CS302

Assignment 5: LISP

Description

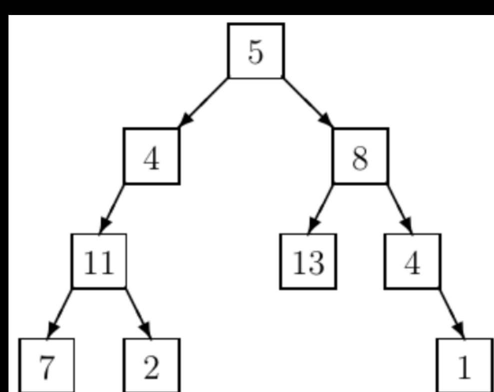
LISP is written in LISP, which makes perfect sense! But for now, LISP will be written/interpreted in C++. Well this is not going to be actual LISP language interpretation but rather a simplified version of it. Every LISP expression starts with a left parenthesis and ends with a right parenthesis, and each expression can have sub expressions, in fact this version of LISP every expression will have exactly 2 sub expressions (almost is fitting to be interpreted using a binary tree). The syntax for this LISP type language is

- `Expression_Tree => Integer_Tree` | `()`
- `Integer_Tree => (Number Expression_Tree Expression_Tree)`

Thus the following expression

(5 (4 (11 (7 () ()) (2 () ())) ()) (8 (13 () ()) (4 () (1 () ()))))

Can be represented with the following tree



Once you have the tree built, each LISP expression will have a target sum given along with the expression and you need to determine if a path from the root to a leaf contains the numbers added together equals the target sum. You will need to define a binary tree struct, a stack (to output the actual path), and a set of functions to built and traverse the tree. Here is the linked list implementation of the stack, so the struct node is each element of the stack

Stack Class

```
template <class Type>
class myStack
{
```

```
public:
  myStack();
  myStack(const myStack<Type>&);
  const myStack<Type>& operator=(const myStack<Type>&);
  ~myStack();

  void push(const Type&);
  void pop();
  Type top() const;
  bool isEmpty() const;

private:
  struct node
  {
    Type item;
    node * next;
  };

  node * topOfMyStack;
};
```

Each member will contain/perform the following

- `node * topOfmyStack` - pointer that points to the top of the stack (essentially a head pointer)
- `myStack<Type>::myStack()` - default constructor that sets `topOfMyStack` to `NULL`
- `myStack<Type>::myStack(const myStack<Type>& copy)` - copy constructor that performs a deep copy of the `copy` object to the `*this` object
- `const myStack<Type>& myStack<Type>::operator=(const myStack<Type>& rhs)` - assignment operator that does a deep copy ot the `rhs` to the `*this` object
- `myStack<Type>::~myStack()` - destructor, deallocates the stack object
- `void myStack<Type>::push(const Type& insert)` - pushes a new node to the top of the stack (aka a head insert) and assigns insert into this new node's item field
- `void myStack<Type>::pop()` - removes the top element (head removal) if the stack is not empty, otherwise nothing happens
- `Type myStack<Type>::top() const` - returns the item of the top node in the stack
- `bool myStack<Type>::isEmpty() const` - returns `true` if the stack is empty and `false` if the stack is not empty

Binary Tree

I would recommend that you implement the following struct along functions to build, evaluate, and deallocate the tree (you need to include `template <class type>` above each prototype and when the function is being implemented)

```
template <class Type>
struct binTreeNode
{
  Type item;
  binTreeNode<Type> * left;
  binTreeNode<Type> * right;
};
```

- `void readLISP(binTreeNode<type> * r, ifstream& infile)` - this function reads from the `ifstream infile` variable and builds the tree, the `r` pointer is pointing to some node in the current binary tree, you will build this tree in a preorder type fashion
- `bool evaluate(binTreeNode<type> * r, int runningSum, int targetSum, myStack<type>& path)` - this function does a preorder type traversal to determine if a path from the root to a leaf contains a set of numbers along the path whose sum equals the targetSum, the runningSum is the current sum from the root to the current node `r` that we are currently looking at, the path stack contains the path in the reverse order (all the integers of the tree along the solution path). Once a path is established the function returns `true` up the function call tree each earlier function call pushes `r->item` onto the stack before relaying the `true` value up to its predecessor and so on. You essentially return `true`/`false` value when you reach a leaf node.
- `void destroyTree(binTreeNode<type> * r)` - deallocates the tree in a postorder type fashion

Input

The input file has a series of test cases. Each test case starts with an integer which is the target sum for the test case followed by a LISP expression. Once end of file is reached the file is processed.

Output

For each test case, output whether a solution is possible or not, if there exists a solution, output the integer value path from the root to a leaf, if there is no solution simply say there is no solution

Contents of main

In main you prompt the user for an input file and re-prompt for input file if an invalid file is given. For each test case build the tree, determine if a solution is possible, and then destroy the tree (using the appropriate functions)

Specifications

- Comment your code and your functions
- Do not add extra class members or remove class members and do not modify the member functions of the class
- No global variables (global constants are ok)
- Make sure your program is memory leak free

Sample Run

```
$ g++ main.cpp
$ ./a.out

Enter LISP file (All those parenthesis...): LISP_IS_AWESOME.txt
Enter LISP file (All those parenthesis...): L_input.txt

Path in tree exists
5 + 4 + 11 + 2 = 22
```

```
No such path exists, LISP is a pain anyway

Path in tree exists
3 + 1 + 6 = 10

No such path exists, LISP is a pain anyway
```

Submission

Upload your files, binaryTree.h, myStack.h, and main.cpp to webcampus by deadline

References

- Link to the top image can be found at *https://gopslog.wordpress.com/*