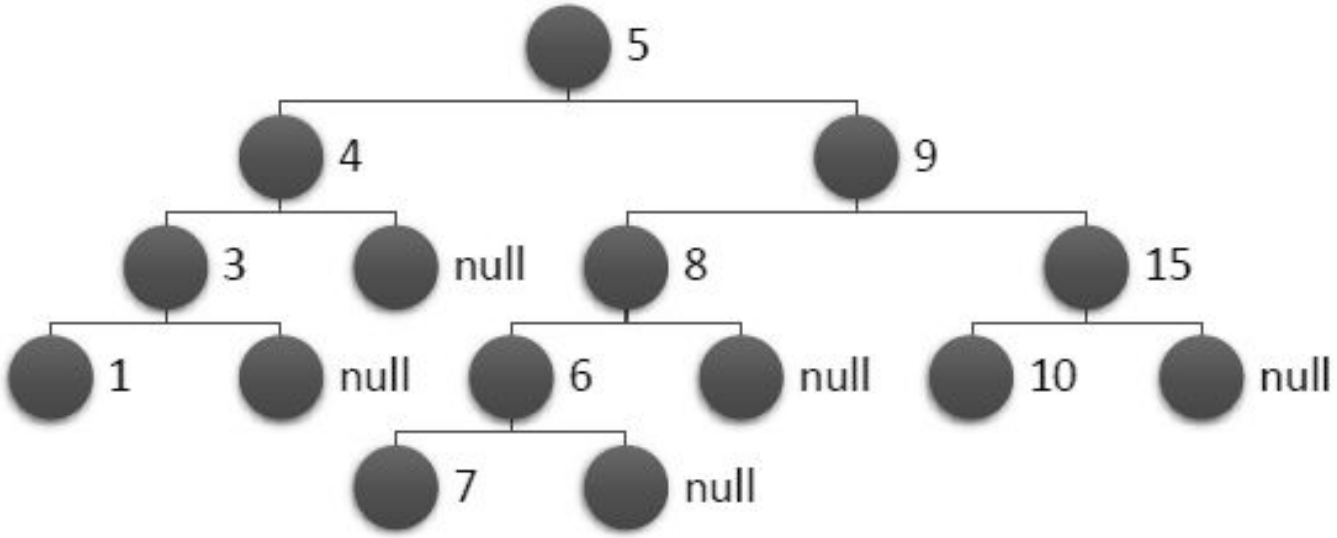
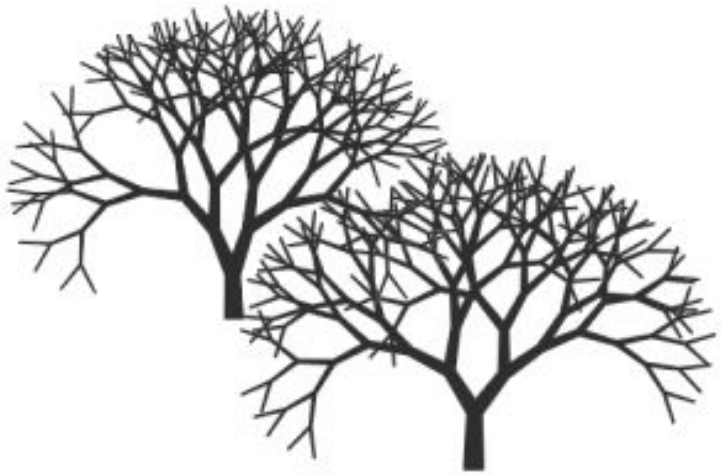
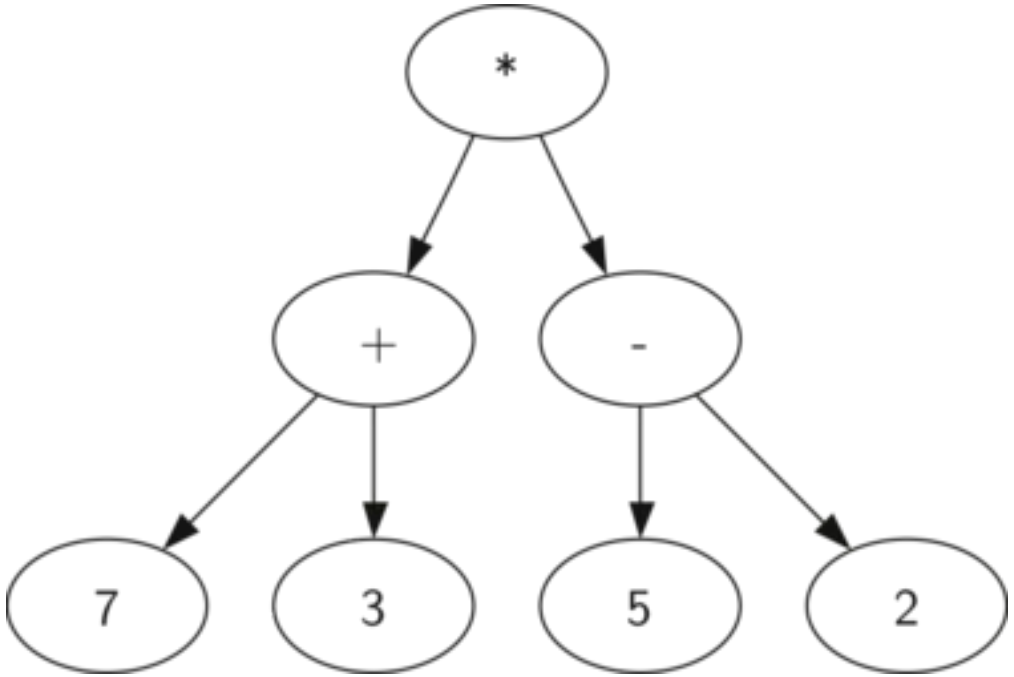
Деревья

Дерево – структура данных, представляющая собой древовидную структуру в виде набора связанных узлов. Является связным графом, не содержащим циклы.

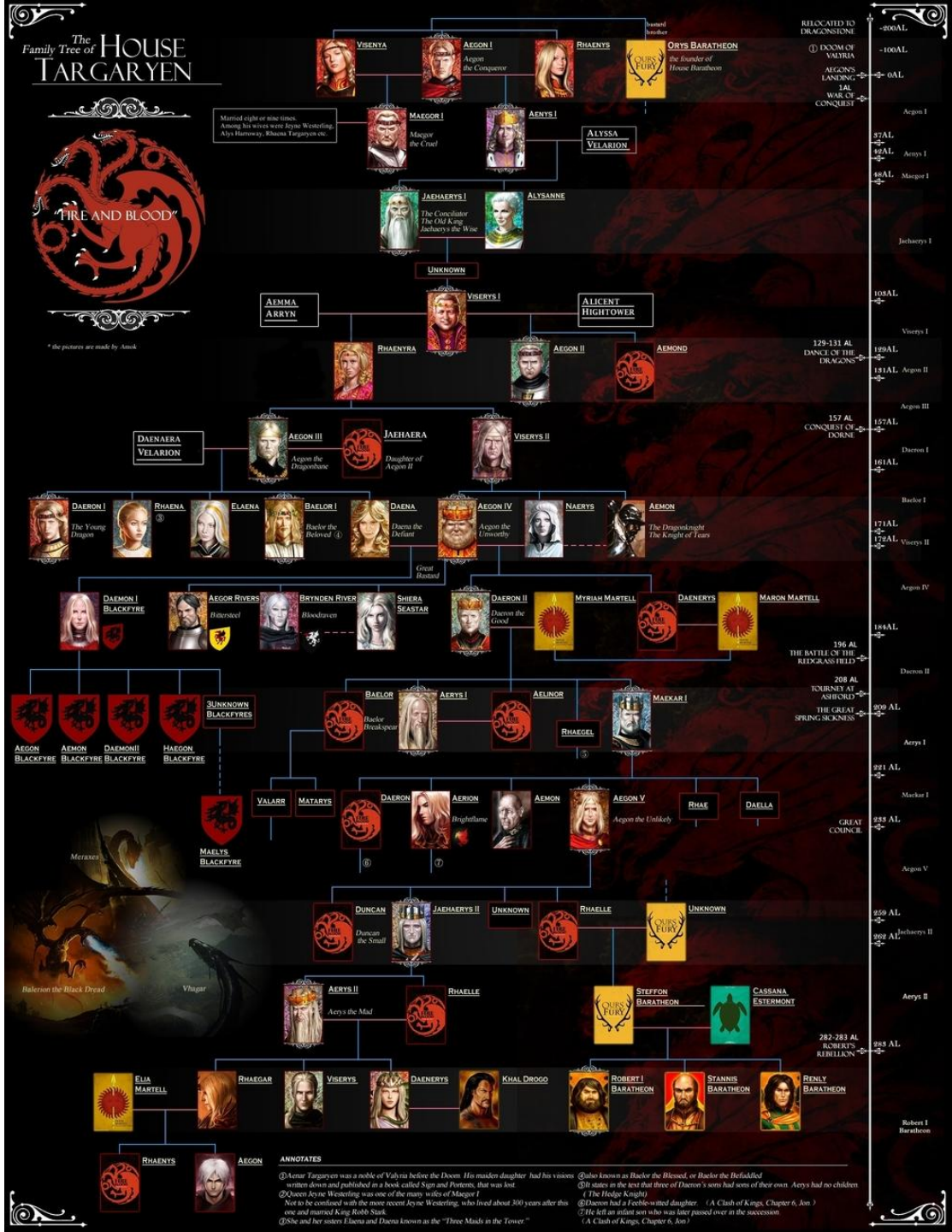




Синтаксическое дерево – это дерево в котором внутренние вершины это операторы языка программирования, а листья операнды

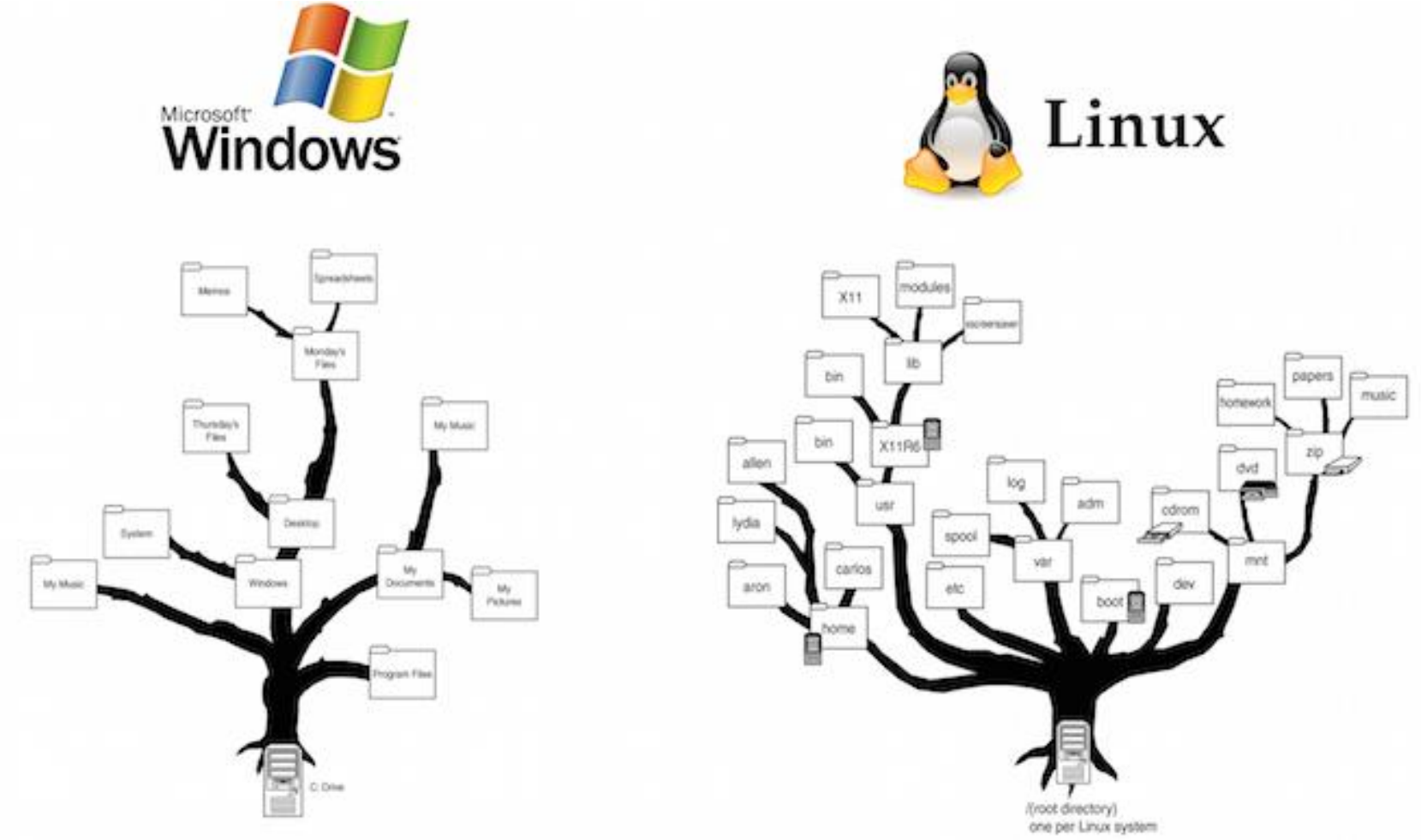


Генеалогическое древо

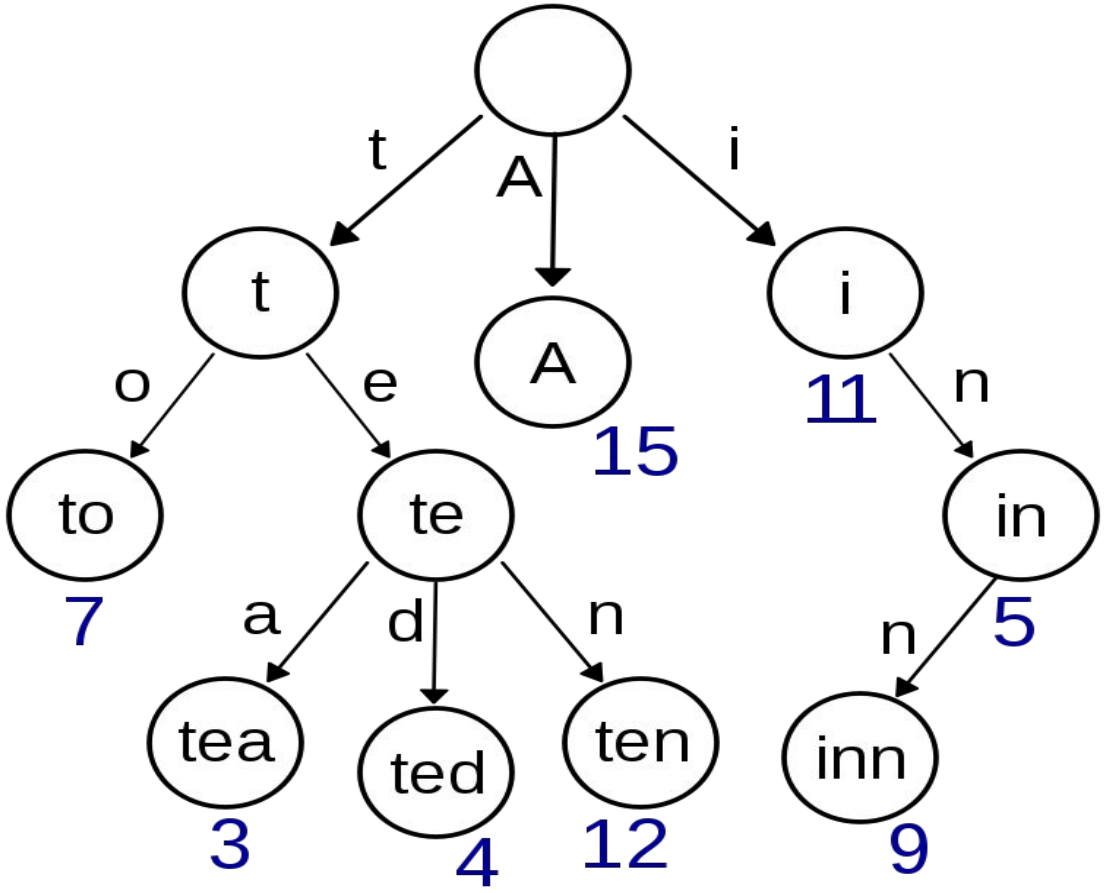


Логические файловые системы

- Деревья используются в файловых системах ОС.



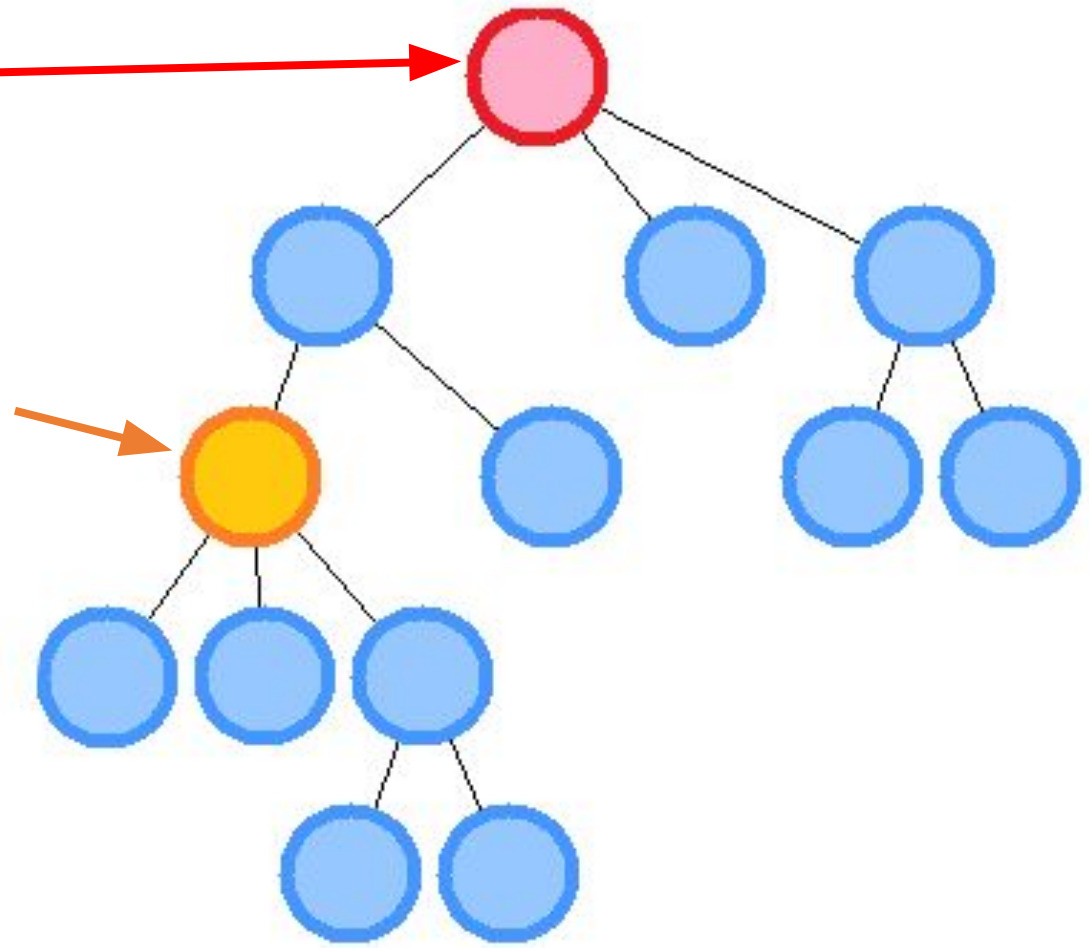
Префиксное дерево – это дерево содержащее все суффиксы некоторой строки и позволяют осуществить поиск подстроки в строке



Деревья решений (decition trees) используются в классификации

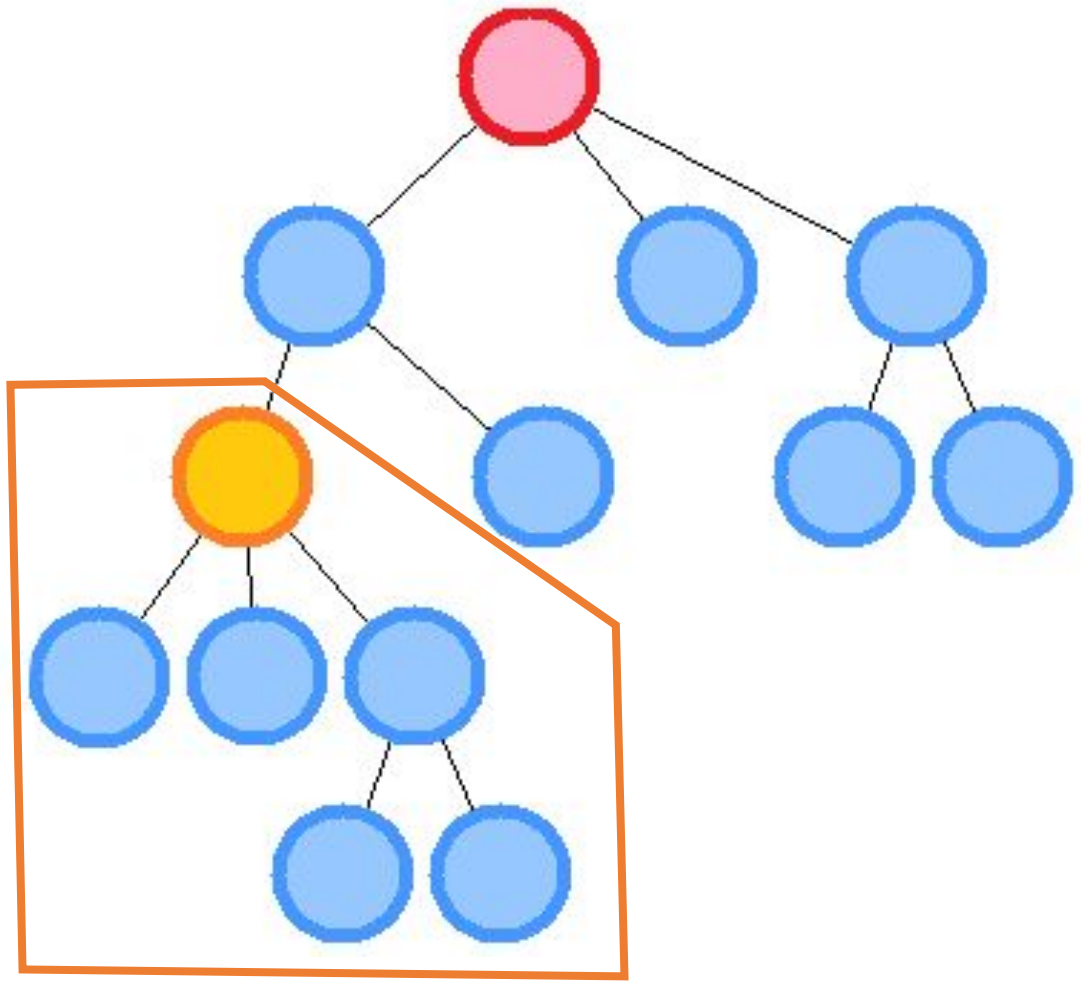
Определения

- **Корневой узел, корень дерева** — самый верхний узел дерева.
- **Корень поддерева** — одна из вершин, по желанию наблюдателя, которая имеет дочерние элементы.



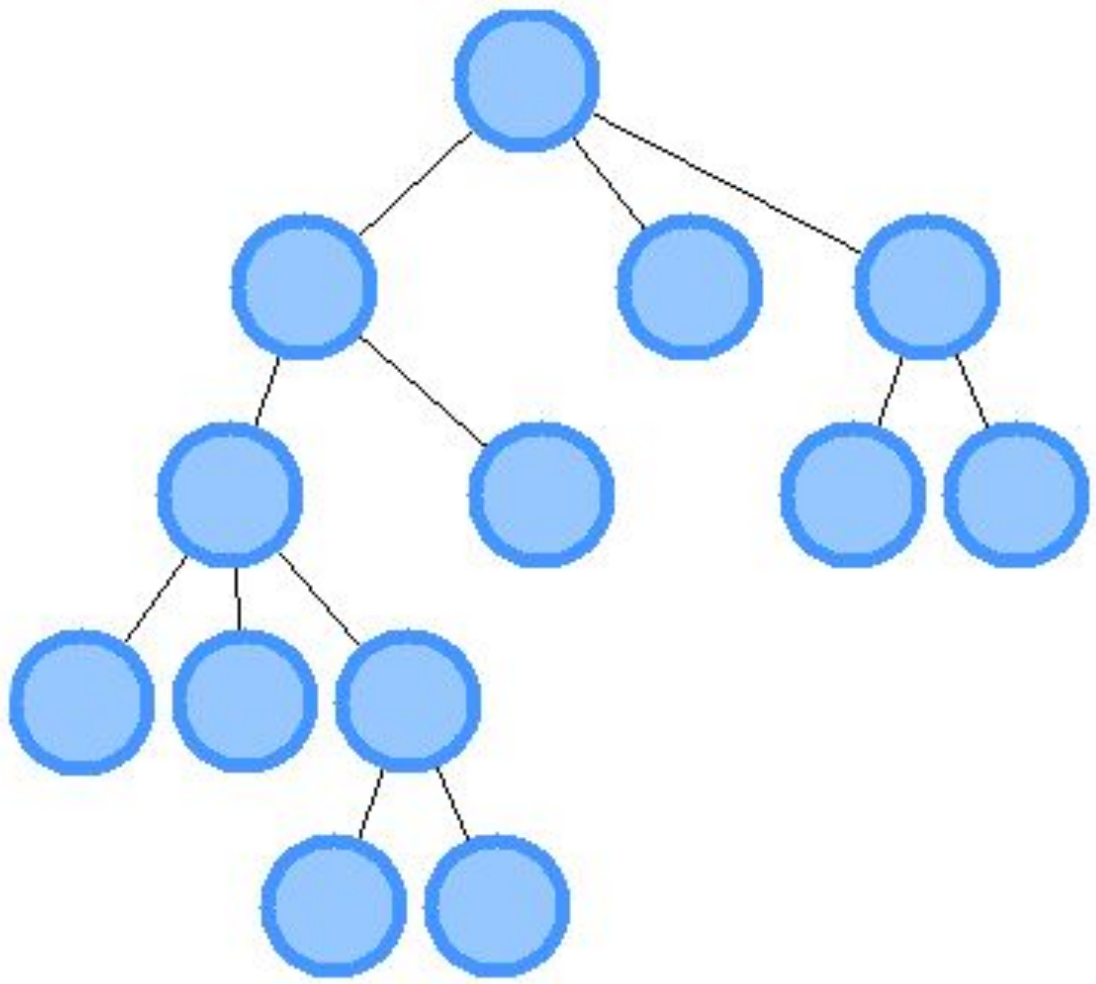
Определения

- **Поддерево** — часть древообразной структуры данных, которая может быть представлена в виде отдельного дерева. Любой узел дерева *T* вместе со всеми его узлами-потомками является



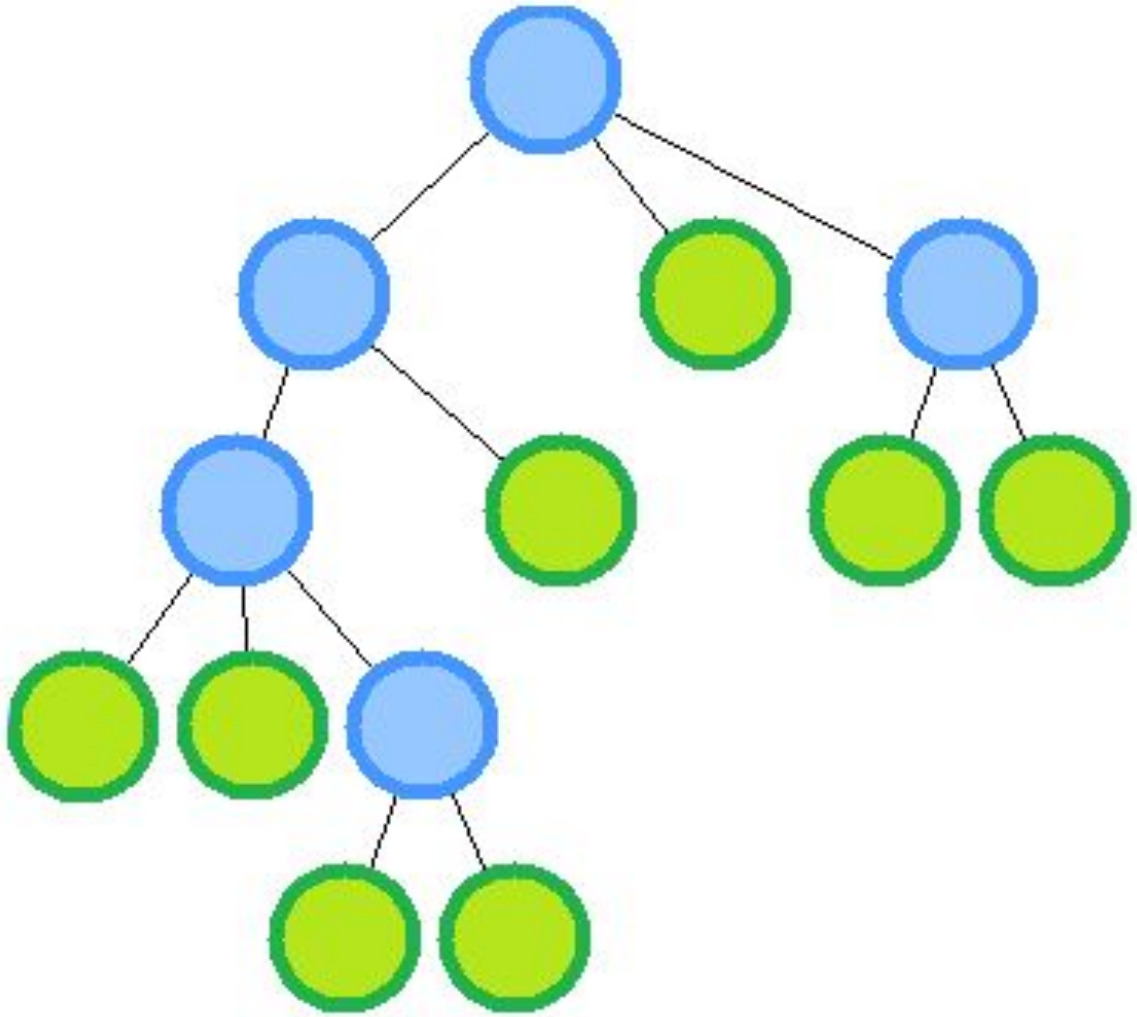
Определения

- **Лист**, листовой или терминальный узел — узел, не имеющий дочерних элементов;
- **Внутренний узел** — любой узел дерева, имеющий потомков, и таким образом, не являющийся листом;



Определения

- **Лист**, листовой или терминальный узел — узел, не имеющий дочерних элементов;
- **Внутренний узел** — любой узел дерева, имеющий потомков, и таким образом, не являющийся листом;

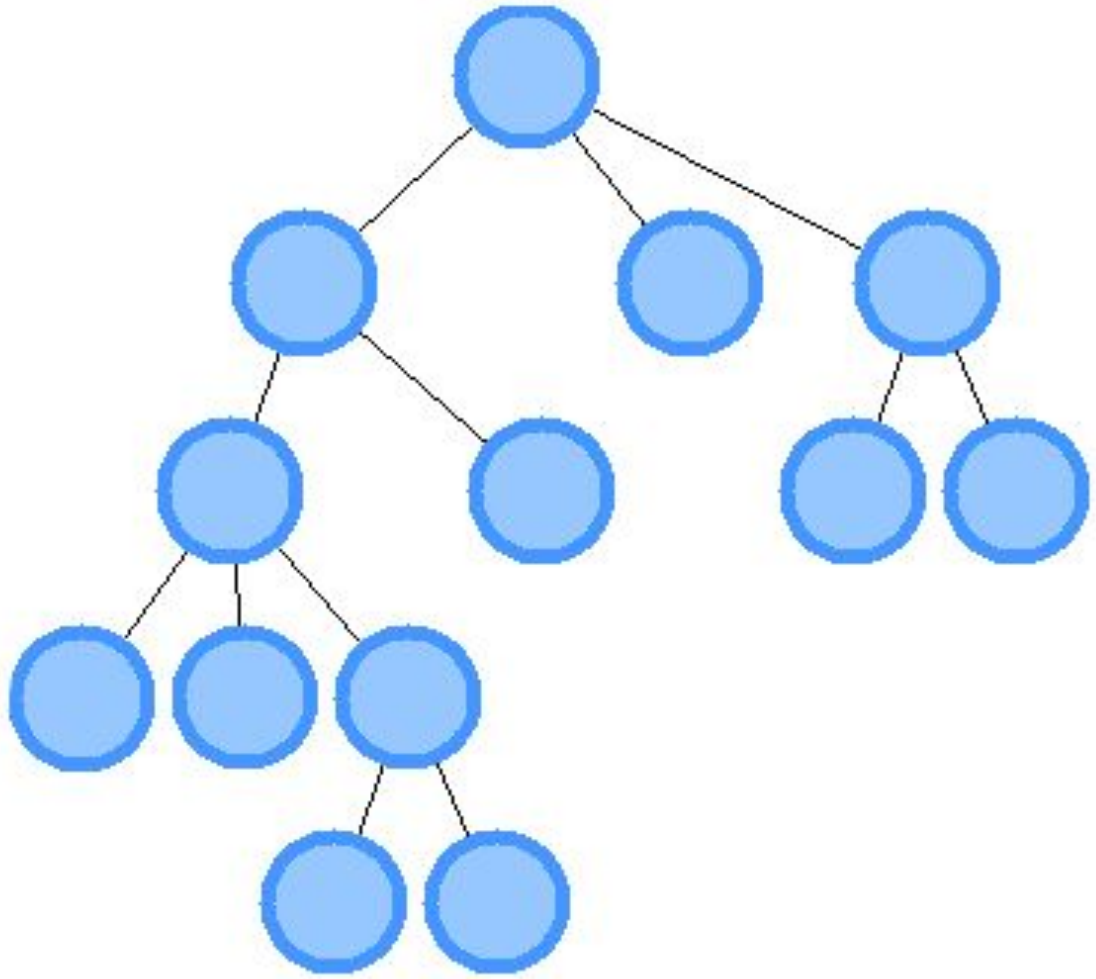


Определения

- **Глубина дерева** – длина самого длинного пути от корня до листа.

Каждое дерево обладает следующими свойствами:

- существует узел, в который не входит ни одной дуги (корень);
- в каждую вершину, кроме корня, входит одна дуга.

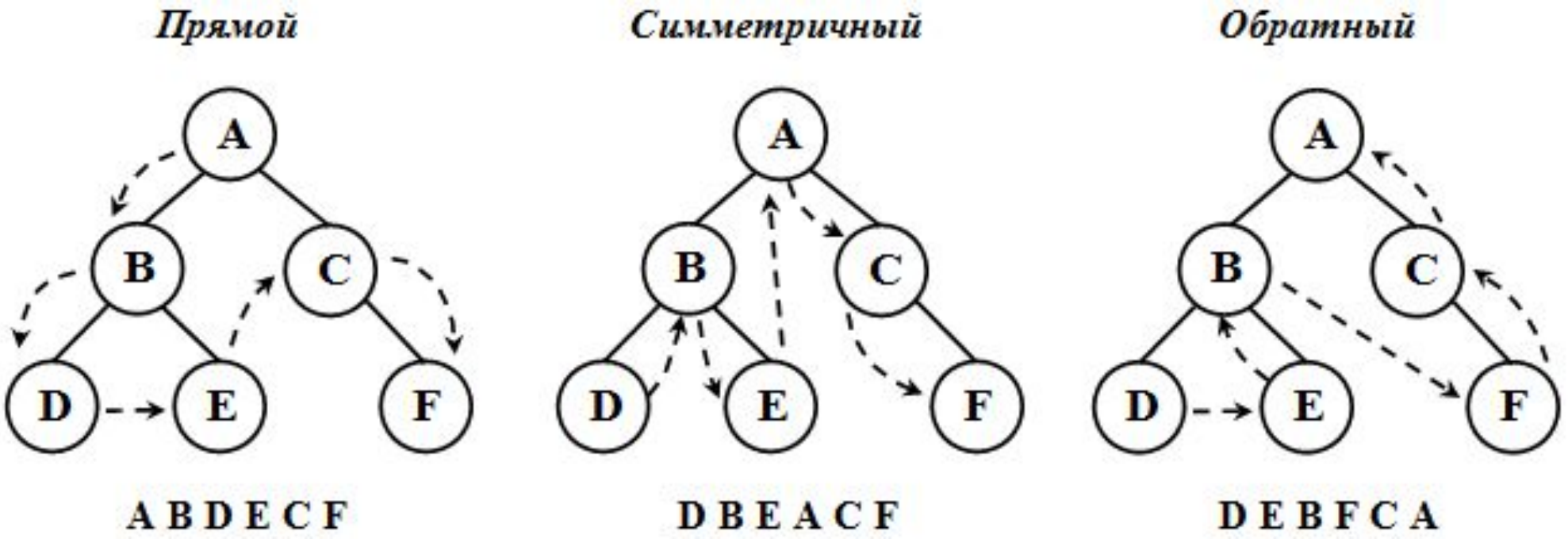


Определения

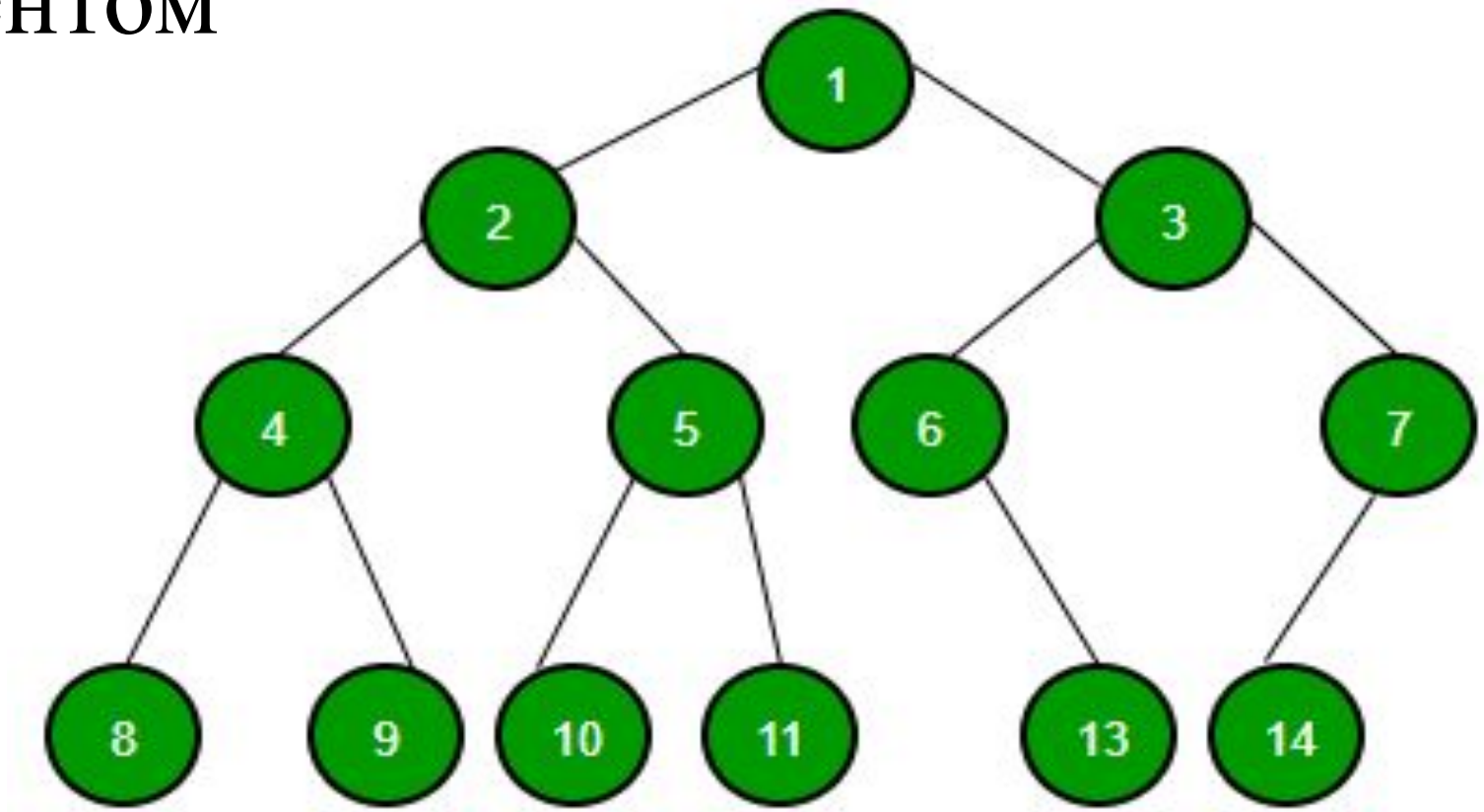
- **Обход дерева** – это упорядоченная последовательность вершин дерева, в которой каждая *вершина* встречается только один раз.
- При обходе все вершины дерева должны посещаться в определенном порядке. Существует несколько способов обхода всех вершин дерева.
- Три наиболее часто используемых способа *обхода дерева*:
  - прямой;
  - симметричный;
  - обратный.

Определения

- Три наиболее часто используемых способа *обхода дерева*:
  - прямой;
  - симметричный;
  - обратный

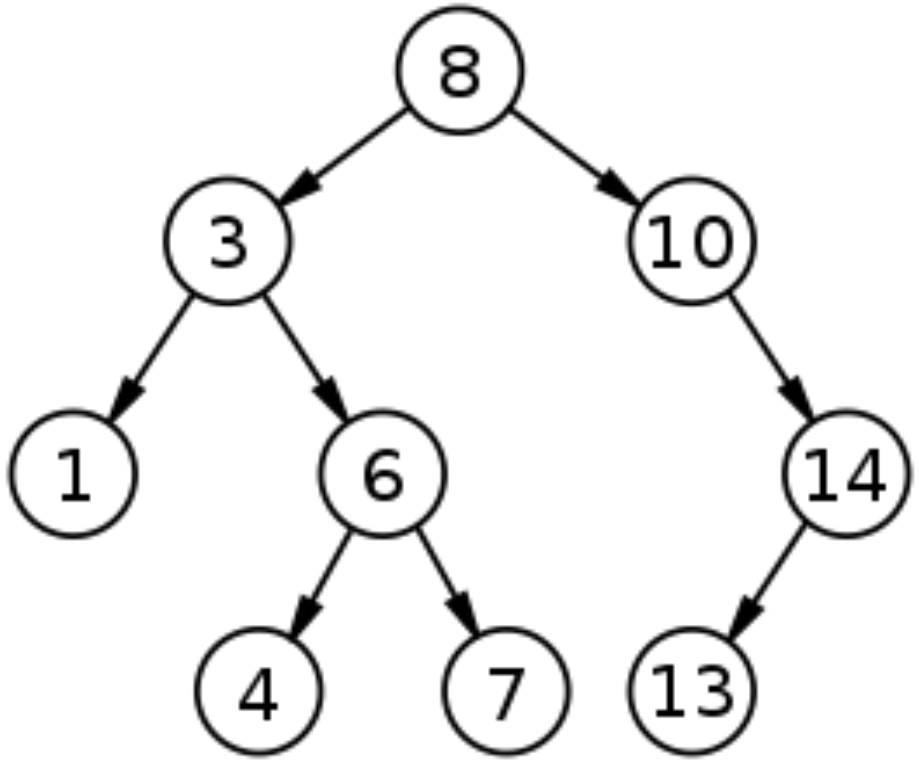


Бинарное дерево – это дерево, в котором каждый узел имеет не более двух дочерних элементов, которые называются левым дочерним элементом и правым дочерним элементом

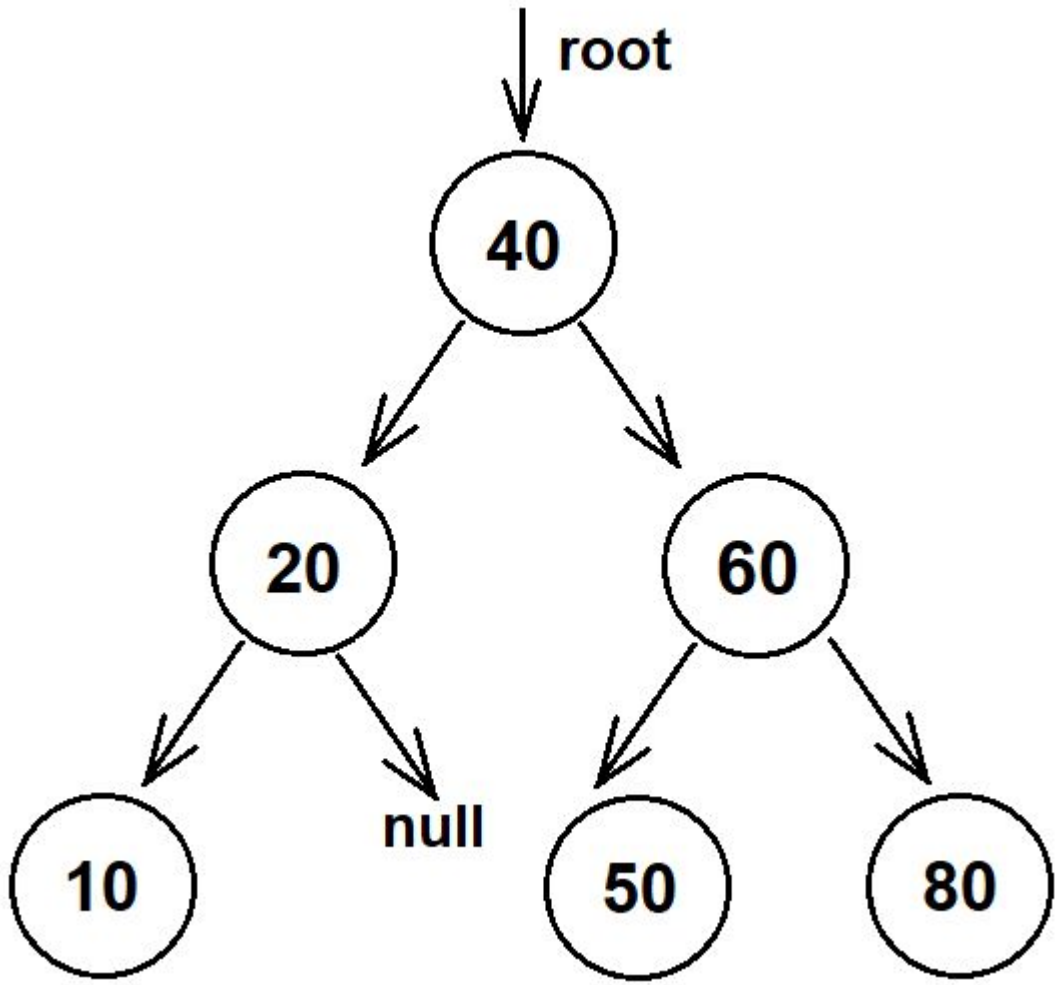


Бинарное дерево поиска – это бинарное дерево, в котором справедливо следующее:

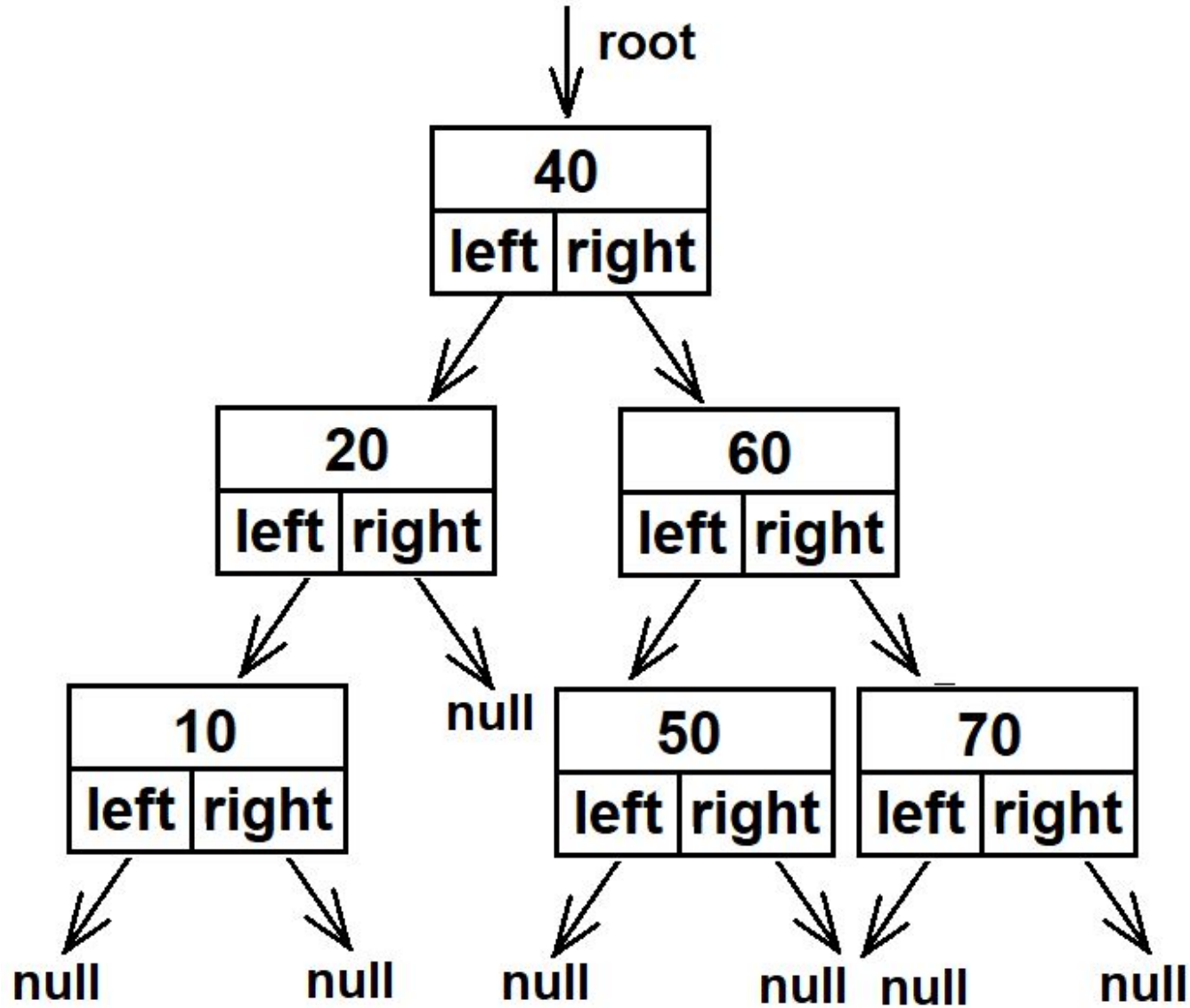
- Оба поддерева — левое и правое — являются двоичными деревьями поиска.
- У всех узлов *левого* поддерева произвольного узла X значения ключей данных *меньше*, нежели значение ключа данных самого узла X.
- У всех узлов *правого* поддерева произвольного узла X значения ключей данных *больше либо равны*, нежели значение ключа данных самого узла X.



Реализация бинарного дерева поиска



Реализация бинарного дерева поиска



Реализация бинарного дерева поиска

Двоичное дерево состоит из узлов (вершин) — записей вида (***data, left, right***), где *data* — некоторые данные, привязанные к узлу, *left* и *right* — ссылки на узлы, являющиеся детьми данного узла — левый и правый сыновья соответственно. Для оптимизации алгоритмов конкретные реализации предполагают также определения поля ***parent*** в каждом узле (кроме корневого) — ссылки на родительский элемент.

Реализация бинарного дерева поиска

- Данные (***data***) обладают ключом (***key***), на котором определена операция сравнения «меньше». В конкретных реализациях это может быть пара (***key***, ***value***) — (ключ и значение), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё.
- Для любого узла **X** выполняются свойства дерева поиска:

$$key[left[X]] < key[X] \leq key[right[X]],$$

то есть ключи данных родительского узла больше ключей данных левого сына и нестрого меньше ключей данных правого.

Класс узла дерева – **TreeNode**

```
class TreeNode<T> where T: IComparable
{
    T value;
    TreeNode<T> left;
    TreeNode<T> right;

    public TreeNode(T value) {
        this.value = value;
    }
    // добавить public свойства
}
```

Класс дерева – **MyTree**

- Добавление узла в дерево
- Поиск значения
- Удаление узла из дерева

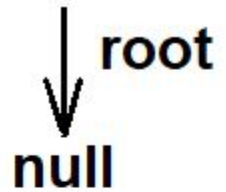
- Вычисление:
  - глубины дерева
  - количества листьев, узлов
- Обходы дерева

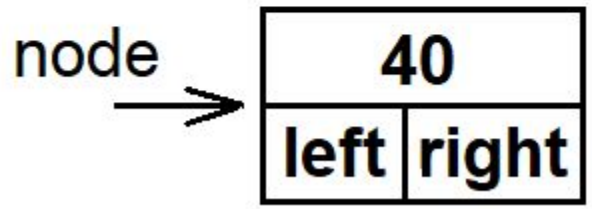
Класс дерева – **MyTree**

```
class MyTree<T> where T: IComparable
{
    TreeNode<T> root;

  public void Add(T value){…}
  public TreeNode<T> Find(T value){…}
  public void Remove(T value){…}
  public int GetDeep(){…}
  public int GetLeafs(){…}
  public int GetNodes(){…}
  public string LNR(){…}
    public string NLR(){…}
  public string ???(){…}
}
```

Добавление узла в дерево **void Add(T value)**

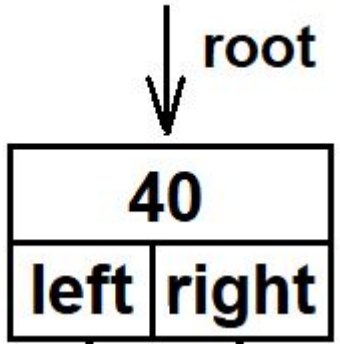
```
public void Add(T value)
{
    TreeNode<T> node = new TreeNode<T>(value);
    if (root == null) root = node;
}
```





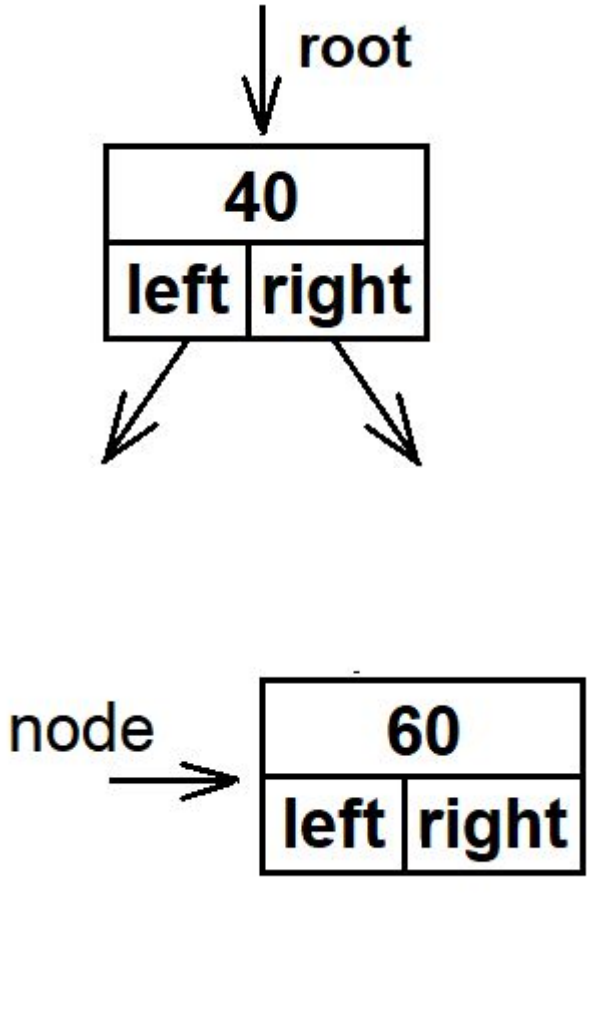
Добавление узла в дерево **void Add(T value)**

```
public void Add(T value)
{
    TreeNode<T> node = new TreeNode<T>(value);
    if (root == null) root = node;
    else ???
}
```



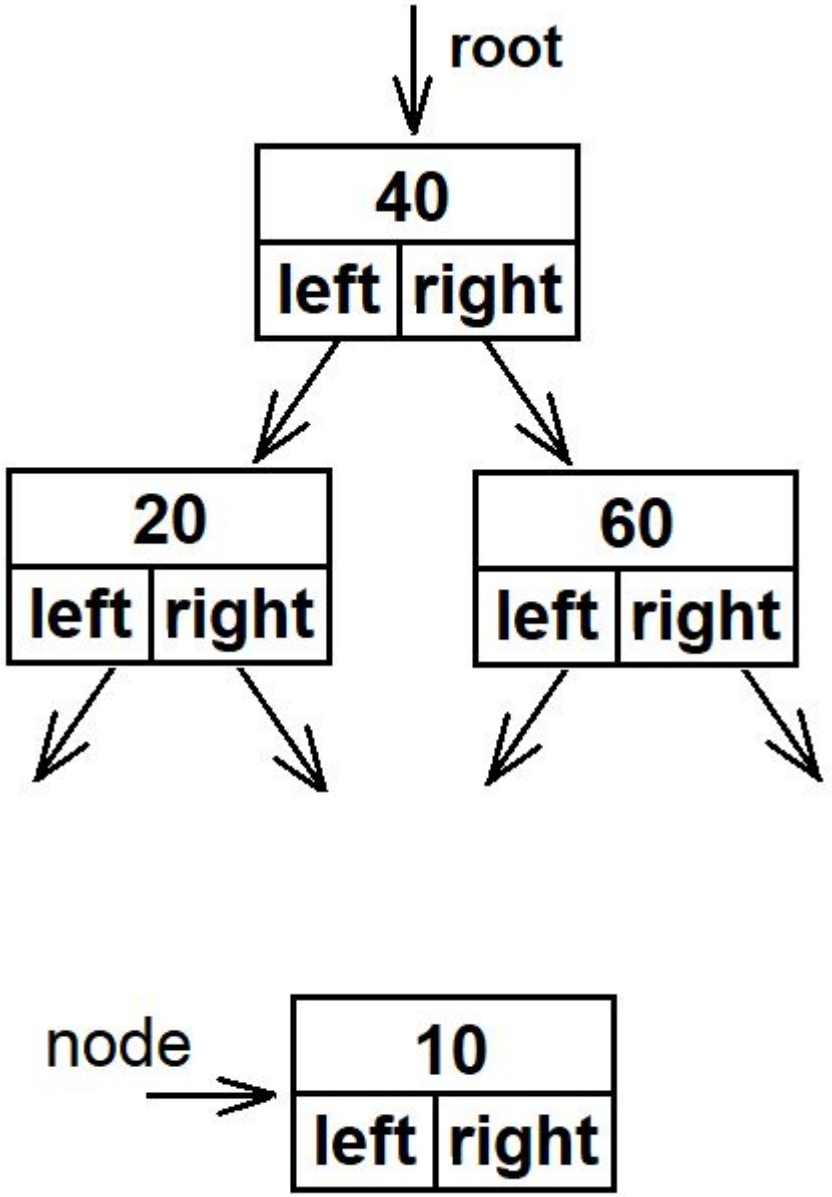
Добавление узла в дерево **void Add(T value)**

```
public void Add(T value)
{
    TreeNode<T> node = new TreeNode<T>(value);
    if (root == null) root = node;
    else
    {
      if (root.Value <= node.Value)
        {
           root.Right = node;
        }
        if (root.Value > node.Value)
        {
           root.Left = node;
        }
    }
}
```

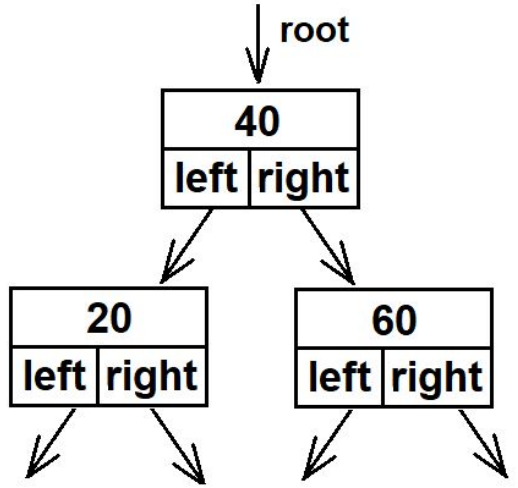


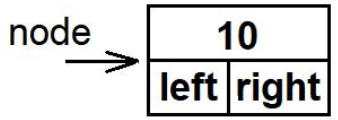
Добавление узла в дерево **void Add(T value)**

```
public void Add(T value)
{
    TreeNode<T> node = new TreeNode<T>(value);
    if (root == null) root = node;
    else
    {
     if (root.Value <= node.Value)
        {
           root.Right = node;
        }
        if (root.Value > node.Value)
        {
           root.Left = node;
        }
    }
}
```

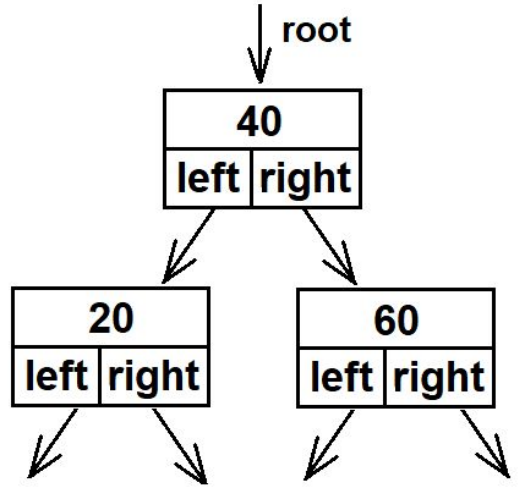


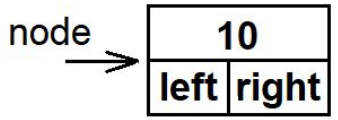
```
public void Add(T value)
{
    TreeNode<T> node = new TreeNode<T>(value);
    if (root == null) root = node;
    else Add(root, node);
}

private void Add(TreeNode<T> subroot, TreeNode<T> node )
{
    if (subroot.Value <= node.Value)
    {
        if (subroot.Right == null) subroot.Right = node;
        else Add(subroot.Right, node);
    }
    if (subroot.Value > node.Value)
    {
        if (subroot.Left == null) subroot.Left = node;
        else Add(subroot.Left, node);
    }
}
```





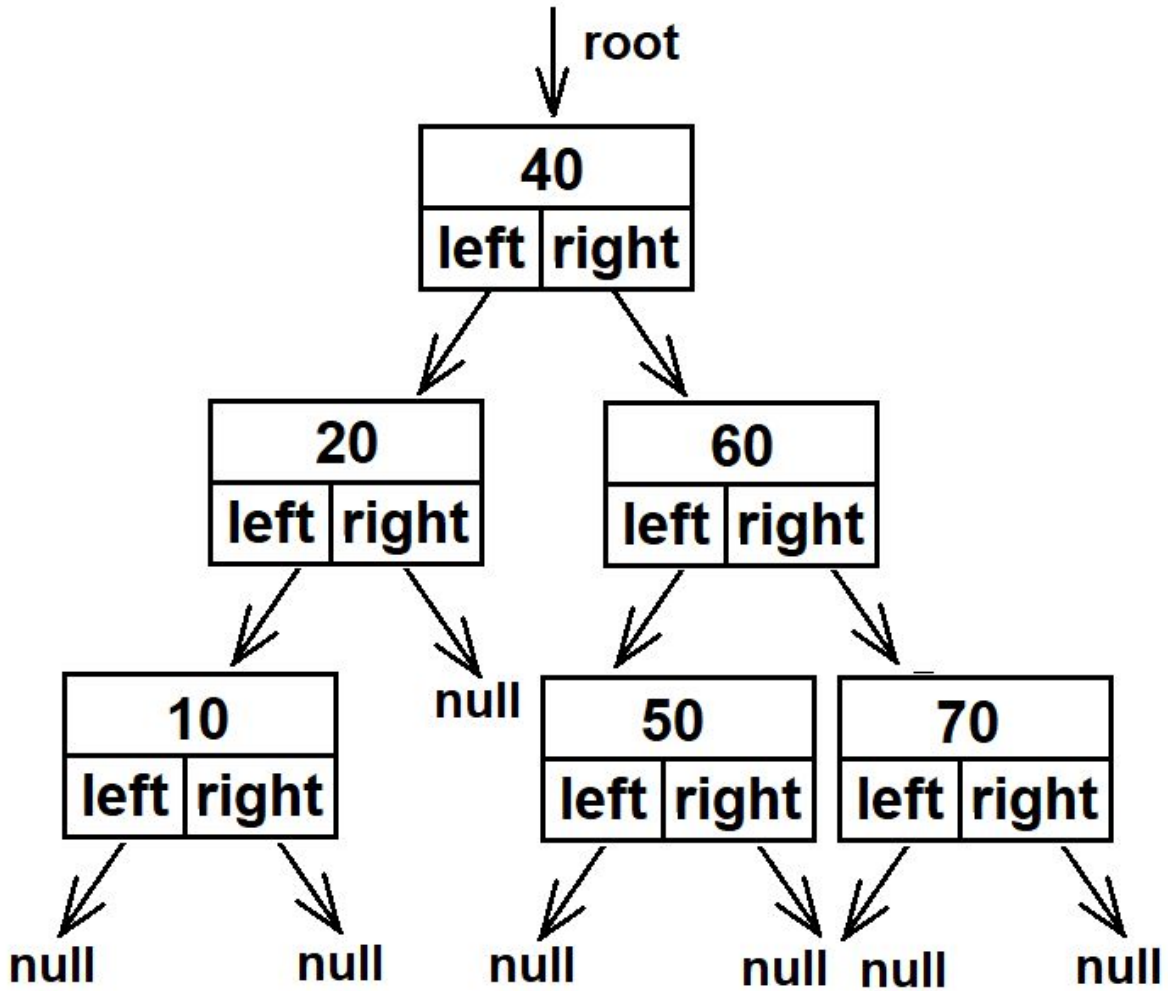
```
public void Add(T value)
{
    TreeNode<T> node = new TreeNode<T>(value);
    if (root == null) root = node;
    else Add(root, node);
}

private void Add(TreeNode<T> subroot, TreeNode<T> node )
{
    if (subroot.Value.CompareTo( node.Value ) <= 0)
    {
        if (subroot.Right == null) subroot.Right = node;
        else Add(subroot.Right, node);
    }
    if (subroot.Value.CompareTo( node.Value ) > 0)
    {
        if (subroot.Left == null) subroot.Left = node;
        else Add(subroot.Left, node);
    }
}
```





Поиск значения

```
public TreeNode<T> Find(T value)
```



Поиск значения
public TreeNode<T> Find(T value)

```
public TreeNode<T> Find(T value)
{
    if (root == null) return null;
    if (root.Value == value) return root;
    if (value < root.Value) return Find(value, root.Left );
    if (value > root.Value) return Find(value, root.Right);
}

private TreeNode<T> Find(T value, TreeNode<T> subroot)
{   if (subroot == null) return null;
    if (subroot.Value == value) return subroot;
    if (value < subroot.Value) return Find(value, subroot.Left);
    if (value > subroot.Value) return Find(value, subroot.Right);
}
```

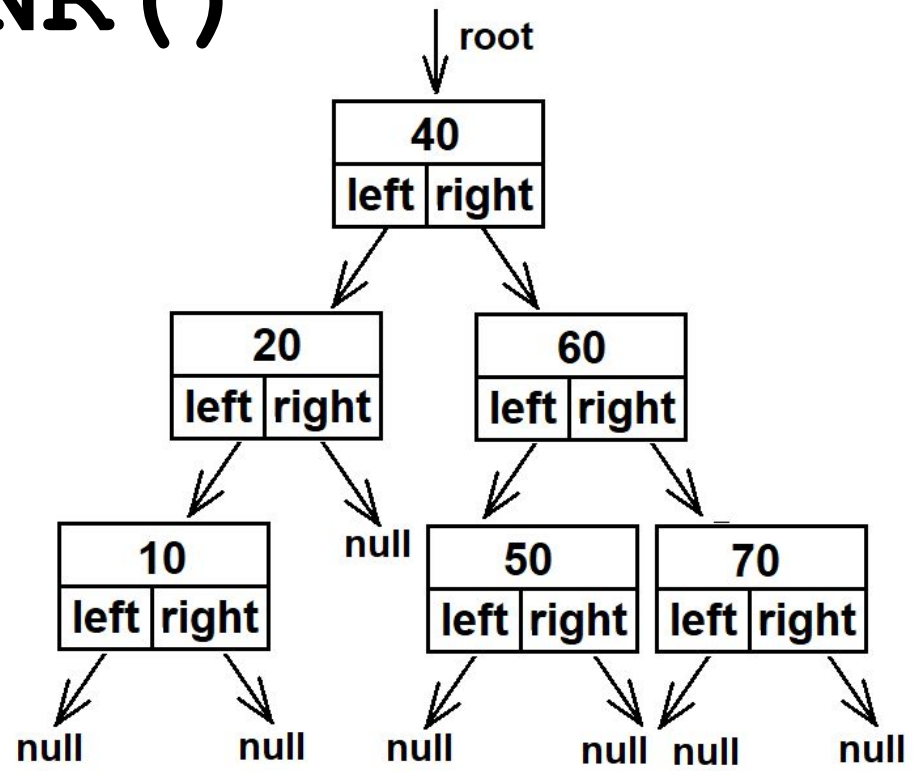
Поиск значения
public TreeNode<T> Find(T value)

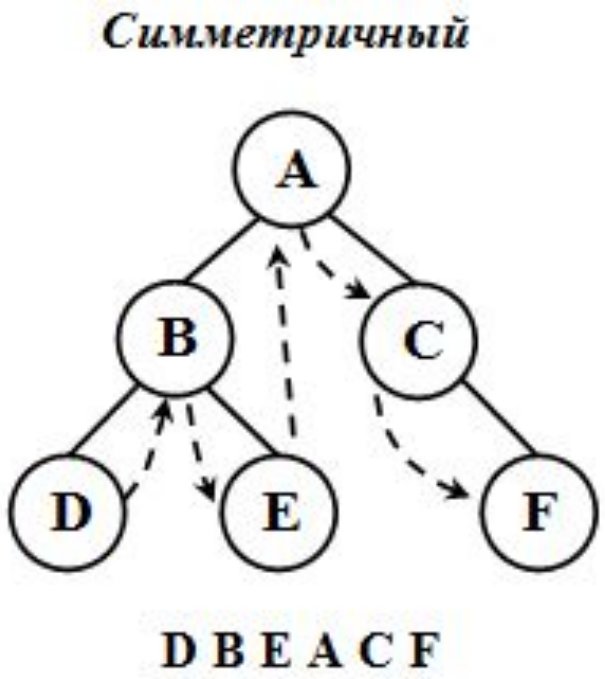
```
public TreeNode<T> Find(T value)
{
    return Find(value, root);
}

private TreeNode<T> Find(T value, TreeNode<T> subroot)
{   if (subroot == null) return null;
    if (value == subroot.Value) return subroot;
    if (value < subroot.Value) return Find(value, subroot.Left);
    if (value > subroot.Value) return Find(value, subroot.Right);
}
```

Поиск значения
public TreeNode<T> Find(T value)

```
public TreeNode<T> Find(T value)
{
    return Find(value, root);
}

private TreeNode<T> Find(T value, TreeNode<T> subroot)
{   if (subroot == null) return null;
    if (value.CompareTo(subroot.Value) == 0) return subroot;
    if (value.CompareTo(subroot.Value) < 0) return Find(value, subroot.Left);
    if (value.CompareTo(subroot.Value) > 0) return Find(value, subroot.Right);
}
```

Симметричный обход
`string LNR()`

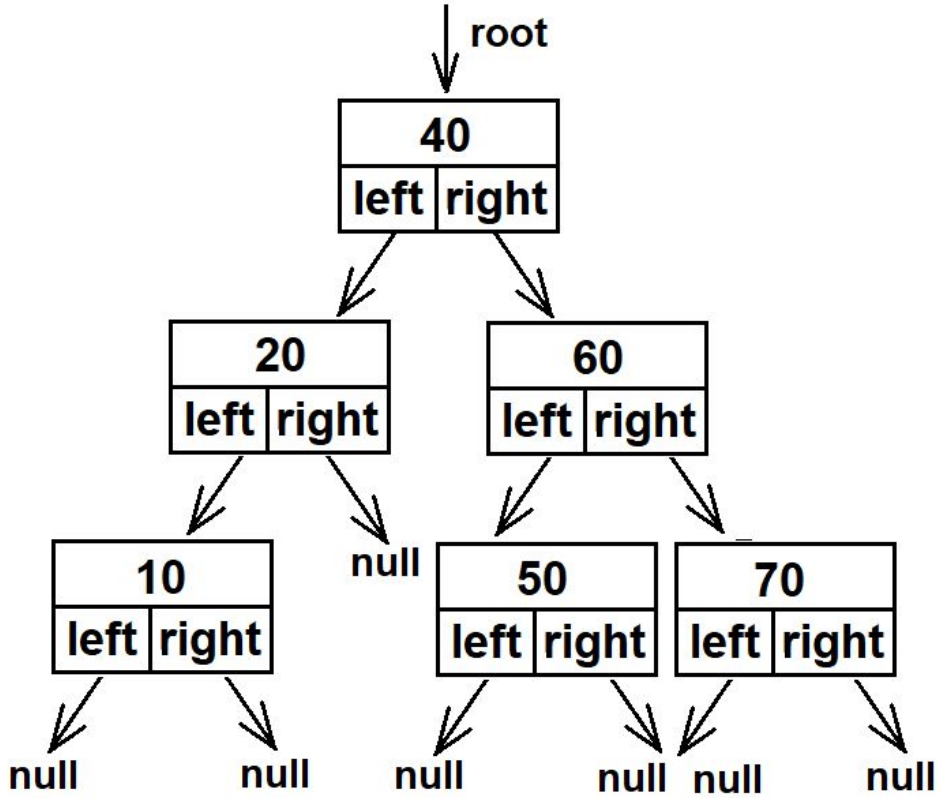




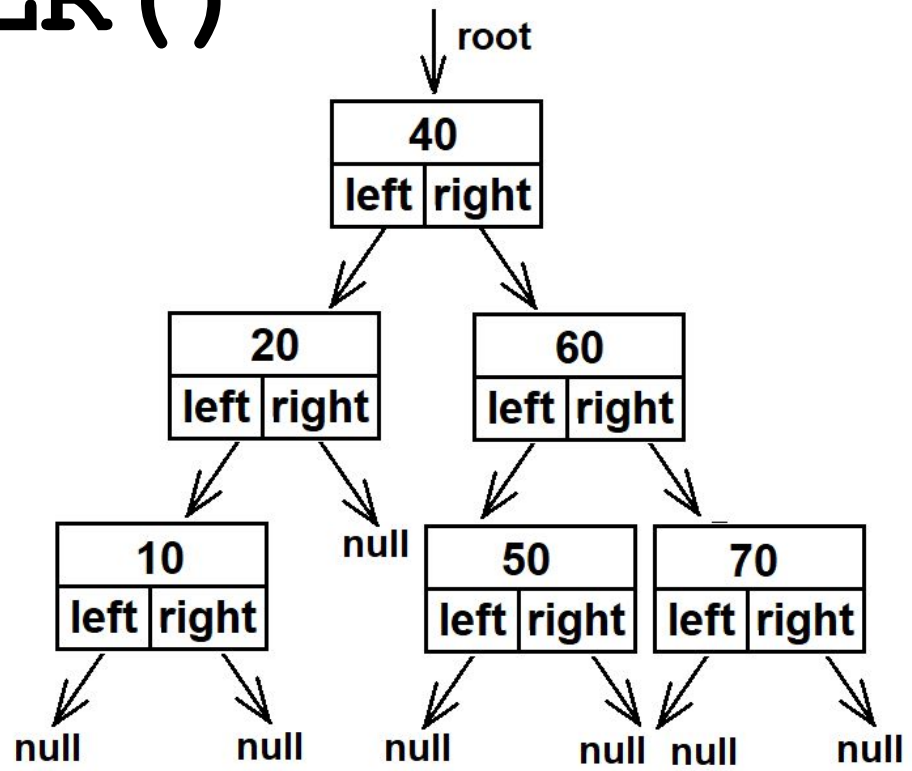
10 20 40 50 60 70

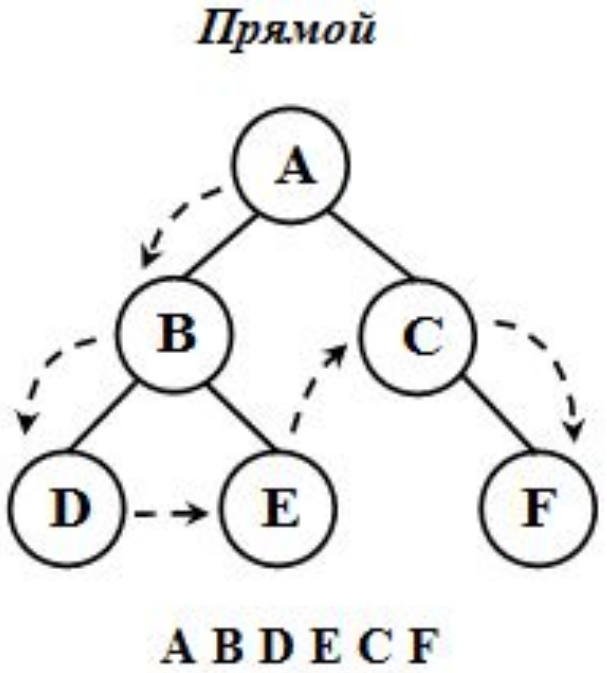
Симметричный обход **string LNR()**

```
public string LNR()
{
    return LNR(root);
}

private string LNR(TreeNode<T> subroot)
{   if (subroot == null) return "";
    return LNR(subroot.Left)
           + " "
           + subroot.Value.ToString()
           + " "
           + LNR(subroot.Right);
}
```



Прямой обход
`string NLR()`

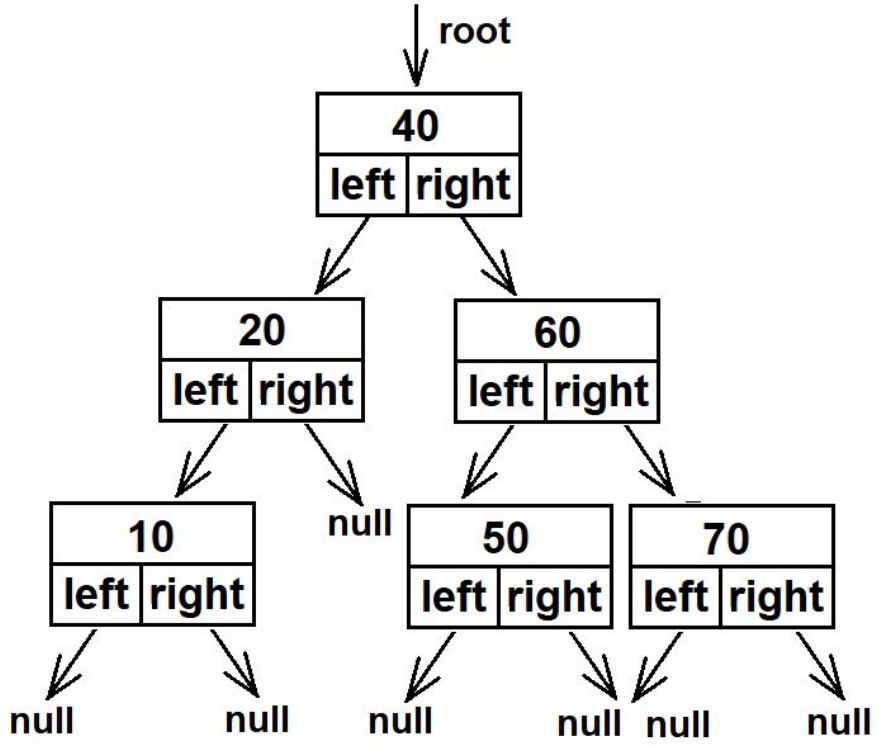




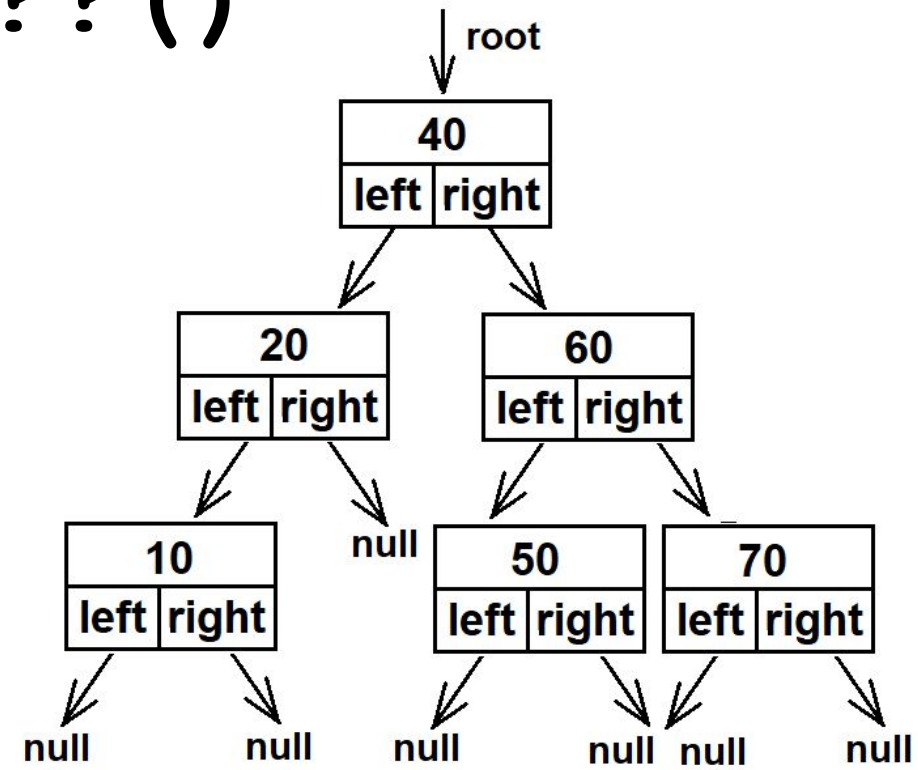
40 20 10 60 50 70

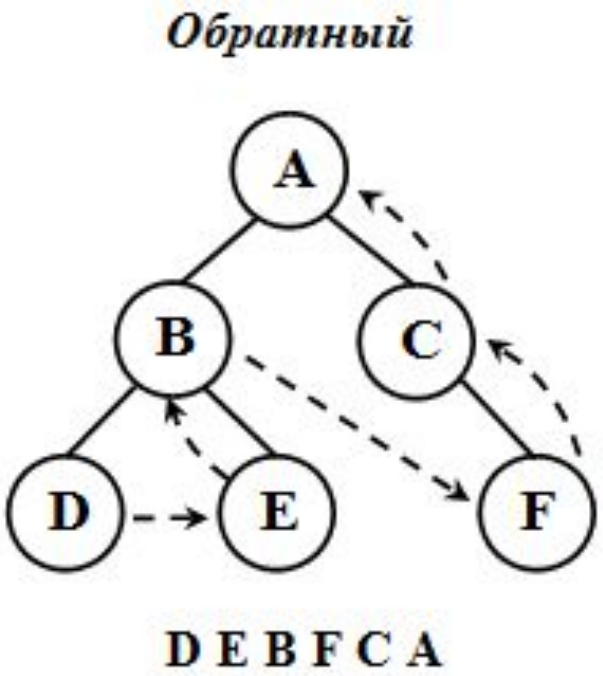
Прямой обход **string NLR()**

```
public string NLR()
{
    return NLR(root);
}

private string NLR(TreeNode<T> subroot)
{   if (subroot == null) return "";
    return subroot.Value.ToString()
           + " "
           + NLR(subroot.Left)
           + " "
           + NLR(subroot.Right);
}
```



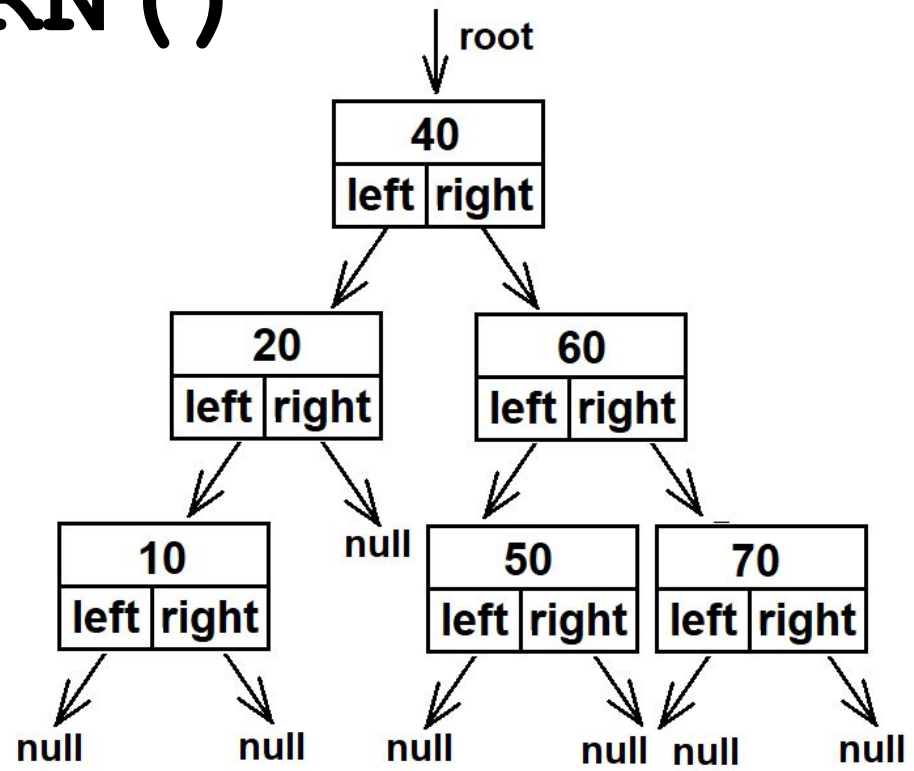
Обратный обход
`string ???()`

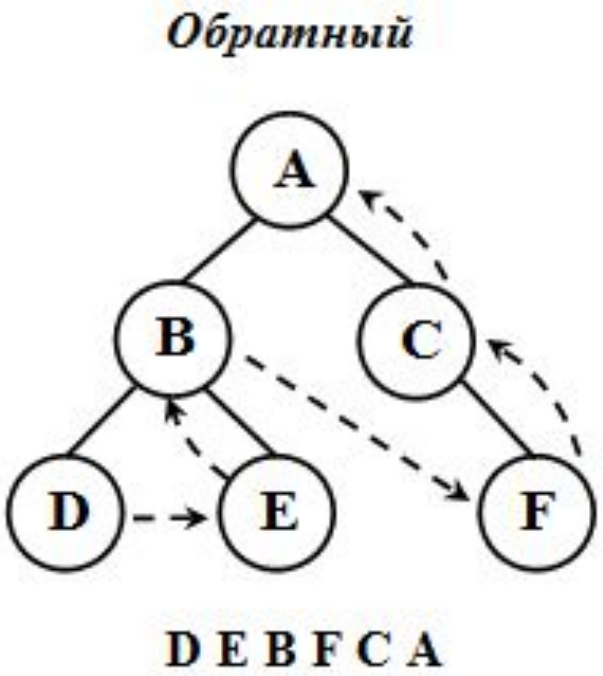




? ? ? ? ? ?

Обратный обход
`string LRN()`

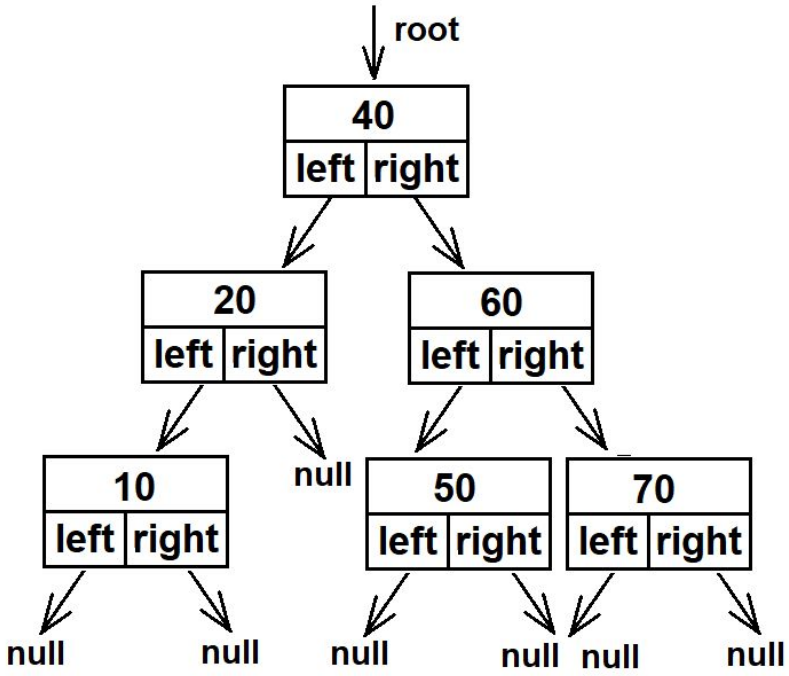




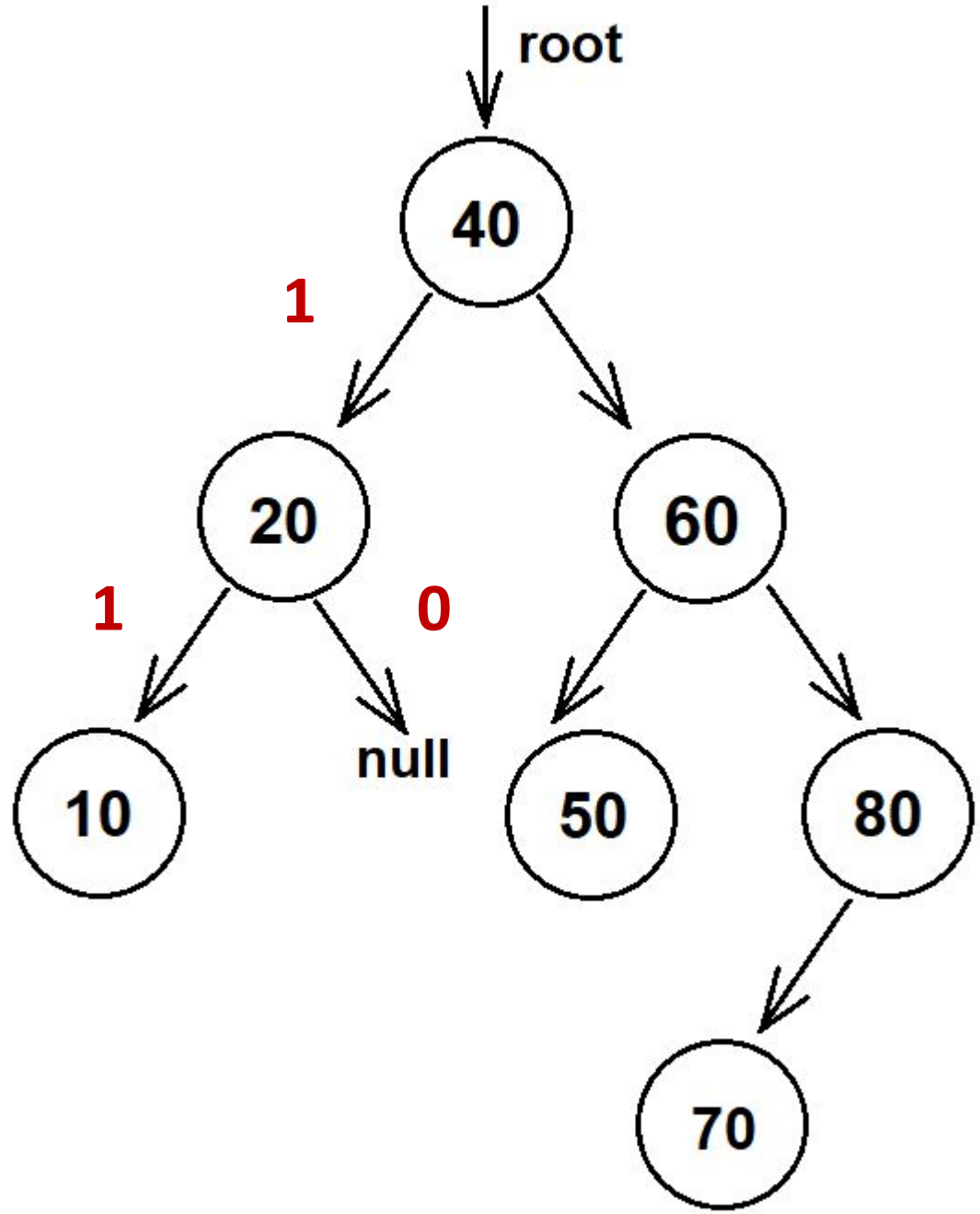
10 20 50 70 60 40

Обратный обход **string LRN()**

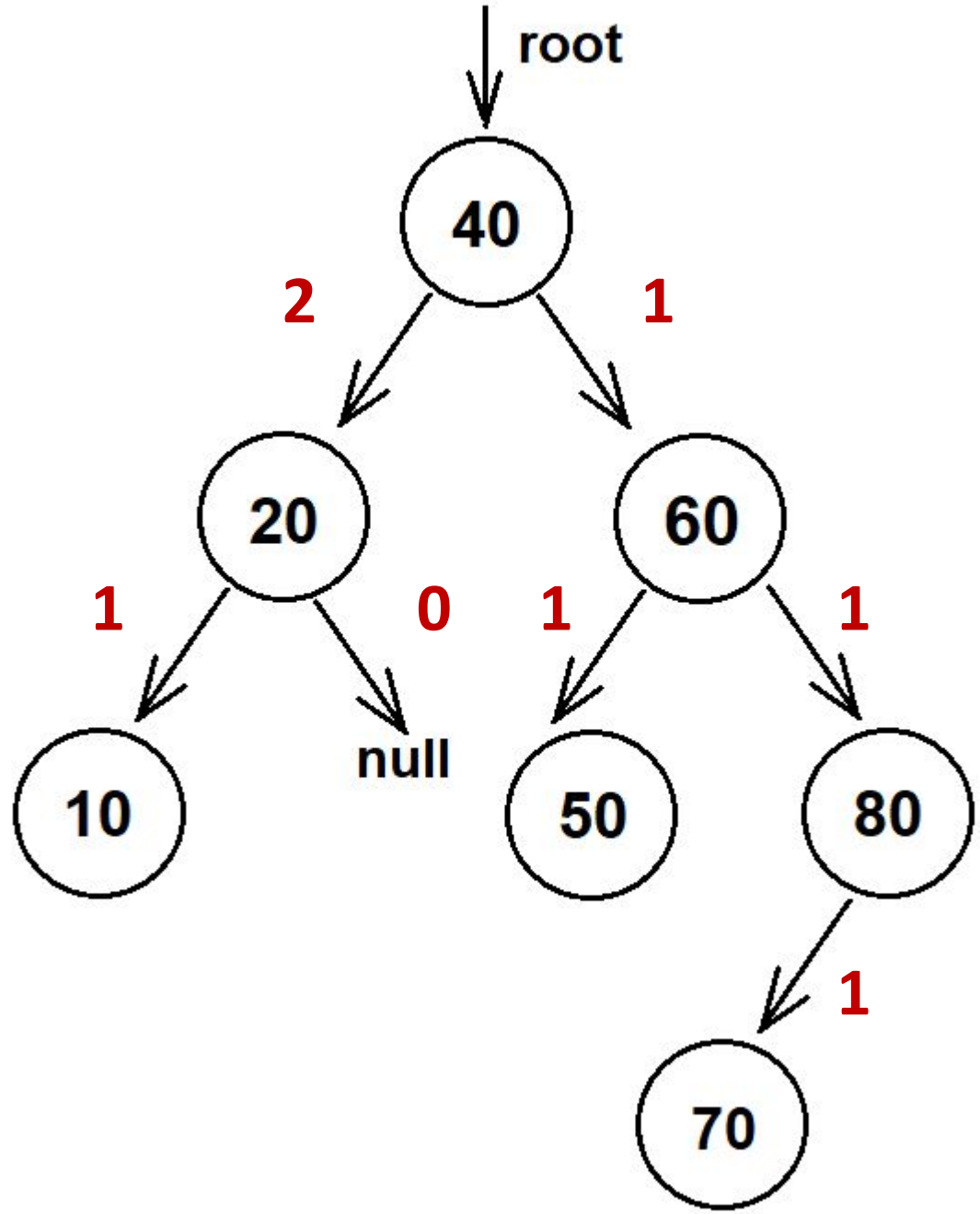
```
public string LRN()
{
    return LRN(root);
}

private string LRN(TreeNode<T> subroot)
{   if (subroot == null) return “”;
    return LRN(subroot.Left)
           + “ ”
           + LRN(subroot.Right)
       + “ ”
           + subroot.Value.ToString();
}
```



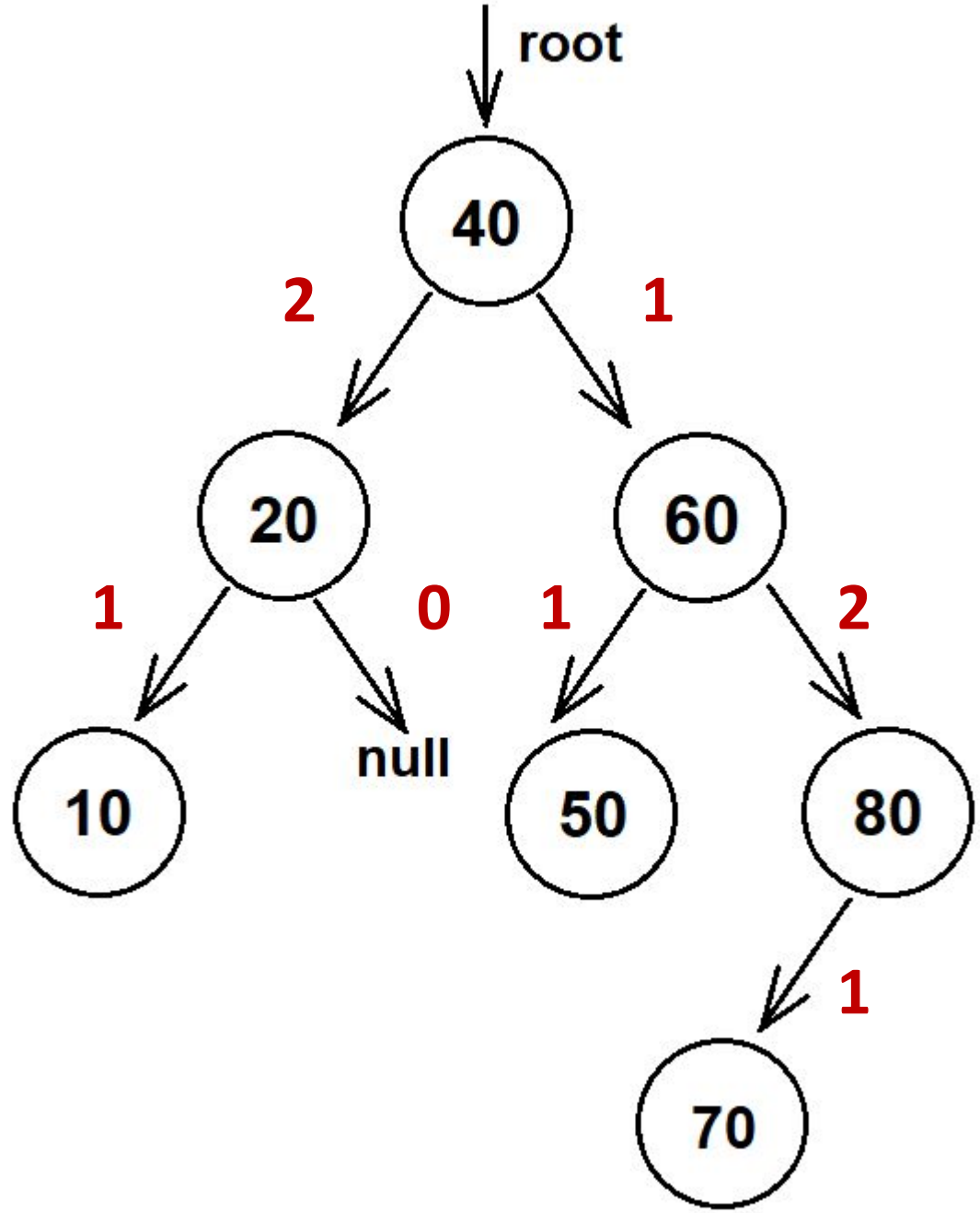
Вычисление глубины дерева `int GetDeep()`



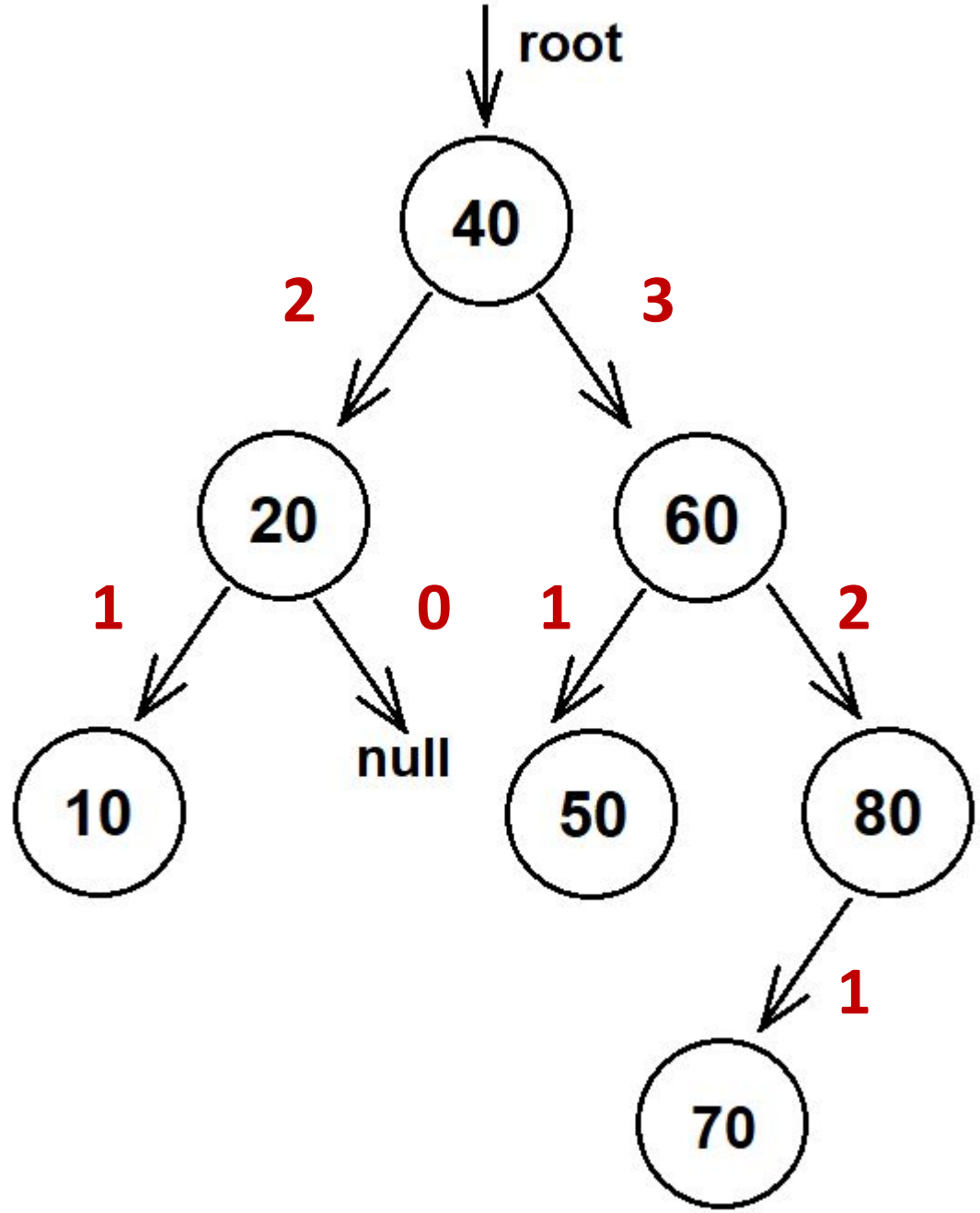
Вычисление глубины дерева `int GetDeep()`



Вычисление глубины дерева `int GetDeep()`

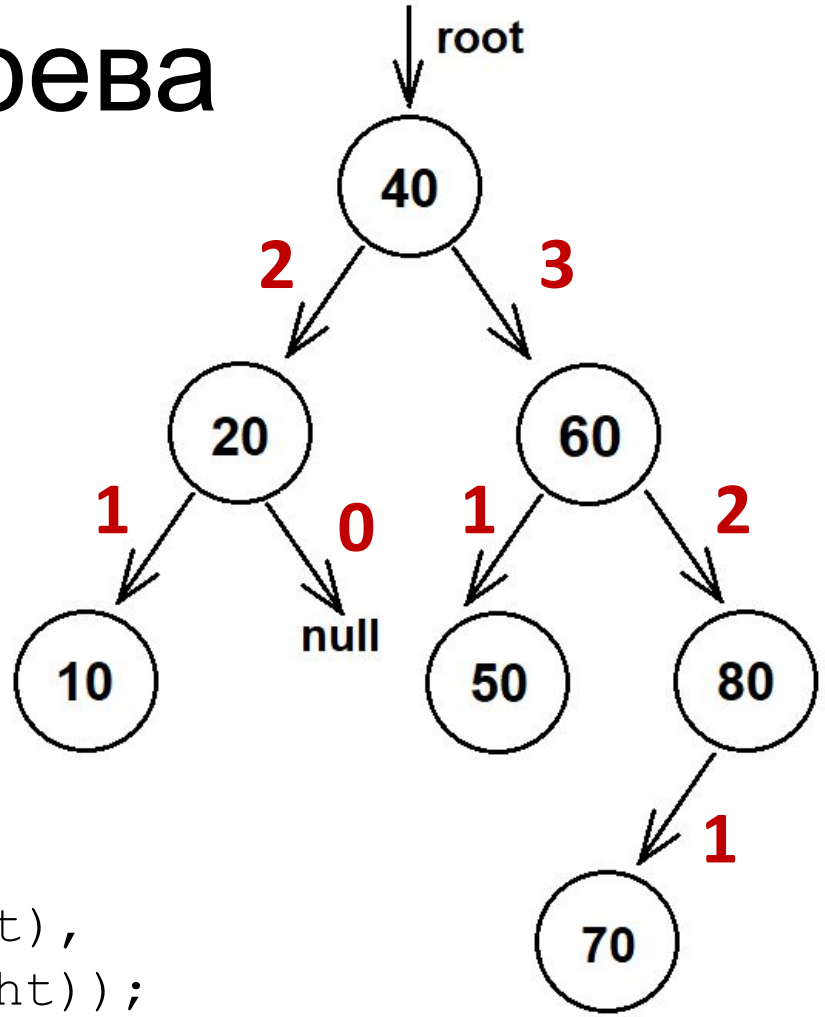


Вычисление глубины дерева `int GetDeep()`



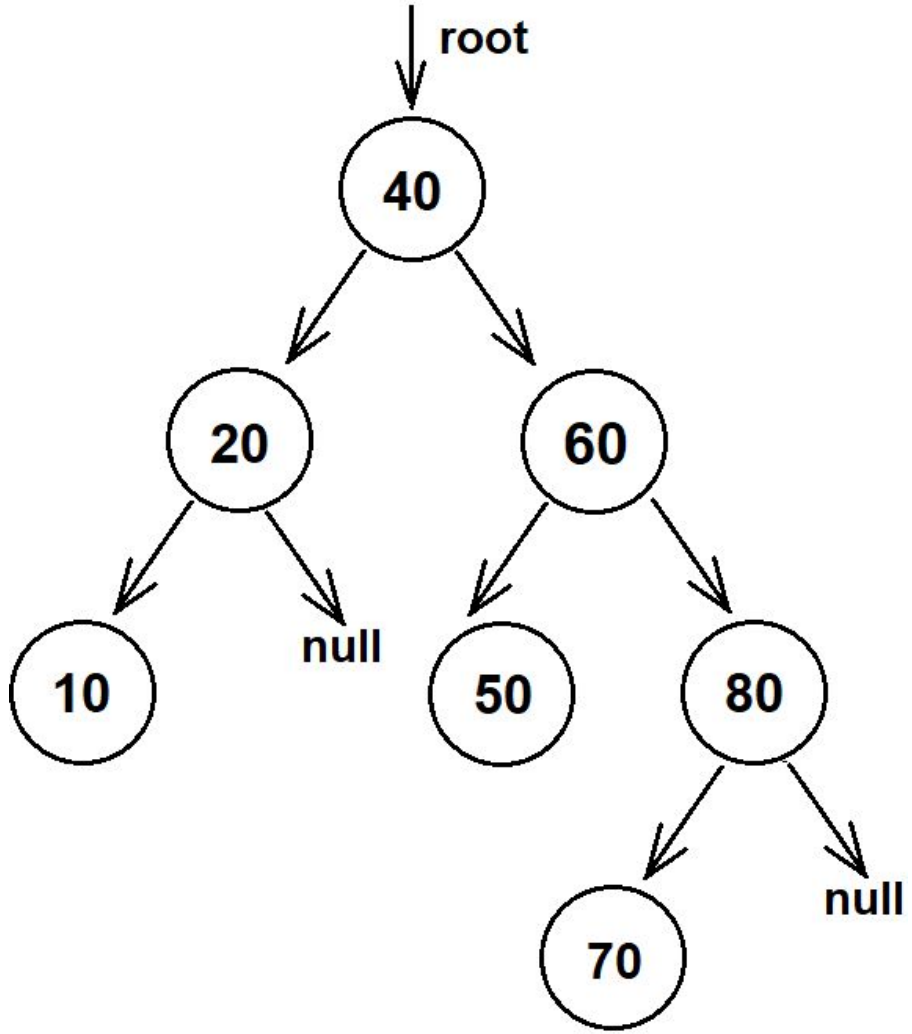
Вычисление глубины дерева
`int GetDeep()`

```
public int GetDeep()
{
    return GetDeep(root);
}

private int GetDeep(TreeNode<T> subroot)
{
    if (subroot == null) return 0;
    return 1 + Math.Max(GetDeep(subroot.Left),
                        GetDeep(subroot.Right));
}
```



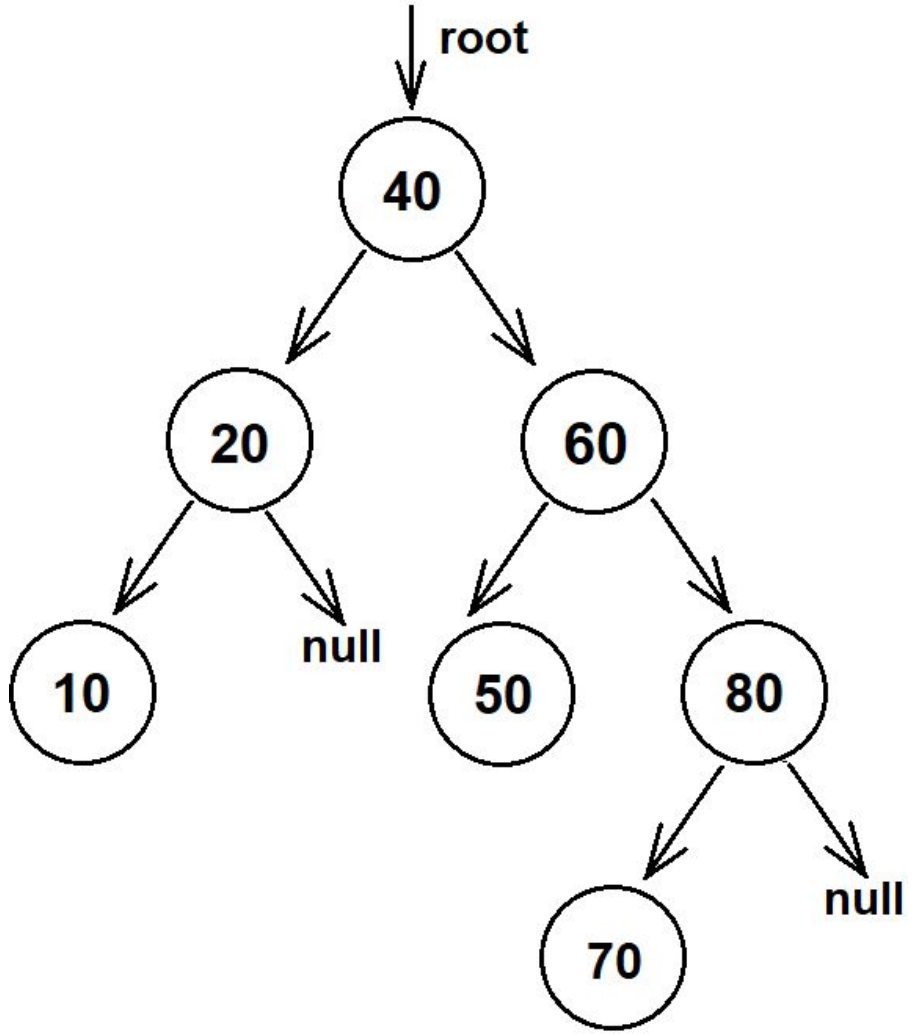
Вычисление количества листьев
`int GetLeafs()`

```
public int GetLeafs()
{
    return GetLeafs(root);
}

private int GetLeafs(TreeNode<T> subroot)
{
    if (subroot == null) return 0;
    if (subroot.Left == null &&
        subroot.Right == null) return 1;
    return GetLeafs(subroot.Left) +
           GetLeafs(subroot.Right);
}
```



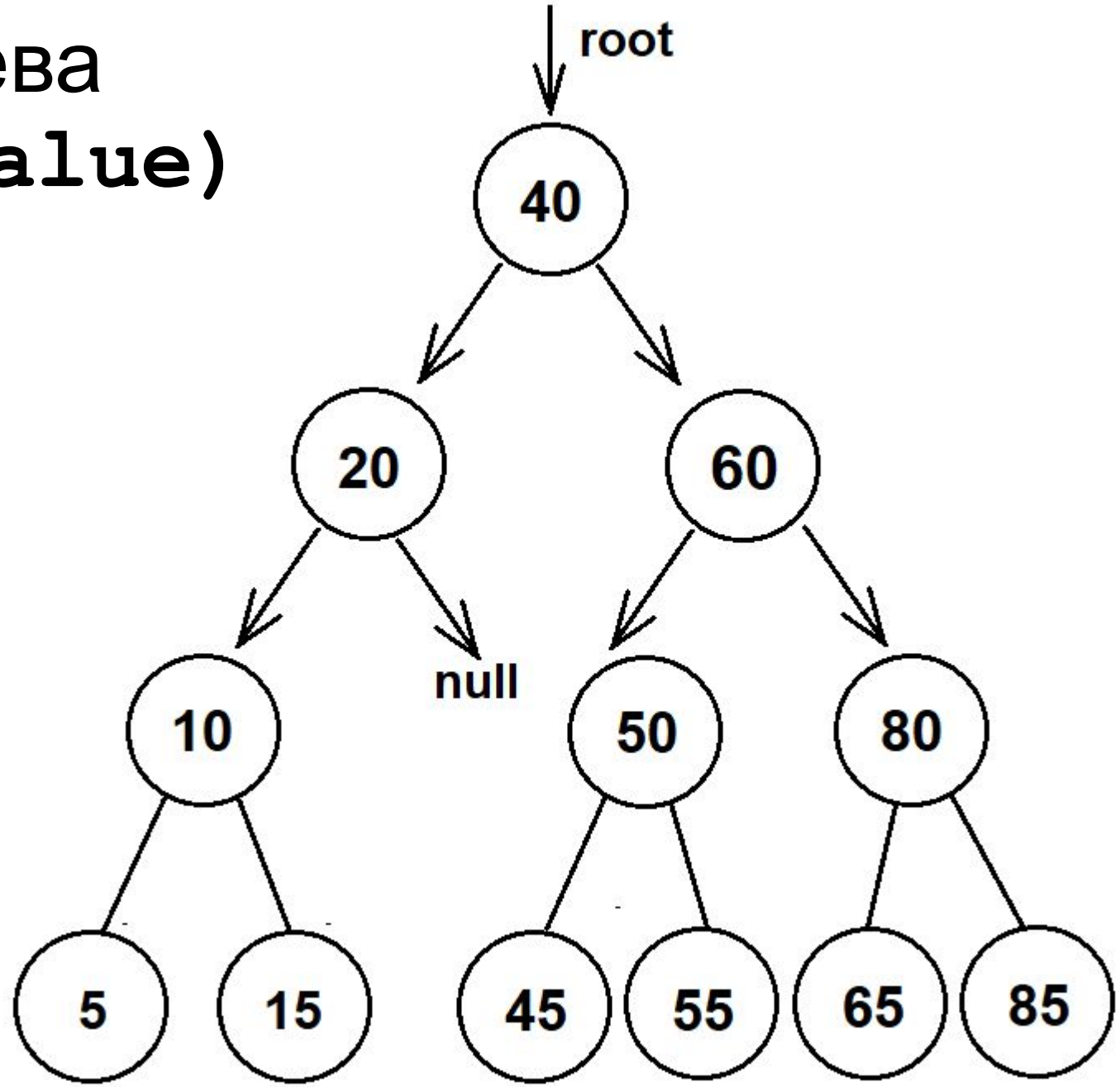
Вычисление количества узлов
`int GetNodes()`

```
public int GetNodes()
{
    return GetNodes(root);
}

private int GetNodes(TreeNode<T> subroot)
{
    if (subroot == null) return 0;
    return 1 +
            GetNodes(subroot.Left) +
        GetNodes(subroot.Right);
}
```



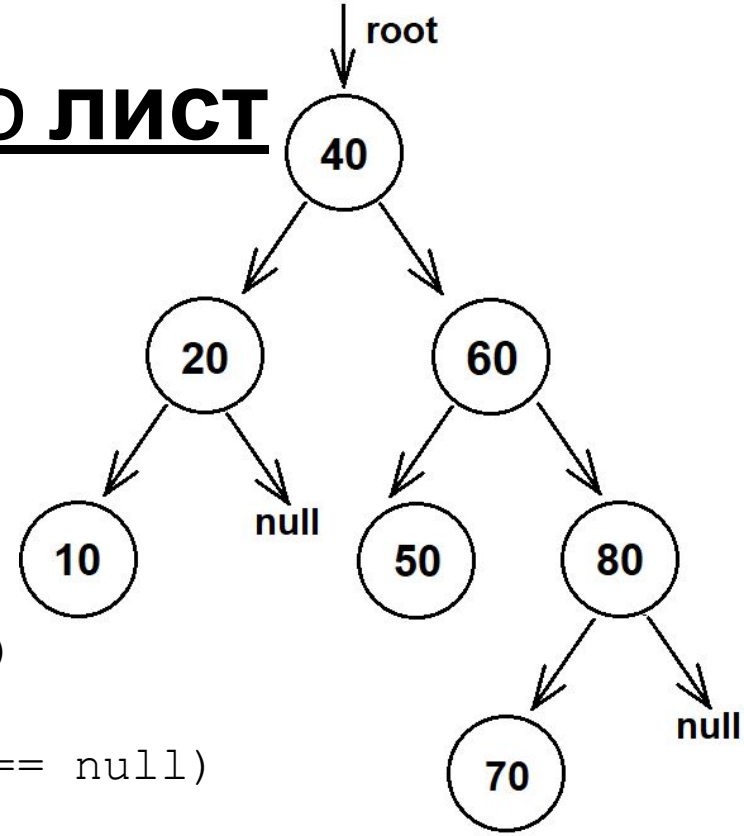
Удаление узла дерева
`void Remove(T value)`

- Если это **лист**
- Если у этого узла **одно** поддерево
- Если у этого узла **два** поддерева



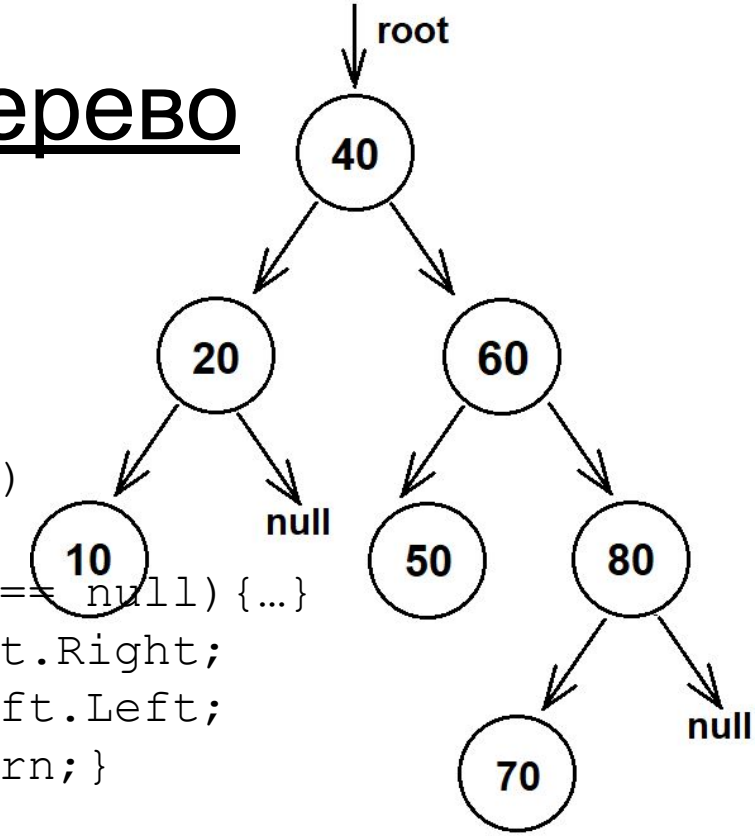
Удаление узла дерева

Если это **лист**




```
public void Remove(T value)
{
   return Remove(value, root);
}
private void Remove(T value, TreeNode<T> subroot)
{
   if (subroot == null) return;
   if (subroot.Left != null && subroot.Left.Value == value)
   {
      if (subroot.Left.Left == null && subroot.Left.Right == null)
      {   subroot.Left = null; return; }
   }
   if (subroot.Right != null && subroot.Right.Value == value)
   {
      if (subroot.Right.Left == null && subroot.Right.Right == null)
      {   subroot.Right = null; return; }
   }
   if (subroot.Value < value) Remove(value, subroot.Right);
   if (subroot.Value > value) Remove(value, subroot.Left);
}
```

Удаление узла Если **одно** поддерево

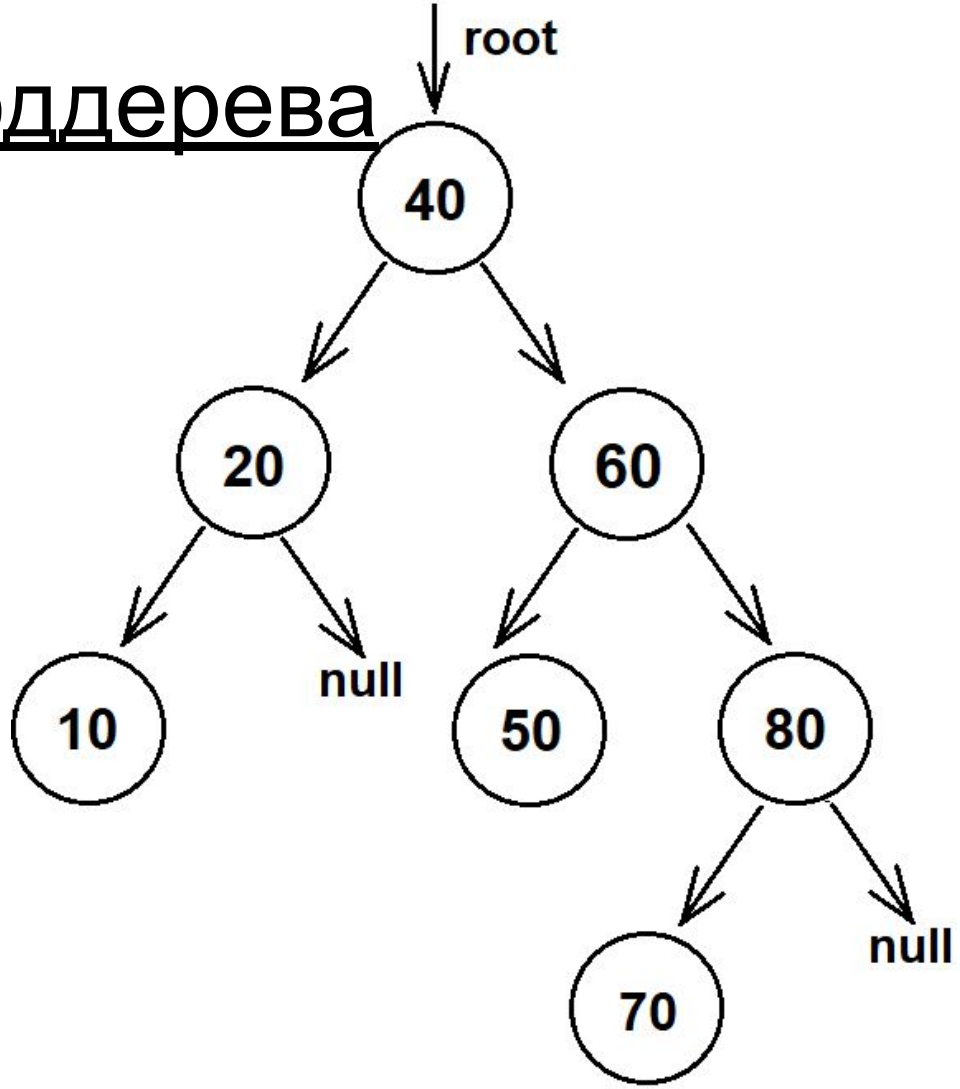



```
private void Remove(T value, TreeNode<T> subroot)
{
   if (subroot == null) return;
   TreeNode<T> subtree = null;
   if (subroot.Left != null && subroot.Left.Value == value)
   {
      if (subroot.Left.Left == null && subroot.Left.Right == null){…}
      if (subroot.Left.Left == null) subtree = subroot.Left.Right;
      if (subroot.Left.Right == null) subtree = subroot.Left.Left;
      if (subtree != null) { subroot.Left = subtree;  return;}
   }
   if (subroot.Right != null && subroot.Right.Value == value)
   {
      if (subroot.Right.Left == null && subroot.Right.Right == null){…}
      if (subroot.Right.Left == null) subtree = subroot.Right.Right;
      if (subroot.Right.Right == null) subtree = subroot.Right.Left;
      if (subtree != null) { subroot.Right = subtree;  return;}
   }
   if (subroot.Value < value) Remove(value, subroot.Right);
   if (subroot.Value > value) Remove(value, subroot.Left);
}
```

Удаление узла

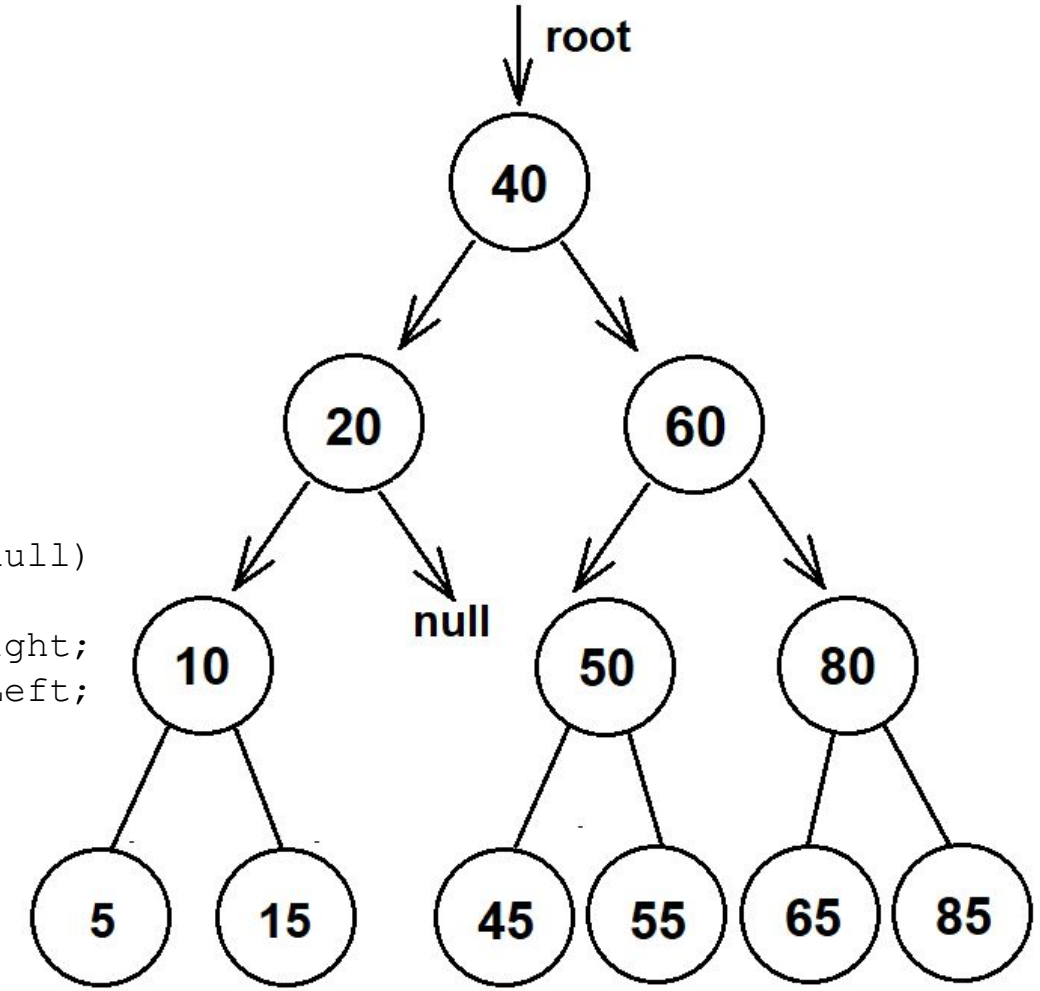
Если **два** поддерева

```
private void Remove(T value, TreeNode<T> subroot)
{
   if (subroot == null) return;  TreeNode<T> subtree = null;
   if (subroot.Left != null && subroot.Left.Value == value)
   {
      … //если лист или одно поддерево
        subtree = subroot.Left.Left;
        subroot.Left = subroot.Left.Right;
        TreeNode<T> min = subroot.Left;
        while (min.Left!=null) min = min.Left;
        min.Left = subtree;  return;
    }
   if (subroot.Right != null && subroot.Right.Value == value)
   {
      … //если лист или одно поддерево
        subtree = subroot.Right.Left;
        subroot.Right = subroot.Right.Right;
        TreeNode<T> min = subroot.Right;
        while (min.Left!=null) min = min.Left;
        min.Left = subtree;  return;
    }
   if (subroot.Value < value) Remove(value, subroot.Right);
   if (subroot.Value > value) Remove(value, subroot.Left);
}
```



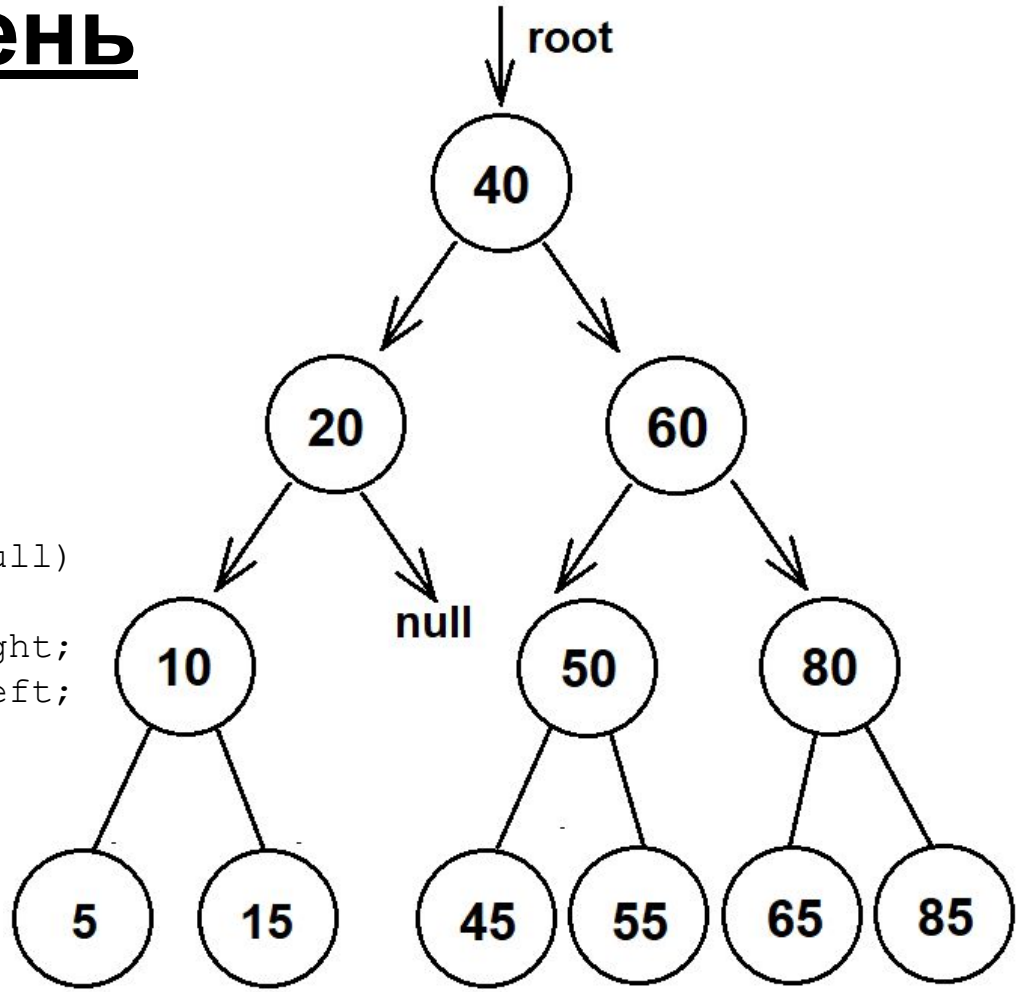
Удаление узла

```
public void Remove(T value)
{
   return Remove(value, root);
}
private void Remove(T value, TreeNode<T> subroot)
{
   if (subroot == null) return;  TreeNode<T> subtree = null;
   if (subroot.Left != null && subroot.Left.Value == value)
   {
      if (subroot.Left.Left == null && subroot.Left.Right == null)
      {   subroot.Left = null; return; }
      if (subroot.Left.Left == null) subtree = subroot.Left.Right;
      if (subroot.Left.Right == null) subtree = subroot.Left.Left;
      if (subtree != null) { subroot.Left = subtree;  return;}
      subtree = subroot.Left.Left;
      subroot.Left = subroot.Left.Right;
      TreeNode<T> min = subroot.Left;
      while (min.Left!=null) min = min.Left;
      min.Left = subtree;  return;
   }
   if (subroot.Right != null && subroot.Right.Value == value)
   {
      …
   }
   if (subroot.Value < value) Remove(value, subroot.Right);
   if (subroot.Value > value) Remove(value, subroot.Left);
}
```

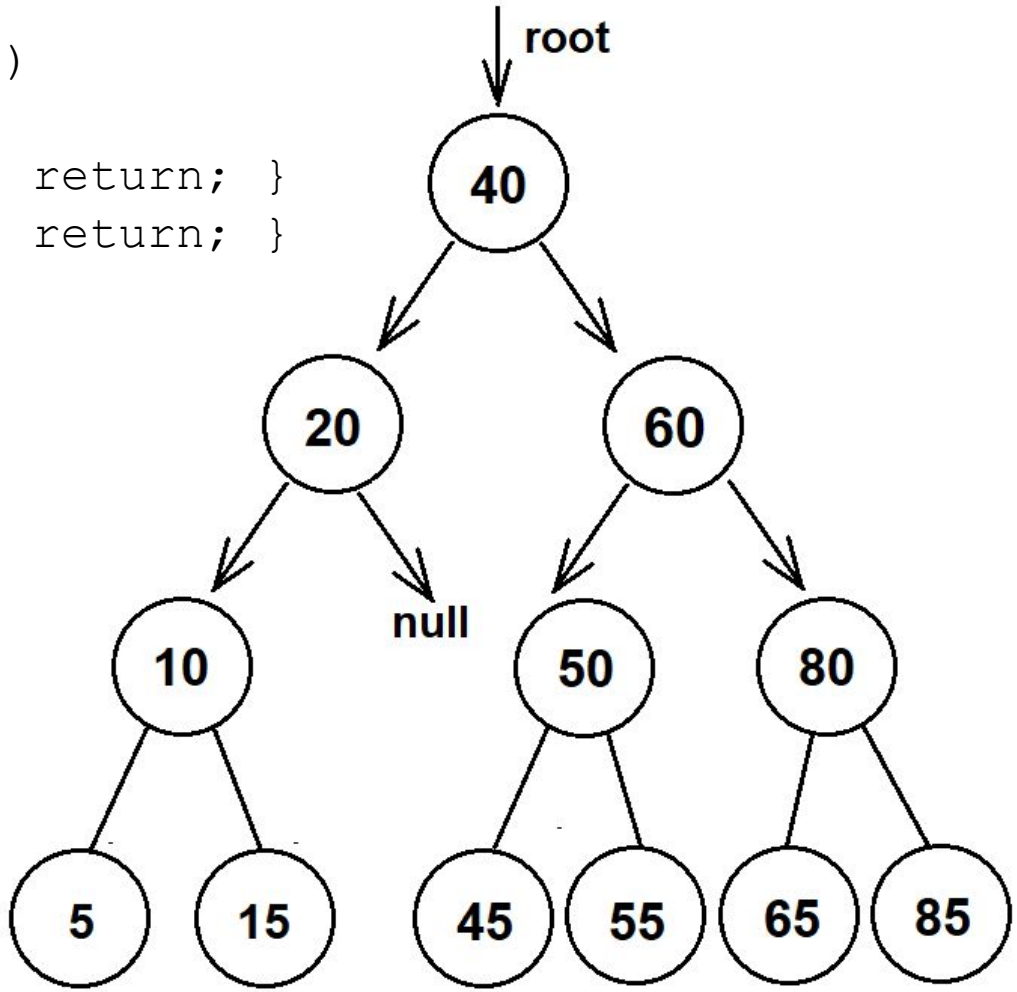


Удаление узла Если **корень**

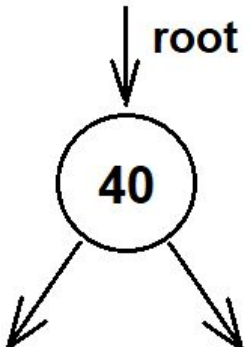
```
public void Remove(T value)
{
   return Remove(value, root);
}
private void Remove(T value, TreeNode<T> subroot)
{
   if (subroot == null) return;  TreeNode<T> subtree = null;
   if (subroot.Left != null && subroot.Left.Value == value)
   {
      if (subroot.Left.Left == null && subroot.Left.Right == null)
      {   subroot.Left = null; return; }
      if (subroot.Left.Left == null) subtree = subroot.Left.Right;
      if (subroot.Left.Right == null) subtree = subroot.Left.Left;
      if (subtree != null) { subroot.Left = subtree;  return;}
      subtree = subroot.Left.Left;
      subroot.Left = subroot.Left.Right;
      TreeNode<T> min = subroot.Left;
      while (min.Left!=null) min = min.Left;
      min.Left = subtree;  return;
   }
   if (subroot.Right != null && subroot.Right.Value == value)
   {
      …
   }
   if (subroot.Value < value) Remove(value, subroot.Right);
   if (subroot.Value > value) Remove(value, subroot.Left);
}
```



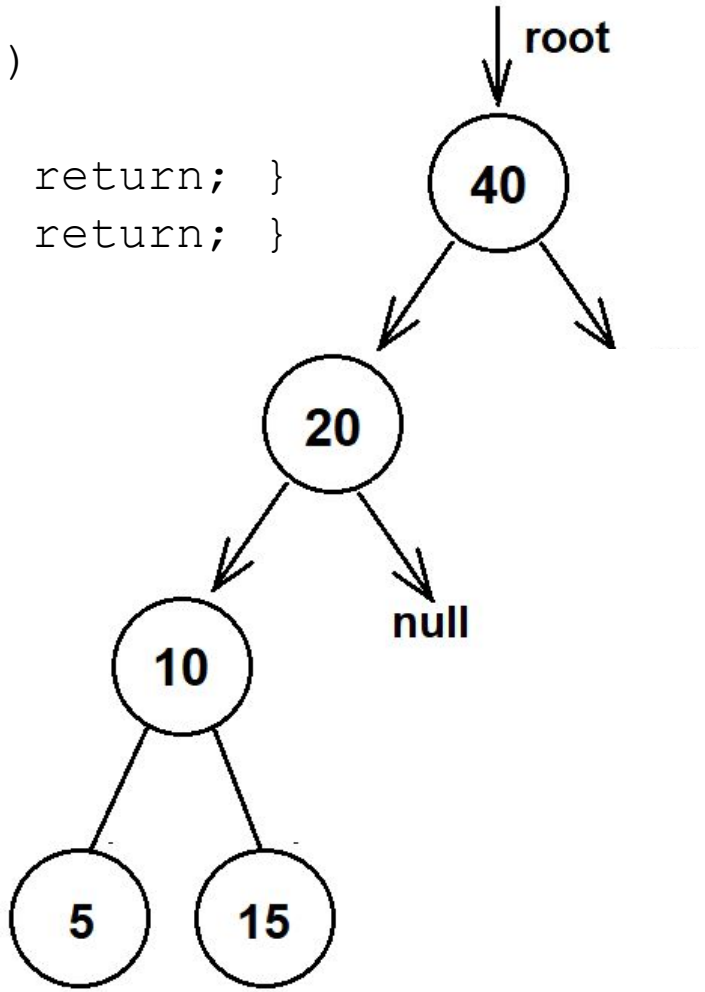
```
public void Remove(T value)
{
    if (root != null && root.Value == value)
    {
        if (root.Left == null && root.Right == null)
        {   root = null; return; }
        if (root.Left == null) { root = root.Right; return; }
        if (root.Right == null) { root = root.Left; return; }
        TreeNode<T> subtree = root.Left;
        root = root.Right;
        TreeNode<T> min = root;
        while (min.Left!=null) min = min.Left;
        min.Left = subtree;
        return;
    }
   return Remove(value, root);
}
private void Remove(T value, TreeNode<T> subroot)
{
   if (subroot == null) return;  TreeNode<T> subtree = null;
   if (subroot.Left != null && subroot.Left.Value == value)
   { … }
   if (subroot.Right != null && subroot.Right.Value == value)
   { … }
   if (subroot.Value < value) Remove(value, subroot.Right);
   if (subroot.Value > value) Remove(value, subroot.Left);
}
```



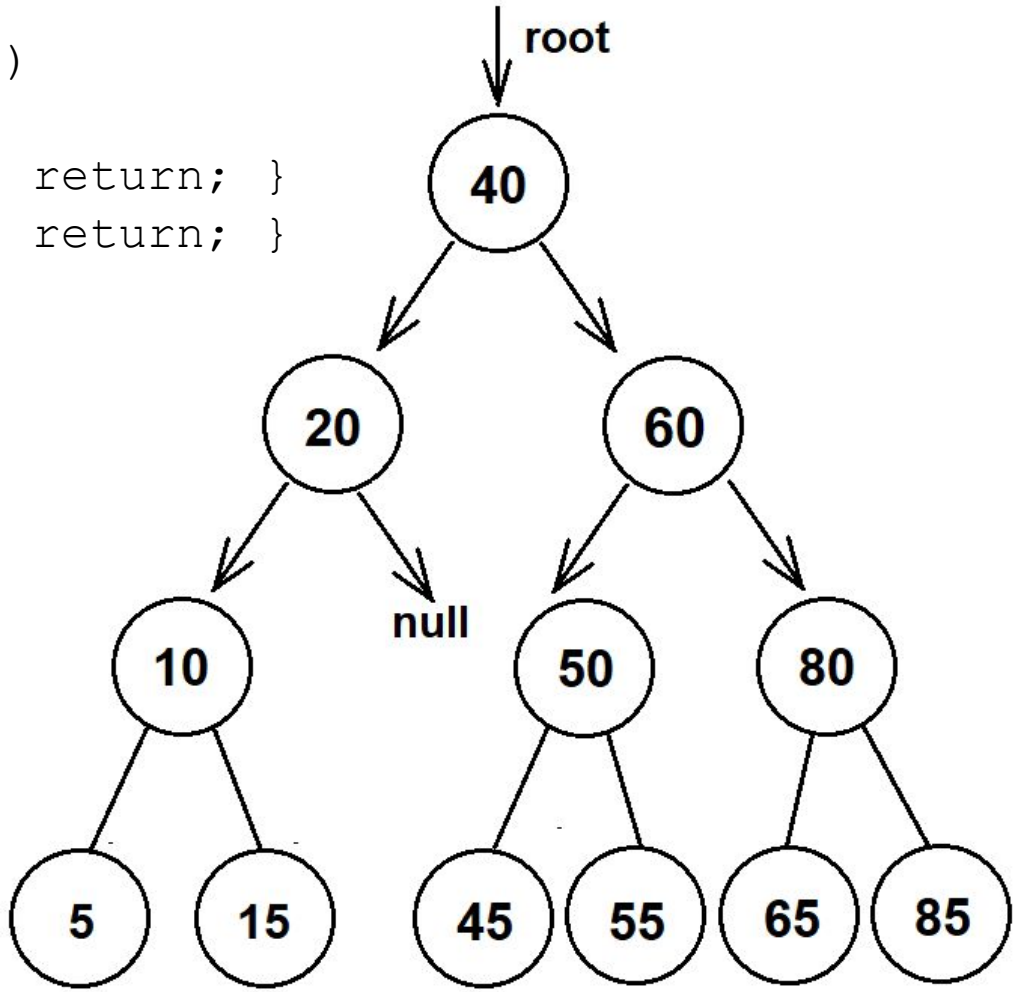
```
public void Remove(T value)
{
    if (root != null && root.Value == value)
    {
        if (root.Left == null && root.Right == null)
        {   root = null; return; }
        if (root.Left == null) { root = root.Right; return; }
        if (root.Right == null) { root = root.Left; return; }
        TreeNode<T> subtree = root.Left;
        root = root.Right;
        TreeNode<T> min = root;
        while (min.Left!=null) min = min.Left;
        min.Left = subtree;
        return;
    }
   return Remove(value, root);
}
private void Remove(T value, TreeNode<T> subroot)
{
   if (subroot == null) return;  TreeNode<T> subtree = null;
   if (subroot.Left != null && subroot.Left.Value == value)
   { … }
   if (subroot.Right != null && subroot.Right.Value == value)
   { … }
   if (subroot.Value < value) Remove(value, subroot.Right);
   if (subroot.Value > value) Remove(value, subroot.Left);
}
```



```
public void Remove(T value)
{
    if (root != null && root.Value == value)
    {
        if (root.Left == null && root.Right == null)
        {   root = null; return; }
        if (root.Left == null) { root = root.Right; return; }
        if (root.Right == null) { root = root.Left; return; }
        TreeNode<T> subtree = root.Left;
        root = root.Right;
        TreeNode<T> min = root;
        while (min.Left!=null) min = min.Left;
        min.Left = subtree;
        return;
    }
   return Remove(value, root);
}
private void Remove(T value, TreeNode<T> subroot)
{
   if (subroot == null) return;  TreeNode<T> subtree = null;
   if (subroot.Left != null && subroot.Left.Value == value)
   { … }
   if (subroot.Right != null && subroot.Right.Value == value)
   { … }
   if (subroot.Value < value) Remove(value, subroot.Right);
   if (subroot.Value > value) Remove(value, subroot.Left);
}
```



```
public void Remove(T value)
{
    if (root != null && root.Value == value)
    {
        if (root.Left == null && root.Right == null)
        {   root = null; return; }
        if (root.Left == null) { root = root.Right; return; }
        if (root.Right == null) { root = root.Left; return; }
        TreeNode<T> subtree = root.Left;
        root = root.Right;
        TreeNode<T> min = root;
        while (min.Left!=null) min = min.Left;
        min.Left = subtree;
        return;
    }
   return Remove(value, root);
}
private void Remove(T value, TreeNode<T> subroot)
{
   if (subroot == null) return;  TreeNode<T> subtree = null;
   if (subroot.Left != null && subroot.Left.Value == value)
   { … }
   if (subroot.Right != null && subroot.Right.Value == value)
   { … }
   if (subroot.Value < value) Remove(value, subroot.Right);
   if (subroot.Value > value) Remove(value, subroot.Left);
}
```



```java
public class TreeNode<T  extends Comparable<T>> {

    T value;
    TreeNode<T> left;
    TreeNode<T> right;
    TreeNode<T> parent;

    public TreeNode(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

    public TreeNode<T> getLeft() {
        return left;
    }

    public void setLeft(TreeNode<T> left) {
        this.left = left;
    }

    public TreeNode<T> getRight() {
        return right;
    }

    public void setRight(TreeNode<T> right) {
        this.right = right;
    }

    public TreeNode<T> getParent() {
        return parent;
    }
    public void setParent(TreeNode<T> parent) {
        this.parent = parent;
    }
}
```

А если добавить родительский узел? (JAVA)

```
public class MyTree<T extends Comparable<T>> {

    TreeNode<T> root;

    public void Delete(T value){
        …
    }

    private void Delete(???) {
        …
    }
}
```

```
public void Delete(T value){
     if (root == null) return;
     if (root.getValue() == value) {
         if (root.getLeft() == null && root.getRight() == null) {
             root = null;
             return;
         }
         if (root.getLeft() == null) {
             root = root.getRight();
             root.setParent(null);
             return;
         }
         if (root.getRight() == null) {
             root = root.getLeft();
             root.setParent(null);
             return;
         }
         TreeNode<T> subtree = root.getLeft();
         root = root.getRight();
         root.setParent(null);
         TreeNode<T> min = root;
         while (min.getLeft()!=null) min = min.getLeft();
         min.setLeft(subtree);
         subtree.setParent(min);
     }
     else …
}

private void Delete(???) {  …  }
```

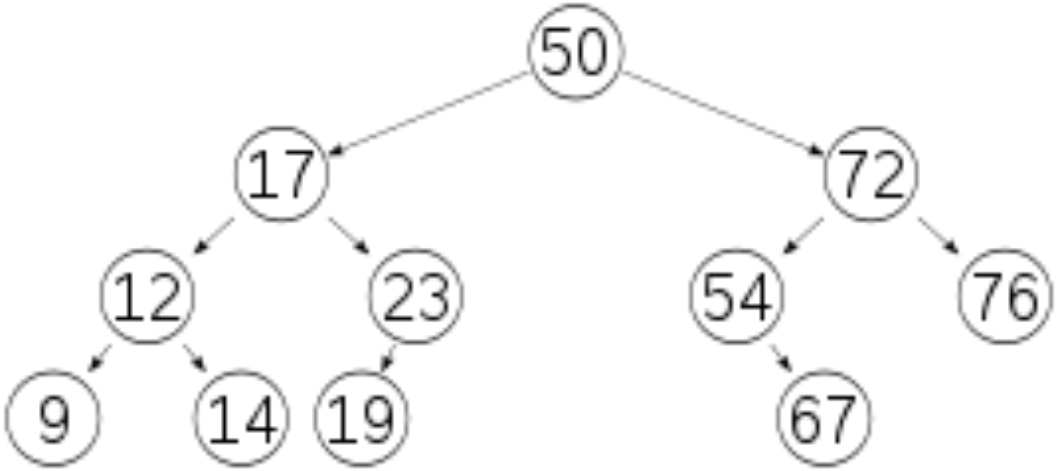
```
public void Delete(T value){
      if (root == null) return;
    if (root.getValue() == value) {
        if (root.getLeft() == null && root.getRight() == null) {
            root = null;
            return;
        }
        if (root.getLeft() == null) {
            root = root.getRight();
            root.setParent(null);
            return;
        }
        if (root.getRight() == null) {
            root = root.getLeft();
            root.setParent(null);
            return;
        }
        TreeNode<T> subtree = root.getLeft();
        root = root.getRight();
        root.setParent(null);
        TreeNode<T> min = root;
        while (min.getLeft()!=null) min = min.getLeft();
        min.setLeft(subtree);
        subtree.setParent(min);
      }
      else if (value.compareTo(root.getValue())>=0) Delete(value, root.getRight(), false);
      else if (value.compareTo(root.getValue())<0) Delete(value, root.getLeft(), true);
}

private void Delete(T value, TreeNode<T> subroot, boolean isLeft) {
      …
}
```

```
private void Delete(T value, TreeNode<T> subroot, boolean isLeft) {
    if (subroot == null) return;
    if (subroot.getValue() == value) {
        if (subroot.getLeft() == null && subroot.getRight() == null) {
            if (isLeft) subroot.getParent().setLeft( null);
            else subroot.getParent().setRight( null);
            return;
        }
        if (subroot.getLeft() == null) {
            subroot.getRight().setParent(subroot.getParent());
            if (isLeft) subroot.getParent().setLeft(subroot.getRight());
            else subroot.getParent().setRight(subroot.getRight());
            return;
        }
        if (subroot.getRight() == null) {
            subroot.getLeft().setParent(subroot.getParent());
            if (isLeft) subroot.getParent().setLeft(subroot.getLeft());
            else subroot.getParent().setRight(subroot.getLeft());
            return;
        }
        TreeNode<T> subtree = subroot.getLeft();
        TreeNode<T> min = subroot.getRight();
        subroot.getRight().setParent(subroot.getParent());
        if (isLeft) subroot.getParent().setLeft(subroot.getRight());
        else subroot.getParent().setRight(subroot.getRight());
        while (min.getLeft()!=null) min = min.getLeft();
        min.setLeft(subtree);
        subtree.setParent(min);
    }
    else if (value.compareTo(subroot.getValue())>= 0) Delete(value, subroot.getRight(),
false);
    else if (value.compareTo(subroot.getValue())< 0) Delete(value, subroot.getLeft(),  true);
```

АВЛ-дерево

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

АВЛ — аббревиатура, образованная первыми буквами фамилий создателей (советских учёных) Адельсон-Вельского Георгия Максимовича и Ландиса Евгения Михайловича.



Максимальная высота АВЛ-дерева при заданном числе узлов

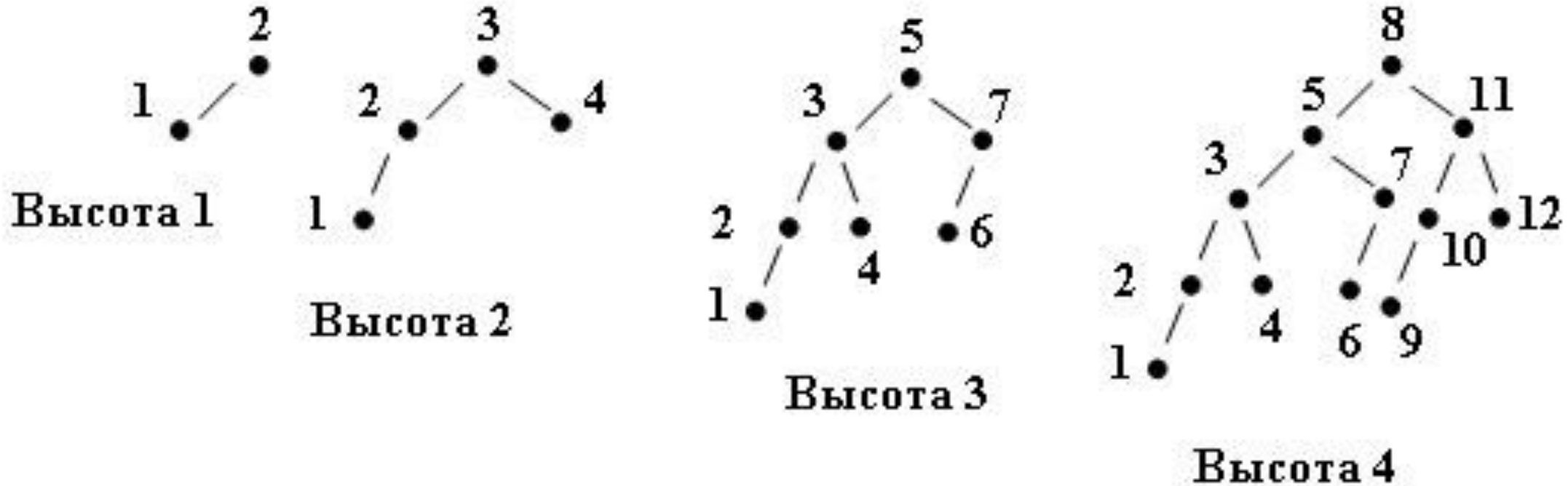
$$h \leq [1.45\ log_2(\mathrm{n} + 2)]$$

Дерево Фибоначчи

Дерево Фибоначчи — АВЛ-дерево с наименьшим числом вершин при заданной высоте (глубине).

- Если для какой-либо из вершин высота поддерева, для которого эта вершина является корнем, равна ***h***, то правое и левое поддерево этой вершины имеют высоты равные соответственно ***h-1*** и ***h-2*** или ***h-2*** и ***h-1*** . Каждое поддерево дерева Фибоначчи также является деревом Фибоначчи.
- Пустое дерево — дерево Фибоначчи высоты 0.
- Дерево с одной вершиной — дерево Фибоначчи высоты 1.

Дерево Фибоначчи



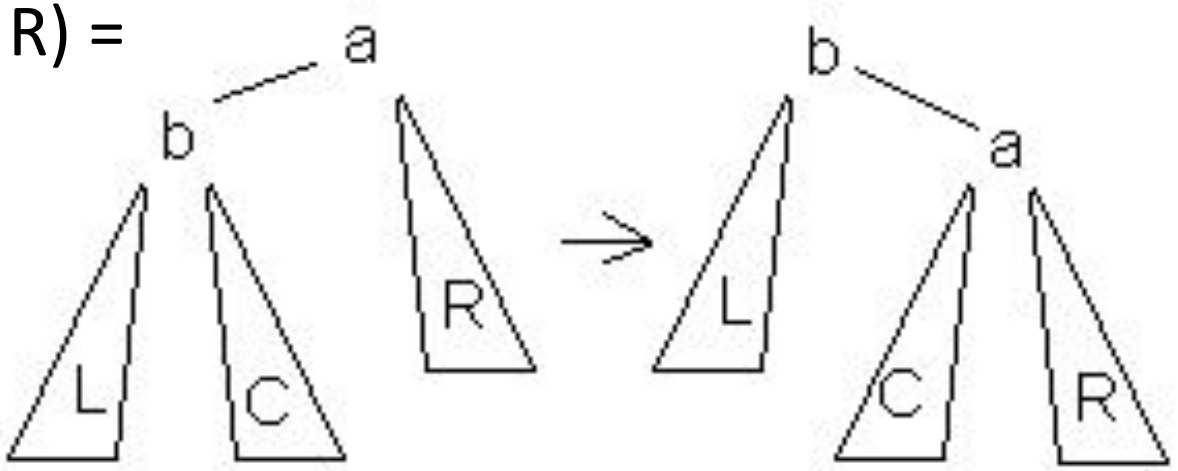
Балансировка

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины.

Правый поворот (малое правое вращение)

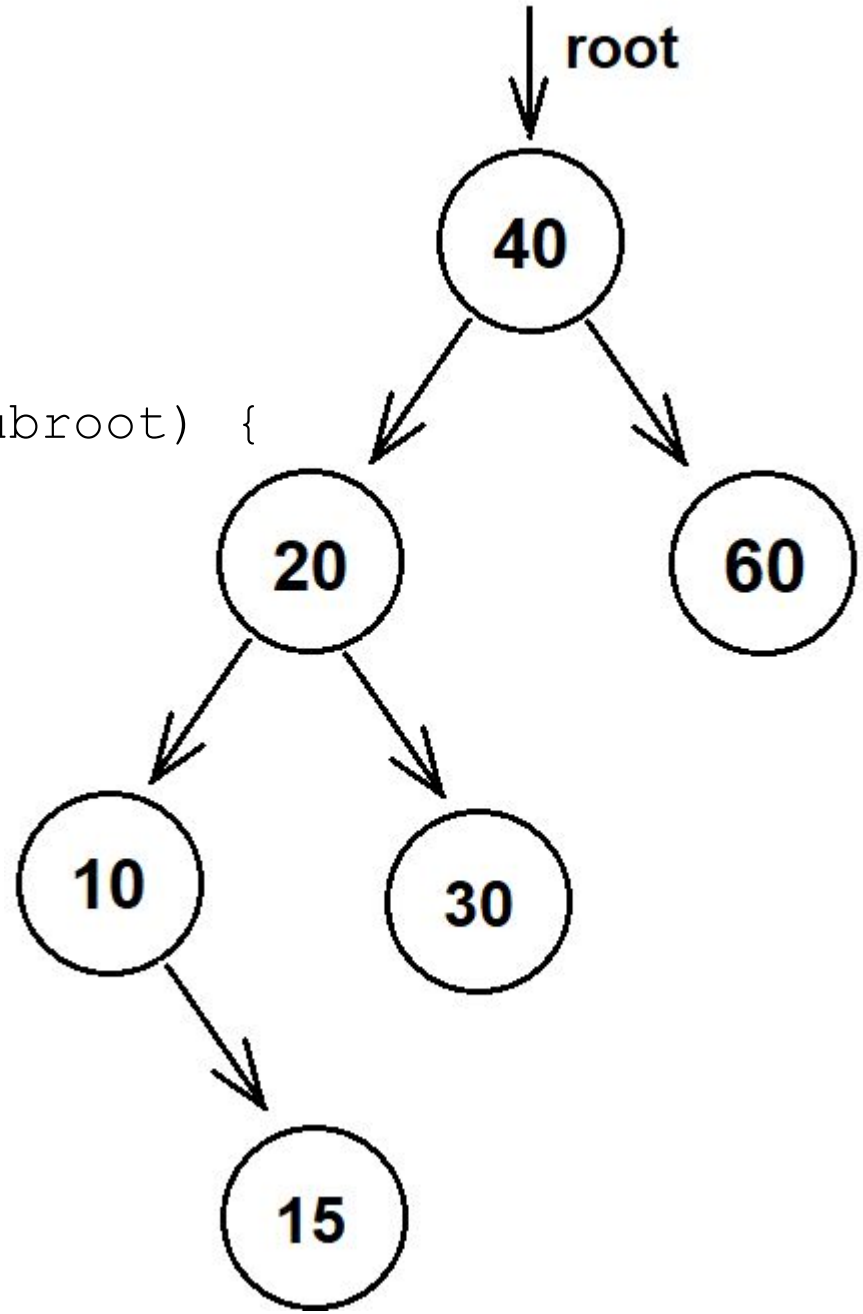
Данное вращение используется тогда, когда:

- (высота b-поддерева — высота R) = 2
- высота C <= высота L.



Правый поворот (малое правое вращение)

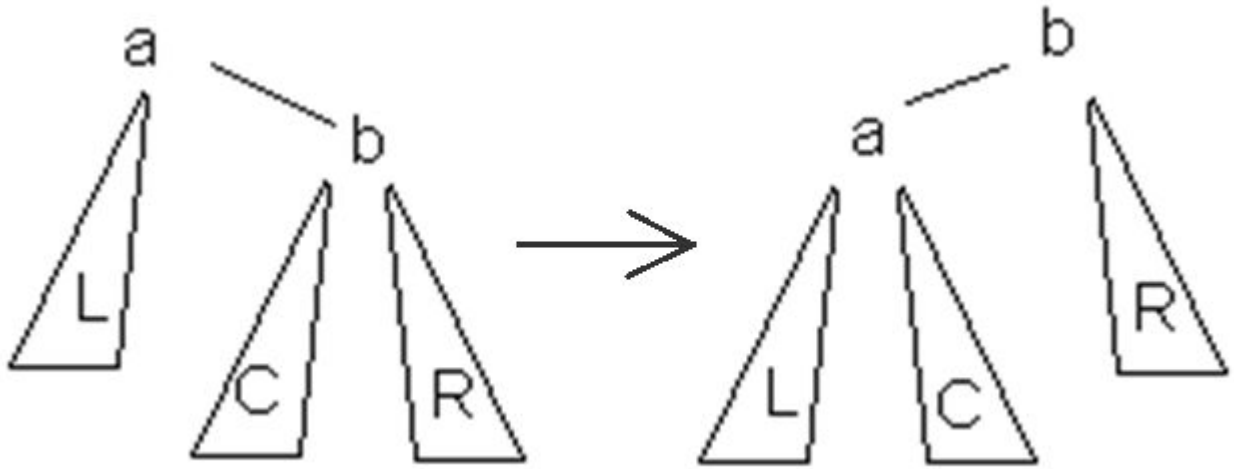
```
private TreeNode<T> RightRotate(TreeNode<T> subroot) {
    TreeNode<T> b = subroot.Left;
    subroot.Left = b.Right;
    b.Right = subroot;
    return b;
}
```



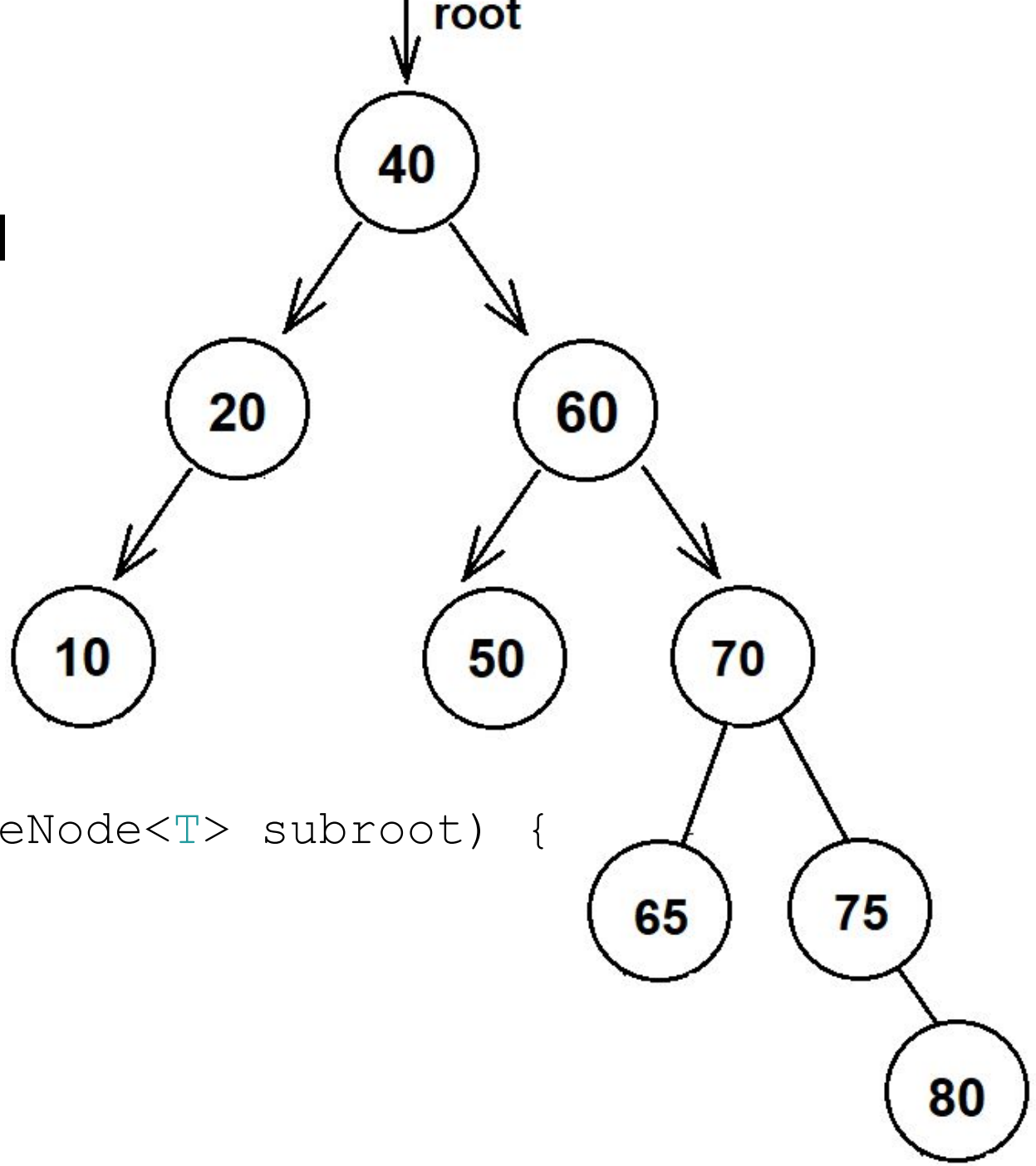
Левый поворот (малое левое вращение)

Данное вращение используется тогда, когда:

- (высота b-поддерева — высота L) = 2
- высота C <= высота R.



Левый поворот (малое левое вращени

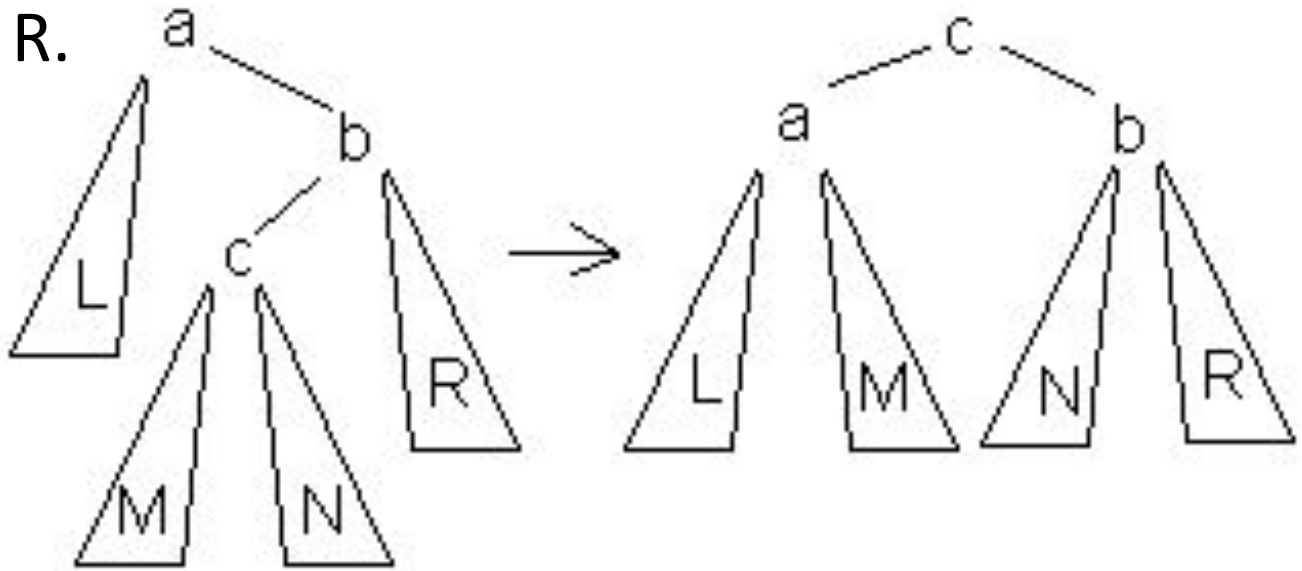



```
private TreeNode<T> LeftRotate(TreeNode<T> subroot) {
    TreeNode<T> b = subroot.Right;
    subroot.Right = b.Left;
    b.setLeft = subroot;
    return b;
}
```

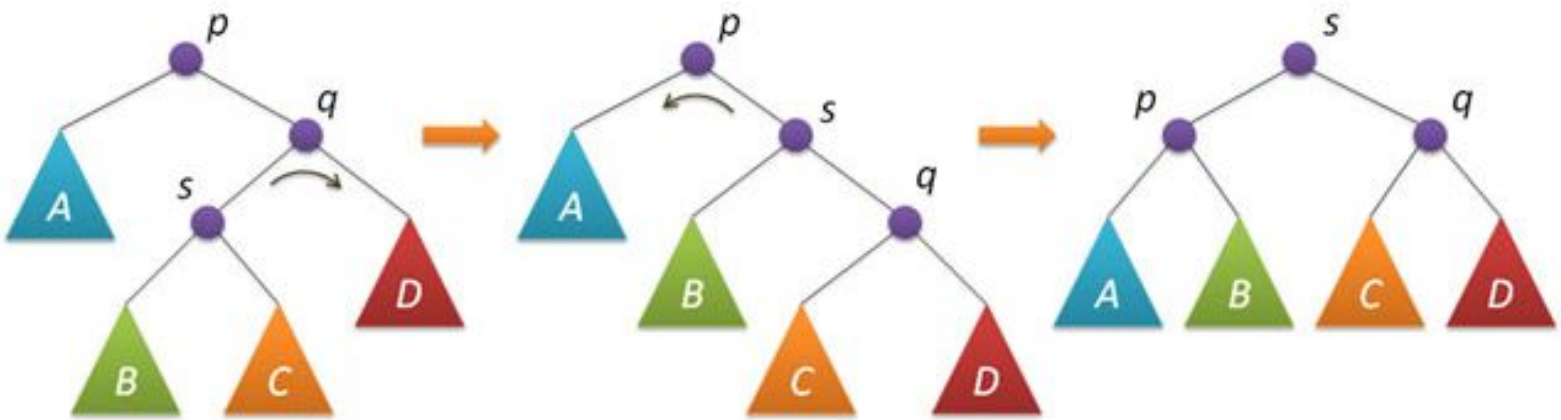
Большой левый поворот (большое левое вращение)

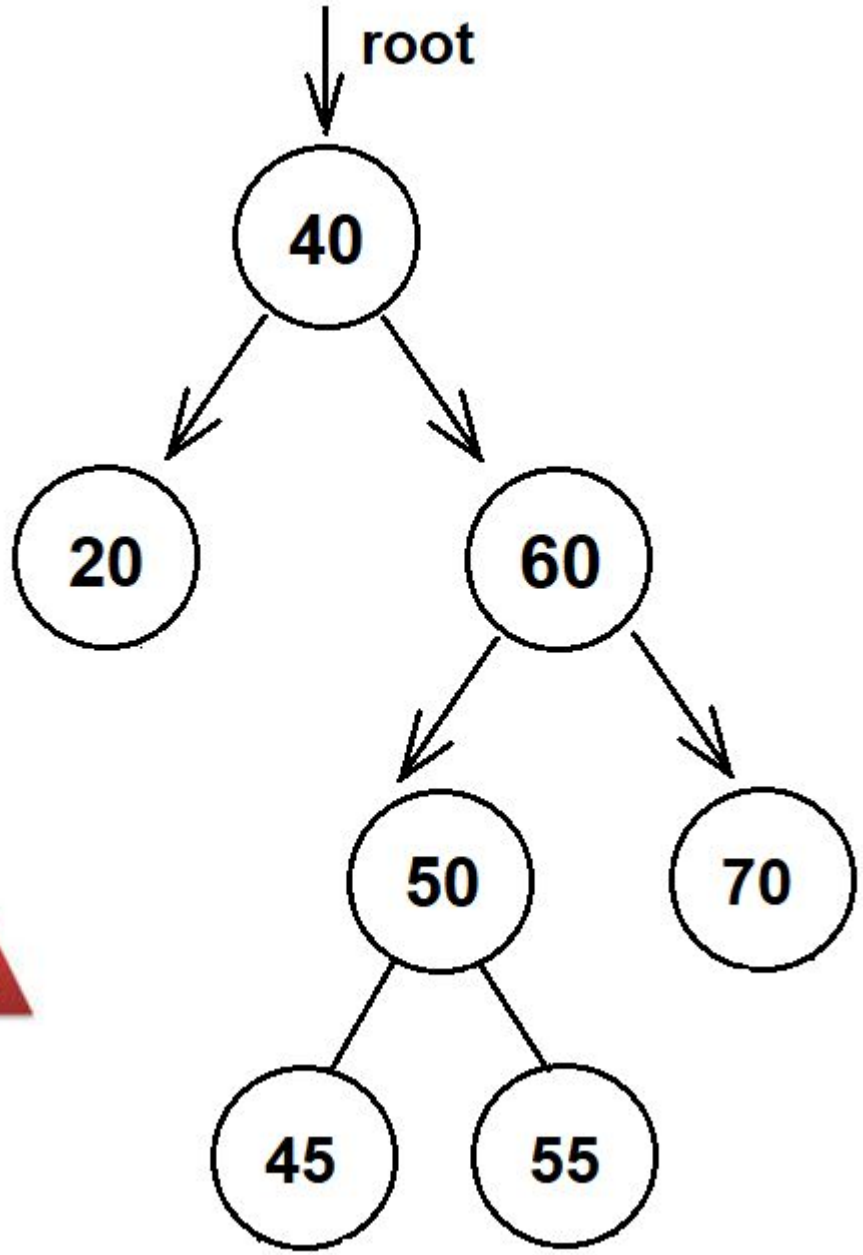
Данное вращение используется тогда, когда

- (высота b-поддерева — высота L) = 2
- высота c-поддерева > высота R.

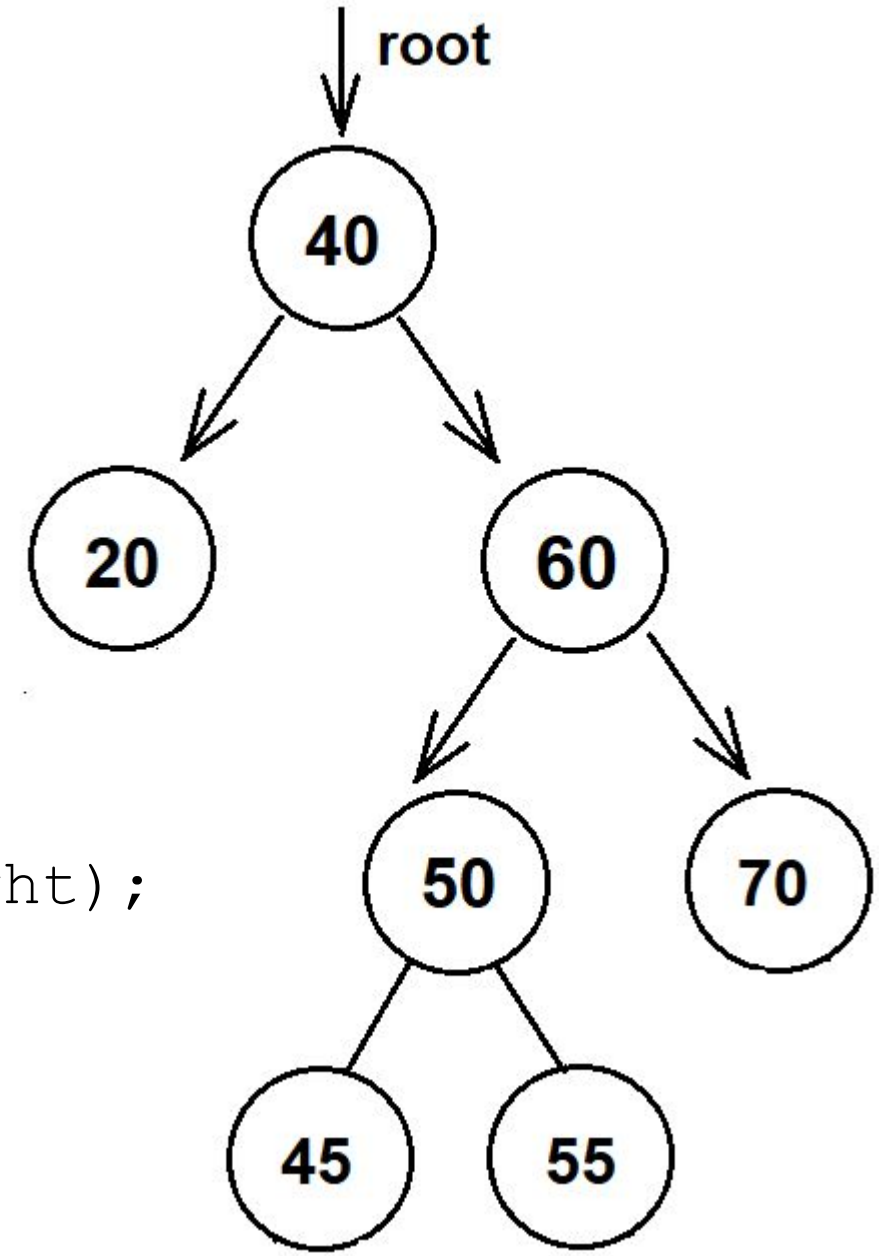


Большой левый поворот (большое левое вращение)





Большой левый поворот (большое левое вращение)

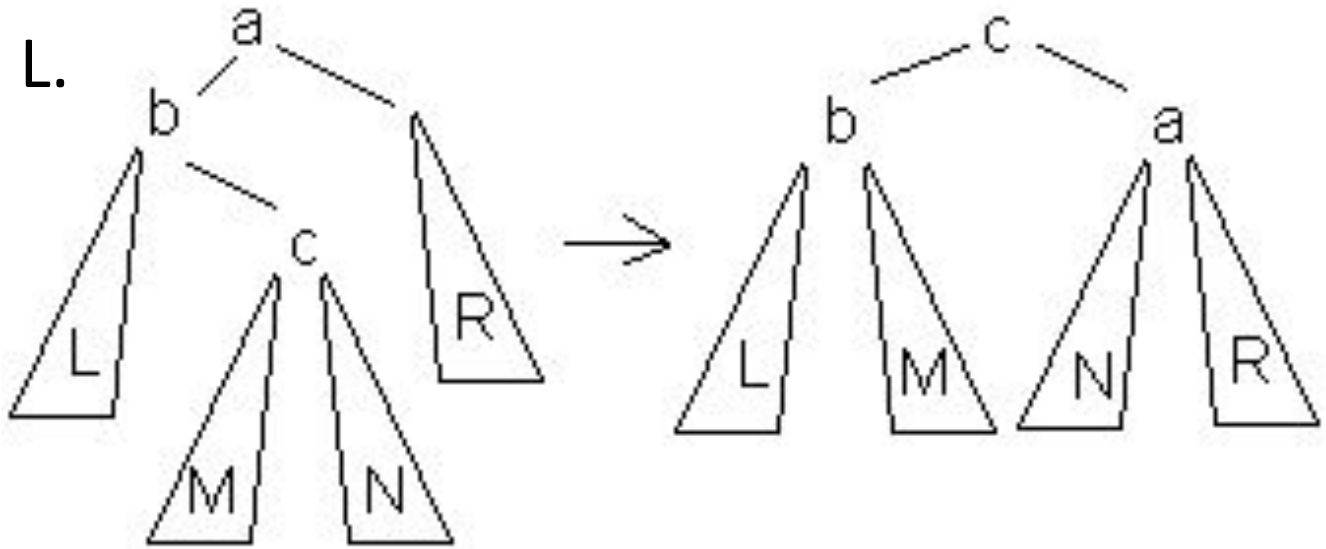



```
subroot.Right = RightRotate(subroot.Right);
return LeftRotate(subroot);
```

Большой правый поворот (большое правое вращение)

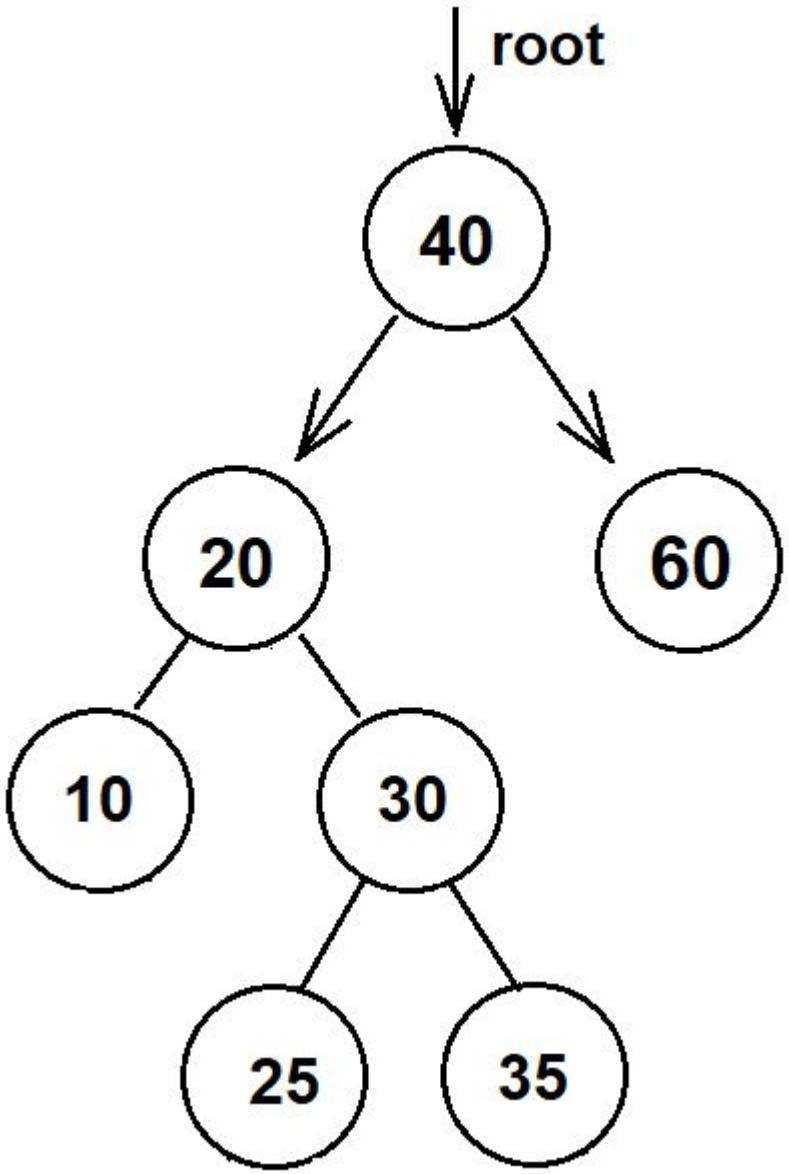
Данное вращение используется тогда, когда

- (высота b-поддерева — высота R) = 2
- высота c-поддерева > высота L.



Большой правый поворот (большое правое вращение)

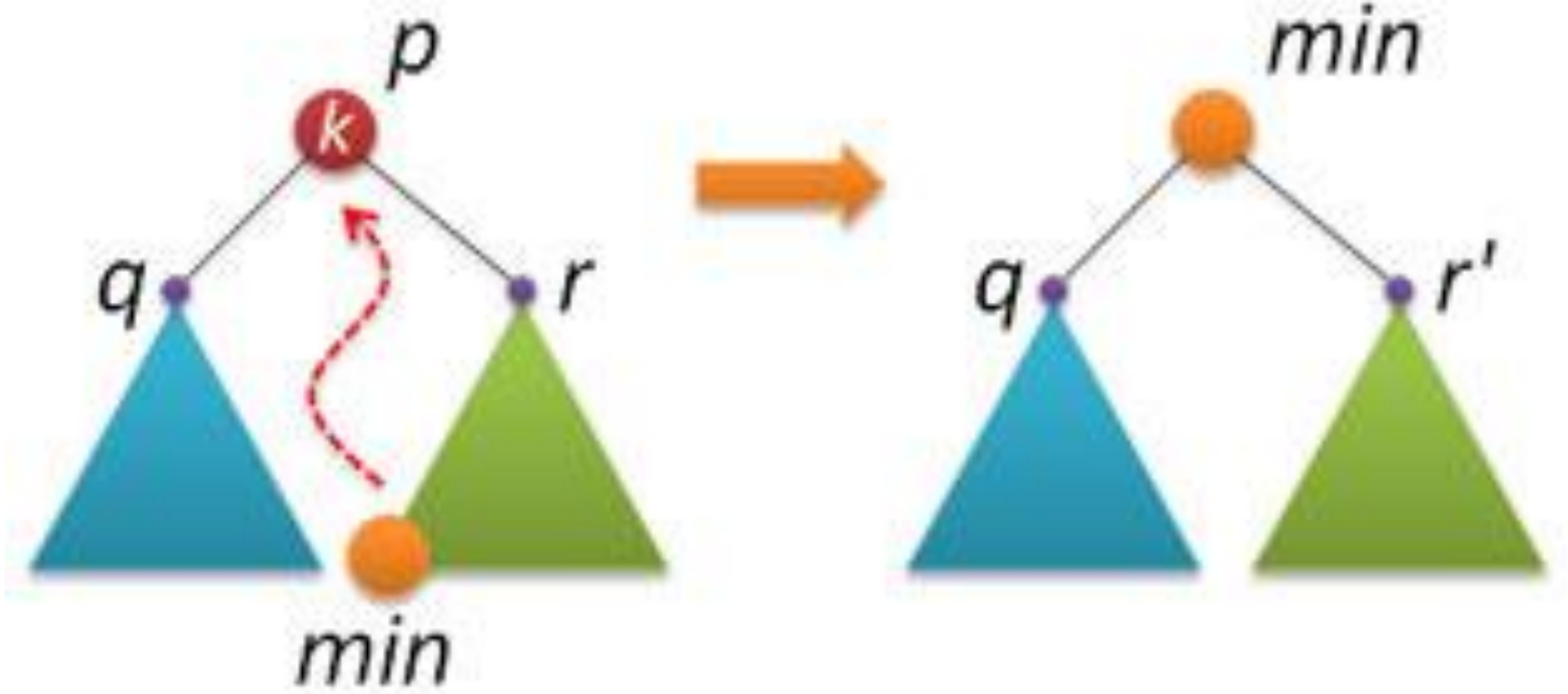
```
subroot.Left = LeftRotate(subroot.Left);
return RightRotate(subroot);
```



Балансировка узла

```
TreeNode<T> Balance(TreeNode<T> subroot) // балансировка узла p
{
    if( (GetHeight(subroot.getRight()) - GetHeight(subroot.getLeft()))==2 )
    {
        if( GetHeight(subroot.getRight().getLeft())
                                > GetHeight(subroot.getRight().getRight()) )
            subroot.setRight(RightRotate(subroot.getRight()));
        return LeftRotate(subroot);
    }
    if( (GetHeight(subroot.getRight()) - GetHeight(subroot.getLeft()))==-2 )
    {
        if( GetHeight(subroot.getLeft().getRight())
                                > GetHeight(subroot.getLeft().getLeft()) )
            subroot.setLeft(LeftRotate(subroot.getLeft()));
        return RightRotate(subroot);
    }
    return subroot; // балансировка не нужна
}
```

Удаление узла из АВЛ дерева



Удаление узла из АВЛ дерева

```
// поиск узла с минимальным ключом в поддереве
TreeNode<T> GetMin(TreeNode<T> subroot)
{
    if (subroot.Left == null) return subroot;
    return GetMin(subroot.Left);
}

// удаление узла с минимальным ключом из поддерева
TreeNode<T> RemoveMin(TreeNode<T> subroot)
{
    if( subroot.Left == null )
        return subroot.Right;
    subroot.Left = RemoveMin(subroot.Left);
    return Balance(subroot);
}
```

Удаление узла из АВЛ дерева

```
TreeNode<T> Remove(TreeNode<T> subroot, T value) // удаление значения из поддерева
{
    if( subroot == null ) return null;
    if( value.CompareTo(subroot.Value) < 0)
        subroot.Left = Remove(subroot.Left, value);
    else if( value.CompareTo(subroot.Value) > 0)
        subroot.Right = Remove(subroot.Right, value));
    else //  нашли тот узел, который надо удалить
    {
        TreeNode<T> l = subroot.Left;
        TreeNode<T> r = subroot.Right;
        if( r == null ) return l;
        TreeNode<T> min = GetMin(r);
        min.Right = RemoveMin(r);
        min.Left = l;
        return Balance(min);
    }
    return Balance(subroot);
}
```


Я
АЛГОРИТМЫ И
СТРУКТУРЫ ДАННЫХ
Я
РАБОТОДАТЕЛЬ
НА СОБЕСЕДОВАНИИ