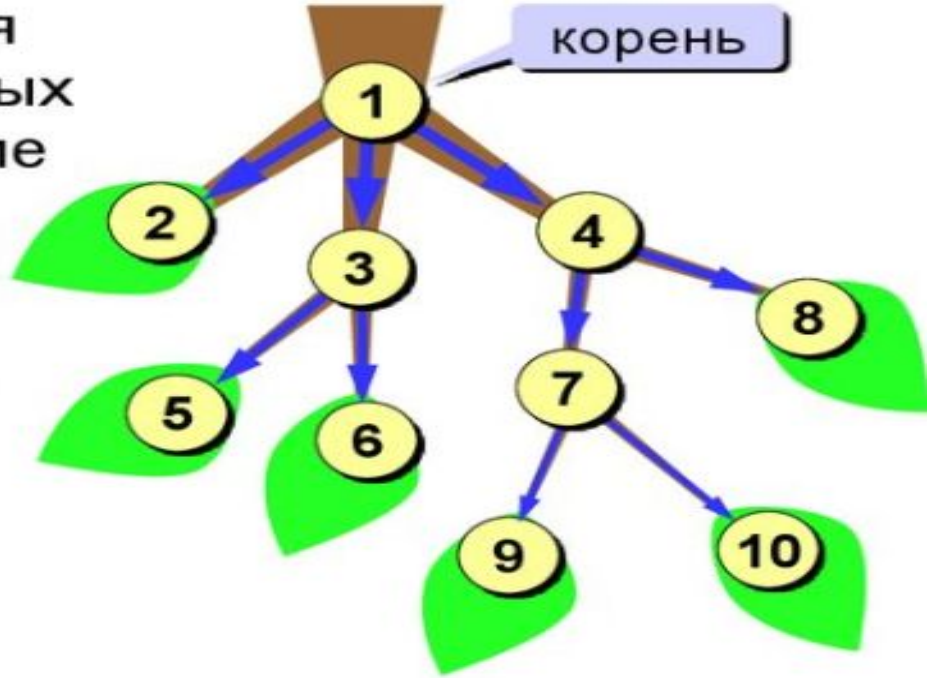
ДЕРЕВЬЯ

Деревья

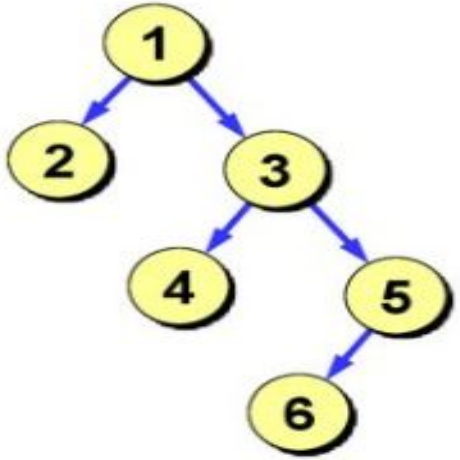
Дерево – это структура данных, состоящая из узлов и соединяющих их направленных ребер (дуг), причем в каждый узел (кроме корневого) ведет ровно одна дуга.

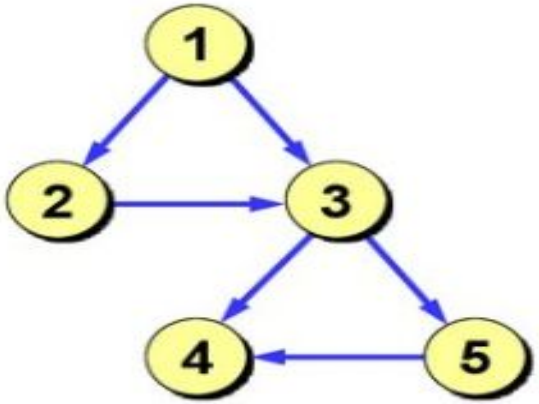
Корень – это начальный узел дерева.

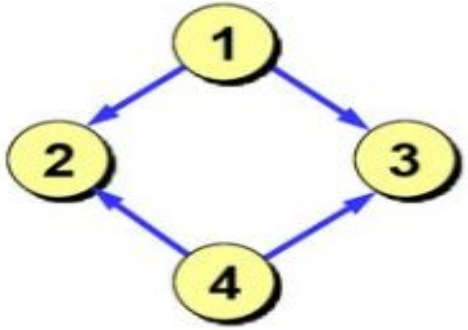
Лист – это узел, из которого не выходит ни одной дуги.



Какие структуры – не деревья?





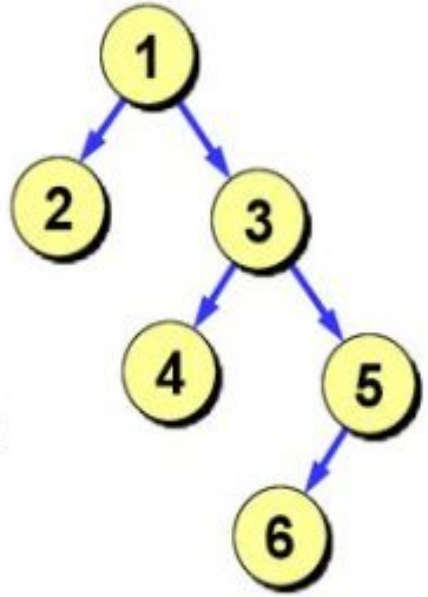


С помощью деревьев отображают отношения подчиненности

Предок узла x – это узел, из которого существует путь по стрелкам в узел x.

Потомок узла x – это узел, в который существует путь по стрелкам из узла x.

Родитель узла x – это узел, из которого существует дуга непосредственно в узел x.



Сын узла x – это узел, в который существует дуга непосредственно из узла x.

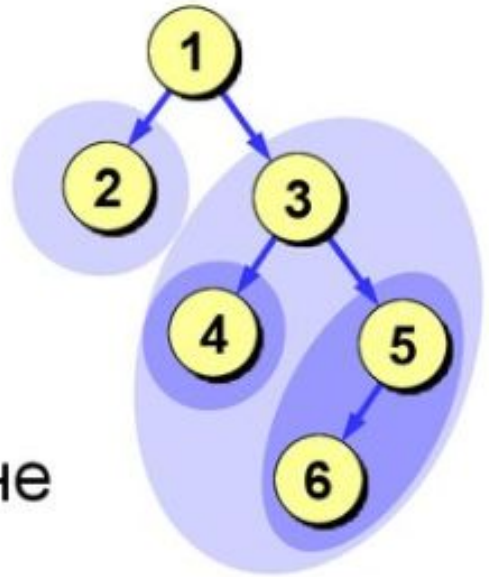
Брат узла x *(sibling)* – это узел, у которого тот же родитель, что и у узла x.

Высота дерева – это наибольшее расстояние от корня до листа (количество дуг).

Дерево – рекурсивная структура данных

Рекурсивное определение:

1. Пустая структура – это дерево.
2. Дерево – это корень и несколько связанных с ним деревьев.



Двоичное (бинарное) дерево – это дерево, в котором каждый узел имеет не более двух сыновей.

1. Пустая структура – это двоичное дерево.
2. Двоичное дерево – это корень и два связанных с ним двоичных дерева (левое и правое поддеревья).

Уровни дерева

Корень дерева имеет *нулевой уровень*.

Уровень потомков увеличивается на 1.

Глубина (высота) дерева равна максимальному уровню потомка плюс 1.

Если элемент не имеет потомков, он
называется *листом* или *терминальным узлом* дерева.

Число потомков внутреннего узла называется его *степенью*.
Максимальная степень всех узлов есть *степень дерева*.

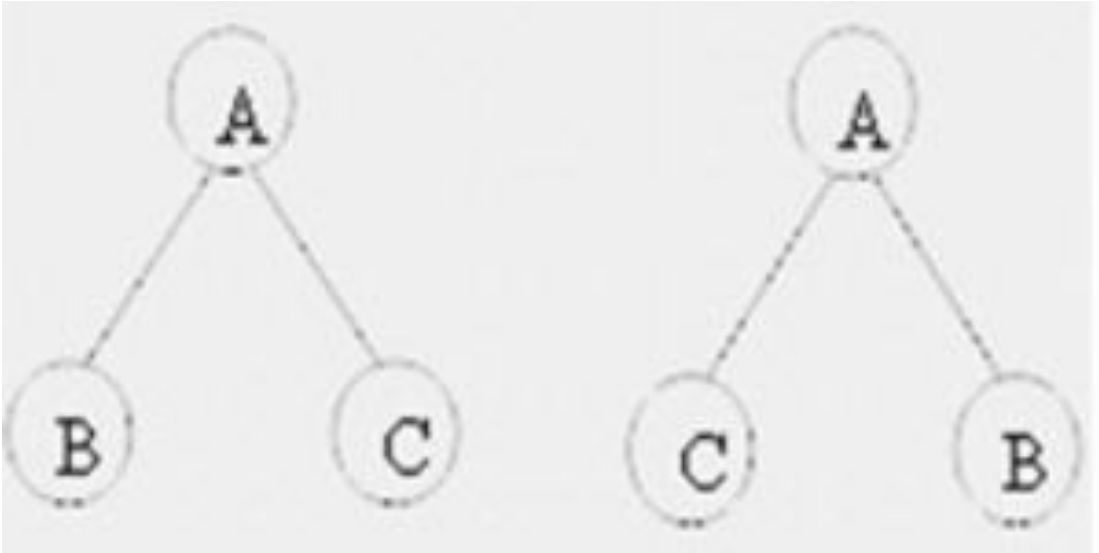
Число ветвей, которое нужно пройти от корня к узлу x,
называется *длиной пути* к x.

Корень имеет длину пути равную 0; узел на уровне i имеет длину пути равную i.

Порядок деревьев

Если в определении дерева имеет значение порядок поддеревьев дерево1, дерево2,.., деревоN, то дерево является упорядоченным.

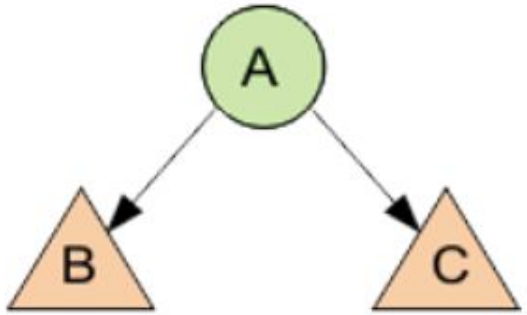
Потомки узла упорядочиваются слева направо.



Обходы дерева

Три наиболее часто используемых обхода деревьев:

Пусть имеем дерево, где A — корень, B и C — левое и правое поддеревья.



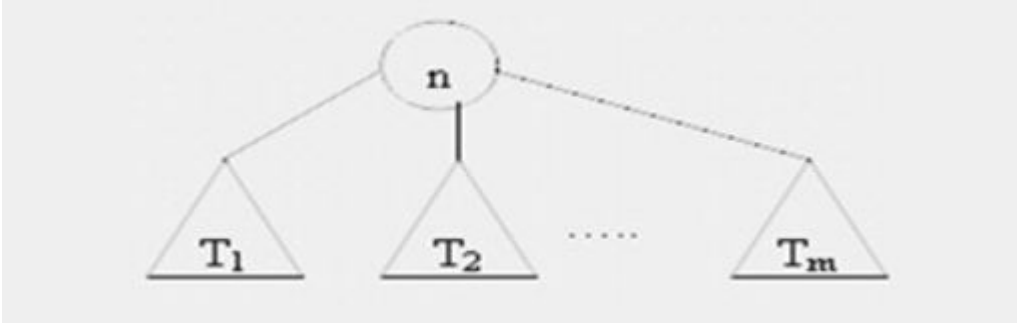
Существует три способа обхода дерева:

- Обход дерева сверху вниз (в прямом порядке): A, B, C — префиксная форма.
- Обход дерева в симметричном порядке (слева направо): B, A, C — инфиксная форма.
- Обход дерева в обратном порядке (снизу вверх): B, C, A — постфиксная форма.

Общей принцип обхода: если дерево является нулевым деревом, то список обхода записывается пустая строка, если дерево состоит из одного узла, то список обхода записывается этот узел.

Общий принцип обхода деревьев

Дерево T имеет корень n и m поддеревьев:T1, T2,..Tm.



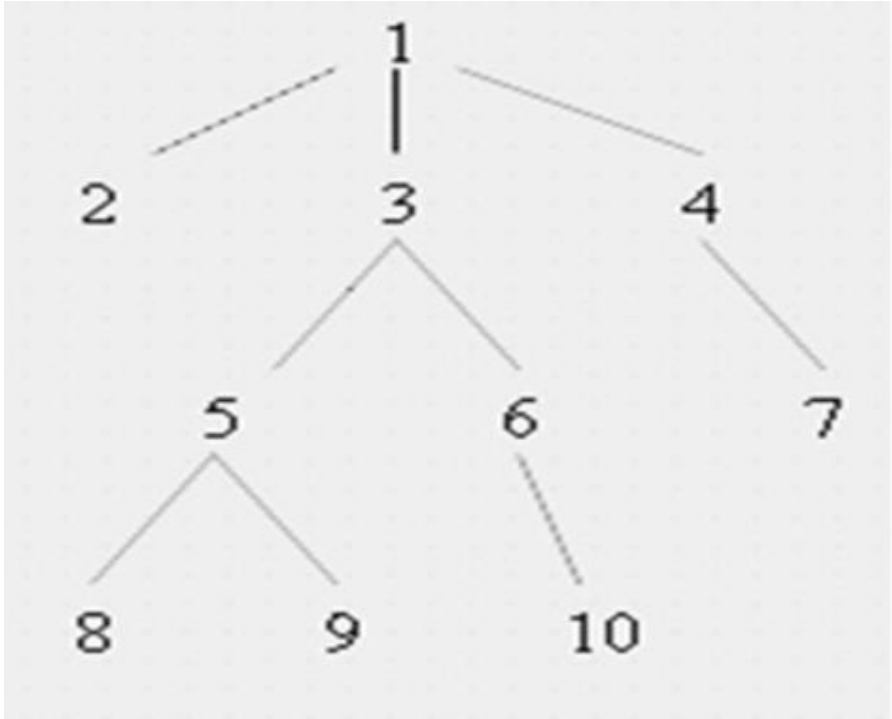
Способы обхода

- Прямой. Сначала посещается корень n, затем в прямом порядке узлы поддерева T1, затем узлы поддерева T2 и т.д. и последним посещают в прямом порядке узлы поддерева Tm.
- Обратный обход. Сначала посещаются в обратном порядке все узлы поддерева T1, затем в обратном порядке узлы поддеревьев T2…Tm и последним рассматривают корень n.
- Симметричный обход. Сначала в симметричном порядке посещают все узлы поддерева T1? Затем корень n, после в симметричном порядке все узлы поддеревьев T2…Tm.

Обход дерева с множеством ветвей

Результаты обходов:

- Прямой 1 2 3 5 8 9 6 10 4 7
- Обратный 2 8 9 5 10 6 3 7 4 1
- Симметричный 2 1 8 5 9 3 10 6 7 4



Помеченные деревья и деревья выражений

Дерево, у которого сопоставлены метки, называют помеченным деревом.

Метка узла – это значение, которое «хранится» в узле.

Дерево- это список;

Узел – это позиция;

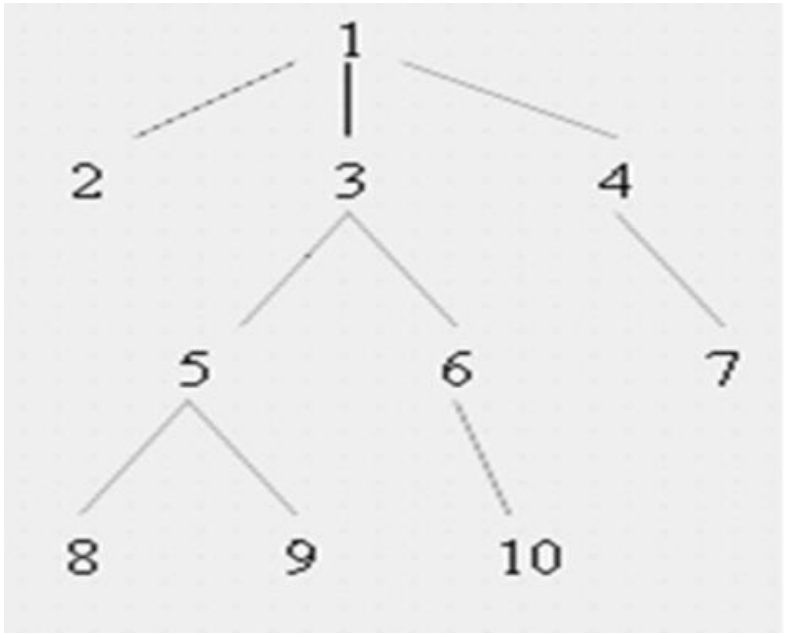
Метка – это элемент.

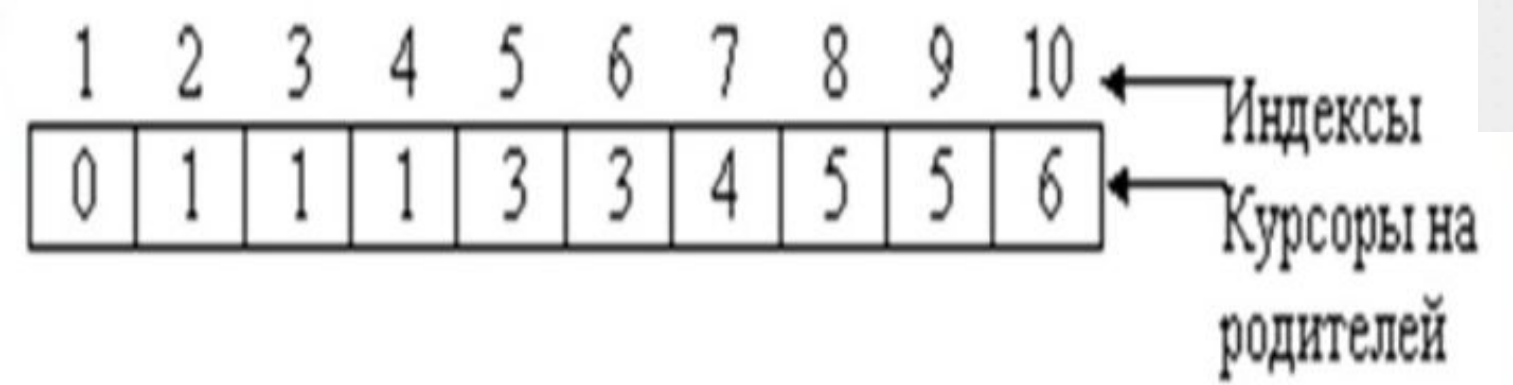
Реализация дерева при помощи массива

Массив где каждый элемент A[i] содержит, номер родительского узла j.

A[i]=j

Корень дерева A[i]=0

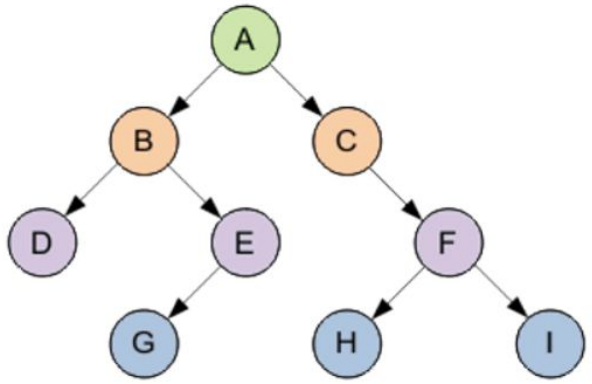




Двоичные(бинарные) деревья

Бинарное дерево — это конечное множество элементов, которое либо пусто, либо содержит элемент (***корень***), связанный с двумя различными бинарными деревьями, называемыми ***левым и правым поддеревьями***. Каждый элемент бинарного дерева называется ***узлом***. Связи между узлами дерева называются его ***ветвями***.

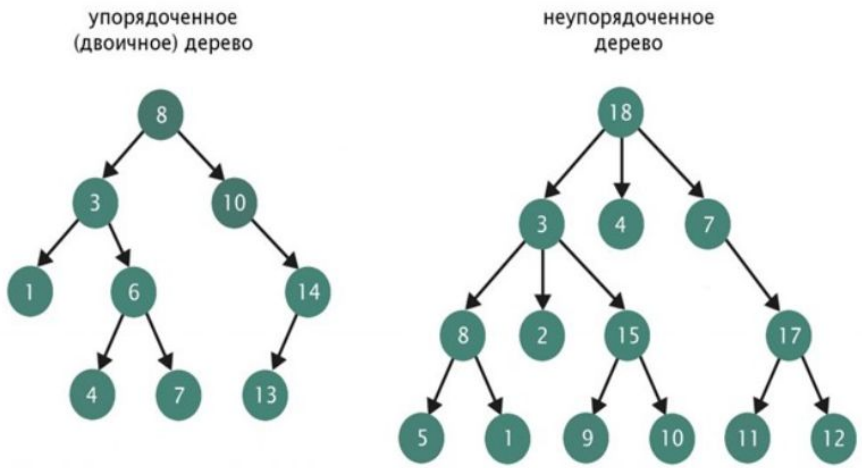
Бинарное дерево



- A — корень дерева
- B — корень левого поддерева
- C — корень правого поддерева

Корень дерева расположен на уровне с минимальным значением.

Виды деревьев



Представление двоичных деревьев

Двоичное дерево можно представить динамической структурой, где каждый узел представляет собой ссылку на правое и левое поддерево.

```
1   struct tnode {
2     int field;           // поле данных
3     struct tnode *left;  // левый потомок
4     struct tnode *right; // правый потомок
5   };
```

Действия с двоичными деревьями

- Обход дерева
- Поиск по дереву
- Включение узла в дерево
- Удаление узла дерева.

Обход двоичных деревьев в прямом, симметричном и обратном порядке

При этом обход дерева в префиксной форме будет иметь вид

```
1  void treeprint(tnode *tree) {
2    if (tree!=NULL) { //Пока не встретится пустой узел
3      cout << tree->field; //Отображаем корень дерева
4      treeprint(tree->left); //Рекурсивная функция для левого поддерева
5      treeprint(tree->right); //Рекурсивная функция для правого поддерева
6    }
7  }
```

Обход дерева в инфиксной форме будет иметь вид

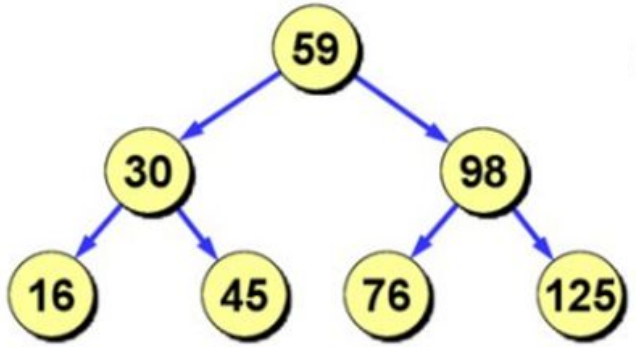
```
1  void treeprint(tnode *tree) {
2    if (tree!=NULL) { //Пока не встретится пустой узел
3      treeprint(tree->left); //Рекурсивная функция для левого поддерева
4      cout << tree->field; //Отображаем корень дерева
5      treeprint(tree->right); //Рекурсивная функция для правого поддерева
6    }
7  }
```

Обход дерева в постфиксной форме будет иметь вид

```
1  void treeprint(tnode *tree) {
2    if (tree!=NULL) { //Пока не встретится пустой узел
3      treeprint(tree->left); //Рекурсивная функция для левого поддерева
4      treeprint(tree->right); //Рекурсивная функция для правого поддерева
5      cout << tree->field; //Отображаем корень дерева
6    }
7  }
```

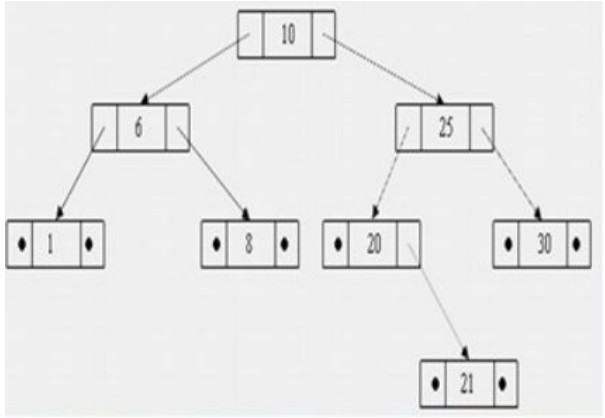
Организация дерева динамической структурой

Все значения узлов(ключи) левого поддерева меньше ключа этого узла, а все ключи правого поддерева больше. Такое дерево является упорядоченным двоичным деревом или деревом поиска.



Алгоритм поиска по упорядоченному

- Если дерево не пусто, то сравниваем искомый ключ с ключом в корне дерева:
  -если ключ совпадает, то поиск завершен;
  -если ключ в корне больше искомого, выполнить поиск в левом поддереве;
  -если ключ в корне меньше искомого, выполнить поиск в правом поддереве.
- Если дерево пусто, то искомый элемент не найден.



Ключ - это характеристика узла по которой выполняется поиск(чаще всего это одно из полей структуры).

Реализация поиска при помощи двоичных деревьев

Бинарное (двоичное) дерево поиска – это бинарное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

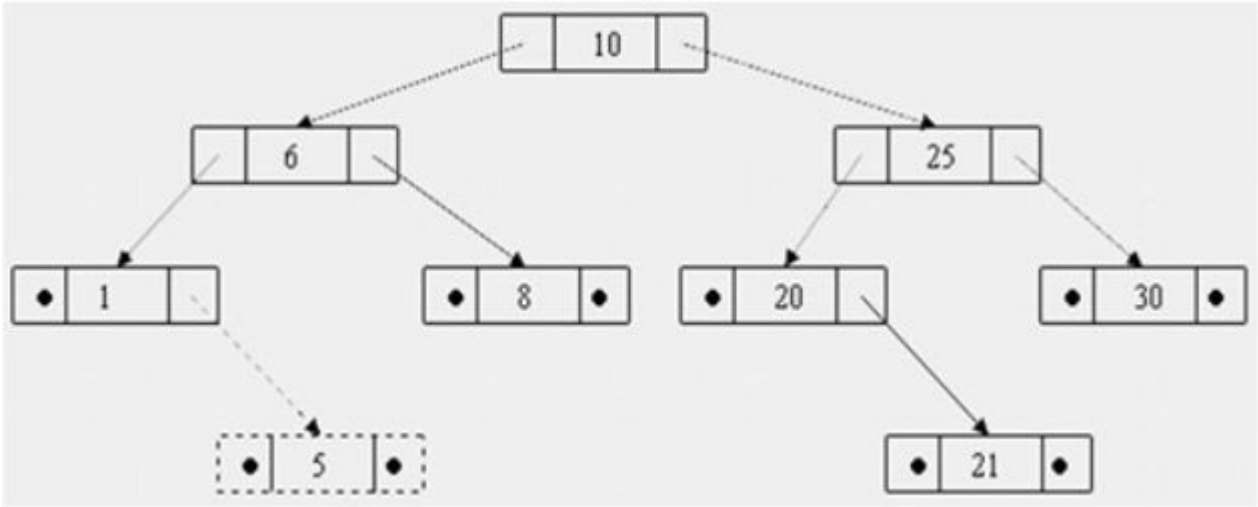
- оба поддерева – левое и правое, являются двоичными деревьями поиска;
- у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, чем значение ключа данных самого узла X;
- у всех узлов правого поддерева произвольного узла X значения ключей данных не меньше, чем значение ключа данных узла X.

Данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше.

Информация, представляющая каждый узел, является структурой, а не единственным полем данных.

Вставка узла (5) в упорядоченное дерево

1. Сравниваем 5 с ключом корня (5<10).
2. Уходим в левое поддерево.
3. Сравниваем 5и 6 (5<6).
4. Уходим влево.
5. Узел конечный (лист).
6. Сравниваем 5 и1 (5>1).
7. Вставляем правого сына.
8. Получили дерево с новым узлом, которое сохранило все свойства дерева поиска.



Добавление узла в двоичное дерево

Добавление элемента – это добавление элемента в конкретное место, которое определяется по алгоритму:

- Сравниваем ключ с корнем, если он меньше ключа корня, то уходим в левое поддерево, если нет, то в правое.
- Действие выполняем до конечного узла (листа).
- Если ключ меньше ключа листа, то добавляем листу левого сына, если нет, то правого сына.

```
1   struct tnode * addnode(int x, tnode *tree) {
2     if (tree == NULL) { // Если дерева нет, то формируем корень
3       tree =new tnode; // память под узел
4       tree->field = x;   // поле данных
5       tree->left =  NULL;
6       tree->right = NULL; // ветви инициализируем пустотой
7     }else  if (x < tree->field)   // условие добавление левого потомка
8       tree->left = addnode(x,tree->left);
9     else    // условие добавление правого потомка
10      tree->right = addnode(x,tree->right);
11    return(tree);
12  }
```

Удаление поддерева

```
1  void freemem(tnode *tree) {
2    if(tree!=NULL) {
3      freemem(tree->left);
4      freemem(tree->right);
5      delete tree;
6    }
7  }
```

Работа с конкретными элементами дерева

Описание структуры

```
typedef struct tnode
{
    int field;// поле данных
    struct tnode *left;//указатель на левое поддерево
    struct tnode *right;//указатель на правое поддерево
    struct tnode *parent;//указатель на предка
} tnode;
```

Инициализация дерева

```
tnode *create(tnode *tree, int key)
{
// выделение памяти под корень
    tnode *tmp = malloc(sizeof(tnode));
// присваивание значения ключу
    tmp -> fild = key;
// присваивание указателю на родителя значения NULL
    tmp -> parent = NULL;
// присваивание указателю на левое и правое поддерево значения NULL
    tmp -> left = tmp -> right = NULL;
    tree = tmp;
    return tree;
}
```

Поиск нужного элемента в дереве

При поиске помним, что слева расположены элементы с меньшим значением ключа, справа — с большим.

```
tnode *search(tnode * tree, int key)
{
// Если дерево пусто или ключ корня равен искомому ключу,
// то возвращается указатель на корень
    if ((tree == NULL) || (tree -> field == key))
        return tree;
// Поиск нужного узла
    if (key < tree -> field)
        return search(tree -> left, key);
    else return search(tree -> right, key);
}
```

Поиск элемента с минимальным или максимальным ключом

```
// Минимальный элемент дерева
node *min(node *tree)
{
    node *l = tree;
    while (l -> left != NULL)
        l = l -> left;
    return l;
}
```

```
// Максимальный элемент дерева
node *max(node *tree)
{
    node *r = tree;
    while (r -> right != NULL)
        r = r -> right;
    return r;
}
```

Поиск нужного элемента в дереве

Поиск элемента за другим (для удаления элементов).

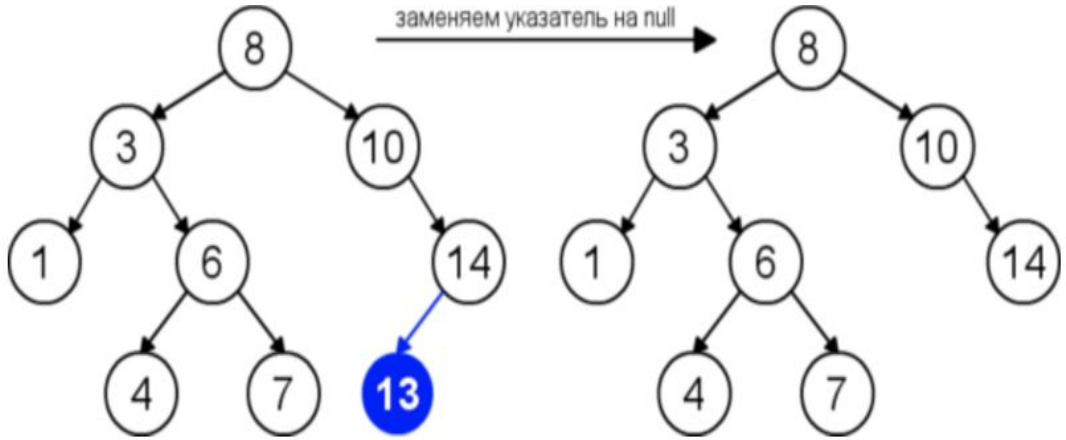
```
tnode *succ(tnode *tree)
{
    tnode *p = tree, *l = NULL;
// если есть правое поддерево,то ищем минимальный элемент в этом поддереве
    if (p -> right != NULL)
        return min(p -> right);
/* правое дерево пусто, идем по родителям до тех пор,
 пока не найдем родителя, для которого наше дерево левое */
    l = p -> parent;
    while ((l != NULL) && (p == l -> right))
    {
        p = l;
        l = l -> parent;
    }
    return l;
}
```

Удаление узла дерева

Удаление узла в дереве связано с его положением и наличием потомков.

Случай 1 : удаляемый узел не имеет левого и правого поддерева, просто удаляем сам узел.

```
tnode *delete(tnode *tree, int key)
{
// Поиск удаляемого узла по ключу
    tnode *p = tree, *l = NULL, *m = NULL;
    l = search(tree, key);
// 1 случай
    if ((l -> left == NULL) && (l -> right == NULL))
    {
        m = l -> parent;
        if (l == m -> right) m -> right = NULL;
        else m -> left = NULL;
        free(l);
    }

    return tree;
}
```

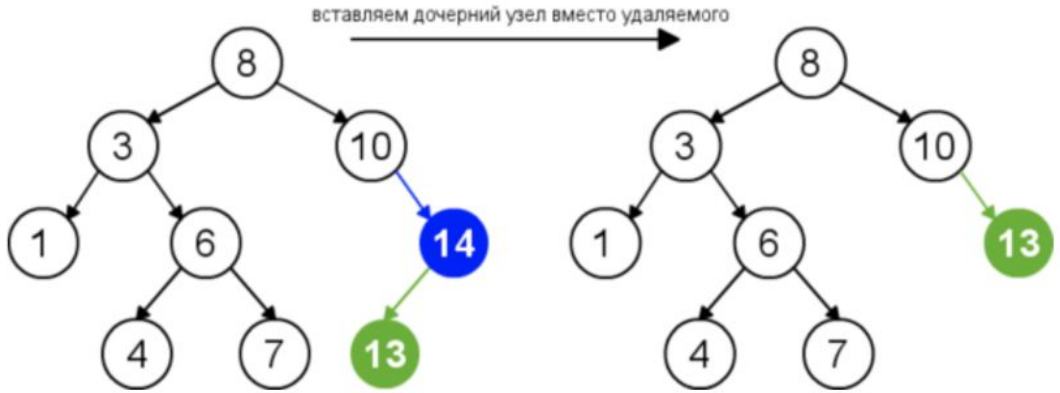


Удаление узла дерева

Удаление узла в дереве связано с его положением и наличием потомков.

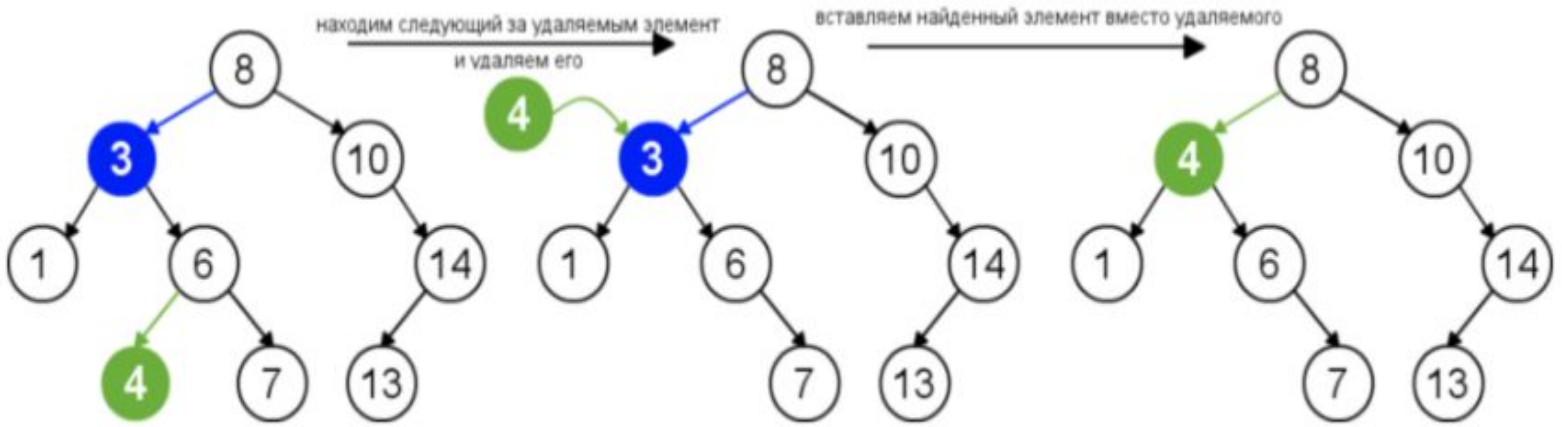
Случай 2 : у удаляемого узла одно поддерево.
Тогда мы просто удаляем данный узел, а на его место ставим поддерево.

```
tnode *delete(tnode *tree, int key)
{
// Поиск удаляемого узла по ключу
    tnode *p = tree, *l = NULL, *m = NULL;
    l = search(tree, key);

// 2 случай, 1 вариант - поддерево справа
    if ((l -> left == NULL) && (l -> right != NULL))
    {
        m = l -> parent;
        if (l == m -> right) m -> right = l -> right;
        else m -> left = l -> right;
        free(l);
    }
// 2 случай, 2 вариант - поддерево слева
    if ((l -> left != NULL) && (l -> right == NULL))
    {
        m = l -> parent;
        if (l == m -> right) m -> right = l -> left;
        else m -> left = l -> left;
        free(l);
    }

    return tree;
}
```



Удаление узла дерева

Случай 3 : у удаляемого узла существуют оба поддерева. Тогда необходимо сначала найти следующий за удаляемым элемент, а потом его поставить на место удаляемого элемента.



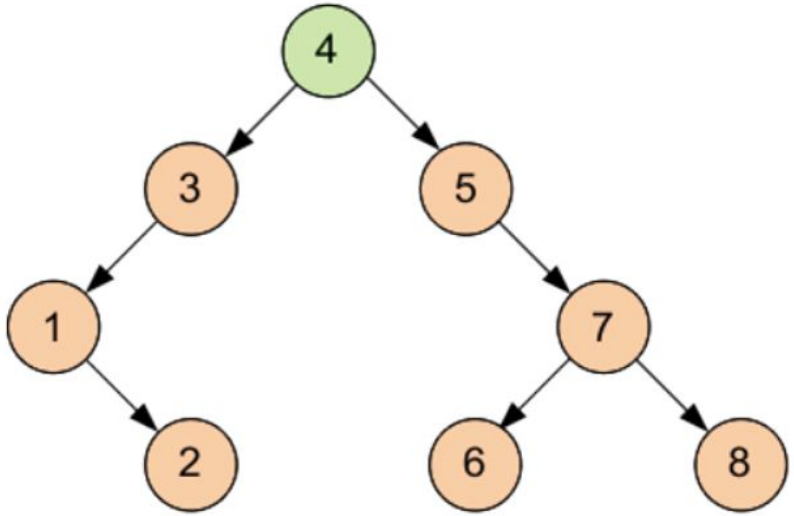

```
tnode *delete(tnode *tree, int key)
{
// Поиск удаляемого узла по ключу
    tnode *p = tree, *l = NULL, *m = NULL;
    l = search(tree, key);

// 3 случай
    if ((l -> left != NULL) && (l -> right != NULL))
    {
        m = succ(l);
        l -> key = m -> key;
        if (m -> right == NULL)
            m -> parent -> left = NULL;
        else m -> parent -> left = m -> right;
        free(m);
    }
    return tree;
}
```

Пример сортировки связанных элементов на основе бинарного дерева поиска

Исходная последовательность имеет вид: **4, 3, 5, 1, 7, 8, 6, 2**

Для сортировки с помощью дерева исходная сортируемая последовательность представляется в виде структуры данных «дерево».

Корень дерева – это начальный элемент последовательности. Остальные элементы, меньшие корневого, располагаются в левом поддереве, все элементы, большие корневого, располагаются в правом поддереве. Причем это правило должно соблюдаться на каждом уровне.



После того, как все элементы размещены в структуре «дерево», необходимо вывести их, используя инфиксную форму обхода.

Пример сортировки связанных элементов на основе бинарного дерева поиска

[*] Сортировка масива на основе дерева поиска.cpp

```cpp
#include <iostream>
using namespace std;
// Структура - узел дерева
struct tnode
{
  int field;           // поле данных
  struct tnode *left;  // левый потомок
  struct tnode *right; // правый потомок
};
// Вывод узлов дерева (обход в инфиксной форме)
void treeprint(tnode *tree)
{
  if (tree != NULL) {      //Пока не встретится пустой узел
    treeprint(tree->left);  //Рекурсивная функция вывода левого поддерева
    cout << tree->field << " "; //Отображаем корень дерева
    treeprint(tree->right); //Рекурсивная функция вывода правого поддерева
  }
}
// Добавление узлов в дерево
struct tnode * addnode(int x, tnode *tree) {
  if (tree == NULL)      // Если дерева нет, то формируем корень
  {
    tree = new tnode; //память под узел
    tree->field = x;   //поле данных
    tree->left = NULL;
    tree->right = NULL; //ветви инициализируем пустотой
  }
  else     // иначе
    if (x < tree->field)   //Если элемент x меньше корневого, уходим влево
      tree->left = addnode(x, tree->left); //Рекурсивно добавляем элемент
    else  //иначе уходим вправо
      tree->right = addnode(x, tree->right); //Рекурсивно добавляем элемент
    return(tree);
}
//Освобождение памяти дерева
void freemem(tnode *tree)
{
  if (tree != NULL)     // если дерево не пустое
  {
    freemem(tree->left);   // рекурсивно удаляем левую ветку
    freemem(tree->right);  // рекурсивно удаляем правую ветку
    delete tree;           // удаляем корень
  }
}
// Основная программа
int main()
{
  struct tnode *root = 0;     // Объявляем структуру дерева
  system("chcp 1251");     // переходим на русский язык в консоли
  system("cls");
  int a;              // текущее значение узла
  // В цикле вводим 8 узлов дерева
  for (int i = 0; i< 8; i++)
  {
    cout << "Введите узел " << i + 1 << ": ";
    cin >> a;
    root = addnode(a, root); // размещаем введенный узел на дереве
  }
  treeprint(root);     // выводим элементы дерева, получаем отсортированный массив
  freemem(root);       // удаляем выделенную память
  cin.get();  cin.get();
  return 0;
}
```

```
Введите узел 1: 4
Введите узел 2: 3
Введите узел 3: 5
Введите узел 4: 1
Введите узел 5: 7
Введите узел 6: 8
Введите узел 7: 6
Введите узел 8: 2
1 2 3 4 5 6 7 8
```

Пример поиска слов в выражении на основе бинарного дерева поиска

Написать программу, подсчитывающую частоту встречаемости слов входного потока(**now is the time for all good men to come to the aid of their party**).

Поскольку список слов заранее не известен, мы не можем предварительно упорядочить его. Неразумно пользоваться линейным поиском каждого полученного слова, чтобы определять, встречалось оно ранее или нет, т.к. в этом случае программа работает слишком медленно.

Один из способов — постоянно поддерживать упорядоченность уже полученных слов, помещая каждое новое слово в такое место, чтобы не нарушалась имеющаяся упорядоченность. Воспользуемся бинарным деревом.

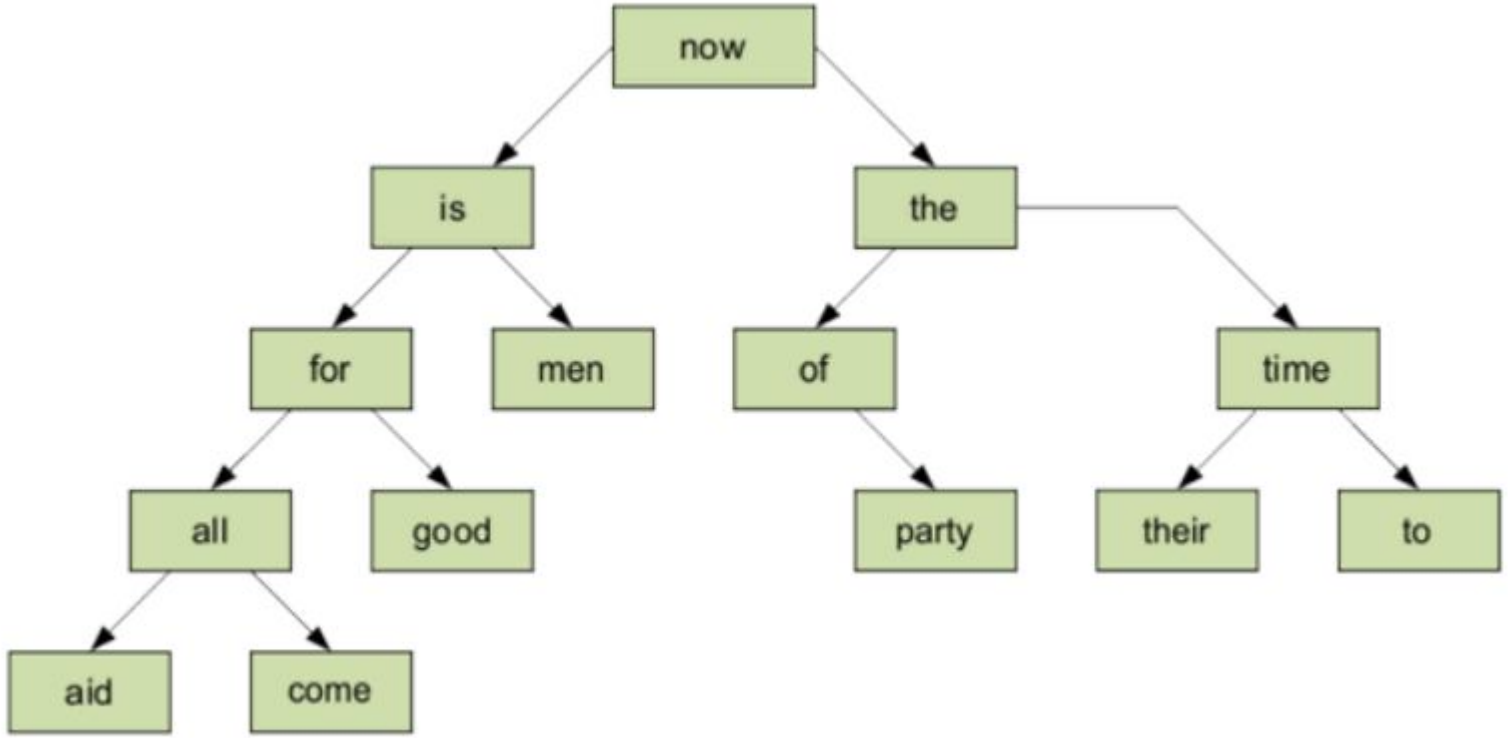
В дереве каждый узел содержит:

- указатель на текст слова;
- счетчик числа встречаемости;
- указатель на левого потомка;
- указатель на правого потомка.

Пример поиска слов в выражении на основе бинарного дерева поиска

Фраза для рассмотрения - **now is the time for all good men to come to the aid of their party (сейчас самое время всем хорошим людям прийти на помощь своей партии).**

Дерево имеет вид:



Пример поиска слов в выражении на основе бинарного дерева поиска

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <cstddef>
#define MAXWORD 100
struct tnode {                  // узел дерева
  char* word;                   // указатель на строку (слово)
  int count;                    // число вхождений
  struct tnode* left;           // левый потомок
  struct tnode* right;          // правый потомок
};
// функция добавления узла к дереву
struct tnode* addtree(struct tnode* p, char* w) {
  int cond;
  if (p == NULL) {
    p = (struct tnode*)malloc(sizeof(struct tnode));
    p->word = _strdup(w);
    p->count = 1;
    p->left = p->right = NULL;
  }
  else if ((cond = strcmp(w, p->word)) == 0)
    p->count++;
  else if (cond < 0)
    p->left = addtree(p->left, w);
  else
    p->right = addtree(p->right, w);
  return p;
}
// функция удаления поддерева
void freemem(tnode* tree) {
  if (tree != NULL) {
    freemem(tree->left);
    freemem(tree->right);
    free(tree->word);
    free(tree);
  }
}
```

```c
// функция вывода дерева
void treeprint(struct tnode* p) {
  if (p != NULL) {
    treeprint(p->left);
    printf("%d %s\n", p->count, p->word);
    treeprint(p->right);  }
}
int main() {
  struct tnode* root;
  char word[MAXWORD];
  root = NULL;
  do {
    scanf_s("%s",word, MAXWORD);
    if (isalpha(word[0]))
      root = addtree(root, word);
  } while (word[0] != '0');     // условие выхода - ввод символа '0'
  treeprint(root);
  freemem(root);
  getchar();
  getchar();
  return 0;
}
```

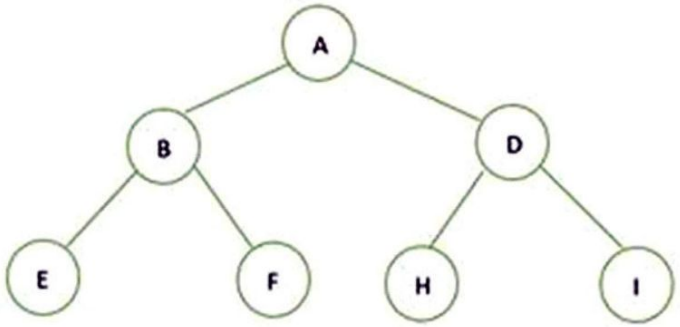
```
now is the time for all good men to come to the aid of their party 0
1 aid
1 all
1 come
1 for
1 good
1 is
1 men
1 now
1 of
1 party
2 the
1 their
1 time
2 to
```

Полное бинарное дерево

Полное бинарное дерево это дерево у каждого узла которого есть либо 2 дочерних элемента, либо 0 дочерних элементов.

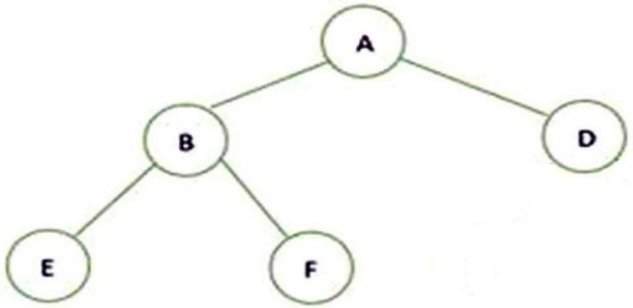
Пример 1

(идеальное двоичное дерево)



Пример 2

(неидеальное двоичное дерево)



Уровни n

Для A n=0, для B,D n=1 , для E,F,H,I n=2

Глубина(высота) дерева

n+1=3

Максимальное количество узлов $2^{n+1}-1$

$2^{2+1}-1=2^3-1=7$

Полное бинарное дерево

Алгоритм:

Для создания полного двоичного дерева нам требуется структура данных очереди для отслеживания вставленных узлов.

- Шаг 1: Инициализируйте корень новым узлом, когда дерево пусто.
- Шаг 2: Если дерево не пусто, получите передний элемент

Если передний элемент не имеет левого дочернего элемента, установите левый дочерний элемент в новый узел
Если правильный дочерний элемент отсутствует, установите правильный дочерний элемент в качестве нового узла.

- Шаг 3: Если у узла есть оба дочерних элемента, извлеките его из очереди.
- Шаг 4: Поставьте в очередь новые данные.

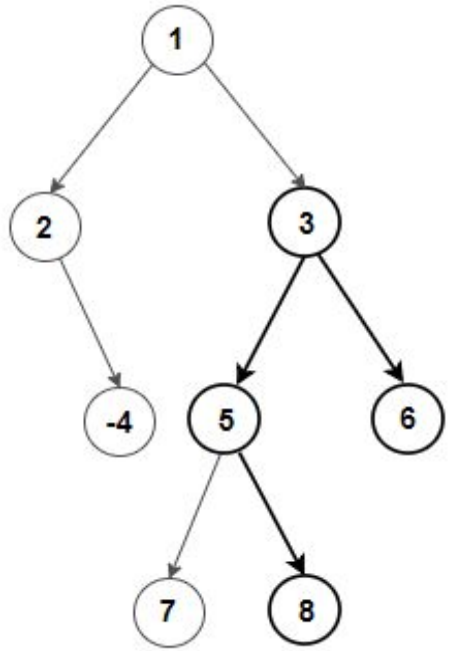
Определение максимального пути между листьями бинарного дерева

Максимальный суммарный путь между двумя листьями, который проходит через узел, имеет значение, равное максимальной сумме пути от узла к листу его левого и правого дочерних элементов плюс значение узла. Максимальное значение среди всех путей с максимальной суммой, найденных для каждого узла в дереве.

Временная сложность этого решения $O(n^2)$, где n - количество узлов в дереве.

Сложность для определения максимальной суммы пути для каждого узла с учетом его левого и правого поддерева занимает $O(n)$ время.

Обход выполняем снизу вверх, определяя максимальную сумму пути от узла к листу для каждого узла до его родителя



Определение максимального пути между листьями бинарного дерева

```cpp
#include <iostream>
#include <climits>
using namespace std;

// Структура узла бинарного дерева
struct Node
{
    int data;
    Node *left, *right;

    Node(int data)
    {
        this->data = data;
        this->left = this->right = nullptr;
    }
};

// Рекурсивная функция для нахождения пути максимальной суммы между двумя листьями
// в бинарном дереве
int findMaxSumPath(Node* root, int &max_sum)
{
    // базовый случай: пустое дерево
    if (root == nullptr) {
        return 0;
    }

    // найти максимальную сумму пути от узла к листу, начиная с левого потомка
    int left = findMaxSumPath(root->left, max_sum);

    // найти максимальную сумму пути от узла к листу, начиная с правого потомка
    int right = findMaxSumPath(root->right, max_sum);

    // важно вернуть максимальную сумму пути от узла к листу, начиная с
    // текущего узла

    // случай 1: левый дочерний элемент равен нулю
    if (root->left == nullptr) {
        return right + root->data;
    }

    // случай 2: правый дочерний элемент равен нулю
    if (root->right == nullptr) {
        return left + root->data;
    }

    // найти максимальный суммарный путь "сквозь" текущий узел
    int cur_sum = left + right + root->data;

    // обновить путь максимальной суммы, найденный до сих пор
    //(обратите внимание, что путь максимальной суммы
    // "исключение" текущего узла в поддереве с корнем в текущем узле
    // уже обновляется, так как мы выполняем обход в обратном порядке)

    max_sum = max(cur_sum, max_sum);

    // случай 3: существуют и левый, и правый дочерние элементы
    return max(left, right) + root->data;
}

// Функция для возврата пути максимальной суммы между
//  двумя листьями в бинарном дереве
int findMaxSumPath(Node* root)
{
    int max_sum = INT_MIN;
    findMaxSumPath(root, max_sum);

    return max_sum;
}
```

Определение максимального пути между листьями бинарного дерева

```
70  int main()
71  {
72      /* Построим следующее дерево
73           1
74          /   \
75         /     \
76        2       3
77         \     / \
78         -4   5   6
79             / \
80            7   8
81      */
82
83      Node* root = new Node(1);
84      root->left = new Node(2);
85      root->right = new Node(3);
86      root->left->right = new Node(-4);
87      root->right->left = new Node(5);
88      root->right->right = new Node(6);
89      root->right->left->left = new Node(7);
90      root->right->left->right = new Node(8);
91
92      cout << findMaxSumPath(root) << endl;
93
94      return 0;
95  }
```

Реализация программного кода

главная.cpp

```
#include <iostream>
#include <climits>
Использование пространства имен std;

Структура данных для хранения узла двоичного дерева
struct Узел
{
    int данные;
    Узел  *влево, *вправо;

    Node(int data)
    {
        this->data = данные;
        this->left  = this->right = nullptr;
    }
};

Рекурсивная функция для нахождения максимального суммарного пути между двумя листьями
в бинарном дереве
int findMaxSumPath(Node* root, int &max_sum)
{
    Базовый вариант: пустое дерево
    if (root == nullptr) {
        возврат 0;
    }

    Найдите путь от узла до листа с максимальной суммой, начиная с левого дочернего элемента
    int  left = findMaxSumPath(root->left, max_sum);
```

Ввод необязательный

Выпуск

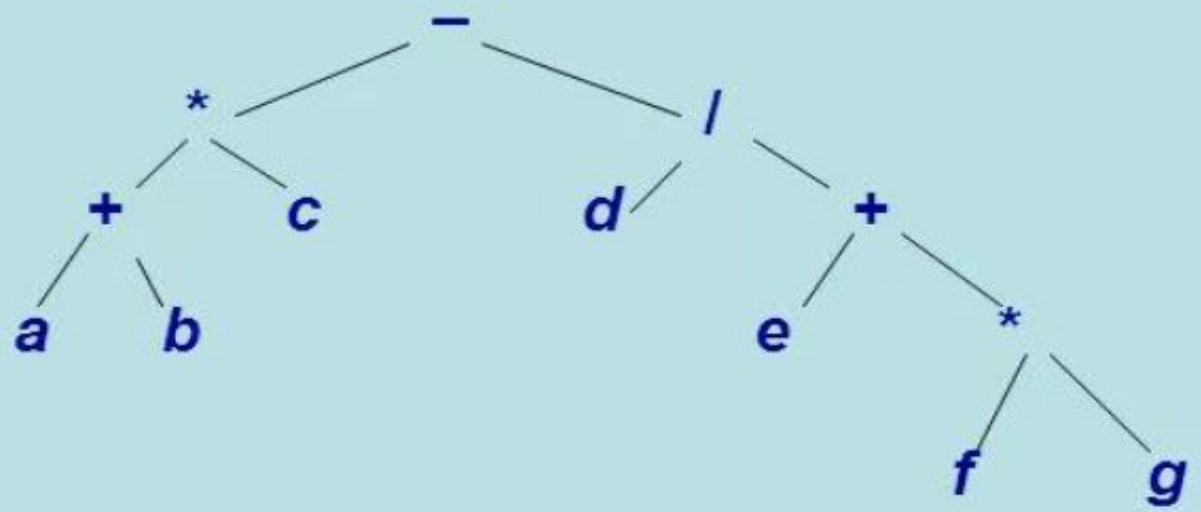
Принятый 0,005 с, 1068 КБ

```
22
```

Обход бинарного дерева, представляющего арифметическое выражение с бинарными операциями

Арифметическое выражение

$$(a + b) * c - d / (e + f * g)$$



1) КЛП – префиксная запись: $- * + a\ b\ c\ /\ d + e * f\ g$;

2) ЛКП – инфиксная запись (без скобок): $a + b * c - d / e + f * g$;

3) ЛПК – постфиксная запись: $a\ b + c * d\ e\ f\ g * + / -$.

Построение бинарного дерева для арифметического выражения

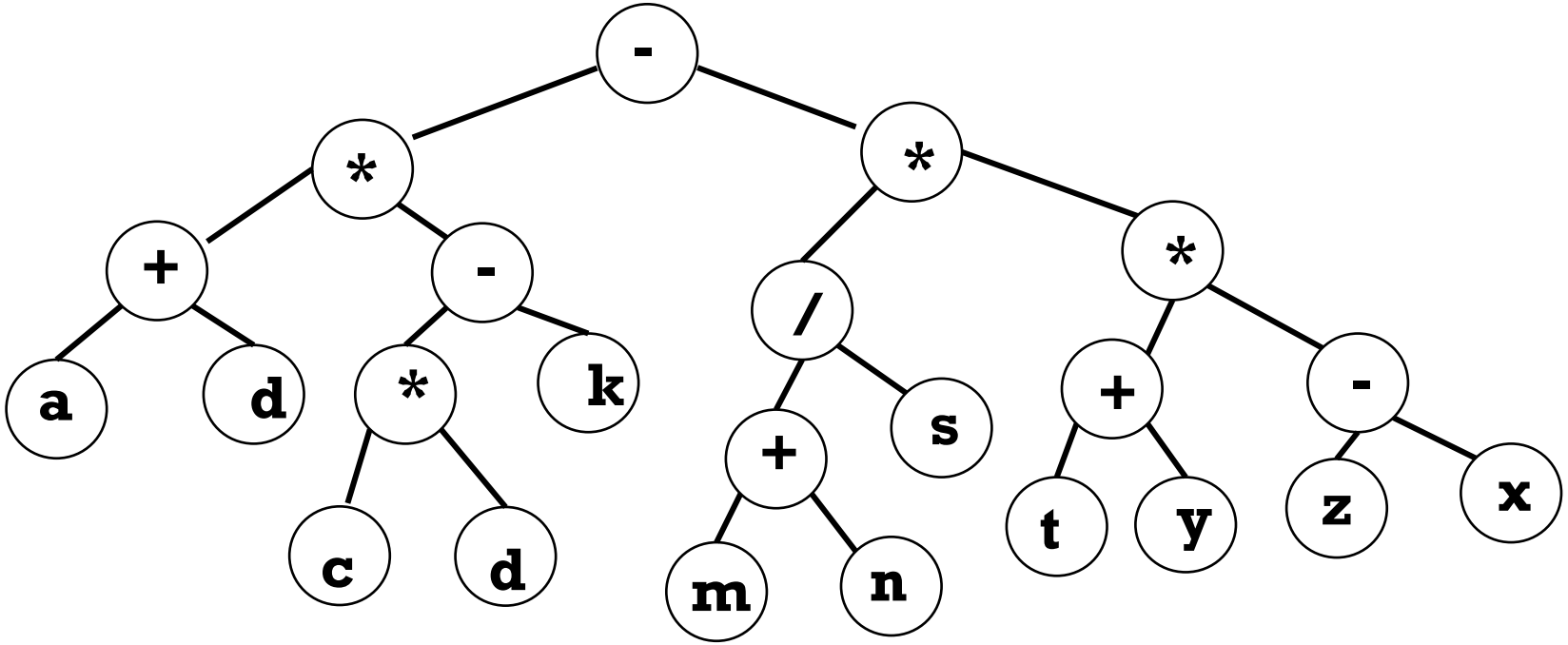
2 1 3 4 11 7 6 5 9 8 10

((a + d) * (c * d - k)) - ((m + n) / s * (t + y) *(z - x))

Представим выражение в префиксной форме

- * + a d *- c d k * + / m n s * + t y – z x

Построим бинарное дерево выражения



Построение бинарного дерева для алгоритма Хаффмана

Кодирование Хаффмана (жадный алгоритм) — это алгоритм сжатия данных, который формирует основную идею сжатия файлов.

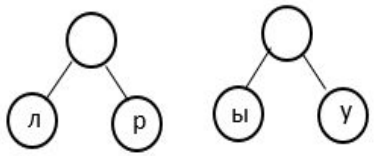
мама мыла раму

Составим таблицу используемых символов и последовательно представим бинарные деревья, сначала для наименьших повторений одновременно корректируем таблицу символов.

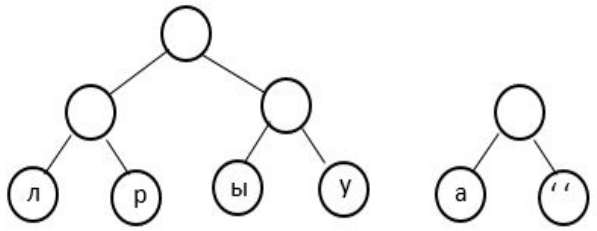
В итоге получаем коды каждой буквы представленной в предложении и пробела.

Кол-во повторений	4	4	2	1	1	1	1
символ	‘м’	‘а’	‘ ‘	‘л’	‘р’	‘ы’	‘у’

Кол-во повторений	4	4	2	2	2
символ	‘м’	‘а’	‘ ‘	‘л’‘р’	‘ы’‘у’



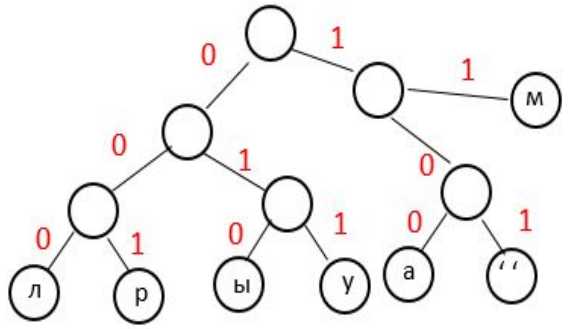
Кол-во повторений	4	4	2	4
символ	‘м’	‘а’	‘ ‘	‘л’‘р’‘ы’‘у’



Кол-во повторений	4	6	4
символ	‘м’	‘а’‘ ‘	‘л’‘р’‘ы’‘у’

Итоговая таблица кодов

коды	11	100	101	000	001	010	011
символ	‘м’	‘а’	‘ ‘	‘л’	‘р’	‘ы’	‘у’



Построение бинарного дерева для алгоритма Хаффмана

- Алгоритм, названный в честь Клода Шеннона и Роберта Фано, работает по следующему правилу:
- Отсортируйте символы по их частоте.
- Разделите символы в этом порядке на две группы, чтобы сумма частот в двух группах была как можно равной. Две группы соответствуют левому и правому поддереву создаваемого дерева Шеннона. Результирующее разделение не обязательно является лучшим, что может быть достигнуто с заданными частотами. Также не ищут лучшего разбиения, поскольку это не обязательно приводит к лучшему результату. Важно, чтобы среднее отклонение от энтропии символов было как можно меньше.
- Если в одной из результирующих групп более одного символа, рекурсивно примените алгоритм к этой группе.
- Важным элементом этого алгоритма является первый шаг. Это гарантирует, что символы с одинаковой вероятностью часто оказываются в одном поддереве. Это важно, потому что узлы в одном дереве получают битовые последовательности с одинаковой длиной кода (максимальная разница может быть только количеством узлов в этом дереве). Если вероятности сильно различаются, невозможно сгенерировать битовые последовательности с необходимыми различиями в длине. Это означает, что редкие символы получают слишком короткие битовые последовательности, а частые символы — слишком длинные битовые последовательности.

Кол-во повторений	4	4	2	1	1	1	1
символ	‘м’	‘а’	‘ ‘	‘л’	‘р’	‘ы’	‘у’

Разделим символы примерно на равновероятные части по мере их повторения

Кол-во повторений	4	2	1
символ	‘м’	‘ ‘	‘л’

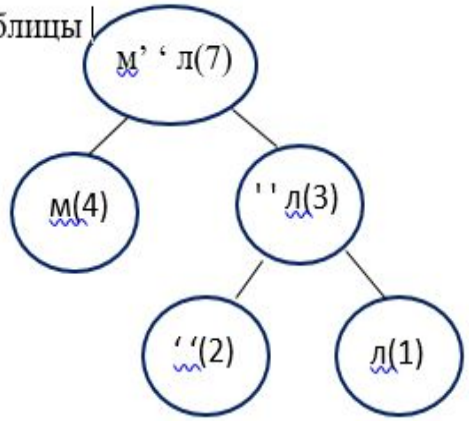
Кол-во повторений	4	1	1	1
символ	‘а’	‘р’	‘ы’	‘у’

Строим дерево

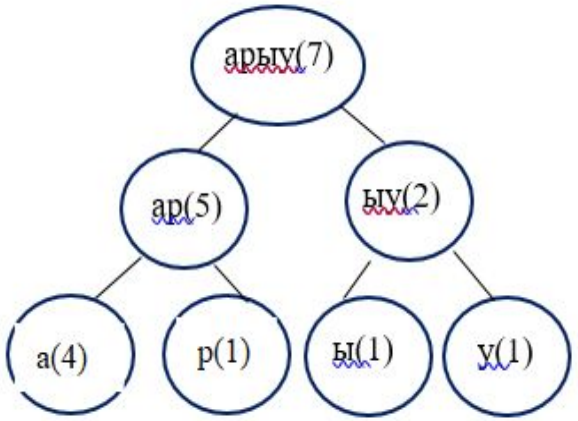
Общее количество повторений всех символов предложения

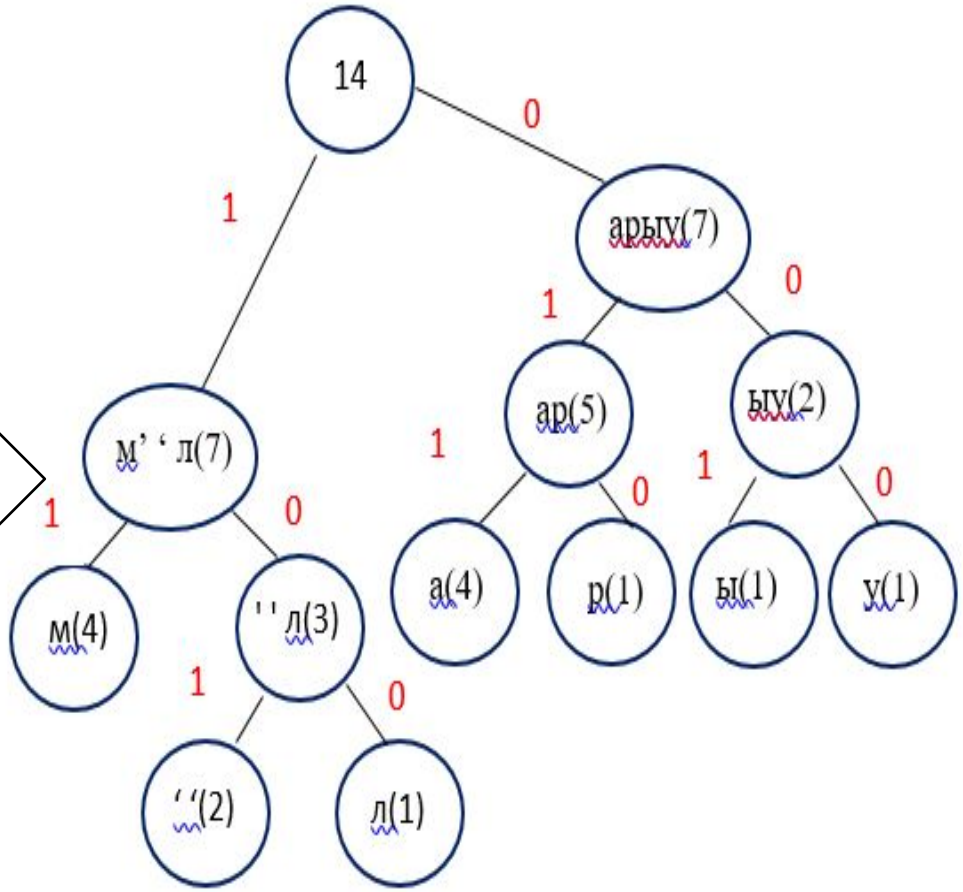
14

Для первой таблицы



Для второй таблицы





Коды	11	011	101	100	010	001	000
символ	‘м’	‘а’	‘ ‘	‘л’	‘р’	‘ы’	‘у’