CS-202

Data Structures

Assignment 2

Due: 11:55pm on Monday, March 8, 2021

In this assignment, you are required to implement a general tree, BST, and AVL. The last part requires you implement an expression tree. You may test your implemented data structures using the test cases provided to you. Please note that your code must compile at the mars server under the Linux environment.

Plagiarism Policy

The course policy about plagiarism is as follows:
1. Students must not share the actual program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students cannot copy code from the Internet.
4. Students must indicate any assistance they received.
5. All submissions are subject to automated plagiarism detection. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Late Submission Policy

You are allowed 5 "free" late days during the semester (that can be applied to one or more assignments; the final assignment will be due tentatively on the final day of classes, i.e., before the dead week and cannot be turned late. The last day to do any late submission is also the final day of classes, even if you have free late days remaining). If you submit your work late for any assignment once your 5 “free” late days are used, the following penalty will be applied:

- 10% for work submitted up to 24 hours late
- 20% for work submitted up to 2 days late
- 30% for work submitted up to 3 days late
- 100% for work submitted after 3 days (i.e., you cannot submit your assignment more than 3 days late after you have used your 5 free late days)

PART 1
GENERAL TREES

In this part you will be implementing a general tree data structure where each node of a tree can have any number of children nodes. Refer to `tree.hpp` which provides all the required definitions. Write your implementation in `tree.cpp` using the boiler code.

Member Functions[1]:

Write implementation for the following methods as described here:

`Tree(shared_ptr<node<T,S>> root)`

- Constructor

`shared_ptr<node<T,S>> findKey(T key)`

- Finds the node with the given key and returns a pointer to that node. `NULL` is returned if the key doesn't exist.

`shared_ptr<node<T,S>> findKeyHelper(shared_ptr<node<T,S>> currNode, T key)`

- Helper function to be used in `findkey(T key)` function

`bool insertChild(shared_ptr<node<T,S>> newNode, T key)`

- Inserts the given node as the child of the given key. Returns true if insertion is successful and false if key doesn't exist. Insertion should also fail if another node with the same key as the new node already exists i.e. duplicates are not allowed.
- If the node at the given key already has children, the new node must be added to the tail of the children of that node.

`vector<shared_ptr<node<T,S>>> getAllChildren(T key)`

- Returns all the children of the node with the given key. Should return an empty vector in case the node has no child or key doesn't exist.

`int findHeight()`

- Returns the height of the tree

`int findHeightHelper(shared_ptr<node<T,S>> currNode)`

[1] In this assignment, you are free to declare any helper functions as per your need but the already declared function declarations should not be modified.

- Helper function to be used in the `findHeight()` function

`void **deleteTree**(shared_ptr<node<T,S>> currNode)`

- Delete the entire tree.

`bool **deleteLeaf**(T key)`

- Delete node with given key if and only if it is a leaf node i.e. have no child. Doesn't delete the root node even if it is the only node in the tree. Returns true on success, false on failure.

`shared_ptr<node<T,S>> **deleteLeafHelper**(shared_ptr<node<T,S>> currNode, T key)`

- Helper function to delete the leaf node.

PART 2
BST AND AVL TREES

In this part, you will be implementing a single class which will serve for both BST and AVL trees. Refer to `avl.hpp` file for the class definitions. Write your implementation in `avl.cpp` using the boiler code.

Member functions:

Write implementation for the following methods as described here.

`AVL(bool isAVL)`

- Simple default constructor which initializes the `isAVL` flag. If the `isAVL` flag is set, insertions and deletions will follow AVL property. Otherwise, it will be a simple BST.

`void insertNode(shared_ptr<node<T,S>> N)`

- Inserts the given node into the tree such that the AVL (or BST) property holds.

`void deleteNode(T k)`

- Deletes the node with the given key such that the AVL (or BST) property holds.

`shared_ptr<node<T,S>> searchNode(T k)`

- Returns the pointer to the node with the given key. `NULL` is returned if the key doesn't exist.

`shared_ptr<node<T,S>> getRoot()`

- Returns the pointer to the root node of the tree.

`int height(shared_ptr<node<T,S>> p)`

- Returns the height of the tree.

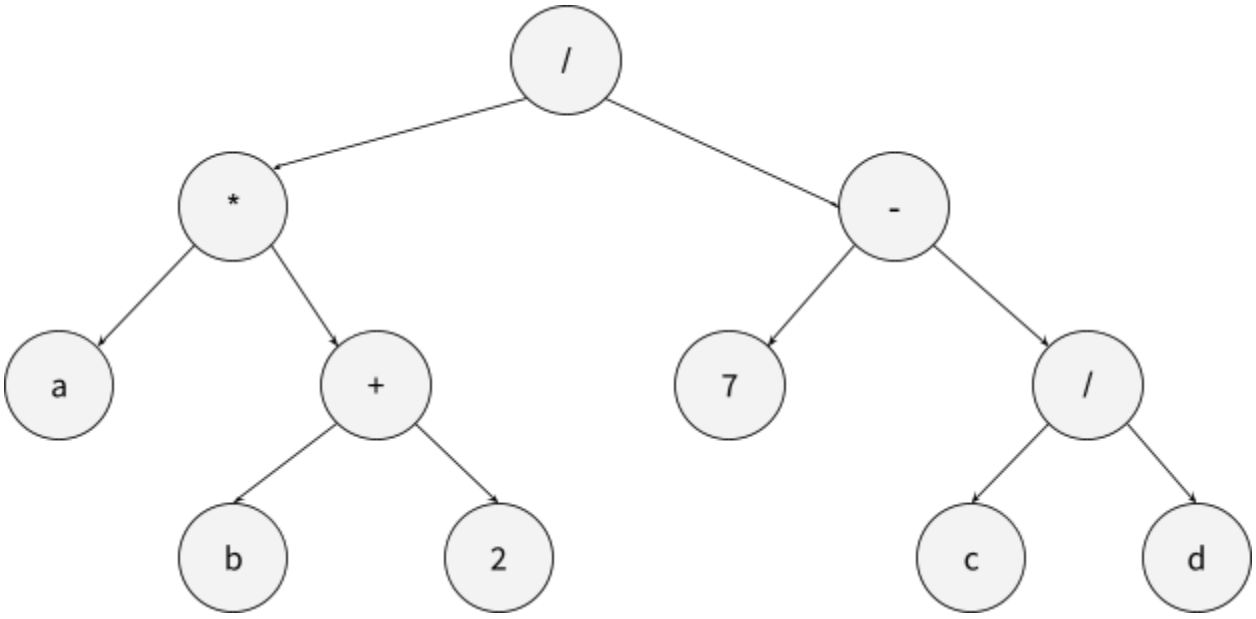
Note:

- In Part 2 of the assignment, the height of a tree is the number of nodes in the path from the root to its deepest descendant. For example, a tree with only two nodes (a root and a child) will have height 2 and so on.
- As a convention, while deleting a node with 2 children, the new root should be selected from the right subtree.

PART 3
EXPRESSION TREE

An important application of trees is calculating expressions. Take for example the following expression: `(a*(b+2))/(7-c/d)`.

This expression has varying levels of precedence as defined by PEMDAS. To calculate such an expression with the right precedence, we can build a binary expression tree:



This tree representation motivates three ways to write this expression:

Infix:

`a*b+2/7-c/d`

Postfix:

`ab2+*7cd/-/`

Prefix:

`/*a+b2-7/cd`

These are the inorder, postorder and preorder traversals of the tree respectively.

The infix notation is our conventional way to write expressions where the operator is placed between the operands. This necessitates addition of parenthesis to denote precedence. On the contrary, in postfix notation, the operator is placed after the operands;

and in prefix notation, the operator is placed before the operands. In both of these notations, each operation is sequentially written in its correct order of precedence, liberating us from the necessity to use parentheses. There are a lot of tools online, like this, to help you play with and internalize these different notations.

In this part of the assignment, we will explore the use of expression trees. Let's head right in!

You will be implementing the `ExpTree` class in `exp_tree.cpp`, defined in `exp_tree.hpp`.

3.1 Building the Expression Tree from Postfix Notation:

Parsing a conventional infix expression to build an expression tree is a really fun problem. Unfortunately for us, it is out of the scope of this course. We will instead be building our expression tree from a postfix notation given to the following function:

```
void buildTree(string postfix_exp, bool optimal)
```

This function takes a postfix expression and a boolean flag named `optimal`, builds the corresponding expression tree, and stores it in the class's private variable `root`. We'll discuss the `optimal` boolean flag later.

You will be using a renowned algorithm for generating the expression tree from a postfix notation. The algorithm is as follows:

1. Initialize an empty stack.
2. Go to the start of the postfix expression.
3. Read the next character.
4. If the character is a numeric value or a variable:
    a. Create a new expression tree with a single node having the numeric value or variable.
    b. Push this new tree into the stack.
5. If the character is an operator:
    a. Pop two trees (T_1 and T_2 respectively) out of the stack.
    b. Create a new expression tree with the operator at the root and trees T_1 and T_2 to its right and left respectively.
    c. Push this new tree into the stack.
6. Keep repeating from step 3 until the whole postfix expression is read.

7. By the end, the stack will only have one tree left. This will be our completed expression tree!

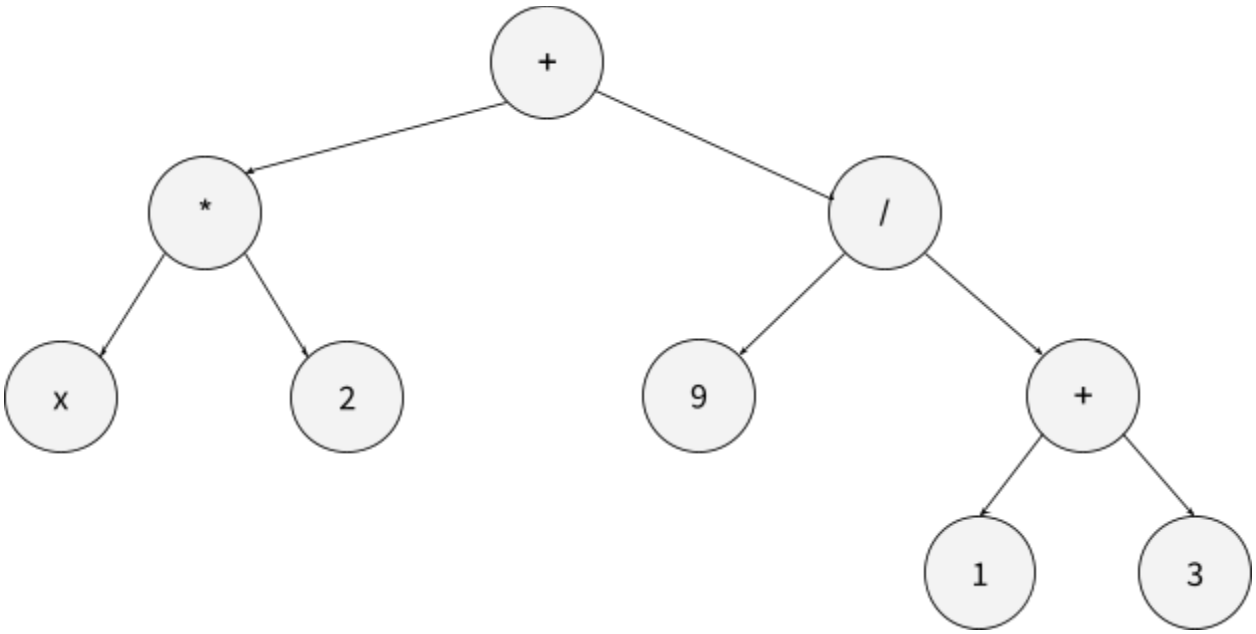
Instructions:

- Our postfix expression will only contain the operators **+**, **-**, *****, **/**, numeric values from **0** to **9**, and variables with lowercase names of length 1 from '**a**' to '**z**'. This means that values in each node of our expression tree are one character long in length.
- You should use the standard C++ stack class. Its documentation is available here, and this is a helpful guide.
- An empty postfix expression should be set to `NULL`.
- We will assume that the given postfix expression is valid. e.g. the following is not a valid postfix expression: `abc+`.

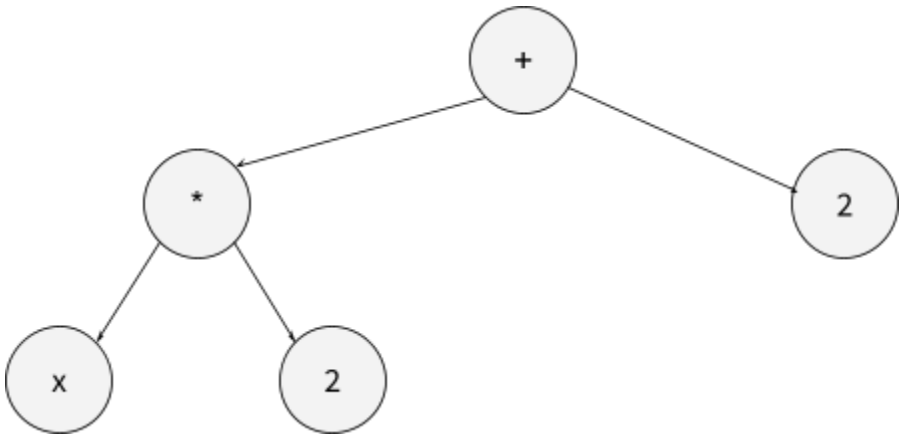
If you have implemented everything above correctly while ignoring the `optimal` flag, great work! You have secured **7** marks. Let's now look into the optimal boolean flag for the rest of the **3** marks.

When this flag is set to true, the subtrees not containing any variables are computed and their result is stored in the tree as only one node. This can save us memory space.

As an example, let's consider the postfix expression `x2*913+/+`. This will generate the following tree when the optimal flag is false:



If the optimal flag is set to true, the tree would be:



Note that the right subtree of the root was all calculated and stored as a single node with a value of `9/(1+3)=2`.

Instructions:

- While calculating, the divisions will be integer divisions.
- You should assume that at every stage of the calculation, the result will be between **0** to **9** i.e. one character long in length.
- Be wary of correctly converting characters to integers, and vice versa where need be.

3.2 Calculating Value for the Expression Tree:

In this part you will implement calculating and outputting the value of the expression tree, given the values of the variables.

This is the definition of the function you are going to implement:

```
double calculateRecurse(shared_ptr<node> currNode, vector<Variable> variables)
```

This is a function called on the root of the tree by `double calculate(vector<Variable> variables)`. `calculateRecurse` recursively calculates

and returns the value of the passed `currNode` and its descendants. `variables` is a vector of the struct `Variable` that stores the variable names and their corresponding values.

Instructions:

- While calculating, divisions should **not** be integer divisions.
- You should assume that all variables used in the expression are provided in the variables vector.
- If the expression tree is `NULL`, `-99` should be returned.

3.3 Outputting the Infix Notation with Parentheses:

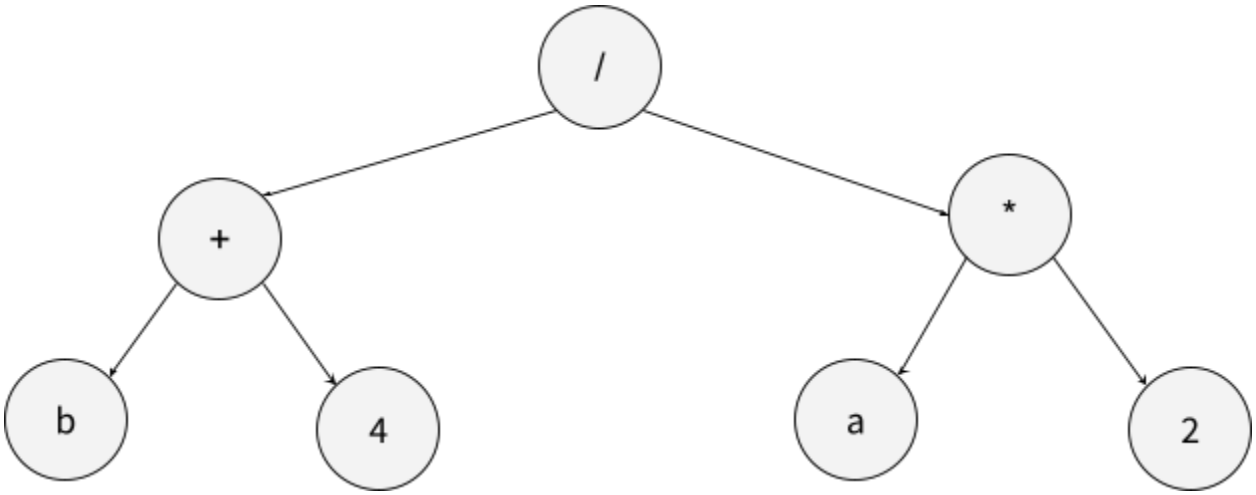
Lastly, you would be implementing a function that generates the infix notation of the expression tree while adding parenthesis.

This is the definition of the function you are going to implement:

```
string getInfixRecurse(shared_ptr<node> currNode)
```

This is a function called on the root of the tree by `string getInfix()`. `getInfixRecurse` recursively returns the infix notation of the `currNode` and its descendants while adding parentheses.

For example take the following tree:



The infix notation corresponding to this tree should `(b+4)/(a*2)`. As another example, let's take the tree made by the postfix expression `a5+b+a+6-1-d+`. This tree will have an infix expression of `(((((a+5)+b)+a)-6)-1)+d`.

Instructions:

- If the expression tree is `NULL`, an empty string should be returned.

Testing

To test the implementation of your code, compile and run the following test files:

Part	Test File
Part 1: General Tree	test1.cpp
Part 2: BST	test2.cpp
Part2: AVL Tree	test3.cpp
Part 3: Expression Tree	test4.cpp

Use the following commands:

Compile: `g++ test<test_number>.cpp -std=c++11`
Run: `./a.out`

Each test case has been divided into subcases to give an idea about any unhandled corner cases.

Marks Distribution

Part	Marks
Part 1: General Tree	30
Part 2: BST and AVL Tree	40
Part 3: Expression Tree	30

Submission Guidelines

Zip the complete folder and use the following naming convention:

PA1_<roll_number>.zip

For example, if your roll number is 22100277, then your zip file name should be:

PA1_22100277.zip