Министерство науки и высшего образования Российской Федерации
федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский институт ИТМО»

Факультет МР и П

Алгоритмы и структуры данных

Лабораторная работа №2.

«Двоичные деревья поиска»

Вариант «9»

Выполнил студент:

Розметов Джалолиддин

Группа № D3210

Преподаватель: Артамонова Валерия Евгеньевна

г. Санкт-Петербург

2024

Оглавление

Задание 1

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. $1 \le x \le 10^9$, $1 \le N \le 300000$, 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	

Код

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

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

    def insert(self, key):
        if not self.root:
            self.root = TreeNode(key)
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        if key < node.key:
            if node.left:
                self._insert(node.left, key)
            else:
                node.left = TreeNode(key)
            node.size += 1
        elif key > node.key:
            if node.right:
                self._insert(node.right, key)
            else:
                node.right = TreeNode(key)
            node.size += 1

    def kth_smallest(self, k):
        return self._kth_smallest(self.root, k)

    def _kth_smallest(self, node, k):
        left_size = node.left.size if node.left else 0

        if k == left_size + 1:
            return node.key
        elif k <= left_size:
            return self._kth_smallest(node.left, k)
        else:
```

```
            return self._kth_smallest(node.right, k - left_size - 1)

with open('input.txt', 'r') as file:
    lines = file.readlines()

bst = BST()
results = []

for line in lines:
    if line.startswith('+'):
        _, x = line.split()
        x = int(x)
        bst.insert(x)
    elif line.startswith('?'):
        _, k = line.split()
        k = int(k)
        results.append(bst.kth_smallest(k))

with open('output.txt', 'w') as file:
    for result in results:
        file.write(f"{result}\n")
```

Вводимые данные:

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

Вывод кода:

1
3
4
3

Описание кода:

1. Инициализирует бинарное дерево поиска (BST).
2. Читает команды из файла и:

- Вставляет элементы в BST при команде "+".
- Обрабатывает запросы на поиск k-ого наименьшего элемента при команде "?".

3. Записывает результаты запросов в выходной файл.

Описание проведенных тестов.

Тесты для кода проверяли корректность вставки элементов в бинарное дерево поиска и точность функции нахождения k-го наименьшего элемента в дереве. Проверка

включала различные последовательности операций добавления и запросов, а также их влияние на структуру дерева и результаты запросов.

Вывод по работе кода:
Код успешно обрабатывает запросы вставки элементов и поиска k-го наименьшего элемента в бинарном дереве поиска. Тестирование подтверждает его корректность и эффективность работы.

Задание 2

Дано некоторое двоичное дерево поиска. Также даны запросы на удаление из него вершин, имеющих заданные ключи, причем вершины удаляются целиком вместе со своими поддеревьями. После каждого запроса на удаление выведите число оставшихся вершин в дереве. В вершинах данного дерева записаны ключи – целые числа, по модулю не превышающие 10^9. Гарантируется, что данное дерево является двоичным деревом поиска, в частности, 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 ключа вершины V .
Высота дерева не превосходит 25, таким образом, можно считать, что оно балансировано.
• Формат ввода / входного файла (input.txt). Входной файл содержит описание двоичного дерева и описание запросов на удаление.
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 ($1 \le i \le N$) находится описание i-ой вершины, состоящее из трех чисел Ki, Li, Ri, разделенных пробелами – ключа Ki в i-ой вершине, номера левого Li ребенка i-ой вершины ($i < Li \le N$ или Li = 0, если левого ребенка нет) и номера правого Ri ребенка i-ой вершины ($i < Ri \le N$ или Ri = 0, если правого ребенка нет).
Все ключи различны. Гарантируется, что данное дерево является деревом поиска.
В следующей строке находится число M – число запросов на удаление. В следующей строке находятся M чисел, разделенных пробелами – ключи, вершины с которыми (вместе с их поддеревьями) необходимо удалить. Все эти числа не превосходят 109 по абсолютному значению. Вершина с таким ключом не обязана существовать в дереве – в этом случае дерево изменять не требуется. Гарантируется, что корень дерева никогда не будет удален.
• Ограничения на входные данные. $1 \le N \le 2 \cdot 10^5$, $|Ki| \le 10^9$, $1 \le M \le 2 \cdot 10^5$

• Формат вывода / выходного файла (output.txt). Выведите M строк. На i-ой строке требуется вывести число

вершин, оставшихся в дереве после выполнения i-го запроса на удаление.

- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
6 -2 0 2 8 4 3 9 0 0 3 6 5 6 0 0 0 0 0 4 6 9 7 8	5 4 4 1

Код

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

def build_tree(nodes_info):
    nodes = [None] * (len(nodes_info) + 1)
    for i, (key, left, right) in enumerate(nodes_info):
        if nodes[i + 1] is None:
            nodes[i + 1] = TreeNode(key)
        nodes[i + 1].key = key
        if left != 0:
            nodes[i + 1].left = nodes[left] if nodes[left] else
TreeNode(None)
            nodes[left] = nodes[i + 1].left
        if right != 0:
            nodes[i + 1].right = nodes[right] if nodes[right] else
TreeNode(None)
            nodes[right] = nodes[i + 1].right
    return nodes[1]  # root node

def count_nodes(node):
    if not node:
        return 0
    return 1 + count_nodes(node.left) + count_nodes(node.right)

def delete_subtree(root, key):
    if not root:
        return None, 0
    if key < root.key:
        root.left, removed_count = delete_subtree(root.left, key)
    elif key > root.key:
        root.right, removed_count = delete_subtree(root.right, key)
    else:
        removed_count = count_nodes(root)
        return None, removed_count
    return root, removed_count
```

```
def process_deletions(root, deletions):
    remaining_nodes = count_nodes(root)
    results = []
    for key in deletions:
        root, removed_count = delete_subtree(root, key)
        remaining_nodes -= removed_count
        results.append(remaining_nodes)
    return results

with open('input.txt', 'r') as file:
    N = int(file.readline().strip())
    nodes_info = [tuple(map(int, file.readline().strip().split())) for _ in
range(N)]
    M = int(file.readline().strip())
    deletions = list(map(int, file.readline().strip().split()))

root = build_tree(nodes_info)

results = process_deletions(root, deletions)

with open('output.txt', 'w') as file:
    for result in results:
        file.write(f"{result}\n")
```

Вводимые данные:
6
-2 0 2
8 4 3
9 0 0
3 6 5
6 0 0
0 0 0
4
6 9 7 8

Вывод кода:
5
4
4
1

Описание кода:

1. Инициализация узлов: Создание узлов TreeNode из предоставленных данных. Каждый узел характеризуется ключом и ссылками на левое и правое поддеревья.
2. Построение дерева: Функция build_tree читает входные данные, создает узлы и связывает их в дерево, возвращая корень дерева.

3. Подсчет узлов: Функция count_nodes рекурсивно считает количество узлов в поддереве.
4. Удаление поддеревьев: delete_subtree удаляет поддерево с заданным ключом, возвращая новое дерево и количество удаленных узлов.
5. Обработка удалений: Функция process_deletions удаляет поддеревья для списка ключей и сохраняет количество узлов после каждого удаления.
6. Запись результатов: Количество узлов после каждого удаления записывается в выходной файл.

Описание проведенных тестов:
Тесты проверяли правильность построения дерева из входных данных, корректность удаления поддеревьев по заданным ключам, и правильное обновление количества оставшихся узлов после каждой операции удаления. Тестировались различные структуры деревьев и последовательности удалений, чтобы убедиться в корректности и устойчивости алгоритма.

Вывод по работе кода:
Код успешно строит бинарное дерево поиска из входных данных, корректно удаляет поддеревья по заданным ключам и точно отслеживает количество оставшихся узлов после каждого удаления. Тестирование подтвердило его правильность и эффективность.

Задание 3

В этой задаче ваша цель – реализовать структуру данных для хранения набора целых чисел и быстрого вычисления суммы элементов в заданном диапазоне. Реализуйте такую структуру данных, в которой хранится набор целых чисел S и доступны следующие операции:

- add(i) – добавить число i в множество S. Если i уже есть в S, то ничего делать не надо;
- del(i) – удалить число i из множества S. Если i нет в S, то ничего делать не надо;
- find(i) – проверить, есть ли i во множестве S или нет;
- sum(l, r) – вывести сумму всех элементов v из S таких, что $l \le v \le r$.

• Формат ввода / входного файла (input.txt). Изначально множество S пусто. Первая строка содержит n – количество операций. Следующие n строк содержат операции. Однако, чтобы убедиться, что ваше решение может работать в режиме онлайн, каждый запрос фактически будет зависеть от результата последнего запроса суммы. Обозначим M = 1 000 000 001. В любой момент пусть x будет результатом последней

операции суммирования или просто 0, если до этого операций суммирования не было. Тогда каждая операция будет являться одной из следующих: – «+ i» – добавить некое число в множество S. Но не само число i, а число ((i + x) mod M). – «– i» – удалить из множества S, т.е. del((i + x) mod M). – «? i» – find((i + x) mod M). – «s l r» – вывести сумму всех элементов множества S из определенного диапазона, т.е. sum((l+x) mod M,(r+x) mod M).

• Ограничения на входные данные. $1 \le n \le 100000$, $1 \le i \le 10^9$.\

• Формат вывода / выходного файла (output.txt). Для каждого запроса «find», выведите только «Found» или «Not found» (без кавычек, первая буква заглавная) в зависимости от того, есть ли число ((i + x) mod M) в S или нет. Для каждого запроса суммы «sum» выведите сумму всех значений v из S из диапазона (l + x) mod M ≤ v ≤ (r + x) mod M, где x – результат подсчета прошлой суммы «sum», или 0, если еще не было таких операций.

• Ограничение по времени. 120 сек. Python

• Ограничение по памяти. 512 мб.

• Пример:

input	output
15 ? 1 + 1 ? 1 + 2 s 1 2 + 1000000000 ? 1000000000 - 1000000000 ? 1000000000 s 999999999 1000000000 - 2 ? 2 - 0 + 9 s 0 9	Not found Found 3 Found Not found 1 Not found 10

Код

```
class DataStructure:
    def __init__(self):
        self.elements = set()
        self.sorted_list = []

    def add(self, value):
        if value not in self.elements:
            self.elements.add(value)
            self.sorted_list.append(value)
            self.sorted_list.sort()

    def delete(self, value):
        if value in self.elements:
            self.elements.remove(value)
            self.sorted_list.remove(value)

    def find(self, value):
        return value in self.elements

    def range_sum(self, left, right):
        return sum(v for v in self.sorted_list if left <= v <= right)

# Чтение входных данных
with open('input.txt', 'r') as file:
    n = int(file.readline().strip())
    operations = [file.readline().strip() for _ in range(n)]

M = 1000000001
x = 0  # результат последней операции sum или 0

ds = DataStructure()
results = []

for operation in operations:
    if operation[0] == '+':
        _, i = operation.split()
        i = int(i)
        value = (i + x) % M
        ds.add(value)
    elif operation[0] == '-':
        _, i = operation.split()
        i = int(i)
        value = (i + x) % M
        ds.delete(value)
    elif operation[0] == '?':
        _, i = operation.split()
        i = int(i)
        value = (i + x) % M
        if ds.find(value):
            results.append("Found")
        else:
            results.append("Not found")
    elif operation[0] == 's':
        _, l, r = operation.split()
        l = int(l)
        r = int(r)
        left = (l + x) % M
        right = (r + x) % M
        if left <= right:
            x = ds.range_sum(left, right)
        else:
```

```
            x = ds.range_sum(left, M - 1) + ds.range_sum(0, right)
        results.append(str(x))

# Запись результатов
with open('output.txt', 'w') as file:
    for result in results:
        file.write(result + "\n")
```

Вводимые данные:
15
? 1
+ 1
? 1
+ 2
s 1 2
+ 1000000000
? 1000000000
- 1000000000
? 1000000000
s 999999999 1000000000
- 2
? 2
- 0
+ 9
s 0 9

Вывод кода:
Not found
Found
3
Found
Not found
1
Not found
10

Описание кода:

1. Чтение данных из input.txt.
2. Инициализация: переменных, экземпляра DataStructure, списка results.
3. Обработка операций: добавление, удаление, поиск, суммирование.
4. Запись результатов в output.txt

Описание проведенных тестов:
Тесты показали, что структура данных корректно выполняет операции добавления, удаления, поиска и суммирования элементов в заданном диапазоне, учитывая модульные операции и влияние переменной x на результаты.

Вывод по работе кода:
Код корректно реализует структуру данных для управления множеством целых чисел, поддерживая операции добавления, удаления, поиска и суммирования элементов в заданном диапазоне. Все операции учитывают модульные преобразования и влияния предыдущих сумм. Тесты подтверждают правильность работы кода в различных сценариях.

Задание 4

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. $0 \le N \le 10^5$, $|xi| \le 10^9$.

• 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	

Код

```
# Задание 11
class Node:
    def __init__(self, key):
        self.key = key
        self.height = 1
        self.left = None
        self.right = None

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

    def get_height(self, node):
        if not node:
            return 0
        return node.height

    def update_height(self, node):
        node.height = max(self.get_height(node.left),
self.get_height(node.right)) + 1

    def get_balance(self, node):
        if not node:
            return 0
        return self.get_height(node.left) - self.get_height(node.right)

    def right_rotate(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        self.update_height(y)
        self.update_height(x)
        return x

    def left_rotate(self, x):
        y = x.right
        T2 = y.left
        y.left = x
        x.right = T2
        self.update_height(x)
```

```
        self.update_height(y)
        return y

    def balance_node(self, node):
        self.update_height(node)
        balance = self.get_balance(node)
        if balance > 1:
            if self.get_balance(node.left) < 0:
                node.left = self.left_rotate(node.left)
            return self.right_rotate(node)
        if balance < -1:
            if self.get_balance(node.right) > 0:
                node.right = self.right_rotate(node.right)
            return self.left_rotate(node)
        return node

    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)
        else:
            return node
        return self.balance_node(node)

    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
            temp = self.get_min_value_node(node.right)
            node.key = temp.key
            node.right = self.delete(node.right, temp.key)
        return self.balance_node(node)

    def get_min_value_node(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current

    def get_max_value_node(self, node):
        current = node
        while current.right is not None:
            current = current.right
        return current

    def exists(self, node, key):
        if not node:
            return False
        if key < node.key:
            return self.exists(node.left, key)
        elif key > node.key:
            return self.exists(node.right, key)
        else:
```

```
            return True

    def next(self, node, key):
        successor = None
        while node:
            if key < node.key:
                successor = node
                node = node.left
            else:
                node = node.right
        return successor

    def prev(self, node, key):
        predecessor = None
        while node:
            if key > node.key:
                predecessor = node
                node = node.right
            else:
                node = node.left
        return predecessor

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

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

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

    def next_key(self, key):
        node = self.next(self.root, key)
        return node.key if node else None

    def prev_key(self, key):
        node = self.prev(self.root, key)
        return node.key if node else None

# Чтение входных данных
with open('input.txt', 'r') as file:
    operations = [line.strip() for line in file]

avl_tree = AVLTree()
results = []

for operation in operations:
    command = operation.split()
    op = command[0]
    x = int(command[1])

    if op == "insert":
        avl_tree.insert_key(x)
    elif op == "delete":
        avl_tree.delete_key(x)
    elif op == "exists":
        result = "true" if avl_tree.exists_key(x) else "false"
        results.append(result)
    elif op == "next":
        result = avl_tree.next_key(x)
        results.append(str(result) if result is not None else "none")
    elif op == "prev":
        result = avl_tree.prev_key(x)
        results.append(str(result) if result is not None else "none")
```

```
# Запись результатов
with open('output.txt', 'w') as file:
    for result in results:
        file.write(result + "\n")
```

Вводимые данные:
insert 2
insert 5
insert 3
exists 2
exists 4
next 4
prev 4
delete 5
next 4
prev 4

Вывод данных:
true
false
5
3
none
3

Описание кода:

1. Чтение данных из файла input.txt.
2. Инициализация экземпляра AVLTree.
3. Обработка операций:
    - insert x: Вставка ключа x.
    - delete x: Удаление ключа x.
    - exists x: Проверка существования ключа x и запись результата в results.
    - next x: Поиск следующего по величине ключа и запись результата в results.
    - prev x: Поиск предыдущего по величине ключа и запись результата в results.
4. Запись результатов в файл output.txt.

Описание проведенных тестов:

Проведенные тесты проверяют корректность работы AVL-дерева. В тестах проверяется вставка элементов, их удаление, существование, а также поиск следующего и предыдущего ключей. Тесты подтверждают, что дерево правильно выполняет балансировку, обеспечивая эффективную работу всех операций.

Выводы по работе кода:

Код корректно реализует AVL-дерево, эффективно выполняя операции вставки, удаления, проверки существования, и поиска соседних элементов. Балансировка дерева после каждой операции обеспечивает высокую производительность и корректность работы алгоритмов.

Вывод:

По результатам лабораторной работы с двоичными деревьями поиска, программа успешно демонстрирует способность строить и управлять структурами данных для эффективного выполнения операций поиска, вставки и удаления. Работа подтвердила, что двоичные деревья поиска обеспечивают хорошую производительность для управления упорядоченными данными, а также показала важность балансировки деревьев для поддержания их оптимальной эффективности. Это подчёркивает их пригодность для использования в приложениях, где требуется быстрый доступ к данным.