БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
факультет радиофизики и компьютерных технологий
кафедра информатики и компьютерных систем

Н.В. Серикова

ПРАКТИЧЕСКОЕ РУКОВОДСТВО

к лабораторному практикуму

«ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ»

по курсу
«ПРОГРАММИРОВАНИЕ»

2018
МИНСК

Практическое руководство к лабораторному практикуму «ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ» по курсу «ПРОГРАММИРОВАНИЕ» предназначено для студентов, изучающих базовый курс программирования на языке C++, специальностей «Радиофизика», «Физическая электроника», «Компьютерная безопасность».

Руководство содержит некоторый справочный материал, примеры решения типовых задач с комментариями.

Все примеры протестированы в среде Microsoft Visual Studio 2005.

Автор будет признателен всем, кто поделится своими соображениями по совершенствованию данного пособия.

Возможные предложения и замечания можно присылать по адресу:
E-mail: Serikova@bsu.by,

ОГЛАВЛЕНИЕ

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Статическими величинами называются такие, память под которые выделяется во время компиляции и сохраняется в течение всей работы программы.

На основании информации из раздела описаний и длины кода самой программы операционная система во время работы программы выделяет определённый объём оперативной памяти, который закрепляется за программой на всё время её выполнения. Такой подход снижает эффективность использования оперативной памяти: например, при объявлении большого массива данных, который используется малую часть общего времени работы программы; или ситуации, при которых требуется объём памяти, больший того, который может быть доступен программе в статическом режиме.

В языках программирования существует другой способ выделения памяти под данные, который называется **динамическим**.

В этом случае память под величины отводится во время выполнения программы. Такие величины будем называть **динамическими**. Раздел оперативной памяти, распределяемый статически, называется **статической памятью**; динамически распределяемый раздел памяти называется **динамической памятью (динамически распределяемой памятью)**.

Использование динамических величин предоставляет программисту ряд дополнительных **возможностей**.

- Во-первых, подключение динамической памяти позволяет увеличить объем обрабатываемых данных.
- Во-вторых, если потребность в каких-то данных отпала до окончания программы, то занятую ими память можно освободить для другой информации.
- В-третьих, использование динамической памяти позволяет создавать структуры данных переменного размера.

В процессе работы программы специальной процедурой программист может запросить у системы некоторый объём памяти, а после использования (также специальной процедурой) возвратить её системе, т.е. динамическое управление памятью – это выделение памяти во время выполнения программы.

Преимущества. Разумное использование динамических структур данных приводит к сокращению объёма памяти, необходимого для работы программы; динамические данные не требуют объявлений их как данных фиксированного размера; ряд алгоритмов более эффективен при реализации их с использованием динамических структур. Например, вставка элемента в массив на определенное место требует перемещения части элементов массива. При вставке в середину списка достаточно несколько операторов присваивания.

Недостатки. Алгоритмы для динамических структур обычно более сложны, трудны для отладки по сравнению с аналогичными алгоритмами для статических данных; использование динамических структур требует затрат на память для ссылок. В некоторых задачах объём памяти, отводимой для ссылок, превосходит объём памяти, выделяемой непосредственно для данных; существуют алгоритмы, реализация которых более эффективна на обычных данных.

НЕСВЯЗАННЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

ОДНОМЕРНЫЕ МАССИВЫ

Если до начала работы неизвестно, сколько в массиве элементов, то следует использовать динамические массивы. Память под них выделяется с помощью функции *new* во время выполнения программы.

```
int   n = 10;
int   *b = new int; n     // выделение памяти для переменной
                          // целого типа
int   *a = new int [n];   // выделение памяти для массива
                          // переменных целого типа
```

Переменная *a* – указатель на целую величину, которому присваивается адрес непрерывной области динамической памяти, выделенной функцией *new*. Выделяется столько памяти, сколько необходимо для хранения *n* величин типа *int*.

Обнуление памяти при ее выделении не происходит.

Инициализировать динамический массив нельзя.

Обращение к элементу динамического массива происходит также, как и к элементу обычного – например a[2] или *(a + 2).

Если динамический массив в какой-то момент работы программы перестает быть нужным, то необходимо освободить память с помощью функции *delete*, например

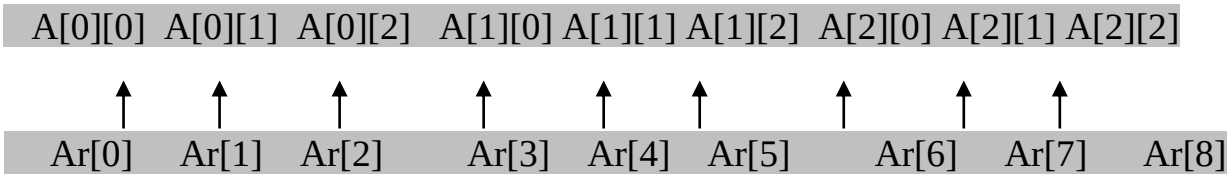
```
delete[] a;
delete b;
```

ДВУМЕРНЫЕ МАССИВЫ

Существует 2 способа организации работы с двумерным массивом.

1 способ

Двумерный массив можно представить в виде одномерного, а «двумерность» учитывать при обращении к элементам массива. Обращение к элементу одномерного массива Ar[i*ncol + j] есть фактически обращение к элементу двумерного массива A с индексами i, j.



Описание

```
int  *Array;
int  nrow,  ncol;      // размерности матрицы
```

Выделение памяти:

```
Array = new int[nrow * ncol];
```

Освобождение памяти:

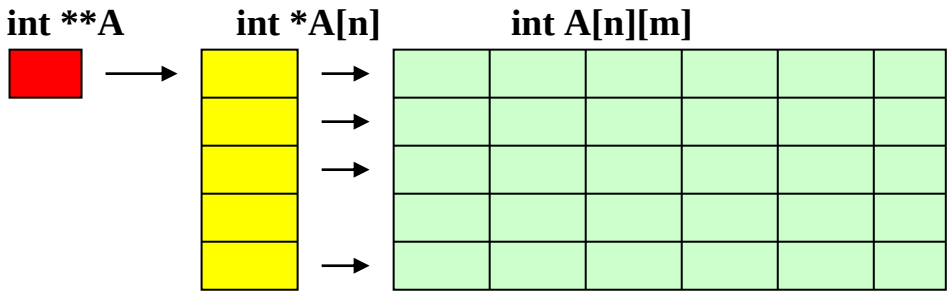
```
delete[] Array;
```

Обращение к элементу [i][j]: `Array[f(i, j, ncol)]`

где f(i, j, ncol) = i * ncol + j - определение номера элемента [i][j] матрицы в одномерном массиве

2 СПОСОБ

Двумерный массив представляется как указатель на массив указателей на строки матрицы.



Описание

```
int **Array;
int nrow, ncol;                 // размерности матрицы
```

Выделение памяти:

```
                                // выделяется память под массив
                                // указателей на строки массива
Array = new int* [nrow];

                                // цикл для выделения памяти под
                                // строки массива
for (int i = 0; i < nrow; i++)
    Array[i] = new int[ncol];
```

Освобождение памяти:

```
for (int i = 0; i < nrow; i++)
    delete Array[i];
delete[] Array ;
```

Обращение к элементу [i][j]:

```
Array[i][j]
```

СВЯЗАННЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Связанные динамические структуры по определению характеризуются отсутствием физической смежности элементов структуры в памяти, непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки.

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Для установления связи между элементами динамической структуры используются указатели, через которые устанавливаются явные связи между элементами.

Такое представление данных в памяти называется **связным**.

Элемент динамической структуры состоит из двух полей:

- информационного поля или поля данных, в котором содержатся те данные, ради которых и создается структура; в общем случае информационное поле само является интегрированной структурой - вектором, массивом, другой динамической структурой и т.п.;
- поле связок, в котором содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры.

Когда связное представление данных используется для решения прикладной задачи, для конечного пользователя "видимым" делается только содержимое информационного поля, а поле связок используется только программистом-разработчиком.

Достоинства связного представления данных - в возможности обеспечения значительной изменчивости структур:

- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
- большая гибкость структуры.

Вместе с тем связное представление не лишено и **недостатков**, основные из которых:

- на поля связок расходуется дополнительная память;
- доступ к элементам связной структуры может быть менее эффективным по времени.

Последний недостаток является наиболее серьёзным и именно им ограничивается применимость связного представления данных. Если в смежном представлении данных для вычисления адреса любого элемента нам во всех случаях достаточно было номера элемента и информации, содержащейся в дескрипторе структуры, то для связного представления адрес элемента не может быть вычислен из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, далее поиск требуемого элемента выполняется следованием по цепочке указателей от элемента к элементу. Поэтому связное представление практически никогда не применяется в задачах, где логическая структура данных имеет вид вектора или массива - с доступом по номеру элемента, но часто применяется в задачах, где логическая структура требует другой исходной информации доступа.

Связные динамические структуры можно разделить на **списки, графы, деревья**.

Список - последовательность однородных элементов (называемых **узлами**), каждый из которых содержит ссылку на следующий (и/или предыдущий) элемент.

Списки делятся на **линейные и кольцевые**.

Списки делятся на **односвязные и двусвязные**.

Размер списка - количество элементов списка.

Список, в котором нет элементов, называется **пустым списком**.

Узлом списка называют элемент списка. Узел в связном списке состоит из двух частей. Первая часть содержит непосредственно данные, вторая часть содержит один или несколько адресов элементов, с которым связан данный узел.

Список, в узлах которого задаётся адрес только одного (предыдущего или последующего) элемента, называются **однонаправленным (односвязным**).

Список, в узлах которого, задаются адреса и предыдущего и последующего элементов, называются **двунаправленным (двусвязным**).

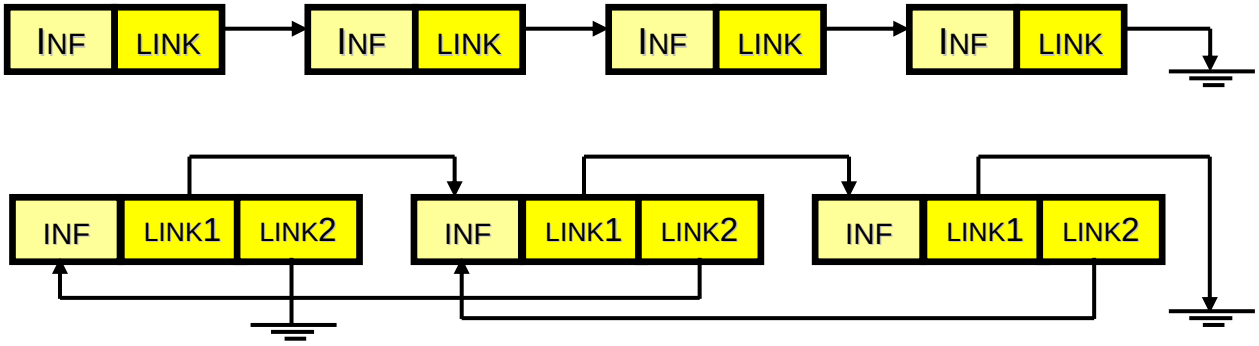
Указатель на первый элемент списка – это **адрес списка**. Значение «**NULL**» (“0”) этого указателя означает, что список пуст.

Линейным списком называется последовательность однородных элементов (называемых узлами), каждый из которых (кроме последнего) содержит ссылку на следующий (и/или предыдущий) элемент.

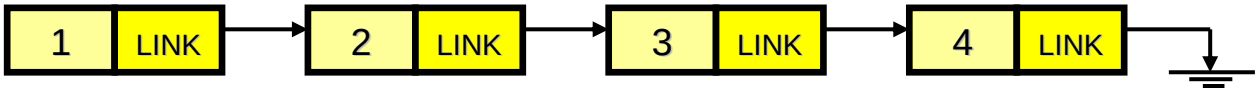
Линейный список может быть односвязным и двусвязным.

Линейные списки делятся на **стеки, очереди, линейные списки**.

Линейные списки – данные динамической структуры, которые представляют собой совокупность линейно связанных однородных элементов, для которых разрешается добавлять элементы между любыми двумя другими и удалять любой элемент.



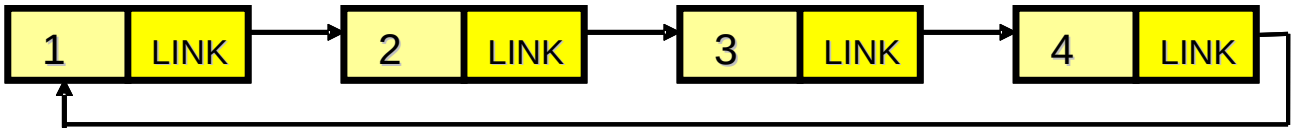
Очередь – частный случай линейного списка – разрешено только 2 действия – добавление элементов в конец (хвост) и удаление из начала (головы) списка.



Стек – частный случай линейного списка – разрешено только 2 действия – добавление и удаление элементов с одного конца (головы) стека.

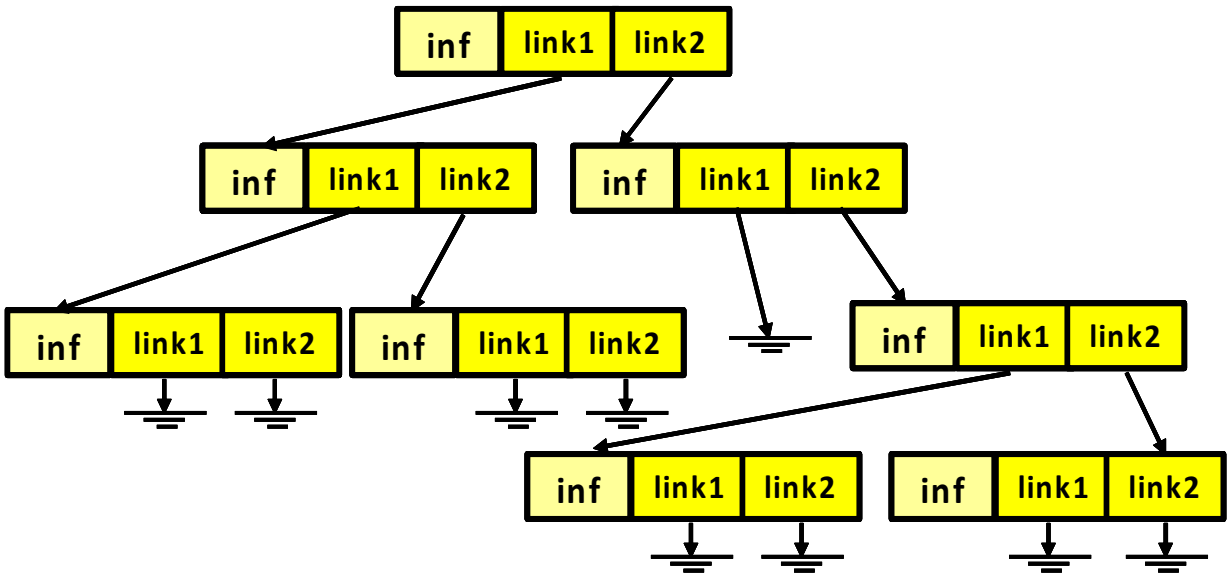
Кольцевой список – это линейный список, последний элемент которого содержит ссылку на первый элемент списка.

Кольцевой список также как линейный список может быть односвязным и двусвязным.

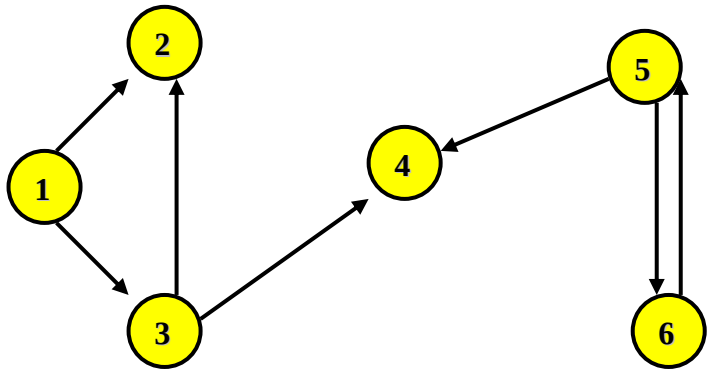


Деревья – иерархические динамические структуры, состоящие из узлов, каждый из которых содержит, кроме данных, ссылки на различные деревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется корнем дерева.

Бинарное дерево – это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных, **не более двух** ссылок на различные бинарные деревья.



Граф – самая общая нелинейная динамическая структура данных. В графе каждый узел содержит несколько указателей на несколько узлов.



СТЕК

Стек – частный случай **односвязного линейного списка** – разрешено только 2 действия – добавление и удаление элементов с одного конца (головы) стека. Другие операции со стеком не определены.

Стек реализует принцип обслуживания **LIFO** (**L**ast **I**n – **F**irst **O**ut, последний пришел – первый ушел).

Базовые операции над стеком:

- добавление в стек нового элемента, новый элемент помещается в вершину стека;
- чтение данных из вершины стека («выталкивание» верхнего элемента);
- вывод всех элементов стека;

Реализация стека в виде массива

Если заранее известно максимальное количество элементов (*l*), одновременно хранящихся в стеке, то целесообразно моделировать стек на массиве *A* постоянной длины. Пусть переменная *top* содержит индекс последнего включенного в стек элемента (первоначально *top*=-1), тогда процедуры включения и исключения элемента из стека реализуются следующим образом:

Включение элемента *x* в стек *A*

```
if (top < l - 1)
{
    top++;
    A[top] = x;
}
```

Исключение элемента из стека *A*

```
if (top >= 0)
{
    x = A[top];
    top--;
}
```

Реализация стека в виде связных компонент

Элемент стека состоит из полей разной природы, поэтому описывается через структуру, где основное поле – это поле для указания адреса следующего элемента в стеке.

Описание элемента стека:

```
struct Node
{
    int   d;      // поля произвольных типов
    Node *link;  // указатель на следующий элемент стека
};
```

АЛГОРИТМ ВЫЧИСЛЕНИЯ ВЫРАЖЕНИЙ ЧЕРЕЗ ОБРАТНУЮ ПОЛЬСКУЮ ЗАПИСЬ

Обычные алгебраические (логические, математические) выражения можно записывать в виде **обратной польской нотации** – записи без скобок.

Нотация – «польская» в честь польского математика Яна Лукашевича, который ее предложил. Нотация – «обратная» из-за того, что порядок операндов и операций обратный относительно исходного.

Пример:

Выражение		Обратная польская нотация
A + (B - C) * D – F / (G + H) =	→	A B C – D * + F G H + / - =

Если A = 6 B = 4 C = 1 D = 2 F = 3 G = 7 H = 5.
Тогда обратная польская нотация: 6 4 1 – 2 * + 3 7 5 + / - =

Алгоритм вычисления выражения.

Вычислить обратное польское выражение проще, чем обычное алгебраическое.

Просматриваем все элементы слева направо. Как только встречается операция, выполняем ее по отношению к двум предыдущим элементам, заменяя два элемента одним.

6 4 1 – 2 * + 3 7 5 + / - = выполняем 4-1=3 →
6 3 2 * + 3 7 5 + / - = выполняем 3*2=6 →
6 6 + 3 7 5 + / - = выполняем 6+6=12 →
12 3 7 5 + / - = выполняем 7+5=12 →
12 3 12 / - = выполняем 3/12=0.25 →
12 0.25 - = выполняем 12-0.25=11.75 →
11.75 = результат

Алгоритм преобразования выражения в обратную польскую нотацию.

Необходимо использовать два стека (списка) X и Y.
Необходимо определить приоритеты операций. Например, так:

операции	приоритет
* /	3
+ -	2
(	1
=	0

1. Просматриваем выражение слева направо. Операнды помещаем в стек X , левые скобки и операции в стек Y.
2. Встретив правую скобку, отыскиваем в стеке соответствующую ей левую. При этом все, что сверху – выталкивается из стека Y и заносится в стек X.
3. Если приоритет очередной операции меньше приоритета операции вершины стека , то операции из вершины стека Y выталкиваются в стек X, пока не найдём операцию с приоритетом ниже, либо пока стек не окажется пустым .
4. Результат оказывается с стеке X.

Пример преобразования выражения в обратную польскую запись

A + (B - C) * D – F / (G + H) =

	стек **X**	стек **Y**
1. A	A	
2.+	A	+
3. (	A	(→ +
4.B	B → A	(→ +
5. -	B → A	- → (→ +
6. C)	C → B → A	- → (→ +
7.)	- → C → B → A	+
8. *	- → C → B → A	* → +
9. D	D → - → C → B → A	* → +
10. –	+ → * → D → - → C → B → A	-
11. F	F → + → * → D → - → C → B → A	-
12. /	F → + → * → D → - → C → B → A	/ → -
13. (	F → + → * → D → - → C → B → A	(→ /→ -
14. G	G → F → + → * → D → - → C → B → A	(→ / → -
15. +	G → F → + → * → D → - → C → B → A	+ → (→ / → -
16. H	H → G → F → + → * → D → - → C → B → A	+→ (→ /→ -
17.)	+ → H → G → F → +→ *→ D → - → C → B → A	/ → -
18. =	- → / → + → H → G → F → + → *→ D→ - → C→ B→ A	

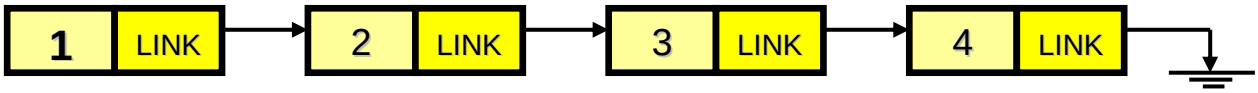
результат в обратном порядке

A B C – D * + F G H + / - =

ОЧЕРЕДЬ

Очередь – частный случай **односвязного линейного списка** – разрешено только 2 действия – добавление элементов в конец (хвост) и удаление из начала (головы) списка. Другие операции со стеком не определены.

Очередь реализует принцип обслуживания **FIFO** (**F**irst **I**n – **F**irst **O**ut, первый пришел – первый ушел).



Базовые операции над очередью:

- добавление в очередь нового элемента, новый элемент помещается в конец очереди;
- чтение данных из начала очереди («выталкивание» первого элемента);
- вывод всех элементов очереди;

Реализация очереди в виде массива

Если заранее известно максимальное количество элементов (*l*), одновременно хранящихся в очереди, то целесообразно моделировать очередь на массиве *A* постоянной длины. С очередью свяжем две переменные *top* и *end*, определяющие индексы первого и последнего элементов очереди и переменную булевского типа *empty*, которая равна *true* в случае, когда очередь пуста. Тогда процедуры включения и исключения элемента из очереди реализуются следующим образом:

Включение элемента *x* в очередь *A*

```
if (empty)
{    top = 0;
     A[top] = x;
     empty = false;
     end = top;
}
else
{    end++;
     if (end < l)
          A[end] = x;
}
```

Исключение элемента из очереди *A*

```
if (!empty)
{
     x = A[top];

     if (top == end)
        empty = true;
     top++;
}
```

Реализация очереди в виде связных компонент

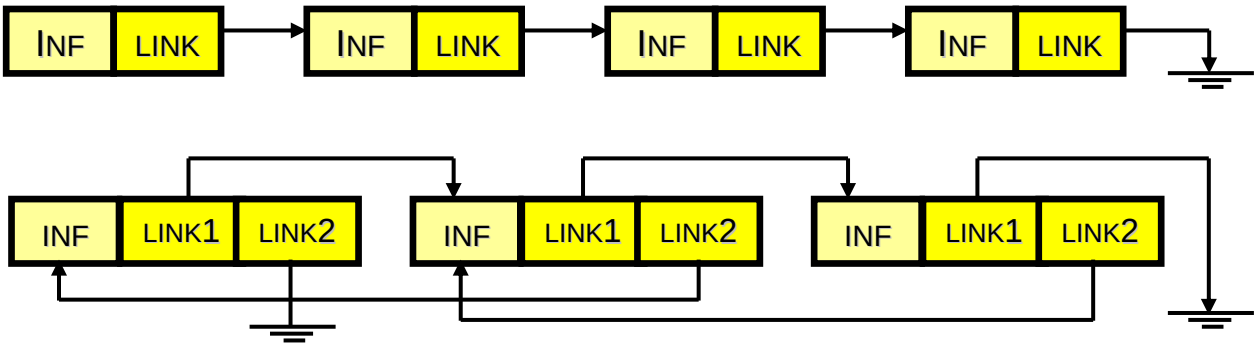
Описание элемента очереди:

```
struct Node
{
    int   d;       // поля произвольных типов
    Node *link;    // указатель на следующий элемент очереди
};
```

При работе с очередью можно использовать один указатель на начало очереди или два указателя: один на начало очереди, другой – на её конец. Использование двух указателей упрощает алгоритм добавления нового элемента.

СПИСОК

Линейные списки – данные динамической структуры, которые представляют собой совокупность линейно связанных однородных элементов, для которых разрешается добавлять элементы между любыми двумя другими и удалять любой элемент.



Реализация списка в виде двух массивов

Список можно реализовать в виде двух массивов A и B.

Пусть i – индекс элемента в списке, тогда $A[i]$ – сам элемент, $B[i]$ – индекс следующего элемента в списке A. Если k – индекс последнего элемента в списке, то $B[k] = 0$. Кроме того, существуют две переменные: nz – индекс начала списка компонент и ns – индекс начала свободных компонент.

Пример.

Список **2, 12, 17** будет реализован следующим образом:

A

2	**17**	**12**	*****	*****	*****

B

2	**0**	**1**	**5**	**6**	**0**

nz = 0 **ns = 3**

Реализация списка в виде связанных компонент

Описание элемента линейного односвязного списка аналогично описанию очереди и стека:

```
struct Node
{
    int   d;       // поля произвольных типов
    Node *link;    // указатель на следующий элемент списка
};
```

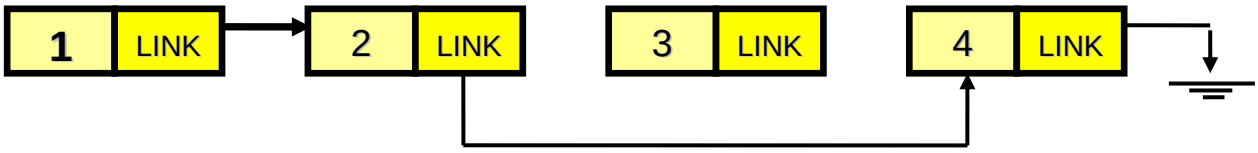
Описание элемента линейного двусвязного списка:

```
struct Node
{
    int   d;       // поля произвольных типов
    Node *next;    // указатель на следующий элемент списка
    Node *pred;    // указатель на предыдущий элемент списка
};
```

Базовые операции над списком:

- начальное формирование списка (создание первого элемента списка);
- добавление элемента в список;
- нахождение заданного элемента;
- вставка элемента в список;
- удаление элемента из списка;
- упорядочивание элементов списка;
- объединение двух списков;
- разделение списка на 2 списка;
- копирование списка;
- определение количества узлов списка;
- чтение и вывод всех элементов списка.

Удаление элемента в списке

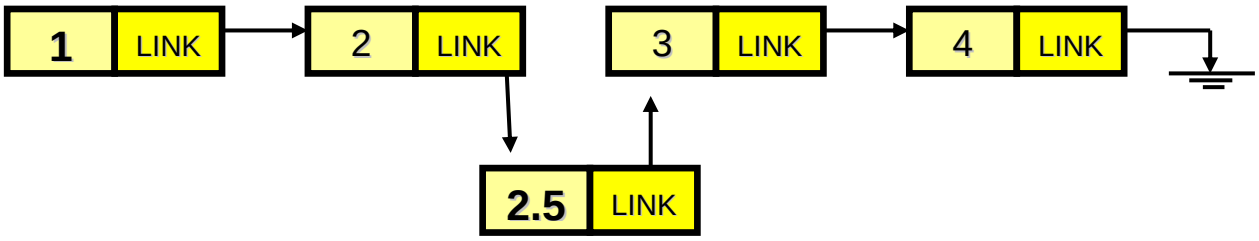


Пусть

- адрес элемента, который нужно удалить содержится в переменной **pv**.
- адрес предыдущего элемента – в переменной **ppv**.

Чтобы удалить элемент с ключом 3 (адрес **pv**), необходимо изменить связи так, чтобы элемент с ключом равным 2 (**ppv**), указывал (**ppv->link**) на элемент с ключом равным 4 (**pv->link**).

Вставка элемента в список



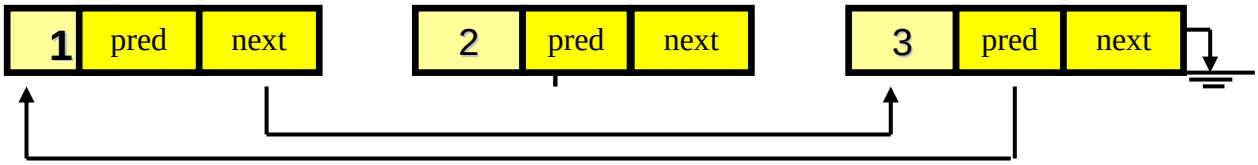
Пусть

- адрес элемента, который нужно вставить **nv**.
- адрес элемента, перед которым нужно вставить элемент содержится в переменной **pv**.
- арес предыдущего элемента – в переменной **ppv**.

Тогда, чтобы вставить элемент с ключом 2.5 (**nv**), необходимо изменить связи так, чтобы:

- элемент с ключом 2 (**ppv**), указывал (**ppv->link**) на элемент с ключом 2.5 (**nv**),
- элемент с ключом 2.5 (**nv**) указывал (**nv->link**) на элемент с ключом 3 (**pv**).

Удаление элемента в двусвязном списке



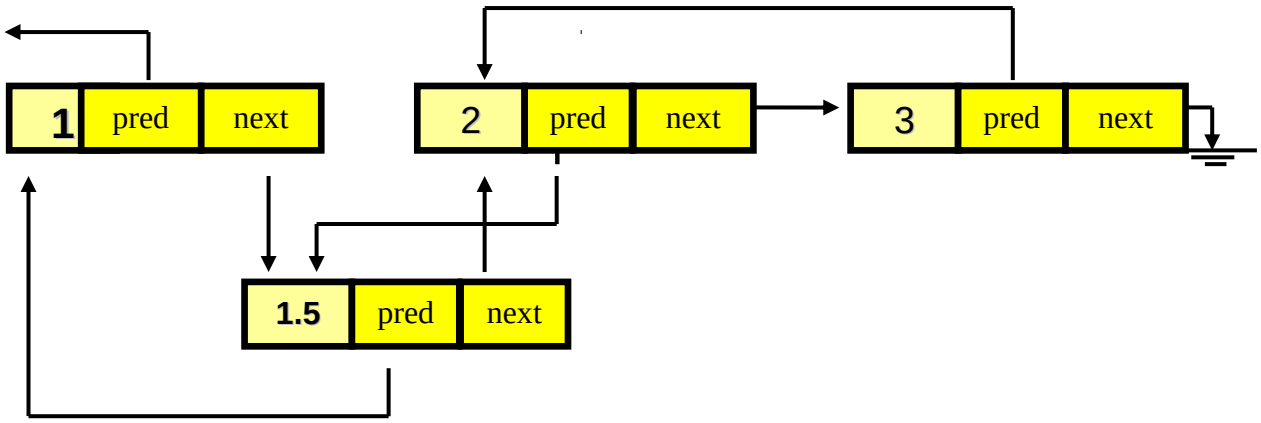
Пусть

- адрес элемента, который нужно удалить содержится в переменной **pv**.
- адрес предыдущего элемента – в переменной **ppv**.

Чтобы удалить элемент с ключом 2 (адрес **pv**), необходимо изменить связи так, чтобы

- элемент с ключом 1 (**ppv**), указывал (**ppv->next**) на элемент с ключом 3 (**pv->next**).
- элемент с ключом 3 (**pv->next)**, указывал (**pv->next->pred**) на элемент с ключом 1 (**ppv**).

Вставка элемента в двусвязный список



Пусть

- адрес элемента, который нужно вставить **nv**.
- адрес элемента, перед которым нужно вставить элемент содержится в переменной **pv**.
- адрес предыдущего элемента – в переменной **ppv**.

Чтобы вставить элемент с ключом равным 1.5 (**nv**), необходимо изменить связи так, чтобы:

- элемент с ключом 1 (**ppv**), указывал (**ppv->next**) на элемент с ключом 1.5 (**nv**),
- элемент с ключом 1.5 (**nv**), указывал (**nv->next**) на элемент с ключом 2 (**pv**),
- элемент с ключом 1.5 (**nv**) указывал (**nv->pred**) на элемент с ключом 1 (**ppv**).
- элемент с ключом 2 (**pv**) указывал (**pv->pred**) на элемент с ключом 1.5 (**nv**).

БИНАРНОЕ ДЕРЕВО

Дерево – нелинейная динамическая структура данных. В дереве каждый узел содержит несколько указателей на несколько узлов.

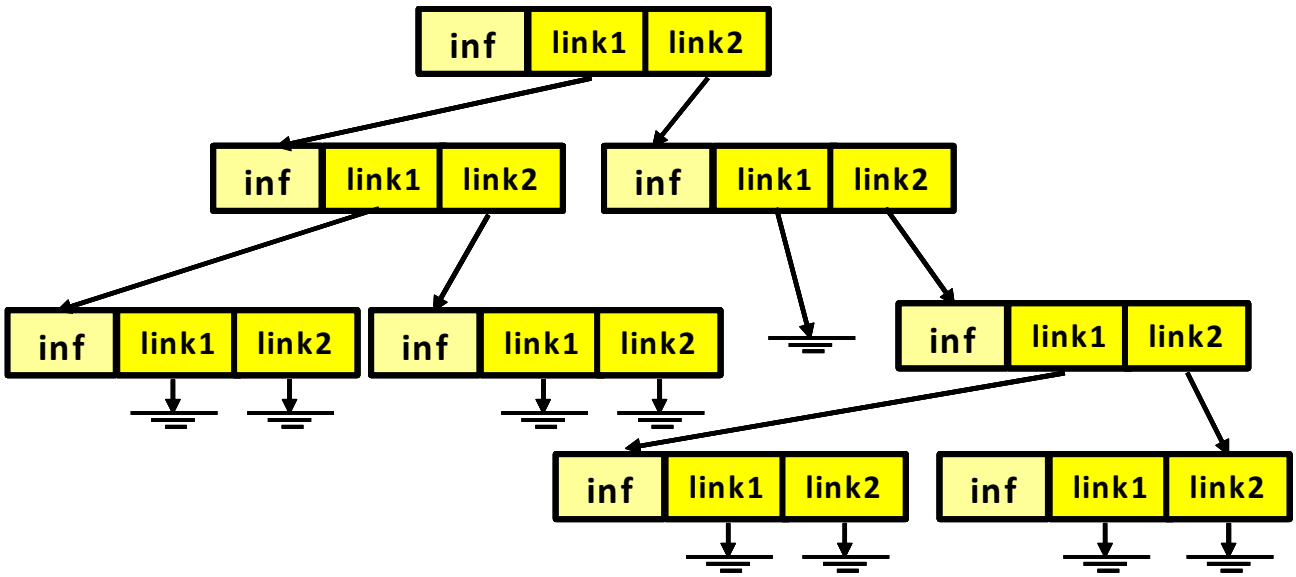
Бинарное дерево – это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных, **не более двух ссылок** на различные бинарные деревья. На каждый узел имеется ровно одна ссылка.

Начальный узел называется **корнем** дерева. У корневого узла нет входящих в него ветвей, есть только исходящие. Узлы корневого дерева, не имеющие ссылок на другие узлы, называются **листьями**.

Исходящие узлы называются **предками**, входящие – **потомками**.

Высота дерева определяется количеством уровней, на которых располагаются его узлы.

Высота двоичного дерева не более $log_2 n$.



Одним из специальных случаев бинарных деревьев являются **полные бинарные деревья**. **Полным бинарным деревом** называют упорядоченное корневое дерево, в котором:

- каждая вершина имеет не более двух потомков;
- заполнение дерева осуществляется от корня к листьям по уровням;
- заполнение уровней производится слева направо.

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева – больше, оно называется **деревом поиска**. Одинаковые ключи не допускаются.

Дерево называется **идеально сбалансированным**, если оно является бинарным деревом поиска и число вершин в его левых и правых поддеревьях отличается не более чем на 1.

Дерево называется **сбалансированным**, если оно является бинарным деревом поиска и высоты двух поддеревьев каждой из его вершин отличаются не более чем на 1. Такие деревья часто называют **АВЛ-деревьями** (по именам открывателей Г.М. Адельсон-Вельский и Е.М. Ландис).

Все идеально сбалансированные деревья являются также АВЛ-деревьями.

Реализация бинарного дерева в виде двух массивов

Бинарное дерево можно реализовать в виде двух массивов *Left* и *Right*. Если узел *j* является левым непосредственным потомком узла *i*, то *Left*[*i*] = *j*; если узел *j* является правым непосредственным потомком узла *i*, то *Right* [*i*] = *j*; если у узла *i* нет левого непосредственным потомка, то *Left*[*i*] = 0; если у узла *i* нет правого непосредственным потомка, то *Right* [*i*] = 0.

Полные бинарные деревья обычно представляют в виде одномерного массива *R*. *R*[0] – корень дерева, а для произвольного узла *I* его левый непосредственным потомок располагается в элементе *R* [2*i*], а правый – *R* [2*i*+1]. Непосредственный предок узла, находящегося в позиции *j*, расположен в позиции [*j/2*].

Реализация дерева в виде связанных компонент

Описание элемента бинарного дерева:

```
struct Node
{
    int d;        // поля произвольных типов
    Node *left;   // указатель на левое поддерево
    Node *right; // указатель на правое поддерево
};
```

Базовые операции над бинарным деревом:

- построить дерево;
- добавить новый элемент к дереву;
- выполнить поиск элемента с заданным значением ключа;
- выполнить обход элементов дерева для выполнения над ними некоторой операции;
- удаление узла с заданным значением ключа;
- уничтожение дерева и освобождение памяти.

Дерево является рекурсивной структурой данных, поскольку каждое поддерево также является деревом. Действия с такими структурами лучше всего описываются с помощью **рекурсивных алгоритмов.**

Обход дерева можно осуществить тремя способами:

1. Обход слева направо: L, K, R (сначала посещаем левое поддерево, затем корень и, наконец, правое поддерево) – **инфиксный** порядок обхода. Для дерева поиска такой порядок обхода дерева позволяет получить отсортированную последовательность.

```
void inorder(Node *top)
{
     if (top)
     {
          inorder(top->left);
          P(top);          // операция над вершиной дерева
          inorder(top->right);
     }
}
```

Или справа налево: R, K, L (сначала посещаем правое поддерево, затем корень и, наконец, левое поддерево).

```
void inorder(Node *top)
{
     if (top)
     {
          inorder(top->right);
          P(top);           // операция над вершиной дерева
          inorder(top->left);
     }
}
```

2. Обход сверху вниз K, L, R (посещаем корень до поддеревьев) – **префиксный** порядок обхода.

```
void preorder(Node *top)
{
     if (top)
     {
          P(top);            // операция над вершиной дерева
          preorder(top->left);
          preorder(top->right);
     }
}
```

3. Обход снизу вверх L, R, K (посещаем корень после поддеревьев) – **постфиксный** порядок обхода.

```
void postorder(Node *top)
{
    if (top)
    {
        postorder(top->left);
        postorder(top->right);
        P(top);            // операция над вершиной дерева
    }
}
```

ГРАФ

Граф — это совокупность непустого множества объектов и множества пар этих объектов. Объекты представляются как вершины, или узлы графа, а связи — как дуги, или рёбра.

Для разных областей применения виды графов могут различаться направленностью, ограничениями на количество связей и дополнительными данными о вершинах или рёбрах. Многие структуры, представляющие практический интерес в математике и информатике, могут быть представлены графами. Например, строение Учебника можно смоделировать при помощи ориентированного графа (орграф), в котором вершины — это статьи, а дуги (ориентированные рёбра) — гиперссылки.

Граф – G = (V, E) состоит из V - конечного непустого множества элементов (называемых **узлами или вершинами**), и E - множества неупорядоченных пар (u, v) вершин из V , называемых **ребрами**.

Ребро $s = (u, v)$ соединяет вершины u, v. Ребро s и вершина u (а также s и v) называются **инцидентными**, а вершины u и v – **смежными**.

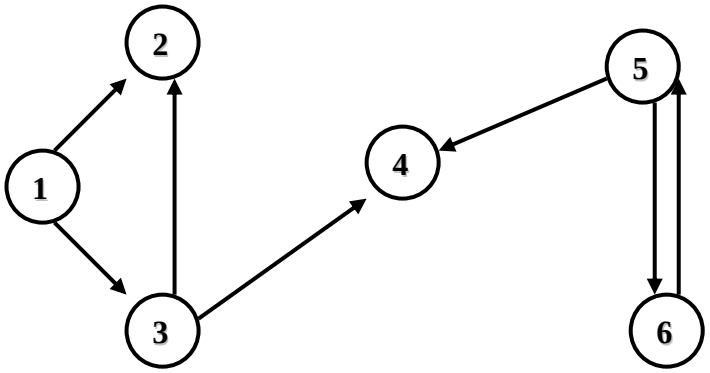
Графы бывают: **ориентированные и неориентированные**, **связные и несвязные, полные, двудольные, мультиграфы** и т.д..

Пустой граф – граф с пустым множеством вершин.

Ориентированный граф, или орграф, $G = (V, E)$ отличается от графа тем, что E – множество упорядоченных пар (u, v) вершин $u, v \in V$, называемых **дугами**. Дуга (u, v) ведет от вершины u к вершине v.

Если некоторая пара вершин соединена несколькими ребрами, то граф называется **мультиграфом**.

На практике граф изображают следующим образом (вершины – точками, ребра – линиями (или для ориентированного графа направленными линиями) :



Существует множество **алгоритмов на графах**, в основе которых лежит такой перебор вершин графа, при котором каждая вершина просматривается ровно один раз:

- поиск в глубину;
- поиск в ширину;
- топологическая сортировка.
- нахождение кратчайшего пути;
- максимальный поток;
- минимальное остовное дерево;

Способы представления графа в памяти.

1. С использованием массивов - **матрица смежности, матрица инцендентности**.
2. С использованием последовательно связанных элементов – **список ребер, список смежности.**

Определения.

Количество вершин графа называется его **порядком.**

Количество ребер графа называется его **размером.**

Если между вершинами существует ребро, то вершины называются **смежными**.

Число ребер , инцедентных некоторой вершине, называется **степенью** данной **вершины.**

Путь, соединяющий вершины u, v, – это последовательность вершин v_0, v_1, …, v_n такая, что $v_0 = u$, $v_n = v$ и для любого i вершины v_i и v_{i+1} соединены ребром.

Длина пути равна количеству ребер.

Путь **замкнут**, если $v_0 = v_n$.

Путь называется **простым**, если все его вершины различны.

Замкнутый путь, в котором все ребра различны, называется **циклом.**

Расстояние между двумя вершинами – длина кратчайшего пути, соединяющего эти вершины.

Граф называется **связным**, если для любой пары вершин существует соединяющий их путь.

При этом вершину v называют **преемником** вершины u, а u – **предшественником** вершины v.

Деревом называется связный граф без циклов.

Корневое дерево – ориентированный граф, который удовлетворяет следующим условиям:

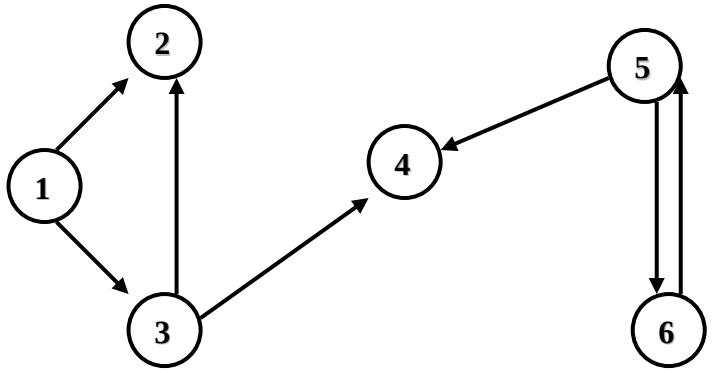
- имеется точно один узел (корневой), в который не входит ни одна дуга;
- в каждый узел, кроме корня входит ровно одна дуга;
- из корня имеется путь в каждый узел.

СПОСОБЫ ПРЕДСТАВЛЕНИЯ ГРАФА В ПАМЯТИ

1. МАТРИЦА СМЕЖНОСТИ

Классическим способом задания графа является **матрица смежности** A рамера n x n (n – количество вершин графа), где $A[i][j]$ =1, если существует ребро(дуга), идущее из вершины i в j, и $A[i][j]$ =0 в противном случае.

Матрица смежности неориентированного графа является симметричной матрицей, на главной диагонали которой стоят нули.



Для орграфа, представленного на рисунке, матрица смежности:

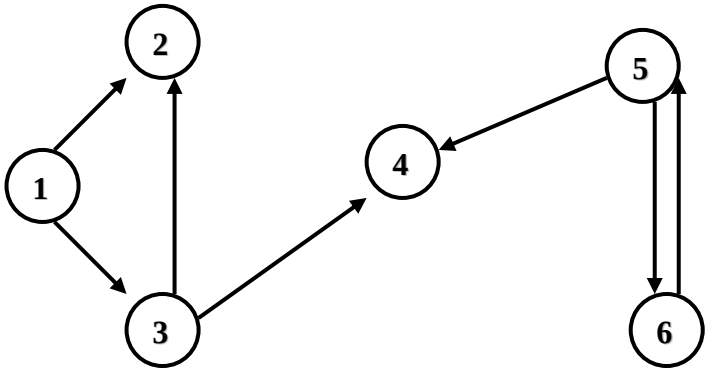
	1 2 3 4 5 6
1	0 1 1 0 0 0
2	0 0 0 0 0 0
3	0 1 0 1 0 0
4	0 0 0 0 0 0
5	0 0 0 1 0 1
6	0 0 0 0 1 0

Недостатками такого представления являются:

- объем памяти вне зависимости от количества ребер (дуг) составляет n^2;
- необходимо заранее знать количество вершин графа.

2. Матрица инцедентности

Другим традиционным способом представления графа служит **матрица инцедентности**. Эта матрица *A* рамера *n* x *m,* где *n* – количество вершин графа, а *m* – количество ребер графа. Для орграфа столбец, соответствующий дуге (*i, j*) содержит 1 в строке, соответствующей вершине *i,* содержит -1 в строке, соответствующей вершине *j*, и нули во всех остальных строках.



Для орграфа, представленного на рисунке, матрица инциндентности:

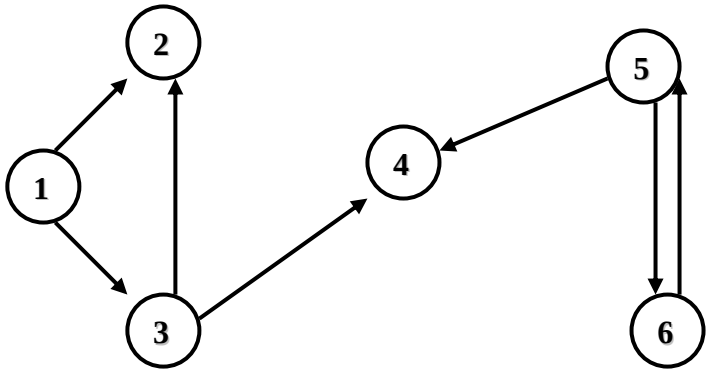
	(1,2)	(1,3)	(3,2)	(3,4)	(5,4)	(5,6)	(6,5)
1	1	1	0	0	0	0	0
2	-1	0	-1	0	0	0	0
3	0	-1	1	1	0	0	0
4	0	0	0	-1	-1	0	0
5	0	0	0	0	1	1	-1
6	0	0	0	0	0	-1	1

Недостатками такого представления являются:

- объем памяти составляет *nm* ячеек, которые в большинстве заполнены нулями;
- чтобы проверить существование дуги между вершинами, необходим перебор столбцов матрицы;
- необходимо заранее знать количество вершин графа.

3. Список пар (список ребер)

Более экономным в отношении памяти, особенно если $m << n^2$, является метод представления графа с помощью **списка пар**, соответствующих ребрам: пара i, j соответствует ребру (i, j)



Для орграфа, представленного на рисунке:

1	1	3	3	5	5	6
2	3	2	4	4	6	5

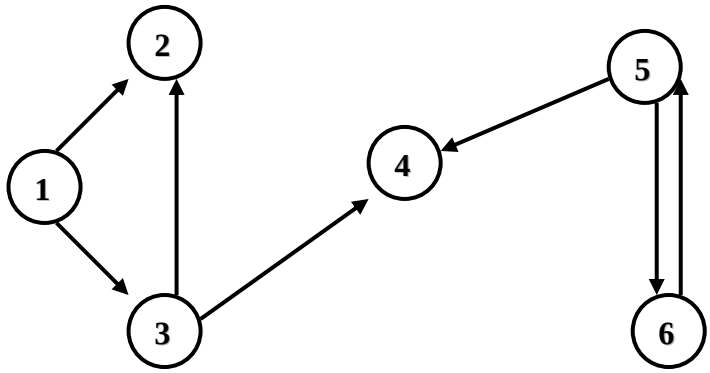
Объем памяти в этом случае составляет O(m), но необходимо большое количество шагов для получения множества вершин, к которым ведут ребра из заданной вершины.

4. Список смежности

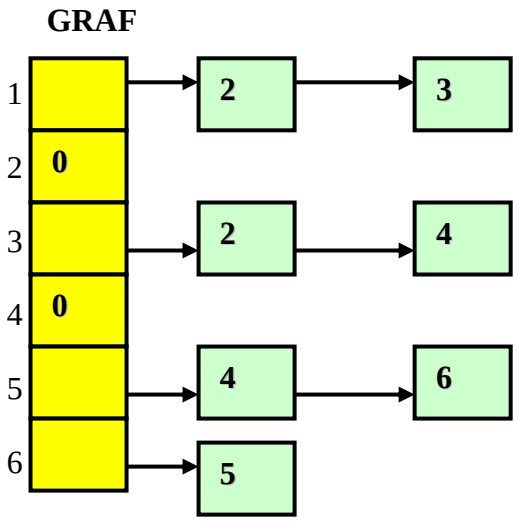
Во многих ситуациях наиболее приемлемым способом задания графа является структура данных, которую называют **списками смежности**.

```
struct SP
{
     int   key;     // номер смежной вершины
     SP   *next;    // указатель на следующую смежную вершину
};
SP *GRAF[n];
```

В этом случае **GRAF** – массив, *i*–й элемент которого является указателем на начало линейного списка вершин, смежных с *i*–ой вершиной; если из вершины *i* нет выходящих дуг, то GRAF[i] = NULL. Таким образом, граф представлен в виде *n* списков смежности, по одному для каждой вершины.



Для орграфа, представленного на рисунке:



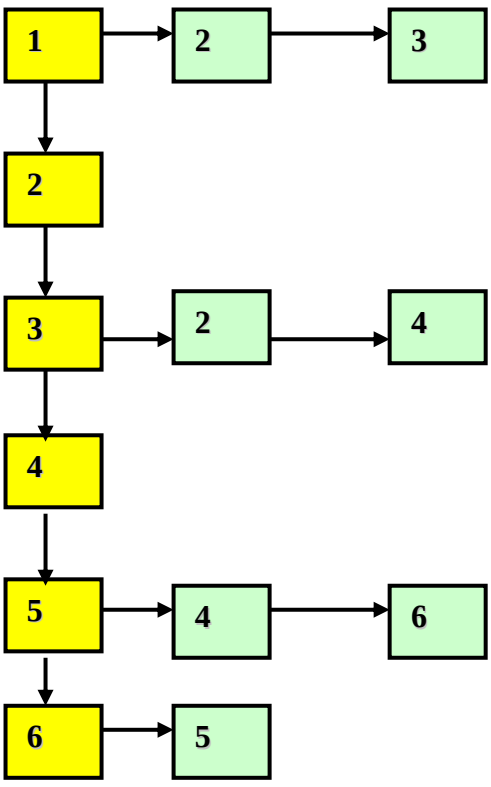
Объем памяти в этом случае O(*n*+*m*), но необходимо заранее знать количество вершин графа.

5. Список смежности

Во многих ситуациях вершины в граф добавляются в ходе выполнения программы, т.е. количество вершин графа может быть неизвестно заранее. Тогда наиболее приемлемым способом задания графа является структура данных, которую называют **списками смежности,** но граф представляется не массивом таких списков, а тоже линейным списком, в котором обязательны поля: номер вершины графа, указатель на список смежных с данной вершин, указатель на следующую вершину графа.

```
struct SP
{
    int   key;      // номер смежной вершины
    SP   *next;     // указатель на следующую смежную вершину
};

struct GRAF
{
    int   NV;       // номер вершины
    SP   *spis_sv; // указатель на линейный список
                    // смежных вершин
    GRAF *next;     // указатель на следующую вершину графа
};
```

Для орграфа, представленного на рисунке:



ПРИМЕРЫ

ПРИМЕР 1. ДИНАМИЧЕСКИЙ ОДНОМЕРНЫЙ МАССИВ

Следующая программа определяет сумму элементов массива любой размерности.

```
#include <iomanip>   // for cetw
#include <iostream>  // for cin cout
using namespace std;

double sum (int *, int);
void zapolnenie (int *, int );
void vyvod (int *, int ) ;

int main()
{
    int n = 0;

    cin >> n;                  //  n - размерность массива

    int *Array = new int[n]; // выделение памяти для массива

    zapolnenie(Array, n);    // заполнение массива

    vyvod(Array, n);       // вывод элементов массива на экран

              // передача динамического массива в функцию
    double Result = sum(Array, n);
    cout << "Result=" << Result << endl;

    delete[] Array;            // освобождение памяти
    return 0;
}

//----------------- функция заполнения массива размерности n
void zapolnenie(int *array, int n)
{
     for (int i = 0; i < n; i++)
         array[i] = i;
}
```

```
//-------------------------- функция вывода массива на экран
void vyvod(int *array,  int n)
{
    for (int i = 0; i < n; i++)
        cout << setw(4) << array[i];
    cout << endl;
}

//--функция нахождения суммы элементов массива размерности n
double sum(int *array, int n)
{
    double Sum = 0;
    for (int i = 0; i < n; i++)
        Sum += array[i];
    return Sum;
}
```

ПРИМЕР 2. ДИНАМИЧЕСКИЙ ДВУМЕРНЫЙ МАССИВ. 1 СПОСОБ

Следующая программа определяет сумму элементов двумерного массива любой размерности.

```
#include <iomanip>
#include <stdlib.h>
#include <iostream>  // for cin cout
using namespace std;

int* get_memory1 (int nrow, int ncol);
void get_memory2 (int *&array, int nrow, int ncol);
void free_memory (int *array, int nrow, int ncol);
void vvod  (int *array,  int nrow, int ncol);
void vyvod (int *array,  int nrow, int ncol);
int  sum   (int *array,  int nrow, int ncol);

int f(int i, int j, int n) // функция для определения номера
{                          // элемента [i][j] в одномерном
    return (i * n + j);    // массиве
}

int main()
{
    int nrow = 0, ncol = 0;
    cin >> nrow >> ncol;      // задаем размерности матрицы

    int *Array = get_memory1(nrow,ncol); // выделение памяти
//  get_memory2(Array,nrow,ncol);    // или так

    vvod (Array, nrow, ncol);         // заполнение массива
    vyvod (Array, nrow, ncol);        //  вывод на экран
                              //  определение суммы элементов
    int Result = sum(Array, nrow, ncol);
    cout << "Result=" << Result << endl;

    free_memory(Array, nrow, ncol);  // освобождение памяти
//  Array[0][0]=1  //  ошибка: массив, на который ссылается
                   //  указатель Array, уже не существует,
                   //  но сама переменная Array всё ещё
                   //  существует и её можно использовать
                   //  повторно
    Array = &Result;        //  например, так
    return 0;
}
```

```
// функция выделения памяти для двумерного массива
// возвращает указатель
int* get_memory1(int nrow, int ncol)
{
    int* array = new int[nrow * ncol];
    return array;
}
// функция выделения памяти для двумерного массива
//   указатель передается по ссылке
void get_memory2(int *&array, int nrow, int ncol)
{
    array = new int[nrow * ncol];
}
// функция освобождения памяти
void free_memory (int *array, int nrow, int ncol)
{
    delete[] array ;
}
// функция заполнения матрицы, например так
void vvod (int *array,int nrow,int ncol)
{
    for (int i = 0; i < nrow;  i++)
        for (int j = 0; j < ncol; j++)
            array[f(i, j, ncol)] = rand() % 20;
}
// функция вывода  матрицы на экран
void vyvod (int *array, int nrow, int ncol)
{
    for (int i = 0; i < nrow;  i++)
    {
        for (int j = 0; j < ncol;  j++)
            cout << setw(5) << array[f(i, j, ncol)];
        cout << endl;
    }
}
// функция суммирования элементов матрицы
int sum(int *array,int nrow,int ncol)
{
    int Sum = 0;
    for (int i = 0; i < nrow;  i++)
        for (int j = 0; j < ncol;  j++)
            Sum += array[f(i, j, ncol)]; //накопление суммы
    return (Sum);
}
```

ПРИМЕР 3. ДИНАМИЧЕСКИЙ ДВУМЕРНЫЙ МАССИВ. 2 СПОСОБ

Следующая программа определяет сумму элементов двумерного массива любой размерности.

```
#include <iomanip>
#include <stdlib.h>
#include <iostream>  // for cin cout
using namespace std;

int** get_memory1 (int nrow, int ncol);
void  get_memory2 (int **&array, int nrow, int ncol);
void  free_memory  (int **array, int nrow, int ncol);
void  vvod  (int **array, int nrow, int ncol);
void  vyvod (int **array, int nrow, int ncol);
int   sum   (int **array, int nrow, int ncol);

int main()
{
    int nrow = 0, ncol = 0;
    cin >> nrow >> ncol;     // задаем размерности матрицы

    int **Array = get_memory1(nrow, ncol);// выделение
                                          // памяти
//  get_memory2(Array, nrow, ncol);     // или так

    vvod (Array, nrow, ncol);       // заполнение массива
    vyvod (Array, nrow, ncol);      //  вывод на экран
    int Result = sum(Array, nrow, ncol);// определение суммы
    cout << "Result=" << Result << endl;

    free_memory(Array, nrow, ncol);  // освобождение памяти

    return 0;
}
```

```
// функция выделения памяти для двумерного массива
// возвращает указатель
int** get_memory1(int nrow, int ncol)
{    int** array = new int*[nrow];
     for (int i = 0; i < nrow; i++)
         array[i] = new int[ncol];
     return(array);
}
// функция выделения памяти для двумерного массива
// указатель передается по ссылке
void get_memory2(int **&array, int nrow, int ncol)
{
     array = new int*[nrow];
     for (int i = 0; i < nrow; i++)
         array[i] = new int[ncol];
}
// функция освобождения памяти
void free_memory (int **array, int nrow, int ncol)
{    for (int i = 0; i < nrow; i++)
         delete array[i];
     delete[] array;
}
// функция заполнения матрицы, например так
void vvod (int **array, int nrow, int ncol)
{    for (int i = 0; i < nrow;  i++)
         for (int j = 0; j < ncol; j++)
             array[i][j] = rand() % 20;
}
// функция вывода  матрицы на экран
void vyvod (int **array, int nrow, int ncol)
{
     for (int i = 0; i < nrow;  i++)
     {  for (int j = 0; j < ncol; j++)
           cout << setw(5) << array[i][j];
         cout << endl;
     }
}
// функция суммирования элементов матрицы
int sum(int **array, int nrow, int ncol)
{    int Sum = 0;
     for (int i = 0; i < nrow; i++)
         for (int j = 0; j < ncol;  j++)
             Sum += array[i][j];    //накопление суммы
     return Sum;
}
```

ПРИМЕР 4. БАЗОВЫЕ ОПЕРАЦИИ СО СТЕКОМ

Программа, реализации базовых операций со стеком:

1. создаем стек из 5 элементов: 1,2,3,4,5.
2. вывод на экран элементов стека (5,4,3,2,1).
3. удаление всех элементов стека.

```
#include <iostream>  // for cin cout
using namespace std;

struct Node                    // описание элемента стека
{
    int d;
    Node* p;          // указатель на следующий элемент стека
};

void  push1 (Node **top, int d);
void  push2 (Node *&top, int d);
void  vyvod_stack (Node *top);
int   pop1  (Node **top);
int   pop2  (Node *&top);

void main( void )
{
    Node *top = NULL;         // top – указатель вершины стека

    for (int i = 1; i < 6; i++)
       push1(&top, i);        // добавляем элементы в стек

//  for (int i = 1; i < 6; i++)
//         push2(top, i);    // или так

    vyvod_stack (top);  // выводим содержимое стека на экран

    while (top)
       cout << pop1(&top) << "  "; // выталкиваем элементы
                                   // стека

//   while (top)
//       cout << pop2(top) << "  ";    // или так

    vyvod_stack(top);           // проверяем, что стек пуст
}
```

```
// функция занесения элемента в вершину стека  вариант 1
void push1(Node **top, int d)
{                 //    top - указатель начала стека
     Node *pv = new Node; // выделение памяти для элемента
                          // стека
     pv->d = d;
     pv->p = *top;  // связываем новый элемент с предыдущим
     *top = pv;     // меняем адрес начала стека
}

// функция занесения элемента в вершину стека вариант 2
void push2(Node *&top, int d)
{                   //    top - указатель начала стека
     Node *pv = new Node; // выделение памяти для элемента
                          // стека
     pv->d = d;
     pv->p = top;     // связываем новый элемент с предыдущим
     top = pv;        // меняем адрес начала стека
}

// функция удаления элемента из вершины стека вариант 1
int pop1(Node **top)
{                     //    top - указатель начала стека
     int temp = (*top)->d;
     Node* pv = *top;
     *top = (*top)->p;       // меняем адрес начала стека
     delete pv;              // освобождение памяти
     return temp;
}

// функция удаления элемента из вершины стека вариант 2
int pop2(Node *&top)
{                     //    top - указатель начала стека
     int temp = top->d;
     Node* pv = top;
     top = top->p;           // меняем адрес начала стека
     delete pv;              // освобождение памяти
     return temp;
}
```

```
// функция  просмотра и вывода элементов стека на экран
void vyvod_stack(Node *top)
{
     while (top)
     {
          cout << top->d << ' ';
          top = top->p; //переход к следующему элементу стека
     }
     cout << "\n";
}
```

ПРИМЕР 5. БАЗОВЫЕ ОПЕРАЦИИ С ОЧЕРЕДЬЮ

Программа, реализации базовых операций с очередью:

1. создаём первый элемент очереди (1)
2. добавляем в очередь из 4 элемента: 2,3,4,5.
3. вывод на экран элементов очереди (1,2,3,4,5).
4. удаление всех элементов очереди.
5. ещё один вариант создания очереди (рекурсивный).
6. вывод на экран элементов очереди (1,2,3,4,5).

```
#include <iostream>  // for cin cout
using namespace std;

struct Node  // описание элемента очереди аналогичен стеку
{    int d;
     Node *p;   // указатель на следующий элемент очереди
};

Node*  first1  (int d);
void   first2  (Node *&, int d);
void   first3  (Node ** , int d);
void   add1    (Node ** end, int d);
void   add2    (Node *&end, int d);
void   push    (Node *&top, int d);
void   vyvod_ochered (Node *top);
int    del1    (Node **top);
int    del2    (Node *&top);

void main( void )
{
     Node *top, *end;
      // top - указатель начала очереди, end - конец очереди

     top = first1(1);      // заносим первый элемент в очередь
     //  first2(top,1);  //  или так
     //  first3(&top,1); //   или так

     end = top;           // последний есть первый
     for (int i = 2; i < 6; i++)
        add1(&end, i);  // добавляем элементы в конец очереди

//   for (int i = 2; i < 6; i++)
//      add2(end, i);                          // или так

     vyvod_ochered (top); // выводим содержимое очереди
                          // на экран
```

```
                // удаляем элементы очереди сначала
    while(top)
        cout << del1(&top) << "  ";
//   while(top)
//       cout << del2(top) << "  ";                // или так

    vyvod_ochered(top); // проверяем, что очередь пуста

            // другой способ создания очереди (рекурсивный)
    for (int i = 1; i < 6; i++)
       push(top, i);

    vyvod_ochered(top); // выводим содержимое очереди
                        // на экран
}

//функция занесения первого элемента в очередь способ 1
//функция аналогична функции для стека
Node* first1(int d)
 {
    Node *pv = new Node;           //  выделение памяти
    pv->d = d;
    pv->p = NULL;
    return pv;
}

// функция занесения первого элемента в очередь способ 2
// функция аналогична функции для стека
void first2(Node *&top, int d)
{
    top = new Node;         //  выделение памяти
    top->d = d;
    top->p = NULL;
}

// функция занесения первого элемента в очередь способ 3
// функция аналогична функции для стека
void first3(Node **top, int d)
{
    *top = new Node;          //  выделение памяти
    (*top)->d = d;
    (*top)->p = NULL;
}
```

```
// функция занесения элемента в конец очереди  вариант 1
// end - адрес последнего элемента очереди
void add1(Node **end, int d)
{
     Node *pv = new Node;            //  выделение памяти
     pv->d = d;
     pv->p = NULL;
                  //связываем с предыдущим элементом очереди
     (*end)->p = pv;
     (*end) = pv;        //  меняем адрес последнего элемента
}

// функция занесения элемента в конец очереди вариант 2
// end - адрес последнего элемента очереди
void add2(Node *&end,int d)
{
     Node *pv = new Node;       //  выделение памяти
     pv->d = d;
     pv->p = NULL;
            // связываем с предыдущим элементом очереди
     end->p = pv;
     end = pv;            //  меняем адрес последнего элемента
}

//функция занесения элемента в конец очереди  вариант 3
// рекурсивный
void push(Node *&top, int d)
{
     if (!top)
     {
          top = new Node;     //  выделение памяти
          top->d = d;
          top->p = NULL;
     }
     else
         push(top->p,d);     // рекурсия
}
```

```
// функция удаления элемента из начала очереди вариант 1
int del1(Node **top)
{                       //    top - указатель начала очереди
     int temp;
     Node *pv;       //  функция аналогична функции для стека
     temp = (*top)->d;
     pv = *top;
     *top = (*top)->p;     // меняем адрес начала очереди
     delete pv;            // освобождение памяти
     return(temp);
}

// функция удаления элемента из начала очереди вариант 2
int del2(Node *&top)
{                  //    top - указатель начала очереди
     Node* pv;  //  функция аналогична функции для стека
     int temp;
     temp = top->d;
     pv = top;
     top = top->p;       // меняем адрес начала очереди
     delete pv;          // освобождение памяти
     return(temp);
}

// функция  просмотра и вывода элементов очереди на экран
// функция аналогична функции для стека
void print_ochered(Node *top)
{
     while (top)
     {
          cout << top->d << ' ';
          top = top->p;
     }
     cout << "\n";
}
```

ПРИМЕР 6. БАЗОВЫЕ ОПЕРАЦИИ С ЛИНЕЙНЫМ ОДНОСВЯЗНЫМ СПИСКОМ

Программа, реализации базовых операций с линейным односвязным списком:

1. создаем первый элемент списка (1)
2. добавляем в список 4 элемента: 6,5,4,3 (элементы в список добавляются по возрастанию ключей).
3. вывод на экран элементов списка (1,2,3,4,5).
4. удаление элемента списка со значением 3.
5. вывод на экран элементов списка (1,2,4,5).

```
#include <iostream>       // for cin cout
using namespace std;

struct Node     // описание элемента списка
{               // аналогичен стеку и очереди
    int d;
    Node *p;    // указатель на следующий элемент списка
};

void   find_eq(Node *, int, Node *&);
void   find_gt(Node *, int, Node *&);
void   add(Node *&, int);
void   vyvod(Node *);
void   del(Node *&, int);

void main(void)
{
    Node *top = NULL;
    add(top, 1);        // заносим первый элемент в список

    for (int i = 5; i > 1; i--)
        add(top, i); // добавляем элементы в список
                     // ключи вводятся по убыванию,
                     // но в список заносятся по возрастанию
    vyvod(top);  //   выводим содержимое списка на экран
    del(top, 3); //   удаляем элемент списка с ключом = 3
    vyvod(top);  //   выводим содержимое списка на экран
}
```

```
// функция нахождения элемента с ключом =key
// top - адрес начала списка
// ppv - адрес элемента списка, указывающего на найденный
void find_eq(Node *top, int key, Node *&ppv)
{
     Node *pv = top;
     ppv = top;
     //поиск элемента с ключом = key
     while (pv && pv->d != key)
     {
          ppv = pv;    // запоминаем адрес предыдущего элемента
          pv = pv->p;
     }
}

// функция нахождения элемента с ключом >key
// top - адрес начала списка
// ppv - адрес элемента списка, указывающего на найденный
void find_gt(Node *top, int key, Node *&ppv)
{
     Node *pv = top;
     ppv = top;
     //поиск элемента с ключом > key
     while (pv && pv->d <= key)
     {
          ppv = pv;    // запоминаем адрес предыдущего элемента
          pv = pv->p;
     }
}

//функция занесения элемента с ключом key в список
// по возрастанию ключей
// top - указатель начала списка
void add(Node *&top, int key)
{
     Node* ppv = NULL;
     Node* nv = new Node;  // образуем новый элемент списка
     nv->d = key;
     nv->p = NULL;
     if (!top)             // если список пуст
          top = nv;        // первый элемент списка
     else
     {    if (top->d >= key)    // вставляем в начало списка
          {
               nv->p = top;
```

```
            top = nv;
        }
        else
        {
            find_gt(top, key, ppv);
            // поиск по ключу места для нового элемента
            nv->p = ppv->p;
        // вставляем между элементами с адресами ppv и pv
            ppv->p = nv;
        }
    }
}

// функция просмотра и вывода элементов списка на экран
void vyvod(Node *top)
{
    while (top)
    {
        cout << top->d << ' ';
        top = top->p;
    }
    cout << "\n";
}

// функция удаления элемента с ключом key
// top - указатель начала списка
void del(Node *&top, int key)
{
    Node *pv, *ppv;
    // поиск элемента с ключом = key
    find_eq(top, key, ppv);
    pv = ppv->p;
    if (pv)                      // если нашли
    {
        if (top->d == key)    // удаляем первый элемент
            top = top->p;
        else
            ppv->p = pv->p;   // удаляем элемент из середины
                              // или конца списка
        delete pv;            // освобождение памяти
    }
}
```

ПРИМЕР 7. БАЗОВЫЕ ОПЕРАЦИИ С ЛИНЕЙНЫМ ДВУСВЯЗНЫМ СПИСКОМ

Программа, реализации базовых операций с линейным двусвязным списком:

1. создаем первый элемент списка (1)
2. добавляем в список 4 элемента: 6,5,4,3 (элементы в список добавляются по возрастанию ключей).
3. вывод на экран элементов списка (1,2,3,4,5).
4. вывод на экран элементов списка в обратном порядке (5,4,3,2,1).
5. удаление элемента списка со значением 3.
6. вывод на экран элементов списка (1,2,4,5).
7. вывод на экран элементов списка в обратном порядке (5,4,2,1).

```
#include <iostream>       // for cin cout
using namespace std;

struct Node
{
    int d;
    Node *next;  // указатель на следующий элемент списка
    Node *pred; // указатель на предыдущий элемент списка
};
void   find  (Node *, int , Node *&, Node *&);
void   find  (Node *, int , Node *&, Node *& , char );
void   add   (Node * &, Node *&, int );
void   vyvod1(Node *);
void   vyvod2(Node *);
void   del   (Node * &,Node *&, int );

void main( void )
{   Node *top = NULL, *end = NULL;
    add(top,end,1);   // заносим первый элемент в список
    vyvod1(top);      //   выводим содержимое списка на экран
    vyvod2(end);     //   выводим содержимое списка на экран
                     // в обратном порядке
    for (int i = 5; i > 1; i--)
      add(top, end, i);     //   добавляем элементы в список
                           //   ключи вводятся по убыванию,
    // но  в список заносятся по возрастанию, начиная с top;
    vyvod1(top);     //   выводим содержимое списка на экран
    vyvod2(end);     //   выводим содержимое списка на экран
                      // в обратном порядке
    del(top, end, 3); // удаляем элемент списка с ключом = 3
    vyvod1(top);     //   выводим содержимое списка на экран
    vyvod2(end);       //выводим содержимое списка на экран в
                       // обратном порядке
}
```

```
// аналогична односвязному списку
// функция нахождения элемента с ключом key
// top - адрес начала списка
// ppv - адрес элемента списка, указывающего на найденный
void find(Node *top, int key, Node *&ppv)
{
     Node *pv = top;
     ppv = top;
               // поиск элемента с ключом = key
     while (pv && pv->d != key )
     {
          ppv = pv;  // запоминаем адрес предыдущего элемента
          pv = pv->next;
     }
}

// аналогична односвязному списку
// перегруженная функция нахождения элемента с ключом >key
//  top - адрес начала списка
// ppv - адрес элемента списка, указывающего на найденный
void find(Node *top, int key, Node *&ppv, char )
{
     Node *pv = top;
     ppv = top;
                    // поиск элемента с ключом > key
     while (pv && pv->d <= key )
     {
          ppv = pv;   // запоминаем адрес предыдущего элемента
          pv = pv->next;
     }
}
```

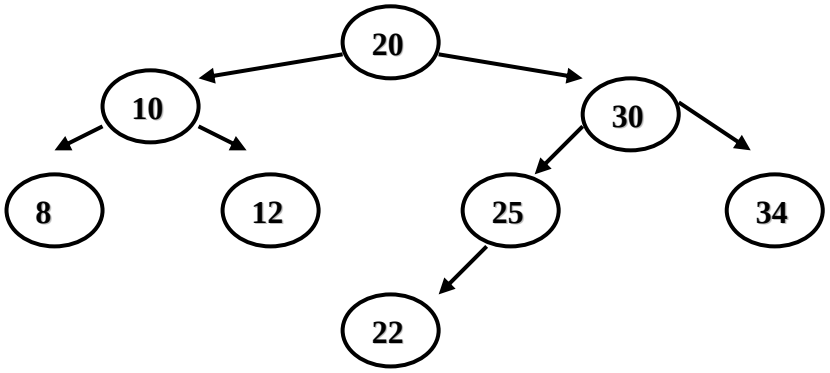
```
// функция  просмотра и вывода элементов списка на экран
// аналогична односвязному списку
void vyvod1(Node *top)
{
     while (top)
     {
          cout << top->d << ' ';
          top = top->next;
     }
     cout << "\n";
}

// функция  просмотра и вывода элементов списка на экран в
// обратном порядке
void vyvod2(Node* end)
{
     while (end)
     {
          cout << end->d << ' ';
          end = end->pred;
     }
     cout << "\n";
}
```

```
// функция занесения элемента с ключом k в список
// по возрастанию ключей элементов
//  top  - указатель начала списка
//  end - конец списка
void add(Node *&top, Node *&end, int key)
{
    Node *pv = NULL,*ppv = NULL;

    Node* nv = new Node;     // образуем новый элемент списка
    nv->d = key;
    nv->next = NULL;
    nv->pred = NULL;

    if (!top)
    {   top = nv;             // первый элемент списка
        end = nv;
    }
    else
    {
        if (top->d >= key)    // вставляем в начало списка
        {
            nv->next = top;
            top->pred = nv;
            top = nv;
        }
        else
        {   find(top, key, ppv, '<');
            pv = ppv->next;
            // поиск по ключу места для нового элемента
            nv->next = pv;
          // вставляем между элементами с адресами ppv и pv
            nv->pred = ppv;
             if (pv)
                 pv->pred = nv;
             else
                 end = nv;
             ppv->next = nv;
       }
    }
}
```

```
// функция удаления элемента с ключом key
// top - указатель начала списка
// end – конец списка
void del(Node *&top, Node *&end, int key)
{
     Node *pv, *ppv;
     if (top->d == key)        // удаляем первый элемент
     {
          pv = top;
          top = top->next;
          if (top)
              top->pred = NULL;
          else
              end = NULL;
          delete pv;            // освобождение памяти
          return;
     }

     find(top, key, ppv);      // поиск по ключу
     pv = ppv->next;
     if (!pv)                  // если не нашли
          return;              // то нечего тут больше делать
     // нашли - удаляем элемент из середины или конца списка
     ppv->next = pv->next;
     if (pv->next)
         pv->next->pred = ppv;
     else
         end = ppv;            // если последний элемент
     delete pv;                // освобождение памяти
}
```

ПРИМЕР 8. БАЗОВЫЕ ОПЕРАЦИИ С БИНАРНЫМ ДЕРЕВОМ ПОИСКА

Программа, реализации базовых операций с бинарным деревом поиска по следующей последовательности чисел: 20 10 8 12 30 25 34 22



1. создаем первый элемент дерева (20);
2. добавляем в дерево остальные элементы: 10, 8, 12, 30, 25, 34, 22; (элементы в дерево добавляются по правилу: меньше – налево, больше – направо);
3. вывод на экран элементов дерева в отсортированном порядке (8,12,10,22,25,30,34);
4. вывод на экран элементов дерева по уровням;
5. получение нового дерева: копии исходного
6. удаление дерева.

```
#include <iostream>  // for cin cout
using namespace std;

struct TREE
{
    int  d;             // число
    TREE *left;         //указатель на левое поддерево
    TREE *right;        //указатель на правое поддерево
};

void    first (TREE *&,int );
TREE*   search (TREE *, int );
void    add (TREE *&, int );
void    print_tree (TREE * );
void    print_tree_level (TREE *, int);
void    del_tree  (TREE *&top);
void    copy_tree (TREE * top, TREE *&top1);
```

```
void main()
{
     int b[ ] = {20, 10, 8, 12, 30, 25, 34, 22};
     TREE *top = NULL;

     for (int i = 0; i < 8; i++)
          add (top, b[i]);  // добавление узлов дерева

     cout << " tree 1" << endl;
     print_tree (top);      // вывод элементов дерева на экран
     cout << endl << endl;
     print_tree_level(top, 0);
            // вывод элементов дерева на экран по уровням

     TREE *top1=NULL;
     cout << " tree 2" << endl;
     copy_tree(top,top1);  // копия дерева
     print_tree (top1);    // вывод нового дерева на экран
     cout << endl << endl;
     print_tree_level(top1, 0);

     del_tree(top);        // удаление дерева
     cout << " tree =?";
     print_tree (top);     // проверка того, что дерево пусто
     cout << endl;
}
// поиск места для нового узла
// возвращаем адрес узла, после которого нужно добавить
// новый элемент
TREE* search(TREE *top, int d)
{
     TREE *pv = top, *ppv=top;
     while (pv)
     {
          ppv = pv;
          if (d < pv->d)
               pv = pv->left;
          else
               pv = pv->right;
     }
     return ppv;
}
```

```
// поиск места для нового узла  рекурсивный вариант
// возвращаем адрес узла, после которого нужно добавить
// новый элемент
TREE* find(TREE *top, int d)
{
     if (!top)
          return NULL;
     TREE *p;
     if (d < top->d )
          p = find(top->left, d);
     else
          p = find(top->right, d);
     return p ? p : top;
}

// включение нового узла в дерево
void add (TREE *&top, int d)
{
     TREE *pnew = new TREE;  // создаем новый узел
     pnew->d = d;
     pnew->left= NULL;
     pnew->right = NULL;
     if (!top)//  формирование первого узла дерева
          top = pnew;
     else
     {
          TREE *ppv;
          ppv = search(top, d); // поиск места для нового узла
          // ppv = find(top, d); // или тоже самое, но
                                 // рекурсивно
          if (d < ppv->d)
               ppv->left = pnew;
                    //присоединение к левому поддереву предка
          else
               ppv->right = pnew;
                    //присоединение к правому поддереву предка
     }
}
```

```
// обход дерева и вывод значений в отсортированном порядке
void print_tree (TREE *top)
{
     if (top)
     {    print_tree (top->left); //обход левого поддерева
          cout << top->d << "  ";
          print_tree(top->right);//обход правого поддерева
     }
}

// обход дерева и вывод значений  по уровням
void print_tree_level(TREE * top, int level)
{
     if (top)
     {    print_tree_level (top->left, level + 1);
                              //обход левого поддерева
          for (int i = 0; i < level; i++)
              cout << "    ";
          cout << top->d << endl;
          print_tree_level(top->right, level + 1);
                               //обход правого поддерева
     }
}

// удаление дерева
void del_tree (TREE *&top)
{
     if (top)
     {
          del_tree (top->left); //обход левого поддерева
          del_tree(top->right);//обход правого поддерева
          delete top;             // освобождение памяти
          top = NULL;
     }
}
// копия дерева
void copy_tree (TREE * top1, TREE * &top2)
{
     if (top1)
     {
      add(top2,top1->d);    //добавление узла в новое дерево
      copy_tree (top1->left,top2); //обход левого поддерева
     copy_tree(top1->right,top2); //обход правого поддерева
     }
}
```