1. Introduction

In this project exercise, we introduce the concept of an abstract linked list, which is a list of nodes. We call that ADT as `LinkedList<E>`, representing a list of nodes. A `node` is a dataholder place that is of the following type:

```
public interface Node<E> {
     E getElement();
     void setElement(E e);
}
```

A node can hold one data element of type E; and supports operations to set the element value or to access the value of a `Node` object.

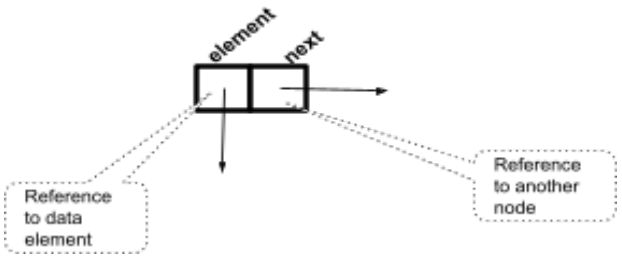
In this project, we work with different implementations of the `LinkedList<E>` ADT, which can later be used as data structures to implement other collection data types of higher abstraction; which is something that we also do in this activity. For now, we shall give an overview of each particular implementation, then we'll talk in more detail.

A *linked data structure* consists of physical memory locations in which data elements (or references to) of some collection instance are stored. These memory locations also include references to other such areas holding other elements of the collection instance. These areas are objects by themselves and are usually referred to as **nodes.** The memory assigned to all nodes in the data structure are not necessarily located in a contiguous memory area, as it is the case of the memory areas for positions in an array. Notice that nodes are not elements of the collection, they represent a hidden implementation feature; and, once again, they are used to hold the elements that are part of a collection instance and to link them all together as one entity. So, *a node actually consists of two parts*: part of *data* and part for *links* (or references; there could be one or more) *to other nodes.*

We will describe this further, but just as an initial illustration, let's consider the simplest type of node. That is a node in which there is a reference to an element and a reference to another node of the same type. This type of node can be used to implement the so-called *singly linked lis*t data structure as we shall see. In this case, the nodes can be implemented as objects of the following data type:

```
... class SNode<E> implements Node<E> {   // Node type for singly linked lists
     private E element;
     private SNode<E> next;
     ... constructors + getters + setters ...
}
```

In this case, each node is an object consisting of two fields: reference to a data element and reference (or link) to another node. It is usually pictorially represented as in the next figure.



A node on a singly linked list data structure

All those nodes that are reachable by following the links (from one node to the next) are part of the current data structure holding the elements of a particular collection; in this case, a list. Each node corresponds to one element of the list; the element (or reference to) being stored there as the element of the node. Notice that nodes are to the linked data structures what array slots are to an array-based data structure. Positions in the array contain, or have references

to, elements in the particular collection instance. In this case, each node contains, or has references to elements in the particular collection instance; but also, each has references to other nodes that may contain data pertaining to the particular collection instance that is represented.

A class implementing the `LinkedList` ADT will provide its own internal implementation of the `Node`. It will also contain reference to at least one of the nodes (initial nodes) of the current instance. In most cases, if the current instance is empty those references are `null` (Exception to this, as we will see, is when dummy nodes are included.). Through the links of those initial nodes (or at least, some of them), and eventually other accessible nodes, the whole set of nodes pertaining to a particular instance can be reached. (Note: some implementations may include marker or sentinel nodes, which are dummy nodes that will always exist and that are commonly used to mark boundaries of the data structure, hence simplifying the implementation of some operations).

When a collection data type is implemented using an array, the implementing class includes reference to the array instance, and the positions in the array hold elements (or references to elements) in the particular collection instance. When a collection data type is implemented using a linked list, with the abstraction to be studied here, what we would need is a reference to an instance of `LinkedList` based on any of the implementations to be studied. And the nodes of that internal list would contain the elements (or references to elements) in that particular collection instance that is represented at the moment. For some types of collections, different versions of the `LinkedList` would give different efficiency measures.

About this project Activity

In this project activity, we will work with a specification of `LinkedList<E>` ADT, which can be treated as a *list of nodes*. This will eventually allow us to use it to implement collections without having to handle links directly and in which nodes are seen, from outside, as places to hold data. For that kind of abstract linked list we will be able to provide different implementations, some of which are described below.

- **`SLList`** - Singly linked list in which there is reference to the first node. One to implement this is provided. All nodes in the list are used to store data; that is to say that all nodes are data nodes.
- **`SLFLList`** - Singly linked list in which there are references to its first node and to its last node. All nodes are data nodes.
- **`SLDHList`** - Singly linked list with "dummy header node", used as a marker node. It has reference to the dummy node. The node following (next node) the dummy node, if any, will be the first data node (which stores the first value in the list). Such dummy node will always be there, even when the list is empty (in this case, the only node).
- **`DLList`** - Doubly linked list in which there is a reference to the first node in the list and each node has a reference to the node physically preceding it (except the first) and the node that follows it (except the last). All nodes are data nodes. NOTE: An important advantage of this type of list is that from each node we can easily move to its predecessor node (if any), or its successor node (if any). This is a great advantage over the singly linked list in the implementation of some operations.
- **`DLDHTList`** - This corresponds to the doubly linked list with "dummy header node" and "dummy trailer node." The list has reference to each one of the dummy nodes. If the list is empty, it only contains the two dummy nodes ... If it's not empty, data nodes correspond to a doubly linked list between the two dummy nodes. The first data node follows the first dummy node (header). The last data node is followed by the second node dummy (trailer) ...

We will consider a specification of the linked list as an abstract data type (**`LinkedList<E>`**), describing a data structure in which we will be able to store a large number of data elements of type **`E`**. We will then work with different implementations of such ADT.

2. Linked List as a Collection of Nodes

In this section, we describe in more detail the abstract version of the linked list: **`LinkedList<E>`**. The purpose of this abstraction is, among other things, to be able to concentrate simply on operations with nodes, enabling its use as a data structure for implementing other abstract data types representing collections.

The linked list is a data structure used to store many different values concurrently. Remember that this can also be done with the array, so the linked list provides another way of storing a large number of data elements concurrently. But generally, linked structures have an immediate advantage over the array, allowing more efficient use of memory required to implement collections based on them.

A linked list is a sequence of zero or more nodes. From the abstract perspective of a linked list, all of its nodes are data nodes. If the list has n nodes (That is, has the capacity to store n data objects.), then we say it has a first node, a second node, ..., up to an n-th node. Seen in another way, each node represents a position in the linked list: first position (first node), second position (second node), ..., last position (last node). Note that the node in the first position is not preceded by any other node. Likewise, the last node is not followed by any other node[1]. If the list has more than one node (assume n nodes), then nodes different from the first node and from the last node will always have a preceding node (previous node) and a succeeding node (next node). Some illustrations will be included shortly. We will also see that in some implementations there are some additional physical nodes in the list that are used to control and not for data storage (these nodes are called "dummy" in the initial descriptions given). Notice that similar to the case of the array, the first node (or memory slot in the linked list) is identified as in position 0, and so on. It does not have to be this way, but it is perhaps more natural given many other instances in which position 0 refers to the first position, position 1 to the second position, and so on.

In this exercise we want to achieve various implementations of what we conceive as a linked list. In addition, each of you is expected to provide your own programs to make the necessary testing on implementations. Any such implementations will then be available to be used as a data structure in the implementation of abstract data types corresponding to some type of collection. Without the linked list (or linked structures in general), the only option we would have to implement collections would be the array. With these implementations, we will then have more options available when deciding on a data structure to use during the implementation of a collection ADT; that is part of the process of implementing an ADT: deciding upon the data structure to use.

Let us go more formal. The *linked list* is a dynamic structure, in which the memory space occupied by an instance is dynamically adjusted as needed. From our current perspective, we can consider an abstract description of the linked as a structure that must allow the following operations. They are aimed to give the linked list all the functionality needed for them to be used in implementations of data types corresponding to collections of data elements. Operations are:

- Operation to instantiate empty linked - constructors
- Operation to determine the size of the linked list - the number of nodes for storing data
- Operations to access node
    i. to access the first node in the linked list
    ii. to access the node that precedes a given node (whose reference is given)
    iii. to access the node that follows a given node (whose reference is given)
    iv. to access the last node
- Operations to insert new node
    i. to add a new node to the beginning of the linked list
    ii. to insert a node after a given node (whose reference is given)

[1] Internally, there might be other nodes that are not used as data holder places but as some sort of marker nodes to facilitate the coding of some algorithms. We will refer to those nodes, if any, as dummy or sentinel nodes. We will see why they might be useful.

iii. to insert a node before a given node (whose reference is given)

- Operations to eliminate or remove node
  i. to remove a given node (whose reference is given)
- Operation to create a new node of type required by the linked list implementation....

Note that the described operations do not work with data that is stored in the nodes. In fact, the linked list itself, as we understand it in this case, is a collection in which its data elements are the nodes: a collection of nodes. On the other hand, in other more abstract types in which one of these types of linked lists are used, the data nodes are used to store data: the elements of a particular instance of the collection. You must clearly understand this difference!

3. Operations on the `LinkedList`

Let us now have a formal description of the operations on the **`LinkedList`** ADT. All implementations must provide these operations, in addition to make it **`Iterable<E>`**, where **`E`** represents the data type of its elements. As already said, in some implementations of linked lists there are nodes whose sole purpose is to mark. Such nodes are not used to store data, they are "dummy" nodes (or sentinel/control nodes) with the particular purpose suggested above. Its purpose will be clear when you see the corresponding description in which they are used. Whenever we talk about a *"node that can store a data"* (or simply: "data node") we will be referring to any node that is not one of those dummy. In each of the following specifications you must be well aware as to what the parameters represent, and what is assumed about them in each case: their *preconditions.* (These preconditions establish how to correctly use the particular operation when using the LinkedList as a data structure to implement a more abstract collection data type.) The algorithms must work properly under such assumptions. If the assumptions are not met when the particular operation is invoked, as per the specifications, there may not be a correct action by the implemented operation and the result of the operation may be unpredicted. This means that you pass the hot potato to whoever uses these functions. *Whoever uses them should know how to use them correctly.* (This is similar to what is assumed for cars: whoever drives it, must know how to drive.... must know how to use it correctly.).

- Default constructor - all implementations must provide a default constructor that initializes the list as empty. Additional constructors may be added.

- **`int length()`** - returns integer value corresponding to the number of data nodes that the current instance of the linked list has.

- **`Node<E> getFirstNode()`** - returns reference to the first data node in the linked list. Returns null if the list is empty. The operation does not alter the data element reference in the affected node or in any other node in the linked list.

- **`Node<E> getLastNode()`** - returns reference to the last data node in the linked list. Returns null if the list is empty. The operation does not alter the data element reference in the affected node or in any other node in the linked list.

- **`Node<E> getNodeBefore(Node<E> target)`** - returns a reference to the data node that precedes the data node referenced by **`target`** in the linked list. As a precondition, it is assumed that **`target`** corresponds to a data node in the current instance of the linked list; otherwise, as briefly said before, an unpredictable result may happen (usually an abrupt end of the program in which it is being used). Later, one may try to capture possible problems by declaring other exceptions... but think of possible problems ... and ask us... Returns null if **`target`** is the first data node of the list. The operation does not alter the data element reference in the affected node or in any other node in the linked list.

- **`Node<E> getNodeAfter(Node<E> target)`** - returns reference to the data node that follows the data node referenced by **`target`** in the current instance of the linked list. It is presumed that **`target`** corresponds to a valid data node in this list. Returns null if **`target`** is the last data node of the list. The operation does not alter the data element reference in the affected node or in any other node in the linked list.

- **`void addFirstNode(Node<E> newNode)`** - inserts node **`newNode`** as a new data node at the beginning of the linked list. The new node becomes the first data node in the linked list. It is presumed that **`newNode`** is a valid node for the current implementation and that it is not an existing node in any list. If valid, the size of the list increases by 1. Notice that this implies all nodes in the original list (if any) are now displaced (automatically!) one position to the right in the new list... The operation does not alter the data element reference in the affected node or in any other node in the linked list.

- **`void addLastNode(Node<E> newNode)`** - inserts node **`newNode`** as a new data node at the end of the linked list. The new node becomes the last data node in the linked list. It is presumed that **`newNode`** is a valid node for the current implementation and that it is not an existing node in any list. If valid, the size of the list increases by 1. The operation does not alter the data element reference in the affected node or in any other node in the linked list.

- **`void addNodeAfter(Node<E> target, Node<E> newNode)`** - Inserts node **`newNode`** as a new data node that will be immediately after the data node being referenced by **`target`**. It is presumed that **`target`** corresponds to a data node in this list. It is also presumed that **`newNode`** is a valid node for the current implementation and that it is not an existing node in any list. If valid, the size of the list increases by 1. Notice that this implies nodes succeeding **`target`** in the original list (if any) are now displaced (automatically!) one position to the right in the new list... The operation does not alter the data element reference in the affected node or in any other node in the linked list.

- **`void addNodeBefore(Node<E> target, Node<E> newNode)`** - Inserts node **`newNode`** as a new data node that will be immediately before the data node being referenced by **`target`**. It is presumed that **`target`** corresponds to a data node in this list. It is also presumed that **`newNode`** is a valid node for the current implementation and that it is not an existing node in any list. If valid, the size of the list increases by 1. Notice that this implies that node **`target`**, as well as those succeeding it in the original list (if any) are now displaced (automatically!) one position to the right in the new list... The operation does not alter the data element reference in the affected node or in any other node in the linked list.

- **`void removeNode(Node<E> target)`** - Removes the data node referenced by **`target`** in the current instance of the linked list. It is presumed that **`target`** corresponds to a data node in this list. If valid, the size of the list decreases by 1. Notice that this implies all nodes in the original list that remain (if any) are now displaced (automatically or virtually!) one position to the left in the new list. The current content of the node that is removed is completely detached from the node (In the implementation, this would mean that all internal fields of the object corresponding to the node, including reference to the element, are set to null. For that, see the operation `clean()` on each implementation of `Node` ADT.).

- **`Iterable<Node<E>> nodes()`** - Returns an **`Iterable<Node<E>>`** object that allows iteration over the nodes in the linked list.

- **`Node<E> createNewNode()`** - creates a new instance of a particular type of node being used by the current list implementation; for now, either **`SNode`** or **`DNode).`** The node is created having all its internal fields initialized to **`null.`** Returns reference to the new node. The purpose of this operation is to allow the creation of nodes of the particular type that is needed by the type of linked list used in the implementation (which is hidden from the outside).

In the following sections, we discuss some implementation ideas, and then include some exercises.

Note: The above specification includes the minimum set of methods that are needed to fulfill the purpose of being able to use the linked list (as specified) as a data structure to implement more abstract collections. Other methods can be added to support simplicity and perhaps more efficiency; but the ones specified are enough.

4. Description of Structures to Implement

Think of the linked list as a structure in memory that can store data elements when it is used to implement some other more abstract type of collection. For example, we will be able to use the linked list to implement the list, stack, queue, set, etc. As you should have noticed, the above operations only work with nodes, and not with the data each node holds.

A linked list, if not empty, consists of a sequence of interconnected nodes. We shall consider different variants of two types of linked lists. These are:

- The **singly linked list** - Each node has an area for data, and another area for reference to the node (if any) that physically follows in the linked list. For the last node in the list, such reference to the next node is set to **`null`**. The nodes for this type are called *simple nodes*[2] (see class **`SNode`** below).
- The **doubly linked list** - Each node, in addition to its area for the data, contains references to two other nodes in the list: to the node that physically precedes (if any) and to the node that follows (if any). For the first physical node in the list, the reference to the preceding node is set to **`null`**. That's also the case for the reference to the following node in the case of the last physical node in the doubly linked list. The nodes for the doubly linked list are to be called: *double nodes* (see **`DNode`** below)

In order to specify a linked list as an ADT of nodes, we also need a simple specification for an ADT corresponding to a node. In particular, this abstract description of node hides all the details about links to other nodes. Every type of linked list implementation requires access to a particular implementation of such abstract node. The node ADT is a type of memory location where data is stored. The "links" (references to other nodes) are details of the class that implements the linked list, but not to other classes implementing other abstract collections that use the linked list. Such ADT for the node is called: **`Node`**, and is specified as follows:

```
// ADT for Node<E>
public interface Node<E> {
  void setElement(E e);      // sets element of the node to e
  E getElement();            // accesses the element in the node
}
```

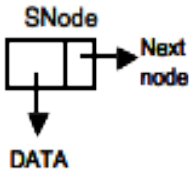
Each type of node in a linked list implementation must implement this interface.

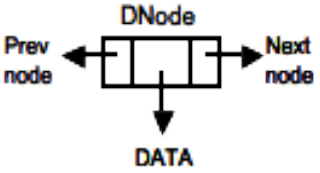
Classes Implementing Nodes

The linked lists can be implemented in many ways. The implementations in this project will either have nodes of type **`SNode<E>`** or nodes of type **`DNode<E>`** (both classes implement **`Node<E>`**).

We commonly draw each type of node as it is shown below:

[2] This is not necessarily a standard name.





The particular node data type (class **SNode** or class **DNode**) is implemented as an internal class inside the class that implements **LinkedList** in which that particular type of node is used. More explicitly, in each implementation of the single linked list **(SLList, SLFLList** and **SLDHList** classes) the class **SNode** is implemented as an inner class, as follows. (See example of partial implementation of **SLList** later in this document.)

```
private static class SNode<E> implements Node<E> {
   private E element;
   private SNode<E> next;
   public SNode() {
      element = null;
      next = null;
   }
   ... Other constructors + getters and setters ...
   public void clean() {   // to set fields of node to null
      element = null;
      next = null;
   }
}
```

As for the case of any implementation based on the doubly linked list, the class **DNode**, also an internal class, is as follows:

```
private static class DNode<E> implements Node<E> {
   private E element;
   private Node <E> prev, next;
   public DNode() {
      element = null;
      prev = next = null;
   }
   ... other constructors + getters and setters ...
   public void clean() {   // to set fields of node to null
      element = null;
      prev = null;
      next = null;
   }
}
```

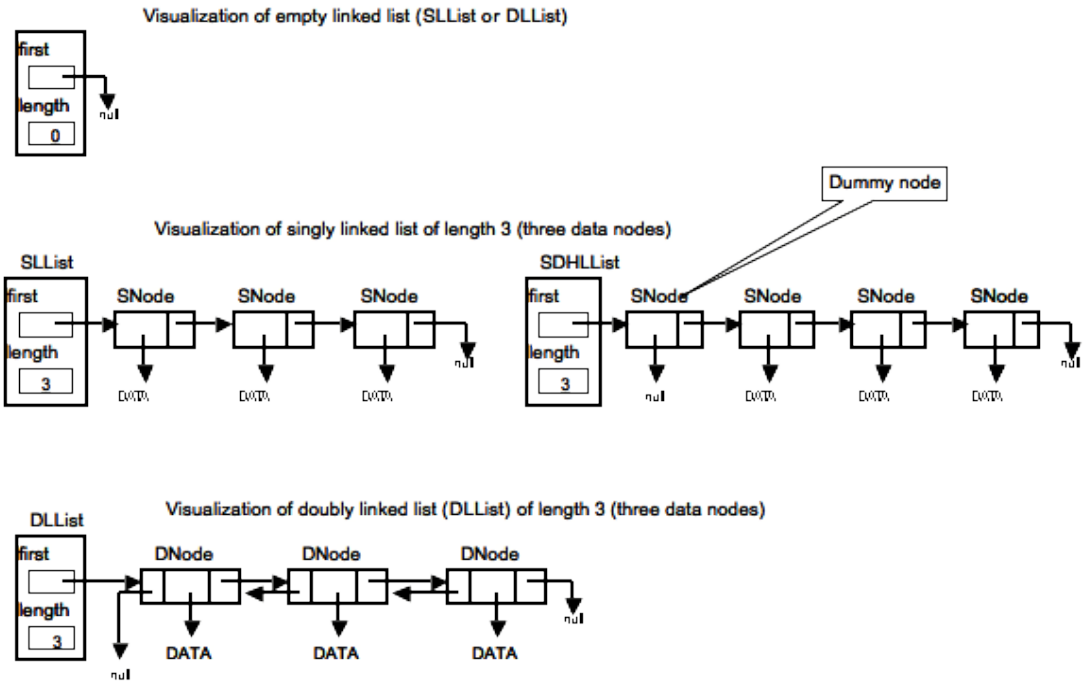
Notice that under this approach, the links to other nodes in the list are only accessible inside the corresponding class implementing the linked list; outside of it, a node only allows access to its element through the methods `setElement` and `getElement`.

5. Some Discussion on How to Implement `LinkedList`

Every class implementing the linked list must have at least the following:

- One instance variable that allows quick access to one of its nodes (a reference to a physical node in the list), and from which you can eventually access the other nodes that are part of an instance. For the implementations described, it is desirable that at least each has a reference to the first physical node in the structure; but, as we will see, some implementations have more than just a reference to that first node.
- One instance variable to hold its size or length (with which you can know how many data nodes the list has at all times without having to count them one to one) and manage it efficiently.

Note that in every case of an operation that works relative to a given node in the linked list I have always used the name **`target`** for the corresponding parameter in the particular method. Such a node is also assumed to be a data node in all those cases. You should also notice that in some implementations all nodes are data nodes. This is the case for every node in lists: **`SLList, SLFLList,`** and **`DLList.`** In the other cases, the list will have nodes to mark the beginning and/or the end of the list; and those marking nodes are not data nodes. They are also called sentinel nodes or dummy nodes. They are useful in simplifying some of the algorithms. The following figure gives a general idea of the memory map for the content of linked lists implemented in classes **`SLList, SDHList,`** and **`DLList`**.



In the case of the **`SDHList`**, you can see that the dummy node is a node that is only used as a marker. Such nodes are also called *sentinel nodes*. The figure for the **`SDHList`** implementation shows a list of size 3; it only has 3 data nodes and one dummy node as its header node. An empty instance of such linked list will only have that dummy node. The simplicity of having that dummy node in some algorithms lies upon the fact that "every data node has a predecessor node". We shall see more about this.

All methods described in the previous section, must be part of an interface as suggested below:

```
public interface LinkedList<E>  {
   int length();
   Node<E> getFirstNode()
          throws NoSuchElementException;
   … specification of other methods previously described …
}
```

Therefore, every class implementing a particular version of the linked list needs to implement the above interface. Each class will have at least two instance fields, as illustrated in the following partial implementation of **`SLList`**.

```
public class SLList<E> implements LinkedList<E> {
    private SNode<E> first;   // reference to first node in the list
    private int length;       // size of list (number of data nodes)
```

```
    // default constructor - an empty linked list
    public SLList() {
      first = null;
      length = 0;
    }

    public int length() {
      return this.length;      // the number of data nodes in this linked list
    }

    public Node<E> getFirstNode() {
      if (length == 0)      // linked list is empty (no data nodes in it)
          return null;

      // if list is not empty then return reference
      // to its first node….

      return first;
    }
    …

   /** Method to create a new node of the appropriate type
       being used by this implementation of the linked list.
      @return the address (reference) of the new node
    **/
    public Node<E> createNewNode() {
      return new SNode<E>();       // SNode is the type of node used here...
    }

    // class SNode – private inner class
    private static class SNode<T>
    implements Node<T>
    {
      private T element;
      private SNode<T> next;
      public SNode() {
        element = null;
        next = null;
      }
      public SNode(T e) {
        element = e;
        next = null;
      }
      public SNode(T e, SNode<T> nn) {
        element = e;
        next = nn;
      }
      public void clean() {   // to set fields of node to null
        element = null;
        next = null;
      }
      … getters and setters for both internal fields
    }  // end of class SNode

} // end of class SLList
```

Remember that we want to implement these linked structures so that they can be used as data structures to implement more abstract collections of objects (lists, etc). Also notice that some of the described operations can be implemented with different levels of efficiency in different types of linked lists. To decide which implementation is the best under given circumstances is part of the analysis work that you have to do when in need to use the linked list data structure.

The use of dummy nodes simplifies the algorithm required for some of the operations. For **SLDHList** each data node has another node (the dummy node or another data node) node that precedes it. In this case, none of the operations will be in need to alter the value of the reference instance field **first.** As an example, the operation to remove a node just needs to work with the reference **next** at the node that precedes the target node. Hence, the main task on that operation to delete is simplified to always search for the node that physically precedes the target node in the linked list. That node will always exist in **SLDHList**.

In the case of **DLDHTList**, it always happens that every data node in the list has one node that precedes it (which might be the dummy header) and one node that follows (which might be the dummy trailer). This simplifies the code for some of the algorithms to remove and to insert nodes. The cost is that implementations based on dummy nodes are required to use additional memory areas that are somehow wasted; however, the efficiency and simplicity gained in algorithms for some operations is worth considering. This is part of the trade-off that needs to be analyzed.

The code you have received includes a full implementation of class **SLList**. You need to study it and understand the algorithms used on each operation. Notice that the implementation is based on nodes of type **SNode**, which is a private inner class in class **SLList**. Such type of node implements the interface **Node**, as previously specified. Pay special attention to the different castings that are being used when handling nodes, and why they are needed on those instances. See that as per the specification of **LinkedList**, node parameters (whenever needed) are specified as of type **Node**. Depending on the particular implementation of the linked list, those nodes are either of type **SNode** or of type **DNode** (also notice that these are both of type **Node**). In every case in which the parameter name *target* is used, it is assumed that such node is of the appropriate type as per the particular list implementation that it is part of. Since that parameter is specified as of type **Node**, and the public specification of such type does not include operations to manage links of the node, then the casting to the appropriate type of node being used in the implementation is required. As an example, study the following operation for class **SLList**:

```
public void addNodeAfter(Node<E> target, Node<E> nuevo){
      // Pre: target is a node in the list
      // Pre: nuevo is not a node in the list
      ((SNode<E>) nuevo).setNext(((SNode<E>)
            target).getNext());
      ((SNode<E>) target).setNext((SNode<E>) nuevo);
      length++;
}
```

One alternative to reduce the number of castings is to do it as follows:

```
public void addNodeAfter(Node<E> target, Node<E> nuevo){
      // Pre: target is a node in the list
      // Pre: nuevo is not a node in the list
      SNode<E> snTarget = (SNode<E>) target;
      SNode<E> snNuevo = (SNode<E>) nuevo;
      snNuevo.setNext(snTarget.getNext());
      snTarget.setNext(snNuevo);
      length++;
}
```

Question: Why can't we just use, in the above code (without casting), expressions such as this:
nuevo.setNext(...**)**?

A common error is to have the following function header:

```
public void addNodeAfter(SNode<E> target, SNode<E> nuevo) …
```

The problem with this is that it does not comply with the **LinkedList** specification, which establishes that both parameters need to be of type **Node**. The compiler would alert (an error message) that some methods have not been implemented. Try it.

Let's quickly look at some of the methods when the list of nodes is a *doubly linked list*. This the case on classes: **DLList** and **DLDHTList**. Both classes need to internally implement the class **DNode<E>** as described elsewhere in this document. We will now see the implementation of some operations in both classes.

The instance attributes in each of these classes are illustrated in the next partial implementations:

```
public class DLList<E> implements LinkedList<E> {
      private DNode<E> header;
      private int length;
      public DLList() {          // creates empty linked list
         header = null;
         length = 0;
      }
      ...
}

public class DLDHTList<E> implements LinkedList<E> {
      private DNode<E> header, trailer; // dummies
      private int length;
      public DLDHTList() {       // creates empty linked list
         header = new Dnode<>();     // node with null fields
         trailer = new Dnode<>();    // node with null fields
         header.setNext(trailer);    // link the two nodes to each other
         trailer.setPrev(header);    // link the two nodes to each other
         length = 0;
      }
      ...
}
```

Need to keep in mind that in these cases each node has links to two other nodes. See that partial implementation of class **DNode** presented earlier. Let's see how we can implement operations to add the first node and to remove a node in both classes.

To add a first node in **DLList**, (new node that shall become the *first data node* of the linked list) we start by modifying the **next** field of the new node (which is the parameter of the operation) to reference to whatever **header** is referencing at the moment, and set its **prev** field to **null**. Then, if the list was not empty, we need to set the **prev** reference to what was the first node to point to the new node. After this is done, we set a reference header to the new node. Finally, increase the length. Here is the code:

```
      public void addFirstNode(Node<E> nuevo) {
         DNode<E> dNuevo = (DNode<E>) nuevo;
         dNuevo.setNext(header);
         dNuevo.setPrev(null);
         if (header != null)
```

```
            header.setPrev(dNuevo);
        header = dNuevo;
        length++;
    }
```

To add the first node in **DLDHTList**, we just need to insert the new node between the dummy header node and the other node that follows it in the list. Note that there is always a node that follows, even in the case of the empty list. Start by modifying the **next** field of the new node (which is the parameter of the operation) to reference whatever follows the dummy header node. Set its **prev** field to **header**. Then, **prev** field to whatever node currently follows the dummy header to point to the new node. After this is done, we set the **next** field of the dummy header node to point to the new node. Finally, increase the length. Here is the code:

```
public void addFirstNode(Node<E> nuevo) {
    DNode<E> dNuevo = (DNode<E>) nuevo;
    dNuevo.setNext(header.getNext());
    dNuevo.setPrev(header);
    header.getNext().setPrev(nuevo);
    header.setNext(dNuevo);
    length++;
}
```

And, as you can see, the empty list case is also covered! See why.

Let see now the operation to remove a given data node.

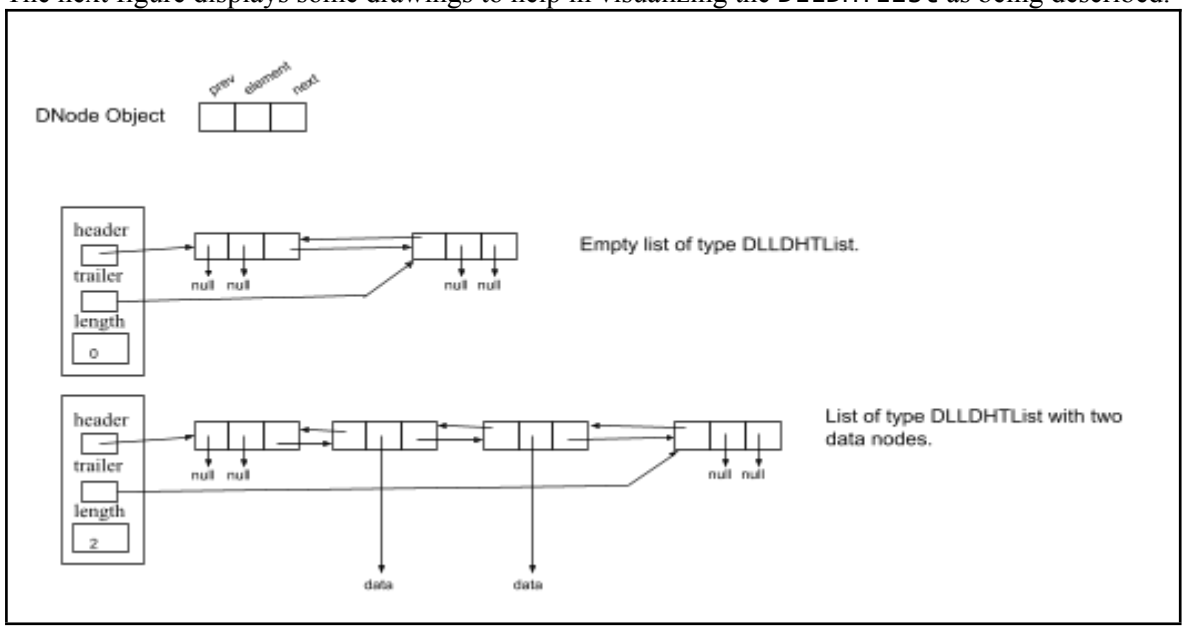
Operation to remove a node in **DLList**.

```
public void removeNode(Node<E> target) {
    DNode<E> dTarget = (DNode<E>) target;
    // check if the node to delete is the first node
    // and modify header if so...
    if (header == dTarget)
        header = dTarget.getNext();
    else
        // the node to delete has a previous node in the list...
        // adjust its next reference....
        dTarget.getPrev().setNext(dTarget.getNext());

    // check if the target has a next node in the list
    // if so, modify its prev reference...
    if (dTarget.getNext() != null)
        dTarget.getNext().setPrev(dTarget.getPrev());

    // now just remove the reference to other nodes
    // from node the node to delete
    dTarget.setNext(null);
    dTarget.setPrev(null);

    // now clean all fields in the removed node.
    dTarget.clean()

    length--;
}
```

Operation to remove a node in **`DLDHTList`**.

```
public void removeNode(Node<E> target) {
   //PRE: target is a data node (never dummy)
   DNode<E> dTarget = (DNode<E>) target;

   //Get surrounding nodes in the linked list....
   //They exist! since at least the dummies are
   // always there....
   DNode<E> prevNode = dTarget.getPrev();
   DNode<E> nextNode = dTarget.getNext();

   // Properly set the links of those surrounding nodes
   prevNode.setNext(nextNode);
   nextNode.setPrev(prevNode);

   // now clean all fields in the removed node.
   dTarget.clean()

   length--;
}
```

The next figure displays some drawings to help in visualizing the `DLLDHTList` as being described.



6. Applications

As has been previously established, the idea of the `LinkedList` (as we have seen it here) is to have it available as a data structure (in addition to the array) to implement more abstract collections. The linked list will provide the memory areas, as well as their proper management, to store and handle the locations of data in any particular collection that is implemented using it as its underlying data structure. The methods to manage the nodes in a linked list are used to properly implement the different operations of that abstract collection.

Note that the linked list as specified does not handle data stored in its nodes. Its different mutating operations only handle nodes. In that sense, the linked list is seen as a collection in which its data elements are its nodes. On the other hand, in a more abstract collection, one wants to handle its data elements, which, if implemented using a linked list, those data elements are stored in the different nodes of that underlying data

structure. For instance, if a particular method for such collection needs to access a particular element in a particular instance, then the algorithm for such method needs to get, from the underlying linked list, the particular node holding the desired element. Once that node is accessed, the data value that it stores is made accessible by applying the **`getElement()`** operation to that particular node. We shall make this more clear as we study some examples later on.

Whenever we are implementing a collection ADT that is based on the linked list, we should be capable of using anyone of its different versions. However, we should be aware that different versions as the underlying data structure may represent different levels of efficiency, or simplicity, on the algorithms for the operations that the particular collection ADT must have.

For the moment, we shall only consider collections ADTs corresponding to different types of lists. For example the so-called "List". In the discussion that follows, we will refer to that list as “IndexList”. You can find the complete interface in the shared code for this project. It specifies operations such as:

```
public interface IndexList<E> {
    public int size();
    public boolean isEmpty();
    public void add(E e);
    public void add(int i, E e)
           throws IndexOutOfBoundsException;
    public E get(int i)
           throws IndexOutOfBoundsException;
    public E remove(int i)
           throws IndexOutOfBoundsException;
    public E set(int i, E e)
           throws IndexOutOfBoundsException;
}
```

For more details on what each operation does (their specification or API), see the textbook. This type of list is one in which every element in it occupies a position that is identified by an index. If not empty, the first item in the list has index 0, the second (if any) has index 1, the third (if any) has index 2, and so on. Note that you have already studied a Java class that has such behavior: the **`ArrayList`**.

We will be able to implement this type of list in such a way that any particular instance can dynamically use any one of the linked lists as its underlying data structure. The implementation of the linked list to be used by a particular instance of **`IndexList`** is decided at the moment of its creation. See the second constructor in the following partial implementation of the class. Its different operations will be based on the abstract specification of **`LinkedList.`** The following is a partial implementation following such an approach. You need to study it carefully.

```
public class LLIndexList<E> implements IndexList<E> {
    private LinkedList<E> internalLL;
    // this is the only instance variable (or instance field).
    // It is used to reference the underlying data structure of
    // the current instance of LLIndexList.

    // No instance field to hold the size is needed since
    // the size in this case is precisely the length() of the
    // underlying linked list. Contrary to the array, we can
    // always guarantee (more efficiently) that the number of
    // slots for data elements (number of data nodes) dynamically
    // increases or decreases based on the size of the list.
```

```
/**
      Creates an empty instance of IndexList. The list
      object created uses, by default, a singly linked
      list is used as the underline data structure of
      instances of LLIndexList.
**/
public LLIndexList() {
      internalLL = new SLLIst<E>();
}

/**
      Creates an instance of IndexList. The list
      instance created uses, as its underlying data structure,
      the singly linked list received as parameter.
      @param theList reference to the LinkedList object over
            which the current instance of list is created….
**/
public LLIndexList(LinkedList<E> theList) {
      internalLL = theList;
}

/**
      Determines the size of the list.
      @return size of the list – number of elements.
**/
public int size() {
      return internalLL.length();
}

/**
      Determines if the list is empty.
      @return true if empty, false if not.
**/
public boolean isEmpty() {
      return this.size() == 0;
}

/**   PRIVATE METHOD to access the data node at the
      position given in the internal linked list.
      If the list is not empty, the first data node
      has position 0, the following data node (if any)
      has position 1, and so on. This should be
      interpreted as first position, second position,
      and so on.
      @param posIndex the index of the position being
             accessed.
             PRE: posIndex is assumed to be a valid index.
                  0 <= posIndex < internalLL.length().
      @return reference to the data node in the given
             position of the internal linked list.
**/
private Node<E> getDataNodeAtPosition(int posIndex) {
      Node<E> target = internalLL.getFirstNode();
      for (int p=1; p<= posIndex; p++)
          target = internalLL.getNodeAfter(target);
```

```
            return target;
      }

      /**
            Adds a new element to the list.
            @param index the index of the position where the
                  new element is to be inserted.
            @param e the new element to insert.
            @throws IdexOutOfBoundsException if the index
                  i does not corresponds to the index
                  of a valid position to insert…
      **/
      public void add(int index, E e)
            throws IndexOutOfBoundsException
      {
            if (index < 0  ||  index >= internalLL.length())
                throw new IndexOutOfBoundsException("add: "
                        + "index=" + index + " is out of bounds.");

            Node<E> nuevoNodo = internalLL.createNewNode();
            nuevoNodo.setElement(e);
            if (index==0)
                internalLL.addFirstNode(nuevoNodo);
            else {
                Node<E> nodoPrevio =
                        getDataNodeAtPosition(index);
                internalLL.addNodeAfter(nodoPrevio,
                        nuevoNodo);
            }
      }
      … other methods as in interface IndexList
}
```

You can think of variants of the above algorithms to achieve the same result. Note also that the second constructor creates an object of type **`LLIndexList<E>`** based on an implementation of the linked list which is specified at the moment when the particular **`IndexList`** instance is created. For example, to create an object **`LLIndexList<E>`** based on the implementation of the doubly linked list we can use the following expression:

```
new LLIndexList<E>(new DLList<E>())
```

Think of how to use this to properly test.

7. Exercises for this project

Exercise 1. Upload the .zip file as an Eclipse (or IntelliJ) project. There, you will find some classes. Execute class **`IndexListTester`**. We shall be making reference to the output it produces to do testing of the code you write as part of other exercises.

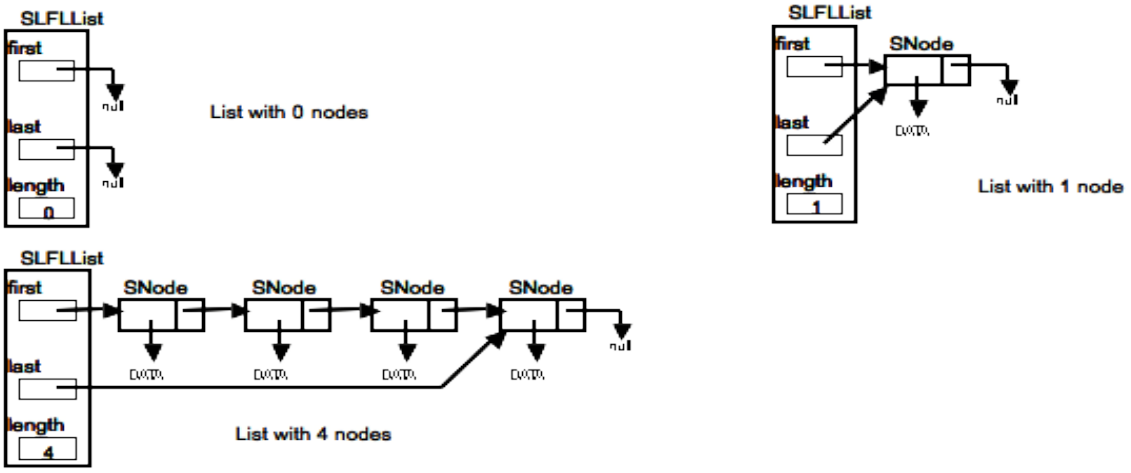
Exercise 2. Study the different classes and interfaces that are part of the uploaded project. In particular, the class **`LLIndexList`** is an implementation of the **`IndexList`** ADT as previously specified. There is also a tester class that tests the **`LLIndexList`** independently of what type of linked list is used on a particular

instance. See how its constructor is used in the tester class included; which, in this case creates an instance of **`IndexList`** based on **`SLList`** as the underlying data structure.

Notice that under this approach, the underlying data structure upon which an instance of the **`LLIndexList`** will run is established at the moment of its creation. Under this approach, we can create an instance of **`LLIndexList`** based on any type of linked list as long it is an implementation of **`LinkedList`**.

Show your instructor how you can create instances based on each of the different types of linked lists described in this document.

Exercise 3. Fully implement class **`SLFLList`** as has been specified. Such a linked list has reference to its first data node, as well as to its last data node. If empty, both references are **null**. If only one node, both make reference to that node. The following figure illustrates three cases of such a list: one having length 4, another having length 1 and another having length 0 (empty list).



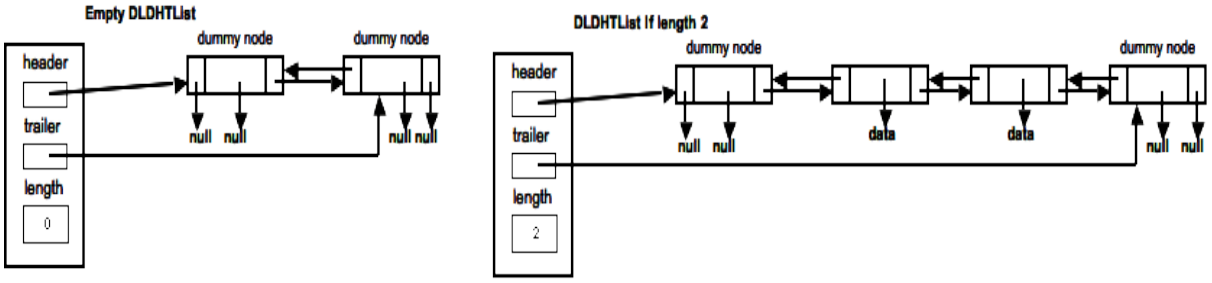
The implementation of each operation will be very similar to the corresponding one in **`SLList`**; however, in some of them you need to consider the possibility of having to modify either the field **`first`** or the field **`last`**.

Once your implementation is complete, modify the tester so as to create an instance of type **`LLIndexList`** based upon **`SLFLList`** (just a minor change). Results should be the same as those using **`SLList`**.

You need to consider the following when writing the algorithms for the different operations. On every operation that alters the structure of the linked list you need to think on different cases to consider. These are the operations to add a new node to, or to remove an existing node from, the linked list. In those cases, you need to analyze if there are cases that need to be treated differently from the norm. For example, if the node to add will be the new first node or if the node to remove is the actual first node, the reason being that in those cases, the internal field that references the first node in the list would need to be modified. If the operation does not change the current first node in the list, then the algorithm would need to modify the **`next`** field on the node that precedes. And in the case of the type of list being considered in this exercise, the algorithms need to also consider whether the internal reference to the last node changes.

Look at the implementation given for **`SLList`** and for some of the operations in the case of doubly linked lists. Once your implementation is complete, test it using the tester provided. You can also develop your own tests one by one as new operations are implemented. In this case, you need to figure out what is the order in which operations need to be implemented in order to be able to test immediately.

Exercise 4. Implement class **`DLDHTList`**. The following figure shows two cases of this type of linked list. The first is the empty list (no data node), the second is a list of length 3 (3 data nodes).



Some of the methods are already implemented. Study those to get an idea of the others needed. Also, notice the method **`destroy()`** that has been implemented and try to figure out its usefulness.

Exercise 5. Suppose that we want to make `LinkedList` to be `Cloneable` too. To do that, we can start by adding the '`extends Cloneable`' phrase to the header line in the `LinkedList` interface. Once you do that, several errors will pop up in the Java code you have been using. It is possible in this case to satisfy the requirement imposed by this new addition by introducing an intermediate abstract class, say `AbstractLinkedList`, which implements `LinkedList` and implements there the clone method. Then, all final implementations of LinkedList just need to have the phrase '`extends AbstractLinkedList`' added to the header line. Is this possible? If not, why not? And, if that is the case, what do you suggest then? Because, somehow it should be possible to make this type of list to be `Cloneable`.

Later on, on your time for studying for this course, implement what is needed to make each implementation cloneable.

8. Deadline and Submission Guidelines

There will be four specific deadlines for this project as described next. Not complying with these precise instructions will cause, in the best case, reduction of point from your final score in the project. In some instances, as explained, it may cause your project to be considered as not submitted. Note that these penalties are easy to avoid, just follow the instructions.

To submit your final version of project 2.	Wednesday, April 28, 2021	Electronic submission at online.upr.edu

Submissions are due at 11:59:59 pm on the announced date. NO LATE SUBMISSIONS SHALL BE ACCEPTED.

On the final submission, you must submit the following:

1. Zip file containing a directory, inside which your project is located when extracted and saved. That zip file shall be named as: **`P2_4020_`*`nnnnnnnnn`*`_202.zip`**. (Where *nnnnnnnnn* are the 9 digits of your student id number)
   - It should contain your project directory.
   - The name of that extracted folder shall be: **`P2_`*`nnnnnnnnn`***. (... the number as before)

- Inside this last directory is where your project folder is located. The name of your project must be **P2_40204035_202**.

2. The extracted directory (`P2_nnnnnnnnn`) will also contain the pdf README file with important information about your project, including clear instructions describing how your program can be executed from the command prompt assuming that such execution is initiated from the terminal window while located at the directory `P2_nnnnnnnnn`.

At the beginning of each file (document or Java file) you should include your ***name***, ***student id***, and ***section number***. You should verify that those instructions work. Also, your program should be able to read the input files (as specified) as long as the directory specified for those input files is a subdirectory of the same directory from where the program is executed (from where the command to execute is typed from the terminal window).

Your code should include at least the proper documentation for its classes and methods, which must follow the standards required for the Javadoc tool. You don't need to submit html files that are produced by the Javadoc tool; just make sure that your comments follow that standard.

Once your zip file is ready, please, submit it in the course portal using the appropriate open project submission homework. ANY SUBMISSION IN A METHOD DIFFERENT FROM THIS, WILL MAKE YOUR PROJECT TO BE CONSIDERED AS NOT SUBMITTED. **Failing to comply with instructions for submission may cause deduction of points in your final grade or for the project to be considered as not submitted. Projects submitted after the specified deadline (day and time) will be considered as not submitted and therefore shall not be graded.**

9. Shared Project Files

A public Git repository where you can find some useful classes for this project is available. You can click the following to access that repo: Initial Partial Implementations for P2.

You can freely use whatever you find there.

10. General Idea Of How Are We Going To Grade Your Project

Grading of this project takes into consideration all the following: (this may not be complete)

- **a.** Instructions are followed - Includes what to submit, deadlines, how to submit,, ...
- **b.** If README file is well-written and details there properly describe how to execute each of the two main classes as per the specifications given, ...
- **c.** Coding style - Verify readability of code, documentation of essential parts, if files have names of authors, etc.
- **d.** If the expected code for all exercises is provided.
- **e.** If execution can be done from the command prompt as specified.
- **f.** If execution of your submitted code works correctly, assuming correct input. For this, we will use several input cases for which the final result is known. Your program will be tested based on those and partial scores will be added as tests are passed. A test is passed if output is as expected. Here, we also verify if the different strategies were implemented as per the specifications given.
- **g.** We may experiment with your solutions to see if each generates experimental data that matches expected results.
- **h.** Other criteria may apply.

On the examination of your code, we also look for possible unauthorized sharing of solutions. This is a project in which collaboration with others is not allowed. Of course, you may discuss ideas with others to better understand the specifications given, but the final implementations should be individual. On the other hand, the preliminary code that has been shared with you and given in lectures, can be used as it is, or adapted as needed. Also, algorithms in many instances may be the same on different implementations, but we shall use our experience to determine if, in addition to those, there is more in the programs of different students that make them unreasonably similar to one another.

In those cases in which (we hope for none of those) we can effectively confirm that two programs are the same (for example, if the only variant is the name of variables and functions; or, as it has happened in the past, that the files show the name of a another student), then those will be subject to more scrutiny and the best that would happen is that all of those involved will get a score of 0. Other disciplinary actions as per UPR's rules may be initiated in such cases. This is easy to avoid, just do your own coding.

To avoid unauthorized use of your code, do not use public repositories. Any repository used by you must be private. Those who do not comply with this will get a score of 0.

Remember, these projects are aimed for you to learn some fundamental skills. Do not try to cut corners, and you will be proud and grateful of yourself in the future when these skills come in need in other courses or in the practice of your profession. After all, that’s the goal.

FINAL NOTE: Be aware that the tester provided in the partial implementation does not test all the methods for the **`LinkedList`**, nor are they exhaustive. You should be able to develop more complete testers to make sure that everything else in your implementations works according to the specifications. You may need these or similar implementations later in the course in order to implement more abstract ADTs and the last thing that you want is to have to come back to debug these implementations. That extra testing is left as exercise for you to do on your own, but, of course, you can always consult us.