КАЛУЖСКИЙ ФИЛИАЛ
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО БЮДЖЕТНОГО
ОБРАЗОВАТЕЛЬНОГО УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ Н.Э. БАУМАНА
(национальный исследовательский университет)»

Факультет «Информатика и управление»

Кафедра «Программное обеспечение ЭВМ, информационные технологии»

Типы и структуры данных

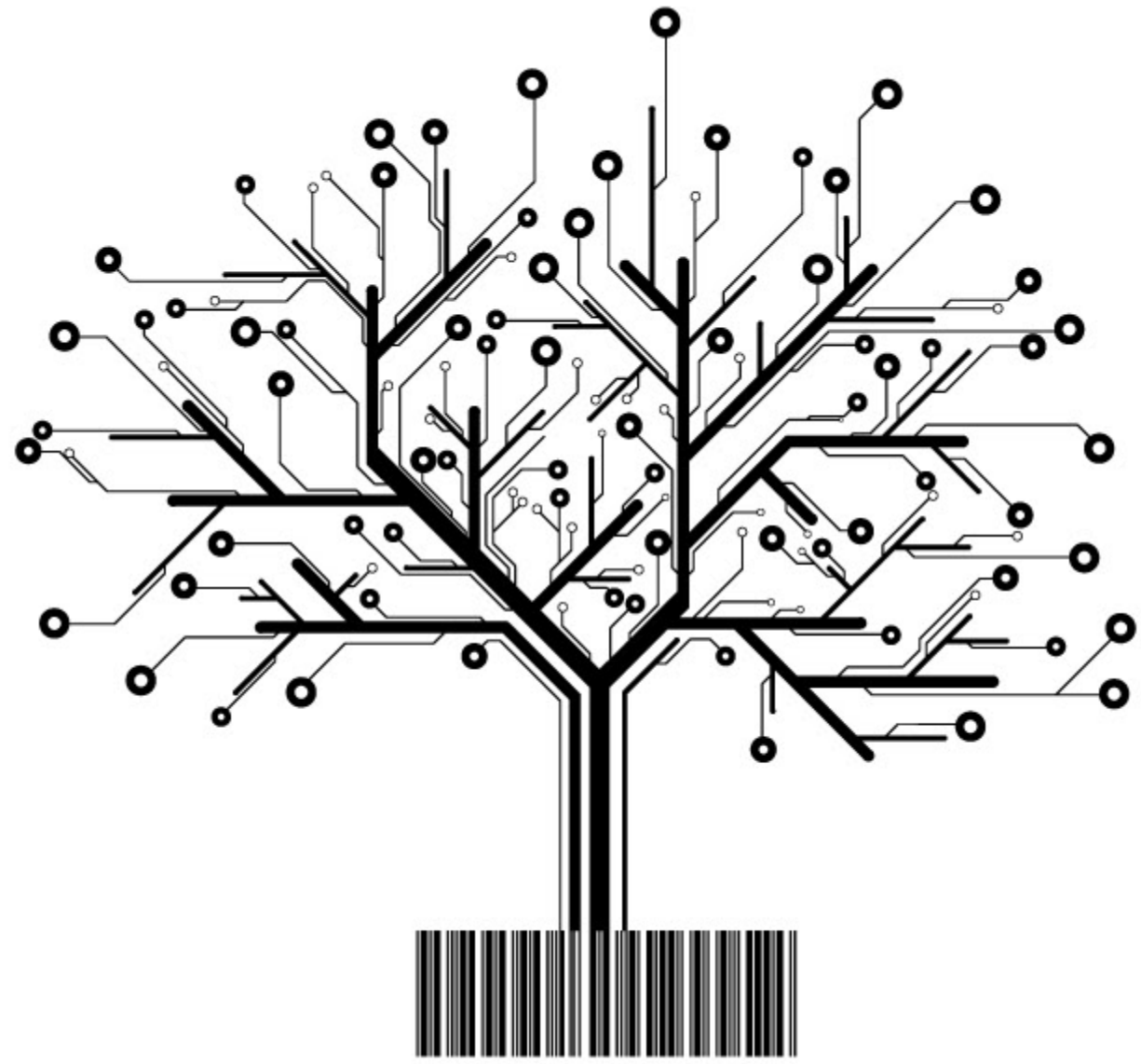
Лекция №4-5. «Бинарные деревья»

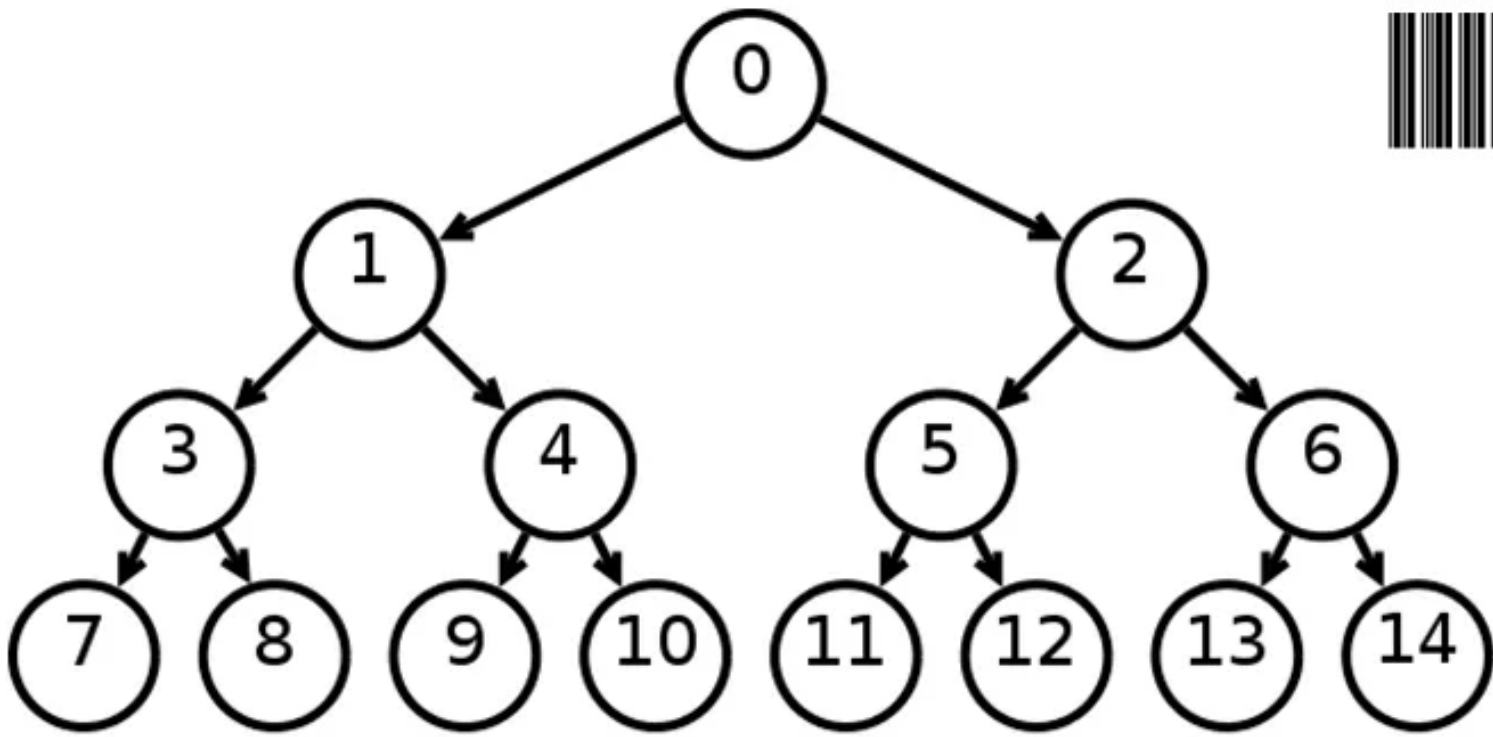
Калуга - 2022

Общие сведения

- **Дерево** – это структура данных, представляющая собой совокупность элементов и отношений, образующих иерархическую структуру этих элементов (рис. 1).
- Каждый элемент дерева называется *вершиной (узлом) дерева*.
- Вершины дерева соединены направленными дугами, которые называют *ветвями дерева*.
- Начальный узел дерева называют *корнем дерева*, ему соответствует нулевой уровень.
- *Листьями дерева* называют вершины, в которые входит одна ветвь и не выходит ни одной ветви.

Деревья



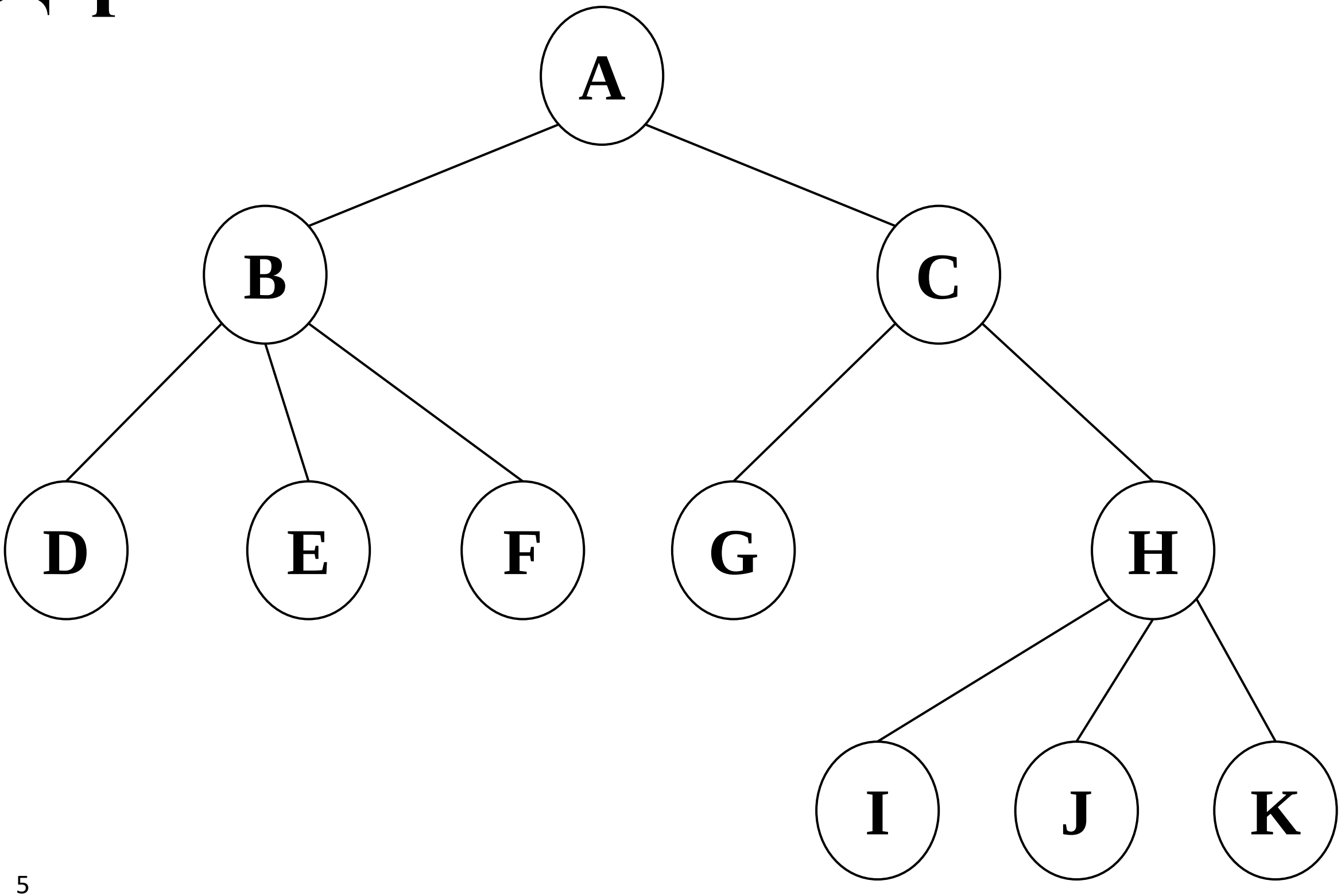


Общие сведения

Каждое дерево обладает следующими ***свойствами***:

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

Дерево



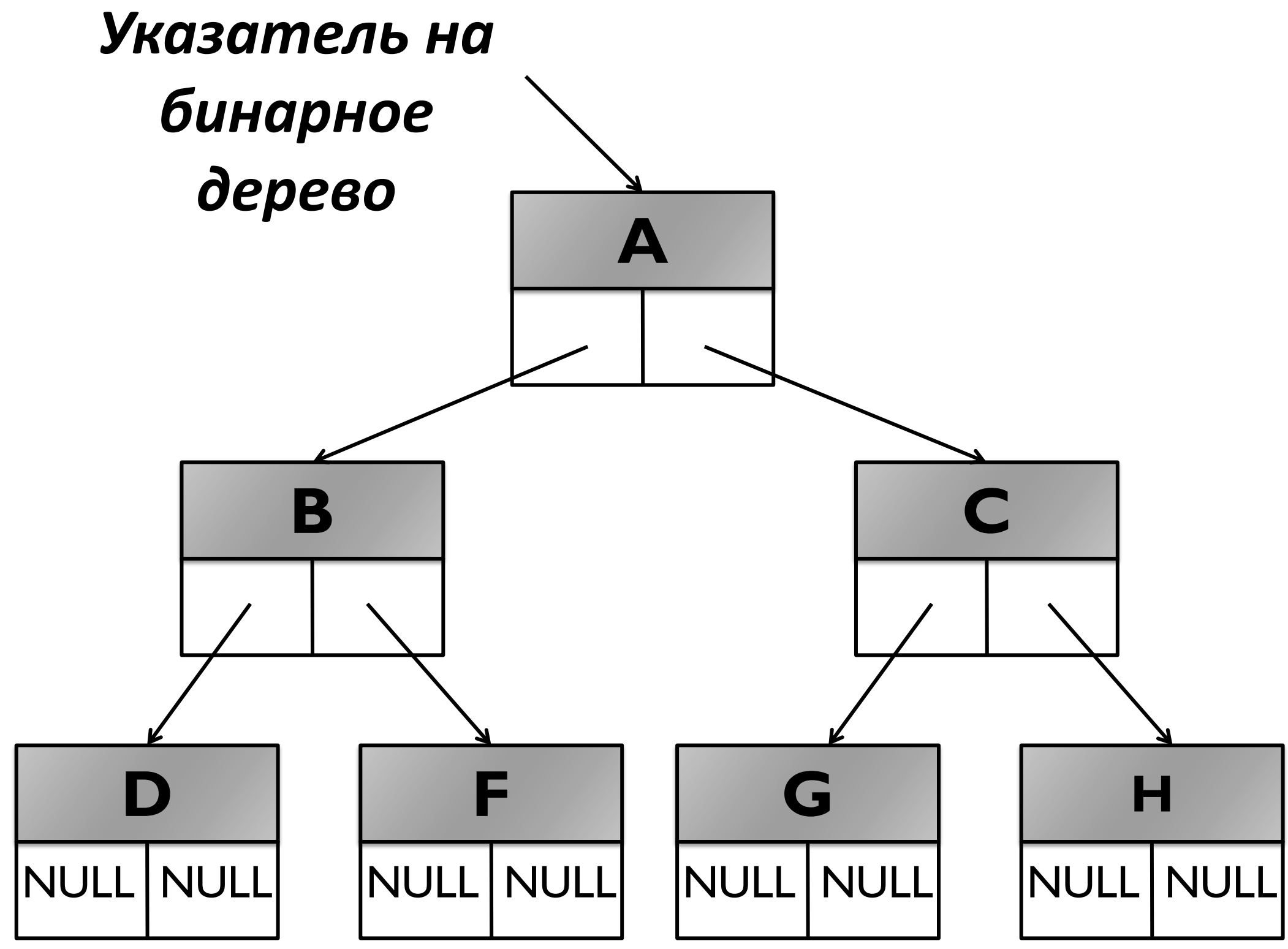
Классификация деревьев

- по максимальному количеству потомков у одного узла:
  - бинарные (2 потомка);
  - тернарные (3 потомка);
  - М-арные (М-потомков);
- по структуре:
  - симметричные;
  - сбалансированные;
  - несбаллансированные;
  - полные;
  - вырожденные;
- по характеру данных:
  - с упорядоченными данными;
  - с неупорядоченными данными.

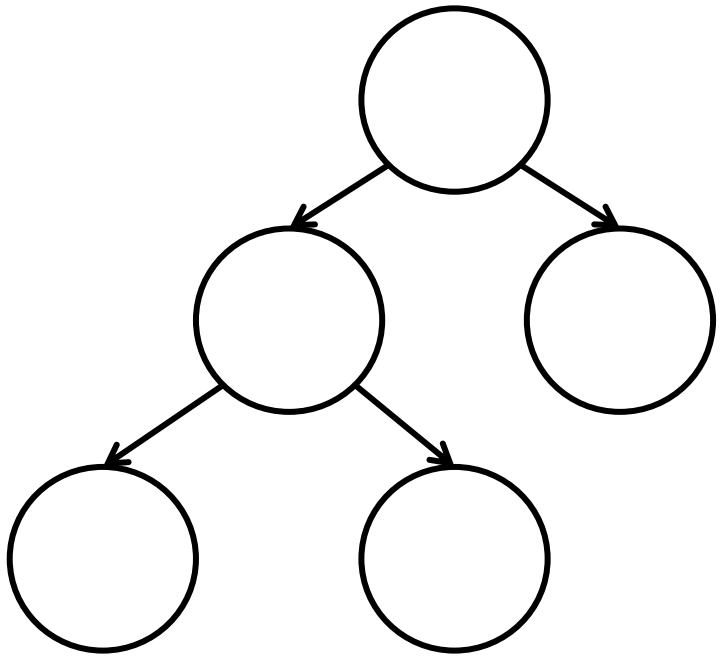
Виды деревьев (примеры)

- Бинарные деревья поиска;
- АВЛ-деревья;
- Красно-черные деревья;
- Сильноветвящиеся деревья (B-деревья);
- Деревья выражений (дерево синтаксического разбора);
- Деревья отрезков.

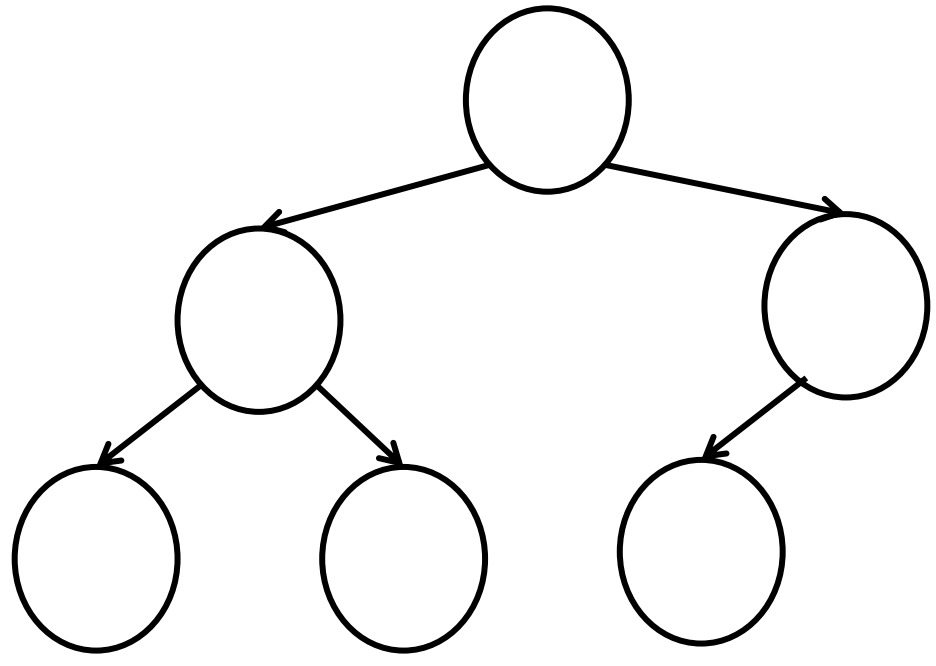
Организация бинарного дерева



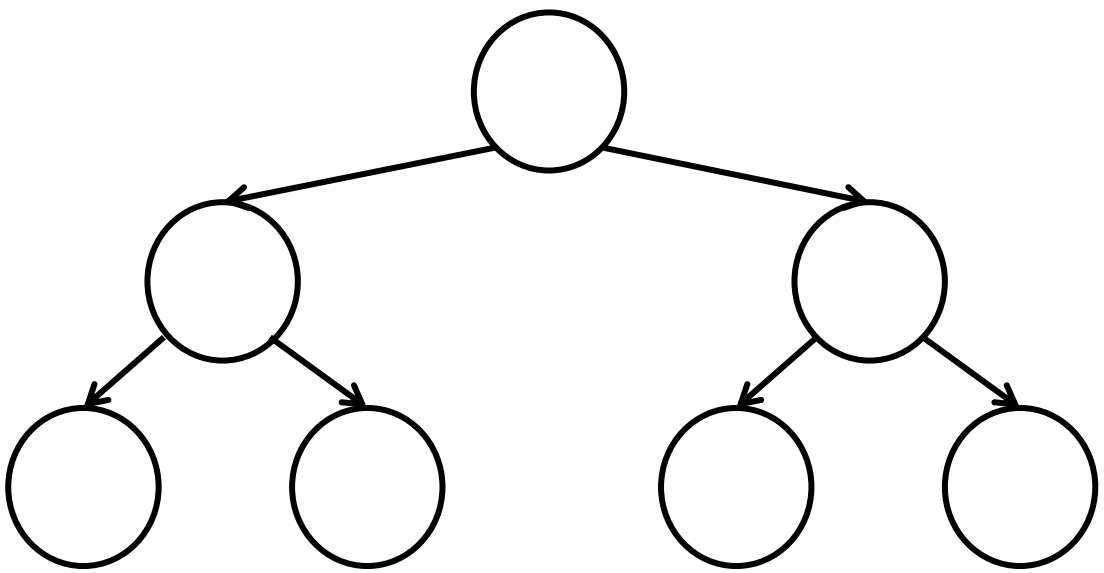
Виды бинарных деревьев



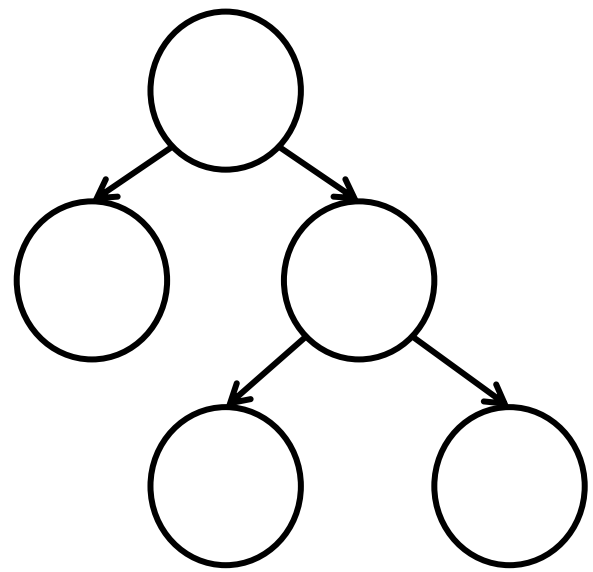
Строгое



Нестрогое

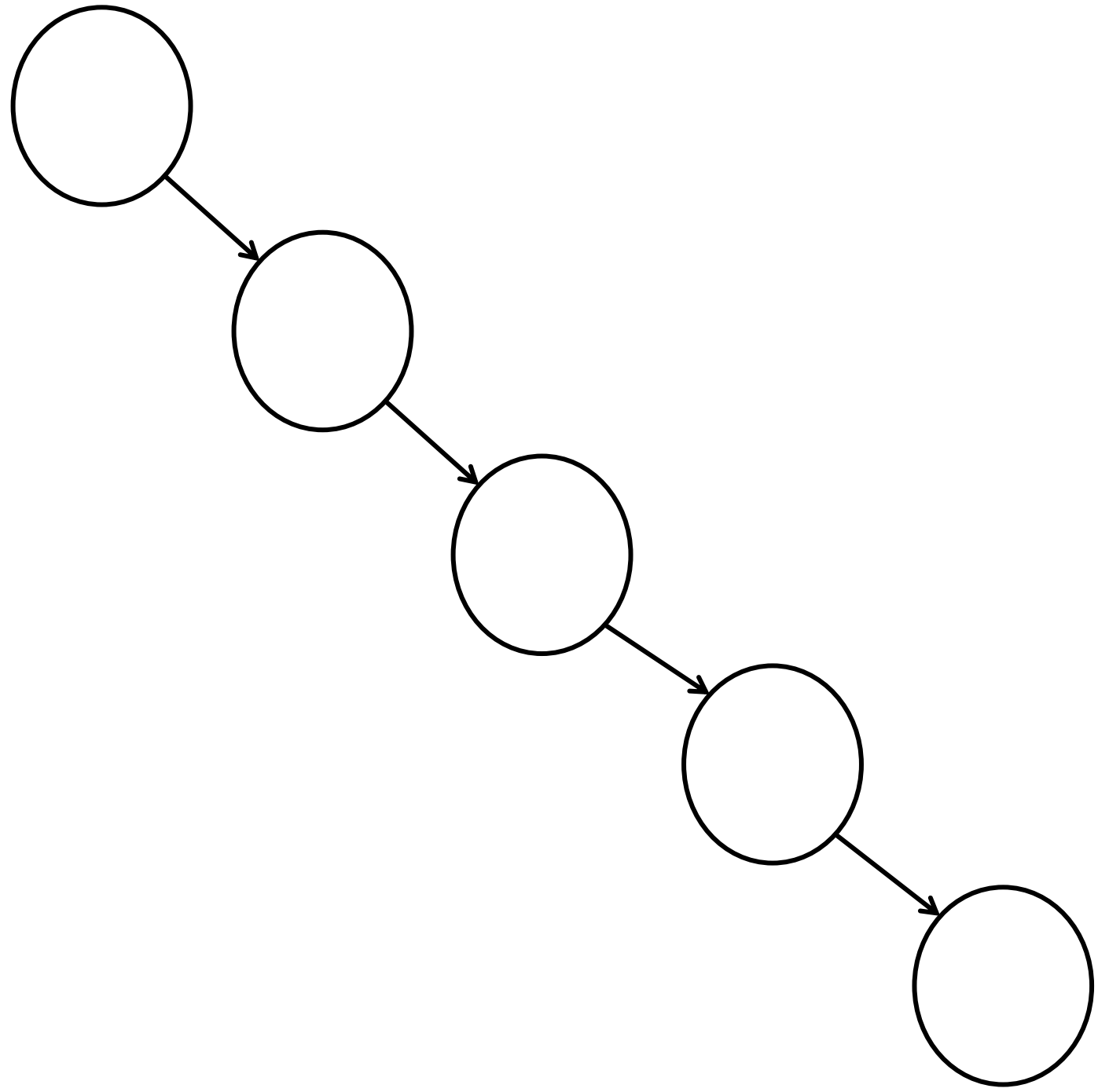


Полное



Неполное

Список как частный случай бинарного дерева



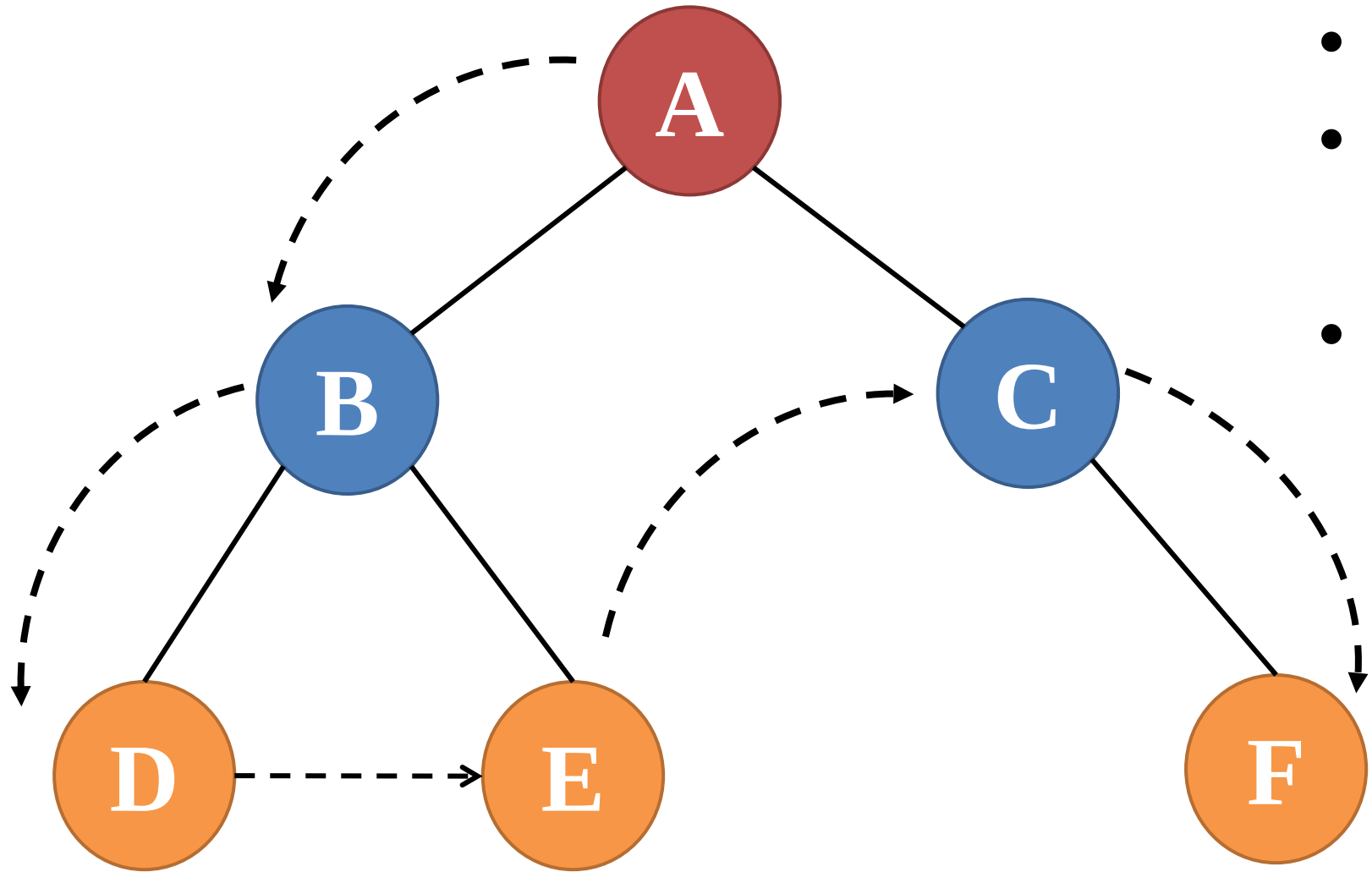
Основные операции, применяемые к бинарным деревьям

- создание *бинарного дерева*;
- печать *бинарного дерева*;
- обход *бинарного дерева*;
- вставка элемента в *бинарное дерево*;
- удаление элемента из *бинарного дерева*;
- проверка пустоты *бинарного дерева*;
- удаление *бинарного дерева*.

Обходы деревьев

- *прямой* – сначала посещается корень, затем в прямом порядке узлы левого поддерева, далее все узлы правого поддерева;
- *симметричный* – сначала в симметричном порядке посещаются все узлы левого поддерева, затем корень, после чего в симметричном порядке все узлы правого поддерева;
- *обратный* – сначала посещаются в обратном порядке все узлы левого поддерева, затем в обратном порядке узлы правого поддерева, последним посещается корень.

Прямой обход (префиксный обход)

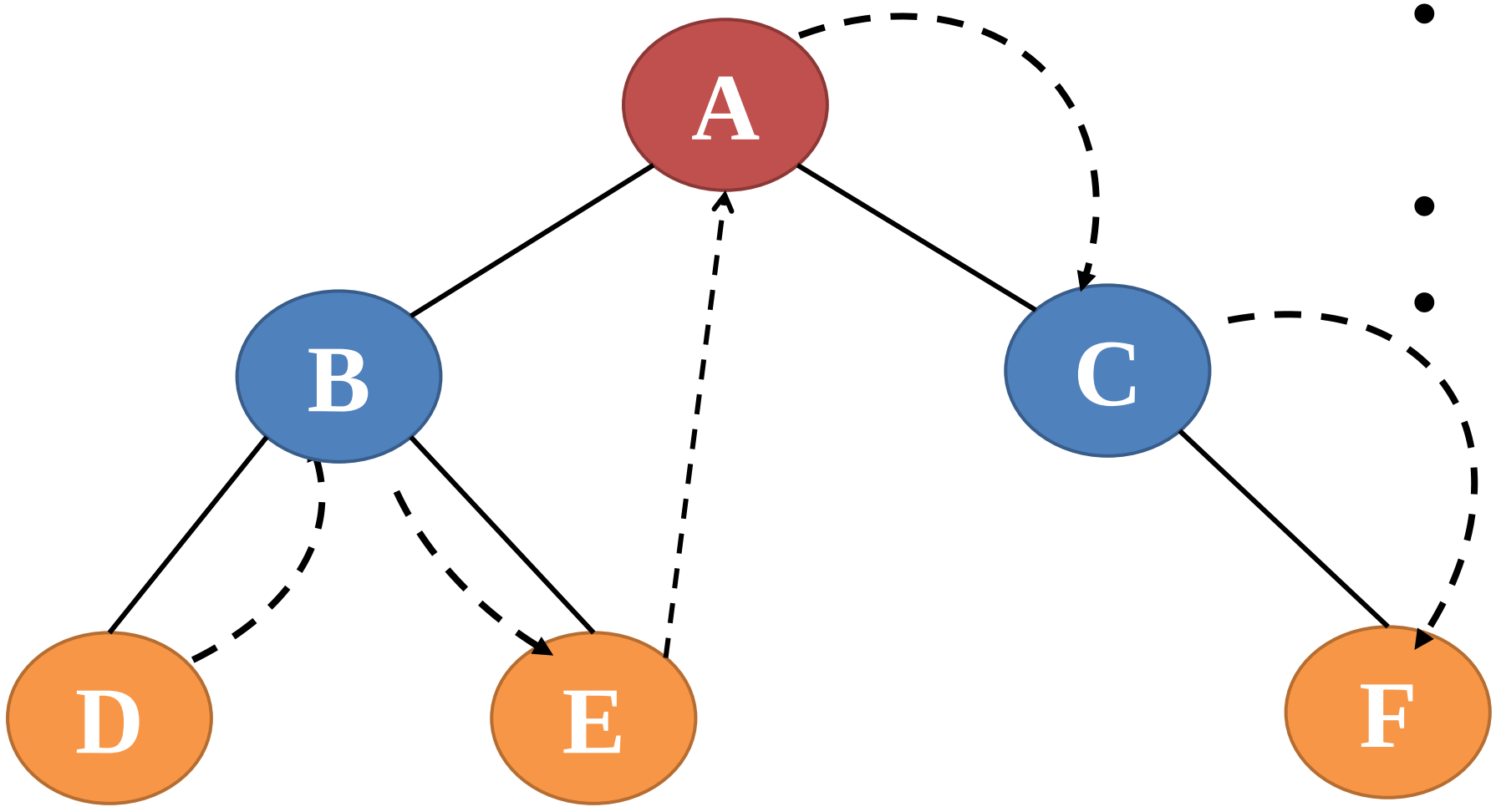


A B D E C F

Рекурсивная процедура:

- Посетить узел
- Обойти левое поддерево
- Обойти правое поддерево

Симметричный обход (инфиксный обход)

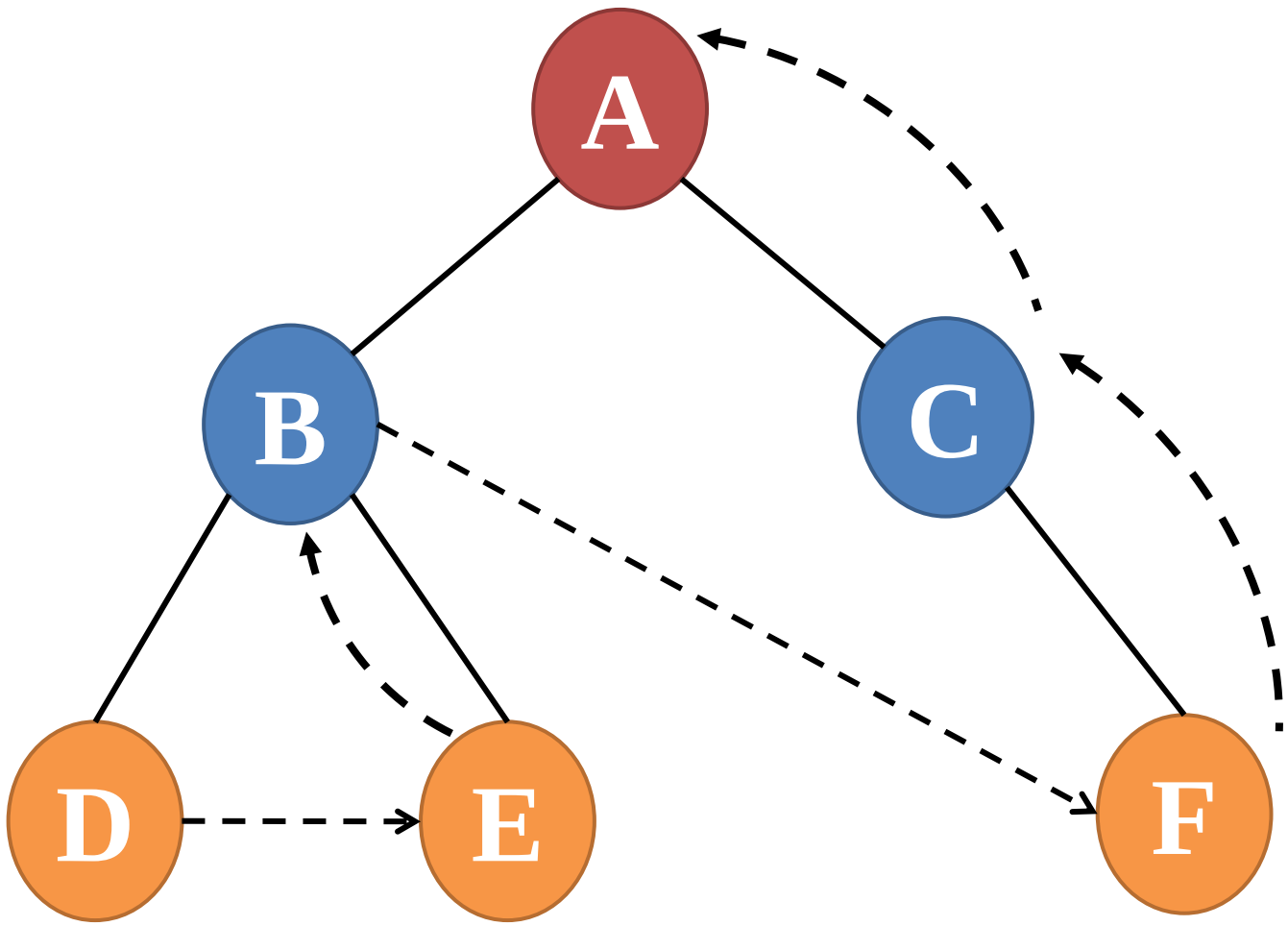


D B E A C F

Рекурсивная процедура:

- Обойти левое поддерево
- Посетить узел
- Обойти правое поддерево

Обратный обход (постфиксный обход)

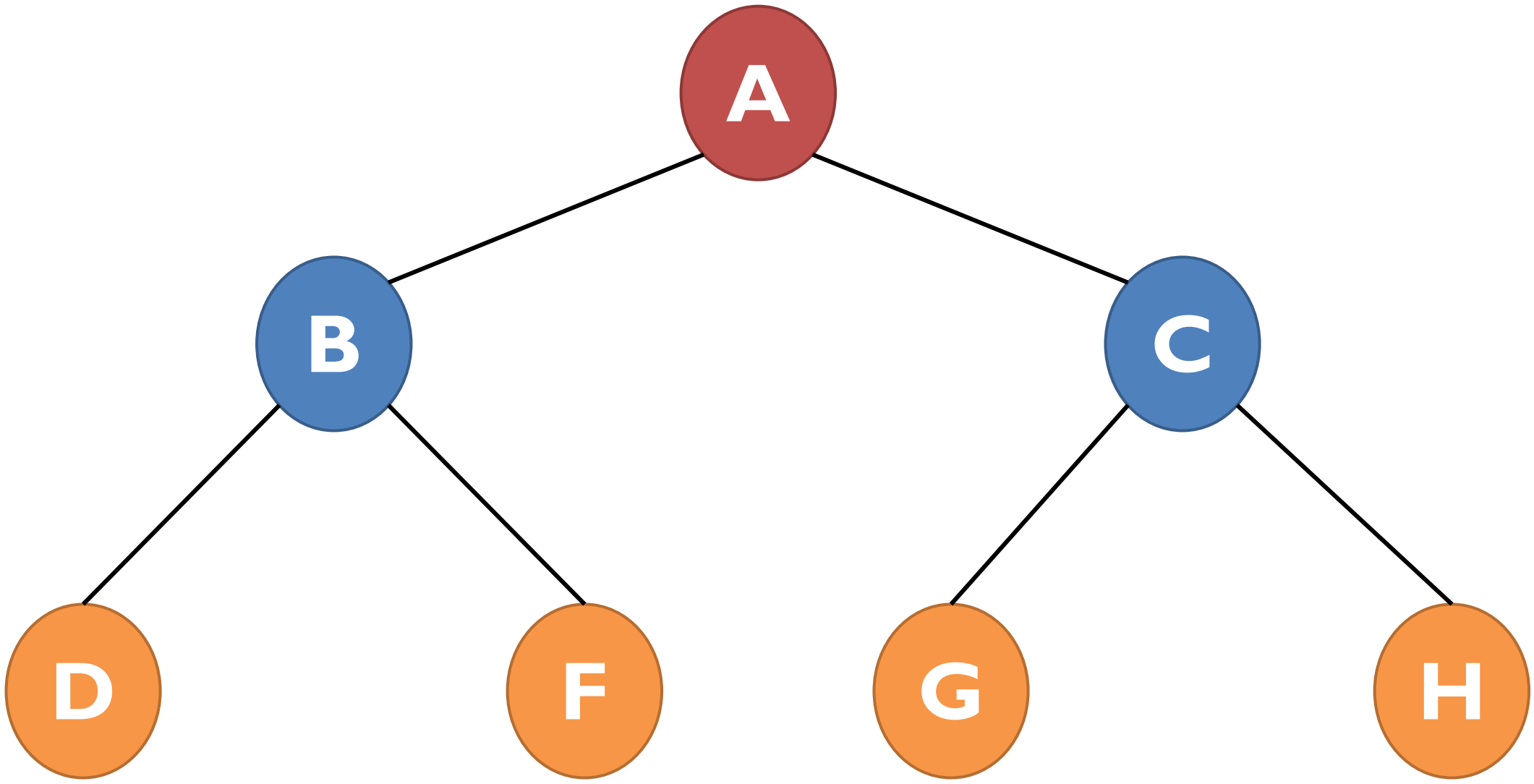


D E B F C A

Рекурсивная процедура:

- Обойти левое поддерево
- Обойти правое поддерево
- Посетить узел

Бинарные деревья



Создание бинарного дерева

```
BinTree *CreateTree(int n) {
  int nl, nr;
  BinTree *t;
  if (n == 0) t = NULL;
  else {
    t = new BinTree;
    nl = n / 2;
    nr = n - nl - 1;
    cout << "Enter tree element: ";
    cin >> t->data;
    t->leftTree = CreateTree(nl);
    t->rightTree = CreateTree(nr);
  }
  return t;
}
```

Печать бинарного дерева

```
void PrintTree(BinTree *t, int n) {
  if (t != NULL) {
    PrintTree(t->rightTree, n+1);
    for (int i = 0; i < n; i++)
      cout << " ";
    cout << t->data << endl;
    PrintTree(t->leftTree, n+1);
     }
}
```

Обходы бинарного дерева

```
// прямой обход
void DirectBypass(BinTree *t) {
  if (t != NULL) {
    cout << t->data << "\t";
    DirectBypass(t->leftTree);
    DirectBypass(t->rightTree);
  }
}
```

Обходы бинарного дерева

```
// симметричный обход
void BalancedBypass(BinTree *t) {
  if (t != NULL) {
    BalancedBypass(t->leftTree);
    cout << t->data << "\t";
    BalancedBypass(t->rightTree);
  }
}
```

Обходы бинарного дерева

```
// обратный обход
void ReverseBypass(BinTree *t) {
  if (t != NULL){
    ReverseBypass(t->leftTree);
    ReverseBypass(t->rightTree);
    cout << t->data << "\t";
  }
}
```

Пример бинарного дерева

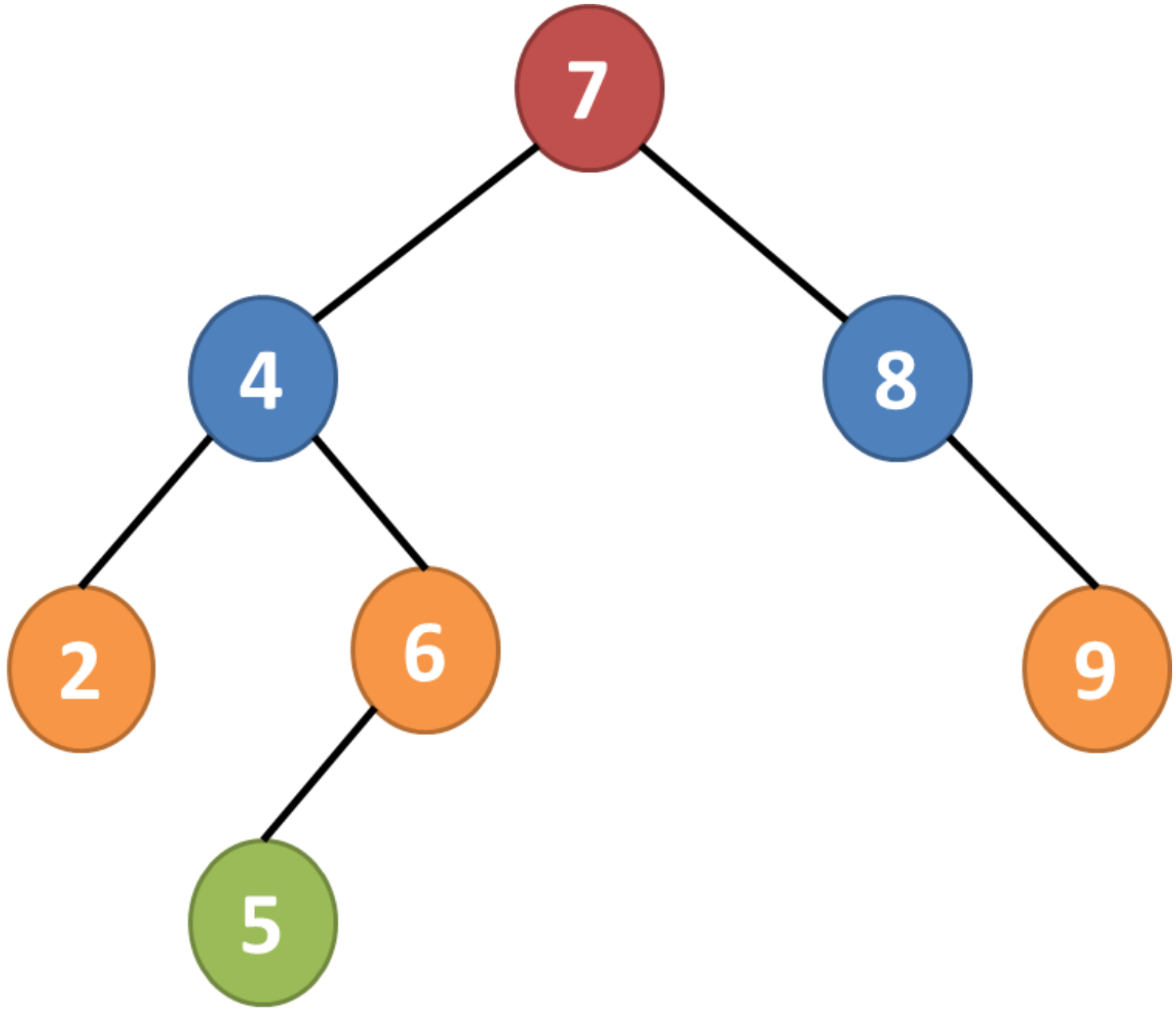
```
Enter elements amount: 6
Enter tree element: 1
Enter tree element: 2
Enter tree element: 3
Enter tree element: 4
Enter tree element: 5
Enter tree element: 6
   5
     6
 1
     4
   2
     3


Direct bypass:          1       2       3       4       5       6
Balanced bypass:        3       2       4       1       6       5
Reverse bypass:         3       4       2       6       5       1
```

Дерево поиска

Двоичное дерево упорядоченно, если для любой его вершины x справедливы такие свойства:

- все элементы в левом поддереве меньше элемента, хранимого в x,
- все элементы в правом поддереве больше элемента, хранимого в x,
- все элементы дерева различны.

Дерево поиска



Основные операции

- поиск вершины;
- добавление вершины;
- удаление вершины;
- вывод (печать) дерева;
- очистка дерева.

Добавление вершины

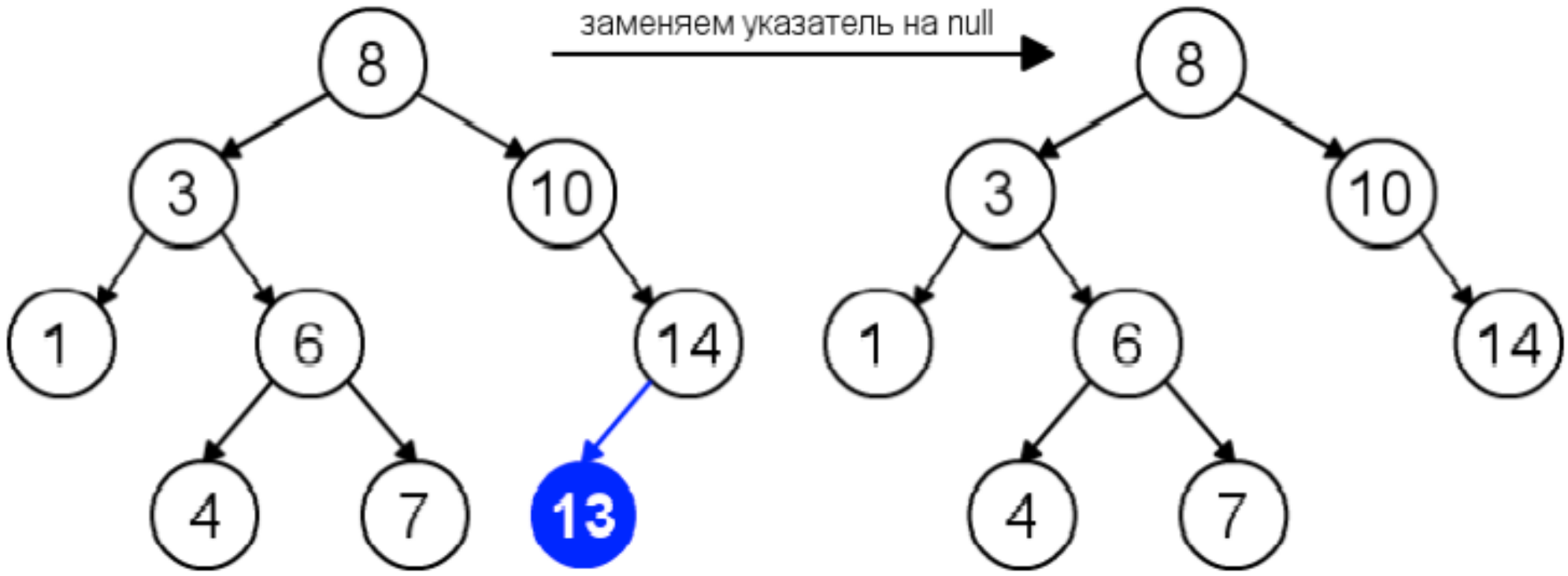
```
int a[] = {8, 6, 7, 4, 5, 3, 3, 2, 10, 15, 9, 12};
```

Удаление вершины

- Функция удаления вершины из дерева является одной из **наиболее сложных операций при работе с бинарными деревьями**. При удалении возможны **три случая**, каждый из которых должен быть предусмотрен в разрабатываемой функции:

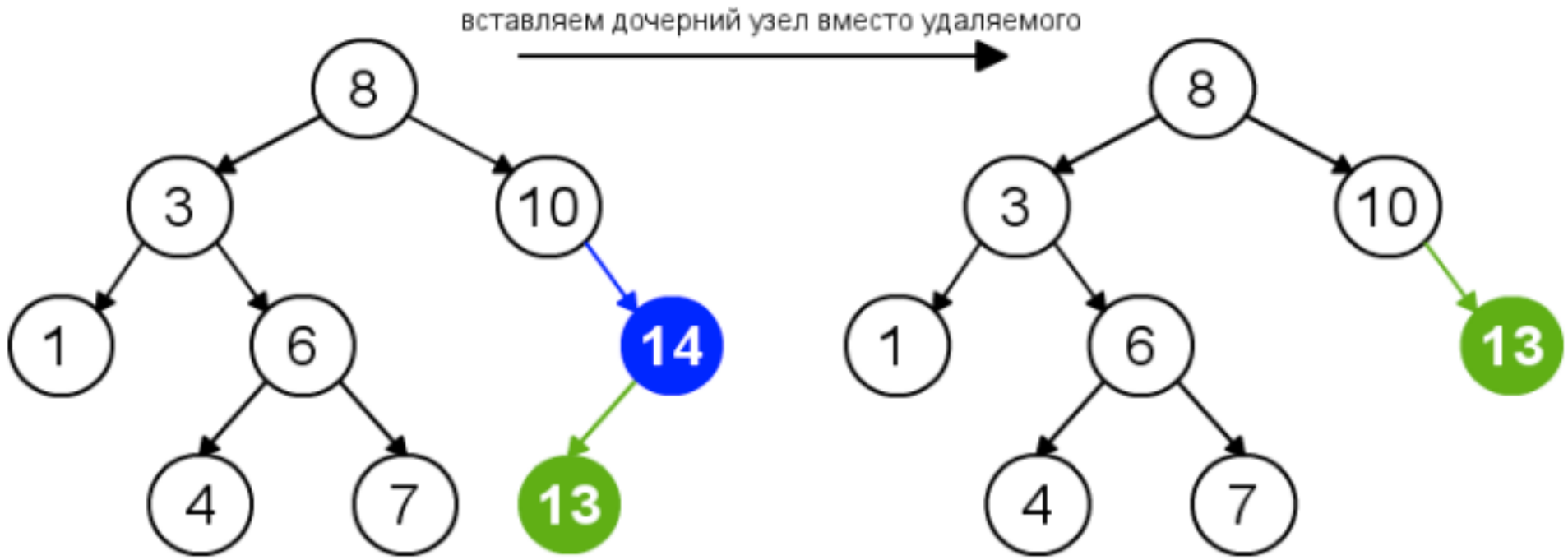
Удаление вершины

1. Удаляемая вершина не имеет потомков: если **удаляемая вершина не имеет потомков**, допустимо просто убрать ее из дерева. При этом порядок остальных вершин не изменяется.



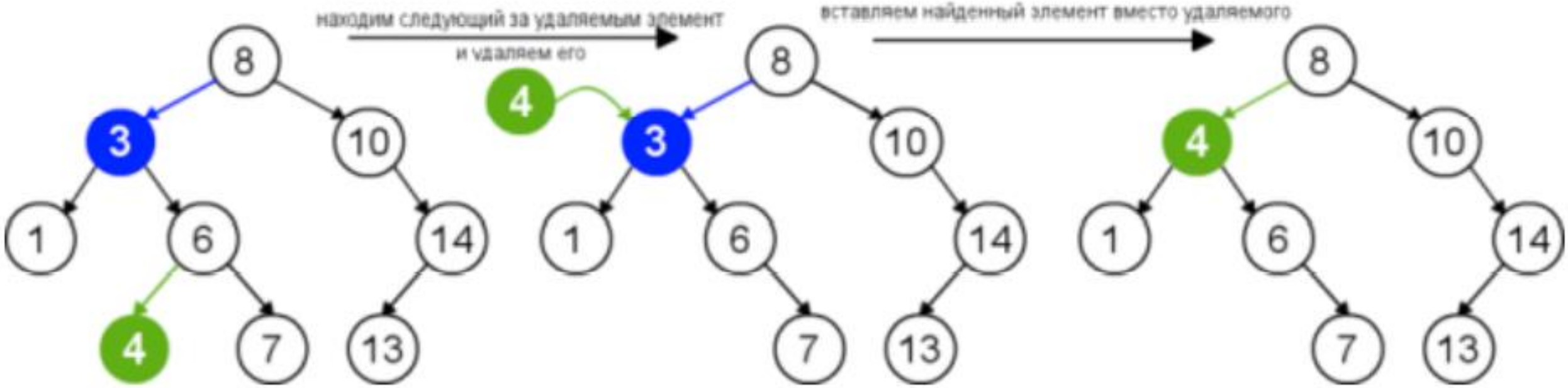
Удаление вершины

2. **Удаляемая вершина имеет одну дочернюю вершину**: если удаляемая вершина имеет одну дочернюю вершину, можно заменить ее дочерней вершиной. При этом порядок потомков данной вершины остается тем же, так как эти вершины также являются и потомками дочерней вершины.



Удаление вершины

3. Удаляемая вершина **имеет две дочерних вершины**: если удаляемая вершина имеет две дочерних вершины, вовсе не обязательно, что одна из них займет ее место. Если потомки вершины также имеют по две дочерних, то для размещения всех дочерних вершин в позиции удаленной вершины просто нет места.



Листинги для двоичных деревьев поиска

```
struct BSTree

{

	int data;

	BSTree * left, * right;

} *root;
```

```
BSTree * InsertNode(BSTree *, int);

void PrintTree(BSTree *, int);

BSTree * GetMinimum(BSTree *);

BSTree * GetMaximum(BSTree *);

BSTree * DeleteNode(BSTree *, int);
```

Листинги для двоичных деревьев поиска

```
void main()
{
	int a[] = {8, 6, 7, 4, 5, 3, 3, 2, 10, 15, 9, 12};
	for (int i = 0; i < sizeof(a)/sizeof(int); i++)
	{
		root = InsertNode(root, a[i]);
	}

	PrintTree(root, 3);

	cout << "\nMin = \t" << GetMinimum(root)->data << endl;
	cout << "Max = \t" << GetMaximum(root)->data << endl;

	int element;
	cout << "\Enter element:\t";
	cin >> element;
	cout << endl;

	root = DeleteNode(root, element);
	PrintTree(root, 3);

	system("pause");
}
```

Листинги для двоичных деревьев поиска

```
void PrintTree(BSTree *_root, int n)
{
        if (_root)
        {
                PrintTree(_root->right, n + 2);
                for (int i = 0; i < n; i++)
                        cout << "-";
                cout << _root->data;
                PrintTree(_root->left, n + 2);
        }
        else
                cout << endl;
}
```

Листинги для двоичных деревьев поиска

```
BSTree * InsertNode(BSTree *_root, int _data)
{
	if (_root == nullptr)
	{
		_root = new BSTree;
		_root->data = _data;
		_root->left = nullptr;
		_root->right = nullptr;
	}
	else
		if (_data < _root->data)
		{
			_root->left = InsertNode(_root->left, _data);
		}
		else
			if (_data > _root->data)
			{
				_root->right = InsertNode(_root->right, _data);
			}
	return _root;
}
```

Листинги для двоичных деревьев поиска

```
BSTree * GetMinimum(BSTree *_root)
{
	if (!_root->left)
	{
		return _root;
	}
	return GetMinimum(_root->left);
}
```

```
BSTree * GetMaximum(BSTree *_root)
{
	if (!_root->right)
	{
		return _root;
	}
	return GetMaximum(_root->right);
}
```

```
BSTree * DeleteNode(BSTree *_root, int _data)
{
	if (_root == nullptr)
	{
		return nullptr;
	}
	if (_data == _root->data)
	{
		if (_root->left == nullptr && _root->right == nullptr)
		{
			delete _root;
			return nullptr;
		}
		if (_root->left == nullptr && _root->right != nullptr)
		{
			BSTree * temp = _root->right;
			delete _root;
			return temp;
		}
```

```
            _root->data = GetMinimum(_root->right)->data;
            _root->right = DeleteNode(_root->right, _root->data);
            return _root;
        }
        if (_data < _root->data)
        {
            _root->left = DeleteNode(_root->left, _data);
            return _root;
        }
        if (_data > _root->data)
        {
            _root->right = DeleteNode(_root->right, _data);
            return _root;
        }

        return _root;
    }
```

Результат работы программы

```
        15
           12
      10
         9
   8
         7
      6
            5
         4
            3
               2


Min =    2
Max =    15
Enter element:  10
```

```
           15
        12
           9
     8
           7
        6
              5
           4
              3
                 2
```