1. Here is a binary tree:

a) Give an in-order traversal of this tree.
   5 10 15 20 30 40 70 90

b) Give a pre-order traversal of this tree.
   20 10 5 15 90 40 30 70

c) Give a post-order traversal of this tree.
   5 15 10 30 70 40 90 20
1. Here is a Binary Search Tree:

![Binary Search Tree Image]

We had an algorithm for removing values from a Binary Search Tree. **Give the tree, or a level-by-level listing of the tree, that results from removing the value 20 from this tree.**

**My solution:**

![My Solution Image]
3. Here is an AVL Tree:

![AVL Tree Diagram]

Give either the AVL tree, or a level-by-level listing of the AVL tree, that results from inserting value 50 into this tree.

**My solution:**

![Level-by-Level Listing Diagram]
4. Here are string data and their hash values for a hash table of size 11 (indexed 0 through 10):

<table>
<thead>
<tr>
<th>String</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We will use linear open addressing for this, as we did for most of our work on hashing in class. Start with an empty hash table and insert the strings in alphabetical order (first A, then B, etc.).

**Give the index where each string ends up in the table.**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>D</td>
<td>G</td>
<td>C</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>A</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

A 9
B 0
C 3
D 1
E 10
F 8
G 2
H 4
5. I want to make an iterator for a hash table that uses linear open addressing. Let’s suppose the data we are hashing has base-type E and the table is based on an array of size CAPACITY. We need to produce an Iterator<E>. This has three methods:

A constructor
boolean hasNext()
E next()

Tell me in English how each of those methods could be implemented. You do not need to write any code.

The iterator’s next() method needs to move through all of the elements in the table. Consider the table you built for question 4. This has no entry at index 5, 6, or 7, so after next() returns the value at index 4 it jumps to the value at index 8.

I would do this by maintaining two integer variables.
  - currentIndex is the index of the item most recently returned by next()
  - nextIndex is the index of the next item that will be returned by next()

The constructor initializes currentIndex to -1 and nextIndex to the index of first non-null entry in the table.

hasNext() returns true if nextIndex < CAPACITY and false otherwise.

next() first sets currentIndex=nextIndex, increases nextIndex to the index of the next occupied entry in the table (This needs a loop. If there are no more occupied entries it sets nextIndex to CAPACITY), and then returns the value stored at currentIndex.
6. I want to produce code for MyList<E>: a singly-linked list with base type E that has a sentinel node at each end. We will use the node class
   
   ```java
   class Node {
       E data
       Node next;
   }
   ```

   The MyList instance variables are `front` and `back` which are both of type `Node`, and `size`, which is an int that stores the number of current entries. Here is the constructor:

   ```java
   MyList() {
       front = new Node();
       back = new Node();
       front.next = back;
       size = 0;
   }
   ```

   Here are pictures of two lists: on the left is an empty list built by the constructor, on the right is a `MyList<Integer>` with three data elements: 5, 10, and 15:

   ![List Diagram]

   Finally, suppose we have already written a method
   ```java
   Node getNth(int n)
   ```
   that returns the actual node (not just its data) at index `n`. For the list on the right `getNth(0)` returns the Node that contains 5, `getNth(3)` returns the sentinel `back`, and `getNth(-1)` returns the sentinel `front`. You do not need to write `getNth()`.

   **Using all of this, write the method**
   ```java
   void insert(E data, int index) throws IndexOutOfBoundsException
   ```

   This should insert `data` into the list so it has the given index, unless the index is bad, in which case it throws the exception.

   My code is on the next page
void insert(E data, int index) throws IndexOutOfBoundsException {
    if ( (index < 0) || (index > size) )
        throw new IndexOutOfBoundsException("Bad Bad Bad");
    else {
        Node p = getNth(index-1);
        Node r = p.next;
        Node q = new Node();
        q.data = data;
        p.next = q;
        q.next = r;
        size += 1;
    }
}

Note how the sentinel nodes help to eliminate special cases.