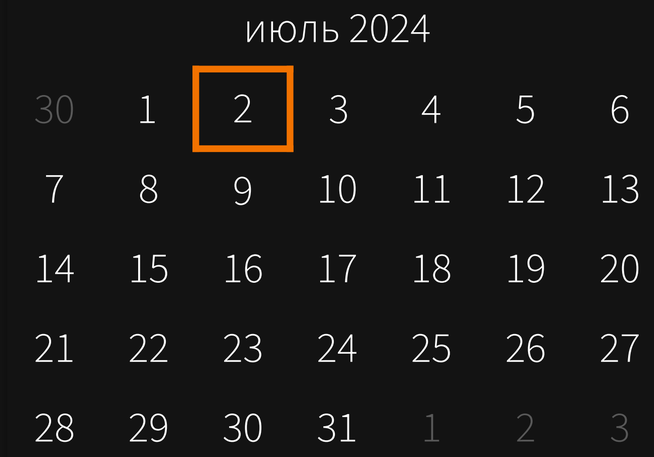
основны оценки сложности алгоритмов

03

бинарное дерево поиска • проблема сбалансированности

Нестеров Р.А., PhD, доцент
департамента программной инженерии



план лекции

01

бинарное дерево: классификация и характеристики

02

бинарное дерево поиска: вставка элементоа и обход

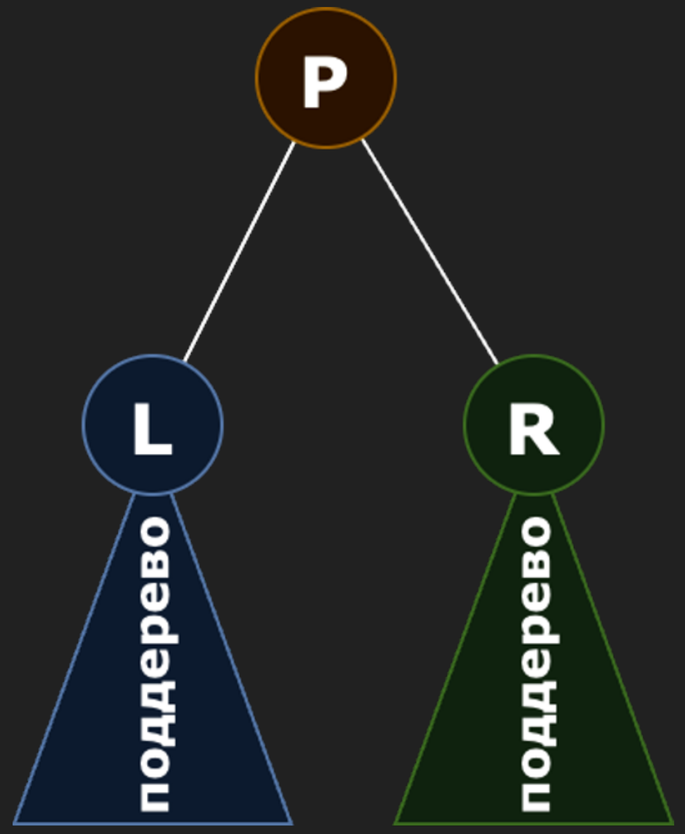
03

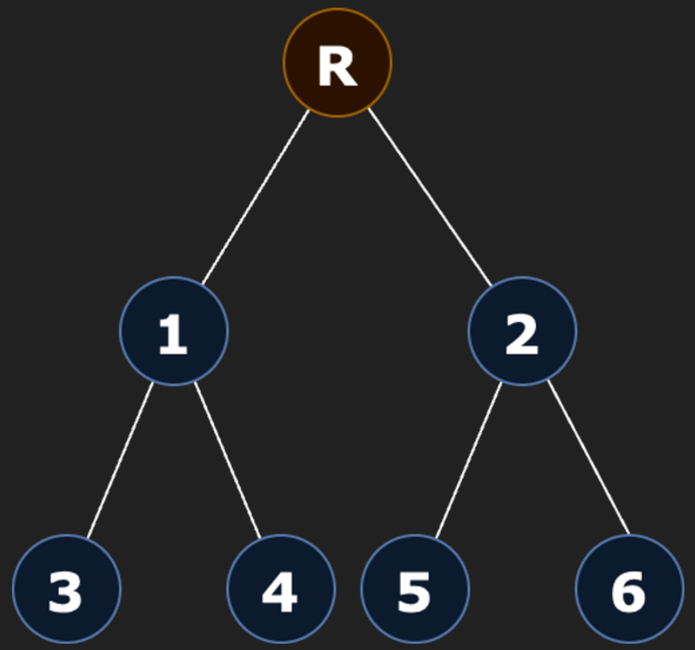
сложность основных операций с деревом поиска

односвязный граф без циклов

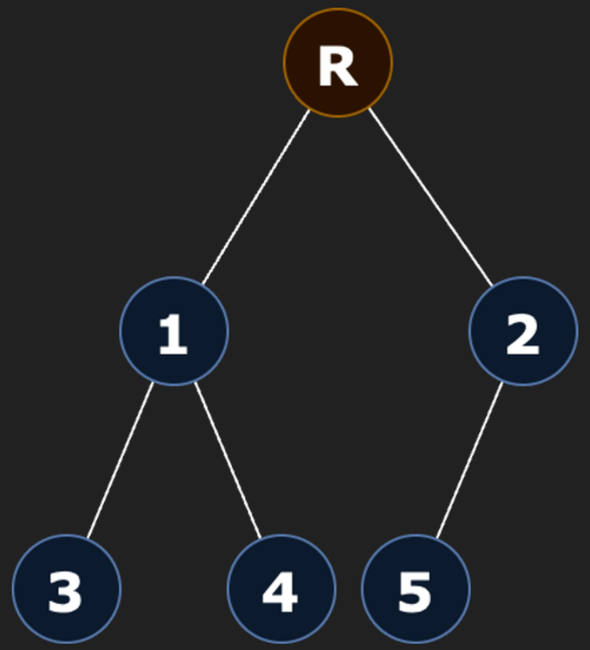
бинарное дерево — рекурсивная структура

1. каждая вершина имеет не более двух потомков (левый L и правый R)
2. вершины–потомки могут определять целые поддеревья

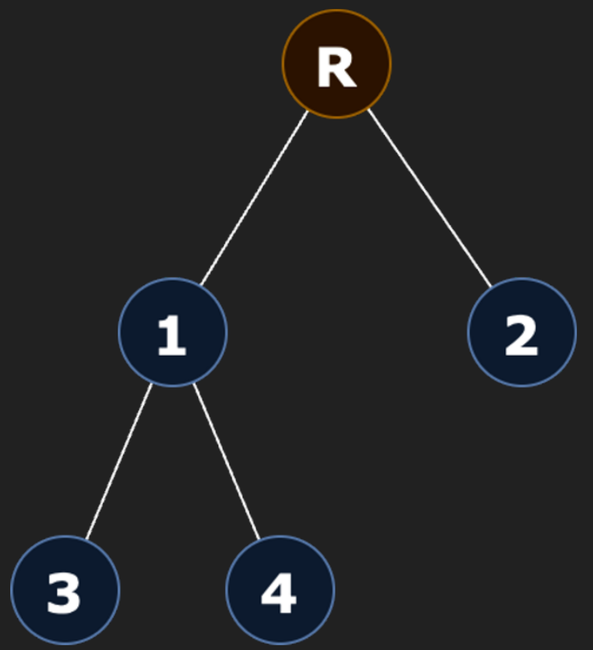




ИДЕАЛЬНОЕ

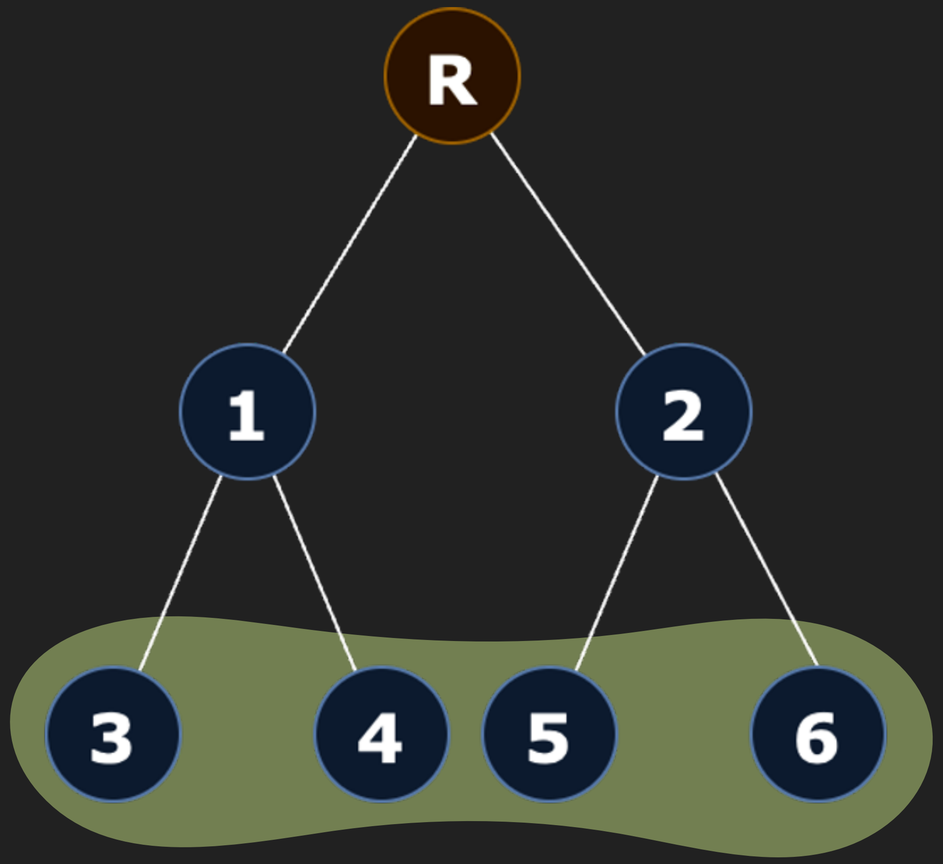


ПОЛНОЕ



СТРОГОЕ

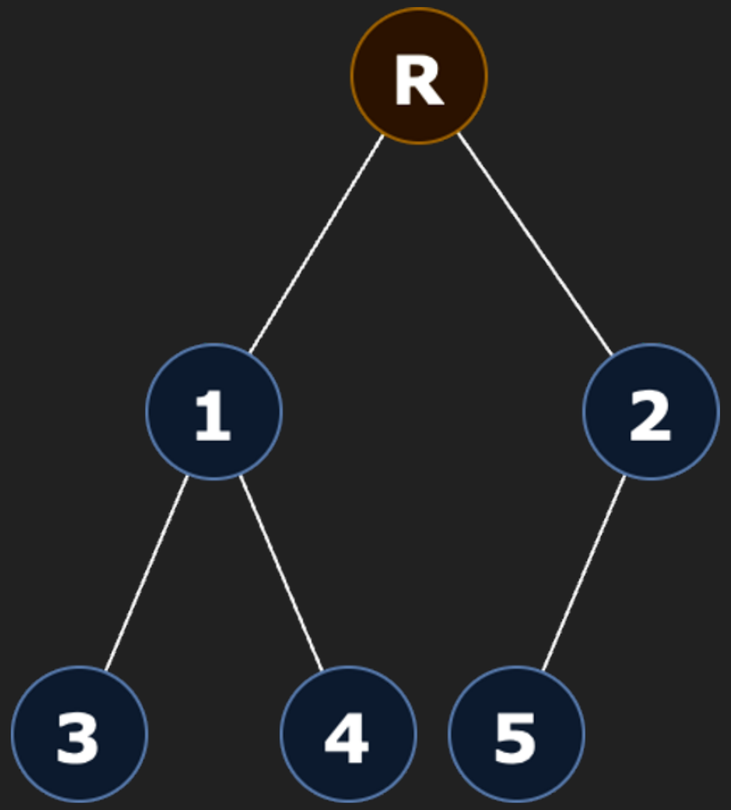
идеальное бинарное дерево • PERFECT



1. все вершины, кроме листьев, имеют двух потомков
2. все листья располагаются на одном уровне

ВЫСОТА	$\Theta(\log n)$
ЧИСЛО ВЕРШИН	$2^{h+1} - 1$
ЧИСЛО ЛИСТЬЕВ	2^h

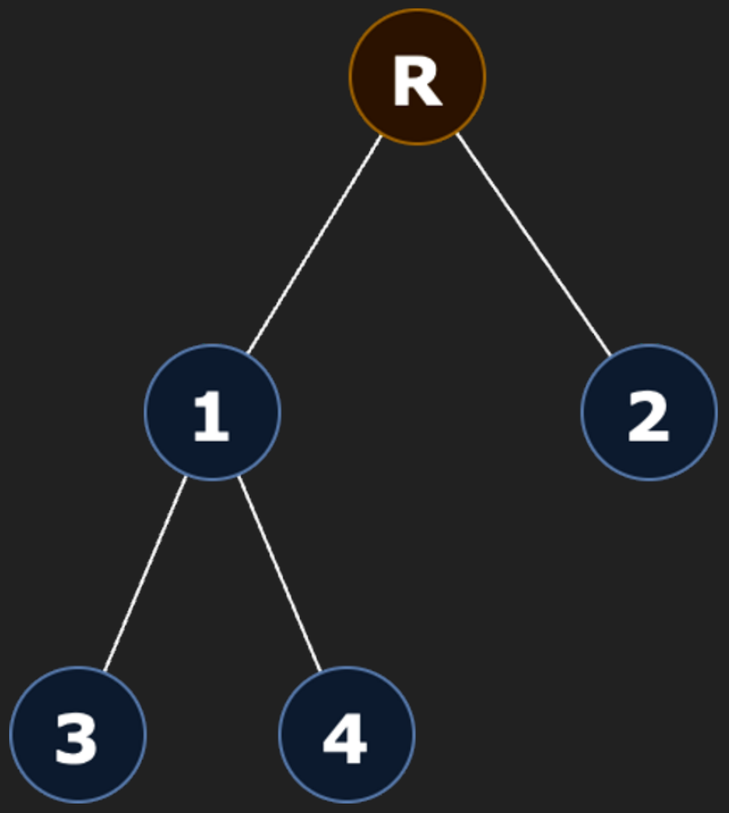
полное бинарное дерево • COMPLETE



1. все уровни дерева, кроме м. б. последнего, заполнены
2. уровни дерева заполняются по порядку слева направо

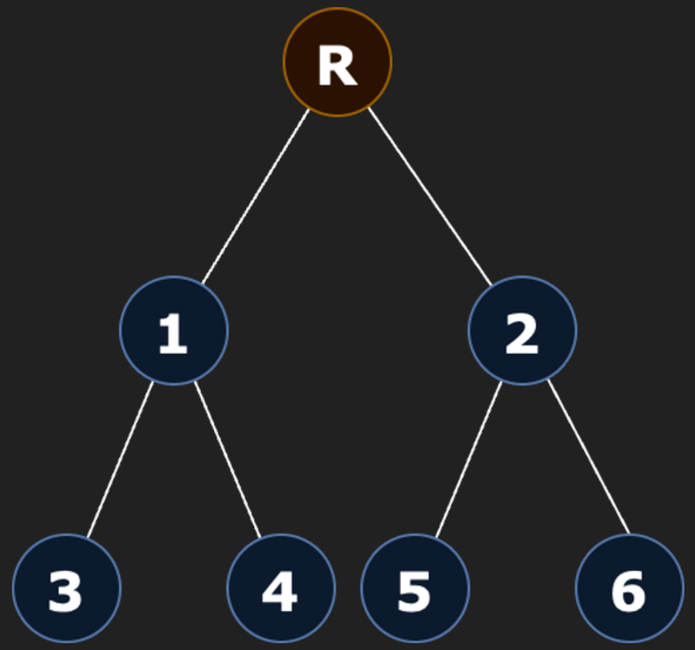
ВЫСОТА	$\lfloor \log n \rfloor$
ЧИСЛО ВЕРШИН	$2^h - 1 + L$
ЧИСЛО ЛИСТЬЕВ	L

строгое бинарное дерево • FULL

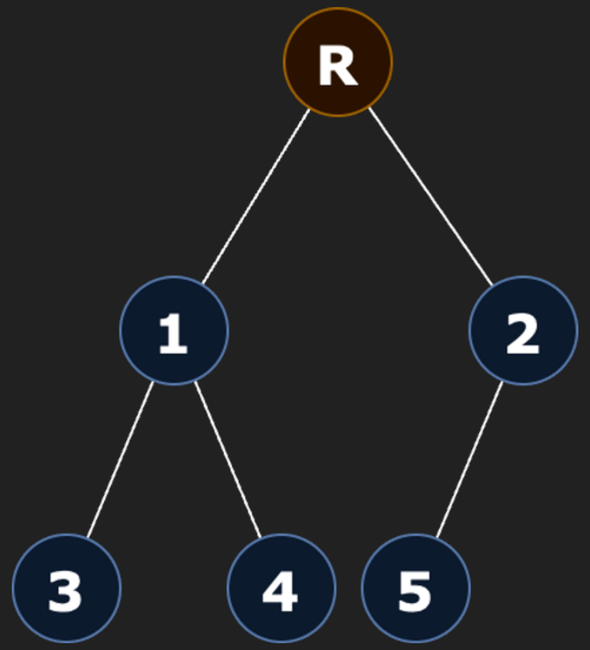


1. каждая вершина, кроме листовой, имеет два потомка
2. число листьев больше числа внутренних вершин I на 1

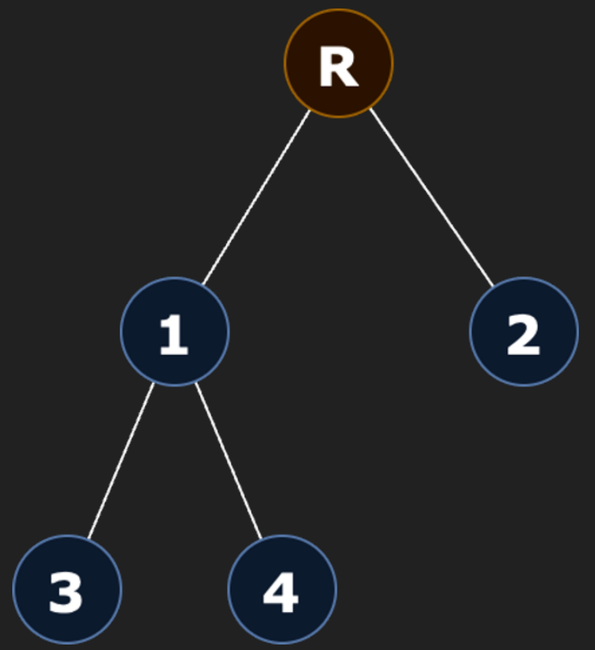
ВЫСОТА	$[\log n, I]$
ЧИСЛО ВЕРШИН	$[2h+1, 2^{h+1}-1]$
ЧИСЛО ЛИСТЬЕВ	$I+1, (n+1)/2$



ИДЕАЛЬНОЕ



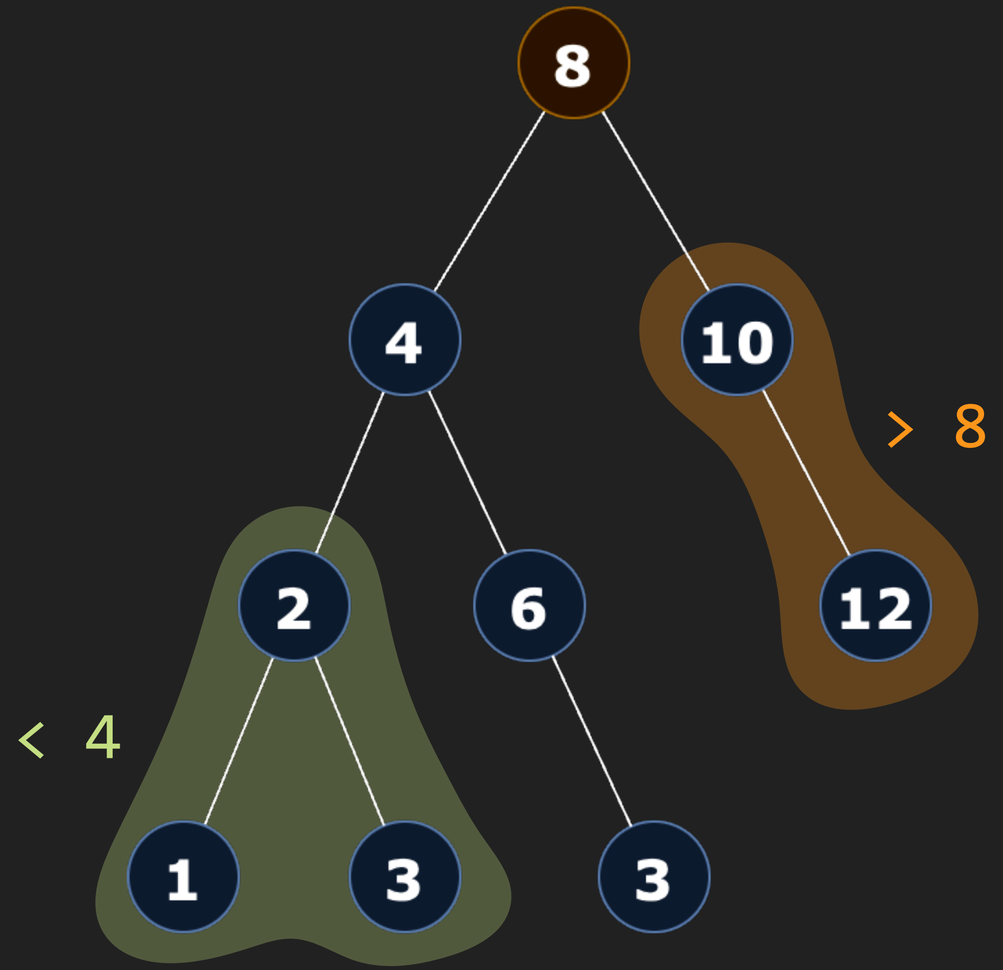
ПОЛНОЕ



СТРОГОЕ

ИДЕАЛЬНОЕ ⇔ ПОЛНОЕ И СТРОГОЕ

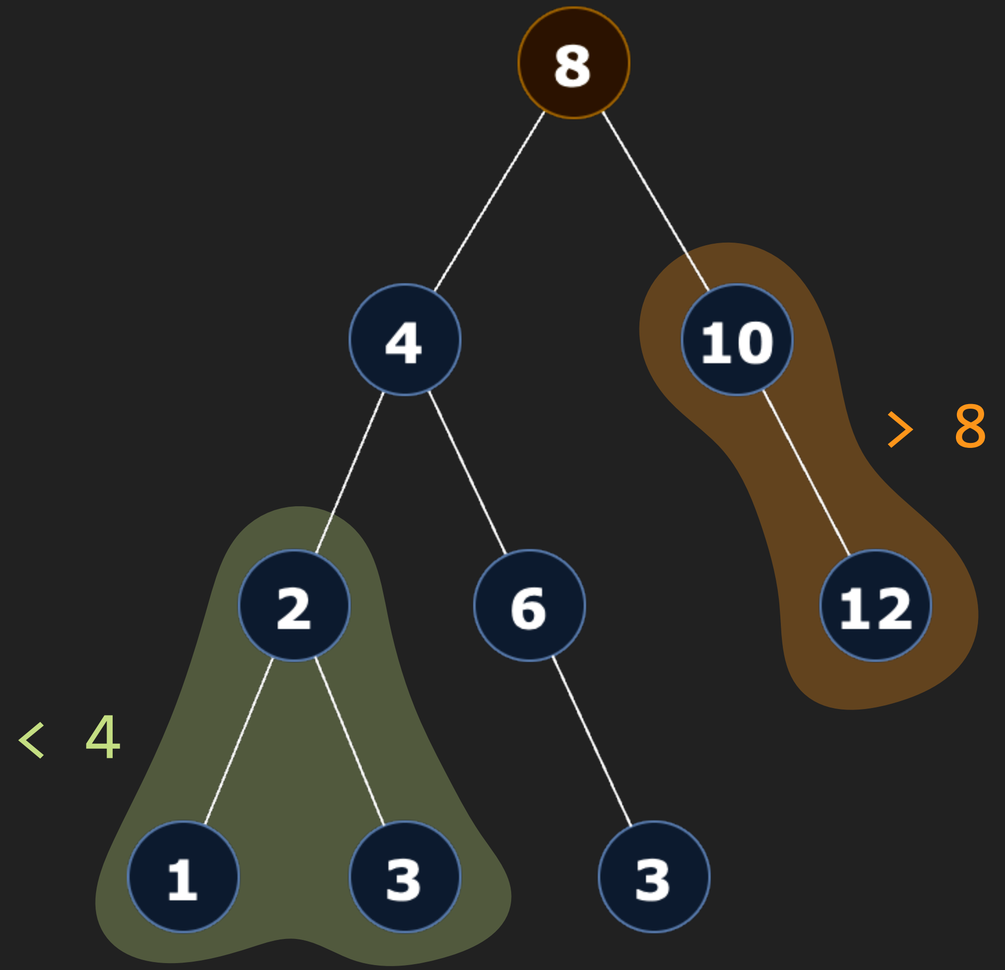
бинарное дерево поиска • BST



для любой вершины верно:

- ➡ значения в левом поддереве меньше
- ➡ значения в правом поддереве больше

бинарное дерево поиска • BST

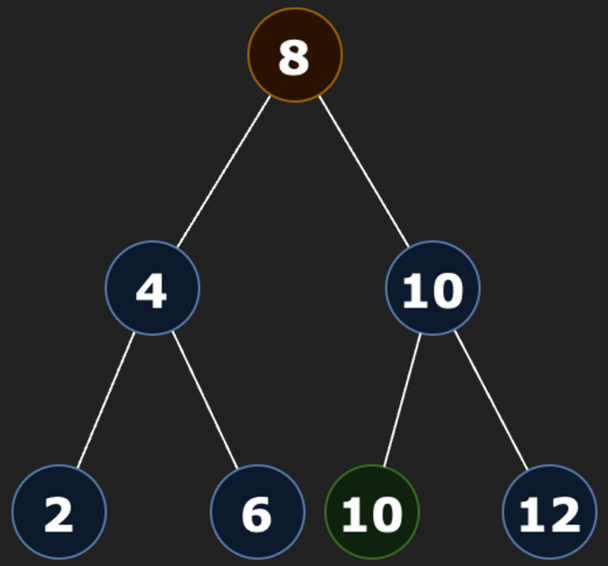


Binary Search Tree.cpp

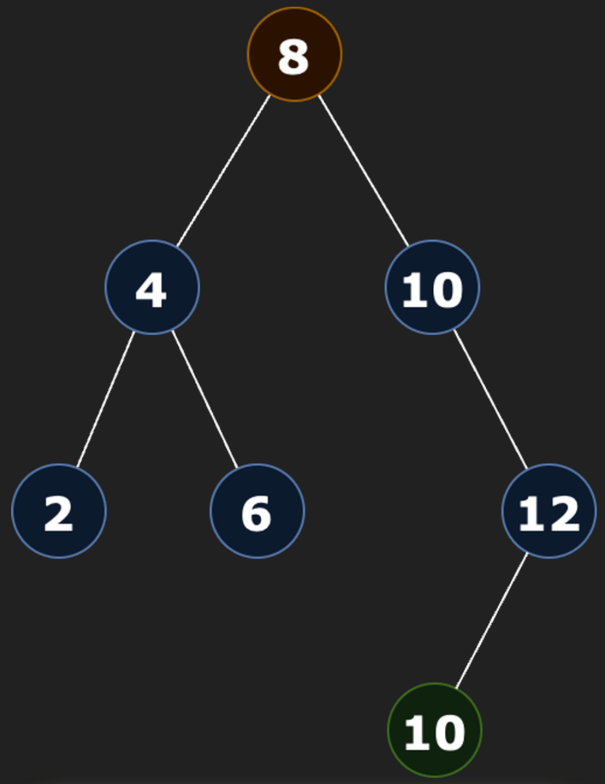
```cpp
class Tree {
public:
    Tree();
    //...

private:
    int data;
    Tree *left;
    Tree *right;
}
```

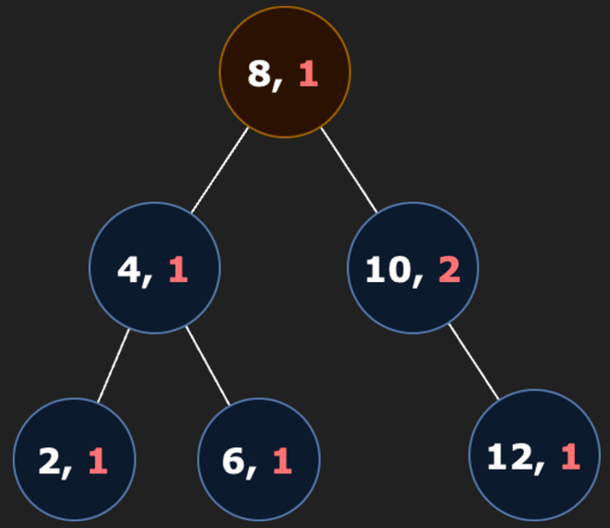
бинарное дерево поиска • BST и дубликаты



МЕНЬШЕ ИЛИ РАВНО

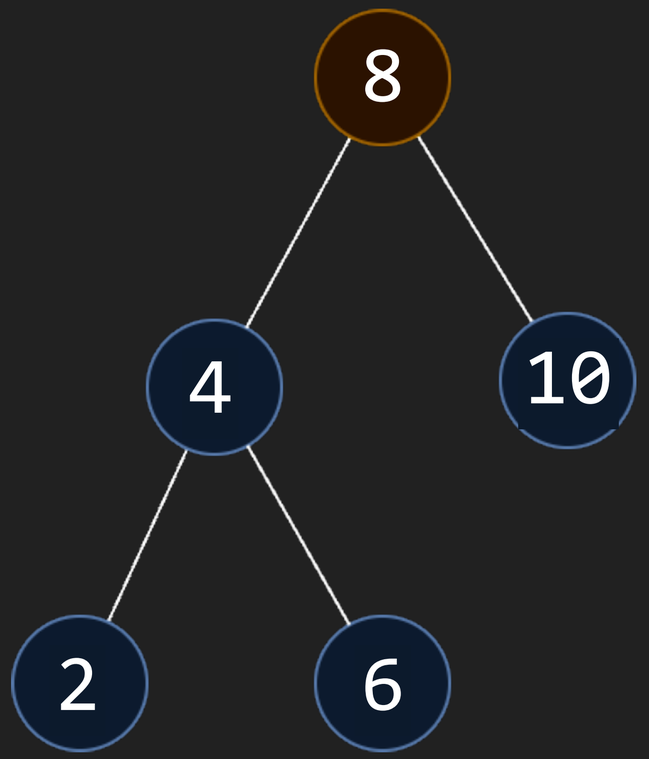


БОЛЬШЕ ИЛИ РАВНО



КРАТНОСТЬ ПОВТОРЕНИЙ

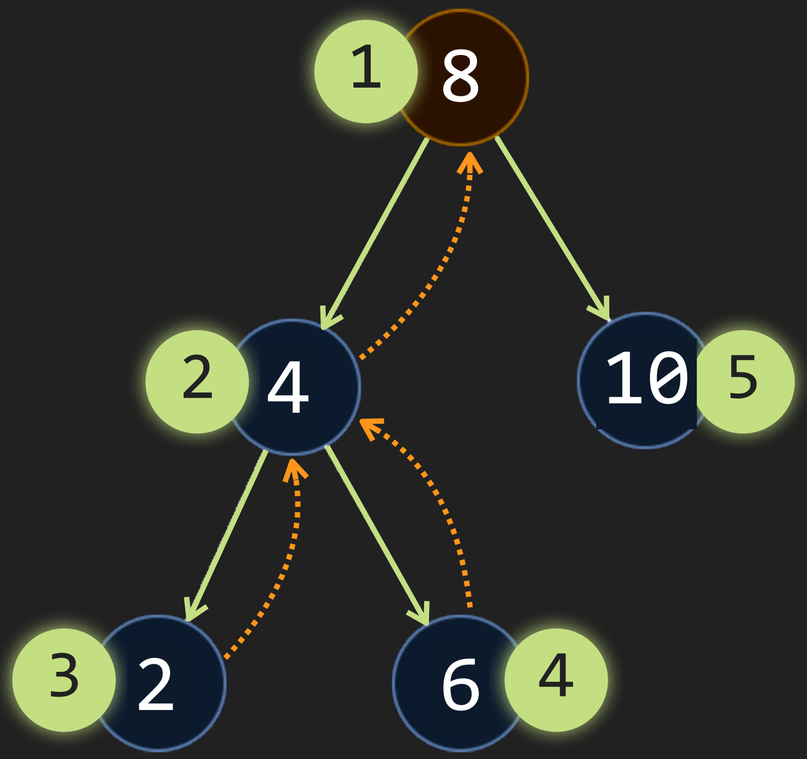
прямой обход



PRE_ORDER_TRAVERSAL.cp...

```
1   void preOrder(Node *current) {
2       if (current) {
3           std::cout << current->data;
4           preOrder(current->left);
5           preOrder(current->right);
6       }
7   }
```

прямой обход



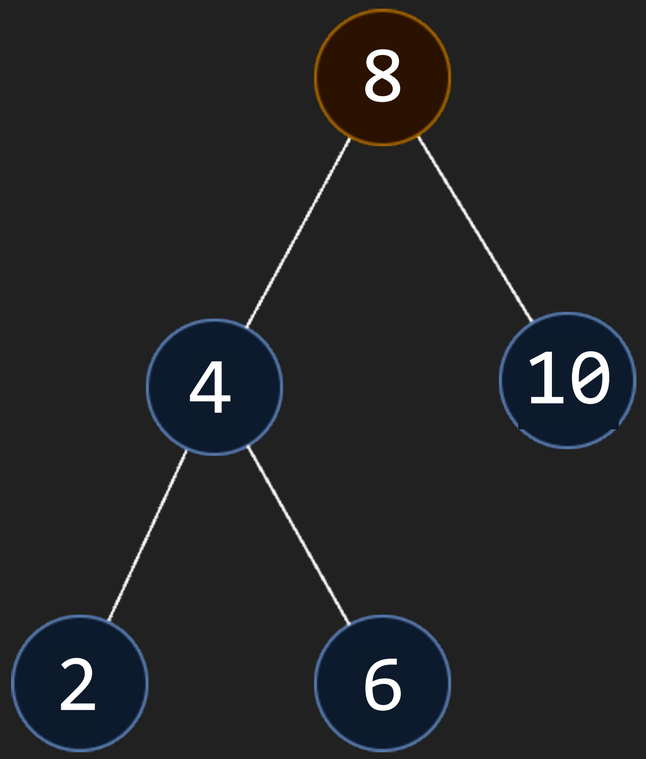
PRE_ORDER_TRAVERSAL.cp...

```cpp
void preOrder(Node *current) {
    if (current) {
        std::cout << current->data;
        preOrder(current->left);
        preOrder(current->right);
    }
}
```

НЕУПОРЯДОЧЕННЫЙ

8 4 2 6 10

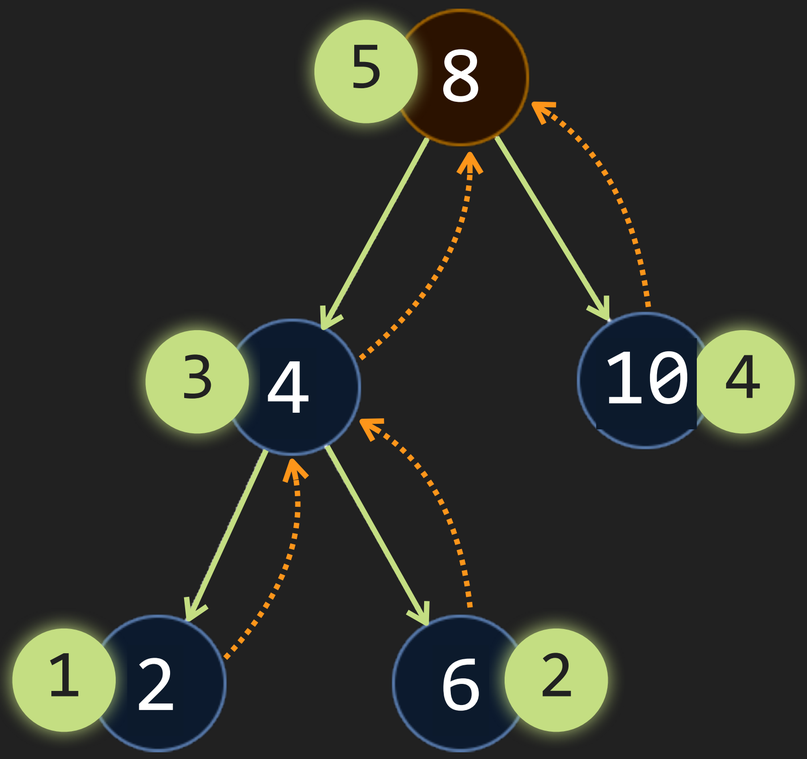
обратный обход



POST_ORDER_TRAVERSAL.cpp

```cpp
void postOrder(Node *current) {
    if (current) {
        postOrder(current->left);
        postOrder(current->right);
        std::cout << current->data;
    }
}
```

обратный обход



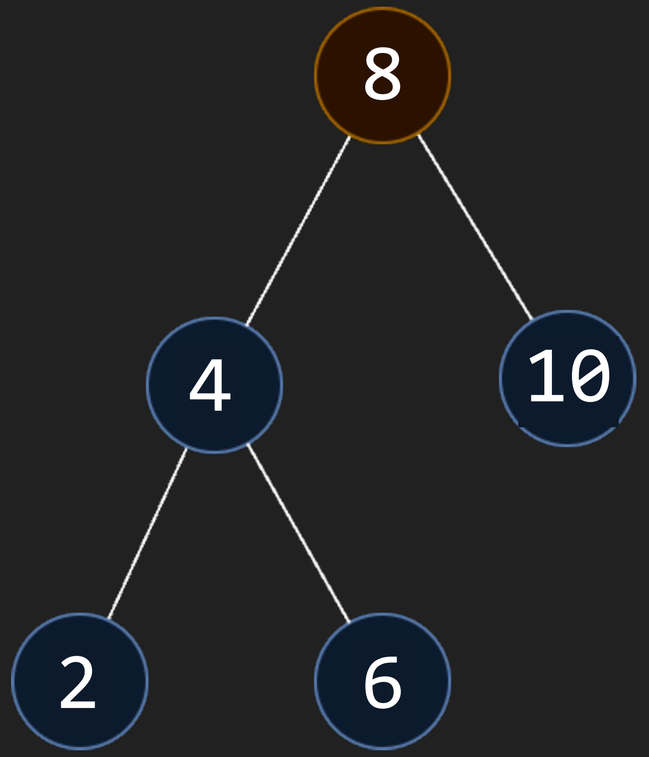
POST_ORDER_TRAVERSAL.cpp

```cpp
void postOrder(Node *current) {
    if (current) {
        postOrder(current->left);
        postOrder(current->right);
        std::cout << current->data;
    }
}
```

НЕУПОРЯДОЧЕННЫЙ

2 6 4 10 8

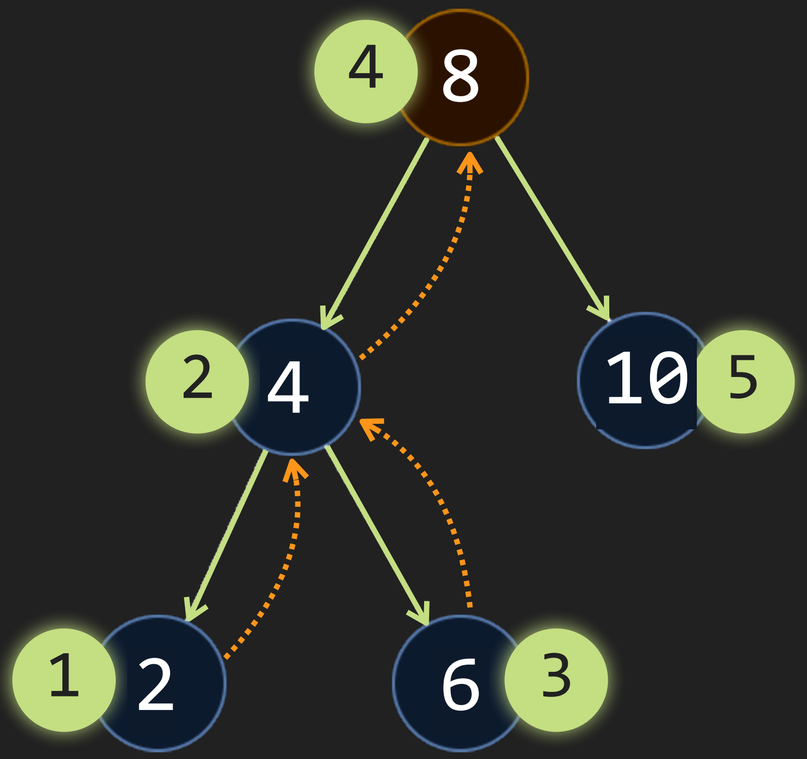
симметричный обход



IN_ORDER_TRAVERSAL.cpp

```cpp
void inOrder(Node *current) {
    if (current) {
        inOrder(current->left);
        std::cout << current->data;
        inOrder(current->right);
    }
}
```

симметричный обход



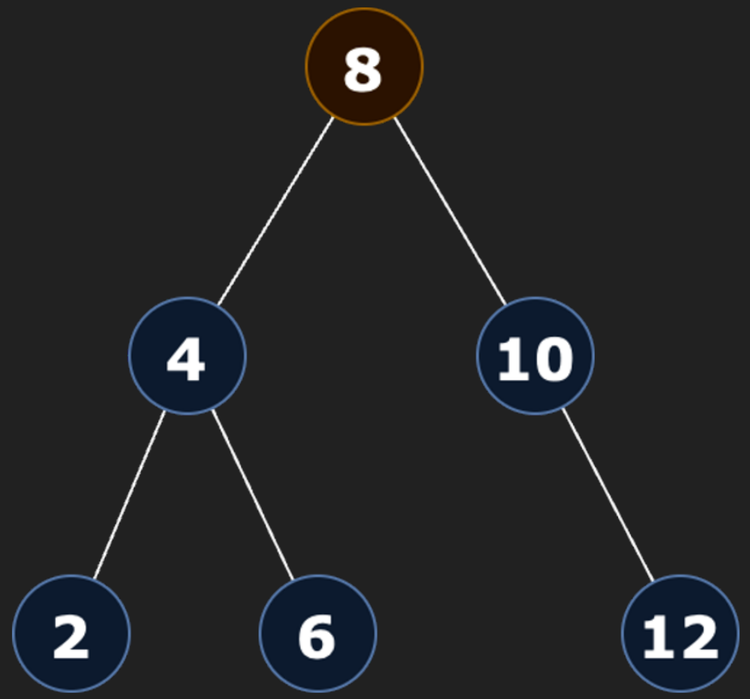
IN_ORDER_TRAVERSAL.cpp

```cpp
void inOrder(Node *current) {
    if (current) {
        inOrder(current->left);
        std::cout << current->data;
        inOrder(current->right);
    }
}
```

УПОРЯДОЧЕННЫЙ

2 4 6 8 10

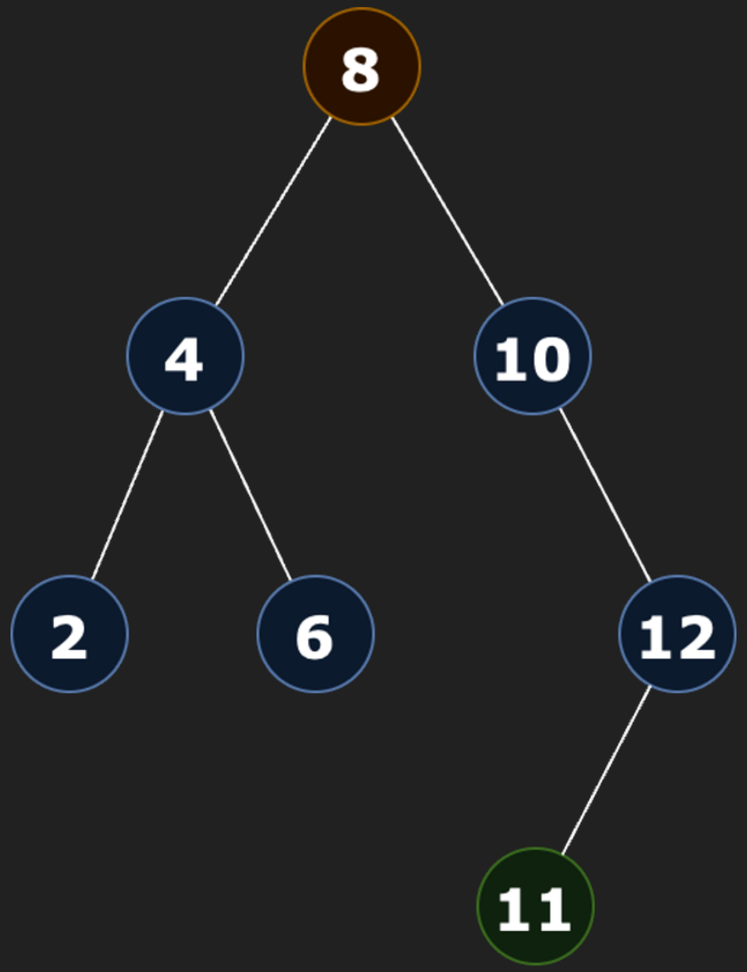
бинарное дерево поиска • `insert(key)`



`insert(11)`

место для вставки нового ключа в дерево ищем последовательным спуском в левое или правое поддерво

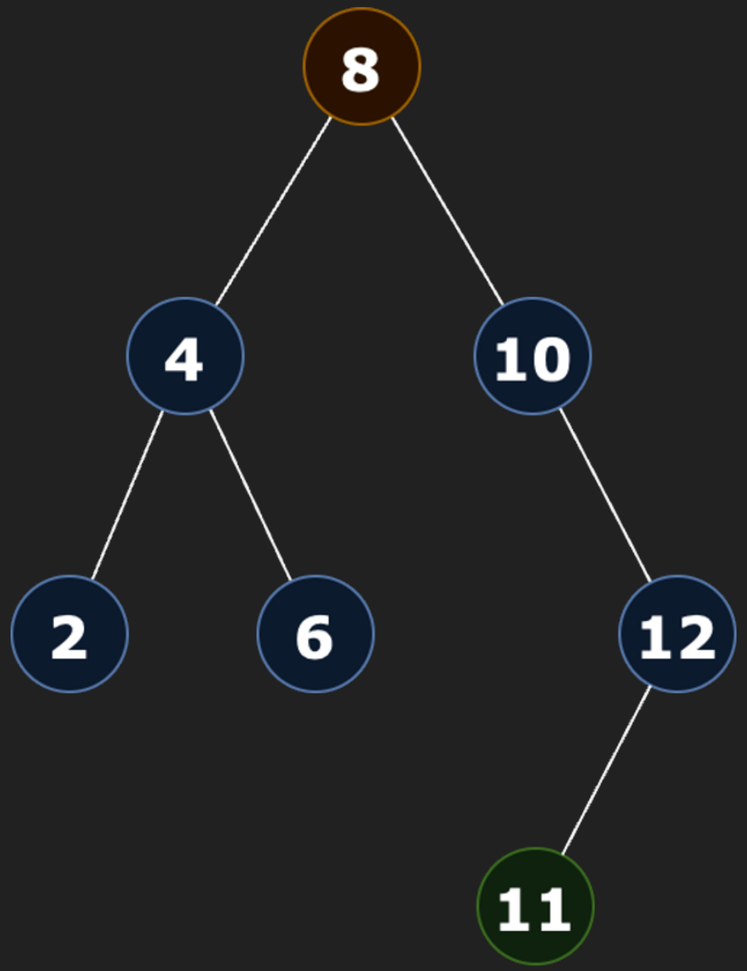
бинарное дерево поиска • `insert(key)`



`insert(11)`

место для вставки нового ключа в дерево ищем последовательным спуском в левое или правое поддерво

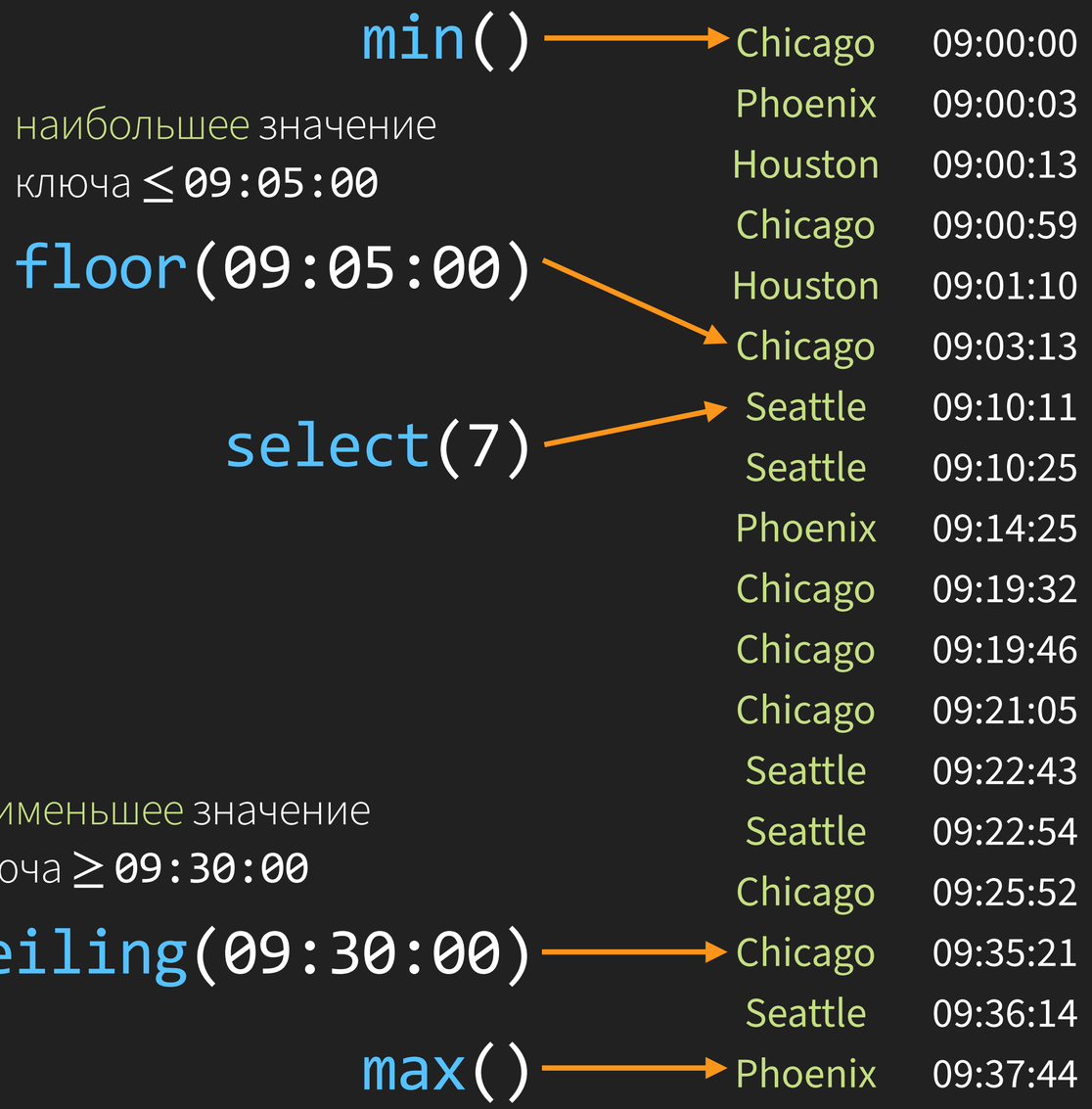
бинарное дерево поиска • `insert(key)`



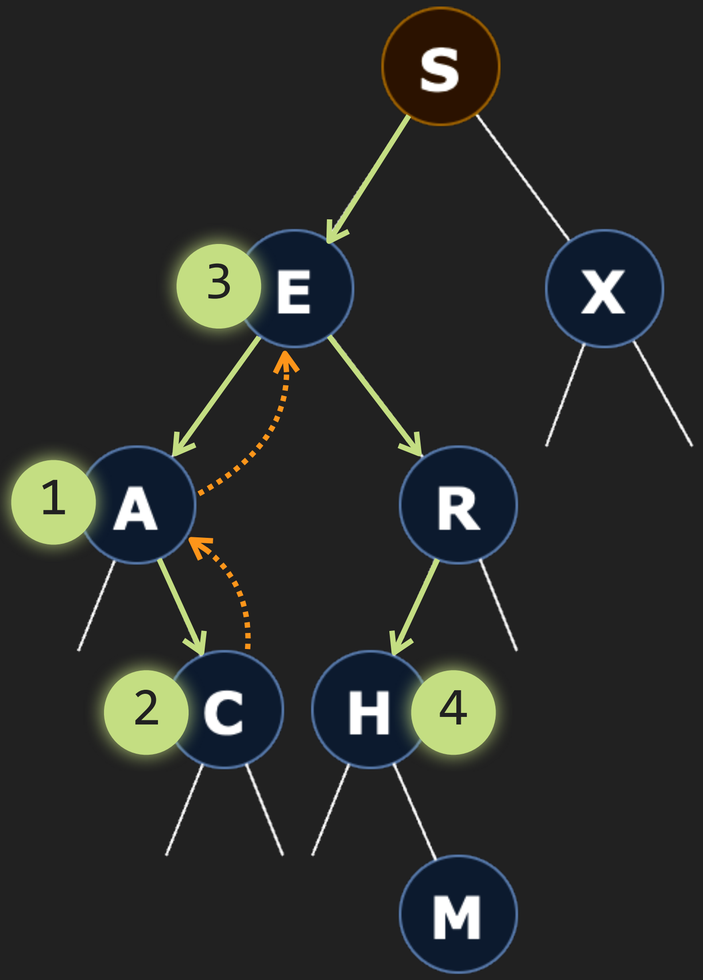
BST_Insert.cpp

```cpp
// Из ADT Sorted List
void insert(const K& key) {
    root = insert(root, const K& key);
}

// Реализация во внутреннем BST
Node *insert(r, const K& key) {
    if (!r) return new Tree(key);
    else if (key < r->data) {
        r->left = insert(r->left, key);
    }
    else {
        r->right = insert(r->right, key);
    }
    return r;
}
```


min()
наибольшее значение
ключа ≤ 09:05:00
floor(09:05:00)
select(7)
наименьшее значение
ключа ≥ 09:30:00
ceiling(09:30:00)
max()
Chicago 09:00:00
Phoenix 09:00:03
Houston 09:00:13
Chicago 09:00:59
Houston 09:01:10
Chicago 09:03:13
Seattle 09:10:11
Seattle 09:10:25
Phoenix 09:14:25
Chicago 09:19:32
Chicago 09:19:46
Chicago 09:21:05
Seattle 09:22:43
Seattle 09:22:54
Chicago 09:25:52
Chicago 09:35:21
Seattle 09:36:14
Phoenix 09:37:44

бинарное дерево поиска • `select(order)`

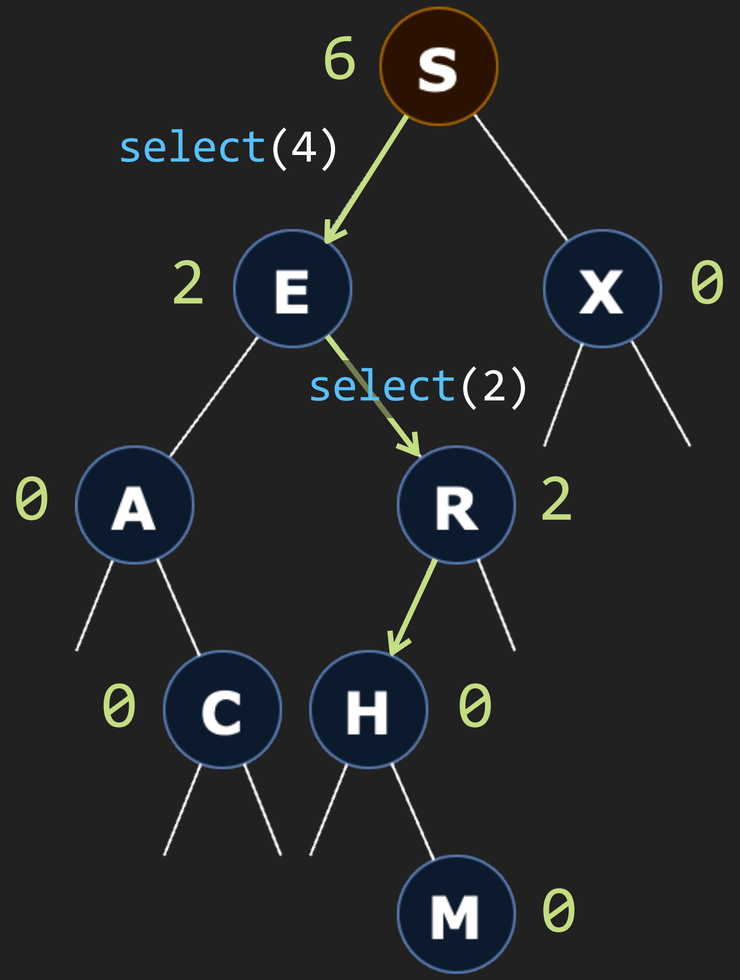


`select(4)`

ПОДХОД 1 — обход в глубину

симметричный обход дерева `inOrder` с дополнительным подсчетом посещенных вершин

бинарное дерево поиска • `select(order)`

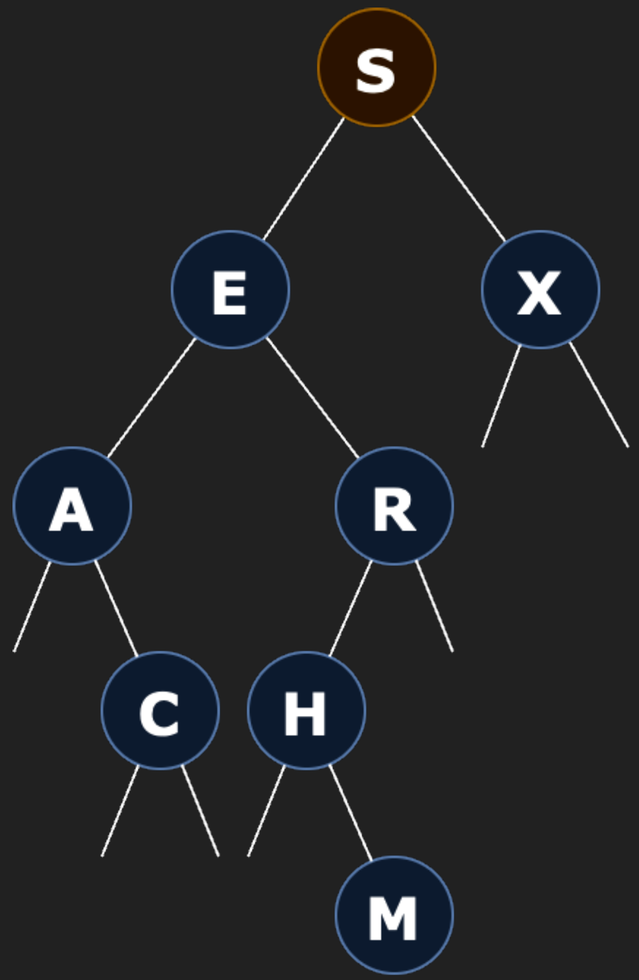


`select(4)`

ПОДХОД 2 — ранжирование

дополнение вершины дерева информацией о количестве вершин в левом поддереве

бинарное дерево поиска • floor(key)

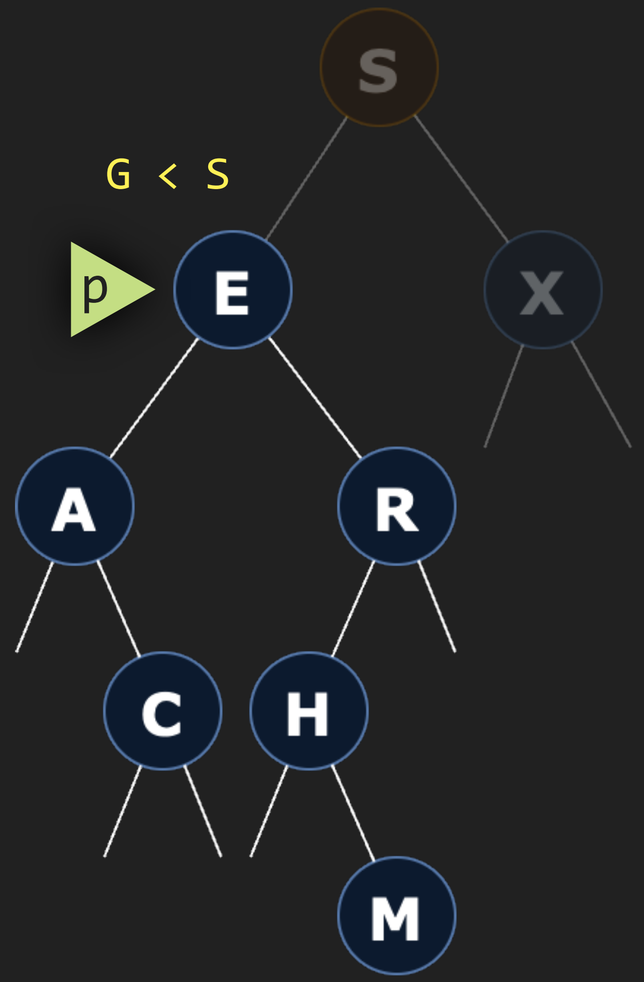


floor(G)

наибольший ключ ≤ G

1. key = node->data
   floor(key) совпадает с node->data
2. key < node->data
   floor(key) в левом поддереве
3. key < node->data
   floor(key) в правом* поддереве

бинарное дерево поиска • floor(key)

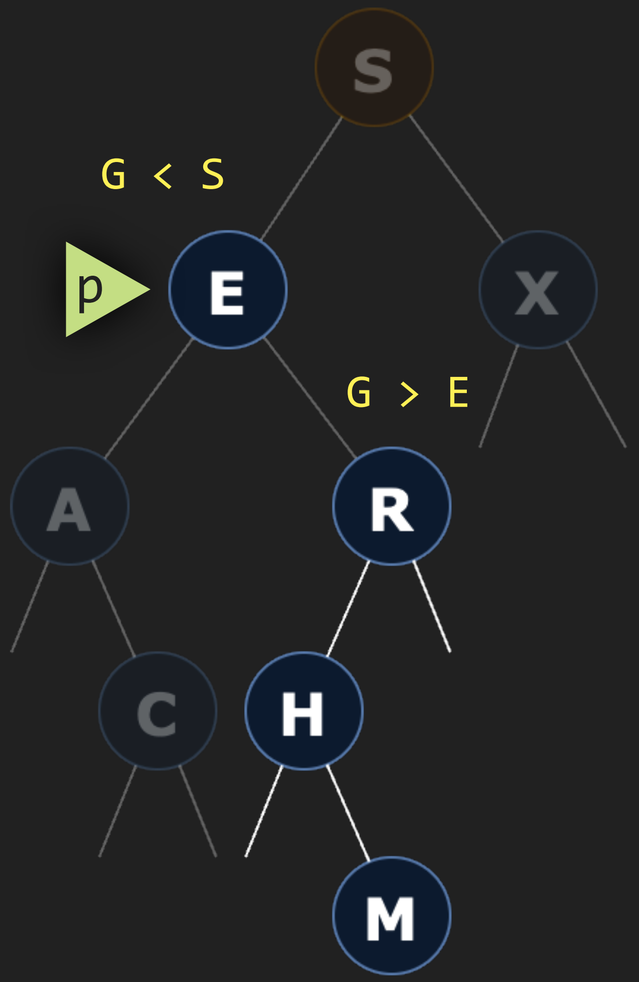


floor(G)

наибольший ключ ≤ G

1. key = node->data
   floor(key) совпадает с node->data
2. key < node->data
   floor(key) в левом поддереве
3. key < node->data
   floor(key) в правом* поддереве

бинарное дерево поиска • floor(key)



floor(G)

наибольший ключ ≤ G

1. key = node->data
   floor(key) совпадает с node->data
2. key < node->data
   floor(key) в левом поддереве
3. key < node->data
   floor(key) в правом* поддереве

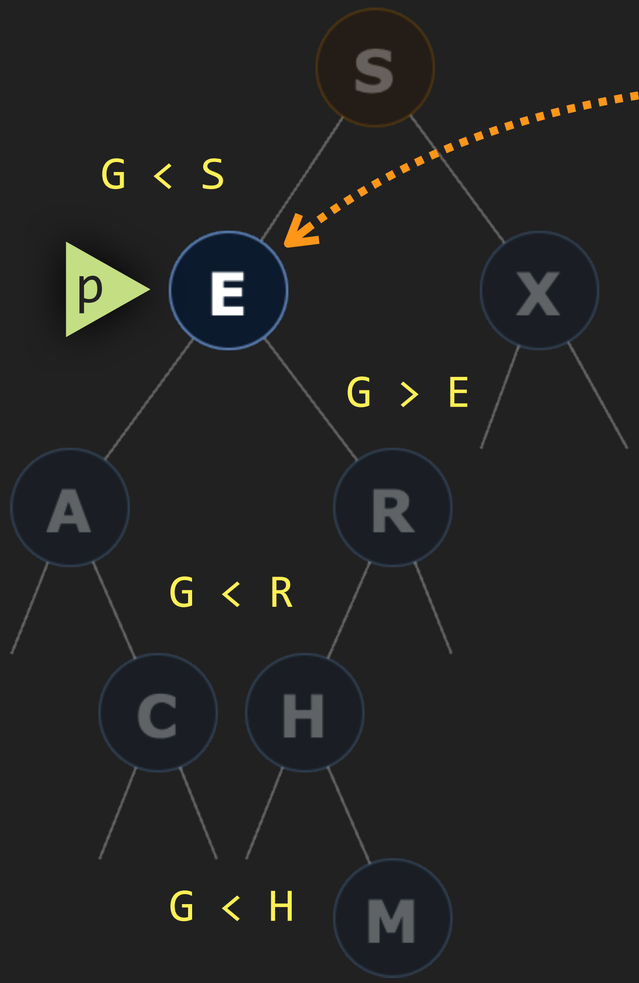
бинарное дерево поиска • floor(key)

floor(G)

наибольший ключ ≤ G

1. key = node->data
   floor(key) совпадает с node->data
2. key < node->data
   floor(key) в левом поддереве
3. key < node->data
   floor(key) в правом* поддереве

бинарное дерево поиска • floor(key)



floor(G)

наибольший ключ ≤ G

1. key = node->data
   floor(key) совпадает с node->data
2. key < node->data
   floor(key) в левом поддереве
3. key < node->data
   floor(key) в правом* поддереве

бинарное дерево поиска • `floor(key)`

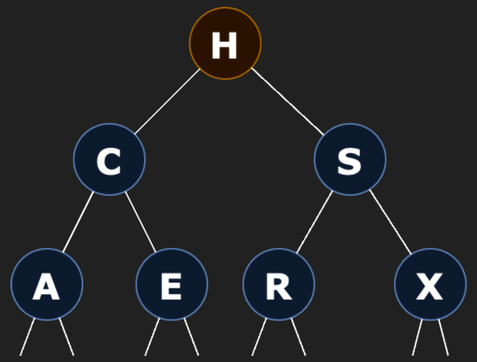
BST_Floor.cpp

```
1   Node *floor(Node *r, Key key) {
2       if (!r) return nullptr;
3
4       if (key == r->data) return r;
5       if (key < r->data) return floor(r->left, key)
6
7       Node *t = floor(x->right, key);
8       if (!t) return t;
9       else return r;
10  }
```

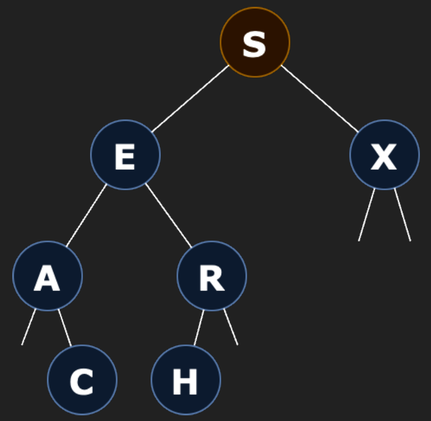
СВОЙСТВО

Форма бинарного дерева поиска полностью определяется порядком вставки элементов.

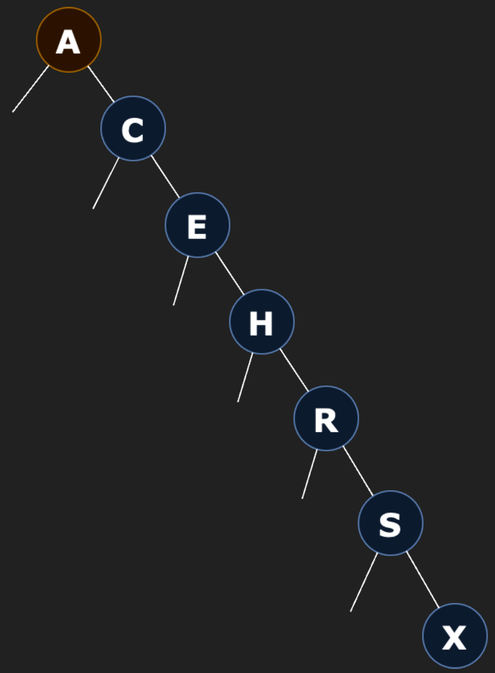
H	C	A	E	S	R	X



S	E	A	R	C	H	X



A	C	E	H	R	S	X



ТЕОРЕМА

Сложность основных операций с бинарным деревом поиска определяется его высотой.

стремимся к оптимальной высоте дерева — $O(\log n)$!

как исправить дисбаланс и не нарушить порядок