CS 0445 Data Structures
Exam 2 Practice Question Solutions

Fill in the Blanks

1) When a **backtracking algorithm** backtracks, an activation record is ___**popped off of** _____ [pushed onto, popped off of] the run-time stack.
2) A Stack organizes data by _____ **Last In First Out**_____ and a Queue organizes data by _____ **First In First Out** _____.
3) Implementing a Priority Queue using a Heap allows the add() and remove() methods to be executed in ____**O(lgN)**____ time .
4) We discussed our **divide and conquer sorting algorithms** by answering **two questions**:
   a) How do we ___ **divide the problem into subproblems?**_____
   b) How do we ___ **use the subproblem solutions to determine the overall solution?**____
5) The **get(i)** method for the Java ArrayList runs in time ____**O(1)**_____ and for the Java LinkedList runs in time ___**O(N)**_____.
6) A tree node that has a parent and at least one child is called a(n) _____**interior**_____ node.

True/False (explain why false answers are false)

1) The **worst case scenario** for Insertionsort is **the same** whether the items are in an **array** or in a **linked list**. **FALSE – for array it is reverse sorted data, for LL it is sorted data.**
2) The **Simple Quicksort** algorithm has both a **worst case** runtime and an **average case** runtime of **$O(N^2)$.** **FALSE – the average case is O(NlgN)**
3) An **Iterator** for a class in Java is limited in that it is dependent on the public methods of the class and thus cannot be tailored to the implementation details of the class. **FALSE – Iterators are implemented so that they have access to the internal details of the classes.**
4) Deleting the **root of a MaxHeap** involves simply removing the root node and moving the greater of its children nodes up the root position**.** **FALSE – it is more complicated than this – see notes.**
5) A **full** binary tree with a height of h will have $2^h - 1$ nodes. **TRUE**

Short Answers

1) Consider a Binary Search Tree with N nodes and the add() method (i.e. an Insert into the tree). Discuss in detail how long this method will take in terms of N in both the average and worst cases. Thoroughly explain your answers.
ANSWER: The run-time for the add() method of the BST is dependent upon the height of the tree. This is because the algorithm starts at the root and proceeds to a null reference (after a leaf) before inserting the new node. In the average case, a BST with N nodes will be relatively balanced, having a height of $O(\log_2 N)$. However, in the worst case, the height can be linear (O(N)), thus causing a very poor run-time for add().

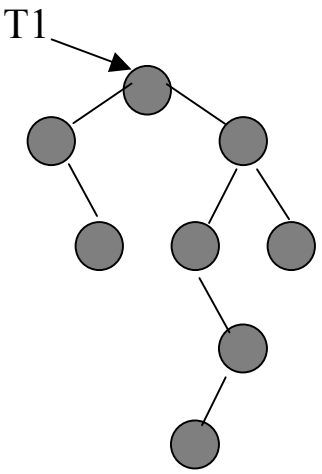
2) Consider the MergeSort and QuickSort sorting algorithms. Give the Big-O run-times for both algorithms (average and worst cases). In real terms, which algorithm has the faster run-time and why? Be specific.
ANSWER:
MergeSort average and worst case: $O(N\log_2 N)$
QuickSort average case: $O(N\log_2 N)$; QuickSort worst case: $O(N^2)$
Despite its possible poor worst case performance, QuickSort tends in general to be faster than MergeSort due to extra overhead incurred in the MergeSort algorithm. This overhead is due to the merge algorithm, which requires the data to be merged into a new, temporary array before being copied back into the original array. This extra copying requires extra time. QuickSort, on the other hand, sorts the data in place, and does not require an extra array.

3) Consider implementations of the **Queue ADT**:
   a) One option we discussed was to use an **ArrayList** and simply do **add() for enqueue** and **remove(0) for dequeue**. Is this a good implementation? Explain why or why not **in detail**.
   Answer: No, it is not, because remove(0) on an ArrayList requires the shifting of the remaining items to the right to fill the gap. This causes remove(0) to take O(N) time when it should only take O(1) time.

   b) If we use the same operations listed in a) above, but with a **LinkedList** rather than an ArrayList as the data, will this be a good implementation? Explain why or why not **in detail**.
   Answer: It depends on how the LinkedList is implemented. If the LinkedList has a Rear or Tail reference (or is circular or doubly-linked) both add() and remove(0) can be easily done in O(1) time and the implementation is good. However, if only a Front reference is available, the add() requires a traversal of the list, making it an O(N) operation.

Traces

1) Give **ALL OUTPUT** to the program below, in the **CORRECT ORDER** that it is generated. Clearly distinguish your output from scratch work by DRAWING A BOX AROUND IT. **Note:** The tree generated in the program is drawn below to help you.



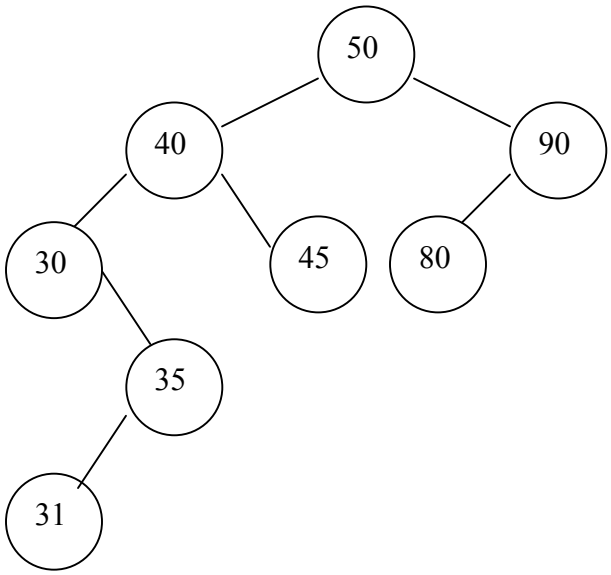

```
import java.util.*;
import TreePackage.*;
public class TreeTrace
{
    public static void main(String [] args)
    {
        BinaryNode<Object> T1, curr1;
        T1 = new BinaryNode<Object>(null);
        T1.setLeftChild(new BinaryNode<Object>(null));
        T1.setRightChild(new BinaryNode<Object>(null));
        curr1 = (BinaryNode<Object>) T1.getLeftChild();
        curr1.setRightChild(new BinaryNode<Object>(null));
        curr1 = (BinaryNode<Object>) T1.getRightChild();
        curr1.setRightChild(new BinaryNode<Object>(null));
        curr1.setLeftChild(new BinaryNode<Object>(null));
        curr1 = (BinaryNode<Object>) curr1.getLeftChild();
        curr1.setRightChild(new BinaryNode<Object>(null));
        curr1 = (BinaryNode<Object>) curr1.getRightChild();
        curr1.setLeftChild(new BinaryNode<Object>(null));
        System.out.println("Height of T1 is " + height(T1, 0));
    }

    public static int height(BinaryNode<Object> T, int level)
    {
        if (T == null)
            return 0;
        else
        {
            int lheight = height(T.getLeftChild(), level + 1);
```

```
            int rheight = height(T.getRightChild(), level + 1);
            int theHeight = 0;
            if (lheight > rheight)
                  theHeight = 1 + lheight;
            else
                  theHeight = 1 + rheight;
            System.out.println("At level " + level + " height is " + theHeight);
            return theHeight;
        }
    }
}
```

```
C:>java TreeTrace
At level 2 height is 1
At level 1 height is 2
At level 4 height is 1
At level 3 height is 2
At level 2 height is 3
At level 2 height is 1
At level 1 height is 4
At level 0 height is 5
Height of T1 is 5
```

2) Consider an EMPTY Binary Search Tree, T, of Integers. Show the resulting tree after the sequence of add() operations below. For simplicity, just draw each node as a circle with the data for that node inside the circle. Note: The actual adds should be of "new Integer()" objects. For brevity, I have only included the actual int values here.

```
T.add(50);  T.add(40);  T.add(30);  T.add(90);  T.add(35);
T.add(45);  T.add(31);  T.add(80);
```



Coding

1) Consider the BinaryNode class as we discussed in lecture and, in particular, a tree of BinaryNode<Integer> nodes. Write a recursive method that will traverse a tree of BinaryNode<Integer> nodes and return the number of values equal to a given key value. For example, a program using this might look like:

```
BinaryNode<Integer> root;
// code to create tree omitted
Integer key;
// code to set key to a specific value omitted
int num = countMatches(root, key)
System.out.println("The key " + key + " was found " + num + " times ");
```

Use the header below:

```
public static int countMatches(BinaryNode<Integer> tree, Integer
key)
{
   int matches = 0;
   if (tree != null)
   {
         if (tree.getData().equals(key))
               matches++;
         matches += countMatches(tree.getLeftChild(), key);
         matches += countMatches(tree.getRightChild(), key);
   }
   return matches;
}
```