**UNIT-I: ( INTRODUCTION TO DATA STRUCTURES)**

# Data structure:

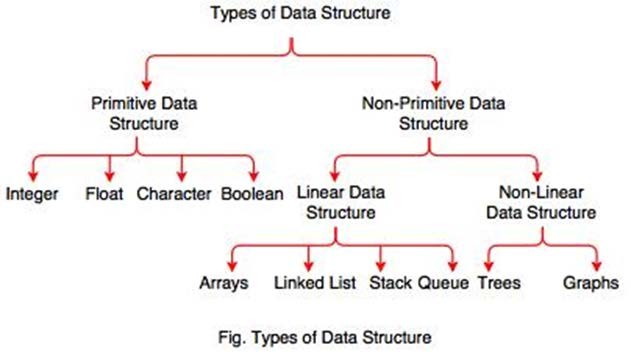
A Data structure is a specialized format for organizing, processing, storing and retrieving the data. It is a method of storing and organizing data efficiently in the memory.

DATA STRUCTURE= DATA ELEMENTS+OPERATIONS+ ALGORITHMS

The operations done on data structures:

1. Data organizing or clubbing.
2. Accessing technique.
3. Manipulating selections for information

# Flowchart:



**PRIMITIVE DATA STRUCTURES:**

These are directly supported by machines without writing any code.

All primary data types are primitive.

EX; int, float, char…

**Non-PRIMITIVE DATA STRUCTURES:**

These are not directly supported by machines.

Classified as linear and non-linear.

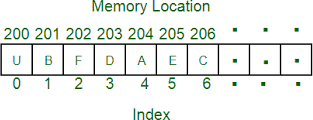
1. **Linear data structure:**

The data in which the values or information is arranged in a linear fashion is known as “linear data structure”.

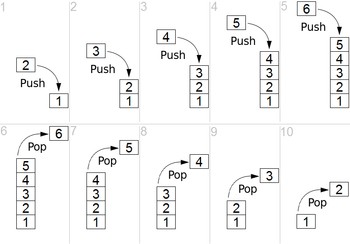
List of linear DS: arrays, stacks, queues, and linked lists.

# Arrays:

The array is the collection of similar type of data stored at contiguous memory locations.

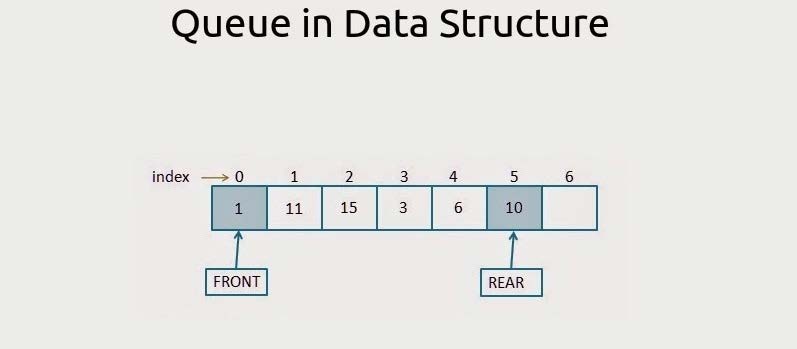
**Stacks:**

Stack is the non-primitive dynamic linear data structure which follows the F.I.L.O or L.I.F.O principle.



# Queues:

Queue is a non-primitive dynamic linear data structure which follows the F.I.F.O or L.I.L.O principle.



# Linked lists:

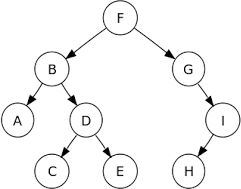
A linked lists is a non-primitive dynamic linear data structure in which the elements are not stored at contiguous memory locations.

# 2) Nonlinear data structure:

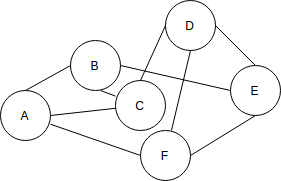
The data structure in which the values in the structure are not arranged in an order is known as “non- linear data structure”. This type is opposite to linear data structure.

List of nonlinear data structures: Trees and Graphs

* **Tree**: Tree is a non-primitive non-linear hierarchical data structure which stores the information naturally in the form of hierarchy style.



**Graphs**: A Graph is a non-primitive non-linear data structure consisting of nodes and edges.



**List of operations:**

1. **Traversing**: Accessing records exactly once so that certain items in the record may be processed.
2. **Searching**: Finding the location of a particular record with a given key value, or

finding the location of all records which satisfy one or more conditions.

1. **inserting**: Adding the new record to the structure.
2. **Deleting**: Removing the record from the structure.
3. **Sorting**: Managing the data or record in some logical order.
4. **Merging**: Combining the record in two different sorted files into single sorted file.

**Applications of data structures:**

* 1. **Stack**: Stacks are used to convert infix expression to postfix expression.

In recursion system stack created stores he previous value of recursive function.

* 1. **Queue**: Queue is used in topological shorting by which appropriate topology to Connect computer and internet.

This is also used in the printer applications.

* 1. **List**: Lists are used to store huge number of records in sequential order.
  2. **Graph**: Graphs are used to establish any network system like LAN, WAN, etc…..

# Abstract Data Type (ADT):

# Abstraction means representing only the essential things and hiding the implementation details.

The data with its type and set of operations on it (for insertion, handling, manipulation and retrieving it) is referred to as “Abstract Data Type”.

It is a specification of set of data and set of operations that can be performed on data.

Every Data structure has an ADT.

ADT minimizes the dependencies in our code, which helps us to modify our code if needed.

This helps us to hide lower- level details such as representation, execution and limitations from the programmers.

Examples: List, Queue, Stack, String, Tree, Set, etc………

It consists of

1. Declaration of data (objects/instances/elements..)
2. Declaration of operations ( functions..)

Example:

|  |
| --- |
| ARRAY ADT: |
| Objects/ Instances/data elements:  Int a[];  Int b[];.  … |
| Operatios:  Void read();  Void display();  Void search();  Void reverse()  Void insert()  ………. |

//IMPLEMENTATION OF ARRAY ADT

#include<stdio.h>

#include<stdlib.h>

#define max 20

int a[max],n;

void read(int);

void display(int a[], int);

void search(int a[], int);

void reverse(int a[], int);

void merge(int a[], int);

void insert(int a[], int);

main()

{

int ch,n1;

system("clear");

while(1)

{

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n ARRAY ADT OPERATIONS ARE:\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n\t1.READ");

printf("\n\t2.DISPLAY");

printf("\n\t3.SEARCH");

printf("\n\t4.REVERSE");

printf("\n\t5.INSERT");

printf("\n\t6.MERGE");

printf("\n\t7.EXIT");

printf("\n Enter ur choice:");

scanf("%d",&ch);

switch(ch)

{

case 1: printf("\n enter size of the array:");

scanf("%d",&n);

read(n);

break;

case 2: display(a,n);

break;

case 3: search(a,n);

break;

case 4: reverse(a,n);

break;

case 5: insert(a,n);

break;

case 6: merge(a,n);

break;

case 7: exit(0);

break;

default: printf("\n wrong choice\n");

}

}

}

void read(int n1)

{

int i;

printf("\n Enter array elements:");

for(i=0;i<n1;i++)

scanf("%d",&a[i]);

}

void display(int x[], int n1)

{

int i;

printf("\n array elements are :");

for(i=0;i<n1;i++)

printf("\t%d",x[i]);

}

void search(int a[], int n1)

{

int i,element,t=0;

printf("\n Enter element to be searched:");

scanf("%d",&element);

for(i=0;i<n;i++)

{

if(a[i]==element)

{

printf("\n element is found at %d position",i+1);

t=1;

break;

}

}

if(t==0)

printf("\n element not found");

}

void reverse(int a[], int n1)

{

int i,b[max];

for(i=0;i<n;i++)

a[i]=a[n-i-1];

printf("\n elements in reverse order:");

display(a,n);

}

void insert(int a[], int n1)

{

int i,element,pos;

printf("\n Enter element to be inserted:");

scanf("%d",&element);

printf("\n Enter the position:");

scanf("%d",&pos);

for(i=n1;i>=pos-1;i--)

{

a[i]=a[i-1];

}

a[pos-1]=element;

n++;

display(a,n);

}

void merge(int a[], int n1)

{

int i,j,b[max],c[max],m;

printf("\n Enter no-of elements in the second array:");

scanf("%d",&m);

printf("\n Enter elements:");

for(i=0;i<m;i++)

scanf("%d",&b[i]);

for(i=0;i<n1;i++)

c[i]=a[i];

for(j=0;j<m;j++)

{

c[i]=b[j];

i++;

}

display(c,n+m);

}

**Stack using arrays:**

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.



A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

## Stack Representation



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

## Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

* **push()** − Pushing (storing) an element on the stack.
* **pop()** − Removing (accessing) an element from the stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

* **peek()** − get the top data element of the stack, without removing it.
* **isFull()** − check if stack is full.
* **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

### peek()

Algorithm of peek() function −

begin procedure peek

return stack[top]

end procedure

Implementation of peek() function in C programming language −

**Example**

int peek() {

return stack[top];

}

### isfull()

Algorithm of isfull() function −

begin procedure isfull

if top equals to MAXSIZE

return true

else

return false

endif

end procedure

Implementation of isfull() function in C programming language −

**Example**

bool isfull() {

if(top == MAXSIZE)

return true;

else

return false;

}

### isempty()

Algorithm of isempty() function −

begin procedure isempty

if top less than 0

return true

else

return false

endif

end procedure

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

**Example**

bool isempty() {

if(top == -1)

return true;

else

return false;

}

## Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

* **Step 1** − Checks if the stack is full.
* **Step 2** − If the stack is full, produces an error and exit(OVERFLOW).
* **Step 3** − If the stack is not full, increments **top** to point next empty space.
* **Step 4** − Adds data element to the stack location, where top is pointing.
* **Step 5** − Returns success.



**Example**

void push(int data) {

if(!isFull()) {

top = top + 1;

stack[top] = data;

} else {

printf("Could not insert data, Stack is full.\n");

}

}

## Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. A Pop operation may involve the following steps −

* **Step 1** − Checks if the stack is empty.
* **Step 2** − If the stack is empty, produces an error and exit(UNDERFLOW).
* **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
* **Step 4** − Decreases the value of top by 1.
* **Step 5** − Returns success.



### Algorithm for Pop Operation

Implementation of this algorithm in C, is as follows −

**Example**

int pop(int data) {

if(!isempty()) {

data = stack[top];

top = top - 1;

return data;

} else {

printf("Could not retrieve data, Stack is empty.\n");

}

}

**Applications of stack:**

* 1. Converting Infix to Postfix expression
  2. Postfix expression evaluation
  3. Implementing recursion
  4. Multi tasking
  5. Scheduling algorithms
  6. **Syntax Parsing**
  7. **Backtracking algorithms**

 //STACK ADT IMPLEMENTAION USING ARRAYS

#include<stdio.h>

#define max 50

#include<stdlib.h>

int top=-1;

int stack[max];

void push();

void pop();

void display();

void main()

{

int ch;

system("clear");

while(1)

{

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n STACK ADT OPERATIONS ARE:\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n\t1.PUSH");

printf("\n\t2.POP");

printf("\n\t3.DISPLAY");

printf("\n\t4.EXIT");

printf("\n Enter ur choice:");

scanf("%d",&ch);

switch(ch)

{

case 1: push();

break;

case 2: pop();

break;

case 3: display();

break;

case 4: exit(0);

break;

default: printf("\n wrong choice\n");

}

}

}

void push()

{

int element;

if(top==max-1)

printf("\n STACK OVERFLOW\n");

else

{

printf("\n Enter element to be inserted:");

scanf("%d",&element);

top=top+1;

stack[top]=element;

}

}

void pop()

{

if(top==-1)

printf("\n STACK UNDERFLOW\n");

else

{

printf("\n deleted element is:%d\n",stack[top]);

top=top-1;

}

}

void display()

{

int i;

if(top==-1)

printf("\n STACK IS EMPTY\n");

else

{

printf("\n STACK ELEMENTS ARE:\n");

for(i=top;i>=0;i--)

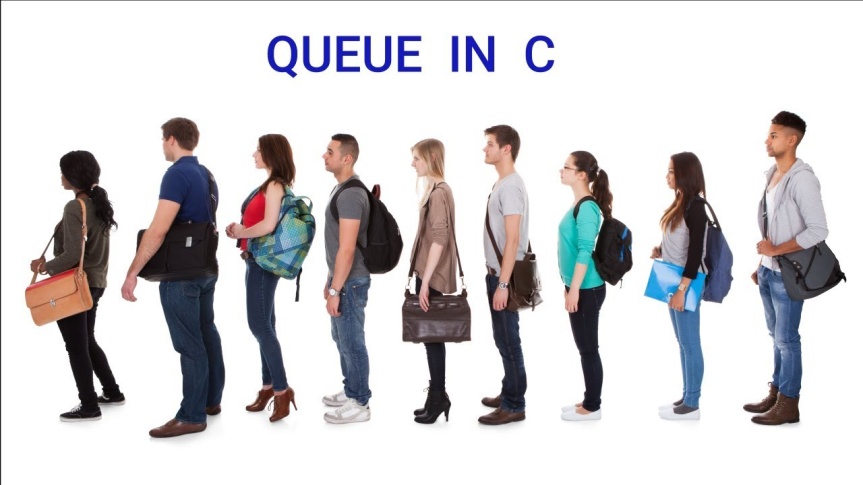
printf("\t%d",stack[i]);

}

}

# QUEUE using Arrays:

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.





A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

## Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

## Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

* **enqueue()** − add (store) an item to the queue.
* **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

* **peek()** − Gets the element at the front of the queue without removing it.
* **isfull()** − Checks if the queue is full.
* **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

### peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

**Algorithm**

begin procedure peek

return queue[front]

end procedure

Implementation of peek() function in C programming language −

**Example**

int peek() {

return queue[front];

}

### isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

**Algorithm**

begin procedure isfull

if rear equals to MAXSIZE

return true

else

return false

endif

end procedure

Implementation of isfull() function in C programming language −

**Example**

bool isfull() {

if(rear == MAXSIZE - 1)

return true;

else

return false;

}

### isempty()

Algorithm of isempty() function −

**Algorithm**

begin procedure isempty

if front is less than MIN OR front is greater than rear

return true

else

return false

endif

end procedure

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

**Example**

bool isempty() {

if(front < 0 || front > rear)

return true;

else

return false;

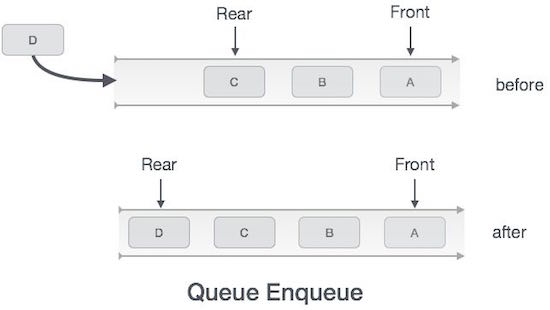
}

## Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

* **Step 1** − Check if the queue is full.
* **Step 2** − If the queue is full, produce overflow error and exit.
* **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
* **Step 4** − Add data element to the queue location, where the rear is pointing.
* **Step 5** − return success.



Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

### Algorithm for enqueue operation

procedure enqueue(data)

if queue is full

return overflow

endif

rear ← rear + 1

queue[rear] ← data

return true

end procedure

Implementation of enqueue() in C programming language −

**Example**

int enqueue(int data)

if(isfull())

return 0;

rear = rear + 1;

queue[rear] = data;

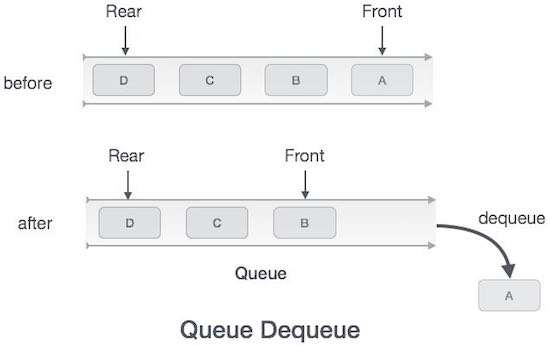
return 1;

end procedure

## Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

* **Step 1** − Check if the queue is empty.
* **Step 2** − If the queue is empty, produce underflow error and exit.
* **Step 3** − If the queue is not empty, access the data where **front** is pointing.
* **Step 4** − Increment **front** pointer to point to the next available data element.
* **Step 5** − Return success.



### Algorithm for dequeue operation

procedure dequeue

if queue is empty

return underflow

end if

data = queue[front]

front ← front + 1

return true

end procedure

Implementation of dequeue() in C programming language −

**Example**

int dequeue() {

if(isempty())

return 0;

int data = queue[front];

front = front + 1;

return data;

}

**Applications**:

* Graph traversals
* CPU scheduling
* Disk Scheduling
* Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.
* Multiprogramming.
* Asynchronous data transfer (file IO, pipes, sockets).

.

//QUEUE ADT IMPLEMENTAION USING ARRAYS

#include<stdio.h>

#define max 50

#include<stdlib.h>

int front=-1; int rear=-1;

int queue[max];

void insertion();

void deletion();

void display();

void main()

{

int ch;

system("clear");

while(1)

{

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n QUEUE ADT OPERATIONS ARE:\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n\t1.INSERTION");

printf("\n\t2.DELETION");

printf("\n\t3.DISPLAY");

printf("\n\t4.EXIT");

printf("\n Enter ur choice:");

scanf("%d",&ch);

switch(ch)

{

case 1: insertion();

break;

case 2: deletion();

break;

case 3: display();

break;

case 4 : exit(0);

break;

default: printf("\n wrong choice\n");

}

}

}

void insertion()

{

int element;

if(front==-1)

front=0;

if(rear==max-1)

printf("\n QUEUE OVERFLOW\n");

else

{

printf("\n Enter element to be inserted:");

scanf("%d",&element);

rear=rear+1;

queue[rear]=element;

}

}

void deletion()

{

if(front==-1||front>rear)

printf("\n QUEUE UNDERFLOW\n");

else

{

printf("\n deleted element is:%d\n",queue[front]);

front=front+1;

}

}

void display()

{

int i;

if(front==-1||front>rear)

printf("\n QUEUE IS EMPTY\n");

else

{

printf("\n QUEUE ELEMENTS ARE:\n");

for(i=front;i<=rear;i++)

printf("%d\t",queue[i]);

}

}

### Linked List Concepts:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

### Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

* + 1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
    2. Linked lists have efficient memory utilization. Here, memory is not pre- allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
    3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
    4. Many complex applications can be easily carried out with linked lists.
    5. Elements may or may not be stored in consecutive memory locations so if we do not have consecutive memory available, even then we can store the data in computer.

### Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.
3. No random access of elements as we do in array by index. We have to access each node sequentially.
4. Reverse traversing is difficult. In case of doubly linked list its easier but extra memory is required for back pointer hence wastage of memory.

### Comparison between array and linked list:

|  |  |
| --- | --- |
| **ARRAY** | **LINKED LIST** |
| Size of an array is fixed | Size of a list is not fixed |
| Memory is allocated from stack | Memory is allocated from heap |
| It is necessary to specify the number of elements during declaration (i.e., during compile time). | It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run  time). |
| It occupies less memory than a linked list for the same number of elements. | It occupies more memory. |
| Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to  make room. | Inserting a new element at any position can be carried out easily. |
| Deleting an element from an array is not possible. | Deleting an element is possible. |

**Trade offs between linked lists and arrays:**

|  |  |  |
| --- | --- | --- |
| **FEATURE** | **ARRAYS** | **LINKED LISTS** |
| Sequential access | Efficient | efficient |
| Random access | Efficient | inefficient |
| Resigning | Inefficient | efficient |
| Element rearranging | Inefficient | efficient |
| Overhead per elements | None | 1 or 2 links |

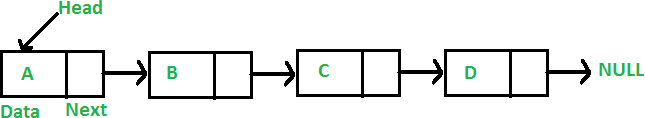
### Types of Linked Lists:

Basically we can put linked lists into the following four items:

* + 1. Single Linked List.
    2. Double Linked List.
    3. Circular Linked List.

# Single Linked List.

Linked List is a non-primitive linear data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain.

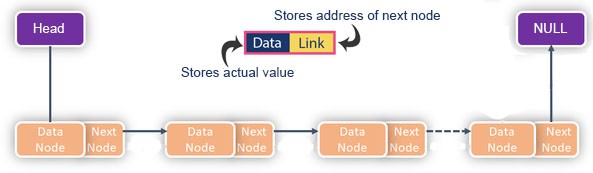


As per the above illustration, following are the important points to be considered:

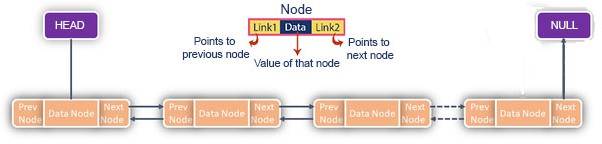
* + Linked List contains a reference element (pointer) called head (which points to first node).
  + Each node carries a data field and an address field called next.
  + Each node is linked with its next node using its next link.
  + Last node carries a link as NULL representing the end of the list.

There are three types of linked lists,

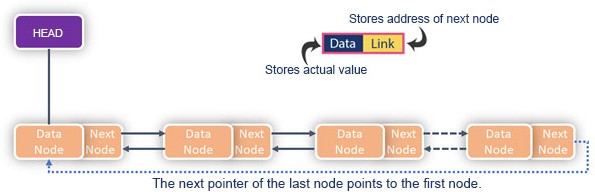
Singly Linked List: Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The link of the last node in the list is NULL, which indicates the end of the list. Item navigation is forward only.



Doubly Linked List: Doubling Linked Lists have two address nodes, in particular, one node serves the purpose of directing to the next node while the other one points to the previous node. A linked list that has only next links is a singly linked list. Adding previous links makes a doubly linked list. Item navigation is both forward and backward.



Circular Linked List: Circular linked list is a linked list where all nodes are connected to form a circle. Simply, every node has the address of next node; putting the address of the first node in the last node makes it circular linked list. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



# Basic operations carried out in a linked list:

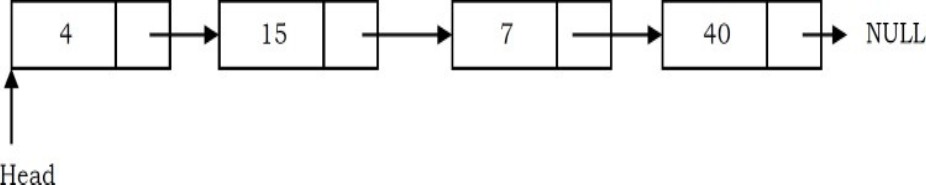
Following are the basic operations supported by a linked list:

1. Insertion:
   * Insertion at beginning: - Inserts the new node at the beginning of the linked list.
   * Insertion at end: - Inserts the new node at the ending of the linked list.
   * Insertion after a given value: - Inserts the new node after a given value.
2. Deletion:
   * Deletion at beginning: - Deletes the node at the beginning of the linked list.
   * Deletion at end: - Deletes the node at the end of the linked list.
   * Deletion of a value: - Deletes the node of given value.
3. Traversing:
   * Traversal through a linked list means travelling through every single node of a list and reaching the end of the list. The traversal can be in only one direction in singly linked list and two directions (forward and reverse) in a doubly linked list.
4. Display:
   * Displays all the values (data) present in the linked list.
5. Search:
   * Searches for the element in the linked list using the given key.

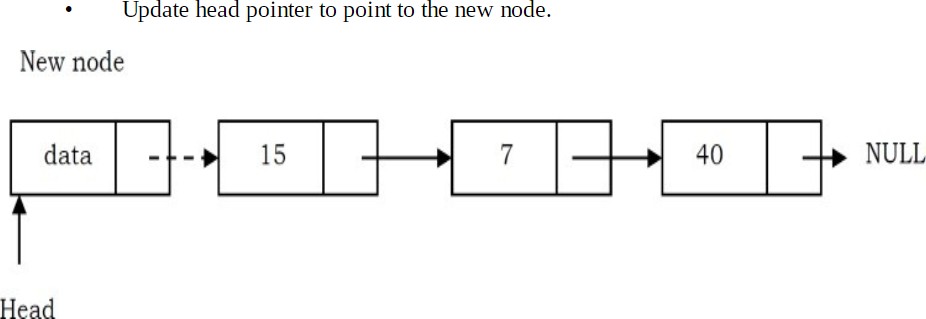
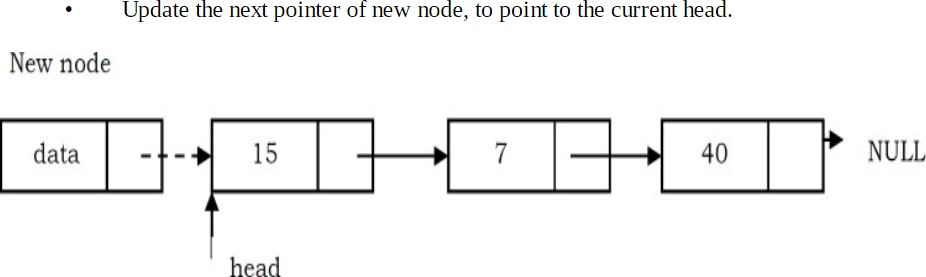
**ADDITIONAL INFORMATION:**

**SINGLY LINKED LIST OPERATIONS ILLUSTRATED:**

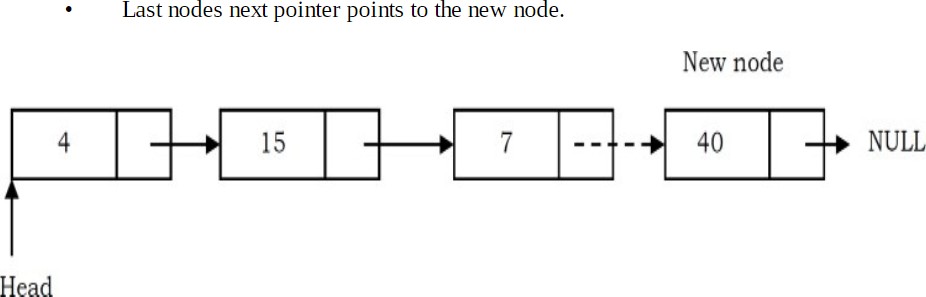
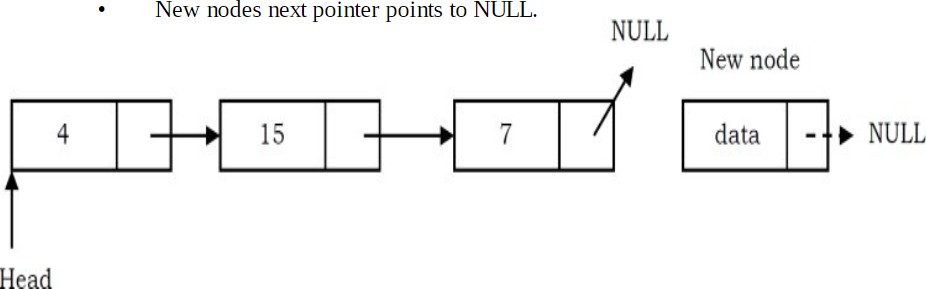
* **Singly linked list:**



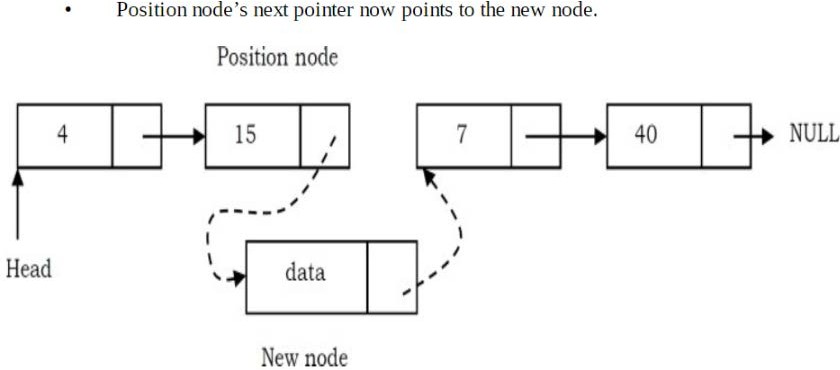
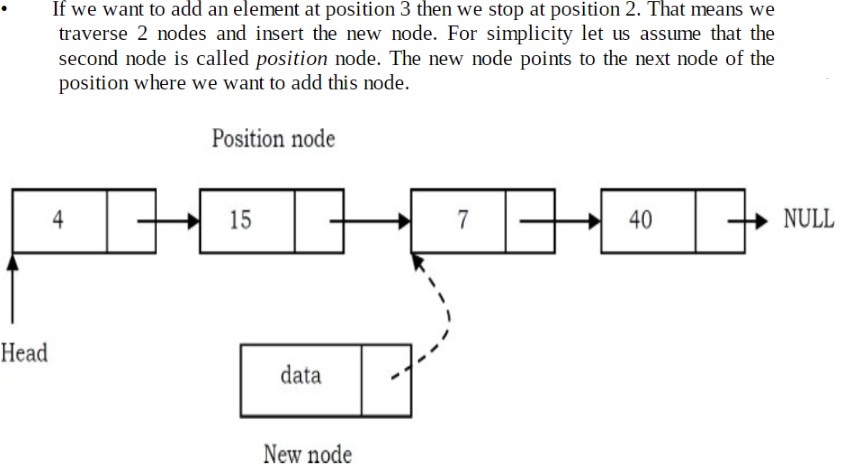
* **Inserting a node at the beginning:**



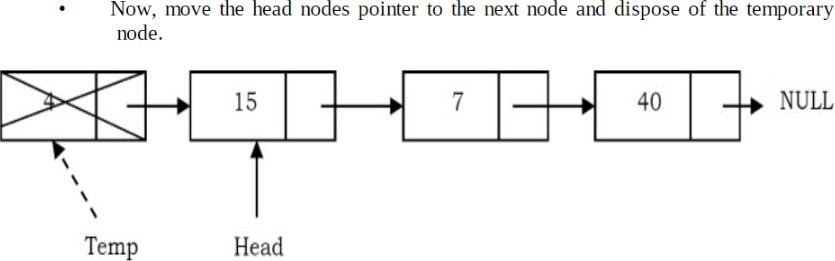
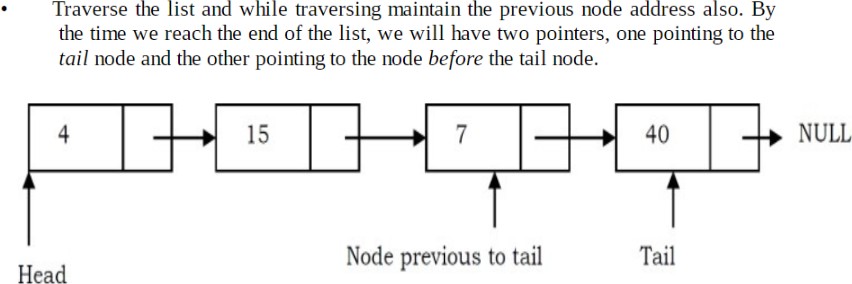
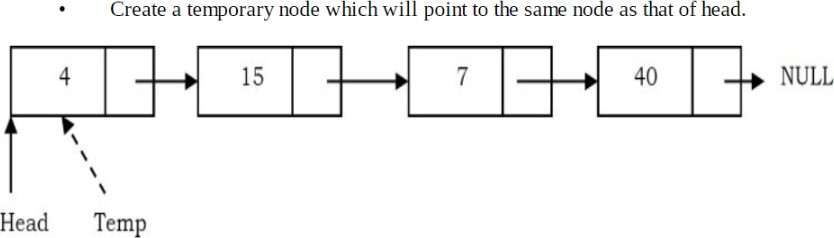
* + **Inserting a node at the end:**



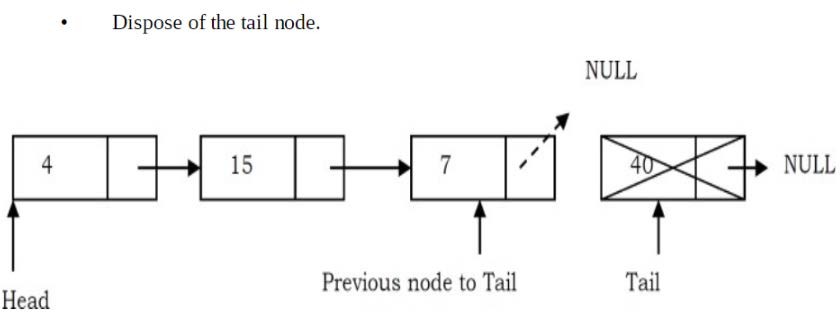
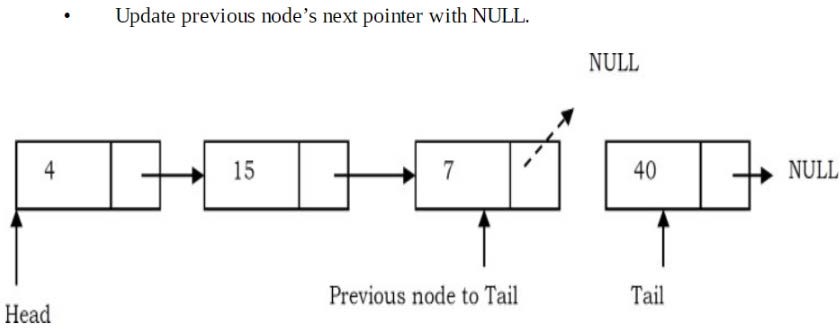
* **Inserting a Node in Singly Linked List at the Middle:**



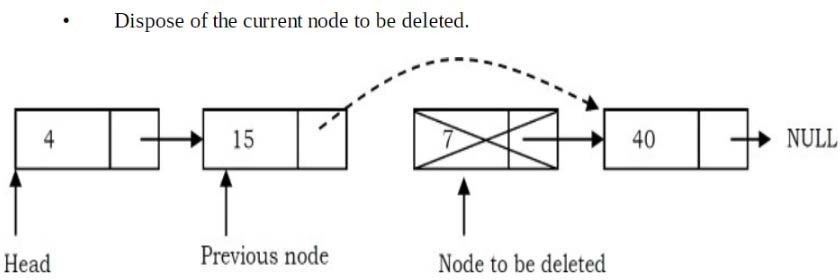
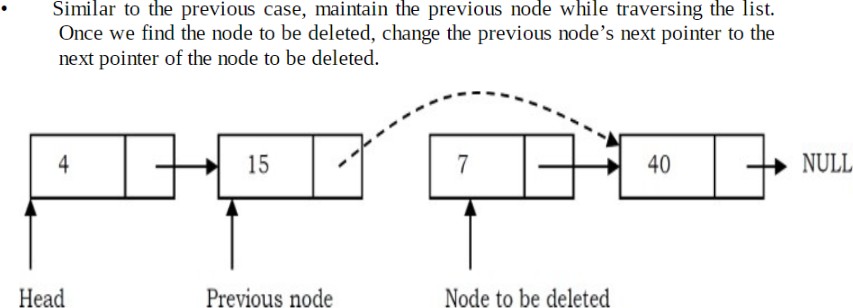
* **Deleting the First Node:**



* **Deleting the Last Node in Singly Linked List:**



* **Deleting an Intermediate Node in Singly Linked List:**



//IMLEMENTAION OF SINGLE LINKED LIST ADT

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*next;

};

struct node \*head;

void create();

void insert\_begin();

void insert\_after();

void insert\_end();

void delete\_begin();

void delete\_info();

void delete\_end();

void display();

void main()

{

int ch;

system("clear");

while(1)

{

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n SINGLE LINKED LIST ADT OPERATIONS ARE:\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n\t1.CREATE");

printf("\n\t2.INSERTION AT THE BEGINNING");

printf("\n\t3.INSERTION AFTER THE GIVEN INFO:");

printf("\n\t4.INSERTION AT THE END");

printf("\n\t5.DELETION AT THE BEGINNING");

printf("\n\t6.DELETION THE GIVEN INFO:");

printf("\n\t7.DELETION AT THE END");

printf("\n\t8.DISPLAY");

printf("\n\t9.EXIT");

printf("\n Enter ur choice:");

scanf("%d",&ch);

switch(ch)

{

case 1: create();

break;

case 2: insert\_begin();

break;

case 3: insert\_after();

break;

case 4: insert\_end();

break;

case 5: delete\_begin();

break;

case 6: delete\_info();

break;

case 7: delete\_end();

break;

case 8: display();

break;

case 9: exit(0);

break;

default: printf("\n wrong choice\n");

}

}

}

void create()

{

struct node \*ptr,\*cptr;

int c;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter first node information:");

scanf("%d",&ptr->data);

head=ptr;

printf("\n Enter 0/1 for more nodes:");

scanf("%d",&c);

while(c==1)

{

cptr=(struct node\*)malloc(sizeof(struct node));

ptr->next=cptr;

ptr=cptr;

printf("\n Enter next node information:");

scanf("%d",&cptr->data);

printf("\n Enter 0/1 for more nodes:");

scanf("%d",&c);

}

ptr->next=NULL;

}

void insert\_begin()

{

struct node \*ptr;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter node information to be inserted:");

scanf("%d",&ptr->data);

ptr->next=head;

head=ptr;

}

void insert\_end()

{

struct node \*ptr, \*cptr;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter node information to be inserted:");

scanf("%d",&ptr->data);

cptr=head;

while(cptr->next!=NULL)

cptr=cptr->next;

cptr->next=ptr;

ptr->next=NULL;

}

void insert\_after()

{

struct node \*ptr, \*cptr;

int d;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter node information to be inserted:");

scanf("%d",&ptr->data);

printf("\n Enter node info after which U want to insert:");

scanf("%d",&d);

cptr=head;

while(cptr->data!=d)

cptr=cptr->next;

ptr->next=cptr->next;

cptr->next=ptr;

}

void delete\_begin()

{

struct node \*ptr;

if(head==NULL)

printf("\n LINKED LIST UNDERFLOW\n");

else

{

ptr=head;

printf("\n deleted element is :%d",ptr->data);

head=ptr->next;

free(ptr);

}

}

void delete\_end()

{

struct node \*ptr, \*cptr;

ptr=head;

while(ptr->next!=NULL)

{

cptr=ptr;

ptr=ptr->next;

}

cptr->next=NULL;

printf("\n deleted element is :%d",ptr->data);

free(ptr);

}

void delete\_info()

{

struct node \*ptr,\*cptr;

int d;

if(head==NULL)

printf("\n LINKED LIST UNDERFLOW\n");

else

{

ptr=head;

printf("\n Enter node info to be deleted:");

scanf("%d",&d);

while(ptr->data!=d)

{

cptr=ptr;

ptr=ptr->next;

}

cptr->next=ptr->next;

printf("\n deleted element is :%d",ptr->data);

free(ptr);

}

}

void display()

{

struct node \*ptr;

ptr=head;

if(head==NULL)

printf("\n LINKED LIST IS EMPTY\n");

else

{

while(ptr!=NULL)

{

printf(" %d->",ptr->data);

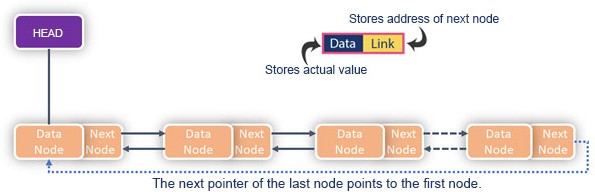
ptr=ptr->next;

}

}

}

**Circular Linked List:** Circular linked list is a linked list where all nodes are connected to form a circle. Simply, every node has the address of next node; putting the address of the first node in the last node makes it circular linked list. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



**//IMLEMENTAION OF CIRCULAR LINKED LIST ADT**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*next;

};

struct node \*head;

void create();

void insert\_begin();

void insert\_after();

void insert\_end();

void delete\_begin();

void delete\_info();

void delete\_end();

void display();

void main()

{

int ch;

system("clear");

while(1)

{

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n SINGLE LINKED LIST ADT OPERATIONS ARE:\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n\t1.CREATE");

printf("\n\t2.INSERTION AT THE BEGINNING");

printf("\n\t3.INSERTION AFTER THE GIVEN INFO:");

printf("\n\t4.INSERTION AT THE END");

printf("\n\t5.DELETION AT THE BEGINNING");

printf("\n\t6.DELETION OF THE GIVEN INFO:");

printf("\n\t7.DELETION AT THE END");

printf("\n\t8.DISPLAY");

printf("\n\t9.EXIT");

printf("\n Enter ur choice:");

scanf("%d",&ch);

switch(ch)

{

case 1: create();

break;

case 2: insert\_begin();

break;

case 3: insert\_after();

break;

case 4: insert\_end();

break;

case 5: delete\_begin();

break;

case 6: delete\_info();

break;

case 7: delete\_end();

break;

case 8: display();

break;

case 9: exit(0);

break;

default: printf("\n wrong choice\n");

}

}

}

void create()

{

struct node \*ptr,\*cptr;

int c;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter first node information:");

scanf("%d",&ptr->data);

head=ptr;

printf("\n Enter 0/1 for more nodes:");

scanf("%d",&c);

while(c==1)

. {

cptr=(struct node\*)malloc(sizeof(struct node));

ptr->next=cptr;

ptr=cptr;

printf("\n Enter next node information:");

scanf("%d",&cptr->data);

printf("\n Enter 0/1 for more nodes:");

scanf("%d",&c);

}

ptr->next=head;

}

void insert\_begin()

{

struct node \*ptr,\*cptr;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter node information to be inserted:");

scanf("%d",&ptr->data);

cptr=head;

while(cptr->next!=head)

{cptr=cptr->next;}

ptr->next=head;

head=ptr;

cptr->next=head;

}

void insert\_end()

{

struct node \*ptr, \*cptr;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter node information to be inserted:");

scanf("%d",&ptr->data);

cptr=head;

while(cptr->next!=head)

cptr=cptr->next;

cptr->next=ptr;

ptr->next=head;

}

void insert\_after()

{

struct node \*ptr, \*cptr;

int d;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter node information to be inserted:");

scanf("%d",&ptr->data);

printf("\n Enter node info after which U want to insert:");

scanf("%d",&d);

cptr=head;

while(cptr->data!=d)

cptr=cptr->next;

ptr->next=cptr->next;

cptr->next=ptr;

}

void delete\_begin()

{

struct node \*ptr,\*cptr;

if(head==NULL)

printf("\n CIRCULAR LINKED LIST UNDERFLOW\n");

else

{

ptr=head;

cptr=head;

printf("\n deleted element is :%d",ptr->data);

while(cptr->next!=head)

{cptr=cptr->next;}

head=ptr->next;

free(ptr);

cptr->next=head;

}

}

void delete\_end()

{

struct node \*ptr, \*cptr;

if(head==NULL)

printf("\n CIRCULAR LINKED LIST UNDERFLOW\n");

else

{

ptr=head;

while(ptr->next!=head)

{

cptr=ptr;

ptr=ptr->next;

}

cptr->next=head;

printf("\n deleted element is :%d",ptr->data);

free(ptr);

}

}

void delete\_info()

{

struct node \*ptr,\*cptr;

int d;

if(head==NULL)

printf("\n CIRCULAR LINKED LIST UNDERFLOW\n");

else

{

ptr=head;

printf("\n Enter node info to be deleted:");

scanf("%d",&d);

while(ptr->data!=d)

{

cptr=ptr;

ptr=ptr->next;

}

cptr->next=ptr->next;

printf("\n deleted element is :%d",ptr->data);

free(ptr);

}

}

void display()

{

struct node \*ptr;

if(head==NULL)

printf("\n LINKED LIST IS EMPTY\n");

else

{

ptr=head;

do

{

printf(" %d->",ptr->data);

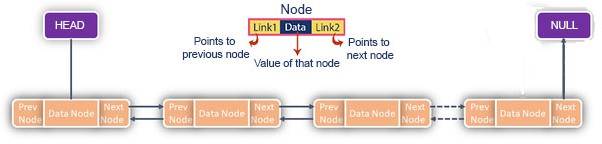
ptr=ptr->next;

} while(ptr!=head);

}

}

**Doubly Linked List:** Doubling Linked Lists have two address nodes, in particular, one node serves the purpose of directing to the next node while the other one points to the previous node. A linked list that has only next links is a singly linked list. Adding previous links makes a doubly linked list. Item navigation is both forward and backward.



//IMLEMENTAION OF DOUBLE LINKED LIST ADT

#include<stdio.h>

#include<stdlib.h>

struct node

{

struct node \*prev;

int data;

struct node \*next;

};

struct node \*head;

void create();

void insert\_begin();

void insert\_after();

void insert\_end();

void delete\_begin();

void delete\_info();

void delete\_end();

void display();

void main()

{

int ch;

system("clear");

while(1)

{

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n DOUBLE LINKED LIST ADT OPERATIONS ARE:\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n\t1.CREATE");

printf("\n\t2.INSERTION AT THE BEGINNING");

printf("\n\t3.INSERTION AFTER THE GIVEN INFO:");

printf("\n\t4.INSERTION AT THE END");

printf("\n\t5.DELETION AT THE BEGINNING");

printf("\n\t6.DELETION THE GIVEN INFO:");

printf("\n\t7.DELETION AT THE END");

printf("\n\t8.DISPLAY");

printf("\n\t9.EXIT");

printf("\n Enter ur choice:");

scanf("%d",&ch);

switch(ch)

{

case 1: create();

break;

case 2: insert\_begin();

break;

case 3: insert\_after();

break;

case 4: insert\_end();

break;

case 5: delete\_begin();

break;

case 6: delete\_info();

break;

case 7: delete\_end();

break;

case 8: display();

break;

case 9: exit(0);

break;

default: printf("\n wrong choice\n");

}

}

}

void create()

{

struct node \*ptr,\*cptr;

int c;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter first node information:");

scanf("%d",&ptr->data);

ptr->prev=NULL;

head=ptr;

printf("\n Enter 0/1 for more nodes:");

scanf("%d",&c);

while(c==1)

{

cptr=(struct node\*)malloc(sizeof(struct node));

ptr->next=cptr;

cptr->prev=ptr;

ptr=cptr;

printf("\n Enter next node information:");

scanf("%d",&cptr->data);

printf("\n Enter 0/1 for more nodes:");

scanf("%d",&c);

}

ptr->next=NULL;

}

void insert\_begin()

{

struct node \*ptr;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter node information to be inserted:");

scanf("%d",&ptr->data);

ptr->prev=NULL;

ptr->next=head;

head=ptr;

}

void insert\_end()

{

struct node \*ptr, \*cptr;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter node information to be inserted:");

scanf("%d",&ptr->data);

cptr=head;

while(cptr->next!=NULL)

cptr=cptr->next;

cptr->next=ptr;

ptr->prev=cptr;

ptr->next=NULL;

}

void insert\_after()

{

struct node \*ptr, \*cptr;

int d;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter node information to be inserted:");

scanf("%d",&ptr->data);

printf("\n Enter node info after which U want to insert:");

scanf("%d",&d);

cptr=head;

while(cptr->data!=d)

cptr=cptr->next;

(cptr->next)->prev=ptr;

ptr->prev=cptr;

ptr->next=cptr->next;

cptr->next=ptr;

}

void delete\_begin()

{

struct node \*ptr;

if(head==NULL)

printf("\n DOUBLE INKED LIST UNDERFLOW\n");

else

{

ptr=head;

printf("\n deleted element is :%d",ptr->data);

head=ptr->next;

head->prev=NULL;

free(ptr);

}

}

void delete\_end()

{

struct node \*ptr, \*cptr;

if(head==NULL)

printf("\n DOUBLE INKED LIST UNDERFLOW\n");

else

{

ptr=head;

while(ptr->next!=NULL)

{

cptr=ptr;

ptr=ptr->next;

}

cptr->next=NULL;

printf("\n deleted element is :%d",ptr->data);

free(ptr);

}

}

void delete\_info()

{

struct node \*ptr,\*cptr;

int d;

if(head==NULL)

printf("\n DOUBLE LINKED LIST UNDERFLOW\n");

else

{

ptr=head;

printf("\n Enter node info to be deleted:");

scanf("%d",&d);

while(ptr->data!=d)

{

cptr=ptr;

ptr=ptr->next;

}

cptr->next=ptr->next;

(ptr->next)->prev=cptr;

printf("\n deleted element is :%d",ptr->data);

free(ptr);

}

}

void display()

{

struct node \*ptr;

ptr=head;

if(head==NULL)

printf("\n DOUBLE LINKED LIST IS EMPTY\n");

else

{

while(ptr!=NULL)

{

printf(" %d<->",ptr->data);

ptr=ptr->next;

}

}

}

**STACK IMPLEMENTATION USING LINKED LIST:**

//IMLEMENTAION OF STACK ADT USING SINGLE LINKED LIST

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*next;

};

struct node \*top;

void push();

void pop();

void display();

void main()

{

int ch;

system("clear");

while(1)

{

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n STACK ADT USING SLL OPERATIONS ARE:\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n\t1.PUSH");

printf("\n\t2.POP");

printf("\n\t3.DISPLAY");

printf("\n\t4.EXIT");

printf("\n Enter ur choice:");

scanf("%d",&ch);

switch(ch)

{

case 1: push();

break;

case 2: pop();

break;

case 3: display();

break;

case 4: exit(0);

break;

default: printf("\n wrong choice\n");

}

}

}

void push()

{

struct node \*ptr;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter element to be inserted:");

scanf("%d",&ptr->data);

ptr->next=top;

top=ptr;

}

void pop()

{

struct node \*ptr;

if(top==NULL)

printf("\n STACK UNDERFLOW\n");

else

{

ptr=top;

printf("\n deleted element is :%d\n",ptr->data);

top=ptr->next;

free(ptr);

}

}

void display()

{

struct node \*ptr;

ptr=top;

if(top==NULL)

printf("\n STACK IS EMPTY\n");

else

{

while(ptr!=NULL)

{

printf(" %d->",ptr->data);

ptr=ptr->next;

}

}

}

**//IMLEMENTAION OF QUEUE ADT USING LINKED LIST**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*next;

};

struct node \*front,\*rear;

void insert();

void del();

void display();

void main()

{

int ch;

struct node \*ptr;

system("clear");

while(1)

{

printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n QUEUE ADT USING SLL OPERATIONS ARE:\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n\t1.INSERT");

printf("\n\t2.DELELE");

printf("\n\t3.DISPLAY");

printf("\n\t4.EXIT");

printf("\n Enter ur choice:");

scanf("%d",&ch);

switch(ch)

{

case 1: insert();

break;

case 2: del();

break;

case 3: display();

break;

case 4: exit(0);

break;

default: printf("\n wrong choice\n");

}

}

}

void insert()

{

struct node \*ptr;

ptr= (struct node\*)malloc(sizeof(struct node));

printf("\n Enter node information:");

scanf("%d",&ptr->data);

if(front==NULL)

{

front=ptr;

rear=ptr;

front->next=NULL;

rear->next=NULL;

}

else

{

rear->next=ptr;

rear=ptr;

rear->next=NULL;

}

}

void del()

{

struct node \*ptr;

if(front==NULL)

{

printf("\n QUEUE UNDERFLOW\n");

count=1;

}

else

{

ptr=front;

printf("\n deleted element is:%d",ptr->data);

front=ptr->next;

free(ptr);

}

}

void display()

{

struct node \*ptr;

ptr=front;

if(front==NULL)

printf("\n QUEUE IS EMPTY\n");

else

{

while(ptr!=NULL)

{

printf(" %d->",ptr->data);

ptr=ptr->next;

}

}

}

# DIFFERENTIATE STACK AND QUEUE?

STACKS QUEUE

|  |  |
| --- | --- |
| Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list. | Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list. |
| Insertion and deletion in stacks takes place only from one end of the list called the top. | Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list. |
| Insert operation is called push operation. | Insert operation is called enqueue operation. |
| Delete operation is called pop operation. | Delete operation is called dequeue operation. |
| In stacks we maintain only one pointer to access the list, called the top, which | In queues we maintain two pointers to access the list. The front pointer always points to the first element |

always points to the last element present in the list.

inserted in the list and is still present, and the rear pointer always points to the last inserted element.