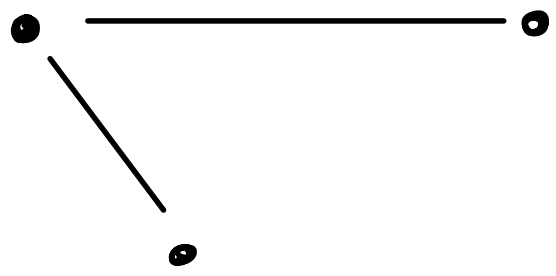
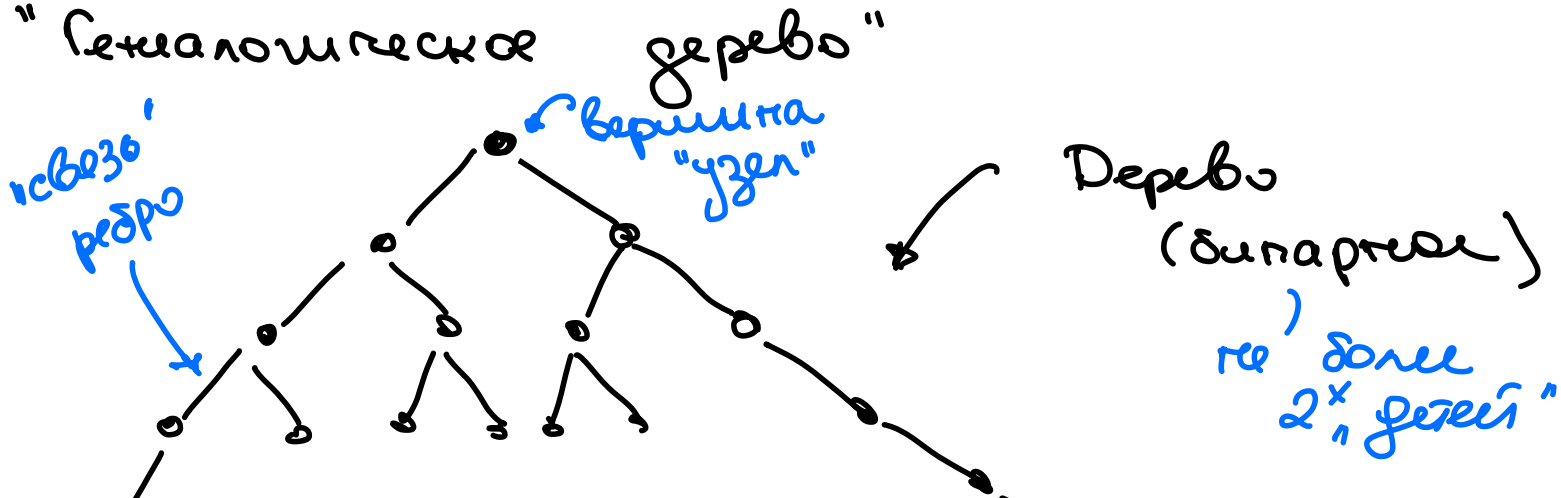
Бинарное дерево поиска (BST)

- Список
- Куча

Математические факты

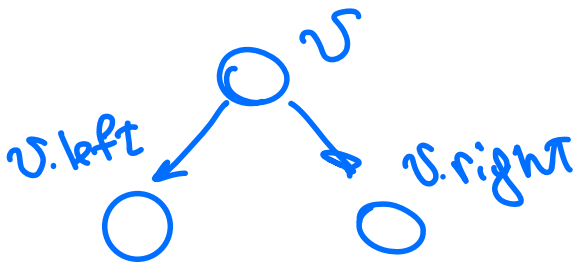
Опр (Дерево из мат-ки) граф набор вершин и ребер, в которых нет цикла.





Опр Путь в дереве – набор вершин, которые следуют друг за другом (по ребрам)

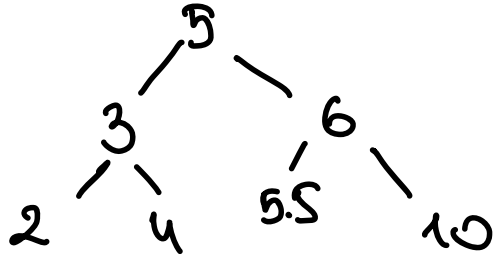
Опр Бинарное дерево поиска (Binary Search Tree) – это структура данных, представляющая собой дерево, состоящее из узлов и "связей". При этом выполнено свойство:



→ ∀ v - вершина не более 2х "детей" (смежных вершин)

→ v.left < v < v.right (поддерево)

Пример:



одинаковых нет!

Вопрос: Где лежит максимальный : самый правый

Где лежит минимальный : левый

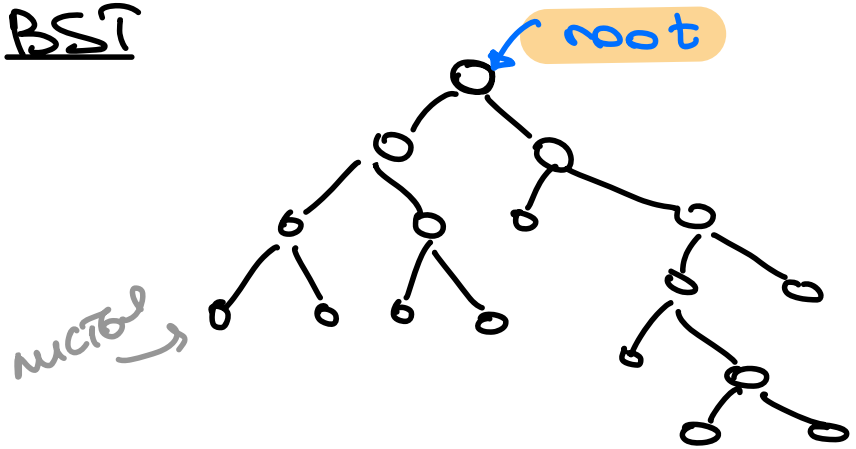
Методы:

- findMax()
- findMin()
- removeMax()
- removeMin()
- search() - проверка наличия элемента в BST
- push() - добавление элемента в дерево
- remove() - удаление ключа в BST

ключ

O(H)

BST



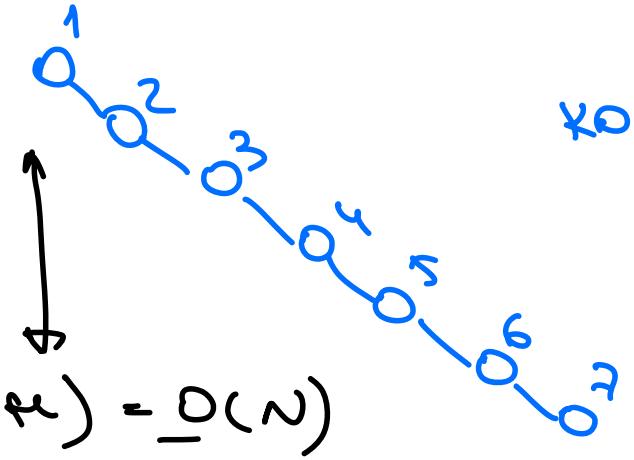
H - высота дерева

максимальная длина пути из вершины в лист.

Худший случай:

N - кол-во элементов

операции выполняются за $O(H) = O(N)$

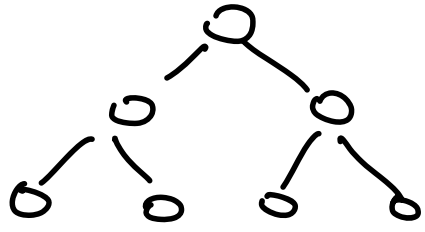


Корректное BST? - ДА

H = N (высота)

Хотим исправить: Уменьшить высоту

Сбалансированное дерево.



$H = \log_2 N$

Классификация BST:

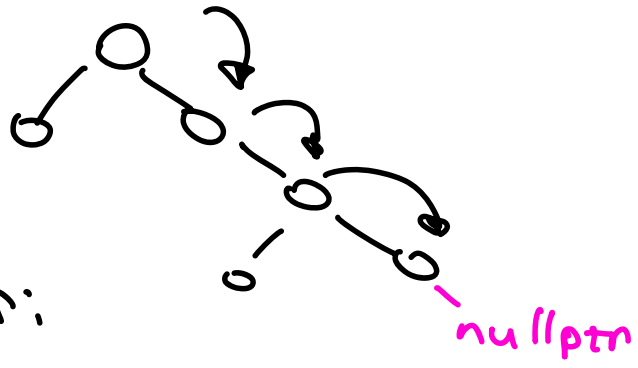
- несбалансированное → плохо
- Декартово
- AVL - дерево
- K-2. деревья
- SPLAY - дерево

→ хорошо

Алгоритм

Узел:

```
struct Node {
    int value;          // то, что нужно
    Node* left = nullptr;
    Node* right = nullptr;
// опционально // Node* parent = nullptr; // родитель.
};
```



рекурсивный алгоритм

```
findMax ( Node* current ) {
    если current->right != nullptr:
        → return findMax (current->right);
    иначе
        → return current->value;
}
```

```
findMax ( Node* root )?
    Node* runner = root;
    while (runner->right != nullptr)?
        runner = runner->right;
    }
    return runner->value;
}
```

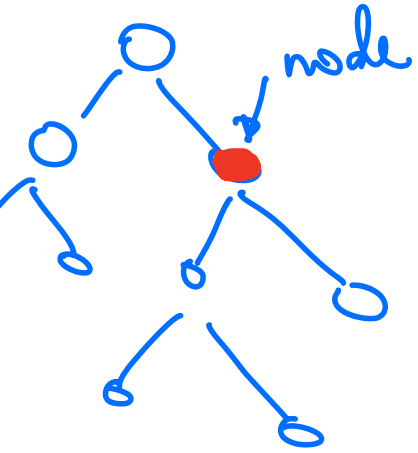
Поиск ключа

Алгоритм

Node* Search (Key , Node* node)?

если node == nullptr → return false
// return nullptr

если key == node->value
↳ return true
// return node;



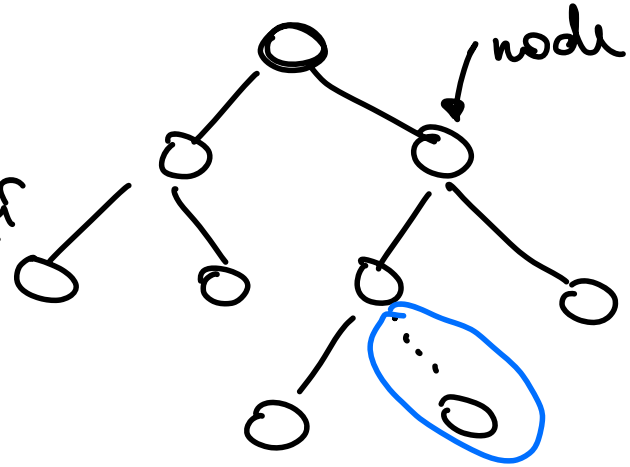
← только в привязке к задаче

иначе если key > node->value:
↳ return Search (key, node->right)

```
        иначе если   key < node → value;
            ↳ return  Search ( key, node → left )
}
```

Добавление элемента

```
Push ( key, Node* node ) {
    если  node == nullptr
    // или - нужно root
    если  key == node → value;
          return true
    если  key > node → value :
```




```
        если   node → right == nullptr:
            ↳ new_node = new Node:
              new_node → value = key
        //    new_node → parent = node
              node → right = new_node
              return true;
```

иначе

return Push(key, node→right);

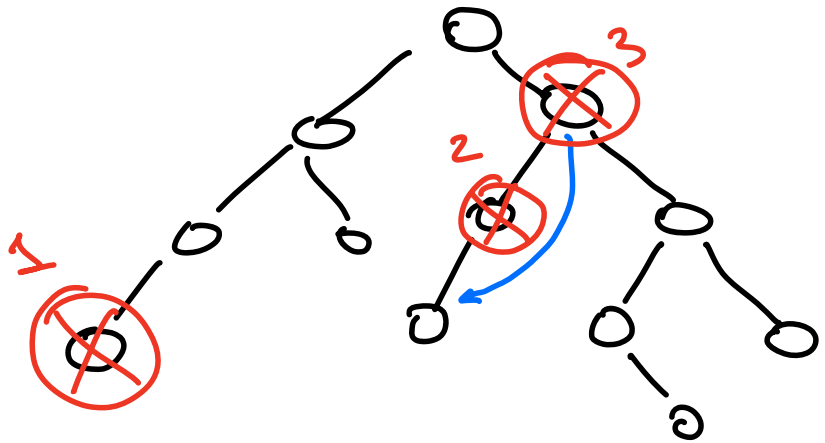
если key < node→value:

аналогично

}

Удаление элемента

0) → node = Search(key) - приватный



1. Удаление листа
   - удалить
   parent→ml = nullptr

2. Удаление вершины с одним ребенком
   - удалить
   parent→ml = ml→ребенок

3. Удаление вершины с двумя детьми

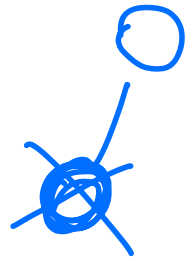
right = node→right

del_val = removeMin(right)

```
node -> value = del_value;
```

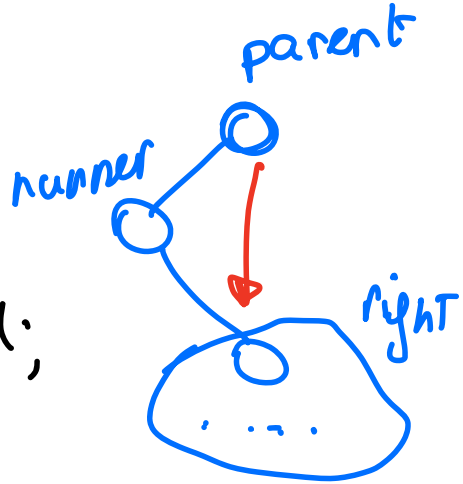
removeMin(key, Node* node)?

```
Node* runner = root;
while (runner -> left != nullptr){
        runner = runner -> left;
}
```



```
else    runner -> right == nullptr:
    b   runner -> parent -> left = nullptr;
        value = runner -> value;
        delete runner
        return value
```

```
else
        runner -> parent -> left =
                        runner -> right;
    value = runner -> value;
```



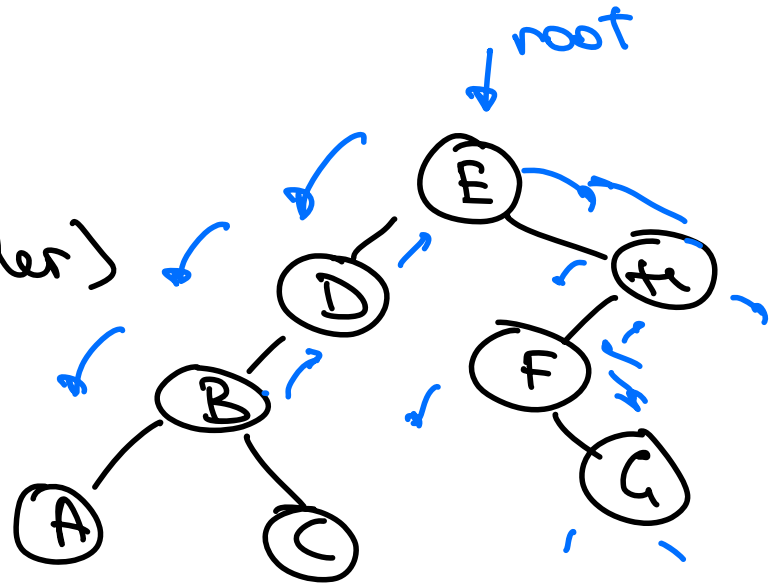
delete runner
return value

}

Обход дерева (в глубину)

DFS

1) сверху-вниз (pre-order)



NLR - левое - правое

E D B A C H F G

```
PreOrder ( node )
{
    print (node → value)
    PreOrder ( node → left )
    PreOrder ( node → right )
}
```

2) Post-order (снизу вверх)

левое - правое - мы

A C B D G F H E

3) In-order (слева - направо)

лево - мы - право

Удаление BST

Освободить память → Post Order обход

delete node