# Lab 6: Binary Search Trees & Priority Queues

F28SG – Introduction to Data Structures and Algorithms (6 marks)

## Plagiarism policy

This lab is not group work, and you are assessed individually.

Therefore, the work you submit for this lab **must be entirely your own**. You must not share your code solution with other students, and you must not copy code solutions from others. The university plagiarism policy is clear:

> *"Plagiarism involves the act of taking the ideas, writings or inventions of another person and using these as if they were one's own, whether intentionally or not."*
> ***Definition 2.1.***

The disciplinary action for plagiarism is an award of an F grade (fail) for the course. Serious instances of plagiarism will result in Compulsory Withdrawal from the university.

## Project Structure

The project is organized as follows:

- The `src` directory contains the following source files:
  - **BinarySearchTree.java** implements a binary tree
  - **DLinkedList.java** implements a a doubly linked list, to be used by your `inOrderTraversal` and `inOrderTraversalPrint` method implementations in **BinarySearchTree.java** (Q1, Q2)
  - **PriorityQueue.java** implements a priority queue for integers
  - **Sort.java** is where you should sort a priority queue (Q4)
- The `test` directory contains the unit tests for the project, including:
  - **PriorityQueueTest.java** is where you should write tests for priority queue implementation in **PriorityQueue.java** (Q3)

## Q1) Implement InOrder Tree Traversal (2 Points)

In the binary search tree lectures, we did an exercise where you had to give the pre-order, post-order and in-order traversals of Trees.

In this Q1 task you should implement a method that performs **in-order traversal** by printing out the numbers in a tree, in the correct order from `inOrderTraversalPrint()`.

You should do this by:

1. Calling `inOrderTraversalPrint()` in the outer `BinarySearchTree` class.
2. That method should call the `inOrderTraversalPrint()` method on the root node.

3. The `inOrderTraversalPrint()` method inside the `BSTNode` class should be recursive, I.e. each node should call the `inOrderTraversalPrint()` method on its children in a certain order, printing node values as it goes.

A method has been provided in the `BinarySearchTree` class in **BinarySearchTree.java**

```
public void inOrderTraversalPrint(){
    if (rootNode != null)
    rootNode.inOrderTraversalPrint();
}
```

In the nested **BSTNode** class, an empty method has been provided for you:

```
public void inOrderTraversalPrint(){

}
```

Your task is to complete this method by implementing a recursive **in-order traversal** that prints the value of each node as the algorithm visits each node. A main method has been provided and when running it, the following should be printed.

```
******* Tree 1 : empty   ***********

******* Tree 2 : 1 node  ***********

 1

******* Tree 3 : 4 nodes ***********

 1

 2

 3

 4

******* Tree 4 : 7 nodes ***********

 1

 2

 3

 5

 6

 7

 8
```

See ***Hint1!*** and ***Hint2!*** *if* you are stuck.

## Q2) A Better Traversal (1 point)

The method from Part 1 is not particularly useful beyond debugging, and it would be much more useful to be able to return the list of elements obtained by traversing the Binary Search Tree, which would allow you to use that list elsewhere in a program.

For this task, a doubly linked list implementation has been provided (**DLinkedList.java**).

In **BinarySearchTree.java** there is an incomplete method:

```
public DLinkedList<T> inOrderTraversal(){
            DLinkedList<Integer> dll = new DLinkedList<>();
            /* your code goes here */
}
```

There is also the incomplete `inOrderTraversal` method inside the `BSTNode` class:

```
public void inOrderTraversal(DLinkedList dll){
      /* your code goes here */
}
```

Your task is to complete these two methods such that a doubly linked list is returned with elements that have been added to it during an in-order traversal. Your `inOrderTraversal` method inside the `BSTNode` class should be recursive. Each recursive call from a node to its children should pass around the linked list, when making the recursive method call on those children. Use the `addAtTail` linked list method to add elements from the tree in the correct order for in-order traversal.

Methods in **BinarySearchTreeTest.java** provide JUnit tests to check your implementation for Q2. Run these tests on your code, they should pass if your code is correct. See ***Hint3!*** if any tests fail.

## Q3) JUnit Tests for Priority Queues (1 point)

Complete the JUnit test for the following Priority Queue operations in **PriorityQueueTest.java**:

- **insert** operation - complete two tests:
  1. `insertTestMin` which should add several integer values into the priority queue in a random order, then your test should ensure that the `min` method returns the smallest value from the priority queue.
  2. `insertTestSize` which should add several values into the priority queue in a random order, then your test should ensure that the `size` method returns a count of how many values there are in the priority queue.
- **removeMin** operation - complete one test:
  1. `removeMinTest` which adds several integer values into the priority queue in a random order, then your test should ensure that values come out from the priority queue in the correct order with the `removeMin` method.

These three tests should pass if correctly implemented from the specification above, because the priority queue implementation in **PriorityQueue.java** is fully implemented.

## Q4) Sorting with a Priority Queue (1 point)

In **Sort.java** there is a method:

```
public static void sort(int[] arr){
}
```

Your task is to complete this method. The method should sort the given array ***using a priority queue***. A Priority Queue implementation has been provided in **PriorityQueue.java**. The sorting should be performed "in place", I.e. update the values in the `arr` variable, rather than creating a new array.

The **Sort.java** contains a `main` method so that you can run your program. When the `main` method in **Sort.java** is executed, the numbers should be printed in order. Right click the **Sort.java** file, then "Run As", then "Java application", and you should see printed:

```
2

3

4

5

53

67
```

**SortTest.java** has JUnit tests that should pass if your implementation of the `sort` method is correct.

All tests in your lab 5 project should now pass.

## Q5) Code Quality (1 point)

- Code quality is vitally important for so many reasons. Not least, for readability and maintainability, not just for yourself but for others too since in industry, software engineering is almost always a group exercise. Real world software engineering is mostly about refactoring and testing code, rather than writing new code (more code means higher maintenance costs!).
- An additional mark is awarded if your code is deemed to be of high quality:
- **Code simplicity**
  - Is the code as short as it can be?
  - Is the Big-Oh complexity as small as it can possibly be?
  - Is the control flow (loops, while statements, if statements, etc.) simple to follow?
- **Documentation**
  - Are you using comments ***inside your implementation methods and test methods*** to provide an algorithmic commentary about what the code is doing.

  Here is a **good** comment:

  ```
  // creates unidirectional connection from the predecessor node to the new node
  prevNode.nextNode = newNode;
  ```

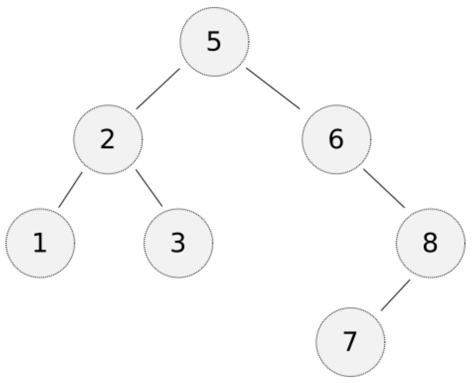
  Here is a **less useful** comment:

  ```
  // sets preNode.nextNode to newNode
  prevNode.nextNode = newNode;
  ```

    - Is Javadoc syntax used for documenting the code? ***Hint!*** In Eclipse move your cursor to the text definition of a field, method or a class, then use the following keyboard shortcut: ***Alt + Shift + j*** , and then fill in the generated Javadoc template. If you are using MacOS, the shortcut is: ***⌘ + Alt + j***
- **Conventions**
    - Are sensible Java code conventions used, e.g. for code indentation, declarations, statements, etc? See Sections 4, 5, 6, 7 and 9 of *Java Code Conventions*. ***Hint!*** *Use the keyboard shortcut in Eclipse:* ***Control + Shift + f*** , or on MacOS: ***⌘ + Shift + f***

***Hint1!*** When recursively traversing a tree, think of it like moving the recursive method around the tree. For example, to move a recursive method from a parent node down to its child nodes, the parent node should call the method on its child nodes.

***Hint2!*** Tree 4 (with 7 nodes) in the `main` method is constructed with a sequence of `insert` calls which adds integers to an empty tree in the following order: 5, 2, 1, 6, 3, 8, 7. The resulting tree looks like this:



Preorder: 5, 2, 1, 3, 6, 8, 7,
Inorder: 1, 2, 3, 5, 6, 7, 8,
Post order: 1, 3, 2, 7, 8, 6, 5

Step through in sequence the code in your recursive `inOrderTraversalPrint` implementation. Your recursive code should print the values in the following order: 1, 2, 3, 5, 6, 7, 8.

***Hint3!*** Find out which in the JUnit tab in Eclipse which lists which tests passed and which failed. Then go to the Java source code for the failing test and draw out the binary tree constructed with the sequence of `bst.insert(..)` calls. Once you have this diagram of the binary tree drawn out, step through in sequence the code in your recursive `inOrderTraversal` implementation. Does your recursive code append values to the list in the way the failing JUnit states it should?