Protocol

Friedl, Granig, Niziolek

# Description of the functions

**h() - function**

Function for calculating height

* If root is NULL (indicating an empty tree/blatt/leaf), it returns 0.
* Otherwise, it recursively calculates the height of the left tree subfirst and then and right.
* It returns the maximum height of the left and right subtrees, plus 1 to account for the current node.

Runtime analysis:

O(n), n is the number of nodes

**bal() - function**

This function calculates the balance factor of a node in a binary tree.

* It computes the height of the right subtree (rightHeight) and the height of the left subtree (leftHeight) by calling the h() function.
* It returns the difference between rightHeight and leftHeight, which gives the balance factor of the node.

Runtime analysis:

O(n), n is the number of nodes

**printBalance() - function**

This function prints the balance factor of each node in a binary tree and checks for AVL violations (nodes with a balance factor greater than 1 or less than -1).

* It recursively traverses the tree using reverse post-order (visiting the right subtree, then the left subtree, and finally the root node).
* For each node, it prints its balance factor using the bal() function.
* If the balance factor indicates an AVL violation, it increments a counter and prints a message indicating the violation.

Runtime analysis:

O(n), n is the number of nodes

**insert() - function**

This function is for inserting a node in the tree.

* Check if the node is NULL. If that is the case a new node is created.
* If the value of the file is smaller than the value of the current node the function is recursively called again with the left child as the new root.
* If the value of the file is greater than the value of the current node the function is recursively called with the right child as the new root-node.
* At the end the whole tree is returned.

Runtime analysis:

O(n), n is the number of nodes

When used together, these functions provide a way to calculate and print the balance factors of all nodes in a binary tree and identify any AVL violations present in the tree. The printBalance() function is the main driver that utilizes the bal() function to check the balance factor of each node and outputs the results accordingly.

**findMin() - function**

Function for searching for the minimum value of the tree.

* It checks if the root is NULL and if it is, it prints a proper message, that the tree is empty.
* After that, it checks if the tree has a left child. If not, then the current root has the minimum value of the whole tree.
* If the root has a left child, the function is called again with the left child as a root parameter.
* This continues until the root has no left child anymore.

Runtime analysis:

O(n), n is the number of nodes in the left subtree

**findMax() - function**

This function is for searching for the maximum value of the tree and it is very similar to the findMin()-function.

* It checks if the root is NULL and if it is, it prints a proper message, that the tree is empty.
* After that, it checks if the tree has a right child. If not, then the current root has the maximum value of the whole tree.
* If the root has a right child, the function is called again with the right child as a root parameter.
* This continues until the root has no right child anymore.

Runtime analysis:

O(n), n is the number of nodes in the right subtree

**calculateSumAndCount() - function**

This function calculates the sum of all the values in the tree and the number of nodes.

* We start at the root-node again. If it is empty the function will stop because there is no tree to check.
* If the root-node is not NULL we add the value of the node to our sum-variable and increment the counter by one.
* After that, the whole left tree will be run through.
* If there is nothing left in the left subtree, it will jump into the right subtree and run it through.
* After it reaches the end of the right subtree, the whole function will stop.

Runtime analysis:

O(n), n is the number of nodes in the tree

**calculateAverage() - function (no recursive function)**

This function calculates the average value of all the values of the tree based on the result of the calculateSumAndCount() - function.

* The needed variables are initialized and the calculateSumAndCount-function is called with the root and the initialized variables (as pointer) as parameters.
* Because the division by zero is not allowed, this potential error is checked before actually calculating.
* If everything is fine, the function returns the average value by calculating it (sum divided by number of nodes).

Runtime analysis:

O(n), n is the number of nodes in the tree (because of the calculateSumAndCount()-function-call)

**search() - function**

This function searches for a specific value in the tree.

* First we check if the root-node is NULL. If so (what means that there is no tree or the value we search for is not found) we return NULL.
* Then we check if the value we search for is equal to the value of the current node. If so, we save the value in the path array on the index of depth (the starting index from the main-function is 0) and we return the current node.
* If this is not the case we check if the value we search for is smaller than the value of the current node. If this is the case, we save the value of the current node in the array and recursively do the same thing with the left child of the current node and a depth of (depth)+1.
* And if this is not the case we check if the value we search for is greater then the value of the current node which would mean, that our value we search for (if it exists in the tree) is in the right subtree. We save the value of our current node in the array again and recursively check the tree again with the right child as the root-node and a depth of (depth)+1.

Runtime analysis:

O(n), n is the number of nodes in a subtree

**isSubtree()-function**

In the process of checking whether a specific subtree is part of a particular main tree, both trees are first read separately from a text file. They are both constructed as trees, and only then are they compared. It is first checked whether the subtree or the main root is NULL. By using recursive queries, it is first checked if the current root matches the subtrees root. If that’s the case, the subtree and main tree are further compared, and if they are the same until the subtree runs out of characters, it returns 1, which translates to the subtree is included in the main tree. If the root of the main tree and the subtree however don’t match, the main tree and subtree get compared through recursvie queries until they eventually match (in which case the subtree is contained in the main tree) or until the subtree or maintree are finished (in which case the subtree is longer than the main tree or has a different structure, indicating it is not contained).

The complexity of reading the text files and constructing the trees both depend on the size of the trees and involves reading each element from the file once. Reading the files would be O(n) with n as the number of integers in the text file. For each element read from the file, a corresponding node is created and inserted into the tree. Since each element is processed only once during the tree construction, this operation is also O(n).

The complexity of the comparison process involves traversing each node of the main tree and recursively checking if it matches the root of the subtree. This process continues until either the subtree is fully matched or the main tree is exhausted. In the worst-case scenario, where the subtree is not found in the main tree, this operation traverses all nodes of the main tree. Therefore, the time complexity for this operation is O(m \* n), where m is the number of nodes in the main tree and n is the number of nodes in the subtree. Overall, the time complexity is O(n + m \* n).

**readTreeFromFile() - function**

This function is for opening a file and reading the integer-values out of it to create a tree.

* First the file will be opened in read-mode and we check if there were any error while opening the file.
* After that the values will be read by the while-loop. It loops through the file until it reaches the end and for each entry the insert-function is called.
* After everything is successfully inserted into the tree, the file will be closed and the root of the tree will be returned.

Runtime analysis:

O(n) for the loop and O(n) for the inserting so O(n2)

**freeAll() - function**

We use this function to free the allocated storage.

* We check if root is NULL. If that is the case the tree is already empty and we end the function.
* Else we go recursively through the left subtree and free every node. After that we go through the right subtree and free everything.
* At the end we free the root-node.

Runtime analysis:

O(n), n is the number of nodes