University of Puerto Rico – Mayagüez
Department of Computer Science and Engineering

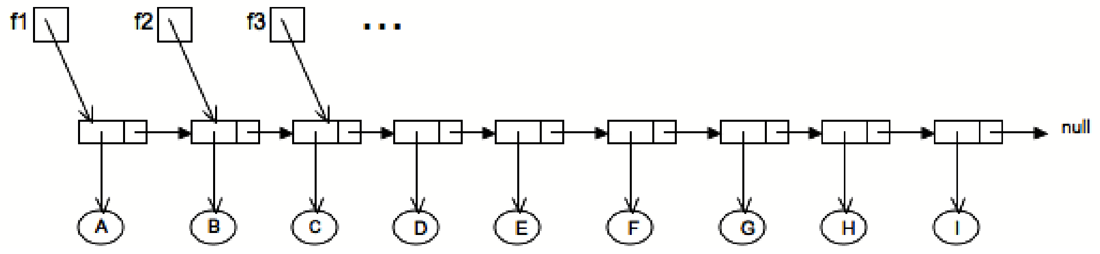
CIIC 4020 / ICOM 4035 – Data Structures
Prof. Juan O. López Gerena
Spring 2019-2020
Laboratory #4 – Recursive Linked Lists

In this activity, we will work with a recursive implementation of the **`List`** ADT based on the singly linked list. We shall discuss some issues that are relevant in deriving a recursive solution for each operation to be implemented, as specified in the **`List`** interface. Be aware that this way of thinking is useful for many other cases regarding operations on more complex data structures that shall be studied in the course. The class to implement is **`RecursiveLinkedList`**.

Recursive Thinking with Lists

One needs to "think recursively" when designing recursive algorithms. One strategy in this case is to leverage on the recursive nature of the list. When working with a particular data structure there might be different recursive ways to view (or define) that particular type of structure. Different recursive algorithms may be designed to traverse the structure, each based on those different recursive views that the structure may allow.

Recall that we use indexes 0, 1, 2, … to refer to the first node, to the second node, and so on in a linked list. The basic principle satisfied by each one of the recursive algorithms to implement is the following: for `i` > 0, the node corresponding to index `i` in a non-empty linked list whose first node is referenced by `f` is the node corresponding to index `i`-1 in the linked list whose first node is being referenced by `f.getNext()`. The node corresponding to index 0 is the first node in the list referenced by `f` (i.e. the node being referenced by `f` itself). Consider the following figure:



The list whose first node is being referenced by `f1` is the list `{A,B,C,D,E,F,G,H,I}`. That list's node with index 3 contains `D`, but that same node has index 2 in the list whose first node is being referenced by `f2` (or `f1.getNext()`). At the same time, that particular node is the node with index 1 in the list whose first node is being referenced by `f3`.

As in many other occasions, when implementing a particular operation for a given ADT using some recursive algorithm, the method complying with the operation's interface needs to invoke some auxiliary (usually private) method. That auxiliary method is the one where the recursive algorithm is finally implemented. *Why have two different methods for one operation?* Because usually the recursive algorithm needs a different set of parameters from those specified in the public interface for the ADT's method. The method complying with the public interface does some initial work to prepare for the recursive one, then appropriately invokes the auxiliary one, and afterwards it might perform some final execution after the auxiliary method has finished executing. Let's consider one of the methods for the particular problem being considered, the **`get`** method. Such method is implemented as follows (where `head` references the first *data node* of the list):

```
public E get(int index) throws IndexOutOfBoundsException {
      if (index < 0 || index >= size())
          throw new IndexOutOfBoundsException(
            "RecursiveLinkedList.get: invalid index = " + index);

      // index is valid
      return recGet(head, index);
}
```

The auxiliary method is **`recGet`**, which presumes that the node corresponding to position `index` exists in the list (otherwise the exception `IndexOutOfBoundsException` would have been thrown as per the above code). The method needs two parameters: one to reference the first node of the list (or sub-list), and another one for the index of the position being targeted. It works recursively as follows:

$$\text{recGet(f, i)} = \begin{cases} \text{f.getElement()} & \text{if } i = 0 \\ \text{recGet(f.getNext(), i-1)} & \text{if } i > 0 \end{cases}$$

Observe that this recursive definition for `recGet(f, i)` comes directly from the recursive view of the linked list, as it was discussed above. This operation is slightly easier than the others since it has no side effect on the list (i.e. it does not alter the list); the other operations do alter the list. But, notice that the goal is to find the sub-list whose first node is "relevant for the operation". For instance, for the previous operation, the relevant node is the node whose corresponding index is the targeted one. The same will be true for the **`set`** operation, but that method has side effects on the list. For the case of the **`remove`** operation, the relevant node is the one to be deleted. For the **`add`** operation (the one with two parameters), the relevant node is the one that currently occupies the position corresponding to the given index in the list. In this particular case, that node will be "virtually displaced" inside the list, increasing its "virtual index" value by one; however, you should notice that no actual movement of data is necessary.

A partial implementation for class **`RecursiveLinkedList`** follows:

```
public class RecursiveLinkedList<E> implements List<E> {

      private Node<E> head; // References first data node
      private int currentSize;

      … Node class implementation here …

      public RecursiveLinkedList() {
            head = null;
            currentSize = 0;
      }

      … Implementation of other methods here …

      public E get(int index) throws IndexOutOfBoundsException {
            if (index < 0 || index >= size())
               throw new IndexOutOfBoundsException(
               "RecursiveLinkedList.get: invalid index = " + index);

            // index is valid
            return recGet(head, index);
      }

      public void add(E e) {
            add(size(), e); // Add at the end of the list
      }

      public void add(int index, E e) throws IndexOutOfBoundsException {
            if (index < 0 || index > size())
               throw new IndexOutOfBoundsException(
               "RecursiveLinkedList.add: invalid index = " + index);

            // index is valid for the add operation
            head = recAdd(head, index, e);
            currentSize++;
      }

      public boolean remove(int index) {
            if (index < 0 || index >= size())
                  return false;

            // index is valid for remove operation

            …

            currentSize--;
            return true;
      }

      public E set(int index, E e) throws IndexOutOfBoundsException {
            if (index < 0 || index >= size())
               throw new IndexOutOfBoundsException(
               "RecursiveLinkedList.set: invalid index = " + index);

            // index is valid for set operation
            return recSet(head, index, e);
      }

      /*******************************/
      /* Auxiliary recursive methods */
```

```
            /******************************/

            private static <E> E recGet(Node<E> f, int i) {
                  if (i == 0)
                      return f.getElement();
                  else
                      return recGet(f.getNext(), i-1);
            }

            private static <E> Node<E> recRemove(Node<E> f, int i) {
                  if (i == 0) {
                      Node<E> ntd = f;
                      f = f.getNext();
                      ntd.clean();
                  }
                  else
                      f.setNext(recRemove(f.getNext(), i-1));

                  return f;
            }

            // TODO… see the code you have received…
            private static <E> Node<E> recAdd(Node<E> f, int i, E e)
            { … }

            private static <E> E recSet(Node<E> f, int index, E e)
            { … }

            …
      }
```

The following is the verbal description of the **`regGet`** and **`recAdd`** methods (contained within the Javadoc comments).

`… <E> Node<E> recGet(Node<E> f, int i)` – Returns the value in the node corresponding to the index value **`i`** in the linked list whose first node is being referenced by **`f`**. On any such list the first node is the one associated to index **`0`**, the second node is the one associated with index **`1`**, and so on. It presumes that the list whose first node is **`f`** has at least **`i+1`** nodes.

`… <E> Node<E> recAdd(Node<E> f, int i, E e)` – Inserts a new node in the linked list whose first node is being referenced by **`f`** so that the new node contains the data element **`e`** and it ends up occupying the position with index value **`i`**. Finally, it returns the reference to the first node of the list that results after the insertion is completed. It presumes that the list whose first node is **`f`** has at least **`i`** nodes.

Carefully study the implementation for the **`add`** operation. Try to connect this implementation with the above verbal description of **`recAdd`**. Notice that the value returned by **`recAdd`** is assigned to the private instance field *head*. Think about this: when is the new value assigned to *head* different from its current value?

```
      public void add(int index, E e) throws IndexOutOfBoundsException {
            if (index < 0 || index > size())
```

```
                throw new IndexOutOfBoundsException(
                    "RecursiveLinkedList.add: invalid index = " + index);

            // index is valid for the add operation
            head = recAdd(head, index, e);
            currentSize++;
    }
```

Exercises on Recursion and Linked Lists

Exercise 1

Write the Javadoc comments, including a formal verbal description (similar to what is done with **`recGet`** and **`recAdd`**), for each of the following two methods:

1. **`... <E> Node<E> recRemove(Node<E> node, int i)`**
2. **`... <E> Node<E> recSet(Node<E> node, int i, E e)`**

Note: Remove the @SuppressWarnings annotation in the ***`recRemove`*** *method.*

Exercise 2

Working Toward a Full Implementation of the `RecursiveLinkedList` Class

You have received a partial implementation of the **`RecursiveLinkedList`** class, as well as a tester class that can be used to run different tests over an implementation of **`List`**. For **`RecursiveLinkedList`**, you are required to implement all of the missing methods, one by one, and execute the test class after each new implementation. You should use the tester as it is, only removing *specific* commented lines to enable testing of new methods as they are implemented. Moreover, you cannot alter any other part of the **`RecursiveLinkedList`** implementation, but only work with the recursive methods that have not yet been implemented and with the **`remove`** method. For this method, the auxiliary recursive method has already been implemented, but you still need to implement the part that calls it. For each one of the methods you need to implement, only a skeleton, or a partial implementation, is provided.

As in any development process, you must first determine an appropriate order to implement your methods so that tests can be run after each implementation. This way, you will be in control of the process, since at each stage you'll know better what possible errors to look for. If from one state to the next, new errors surface, then it will most probably be due to the new modifications. Hence, any possible error will more likely be discovered and corrected. Then the strategy is simple: write code and test, write code and test, … until your product is complete.

You should study the implementation of method **`recGet`** to get a good idea as to how to implement the ones that are missing, but you need to have a clear understanding of the description given for each of the "auxiliary" recursive methods.

Let's determine an appropriate order to implement the different methods that are still missing so that you can effectively test each method immediately after implementing it. The tester class creates an object of type **`List`** (in this case, of type **`RecursiveLinkedList<Integer>`**) then keeps applying different operations and showing the content of the list after the operation is completed. Notice that the first method to implement and test should *not* be **`remove`**? (Why?) You may need to implement more than one method before doing any tests. In this particular case, since the **`get`** method is already implemented, we should begin with the **`add`** method, then we can implement either **`remove`** or **`set`** in any order. Also, notice that since the class implements the **`List`** interface, we need to provide some dummy implementation for each of the other methods that are yet to be implemented.

Implement the missing methods in the determined order. After each implementation, run your tester class. Do not work on the next method until you are convinced that your previous implementations are correct. Remember that it might happen that, after testing a new method, errors on the implementation of previously implemented methods may arise. In that case, you'll need to go back to those incorrect implementations and make the appropriate changes. Be aware of the possibility that apparent errors being detected on other already implemented methods may not be real errors, but incorrect behavior due to errors in the most recently implemented method.

Implementing the `recAdd` method

Work on the implementation of the **`recAdd`** method in accordance to the verbal description given in the discussion part. A general outline for this operation is as follows:

```
private static <E> Node<E> recAdd(Node<E> f, int i, E e) {
       // adds a new node (containing e) as the ith node in
       // the list whose first node is f…

       if (i==0) {
          … add the new node (properly initialized) as the first node in the current list
          … whose current first node is being referenced by f
          … Afterwards return reference to the new node, because it is now the first node…
       }
       else {
          … recursively invoke the method to the sub-list whose first node is the node
            right after node f  (since i is valid, such node exists) – remember
            that the index to look for in that sub-list is i-1  …
          … set the reference  to the new list (whatever the recursive call returns)
            as the new next node to node f…
          … then return the reference to the first node of the current sublist – since the
            new node is added somewhere after f (which is the first node of the
              current list), then the first node of the new list is still f …
       }
}
```

Implementing the `remove` method

The auxiliary recursive method is already included, but not the part inside the public method, **`remove`**, that does the initial work, calls the recursive auxiliary method, and returns the appropriate value. As you can see, the node to remove is the first node of the sub-list that satisfies the condition for the *base case* in the recursive method. In that case, that first node (corresponding to index 0 in the current list) is disconnected from the list and its data. Since the method returns the reference to the first node of the resulting list, then the reference to the second node (corresponding to index 1 in the current list) is returned. Whenever the base condition is not met, then the new call to **recRemove** is initiated with the list whose first node is the second node in the current list (the one whose first node is f). This may sound repetitive, but that's recursion…

Implementing the `recSet` method

The implementation of **`recSet`** should be somewhat similar to **`recGet`**. Think recursively.

Exercise 3

Let's now work on a recursive algorithm to sort our linked list. To the **`List`** interface in the partial project received, add the specification of the following method to *sort elements in the list* based on a comparator object that is received as a parameter. The method is specified as follows:

```
/**
 * Sorts the elements in non-decreasing order based on the
 * comparator object received.
 *
 * @param cmp   Comparator object that establishes the relation order
 *              upon which the ordering will be based.
 */
 public void sort(Comparator<E> cmp);
```

Remember that a comparator object is one that implements the Comparator<E> interface:

```
public interface Comparator<E> {
  int compare(E first, E second);
}
```

The objective of the comparator is to define how comparisons are performed for a *particular* order relation, so every implementation depends on particular object data types. For example, if we need an order relation for students, which is based on GPA, we can have something as follows:

```
public class StudentGPAComparator implements Comparator<Student> {
      public int compare(Student s1, Student s2) {
            return s1.getGPA() – s2.getGPA();
      }
}
```

Recall that a comparator's **`compare`** method returns a negative integer, zero, or a positive integer depending on whether the first argument is less than, equal to, or greater than the second (respectively). Our sample comparator establishes an order relation on students, which is based on their GPA. With this type of comparator, to sort a **`RecursiveLinkedList<Student>`** object being referenced by **`w`**, in non-decreasing order by GPA's, we just need to write the following instruction:

```
w.sort(new StudentGPAComparator())
```

If the sort is correctly implemented, then the following must be satisfied afterwards:

```
sc.compare(w.get(i), w.get(i+1)) <= 0,  for  i=0, …, w.size()−2
```

where **`sc`** is an object of type **`StudentGPAComparator`**.

It is important that you understand the use of the comparator object as it is shown above, since we shall be using objects of this type throughout the course. This technique is used for many types of operations, not only for comparisons. For example, we can have a method to apply some generic operation to all the elements in a particular type of collection. By using the technique of sending an operator object each time the method is invoked, we can apply different operations without having to rewrite the whole code. Hence, code reusability can be achieved by using such techniques.

Recursive Thinking to Sort a SLL

We can sort the elements in a linked list by using the idea of the recursive insertion sort method discussed in class (in that case, for sorting an array), adapted to sort a list based on the following recursive idea. To sort a linked list whose first node is referenced by **`head`**, we can do the following:

1. If empty, or only one node, then nothing needs to be done -- the list is sorted
2. Otherwise,
   a. Recursively sort the sub-list whose first node is **`head.getNext()`**.
   b. Insert that first node (referenced by **`head`**) into the right place in that sorted sub-list.

In more concrete terms, for the particular implementation being considered, we can proceed as follows:

```
/**
 * Sorts the elements in non-decreasing order based on the
 * comparator object received.
 *
 * @param cmp   Comparator object that establishes the relation order
 *              upon which the ordering will be based.
 */
public void sort(Comparator<E> cmp) {
   if (size() <= 1) // Empty or only one node?
      return;  // List is already sorted
   else
      head = recInsertionSort(head, cmp);
}

/**
 * Recursively sorts the linked list whose first node is referenced by
 * variable “first” based on the given comparator object.
 * It finally returns the reference to first node of the sorted list.
 *
 * @param first   Reference to the first node of the list to be sorted
 * @param cmp     Comparator object that establishes the relation order
 *                upon which the ordering will be based.
 * @return        Reference to the first node of the sorted list
 **/
private static <E> Node<E> recInsertionSort(Node<E> first,
                                            Comparator<E> cmp) {
      if (first.getNext() == null)
            return first;  // Only 1 node, so the list is already sorted
      else {  // The list has more than one node
            /* Start by recursively sorting the sub-list starting at the
```

```
                  current list's second node (the node after first). */
               Node<E> first2 = recInsertionSort(first.getNext(), cmp);

               /* Now the sub-list must be sorted.  Just need to insert the
                  first node of the current list into that sorted sub-list.
                  Then, return the first node of the final sorted list. */
               return recInsertByOrder(first, first2, cmp);
         }
   }

   /**
    * Inserts a new node into a SLL whose first node is given and which is
    * assumed ordered in non-decreasing order based on comparator given.
    *
    * @param nti    Node to insert into the linked list given
    * @param first  First node of sorted list where insertion takes place
    * @param cmp    Comparator upon which the sorting is based
    * @return       Reference to first node of the resulting sorted list
    */
   private static <E> Node<E> recInsertByOrder(Node<E> nti,
                                 Node<E> first, Comparator<E> cmp)
   { … }
```

To implement this method, think *recursively* following the ideas that were discussed previously. The base case is when the list given (whose first node is referenced by parameter **`first`**) is empty (i.e., the value of **`first`** is **`null`**). In this case, the new list will be the list containing only one node: the one being referenced by **`nti`**. Moreover, that node will be its first node (since it is the only one in the list). Keep in mind that **`nti`** might have had a different position in the list, so it might still be pointing to another node. We need to fix this by setting it's next to **`null`**, otherwise we could inadvertently create a circular linked list. Hence, the base case is:

```
   if (first == null) {
         nti.setNext(null);
         return nti;
   }
```

Now, if that is not the case (**`first != null`**), we need to compare the value in node **`nti`** with the value in node **`first`** (our comparator is for values, not for nodes). If the value in **`nti`** is less than or equal to the value in **`first`** (according to the comparator), then node **`nti`** shall be the first node of the list that results. We just need to link node **`first`** as the next node of **`nti`**. In this case, the first node of the resulting list is **`nti`**.

If the value in **`nti`** is greater than the value in **`first`** (according to the comparator), then node **`nti`** needs to be inserted (by properly applying the recursive algorithm again) somewhere in the list whose first node is the one that follows node **`first`**. That is accomplished by just invoking:

```
   recInsertByOrder(nti, first.getNext(), cmp)
```

Here, the reference that is returned by the execution of the previous statement needs to be assigned as the node that follows first in the resulting list. Hence, you need to enclose the previous call as the argument value for: **`first.setNext(…)`**. Then, the method should return **`first`**. Think about it…