas[i];
            }
        InsertSort(mas, n); //вызов функции
        delete[] mas;
    }
```

Код программы на Pascal:

```
type arr=array[1..1000] of integer;
var mas: arr;
i, j, temp, nom, n: integer;

procedure InsertSort(mas: arr; n: integer);     {процедура сортировки вставками}
    begin
        for i:=1 to n-1 do
            begin
                nom:=i+1;
                temp:=mas[nom];
                for j:=i+1 downto 2 do
                    begin
                        if (temp<mas[j-1]) then
                            begin
                                mas[j]:=mas[j-1];
                                nom:=j-1;
                            end;
                    end;
                mas[nom]:=temp;
            end;
        write('Результирующий массив: ');
        for i:=1 to n do write(mas[i], ' '); {вывод массива}
    end;

begin       {основной блок программы}
    write('Количество элементов в массиве > '); read(n);
    for i:=1 to n do {ввод массива}
        begin
            write(i,' элемент > ');
            read(mas[i]);
        end;
    InsertSort(mas, n); {вызов функции}
end.
```

Обе программы сортируют массив по возрастанию. В их основной части выполняются три операции: определение количества элементов в массиве, ввод этих элементов и вызов подпрограммы. Подпрограмма состоит из алгоритма сортировки и

цикла, выводящего результирующую упорядоченную последовательность. Алгоритм включает в себя классическую для многих алгоритмов сортировки структуру вложенных циклов. Количество итераций внешнего цикла не превышает *n*-1, где *n* – число элементов в массиве; внутренний цикл, начиная с шага *i*+1, заканчивает свое выполнение при *j*=0 (значение переменной-счетчика *j* уменьшается с каждым шагом на 1). Переменным *key* и temp на *i*-ом шаге присваиваются значения, зависящие от шага и значения элемента массива *mas* на этом шаге. В переменной *temp* храниться значение элемента массива mas[*i*+1], оно во внутреннем цикле сравнивается со значениями других элементов. *Key* запоминает индекс элемента, который последним был больше *temp*, и ему, по завершению работы внутреннего цикла, присваивается значение переменной *temp*.

6.4 Сортировка Шелла

В 1959 году американский ученый Дональд Шелл опубликовал алгоритм сортировки, который впоследствии получил его имя – «*Сортировка Шелла*». Этот алгоритм может рассматриваться и как обобщение пузырьковой сортировки, так и сортировки вставками.

Идея метода заключается в сравнение разделенных на группы элементов последовательности, находящихся друг от друга на некотором расстоянии. Изначально это расстояние равно d или $n/2$, где n — общее число элементов. На первом шаге каждая группа включает в себя два элемента расположенных друг от друга на расстоянии $n/2$; они сравниваются между собой, и, в случае необходимости, меняются местами. На последующих шагах также происходят проверка и обмен, но расстояние d сокращается на $d/2$, и количество групп, соответственно, уменьшается. Постепенно расстояние между элементами уменьшается, и на $d=1$ проход по массиву происходит в последний раз.

Далее, на примере последовательности целых чисел, показан процесс сортировки массива методом Шелла (рис. 6.4). Для удобства и наглядности, элементы одной группы выделены одинаковым цветом.

Исходный массив	5	3	8	0	7	4	9	1	6	2
d=5	4	3	1	0	2	5	9	8	6	7
d=2	1	0	2	3	4	5	6	7	9	8
d=1	0	1	2	3	4	5	6	7	8	9

Рисунок 6.4 – Пример сортировки Шелла

Первое значение, соответствующее расстоянию d равно 10/2=5. На каждом шаге оно уменьшается вдвое. Элементы, входящие в одну группу, сравниваются и если значение какого-либо элемента, стоящего левее того с которым он сравнивается, оказывается больше (сортировка по возрастанию), тогда они меняются местами. Так, элементы путем внутригрупповых перестановок постепенно становятся на свои позиции, и на последнем шаге ($d=1$) сортировка сводится к проходу по одной группе, включающей в себя все n элементов массива. При этом число требуемых обменов оказывается совсем небольшим.

Код программы на C++:

```
int i, j, n, d, count;
void Shell(int A[], int n) //сортировка Шелла
	{
		d=n;
		d=d/2;
		while (d>0)
			{
				for (i=0; i<n-d; i++)
					{
						j=i;
						while (j>=0 && A[j]>A[j+d])
							{
								count=A[j];
								A[j]=A[j+d];
								A[j+d]=count;
								j--;
							}
					}
				d=d/2;
			}
		for (i=0; i<n; i++) cout<<A[i]<<" "; //вывод массива
	}

void main()	//главная функция
	{
		cout<<"Размер массива > "; cin>>n;
		int *A= new int[n]; //объявление динамического массива
		for (i=0; i<n; i++) //ввод массива
			{
				cout<<i+1<<" элемент > ";
				cin>>A[i];
			}
		cout<<"\nРезультирующий массив: ";
		Shell(A, n);
		delete [] A; //освобождение памяти
	}
```

Код программы на Pascal:

```
type massiv=array[1..100] of integer;
var i, j, n, d, count: integer;
A: massiv;
procedure Shell(A: massiv; n: integer); {сортировка Шелла}
	begin
		d:=n;
		d:=d div 2;
		while (d>0) do
```

```
                begin
                        for i:=1 to n-d do
                                begin
                                        j:=i;
                                        while ((j>0) and (A[j]>A[j+d])) do
                                                begin
                                                        count:=A[j];
                                                        A[j]:=A[j+d];
                                                        A[j+d]:=count;
                                                        j:=j-1;
                                                end;
                                end;
                        d:=d div 2;
                end;
        writeln;
        for i:=1 to n do write(' ', A[i]); {вывод массива}
end;

begin   {основной блок программы}
        write('Размер массива > ');
        read(n);
        for i:=1 to n do {ввод массива}
                begin
                        write(i, ' элемент > ');
                        readln(A[i]);
                end;
        write('Результирующий массив: ');
        Shell(A, n);
end.
```

Сортировка Шелла уступает в эффективности быстрой сортировки, но выигрывает у сортировки вставками. Сравнить скорость работы этих трех алгоритмов можно, воспользовавшись следующей таблицей.

	Сортировка вставками	Сортировка Шелла	Быстрая сортировка
Худший случай	$O(n^2)$	$O(n^2)$	$O(n^2)$
Лучший случай	$O(n)$	$O(n \log n)$	$O(n \log n)$
Средний случай	$O(n^2)$	Зависит от расстояния между элементами	$O(n \log n)$

6.5 Быстрая сортировка

Выбирая алгоритм сортировки для практических целей, программист, вполне вероятно, остановиться на методе, называемом «*Быстрая сортировка*» (также «qsort» от англ. quick sort). Его разработал в 1960 году английский ученый Чарльз Хоар, занимавшийся тогда в МГУ машинным переводом. Алгоритм, по принципу функционирования, входит в класс обменных сортировок (сортировка перемешиванием, пузырьковая сортировка и др.), выделяясь при этом высокой скоростью работы.

Отличительной особенностью быстрой сортировки является операция разбиения массива на две части относительно опорного элемента. Например, если последовательность требуется упорядочить по возрастанию, то в левую часть будут помещены все элементы, значения которых меньше значения опорного элемента, а в правую элементы, чьи значения больше или равны опорному. Вне зависимости от того, какой элемент выбран в качестве опорного, массив будет отсортирован, но все же наиболее удачным считается ситуация, когда по обеим сторонам от опорного элемента оказывается примерно равное количество элементов. Если длина какой-то из получившихся в результате разбиения частей превышает один элемент, то для нее нужно рекурсивно выполнить упорядочивание, т. е. повторно запустить алгоритм на каждом из отрезков.

Таким образом, алгоритм быстрой сортировки включает в себя два основных этапа:

- разбиение массива относительно опорного элемента;
- рекурсивная сортировка каждой части массива.

6.5.1 Разбиение массива

Еще раз об опорном элементе. Его выбор не влияет на результат, и поэтому может пасть на произвольный элемент. Тем не менее, как было замечено выше, наибольшая эффективность алгоритма достигается при выборе опорного элемента, делящего последовательность на равные или примерно равные части. Но, как правило, из-за нехватки информации не представляется возможности наверняка определить такой элемент, поэтому зачастую приходиться выбирать опорный элемент случайным образом.

В следующих пяти пунктах описана общая схема разбиения массива (сортировка по возрастанию):

1. вводятся указатели *first* и *last* для обозначения начального и конечного элементов последовательности, а также опорный элемент *mid*;
2. вычисляется значение опорного элемента (*first*+*last*)/2, и заноситься в переменную *mid*;

3. указатель *first* смещается с шагом в 1 элемент к концу массива до тех пор, пока *Mas*[*first*]>*mid*. А указатель *last* смещается от конца массива к его началу, пока *Mas*[*last*]<*mid*;
4. каждые два найденных элемента меняются местами;
5. пункты 3 и 4 выполняются до тех пор, пока first<last.

После разбиения последовательности следует проверить условие на необходимость дальнейшего продолжения сортировки его частей. Этот этап будет рассмотрен позже, а сейчас на конкретном примере выполним разбиение массива.

Имеется массив целых чисел *Mas*, состоящий из 8 элементов (рис. 6.5): *Mas*[1..8]. Начальным значением *first* будет 1, а *last* – 8. Пройденная часть закрашивается голубым цветом.

Рисунок 6.5 – Исходный массив

В качестве опорного элемента возьмем элемент со значением 5, и индексом 4. Его мы вычислили, используя выражение (*first*+*last*)/2, отбросив дробную часть (рис. 6.6). Теперь *mid*=5.

Рисунок 6.6 – Разбиение массива надвое

Первый элемент левой части сравнивается с *mid*. *Mas*[1]>*mid*, следовательно *first* остается равным 1. Далее, элементы правой части сравниваются с *mid*. Проверяется элемент с индексом 8 и значением 8. *Mas*[8]>*mid*, следовательно *last* смещается на одну позицию влево. *Mas*[7]<*mid*, следовательно *last* остается равным 7. На данный момент *first*=1, а *last*=7. Первый и седьмой элементы меняются местами (рис. 6.7). Оба указателя смещаются на одну позицию каждый в своем направлении.

Рисунок 6.7 – Перестановка 1 и 7 элементов

Алгоритм снова переходит к сравнению элементов. Второй элемент сравнивается с опорным: *Mas*[2]>*mid*, следовательно *first* остается равным 2. Далее, элементы

правой части сравниваются с *mid*. Проверяется элемент с индексом 6 и значением 1: *Mas*[6]<*mid*, следовательно *last* не изменяет своей позиции. На данный момент *first*=2, а *last*=6. Второй и шестой элементы меняются местами (рис. 6.8). Оба указателя смещаются на одну позицию каждый в своем направлении.

Рисунок 6.8 – Перестановка 2 и 6 элементов

Алгоритм снова переходит к сравнению элементов. Третий элемент сравнивается с опорным: *Mas*[3]<*mid*, следовательно *first* смещается на одну позицию вправо. Далее, элементы правой части сравниваются с *mid*. Проверяется элемент с индексом 5 и значением 9: *Mas*[5]>*mid*, следовательно *last* смещается на одну позицию влево. Теперь *first*=*last*=4, а значит, условие *first*<*last* не выполняется, этап разбиения завершается (рис. 6.9).

Рисунок 6.9 – Результат разбиения массива

На этом этап разбиения закончен. Массив разделен на две части относительно опорного элемента. Осталось произвести рекурсивное упорядочивание его частей.

6.5.2 Рекурсивная сортировка частей массива

Если в какой-то из получившихся в результате разбиения массива частей находиться больше одного элемента, то следует произвести рекурсивное упорядочивание этой части, то есть выполнить над ней операцию разбиения, описанную выше. Для проверки условия «количество элементов > 1», нужно действовать примерно по следующей схеме:

Имеется массив *Mas*[*L*..*R*], где *L* и *R* – индексы крайних элементов этого массива. По окончанию разбиения, указатели *first* и *last* оказались примерно в середине последовательности, тем самым образуя два отрезка: левый от *L* до *last* и правый от *first* до *R*. Выполнить рекурсивное упорядочивание левой части нужно в том случае, если выполняется условие *L*<*last*. Для правой части условие аналогично: *first*<*R*.

Код программы на C++:

```
#include <ctime>
const int n=7;
int first, last;
void quicksort(int *mas, int first, int last)        //функция сортировки
    {
        int mid, count;
        int f=first, l=last;
        mid=mas[(f+l) / 2];                          //вычисление опорного элемента
        do
            {
                while (mas[f]<mid) f++;
                while (mas[l]>mid) l--;
                if (f<=l)                            //перестановка элементов
                    {
                        count=mas[f];
                        mas[f]=mas[l];
                        mas[l]=count;
                        f++;
                        l--;
                    }
            } while (f<l);
        if (first<l) quicksort(mas, first, l);
        if (f<last) quicksort(mas, f, last);
    }

void main()                                          //главная функция
    {
        int *A=new int[n];
        srand(time(NULL));
        cout<<"Исходный массив: ";
        for (int i=0; i<n; i++)
            {
                A[i]=rand()%10;
                cout<<A[i]<<" ";
            }
        first=0; last=n-1;
        quicksort(A, first, last);
        cout<<endl<<"Результирующий массив: ";
        for (int i=0; i<n; i++) cout<<A[i]<<" ";
        delete []A;
    }
```

Код программы на Pascal:

```
const n=7;
var A: array[1..n] of integer;
first, last, i: integer;
```

```
procedure quicksort(var mas: array[1..n] of integer; first, last: integer);       {процедура сортировки}
    var f, l, mid, count: integer;
    begin
        f:=first;
        l:=last;
        mid:=mas[(f+l) div 2];          {вычисление опорного элемента}
        repeat
            while mas[f]<mid do inc(f);
            while mas[l]>mid do dec(l);
            if f<=l then                {перестановка элементов}
                begin
                    count:=mas[f];
                    mas[f]:=mas[l];
                    mas[l]:=count;
                    inc(f);
                    dec(l);
                end;
        until f>l;
        if first<l then quicksort(A, first, l);
        if f<last then quicksort(A, f, last);
    end;

begin       {основной блок программы}
    randomize;
    write('Исходный массив: ');
    for i:=1 to n do
        begin
            A[i]:=random(10);
            write(A[i]:2);
        end;
    first:=1; last:=n;
    quicksort(A, first, last);
    writeln; write('Результирующий массив: ');
    for i:=1 to n do write(A[i]:2);
end.
```

Стоит отметить, что быстрая сортировка может оказаться малоэффективной на массивах, состоящих из небольшого числа элементов, поэтому при работе с ними разумнее отказаться от данного метода. В целом алгоритм неустойчив, а также использование рекурсии в неверно составленном коде может привести к переполнению стека. Но, несмотря на эти и некоторые другие минусы, быстрая сортировка все же является одним из самых эффективных и часто используемых методов.

6.6 Сортировка слиянием

Алгоритм *сортировки слиянием* был предложен праотцом современных компьютеров – Джоном фон Нейманом. Сам метод является устойчивым, т. е. он не меняет одинаковые по значению элементы в списке.

В основе сортировки слиянием лежит принцип «разделяй и властвуй». Список разделяется на равные или практически равные части, каждая из которых сортируется отдельно. После чего уже упорядоченные части сливаются воедино. Несколько детально этот процесс можно расписать так:

1. массив рекурсивно разбивается пополам, и каждая из половин делиться до тех пор, пока размер очередного подмассива не станет равным единице;
2. далее, выполняется операция алгоритма, называемая слиянием. Два единичных массива сливаются в общий результирующий массив, при этом из каждого выбирается меньший элемент (сортировка по возрастанию) и записывается в свободную левую ячейку результирующего массива. После чего из двух результирующих массивов собирается третий общий отсортированный массив, и так далее. В случае если один из массивов закончиться, элементы другого дописываются в собираемый массив;
3. в конце операции слияния, элементы перезаписываются из результирующего массива в исходный.

Подпрограмма *MergeSort* рекурсивно разбивает и сортирует массив, а *Merge* отвечает за его слияние. Так можно записать псевдокод основной подпрограммы:

```
Подпрограмма MergeSort(A, first, last)
A – массив
first, last – номера первого и последнего элементов соответственно
Если first<last то
{
Вызов MergeSort(A, first, (first+last)/2)        //сортировка левой части
Вызов MergeSort(A, (first+last)/2+1, last)       //сортировка правой части
Вызов Merge(A, first, last)                      //слияние двух частей
}
```

Эта подпрограмма выполняется только в том случае, если номер первого элемента меньше номера последнего. Как уже говорилось, из подпрограммы *MergeSort* вызывается подпрограмма *Merge*, которая выполняет операцию слияния. Перейдем к рассмотрению последней.

Работа *Merge* заключается в образовании упорядоченного результирующего массива путем слияния двух также отсортированных массивов меньших размеров. Вот псевдокод этой подпрограммы:

```
Подпрограмма Merge(A, first, last)
{
start, final – номера первых элементов левой и правой частей
mas – массив, middle - хранит номер среднего элемента
middle=(first+last)/2                    //вычисление среднего элемента
start=first                                    //начало левой части
final=middle+1                           //начало правой части
Цикл j=first до last выполнять     //выполнять от начала до конца
Если ((start<=middle) и ((final>last) или (A[start]<A[final]))) то
{
mas[j]=A[start]
увеличить start на 1
}
Иначе
{
mas[j]=A[final]
увеличить final на 1
}
Цикл j=first до last выполнять     //возвращение результата в список
A[j]=mas[j]
}
```

Разберем алгоритм сортировки слиянием на следующем примере (рис. 6.10). Имеется неупорядоченная последовательность чисел: 2, 6, 7, 1, 3, 5, 0, 4. После разбивки данной последовательности на единичные массивы, процесс сортирующего слияния (по возрастанию) будет выглядеть так:

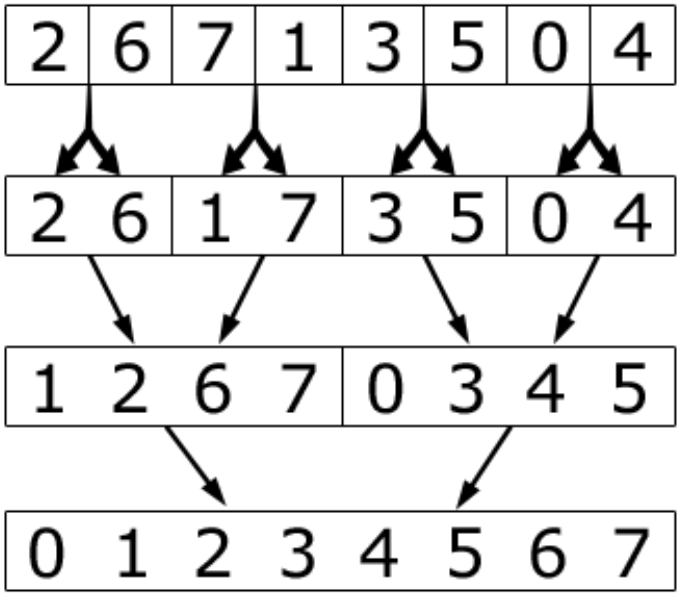


Рисунок 6.10 – Пример сортировки слиянием

Массив был разделен на единичные массивы, которые алгоритм сливает попарно до тех пор, пока не получится один массив, все элементы которого стоят на своих позициях.

Код программы на C++:

```
void Merge(int *A, int first, int last)       //функция, сливающая массивы
    {
        int middle, start, final, j;
        int *mas=new int[100];
        middle=(first+last)/2;          //вычисление среднего элемента
        start=first;                    //начало левой части
        final=middle+1;                 //начало правой части
        for(j=first; j<=last; j++)      //выполнять от начала до конца
            if ((start<=middle) && ((final>last) || (A[start]<A[final])))
                {
                    mas[j]=A[start];
                    start++;
                }
        else
            {
                mas[j]=A[final];
                final++;
            }
        for (j=first; j<=last; j++) A[j]=mas[j];      //возвращение результата в список
        delete[]mas;
    };

void MergeSort(int *A, int first, int last)           //рекурсивная процедура сортировки
    {
        if (first<last)
            {
                MergeSort(A, first, (first+last)/2);        //сортировка левой части
                MergeSort(A, (first+last)/2+1, last);       //сортировка правой части
                Merge(A, first, last);                      //слияние двух частей
            }
    };

void main()                   //главная функция
    {
        int i, n;
        int *A=new int[100];
        cout<<"Размер массива > "; cin>>n;
        for (i=1; i<=n; i++)
            {
                cout<<i<<" элемент > ";
                cin>>A[i];
            }
        MergeSort(A, 1, n);                   //вызов сортирующей процедуры
        cout<<"Упорядоченный массив: ";       //вывод упорядоченного массива
```

```
        for (i=1; i<=n; i++) cout<<A[i]<<" ";
        delete []A;
    }
```

Код программы на Pascal:

```
type massiv=array[1..100] of integer;
var n, i: integer;
A: massiv;
procedure Merge(var A: massiv; first, last: integer);          {процедура, сливающая массивы}
    var middle, start, final , j: integer;
    mas: massiv;
    begin
        middle:=(first+last) div 2;                            {вычисление среднего элемента}
        start:=first;                                          {начало левой части}
        final:=middle+1;                                       {начало правой части}
        for j:=first to last do                                {выполнять от начала до конца}
            if (start<=middle) and ((final>last) or (A[start]<A[final])) then
                begin
                    mas[j]:=A[start];
                    inc(start);
                end
        else
            begin
                mas[j]:=A[final];
                inc(final);
            end;
        for j:=first to last do A[j]:=mas[j];                  {возвращение результата в массив}
    end;

procedure MergeSort(var A: massiv; first, last: integer);     {рекурсивная процедура сортировки}
    begin
        if first<last then
            begin
                MergeSort(A, first, (first+last) div 2);       {сортировка левой части}
                MergeSort(A, (first+last) div 2+1, last);      {сортировка правой части}
                Merge(A, first, last);                         {слияние двух частей}
            end;
    end;

begin       {основной блок программы}
    write('Размер массива > ');
    readln(n);
    for i:=1 to n do
        begin
            write(i, ' элемент > ');
            read(A[i]);
        end;
    MergeSort(A, 1, n);                          {вызов сортирующей процедуры}
```

```
    write('Упорядоченный массив: ');          {вывод отсортированного массива}
    for i:=1 to n do write(A[i], ' ');
end.
```

Недостатком сортировки слиянием является использование дополнительной памяти. Но когда работать приходиться с файлами или списками, доступ к которым осуществляется только последовательно, то очень удобно применять именно этот метод. Также, к достоинствам алгоритма стоит отнести его устойчивость и неплохую скорость работы $O(n*log\ n)$.

6.7 Гномья сортировка

Гномья сортировка впервые была предложена 2 октября 2000 года Хамидом Сарбази-Азадом (Hamid Sarbazi-Azad). Он назвал ее «Глупая сортировка, простейший алгоритм сортировки всего с одним циклом...». И действительно, глупый этот метод или нет, но в нем задействован, никак в большинстве сортировок – два или более циклов, а только один. Позже, голландский ученый Дик Грун, на страницах одной из своих книг, привел для гномьей сортировки следующую аналогию.

«Gnome Sort is based on the technique used by the standard Dutch Garden Gnome (Du.: tuinkabouter). Here is how a garden gnome sorts a line of flower pots. Basically, he looks at the flower pot next to him and the previous one; if they are in the right order he steps one pot forward, otherwise he swaps them and steps one pot backwards. Boundary conditions: if there is no previous pot, he steps forwards; if there is no pot next to him, he is done».

Перевод:

«Гномья сортировка основана на технике, используемой обыкновенным голландским садовым гномом. Вот как садовый гном сортирует ряд цветочных горшков. По существу, он смотрит на два соседних цветочных горшка, если они находятся в правильном порядке, то он переходит на один горшок вперед, иначе он меняет их местами и возвращается на один горшок назад. Граничные условия: если позади нет ни одного горшка – он шагает вперед, а если нет следующего горшка, тогда он закончил».

Так описал основную идею алгоритма гномьей сортировки Дик Грун. Заменим гнома и горшки на более формальные объекты, например на указатель (номер текущего элемента) и элементы массива, и рассмотрим принцип работы алгоритма еще раз. Вначале указатель ставиться на 2-ый элемент массива (в C++ он имеет номер 1, а в Паскале номер 2). Затем происходит сравнение двух соседних элементов $A[i]$ и $A[i-1]$, т. е. сравниваются первый элемент (i-1) и второй (i). Если сравниваемые элементы стоят на своих позициях, то сдвигаем указатель на позицию i+1 и продолжаем сравнение с нового положения; иначе меняем элементы местами, и, поскольку в какой-то момент может оказаться, что элементы в левом подмассиве стоят не на своих местах, сдвигаем указатель на одну позицию влево, осуществляя следующее сравнение с новой позиции. Так алгоритм продолжает выполняться до тех пор, пока весь массив не будет упорядочен. Здесь следует выделить два важных момента. Во-первых, процесс упорядочивания заканчивается, тогда когда не выполняется условие $i<n$, где n – размер массива. Во-вторых, как было сказано, указатель перемещается как вперед по списку,

так и назад, и в том случае если он окажется над первым элементом, его сразу же следует сместить на одну позицию вправо, т. к. в противном случае придется сравнивать два элемента, одного из которых не существует.

Перейдем собственно к коду. На своем сайте Дик Грун выложил следующий листинг:

```
void gnomesort(int n, int ar[])
    {
        int i = 0;
        while (i < n)
            {
                if (i == 0 || ar[i-1] <= ar[i]) i++;
                else
                    {
                        int tmp = ar[i];
                        ar[i]=ar[i-1];
                        ar[--i]=tmp;
                    }
            }
    }
```

Приведенный Груном код позволяет упорядочить массив, но все же он может быть немного улучшен. Оптимизация потребует небольшого редактирования, в частности, добавления одной вспомогательной переменной, что позволит избавиться от излишних сравнений. Далее показаны листинги программ, сортирующих элементы массива по возрастанию.

Код программы на C++:

```
int n;
void Gnome_sort(int i, int j, int *mas)    //Гномья сортировка
    {
        while (i<n)
            {
                if (mas[i-1]<=mas[i]) { i=j; j++; }
                else
                    {
                        int t=mas[i];
                        mas[i]=mas[i-1]; mas[i-1]=t;
                        i--;
                        if (i==0)
                            {
                                i=j;
                                j++;
                            }
                    }
            }
        for (i=0; i<n; i++) cout<<mas[i]<<" ";
```

```
    }

void main()       //главная функция
    {
        int m, k;
        cout<<"Размер массива > "; cin>>n;
        int *a=new int[n];
        for (k=0; k<n; k++)
            {
                cout<<k+1<<" элемент > ";
                cin>>a[k];
            }
        k=1; m=2;
        Gnome_sort(k, m, a);
        delete []a;
    }
```

Код программы Pascal:

```
type massiv=array[1..100] of integer;
var k, m, n: integer;
a: massiv;
procedure PGnome_sort(i, j: integer; mas: massiv); {Гномья сортировка}
    var t: integer;
    begin
        while i<=n do
            begin
                if mas[i-1]<=mas[i] then
                    begin
                        i:=j;
                        j:=j+1;
                    end
                else
                    begin
                        t:=mas[i];
                        mas[i]:=mas[i-1];
                        mas[i-1]:=t;
                        i:=i-1;
                        if i=1 then
                            begin
                                i:=j;
                                j:=j+1;
                            end;
                    end;
            end;
        for i:=1 to n do write(mas[i], ' ');
    end;

begin       {основной блок программы}
```

```
    write('Размер массива > '); read(n);
    for k:=1 to n do
        begin
            write(k, ' элемент > ');
            read(a[k]);
        end;
    k:=2;
    m:=3;
    PGnome_sort(k, m, a);
end.
```

Кажется немного необычным, тот факт, что алгоритм сортировки использует всего один цикл. Плюс к этому, на практике гномья сортировка не уступает в скорости сортировки вставками, что видно из таблицы трех случаев этих сортировок.

	Гномья сортировка	Сортировка вставками
Худший случай	$O(n^2)$	$O(n^2)$
Лучший случай	$O(n)$	$O(n)$
Средний случай	$O(n^2)$	$O(n^2)$

6.8 Шейкерная сортировка

Рассматриваемый в данной статье алгоритм имеет несколько непохожих друг на друга названий. Среди них: *сортировка перемешиванием*, *двунаправленная пузырьковая сортировка*, *шейкерная сортировка*, *пульсирующая сортировка* (ripple sort), *трансфертная сортировка* (shuttle sort), и даже *сортировка* «*счастливый час*» (happy hour sort). Второй вариант (двунаправленная пузырьковая сортировка) наиболее точно описывает процесс работы алгоритма. Здесь, в его название, довольно-таки удачно включен термин «пузырьковая». Это действительно альтернативная версия известного метода, модификации в котором заключаются, по большей части, в реализации, упомянутой в названии, двунаправленности: алгоритм перемещается, ни как в обменной (пузырьковой) сортировке – строго снизу вверх (слева направо), а сначала снизу вверх, потом сверху вниз.

Перестановка элементов в шейкерной сортировке выполняется аналогично той же в пузырьковой сортировке, т. е. два соседних элемента, при необходимости, меняются местами. Пусть массив требуется упорядочить по возрастанию. Обозначим каждый пройденный путь от начала до конца последовательности через W_i, где i – номер пути; а обратный путь (от конца к началу) через $-W_j$, где j – номер пути. Тогда после выполнения W_i, один из неустановленных элементов будет помещен в позицию справа, как наибольший из еще неотсортированных элементов, а после выполнения $-W_j$, наименьший из неотсортированных, переместиться в некоторую позицию слева. Так, например, после выполнения W_1 в конце массива окажется элемент, имеющий наибольшее значение, а после $-W_1$ в начало отправиться элемент с наименьшим значением.

Код программы на C++:

```
void Swap(int *Mas, int i)                    //функция обмена
    {
        int temp;
        temp=Mas[i];
        Mas[i]=Mas[i-1];
        Mas[i-1]=temp;
    }

void ShakerSort(int *Mas, int Start, int n)   //шейкерная сортировка
    {
        int Left, Right, i;
        Left=Start;
        Right=n-1;
        while (Left<=Right)
            {
                for (i=Right; i>=Left; i--)
                    if (Mas[i-1]>Mas[i])
                        Swap(Mas, i);
                Left++;
```

```
                for (i=Left; i<=Right; i++)
                        if (Mas[i-1]>Mas[i])
                                Swap(Mas, i);
                        Right--;
                }
        }

void main()     //главная функция
        {
                int n, k;
                cout<<"Размер массива > "; cin>>n;
                int *A=new int[n];
                for (k=0; k<n; k++)
                        {
                                cout<<k+1<<" элемент > ";
                                cin>>A[k];
                        }
                ShakerSort(A, 1, n);
                cout<<"Результирующий массив: ";
                for (k=0; k<n; k++)cout<<" "<<A[k];
        }
```

Код программы на Pascal:

```
type Massiv=array[1..100] of integer;
var
n, k: integer;
A: Massiv;

procedure ShakerSort(Mas: Massiv; Start, m: integer);      {шейкерная сортировка}
        var Left, Right, temp, i: integer;
        begin
                Left:=Start;
                Right:=m;
                while Left<=Right do
                        begin
                                for i:=Right downto Left do
                                        if (Mas[i-1]>Mas[i]) then
                                                begin
                                                        temp:=Mas[i];
                                                        Mas[i]:=Mas[i-1];
                                                        Mas[i-1]:=temp;
                                                end;
                                Left:=Left+1;
                                for i:=Left to Right do
                                        if Mas[i-1]>Mas[i] then
                                                begin
                                                        temp:=Mas[i];
                                                        Mas[i]:=Mas[i-1];
```

```
                                        Mas[i-1]:=temp;
                                    end;
                                Right:=Right-1;
                            end;
                    for i:=1 to m do write(' ', Mas[i]);
            end;

    begin       {основной блок программы}
            write('Размер массива > '); read(n);
            for k:=1 to n do
                    begin
                            write(k, ' элемент  > ');
                            read(A[k]);
                    end;
            write('Результирующий массив: ');
            ShakerSort(A, 2, n);
    end.
```

Следующая таблица отражает временную сложность алгоритма шейкерной сортировки для трех случаев.

Лучший случай	Средний случай	Наихудший случай
$O(n)$	$O(n^2)$	$O(n^2)$

Приведенные временные показатели не позволяют рекомендовать алгоритм для использования его в практических целях. В обучении, ввиду своей относительной сложности, метод, несомненно, будет полезен.

Глава 7. Поиск

7.1 Линейный поиск

Для нахождения некоторого элемента (ключа) в неупорядоченном массиве используется алгоритм *линейного* (*последовательного*) *поиска*. Он работает как с неотсортированными массивами, так и отсортированными, но для вторых существуют алгоритмы эффективнее линейного поиска. Эту неэффективность компенсируют простая реализация алгоритма и сама возможность применять его к неупорядоченным последовательностям. Здесь, а также при рассмотрении всех других алгоритмов поиска, будем полагать, что в качестве ключа выступает некоторая величина, по мере выполнения алгоритма, сравниваемая именно со значениями элементов массива.

Слово «последовательный» содержит в себе основную идею метода. Начиная с первого, все элементы массива последовательно просматриваются и сравниваются с искомым. Если на каком-то шаге текущий элемент окажется равным искомому, тогда элемент считается найденным, и в качестве результата возвращается номер этого элемента, либо другая информация о нем. (Далее, в качестве выходных данных будет выступать номер элемента). Иначе, следуют возвратить что-то, что может оповестить о его отсутствии в пройденной последовательности.

Примечателен тот факт, что имеется вероятность наличия нескольких элементов с одинаковыми значениями, совпадающими со значением ключа. В таком случае, если нет конкретных условий, можно, например, за результирующий взять номер первого найденного элемента (реализовано в листинге ниже), или записать в массив номера всех тождественных элементов.

Код программы на C++:

```
#include <ctime>
int i, n;
int LineSearch(int A[], int key)       //линейный поиск
    {
        for (i=0; i<n; i++)
            if (A[i]==key) return i;
        return -1;
    }

void main()       //главная функция
    {
        int key,  A[1000];
        srand(time(NULL));
        cout<<"Размер массива > "; cin>>n;
        cout<<"Искомый элемент > "; cin>>key;
        cout<<"Исходный массив: ";
        for (i=0; i<n; i++)
            {
```

```
                A[i]=rand()%10;
                cout<<A[i]<<" ";
            }
        if (LineSearch(A, key)==-1) cout<<"\nЭлемент не найден";
        else cout<<"\nНомер элемента: "<<LineSearch(A, key)+1;
    }
```

Код программы на Pascal:

```
type Arr=array[1..1000] of integer;
var key, i, n: integer;
A: Arr;

function lineSearch(A: Arr; key: integer): integer;     {линейный поиск}
    begin
        lineSearch:=-1;
        for i:=1 to n do
        if A[i]=key then
            begin
                lineSearch:=i;
                exit;
            end;
    end;

begin       {основной блок программы}
    randomize;
    write('Размер массива > '); readln(n);
    write('Искомый элемент > '); read(key);
    write('Исходный массив: ');
    for i:=1 to n do
        begin
            A[i]:=random(10);
            write(A[i],' ');
        end;
    writeln;
    if (lineSearch(A, key)=-1) then write('Элемент не найден')
    else write('Номер элемента: ', lineSearch(A, key));
end.
```

В лучшем случае, когда искомый элемент занимает первую позицию, алгоритм произведет всего одно сравнение, а в худшем *n*, где *n* — количество элементов в массиве. Худшему случаю соответствуют две ситуации: искомый элемент занимает последнюю позицию, или он вовсе отсутствует в массиве.

Алгоритм линейного поиска не часто используется на практике, поскольку принцип работы «в лоб» делает скорость решения им задачи очень низкой. Его применение оправдано на небольших и/или неотсортированных последовательностях, но когда последовательность состоит из большого числа элементов, и с ней предстоит

работать не раз, тогда наиболее оптимальным решением оказывается предварительная сортировка этой последовательности с последующим применением двоичного или другого, отличного от линейного, алгоритма поиска.

7.2 Двоичный поиск

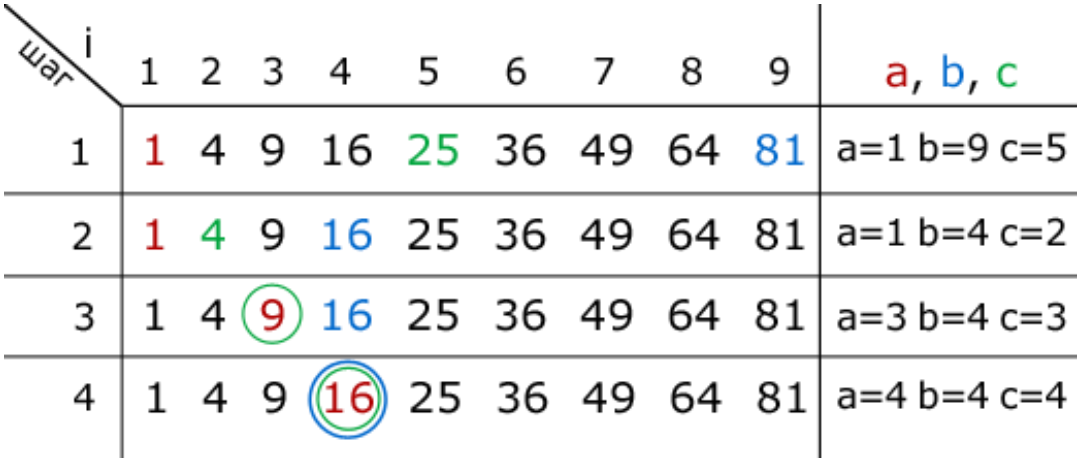
Когда поиск некоторого элемента необходимо осуществить в упорядоченной по возрастанию или убыванию последовательности, тогда применѝм алгоритм *двоичного* (*бинарного*) *поиска*. Метод использует стратегию «разделяй и властвуй», а именно: заданная последовательность делится на две равные части и поиск осуществляется в одной из этих частей, которая потом также делится надвое, и так до тех пор, пока обнаружится наличие искомого элемента или его отсутствие. Использовать эту операцию, уменьшая каждый раз зону поиска вдвое, позволительно лишь исходя из того факта, что элементы последовательности заранее упорядочены. Найдя средний элемент (сделать это, зная число элементов массива, не составит труда), и сравнив его значение с искомым, можно уверено сказать, где относительно среднего элемента находится искомый элемент.

Далее, будем полагать, что элементы массива располагаются в порядке возрастания, поскольку нет существенной разницы, как именно они упорядочены: по возрастанию или убыванию значение. Также обозначим границы зоны поиска через *left* и *right* – крайний левый и крайний правый элементы соответственно.

Ход работы алгоритма, разделенный на этапы, выглядит следующим образом:

1. зона поиска (на первом шаге ей является весь массив) делиться на две равные части, путем определения ее среднего (*mid*) элемента;
2. средний элемент сравнивается с искомым (*key*), результатом этого сравнения будет один из трех случаев:
    - *key*<*mid*. Крайней правой границей области поиска становится элемент, стоящий перед средним (*right* ← *mid*-1);
    - *key*>*mid*. Крайней левой границей области поиска становится следующий за средним элемент (*left* ← *mid*+1);
    - *key*=*mid*. Значения среднего и искомого элементов совпадают, следовательно элемент найден, работа алгоритма завершается.
3. если для проверки не осталось ни одного элемента, то алгоритм завершается, иначе выполняется переход к пункту 1.

В таблице (рис. 7.1) представлен конкретный целочисленный массив, и пошаговое выполнение алгоритма бинарного поиска применительно к его элементам. Для экономии места в таблице *left*, *right* и *mid* заменены на *a*, *b* и *c*.



шаг \ i	1	2	3	4	5	6	7	8	9	a, b, c
1	1	4	9	16	25	36	49	64	81	a=1 b=9 c=5
2	1	4	9	16	25	36	49	64	81	a=1 b=4 c=2
3	1	4	9	16	25	36	49	64	81	a=3 b=4 c=3
4	1	4	9	16	25	36	49	64	81	a=4 b=4 c=4

Рисунок 7.1 – Пример двоичного поиска

Имеется последовательность целых чисел, расположенных в порядке возрастания, а также ключ – число 16. Изначально граничными элементами являются элементы с номерами 1 и 9, и значениями 1 и 81. Вычисляется номер среднего элемента, для чего, как правило, используется выражение (*right*+*left*)/2, либо *left*+(*right*-*left*)/2 (далее, в программах будет использована второе выражение, как наиболее устойчивая к переполнениям). После сравнения оказывается, что искомый элемент меньше среднего, и поэтому последующий поиск осуществляется в левой части последовательности. Алгоритм продолжает выполняться подобным образом, до нахождения на 4 шаге искомого элемента.

Стоит отметить, что здесь потребуется гораздо меньше времени, чем, если бы мы воспользовались линейным поиском, но в отличие от линейного поиска двоичный работает только с массивами, элементы которых упорядочены, что, несомненно, придает ему отрицательной специфичности.

Код программы на C++:

```
const int n=10;
int BinarySearch(int A[], int key)    //двоичный поиск
    {
        int left=0, right=n, mid;
        while (left<=right)
            {
                mid=left+(right-left)/2; //вычисление среднего элемента
                if (key<A[mid]) right=mid-1;
                else if (key>A[mid]) left=mid+1;
                else return mid;
            }
        return -1;
    }

void main()       //главная функция
```

```
{
	int i, key;
	int A[n];
	cout<<"Искомый элемент > ";
	cin>>key; //ввод ключа
	cout<<"Исходный массив: ";
	for (i=0; i<n; i++) //заполнение и вывод массива
		{
			A[i]=n*i;
			cout<<A[i]<<" ";
		}
	if (BinarySearch(A, key)==-1) cout<<"\nЭлемент не найден";
	else cout<<"\nНомер элемента: "<<BinarySearch(A, key)+1;
}
```

Код программы на Pascal:

```
const n=10;
type Arr=array[1..n] of integer;
var mid, left, right, key, i: integer;
A: Arr;
function BinarySearch(A: Arr; key: integer): integer;{двоичный поиск}
	begin
		left:=1; right:=n;
		while left<=right do
			begin
				mid:=left+(right-left) div 2;	{вычисление среднего элемента}
				if (key<A[mid]) then right:=mid-1
				else if (key>A[mid]) then left:=mid+1
				else begin BinarySearch:=mid; exit; end;
			end;
		BinarySearch:=-1;
	end;

begin	{основной блок программы}
	write('Искомый элемент > '); read(key); {ввод ключа}
	write('Исходный массив: ');
	for i:=1 to n do {заполнение и вывод массива}
		begin
			A[i]:=n*i;
			write(A[i], ' ');
		end;
	writeln;
	if (BinarySearch(A, key)=-1) then write('Элемент не найден')
	else write('Номер элемента: ', BinarySearch(A, key));
end.
```

В программе можно было бы реализовать проверку массива на наличие в нем элементов, но так как он заполняется, независимо от пользователя, строго определенным количеством константных значений, это делать необязательно.

В случае, когда первое значение *mid* совпадает с ключом, тогда считается, что алгоритм выполнился за свое лучшее время $O(1)$. В среднем и худшем случае время работы алгоритма составляет $O(log\ n)$, что значительно быстрее, чем у линейного поиска, требующего линейного времени.

7.3 Интерполяционный поиск

В основе интерполяционного поиска лежит операция интерполирование. Интерполирование – нахождение промежуточных значений величины по имеющемуся дискретному набору известных значений. Интерполяционный поиск работает только с упорядоченными массивами; он похож на бинарный, в том смысле, что на каждом шаге вычисляется некоторая область поиска, которая, по мере выполнения алгоритма, сужается. Но в отличие от двоичного, интерполяционный поиск не делит последовательность на две равные части, а вычисляет приблизительное расположение ключа (искомого элемента), ориентируясь на расстояние между искомым и текущим значением элемента. Идея алгоритма напоминает хорошо знакомый старшим поколениям поиск телефонного номера в обычном справочнике: список имен абонентов упорядочен, поэтому не составит труда найти нужный телефонный номер, так как, если мы, например, ищем абонента с фамилией, начинающейся на букву «Ю», то для дальнейшего поиска разумно будет перейти в конец справочника.

Формула, определяющая алгоритм интерполяционного поиска выглядит следующим образом:

$$mid = left + \frac{(key - A[left]) * (right - left)}{A[right] - A[left]}$$

Здесь *mid* – номер элемента, с которым сравнивается значение ключа, *key* – ключ (искомый элемент), *A* – массив упорядоченных элементов, *left* и *right* – номера крайних элементов области поиска. Важно отметить, операция деления в выражении строго целочисленная, т. е. дробная часть, какая бы она ни была, отбрасывается.

Код программы на C++:

```
const int n=17;
int InterpolSearch(int A[], int key) //интерполяционный поиск
    {
        int mid, left=0, right=n-1;
        while (A[left]<=key && A[right]>=key)
            {
                mid=left+((key-A[left])*(right-left))/(A[right]-A[left]);
                if (A[mid]<key) left=mid+1;
                else if (A[mid]>key) right=mid-1;
                else return mid;
            }
        if (A[left]==key) return left;
        else return -1;
    }
```

```
void main()       //главная функция
    {
        int i, key;
        int A[n]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59};
        cout<<"Искомый элемент > "; cin>>key; //ввод ключа
        cout<<"Исходный массив: ";
        for (i=0; i<n; i++) cout<<A[i]<<" ";  //вывод массива
        if (InterpolSearch(A, key)==-1) cout<<"\nЭлемент не найден";
        else cout<<"\nНомер элемента: "<<InterpolSearch(A, key)+1;
    }
```

Код программы на Pascal:

```
const n=17;
type Arr=array[1..n] of integer;
var mid, left, right, key, i: integer;
const A: Arr=(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59);
function InterpolSearch(A: Arr; key: integer): integer;       {интерполяционный поиск}
    begin
        left:=1; right:=n;
        while ((A[left]<=key) and (A[right]>=key)) do
            begin
                mid:=left+((key-A[left])*(right-left)) div (A[right]-A[left]);
                if (A[mid]<key) then left:=mid+1
                else if (A[mid]>key) then right:=mid-1
                else
                    begin
                        InterpolSearch:=mid;
                        exit;
                    end;
            end;
        if (A[left]=key) then InterpolSearch:=left
        else InterpolSearch:=-1;
    end;

begin       {основной блок программы}
    write('Искомый элемент > '); read(key); {ввод ключа}
    write('Исходный массив: ');
    for i:=1 to n do write(A[i], ' '); {вывод массива}
    writeln;
    if (InterpolSearch(A, key)=-1) then write('Элемент не найден')
    else write('Номер элемента: ', InterpolSearch(A, key));
end.
```

Интерполяционный поиск в эффективности, как правило, превосходит двоичный, в среднем требуя *log*log n* операций, где *n* – объем входных данных. Тем самым время его работы составляет *O*(*log*log n*). Но если, к примеру, последовательность

экспоненциально возрастает, то скорость увеличиться до $O(n)$. Наибольшую производительность алгоритм показывает на последовательности, элементы которой распределены равномерно относительно друг друга.

Глава 8. Теория чисел

8.1 Алгоритм Евклида

Когда говорят «число делиться», то имеют в виду, что оно делиться без остатка. Так *A* делиться на *B*, лишь в том случае, если остаток от их деления равен нулю. На этом свойстве основывается понятие *наибольшего общего делителя* (НОД). НОД двух чисел – это наибольший из всех их общих делителей.

Одним из простейших алгоритмов нахождения наибольшего общего делителя является *Алгоритм Евклида*. Он назван в честь известного древнегреческого математика, автора первого из дошедших до нас теоретических трактатов по математике – Евклида Александрийского. Выделяют два способа реализации алгоритма: методом *деления* и методом *вычитания*. Рассмотрим отдельно каждый из них.

8.1.1 Метод вычитания

Найти НОД двух целых чисел немного проще используя операцию вычитания. Для этого потребуется следовать такому условию: если *A=B*, то НОД найден и он равен одному из чисел, иначе необходимо большее из двух чисел заменить разностью его и меньшего. В наиболее понятной форме данный метод описывается блок-схемой (рис. 8.1).

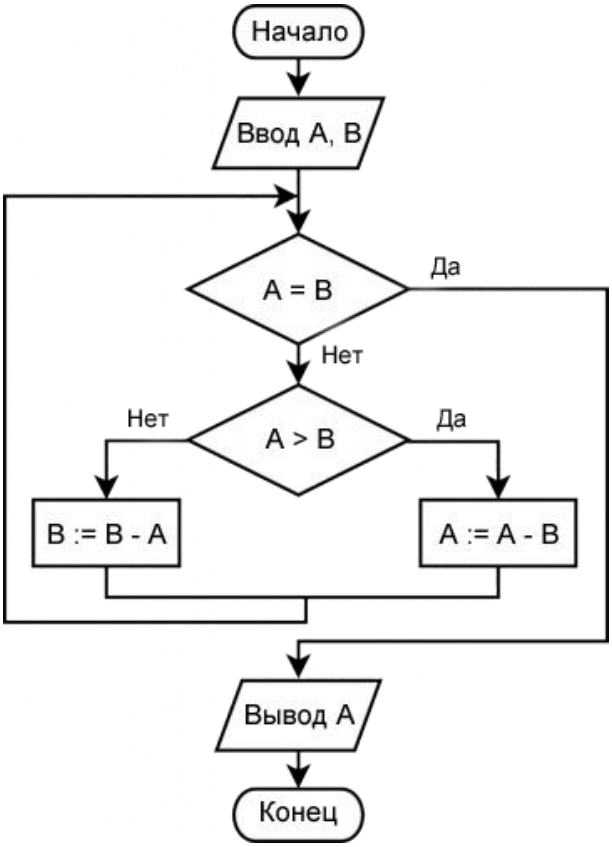


Рисунок 8.1 – Блок-схема алгоритма Евклида. Метод вычитания

Оперируя данной блок-схемой – составляя по ней программный код, вполне целесообразно включить в него оператор цикла с вложенным условным оператором ветвления, имеющим две ветви.

Код программы на C++ (вычитание):

```
int NOD(int A, int B)    //алгоритм Евклида. Вычитание
    {
        while (A!=B)
            if (A>B) A-=B;
            else B-=A;
        return A;
    }

void main ()      //главная функция
    {
        int A, B;
        cout<<"A > "; cin>>A;
        cout<<"B > "; cin>>B;
        cout<<"НОД("<<A<<", "<<B<<")="<<NOD(A, B);
    }
```

Код программы на Pascal (вычитание):

```
var A, B: integer;
function NOD(A, B: integer): integer;     {алгоритм Евклида. Вычитание}
    begin
        while A<>B do
            if A>B then A:=A-B
            else B:=B-A;
        NOD:=A;
    end;

begin      {основной блок программы}
    write('A > '); read(A);
    write('B > '); read(B);
    write('НОД(', A, ', ', B, ')=', NOD(A, B));
end.
```

8.1.2 Метод деления

Второй способ отличается от первого тем, что в основной части программы операция вычитания заменяется на операцию деления, а точнее на взятие остатка от деления большого числа на меньшее. Этот способ предпочтительнее предыдущего, так как он в большинстве случаев эффективнее и требует меньше времени.

За исключением условия выхода из цикла и операций в выражениях, блок-схема метода деления описывается аналогично методу вычитания (рис. 8.2).

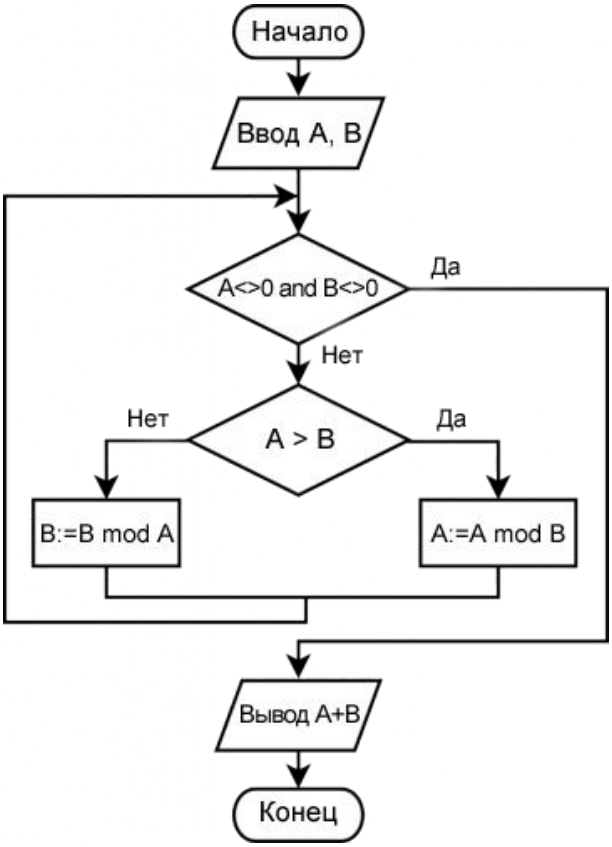


Рисунок 8.2 – Блок-схема алгоритма Евклида. Метод деления

Достаточно то условие, при котором тело цикла будет выполняться до тех пор, пока обе переменных имеют значения отличные от нуля, поскольку, когда условие перестанет быть истинным, то из этого последует, что одно из теперешних значений является искомым наибольшим общим делителем. Да и потом, никак нельзя допустить следующей итерации, в которой будет предпринята попытка деления на нуль.

На конкретных примерах продемонстрируем работу каждого из видов реализации алгоритма. Начнем с того, в основе которого лежит операция взятия остатка от деления. Имеем два числа: 112 и 32. Первое больше второго – заменим его остатком от деления 112 на 32. Новая пара чисел включает 16 и 32. Второе больше, поэтому также заменим его остатком от деления 32 на 16, т. е. нулем. В результате получаем НОД=16:

Начальные данные	112	32
Шаг 1	16	32
Шаг 2	16	0

А теперь составим с теми же числами таблицу для алгоритма вычитанием:

Начальные данные	112	32
Шаг 1	80	32
Шаг 2	48	32
Шаг 3	16	32
Шаг 4	16	0

Приведенный пример продемонстрировал, как в частном случае, предпочтя деление (взятие остатка от деления) вычитанию, можно выиграть в быстродействии. Преимущество деления становится видно наиболее отчетливо после следующих рассуждений. Предположим, что *A* меньше *B*, а так как НОД двух целых чисел меньше или равен наименьшему из них, то и тут он меньше или равен *A*; поэтому оптимальным будет уже при первой операции заменить *B* числом меньшим или равным *A*. Далее, известно, что в одном случае большее число заменяется разностью его и меньшего числа, а в другом остатком от деления. При делении *B* на *A* (большее на меньшее), остаток не может превышать число, стоящее в знаменателе (т. е. *A*), следовательно, взятие остатка от деления гарантирует оптимальный исход. Но то же самое нельзя сказать в отношении операции вычитания, поскольку совсем необязательно, что сразу после выполнения первого вычитания, *B* станет меньше или равно *A*. К примеру, пусть *A* будет равняться 150, а *B* – 1100. Так, используя вычитание, мы в первом действии получим *B* равное 950, в то время как метод деления даст 50.

Код программы на C++ (деление):

```
int NOD(int A, int B)    //алгоритм Евклида. Деление
    {
        while (A!=0 && B!=0)
            if (A>B) A%=B;
            else B%=A;
        return A+B;
    }

void main ()      //главная функция
    {
        int A, B;
        cout<<"A > "; cin>>A;
        cout<<"B > "; cin>>B;
        cout<<"НОД("<<A<<", "<<B<<")="<<NOD(A, B);
    }
```

Код программы на Pascal (деление):

```
var A, B: integer;
function NOD(a, B: integer): integer;      {алгоритм Евклида. Деление}
      begin
            while (A<>0) and (B<>0) do
                  if A>B then A:=A mod B
                  else B:=B mod A;
            NOD:=A+B
      end;

begin       {основной блок программы}
      write('A > '); read(A);
      write('B > '); read(B);
      write('НОД(', A, ', ', B, ')=', NOD(A, B));
end.
```

8.2 Бинарный алгоритм вычисления НОД

Бинарный алгоритм вычисления НОД, как понятно из названия, находит наибольший общий делитель, а именно НОД двух целых чисел. В эффективности данный алгоритм превосходит метод Евклида, что связано с использованием сдвигов, то есть операций деления на степень 2-ки, в нашем случае на 2. Компьютеру проще поделить (умножить) на 2, 4, 8 и т.д., чем на какое-либо другое число. Но в тоже время бинарный алгоритм уступает алгоритму Евклида в простоте реализации. Для дальнейшего усвоения материала следует ознакомиться со свойствами, которыми обладает НОД двух чисел *A* и *B*. Потребуются не все свойства, а только три следующих тождества:

1. НОД(2*A*, 2*B*) = 2НОД(*A*, *B*)
2. НОД(2*A*, 2*B*+1) = НОД(*A*, 2*B*+1)
3. НОД(-*A*, *B*) = НОД(*A*, *B*)

Теперь рассмотрим этапы работы алгоритма. Они основываются на приведенных свойствах наибольшего общего делителя.

1. инициализируем переменную *k* значением 1. Ее задача – подсчитывать «несоразмерность», полученную в результате деления. В то время как *A* и *B* сокращаются вдвое, она будет увеличиваться вдвое;
2. пока *A* и *B* одновременно не равны нулю, выполняем
   - если *A* и *B* – четные числа, то делим надвое каждое из них: $A \leftarrow A/2$, $B \leftarrow B/2$, а *k* увеличивать вдвое: $k \leftarrow k*2$, до тех пор, пока хотя бы одно из чисел *A* или *B* не станет нечетным;
   - если *A* – четное, а *B* – нечетное, то делим *A*, пока возможно деление без остатка;
   - если *B* – четное, а *A* – нечетное, то делим *B*, пока возможно деление без остатка;
   - если $A \geq B$, то заменим *A* разностью *A* и *B*, иначе *B* заменим разностью *B* и *A*.
3. после выхода из 2-ого пункта, остается вернуть в качестве результата произведение *B* и *k*: НОД(*A*, *B*)=*B***k*.

Код программы на C++:

```
int NOD(int A, int B)    //бинарный алгоритм вычисления НОД
    {
        int k=1;
        while ((A!=0) && (B!=0))
            {
                while ((A%2==0) && (B%2==0))
                    {
                        A/=2;
                        B/=2;
                        k*=2;
```

```
                }
            while (A%2==0) A/=2;
            while (B%2==0) B/=2;
            if (A>=B) A-=B;
            else B-=A;
        }
    return B*k;
}

void main()     //главная функция
{
    int A, B;
    cout<<"A > "; cin>>A;
    cout<<"B > "; cin>>B;
    cout<<"НОД("<<A<<", "<<B<<")="<<NOD(A, B);
}
```

Код программы на Pascal:

```
var A, B: integer;
function NOD(A, B: integer): integer;     {бинарный алгоритм вычисления НОД}
    var k: integer;
    begin
        k:=1;
        while (A<>0) and (B<>0) do
            begin
                while (A mod 2=0) and (B mod 2=0) do
                    begin
                        A:=A div 2;
                        B:=B div 2;
                        k:=k*2;
                    end;
                while A mod 2=0 do A:=A div 2;
                while B mod 2=0 do B:=B div 2;
                if A>=B then A:=A-B
                else B:=B-A;
            end;
        NOD:=B*k;
    end;

begin       {основной блок программы}
    write('A > '); read(A);
    write('B > '); read(B);
    write('НОД(', A, ', ', B, '): ', NOD(A, B));
end.
```

Интересен тот факт, что алгоритм был известен еще в Китае 1-го века н. э., но годом его обнародования оказался лишь 1967, когда израильский программист и физик

Джозеф Стейн опубликовал алгоритм. Ввиду этого встречается альтернативное название метода – алгоритм Стейна.

8.3 Быстрое возведение в степень

Для возведения числа *x* в степень *n*, как правило, используют стандартный метод, т. е. число *x* умножают *n* раз само на себя. В задачах математического толка, решаемых при помощи бумаги и ручки, такой метод вполне приемлем, ведь степенная функция быстро растет и поэтому сомнительно, что придется производить затруднительные операции вручную.

Другое дело программирование, где важно не только решить поставленную задачу, но и составить оптимальное решение, удовлетворяющее предусмотренному диапазону входных данных. Так, в частности, для операции возведения числа в степень имеется алгоритм, позволяющий значительно сократить число требуемых операций. Он достаточно прост и основывается на математических свойствах степеней.

Пусть имеется некоторая степень x^n, где x – действительное число, а n – натуральное. Тогда для x^n справедливо равенство:

$$x^n = (x^m)^k$$

При этом $m*k=n$. Например: $3^6=(3^3)^2$, $5^7=(5^7)^1$. Это свойство является одним из основных степенных свойств, и именно на нем основывается рассматриваемый метод. Далее, заметим, что в случае, если n является четным числом, то верно следующее равенство:

$$x^n = (x^{n/2})^2 = x^{n/2} * x^{n/2}$$

Так, если x=3, а n=6, то имеем $3^6 = (3^{6/2})^2 = 3^{6/2} * 3^{6/2}$. Используя это свойство, удастся существенно уменьшить число операций необходимых для возведения x в степень *n*. Теперь адаптируем формулу для случая с нечетным *n*. Для этого понадобиться просто перейти к степени на единицу меньшей. Например: $5^7 = 5^6*5$, $12^5 = 12^4*12$. Общая форма равенства перехода:

$$x^n = x^{n-1}*x$$

В программе, реализующей алгоритм быстрого возведения в степень, используются указанные свойства: если степень *n* четная, то переходим к степени вдвое меньшей, иначе заменяем нечетную степень четной.

Код программы на C++:

```
float bpow(float x, int n)      //быстрое возведение в степень
    {
        float count=1;
        if (!n) return 1;
        while (n)
            {
                if (n%2==0)
                    {
                        n/=2;
                        x*=x;
```

```
                }
            else
                {
                    n--;
                    count*=x;
                }
        }
    return count;
}

void main()     //главная функция
{
    float x; int n;
    cout<<"Основание > "; cin>>x;
    cout<<"Степень > "; cin>>n;
    cout<<x<<"^"<<n<<"="<<bpow(x, n);
}
```

Код программы на Pascal:

```
var x: real; n: integer;
function bpow(x: real; n: integer): real;  {быстрое возведение в степень}
    var count: real;
    begin
        if n=0 then
            begin
                bpow:=1;
                exit;
            end;
        count:=1;
        while n>0 do
            begin
                if n mod 2=0 then
                    begin
                        n:=n div 2;
                        x:=x*x;
                    end
                else
                    begin
                        n:=n-1;
                        count:=count*x;
                    end;
            end;
        bpow:=count;
    end;

begin    {основной блок программы}
    write('Основание > ');
    read(x);
```

```
    write('Степень > ');
    read(n);
    write(x, '^', n, '=', bpow(x, n));
end.
```

8.4 Решето Эратосфена

Вполне вероятно, что алгоритм, придуманный более 2000 лет назад греческим математиком Эратосфеном Киренским, был первым в своем роде. Его задача – нахождение всех простых чисел до некоторого заданного числа *n*. Термин «решето» подразумевает фильтрацию, а именно фильтрацию всех чисел за исключением простых. Так, обработка алгоритмом числовой последовательности оставит лишь простые числа, все составные же отсеются.

Рассмотрим в общих чертах работу метода. Дана упорядоченная по возрастанию последовательность натуральных чисел. Следуя методу Эратосфена, возьмем некоторое число *P* изначально равное 2 – первому простому числу, и вычеркнем из последовательности все числа кратные *P*: 2*P*, 3*P*, 4*P*, …, *iP* (*iP*≤*n*). Далее, из получившегося списка в качестве *P* берется следующее за двойкой число – тройка, вычеркиваются все кратные ей числа (6, 9, 12, …). По такому принципу алгоритм продолжает выполняться для оставшейся части последовательности, отсеивая все составные числа в заданном диапазоне.

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100

В приведенной таблице записаны натуральные числа от 2 до 100. Красным помечены те, которые удаляются в процессе выполнения алгоритма «Решето Эратосфена».

Программная реализация алгоритма Эратосфена потребует:

1. организовать логический массив и присвоить его элементам из диапазона от 2 до *n* логическую единицу;
2. в свободную переменную *P* записать число 2, являющееся первым простым числом;
3. исключить из массива все числа кратные P^2, ступая с шагом по *P*;
4. записать в *P* следующее за ним не зачеркнутое число;
5. повторять действия, описанные в двух предыдущих пунктах, пока это возможно.

Обратите внимание: на третьем шаге мы исключаем числа, начиная сразу с P^2, это связано с тем, что все составные числа меньшие P будут уже зачеркнуты. Поэтому

процесс фильтрации следует остановить, когда P^2 станет превышать n. Это важное замечание позволяет улучшить алгоритм, уменьшив число выполняемых операций.

Псевдокод алгоритма:

```
P←2
пока P²≤n выполнять
    {
        i←P²
        если B[P]=true то
            пока i≤n выполнять
                {
                    B[i]←false
                    i←i+P
                }
        P←P+1
   }
```

Он состоит из двух циклов: внешнего и внутреннего. Внешний цикл выполняется до тех пор, пока P^2 не превысит n. Само же P изменяется с шагом P+1. Внутренний цикл выполняется лишь в том случае, если на очередном шаге внешнего цикла окажется, что элемент с индексом P не зачеркнут. Именно во внутреннем цикле происходит отсеивание всех составных чисел.

Код программы на C++:

```
void Eratosthenes(bool B[], int n) //решето Эратосфена
    {
        int i, P;
        for (P=2; P<=n; P++) B[P]=true;
        P=2;
        while (P*P<=n)
            {
                i=P*P;
                if (B[P])
                    while (i<=n)
                        {
                            B[i]=false;
                            i=i+P;
                        }
                P=P+1;
            }
        cout<<"Простые числа: ";
        for (P=2; P<=n; P++)
            if (B[P]==true) cout<<" "<<P;
    }

void main()     //главная функция
```

```
{
        int n;
        cout<<"n > "; cin>>n;
        bool *B=new bool[n];
        Eratosthenes(B, n);
}
```

Код программы на Pascal:

```
type Arr=array[1..1000] of boolean;
var
n, i, P: integer;
B: Arr;
procedure Eratosthenes(B: Arr; n: integer);    {решето Эратосфена}
    begin
        for P:=2 to n do B[P]:=true;
        P:=2;
        while (P*P<=n) do
            begin
                i:=P*P;
                if B[P] then
                    while i<=n do
                        begin
                            B[i]:=false;
                            i:=i+P;
                        end;
                P:=P+1;
            end;
        write('Простые числа: ');
        for P:=2 to n do
            if B[P]=true then write(P, ' ');
    end;

begin      {основной блок программы}
    write('n > ');
    read(n);
    Eratosthenes(B, n);
end.
```

Решето Эратосфена для выявления всех простых чисел в заданной последовательности ограниченной некоторым *n* потребует *O*(*n***log* (log *n*)) операций. Поэтому уместнее использовать данный метод чем, например, наиболее тривиальный и затратный перебор делителей.

8.5 Решето Сундарама

Решето Сундарама – алгоритм поиска всех простых чисел в некотором заданном диапазоне. Он был разработан в 1934 году ныне безызвестным студентом из Индии С. П. Сундарамом.

Принцип работы алгоритма Сундарама сводится, как и в его знаменитом предшественнике, к последовательному отсеиванию всех ненужных чисел. Но у него есть одна небольшая особенность: результатом работы алгоритма будет последовательность простых чисел из диапазона от 2 до удвоенного значения граничного числа. Допустим необходимо получить все простые числа до некоторого n, тогда выходными данными будут все простые числа от 2 до $2n+1$.

Решето Сундарама из ряда натуральных чисел, не превышающих n, исключает числа вида $2ij+i+j$. Результат данного выражения, ни при каких значениях входящих в него переменных, не превышает n ($2ij+i+j \le n$). Соблюдая это условие, а также то, что i всегда меньше или равно j, переменные i и j пробегают все натуральные значения из множеств:

$i = 1, 2, 3, \ldots, \frac{\sqrt{2N+1}-1}{2}$

$j = i, i+1, i+2, \ldots, \frac{N-i}{2i+1}$

После исключения всех ненужных чисел необходимо увеличить каждое оставшиеся число в два раза и прибавить единицу ($2i+1$). Итоговое множество будет содержать числа: 2, 3, …, $2n+1$.

Код программы на C++:

```
void Sundaram(bool A[], int n)      //Решето Сундарама
    {
        int i, j;
        for (i=1; i<=n; i++) A[i]=true;
        i=1;
        j=1;
        while ((2*i*j+i+j)<=n)
            {
                while (j<=(n-i)/(2*i+1))
                    {
                        A[2*i*j+i+j]=false;
                        j++;
                    }
                i++;
                j=i;
            }
        for (i=1; i<=n; i++)
            if (A[i]) cout<<2*i+1<<" ";
    }
```

```
void main()	//главная функция
	{
		int n, i, j;
		bool A[1000];
		cout<<"n > "; cin>>n;
		cout<<"Простые числа: 2 ";
		Sundaram(A, n);
	}
```

Код программы на Pascal:

```
type arr=array [1..1000] of boolean;
var
n: integer;
A: arr;
procedure Sundaram(A: arr; n: integer);{Решето Сундарама}
	var i, j: integer;
	begin
		for i:=1 to n do A[i]:=true;
		i:=1;
		j:=1;
		while (2*i*j+i+j)<=n do
			begin
				while j<=(n-i)/(2*i+1) do
					begin
						A[2*i*j+i+j]:=false;
						j:=j+1;
					end;
				i:=i+1;
				j:=i;
			end;
		write(2, ' ');
		for i:=1 to n do
			if A[i] then write(2*i+1, ' ');
	end;

begin	{основной блок программы}
	write('n > ');
	read(n);
	writeln('Простые числа:');
	Sundaram(A, n);
end.
```

Глава. 9 Графы

9.1 Поиск в ширину

Поиск в ширину (англ. breadth-first search, BFS) – один из алгоритмов обхода графа. Метод лежит в основе некоторых других алгоритмов близкой тематики. Поиск в ширину подразумевает поуровневое исследование графа: вначале посещается корень – произвольно выбранный узел, затем – все потомки данного узла, после этого посещаются потомки потомков и т.д. Вершины просматриваются в порядке возрастания их расстояния от корня.

Пусть задан граф $G=(V, E)$ и корень s, с которого начинается обход. После посещения узла s, следующими за ним будут посещены смежные с s узлы (множество смежных с s узлов обозначим как q; очевидно, что $q \subseteq V$, то есть q – некоторое подмножество V). Далее, эта процедура повториться для вершин смежных с вершинами из множества q, за исключением вершины s, т. к. она уже была посещена. Так, продолжая обходить уровень за уровнем, алгоритм обойдет все доступные из s вершины множества V. Алгоритм прекращает свою работу после обхода всех вершин графа, либо в случае выполнения наличествующего условия.

Рассматривая пример, будем считать, что в процессе работы алгоритма каждая из вершин графа может быть окрашена в один из трех цветов: черный, белый или серый. Изначально все вершины белые. В процессе обхода каждая из вершин, по мере обнаружения, окрашивается сначала в серый, а затем в черный цвет. Определенный момент обхода описывает следующие условие: если вершина черная, то все ее потомки окрашены в серый или черный цвет.

Имеем смешанный граф (рис. 9.1) с $|V| = 4$ и $|E| = 5$. Выполним обход его вершин, используя алгоритм поиска в ширину. В качестве стартовой вершины возьмем узел 3. Сначала он помечается серым как обнаруженный, а затем черным, поскольку обнаружены смежные с ним узлы (1 и 4), которые, в свою очередь, в указанном порядке помечаются серым. Следующим в черный окрашивается узел 1, и находятся его соседи – узел 2, который становится серым. И, наконец, узлы 4 и 2, в данном порядке, просматриваются, обход в ширину завершается.

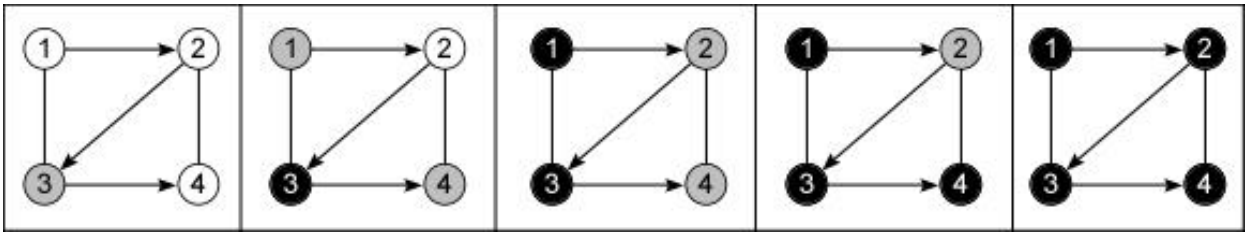


Рисунок 9.1 – Смешанный граф

Алгоритм поиска в ширину работает как на ориентированных, так и на неориентированных графах. Дать понять это был призван смешанный граф, используемый в примере. Стоит отметить, в неориентированном связном графе данный метод обойдет все имеющиеся узлы, а в смешанном или орграфе это необязательно произойдет. К тому же, до сих пор рассматривался обход всех вершин, но вполне вероятно, достаточным окажется, например просмотр определенного их количества, либо нахождение конкретной вершины. В таком случае придется немного приспособить алгоритм, а не изменять его полностью или вовсе отказаться от использования такового.

Теперь перейдем к более формальному описанию алгоритма поиска в ширину. Основными объектами – тремя структурами данных, задействованными в программе, будут:

- матрица смежности графа *GM*;
- очередь *queue*;
- массив посещенных вершин *visited*.

Две первые структуры имеют целочисленный тип данных, последняя – логический. Посещенные вершины, заносятся в массив *visited*, что предотвратит зацикливание, а очередь queue будет хранить задействованные узлы. Напомним, что структура данных «очередь» работает по принципу «первый пришел – первый вышел». Рассмотрим разбитый на этапы процесс обхода графа:

1. массив *visited* обнуляется, т. е. ни одна вершина графа еще не была посещена;
2. в качестве стартовой, выбирается некоторая вершина *s* и помещается в очередь (в массив *queue*);
3. вершина *s* исследуется (помечается как посещенная), и все смежные с ней вершины помещаются в конец очереди, а сама она удаляется;
4. если на данном этапе очередь оказывается пустой, то алгоритм прекращает свою работу, иначе посещается вершина, стоящая в начале очереди, и помечается как посещенная, все ее потомки заносятся в конец очереди;
5. выполнять пункт 4, пока это возможно.

Поиск в ширину, начиная со стартовой вершины, постепенно уходит от нее все дальше и дальше, проходя уровень за уровнем. Получается, что по окончанию работы алгоритма будут найдены все кратчайшие пути из начальной вершины до каждого доступного из нее узла.

Для реализации алгоритма на ЯП потребуется: умение программно задавать граф, а также иметь представление о такой структуре данных, как очередь.

Код программы на C++:

```
const int n=4;
int i, j;
int GM[n][n] =   //матрица смежности графа
	{
		{0, 1, 1, 0},
		{0, 0, 1, 1},
		{1, 0, 0, 1},
		{0, 1, 0, 0}
	};

void BFS(bool *visited, int unit)     //поиск в ширину
	{
		int *queue=new int[n];
		int count, head;
		for (i=0; i<n; i++) queue[i]=0;
		count=0; head=0;
		queue[count++]=unit;
		visited[unit]=true;
		while (head<count)
			{
				unit=queue[head++];
				cout<<unit+1<<" ";
				for (i=0; i<n; i++)
					if (GM[unit][i] && !visited[i])
						{
							queue[count++]=i;
							visited[i]=true;
						}
			}
		delete []queue;
	}

void main()       //главная функция
	{
		int start;
		cout<<"Стартовая вершина >> ";
		cin>>start;
		bool *visited=new bool[n];
		cout<<"Матрица смежности графа: "<<endl;
		for (i=0; i<n; i++)
			{
				visited[i]=false;
				for (j=0; j<n; j++)
					cout<<" "<<GM[i][j];
				cout<<endl;
			}
		cout<<"Порядок обхода: ";
		BFS(visited, start-1);
```

```
        delete []visited;
    }
```

Код программы на Pascal:

```
const n=4;
type
MassivInt=array[1..n, 1..n] of integer;
MassivBool=array[1..n] of boolean;
var
i, j, start: integer;
visited: MassivBool;

const GM: MassivInt =       {матрица смежности графа}
    (
        (0, 1, 1, 0),
        (0, 0, 1, 1),
        (1, 0, 0, 1),
        (0, 1, 0, 0)
    );

procedure BFS(visited: MassivBool; _unit: integer);  {поиск в ширину}
    var
    queue: array[1..n] of integer;
    count, head: integer;
    begin
        for i:=1 to n do queue[i]:=0;
        count:=0; head:=0;
        count:=count+1;
        queue[count]:=_unit;
        visited[_unit]:=true;
        while head<count do
            begin
                head:=head+1;
                _unit:=queue[head];
                write(_unit, ' ');
                for i:=1 to n do
                    begin
                    if (GM[_unit, i]<>0) and (not visited[i]) then
                        begin
                            count:=count+1;
                            queue[count]:=i;
                            visited[i]:=true;
                        end;
                    end;
            end;
    end;

begin       {основной блок программы}
    write('Стартовая вершина >> ');
```

```
    read(start);
    writeln('Матрица смежности графа: ');
    for i:=1 to n do
        begin
            visited[i]:=false;
            for j:=1 to n do write(' ', GM[i, j]);
            writeln;
        end;
    write('Порядок обхода: ');
    BFS(visited, start);
end.
```

В двух этих программах используется граф, изображенный на предыдущем рисунке, точнее его матрица смежности. На ввод может поддаваться только одно из 4 значений, т. к. в качестве стартовой возможно указать лишь одну из 4 имеющихся вершин (программы не предусматривают некорректных входных данных):

Входные данные	Выходные данные
1	1 2 3 4
2	2 3 4 1
3	3 1 4 2
4	4 2 3 1

Граф представлен матрицей смежности, и относительно эффективности это не самый лучший вариант, так как время, затраченное на ее обход, оценивается в $O(|V|^2)$, а сократить его можно до $O(|V|+|E|)$, используя список смежности.

9.2 Поиск в глубину

Поиск в глубину (англ. depth-first search, DFS) – это рекурсивный алгоритм обхода вершин графа. Если метод поиска в ширину производился симметрично (вершины графа просматривались по уровням), то данный метод предполагает продвижение вглубь до тех пор, пока это возможно. Невозможность дальнейшего продвижения, означает, что следующим шагом будет переход на последний, имеющий несколько вариантов движения (один из которых исследован полностью), ранее посещенный узел (вершина). Отсутствие последнего свидетельствует об одной из двух возможных ситуаций: либо все вершины графа уже просмотрены, либо просмотрены все те, что доступны из вершины, взятой в качестве начальной, но не все (несвязные и ориентированные графы допускают последний вариант).

Рассмотрим то, как будет вести себя алгоритм на конкретном примере. Приведенный далее неориентированный связный граф (рис. 9.2) имеет в сумме 5 вершин. Сначала необходимо выбрать начальную вершину. Какая бы вершина в качестве таковой не была выбрана, граф в любом случае исследуется полностью, поскольку, как уже было сказано, это связный граф без единого направленного ребра. Пусть обход начнется с узла 1, тогда порядок последовательности просмотренных узлов будет следующим: 1 2 3 5 4. Если выполнение начать, например, с узла 3, то порядок обхода окажется иным: 3 2 1 5 4.

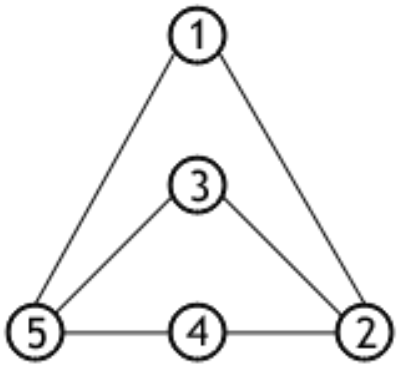


Рисунок 9.2 – Неориентированный связный граф

Алгоритм поиска в глубину базируется на рекурсии, т. е. функция обхода, по мере выполнения, вызывает сама себя, что делает код в целом довольно компактным.

Псевдокод алгоритма:

```
Заголовок функции DFS(st)
Вывести (st)
visited[st] ← посещена;
Для r←1 до n выполнять
    Если (graph[st, r] ≠ 0) и (visited[r] не посещена) то DFS(r)
```

Здесь *DFS* (depth-first search) – название функции. Единственный ее параметр *st* – стартовый узел, передаваемый из главной части программы как аргумент. Каждому из элементов логического массива *visited* заранее присваивается значение false, т. е. каждая из вершин изначально будет значиться как не посещенная. Двумерный массив *graph* – это матрица смежности графа. Большую часть внимания следует сконцентрировать на последней строке. Если элемент матрицы смежности, на каком то шаге равен 1 (а не 0) и вершина с тем же номером, что и проверяемый столбец матрицы при этом не была посещена, то функция рекурсивно повторяется. В противном случае функция возвращается на предыдущую стадию рекурсии.

Код программы на C++:

```
const int n=5;
int i, j;
bool *visited=new bool[n];
int graph[n][n] =         //матрица смежности графа
    {
        {0, 1, 0, 0, 1},
        {1, 0, 1, 1, 0},
        {0, 1, 0, 0, 1},
        {0, 1, 0, 0, 1},
        {1, 0, 1, 1, 0}
    };

void DFS(int st)  //поиск в глубину
    {
        int r;
        cout<<st+1<<" ";
        visited[st]=true;
        for (r=0; r<=n; r++)
            if ((graph[st][r]!=0) && (!visited[r]))
                DFS(r);
    }

void main()       //главная функция
    {
        int start;
        cout<<"Матрица смежности графа: "<<endl;
        for (i=0; i<n; i++)
            {
                visited[i]=false;
                for (j=0; j<n; j++)  cout<<" "<<graph[i][j];
                cout<<endl;
            }
        cout<<"Стартовая вершина >> ";
        cin>>start;
```

```
        bool *vis=new bool[n]; //массив посещенных вершин
        cout<<"Порядок обхода: ";
        DFS(start-1);
        delete []visited;
    }
```

Код программы на Pascal:

```
const
n=5;
var
i, j, start: integer;
visited: array[1..n] of boolean;
const graph: array[1..n,1..n] of byte =
    (
        (0, 1, 0, 0, 1),
        (1, 0, 1, 1, 0),
        (0, 1, 0, 0, 1),
        (0, 1, 0, 0, 1),
        (1, 0, 1, 1, 0)
    );

procedure DFS(st: integer);          //поиск в глубину
    var r: integer;
    begin
        write(st:2);
        visited[st]:=true;
        for r:=1 to n do
            if (graph[st, r]<>0) and (not visited[r]) then
                DFS(r);
    end;

begin    {основной блок программы}
    writeln('Матрица смежности:');
    for i:=1 to n do
        begin
            visited[i]:=false;
            for j:=1 to n do    write(graph[i, j],' ');
            writeln;
        end;
    write('Стартовая вершина >> '); read(start);
    writeln('Результат обхода'); DFS(start);
end.
```

Для простоты понимания результата, выдаваемого двумя программами, взят неориентированный граф, приводимый ранее в качестве примера, и представленный матрицей смежности *graph*[5][5].

9.3 Алгоритм Дейкстры

Алгоритм голландского ученого Эдсгера Дейкстры находит все кратчайшие пути из одной изначально заданной вершины графа до всех остальных. С его помощью, при наличии всей необходимой информации, можно, например, узнать какую последовательность дорог лучше использовать, чтобы добраться из одного города до каждого из многих других, или в какие страны выгодней экспортировать нефть и тому подобное. Минусом данного метода является невозможность обработки графов, в которых имеются ребра с отрицательным весом, т. е. если, например, некоторая система предусматривает убыточные для Вашей фирмы маршруты, то для работы с ней следует воспользоваться отличным от алгоритма Дейкстры методом.

Для программной реализации алгоритма понадобиться два массива: логический *visited* – для хранения информации о посещенных вершинах и численный *distance*, в который будут заноситься найденные кратчайшие пути. Итак, имеется граф $G=(V, E)$. Каждая из вершин входящих во множество V, изначально отмечена как не посещенная, т. е. элементам массива *visited* присвоено значение false. Поскольку самые выгодные пути только предстоит найти, в каждый элемент вектора distance записывается такое число, которое заведомо больше любого потенциального пути (обычно это число называют бесконечностью, но в программе используют, например максимальное значение конкретного типа данных). В качестве исходного пункта выбирается вершина s и ей приписывается нулевой путь: $distance[s]=0$, т. к. нет ребра из s в s (метод не предусматривает петель). Далее, находятся все соседние вершины (в которые есть ребро из s) [пусть таковыми будут t и u] и поочередно исследуются, а именно вычисляется стоимость маршрута из s поочередно в каждую из них:

- $distance[t]=distance[s]$+вес инцидентного s и t ребра;
- $distance[u]=distance[s]$+ вес инцидентного s и u ребра.

Но вполне вероятно, что в ту или иную вершину из s существует несколько путей, поэтому цену пути в такую вершину в массиве distance придется пересматривать, тогда наибольшее (неоптимальное) значение игнорируется, а наименьшее ставиться в соответствие вершине.

После обработки смежных с s вершин она помечается как посещенная: $visited[s]$=true, и активной становится та вершина, путь из s в которую минимален. Допустим, путь из s в u короче, чем из s в t, следовательно, вершина u становиться активной и выше описанным образом исследуются ее соседи, за исключением вершины s. Далее, u помечается как пройденная: $visited[u]$=true, активной становится вершина t, и вся процедура повторяется для нее. Алгоритм Дейкстры продолжается до тех пор, пока все доступные из s вершины не будут исследованы.

Теперь на конкретном графе (рис. 9.3) найдем все кратчайшие пути между истоковой и всеми остальными вершинами, используя алгоритм Дейкстры. Размер (количество ребер) изображенного ниже графа равен 7 ($|E|=7$), а порядок (количество вершин) – 6 ($|V|=6$). Это взвешенный граф, каждому из его ребер поставлено в

соответствие некоторое числовое значение, поэтому ценность маршрута необязательно определяется числом ребер, лежащих между парой вершин.

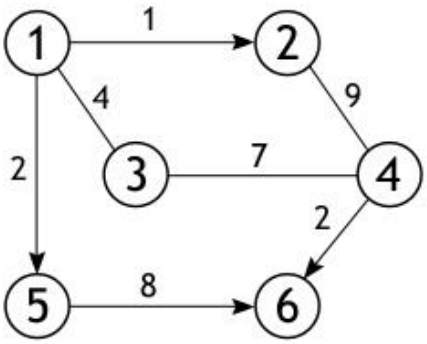


Рисунок 9.3 – Взвешенный граф

Из всех вершин входящих во множество *V* выберем одну, ту, от которой необходимо найти кратчайшие пути до остальных доступных вершин. Пусть таковой будет вершина 1 (рис. 9.4). Длина пути до всех вершин, кроме первой, изначально равна бесконечности, а до нее – 0, т. к. граф не имеет петель.

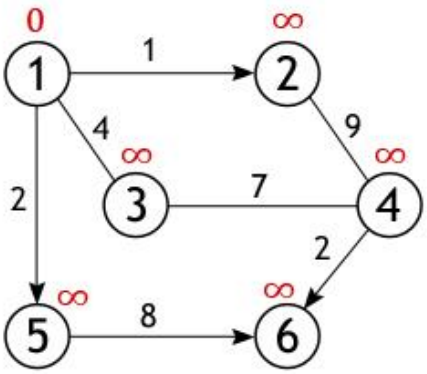


Рисунок 9.4 – Выбор стартовой вершины

У вершины 1 ровно 3 соседа (вершины 2, 3, 5), и чтобы вычислить длину пути до них, нужно сложить вес дуг, лежащих между вершинами: 1 и 2, 1 и 3, 1 и 5 со значением первой вершины (с нулем):

- 2←1+0
- 3←4+0
- 5←2+0

Как уже отмечалось, получившиеся значения присваиваются вершинам, лишь в том случае если они «лучше» (меньше) тех, которые значатся на настоящий момент. А

так как каждое из трех чисел меньше бесконечности, они становятся новыми величинами, определяющими длину пути из вершины 1 до вершин 2, 3 и 5 (рис. 9.5).

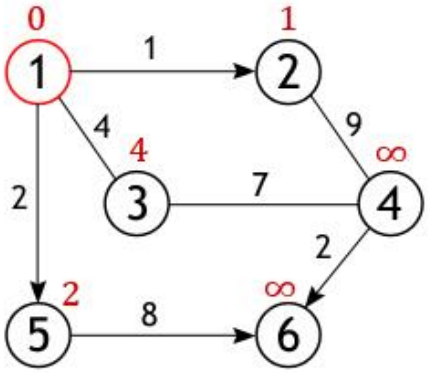


Рисунок 9.5 – Ближайшие расстояния до вершин 2, 3 и 5

Далее, активная вершина помечается как посещенная, статус «активной» (красный круг) переходит к одной из ее «соседок», а именно к вершине 2 (рис. 9.6), поскольку она ближайшая к ранее активной вершине.

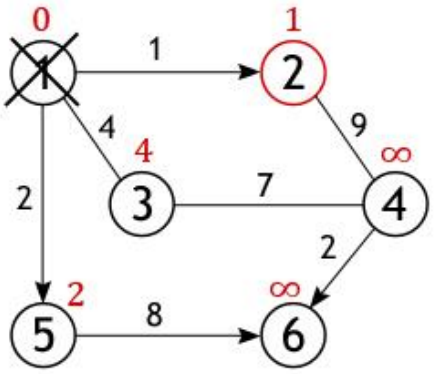


Рисунок 9.6 – Новая активная вершина

У вершины 2 всего один не рассмотренный сосед, расстояние до которого из нее равно 9, но нам необходимо вычислить длину пути из истоковой вершины, для чего нужно сложить величину приписанную вершине 2 с весом дуги из нее в вершину 4: $4 \leftarrow 1+9$

Условие «краткости» ($10<\infty$) выполняется, следовательно, вершина 4 получает новое значение длины пути (рис. 9.7).

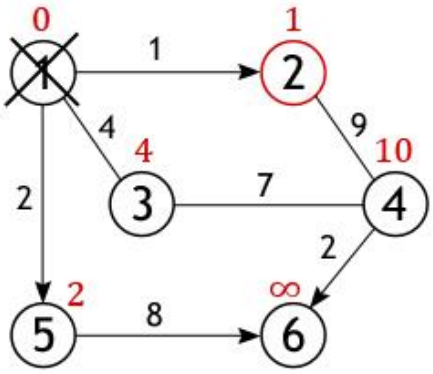


Рисунок 9.7 – Расстояние до 4-ой вершины через вершину 2

Вершина 2 перестает быть активной и удаляется из списка не посещенных вершин. Теперь тем же способом исследуются соседи вершины 5, и вычисляется расстояние до них (рис. 9.8).

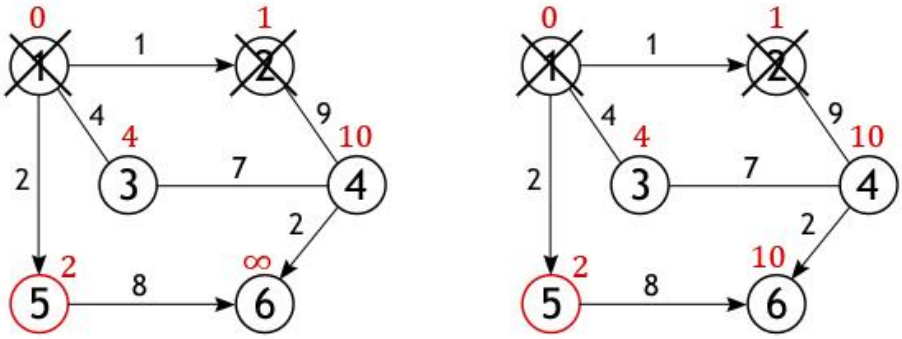


Рисунок 9.8 – Расстояния до вершин 5 и 6

Когда дело доходит до осмотра соседей вершины 3, то тут важно не ошибиться, т. к. вершина 4 уже была исследована и расстояние одного из возможных путей из истока до нее вычислено. Если двигаться в нее через вершину 3, то путь составит 4+7=11, а 11>10, поэтому новое значение игнорируется, старое остается (рис. 9.9).

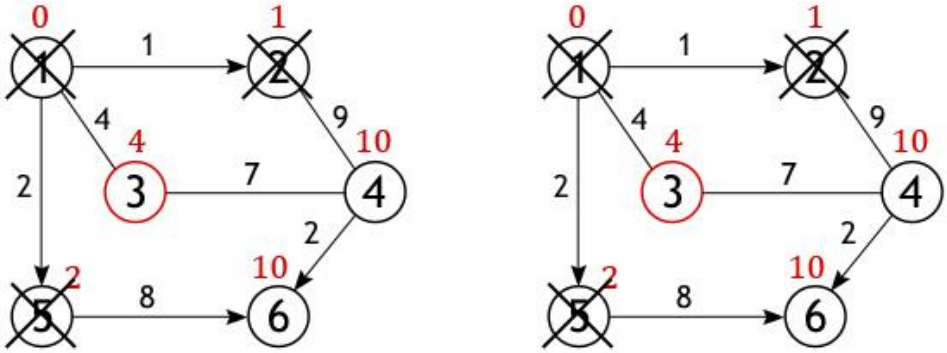


Рисунок 9.9 – Оптимальное расстояние до вершины 4

Аналогичная ситуация с вершиной 6. Значение самого близкого пути до нее из вершины 1 равно 10, а оно получается только в том случае, если идти через вершину 5.

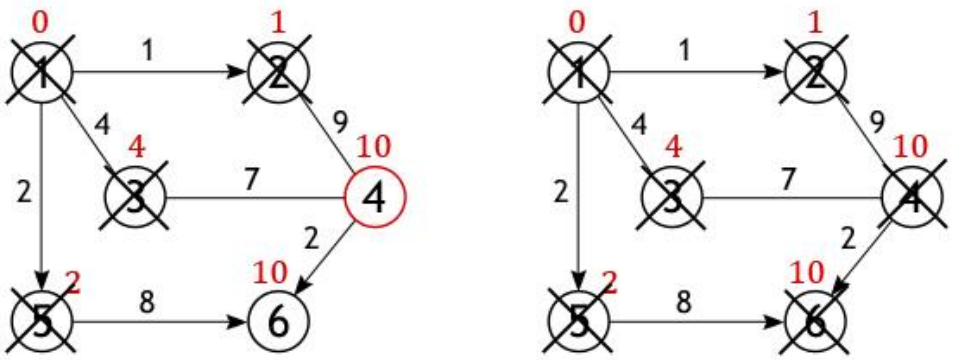


Рисунок 9.10 – Оптимальные расстояния до всех вершин

Когда все вершины графа, либо все те, что доступны из истока, будут помечены как посещенные, тогда работа алгоритма Дейкстры завершится, и все найденные пути будут кратчайшими. Так, например, будет выглядеть список самых оптимальных расстояний лежащих между вершиной 1 и всеми остальными вершинами, рассматриваемого графа:

- 1→1=0
- 1→2=1
- 1→3=4
- 1→4=10
- 1→5=2
- 1→6=10

В программе, находящей ближайшие пути между вершинами посредствам метода Дейкстры, граф будет представлен в виде не бинарной матрицы смежности. Вместо единиц в ней будут выставлены веса ребер, функция нулей останется прежней: показывать, между какими вершинами нет ребер или же они есть, но отрицательно направленны.

Код программы на C++:

```
const int V=6;
void Dijkstra(int GR[V][V], int st)   //алгоритм Дейкстры
    {
        int distance[V], count, index, i, u, m=st+1;
        bool visited[V];
        for (i=0; i<V; i++)
```

```
		{
			distance[i]=INT_MAX;
			visited[i]=false;
		}
	distance[st]=0;
	for (count=0; count<V-1; count++)
		{
			int min=INT_MAX;
			for (i=0; i<V; i++)
			if (!visited[i] && distance[i]<=min)
				{
					min=distance[i];
					index=i;
				}
			u=index;
			visited[u]=true;
			for (i=0; i<V; i++)
				if (!visited[i] && GR[u][i] && distance[u]!=INT_MAX &&
				distance[u]+GR[u][i]<distance[i])
					distance[i]=distance[u]+GR[u][i];
		}
	cout<<"Стоимость пути из начальной вершины до остальных:\t\n";
	for (i=0; i<V; i++) if (distance[i]!=INT_MAX)
		cout<<m<<" > "<<i+1<<" = "<<distance[i]<<endl;
	else cout<<m<<" > "<<i+1<<" = "<<"маршрут недоступен"<<endl;
}

void main()	//главная функция
{
	int start, GR[V][V]=
		{
			{0, 1, 4, 0, 2, 0},
			{0, 0, 0, 9, 0, 0},
			{4, 0, 0, 7, 0, 0},
			{0, 9, 7, 0, 0, 2},
			{0, 0, 0, 0, 0, 8},
			{0, 0, 0, 0, 0, 0}
		};
	cout<<"Начальная вершина >> "; cin>>start;
	Dijkstra(GR, start-1);
}
```

Код программы на Pascal:

```
const V=6; inf=100000;
type vektor=array[1..V] of integer;
var start: integer;
const GR: array[1..V, 1..V] of integer=
	(
```

```
        (0, 1, 4, 0, 2, 0),
        (0, 0, 0, 9, 0, 0),
        (4, 0, 0, 7, 0, 0),
        (0, 9, 7, 0, 0, 2),
        (0, 0, 0, 0, 0, 8),
        (0, 0, 0, 0, 0, 0)
    );

procedure Dijkstra(GR: array[1..V, 1..V] of integer; st: integer);  {алгоритм Дейкстры}
    var count, index, i, u, m, min: integer;
    distance: vektor;
    visited: array[1..V] of boolean;
    begin
        m:=st;
        for i:=1 to V do
            begin
                distance[i]:=inf;
                visited[i]:=false;
            end;
        distance[st]:=0;
        for count:=1 to V-1 do
            begin
                min:=inf;
                for i:=1 to V do
                    if (not visited[i]) and (distance[i]<=min) then
                        begin
                            min:=distance[i];
                            index:=i;
                        end;
                u:=index;
                visited[u]:=true;
                for i:=1 to V do
                    if (not visited[i]) and (GR[u, i]<>0) and (distance[u]<>inf) and
                    (distance[u]+GR[u, i]<distance[i]) then
                        distance[i]:=distance[u]+GR[u, i];
            end;
        write('Стоимость пути из начальной вершины до остальных:'); writeln;
        for i:=1 to V do
            if distance[i]<>inf then
                writeln(m,' > ', i,' = ', distance[i])
            else writeln(m,' > ', i,' = ', 'маршрут недоступен');
    end;

begin       {основной блок программы}
    write('Начальная вершина >> '); read(start);
    Dijkstra(GR, start);
end.
```

9.4 Алгоритм Флойда – Уоршелла

Наиболее часто используемое название, метод получил в честь двух американских исследователей Роберта Флойда и Стивена Уоршелла, одновременно открывших его в 1962 году. Реже встречаются другие варианты наименований: алгоритм Рой – Уоршелла или алгоритм Рой – Флойда. Рой – фамилия профессора, который разработал аналогичный алгоритм на 3 года раньше коллег (в 1959 г.), но это его открытие осталось безвестным. Алгоритм Флойда – Уоршелла – динамический алгоритм вычисления значений кратчайших путей для каждой из вершин графа. Метод работает на взвешенных графах, с положительными и отрицательными весами ребер, но без отрицательных циклов, являясь, таким образом, более общим в сравнении с алгоритмом Дейкстры, т. к. последний не работает с отрицательными весами ребер, и к тому же классическая его реализация подразумевает определение оптимальных расстояний от одной вершины до всех остальных.

Для реализации алгоритма Флойда – Уоршелла сформируем матрицу смежности $D[][]$ графа $G=(V, E)$, в котором каждая вершина пронумерована от 1 до $|V|$. Эта матрица имеет размер $|V|\times|V|$, и каждому ее элементу $D[i][j]$ присвоен вес ребра, идущего из вершины i в вершину j. По мере выполнения алгоритма, данная матрица будет перезаписываться: в каждую из ее ячеек внесется значение, определяющее оптимальную длину пути из вершины i в вершину j (отказ от выделения специального массива для этой цели сохранит память и время). Теперь, перед составлением основной части алгоритма, необходимо разобраться с содержанием матрицы кратчайших путей. Поскольку каждый ее элемент $D[i][j]$ должен содержать наименьший из имеющихся маршрутов, то сразу можно сказать, что для единичной вершины он равен нулю, даже если она имеет петлю (отрицательные циклы не рассматриваются), следовательно, все элементы главной диагонали ($D[i][i]$) нужно обнулить. А чтобы нулевые недиагональные элементы (матрица смежности могла иметь нули в тех местах, где нет непосредственного ребра между вершинами i и j) сменили по возможности свое значение, определим их равными бесконечности, которая в программе может являться, например, максимально возможной длинной пути в графе, либо просто – большим числом.

Ключевая часть алгоритма, состоя из трех циклов, выражения и условного оператора, записывается довольно компактно:

Для k от 1 до $|V|$ выполнять
Для i от 1 до $|V|$ выполнять
Для j от 1 до $|V|$ выполнять
Если $D[i][k]+D[k][j]<D[i][j]$ то $D[i][j] \leftarrow D[i][k]+D[k][j]$

Кратчайший путь из вершины i в вершину j может проходить, как только через них самих, так и через множество других вершин $k\in(1, \ldots, |V|)$. Оптимальным из i в j будет путь или не проходящий через k, или проходящий. Заключить о наличии второго случая,

значит установить, что такой путь идет из *i* до *k*, а затем из *k* до *j*, поэтому должно заменить, значение кратчайшего пути *D*[*i*][*j*] суммой *D*[*i*][*k*]+*D*[*k*][*j*].

Рассмотрим полный код алгоритма Флойда – Уоршелла на C++ и Паскале, а затем детально разберем последовательность выполняемых им действий.

Код программы на C++:

```
const int maxV=1000;
int i, j, n;
int GR[maxV][maxV];

void FU(int D[][maxV], int V)        //алгоритм Флойда-Уоршелла
    {
    int k;
    for (i=0; i<V; i++) D[i][i]=0;

    for (k=0; k<V; k++)
        for (i=0; i<V; i++)
        for (j=0; j<V; j++)
            if (D[i][k] && D[k][j] && i!=j)
                if (D[i][k]+D[k][j]<D[i][j] || D[i][j]==0)
                    D[i][j]=D[i][k]+D[k][j];
    for (i=0; i<V; i++)
        {
            for (j=0; j<V; j++) cout<<D[i][j]<<"\t";
            cout<<endl;
        }
    }

void main()      //главная функция
    {
        cout<<"Количество вершин в графе > "; cin>>n;
        cout<<"Введите матрицу весов ребер:\n";
        for (i=0; i<n; i++)
            for (j=0; j<n; j++)
                {
                    cout<<"GR["<<i+1<<"]["<<j+1<<"] > ";
                    cin>>GR[i][j];
                }
        cout<<"Матрица кратчайших путей:"<<endl;
        FU(GR, n);
    }
```

Код программы на Pascal:

```
const maxV=1000;
type matr=array[1..maxV, 1..maxV] of integer;
var i, j, n, inf: integer;
GR: matr;
```

```
Procedure FU(D: matr; V: integer);        {алгоритм Флойда-Уоршелла}
    var k: integer;
    begin
        inf:=1000000;
        for i:=1 to V do D[i, i]:=0;

        for k:=1 to V do
            for i:=1 to V do
                for j:=1 to V do
                    if (D[i, k]<>0) and (D[k, j]<>0) and (i<>j) then
                        if (D[i, k]+D[k, j]<D[i, j]) or (D[i, j]=0) then
                            D[i, j]:=D[i, k]+D[k, j];

        for i:=1 to V do
            begin
                for j:=1 to V do
                    write(D[i, j]:4);
                writeln;
            end;
    end;

begin       {главный блок программы}
    write('Количество вершин в графе > ');
    readln(n);
    writeln('Введите матрицу весов ребер:');
    for i:=1 to n do
        for j:=1 to n do
            begin
                write('GR[', i, '][', j, '] > ');
                read(GR[i, j]);
            end;
    writeln('Матрица кратчайших путей:');
    FU(GR, n);
end.
```

Положим, что в качестве матрицы смежности, каждый элемент которой хранит вес некоторого ребра, была задана следующая матрица:

0	9	2
1	0	4
2	4	0

Количество вершин в графе, представлением которого является данная матрица, равно 3, и, причем между каждыми двумя вершинами существует ребро. Ниже показан граф этой матрицы смежности (рис. 9.11).

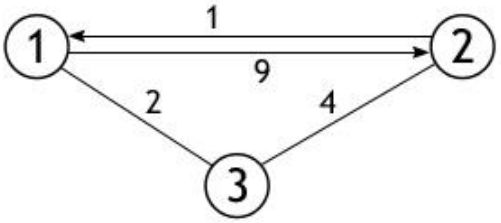


Рисунок 9.11 – Смешанный граф

Задача алгоритма: перезаписать данную матрицу так, чтобы каждая ячейка вместо веса ребра из *i* в *j*, содержала кратчайший путь из *i* в *j*. Для примера взят совсем маленький граф, и поэтому не будет не чего удивительного, если матрица сохранит свое изначальное состояние. Но результат тестирования программы показывает замену двух значений в ней. Таблица (рис. 9.12) поможет с анализом этого конкретного примера.

k	i	j	замена
1	1	1	
1	1	2	
1	1	3	
1	2	1	
1	2	2	
1	2	3	3<4, D[2][3] ←3
1	3	1	
1	3	2	
1	3	3	
2	1	1	
2	1	2	
2	1	3	
2	2	1	
2	2	2	
2	2	3	
2	3	1	
2	3	2	
2	3	3	
3	1	1	
3	1	2	6<9, D[1][2] ←6
3	1	3	
3	2	1	
3	2	2	
3	2	3	
3	3	1	
3	3	2	
3	3	3	

0	9	2
1	0	4
2	4	0

мат. смеж.

0	9	2
1	0	3
2	4	0

0	6	2
1	0	4
2	4	0

Рисунок 9.12 – Нахождением оптимальных расстояний для графа 8.11

В данной таблице показаны 27 шагов выполнения основной части алгоритма. Их столько по той причине, что время выполнения метода равно $O(|V|^3)$. Наш граф имеет 3 вершины, а 3^3=27. Первая замена происходит на итерации, при которой k=1, i=2, а j=3. В тот момент D[2][1]=1, D[1][3]=2, D[2][3]=4. Условие истинно, т. е. D[1][3]+D[3][2]=3, а 3<4, следовательно, элемент матрицы D[2][3] получает новое значение. Следующий шаг, когда условие также истинно привносит изменения в элемент, расположенный на пересечении второй строки и третьего столбца.

9.5 Алгоритм Беллмана — Форда

История алгоритма связана сразу с тремя независимыми математиками: Лестером Фордом, Ричардом Беллманом и Эдвардом Муром. Форд и Беллман опубликовали алгоритм в 1956 и 1958 годах соответственно, а Мур сделал это в 1957 году. И иногда его называют алгоритмом Беллмана – Форда – Мура. Метод используется в некоторых протоколах дистанционно-векторной маршрутизации, например в RIP (Routing Information Protocol – Протокол маршрутной информации).

Также как и алгоритм Дейкстры, алгоритм Беллмана — Форда вычисляет во взвешенном графе кратчайшие пути от одной вершины до всех остальных. Он подходит для работы с графами, имеющими ребра с отрицательным весом. Но спектр применимости алгоритма затрагивает не все такие графы, ввиду того, что каждый очередной проход по пути, составленному из ребер, сумма весов которых отрицательна (т. е. по отрицательному циклу), лишь улучшает требуемое значение. Бесконечное число улучшений делает невозможным определение одного конкретного значения, являющегося оптимальным. В связи с этим алгоритм Беллмана — Форда не применим к графам, имеющим отрицательные циклы, но он позволяет определить наличие таковых, о чем будет сказано позже.

Решить задачу, т. е. найти все кратчайшие пути из вершины *s* до всех остальных, используя алгоритм Беллмана — Форда, это значит воспользоваться методом динамического программирования: разбить ее на типовые подзадачи, найти решение последним, покончив тем самым с основной задачей. Здесь решением каждой из таких подзадач является определение наилучшего пути от одного отдельно взятого ребра, до какого-либо другого. Для хранения результатов работы алгоритма заведем одномерный массив *d*[]. В каждом его *i*-ом элементе будет храниться значение кратчайшего пути из вершины s до вершины *i* (если таковое имеется). Изначально, присвоим элементам массива *d*[] значения равные условной бесконечности (например, число заведомо большее суммы всех весов), а в элемент *d*[*s*] запишем нуль. Так мы задействовали известную и необходимую информацию, а именно известно, что наилучший путь из вершины *s* в нее же саму равен 0, и необходимо предположить недоступность других вершин из *s*. По мере выполнения алгоритма, для некоторых из них, это условие окажется ложным, и вычисляться оптимальные стоимости путей до этих вершин из *s*.

Задан граф $G=(V, E)$, $n=|V|$, а $m=|E|$. Обозначим смежные вершины этого графа символами *v* и *u* ($v \in V$ и $u \in V$), а вес ребра (*v*, *u*) символом *w*. Иначе говоря, вес ребра, выходящего из вершины *v* и входящего в вершину *u*, будет равен *w*. Тогда ключевая часть алгоритма Беллмана — Форда примет следующий вид:

```
Для i от 1 до n-1 выполнять
Для j от 1 до m выполнять
Если d[v]+w(v, u)<d[u] то
d[u]←d[v]+w(v, u)
```

На каждом *n*-ом шаге осуществляются попытки улучшить значения элементов массива *d*[]: если сумма, составленная из веса ребра *w*(*v*, *u*) и веса хранящегося в элементе *d*[*v*], меньше веса *d*[*u*], то она присваивается последнему.

Код программы на C++:

```
#define inf 100000
struct Edges
    { int u, v, w; };
const int Vmax=1000;
const int Emax=Vmax*(Vmax-1)/2;
int i, j, n, e, start;
Edges edge[Emax];
int d[Vmax];

void bellman_ford(int n, int s)      //алгоритм Беллмана-Форда
    {
        int i, j;
        for (i=0; i<n; i++) d[i]=inf;
        d[s]=0;
        for (i=0; i<n-1; i++)
            for (j=0; j<e; j++)
                if (d[edge[j].v]+edge[j].w<d[edge[j].u])
                    d[edge[j].u]=d[edge[j].v]+edge[j].w;
        for (i=0; i<n; i++)
            if (d[i]==inf)
                cout<<endl<<start<<"->"<<i+1<<"="<<"Not";
            else cout<<endl<<start<<"->"<<i+1<<"="<<d[i];
    }

void main()      //главная функция
    {
        int w;
        cout<<"Количество вершин > "; cin>>n;
        e=0;
        for (i=0; i<n; i++)
            for (j=0; j<n; j++)
                {
                    cout<<"Вес "<<i+1<<"->"<<j+1<<" > "; cin>>w;
                    if (w!=0)
                        {
                            edge[e].v=i;
                            edge[e].u=j;
                            edge[e].w=w;
                            e++;
                        }
                }
        cout<<"Стартовая вершина > "; cin>>start;
        cout<<"Список кратчайших путей:";
        bellman_ford(n, start-1);
```

```
}
```

Код программы на Pascal:

```
const
inf=100000;
Vmax=1000;
Emax=Vmax*(Vmax-1) div 2;
type Edges=record
u, v, w: integer;
end;
Var
i, j, e, n, w, start: integer;
edge: array[1..Emax] of Edges;
d: array[1..Vmax] of integer;

procedure FB(n, s: integer); {алгоритм Беллмана-Форда}
    begin
        for i:=1 to n do d[i]:=inf;
        d[s]:=0;
        for i:=1 to n-1 do
            for j:=1 to e-1 do
                if d[edge[j].v]+edge[j].w<d[edge[j].u] then
                    d[edge[j].u]:=d[edge[j].v]+edge[j].w;
        for i:=1 to n do
            if d[i]=inf then
                writeln(start, '->', i, '=', 'Not')
            else writeln(start, '->', i, '=', d[i]);
    end;

begin    {основной блок программы}
    write('Количество вершин > ');
    read(n);
    e:=1;
    for i:=1 to n do
        for j:=1 to n do
            begin
                write('Вес ', i, '->', j, ' > ');
                read(w);
                if w<>0 then
                    begin
                        edge[e].v:=i;
                        edge[e].u:=j;
                        edge[e].w:=w;
                        e:=e+1;
                    end;
            end;
    write('Стартовая вершина > '); read(start);
    writeln('Список кратчайших путей:');
```

```
    FB(n, start);
end.
```

Здесь граф представлен упрощенным списком ребер, который формируется по мере ввода пользователем матрицы весов. Основная часть алгоритма Беллмана – Форда (проверка условия) выполняется $m*(n-1)$ раз, т. к. число повторений внешнего цикла равно n-1, а внутреннего – m. Отказ от n-ой итерации целесообразен, поскольку алгоритм справляется со своей задачей за n-1 шаг, но запуск внешнего цикла n раз позволит выявить наличие отрицательного цикла в графе. Проверить это можно, например, при помощи следующей модификации:

```
bool x=true;
for (i=0; i<n; i++)
    {
        if (i!=n-1)
            for (j=0; j<e; j++)
                if (d[edge[j].v]+edge[j].w<d[edge[j].u])
                    d[edge[j].u]=d[edge[j].v]+edge[j].w;
        if (i==n-1)
            for (j=0; j<e; j++)
                if (d[edge[j].v]+edge[j].w<d[edge[j].u])
                    x=false;
    }
if (!x) cout<<endl<<"Граф содержит отриц. циклы"<<endl;
```

Здесь внешний цикл выполняется n раз (в C++ принято начальным указывать значение 0, поэтому для данного кода n-1 раз), и если на n-ой стадии будет возможно улучшение, то это свидетельствует о наличие отрицательного цикла.