CS-202

Data Structures

Assignment 2

Topics: Trees

Due Date: March 9, 11:55 pm

Learning Outcomes

In this assignment, you will:

- Practice implementation of general BSTs and AVL Trees (i.e., self-balancing BSTs)
- Perform various operations on general BSTs and AVL Trees
- Practice how Trees can be used in real-world scenarios

Pre-Requisites

For this assignment, you will require:

- Thorough understanding of Binary Search Trees
- Thorough understanding of AVL Trees
- Time and space complexity

Structure

This assignment has three parts. **Part 1** is on Binary Search Trees. **Part 2** is on AVL Trees. **Part 3** is a real-world application of trees.

The deadline for this assignment is strict. No extension will be provided. You can use your grace days if you are unable to finish your assignment on time.

Refer to the end of this manual (**Page 10**) for the Grading Scheme and instructions for automated testing.

Plagiarism Policy

The course policy about plagiarism is as follows:

1. Students must not share the actual program code with other students.

2. Students must be prepared to explain any program code they submit.

3. Students cannot copy code from the Internet.

4. Students must indicate any assistance they received.

5. All submissions are subject to automated plagiarism detection. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Best of Luck!

Part 1: Binary Search Trees

In **Part 1**, you will apply and build upon what you learned in class to implement different functionalities of a Binary Search Tree. A skeleton code in given in the file **part1.py**.

It contains a class **node** that has three attributes: value, left child, and right child.

Task:

Your task is to complete all functions of the Binary Search Tree. ***Do not modify any function definitions or add/remove any parameters that each function receives***. You may add helper functions if you want, but your test cases should pass for you to receive credit.

It is necessary to complete all functions within this part before moving to the next. Else, there is no guarantee that subsequent parts would work.

Note: When implementing functions, pay special attention to corner cases. There should be no hard coding, and we will test your code on a separate set of test cases in order to ensure you haven't hardcoded anything. You should implement all functions **recursively**.

Member functions:

Write implementation for the following methods as described here:

- insert(self, child): **(2 points)**
  - insert a new node of value *child* in the BST. If node addition unsuccessful, return False.
- delete(self, key): **(3 points)**
  - delete a node in the BST
  - Properties of the BST must not be violated
- delete_leaf(self, key): **(2 points)**
  - check if key is a leaf node and delete it from the tree
- delete_leaves(self): **(3 points)**
  - delete all leaves of a tree
- find_node(self, key): **(1.5 points)**
  - return True if node with the given key exists. Return False otherwise
- get_node(self, key): **(1 points)**
  - find node with the given key and return pointer to that node
- get_children(self, key): **(2 points)**
  - return all children of the node with the given key
- update_node(self, key, val): **(3 points)**

    - update the node with value *key* and replace *key* with *val* only if the properties of a BST are not violated
    - Return False otherwise
- height(self): **(3 points)**
    - return the height of the tree (tree with only root node has height 1 in this case)
- in_order(self): **(0.5 points)**
    - return a in-order traversal of the tree, and return it in a list
- pre_order(self): **(0.5 points)**
    - return a pre-order traversal of the tree, and return it in a list
- post_order(self): **(0.5 points)**
    - return a post-order traversal of the tree, and return it in a list
- path(self, key): **(4 points)**
    - return a path from the root node to the node with value key
- avg_diff(self): **(4 points)**
    - return the difference in average of all keys in the left subtree and the average of all keys in the right subtree [avg(right) – avg(left)]

Note: You should implement a print function first, so you can test your code easily! Further, please write the time complexity for each function as a comment in front of the function declaration! 5 marks will be deducted from part 1 of the assignment if you fail to do so.

Part 2: AVL Trees

In **Part 2**, you will apply and build upon what you learned in class to implement different functionalities of an AVL Tree. A skeleton code in given in the file **part2.py**. It contains a class **node** that has four attributes: value, left child, right child, and height.

Task:

Your task is to complete all functions of the AVL Tree. ***Do not modify any function definitions or add/remove any parameters that each function receives***. You may add helper functions if you want, but your test cases should pass for you to receive credit.

It is necessary to complete all functions within this part before moving to the next. Else, there is no guarantee that subsequent parts would work.

Note: When implementing functions, pay special attention to corner cases. There should be no hard coding, and we will test your code on a separate set of test cases in order to ensure you haven't hardcoded anything. **Some functions (e.g. pre_order) have been implemented in part 1 as well, and you may reuse them here. They have been declared in the skeleton code, but there is no credit for them in part 2.**

Member functions:

Write implementation for the following methods as described here:

- def insert(self, root, key): **(3 points)**
  - Used to insert the key in the tree. Make sure after the each insertion the AVL property holds.
- def l_rotate(self, z): **(2 point)**
  - Used to rotate the tree when it is unbalanced
- def r_rotate(self, z): **(2 point)**
  - Used to rotate the tree when unbalanced
- def get_height(self): **(2 points)**
  - Used to get the **height of the node**
- def get_bal(self): **(2 points)**
  - helper function to obtain the difference in heights of the left and right subtrees of a node (The leaf node should have the height of 1)
- def delete_node(self, node_to_be_deleted): **(7 points)**
  - You are required to delete the node passed
- def update_node(self, new_value_of_node): **(2 points)**
  - Used to update the value of an existing node

- def delete_all_nodes_at_given_height(self, level): **(10 points)**
    - This function is the real deal-- you have to delete all the nodes present at give level for example. If level = 1, you are required to delete 2 and 6.

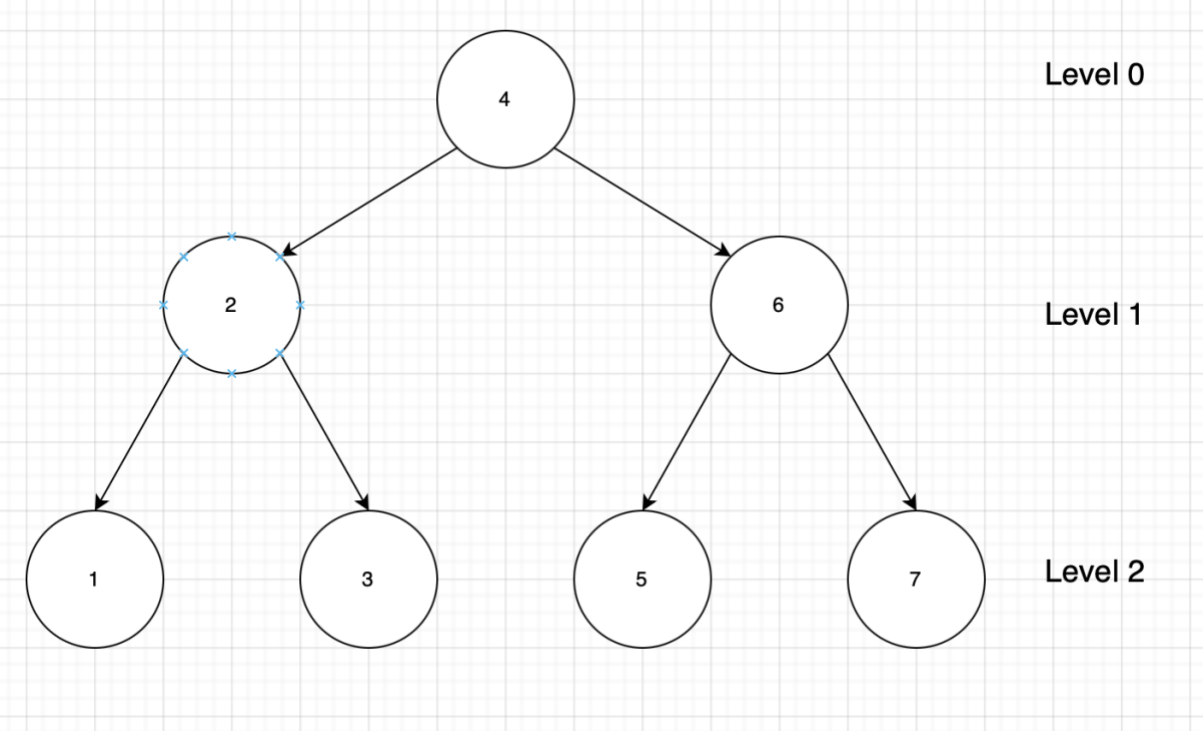

Figure 1

Note: please write the time complexity for each function as a comment in front of the function declaration! 5 marks will be deducted from part 2 of the assignment if you fail to do so.

Part 3: Application

In **Part 3,** you will implement everything you have learned in a real-life scenario. A skeleton code in provided in the file **part3.py**. A riddle before starting the third part. Just imagine you are in a room with no windows and a door surrounded by fire. What do you do to escape it? Think! Read the question again ;) Unable to solve it? Don't worry, the answer is simple-- just stop imagining!

Now, let's start with part three. Now just imagine you have been hired as manager of one of the most unique hotels in the world. This hotel is unique because there are no corridors in the hotel. In fact, each room in the hotel has two doors, which lead to rooms on the next floor. For instance, in *Figure 2*, room 1 (which is the lobby) has two doors—one has stairs leading to room 2, while the other door has stairs leading to room 3 (which are both on the same floor but are accessible only through room 1).

This hotel structure is like a **general tree**, where each room can have a maximum of 2 doors. The top floor (floor number 1) starts with only one room and as you move up the floors, the number of rooms on that floor is equal to 2^n. For example, when you are on the first floor, the number of rooms on that floor would be 2; when you are on the third floor, the number of rooms on that floor would be 2^3 = 8. In this hotel, on each odd level floor (1, 3, 5, ...), the rooms should be in descending order, while on even number floors (0, 2, 4), the rooms should be in ascending order. In *Figure 2*, we have two floors so the rooms would have the following order:

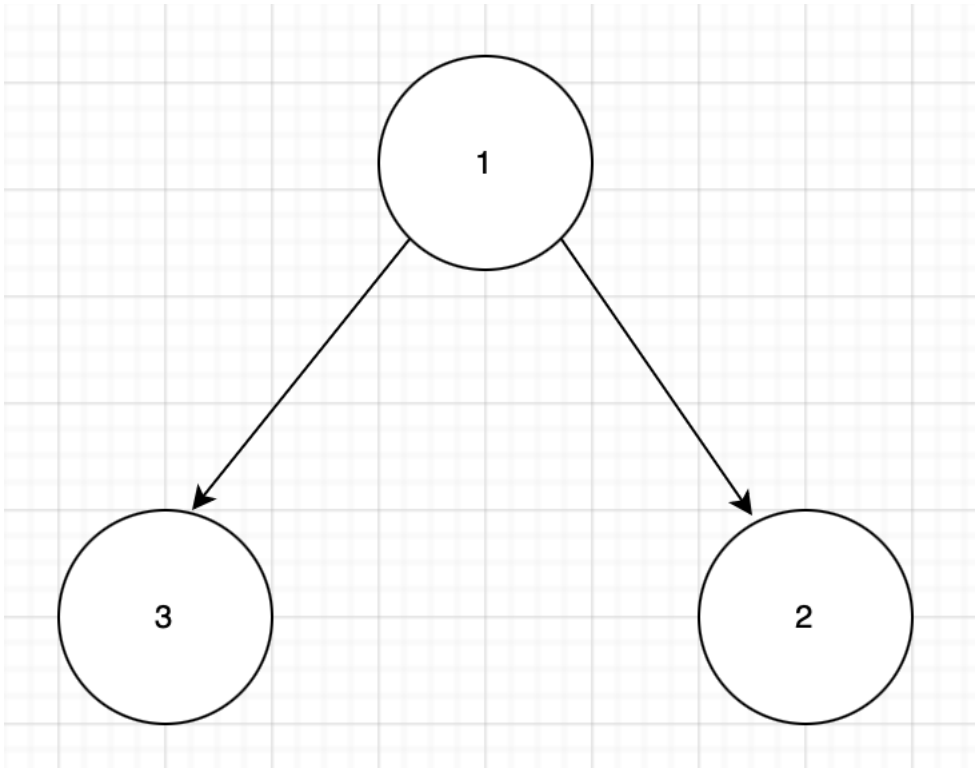

Figure 2

Unfortunately, during some renovation processes, rooms on the odd floor have been assigned wrong numbers, as shown in *Figure 3*, where the rooms on the odd floors are also arranged in ascending order, resulting in the following numbering:

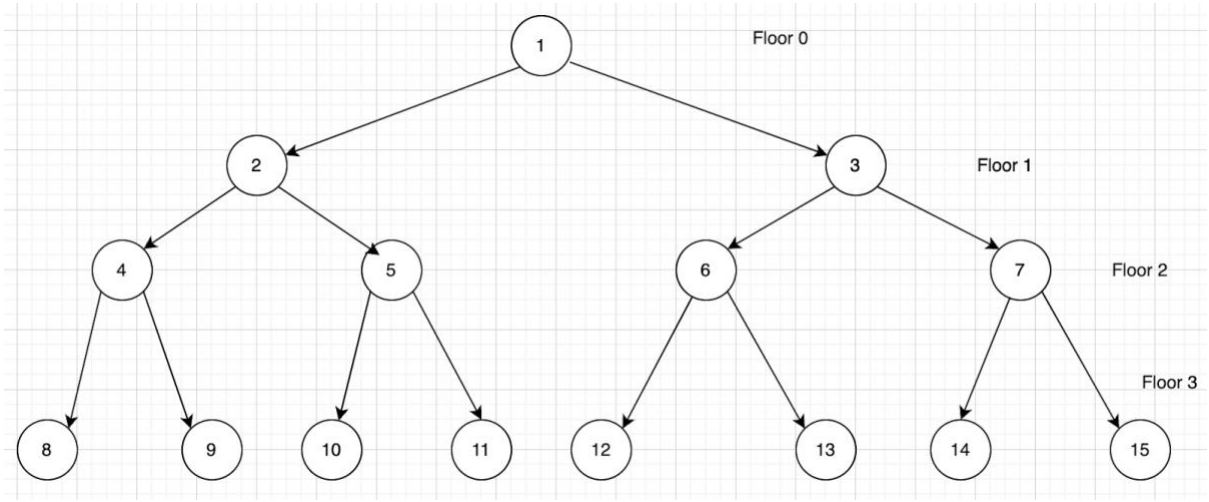

Figure 3

But all of the staff is not worried at all because they have an intelligent manager like you to solve this issue. Being a manager, your task is to come up with an algorithm to convert the above give structure of the hotel to the structure given in *Figure 4*:

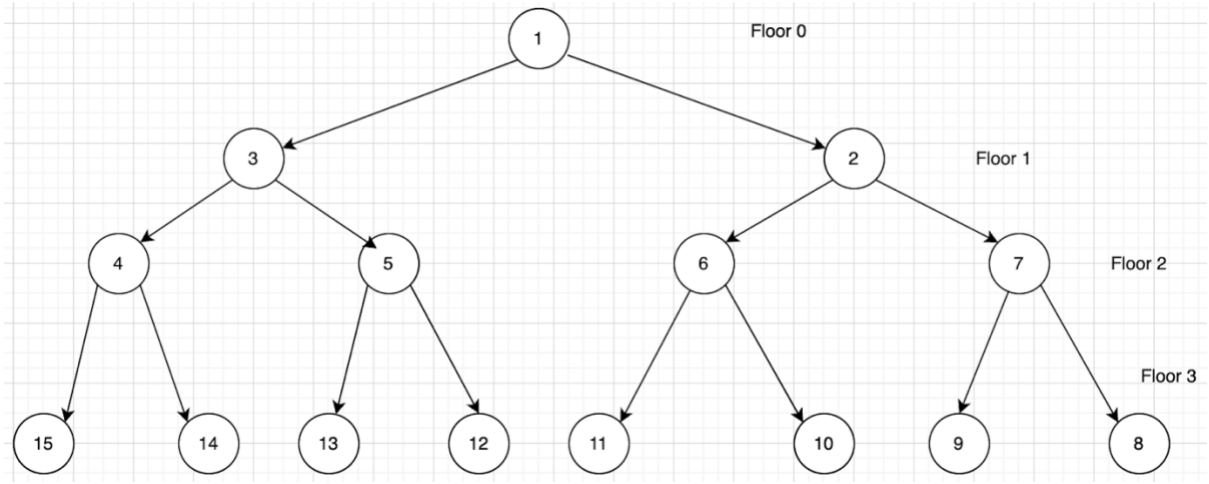

Figure 4

Constraints:

1. Your code must be generic enough to work on **any** number of floors
2. Your algorithm should be in O(n) (or better!)

Result:

The answer to problem above you should be : [‘1’, ‘3’, ‘2’ ,’4’, ‘5’, ‘6’, ‘7’,’15’, ‘14’ ,’13’, ‘12’ ,‘11’ ,‘10’ ,‘9’ ,‘8’] and you are required to visit each floor and store the room

number inside the list return the list of these values in visiting_floor function. List must be of type string

You need to implement the following two functions to get the desired result: **visiting_floors** and **changing_room_numbers**. Each function is worth 20 points.

Your solution should be general and will be tested on a different set of test cases. Hard-coding your answer will result in a 0 in the entire part 3.

Note: please write the time complexity for each function as a comment in front of the function declaration! 5 marks will be deducted from part 3 of the assignment if you fail to do so.

Best of luck!

Grading Scheme

There is a single file that will be used to test all parts: *test.py*

To test your implementation, run this file on the terminal in the same folder as your other files. Each function across all three parts is tested independently of each other. The test will show marks for each individual function, along with your total marks. Overall breakdown is as below:

Part 1: 30 marks

Part 2: 30 marks

Part 3: 40 marks

Each function carries individual marks. The points breakdown is written in front of each function. Your assignment will be graded against a different set of test cases (similar to the ones provided to you, but with different values), so any attempt at hard-coding your solutions will result in a 0 in that specific part.

There will be no partial marking between each function. Pay special emphasis to corner cases, as they will be checked by the grader provided to you, and by the test cases we will use.

Note: You can create an object of the particular class within each part and use the functions that you have written in order to test your implementation. However, please make sure to comment out this portion before you run your tests as it can mix up with the test cases.

If you need any help, please do not resort to unfair means, and feel free to reach out to any of the teaching staff. All of us are here to help you learn and grow as budding computer scientists, and above all, as responsible professionals.

Best of luck! All of you are fully capable of attempting this assignment, and we are sure you will do great.