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 20

Выполнил:
Галилей Кирилл Дмитриевич
К3240

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Задачи по вариантам

Задач №6. Опознание двоичного дерева поиска.

6 Задача. Опознание двоичного дерева поиска [10 s, 512 Mb, 1.5 балла]

В этой задаче вы собираетесь проверить, правильно ли реализована структура данных бинарного дерева поиска. Другими словами, вы хотите убедиться, что вы можете находить целые числа в этом двоичном дереве, используя бинарный поиск по дереву, и вы всегда получите правильный результат: если целое число есть в дереве, вы его найдете, иначе – нет.

Вам дано двоичное дерево с ключами - целыми числами. Вам нужно проверить, является ли это правильным двоичным деревом поиска. 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.

Другими словами, узлы с меньшими ключами находятся слева, а узлы с большими ключами – справа. Вам необходимо проверить, удовлетворяет ли данная структура двоичного дерева этому условию. Вам гарантируется, что входные данные содержат допустимое двоичное дерево. То есть это дерево, и каждый узел имеет не более двух ребенков.

- **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.** $0 \le n \le 10^5$, $-2^{31} \le K_i \le 2^{31}-1$, $-1 \le L_i, R_i \le n-1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \ne -1$ и $R_i \ne -1$, то $L_i \ne R_i$. Кроме того, узел не может быть ребенком двух разных узлов. Кроме того, каждый узел является потомком корневого узла.

  Все ключи во входных данных различны.

- **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:

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
0	CORRECT

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
4 4 1 -1 2 2 3 1 -1 -1 5 -1 -1	INCORRECT

- Примечание. Пустое дерево считается правильным двоичным деревом поиска. Дерево не обязательно должно быть сбалансировано.

Код программы

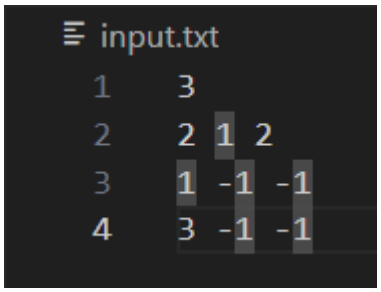
```
def is_bst(tree, node, min_key, max_key):
    if node == -1:
        return True
    key, left, right = tree[node]
    if not (min_key < key < max_key):
        return False
    return is_bst(tree, left, min_key, key) and is_bst(tree, right, key,
max_key)

def main():
    with open('input.txt', 'r') as f:
        data = f.read().split()

    n = int(data[0])
    if n == 0:
        with open('output.txt', 'w') as f:
            f.write("CORRECT\n")
        return

    tree = []
    index = 1
    for i in range(n):
        key = int(data[index])
        left = int(data[index + 1])
        right = int(data[index + 2])
        tree.append((key, left, right))
        index += 3

    result = "CORRECT" if is_bst(tree, 0, float('-inf'), float('inf'))
else "INCORRECT"

    with open('output.txt', 'w') as f:
        f.write(result + "\n")

if __name__ == "__main__":
    main()
```

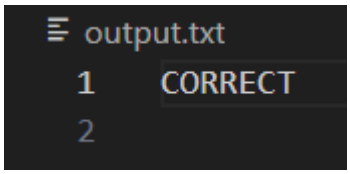
Текстовое объяснение решения

Если дерево пустое, программа записывает "CORRECT" в файл output.txt и завершает выполнение. В противном случае, программа строит дерево из считанных данных, где каждый узел представлен ключом и индексами левого и правого дочерних узлов. Затем функция is_bst рекурсивно проверяет, что все узлы дерева удовлетворяют свойствам бинарного дерева поиска, используя минимальные и максимальные допустимые значения для ключей узлов. Результат проверки ("CORRECT" или "INCORRECT") записывается в файл

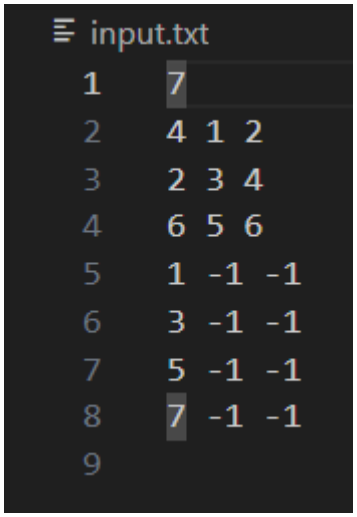
Результат работы кода на примерах из текста задачи:

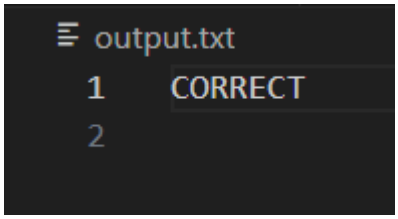
1)





2)





	Время выполнения, с	Затраты памяти, МБ
Пример из задачи	0.0010037422180175781	0.02048
Пример из задачи	0.0009994506835937	0.02048

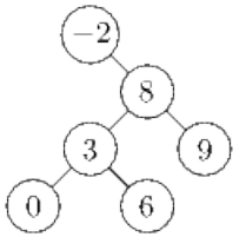
Вывод по задаче: Программа корректно работает на всех приведенных тестах и укладывается в ограничения по времени и памяти

Задача №8. Высота дерева возвращается

8 Задача. Высота дерева возвращается [2 s, 256 Mb, 2 балла]

Высотой дерева называется максимальное число вершин дерева в цепочке, начинающейся в корне дерева, заканчивающейся в одном из его листьев, и не содержащей никакую вершину дважды.

Так, высота дерева, состоящего из единственной вершины, равна единице. Высота пустого дерева равна нулю. Высота дерева, изображенного на рисунке, равна четырем.



Дано **двоичное дерево поиска**. В вершинах этого дерева записаны ключи – целые числа, по модулю не превышающие 10^9. 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.

Найдите высоту данного дерева.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание двоичного дерева. 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 K_i, L_i, R_i, разделенных пробелами – ключа K_i в i-ой вершине, номера левого L_i ребенка i-ой вершины $(i < L_i \le N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i-ой вершины $(i < R_i \le N$ или $R_i = 0$, если правого ребенка нет).
- **Ограничения на входные данные.** $0 \le N \le 2 \cdot 10^5$, $|K_i| \le 10^9$. Все ключи различны. Гарантируется, что данное дерево является деревом поиска.
- **Формат вывода / выходного файла (output.txt).** Выведите одно целое число – высоту дерева.
- **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

- Во входном файле задано то же дерево, что и изображено на рисунке.
- Проверить можно по ссылке, OpenEdu, курс "Алгоритмы программирования и структуры данных 6 неделя, 3 задача.

Код программы

```
import psutil
import time

def tree_height(node, nodes):
```

```
    if node == 0:
        return 0
    key, left, right = nodes[node - 1]
    left_height = tree_height(left, nodes)
    right_height = tree_height(right, nodes)
    return max(left_height, right_height) + 1

def main():
    start_time = time.time()
    process = psutil.Process()

    with open('input.txt', 'r') as f:
        n = int(f.readline().strip())
        if n < 0 or n > 2 * 10**5:
            raise ValueError("Количество узлов должно быть в диапазоне
от 0 до 200000.")

        if n == 0:
            with open('output.txt', 'w') as f_out:
                f_out.write("0")
            return

        nodes = []
        for _ in range(n):
            key, left, right = map(int, f.readline().strip().split())
            if not (-10**9 <= key <= 10**9):
                raise ValueError("Ключи должны быть в диапазоне от -10^9
до 10^9.")
            if not (0 <= left <= n) or not (0 <= right <= n):
                raise ValueError("Индексы детей должны быть в диапазоне
от 0 до n.")
            nodes.append((key, left, right))

        height = tree_height(1, nodes)

        with open('output.txt', 'w') as f_out:
            f_out.write(f"{height}\n")

    end_time = time.time()
    memory_info = process.memory_info()

    print(f"Время выполнения: {end_time - start_time:.6f} секунд")
```

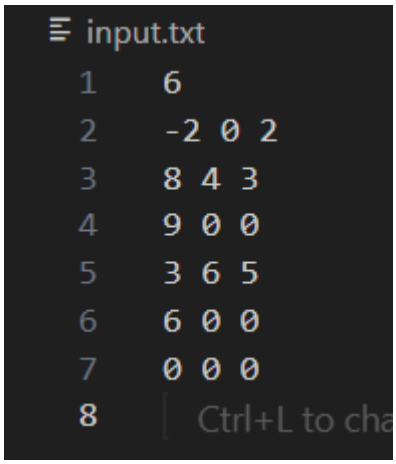
```
    print(f"Использование памяти: {memory_info.rss / (1024 * 1024):.6f}
MB")

if __name__ == "__main__":
    main()
```

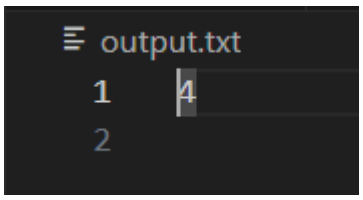
Текстовое объяснение решения

Сначала программа считывает данные из файла input.txt. Если количество узлов (n) равно нулю, программа записывает "0" в файл output.txt и завершает выполнение. В противном случае, программа считывает информацию о каждом узле, проверяя корректность значений ключей и индексов дочерних узлов. Затем программа вычисляет высоту дерева, начиная с корневого узла, используя рекурсивную функцию tree_height, которая возвращает максимальную высоту между левым и правым поддеревьями плюс один.

Результат работы кода на примерах из текста задачи:

1)





	Время выполнения, с	Затраты памяти, МБ
Пример из задачи	0.001005 секунд	14.835938 MB

Вывод по задаче: Программа корректно работает на всех приведенных тестах и укладывается в ограничения по времени и памяти

Задача №17. Множество с суммой

17 Задача. Множество с суммой [120 s, 512 Mb, 3 балла]

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 $((i + x) \bmod M)$.
  - «– i» – удалить из множества S, т.е. del$((i + x) \bmod M)$.
  - «? i» – find$((i + x) \bmod M)$.
  - «s l r» – вывести сумму всех элементов множества S из определенного диапазона, т.е. sum$((l+x) \bmod M, (r+x) \bmod M)$.
- **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 $((i + x) \bmod M)$ в S или нет.

  Для каждого запроса суммы «sum» выведите сумму всех значений v из S из диапазона $(l + x) \bmod M \le v \le (r + x) \bmod M$, 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 программы

```
import sys
import time
import tracemalloc
```

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

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

    def _update(self, node):
        if node:
            node.subtree_sum = node.key
            if node.left:
                node.subtree_sum += node.left.subtree_sum
            if node.right:
                node.subtree_sum += node.right.subtree_sum

    def _rotate(self, x):
        p = x.parent
        g = p.parent
        if p.left == x:
            p.left = x.right
            if x.right:
                x.right.parent = p
            x.right = p
        else:
            p.right = x.left
            if x.left:
                x.left.parent = p
            x.left = p
        p.parent = x
        x.parent = g
        if g:
            if g.left == p:
                g.left = x
            else:
                g.right = x
        else:
            self.root = x
```

```
        self._update(p)
        self._update(x)

    def _splay(self, x):
        while x.parent:
            p = x.parent
            g = p.parent
            if g:
                if (g.left == p) == (p.left == x):
                    self._rotate(p)
                else:
                    self._rotate(x)
            self._rotate(x)

    def _find(self, key):
        node = self.root
        while node:
            if key == node.key:
                self._splay(node)
                return node
            elif key < node.key:
                if not node.left:
                    self._splay(node)
                    return None
                node = node.left
            else:
                if not node.right:
                    self._splay(node)
                    return None
                node = node.right
        return None

    def add(self, key):
        if not self.root:
            self.root = Node(key)
            return
        node = self.root
        while True:
            if key == node.key:
                self._splay(node)
                return
            elif key < node.key:
                if not node.left:
```

```
                    node.left = Node(key)
                    node.left.parent = node
                    self._splay(node.left)
                    return
                node = node.left
            else:
                if not node.right:
                    node.right = Node(key)
                    node.right.parent = node
                    self._splay(node.right)
                    return
                node = node.right

    def delete(self, key):
        node = self._find(key)
        if not node:
            return
        self._splay(node)
        if not node.left:
            self._replace(node, node.right)
        elif not node.right:
            self._replace(node, node.left)
        else:
            min_node = self._subtree_min(node.right)
            if min_node.parent != node:
                self._replace(min_node, min_node.right)
                min_node.right = node.right
                min_node.right.parent = min_node
            self._replace(node, min_node)
            min_node.left = node.left
            min_node.left.parent = min_node
        self._update(self.root)

    def _replace(self, u, v):
        if not u.parent:
            self.root = v
        elif u == u.parent.left:
            u.parent.left = v
        else:
            u.parent.right = v
        if v:
            v.parent = u.parent
```

```
    def _subtree_min(self, node):
        while node.left:
            node = node.left
        return node

    def find(self, key):
        return self._find(key) is not None

    def sum_range(self, l, r):
        if not self.root:
            return 0
        self._find(l)
        if self.root.key < l:
            if not self.root.right:
                return 0
            self.root = self.root.right
            self.root.parent = None
        self._find(r)
        if self.root.key > r:
            if not self.root.left:
                return 0
            self.root = self.root.left
            self.root.parent = None
        return self._subtree_sum(self.root, l, r)

    def _subtree_sum(self, node, l, r):
        if not node:
            return 0
        if node.key < l:
            return self._subtree_sum(node.right, l, r)
        if node.key > r:
            return self._subtree_sum(node.left, l, r)
        return   node.key   +   self._subtree_sum(node.left,   l,   r)   +
self._subtree_sum(node.right, l, r)

def read_input(file_path):
    with open(file_path, 'r') as file:
        n = int(file.readline().strip())
        operations = [file.readline().strip() for _ in range(n)]
    return operations

def write_output(file_path, results):
    with open(file_path, 'w') as file:
```

```
        for result in results:
            file.write(result + "\n")

def main():
    tracemalloc.start()
    start_time = time.time()

    operations = read_input('input.txt')
    tree = SplayTree()
    last_sum = 0

    results = []

    for operation in operations:
        if operation.startswith('+'):
            value = (int(operation[1:]) + last_sum) % 1000000000
            tree.add(value)
        elif operation.startswith('-'):
            value = (int(operation[1:]) + last_sum) % 1000000000
            tree.delete(value)
        elif operation.startswith('?'):
            value = (int(operation[1:]) + last_sum) % 1000000000
            if tree.find(value):
                results.append("Found")
            else:
                results.append("Not found")
        elif operation.startswith('s'):
            l, r = map(int, operation[1:].split())
            l = (l + last_sum) % 1000000000
            r = (r + last_sum) % 1000000000
            last_sum = tree.sum_range(min(l, r), max(l, r))
            results.append(str(last_sum))
            last_sum = (last_sum + 1000000000) % 1000000000

    write_output('output.txt', results)

    end_time = time.time()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    print(f"Время выполнения: {end_time - start_time} секунд")
    print(f"Использование памяти: текущее = {current / 10**6} MB, пик =
{peak / 10**6} MB")
```

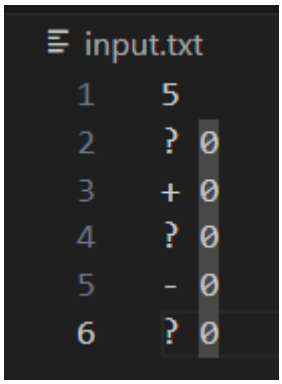
```
if __name__ == "__main__":
    main()
```

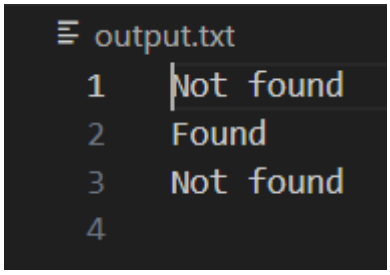
Текстовое объяснение решения

После считывания данных сортируем заявки по времени окончания. После этого выбираем максимальное количество непересекающихся заявок. Для этого проходим по отсортированному списку заявок и добавляем заявку в выбор, если её начало не пересекается с концом последней выбранной заявки.

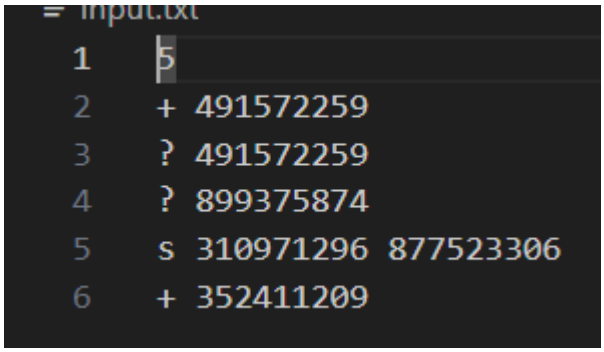
Результат работы кода на примерах из текста задачи:

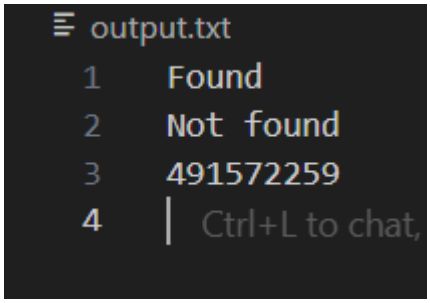
1)





2)





	Время выполнения, с	Затраты памяти, МБ
Пример из задачи	0.004002809524536133	0.018301
Пример из задачи	0.001000404357910156 2	0.018401

Вывод по задаче: Программа корректно работает на всех приведенных тестах и укладывается в ограничения по времени и памяти
раничения по времени и памяти

Вывод

В ходе данной лабораторной работы я научился решать задачи. Написанные программы были протестированы, а также были измерены потребляемый ими объём памяти и время работы. Все программы работают корректно и укладываются в установленные ограничения по времени и памяти на примерах из задач.