Университет ИТМО

Факультет инфокоммуникационных технологий
1 курс
Лабораторная работа №2

Выполнила:
Чагина Вероника Александровна

группа К3144

Преподаватель:

Харьковская Татьяна Александровна

Дата выполнения: 19.04.2022

Санкт-Петербург

Основная часть

6 Задача. Опознание двоичного дерева поиска

В этой задаче 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:

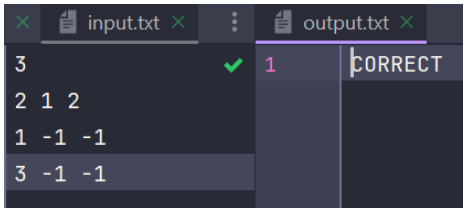
• все ключи вершин из левого поддерева меньше ключа вершины V;

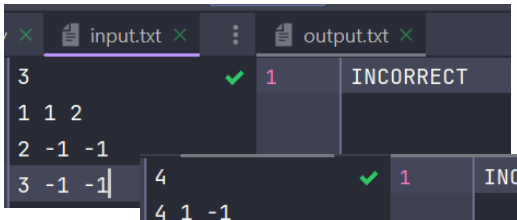
• все ключи вершин из правого поддерева больше ключа вершины V.

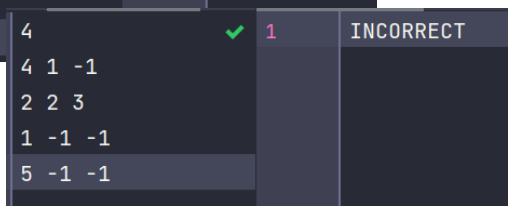
Решение:

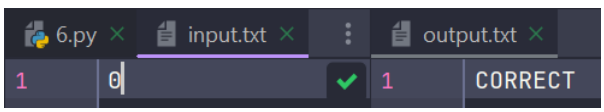
```
def check(root):
    if root is None:
        return None
    que = [(root, -1000000000, 10000000000)]
    while len(que) > 0:
        line = que.pop(0)
        isnode = line[0]
        if isnode.left is not None:
            if (isnode.left.key >= isnode.key) or (isnode.left.key >= line[2]) or
(isnode.left.key <= line[1]):
                return False
            que.append((isnode.left, line[1], isnode.key))
        if isnode.right is not None:
            if (isnode.right.key <= isnode.key) or (isnode.right.key >= line[2]) or
(isnode.right.key <= line[1]):
                return False
            que.append((isnode.right, isnode.key, line[2]))
    return True
~~~
for i in range(n):
    left = L[i]
    right = R[i]
    if left != -1:
        nodes[i].left = nodes[left]
    if right != -1:
        nodes[i].right = nodes[right]
filein.close()

result = 0
if nodes:
    result = check(nodes[0])
else:
    result = True
```

Тесты:









Вывод:

Для работы с деревьями нужно использовать не рекурсивный подход, так как из-за большого количества элементов в дереве высока вероятность переполнения стека вызовов.

8 Задача. Высота дерева возвращается

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

Решение:

```
for i in range(n - 1, -1, -1):
    if (woods[i][0] == 0) and (woods[i][1] == 0):
        deeps[i + 1] = 1
    else:
        deeps[i + 1] = max(deeps[woods[i][0]], deeps[woods[i][1]]) + 1
fileout.write(str(deeps[1]) if n > 0 else "0")
```

В решении я сохраняю все данные в массиве, каждый элемент которого хранит в себе номера левой и правой ветви(кортеж). Прохожу по всему дереву начиная с листьев. Если рассматриваемая вершина - лист, устанавливаю в массиве глубин ей единицу, если ветви вершины, то беру максимальную глубину и прибавляю к ней единицу. Таким образом в корне нашего дерева будет хранится максимальная глубина дерева.

Сложность алгоритма O(n).

Тесты:

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.406	38408192	3989144	6
51	OK	0.046	10342400	200543	2
52	OK	0.406	31907840	3953465	2

Вывод:

При обратном прохождении по дереву - от листьев к корню значительно уменьшается время выполнения задачи.

11 Задача. Сбалансированное двоичное дерево поиска

Входной файл содержит описание операций с деревом, их количество N не превышает 105. 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.

Решение:

```
def main(f):
    tree = Tree()
    n = f.readlines()
    for s in n:
        s = s.split()
        if s[0] == 'insert':
            tree.insert(int(s[1]))
        elif s[0] == 'exists':
            print(tree.exists(int(s[1])))
        elif s[0] == 'prev':
            print(tree.prev(int(s[1])))
        elif s[0] == 'next':
            print(tree.next(int(s[1])))
        else:
            tree.delete(int(s[1]))
```

Тесты:

input.txt

```
insert 2
insert 5
insert 3
exists 2
exists 4
next 4
prev 4
delete 5
next 4
prev 4
```

```
true
false
5
3
none
3
```

Доп.Задачи (Задачи 2, 3, 7, 14, 15, 17)

2 Задача. Гирлянда

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 не концевой лампы на 1 мм меньше, чем средняя высота ближайших соседей. Требуется найти минимальное значение высоты второго конца B, такое что для любого ε > 0 при высоте второго конца B + ε для всех лампочек выполняется условие h > 0. Обратите внимание на то, что при данном значении высоты либо ровно одна, либо две соседних лампочки будут иметь нулевую высоту.

Решение:

```
while less(left, right):
    heights[1] = (left + right) / 2
    heights[-1] = 0
    isUp = False
    for i in range(2, n):
        heights[i] = 2 * heights[i-1] - heights[i-2] + 2
        if not more(heights[i], 0):
            isUp = True
            break
    if more(heights[-1], 0):
        res = min(res, heights[-1])
    if isUp:
        left = heights[1]
    else:
        right = heights[1]
```

В решении используется двоичный поиск для установки значения лампочки, идущей за первой, из уравнения высоты каждой лампочки выражено значение высоты правой лампочки и вычисляется в цикле на основании значений двух левых лампочек.

Тесты:

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.093	9265152	14	13
249	OK	0.015	9121792	12	12
250	OK	0.015	9150464	12	13

3 Задача. Простейшее BST

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).

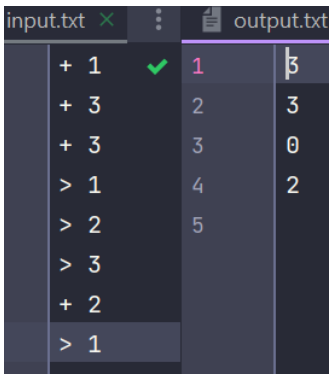
«> x» – вернуть минимальный элемент больше x или 0, если таких нет.

Решение:

```
filein = open("input.txt")
fileout = open("output.txt", "w")

tree = Tree()


line = filein.readline()
while line != "":
    items = line.split()
    if items[0] == "+":
        tree.insert(int(items[1]))
    else:
        fileout.write(f"{tree.min(int(items[1]))}\n")
    line = filein.readline()
```

В поддереве производим поиск элемента большего чем данный.

Тесты:



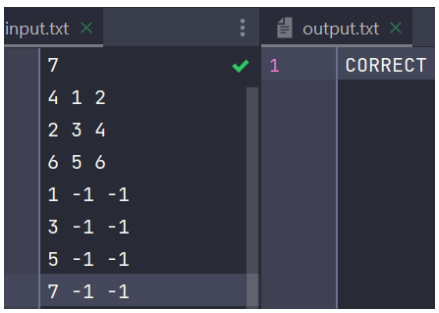
7 Задача. Усложненная задача 6

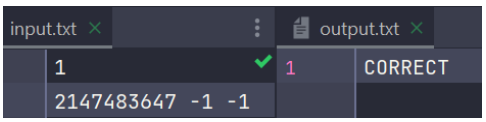
Решение:

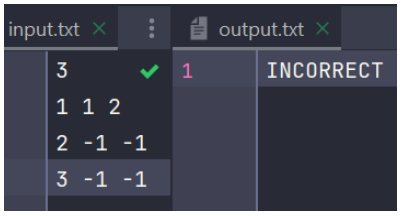
Решение остаётся тем же что и в 6 задаче, меняется только проверка правой ветки.

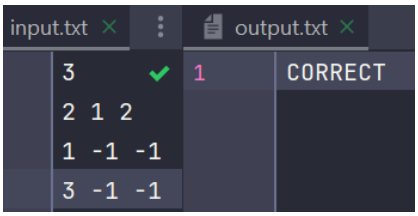
```
if isnode.right is not None:
            if (isnode.right.key < isnode.key) or (isnode.right.key > line[2]) or
(isnode.right.key <= line[1]):
                return False
            que.append((isnode.right, isnode.key, line[2]))
    return True
```

Тесты:









14 и 15 Задачи.

АВЛ-дерево- вставка и удаление

Решение(задача 15):

```
class Tree:
    def height(self, root):
        return root.height if root is not None else 0

    def balance_factor(self, root):
        return self.height(root.right) - self.height(root.left)

    def fix_height(self, root):
        left = self.height(root.left)
        right = self.height(root.right)
        root.height = max(left, right) + 1

    def rotateR(self, root):
        q = root.left
        root.left = q.right
        q.right = root
        self.fix_height(root)
        self.fix_height(q)
        return q

    def rotateL(self, root):
        p = root.right
        root.right = p.left
        p.left = root
        self.fix_height(root)
        self.fix_height(p)
        return p

    def balance(self, root):
        self.fix_height(root)
        if self.balance_factor(root) == 2:
            if self.balance_factor(root.right) < 0:
                root.right = self.rotateR(root.right)
            return self.rotateL(root)
        if self.balance_factor(root) == -2:
            if self.balance_factor(root.left) > 0:
                root.left = self.rotateL(root.left)
            return self.rotateR(root)
        return root

    def insert(self, root, key):
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)
        return self.balance(root)

    def find_right(self, root):
        if root.right is not None:
            return self.find_right(root.right)
        return root

    def find_right_and_delete(self, root):
        if root.right is not None:
            if root.right.right is None:
                root.right = root.right.left if (root.right.left is not None)
else None
            else:
                root.right = self.find_right_and_delete(root.right)
```

```
        return self.balance(root)

    def remove(self, root, key):
        if root is None:
            return None
        if key < root.key:
            root.left = self.remove(root.left, key)
        elif key > root.key:
            root.right = self.remove(root.right, key)
        else:
            if (root.left is None) and (root.right is None):
                return None
            elif root.left is None:
                return root.right
            else:
                new_root = self.find_right(root.left)
                root.key = new_root.key
                if root.left.key == new_root.key:
                    root.left = None if (root.left.left is None) else
root.left.left
                else:
                    root.left = self.find_right_and_delete(root.left)
        return self.balance(root)

    def get_str(self, root):
        if root is None:
            return None
        que = []
        number = 1
        que.append(root)
        ans = []
        while len(que) > 0:
            node = que.pop(0)
            line = f"{node.key} "
            if node.left is not None:
                number += 1
                line += f"{number} "
                que.append(node.left)
            else:
                line += "0 "

            if node.right is not None:
                number += 1
                line += f"{number}\n"
                que.append(node.right)
            else:
                line += "0\n"
            ans.append(line)
        return ans
```

В данном решение реализованы стандартные операция вставки и удаления из АВЛ-дерева, для операции вставки реализованы дополнительные методы для поворота дерева вокруг элемента – влево и вправо, для удаления реализованы дополнительные методы поиска максимума и его удаления.

Тесты:

14:

221	OK	0.687	57110528	1932558	1932526
222	OK	0.718	56864768	1875036	1875006
223	TL	Превышение	94883840	3597394	0

15:

257	OK	0.703	56807424	1930200	1930208
258	OK	0.687	57671680	1861244	1861252
259	TL	Превышение	97222656	3510448	0

Вывод:

Сбалансированные деревья предоставляют огромное преимущество для поиска элементов и работе над деревом.

17 Задача. Множество с суммой

В этой задаче ваша цель – реализовать структуру данных для хранения набора целых чисел и быстрого вычислениясуммы 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 ≤ v ≤ r.

Решение:

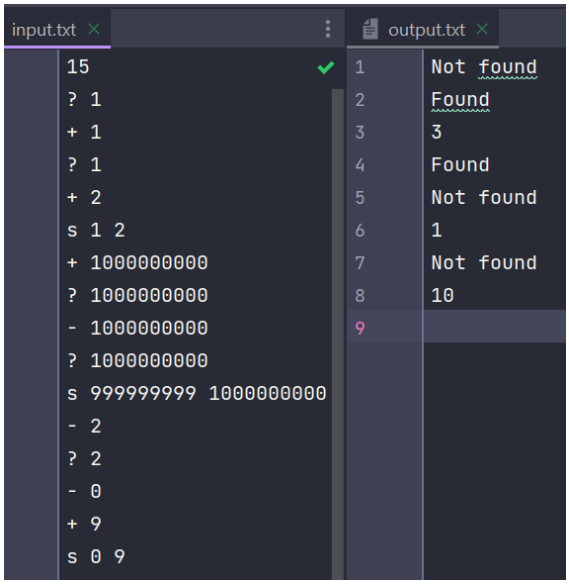
```
class SplayTree:
    def set_parent(self, child, parent):
        if child is not None:
            child.parent = parent

    def keep_parent(self, node):
        self.set_parent(node.left, node)
        self.set_parent(node.right, node)

    def rotate(self, parent, child):
        grandparent = parent.parent
        if grandparent is not None:
            if grandparent.left == parent:
                grandparent.left = child
            else:
                grandparent.right = child

        if parent.left == child:
            parent.left, child.right = child.right, parent
        else:
            parent.right, child.left = child.left, parent

        self.keep_parent(child)
        self.keep_parent(parent)
        child.parent = grandparent

    def splay(self, node):
        if node.parent is None:
            return node
        parent = node.parent
        grandparent = parent.parent
        if grandparent is None:
            self.rotate(parent, node)
            return node
        else:
            zigzig = (grandparent.left == parent) == (parent.left == node)
            if zigzig:
```

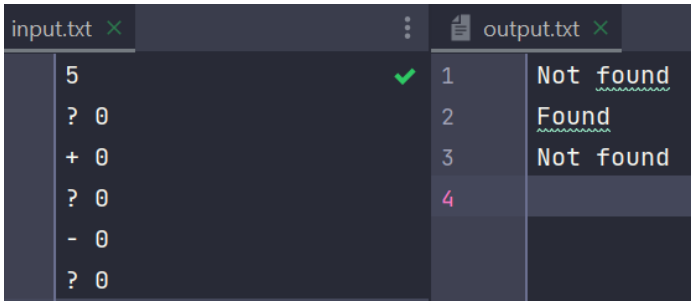
```
                self.rotate(grandparent, parent)
                self.rotate(parent, node)
            else:
                self.rotate(parent, node)
                self.rotate(grandparent, node)
        return self.splay(node)

    def find(self, node, key):
        if node is None:
            return None
        if key == node.key:
            return self.splay(node)
        if key < node.key and node.left is not None:
            return self.find(node.left, key)
        if key > node.key and node.right is not None:
            return self.find(node.right, key)
        return self.splay(node)

    def split(self, root, key):
        if root is None:
            return None, None
        root = self.find(root, key)
        if root.key == key:
            self.set_parent(root.left, None)
            self.set_parent(root.right, None)
            return root.left, root.right
        if root.key < key:
            right, root.right = root.right, None
            self.set_parent(right, None)
            return root, right
        else:
            left, root.left = root.left, None
            self.set_parent(left, None)
            return left, root

    def insert(self, root, key):
        left, right = self.split(root, key)
        root = Node(key, left, right)
        self.keep_parent(root)
        return root

    def merge(self, left, right):
        if right is None:
            return left
        if left is None:
            return right
        right = self.find(right, left.key)
        right.left, left.parent = left, right
        return right

    def remove(self, root, key):
        root = self.find(root, key)
        self.set_parent(root.left, None)
        self.set_parent(root.right, None)
        return self.merge(root.left, root.right)

    def sum(self, root, l, r):
        stack = deque()
        nums = []
        if root is None:
            return 0
        while True:
            stack.append(root)
            if (root.left is None) or (root.key <= l):
                break
            root = root.left
        while True:
            if len(stack) != 0:
                nums.append(stack[-1].key)
```

```
                if (stack[-1].right is not None) and (stack[-1].key <= r):
                    root = stack.pop().right
                else:
                    stack.pop()
                    continue
                while True:
                    stack.append(root)
                    if (root.left is None) or (root.key <= l):
                        break
                    root = root.left
            if len(stack) == 0:
                summ = 0
                for num in nums:
                    if (num >= l) and (num <= r):
                        summ += num
                return summ
```

В данном решении реализовано Splay дерево (самобалансирующееся бинарное дерево поиска. Дереву не нужно хранить никакой дополнительной информации, что делает его эффективным по памяти.), с функциями добавления, удаления, поиска и получения суммы чисел в определённом диапазоне.

Тесты:





Вывод:

В данной лабораторной работе мы изучили бинарные деревья, их балансировку и функции, которые они могут выполнять. Реализовали простейшее бинарное дерево, сбалансированное дерево AVL, разобрали правые и левые повороты и другие методы.

1. АВЛ-деревья полезны для быстрого поиска элементов, поиска минимума, максимума и прочих полезных операций.
2. Splay деревья необходимы в случаях работы с большим объёмом данных, в котором очень часто ищется/используется конкретное множество данных.