COMP 2140 Winter 2019
Assignment 3: Binary Trees

Due: Thursday 7 March 2019 at 11:59 p.m. CDT (Winnipeg Time)

Important Announcement

If you haven't previously agreed to the honesty declaration (see the instructions at the end of this assignment under "Honesty Declaration"), you cannot hand in any assignments — that is, until you agree to the honesty declaration, you will not be able to see the assignment dropboxes in UMLearn. (You only have to agree ONCE per term — the agreement covers all your work in this course.)

How to Get Help

Your course instructor: Email, phone or drop in to office hours. For email, please remember to put "[comp2140]" in the subject, and to send from your UofM email account.

Course discussion groups on UM Learn: A discussion group for this assignment is available in the COMP 2140 course site on UM Learn (click on "Discussions" under "Communications"). Post questions and comments related to Assignment 3 there, and I will respond. Questions sent via email will be answered on the discussion group so that all students can benefit. Please do not post solutions, not even snippets of solutions, here or anywhere else. I strongly suggest that you read the assignment discussion group for helpful information.

Computer science help centre: The staff in the help centre can help you (but not give you assignment solutions!). See their website at `http://www.cs.umanitoba.ca/undergraduate/computer-science-help-centre.php` for location and hours. You can also email them at `helpctr@cs.umanitoba.ca`.

Programming Standards

When writing code for this course, follow the programming standards available at the end of the course syllabus in UMLearn. Failure to do so will result in the loss of marks. (Yes, you should write private "helper" methods as needed to make your methods (including the `main` method) readable.)

The use of Java Collections is not allowed in COMP 2140 assignments. Java Collections (java.util.List, ArrayList, Vector, etc.) should not be used in any assignments in this course. Because this is a course on data structures, you must implement all data structures yourself. You should not be using any Java Collections (such as ArrayList) where methods are provided and the structure is hidden. You must demonstrate that you can build your programs from the ground up, using basic datatypes to build objects.

Initial Steps

In this assignment you will work with binary trees. In question 1 you will work with a simple binary search tree where each node contains an integer (key) data field. In question 2 you will work with expression trees, used to store and evaluate arithmetic expressions.

Read Chapter 8 in Lafore, which discusses binary trees. You should also review the "Parsing Arithmetic Expressions" discussion in Chapter 4. Unit 6 online readings are also useful. In particular, the "Introducing Trees" online notes in Unit 6 include a discussion of expression trees (including how to solve a postfix expression using a stack, and how to solve a prefix expression using a queue).

In completing the questions below, you will need Node and Tree classes, as well as Stack and Queue classes. The Stacks and Queues that you implement will hold Nodes (further details below), and **must** be implemented by you. You may **not** use any existing Java classes or Collections (java.util.List, ArrayList, LinkedList, Vector, Stack, etc.). You must demonstrate that you understand the underlying structure of a Stack, Queue, and Tree by developing your own classes.

It is strongly recommended that you work with pencil and paper to develop your algorithms in this assignment. Draw some sample trees on paper. As you build and modify trees, carefully keep track of all references to Nodes.

Question 1: A Breadth-First Traversal of a Binary Tree

You will use a queue to assist you with performing a breadth-first traversal of a binary tree.

The Tree Class

The recommended starting point for this question is Listing 8.1 in Chapter 8 of Lafore.

- Copy the tree.java code (Listing 8.1, page 406-415). Example programs from Lafore are available on the Downloads tab at `http://www.informit.com/store/data-structures-and-algorithms-in-java-9780672324536`. Be sure to add a credit to the author in the .java file that you will submit.
- Make sure that you understand the methods in the above code, and post on the course discussion forum if you have any questions.
- Modify the code as appropriate so that the Nodes contain only the integer (key) data field.
- Read through the remainder of this question before proceeding.

A Queue to store Nodes

Create a Queue class where each item stored in the Queue is a Node.

You may choose the underlying implementation for your queue that you think is most appropriate (e.g. linked list or array). The implementation that you choose should be hidden from a user of the class. That is, the user will enqueue and dequeue Nodes, and will not know how they are managed inside the queue.

Your queue must have the following methods:

- A **constructor** that creates an empty queue.
- A `boolean isEmpty()` method that returns a boolean, indicating whether the queue is empty.
- A `boolean enqueue(Node toAdd)` method that will insert the given Node into the queue. This method will return true if the enqueue is successful, and return false if the enqueue fails.
- A `Node dequeue()` method that will dequeue and return the Node at the front of the queue. This method removes the returned Node from the queue. This method should return `null` if the user tries to dequeue from an empty queue.
- A `Node peek()` method that returns the Node at the front of the queue. This method does **not** remove the returned Node from the queue. This method should return `null` if the user tries to peek at an empty queue.

A Breadth-First Traversal of the Tree

Code listing 8.1 included three types of depth-first traversals (preorder, inorder, postorder). You will implement a breadth-first traversal, that prints the data in the Nodes, in the order that you visit each Node.

A breadth-first traversal is one in which all of the nodes on a particular level of the tree are traversed (left to right) before descending to a lower level of the tree. A queue is used to keep track of the nodes so that they are visited level-by-level.

The exact details of this method are intentionally not provided and you must use your knowledge of trees and queues to develop a working algorithm.

In the comments for your breadth-first method, in one paragraph, describe the algorithm you use to perform the breadth-first traversal.

Some hints:

- The queue is what makes this type of traversal possible.
- You will start by adding the root node of your tree to a queue of nodes.

- You will print the data as Nodes are removed from the queue.
- Note that while the three depth-first traversals are implemented using recursion, you will not use recursion in this method and the breadth-first traversal will look entirely different.
- *If you choose to use a linked list implementation for the queue, be careful when setting references to Nodes in the list. The nodes in the queue's linked list will in turn hold nodes from your tree (i.e. you need a second type of node that holds the tree nodes). Each tree node in your queue must retain its references to its children.*

Your Main Class (please name it `<your last name><your first name>A3Q1`)

Your main method should perform the following:

- Insert 20 random integers (with values between 1 and 999) in a binary search tree.
- Print the tree using a preorder traversal.
- Print the tree using an inorder traversal.
- Print the tree using a postorder traversal.
- Print the tree using a breadth-first traversal.

Additional Notes

- Please place all classes for question 1 in the same file.
- Submit ONE .java file containing all the source code for question 1. The .java file MUST be named `<your last name><your first name>A3Q1.java` (e.g. `SmithJohnA3Q1.java`). Do not submit any output. Your code will be run during marking. Make sure that your code compiles without errors.

Question 2: Expression Trees

Read through all of the instructions for question 2 before beginning.

You will use an expression tree to store and manipulate algebraic expressions. Your expression tree will be constructed from nodes, as described below.

Examples of expression trees are shown in Figure 8.11 of Lafore, and in "Introducing Trees" in the Unit 6 notes. Some additional examples of expression trees are

```
   +                  *                     *
  / \                / \                   / \
 A   -              A   ^                 +   -
    / \                / \               / \ / \
   B   C              B   +             A  B C  D
                         / \
                        C   D


 (A+(B-C))        (A*(B^(C+D)))         ((A+B)*(C-D))
```

One advantage of postfix and prefix notations is that parentheses are not needed to specify order of operations. In this question you will create expression trees from both postfix and prefix forms of algebraic expressions. You will also simplify the expression trees and print infix, postfix, and prefix versions of the expressions stored in the trees. Your program will read commands from a file and output the result of executing those commands.

Input

The input file will consist of one command per line, where the command will be one of the 6 options below. If additional information is needed, it will follow on the same line. Possible commands are:

- **COMMENT** – A line beginning with "COMMENT" should be echoed to the screen. No manipulation of the tree is required.
- **NEW** – A line beginning with "NEW" will construct a new expression tree. Any existing tree will be discarded and replaced with the new tree. The expression that follows "NEW" will be either prefix or postfix notation. Your program should be able to determine which notation is used. All operands and operators will be separated by spaces, so that you may split the line read from file into tokens, where each token contains a String that will correspond to a node in the tree.

  Output a single statement, "New tree constructed", when a tree is successfully created.

  Please see the sections below on constructing expression trees for some hints on how to read in expressions and construct trees.
- **PRINTPREFIX** – A line beginning with "PRINTPREFIX" will print the current tree using prefix notation. Similar to the prefix notation used when reading from the input file, separate all operands and operators with a space.
- **PRINTPOSTFIX** – A line beginning with "PRINTPOSTFIX" will print the current tree using postfix notation. Similar to the postfix notation used when reading from the input file, separate all operands and operators with a space.
- **PRINTINFIX** – A line beginning with "PRINTINFIX" will print the current tree using infix notation. Infix notation requires parentheses to indicate order of operations, and you will print a **fully parenthesized expression**, where each operand-operator-operand is enclosed in parentheses. Examples of fully parenthesized expressions are `( ( 8 + 6 ) * ( 4 - 5 ) )` and `( ( ( B + 5 ) * 4 ) ^ 3 )`.
- **SIMPLIFY** – A line beginning with "SIMPLIFY" will simplify the current tree as much as possible, following common arithmetic rules, and output the statement "Tree simplified" when complete. Traverse the tree and stop to consider each node containing an operator. Look at the subtree consisting of the operator node and its left and right subtrees, and simplify where possible. Some examples to get you started:
  - If both children of an operator are numeric values, perform the operation and replace that subtree with the result stored in a single node.
  - A * 1 = A. If the operator is * and one of the children is 1, replace the subtree with the other child.
  - A * 0 = 0. If the operator is * and one of the children is 0, replace the subtree with 0.
  - A ˆ 1 = A. If the operator is ˆ and the right child is 1, replace the subtree with the left child.

Nodes for Expression Trees

The nodes in the expression tree will have the ability to hold three types of information: operators, variables, or constants. Possible operators will be +, -, *, and ˆ. We will omit division so that we can deal only with integers. Variables will be a string of any number of alphabetic characters (e.g. A, first, cat). Constants will be integers.

The fields in your nodes should be:

- A type that identifies the node as an operator, variable, or a number. Something similar to `enum NodeType{OPERATOR, VARIABLE, NUMBER;}` is appropriate.
- A character that will store the operator for operator nodes. The only valid characters are '+', '-', '*', and 'ˆ'.
- A String that will store the variable name for variable nodes.
- An integer that will store the value for number nodes.
- Two Node references, that refer to the left child and the right child. For leaf nodes these will be `null`.

The Expression Tree Class

Your expression tree class must have methods that allow it to carry out the expected commands. Recursion should be used as appropriate. While the tree class from question 1 gives you an idea of how to manipulate tree objects, note that expression trees are sufficiently different that you should write a new class rather than trying to modify the class from question 1.

A Queue to store Nodes

You will need a queue of nodes to aid in the construction of an expression tree from prefix notation. This queue should be similar to the queue in question 1, except that the nodes stored are now nodes from your expression tree rather than nodes that contain only an integer.

Hint: *If you name your node classes "Node" in both questions, and use good coding practices, you should be able to use the Queue class from question 1 without modification.*

A Stack to store Nodes

You will need a stack of nodes to aid in the construction of an expression tree from postfix notation. Create a Stack class where each item stored in the Stack is a Node.

As for the Queue class, you may choose the underlying implementation for your stack that you think is most appropriate (e.g. linked list or array). The implementation that you choose should be hidden from a user of the class. That is, the user will push and pop Nodes, and will not know how they are managed inside the stack.

Your stack must have the following methods:

- A **constructor** that creates an empty stack.
- A `boolean isEmpty()` method that returns a boolean, indicating whether the stack is empty.
- A `boolean push(Node toAdd)` method that will insert the given Node into the stack. This method will return true if the push is successful, and return false if the push fails.
- A `Node pop()` method that will pop and return the Node on the top of the stack. This method removes the returned Node from the stack. This method should return `null` if the user tries to pop from an empty stack.
- A `Node peek()` method that returns the Node on the top of the stack. This method does **not** remove the returned Node from the stack. This method should return `null` if the user tries to peek at an empty stack.

Constructing An Expression Tree From Postfix Notation

Constructing an expression tree from postfix notation is discussed in Lafore, on page 387-388. Also see the comments on solving postfix expressions in "Introducing Trees" of the Unit 6 online notes.

Using a stack of nodes to temporarily store subtrees, you will construct an expression tree from postfix notation. As you process the expression from the input file, consider one token at a time, create a node of the appropriate type and proceed as appropriate, either (i) pushing the node onto the stack or (ii) popping nodes, creating a larger subtree, and pushing that onto the stack.

Note that if you push the root of a subtree onto the stack and the child references are appropriately set, you are effectively pushing the entire subtree onto the stack.

In the comments of your postfix-to-tree method, in one paragraph, describe the algorithm you use to construct an expression tree from postfix notation.

Constructing An Expression Tree From Prefix Notation

To constuct an expression tree from prefix notation, make use of a queue of nodes to temporarily store subtrees as you build the full tree. See the comments on solving prefix expressions in "Introducing Trees" of the Unit 6 online notes to help you get started on designing your algorithm.

Hint: Note that while operator nodes should always have two non-null children in a valid tree, you will sometimes want to queue operator nodes temporarily without children while building the tree.

In the comments of your prefix-to-tree method, in one paragraph, describe the algorithm you use to construct an expression tree from prefix notation.

Your Main Class (please name it `<your last name><your first name>A3Q2`)

Write a main method that will process commands, as described above, until the end of file. You may assume that the data file will be named **A3Q2input.txt**.

Sample program input:

```
COMMENT Starting tests...
NEW C 3 + 5 4 - *
PRINTINFIX
SIMPLIFY
PRINTINFIX
PRINTPOSTFIX
PRINTPREFIX
COMMENT End of tests.
```

Sample program output:

```
Starting tests...
New tree constructed
( ( C + 3 ) * ( 5 - 4 ) )
Tree simplified
( ( C + 3 ) * 1 )
C 3 + 1 *
* + C 3 1
End of tests.
```

Additional Notes

- You may assume that the input file is free of errors. That is, you may assume that the file contains one command per line, that all tokens within a line will be separated by a space, and that only valid commands are provided. You may also assume that given expressions are valid expressions (i.e. the number and order of operands and operators are such that they construct a valid algebraic expression).
- Place all classes for question 2 in the same file.
- Submit ONE .java file containing all the source code for question 2. The .java file MUST be named `<your last name><your first name>A3Q2.java` (e.g. `SmithJohnA3Q2.java`). Do not submit any output. Your code will be run during marking. Make sure that your code compiles without errors.

Hand-in Instructions

Go to COMP2140 in UM Learn, then click “Dropbox” under “Assessments” at the top. You will find a dropbox folder called “Assignment 3”. (If you cannot see the assignment dropbox, follow the directions under “Honesty Declaration” below.) Click the link and follow the instructions. Please note the following:

- Submit ONE .java file per question. The .java file must contain all the source code. The .java files must be named `<your last name><your first name>A3Q1.java` (e.g. `SmithJohnA3Q1.java`) and `<your last name><your first name>A3Q2.java` (e.g. `SmithJohnA3Q2.java`).
- Please do not submit anything else.
- Do NOT zip your files. Attach the two .java files to your submission.
- You can submit as many times as you like, but only the most recent submission is kept.

- We only accept homework submissions via UM Learn. Please DO NOT try to email your homework — it will not be accepted.
- We reserve the right to refuse to grade the homework or to deduct marks if you do not follow instructions.
- **Assignments become late immediately after the posted due date and time.** Late assignments will be accepted up to 49 hours after that time, at a penalty of 2% per hour or portion thereof. After 49 hours, an assignment is worth 0 marks and will no longer be accepted.

Honesty Declaration

To be able to hand in any assignment (i.e., to make the dropbox visible to you), you must first agree to the blanket honesty declaration. (This task only needs to be done ONCE during the course, not every time you hand in an assignment.) This declaration covers ALL your work in the course. To agree to the honesty declaration:

- Go to the COMP 2140 homepage on UM Learn, look at the navigation bar at the top. Under "Assessments", you will see "Checklist" — click on "Checklist".
- Click on the blue "Honesty Declaration" at the top.
- Read the honesty declaration, check off "Agreement" (verifying that you have read the honest declaration and agree to it), and finally, click "Save". Only after this task is completed will you be able to see any assignment dropboxes.