COMP 410

List ADT (Implemented with Linked Cells)

Overview

You are going to build in Java a general List ADT (LIST) using linked cells to do it. This means do not use arrays to contain the elements stored in the List. It also means do not use any of the code from the Java Collections library to get your work done.

The LIST you will implement will be very similar to the ADT shown in your text. It will have an operation to put an element into the list, to take an element out, and to retrieve an element (without altering the data structure). It will have an operation telling how many elements are stored in the list, and a boolean to tell if it is empty or not. It has an operation to make a fresh, empty list. We have left out the operations shown in the text that search for elements by value.

The class Node below shows how we do linked cells in Java. A linked cell is an object (class instance) that has one (or more) references to other objects of the same class. So a list cell refers to another list cell (the next one in the linked chain).

Our implementation will be a doubly linked list. As you see in the Node class, there is a next field and a prev field. The next field points to the cell that follows the current cell in the list. The prev field points to the cell that came just before the current one in the list. To go through the list item by item in the "forward" direction you would start with the first cell and use the next field until you get to the end of the list. Since it is doubly linked, you can also traverse the list "backwards" by going to the last cell and following the prev links until you get to the first cell.

Note also that our implementation given in outline form below always has at least one node (cell) in it. This is a special "header" node cell we call a sentinel (see the PPT about the sentinel back on the main assignment page). We are using a single sentinel node as both a header (points to the first cell in the list) and as a trailer (last cell in the list points to the sentinel). When a brand new list is created and there are no elements stored, there still is the one sentinel node in the list structure. It is accessed by the "sentinel" field in the list object. The data value in the sentinel node doesnt matter, as it is not a data node. It serves as an item that will always be there so there are fewer null pointers in the list structure. It makes checking for end-of-list a bit easier, and it helps eliminate a lot of null pointer errors. In essence an empty list is not a "null", but a single cell with no next and no prev cells. We will (better) detect empty list by checking the size method for 0.

We ask that you use the boilerplate code provided and work out of it. You should copy and paste in the boilerplate below and structure it as we described.

Project,package,zip names matter!! It's how we grade your code! A good reference if you get stuck is your text book (pg 75-82).

What to Hand-in

Hand in your code via Sakai, and follow the instructions there for how to remove various parts and zip it all up.

Interfaces

For uniformity, to aid grading, you must write your program with these specific structures.

Interface LIST_Interface

```
/**
 * COMP 410
 *
 * Make your class and its methods public!
 * Don't modify this file!
 *
 * Your LinkedList implementation will implement this interface.
 *
*/


package LinkedList_A1;
/*
  Interface: The LIST will provide this collection of operations:

  clear:
    in: nothing
    return: void
    effect: list is left with size 0, no elements in it,
            consists of just the original root Node
  size:
    in: nothing
    return: number of elements stored in the list
    effect: no change to list state
  isEmpty:
    in: nothing
    return: boolean, true if the list has no elements in it, true is size is 0
    effect: no change to list state
  insert
    in: a double (the data element), and an int (position index)
    return: boolean, return true if insert is successful, false otherwise
    effect: the list state will be altered so that the element is located at the
```

```
            specified index; the list has size bigger by one; all elements that were
            at the specified index or after have been moved down one slot
    error: if index is beyond list size range return false
           valid inserts take place either at a location where a list element
           already exists, or at the location that is one beyond the last element
  remove
    in: an int (the index of the element to take out of the list)
    return: boolean.. return true if the remove is successful, false otherwise
    effect: list state is altered in that the Node at the specified index is decoupled
            list size decreases by one
    errors: what if specified index is not in the list? return false
  get
    in: an int, the index of the element item to return
    return: double, the element stored at index, or Double.NaN
    effect: no change in state of the list
    error: what if index is not in the list? return Double.NaN
*/

// ADT operations

public interface LIST_Interface {
    public boolean insert(double elt, int index);
    public boolean remove(int index);
    public double get(int index);
    public int size();
    public boolean isEmpty();
    public void clear();
}
```

Class LinkedListImpl

```
/**
 * COMP 410
 *See inline comment descriptions for methods not described in interface.
 *
*/
package LinkedList_A1;

public class LinkedListImpl implements LIST_Interface {
  Node sentinel; //this will be the entry point to your linked list (the head)

  public LinkedListImpl(){//this constructor is needed for testing purposes. Please don't modify!
    sentinel=new Node(0); //Note that the root's data is not a true part of your data set!
  }

  //implement all methods in interface, and include the getRoot method we made for testing purposes. Feel free to implement private helper method

  public Node getRoot(){ //leave this method as is, used by the grader to grab your linkedList easily.
    return sentinel;
  }
}
```

Class Node

```
/**
 * COMP 410
 * Don't modify this file!
*/

package LinkedList_A1;

public class Node { //this is your Node object, these are the Objects in your list
  public double data;
  public Node next; //links this node to the next Node in the List
  public Node prev; //links this node to the preceeding Node in the List (ie this Node is the prev Node's next node)
  public Node(double data){
    this.data=data;
    this.next=null;
    this.prev=null;
  }
  public String toString(){
    return "data: "+data+"\thasNext: "+(next!=null)+"\t\thasPrev: "+(prev!=null);
  }

  /*  Below are "getters" for our testing purposes. Please do not modify.
      I would advise just referencing the fields of your Nodes since
      they are public
  */
  public boolean isNode(){ //testing purposes please do not touch!
    return true;
  }
  public double getData(){ //testing purposes please do not touch!
    return data;
  }
  public Node getNext(){ //testing purposes please do not touch
    return next;
  }
  public Node getPrev(){ //testing purposes please do not touch
    return prev;
  }
}
```

Class LinkedListPlayground

```
package LinkedList_A1;

public class LinkedListPlayground {

  public static void main(String[] args) {
    /*
     here you can instantiate your LinkedList and play around with it to check
     correctness. We've graciously also provided you a bit of extra test data for debugging.
     It doesn't matter what you have in here. We will not grade it. This is for your use in testing your implementation.
      */
    test1();
```

```
    test2();

  }

  public static void test1(){
    // example test cases
    LinkedListImpl L= new LinkedListImpl();
    System.out.println(L.isEmpty());
    printList(L);
    L.clear();
    System.out.println(L.isEmpty());
    printList(L);
    System.out.println(L.sentinel.toString());
    L.insert(3.3,0);
    System.out.println(L.isEmpty());
    printList(L);
    System.out.println(L.sentinel.next.toString());
    L.insert(3.4, 0);
    L.insert(3.5, 0);
    L.insert(3.67, 1);
    L.insert(3.357, 0);
    L.insert(3.333, 4);
    System.out.println(L.size());
    printList(L);
    L.remove(3);
    System.out.println(L.size());
    printList(L);
    L.clear();
    L.insert(3.4, 0);
    L.insert(3.5, 0);
    L.insert(3.67, 1);
    L.insert(3.357, 0);
    L.insert(3.333, 3);
    L.remove(0);
    System.out.println(L.size());
    printList(L);
  }

  public static void test2(){
    // example test cases
    LinkedListImpl L= new LinkedListImpl();
    L.insert(3.4,0);
    L.insert(3.5,1);
    L.insert(3.67,2);
    L.remove(0);
    System.out.println(L.size());
    printList(L);
  }

  public static void printList(LinkedListImpl L){
    //note that this is a good example of how to iterate through your linked list
    // since we know how many elements are in the list we can use a for loop
    // and not worry about checking the next field to see if we hit the end sentinel
    Node curr=L.sentinel.next; // the first data node in the list is the one after sentinel.
    System.out.print("sentinel");
    for(int i=0; i<L.size(); i++) {
      System.out.print(" --> " + curr.data);
      curr=curr.next;
    }
    System.out.println();
  }
}
```

Class RunTests

```
/**
 * COMP 410
 * Don't modify this file!
 * This file is optional and is only needed to run the Oracle if
 * it is in your build path and the jar is in your project.
*/

package LinkedList_A1;
import gradingTools.comp410s19.assignment1.testcases.Assignment1Suite;

public class RunTests {
  public static void main(String[] args){
    Assignment1Suite.main(args);
  }
}
```

Notes on Operations and Objects

size

The size operation simply returns the count of how many elements are in the list. Do not count the sentinel. A brand new empty list will have a sentinel node, but is numElts field should be 0. size will return 0 if the list is empty, and always return a 0 or greater. Note the isEmpty() is synonymous with size()==0 .

insert and remove

Note that we wish for you to look out for the edge case of inserting or removing in an index that is not reachable (ie <0 or > size of list) We ask that you return false in these cases. Also when inserting and removing be careful to **connect your nodes to the next (and prev)** nodes inside the list. You don't want to get a null pointer when iterating, inserting, and removing. Remember the special case of inserting the very first data node... you will have to alter both the next and prev fields of the sentinel. You will have a similar special case when removing the last data cell and changing the list from size 1 to size 0.

Testing and Output

For testing, use your own test cases in LinkedListPlayground class. Think in detail about that tests will cover the full range of behavior your list might go through in use, and make sure your code handles those situations correctly. We also encourage using the oracle program for incremental development. Instructions on using the oracle will be in the form of the ppt presented in class (and posted online).

There is no specific output or runnable class required for this assignment. We assume you will use the "print" capability provided that will assist you in seeing how your code is working.

For grading, we will be using a grading program that will instantiate your objects and perform the LIST ADT operations on it. We will then observe if the functionality of the data structure is as we requested. Be sure to test beyond the scope of the Oracle as mentioned in class! The order in which ADT operations are performed and what data is entering your list will be different in the official grading script.