Activity 8-1

size() for MyArrayList

1. Below is the custom implementation of a custom version of ArrayList (taken from slides) called ***MyArrayList.***
2. Currently, it has a method ***add()***, which add an element to the end of the structure.

```
class MyArrayList<E> {
    int INITIAL_CAPACITY =10;
    E a[] = (E[]) new Object[INITIAL_CAPACITY];

    void add (E  x){
      if (a[a.length - 1]==null){
          int i=0;
          while (a[i]!=null)
              i++;
          a[i]=x;
      }
      else  {
          // increase size by 10
          int originalSize = a.length;
          int newSize = originalSize +10;
          E b[] = (E[]) new Object[newSize];
          b=Arrays.copyOf(a,newSize);
          a=b;
          a[originalSize]=x;
      }
    }

    E get(int index){
        return(a[index]);
    }
}
```

3. Add code to include a new method ***size()*** with the following signature to return the number of elements currently stored in the ***MyArrayList*** structure.

```
int size()
```

Activity 8-2

remove() for MyArrayList

1. Continue from Activity 8-1.
2. Add another method remove() to remove an element from the index.

```
void remove(index)
```

3. Check if the index is within range. Return 0 if ok. Otherwise, return -1.
4. Advance all elements after the deleted element, forward by 1 index.
5. Any special case to take care of?

Activity 8-3

Doubly Linked List

1. Below is the code for a simple custom linked list class ***MyLinkedList*** extracted from the slides.

```
class MyLinkedList<E> {
    Node<E> head = null;
    Node<E> current = null;
    Node<E> newNode;
    public void append (E x){
       newNode = new Node<E> (x);

       if (head==null) {
         // for very first mode
         head = new Node(x);
       }
       else {
          current = head;

          // track down to tail node
          while (current.next != null)
             current = current.next;
          // add in the new node
          current.next = newNode;
     }

     public String toString (){
          String s="";
          current = head;
          while (current!=null) {
              s += current.data.toString();
              current = current.next;
          }
          return(s);
      }
}

class Node<E> {
    E data;
    Node<E> next = null;
    Node (E data) { this.data = data; }
}
```

2. Add code to make it into a doubly linked list. A doubly linked list is a linked list not only can track the element from the '***head***', but is also able to track the elements from the '***tail***'.
3. Also, each Node must have an additional attribute called '***previous***'.
4. Hence, whenever an element is added to the end of the linked list using the existing ***append()*** method, the '***next***' and '***previous***' of the affected ***Nodes*** have to be handled properly..

Activity 8-4

Building BST and Pre-Order Traversal

1. Construct the BST with the following input sequence of integers (with the first number 35 as the root node):

   35, 67, 12, 25, 19, 29, 41, 7, 14, 32, 21

2. Write down the ***pre-order traversal*** sequence of the following BST, manually.

Activity 8-5

Pre-Order Traversal

1. Below is the code for a simple BST class ***MySimpleBST*** extracted from the slides.

```
class MySimpleBST {
    Node root = null;
    void append(int x) {
        Node newNode = new Node (x);
        if (root == null){ root = newNode; }
        else {
            if (newNode.data < root.data){
                root.left = newNode;
            }
            else if (newNode.data > root.data){
                root.right = newNode;
            }
        }
    }

    void postOrderTraversal(){
        if (root!=null){
            if (root.left!=null){
                System.out.println(root.left.data );
            }
            if (root.right!=null){
                System.out.println(root.right.data );
            }
            System.out.println(root.data);
        }
    }
}

class Node {
    int data;
    Node left = null;
    Node right = null;
    Node (int data) {
      this.data = data;
    }
}
```

2. Add a method ***preOrderTraversal()*** to print out the ***pre-order traversal*** sequence.