CSCI 1107

Spring 2022

Assignment 4: AVLTree

Submit a single-file Python program named avltree.py to D2L by the deadline specified there.

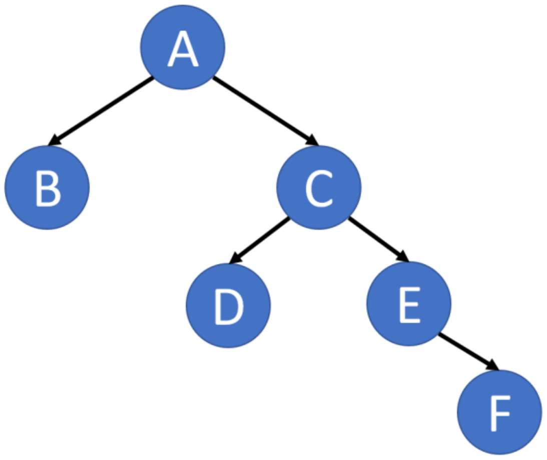
By submitting any work for this (or any other) assignment, you attest that all work you have submitted is your own, and that you are able and willing to fully explain the process you went through to reach any of the answers you submitted on request. Turning in any work that is not your own will result in a score of zero on the full assignment with the possibility of additional consequences as per the course syllabus.

In this assignment, you will be writing a link-based, wrapped implementation of the AVL tree data structure. An **AVL tree** is a binary search tree that has one critical additional characteristic, which we will call the **AVL property**: for every node in the tree, the height of its left subtree must be within 1 of the height of its right subtree. Note that the AVL property is weaker than the "balanced" property. It is possible for a tree to have the AVL property but not be "balanced" in the sense of only missing nodes in its lowest row.

Technically, the **height** of a tree is defined as the longest number of edges from the root of the tree to any of its leaves. By this definition, the height of a single-node tree is 0 and the height of an empty tree is undefined. This does not work well for our purposes here, or for many algorithms that apply recursively to trees – we need an empty tree to have a valid height and for it to be one less than the height of a one-node tree. **We resolve that problem here by defining the height of an empty tree as -1.**

Every node in a tree is the root of its own **subtree**. The **left subtree** of a given node X is the subtree with X's left child as the root. If X has no left child, then X's left subtree is the empty tree. Conversely, the **right subtree** of a node X is the subtree with X's right child as root, or the empty tree if X has no right child.

Consider the example tree below:



A is a binary tree of height 3, because A->C->E->F is the longest path from root to leaf. A's left subtree is of height 0, and A's right subtree is of height 2. Therefore, while A is a binary tree, it is not an AVL tree – it lacks the AVL property. The subtree C, on the other hand, *does* have the AVL property: its left subtree is of height 0 and its right subtree is of height 1, which are within 1 of each other. The subtree D also has the AVL property: both of its subtrees are the empty tree, which have height -1 by our definition.

What is stored in each node of a binary search/AVL tree?

Both binary search trees (BSTs) and AVL trees can be implemented in multiple ways.

One option is to have them behave like a dictionary. Each node in the tree stores both a key and a value. The keys must be unique, but the values need not be. The tree's sorting order is based on comparisons between the keys, not the values.

Another option is to have the tree behave like a set. In this case, each node in the tree stores just one value. To keep things relatively simple, our AVL tree implementation will be this set-like type of tree, not the dictionary type.

In the review of BST functions below, the explanations are tailored to this type of tree.

Review of key binary search tree functions

Binary search trees (BSTs) permit three main operations.

Lookup takes in a value and delves into the tree starting from the root, checking for the presence or absence of that value in the tree, and returning a boolean value. Because of the structure of a binary search tree, lookup can be performed in O(log n) time.

Insert or **add** takes in a value. The add process begins very much like the lookup process – the tree is navigated until the point is reached at which the node *would* exist if it was present. If a node *is* present that already has the value, we can simply stop without altering the tree. (Since we want to behave like a set, "adding" something already present simply does nothing.) If the node is not already present, it's created and made a child of an existing node.

Delete or **remove** takes in a value. Like the other two operations, we begin by navigating down the tree to find the node with this value, if it exists. If it's not present, we can simply stop the operation without changing the tree. If it *is* present, we need to remove the node from the tree while preserving the binary search property. There are three possibilities:

1. If the node we're removing has no children, hooray! We can just remove the link(s) between the parent node and this one and we're done.
2. If the node we're removing has only one child, that's okay. We detach this node from its parent and have the parent link to its child instead.
3. If the node we're removing has two children, uh-oh. The easiest way to resolve this situation is to find another node that we can remove and slide into place of this one. We could pick the rightmost node of our left subtree or the leftmost node of our right subtree, but we'll do the former one. Unfortunately it's possible that the node we kidnap itself falls into category 2 above and thus requires some extra stitching, but fortunately it cannot, by definition, fall into this category 3, so we don't have to worry about recursive replacement.

How do these functions differ in our AVL tree?

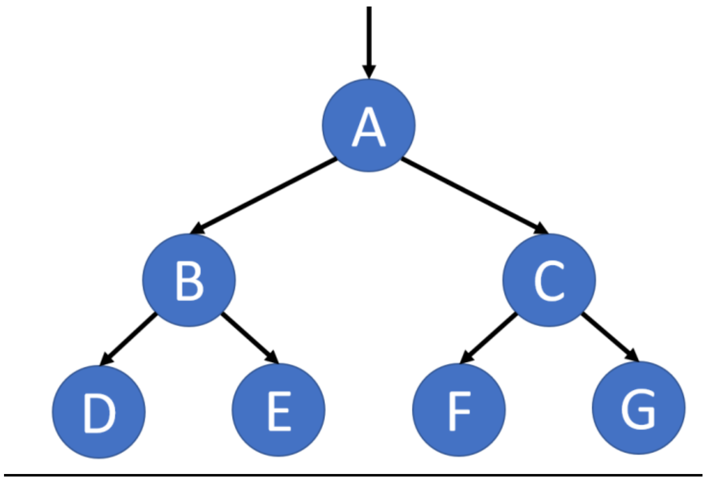
Lookup doesn't differ, but after adding or removing a node our tree may no longer have the AVL property. We have to check our tree and perform any necessary "rotations" to regain the AVL property before we can consider the add or remove operation complete. Rotations will be described afterward.

After adding a new node, we need to check that node *and every node above it in the hierarchy* to ensure that each of those subtrees still has the AVL property. If it does not, we must perform rotations to reassert it, and then continue upward until we reach and validate the AVL property at the root.

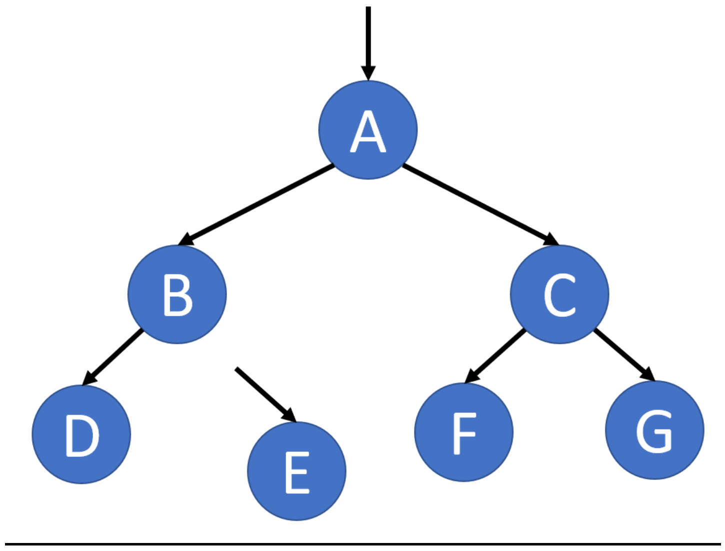
After removing a node, we need to check *the parent of the physically removed node* and every node above it in the hierarchy. When we say "physically removed node", consider the three possibilities listed above when removing a node from a BST. In situations 1 or 2, the node we're physically removing from the tree is the one with the value that was passed in. However, in situation 3 the node with the value we're removing stays where it is, and it's the rightmost member of the left subtree that gets physically removed. We need to check for imbalances upward from the *physically* removed node, not from the node that had its value changed.

Rotation

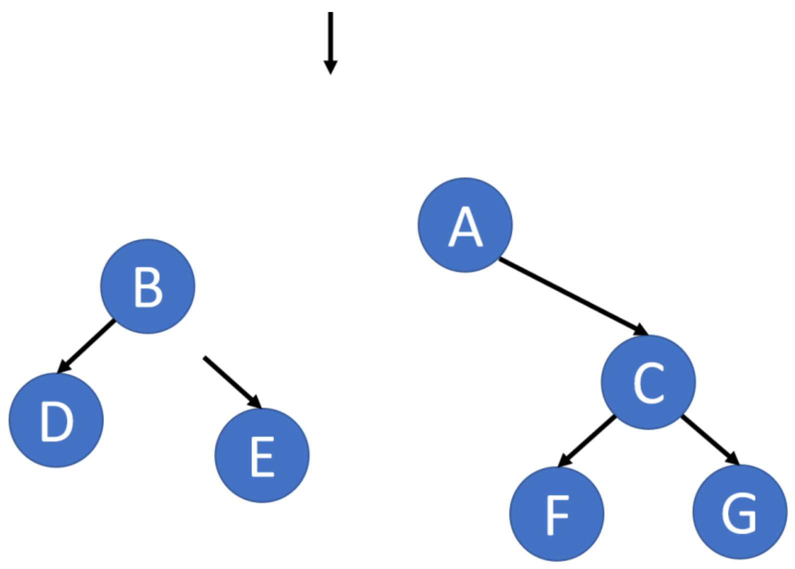
There are two fundamental rotation operations: left and right. The two are simple opposites of each other. Depicted here is a *right* rotation on the node A.



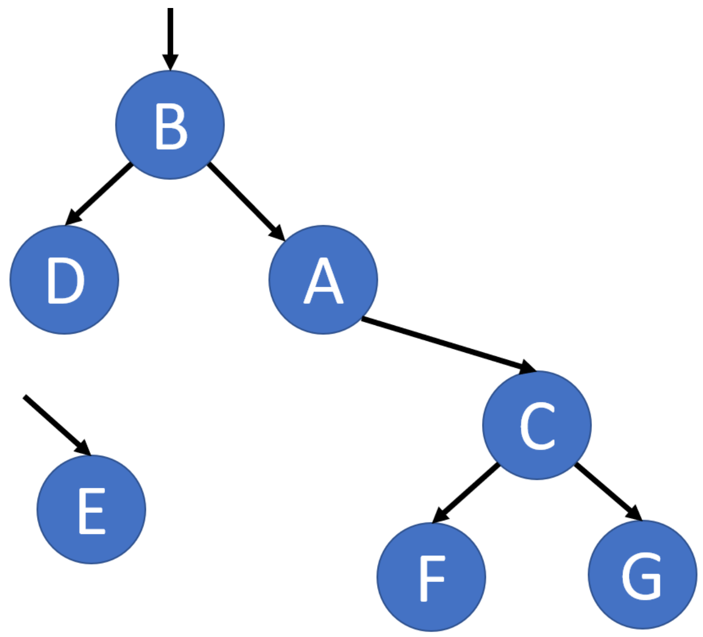
Our goal here is to “rotate” A to the right, meaning that A’s left child, B, will take its place, and A will become B’s right child. Because A is going to be B’s new right child, we have to detach B from its right subtree:



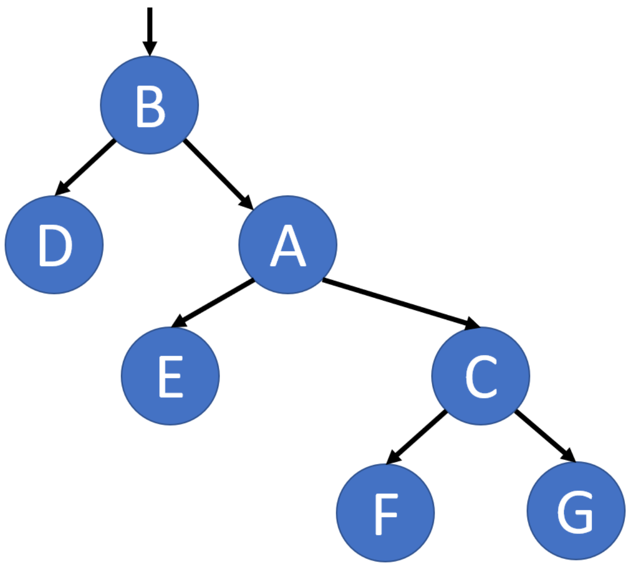
We then remove A from its parent…



…and attach B in its place, with A as its right child.



Lastly, the orphaned right subtree of B is connected to A… as its *left* subtree.



A left rotation is exactly the opposite: C would take A's place in the tree, with A becoming C's left child and F becoming A's right child.

When and how to rotate

We've now discussed both…

1. When we have to scan our tree to see if rotations have to be performed (after insertions and deletions, starting at the inserted node or above the physically deleted node, and climbing upwards until we've validated the root)
2. How a left or right rotation is performed

What we have not discussed is exactly what type of rotation should be performed when we scan a node and find that it no longer has the AVL property.

As we scan our tree upward, we check the **balance factor** of each node: the height of the right subtree minus the height of the left subtree. If the balance factor of the node is -1, 0, or 1, the subtree still has the AVL property. If the balance factor goes to -2 or +2, the AVL property has been lost and a rotation is necessary. In this case, we need to determine which of four situations we are in to know how to rotate correctly.

1. If our balance factor is +2 (indicating that we are right-heavy) and our right child's balance factor is >= 0 (indicating that it is either perfectly balanced or slightly right-skewed), we call this **right-right**. We solve this by rotating the **current node left**.
2. If our balance factor is +2 (indicating that we are right-heavy) but our right child's balance factor is -1 (indicating that it is slightly left-skewed), we call this **right-left**. We

solve this by rotating our **right child right** (to remove its slight left-skew) and then rotating the **current node left**.

3. If our balance factor is -2 (indicating that we are left-heavy) and our left child's balance factor is <= 0 (indicating that it is either perfectly balanced or slightly left-skewed), we call this **left-left.** We solve this by rotating the **current node right**.
4. If our balance factor is -2 (indicating that we are left-heavy) and our left child's balance factor is 1 (indicating that it is slightly right-skewed), we call this **left-right**. We solve this by rotating our **left child left** (to remove its slight right-skew) and then rotating the **current node right.**

Classes

This module requires two classes, both already defined in the provided file.

class AVLTree

This is our wrapper class. It contains only one instance variable: self.root. If self.root is None, then this wrapper represents an empty tree. Any operations the user wishes to perform on the tree should be performed via the wrapper class – the user should not directly access nodes.

The wrapper classes's methods primarily just check to see if the root is None, handle this special case appropriately, and otherwise initiate a recurisve call to the root of the tree.

class AVLTreeNode

Each instance of AVLTreeNode must maintain four instance variables. self.value holds the actual value of the node. self.left and self.right are for the AVLTreeNodes that are self's left and right children. If either child is not present, they're set to None.

The fourth instance variable is self.parent. This is a reference to the AVLTreeNode *above* this node in the tree. This initially seems an unwelcome complication: every edge in the tree wil have to be represented with *two* references rather than one, and anytime we need to move nodes around, we'll have to manipulate *both* references or else end up with an incoherent tree. However, the parent reference is essential. After an insertion or deletion, we have to check each node of the tree from the modified location *upward* until we reach the root. This is only possible if we maintain parent pointers.

Provided methods

AVLTree.__init__(self)

Sets root to None.

AVLTree.__len__(self)

Only actually needed for testing purposes in our case, though a handy method to have in general. If self.root is None, returns 0. Otherwise, passes back the return value of a recursive call to AVLTreeNode.size on self.root.

AVLTree.display(self)

Either prints out [Empty tree] or starts a recursive call that prints out a semi-human-readable form of the tree.

AVLTree.assert_valid(self)

To be used for testing purposes. Checks the tree to ensure that three properties are maintained, raises an AssertionError with a (hopefully) helpful message if any of the three properties is not. These are described in the specific recursive methods below. Note that an empty tree has all three properties and so is automatically valid.

AVLTreeNode.__init__(self,parent,value)

Sets self.parent and self.value based on parameters and self.left and self.right to None. Note that when AVLTree creates the first node in a tree, it must manually specify that the parent value should be None.

AVLTreeNode.display(self,depth=0)

Recursive function that prints out the tree. Each call is responsible for printing out that node and for the left and right children either printing a line representing them (if they are empty) or making a recursive call (if they are not).

AVLTreeNode.__repr__(self)

A repr used by the display method.

AVLTreeNode.assert_sorted(self)

The first of three recursive methods called by AVLTree.is_valid. Checks to see if the subtree starting at this node has the binary search property – all values to the left are smaller than this one, all values to the right are greater than this one. If not, raises an AssertionError.

AVLTreeNode.assert_avl(self)

The second of three recursive methods called by AVLTree.is_valid. Checks to see if the subtree starting at this node has the AVL property – if this node, and all nodes below it, have a balance factor of -1, 0, or 1. If not, raises an AssertionError. This method *does* depend on you having correctly implemented the AVLTreeNode.balance_factor method.

AVLTreeNode.assert_tree(self)

The third of three recursive methods called by AVLTree.is_valid. Checks to see if the subtree starting at this node has fully valid parent references. That is, if self.left exists, self.left.parent should be self, the same should be true for self.right, and the same should be true recursively. If this is not true for all nodes, raises an AssertionError.

AVLTreeNode.size(self)

Recursively computes and returns the total number of nodes in the subtree with this node as the root, including this node itself.

Methods you must implement

AVLTree.lookup(self,value)

Return False if self.root is None, otherwise make a recursive call to the AVLTreeNode.lookup method on self.root.

AVLTree.add(self,value)

If self.root is None, create a new AVLTreeNode with no parent and this value and set it as self.root. Otherwise, make a recursive call to the AVLTreeNode.add method on self.root, *and set the result as self.root*. (The process of adding a new node to the tree might cause the current root to change, so we have to return the root.)

AVLTree.remove(self,value)

If self.root is not None, make a recursive call to the AVLTreeNode.remove method on self.root, *and set the result as self.root*. (For the same reasons as with "add" above.)

AVLTreeNode.lookup(self,value)

If the value we're searching for is present at this node, return True. Otherwise, determine whether the value would exist in the left or right subtree. If said subtree is absent, return False. Otherwise, return the result of a recursive call.

AVLTreeNode.root(self)

Recurisvely find and return the root of the tree this node belongs to. (The base case is that this node is its own root if it has no self.parent.)

AVLTreeNode.add(self,value)

Check if the value should be placed to the left or right of this node. If the left/right subtree already exists, make a recursive call, passing along the new value.

If the appropriate subtree *doesn't* exist, we need to create the new node and add it as the subtree at that location. Having done so, we need to initiate a check_up procedure starting at ourselves, because the addition of this new node may have caused the loss of the AVL property at or above this node.

If the value being added is equal to the value at this node, we needn't modify the tree at all.

After performing whatever alterations are necessary, this method must recursively return a reference to the tree's root, since this may have changed in the process.

I suggest you implement this method first *without* the check_up procedure, which should allow you to add to your tree as though it was a vanilla BST rather than an AVL tree.

AVLTreeNode.rightmost_descendent(self)

Recursively find and return the rightmost descendent of the current node. (The base case is that this node is its own rightmost descendent if it has no self.right.)

AVLTreeNode.notify_parent(self,new)

In the rotate_right, rotate_left, and remove functions, it is sometimes necessary to change our parent node's reference to point at something else. This is a slightly complicated procedure, since a node doesn't immediately know whether it is its parent's right or left child. This can be determined, however, by using the "is" operator to compare self.parent.left and self.parent.right to self and see which one matches.

Rather than have to repeat this process in multiple places, we restrict it to one method. When notify_parent is called on a node, it checks its parent's left and right references to see which one matches self, and then reassigns that reference to the node in "new".

AVLTreeNode.remove(self,value)

Identify first whether the value we're attempting to remove would exist to the left of, right of, or at this node.

If it would exist to the left or right of this node, our behavior depends on whether or not that subtree exists. If it does, we make and return a recursive call. If it doesn't, the value we're trying to remove isn't present in the tree, and we simply return a reference to the tree's root.

If instead the node we're trying to remove is the *current* one, we have to identify which of the three BST removal cases we're in: zero-child, one-child, or two-child. The process by which we handle these three cases is largely described above, but note that in the two-child case we will want to make use of the AVLTree.rightmost_descendent method and then swap *the value* of that rightmost descendant into this node's self.value before proceeding with removing the *rightmost descendent*. This is far simpler than detaching this node and moving the rightmost descendent into its place.

We must then initiate a check_up, either starting at the old parent of this node (if it was a 0- or 1-child case) or the old parent of the rightmost descendent (in the 2-child case).

Regardless of what path we take through the function, we must ultimately return a reference to the tree's new root.

Like the add method, I recommend that you first implement this method without the check_up call, such that you have a fully-functioning BST before adding the AVL-specific functionality.

AVLTreeNode.height(self)

Recursively calculate and return this node’s height, using our slightly-altered definition of height. The easiest way to go about this is to calculate the height of both subtrees and then return the max of the two heights +1. *However*, note that it is not safe to simply recursively ask self.left and self.right for their heights every time – it is possible that either or both of these child nodes do not exist. If you treat their heights as being -1 in those cases, as we defined above, things will work smoothly.

AVLTreeNode.balance_factor(self)

Calculate and return the balance factor of this node, defined as the height of its right subtree minus the height of its left subtree. Remember that subtrees may be empty.

AVLTreeNode.rotate_right(self)

Perform a right rotation, wherein this node’s left child becomes its parent, and this node takes over that left child’s right subtree as this node’s new left subtree. Note that there are a total of SIX references that must be altered to complete this process.

AVLTreeNode.rotate_left(self)

Perform a left rotation, wherein this node’s right child becomes its parent, and this node takes over that right child’s left subtree as this node’s new right subtree. Note that there are a total of SIX references that must be altered to complete this process.

AVLTreeNode.check_up(self)

Check the balance factor of this node. If it’s -2 or +2, determine whether we’re in a right-right, right-left, left-right, or left-left situation (all described above under “when and how to rotate”), and make the appropriate calls to the rotate_right and rotate_left methods to resolve the issue.

After doing so, if we’re not the root of our own tree, recursively call this method on our parent.