The Great Binary Adventure: Five Days of BST Mastery

Step 1: The Quest Begins – Creating your BST

I created two classes: a Node class and a BST class.

A Node has 4 fields: id, left, right, height. The id field is a unique integer that represents the node in the tree (it’s also its value). The left and right fields are pointers to nodes that are the left and right sons of the node in the tree. The height fields save the current height of the node in the tree.

The BST class is made of one field: root. The root is a pointer to a node which is the root node of the BST.

I implemented the BST class so that it has the property of an AVL Tree (which is a balanced BST).

When a BST is created, a document is created to represent it. This document simply contains a root field that indicates the id of the current root of the BST.

In this part I implemented an insert(treasure) method for the BST class. In this function, we create a new node object and we also insert a new document in mongoDB corresponding to this node. Then we search the right place to insert a new node according to the properties of a BST (all the nodes in left subtree have a smaller id, all in right subtree have a bigger id). Once we found it, we connect the new node to the tree in the corresponding place. Then we have to update heights for each node along the insertion path and check if the balance factor of each node doesn’t exceed an absolute value of 2 (so that the tree stays balanced). If we found an unbalanced node, we perform the required rotation on its subtree (LL, LR, RR, or RL). While performing a rotation we update heights.

All along the insertion process, everytime we update connections between nodes we make sure to update the corresponding mongoDB documents.

Step 2: Dealing with Dark Forces – Deleting Treasures

In this part I implemented a delete(treasure) method to the BST class. In this function, we search for the desired node according to the properties of a BST. Once we found it, we delete it according to the delete procedure in a BST. Then we have to check for unbalanced nodes along the deletion path. For each node we update heights and check if the node is unbalanced. If necessary we perform corresponding rotations.

Again we make sure to update the mongoDB documents impacted by the delete.

Step 3: The Search for the Hidden Gem – Seek and You Shall Find

In this part I implemented a search(treasure) method for the BST class. In this function, we search for the desired hidden gem according to its id in the BST tree. The search procedure is done according to the properties of the BST. Once we found it we print a statement saying that we found it and we print its id.

Step 4: The Secrets of Tree Knowledge – Traversals

In this part I implemented three new methods to the BST class: in\_order(), pre\_order() and post\_order(). These three methods have a similar implementation. The only thing that changes is the printing order among the subtrees.

For the in\_order method, the function first tries to call recursively to in\_order on the left subtree (if there is one) in order to traverse it first. Then it prints the current node. Finally it tries tries to call recursively to in\_order on the right subtree.

The logic is the same for the other two methods, but they call to themselves in a different order according to the traversal method.

Step 5: The Grande Finale – Validation and Visualization

In this part I implemented a validate() method. In this function, we check recursively for each node and its subtrees if the properties of an AVL tree are respected. We start from the root and we check if the root id is bigger than the maximum id in its left subtree (the max is calculated by an helper function that finds it by running recursively on the whole subtree), we also check if the root id is smaller than the minimum id in its right subtree. Finally we check that the root is balanced. Then we check recursively if all these conditions hold for the left and right subtree and so on.

Finally if all the conditions hold we return True, else we return False.

I also implemented a visualize() method. This method builds recursively a json document that represents the entire tree. Each node is represented with three fields: id (which is its id as an integer), left (which is a nested json document representing the left son node), right (same as left but for the right son node).

To build the representation, the function calls recursively to itself to build the left and right node representation, and so on all along the tree.

Finally we get a hierarchy of nested documents that allow us to follow the structure of the tree by looking at its nodes.

We also save this representation in mondoDB as a document.

For example:

{

"tree": {

"treasure": 2,

"left": {

"treasure": 1,

"left": null,

"right": null

},

"right": {

"treasure": 6,

"left": {

"treasure": 3,

"left": null,

"right": null

},

"right": null

}

},

"\_id": {

"$oid": "65b14e41897de144280849d6"

}

}