**Нелинейные структуры данных. Бинарное дерево.**

**Цель. Получение умений и навыков разработки и реализации операций над структурой данных - бинарное дерево.**

Дерево Т называется бинарным деревом, если:
• это пустое дерево
• либо состоит из одного узла – корня дерева
• либо имеет два поддерева – левое и правое, либо имеет одно поддерево: левое или правое, каждое из которых так же двоичное дерево
Т.е. бинарное (двоичное) дерево это дерево, степень которого не больше двух.

Пример 1. Двоичное дерево.

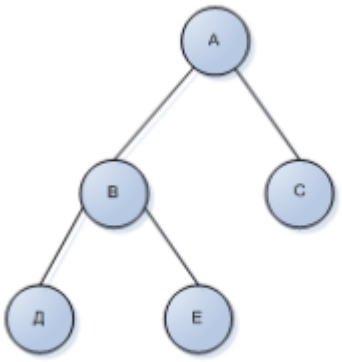



Рис.1. Бинарное дерево

В данном дереве всего поддеревьев 11:

дерево A (является поддеревом самого себя),
дерево B, C, D, E
и 6 пустых (у листьев).

Левое поддерево дерева A - это дерево с корнем B.
Правое поддерево дерева A – это дерево с корнем C.
Путь в дереве от узла A к узлу C: последовательность из элементов – узлов A, B, C.

Количество листьев в дереве можно подсчитать рекурсивно:

$$Leaves(T) = \begin{cases} 0 \text{ если дерево } T \text{ пусто} \\ 1 \text{ если дерево } T \text{ лист} \\ Leaves(LeftTree(T)) + Leaves(RightTree(T)) + 1 \text{ иначе} \end{cases}$$

Неформально высота дерева это количество ветвей между корневым узлом и наиболее удаленным листом.
Ветвь – это линия, соединяющая корень дерева с его поддеревом.
Высоту дерева можно подсчитать рекурсивно:

$$Hight(T) = \begin{cases} -1 \text{ если } T \text{ пусто} \\ 1 + \max(Hight(LeftTree(T)), Hight(RightTree(T))) \text{ иначе} \end{cases}$$

Уровень, на котором находится узел, можно рассчитать рекурсивно:

$$level(n) = \begin{cases} 0 \text{ если n корень} \\ 1 + level(Parent(n)) \text{ иначе} \end{cases}$$

Количество узлов бинарного дерева T можно определить рекурсивно:

$$CountNodes(T) = \begin{cases} 0 \text{ если дерево } T \text{ пусто} \\ 1 + CountNodes(LeftTree(T)) + CountNodes(RightTree(T)) \text{ иначе} \end{cases}$$

Виды двоичных деревьев:

**1) Раздваивающееся бинарное дерево**

Бинарное дерево T называется раздваивающимся деревом, если:
• оно пустое дерево
• либо оба его поддерева являются пустыми
• либо оба поддерева являются непустыми
раздваивающимися деревьями.

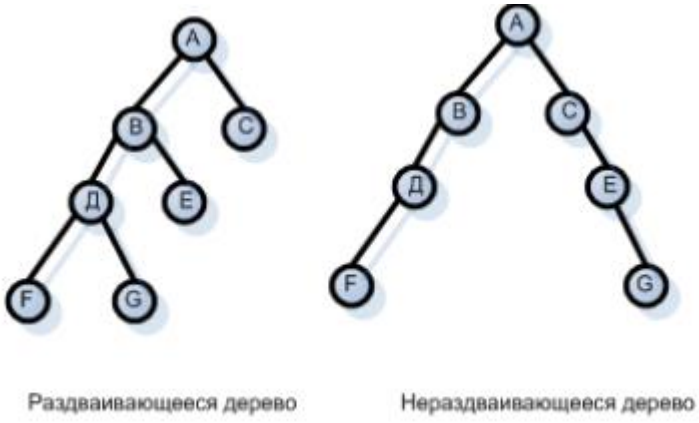



*Рис. 2. Примеры*

**2) Полное бинарное дерево.**

Определение 1. Полное дерево – это раздваивающееся дерево, все листья которого находятся на одном уровне.

Определение 2. Бинарное дерево T называется полным деревом, если:

• оно пустое дерево

• либо оба его поддерева имеют одинаковую высоту, и оба являются полными.

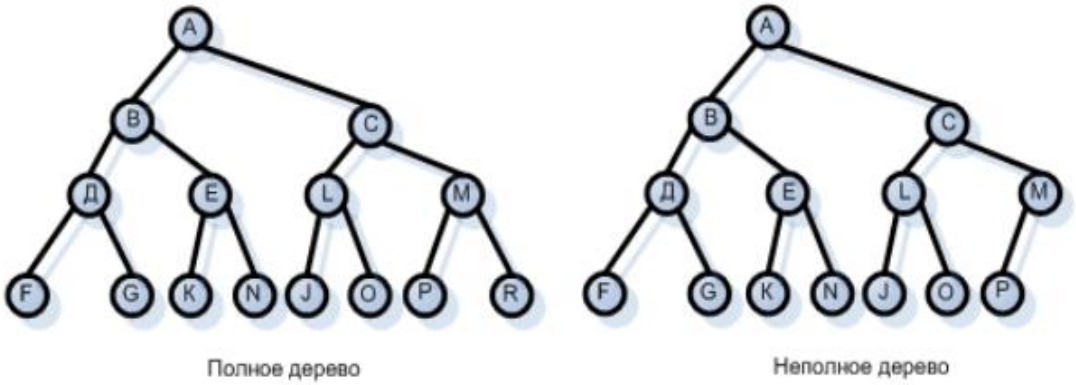



*Рис. 3. Примеры*

**3) Законченное бинарное дерево.**
Это полное дерево до уровня Height(T)-1 (высота-1), а все листья на нижнем уровне располагаются ближе к левому краю дерева.

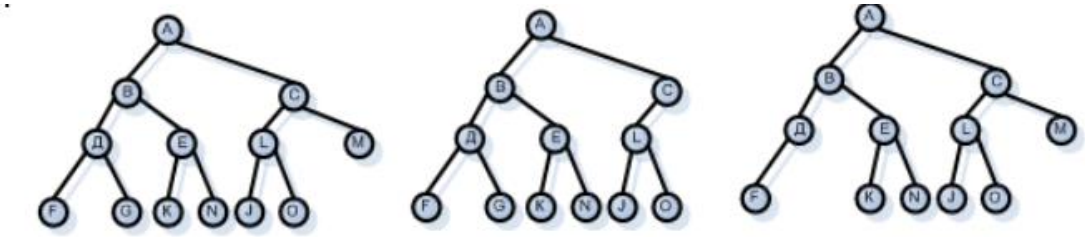


Рис. 4. Дерево 1 –законченное дерево, деревья 2 и 3 не являются законченными

Дерево 2 не является законченным деревом, так как оно неполное до уровня Height(T)-1 (у узла C нет правого поддерева). Дерево 3, так же не является законченным, так как узлы K, N, J, O не являются максимально приближенными к левому краю дерева.

Узлам законченного дерева можно присвоить индексы. Корневой узел будет иметь индекс 0, тогда для любого узла с индексом i, который имеет дочерние узлы, индекс левого дочернего вычисляется по формуле 2i+1, а правого 2i+2.

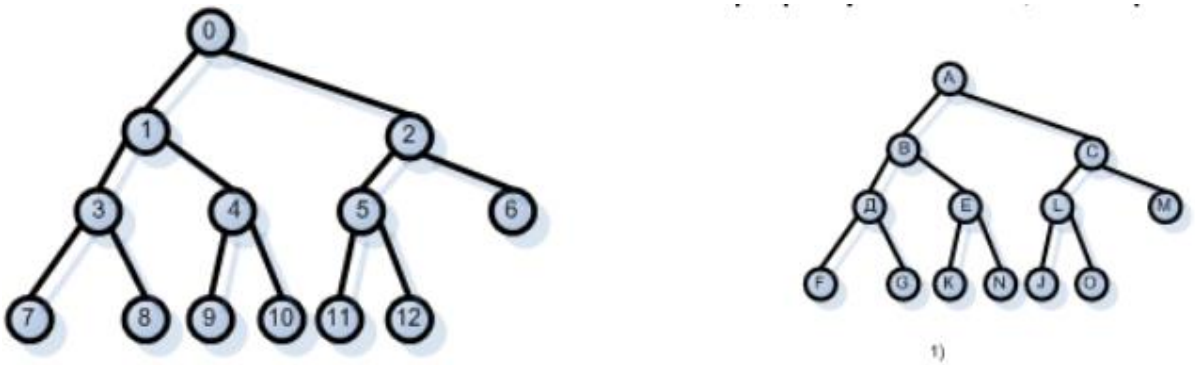



Рис.5. Индексированное законченное дерево

Так как узлы дерева можно индексировать, то его можно хранить в одномерном массиве.
Например, дерево 1 с рисунка 4, можно представить массивом

| A | B | C | D | E | L | M | F | G | K | N | J | O |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

## Идеально сбалансированное бинарное дерево

*Бинарное дерево называется идеально сбалансированным, если для каждой его вершины количество вершин в левом и правом поддереве различаются не более чем на 1.*

Пусть требуется построить бинарное дерево с n узлами и минимальной высотой (максимально "ветвистое"и"низкое").

Такие деревья имеют большое практическое значение, так как их использование сокращает машинное время, требуемое на выполнение различных алгоритмов.

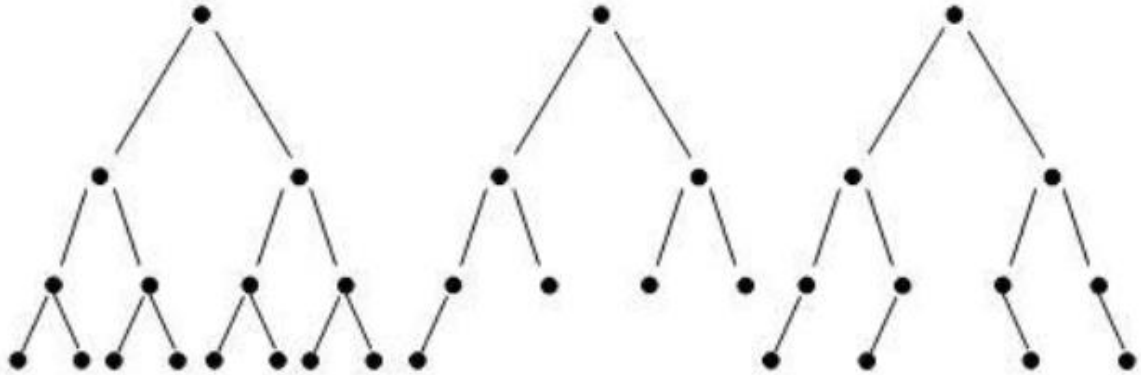


Рис.6. Примеры идеально сбалансированных бинарных деревьев

Алгоритм построения идеально сбалансированного дерева:
При известном числе вершин n лучше всего формулируется с помощью рекурсии. При этом необходимо лишь учесть, что для достижения минимальной высоты при заданном числе вершин, нужно располагать максимально возможное число вершин на всех уровнях, кроме самого нижнего. Это можно сделать очень просто, если распределять все поступающие в дерево вершины поровну слева и справа от каждой вершины.

**Суть алгоритма**

• взять одну вершину в качестве корня.

• построить левое поддерево с nl = n DIV 2 вершинами тем же способом.

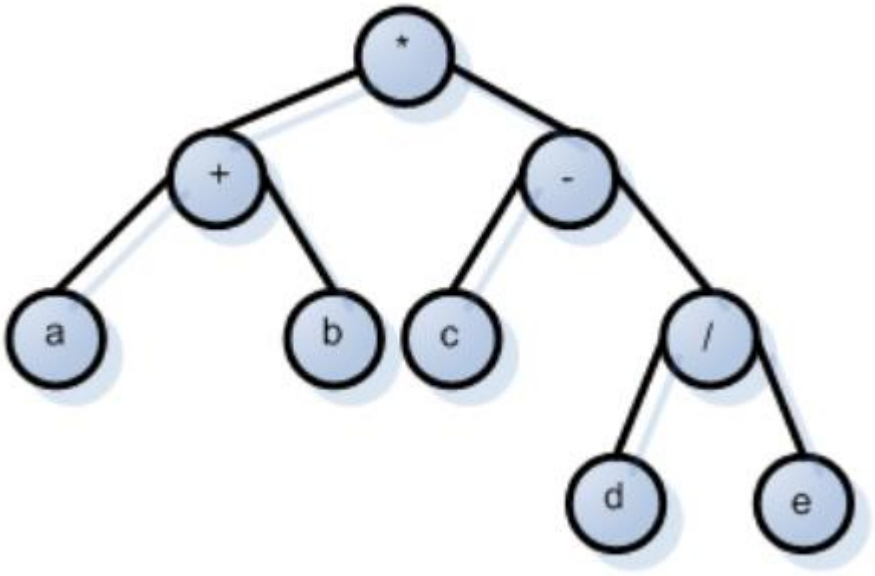
• построить правое поддерево с nr = n-nl-1 вершинами тем же способом.

```
createtree(Tree &T, int n)
int nl, nr, x;
if (n!=0)
создать узел T
nl=n/2;
nr=n-nl-1;
createtree (T.left,nl);
createtree (T.right,nr);
endif
```

## Бинарное дерево выражений

Выражение, содержащее бинарные операции можно представить как бинарное дерево, в котором внутренние узлы будут хранить операции,  а листья операнды.

Например, выражение **(a+b)*(c-d/e)** можно представить дерево T, в корневом узле которого будет находиться операция *, левое поддерево будет представлять левый операнд (левая скобка), а правое поддерево–правый операнд (правая скобка).




Если операнды–числа, то дерево можно использовать для вычисления значения выражения.
Если операнды - переменные, то в узел операнда записывается ссылка на ячейки переменных.

Дерево выражений можно использовать при вычислении значения выражения. При этом необходимо учитывать приоритетность операций и правило: сначала надо вычислить значение выражения левого поддерева, затем правого, а затем выполнить операцию, находящуюся в корне.

## Алгоритмы обхода бинарного дерева

Обход дерева – это алгоритм, который позволяет осуществить доступ ко всем узлам дерева, только один раз. Над данными узла можно выполнять операции.
Обход формирует список пройденных узлов.

**Алгоритмы обхода методом в “глубину”**

***1) Обход в прямом порядке бинарного дерева***

• посетить корневой узел
• обойти в прямом порядке левое поддерево
• обойти в прямом порядке правое поддерево

```
Pereorder (BinTree T)
if (! Empty(T))
Visit(Root(T));
Pereorder(LeftTree(T));
Pereorder(RightTree(T));
Endif
```

***2) Обход бинарного дерева в симметричном порядке***

• обойти в симметричном порядке левое поддерево
• посетить корневой узел
• обойти в симметричном порядке правое поддерево

```
void Pereorder (BinTree T){
if( ! Empty(T))
Pereorder(LeftTree(T));
Visit(Root(T));
Pereorder(RightTree(T));
endif
```

***3) Обход бинарного дерева в обратном порядке***

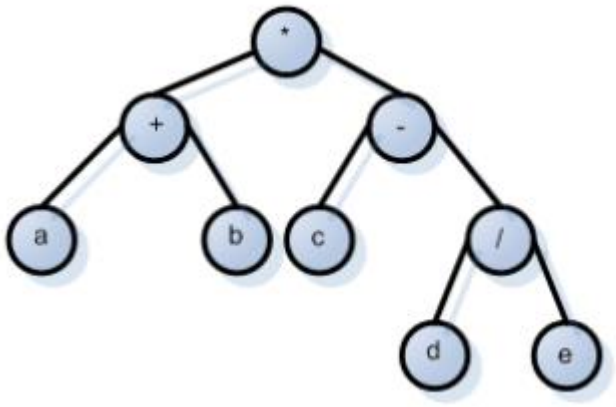
• обойти в обратном порядке левое поддерево
• обойти в обратном порядке правое поддерево
• посетить корневой узел

```
Pereorder (BinTree T)
If (!Empty(T) )
Pereorder(LeftTree(T));
Pereorder(RightTree(T));
Visit(Root(T));
endif
```

*Пример. Использование алгоритмов обхода для создания: Польской постфиксной записи выражения*

- Постфиксная запись (знак операции следует за операндами) выражения a+b/c имеет вид **abc/+** Польской префиксной записи выражения
- Префиксная запись(знак операции указывается перед операндами) выражения a+b/c имеет вид **+a/bc**
- Инфиксная запись (знак операции между операндами) выражения имеет вид **a+b/c**

При создании таких форм записи выражения используется алгоритм обхода дерева выражения.



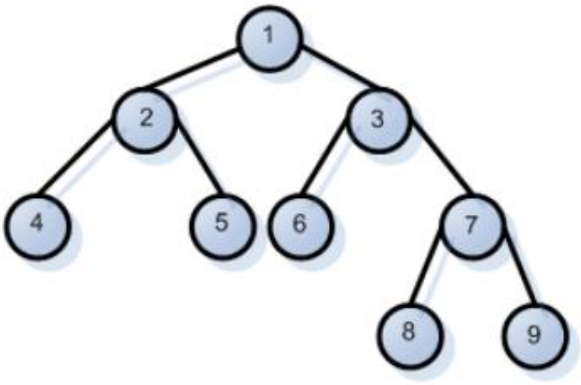

Обход дерева выражения в обратном порядке позволяет создать польскую постфиксную запись выражения.
ab+cde/-*

Обход дерева выражения в прямом порядке позволяет создать польскую префиксную запись выражения.
*+ab-c/de

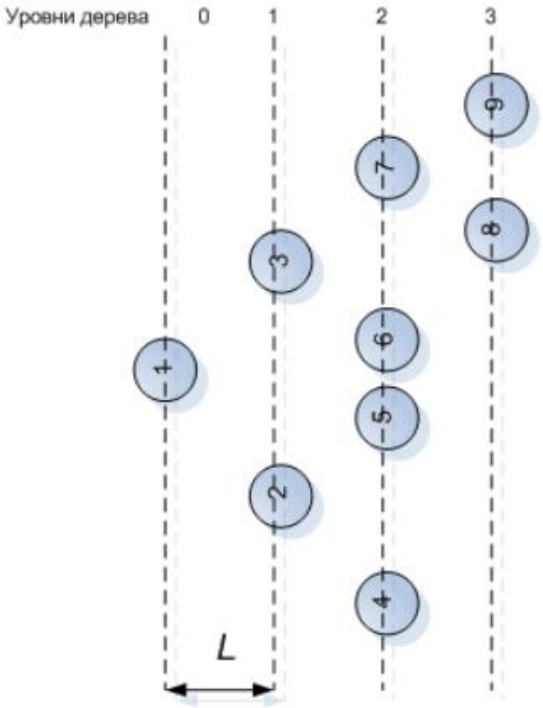
Обход дерева выражения в симметричном порядке позволяет создать инфиксную запись выражения.
(a+b)*(c-d/e)

*Пример. Использование алгоритма обхода для отображения бинарного дерева на  мониторе.*

Так как узлов в дереве может быть много, то при выводе на монитор выводить информацию надо построчно, то удобно дерево представить в перевернутом виде, как представлено на рисунке – вид 2.




1




2

// выводит дерево
// level –уровень выводимого узла.
// Количество пробелов от уровня 0 до уровня узла вычисляется по формуле: номер уровня * L

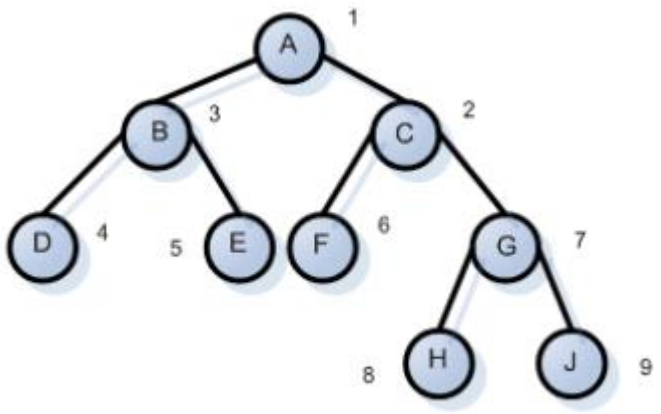
```
printBinTree(BinTree T.int level,int L){
int i;
If (! Empty(T))
printBinTree((rightTree(T), level+1,L);
for i←1 to i<=level*L;) do
cout<<’ ‘;
od
cout<<Data(T);
printBinTree((LeftTree(T), level+1,L);
endif
```

## Обход бинарного дерева алгоритмом “в ширину”

Такой обход выводит дерево по уровням. При обходе, пройденные узлы записываются в очередь. Затем из очереди извлекается узел и обрабатывается, а в очередь отправляются его сыновья (узлы левого и правого деревьев).

*Пример. Обход дерева в ширину*

Рассмотрим алгоритм обхода в ширину на приведенном дереве



1) В очередь помещается первый узел с меткой A
Очередь
Q: A
2) Из очереди извлекается узел A для обработки и в очередь помещаются его сыновья
Q: B C Вывод A
3) Из очереди извлекается узел B для обработки и в очередь помещаются его сыновья
Q: C D E Вывод B
4) Из очереди извлекается узел C для обработки и в очередь помещаются его сыновья
Q: D E F G Вывод C
5) Из очереди извлекается узел D для обработки и в очередь должны
помещаться его сыновья, но их нет, значит, ничего не помещается
Q: E F G
6) Из очереди извлекается узел E для обработки и в очередь должны помещаться его сыновья, но их нет, значит, ничего не помещается
Q: F G
7) Из очереди извлекается узел F для обработки и в очередь должны помещаться его сыновья, но их нет, значит, ничего не помещается
Q: G
8) Из очереди извлекается узел G для обработки и в очередь должны помещаться его сыновья,
Q: H J
9) Из очереди извлекается узел H для обработки и в очередь должны помещаться его сыновья, но их нет, значит, ничего не помещается
Q: J
10) Из очереди извлекается узел J для обработки и в очередь должны помещаться его сыновья, но их нет, значит, ничего не помещается
Q: Очередь пуста - дерево пройдено

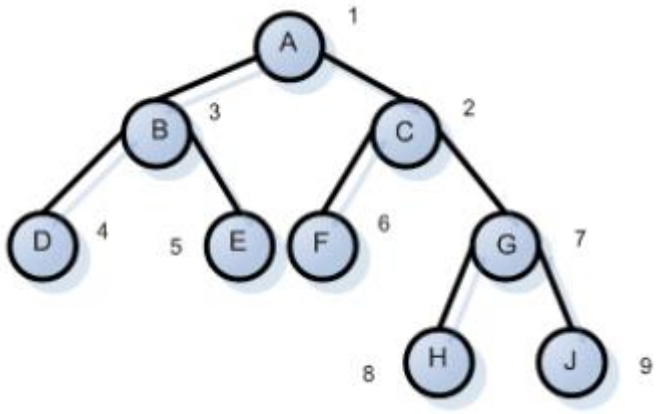
## Представление бинарного дерева в оперативной памяти

1. Таблица
2. Массив родителей
3. Массив законченного двоичного дерева
4. Реализация на указателях

**1. Таблица**

Таблица состоит из элементов, включающих ссылку на правое поддерево, левое поддерево и метку узла.
Данные узлов могут храниться в другой структуре данных, например, массив.

Рассмотрим представление в памяти дерева, элементы которого проиндексированы по мере его добавления.



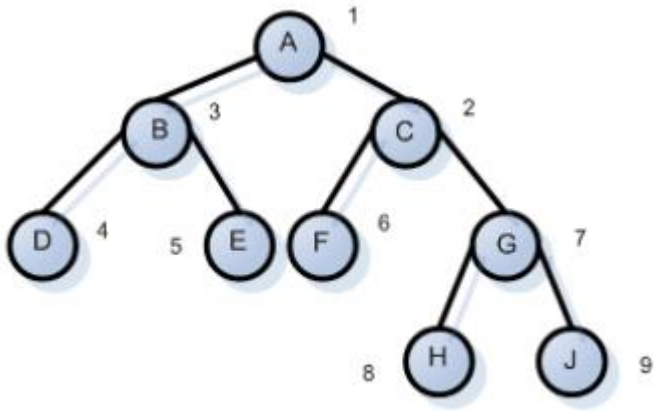

| Индекс узла | LeftTree | RightTree | label |
|---|---|---|---|
| 1 | 3 | 2 | A |
| 2 | 6 | 7 | C |
| 3 | 4 | 5 | B |
| 4 | 0 | 0 | D |
| 5 | 0 | 0 | E |
| 6 | 0 | 0 | F |
| 7 | 8 | 9 | G |
| 8 | 0 | 0 | H |
| 9 | 0 | 0 | J |

Определение реализации дерева в программе может быть таким:

```
struct Tnode{
int LeftTree;
int rightTree;
Tdata Label;
};
struct BinTree{
Tnode *Table;
int Root;
int N; //количество узлов
};
```

## 2. Массив родителей

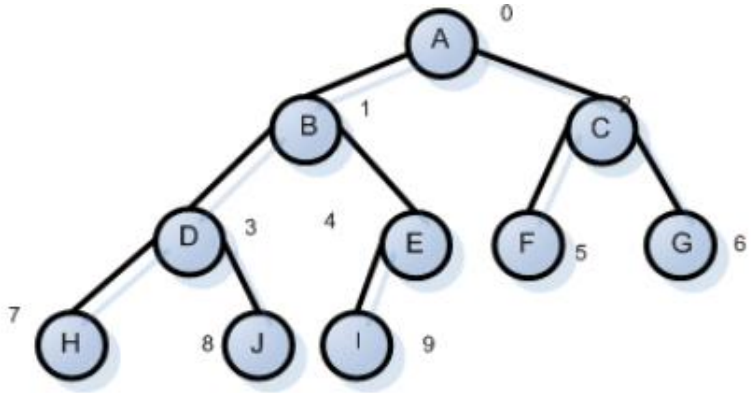
Бинарное дерево можно представит в памяти через массив родителей.




| 0 | 1 | 1 | 3 | 3 | 2 | 2 | 7 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | J |

## 3. Реализация законченного двоичного дерева

Согласно теории по данному дереву, его узлы можно индексировать: корневому узлу присвоить индекс 0, если i – индекс узла, то узел его правого поддерева имеет индекс 2i+2, а узел его левого поддерева будет иметь индекс 2i+1.




| A | B | C | D | E | F | G | H | J | I |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

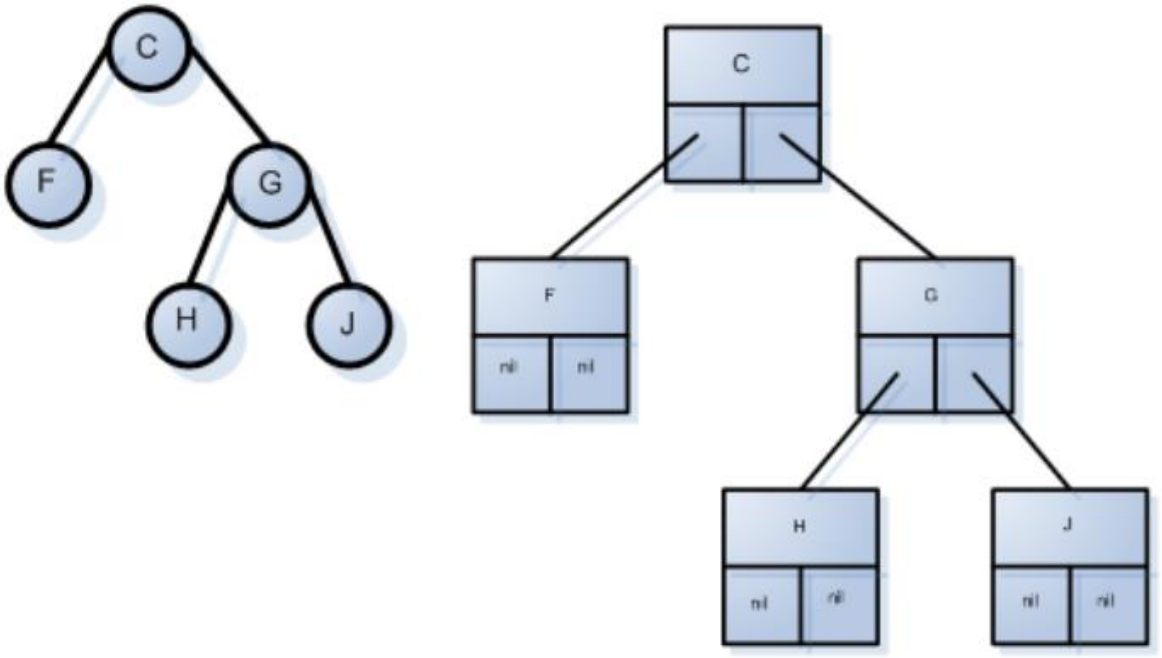
## 4. Реализация на указателях

Структура узла

| Область данных | |
|---|---|
| Ссылка на левое поддерево | Ссылка на правое поддерево |

• Область данных
• Ссылка на левое поддерево
• Ссылка на правое поддерево

Тогда бинарное дерево, со значениями, представляющими метки реализуется:



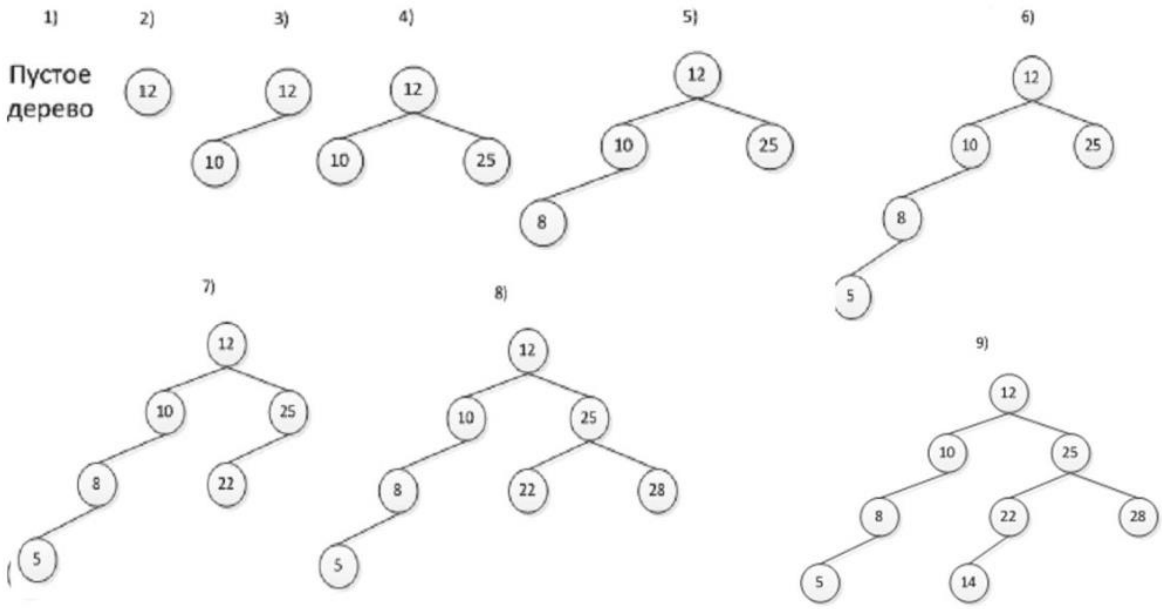
Программная реализация дерева на указателях.

```
struct Tnode{
Tdata Label;
Tnode *LeftTree;
Tnode * RightTree;
}
Tnode *RootTree; //указатель на дерево
```

**Бинарное дерево поиска**
Используется для поиска данных, снабженных ключом в динамических таблицах данных, т.е. в которых при выполнении поиска требуется удалять записи по ключу и вставлять записи в определенном порядке.

Данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше.
Информация каждого узла является структурой, и не является единственным полем данных.
**Определение бинарного поиска**
Это бинарное дерево, узлы которого упорядочены по значениям ключей по правилу:
• слева от корня узлы со значением ключа меньше, чем ключ в корне,
• а справа со значением ключа большими, чем в корне

Описание процесса построения дерева для ключей
12 10 25 8 5 22 28 14



1. Сначала дерево пустое.

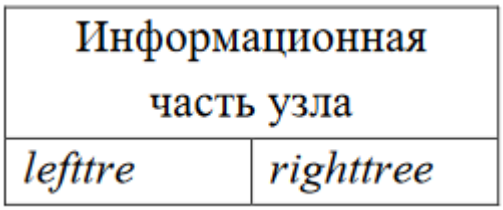
2. Считываются последовательно ключи.
2.1. Ключ 12. Так как дерево пусто, то ключ вставляется в корень – первый узел.
2.2. Ключ 10. Выполняется поиск места вставки нового ключа, при этом выполняется обход дерева в соответствии со свойством дерева бинарного поиска: значение 10 сравнивается со значением ключа в корне, так как 10<12, то поиск места вставки осуществляется в левом поддереве, но пока левое поддерево пусто, то вставка происходит в корень этого поддерева.
2.3. Ключ 25. Аналогично п.2.2. осуществляется поиск места вставки ключа 25, при этом сравнивается значение ключа в корне, та как 25>12, то этот ключ должен разместиться в правом поддереве дерева с корнем 12, так как правое поддерево узла со значением 12 пусто, то 25 становиться корнем правого поддерева дерева с корнем 12.
2.4. Ключ 8. Так как 8<12, то ключ 8 должен быть расположен в левом поддереве дерева с ключом 12, но левое поддерево не пусто, тогда сравниваем ключи 8<10, тогда ключ 8 должен разместиться в левом поддереве дерева с корнем 10.

**Способы реализации бинарного дерева поиска**
Как и к-арное дерево, бинарное дерево поиска можно реализовать:

- на массиве родителей;
- на списках левых сыновей и правых братьев;
- на таблице, со ссылками на сыновей;
- на курсорах;
- на явно представленной иерархической структуре, в которой отношения между узлами (поддеревьями) явно реализуются посредством указателей.

Так как дерево поиска применяется к динамическим таблицам данных, т.е. таким, в которых при выполнении поиска требуется удалять записи по ключу и вставлять записи в определенном порядке, то и само дерево является динамической структурой.

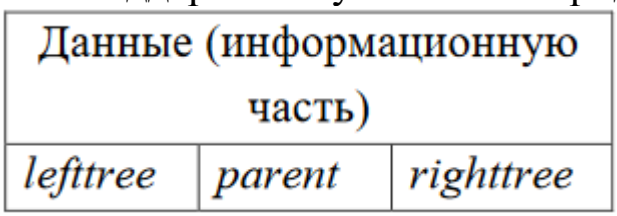
**Способ 1. Указатели на левое и правое поддеревья**
Узел содержит: информационную часть узла (или ссылку на данные в другой структуре данных) и указатели на левое и правое поддеревья.




lefttree – указатель на левое поддерево
righttree – указатель на левое поддерево
Определение структуры узла, где информационная часть узла представлена ключом (целое число – код направления бакалавриата) и данными (строка – Название направления)

```
struct Node{
int key; string Name;
    Node *lefttree, *righttree;
    };
    Node *Root;
```

**Способ 2. Указатели на левое и правое поддеревья и на родительский узел**
Узел содержит: информационную часть узла и указатели на левое и правое поддеревья и указатель на родительский узел




lefttree – указатель на левое поддерево
righttree – указатель на левое
parent - указатель на родительский узел

```
struct Node{
int key; string Name;
Node *lefttree, *righttree;
Node *parent;
};
Node *Root;
```

**Операции над бинарным деревом поиска**

1. Обход бинарного дерева поиска (алгоритмы обхода бинарного дерева)
2. Создание узла бинарного дерева.
3. Вставка узла в бинарное дерево поиска.
4. Удаление узла из бинарного дерева поиска.
5. Поиск узла, содержащего заданный ключ.

**Абстрактный тип «Бинарное дерево поиска»**

При разработке программы для поставленной задачи разработчик определяет постановку задачи, определяя данные задачи и ее то, что должно быть получено в результате решения задачи.

Абстракция данных позволяет определить данные задачи и действия, которые с данными требуется выполнить, только конкретно для рассматриваемой задачи. При этом представление данных в памяти пока не оговаривается, алгоритмы реализации действий не приводятся.

Абстрактный тип данным (далее АТД) определяет данные и операции (действия) над ними, пока без указания способа реализации данных.

АТД для бинарного дерева поиска, с определенным набором операций, для удобства демонстрации алгоритмов.

АТД BinSearchTree
{
Данные
Root – корень дерева key – ключ узла типа Tkey
data – информационная часть узла дерева типа Tdata
leftTree – левое поддерево
rightTree - правое поддерево
Операции
//Создание узла дерева
createNode(key,data) – возвращает созданный узел дерева - BinSearchTree
//Вставка узла в дерево
insertBinSearchTree(T, node) – вставляет узел node в бинарное дерево поиска
//Удаление узла из дерева
deleteBinSearchTree(T, key) – удаляет узел с ключом key из дерева
// получить значение ключа узла node
getKey(node)
//получить левое поддерево дерева T
getLeftTree(T)
//получить правое поддерево дерева T
getRightTree(T) – возвращает правое поддерево дерева T
//пусто ли дерево
isEmpty(T) – возвращает true если дерево пусто и false если не пусто
//сделать дерево пустым
MakeNull(T)
}

**Алгоритмы операций над бинарным деревом поиска**

**Создание узла бинарного дерева поиска**

Структура узла на указателях

```
typedef string Tdata;
struct Node
{
int key; //или может быть Tkey
Tdata data;
Node *lefttree, *righttree;
};
typedef Node* BinSearchTree; //тип для указателя на узел
```

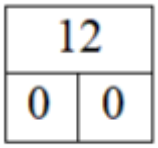
Реализация операции createNode

```
//предусловие: data – значение информационной части узла, key –
//ключ идентификации данных в узле
//постусловие: создает узел в динамической памяти, заполняет его
данными и возвращает на него указатель.
BinSearchTree createNode(int key, Tdata data){
Node * node=new Node;
node->key=key;
node->data=data;
node->lefttree=0;
node->righttree=0;
return node;
}
```

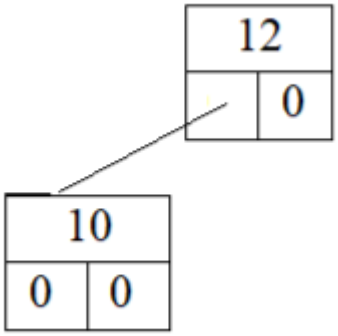
**Вставка узла в бинарное дерево поиска**

```
//предусловие: T – корень дерева (указатель на узел в вершине),
node – указатель на заполненный данными узел
//постусловие: вставляет узел node в дерево T
insertBinSearchTree(BinSearchTree T, BinSearchTree node){ Пустое
дерево if (isEmpty(T ))
T←node
else
if (getKey(node)<Key(T))
insertBinSearchTree(getLeftTree,node)
else
if (getKey(node)>Key(T))
insertBinSearchTree(getRightTree,node)
}
```

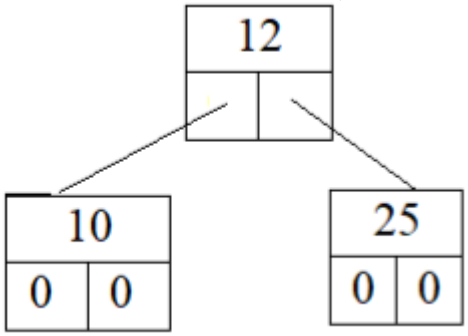
Пример построения дерева из четырех узлов и протестируем работу функции insertBinSearchTree. Выполнение алгоритма продемонстрированы на рисунке.
Для этого выполним последовательно четыре вызова:
1) Пустое дерево
2) Вставка первого узла
insertBinSearchTree(Root, node(12))



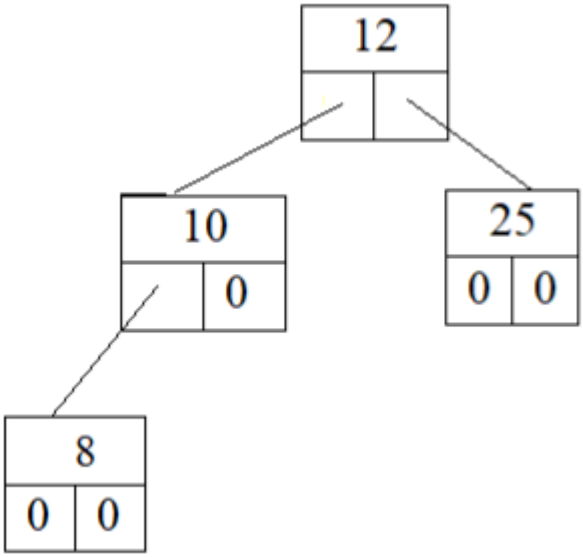

3) Вставка второго узла
insertBinSearchTree(Root, node(10))




4) Вставка третьего узла
insertBinSearchTree(Root, node(25))

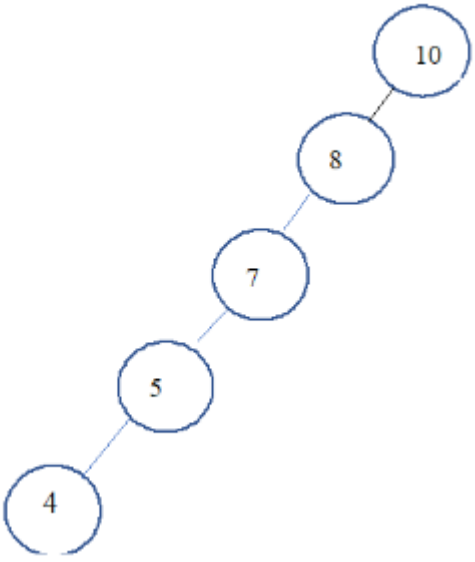



5) Вставка четвертого узла

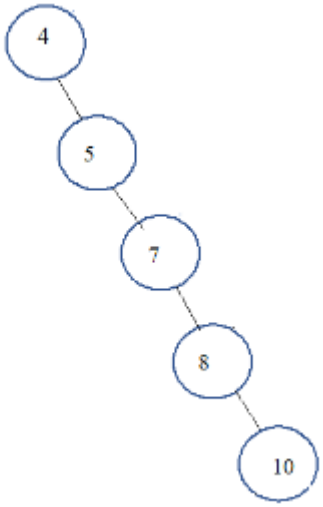



Дерево бинарного поиска может быть вырождено в список, т.е. может иметь только одно поддерево: только левое или только правое.

Пример. Включение в бинарное дерево поиска ключей, упорядоченных по убыванию. Дан список записей с ключами: 10 8 7 5 4. Изобразить дерево, включая ключи в указанном порядке.



Пример. Включение в бинарное дерево поиска ключей, упорядоченных по убыванию. Дан список записей с ключами: 4 5 7 8 10. Изобразить дерево, включая ключи в указанном порядке.




**Алгоритм сортировки списка значений и алгоритм симметричного обхода методом в глубину бинарного дерева поиска**

Так как бинарное дерево поиска является бинарным деревом, то посещение его узлов можно выполнить алгоритмами обхода бинарного дерева, реализованных как по методу в глубину, так и по методу в ширину.
Алгоритм симметричного обхода бинарного дерева поиска позволяет выполнить сортировку элементов, размещенных в дереве.
Выполним обход дерева, представленного на рис.1 алгоритмом симметричного обхода

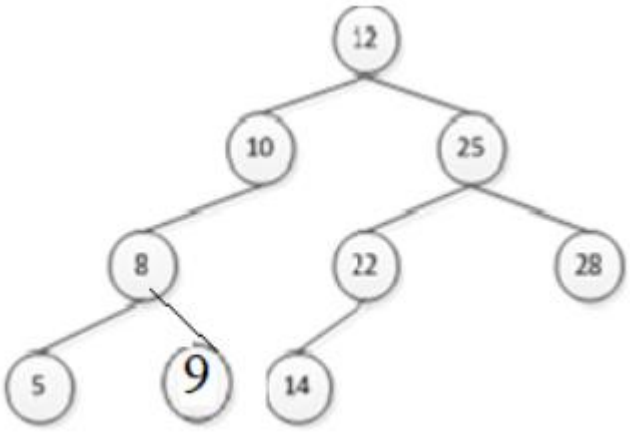



Рисунок 1 Дерево бинарного поиска
Результат симметричного обхода: 5, 8, 9, 10, 12, 14, 22, 25, 28

**Поиск узла с заданным ключом**
Алгоритм аналогичен алгоритму вставки узла.
Данный алгоритм принимает на вход ключ поиска, обходит бинарное дерево поиска в соответствии со свойством бинарного дерева поиска. Возвращает указатель на узел со значением ключа или NULL, если ключ не найден.

Выполним процесс поиска ключа со значением ключа 8 в дереве, представленном на рисунке 2.
Понятно, что обходим дерево с корня. Так как 8<12 (ключа в корне), то поиск продолжаем в левом поддереве корня с ключом 12, т.е. обходим дерево с ключом 10, т.к. 8<10, то поиск продолжаем в левом поддереве дерева с ключом 10. Так как ключ в корне левого поддерева совпал с ключом поиска, то процесс поиска завершается, алгоритм возвращает указатель на найденный узел, который хранит данные, идентифицируемые ключом поиска.

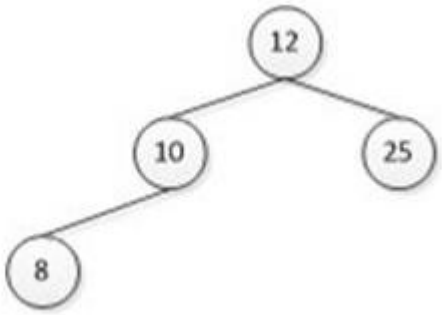



Рисунок 2. Дерево поиска для рассмотренного примера.

Алгоритм поиска ключа по бинарному дереву поиска
//предусловие: T – корень дерева, key – ключ поиска
//постусловие: результат – указатель на узел с заданным ключом при успешном поиске, и
значение NULL при безуспешном поиске
SearchBinSearchTree(BinSearchTree T, Tkey key){
if(isEmpty(T)) вернуть NULL
else
if(key<getKey(T)) SearchBinSearchTree (getLeftTree(T),key)
else
if(key)>getKey(T)) SearchBinSearchTree (getRightTree(T), key);
else
if(key=getKey(T)) вернуть T
}

**Удаление узла из дерева по значению ключа**

Удаление узла из дерева сводиться к операциям:

1. Найти узел в дереве (получить на него ссылку)
2. Удалить узел. После удаления узла упорядоченность дерева должна сохраниться. Поэтому при удалении узла возможен один из 4-х случаев:
   2.1. Удаляемый узел – лист.
   2.2. Удаляемый узел имеет правое поддерево, но не имеет левого.
   2.3. Удаляемый узел имеет только левое поддерево, но не имеет правого.
   2.4. Удаляемый узел имеет два поддерева.

Идея алгоритма: подобрать узел в дереве (замещающий узел), который бы заменил удаляемый узел и при этом свойство дерева бинарного поиска не нарушилась.

Рассмотрим все случаи удаления из дерева (с 2.1. по 2.4) на примерах, чтобы понять суть алгоритма.

Введем обозначения:

D – удаляемый узел.

R – замещающий узел (узел, на который будет заменяться удаляемый узел в некоторых
указанных случаях).

P – родитель удаляемого узла.

Pr – родитель замещающего узла.

**Случай 2.1. Удаляемый узел – лист**

Удаляется узел (D) со значением 130 - этот узел лист.

Алгоритм:

1. Спуск по дереву алгоритмом поиска с сохранением ссылки на родителя.
2. Необходимо просто у родителя этого узла установить в NULL ссылку на правое (в примере так) поддерево:

P->right = NULL.

Замещающий узел в этом случае не используется.

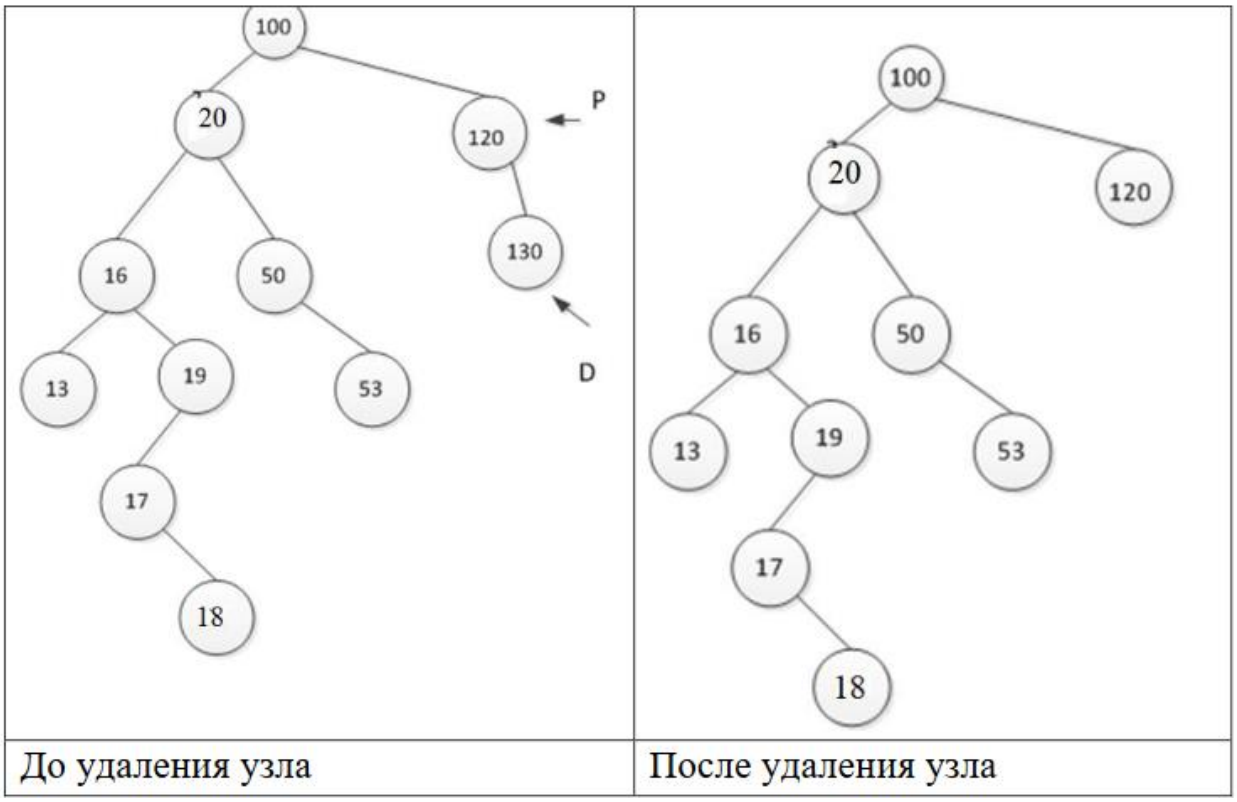



Рисунок. Удаление листового узла

**Случай 2.2. Удаляемый узел имеет левое поддерево, но не имеет правого**

Пусть удаляется узел со значением 19.

Этот узел имеет одно поддерево по левой ветви. Сам узел 19 является правой ветвью узла 16, своего родителя P, тогда удаление сводится к операции замещения удаляемого узла D его узлом в левой ветви (на узел 17).

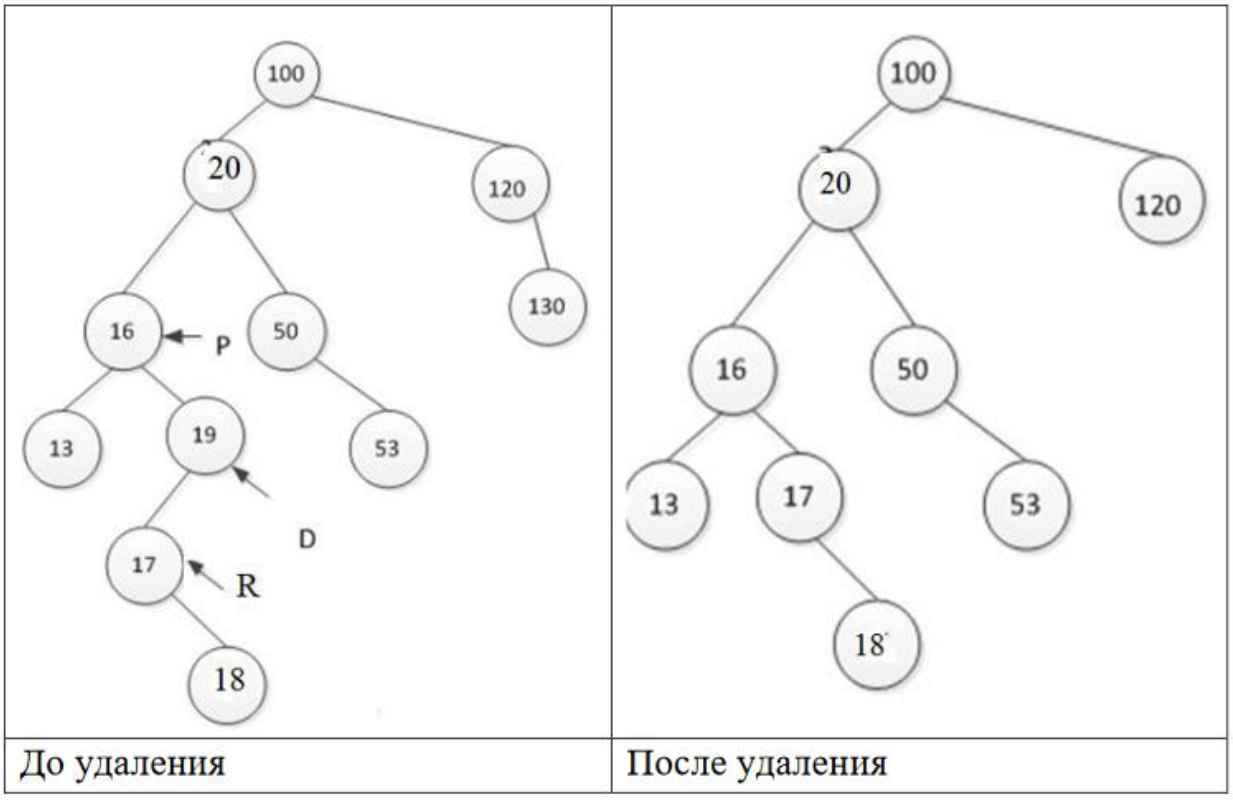



Рисунок. Удаление узла, имеющего только одно поддерево - левое

Алгоритм

1. Находим удаляемый узел D.
2. Проверяем. что у него одно поддерево - левое.
3. Запоминаем родителя удаляемого узла – P.
4. Запоминаем замещающий узел: R= D->left.
5. Заменяем P->right=R. Вместе с одним поддеревом.

**Случай 2.3. Удаляемый узел имеет только правое поддерево, но не имеет левого.**

Пусть удаляется узел со значением 50. Этот узел имеет одно поддерево по правой ветви.

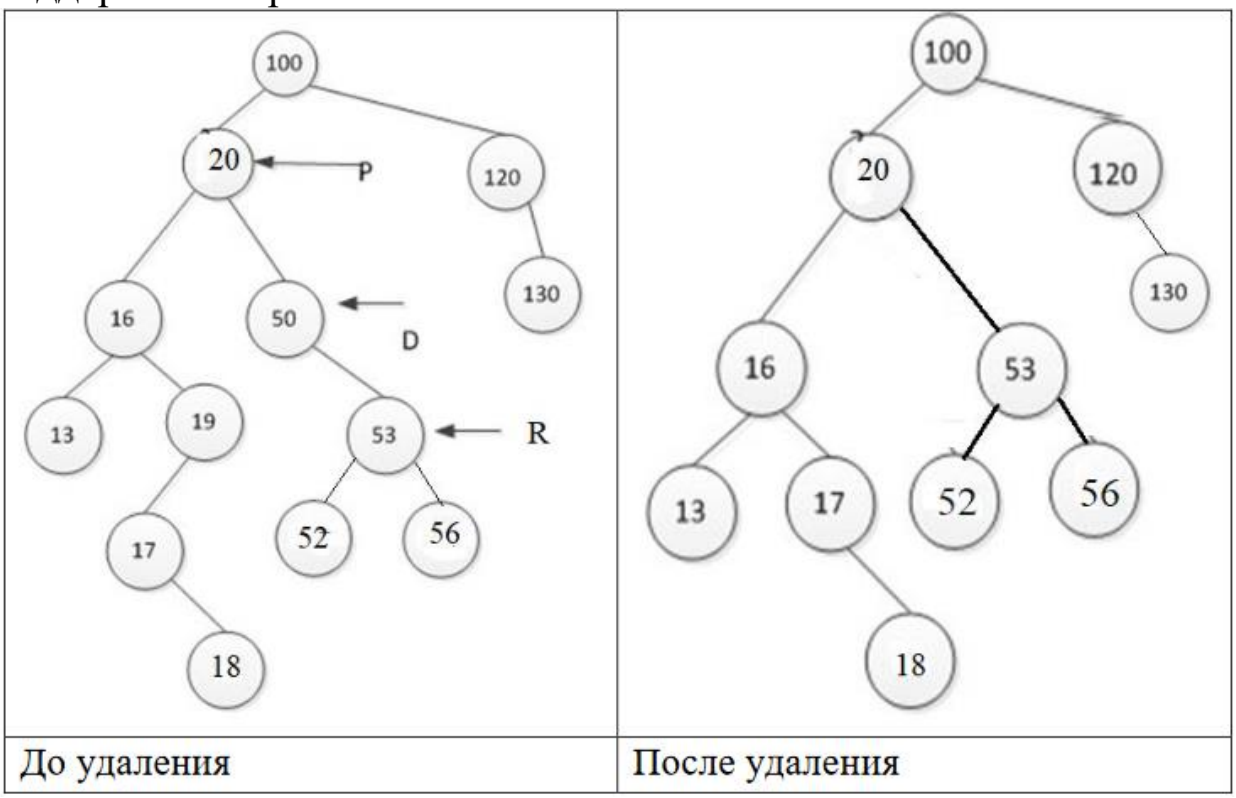



Рисунок. Удаление узла, имеющего только одно поддерево - правое

Сам узел 50 является правой ветвью узла 20 своего родителя P, тогда удаление сводится к операции замещения удаляемого узла D его узлом в правой ветви (на 53):

Алгоритм
R= D->right
P->right=R.

**Случай 2.4. Удаляемый узел имеет два поддерева**

Пусть удаляется узел со значением 20.

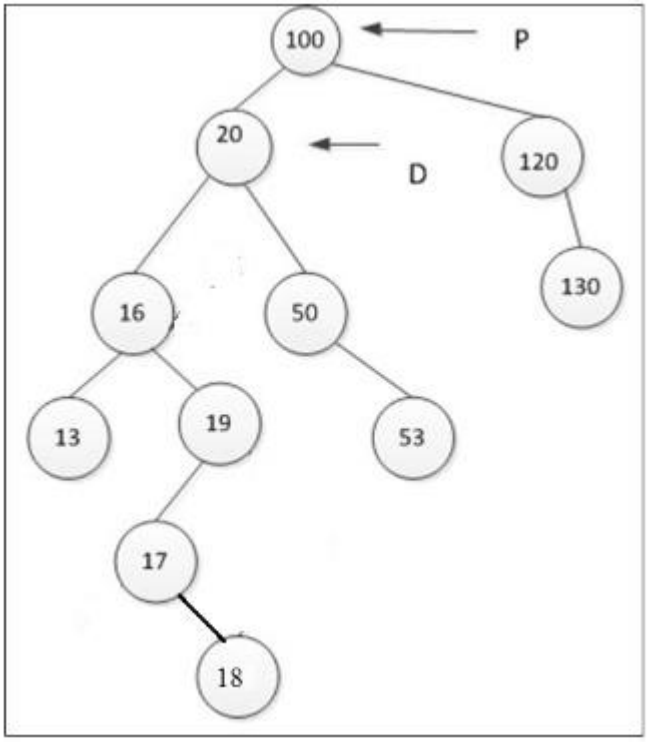



Рисунок. Удаление узла, имеющего два поддерева

При поиске замещающего узла можно использовать 1 из 2-х подходов:
a) замещающий узел – это узел с максимальным значением ключа в левом поддереве удаляемого узла, т.е. самый правый левого поддерева удаляемого узла;
b) замещающий – это узел с минимальным значением ключа правого поддерева удаляемого узла т.е. самый левый правого поддерева удаляемого узла.

Алгоритм удаления - а)

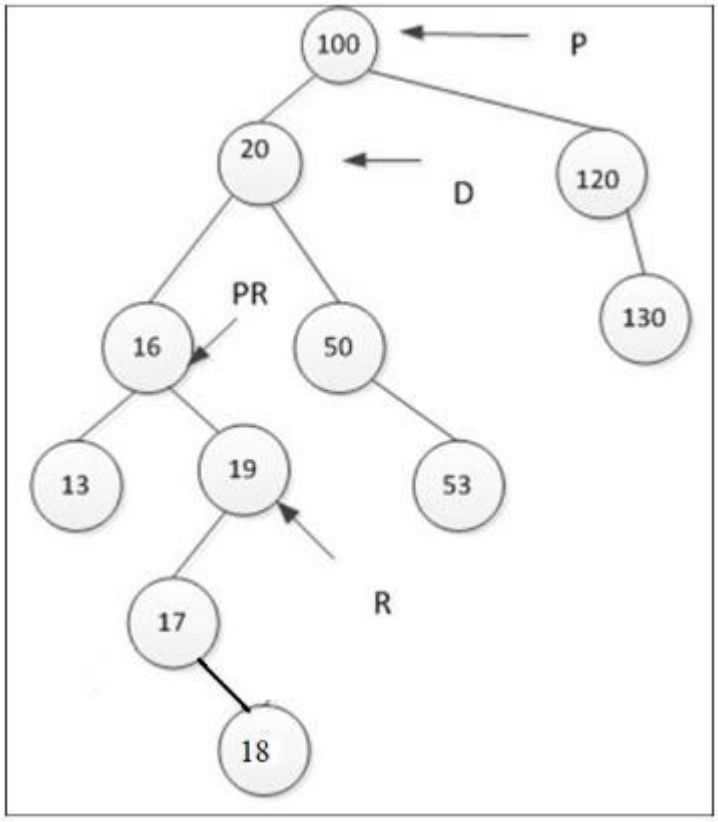



Рисунок. Дерево до удаления узла, имеющего два поддерева замещением на самый правый узел левого поддерева удаляемого узла

Этот алгоритм включает следующие действия:
1. Найти замещающий узел R, спускаясь по левому поддереву удаляемого узла и найти
самый правый узел.
Результат поиска предполагает 2 случая:
c) правое поддерево не пусто, то проход завершается узлом, имеющим только левое поддерево или листом, если R лист;
d) правое поддерево окажется пусто.
Случай c). Тогда D это - 20, его родитель P-100, замещающий узел - самый правый в левом поддереве узла D, тогда R – заменяющийся узел - 19, Pr – родитель заменяющегося узла R это узел со значением 16 и R не лист, тогда алгоритм сводиться к операциям:
Pr=D->left;
R=Pr->right;
Pr->right=R->left;
D->data=R->data

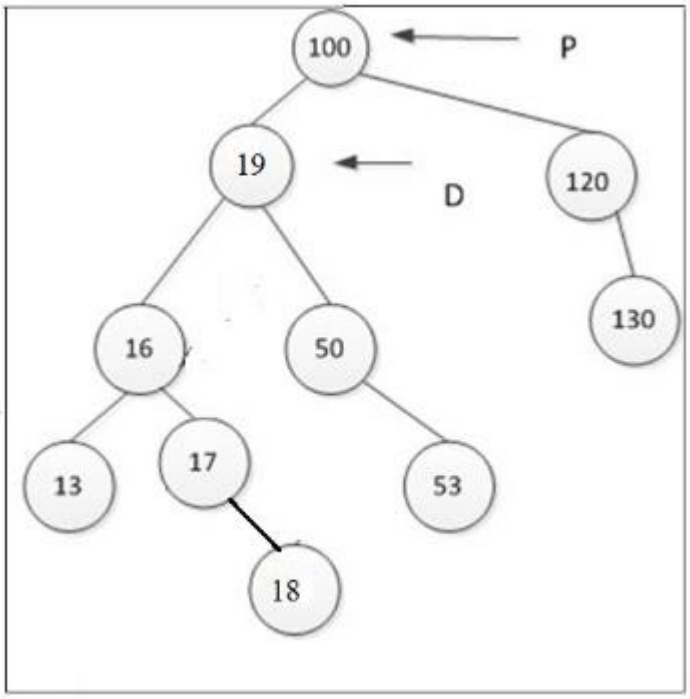



Рисунок. Дерево после удаления узла, имеющего два поддерева замещением на самый правый узел левого поддерева удаляемого узла (случай c)

Случай d) если в левом поддереве удаляемого узла правое поддерево пусто.

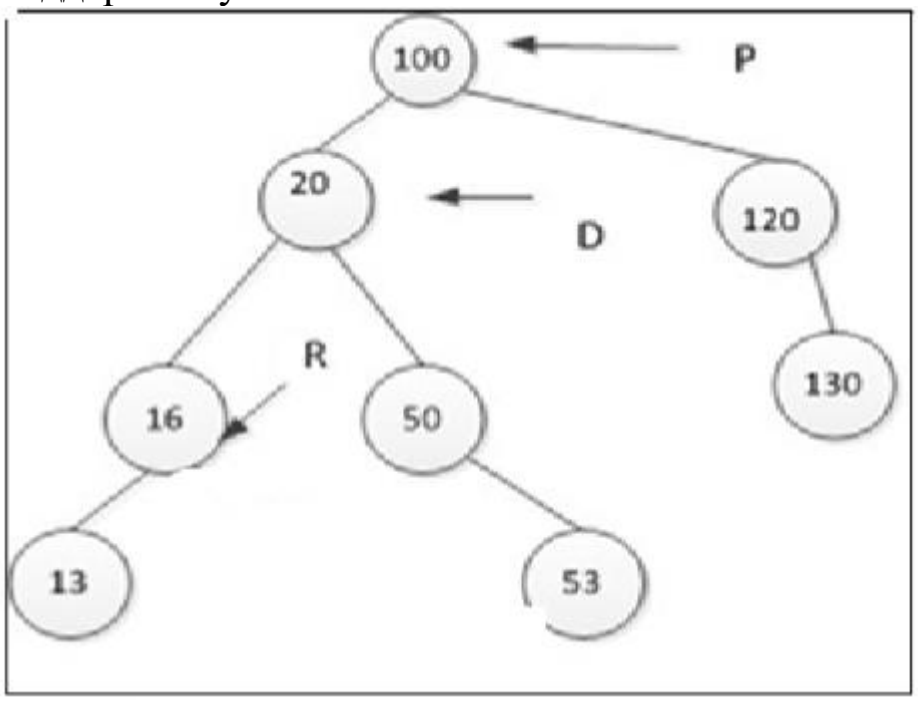



Рисунок. Дерево до удаления узла, имеющего два поддерева замещением на самый правый узел левого поддерева удаляемого узла (случай d)

Алгоритм
R=D->left;
D->data=R->data;
D->left=R.

**Обобщенный алгоритм удаления узла**
1. Найти удаляемый узел и запомнить на него ссылку D
2. Определить есть ли у него поддеревья:
- Одно правое. Ищем замещающий узел R и выполняем перестройку дерева.
- Одно левое. Ищем замещающий узел R и выполняем перестройку дерева.
- Нет поддеревьев-оно лист. Заменяем у родителя соответствующую ссылку на NULL.
- Если два поддерева, то выбираем один из вариантов поиска замещающего узла:
а) либо самый правый в левом поддереве;
б) либо самый левый в правом поддереве удаляемого узла.
И рассматриваем два случая:
Либо дерево, в котором ищем замещающий узел пусто, либо нет, тогда тоже два случая:
либо поддерево (для случая а) завершается листом если R лист, либо, завершается узлом, имеющим только левое поддерево.

```
deleteBinSearchTree(T, key){
  If(!isEmpty(T)) ---- не пусто
  { //поиск удаляемого узла
     if(key<getKey(T))
         deleteBinSearchTree(LeftTree(T),key);
     else
         if((key>getKey(T))
              deleteBinSearchTree(RightTree(T),key);
         else
{// нашли удаляемый узел - простые удаления
                  D=T;
                  if(isEmpty(getRightTree(D)) //правое пусто
                 {
                      T<-getLeftTree(T)
```

```
                }
                else
                    if(isEmpty(getLeftTree(D) ) //леаое пусто
                    {
                        T<-RightTree(T)
                    }
                    else
                    {//реализуем случай A)
                       ищем замещающий узел
                       Del(key, getLeftTree(D),T)
                    if (D- не пусто) delete D;
                    }
  }
}
//Алгоритм поиска замещающего узла
Del(key,T,R){
If(!isEmpty(RightTree(T)) //не пусто
{//спуск по правому дереву до листа или с одним левым п.д.
  Del(key, RightTree(T) ,R)
}
else
{
//Заменить данные R на данные T
T=LeftTree(T);
}
}
```