CCDSALG Term 3, AY 2020 – 2021

Project 3 – Binary Search Trees

Section	Names	Task 1	Task 2
S13	Tolentino, John Enrico	X	X
	Cruz, Airon John	X	X
	Hernandez, Pierre Vincent	X	X

Fill this part with your section and names. For the tasks, put an X mark if you have performed the specified task. Please refer to the project specifications for the tasks.

IMPLEMENTATION DISCUSSION

Explain the implementation and indicate the worst-case time complexity of the functions.

Task 1 – Implement Binary Search Tree (BST) operations

Function	Big-oh	Implementation
`create()`	$O(1)$	This function creates a binary search tree by allocating a memory with a size of the struct "bst" and returning the first byte of that allocated memory. It has a constant time complexity because the function only has a return statement in it.
`insert()`	$O(h)$, h = tree height	This function creates two sNode pointers one will be the responsible for being the basis for the parent of a subtree to which whether the inserted node will be the left or right child (variable parent), and the other one will be used to move down to each node in the tree and it will also store the newly created node to be attached to the parent of the subtree (variable tmp). The 'tmp' will get the address of the tree. While tmp hasn't passed the leaf node then it will check whether the data to be inserted is greater than or less than. If the data is greater than, then it will go to the right child of its current node, otherwise it will go to the left child. Before the 'tmp' moves to its next node, 'parent' will take the current address of the current node that is stored in 'tmp', then 'tmp' will move to its next child node. If it passes beyond the leaf node of its path (NULL), then it will exit the loop. From there, a new node will be created in which it will store the data to be inserted. The 'tmp' will temporarily store the address of this new node. Since 'parent' takes the address of the current node of 'tmp' before it moves, then 'parent' will store the leaf node, which would be the

		parent node of the new node that is going to be inserted in the tree. From there, it will check whether the data to be inserted is greater than or less than the data in the 'parent'. If greater than, then the address of the new node containing the data to be inserted will be attached or inserted as the right child of the current leaf node that is stored in 'parent', otherwise it will be attached or inserted as the right child of the current leaf node. It is observable that the function moves down a node from the root. From this we can conclude that the time complexity would be O(h), where h is the tree height. This is because the tree height is the distance between the root to its most distant leaf node, and when inserting a new node it would be attached as a child node of a leaf node. To prove this we calculated the frequency count of the function. The total frequency count we got is 6n + 20. The n represents the total number of nodes where the function will pass through in a single path going to the desired leaf node. So the worst-case scenario of this function is to attach a new node at the end of the longest path down the tree, which is the most distant leaf node from the root. From here we can deduce that the height of the tree is the worst-case scenario since the definition of tree height is the distance from the root node to its most distant leaf node.
`search()`	$O(h)$, h = tree height	This function's implementation is identical to the insert() function. It will also use a local variable that will be used to move down the tree. It will continue to move down until it either finds the node which contains the data that is being searched or it has passed beyond the leaf node. Since it has an identical implementation to the insert() function then the function moves down a node in a specific path which means they have the same behavior. So the worst-case scenario would be the longest path, or the data is stored in the most distant node from the root of the tree. Therefore we can conclude that the height of the tree is the worst-case scenario; same as the insert() function.

`inorder()`	$O(n)$	This binary search tree (bst) operation is a recursive function, and within its implementation it has a recursive case that traverses down a path until the pointer reaches a leaf node which makes it equal to NULL. Firstly, this function traverses the left subtree of the current node pointer (current node's pLeft) by calling the recursive function, then it prints the stored data in the current node if there is no more left node to visit; finally, it traverses the current node's right subtree (current node's pRight) using the recursive function as well. This has a worst-case time complexity of O(n), (where n is the number of nodes) since this function visits every node only once; hence, the complexity highly depends on the tree's total number of nodes.
`preorder()`	$O(n)$	This function is a recursive function and its recursive case is while the pointer that traverses down a certain path in a subtree hasn't reached beyond the leaf node (pointer not equal to NULL). This will first print the data in the current node then traverse to the left and right subtrees of the current node where the pointer is pointing to. This function's behavior is linear since it prints and moves to each node of the tree. The frequency count of this would depend on the number of nodes a binary tree has.
`postorder()`	$O(n)$	The postorder function recursively traverses down a path until the pPointer parameter becomes null (as checked by the function). From the root node, the function traverses left until the pPointer parameter becomes NULL at which point it will traverse right to do the same thing. When both recursive cases have been executed, the function then prints the value of the node starting at the current node. The function takes O(n) time to complete the function as it checks every node in the binary tree.

`maximum()`	$O(h)$ h = height of the tree	This function is implemented through searching the rightmost node of a tree inside a while loop statement. The loop will continuously pass the current node's pRight to sNode temp or the pointer heading towards where its right node is situated until it returns a NULL pointer value. The function returns the pointer to the node whose right node is NULL. Moreover, this function has a worst-case time complexity of O(h), since this searches down every right child node in the bst. Thus, h is equal to the height of the tree which this function's worst-case is proportional to.
`minimum()`	$O(h)$ h = height of the tree	The implementation of this function is basically just passing the left node's pointer of the current node inside a loop until it reaches the left-most node of the bst which will result in a NULL value of the current node's pLeft. This function returns the sNode pointer pointing towards the aforementioned leftmost node. The minimum function has a worst-case time complexity of O(h), where h corresponds to the height of the tree, since it searches down for the node whose left node is NULL which is proportional to the tree's height.
`parent()`	$O(h)$, h = tree height	This function's implementation is identical to the insert() and search() function, which means it also has the same behavior as those two said functions. First it will search for the data in the tree. While it is searching and moving down the tree, the 'parent' variable takes the address of the current node being checked that is stored in the 'tmp' variable. After storing it, the 'tmp' will move either to the left or right child depending on whether the data that the function is looking for is less than or greater than. Since 'parent' stores the node before the next node, then when the next node contains the data that the function is looking for in the tree it can easily find and return its parent node by returning that value of 'parent'. Since it has the same behavior as the said functions then it has the same time complexity. Its worst case scenario would be taking the longest path from the root, which is equivalent to the height of the binary tree itself.

`successor()`	$O(h)$, h = tree height	The successor function takes a binary tree and an integer value as parameters. The function first checks if a node within the tree that contains the integer parameter exists using the search function then stores its pointer value to the key variable. Given that the pointer for the key variable is not null, two conditions are then checked: whether the pRight pointer for the key node is NULL or not. Should the second condition be satisfied, the minimum function is called, passing the pointer of the key node's right subtree as the parameter, returning the value of its leftmost leaf node. Otherwise, it returns the lowest ancestor after looping through the parents of the key node. Initially, the parent of the key node is assigned to the ancestor variable after which a loop of assigning the ancestor variable to the key node and passing the ancestor to the parent function is done while the ancestor is not NULL and the left subtree of the ancestor is null or the data of the ancestor's left child node is not equal to the value of the key node. It only takes O(h), where the h is tree height, time to complete the function as the function only traverses the tree up or down in one path.
`predecessor()`	$O(h)$, h = tree height	The predecessor function has an almost identical implementation as the successor function. Similarly to the successor function, it takes O(h) time to complete the function given that the function only traverses the tree up or down in one path. As with the successor function, it checks for two conditions. However the similarities diverge when it checks whether the pLeft pointer of the key node is NULL or not, calling the maximum function with the pointer of the key node's left subtree as the parameter. Similarly to the successor function, it also returns the lowest ancestor after looping through the parents of the key nodes. While it still checks whether the ancestor is not NULL while looping, it checks whether the right

		subtree is null or the data of the ancestor's left child node is not equal to the value of the key node instead.