CSE221 Data Structures (Spring 2018)
Instructor: Prof. Tsz-Chiu Au
Due date: April 15, 2018, 11:59 pm.

Assignment 2: Binary Trees

A binary tree is a tree data structure in which each node has at most two children called the left child and the right child. Binary trees are the most basic data structures that are widely used in all kinds of algorithms. Therefore, computer science students should learn how to implement a binary tree correctly.

In this assignment, you will implement a binary tree to store a numerical expression that consists of integer constants, variables, and operators. We call this binary tree *an expression tree*. The internal nodes of an expression tree are operators, while the external nodes (i.e., leaf nodes) can be either integer constants or variables. For example, let consider the following expression:

$$S_1 = (((X + 1) * X / ((9 - 5) + 2)) – ((X * (7 – 4)) + Y))$$

The corresponding expression tree is shown in Figure 1.

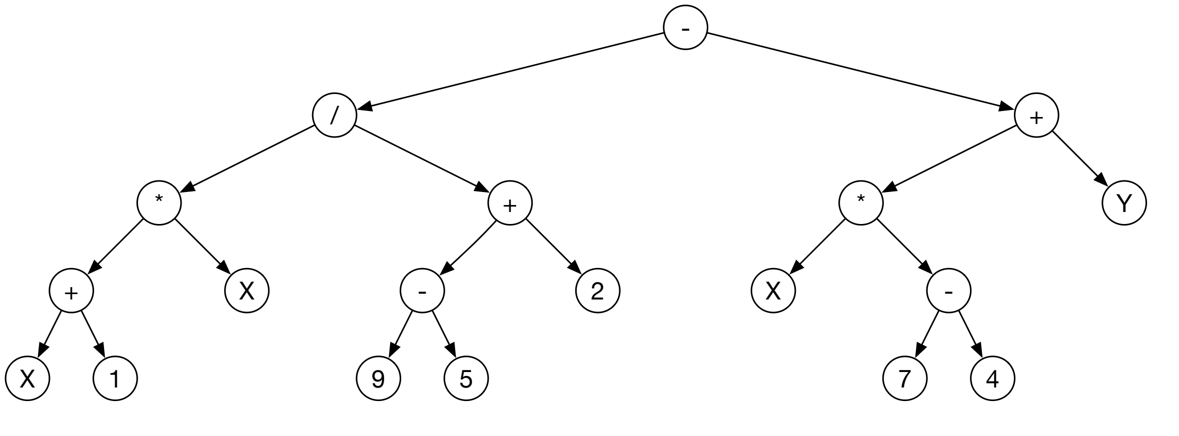

Figure 1: The expression tree of S_1.

In this expression, there are two variables: X and Y. Here a variable is any token that starts with an alphabet in either lower case or upper case. As you can see, all internal nodes are operators, and all external nodes are either integer constants (which can be negative integers) or variables. We consider four different kinds of operators only: +, -, *, and /. All of these are integer operators.

The main objective of this exercise is for you to implement the class template

LinkedBinaryTree as described in the textbook and use your implementation to perform the following three operations:

1. Parse an expression in postfix notation into an expression tree
2. Substitute constants for variables in an expression tree.
3. Simplify the expression tree by evaluating subtrees with constants only.

Task 1: Implementing Linked Binary Trees

You will implement the Linked Binary Tree class template as described in Section 7.3.4 of our textbook Goodrich et al. (2011). Most of time, your implementation should faithfully follow the description in the textbook. However, your implementation should provide better error handling and include four more member functions that help to implement the expression trees in Task 2.

Notice that some codes in Section 7.3.4 of the textbook do *not* conform with the C++ standard. Moreover, there are some minor bugs in the codes. You cannot directly copy the codes in the textbook and expect they will work perfectly; you need to convert the codes to correct C++ codes that can be compiled using the gcc compiler on our submission server.

You will put your codes of Linked Binary Trees in a file named LinkedBinaryTree.h, which is left blank so that you have to implement everything yourself. First, you should choose to include a minimum set of header files that your program truly uses, as it is not a good practice to include unused header files. More importantly, you should not include "Symbol.h" or "assignment2.h" as your implementation of Linked Binary Trees should be generic.

Second, since you are implementing a class template, your member functions cannot be put in a separated .cpp file and link it with other files later. In general, you should include the definition of the member functions of a class template in the header file, preferably as inline functions in the template class definition. However, you can also put the non-inline member functions in the header files. For this reason, you need to include #ifndef, #define, and #endif directives at the beginning and the end of the header file so as to prevent cyclic file inclusion that will cause the same member functions to be defined multiple times. In this exercise, you should put both the class template of Linked Binary Trees and the definition of the member functions in LinkedBinaryTree.h.

As described in the textbook, the class template LinkedBinaryTree should have two inner classes: Node and Position. Node is a structure that holds the data of a node in a binary tree, whereas Position is a pointer-like class that refers to a position in a tree. Note that a position object can be a "null" pointer, which refers to no position. As described in the textbook, The Position class should include these member functions:

`operator*()`, `left()`, `right()`, `parent()`, `isRoot()`, and `isExternal()`. Moreover, we want you to include one more member function:

```
bool isNull() const;
```

which checks whether a position object refers to a non-existing position. The behavior of `isNull()` is somewhat similar to `end()` in an iterator in STL. Since we are not implementing a full-fledged iterator using the Position class, we will call this function `isNull()` instead of `end()`.

In addition, the member function `operator*()` in the class Position could be declared `const` as follows:

```
E& operator*() const { return v->elt; }
```

where `E` is a type parameter. `const` after this member function declaration means that the function is not allowed to change any class members in Position; more precisely, the `this` pointer being passed to this function is a constant. As you can see, `operator*()` does not modify the position object. Hence it is better to add `const` to its declaration.

The class template `LinkedBinaryTree` should include all member functions as described in the textbook: `size()`, `empty()`, `root()`, `positions()`, `addRoot()`, `expandExternal()`, and `removeAboveExternal()`. However, the implementation of these functions in the textbook performs no error checking. For example, you cannot add a root to a tree when the tree is not empty, and you cannot expand an internal node. If you pass a "null" position to these member functions, most of them will fail. Therefore, in your implementation you are required to throw runtime errors with appropriate error messages in these cases as well as other problematic cases. We will check whether your program can throw exceptions when necessary.

The definition of `removeAboveExternal()` in Code Fragment 7.22 fails when the position `p` is the root. The correct behavior is that the root node should be removed if the root node is an external node, even though the root node has no parent to be removed. Please fix this bug and provide a correct implementation of `removeAboveExternal()`. In addition, we want you to implement four additional member functions:

- `int height() const;`

  This function returns the height of the tree. Note that if the tree is empty, there is no height. You need to throw an exception if the tree is empty.

- `void attachLeftSubtree(const Position& p, LinkedBinaryTree& subtree);`

This function attaches another tree to the tree as the left subtree of a node at position `p`. Clearly, the left child of `p` should be empty; otherwise an exception should be thrown. This operation is destructive to subtree: after attaching to `p`, subtree will become an empty tree, as the 'gut' of subtree is transferred to the tree of `p`.

- `void attachRightSubtree(const Position& p, LinkedBinaryTree& subtree);`

  This function attaches another tree to the tree as the right subtree of a node at position `p`. Clearly, the right child of `p` should be empty; otherwise an exception should be thrown. This operation is destructive to subtree: after attaching to `p`, subtree will become an empty tree, as the 'gut' of subtree is transferred to the tree of `p`.

- `void removeSubtree(const Position& p);`

  Remove the subtree starting at position `p`. You have to free the memory allocated to the subtree.

You are free to include any number of private or protected member fields and functions in the class template `LinkedBinaryTree`. However, you should not add any other public member functions. The functions in `assignment2.cpp` should only use the above member functions and no other member functions of `LinkedBinaryTree`. Please write your name, your student ID, and your email as a comment at the top of the file. You should also briefly describe your implementation at the top of the file.

In this exercise, you can assume that the users of `LinkedBinaryTree` objects will call `tree.removeSubtree(tree.root())` before the tree is deleted. This will simplify the problem and there is no need to define a destructor and a copy constructor. In fact, the textbook used this assumption. However, you can have already defined the copy constructor and the destructor, it is okay to keep your program in your way and ignore this assumption.

Task2: Expression Tree

We will use your implementation of the class template `LinkedBinaryTree` to store an expression tree. You will implementation several functions in `assignment2.cpp` in order to support the following operations:

1. Parsing an expression in postfix notation

Your program will allow a user to enter an expression in *postfix* notation. For example, when your program wrote, “Please enter an expression terminated with ‘#’: ”, a user can enter the following expression in postfix notation that ended with a pound sign:

X 1 + X * 9 5 - 2 + / X 7 4 - * Y + - #

We opt for postfix notation instead of the usual infix notation because it is easier to paste an expression in postfix notation; there is no need to consider the operator precedence and associativity, as well as there is no parenthesis. More importantly, it is easy to parse an expression in postfix notation to an expression tree—all your program need is to do is to maintain ***a stack of expression trees***. The algorithm is similar to the evaluation function for postfix notation, which scans the expression from left to right until it hits a pound sign. When it encounters an operand during the scan (which can be either an integer constant or a variable), it also pushes the operand on the stack as a single-node expression tree. The key difference is that when it encounters an operator, it pops the top two expression trees in the stack and attaches them to the left and right children of a new expression tree whose root is the operator. If the expression is a correct expression, there should be exactly one expression tree left in the stack after scanning through the expression, and that is the expression tree of the expression. In our example, the expression tree your program generates should be the one in Figure 1.

In this exercise, you are allowed to use the stack class in the C++ Standard Template Library (STL) to implement this algorithm. Moreover, whenever your program found that the expression is incorrect, your program should

```
throw runtime_error("Invalid expression.")
```

where `runtime_error` is an exception defined in the `stdexcept.h` header file in the C++ Standard Library.

2. Variable substitution

After parsing a tree, your program will allow a user to substitute a constant for a variable. For example, if a user substitutes 3 for the variable X in S_1, the new expression is S_2 = (((3 + 1) * 3 / ((9 - 5) + 2)) – ((3 * (7 – 4)) + Y)), and the expression tree will become the expression tree in Figure 2.

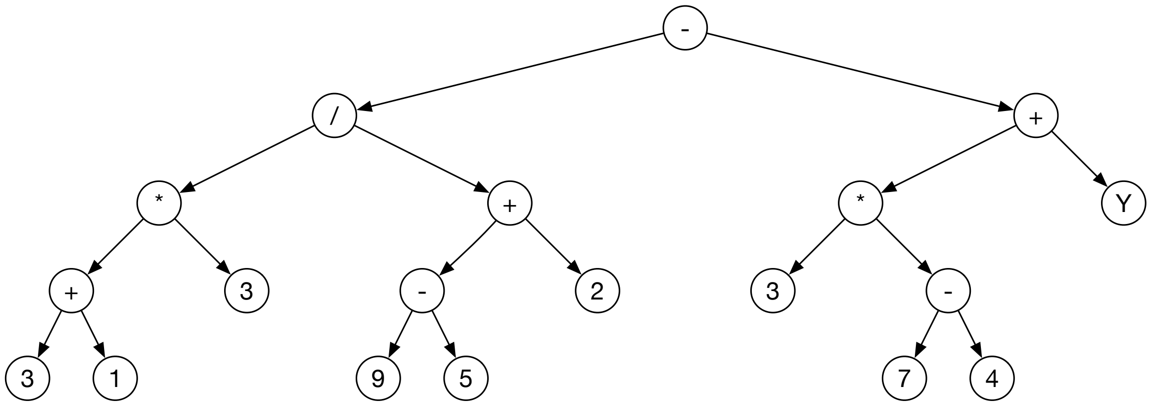

Figure 2: The expression tree of S_2 obtained by substituting 3 for X in S_1.

3. Simplify the expression tree by evaluating subtrees with constants only.

In the expression tree in Figure 2, there are subtrees that contain constants only and have no variables in their leaf nodes. For example, the subtree starting at the left child of the root contains no variable but constants and operators only. It is clear that this subtree can be safely replaced with an integer constant 2, which is the result of the evaluation of ((3 + 1) * 3 / ((9 - 5) + 2). Likewise, we can replace the subtree of 3 * (7 – 4) by 9. Then the expression tree will be simplified to the expression tree in Figure 3.

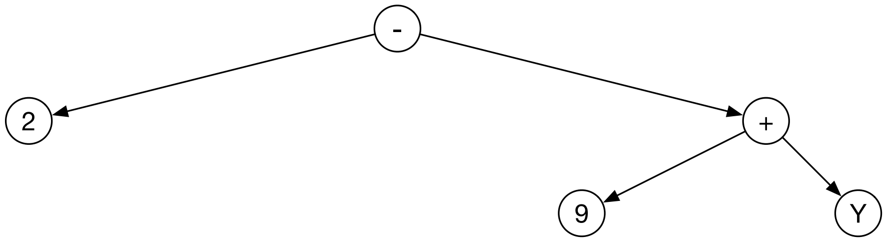

Figure 2: The expression tree of S_3 obtained by simplifying S_2.

As can be seen, this simplified expression tree corresponds to the expression S_3 = 2 – (9 + Y). More importantly, no more simplification is possible because there is no subtree whose external nodes are constants only. Your program should simplify the expression tree until no more simplification is possible.

Since the simplification process involves evaluations of subexpressions, your program should be able to handle errors during the evaluations, notably the division-by-zero error. When the right operand of / is zero, your program needs to

```
throw runtime_error("Divide by zero.")
```

One obvious way to perform this simplification is to evaluate nodes in a bottom-up fashion. However, we intentionally do not disclose our method and let you design your own algorithm.

Implementing the functions in assignment2.cpp

We provide you the main function in `main.cpp`, which makes use of the functions defined in `assignment2.h` and `assignment2.cpp`. You are not allowed to modify the main function and main.cpp to bypass the calls to these functions and implement your own solution. To complete the program, you are required to implement the following functions in `assignment2.cpp`:

- `LinkedBinaryTree<Symbol> parsePostfixExpression(list<string> tokens);`

  This function takes an expression in postfix notation and parse it into an expression tree. The expression is stored in a list of strings in which each string is a token. Please take a look how the list of tokens is read from `cin` in the main function. The function should return an expression tree of the expression. We recommend you to use

  ```
  LinkedBinaryTree<Symbol>::attachLeftSubtree()
  ```

  and

  ```
  LinkedBinaryTree<Symbol>::attachRightSubtree()
  ```

  to build the tree. Furthermore, the function should

  ```
  throw runtime_error("Invalid expression.");
  ```

  if the expression is not in a correct postfix notation (e.g., missing operands, dangling operators, etc.). Any invalid symbols in the expression is taken care by the Symbol class we provide (by throwing runtime exceptions), and you don't need to check the validity of the symbols.

- `void print_inorder(const LinkedBinaryTree<Symbol>& tree);`

  This function prints the expression in infix notation to `cout`. The output should include all parentheses. Please take a look at the output of the sample program we provide to see what output this function should generate.

- `void print_postorder(const LinkedBinaryTree<Symbol>& tree);`

  This function prints the expression in postfix notation to `cout`. The output should include no parenthesis. Please take a look at the output of the sample program we provide to see what output this function should generate.

- `int findMinimumDepth(const LinkedBinaryTree<Symbol>& tree, const Symbol& sym);`

  This function returns the minimum depth of a variable `sym` in an expression tree. If the variable appears only once in the expression tree, this function returns the depth of the node storing the variable. If the variable appears more than once in the expression tree, this function returns the minimum value of the depths of the nodes storing the variable. If the variable does not appear in the expression tree, this function returns -1. Notice that this function does not throw exceptions when the variable does not appear in the expression tree.

- `void substitute(const LinkedBinaryTree<Symbol>& tree, const Symbol& variable, const Symbol& constant);`

  This function replaces all appearances of a given variable in an expression tree with a given constant. The replacement should be taken places in the original tree and there is no need to create a new tree.

- `void simplify_subtree(LinkedBinaryTree<Symbol>& tree);`

  This function simplifies a subtree as described in the introduction of this handout. Basically, it finds all subtrees whose external nodes are all constants, evaluates the subtrees, and replaces them with the results of the evaluations. You should not create a new tree to store the simplified tree; instead, you should call

  ```
  LinkedBinaryTree<Symbol>::removeSubtree()
  ```

  to modify the expression tree in place.

You should read `main.cpp` to see exactly how these functions are being used. `Assignment2.cpp` has been partially filled out, and you need to implement the functions in this file. Notice that you are free to implement other helper functions in `assignment2.cpp` that are used by your functions. But you should not put the function prototypes of these helper functions in `assignment2.h`. In fact, you will not submit `assignment2.h` for grading. Please write your name, your student ID, and your email as a comment at the top of the file. You should also briefly describe your implementation at the top of the file.

Testing and Submission

All files that you need in this assignment are put in a zip file called `prog2.zip`, which contains the following files:

- `handout-2.pdf` — the handout of this assignment (i.e., this file).
- `ChangeLog.txt` — the change log.
- `exptree-sample` — the sample program.
- `main.cpp` — the main function of your program.
- `LinkedBinaryTree.h` — your implementation of the binary tree. You will submit this file.
- `Symbol.h` — the declaration of the `Symbol` class.
- `Symbol.cpp` — the implementation of the `Symbol` class.
- `assignment2.h` — the declaration of the functions for handling expression trees.
- `assignment2.cpp` — the implementation of the functions for handling expression trees. You will submit this file.
- `README.TXT` — your readme file. You will submit this file.

We provide you a sample program called "`exptree-sample`" that implements all functionalities of this program. Please play with the program to see how it behaves, and make sure that the output of your program is the same as the output of the sample program (except the white spaces and the error messages in the exceptions).

We also provide two files that implement the Symbol class: `Symbol.h` and `Symbol.cpp`. The implementation is pretty straightforward and self-explanatory. Please take a look at the source code to see how to use it.

In this exercise, you are allowed to use two data structures in the Standard Template Library: stack and list. However, you should not use other data structures in STL except the iterators of stack and list and some common exceptions such as `runtime_error`. If in doubt, please contact the instructor to ask whether a data structure in STL can be used.

We will test your implementation of binary trees using a different main function that is different from the one in `main.cpp`. We will test the completeness and correctness of your implementation, including whether your code will throw all necessary exceptions. We will also check whether your program will cause memory leak. We will also test whether `assignment2.cpp` uses the member functions in `LinkedBinaryTree` correctly by replacing `LinkedBinaryTree.h` with the instructor's implementation of `LinkedBinaryTree.h`.

The error messages in `runtime_error` exceptions do not have to be exactly the same as the messages in the sample program we provided. You can write the error messages in your own sentences. However, we will check whether you have thrown the exceptions.

To compile your program on our submission servers, use this command:

```
g++ -o exptree Symbol.cpp assignment2.cpp main.cpp
```

Before you submit your program, please check whether your program can be compiled correctly using this command on our submission server. We do not grade your program using other compilers.

Please also submit a plain text file called "`README.TXT`" to tell us the extent of your implementation, more specifically which parts have been implemented and which parts have not. Also, if there are some known bugs in your program, you should state them in the plain text file. An empty `README.TXT` is included in `prog2.zip`.

You will submit only three files: (1) **`LinkedBinaryTree.h`**, (2) **`assignment2.cpp`**, and (3) **`README.TXT`**. These files should be self-contained and you cannot submit additional files.

Please store these files in a directory called **assignment2** in the course project on Gitlab, which is set up according to our Gitlab guidelines. Please remember to push these files to Gitlab after you finish writing them. For more detail on how to use Gitlab, please read our Gitlab guidelines on our course website. Please email **Sangwoo Ha** at **swha@unist.ac.kr** if you fail to upload your files to Gitlab.

Automatic Grading and Late Submission Penalty

We developed shell scripts to grade your programs automatically without human intervention. In the past, many students didn't read this handout carefully and implemented their programs with wrong function names or filenames, etc., causing compilation errors. Our TAs spent too much time to help students to fix these issues in the past. In order to avoid wasting our TAs' time, we decide to give students opportunities to see the results of the grading script running with their programs *before* the deadline, such that students can fix any issues *themselves* before the final submission. We will start running the grading scripts at every midnight at least three days before the deadline. If you submit your program earlier, you will see the grading report generated by the grading script in the home directory in your account. Please fix any problems in your program according to the grading report, especially those that are caused by incorrect naming. *There will be a*

10% penalty if our TA needs to help you to fix any issues related to the submission and automatic grading after the deadline.

The scores in the pre-deadline grading reports are not final and they will not be recorded. However, the grading reports after the deadline are final. In fact, we will grade your programs three times after the deadline in order implement our late submission policy as described on our course webpage. The first real grading report will be generated right after the deadline. The second real grading report will be exactly one day after the deadline with a penalty of 15%. The third real grading report will be exactly third day after the deadline with a penalty of 30%. We will select the highest scores among these three real grading reports as your final score of this assignment. Clearly, if you do not update and resubmit your program after the deadline, the highest score will be the one right after the deadline. However, if you are not happy with the score on the first real grading report, you still have the chance to improve it by submitting your programs again. Notice that late submission will never decrease your final score.

Our grading script is evolving and hence we reserve rights to regrade your programs with a different script in a later time. Hence, the scores you see in the grading reports may not be final. Please report any bugs in our grading script if you find them.

Bug Reports

Please report any bugs in the codes we provide as well as any typos and errors in this handout and the grading reports to **Prof. Tsz-Chiu Au** at **chiu@unist.ac.kr**. We will look into the bug reports and fix the problems. For any other problems, please do not contact the instructor directly; instead, please email our TA who should be able to help you. Please cc the email to the instructor so that he also knows what is going on between you and the TA. Notice all emails between you, the TA, and the instructor should be written in English.

If we have to release a new version of the codes to fix the bugs, we will announce it on our course webpage. Before you submit your program, you should check the announcement to see whether the codes have been updated. Please make sure that your program works with the latest version of the codes we provide.