Data Structures Fall 2023
Lab Task 05: Array-Based List Implementation

Introduction to Array-Based List

Everyone is familiar with the term list. You might have a list consisting of employee data, student data, sales data, or a list of rental properties. One thing common to all lists is that all the elements of a list are of the same type. More formally, we can define a list as follows:

List: A collection of elements of the same type.

The length of a list is the number of elements in the list.

Following are some of the operations performed on a list:

1. Create the list. The list is initialized to an empty state.
2. Determine whether the list is empty.
3. Determine whether the list is full.
4. Find the size of the list.
5. Destroy, or clear, the list.
6. Determine whether an item is the same as a given list element.
7. Insert an item in the list at the specified location.
8. Remove an item from the list at the specified location.
9. Replace an item at the specified location with another item.
10. Retrieve an item from the list from the specified location.
11. Search the list for a given item.

Because all the elements of a list are of the same type, an effective and convenient way to process a list is to store it in an array. The size of the array can be specified when a list object is declared. It follows that, to maintain and process the list in an array, we need the following three variables:

- The array holding the list of elements
- A variable to store the length of the list (that is, the number of list elements currently in the array)
- A variable to store the size of the array (that is, the maximum number of elements that can be stored in the array)

LabTask

Task 1:

Design an application for **List** class using templates, with the following data members:

- A pointer of type T
- A variable *capacity* of type int to hold the total capacity of the list
- A variable *counter* of type int which holds the current index of the

list Your class should be able to perform the following operations:

- **insert(T item)**

  Inserts an item and increments the counter only if the list is **not full**

- **insertAt(T item, int index)**

  The function takes two parameters i.e., item of type T and an integer parameter which points to the position where the insertion is to be done. **Make sure you do not overwrite values.** For this you will have to increment the counter to create additional space for the item to be inserted.

- **insertAfter(T itemTobeInserted, T item)**

  The first parameter of this function represents the item to be inserted in the list, the second parameter represents the item already present in the list. You must search the item already present in the list and then insert the **itemtobeinserted** after **item**. *Use the functions of your own code instead of replicating logic.*

- **insertBefore(T itemTobeInserted, int item)**

  The first parameter of this function represents the item to be inserted in the list, the second parameter represents the item already present in the list. You must search the item already present in the list and then insert the **itemtobeinserted** before **item**. *Use the functions of your own code instead of replicating logic, i.e.*

- **isEmpty()**

  Checks if the *counter* variable is zero, i.e., the list is empty there are no elements.

- **isFull()**

  Checks if the counter variable has reached the total capacity of the **List**, which means *no more insertions* are possible.

  remove(T item)

  Removes item passed through parameters. Make sure you *update the counter* after removing one item.

- **removeBefore(T item)**

  You pass an item through parameter, then search for it in the array, then remove the item present before this item. i.e. if I have {1, 2, 3, 4} in array. I pass 3 to this function the resultant array should look like {1,3,4}. Make sure *you update counter after removing* one item.

- **removeAfter(T item)**

  You pass an item through parameter, then search for it in the array, then remove the item present after this item. i.e. if I have {1, 2, 3, 4, 5} in array. I pass 3 to this function the resultant array should look like {1,2, 3, 5}. Make sure you *update counter after removing* one item.

- **search(T item)**

  This function searches for the item passed through the parameter. If item exists, return index number. But if you didn't get the item in the list, what are you going to return?

- **print()**

  Print all the items of the List.

- **operator==(List& L)** we can overload == to compare the items of two lists. If the lengths

  of both the lists *aren't*

same, you don't have to go further to check for the items, i.e., the lists are already unequal. ·
reverse()

  Reverse all the elements of the List.

Don't forget to allocate memory to the array through constructor.

Task2

(Safe Arrays) In C++, there is no check to determine whether the array index is out of bounds. During program execution, an out-of-bound array index can cause serious problems. Also, recall that in C++ the array index starts at 0.

Design a class safeArray that solves the out-of-bound array index problem and allows the user to begin the array index starting at any integer, positive or negative. Every object of type safeArray should be an array of type int. During execution, when accessing an array component, if the index is out of bounds, the program must terminate with an appropriate error message.

Example Usage

Let's consider two examples to illustrate how the safeArray class works:

1. **safeArray list(2, 13);**

   In this example, ***list*** is an array of 12 components. The component type is int, and the valid components are list[2], list[3], ..., list[13].

   Range of indices : 2 till 13

   Out of bound cases : below 2 and above 13

2. **safeArray yourList(-5, 9);**

   In this example, ***yourList*** is an array of 15 components. The component type is int, and the valid components are yourList[-5], yourList[-4], ..., yourList[0], ..., yourList[9]

   Range of indices: -5 till 9

   Out of bound cases: below -5 and above 9

Design a class named safeArray that has the following

A pointer of type int
An integer variable upperbound
An integer variable lowerbound

Your class should be able to perform the following functions

safeAraay(lower, upper)

Constructor to initialize the array with given lower and upper bounds

Int & Operator [] (int index)

Function to access elements of the array with bounds checking

~safeArray()

Destructor to free the dynamically allocated memory