

## RECURSION:

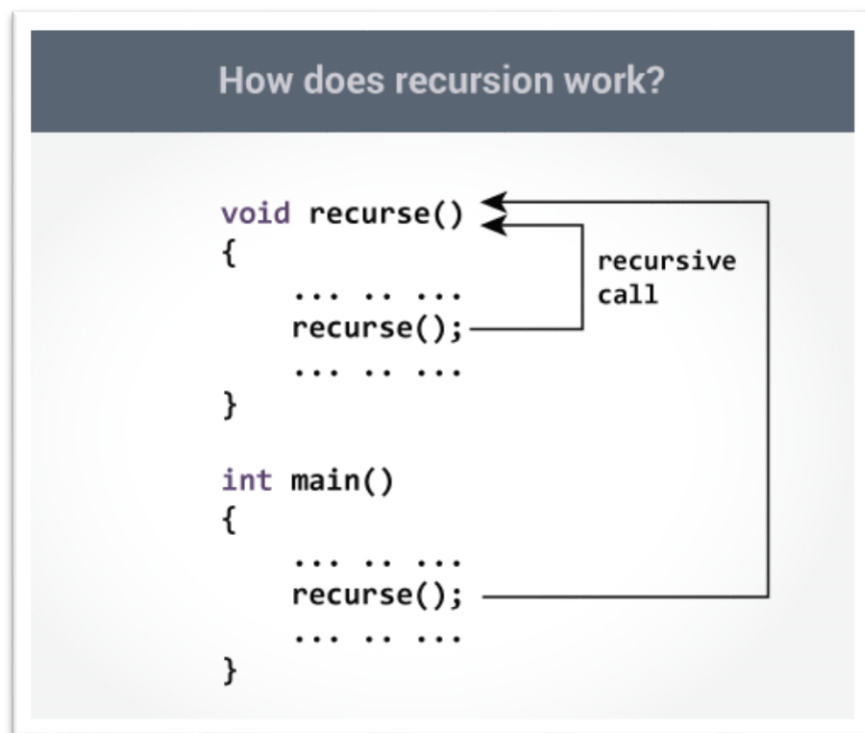
Recursion is a technique followed by recursive function, function that call itself until some condition gets false.

Syntax:

```
data_type <function_name> ()  
{  
Statement(s);  
function_name();  
}
```

eg.)

```
void recursion()  
{  
.  
.  
recursion();  
}
```



While using recursion, one needs to be careful to define an exit condition from the function, otherwise the recursive function can go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, powering a number etc.

Let's see a code of recursive function:

```
#include <stdio.h>
int factorial(unsigned int i)
{
    if(i <= 1)
        return 1;

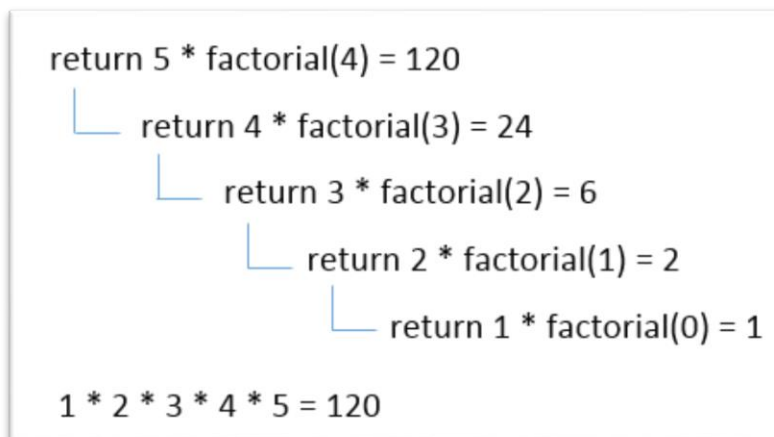
    return i * factorial(i - 1);
}
```

```
int main() {
    int i = 5;
    printf("Factorial of %d is %d\n", i, factorial(i));
    return 0;
}
```

Factorial of 5 is 120

Process returned 0 (0x0) execution time : 0.031 s  
Press any key to continue.

This is how in each run loop will work.



#### Advantages

Reduce unnecessary calling of function.

Through Recursion one can solve problems in easy way while its iterative solution is very big and complex.

#### Disadvantages

Recursive solution is always logical and it is very difficult to debug and understand.

In recursive we must have if statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.

Recursion takes a lot of memory space.

Recursion uses more processor time.

# Memoization

The general technique of storing already-calculated values for a function in a cache is called memoization.

In simpler term, Memoization is a term describing an optimization technique where we save previously computed results, and return the saved result when the same computation is needed again.

The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need solution to a sub problem, we first look into the lookup table. If the required value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

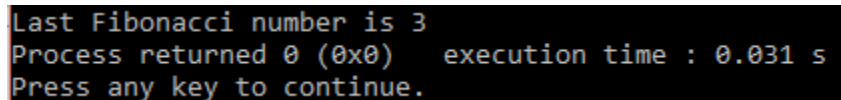
Let's see a small code of Fibonacci series:

```
#include<stdio.h>
#define NIL -1
#define MAX 100
int lookup[MAX];

void _initialize()    /* Function to initialize NIL values in lookup table */
{
    int i;
    for (i = 0; i < MAX; i++)
        lookup[i] = NIL;
}

int fib(int n)        /* function for nth Fibonacci number */
{
    if (lookup[n] == NIL)
    {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n-1) + fib(n-2);
    }
    return lookup[n];
}

int main ()
{
    int n = 4;
    _initialize();
    printf("Last Fibonacci number is %d ", fib(n));
    return 0;
}
```



A terminal window with a black background and green text. It displays the output of the program: "Last Fibonacci number is 3", followed by "Process returned 0 (0x0) execution time : 0.031 s", and "Press any key to continue.".

In this code the last element of Fibonacci series is printed, for series up to 5<sup>th</sup> element is 0, 1, 1, 2, and 3. So the last element is 3.

We used lookup array which stores the result for each call of function. This lookup array will give the values of previous result when needed, in other terms it memorized the each result.

# STACK

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out).

Mainly the following three basic operations are performed in the stack:

**Push:** Adding an item in the stack is called push. If the stack is full, then it is said to be an Overflow condition.

**Pop:** Removing an item from the stack is called pop. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

`peek()` – get the top data element of the stack, without removing it.

`isFull()` – check if stack is full.

`isEmpty:` Returns true if stack is empty, else false.

**NOTE!** Top about the stack, in each time value get stores in top and top value is increases by 1 and if the top is NULL or -1 that means stack is empty.

The real time example of stack is tower of plate in kitchen, the plate at the top will be taken first to wash though last plate came first. Also playing cards deck, the card at the top is taken while playing, and so many other problems.

How to perform operation on stack:

Using array

Using link list (discussed later).

## 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 –

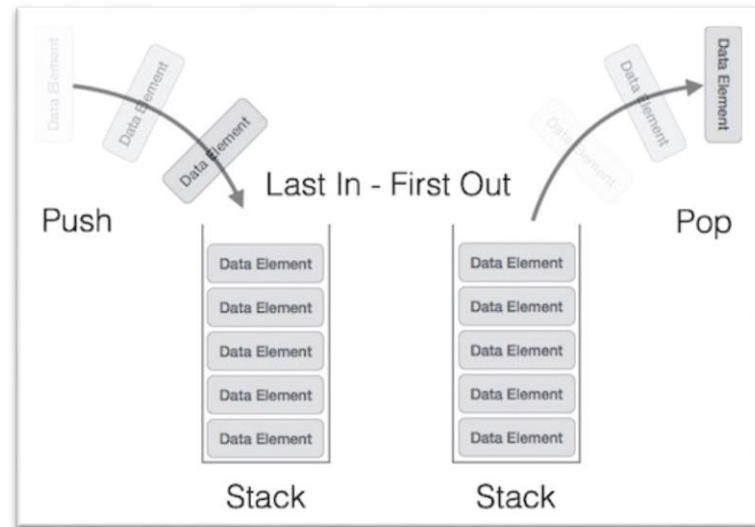
Step 1 – Checks if the stack is full.

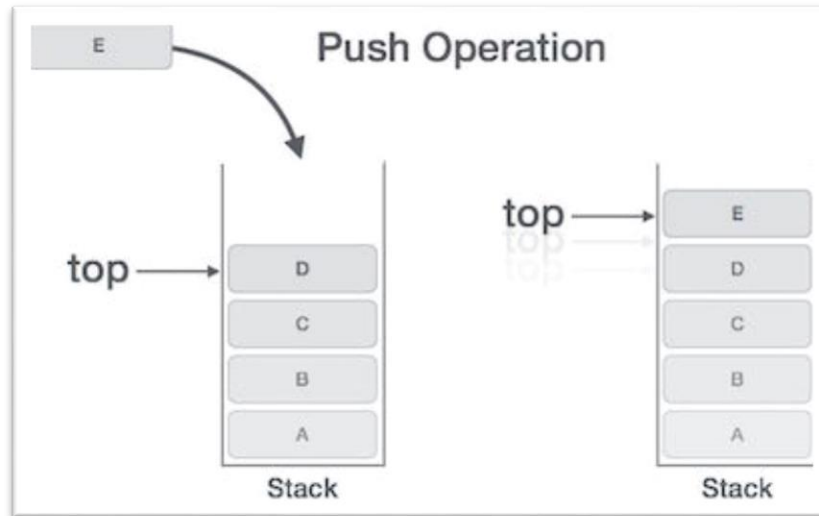
Step 2 – If the stack is full, produces an error and exit.

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.





If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

#### 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. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

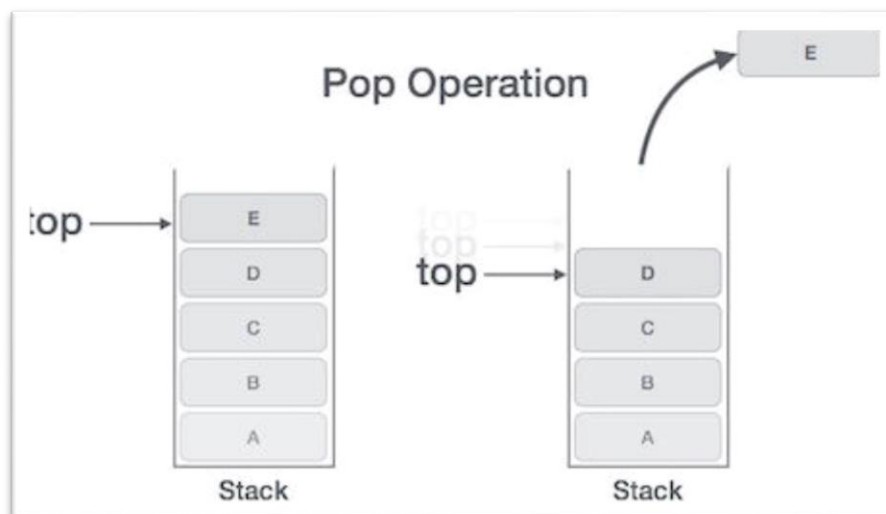
Step 1 – Checks if the stack is empty.

Step 2 – If the stack is empty, produces an error and exit.

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.



Let's discuss a code using array:

```
#include<stdio.h>
#define SIZE 10
void push(int);
void pop();
void display();

int stack[SIZE], top = -1;
void main()
{
    int value, choice;
    while(1)
    {
        printf("\n\n***** MENU *****\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1: printf("Enter the value to be insert: ");
                    scanf("%d",&value);
                    push(value);
                    break;
            case 2: pop();
                    break;
            case 3: display();
                    break;
            case 4: exit(0);
            default: printf("\nWrong selection!!! Try again!!!");
        }
    }
}

void push(int value){
    if(top == SIZE-1)
        printf("\nStack is Full!!! Insertion is not possible!!!");
    else{
        top++;
        stack[top] = value;
        printf("\nInsertion success!!!");
    }
}

void pop(){
    if(top == -1)
        printf("\nStack is Empty!!! Deletion is not possible!!!");
    else{
        printf("\nDeleted : %d", stack[top]);
        top--;
    }
}
```

```
***** MENU *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 55

Insertion success!!!

***** MENU *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 90

Insertion success!!!

***** MENU *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3

Stack elements are:
90
55

***** MENU *****
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2

Deleted : 90
```

```

    }
}
void display(){
    if(top == -1)
        printf("\nStack is Empty!!!");
    else{
        int i;
        printf("\nStack elements are:\n");
        for(i=top; i>=0; i--)
            printf("%d\n",stack[i]);
    }
}
}

```

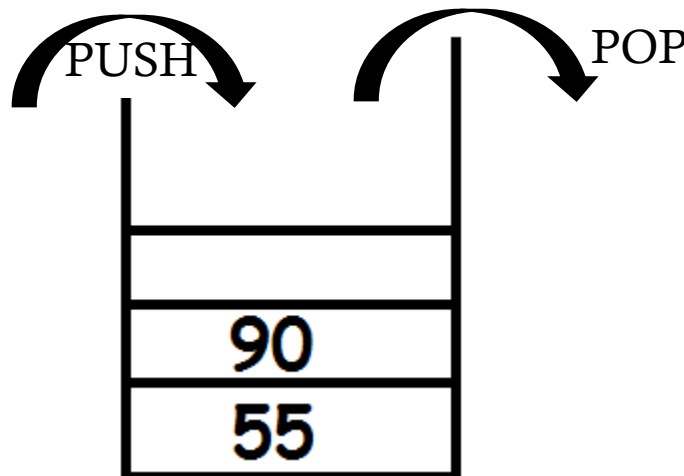
Explanation:

**For push:** For pushing first we will check whether the top is equal to size of stack if so then the condition will be of overflow and no item will be added.

If the condition is false then top is incremented and the item will be added in stack.

**For pop:** For popping first it is checked that whether top= -1 if so then the condition will be of underflow that means stack is empty and no item will be popped.

If the condition is false then the item at the top will be deleted and value of top will be decremented. 90 was at the top therefore it got popped out.



#### Applications of stack:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, Photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and Sudoku solver.



## INFIX TO POSTFIX CONVERSION:

Conversion of infix expressions into postfix is the common example of stack.

What is infix expression? Any expression normally written with the operator is in between the two operands that it is working on is infix expression. eg.)  $(a+b)*(c-d)$ .

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

Let's see the code for conversion:

```
#include<stdio.h>
char stack[20];
int top = -1;
void push(char x)
{
    stack[++top] = x;
}
char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}
int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
}
main()
{
    char exp[20];
    char *e, x;
    printf("Enter the expression :: ");
    scanf("%s",exp);
    e = exp;
    printf("Postfix Expression:");
    while(*e != '\0')
    {
        if(isalnum(*e))
```

```

        printf("%c",*e);
    else if(*e == '(')
        push(*e);
    else if(*e == ')')
    {
        while((x = pop()) != '(')
            printf("%c", x);
    }
    else
    {
        while(priority(stack[top]) >= priority(*e))
            printf("%c",pop());
        push(*e);
    }
    e++;
}
while(top != -1)
{
    printf("%c",pop());
}
printf("\n");
}

```

```

Enter the expression :: a+b-c*d
Postfix Expression:ab+cd*-
Process returned 10 (0xA)   execution time : 8.349 s
Press any key to continue.

```

### Algorithm

- Scan the infix expression from left to right.
- If the scanned character is an operand, output it.
- Else,
  - If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty), push it.
  - Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.
- If the scanned character is an '(', push it to the stack.
- If the scanned character is an ')', pop and output from the stack until an '(' is encountered.
- Repeat steps 2-6 until infix expression is scanned.
- Pop and output from the stack until it is not empty

## **EXERCISE:**

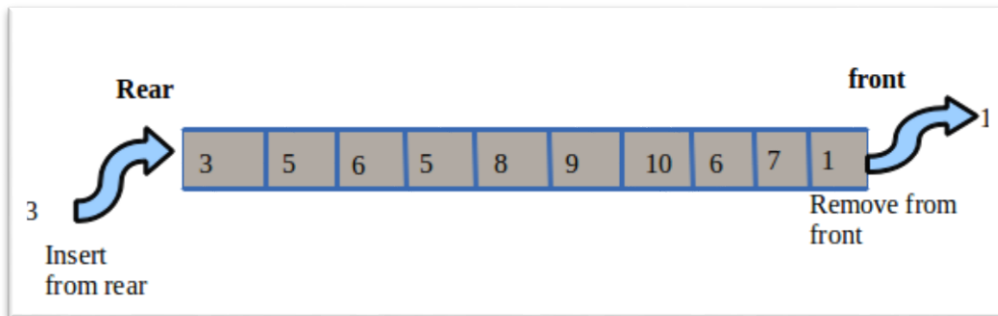
1. Reverse a string given by user using Stack
2. Implement two stacks in an array.
3. Check for balanced parentheses in an expression. e.g.) (A+B)\*C will be accepted but, (a+b(c-d) will not be accepted.
4. Next Greater Element in the stack.
5. Reverse a stack using recursion.
6. Sort a stack using recursion.
7. Design a stack with operations on middle element. e.g.) add element in middle, delete element on middle.
8. How to efficiently implement k stacks in a single array?
9. Length of the longest valid substring.
10. Delete consecutive same words in a sequence. e.g.) abcccde should become abcde.
11. Find maximum sum possible equal sum of three stacks.
12. Infix to Prefix Conversion using Stack.
13. Evaluation of Postfix Expression.
14. Evaluation of Prefix expression.
15. Implement Stack using Queues.
16. Find next Smaller of next Greater in an array
17. Form minimum number from given sequence.

[Click here](#) for logics of questions.

# QUEUE

Queue is an abstract data structure, somewhat similar to Stacks.

Queue is a linear structure which 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.

Queue is opened from both the side like pipe, one end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

NOTE: The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Operations on Queue: Mainly the following four basic operations are performed on queue:

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Front: Get the front item from queue.

Rear: Get the last item from queue.

Enqueue Operation:

Queues maintain two data pointers, front and rear. The following steps should be taken to enqueue (insert) data into a queue –

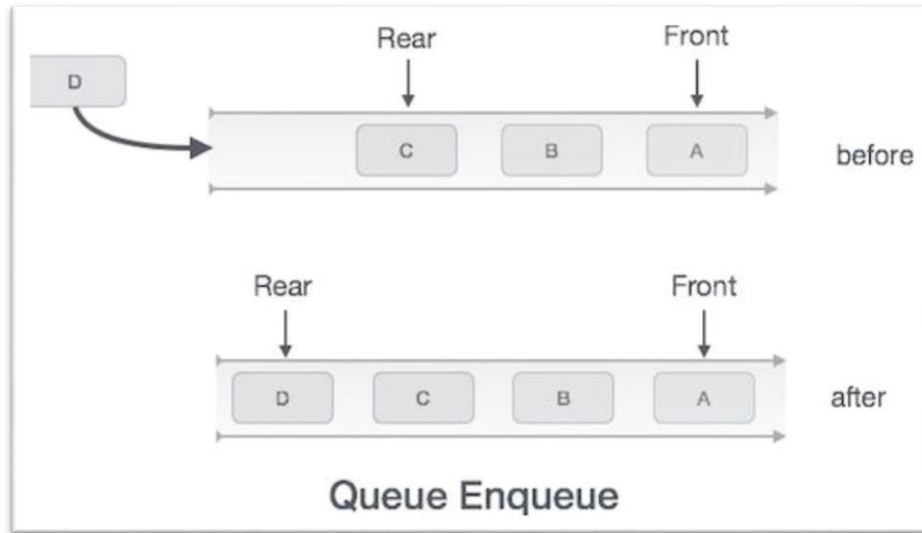
Step 1 – Check if the queue is full.

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.



#### 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 –

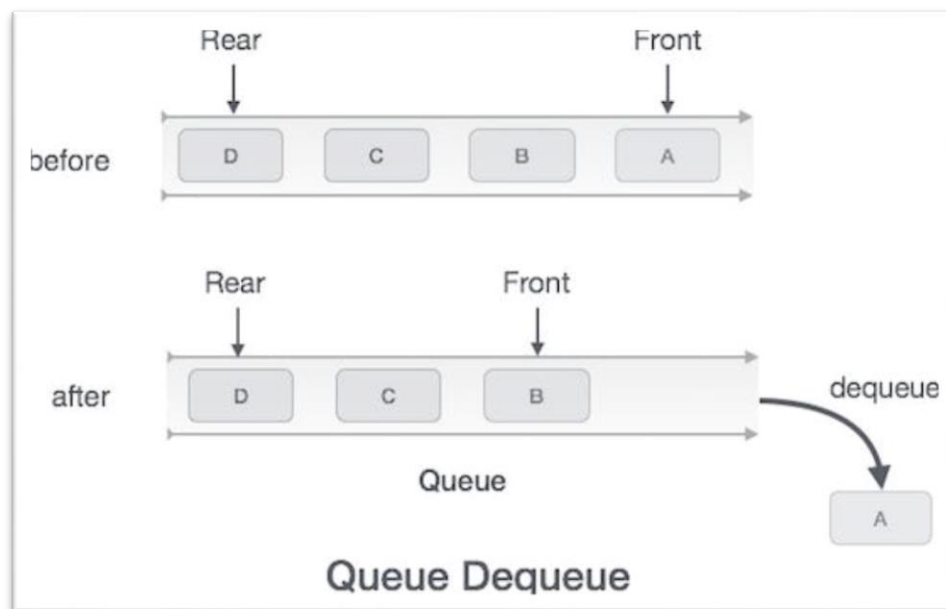
Step 1 – Check if the queue is empty.

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.



Queue can be implemented using two ways:

- Using array
- Using link list (discussed later)

Let's see a code:

```
#include<stdio.h>
#define SIZE 10
void enQueue(int);
void deQueue();
void display();

int queue[SIZE], front = -1, rear = -1;

void main()
{
    int value, choice;
    while(1){
        printf("\n\n***** MENU *****\n");
        printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1: printf("Enter the value to be insert: ");
                    scanf("%d",&value);
                    enQueue(value);
                    break;
            case 2: deQueue();
                    break;
            case 3: display();
                    break;
            case 4: exit(0);
            default: printf("\nWrong selection!!! Try
again!!!");
        }
    }
}

void enQueue(int value){
    if(rear == SIZE-1)
        printf("\nQueue is Full!!! Insertion is not possible!!!");
    else{
        if(front == -1)
            front = 0;
        rear++;
        queue[rear] = value;
        printf("\nInsertion success!!!");
    }
}

void deQueue(){
    if(front == rear)
```

```
***** MENU *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 66

Insertion success!!!

***** MENU *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 88

Insertion success!!!

***** MENU *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3

Queue elements are:
66      88

***** MENU *****
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 2

Deleted : 66
```

```

    printf("\nQueue is Empty!!! Deletion is not possible!!!");
else{
    printf("\nDeleted : %d", queue[front]);
    front++;
    if(front == rear)
        front = rear = -1;
}
}
void display(){
    if(rear == -1)
        printf("\nQueue is Empty!!!");
    else{
        int i;
        printf("\nQueue elements are:\n");
        for(i=front; i<=rear; i++)
            printf("%d\t",queue[i]);
    }
}
}

```

Explanation:

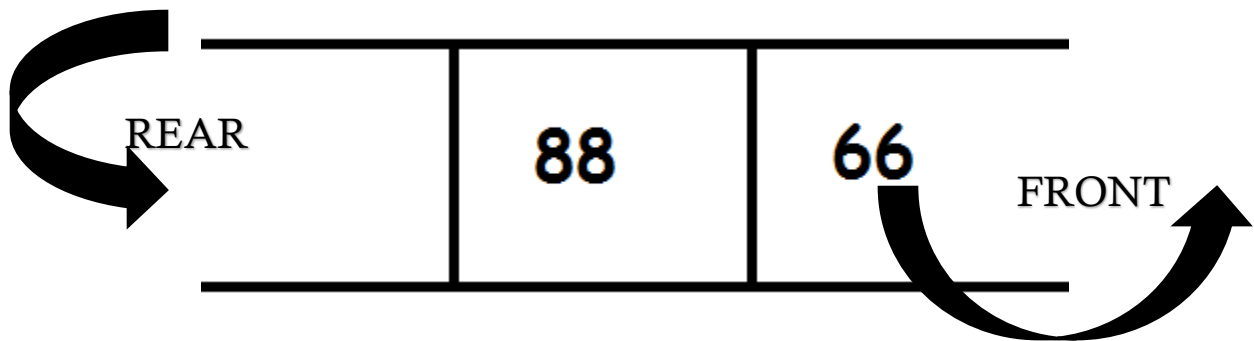
**For enqueue:** - The first condition we checked for enqueue is checking whether rear at the last, i.e. size -1 if so then it will be overflow condition and no item will be entered.

If not then rear position is incremented and an element is added. i.e. 66.

Again 88 was entered.

**For dequeue:** - The front was checked whether it not at position of rear, that means queue is empty and it is the condition of underflow.

If not then first added element, i.e. 66 is deleted from queue and front is incremented.

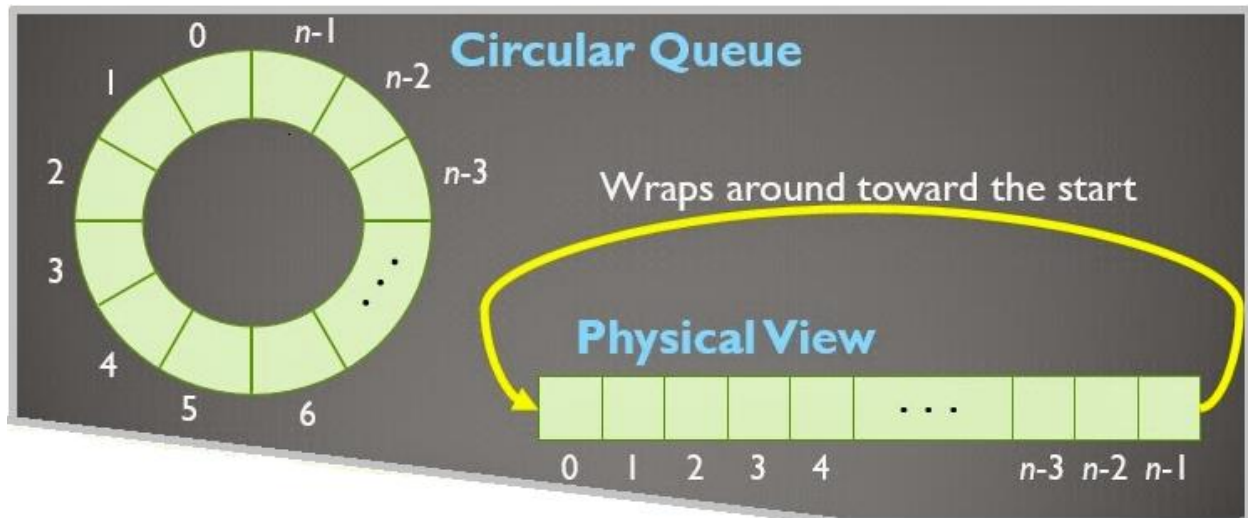


### Applications of Queue:

- Queue is used when things don't have to be processed immediately, but have to be processed in **First In First Out** order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.
- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

## CIRCULAR QUEUE:

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called '**Ring Buffer**'.



**NOTE:** In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we cannot insert the next element even if there is a space in front of queue.

Operations on Circular Queue:

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

**Steps:**

- Check whether queue is Full – Check  $((\text{rear} == \text{SIZE}-1 \ \&\& \ \text{front} == 0) \ || \ (\text{rear} == \text{front}-1))$ .
- If it is full then display Queue is full. If queue is not full then, check if  $(\text{rear} == \text{SIZE} - 1 \ \&\& \ \text{front} != 0)$  if it is true then set  $\text{rear}=0$  and insert element.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

**Steps:**

- Check whether queue is Empty means check  $(\text{front} == -1)$ .
- If it is empty then display Queue is empty. If queue is not empty then step 3



- Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

Circular queue can be implemented

- Using array
- Using link list

Let's see code using array:

```
#include<stdio.h>
#define SIZE 5
void enQueue(int);
void deQueue();
void display();

int cQueue[SIZE], front = -1, rear = -1;

void main()
{
    int choice, value;
    while(1){
        printf("\n***** MENU *****\n");
        printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1: printf("\nEnter the value to be insert: ");
                    scanf("%d",&value);
                    enQueue(value);
                    break;
            case 2: deQueue();
                    break;
            case 3: display();
                    break;
            case 4: exit(0);
            default: printf("\nPlease select the correct choice!!!\n");
        }
    }
}

void enQueue(int value)
{
    if((front == 0 && rear == SIZE - 1) || (front == rear+1))
        printf("\nCircular Queue is Full! Insertion not possible!!!\n");
    else{
        if(rear == SIZE-1 && front != 0)
```

```
***** MENU *****
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the value to be insert: 3

Insertion Success!!!

***** MENU *****
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the value to be insert: 6

Insertion Success!!!

***** MENU *****
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3

Circular Queue Elements are :
3      6
***** MENU *****
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2

Deleted element : 3
```

```

        rear = -1;
        cQueue[++rear] = value;
        printf("\nInsertion Success!!!\n");
        if(front == -1)
            front = 0;
    }
}
void deQueue()
{
    if(front == -1 && rear == -1)
        printf("\nCircular Queue is Empty! Deletion is not possible!!!\n");
    else{
        printf("\nDeleted element : %d\n",cQueue[front++]);
        if(front == SIZE)
            front = 0;
        if(front-1 == rear)
            front = rear = -1;
    }
}
void display()
{
    if(front == -1)
        printf("\nCircular Queue is Empty!!!\n");
    else{
        int i = front;
        printf("\nCircular Queue Elements are : \n");
        if(front <= rear){
            while(i <= rear)
                printf("%d\t",cQueue[i++]);
        }
        else{
            while(i <= SIZE - 1)
                printf("%d\t", cQueue[i++]);
            i = 0;
            while(i <= rear)
                printf("%d\t",cQueue[i++]);
        }
    }
}
}

```

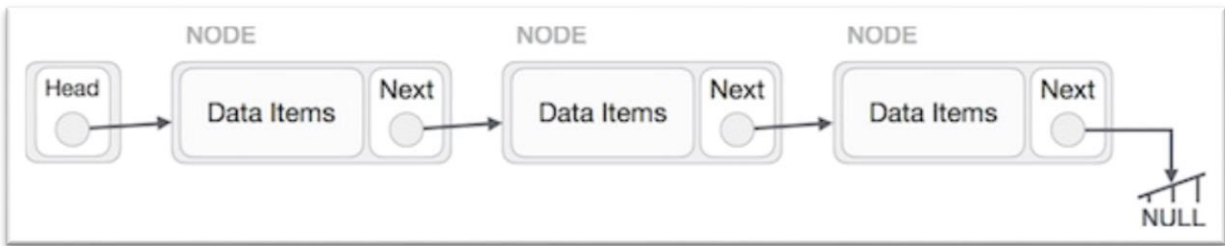
So if the queue is full and some elements are deleted from front then we can add more values in queue.

### **Applications:**

- **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
- **Traffic system:** In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
- **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

# Link List

Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.



A linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node".

The list gets its overall structure by using pointers to connect all its nodes together like the links in a chain.

Each node contains two fields: a "data" field to store whatever element type the list by user, and a "next" field which is a pointer used to link one node to the next node.

Each node is allocated in the heap with a call to `malloc ()`, so the node memory continues to exist until it is explicitly deallocated with a call to `free ()`.

Following are the important points to be considered:

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

## Types of Linked List:

Following are the various types of linked list:

- Simple Linked List – Item navigation is forward only.
- Doubly Linked List – Items can be navigated forward and backward.
- Circular Linked List – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Linked list provides following two advantages over arrays:-

- Dynamic size: Size of the list can be increased during runtime, no static size is required.
- Ease of insertion/deletion: Insertion and deletion can be done at any point, unlike array if needed to add an element in starting the whole array is required get shifted which increases the cost.

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

Let's see operation in Simple linked list:

```
#include<stdlib.h>
#include<stdio.h>
void create();
void display();
void insert_begin();
void delete_begin();
struct node
{
    int info;
    struct node *next;
};
struct node *start=NULL;
int main()
{
    int choice;
    while(1){
        printf("\nMENU \n");
        printf("\n 1.Create");
        printf("\n 2.Display ");
        printf("\n 3.Insert at the beginning");
        printf("\n 4.Delete from beginning");
        printf("\n 5.Exit \n");
        printf("\nEnter your choice:\t");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                create();
                break;
            case 2:
                display();
                break;
            case 3:
                insert_begin();
                break;
            case 4:
                delete_begin();
                break;
            case 5:
                exit(0);
                break;
            default:
                printf("\n Wrong Choice:\n");
                break;
        }
    }
}
```

```
MENU

1.Create
2.Display
3.Insert at the beginning
4.Delete from beginning
5.Exit

Enter your choice:1

Enter the data value for the node:      2

MENU

1.Create
2.Display
3.Insert at the beginning
4.Delete from beginning
5.Exit

Enter your choice:1

Enter the data value for the node:      4

MENU

1.Create
2.Display
3.Insert at the beginning
4.Delete from beginning
5.Exit

Enter your choice:2

The List elements are:=>2 =>4
MENU

1.Create
2.Display
3.Insert at the beginning
4.Delete from beginning
5.Exit
```

```

    }
    return 0;
} //end of main()
void create()
{
    struct node *temp,*ptr;
    temp=(struct node *)malloc(sizeof(struct node));
    if(temp==NULL)
    {
        printf("\nOut of Memory Space:\n");
        exit(0);
    }
    printf("\nEnter the data value for the node:\t");
    scanf("%d",&temp->info);
    temp->next=NULL;
    if(start==NULL)
    {
        start=temp;
    }
    else
    {
        ptr=start;
        while(ptr->next!=NULL)
        {
            ptr=ptr->next;
        }
        ptr->next=temp;
    }
} //end of create()
void display()
{
    struct node *ptr;
    if(start==NULL)
    {
        printf("\nList is empty:\n");
        return;
    }
    else
    {
        ptr=start;
        printf("\nThe List elements are:");
        while(ptr!=NULL)
        {
            printf("=>%d\t",ptr->info );
            ptr=ptr->next ;
        } //end of while
    }
}

```

```

1.Create
2.Display
3.Insert at the beginning
4.Delete from beginning
5.Exit

Enter your choice:3

Enter the data value for the node:0

MENU

1.Create
2.Display
3.Insert at the beginning
4.Delete from beginning
5.Exit

Enter your choice:2

The List elements are:=>0 =>2 =>4

MENU

1.Create
2.Display
3.Insert at the beginning
4.Delete from beginning
5.Exit

Enter your choice:4

The deleted element is :0

MENU

1.Create
2.Display
3.Insert at the beginning
4.Delete from beginning
5.Exit

Enter your choice:2

The List elements are:=>2 =>4

```

```

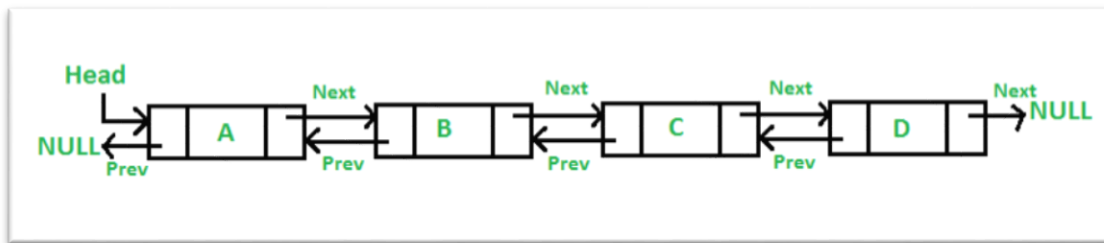
        }//end of else
    }//end of display()
void insert_begin()
{
    struct node *temp;
    temp=(struct node *)malloc(sizeof(struct node));
    if(temp==NULL)
    {
        printf("\nOut of Memory Space:\n");
        return;
    }
    printf("\nEnter the data value for the node:\t" );
    scanf("%d",&temp->info);
    temp->next =NULL;
    if(start==NULL)
    {
        start=temp;
    }
    else
    {
        temp->next=start;
        start=temp;
    }
}
} //end of insert_begin()
void delete_begin()
{
    struct node *ptr;
    if(ptr==NULL)
    {
        printf("\nList is Empty:\n");
        return;
    }
    else
    {
        ptr=start;
        start=start->next ;
        printf("\nThe deleted element is :%d\t",ptr->info);
        free(ptr);
    }
}
} //end of delete_begin()

```

In this list we entered the elements from the beginning of list and deleted from beginning. Notice one thing that the as the values get added on the size of list also increases at run time only.

Doubly link list (DLL):

A Doubly Linked List contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next node using its next link.
- Each link is linked with its previous node using its previous link.
- The last link carries a link as null to mark the end of the list.

#### Advantages over singly linked list:

- A DLL can be traversed in both forward and backward direction.
- The delete operation in DLL is more efficient if pointer to the node to be deleted is given. In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

#### Insertion Operation

The insertion operation at the beginning of a doubly linked list.

Example:

```
void insertFirst(int key, int data)
```

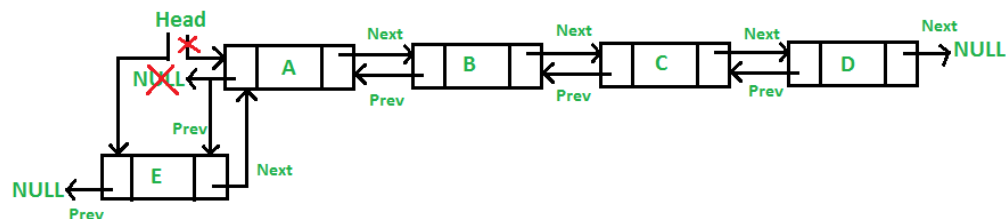
```
{
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;
```

```
if(isEmpty()) {
    //make it the last link
    last = link;
} else {
    //update first prev link
    head->prev = link;
}
```

```
//point it to old first link
link->next = head;
```

```
//point first to new first link
head = link;
```

```
}
```





## Deletion Operation

The deletion operation at the beginning of a doubly linked list.

Example

```
//delete first item
struct node* deleteFirst() {

    //save reference to first link
    struct node *tempLink = head;

    //if only one link
    if(head->next == NULL) {
        last = NULL;
    } else {
        head->next->prev = NULL;
    }

    head = head->next;

    //return the deleted link
    return tempLink;
}
```

## Insertion at the End of an Operation

The insertion operation at the last position of a doubly linked list.

Example

```
void insertLast(int key, int data)
{
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;

    if(isEmpty()) {
        //make it the last link
        last = link;
    } else {
        //make link a new last link
        last->next = link;

        //mark old last node as prev of new link
        link->prev = last;
    }

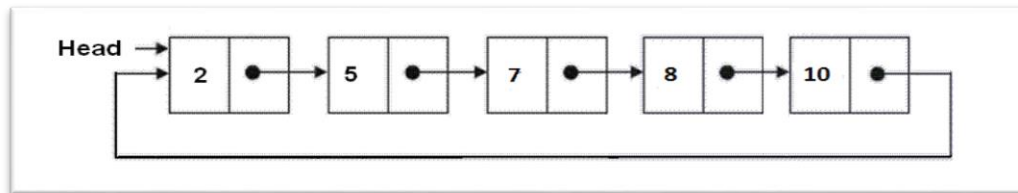
    //point last to new last node
    last = link;
}
```

### **CIRCULAR LINK LIST (CLL):-**

Circular linked list is a linked list where all nodes are connected to form a circle. The last node is not connected to NULL at the end.

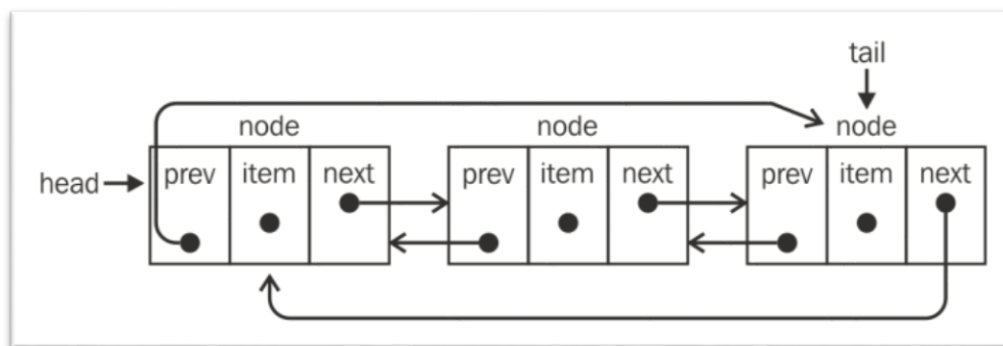
A circular linked list can be a singly circular linked list or doubly circular linked list.

In singly linked list, the next pointer of the last node points to the first node.



### **Doubly Circular Linked List:**

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



### **Advantages of Circular Linked Lists:-**

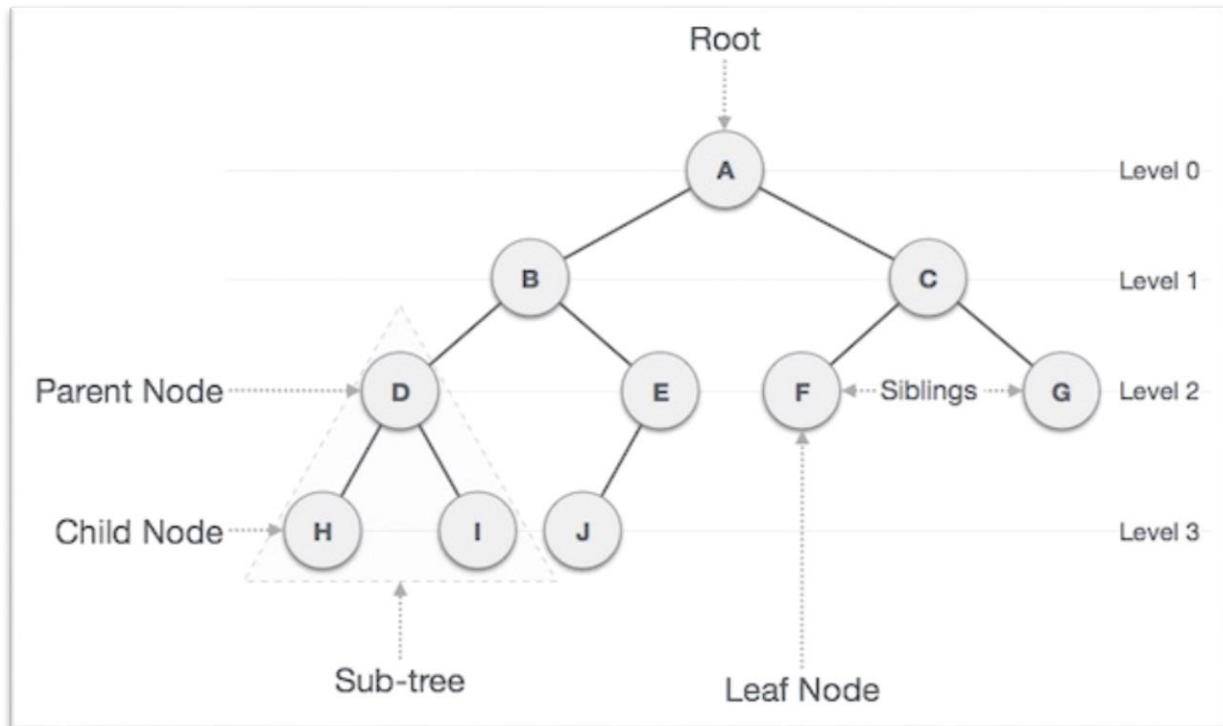
- Any node can be a starting point.
- We can traverse the whole list by starting from any point.
- Useful for implementation of queue.
- Circular lists are useful in applications to repeatedly go around the list e.g.) CPU scheduling.
- Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

# Tree

Trees are hierarchical data structures. A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. Two adjacent node having same parent is called sibling.

Connect the tree with your family tree you will understand much better.  
See the diagram to understand the relations.



Why tree?

1. To store information that naturally forms a hierarchy. For example, the file system on a computer:
2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.

# BINARY TREE

A tree whose elements have at most 2 children is called a binary tree, the left (node-> left) and right child (node->right).

A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.

A Tree node contains following parts.

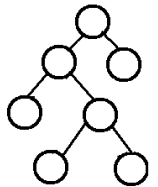
- Data
- Pointer to left child
- Pointer to right child

Syntax:

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
```

## Types of binary tree:

1. **Full Binary Tree:** A Binary Tree is full if every node has 0 or 2 children. Following are examples of full binary tree.

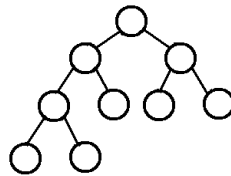


In a Full Binary, number of leaf nodes is number of internal nodes plus 1

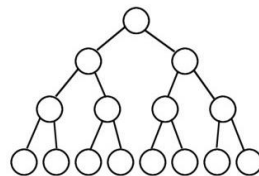
$$L = I + 1$$

Where L = Number of leaf nodes, I = Number of internal nodes

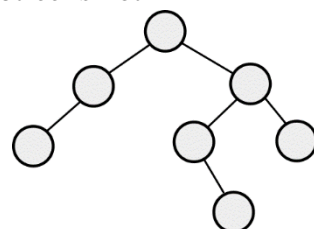
2. **Complete Binary Tree:** A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible.



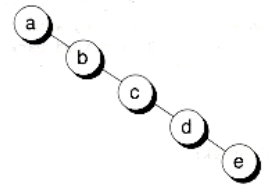
3. **Perfect Binary Tree:** A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level. A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has  $2^h - 1$  node.



4. **AVL Tree:** AVL tree is tree in which the difference between left and right subtree is not more than 1 for all nodes.



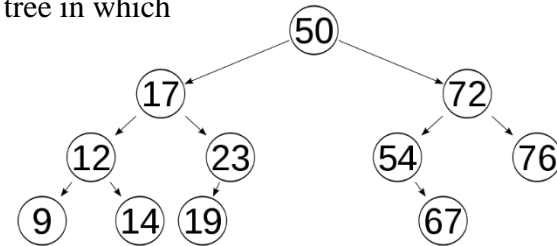
5. **A degenerate (or pathological) tree** A Tree where every internal node has one child. Such trees are performance-wise same as linked list.



6. **Binary search tree (BST)\***: Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

(\*Will be briefly discussed later)

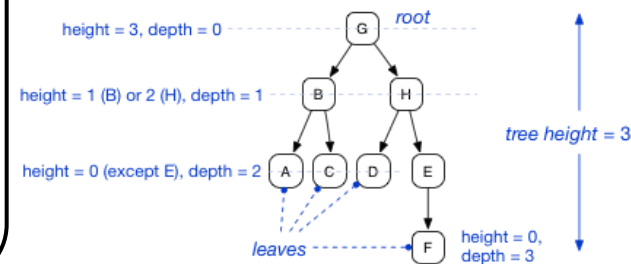


### Operations on tree:

- Insertion
- Deletion
- Traversal
  - Inorder Traversal (Left, Root, Right)
  - Preorder Traversal (Root, Left, Right)
  - Postorder Traversal (Left, Right, Root)
- Searching

### NOTE!

- Length of a path= number of edges
- Depth of a node N= length of path from root to N
- Height of node N= length of longest path from N to a leaf
- Depth of tree= depth of deepest node
- Height of tree= height of root



Let's try a code for simple insertion and traversal in Binary Tree:

```

#include<stdio.h>
typedef struct node
{
    int data;
    struct node *left;
    struct node *right;
} node;
  
```

```

node *create()
{
    node *p;
    int x;
    printf("Enter data(-1 for no data):");
    scanf("%d",&x);
    if(x==-1)
        return NULL;
    p=(node*)malloc(sizeof(node));
    p->data=x;
    printf("Enter left child of %d:\n",x);
    p->left=create();
    printf("Enter right child of %d:\n",x);
    p->right=create();
    return p;
}

void preorder(node *t)    //address of root
node is passed in t
{
    if(t!=NULL)
    {
        printf("\n%d",t->data);    //visit the root
        preorder(t->left);    //preorder traversal on left subtree
        preorder(t->right);    //preorder traversal om right subtree
    }
}

int main()
{
    node *root;
    root=create();
    printf("\nThe preorder traversal of tree is:\n");
    preorder(root);
    return 0;
}

```

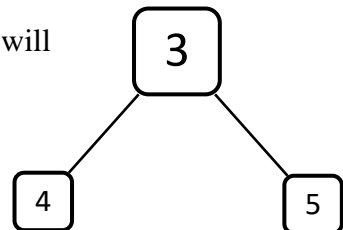
```

Enter data(-1 for no data):3
Enter left child of 3:
Enter data(-1 for no data):4
Enter left child of 4:
Enter data(-1 for no data):5
Enter left child of 5:
Enter data(-1 for no data):-1
Enter right child of 5:
Enter data(-1 for no data):-1
Enter right child of 4:
Enter data(-1 for no data):-1
Enter right child of 3:
Enter data(-1 for no data):-1

The preorder traversal of tree is:
3
4
5
Process returned 0 (0x0)   execution time : 38.739 s
Press any key to continue.

```

Notice in the program, the tree was like this. In preorder traversal first the root node, i.e. 3 is traversed then left node, i.e. 4 then right node, i.e. 5.  
Each time in insertion the size of the tree gets increases. If the tree is empty then it will have NULL value.



## BINARY SEARCH TREE

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.

The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

**Searching a key:** To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we check for right subtree of root node. Otherwise we check for left subtree.

Let's see a code of searching:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int info;
    struct node*left;
    struct node*right;
};

typedef struct node BST;
BST *LOC, *PAR;
void search(BST *root, int item)
{
    BST *save,*ptr;
    if (root == NULL)
    {
        LOC = NULL;
        PAR=NULL;
    }

    if (item == root -> info)

    {

        LOC = root;

        PAR = NULL;

        return;
    }
}
```

```

    }

    if (item < root->info)

    {

        save = root;

        ptr = root->left;

    }

    else

    {

        save = root;

        ptr = root -> right;

    }

    while( ptr != NULL)

    {

        if (ptr -> info == item)

        {

            LOC = ptr;

            PAR = save;

            return;

        }

        if(item < ptr->info)

        {

            save = ptr;

            ptr = ptr->left;

```



```

    }

    else

    {

        save = ptr;

        ptr = ptr->right;

    }

}

LOC = NULL;

PAR = save;

return;

}

```

```

struct node* findmin(struct node*r)

{

    if (r == NULL)

        return NULL;

    else if (r->left!=NULL)

        return findmin(r->left);

    else if (r->left == NULL)

        return r;

}

struct node*insert(struct node*r, int x)

{

```

```

    if (r == NULL)
    {
        r = (struct node*)malloc(sizeof(struct node));

        r->info = x;

        r->left = r->right = NULL;

        return r;
    }

    else if (x < r->info)

        r->left = insert(r->left, x);

    else if (x > r->info)

        r->right = insert(r->right, x);

    return r;
}

```

```

int main()
{
    struct node* root = NULL;

    int x, c = 1, z;

    int element;

    char ch;

    printf("\nEnter an element: ");

    scanf("%d", &x);

```

```

root = insert(root, x);

printf("\nDo you want to enter another element :y or n");

scanf(" %c",&ch);

while (ch == 'y')

{

    printf("\nEnter an element:");

    scanf("%d", &x);

    root = insert(root,x);

    printf("\nPress y or n to insert another element: y or n: ");

    scanf(" %c", &ch);

}

while(1)

{

    printf("\n1 Insert an element ");

    printf("\n2 Search for an element");

    printf("\n3 Exit ");

    printf("\nEnter your choice: ");

    scanf("%d", &c);

    switch(c)

    {

        case 1:

            printf("\nEnter the item:");

            scanf("%d", &z);

```

```

        root = insert(root,z);

        break;

case 2:

    printf("\nEnter element to be searched: ");

    scanf("%d", &element);

    search(root, element);

    if(LOC != NULL)

        printf("\n%d Found in Binary Search Tree !!\n",element);

    else

        printf("\nIt is not present in Binary Search Tree\n");

        break;

case 3:

    printf("\nExiting...");

    return;

default:

    printf("Enter a valid choice: ");

    }

}

return 0;

}

```

## AVL Trees

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

### Insertion:-

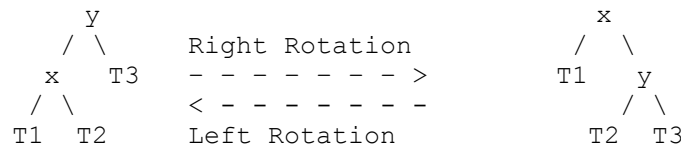
To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.

Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys (left) < key (root) < keys (right)).

1) Left Rotation

2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)



Keys in both of the above trees follow the following order  $\text{keys}(T1) < \text{key}(x) < \text{keys}(T2) < \text{key}(y) < \text{keys}(T3)$

So BST property is not violated anywhere.

### Steps to follow for insertion:

Let the newly inserted node be w

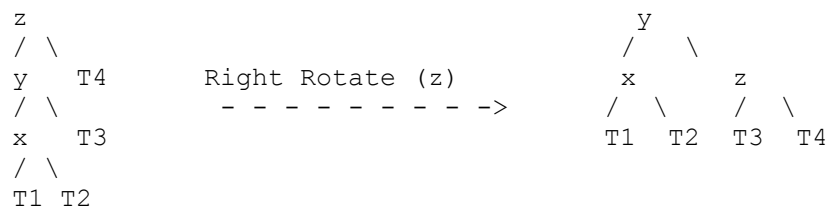
- Perform standard BST insert for w.
- Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways.

Following are the possible 4 arrangements:

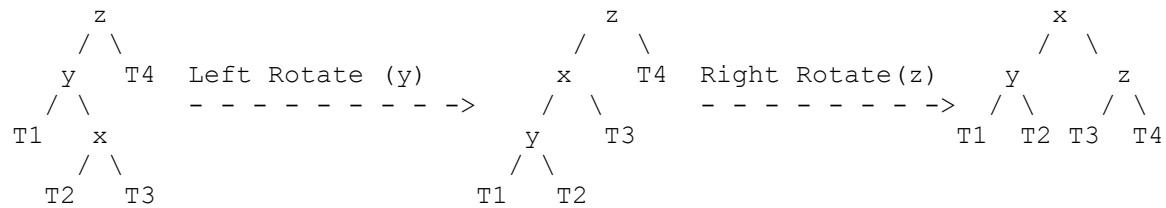
- y is left child of z and x is left child of y (Left Left Case)
- y is left child of z and x is right child of y (Left Right Case)
- y is right child of z and x is right child of y (Right Right Case)
- y is right child of z and x is left child of y (Right Left Case)

#### a) Left Left Case

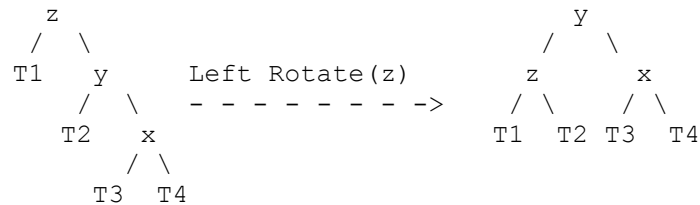
T1, T2, T3 and T4 are subtrees.



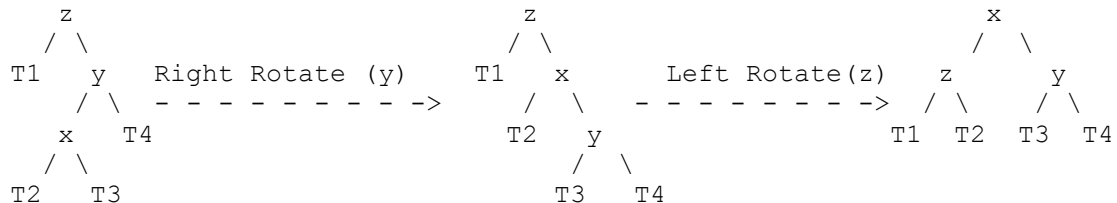
## b) Left Right Case



## c) Right Right Case



## d) Right Left Case



## Algorithm:

// Recursive function to insert key in subtree rooted with node and returns new root of subtree.

```
struct Node* insert(struct Node* node, int key)
```

```
{
```

```
    /* 1. Perform the normal BST insertion */
```

```
    if (node == NULL)
```

```
        return(newNode(key));
```

```
    if (key < node->key)
```

```
        node->left = insert(node->left, key);
```

```
    else if (key > node->key)
```

```
        node->right = insert(node->right, key);
```

```
    else // Equal keys are not allowed in BST
```

```
        return node;
```

```
    /* 2. Update height of this ancestor node */
```

```
    node->height = 1 + max(height(node->left),
```

```
        height(node->right));
```

```

/* 3. Get the balance factor of this ancestor node to check whether this node became
unbalanced */
int balance = getBalance(node);

// If this node becomes unbalanced, then
// there are 4 cases

// Left Left Case
if (balance > 1 && key < node->left->key)
    return rightRotate(node);

// Right Right Case
if (balance < -1 && key > node->right->key)
    return leftRotate(node);

// Left Right Case
if (balance > 1 && key > node->left->key)
{
    node->left = leftRotate(node->left);
    return rightRotate(node);
}

// Right Left Case
if (balance < -1 && key < node->right->key)
{
    node->right = rightRotate(node->right);
    return leftRotate(node);
}

/* return the (unchanged) node pointer */
return node;
}

```

**For deletion:**

- Perform the normal BST deletion.
- The current node must be one of the ancestors of the deleted node. Update the height of the current node.
- Get the balance factor (left subtree height – right subtree height) of the current node.
- If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
- If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

**Algorithm:**

```
// Recursive function to delete a node with given key
// from subtree with given root. It returns root of the modified subtree.
struct Node* deleteNode(struct Node* root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the root's key, then it lies in left subtree
    if ( key < root->key )
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key, then it lies in right subtree
    else if( key > root->key )
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then this is the node to be deleted
    else
    {
        // node with only one child or no child
        if( (root->left == NULL) || (root->right == NULL) )
        {
            struct Node *temp = root->left ? root->left :
                                root->right;

            // No child case
            if (temp == NULL)
            {
```



```

        temp = root;
        root = NULL;
    }
    else // One child case
        *root = *temp; // Copy the contents of
                        // the non-empty child
        free(temp);
    }
    else
    {
        // node with two children: Get the inorder successor (smallest in the right subtree)
        struct Node* temp = minValueNode(root->right);

        // Copy the inorder successor's data to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
}

// If the tree had only one node then return
if (root == NULL)
    return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root->height = 1 + max(height(root->left),
                      height(root->right));

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to
// check whether this node became unbalanced)
int balance = getBalance(root);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case
if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);

// Left Right Case
if (balance > 1 && getBalance(root->left) < 0)
{
    root->left = leftRotate(root->left);
    return rightRotate(root);
}

```

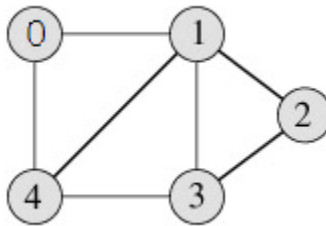
```
}  
  
// Right Right Case  
if (balance < -1 && getBalance(root->right) <= 0)  
    return leftRotate(root);  
  
// Right Left Case  
if (balance < -1 && getBalance(root->right) > 0)  
{  
    root->right = rightRotate(root->right);  
    return leftRotate(root);  
}  
return root;  
}
```

# GRAPHS

Graph is a data structure that consists of following two components:

- A finite set of nodes also called as vertices.
- A finite set of ordered pair of the form  $(u, v)$  called as edge.

The pair is ordered because  $(u, v)$  is not same as  $(v, u)$  in case of directed graph (di-graph). The pair of form  $(u, v)$  indicates that there is an edge from vertex  $u$  to vertex  $v$ . The edges may contain weight/value/cost.



In the above graph,

$V = \{0, 1, 2, 3, 4\}$

$E = \{01, 04, 10, 12, 13, 14, 21, 23, 31, 32, 34, 40, 41, 43\}$

Following two are the most commonly used representations of graph.

- Adjacency Matrix
- Adjacency List

There are other representations also like, Incidence Matrix and Incidence List.

## Adjacency Matrix:

- Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of vertices in a graph. Let the 2D array be  $adj[][]$ , a slot  $adj[i][j] = 1$  indicates that there is an edge from vertex  $i$  to vertex  $j$ . Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs.
- If  $adj[i][j] = w$ , then there is an edge from vertex  $i$  to vertex  $j$  with weight  $w$ .

The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

*Pros:* Representation is easier to implement and follow. Removing an edge takes  $O(1)$  time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done  $O(1)$ .

*Cons:* Consumes more space  $O(V^2)$ . Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is  $O(V^2)$  time.

### Adjacency List:

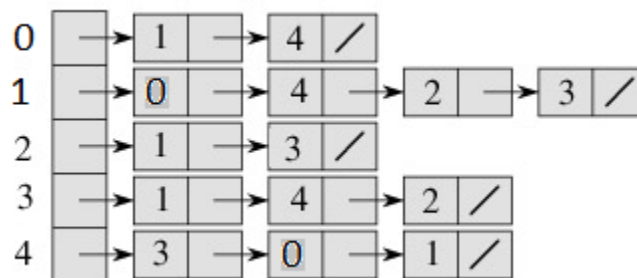
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be `array[]`.

An entry `array[i]` represents the linked list of vertices adjacent to the  $i$ th vertex.

This representation can also be used to represent a weighted graph.

The weights of edges can be stored in nodes of linked lists.

Following is adjacency list representation of the above graph.

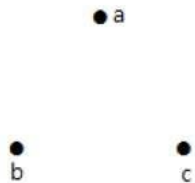


### TYPES OF GRAPHS:

- Null Graph

A **graph having no edges** is called a Null Graph.

Example, In the graph, there are three vertices named 'a', 'b', and 'c', but there are no edges among them. Hence it is a Null Graph.



- Trivial Graph

A **graph with only one vertex** is called a Trivial Graph.

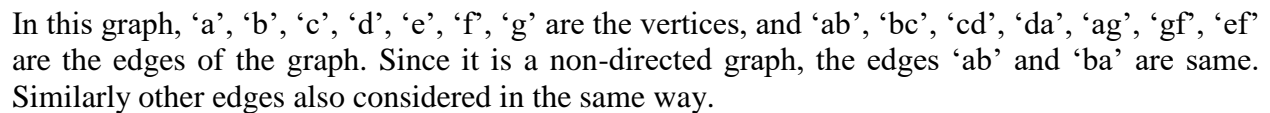
Example, In the shown graph, there is only one vertex 'a' with no other edges. Hence it is a Trivial graph.



- Non-Directed Graph

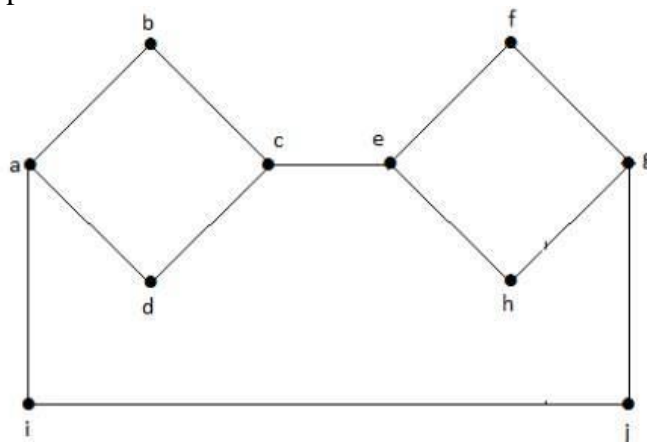
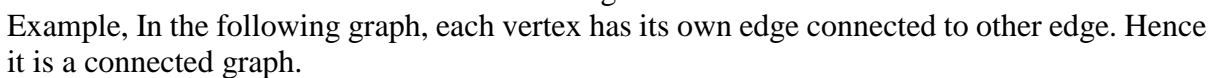
A non-directed graph contains edges but the edges are not directed ones.

Example



### Example

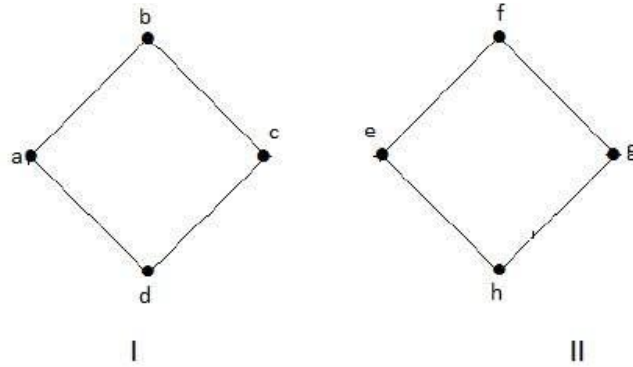
As it is a directed graph, each edge bears an arrow mark that shows its direction. Note that in a directed graph, ‘ab’ is different from ‘ba’.



- **Disconnected Graph**

A graph  $G$  is disconnected, if it does not contain at least two connected vertices.

Example, The following graph is an example of a Disconnected Graph, where there are two components, one with 'a', 'b', 'c', 'd' vertices and another with 'e', 'f', 'g', 'h' vertices.

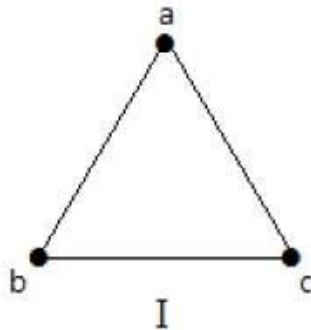


The two components are independent and not connected to each other. Hence it is called disconnected graph.

- **Regular Graph**

A graph  $G$  is said to be regular, if all its vertices have the same degree. In a graph, if the degree of each vertex is 'k', then the graph is called a 'k-regular graph'. Degree can be termed as the count of vertices connected to each other.

Example, In the following graphs, all the vertices have the same degree, a is connected to two vertices b & c, similarly b is connected to exactly two vertices a & c, and same with c. So these graphs are called regular graphs. All the vertices have degree 2. They are called 2-Regular Graphs.

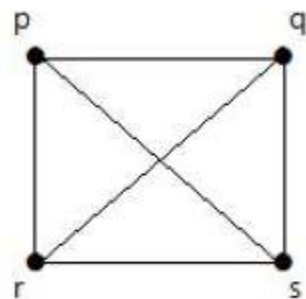


- **Complete Graph**

A simple graph with 'n' mutual vertices is called a complete graph and it is denoted by ' $K_n$ '. In the graph, a vertex should have edges with all other vertices, then it called a complete graph.

In other words, if a vertex is connected to all other vertices in a graph, then it is called a complete graph.

Example, In the following graphs, each vertex in the graph is connected with all the remaining vertices in the graph except by itself.



- Cycle Graph

A simple graph with 'n' vertices ( $n \geq 3$ ) and 'n' edges is called a cycle graph if all its edges form a cycle of length 'n'.

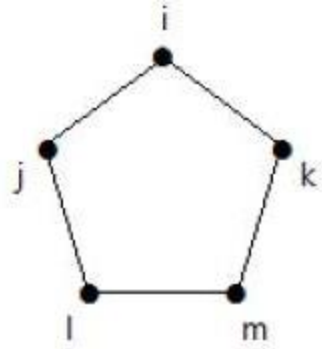
If the degree of each vertex in the graph is two, then it is called a Cycle Graph.

Notation –  $C_n$

Example, Take a look at the following graph –

Graph has 5 vertices with 5 edges which is forming a cycle 'ik-km-ml-lj-ji'.

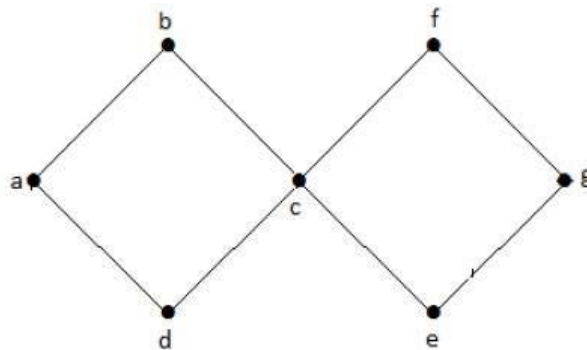
Hence the given graphs are cycle graphs.



- Cyclic Graph

A graph **with at least one** cycle is called a cyclic graph.

Example

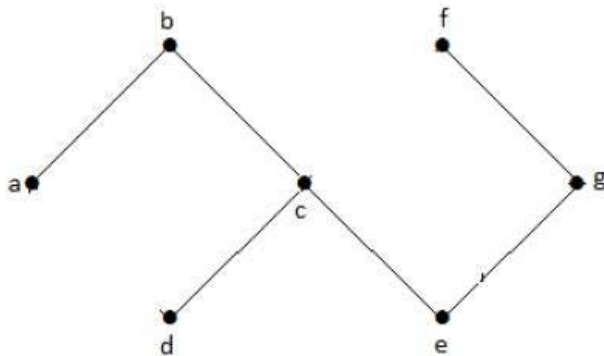


In the above example graph, we have two cycles a-b-c-d-a and c-f-g-e-c. Hence it is called a cyclic graph.

- Acyclic Graph

A graph **with no cycles** is called an acyclic graph.

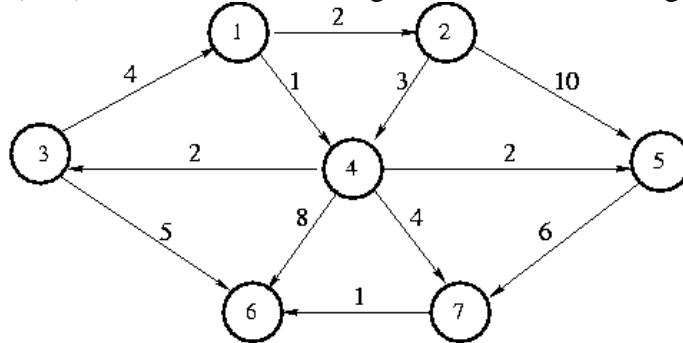
Example



In the above example graph, we do not have any cycles. Hence it is a non-cyclic graph.

- Weighted graph

When there is value (cost) attached with each edge, then it is called weighted graph.



In the above graph the cost is attached to each edge, i.e. moving from one vertex to another will cost some amount.

Let's see a code of directed and undirected graph using adjacency array:

```

#include <stdio.h>
#include <stdlib.h>
void main()
{
    int option;
    do
    {
        printf("\n 1. Directed Graph ");
        printf("\n 2. Un-Directed Graph ");
        printf("\n 3. Exit ");
        printf("\n\n Select a proper option : ");
        scanf("%d", &option);
        switch(option)
  
```



```

        {
        case 1 : dir_graph();
            break;
        case 2 : undir_graph();
            break;
        case 3 : exit(0);
        } // switch
    }while(1);
}
int dir_graph()
{
    int adj_mat[50][50];
    int n;
    int in_deg, out_deg, i, j;
    printf("\n How Many Vertices ? : ");
    scanf("%d", &n);
    read_graph(adj_mat, n);

    printf("\n Vertex \t In_Degree \t Out_Degree
\t Total_Degree ");
    for (i = 1; i <= n ; i++ )
    {
        in_deg = out_deg = 0;
        for ( j = 1 ; j <= n ; j++ )
        {
            if ( adj_mat[j][i] == 1 )
                in_deg++;
        }
        for ( j = 1 ; j <= n ; j++ )
            if (adj_mat[i][j] == 1 )
                out_deg++;
        printf("\n\n %5d\t\t\t%d\t\t%d\t\t%d\n\n",i,in_deg,out_deg,in_deg+out_deg);
    }
    return;
}
int undir_graph()
{
    int adj_mat[50][50];
    int deg, i, j, n;
    printf("\n How Many Vertices ? : ");
    scanf("%d", &n);
    read_graph(adj_mat, n);
    printf("\n Vertex \t Degree ");
    for ( i = 1 ; i <= n ; i++ )
    {
        deg = 0;

```

```

1. Directed Graph
2. Un-Directed Graph
3. Exit

```

Select a proper option : 1

How Many Vertices ? : 3

Vertices 1 & 2 are Adjacent ? (Y/N) :  
 Vertices 1 & 3 are Adjacent ? (Y/N) :y

Vertices 2 & 1 are Adjacent ? (Y/N) :  
 Vertices 2 & 3 are Adjacent ? (Y/N) :y

Vertices 3 & 1 are Adjacent ? (Y/N) :  
 Vertices 3 & 2 are Adjacent ? (Y/N) :y

Vertex	In_Degree	Out_Degree	Total_Degree
1	0	1	1
2	1	1	2
3	2	1	3

```

        for ( j = 1 ; j <= n ; j++ )
            if ( adj_mat[i][j] == 1)

                deg++;
        printf("\n\n %5d \t\t %d\n\n", i, deg);
    }
    return;
}

int read_graph ( int adj_mat[50][50], int n )
{
    int i, j;
    char reply;
    for ( i = 1 ; i <= n ; i++ )
    {
        for ( j = 1 ; j <= n ; j++ )
        {
            if ( i == j )
            {
                adj_mat[i][j] = 0;
                continue;
            }
            printf("\n Vertices %d & %d are Adjacent ? (Y/N) :", i, j);
            scanf("%c", &reply);
            if ( reply == 'y' || reply == 'Y' )
                adj_mat[i][j] = 1;
            else
                adj_mat[i][j] = 0;
        }
    }
    return;
}

```

```

1. Directed Graph
2. Un-Directed Graph
3. Exit

Select a proper option : 2

How Many Vertices ? : 3

Vertices 1 & 2 are Adjacent ? (Y/N) :
Vertices 1 & 3 are Adjacent ? (Y/N) :n

Vertices 2 & 1 are Adjacent ? (Y/N) :
Vertices 2 & 3 are Adjacent ? (Y/N) :y

Vertices 3 & 1 are Adjacent ? (Y/N) :
Vertices 3 & 2 are Adjacent ? (Y/N) :y

Vertex          Degree

    1              0

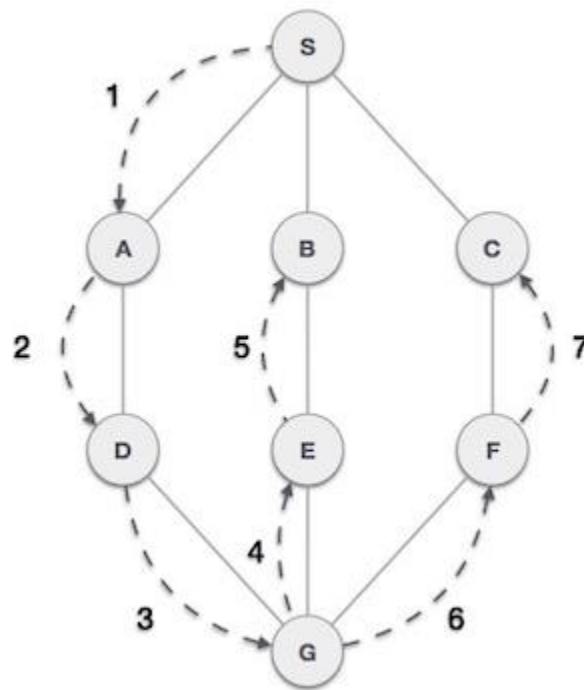
    2              1

    3              1

```

## Depth first Search:

Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



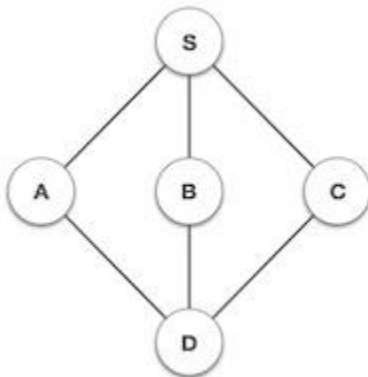
As in the example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

### Step Traversal

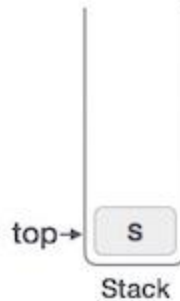
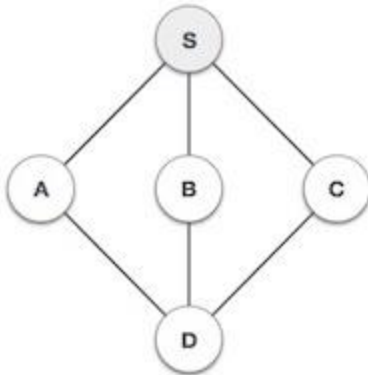
### Description

1.



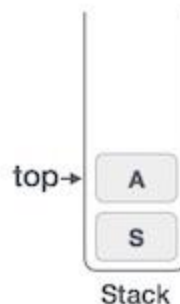
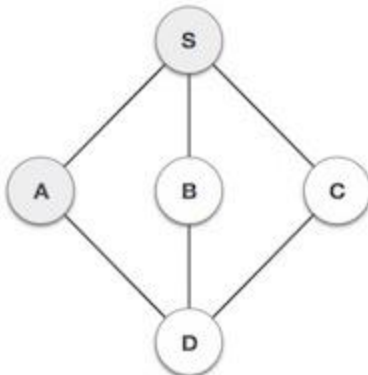
Initialize the stack.

2.



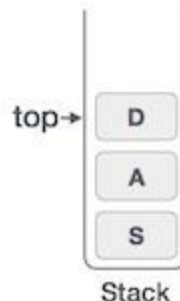
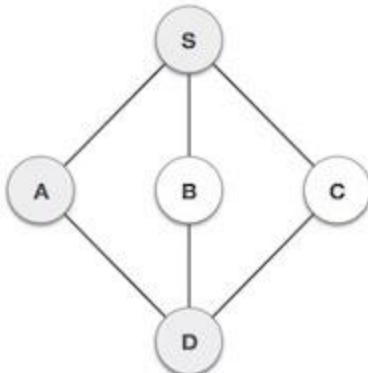
Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

3.



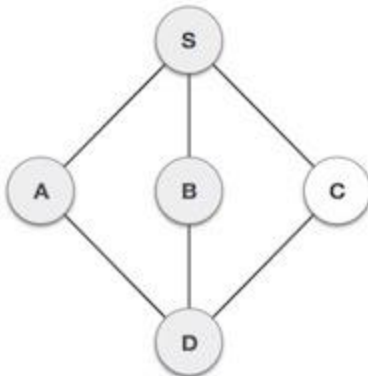
Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from **A**. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4.



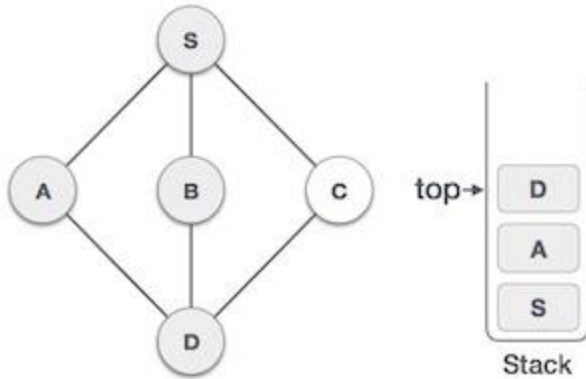
Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

5.



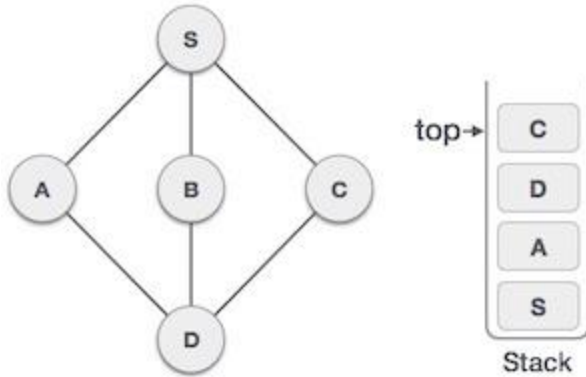
We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

6.



We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7.

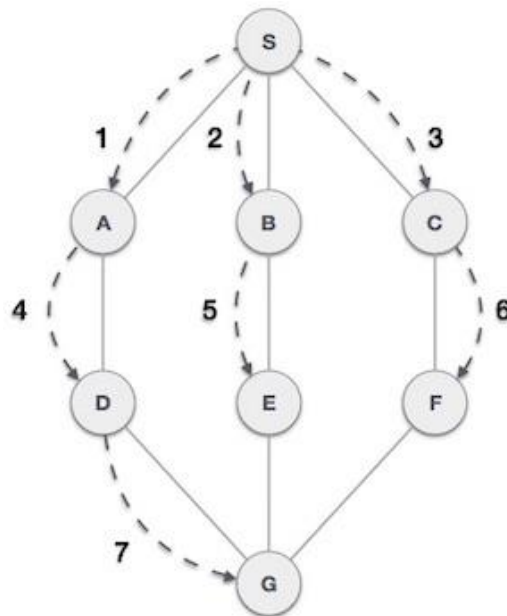


Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

## Breadth first search:

Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

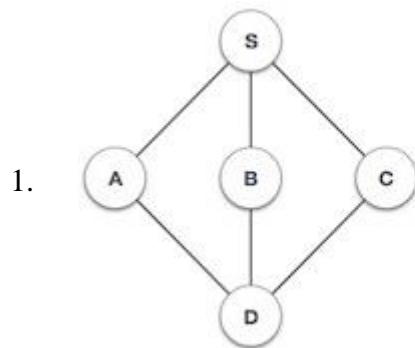


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

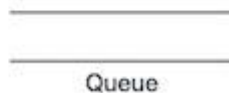
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

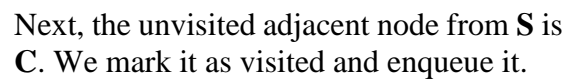
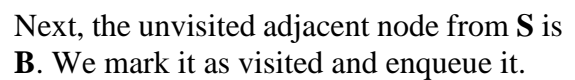
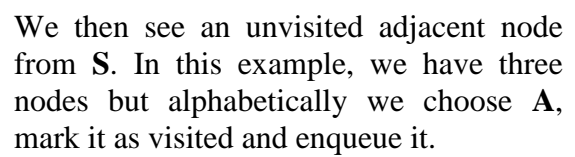
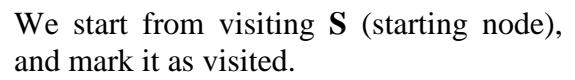
### Step Traversal

### Description

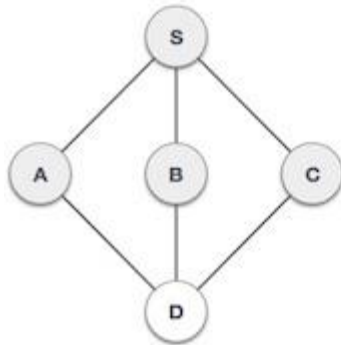


Initialize the queue.

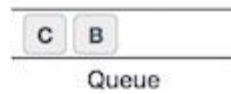




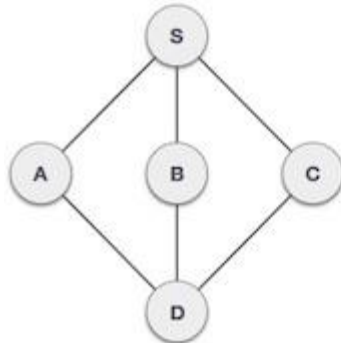
6.



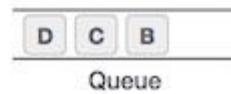
Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.



7.



From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.



At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.



## REFERENCES:

1. <https://www.tutorialspoint.com/>
2. <http://www.geeksforgeeks.org/>
3. <http://cslibrary.stanford.edu/>
4. <https://www.hackerearth.com/>