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

Выполнил:
Малыхин Н.С.
К3141

Санкт-Петербург 2025

Содержание

Задачи

Задание №1. Обход двоичного дерева (1 балл)

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, ..., n − 1 по порядку. Каждая из этих строк содержит три целых числа Ki, Li и Ri. Ki – ключ i-го узла, Li - индекс левого ребенка i-го узла, а Ri - индекс правого ребенка i-го узла. Если у i-го узла нет левого или правого ребенка (или обоих), соответствующие числа Li или Ri (или оба) будут равны −1.
- Ограничения на входные данные. 1 ≤ n ≤ 105, 0 ≤ Ki ≤ 109, −1 ≤ Li, Ri ≤ n−1. Гарантируется, что данное дерево является двоичным деревом. В частности, если Li≠ −1 и Ri≠ −1, то Li≠ Ri
- 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).

Решение:

```
from utils.file_utlis import read_from_file, write_to_file
from utils.time_memory_utlis import measure_time_and_memory
```

```python
input_path = "../txtf/input.txt"
output_path = "../txtf/output.txt"


class Node:
    """Класс для представления узла дерева"""
    def __init__(self, key, left, right):
        self.key = key
        self.right = right
        self.left = left


def parse_tree(n, nodes):
    """Функция для парсинга дерева"""
    tree = {}
    for i in range(n):
        k, l, r = nodes[i]
        tree[i] = Node(k, l, r)
    return tree


def in_order(tree, root):
    """Центрированный обход дерева (in_order)"""
    stack = [] # Стек для хранения узлов
    node = root # Начинаем с корня
    result = []

    while stack or node != -1:
        if node != -1:
            # Идем до самого левого узла, добавляя все узлы в стек
            stack.append(node)
            node = tree[node].left
        else:
            # Достаем узел из стека и обрабатываем его
            node = stack.pop()
            result.append(tree[node].key)
            # Переходим к правому поддереву
            node = tree[node].right

    return result
```

```python
def pre_order(tree, root):
    """Прямой обход дерева (pre_order)"""
    if root == -1:
        return []

    stack = [root] # Начинаем с корня
    result = []

    while stack:
        node = stack.pop()
        result.append(tree[node].key)

        # Сначала добавляем правого потомка, чтобы левый обработался первым
        if tree[node].right != -1:
            stack.append(tree[node].right)
        if tree[node].left != -1:
            stack.append(tree[node].left)

    return result


def post_order(tree, root):
    """Обратный обход дерева (post_order)"""
    if root == -1:
        return []

    stack, result = [], []
    last_visited = None # Последний посещенный узел
    current = root # Текущий узел

    while stack or current != -1:
        if current != -1:
            # Идем до самого левого узла
            stack.append(current)
            current = tree[current].left
        else:
            # Смотрим верхний узел в стеке
            peek_node = stack[-1]
            # Если есть правое поддерево и мы его еще не посещали, переходим к нему
            if tree[peek_node].right != -1 and last_visited != tree[peek_node].right:
```

```
            current = tree[peek_node].right
        else:
            # Иначе обрабатываем текущий узел
            result.append(tree[peek_node].key)
            last_visited = stack.pop()

    return result


@measure_time_and_memory
def do_traversals(n, nodes):
    tree = parse_tree(n, nodes)

    in_order_res = in_order(tree, 0)
    pre_order_res = pre_order(tree, 0)
    post_order_res = post_order(tree, 0)

    return [in_order_res, pre_order_res, post_order_res]


def main():
    # Чтение входных данных из файла
    input_data = read_from_file(input_path)

    # Извлечение параметров из входных данных
    n = input_data[0]
    nodes = input_data[1:]

    # Вычисление результата
    result = do_traversals(n, nodes)

    # Запись результата в выходной файл
    output_data = "\n".join(str(x) for x in result)
    write_to_file(output_path, output_data)


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

Объяснение:

1. In-order обход:

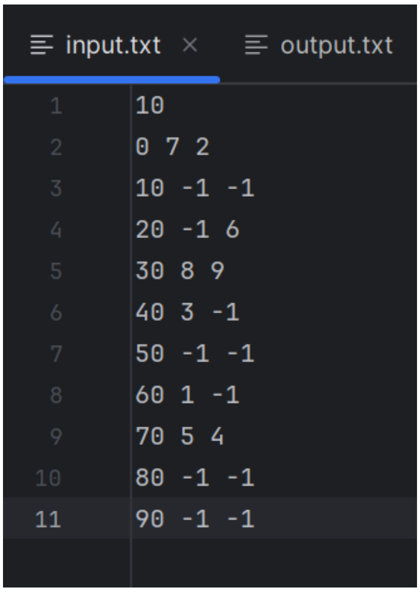
- Идем до самого левого узла, добавляя все узлы в стек
- Извлекаем узел из стека, добавляем его значение в результат
- Переходим к правому поддереву

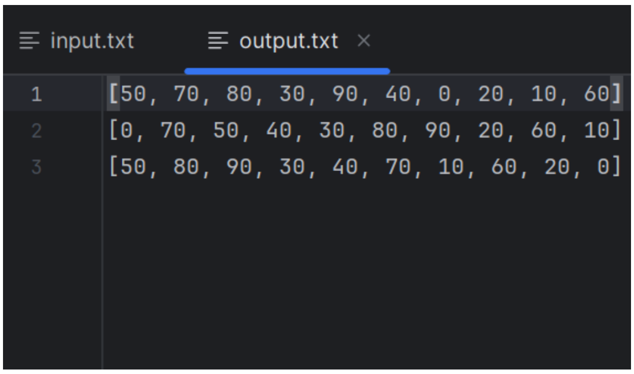
2. Pre-order обход:
    - Начинаем с корня, сразу добавляем его в результат
    - Сначала кладем в стек правого потомка, затем левого (чтобы левый обрабатывался первым)
3. Post-order обход:
    - Используем переменную **last_visited** для отслеживания уже обработанных узлов
    - Идем до самого левого узла
    - Проверяем, есть ли правое поддерево и не было ли оно посещено
    - Если правое поддерево обработано или отсутствует, добавляем текущий узел в результат

Сложность:

- O(n) для каждого обхода, итого O(3n) = O(n)

Результат работы кода:





Задание №3. Простейшее BST (1 балл)

В этой задаче вам нужно написать простейшее BST по явному ключу и отвечать им на запросы: «+ x» – добавить в дерево x (если x уже есть, ничего не делать). «> x» – вернуть минимальный элемент больше x или 0, если таких нет.

- 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. $1 \leq x \leq 109$, $1 \leq N \leq 300000$
- Формат вывода / выходного файла (output.txt). Для каждого запроса вида «> x» выведите в отдельной строке ответ.

Решение:

```
from utils.file_utlis import read_from_file, write_to_file
from utils.time_memory_utlis import measure_time_and_memory


input_path = "../txtf/input.txt"
output_path = "../txtf/output.txt"


class Node:
    """Класс для представления узла дерева"""
    def __init__(self, key):
        self.key = key
        self.right = None
        self.left = None


class BinarySearchTree:
    """Класс бинарного дерева"""
    def __init__(self):
        self.root = None
```

```python
    def insert(self, key):
        if not self.root:
            self.root = Node(key)
            return

        cur_node = self.root # Начинаем с корня

        while True:
            if key < cur_node.key: # Идем в левое поддерево
                if cur_node.left:
                    cur_node = cur_node.left
                else:
                    cur_node.left = Node(key) # Создаем новый узел
                    return
            elif key > cur_node.key: # Идем в правое поддерево
                if cur_node.right:
                    cur_node = cur_node.right
                else:
                    cur_node.right = Node(key) # Создаем новый узел
                    return
            else: # Ключ уже существует в дереве
                return

    def find_min(self, key):
        """Поиск минимального элемента"""
        node = self.root
        res = None

        while node:
            if node.key > key:
                res = node
                node = node.left
            else:
                node = node.right
        return res.key if res else 0



@measure_time_and_memory
def do_oparations(operations):
    """Обработка всех операций с деревом"""
    bst = BinarySearchTree()
    min_keys = []
```

```
    for oper in operations:
        to_do = oper[0]
        key = oper[1]

        if to_do == "+":
            bst.insert(key)
        else:
            min_key = bst.find_min(key)
            min_keys.append(min_key)
    return min_keys


def main():
    # Чтение входных данных из файла
    input_data = read_from_file(input_path)

    # Извлечение параметров из входных данных
    operations = input_data

    # Вычисление результата
    result = do_oparations(operations)

    # Запись результата в выходной файл
    output_data = "\n".join(str(x) for x in result)
    write_to_file(output_path, output_data)


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

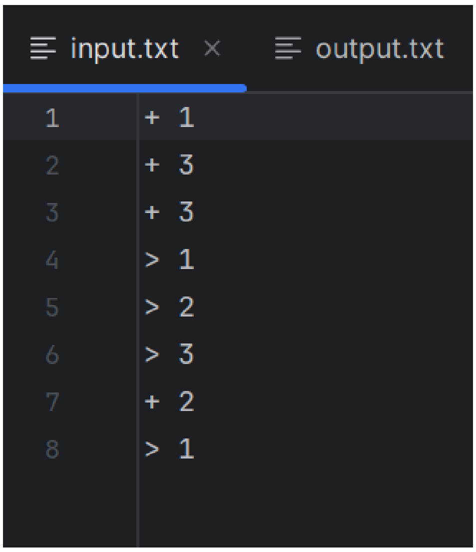
Объяснение:

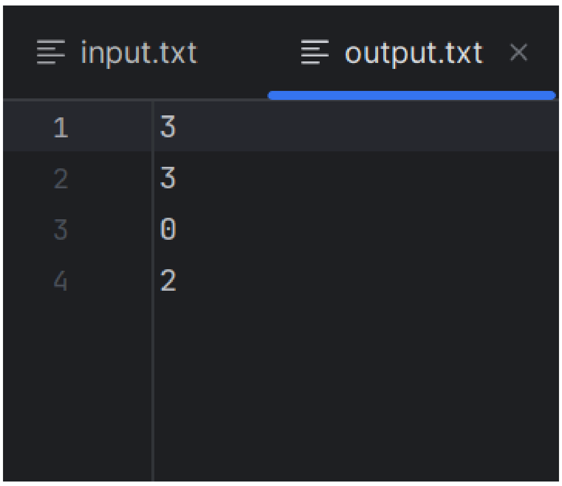
1. Вставка элемента:
    - Если дерево пустое, создаем корневой узел
    - Иначе идем от корня:
        - Если ключ меньше текущего узла - идем влево
        - Если ключ больше - идем вправо
        - Если нашли такой же ключ - ничего не делаем (дубликаты не добавляем)

        ■ Дойдя до пустого места (None), создаем новый узел
2. Поиск минимального элемента больше x:
    - Идем от корня, сохраняя "кандидата" (наименьший найденный элемент больше x)
    - Если текущий узел > x - он становится новым кандидатом, идем влево (ищем меньший, но все еще > x)
    - Иначе идем вправо (ищем большие значения)
    - Возвращаем кандидата или 0, если не нашли

Сложность:
1. Вставка элемента:
    - Средний случай: O(log n) - при случайных данных (равномерное распределение)
    - Худший случай: O(n) - при вырожденном дереве (например, элементы добавляются по порядку)
2. Поиск минимального больше x:
    - Средний случай: O(log n)
    - Худший случай: O(n)

Результат работы кода:





Задание №8. Высота дерева возвращается (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.

Дано двоичное дерево поиска. В вершинах этого дерева записаны ключи – целые числа, по модулю не превышающие 109. 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 ≤ i ≤ 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 ≤ N или Li = 0, если левого ребенка нет) и номера правого Ri ребенка i-ой вершины (i < Ri ≤ N или Ri = 0, если правого ребенка нет).
- Ограничения на входные данные. 0 ≤ N ≤ 2·105, |Ki| ≤ 109. Все ключи различны. Гарантируется, что данное дерево является деревом поиска.
- Формат вывода / выходного файла (output.txt). Выведите одно целое число – высоту дерева.

Решение:

```
from utils.file_utlis import read_from_file, write_to_file
from utils.time_memory_utlis import measure_time_and_memory


input_path = "../txtf/input.txt"
output_path = "../txtf/output.txt"
```

```python
class Node:
  """Класс для представления узла дерева"""
  def __init__(self, key, left, right):
      self.key = key
      self.right = right
      self.left = left


def parse_tree(n, nodes):
  """Функция для парсинга дерева"""
  tree = {0: None}
  for i in range(1, n + 1):
      k, l, r = nodes[i - 1]
      tree[i] = Node(k, l, r)
  return tree


@measure_time_and_memory
def tree_height(n, nodes):
  if n == 0: return 0

  tree = parse_tree(n, nodes)
  stack = [(1, 1)] # (номер узла, текущая глубина)
  max_height = 1

  while len(stack) != 0:
      i, cur_height = stack.pop()
      node = tree[i] # Получаем узел

      # Обновляем максимальную высоту
      if cur_height > max_height:
          max_height = cur_height

        # Добавляем потомков в стек с увеличенной глубиной (сначала правый, чтобы левый обрабатывался первым)
      if node.right != 0:
          stack.append((node.right, cur_height + 1))
      if node.left != 0:
          stack.append((node.left, cur_height + 1))
```

```
    return max_height


def main():
    # Чтение входных данных из файла
    input_data = read_from_file(input_path)

    # Извлечение параметров из входных данных
    n = input_data[0]
    nodes = input_data[1:]

    # Вычисление результата
    result = tree_height(n, nodes)

    # Запись результата в выходной файл
    output_data = str(result)
    write_to_file(output_path, output_data)


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

Объяснение:

1. Представление дерева:
   - Класс **Node** хранит ключ узла, индексы левого и правого потомков
   - Функция **parse_tree** преобразует входные данные в словарь узлов
2. Алгоритм вычисления высоты:
   - Используется итеративный обход в глубину (DFS) с помощью стека
   - В стеке хранятся пары (номер узла, текущая глубина)
   - Начинаем с корня (узел 1) с глубиной 1
   - Для каждого узла:
     - Обновляем максимальную найденную высоту
     - Добавляем в стек потомков с увеличенной на 1 глубиной

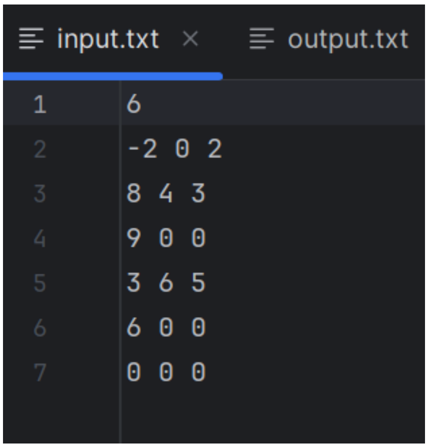
- Правый потомок добавляется первым, чтобы левый обрабатывался раньше (но это не влияет на результат)

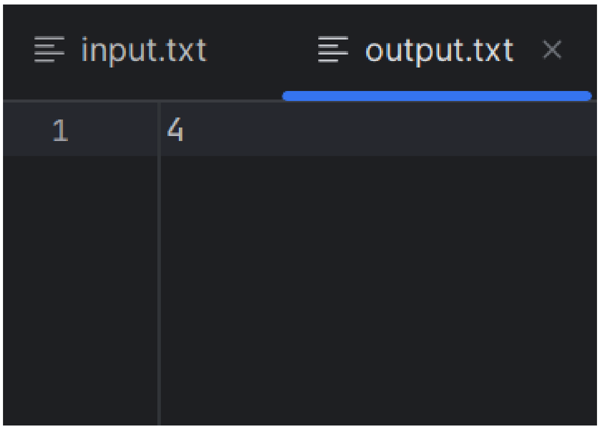
3. Обработка краевых случаев:
   - Если дерево пустое (N=0), возвращаем 0
   - Если у узла нет потомков (лист), его глубина учитывается в максимальной высоте

Сложность:

- O(N) - где N количество узлов в дереве

Результат работы кода:





Вывод

Все три задачи показывают, что эффективная работа с деревьями строится на понимании их структуры и правильном выборе алгоритмов обхода. Итеративные решения часто предпочтительнее рекурсивных из-за ограничений на глубину рекурсии. Для BST критично сохранять свойства дерева поиска, иначе операции перестают быть эффективными.