CSCI 140 PA 9 Submission

Due Date: 10/28/2021 Late (date and time):________________

Name(s): Nero Li

Exercise 1 **(with extra credit)** -- need to submit source code and I/O

-- check if completely done ✓; otherwise, discuss issues below

Source code below:

```
/*  Program: PA_9_exercise_1
    Author: Nero Li
    Class: CSCI 220
    Date: 10/28/2021
    Description:
        Set up LinkedBinaryTree class and then create a simple test driver
to perform
        some basic operations on a binary tree of strings.

    I certify that the code below is my own work.

      Exception(s): N/A

*/
#include <iostream>
#include <list>
#include <queue>

using namespace std;

typedef char Elem;                          // base element type
class LinkedBinaryTree
{
    protected:
        struct Node // a node of the tree
        {
            Elem    elt;                          // element value
            Node*   par;                          // parent
            Node*   left;                               // left child
            Node*   right;                              // right child
            Node() : elt(), par(NULL), left(NULL), right(NULL) { } //
constructor
        };
    public:
        class Position // position in the tree
        {
            private:
                Node* v;                                // pointer to the
node
            public:
                Position(Node* _v = NULL) : v(_v) { }          //
constructor
                Elem& operator*()                              // get
element
                    { return v->elt; }
                Position left() const                   // get left child
                    { return Position(v->left); }
                Position right() const                         // get right
child
                    { return Position(v->right); }
                Position parent() const                        // get
parent
                    { return Position(v->par); }
                bool isRoot() const                     // root of the
tree?
                    { return v->par == NULL; }
                bool isExternal() const                        // an
external node?
                    { return v->left == NULL && v->right == NULL; }
                friend class LinkedBinaryTree;                 // give tree
access
        };
        typedef list<Position> PositionList;            // list of
positions
    public:
        LinkedBinaryTree();                             // constructor
        int size() const;                               // number of nodes
        bool empty() const;                             // is tree empty?
        Position root() const;                    // get the root
        PositionList positions(int choice) const;                    //
list of nodes
        void addRoot();                           // add root to empty
tree
        void expandExternal(const Position& p, Elem &e);       // expand
external node
        Position removeAboveExternal(const Position& p);// remove p and
parent
        // housekeeping functions omitted...
    protected:                                    // local utilities
        void preorder(Node* v, PositionList& pl) const; // preorder
utility
        void inorder(Node* v, PositionList& pl) const;
        void postorder(Node* v, PositionList& pl) const;
        void levelorder(Node* v, PositionList& pl) const;
    private:
        Node* _root;                              // pointer to the root
        int n;                                    // number of nodes
};
```

```
LinkedBinaryTree::PositionList LinkedBinaryTree::positions(int choice)
const  // list of all nodes
{
    PositionList pl;
    switch (choice)
    {
    case 1:
        preorder(_root, pl);                            // preorder
traversal
        break;

    case 2:
        inorder(_root, pl);
        break;

    case 3:
        postorder(_root, pl);
        break;

    case 4:
        levelorder(_root, pl);
        break;

    default:
        break;
    }

    return PositionList(pl);                      // return resulting list
}

void LinkedBinaryTree::preorder(Node* v, PositionList& pl) const  //
preorder traversal
{
    pl.push_back(Position(v));                    // add this node
    if (v->left != NULL)                          // traverse left subtree
        preorder(v->left, pl);
    if (v->right != NULL)                               // traverse right
subtree
        preorder(v->right, pl);
}

void LinkedBinaryTree::inorder(Node* v, PositionList& pl) const
{
    if (v->left != NULL)
        inorder(v->left, pl);
    pl.push_back(Position(v));
    if (v->right != NULL)
        inorder(v->right, pl);
}

void LinkedBinaryTree::levelorder(Node* v, PositionList& pl) const
{
    Node *cur = v;
    queue<Node*> que;

    que.push(cur);
    while (!que.empty())
    {
        cur = que.front();
        que.pop();
        pl.push_back(Position(cur));
        if (cur->left)
        {
            que.push(cur->left);
        }
        if (cur->right)
        {
            que.push(cur->right);
        }
    }
}

void LinkedBinaryTree::postorder(Node* v, PositionList& pl) const
{
    if (v->left != NULL)
        postorder(v->left, pl);
    if (v->right != NULL)
        postorder(v->right, pl);
    pl.push_back(Position(v));
}

LinkedBinaryTree::LinkedBinaryTree()              // constructor
    : _root(NULL), n(0) { }
int LinkedBinaryTree::size() const                // number of nodes
    { return n; }
bool LinkedBinaryTree::empty() const              // is tree empty?
    { return size() == 0; }
LinkedBinaryTree::Position LinkedBinaryTree::root() const // get the root
    { return Position(_root); }
void LinkedBinaryTree::addRoot()                  // add root to empty
tree
    { _root = new Node; n = 1; }

                                            // expand external node
void LinkedBinaryTree::expandExternal(const Position& p, Elem &e)
{
    Node* v = p.v;                          // p's node
    v->left = new Node;                           // add a new left child
    v->left->par = v;                             // v is its parent
    v->right = new Node;                    // and a new right child
    v->right->par = v;                            // v is its parent
    v->elt = e;
    n += 2;                                 // two more nodes
}
```

```
LinkedBinaryTree::Position                        // remove p and parent
LinkedBinaryTree::removeAboveExternal(const Position& p)
{
    Node* w = p.v;  Node* v = w->par;             // get p's node and
parent
    Node* sib = (w == v->left ?  v->right : v->left);
    if (v == _root) // child of root?
    {
        _root = sib;                              // ...make sibling root
        sib->par = NULL;
    }
    else
    {
        Node* gpar = v->par;                      // w's grandparent
        if (v == gpar->left) gpar->left = sib;          // replace parent
by sib
        else gpar->right = sib;
        sib->par = gpar;
    }
    delete w; delete v;                           // delete removed nodes
    n -= 2;                                 // two fewer nodes
    return Position(sib);
}

void printTree(LinkedBinaryTree tree)
{
    LinkedBinaryTree::PositionList l;
    int choice{1};

    cout << "Pre-order: ";
    l = tree.positions(choice);
    while (!l.empty())
    {
        if (*l.front())
        {
            cout << *l.front() << ' ';
        }
        l.pop_front();
    }
    cout << endl;

    ++choice;
    cout << "In-order: ";
    l = tree.positions(choice);
    while (!l.empty())
    {
        if (*l.front())
        {
            cout << *l.front() << ' ';
        }
        l.pop_front();
    }
    cout << endl;

    ++choice;
    cout << "Post-order: ";
    l = tree.positions(choice);
    while (!l.empty())
    {
        if (*l.front())
        {
            cout << *l.front() << ' ';
        }
        l.pop_front();
    }
    cout << endl;

    ++choice;
    cout << "Level-order: ";
    l = tree.positions(choice);
    while (!l.empty())
    {
        if (*l.front())
        {
            cout << *l.front() << ' ';
        }
        l.pop_front();
    }
    cout << endl << endl;
}

int main()
{
    LinkedBinaryTree test;
    LinkedBinaryTree::Position p;
    char c{'A'};

    test.addRoot();
    p = test.root();
    test.expandExternal(p, c);

    ++c;
    p = p.left();
    test.expandExternal(p, c);

    ++c;
    p = p.parent();
    p = p.right();
    test.expandExternal(p, c);

    ++c;
    p = p.parent();
    p = p.left();
    p = p.left();
    test.expandExternal(p, c);

    ++c;
    p = p.parent();
    p = p.right();
    test.expandExternal(p, c);

    printTree(test);

    p = p.left();
    test.removeAboveExternal(p);
    cout << "After remove node E:\n";

    printTree(test);

    p = test.root();
    p = p.left();
    p = p.right();
    test.expandExternal(p, c);

    ++c;
    p = test.root();
    p = p.right();
    p = p.right();
    test.expandExternal(p, c);

    ++c;
    p = test.root();
    p = p.left();
    p = p.right();
    p = p.left();
    test.expandExternal(p, c);
    cout << "For Question 1 Answer:\n";

    printTree(test);

    cout << "Modified by: Nero Li\n";
    return 0;
}
```

Input/output below:

```
Pre-order: A B D E C
In-order: D B E A C
Post-order: D E B C A
Level-order: A B C D E

After remove node E:
Pre-order: A B D C
In-order: D B A C
Post-order: D B C A
Level-order: A B C D

For Question 1 Answer:
Pre-order: A B D E G C F
In-order: D B G E A C F
Post-order: D G E B F C A
Level-order: A B C D E F G

Modified by: Nero Li
```

Answer for Question 1:

Preorder: A B D E G C F

In order: D B G E A C F

Post-order: D G E B F C A

Level order: A B C D E F G

Answer for Question 2:

For an arithmetic expression tree, post-order will be the most useful traversal for computer to calculate the answer. We create a stack for saving number, when we read a number, push it into the stack; when we read an operator, pop two numbers, calculate, and push the result back into the stack. Finally, the last one element that remain in the stack will become the result for the expression.

Then, in order traversal is also useful because this is how we see a general expression in our normal life. However, to help the program calculate the answer for this arithmetic expression tree, we need to transfer the in order to post-order so that we can use the way that post-order do to calculate the answer.