HW#3 Code File (Updated)

Clarification: the ListInterface<T> is NOT the Java library version, but our own given in the Linked List Code files under Week 1 in Catalyst.

```
public class SortedLList2<T extends Comparable<T>> implements ListInterface<T>{
    private Node2 firstDummyNode;  // Reference to first node of chain
    private Node2 lastDummyNode;
    private int  numberOfEntries;

    public SortedLList2()
    {
         firstDummyNode = new Node2(null);
         lastDummyNode = new Node2(null);
        initializeDataFields();
    } // end default constructor


    @Override
    public void add(T newEntry) {
        // CHANGE the LList2 version so it inserts in the correct place
        // so the list stays in order, BUT stop traversing at the lastDummyNode OR
        // when the newEntry <= current Node's data (using compareTo)
        // (don't forget to increment the numberOfEntries)

    }

    @Override
    public boolean add(int newPosition, T newEntry) {
        // COMPLETELY REWRITE the LList2 version so you
        // call add(T) passing newEntry, and ignore newPosition
        // and return true
    }

    @Override
    public boolean remove(T anEntry) {
        // CHANGE from Lab Ex. 4.2 Answers so it STOPS traversing
        //    when you reach the lastDummyNode
        //    OR when you either find anEntry (and USE compareTo)
        //    or if the currentNode's data is > anEntry (using compareTo)
        //         //
        //    if found, do as done in LList2 (including decrementing
        //    numberOfEntries) and return true,
        //    otherwise return false
        return false;
    }

    @Override
    public T remove(int givenPosition) {
         T result = null;                           // Return value

         if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
```

```
        {
            // same code if givenPosition is 1 or > 1
            Node2 nodeToRemove = getNodeAt(givenPosition);
            Node2 nodeBefore = nodeToRemove.getPrevNode();
            result = nodeToRemove.getData();     // Save entry to be removed
            Node2 nodeAfter = nodeToRemove.getNextNode();
            nodeBefore.setNextNode(nodeAfter);   // Remove entry
            nodeAfter.setPrevNode(nodeBefore);

            numberOfEntries--;                          // Update count
            return result;                              // Return removed entry
        }
        else
            return null;
    } // end remove(int)

    @Override
    public void clear() {
        initializeDataFields();
    }

    @Override
    public T getEntry(int givenPosition) {
        if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
        {
            return getNodeAt(givenPosition).getData();
        }
        else
            return null;
    }

    @Override
    public boolean contains(T anEntry) {
        // CHANGE the LList2 version so it traverses the
        //    list like in remove(T) so it STOPS traversing
        //    when you reach the lastDummyNode
        //    OR when you either find anEntry (and USE compareTo)
        //    or if the currentNode's data is > anEntry (using compareTo)
        //
        // return true if found, false otherwise
    }

    @Override
    public int getLength() {
          return numberOfEntries;
    }

    @Override
    public boolean isEmpty() {
        return numberOfEntries==0;
    }

    @Override
```

```
    public void display() {
        Node2 currNode;

        currNode = firstDummyNode.getNextNode();  // FOR LAB EXERCISE 4.2
        while( currNode != lastDummyNode )  // FOR LAB EXERCISE 4.2
        {
                System.out.println(currNode.getData());
                currNode = currNode.getNextNode();
        }
    } // end display

    // YOU FINISH THIS METHOD SO IT DISPLAYS
    // ALL THE DATA IN THE LIST BACKWARDS
    // (remember to ignore the dummy nodes' data)
    public void displayBackwards()
    {
    }

    // Initializes the class's data fields to indicate an empty list.
    private void initializeDataFields()
    {
        firstDummyNode.setNextNode(lastDummyNode);
        lastDummyNode.setPrevNode(firstDummyNode);
        numberOfEntries = 0;
    } // end initializeDataFields

    // Returns a reference to the Node2 at a given position.
    // Precondition: The chain is not empty;
    //               1 <= givenPosition <= numberOfEntries.
    // Returns a reference to the node at a given position.
    // Precondition: The chain is not empty;
    //               1 <= givenPosition <= numberOfEntries.
// CHANGE TO search forwards ONLY if givenPosition is between 1 and
//     numberOfEntries/2 (inclusive)
    private Node2 getNodeAt(int givenPosition)
    {
        if( givenPosition < 1 )
            return firstDummyNode;
        else
        if( givenPosition <= numberOfEntries )//CHANGE
        {
             Node2 currentNode = firstDummyNode.getNextNode();

             // Traverse the chain to locate the desired node
             // (skipped if givenPosition is 1)
             for (int counter = 1; counter < givenPosition; counter++)
                 currentNode = currentNode.getNextNode();

             return currentNode;
        }
        else
        { // CHANGE SO YOU WILL SEARCH FROM THE END BACKWARDS IF
          //   THE givenPosition is > numberOfEntries/2 and <= numberOfEntries
```

```
        // PUT SEVERAL LINES HERE

    }
    return lastDummyNode;

} // end getNodeAt

private class Node2
{
     private T    data; // Entry in list
     private Node2 next; // Link to next Node2
     private Node2 prev; // Link to previous Node2

     private Node2(T dataPortion)
     {
        data = dataPortion;
        next = null;
        prev = null;
     } // end constructor

      private Node2(T dataPortion, Node2 nextNode)
      {
         data = dataPortion;
         next = nextNode;
         prev = null;
      } // end constructor

      private T getData()
      {
         return data;
      } // end getData

      private void setData(T newData)
      {
         data = newData;
      } // end setData

      private Node2 getNextNode()
      {
         return next;
      } // end getNextNode

      private void setNextNode(Node2 nextNode)
      {
         next = nextNode;
      } // end setNextNode

      private Node2 getPrevNode()
      {
         return prev;
      } // end getNextNode

      private void setPrevNode(Node2 prevNode)
```

```
        {
            prev = prevNode;
        } // end setNextNode
    } // end Node2
} // end SortedLList2 class


public class Date implements Comparable<Date>{

    static final int MIN_MONTH = 1;
    static final int MAX_MONTH = 12;
    static final int MIN_YEAR = 1000;
    static final int MAX_YEAR = 9999;
    static final int [] DAYS_IN_MONTH =
        { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    private int month = 1;
    private int day = 1;
    private int year = 1000;

    public Date() {}

    public Date( int m, int d, int y )
    {
        setDate(m, d, y); // else leave default values
    }

    public static boolean isLeapYear(int y)
    {
        return (y % 4 == 0 && y % 100 != 0 || y % 400 == 0);
    }

    public boolean setDate( int m, int d, int y )
    {
        int isLeap = 0;

        if (y >= MIN_YEAR && y <= MAX_YEAR
            && m >= MIN_MONTH && m <= MAX_MONTH)
        {
            if (m == 2 && isLeapYear(y))
                isLeap = 1;
            if (d >= 1 && d <= (DAYS_IN_MONTH[m] + isLeap))
            {
                month = m;
                day = d;
                year = y;
                return true;
            }
        }
        return false; // leaves instance vars. as they were before
    } // end setDate

    public int getMonth(){ return month; }
```

```
    public int getDay(){ return day; }

    public int getYear(){ return year; }

    public String toString(){
        return month + "/" + day + "/" + year;
    }

    // COMPLETE the compareTo method:
    //    Return the correct int as described in the Comparable example file
    //    so it compares the years, then the months (if the years are equal),
    //    then the days (if the years and months are equal)
    // Remember, return 0 if all 3 are the same, an int > 0 if this > param,
    //    otherwise, return an int < 0
    @Override
    public int compareTo(Date param)
    {

    }
}



//-------------------------------------------------------------------------
//USE THE FOLLOWING IN YOUR MAIN class:
// Remember to import the correct package for Scanner and Files

    public static Scanner userScanner = new Scanner(System.in);

    // opens a text file for input, returns a Scanner:
    public static Scanner openInputFile()
    {
        String filename;
         Scanner scanner=null;

        System.out.print("Enter the input filename: ");
        filename = userScanner.nextLine();
            File file= new File(filename);

            try{
                scanner = new Scanner(file);
            }// end try
            catch(FileNotFoundException fe){
               System.out.println("Can't open input file\n");
            return null; // array of 0 elements
            } // end catch
            return scanner;
    }

    public static void testSortedLList2(SortedLList2<Date> dateList)
    {
        if( dateList== null || dateList.getLength() < 2 )
```

```
        {
            System.out.println("\nEither empty or not enough nodes in the list; no
testing done\n");
            return;
        }
        Date date1, date2, newDate;

        int middle = dateList.getLength()/2;

        date1 = dateList.getEntry(middle);
        date2 = dateList.getEntry(middle+1);

        System.out.println("Testing removing element #" + (middle+1)
                                    + ": " + date2);
        date2 = dateList.remove(middle+1);
        if(  date2 != null )
            System.out.println("Successfully removed: " + date2);
        else
            System.out.println("Error: unable to remove element #" + (middle+1));

        System.out.println("\nTesting removing element #" + middle + ": " + date1);
        date1 = dateList.remove(middle);
        if(  date1 != null )
            System.out.println("Successfully removed: " + date1);
        else
            System.out.println("Error: unable to remove element #" + middle);

        newDate = new Date();
        System.out.println("Testing adding default Date: "+ newDate);
        if( dateList.add(dateList.getLength()+1, newDate) )
            System.out.println("Default date successfully added!");
        else
            System.out.println("Error: unable to add default Date");
        System.out.println("\nNow the list has: ");
        dateList.display();
    }
```