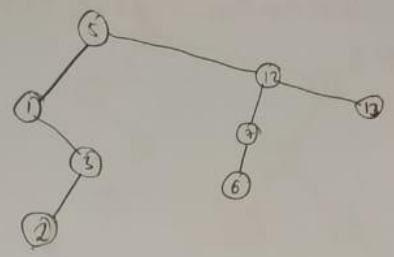
Khasmamad Shabanovi

21701333

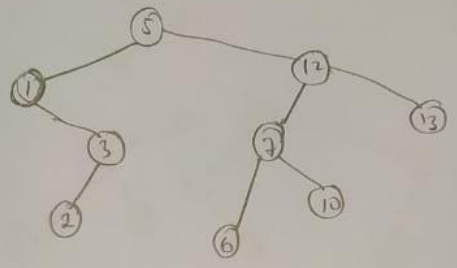
Section 2

CS202 HW2

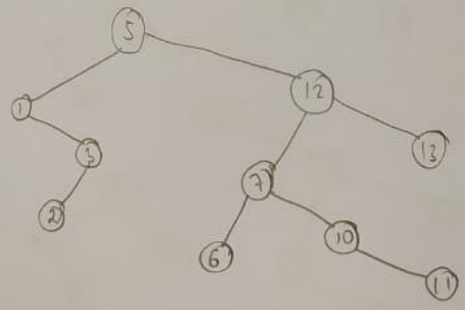
Inserting 2:



Inserting 10:



Inserting 11:



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

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

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

Khasmamad Shabanov 2170 1333 Section.2

CS202 HW 2

Q 1

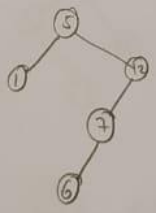
a) Inserting 5:

Inserting 12:

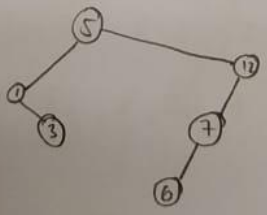
Inserting 7:

Inserting 1:

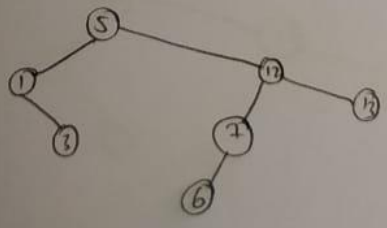
Inserting 6:



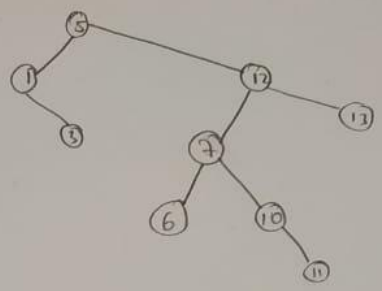
Inserting 3:



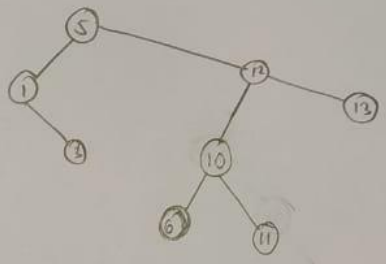
Inserting 13:



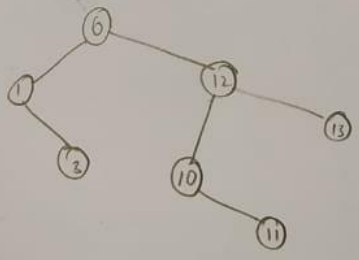
c) Deleting 2:



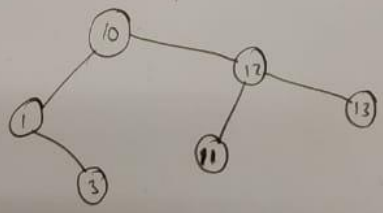
Deleting 7:



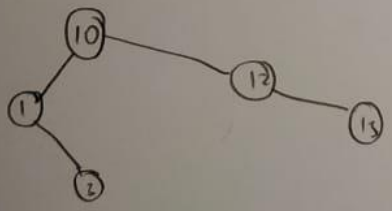
Deleting 5:



Deleting 6:



Deleting 11:



Question 3.

void BST::inserItem(int key)

Time complexity:

Worst-case: $O(N)$ Average-case: $O(\log_2 N)$

The worst case might happen if the nodes are connected in linear order, which would possibly require to traverse all the nodes in the tree (exactly all the nodes if the inserted item would increase the height of the tree). In other words, each recursive call would decrease the number of elements to be visited by only 1:

$T(N) = T(N-1) + O(1)$ and $T(1) = 1$

$T(N) = T(N-2) + 2O(1)$

$T(N) = T(N-3) + 3O(1)$

…..

$T(N) = T\big(N-(N-1)\big) + (N-1)O(1) = T(1) + (N-1)O(1) = NO(1) = O(N)$

In the average-case, however, we assume that each recursive call decreases the number of nodes to be visited by half:

$T(N) = T\left(\frac{N}{2}\right) + O(1)$ and $T(1) = 1$

$T(N) = T\left(\frac{N}{2^2}\right) + 2O(1)$

…..

$T(N) = T\left(\frac{N}{2^k}\right) + kO(1)$

$T(N) = T\left(\frac{N}{2^{\log_2 N}}\right) + \log_2 NO(1) = T(1) + \log_2 N\, O(1) = O(\log_2 N)$

Space complexity: $O(N)$ to store the tree itself. Auxiliary space used is $O(1)$ to store the new node.

Time and space complexities of delete and retrieve algorithms are similar to this one.

int* BST::inorderTraversal(int &length)

Time complexity:

Average case and worst-case: $O(N)$

Traversals, in general, requires visiting and processing all the nodes in a given tree. In my case, I traverse the tree two times: to calculate the number of nodes in the tree, and to add the node

items into an array. The time complexity of both traversals is derived as follows (I assume uniform and random distribution):

$T(N) = T\left(\frac{N}{2}\right) + 1 + T\left(\frac{N}{2}\right) = 2T\left(\frac{N}{2}\right) + O(1)$ and $T(1) = 1$

$T(N) = 4T\left(\frac{N}{4}\right) + 3O(1)$

$T(N) = 8T\left(\frac{N}{8}\right) + 7O(1)$

…..

$T(N) = 2^k T\left(\frac{N}{2^k}\right) + (2^k - 1)O(1)$

$T(N) = 2^{\log_2 N} T\left(\frac{N}{2^{\log_2 N}}\right) + (2^{\log_2 N} - 1)O(1) = N + (N-1)O(1) = O(N)$

Space complexity: $O(N)$. Auxiliary space used is also $O(N)$ to store the array.

bool BST::containsSequence(int *seq, int length)

Implementation: the function makes an implicit call to a helper function with the following prototype:

void BST::containsSequence(BSTNode *treePtr, int *seq, int length, int &i, int low, int high, bool &contains)

Where low and high are the minimum and maximum valued elements in the given sequence, respectively, i is the index of the element of the array to be processed, and contains is a flag which becomes a return value for the caller function. The implementation of the helper function is similar to inorder traversal. It first finds all the elements in the tree whose value is in the range [low, high] and compares them with the respective elements in the sequence. If a mismatch is found contains becomes false no further calls are maid. Otherwise, contains remains true and caller function returns true.

Time complexity:

Worst-case: $O(N)$ Average-case: $O(N)$

The derivation of the time complexity of this function is similar to inorder traversal.

Space complexity:

$O(N)$ and it uses $O(1)$ auxiliary space.

int BST::countNodesDeeperThan(int level)

Implementation: the function makes an implicit call to a helper function with the following prototype:

void BST::countNodesDeeperThan(BSTNode *treePtr, int level, int &currentLevel, int &count)

where, currentLevel holds the level of the current node and count holds the count of the nodes with level deep enough. The helper function performs an inorder traversal and updates count variable if necessary. currentLevel variable is increased before each recursive call to a subtree and decreased afterwards.

Time complexity:

Worst-case and Average-case: $O(N)$

The function traverses all the nodes in a similar fashion to inorder traversal. Thus, the derivation of the time complexity is similar to inorder traversal.

Space complexity:

$O(N)$ and it uses $O(1)$ auxiliary space.

Int BST::maxBalancedHeight()

Implementation:

This function uses a helper function, which takes a BSTNode pointer as a parameter, as well. The helper function performs a postorder traversal. First, it calculates the max balanced height of the left and right subtrees recursively. Then it compares the heights of left and right subtrees. If they are equal function returns 1 + height of the left (or right) subtree (1 for the current node itself). Otherwise, it returns 1+1 + min (height of left, height of right): since the difference in the heights of the subtrees cannot be more than one, we find the shortest subtree and add 1, then we add another 1 for the current node itself.

Time complexity:

Worst-case and Average-case: $O(N)$

The function traverses all the nodes in a similar fashion to postorder traversal. The derivation of the time complexity is similar to inorder traversal.

Space complexity:

$O(N)$ and it uses $O(1)$ auxiliary space.