CHAPTER 3

Linked Lists, Stacks and Queues

III.1. Introduction

If your program needs to store a few items - numbers, payroll records, or job descriptions, for example - the simplest and most efficient approach is to place them in a list. The array is one of the most popular representations, offering a convenient way of storing collections of items, and whose loops and iterations allow us to process these items sequentially. However, arrays are not always the most efficient way to store collections of items. In this chapter, we shall see that lists may be a better way to store collections of items, and how recursion may be used to process them.

This chapter describes list representations in general, as well as two important linked list structures: the stack and the queue.

III.2. Pointers

Pointers play a fundamental role in many algorithms and programming languages, particularly in C (C++). In algorithms, pointers enable data to be accessed and manipulated efficiently by referring to their memory address rather than to direct values. A pointer is a variable that contains the memory address of another variable.

Therefore, when we manipulate a pointer in C (C++), we have direct control over the computer's memory. This provides considerable flexibility and power in the design of efficient algorithms. Using pointers, programmers can dynamically allocate and free memory, pass data addresses between functions, and implement complex data structures such as linked lists and trees.

Definition III.1

A pointer is a variable that stores the memory address of another variable. It allows indirect access to the value stored at that address. A pointer is defined by its :

- name,
- pointed type and
- memory location reserved for a specific address (Value).

A pointer to a type T is a variable that contains the address of a **variable of type T**.

Declaring a pointer

The syntax of pointers is similar to the variable declaration in C, but we use the **(*) dereferencing operator** in the pointer declaration.

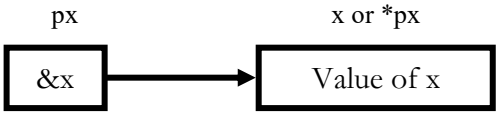
```
datatype *px;
```

Where :

- **px** is the name of the pointer.
- **datatype** is the type of data it is pointing to.

Example :

```
#include <stdio.h>
      int main(void) {
      int x;
      int *px;    //pointer to int
      x = 7;
      px = &x;
      printf(" x = %d\n", x);
      printf(" px = %p\n", px);
      printf("*px = %d\n", *px);
      return 0;
}
```



- If $\boldsymbol{x}$ is an integer: $\&\boldsymbol{x}$ is a pointer to $\boldsymbol{x}$ (memory address of $\boldsymbol{x}$)
- If $\boldsymbol{px}$ is a pointer to an integer: $*\,\boldsymbol{px}$ is the integer

III.3. Linked-Lists

The linked list is a linear data structure, in which data elements are not stored in contiguous memory locations, as is the case with arrays. The elements of the linked list are connected by pointers, which serve as a link to the next data element. Each element of the linked list is a separate object called a node. Each node contains data and points to the next node, creating a sequence. Linked list is very common data structure that is used to create tree, graph and other abstract data types.

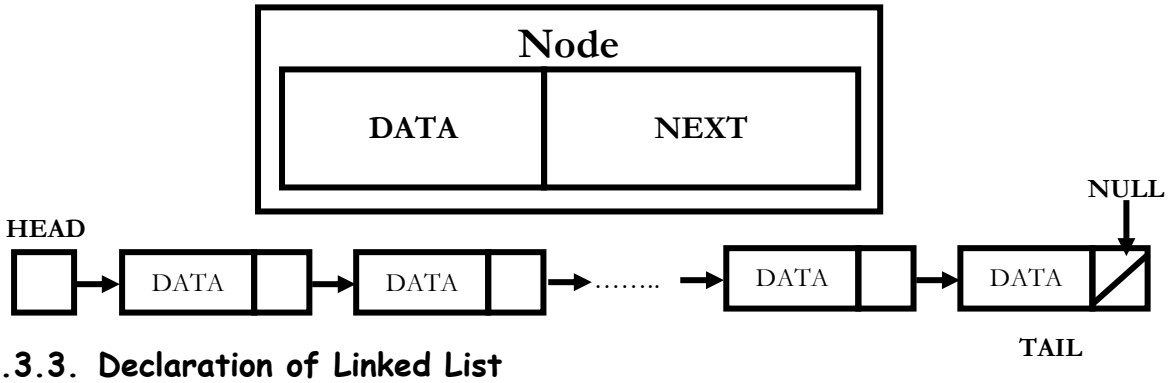
III.3.1. Properties of Linked List

- A linked list is a **linear collection of data-elements**, called **"nodes".** The linear order is maintained by pointers.
- The linked list starts with a **HEAD** which denotes the starting point or the memory location of the first node, the end of the list is called the **TAIL**.
- The Linked-list ends with the last node pointing to **NULL** value.
- It can be visualized as a chain of nodes where each node contains the location of the next node.
- Unlike arrays, linked list elements are not stored at contiguous memory locations.
- Linked lists address some of the limitations of arrays that have a fixed size because Linked lists link dynamically.
- Linked List does not waste memory space.

III.3.2. Representation of Linked List

A linked list is a chain of nodes, and each node has the following parts:

- **Data** : Stores the information
- **Next** : Stores the address to next node



III.3.3. Declaration of Linked List

Declaration of the type of elements or nodes

We first define a node with two components: (1) a place for storing data, and (2) a pointer to the next node in the chain. We define a linked list in a struct using the below code:

```
struct Node {
      int data;
      struct Node* next;
};
typedef Node* LIST;
```

Now, we might have a variable of type **LIST** called **List**:

```
LIST List;
```

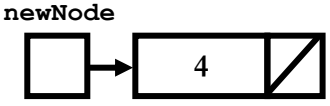
Now, we might have a variable of type **Node*** called **newNode**:

```
Node* newNode;
```

III.3.4. Creating a Node in a Linked List

The creation of a linked list starts with creating a node. Sufficient memory must be allocated when a node is created. Information is stored in memory using the **malloc()** or **new()** functions.

```
struct Node * newNode;
newNode = new(Node);
newNode -> data = 4;
newNode -> next = NULL;
```



- The above definition is used to create every node in the list.
- The **data** field stores the element and the **next** is a pointer to store the address of the next node.
- We can only move in the direction of the link. That is, we can move from a node to the node next to it but we cannot go back to the previous node.

III.3.5. Implementing the Basic linked list Operations

There are several operations on linked lists that allow us to perform different actions on linked lists. The following operations perform the required action on a linked list:

A. Constructor

The constructor simply initializes the **head (List)** pointer to be equal to **NULL** to indicate an initially empty list (it does not refer to any node).

```
List = NULL;      //We can use nullptr instead of NULL
```

B. Empty

We can then perform the second list operation, determining whether a list is empty, simply by checking whether the **head (List)** is **NULL**:

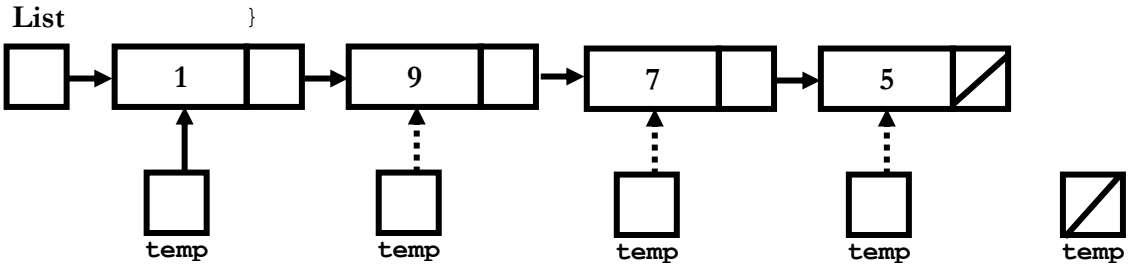
```
List  == NULL;    //We can use nullptr instead of NULL
```

C. Traverse

The third basic list operation is a traversal of the list. Traversing a linked list is very simple, and means going through the linked list by following the links from one node to the next.

To traverse a linked list, we begin by initializing an auxiliary variable **temp** that points to the first node: We keep moving the **temp** node to the next one and display its contents. When **temp** is **NULL**, we know that we have reached the end of the linked list so we get out of the **while** loop.

```
struct node *temp = List;
while(temp != NULL) {
  temp = temp -> next;
}
```



D. Displaying a Linked List

To display the linked list elements, the processing step in this algorithm is simply to output the **data** part of the node.

```
void display(LIST List) {
      cout << "\n\n List elements are : " << endl;
      while(List != NULL) {
            cout << List -> data << "    ";
            List = List -> next;
      }
}
```

E. Insert a Node in a Linked List

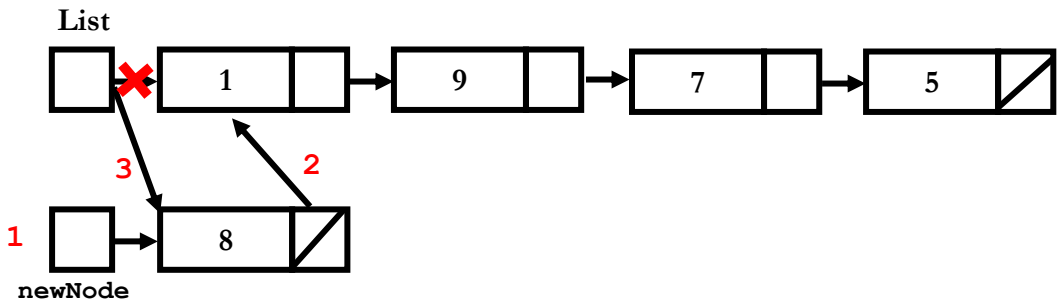
You can add elements to either the beginning, middle or end of the linked list.

There can be three cases that will occur when we are inserting a node in a linked list.

- Insertion at the beginning.
- Insertion at the end (Append).
- Insertion after a given node.

Insertion at the beginning

Since, there is no need to find the end of the list. If the list is empty, we make the new node as the head of the list. Otherwise, we have to connect the new node to the current head of the list and make the new node, the head of the list.

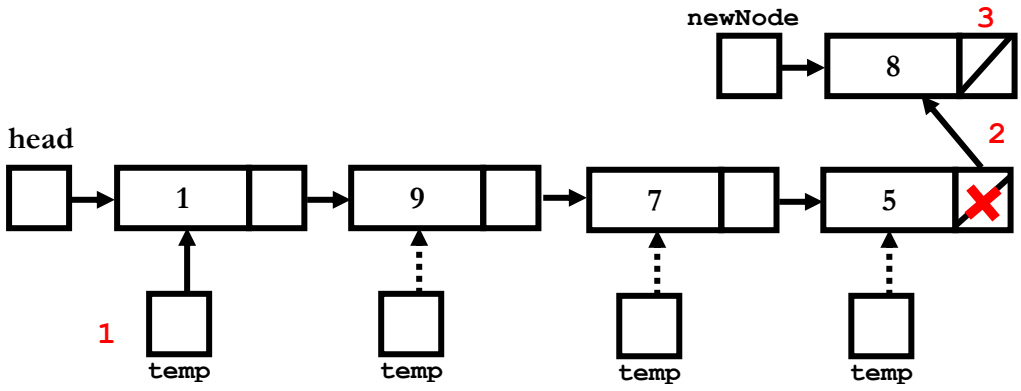



```
void insertAtBegin(LIST &List, int val){
      Node* newNode = new Node;     // creating new node of linked list
      newNode -> data = val;
      newNode -> next = List;
      List = newNode;
}
```

Insertion at the end

We will traverse the list until we find the last node. Then we insert the new node to the end of the list. Note that we have to consider special cases such as list being empty.

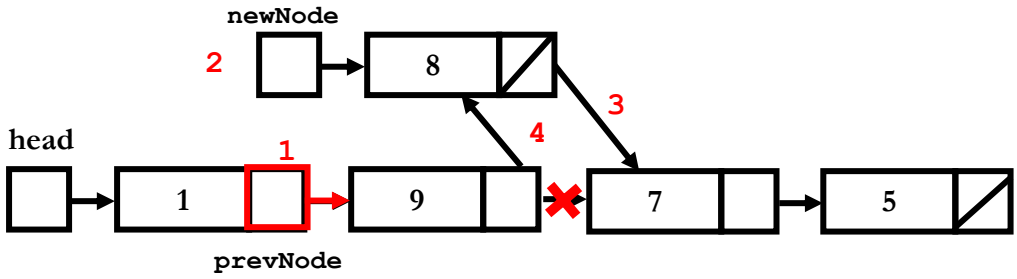
In case of a list being empty, we will return the updated head of the linked list because in this case, the inserted node is the first as well as the last node of the linked list.



```
void insertAtEnd(LIST &List, int val){
    Node* newNode = new (Node);     // creating new node of linked list
    newNode -> data = val;
    if( List == NULL ) {                    // handing the special case
        newNode -> next = List;
        List = newNode;
        return ;
    }
    LIST temp = List;
    while( temp -> next != NULL ){ //traversing the list to get the last node
        temp = temp -> next;
    }
    temp -> next = newNode;
    newNode -> next = NULL;
}
```

Insertion after a given node

We are given the reference to a node, and the new node is inserted after the given node.



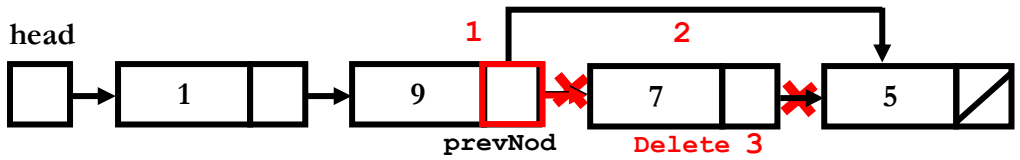

```
void insertAfter(Node* prevNode, int val){
    Node* newNode = new (Node);     // creating new node of linked list
    newNode -> data = val;
    newNode -> next = prevNode -> next;
    prevNode -> next = newNode;
}
```

F. Delete a node in a linked list

To delete a node from a linked list, we need to do these steps

- Find the previous node of the node to be deleted.
- Change the next pointer of the previous node
- Free the memory of the deleted node.

In the deletion, there is a special case in which the first node is deleted. In this, we need to update the head of the linked list.




```
void deleteLL(LIST &List, Node* del){
      if(List == del){          // if the node to be deleted is the head node
            List = List -> next;    // special case for the first Node
            delete del;
            return;
      }
      LIST temp = List;
      while( temp -> next != NULL ){
            if(temp -> next == del){        // finding the node to be deleted
                  temp -> next = temp->next->next;
                  delete del;               // free the memory of that Node
                  return;
            }
            temp = temp -> next;
      }
}
```

G. Linked List node Searching

To search any value in the linked list, we can traverse the linked list and compares the value present in the node.

```
bool searchLL(LIST List, int val){
      LIST temp = List;
      while( temp != NULL){                    // traversing the list
            if( temp -> data == val )
                  return true;
            temp = temp -> next;
      }
      return false;
}
```

H. Updating a Linked List Node

To update the value of the node, we just need to set the data part to the new value.

Below is the implementation in which we had to update the value of the first node in which data is equal to val and we have to set it to newVal .

```
void updateLL(LIST &List, int val, int newVal){
      LIST temp = List;
      while(temp != NULL){
            if( temp -> data == val){
                  temp -> data = newVal;
                  return ;
            }
            temp = temp -> next;
      }
}
```

III.3.6. Type of Linked List

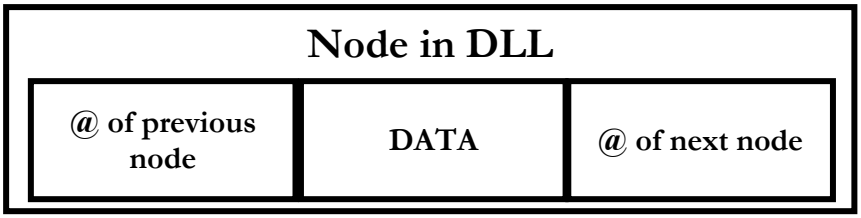
Basically, we can put linked lists into the following three items:

Singly Linked List

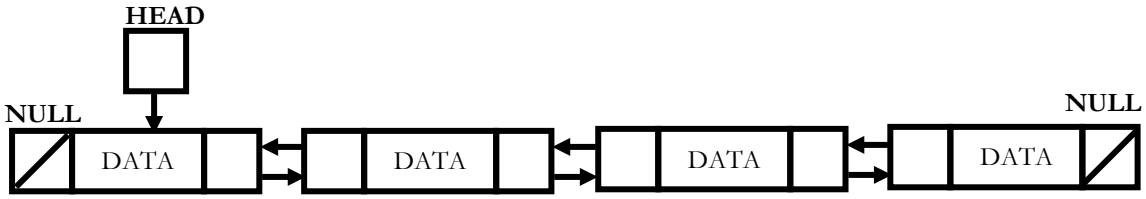
A Singly-linked list is a collection of nodes linked together in a sequential way where each node of the singly linked list contains a data field and an address field that contains the reference of the next node. Elements are traversed in the forward direction.

Doubly Linked List

A Doubly Linked List contains an extra memory to store the address of the previous node, together with the address of the next node and data which are there in the singly linked list. So, here we are storing the address of the next as well as the previous nodes. This pointer makes it easy to navigate the list in both directions and thus simplifies the process of deleting or inserting an element before the selected one.



The nodes are connected to each other in this form where the first node has **prev = NULL** and the last node has **next = NULL** .



We define a linked list in a struct using the below code:

```
struct DLLNode {
      int data;
```

```
        struct DLLNode* prev;
        struct DLLNode* next
};
typedef DLLNode* DLList;
```

Advantages over Singly Linked List-

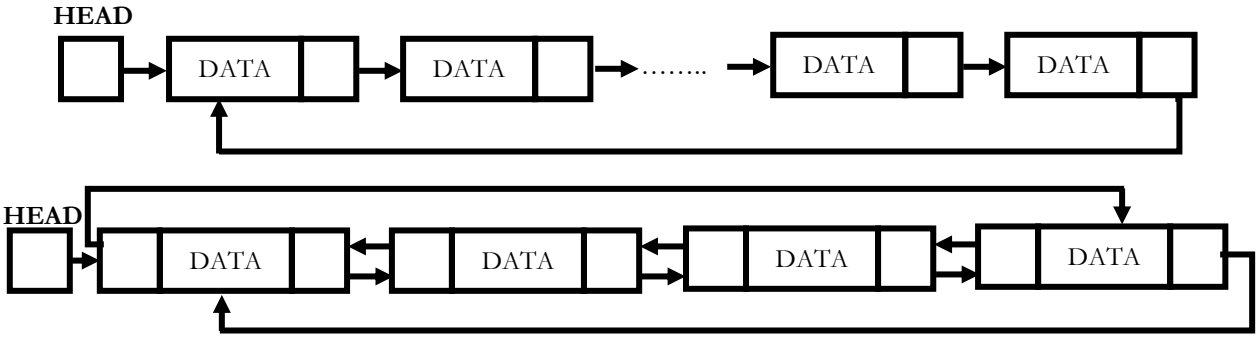
- It can be traversed both forward and backward direction.
- The delete operation is more efficient if the node to be deleted is given.
- The insert operation is more efficient if the node is given before which insertion should take place.

Disadvantages over Singly-Linked List

- It will require more space as each node has an extra memory to store the address of the previous node.
- The number of modifications increase while doing various operations like insertion, deletion, etc.

Circular Linked List

A circular linked list is either a singly or doubly linked list in which there are no NULL values. Here, we can implement the Circular Linked List by making the use of Singly or Doubly Linked List. In the case of a singly linked list, the next of the last node contains the address of the first node and in case of a doubly-linked list, the next of last node contains the address of the first node and previous of the first node contains the address of the last node.



Advantages of a Circular linked list

- The list can be traversed from any node.
- Circular lists are the required data structure when we want a list to be accessed in a circle or loop.
- We can easily traverse to its previous node in a circular linked list, which is not possible in a singly linked list.

Disadvantages of Circular linked list

- If not traversed carefully, then we could end up in an infinite loop because here we don't have any ***NULL*** value to stop the traversal.

- Operations in a circular linked list are complex as compared to a singly linked list and doubly linked list like reversing a circular linked list, etc.

III.3.7. Linked Lists Versus Arrays

In general, the main advantage of arrays over linked list is that arrays allow random access to any element in $\mathcal{O}(1)$ and whereas linked-lists allow only sequential access. Hence, in an array search can be performed on sorted data in $\mathcal{O}(log\ n)$ whereas it takes $\mathcal{O}(n)$ in the case of a linked list. In almost all other cases linked-list is a better choice.

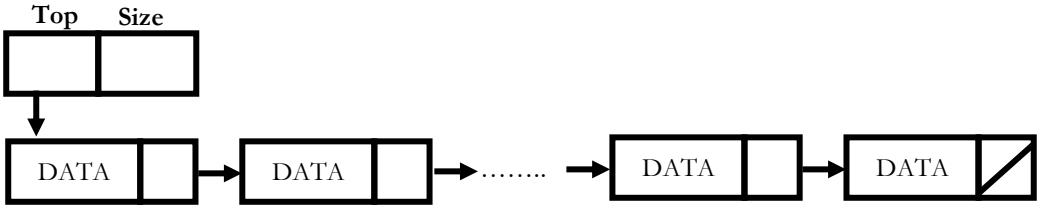
The following tables summarize tradeoffs between arrays and linked-lists:

	Array	Linked-list
Indexing	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Insert/delete at end	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert/delete at start	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Insert/delete in the middle (insertion point given)	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Locality (affects cache hit ratio)	great	poor
Size limitations	Yes (for fixed arrays)	no
Resizing	expensive	built-in
memory wastage	yes	no
extra storage needed for storing addresses	no	yes
allocation/deallocation for each element	no	yes
tail sharing	no	yes (singly-linked list only)

III.4. Stacks

Stack is a data structure that can be used to store data which can later be retrieved in the reverse or LIFO (Last In First Out) or FILO (First In Last Out) order. Stack is an ordered-list in which all the insertions and deletions are made at the same end to maintain the LIFO order called the top. Real-life examples of a stack are a deck of cards, piles of books, piles of money, and many more.

Most microprocessors use a stack-based architecture. When a function is called, its return address and arguments are pushed onto a stack and when that function terminates, those values are popped off from the stack. These stack operations are built into microprocessors. Note that one danger in recursive programming is that the stack that keeps return address and arguments could overflow if too many recursive calls are made.



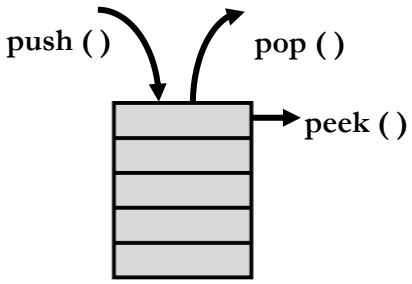
III.4.1. Declaration

The creation of a structure with a "top" field representing the add or delete position, and a "size" field representing the stack size.

```
struct Node {
    int data;
    struct Node* next;
};
typedef struct  {
    struct Node *top;
    int size;
} STACK;
```

Now, we might have a variable of type **STACK** called **Stack**: STACK Stack;

III.4.2. Implementing Stack Operations



The operations defined on a stack are:

- **isEmpty** : Check if the stack is empty
- **isFull** : Check if the stack is full
- **Size**: Returns the number of elements in the stack.
- **push** : Adds an element to the top of the stack
- **peek** : Examine the top element in the stack
- **pop**: Removes and returns the element at the top of the stack

```
// Function to initialize the Stack
STACK init_Stack(){
    STACK Stack;
    Stack.top = NULL;
    Stack.size = 0;
    return Stack;
}
// Function to check if the Stack is empty or not
bool isEmpty(STACK Stack) {
    return Stack.top == NULL;
}
// Function to check if the Stack is full or not
bool isFull(){
    Node *newNode = new Node;                //create new node
    if(newNode == NULL){
        cout << "\nStack is Full " << endl;
        return true;
    }
```

```
            else{
                delete newNode;
                return false;
            }
        }
//Function to return Stack Size
        int Size(STACK Stack) {
            return Stack.size;
        }
// Function to push an element onto the Stack
        void push(STACK &Stack, int val){
            if (isFull()){
                return;
            }
            Node *newNode = new Node;                //create new node
            newNode -> data = val;
            newNode -> next = Stack.top;
            Stack.top = newNode;
            Stack.size++;
        }
// Function to return the top element of the Stack
        int peek(STACK Stack) {
            if (!isEmpty(Stack))
                return Stack.top -> data;
            else {
                cout << "Stack is Empty" << endl;
                return -1;
            }
        }
// Function to remove the top element from the Stack
        int pop(STACK &Stack){
            if (isEmpty(Stack)) {
                cout << "Stack is Empty" << endl;
                return -1;
            }
            else{
```

```
                    Node* newNode;
                    int popped = peek(Stack);
                    cout << "Popped Element = " << popped << endl;
                    newNode = Stack.top;
                    Stack.top = Stack.top -> next;
                    Stack.size--;
                    delete newNode ;        //free the popped element's memory
                    return popped;
                }
            }
// Display all elements of the Stack
            void display(STACK Stack){
                Node *nowNode = Stack.top;
                while(nowNode != NULL){
                     cout << nowNode -> data << "-->";
                     nowNode = nowNode -> next;
                }
                cout << "NULL" << endl;
            }
```

III.4.3. Stack Complexity Analysis

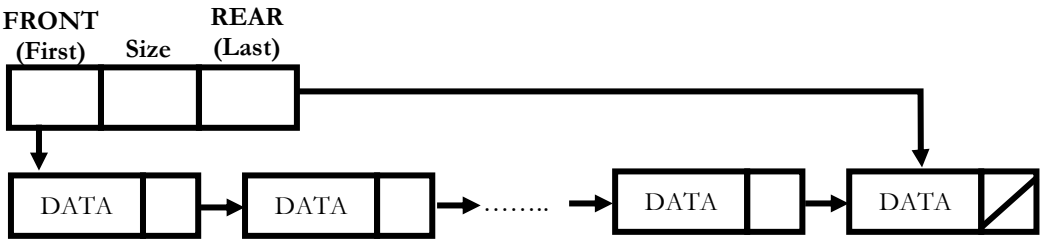
Function	Time Complexity	Auxiliary Space Complexity
isEmpty()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
isFull()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Size()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
peek()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
push()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
pop()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
display()	$\mathcal{O}(n)$	$\mathcal{O}(1)$

III.5. Queues

The queue is also a list, with insertion at one end and deletion at the other. In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'. It's also called as FIFO (First In First Out) or FCFS (first-come-first-served) basis which means the element, meaning that the element inserted first will be removed first from the queue.

As in the case of the queue of people at the ticket window where newcomers join the queue at the tail or rear end of the queue and service is provided to the one at the front or head of the queue,

in the queue data structure insertions and deletions are made at two different ends (compare it with stack where these two operations are performed at the same end) to maintain the FIFO order.



III.5.1. Declaration

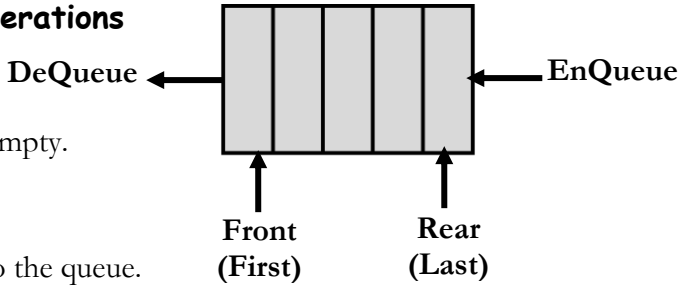
The creation of a structure with a "top" field representing the add or delete position, and a "size" field representing the queue size.

```
struct Node {
    int data;
    struct Node* next;
};
typedef struct  {
    struct Node *first, *last;
    int size;
} QUEUE;
```

Now, we might have a variable of type **QUEUE** called **Queue**: QUEUE Queue;

III.5.2. Implementing Queue Operations

The basic operations on a queue are :



- **isEmpty:** Checks if the queue is empty.
- **Is_Full:** check if the Queue is full
- **EnQueue:** Adds a new element to the queue.
- **DeQueue:** Removes and returns the first (front) element from the queue.
- **Size:** Finds the number of elements in the queue.

As can be easily seen, the list of operations supported by a queue is pretty similar to the ones supported by the Stack.

```
// Function to initialize the Queue
QUEUE init_Queue(){
    QUEUE Queue;
    Queue.first = NULL;             //Front
    Queue.last = NULL;              //Rear
    Queue.size = 0;
    return Queue;
}
```

```
// Function to check if the stack is empty or not
            bool isEmpty(QUEUE Queue){
                return Queue.first == NULL;
            }
// Function to check if the Queue is full or not
            bool isFull(){
                Node *newNode = new Node;
                if (newNode == NULL){                    //When heap exhausted
                    cout << "Queue is Full" << endl;
                    return true;
                }
                delete newNode;
                return false;
            }
//Function to return Queue Size
            int Size(QUEUE Queue) {
                return Queue.size;
            }
// Function to add an element to the Queue
            void EnQueue(QUEUE &Queue, int val) {
                if (isFull()){
                    return;
                }
                Node *newNode = new Node;
                newNode -> data = val;
                newNode -> next = NULL;
                if(isEmpty(Queue)){
                    Queue.first = newNode;
                    Queue.last = newNode;
                }
                else{
                    Queue.last -> next = newNode ;
                    Queue.last = newNode;
                }
                Queue.size++;
            }
```

```
// Function to remove an element from the Queue
            int DeQueue(QUEUE &Queue) {
                if(isEmpty(Queue)){
                    cout<< "Queue is Empty" << endl;
                    return 0;
                }
                int data = Queue.first -> data;
                Cout << Queue.first -> data << " Removed From Queue" << endl;
                Node *newNode = Queue.first;
                if (Queue.first == Queue.last){
                    Queue.first = Queue.last = NULL;
                }
                else {
                    Queue.first = Queue.first -> next;
                }
                Queue.size--;
                delete newNode;
                return data;
            }
// Display all elements of the Queue
            void display(QUEUE Queue){
                Node *newNode = Queue.first;
                while(newNode != NULL){
                     cout << newNode -> data << "-->";
                     newNode = newNode -> next;
                }
                cout << "NULL" << endl;
            }
```

III.5.3. Queue Complexity Analysis

Function	Time Complexity	Auxiliary Space Complexity
isEmpty()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
isFull()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Size()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
EnQueue()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
DeQueue()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
display()	$\mathcal{O}(n)$	$\mathcal{O}(1)$