CS-202

Fundamental Structures of Computer Science II

Section: 1

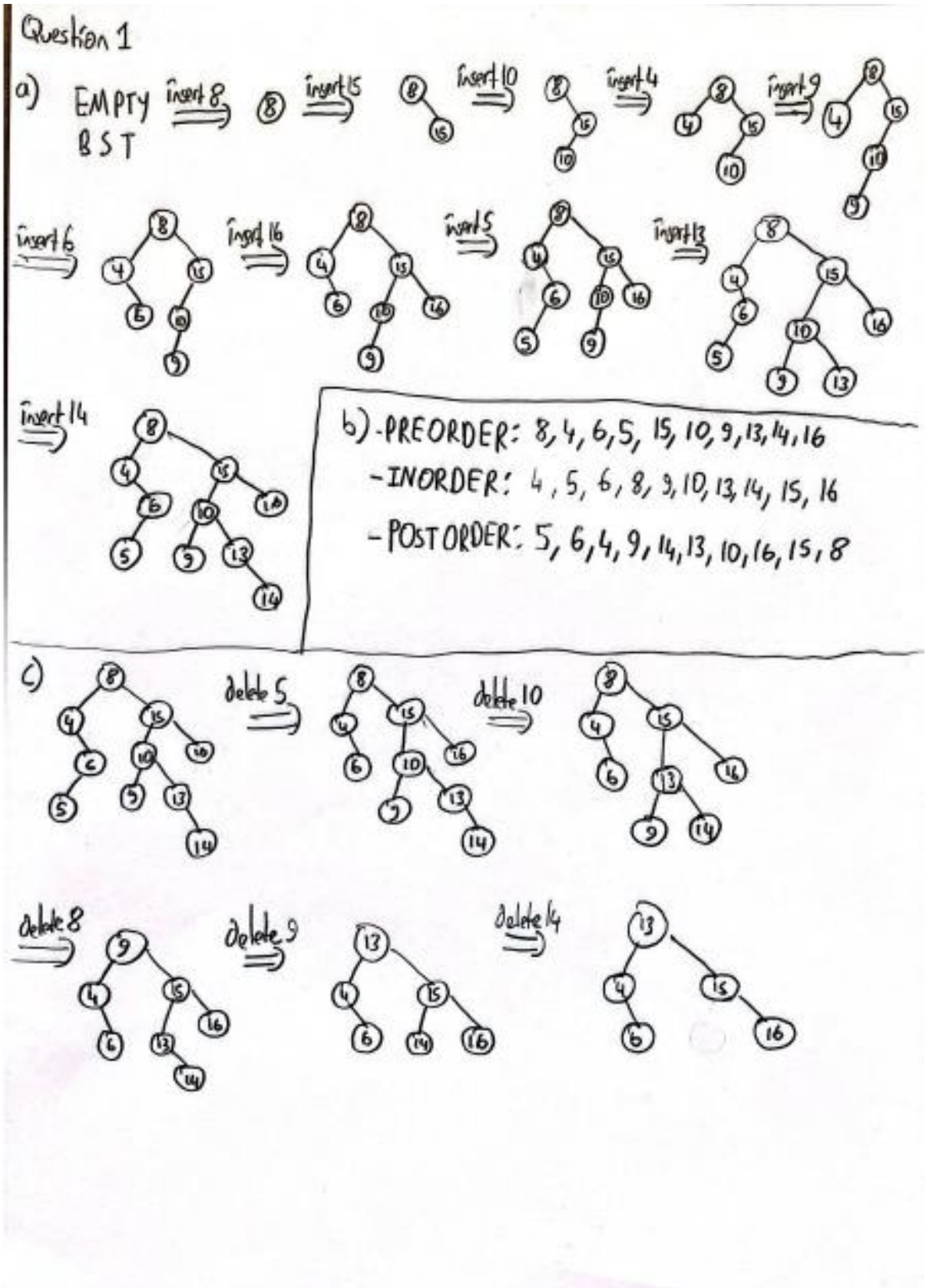
Homework 2

Mehmet Hasat Serinkan

21901649

02.07.2022

QUESTION 1


Question 1
a)
EMPTY BST
insert 8
insert 15
insert 10
insert 4
insert 9
insert 6
insert 16
insert 5
insert 13
insert 14
b) -PREORDER: 8, 4, 6, 5, 15, 10, 9, 13, 14, 16
-INORDER: 4, 5, 6, 8, 9, 10, 13, 14, 15, 16
-POSTORDER: 5, 6, 4, 9, 14, 13, 10, 16, 15, 8
c)
delete 5
delete 10
delete 8
delete 9
delete 14

QUESTION 3

- ## void BST::insertItem(int key)

Firstly, the insertion function is performed by finding the appropriate position in the binary search tree. The finding statement will made by binary searching. When the function finds a suitable place for the node that will be inserted, it creates the node with no children.

In order to insert a node, the function must traverse the number of nodes on the longest path of the binary tree. It also equals to height of the binary tree. Since the height of a binary search tree can differ between the log(N) and N (N means the total number of nodes), the function will occur O(log(N)) for the best and average case, and O(N) for the worst case.

Recurrence relation for worst case is T(N) = T(N-1) + O(1) and T(1)=1

Recurrence relation for the best and average case is T(N) = T($\frac{N}{2}$) + O(1) and T(1) = 1

Space complexity will be O(N) because the function will store the tree since it is a recursive function.

Results are:

O(N) for the worst case,

O(log(N)) for the best and average case,

Space complexity will be O(N).

- ## void BST::deleteItem(int key)

There are 3 possible cases: deleting a node with no children, deleting a node with one child, deleting a node with two children. For the first case, we delete the node. In the second case, we attach the node's child to the node's parent and we delete the node. The third case requires a more complex way. First, we get in order successor of the node, then swap the values. After all, we delete the swapped node.

The case number does not affect the result of space complexity or time complexity. In every case, the function reaches a leaf node. To find the node that will be deleted, function uses binary search. Therefore, it will process for the height of the tree. Since the minimum value of

tree's height is log(N) and the maximum value for the height is N, the function will occur O(log(N)) for the best and average case, and O(N) for the worst case.

Recurrence relation for the worst case is T(N) = T(N-1) + O(1) and T(1)=1

Recurrence relation for the best and average case is T(N) = T($\frac{N}{2}$) + O(1) and T(1) = 1

Space complexity will be O(N) because the function will store the tree since it is a recursive function.

Results are:

O(N) for the worst case,

O(log(N)) for the best and average case,

Space complexity will be O(N).

- BSTNode* retrieveItem(int key)

The function takes a key value and returns the node that has the key value. It will find the node by binary search.

The minimum value of the tree's height is log(N) and the maximum value for the height is N; Hence, the function will occur O(log(N)) for the best and average case, and O(N) for the worst case.

Recurrence relation for worst case is T(N) = T(N-1) + O(1) and T(1)=1

Recurrence relation for the best and average case is T(N) = T($\frac{N}{2}$) + O(1) and T(1) = 1

Space complexity will be O(N) because the function will store the tree since it is a recursive function.

Results are:

O(N) for worst the case,

O(log(N)) for the best and average case,

Space complexity will be O(N).

- int* inorderTraversal(int& length)

This function returns an array of in order traversal of the tree. It must pass all the nodes. Therefore, the best, average and worst case for this function is O(N). Space complexity will be the number of nodes which means O(N).

Results are:

O(N) for the worst case,

O(N) for best and average case,

Space complexity will be O(N).

- bool hasSequence(int* seq, int length)

This function checks whether a sorted array's elements are sub-array of in order traversal of the tree. However, it does not traverse redundant nodes but only the nodes that are required for the check.

The function takes the largest and the smallest values from the array. Then it compares the nodes whose items are in the range of between the smallest and the largest value of the array and elements in the sequence. If a mismatch occurs, the rest of the nodes will be not called. When it reads and matches all the values of the sequence with the tree, the function returns true. If not, it returns false.

The complexity for the average case of this function is O(N). If the given array's length is equal to the tree's total nodes, then the function would traverse all the nodes. Therefore, the worst case will be O(N). Space complexity will be O(N).

Results are:

O(N) for the worst case,

O(N) for the average case,

Space complexity will be O(N).

- int countNodesBelowLevel(int level)

The function calculates the number of nodes whose level is greater than or equal to the given level. To find the total number of nodes, the function adds all nodes in a level from the given level to the max level. In order to find the total nodes of given level, function traverses the tree until it reaches the level by a helper method. After that, it sums up all the nodes. Helper method's best, average and worst case is O(n) since it traverses that tree. The main function calls this function for (maxLevel – level). Therefore, the complexity is proportional to O(N^2). Space complexity will be O(N) because the function stores the tree recursively.

Results are:

O(N^2) for worst case,

O(N^2) for average case,

Space complexity will be O(N).

- BST* merge(const BST& tree1, const BST& tree2)

This function takes two BSTs and merges them by creating a new balanced tree containing all the values of these two BSTs. Now, assume that the total number of nodes of tree1 is N and total number of nodes of tree2 is M. Firstly, function creates two copies of tree1 and tree2 since they are const and only usable with const functions. Then takes in order sorted arrays of these functions. After that, two arrays are merged and the tree is created by a helper method with this merged array. Helper function takes middle elements of array, creates a node and attaches them into tree recursively.

Since we create a balanced binary search tree, creating a node will be log(M+N). There are N+M elements, therefore, creating a tree will be (N+M)log(M+N). Space complexity will be proportional to N+M.

Results are:

O((N+M)log(M+N)) for the worst case,

O((N+M)log(M+N)) for the average case,

Space complexity will be O(N+M).