







---

## DATA STRUCTURE

---

TOPICS	PAGE
<ul style="list-style-type: none"><li>○ DYNAMIC MEMORY ALLOCATION</li><li>○ SELF-REFERENTIAL STRUCTURE<ul style="list-style-type: none"><li>○ TYPEDEF</li></ul></li></ul>	2-5
 LINK LIST	5-16
 STACK	17-28
<ul style="list-style-type: none"><li>○ APPLICATION OF STACK</li></ul>	28-38
 QUEUE	39-43
 TREE	43-48
<ul style="list-style-type: none"><li>○ BINARY SEARCH TREE</li><li>○ HEAP</li></ul>	48-50 51-56
 GRAPH	58-60
<ul style="list-style-type: none"><li>○ GRAPH TRAVERSAL(DFS&amp;BFS)</li><li>○ TOPOLOGICAL ORDERING</li></ul>	62-67 68-70
 TIME COMPLEXITY & SPACE COMPLEXITY	56 & 57

## Dynamic Memory Allocation:

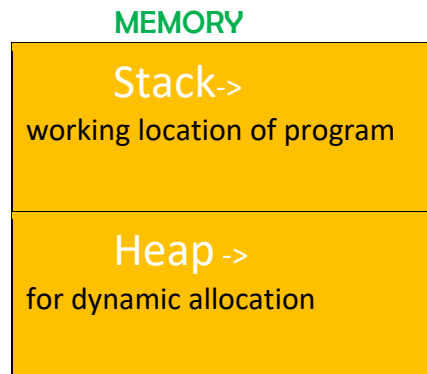
It's the process that allows your program to obtain more memory space while running, or to release memory if that is not required. In brief dynamic memory allocation allows you to manually handle memory space for your program.

In C language there are 4 library functions under "**stdlib.h**" header file for dynamic memory allocation.

Function	Use of function
<u>malloc()</u>	Allocates requested size of bytes and returns a pointer first byte of allocated space
<u>calloc()</u>	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
<u>free()</u>	deallocate the previously allocated space
<u>realloc()</u>	Change the size of previously allocated space

We normally use **malloc()** function for Dynamic Memory Allocation.

- This malloc function allocate memory in Byte ( from the "Heap" )



Memory in a program is divided into two parts:

- **The stack:** All variables declared inside the function will take up memory from the stack.
- **The heap:** This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

## Syntax of malloc()

```
ptr = (cast-type*) malloc(byte-size);
```

Here, **ptr** is **pointer** of cast-type. The **malloc()** function returns a **pointer** to an area of memory with size of **byte size**. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr = (int*) malloc(n * sizeof(int));
```

This statement will allocate either **n\*2** or **n\*4** according to size of **int 2 or 4 bytes** respectively and the pointer points to the address of first byte of memory.

### \*\*Simple program:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *p , n = 10 , i;
    p = (int*)malloc(n*sizeof(int)); //here we allocate memory in heap
    for(i = 0 ; i< n ; i++)
    {
        p[i] = i;
    }
    for(i = 0; i<n ; i++)
    {
        printf("%d\n",p[i]);
    }
    return 0;
}
```

## SELF REFERENTIAL STRUCTURE

If a structure contains one or more pointers to itself (a structure of same type) as its members, then the structure is said to be a self-referential structure, that is, a structure that contains a reference to its own structure type.

In brief, One of the member of the structure is pointer of the same type.

For example:

```
struct node                //{here marked portion is "data type"}
{
    int data1;
    int data2;
    struct node *next;    //{here red marked portion is "self referential member"}
};
```

Here, the structure '*node*' contains a pointer named '*next*', which is of the same type (*struct node*) as the structure it contains in (*struct node*).

The above illustrated structure prototype describes one node that comprises of two logical segments. One of them stores data/information and the other one is a pointer indicating where the next component can be found. Several such inter-connected nodes create a chain of structures.

The following figure depicts the composition of such a node.



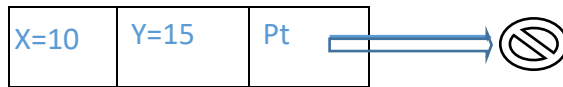
### TYPEDEF:

**typedef** is used to give a new name to a type. Lets define a new type for the given *struct node*,

```
typedef struct node point;    //{green marked is old data type and blue marked is new data type}
```

Now structure is declared as "point" instead of node.

**typedef** really does simply declare a new *name* for a type. It does not create a new type. **#simple program on creating nodes and assigning data:**



```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int x,y;
    struct Node *next;
};
typedef struct Node  node;
int main()
{
    node *head;
    head=(node*)malloc(sizeof(node));
    head->x=10;
    head->y=15;
    head->pt=NULL;
    printf("%d %d\n",head->x,head->y);
    return 0;
}
```

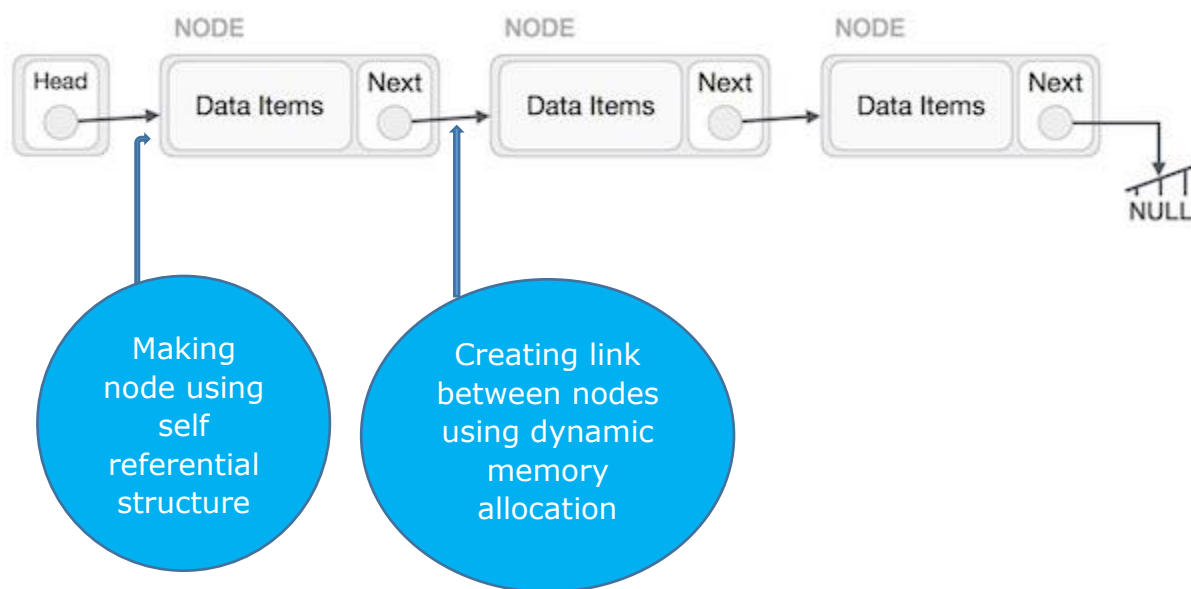
\*\*\*\*\*

---

# LINK LIST

---

- Linked List is a *linear data structure* which is a sequence of data structures, that are connected together via links and it is very common data structure which consists of group of nodes in a sequence that is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain.



- Linked List contains a link element called *head*.
- 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.

## 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.

## A simple C program to introduce a linked list

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

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

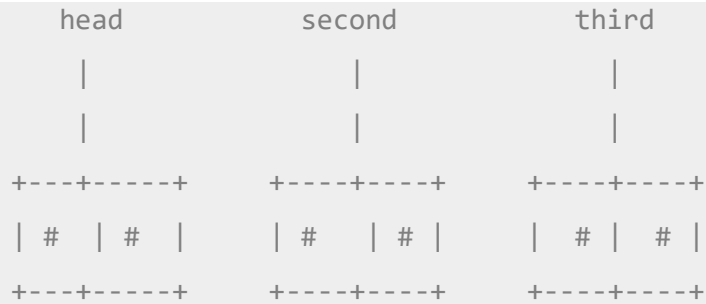
// Program to create a simple linked list with 3 nodes

int main()
{
    struct node* head = NULL;
    struct node* second = NULL;
    struct node* third = NULL;

    // allocate 3 nodes in the heap

    head = (struct node*)malloc(sizeof(struct node));
    second = (struct node*)malloc(sizeof(struct node));
    third = (struct node*)malloc(sizeof(struct node));

    /* Three blocks have been allocated dynamically.
       We have pointers to these three blocks as first, second and third
```



'#' represents any random value.

Data is random because we haven't assigned anything yet \*/

```
head->data = 1;           //assign data in first node
head->next = second;     // Link first node with the second node
```

/\* data has been assigned to data part of first block (block pointed by head). And next pointer of first block points to second. So they both are linked.

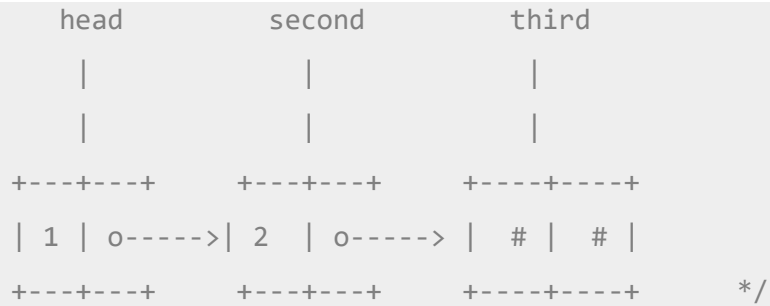


\*/

```
second->data = 2;        //assign data to second node
second->next = third;    // Link second node with the third node
```

/\* data has been assigned to data part of second block (block pointed by second). And next pointer of the second block points to third block. So all three blocks are linked.



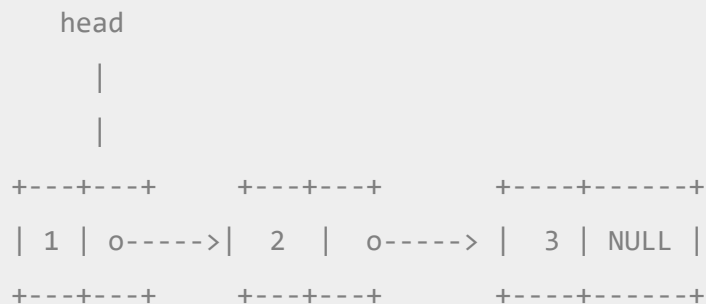


\*/

```
third->data = 3;           //assign data to third node
third->next = NULL;
```

/\* data has been assigned to data part of third block (block pointed by third). And next pointer of the third block is made NULL to indicate that the linked list is terminated here.

We have the linked list ready.



Note that only head is sufficient to represent the whole list. We can traverse the complete list by following next pointers. \*/

```
return 0;
```

```
}
```

### *Advantages of Linked Lists:*

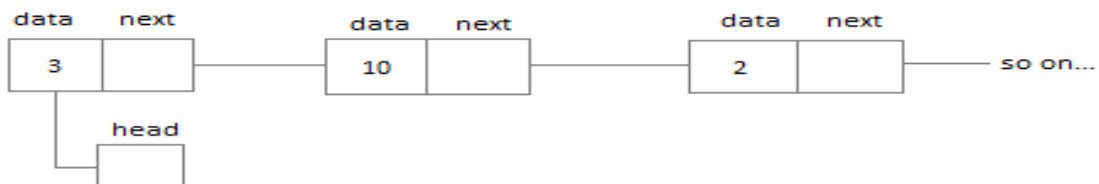
- They are a dynamic in nature which **allocates the memory when required**.
- **Insertion and deletion operations** can be easily implemented.
- Linked List **reduces the access time**.

### *Disadvantages of Linked Lists:*

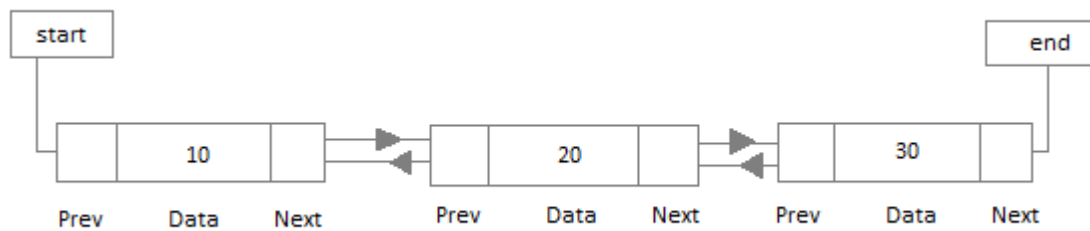
- The memory is wasted as pointers require extra memory for storage.
- **No element can be accessed randomly; it has to access each node sequentially.**
- Reverse Traversing is difficult in linked list.

### **Types of Linked Lists**

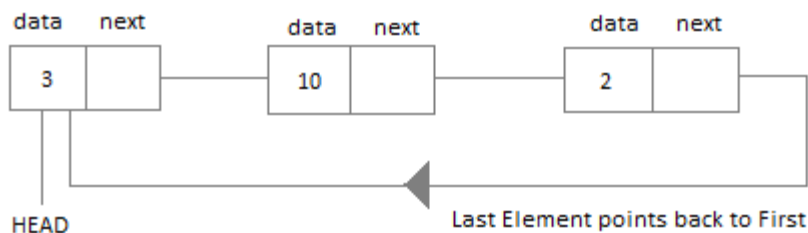
- **Singly Linked List :** Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



- **Doubly Linked List :** In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.

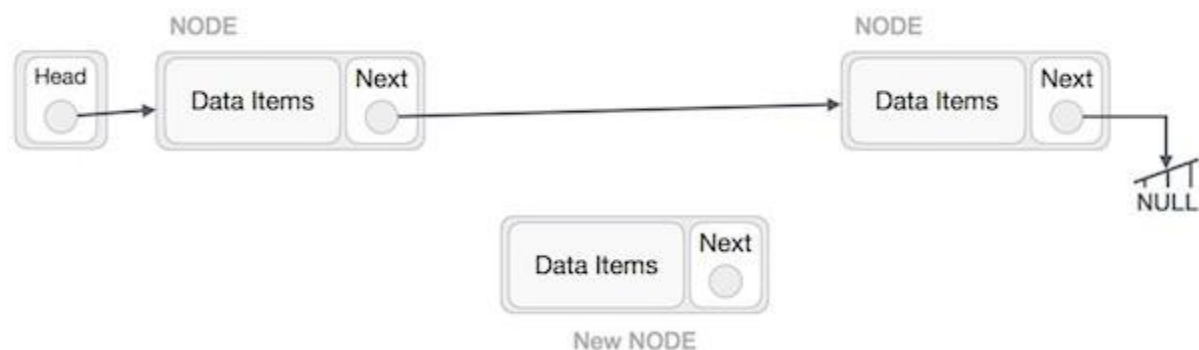


- **Circular Linked List :** In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



## Insertion Operation

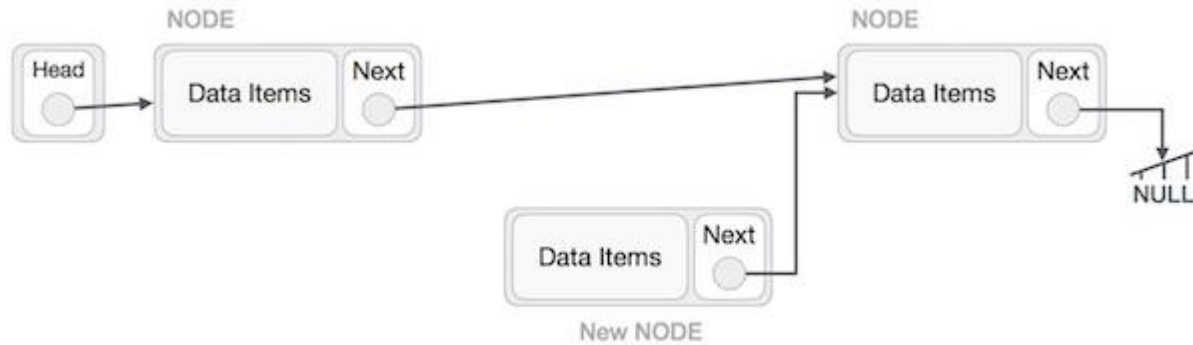
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C –

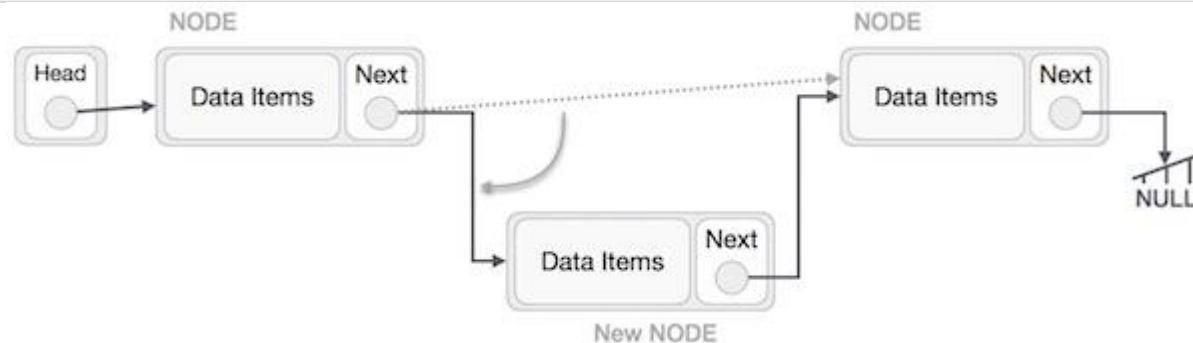
*NewNode.next -> RightNode;*

It should look like this –



Now, the next node at the left should point to the new node.

*LeftNode.next -> NewNode;*



This will put the new node in the middle of the two. The new list should look like this –



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

It's highly recommended that before seeing code assure that you have tried enough to implement in code.

Insert at the **front** of linklist:

```
void insert_front(node *list,int data)
{
    node *temp;
    temp=(node*)malloc(n*sizeof(node));
    temp->data=data;
    temp->next=list;
    list=temp;
}
```

Insert at **after** of a node of linklist:

```
void insert_after(node *list,int data)
{
    if(list==NULL)
    {
        printf("previous node cann't be NULL!");
    }
    node *temp= (node*)malloc(1*sizeof(node));
    temp->data=data;
    temp->next=list->next;
    list->next=temp;
}
```

Insert at the **end** of linklist:

```
void insert_end(node *list, int data)
{
    node *temp=(node*)malloc(1*sizeof(node));
    temp->data=data;
    temp->next=NULL;
    while(list->next!=NULL)
        list=list->next;
    list->next=temp;
    return;
}
```

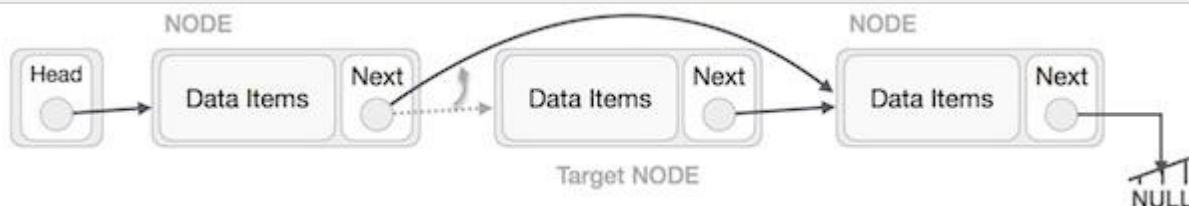
## Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



The left (previous) node of the target node now should point to the next node of the target node –

```
LeftNode.next -> TargetNode.next;
```

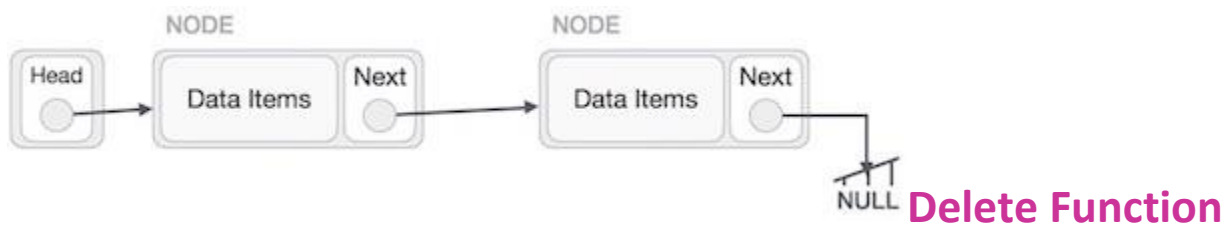


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```



We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



**Delete Function**

## implementation:

It's highly recommended that before seeing code assure that you have tried enough to implement this function in code.

Delete function implementation:

```
void delete(node *list,int data)
{
    node *temp;
    while(list->next->data != data)
        list=list->next;
    temp=list->next;
    list->next=list->next->next;
    free(temp);
    printf("%d is deleted",data);
}
```

## Update function implementation:

```
void update(node *list,int data1,int data2)
{
    while(list -> data!=data)
        list=list->next;
    list->data=data2;
    printf("%d updated!\n",data1);
}
```

## Count function implementation:

```
void count(node *list)
{
    int cnt=0;
    while(list->next != NULL)
    {
        cnt++;
        list=list->next;
    }
    return cnt;
}
```

## Exercise:

**\*\*write a function "sum" that returns summation of all data in the linklist.**



\*\*using the given functions implement a linklist. Where you will insert 12,15,18. Then print them. Then delete the data 15 and print. Now you update the data "18" to "25". Now print all data again.

\*\*\*\*\*

---

## Stack

---

Stack is an **abstract data type** with a bounded(predefined) capacity. It is a simple **linear data structure** that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack, the only element that can be removed is the element that was at the **top** of the stack, just like a pile of objects.

A real-world stack allows operations at **one end only**. Firstly take real time example, Suppose we have created stack of the book like this shown in the following fig –

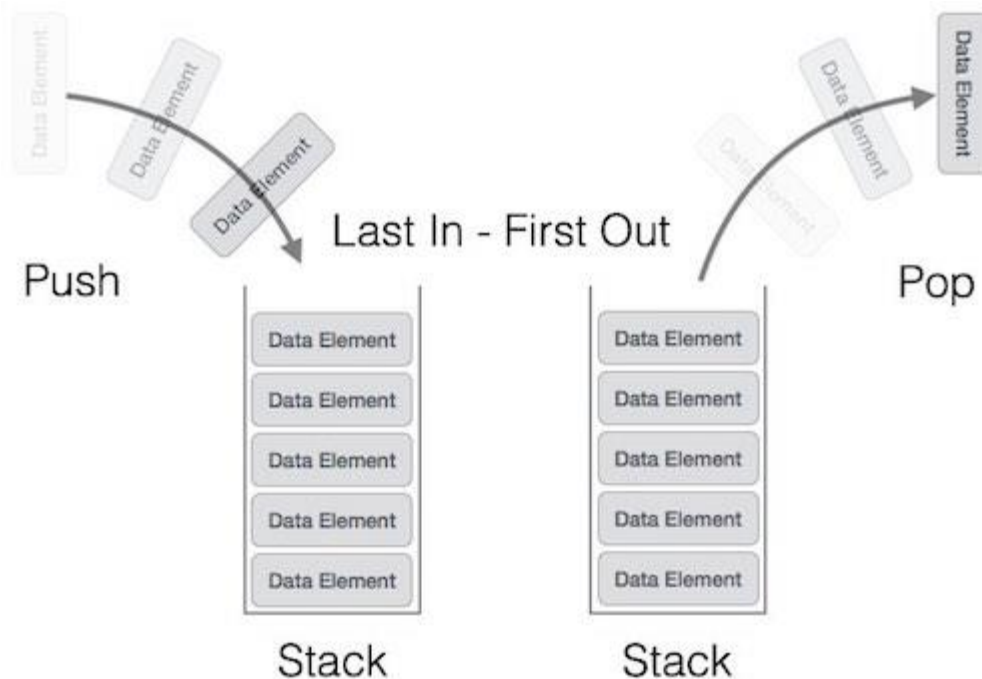


How books are arranged in that stack???

1. Books are kept one above the other
2. Book which is inserted first is Taken out at last.(Brown)
3. Book which is inserted Lastly is served first.(Light Green)

Stack is **first in last out (LIFO)** **last in first out(LIFO)** Data Structure. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

The following diagram depicts a stack and its operations –



### **Basic Operations:**

A stack is used for the following two primary operations –

- **push()** – Pushing (storing/inserting) an element on the stack.
- **pop()** – Removing (accessing/removing) an element from the stack.

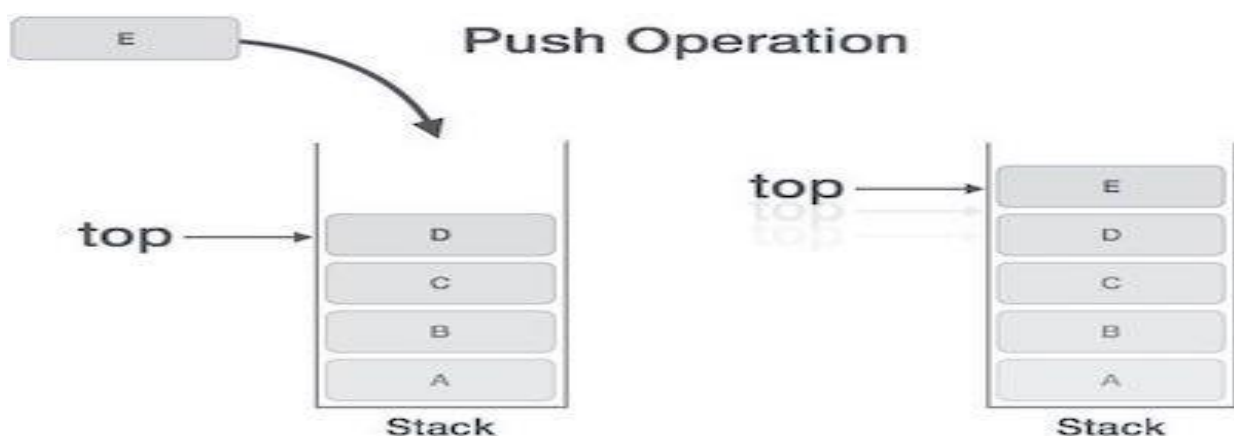
To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

- **peek()** – get the **top data element** of the stack, without removing it.
- **isFull()** – check if stack is full.(if **top** is equal to **maximum size** then stack is full)
- **isEmpty()** – check if stack is empty.( if **top** is less than **0** then stack is empty)

### 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.

## PUSH Operation implementation in array

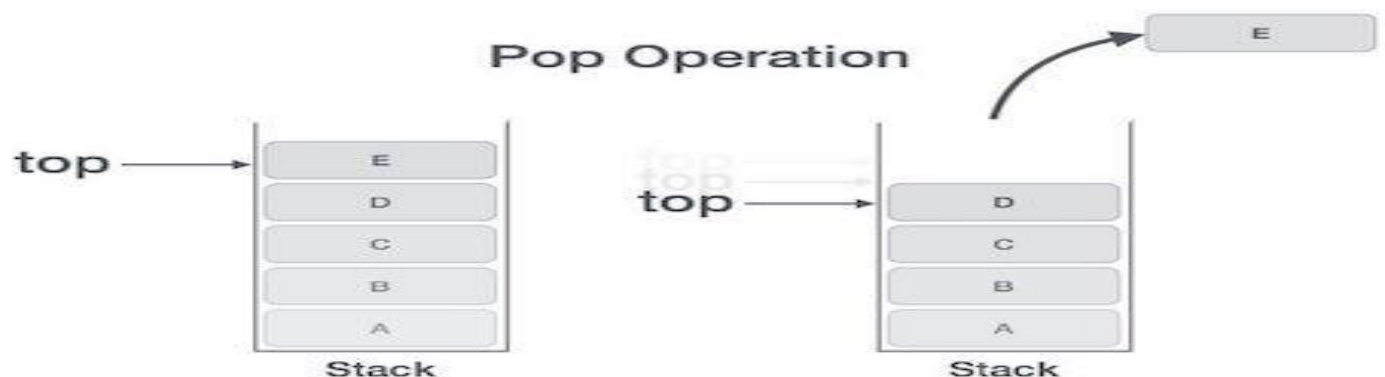
```
void push(int data) {  
    if(!isFull()) {  
        top = top + 1;  
        stack[top] = data;  
    }  
    Else {  
        printf("Could not insert data, Stack is full.\n");  
    }  
}
```

## 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. 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.



## Pop Operation implementation in array

```
int pop(int data)
{
    if(top<0){
        printf("stack is empty!\n");
        return -1;
    }
    top--;
    return stack[top]
}
```

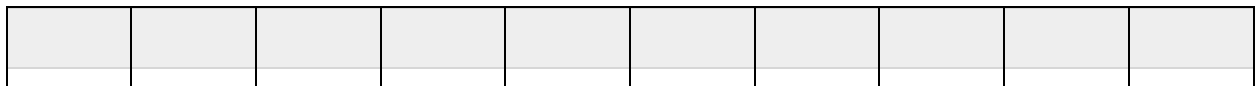
### *Implementation:*

There are two ways to implement a stack:

- Using array
- Using linked list

## **Implementing Stack using Arrays**

```
#include<stdio.h>
#define max 10
int stack[max];
/*
```



```
0      1      2      3      4      5      6      7      8      9
*/
int top=-1;
void push(int data)
{
    if(top==max)
    {
        printf("Stack is full\n");
    }
}
```

```

        return;
    }
    top++;
    stack [top]=data;
    printf("%d is pushed\n",data);
}

int pop()
{
    if(top<0)
    {
        printf("Stack is empty\n");
        return -1;
    }
    return stack [top--];
}

int main()
{
    push(7);
    /*

```

7									
---	--	--	--	--	--	--	--	--	--

0(top)      1      2      3      4      5      6      7      8      9

\*/

```

    push(9);
    /*

```

7	9								
---	---	--	--	--	--	--	--	--	--

0      1(top)      2      3      4      5      6      7      8      9

\*/

```

    push(11);

```

```
/*
```

7	9	11							
---	---	----	--	--	--	--	--	--	--

```
0      1      2(top)  3      4      5      6      7      8      9
```

```
*/
```

```
    printf("%d\n",pop());
```

```
/*
```

7	9								
---	---	--	--	--	--	--	--	--	--

```
0      1(top)  2      3      4      5      6      7      8      9
```

```
*/
```

```
    printf("%d\n",pop());
```

```
/*
```

7									
---	--	--	--	--	--	--	--	--	--

```
0(top)  1      2      3      4      5      6      7      8      9
```

```
*/
```

```
    printf("%d\n",pop());
```

```
/*
```

--	--	--	--	--	--	--	--	--	--

```
0      1      2      3      4      5      6      7      8      9
```

```
top becomes -1 now */
```

```
    printf("%d\n",pop());
```

```
/*stack is empty now so it will return -1 and print stack is empty*/
```

```
    return 0;
```

```
}
```

## Implementation of *stack* using *linklist*:

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int data;
    struct Node *next;
};
typedef struct Node node;
void push(node *stack,int data)
{
    node *temp;
    temp=(node*)malloc(sizeof(node));
    /*
        temp
        |
        |
        +---+-----+
        | # | # |
        +---+-----+
    */
    temp->data=data;
    temp->next=NULL;
    /*
        temp
        |
        |
        +-----+-----+
        | data |   •----->∅
        +-----+-----+
    */
}
```



```

while(stack->next !=NULL)
{
    stack=stack->next;
}
stack->next=temp;
/*
    stack                temp
    |                    |
    |                    |
+-----+-----+      +-----+-----+
|  #  |  ●-----> | data |  ●-----> ∅
+-----+-----+      +-----+-----+
*/
printf("%d is pushed!\n",data);
}
int pop(node *stack)
{
    int data;
    node *temp;
    if(stack->next==NULL)
    {
        printf("Stack is empty\n");
        return -1;
    }
    while(stack->next->next!=NULL)
    {
        stack=stack->next;
    }
    temp=stack->next;

```

```

    data=temp->data;
    stack->next=NULL;
    free(temp);
    return data;
}
int main()
{
    node *stack;
    stack=(node*)malloc(sizeof(node));
    stack->next=NULL;

```

```

/*      stack(top)
        |
        |
+-----+-----+
|  #   |   •----->∅
+-----+-----+
*/

```

```

    push(stack,9);

```

```

/*      stack          stack->next(top)
        |              |
        |              |
+-----+-----+    +-----+-----+
|  #   |   •-----> |  9   |   •-----> ∅
+-----+-----+    +-----+-----+
*/

```

```

    push(stack,11);

```

```

/*      stack      stack->next(top)
      |            |
      |            |
+-----+-----+ +-----+-----+ +-----+-----+
| # | •----→ | 9 | •----→ | 11 | •----→ ∅
+-----+-----+ +-----+-----+ +-----+-----+
*/

push(stack,7);

/* stack      stack->next      stack->next->next      stack->next->next->next(top)
      |            |            |            |
      |            |            |            |
+-----+-----+ +-----+-----+ +-----+-----+ +-----+-----+
| # | •----→ | 9 | •----→ | 11 | •----→ | 7 | •----→ ∅
+-----+-----+ +-----+-----+ +-----+-----+ +-----+-----+
*/

printf("\n%d is popped\n",pop(stack));

stack      stack->next      stack->next->next(top)
      |            |            |
      |            |            |
+-----+-----+ +-----+-----+ +-----+-----+
| # | •--→ | 9 | •--→ | 11 | •--→ ∅
+-----+-----+ +-----+-----+ +-----+-----+

printf("%d is popped\n",pop(stack));

      stack      stack->next(top)
      |            |
      |            |
+-----+-----+ +-----+-----+
| # | •-----> | 9 | •-----> ∅
+-----+-----+ +-----+-----+

printf("%d is popped\n",pop(stack));

```

```
stack(top)
|
|
+-----+-----+
|  #  |  •----->∅
+-----+-----+

return 0;
}
```

## Applications of Stack

**Expression Parsing** (Infix to Postfix, Postfix to Prefix etc) and many more.

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.



## Expression Parsing

1+2-5 ←-arithmetic expression

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are,

- Infix Notation
- Prefix Notation
- Postfix Notation

Expression	Example	Note
Infix	a + b	Operator <b>Between</b> Operands
Prefix	+ a b	Operator <b>before</b> Operands
Postfix	a b +	Operator <b>after</b> Operands

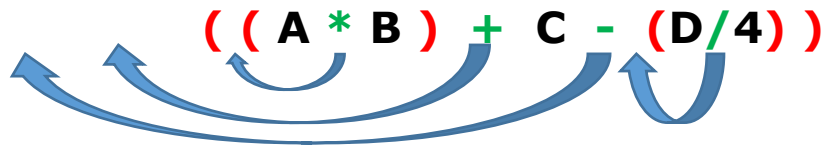
## Infix to Prefix:

**Infix** →  $A * b + C - D / 4$

Step1: Parenthesize (put parenthesis) according to precedence of operator.

$((A * B) + C - (D / 4))$

Step2: Move operator to the left out of enclosing parenthesis.



Step3: write the prefix.

$+ - * A B C / D 4 \leftarrow \text{PREFIX}$

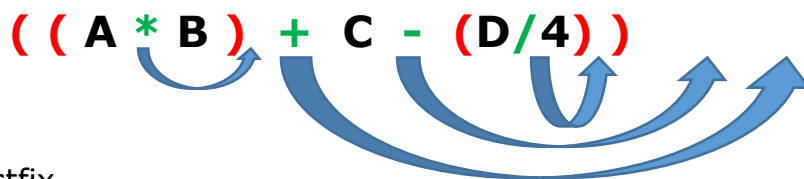
## Infix to Postfix:

**Infix** →  $A * b + C - D / 4$

Step1: Parenthesize (put parenthesis) according to precedence of operator.

$((A * B) + C - (D / 4))$

Step2: Move operator to the right out of enclosing parenthesis.



Step3: write the postfix.

$AB * CD 4 / - + \leftarrow \text{POSTFIX}$

The following table briefly tries to show the difference in all three notations –

Sr. No.	Infix Notation	Prefix Notation	Postfix Notation
1	$a + b$	$+ a b$	$a b +$
2	$(a + b) * c$	$* + a b c$	$a b + c *$
3	$a * (b + c)$	$* a + b c$	$a b c + *$
4	$a / b + c / d$	$+ / a b / c d$	$a b / c d / +$
5	$(a + b) * (c + d)$	$* + a b + c d$	$a b + c d + *$
6	$((a + b) * c) - d$	$- * + a b c d$	$a b + c * d -$


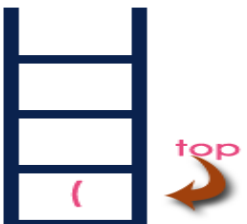




Sr. No.	Operator	Precedence	Associativity
1	Esponentiation ^	Highest	Right Associative
2	Multiplication ( * ) & Division ( / )	Second Highest	Left Associative
3	Addition ( + ) & Subtraction ( - )	Lowest	Left Associative

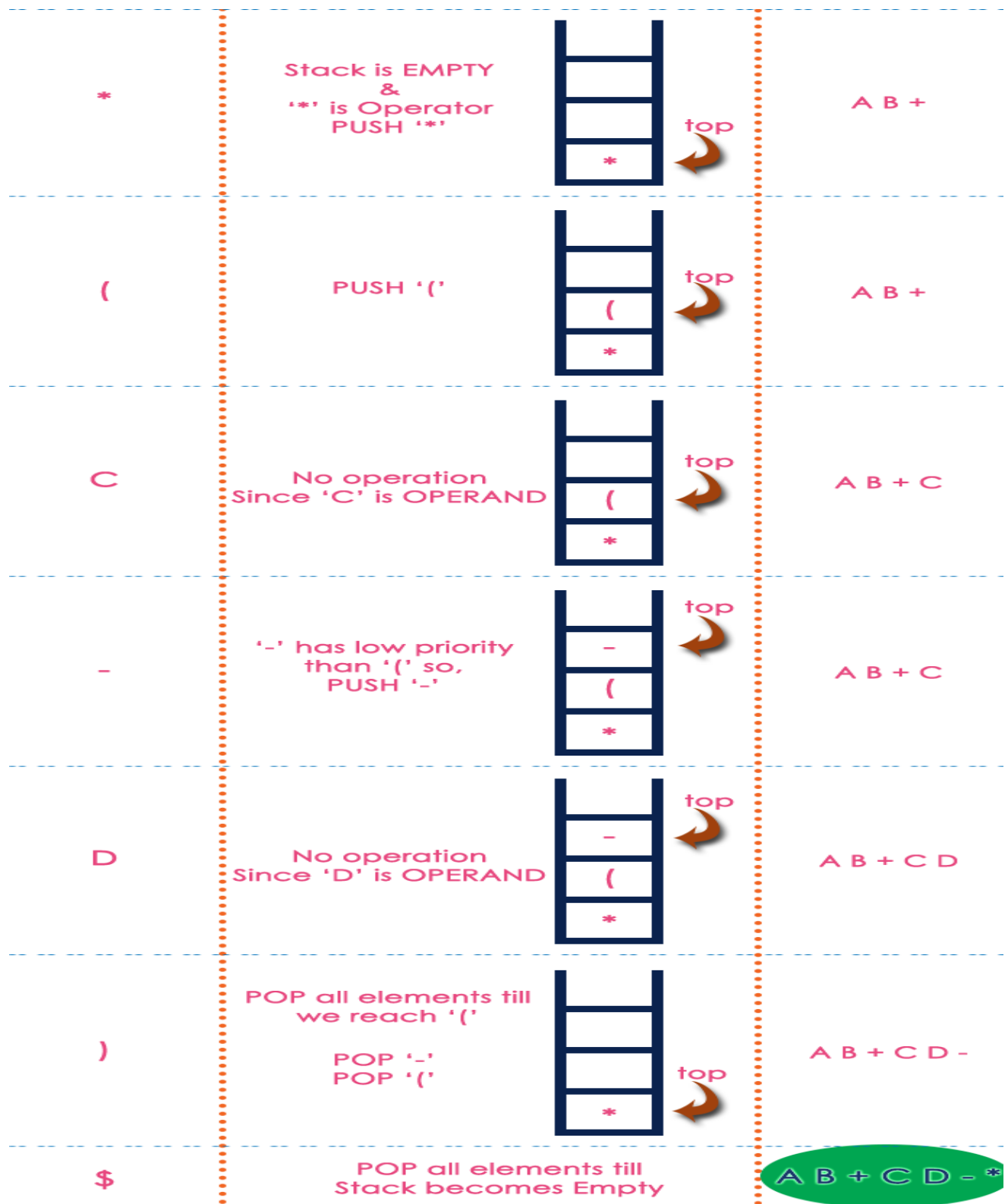
In  $a + b * c$ , the expression part  $b * c$  will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for  $a + b$  to be evaluated first, like  $(a + b) * c$ .

## Infix to postfix conversion using Stack:

Considering the following Infix Expression

$(A+B)*(C-D)$

Reading Character	STACK	Postfix Expression
Initially	Stack is EMPTY 	EMPTY
(	Push '(' 	EMPTY
A	No operation Since 'A' is OPERAND 	A
+	'+' has low priority than '(' so, PUSH '+' 	A
B	No operation Since 'B' is OPERAND 	A B
)	POP all elements till we reach '('  POP '+' POP '(' 	A B +



The final postfix expression is as follows:

A B + C D - \*



## Infix to postfix conversion implementation:

```
#include<stdio.h>
#include<string.h>
#define max 100
char stack[max];
int top=-1;
void push(char k){
    if(top+1==max)
        printf("Stack is full!");
    else{
        top++;
        stack[top]=k;
    }
}
char pop(){
    char a;
    if(top<0)
    {
        printf("Stack is empty\n");
        return '\0';
    }
    else
    {
        a=stack[top];
        top--;
        return a;
    }
}
int main()
```

```

{
    int i,l;
    char ch,e[max];
    printf("Enter string with():");
    gets(e);
    l=strlen(e);
    printf("Postfix is:\n");
    for(i=0; i<=l-1; i++){
        ch=e[i];
        if(ch=='(' || ch=='*' || ch=='-' || ch=='+' || ch=='/')
            push(ch);
        else if(ch!=')')
            putchar(ch);
        else{
            do{
                ch=pop();
                if(ch!='(')
                    putchar(ch);
            }
            while(ch!='(');
        }
    }
    return 0;
}

```

## Stack Evaluation:

(5+3)\*(8-2)

Value: 48





Now lets evaluate given expression using stack.






Step 1.Postfix      →      **43/232\*-+**

## Step 2. Evaluation→

### Rules:

- If operand then push into stack.
- If operator ( ^, +, -, \*, / ) then pop two operands and do the operation and push into stack.

Reading Symbol	Stack Operations	Stack	Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)		Nothing
3	push(3)		Nothing
+	value1 = pop() value2 = pop() result = value2 + value1 push(result)		<pre> value1 = pop(); // 3 value2 = pop(); // 5 result = 5 + 3; // 8 Push( 8 ) </pre> <b>(5 + 3)</b>

8	push(8)		(5 + 3)
2	push(2)		(5 + 3)
-	value1 = pop() value2 = pop() result = value2 - value1 push(result)		value1 = pop(); // 2 value2 = pop(); // 8 result = 8 - 2; // 6 Push( 6 ) <b>(8 - 2)</b> (5 + 3) , (8 - 2)
*	value1 = pop() value2 = pop() result = value2 * value1 push(result)		value1 = pop(); // 6 value2 = pop(); // 8 result = 8 * 6; // 48 Push( 48 ) <b>(6 * 8)</b> (5 + 3) * (8 - 2)
\$ End of Expression	result = pop()		Display (result) <b>48</b> As final result

Infix Expression **(5 + 3) \* (8 - 2) = 48**

Postfix Expression **5 3 + 8 2 - \* value is 48**

## Infix to postfix conversion implementation:

```
#include<stdio.h>
#include<ctype.h>
int stack[50];
int top=-1;
void push(int data){
    stack[++top]=data;
}
int pop(){
    return (stack[top--]);
}
int main()
{
    char post[50],ch;
    int i=0, op1,op2;
    printf("Enter postfix:");
    scanf(" %s",post);
    while((ch=post[i++])!='\0'){
        if(isdigit(ch)){
            push(ch-'0');
        }
        else{
            op2=pop();
            op1=pop();
            printf("%d %d\n",op1,op2);
            switch(ch){
                case '+':push(op1+op2);
                    break;
                case '-':push(op1-op2);
```

```
        break;
        case '*':push(op1*op2);
        break;
        case '/':push(op1/op2);
        break;
    }
}
}
printf("result=%d\n",stack[top]);
return 0;
}
```

\*\*\*\*\*

---

## Queue:

---

- Linear data structure
- First in first out (FIFO/LILO)
- Priority queue



### Basic Operations:

enqueue() : add (store)/ insert an item to the queue

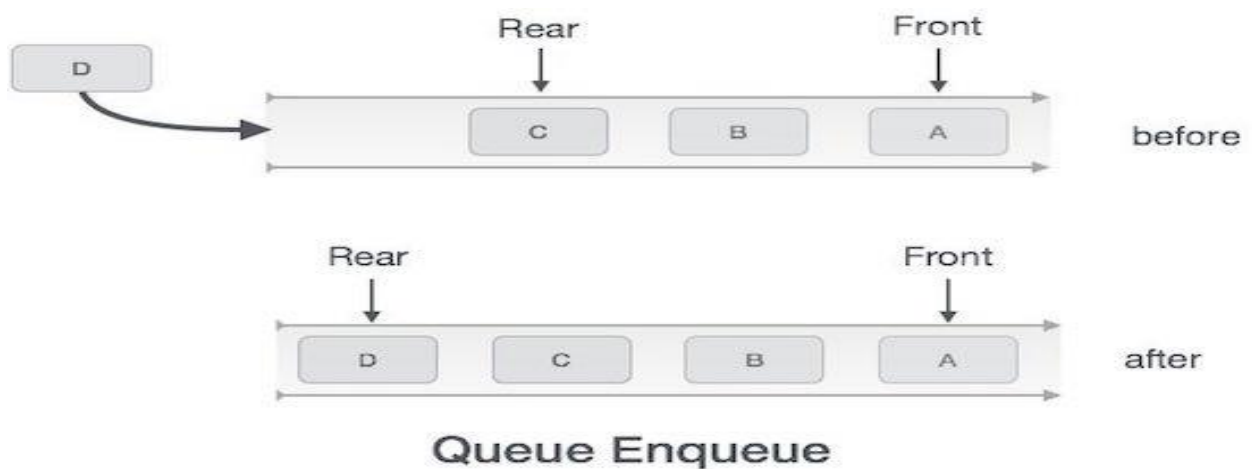
dequeue() : remove an item from the queue

### Enqueue Operations:

**Step – 1:** Check if queue is full

**Step – 2:** If queue is full procedure overflow error

**Step – 3:** If queue is not full create a temp node and assign node at last



## Enqueue Implementation:

```
void enqueue(int data)
{
    if(rear==full)
    {
        printf("Queue is full!\n");
        return;
    }
    queue[rear]=data;
    rear++;
    printf("%d is enqueue!\n",data);
}
```

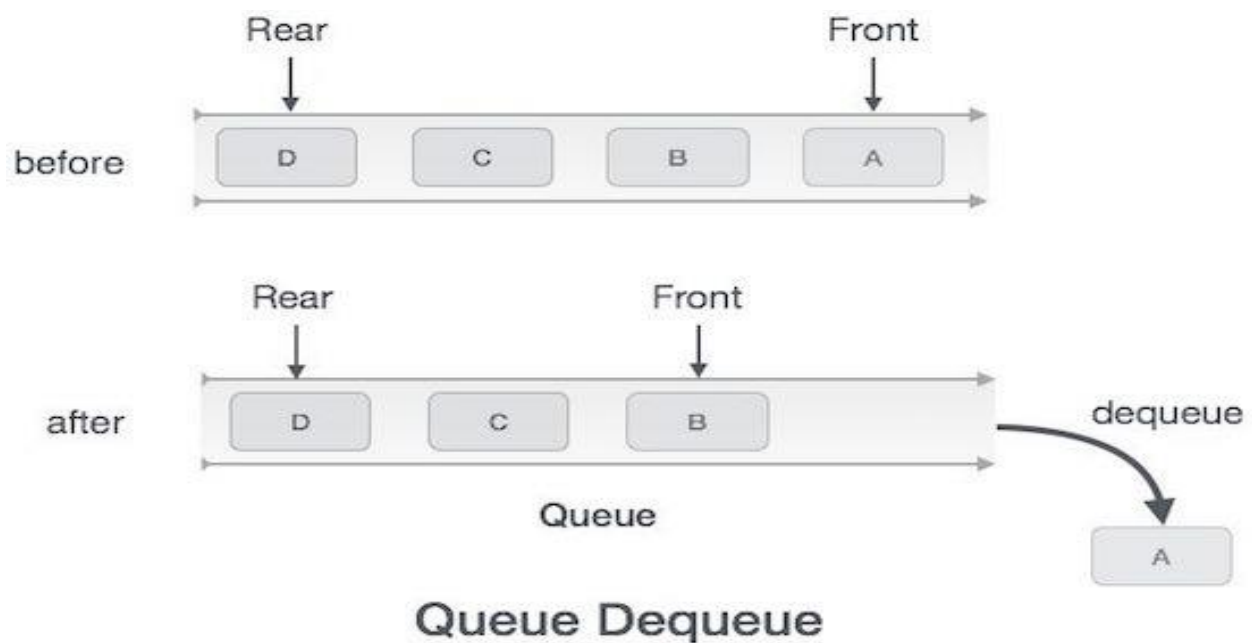
## Dequeue Operation:

**Step 1** – Check if the queue is empty.

**Step 2** – If empty then produce underflow

**Step 3** – If not then access the data where front is pointing.

**Step 4** – Increment front pointer to point to the next available data element.





## Deque Implementation:

```
int dequeue()
{
    int temp;
    if(front==rear)
    {
        printf("Queue is empty!\n");
        return;
    }
    int data=queue[front];
    front++;
    return data;
}
```

## Queue Implementation Using Linklist:

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int data;
    struct Node *next;
};
typedef struct Node queue;
void enqueue(queue *q,int data)
{
    queue *temp;
    temp=(queue*)malloc(sizeof(queue));
```

```

        temp->data=data;
        temp->next=NULL;
        while(q->next!=NULL)
        {
            q=q->next;
        }
        q->next=temp;
        printf("%d is enqueued!\n",data);
    }
int dequeue(queue *q)
{
    queue *temp;
    int data;
    if(q->next==NULL)
    {
        printf("Queue is empty!\n");
        return -1;
    }
    temp=q->next;
    data=temp->data;
    q->next=temp->next;
    free(temp);
    return data;
}
int main()
{
    queue *q;
    q=(queue*)malloc(sizeof(queue));
    q->next=NULL;
    enqueue(q,10);
    enqueue(q,5);
    enqueue(q,11);

```

```
printf("%d\n",dequeue(q));  
printf("%d\n",dequeue(q));  
printf("%d\n",dequeue(q));  
return 0;  
}
```

\*\*\*\*\*

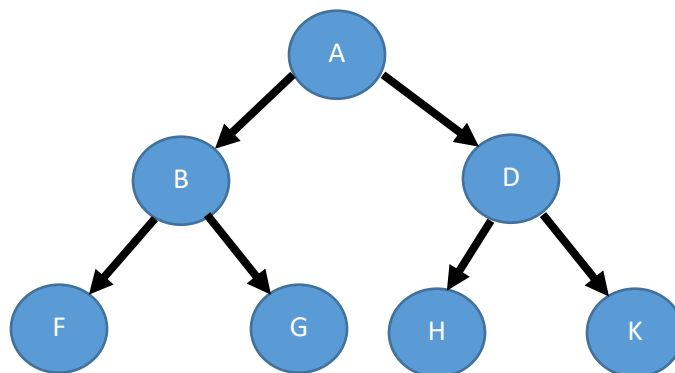
---

## TREE

---

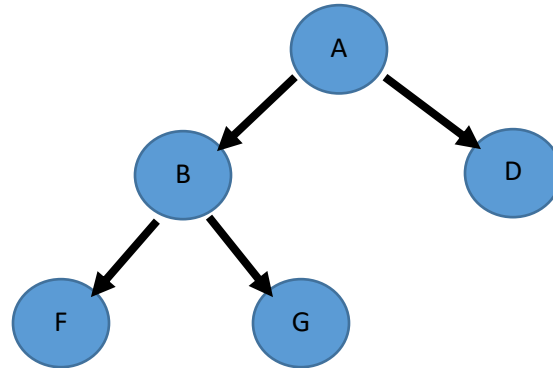
### Full Binary Tree:

Binary tree with required number of child as per level



## Complete Binary Tree:

Binary tree but childs from left to right



## Array to Tree:

0	1	2	3	4
7	5	4	6	9

→  $X[i] = \text{root}; i=0$

→ **Left** will be =  $x[2i+1]$

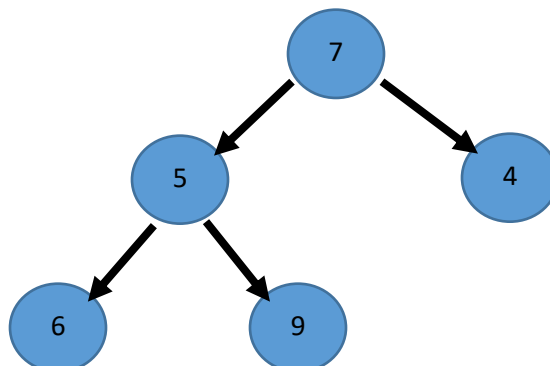
When  $i=0$   $x[1]=\text{left}$

When  $i=1$   $x[3]=\text{left}$

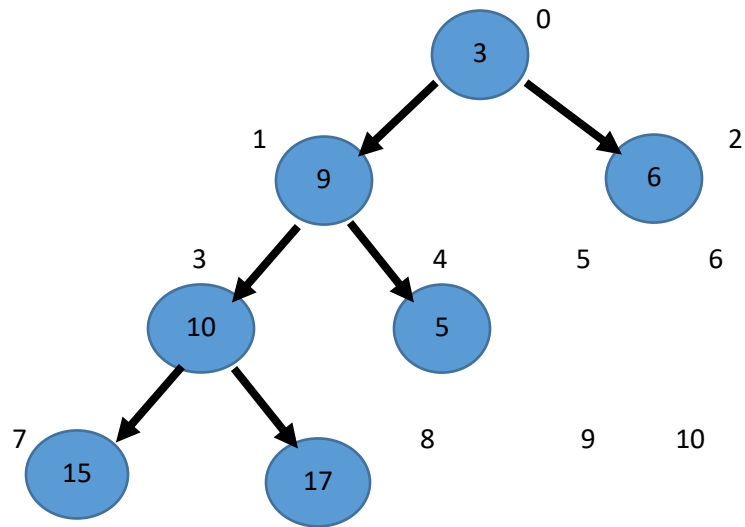
→ **Right** will be =  $x[2i+2]$

When  $i=0$   $x[2]=\text{right}$

When  $i=1$   $x[4]=\text{right}$



3	9	6	10	5			15	17	
0	1	2	3	4	5	6	7	8	9



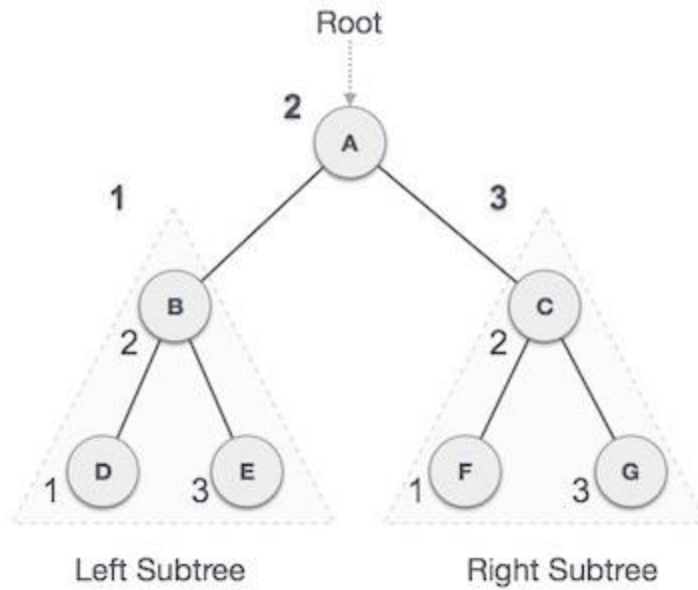
## Tree Traversal:

Traversal is a process to visit all the nodes of a tree.

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

## In-order Traversal:

In this traversal method, the left subtree is visited first, then the root and later the right subtree.



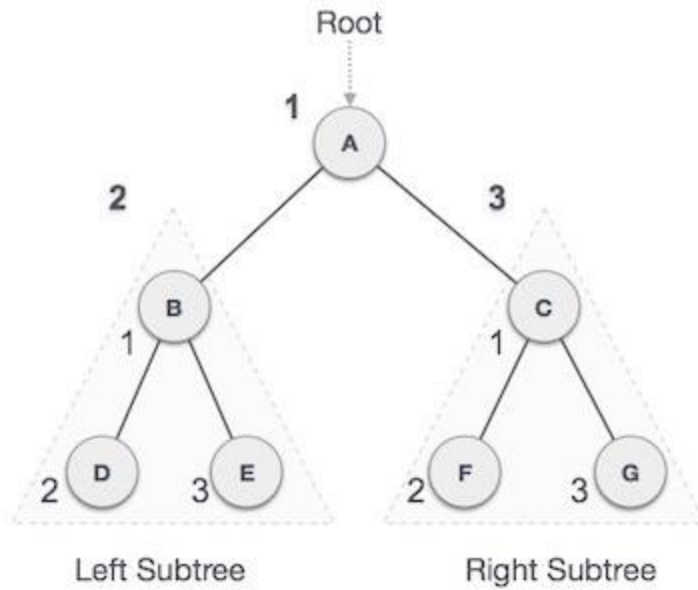
**D → B → E → A → F → C → G**

### In-order Implementation:

```
void inorder(tree *t)
{
    if(t)
    {
        inorder(t->left);
        printf("%d\n",t->data);
        inorder(t->right);
    }
}
```

### Pre-order Traversal:

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



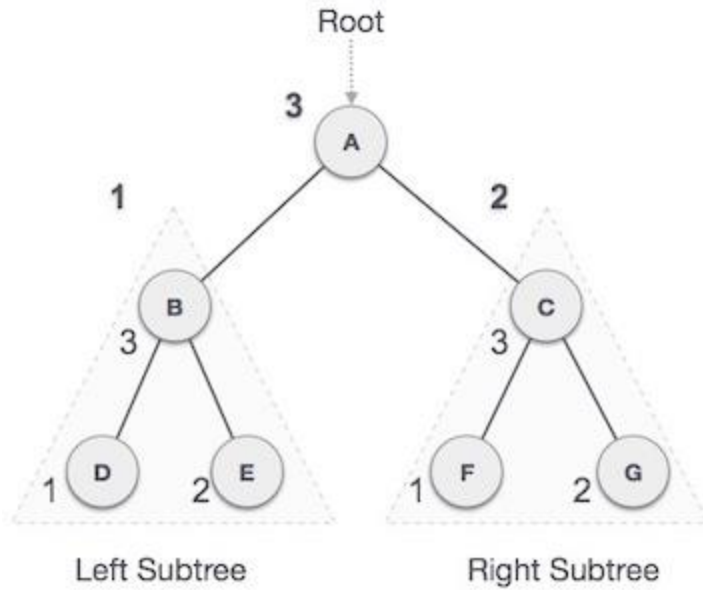
**A → B → D → E → C → F → G**

### Pre-order Implementation:

```
void preorder(tree *t)
{
    if(t)
    {
        printf("%d\n", t->data);
        preorder(t->left);
        preorder(t->right);
    }
}
```

### Post-order Traversal:

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



**D → E → B → F → G → C → A**

### Post-order Implementation:

```
void postorder(tree *t)
{
    if(t)
    {
        postorder(t->left);
        postorder(t->right);
        printf("%d\n", t->data);
    }
}
```

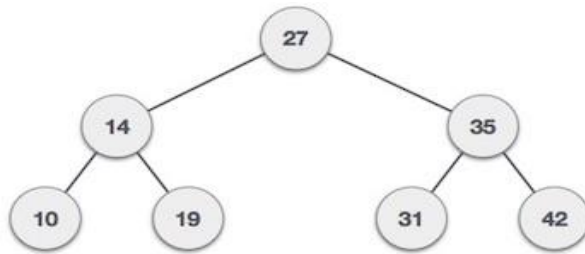
---

### **Binary Search Tree (BST):**

---

- Left child < Node
- Right child > Node





## Basic Operations:

### Insert:

To insert a data element first search from the root node. If the data is less than the key value then search an empty location in the left subtree and insert the data. Otherwise, search empty location in the right subtree and insert the data.

### Insert function Implementation:

```
void insert(tree **t, int val)
{
    tree *temp=NULL;
    if(!(*t))
    {
        temp=(tree*)malloc(sizeof(tree));
        temp->left=temp->right=NULL;
        temp->data=val;
        *t=temp;
        return;
    }
    if(val<(*t)->data)
    {
        insert(&(*t)->left, val);
    }
    else if(val>(*t)->data)
    {
        insert(&(*t)->right, val);
    }
}
```

```
}  
}
```

## Search:

Whenever searching an element start from the root node. Then if the data is less than the key value, then search in the left subtree. Else if the data is greater than the key value search in the right subtree.

## Search Implementation:

```
void search(tree *t,int data)  
{  
    if(t!=NULL)  
    {  
        if(t->data==data)  
        {  
            printf("\nYes\n");  
        }  
        else if(t->data > data)  
        {  
            search(t->left,data);  
        }  
        else if(t->data < data)  
        {  
            search(t->right, data);  
        }  
    }  
    else  
    {  
        printf("\nNo\n");  
    }  
}
```

\*\*\*\*\*

---

## HEAP

---

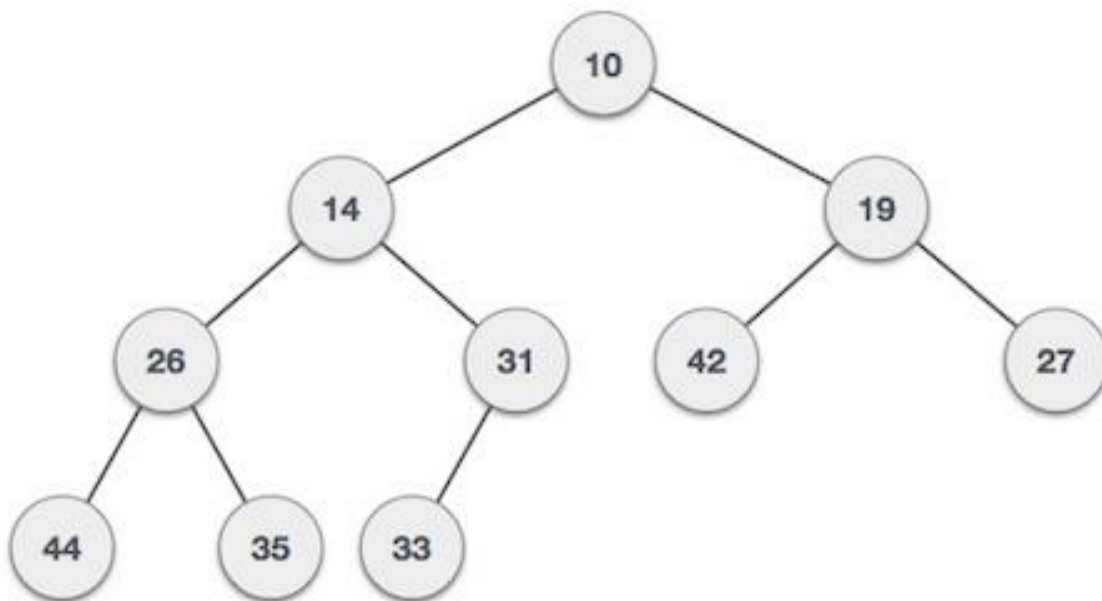
A special case of complete binary where root-node element is compared with its child and arranged accordingly

→ Root  $\geq$  left – right  
[max heap]

→ Root  $\leq$  left – right  
[min heap]

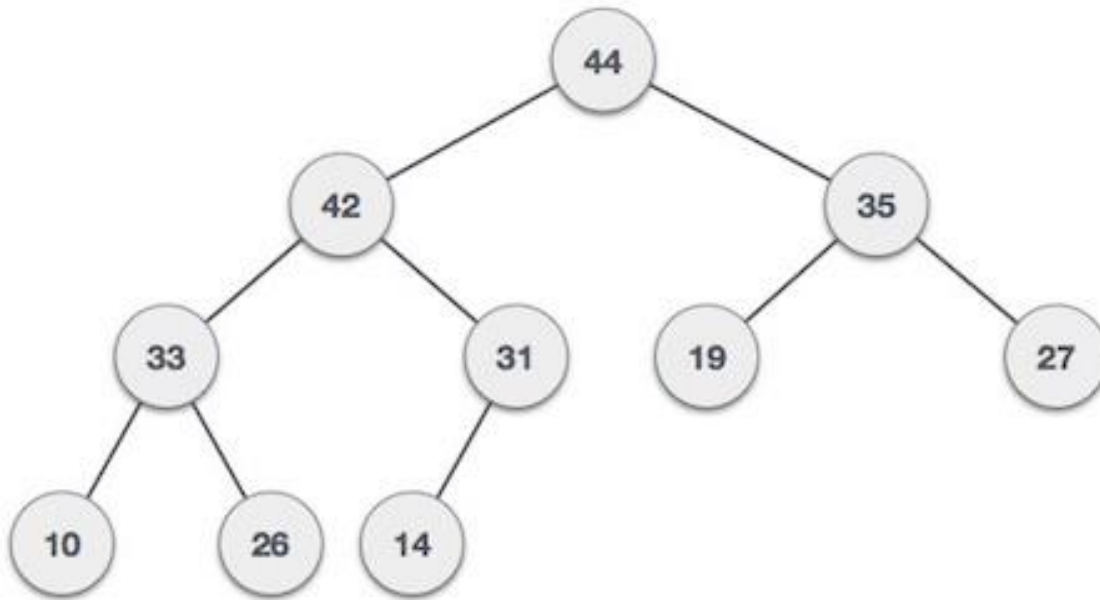
### Min Heap:

Value of the root node is less than or equal to either of its children.



### Max Heap:

Value of the root node is greater than or equal to either of its children.



### Max Heap Construction:

**Step 1** – Create a new node at the end of heap.

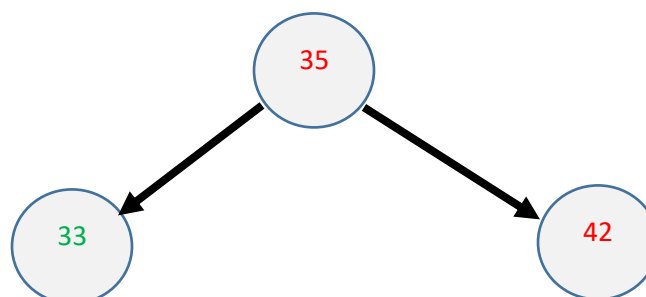
**Step 2** – Compare the value of this child node with its parent.

**Step 3** – If value of parent is less than child, then swap them.

**Step 4** – Repeat step 2 & 3 until Heap property holds.

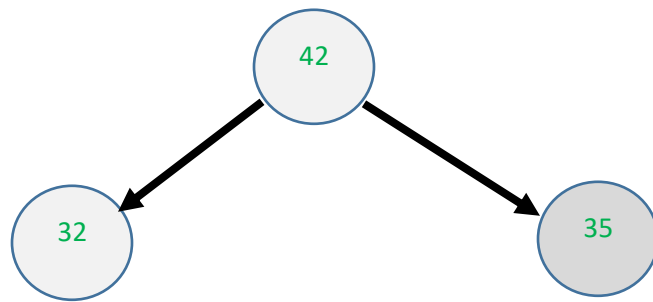
**Input** → 35 33 42 10 14 19 27 44

**Input** → 35 33 42 10 14 19 27 44

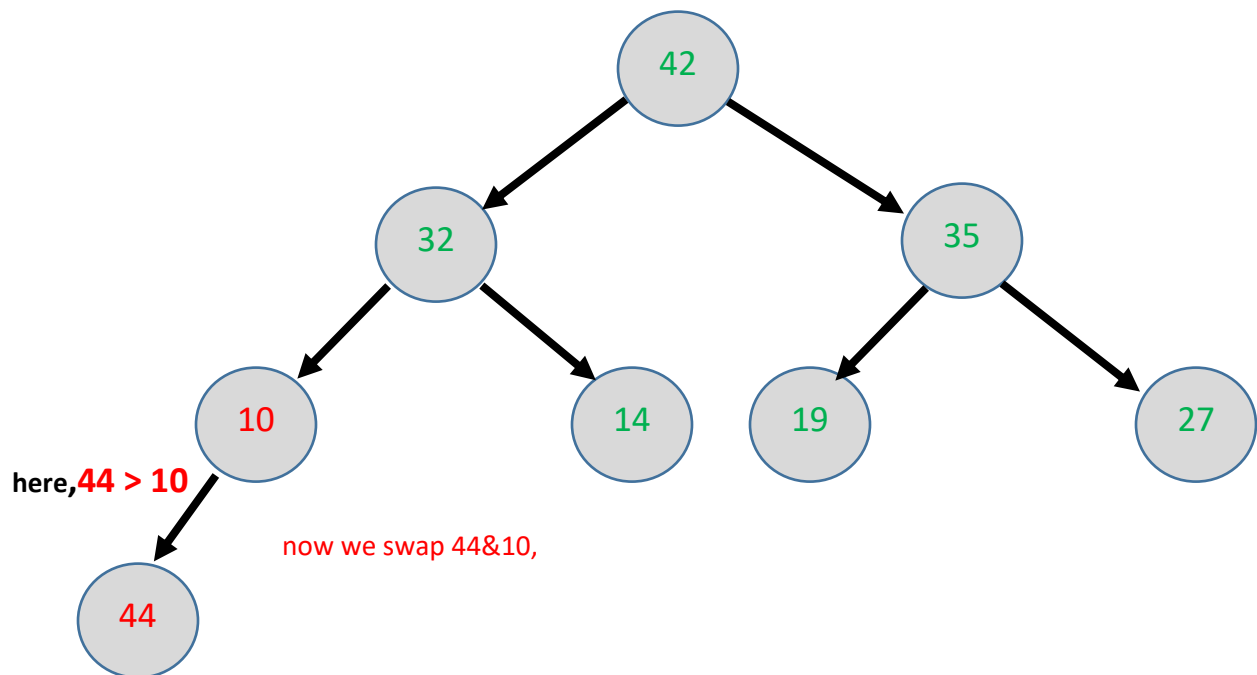


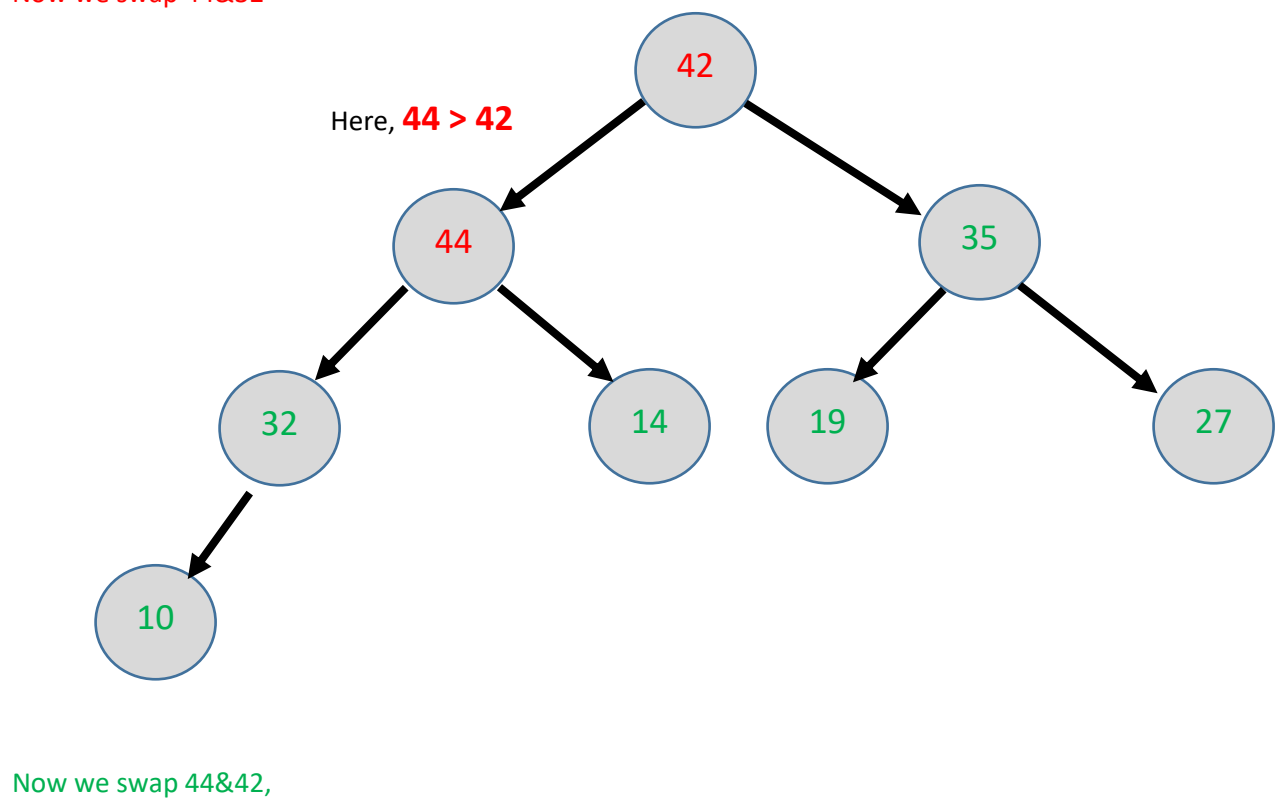
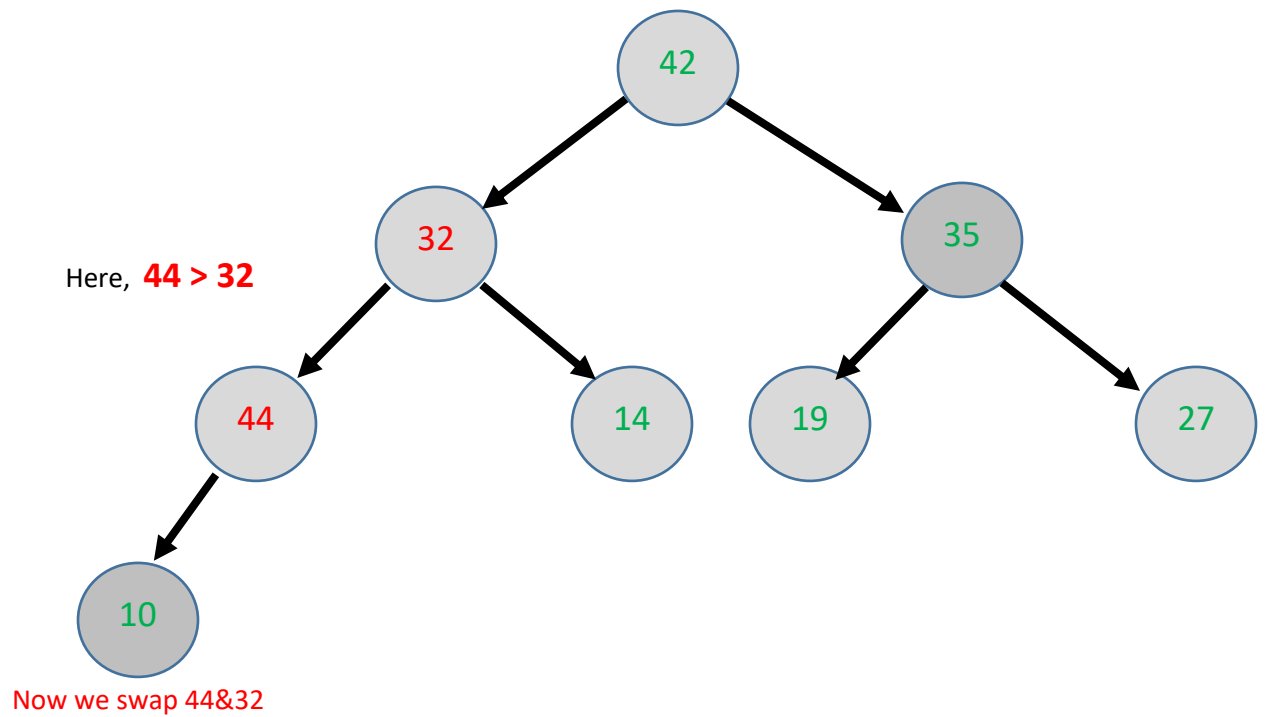
Here, "**42 > 35**"

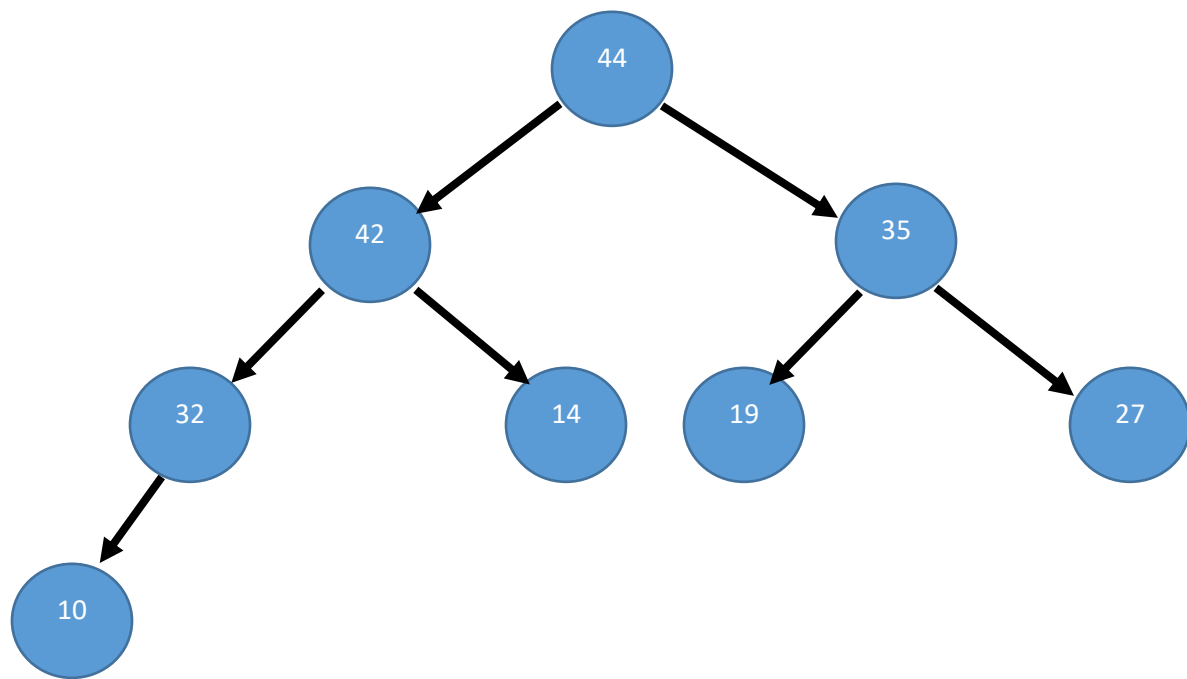
Now we swap 42&35.



Input → 35 33 42 10 14 19 27 44







\*\*\*\*\*

---

## Time Complexity:

---

Time complexity of an algorithm signifies the total time required by the program to run to completion. Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm.

Calculate  $T(n)$  : no of operations

```
1. int x = 0;           2 operations (declaration & assign)
   int sum = 0;         2 operations (declaration & assign)
   T(n) = 2+2
       = 4
```

```
2. int i, sum, n;       3 operations (3 declaration)
   sum = 0, n=5;        2 operations (assign)
   for(i = 0; i < n; i++)  n operation [looping n times]
       sum += i;         2 operations [inside each loop]
   T(n) = 3+2+(n*2)
       = 5+2n
```

The time required by an algorithm falls under three types –

- **Best Case** – Minimum time required for program execution.
- **Average Case** – Average time required for program execution.
- **Worst Case** – Maximum time required for program execution.

## Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm. Using these notation we can represent the 3 cases.

- **O Notation** (The notation  $O(n)$  is the formal way to express the upper bound of an algorithm's running time. It measures the **worst case** time complexity or the longest amount of time an algorithm can possibly take to complete.)



- **$\Omega$  Notation** (The notation  $\Omega(n)$  is the formal way to express the lower bound of an algorithm's running time. It measures the **best case** time complexity or the best amount of time an algorithm can possibly take to complete.)
- **$\theta$  Notation** (The notation  $\theta(n)$  is the formal way to express **both the lower bound and the upper bound** of an algorithm's running time.)

## Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components –

- **A fixed part** that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- **A variable part** is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity  $S(P)$  of any algorithm  $P$  is  $S(P) = C + SP(I)$ , where  $C$  is the **fixed part** and  $S(I)$  is the **variable part** of the algorithm, which depends on instance characteristic  $I$ . Following is a simple example that tries to explain the concept –

Algorithm: SUM(A, B)  
 Step 1 - START  
 Step 2 -  $C \leftarrow A + B + 10$   
 Step 3 - Stop

```
{
    int A,B,C;

    C=A+B;
}
```

Here we have three variables  $A$ ,  $B$ , and  $C$  and one constant. Hence  $S(P) = 1 + 3$ . Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

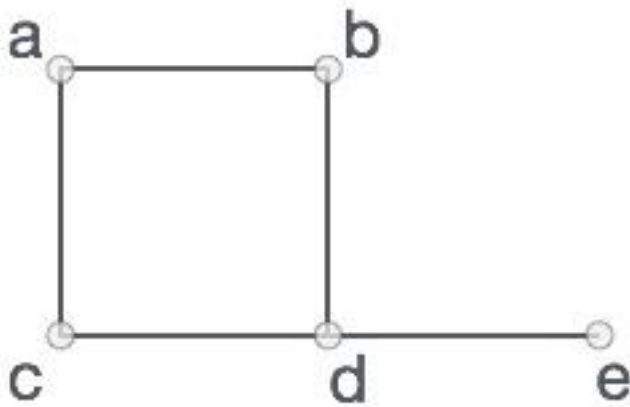
\*\*\*\*\*

---

## Graph:

---

- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links.
- Interconnected objects are called vertices (vertex)
- Links that connect the vertices are called edges.

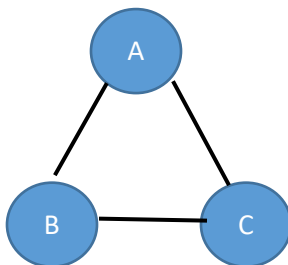


In the above graph,

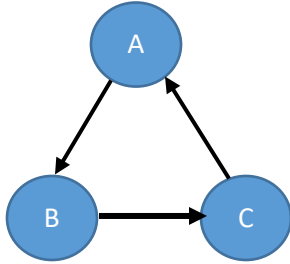
$V = \{a, b, c, d, e\}$

$E = \{ab, ac, bd, cd, de\}$

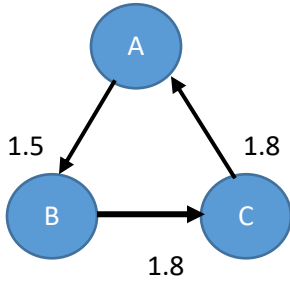
### Different Types of Graph:



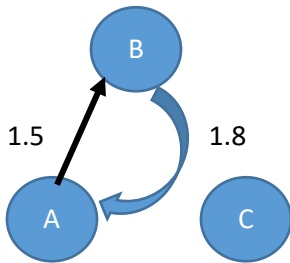
Unweighted connected graph



Unweighted directed cycle connected



Directed connected weighted



Directed weighted disconnected

\*\*\*\*\*

---

## *Presenting a Graph in Memory:*

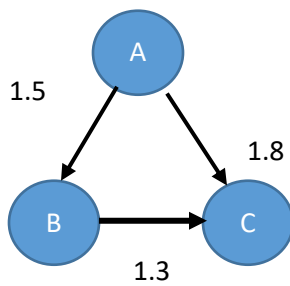
---

→ Adjacency matrix

→ Adjacency list

### Adjacency matrix:

→ Based on array

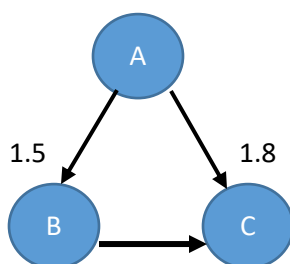


	A	B	C
A		1.5	1.8
B			1.3
C			

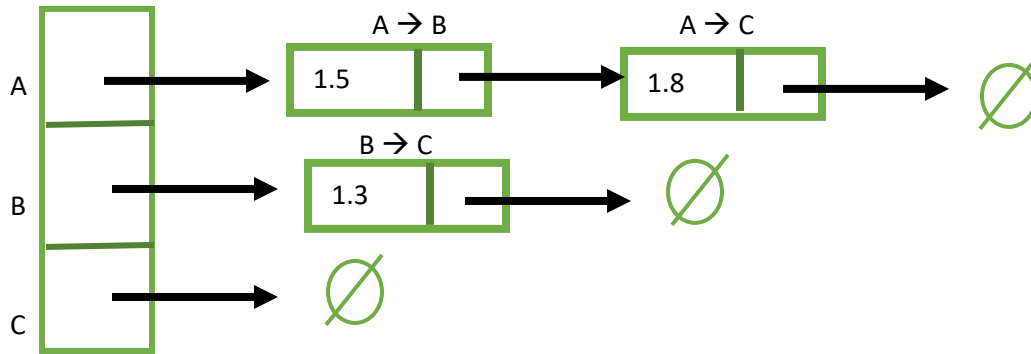
0	1.5	1.8	
0	0	1.3	
0	0	0	

### Adjacency list:

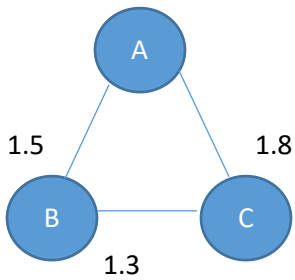
→ Using link list



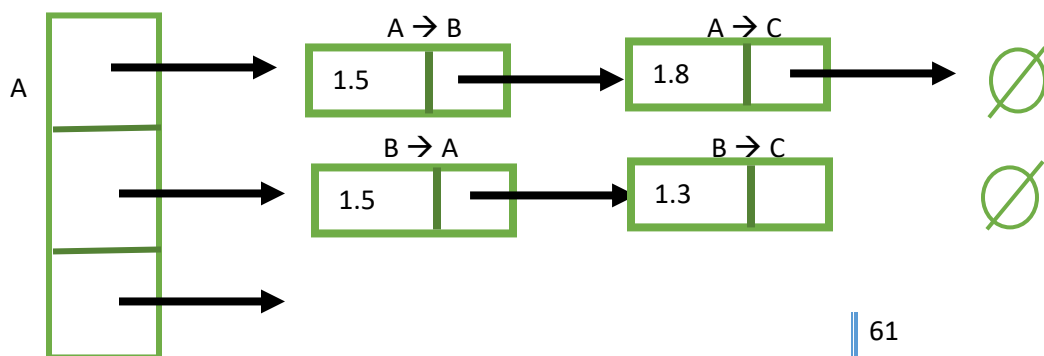
1.3



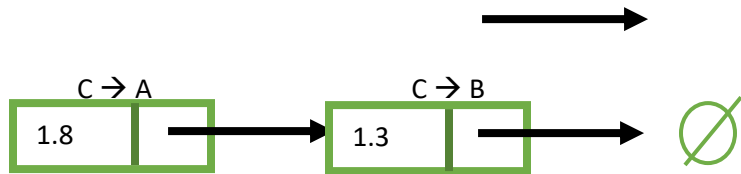
Adjacency List for undirected graph:


~~|   | A   | B   | C   |
|---|-----|-----|-----|
| A |     | 1.5 | 1.8 |
| B | 1.5 |     | 1.3 |
| C | 1.8 | 1.3 |     |~~

\*Adjacency list for undirected graph is always **symmetric**.



B

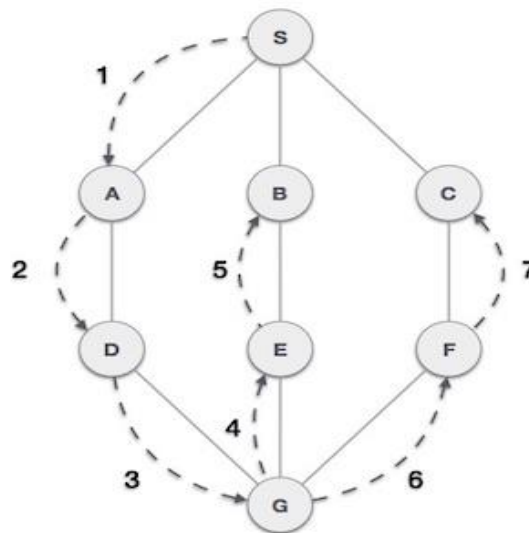


## Graph Traversal:

### DFS(depth first search):

In DFS we start from a origin node then we go deeper and deeper until we find an end node. After an end node found we find another path to another end.

Uses a stack to remember to get the next vertex:



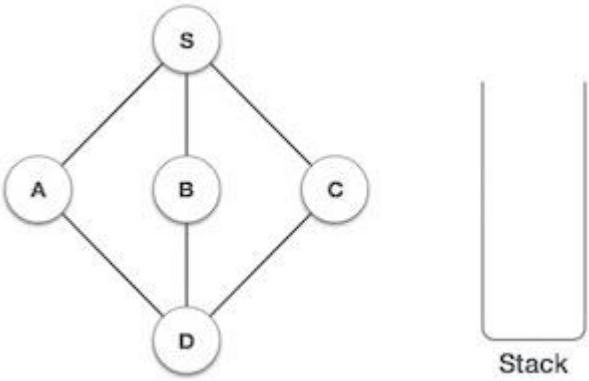
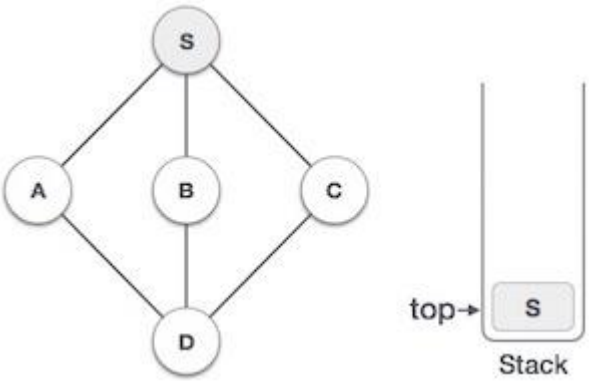
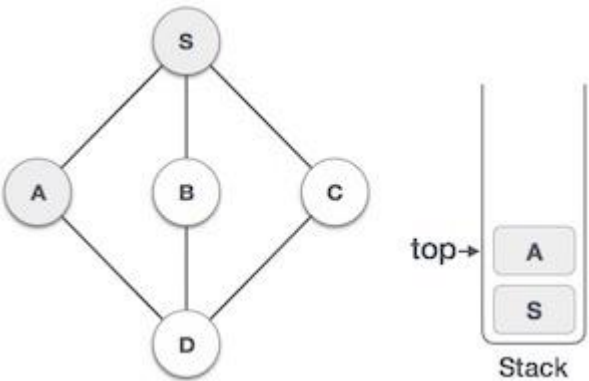
Here, we start from **S**. then visit **A**, then **D**, then **G**. now we got two way one is to **E** and other is to **F**. we visit **E**, then **B**. now there are no further unvisited vertex. So we backtrack to **G** and visit **F**, then **C**. now there are no unvisited vertex.

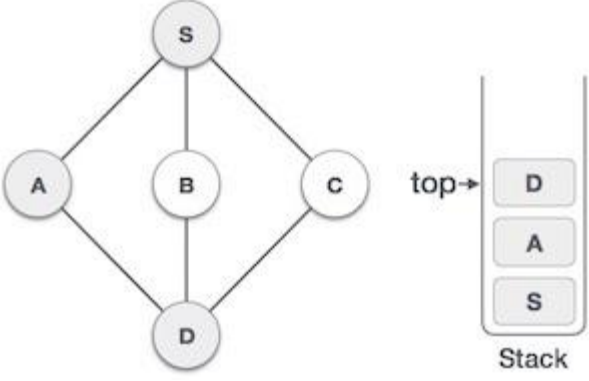
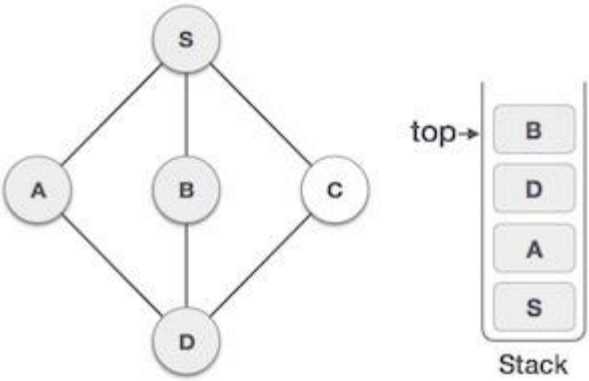
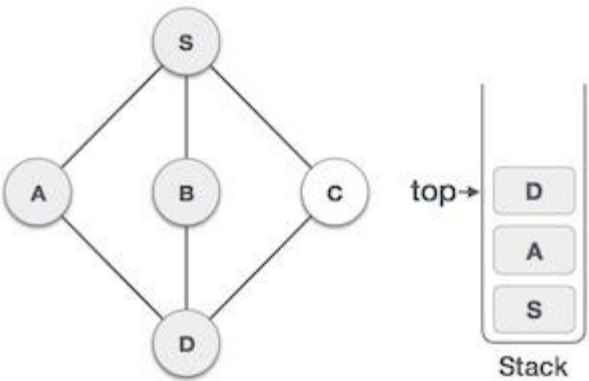
### RULES OF TRAVERSAL IN DFS:

**Rule 1** – Start from a vertex then Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

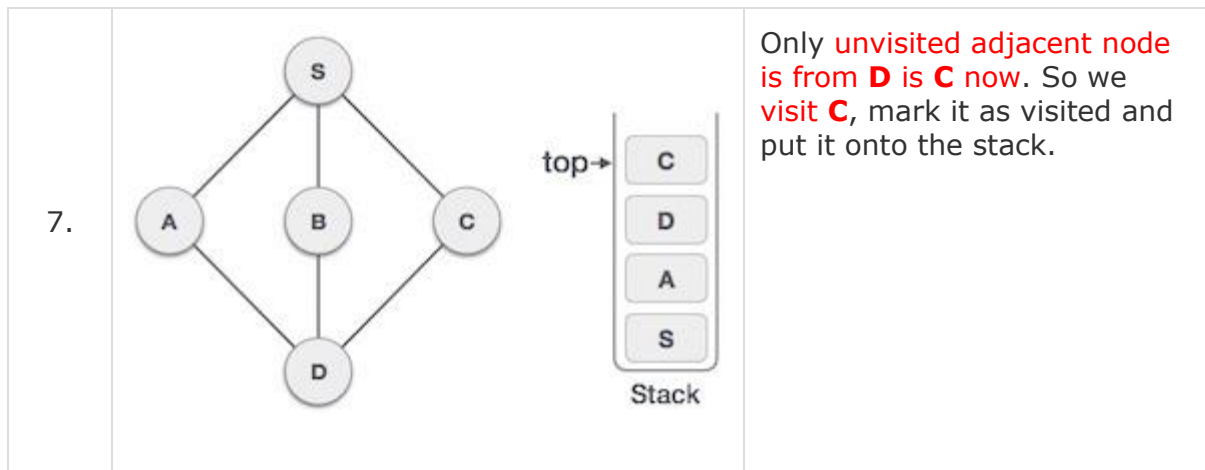
**Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

**Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1.		Initialize the stack.
2.		Mark <b>S</b> as visited and put it onto the stack. Explore any <b>unvisited adjacent node from S</b> . 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 <b>A</b> as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both <b>S and D are adjacent to A</b> but we are concerned for unvisited nodes only.

4.		<p>Visit <b>D</b> and mark it as visited and put onto the stack. Here, we have <b>B</b> and <b>C</b> nodes, which are adjacent to <b>D</b> and both are <b>unvisited</b>. However, we shall again choose in an alphabetical order.</p>
5.		<p>We choose <b>B</b>, mark it as visited and put onto the stack. Here <b>B</b> does not have any unvisited adjacent node. So, we pop <b>B</b> from the stack.</p>
6.		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find <b>D</b> to be on the top of the stack.</p>

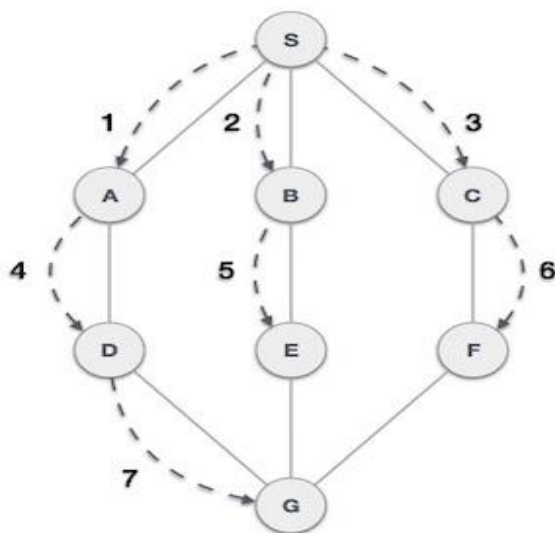




As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

## BFS(BREATH FIRST SEARCH):

In BFS firstly we visit each nodes adjacent nodes before visiting their grandchild or further away nodes.



Here we start from S then visit A then backtrack to S then visit B, then again backtrack to S and visit C.

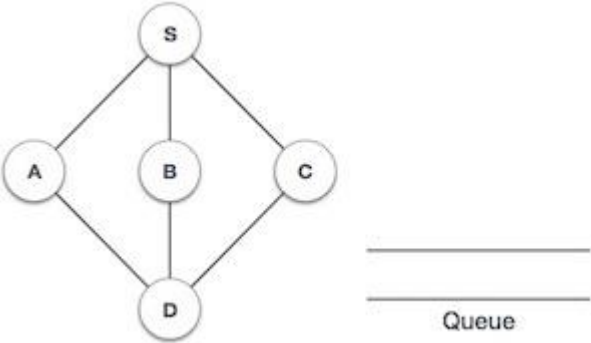
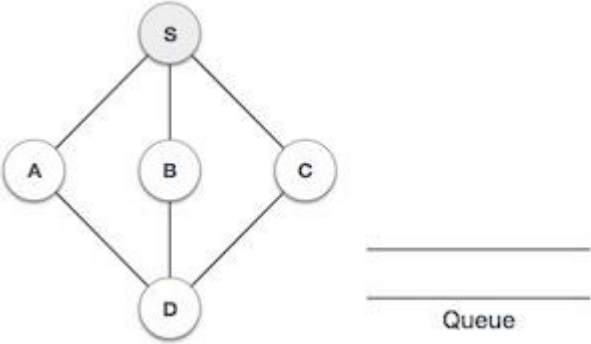
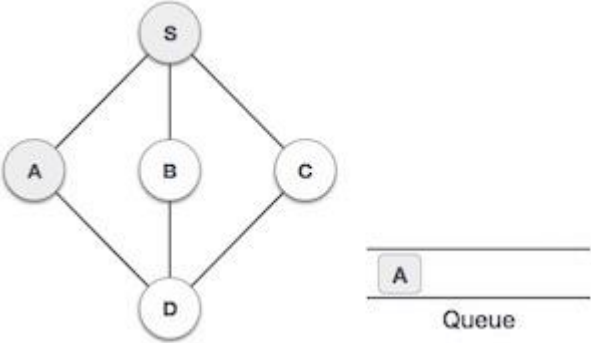
Then we backtrack to A and visit D, after that we backtrack to B and visit E, then we backtrack to C and visit F. after visiting them we again backtrack to D and visit G.

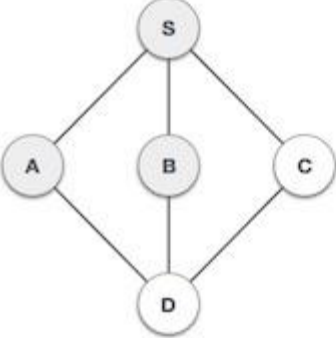
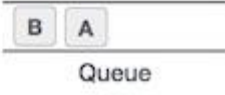
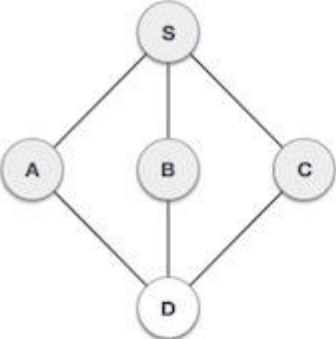
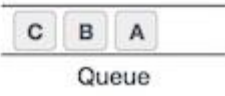
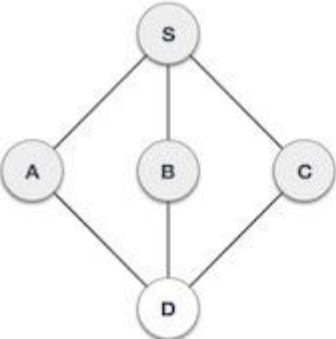
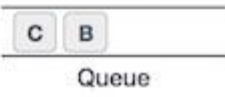
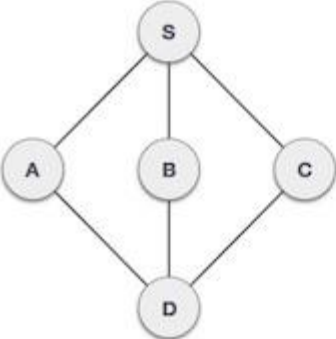
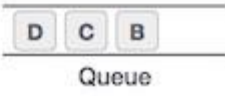
## RULES:

**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
1.		Initialize the queue.
2.		We start from visiting <b>S</b> (starting node), and mark it as visited.
3.		We then see an unvisited adjacent node from <b>S</b> . In this example, we have three nodes but alphabetically we choose <b>A</b> , mark it as visited and enqueue it.

4.	 	<p>Next, the unvisited adjacent node from <b>S</b> is <b>B</b>. We mark it as visited and enqueue it.</p>
5.	 	<p>Next, the unvisited adjacent node from <b>S</b> is <b>C</b>. We mark it as visited and enqueue it.</p>
6.	 	<p>Now, <b>S</b> is left with no unvisited adjacent nodes. So, we dequeue and find <b>A</b>.</p>
7.	 	<p>From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it as visited and enqueue it.</p>

At this stage, there 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.

\*\*\*\*\*

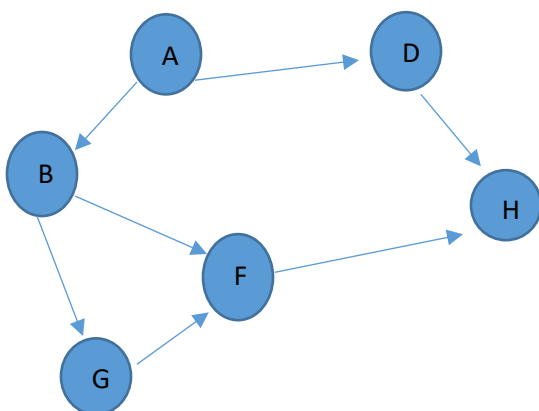
---

### TOPOLOGICAL ORDERING:

---

This type of order is based on indegree zero. Here we start with a vertex having indegree 0.

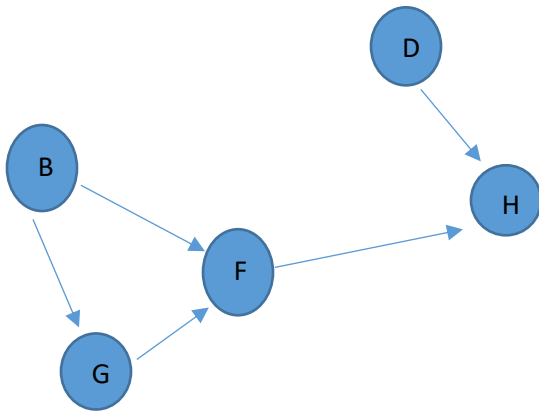
Step 1:



A" has indegree zero. So, we take A and remove it from the graph.

So, {A,

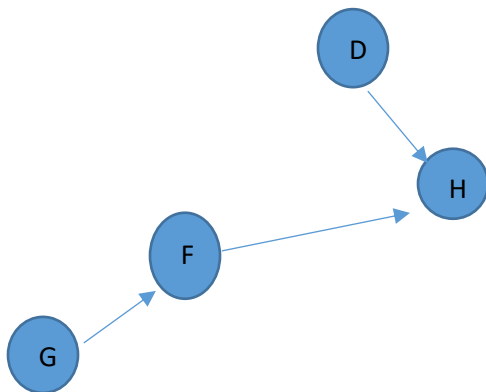
STEP 2:



Here both B&D have indegree zero. So we will take B as B has higher outdegree(2). Then we will remove B from graph.

So, {A,B,.

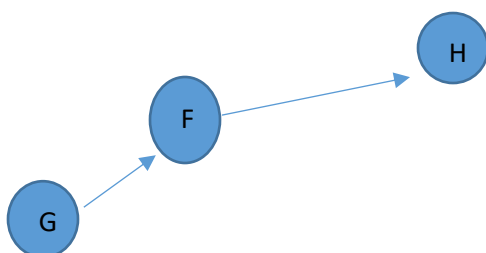
STEP 3:



Here both D&G have indegree zero. we will take D. Then we will remove D from graph.

So, {A,B,D,.

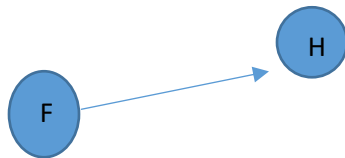
STEP 4:



we will take G. Then we will remove G from graph.

So, { A,B,D,G..

STEP 5:



we will take F. Then we will remove F from graph.

So, { A,B,D,G,F..

STEP 6:



we will take H. Then we will remove H from graph.

So, { A,B,D,G,F,H}

HERE THE ORDER BECOMES { A, B, D, G, F, H }

*(Prepared by: Md Rahim Iqbal)*