BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS-F211: Data Structures and Algorithms

Labs 1 and 2

Introduction to the CS-F211 labs:

Welcome to the first lab of Data Structures and Algorithms! In this labsheet, we shall cover some crucial concepts that will be frequently used throughout the semester across all the labs; you will even need them in the lab tests. Most of these concepts were already covered in your Computer Programming course. Still, it is always a good idea to brush up on the foundational concepts of a subject before moving ahead. **We shall use this labsheet for the first two weeks of our labs.** Be sure to make full use of the labs to prepare yourself for the lab tests in the course. Do not be frightened by the length of this and the coming labsheets. They have been designed to act as reference material for you and as such contain all the relevant information and elaboration about the topics covered. You may feel free to go through the labsheets at your desired pace and take things easy.

With that in mind, let us begin.

Topics to be covered in this lab sheet:

- Multi-file Projects in C
- Makefiles
- Pointers and Dynamic Arrays
- File I/O handling
- Linked Lists
- Efficiency Estimation (Time Measurement)

Multi-file Projects in C:

Introduction:

Most of the programs that you have seen so far involve a single file that contains the entire source code of the program. But a project in C involves multiple **".h"** and **".c"** files that have to be integrated so that they work together. A **".h**" file is a header file that can contain global variables, structure definitions, constants, function declarations and macros, all of which may be shared across multiple C files and used by them. A **".c"** file is a source code file that contains actual source code and might be a program in and of itself.

Including a header file into a ".c" file enables us to use the variables declared and the libraries included in the header file. It also enables us to implement or use the functions declared in the header file. A simple practice in C programming is to keep all the constants, macros, global variables, and function prototypes (declarations) in the header files and include that header file wherever it is required.

Let us now see how to include a header file into our program.

Include Syntax:

The preprocessing directive **#include** is what is used for including both *system header files* (such as stdio.h, stdlib.h, string.h, etc.) as well as *user-defined header files* into a program. We use angled braces (< >) for system headers whereas we use quotes (" ") for user headers. Let us understand the two forms more closely:

```
#include <file.h>
```

This form is used for system header files that come with the compiler. It searches for a file named 'file.h' in a standard list of system directories. An example of this type of inclusion is `#include <stdio.h>`. This file inclusion makes functions such as *printf()* and *scanf()* accessible to our program. Without this inclusion, we will not be able to use the above functions. It is an exercise to try it out!

```
#include "myfile.h"
```

This form is used for header files that we create for our programs. It searches for a file named "*myfile.h*" in the directory containing the current ".c" file (the file in which you have written the above statement to include "myfile.h"). All the function declarations, structure definitions, and global variables that are present in myfile.h then become accessible in our ".c" file.

Example of a multi-file project:

Let us build a basic multi-file project step by step. (Create a separate directory for this project and place the upcoming files in that directory.)

Consider a function that takes as input a string (character pointer) and returns the number of vowels in the string. Such a function can be reused as many times as desired within the same program, but our aim is to make use of it even across multiple programs. How might we do that?

Write the following program into a file named *"count_vowels.c".*

```
#include "count.h"

int count(char* string)
{
    int count = 0;
    for (int i = 0; i < strlen(string); i++)
    {
        char character = tolower(string[i]);
        if (character == 'a' || character == 'e' || character == 'i' || character
== 'o' || character == 'u')
        {
            count++;
        }
    }
    return count;
}
```

The above program is defining a function count() that counts the number of vowels in a given input string. Now, create another file named *"count.h"* and put the following code into it:

```
#include <ctype.h>
#include <string.h>
int count(char* string);
```

The file *count.h* that we have created includes the function declaration of the function count(), along with its necessary header files. Whereas the file *count_vowels.c* contains the actual

function definition, and the header file *count.h* is also included in it. Notice how none of these files has a main() function in them. And the function count is also not being called anywhere.

Now, let us create a third file that contains a main() function and calls the function count() that we have just declared inside "*count.h"* and defined inside "*count_vowels.c"*. Create a file named *"master.c"* and paste the following code onto it:

```c
#include <stdio.h>
#include "count.h"

int main(void)
{
    char s[100];
    printf("Enter a string: ");
    scanf("%[^\n]s", s);

    int n = count(s);
    printf("Count = %d", n);
}
```

Notice that we have included *"count.h"* header file in "master.c". Now, if we try to compile the *master.c* using **`gcc master.c`**, you will encounter an error. This is because when we include the "count.h" file at its top, we are merely including the function declaration in our program (take another look at the contents of the "count.h" file), whereas the implementation of that function is still missing. In order to get the implementation, we would have to link the implementation file (which is actually "count_vowels.c") with our master file. Follow these steps for the same:

1. First, we must compile each of the ".c" files with the **"-c"** option with the gcc command. This option generates a corresponding ".o" or object file (an object file is a file containing object code, that is, machine code output of an assembler or compiler; the object code is usually relocatable, and not usually directly executable; it is an intermediate file between the source code and the executable files), but not the actual executable. Once we generate ".o" files for all the ".c" files, we can link them together to generate an executable.

   `gcc -c count_vowels.c`

```
gcc -c master.c
```

2. Step 1 has generated *count_vowels.o* and *master.o* files. You may verify that they have indeed been generated with the help of the **ls** command. These are the object files of the corresponding source files. Now, we link them together to create a single executable file using the gcc -o option:

   ```
   gcc -o count_vowels_exe count_vowels.o master.o
   ```

   Here, count_vowels_exe is the name of the executable that is created by linking count_vowels.o and master.o.

3. Now, we can run the above executable as follows

   ```
   ./count_vowels_exe
   ```

An example execution of the above executable is as follows:

```
Enter a string: I can now perform multi-file compilation to create a C project!
Count = 21
```

There are multiple advantages to this kind of programming. For example, we can see that "count_vowels.c" and "count.h" can be reused in multiple projects without any need to copy or change the code. In fact, there is no need to even compile "count_vowels.c" again, provided we have made no changes to it. The previously compiled "count_vowels.o" can be directly used for linking while creating the second executable.

There is another advantage of creating such modular files. In "count_vowels.c", we implemented the count() function declared in the "count.h" file. In this implementation, we counted the number of vowels in the input string. Now, say we wish to implement the count function but for counting consonants. And whenever a user wants to count the number of consonants, they may use this count() function, and whenever they wish to count the number of vowels, they may use the previous count() function. The switching should be doable without any change to our "master.c" file. This can be very easily achieved through the modular programming approach that we have just described.

Task 1: Create another file "count_consonants.c", in which you include "count.h" and implement the count() function, which this time counts the number of consonants in the input string. Now, generate the object file for this program and then create a different executable called

"count_consonants_exe" that links the original "master.o" file and your newly created "count_consonants.o" file. See that the two executable files (count_vowels_exe and count_consonants_exe) may now be executed independent of each other. Note that you shall not need to recompile "master.o" as no change has been made.

Shell scripts for compilation and linking:

We can write multiple stages of compilation and linking into a single shell script file. This will enable us to do recompilation for every change we make in any of the files in the project, to a single line.

- Create a file **"myScript.sh"** in the same directory where our **".c"** and **".h"** files are present.

- Add the following lines into **"myScript.sh"**
  ```
  gcc -c count_vowels.c
  gcc -c search_main.c
  gcc -o count_vowels_exe count_vowels.o search_main.o
  ./count_vowels_exe
  ```

- Now simply run the following command:
  ```
  sh myScript.sh
  ```
  This will execute all of the above commands. Shell scripts can be used to bundle multiple commands together and execute them in one go. Any command like **cp**, **mv**, **mkdir**, **echo**, etc. can be put inside a **".sh"** file. You can also run a shell script by: **bash myScript.sh**.

  You can also include the following lines in myScript.sh before the gcc statements:
  rm *.o
  rm *exe
  These two lines will remove the previously compiled **".o"** files and the previously created executables (files that end with the string **exe**). Then we can execute the remaining lines to freshly compile and create **".o"** files and the **exe**.

Task 2: Create a shell script for the count_consonants variant of count() that you have created in task 1 and then run it.

Home Exercise 1: Create a quiz bot that prompts a student the answers (A,B,C,D for the respective option; N for not attempted) to ten questions in sequence and stores that into a character array. There are three sets of question papers, and hence there are three sequences of correct answers. The array thus stored gets passed to a checker function present in the file containing the appropriate answer key for the student. The answer checker uses the answer key

that is inbuilt to the function, and this answer key is different for each set. The student's score is then calculated (+4 for correct answer, -1 for incorrect answer, 0 for unattempted questions; note that the minimum possible score is 0, a student cannot score negative overall) by the answer checker, and the master program that is conducting the quiz then returns the score.

You should create the main() function in quiz.c file. This function takes the answers from the users, stores them in a character array and invokes the anwer_checker() function by passing the above character array as an argument. The answer_checker() function is to be declared in set.h header file, and it has to be implemented in setA.c, setB.c and setC.c, separately as per the key for each of the sets. You must include header files appropriately in each of the files. Create three separate shell scripts for each of the answer sets. [The idea behind this is that the quizzer knows which set the particular student is getting and runs the respective shell script accordingly, and once the student is done answering, it returns the score to them.]

Makefiles:

A makefile exists to aid in compilation and recompilation of a C project. Let us understand the same with the help of an example.

Consider the vowel counting program that has been described earlier in this labsheet. We intend to run the **make** command on this project. The **make** command searches for a file named "makefile" in the current directory, and executes the shell commands written in it. So let us first create a file named *"makefile"* in the directory that contains the program files of our desired program. Now, write the following lines into the file:

```
# Note: Comment lines in makefiles begin with a hashtag
vowel : count_vowels_exe
	./count_vowels_exe
count_vowels_exe : count_vowels.o master.o
	gcc -o count_vowels_exe count_vowels.o master.o
count_vowels.o : count_vowels.c
	gcc -c count_vowels.c
master.o : master.c
	gcc -c master.c
clean :
	rm -f *.o
	rm count_vowels_exe
```

A Makefile consists of a set of *rules*. A rule generally looks like this:

```
target : prerequisite prerequisite prerequisite...
	command
	command
	command...
```

The *targets* are file names, separated by spaces. Typically, there is only one target per rule. The *commands* are a series of steps typically used to make the target(s). These **need to start with a tab character**, not spaces (ensure that the default indentation in your code editor is a Tab and not a bunch of whitespaces, otherwise the makefile would fail). The *prerequisites* are also file names, separated by spaces. These files need to exist before the commands for the target are run. These are also called *dependencies.*

The above makefile contains a few rules. Now, if we execute "make vowel" on the terminal, it would execute all the commands under the target "vowel" (running just "make" would also, by default, execute all commands under the first rule). Notice how the prerequisite for "make

vowel" is the executable file "count_vowels_exe". If this file doesn't exist in our program directory, it goes to the target "count_vowels_exe" and executes it to create the "count_vowels_exe" file. Note that the target "count_vowels_exe" is also dependent on "count_vowels.o" and "master.o" files. If these files don't exist in the program directory, the targets "count_vowels.o" and "master.o" would be executed to have these object files created. In this way, the makefile targets are recursively executed.

So, simply by calling the command "make vowel" on the shell, your entire C projects gets compiled and executed if the prerequisite files are present. This reduces the redundant recompilation overhead and adds huge convenience to the programmer as we no longer need to burden ourselves with syntactical sugar now.

You can also execute the other targets independently. For instance, if we run "make master.o", it will check first that master.c exists, and then proceed to compile it upto the object file stage.

Lastly, notice the "clean" rule. Executing this ("make clean" command) will delete all the ".o" files in this project and the linked executable as well.

Task 3: Experiment with the above makefile, and add more rules to perform the same operations with the consonants variant of the count function.

Pointers and Dynamic Memory Allocation:

Pointers:

Let us revise the basics of pointers in this section. A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is:

```
type* ptr_var_name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk ***** used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate that a variable is a pointer. Take a look at some of the valid pointer declarations:

```
int* ip;        /* pointer to an integer */
double* dp;     /* pointer to a double */
float* fp;      /* pointer to a float */
char* ch        /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between the pointers of different data types, is the data type of the variable or constant that the pointer points to. So, for instance, the variable ip in the above declaration is a pointer variable (a long hexadecimal) which holds the address of a variable which is an integer.

How to Use Pointers?

There are a few important operations which we will do with the help of pointers very frequently. (a) We define a pointer variable, (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using the unary operator * that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations:

```c
#include <stdio.h>

int main (void)
{
    int var = 20; /* actual variable declaration */
    int *ip; /* pointer variable declaration */
```

```
    ip = &var; /* store address of var in pointer variable*/
    printf("Address of var variable: %x\n", &var );
    /* address stored in pointer variable */
    printf("Address stored in ip variable: %x\n", ip );
    /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip );
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers: It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer. The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>
int main (void)
{
    int *ptr = NULL;
    printf("The value of ptr is : %x\n", ptr );
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
The value of ptr is 0
```

You can therefore use an **if** statement as shown to test whether or not a given pointer is a NULL pointer:

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

Dynamically Allocated Arrays (or simply, Dynamic Arrays):

Let us now use the concept of pointers to create dynamically allocated arrays and explore all their usecases.

The major problem with your typical arrays (in C) defined on stack is that its size is fixed. Once declared with a specific size, it can't be changed. Oftentimes we require array-like storage whose length can be altered (either increased or decreased) dynamically at runtime. This is the purpose that a dynamic array serves. As such, dynamic arrays are very useful data structures. They can be initialized with a variable size at runtime. This size can be modified later in the program so as to *expand* or *shrink* the array. Unlike fixed-size arrays, dynamically sized arrays are allocated in the heap area of memory. Despite being resisable, they still provide random access to their elements.

We use pointers and C functions such as malloc(), free(), calloc(), and realloc() to implement dynamic-sized arrays. So, let us briefly revise what these functions do:

`malloc`: Calling this function is equivalent to requesting the OS to allocate some bytes of memory to our program. If the memory allocation is successful, malloc returns the pointer to the memory block. Else it returns NULL. malloc stands for "memory allocation". Its declaration looks like the following:

```
void *malloc(size_t size);
```

Creation:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n = 10;
    int* p = (int *) malloc(n);
    if (p == NULL)
    {
        printf("Unable to allocate memory\n");
```

```
        return -1;
    }
    printf("Allocated %d bytes of memory\n", n);
    return 0;
}
```

In the above snippet, we use malloc() to create **n** bytes of memory and assign it to the integer pointer **p**. However, it should be evident that something wrong has been done in the above program. We did not deallocate the memory. We must use the **free()** function to deallocate all our allocated dynamic memory, which we shall soon learn about.

Here is a simple example where we create a dynamically allocated array of 10 floats and store some data into it:

```
// Create a pointer to a float
float *p;

// Allocate 10 floats
p = (float*) malloc(10 * sizeof(float));

// Check that malloc didn't return NULL
if (p == NULL)
{
    return 1;
}

// Storing some float data into the array
for(int i = 0; i < 10; i++)
{
    p[i] = (float) i / 10;
}

// Do some tasks and then deallocate the memory
```

Accessing Array Elements: There are two ways of accessing elements of an array. One way is illustrated in the above example. **p[0]** refers to the first element, **p[1]** refers to the second

element, and so on until **p[n-1]** refers to the n'th element. Note that the array elements are contiguously stored in the program memory as illustrated in Figure 1 below.

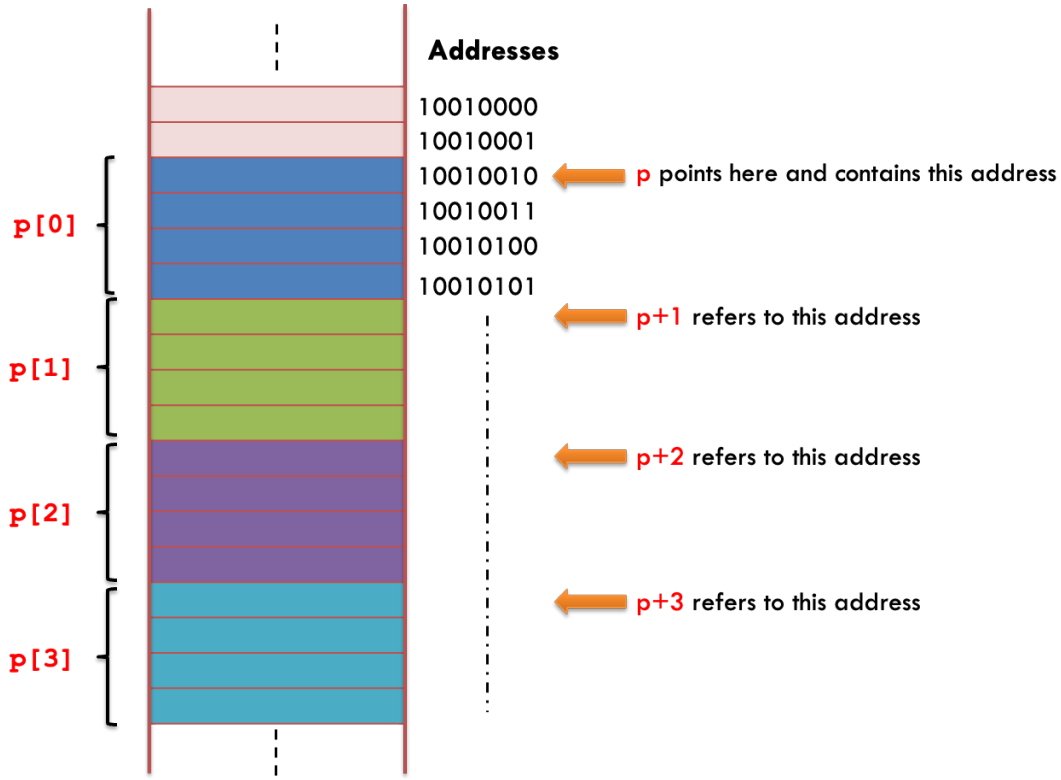

Figure 1: Illustration of float array **p** in memory

There is another way to access the array elements using pointer dereferencing. Note that the name of the array (say **p**) actually stores the address of the first element of the array **p**. So, to access the value of first element of the array we can access it using ***p**.

If we then increment the value of pointer **p** and dereference it, we can access the value of second element of the array, i.e., **p+1** gives us the address of the second element of the array and ***(p+1)** gives us the value of the second element of the array by dereferencing the address.

In general, if we increment a pointer, then NewPtr := CurrentPtr + N bytes, where N is the size of the datatype that CurrentPtr points to. In the above example, incrementing **p** by **1** advances **p** by 4 bytes (assuming float takes up 4 bytes). Similarly, if we add K to a pointer, then NewPtr := CurrentPtr + K * N bytes, where K is a constant integer and N is the number of bytes occupied by the datatype CurrentPtr points to. So, for example, **p + 3**, refers to the address of the fourth element of the array. And the difference in the addresses referred to by **p + 3** and **p** is 3*4 = 12

bytes. And we can access the value of the fourth element of the array using ***(p + 3)**.

This idea can be used to access the i'th element of any dynamically allocated array by simply adding **i** to its base pointer. Therefore: ***(ptr + i)** is equivalent to **a[i]**, if **ptr** is a pointer pointing to the base address of the array **a**.

In the above example, let us add the following lines of code:

```
// Displaying the stored data
for(int i = 0; i < 10; i++)
{
    printf("%f ", *(p + i));    // We could have also directly written p[i]
}
```

We are now able to access the elements stored in the array **p** (of course it is utter garbage unless you have stored something into it first), because the pointer **p** points to the base of the array, to which we are adding the index **i** to obtain the address of our element, and then we dereference that address with the ***** operator to get the array element itself. As mentioned in the comment, the array element can also be accessed directly as **p[i]**.

`calloc`: This function can allocate multiple contiguous memory blocks of given size and initializes each block to zero value, whereas malloc allocates a single memory block and the value at the pointed location is random or garbage. calloc allocates memory and zeroes the allocated blocks. calloc stands for "contiguous allocation".

```
void *calloc(size_t no_of_members, size_t size);
```

`no_of_members` represents number of memory blocks. **`size`** represents the size of each block. This is more suitable for allocating memory for arrays. Note that "zero value" doesn't just mean 0. If we are allocating an array of structs, calloc() assigns NULL value to strings, 0 to ints/floats, etc.

`realloc`: The realloc() function is used to resize the memory block pointed to a pointer that was previously allocated to the variable by the malloc() or calloc() function. realloc stands for reallocation. Its function header is the following:

```
void *realloc(void *ptr, size_t size)
```

- `ptr`: It is the pointer of the memory block which was previously allocated to the calloc(), malloc(), or realloc() function that is to be reallocated. If this pointer is NULL, then a new block is allocated and the pointer to it is returned by the realloc() function.

- `size`: It is the new size of the memory block which is to be reallocated. It is passed in bytes. If the size is 0, then the memory block pointed by ptr is deallocated and a NULL pointer is returned by the realloc() function even if ptr points to an existing block of memory.

If the realloc() request is successful, then it will return a pointer to the block of newly allocated memory. If the request fails, it will return a NULL pointer.

`free:` The free function deallocates dynamic memory. Calling free(p) just before return in the above snippet would have prevented the error. free *MUST* be called explicitly after the usage of dynamic memory, irrespective of which function is used to create it (malloc, calloc etc.). Its function header looks as follows:

```
void free(void *ptr);
```

Consider the following scenario:

```
int* p;
p = (int*) malloc(sizeof(int));
printf("Address pointed by p = %p\n", p);
free(p);
```

While we have deallocated the memory pointed to by the pointer **p** using the free() function, now the pointer **p** still points to the same memory location. Hence, p now becomes a *dangling pointer*, which means that **p** is not referring to a valid memory location of the program.

Now, consider a different scenario:

```
int* p; int* q;
p = (int*) malloc(1000*sizeof(int));
q = (int*) malloc(sizeof(int));
p = q;
```

Notice how **p** is made to point to another memory block without freeing the previous one. The memory previously allocated to **p** now becomes inaccessible. This is known as a *memory leak*.

Now, let us take a look at a detailed example using the above functions. The program given below creates a structure (**struct name**) for holding names (first name and last name) as character arrays. It creates a dynamically allocated array of **struct name** of size **n** to hold some names which it takes as input and then puts into the array. Then, it reallocates the array to have space for one more element, which is also then taken as input and added to the array. The contents of the array are displayed at both these steps so that we can verify the correctness of our program.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct name {
    char first[20];
    char last[20];
} Name;

int main()
{
    int n;
    printf("Enter size of the array: ");
    scanf("%d", &n);
    Name* arr = calloc(n, sizeof(Name)); // Creating enough space for 'n' names.
    if (arr == NULL)
    {
        printf("Unable to allocate memory\n");
        return -1;
    }
    printf("Enter the names (space separated): ");
    for (int i = 0; i < n; i++)
    {
        // Using . to access members of the struct
        scanf("%s %s", arr[i].first, arr[i].last);
        // Note that arr[i].first is equivalent to (arr+i)->first
    }
```

```
    printf("\nGiven array of names: ");
    for (int i = 0; i < n; i++)
        printf("%s %s\n", arr[i].first, arr[i].last);
    printf("\n");

    printf("Adding an element to the array.\n");
    Name newname;
    printf("Enter the name to be added: ");
    scanf("%s %s", newname.first, newname.last);

    arr = realloc(arr, (n + 1) * sizeof(Name));

    // Copying the new name to the end of the array
    strcpy(arr[n].first, newname.first);
    strcpy(arr[n].last, newname.last);

    printf("\nModified Array: ");
    for (int i = 0; i < n + 1; i++)
        printf("%s %s\n", arr[i].first, arr[i].last);
    printf("\n");

    free(arr);
}
```

An example run of the above program is given below (user input is underlined):

```
Enter size of the array: 3
Enter the names (space separated): Vito Corleone
Sonny Corleone
Fredo Corleone

Given array of names: Vito Corleone
Sonny Corleone
Fredo Corleone

Adding an element to the array.
Enter the name to be added: Michael Corleone

Modified Array: Vito Corleone
```

```
Sonny Corleone
Fredo Corleone
Michael Corleone
```

The above example demonstrates the use of dynamic memory allocation to create a particular dynamic array of structs and then resize it. It is illustrative of the powers of dynamic memory allocation, and you are expected to explore these ends on your own beyond this example.

Task 4: Create an interactive interface for a user who wishes to perform some operations on a dynamic array of *strings*. For this, you need to write a menu-driven program that asks the user for the initial length of the array, and then stores those corresponding strings (also taken as input) into a *dynamic array* initialised with that length. Then, begin a loop of the menu that prompts the user to select one of five options:

(a) Add a string to the end of the array,
(b) Add a string to the beginning of the array,
(c) Delete the element at index 'x' (taken as input) of the array,
(d) Display the length of the array,
(e) Display all the elements of the array in sequence.

The addition and deletion of strings must result in resizing of the array. The user should have a sixth option that would enable them to close the menu-driven program (thereby terminating the loop).

Perform the operation that the user selected. Write separate functions for each of the options.

Home Exercise 2: Mr David Hilbert is the manager of a hotel which he calls *The Grand Hotel*. Create a structure to hold metadata about *The Grand Hotel* such as its name (a string), address (also a string), number of rooms currently occupied (an integer), and a dynamic array of structs (pointer) that is initially empty. These structs will hold the information about the occupants of the hotel such as their names (string), their age (int), and permanent residential address (string).

After constructing both the structs, design a function that will allow Mr Hilbert to assign the first unoccupied room to a new visitor. Then, design another function that enables Mr Hilbert to assign Room #0 to a new visitor. This entails that he would request every occupant to shift to the next room (Room #1 -> Room #2, Room #2 -> Room #3, etc). Also, write a function that empties a particular room (when a visitor leaves). Mr Hilbert insists that all the occupants of

every succeeding room be shifted one place behind each to fill this gap (obviously, each of the above three functions would work by altering the dynamic array). Finally, create a function that displays the information of a particular visitor given the room number that they occupy presently. **[You must use the realloc function whenever a room is emptied or newly occupied to ensure that your dynamic array is always occupying only just as much space as it needs.]**

Now, write a menu-driven main program that initialises *The Grand Hotel* and then prompts the user (Mr Hilbert) to choose one of the above four options. The menu keeps reappearing after a task is completed as long as Mr Hilbert wants to operate the hotel.

Brain teaser: Now, just imagine if Hilbert's hotel were infinitely long (ie., if he had infinite rooms in his hotel!). This can obviously not be programmed due to memory restrictions, but understand that, theoretically, given infinite memory, we could keep resizing the dynamic array holding the information of the visitors, even if there came an infinite number of them. Just for fun, you can look up "Hilbert's infinite hotel paradox", which is illustrative of an interesting quirk of infinite sets in mathematics, which we have averted here since our set (array) is not infinite in the first place.

File I/O Handling:

Introduction:

What we have done so far is take input from the user directly at runtime by means of the scanf() function. But often we have data given to us in files that we may want to read directly from, rather than input one by one at runtime. Moreover, frequently, we would want to have our program outputs stored in a more permanent form on the disk (in files). In such cases, it becomes extremely important to be well-acquainted with file-handling concepts in your language of choice. All our labs shall be conducted in C, and will involve file I/O operations.

Files and File-related Functions in C:

A file is an abstraction of the way data gets stored in external or secondary memory (i.e. hard disk, flash memory device, optical disk etc.). A file can be treated as a sequential access FIFO list by a program i.e. if the contents of a file were (say, for instance):

Q W E R T Y

Then, the first accessible element is Q, then W, then E, and so on – i.e. to access E one has to access Q, then W, and then access E; and when say a value U is added to the above file, the contents would be:

Q W E R T Y U

i.e. if and when a new element is added it is added to the end (of the list i.e. file). Linux supports text and binary files. We will only deal with text files in this course. Text files can be accessed character by character or word by word (i.e strings separated by space).

C provides the I/O library **stdio** that contains procedures (or functions) for I/O access. The header file “stdio.h” contains the headers (i.e. declarations) of these functions. We must include this header file in our program to perform I/O Operations.

C libraries support procedures **fscanf(), fgets(), fprintf(), and fputs()** for reading and writing to a file. Refer to the manual pages for information on how to use these procedures. They are similar to scanf and printf but take an additional (first) argument which is the file pointer.

Since files are abstractions of persistent physical storage, typically initialization and finalization are required, i.e. initialization must be done before any read/write operations, and finalization must be done after all read/write operations (particularly before close of program execution). C

libraries provide procedures **fopen()** and **fclose()** for initialization and finalization of a file. Refer to the man pages for information on how to use these procedures.

We have briefly summarised the use of some file-related functions below:

`fopen()`: To open a file.
Declaration: `FILE *fopen (const char *filename, const char *mode)`
Description: fopen() function is used to open a file to perform operations such as reading or writing on it. In a C program, we declare a file pointer and use fopen() as shown below.

```
FILE* fp;
fp = fopen("filename", "mode");
```

fp: file pointer to the data type "FILE".
filename: the actual file name including the path of the file.
mode: refers to the operation that will be performed on the file. This shall be elaborated below.

`fclose()`: To close a file.
Declaration: `int fclose(FILE *fp)`
Description: fclose() function closes the file that is being pointed by file pointer **fp**. We close a file as shown below.

```
fclose(fp);
```

`fgets()`: To read from a file.
Declaration: `char *fgets(char *string, int n, FILE *fp)`
Description: fgets() function is used to read a file line by line as shown below.

```
fgets (buffer, size, fp);
```

buffer: the buffer where the data is to be put.
size: size of the buffer.
fp: file pointer.

`fprintf()`: To write into a file.

Declaration: `int fprintf(FILE *fp, const char *format, ...)`
Description: fprintf() function writes a string into a file pointed by **fp** as shown below.

```
fprintf (fp, “some_data”);       //or
fprintf (fp, “text %d”, variable_name);
```

`fscanf()`: To read from a file.
Declaration: `int fscanf(FILE *fp, const char *format, ...)`
Description: fscanf() function reads formatted input from the file as shown below.

```
fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);
```

The typical structure of a program fragment that reads from and/or writes to a file is as follows:

```
FILE *f = fopen(“testfile.txt”, <mode>); // returns a file pointer
/* read or write operations performed using the file pointer f */
fclose(f);
```

The <mode> in the above statement refers to the mode in which you want to open your file. The table below describes the modes that can be used:

r	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. If the file cannot be opened fopen( ) returns NULL.
rb	Open for reading in binary mode. If the file does not exist, fopen( ) returns NULL.
w	Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open the file.
wb	Open for writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open the file.

ab	Open for append in binary mode. Data is added to the end of the file. If the file does not exist, it will be created.
r+	Searches file. It is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. Returns NULL, if unable to open the file.
rb+	Open for both reading and writing in binary mode. If the file does not exist, fopen( ) returns NULL.
w+	Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open the file.
wb+	Open for both reading and writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
ab+	Open for both reading and appending in binary mode. If the file does not exist, it will be created.

As described above, if you want to perform operations on a binary file, then you have to append '**b**' at the end of the mode. For example, to perform write operations on a text file named "example.txt", the file pointer can be opened as:

```
filePointer = fopen("example.txt", "w")
```

filePointer now holds the file pointer for that file and can subsequently be passed to functions like fprintf(). The second parameter can be changed to contain any of the attributes listed in the above table.

We shall now take a look at some examples to illustrate the use of file pointers in reading from and writing to files. Go through them and understand the exact workings of the functions used.

Sample 1: Writing something to a text file using fprintf()

```c
#include <stdio.h>

int main()
{
```

```
    int num;
    FILE *fptr;

    fptr = fopen("program.txt", "w");
    if(fptr == NULL)
    {
        printf("Error opening file!");
        exit(1);
    }

    printf("Enter num: ");
    scanf("%d", &num);
    fprintf(fptr, "%d", num);
    fclose(fptr);

    return 0;
}
```

Note that opening a file in write mode (“w”) automatically creates the file (empty file) in case it does not exist already.

Sample 2: Reading a numeral from a text file using fscanf()

```
#include <stdio.h>
int main()
{
    int num;
    FILE *fptr;

    fptr = fopen("program.txt", "r");
    if (fptr == NULL)
    {
        printf("Error opening file");
        exit(1);
    }

    fscanf(fptr, "%d", &num);
    printf("Value of n = %d", num);
    fclose(fptr);
```

```
    return 0;
}
```

Sample 3: Reading from a .csv file using fscanf()

```c
#include <stdio.h>

int main(void)
{
    FILE *fptr;
    char name1[20], name2[20];
    int ID1, ID2;

    fptr = fopen("test.csv", "r");
    fscanf(fptr, "%[^,],%d\n", name1, &ID1);
    fscanf(fptr, "%[^,],%d\n", name2, &ID2);
    // Recall from CP that %[^,] means read until comma

    printf("Name1: %s, ID1: %d\n", name1, ID1);
    printf("Name2: %s, ID2: %d\n", name2, ID2);
    fclose(fptr);
}
```

Note that if the .csv file has multiple records, we can read them all very conveniently with a loop [prototype: `while(fgets(line,100,fp)){...}`], and we can keep storing the values wherever we desire. The way shown above is one way to tokenise the read lines. There are other ways to do the same, for instance, using the strtok() function. You may explore these techniques on your own. [You may refer to the following article for the same: https://www.geeksforgeeks.org/relational-database-from-csv-files-in-c/].

What you have seen above is one way of performing what is known as “inter-process communication”. One program can “communicate” with another by writing what it wants to communicate to a file, which can then be read by the other program. At a different level, even users can communicate similarly. One user can place their message into some text file, which is then read by the other user.

Task 5: Write a program that reads *itself* (the .c file containing its own source code) and displays on the terminal each line present in the program. You may explore the use of the “`__FILE__`” macro to solve the same without hardcoding the filename into the fopen() function (the

`__FILE__` macro expands to a string whose contents are the filename of the current program; for example, if we were to write `printf("%s", __FILE__)`, then we would end up printing the name of the file containing the source code of our current program).

Task 6: Write a program that cuts/moves the entire contents of one text file into another, leaving behind the original file blank. Before writing the program, manually create two files "test1.txt" and "test2.txt" and write some sample text into "test1.txt", which you must then programmatically move to "test2.txt", thereby leaving behind "test1.txt" blank.

Home Exercise 3: You have been given a text file that contains the complete text of J. R. R. Tolkien's magnum opus, *"The Lord of the Rings"*. This text file is called "LOTR.txt". Write a program that reads it and counts the number of times the word "hobbit" appears in the text (case insensitive) and then displays the count. You may need to use some functions supported by string.h and ctype.h in addition to the file I/O functions.
[Refer to https://www.tutorialspoint.com/c_standard_library/ctype_h.htm and https://www.tutorialspoint.com/c_standard_library/string_h.htm if required.]

Home Exercise 4: You are an engineer working at the Space Communications and Navigation division of NASA. Earlier, NASA had sent two astronauts to the moon to inspect a recent crater that has supposedly formed out of nowhere on its surface. Due to certain restrictions, the astronauts can only telecommunicate with you in *Morse code*.

For some reason, there has been no information from the astronauts ever since they landed. This is quite alarming because the supplies that they had taken with them could only last them a month on the moon. After months of frightening silence, finally, it seems that a message has been sent to your receptor. Your colleagues have already transcribed the Morse code into a text file named "msg.txt". Now, your job is to write a program that reads the text file and decodes the message that has been sent from the astronauts' transmitters.

The complete Morse code chart is given to you in the file "Morse.jpg". Please refer to it while constructing your program. (A forward slash '/' has been used to denote space between words in the message; blankspace ' ' has been used to denote space between letters.)

Linked Lists:

Introduction:

Recall how the major disadvantage of using fixed-size arrays was the fact that their length could not be altered at runtime. To avert this, we decided to make use of dynamic arrays whereby we would realloc() every time we had exhausted the space allocated to us. There is actually another way to go about this, and that is with the help of the data structure known as "linked list".

Arrays are random access lists, whereas linked lists are sequential access lists. The key difference here is that unlike arrays, linked list elements are not stored at a contiguous location in the memory; the elements are linked using pointers. They include a series of connected nodes. Here, each node stores the data and the address of the next node. There is a head pointer that points to the first node in the linked list and from there on, each node contains its own data as well as the pointer to the next node. The very last node in the linked list points to **NULL**, since it is the last node in the list, thus marking the end of the linked list. Figure 2 below pictorially shows what a linked list looks like, while Figure 3 illustrates how linked lists are stored in memory.

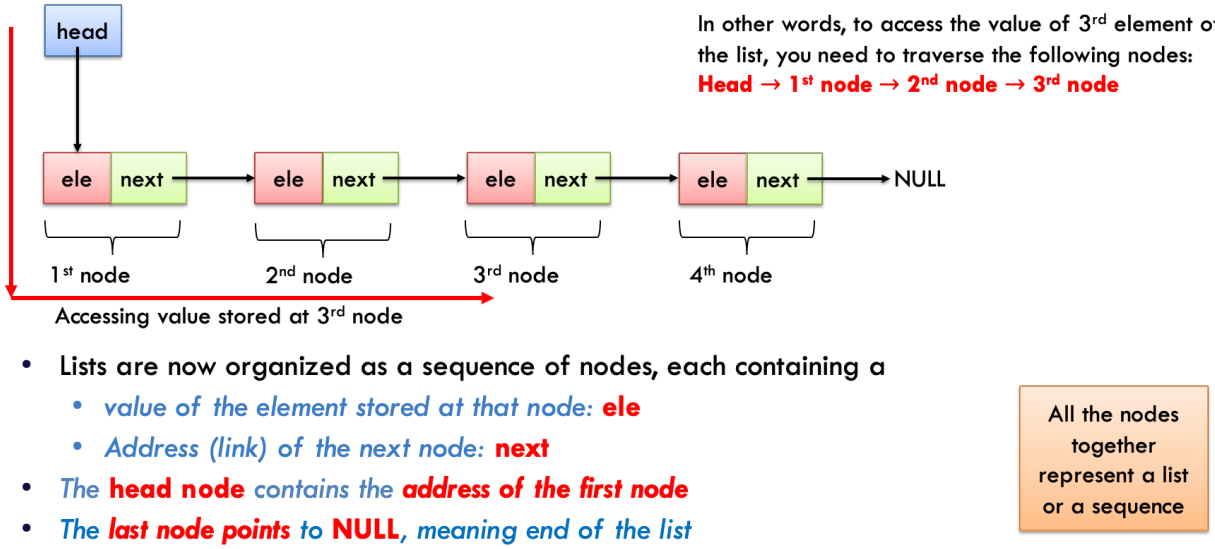

Figure 2: Illustrating a linked list

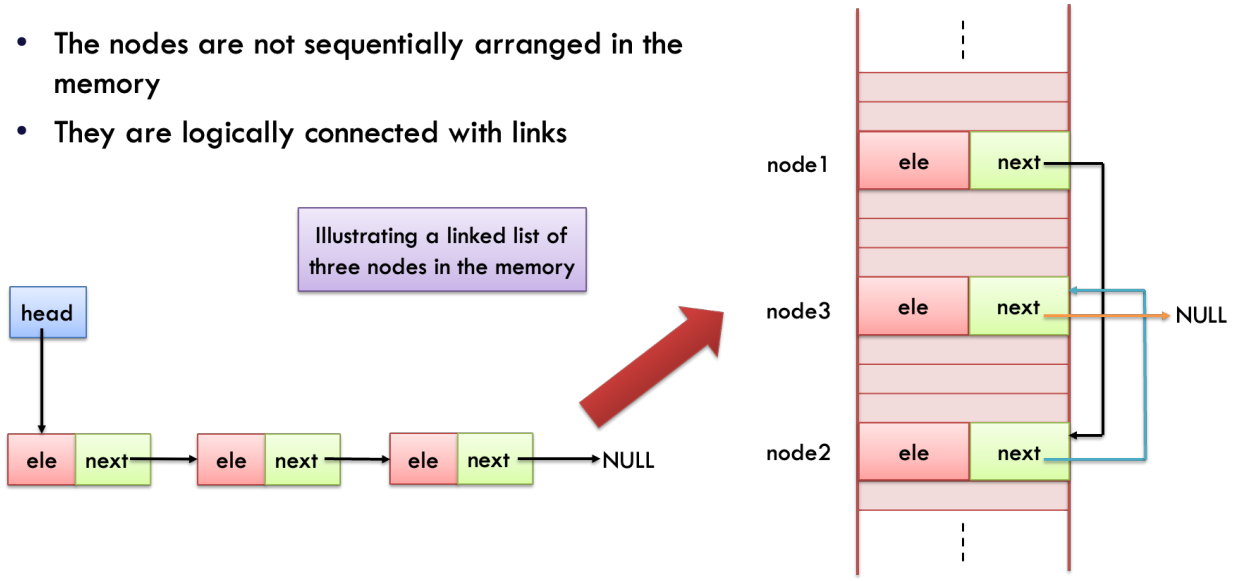

Figure 3: Illustrating linked lists in memory

Advantages and Disadvantages:

Consider the scenario where you have to delete an element from a dynamic array (say this element is somewhere in the middle). In that case, after deleting the element, to avoid leaving behind gaps in memory, you would need to shift all the elements to the right of the deleted element one step each to the left. Something similar also happens when you want to insert a new element at a particular position in the array. Even in that case, you cannot insert the element until you have made some space for the element; that can only be done by moving every element from that position up to the rightmost end rightward by one step.

This makes insertion and deletion very expensive processes in dynamic arrays. There is also the additional disadvantage of performing a realloc() whenever we want to resize the array, which is also potentially a very expensive process.

Linked lists avert the above problems by making insertion and deletion very simple tasks. Insertion for a linked list simply entails that you create a node and then traverse to the position where you want to perform the insertion, and then just adjust the *next* pointers of the nodes there to fit this node into the position. This is a far simpler process than shifting every element to the right. Deletion is also implemented very similarly to insertion (Hint: the free() function will be used here).

The diagram below depicts how insertion is to be done into a linked list. We are inserting a new element at the beginning of an existing list. Please note that changes in the pointers. The blue

lines indicate new connections and the crossed connections are removed. Also note the increment in the value of the variable count which stores the number of nodes in the list.

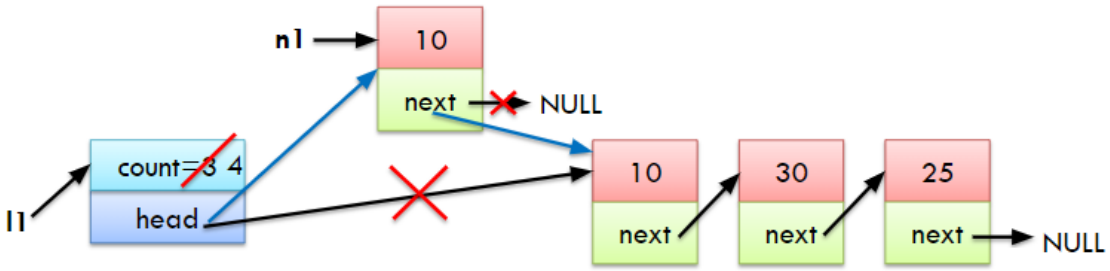

Figure 4: Illustrating insertion of a node at the beginning of a given linked list

We have provided you with some sample code for the implementation of linked lists. Note that we have only implemented the **insertAfter()** and **printList()** functions as demonstrative examples. You may refer to the **lectures slides for lectures 3-5** to get a better understanding of these and also on how to implement the rest.

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node* NODE;
struct node{
    int ele;
    NODE next;
};

typedef struct linked_list* LIST;
struct linked_list{
    int count;
    NODE head;
};

LIST createNewList()
{
    LIST myList;
    myList = (LIST) malloc(sizeof(struct linked_list));
    myList->count=0;
    myList->head=NULL;
    return myList;
```

```
}

NODE createNewNode(int value)
{
    NODE myNode;
    myNode = (NODE) malloc(sizeof(struct node));
    myNode->ele=value;
    myNode->next=NULL;
    return myNode;
}

void insertAfter(int searchEle, NODE n1, LIST l1)
{
    if(l1->head==NULL)
    {
        l1->head = n1;
        n1->next = NULL;
        l1->count++;
    }
    else
    {
        NODE temp = l1->head;
        NODE prev = temp;
        while(temp!=NULL)
        {
            if (temp->ele == searchEle)
                 break;
            prev = temp;
            temp = temp->next;
        }
        if(temp==NULL)
        {
           printf("Element not found\n");
           return;
        }
        else
        {
            if(temp->next == NULL)
            {
```

```
                temp->next = n1;
                n1->next = NULL;
                l1->count++;
            }
            else
            {
                prev = temp;
                temp = temp->next;
                prev->next = n1;
                n1->next = temp;
                l1->count++;
            }
            return;
        }
    }
    return;
}

void printList(LIST l1)
{
    NODE temp = l1->head;
    printf("[HEAD] ->");
    while(temp != NULL)
    {
        printf(" %d ->", temp->ele);
        temp = temp->next;
    }
    printf(" [NULL]\n");
}
```

It is left as an exercise for you to write a **main()** program for the above to test the functions.

Task 7: Expand the above program to include the following functionalities:

(a) deleteAt(): Deletes the node at the specified position.
(b) insertFirst(): Inserts a new node at the starting position of the linked list.
(c) deleteFirst(): Deletes the node at the starting position of the linked list.
(d) search(): Scans through the linked list for a specific value (data) and returns the position of the node where the data is found. If not found, it returns -1.

Bear in mind that linked lists also have certain disadvantages. For instance, we no longer have the ability to perform random access. This means, given a position, we would need to traverse through all the elements prior to it to get to that position in a linked list. Whereas in a dynamic array we can directly access it using **arr[i-1]** for i^{th} position (element). Even in a sorted linked list, we cannot perform binary search (since we cannot perform random access), whereas the same can be done in a sorted dynamic array. In essence, arrays are faster to access and are more suitable for searching. And linked lists are more suitable for faster inserts if they are random and dynamic. Whether to choose linked lists or arrays for representing our data is a choice that we as programmers make depending on our application and the problem that we are trying to solve.

There are also variations to linked lists. Some of them are briefly described below:

(1) Singly Linked List: The linked list that we have discussed above is a singly linked list. Each node contains a single pointer, and that pointer points to the next node (which is NULL in case of the last or right-most node). Here, we can traverse the list in one direction only.
(2) Doubly Linked List: Very similar to singly linked list, but now each node contains two pointers. One points to the next node and the other points to the previous node. This allows for two-way traversal in the list. Of course, here we will have two pointers pointing to NULL (the previous of the left-most node and the next of the right-most node).
(3) Circular Linked List: This is a singly linked list with its last node's next pointer pointing to the first node in the list, instead of NULL. This means that if we were to traverse this list, the traversal would never terminate as we would be encircling the same set of nodes again and again.

Linked lists, in all their forms, are extremely useful data structures and are used in multiple applications. It is left as an exercise for you to implement the above variations of linked lists and find out their applications.

Task 8: Write a function rotate() that takes a linked list of integers as input and rotates the elements left/anticlockwise/towards head by k steps. You will need to create your own linked list of integers for this, before implementing the above function.

For example, if the linked list originally was `[HEAD] -> 1 -> 2 -> 3 -> 4 -> 5 -> [NULL]`, and k = 2, then the linked list should become `[HEAD] -> 3 -> 4 -> 5 -> 1 -> 2 -> [NULL]`.

[Hint: For solving the problem, it may be helpful to make the linked list circular first.]

Home Exercise 5: Write a function that takes a linked list of integers as input and reverses all of its elements.
For example, if the linked list originally was `[HEAD] -> 1 -> 2 -> 3 -> 4 -> [NULL]`, then after reversal, the linked list should be `[HEAD] -> 4 -> 3 -> 2 -> 1 -> [NULL]`.
[Ideally, you should be able to perform this in one traversal of the linked list without using too much extra space; in terms of complexity, your solution should run in O(N) time and take O(1) auxiliary space]

Home Exercise 6: The largest numeric datatype in C, long long integers, has a maximum value of 9,223,372,036,854,775,807 (and 18,446,744,073,709,551,615 if unsigned). Oftentimes, this is not enough, and we want to store more digits than these datatypes allow us to. One approach towards solving this is to have a linked list of numbers, with each node of the linked list representing one digit of the number. So, for example, the number 2023 could be represented in the linked list as: `[HEAD] -> 3 -> 2 -> 0 -> 2 -> [NULL]`. It is clear that we store the digits in a reversed order: LSB to MSB. Furthermore, let us restrict ourselves to positive integers only for now (though this scheme can very easily be expanded to negative numbers as well).

Create structure definitions necessary for representing such large numbers as described above. Then arbitrarily store two large numbers (each having more than 30 digits) in two such variables. Create a function add() that takes two such large numbers as inputs, and returns another such large number containing the sum of the two. Write a suitable main() function to test the same.

Cycle Detection:

In a linked list (ie, a bunch of nodes pointing to other nodes), it is entirely possible that there is a node (think of it as the last node) pointing to a node somewhere behind in the same linked list. In other words, there is at least one node that can be reached more than once by following the stream of **next** pointers. This is known as a *cycle*. If there is a cycle in a linked list, it means we cannot traverse through it the way we are used to (because there is no node pointing to NULL, which means the traversal will never terminate; we would keep looping in the cycle). Thus, there must be a way for us to detect cycles in a linked list. This idea can then be extended later to other complex data structures such as graphs (you will learn about graphs later in the course). Also, notice that a circular linked list also has a cycle, in fact, the biggest cycle possible.

The adjoining figure (Figure 5) depicts a cyclic linked list graphically. Here, the last node (-4) points back to the second node (2), creating a cycle.

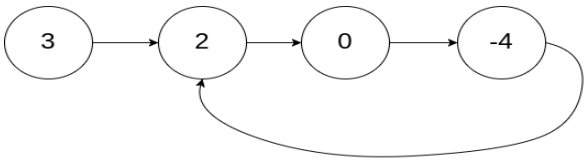

Figure 5: A cyclic linked list of integers

There are several ways of detecting cycles. The simplest approach is to include an integer attribute called *visited* in the struct of the node, and then traverse the list whilst simultaneously marking the nodes that we visit as "visited" by setting that attribute to 1 (by default the attribute should be reset to 0). This way, when we traverse, if we happen to come upon a node that we have already visited previously (it's *visited* is set to 1), then we know that there is a cycle in the linked list.

Another approach is known as "Floyd's Cycle-Detection Algorithm" or the "Hare-Tortoise Algorithm". In this approach, we traverse the linked list using two pointers. One of the pointers, called the *tortoise,* moves ahead by one node (the normal way), but the other pointer, the *hare*, moves ahead by two nodes in each traversal. If there is a cycle in the list, at some point the two pointers will intersect and be pointing to the same node. If that happens, then there is a cycle in the linked list. There is no cycle in the linked list if either of our pointers reaches the end of the linked list (either points to NULL or itself becomes NULL).

Task 9: Write a function hasCycle() that will perform the Hare-Tortoise algorithm on an input linked list and determine whether or not there is a cycle in it. Write a main() function where you test it on an acyclic linked list, a cyclic linked list, and a circular linked list. You will have to create your own linked list before calling this function. Alternatively, you can append this function to Task 7.

Efficiency Estimation:

As students of Data Structures and Algorithms, *time and space* are the most important assets to us. They are the currency of our subject. In our field, we compare the efficiency of algorithms and data structures with respect to solving specific problems depending on which takes lesser time to execute and also on which takes up lesser space in memory. Therefore it becomes imperative for us to be equipped with the tools that enable us to perform these measurements, programmatically. In this section, we take a look at time measurements. You will learn about space/memory measurements in the next labsheet (third week).

Measuring Execution Time:

There are multiple programmatic ways to measure execution time of programs. The simplest and most **precise way** is to use the gettimeofday() function from the **sys/time.h** library. The gettimeofday() function obtains the time of the day and stores it into a *timeval* struct variable (this struct is defined in the sys/time.h library). So, therefore, we can call the gettimeofday() function at the instant when we want to start measuring the time and store the value into a timeval variable, and then call the gettimeofday() function once again when we want to stop measuring the time and store this value into another timeval variable. We may get the difference between the two events in seconds by subtracting their corresponding *tv_sec* attributes. We then add the microsecond difference to that time difference to obtain the time elapsed between our start and end times with microsecond-level precision.

A standard template for doing the above measurement in C is provided below for your convenience.

```c
#include <sys/time.h>

struct timeval t1, t2;
double time_taken;

gettimeofday(&t1, NULL);
// Perform the tasks whose execution time is to be measured
gettimeofday(&t2, NULL);

time_taken = (t2.tv_sec - t1.tv_sec) * 1e6;
time_taken = (time_taken + (t2.tv_usec - t1.tv_usec)) * 1e-6;

printf("The tasks took %f seconds to execute\n", time_taken);
```

Task 10: Create a structure for holding the following two attributes about a student: ID (string), and CGPA (float). Observe that you are given a text file named "data.txt" that holds the above information for 10,000 students in a comma-separated form (you may open the file and check the data yourself).

10a: Read the file and store the data that you read into first a dynamic array of structs and then a linked list of structs. Separately measure the times for populating the above two different data structures. Compare the time efficiency for both data structures.

10b: Now, prompt the user to enter ten new entries to be entered into the records, but at specific locations (also mentioned by the user). Then, insert those ten records into the two data structures separately and measure the time taken by them for the same. Compare the time efficiency for the above task as well for both data structures. **[Note: Do not include user input time into the time measurements.]**

10c: Prompt the user to enter a position (within the valid range, ideally from somewhere in the middle) from which to retrieve data from your data structure. Then, proceed to retrieve it and display the retrieved data on the screen. Measure the time taken to do the above for both structures and then compare their efficiency. **[Note: Do not include user input time into the time measurements; also do not include time taken to print the retrieved data.]**

10d: Finally, delete every entry from both of the above data structures one by one (but separately for both data structures), and measure the time taken to do the same for both. Compare this time efficiency as well.

Note that you are expected to create helper functions to perform the above micro tasks (insertion at the end, insertion at given position, retrieve, deletion from the end, etc). Also, make sure that you are not interleaving the tasks between the two data structures, ie., you must perform an entire task for one data structure before attempting it for the other. Otherwise, you will not be able to measure and compare the time taken by both. You should also create a suitable display function to help you verify what you are doing at each stage is correct.