# Бинарные деревья поиска

Denis Bakin

# Зачем нужны деревья поиска?

Хотим структуру, которая умеет быстро:

- искать элемент по ключу
- добавлять новый элемент
- удалять элемент

# Зачем нужны деревья поиска?

Хотим структуру, которая умеет быстро:

- искать элемент по ключу
- добавлять новый элемент
- удалять элемент

И всё это за $O(\log n)$

# Зачем нужны деревья поиска?

Хотим структуру, которая умеет быстро:

- искать элемент по ключу
- добавлять новый элемент
- удалять элемент

И всё это за $O(\log n)$

Массив так не умеет: поиск за $O(\log n)$, но вставка/удаление за $O(n)$

# Напоминание: деревья

**Дерево** — набор вершин и рёбер, в котором от корня до любой вершины ведёт единственный путь

# Напоминание: деревья

**Дерево** — набор вершин и рёбер, в котором от корня до любой вершины ведёт единственный путь

- **Родитель** — вершина, из которой ведёт ребро
- **Дети** — вершины, в которые ведут рёбра
- **Лист** — вершина без детей
- **Глубина** вершины — расстояние от корня
- **Высота** дерева — максимальная глубина

# Бинарное дерево поиска

**BST** (*binary search tree*) — дерево, в котором:

- у каждой вершины не более двух детей
- у каждой вершины есть **ключ**

# Бинарное дерево поиска

**BST** (*binary search tree*) — дерево, в котором:

- у каждой вершины не более двух детей
- у каждой вершины есть **ключ**

Главное свойство:

- в **левом** поддереве все ключи **не больше** ключа вершины
- в **правом** поддереве все ключи **больше** ключа вершины

# Бинарное дерево поиска

**BST** (*binary search tree*) — дерево, в котором:

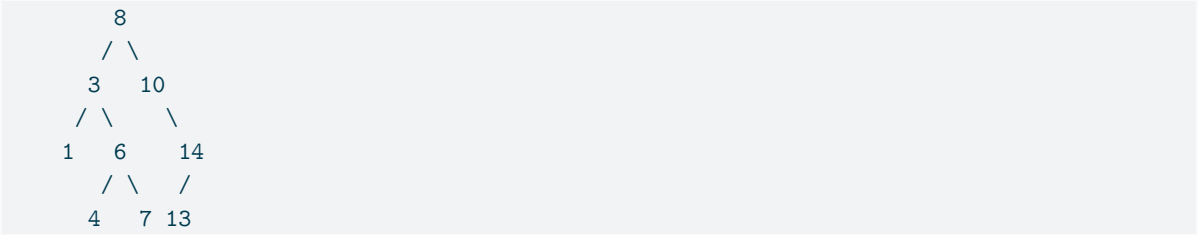
- у каждой вершины не более двух детей
- у каждой вершины есть **ключ**

Главное свойство:

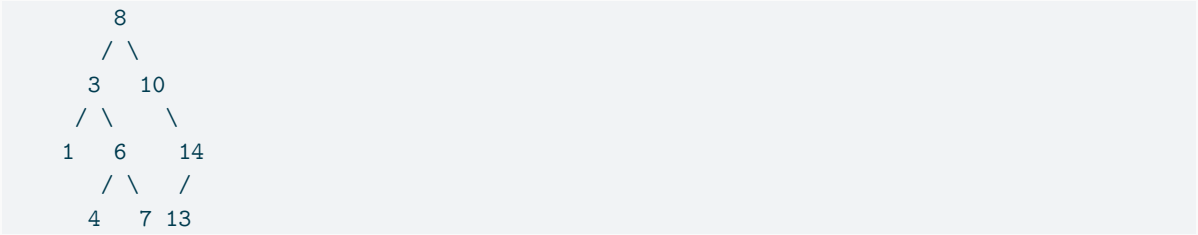
- в **левом** поддереве все ключи **не больше** ключа вершины
- в **правом** поддереве все ключи **больше** ключа вершины

Это свойство выполняется рекурсивно для каждой вершины

# Пример BST



# Пример BST




Все ключи слева от 8: $\{1, 3, 4, 6, 7\}$ — все $\leq 8$

Все ключи справа от 8: $\{10, 13, 14\}$ — все $> 8$

# Представление в памяти

```
struct Node {
    int key;
    Node* left = nullptr;
    Node* right = nullptr;
};
```

# Представление в памяти

```
struct Node {
    int key;
    Node* left = nullptr;
    Node* right = nullptr;
};
```

- Каждая вершина — объект с ключом и двумя указателями
- Пустое поддерево — `nullptr`
- В отличие от кучи, массивом хранить неудобно: дерево может быть несбалансированным

# Поиск элемента

Начинаем с корня. На каждом шаге:

- $x =$ ключ вершины $\Rightarrow$ нашли
- $x <$ ключ $\Rightarrow$ идём налево
- $x >$ ключ $\Rightarrow$ идём направо

# Поиск элемента

Начинаем с корня. На каждом шаге:

- $x =$ ключ вершины $\Rightarrow$ нашли
- $x <$ ключ $\Rightarrow$ идём налево
- $x >$ ключ $\Rightarrow$ идём направо

Дошли до `nullptr` $\Rightarrow$ элемента нет

# Поиск элемента

Начинаем с корня. На каждом шаге:

- $x =$ ключ вершины $\Rightarrow$ нашли
- $x <$ ключ $\Rightarrow$ идём налево
- $x >$ ключ $\Rightarrow$ идём направо

Дошли до `nullptr` $\Rightarrow$ элемента нет

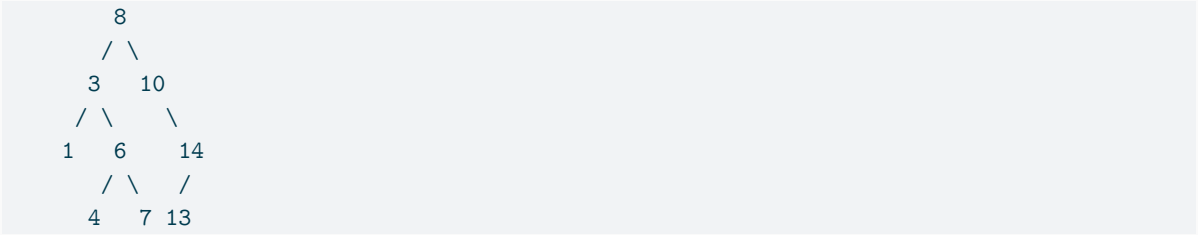
По сути это **двоичный поиск**, только вместо массива — дерево

# Поиск: код

```
Node* Find(int key, Node* root) {
    if (root == nullptr) {
        return nullptr;
    }
    if (key < root->key) {
        return Find(key, root->left);
    }
    if (key > root->key) {
        return Find(key, root->right);
    }
    return root;
}
```

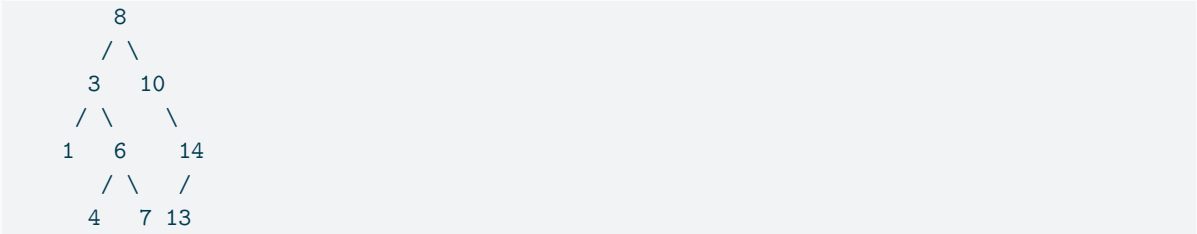
# Живой пример: поиск

Ищем ключ 6 в дереве:



# Живой пример: поиск

Ищем ключ 6 в дереве:




$6 < 8 \rightarrow$ влево $\rightarrow 6 > 3 \rightarrow$ вправо $\rightarrow 6 = 6$ ✓

# Вставка элемента

Спускаемся по дереву, как при поиске. Когда дошли до `nullptr` — создаём новую вершину

# Вставка элемента

Спускаемся по дереву, как при поиске. Когда дошли до `nullptr` — создаём новую вершину

```
Node* Insert(int key, Node* root) {
    if (root == nullptr) {
        return new Node{key};
    }
    if (key < root->key) {
        root->left = Insert(key, root->left);
    } else if (key > root->key) {
        root->right = Insert(key, root->right);
    }
    return root;
}
```

# Вставка элемента

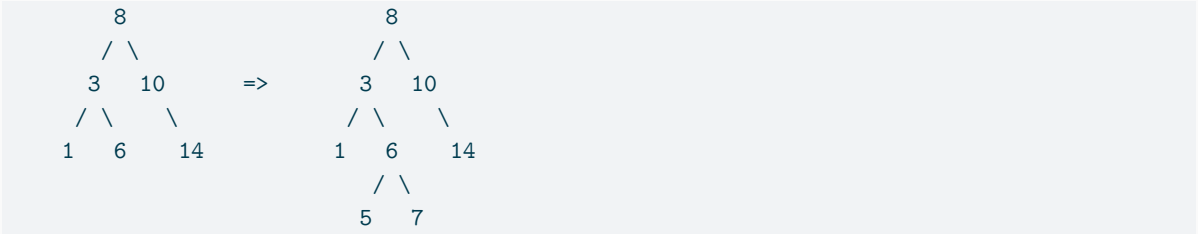
Спускаемся по дереву, как при поиске. Когда дошли до `nullptr` — создаём новую вершину

```
Node* Insert(int key, Node* root) {
    if (root == nullptr) {
        return new Node{key};
    }
    if (key < root->key) {
        root->left = Insert(key, root->left);
    } else if (key > root->key) {
        root->right = Insert(key, root->right);
    }
    return root;
}
```

Новый элемент всегда становится **листом**

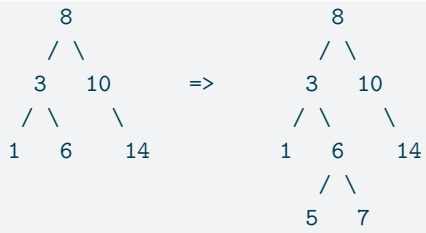
# Живой пример: вставка

Вставляем 5 в дерево:



# Живой пример: вставка

Вставляем 5 в дерево:




$5 < 8 \to 5 > 3 \to 5 < 6 \to$ `nullptr` $\to$ создаём вершину

# Удаление: три случая

Удаление — самая нетривиальная операция

# Удаление: три случая

Удаление — самая нетривиальная операция

**Случай 1:** вершина — лист

$\Rightarrow$ просто удаляем

# Удаление: три случая

Удаление — самая нетривиальная операция

**Случай 1:** вершина — лист

$\Rightarrow$ просто удаляем

**Случай 2:** у вершины один ребёнок

$\Rightarrow$ заменяем вершину на этого ребёнка

# Удаление: три случая

Удаление — самая нетривиальная операция

**Случай 1:** вершина — лист

$\Rightarrow$ просто удаляем

**Случай 2:** у вершины один ребёнок

$\Rightarrow$ заменяем вершину на этого ребёнка

**Случай 3:** у вершины два ребёнка

$\Rightarrow$ ???

# Удаление: случай с двумя детьми

Находим **минимум правого поддерева** (следующий по порядку элемент)

# Удаление: случай с двумя детьми

Находим **минимум правого поддерева** (следующий по порядку элемент)

- Копируем его ключ в удаляемую вершину

# Удаление: случай с двумя детьми

Находим **минимум правого поддерева** (следующий по порядку элемент)

- Копируем его ключ в удаляемую вершину
- Рекурсивно удаляем этот минимум из правого поддерева

# Удаление: случай с двумя детьми

Находим **минимум правого поддерева** (следующий по порядку элемент)

- Копируем его ключ в удаляемую вершину
- Рекурсивно удаляем этот минимум из правого поддерева
- У минимума правого поддерева нет левого ребёнка $\Rightarrow$ удаление простое

# Удаление: код

```
Node* Remove(int key, Node* root) {
    if (root == nullptr) return nullptr;

    if (key < root->key) {
        root->left = Remove(key, root->left);
    } else if (key > root->key) {
        root->right = Remove(key, root->right);
    } else {
        if (root->left == nullptr) {
            Node* child = root->right;
            delete root;
            return child;
        }
        if (root->right == nullptr) {
            Node* child = root->left;
            delete root;
            return child;
        }
```

# Обходы дерева

Три классических обхода бинарного дерева:

# Обходы дерева

Три классических обхода бинарного дерева:

- **Inorder**: левое поддерево $\rightarrow$ вершина $\rightarrow$ правое поддерево
- **Preorder**: вершина $\rightarrow$ левое поддерево $\rightarrow$ правое поддерево
- **Postorder**: левое поддерево $\rightarrow$ правое поддерево $\rightarrow$ вершина

# Обходы дерева

Три классических обхода бинарного дерева:

- **Inorder**: левое поддерево $\rightarrow$ вершина $\rightarrow$ правое поддерево
- **Preorder**: вершина $\rightarrow$ левое поддерево $\rightarrow$ правое поддерево
- **Postorder**: левое поддерево $\rightarrow$ правое поддерево $\rightarrow$ вершина

Для BST **inorder-обход** даёт ключи **в отсортированном порядке**

# Inorder-обход: код

```
void Inorder(Node* root, std::vector<int>& result) {
    if (root == nullptr) return;
    Inorder(root->left, result);
    result.push_back(root->key);
    Inorder(root->right, result);
}
```

# Inorder-обход: код

```
void Inorder(Node* root, std::vector<int>& result) {
    if (root == nullptr) return;
    Inorder(root->left, result);
    result.push_back(root->key);
    Inorder(root->right, result);
}
```

Для дерева из примера: $1, 3, 4, 6, 7, 8, 10, 13, 14$

# Inorder-обход: код

```
void Inorder(Node* root, std::vector<int>& result) {
    if (root == nullptr) return;
    Inorder(root->left, result);
    result.push_back(root->key);
    Inorder(root->right, result);
}
```

Для дерева из примера: $1, 3, 4, 6, 7, 8, 10, 13, 14$

Обратный порядок — поменять местами рекурсивные вызовы

# Сложность операций

Все операции работают за $O(h)$, где $h$ — высота дерева

# Сложность операций

Все операции работают за $O(h)$, где $h$ — высота дерева

Какой может быть высота?

# Сложность операций

Все операции работают за $O(h)$, где $h$ — высота дерева

Какой может быть высота?

| | Лучший случай | Худший случай |
|---|---|---|
| Высота | $O(\log n)$ | $O(n)$ |

# Когда всё плохо: «бамбук»

Если вставлять элементы в отсортированном порядке:

```
1                    Вместо дерева -
 \                   получается цепочка
  2
   \                 h = n - 1
    3
     \               Все операции за O(n)
      4
       \
        5
```

# Когда всё плохо: «бамбук»

Если вставлять элементы в отсортированном порядке:

```
1                    Вместо дерева -
 \                   получается цепочка
  2
   \                 h = n - 1
    3
     \               Все операции за O(n)
      4
       \
        5
```

Все преимущества BST теряются

# Сбалансированные деревья

Хотим **гарантировать** $h = O(\log n)$

# Сбалансированные деревья

Хотим **гарантировать** $h = O(\log n)$

Идея: при вставке и удалении поддерживать инвариант, ограничивающий высоту

# Сбалансированные деревья

Хотим **гарантировать** $h = O(\log n)$

Идея: при вставке и удалении поддерживать инвариант, ограничивающий высоту

- **AVL-дерево** — высоты поддеревьев отличаются не более чем на 1
- **Красно-чёрное дерево** — правила на цвета вершин гарантируют $h \leq 2\log_2(n+1)$

# Сбалансированные деревья

Хотим **гарантировать** $h = O(\log n)$

Идея: при вставке и удалении поддерживать инвариант, ограничивающий высоту

- **AVL-дерево** — высоты поддеревьев отличаются не более чем на 1
- **Красно-чёрное дерево** — правила на цвета вершин гарантируют $h \leq 2\log_2(n+1)$

Во всех случаях основные операции работают за $O(\log n)$

# Итоги

- BST — дерево, в котором ключи слева не больше, а справа больше
- Поиск, вставка, удаление работают за $O(h)$
- Inorder-обход даёт элементы в отсортированном порядке
- В худшем случае $h = O(n)$ — «бамбук»
- Сбалансированные деревья гарантируют $h = O(\log n)$