Runtime analysis, Classes, and functions’ documentation:

Class: AVLNode

Description**:**

Represents a node in an AVL tree. Each node has the following fields:

key - sorted type, allows to determine the location of the node.

value - the data we will keep in this node.

left - the root of the left subtree - a node with a smaller key.

right - the root of the right subtree - a node with a bigger key.

parent - the node that connects our node to the rest of the tree.

height - the maximum size of the node’s subtrees- represents the longest path from the node to one of the leaves in its subtree.

isVirtual- a boolean indicator that defines if the node is an actual node or a virtual one. This field is used for balancing and programming reasons.

Functions:

__init__(*self*) | $O(1)$

Set the key and the value of the node to be the input values, and initiate all other fields as none. The default height is -1, and we set the default node to be of ‘not virtual’ type.

get_left(*self*) | $O(1)$

Return the left child of the node, None if there is no left child.

get_right(*self*) | $O(1)$

Return the right child of the node, None if there is no right child.

get_parent(*self*) | $O(1)$

Return the parent of the node.

get_key(*self*) | $O(1)$

Return the key of the node, None for a virtual node.

get_value(*self*) | $O(1)$

Return the value of the node, None for a virtual node.

get_height(*self***) |** $O(1)$

Return the height of the node, -1 for a virtual node.

get_bf(*self***) |** $O(1)$

Return the balance factor of the node, by calculating the subtraction of the height of the left child and the height of the right child.

set_left(*self, node***) |** $O(1)$

Sets the left child of the node to be the input of the function.

set_right(*self, node***) |** $O(1)$
Sets the right child of the node to be the input of the function.

set_parent(*self, node***) |** $O(1)$

Sets the parent of the node to be the input of the function.

set_key(*self, key***) |** $O(1)$

Sets the key of the node to be the input of the function.

set_value(*self, value***) |** $O(1)$

Sets the value of the node to be the input of the function.

set_height(*self, h***) |** $O(1)$

Sets the height of the node to be the input of the function.

set_virtual(*self***) |** $O(1)$

Setting the field “isVirtual” as true.

set_not_virtual(*self***) |** $O(1)$

Setting the field “isVirtual” as false.

is_virtual(*self***) |** $O(1)$

Returns the value of “isVirtual”- true for a virtual node, false otherwise.

is_leaf(*self***) |** $O(1)$

Return true if both of the node’s children are virtual nodes. False otherwise.

is_inner_node_right_sonl(*self*) | $O(1)$

Returns true if the key of the current node is bigger then the parent's key, false otherwise.

Class: AVLTree

Description**:**

Represents an AVL tree. Each tree has the following fields:

Root - the first key in the tree. The root is an AVLNode and has all of its traits.

Size - the total amount of nodes in the whole tree.

Every AVL tree consists of AVLNodes which are comparable objects due to the ordered keys they possess. This allows us a fast way to reach and manipulate every node, which can contain many different types of data.

Functions:

__init__(*self*) | $O(1)$

Set the root of the new tree to None, and the size to 0.

search(*self*, *key*) | $O(logn)$

Assuming the key we are looking for is not a key in the tree then we have 'fallen' from the tree. In this case, we have gone through the full path from the root to one of the leaves of the tree. Since the height of an AVL is $O(logn)$,and in each iteration of the loop we dive one level deeper in the tree, then we have $O(logn)$ iterations. In each iteration we perform $O(1)$ work because all we do is small finite comparisons with no correlation to the input. Therefore comparisons will cost us $O(1)$. All in all, we have run time of Search is $O(logn) * O(1) = O(logn)$.

bst_tree_insert(*self*, *key*, *val*) | $O(1)$

This is a helper function for the insert function. First we create the new node, give it virtual children and set its height. Next, we give it a parent, and tell its parent it has a new right or left child. Finally return the new node.

bst_tree_position(*self*, *key*, *val*) | $O(logn)$

Assuming the key we are looking for is not a key in the tree then we have 'fallen' from the tree whilst keeping a pointer of the potential father of this key. In this case, we have gone through the full path from the root to one of the leaves of the tree. Since the height of an AVL is $O(logn)$, and in each iteration of the loop we dive one level deeper in the tree, then we have $O(logn)$ iterations. In each iteration we perform $O(1)$ work because all we do is small finite comparisons with no correlation to the input. Therefore the whole algorithm will cost us $O(logn) * O(1) = O(logn)$.

balance_sub_tree(*self, pivot, balanceFactor***) |** $O(1)$

First, check the balance factor of the given node. Now denote 4 turns to rule them all:

1. BF = -2, and BF of right son is 1: perform right then left rotation (RL).
2. BF = -2, and BF of the right son is -1: perform left rotation (L).
3. BF = 2, and BF of left son is -1: perform left then right rotation (LR).
4. BF = 2, and BF of the left son is 1: perfrom right rotation (R).

Keep track of the number of rebalancing operations.
Raise Exception if we calculated a different BF then expected.

nested: **rotation(***node, clockWise, isZigZag***) |** $O(1)$

The node we get is a child. We save its parent, and also a pointer to the parent of the whole subtree.
Next, we change pointers between the parent and his child.
Lastly, update root if needed.

Insert(*self, key, val***) |** $O(logn)$

By design the node we would like to insert is going to be a leaf. In this function we are using the helper functions:

1. **bst_tree_position(***self, key***)**
2. **bst_tree_insert(***self, key, val***)**
3. **balance_sub_tree(***self, pivot, balanceFactor***)**

After inserting the desired node, we climb up from the node we inserted until we reach the root, or until we reach a balanced node (legal balance factor). Denote, as we iterate up towards the root, we will perform balance actions if needed using the **balance_sub_tree** and **rotation** functions. As proved in class, each insert will cause $O(1)$ rotations at most. Therefore, in the worst case we will climb from a leaf to the root, and perform $O(1)$ work on every level while going up.
All in all, the insert algorithm will take:
$O(logn) + O(1) + O(logn) * O(1) = O(logn)$.

delete(*self, node*) | $O(logn)$

This function is composed of two parts:

1. First- deleting the node from the tree - using the helper function **bst_delete** which uses the same logic as deleting a root from a regular binary search tree. In the worst case, we will reach a node that has two children. In this case we'll have to look for the node's successor, and change between their places. Worst case of the **successor** function is $O(logn)$. After finding the successor, we'll detach the node from the tree at a constant price $O(1)$.
2. After detaching the node from the tree, we start iterating from the parent of the deleted node all the way to the root. As we iterate up the tree we'll check for unbalanced nodes, and call on **balance_sub_tree** if needed. As described in the insert algorithm runtime analysis, balancing will cost us $O(logn)$.
   All in all we have delete at the cost of:
   $O(logn) * O(1) + O * (logn) = O(logn)$.

bst_delete(*self, node*) | $O(1)$

Notice 4 types of cases:
1. We are wishing to delete the root or the tree is empty - both cases are taken care of in the delete function.
2. Node is a leaf - detach from tree and put a virtual son instead.
3. Node has an only right\left child - connect between the child and the parent of the deleted node.
4. Node has 2 children - find the successor using a helper function, interchange between them and repeat phase 1,2, or 3 on the successor since it has no left son, by design.

successor(*self, node*) | $O(logn)$

To find a node's successor we must follow one of two algorithms:
1. If the node has a right son, we'll go to the node's right subtree and look for its minimum node. Worst case this will cost us $O(logn)$.
2. Else, we'll climb all the way up until we will reach the root or a left son of its parent (a right turn). In this case, the node we'll stop on will be the required successor. In the worst case we must climb all the way to the root. This will cost us $O(logn)$.

All in all, the total cost of this function is $O(logn)$.

minimum(*self, root*) | $O(logn)$

To find the minimum node of the sub-tree, we must start at the sub-root and keep going left until we reach a node with no left son. This node will be the minimum node in the sub-tree by design of BSTs.
In the worst case we are searching for the minimum of the whole tree, and we must dive down all the way to the deepest leaf. In AVL trees, the deepest node is $O(logn)$ distance from the root, so we have $O(logn)$ iterations. Each iteration will cost $O(1)$ work. All in all, total cost of the function is $O(logn)$.

avl_to_array(*self*) | $O(n)$

This function will start with the minimum node which is retrieved using the **minimum** function. Now we call each node in the tree using the **successor** function. Getting to each node in the tree results in calling the successor $n-1$ times, with n as the size of the tree. As proved in class, calling the successor n times on a tree with n nodes will take O(n) time. All in all, the function will cost $O(logn) + O(n) = O(n)$.

size(*self*) | $O(1)$

Returns the field tree_size that keeps track of how many nodes are in the tree.

split(*self, node*) | $O(logn)$

In this function we will first check if the separating node is the root of the tree. If so, take both its children and set each of them as a root of a tree. Our output in this case is a list that contains both trees. Otherwise, the separating node is not the root. In this case, we will check if the separating node is a right or left inner node, and we'll change the pointers accordingly. Next, detach the separating node from the tree, and create two trees: one from its left son, and another from its right son. Now, we must climb all the way until we reach the root of the original tree. On every level check if the node we are looking at is a left or right inner node. If it is a right child, save the parent's left sub tree as a new tree, and join the new tree with the smaller tree we have created before, using the parent as the joining node. Else, if the node is a left child, repeat the same logic as before on the symmetric case. In the worst case, the separating node is a leaf. In this case, we must iterate all the way to the root, joining one of the two trees we have created with the current node, and its other subtree at every level. As displayed in the presentation, we can sum up all join operations to cost $O(logn)$. All in all the time complexity for this function is $O(logn)$.

join(*self, key, val***) |** O(logn)

Create the joinNode and call the helper function: **actual_join**.
This allows us to avoid duplicating code and using **actual_join** for the **split** function as well, which uses a join method using an existing node and doesn't create it given a key and a value.

actual_join(*self, tree2, joinNode***) |** O(logn)

In this analysis assume the larger tree has n nodes.

In this function, we have four options of joining trees;

1. Joining two empty trees - in this case we will set the joining node as a root of the new tree at the cost of $O(1)$.
2. Joining an empty tree with a non-empty tree. In this case the joining function will operate the same as insert. We'll insert the joining node into the non-empty tree at the cost of $O(logn)$.
3. Joining two trees with 1 or less height difference. In this case each tree is an AVL tree, which is balanced by design. Joining these two trees with the joining node, will create one tree with a legal balance factor for the root since the height difference of its children is one at most. Connecting the joining node as a root will cost us $O(1)$.
4. Joining two trees with a 2 or more height difference. In this case we will look at the taller tree. If this tree has bigger keys then the other tree (or vice versa), we'll dive all the way from the taller tree's root until we will reach a node with the same height as the other tree or with a height difference of 1. If the taller tree has bigger keys, we'll dive to the left, otherwise we'll dive to the right. Next we connect the subtree we just found that has almost the same height as the other tree, using the joining node. Finally, connect the joining node to the parent of the subtree we just detached. Knowing that the subtree we detached is an AVL tree, and the tree we are joining is also an AVL tree, we can assume that the joining tree is also an AVL tree, which is balanced by design. Denote, the worst case in regards to runtime occurs when we go down until we reach a leaf in the larger tree. Once we reach the leaf all that is left is connecting a couple pointers at the cost of $O(1)$. All in, total run time is $O(logn)$.

get_root(*self***) |** O(1)

Returns the AVLNode which represents the root of the tree.

Theoretical section:

i	Average join's cost for randomly chosen splitNode	Maximum join's cost for randomly chosen splitNode	Average join's cost for left tree's maximum node as splitNode	Maximum join's cost for left tree's maximum node as splitNode
1	2.545	5	2.8	13
2	2.727	9	2.25	14
3	2.6	4	2.667	15
4	2.214	5	2.692	17
5	2.77	12	2.93	18
6	2.467	9	2.6875	19
7	3	4	2.5	20
8	2.529	7	2.647	22
9	2.55	8	2.647	22
10	2.473	8	2.526	23

Theoretical questions:

Question 2

In class we have seen the cost of the **join** function is based on the height difference between the two trees.

1. First case - we are splitting by the maximum node of the left sub tree. Let us call this node the splitNode. In this case all the nodes with keys larger than splitNode are in the right subtree of the root. As we iterate up the tree and towards the root we will have to perform many join operations. Since the original tree is a balanced AVL tree all of it's subtrees are also balanced AVL trees. Therefore, after removing the splitNode, all calls to join until we reach the root are going to cost us no more than 2 (height difference). These calls are considered very efficient. The last **join** call will join an empty tree with the whole right subtree of the root. This means that the height difference between the trees can be major and this **join** operation can be very expensive. All in all, we expect the average join to cost us just over 2, because the last **join** call will raise our average, but since it's still only 1 call over many calls the average cannot increase dramatically.
2. Second case - we are splitting by a random node. In this case, we cannot determine with certainty the number and type of **join** calls we'll be performing.

We expect most **join** calls to be less efficient and cost on average more than 2. On the other hand, we won't be paying such a high amount on the **join** call. On average we expected the result between this case and case #1 to be similar.

Looking at the differences between the two experiments, as the size of the tree grows larger we expect the randomly chosen splitNode to deliver smaller changes than the deterministic splitNode, because when randomly chosen the splitNode isn't necessarily deeper down the tree. On the other hand, we expect the deterministic splitNode to deliver bigger changes as the tree grows because the last **join** call will be expensive. This argument is only partially true, because in this case there will be more **join** calls in total and therefore the average will also be brought back down and not just up.

To sum up, our expectation was to see similar results between both ways of choosing the splitNode and a small rise of the average cost as the tree grows. We can see the results in our table are in correlation with theorem.

<u>Question 3</u>
In this case, we will take the maximum node of the left subtree. This node has no right child- otherwise it would be the node we will be splitting by it. As explained earlier, all of the joins we will do in that case on the way up to the root will cost us approximately 2, due to the fact that we are working with a balanced AVL tree. When arriving at the root the last **join** call will be the most expensive join operation. This happens, because we'll have to join the right subtree which has an approximate height of log(n) with an empty tree, and the root of the tree as the joining node. In this case, we'll insert the root to the right subtree. This process will cost us the length of the path from the root of the right subtree to its leaves. This cost is approximately log(n). Looking at the results from our experiment we can see that the maximum cost is log(n)+2 which is the difference between the right subtree and an empty tree. This stands in correlation to what we assumed in the theorem.