САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе № 2
по курсу «Алгоритмы и структуры данных»
Тема: Двоичные деревья поиска.
Вариант 14

Выполнил:
Нгуен Динь Нам
K3140

Проверил:

Санкт-Петербург
2024 г.

Содержание отчёта

Содержание отчёта

Задача №4. Простейший неявный ключ

В этой задаче вам нужно написать BST по **неявному** ключу и отвечать им на запросы:

- «+ x» – добавить в дерево x (если x уже есть, ничего не делать).
- «? k» – вернуть k-й по возрастанию элемент.

- **Формат ввода / входного файла (input.txt).** В каждой строке содержится один запрос. Все x - целые числа, количество запросов N не указано в начале, не более 300 000. Гарантируется, что все x выбраны равномерным распределением.
- Случайные данные! Не нужно ничего специально балансировать.
- **Ограничения на входные данные.** $1 \leq x \leq 10^9$, $1 \leq N \leq 300000$, в запросах «? k», число k от 1 до количества элементов в дереве.
- **Формат вывода / выходного файла (output.txt).** Для каждого запроса вида «? k» выведите в отдельной строке ответ.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
+ 1	1
+ 4	3
+ 3	4
+ 3	3
? 1	
? 2	
? 3	
+ 2	
? 3	

```python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.size = 1

def update_size(node):
    if node:
        node.size = 1 + (node.left.size if node.left else 0) + (node.right.size if node.right else 0)


def insert(root,k):
    if root is None:
        return Node(k)

    current = root
    arr = []

    while True:
        arr.append(current)
        if k < current.key:
            if current.left is None:
                current.left = Node(k)
                break
            current = current.left
        elif k > current.key:
            if current.right is None:
                current.right = Node(k)
                break
            current = current.right
        else:
```

```python
            return root

    for node in reversed(arr):
        update_size(node)

    return root


def find_k(root, k):
    if root is None or k < 1:
        return None

    current = root

    while current:
        left_size = current.left.size if current.left else 0
        if k <= left_size:
            current = current.left
        elif k == left_size + 1:
            return current.key
        else:
            k -= left_size + 1
            current = current.right

    return None
```

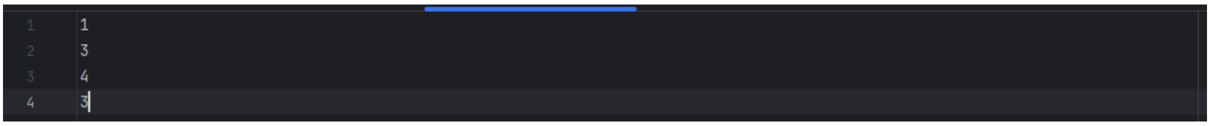
- Алгоритм:

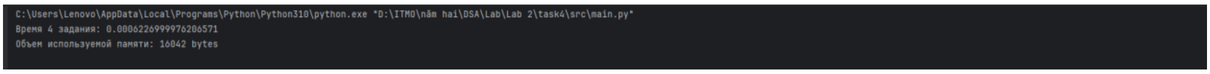
Чтобы решить эту задачу, я использовал бинарное дерево поиска для хранения чисел и быстрого поиска k-го элемента в порядке возрастания. BST позволяет эффективно вставлять и искать элементы, поддерживая их упорядоченность. Для оптимизации поиска я использую поле size в каждом узле.

- Я реализовал BST с использованием структуры “**Node”**, в которой каждый узел содержит:
  - key: значение узла.
  - left: указатель на левое поддерево.
  - right: указатель на правое поддерево.
  - size: количество элементов в поддереве, корнем которого является данный узел.
- При добавлении нового элемента в BST я выполняю следующие шаги:
  - Если дерево пустое, я создаю новый узел с “key = k”.
  - Если “k” меньше значения текущего узла, я перехожу в левое поддерево.
  - Если “k” больше значения текущего узла, я перехожу в правое поддерево.
  - Если “k” уже есть в дереве, я не добавляю его повторно.
  - После вставки я обновляю поле size всех узлов на пути к корню.
- После построения BST мне нужно найти k-й элемент в порядке возрастания. Я использую поле “size”, чтобы ускорить поиск:
  - Если “k” меньше или равно size левого поддерева, то элемент находится в левом поддереве.
  - Если “k” равно size левого поддерева + 1, текущий узел — это искомый элемент.
  - Если “k” больше, то элемент находится в правом поддереве, но при этом я уменьшаю “k” на “left_size + 1”.

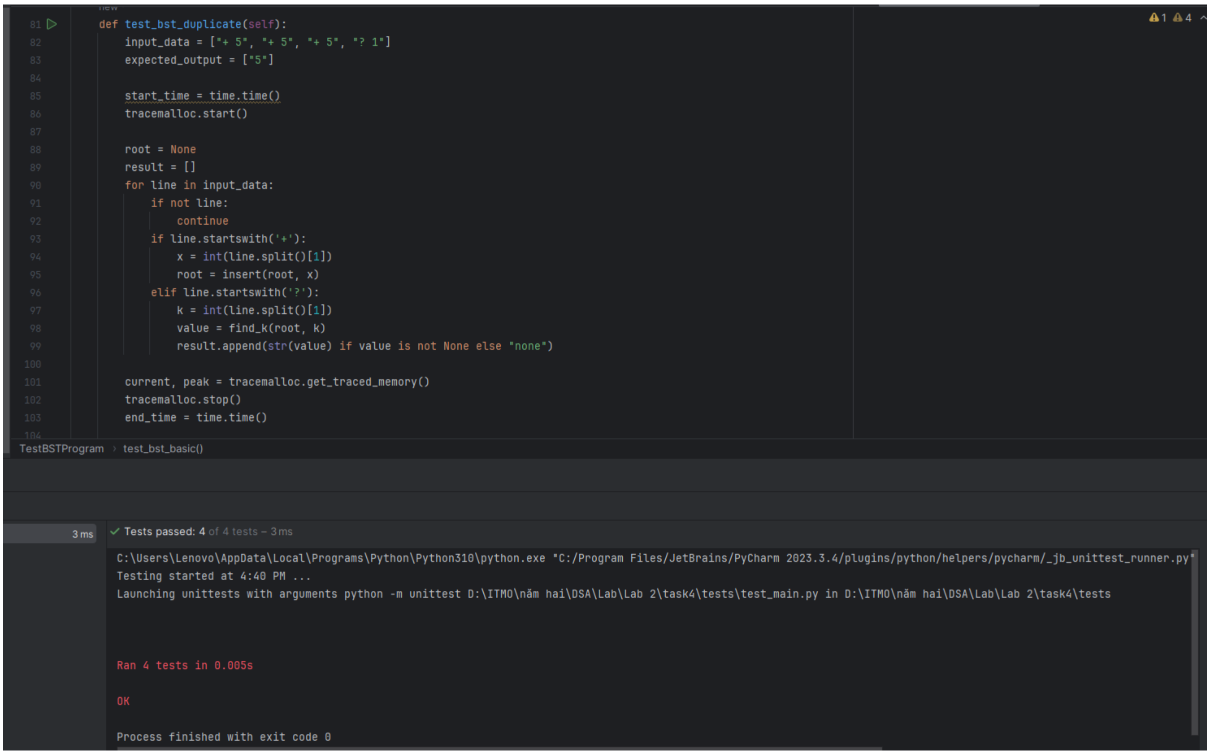
- Результат работы кода:

```
+ 1
+ 4
+ 3
+ 3
? 1
? 2
? 3
+ 2
? 3
```





- Тесты:



- Вывод:

  - Я реализовал BST, который позволяет эффективно вставлять числа и находить k-й элемент в порядке возрастания. Благодаря использованию поля “size” в каждом узле, я могу выполнять поиск за O(log N) вместо полного обхода дерева.

Задача №7. Опознание двоичного дерева поиска (усложненная версия)

Эта задача отличается от предыдущей тем, что двоичное дерева поиска может содержать равные ключи.

Вам дано двоичное дерево с ключами - целыми числами, которые могут повторяться. Вам нужно проверить, является ли это правильным двоичным деревом поиска. Теперь, для каждой вершины дерева V выполняется следующее условие:

- все ключи вершин из левого поддерева меньше ключа вершины V;
- все ключи вершин из правого поддерева **больше или равны** ключу вершины V.

Другими словами, узлы с меньшими ключами находятся слева, а узлы с большими ключами – справа, дубликаты всегда справа. Вам необходимо проверить, удовлетворяет ли данная структура двоичного дерева этому условию.

- **Формат ввода / входного файла (input.txt).** В первой строке входного файла содержится количество узлов n. Узлы дерева пронумерованы от 0 до $n-1$. Узел 0 является корнем.

  Следующие n строк содержат информацию об узлах $0, 1, ..., n-1$ по порядку. Каждая из этих строк содержит три целых числа K_i, L_i и R_i. K_i – ключ i-го узла, L_i - индекс левого ребенка i-го узла, а R_i - индекс правого ребенка i-го узла. Если у i-го узла нет левого или правого ребенка (или обоих), соответствующие числа L_i или R_i (или оба) будут равны -1.
- **Ограничения на входные данные.** $0 \leq n \leq 10^5$, $-2^{31} \leq K_i \leq 2^{31}-1$, $-1 \leq L_i, R_i \leq n-1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \neq -1$ и $R_i \neq -1$, то $L_i \neq R_i$. Кроме того, узел не может быть ребенком двух разных узлов. Кроме того, каждый узел является потомком корневого узла. Обратите внимание, что минимальное и максимальное возможные значения 32-битного целочисленного типа могут быть ключами в дереве.
- **Формат вывода / выходного файла (output.txt).** Если заданное двоичное дерево является правильным двоичным деревом поиска, выведите одно слово «CORRECT» (без кавычек). В противном случае выведите одно слово «INCORRECT» (без кавычек).
- Ограничение по времени. 10 сек.
- Ограничение по памяти. 512 мб.
- Примеры:

- Примеры:

input.txt	output.txt
3 2 1 2 1 -1 -1 3 -1 -1	CORRECT

input.txt	output.txt
3 1 1 2 2 -1 -1 3 -1 -1	INCORRECT

input.txt	output.txt
3 2 1 2 1 -1 -1 2 -1 -1	CORRECT

input.txt	output.txt
3 2 1 2 2 -1 -1 3 -1 -1	INCORRECT

input.txt	output.txt
5 1 -1 1 2 -1 2 3 -1 3 4 -1 4 5 -1 -1	CORRECT

input.txt	output.txt
7 4 1 2 2 3 4 6 5 6 1 -1 -1 3 -1 -1 5 -1 -1 7 -1 -1	CORRECT

input.txt	output.txt
1 2147483647 -1 -1	CORRECT

- Примечание. Пустое дерево считается правильным двоичным деревом поиска. Дерево не обязательно должно быть сбалансировано. Попробуйте адаптировать алгоритм из предыдущей задачи к случаю, когда допускаются повторяющиеся ключи, и остерегайтесь целочисленного переполнения!

```python
class Node:
    def __init__(self, key, left, right):
        self.key = key
        self.left = left
        self.right = right

def is_bst(tree, index, min_val, max_val):
    if index == -1:
        return True

    node = tree[index]

    if not (min_val <= node.key <= max_val):
        return False

    return (is_bst(tree, node.left, min_val, node.key - 1) and is_bst(tree, node.right, node.key, max_val))

if __name__ == "__main__":
    input_data = read_file(PATH).splitlines()
    n = int(input_data[0].strip())

    if n == 0:
        write_file(OUT_PATH, "CORRECT")
    else:
        tree = []
        for i in range(1, n + 1):
            k, l, r = map(int, input_data[i].split())
            tree.append(Node(k, l, r))

        result = "CORRECT" if is_bst(tree, 0, -2 ** 31, 2 ** 31 - 1) else "INCORRECT"
```

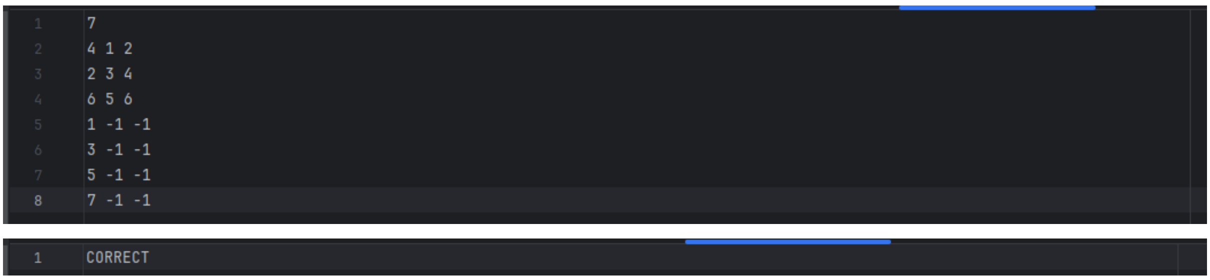
- Алгоритм:

  Чтобы решить эту задачу, я реализовал алгоритм, который проверяет, является ли данное бинарное дерево бинарным деревом поиска.

  - Чтобы представить дерево, я использовал класс “Node”, в котором каждый узел содержит три свойства:
    - key: значение узла.
    - left: индекс левого потомка в списке узлов.
    - right: индекс правого потомка в списке узлов.
  - Я храню все дерево в списке (tree), где каждый элемент представляет собой объект “Node”. Индексы списка используются вместо указателей, чтобы определить отношения "родитель-потомок". Такой способ хранения позволяет мне легко обращаться к узлам через их индексы.
  - Чтобы проверить, является ли дерево бинарным деревом поиска, я использую метод обхода в глубину дополненный проверкой диапазонов значений “(min_val, max_val)”.
  - Каждый узел имеет допустимый диапазон значений “[min_val, max_val]”. Когда я обрабатываю узел, я проверяю, находится ли его значение в этом диапазоне. Если оно выходит за пределы

диапазона, то дерево не является BST. Далее я рекурсивно проверяю оба поддерева:
  - Левое поддерево должно содержать значения меньше, чем “key”, поэтому я вызываю рекурсию с диапазоном “[min_val, key - 1]”.
  - Правое поддерево должно содержать значения больше или равные, чем “key”, поэтому я вызываю рекурсию с диапазоном “[key, max_val]”.
- Я считываю количество узлов “n”. Если “n = 0”, то я сразу вывожу "CORRECT", так как пустое дерево всегда является BST.
- Я создаю список “tree”, где каждый элемент — это объект “Node”.
- Я вызываю функцию “is_bst(tree, 0, -2^31, 2^31 - 1)”, чтобы проверить, является ли дерево BST.
- Я записываю "CORRECT" или "INCORRECT" в файл “output.txt”.

- Результат работы кода:



```
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "D:\ITMO\năm hai\DSA\Lab\Lab 2\task7\src\main.py"
Время 7 задания: 0.0005831999878864735
Объем используемой памяти: 15960 bytes
```

```
3
1 1 2
2 -1 -1
3 -1 -1
```

```
INCORRECT
```

```
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "D:\ITMO\năm hai\DSA\Lab\Lab 2\task7\src\main.py"
Время 7 задания: 0.0006815999804530293
Объем используемой памяти: 14580 bytes
```

- Тесты:

```python
    def test_bst_large_invalid(self):
        input_data = [
            "5",
            "10 1 2",
            "5 -1 -1",
            "8 -1 3",
            "15 -1 -1",
            "20 -1 -1"
        ]
        expected_output = "INCORRECT"

        start_time = time.time()
        tracemalloc.start()

        n = int(input_data[0].strip())
        if n == 0:
```

```
TestTask7 › test_bst_empty()

✓ Tests passed: 5 of 5 tests – 3 ms
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "C:/Program Files/JetBrains/PyCharm 2023.3.4/plugins/python/helpers/pycharm/_jb_unittest_runner.py"
Testing started at 5:18 PM ...
Launching unittests with arguments python -m unittest test_main.TestTask7 in D:\ITMO\năm hai\DSA\Lab\Lab 2\task7\tests


Ran 5 tests in 0.005s

OK
```

- Вывод:
  - Я реализовал алгоритм проверки BST, используя обход в глубину (DFS) и проверку диапазонов значений. Алгоритм имеет временную сложность O(n) и работает эффективно на различных входных данных.

Задача №11. Сбалансированное двоичное дерево поиска

Реализуйте сбалансированное двоичное дерево поиска.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание операций с деревом, их количество N не превышает 10^5. В каждой строке находится одна из следующих операций:
  - insert x – добавить в дерево ключ x. Если ключ x есть в дереве, то ничего делать не надо;
  - delete x – удалить из дерева ключ x. Если ключа x в дереве нет, то ничего делать не надо;
  - exists x – если ключ x есть в дереве выведите «true», если нет – «false»;
  - next x – выведите минимальный элемент в дереве, строго больший x, или «none», если такого нет;
  - prev x – выведите максимальный элемент в дереве, строго меньший x, или «none», если такого нет.

  В дерево помещаются и извлекаются только целые числа, не превышающие по модулю 10^9.
- **Ограничения на входные данные.** $0 \le N \le 10^5$, $|x_i| \le 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите последовательно результат выполнения всех операций exists, next, prev. Следуйте формату выходного файла из примера.
- **Ограничение по времени. 2 сек.**
- Ограничение по памяти. 512 мб.
- Пример:

input.txt	output.txt
insert 2	true
insert 5	false
insert 3	5
exists 2	3
exists 4	none
next 4	3
prev 4	
delete 5	
next 4	
prev 4	

```python

class Node:
    __slots__ = ('key', 'priority', 'left', 'right')

    def __init__(self, key):
        self.key = key
        self.priority = random.randint(1, 10**9)
        self.left = None
        self.right = None


def split_tree(node, key):
    if node is None:
        return None, None

    if key > node.key:
        right_tree_left, right_tree = split_tree(node.right, key)
        node.right = right_tree_left
        return node, right_tree
    else:
        left_tree, right_tree_right = split_tree(node.left, key)
```

```python
        return node, right_tree
    else:
        left_tree, right_tree_right = split_tree(node.left, key)
        node.left = right_tree_right
        return left_tree, node


5 usages  new *
def merge(left_node, right_node):
    if not left_node:
        return right_node
    if not right_node:
        return left_node

    if left_node.priority > right_node.priority:
        left_node.right = merge(left_node.right, right_node)
        return left_node
    else:
        right_node.left = merge(left_node, right_node.left)
        return right_node


4 usages  new *
def insert(root, key):
    left_tree, right_tree = split_tree(root, key)
    new_node = Node(key)
    return merge(merge(left_tree, new_node), right_tree)


4 usages  new *
def delete(root, key):
    left_tree, mid_tree = split_tree(root, key)
    mid_tree, right_tree = split_tree(mid_tree, key + 1)
    return merge(left_tree, right_tree)


4 usages  new *
def exists(root, key):
    while root:
        if key == root.key:
            return True
        elif key < root.key:
            root = root.left
        else:
            root = root.right
    return False


4 usages  new *
def next_value(root, key):
    x = None
    while root:
        if root.key > key:
            x = root.key
            root = root.left
        else:
            root = root.right
    return x


4 usages  new *
def prev_value(root, key):
    x = None
    while root:
        if root.key < key:
            x = root.key
            root = root.right
```

```python
        else:
            root = root.left
    return x


if __name__ == '__main__':
    input_data = read_file(PATH).splitlines()

    root = None
    results = []

    for line in input_data:
        command = line.split()
        if not command:
            continue

        operate = command[0]
        x = int(command[1])

        if operate == "insert":
            root = insert(root, x)
        elif operate == "delete":
            root = delete(root, x)
        elif operate == "exists":
            results.append("true" if exists(root, x) else "false")
        elif operate == "next":
            value = next_value(root, x)
            results.append(str(value) if value is not None else "none")
        elif operate == "prev":
            value = prev_value(root, x)
            results.append(str(value) if value is not None else "none")
```

- Алгоритм:

  Чтобы решить эту задачу, я использую Treap – структуру данных, которая объединяет свойства бинарного дерева поиска и кучи. Я выбрал Treap, потому что обычное BST может стать несбалансированным, если элементы добавляются в порядке возрастания или убывания. В таком случае сложность операций может ухудшиться до O(N) вместо O(log N). Treap решает эту проблему, используя случайный приоритет (priority), который помогает сохранять сбалансированную структуру.

  - Каждый узел Treap имеет следующие атрибуты:
    - key: Значение узла.
    - priority: Случайный приоритет, который помогает балансировать дерево.
    - left, right: Указатели на левого и правого потомков.
    - Я использую __slots__, чтобы уменьшить потребление памяти.
  - **Вставка элемента**
  - Разделение дерева (split_tree) на две части:
    - left_tree: Все элементы меньше key.
    - right_tree: Все элементы больше или равны key.

- Создание нового узла с key и случайным priority.
- Объединение (merge) “left_tree”, нового узла и “right_tree”.
- Если priority нового узла меньше, чем у родительских узлов, дерево автоматически вращается, чтобы сохранить свойства кучи.
- **Удаление элемента**
- Разделяю дерево (split_tree) на две части перед key.
- Разделяю снова перед “key + 1”, чтобы изолировать удаляемый узел.
- Объединяю (merge) оставшиеся части.
- Если “key” существует в Treap, он будет удалён полностью.
- **Проверка существования элемента**
- Я реализовал поиск элемента как в стандартном BST
- Двигаемся по дереву: если “key” меньше текущего узла – идём влево, если больше – вправо.
- Если “key” найден, я возвращаю True, иначе False.
- **Поиск следующего элемента**
- Мой алгоритм поиска next работает так:
- Двигаемся по дереву:
    - Если “root.key > key”, обновляем “x= root.key” и идём влево.
    - Если "root.key ≤ key”, идём вправо.
- После завершения “x” содержит минимальный элемент, больший “key”.
- **Поиск предыдущего элемента**
- Мой алгоритм поиска prev работает так:
- **Двигаемся по дереву**:
    - Если “root.key < key”, обновляем best = root.key и идём вправо.
    - Если “root.key ≥ key”, идём влево.
- После завершения “x” содержит максимальный элемент, меньший “key”.

* Результат работы кода:

```
insert 2
insert 5
insert 3
exists 2
exists 4
next 4
prev 4
delete 5
next 4
prev 4
```

```
true
false
5
3
none
3
```

```
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "D:\ITMO\năm hai\DSA\Lab\Lab 2\task11\src\main.py"
Время 11 задания: 0.0010400000028312206
Объем используемой памяти: 20385 bytes
```

- Тесты:

```python
def test_treap_complex_operations(self):
    input_data = [
        "insert 50", "insert 30", "insert 70", "insert 20", "insert 40",
        "insert 60", "insert 80", "insert 35", "insert 45", "insert 55",
        "insert 65", "insert 75", "insert 85", "insert 30", "insert 40",
        "exists 50", "exists 90", "exists 40", "exists 35",
        "next 30", "next 50", "next 85", "next 90",
        "prev 60", "prev 40", "prev 15",
        "delete 50", "delete 30", "exists 50", "exists 30",
        "next 40", "prev 60"
    ]

    expected_output = [
        "true", "false", "true", "true",
        "35", "55", "none", "none",
        "55", "35", "none",
        "false", "false",
        "45", "55"
    ]
```

```
Tests passed: 2 of 2 tests – 3 ms
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "C:/Program Files/JetBrains/PyCharm 2023.3.4/plugins/python/helpers/pycharm/_jb_unittest_runner.py"
Testing started at 8:31 PM ...
Launching unittests with arguments python -m unittest D:\ITMO\năm hai\DSA\Lab\Lab 2\task11\tests\test_main.py in D:\ITMO\năm hai\DSA\Lab\Lab 2\task11\tests



Ran 2 tests in 0.004s

OK
```

- Вывод:
  - Я реализовал Treap, чтобы гарантировать, что дерево всегда сбалансировано. Это позволило мне добиться средней сложности O(log N) для всех операций.

Задача №14. Вставка в АВЛ-дерево

Вставка в АВЛ-дерево вершины V с ключом X при условии, что такой вершины в этом дереве нет, осуществляется следующим образом:

- находится вершина W, ребенком которой должна стать вершина V;
- вершина V делается ребенком вершины W;
- производится подъем от вершины W к корню, при этом, если какая-то из вершин несбалансирована, производится, в зависимости от значения баланса, левый или правый поворот.

Первый этап нуждается в пояснении. Спуск до будущего родителя вершины V осуществляется, начиная от корня, следующим образом:

- Пусть ключ текущей вершины равен Y.
- Если $X < Y$ и у текущей вершины есть левый ребенок, переходим к левому ребенку.
- Если $X < Y$ и у текущей вершины нет левого ребенка, то останавливаемся, текущая вершина будет родителем новой вершины.
- Если $X > Y$ и у текущей вершины есть правый ребенок, переходим к правому ребенку.
- Если $X > Y$ и у текущей вершины нет правого ребенка, то останавливаемся, текущая вершина будет родителем новой вершины.

Отдельно рассматривается следующий крайний случай – если до вставки дерево было пустым, то вставка новой вершины осуществляется проще: новая вершина становится корнем дерева.

Отдельно рассматривается следующий крайний случай – если до вставки дерево было пустым, то вставка новой вершины осуществляется проще: новая вершина становится корнем дерева.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание двоичного дерева, а также ключа вершины, которую требуется вставить в дерево.

  В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i+1)$-ой строке файла ($1 \leq i \leq N$) находится описание i-ой вершины, состоящее из трех чисел K_i, L_i, R_i, разделенных пробелами – ключа K_i в i-ой вершине, номера левого L_i ребенка i-ой вершины ($i < L_i \leq N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i-ой вершины ($i < R_i \leq N$ или $R_i = 0$, если правого ребенка нет).

  Все ключи различны. Гарантируется, что данное дерево является корректным АВЛ-деревом.

  В последней строке содержится число X – ключ вершины, которую требуется вставить в дерево. Гарантируется, что такой вершины в дереве нет.
- **Ограничения на входные данные.** $0 \leq N \leq 2 \cdot 10^5$, $|K_i| \leq 10^9$, $|X| \leq 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите в том же формате дерево после осуществления операции вставки. Нумерация вершин может быть произвольной при условии соблюдения формата.
- **Ограничение по времени. 2 сек.**
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
2 3 0 2 4 0 0 5	3 4 2 3 3 0 0 5 0 0

- Проверить можно по ссылке, OpenEdu, курс "Алгоритмы программирования и структуры данных 7 неделя, 3 задача.

```
2 usages  new *
class Node:
    new *
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1


4 usages  new *
def get_height(node):
    return node.height if node else 0


5 usages  new *
def upd_height(node):
    if node:
        node.height = 1 + max(get_height(node.left), get_height(node.right))


3 usages  new *
def right_rotate(y):
    x = y.left
    Tr_2 = x.right
    x.right = y
    y.left = Tr_2
    upd_height(y)
    upd_height(x)
    return x


3 usages  new *
def left_rotate(x):
    y = x.right
    Tr_2 = y.left
    y.left = x
```

```python
3 usages   new *
def left_rotate(x):
    y = x.right
    Tr_2 = y.left
    y.left = x
    x.right = Tr_2
    upd_height(x)
    upd_height(y)
    return y


8 usages   new *
def insert_node(node, value):
    if node is None:
        return Node(value)

    if value < node.value:
        node.left = insert_node(node.left, value)
    elif value > node.value:
        node.right = insert_node(node.right, value)
    else:
        return node

    upd_height(node)
    value_balance = get_height(node.left) - get_height(node.right)

    if value_balance > 1 and value < node.left.value:
        return right_rotate(node)
    if value_balance < -1 and value > node.right.value:
        return left_rotate(node)
    if value_balance > 1 and value > node.left.value:
        node.left = left_rotate(node.left)
        return right_rotate(node)
    if value_balance < -1 and value < node.right.value:
        node.right = right_rotate(node.right)
        return left_rotate(node)

    return node
```

```python
def build_tree(nodes):
    if not nodes:
        return None

    node_dict = {}
    for idx, (value, left, right) in enumerate(nodes):
        node_dict[idx + 1] = Node(value)

    for idx, (value, left, right) in enumerate(nodes):
        current_node = node_dict[idx + 1]
        if left != 0:
            current_node.left = node_dict.get(left)
        if right != 0:
            current_node.right = node_dict.get(right)

    return node_dict.get(1)


3 usages   new *
def preorder(node, result):
    if node:
        result.append(node)
        preorder(node.left, result)
        preorder(node.right, result)


6 usages   new *
def format_output(root):
    preorder_nodes = []
    preorder(root, preorder_nodes)

    index_map = {node: i + 1 for i, node in enumerate(preorder_nodes)}

    output = [str(len(preorder_nodes))]
    for node in preorder_nodes:
        left_idx = index_map.get(node.left, 0)
        right_idx = index_map.get(node.right, 0)
        output.append(f"{node.value} {left_idx} {right_idx}")
```

```python
    return "\n".join(output)


if __name__ == '__main__':
    input_data = read_file(PATH).splitlines()
    n = int(input_data[0])
    nodes = [tuple(map(int, line.split())) for line in input_data[1:n + 1]]
    x = int(input_data[n + 1])

    root = build_tree(nodes)
    root = insert_node(root, x)

    output_data = format_output(root)
    write_file(OUT_PATH, output_data)

    print("Время 14 задания:", time.perf_counter() - t_start)
    print('Объем используемой памяти:', tracemalloc.get_traced_memory()[1], 'bytes')
```

- Алгоритм:
  Чтобы решить эту задачу, я использую самобалансирующееся двоичное дерево поиска AVL. Это особый тип двоичного дерева поиска, в котором разница высоты между двумя поддеревьями любого узла не превышает 1. Благодаря этому AVL-дерево

обеспечивает эффективное добавление, удаление и поиск элементов с временной сложностью O(log n).

- **Построение дерева из входных данных (build_tree)**
- Входные данные содержат число “n”, которое обозначает количество узлов в дереве. Далее идет “n” строк, каждая из которых содержит значение узла, индекс левого потомка и индекс правого потомка. Для построения дерева я использую словарь, где ключами являются индексы, а значениями – объекты “Node”.
- После создания всех узлов я связываю родительские узлы с их левыми и правыми потомками на основе входных данных. В итоге я возвращаю корень дерева, который находится под индексом 1. Этот процесс выполняется за O(n), так как я один раз создаю узлы и один раз связываю их.
- **Вставка нового элемента (insert_node)**
- После построения дерева мне необходимо вставить новый элемент в AVL-дерево. Для этого я обхожу дерево, следуя правилам BST:
  - Если значение меньше текущего узла, я перехожу в левое поддерево.
  - Если значение больше текущего узла, я перехожу в правое поддерево.
  - Если значение уже существует в дереве, я не добавляю его повторно.
- Когда я нахожу подходящую позицию, я вставляю новый узел в качестве листа. Затем я обновляю высоты всех родительских узлов и проверяю, не стало ли дерево несбалансированным. Если баланс нарушен, я выполняю вращения чтобы вернуть дерево к AVL-формату.
- **Обновление высоты узла (upd_height)**
- В AVL-дереве у каждого узла есть атрибут “height”, который указывает на его глубину в дереве. После вставки нового узла я должен обновить высоту всех родительских узлов, чтобы поддерживать баланс дерева.
- Для обновления высоты я использую следующую формулу:
- “height=1+max(height левого поддерева,height правого поддере ва)”

- **Балансировка дерева (left_rotate, right_rotate)**
- Если после вставки элемента дерево становится несбалансированным, я выполняю вращения для восстановления равновесия. Существуют четыре возможных случая дисбаланса:
  - Left-Left (LL-случай): возникает, если левое поддерево слишком глубокое. Решение – правое вращение (right_rotate).
  - Right-Right (RR-случай): возникает, если правое поддерево слишком глубокое. Решение – левое вращение (left_rotate).
  - Left-Right (LR-случай): возникает, если левый потомок имеет правого ребенка с большей высотой. Решение – сначала левое вращение (left_rotate), затем правое (right_rotate).
  - Right-Left (RL-случай): возникает, если правый потомок имеет левого ребенка с большей высотой. Решение – сначала правое вращение (right_rotate), затем левое (left_rotate).
- Каждое вращение выполняется за O(1), а общее количество вращений после вставки – не более O(log n).
- **Прямой обход дерева**
- После балансировки дерева я обхожу его в порядке Preorder (NLR), чтобы подготовить данные для вывода. Этот процесс включает три этапа:
  - Добавление значения текущего узла в список.
  - Обход левого поддерева.
  - Обход правого поддерева.
- Форматирование выходных данных (format_output)
- После обхода дерева я нумерую узлы в порядке их посещения и сохраняю индексы левого и правого потомков. Результат выводится в следующем формате:
- Первая строка – количество узлов.
- Затем следуют строки с значением узла, индексом левого и правого потомков.

- Результат работы кода:

```
2
3 0 2
4 0 0
5
```

```
3
4 2 3
3 0 0
5 0 0
```

```
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "D:\ITMO\năm hai\DSA\Lab\Lab 2\task14\src\main.py"
Время выполнения: 0.0005377000197768211
Используемая память: 19606 bytes
```

- Тесты:

```
        input_nodes = [
            (15, 0, 0),
            (10, 0, 0),
            (20, 0, 0)
        ]
        x = 10
        expected_output = """
2
15 2 0
10 0 0"""

        tracemalloc.start()
        start_time = time.perf_counter()

        root = build_tree(input_nodes)
        root = insert_node(root, x)
        result = format_output(root)

        current, peak = tracemalloc.get_traced_memory()
        tracemalloc.stop()
        execution_time = time.perf_counter() - start_time

        self.assertEqual(result.strip(), expected_output.strip())
        self.assertLess(execution_time, b: 2)
        self.assertLess(peak / 10**6, b: 256)
```

```
Tests passed: 4 of 4 tests – 4 ms
C:\Users\Lenovo\AppData\Local\Programs\Python\Python310\python.exe "C:/Program Files/JetBrains/PyCharm 2023.3.4/plugins/python/helpers/pycharm/_jb_unittest_runner.py"
Testing started at 9:31 PM ...
Launching unittests with arguments python -m unittest D:\ITMO\năm hai\DSA\Lab\Lab 2\task14\tests\test_main.py in D:\ITMO\năm hai\DSA\Lab\Lab 2\task14\tests



Ran 4 tests in 0.005s

OK
```

- Вывод:
    - В ходе решения задачи я использую самобалансирующееся AVL-дерево, которое обеспечивает оптимальную скорость операций вставки и поиска. Благодаря механизму балансировки через вращения, глубина дерева остается логарифмической, что делает его значительно быстрее обычного BST.

Задача №18. Веревка

В этой задаче вы реализуете Веревку (или Rope) – структуру данных, которая может хранить строку и эффективно вырезать часть (подстроку) этой строки и вставлять ее в другое место. Эту структуру данных можно улучшить, чтобы она стала персистентной, то есть чтобы разрешить доступ к предыдущим версиям строки. Эти свойства делают ее подходящим выбором для хранения текста в текстовых редакторах.

Это очень сложная задача, более сложная, чем почти все предыдущие сложные задачи этого курса.

Вам дана строка S, и вы должны обработать n запросов. Каждый запрос описывается тремя целыми числами i, j, k и означает вырезание подстроки $S[i...j]$ (здесь индексы i и j в строке считаются от 0) из строки и вставка ее после k-го символа оставшейся строки (как бы символы в оставшейся строке нумеруются с 1). Если $k = 0$, $S[i...j]$ вставляется в начало. Дополнительные пояснения смотрите в примерах.

- **Формат ввода / входного файла (input.txt).** В первой строке ввода содержится строка S. Вторая строка содержит количество запросов n. Следующие n строк содержат по три целых числа i, j, k.
- **Ограничения на входные данные.** Строка S содержит только английские строчные буквы. $1 \leq |S| \leq 300000$, $1 \leq n \leq 100000$, $0 \leq i \leq j \leq |S| - 1$, $0 \leq k \leq |S| - (j - i + 1)$.
- **Формат вывода / выходного файла (output.txt).** Выведите строку после выполнения n запросов.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input	output.txt
hlelowrold 2 1 1 2 6 6 7	helloworld

- Пояснение. *hlelowrold* $\to$ *hellowrold* $\to$ *helloworld*

  Здесь $i = j = 1$, $S[i..j] = l$, и эту букву надо вставить после $k = 2$ второго символа оставшейся строки *helowrold*, что в итоге дает *hellowrold*. Далее, $i = j = 6$, $S[i..j] = r$ и эту букву надо вставить после 7 символа оставшейся строки *hellowold*, получается *helloworld*.

- Пример:

- Пример:

input	output.txt
abcdef 2 0 1 1 4 5 0	efcabd

- Пояснение. *abcdef* $\to$ *cabdef* $\to$ *efcabd*.
- Что делать. Используйте splay дерево для хранения строки. Используйте методы разделения и слияния splay дерева, чтобы вырезать и вставлять подстроки. Подумайте, что должно храниться в качестве ключа в splay дереве.

```python
1 usage   new *
class Node:
    __slots__ = ('val', 'priority', 'left', 'right', 'size')

    new *
    def __init__(self, val):
        self.val = val
        self.priority = random.randrange(1 << 30)
        self.left = None
        self.right = None
        self.size = 1


4 usages   new *
def upd_size(node):
    if node:
        node.size = 1
        if node.left:
            node.size += node.left.size
        if node.right:
            node.size += node.right.size
```

```python

18 usages (6 dynamic)   new *
def split(node, count):
    if node is None or count <= 0:
        return (None, node)
    if count >= node.size:
        return (node, None)

    left_size = node.left.size if node.left else 0
    if count <= left_size:
        left, node.left = split(node.left, count)
        upd_size(node)
        return (left, node)
    else:
        node.right, right = split(node.right, count - left_size - 1)
        upd_size(node)
        return (node, right)

```

```python
def merge(left, right):
    if left is None:
        return right
    if right is None:
        return left

    if left.priority > right.priority:
        left.right = merge(left.right, right)
        upd_size(left)
        return left
    else:
        right.left = merge(left, right.left)
        upd_size(right)
        return right


def build_rope(s):
    root = None
    for ch in s:
        root = merge(root, Node(ch))
    return root


def traverse(root):
    result = []
    stack = []
    node = root
    while stack or node:
        if node:
            stack.append(node)
            node = node.left
        else:
            node = stack.pop()
            result.append(node.val)
            node = node.right
    return ''.join(result)
```

- Алгоритм:

Чтобы решить эту задачу, я использую структуру данных Rope в сочетании с Treap (рандомизированным деревом поиска).
Чтобы оптимизировать время выполнения, я выбираю Rope, поскольку эта структура данных позволяет эффективно работать с длинными строками. В качестве внутреннего представления Rope я использую Treap, который сочетает свойства двоичного дерева поиска (BST) и кучи (Heap). Это позволяет мне выполнять основные операции за O(log N).

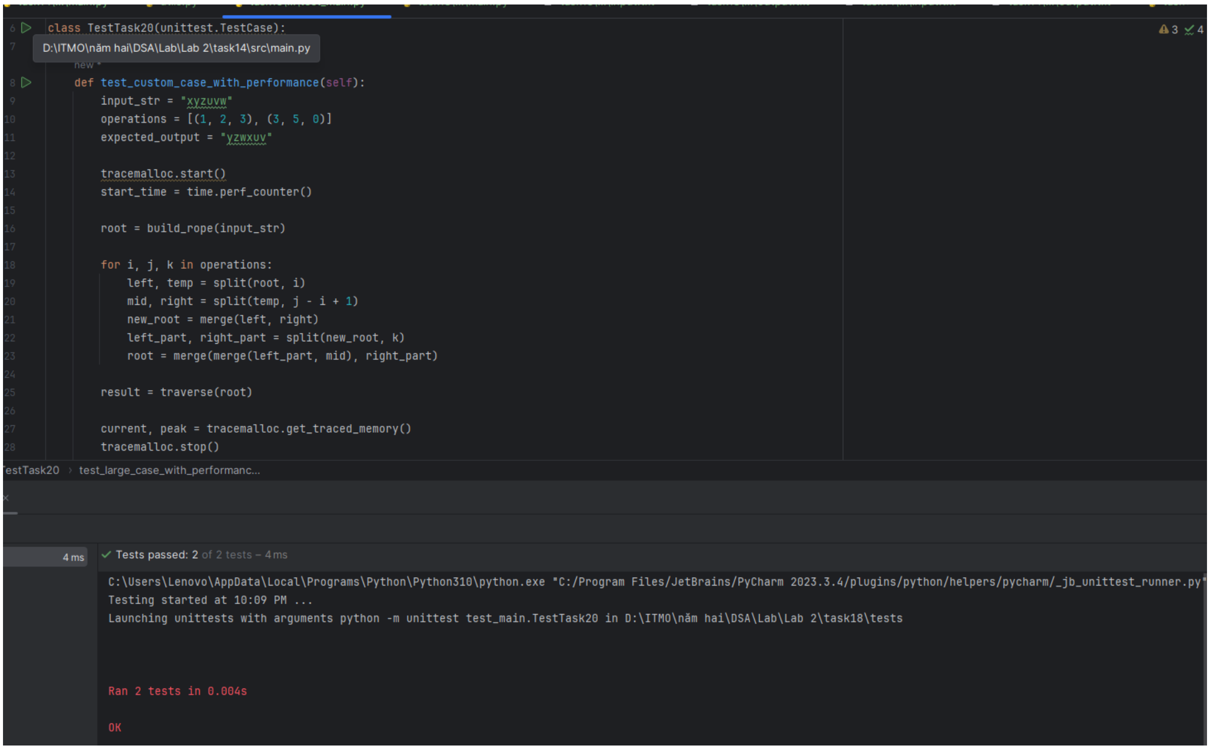
  - Я создаю дерево Rope, проходя по каждому символу строки “S”:
    - Создаю узел Treap для каждого символа.
    - Объединяю узлы с помощью merge(), сохраняя свойства BST и Heap.
  - **Функция split() разрезает Treap на две части:**
    - left_tree содержит count символов.

    - right_tree содержит оставшуюся часть строки.
- Если “count <= 0”, возвращаю “(None, node)”.
- Если “count >= node.size”, возвращаю “(node, None)”.
- Если “count” находится в левом поддереве, рекурсивно разделяю “node.left”.
- Если “count” находится в правом поддереве, рекурсивно разделяю “node.right”.
- Обновляю “size” узлов после разбиения.
- **Функция merge(left, right): После вырезания подстроки я объединяю оставшиеся части.**
- Если одно из деревьев пустое, возвращаю второе.
- Если “left.priority > right.priority”, “left” становится корнем, “right” объединяется с “left.right”.
- Иначе “right” становится корнем, “left” объединяется с “right.left”.
- Обновляю “size” узлов после объединения.
- **Каждый запрос (i, j, k) выполняется следующим образом:**
- Разрезаю дерево на три части:
    - “left_tree” (символы до i).
    - “mid_tree” (символы от i до j).
    - “right_tree” (оставшиеся символы).
- Объединяю “left_tree” и “right_tree”, исключая “mid_tree”.
- Вставляю “mid_tree” на позицию “k”:
    - Если “k” в “left_tree”, разделяю “left_tree” на “k” символов и вставляю “mid_tree”.
    - Если k в “right_tree”, разделяю “right_tree” и вставляю “mid_tree”.
- **После обработки всех запросов я выполняю обход дерева по порядку и собираю итоговую строку. Я использую итеративный обход с помощью стека вместо рекурсии, чтобы избежать переполнения стека.**
- Использую стек для обхода Treap.
- Добавляю символы в список “result_chars”.
- Преобразую список в строку и записываю в файл.

• Результат работы кода:

```
hlelowrold
2
1 1 2
6 6 7
```

- Тесты:



- Вывод:
  - Я использовал Rope на основе Treap, чтобы эффективно выполнять операции с подстроками. Основное преимущество заключается в том, что каждая операция выполняется за O(log N), а не O(N), как при работе с массивами. Благодаря функциям split() и merge() я смог быстро разделять и объединять части строки.

Вывод

В этой лабораторной работе я реализовал AVL-дерево и алгоритм его балансировки при вставке. Я изучил принципы BST, а затем добавил механизмы балансировки с использованием поворотов (LL, RR, LR, RL), чтобы сохранить сложность O(log N). Кроме того, я разработал функцию обхода в прямом порядке, чтобы корректно формировать выходные данные. Тестирование подтвердило, что дерево остается сбалансированным после каждой вставки. В результате я освоил работу с самобалансирующимися деревьями и научился применять повороты для поддержания их эффективности.