Data Structures (CS 102), Professor Yap – Fall 2019

HOMEWORK

November 11, 2019

Homework 6: Due on Tue Nov 19, 2019

- Carefully read the Homework Policy in the class wiki. Pay attention to the rules for submitting programs in a zip file (not rar or other formats).

PART A: WRITTEN ASSIGNMENT

A.1 (5 Points)
Consider the BSTs

$$T_1 = 2(1)[4(3)[5]], \qquad T_2 = 4(1(\ )[3(2)[\ ]])[5].$$

Here we are using a special **parenthesized representation** of BST's (see below). Note that T_1 and T_2 are equivalent as they both contain the set $\{1, 2, 3, 4, 5\}$ of keys. First draw both T_1 and T_2. Then give a sequence of rotations to transform T_1 to T_2. Use as few rotations as possible. Since the keys are unique, we may say "$rot(2)$" to mean rotate the node whose key is 2.

NOTES on parenthesized representation of BST: If T is a BST rooted at a node u, its parenthesized representation is a string of denoted by $E(u)$. This is defined recursively as follows:
If u is null, then $E(u)$ is just the empty string "" (nothing in the string).
If u is non-null but has no children, then $E(u) =$ "$u.key$" (note that there are no parentheses in this case).
Otherwise,

$$E(u) = \text{"}u.key(E(u_L))[E(u_R)]\text{"}$$

where u_L, u_R) are the left and right child of u. Note that we use two kinds of parenthesis: (..) for the left child, and [..] for the right child. E.g., $E(u) =$ "2()[3]" if $u.key = 2$, u_L is null and $u_R.key = 3$ (and u_R is a leaf). But usually, we omit the double-quotes and simply write $E(u) = 2()[3]$. Moreover, we treat the keys as if they are nodes. If we rotate at 3, we get the equivalent BST 3(2)[ ] (rooted at 3 with left child 2).

A.2 (12 Points)
Suppose we have a binary tree T in which each node stores a key. But T is not necessarily a BST. Describe a method called $fix(u)$ which will convert the BST rooted at u into a BST. However, $fix(u)$ must not change the shape of the tree.

Please describe your algorithm using in Pseudo-Code. The only operation on T which you can use is "$swapKey(u)$" which swaps the key of u with that of its parent. If u has no parent, then $swapKey(u)$ is a no-op (nothing happens). We assume that nodes have parent pointers.

NOTE: "Pseudo-code" includes specifying any loop structure and has enough detail for any one who knows Java ("but not much else") to be able to implement your method. It means "coding in English".
PLEASE DO NOT GIVE US ANY JAVA CODE.

A.3 (6 Points) Given a node u (representing a singly linked list), describe in pseudo-code a method called $check(u)$ that returns true iff u represents a list (and not a rho-list).

A.4 (8 Points) Given a node u (representing the head of a doubly-linked list). Describe in pseudo-code a method $check2(u)$ that returns true iff u represents a doubly-linked list.

PART B: PROGRAMMING (70 Points)

You have two programs to write. A large part of this exercise is to read and understand two large pieces of code (our implementation of BST and Eval). Then you are asked to modify them.

B.1 We have provided a BST implementation in the file called `BST.java`. It uses an internal class `Bnode2` that has parent pointers. Please re-implement it using a binary node class (call it `Bnode`) that does not have parent pointers. Your file should be called `BST1.java`. *You DO NOT have to re-implement the* `successor` *and* `predecessor` *methods, as these algorithms are a bit subtle.*

You can test both of these files in our Makefile by calling the targets `testBST` and `testBST1`. Their results should agree!

Please do not change the main method or the headers of the methods. So we suggest that copying `BST.java` to `BST1.java` first.

B.2 We have provided an implementation of evaluating arithmetic expressions in the file called `Expr.java`. Note that it extends `ExprAbstract` class, and uses our own implementations of stacks and queues (`MyQueue, MyStack`). Given an arithmetic expression represented by a string S, we want to evalute it using $eval(S)$. Recall from lectures that evaluation can be broken down into three phases:

- Phase 1 is **parsing**, which converts the input string into a sequence of tokens which is stored in an input queue inQ, with the help of a stack. E.g., "$(1 + 20) - 3$" is parsed into $inQ = (\text{'('}, 1, +, 20, \text{')'}, -, 3)$.
- Phase 2 is **conversion** that transforms the tokens in inQ into a postfix (a.k.a. Reverse Polish) notation, and stores them in an output queue $outQ$. E.g., $inQ = (\text{'('}, 1, +, 20, \text{')'}, -, 3)$ is converted into $outQ = (1, 20, +, 3, -)$. Although inQ may have parentheses, but $outQ$ has no parentheses.
- Phase 3 is **postfixEval**, evaluates the contents of $outQ$ (again, with the help of a stack).

We want you to write a class class called `ExprX` which is an extension of the `Expr` class:

- The expressions in `Expr` only allow integer operands. We want `ExprX` to allow decimal operands. E.g., we allow the expression "$1.23 + 4.5$" Note that this will evaluate to approximately 5.73, but not exactly. That is OK (do you know why?).
- `ExprX` also allows a new operator, division '$\div$'.

- The value of expressions in `ExprX` are represented by `double` values. When you print these values, please print only 3 decimal digits. E.g. "$1 \div 3$" should finally print as 1.333 (an approximation to one third).

You can test both of these files in our Makefile by calling the targets `testExpr` and `testExprX`.