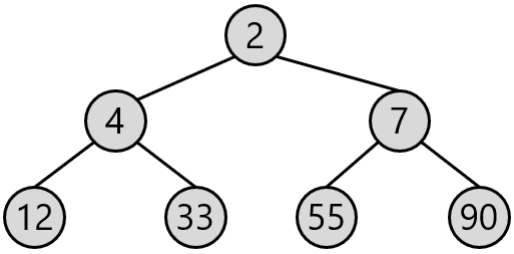
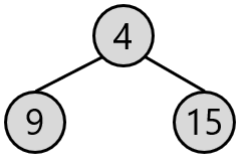
# COSC 222: Data Structures
## Lab 6 – AVL Tree, Priority Queue & Heaps

**Question 1 [6 marks]:** This question does not require coding. You will need to submit a PDF file with your answers. This can be handwritten and scanned, drawn on a tablet, or created using a diagramming program and word processor. Name your file **Lab6Question1.pdf**.

a. **[3 marks]** Draw the tree state of the min-heaps after adding the following elements into the tree. Once an element is added to the tree, use the updated tree to insert the next element. Add to Tree 1: 10, 1, 35, 100, 5. Add to Tree 2: 5, 2, 7. Show the trees after each insertion.

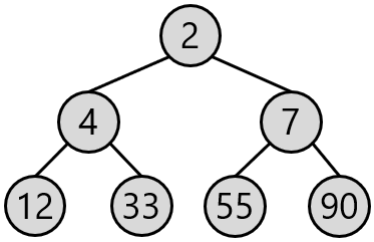



Tree 1

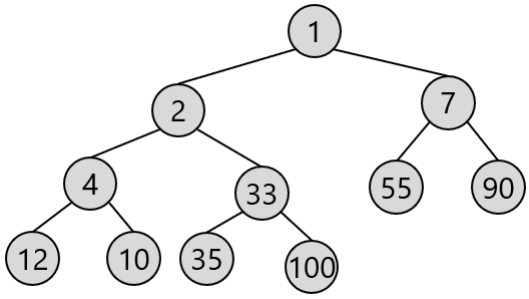



Tree 2

b. **[3 marks]** Suppose we have two min-heaps as shown below. Show the state of the min-heaps after calling the `remove()` method (discussed in the class) 3 times. Show the tree after removing each element.




Tree 3




Tree 4

**Question 2 (Balanced Trees): Is It Balanced? [4 marks]**

Write a program which will evaluate a tree and determine whether or not it is an AVL balanced tree. You will need the given **BSTNode.java**, **BST.java**, and **BalanceTest.java** files provided. The **BSTNode** class is similar to the Node or TreeNode classes seen in lectures and earlier labs. The **BST** class models a simple binary search tree, which is NOT self-balancing. The **BalanceTest** class creates several simple trees and tests them to determine if they are already balanced (like an AVL tree). *Do not change anything in either of the **BSTNode** or **BalanceTest** classes.*

Your task is to complete two methods in the **BST.java** file: `recIsBalanced()` and `height()`. These two methods act as helper methods for the method `isBalanced()`, which returns a boolean: true if the given BinaryTree is balanced, false if it is not.

Use the following *recursive* definition of a balanced tree. A tree is balanced if and only if: (a) the difference between the heights of the root's left and right subtrees is between 1 and -1, and (b) the left and right subtrees are also balanced.

If you complete these methods properly, you will see the following output when running the **BalanceTest.java** file.

**Sample Output:**

```
Tree 1: (((1(2))3)4((5)6))
Is it balanced?  No

Tree 2: (((1)2(3))4((5)6))
Is it balanced?  Yes

Tree 3: (1(2(3(4(5(6))))))
Is it balanced?  No

Tree 4: ((1(2))3((4(5))6))
Is it balanced?  No

Tree 5: (((1)2)3((4)5(6)))
Is it balanced?  Yes
```

**Question 3 (Priority Queue): Emergency Triage [10 + 2 marks]**

In this question, you will implement a priority queue to represent the triage in a hospital emergency department. You will need the following files: **PriorityQueue.java**, **PQNode.java**, and **PriorityQueueTest.java**.

First review the file **PQNode.java**. This class represents a patient as a node. Your priority queue will resemble a doubly-linked list, so each node has a reference to its previous and next nodes. *Do not edit anything in this file.*

Next review the **PriorityQueueTest.java** file. This class will be used to test your priority queue in the context of an emergency triage. You may wish to make changes to this file to better test your methods, but you will not submit this file, so don't change it just to make your **PriorityQueue** work.

Finally check the **PriorityQueue.java** file. Note that we only have reference to the first node in the queue, rather than having both first and last nodes. You may add a reference to the last node; however, for the incomplete methods below, you don't need it.

This class is incomplete. Your task is to complete the following methods:

***Part A [4 marks]:***

`add()`: There are two add methods in this class. The first takes a patient's name, problem, and priority level, and creates a new node. It then calls the second add method, which is the one you need to complete. This second method takes a `PQNode` and adds it to the queue in its proper position, according to these guidelines:

In Canadian hospitals, a 5-level triage is used: 1 is most urgent (patient requires resuscitation), and 5 is least urgent (does not need to be treated immediately). If a patient arrives whose situation is more urgent than someone else in the queue, the new arrival "cuts in line" before the previous patient. Note that if a new patient comes in at the same level as a patient already in the queue, the new patient is placed after the previous patient.

[*Hitns: check lecture 3 slides 51-61. Don't forget to update the code for doubly linked list*]

***Part B [4 marks]:***

`remove()`: This method completes a typical remove action for a priority queue. Note that it does not take any parameters, but it should return the node which has been removed. For convenience in other methods (i.e., specifically for `changePriority()`), you should do most of the work in the `remove(PQNode node)` method and then call `remove(PQNode node)` from `remove()`.

***Part C [1 mark]:***

`isEmpty()`: This method returns a boolean: *true* if the queue is empty, and *false* if there are any elements in it.

***Part D [1 mark]:***

`size()`: This method returns an int, representing the number of elements in the queue.

***Part E [2 marks (BONUS)]:***

`changePriority()`: Sometimes, in an emergency waiting room, the priority level of a patient needs to be changed. For example, suppose a patient who originally came in with a sore throat suddenly can't breathe and becomes unconscious. While they may have originally been a Level 5 priority, they immediately become a Level 1 priority. Write a method that handles this situation. Take in the name of the patient (String) and the new priority level (int) as arguments. Then find the corresponding patient node, change that patient's priority, and move the patient to their new spot in the queue, as if they were coming in as a new patient.

If you complete all of these methods correctly, you should find the following output when running **PriorityQueueTest.java**.

```
add Alice
add Bob
add Cameron
add Dorothy
```

```
add Eric
add Fred
----------
-  #1: [Fred - heart attack, resuscitation (1)]
-  #2: [Cameron - head injury (2)]
-  #3: [Eric - deep cut, serious blood loss (2)]
-  #4: [Alice - broken leg (3)]
-  #5: [Dorothy - broken arm (3)]
-  #6: [Bob - sore throat (5)]
----------
remove
remove Fred
----------
-  #1: [Cameron - head injury (2)]
-  #2: [Eric - deep cut, serious blood loss (2)]
-  #3: [Alice - broken leg (3)]
-  #4: [Dorothy - broken arm (3)]
-  #5: [Bob - sore throat (5)]
----------
*BONUS*
Bob is suddenly unable to breathe and becomes unconscious.
change priority of Bob to 1
remove Bob
add Bob
----------
-  #1: [Bob - sore throat (1)]
-  #2: [Cameron - head injury (2)]
-  #3: [Eric - deep cut, serious blood loss (2)]
-  #4: [Alice - broken leg (3)]
-  #5: [Dorothy - broken arm (3)]
```

**Submission Instructions:**

- Create a folder called "Lab6_<*student_ID* >" where <*student_ID* > is your student number/ID (e.g. **Lab6_12345678**). Only include the mentioned pdf and java files in the folder. **DO NOT** include any other files (e.g., .class, .java~ or any other files)
    - For question 1, include your **Lab6Question1.pdf** file.
    - For Question 2, include your **BST.java** file.
    - For Question 3, include your **PriorityQueue.java** file.
- Make a **zip file** of the folder and upload it to Canvas.
- To be eligible to earn full marks, your Java programs **must compile and run** upon download, without requiring any modifications.
- These assignments are your chance to learn the material for the exams. **Code your assignments independently.**