# MTE 140 - Spring 2009. Project 1.

A List Abstract Data Type. Due Date: Monday June 1st.

Hanan Ayad

## 1 Project Description

In this project, you will implement a List ADT using two different representations: a fixed-size array and a doubly-linked list. You are given the following source files:

1. The interface (header) file `listADT.h`, which has the type definitions and the function declarations for the list ADT.

2. An incomplete implementation file `fixedArray.c`, for which you are required to implement the body of each of the functions, based on the fixed array representation.

3. An incomplete implementation file `doublyLinkedList.c`, for which you are required to implement the body of each of the functions, based on the doubly-linked list representation.

4. An incomplete file `main.c` for which you are required to make functions calls for testing and demonstrating the implementation of each function [1]. Note that you need only one `main.c` that is common for both representations.

As a part of this projects, you are required to generate two different executable programs based on each representation, as follows.

1. Use the C compiler to compile - with no linking - each of the `.c` files separately. This step generates a `.o` file named identically to the source file. That is, for `fixedArray.c`, we obtain `fixedArray.o`, for `doublyLinkedList.c`, we obtain `doublyLinkedList.o`, and for `main.c`, we obtain `main.o`.

2. Use the compiler - with the linker option - to link `main.o` and `fixedArray.o`. This linking generates the first executable program based on the fixed array representation. Name this executable file `arrayTester` with extension `.exe` for an executable under Windows and without an extension for an executable under Unix.

3. Use the compiler - with the linker option - to link `main.o` and `doublyLinkedList.o`. This linking generates the first executable program based on the doubly linked list representation. Name this executable file `linkedListTester` with extension `.exe` for an executable under Windows and without an extension for an executable under Unix.

The above process will be further demonstrated during the lab using `Dev-C++` and `gcc` under Unix. Also, please read the submission instructions before starting on your implementation.

[1] You might also be provided with a basic command-line interface, which you would link with your program and use for testing your code (this is under development).

## 2 Module Design

Note that the way the type definitions are designed in this project are such that we don't need to explicitely pass a pointer to a pointer (i.e., using **), in the case of the linked list representation. By examining the file `doublyLinkedList.c`, you can observed that the pointer to the first node of the linked list (called `head`) is embedded as a field within a the `struct listTag`. The same can be said about the `tail` pointer.

Furthermore, note that the type `List` is defined as a pointer to a `struct listTag`. This way of defining `List` enhances the information hiding and enables the use the same of interface file `listADT.h` for both implementations. As a **Hint**, the function `list_create()`, for both implementations, should be responsible allocating a storage block of size `struct listTag`.

Also note that passing `List`, which is a pointer, to the required functions is the most efficient way to operate on the list and that there isn't any copying of data structures involved.

## 3 Specifications of the List ADT Operations

The functions you will be implementing for this project are described as follows:

1. `List list_create( )`. This function allocates memory space for a new list, initializes its fields, and returns a pointer to the allocated space. A `NULL` pointer is returned if the allocation fails.

2. `void list_destroy( List L )`. This function takes a `List L` (which is a pointer) and disposes (free) the space allocated for it.

3. `bool list_isEmpty( const List L )`. This function returns `true` is its argument refers to an empty list and returns `false` if the list contains one or more items.

4. `int list_length( const List L )`. This function returns the number of items in the list. It returns $-1$ if L is `NULL`.

5. `void list_print( const List L )` This function prints the data stored in the list, sequentially, in their linear order.

6. `bool list_insertFront( itemType value, List L )`. This function inserts a new item that stores the `value` argument as the first item of the `List` argument. It returns `true` if the item is successfully inserted and `false` otherwise.

7. `bool list_insertBack( itemType value, List L )`. This function inserts a new item that stores the `value` argument as the last item of the `List` argument. It returns `true` if the item is successfully inserted and `false` otherwise.

8. `bool list_deleteFront( List L )` This function removes the first item of the `List` argument. It returns `true` if the item is successfully removed and `false` otherwise.

9. `bool list_deleteBack( List L )`. This function removes the last item of the `List` argument. It returns `true` if the item is successfully removed and `false` otherwise.

10. `bool list_insert( unsigned int position, itemType value, List L )`. This function inserts a new item that stores the `value` argument at the given `position` of the `List` argument. It returns `true` if the item is successfully inserted and `false` otherwise.

11. `bool list_delete( unsigned int position, List L )`. This function removes the item at the given `position` of the `List` argument. It returns `true` if the item is successfully removed and `false` otherwise.

12. `int list_search( itemType Value, List L )`. This function searches for the first occurrence of the item that stores the `value` argument in the `List` argument. It returns the position of the item in the list if it is found and returns $-1$ otherwise.

13. `itemType *list_select( unsigned int position, List L )`. This function retrieves (access) the item at the given `position` argument in the `List`. It returns a pointer to the itemType at the selected `position`. It returns a `NULL` pointer if the element cannot be accessed.

14. `bool list_replace( unsigned int position, itemType value, List L)`. This function replaces the value of the item at the given `position` argument with the new `value` argument for the given `List`. It returns `true` if the replacement was successful and `false` otherwise.

## 4 Documentation and Programming Style

It is required that your name and ID appear at the top of all your submitted text files. Your code should be clear and understandable. You are required to use proper spacing and indentation to enhance the clarity of your code, use meaningful variable names, and useful comments. You should include comments for each function indicating its specifications. This includes a description of what it does, its parameters, return values, and any assumptions that it makes.

As part of your project, you are required to analyze the running time of each function. So, in your documentation within the implementation file, indicate with each function its running time as a function of the list size $n$ using the O-notation.

## 5 Submission Requirements

For the submission of this project, you are required to follow the steps outlined below:

1. Create a directory with the name `uwuserid_NNNNNNNNp1`, where `uwuserid` is your UW user ID, `NNNNNNNN` is your UW student ID number, and p1 denotes project 1.

2. Save all the sources files under this directory (do not create subdirectories). The files are: `listADT.h`, `fixedArray.c`, `doublyLinkedList.c` and `main.c`.

3. Implement the project and generate all the object (.o) files and the executable files under the same directory.

4. Zip the directory into one .zip file with the same name. That is, `uwuserid_NNNNNNNNp1.zip`. To check that this step was done successfully, create a new `test` directory and unzip the `.zip` file under `test`. The unzipping should result in creating a subdirectory under test with the name `uwuserid_NNNNNNNNp1`, with all the project files being extracted under this subdirectory.

5. Submit the file `uwuserid_NNNNNNNNp1.zip` into the dropbox designated for project 1 on UW-ACE (under MTE 140).