**UPR-Mayaguez**

**ICOM 4035 – Ejercicio de Laboratorio**[[1]](#footnote-0)

Array-based Implementations for **IndexList** ADT

**Introduction**

Once again, recall that an (**abstract data type)** ADT is a specification about a data type. It includes a general description of a data type, as well as the properties and the different operations that are valid for that type of object. For each operation, the ADT includes its name, the return type, the description of its parameters, and a specific description as to what the operation does based on the parameters. An ADT does not include details about implementation. A valid implementation for a particular ADT is a class that defines a data type that complies with the specification.

In this lab, we will revisit **IndexList** ADT, and work on another implementation: an array-based

**public interface IndexList<T> {**

**/\*\* Determines the size of the current list instance.**

**@return number of elements in the list**

**\*\*/**

**int size();**

**/\*\* Determines if the list is empty or not.**

**@returns true if empty, false if not.**

**\*\*/**

**boolean isEmpty();**

**/\*\* Accesses an element in the list that is located at the**

**position whose index is given.**

**@param index the index value of the position**

**whose element is being requested**

**@throws IndexOutOfBoundsException whenever the value**

**of index is not in the valid range from 0 to size-1.**

**@return reference to the particular element**

**\*\*/**

**T get(int index) throws IndexOutOfBoundsException;**

**/\*\* Removes an element from the list that is located at the**

**position whose index is given. No other element on the**

**current instance is affected. All remaining elements**

**maintain the same relative order. Any current member**

**(before the operation is applied to the list) that**

**is located at a position with index larger than the index**

**of the element to remove will occupy the position whose**

**index is one unit less than its current position. The size**

**of the list is reduced by 1.**

**@param index the index value of the position**

**whose element is being deleted**

**@throws IndexOutOfBoundsException whenever the value**

**of index is not in the valid range from 0 to size-1.**

**@return reference to the deleted element**

**\*\*/**

**T remove(int index) throws IndexOutOfBoundsException;**

**/\*\* Replaces an element in the list that is located at the**

**position whose index is given by a new value, also given.**

**No other element in the list is affected, and its size**

**remains the same.**

**@param index the index value of the position**

**whose element is being replaced**

**@param e reference to the new (replacing) element**

**@throws IndexOutOfBoundsException whenever the value**

**of index is not in the valid range from 0 to size-1.**

**@return reference to the element that was replaced**

**\*\*/**

**T set(int index, T e) throws IndexOutOfBoundsException;**

**/\*\* Adds a new element to the list. The new element will**

**occupy the position whose index is given. Any current**

**member of the list that is located at a position**

**with index larger than, or equal to, the index for the**

**new element to add will occupy the position whose**

**index is one unit more than its current position. The size**

**if the list is increased by 1.**

**@param index the index value of the position to be**

**occupied by the new element**

**@param e reference to the new element**

**@throws IndexOutOfBoundsException whenever the value**

**of index is not in the valid range from 0 to size.**

**@return reference to the element that was replaced**

**\*\*/**

**void add(int index, T e) throws IndexOutOfBoundsException;**

**/\*\* Appends a new element to the list. The size of the list**

**is increased by 1.**

**@param e reference to the new element**

**\*\*/**

**void add(T e);**

**}**

In the case of the array-based implementation, the internal data structure used to store the elements in a particular instance of the list is the 1-dimensional array. The array has a fixed initial capacity, but that capacity will be increased as the list capacity (available spaces) approaches its limit because the list size has increased. Moreover, we want to have some control of the space not being used. We do not want too many positions to be reserved, but not in use (in other words, we do not want too much wasted memory). Our approach will be as follows: the initial capacity of the internal array is fixed to 5. If a new element is to be added to the list, but that capacity has been exhausted, then the array capacity will be increased by 5. However, the array must be shrunk whenever there are more than 10 positions available. In that case, the capacity is reduced by 5. (These numbers must be planned carefully on a real implementation.)

**Implementation**  Let’s now work on the array-based implementation. We shall implement a class named ArrayIndexList. We need an instance field for the array and another for the size of the list at the moment. In addition, we include constant fields for the initial capacity (INITCAP), for the capacity to add or reduce when needed (CAPTOAR), and for the number of empty positions used as a reference to reduce the capacity (MAXEMPTYPOS).

**public class ArrayIndexList<T> implements IndexList<T> {**

**private static final int INITCAP = 5;**

**private static final int CAPTOAR = 5;**

**private static final int MAXEMPTYPOS = 10;**

**private T[] element;**

**private int size;**

**public ArrayIndexList() {**

**element = (T[]) new Object[INITCAP];**

**size = 0;**

**}**

… other methods required …

**}**

Figure 1 illustrates the memory organization for w, which corresponds to the list with five elements: (A, C, X, R, F). In this case, we assume an internal array of length 6.

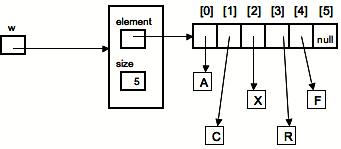


Fig. 1 : List w = (A, C, X, R, F).

The strategy to pursue is summarized as follows. The internal array is used to store the elements of the list’s instance. If not empty, say that its size is *n*, then the current elements of the list are stored in the first *n* positions of the internal array. The reference to the element of the list corresponding to the position indexed as *k* is stored in the position indexed as *k* in the internal array, for *k* = 0, 1, …, *n*-1.

Figure 2 illustrates what would be the memory content after executing **w.remove(1)**, where **w** was as in Figure 1.

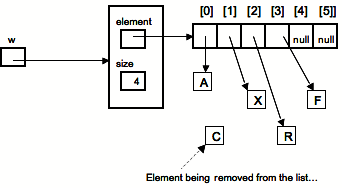


Fig. 2 : Illustration of memory after execution of **w.remove(1)**.

Figure 3 shows what would be the memory content after the execution of **w.set(3, e)**, where **w** was as in Figure 1.

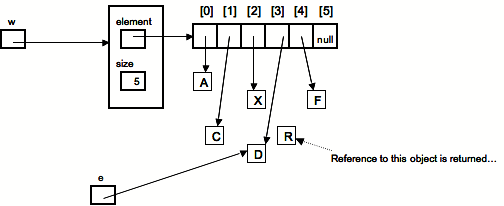


Fig. 3 : Illustration of memory after execution of **w.set(3, e)**.

Notice that the length of the array is the capacity of the list object, which implies the maximum number of elements that the list can hold. Once an array object is created, its length cannot be changed. However, the **IndexList** ADT specifies a list of unlimited capacity. Hence, we need to comply with this by changing the array to a new array of larger size whenever the current capacity is about to be exceeded. This happens when we apply the add operation to an instance whose size is equal to the length of the array. The instance shown in Fig. 1 has only one place left. The effect of executing **w.add(2, e)** over that particular instance is shown in Figure 4. Notice that values in the array are moved one position to its right as needed to open the space to accommodate the new element in the right position.

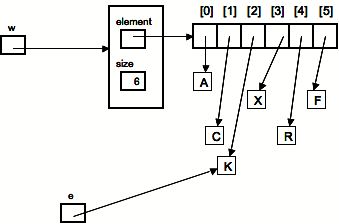


Fig. 4 : Execution of **w.add(2, e)**.

Now, if we try to add a new element to the instance shown in Figure 4, the array needs to be changed to a larger one. The add method needs to test for such possibility and execute the necessary instructions to increase the capacity, without losing any data currently in the actual list. For this, a larger array is created, the elements from the old array are copied to the new array, in the same order, and the value of the internal field element is changed to reference that new array. To help GC operations, assign **null** to each position in the old array. Figure 5 illustrates how the memory looks just after the new array is created, the data has been moved, the old array elements have been set to null, and element is set to reference the new array. Here, we increase the capacity by adding 5 new locations[[2]](#footnote-1).

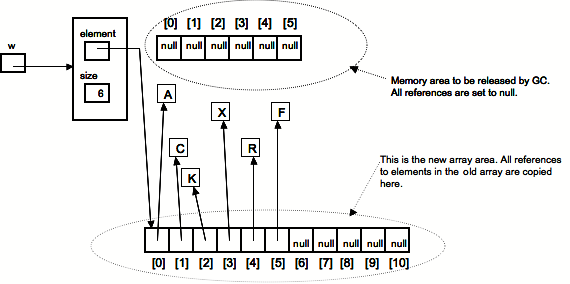


Fig. 5 : Increasing the capacity of the internal array.

What the figure shows can be achieved by calling **changeCapacity** with the right parameter value, which is the following private instance method.

**private void changeCapacity(int change) {**

**int newCapacity = element.length + change;**

**T[] newElement = (T[]) new Object[newCapacity];**

**for (int i=0; i<size; i++) {**

**newElement[i] = element[i];**

**element[i] = null;**

**}**

**element = newElement;**

**}**

The same method can be invoked (with appropriate argument value) to increase or to decrease the capacity…

**Exercises for this Lab Session**

**Exercise 1** Import the partial project received as a .zip file. You will finish the implementation of the class **ArrayIndexList**. We shall work in stages, each adding some new functionality, and test after each stage to verify the correctness of the particular functionality added. If not correct, then make the necessary corrections and test again. The process is repeated until successful tests are achieved. In this process, one needs to determine the proper order to proceed for the testing to be feasible. In addition, tests have to be carefully thought out for them to be effective and convincing. Remember, poorly designed tests may run successfully as they may not capture existing errors.

Testing is an essential process in the development of correct software. Some parts of a software system are simpler, and hence testing may be simple for those. Nevertheless, never hesitate to test: it is an essential part of producing correct software. As you get experience, you will notice that you are naturally driven to derive suitable tests for the particular code.

The particular case at hand is perhaps simple and you may start by implementing the whole class and test afterwards. However, you should learn to work on stages and to be able to perform testing incrementally as each state is worked out. In this way, you will be able to experiment partial progress that usually makes one feels good about what has been done. During the process, there might be parts of a project for which you do not devise immediate ideas of how to solve; in that case, you can concentrate on other parts while those ideas finally come. The goal is not to be delayed by those harder parts. Let’s derive a path based on testing.

* 1. Test for the appropriate change in size. Think about what is the minimum that you need to include on each operation to be able to effectively test that the **size** and the **isEmpty** methods work correctly. The methods that alter the size are just the methods to add a new element and the method to remove an existing element. Add the necessary code to just change the size in every operation where is needed, and as needed. For those operations, only include the instruction to change size as needed. Then, execute the tester; the output at this moment should be as shown in **Output\_Stage1.pdf**.
  2. Test for correct triggering of exceptions. Verify that each method that declares an exception triggers it only when it is correct to do so. Since all the exceptions in this case are based on the size of the list, if the size is already being handled properly, then it is feasible to proceed with this stage. Then, execute the tester; the output at this moment should be as shown in **Output\_Stage2.pdf**.
  3. Implement and test the methods **add(E e)** and **get(int index)**. As you proceed with this pattern: implement something more and test, the most appropriate methods to implement next are perhaps the previous two methods. We should begin the version of **add** that has only one parameter since it is simpler: just append a new element to the list. The harder part here is for the case when the internal capacity needs to be increased. For this, you just need to include the appropriate test and to invoke. The method to increase the capacity that was discussed earlier in this document is already inserted as a private method; just need to invoke as needed and whenever needed. After completing the implementation of these two operations, execute the tester; the output at this moment should be as shown in **Output\_Stage3.pdf**.
  4. Implement the **set(int index, E e)** operation. Then, execute the tester; the output at this moment should be as shown in **Output\_Stage4.pdf**.
  5. Implement the **add(int index, E e)** operation. Remember to increase capacity if needed to do so. Then, execute the tester; the output at this moment should be as shown in **Output\_Stage5.pdf**.
  6. Implement the **remove(int index)** operation. Remember to decrease capacity when needed to do so. When testing with this operation, remove the commented lines that prevent execution of the while loop **(while (!list.isEmpty()) …** inside the **main** method in the tester class. The output of executing the tester at this moment should be as shown in **Output\_Stage6.pdf**.

**Exercise 2** Add the following changes to test that the capacity of the internal array is being managed properly.

* + - 1. Modify the **IndexList** interface by adding the following method.

**int capacity();**

The purpose of this method is to return the integer value corresponding to the capacity of the current instance (the length of the internal array).

* + - 1. Add the correct implementation of the previous method inside the class **ArrayIndexList**; it is as follows:

**public int capacity() {**

**return element.length;**

**}**

* + - 1. Change the value for each of the internal static fields as follows:

**private static final int *INITCAP* = 1;**

**private static final int *CAPTOAR* = 1;**

**private static final int *MAXEMPTYPOS* = 2;**

* + - 1. In the tester class, append a new line to show the capacity of the list inside method **showList**. It should look as follows:

**private static void showList(IndexList<Integer> list) {**

**System.*out*.println("\n\*\*\* The "+ list.size()+**

**" elements in the list are: ");**

**int lpindex = list.size();**

**for (int i=0; i< lpindex; i++)**

***showElement*(list,i);**

**System.*out*.println("\n The capacity of the list is " +**

**list.capacity() + "\n");**

**}**

* + - 1. Activate the **while (!list.isEmpty())** … loop by properly removing the comment lines. Then run the tester again; the output produced should be as in file **Output\_Exercise2**.

1. Prof. Pedro I. Rivera Vega, ECE Department, UPR-Mayaguez, p.rivera@upr.edu [↑](#footnote-ref-0)
2. Actually, it can be shown that this is not a good alternative to increase the capacity by a constant value. This is further discussed in the textbook (Ch. 7), and we will see it in a future class. For the moment, we increase the capacity by a constant value when needed. [↑](#footnote-ref-1)