Computer Science 3A

Practical Assignment 10

11 May 2023

Time: 11 May 2023 — 17h00 Marks: 50

Practical assignments must be uploaded to `eve.uj.ac.za` **before** 17h00 in the practical session.

You are **not allowed to collaborate** with any other student.

AVL Trees are a form of a self-balancing binary search tree. The self-balancing nature of the AVL tree allows it to achieve a worst-case runtime proportional to $O(\log n)$. An AVL tree is relatively easy to create. However, it does require you to understand how the restructuring operations operate.

The following properties define AVL trees:

1. **Binary Search Tree Property** — A left child's key is less than its parent's key, which is, in turn, less than the key of its right child.
2. **Height Balance Property** — The heights of two siblings in the tree cannot differ by more than one.

If the above two properties are maintained, the AVL tree is an efficient and easy-to-create search tree. However, if these properties are not maintained, the tree will run proportional to $O(n)$. Your AVL tree will have to use **sentinel nodes** as leaf nodes in the tree.

Keyword-indexed Restaurant Searching using an AVL Tree
You must create an AVL tree that stores restaurant keywords and information and allows users to search for a restaurant using a keyword efficiently. You will have to create some functions for the tree structure to work as expected. You will be required to complete the following functions:

1. `treeSearch(node)` — which will search the AVL tree for a node.
2. `insert(k,v)` — which will insert an item into the AVL tree.
3. `checkTreeBalance(node)` — which will check to see if the tree is balanced and perform a rebalance if necessary.

4. `rebalance(node)` — which will change the structure of the tree to ensure that the **height-balance property** is maintained.

Auxiliary functions have been provided to you that will aid you in constructing your tree. To allow for completeness, the following description of the functions you are required to complete is provided below:

treeSearch

The `treeSearch` function searches for a node in the tree or will end up on an external node where the item should exist in the tree. As you are searching through the tree, you will move left and right depending on the value of the item you are searching for if you end up on a leaf node (sentinel node). Then the item does not exist in the tree.

The following algorithm can be used to search for an item in an AVL tree; this function is declared recursively where you are considering a $node$ and all its children and searching for a key.

1. If the tree size is 0, return the tree's root (the root will be a sentinel node).
2. If the node you are currently considering is external, return the $node$ (the item was not found in the tree).
3. If the key is less then the key found in $node$ then recursively call `treeSearch` on the left child of $node$.
4. If the key is greater then the key found in $node$ then recursively call `treeSearch` on the right child of $node$.
5. If the key is equal then the key found in $node$ then return $node$ (you have found the item).

insert

The insert functions will insert an item into the AVL tree. The function is quite simple to create and uses the `treeSearch` function and the `checkTreeBalance` function:

The following algorithm can be used to insert an item into an AVL tree

1. Search for the node that contains the key in the tree, the result is stored in $node$.
2. If $node$ is not external, then the key exists, and the value should be updated.
3. If $node$ is external then
   (a) Insert the passed key and $value$ into $node$ ($node$ will exist - it is a sentinel node).
   (b) Create new sentinel nodes for the left and right children of $node$.

   (c) Check the balance of *node* to correct the **height-balance property**.
   (d) Increase the size of the number of items stored.

4. Return the inserted node.

checkTreeBalance

The `checkTreeBalance` function ensures that the **height-balance property** has not been violated; it traverses the tree from the current position to the root updating the heights of the nodes as it goes. It is essential to update the heights. Otherwise, the tree will not be properly balanced. This function is defined recursively.

The following algorithm can be used to check the height balance of the AVL tree that considers a passed *node*:

1. If the *node* is the root then update the height to `1 + max(height(node.left()), height(node.right())` and return. *NOTE: watch out for the types for the nodes - use the functions provided.*
2. If the *node* is an external node, set the height to 0 and return.
3. If the *node* is an internal node
   (a) Update the height of *node* to `1 + max(height(node.left()), height(node.right())`.
   (b) Obtain the *sibling* of *node*.
   (c) Check to balance between the *sibling* and *node*.
   (d) If *sibling* and *node* are unbalanced - *NOTE: use the provided function.*
      i. Store the parent of the unbalanced nodes in *zPos*.
      ii. Obtain the node *yPos*, which is the taller child of *zPos* (the child that has the greater height).
      iii. Obtain the node *xPos*, which is the taller child of *yPos* (the child that has the greater height).
      iv. Call the `rebalance` function on *xPos*. The result of this function should be stored in *b*. This function will restructure the tree to restore balance. Once the tree has been rebalanced, the height of the nodes needs to be restored.
      v. Set the height of the left child of *b*. *NOTE: use the provided function.*
      vi. Set the height of the right child of *b*. *NOTE: use the provided function.*
      vii. Set the height *b*. *NOTE: use the provided function.*
      viii. Recursively call `checkTreeBalance` on *b*.
   (e) If *sibling* and *node* are balanced, then recursively call this function on the parent of *node*.

rebalance

The `rebalance` function is used to restructure the tree. This function is coded as a static set of transformations on the tree structure. The tree nodes have to change then you will have to set the left and right children and the parent of the subtrees in all cases. There are four distinct cases for structuring the tree, and this has been outlined in the provided code.

The basic premise is that you need to obtain the nodes x, y, and z, and then redefine them as a, b, and c which is the order x, y, and z would appear in an in-order traversal of the tree (you hard code this and **do not** need to perform an actual in-order traversal. b then becomes the root of the new rebalanced tree a then becomes the left child of b, and c becomes the right child of b. The subtrees are reattached to the tree from left to right in the correct order.

The following algorithm can restructure the tree given the node x.

1. Obtain the node y which is the parent of x.
2. Obtain the node z which is the parent of y.
3. Determine which nodes are a, b, and c based on the four cases outlined in the provided code. This is hard-coded based on whether x is a left or right child of y and if y is a left or right child of z.
4. Determine which nodes are $t1$, $t2$, $t3$ and $t4$ based on the same criteria as defined above.
5. Once you know which nodes are a, b, and c, then:
   (a) p is the parent of z.
   (b) Make b the root of the restructured subtree and set the parent of b to p
   (c) Set the left child of p to b if z was a left child, or set the right child of p to b if z was a right child.
   (d) Set a as the left child of b, and set the parent of a to b.
   (e) Set c as the right child of b, and set the parent of c to b.
   (f) Set $t1$ as the left child of a and $t2$ as the right child of a. Set the parent of $t1$ and $t2$ to a.
   (g) Set $t3$ as the left child of c and $t4$ as the right child of c. Set the parent of $t3$ and $t4$ to c.
   (h) Return b.

You must complete the classes and methods marked by:

```
//TODO: COMPLETE CODE HERE
```

A test class has been provided to test your implementation. You should not add any extra functions to the provided classes, only complete the methods that have been indicated.

The results of your application should be as follows:

```
Building restaurant index
Inserting: The Great Eastern Food Bar
Inserting: Dukes Burgers
Inserting: Parreirinha
Inserting: Thava
Inserting: Villa Bianca
Inserting: Les Delices de France
Inserting: PRON
Inserting: BBQ Qorkshop
Inserting: The Grillhouse
Inserting: Ant Cafe
Testing AVL insert
Testing AVL tree Search
Searching for: grill
Restauarant found:
Grillhouse,The Grillhouse: A well established upmarket steakhouse.
{28.35954,-26.62294}
User search (q to quit):
les
Restauarant found:
Les,Les Delices de France: An upmarket authentic french with attention to detail
{27.93382,-26.16768}
User search (q to quit):
q
Thank you for using restaurant keyword search
```

The following files must be submitted to EVE:

1. *studentnumber*_p10.zip

Marksheet

1. AVLTree: insert [5]
2. AVLTree: checkTreeBalance [10]
3. AVLTree: rebalance [15]
4. AVLTree: treeSearch [5]
5. Compilation and Correct execution. [15]