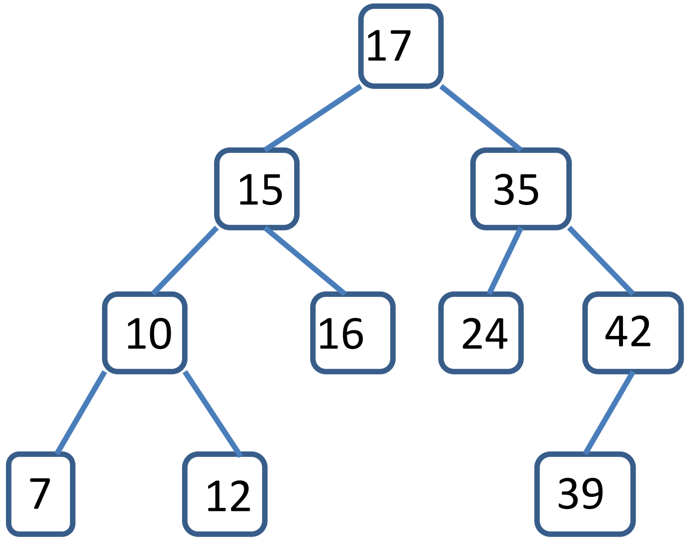
Review Questions

1. Java has an AbstractList class and a Comparable interface. Why is one an abstract class and the other an interface?

2. Give an algorithm for deleting a node in a binary search tree. The algorithm should be able to handles nodes such as 24 (no children), 42 (one child) or 17 (two children) in the following tree:



3. A grocery store hires you to maintain their inventory in a database. You need to use as keys the Universal Product Code stored in the barcode. The UPC consists of 12 numeric decimal digits, such as the following one for Kellogg's Raisin Bran: 038000596636. The value corresponding to each key is the number of items of this product that you have on hand. Here are some options:

   You could store the counts in an array indexed by the UPC; for example, if the store has 20 boxes of Kellogg's Raisin Bran the entry at index 38000596636 would be 20.

   You could make an array of base type Entry, which has a UCP-count pair, with an entry for each of the 10,000 products stocked in the store.

   You could make a HashMap where the keys are UPC's and the values are Integer counts.

   You could make a TreeMap where the keys are UPCs and the values are Integer counts.

Which would you use, and why?

4. Here is a Node type for a binary search tree that holds integer data:

```
class Node {
        int data;
        Node leftChild, rightChild;
}
```

Give a definition of a recursive function

```
Node insert( int x, Node p)
```

that inserts value x into the tree rooted at p.

5. I have a large collection of n data items already stored in an AVL tree. I want to use this tree as a Priority Queue. How long will each of these queue operations take?

   Peek (i.e., find the maximum element)

   Poll (find, remove and return the maxiumum element)

   Offer (x) (insert x into the queue)

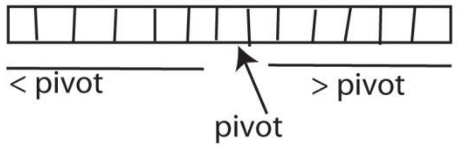
~~6. Here is a recursive function:~~

```
int f(int n)  {
        If (n == 0)
                return 1;
        else if (n == 1)
        return 3;
        else
                return 2*f(n-2)  + 3*f(n-1) + 5;
}
```

~~Give a dynamic programming version of f.~~

7. Your boss gives you a dataset of 1,000,000 items in na big array. He wants you to find the 100 largest values in this dataset as quickly as you can. If your program runs quickly enough you get promoted and get to take that dream vacation to Fiji. If your program runs too slowly you get fired and have to go to work at MacDonalds. What technique will you use to find the 100 largest values? What is your estimate of its running time in terms of the number of items in the dataset?

8. Here is an algorithm for finding the kth smallest element in an unsorted array of values. It works much like Quicksort – we choose a pivot value and rearrange (partition) thehe data so that all of the values less than the pivot are to its left and all of the values greater than the pivot are to its right:



If the index of the pivot is larger than k recurse on the portion of the array between index 0 and the index of the pivot. If the index of the pivot is less than k look for the k-index(pivot) smallest value in the portion of the array to the right of k. Of course, if index(pivot) == k you are done.

What is the big-Oh worst-case running time of this on an array of size n?

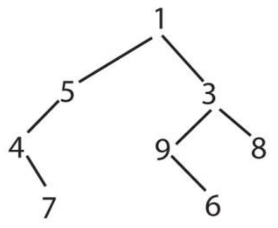
What is the average case running time on an array of size n?

Give an argument that your answer for the average cases is correct.

9. I have a tree based on the following Node class:

```
class Node {
        int data;
        Node leftChild, rightChild;
}
```

Give a procedure void Print(Node p) that prints this tree in breadth-first order: print the root, print the root's children, print the children of the root's children, and so forth. For the tree



It prints 1 5 3 4 9 8 7 6

10. Write method int largest(LinkedList<Integer> L). Your method should run in time O(N), where N is L.size( ). Note that you do not have access to the Node structure of Java’s LinkedList class.

11. How should I think about a recursive function?

Just as there is no one way to write a function, there is no single way to write a recursive function. There are, however, a few things to think about that will help:

A. Find the base case or cases.

Every recursive function has some value of the argument for which it can give an answer without recursing. For the factorial function that argument is 0: we don’t have to compute to find that factorial(0) is 1. For the Towers of Hanoi puzzle we move 0 disks by doing nothing. For mergeSort(A, low, high) we only do something if high > lo.

B. Look for the recursion.

You need to find a step that will leave you with a similar but simpler version of the same problem. “Simpler” might just mean that the arguments are 1 step closer to the base case. For factorial we compute factorial(n) by returning n*factorial(n-1). The base case is when n is 0; the recursive argument n-1 is one step closer to the base case than the initial argument n. For MergeSort the base case is when we are only sorting 0 or 1 elements of the array (so there is nothing to do). The recursive case splits the array in half (so both halves are closer to the base case), recursively sorts both halves, and then merges those halves back together.

C. Trust the recursion

Recursion sometimes feels like you are cheating and not really solving anything. “Sorting n items by sorting n/2 items” sounds like you are playing a game with words. Don’t worry about it. If your recursive calls really do lead you to the non-recursive base cases you will solve the problem.

For example

Consider the last question from Exam 2, where you need to write heightCount(h), which is the number of nodes in the tree that have height h. There are two base cases: If the tree’s height is h there is exactly one node in it of height h – the root. If the tree’s height is less than h then none of its nodes have height h.

That leaves the recursive case to cover trees that have height larger than h. The height of a tree is the height of its root, so to find nodes with height exactly h we need to look to the children: leftChild.heightCount(h) + rightChild.heightCount(h) This is an unusual recursion where the recursive calls have the same argument h as the original call but the base case refers to the tree you call it on: the base case is for trees whose height is less than or equal to h. If the original tree is taller than h its children will be closer to the base case than the original tree, and that is what we need.

12. Big- Oh estimates for our data structures

Most of the algorithms (other than sorting) we discussed this term were connected to various data structures. Here is a catalog of them:

ArrayLists
- get(i) is O(1)
- add(elt) is O(1) unless you need to expand the array
- add(i, elt), remove(i) are O(n)
- sorting is O(n*log(n)) if you choose a good algorithm

Stacks
- push and pop are both O(1) with either an ArrayList or linked implementation

Queues
- enqueue and dequeue (or offer and poll) are O(1) with a linked implementation; one of them is O(n) with a naïve ArrayList implementation

LinkedLists

Adding to the front or rear is O(1).

get, adding at an index, remove are all O(n).

Iterating through all of the elements of a LinkedList using indexes is $O(n^2)$. Using an iterator it is O(n).

BinarySearch Trees

Adding, removing and searching for an element are all O(n)

AVL trees use the same algorithms as BSTs but finish by rebalancing the tree so adding, removing and searching are all O(log n)

TreeMaps are AVL trees with (key value) pairs in each node.

HashMaps have worst-case O(n) time for adding and searching; we don't usually have a remove operation. If you keep the load factor from getting too big the *average* case add and search times are constant.

Priority Queues

offer and poll are both O(log n)

You can turn an array into a heap-based priority queue in time O(n)

Sorting Algorithms

BubbleSort, SelectionSort, and InsertionSort are O(n^2)

MergeSort, HeapSort, and QuickSort are O(n*log(n)).

Shortest path algorithms in graphs

- Unweighted graphs: O(|E|)
- Weighted, but only non-negative weights
  - (Dijkstra’s Algorithm) O(|E|*log(|V|))
- Shortest paths with arbitrary weights,
  - or longest paths with arbitrary weights
  - (Bellman-Ford Algorithm) O(|E|*|V|)