Алгоритмы поиска

Двоичный поиск в упорядоченном массиве

Бинарное дерево поиска

АТД «Словарь» (dictionary)

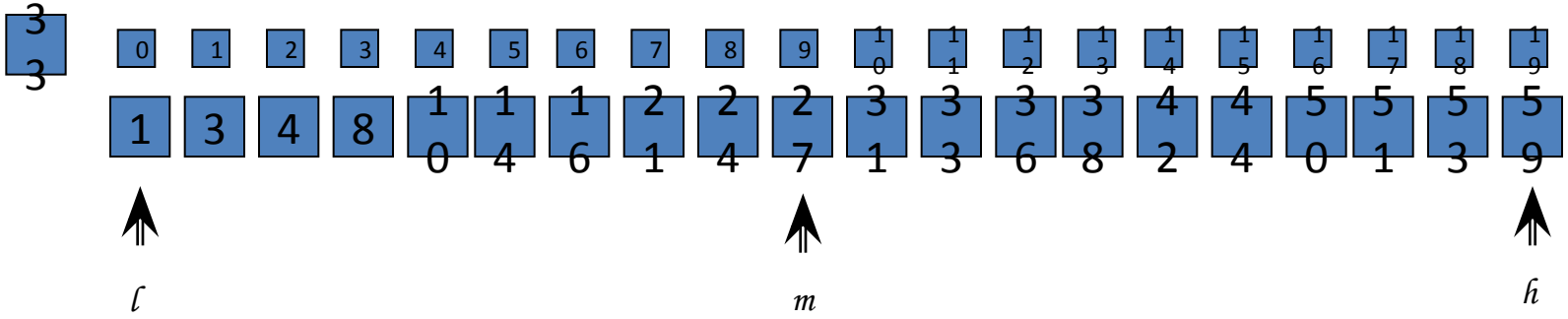
- **Словарь** (dictionary) – структура данных для хранения пар вида «ключ» – «значение» (key – value)
- **Альтернативные название** – ассоциативный массив (associative array, map)
- В словаре может быть только одна пара с заданным ключом

Ключ (key)	Значение (value)
890	Слон
1200	Кит
260	Лев
530	Жираф

АТД «Словарь» (dictionary)

Операция	Описание
Add(*map, key, value*)	Добавляет в словарь *map* пару (*key*, *value*)
Lookup(*map, key*)	Возвращает из словаря *map* значение ассоциированное с ключом *key*
Remove(*map, key*)	Удаляет из словаря *map* пару с ключом *key*
Min(*map*)	Возвращает из словаря *map* минимальное значение
Max(*map*)	Возвращает из словаря *map* максимальное значение

Двоичный поиск в упорядоченном массиве




```
private static int binSearch(int[] data, int key) {
    int l = 0,
        h = data.length-1;
    while (l < h) {
        int m = (l + h) / 2;     // (l <= m < h)
        if (data[m] < key) l = m + 1; else h = m;
    }
    return (data[l] == key ? l : -1);
}
```

Инвариант цикла: `l <= h` && «если `key == data[k]`, то `l <= k <= h`»

Реализация словаря на основе массива

Операция	Неотсортированный массив	Отсортированный массив
Add (*map*, *key*, *value*)	$O(1)$ (добавление в конец)	$O(n)$ (поиск позиции)
Lookup (*map*, *key*)	$O(n)$	$O(\log n)$ (бинарный поиск)
Remove (*map*, *key*)	$O(n)$ (поиск элемента и перенос последнего на место удаляемого)	$O(n)$ (перемещение элементов)
Min(*map*)	$O(n)$	$O(1)$ (элемент $v[1]$)
Max(*map*)	$O(n)$	$O(1)$ (элемент $v[n]$)

Реализация АТД «Словарь»

- Реализации словарей отличаются вычислительной сложностью операций и объемом требуемой памяти для хранения пар «ключ-значение»

- Распространение получили следующие реализации:

1. **Деревья поиска (Search trees)**
2. **Хэш-таблицы (Hash tables)**
3. **Списки с пропусками (Skip lists)**
4. Связные списки, массивы

Реализация словаря на основе связного списка

Операция	Неотсортированный связный список	Отсортированный связный список
Add (*map*, *key*, *value*)	$O(1)$ (добавление в начало)	$O(n)$ (поиск позиции)
Lookup (*map*, *key*)	$O(n)$	$O(n)$
Remove (*map*, *key*)	$O(n)$ (поиск элемента)	$O(n)$ (поиск элемента)
Min(*map*)	$O(n)$	$O(1)$
Max(*map*)	$O(n)$	$O(n)$ или $O(1)$, если поддерживать указатель на последний элемент

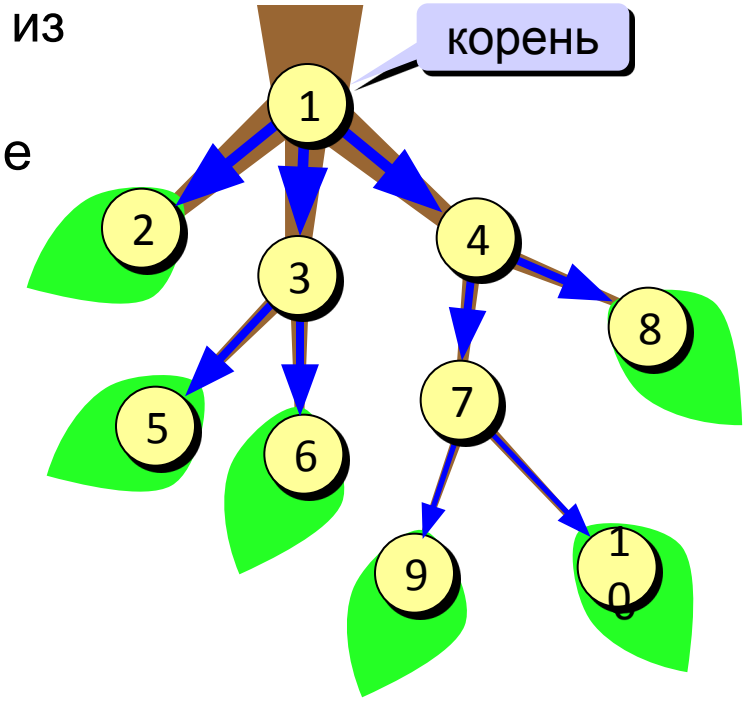
Бинарные деревья поиска

Деревья

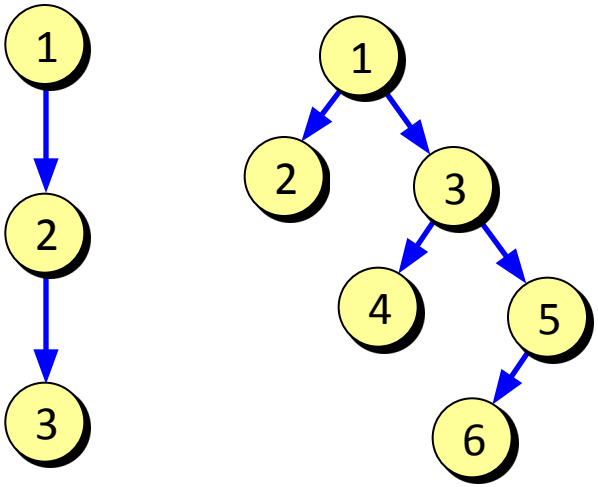
Дерево – это структура данных, состоящая из узлов и соединяющих их направленных ребер (дуг), причем в каждый узел (кроме корневого) ведет ровно одна дуга.

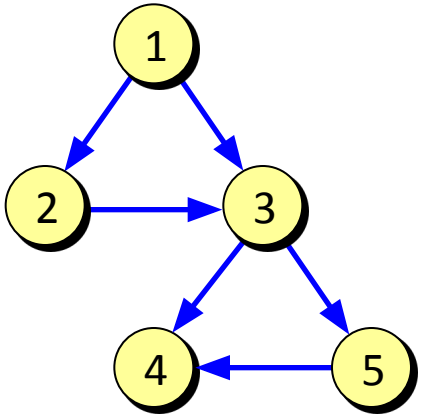
Корень – это начальный узел дерева.

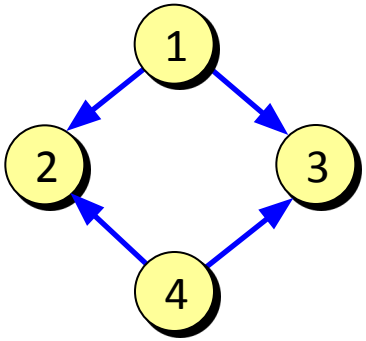
Лист – это узел, из которого не выходит ни одной дуги.



Какие структуры – не деревья?







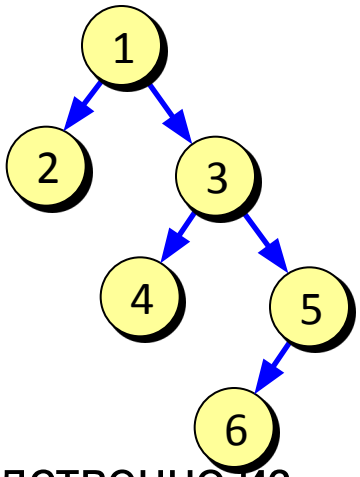
Деревья

С помощью деревьев изображаются отношения подчиненности (иерархия, «старший – младший», «родитель – ребенок»).

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.



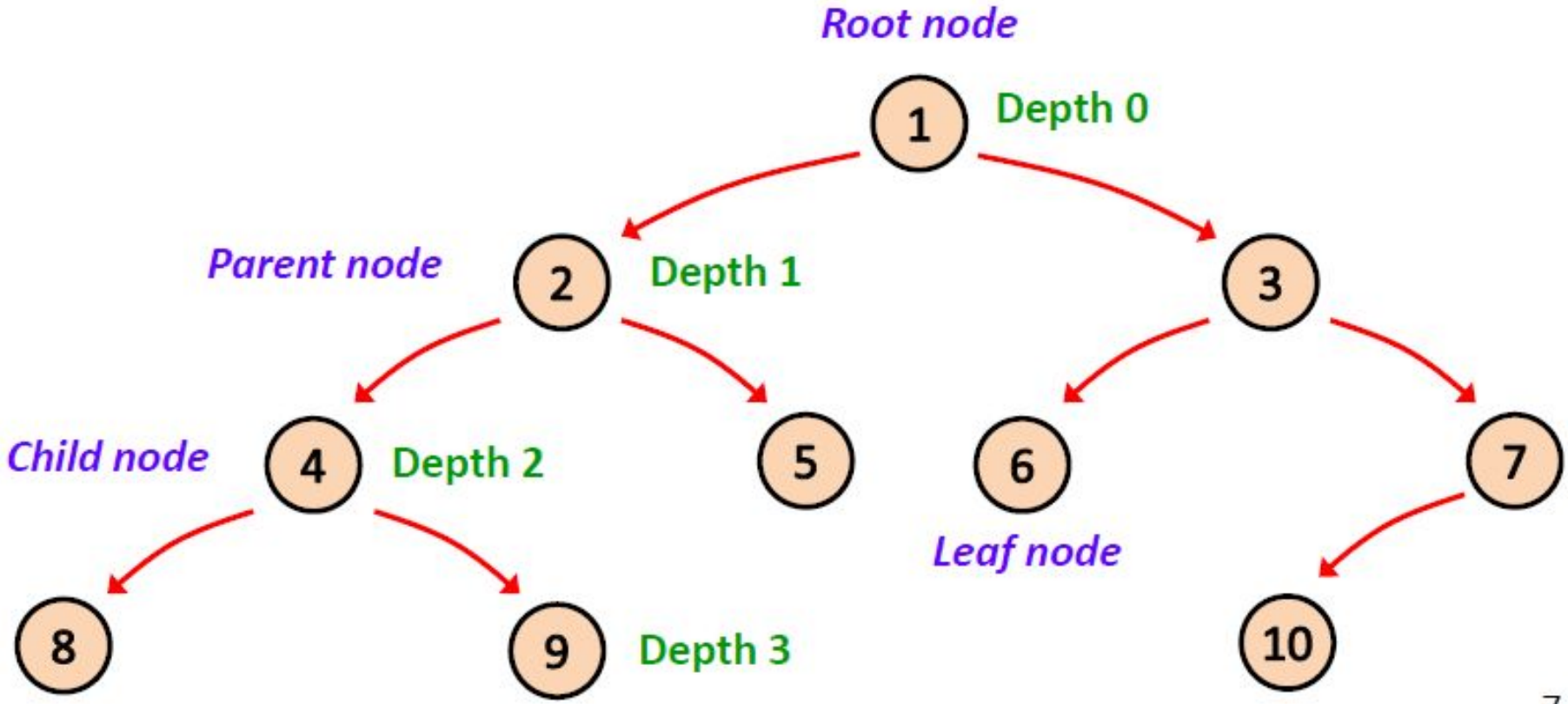
Сын узла x – это узел, в который существует дуга непосредственно из узла x.

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).

Бинарные деревья (binary trees)

- ***Бинарное дерево* (binary tree)** – это дерево (структура данных), в котором каждый узел (node) имеет не более двух дочерних узлов (child nodes)



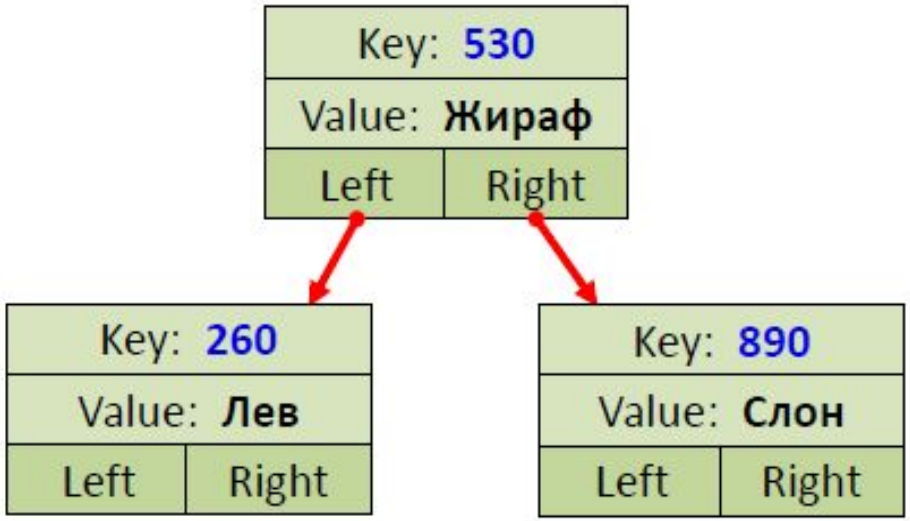
Бинарные деревья поиска (binary search trees)

- **Двоичное дерево поиска (binary search tree, BST)** – это двоичное дерево, в котором:

  1) каждый узел ***x*** (node) имеет не более двух дочерних узлов (child nodes) и содержит ключ (key) и значение (value)

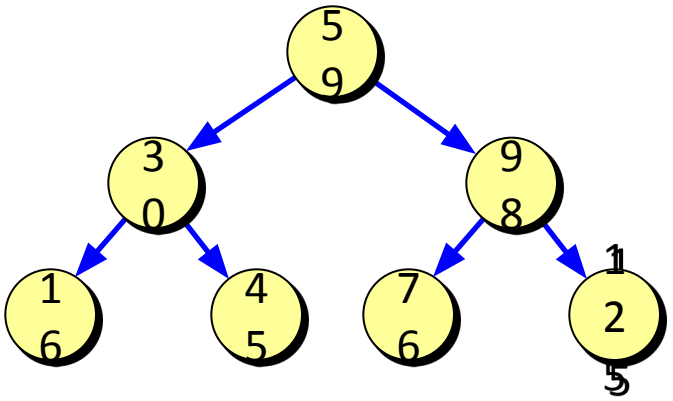
  2) ключи всех узлов левого поддерева узла *x* меньше значения его ключа

  3) ключи всех узлов правого поддерева узла *x* больше значения его ключа



Двоичные деревья поиска

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



? Какая закономерность?

Слева от каждого узла находятся узлы с меньшими ключами, а справа – с бóльшими.

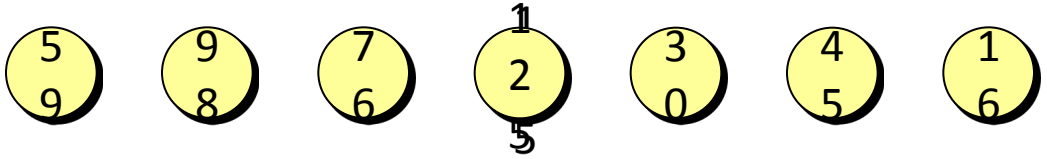
Как искать ключ, равный x:

1) если дерево пустое, ключ не найден;
2) если ключ узла равен x, то стоп.
3) если ключ узла меньше x, то искать x в левом поддереве;
4) если ключ узла больше x, то искать x в правом поддереве.

? Сведение задачи к такой же задаче меньшей размерности – это …?

Двоичные деревья поиска

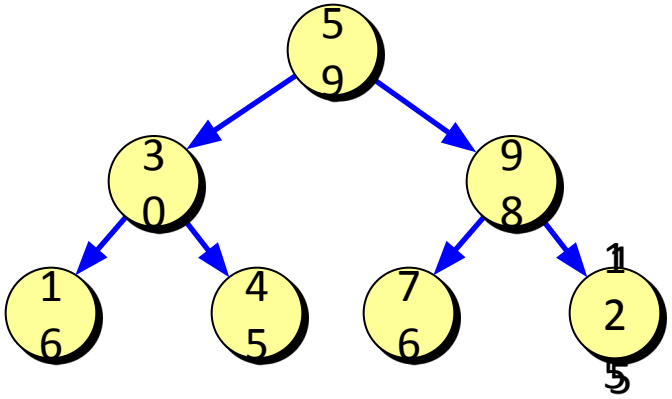
Поиск в массиве (N элементов):



При каждом сравнении отбрасывается 1 элемент.

Число сравнений – N.

Поиск по дереву (N элементов):



При каждом сравнении отбрасывается половина оставшихся элементов.

Число сравнений ~ $\log_2 N$.

быстрый поиск

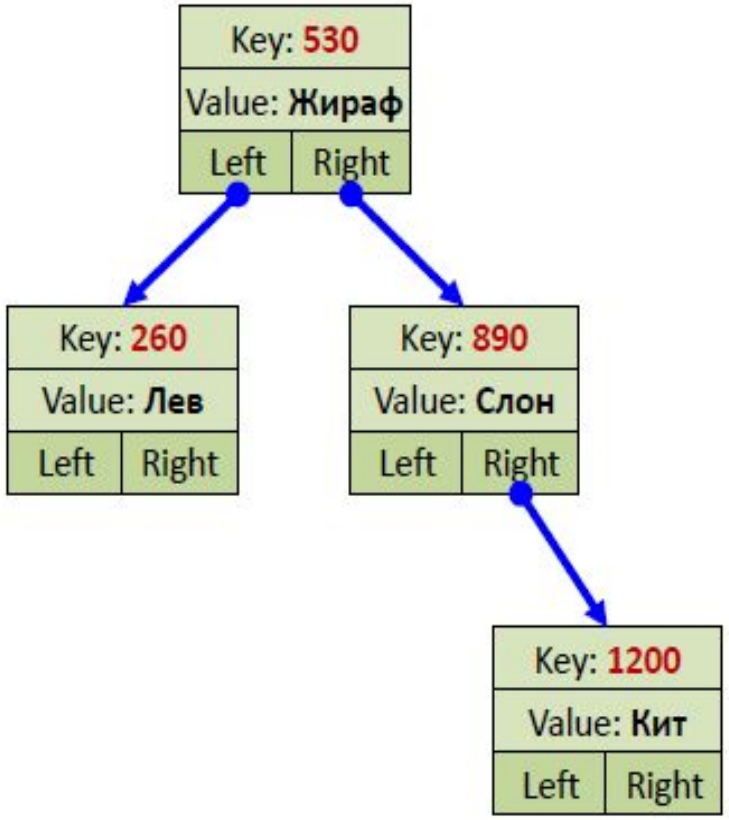
1) нужно заранее построить дерево;
2) желательно, чтобы дерево было минимальной высоты.

Двоичные деревья поиска (binary search trees)

Словарь

Ключ (Key)	Значение (Value)
530	Жираф
260	Лев
890	Слон
1200	Кит

Двоичное дерево поиска



Двоичные деревья

Применение:

1) поиск данных в специально построенных деревьях (базы данных);
2) сортировка данных;
3) вычисление арифметических выражений;
4) кодирование (метод Хаффмана).

Структура узла:

– *key* – ключ узла;
– *left* – указатель на левый дочерний узел;
– *right* – указатель на правый дочерний узел;
– *value* – данные (необходимо только при реализации АТД ассоциативный массив).

Структура узла

C#

```
class BinaryTreeNode<T>  where T : IComparable
    {   private T value;
        private BinaryTreeNode<T> leftChild;
        private BinaryTreeNode<T> rightChild;
        private BinaryTreeNode<T> parent;
        private BinaryTree<T> tree;

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

public T Value{get{ return value;} set{this.value = value;}}
public  BinaryTreeNode<T> LeftChild  {
    get { return leftChild; }  set { leftChild = value; }}
public  BinaryTreeNode<T> RightChild {
    get { return rightChild; } set { rightChild = value; }}
public  BinaryTreeNode<T> Parent  {
       get { return parent; }  set { parent = value; }}
 public  int ChildCount    {
            get   {  int count = 0;
                if (this.LeftChild != null)      count++;
                if (this.RightChild != null)     count++;
                return count;  } }
     }
```

Класс дерева

```
class BinaryTree<T> : ICollection<T>
        where T : IComparable
  {
private BinaryTreeNode<T> root;
private Comparison<IComparable> comparer = CompareElements;
private int size;
private TraversalMode traversalMode = TraversalMode.InOrder;

public BinaryTreeNode<T> Root  {
        get { return root; }  set { root = value; }
/// Получает количество элементов, хранящихся в дереве
public int Count    {   get { return size; }    }
/// Проверка двоичного дерева, пустое или нет
public bool IsEmpty { get { return Root == null; } }

public BinaryTree(){ root = null;  size = 0; }

public static int CompareElements(IComparable x, IComparable y)
        {
            return x.CompareTo(y);
        }
```

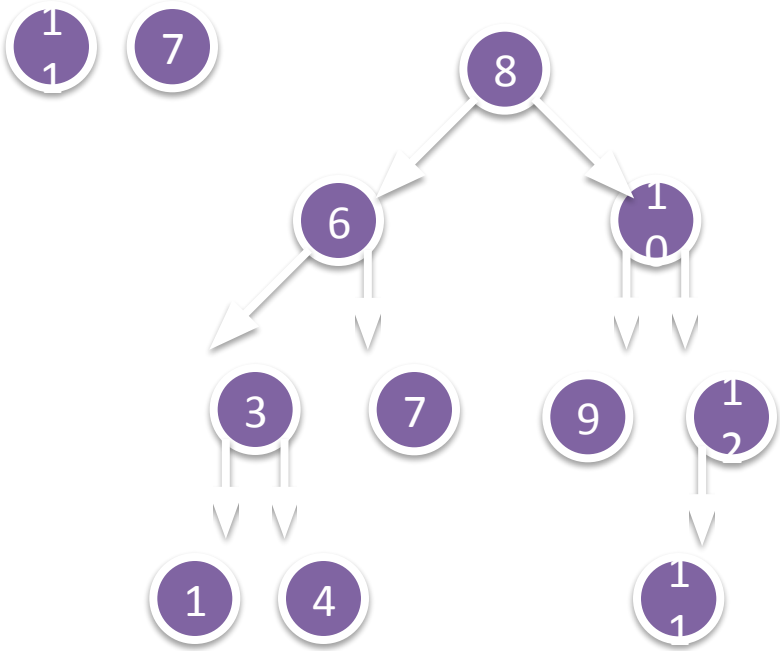
Создание элемента BST

```
public virtual void Add(T value)
{
    BinaryTreeNode<T> node = new BinaryTreeNode<T>(value);
    if (this.root == null) //first element being added
    {
        this.root = node; //set node as root of the tree
        node.Tree = this;
        size++;
    }
    else
    this.Add(node);
}
```

Добавление элемента в BST

```
public void Add(BinaryTreeNode<T> node)
{       if (node.Parent == null) node.Parent = root; //start at head
        if (node.Value.CompareTo(node.Parent.Value)<0) //insert on the left
        {
            if (node.Parent.LeftChild == null)
            {
                node.Parent.LeftChild = node; //insert in left
                size++;
                node.Tree = this; //назначить узел этому двоичному дереву
            }
            else
            {
                node.Parent = node.Parent.LeftChild; //спуск до левого ребенка
                this.Add(node); //recursive call
            }
        }
        else //insert on the right
        {
            if (node.Parent.RightChild == null)
            {
                node.Parent.RightChild = node; //insert in right
                size++;
                node.Tree = this; //assign node to this binary tree
            }
            else
            {
                node.Parent = node.Parent.RightChild;
                this.Add(node);
            }
        }
    }
}
```

Добавление элемента в BST



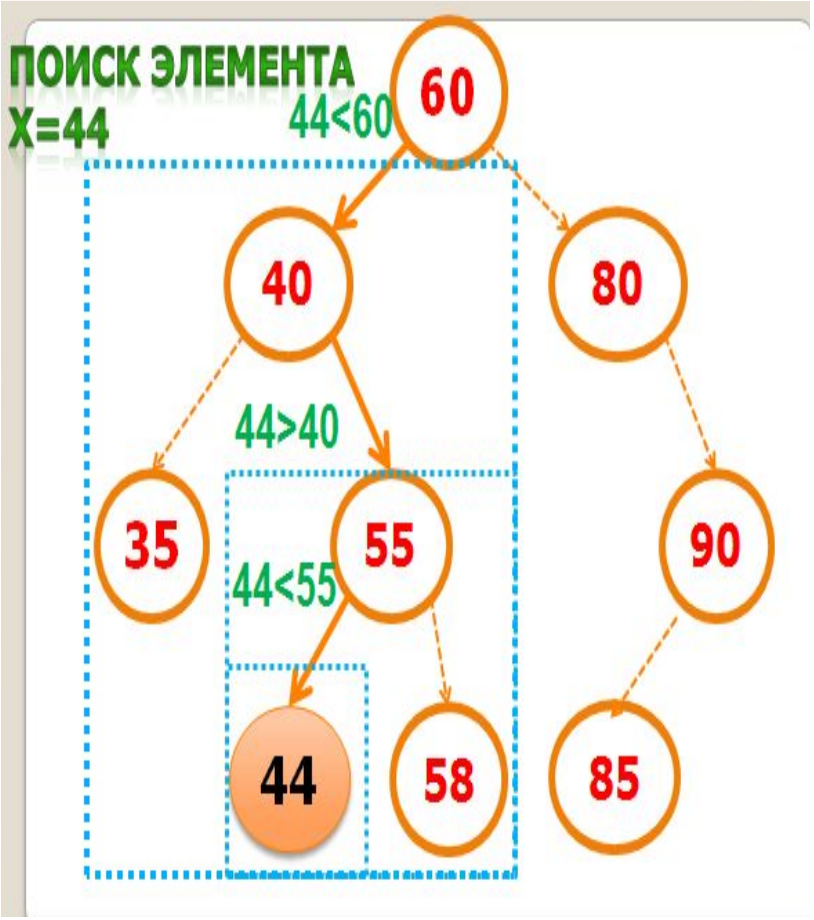

```
public void AddNode(BinaryTreeNode<T> node)
{
    if (node.nodeValue.CompareTo(nodeValue) < 0)
    {
        if (leftNode == null)
        {
            leftNode = node;
        }
        else
        {
            leftNode.AddNode(node);
        }
    }
    else if (node.nodeValue.CompareTo(nodeValue) >= 0)
    {
        if (rightNode == null)
        {
            rightNode = node;
        }
        else
        {
            rightNode.AddNode(node);
        }
    }
}
```

$T_{Add} = O(h)$

Ищем листовой узел (leaf node) для вставки нового элемента

Поиск элемента в BST

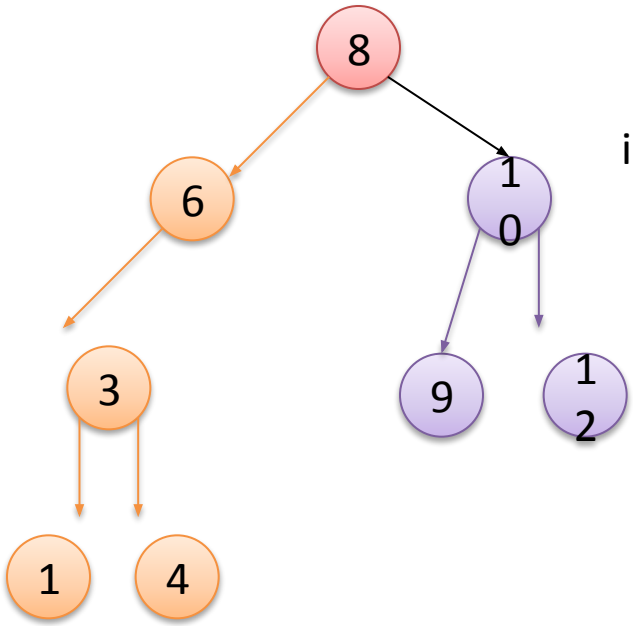
1. Сравниваем ключ корневого узла с искомым. Если совпали, то элемент найден
2. Переходим к левому или правому дочернему узлу и повторяем шаг 1



Возможны рекурсивная и не рекурсивная реализации

Деревья поиска. Индексация и поиск данных.

Ищем ключ 9



Поиск в дереве по ключу

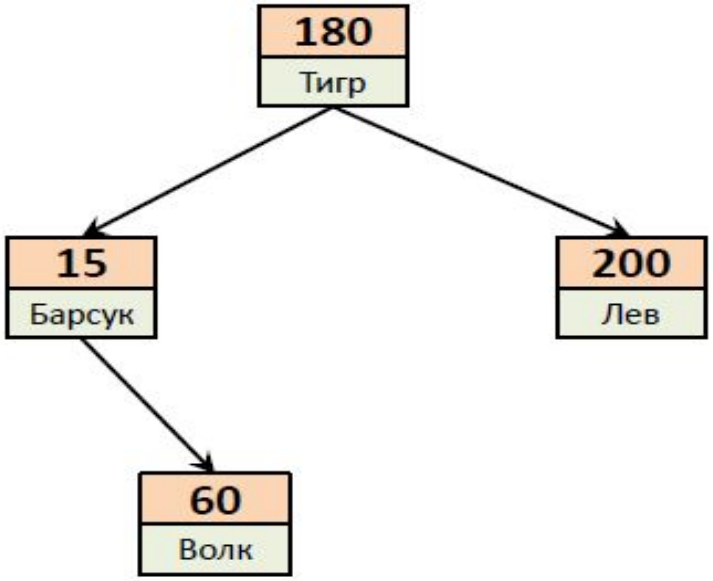
```
public bool Find(T Value) // Поиск узла с заданным ключом
 {
        BinaryTreeNode Iterator = Root;
        while (Iterator != null)
   {
 int Compare =Value.CompareTo(Iterator.Data);
          if (Compare == 0) return true;
          if (Compare < 0) ) // Двигаться налево?
          {
             Iterator = Iterator.Left;
             continue;
          }
          else
          Iterator = Iterator.Right; // Или направо?
     }
        return false;  // Элемент найден
 }
```

Поиск элемента в BST

```
public BinaryTreeNode<T> Find(T value)
{
    BinaryTreeNode<T> node = this.root; //start at root
    while (node != null)
    {
        if (node.Value.Equals(value)) //parameter value found
            return node;
        else
        {   if (value.CompareTo(node.Value) < 0)
                node = node.LeftChild; //search left
            else
                node = node.RightChild; //search right
        }
    }
    return null; //not found
}
/// Возвращает, хранится ли значение в дереве
    ссылка: 1
    public bool Contains(T value)
{
    return (this.Find(value) != null);
}
```

Поиск минимального элемента в BST

- Минимальный элемент всегда расположен в левом поддереве корневого узла
- Требуется найти самого левого потомка корневого узла

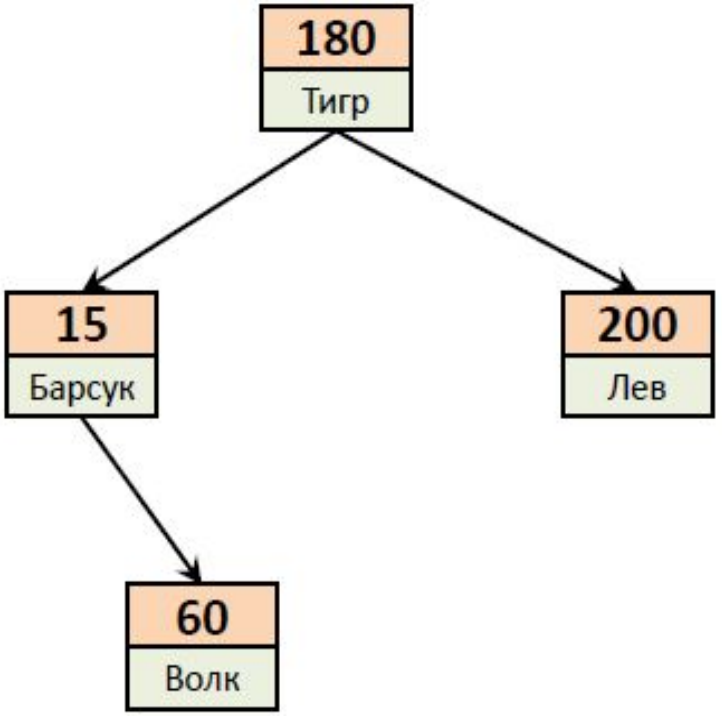



```
public BinaryTreeNode<T> minimum()    {
BinaryTreeNode current, last;
current = root; // Обход начинается с корневого узла
while(current != null)  {
last = current;
current = current.leftChild; // Переход к левому потомку
}
return last;}
```

$T_{Min} = O(h)$

Поиск максимального элемента в BST

- Максимальный элемент всегда расположен в правом поддереве корневого узла
- Требуется найти самого правого потомка корневого узла

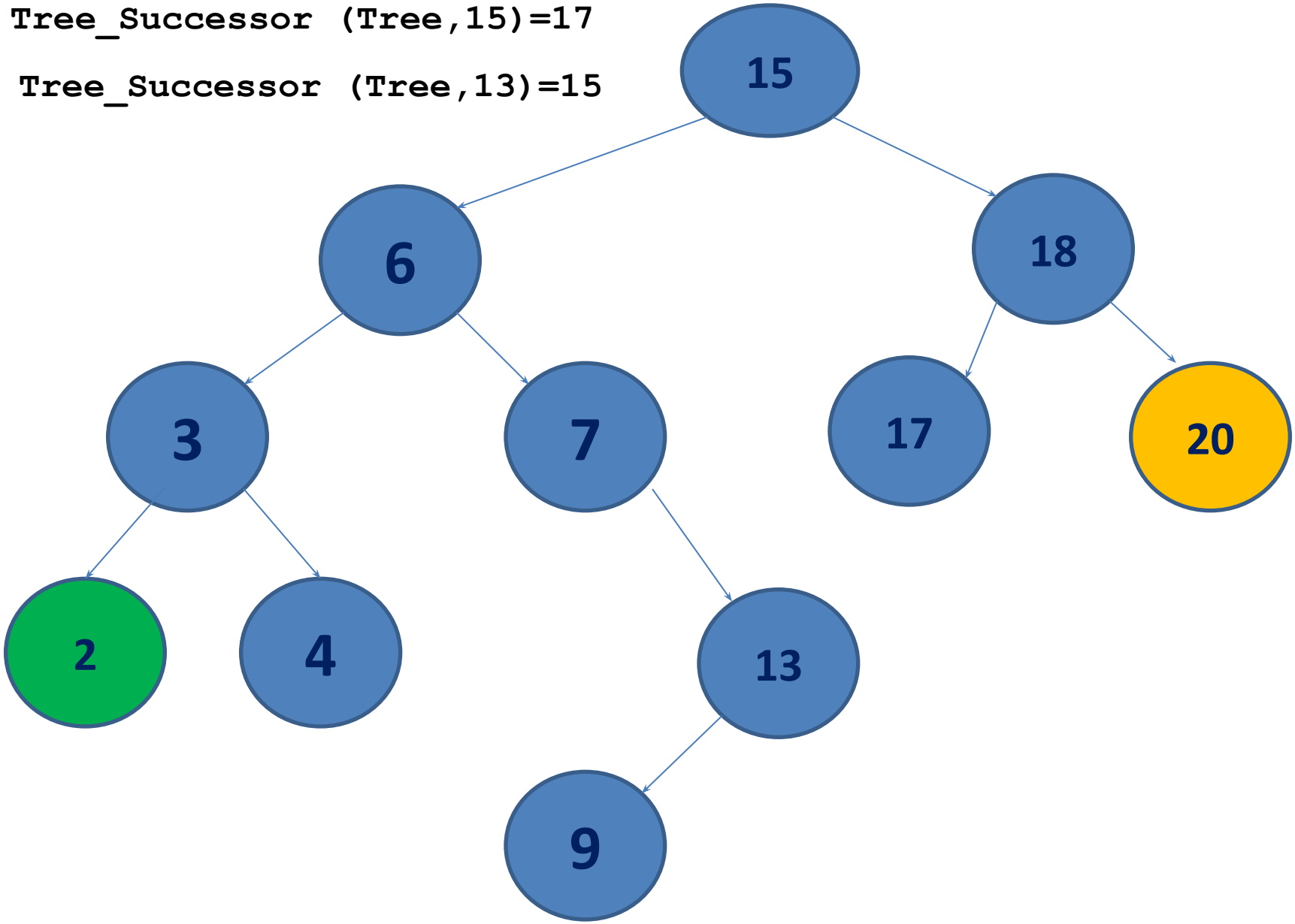


$T_{Max} = O(h)$

Поиск следующего элемента

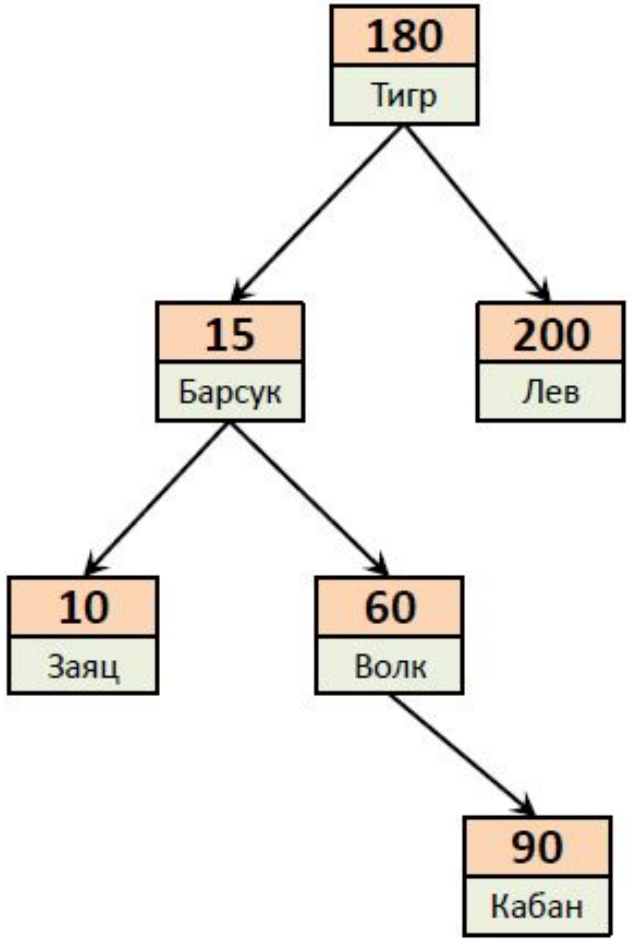
```
Tree_Successor (Tree,15)=17

Tree_Successor (Tree,13)=15
```



Удаление элемента из BST

1. Находим узел *z* с заданным ключом – $O(n)$
2. Возможны 3 ситуации:
   - узел *z* не имеет дочерних узлов
   - узел *z* имеет 1 дочерний узел
   - узел *z* имеет 2 дочерних узла



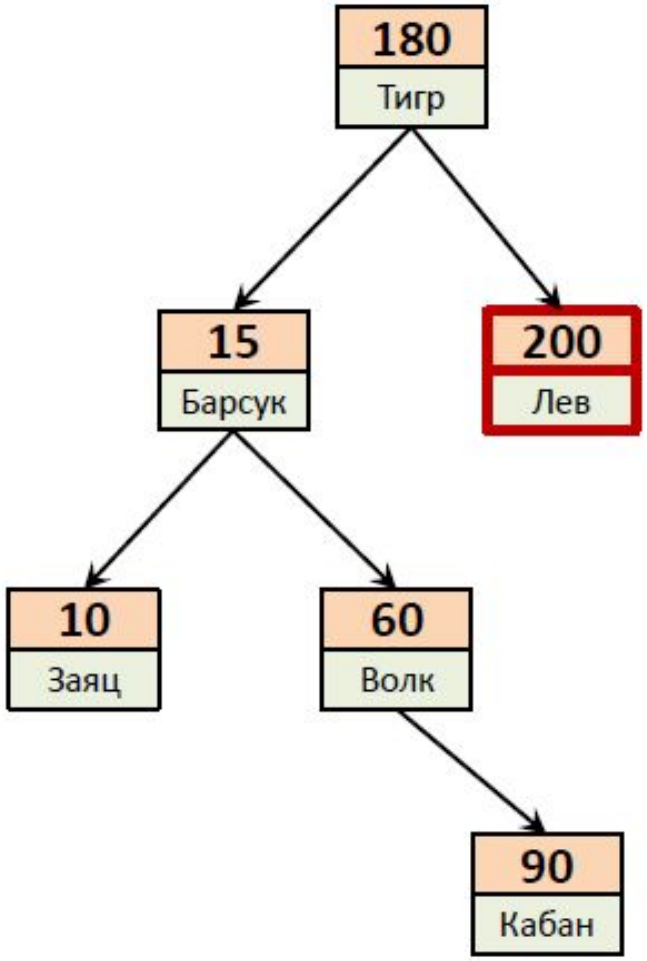
Удаление элемента из BST

Удаление узла **“Лев”** (случай 1)

1. Находим и удаляем узел “Лев” из памяти (free)

2. Родительский указатель (left или right) устанавливаем в значение NULL

   “Тигр”->right = NULL

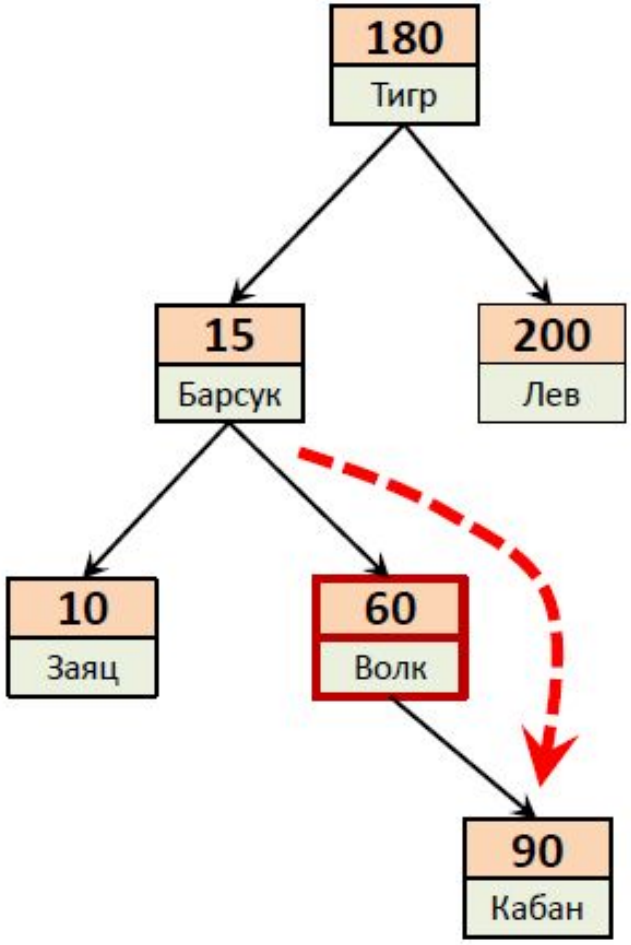


Удаление элемента из BST

Удаление узла **“Волк”** (случай 2)

1. Находим узел “Волк”
2. Родительский указатель узла “Волк” (left или right) устанавливаем на его дочерний элемент
3. Удаляем узел “Волк” из памяти

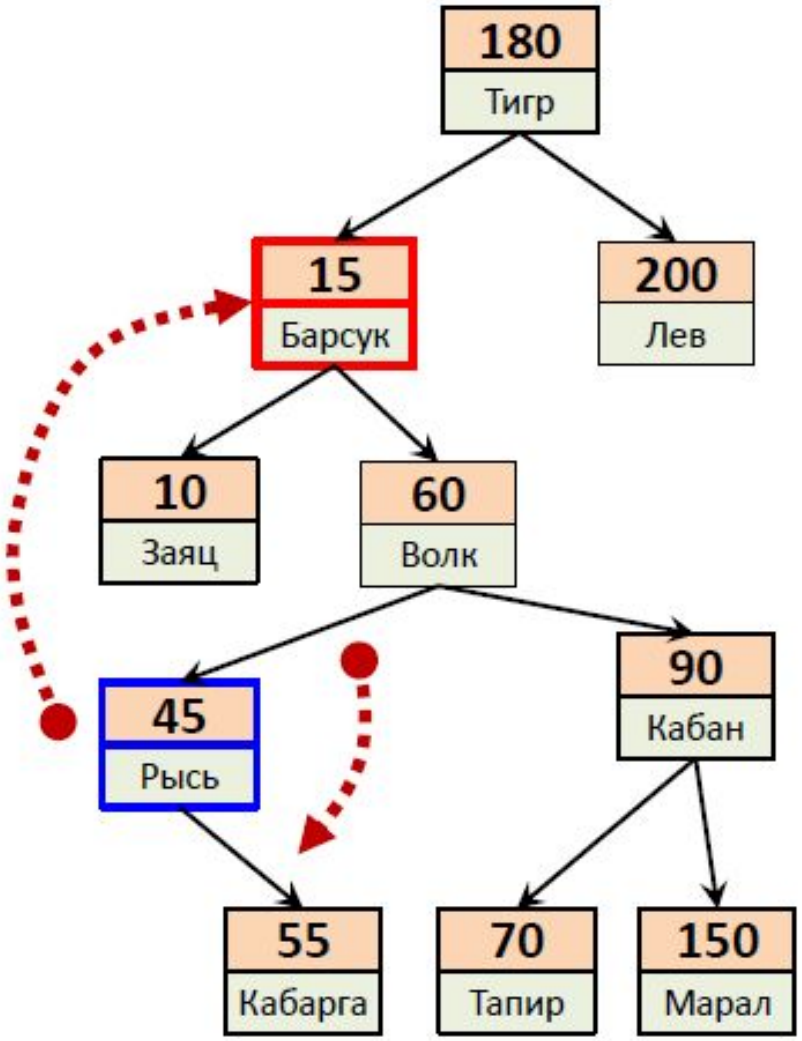
“Барсук”->right = “Волк”->right



Удаление элемента из BST

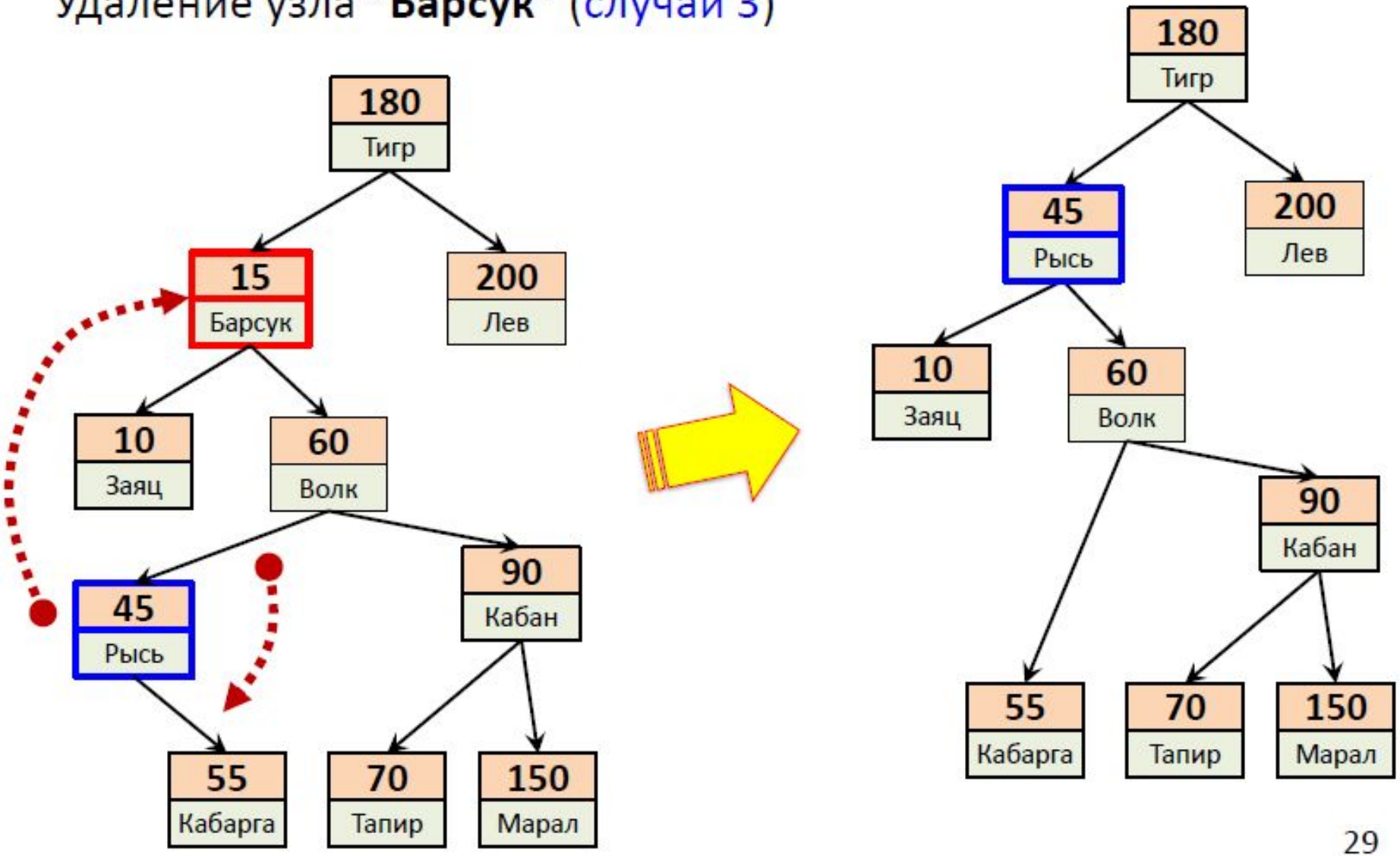
Удаление узла **“Барсук”** (случай 3)

1. Находим узел “Барсук”
2. Находим узел с минимальным ключом в правом поддереве узла “Барсук” – **самый левый лист в поддереве** (узел “Рысь”)
3. Заменяем узел “Барсук” узлом “Рысь”



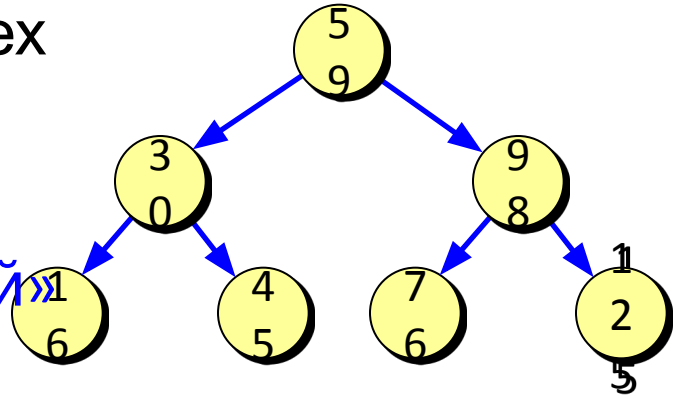
Удаление элемента из BST

Удаление узла **“Барсук”** (случай 3)

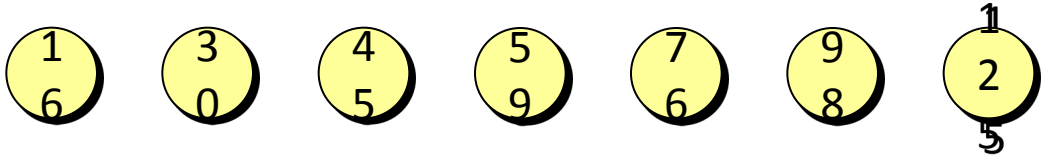


Обход дерева

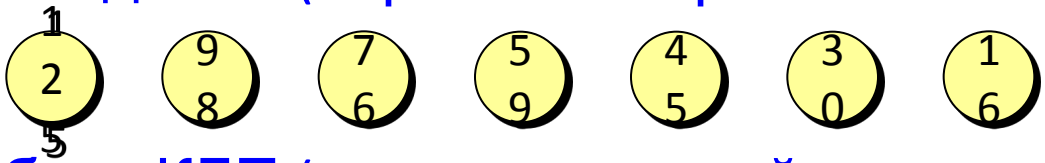
Обход дерева – это перечисление всех узлов в определенном порядке.



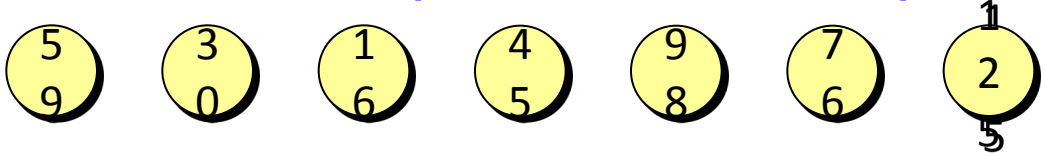
Обход ЛКП («левый – корень – правый» InOrderTraversal):



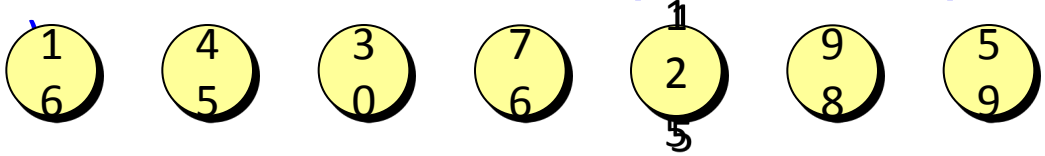
Обход ПКЛ («правый – корень – левый»):



Обход КЛП («корень – левый – правый» PreOrderTraversal):



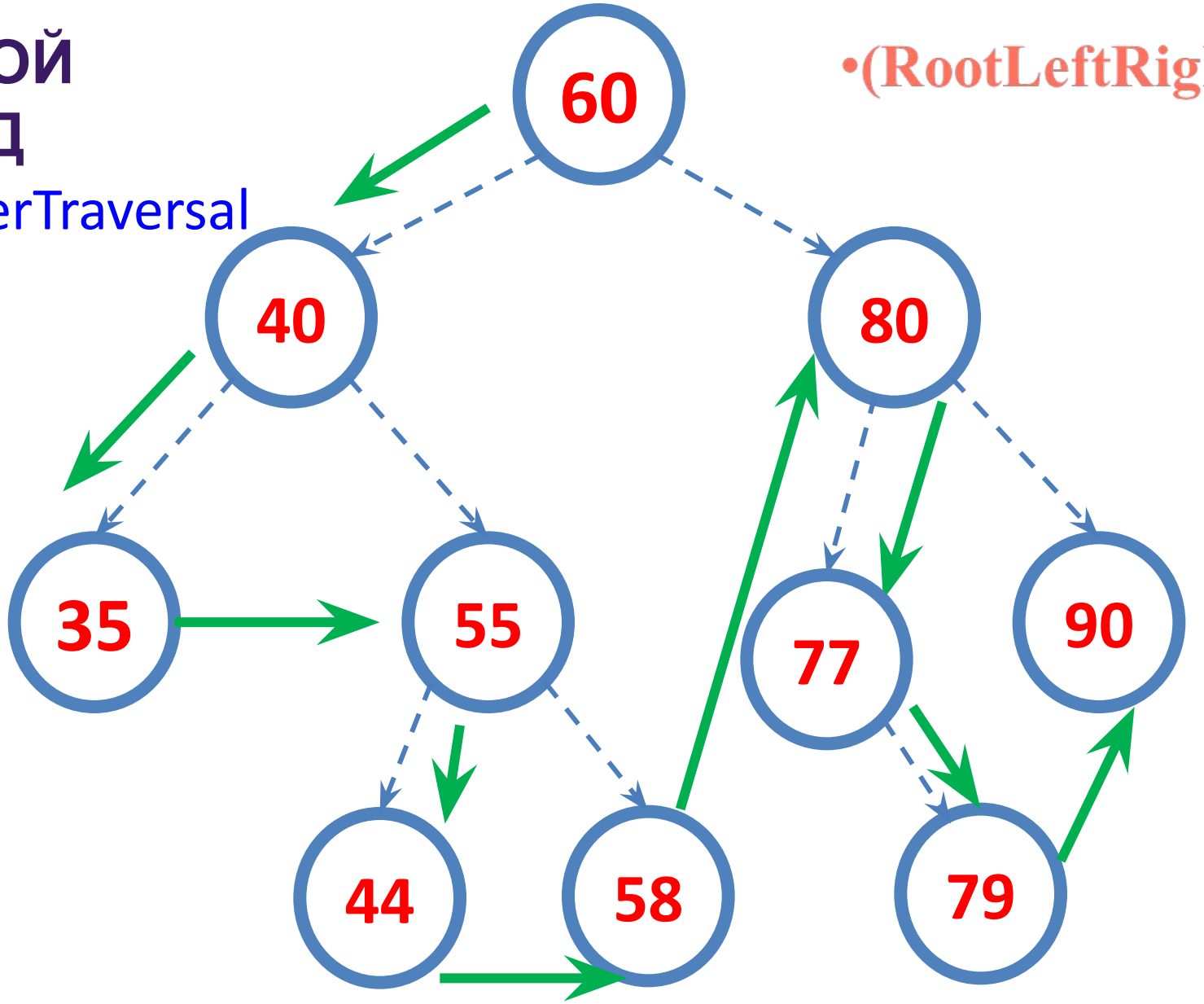
Обход ЛПК («левый – правый – корень» PostOrderTraversal):



ПРЯМОЙ ОБХОД

PreOrderTraversal

•(RootLeftRight)

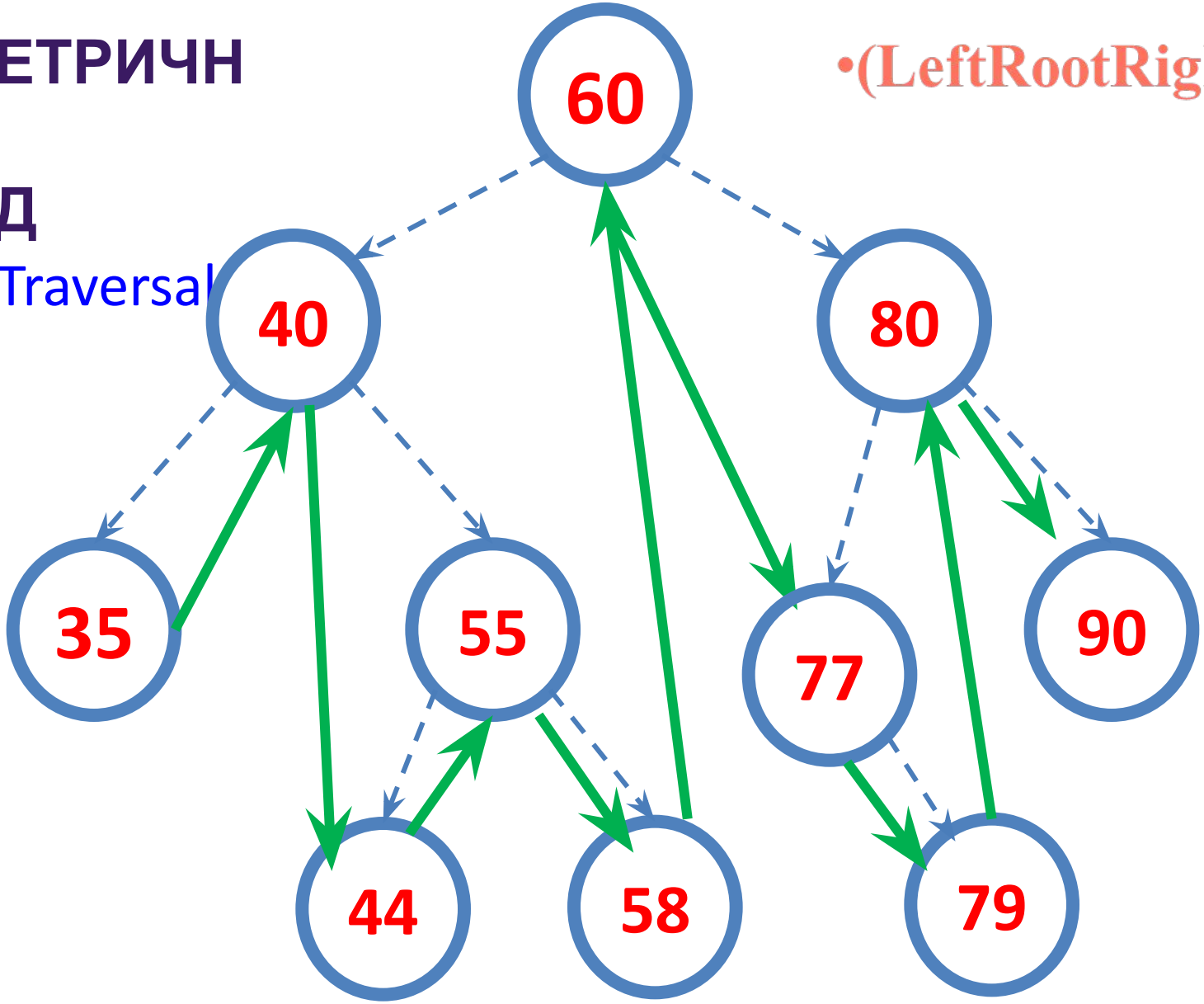


60-40-35-55-44-58-80-77-79-90

СИММЕТРИЧНЫЙ ОБХОД

InOrderTraversal

•(LeftRootRight)

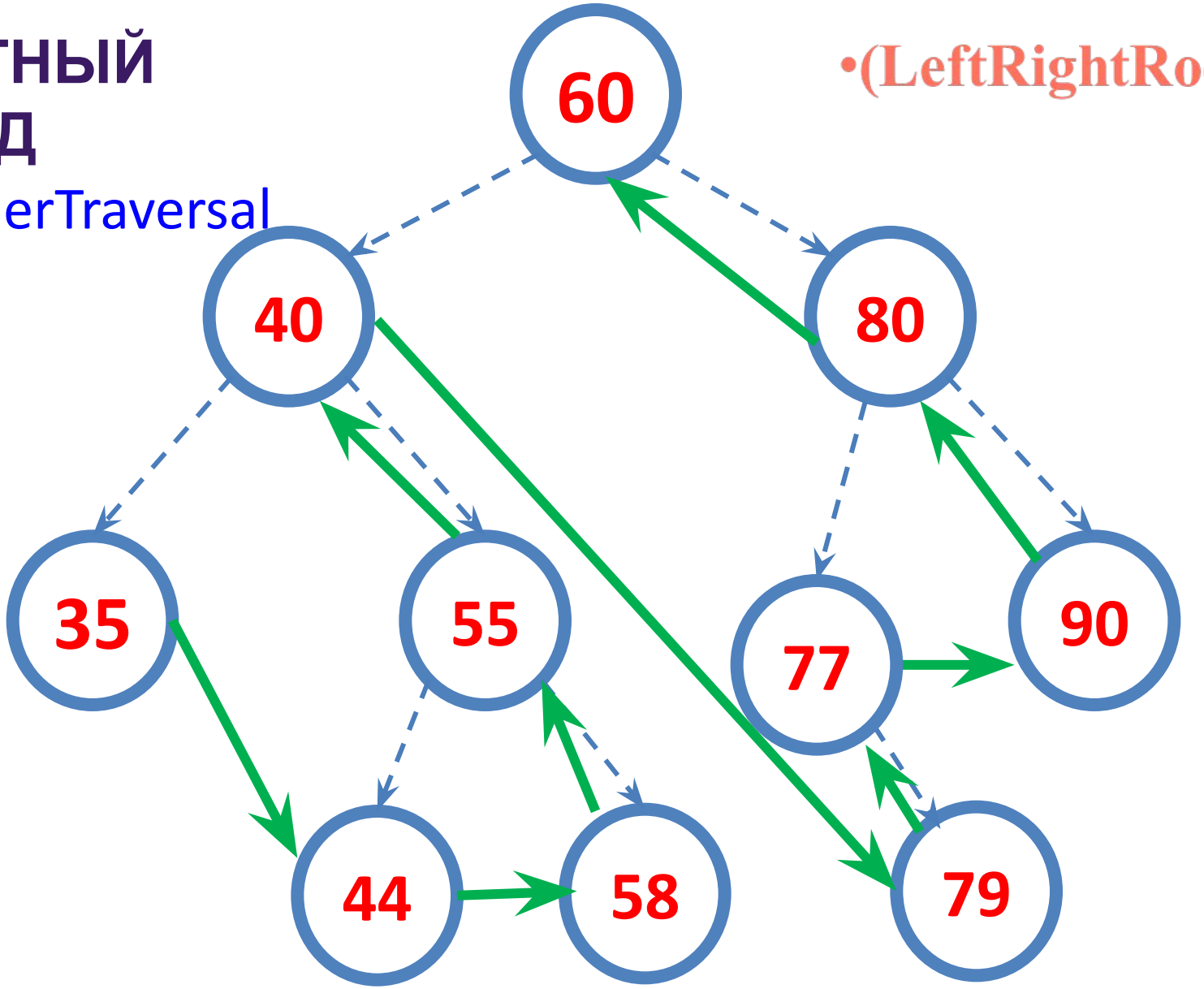


35-40-44-55-58-60-77-79-80-90

ОБРАТНЫЙ ОБХОД

PostOrderTraversal

•(LeftRightRoot)



35-44-58-55-40-79-77-90-80-60

Обход в ширину производится с помощью очереди. Первоначально в очередь помещается корень, затем, пока очередь не пуста, выполняются следующие действия:

- Из очереди выталкивается очередной узел;
- Этот узел обрабатывается;
- В очередь добавляются оба сына этого узла.

60-40-80-35-55-77-90-44-58-79

Обход дерева – реализация

```
//-----------------------------------------------
// Функция LKP - обход дерева в порядке ЛКП
//                (левый - корень - правый)
// Вход: Tree - адрес корня
//-----------------------------------------------
void LKP( PNode Tree )
{
if ( ! Tree ) return;
LKP( Tree->left );
printf( "%d ", Tree->data );
LKP( Tree->right );
}
```

обход этой ветки закончен

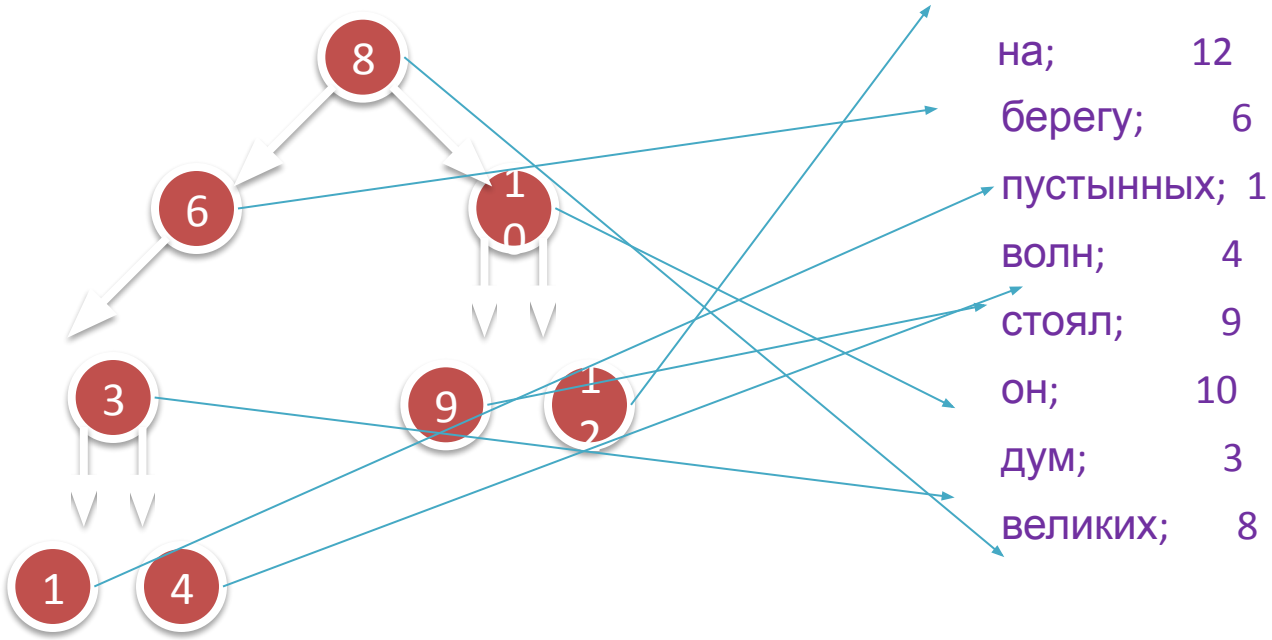
обход левого поддерева

вывод данных корня

обход правого поддерева

Для рекурсивной структуры удобно применять рекурсивную обработку!

Индексация данных



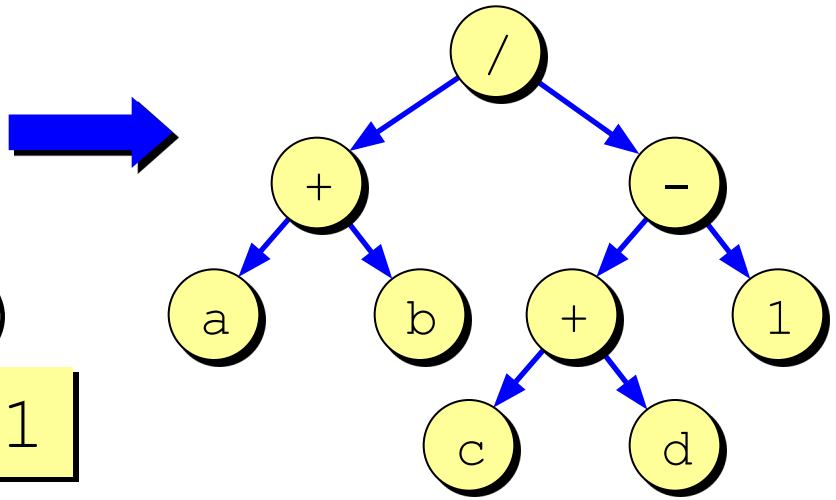
С помощью поиска по индексу можно получить ответы на вопросы:

- Какое слово встречается ровно 6 раз?
- Какие слова встречаются больше 10 раз?
- Какое слово встречается чаще всего?

Разбор арифметических выражений

Как вычислять автоматически:

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



Инфиксная запись, обход ЛКП

(знак операции между операндами)

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

необходимы скобки!

Префиксная запись, КЛП (знак операции до операндов)

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

польская нотация,
Jan Łukasiewicz (1920)

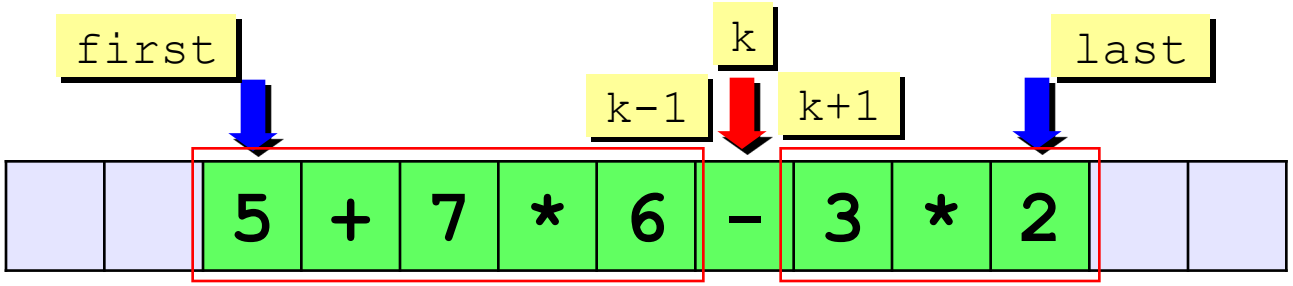
скобки не нужны, можно однозначно вычислить!

Постфиксная запись, ЛПК (знак операции после операндов)

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

обратная польская нотация,
F. L. Bauer and E. W. Dijkstra

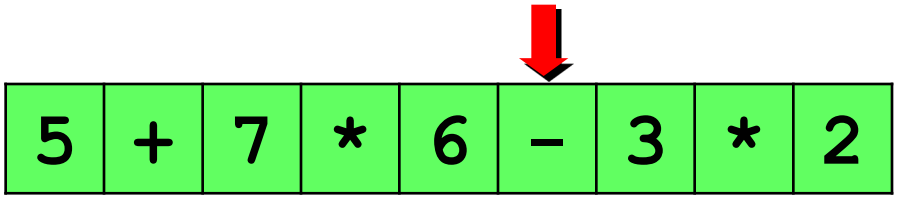
Построение дерева



Алгоритм:

1) если `first=last` (остался один символ – число), то создать новый узел и записать в него этот элемент; иначе...
2) среди элементов от `first` до `last` включительно найти последнюю операцию (элемент с номером `k`);
3) создать новый узел (корень) и записать в него знак операции;
4) рекурсивно применить этот алгоритм два раза:
   - построить левое поддерево, разобрав выражение из элементов массива с номерами от `first` до `k-1`;
   - построить правое поддерево, разобрав выражение из элементов массива с номерами от `k+1` до `last`.

Как найти последнюю операцию?



Порядок выполнения операций

- умножение и деление;
- сложение и вычитание.

Приоритет (старшинство) – число, определяющее последовательность выполнения операций: раньше выполняются операции с большим приоритетом:

- умножение и деление (приоритет 2);
- сложение и вычитание (приоритет 1).

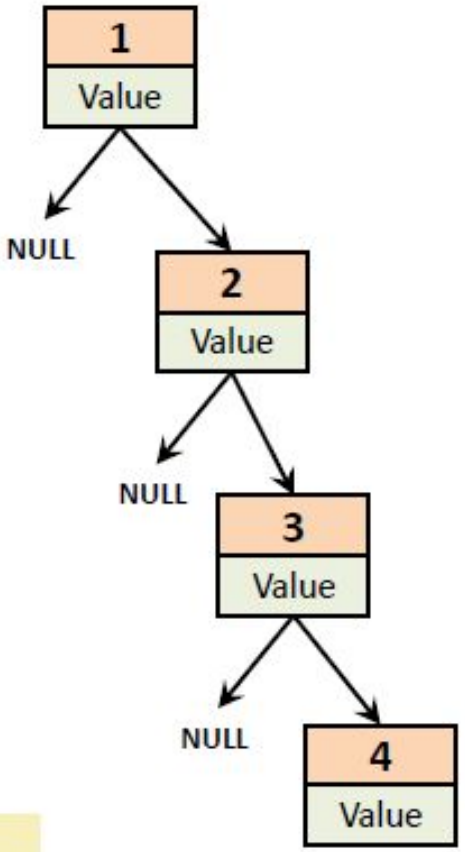
Нужно искать последнюю операцию с наименьшим приоритетом!

Анализ эффективность BST

1. Операции имеют трудоемкость пропорциональную высоте *h* дерева

2. В **худшем** случае высота дерева ***O*(*n*)** (вставка элементов в отсортированной последовательности)

3. В **среднем** случае высота дерева ***O*(log*n*)**




```
bstree_add(tree, “Item”, 1);
bstree_add(tree, “Item”, 2);
bstree_add(tree, “Item”, 3);
bstree_add(tree, “Item”, 4);
```

Дерево вырождается в связный список

Реализация словаря на основе BST

Операция	Средний случай (average case)	Худший случай (worst case)
Add (*map, key, value*)	$O(\log n)$	$\boldsymbol{O(n)}$
Lookup (*map, key*)	$O(\log n)$	$\boldsymbol{O(n)}$
Remove (*map, key*)	$O(\log n)$	$\boldsymbol{O(n)}$
Min(*map*)	$O(\log n)$	$\boldsymbol{O(n)}$
Max(*map*)	$O(\log n)$	$\boldsymbol{O(n)}$

Сбалансированные деревья поиска

- **Сбалансированное по высоте дерево поиска (self-balancing binary search tree)** – дерево поиска, в котором высоты поддеревьев узла различаются не более чем на заданную константу *k*
- Баланс высоты поддерживается при выполнении операций добавления и удаления элементов
- Типы сбалансированных деревьев поиска:
  - Красно-черные деревья (Red-black tree): $h \leq 2\log_2(n+1)$
  - АВЛ-деревья (AVL-tree): $h < 1.4405 \cdot \log_2(n+2) - 0.3277$
  - B-деревья
  - Деревья Ван Эмде Боаса
  - ...

Все операции на красно-черном дереве и АВЛ-дереве в худшем случае выполняются за время *O*(log*n*)

Таблица 10.3. *Интерфейсы коллекций*

Интерфейс	Описание	Методы
`ICollection<T>`	Основные методы, имеющие отношение к большинству типов коллекций. Наследует от интерфейса `IEnumerable<T>`	`Add`, `Clear`, `Contains`, `CopyTo`, `Remove`
`IEnumerable<T>`	Указывает, что объекты коллекции можно перебрать с помощью конструкции `foreach`	`GetEnumerator`
`IDictionary<TKey, TValue>`	Позволяет осуществлять доступ к элементам коллекции через пары "ключ/значение". Наследует от интерфейса `ICollection<KeyValuePair<TKey, TValue>>`	`ContainsKey`, `TryGetValue`
`IList<T>`	Указывает, что к элементам коллекции можно обращаться по индексу. Наследует от интерфейса `ICollection<T>`	`IndexOf`, `Insert`, `RemoveAt`