Date: 23/03/2019

Title: Binary Search Trees

Author: Zeynep Cankara

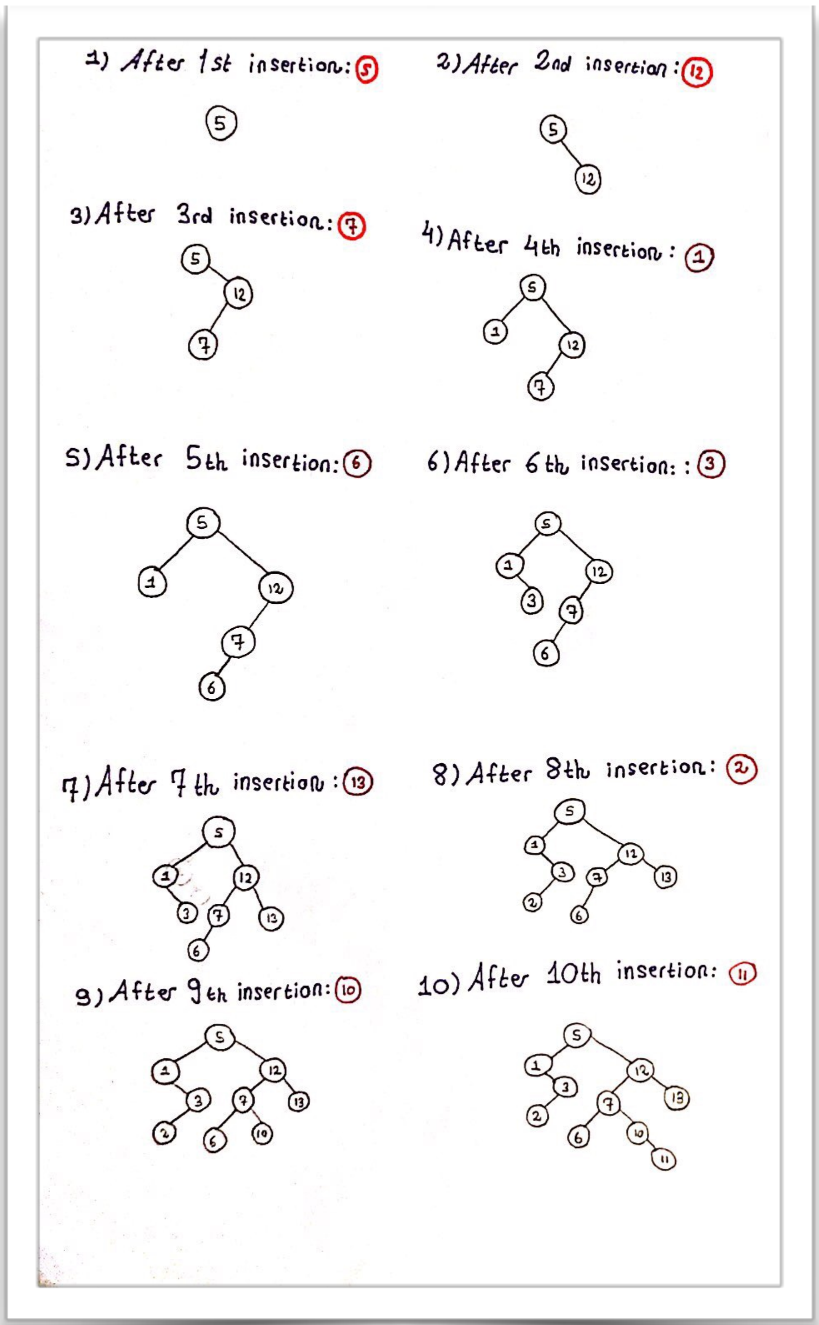
ID: 21703381

Section: 2

Assignment: 2

Description: My Solutions for Question1 and 3.

Question 1: Part(a)


1) After 1st insertion: 5
2) After 2nd insertion: 12
3) After 3rd insertion: 7
4) After 4th insertion: 1
5) After 5th insertion: 6
6) After 6th insertion: : 3
7) After 7th insertion: 13
8) After 8th insertion: 2
9) After 9th insertion: 10
10) After 10th insertion: 11

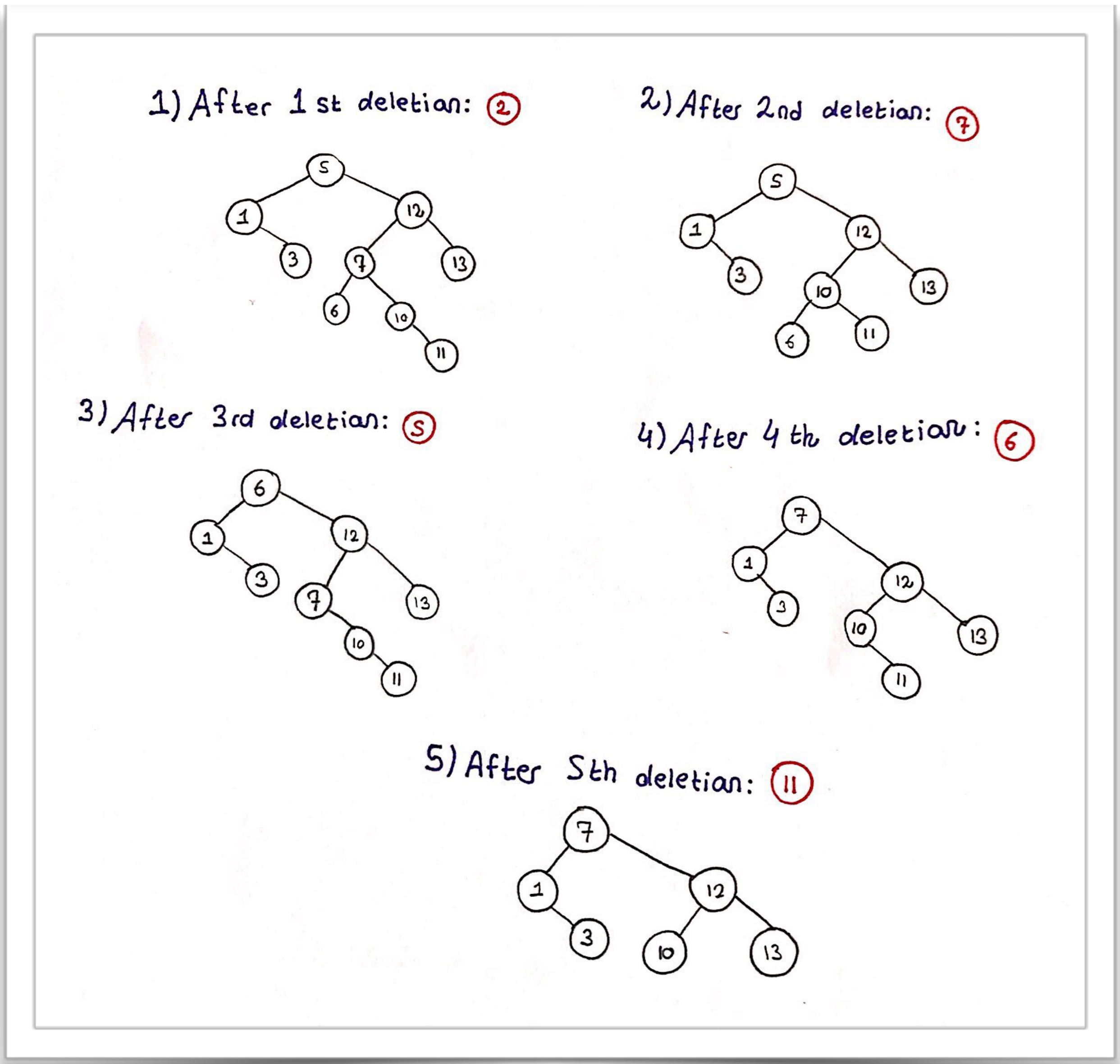
Question 1: Part(b)

Preorder Traversal: 5, 1, 3, 2, 12, 7, 6, 10, 11, 13

Inorder Traversal: 1, 2, 3, 5, 6, 7, 10, 11, 12, 13

Postorder Traversal: 2, 3, 1, 6, 11, 10, 7, 12, 13, 12, 5

Question 1: Part(c)



Question 3: Analyzing the Question 2 Tree Algorithms

3.1) `void BST::insertItem(int key);`

The insertion performed by first finding the insertion position within the binary search tree. Rest of the procedure is just creating the node and arranging child pointers. In order to reach an insertion position which will be at left or right child of a leaf. We must traverse number of nodes on the longest path of the binary search tree which also equals to the max height of the tree. If the data that we are creating binary search tree from is random, our binary search tree will have max height of $O(logn)$ in both best-case (complete & height-balanced) and in the average case. If our data is sorted worst-case max-height occurs which is $O(n)$. Space complexity will be number of nodes stored which is $O(n)$.

Thus we arrived at the conclusion that:

Best Case Insertion: $O(logn)$

Average Case Insertion: $O(logn)$

Worst Case Insertion: $O(n)$

Space Complexity: $O(n)$

3.2) `void BST::deleteItem(int key);`

There are 3 cases when we want to delete a node. Either node we want to delete has no children. Has only one child or has both children. It won't affect the result which one of those 3 cases is our interest. In every case we always perform operation which requires reaching at a leaf node. We know that binary search trees can have minimum height of $O(logn)$ given it is a height balanced complete binary search tree. If it is not then we can suggest that in the worst case data sorted thus we must traverse all nodes in order to reach a leaf node to perform deletion. The resulting worst case deletion will be $O(n)$. Space complexity will be the nodes stored in binary search tree which is $O(n)$ after deletion we will have $n - 1$ nodes which does affect asymptotic notation $O(n)$.

Thus we arrived at the conclusion that:

Best Case Deletion: $O(logn)$

Average Case Deletion: $O(logn)$

Worst Case Deletion: $O(n)$

Space Complexity: $O(n)$

3.3) `BSTNode* BST::retrieveItem(int key);`

As discussed in 3.1 (insert item) for complete height balanced binary search tree we can take the longest path which is from root to leaf in $O(logn)$. Thus, if we know our data is random it is theoretically fine to assume retrieving an item will take $O(logn)$ on average. If the data is sorted and the item with the given key we want to reach is a leaf node; worst case scenario happens we traverse all nodes $O(n)$. If the root is same as the item with the key we want to retrieve best case happens taking $O(1)$. Retrieval does not require space manipulations. The complexity of space is still number of nodes stored in binary search tree $O(n)$.

Best Case Retrieval: $O(1)$

Average Case Retrieval: $O(logn)$

Worst Case Retrieval: $O(n)$

Space Complexity: $O(n)$

3.4) `int* BST::inorderTraversal(int& length);`

Purpose of traversing the BST is to reach all nodes in the Binary Search Tree (BST). Therefore worst-case, average-case, best-case time complexities will be $O(n)$.

Space complexity will be the number of nodes BST contains which is $O(n)$.

3.5) `bool BST::containsSequence(int* seq, int length);`

In the worst case, sequence will contains all nodes of the Binary Search Tree (BST) in sorted order. Therefore to perform the check all nodes of BST will be visited which is $O(n)$. In average case sequence data will contain random numbers requires both traversals of the subtrees of the given root node which is again $O(n)$. Space complexity will be number of nodes contained in the sequence summed up with number of items in the sequence. In the worst case, sequence contains number of items equals to the number of nodes in the BST which is n. Thus space complexity is $2n$ which is $O(n)$.

Average Case: $O(n)$

Worst Case: $O(n)$

Space Complexity: $O(n)$

3.6) `int BST::countNodesDeeperThan(int level);`

The function traverses the tree in in-order fashion. Gets level of the node. Level depends on the number of nodes between the path root to the node. In order to find level of a node second traversal from the root performed by using `getLevel(BSTNode *&node, int targetKey, int &nodeLevel)` helper function. In worst case we are interested with a level which equals maximum height of the BST or a level which does not exist. Thus worst case time complexity is $O(n^2)$. If we assume data is random the average case for BST will still be $O(n^2)$ due to traversals. The space complexity equals number of nodes stored in BST which is $O(n)$.

Average Case: $O(n^2)$

Worst Case: $O(n^2)$

Space Complexity: $O(n)$

3.7) `int BST::maxBalancedHeight();`

A tree is balanced if both subtrees are balanced and difference between subtrees do not exceed 1. Thinking recursively we can get height of both left and right subtree which demands traversing the subtrees $O(n)$. After we know the heights of the subtrees we can check whether they differ more than 1 level or not. If they differ more than 1 level get minimum of the subtree height. If not get the maximum of the subtree height. Overall process takes only 1 traversal from root to the leaves. Thus it is <u>$O(n)$ in both average and in both worst-case</u>. <u>Space complexity same with number of nodes in BST: $O(n)$</u>