

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

Отчет по лабораторной работе №2  
по курсу «Алгоритмы и структуры данных»

Вариант 16

Выполнила:  
Лаузер Я.П. К3144

Проверил:

Санкт-Петербург

2024 г.

**Содержание отчета**

## **Лабораторная №2**

**Задача №1. Обход двоичного дерева**

**Задача №11. Сбалансированное двоичное дерево поиска**

**Задача №16. К-й максимум**

## 2 лаба “Двоичные деревья поиска”.

### Задача №1.Обход двоичного дерева

#### Текст задачи

В этой задаче вы реализуете три основных способа обхода двоичного дерева «в глубину»: центрированный (in-order), прямой (pre-order) и обратный (post-order). Очень полезно попрактиковаться в их реализации, чтобы лучше понять бинарные деревья поиска.

Вам дано корневое двоичное дерево. Выведите центрированный (in-order), прямой (pre-order) и обратный (post-order) обходы в глубину.

- **Формат ввода: стандартный ввод или 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  $0, 1, \dots, n - 1$  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.**  $1 \leq n \leq 10^5, 0 \leq K_i \leq 10^9, -1 \leq L_i, R_i \leq n - 1$ . 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.
- **Формат вывода / выходного файла (output.txt).** Выведите три строки. Первая строка должна содержать ключи узлов при центрированном обходе дерева (in-order). Вторая строка должна содержать ключи узлов при прямом обходе дерева (pre-order). Третья строка должна содержать ключи узлов при обратном обходе дерева (post-order).
- Ограничение по времени. 5 сек.

#### Листинг кода.

```
f = open("input.txt")
n = int(f.readline())
tree = []
tree_version = [[-1] * 3 for _ in range(n)]
for i in range(n):
    s = f.readline().split()
    x = [int(i) for i in s]
    tree.append(x)
print(tree)
for i in range(n):
    tree_version[i][0] = tree[i][0]
    if tree[i][1] != -1:
        tree_version[tree[i][1]][1] = i
    if tree[i][2] != -1:
        tree_version[tree[i][2]][2] = i
```

```
top = -1
b = []
a = []
c = []
for i in range(n):
    if tree_version[i][1] == -1 and tree_version[i][2]
    == -1:
        top = i
print(tree_version, top)

def inorder(v, x1):
    if v[x1][1] != -1:
        inorder(v, v[x1][1])
    a.append(v[x1][0])
    if v[x1][2] != -1:
        inorder(v, v[x1][2])
    return a

def preorder(v, x1):
    b.append(v[x1][0])
    if v[x1][1] != -1:
        preorder(v, v[x1][1])
    if v[x1][2] != -1:
        preorder(v, v[x1][2])
    return b

def postorder(v, x1):
    if v[x1][1] != -1:
        postorder(v, v[x1][1])
    if v[x1][2] != -1:
        postorder(v, v[x1][2])
    c.append(v[x1][0])
```

```

    return c

z = open("output.txt", "w")
a1 = (postorder(tree, top))
a2 = (preorder(tree, top))
a3 = (inorder(tree, top))
for i in a1:
    z.write(str(i) + " ")
z.write("\n")
for i in a2:
    z.write(str(i) + " ")
z.write("\n")
for i in a3:
    z.write(str(i) + " ")

```

Текстовое объяснение решения.

В этом коде реализуется обход дерева. Дерево - это структура данных, в которой элементы расположены иерархически, как ветки на дереве. В коде вводятся данные из файла input.txt. tree - это список списков, представляющий дерево. tree\_version - это список списков, который представляет "дерево" в виде списка (для удобства работы с индексами). top - это корень дерева. Далее код строит три списка:

a - список значений, полученных после обхода дерева в порядке "порядок".

b - список значений, полученных после обхода дерева в порядке "предварительный порядок".

c - список значений, полученных после обхода дерева в порядке "постфиксный порядок". В конце код записывает эти три списка в файл output.txt.

В общем, код работает с данными из файла input.txt, преобразует их в дерево, и записывает результаты обхода дерева в файл output.txt.

Результат работы кода на примерах из текста задачи:

1.py	input.txt	output.txt
1	5	
2	4 1 2	
3	2 3 4	
4	5 -1 -1	
5	1 -1 -1	
6	3 -1 -1	

1.py	input.txt	output.txt
1	1 3 2 5 4	
2	4 2 1 3 5	
3	1 2 3 4 5	

## Задача №11. Сбалансированное двоичное дерево поиска

### Текст задачи

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.**  $0 \leq N \leq 10^5$ ,  $|x_i| \leq 10^9$ .

### Листинг кода

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, node, key):
        if not node:
            return Node(key)

        if key < node.key:
            node.left = self._insert(node.left, key)
        elif key > node.key:
```

```

        node.right = self._insert(node.right, key)

    return node

def delete(self, key):
    self.root = self._delete(self.root, key)

def _delete(self, node, key):
    if not node:
        return node

    if key < node.key:
        node.left = self._delete(node.left, key)
    elif key > node.key:
        node.right = self._delete(node.right, key)
    else:
        if not node.left:
            return node.right
        elif not node.right:
            return node.left
        else:
            node.key = self._min_value(node.right)
            node.right = self._delete(node.right,
node.key)

            return node

def _min_value(self, node):
    current = node
    while current.left:
        current = current.left
    return current.key

def exists(self, key):
    return self._exists(self.root, key)

```



```

def _exists(self, node, key):
    if not node:
        return False

    if node.key == key:
        return True
    elif key < node.key:
        return self._exists(node.left, key)
    else:
        return self._exists(node.right, key)

def next(self, key):
    return self._next(self.root, key)

def _next(self, node, key):
    if not node:
        return None

    if key < node.key:
        left_result = self._next(node.left, key)
        if left_result is None or left_result <=
key:
            return node.key
        else:
            return left_result
    else:
        return self._next(node.right, key)

def prev(self, key):
    return self._prev(self.root, key)

def _prev(self, node, key):
    if not node:
        return None

```

```

        if key > node.key:
            right_result = self._prev(node.right, key)
            if right_result is None or right_result >=
key:
                return node.key
            else:
                return right_result
        else:
            return self._prev(node.left, key)

bst = BinarySearchTree()

with open('input.txt', 'r') as file,
open('output.txt', 'w') as output:
    for line in file:
        operation, key = line.split()
        key = int(key)
        if operation == 'insert':
            bst.insert(key)
        elif operation == 'delete':
            bst.delete(key)
        elif operation == 'exists':
            output.write('true\n' if bst.exists(key)
else 'false\n')
        elif operation == 'next':
            result = bst.next(key)
            output.write(str(result) + '\n' if result
is not None else 'none\n')
        elif operation == 'prev':
            result = bst.prev(key)
            output.write(str(result) + '\n' if result
is not None else 'none\n')

```

Объяснение кода:

Класс Node - Представляет отдельный узел в дереве.

Имеет атрибуты key (значение узла), left (ссылка на левый дочерний узел) и right (ссылка на правый дочерний узел).

Класс BinarySearchTree - Представляет бинарное дерево поиска.

Имеет атрибут root (ссылка на корневой узел).

Метод insert(key): Вставляет новый узел с заданным ключом (key) в дерево.

Рекурсивно проходит по дереву, сравнивая ключ с ключами существующих узлов. Если ключ меньше текущего ключа, вставка происходит в левое поддерево, иначе - в правое.

delete(key): Удаляет узел с заданным ключом (key) из дерева. Если узел не найден, ничего не происходит. Если узел не имеет дочерних узлов, он просто удаляется. Если узел имеет только один дочерний узел, он заменяется этим дочерним узлом. Если узел имеет два дочерних узла, его ключ заменяется минимальным ключом из правого поддерева, после чего минимальный узел удаляется из правого поддерева.

exists(key): Проверяет, существует ли узел с заданным ключом (key) в дереве.

Рекурсивно проходит по дереву, сравнивая ключ с ключами существующих узлов. Возвращает True, если узел найден, иначе - False.


next(key): Возвращает следующий по величине ключ в дереве после заданного ключа (key). если меньше всех – none

prev(key): Возвращает предыдущий по величине ключ в дереве до заданного ключа (key)

Результат работы кода на примерах из текста задачи:

≡ input.txt ×


≡ output.txt

 11.py

1	insert 2
2	insert 5
3	insert 3
4	exists 2
5	exists 4
6	next 4
7	prev 4
8	delete 5
9	next 4
10	prev 4

≡ input.txt

≡ output.txt ×

 11.py

1	true
2	false
3	5
4	3
5	<u>none</u>
6	<u>none</u>
7	

## Задача №16. К-й максимум

### Текст задачи

Напишите программу, реализующую структуру данных, позволяющую добавлять и удалять элементы, а также находить  $k$ -й максимум.

- **Формат ввода / входного файла (input.txt).** Первая строка входного файла содержит натуральное число  $n$  – количество команд. Последующие  $n$  строк содержат по одной команде каждая. Команда записывается в виде двух чисел  $c_i$  и  $k_i$  – тип и аргумент команды соответственно. Поддерживаемые команды:

- +1 (или просто 1): Добавить элемент с ключом  $k_i$ .
- 0 : Найти и вывести  $k_i$ -й максимум.
- -1 : Удалить элемент с ключом  $k_i$ .

Гарантируется, что в процессе работы в структуре не требуется хранить элементы с равными ключами или удалять несуществующие элементы. Также гарантируется, что при запросе  $k_i$ -го макс-симула, он существует.

- **Ограничения на входные данные.**  $n \leq 100000$ ,  $|k_i| \leq 10^9$ .

### Листинг кода:

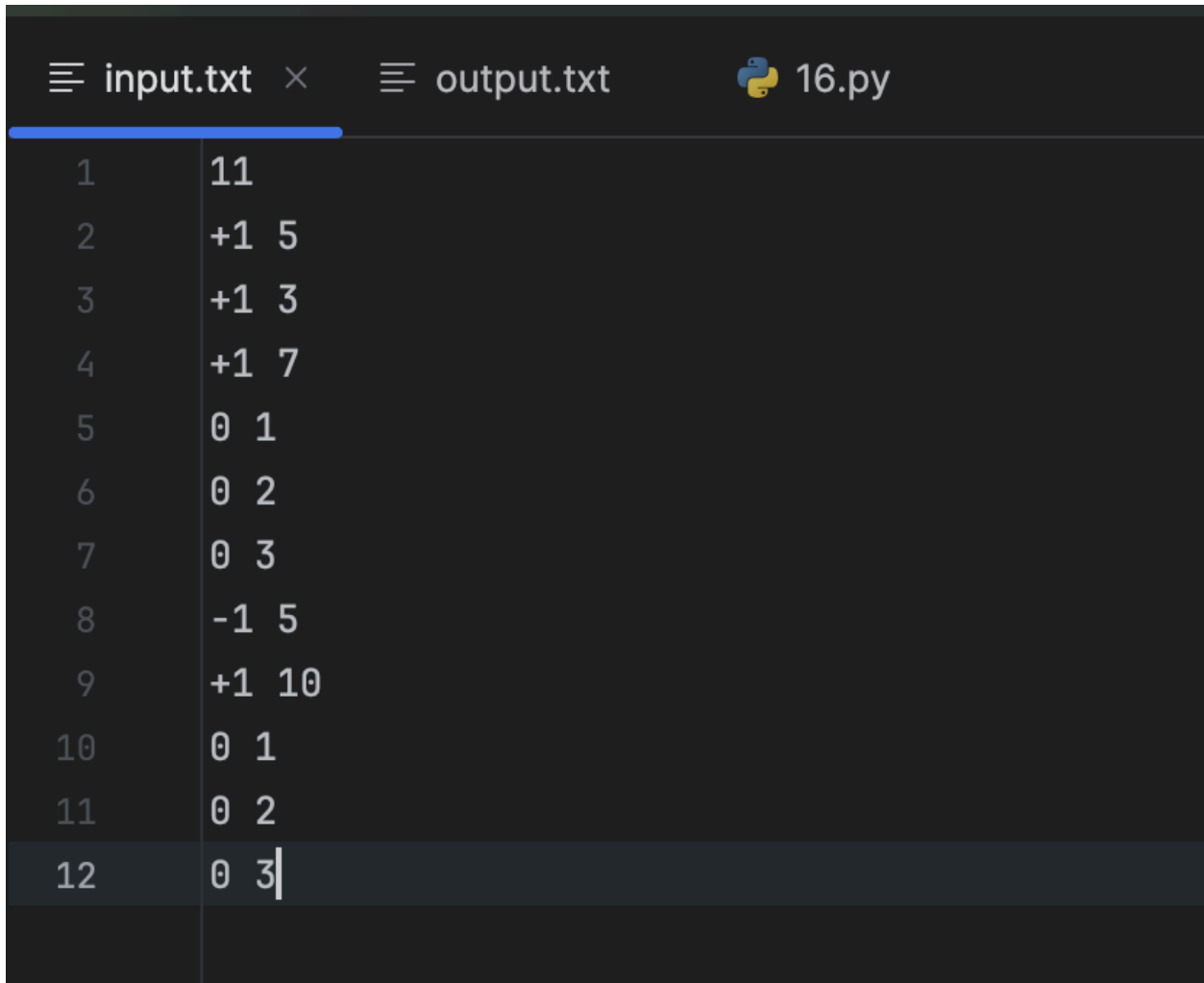
```
f = open("input.txt")
n = int(f.readline())
rez = []
z = open("output.txt", "w")
for i in range(n):
    s = f.readline().split()
    if s[0] == "+1": #записывает число в rez
        rez.append(int(s[1]))
        rez.sort()
    elif s[0] == "0": # находит последнее и записывает
в output
        z.write(str(rez[len(rez) - int(s[1])]) + "\n")
    elif s[0] == "-1": # удаляет из rez
        rez.remove(int(s[1]))
```

### Объяснение кода

Код считывает команды, обрабатывает их (добавляет, удаляет или ищет элементы) и записывает результаты в выходной файл output.txt. Код использует список rez для хранения отсортированных элементов, и реализует

три типа команд: "+1 x" добавляет элемент x в список, "-1 x" удаляет элемент x из списка, "0 x" выводит x-й наименьший элемент списка.

Результат работы кода на примерах из текста задачи:



The screenshot shows a code editor with two tabs: 'input.txt' and 'output.txt'. The 'input.txt' tab is active and displays a list of 12 commands. The 'output.txt' tab is empty. The commands in 'input.txt' are as follows:

Line	Command
1	11
2	+1 5
3	+1 3
4	+1 7
5	0 1
6	0 2
7	0 3
8	-1 5
9	+1 10
10	0 1
11	0 2
12	0 3

☰ input.txt

☰ output.txt ×

🐍 16.py

1 7

2 5

3 3

4 10

5 7

6 3

7 |