Title: Trees

Author: Tolga Han Arslan

ID: 22003061

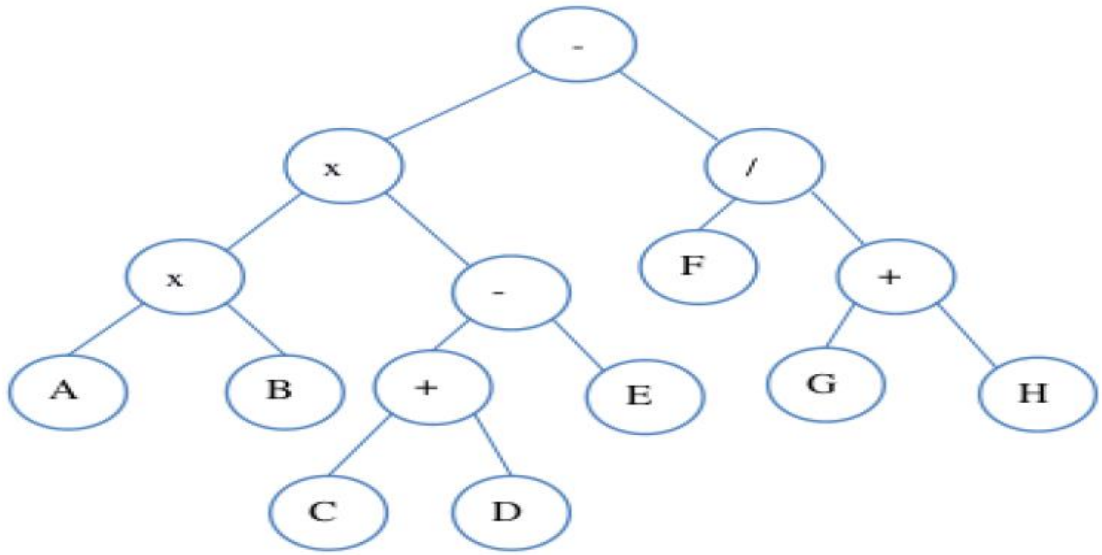
Section: 1

Assignment: 2

Description: HW2 Report

Question 1:

Give the prefix, infix, and postfix expressions obtained by preorder, inorder, and postorder traversals, respectively, for the expression tree below:



Question 1

Preorder Traversal (Prefix) = $r \rightarrow T_L \rightarrow T_R$

$- x x A B - + C D E / F + G H$

Inorder Traversal (Infix): $T_L \rightarrow r \rightarrow T_R$

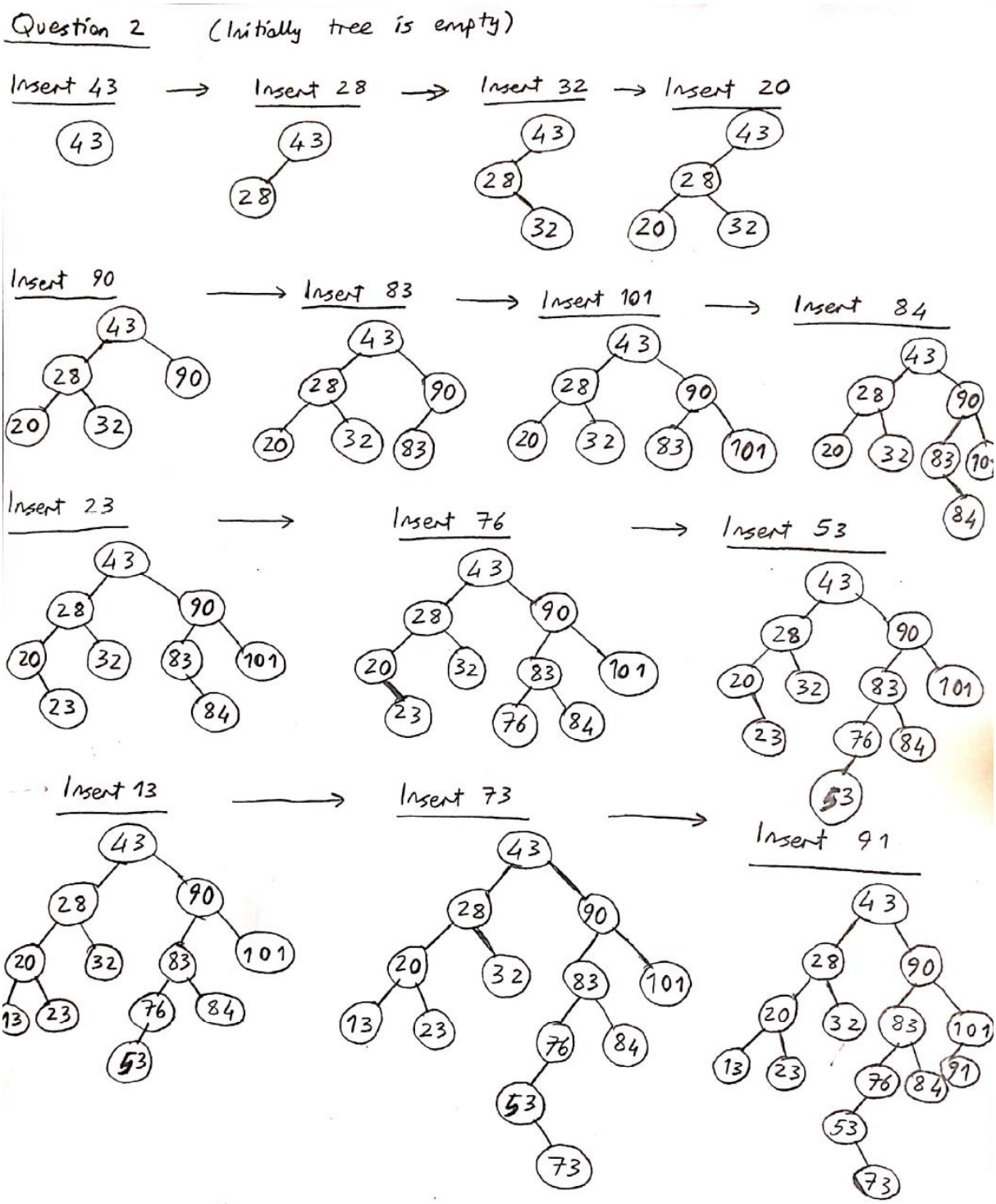
$A x B x C + D - E - F / G + H$

Postorder Traversal (Postfix): $T_L \rightarrow T_R \rightarrow r$

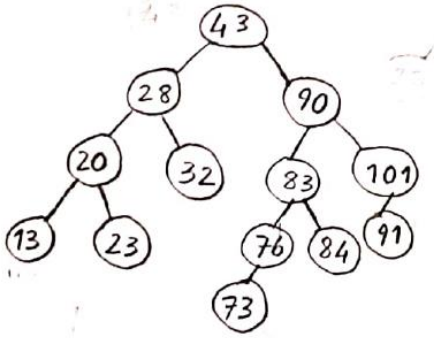
$A B x C D + E - x F G H + / -$

Question 2:

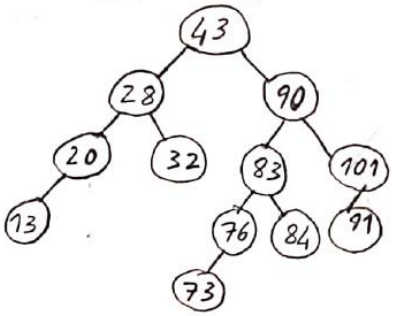
Draw the initially empty Binary Search Tree after operations as follows (show all intermediate steps):
insert 43, 28, 32, 20, 90, 83, 101, 84, 23, 76, 53, 13, 73, 91; then delete 53, 23, 43.



Delete 53 (One right child) → Delete 23 (No child)

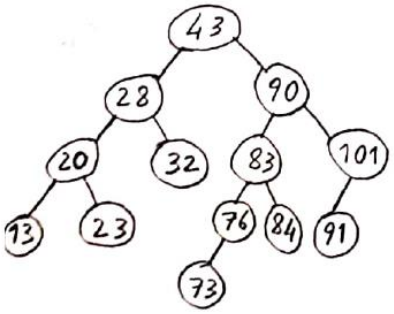


53 has only one right child (73).
73 replaces 53.

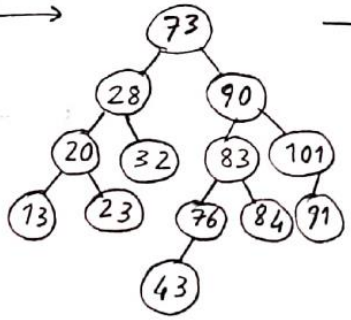


23 has no child (leaf).
No replacement needed.

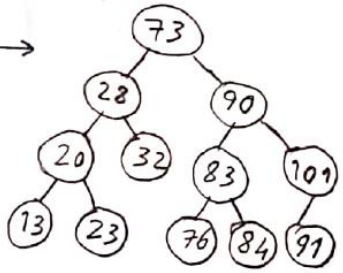
Delete 43 (Two child)



43 has two children.
Inorder successor = 73

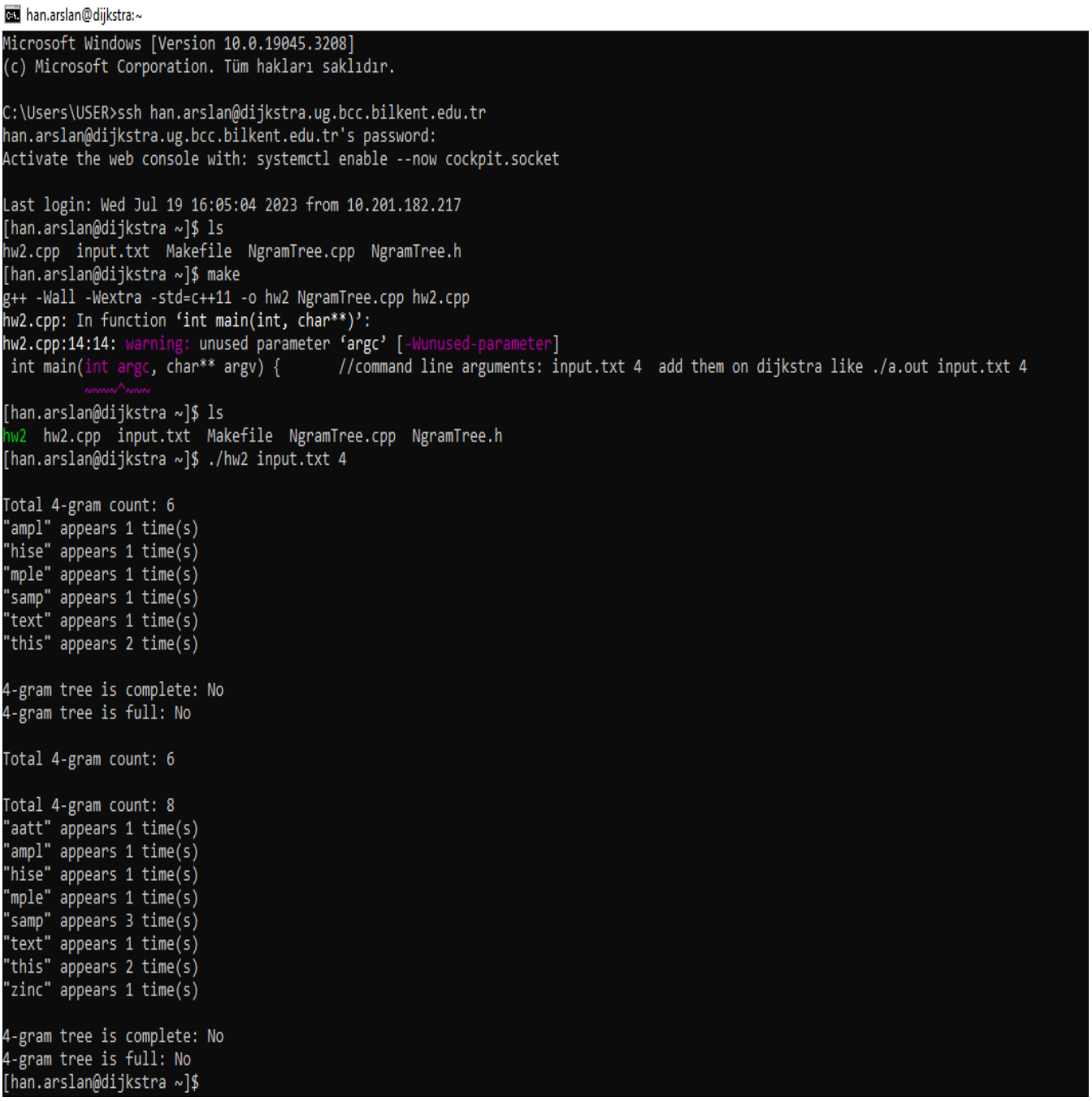


73 replaces 43.



43 is deleted.

Question 3 (Sample output):



Question 4:

- ## addNgram:

```
void NgramTree::addNgram(const string& ngram ){

TreeNode* node = nullptr;

      bool exist = retrieveItem(root, ngram, node);

      if(exist){

         node->count++;

      }

      else{

         insertItem(root, ngram);

      }

   }
```

```
void NgramTree::insertItem(TreeNode *& treePtr, const string& newItem){

      if(treePtr == nullptr){

           treePtr = new TreeNode(newItem, nullptr, nullptr);

       if(treePtr == nullptr){

         cout << "Insertion failed" << endl;

           }

      }

      else if(newItem.compare(treePtr->item) < 0){    //search for left subtree

         insertItem(treePtr->leftChildPtr, newItem);

      }

      else{                                 //search for right subtree

         insertItem(treePtr->rightChildPtr, newItem);

      }

   }
```

```
bool NgramTree::retrieveItem(TreeNode * treePtr, string ngram, TreeNode *& treeItem){
    if(treePtr == nullptr){
        return false;
    }
    else if(ngram.compare(treePtr->item) == 0){
        treeItem = treePtr;
        return true;
    }
    else if(ngram.compare(treePtr->item) < 0){
        return retrieveItem(treePtr->leftChildPtr, ngram, treeItem);
    }
    else{
        return retrieveItem(treePtr->rightChildPtr, ngram, treeItem);
    }
}
```

- addNgram function uses two helper functions insertItem and retrieveItem from the binary search tree class provided in the lecture slides. It first checks if the given ngram is present in the tree with retrieveItem, and do insertion or simple increment accordingly. Since both retrieveItem and insertItem are O(n) algorithms in the worst case (when the tree's height is n, and therefore they both search all the nodes of the tree), addNgram is also O(n) in the worst case.

- **operator<<:**

```
//prints each Ngram alphabetical order, uses inorder traversal
    ostream& operator<<(ostream& out, const NgramTree& tree) {
            tree.printInorder(tree.root, out);
           return out;
    }
    void NgramTree::printInorder(TreeNode * treePtr, ostream& out) const{
       if(treePtr != nullptr){
          printInorder(treePtr->leftChildPtr, out);
          out << "\"" << treePtr->item << "\" " << "appears " << treePtr->count << " time(s)" << endl;
          printInorder(treePtr->rightChildPtr, out);
       }
    }
```

❖ Operator<< uses helper function called printInorder to print the ngrams in the tree in alphabetical order. Since NgramTree is a binary search tree, inorder traversal will visit its nodes in sorted (alphabetical, in this case) search-key order. Since the traversal must visit all the nodes of the tree, best case, average case and worst case behaviour of inorder traversal is O(n), the worst case behaviour of operator<< is also O(n).