CSC 225 Assignment 4

Heaps, Trees and Word Frequency

University of Victoria

June 22, 2022

Due date

The submission deadline is 11:59pm on Wednesday July 13, 2022.

How to hand it in: Submit your ASSIGNMENT4.PDF and WORDFREQUENCYHEAP.JAVA, WORDFREQUENCYBST.JAVA and WORDFREQUENCYREPORT.JAVA files through the Assignment 4 link on the CSC225 BrightSpace page.

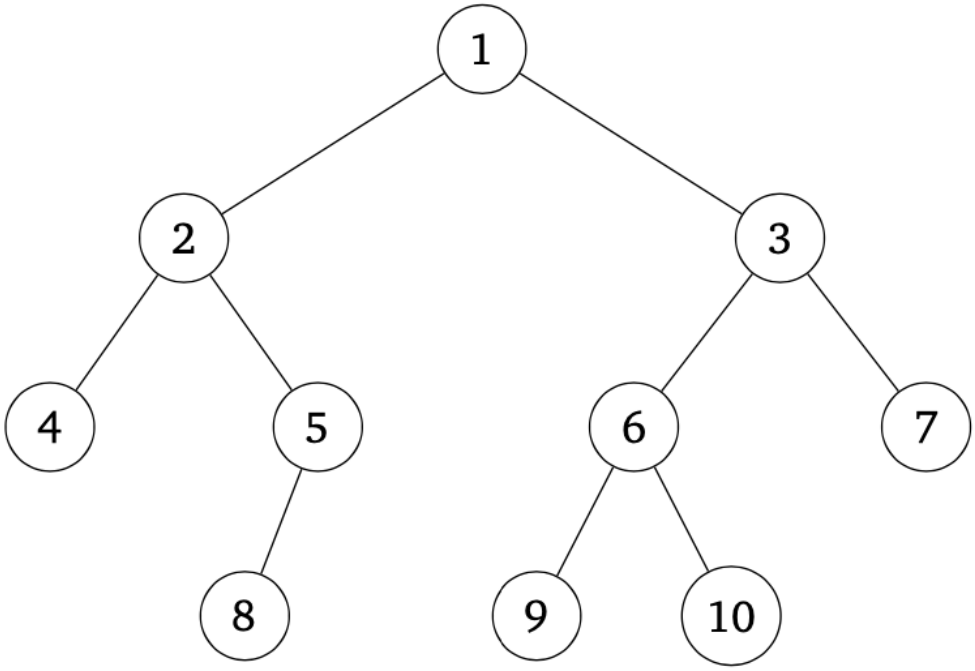
IMPORTANT: The files submitted must have a .PDF or a .JAVA extension.

Exercises

Question 1

Using the binary tree below for the entirety of question 1, for each of the following tree traversal methods, record the order in which your nodes are traversed. (**Note:** please use the following format: [#,#,#,#,#,#,#,#,#,#] in your submission. This will help the TA team in grading.)

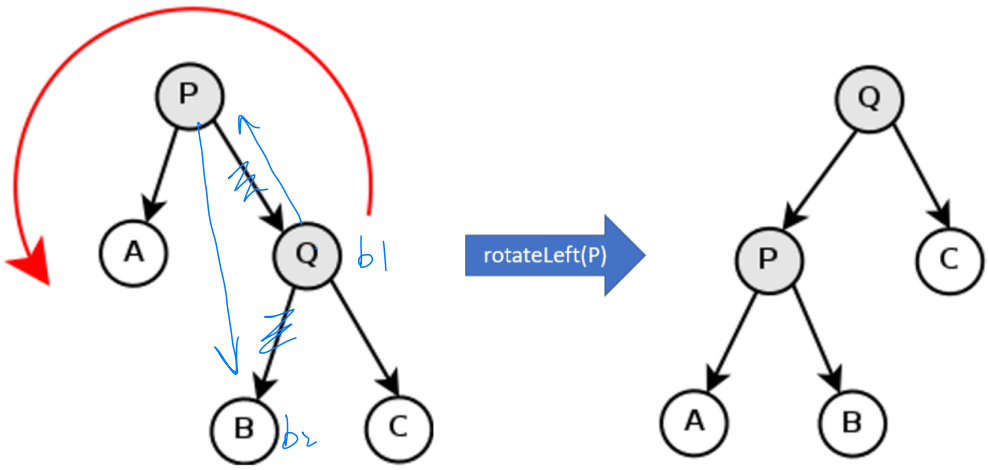
1. **(10 points)** Post-order traversal [4,8,5,2,9,10,7,6,3,1]
2. **(10 points)** Pre-order traversal [1,2,4,5,8,3,6,9,10,7]
3. **(10 points)** In-order traversal [4,2,8,5,1,9,6,10,3,7]



Question 2

You learned in lecture about simple rotations in binary trees. In last week's lab, you implemented a binary tree in Java. For this question, you will write pseudocode to implement a basic rotate left method. Your method will take a node as input. All nodes in this example have access to their parent, leftChild, and rightChild fields. You can assume the input node has a right child. Your method should rotate left around the given node as in the image provided below. Pseudocode should be a maximum of 15 lines long. Do not assume the input node has no parent.

```
rotateLeft(Node n) {
//Your
//Pseudocode
//Here
}
```




```
rotateleft(Node n){
    Node buffer_1= n.rightchild
    Node buffer_2 = buffer_1.leftchild
    buffer_1.parent = n.parent
    buffer_1.leftchild = n
    n.parent = buffer_1
    n.rightchild = buffer_2
    buffer_2.parent = n
}
```

Question 3

For this coding exercise, you will be counting words in text files and keeping track of the frequency (number of times) each word is found in a text file.

To do this, you will be populating a heap and a binary search tree with entries. An entry consists of a **word** (string) and the number times that word is found in the input file, the **frequency** (int).

In part 1, you will be implementing a max-heap, prioritized by word frequency.

In part 2, you will be implementing a binary search tree, sorted alphabetically.

In part 3, you will use your implementations from Part 1 and 2 to explore the word frequencies of different files.

In the assignment .zip file, the **A4TESTER.JAVA** file has tests for each part in separate methods. We've provided some sample output based on the solution for Part 3. Further instructions on how to use your implementation to analyze any text file are provided in the Part 3 section.

Part 1: Max-Heap Implementation

For Part 1, you will need to implement the methods specified in the **PRIORITYQUEUE.JAVA** interface.

You will be implementing max-heap of `Entry` objects. Read the **ENTRY.JAVA** file to get an idea of what an entry is, and the `testHeapOperations` method in **A4TESTER.JAVA**.

In order to complete the implementation, we suggest writing some helper methods. Remember the Heap Properties must be maintained; refer to the lecture videos, slides, and in-class worksheet as necessary.

For Part 1, the only file you need to add implementation to is **MAXFREQUENCYHEAP.JAVA**. We suggest adding some further tests in the `testHeapOperations` method in **A4TESTER.JAVA** until you are satisfied your implementation is correct.

Part 2: Binary Search Tree Implementation

For Part 2, you will need to complete the implementation of the `handleWord` and `getFrequency` methods specified in **WORDFREQUENCYBST.JAVA**. Notice that these two methods have a lot in common wiht some common BST operations. For

`handleWord`, the idea is that whenever a word is seen for the first time when reading the contents of a dataset, it should be added to the BST (in the correct location). At this point, the frequency associated with that word should be one (as it has only been seen once). Any subsequent time that word is seen in the dataset, its frequency value should be incremented. The `testBSTOperations` in **A4TESTER.JAVA** might help you get a better idea of what is expected from this method.

The `getFrequency` method just searches through the BST for a given word and returns its associated frequency. This would be useful to see how many times a word was found in a dataset.

For part 2, the only file you need to add implementation to is **WORDFREQUENCYBST.JAVA**. We suggest adding some further tests in the `testBSTOperations` method in **A4TESTER.JAVA** until you are satisfied your implementation is correct.

Part 3: Word Frequency Report

For Part 3, you will use your completed BST and Heap implementations to perform a word frequency analysis on a text file. A Scanner has been set up for you to read through each word in the file. As each word is scanned, it will be "handled" in the BST (by either inserting or updating an entry). After reading through the input file, the BST should contain entries for each word that includes its number of occurrences in the input file.

The Heap is then populated with the tree entries and used to determine which words were found the highest number of times in the input file.

We have provided three sample input files to test with (SAMPLE1.TXT, SAMPLE2.TXT and SAMPLE3.TXT). For each input file, you will determine the top 5 most common words overall, the top 5 most common words with at most *n* letters, and the top 5 most common words that begin with a given letter.

Specify the input file to examine through the command line when executing the program. For example, to examine SAMPLE2.TXT, we would type: `java A4Tester.java sample2.txt`.

Here is our sample output for SAMPLE3.TXT:

```
Overall most frequent:
{"the", 40575}
{"of", 19656}
{"and", 14789}
{"a", 14400}
{"to", 13777}

Most frequent words with 6 or more characters:
{"marius", 1358}
{"valjean", 1112}
{"himself", 1085}
{"cosette", 1013}
{"little", 973}

Most frequent words starting with a c:
{"cosette", 1013}
{"could", 676}
{"come", 556}
{"child", 470}
{"can", 433}
```

For Part 3, the only file you need to add implementation to is **WORDFREQUENCYREPORT.JAVA**. We suggest adding some further tests in the `testFrequencyReport` method in **A4TESTER.JAVA** until you are satisfied your implementation is correct. In particular, you should ensure your solution works for other minimum word lengths and also on words that begin with a letter other than "c".

If fewer than five words meet the given criteria, the locations in the `top5` array should be left as `null`.

Good luck!