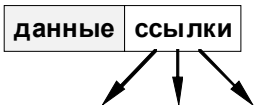
Глава IV.
Динамические структуры данных

1. Списки

Динамические структуры данных

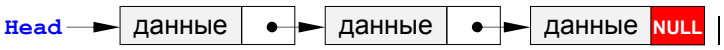
Часто в серьезных программах надо использовать данные, размер и структура которых должны **`меняться`** в процессе работы. Динамические массивы здесь не выручают, поскольку заранее нельзя сказать, сколько памяти надо выделить – это выясняется только в процессе работы. Например, надо проанализировать текст и определить, какие слова и в каком количество в нем встречаются, причем эти слова нужно расставить по алфавиту.



В таких случаях применяют данные особой структуры, которые представляют собой отдельные элементы, связанные с помощью **ссылок**. Каждый элемент (**узел**) состоит из двух областей памяти: **поля данных** и **ссылок**. Ссылки – это адреса других узлов этого же типа, с которыми данный элемент логически связан. В языке Си для организации ссылок используются переменные-указатели. При добавлении нового узла в такую структуру выделяется новый блок памяти и (с помощью ссылок) устанавливаются связи этого элемента с уже существующими. Для обозначения конечного элемента в цепи используются **нулевые ссылки** (**`NULL`**).

Линейный список

В простейшем случае каждый узел содержит всего одну ссылку. Для определенности будем считать, что решается задача частотного анализа текста – определения всех слов, встречающихся в тексте и их количества. В этом случае область данных элемента включает строку (длиной не более 40 символов) и целое число.



Каждый элемент содержит также ссылку на **следующий** за ним элемент. У последнего в списке элемента поле ссылки содержит **`NULL`**. Чтобы не потерять список, мы должны где-то (в переменной) хранить адрес его первого узла – он называется «головой» списка. В программе надо объявить два новых типа данных – узел списка **`Node`** и указатель на него **`PNode`**. Узел представляет собой структуру, которая содержит три поля - строку, целое число и указатель на такой же узел. Правилами языка Си допускается объявление

```
struct Node {
      char  word[40];  // область данных
      int   count;
      Node  *next;     // ссылка на следующий узел
      };
typedef Node *PNode;  // тип данных: указатель на узел
```

В дальнейшем мы будем считать, что указатель **`Head`** указывает на начало списка, то есть, объявлен в виде

```
PNode Head = NULL;
```

Первая буква «P» в названии типа **`PNode`** происходит от слова *pointer* – указатель (англ.) В начале работы в списке нет ни одного элемента, поэтому в указатель **`Head`** записывается нулевой адрес **`NULL`**.

Создание элемента списка

Для того, чтобы добавить узел к списку, необходимо создать его, то есть выделить память под узел и запомнить адрес выделенного блока. Будем считать, что надо добавить к списку узел, соответствующий новому слову, которое записано в переменной **NewWord**. Составим функцию, которая создает новый узел в памяти и возвращает его адрес. Обратите внимание, что при записи данных в узел используется обращение к полям структуры через указатель.

```
PNode CreateNode ( char NewWord[] )
{
   PNode NewNode = new Node; // указатель на новый узел
   strcpy(NewNode->word, NewWord); // записать слово
   NewNode->count = 1;             // счетчик слов = 1
   NewNode->next = NULL;           // следующего узла нет
   return NewNode;  // результат функции – адрес узла
}
```

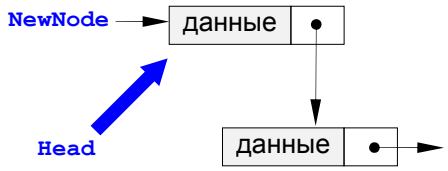
После этого узел надо добавить к списку (в начало, в конец или в середину).

Добавление узла

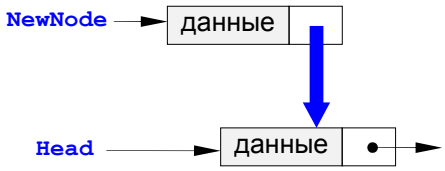
Добавление узла в начало списка

При добавлении нового узла **NewNode** в начало списка надо 1) установить ссылку узла **NewNode** на голову существующего списка и 2) установить голову списка на новый узел.

1)



2)



По такой схеме работает процедура **AddFirst**. Предполагается, что адрес начала списка хранится в **Head**. Важно, что здесь и далее адрес начала списка передается *по ссылке*, так как при добавлении нового узла он изменяется внутри процедуры.

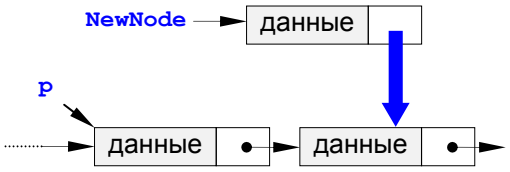
```
void AddFirst (PNode &Head, PNode NewNode)
{
  NewNode->next = Head;
  Head = NewNode;
}
```

Добавление узла после заданного

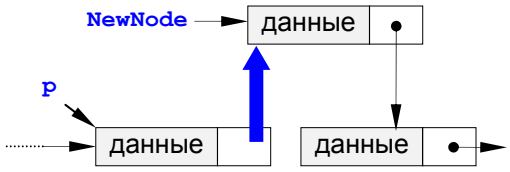
Дан адрес **NewNode** нового узла и адрес **p** одного из существующих узлов в списке. Требуется вставить в список новый узел после узла с адресом **p**. Эта операция выполняется в два этапа:

1) установить ссылку нового узла на узел, следующий за данным;
2) установить ссылку данного узла **p** на **NewNode**.

1)



2)



Последовательность операций менять нельзя, потому что если сначала поменять ссылку у узла **p**, будет потерян адрес следующего узла.

```
void AddAfter (PNode p, PNode NewNode)
{
   NewNode->next = p->next;
   p->next = NewNode;
}
```

Добавление узла перед заданным

Эта схема добавления самая сложная. Проблема заключается в том, что в простейшем линейном списке (он называется *односвязным*, потому что связи направлены только в одну сторону) для того, чтобы получить адрес предыдущего узла, нужно пройти весь список сначала. Задача сведется либо к вставке узла в начало списка (если заданный узел – первый), либо к вставке после заданного узла.

```
void AddBefore(PNode &Head, PNode p, PNode NewNode)
{
PNode q = Head;

if (Head == p) {
   AddFirst(Head, NewNode); // вставка перед первым узлом
   return;
   }
while (q && q->next!=p)  // ищем узел, за которым следует p
  q = q->next;
if ( q )                 // если нашли такой узел,
  AddAfter(q, NewNode);  //       добавить новый после него
}
```

Такая процедура обеспечивает «защиту от дурака»: если задан узел, не присутствующий в списке, то в конце цикла указатель **q** равен **NULL** и ничего не происходит.

Существует еще один интересный прием: если надо вставить новый узел **NewNode до** заданного узла **p**, вставляют узел **после** этого узла, а потом выполняется обмен данными между узлами **NewNode** и **p**. Таким образом, по адресу **p** в самом деле будет расположен узел с новыми данными, а по адресу **NewNode** – с теми данными, которые были в узле **p**, то есть мы решили задачу. Этот прием не сработает, если адрес нового узла **NewNode** запоминается где-то в программе и потом используется, поскольку по этому адресу будут находиться другие данные.

Добавление узла в конец списка

Для решения задачи надо сначала найти последний узел, у которого ссылка равна **NULL**, а затем воспользоваться процедурой вставки после заданного узла. Отдельно надо обработать случай, когда список пуст.

```
void AddLast(PNode &Head, PNode NewNode)
{
PNode q = Head;
if (Head == NULL) {          // если список пуст,
   AddFirst(Head, NewNode); // вставляем первый элемент
   return;
   }
while (q->next) q = q->next; // ищем последний элемент
AddAfter(q, NewNode);
}
```

Проход по списку

Для того, чтобы пройти весь список и сделать что-либо с каждым его элементом, надо начать с головы и, используя указатель **next**, продвигаться к следующему узлу.

```
PNode p = Head;          // начали с головы списка
while ( p != NULL ) {    // пока не дошли до конца
     // делаем что-нибудь с узлом p
   p = p->next;          // переходим к следующему узлу
   }
```

Поиск узла в списке

Часто требуется найти в списке нужный элемент (его адрес или данные). Надо учесть, что требуемого элемента может и не быть, тогда просмотр заканчивается при достижении конца списка. Такой подход приводит к следующему алгоритму:

1) начать с головы списка;
2) пока текущий элемент существует (указатель – не **NULL**), проверить нужное условие и перейти к следующему элементу;
3) закончить, когда найден требуемый элемент или все элементы списка просмотрены.

Например, следующая функция ищет в списке элемент, соответствующий заданному слову (для которого поле **word** совпадает с заданной строкой **NewWord**), и возвращает его адрес или **NULL**, если такого узла нет.

```
PNode Find (PNode Head, char NewWord[])
{
   PNode q = Head;
   while (q && strcmp(q->word, NewWord))
      q = q->next;
   return q;
}
```

Вернемся к задаче построения алфавитно-частотного словаря. Для того, чтобы добавить новое слово в нужное место (в алфавитном порядке), требуется найти адрес узла, *перед* которым надо вставить новое слово. Это будет первый от начала списка узел, для которого «его» слово окажется «больше», чем новое слово. Поэтому достаточно просто изменить условие в цикле **while** в функции **Find**., учитывая, что функция **strcmp** возвращает «разность» первого и второго слова.

```
PNode FindPlace (PNode Head, char NewWord[])
{
   PNode q = Head;
   while (q && (strcmp(q->word, NewWord) > 0))
      q = q->next;
   return q;
}
```

Эта функция вернет адрес узла, перед которым надо вставить новое слово (когда функция **strcmp** вернет положительное значение), или **NULL**, если слово надо добавить в конец списка.

Алфавитно-частотный словарь

Теперь можно полностью написать программу, которая обрабатывает файл **input.txt** и составляет для него алфавитно-частотный словарь в файле **output.txt**.

```
void main()
{
  PNode Head = NULL, p, where;
  FILE  *in, *out;
  char  word[80];
  int   n;
  in = fopen ( "input.dat", "r" );

  while ( 1 ) {
     n = fscanf ( in, "%s", word ); // читаем слово из файла
     if ( n <= 0 ) break;

     p = Find ( Head, word ); // ищем слово в списке

     if ( p != NULL )  // если нашли слово,
        p->count ++;   //       увеличить счетчик
     else {
        p = CreateNode ( word );  // создаем новый узел
        where = FindPlace ( Head, word ); // ищем место
        if ( ! where )
             AddLast ( Head, p );
        else AddBefore ( Head, where, p );
        }
     }
  fclose(in);

  out = fopen ( "output.dat", "w" );
  p = Head;
  while ( p ) {  // проход по списку и вывод результатов
     fprintf ( out, "%-20s\t%d\n", p->word, p->count );
     p = p->next;
     }
  fclose(out);
}
```

В переменной **n** хранится значение, которое вернула функция **fscanf** (количество удачно прочитанных элементов). Если это число меньше единицы (чтение прошло неудачно или закончились данные в файле), происходит выход из цикла **while**.

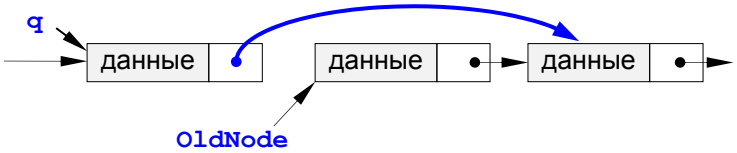
Сначала пытаемся искать это слово в списке с помощью функции **Find**. Если нашли – просто увеличиваем счетчик найденного узла. Если слово встретилось впервые, в памяти созда-

ется новый узел и заполняется данными. Затем с помощью функции **FindPlace** определяем, перед каким узлом списка надо его добавить.

Когда список готов, открываем файл для вывода и, используя стандартный проход по списку, выводим найденные слова и значения счетчиков.

Удаление узла

Эта процедура также связана с поиском заданного узла по всему списку, так как нам надо поменять ссылку у предыдущего узла, а перейти к нему непосредственно невозможно. Если мы нашли узел, за которым идет удаляемый узел, надо просто переставить ссылку.



Отдельно обрабатывается случай, когда удаляется первый элемент списка. При удалении узла освобождается память, которую он занимал.

Отдельно рассматриваем случай, когда удаляется первый элемент списка. В этом случае адрес удаляемого узла совпадает с адресом головы списка **Head** и надо просто записать в **Head** адрес следующего элемента.

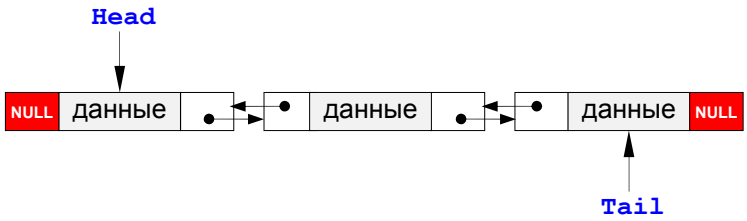
```
void DeleteNode(PNode &Head, PNode OldNode)
{
PNode q = Head;
if (Head == OldNode)
   Head = OldNode->next;    // удаляем первый элемент
else {
   while (q  &&  q->next != OldNode) // ищем элемент
      q = q->next;
   if ( q == NULL ) return; // если не нашли, выход
   q->next = OldNode->next;
   }
delete OldNode;             // освобождаем память
}
```

Барьеры

Вы заметили, что для рассмотренного варианта списка требуется отдельно обрабатывать граничные случаи: добавление в начало, добавление в конец, удаление одного из крайних элементов. Можно значительно упростить приведенные выше процедуры, если установить два барьера – фиктивные первый и последний элементы. Таким образом, в списке всегда есть хотя бы два элемента-барьера, а все рабочие узлы находятся между ними.

Двусвязный список

Многие проблемы при работе с односвязным списком вызваны тем, что в них невозможно перейти к предыдущему элементу. Возникает естественная идея – хранить в памяти ссылку не только на следующий, но и на предыдущий элемент списка. Для доступа к списку используется не одна переменная-указатель, а две – ссылка на «голову» списка (**Head**) и на «хвост» - последний элемент (**Tail**).



Каждый узел содержит (кроме полезных данных) также ссылку на **следующий** за ним узел (поле **next**) и предыдущий (поле **prev**). Поле **next** у последнего элемента и поле **prev** у первого содержат **NULL**. Узел объявляется так:

```
struct Node {
   char word[40];      // область данных
   int  count;
   Node *next, *prev; // ссылки на соседние узлы
};
typedef Node *PNode;  // тип данных «указатель на узел»
```

В дальнейшем мы будем считать, что указатель **Head** указывает на начало списка, а указатель **Tail** – на конец списка:

```
PNode Head = NULL, Tail = NULL;
```

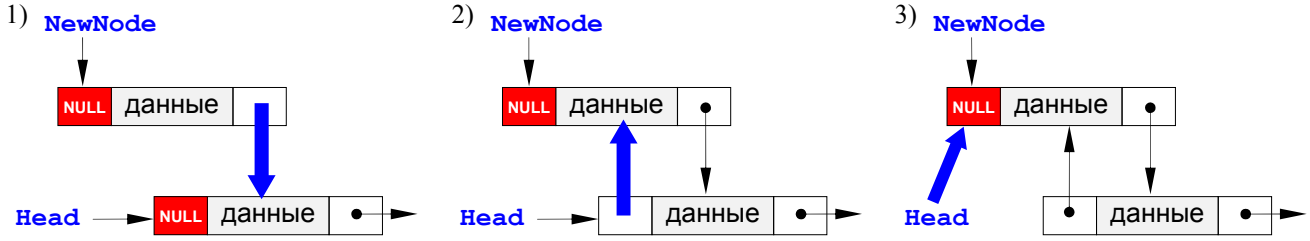
Для пустого списка оба указателя равны **NULL**.

Операции с двусвязным списком

Добавление узла в начало списка

При добавлении нового узла **NewNode** в начало списка надо

1) установить ссылку **next** узла **NewNode** на голову существующего списка и его ссылку **prev** в **NULL**;
2) установить ссылку **prev** бывшего первого узла (если он существовал) на **NewNode**;
3) установить голову списка на новый узел;
4) если в списке не было ни одного элемента, хвост списка также устанавливается на новый узел.



По такой схеме работает следующая процедура:

```
void AddFirst(PNode &Head, PNode &Tail, PNode NewNode)
{
   NewNode->next = Head;
   NewNode->prev = NULL;
   if ( Head ) Head->prev = NewNode;
   Head = NewNode;
   if ( ! Tail ) Tail = Head; // этот элемент – первый
}
```

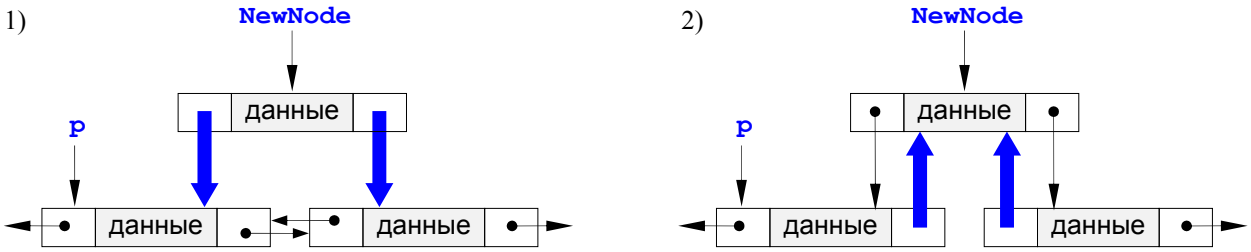
Добавление узла в конец списка

Благодаря симметрии добавление нового узла **NewNode** в конец списка проходит совершенно аналогично, в процедуре надо везде заменить **Head** на **Tail** и наоборот, а также поменять **prev** и **next**.

Добавление узла после заданного

Дан адрес **NewNode** нового узла и адрес **p** одного из существующих узлов в списке. Требуется вставить в список новый узел после **p**. Если узел **p** является последним, то операция сводится к добавлению в конец списка (см. выше). Если узел **p** – не последний, то операция вставки выполняется в два этапа:

1) установить ссылки нового узла на следующий за данным (***next***) и предшествующий ему (***prev***);
2) установить ссылки соседних узлов так, чтобы включить **NewNode** в список.



Такой метод реализует приведенная ниже процедура (она учитывает также возможность вставки элемента в конец списка, именно для этого в параметрах передаются ссылки на голову и хвост списка):

```
void AddAfter (PNode &Head, PNode &Tail,
               PNode p, PNode NewNode)
{
if ( ! p->next )
 AddLast (Head, Tail, NewNode);  // вставка в конец списка
else {
   NewNode->next = p->next;  // меняем ссылки нового узла
   NewNode->prev = p;
   p->next->prev = NewNode;  // меняем ссылки соседних узлов
   p->next = NewNode;
   }
}
```

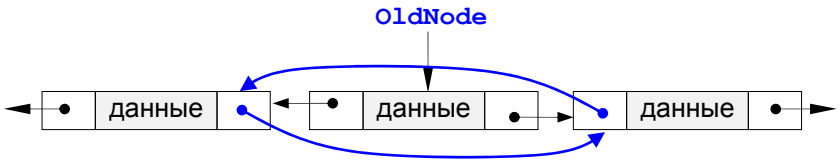
Добавление узла перед заданным выполняется аналогично.

Поиск узла в списке

Проход по двусвязному списку может выполняться в двух направлениях – от головы к хвосту (как для односвязного) или от хвоста к голове.

Удаление узла

Эта процедура также требует ссылки на голову и хвост списка, поскольку они могут измениться при удалении крайнего элемента списка. На первом этапе устанавливаются ссылки соседних узлов (если они есть) так, как если бы удаляемого узла не было бы. Затем узел удаляется и память, которую он занимает, освобождается. Эти этапы показаны на рисунке внизу. Отдельно проверяется, не является ли удаляемый узел первым или последним узлом списка.




```
void Delete(PNode &Head, PNode &Tail, PNode OldNode)
{
if (Head == OldNode) {
   Head = OldNode->next;     // удаляем первый элемент
   if ( Head )
        Head->prev = NULL;
   else Tail = NULL;         // удалили единственный элемент
   }
else {
   OldNode->prev->next = OldNode->next;
   if ( OldNode->next )
   OldNode->next->prev = OldNode->prev;
   else Tail = NULL;        // удалили последний элемент
   }
delete OldNode;
}
```

Циклические списки

Иногда список (односвязный или двусвязный) замыкают в кольцо, то есть указатель **next** последнего элемента указывает на первый элемент, и (для двусвязных списков) указатель **prev** первого элемента указывает на последний. В таких списках понятие «хвоста» списка не имеет смысла, для работы с ним надо использовать указатель на «голову», причем «головой» можно считать любой элемент.

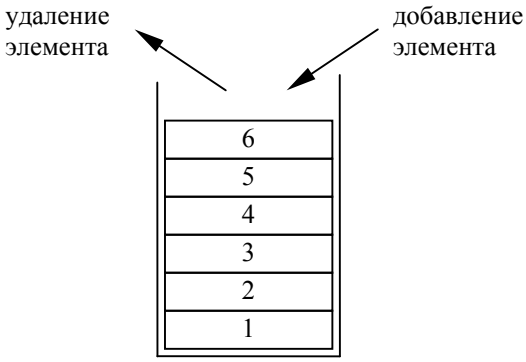
2. Стеки, очереди, деки

Структуры, перечисленные в заголовке раздела, представляют собой списки, для которых разрешены только операции вставки и удаления первого и/или последнего элемента.

Стек

Стек – это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо только с одного конца, который называется **вершиной стека**.

Стек называют структурой типа ***LIFO*** (*Last In – First Out*) – последним пришел, первым ушел. Стек похож на стопку с подносами, уложенными один на другой – чтобы достать какой-то поднос надо снять все подносы, которые лежат на нем, а положить новый поднос можно только сверху всей стопки. На рисунке показан стек, содержащий 6 элементов.



В современных компьютерах стек используется для

- размещения локальных переменных;
- размещения параметров процедуры или функции;
- сохранения адреса возврата (по какому адресу надо вернуться из процедуры);
- временного хранения данных, особенно при программировании на Ассемблере.

На стек выделяется ограниченная область памяти. При каждом вызове процедуры в стек добавляются новые элементы (параметры, локальные переменные, адрес возврата). Поэтому если вложенных вызовов будет много, стек переполнится. Очень опасной в отношении переполнения стека является **рекурсия**, поскольку она как раз и предполагает вложенные вызовы одной и той же процедуры или функции. При ошибке в программе рекурсия может стать бесконечной, кроме того, стек может переполниться, если вложенных вызовов будет слишком много.

Реализация стека с помощью массива

Если максимальный размер стека заранее известен, его можно реализовать в программе в виде массива. Удобно объединить в одной структуре сам массив и его размер. Объявим новый тип данных – стек на 100 элементов-символов.

```
const int MAXSIZE = 100;
struct Stack {
   char data[MAXSIZE];
   int  size;
   };
```

Для работы со стеком надо определить, как выполняются две операции – добавление элемента на вершину стека (**Push**) и снятие элемента с вершины стека (**Pop**).

```
void Push ( Stack &S, char x )
{
   if ( S.size == MAXSIZE ) {
      printf ("Стек переполнен");
      return;
      }
   S.data[S.size] = x;
   S.size ++;
}
```

Поскольку нумерация элементов массива **data** начинается с нуля, надо сначала записать новый элемент в **S.data[S.size]**, а затем увеличить размер стека. В процедуре предусмотрена обработка ошибки «переполнение стека». В этом случае на экран будет выдано сообщение «Стек переполнен». Можно также сделать функцию **Push**, которая будет возвращать 1 в случае удачного добавления элемента и 0 в случае ошибки.

Обратите внимание, что стек **S** передается в процедуру по ссылке, то есть, фактически передается адрес этого стека в памяти. Поэтому все операции со стеком в процедуре выполняются непосредственно со стеком вызывающей программы.

```
char Pop ( Stack &S )
{
   if ( S.size == 0 ) {
      printf ("Стек пуст");
      return char(255);
      }
   S.size --;
   return S.data[S.size];
}
```

Функция **Pop** возвращает символ, «снятый» с вершины стека, при этом размер стека уменьшается на единицу. Если стек пуст, функция возвращает символ с кодом 255 (который никогда не может находиться в стеке по условию задачи и сигнализирует об ошибке).

Задача. Ввести символьную строку, которая может содержать три вида скобок: (), [] и {}. Определить, верно ли расставлены скобки (символы между скобками не учитывать). Например, в строках **()[{}]** и **[{}([])]** скобки расставлены верно, а в строках **([)]** и **]]](((** - неверно.

Для одного вида скобок решение очень просто – ввести счетчик «вложенности» скобок, просмотреть всю строку, увеличивая счетчик для каждой открывающей скобки и уменьшая его для каждой закрывающей. Выражение записано верно, если счетчик ни разу не стал отрицательным и после обработки всей строки оказался равен нулю.

Если используются несколько видов скобок, счетчики не помогают. Однако эта задача имеет красивое решение с помощью стека. Вначале стек пуст. Проходим всю строку от начала до символа с кодом 0, который обозначает конец строки. Если встретили открывающую скобку, заносим ее в стек. Если встретили закрывающую скобку, то на вершине стека должна быть соответствующая ей открывающая скобка. Если это так, снимаем ее со стека. Если стек пуст или на вершине стека находится скобка другого вида, выражение неверное. В конце прохода стек должен быть пуст.

В приведенной ниже программе используются написанные ранее объявление структуры **Stack** и операции **Push** и **Pop**.

```
void main()
{
   char br1[3] = { '(', '[', '{' }; // открывающие скобки
   char br2[3] = { ')', ']', '}' }; // закрывающие скобки
   char s[80], upper;
   int i, k, OK;
   Stack S;        // стек символов

   printf("Введите выражение со скобками> ");
   gets ( s );

   S.size = 0;     // сначала стек пуст
   OK = 1;

   for (i = 0; OK  &&  (s[i] != '\0'); i ++)

      for ( k = 0; k < 3; k ++ ) // проверить 3 вида скобок
        {
        if ( s[i] == br1[k] ) {  // открывающая скобка
          Push ( S, s[i] ); break;
          }
        if ( s[i] == br2[k] ) {  // закрывающая скобка
          upper = Pop ( S );
          if ( upper != br1[k] ) OK = 0;
          break;
          }
        }

   if ( OK && (S.size == 0) )
        printf("\nВыражение правильное\n");
   else printf("\nВыражение неправильное \n");
}
```

Открывающие и закрывающие скобки записаны в массивах **br1** и **br2**. В самом начале стек пуст и его размер равен нулю (**S.size = 0**). Переменная **OK** служит для того, чтобы выйти из внешнего цикла, когда обнаружена ошибка (и не рассматривать оставшуюся часть строки). Она устанавливается в нуль, если в стеке обнаружена скобка другого типа или стек оказался пуст.

Реализация стека с помощью списка

Рассмотрим пример стека, в котором хранятся символы (это простейший вариант, элементом стека могут быть любые типы данных или структур, так же, как и для списка). Реализуем стек на основе двусвязного списка. При этом количество элементов стека ограничивается только доступным объемом памяти.

```
struct Node {
   char data;
   Node *next, *prev;
   };
typedef Node *PNode;
```

Чтобы не работать с отдельными указателями на хвост и голову списка, объявим структуру, в которой будет храниться вся информация о стеке:

```
struct Stack {
   PNode Head, Tail;
   };
```

В самом начале надо записать в обе ссылки стека **NULL**.

Добавление элемента на вершину стека

Фактически это добавление нового элемента в начало двусвязного списка. Эта процедура уже была написана ранее, теперь ее придется немного переделать, чтобы работать не с отдельными указателями, а со структурой типа **Stack**. В параметрах процедуры указывается не новый узел, а только *данные* для этого узла, то есть целое число. Память под новый узел выделяется в процедуре, то есть, скрыта от нас и снижает вероятность ошибки.

```
void Push ( Stack &S, char x )
{
   PNode NewNode;
   NewNode = new Node;      // создать новый узел...
   NewNode->data = x;       // и заполнить его данными
   NewNode->next = S.Head;
   NewNode->prev = NULL;

   if ( S.Head )               // добавить в начало списка
     S.Head->prev = NewNode;
   S.Head = NewNode;

   if ( ! S.Tail ) S.Tail = S.Head;
}
```

Получение верхнего элемента с вершины стека

Функция ***Pop*** удаляет верхний элемент и возвращает его данные.

```
char Pop ( Stack &S )
{
   PNode TopNode = S.Head;
   char x;

   if ( ! TopNode )    // если стек пуст, то
     return char(255); //      вернуть символ с кодом 255
   x = TopNode->data;

   S.Head = TopNode->next;

   if ( S.Head ) S.Head->prev = NULL; // переставить ссылки
   else          S.Tail = NULL;

   delete TopNode; // освободить память
   return x;
}
```

Системный стек в программах

При выполнении программ определенная область памяти отводится на стек программы. Более того, в процессоре есть специальная ячейка (регистр), в которой хранится адрес вершины стека. Программа использует стек для хранения

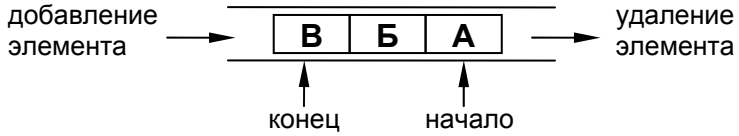
1) адресов возврата из процедур и функций (это адреса, на которые переходит программа после выполнения процедуры или функции);
2) параметров, передаваемых в процедуры и функции;
3) локальных переменных в процедурах и функциях;
4) временных данных (в основном в программах на ассемблере).

Больше всего места занимают в стеке локальные переменные. Поэтому память под большие массивы надо выделять динамически. Кроме того, желательно не передавать в процедуры большие структуры, вместо этого можно передать их адрес или использовать передачу по ссылке (при этом перед именем параметра должен стоять знак **&**).

Очередь

Очередь – это упорядоченный набор элементов, в котором добавление новых элементов допустимо с одного конца (он называется **начало очереди**), а удаление существующих элементов – только с другого конца, который называется **концом очереди**.

Хорошо знакомой моделью является очередь в магазине. Очередь называют структурой типа ***FIFO*** (*First In – First Out*) – первым пришел, первым ушел. На рисунке изображена очередь из 3-х элементов.



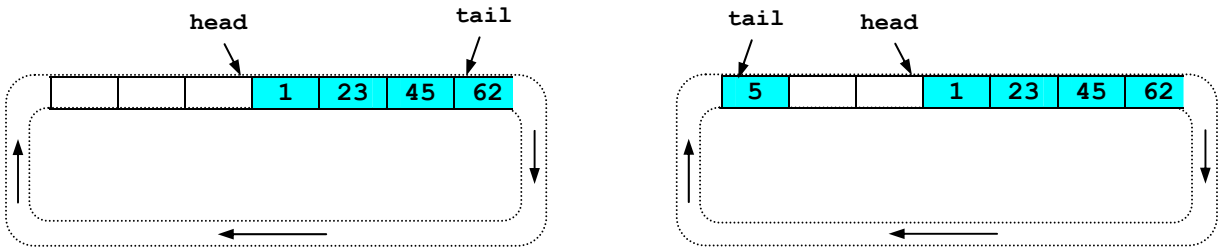
Наиболее известные примеры применения очередей в программировании – очередь событий системы *Windows* и ей подобных. Очереди используются также для моделирования в **задачах массового обслуживания** (например, обслуживания клиентов в банке).

Реализация очереди с помощью массива

Если максимальный размер очереди заранее известен, его можно реализовать в программе в виде массива. Удобно объединить в одной структуре сам массив и его размер. Объявим новый тип данных – очередь на 100 элементов (целых чисел).

```
const int MAXSIZE = 100;
struct Queue {
   int data[MAXSIZE];
   int size, head, tail;
   };
```

Если у стека один конец «закреплен» (не двигается), то у очереди «подвижны» оба конца. Чтобы не сдвигать все элементы в массиве при удалении или добавлении элемента, обычно использую две переменные **head** и **tail** – первая из них обозначает номер первого элемента в очереди, а вторая – номер последнего. Если они равны, то в очереди всего один элемент. Массив как бы замыкается в кольцо – если массив закончился, но в начале массива есть свободные места, то новый элемент добавляется в начало массива, как показано на рисунках.



Для работы с очередью надо определить, как выполняются две операции – добавление элемента в конец очереди (**PushTail**) и удаление элемента с начала очереди (**Pop**).

```
void PushTail ( Queue &Q, int x )
{
   if ( Q.size == MAXSIZE ) {
      printf ("Очередь переполнена\n");
   return;
      }
   Q.tail++;
   if ( Q.tail >= MAXSIZE ) // замыкание в кольцо
      Q.tail = 0;
   Q.data[Q.tail] = x;
   Q.size ++;
}
```

Поскольку очередь может начинаться не с начала массива (за счет того, что некоторые элементы уже «выбраны»), после увеличения **Q.tail** надо проверить, не вышли ли мы за границу массива. Если это случилось, новый элемент записывается в начало массива (хотя является хвостом очереди). В процедуре предусмотрена обработка ошибки «переполнение очереди». В этом случае на экран будет выдано сообщение «Очередь переполнена». Можно также сделать функцию **PushTail**, которая будет возвращать 1 в случае удачного добавления элемента и 0 в случае ошибки.

Обратите внимание, что очередь ***Q*** передается в процедуру по ссылке, то есть, фактически передается адрес этой структуры в памяти.

```
int Pop ( Queue &Q )
{
   int temp;
   if ( Q.size == 0 ) {
      printf ("Очередь пуста\n");
      return 32767;        // сигнал об ошибке
      }
   temp = Q.data[Q.head];
   Q.head ++;
   if ( Q.head >= MAXSIZE ) Q.head = 0;
   Q.size --;
   return temp;
}
```

Функция **Pop** возвращает число, полученное с начала очереди, при этом размер очереди уменьшается на единицу. Если стек пуст, функция возвращает число 32767 (предполагается, что оно не может находиться в очереди по условию задачи и сигнализирует об ошибке).

Реализация очереди с помощью списка

Если максимальный размер заранее неизвестен или требуется сделать его динамическим, для реализации используют список. Рассмотрим пример очереди, элементами которой являются целые числа. При этом количество ее элементов ограничивается только доступным объемом памяти. Новые типы данных (узел и указатель на него) объявляются так же, как для списка:

```
struct Node {
   int  data;
   Node *next, *prev;
   };
typedef Node *PNode;
```

Чтобы не работать с отдельными указателями на хвост и голову списка, объявим структуру, в которой будет храниться вся информация об очереди:

```
struct Queue {
   PNode head, tail;
   };
```

В самом начале надо записать в обе ссылки **NULL**. Заметим, что такая же структура использовалась и для стека. Более того, функция для получения первого элемента очереди (**Pop**) совпадает с функцией снятия элемента с вершины стека (напишите ее в качестве упражнения).

Добавление элемента в конец очереди

Фактически это добавление нового элемента в конец двусвязного списка. В параметрах процедуры указывается не новый узел, а только *данные* для этого узла, то есть целое число. Память под новый узел выделяется в процедуре, то есть, скрыта от нас и снижает вероятность ошибки.

```
void PushTail ( Queue &Q, int x )
{
   PNode NewNode;
   NewNode = new Node;      // создать новый узел
   NewNode->data = x;       // заполнить узел данными
   NewNode->prev = Q.Tail;
   NewNode->next = NULL;

   if ( Q.tail )              // добавить узел в конец списка
     Q.tail->next = NewNode;
   Q.tail = NewNode;

   if ( ! Q.head ) Q.head = Q.tail;
}
```

Дек

Дек (*deque*) - это упорядоченный набор элементов, в котором добавление новых и удаление существующих элементов допустимо с любого конца.

Дек может быть реализован на основе массива или двусвязного списка. Для дека разрешены четыре операции:

1) добавление элемента в начало;
2) добавление элемента в конец;
3) удаление элемента с начала;
4) удаление элемента с конца.

Их можно реализовать, используя написанные выше процедуры для стека и очереди.

3. Деревья

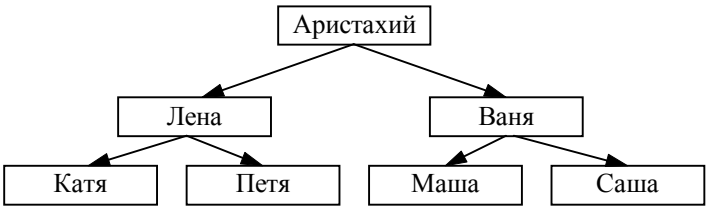
Что такое деревья?

Основные понятия

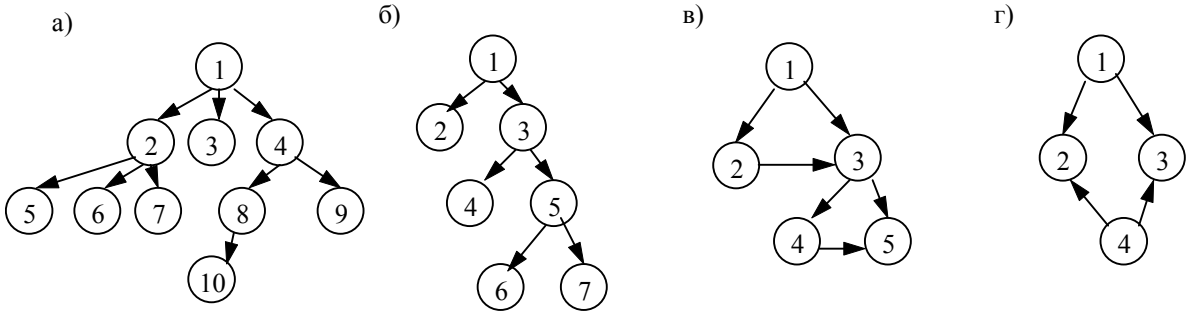
Дерево – это совокупность **узлов** (**вершин**) и соединяющих их направленных **ребер** (**дуг**), причем в каждый узел (за исключением одного **- корня**) ведет ровно одна дуга.

Корень – это начальный узел дерева, в который не ведет ни одной дуги.

Примером может служить **генеалогическое дерево** - в корне дерева находитесь вы сами, от вас идет две дуги к родителям, от каждого из родителей - две дуги к их родителям и т.д.



Например, на рисунке структуры а) и б) являются деревьями, а в) и г) - нет.



Предком для узла ***x*** называется узел дерева, из которого существует путь в узел ***x***.

Потомком узла ***x*** называется узел дерева, в который существует путь (по стрелкам) из узла ***x***.

Родителем для узла ***x*** называется узел дерева, из которого существует непосредственная дуга в узел ***x***.

Сыном узла ***x*** называется узел дерева, в который существует непосредственная дуга из узла ***x***.

Уровнем узла ***x*** называется длина пути (количество дуг) от корня к данному узлу. Считается, что корень находится на уровне 0.

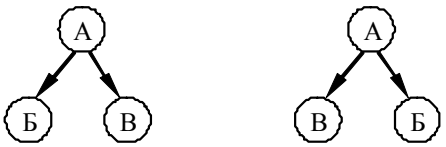
Листом дерева называется узел, не имеющий потомков.

Внутренней вершиной называется узел, имеющий потомков.

Высотой дерева называется максимальный уровень листа дерева.

Упорядоченным деревом называется дерево, все вершины которого упорядочены (то есть имеет значение последовательность перечисления потомков каждого узла).

Например, два упорядоченных дерева на рисунке ниже – разные.



Рекурсивное определение

Дерево представляет собой типичную рекурсивную структуру (определяемую через саму себя). Как и любое рекурсивное определение, определение дерева состоит из двух частей – первая определяет условие окончания рекурсии, а второе – механизм ее использования.

1) пустая структура является деревомя;
2) дерево – это корень и несколько связанных с ним деревьев (поддеревьев).

Таким образом, размер памяти, необходимый для хранения дерева, заранее неизвестен, потому что неизвестно, сколько узлов будет в него входить.

Двоичные деревья

На практике используются главным образом деревья особого вида, называемые **двоичными (бинарными).**

Двоичным деревом называется дерево, каждый узел которого имеет не более двух сыновей.

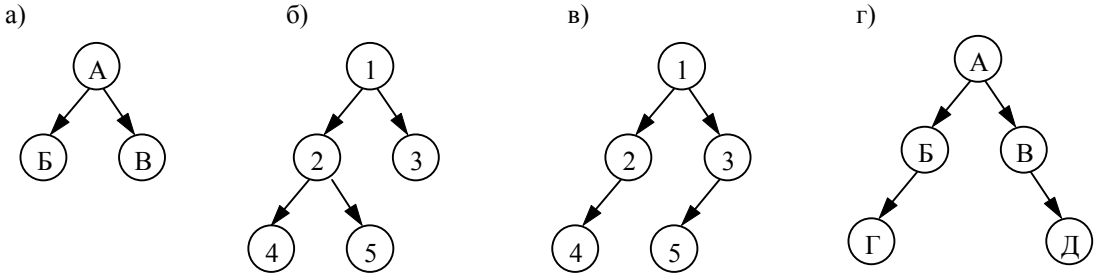
Можно определить двоичное дерево и рекурсивно:

1) пустая структура является двоичным деревом;
2) дерево – это корень и два связанных с ним двоичных дерева, которые называют **левым** и **правым** поддеревом .

Двоичные деревья **упорядочены**, то есть различают левое и правое поддеревья. Типичным примером двоичного дерева является генеалогическое дерево (родословная). В других случаях двоичные деревья используются тогда, когда на каждом этапе некоторого процесса надо принять одно решение из двух возможных. В дальнейшем мы будем рассматривать только двоичные деревья.

Строго двоичным деревом называется дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Это означает, что в строго двоичном дереве нет вершин, у которых есть только одно поддерево. На рисунке даны деревья а) и б) являются строго двоичными, а в) и г) – нет.



Полным двоичным деревом называется дерево, у которого все листья находятся на одно уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.

На рисунке выше только дерево а) является полным двоичным деревом.

Реализация двоичных деревьев в языке Си

Описание вершины

Вершина дерева, как и узел любой динамической структуры, имеет две группы данных: полезную информацию и ссылки на узлы, связанные с ним. Для двоичного дерева таких ссылок будет две – ссылка на **левого сына** и ссылка на **правого сына**. В результате получаем структу-

ру, описывающую вершину (предполагая, что полезными данными для каждой вершины является одно целое число):

```
struct Node {
    int    key;              // полезные данные (ключ)
    Node   *left, *right;    // указатели на сыновей
 };
typedef Node *PNode;         // указатель на вершину
```

Деревья минимальной высоты

Для большинства практических задач наиболее интересны такие деревья, которые имеют минимально возможную высоту при заданном количестве вершин **n**. Очевидно, что минимальная высота достигается тогда, когда на каждом уровне (кроме, возможно, последнего) будет максимально возможное число вершин.

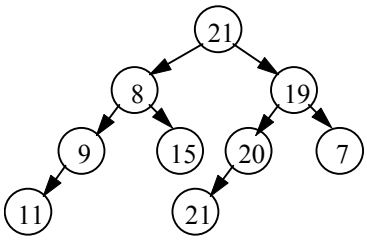
Предположим, что задано ***n*** чисел (их количество заранее известно). Требуется построить из них дерево минимальной высоты. Алгоритм решения этой задачи предельно прост.

1. Взять одну вершину в качестве корня и записать в нее первое нерассмотренное число.
2. Построить этим же способом левое поддерево из **n1 = n/2** вершин (деление нацело!).
3. Построить этим же способом правое поддерево из **n2 = n-n1-1** вершин.

Заметим, что по построению левое поддерево всегда будет содержать столько же вершин, сколько правое поддерево, или на 1 больше. Для массива данных

21, 8, 9, 11, 15, 19, 20, 21, 7

по этому алгоритму строится дерево, показанное на рисунке справа.



Как будет выглядеть эта программа на языке Си? Надо сначала разобраться, что означает «взять одну вершину в качестве корня и записать туда первое нерассмотренное число». Поскольку вершины должны создаваться динамически, надо **выделить память** под вершину и записать в поле данных нужное число. Затем из оставшихся чисел построить левое и правое поддеревья.

В основной программе нам надо объявить указатель на корень нового дерева, задать массив данных (в принципе можно читать данные из файла) и вызвать функцию, возвращающую указатель на построенное дерево.

```
int data[] = { 21, 8, 9, 11, 15, 19, 20, 21, 7 };
PNode Tree;   // указатель на корень дерева
n = sizeof(data) / sizeof(int) - 1; // размер массива
Tree = MakeTree (data, 0, n);  // использовать n элементов,
                               // начиная с номера 0
```

Сама функция `MakeTree` принимает три параметра: массив данных, номер первого неиспользованного элемента и количество элементов в новом дереве. Возвращает она указатель на новое дерево (типа `PNode`).

```
PNode MakeTree (int data[], int from, int n)
{
PNode Tree;
int n1, n2;
if ( n == 0 ) return NULL;  // ограничение рекурсии
Tree = new Node;  // выделить память под вершину
Tree->key = data[from];     // записать данные (ключ)
```

```
  n1 = n / 2;                  // размеры поддеревьев
  n2 = n - n1 - 1;
  Tree->left  = MakeTree(data, from+1, n1);
  Tree->right = MakeTree(data, from+1+n1, n2);
  return Tree;
  }
```

Выделенные строчки программы содержат рекурсивные вызовы. При этом левое поддерево содержит **n1** элементов массива начиная с номера **from+1**, тогда как правое – **n2** элементов начиная с номера **from+1+n1**.

Обход дерева

Одной из необходимых операций при работе с деревьями является **обход дерева**, во время которого надо посетить каждый узел по одному разу и (возможно) вывести информацию, содержащуюся в вершинах.

Пусть в результате обхода надо напечатать значения поля данных всех вершин в определенном порядке. Существуют **три варианта обхода**:

1) **КЛП (*корень – левое – правое*)**: сначала посещается корень (выводится информация о нем), затем левое поддерево, а затем – правое;
2) **ЛКП (*левое – корень – правое*)**: сначала посещается левое поддерево, затем корень, а затем – правое;
3) **ЛПК (*левое – правое – корень*)**: сначала посещается левое поддерево, затем правое, а затем – корень.

Для примера ниже дана рекурсивная процедура просмотра дерева в порядке ЛКП. Обратите внимание, что поскольку дерево является рекурсивной структурой данных, при работе с ним естественно широко применять рекурсию.

```
void PrintLKP(PNode Tree)
{
if ( ! Tree ) return;      // пустое дерево – окончание рекурсии
PrintLKP(Tree->left);      // обход левого поддерева
printf("%d ", Tree->key);  // вывод информации о корне
PrintLKP(Tree->right);     // обход правого поддерева
}
```

Остальные варианты обхода программируются аналогично.

Поиск с помощью дерева

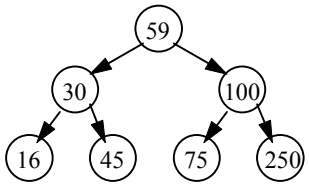
Как быстрее искать?

Деревья очень удобны для поиска в них информации. Однако для быстрого поиска требуется предварительная подготовка – дерево надо построить специальным образом.

Предположим, что существует массив данных и с каждым элементом связан ключ - число, по которому выполняется поиск. Пусть ключи для элементов таковы:

59, 100, 75, 30, 16, 45, 250

Для этих данных нам надо много раз проверять, есть ли среди ключей заданный ключ `x`, и если есть, то вывести всю связанную с этим элементом информацию.



Если данные организованы в виде массива (без сортировки), то для поиска в худшем случае надо сделать **n** сравнений элементов (сравнивая последовательно с каждым элементом пока не найдется нужный или пока не закончится массив).

Теперь предположим, что данные организованы в виде дерева, показанного на рисунке. Такое дерево (оно называется **дерево поиска**) обладает следующим важным свойством:

> Значения ключей всех вершин левого поддерева вершины **x** меньше ключа **x**, а значения ключей всех вершин правого поддерева **x** больше или равно ключу вершины **x.**

Для поиска нужного элемента в таком дереве требуется не более 3 сравнений вместо 7 при поиске в списке или массиве, то есть поиск проходит значительно быстрее. С ростом количества элементов эффективность поиска по дереву растет.

Построение дерева поиска

Как же, имея массив данных, построить такое дерево?

1. Сравнить ключ очередного элемента массива с ключом корня.
2. Если ключ нового элемента меньше, включить его в левое поддерево, если больше или равен, то в правое.
3. Если текущее дерево пустое, создать новую вершину и включить в дерево.

Программа, приведенная ниже, реализует этот алгоритм:

```
void AddToTree (PNode &Tree,  // указатель на корень (ссылка)
                int data)     // добавляемый ключ
{
if ( ! Tree ) {
   Tree = new Node;     // создать новый узел
   Tree->key = data;
   Tree->left = NULL;
   Tree->right = NULL;
   return;
   }
if ( data < Tree->key )  // добавить в нужное поддерево
       AddToTree ( Tree->left, data );
else AddToTree ( Tree->right, data );
}
```

Важно, что указатель на корень дерева надо передавать по ссылке, так как он может измениться при создании новой вершины.

Надо заметить, что в результате работы этого алгоритма не всегда получается дерево минимальной высоты – все зависит от порядка выбора элементов. Для оптимизации поиска используют так называемые **сбалансированные** или **АВЛ-деревья**[1] деревья, у которых для любой вершины высоты левого и правого поддеревьев отличаются не более, чем на 1. Добавление в них нового элемента иногда сопровождается некоторой перестройкой дерева.

Поиск по дереву

Теперь, когда дерево сортировки построено, очень легко искать элемент с заданным ключом. Сначала проверяем ключ корня, если он равен искомому, то нашли. Если он меньше искомого, ищем в левом поддереве корня, если больше – то в правом. Приведенная функция возвращает адрес нужной вершины, если поиск успешный, и **NULL**, если требуемый элемент не найден.

[1] Сбалансированные деревья называют так в честь изобретателей этого метода Г.М. Адельсона-Вельского и Е.М. Ландиса.

```
PNode Search (PNode Tree, int what)
{
if ( ! Tree ) return NULL;    // ключ не найден
if ( what == Tree->key ) return Tree; // ключ найден!
if ( what < Tree->key )               // искать в поддеревьях
     return Search ( Tree->left, what );
else return Search ( Tree->right, what );
}
```

Сортировка с помощью дерева поиска

Если дерево поиска построено, очень просто вывести отсортированные данные. действительно, обход типа ЛКП (*левое поддерево – корень – правое поддерево*) даст ключи в порядке возрастания, а обход типа ПКЛ (*правое поддерево – корень – левое поддерево*) – в порядке убывания.

Поиск одинаковых элементов

Приведенный алгоритм можно модифицировать так, чтобы быстро искать одинаковые элементы в массиве чисел. Конечно, можно перебрать все элементы массива и сравнить каждый со всеми остальными. Однако для этого требуется очень большое число сравнений. С помощью двоичного дерева можно значительно ускорить поиск. Для этого надо в структуру вершины включить еще одно поле – счетчик найденных дубликатов **count**.

```
struct Node {
 int   key;
 int   count;          // счетчик дубликатов
 Node  *left, *right;
 };
```

При создании узла в счетчик записывается единица (найден один элемент). Поиск дубликатов происходит по следующему алгоритму:

1. Сравнить ключ очередного элемента массива с ключом корня.
2. Если ключ нового элемента равен ключу корня, то увеличить счетчик корня и стоп.
3. Если ключ нового элемента меньше, чем у корня, включить его в левое поддерево, если больше или равен – в правое.
4. Если текущее дерево пустое, создать новую вершину (со значением счетчика 1) и включить в дерево.

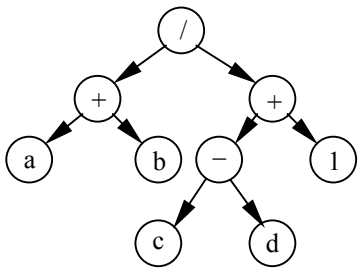
Разбор арифметического выражения

Дерево для арифметического выражения

Вы задумывались над тем, как транслятор обрабатывает и выполняет арифметические и логические выражения, которые он встречает в программе? Один из вариантов - представить это выражение в виде двоичного дерева. Например, выражению

(a + b) / (c - d + 1)

соответствует дерево, показанное на рисунке слева. Листья содержат числа и имена переменных (операндов), а внутренние вершины и корень – арифметические действия и вызовы функций. Вычисляется такое выражение снизу, начиная с листьев. Как видим, скобки отсутствуют, и дерево полностью определяет порядок выполнения операций.



Формы записи арифметического выражения

Теперь посмотрим, что получается при прохождении таких двоичных деревьев. Прохождение дерева в ширину (корень – левое – правое) дает

```
/ + a b + - c d 1
```

то есть знак операции (корень) предшествует своим операндам. Такая форма записи арифметических выражений называется **префиксной**. Проход в прямом порядке (левое – корень – правое) дает **инфиксную форму**, которая совпадает с обычной записью, но без скобок:

```
a + b /  c -  d + 1
```

Поскольку скобок нет, по инфиксной записи невозможно восстановить правильный порядок операций.

В трансляторах широко используется **постфиксная запись** выражений, которая получается в результате обхода в порядке ЛПК (левое – правое – корень). В ней знак операции стоит **после** обоих операндов:

```
a  b  +  c  d - 1 /
```

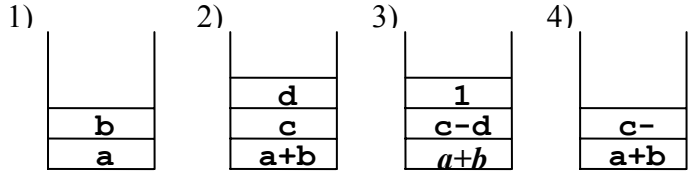
Порядок выполнения такого выражения однозначно определяется следующим алгоритмом, который использует стек:

Пока в постфиксной записи есть невыбранные элементы,
1) взять очередной элемент;
2) если это операнд (не знак операции), то записать его в стек;
3) если это знак операции, то
   - выбрать из стека второй операнд;
   - выбрать из стека первый операнд;
   - выполнить операцию с этими данными и результат записать в стек.

Проиллюстрируем на примере вычисление выражения в постфиксной форме

```
a  b  +  c  d - 1 /
```

Согласно алгоритму, сначала запишем в стек **a**, а затем **b** (рисунок 1).



Результат выполнения операции **a+b** запишем обратно в стек, а сверху – выбранные из входного потока значения переменных **c** и **d** (рисунок 2). Дальнейшее развитие событий показано на рисунках 3 и 4. Выполнение последней операции (деления) для стека на рисунке 4 дает искомый результат.

Алгоритм построения дерева

Пусть задано арифметическое выражение. Надо построить для него дерево синтаксического разбора и различные формы записи.

Чтобы не слишком усложнять задачу, рассмотрим самый простой вариант, введя следующие упрощения.

1. В выражении могут присутствовать только однозначные целые числа знаки операций **+ - * /.**
2. Запрещается использование вызовов функций, скобок, унарных знаков плюс и минус (например, запрещено выражение **-a+5**, вместо него надо писать **0-a+5**).
3. Предполагается, что выражение записано верно, то есть не делается проверки на правильность.

Вспомним, что порядок выполнения операций в выражении определяется **приоритетом операций** – первыми выполняются операции с более высоким приоритетом. Например, умножение и деление выполняются раньше, чем сложение и вычитание.

Будем правильное арифметическое выражение записано в виде символьной строки **Expr** длиной **N**. Построим дерево для элементов массива с номерами от **first** до **last** (полное дерево дает применение этого алгоритма ко всему массиву, то есть при **first=0** и **last=N-1**). В словесном виде алгоритм выглядит так:

1. Если **first=last** (остался один элемент – переменная или число), то создать новый узел и записать в него этот элемент. Иначе...
2. Среди элементов от **first** до **last** включительно найти **последнюю** операцию с наименьшим приоритетом (пусть найденный элемент имеет номер **k**).
3. Создать новый узел (корень) и записать в него знак операции **Expr[k]**.
4. Рекурсивно применить этот алгоритм два раза:
   - построить левое поддерево, разобрав выражение из элементов массива с номерами от **first** до **k-1**
   - построить правое поддерево, разобрав выражение из элементов массива с номерами от **k+1** до **last**

Объявим структуру, описывающую узел такого дерева. Так как мы используем только однозначные целые числа и знаки, область данных может содержать один символ.

```
struct Node {
        char data;
        Node *left, *right;
        };
typedef Node *PNode;
```

Далее надо определить функцию, возвращающую приоритет операции, которая ей передана. Определим приоритет 1 для сложения и вычитания и приоритет 2 для умножения и деления.

```
int Priority ( char c )
{
switch ( c ) {
   case '+': case '-': return 1;
   case '*': case '/': return 2;
   }
return 100; // это не арифметическая операция
}
```

Приведенная ниже процедура строит требуемое дерево, используя эту функцию, и возвращает адрес построенного дерева в памяти. Обратите внимание, что при сравнении приоритета текущей операции с минимальным предыдущим используется условие **<=**. За счет этого мы ищем именно **последнюю** операцию с минимальным приоритетом, то есть, операцию, которая будет выполняться самой последней. Если бы мы использовали знак **<**, то нашли бы **первую** операцию с наименьшим приоритетом, и дерево было бы построено неверно (вычисления дают неверный результат, если встречаются два знака вычитания или деления).

```
PNode MakeTree (char Expr[], int first, int last)
{
int MinPrt, i, k, prt;
PNode Tree = new Node;  // создать в памяти новую вершину
if ( first == last ) {  // конечная вершина: число или
   Tree->data = Expr[first];  //              переменная
```

```
    Tree->left = NULL;
    Tree->right = NULL;
    return Tree;
    }
  MinPrt = 100;
  for ( i = first; i <= last; i ++ ) {
     prt = Priority ( Expr[i] );
    if ( prt <= MinPrt ) { // ищем последнюю операцию
       MinPrt = prt;       //      с наименьшим приоритетом
       k = i;
       }
   }
  Tree->data = Expr[k];  // внутренняя вершина (операция)
  Tree->left  = MakeTree (Expr,first,k-1); // рекурсивно строим
  Tree->right = MakeTree (Expr,k+1,last);  // поддеревья
  return Tree;
  }
```

Теперь обход этого дерева разными способами дает различные формы представления соответствующего арифметического выражения.

Вычисление выражения по дереву

Пусть для некоторого арифметического выражения построено дерево и известен его адрес **Tree**. Напишем функцию, которая возвращает целое число – результат вычисления этого выражения. Учтем, что деление выполняется нацело (остаток отбрасывается).

```
  int CalcTree (PNode Tree)
  {
     int num1, num2;
     if ( ! Tree->left )            // если нет потомков,
        return Tree->data - '0';    //      вернули число
     num1 = CalcTree(Tree->left);  // вычисляем поддеревья
     num2 = CalcTree(Tree->right);
     switch ( Tree->data ) {       // выполняем операцию
        case '+': return  num1+num2;
        case '-': return  num1-num2;
        case '*': return  num1*num2;
        case '/': return  num1/num2;
        }
     return 32767;  // неизвестная операция, ошибка!
  }
```

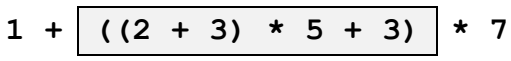
Если дерево не имеет потомков, значит это число. Чтобы получить результат как целое число, из кода этой цифры надо вычесть код цифры `'0'`. Если потомки есть, вычисляем левое и правое поддеревья (рекурсивно!) и выполняем операцию, записанную в корне дерева. Основная программа может выглядеть так, как показано ниже.

```
  void main()
  {
    char s[80];
    PNode Tree;
    printf("Введите выражение > ");
```

```
    gets(s);
    Tree = MakeTree(s, 0, strlen(s)-1);
    printf ( "= %d \n", CalcTree ( Tree ) );
    getch();
}
```

Разбор выражения со скобками

Немного усложним задачу, разрешив использовать в выражении скобки одного вида (допустим, круглые). Тогда при поиске в заданном диапазоне операции с минимальным приоритетом не надо брать во внимание выражения в скобках (они выделены на рисунке).



Самый простой способ добиться этого эффекта – ввести счетчик открытых скобок **nest**. В начале он равен нулю, с каждой найденной открывающей скобкой будем увеличивать его на 1, а с каждой закрывающей – уменьшать на 1. Рассматриваются только те операции, которые найдены при **nest=0**, то есть, расположены вне скобок.

Если же ни одной такой операции не найдено, то мы имеем выражение, ограниченной скобками, поэтому надо вызвать процедуру рекурсивно для диапазона **from+1..last-1** (напомним, что мы предполагаем, что выражение корректно). Для сокращения записи показаны только те части процедуры, которые изменяются:

```
PNode MakeTree (char Expr[], int first, int last)
{
int MinPrt, i, k, prt;
int nest = 0;    // счетчик открытых скобок
PNode Tree = new Node;
...
MinPrt = 100;
for ( i = first; i <= last; i ++ ) {
   if ( Expr[i] == '(' )       // открывающая скобка
      { nest ++; continue; }
   if ( Expr[i] == ')' )       // закрывающая скобка
      { nest --; continue; }
   if ( nest > 0 ) continue;   // пропускаем все, что в скобках
   prt = Priority ( Expr[i] );
   if ( prt <= MinPrt ) {
      MinPrt = prt;
      k = i;
      }
 }
if ( MinPrt == 100  &&  // все выражение взято в скобки
     Expr[first]== '('  &&  Expr[last]==')' ) {
    delete Tree;
    return MakeTree(Expr, first+1, last-1);
    }
...
return Tree;
}
```

Поскольку новый узел создается в самом начале функции, его надо удалить, если все выражение взято в скобки.

Многозначные числа и переменные

Для хранения многозначных чисел и имен переменных надо использовать массив символов в области данных узла.

```
struct Node {
        char data[40];
        Node *left, *right;
        };
typedef Node *PNode;
```

Будем по-прежнему считать, что выражение не содержит ошибок. Тогда, если в строке нет ни одного знака арифметической операции (вне скобок) и нет скобок по краям, все выражение представляет собой единый элемент (он называется **операндом**) – число или имя переменной. Для записи этого элемента в область данных узла используется функция **strncpy** , которая копирует заданное количество символов. Она не ставит символ конца строки, поэтому приходится делать это вручную.

```
PNode MakeTree (char Expr[], int first, int last)
{
int MinPrt, i, k, prt;
PNode Tree = new Node;

MinPrt = 100;
for ( i = first; i <= last; i ++ ) {
   prt = Priority ( Expr[i] );
   if ( prt <= MinPrt ) {
      MinPrt = prt;
      k = i;
      }
   }
if ( MinPrt == 100 )
    if ( Expr[first]== '('  &&  Expr[last]==')' ) {
       delete Tree;
       return MakeTree(Expr, first+1, last-1);
       }
    else {            // число или переменная
       k = last - first + 1;
       strncpy(Tree->data, Expr+first, k);
       Tree->data[k] = '\0';
       Tree->left = NULL;
       Tree->right = NULL;
       return Tree;
       }

Tree->data[0] = Expr[k];     //знак операции
Tree->data[1] = '\0';
Tree->left  = MakeTree (Expr,first,k-1);
Tree->right = MakeTree (Expr,k+1,last);

return Tree;
}
```

Если обнаружено число или переменная, сначала вычисляем ее длину и записываем в переменную **k**.

Для вычисления такого выражения по дереву надо несколько изменить функцию **CalcTree** с учетом того, что поле данных узла – символьная строка. Для преобразования чис-

ла из символьного вида в числовой используем стандартную функцию **atoi** (для этого надо подключить заголовочный файл **stdlib.h**).

```
int CalcTree (PNode Tree)
{
   int num1, num2;
   if ( ! Tree->left )            // если нет потомков,
      return atoi(Tree->data);    // раскодировали число
   // ... дальше все то же самое...
}
```

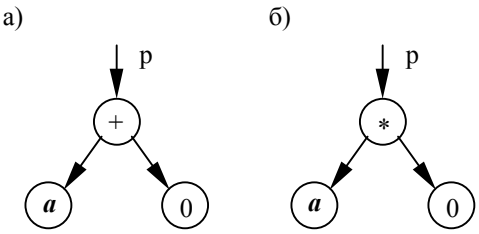
Конечно, для того, чтобы выражение можно было вычислить, оно не должно содержать имен переменных.

Упрощение выражения с помощью дерева

Некоторые выражения можно сразу значительно упростить, используя очевидные тождества, верные для любого ***x***:

```
0 + x = x       x + 0 = x       0 * x = 0
0 - x = - x     x - 0 = x       1 * x = x
```

Пусть, например, мы нашли такую структуру, как показано на рисунке *а*. Значение всего первого выражения равно **a**, поэтому нам надо сделать следующее: указатель **p** поставить на вершину **a**, а две ненужные вершины удалить из памяти.



В случае б) аналогично надо указатель **p** переставить на вершину со значением 0. при этом надо учесть, что второй узел (со значением **a**) может иметь потомков, которых также надо корректно удалить. Это делается рекурсивно:

```
void DeleteNode ( PNode Tree )
{
 if ( Tree == NULL ) return;
 DeleteNode ( Tree->left );
 DeleteNode ( Tree->right );
 delete Tree;
}
```

Кроме того, если оба сына какой-то вершины являются листьями и содержат числа, такое выражение можно сразу посчитать, также удалив два ненужных узла. Один из вариантов реализации этой операции приведен ниже. Здесь используется функция **IsNumber**, которая возвращает 1, если узел является листом и содержит число, и 0 в противном случае:

```
int IsNumber ( PNode Tree )
{
int i = 0;
if ( ! Tree ) return 0;    // пустое дерево
while ( Tree->data[i] )    // пока не дошли до конца строки
   if ( ! strchr("0123456789", Tree->data[i++]) )
      return 0;            // если не нашли цифру, выход
return 1;
}
```

Сама процедура вычисления выражения выглядит так:

```
void Calculate(PNode Tree)
{
int num1, num2, result = 0;

if ( ! Tree  ||   // если нельзя вычислить, выход
     ! IsNumber(Tree->left)  ||
     ! IsNumber(Tree->right) ) return;

num1 = atoi(Tree->left->data);   // получить данные от сыновей
num2 = atoi(Tree->right->data);

switch ( Tree->data[0] ) {       // выполнить операцию
   case '+': result = num1 + num2; break;
   case '-': result = num1 - num2; break;
   case '*': result = num1 * num2; break;
   case '/': result = num1 / num2; break;
   }
delete Tree->left;   // удалить ненужные поддеревья
delete Tree->right;
sprintf(Tree->data, "%d", result);
Tree->left = NULL;
Tree->right = NULL;
}
```

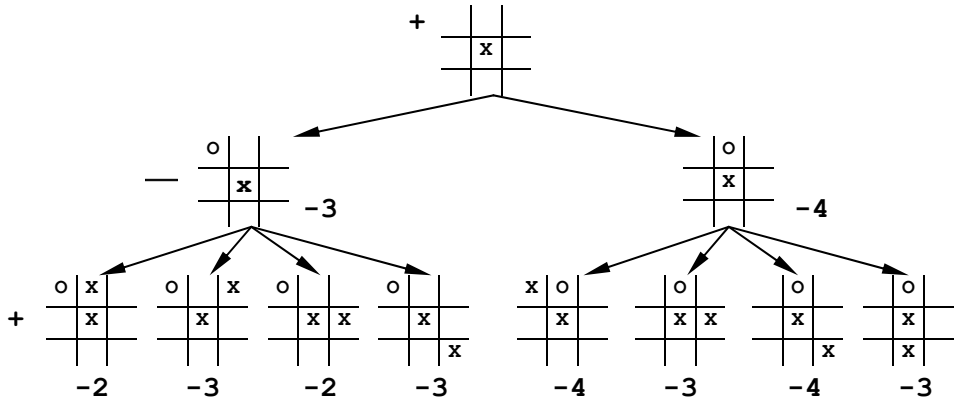
Дерево игр

Одно из применений деревьев - игры с компьютером. Рассмотрим самый простой пример – игру в крестики-нолики на поле 3 на 3.

Программа должны анализировать позицию и находить лучший ход. Для этого нужно определить **оценочную функцию**, которая получая позицию на доске и указание, чем играет игрок (крестики или нолики) возвращает число – оценку позиции. Чем она выше, тем более выгодна эта позиция для игрока. Примером такой функции может служить сумма строк, столбцов и диагоналей, которые может занять игрок минус такая же сумма для его противника.

Однако, в этой ситуации программа не ведет просчет вперед и не оценивает позиции, которые могут возникнуть из текущей. Это недостаточно для предсказания исхода игры. Хотя для крестиков-ноликов можно перебрать все варианты и найти выигрышную позицию, большинство игр слишком сложно, чтобы допускать полный перебор.

Выбор хода может быть существенно улучшен, если просматривать на несколько ходов вперед. **Уровнем просмотра** называется число будущих рассматриваемых ходов. Начиная с любой позиции можно построить дерево возможных позиций, получающихся после каждого хода игры. Для крестиков-ноликов ниже приведено дерево с уровнем просмотра 3 для того случая, когда крестики сделали первый ход в центр доски.



Обозначим игрока, который ходит в корневой позиции (в данном случае – нолики) знаком «плюс», а его соперника – знаком «минус». Попытаемся найти лучший ход для игрока «плюс» в этой позиции. Пусть все варианты следующих ходов были оценены для игрока «плюс». Он должен выбрать такой, в котором оценка максимальная для него.

С другой стороны, как только игрок «плюс» сделает свой ход, игрок «минус» из всех возможных ходов сделает такой, чтобы его оценка с позиции игрока «плюс» была минимальной. Поэтому значение минусового узла для игрока «плюс» равно минимальному из значений сыновей этого узла. Это означает, что на каждом шаге соперники делают наилучшие возможные ходы.

Для того, чтобы выбрать оптимальный ход в корне дерева, надо оценить позицию в его листьях. После этого каждому плюсовому узлу присваивается **максимальное** из значений его сыновей, а каждому минусовому – **минимальное**. Такой метод называется методом **минимакса**, так как по мере продвижения вверх используются попеременно функции максимума и минимума. Общая идея метода состоит в том, чтобы выбрать лучший ход на случай худших (для нас) действий противника. Таким образом, лучшим для ноликов в корневой позиции будет ход в угол.

4. Графы

Основные понятия[2]

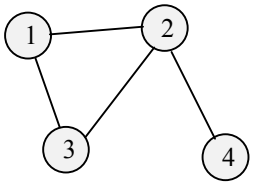
Определения

Во многих жизненных ситуациях старая привычка толкает нас рисовать на бумаге точки, обозначающие людей, города, химические вещества, и показывать линиями (возможно со стрелками) связи между ними. Описанная картинка называется **графом**.

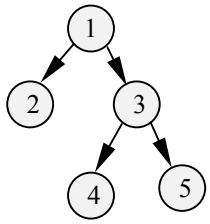
Граф - это совокупность **узлов** (**вершин**) и соединяющих их **ребер** (**дуг**).

Ниже показаны примеры графов

а)



б)



Если дуги имеют направление (вспомните улицы с односторонним движением), то такой граф называется **направленным** или **ориентированным** графом (**орграфом)**.

Цепью называется последовательность ребер, соединяющих две (возможно не соседние) вершины ***u*** и ***v***. В направленном графе такая последовательность ребер называется «**путь**».

Граф называется **связным**, если существует цепь между любой парой вершин. Если граф не связный, то его можно разбить на **k** связных компонент – он называется **k-**связным.

В практических задачах часто рассматриваются **взвешенные графы**, в которых каждому ребру приписывается **вес** (или длина). Такой граф называют **сетью**.

Циклом называется цепь из какой-нибудь вершины **v** в нее саму.

Деревом называется граф без циклов.

Полным называется граф, в котором проведены все возможные ребра (для графа, имеющего **n** вершин таких ребер будет **n(n-1)/2.**

Описание графов

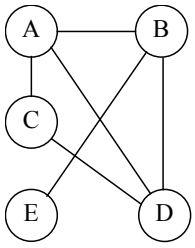
Для описания графов часто используют два типа матриц – **матрицу смежности** (для невзвешенных графов) и **весовую матрицу** (для взвешенных).

Матрица смежности графа с **N** вершинами – это матрица размером **N** на **N**, где каждый элемент с индексами **(i,j)** является логическим значением и показывает, есть ли дуга из вершины **i** в вершину **j**.

Часто вместо логических значений (истина/ложь) используют целые числа (1/0). Для неориентированных графов матрица смежности всегда симметрична относительно главной диагонали (рисунок а). Для ориентированных графов (рисунок б) это не всегда так, потому что может существовать путь из вершины **i** в вершину **j** и не существовать обратного пути.

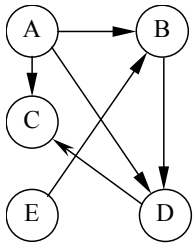
[2] Этот раздел написан на базе материала книги В.М. Бондарев, В.И. Рублинецкий, Е.Г. Качко. Основы программирования. – Харьков: «Фолио», 1998.

а)



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	0	0
E	0	1	0	0	0

б)

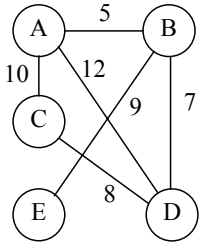


	A	B	C	D	E
A	0	1	1	1	0
B	0	0	0	1	0
C	0	0	0	0	0
D	0	0	1	0	0
E	0	1	0	0	0

Для взвешенных графов недостаточно просто указать, есть ли связь между вершинами. Требуется еще хранить в памяти «вес» каждого ребра, например, стоимость проезда или длину пути. Для этого используется **весовая матрица**.

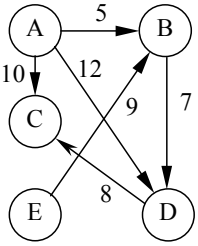
Весовая матрица графа с **N** вершинами – это матрица размером **N** на **N**, где каждый элемент с индексами **(i,j)** равен «весу» ребра из вершины **i** в вершину **j**.

а)



	A	B	C	D	E
A	0	5	10	12	0
B	5	0	0	7	9
C	10	0	0	8	0
D	12	7	8	0	0
E	0	9	0	0	0

б)



	A	B	C	D	E
A	0	5	10	12	0
B	0	0	0	7	0
C	0	0	0	0	0
D	0	0	8	0	0
E	0	9	0	0	0

Задача Прима-Краскала

Формулировка задачи

Задача. Дана плоская страна и в ней **n** городов с известными координатами. Нужно соединить все города телефонной сетью так, чтобы длина телефонных линий была минимальная.

Город будем изображать узлом (точкой). Телефонные линии могут разветвляться только на телефонных станциях, а не в чистом поле. Поскольку требуется линия минимальной общей длины, в ней не будет циклов, потому что иначе можно было бы убрать одно звено цикла и станции по-прежнему были бы связаны. В терминах теории графов эта задача звучит так:

Дан граф с **n** вершинами; длины ребер заданы матрицей $\{d_{ij}\}$, **i,j=1..n**. Найти набор ребер, соединяющий все вершины графа (он называется **остовным деревом**) и имеющий минимальную длину

Эта задача – одна из тех немногих, для которых известно точное и несложное решение, причем алгоритм предельно прост.

Жадные алгоритмы

Представим себе зимовщика, которому предоставили некоторый запас продуктов на всю зиму. Конечно, он может сначала съесть все самое вкусное – шоколад, мясо и т.п., но за такой подход придется жестоко расплачиваться в конце зимовки, когда останется только соль и маргарин.

Подобным образом, если оптимальное решение строится по шагам, обычно нельзя выбирать на каждом этапе «самое вкусное» – за это придется расплачиваться на последних шагах. Такие алгоритмы называют **жадными**.

Решение

Удивительно, но для непростой задачи Прима-Краскала жадный алгоритм дает точное оптимальное решение. Алгоритм формулируется так:

> В цикле **n-1** раз выбрать из оставшихся ребер самое короткое ребро, которое не образует цикла с уже выбранными.

Как же проследить, чтобы не было циклов? Оказывается очень просто: в самом начале покрасим все вершины в разные цвета и затем, выбрав очередное ребро между вершинами **i** и **j**, где **i** и **j** имеют разные цвета, перекрасим вершину **j** и все соединенные с ней (то есть имеющие ее цвет) в цвет **i**. Таким образом, при выборе ребер, соединяющих вершины разного цвета, цикл не возникнет никогда, а после **n-1** шагов все вершины будут иметь один цвет.

Для записи информации о ребрах введем структуру

```
struct rebro { int i, j; }; // ребро соединяет вершины i и j
```

причем будем считать, что в паре номер первой вершины **i** меньше, чем номер второй **j**.

Приведенная ниже программа действует по следующему «жадному» алгоритму/

1. Покрасить все вершины в разные цвета.
2. Сделать **n-1** раз в цикле
   - выбрать ребро **(i,j)** минимальной длины, соединяющее вершины разного цвета;
   - запомнить его в массиве ребер;
   - перекрасить все вершины, имеющие цвет **j**, в цвет **i**.
3. Вывести результат.

```
const int N = 5;

void main()
{
int D[N][N], Col[N], i, j, k, Dmin, jMin, iMin, col_j;
rebro Reb[N-1];
 // здесь надо ввести матрицу D
for ( i = 0; i < N; i ++ ) // покрасить все вершины
   Col[i] = i;             // в разные цвета
for ( k = 0; k < N-1; k ++ ) {
 Dmin = 30000;
 for ( i = 0; i < N-1; i ++ )
   for ( j = i+1; j < N; j ++ )
      if ( Col[i] != Col[j] &&  // ищем самое короткое ребро,
           D[i][j] < Dmin ) {   // не образующее цикла
         Dmin = D[i][j];
         iMin = i;
         jMin = j;
         }
 Reb[k].i = iMin;   // запомнить найденное ребро
 Reb[k].j = jMin;
 col_j = Col[jMin];
 for ( i = 0; i < N; i ++ )    // перекрасить все вершины, цвет
    if ( Col[i] == col_j )     // которых совпал с цветом
          Col[i] = Col[iMin]; // вершины jMin
 }
// здесь надо вывести найденные ребра
}
```

Дополнительно можно рассчитывать общую длину выбранных ребер, но это не меняет принципиально алгоритм. Этот алгоритм требует памяти порядка n^2 (обозначается $O(n^2)$, то есть при увеличении n в 2 раза объем требуемой памяти увеличивается в 4 раза). Его временная сложность – $O(n^3)$, то есть при увеличении n в 2 раза время вычислений увеличивается в 8 раз (надо просмотреть $O(n^3)$ чисел и сделать это $n-1$ раз).

Кратчайший путь

Формулировка задачи

Задача. Задана сеть дорог между населенными пунктами (часть из них могут иметь одностороннее движение). Требуется найти кратчайший путь между двумя заданными пунктами.

Обычно задачу несколько расширяют и находят сразу кратчайшие пути от заданной вершины ко всем остальным. В терминах теории графов задача выглядит так:

В сети, где часть дорог имеет одностороннее движение, найти кратчайшие пути от заданной вершины ко всем остальным.

Алгоритм Дейкстры

Ниже описывается алгоритм, предложенный Дейкстрой в 1959 г. Дана матрица $\{d_{ij}\}$ длин дуг между вершинами, если дуги между вершинами i и j нет, то $d_{ij}=\infty$. Если сеть образуют n вершин, то для реализации алгоритма требуется три массива длиной n:

1. Массив $\{a_i\}$, в котором $a_i=0$, если вершина i еще не рассмотрена, и $a_i=1$, если вершина i уже рассмотрена.
2. Массив $\{b_i\}$, в котором b_i – текущее кратчайшее расстояние от выбранной стартовой вершины x до вершины i.
3. Массив $\{c_i\}$, в котором c_i – номер предпоследней вершины в текущем кратчайшем пути из выбранной стартовой вершины x до вершины i.

Сам алгоритм состоит из трех этапов: инициализации, основного цикла и вывода результата.

Инициализация

Пусть x – номер выбранной стартовой вершины.

1. Заполним весь массив a значением 0 (пока ни одна вершина не рассмотрена).
2. В i-ый элемент массива b запишем расстояние от вершины x до вершины i (если пути нет, то оно равно ∞, в программе укажем очень большое число).
3. Заполним весь массив c значением x (пока рассматриваем только прямые пути от x к i).
4. Рассмотрим стартовую вершину: присвоим $a[x]=1$ и $c[x]=0$ (начало всех путей).

Основной цикл

Среди нерассмотренных вершин (для которых $a[i]=0$) найти такую вершину j, что расстояние b_j – минимальное. Рассмотреть ее:
1) установить $a[j]=1$ (вершина рассмотрена);
2) сделать для всех вершин k:
если путь от вершины x к вершине k через j короче, чем уже найденный кратчайший путь (то есть $b_j+d_{jk}<b_k$), то запомнить его: $c[k]=j$ и $b_k=b_j+d_{jk}$.

Вывод результата

Можно доказать, что в конце такой процедуры массив **b**, будет содержать кратчайшие расстояния от стартовой вершины **x** до вершины **i** (для всех **i**). Однако хотелось бы еще получить сам оптимальный путь.

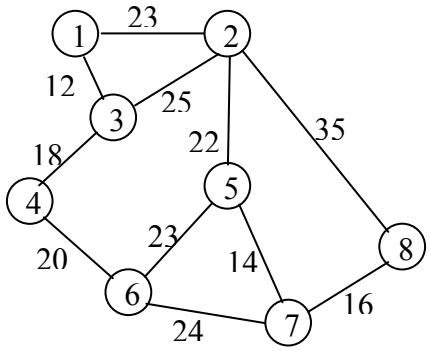
Оказывается, массив **c** содержит всю необходимую информацию для решения этой задачи. Предположим, что надо вывести оптимальный путь из вершины **x** в вершину **i**. По построению предпоследняя вершина в этой цепи имеет номер **z=c[i]**. Теперь надо найти оптимальный путь в вершину **z** тем же способом. Таким образом, путь «раскручивается» с конца. Когда мы получили **z=0**, путь закончен, мы вернулись в стартовую вершину. В виде алгоритма это можно записать так:

1) установить **z=i**;
2) пока **c[z]** не равно нулю,
   - **z=c[z];**
   - вывести **z.**

Для выполнения алгоритма надо **n** раз просмотреть массив **b** из **n** элементов, поэтому он имеет квадратичную сложность $O(n^2)$.

Пример

Пусть дана сеть дорог, показанная на рисунке справа. Найдем кратчайшие расстояния от вершины 3 до всех остальных вершин. Покажем ход выполнения алгоритма Дейкстры по шагам.



- **Инициализация.** В результате этого шага получаем такие матрицы:

	1	2	3	4	5	6	7	8
a	0	0	1	0	0	0	0	0
b	12	25	0	18	∞	∞	∞	∞
c	3	3	0	3	3	3	3	3

- **Основной цикл.** Последовательно массивы преобразуются к виду:

1) $\min b_j = 12$ для **j = 1**

	1	2	3	4	5	6	7	8
a	1	0	1	0	0	0	0	0
b	12	25	0	18	∞	∞	∞	∞
c	3	3	0	3	3	3	3	3

2) $\min b_j = 18$ для **j = 4**

	1	2	3	4	5	6	7	8
a	1	0	1	1	0	0	0	0
b	12	25	0	18	∞	38	∞	∞
c	3	3	0	3	3	4	3	3

3) $\min b_j = 25$ для **j = 2**

	1	2	3	4	5	6	7	8
a	1	1	1	1	0	0	0	0
b	12	25	0	18	47	38	∞	60
c	3	3	0	3	2	4	3	2

4) $\min b_j = 38$ для **j = 6**

	1	2	3	4	5	6	7	8
a	1	1	1	1	0	1	0	0
b	12	25	0	18	47	38	62	60
c	3	3	0	3	2	4	6	2

5) $\min b_j = 47$ для **j = 5**

	1	2	3	4	5	6	7	8
a	1	1	1	1	1	1	0	0
b	12	25	0	18	47	38	61	60
c	3	3	0	3	2	4	5	2

6) $\min b_j = 60$ для **j = 8**

	1	2	3	4	5	6	7	8
a	1	1	1	1	1	1	0	1
b	12	25	0	18	47	38	61	60
c	3	3	0	3	2	4	5	2

- **Вывод кратчайшего пути.** Найдем, например, кратчайший путь из вершины **3** в вершину **8**. «Раскручивая» массив **c**, получаем **8←2←3**.

Алгоритм Флойда-Уоршелла

Если надо только вычислить кратчайшие расстояния между всеми вершинами, но не требуется знать сами кратчайшие маршруты, можно использовать весьма элегантный и простой алгоритм Флойда-Уоршелла:

```
for (k = 0; k < n; k ++)
 for (i = 0; i < n; i ++)
  for (j = 0; j < n; j ++)
    if ( d[i][j] > d[i][k]+d[k][j] ) {
        d[i][j] = d[i][k]+d[k][j];
        p[i][j] = p[k][j];
        }
```

Сначала в матрице $\{d_{ij}\}$ записаны расстояние между вершинами напрямую. Затем рассмотрим все пути, которые проходят через первую вершину. Если путь через нее короче, чем напрямую, то заменяем значение в матрице на новый кратчайший путь. В конце элемент матрицы $\{d_{ij}\}$ содержит длину кратчайшего пути из **i** в **j**. Каждая строка матрицы $\{p_{ij}\}$ сначала заполняется так, как массив **c** в алгоритме Дейкстры, а в конце элемент $\{p_{ij}\}$ равен номеру предпоследней вершины для кратчайшего пути из вершины **i** в вершину **j**.

Оптимальное размещение

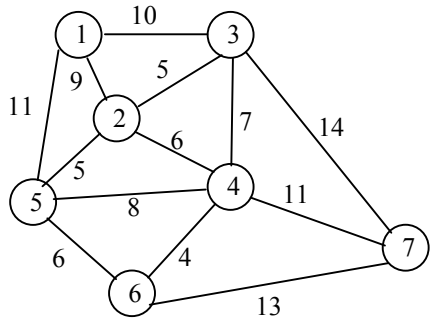
Задача на минимум суммы

Задача. Имеется **n** населенных пунктов, в каждом из которых живет p_i школьников **(i=1,...,n)**. Надо разместить школу в одном из них так, чтобы общее расстояние, проходимое всеми учениками по дороге в школу, было минимальным.

Эта задача просто решается на основе алгоритма Флойда-Уоршелла. Сначала получаем матрицу минимальных путей из каждой вершины в каждую $\{d_{ij}\}$. Пусть школа размещается в вершине с номером **k**. Тогда общее расстояние, которое должны пройти ученики из пункта **j** по дороге в школу, равно произведению расстояния между вершинами **i** и **j** на количество учеников в вершине **j**, то есть $d_{ij}\ p_j$. Суммируя эти произведения для всех вершин, найдем общее расстояние, которое будут проходить ученики, если школа будет в пункте **i**. Естественно, надо выбрать такую вершину **k**, для которой это число будет наименьшим.

Рассмотрим сеть, показанную на рисунке а). Цифры у дуг показывают расстояние между населенными пунктами. Количество учеников в пунктах 1..7 задано числами (80, 100, 140, 90, 60, 50, 40).

а)



б)

	1	2	3	4	5	6	7	
1	0	9	10	15	11	17	24	6120
2	9	0	5	6	5	10	17	3440
3	10	5	0	7	10	11	14	3640
4	15	6	7	0	8	4	11	3800
5	11	5	10	8	0	6	19	4560
6	17	10	11	4	6	0	13	4930
7	24	17	14	11	19	13	0	8360

На рисунке б) показана матрица длин кратчайших путей. В последнем столбце сосчитано общее расстояние, проходимое учениками, если школа будет в данной вершине. Поскольку надо выбрать наименьшее расстояние, то школу надо размещать в вершине 2.

Минимаксная задача размещения

Имеется **n** населенных пунктов, в одном из которых надо разместить спецподразделение, которое должно выезжать на вызовы при срабатывании сигнализации на одном из охраняемых объектов. Надо расположить его так, чтобы самый дальний охраняемый объект достигался за минимальное время.

Прежде всего, находится матрица кратчайших путей. Затем в каждой строке **i** выбирается максимальное число – это будет расстояние до самого отдаленного охраняемого объекта, если пункт полиции разместить в вершине **i**. Затем выбирается такой пункт **j**, для которого это число наименьшее.

Более сложной является минимаксная задача, в которой допускается размещать объект **на ребрах сети** (между пунктами). Для решения этой задачи можно использовать следующие идеи.

Рассмотрим ребро между вершинами **i** и **j**. Если расположить пункт полиции на нем, то расстояние от него до другой вершины **k** не может быть меньше, чем

$$\delta_k = \min(d_{ik}, d_{jk})$$

Вычислив эти величины для всех вершин **k**, определим максимальную из них. Это число δ_0 называется **нижней оценкой ребра (i,j)**, поскольку при любом выборе места для пункта полиции на этом ребре расстояние до самого дальнего объекта не будет меньше δ_0. Мы уже знаем минимально возможное расстояние до самого дальнего пункта при размещении отряда в одном из городов. Очевидно, имеет смысл рассматривать только те ребра, для которых нижняя оценка не превышает это расстояние.

Для каждого ребра **(i,j)** с нижней оценкой, меньше уже полученной, сделать следующие операции:

1. найти самую дальнюю вершину **k** из всех тех вершин, в которые надо ехать через вершину **i** (то есть $d_{ik} > d_{jk}$) ;
2. найти самую дальнюю вершину **l** из всех тех вершин, в которые надо ехать через вершину **j** (то есть $d_{il} < d_{jl}$) ;
3. вычислить середину расстояния между вершинами **k** и **l**; если она попадает на ребро **(i,j)** и расстояние до самых дальних вершин меньше уже найденного, запомнить это место

Ниже этот алгоритм записан на на языке Си:

```
minDist = 32767;  // большое число
for (i = 0; i < N-1; i ++ )
 for (j = i+1; j < N; j ++ ) { // перебрать все ребра (i,j)
  maxDi = maxDj = 0;
  for (k = 0; k < N; k ++ ) {  // перебрать все вершины
    if ( k == i  ||  k == j ) continue;
    if (D[j,k] < D[i,k])
      if (D[j,k] < maxDj) maxDj = D[j,k];
    else
      if (D[j,k] < maxDj) maxDj = D[j,k];
    }
 dist = (maxDi + D[i][j] + maxDj) / 2;
 if ( dist < minDist  &&  dist > maxDi  &&
      dist < maxDi+D[i][j] ) {
```

```
        minDist = dist; // запомнить новый вариант
        ...
        }
 }
```

Задача коммивояжера

Формулировка задачи

Эта знаменитая задача была сформулирована в 1934 г. В области дискретной оптимизации она стала испытательным полигоном, на котором проверяются новые методы. Проблема заключается в том, что эта задача относится к многочисленному классу задач, которые теоретически не решаются иначе, чем перебором.

Задача коммивояжера. Коммивояжер (бродячий торговец) должен выйти из первого города и, посетив по разу в неизвестном порядке города `2,3,...n`, вернуться обратно в первый город. В каком порядке надо обходить города, чтобы замкнутый путь (тур) коммивояжера был кратчайшим?

Метод грубой силы

Этот метод предусматривает просто перебор всех вариантов. Сначала нам надо получить некоторую допустимую последовательность обхода вершин. Поскольку первый город (1) зафиксирован, всего таких перестановок будет `(n-1)!`. Получив очередную перестановку надо найти длину пути и, если она окажется меньше уже найденного кратчайшего пути, ее надо запомнить и перейти к следующей перестановке.

Поскольку нами уже написана процедура получения всех перестановок из целых чисел от `1` до `N` (см. пункт *Комбинации и перестановки* раздела *Рекурсия* в главе III), имеет смысл ее использовать. Удобно ввести глобальные переменные

```
int Pmin[N],    // лучшая перестановка
    P[N],       // текущая перестановка
    Lmin,       // минимальная длина
    L,          // текущая длина
    D[N][N];    // матрица расстояний
```

Основная программа будет инициализировать глобальные переменные, вызывать рекурсивную процедуру построения оптимального пути и распечатывать результат:

```
void main()
{
Lmin = 32767;   // большое число
L = 0;
P[0] = 1;       // начальная вершина 1
Commi(1);       // построить тур
for ( int i = 0; i < N; i ++ )  // вывести результат
 printf("%d ", Pmin[i]);
}
```

Ниже приведена рекурсивная процедура, решающая задачу коммивояжера, приводится ниже. Текущая длина пути вычисляется последовательно с добавление каждого нового звена.

```
void Commi( int q ) // q - число уже поставленных вершин
{
int i, temp;
```

```
  if ( q == N ) {     // перестановка получена
     if ( L < Lmin ) {
     Lmin = L;
     for ( i = 0; i < N; i ++ ) // запомнить новый минимальный тур
        Pmin[i] = P[i];
     }
     return;
     }
  for ( i = q; i < N; i ++ ) {
     temp = P[q]; P[q] = P[i]; P[i] = temp; // P[q] <-> P[i]
     L += D [P[q-1]] [P[q]];   // добавить ребро
     Commi ( q+1 );            // рекурсивный вызов
     L -= D [P[q-1]] [P[q]];   // убрать ребро
     temp = P[q]; P[q] = P[i]; P[i] = temp; // P[q] <-> P[i]
     }
  }
```

Главная трудность такого переборного способа заключается в том, что для больших **n** (например, для **n>20**) число комбинаций настолько велико, что перебор нереален.

Однако, часто можно найти такую стратегию перебора, при которой сразу отбрасывается целая группа вариантов, которые заведомо не лучше уже найденного минимального решения. Одним из таких алгоритмов является метод ветвей и границ.

Метод ветвей и границ

Идея этого подхода проста и естественна: если частичная длина пути стала больше длины самого лучшего из уже найденных туров, дальнейшую проверку делать бессмысленно. При этом второй цикл **for** в процедуре приобретает вид

```
  for ( i = q; i < N; i ++ ) {
     temp = P[q]; P[q] = P[i]; P[i] = temp;
     L += D [P[q-1]] [P[q]];
     if ( L < Lmin ) Commi ( q+1 );
     L -= D [P[q-1]] [P[q]];
     temp = P[q]; P[q] = P[i]; P[i] = temp;
     }
```

Все варианты туров можно изобразить в виде дерева с корнем в вершине 1. На первом уровне находятся все вершины, в которые можно идти из вершины 1 и т.д. Если мы получили **L>Lmin**, то сразу отсекается целая ветка такого дерева.

Алгоритм Литтла

Эффективность метода ветвей и границ очень сильно зависит от порядка рассмотрения вариантов. При удачном выборе стратегии перебора можно сразу же найти решение, близкое к оптимальному, которое позволит «отсекать» очень большие ветви, для которых нижняя оценка больше, чем стоимость уже найденного кратчайшего тура. Литтл с соавторами на основе метода ветвей и границ разработали удачный алгоритм, который позволяет во многих случаях быстро решить задачу коммивояжера.

Идея очень проста – разбить все варианты на классы и получить **оценки снизу** для каждого класса (оценкой снизу называется такое число, что стоимость любого варианта из данного класса заведомо не может быть ниже него).

Продемонстрируем на примере применение метода Литтла. Пусть есть 6 городов и матрица а) задает стоимость c_{ij} проезда из города **i** в город **j**. Прочерк означает, что из города **i** в город **i** ходить нельзя.

а)

	1	2	3	4	5	6
1	-	6	4	8	7	14
2	6	-	7	11	7	10
3	4	7	-	4	3	10
4	8	11	4	-	5	11
5	7	7	3	5	-	7
6	14	10	10	11	7	-

б)

	1	2	3	4	5	6
1	-	2	0	4	3	10
2	0	-	1	5	1	4
3	1	4	-	1	0	7
4	4	7	0	-	1	7
5	4	4	0	2	-	4
6	7	3	3	4	0	-

в)

	1	2	3	4	5	6
1	-	0	0	3	3	6
2	0	-	1	4	1	0
3	1	2	-	0	0	3
4	4	5	0	-	1	3
5	4	2	0	1	-	0
6	7	1	3	3	0	-

Предположим, что добрый мэр города **j** постановил выплачивать всем, кто въедет в его город 5 долларов. При этом из каждого элемента **j**-ого столбца матрицы надо вычесть 5. Так как в **каждом** туре надо въехать в город **j** ровно **один** раз, то все туры подешевеют одинаково, и оптимальный тур останется оптимальным. Так же если мэр города **i** будет выплачивать 5 долларов каждому **выехавшему,** из каждого элемента **i**-ой строки надо вычесть 5. Но все туры подешевеют одинаково, и это снова не повлияет на результат.

> Если из всех элементов любой строки или столбца вычесть одинаковое число, то минимальный тур остается минимальным

Теперь вычтем минимальный элемент из каждой строки и каждого столбца матрицы а) (эта операция называется **приведением по строкам**) и получим матрицу б) (надо вычесть числа 4, 6, 3, 4, 3 и 7 из строк 1-6 соответственно). Затем из каждого столбца вычтем минимальный в нем элемент (числа 2, 1 и 4 из столбцов 2, 4 и 6, соответственно). После такого **приведения по столбцам** получим матрицу в). Если нам удастся найти минимальный тур в этой матрице, то он же будет минимальным и в исходной, только к его стоимости надо прибавить сумму всех чисел, которые мы вычитали из строк и столбцов, то есть 34. Поскольку в матрице в) нет отрицательных чисел, то стоимость минимального тура в ней больше или равна нулю, поэтому **оценка снизу** всех туров равна 34.

Теперь начинается основной шаг алгоритма. Выполним **оценку нулей** следующим образом. Рассмотрим нуль в клетке (1,2) матрицы в). Он означает, что цена перехода из 1 в 2 равна 0. А если мы не пойдем из 1 в 2, то все равно придется въехать в 2 (за цены, указанные в столбце 2 – дешевле всего за 1 из города 6). Также придется выехать из города 1 (за цену, указанную в первой строке – дешевле всего за 0) в город 3. Таким образом, если не ехать из 1 в 2 «по нулю», надо заплатить минимум 1. Это и есть «**оценка нуля**».

В матрице г) поставлены все оценки нулей, которые не равны нулю. Выберем максимальную из этих оценок (в примере - любой нуль с оценкой 1, например в клетке (1,2)).

г)

	1	2	3	4	5	6
1	-	0^1	0	3	3	6
2	0^1	-	1	4	1	0
3	1	2	-	0^1	0	3
4	4	5	0^1	-	1	3
5	4	2	0	1	-	0
6	7	1	3	3	0^1	-

д)

	1	3	4	5	6
2	-	1	4	1	0
3	1	-	0^1	0	3
4	4	0^1	-	1	3
5	4	0	1	-	0
6	7	3	3	0^1	-

е)

	1	3	4	5	6
2	-	1	4	1	0^1
3	0^3	-	0^1	0	3
4	3	0^1	-	1	3
5	3	0	1	-	0
6	6	3	3	0^1	-

Пусть выбрано ребро (1,2). Разобьем все туры на 2 класса – включающие ребро (1,2) и не включающие его. Про второй класс можно сказать, что туры этого класса «стоят» 35 и более – это нижняя граница класса. Чтобы исследовать дальше первый класс, вычеркнем из матрицы г) первую строку и второй столбец – получим матрицу д). Чтобы не было цикла, надо поставить запрет в клетке (2,1). Эту матрицу можно привести на 1 по первому столбцу, таким образом, оценка этого класса – 35. Оценим нули в этой матрице (см. матрицу е)).

Снова выбираем клетку (3,1) с наибольшей оценкой нуля, равной 3. Оценка для класса «без (3,1)» равна 35+3=38. Вычеркиваем строку 3 и столбец 1 и ставим запрет на преждевременное замыкание (2,3) (матрица ж)).

ж)

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

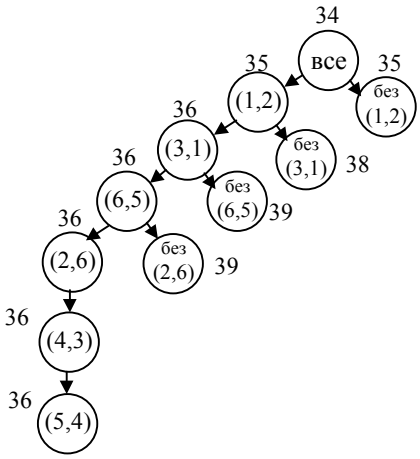
з)

	3	4	5	6
2	-	3	1	0^1
4	0^1	-	1	3
5	0	0^2	-	0
6	3	2	0^3	-

и)

	3	4	6
2	-	3	0^3
4	0^3	-	3
5	0	0^3	-

Эта матрица приводится по столбцу 4 на 1. Таким образом, получаем матрицу з) и оценку класса 35+1=36. Нуль с максимальной оценкой 3 находится в клетке (6,5). отрицательный вариант имеет оценку 36+3=39. Для оценки положительно варианта вычеркиваем столбец 5 и строку 6, а также ставим запрет в клетку (5,6). Получим матрицу и), которая неприводима и не увеличивает оценку класса. Далее получаем ветвление по выборы ребра (2,6), отрицательный вариант имеет оценку 36+3=39. Далее вычеркиваем строку 2 и столбец 6 и ставим запрет в клетку (5,3) - получается матрица к). Теперь, когда осталась матрица 2 на 2 с запретами по диагонали, достраиваем тур ребрами (4,3) и (5,4). Мы получили тур 1-2-6-4-3-1 стоимостью в 36. На рисунке справа показано дерево ветвлений и оценки вариантов.



Теперь можно начать отбрасывать варианты. Все классы, имеющие оценку 36 и выше, лучшего тура не содержат. Поэтому отбрасываем все вершины дерева перебора, имеющие оценку 36 и выше, а также вершины, у которых «убиты» оба потомка. Осталось проверить, не содержит ли лучшего тура класс «без (1,2)». Для этого в приведенной матрице в) поставим запрет в клетку (1,2), приведем столбец 2 на 1 и оценим нули. Полученная матрица л) приведена ниже.

л)

	1	2	3	4	5	6
1	-	-	0^3	3	3	6
2	0^1	-	1	4	1	0
3	1	1	-	0^1	0	3
4	4	4	0^1	-	1	3
5	4	1	0	1	-	0
6	7	0^1	3	3	0	-

м)

	1	2	4	5	6
2	0	-	4	0	1
3	-	1	0	0	3
4	4	4	-	1	3
5	4	1	1	-	0
6	7	0	3	0	-

Оценка нулей дает 3 для (1,3), поэтому стоимость отрицательного варианта 35+3=38 превосходит стоимость уже полученного тура и отрицательный вариант отсекается. Для оценки положительного варианта вычеркиваем первую строку и 3-й столбец, ставим запрет в (3,1) и получаем матрицу м). Она приводится по строке 4 на 1, поэтому оценка положительного варианта равна 35+1=36 и этот класс также отбрасывается. Таким образом мы получили минимальный тур.

Хотя теоретических оценок быстродействия этого и подобных алгоритмов не получено, на практике они часто позволяют решить задачи с **`n=100`**.

Метод случайных перестановок

Для того, чтобы приближенно решать задачу коммивояжера для больших **`n`**, на практике часто используют метод случайных перестановок. В алгоритме повторяются следующие шаги:

1. Выбрать случайным образом номера вершин **`i`** и **`j`** в перестановке.
2. Если при перестановке вершин с номерами **`i`** и **`j`** длина пути уменьшилась, такая перестановка принимается.

Такой метод не гарантирует, что будет найдено точное решение, но очень часто позволяет найти приемлемое решение за короткое время в тех случаях, когда другие алгоритмы неприменимы.

Задача о паросочетаниях

Формулировка задачи

Задача. Есть **m** мужчин и **n** женщин. Каждый мужчина указывает несколько (от 0 до **n**) женщин, на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от 0 до **m**), за которых она согласна выйти замуж. Требуется заключить наибольшее количество моногамных браков.

Для формулировки задачи в терминах теории графов надо ввести несколько определений.

Двудольным называется граф, все вершины которого разбиты на две доли (части), а ребра проходят только между вершинами разных частей.

Паросочетанием называется множество ребер, не имеющих общих вершин.

Таким образом, эта задача равносильна следующей:

Задача о наибольшем паросочетании. В заданном двудольном графе найти наибольшее паросочетание.

Для описания графа введем матрицу возможных браков **A={a_{ij}}**, где **a_{ij}=1**, если возможен брак между мужчиной **i (i=1..m)** и женщиной **j (j=1..n)** (оба согласны) и **a_{ij}=0**, если брак невозможен.

Достижимость вершин в графе

Для начала решим вспомогательную задачу, которая понадобится в дальнейшем.

Задача о достижимости. Найти, какие вершины достижимы в заданном ориентированном графе из данной вершины **s**.

Эта задача решается с помощью алгоритма, очень похожего на алгоритм Дейкстры. Заведем три вспомогательных массива **Q**, **R** и **P** размером **n**, где **n** – число вершин:

- в массив **Q** будем записывать номера вершин в порядке их просмотра (начиная с **s**);
- в массиве **R** элемент **R_i=1**, если уже найден путь из **s** в **i**, и **R_i=0**, если ни один путь не найден;
- в массиве **P** элемент **P_i** есть номер предпоследней вершины на пути от **s** к **i**, или **P_i=-1**, если ни один путь не найден.

Две переменные **a** и **z** обозначают начало и конец «рабочей области» массива **Q**. Теперь можно привести алгоритм. Он состоит из двух частей.

Инициализация

Начать с вершины **s**, ни одна из вершин не рассмотрена:

1. Присвоить `a=0; z=0; Q[0]=s;`

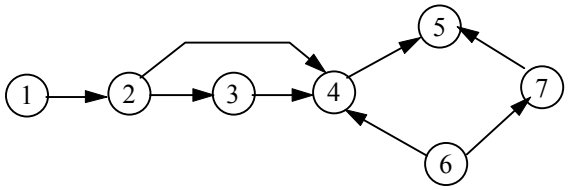
2. Для всех **i** присвоить `R[i]=0, P[i]=-1;`

Общий шаг

В цикле рассмотреть все вершины орграфа, которые достижимы непосредственно из **Q[a]**. Если путь до вершины **k** еще не был найден (то есть R_k=0), сделать
1. `z ++; Q[z] = k;  P[k] = Q[a];  R[k] = 1;`
2. `a ++;`

Повторять общий шаг пока **a <= z.**

Покажем работу алгоритма на примере. Найти вершины данного графа, достижимые из вершины 1. Последовательно имеем



Рассмотрена вершина 2:
- `Q = {1, 2}`
- `R = [0, 1, 0, 0, 0, 0, 0]`
- `P = [-1, 1, -1, -1, -1, -1, -1]`

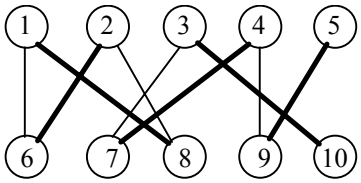
Рассмотрены вершины 3 и 4:
- `Q = 1, 2, {3, 4}`
- `R = [0, 1, 1, 1, 0, 0, 0]`
- `P = [-1, 1, 2, 2, -1, -1, -1]`

Рассмотрена вершины 5:
- `Q = 1, 2, 3, 4, {5}`
- `R = [0, 1, 1, 1, 1, 0, 0]`
- `P = [-1, 1, 2, 2, 4, -1, -1]`

После этого новых достижимых вершин нет, рабочая зона массива **Q** (в фигурных скобках) закончилась.

Массив **P** можно «раскрутить» как в алгоритме Дейкстры. Например, вершина 5 достижима из 4, вершина 4 – из 4, а 2 – из 1, поэтому существует путь 1–2–4–5.

Метод чередующихся цепей



Задача о наибольшем паросочетании решается методом чередующихся цепей. Пусть **M** – некоторое паросочетание в двудольном графе **B**. Ребра, которые входят в **M**, будем обозначать толстыми линиями, а все остальные – тонкими. Цепь, в которую входят поочередно жирные и тонкие ребра, будем называть чередующейся. Например, цепь (1, 6, 2, 8) – чередующаяся. По определению цепь из одного ребра тоже чередующаяся.

Вершины, которые связаны с жирными ребрами (участвующими в паросочетании), называются **занятыми** или **насыщенными**). Остальные вершины называются **свободными** или **ненасыщенными**.

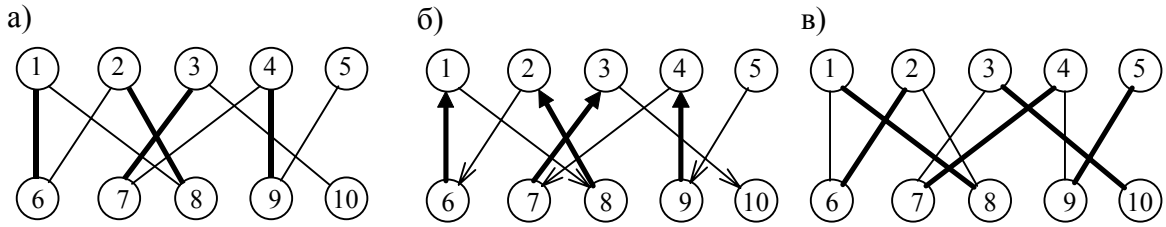
Очевидно, что если в графе есть цепь, которая начинается и заканчивается на свободной вершине, то в ней тонких ребер на 1 больше, чем жирных. Поэтому такую цепь можно «перекрасить», то есть заменить все жирные ребра тонкими и наоборот. При этом пар станет на 1 больше, то есть решение улучшится. Такая цепь называется **увеличивающейся**.

Можно доказать, что паросочетание является наибольшим тогда и только тогда, когда в графе нет увеличивающихся цепей. На этой идее основан следующий алгоритм.

1. Построим начальное паросочетание «жадным» алгоритмом: будем брать по очереди незанятые вершины первой части и, если существует допустимое ребро, вводить его, не думая о последствиях.
2. Строим ориентированный граф так: все дуги, вошедшие в паросочетание («жирные») направляем вверх, а все остальные – вниз.

3. Просматриваем все свободные вершины первой части. Если среди достижимых вершин есть незанятая вершина из второй части, то есть увеличивающаяся цепь. Ее надо увеличить и перейти снова к пункту 2.
4. Если увеличивающихся цепей нет, получено наибольшее паросочетание.

Например, граф на рисунке а) (паросочетание получено «жадным» алгоритмом) после введения ориентации принимает вид б).



Из свободной вершины 5 достижимы вершины 7, 3, 4, 9 и 10, причем из них вершина 10 свободна, то есть существует увеличивающаяся цепь (5, 9, 4, 7, 3, 10). После ее увеличения получаем наибольшее паросочетание (рисунок *в*).