CS 0445 Data Structures
Exam 2 Practice Questions

Below are some examples of the types of questions that may appear on the Exam 2. I recommend trying all of the questions before looking at the answer key. Please remember that these are just examples, and to adequately prepare for the exam you must study everything indicated in the review sheet.

Fill in the Blanks

1) When a **backtracking algorithm** backtracks, an activation record is _________________________ [pushed onto, popped off of] the run-time stack.
2) A Stack organizes data by _____________________________________ and a Queue organizes data by ___________________________________.
3) Implementing a Priority Queue using a Heap allows the add() and remove() methods to be executed in ________________ time .
4) We discussed our **divide and conquer sorting algorithms** by answering **two questions**:
   a) How do we ____
   b) How do we ____
5) The **get(i)** method for the Java ArrayList runs in time _______________ and for the Java LinkedList runs in time ________________.
6) A tree node that has a parent and at least one child is called a(n) _______________________ 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**.
2) The **Simple Quicksort** algorithm has both a **worst case** runtime and an **average case** runtime of **$O(N^2)$.**
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.

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.
5) A **full** binary tree with a height of h will have $2^h - 1$ nodes.

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.

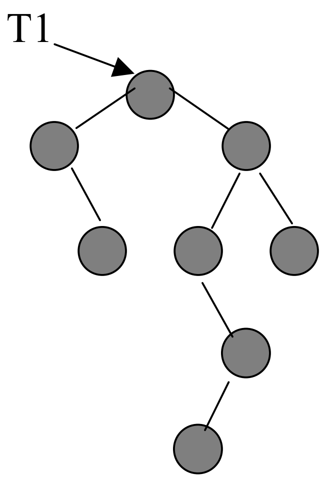
2) Consider the MergeSort and QuickSort sorting algorithms. Give the Big-O run-times for both algorithms (average and worst cases). In real terms, which algorithm has the faster run-time and why? Be specific.

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**.

   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**.

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;
        }
    }
}
```

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)
{
```