CS 151
Final Exam Solutions
Fall 2021

Unless you qualify for extended time on exams, **you have 2 hours to complete this exam** once you start it.

The 8 numbered questions are equally weighted.

If you forget the name of something (oh, what does Java call the length of a String??) just write a note saying "I'm going to call this X" and then use that.

Please do not write on the backs of the pages. If you need more space for a question there are two blank pages at the end.

After you have finished the exam please indicate whether you followed the Honor Code on the exam.

I ☐ did ☐ did not

adhere to the Honor Code while taking this exam.

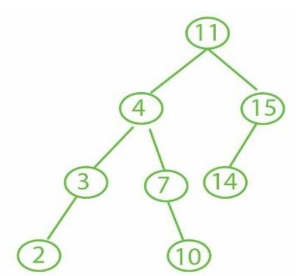
I started the exam at time ________ and finished at time _________

____

Signature

1. Here is a list of data: 11 4 15 7 3 14 2 10. For each of the following structures I will walk through the data list in order, add each item to the structure and then go into a loop in which I remove elements one at a time from the structure and print them as I remove them. **In what order do I print the items for**
   a) **A stack**. Using push( ) to add to the structure and pop( ) to remove.
      This reverses the order: 10 2 14 3 7 15 4 11
   b) **A queue**. Using offer( ) to add to the structure and poll( ) to remove.
      Same order: 11 4 15 7 3 14 2 10
   c) **A priority queue**. Using offer( ) to add to the structure and poll( ) to remove.
      Increasing order: 2 3 4 7 10 11 14 15

   d) In the add stage, I insert the values into a **BinarySearchTree** that starts off empty. So 11 becomes the root and I insert the other values around it. Skip the remove stage and instead give an **inorder traversal** of the tree.
      An inorder traversal of a BST gives the data in increasing order (or just look at the tree in part (e)): 2 3 4 7 10 11 14 15
   e) This is the same as (E) only I do a **preorder traversal** of the tree.



11 4 3 2 7 10 15 14

   f) **In the add stage I form a hash table of size 8** (the data fits; you don't need to resize the table) with linear open addressing, using each data value as its own hash code (so the **hash value is the remainder when we divide the value by 8 -- 4 hashes to index 4, 11 to index 3, 15 to index 7, etc.**) In the print stage I print the data at index 0, then the data at index 1, then index 2, etc.

0	1	2	3	4	5	6	7
7	10	2	11	4	3	14	15

Answer: 7 10 2 11 4 3 14 15

2. In each part **give a Big-Oh estimate of the worst-case time it takes to complete the operation in the given strtucure**
   a) Inserting into an AVL tree with n nodes

      O(log(n))

   b) Finding, then removing, a node from a Binary Search tree

      O(n) Remember: a BST is not necessarily balanced

   c) Finding an element in a sorted ArrayList with n elements. Here “finding” means determining if the list contains an element with a particular value.

      O(log(n)): Binary Search

   d) Finding an element in a sorted LinkedList with n elements.

      O(n) You can’t do binary search on a linked list

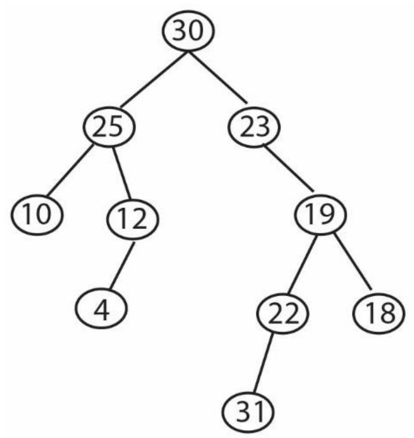
   e) Polling a Priority Queue with n values.
      O(log(n)) That’s why we created priority queues

   f) Inserting an Edge in the graph structure we used in Lab 9, if there are n vertices in the graph and we are given the names of the source and destination nodes of the edge. For this one give the average-case time rather than worst-case time.

      O(1): We have to look up both endpoints in the vertex table (a hashmap), but those are constant-time operations. Then we add the edge to the source vertex’s edge list (a linked list) but that is also a constant time operation.

3. Here is a binary tree based on the following Node type:

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



}

A breadth-first traversal of a binary tree lists the root, then the root's children, then their children, and so forth. A breadth-first traversal of the example tree is

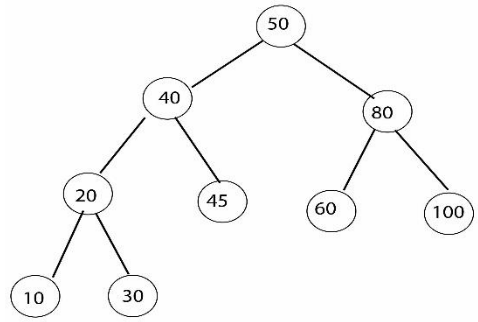
30 25 23 10 12 19 4 22 18 31

Write the method **void PrintBreadthFirst(Node root)** which prints a breadth-first traversal of the tree with the given root.

```
void PrintBeadthFirst( Node root ) {
        LinkedList<Node> queue = new LinkedList<Node>();
        queue.offer(  root );
        while( queue.size() > 0) {
                Node x = queue.poll( );
                 System.out.print( x.data );
                 if (x.leftChild != null)
                         queue.offer(x.leftChild);
                 if (x.rightChild != null)
                         queue.offer(x.rightChild);
       }
       System.out.println( );
}
```

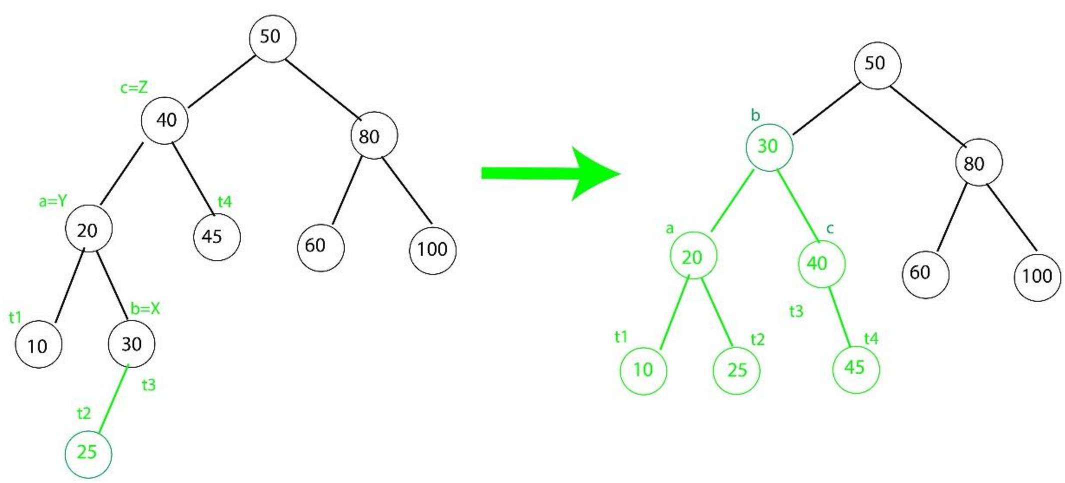
If you want to worry about the case where the initial root is null it is easy to addd a condition to handle that.

4. Here is an AVL tree:



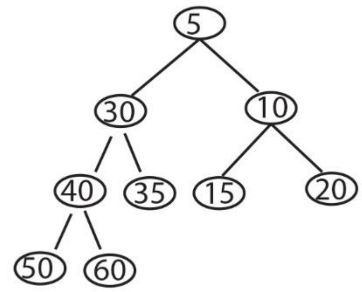
Give either the AVL tree or a level-by-level listing of the AVL tree that results from inserting value 25 into this tree. If you can't easily draw a tree, a "level-by-level listing" of a tree lists the root on the first level, all of the children of the root on the second level, the grandchildren of the root on the third level, and so forth. A level-by-level listing of the tree shown is

50
40 80
20 45 60 100
10 30



Level-by-level this is 50
30 80
20 40 60 100
10 25 45

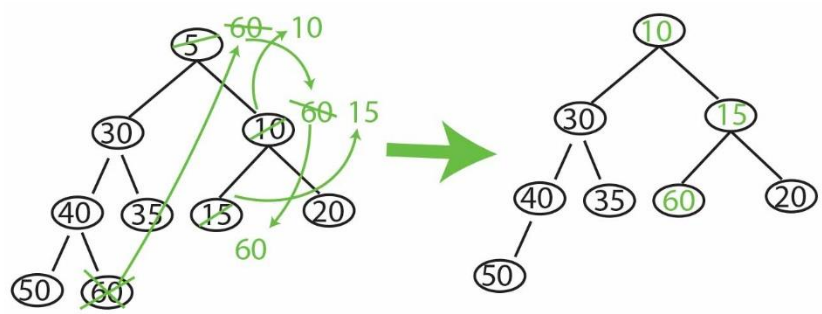
5. Here is a picture of a binary Heap represented as a tree:



If you prefer this could be represented as an array:

	5	30	10	40	35	15	20	50	60

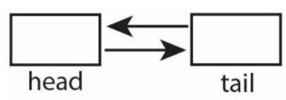
Give the heap (either the array or the tree) that results from polling the heap to remove the root value 5.



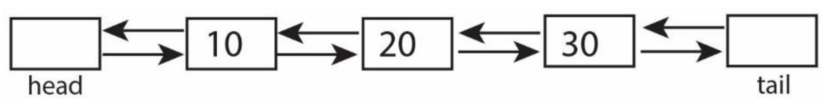
6. We have a doubly-linked list based on the following node structure:

```
class Node {
       int data;
       Node next;
       Node previous;
}
```

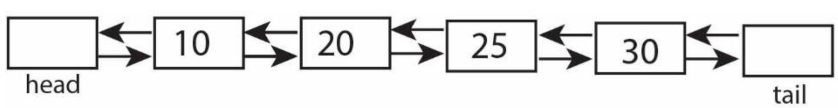
Our list has sentinel nodes with no data at each end. Here is the empty list created by the list constructor:



Here is a list with three elements:



Give code for the method void InsertInOrder(int v). If the list is sorted this inserts v in the proper location for it to remain sorted; if the list is not sorted when this is called it can insert v anywhere. If we call InsertInOrder(25) with the 3-element example above it produces




```
void InsertInOrder( int v ) {
        Node r = head.next;
        while (r != tail && v > r.data)
                r = r.next;
        // we want to insert v between r.previous and r
        Node p  = r.previous
        Node q = new Node(v);
        p.next = q;
        q.previous = p;
        q.next = r;
        r.previous = q;
}
```

7. In Lab 3 we solved mazes. A maze in that lab was a rectangular grid of Squares, where a Square could be the maze's entrance, its exit, a wall, or an open space. A solution was a path of open squares connecting the entrance to the exit. We wrote two programs to solve mazes: MazeSolverStack and MazeSolverQueue. They both worked but you have learned so much since then. Give an algorithm in English for finding the *shortest* path from the entrance to the exit of a maze.

   If you make a graph out of the maze (give a node for every open Square and also the entrance and exit, and two edges for each pair of adjacent Squares, then our MazeSolverQueue program and our shortest path for unweighted graphs algorithm do exactly the same thing – they walk through the same Squares in the same order. In other words, MazeSolverQueue finds the shortest path. MazeSolverStack does not necessarily find the shortest path. For answers I was looking for either a description of the MazeSolverQueue algorithm or something like our shortest path algorithm for unweighted graphs. A queue needs to be involved.

8. Suppose you have a Queue<E> implementation and you need to add the following method to it: boolean MoveToFront(E elt). If object elt is in the queue this method removes the first instance of it from the queue, moves it to the head of the queue (so it will be the next thing returned by poll())and returns true. If elt is not in the queue MoveToFront(elt) doesn't change the queue at all but returns false. Give an algorithm in English for MoveToFront( ). You can use any data structures you want but you cannot make any assumptions about the underlying structure of Queue<E>.

   The only operation that lets you explore a queue is poll( ) and that removes data from the queue; we need a place to store that data. I would do this in another queue, which I will call queue2. In a loop poll the first queue and offer those values to queue2 until you either find elt or get to the end of the first queue. If elt is found don't add it to queue2, but with a loop poll the rest of the first queue into queue2. When the first queue is empty offer it elt, then offer it each element polled from queue2 and return true. On the other hand, if elt is not found use a loop to repeatedly poll queue2 and offer the polled value to the first queue until queue2 is empty. In this case return false.

   You can do the same thing with an ArrayList instead of a queue, but you end up using the ArrayList as a queue so that doesn't make a lot of difference.