Homework 5

Data Structures & Algorithms (Term III/2018–19)

built on 2019/05/31 at 20:30:53 ***due:*** *fri jun 7 @ 11:59pm*

This assignment contains both a coding portion and a written portion. The aim of this assignment is to provide you with more experience solving algorithmic problems in Java and reasoning about code. This assignment requires a starter package, which can be downloaded from the course website.

READ THIS BEFORE YOU BEGIN:

- All your work will be handed in as a single zip file. Call this file `a5.zip`. You'll upload this to Canvas before the assignment is due.
- For the written part, you *must* typeset your answers and hand it in as a PDF file called `hw5.pdf`, which will go inside your zip file. No other format will be accepted. To typeset your homework, apart from Microsoft Word, there are LibreOffice and LaTeX, which we recommend. Note that a scan of your handwritten solution will not be accepted.
- Be sure to **disclose your collaborators in the PDF file.**
- For each task, save your work in a file as described in the task description.
- A script will process and grade your submission before any human being looks at it. *Do not use different function/file names.* The script is not as forgiving as we are.
- Use the Internet to help you learn: It's OK to look up syntax or how a function/class/method is used. It's **NOT** OK to look up how to solve a problem or answers to a problem. The goal here is to learn, not to just hand in an answer. If you wish to do that, just give us a link to the solution!
- You are encouraged to work with other students. However, **you must write up the solutions separately on your own and in your own words**. This also means you must not look at or copy someone else's code.
- Finally, the course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online on Canvas discussion.

Collaboration Policy: To facilitate cooperative learning, you are permitted to discuss homework questions with other students provided that the following whiteboard policy is respected. A discussion may take place at the whiteboard (or using scrap paper, etc.), but no one is allowed to take notes or record the discussion or what is written on the board. *The fact that you can recreate the solution from memory is taken as proof that you actually understood it, and you may actually be interviewed about your answers.*

Exercise 1: Mathematical Truths (2 points)

Remember binary trees from class? You will prove some propositions about them using a proof method of your choice:

1. **Proposition A:** Every binary tree on n nodes where each has either zero or two children has precisely $\frac{n+1}{2}$ leaves.
2. **Proposition B:** Every binary heap tree (see lecture notes) with n nodes has height at most $O(\log n)$.

Exercise 2: Higher-Order Merge (2 points)

For this task, save your code in `MultiwayMerge.java`

This is a standard problem in data processing. You could find solutions everywhere on the Internet; please resist the urge to look them up :)

Say you have k `LinkedList<Integer>` instances, each of which is ordered from small to large, though there may be duplicates. Your goal is to merge these lists to create a combined sorted list. You will implement a function

```
public static LinkedList<Integer> mergeAll(LinkedList<Integer>[] lists)
```

that takes an array of `LinkedList<Integer>`'s and returns a `LinkedList<Integer>` that is the combined list in sorted order (small to large).

Your algorithm must run in $O(N \log k)$, where N is the sum of the list lengths and $k =$ `lists.length`. Keep in mind, accessing (i.e., reading from or writing to) a linked list is cheap at the front and the back of the list; however, accessing it anywhere else is generally expensive. The cost is proportional to how far it is from the end. *(Hint: Use a PriorityQueue.)*

Exercise 3: Expression Monger (6 points)

For this task, save your code in `ExprMonger.java`

In class, we saw a two-stack solution for evaluating a *fully-parenthesized* expression. In this problem, you will extend the solution presented in class to additionally support exponentiation `**`. You will also extend it to support expressions that aren't fully parenthesized. To accomplish this, you will work in two steps, broken down into a few subtasks below. *Throughout this problem, assume all numbers are floating-point numbers*, so the expression `5/2` evaluates to 2.5 (unlike in Python or Java). Moreover, it is guaranteed that all the expressions your functions will be tested on evaluate to a number; we will **not** ask for the result of `4/0`, for example.

To help you work through this problem, we're supplying a function in `Utils.java` that takes a string representing an expression and returns a `List<String>` of string tokens. For instance (abusing Java's list notation):

- `Utils.tokenize("1+41*2") == ['1', '+', '41', '*', '2']`
- `Utils.tokenize("1-3*2**2") == ['1', '-', '3', '*', '2', '**', '2']`

Subtask I: Implement a function

```
public static double evalFullyParenthesized(List<String> tokens)
```

takes an expression represented as a list of fully-parenthesized tokens (e.g., obtained from `tokenize` above) and returns the result of evaluating that expression. This function must support $+, -, *, /, **$ and an arbitrary nesting of parentheses. Remember that the algorithm we worked out in class doesn't support exponentiation (`**`).

As a few examples:

- `evalFullyParenthesized(['(', '1', '+', '4', ')'])==5`

- `evalFullyParenthesized('(','(', '(', '2', '*', '6', ')', '**', '3', ')', '/', '10',')'])==172.8`

Subtask II: Most expressions we write and encounter in real life, however, are *not* fully parenthesized. Early in life, we learned and have internalized certain precedence rules that make writing, for example, `1 + 3*5` unambiguous—it is $1+(3\times 5)=16$. Similarly, we know that result of `1 + 3 * 5 - 21*2/7` is 10.0. The rules of precedence, or what many of us remember as the acronym PEMDAS, state that

- **P**arentheses have the highest precedence. This means, $2*(5-1)=2\times 4=8$.
- **E**xponentiation has the next highest precedence $2*1**3+1==((2\times(1^3))+1=3$.
- After that, **M**ultiplication and **D**vision have the same precedence;
- Below that, **A**ddition and **S**ubtration have the same precedence;
- Finally, operators with the same precedence are evaluated left to right.

For this subtask, you are to implement a function

```
public static List<String> augmentExpr(List<String> tokens)
```

that takes a list of tokens and returns a list of tokens that is a fully-parenthesized version of the input expression. The format of the input list is similar to the output of the `tokenize` function from `Utils`. The input list may already be partially or fully parenthesized.

You understand intuitively what fully-parenthesized expressions are. To be precise, we give a recursive definition as follows: a fully-parenthesized expression (FPE) is

- either a lone number by itself, or
- a sequence of open paren followed by an FPE, then an operator (i.e., one of $+,-,*,/,**$), then an FPE, then a close paren.

Formally, the BNF grammar of FPEs is

$$\langle op\rangle ::= \text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'/'} \mid \text{'**'}$$
$$\langle FPE\rangle ::= \langle \text{a number}\rangle \mid \text{'('}\ \langle FPE\rangle\langle op\rangle\langle FPE\rangle\ \text{')'}$$

For example:

- `augmentExpr(['42']) == ['42']`
- `augmentExpr(['42', '+', '1'])==['(','42','+','1',')']`. That is, "42 + 1" becomes "(42 + 1)"
- `augmentExpr(['42', '+', '2', '*', '5']) == ['(','42','+','(','2','*','5',')',')']`. That is, "42 + 2 ∗ 5" becomes "(42 + (2 ∗ 5))".
- `augmentExpr(['4', '-', '1', '+', '2']) == ['(','(','4','-','1',')','+','2',')']` That is, "4 − 1 + 2" becomes "((4 − 1) + 2)".
- `augmentExpr(['1','+','3','/','(','5','-','2',')','*','1.5'])` should return
  `['(','1','+','(','(','3','/','(','5','-','2',')',')','*','1.5',')',')']`
  that is "`(1+((3/(5-2))*1.5))`".

Performance Expectations: Assuming each token is a string of of length bounded by a constant C, your `augmentExpr` function must run in time $O(n^2)$ or faster where n is the number of tokens in the input list (i.e., $n=$ `len(tokens)`). For performance analysis, we'll assume that the input list `List<String>` will given as `ArrayList<String>`. You can actually obtain an $O(n)$-time algorithm.

Hints: This task can be solved by extending the Dijkstra's two-stack algorithm studied in class. You may wish to attempt this task in two stages. Again, you will scan the token list from left to right:

- *Stage I: No Partial Parentheses.* Keep two stacks like before, but what should happen when you see an operator? When should you push that to one of the stacks? And when should you not—but rather popping certain things out?
- *Stage II: Partial Parentheses.* What should be done upon seeing an open parenthesis? What about a close parenthesis? If popping, how far?

Subtask III: Finally, using what you implemented in the previous two subtasks, you'll write a function

```
public static double evalExpr(List<String> tokens)
```

that takes a list of tokens (possibly partially parenthesized) and returns a floating number equal to the result of evaluating the given expression.

Exercise 4: Double Trouble: Rank and Select (4 points)

For this task, save your code in `DoubleTrouble.java`

The *rank* of an element e with respect to a list L, denoted by $r(L,e)$, is the number of elements in L that are less than e. For example, if L is the list $[10,21,32,53,54]$, then we know

- $r(L,9)$ is 0
- $r(L,10)$ is 0
- $r(L,33)$ is 3 because 10, 21, and 32 are *all* smaller than 33.

The ultimate goal of this task is to efficiently locate the k-th rank element in the combined list given two sorted lists A and B. Throughout, let $n = \texttt{len}(A)$ and $m = \texttt{len}(B)$. Let $A+B$ the list obtained by concatenating A with B. We guarantee that $A+B$ *contains no repeated numbers.* We aim to obtain $O(\log^2(n+m))$ or faster running time. You will proceed by completing the following subtasks:

Subtask I: Your first subtask involves writing a function

```
public static int rank(int[] A, int[] B, int e)
```

that takes two *sorted* arrays A and B and an integer e, and computes the rank $r(A+B,e)$ *without* actually expanding out $A+B$. Your code must run in at most $O(\log(n+m))$ time or faster.

Subtask II: Write a function

```
public static Integer select(int[] A, int[] B, int k)
```

that locates and returns the rank k element from $A+B$. To be precise, your code must return the same answer as Python's `sorted`$(A+B)[k]$ (only faster!). If the rank k element is not present, you will return `null`. Your code must run in at most $O(\log^2(n+m))$ time or faster.

NOTE: Keep in mind that when $n,m>0$, $\log n \le \log(n+m)$ and $\log m \le \log(n+m)$. Hence, we know that $\log n + \log m$ is still $O(\log(n+m))$.

BONUS: (Do this for brownie points!) Redo Subtask II so that the running time is $O(\log(n+m))$ or faster.

Exercise 5: HackerRank Problems (6 points)

For this task, save your code in `hackerrank.txt`

There are *three* problems in this set. You **must** write your solutions in Java (1.8). You will hand them in electronically on the HackerRank website. If you're planning on using a late token, please finish these HackerRank problems before the due date. The HackerRank "contest" ends at 11:59pm on the day the assignment is due.

Important: You will write down your Hacker ID username in a file called `hackerrank.txt`, which you will submit as part of the assignment. This will be used to match you with your submission on HackerRank.

You can find your problems at

https://www.hackerrank.com/muic-data-structures-t-318-assignment-5