Lab Serie No. 02: Linear Linked Lists

Declaration Syntax

To represent a node in a Linear Linked List, we need to create a structure. Here's the syntax of the node's declaration:

```
/* Anonymous Structure */
typedef struct {
    <Type> data;
    struct node* next;
} Node;
```

*In this declaration, the structure is defined without a name (anonymous). The **typedef** is then used to create an alias **Node** for this unnamed structure. It is sometimes referred to as an anonymous structure because the structure itself doesn't have a name.*

```
/* Named Structure */
typedef struct node{
    <Type> data;
    Struct node* next;
} Node;
```

*In this declaration, the structure is given the tag name **node** as part of the definition. Then, the **typedef** is used to create an alias **Node** for this named structure.*

Exercise 01

1. Write functions to implement the following basic operations:
   - Chek if a Linear Linked List is empty.
   - Create a Linear Linked List of integers.
   - Insert new items with the given value into the list at any position. You must suggest the following menu to the user:

     Where do you want to insert the new element?
     1. At the beginning
     2. At the end
     3. At a given position

   - Travers the list and display the data member value of each node.
   - Count the number of items in the list.
   - Remove an item from the list (*by value*).
   - Destroy the list (*remove all its nodes*).
2. Assuming that the list is doubly linked, reimplement the previous operations.

Exercise 02 *(Additional questions)*

Write functions to implement the following problems:

- Sort the items of a Linear linked List of characters.
- Add a new character to the sorted list of characters.
- Reverse a singly Linked List.
- Count and return the occurrence of each character in the list (We only consider the alphabet characters).
- Remove duplicates of a given character from the list.

Appendix: *Reminder on Modular Approach in Programming*

Definition

Modular programming refers to the process of subdividing a program into separate sub-programs (module). It emphasis on breaking of large programs into small problems to increase the maintainability, readability of the code and to make the program handy to make any changes in future or to correct the errors. The syntax of function can be divided into 3 aspects:

1. **Function Declaration:** In a function declaration, we must provide the function name, its return type, and the number and type of its parameters. A function declaration tells the compiler that there is a function with the given name defined somewhere else in the program.

```
return_type name_of_the_function (parameter_1, parameter_2);
```

2. **Function Definition:** The function definition consists of actual statements which are executed when the function is called.

```
return_type function_name (para1_type    para1_name , para2_type    para2_name) {
// body of the function
 }
```

3. **Function Cals:** A function call is a statement that instructs the compiler to execute the function. We use the function name and parameters in the function call.

Code organization

— **Headers files (.h):** declaring function prototypes in header files is a fundamental practice in C programming that promotes code organization, reusability, and maintainability. It helps create clear interfaces, reduces code duplication, and facilitates collaboration in large projects.

— **Source files (.c):** Source files contain the actual C code that defines declaring functions.

When we declare function prototypes in a header file (h.), we need to use the ***#include*** *preprocessor directive to inform the compiler about those prototypes. This allows the compiler to recognize the functions' signatures before encountering their actual implementations. we have to include the header file at the top of the main source file (e.g., main.c), or any other file where we want to use these functions,*

```
#include "myheader.h"
```

Double Quotes ("")

When we include a header file using double quotes, the compiler searches for the header file in the current directory first. If it doesn't find the file in the current directory, it looks in the IDE include directories.

Angle Brackets (<>)

When we include a header file using angle brackets, like #include <stdio.h>, the compiler only searches for the file in the IDE include directories. It doesn't check the current directory.