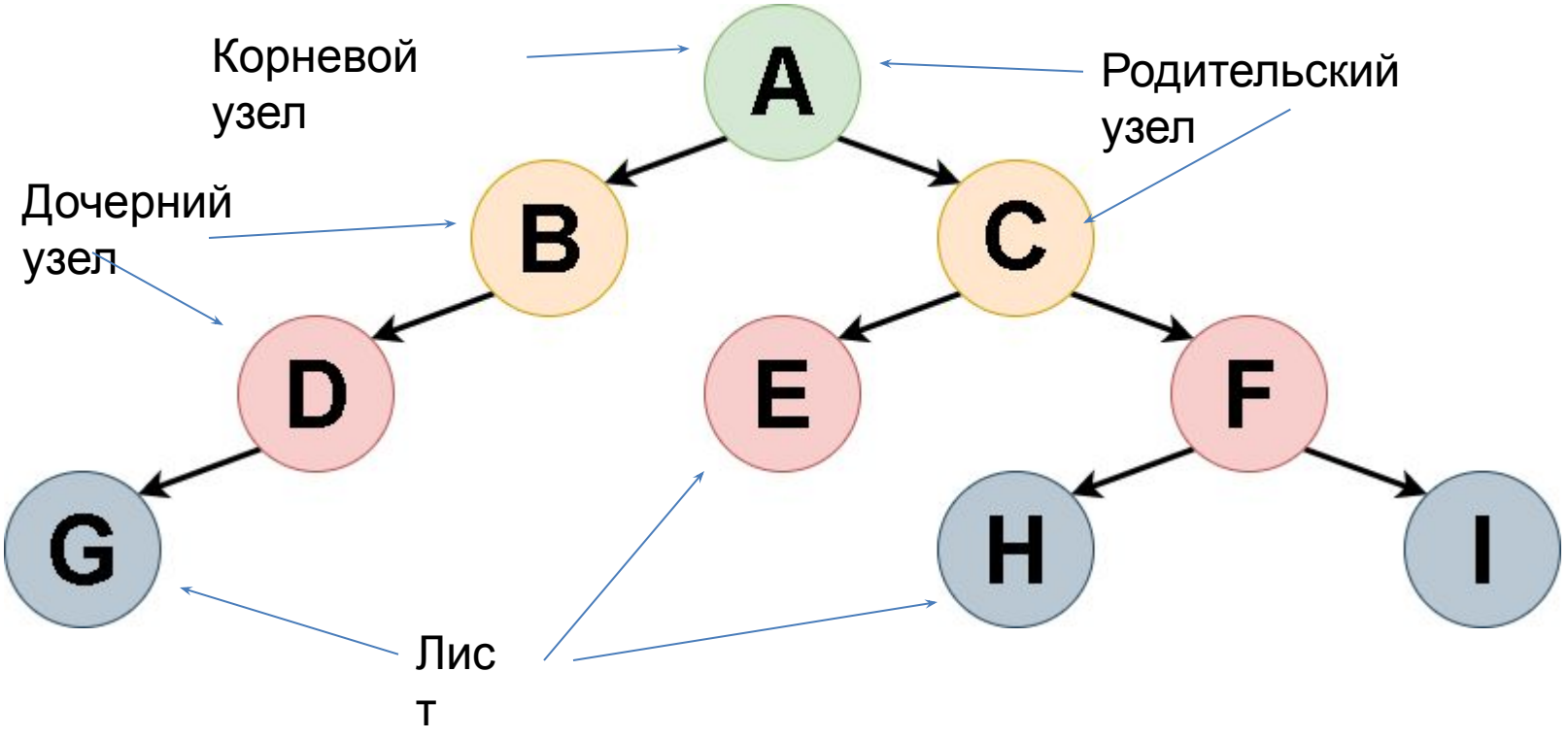

Бинарные
деревья

Вспоминаем…

Дерево – это множество данных, представляющее собой древовидную структуру в виде набора связанных узлов.

Узел – это единица хранения данных в дереве, имеющая также указатели на связанные с ней узлы.



Когда дерево бинарно?

Бинарное дерево связывает до двух других дочерних узлов, которые можно визуализировать пространственно ниже родительского узла, один из которых расположен слева, а другой справа.

Именно связь между дочерними и родительским узлами делает бинарное дерево такой эффективной структурой данных.

Почему это удобно?

- Если информация должна быть расположена иерархично. Примером является файловая система компьютера.
- Если информация должна быть структурирована. Тогда хранение в виде бинарного дерева позволяет уменьшить скорость поиска данных и доступа к хранимой информации.
- Если необходима высокая скорость добавления или удаления данных.
- Если заранее неизвестен хранимый объем данных. Бинарные деревья не имеют ограничения на количество узлов, поскольку узлы связаны указателями.

Бинарное дерево поиска

- Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева

Реализуем узел

```
struct node
{
    int x; //ключ – значение узла, типа int
    node* l; //указатель на левого потомка
    node* r; //указатель на правого потомка
};
```

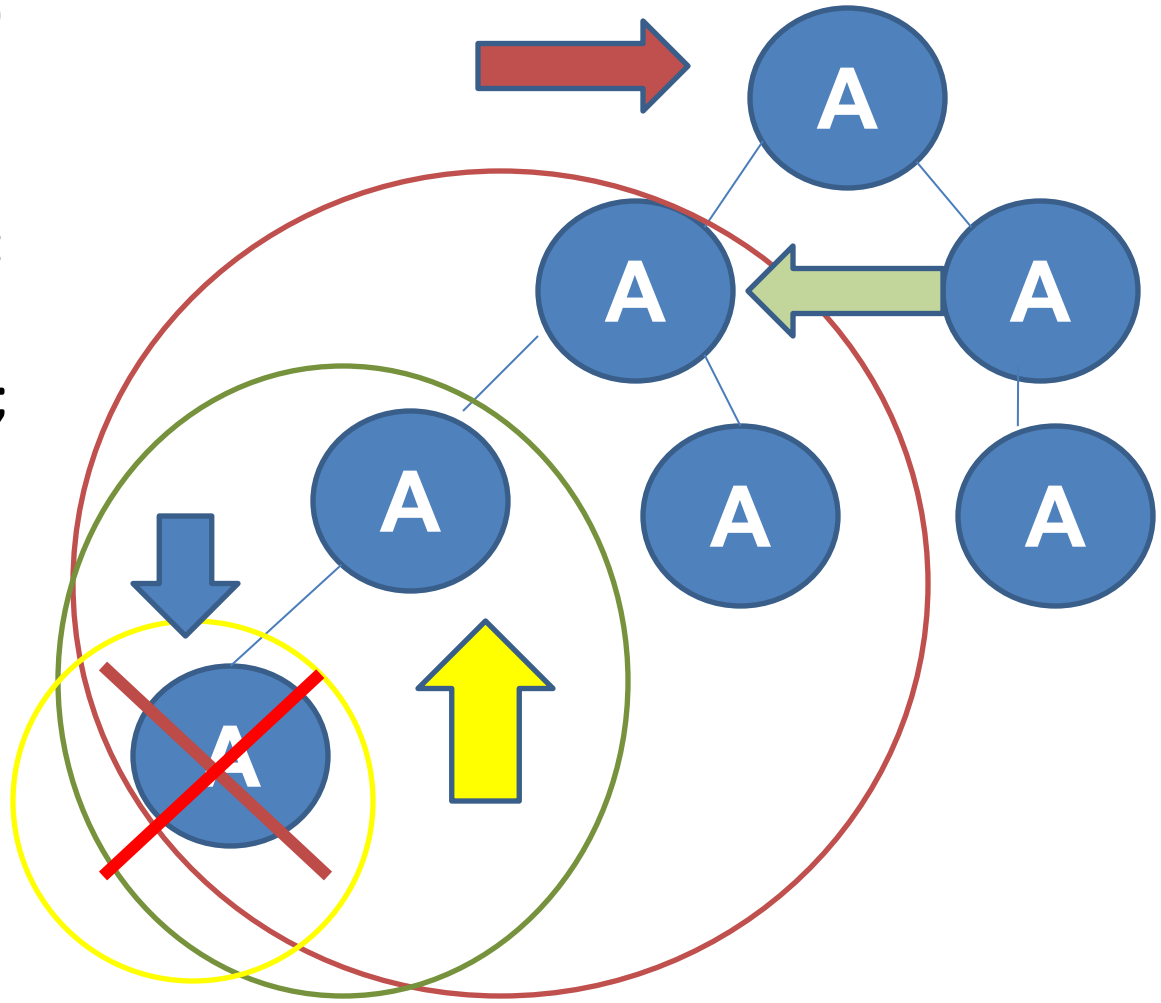
Что реализуем?

- Создание дерева
- Добавление узла в дерево
- Печать дерева
- Удаление дерева

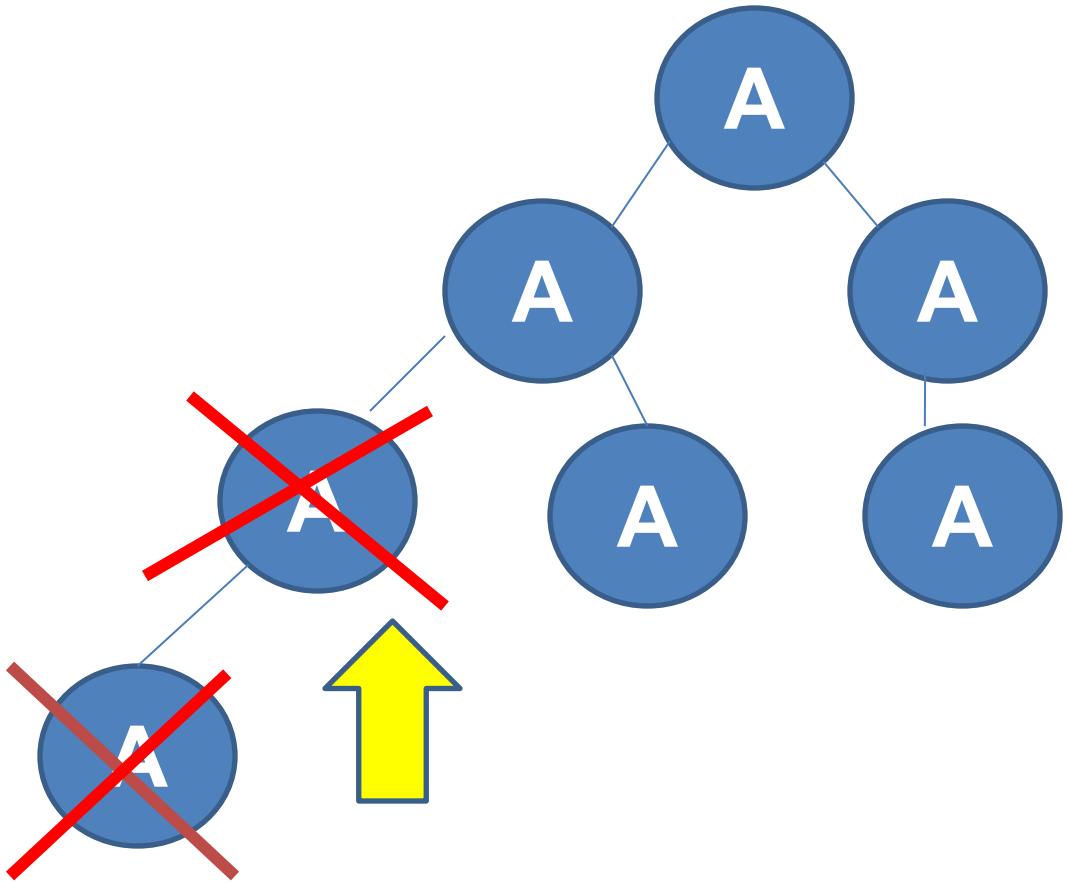
Срубаем дерево

```
void del(Node*& Tree) {
    if (Tree != NULL)  //Пока не встретится пустое звено
    {
        del(Tree->l);
//Рекурсивная функция прохода по левому поддереву
        del(Tree->r);
//Рекурсивная функция для прохода по правому поддереву
        delete Tree;
//Убиваем конечный элемент дерева
        Tree = NULL;
//Может и не обязательно, но плохого не будет
    }
}
```

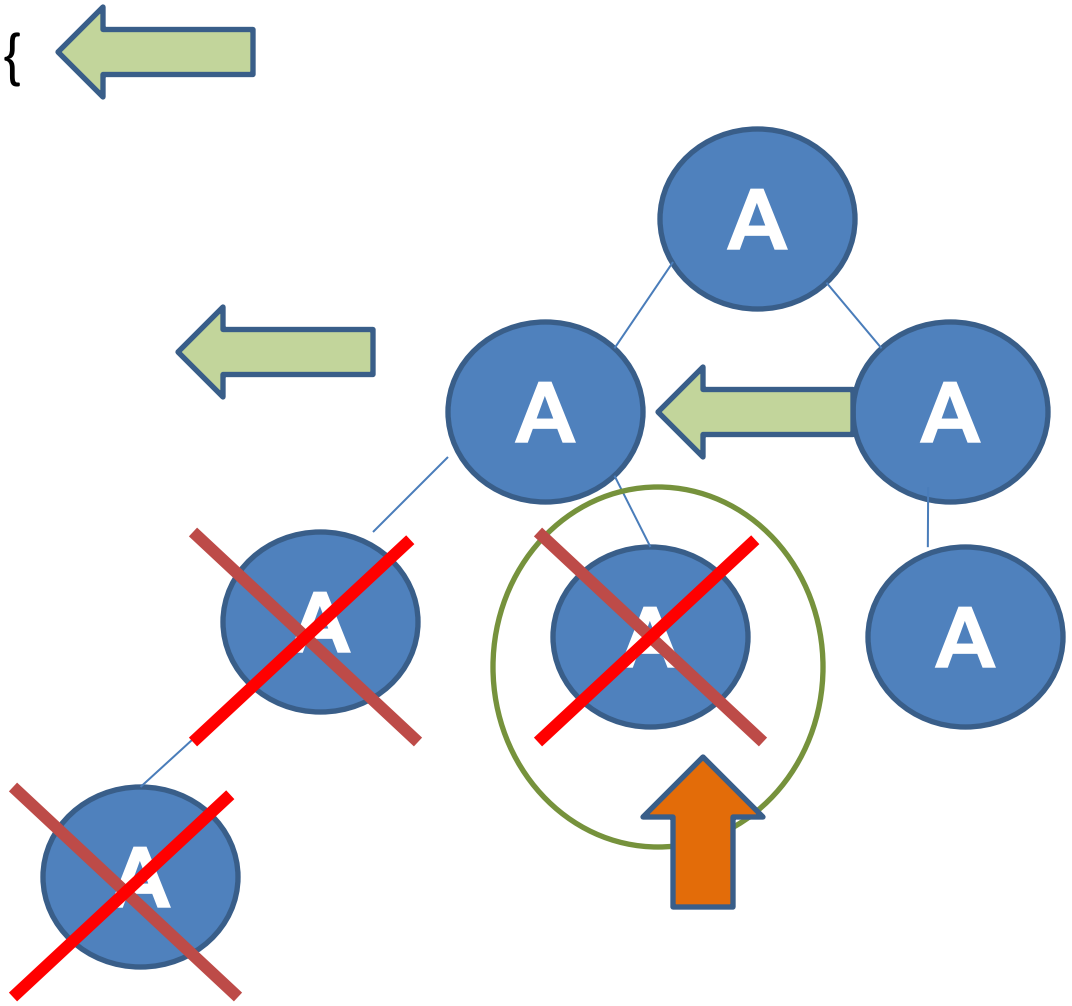
```
void del(Node*& Tree) {
    if (Tree != NULL)
    {
        del(Tree->l);
        del(Tree->r);
        delete Tree;
        Tree = NULL;
    }
}
```



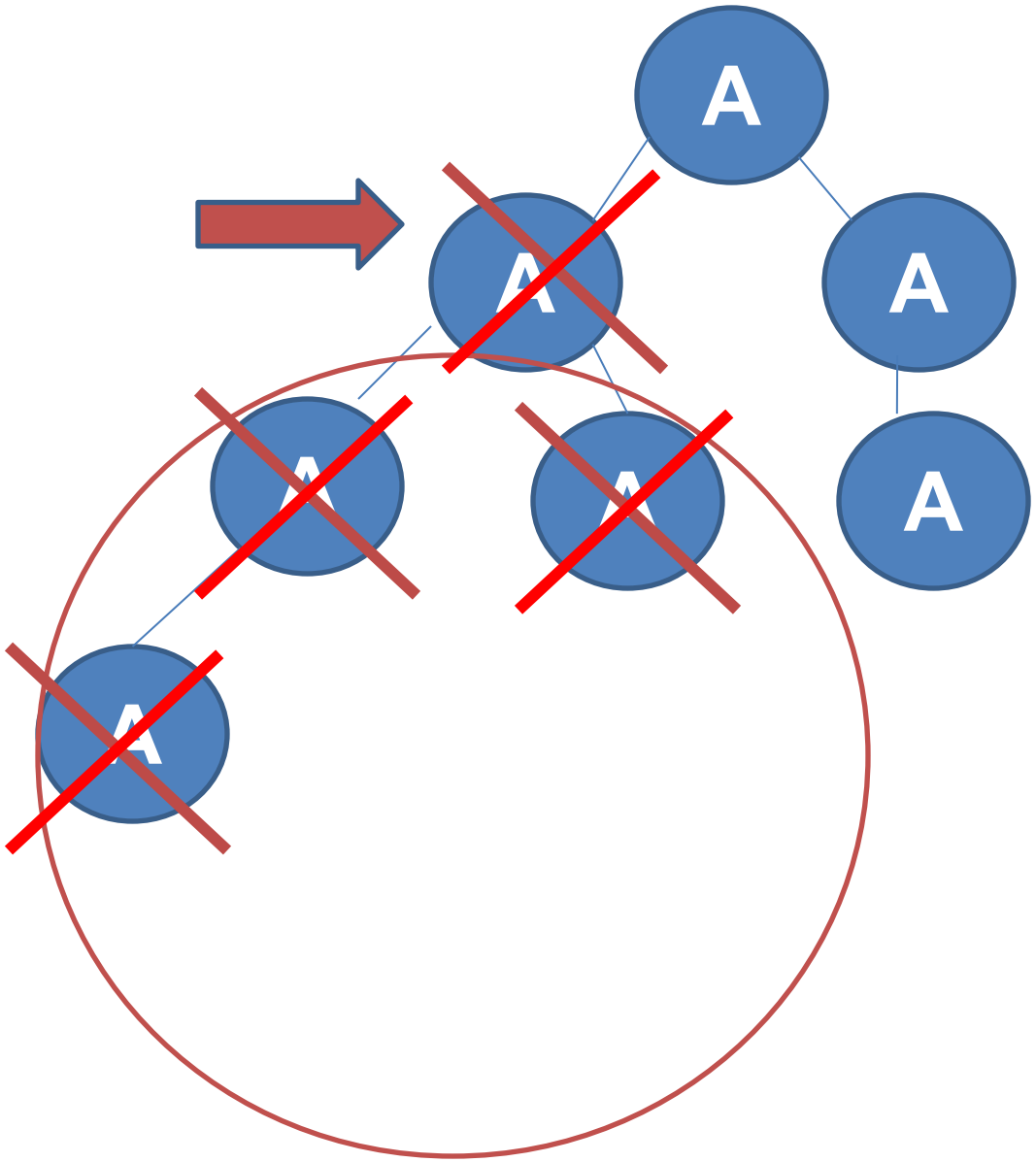
```
void del(Node*& Tree) {
    if (Tree != NULL)
    {
        del(Tree->l);
        del(Tree->r);
        delete Tree;
        Tree = NULL;
    }
}
```



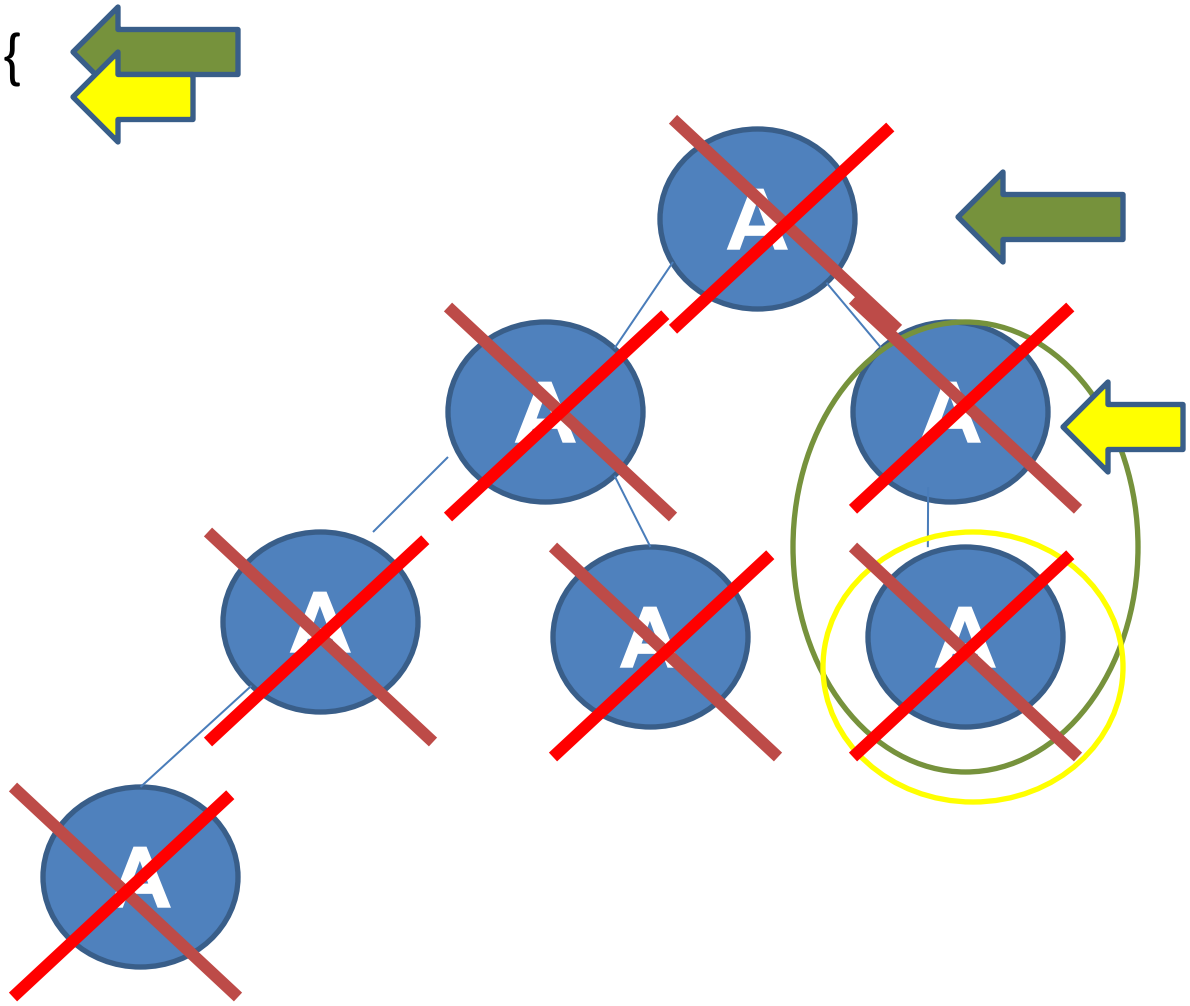
```
void del(Node*& Tree) {
    if (Tree != NULL)
    {
        del(Tree->l);
        del(Tree->r);
        delete Tree;
        Tree = NULL;
    }
}
```



```
void del(Node*& Tree) {
    if (Tree != NULL)
{
        del(Tree->l);
        del(Tree->r);
        delete Tree;
        Tree = NULL;
    }
}
```

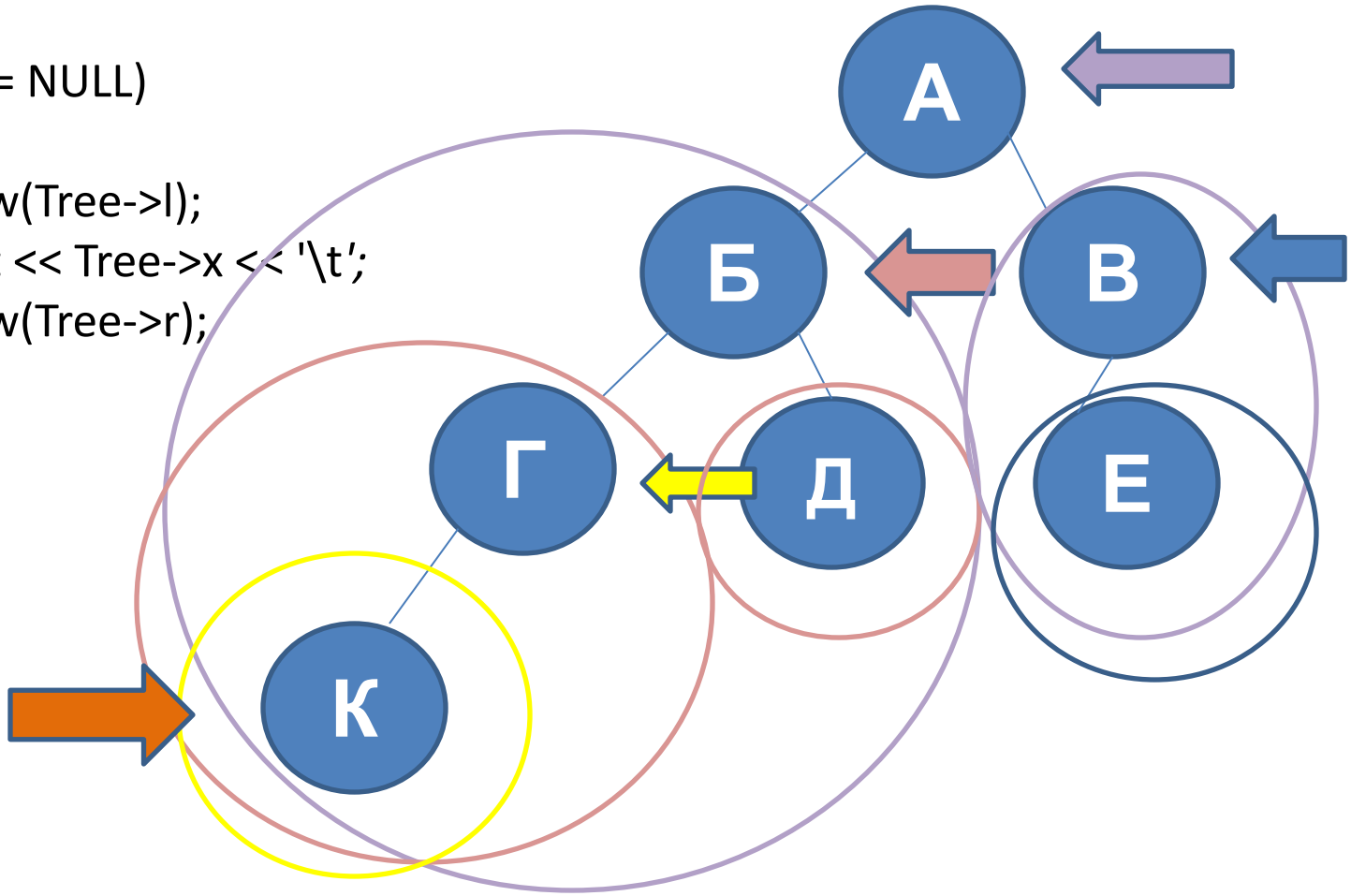


```
void del(Node*& Tree) {
    if (Tree != NULL)
    {
        del(Tree->l);
        del(Tree->r);
        delete Tree;
        Tree = NULL;
    }
}
```

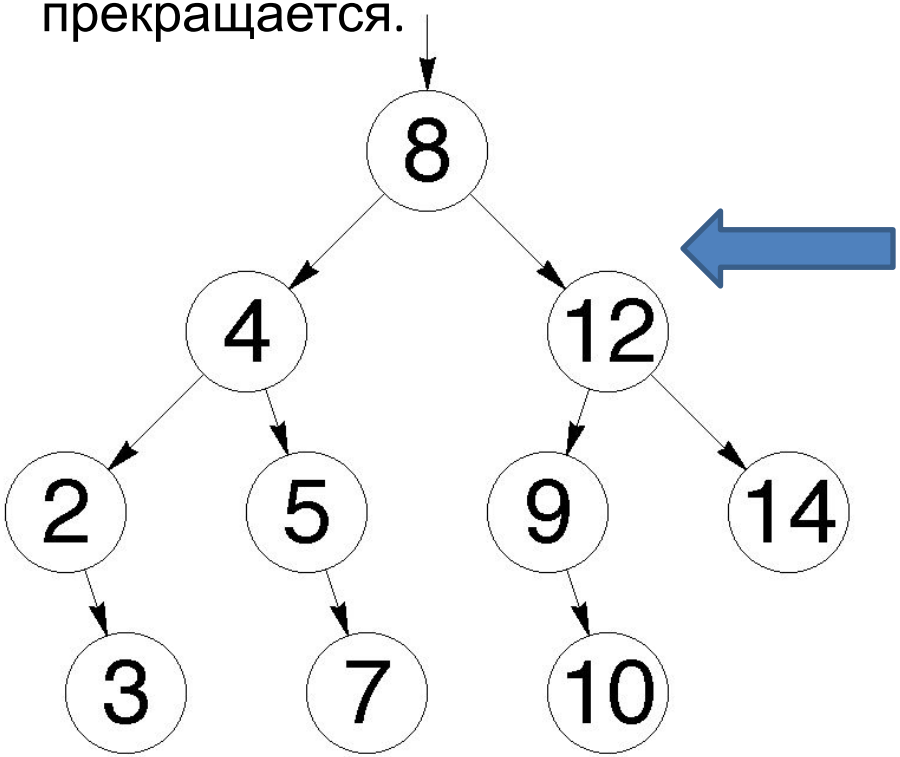


```
void show(Node*& Tree)          //Функция обхода
{
    if (Tree != NULL)
 //Пока не встретится пустое звено
    {
        show(Tree->l);
//Рекурсивная функция для вывода левого поддерева
        cout << Tree->x << '\t';
 //Отображаем корень дерева
        show(Tree->r);
 //Рекурсивная функция для вывода правого поддерева
    }
}
```

```
void show(Node*& Tree)
{
        if (Tree != NULL)
        {
                show(Tree->l);
                cout << Tree->x << '\t';
                show(Tree->r);
        }
}
```

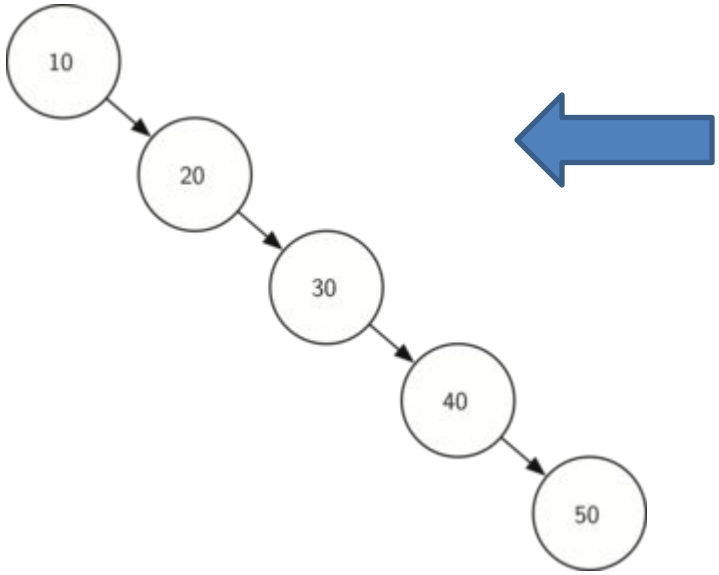


При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.



Сбалансировн
но

НО! Если не планируется вставка / удаление элемента – массив – лучшее решение для поиска и хранения информации.



Экстремально несбалансированно

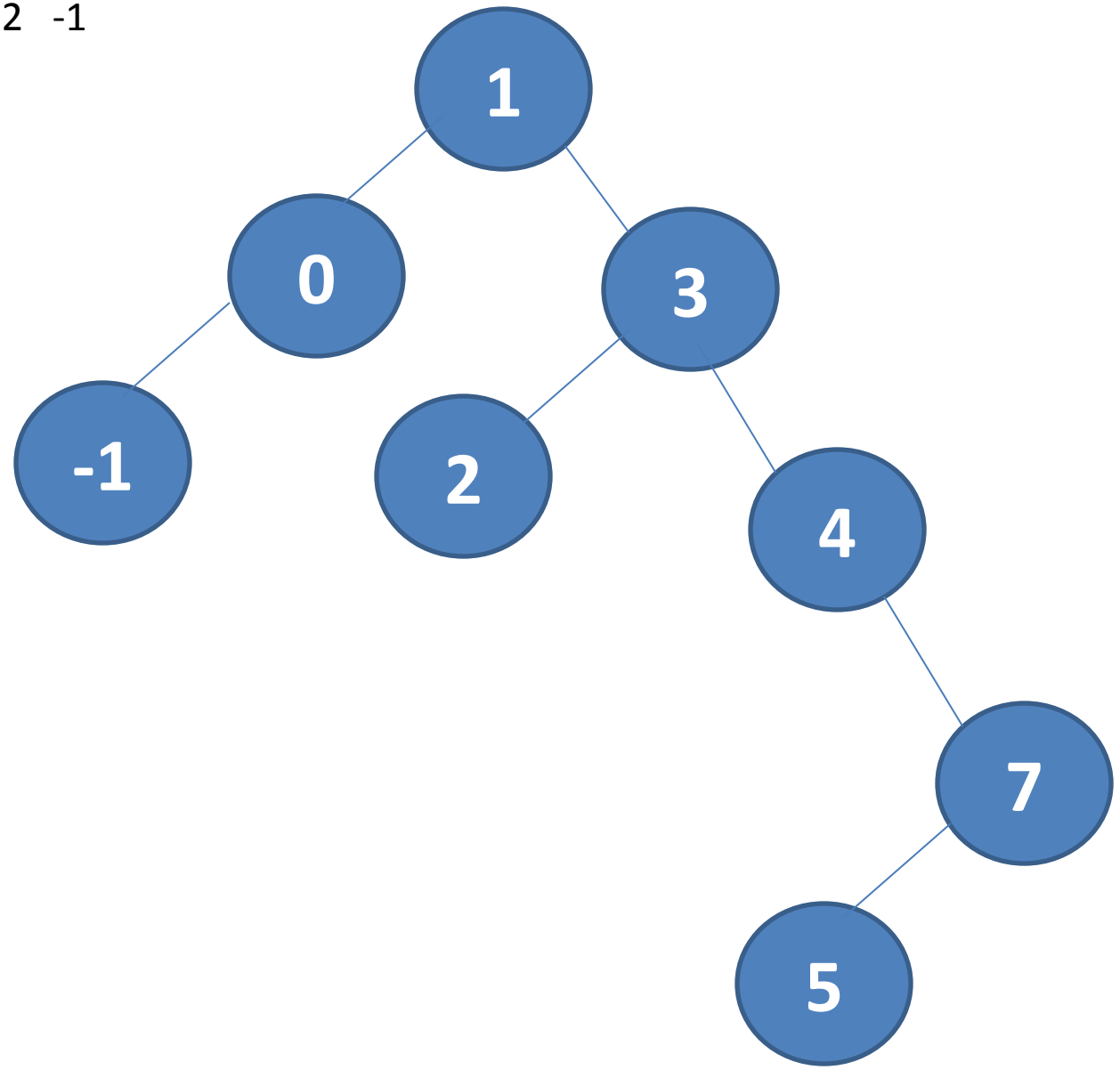
```
void add_node(int x, Node*& MyTree) //Функция добавления звена в дерево
{
	if (NULL == MyTree)		//Если дерева нет, то формируем корень
	{
		MyTree = new Node;
  //Выделяем память под звено дерева
		MyTree->x = x;		//Записываем данные в звено
		MyTree->l = MyTree->r = NULL;
//Подзвенья инициализируем пустотой во избежание ошибок
	}

	…
```

```
if (x < MyTree->x)
//Если нововведенный элемент x меньше чем элемент x узла дерева
    {
        if (MyTree->l != NULL) add_node(x, MyTree->l);
//При помощи рекурсии заталкиваем элемент на свободный участок
        else        //Если элемент получил свой участок, то
        {
            MyTree->l = new Node;
  //Выделяем память левому подзвену. Именно подзвену, а не просто звену
            MyTree->l->l = MyTree->l->r = NULL;
 //У левого подзвена будут свои левое и правое подзвенья, инициализируем их пустотой
            MyTree->l->x = x;
 //Записываем в левое подзвено записываемый элемент
        }
    }
```

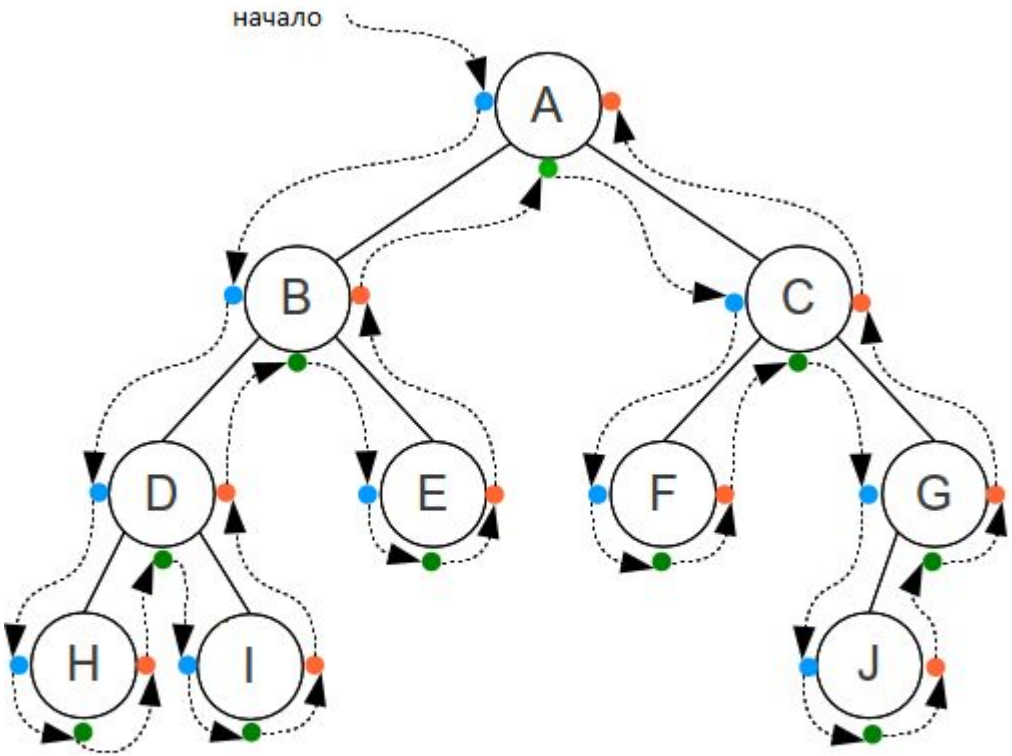
```
if (x > MyTree->x)
 //Если нововведенный элемент x больше чем элемент x из текущего узла
	{
		if (MyTree->r != NULL) add_node(x, MyTree->r); //При помощи рекурсии заталкиваем элемент на свободный участок
		else		//Если элемент получил свой участок, то
		{
			MyTree->r = new Node;
 //Выделяем память правому подзвену. Именно подзвену, а не просто звену
			MyTree->r->l = MyTree->r->r = NULL;
//У правого подзвена будут свои левое и правое подзвенья, инициализируем их пустотой
			MyTree->r->x = x;
 //Записываем в правое подзвено записываемый элемент
		}
	}
```

1 3 0 4 7 5 2 -1



Техника обхода дерева

Прямой:
вызов
Левый
правый



Симметричный
Левый
вызов
правый

Обратный
Левый
Правый
вызов

Задание

- Написать функцию симметричного обхода дерева
- Написать функцию обратного обхода дерева
- Написать функцию поиска в бинарном дереве поиска