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 №4

Выполнил:

Волжева М.И.

К3141

Проверила:

Артамонова В.Е.

Санкт-Петербург

2023 г.

# Содержание отчета

[Содержание отчета 2](#_Toc137669122)

[Задачи по варианту 3](#_Toc137669123)

[Задача №4. Простейший неявный ключ [2 s, 256 Mb, 1 балл] 3](#_Toc137669124)

[Задача №8. Высота дерева возвращается [2 s, 256 Mb, 2 балла] 5](#_Toc137669125)

[Задача №15. Удаление из АВЛ-дерева [2 s, 256 Mb, 3 балла] 9](#_Toc137669126)

[Задачи по выбору 15](#_Toc137669127)

[Задача №1. Обход двоичного дерева [5 s, 512 Mb, 1 балл] 15](#_Toc137669128)

[Задача № 2. Гирлянда [2 s, 256 Mb, 1 балл] 18](#_Toc137669129)

[Задача №5. Простое двоичное дерево поиска [2 s, 512 Mb, 1 балл] 20](#_Toc137669130)

[Задача №16. K-й максимум [2 s, 512 Mb, 3 балла] 24](#_Toc137669131)

[Задача №12. Проверка сбалансированности [2 s, 256 Mb, 2 балла] – НЕ РАБОТАЕТ!!! 31](#_Toc137669132)

[Вывод 32](#_Toc137669133)

# Задачи по варианту

## Задача №4. Простейший неявный ключ [2 s, 256 Mb, 1 балл]

* Текст задачи:  
  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 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.
* Ограничения на входные данные.   
  1 ≤ x ≤ 10^9, 1 ≤ N ≤ 300000, в запросах «? k», число k от 1 до количества элементов в дереве.
* Формат выходного файла (output.txt).   
  Для каждого запроса вида «? k» выведите в отдельной строке ответ.
* Ограничение по времени. 2сек.
* Ограничение по памяти. 256 мб.
* Пример:

|  |  |
| --- | --- |
| Input.txt | + 1  + 4  + 3  + 3  ? 1  ? 2  ? 3  + 2  ? 3 |
| Output.txt | 1  3  4  3 |

Листинг кода:

import time  
import os, psutil  
  
  
class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.left = self.right = None  
  
  
class Tree:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def \_\_find(self, node, parent, value):  
 if node is None:  
 return None, parent, False  
  
 if value == node.data:  
 return node, parent, True  
  
 if value < node.data:  
 if node.left:  
 return self.\_\_find(node.left, node, value)  
  
 if value > node.data:  
 if node.right:  
 return self.\_\_find(node.right, node, value)  
  
 return node, parent, False  
  
 def append(self, obj):  
 if self.root is None:  
 self.root = obj  
 return obj  
  
 s, p, fl\_find = self.\_\_find(self.root, None, obj.data)  
  
 if not fl\_find and s:  
 if obj.data < s.data:  
 s.left = obj  
 else:  
 s.right = obj  
  
 return obj  
  
 def find\_k\_element(self, node, result, k, flag, file):  
 if len(result) == k:  
 flag = True  
 file.write(str(result[k - 1]) + "\n")  
 if node is None:  
 return  
 if not flag:  
 self.find\_k\_element(node.left, result, k, flag, file)  
 result.append(node.data)  
 self.find\_k\_element(node.right, result, k, flag, file)  
  
  
t\_start = time.perf\_counter()  
process = psutil.Process(os.getpid())  
f = open("input.txt")  
m = open("output.txt", "w")  
t = Tree()  
for each in f.readlines():  
 operation, number = each.split()  
 if operation == "+":  
 t.append(Node(int(number)))  
 if operation == "?":  
 t.find\_k\_element(t.root, [], int(number), False, m)  
  
f.close()  
m.close()  
print("Time of working: %s second" % (time.perf\_counter() - t\_start))  
print("Memory", process.memory\_info().rss/(1024\*1024), "mb")

Текстовое объяснение решения:

Мы реализовали бинарное дерево, как класс. Поиск k-ого по возрастанию элемента осуществляется поиском в глубину, который в определенный момент останавливается.

Результат работы кода на примерах из текста задачи:

|  |
| --- |
|  |

Тестирование алгоритма:

|  |  |  |
| --- | --- | --- |
|  | Время выполнения (seconds) | Затраты памяти (mbs) |
| Пример | 0.0008377999999999997 | 14.1953125 |

Вывод по задаче: Мы научились реализовывать бинарное дерево, как класс и можем искать k-по возрастанию элемент.

## Задача №8. Высота дерева возвращается [2 s, 256 Mb, 2 балла]

* Текст задачи:  
  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. 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 ≤ 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 iой 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 выходного файла (output.txt).   
  Выведите одно целое число – высоту дерева.
* Пример:

|  |  |
| --- | --- |
| Input.txt | 6  -2 0 2  8 4 3  9 0 0  3 6 5  6 0 0  0 0 0 |
| Output.txt | 4 |

Листинг кода:

import time  
import os, psutil  
  
  
class Node:  
  
 def \_\_init\_\_(self):  
 self.key\_t = None  
 self.key = None  
 self.left = None  
 self.right = None  
 self.parent = None  
 self.height = 0  
  
  
class BinTree:  
  
 def \_\_init\_\_(self):  
 self.root = None  
 self.nodes = {}  
  
 def set\_height(self, node):  
 node.height = 1  
 while node.parent:  
 parent = node.parent  
 if parent.height <= node.height:  
 parent.height = node.height + 1  
 node = parent  
  
  
t\_start = time.perf\_counter()  
process = psutil.Process(os.getpid())  
  
with open('input.txt') as f:  
 n = int(f.readline())  
 if n == 0:  
 with open('output.txt', 'w') as f:  
 f.write('0')  
 exit()  
 tree = BinTree()  
 data = []  
 leaves = []  
 nodes = {}  
 for i in range(1, n+1):  
 data.append(list(map(int, f.readline().split())))  
 tree.nodes[i] = Node()  
 tree.nodes[i].key = data[i-1][0]  
 if data[i-1][1] == 0 and data[i-1][2] == 0:  
 leaves.append(i)  
  
for i in range(1, n+1):  
 if data[i-1][1] != 0:  
 tree.nodes[i].left = tree.nodes[data[i-1][1]]  
 tree.nodes[data[i-1][1]].parent = tree.nodes[i]  
 if data[i - 1][2] != 0:  
 tree.nodes[i].right = tree.nodes[data[i - 1][2]]  
 tree.nodes[data[i - 1][2]].parent = tree.nodes[i]  
 if i == 1:  
 tree.root = tree.nodes[i]  
  
for i in leaves:  
 tree.set\_height(tree.nodes[i])  
with open('output.txt', 'w') as f:  
 f.write(str(tree.root.height))  
  
  
print("Time of working: %s second" % (time.perf\_counter() - t\_start))  
print("Memory", process.memory\_info().rss/(1024\*1024), "mb")

Текстовое объяснение решения:

Мы создали два класса – лист и дерево. Сначала мы читаем данные из файла, а далее для каждого листа применяем функцию set\_height. Ответом будет является значение этой функции от корня. Немного про функцию (. Высота начального корня = 1 Пока дальше есть родитель, он становится корнем. Если высота родителя меньше или равна высоты корня, то мы увеличиваем высоту корня. Иначе корень становится родителем.)

Результат работы кода на примерах из текста задачи:

|  |
| --- |
|  |

Тестирование алгоритма:

|  |  |  |
| --- | --- | --- |
|  | Время выполнения (seconds) | Затраты памяти (mbs) |
| Пример | 0.0012832999999999872 | 14. 23828125 |

Вывод по задаче: Мы создали 2 класса и нашли высоту дерева.

## Задача №15. Удаление из АВЛ-дерева [2 s, 256 Mb, 3 балла]

* Текст задачи:  
  Удаление из АВЛ-дерева вершины с ключом X, при условии ее наличия, осуществляется следующим образом:

• путем спуска от корня и проверки ключей находится V – удаляемая вершина;

• если вершина V – лист (то есть, у нее нет детей):

– удаляем вершину;

– поднимаемся к корню, начиная с бывшего родителя вершины V , при этом если встречается несбалансированная вершина, то производим поворот.

• если у вершины V не существует левого ребенка:

– следовательно, баланс вершины равен единице и ее правый ребенок – лист;

– заменяем вершину V ее правым ребенком;

– поднимаемся к корню, производя, где необходимо, балансировку.

• иначе:

– находим R – самую правую вершину в левом поддереве;

– переносим ключ вершины R в вершину V ;

– удаляем вершину R (у нее нет правого ребенка, поэтому она либо лист, либо имеет левого ребенка, являющегося листом);

– поднимаемся к корню, начиная с бывшего родителя вершины R, производя балансировку.

Исключением является случай, когда производится удаление из дерева, состоящего из одной вершины - корня. Результатом удаления в этом случае будет пустое дерево. Указанный алгоритм не является единственно возможным, но мы просим Вас реализовать именно его, так как тестирующая система проверяет точное равенство получающихся деревьев.

* Формат входного файла (input.txt).  
  Входной файл содержит описание двоичного дерева, а также ключа вершины, которую требуется удалить из 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). Все ключи различны. Гарантируется, что данное дерево является деревом поиска. В последней строке содержится число X – ключ вершины, которую требуется удалить из дерева. Гарантируется, что такая вершина в дереве существует.
* Ограничения на входные данные.   
  1 ≤ N ≤ 2 · 105, |Ki| ≤ 109, |X| ≤ 109
* Формат выходного файла (output.txt).   
  Выведите в том же формате дерево после осуществления операции удаления. Нумерация вершин может быть произвольной при условии соблюдения формата.
* Пример:

|  |  |
| --- | --- |
| Input.txt | 3  4 2 3  3 0 0  5 0 0  4 |
| Output.txt | 2  3 0 2  5 0 0 |

Листинг кода:

import time  
import os, psutil  
import sys  
from collections import deque  
  
  
res = []  
class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.par = None  
 self.left = None  
 self.right = None  
 self.height = -1  
 self.id = 0  
 self.next = None  
  
  
def dfs(root):  
 if root.left is not None:  
 dfs(root.left)  
 if root.right is not None:  
 dfs(root.right)  
 fix\_height(root)  
  
  
def height\_right(root):  
 if root.right is None:  
 return 0  
 return root.right.height  
  
  
def height\_left(root):  
 if root.left is None:  
 return 0  
 return root.left.height  
  
  
def fix\_height(root):  
 root.height = max(height\_left(root), height\_right(root)) + 1  
  
  
def blc(root):  
 r = 0  
 l = 0  
 if root.right is not None:  
 r = root.right.height  
 if root.left is not None:  
 l = root.left.height  
 return r - l  
  
  
def Rotate(node, side):  
 if side == 'left':  
 if node is None or node.right is None:  
 return node  
 parent = node.par  
 right = node.right  
 right\_left = right.left  
 if parent:  
 if parent.right == node:  
 parent.right = right  
 else:  
 parent.left = right  
 right.par = parent  
 right.left = node  
 node.par = right  
 node.right = right\_left  
 if right\_left:  
 right\_left.par = node  
 fix\_height(node)  
 fix\_height(right)  
 return right  
 else:  
 if node is None or node.left is None:  
 return node  
 parent = node.par  
 left = node.left  
 left\_right = left.right  
 if parent:  
 if parent.left == node:  
 parent.left = left  
 else:  
 parent.right = left  
 left.par = parent  
 left.right = node  
 node.par = left  
 node.left = left\_right  
 if left\_right:  
 left\_right.par = node  
 fix\_height(node)  
 fix\_height(left)  
 return left  
  
  
def getMax(root):  
 if root is None:  
 return root  
 while root.right is not None:  
 root = root.right  
 return root  
  
  
def Balance(root):  
 fix\_height(root)  
 balance = blc(root)  
 if balance > 1:  
 if blc(root.right) < 0:  
 root.right = Rotate(root.right, 'right')  
 return Rotate(root, 'left')  
 elif balance < -1:  
 if blc(root.left) > 0:  
 root.left = Rotate(root.left, 'left')  
 return Rotate(root, 'right')  
 return root  
  
  
def delete(root, key):  
 if root is None:  
 return root  
 elif key < root.data:  
 root.left = delete(root.left, key)  
 elif key > root.data:  
 root.right = delete(root.right, key)  
 else:  
 if root.left is None and root.right is None:  
 return None  
 if root.left is None:  
 root = root.right  
 return Balance(root)  
 temp = getMax(root.left)  
 root.data = temp.data  
 root.left = delete(root.left, temp.data)  
 return Balance(root)  
  
  
def printBST(root, n):  
 global res  
 queue = deque()  
 queue.append((root, (-1, -1)))  
 while queue:  
 u, v = queue.popleft()  
 if v[0] >= 0 and v[1] >= 0:  
 res[v[0]][v[1]] = len(res) + 1  
 if u is None:  
 continue  
 tmp = [0, 0, 0]  
 tmp[0] = u.data  
 res.append(tmp)  
 cur = len(res)  
 if u.left is not None:  
 queue.append((u.left, (cur - 1, 1)))  
 if u.right is not None:  
 queue.append((u.right, (cur - 1, 2)))  
  
  
t\_start = time.perf\_counter()  
process = psutil.Process(os.getpid())  
sys.stdin = open("input.txt", "r")  
sys.stdout = open("output.txt", "w")  
n = int(sys.stdin.readline())  
nodes = []  
for i in range(n + 10):  
 nodes.append(Node(0))  
for i in range(n):  
 k, l, r = map(int, sys.stdin.readline().split())  
 nodes[i + 1].data = k  
 if l:  
 nodes[i + 1].left = nodes[l]  
 nodes[l].par = nodes[i + 1]  
 if r:  
 nodes[i + 1].right = nodes[r]  
 nodes[r].par = nodes[i + 1]  
  
val = int(sys.stdin.readline())  
dfs(nodes[1])  
nodes[1] = delete(nodes[1], val)  
printBST(nodes[1], n)  
sys.stdout.write(str(len(res)) + "\n")  
n = len(res)  
for i, j, k in res:  
 sys.stdout.write(str(i) + ' ' + str(j) + ' ' + str(k) + '\n')  
print("Time of working: %s second" % (time.perf\_counter() - t\_start))  
print("Memory", process.memory\_info().rss/(1024\*1024), "mb")

Текстовое объяснение решения:

Основной принцип решения включает:

1. Определение класса Node, представляющего узел дерева, с полями для данных, ссылок на родителя и потомков, а также для высоты узла.

2. Реализацию функций для обхода и балансировки дерева, а также для удаления узла.

3. Считывание данных из файла ввода, создание и связывание узлов дерева на основе считанных значений.

4. Удаление узла с заданным значением из дерева.

5. Обход дерева в ширину для сохранения результата в переменную.

6. Запись результата в выходной файл.

7. Вывод информации о времени выполнения программы и доступной памяти.

Результат работы кода на примерах из текста задачи:

|  |
| --- |
|  |

Тестирование алгоритма:

|  |  |  |
| --- | --- | --- |
|  | Время выполнения (seconds) | Затраты памяти (mbs) |
| Пример | 0.0008377999999999997 | 14.1953125 |

Вывод по задаче: В этой задаче я научилась удалять из АВЛ-дерева вершины с ключом X

# Задачи по выбору

## Задача №1. Обход двоичного дерева [5 s, 512 Mb, 1 балл]

* 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, ..., 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 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).
* Пример:

|  |  |  |
| --- | --- | --- |
| Input.txt | 5  4 1 2  2 3 4  5 -1 -1  1 -1 -1  3 -1 -1 |  |
| Output.txt | 1 2 3 4 5  4 2 1 3 5  1 3 2 5 4 |  |

Листинг кода:

import time  
import os, psutil  
  
  
class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.left = self.right = None  
  
  
class Tree:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def \_\_find(self, node, parent, value):  
 if node is None:  
 return None, parent, False  
  
 if value == node.data:  
 return node, parent, True  
  
 if value < node.data:  
 if node.left:  
 return self.\_\_find(node.left, node, value)  
  
 if value > node.data:  
 if node.right:  
 return self.\_\_find(node.right, node, value)  
  
 return node, parent, False  
  
 def append(self, obj):  
 if self.root is None:  
 self.root = obj  
 return obj  
  
 s, p, fl\_find = self.\_\_find(self.root, None, obj.data)  
  
 if not fl\_find and s:  
 if obj.data < s.data:  
 s.left = obj  
 else:  
 s.right = obj  
  
 return obj  
  
 def find\_k\_element(self, node, result, k, flag, file):  
 if len(result) == k:  
 flag = True  
 file.write(str(result[k - 1]) + "\n")  
 if node is None:  
 return  
 if not flag:  
 self.find\_k\_element(node.left, result, k, flag, file)  
 result.append(node.data)  
 self.find\_k\_element(node.right, result, k, flag, file)  
  
  
t\_start = time.perf\_counter()  
process = psutil.Process(os.getpid())  
f = open("input.txt")  
m = open("output.txt", "w")  
t = Tree()  
for each in f.readlines():  
 operation, number = each.split()  
 if operation == "+":  
 t.append(Node(int(number)))  
 if operation == "?":  
 t.find\_k\_element(t.root, [], int(number), False, m)  
  
f.close()  
m.close()  
print("Time of working: %s second" % (time.perf\_counter() - t\_start))  
print("Memory", process.memory\_info().rss/(1024\*1024), "mb")

Текстовое объяснение решения:

Мы реализовали бинарное дерево, как класс. Поиск k-ого по возрастанию элемента осуществляется поиском в глубину, который в определенный момент останавливается.

Результат работы кода на примерах из текста задачи:

|  |
| --- |
|  |

Тестирование алгоритма:

|  |  |  |
| --- | --- | --- |
|  | Время выполнения (seconds) | Затраты памяти (mbs) |
| Пример | 0.0008377999999999997 | 14.1953125 |

Вывод по задаче: Мы научились реализовывать бинарное дерево, как класс и можем искать k-по возрастанию элемент.

## Задача № 2. Гирлянда [2 s, 256 Mb, 1 балл]

* Текст задачи:  
  Гирлянда состоит из n лампочек на общем проводе. Один её конец закреплён на заданной высоте A мм (h1 = A). Благодаря силе тяжести гирлянда прогибается: высота каждой неконцевой лампы на 1 мм меньше, чем средняя высота ближайших соседей (hi = (hi−1 + hi+1)/2 − 1 для 1 < i < N). Требуется найти минимальное значение высоты второго конца B (B = hn), такое что для любого ϵ > 0 при высоте второго конца B + ϵ для всех лампочек выполняется условие hi > 0. Обратите внимание на то, что при данном значении высоты либо ровно одна, либо две соседних лампочки будут иметь нулевую высоту. Подсказка: для решения этой задачи можно использовать двоичный поиск
* Формат ввода.  
  В первой строке входного файла содержится два числа n и A.
* Ограничения на входные данные.   
  3 ≤ n ≤ 1000, n – целое, 10 ≤ A ≤ 1000, A – вещественное и дано не более чем с тремя знаками после десятичной точки.
* Формат выходного файла (output.txt).   
  Выведите одно вещественное число B – минимальную высоту второго конца. Ваш ответ будет засчитан, если он будет отличаться от правильного не более, чем на 10−6
* Пример:

|  |  |  |
| --- | --- | --- |
| Input.txt | 8 15 | 692 532.81 |
| Output.txt | 9.75 | 446113.34434782615 |

Листинг кода:

import time  
import os, psutil  
  
  
def high(h, n):  
 left = 0  
 right = h[0]  
 while (right - left > 0.0000000001):  
 h[1] = (left + right) / 2  
 Up = True  
 for i in range(2,n):  
 h[i] = 2 \* h[i - 1] - h[i - 2] + 2  
 if h[i] < 0:  
 Up = False  
 break  
 if Up:  
 right = h[1]  
 else:  
 left = h[1]  
 return h[n - 1]  
  
  
t\_start = time.perf\_counter()  
process = psutil.Process(os.getpid())  
  
with open('input.txt') as m:  
 a = list(map(float, m.readline().split()))  
 n = int(a[0])  
 A = a[1]  
  
h = []  
for i in range(n):  
 h.append(0)  
h[0]=float(A)  
res = high(h, n)  
  
f = open('output.txt', 'w')  
f.write(str(res))  
  
f.close()  
m.close()  
print("Time of working: %s second" % (time.perf\_counter() - t\_start))  
print("Memory", process.memory\_info().rss/(1024\*1024), "mb")

Текстовое объяснение решения:

Одушевлять перебор по всем значениям будет не очень эффективно. Поэтому мы ищем среднее между левым и правым, и дальше отталкиваясь от этого решаем в которую сторону нужно двигаться и решаем дальше.

Результат работы кода на примерах из текста задачи:

|  |  |
| --- | --- |
|  |  |

Тестирование алгоритма:

|  |  |  |
| --- | --- | --- |
|  | Время выполнения (seconds) | Затраты памяти (mbs) |
| Пример | 0.017354399999999992 | 14.1953125 |
| Пример | 0.004493500000000011 | 14.08203125 |

Вывод по задаче: Мы научились оптимизировать перебор. Мы нашли минимальное значение второго конца гирлянды.

## Задача №5. Простое двоичное дерево поиска [2 s, 512 Mb, 1 балл]

* Текст задачи:  
  Реализуйте простое двоичное дерево поиска
* Формат ввода / входного файла (input.txt). Входной файл содержит описание операций с деревом, их количество N не превышает 100. 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 109.

* Ограничения на входные данные. 0 ≤ N ≤ 100, |x| ≤ 109.
* Формат выходного файла (output.txt).   
  Выведите последовательно результат выполнения всех операций exists, next, prev. Следуйте формату выходного файла из примера. Пример:

|  |  |
| --- | --- |
| Input.txt | insert 2  insert 5  insert 3  exists 2  exists 4  next 4  prev 4  delete 5  next 4  prev 4 |
| Output.txt | true  false  5  3  none  3 |

Листинг кода:

import time  
import os, psutil  
  
t\_start = time.perf\_counter()  
process = psutil.Process(os.getpid())  
  
class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.left = None  
 self.right = None  
  
  
class BinarySearchTree:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def insert(self, value):  
 if self.root is None:  
 self.root = Node(value)  
 else:  
 self.\_insert(self.root, value)  
  
 def \_insert(self, node, value):  
 if value < node.value:  
 if node.left is None:  
 node.left = Node(value)  
 else:  
 self.\_insert(node.left, value)  
 elif value > node.value:  
 if node.right is None:  
 node.right = Node(value)  
 else:  
 self.\_insert(node.right, value)  
  
 def delete(self, value):  
 self.root = self.\_delete(self.root, value)  
  
 def \_delete(self, node, value):  
 if node is None:  
 return None  
 if value < node.value:  
 node.left = self.\_delete(node.left, value)  
 elif value > node.value:  
 node.right = self.\_delete(node.right, value)  
 else:  
 if node.left is None:  
 return node.right  
 elif node.right is None:  
 return node.left  
 else:  
 min\_node = self.\_find\_min(node.right)  
 node.value = min\_node.value  
 node.right = self.\_delete(node.right, min\_node.value)  
 return node  
  
 def exists(self, value):  
 return self.\_exists(self.root, value)  
  
 def \_exists(self, node, value):  
 if node is None:  
 return False  
 if node.value == value:  
 return True  
 elif value < node.value:  
 return self.\_exists(node.left, value)  
 else:  
 return self.\_exists(node.right, value)  
  
 def next(self, value):  
 node = self.root  
 result = None  
 while node is not None:  
 if node.value > value:  
 if result is None or node.value < result.value:  
 result = node  
 node = node.left  
 else:  
 node = node.right  
 if result is None:  
 return "none"  
 else:  
 return str(result.value)  
  
 def prev(self, value):  
 node = self.root  
 result = None  
 while node is not None:  
 if node.value < value:  
 if result is None or node.value > result.value:  
 result = node  
 node = node.right  
 else:  
 node = node.left  
 if result is None:  
 return "none"  
 else:  
 return str(result.value)  
  
  
with open("input.txt") as f\_in, open("output.txt", "w") as f\_out:  
 tree = BinarySearchTree()  
 for line in f\_in:  
 operation, value = line.strip().split()  
 value = int(value)  
 if operation == "insert":  
 tree.insert(value)  
 elif operation == "delete":  
 tree.delete(value)  
 elif operation == "exists":  
 f\_out.write(str(tree.exists(value)).lower() + "\n")  
 elif operation == "next":  
 f\_out.write(tree.next(value) + "\n")  
 elif operation == "prev":  
 f\_out.write(tree.prev(value) + "\n")  
  
print("Time of working: %s second" % (time.perf\_counter() - t\_start))  
print("Memory", process.memory\_info().rss/(1024\*1024), "mb")

Текстовое объяснение решения:

Для реализации двоичного дерева поиска были определены два класса: Node и BinarySearchTree. Класс Node представляет узел дерева и содержит ссылки на левое и правое поддеревья, а также на значение, хранящееся в узле. Класс BinarySearchTree содержит ссылку на корень дерева и определенные методы для выполнения операций над деревом. Входные данные читаются из файла, результаты выводятся в выходной файл. Общая идея заключается в использовании базовых операций над двоичным деревом для выполнения заданных операций.

Результат работы кода на примерах из текста задачи:

|  |
| --- |
|  |

Тестирование алгоритма:

|  |  |  |
| --- | --- | --- |
|  | Время выполнения (seconds) | Затраты памяти (mbs) |
| Пример | 0. 0008553999999999923 | 14. 23828125 |

Вывод по задаче: В этой задаче я реализовала простое двоичное дерево поиска

## Задача №16. K-й максимум [2 s, 512 Mb, 3 балла]

* 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 ci и ki – тип и аргумент команды соответственно. Поддерживаемые команды:

– +1 (или просто 1): Добавить элемент с ключом ki .

– 0 : Найти и вывести ki-й максимум.

– -1 : Удалить элемент с ключом ki .

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

* Ограничения на входные данные. N ≤ 10^5, |Ki| ≤ 10^9.
* Формат выходного файла (output.txt).   
  Для каждой команды нулевого типа в выходной файл должна быть выведена строка, содержащая единственное число – ki-й максимум.
* Пример:

|  |  |
| --- | --- |
| Input.txt | 11  +1 5  +1 3  +1 7  0 1  0 2  0 3  -1 5  +1 10  0 1  0 2  0 3 |
| Output.txt | 7  5  3  10  7  3 |

Листинг кода:

import time  
import psutil  
import sys  
  
t\_start = time.perf\_counter()  
  
  
class Node:  
 def \_\_init\_\_(self, v=int()): #значение (value  
 self.value = v  
 self.left = None #ссылки на левого и правого потомков (left и right)  
 self.right = None  
 self.count\_left = 0 #количество узлов в поддеревьях слева и справа  
 self.count\_right = 0  
 self.height\_left = 0 #высоту левого и правого поддеревьев  
 self.height\_right = 0  
 self.parent = None #ссылка на родительский узел  
  
  
class AVLTree:  
 root = None  
  
 def \_\_init\_\_(self): #дерево в данный момент не содержит ни одного узла и является пустым  
 self.root = None  
  
 def insert(self, value): #вставляет новый узел со значением value в дерево, сохраняя его самобалансировку.  
 if not self.find(value):  
 time = Node(value)  
 if self.root is None:  
 self.root = time  
 else:  
 index = self.root  
 flag = True  
 while flag:  
 if time.value > index.value and index.right is not None:  
 index = index.right  
 elif time.value < index.value and index.left is not None:  
 index = index.left  
 else:  
 flag = False  
 if time.value > index.value:  
 time.parent = index  
 index.right = time  
 else:  
 time.parent = index  
 index.left = time  
 self.balance\_number(index)  
 while index is not None:  
 self.balance\_number(index)  
 index = self.balance(index)  
 self.root = index  
 index = index.parent  
 return time  
  
 def delete(self, value): #удаляет узел с заданным значением value из дерева, сохраняя его самобалансировку.  
 self.delete\_vertex(self.find(value))  
  
 def delete\_vertex(self, time): #удаление узла time из АВЛ-дерева.  
 if time:  
 if max(time.height\_right, time.height\_left) == 0:  
 if time.parent is None:  
 self.root = None  
 else:  
 index = time.parent  
 if time.parent.left == time:  
 time.parent.left = None  
 else:  
 time.parent.right = None  
 while index is not None:  
 self.balance\_number(index)  
 index = self.balance(index)  
 self.root = index  
 index = index.parent  
 else:  
 if time.height\_right > time.height\_left:  
 time.value = time.right.value  
 self.delete\_vertex(time.right)  
 else:  
 time.value = time.left.value  
 self.delete\_vertex(time.left)  
  
 def find(self, value): #выполняет поиск узла с заданным значением value в дереве и возвращает его. Если узел не найден, возвращается None.  
 index = self.root  
 flag = True  
 while flag and index is not None:  
 if value > index.value:  
 index = index.right  
 elif value < index.value:  
 index = index.left  
 else:  
 flag = False  
 if index is None:  
 return None  
 else:  
 return index  
  
 # log(n)  
 def exists(self, value): #проверяет, существует ли узел с заданным значением value в дереве. Возвращает строку 'true', если узел существует, и 'false' в противном случае.  
 if self.find(value):  
 return 'true'  
 return 'false'  
  
 def next(self, value): #находит минимальное значение, большее заданного value в дереве, и возвращает его. Если такого значения нет, возвращается строка 'none'.  
 time = self.next\_time(value, self.root)  
 if time == 10 \*\* 10:  
 return 'none'  
 else:  
 return time  
  
 def next\_time(self, value, link): #рекурсивно находит минимальное значение, которое больше заданного значения value в дереве, начиная с узла link.  
 if link is not None:  
 min\_l = self.next\_time(value, link.left)  
 if min\_l <= value:  
 min\_l = 10 \*\* 10  
 min\_r = self.next\_time(value, link.right)  
 if min\_r <= value:  
 min\_r = 10 \*\* 10  
 return min(min\_l, min\_r, link.value if link.value > value else 10 \*\* 10)  
 else:  
 return 10 \*\* 10  
  
 def prev(self, value): #использует prev\_time(value, link) для поиска максимального значения, меньшего заданного значения value в дереве.  
 time = self.prev\_time(value, self.root) #Функция prev возвращает найденное значение или строку 'none', если такого значения нет.  
 if time == -10 \*\* 11:  
 return 'none'  
 else:  
 return time  
  
 def prev\_time(self, value, link): #рекурсивно находит максимальное значение, которое меньше заданного значения value в дереве, начиная с узла link.  
 if link is not None:  
 min\_l = self.prev\_time(value, link.left)  
 if min\_l >= value:  
 min\_l = -10 \*\* 11  
 min\_r = self.prev\_time(value, link.right)  
 if min\_r >= value:  
 min\_r = -10 \*\* 11  
 return max(min\_l, min\_r, link.value if link.value < value else -10 \*\* 11)  
 else:  
 return -10 \*\* 11  
  
 def balance\_number(self, vertex): #пересчитывает значения высоты и количества узлов в поддеревьях для указанного узла vertex.  
 if vertex:  
 vertex.height\_left = 0  
 vertex.height\_right = 0  
 if vertex.left:  
 vertex.height\_left = 1 + max(vertex.left.height\_right, vertex.left.height\_left)  
 if vertex.right:  
 vertex.height\_right = 1 + max(vertex.right.height\_right, vertex.right.height\_left)  
 self.balance\_sum(vertex)  
  
 def balance\_sum(self, vertex): # пересчитывает суммарное количество узлов в поддеревьях для указанного узла vertex.  
 if vertex:  
 vertex.count\_left = 0  
 vertex.count\_right = 0  
 if vertex.left:  
 vertex.count\_left = 1 + vertex.left.height\_right + vertex.left.height\_left  
 if vertex.right:  
 vertex.count\_right = 1 + vertex.right.height\_right + vertex.right.height\_left  
  
 def balance(self, vertex): #выполняет балансировку дерева для указанного узла vertex в случае несоответствия баланса.  
 if vertex:  
 dif = vertex.height\_right - vertex.height\_left  
 if dif > 1:  
 if vertex.right.height\_right < vertex.right.height\_left:  
 vertex = self.blt(vertex)  
 else:  
 vertex = self.slt(vertex)  
 elif dif < -1:  
 if vertex.left.height\_right > vertex.left.height\_left:  
 vertex = self.brt(vertex)  
 else:  
 vertex = self.srt(vertex)  
 return vertex  
  
 def slt(self, vertex): #операцию малого левого поворота (single left rotation) для балансировки АВЛ-дерева.  
 if vertex is not None and vertex.height\_right - vertex.height\_left > 0 and vertex.right.height\_left <= vertex.right.height\_right:  
 time\_par = vertex.parent  
 time\_left = vertex.right.left  
 vertex.right.left = vertex  
 vertex.parent = vertex.right  
 vertex.right = time\_left  
 vertex.parent.parent = time\_par  
 if time\_par is not None:  
 if time\_par.left == vertex:  
 time\_par.left = vertex.parent  
 else:  
 time\_par.right = vertex.parent  
 self.balance\_number(vertex)  
 vertex = vertex.parent  
 self.balance\_number(vertex)  
 return vertex  
  
 def srt(self, vertex):  
 if vertex is not None and vertex.height\_left - vertex.height\_right > 0 and vertex.left.height\_right <= vertex.left.height\_left:  
 time\_par = vertex.parent  
 time\_right = vertex.left.right  
 vertex.left.right = vertex  
 vertex.parent = vertex.left  
 vertex.left = time\_right  
 vertex.parent.parent = time\_par  
 if time\_par is not None:  
 if time\_par.left == vertex:  
 time\_par.left = vertex.parent  
 else:  
 time\_par.right = vertex.parent  
 self.balance\_number(vertex)  
 vertex = vertex.parent  
 self.balance\_number(vertex)  
 return vertex  
  
 def blt(self, vertex):  
 if vertex is not None and vertex.height\_right - vertex.height\_left >= 2 and vertex.right.height\_left > vertex.right.height\_right:  
 vertex.right = self.srt(vertex.right)  
 vertex = self.slt(vertex)  
 return vertex  
  
 def brt(self, vertex):  
 if vertex is not None and vertex.height\_left - vertex.height\_right >= 2 and vertex.left.height\_right <= vertex.left.height\_left:  
 vertex.left = self.slt(vertex.left)  
 vertex = self.srt(vertex)  
 return vertex  
  
 def find\_max(self, n, vertex=None): #находит n-ое по величине значение в дереве и возвращает его.  
 if not vertex:  
 vertex = self.root  
 if vertex.count\_right + 1 == n:  
 return vertex.value  
 elif vertex.count\_right >= n:  
 return self.find\_max(n, vertex.right)  
 else:  
 return self.find\_max(n - vertex.count\_right - 1, vertex.left)  
  
  
sys.stdin = open("input.txt", "r")  
sys.stdout = open("output.txt", "w")  
  
tree = AVLTree()  
n = int(sys.stdin.readline())  
for i in range(n):  
 s = sys.stdin.readline().split()  
 if s[0] == '+1' or s[0] == '1':  
 tree.insert(int(s[1]))  
 elif s[0] == '-1':  
 tree.delete(int(s[1]))  
 elif s[0] == '0':  
 print(tree.find\_max(int(s[1])))  
  
print("Время работы: %s секунд " % (time.perf\_counter() - t\_start))  
print(str(psutil.virtual\_memory().available \* 100 / psutil.virtual\_memory().total) + " MB")

Текстовое объяснение решения:

В данной задаче воспользуемся обходом узлов в отсортированном порядке. Функции insert для вставки kтого узла, delete для удаления kтого узла. В функции удаления есть

важные моменты:

1) если удаляемый узел – лист, то просто удаляем его,

2) У удаляемого узла есть только один дочерний элемент – скопируем дочерний элементв узел и удалим его,

3) У узла, который нужно удалить, есть два потомка - найдем неупорядоченный преемник узла, скопируем содержимое неупорядоченного преемника в узел и удалим неупорядоченного преемника.

Для каждого узла мы знаем, сколько элементов слева и справа. На основании этого смотрим условия.

Результат работы кода на примерах из текста задачи:

|  |
| --- |
|  |

Тестирование алгоритма:

|  |  |  |
| --- | --- | --- |
|  | Время выполнения (seconds) | Затраты памяти (mbs) |
| Пример | 0. 0008553999999999923 | 14. 23828125 |

Вывод по задаче: В этой задаче я написала программу, реализующую структуру данных, позволяющую

добавлять и удалять элементы, а также находить k-й максимум.

## Задача №10. Проверка корректности [2 s, 256 Mb, 2 балла]

* Текст задачи:  
  Свойство двоичного дерева поиска можно сформулировать следующим образом: для каждой вершины дерева выполняется следующее условие:

• все ключи вершин из левого поддерева меньше ключа вершины V ;

• все ключи вершин из правого поддерева больше ключа вершины V.

Дано двоичное дерево. Проверьте, выполняется ли для него свойство двоичного дерева поиска.

* 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\*10^5, |Ki| ≤ 109.
* Формат выходного файла (output.txt).   
  Выведите «YES», если данное во входном файле дерево является двоичным деревом поиска, и «NO», если не является.

Листинг кода:

import time  
import os, psutil  
  
t\_start = time.perf\_counter()  
process = psutil.Process(os.getpid())  
  
  
class BNode:  
 def \_\_init\_\_(self, value=0, left=0, right=0):  
 self.value = value  
 self.left = left  
 self.right = right  
  
def tree\_input():  
 with open("input.txt", "r") as f:  
 n = int(f.readline())  
 if n == 0:  
 return True  
 arr = []  
 for \_ in range(n):  
 inp = list(map(int, f.readline().split()))  
 arr.append(BNode(inp[0], inp[1] - 1, inp[2] - 1))  
 result = binary\_tree\_check(arr, 0, -float('inf'), float('inf'))  
 if result:  
 return True  
 return False  
  
  
def binary\_tree\_check(inp, i, left, right):  
 if i == -1:  
 return True  
 if inp[i].value <= left or inp[i].value >= right:  
 return False  
 check = binary\_tree\_check(inp, inp[i].left, left, inp[i].value) and binary\_tree\_check(inp, inp[i].right, inp[i].value, right)  
 return check  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 res = tree\_input()  
 with open("output.txt", "w") as f:  
 if res:  
 f.write('YES')  
 else:  
 f.write('NO')  
  
  
print("Time of working: %s second" % (time.perf\_counter() - t\_start))  
print("Memory", process.memory\_info().rss/(1024\*1024), "mb")

Текстовое объяснение решения:

1) Определение класса BNode для узлов дерева, имеющего значение,

левого и правого потомков.

2) Определение функции tree\_input для чтения данных из файла,

создания дерева и проверки на соответствие условиям двоичного

дерева.

3) Определение функции binary\_tree\_check для проверки дерева на

соответствие условиям двоичного дерева.

4) Запись результата проверки в файл output.txt.

5) Вывод времени выполнения и использованной памяти в консоль.

Результат работы кода на примерах из текста задачи:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Тестирование алгоритма:

|  |  |  |
| --- | --- | --- |
|  | Время выполнения (seconds) | Затраты памяти (mbs) |
| Пример | 0. 0008553999999999923 | 14. 23828125 |
| Пример | 0. 0008553999999999923 | 14. 23828125 |
| Пример | 0. 0008553999999999923 | 14. 23828125 |

Вывод по задаче: В этой задаче я научилось проверять является ли дерево двоичным деревом поиска.

# Вывод

Я вспомнила как работать с классами, узнала, что такое бинарное дерева и узнала как выполнять проверки и производить какие-то действия с данной структурой данных.