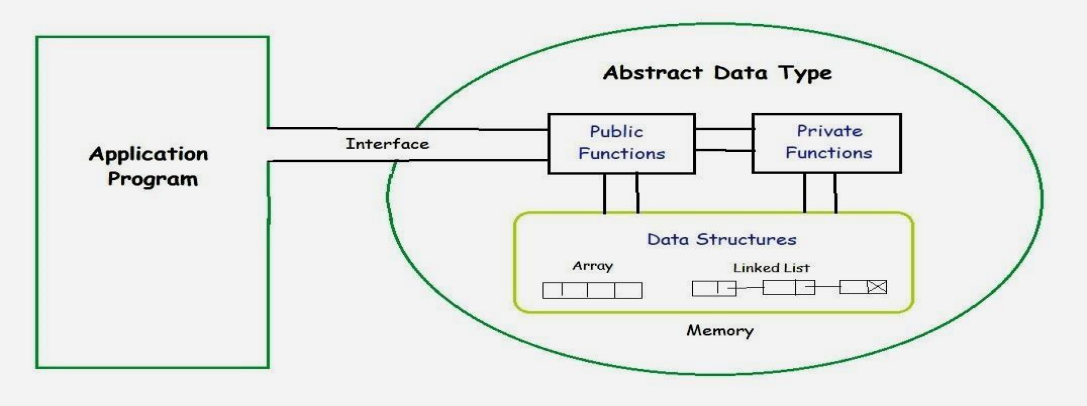
UNIT -1

Abstract Data Types

Data types such as int, float, double, long, etc. are considered to be in-built data types and we can perform basic operations with them such as addition, subtraction, division, multiplication, etc. Now there might be a situation when we need operations for our user-defined data type which have to be defined. These operations can be defined only as and whenwe require them. So, in order to simplify the process of solving problems, we can create data structures along with their operations, and such data structures that are not in-built are known as **Abstract Data Type (ADT).**

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view.

The process of providing only the essentials and hiding the details is known as **abstraction**.



The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.

So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we"ll define three ADTs namely List ADT, Stack ADT, Queue ADT.

List ADT

The data is generally stored in key sequence in a list.

The data node contains the pointer to a data structure and a self-referential pointer which points to the next node in the list.

The **List ADT Functions** is given below:

- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace() – Replace an element at any position by another element.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise return false.

- isFull() – Return true if the list is full, otherwise return false.

Stack ADT

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.

The program allocates memory for the data and address is passed to the stack ADT.

The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.

The stack head structure also contains a pointer to top and count of number of entries currently in stack.

The **Stack ADT Functions** is given below:

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false.
- isFull() – Return true if the stack is full, otherwise return false.

Queue ADT

The queue abstract data type (ADT) follows the basic design of the stack abstract data type.

Each node contains a void pointer to the data and the link pointer to the next element in the queue. The program‟s responsibility is to allocate memory for storing the data.

The **Queue ADT Functions** is given below:

- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- peek() – Return the element of the queue without removing it, if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull() – Return true if the queue is full, otherwise return false.

Features of ADT:

Abstraction:

The user does not need to know the implementation of the data structure.

Better Conceptualization:

ADT gives us a better conceptualization of the real world.

Robust:

The program is robust and has the ability to catch errors.

From these definitions, we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.

Data Structure:

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data.

There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed.

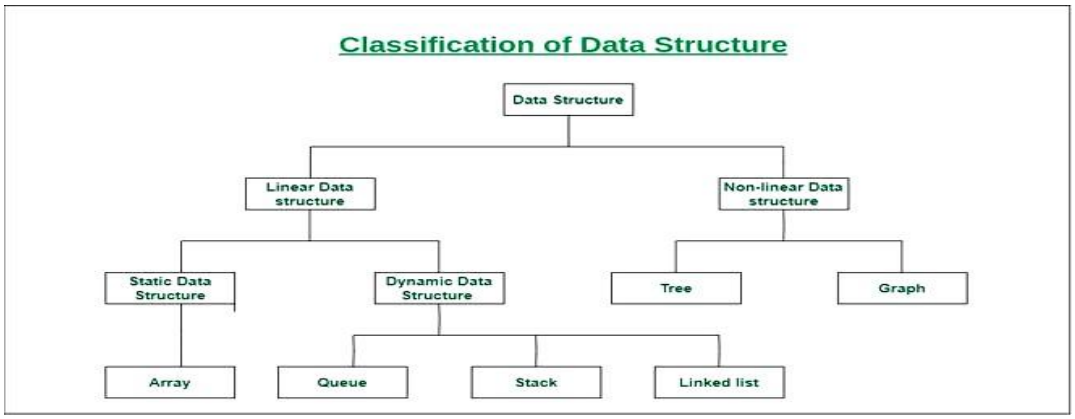
How Data Structure varies from Data Type:

People get confused between data type and data structure. So let‟s see a few differences between data type and data structure to make it clear.

Data Type	Data Structure
The data type is the form of a variable to which a value can be assigned. It defines that the particular variable will assign the values of the given data type only.	Data structure is a collection of different kinds of data. That entire data can be represented using an object and can be used throughout the program.
It can hold value but not data. Therefore, it is dataless.	It can hold multiple types of data within a single object.
The implementation of a data type is known as abstract implementation.	Data structure implementation is known as concrete implementation.
There is no time complexity in the case of data types.	In data structure objects, time complexity plays an important role.
In the case of data types, the value of data is not stored because it only represents the type ofdata that can be stored.	While in the case of data structures, the data and its value acquire the space in the computer‟s main memory.
Data type examples are int, float, double, etc.	Data structure examples are stack, queue, tree, etc.

Classification of Data Structure

Data structure has many different uses in our daily life. There are many different data structures that are used to solve different mathematical and logical problems. By using data structure, one can organize and process a very large amount of data in a relatively short period. Let‟s look at different data structures that are used in different situations.



Linear data structure:

Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent element, is called a linear data structure.

Linear data structure, single level is involved. Therefore, we can traverse all the elements in single run only. Linear data structures are easy to implement because computer memory is arranged in a linear way.

Examples of linear data structures are array, stack, queue, linked list, etc.

(i) Static data structure:

Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.

Example: array

Array

The array is a type of data structure that stores elements of the same type. These are the most basic and fundamental data structures. Data stored in each position of an array is given a positive value called the index of the element. The index helps in identifying the location of the elements in an array.

If supposedly we have to store some data i.e. the price of ten cars, then we can create a structure of an array and store all the integers together. This doesn‟t need creating ten separate integer variables. Therefore, the lines in a code are reduced and memory is saved. The index value starts with 0 for the first element in the case of an array.

(ii) Dynamic data structure:

In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.

Examples: Linked List, queue, stack

Stack

The data structure follows the rule of **LIFO (Last In-First Out)** where the data last added element is removed first. Push operation is used for adding an element of data on a stack and the pop operation is used for deleting the data from the stack.

This can be explained by the example of books stacked together. In order to access the last book, all the books placed on top of the last book have to be safely removed.

Queue

This structure is almost similar to the stack as the data is stored sequentially. The difference is that the queue data structure follows **FIFO** which is the rule of **First In-First Out** where the first added element is to exit the queue first. Front and rear are the two terms to be used in a queue.

Enqueue is the insertion operation and dequeue is the deletion operation. Enqueue is performed at the rear and dequeue is performed at the front of the queue.

The data structure might be explained with the example of people queuing up to get a ticket. The first person in the line will get the chance to exit the queue while the last person will be the last to exit.

Linked List

Linked lists are the types where the data is stored in the form of nodes which consist of an element of data and a pointer. The use of the pointer is that it points or directs to the node which is next to the element in the sequence.

The data stored in a linked list might be of any form, strings, numbers, or characters. Both sorted and unsorted data can be stored in a linked list along with unique or duplicate elements.

Non-linear data structure:

Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can‟t traverse all the elements in a single run.

Non-linear data structures are not easy to implement in comparison to linear data structure. It utilizes computer memory efficiently in comparison to a linear data structure.

Examples: trees and graphs.

Trees

A tree data structure consists of various nodes linked together. The structure of a tree is hierarchical that forms a relationship like that of the parent and a child. The structure of the tree is formed in a way that there is one connection for every parent-child node relationship. Only one path should exist between the root to a node in the tree.

Various types of trees are present based on their structures like AVL tree, binary tree, binary search tree, etc.

Graph

Graphs are those types of non-linear data structures which consist of a definite quantity of vertices and edges. The vertices or the nodes are involved in storing data and the edges show the vertices relationship.

The difference between a graph to a tree is that in a graph there are no specific rules for the connection of nodes.

Real-life problems like social networks, telephone networks, etc. can be represented through the graphs.

Hash Tables

These types can be implemented as linear or non-linear data structures. The data structures consist of key-value pairs.

Difference between Linear and Non-linear Data Structures:

S. No.	Linear Data Structure	Non-linear Data Structure
1.	Data elements are arranged in a linear order where each and every element is attached to its previous and next adjacent.	Data elements are attached in hierarchically manner.
2.	Single level is involved.	Multiple levels are involved.
3.	Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
4.	Data elements can be traversed in a single run only.	Data elements can‟t be traversed in a single run only.
5.	Memory is not utilized in an efficient way.	Memory is utilized in an efficient way.
6.	Examples : array, stack, queue, linked list	Examples : trees and graphs.
7.	Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

Need of Data Structure:

The structure of the data and the synthesis of the algorithm are relative to each other. Data presentation must be easy to understand so the developer, as well as the user, can make an efficient implementation of the operation.

Data structures provide an easy way of organizing, retrieving, managing, and storing data.

Here is a list of the needs for data structure.

- Data structure modification is easy.
- It requires less time.
- Save storage memory space.
- Data representation is easy.
- Easy access to the large database.

Advantages of Linked Lists over arrays:

- Dynamic Memory Allocation.
- Ease of Insertion/Deletion.

Drawbacks of Linked Lists:

- Random access is not allowed. We have to access elements sequentially starting from the first node (head node).
- Extra memory space for a pointer is required with each element of the list.
- Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Types of Linked Lists:

1. **Singly Linked List**
   In this type of linked list, one can move or traverse the linked list in only one direction.

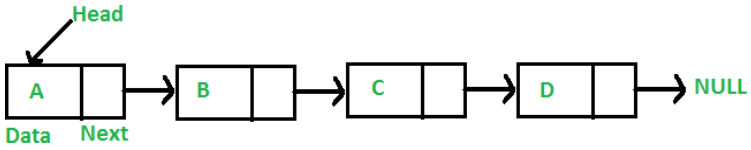
2. **Doubly Linked List**
   In this type of linked list, one can move or traverse the linked list in both directions (Forward and Backward)

3. **Circular Linked List**
   In this type of linked list, the last node of the linked list contains the link of the first/head node and the first/head node contains the link of the last node.

Basic operations on Linked Lists:

- Insertion
- Deletion
- Search
- Display

Singly Linked List



A Singly linked list is a collection of data called **node**s, where each node is divided into two parts to store data and address at some random addresses. The pointer next, points to the address of the next node in a list.

Compared to the array data structure, the size of the linked list elements is not fixed. Due to this, there is efficient memory utilization in a singly linked list.

Implementing a singly linked list to perform operations like insertion and deletion is easy. Elements are accessed easily in a singly linked list.

Node is declared as follows

```
struct node
{
   int data;
   struct node *next;
  };
```

Insertion

A node can be added in three ways

- At the beginning of the linked list
- After a given node.
- At the end of the linked list.

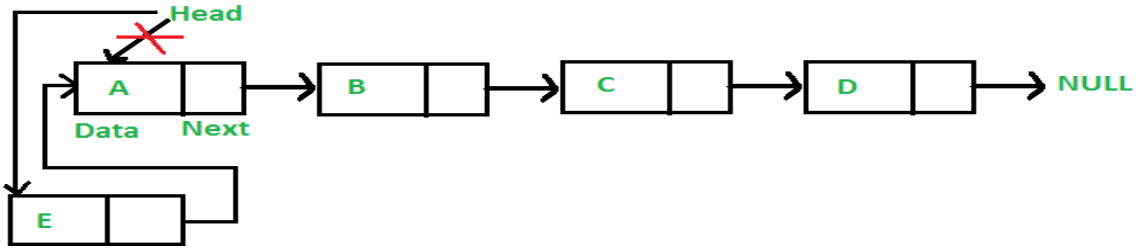
Add a node at the beginning:

Approach: The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List.

For example, if the given Linked List is A→B→C→D and we add an item E at the front, then the Linked List becomes E→A→B→C→D.

Follow the steps to add a node at beginning.

1. Allocate node
2. Put in the data
3. Make next of new node as head
4. Move the head to point new node

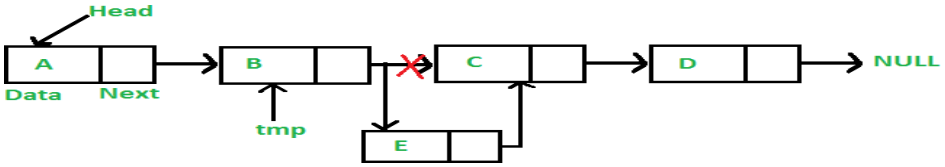



```
void insert_begin(struct node *head, int data)
{
   struct node p = malloc(sizeof(struct node));      /* 1. allocate node */
   p->data = data;                                   /* 2. put in the data */
   p->next = head;                                   /* 3. Make next of new node as head */
   head    = p;                                      /* 4. move the head to point new node */
  }
```

Add a node after a given node:

Approach: We are given a pointer to a node, and the new node is inserted after the given node. Follow the steps to add a node after a given node:

1. Allocate a new node.
2. Assign the data to the new node
3. Find the given previous node in a list.
4. And then make the next of new node as the next of given previous node.
5. Finally, move the next of the previous node as a new node.



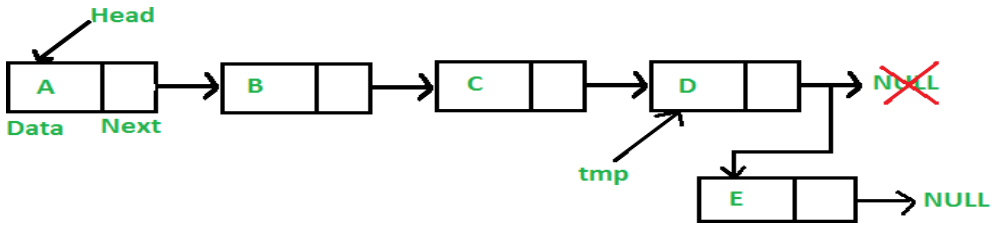
```
void insertAfter(struct node *head, int data, int x)
{
   p=malloc(sizeof(struct node));          /* 1. allocate new node */
   p->data = data;                         /* 2. put in the data */
   t=head;
   while(x!=t->data)
         t=t->next;                        /* 3. Find the given previous node */
   p->next = t->next;                      /* 4. Make next of new node as next of prev_node */
   t->next = p;                            /* 5. move the next of prev_node as new_node */
  }
```

Add a node at the end:

The new node is always added after the last node of the given Linked List.

For example if the given Linked List is A→B→C→D and we add an item **E** at the end, then the Linked List becomes A→B→C→D→E.

Since a Linked List is typically represented by the head of it, we have to traverse the list till the end and then change the next of last node to a new node.




```
void insert_end(struct node *head, int data)
{
 p=malloc(sizeof(struct node));          /* 1. allocate node */
 p->data = data;                         /* 2. put in the data */
 p->next=NULL;                           /* 3.  new node is going to be the last node */
 t=head;
 while(t->next!=NULL)
          t=t->next;                     /* 4. Else traverse till the last node */

 t->next=p;                              /* 5. Change the next of last node */
 }
```

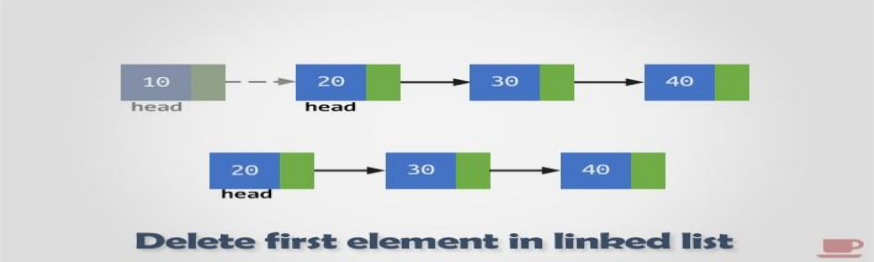
Delete from a Linked List:-

A node can be deleted in three ways

- At Beginning
- At End
- At Middle

Delete at Beginning:

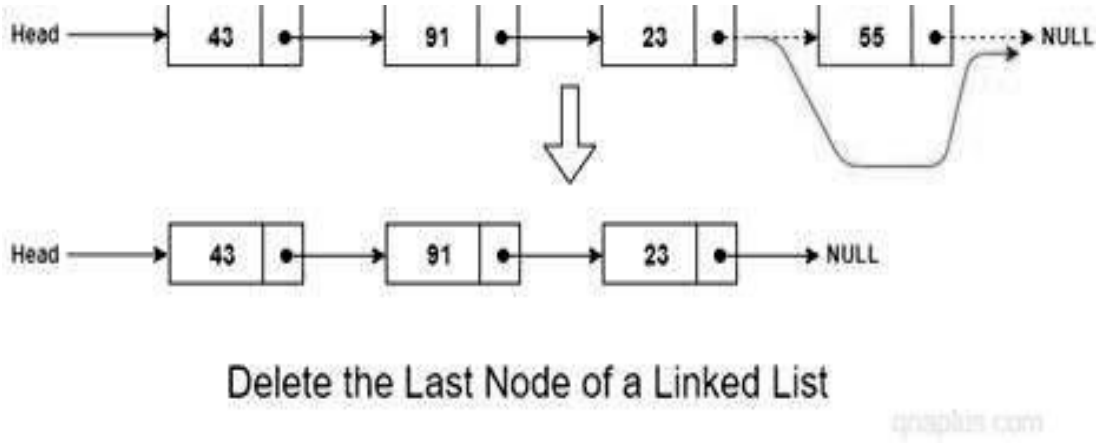
Point head to the next node i.e. second node and make sure to free unused memory by **free(t);** or **delete t;**




```
void del_begin(struct node *head)
{
   t = head;
   head = head->next;
  }
```

Delete at End:

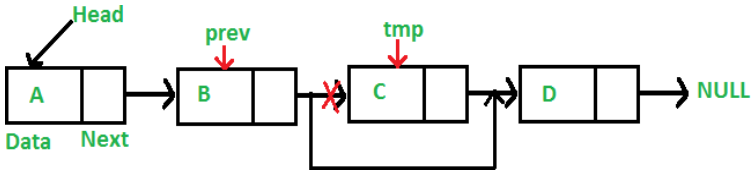
Find the last second element and change next pointer to null. Make sure to free unused memory by **free(t)** or **delete t**



```
void del_end(struct node *head)
{
 t=head;
 while(t->next!=NULL)
{
   p=t;
   t=t->next;
    }
if(t==head)
    head=NULL;
else
     p->next=NULL;
 }
```

Delete a given node:

Keeps track of pointer (p) before node to delete and pointer (t) to node to delete.
Make sure to free unused memory by **free(t)** or **delete t**




```
void del(struct node *head, int x)
{
  t=head;
  while(x!=t->data)
  {
    p=t;
    t=t->next;
    }
  if(t==head)
        head=head->next;
  else
        p->next=t->next;
  }
```

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
Step 4 - Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches NULL

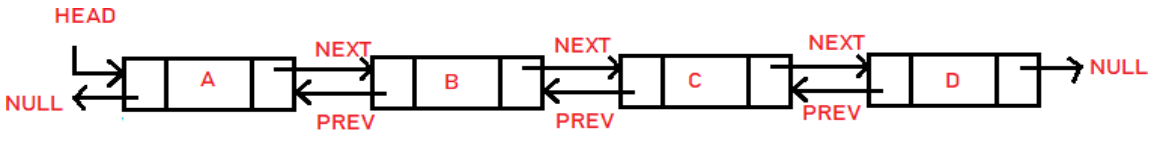
```
t=head;
while(t!=NULL)
{
   printf("%d-->",t->data);
   t=t->next;
}
printf("NULL\n");
```

Applications of Singly Linked List

- It is used to implement **stacks** and **queues** which are like fundamental needs throughout computer science.
- To prevent the collision between the data in the **hash map**, we use a singly linked list.
- If we ever noticed the functioning of a casual notepad, it also uses a singly linked list to perform undo or redo or deleting functions.
- We can think of its use in a photo viewer for having look at photos continuously in a slide show.
- In the system of train, the idea is like a singly linked list, as if you want to add a Boggie, either you have to take a new boggie to add at last or you must spot a place in between boggies and add it.

Doubly Linked List

It is a complex type of linked list in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list.



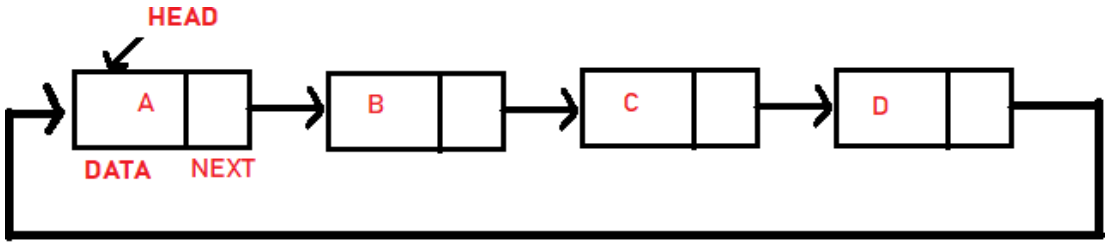
DOUBLY LINKED LIST

Applications of Doubly Linked List

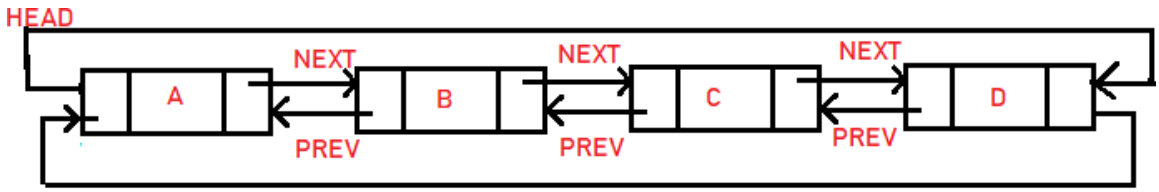
- In the browser when we want to use the back or next function to change the tab, it used the concept of a doubly-linked list here.
- Again, it is easily possible to implement other data structures like a binary tree, hash tables, stack, etc.
- It is used in music playing system where you can easily play the previous one or next one song as many times one person wants to.
- In many operating systems, the thread scheduler maintains a doubly-linked list of all the processes running at any time. This makes it easy to move a process from one queue into another queue.
- It is used in a famous game concept which is a deck of cards.

Circular Linked List

This linked list performs a circular form, as its first node points to its next node, and the last node points to the first head node forming a circle. Both singly and doubly linked lists can be made into a circular linked list.



CIRCULAR SINGLY LINKED LIST



CIRCULAR DOUBLY LINKED LIST

Applications of Circular Linked List

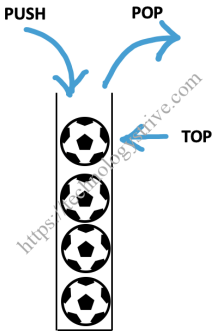
- It can also be used to implement queues by maintaining a pointer to the last inserted node and the front can always be obtained as next of last.
- Circular Doubly Linked Lists are used for the implementation of advanced data structures like Fibonacci Heap. Also reference to the previous node can easily be found in this.
- It is also used by the Operating system to share time for different users, generally uses a Round-Robin time-sharing mechanism (this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking).
- Multiplayer games use a circular list to swap between players in a loop.
- In photoshop, word, or any paint we use this concept in undo function.

STACK

Stack is a linear data structure to store and manipulate data which follows LIFO (Last-In-First-Out) order during adding and removing elements in it. There are two basic ways to implement stack data structure :

- Array based Implementation
- Linked List Based Implementation

In Array based implementation, an array of fixed size is used to create a stack.



Stack of Balls

Stack Implementation using Array

Stack implementation is defined in terms of adding an element which is called **Push** and deleting an element from the stack which is called **Pop**.

In an array, elements can be added or deleted at any position but while implementing stack, we have to permit insertions and deletions at top of the stack only. To do this we can use a variable called top which points to the last element in the stack.

Initially when the stack is empty, the value of top is initialized to -1.

For push operation, first the value of top is increased by -1 and then the new element is pushed at the position of top.

For pop operation, first the element at the position of top is popped and then top is decreased by 1.

Below are the points to aware about implementation of stack using array

- **Fixed Capacity:** A stack can hold at max of **n** elements which is given during initialization of the stack.
- **Overflow:** Once stack reaches the max capacity, it is not possible to push any new element.
- **Underflow:** When no elements are left in the stack, it is not possible to pop the elements from the stack.

Initialization

Initialize array a[] with size n which is used to store stack elements.
Initialize an index top with -1. It means no elements left in the stack.

Push Operation Algorithm

push(int data)

```
if stack is full
    print error message
else
  top = top+1
  a[top] = data
```

Pop Operation Algorithm

pop()

```
if stack is empty
   print error message
else
    print a[top]
    top = top-1
```

Traverse (Display) Operation Algorithm

display()

```
if stack is empty
   print error message
else
    print a[top] to a[0]
```

Auxiliary Stack Operations:

peek(): This operation will return the last inserted element that is at the top without removing it.

size(): This operation will return the size of the stack i.e. the total number of elements present in the stack.

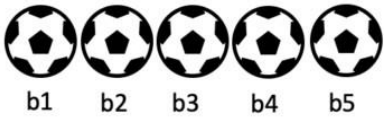
isEmpty(): This operation indicates whether the stack is empty or not.

isFull(): This operation indicates whether the stack is full or not.

Stack Operations Dry Run

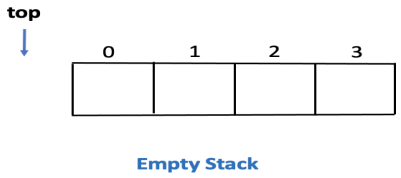
To understand push and pop operations clearly, let us perform below operations. Let us consider we have few balls and we want to place them or delete them on request basis in a container (stack).

1. push b1
2. push b2
3. push b3
4. push b4
5. push b5
6. pop

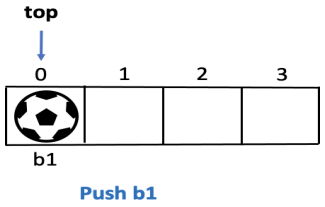


Stack Operations – Example

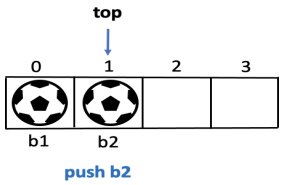
- **Initialization** :
  - ❖ Let us consider max capacity of the stack as 4
  - ❖ Initialize top index as -1



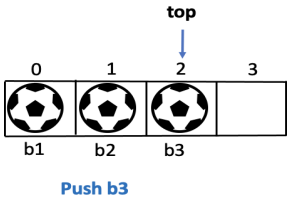
- **Push b1** :
  - ❖ Increment top by 1, it means top now points at index 0
  - ❖ store ball b1 at top index which is our top or last element



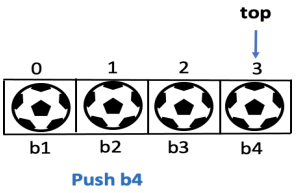
- **Push b2 :**
  - ❖ Increment top by 1 => top = 1
  - ❖ Store ball b2 at top position => b2 is the top or last element in the stack



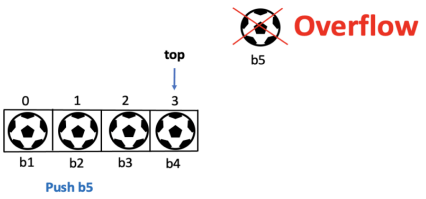
- **Push b3 :**
  - ❖ Increment top by 1 => top = 2
  - ❖ Store ball b3 at top position => b3 is last element in the stack



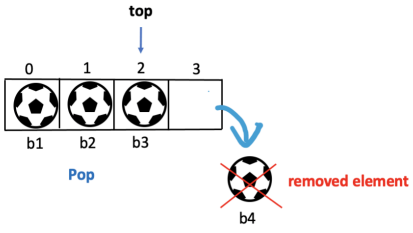
- **Push b4** :
  - ❖ Increment top by 1 => top = 3
  - ❖ Store ball b4 at top position => b4 is last element in the stack



- **Push b5 :**
  - ❖ As stack has reached it max capacity, adding new element will result in **overflow**



- **Pop :**
  - ❖ top element b4 gets removed from the stack
  - ❖ top index gets decremented by 1



isFull ( ) Operation

This function checks if the stack is full or not. If the stack is full, then the insertion of elements is not possible in a stack. This condition is known as **overflow** error. We need to perform the following steps while carrying this operation:

```
if (top == n-1)
      return 1
else
      return 0
```

isEmpty ( ) Operation

This function validates if the stack is empty. If top is equal to -1, then we can consider the queue as empty.

```
if ( top == -1)
   return 1
else
   return 0
```

Peek ( ) Operation

This function helps in extracting the data element where the top is pointing without removing it from the stack.

```
if ( isEmpty() )
   return STACK IS EMPTY
else
   return a[top]
```

Stack Implementation using Linked List

The benefit of implementing a stack using a linked list over arrays is that it allows to grow of the stack as per the requirements, i.e., memory can be allocated dynamically.

A stack is represented using nodes of a linked list. Each node consists of two parts: data and next(storing the address of the next node). The data part of each node contains the assigned value, and the next points to the node containing the next item in the stack. The top refers to the topmost node in the stack. Both the push() and pop() operations are carried out at the top of the stack.

Node Structure:

```
struct Node
{
   int data;
   struct Node *next;
};
Struct Node *top = NULL;
```

PUSH Operation

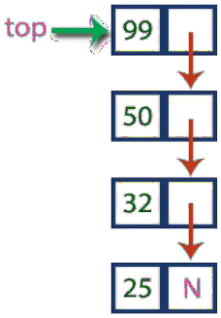
Adding or inserting a new element to a stack is known as the Push() operation in the stack. Elements can only be pushed at the top of the stack.

Steps to push an element into a Stack:

- Create a new node (p) using dynamic memory allocation and assign value to the node.
  ```
  struct Node *p = malloc(sizeof(struct Node));
  p->data = value;
  ```
- Check if stack is empty or not, i.e, (if top == NULL).
- If it is empty, then set the next pointer of the node to NULL.
  ```
  p->next = NULL;
  ```
- If it is not empty, the newly created node should be linked to the current top element of the stack, i.e.,
  ```
  p->next = top;
  ```
- Make sure that the top of the stack should always be pointing to the newly created node.
  ```
  top = p;
  ```

Algorithm for push()

```
if top is equal to NULL
  p-> next = NULL
else
  p-> next = top
```



Function for Push() :

```
struct Node  *p;

int push( value )
{
   p = malloc(sizeof(struct Node))
   p->data = value;
   if (top == NULL)
            p->next = NULL;
   else
            p->next = top;
   top = p;
   }
```

POP Operation

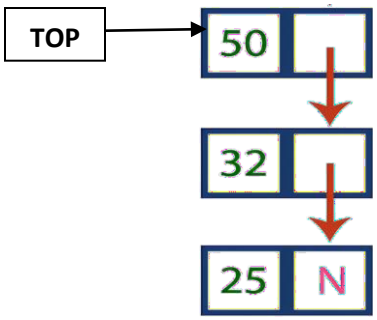
Removing or deleting an element from a stack is known as the Pop() operation in the stack. Elements are popped from the top of the stack. There should be at least one element in the stack to perform the pop() operation.

Steps to pop an element from a Stack:

- Check if stack is empty or not, i.e, (TOP == NULL).
- If it is empty, then print Stack Underflow.
- If it is not empty, then create a temporary node (t) and set it to top. And print data of temporary node ie deleted element from the stack.
  ```
  struct Node *t = top;
  print t->data;
  ```
- Now, make top point to the next node.
  ```
  top = top->next;
  ```
- Delete the temporary node.
  ```
  free(t);
  ```

Algorithm for pop( )

```
if top == NULL
    print "Stack Underflow"
else
   create temporary node (t) and assign t = top
   print the deleted value ie. top->data
   top = top->next
   free(t)
```



Function for Pop( ) :

```
struct Node *t;

int pop( )
{
   if (top == NULL)
      printf("\nEMPTY STACK");
   else
  {
      t = top;
      printf("%d", top->data);
      top = top->next;
      free(t);
    }
 }
```

Applications of Stack:

- Stack data structure is used in evaluation and conversion of arithmetic expressions.
- Stack is used in Recursion.
- It is used for parenthesis checking.
- While reversing a string, stack is used as well.
- Stack is used in memory management.
- It is also used for processing of function calls.
- The stack is used to convert expressions from infix to postfix.
- The stack is used to perform undo as well as redo operations in word processors.
- The stack is used in virtual machines like JVM.
- The stack is used in the media players. Useful to play the next and previous song.
- The stack is used in recursion operation.

Real Life Applications of Stack:

- Real life example of a stack is the layer of eating plates arranged one above the other. When you remove a plate from the pile, you can take the plate on the top of the pile. But this is exactly the plate that was added most recently to the pile. If you want the plate at the bottom of the pile, you must remove all the plates on top of it to reach it.
- Browsers uses stack data structure to keep track of previously visited sites.
- Call log in mobile also uses stack data structure.

Arithmetic Expression Evaluation using Stack

The precedence of operators needs to be taken case of:

Exponentiation (^) > Multiplication (*) or Division (/) > Addition (+) or Subtraction (-)

Brackets have the highest priority and their presence can override the precedence order.

Arithmetic expressions can be represented in 3 forms:

1. Infix notation
2. Postfix notation (Reverse Polish Notation)
3. Prefix notation (Polish Notation)

Infix Notation is of the form *operand1* **operator** *operator2*.

Eg: 5 + 3

Postfix Notation can be represented as *operand1 operand2* **operator**.

Eg: 5 3 +

Prefix notation can be represented as **operator** *operand1 operand2*.

Eg: + 5 3

Why postfix representation of the expression?

- Infix expressions are readable and solvable by humans because of easily distinguishable order of operators, but compiler doesn't have integrated order of operators.
- Hence to solve the Infix Expression compiler will scan the expression multiple times to solve the sub-expressions in expressions orderly which is very in-efficient.
- To avoid this traversing, Infix expressions are converted to Postfix expression before evaluation.

To evaluate an infix expression, we need to perform 2 main tasks:

1. Convert infix to postfix
2. Evaluate postfix

(1) Convert infix to postfix

To convert Infix expression to Postfix expression, we will use the **stack** data structure. By scanning the infix expression from left to right, if we get any operand, simply add it to the postfix form, and for the operator and parenthesis, add them in the stack maintaining theprecedence of them.

Algorithm

Step 1 : Scan the Infix Expression from left to right.

Step 2 : If the scanned character is an operand, append it with final Postfix string.

Step 3 : Else,

Step 3.1 : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a „(„ or „[„ or „{„), push it on stack.

Step 3.2 : Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that "Push" the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

Step 4 : If the scanned character is an „(„ or „[„ or „{„, push it to the stack.

Step 5 : If the scanned character is an „)"or „]" or „}", pop the stack and output it until a „(„ or „[„ or „{„ respectively is encountered, and discard both the parenthesis.

Step 6 : Repeat steps 2 to 5 until infix expression is scanned.

Step 7 : Print the output

Step 8 : Pop and output from the stack until it is not empty.

Example 1:

Infix Expression : 3+4*5/2

Step 1 : Initially Stack is Empty and the very first literal of Infix Expression is '3' which is operand hence push it on output stack.

Stack :	Output : 3

Step 2 : Next literal of expression is + which is operand, hence needed to be pushed on stack but intially stack is empty hence literal will directly pushed on to stack.

Stack : +	Output : 3

Step 3 : Further 4 is an operand should be pushed on stack.

Stack : +	Output : 3 4

Step 4 : Next literal is * which is an operator, as stack is not empty, priority should be checked of in-stack operator(top of stack) and of incoming operator i.e * as priority of in-stack operator is less than incoming operator, * will be pushed on to stack.

Stack : + *	Output : 3 4

Step 5 : Next literal is 5 which is an operand, hence should be pushed on to output stack.

Stack : + *	Output : 3 4 5

Step 6 : Next literal is / which is an operator, as stack is not empty, priority should be checked of in-stack operator(top of stack) i.e * and of incoming operator i.e /, as priority of / and * are equal hence * will be poped out of stack and will be stored on output stack and operator / will be stored on stack.

Stack : + /	Output : 3 4 5 *

Step 7 : Next literal is 2 which is an operand, hence should be pushed on output stack.

Stack : + /	Output : 3 4 5 * 2

Step 8 : As now all literals are traversed, despite stack is not empty, hence pop all literals from stack and pushed it on to output stack.

Postfix Expression: 3 4 5 * 2 / +

Once the expression is converted to postfix notation, step 2 can be performed:

(2) Evaluate postfix

Step 1 : Scan the postfix from left to right.

Step 2 : If the character is an operand, push it into the stack.

Step 3 : If the character is an operator, then do the following.

i. Pop the two top most elements from the stack and
ii. Perform the operation
iii. Push the result back to the stack.

Once the expression is fully traversed, the element in the stack is the result.

Example

Given infix expression is: **3+4*5/2** and equivalent postfix by the step1 is **3 4 5 * 2 / +**

Stack S = [], traverse the string from left to right:

3 : Operand,

push into the stack, S = [3], top → 3

4 : Operand,

push into the stack, S = [3, 4], top → 4

5 : Operand,

push into the stack, S = [3, 4, 5], top → 5

*** : Operator,**

pop top two elements, op2 = 5, op1 = 4.

Stack after pop operations S = [3], top → 3.

Now, push the result of op1 * op2, i.e 4 * 5 = 20 into the stack. S = [3, 20], top → 20

2 : Operand,

push into the stack, S = [3, 20, 2], top → 2

/ : Operator,

pop two elements, op2 = 2, op1 = 20

Stack after pop operations S = [3], top → 3.

Now, push the result of op1 / op2, i.e 20 / 2 = 10 into the stack. S = [3, 10], top → 10

+ : Operator,

pop top two elements, op2 = 10, op1 = 3.

Stack after pop operations S = [].

Push the result of op1 + op2 into the stack, i.e 3 + 10 = 13, S = [13]

The string has been completely traversed, the stack contains only one element which is the result of the given expression ie. **3+4*5/2** = **13**.

Example 2: Infix string 2 * 3 / (2 - 1) + 5 * 3

1.**Infix to Postfix Conversion**

S. No.	Input	Operator Stack	Output String
1	2		2
2	*	*	
3	3	*	2 3
4	/	/	2 3 *
5	(	/ (	2 3 *
6	2	/ (	2 3 * 2
7	-	/ (-	2 3 * 2
8	1	/ (-	2 3 * 2 1
9	)	/	2 3 * 2 1 -
10	+	+	2 3 * 2 1 - /
11	5	+	2 3 * 2 1 - / 5
12	*	+ *	2 3 * 2 1 - / 5
13	3	+ *	2 3 * 2 1 - / 5 3
			2 3 * 2 1 - / 5 3 * +

2.Postfix Evaluation

Input String : 2 3 * 2 1 - / 5 3 * +

S.No.	Input	Operand 1	Operand 2	Calculation	Output Stack
1	2				2
2	3				2 3
3	*	2	3	2 * 3 = 6	6
4	2				6 2
5	1				6 2 1
6	-	2	1	2 – 1 = 1	6 1
7	/	6	1	6 / 1 = 6	6
8	5				6 5
9	3				6 5 3
10	*	5	3	5 * 3 = 15	6 15
11	+	6	15	6 + 15 = 21	**21**

Result : 21

QUEUE

Queue is a linear data structure that follows a particular order in which the operations are performed. The order is **First-In-First-Out (FIFO)** i.e. the data item stored first will be accessed first. In this, entering and retrieving data is not done from only one end.

An example of a queue is any queue of consumers for a resource where the consumer that came first is served first.

It can be implemented by using both array and linked list.

Example:

Queue as the name says is the data structure built according to the queues of a bus stop or train where the person who is standing in the front of the queue (standing for the longest time) is the first one to get the ticket. So any situation where resources are shared amongmultiple users and served on a first come first serve basis.

Examples include CPU scheduling, Disk Scheduling.

Basic Operations on Queue:

- **enqueue():** Inserts an element at the end of the queue i.e. at the rear end.
- **dequeue():** This operation removes and returns an element that is at the front end of the queue.

Auxiliary Operations on Queue:

- **front():** This operation returns the element at the front end without removing it.
- **rear():** This operation returns the element at the rear end without removing it.
- **isEmpty():** This operation indicates whether the queue is empty or not.
- **isFull()**: This operation indicates whether the queue is full or not.
- **size():** This operation returns the size of the queue i.e. the total number of elements it contains.

Types of Queues:

(1) Simple Queue:

Simple queue also known as a linear queue is the most basic version of a queue. Here, insertion of an element i.e. the Enqueue operation takes place at the rear end and removal of an element i.e. the Dequeue operation takes place at the front end.

a) Input restricted Queue:

In this type of Queue, the input can be taken from one side only(rear) and deletion of elements can be done from both sides(front and rear). This kind of Queue does not follow FIFO(first in first out).

b) Output restricted Queue:

In this type of Queue, the input can be taken from both sides(rear and front) and the deletion of the element can be done from only one side(front).

(2) Circular Queue:

In a circular queue, the elements of the queue act as a circular ring. The working of a circular queue is similar to the linear queue except for the fact that the last elementis connected to the first element. Its advantage is that the memory is utilized in a better way. This is because if there is an empty space i.e. if no element is present at a certain position in the queue, then an element can be easily added at that position.

(3) Priority Queue:

This queue is a special type of queue. Its specialty is that it arranges the elements in a queue based on some priority.

a) Descending Priority Queue

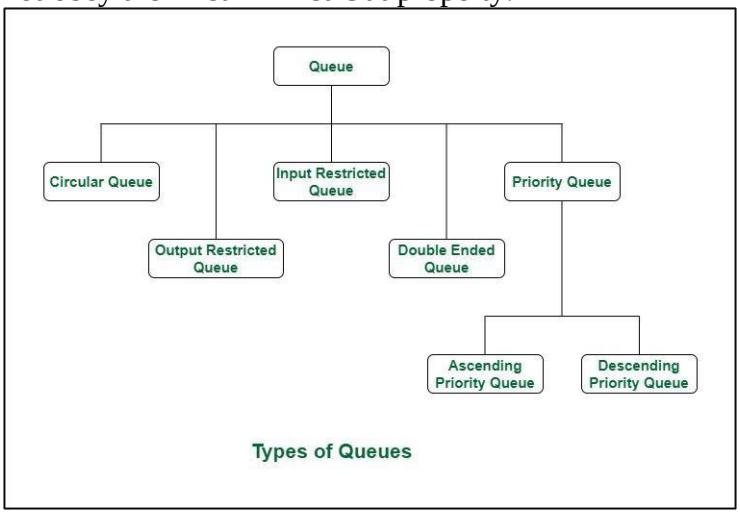
The priority can be something where the element with the highest value has the priority so it creates a queue with decreasing order of values.

b) Ascending Priority Queue

The priority can also be such that the element with the lowest value gets the highest priority so in turn it creates a queue with increasing order of values.

(4) Dequeue:

Dequeue is also known as Double Ended Queue. As the name suggests double ended, it means that an element can be inserted or removed from both the ends of the queue unlike the other queues in which it can be done only from one end. Because ofthis property it may not obey the First In First Out property.



Types of Queues

Array implementation of Queue:

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple.

Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear**'.

Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Enqueue () - Inserting value into the queue

In a queue data structure, Enqueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. We can use the following steps to insert an element into the queue...

1. Check the queue is full or not
2. If full, print overflow and exit
3. If queue is not full, increment **rear** and add the element

```
if(rear == MAX - 1)
   printf("Queue Overflow \n");
else
{
 if(front== - 1)
      front = 0;
 printf("Enter the value to be inserted into queue : ");
 scanf("%d", &value);
 rear = rear + 1;
 a[rear] = value;
 }
```

deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

1. Check queue is empty or not
2. if empty, print underflow and exit
3. if not empty, print element at the **front** and increment **front**

```
if(front == - 1 || front > rear)
{
 printf("Queue Underflow \n");
 front = rear = -1;
 return;
 }
else
{
 printf("Element deleted from queue is : %d \n", a[front]);
 front = front + 1;
 }
```

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

1. Check queue is empty or not
2. if empty, print underflow and exit
3. Else,
   i. then define an integer variable '**i**' and set '**i** = **front+1**'.
   ii. Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

```
int i;
 if(front > rear || rear == -1)
         printf("Queue is empty \n");
 else
 {
   for(i = front; i <= rear; i++)
       printf("%d \t", a[i]);
 }
```

isFull ( ) Operation

This function checks if the queue is full or not. If the queue is full, then the insertion of elements is not possible in a queue. This condition is known as overflow error. You need to perform the following steps while carrying this operation:

```
  if (rear == n-1)
     return 1
  else
    return 0
```

isEmpty ( ) Operation

This function validates if the queue is empty. If both the front and rear nodes are pointing to null memory space (-1), then you can consider the queue as empty.

```
       if (front == -1 || front > rear)
          return 1
       else
          return 0
```

Peek ( ) Operation

This function helps in extracting the data element where the **front** is pointing without removing it from the queue.

```
        if ( isEmpty())
           print QUEUE IS EMPTY
        else
           return a[front]
```

Drawbacks of Queue Implementation Using Array

Although this method of creating a queue using an array is easy, some drawbacks make this method vulnerable. The drawbacks of queue implementation using array are,

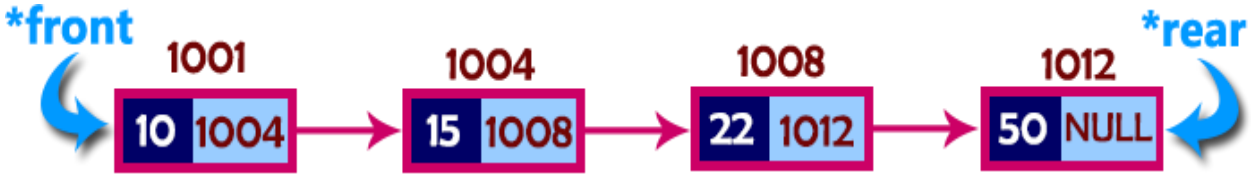
- **Memory Wastage:** The memory space utilized by the array to store the queue elements can never be re-utilized to store new queue elements. If we simply increment front and rear indices, then there may be problems, the front may reach the end of the array. The solution to this problem is to increase front and rear in circular manner.
- **Deciding the array size:** In this method, you have to predetermine the size of an array and it is almost impossible to extend an array size at runtime.

Queue Using Linked List

The major problem with the queue implemented using an array is, it will work for an only fixed number of data values. That means, the number of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. Thatmeans, queue using linked list can work for the variable size of data (No need to fix the size atthe beginning of the implementation).

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program. And declare all the **user defined functions**.

Step 2 - Define a '**Node**' structure with two members **data** and **next**.

Step 3 - Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.

Step 4 - Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- **Step 1 -** Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- **Step 2 -** Check whether queue is **Empty** (**rear** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode**.

```
    newNode = malloc(sizeof(struct Node));
    newNode -> data = value;
    newNode -> next = NULL;
    if(front == NULL)
      front = rear = newNode;
    else
    {
      rear -> next = newNode;
      rear = newNode;
     }
```

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- **Step 1 -** Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Queue is Underflow!!! Deletion is not possible!!!"** and terminate from the function
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- **Step 4 -** Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

```
  if(front == NULL)
    printf("\n Queue is Underflow!!!\n");
  else
  {
    struct Node *temp = front;
    front = front -> next;
    printf("\n Deleted element: %d\n", temp->data);
    free(temp);
   }
```

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- **Step 1 -** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2 -** If it is **Empty** then, display '**Queue is Empty!!!**' and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with **front**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

```
if(front == NULL)
     printf("\n Queue is Empty!!!\n");
  else
  {
   struct Node *temp = front;
   while(temp->next != NULL)
   {
         printf("%d--->",temp->data);
            temp = temp -> next;
    }
    printf("%d--->NULL\n",temp->data);
  }
```

Applications of Queue:

- Queue is used for handling website traffic.
- It helps to maintain the playlist in media players.
- Queue is used in operating systems for handling interrupts.
- It helps in serving requests on a single shared resource, like a printer, CPU task scheduling, etc.
- It is used in asynchronous transfer of data for e.g. pipes, file IO, sockets.
- Queues are used for job scheduling in operating system.
- In social media to upload multiple phots or videos queue is used.
- To send an e-mail queue data structure is used.
- To handle website traffic at a time queue are used.
- In windows operating system, to switch multiple application.

Real-Life Applications of Queue:

- A real world example of queue is a single lane one way road, where the vehicle that enters first will exit first.
- A more real-world example can be seen in the queue at the ticket windows.
- Cashier line in a store is also an example of queue.
- People on an escalator