# Homework # 13
# due Monday, Decemeber 9, 10:00 PM

In this assignment you will implement sorting on a binary tree using insertion sort and mergesort pages 630–640 (3rd ed: pp. 614–624). You will use "comparator" objects.

## 1 Concerning Merge Sort

The mergesort algorithm has the following structure:

**mergesort(s)** where "s" is a list of elements

1. If "s" is empty or has only one element, return it unchanged.
2. Otherwise:
   (a) split "s" into two parts, "$s_1$" and "$s_2$".
   (b) mergesort($s_1$) recursively
   (c) mergesort($s_2$) recursively
   (d) merge the resulting sorted elements into one
   (e) return the result.

## 2 Concerning the BasicTree ADT

This tree is somewhat different from Binary Search Trees we have seen before. It does not support removal or iteration, or allow null elements. It does allow duplicate items, as well as different items that are equivalent according to a comparator. When an equivalent element is added, it is always place in the right subtree of any equivalent element already in the tree.

The ADT supports two sorting methods, `insertionSort` and `mergeSort`. These methods do not change the data structure. Rather, they accept a `Comparator` as an argument, and return an array of the contents in sorted order. Furthermore, `mergeSort` must be stable, meaning that two elements that are sortedness equivalent according to the comparator should be relatively sorted according to the inorder traversal on the tree.

## 3 Implementing the Algorithms

The general strategy for both insertion sort and merge sort is as follows: First, allocate an array large enough to hold all the elements in the tree. (Don't forget that you have to cast an `Object` array to serve as a generic array) Then, recursively sort the elements of the tree into the array. Elements should be added to the array in the order of preorder traversal.

In insertion sort, you should insert the element into the array as soon as you encounter it during recursion. This means the (incomplete) contents of the array should be sorted at the beginning of every recursive call. So you will insert elements one-by-one into the array, but you will do so in the order of a preorder traversal on the tree.

In merge sort, when you encounter an element on a recursive call, you will immediately place it in the array at a specific index. You must leave space in the array for all the elements in the left subtree, and all the elements in the right subtree. For example, if your tree has ten nodes, and the root's left subtree has three nodes, the root element should be placed at index three. This leaves space to the left for the three elements of the left subtree, and space to the right for the right subtree.

This open space will be filled during the recursive sorting of the left and right subtrees. By the time the left recursion finishes, that range will be not only filled, but also sorted. After both left and right recursions are finished, a merge operation needs to be carried out. This operation needs to merge the sorted left subarray, the sorted right subarray, and the single middle element corresponding to the root of the subtree. This requires a three-way merge, which must choose the smallest of three elements on each iteration. We suggest using a `Queue` to hold the elements as you select them from the array, and then overwriting the relevant range of the array with the contents of the queue after filling it. This will result in the sorted combination of the left subarray, right subarray, and middle node.

Because merge sort requires stability, you must choose the element that is currently earlier when the comparator does not discriminate.

Note that this version of merge sort may not always divide the elements in half on recursion. Our performance guarantees depend on the balance of the tree.

You MUST follow this design in order to receive full credit.

## 4 What you need to do

Implement methods in the following bottom-up order, so that you can pass more tests as you go:

**insert**

**doInsertionSort**

**insertionSort**

**smallestIndex**

**merge**

**doMergeSort**

**mergeSort**

## 5 Files

The repository for this homework includes the following files:

**src/TestBasicTree.java** JUnit test cases

**src/TestEfficiency.java** JUnit test cases for efficiency. Insertion sort should be slow!

**src/TestInternals.java** JUnit test cases for private helper methods

**src/edu/uwm/cs351/BasicTree.java** skeleton file.