COP4530 – Project 4

Binary Tree and Its Applications

Submission:

1. Create a tar file with the required files.
2. Make an electronic submission of your tarball file on Canvas.

Due Date: Nov 12, 2019, 11:50pm for all students
Late Penalty: maximum one day at the penalty of 15%.

1. Overview

This project is our second designated ABET assessment assignment. You need to achieve a grade of C- for at least one of these two assignments.

In this project, you shall gain experience with a binary tree and its application in converting postfix expressions into infix expressions. Portions of this assignment (including your tree implementation and usage of the tree) will be used for the ABET outcome that verifies that students are able to effectively use recursive functions. Overall objectives of this project include the following:

- Understand the binary trees and their applications;
- Grasp the design and implementation of binary expression tree;
- Leverage the binary expression tree to convert postfix expressions into infix expressions; and
- Practice the implementation of recursive algorithms using binary trees.

2. Description of Provided Source Files

A tar file `proj4src.tgz` is available on Canvas. To extract these files from the tar file `proj4src.tgz`, you may use the following command **on the `linprog` machines, where your code will be tested and graded**.

- `tar zxf proj4src.tgz`

This project includes two sets of documents. One is essential for the project completion. The other set is provided to encourage the use of the `flex` tool for lexical analysis in this project.

Essential files

You can three essential documents as described below. These provided documents provide a good set of sample tests. You are welcome to come up with more test cases, for testing Stack features and for testing your Infix-To-Postfix program.

- `token.h:` a source file that defines a class Token for your use in this project.
- `cases:` A directory with 6 different test cases (`test[0-5].txt`) for the test driver. The expected output with these testcases are also provided in a file `output.txt` in this directory.
- `bet_driver.cpp:` A sample test driver to test the BET implementation. It accepts input from terminal, or an input file that contains the postfix expressions to be converted. Each line in the file (or terminal input typed by user) represents a postfix expression. We assume that the tokens in a postfix expression are separated by space.
- `bet_driver:` It is the executable created from the sample test driver program. You can run it with any of the test files. For example, you can run with the command below.

```
bet_driver test0.txt
```

Sample token generation using flex

Throughout your undergraduate program, you will have, or may already have had, multiple needs to parse an input file and generate many different operators, keywords, numerical values and identifiers. Three files are provided here to demonstrate the generation of operators, numerical values for this project.

- `opnum.h:` A header file with the definition of some constants and the declaration of utility functions for flex-based token generation.
- `opnum.fl:` The main file defines how to match the tokens from an input file.
- `line_parser.cpp:` A sample driver program that takes an input file, extracts every token and input them to a list. It finishes by printing out all tokens in each line of the input file.
- `Makefile:` A sample Makefile that allows you to compile line_parser.cpp. You can run the test program using commands below.
  - `make line_parser`
  - `line_parser test0.txt`

You will need to make some minor modifications so that these utility files can support your bet_driver program correctly. If your bet_driver program is dependent on your modifications on these files, please include your modified files as part of your project 4 submission.

3. Project Tasks:

In this project, you are asked to develop a binary expression tree and use the tree to convert postfix expressions into infix expressions. In this project, a postfix expression may contain 4 types of operators: multiplication (*), division (/), plus (+), and minus (-). We assume that multiplication and division have the same precedence, plus and minus have the same precedence, and multiplication and division have higher precedence than plus and minus. All operators are left-associative (i.e. associate left-to-right).

A. Binary Expression Tree

Build a binary expression tree class called "BET". Your BET class must have a nested structure named "BinaryNode" to contain the node-related information (including, e.g., element and pointers to the children nodes). In addition, BET must at least support the following interface functions:

- **Public interface**

  - `BET():` default zero-parameter constructor. Builds an empty tree.
  - `BET(const list<Token> & postfix):` one-parameter constructor, where parameter "postfix" is a list representing a postfix expression. The tree should be built based on the postfix expression.
  - `BET(const BET&):` copy constructor -- makes appropriate deep copy of the tree
  - `~BET():` destructor -- cleans up all dynamic space in the tree
  - `bool buildFromPostfix(const list<Token> & postfix):` parameter "postfix" is a list representing a postfix expression. A tree should be built based on each postfix expression. Tokens in the postfix expression are separated by spaces. If the tree contains nodes before the function is called, you need to first delete the existing nodes. Return true if the new tree is built successfully. Return false if an error is encountered.

- `const BET & operator= (const BET &):` assignment operator -- makes appropriate deep copy.
- `void printInfixExpression():` Print out the infix expression. Should do this by making use of the private (recursive) version
- `void printPostfixExpression():` Print the postfix form of the expression. Use the private recursive function to help
- `size_t size():` Return the number of nodes in the tree (using the private recursive function)
- `size_t leaf_nodes():` Return the number of leaf nodes in the tree. (Use the private recursive function to help)
- `bool empty():` return true if the tree is empty. Return false otherwise.

- **Private helper functions**

All the required private member functions must be implemented recursively.

- `void printInfixExpression(BinaryNode *n):` print to the standard output the corresponding infix expression. Note that you may need to add parentheses depending on the precedence of operators. You should not have unnecessary parentheses.
- `void makeEmpty(BinaryNode* &t):` delete all nodes in the subtree pointed to by t.
- `BinaryNode * clone(BinaryNode *t):` clone all nodes in the subtree pointed to by t. Can be called by functions such as the assignment operator=.
- `void printPostfixExpression(BinaryNode *n):` print to the standard output the corresponding postfix expression.
- `size_t size(BinaryNode *t):` return the number of nodes in the subtree pointed to by t.
- `size_t leaf_nodes(BinaryNode *t):` return the number of leaf nodes in the subtree pointed to by t.

B. Conversion to Infix Expression

To convert a postfix expression into an infix expression using a binary expression tree involves two steps. First, build a binary expression tree from the postfix expression. Second, print the nodes of the binary expression tree using an inorder traversal of the tree.

The basic operation of building a binary expression tree from a postfix expression is similar to that of evaluating postfix expression. Refer to Chapter 4 for the basic process of building a binary expression tree from a postfix expression.

Note that during the conversion from postfix to infix expression, parentheses may need to be added to ensure that the infix expression has the same value (and the same evaluation order) as the corresponding postfix expression. Your result should not add unnecessary parentheses. Tokens in an infix expression should also be separated by a space. The following are a few examples of postfix expressions and the corresponding infix expressions.

postfix expression	**infix expression**
4 50 6 + +	4 + (50 + 6)
4 50 + 6 +	4 + 50 + 6
4 50 + 6 2 * +	4 + 50 + 6 * 2

4 50 6 + + 2 *	(4 + (50 + 6)) * 2
a b + c d e + * *	(a + b) * (c * (d + e))

4. Other Requirements:

- **You can use any C++/STL containers and algorithms.** If you do use any containers, you must use the ones provided in C++/STL. Do not use the ones you developed in the previous projects.
- Your program MUST check invalid postfix expressions and report errors. We consider the following types of postfix expressions as invalid expressions: 1) an operator does not have the corresponding operands, 2) an operand does not have the corresponding operator. Note that an expression containing only a single operand is a valid expression (for example, "6"). In all other cases, an operand needs to have an operator.
- Analyze the worst-case time complexity of the private member function makeEmpty (BinaryNode* & t). Give the complexity in the form of Big-O. Your analysis only needs to be informal; however, it must be clearly understandable by pointing the complexity of these functions with respect to the size of the tree. Name the file containing the complexity analysis as "analysis.txt".
- If you have a modified Makefile, it should also compile the provided driver program (see below) with your BET class, into an executable called `"bet_driver".`

5. Programs and files you need to submit.

In this project, you are allowed to use any C++/STL containers and algorithms. You can also modify any file that has been provided. If you indeed modify the provided files, you need to include all your source files and Makefile in the submission. For example, you may need to expand token.h for a richer nested class implementation.

In short, you need to submit at least three files, `bet.h, Makefile and analysis.txt,` along with any other files required for your code to work. Create a tarball file from all your files using the command below, where `fsuid` is your FSU ID.

- `tar zcf <fsuid>_proj4.tgz bet.h Makefile analysis.txt <other_files>`

6. General Requirements

- Document your code appropriately so that it is readable and easy to navigate
- Make sure your implementation compile and run on linprog.cs.fsu.edu with g++, using the C++11 standard compilation flag.
- Your submission should work in a similar manner to produce the driver programs, as shown in the file `cases/output.txt`.

7. Breakdown of points

Note that we will use more thorough tests than the test driver program, test_list.cpp. We have designed four different ABET assessments for this project.

ABET Rubric	Details of ABET Assessments
A1	Effective understanding of C++ class/function declarations, effective creation of C++ utility functions; and effective creation and use of UNIX commands and Makefile for program creation and execution.
A2	Effective understanding, design and implementation of nested classes and their member functions for smooth program solutions.

A3	Effective understanding, design and implementation of container classes and their member functions for complex program solutions.
A4	Effective understanding of problems of different complexities, design and implementation of solutions with different asymptotic complexities.
A5	Effective understanding of asymptotic notations and clear indication of the complexity of makeEmpty().

The grade breakdown of points for these assessments is provided as follows:

ABET	**Points**	**Requirements**
A1	20	Have completed the required public functions in bet.h in a reasonably correct manner. bet_driver.cpp can compile with bet.h via a Makefile.
A2+A3	45	Reasonable implementation of nested classes and private helper functions for an executable driver program. Successful passing of the driver program using test0.txt.
A4	25	Correct implementation and successful passing of all other test cases in test files test[1-5].txt. 5 points per test case.
A5	10	Good understanding on the complexity of the function makeEmpty(); and clear indication of its complexity with an asymptotic notation.
Bonus	10	Have used the flex tool for parsing the input and the driver program works correctly on top of the modified flex tool.

8. Miscellaneous

The first person to report any compilation or execution errors in the provided materials will be given 3% extra credit (maximum 15% per person). Automatic plagiarism detection software will be used on all submissions – any cases detected will result in a grade of 0 for those involved.