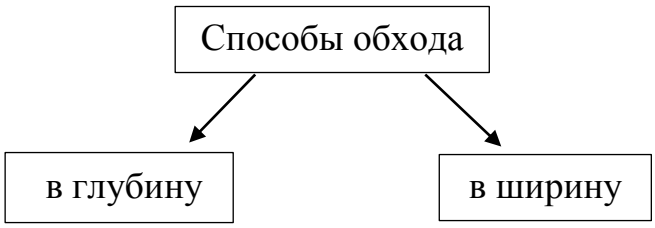
Обход дерева

В программировании часто возникает потребность перебрать один за другим все узлы некоторого дерева и предпринять определенные действия для каждого из них. Такой перебор и называется обходом дерева.

Пример – распечатка узлов дерева, которая уже встречалась нам на прошлых занятиях. Также обход дерева встречается в задачах поиска.



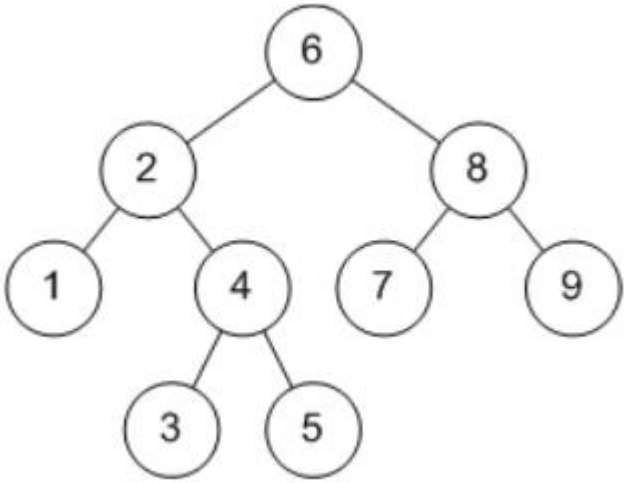
Сначала рассмотрим обход дерева в глубину. Лучше всего этот обход описывается рекурсивно.

Существует различные варианты обхода в глубину. Рассмотрим некоторые из них. Предположим, что у нас есть двоичное дерево.

Вариант 1.

Обойти левое поддерево
Посетить корень
Обойти правое поддерево

Номера вершин дерева при таком способе обхода:



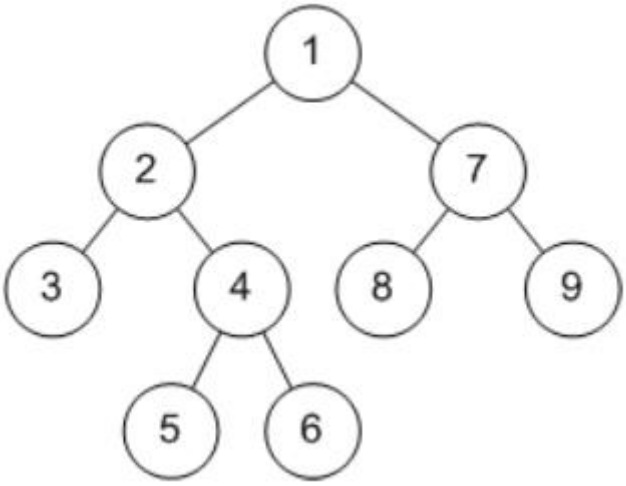
Сначала мы обходим левое поддерево, затем посещаем корень, и, наконец, обходим правое поддерево. При этом обход поддеревьев выполняется рекурсивно с помощью того же самого алгоритма.

Такой алгоритм начинает обработку узлов дерева с самого левого узла, и затем обрабатывает их слева направо, т. е. если есть два узла в дереве, то тот из них, который расположен левее, будет обработан раньше.

Вариант 2.

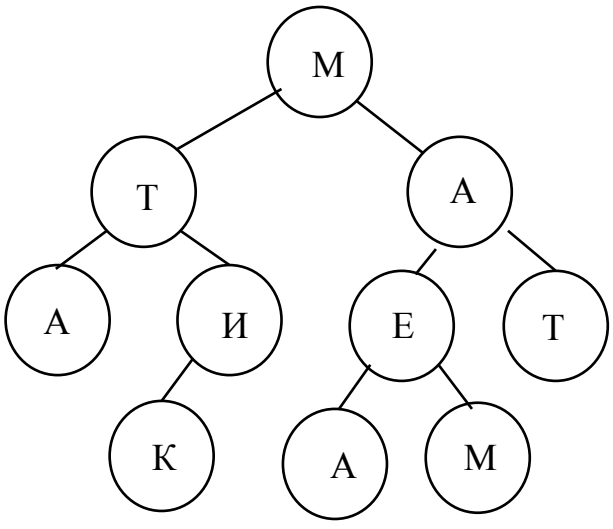
Посетить корень
Обойти левое поддерево
Обойти правое поддерево

Номера вершин дерева при таком способе обхода:



Вопрос 1. Сколько всего существует вариантов обхода в глубину (мы рассмотрели два из них)?

Вопрос 2. Пусть указатель q указывает на корень дерева



Что будет напечатано в результате вызова функции f(q)?

```
template<class T>
void f(Node<T> *p)
{
  if (p!=0)
     {
     cout<<p->data<<endl;
     f(p->right);
     f(p->left);
     }
}
```

Какое слово у Вас получилось?

Пример. Добавить в класс “Двоичные деревья” метод count, вычисляющий число узлов дерева.

```
int count ()
{return f_count(root);}
```

Вспомогательная рекурсивная функция `f_count` будет выглядеть следующим образом:

```
template <class T>
int f_count(Node<T> *p)
{    if(p == nullptr) return 0;
     return 1 + f_count(p->left) + f_count(p->right);
}
```

Задание 1.

Добавить в класс “Двоичные деревья” следующие методы:

1. scale, умножающий данные каждого узла на 3.

2. sum, вычисляющий сумму всех данных в узлах дерева.

3. count_neg, возвращающий число узлов с отрицательными данными.

4. height, возвращающий высоту дерева.

5. find, принимающий параметр d типа T, и возвращающую указатель на любой узел, содержащий данные d, если такие узлы в дереве есть, и nullptr, если таких узлов нет.

6. eval, вычисляющий выражение, заданное данным двоичным деревом (числа в листьях рассматриваются как операнды, а в промежуточных узлах – как коды операций 1 +, 2 - , 3 *, 4 /).

7. reflect, меняющий дерево на его зеркальное отражение (правые и левые сыновья каждого узла меняются местами).

8. mult, возвращающий произведение данных в узлах, которых есть оба сына.

9. min, возвращающую минимальное значение данных в узлах дерева.

Теперь перейдем к методам, касающимся удаления узлов дерева.

Пример.

Добавить в класс деструктор, уничтожающий все узлы дерева, т. е. освобождающий занимаемую ими память.

Очевидно, что сначала нужно удалить поддеревья, а затем корень дерева. Вспомогательная рекурсивная функция `f_del` будет выглядеть следующим образом:

```
void f_del (Node<T> * & p)
{
  if (p==nullptr) return;  // дерево пусто
  f_del (p-> left);   // удалить левое поддерево
  f_del (p-> right);  // удалить правое поддерево
  delete p;        // удалить сам узел
  p = nullptr;   // обнулить указатель
}
```

Сам деструктор:

```
template <class T>
~Tree ()
{ f_del(root); }  // удалить дерево
```

Замечание. В заголовке функции `f_del` фигурирует знак &, это означает, что указатель p передается в функцию по ссылке. При вызове `f_del(root)` дерево будет удалено и, что важно, указатель root станет равным нулю (последнее

выполнится только в случае передачи параметра p по ссылке). На этот момент следует обратить внимание при решении последующих задач.

Задание 2.

Добавить в класс “Двоичные деревья” следующие методы:

1. del0, уничтожающий все поддеревья узлов, данные которых равны 0 (вместе с самими этими узлами).

2. delLeaves, уничтожающий все листья (уничтожаются только те узлы, которые были листьями до этой операции; те узлы, которые стали листьями в процессе выполнения этой операции, сохраняются).

3. del1, уничтожающий все узлы с данными 1 вместе с их левыми поддеревьями (при этом их правые поддеревья, если такие есть, становятся поддеревьями вышестоящих узлов с той стороны, с которой был прикреплен удаляемый узел).

4. enlarge, принимающий объект d типа T, и заменяющий в дереве все нулевые указатели указателями на новые узлы дерева с данными d (без детей).

5. sum_alt, возвращающий сумму данных в узлах дерева, в которую правые сыновья своих отцов входят со знаком "+", а левые – со знаком "–". Данные корня в этой сумме не участвуют.

Об обходе в глубину и ширину

Наиболее просто обойти дерево в глубину, используя рекурсию. Однако, существуют способы, позволяющие обходить дерево в глубину и без использования рекурсии. Правда, для этого нам потребуется структура данных, называемая стеком. Эта структура хранит последовательность элементов, доступ к которой возможен только "с одного конца": мы можем добавить элемент в конец или взять его оттуда; также, предоставляется возможность проверить, есть ли в этой последовательности хотя бы один элемент.

В нашем случае стек должен хранить последовательность указателей на узлы дерева.

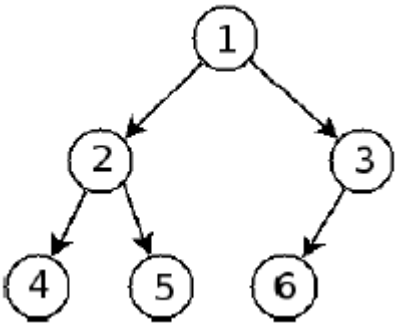
Функция, подсчитывающая число узлов дерева

```
template<class T>
int count(Node<T> *p)
{
   if (p==nullptr) return 0;
   return 1+count(p->left)+count(p->right);
}
```

в нерекурсивном варианте будет выглядеть так:

```
template<class T>
int count1 (Node<T> *p)
{
   int c = 0;
   Stack<Node<T>*> S;
   Node<T> *t;
   if(p==nullptr) return 0;
   S.push(p);
   while(!S.empty())
   {
      c++;
      t = S.pop();
      if(t->left!=nullptr)  S.push(t->left);
      if(t->right!=nullptr) S.push(t->right);
   }
   return c;
}
```

Как уже говорилось выше, существует также способ обхода дерева в ширину. В этом случае узлы дерева просматриваются по уровням от корня к листьям, причем на каждом уровне узлы перебираются слева направо. Номера вершин дерева при обходе в ширину даны на рисунке:



Для обхода дерева в ширину рекурсия совершенно непригодна.

Нерекурсивный же метод очень похож на обход в глубину, только стек заменяется на очередь – другую структуру данных, которая, как и стек, хранит последовательность элементов, но вставляются они с одного конца, а берутся с другого.

Чтобы использовать обход в ширину вместо обхода в глубину, достаточно в функции count1 заменить Stack на Queue. Обычно обход в ширину используется тогда, когда нужно найти узел дерева с заданными свойствами, ближайший к корню.