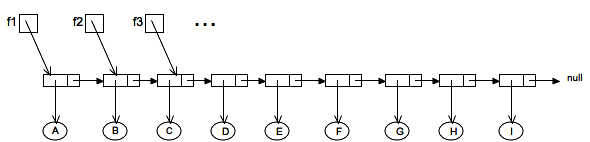
**ICOM 4035 / CIIC 3011 : Lab Exercises on Recursion and Linked Lists**

In this activity, we will work with a recursive implementation of the **IndexList** ADT based on singly linked list. The implementation will be direct: not depending on another object of type **LinkedList** as was used in the previous lab session. We shall discuss some issues that are relevant in deriving a recursive solution for each operation to be implemented, as specified in the **IndexList** 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 **SLRIndexList** .

**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, or data space, 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 (the node being referenced by f itself). Consider the following figure.



The list being referenced by f1 (the list whose first node is being referenced by f1) is the list (A, B, C, D, E, F, G, H, I). Its node for index 3 contains D. But this particular node is the node for 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? This is so 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 perform some final execution after its completion. Let’s consider one of the methods for the particular problem being considered: the *get* method. Such method is implemented as follows: (**first** is the instance field pointing to the first node in the list)

**public E get(int index) throws IndexOutOfBoundsException**

**{**

**if (index < 0 || index >= size)**

**throw new IndexOutOfBoundsException(**

**“SLRIndexList.get: invalid index = “ + index);**

**// index is valid**

**return recGet(first, 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, and another one for the index of the position being targeted. It must work as recursively suggested next: *recGet(f, i)* is *f.getElement()* if *i==0*; otherwise (when *i>0*), the result is that of *recGet(f.getNext(), i-1)*.

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., is does not alter the list. The other operations do alter the list. But, notice that the goal is to hit with 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 this one has side effects on the list. For the case of the *remove* operation, the relevant node is 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, by virtually increasing its corresponding index value by one; however, you should notice that no data movement is necessary.

A partial implementation for class **SLRIndexList** follows:

**public class SLRIndexList<E> implements IndexList<E> {**

**private SNode<E> first;**

**private int size;**

**public SLRIndexList() {**

**first = null;**

**size = 0;**

**}**

**public boolean isEmpty() {**

**return size == 0;**

**}**

**public int size() {**

**return size;**

**}**

**public E get(int index)**

**throws IndexOutOfBoundsException**

**{**

**if (index < 0 || index >= size)**

**throw new IndexOutOfBoundsException(**

**“SLRIndexList.get: invalid index = “ + index);**

**// index is valid**

**return recGet(first, index);**

**}**

**public void add(int index, E e)**

**throws IndexOutOfBoundsException**

**{**

**if (index < 0 || index > size)**

**throw new IndexOutOfBoundsException(**

**“SLRIndexList.add: invalid index = “ + index);**

**// index is valid for the add operacion**

**first = recAdd(first, index, e);**

**size++;**

**}**

**public E remove(int index)**

**throws IndexOutOfBoundsException**

**{**

**if (index < 0 || index >= size)**

**throw new IndexOutOfBoundsException(**

**“SLRIndexList.remove:invalid index = “ + index);**

**// index is valid for remove operation**

**E etr = …**

**…**

**size--;**

**return etr;**

**}**

**public E set(int index, E e)**

**throws IndexOutOfBoundsException**

**{**

**if (index < 0 || index >= size)**

**throw new IndexOutOfBoundsException(**

**“SLRIndexList.set: invalid index = “ + index);**

**// index is valid for set operation**

**return recSet(first, index, e);**

**}**

**// Auxiliary recursive methods**

**private static <E> E recGet(SNode<E> f, int i)**

**{**

**if (i == 0)**

**return f.getElement();**

**else**

**return recGet(f.getNext(), i-1);**

**}**

**private static <E> SNode<E> recRemove(SNode<E> f, int i)**

**{**

**if (i == 0) {**

**SNode<E> ntd = f;**

**f = f.getNext();**

**ntd.clean();**

**return f;**

**}**

**else {**

**f.setNext(recRemove(f.getNext(), i-1));**

**return f;**

**}**

**}**

**// TODO… see the code you have received…**

**private static <E> SNode<E> recAdd(SNode<E> f, int i,E e) { … }**

**private static <E> E recSet(SNode<E> f, int index, E e)**

**{ … }**

**…**

**}**

The following is the verbal description of the *regGet* and *recAdd* methods.

**… <E> SNode<E> recGet(SNode<E> f, int i)** – Returns the reference to 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 value **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> SNode<E> recAdd(SNode<E> f, int i, E e)** – Inserts a new node to the linked list whose first node is being referenced by **f** so than the new node contains the data element **e** and it ends up occupying 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 *first*. Think about this: when is the new value assigned to first different from its current value?

**public void add(int index, E e)**

**throws IndexOutOfBoundsException**

**{**

**if (index < 0 || index > size)**

**throw new IndexOutOfBoundsException(**

**“SLRIndexList.add: invalid index = “ + index);**

**// index is valid for the add operacion**

**first = recAdd(first, index, e);**

**size++;**

**}**

**Exercises on Recursion and Linked Lists**

**Exercise 1.** Write a verbal description (similar to what is done with other methods in the discussion section) for each of the following two methods

* **... <E> SNode<E> recRemove(SNode<E> f, int i)** –
* ... **<E> SNode<E> recSet(SNode<E> f, int i, E e)** –

**Exercise 2. Working Toward a Full Implementation of the SLRIndexList Class** You have received a partial implementation of class **SLRIndexList**, as well as a tester class which can be used to run different tests over an implementation of **IndexList**. For the **SLRIndexList**, you are required to implement all the missing methods, one by one, and execute the test class after each new implementation. Remember, we will use the tester as it is. Moreover, you cannot alter any other part of the **SLRIndexList** 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 methods 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 know better what 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 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 sill missing so that you can effectively test each method immediately after implementing it. The tester class creates an object of type **IndexList** (in this case, of type **SLRIndexList<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 the remove? Why? Notice that, you may need to implement more than one method before doing any test. In the particular case at hand, with the *get* method already implemented, we should begin with the *add* method, then we can implement either the remove or the set in any order. Also notice that since the class implements the **IndexList** 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 need to go back to those incorrect implementations and make the appropriate changes. Be alert to the possibility that apparent errors being detected on other already implemented methods may not be real errors, but incorrect behavior due to real errors in the most recently implemented method.

Let’s have some guidance in this work.

**Implementing the *add* and the *recAdd* methods**. In the case of the *add* method, just remove the comment operators surrounding the two last statements inside the method and work on the implementation of the *recAdd* method in accordance to the verbal description given in the discussion part. A general skeleton for this operation is as follows:

**private static <E> SNode<E> recAdd(SNode<E> f, int i, E e)**

**{**

**// adds a new node (containing e) as the ith node in**

**// the list (the current list) whose first node is f…**

**if (i==0) {**

… add the new node (properly initialized) as the first node in the current list

… whose first node is being referenced by f

… return reference to the new 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 too repetitive, but that’s recursion…

In the partial implementation that you have received, you need to remove the comment delimiters surrounding the following incomplete code:

**E etr = ...;**

**...**

**size--;**

**return etr;**

The objective of **etr** is to keep a reference to the data element that the remove returns (the one being removed from the list). You need to get hold of that data element before removing its node from the linked list. That removal is done by **recRemove**.

**Implementing the recSet methods**. Study the code (set method) and complete the implementation of **recSet**. Think recursively.

**Exercise 3**. Let’s now work on a recursive algorithm to **sort a list** of the type studied. To the **IndexList** 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 the comparator object. It establishes the**

**relation order upon which the ordering will be based.**

**\*\*/**

**void sort(Comparator<E> cmp);**

Remember that the 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 the comparator for a particular order relation. It 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();**

**}**

**}**

Notice that the method **compare** returns negative value (<0) if **s1**’s gpa is less than **s2**’s gpa…. and so on… This operator establishes an order relation on students, which is based on their gpa. With this type of comparator, to sort an **SLRIndexList<Student>** object being referenced by **w**, in increasing order of 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, we shall be using objects for 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 technique. .

**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 studied in lectures (in that case, for sorting an array), adapted to sort a list based on the following recursive idea. To sort a (non-empty) linked list whose first node is referenced by variable **first**, we can do the following:

1. If only one node, then nothing needs to be done -- the list is sorted
2. Otherwise,
   1. Sort the list whose first node is **first.getNext()** -- apply the same recursive strategy
   2. Insert that first node (reference by **first**) into the right place in that sorted 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 the comparator object. It establishes the**

**\* relation order upon which the ordering will be based.**

**\*\*/**

**public void sort(Comparator<E> cmp) {**

**if (first == null)**

**return; // list is empty – nothing to sort**

**else**

**first = *recInsertionSort*(first, cmp);**

**}**

**/\*\* Recursively sorts the linked list whose first node is**

**\* being referenced by variable “first” based on the given**

**\* comparator object. It finally returns the reference to**

**\* the first node of the sorted list.**

**\*\*/**

**private static <E> SNode<E> recInsertionSort(SNode<E> first,**

**Comparator<E> cmp)**

**{**

**if (first.getNext() == null)**

**return first; // size is 1, the list whose first node is first is already sorted**

**else {**

// The list has more than one element (more than one node). Sart by sorting the sublist whose first

// node is its second node (the second node in the list whose initial node is referenced by field: first.

**SNode<E> first2 = *recInsertionSort*(first.getNext(), cmp);**

// Now, that sublist must be sorted. We just need to insert the first node of the current list into that

// sorted sublist. Then return the first node of the list that results after insertion.

**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**

**\* the order implied by a comparator given.**

**\* @param <E> the generic data type of elements in nodes**

**\* @param nti the node to insert into the linked list given**

**\* @param first the first node of the list where the insertion**

**\* is to take place. The list is assumed sorted.**

**\* @param cmp the comparator upon which the sorting is based**

**\* @return reference to the first node of the resulting list**

**\* (the original list with the new node inserted into the right**

**\* place...)**

**\*/**

**private static <E> SNode<E> recInsertByOrder(SNode<E> nti,**

**SNode<E> first, Comparator<E> cmp)**

**{ … }**

To implement this method, think recursively following the idea that was discussed above. 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). Hence, the base case is:

**if (first == 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**. If the value in **nti** is less or equal (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 (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:

**recInsert(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 returns **first**. … think about it.