Университет ИТМО

Инфокоммуникационные технологии и системы связи Программирование в инфокоммуникационных сетях

Лабораторная работа #2

Самойлин Артём Олегович (K3120)

Преподаватель: Харьковская Татьяна Александровна

Г. Санкт-Петербург

19.04.2021

1 Задача. Обход двоичного дерева.

В этой задаче вы реализуете три основных способа обхода двоичного дерева «в глубину»: центрированный (inorder), 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 (postorder) обходы в глубину.

Решение:

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


def in_order(root: Node):
    if root:
        yield from in_order(root.left)
        yield root.key
        yield from in_order(root.right)


def pre_order(root: Node):
    if root:
        yield root.key
        yield from pre_order(root.left)
        yield from pre_order(root.right)


def post_order(root: Node):
    if root:
        yield from post_order(root.left)
        yield from post_order(root.right)
        yield root.key
```

Тесты:

input	output
5 4 1 2 2 3 4 5 -1 -1 1 -1 -1	1 2 3 4 5 4 2 1 3 5 1 3 2 5 4

3 -1 -1	

Вывод по задаче:

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:

```
def output(result):
    print(result)
    with open("output.txt", "w") as output_file:
        output_file.write(result)


def equal(a, b):
    return abs(a-b) <= 0.1 ** 10


def less(a, b):
    return a < b and not equal(a, b)


def more(a, b):
    return a > b and not equal(a, b)


def main():
    n, height = read_input_file()
    heights = [0] * n
    heights[0] = height
    res = inf
    left, right = 0, heights[0]
    while less(left, right):
        heights[1] = (left + right)/2
        heights[-1] = 0
        flag = False
        for i in range(2, n):
            heights[i] = 2 * heights[i-1] - heights[i-2] + 2
```

```
            if not more(heights[i], 0):
                flag = True
                break
        if more(heights[-1], 0):
            res = min(res, heights[-1])
        if flag:
            left = heights[1]
        else:
            right = heights[1]

    output(("%.6f" % res))
```

Тесты:

input	output
8 15 692 532.81	9.75 446113.34434782615

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.078	9285632	14	13
1	OK	0.015	9191424	9	8
2	OK	0.046	9191424	12	13
3	OK	0.031	9166848	9	8
4	OK	0.015	9191424	11	8
5	OK	0.031	9179136	9	8
6	OK	0.015	9207808	9	8
7	OK	0.031	9175040	14	13
8	OK	0.031	9211904	12	13
9	OK	0.031	9183232	11	13
10	OK	0.031	9179136	13	13
11	OK	0.031	9220096	10	8
12	OK	0.031	9166848	13	13
13	OK	0.031	9125888	10	8
14	OK	0.015	9240576	10	8
15	OK	0.031	9236480	12	13

Вывод по задаче:

В некоторых задачах необходимо преобразовывать первоначальные формулы и задумываться над использованием двоичного поиска

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:

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

    def insert(self, key):
        parent = None
        node = self.root
        while node is not None:
            parent = node
            if key < node.key:
                node = node.left
            elif key > node.key:
                node = node.right
            else:
                return None

        new = Node(key)
        if parent is None:
            self.root = new
        elif key < parent.key:
            parent.left = new
        elif key > parent.key:
            parent.right = new

    def min(self, x):
        if self.root is None:
            return 0

        way = []
        node = self.root
        while True:
            way.append(node)
            if x > node.key:
                if node.right is None:
                    break
                node = node.right
            elif x < node.key:
                if node.left is None:
                    return node.key
                node = node.left
```

```
            else:
                if node.right is None:
                    break
                node = node.right
                while node.left is not None:
                    node = node.left
                return node.key
        for i in range(len(way) - 1, -1, -1):
            if way[i].key > x:
                return way[i].key
        return 0
```

Тесты:

input	output
+ 1 + 3 + 3 > 1 > 2 > 3 + 2 > 1	3 3 0 2

Вывод по задаче:

6 Задача. Опознание двоичного дерева поиска

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.

Решение:

```
def descend(root: Node, left_ancestor=None, right_ancestor=None) ->
bool:
   left_is_descending = True
   if root.left is not None:
       if right_ancestor is not None:
           left_is_descending = root.left.key > right_ancestor
       left_is_descending = left_is_descending and root.left.key <
root.key and descend(root.left,

left_ancestor=root.key)

   right_is_descending = True
   if root.right is not None:
       if left_ancestor is not None:
           right_is_descending = root.right.key < left_ancestor
       right_is_descending = right_is_descending and root.key <=
root.right.key and descend(root.right,

right_ancestor=root.key)

   return left_is_descending and right_is_descending
```

Тесты:

input	output
3 2 1 2 1 -1 -1 3 -1 -1	CORRECT

input	output
3 1 1 2 2 -1 -1 3 -1 -1	INCORRECT

input	output
0	CORRECT

input	output
4 4 1 -1 2 2 3 1 -1 -1 5 -1 -1	INCORRECT

7 Задача. Опознание двоичного дерева поиска (усложненная версия)

Эта задача отличается от предыдущей тем, что двоичное дерево поиска может содержать равные ключи.
Вам дано двоичное дерево с ключами - целыми числами, которые могут повторяться. Вам нужно проверить, является ли это правильным двоичным деревом поиска. Теперь, 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 или равны ключу вершины V .

Другими словами, узлы с меньшими ключами находятся слева, а узлы с большими ключами – справа, дубликаты всегда справа. Вам необходимо проверить, удовлетворяет ли данная структура двоичного дерева этому условию.

Решение:

```
def descend(root: Node, left_ancestor=None, right_ancestor=None) ->
bool:
   left_is_descending = True
   if root.left is not None:
       if right_ancestor is not None:
           left_is_descending = root.left.key > right_ancestor
       left_is_descending = left_is_descending and root.left.key <
root.key and descend(root.left,
```

```
left_ancestor=root.key)

    right_is_descending = True
    if root.right is not None:
        if left_ancestor is not None:
            right_is_descending = root.right.key < left_ancestor
        right_is_descending = right_is_descending and root.key <=
root.right.key and descend(root.right,

right_ancestor=root.key)

    return left_is_descending and right_is_descending
```

Тесты:

input	output
3 1 1 2 2 -1 -1 3 -1 -1	INCORRECT

input	output
3 2 1 2 1 -1 -1 2 -1 -1	CORRECT

input	output
1 2147483647 -1 -1	CORRECT

input	output
3 2 1 2	INCORRECT

2 1 -1 3 -1 -1	

Вывод по задаче:

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:

```
def read_input_file():
    with open("input.txt") as inp:
        n = int(inp.readline())
        woods = []
        for _ in range(n):
            value, left, right = map(int, inp.readline().split())
            woods.append((left, right))
    return n, woods

def output(answer: str):
    print(answer)
    with open("output.txt", "w") as out:
        out.write(answer)

def main():
    n, woods = read_input_file()
    deeps = [0] * (n + 1)
    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
    output(str(deeps[1]) if n > 0 else "0")

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

Тесты:

input	output
6 -2 0 2 8 4 3 9 0 0 3 6 5 6 0 0 0 0 0	4

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.359	38277120	3989144	6
1	OK	0.031	9015296	46	1
2	OK	0.031	9019392	3	1
3	OK	0.015	9007104	11	1
4	OK	0.031	9011200	18	1
5	OK	0.046	8970240	103	1
6	OK	0.015	9015296	76	2
7	OK	0.015	9019392	155	2
8	OK	0.015	9027584	163	2
9	OK	0.031	9027584	57	1
10	OK	0.015	9060352	161	1
11	OK	0.015	9015296	2099	1
12	OK	0.031	9015296	1197	3
13	OK	0.015	9011200	2073	3
14	OK	0.015	9011200	2139	3
15	OK	0.031	9043968	686	1
16	OK	0.000	9080832	2128	2
17	OK	0.015	9080832	8777	1
18	OK	0.031	9007104	10426	3
19	OK	0.015	9121792	16336	3
20	OK	0.031	9080832	16835	3
21	OK	0.062	9019392	3520	1
22	OK	0.015	9089024	16969	2

Вывод по задаче:

Изначально я сделал прямой проход, от корня к листьям, и такое решение не прошло последний тест, после этого я решил изменить подход и пройтись с листьев к корню,

что значительно увеличило скорость, поэтому при решении задач необходимо задумываться: не будет ли эффективно развернуть решение и пойти от обратного. Также не всегда необходимы все данные, которые предоставляются в задаче.

14 Задача. Вставка в АВЛ-дерево

Вставка в АВЛ-дерево вершины V с ключом X при условии, что такой вершины в этом дереве нет, осуществляется следующим образом:

• находится вершина W, ребенком которой должна стать вершина V ;

• вершина V делается ребенком вершины W;

• производится подъем от вершины W к корню, при этом, если какая-то из вершин несбалансирована, производится, в зависимости от значения баланса, левый или правый поворот.

Первый этап нуждается в пояснении. Спуск до будущего родителя вершины V осуществляется, начиная от корня, следующим образом:

• Пусть ключ текущей вершины равен Y .

• Если X < Y и у текущей вершины есть левый ребенок, переходим к левому ребенку.

• Если X < Y и у текущей вершины нет левого ребенка, то останавливаемся, текущая вершина будет родителем новой вершины.

• Если X > Y и у текущей вершины есть правый ребенок, переходим к правому ребенку.

• Если X > Y и у текущей вершины нет правого ребенка, то останавливаемся, текущая вершина будет родителем новой вершины.

Отдельно рассматривается следующий крайний случай – если до вставки дерево было пустым, то вставка новой вершины осуществляется проще: новая вершина становится корнем дерева.

Решение:

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


def height(root):
```

```
    return root.height if root is not None else 0


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


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


class Tree:
    @staticmethod
    def rotate_r(root):
        q = root.left
        root.left = q.right
        q.right = root
        fix_height(root)
        fix_height(q)
        return q

    @staticmethod
    def rotate_l(root):
        p = root.right
        root.right = p.left
        p.left = root
        fix_height(root)
        fix_height(p)
        return p

    def balance(self, root):
        fix_height(root)
        if balance_factor(root) == 2:
            if balance_factor(root.right) < 0:
                root.right = self.rotate_r(root.right)
            return self.rotate_l(root)
        if balance_factor(root) == -2:
            if balance_factor(root.left) > 0:
                root.left = self.rotate_l(root.left)
            return self.rotate_r(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 = None if root.right.left is None else
root.right.left
            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)

    @staticmethod
    def stringify(root):
        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
```

Тесты:

input	output
2 3 0 2 4 0 0 5	3 4 2 3 3 0 0 5 0 0

Превышение ограничения на время работы, тест 259

Подсказка: это случайно сгенерированный корректный тест размера 179211 и высоты 18.

Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		Превышение	98455552	3510448	2757348
1	OK	0.031	8835072	20	24
2	OK	0.031	8777728	6	10
3	OK	0.015	8814592	14	18
4	OK	0.031	8863744	13	17

Вывод по задаче:

Превышение времени происходит из-за долгого ввода данных

15 Задача. Удаление из АВЛ-дерева

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.

Решение:

Приведенный выше класс для 14 задания (avl.Tree)

Тесты:

input	output
3 4 2 3 3 0 0 5 0 0 4	2 3 0 2 5 0 0

Превышение ограничения на время работы, тест 223
Подсказка: это случайно сгенерированный корректный тест размера 179211 и высоты 18.

Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		Превышение	95961088	3597394	1932526
1	OK	0.015	9244672	27	17
2	OK	0.031	9183232	13	1
3	OK	0.015	9265152	20	10
4	OK	0.031	9220096	20	10
5	OK	0.031	9273344	20	10
6	OK	0.031	9293824	20	10
7	OK	0.031	9121792	27	17
8	OK	0.031	9289728	27	17
9	OK	0.015	9220096	27	17
10	OK	0.015	9281536	34	24
11	OK	0.015	9289728	34	24
12	OK	0.031	9240576	34	24
13	OK	0.031	9256960	34	24
14	OK	0.015	9216000	34	24
15	OK	0.000	9211904	34	24
16	OK	0.062	9236480	34	24
17	OK	0.031	9265152	34	24
18	OK	0.031	9236480	34	24
19	OK	0.031	9240576	34	24
20	OK	0.031	9261056	34	24

Вывод по задаче:
Превышение времени происходит из-за долгого ввода данных

16 Задача. K-й максимум

Напишите программу, реализующую структуру данных, позволяющую добавлять и удалять элементы, а также находить k-й максимум.

Решение:

```
class MaxHeap:
    def __init__(self):
        self.heap_list = []
        self.size = 0

    def heapify(self, root_index):
        largest = root_index
        left_child = (2 * root_index) + 1
        right_child = (2 * root_index) + 2

        if left_child < self.size and self.heap_list[left_child] >
self.heap_list[largest]:
            self.heap_list[largest], self.heap_list[left_child] =
self.heap_list[left_child], self.heap_list[largest]
            self.heapify(left_child)

        if right_child < self.size and self.heap_list[right_child] >
self.heap_list[largest]:
            self.heap_list[largest], self.heap_list[right_child] =
self.heap_list[right_child], self.heap_list[largest]
            self.heapify(right_child)

    def build_heap(self, l: list):
        self.heap_list = l
        self.size = len(l)
        for i in range(len(l) // 2, -1, -1):
            self.heapify(i)

    def add(self, x):
        self.heap_list.append(x)
        self.size = len(self.heap_list)
        index = self.size - 1
        parent = (index-1) // 2

        while parent >= 0 and index > 0:
            if x > self.heap_list[parent]:
                self.heap_list[index], self.heap_list[parent] =
self.heap_list[parent], self.heap_list[index]
            index = parent
            parent = (index-1) // 2

    def index(self, x):
        if x == self.heap_list[0]:
            return 0
        for i in range(self.size//2):
            if x == self.heap_list[2*i + 1]:
                return 2*i + 1
```

```
            if x == self.heap_list[2*i + 2]:
                return 2*i + 2
        return -1

    def remove(self, index):
        self.heap_list[index], self.heap_list[-1] = self.heap_list[-1],
self.heap_list[index]
        element = self.heap_list.pop(-1)
        self.size = len(self.heap_list)
        self.heapify(0)
        return element

    def find_n_max(self, n):
        return nlargest(n, self.heap_list)[-1]
```

Тесты:

input	output
11 +1 5 +1 3 +1 7 0 1 0 2 0 3 -1 5 +1 10 0 1 0 2 0 3	7 5 3 10 7 3

Вывод по задаче:
Использовал бинарную кучу с максимумом

17 Задача. Множество с суммой

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 ≤ v ≤ r.

Решение:

```
class SplayTree:
    @staticmethod
    def set_parent(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
```

Тесты:

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

s 999999999 1000000000 - 2 ? 2 - 0 + 9 s 0 9	

Вывод по задаче:

Splay деревья выгодно использовать в том случае, когда у нас система, которая часто ищет и обращается к одним и тем же элементам.