CS 340 - Assignment 2

Question 3.6

a. Source code will be submitted separately.
b. Assume the number of players is M and the number of passes is also M. Since the circle is an array (vector is used to implement the array in the source code), after M passes, a player is removed, that empty cell will be disposed. In order to do that, all elements in the right side of the empty cell (if any) have to be shifted to the left to fill that empty spot. As the result, the running time of the program is O(N^2)
c. If M = 1, the running time will be only O(M) since in each turn, there is only empty cell disposal action consumes time (because it requires data shifting), the potato is only passed 1 time per turn.

Question 3.21

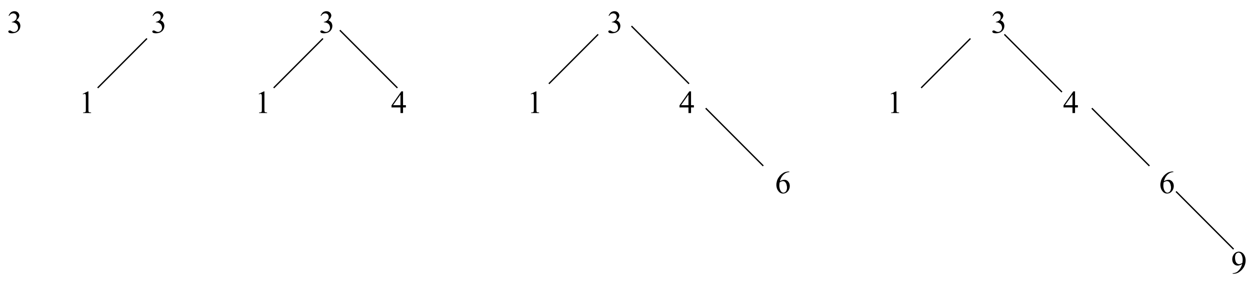
a. Source code will be submitted separately.
b. Source code will be submitted separately (same program as a)
c. The algorithm which is used in the program:
- Read the expression character by character.
- If the current character is an opening operator such as begin, /*, (, [or {, (use substring to retrieve begin and /*), push that character on top of the stack.
- If the current character is a closing operator such as end, */,),] or }:
  + If the stack is empty: there is a closing operator without a corresponding opening operator ——> display error "The number of closing operators is greater than opening operators".
  + If the stack is not empty: check the top. If top is not the corresponding opening operator ——> display error "There is a closing operator that doesn't match with its corresponding opening". Otherwise, pop the stack and move to the next character in the expression.
- Repeat the first step until reach the end of the expression.
- If the stack is not empty, there are opening operators that don't have corresponding closing —-> display error "The number of opening operators is greater than closing". Otherwise, the expression is balanced.

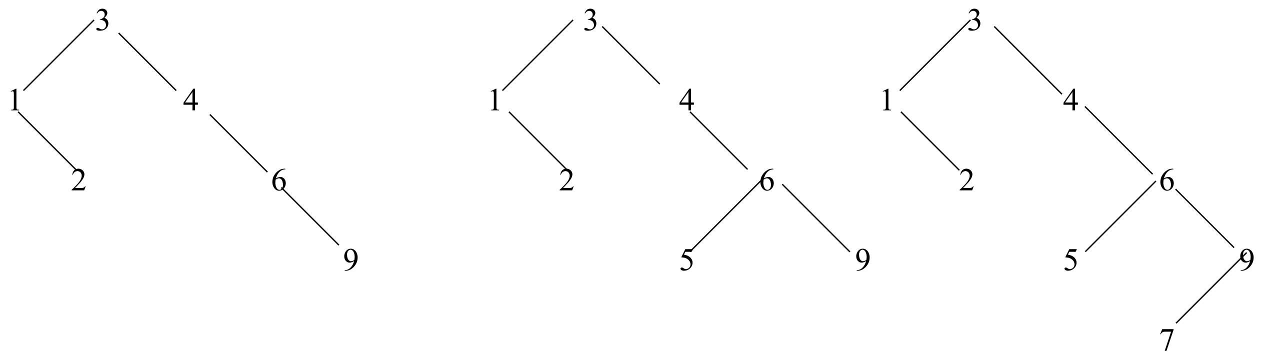
Question 3.23: Source code is submitted separately.

Question 4.8

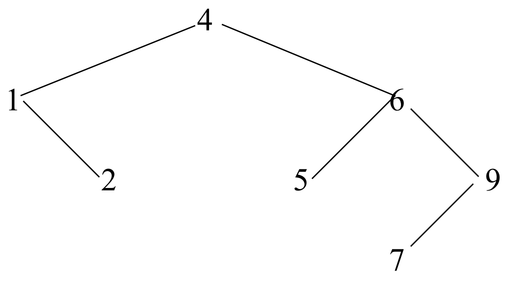
Prefix: *ab*+cd-e
Infix: (a * b * (c + d)) - e
Postfix: ab*cd+*e-

Question 4.9





b. Result of deleting the root



Question 4.16

BST.h

```
#ifndef BST_h
#define BST_h

struct TreeNode {
    int element;
    bool marked; // true if the element is deleted
    TreeNode *left;
    TreeNode *right;
};

class BST {
private:
    TreeNode *head;
    int size;
public:
    BST();
    TreeNode* MakeEmpty(TreeNode*);
    TreeNode* Find(int item, TreeNode* T);
    TreeNode* FindMin(TreeNode*);
    TreeNode* FindMax(TreeNode*);
    TreeNode* Insert(int, TreeNode*);
    TreeNode* Delete(int, TreeNode*);

};

#endif
```

BST.cpp

```
#include <iostream>
#include "BST.h"

using namespace std;

BST::BST() {
    head = NULL;
    size = 0;
}

// Function MakeEmpty: Remove all elements of the tree 
(mark all element)
TreeNode* BST::MakeEmpty(TreeNode* T) {
    if(size == 0)
        return NULL;
    else {
        // If an element is marked, mark its left and 
right subtree
        if(T->marked) {
            MakeEmpty(T->left);
            MakeEmpty(T->right);
        }
        // If an element isn't mark, mark it, reduce 
the size and mark its left and right subtree
        else{
            T->marked = true;
            size--;
            MakeEmpty(T->left);
            MakeEmpty(T->right);
        }
    }
}

TreeNode* BST::Find(int item, TreeNode* T) {
    // Return null if the node is null or there is no 
node (all elements are marked)
    if (size == 0 || T == NULL)
        return NULL;
    // if item is less than the current node, move to 
its left subtree
    if (item < T->element)
        return Find(item, T->left);
    // if item is greater than the current node, move 
to its right subtree
    else if (item > T->element)
        return Find(item, T->right);
    // if item is equal the current node and the node 
is not marked, found!
    else if (item == T->element && !T->marked)
        return T;
    // other cases, return null
    else
        return NULL;
}

TreeNode* BST::FindMin(TreeNode* T) {
    if(size == 0 || T == NULL)
        return NULL;
    else if (T->marked){   // T is marked as deleted
```

```
        if (T->left == NULL) // If T has no left child, 
find min in its right subtree
            return FindMin(T->right);
        // If all elements of T's left subtree are 
marked (deleted), find min in its right subtree
        else if(FindMin(T->left) == NULL)
            return FindMin(T->right);
        else // Find min in T's left subtree
            return FindMin(T->left);
    }
    else { // T is not marked as deleted
        if (T->left == NULL) // The current node has no 
left child
            return T;
        else if (FindMin(T->left) == NULL) // The 
current node has left children but they are marked as 
deleted
            return T;
        else
            return FindMin(T->left)
    }
}

TreeNode* BST::FindMax(TreeNode* T){
    if (size == 0 || T == NULL)
        return NULL;
    else if (T->marked) {    // T is marked as deleted
        if (T->right == NULL) // The current node has 
no right child -> seach left children
            return FindMax(T->left);
        else if(FindMax(T->right) == NULL) // There are 
right children but they are marked
            return FindMax(T->left);
        else
            return FindMax(T->right);
    }
    else { // The current node is not marked
        if(T->right == NULL) // it has no right child
            return T;
        else if(FindMax(T->right) == NULL) // all right 
children are marked
            return T;
        else
            return FindMax(T->right);
    }
}

TreeNode* BST::Insert(int item, TreeNode* T){
    // The tree is empty at the beginning
    if (T == NULL) {
        T = new TreeNode(); // Create a new node
        if (T == NULL) { // Exit the program if there 
is no space
            cout << "Out of space!" << endl;
            return NULL;
        }
        else { // Assign values to the new node and 
mark it false
            T->element = item;
            T->left = NULL;
            T->right = NULL;
            T->marked = false;
            size++;
        }
    }
    // If the item is in tree already but marked, then 
just unmark it
    else if(T->element == item && T->marked)
        T->marked = false;
    // If the item is smaller than the current node, 
insert it to the left substree
    else if(item < T->element)
        T->left = Insert(item, T->left);
    // If the item is bigger than the current node, 
insert it to the right subtree
    else if(item > T->element)
        T->right = Insert(item, T->right);
    // If the item is in the tree and not marked, do 
nothing
```

```
    
    return T;
}

TreeNode* BST::Delete(int item, TreeNode* T) {
    if(T == NULL) {
        cout << "The element is not found." << endl;
        return NULL;
    }
    // Found the item and it's not been marked yet
    else if (T->element == item && !T->marked){
        T->marked = true;
    }
    // Found the item but it's marked already
    else if(T->element == item && T->marked){
        cout << "The element is not found." << endl;
        return NULL;
    }
    // If the item is smaller, move to delete the left subtree
    else if(item < T->element)
        T->left = Delete(item, T->left);
    // If the item is bigger, move to delete the right subtree
    else if(item > T->element)
        T->right = Delete(item, T->right);
    
    return T;
}
```