LECTURE NOTES ON

DATA STRUCTURES

I B.TECH II SEMESTER

(JNTUA-R15)

NARAYANA
ENGINEERING COLLEGE
(AUTONOMOUS)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NARAYANA ENGINEERING COLLEGE :: GUDUR
(AUTONOMOUS)
Approved by AICTE and Permanently Affiliated to JNTUA, Ananthapuramu, An ISO 9001:2015 Certified Institution, Recognized by UGC U/S 2(f) & 12(B)
Dhoorjati Nagar, Gudur, SPSR Nellore Dist.- 524101

NAAC

UNIT-1

Asymptotic NotationNotation?

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Def:Asymptotic notation of an algorithm is a mathematical representation of its complexity

In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity may be Space Complexity or Time Complexity).

we use THREE types of Asymptotic Notations and those are as follows...

1. Big - Oh (O)
2. Big - Omega (Ω)
3. Big - Theta (Θ)

Big - Oh Notation (O)

Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.

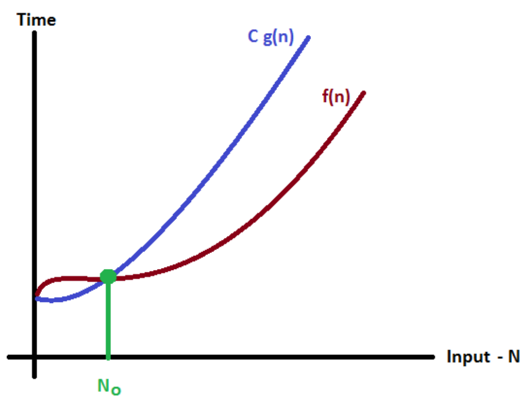
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as O(g(n)).

f(n) = O(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C x g(n)** for all values of **C > 0** and **n_0>= 1**
f(n) <= C g(n)
⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
By using Big - Oh notation we can represent the time complexity as follows...
3n + 2 = O(n)
Big - Omege Notation (Ω)
Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

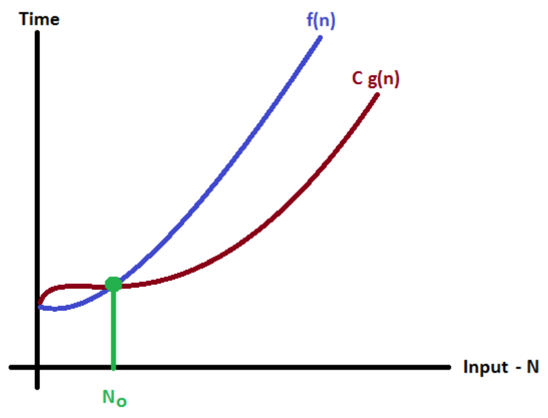
That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C x g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).

f(n) = Ω(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C x g(n) is less than f(n) which indicates the algorithm's lower bound.
Example
Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**
f(n) >= C g(n)
⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 1** and **n >= 1**.
By using Big - Omega notation we can represent the time complexity as follows...
3n + 2 = Ω(n)

Big - Theta Notation (Θ)

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

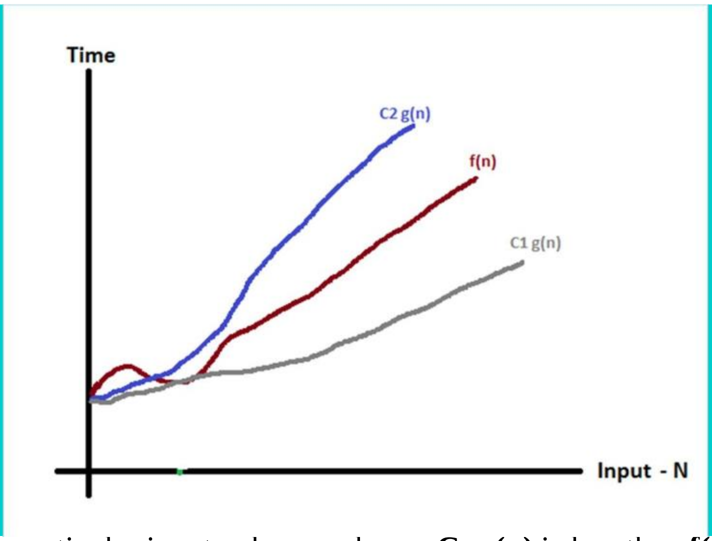
That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) >= C_2 g(n) for all n >= n_0, C_1, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always **C_1 g(n)** is less than **f(n)** and **C_2 g(n)** is greater than **f(n)** which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) >= C_2 g(n)** for all values of **C_1, C_2 > 0** and **n_0>= 1**
C_1 g(n) <= f(n) >= ⇒C_2 g(n)
C_1 n <= 3n + 2 >= C_2 n

Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 1**.
By using Big - Theta notation we can represent the time compexity as follows...
3n + 2 = Θ(n)

ARRAY:-

An array is a group of related same data items that share a common name. For instance, we can define an array name "salary" to represent a set of salaries of employees. A particular value is indicated by writing a number of subscript in brackets after the array name.

Ex: salary[10];

The above example represents the salary of the 10 employees. While the complete set of values is referred to as an array, the individual values are called elements. Arrays are classified into two types

1. One dimensional array or Single dimensional array
2. Two dimensional array
3. Multidimensional Arrays

SINGLE DIMENSIONAL ARRAYS:-

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one-dimensional array.

Declaring an Array:-

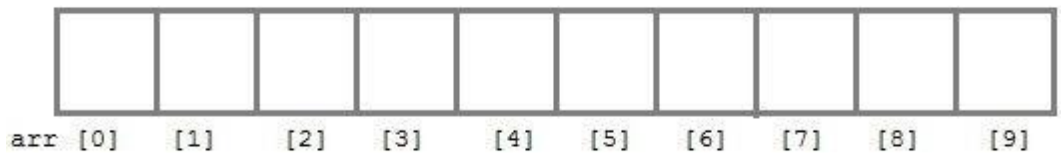
Like any other variable, arrays must be declared before they are used.

General form of array declaration is,

data-type variable-name[size];

for example :

int arr[10];



Here int is the data type, arr is the name of the array and 10 is the size of array. It means array arr can only contain 10 elements of int type. Index of an array starts from 0 to size-1 i.e first element of arr array will be stored at arr[0] address and last element will occupy arr[9].

Initialization of an Array:-

After an array is declared it must be initialized. Otherwise, it will contain garbage value (any random value). An array can be initialized at either compile time or at runtime.

Compile time Array initialization:-

Compile time initialization of array elements is same as ordinary variable initialization. The general form of initialization of array is,

type array-name[size] = { list of values };

int marks[4]={ 67, 87, 56, 77 }; //integer array initialization

```
float area[5]={ 23.4, 6.8, 5.5 };  //float array initialization

int marks[4]={ 67, 87, 56, 77, 59 };   //Compile time error
```

One important thing to remember is that when you will give more initializer(array elements) than declared array size than the compiler will give an error.

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int i;
 int arr[3]={2,3,4};    //Compile time array initialization
 for(i=0 ; i<3 ; i++)
{
 printf("%d\t",arr[i]);
 }
 getch();
}
```

Output

```
2 3  4
```

Runtime Array initialization:-

An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large array, or to initialize array with user specified values. Example,

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int arr[4];
 int i, j;
 printf("Enter array element");
 for(i=0;i<4;i++)
 {
  scanf("%d",&arr[i]);    //Run time array initialization
 }
 for(j=0;j<4;j++)
 {
  printf("%d\n",arr[j]);
 }
 getch();
}
//Write a program to read a values and display them
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,n[5];
clrscr();
printf("enter n values");
for(i=0;i<5;i++)
scanf("%d",&n[i]);
printf("The values are\n");
for(i=0;i<5;i++)
printf("%d\n",n[i]);
}
//Write a program to read up to n values and display them
#include<stdio.h>
#include<conio.h>
void main()
{
int a[32],n,i;
clrscr();
printf("enter  n");
scanf("%d",&n);
printf("enter a values");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("output  values  are\n");
for(i=0;i<n;i++)
printf("%d\n",a[i]);
}
//Write a program to find the smallest number for given numbers
#include<stdio.h>
#include<conio.h>
void main()
{
int a[100],n,i,small,big;
clrscr();
printf("Input range");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter a[%d] value",i);
scanf("%d",&a[i]);
}
```

```
small=a[0];
for(i=0;i<n;i++)
{
if(a[i]<small)
{
small=a[i];
}
big=a[0];
for(i=0;i<n;i++)
{
if(a[i]>big)
{
big=a[i];
}
}
printf("Small number is=%d",small);
printf("Big number is=%d",big);
}
//Write a program to find the biggest and smallest numbers  for given numbers
#include<stdio.h>
#include<conio.h>
void main()
{
int a[12],i,n,big,small;
clrscr();
printf("enter n");
scanf("%d",&n);
printf("enter  a  values\n");
for(i=0;i<n;i++)//i<=n-1
scanf("%d",&a[i]);
big=small=a[0];
for(i=0;i<n;i++)
{
    if(a[i]>big)
    {
    big=a[i];
    }
    if(a[i]<small)
    {
    small=a[i];
    }
}
printf("Big=%d Small=%d",big,small);
```

```
}
//Write a program the given numbers are even or odd
#include<stdio.h>
#include<conio.h>
void main()
{
int a[100],i,n;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter the a[%d] value",i);
scanf("%d",&a[i]);
}
for(i=0;i<n;i++)
{
if(a[i]%2==0)
printf("\n%d is Even number",a[i]);
else
printf("\n%d is Odd number",a[i]);
}
}
//Write a program the given numbers are positive or negative
#include<stdio.h>
#include<conio.h>
void main()
{
int a[100],i,n;
clrscr();
printf("Enter n");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter the a[%d] value",i);
scanf("%d",&a[i]);
}
for(i=0;i<n;i++)
{
if(a[i]<0)
printf("\n%d is negative",a[i]);
else
printf("\n%d is positive",a[i]);
}
}
```

Two dimensional arrays

A list of items can be given one variable name using two subscripts and such a variable is called a double-subscripted variable or a two-dimensional array. C allows us to define tables by using two dimensional arrays.

Two-dimensional array is declared as follows,

type array-name[row-size][column-size];

Note: We have not assigned any row value. It means we can initialize any number of rows. But, we must always specify number of columns, else it will give a compile time error. Here, a 2*3 multi-dimensional matrix is created.

Initializing Two Dimensional Arrays:-

Like the one dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,

int table[2][3]={0,0,0,1,1,1};

Initialize the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can be equivalently written as

int table[2][3]={{0,0,0},{1,1,1}};

We can also initialize a two dimensional array in the form of a matrix as shown below:

```
     int table[2][3]={
                        {0,0,0},
                         {1,1,1}
                                        };
```

```
//Write a program to enter a matrix A.
#include<stdio.h>
#include<conio.h>
void main()
{
  int  a[12][12],ar,ac,r,c;
  clrscr();
  printf("enter ar ac");
  scanf("%d%d",&ar,&ac);
  printf("enter  a  matrix");
  for(r=0;r<ar;r++)
  {
  for(c=0;c<ac;c++)
  {
  scanf("%d",&a[r][c]);
  }
  }
  printf("a  matrix  is\n");
  for(r=0;r<ar;r++)
  {
```

```
  for(c=0;c<ac;c++)
  {
  printf("%d\t",a[r][c]);
  }
  printf("\n");
  }
  }
//Write a program to add the two matrix
#include<stdio.h>
#include<conio.h>
void main()
{
  int  a[12][12],b[10][12],d[12][10],br,bc,ar,ac,r,c;
  clrscr();
  printf("enter ar  ac br  bc");
  scanf("%d%d%d%d",&ar,&ac,&br,&bc);
  if(ar==ac&&br==bc)
  {
  printf("enter  a  matrux");
  for(r=0;r<ar;r++)
  {
  for(c=0;c<ac;c++)
  {
  scanf("%d",&a[r][c]);
  }
  }
          printf("enter  b  matrux");
  for(r=0;r<br;r++)
  {
  for(c=0;c<bc;c++)
  {
  scanf("%d",&b[r][c]);
  }
  }
             for(r=0;r<ar;r++)
             {
             for(c=0;c<ac;c++)
             {
          d[r][c]=a[r][c]+b[r][c];
          }
          }
          printf("addition  of  a & b\n");
          for(r=0;r<ar;r++)
```

```
                {
                for(c=0;c<ac;c++)
                {
                printf("%d\t",d[r][c]);
                }
                printf("\n");
                }
}
else
printf("addition is  not  possible");
}
//Write a program to The multiplication of 2 matrix
#include<stdio.h>
#include<conio.h>
void main()
{
   int  a[12][12],b[10][12],d[12][10],br,bc,ar,ac,r,c,k;
   clrscr();
   printf("enter ar  ac br  bc");
   scanf("%d%d%d%d",&ar,&ac,&br,&bc);
   if(ar==bc)
   {
   printf("enter  a  matrux");
   for(r=0;r<ar;r++)
   {
   for(c=0;c<ac;c++)
   {
   scanf("%d",&a[r][c]);
   }
   }
                printf("enter  b  matrux");
   for(r=0;r<br;r++)
   {
   for(c=0;c<bc;c++)
   {
   scanf("%d",&b[r][c]);
   }
   }
                    for(r=0;r<ar;r++)  //or  bc
                    {
                    for(c=0;c<bc;c++)  //or  ar
                    {
                    d[r][c]=0;
                    for(k=0;k<ar;k++)    //or  bc
                    {
                d[r][c]=a[r][k]*b[k][c]+d[r][c];
                }
                }
                }
                printf("addition  of  a & b\n");
                for(r=0;r<ar;r++)
                {
                for(c=0;c<ac;c++)
                {
                printf("%d\t",d[r][c]);
                }
                printf("\n");
                }
}
else
printf("Multiplication  is  not  possible");
}
```

Multidimensional Arrays

In C/C++, we can define multidimensional arrays in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).
General form of declaring N-dimensional arrays:

data_type array_name[size1][size2] ... [sizeN];

data_type: Type of data to be stored in the array.
        Here data_type is valid C/C++ data type
array_name: Name of the array
size1, size2, ... ,sizeN: Sizes of the dimensions

Examples:

Two dimensional array:

int two_d[10][20];

Three dimensional array:

int three_d[10][20][30];

Size of multidimensional arrays
Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.
For example:
The array **int x[10][20]** can store total (10*20) = 200 elements.
Similarly array **int x[5][10][20]** can store total (5*10*20) = 1000 elements.

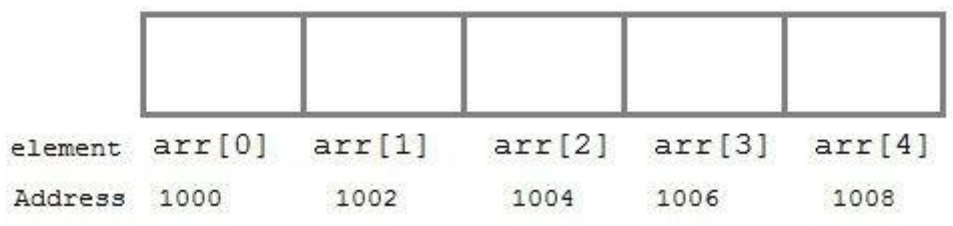
Pointer Arrays

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array arr,

```
int arr[5] = { 1, 2, 3, 4, 5 };
```

Assuming that the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:



Here variable arr will give the base address, which is a constant pointer pointing to the first element of the array, arr[0]. Hence arr contains the address of arr[0] i.e 1000. In short, arr has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.

arr is equal to &arr[0] by default

We can also declare a pointer of type int to point to the array arr.

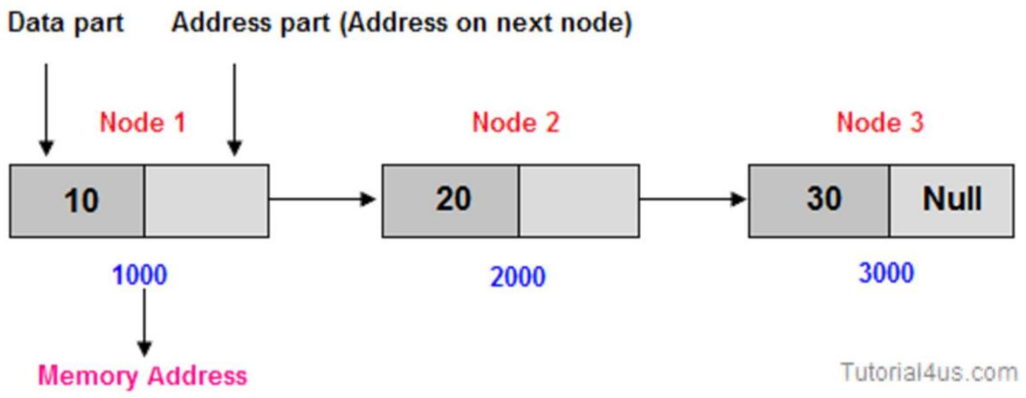
```
int *p;
p = arr;
// or,
p = &arr[0];    //both the statements are equivalent.
```

Now we can access every element of the array arr using p++ to move from one element to another.

NOTE: You cannot decrement a pointer once incremented. p-- won't work.

LINKED LIST

Linked list is a special type of data structure where all data elements are linked to one another. Linked list is the collection of nodes and every nodes contains two parts data part and address part.



Why use Linked List

Suppose you want ot store marks of 50 students, so need to write code like below;

Example

```
int marks[50];
```

But some time you need to store more than 50 students marks, in that case you can not increase memory of array, and some time you need to store less than 50 students marks in this case extra memory will be wastage. To overcome this problem you need to use Linked List because in linked list memory will be created at run time.

Advantages of linked list

- Linked List is Dynamic data Structure.
- You can change size of Linked List during program run time.
- Insertion and Deletion Operations are Easier, you can insert any node at any place and also delete any node easily..
- No memory wastage ,i.e no need to pre-allocate memory
- Faster Access time,can be expanded in constant time without memory overhead
- You can easily implement Linear Data Structures such as Stack,Queue using Linked list

Dis-Advantages of linked list

- **Need more memory:** For store data in linked list you need more memory space, you need memory space for both data and address part.

In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.

```
// A linked list node

struct Node

{

  int data;

  struct Node *next;
```

};

Linked lists are classified into 3 types

1.Singly linked list
2.Doubly linked list
3.Circular linked list
4.Doule circular linked list

Singly linked list:-

A singly linked list a linear data structure in which each node contains only one link field.
The following diagram illustrates the singly linked list.

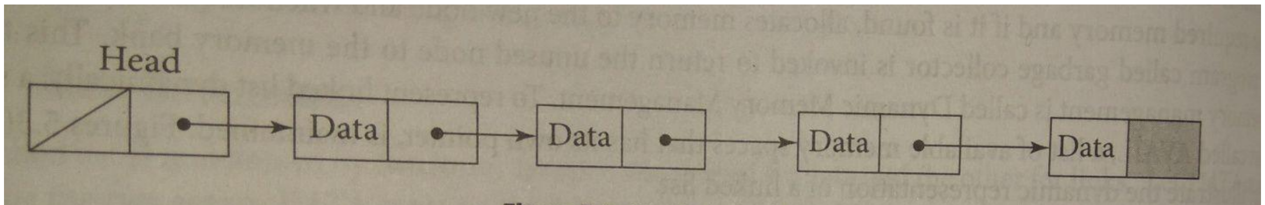


Figure 5.4 Singly linked list

Operations:

The following are the operations of single linked list
1.Traverasing
2.Searching
3.Insertion
4.Deletion
Taversing:-
Traversing the list implies visit each every node in the list from the first node to last node only once.

Algorithm:
```
struct node *new_node = start;
printf("\n\nList elements are - \n");
while(new_node != NULL)
{
    printf("%d --->",new_node->data);
    new_node = new_node->next;
}
```

Insertion:-

Insertion is an operation to add a node into the list. In a singly linked list, a node can be inserted at three different locations

- Insertion as first node
- Insertion as last node

- Insertion as the specified position

Insertion as first node

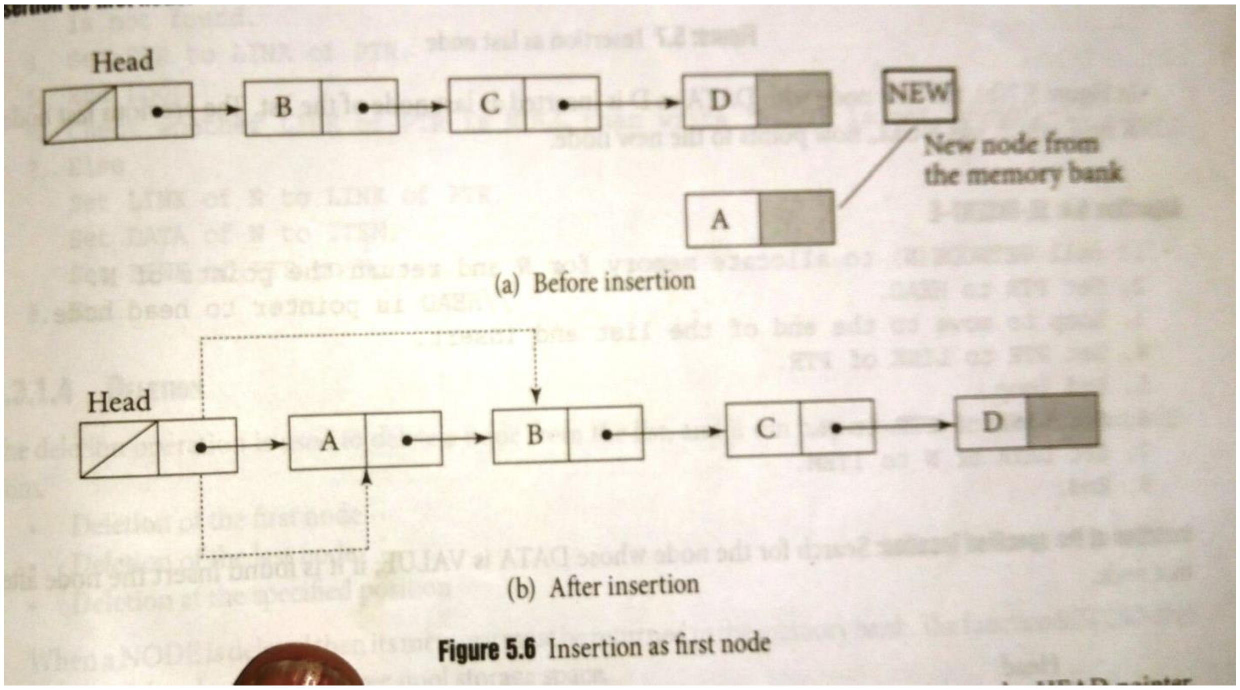


Figure 5.6 Insertion as first node

In above diagram, the node with data A is inserted as first node of the list and now the HEAD pointer points to the node A, previously which was pointing to the node B and the LINK of node A points to the node B.

Add to beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

Algorithm:

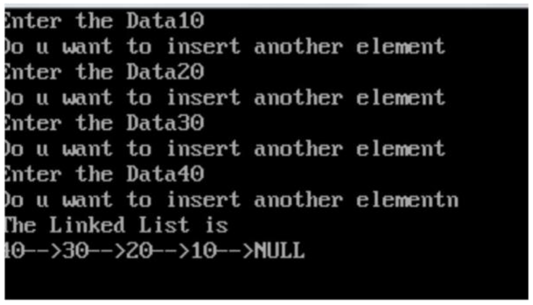
```
struct node *newNode;
newNode = (struct node *)malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
```

```
#include<stdio.h>
#include<conio.h>
struct node
{
int data;
struct node *next;
};
struct node *start=NULL;
```

```
void insert()
{
struct node *new_node;
new_node=(struct node *)malloc(sizeof(struct node *));
printf(" Enter the Data");
scanf("%d",&new_node->data);
new_node->next=NULL;
if(start==NULL)
{
start=new_node;
}
else
{
new_node->next=start;
start=new_node;
}
}
void display()
{
struct node *new_node;
new_node=start;
printf("The Linked List is\n");
while(start!=NULL)
{
printf("%d-->",start->data);
start=start->next;
}
printf("NULL");
}
int main()
{
char ch;
clrscr();
do
{
insert();
printf(" Do u want to insert another element");
ch=getche();
printf("\n");
}
while(ch!='n');
display();
getch();
```

return 0;
}

OUTPUT



```
nter the Data10
o u want to insert another element
nter the Data20
o u want to insert another element
nter the Data30
o u want to insert another element
nter the Data40
o u want to insert another elementn
he Linked List is
0-->30-->20-->10-->NULL
```

Insertion as last node:-

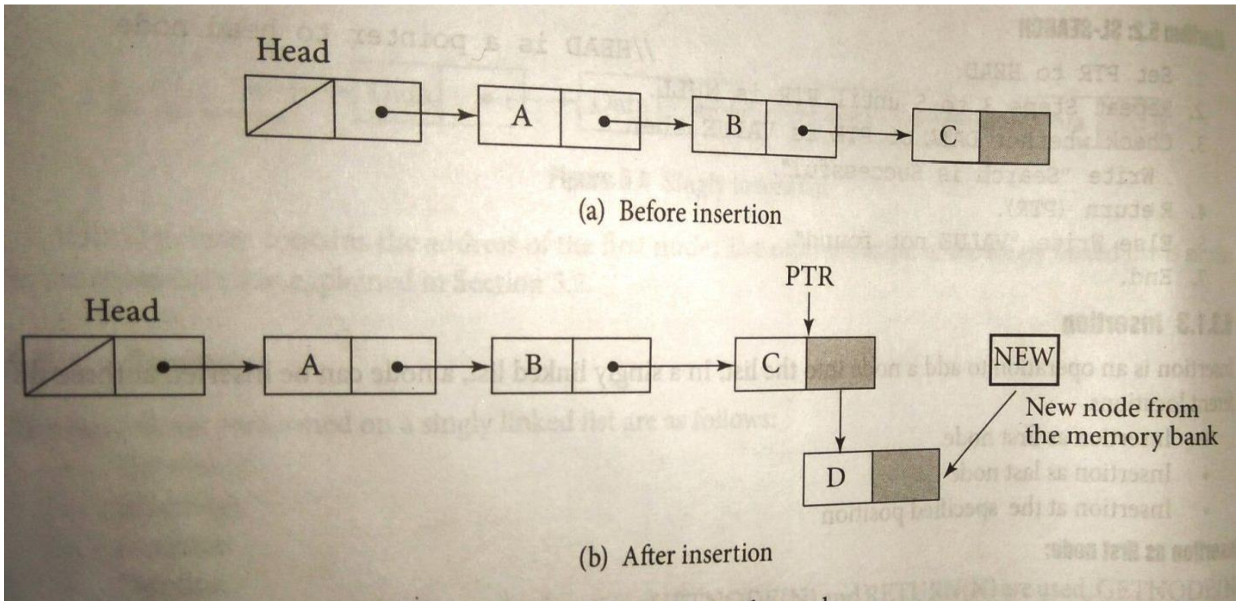


Figure: 5.7 Insertion as last node

In above diagram, the new node with DATA as D is inserted as last node of the list. The previous last node LINK field ,which was NULL, now it point to new node.

Add to end

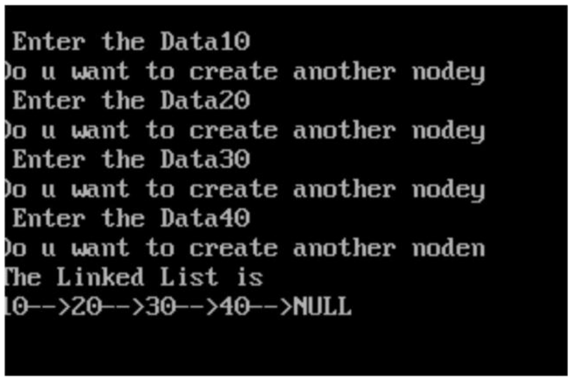
- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

Algorithm:-

```
struct node *newNode;
newNode = (struct node *)malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;

struct node *temp = head;
```

```
while(temp->next != NULL){
  temp = temp->next;
}
temp->next = newNode;
```

```
#include<stdio.h>
#include<conio.h>
struct node
{
int data;
struct node *next;
};
struct node *start=NULL;
int main()
{
char ch;
struct node *new_node,*current;
clrscr();
do
{
new_node=(struct node *)malloc(sizeof(struct node *));
printf("\n Enter the Data");
scanf("%d",&new_node->data);
new_node->next=NULL;
if(start==NULL)
{
start=new_node;
current=new_node;
}
else
{
current->next=new_node;
current=new_node;
}
printf("Do u want to create another node");
ch=getche();
}while(ch!='n');
printf("\nThe Linked List is\n");
while(start!=NULL)
{
printf("%d-->",start->data);
start=start->next;
}
printf("NULL");
getch();
return 0;
}
```



```
 Enter the Data10
Do u want to create another nodey
 Enter the Data20
Do u want to create another nodey
 Enter the Data30
Do u want to create another nodey
 Enter the Data40
Do u want to create another noden
The Linked List is
10-->20-->30-->40-->NULL
```

Insertion as the specified position:-

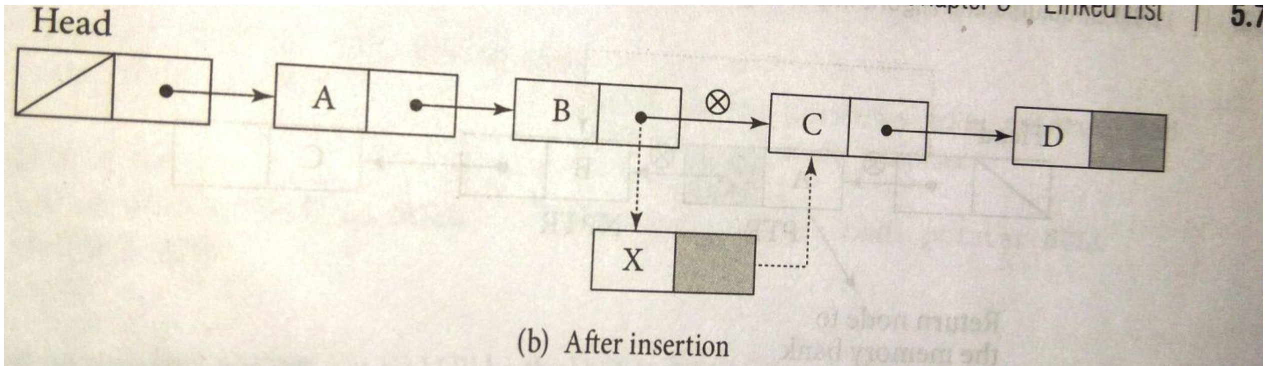


Figure 5.8 Inspection at specified location

In above diagram, the new node with DATA as X is inserted between the nodes B and C. The LINK field of B pointing to new node and the link of the new node points to the node C. Let value of be the DATA of the node after which the node is inserted. Here B is the key node after which the new node has to the inserted.

Add to middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

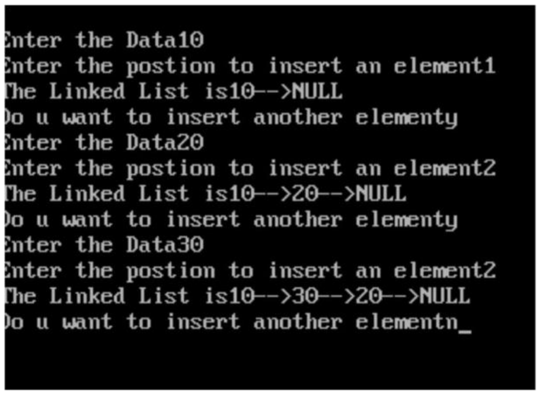
Algorithm:-

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
struct node *temp = head;
for(int i=1; i < position-1; i++) {
      temp = temp->next;
   }
```

```
newNode->next = temp->next;
temp->next = newNode;
```

```
#include<stdio.h>
struct node
{
int data;
struct node *next;
};
struct node *head=NULL;
void insert()
{
struct node *new_node;
int n,i;
new_node=(struct node *)malloc(sizeof(struct node *));
printf("\nEnter the Data");
scanf("%d",&new_node->data);
new_node->next=NULL;
printf("Enter the postion to insert an element");
scanf("%d",&n);
if(n==1)
{
new_node->next=head;
head=new_node;
}
else
{
struct node *temp=head;
for(i=1;i<n-1;i++)
{
temp=temp->next;
}
new_node->next=temp->next;
temp->next=new_node;
}
}
void display()
{
struct node *new_node;
new_node=head;
printf("The Linked List is");
while(new_node!=NULL)
{
```

```
printf("%d-->",new_node->data);
new_node=new_node->next;
}
printf("NULL");
}
int main()
{
char ch;
clrscr();
do
{
insert();
display();
printf("Do u want to insert another element");
ch=getche();
printf("\n");
}
while(ch!='n');
getch();
return 0;
}
```



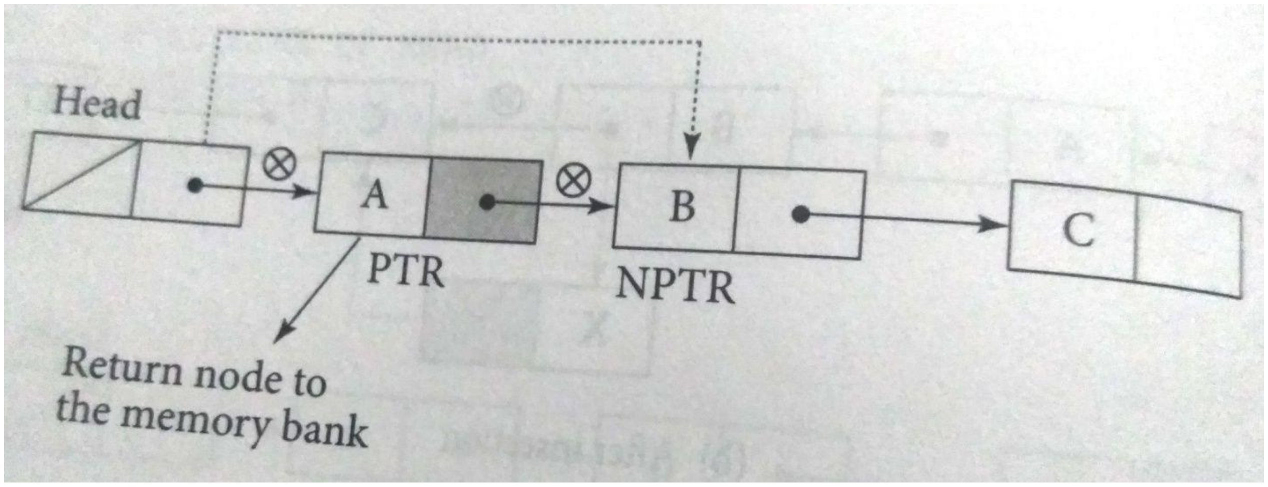
```
Enter the Data10
Enter the postion to insert an element1
The Linked List is10-->NULL
Do u want to insert another elementy
Enter the Data20
Enter the postion to insert an element2
The Linked List is10-->20-->NULL
Do u want to insert another elementy
Enter the Data30
Enter the postion to insert an element2
The Linked List is10-->30-->20-->NULL
Do u want to insert another elementn_
```

Deletion:-

Deletion is an operation to delete a node into the list. In a singly linked list, a node can be deleted at three different locations

- Deletion as first node
- Deletion as last node
- Deletion as the specified position

Deletion as first node:-



In above figure, the first node of the list is deleted and the head nodes LINK field pointing to the deleted node now points to the next node. The LINK field of tnode to be deleted is set to NULL.

Delete from beginning

- Point head to the second node

```
head = head->next;
```

Deletion as last node:-

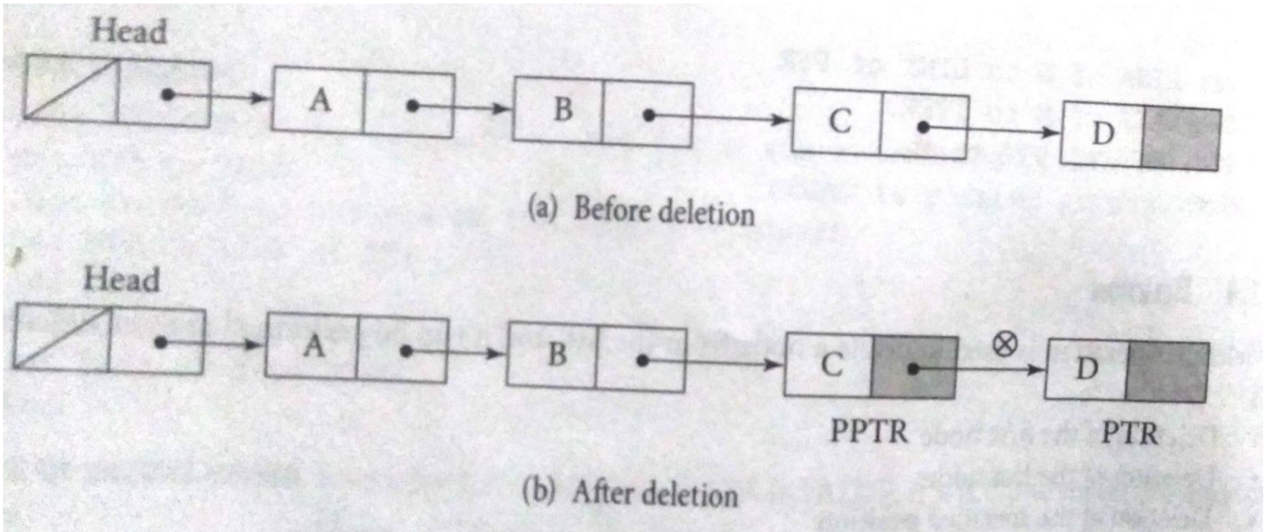


Figure 5.10 Deletion of the last node

Delete from end

- Traverse to second last element
- Change its next pointer to null

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

Deletion as the specified position:-

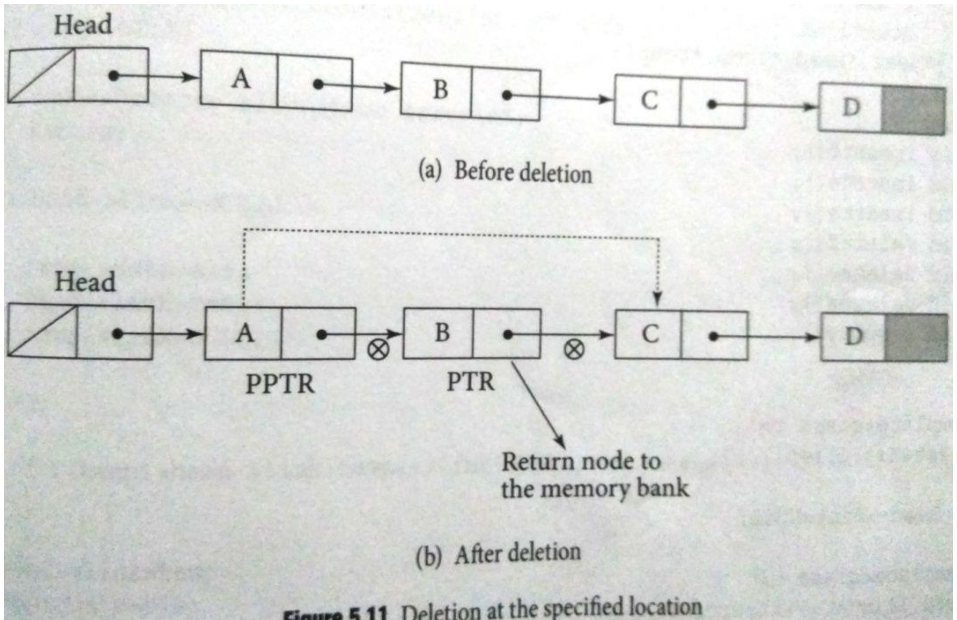


Figure 5.11 Deletion at the specified location

Delete from n th position

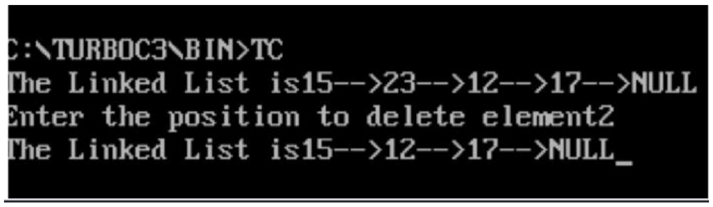
- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
   if(temp->next!=NULL) {
       temp = temp->next;
   }
}
temp->next = temp->next->next;
```

```
#include<stdio.h>
#include<conio.h>
struct node
{
int data;
struct node *next;
};
struct node *head=NULL;
void insert(int x )
{
struct node *new_node;
new_node=(struct node *)malloc(sizeof(struct node *));
new_node->data=x;
new_node->next=head;
head=new_node;
}
void display()
{
```

```
struct node *new_node;
new_node=head;
printf("The Linked List is");
while(new_node!=NULL)
{
printf("%d-->",new_node->data);
new_node=new_node->next;
}
printf("NULL");
}
void delete(int n)
{
struct node *temp1=head,*temp2;
if(n==1)
{
head=temp1->next;
free(temp1);
}
else
{
int i;
for(i=1;i<=n-2;i++)
{
temp1=temp1->next;
}
temp2=temp1->next;
temp1->next=temp2->next;
free(temp2);
}
}
int main()
{
int n;
insert(17);
insert(12);
insert(23);
insert(15);
display();
printf("\nEnter the position to delete element");
scanf("%d",&n);
delete(n);
display();
getch();
```

```
return 0;
}
```



```
C:\TURBOC3\BIN>TC
The Linked List is15-->23-->12-->17-->NULL
Enter the position to delete element2
The Linked List is15-->12-->17-->NULL_
```

Complete program for linked list operations

Here is the complete program for all the linked list operations we learnt till now. Lots of edge cases have been left out to make the program short.

We suggest you to just have a look at the program and try to implement it yourself.

Also, notice how we pass address of head as `struct node **headRef` in the functions `insertAtFront` and `deleteFromFront`. These two functions change the position of head pointer so we need to pass the address of head pointer and change its value within the function.

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

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

void display(struct node* head)
{
      struct node *temp = head;
      printf("\n\nList elements are - \n");
      while(temp != NULL)
      {
   printf("%d --->",temp->data);
   temp = temp->next;
      }
}

void insertAtMiddle(struct node *head, int position, int value) {
    struct node *temp = head;
    struct node *newNode;
    newNode = malloc(sizeof(struct node));
    newNode->data = value;

    int i;
    for(i=2; inext != NULL) {
        temp = temp->next;
        }
    }
    newNode->next = temp->next;
    temp->next = newNode;
}

void insertAtFront(struct node** headRef, int value) {
    struct node* head = *headRef;

    struct node *newNode;
    newNode = malloc(sizeof(struct node));
    newNode->data = value;
    newNode->next = head;
    head = newNode;

    *headRef = head;
}
void insertAtEnd(struct node* head, int value){
    struct node *newNode;
    newNode = malloc(sizeof(struct node));
    newNode->data = value;
    newNode->next = NULL;
        struct node *temp = head;
    while(temp->next != NULL){
      temp = temp->next;
    }
    temp->next = newNode;
}
void deleteFromFront(struct node** headRef){
    struct node* head = *headRef;
    head = head->next;
    *headRef = head;
}
void deleteFromEnd(struct node* head){
    struct node* temp = head;
    while(temp->next->next!=NULL){
      temp = temp->next;
    }
    temp->next = NULL;
}
```

```
void deleteFromMiddle(struct node* head, int position){
    struct node* temp = head;
    int i;
    for(i=2; inext != NULL) {
    temp = temp->next;
    }
}
temp->next = temp->next->next;
}
int main() {
  /* Initialize nodes */
  struct node *head;
  struct node *one = NULL;
  struct node *two = NULL;
  struct node *three = NULL;
  /* Allocate memory */
  one = malloc(sizeof(struct node));
  two = malloc(sizeof(struct node));
  three = malloc(sizeof(struct node));
  /* Assign data values */
  one->data = 1;
  two->data = 2;
  three->data = 3;
  /* Connect nodes */
  one->next = two;
  two->next = three;
  three->next = NULL;
  /* Save address of first node in head */
  head = one;
  display(head); // 1 --->2 --->3 --->
  insertAtFront(&head, 4);
  display(head); // 4 --->1 --->2 --->3 --->
  deleteFromFront(&head);
  display(head); // 1 --->2 --->3 --->
  insertAtEnd(head, 5);
  display(head); // 1 --->2 --->3 --->5 --->
  deleteFromEnd(head);
  display(head); // 1 --->2 --->3 --->
  int position = 3;
  insertAtMiddle(head, position, 10);
  display(head); // 1 --->2 --->10 --->3 --->
  deleteFromMiddle(head, position);
  display(head); // 1 --->2 --->3 --->
```

```
}
```

The output of the above program is

```
List elements are -
1 --->2 --->3 --->

List elements are -
4 --->1 --->2 --->3 --->

List elements are -
1 --->2 --->3 --->

List elements are -
1 --->2 --->3 --->5 --->

List elements are -
1 --->2 --->3 --->


List elements are -
1 --->2 --->10 --->3 --->
```

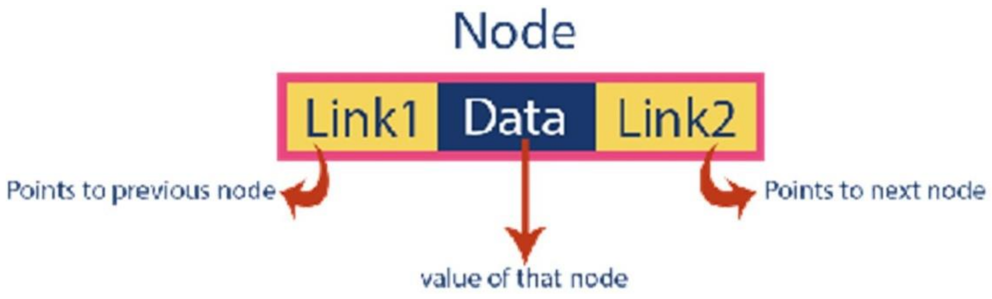
Double Linked List

What is Double Linked List?

In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we can not traverse back. We can solve this kind of problem by using double linked list. Double linked list can be defined as follows...

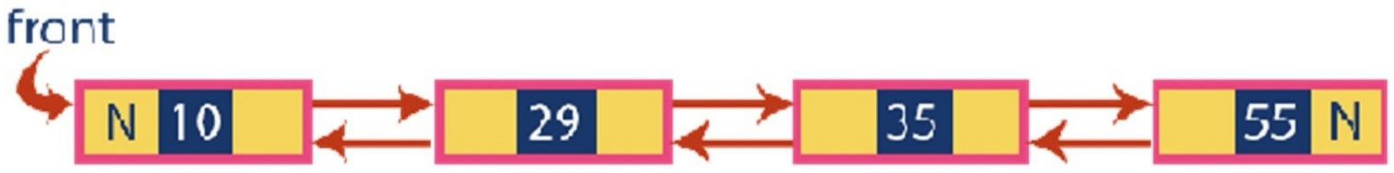
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example



Note:

☀ In double linked list, the first node must be always pointed by **head**.

☀ Always the previous field of the first node must be **NULL**.

☀ Always the next field of the last node must be **NULL**.

Operations

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

- **Step 1:** Create a **newNode** with given value and **newNode → previous** as **NULL**.
- **Step 2:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3:** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
- **Step 4:** If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

- **Step 1:** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3:** If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.

- **Step 4:** If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6:** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3:** If it is **Empty** then, assign **NULL** to **newNode → previous** & **newNode → next** and **newNode** to **head**.
- **Step 4:** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5:** Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6:** Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
- **Step7:** Assign **temp1 → next** to **temp2**,
- **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3:** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4:** Check whether list is having only one node (**temp → previous** is equal to **temp → next**)
- **Step 5:** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6:** If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.

- **Step 3:** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4:** Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)
- **Step 5:** If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6:** If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)
- **Step 7:** Assign **NULL** to **temp → previous → next** and delete **temp**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3:** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4:** Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.
- **Step 5:** If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the fuction.
- **Step 6:** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7:** If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).
- **Step 8:** If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).
- **Step 9:** If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head**of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.
- **Step 10:** If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).
- **Step 11:** If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp)**).
- **Step 12:** If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp**of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous**(**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

Displaying a Double Linked List

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

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- **Step 3:** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4:** Display **'NULL <--- '**.
- **Step 5:** Keep displaying **temp → data** with an arrow (<===>) until **temp** reaches to the last node

- **Step 6:** Finally, display **temp → data** with arrow pointing to **NULL** (**temp → data --->** **NULL**).

Complete Program in C Programming Language

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

void insertAtBeginning(int);
void insertAtEnd(int);
void insertAtAfter(int,int);
void deleteBeginning();
void deleteEnd();
void deleteSpecific(int);
void display();

struct Node
{
  int data;
  struct Node *previous, *next;
}*head = NULL;

void main()
{
  int choice1, choice2, value, location;
  clrscr();
  while(1)
  {
    printf("\n*********** MENU *************\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ");
    scanf("%d",&choice1);
    switch()
    {
      case 1: printf("Enter the value to be inserted: ");
                      scanf("%d",&value);
              while(1)
              {
                       printf("\nSelect from the following Inserting options\n");
                       printf("1. At Beginning\n2. At End\n3. After a Node\n4. Cancel\nEnter your
choice: ");
                scanf("%d",&choice2);
                switch(choice2)
                {
                  case 1:       insertAtBeginning(value);
                                break;
```

```
                    case 2:           insertAtEnd(value);
                                      break;
                    case 3:           printf("Enter the location after which you want to insert: ");
                                      scanf("%d",&location);
                                      insertAfter(value,location);
                                      break;
                    case 4:           goto EndSwitch;
                    default: printf("\nPlease select correct Inserting option!!!\n");
                  }
                }
        case 2: while(1)
                {
                        printf("\nSelect from the following Deleting options\n");
                        printf("1. At Beginning\n2. At End\n3. Specific Node\n4. Cancel\nEnter your
choice: ");
                  scanf("%d",&choice2);
                  switch(choice2)
                  {
                    case 1:           deleteBeginning();
                                      break;
                    case 2:           deleteEnd();
                                      break;
                    case 3:           printf("Enter the Node value to be deleted: ");
                                      scanf("%d",&location);
                                      deleteSpecic(location);
                                      break;
                    case 4:           goto EndSwitch;
                    default: printf("\nPlease select correct Deleting option!!!\n");
                  }
                }
                EndSwitch: break;
        case 3: display();
                break;
        case 4: exit(0);
        default: printf("\nPlease select correct option!!!");
      }
    }
  }

void insertAtBeginning(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
     newNode -> data = value;
     newNode -> previous = NULL;
     if(head == NULL)
     {
        newNode -> next = NULL;
        head = newNode;
     }
     else
     {
        newNode -> next = head;
        head = newNode;
     }
     printf("\nInsertion success!!!");
  }
  void insertAtEnd(int value)
  {
     struct Node *newNode;
     newNode = (struct Node*)malloc(sizeof(struct Node));
     newNode -> data = value;
     newNode -> next = NULL;
     if(head == NULL)
     {
        newNode -> previous = NULL;
        head = newNode;
     }
     else
     {
        struct Node *temp = head;
        while(temp -> next != NULL)
           temp = temp -> next;
        temp -> next = newNode;
        newNode -> previous = temp;
     }
     printf("\nInsertion success!!!");
  }
  void insertAfter(int value, int location)
  {
     struct Node *newNode;
     newNode = (struct Node*)malloc(sizeof(struct Node));
     newNode -> data = value;
     if(head == NULL)
     {
        newNode -> previous = newNode -> next = NULL;
```

```
            head = newNode;
         }
         else
         {
            struct Node *temp1 = head, temp2;
            while(temp1 -> data != location)
            {
               if(temp1 -> next == NULL)
               {
                  printf("Given node is not found in the list!!!");
                  goto EndFunction;
               }
               else
               {
                  temp1 = temp1 -> next;
               }
            }
            temp2 = temp1 -> next;
            temp1 -> next = newNode;
            newNode -> previous = temp1;
            newNode -> next = temp2;
            temp2 -> previous = newNode;
            printf("\nInsertion success!!!");
         }
         }
      void deleteBeginning()
      {
         if(head == NULL)
            printf("List is Empty!!! Deletion not possible!!!");
         else
         {
            struct Node *temp = head;
            if(temp -> previous == temp -> next)
            {
               head = NULL;
               free(temp);
            }
            else{
               head = temp -> next;
               head -> previous = NULL;
               free(temp);
            }
            printf("\nDeletion success!!!");
```

```
   }
}
void deleteEnd()
{
   if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
   else
   {
      struct Node *temp = head;
      if(temp -> previous == temp -> next)
      {
         head = NULL;
         free(temp);
      }
      else{
         while(temp -> next != NULL)
            temp = temp -> next;
         temp -> previous -> next = NULL;
         free(temp);
      }
      printf("\nDeletion success!!!");
   }
}
void deleteSpecific(int delValue)
{
   if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
   else
   {
      struct Node *temp = head;
      while(temp -> data != delValue)
      {
         if(temp -> next == NULL)
         {
            printf("\nGiven node is not found in the list!!!");
            goto FuctionEnd;
         }
         else
         {
            temp = temp -> next;
         }
      }
      if(temp == head)
      {
         head = NULL;
         free(temp);
      }
      else
      {
         temp -> previous -> next = temp -> next;
         free(temp);
      }
      printf("\nDeletion success!!!");
   }
   FuctionEnd:
}
void display()
{
   if(head == NULL)
      printf("\nList is Empty!!!");
   else
   {
      struct Node *temp = head;
      printf("\nList elements are: \n");
      printf("NULL <--- ");
      while(temp -> next != NULL)
      {
         printf("%d <===> ",temp -> data);
      }
      printf("%d ---> NULL", temp -> data);
   }
}
```

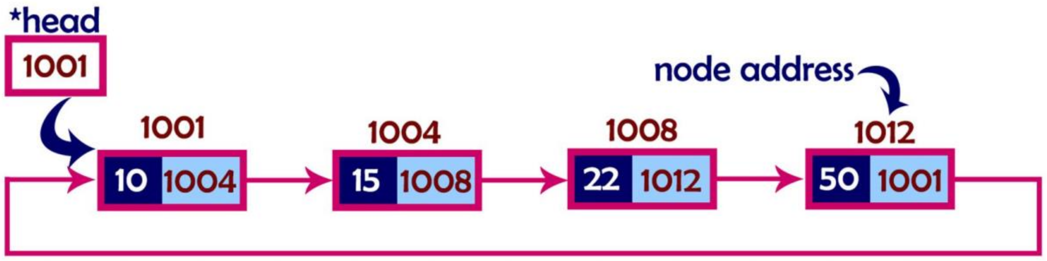
Circular Linked List?

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and the last element has a link to the first element in the sequence.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

Example



Operations
In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- **Step 1:** Include all the **header files** which are used in the program.
- **Step 2:** Declare all the **user defined** functions.
- **Step 3:** Define a **Node** structure with two members **data** and **next**
- **Step 4:** Define a Node pointer '**head**' and set it to **NULL**.
- **Step 4:** Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

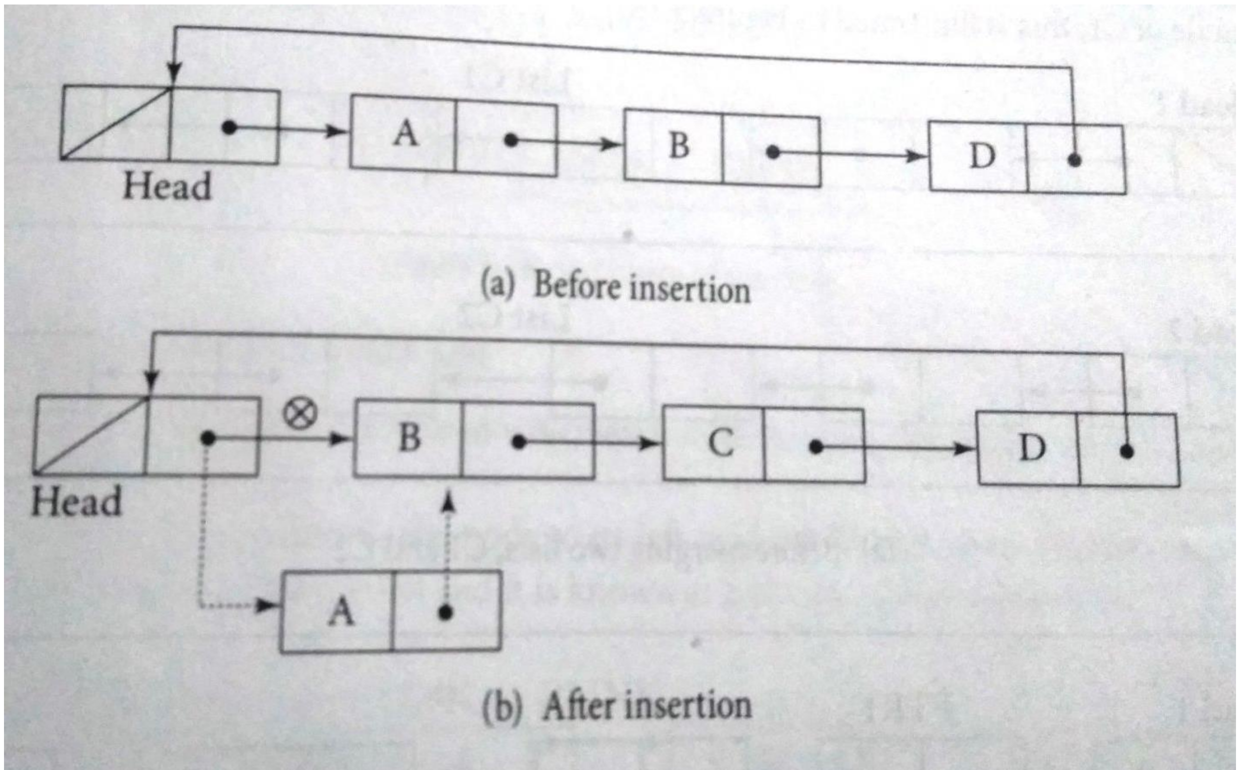


Figure 5.13 Insertion in a circular linked list

In above figure, the node A is inserted between the head and node B. first the value insert at node A and make the new node pointer point to the node B. Now assign the head node pointer with the address of new node i.e head node pointing to the new node.
In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

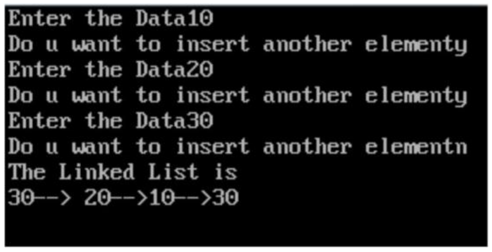
Inserting At Beginning of the list
We can use the following steps to insert a new node at beginning of the circular linked list...

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3:** If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .
- **Step 4:** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
- **Step 5:** Keep moving the '**temp**' to its next node until it reaches to the last node (check the loop 'While(**temp → next** == **head**).
- **Step 6:**If it is true then set temp=temp->next
- **Step 7:** Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.

```
#include<stdio.h>
#include<conio.h>
struct node
{
int data;
struct node *next;
};
struct node *start=NULL;
void insert()
{
struct node *new_node;
new_node=(struct node *)malloc(sizeof(struct node *));
printf(" Enter the Data");
scanf("%d",&new_node->data);
//new_node->next=NULL;
if(start==NULL)
{
start=new_node;
new_node->next=start;
}
else
{
struct node *temp=start;
while(temp->next!=start)
{
```

```
temp=temp->next;
}
new_node->next=start;
start=new_node;
temp->next=start;
}
}
void display()
{
struct node *temp=start;
printf("The Linked List is\n");
while(temp->next!=start)
{
printf("%d-->",temp->data);
temp=temp->next;
}
printf("%d-->",temp->data,next-->data);
}
int main()
{
char ch;
clrscr();
do
{
insert();
printf(" Do u want to insert another element");
ch=getche();
printf("\n");
}
while(ch!='n');
display();
getch();
return 0;
}
```

Output



```
Enter the Data10
Do u want to insert another elementy
Enter the Data20
Do u want to insert another elementy
Enter the Data30
Do u want to insert another elementn
The Linked List is
30--> 20-->10-->30
```

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3:** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** == **head**).
- **Step 6:** Set **temp → next** = **newNode** and **newNode → next** = **head**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3:** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6:** Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7:** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8:** If **temp** is last node then set **temp → next** = **newNode** and **newNode → next** = **head**.
- **Step 8:** If **temp** is not last node then set **newNode → next** = **temp → next** and **temp → next** = **newNode**.

Deletion

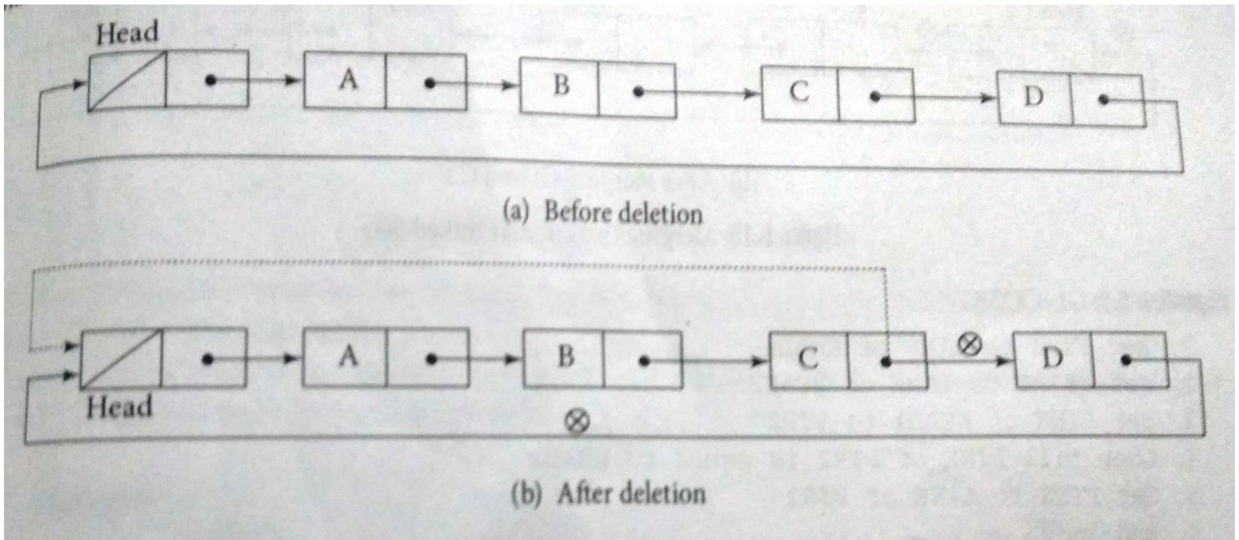


Figure 5.14 Deletion of the node in a circular linked list

In above figure, node D is deleted from the list. So link of node C is pointing to head, which shown by dashed line. Finally return unused node(node D) to the memory bank.

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
- **Step 4:** Check whether list is having only one node (**temp1 → next** == **head**)
- **Step 5:** If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
- **Step 6:** If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** == **head**)
- **Step 7:** Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4:** Check whether list has only one Node (**temp1 → next** == **head**)
- **Step 5:** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6:** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next** == **head**)
- **Step 7:** Set **temp2 → next** = **head** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4:** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

- **Step 5:** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6:** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)
- **Step 7:** If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8:** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9:** If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → nex**t = **head** and delete **temp1**.
- **Step 10:** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- **Step 11:** If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).
- **Step 12:** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Displaying a circular Linked List

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

- **Step 1:** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2:** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4:** while(temp->next!=head)
  Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
- **Step 5:** Finally display **temp → data** with arrow pointing to **head → data**.

Complete Program in C Programming Language

```
#include<stdio.h>
#include<conio.h>
void insertAtBeginning(int);
void insertAtEnd(int);
void insertAtAfter(int,int);
void deleteBeginning();
void deleteEnd();
void deleteSpecific(int);
void display();
struct Node
{
  int data;
  struct Node *next;
}*head = NULL;
void main()
{
  int choice1, choice2, value, location;
```

```
   clrscr();
   while(1)
   {
      printf("\n*********** MENU *************\n");
      printf("1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ");
      scanf("%d",&choice1);
      switch(choice1)
      {
       case 1: printf("Enter the value to be inserted: ");
                         scanf("%d",&value);
               while(1)
               {
                     printf("\nSelect from the following Inserting options\n");
                     printf("1. At Beginning\n2. At End\n3. After a Node\n4. Cancel\nEnter
your choice: ");
                  scanf("%d",&choice2);
                  switch(choice2)
                  {
                     case 1:     insertAtBeginning(value);
                                 break;
                     case 2:     insertAtEnd(value);
                                 break;
                     case 3:     printf("Enter the location after which you want to insert: ");
                                 scanf("%d",&location);
                                 insertAtAfter(value,location);
                                 break;
                     case 4:     goto EndSwitch;
                     default: printf("\nPlease select correct Inserting option!!!\n");
                  }
               }
       case 2: while(1)
               {
                     printf("\nSelect from the following Deleting options\n");
                     printf("1. At Beginning\n2. At End\n3. Specific Node\n4. Cancel\nEnter
your choice: ");
                  scanf("%d",&choice2);
                  switch(choice2)
                  {
                     case 1:     deleteBeginning();
                                 break;
                     case 2:     deleteEnd();
                                 break;
                     case 3:     printf("Enter the Node value to be deleted: ");
```

```
                                        scanf("%d",&location);
                                        deleteSpecific(location);
                                        break;
                              case 4:   goto EndSwitch;
                              default: printf("\nPlease select correct Deleting option!!!\n");
                            }
                          }
                          EndSwitch: break;
              case 3: display();
                          break;
              case 4: exit(0);
              default: printf("\nPlease select correct option!!!");
             }
           }
         }

         void insertAtBeginning(int value)
         {
            struct Node *newNode;
            newNode = (struct Node*)malloc(sizeof(struct Node));
            newNode -> data = value;
            if(head == NULL)
            {
              head = newNode;
              newNode -> next = head;
            }
            else
            {
              struct Node *temp = head;
              while(temp -> next != head)
              temp = temp -> next;
              newNode -> next = head;
              head = newNode;
              temp -> next = head;
            }
            printf("\nInsertion success!!!");
         }
         void insertAtEnd(int value)
         {
           struct Node *newNode;
           newNode = (struct Node*)malloc(sizeof(struct Node));
           newNode -> data = value;
           if(head == NULL)
```

```
    {
      head = newNode;
      newNode -> next = head;
    }
    else
    {
      struct Node *temp = head;
      while(temp -> next != head)
      temp = temp -> next;
      temp -> next = newNode;
      newNode -> next = head;
    }
    printf("\nInsertion success!!!");
  }
  void insertAtAfter(int value, int location)
  {
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode -> data = value;
    if(head == NULL)
    {
      head = newNode;
      newNode -> next = head;
    }
    else
    {
      struct Node *temp = head;
      while(temp -> data != location)
      {
        if(temp -> next == head)
        {
          printf("Given node is not found in the list!!!");
          goto EndFunction;
        }
        else
        {
          temp = temp -> next;
        }
      }
      newNode -> next = temp -> next;
      temp -> next = newNode;
      printf("\nInsertion success!!!");
    }
    EndFunction:
  }
  void deleteBeginning()
  {
    if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
    else
    {
      struct Node *temp = head;
      if(temp -> next == head)
      {
        head = NULL;
        free(temp);
      }
      else{
        head = head -> next;
        free(temp);
      }
      printf("\nDeletion success!!!");
    }
  }
  void deleteEnd()
  {
    if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
    else
    {
      struct Node *temp1 = head, *temp2;
      if(temp1 -> next == head)
      {
       head = NULL;
        free(temp1);
      }
      else{
        while(temp1 -> next != head){
          temp2 = temp1;
            temp1 = temp1 -> next;
        }
        temp2 -> next = head;
        free(temp1);
      }
      printf("\nDeletion success!!!");
    }
```

```
}
void deleteSpecific(int delValue)
{
   if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
   else
   {
      struct Node *temp1 = head, *temp2;
      while(temp1 -> data != delValue)
      {
       if(temp1 -> next == head)
       {
          printf("\nGiven node is not found in the list!!!");
          goto FuctionEnd;
       }
       else
       {
          temp2 = temp1;
          temp1 = temp1 -> next;
       }
      }
      if(temp1 -> next == head){
       head = NULL;
       free(temp1);
      }
      else{
       if(temp1 == head)
         {
            temp2 = head;
            while(temp2 -> next != head)
               temp2 = temp2 -> next;
            head = head -> next;
            temp2 -> next = head;
            free(temp1);
         }
         else
         {
            if(temp1 -> next == head)
            {
               temp2 -> next = head;
            }
            else
            {
```

```
            temp2 -> next = temp1 -> next;
          }
          free(temp1);
        }
      }
      printf("\nDeletion success!!!");
    }
    FuctionEnd:
  }
  void display()
  {
    if(head == NULL)
      printf("\nList is Empty!!!");
    else
    {
      struct Node *temp = head;
      printf("\nList elements are: \n");
      while(temp -> next != head)
      {
        printf("%d ---> ",temp -> data);
        temp=temp->next;
      }
      printf("%d ---> %d", temp -> data, head -> data);
    }
  }
```

Applications of linked list

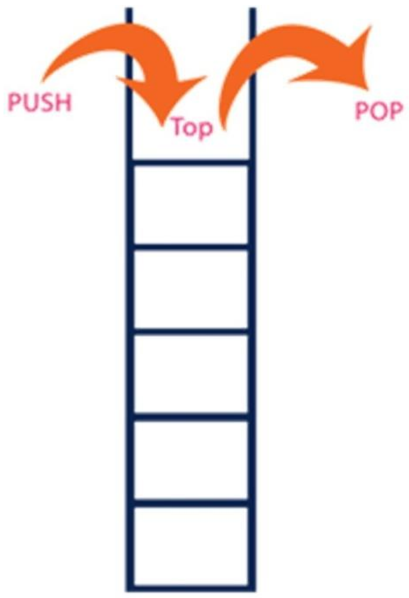
1. Implementation of stacks and queues
2. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
3. Dynamic memory allocation : We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. representing sparse matrices

UNIT-2

What is a Stack?

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at single position which is known as "**top**". That means, new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.

Def:- **"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".**



The following are the operations of stack

1.Push

2. Pop

3.Display

4.Isempty

5.Isfull

6.Peek

1.Push:-

In a stack, the insertion operation is performed using a function called **"push"**.

2.Pop:-

In a stack, the deletion operation is performed using a function called **"pop"**.

In the figure, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

3. Display:-

By using display operation, to display the elements of stack

4. Isempty:-

It is used to check the stack is empty or not. If it is empty it display the stack is under flow i.e deletion is not possible

5. Isfull:-

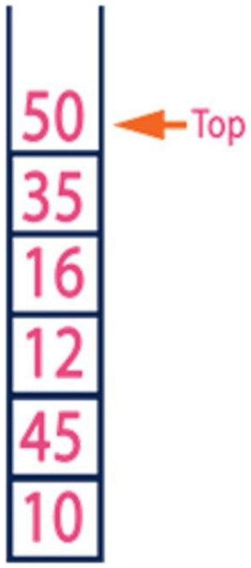
It is used to check the stack is full or not. If stack is full, it display overflow. i.e insertion is not possible

6. Peek:-

It is used to get the value of the top element without removing it.

Example

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. Top is at 50 as shown in the image below...



Stack data structure can be implement in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

Stack Using Array:-

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable **'top'**. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array

A stack can be implemented using array as follows...

Before implementing actual operations, first follow the below steps to create an empty stack.

- **Step 1:** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2:** Declare all the **functions** used in stack implementation.
- **Step 3:** Create a one dimensional array with fixed size (**int stack[SIZE]**)
- **Step 4:** Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)
- **Step 5:** In main method display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

push(value) - Inserting value into the stack:-

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

- **Step 1:** Check whether **stack** is **FULL**. (**top == SIZE-1**)
- **Step 2:** If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

pop() - Delete a value from the Stack:-

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

- **Step 1:** Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2:** If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).

display() - Displays the elements of a Stack:-

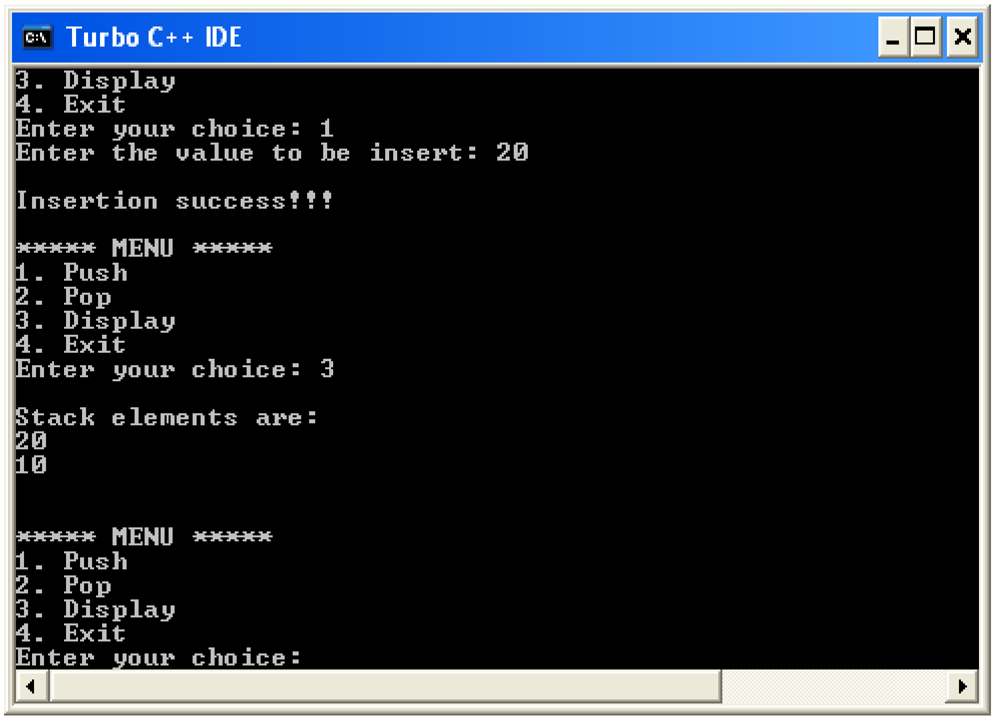
We can use the following steps to display the elements of a stack...

- **Step 1:** Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2:** If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
- **Step 4:** Repeat above step until **i** value becomes '0'.

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
void push(int);
void pop();
void display();
int stack[SIZE], top = -1;
void main()
{
  int value, choice;
  clrscr();
  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--;
    }
}
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]);
    }
}
```

OUTPUT:

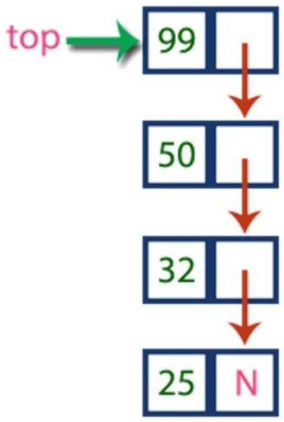


```
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 20

Insertion success!!!

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

Stack elements are:
20
10


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

Stack using Linked List
The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its next node in the list. The **next** field of the first element must be always **NULL**.

Example



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

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

- **Step 1:** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2:** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3:** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4:** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack
We can use the following steps to insert a new node into the stack...

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3:** If it is **Empty**, then set **newNode** → **next** = **NULL**.
- **Step 4:** If it is **Not Empty**, then set **newNode** → **next** = **top**.
- **Step 5:** Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack
We can use the following steps to delete a node from the stack...

- **Step 1:** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2:** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3:** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4:** Then set '**top** = **top** → **next**'.
- **Step 7:** Finally, delete '**temp**' (**free(temp)**).

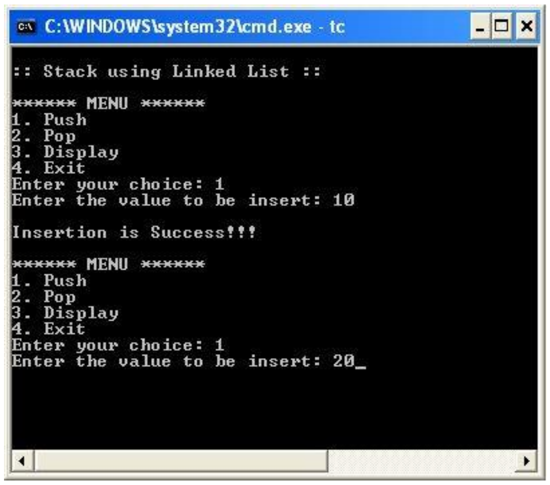
display() - Displaying stack of elements
We can use the following steps to display the elements (nodes) of a stack...

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

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

struct Node
```

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

void push(int);
void pop();
void display();

void main()
{
   int choice, value;
   clrscr();
   printf("\n:: Stack using Linked List ::\n");
   while(1){
      printf("\n****** MENU ******\n");
      printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
      printf("Enter 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!!! Please try again!!!\n");
      }
   }
}
void push(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(top == NULL)
      newNode->next = NULL;
   else
      newNode->next = top;
   top = newNode;
   printf("\nInsertion is Success!!!\n");
}
void pop()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      printf("\nDeleted element: %d", temp->data);
      top = temp->next;
      free(temp);
   }
}
void display()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
      }
      printf("%d--->NULL",temp->data);
   }
}
```

Out Put:



```
:: Stack using Linked List ::

****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 10

Insertion is Success!!!

****** MENU ******
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 20_
```

<u>**Stack Expression**</u>

In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression can be defined as follows...

An expression is a collection of operators and operands that represents a specific value.

In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

1. **Infix Expression**

2. **Postfix Expression**
3. **Prefix Expression**

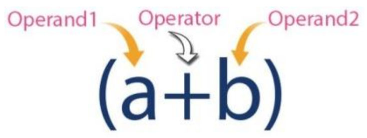
Infix Expression

In infix expression, operator is used in between operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Example



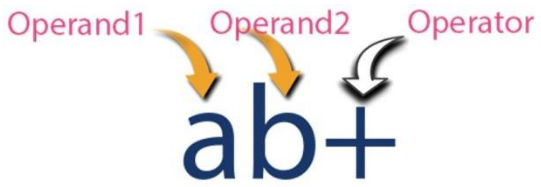
Postfix Expression

In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**".

The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

Example



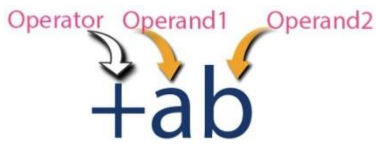
Prefix Expression

In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**".

The general structure of Prefix expression is as follows...

Operator Operand1 Operand2

Example



Any expression can be represented using the above three different types of expressions. And we can convert an expression from one form to another form like **Infix to Postfix**, **Infix to Prefix**, **Prefix to Postfix** and vice versa.

Expression Conversion

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

1. Find all the operators in the given Infix Expression.
2. Find the order of operators evaluated according to their Operator precedence.
3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Example

Consider the following Infix Expression to be converted into Postfix Expression...

D = A + B * C

- **Step 1:** The Operators in the given Infix Expression : **= , + , ***
- **Step 2:** The Order of Operators according to their preference : *** , + , =**
- **Step 3:** Now, convert the first operator ***------ D = A + B C ***
- **Step 4:** Convert the next operator **+------D = A BC* +**
- **Step 5:** Convert the next operator **=------D ABC*+ =**

Finally, given Infix Expression is converted into Postfix Expression as follows...

D A B C * + =

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

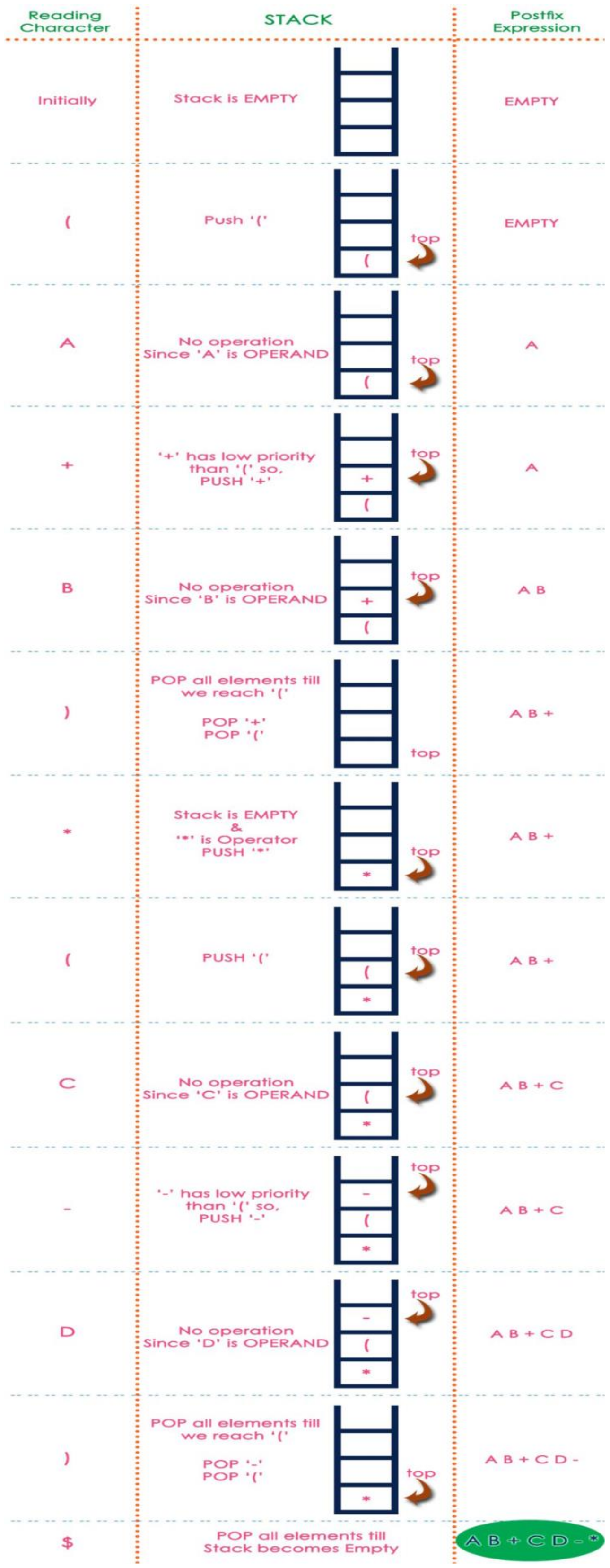
1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
5. If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Example

Consider the following Infix Expression...

(A + B) * (C - D)

The given infix expression can be converted into postfix expression using Stack data Structure as follows...



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 +
*	Stack is EMPTY & '*' is Operator PUSH '*'	A B +
(	PUSH '('	A B +
C	No operation Since 'C' is OPERAND	A B + C
-	'-' has low priority than '(' so, PUSH '-'	A B + C
D	No operation Since 'D' is OPERAND	A B + C D
)	POP all elements till we reach '(' POP '-' POP '('	A B + C D -
$	POP all elements till Stack becomes Empty	A B + C D - *

h

The final Postfix Expression is as follows...A B + C D - *

Examples

1)a+(b*c)/d   abc*d/+   2346

2)(a+b)^c-(d*e)/f	ab+c^de*f/-
3)(a+b)/(c+d)-(d*e)	ab+cd+/de*-

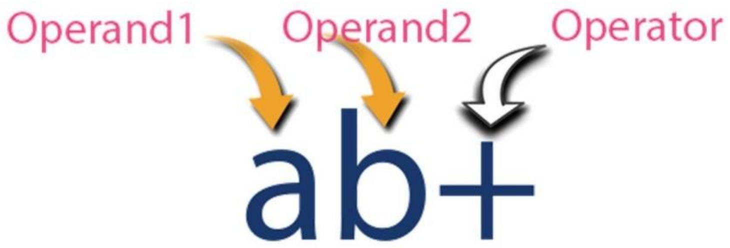
Postfix Expression Evaluation

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

ePostfix Expression has following general structure...

Operand1 Operand2 Operator

Example



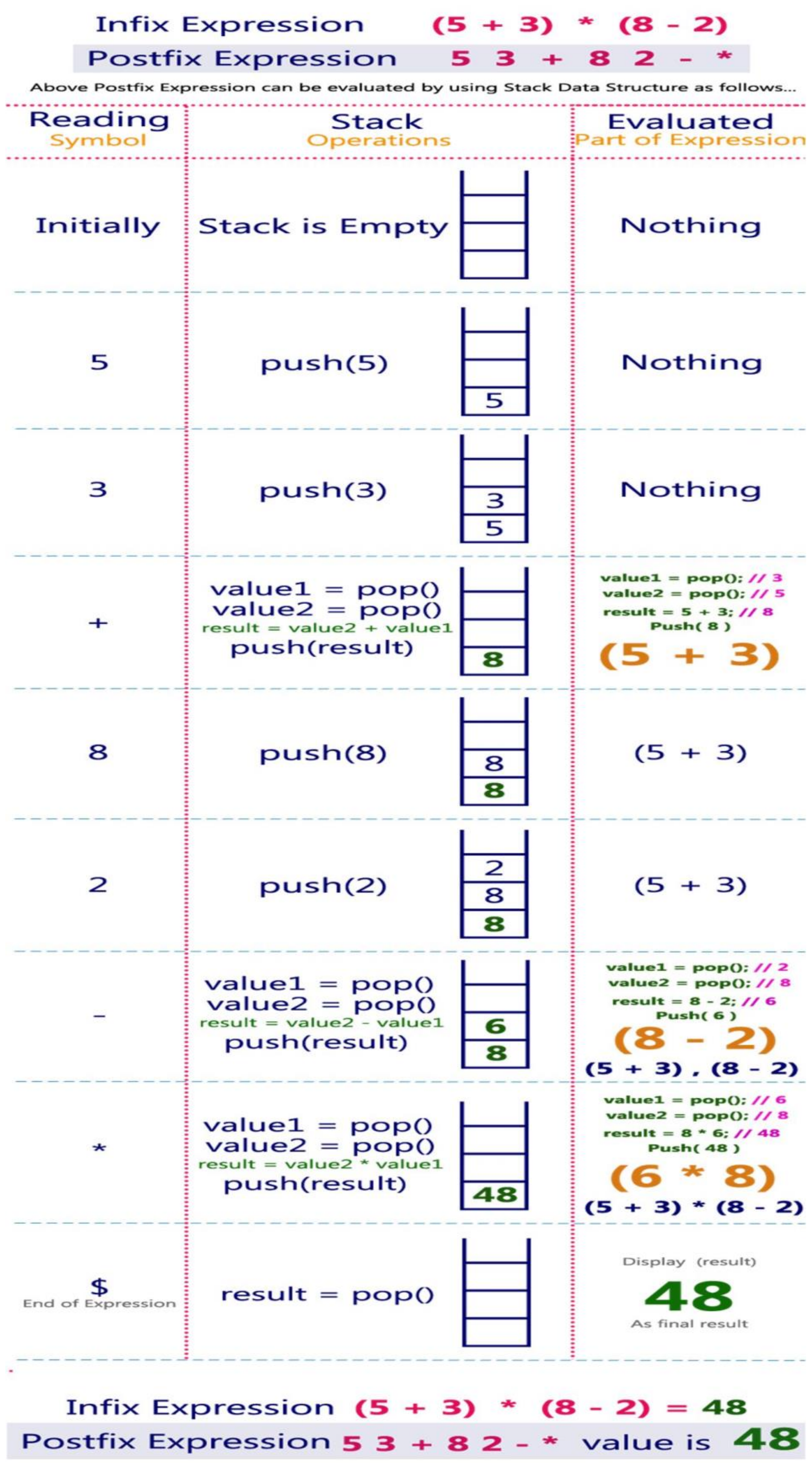
Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Example

Consider the following Expression...



Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Stack	Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)	5	Nothing
3	push(3)	3, 5	Nothing
+	value1 = pop() value2 = pop() result = value2 + value1 push(result)	8	value1 = pop(); // 3 value2 = pop(); // 5 result = 5 + 3; // 8 Push(8) (5 + 3)
8	push(8)	8, 8	(5 + 3)
2	push(2)	2, 8, 8	(5 + 3)
-	value1 = pop() value2 = pop() result = value2 - value1 push(result)	6, 8	value1 = pop(); // 2 value2 = pop(); // 8 result = 8 - 2; // 6 Push(6) (8 - 2) (5 + 3) , (8 - 2)
*	value1 = pop() value2 = pop() result = value2 * value1 push(result)	48	value1 = pop(); // 6 value2 = pop(); // 8 result = 8 * 6; // 48 Push(48) (6 * 8) (5 + 3) * (8 - 2)
$ End of Expression	result = pop()		Display (result) 48 As final result

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

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

<u>Applications of Stack</u>

Expression Evaluation

Stack is used to evaluate prefix, postfix and infix expressions.

Expression Conversion

An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

Syntax Parsing

Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Backtracking

Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

Parenthesis Checking

Stack is used to check the proper opening and closing of parenthesis.

String Reversal

Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

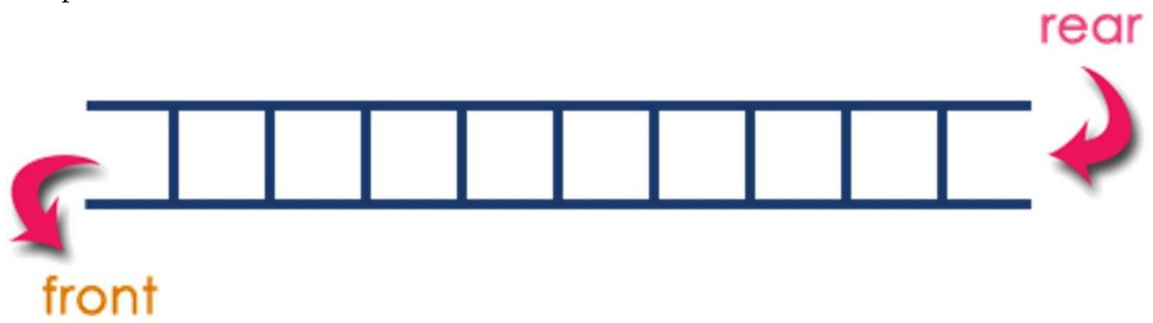
Function Call

Stack is used to keep information about the active functions or subroutines.

Queue

What is a Queue?

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at other end. In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'. In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.



In a queue data structure, the insertion operation is performed using a function called "**enQueue()**" and deletion operation is performed using a function called "**deQueue()**".
Queue data structure can be defined as follows...

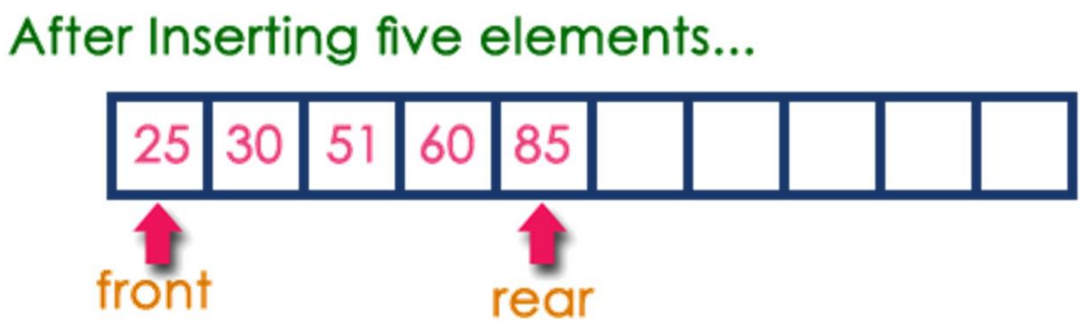
Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.

A queue can also be defined as

"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle".

Example

Queue after inserting 25, 30, 51, 60 and 85.



Operations on a Queue

The following operations are performed on a queue data structure...

1. **enQueue(value) - To insert an element into the queue**
2. **deQueue() - To delete an element from the queue**
3. **display() - (To display the elements of the queue)**

Queue data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number of elements.

Queue Using Array

A queue data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear**'. Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at '**front**' position as deleted element.

Queue Operations using Array

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to create an empty queue.

- **Step 1:** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2:** Declare all the **user defined functions** which are used in queue implementation.
- **Step 3:** Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- **Step 4:** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
- **Step 5:** Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- **Step 1:** Check whether **queue** is **FULL**. (**rear == SIZE-1**)

- **Step 2:** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear]** = **value**.

deQueue() - Deleting a value from the Queue

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

- **Step 1:** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element.

display() - Displays the elements of a Queue

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

- **Step 1:** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
- **Step 3:** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value is equal to **rear** (**i** <= **rear**)

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
void enQueue(int);
void deQueue();
void display();
int queue[SIZE], front = -1, rear = -1;
void main()
{
  int value, choice;
  clrscr();
  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();
```

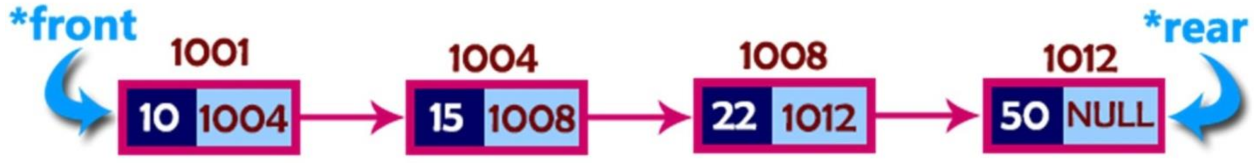
```
                    break;
            case 4: exit(0);
            default: printf("\nWrong selection!!! Try again!!!");
      }
   }
}
void enQueue(int value){
   if(rear == SIZE-1)
      printf("\nQueue is Full!!! Insertion is not possible!!!");
   else{
      if(front == -1)
            front = 0;
      rear++;
      queue[rear] = value;
      printf("\nInsertion success!!!");
   }
}
void deQueue(){
   if(front == rear)
      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]);
   }
}
```

Queue using Linked List

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

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

Example



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

Operations

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

- **Step 1:** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2:** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3:** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4:** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

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

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

deQueue() - Deleting an Element from Queue

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

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

display() - Displaying the elements of Queue

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

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

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

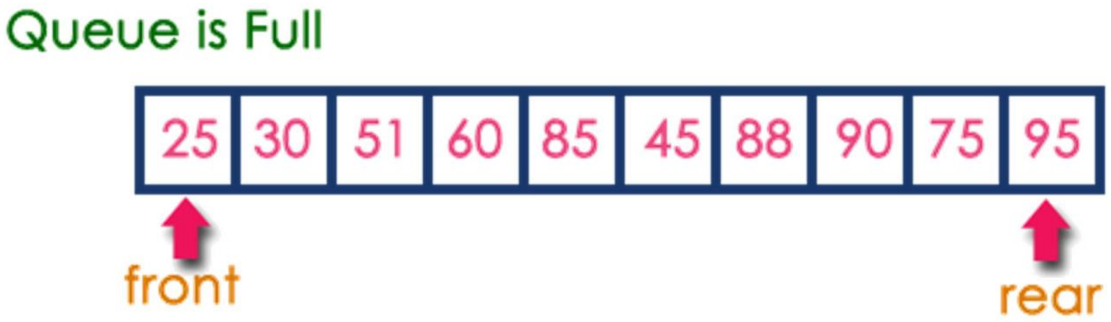
```
struct Node
{
   int data;
   struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
void delete();
void display();

void main()
{
   int choice, value;
   clrscr();
   printf("\n:: Queue Implementation using Linked List ::\n");
   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("Enter the value to be insert: ");
                                scanf("%d", &value);
                                insert(value);
                                break;
                 case 2: delete(); break;
                 case 3: display(); break;
                 case 4: exit(0);
                 default: printf("\nWrong selection!!! Please try again!!!\n");
      }
   }
}
void insert(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode -> next = NULL;
   if(front == NULL)
      front = rear = newNode;
   else{
      rear -> next = newNode;
      rear = newNode;
   }
   printf("\nInsertion is Success!!!\n");
}
void delete()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
```

```
  else{
    struct Node *temp = front;
    front = front -> next;
    printf("\nDeleted element: %d\n", temp->data);
    free(temp);
  }
}
void display()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    struct Node *temp = front;
    while(temp->next != NULL){
           printf("%d--->",temp->data);
           temp = temp -> next;
    }
    printf("%d--->NULL\n",temp->data);
  }
}
```

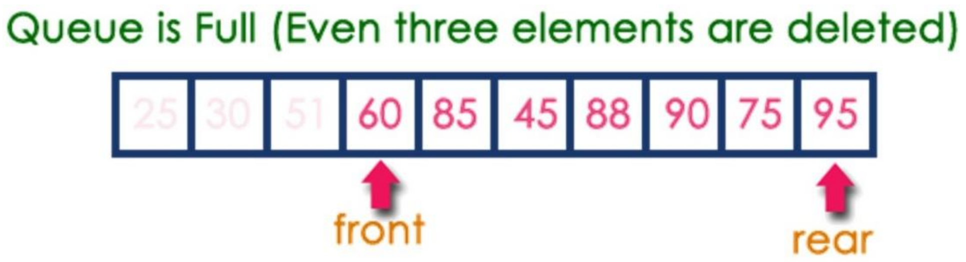
Circular Queue

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we can not insert the next element until all the elements are deleted from the queue. For example consider the queue below...

After inserting all the elements into the queue.



Now consider the following situation after deleting three elements from the queue...



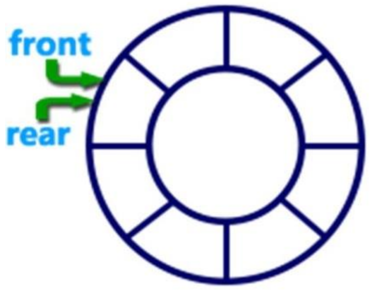
This situation also says that Queue is Full and we can not insert the new element because, '**rear**' is still at last position. In above situation, even though we have empty positions in the queue we can not make use of them to insert new element. This is the major problem in normal queue data structure. To overcome this problem we use circular queue data structure.

What is Circular Queue?

A Circular Queue can be defined as follows...

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.

Graphical representation of a circular queue is as follows...



Implementation of Circular Queue

To implement a circular queue data structure using array, we first perform the following steps before we implement actual operations.

- **Step 1:** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2:** Declare all **user defined functions** used in circular queue implementation.
- **Step 3:** Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)
- **Step 4:** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
- **Step 5:** Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at **rear**position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

- **Step 1:** Check whether **queue** is **FULL**. (**(rear == SIZE-1 && front == 0) || (front == rear+1)**)
- **Step 2:** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.
- **Step 4:** Increment **rear** value by one (**rear++**), set **queue[rear]** = **value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

deQueue() - Deleting a value from the Circular Queue

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

- **Step 1:** Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then check whether both **front - 1** and **rear** are equal (**front -1** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

display() - Displays the elements of a Circular Queue

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

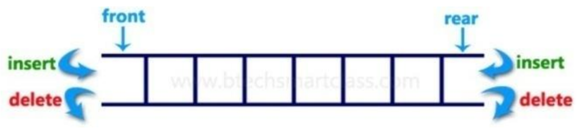
- **Step 1:** Check whether **queue** is **EMPTY**. (**front == -1**)
- **Step 2:** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4:** Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.
- **Step 5:** If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.
- **Step 6:** Set **i** to **0**.
- **Step 7:** Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
void enQueue(int);
void deQueue();
void display();
int cQueue[SIZE], front = -1, rear = -1;
void main()
{
  int choice, value;
  clrscr();
  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)
            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++]);
      }
    }
  }
```

Double Ended Queue (Dequeue)

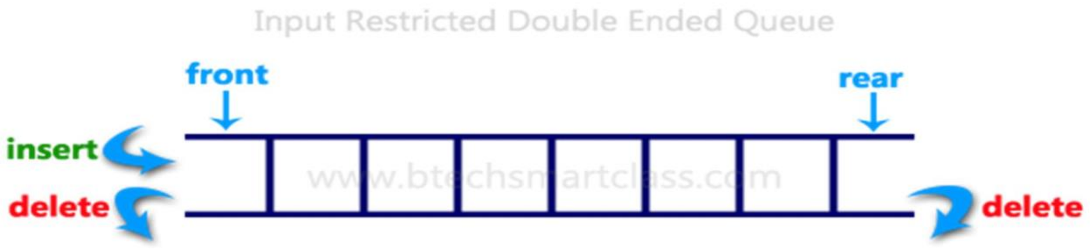
Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



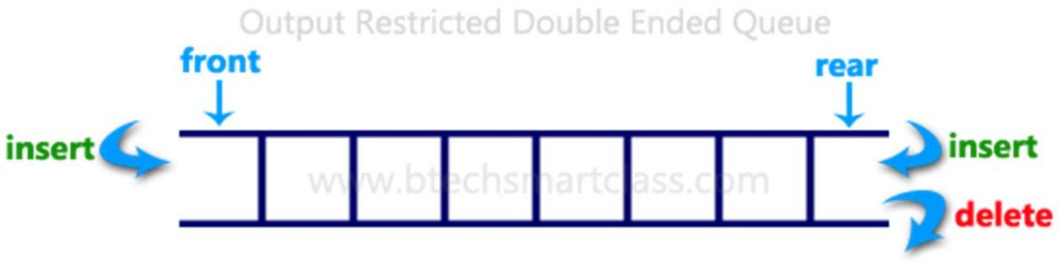
Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

Input Restricted Double Ended Queue
In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue
In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



```
#include<stdio.h>
#include<conio.h>
#define SIZE 100
void enQueue(int);
int deQueueFront();
int deQueueRear();
void  enQueueRear(int);
void enQueueFront(int);
void display();
int queue[SIZE];
int rear = 0, front = 0;
int main()
{
```

```
char ch;
int choice1, choice2, value;
printf("\n******* Type of Double Ended Queue *******\n");
 do
 {
     printf("\n1.Input-restricted deque \n");
     printf("2.output-restricted deque \n");
     printf("\nEnter your choice of Queue Type : ");
     scanf("%d",&choice1);
     switch(choice1)
     {
         case 1:
             printf("\nSelect the Operation\n");
             printf("1.Insert\n2.Delete from Rear\n3.Delete from Front\n4. Display");
             do
             {
               printf("\nEnter your choice for the operation in c deque: ");
               scanf("%d",&choice2);
               switch(choice2)
               {
                 case 1: enQueueRear(value);
                       display();
                                 break;
                      case 2: value = deQueueRear();
                                 printf("\nThe value deleted is %d",value);
                       display();
                                 break;
                 case 3: value=deQueueFront();
                            printf("\nThe value deleted is %d",value);
                       display();
                            break;
                 case 4: display();
                       break;
                 default:printf("Wrong choice");
               }
               printf("\nDo you want to perform another operation (Y/N): ");
               ch=getch();
             }while(ch=='y'||ch=='Y');
             getch();
             break;
         case 2 :
            printf("\n---- Select the Operation ----\n");
            printf("1. Insert at Rear\n2. Insert at Front\n3. Delete\n4. Display");
            do
            {
              printf("\nEnter your choice for the operation: ");
              scanf("%d",&choice2);
              switch(choice2)
              {
                case 1: enQueueRear(value);
```

```
                              display();
                              break;
                    case 2: enQueueFront(value);
                              display();
                              break;
                    case 3: value = deQueueFront();
                              printf("\nThe value deleted is %d",value);
                              display();
                              break;
                    case 4: display();
                              break;
                    default:printf("Wrong choice");
                  }
                  printf("\nDo you want to perform another operation (Y/N): ");
                  ch=getch();
                } while(ch=='y'||ch=='Y');
                getch();
                break ;
        }
        printf("\nDo you want to continue(y/n):");
        ch=getch();
     }while(ch=='y'||ch=='Y');
}
void enQueueRear(int value)
{
    char ch;
    if(front == SIZE/2)
     {
        printf("\nQueue is full!!! Insertion is not possible!!! ");
        return;
     }
     do
     {
        printf("\nEnter the value to be inserted:");
        scanf("%d",&value);
        queue[front] = value;
        front++;
        printf("Do you want to continue insertion Y/N");
        ch=getch();
     }while(ch=='y');
}

void enQueueFront(int value)
{
    char ch;
    if(front==SIZE/2)
     {
        printf("\nQueue is full!!! Insertion is not possible!!!");
        return;
     }
```

```
        do
        {
            printf("\nEnter the value to be inserted:");
            scanf("%d",&value);
            rear--;
            queue[rear] = value;
            printf("Do you want to continue insertion Y/N");
            ch = getch();
        }
        while(ch == 'y');
}
int deQueueRear()
{
    int deleted;
    if(front == rear)
    {
            printf("\nQueue is Empty!!! Deletion is not possible!!!");
            return 0;
    }
    front--;
    deleted = queue[front+1];
    return deleted;
}
int deQueueFront()
{
    int deleted;
    if(front == rear)
    {
            printf("\nQueue is Empty!!! Deletion is not possible!!!");
            return 0;
    }
    rear++;
    deleted = queue[rear-1];
    return deleted;
}
void display()
{
    int i;
    if(front == rear)
      printf("\nQueue is Empty!!! Deletion is not possible!!!")
    else{
      printf("\nThe Queue elements are:");
      for(i=rear; i < front; i++)
      {
        printf("%d\t ",queue[i]);
      }
    }
}
```

Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

UNIT-3

Tree Terminology

In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order. Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows...

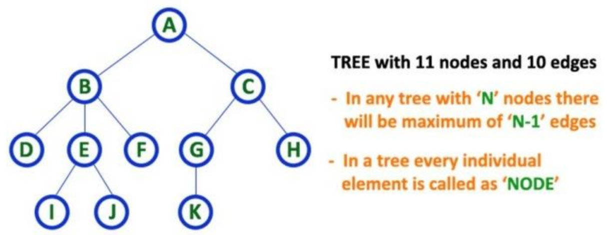
Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

In tree data structure, every individual element is called as **Node**. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure. In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

Example

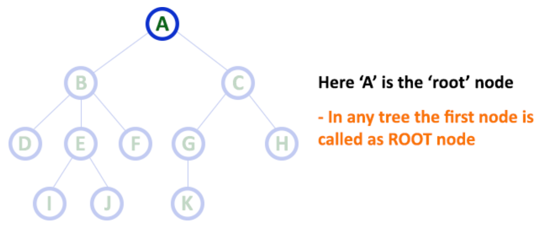

A
B
C
D
E
F
G
H
I
J
K
TREE with 11 nodes and 10 edges
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Terminology

In a tree data structure, we use the following terminology...

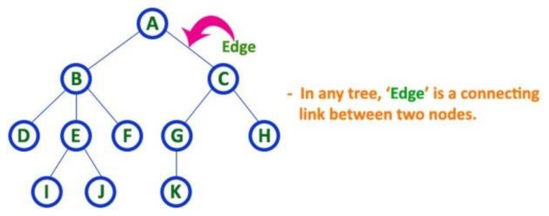
1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.


A
B
C
D
E
F
G
H
I
J
K
Here 'A' is the 'root' node
- In any tree the first node is called as ROOT node

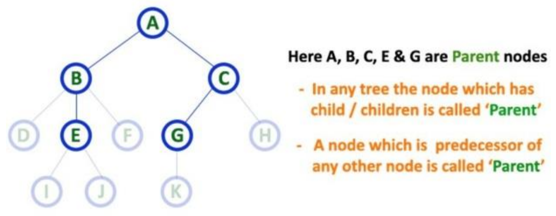
2. Edge

In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.


A
Edge
B
C
D
E
F
G
H
I
J
K
- In any tree, 'Edge' is a connecting link between two nodes.

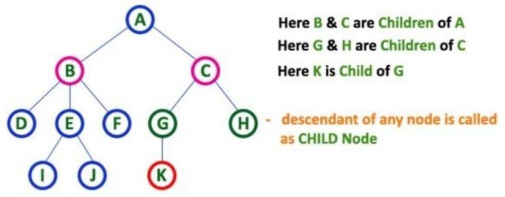
3. Parent

In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "**The node which has child / children**".


A
B
C
D
E
F
G
H
I
J
K
Here A, B, C, E & G are Parent nodes
- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

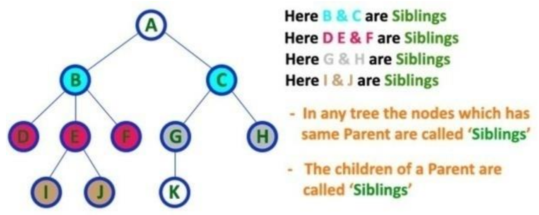
4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.


A
B
C
D
E
F
G
H
I
J
K
Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G
- descendant of any node is called as CHILD Node

5. Siblings

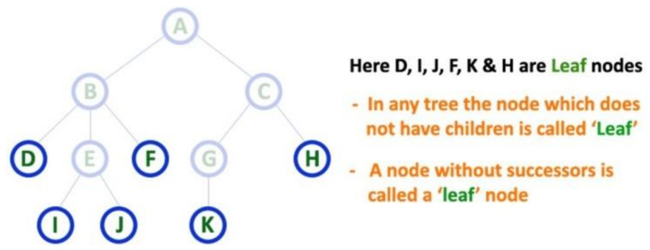
In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.


A
B
C
D
E
F
G
H
I
J
K
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings
- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

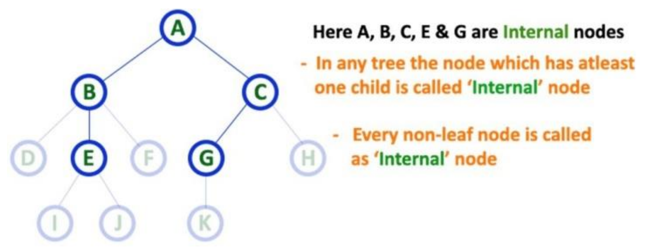
In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, <u>leaf node is also called as '**Terminal**' node.</u>


A
B
C
D
E
F
G
H
I
J
K
Here D, I, J, F, K & H are Leaf nodes
- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes

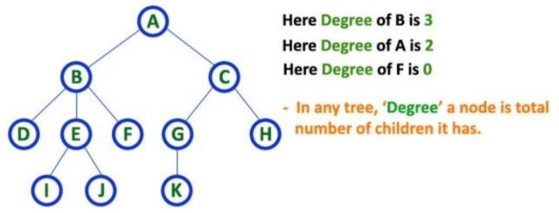
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.
In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**.

<u>**The root node is also said to be Internal Node**</u> if the tree has more than one node. <u>Internal nodes are also called as '**Non-Terminal**' nodes.</u>


Here A, B, C, E & G are Internal nodes
- In any tree the node which has atleast one child is called 'Internal' node
- Every non-leaf node is called as 'Internal' node

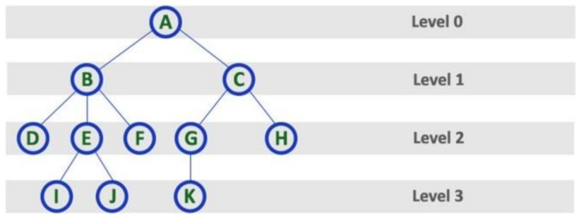
8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'


Here Degree of B is 3
Here Degree of A is 2
Here Degree of F is 0
- In any tree, 'Degree' a node is total number of children it has.

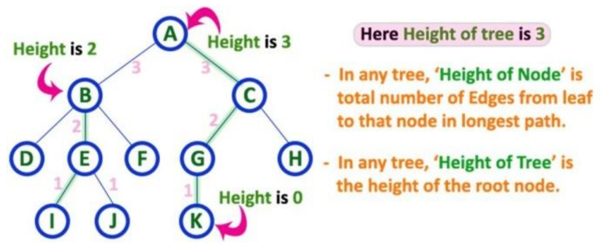
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



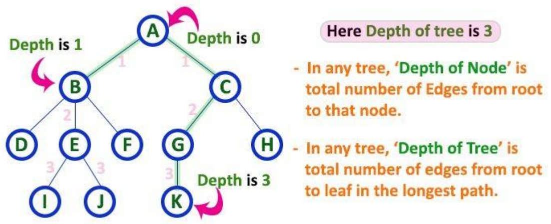
10. Height

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. <u>In a tree, height of the root node is said to be **height of the tree**</u>. In a tree, **<u>height of all leaf nodes is '0'.</u>**

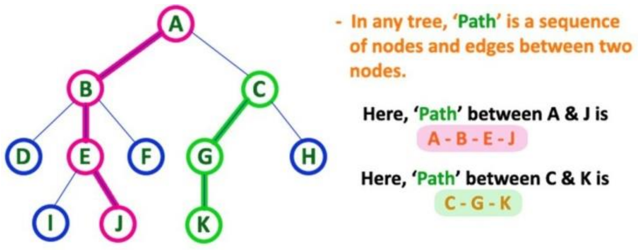

Height is 2
Height is 3
Here Height of tree is 3
- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.
Height is 0

11. Depth

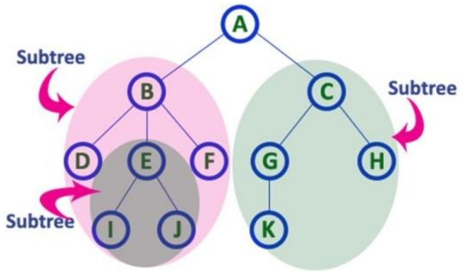
In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. <u>In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**</u>. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **<u>depth of the root node is '0'.</u>**


Depth is 1
Depth is 0
Depth is 3
Here Depth of tree is 3
- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path
In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.


- In any tree, 'Path' is a sequence of nodes and edges between two nodes.
Here, 'Path' between A & J is
A - B - E - J
Here, 'Path' between C & K is
C - G - K

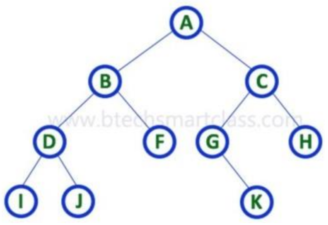
13. Sub Tree
In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



<u>Binary Tree and types</u>
In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.
Example



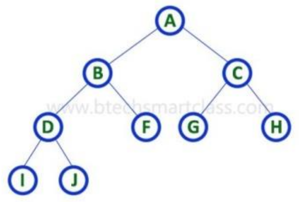
There are different types of binary trees and they are...

1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

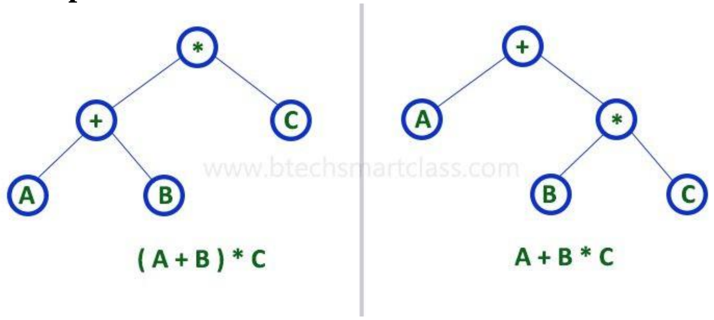
A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

Strictly binary tree is also called as **Full Binary Tree** or **Proper Binary Tree** or **2-Tree**



Strictly binary tree data structure is used to represent mathematical expressions.

Example

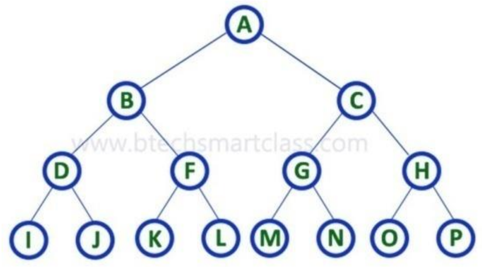


2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be 2^2 = 4 nodes and at level 3 there must be 2^3 = 8 nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

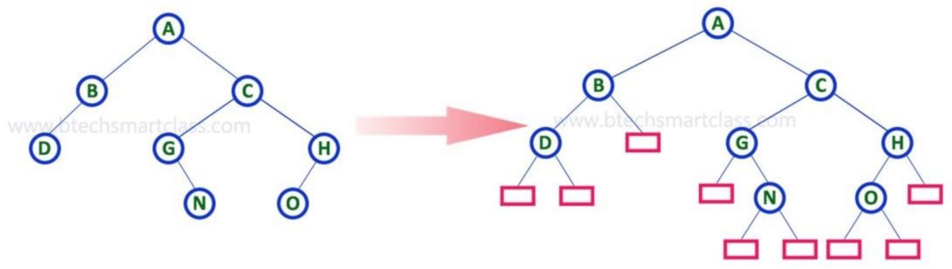
Complete binary tree is also called as **Perfect Binary Tree**



3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

3. Skewed Binary Tree

If a tree which is dominated by left child node or right child node, is said to be a Skewed Binary Tree.

In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child**.**

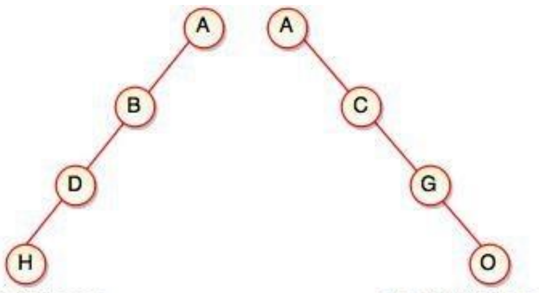


Fig. Left Skewed Binary Tree

Fig. Right Skewed Binary Tree

In a left skewed tree, most of the nodes have the left child without corresponding right child.

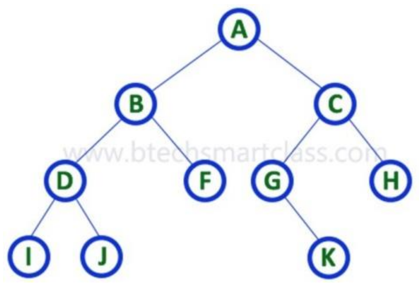
In a right skewed tree, most of the nodes have the right child without corresponding left child.

<u>**Binary Tree Representations**</u>

A binary tree data structure is represented using two methods. Those methods are as follows...

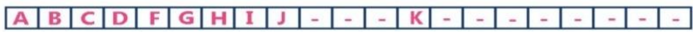
1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...



1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.
Consider the above example of binary tree and it is represented as follows...

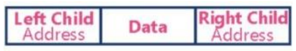


A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of $\mathbf{2^{n+1} - 1}$.

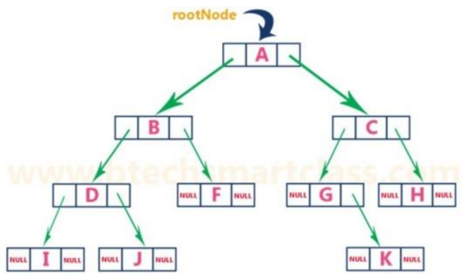
2. Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.
In this linked list representation, a node has the following structure...



Left Child Address	Data	Right Child Address

The above example of binary tree represented using Linked list representation is shown as follows...



__**Binary Tree Traversals**__

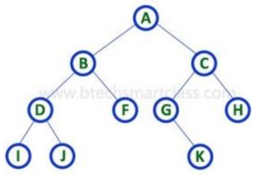
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

Consider the following binary tree...



1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree.
This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit **'I'** then go for its root node **'D'** and later we visit D's right child **'J'**. With this we have completed the left part of node B. Then visit **'B'** and next B's right child **'F'** is visited. With this we have completed left part of node A. Then visit root node **'A'**. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit **'G'** and then visit G's right child K. With this we have completed the left part of node C. Then visit root node **'C'** and next visit C's right child **'H'** which is the right most child in the tree so we stop the process.
That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.
In the above example of binary tree, first we visit root node **'A'** then visit its left child **'B'** which is a root for D and F. So we visit B's left child **'D'** and again D is a root for I and J. So we visit D's left child **'I'** which is the left most child. So next we go for visiting D's right child **'J'**. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child **'F'**. With this we have completed root and left parts of node A. So we go for A's right child **'C'** which is a root node for G and H. After visiting C, we go for its left child **'G'** which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child **'K'**. With this we have completed node C's root and left parts. Next visit C's right child **'H'** which is the right most child in the tree. So we stop the process.
That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

2. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.
Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A** using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

<u>Binary Search Tree</u>

In a binary tree, every node can have maximum of two children but there is no order of nodes based on their values. In binary tree, the elements are arranged as they arrive to the tree, from top to bottom and left to right.
A binary tree has the following time complexities...

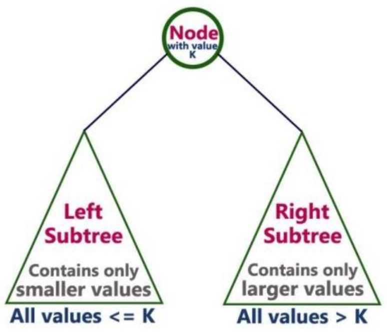
1. **Search Operation - O(n)**
2. **Insertion Operation - O(1)**

3. **Deletion Operation - O(n)**

To enhance the performance of binary tree, we use special type of binary tree known as **Binary Search Tree**. Binary search tree mainly focus on the search operation in binary tree. Binary search tree can be defined as follows...

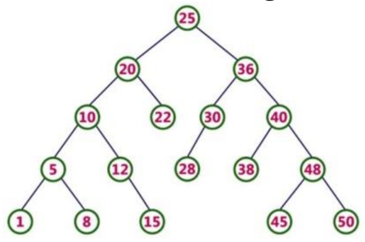
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in left subtree of any node contains smaller values and all the nodes in right subtree of that contains larger values as shown in following figure...



Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Every Binary Search Tree is a binary tree but all the Binary Trees need not to be binary search trees.

Operations on a Binary Search Tree

The following operations are performed on a binary earch tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in BST

In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
- **Step 5:** If search element is smaller, then continue the search process in left subtree.

- **Step 6:** If search element is larger, then continue the search process in right subtree.
- **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1:** Create a newNode with given value and set its **left** and **right** to **NULL**.
- **Step 2:** Check whether tree is Empty.
- **Step 3:** If the tree is **Empty**, then set set **root** to **newNode**.
- **Step 4:** If the tree is **Not Empty**, then check whether value of newNode is **smaller** or **larger** than the node (here it is root node).
- **Step 5:** If newNode is **smaller** than **or equal** to the node, then move to its **left** child. If newNode is **larger** than the node, then move to its **right** child.
- **Step 6:** Repeat the above step until we reach to a **leaf** node (e.i., reach to NULL).
- **Step 7:** After reaching a leaf node, then isert the newNode as **left child** if newNode is **smaller or equal** to that leaf else insert it as **right child**.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree has follwing three cases...

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Case 1: **Deleting a leaf node**

We use the following steps to delete a leaf node from BST...

- **Step 1: Find** the node to be deleted using **search operation**
- **Step 2:** Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: **Deleting a node with one child**

We use the following steps to delete a node with one child from BST...

- **Step 1: Find** the node to be deleted using **search operation**
- **Step 2:** If it has only one child, then create a link between its parent and child nodes.
- **Step 3:** Delete the node using **free** function and terminate the function.

Case 3: **Deleting a node with two children**

We use the following steps to delete a node with two children from BST...

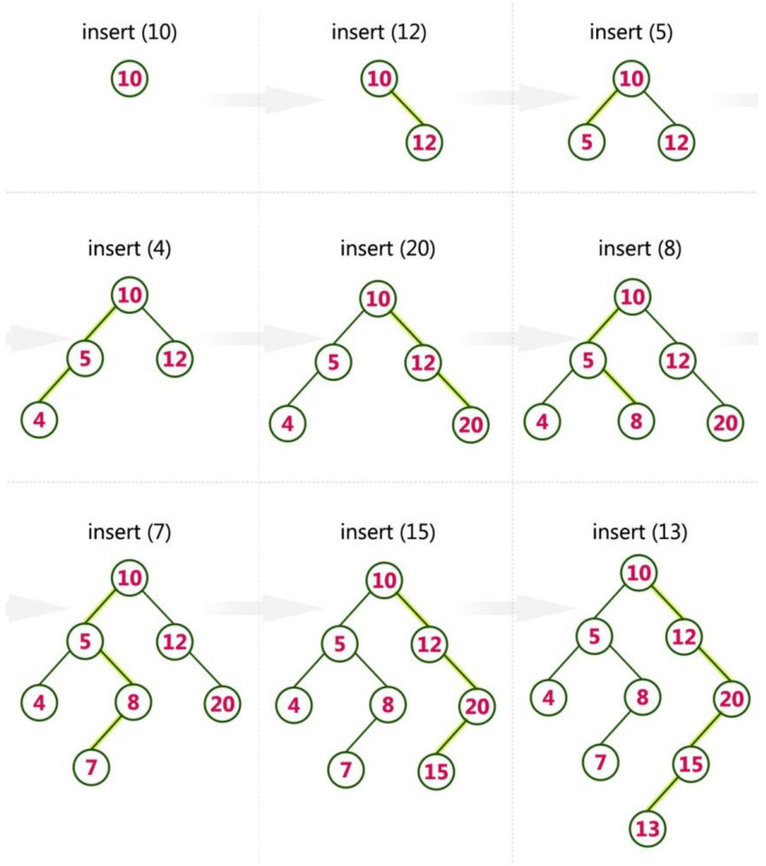
- **Step 1: Find** the node to be deleted using **search operation**
- **Step 2:** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3: Swap** both **deleting node** and node which found in above step.
- **Step 4:** Then, check whether deleting node came to **case 1** or **case 2** else goto steps 2
- **Step 5:** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6:** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7:** Repeat the same process until node is deleted from the tree.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...



<u>Graphs</u>

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of linkes known as edges (or Arcs) which connets the vertices. A graph is defined as follows...

Graph is a collection of vertices and arcs which connects vertices in the graph

Graph is a collection of nodes and edges which connects nodes in the graph

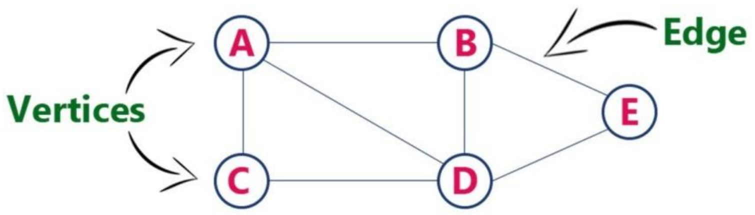
Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Example

The following is a graph with 5 vertices and 6 edges.

This graph G can be defined as G = (V , E)

Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



<u>Graph Terminology</u>

We use the following terms in graph data structure...

Vertex
A individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge
An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).
Edges are three types.

1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge -** A directed egde is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted egde is an edge with cost on it.

Undirected Graph
A graph with only undirected edges is said to be undirected graph.

Directed Graph
A graph with only directed edges is said to be directed graph.

Mixed Graph
A graph with undirected and directed edges is said to be mixed graph.

End vertices or Endpoints
The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Origin
If an edge is directed, its first endpoint is said to be origin of it.

Destination
If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Adjacent
If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

Incident
An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge
A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge
A directed edge is said to be incoming edge on its destination vertex.

Degree
Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree
Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree
Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges
If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

<u>Graph Representations</u>

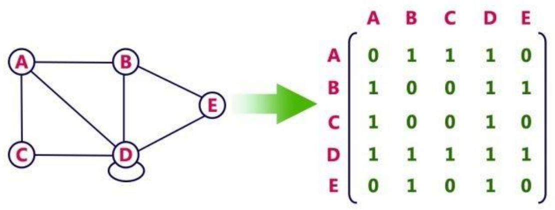
Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

Adjacency Matrix

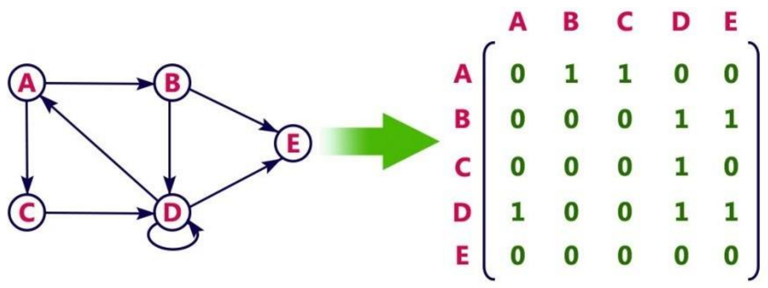
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed graph representation...

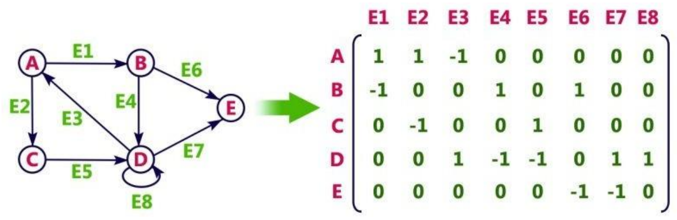


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

Incidence Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

For example, consider the following directed graph representation...

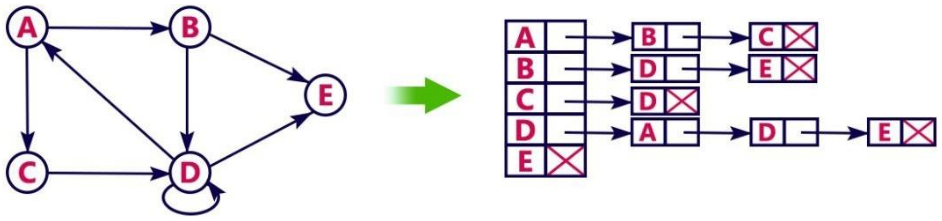


$$\begin{array}{c|cccccccc} & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\ \hline A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

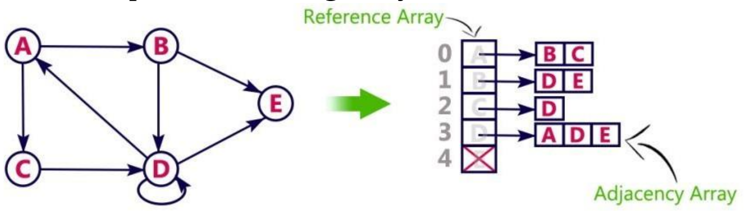
Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



Graph Traversals - DFS

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be used in the search process without creating loops that means using graph traversal we visit all verticces of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

DFS (Depth First Search)

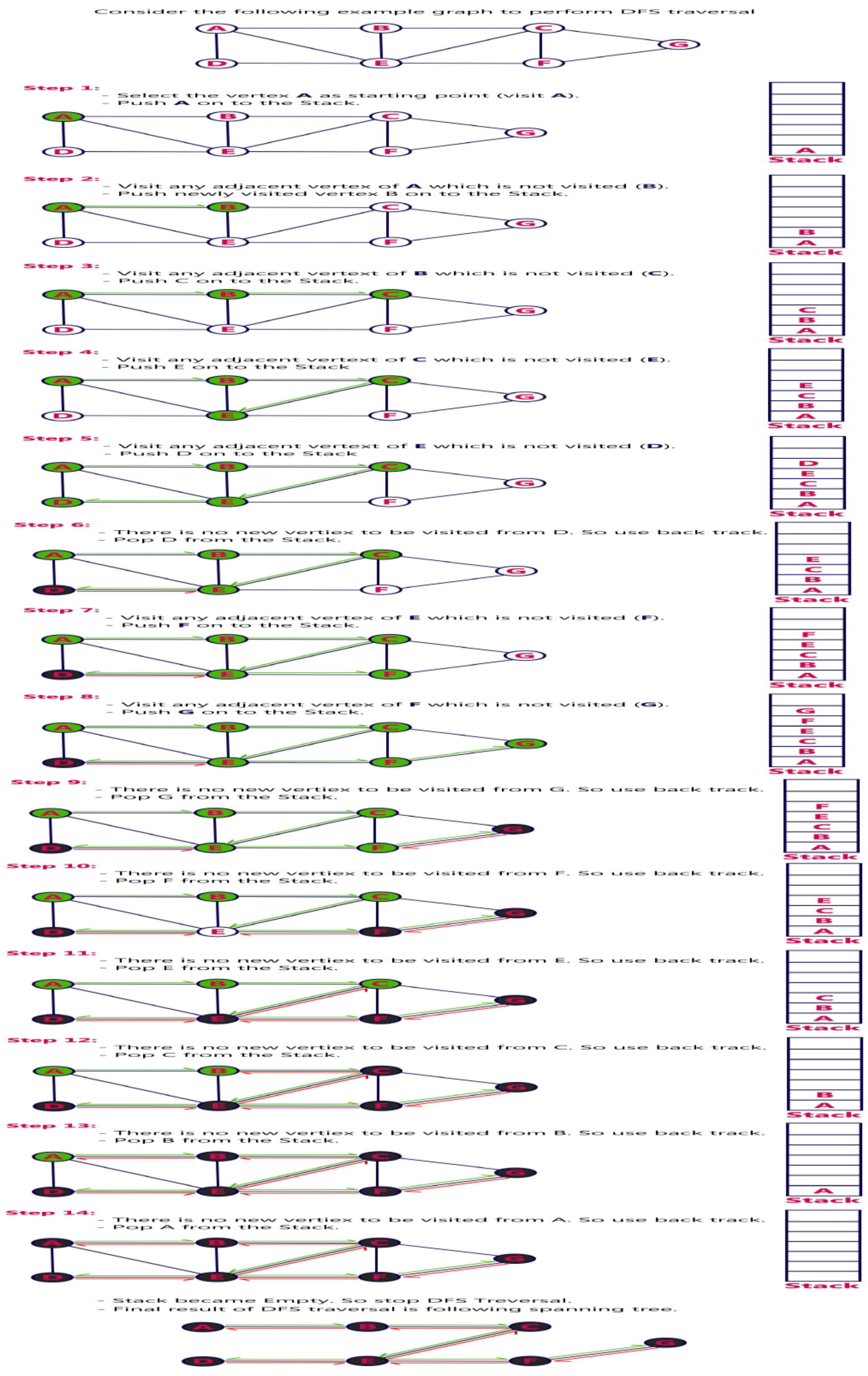
DFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

We use the following steps to implement DFS traversal...

- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the verex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Back tracking is coming back to the vertex from which we came to current vertex.

Example



Consider the following example graph to perform DFS traversal

Step 1:
- Select the vertex A as starting point (visit A).
- Push A on to the Stack.

Step 2:
- Visit any adjacent vertex of A which is not visited (B).
- Push newly visited vertex B on to the Stack.

Step 3:
- Visit any adjacent vertext of B which is not visited (C).
- Push C on to the Stack.

Step 4:
- Visit any adjacent vertext of C which is not visited (E).
- Push E on to the Stack

Step 5:
- Visit any adjacent vertext of E which is not visited (D).
- Push D on to the Stack

Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

Step 7:
- Visit any adjacent vertext of E which is not visited (F).
- Push F on to the Stack.

Step 8:
- Visit any adjacent vertext of F which is not visited (G).
- Push G on to the Stack.

Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

Step 10:
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

Step 11:
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

Step 12:
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

Step 13:
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

Step 14:
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

5)Explain about Graph Traversals - BFS

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be used in the search process without creating loops that means using graph traversal we visit all verticces of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

BFS (Breadth First Search)

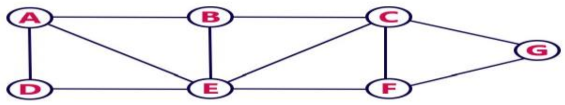
BFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

- **Step 1:** Define a Queue of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3:** Visit all the **adjacent** vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.
- **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- **Step 5:** Repeat step 3 and 4 until queue becomes empty.
- **Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

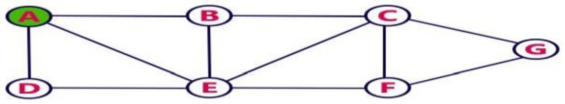
Example

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

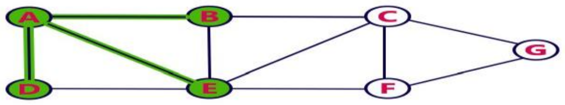


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

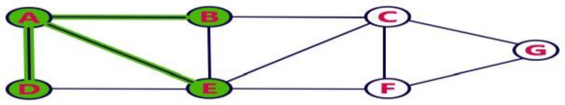


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

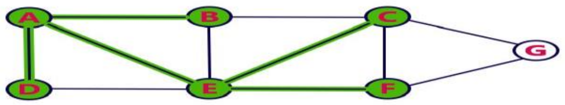


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

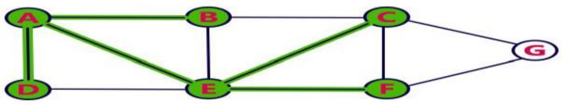


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

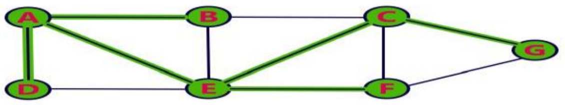


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

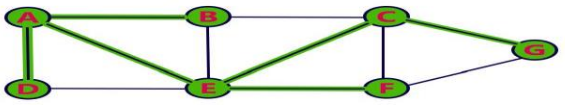


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

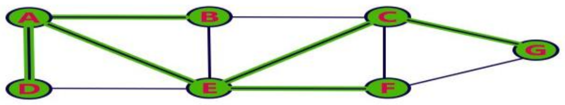


Queue

						G

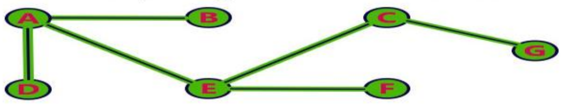
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



UNIT-4

SORTING:

Sorting is nothing but arranging the data in ascending or descending order. The term **sorting** came into picture, as humans realised the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next few tutorials.

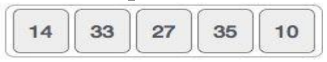
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort
7. Shell Sort

1. Bubble sort:-

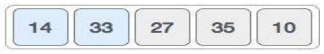
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2) where **n** is the number of items.

How Bubble Sort Works?

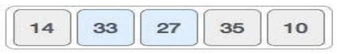
We take an unsorted array for our example. Bubble sort takes O(n^2) time so we're keeping it short and precise.



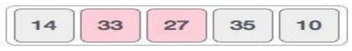
Bubble sort starts with very first two elements, comparing them to check which one is greater.



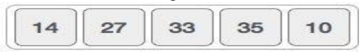
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



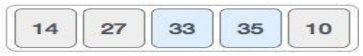
We find that 27 is smaller than 33 and these two values must be swapped.



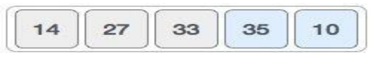
The new array should look like this −



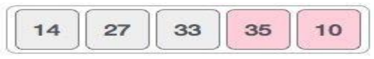
Next we compare 33 and 35. We find that both are in already sorted positions.



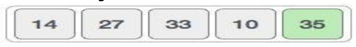
Then we move to the next two values, 35 and 10.



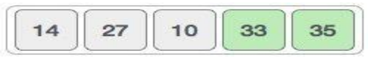
We know then that 10 is smaller 35. Hence they are not sorted.



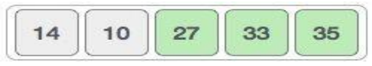
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



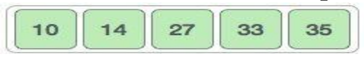
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

```
// Below we have a simple C program for bubble sort
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
   int i, j, temp;
   for(i = 0; i < n; i++)
   {
      for(j = 0; j < n-i-1; j++)
      {
         if( arr[j] > arr[j+1])
         {
            // swap the elements
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }

   // print the sorted array
   printf("Sorted Array: ");
   for(i = 0; i < n; i++)
   {
      printf("%d  ", arr[i]);
   }
}

int main()
{
   int arr[100], i, n, step, temp;
   // ask user for number of elements to be sorted
   printf("Enter the number of elements to be sorted: ");
   scanf("%d", &n);
   // input elements if the array
   for(i = 0; i < n; i++)
   {
      printf("Enter element no. %d: ", i+1);
      scanf("%d", &arr[i]);
   }
   // call the function bubbleSort
   bubbleSort(arr, n);
   getch();
   return 0;
}
```

2.Selection Sort

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped. Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.

Step by Step Process

The selection sort algorithm is performed using following steps...

- **Step 1:** Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all other elements in the list.
- **Step 3:** For every comparision, if any element is smaller than selected element (for Ascending order), then these two are swapped.
- **Step 4:** Repeat the same procedure with next position in the list till the entire list is sorted.

Sorting Logic

Following is the sample code for selection sort...

```
  //Selection sort logic
 for(i=0; i<size; i++){
     for(j=i+1; j<size; j++){
          if(list[i] > list[j])
                 {
              temp=list[i];
              list[i]=list[j];
              list[j]=temp;
          }
     }
 }
```

Example

Complexity of the selection Sort Algorithm

To sort a unsorted list with **'n'** number of elements we need to make **((n-1)+(n-2)+(n-3)+......+1) = (n (n-1))/2** number of comparisions in the worst case. If the list already sorted, then it requires **'n'** number of comparisions.

Worst Case : O(n^2)
Best Case : Ω(n^2)
Average Case : Θ(n^2)

Selection Sort Program in C Programming Language

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

void main(){

  int size,i,j,temp,list[100];
  clrscr();

  printf("Enter the size of the List: ");
  scanf("%d",&size);

  printf("Enter %d integer values: ",size);
  for(i=0; i<size; i++)
      scanf("%d",&list[i]);

  //Selection sort logic

  for(i=0; i<size; i++){
      for(j=i+1; j<size; j++){
           if(list[i] > list[j])
                  {
               temp=list[i];
               list[i]=list[j];
               list[j]=temp;
           }
      }
  }

  printf("List after sorting is: ");
  for(i=0; i<size; i++)
      printf(" %d",list[i]);

  getch();
}
```

3) Insertion Sort

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Step by Step Process

The insertion sort algorithm is performed using following steps...

- **Step 1:** Asume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.
- **Step 2:** Consider first element from the unsorted list and insert that element into the sorted list in order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted list are moved into the sorted list.

Sorting Logic

Following is the sample code for insrtion sort...

```
//Insertion sort logic
for i = 1 to size-1 {
   temp = list[i];
   j = i;
   while ((temp < list[j]) && (j > 0)) {
      list[j] = list[j-1];
      j = j - 1;
   }
   list[j] = temp;
}
```

Complexity of the Insertion Sort Algorithm

To sort a unsorted list with **'n'** number of elements we need to make **(1+2+3+......+n-1) = (n (n-1))/2** number of comparisions in the worst case. If the list already sorted, then it requires **'n'** number of comparisions.

Worst Case : O(n^2)
Best Case : Ω(n)
Average Case : Θ(n^2)

Insertion Sort Program in C Programming Language

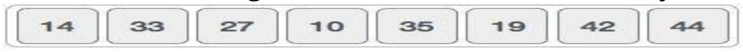
```
#include<stdio.h>
#include<conio.h>

void main(){

   int size, i, j, temp, list[100];

   printf("Enter the size of the list: ");
   scanf("%d", &size);

   printf("Enter %d integer values: ", size);
   for (i = 0; i < size; i++)
      scanf("%d", &list[i]);

   //Insertion sort logic
   for (i = 1; i < size; i++) {
      temp = list[i];
      j = i - 1;
      while ((temp < list[j]) && (j >= 0)) {
         list[j + 1] = list[j];
         j = j - 1;
      }
      list[j + 1] = temp;
   }
    printf("List after Sorting is: ");
   for (i = 0; i < size; i++)
      printf(" %d", list[i]);

  getch();
}
```

4. Merge sort:-

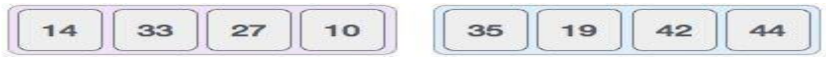
Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

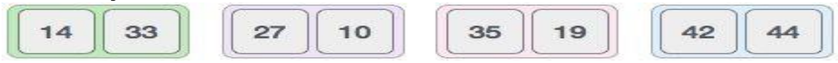
To understand merge sort, we take an unsorted array as the following −



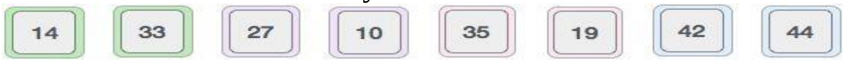
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

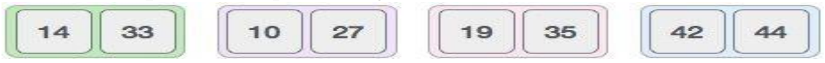


We further divide these arrays and we achieve atomic value which can no more be divided.

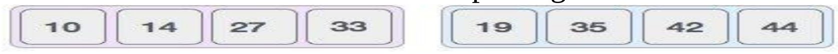


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

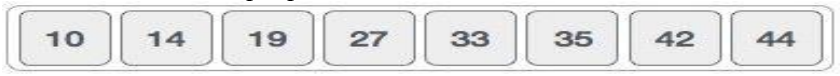
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this −



Now we should learn some programming aspects of merge sorting.

UNIT-5

Search

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Linear Search Algorithm (Sequential Search Algorithm)

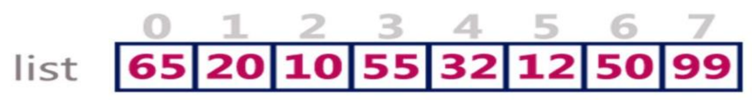
Linear search algorithm finds given element in a list of elements with **O(n)** time complexity where **n** is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching then results with element found otherwise search element is compared with next element in the list. If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". Thatmeans, the search element is compared with element by element in the list.

Linear search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the first element in the list.
- **Step 3:** If both are matching, then display "Given element found!!!" and terminate the function
- **Step 4:** If both are not matching, then compare search element with the next element in the list.
- **Step 5:** Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- **Step 6:** If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Example

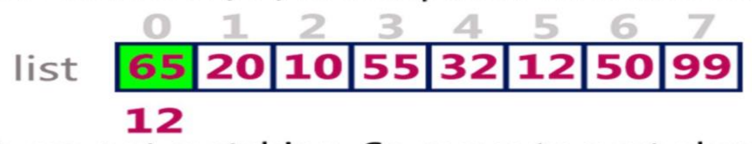
Consider the following list of element and search element...



search element **12**

Step 1:

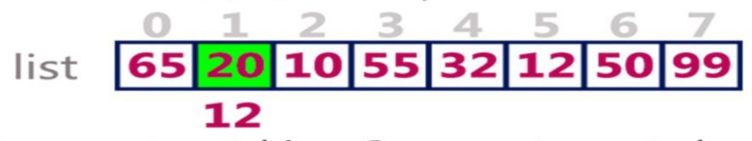
search element (12) is compared with first element (65)



Both are not matching. So move to next element

Step 2:

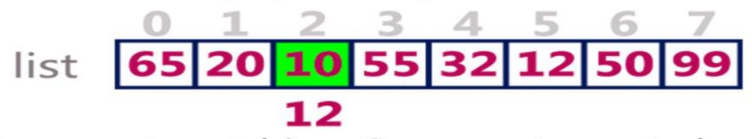
search element (12) is compared with next element (20)



Both are not matching. So move to next element

Step 3:

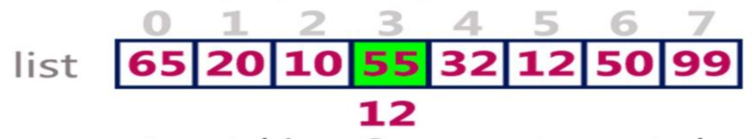
search element (12) is compared with next element (10)



Both are not matching. So move to next element

Step 4:

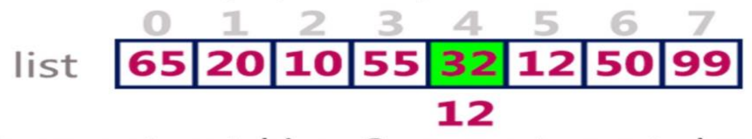
search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

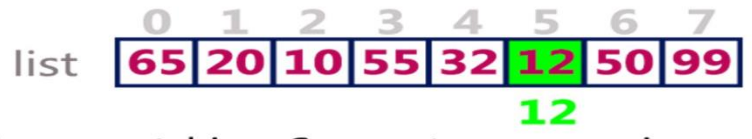
search element (12) is compared with next element (32)



Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)



Both are matching. So we stop comparing and display element found at index 5.

Linear Search Program in C Programming Language

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

void main(){
  int list[20],size,i,sElement;

  printf("Enter size of the list: ");
  scanf("%d",&size);

  printf("Enter any %d integer values: ",size);
  for(i = 0; i < size; i++)
  scanf("%d",&list[i]);

  printf("Enter the element to be Search: ");
  scanf("%d",&sElement);

  // Linear Search Logic
  for(i = 0; i < size; i++)
  {
     if(sElement == list[i])
     {
        printf("Element is found at %d index", i);
        break;
     }
  }
  if(i == size)
     printf("Given element is not found in the list!!!");
  getch();
}
```

Binary Search Algorithm

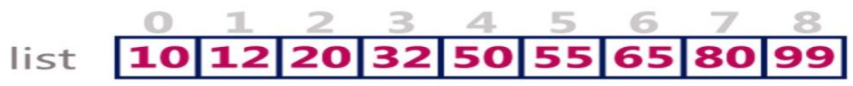
Binary search algorithm finds given element in a list of elements with **O(log n)** time complexity where **n**is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arranged in a order. The binary search can not be used for list of element which are in random order. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Find the middle element in the sorted list
- **Step 3:** Compare, the search element with the middle element in the sorted list.
- **Step 4:** If both are matching, then display "Given element found!!!" and terminate the function
- **Step 5:** If both are not matching, then check whether the search element is smaller or larger than middle element.
- **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8:** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

Example

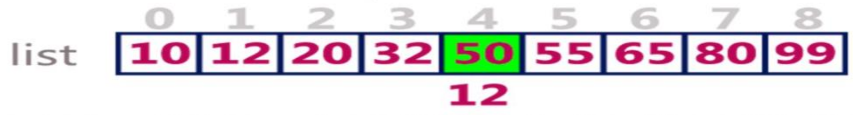
Consider the following list of element and search element...



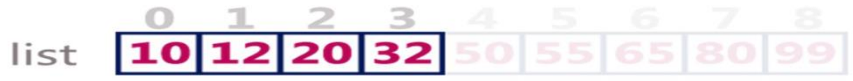
search element **12**

Step 1:

search element (12) is compared with middle element (50)

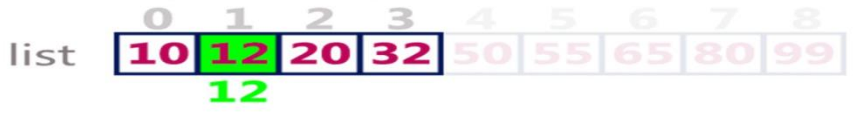


Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).



Step 2:

search element (12) is compared with middle element (12)

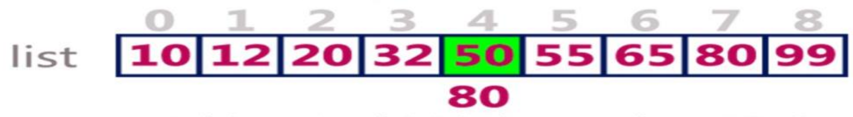


Both are matching. So the result is "Element found at index 1"

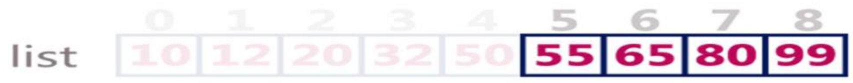
search element **80**

Step 1:

search element (80) is compared with middle element (50)

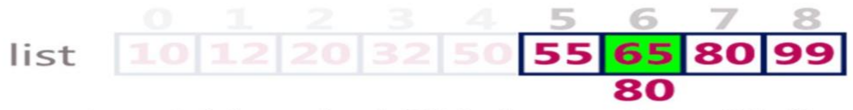


Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

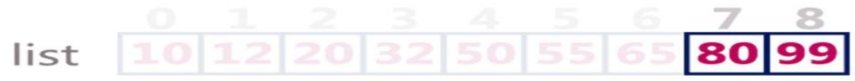


Step 2:

search element (80) is compared with middle element (65)

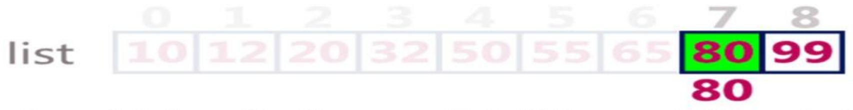


Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).



Step 3:

search element (80) is compared with middle element (80)



Both are not matching. So the result is "Element found at index 7"

Binary Search Program in C Programming Language

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

void main()
{
   int first, last, middle, size, i, sElement, list[100];
   clrscr();

   printf("Enter the size of the list: ");
   scanf("%d",&size);

   printf("Enter %d integer values in Assending order\n", size);

   for (i = 0; i < size; i++)
      scanf("%d",&list[i]);

   printf("Enter value to be search: ");
   scanf("%d", &sElement);

   first = 0;
   last = size - 1;
   middle = (first+last)/2;

   while (first <= last) {
      if (list[middle] < sElement)
         first = middle + 1;
      else if (list[middle] == sElement) {
         printf("Element found at index %d.\n",middle);
         break;
      }
      else
         last = middle - 1;

      middle = (first + last)/2;
   }
   if (first > last)
      printf("Element Not found in the list.");
   getch();
}
```

Hash Tables

Hash table is a data structure used for storing and retrieving data quickly. Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associate with some key. For example, for an employee record in the hash table employee ID will works as a key.

→ Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is dependent upon the size of the hash table.
→ The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.

Buck and Home bucket

The hash function H(key) is used to map a several dictionary entries in the hash table. Each function of hash table is called bucket. The function H(k) is home bucket for the dictionary with pail whose value is key.

	5
15	4
	3
10	2
	1
	0

Hash table

In the above diagram or hash table location 2 or 4 is called as home bucket and location 0,1,3,5 are called as bucket.

Static and Dynamic hashing

There are two types of hashing. They are:
1. Static hashing
2. Dynamic hashing

Static hashing

Static hashing is a hashing technique in which keys are stored in which keys are stored in hash table with fixed size.

Dynamic hashing

In this hashing table, the hash function is modified dynamically number of records grow.

Hash function

Hash function is a function which is used to put data into hash table. Hence one can use the same as function to retrieve the data from hash table. Thus hash function is used implement a hash table.

There are several types of hash function.
1. Division hash function method
2. Mid square hash function method
3. Multiplication or multiplicative hash function
4. Digit folding or folding hash function

Division hash function method

The hash function depends upon the remainder of the division. Typically the division is the table length.

Syntax or Formula

H (key) = K % table size

Example:-

Insert following values or records
54, 72, 89, 37 into hash table. The hash table size is 10.

The following determines hash table with size 10.

	9
	8
	7
	6
	5
	4
	3
	2
	1
	0

The record 54 is inserted into above hash table by using division hash function.

H (key)=k % table size

H(Key) =54%10=4

The record 54 is inserted at 4th location.
The record 72 is inserted into above hash table by using division hash function.

H (key)=k % table size

H(Key)=72%10=2

The record 72 is inserted at 2nd location.
The record 89 is inserted into above hash table by using division hash function.

H (key)=k % table size

H(Key) =89%10=9

The record 89 is inserted at 9th position or location.
The record 37 is inserted into above hash table by using division hash function.

H (key)=k % table size

H(Key) =37%10=7
The record 37 is inserted at 7th position.
The following hash table determines the inserting records 54, 72, 89, 37 into hash table.

89	9
	8
37	7
	6
	5
54	4
	3
72	2
	1
	0

Mid square hash function

In the mid square method, the key is squared and the middle or mid part of the result is used as index or position or location.

Example the records 311, 3112, 3114 are inserted to hash table. Assume that hash table size is 1000.

Syntax or formula

H(Key) = K^2

The record 3111 by using mid square.

H(key) =K^2

= $(3111)^2$

=9678321

783 is the middle part of 9678321. So, 783 is the index of 3111.

The record 3112 by using mid square

H (Key) = $(3112)^2$

= 9684544

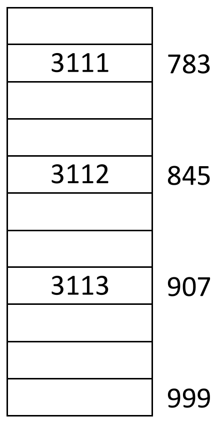
845 is the middle part of 9684544. So, 845 is the index of 3112.

The record 3113 by using mid square

H (Key) = $(3113)^2$

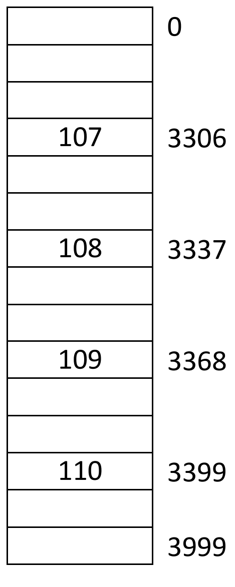
= 9690769

907 is the middle part of 9690769. So, 907 is the index of 3113.



Multiplicative hash function

The given record is multiplied by some constant value. The formula computing hash key is
H (Key) = floor (P*(fractional part of key*A))
Where 'P' is an integer constant and 'A' is real constant.
Donald Knuth suggested to use constant A = 0.61803398987.
Example:-
Insert the following records 107, 108, 109, 110 into hash table . Here P =50.
107 inserted into hash table by using multiplicative hash function.
H (Key) = floor (P*(fractional part of key*A))
= floor (50*(107* 0.61803398987)
= floor (3306.4818)
=3306
108 inserted into hash table by using multiplicative hash function.
H (Key) = floor (50*(108* 0.61803398987)
= floor (3337.3835)
= 3337
109 inserted into hash table by using multiplicative hash function.
H (Key) = floor (50*(109* 0.61803398987)
= floor (3368.2852)
= 3368
110 inserted into hash table by using multiplicative hash function.
H (Key) = floor (50*(110* 0.61803398987)
= floor (3399.1869)
= 3399
The following diagram determines the 107, 108, 109, 110 values into hash table.



Digit folding or folding hash function

The key value is divided into separate parts and using some simple operation this parts are combined to produce hash key.

Example:-

Consider the record 1, 2, 3, 6, 5, 4, 1, 2 then it is divided into separate parts 123, 654, 12 and this all are added together.

H (Key) = 123+ 654 + 12 +789

The record 123, 654, 12 will be placed at a location 789 in the hash table.

Collision Resolution Technique

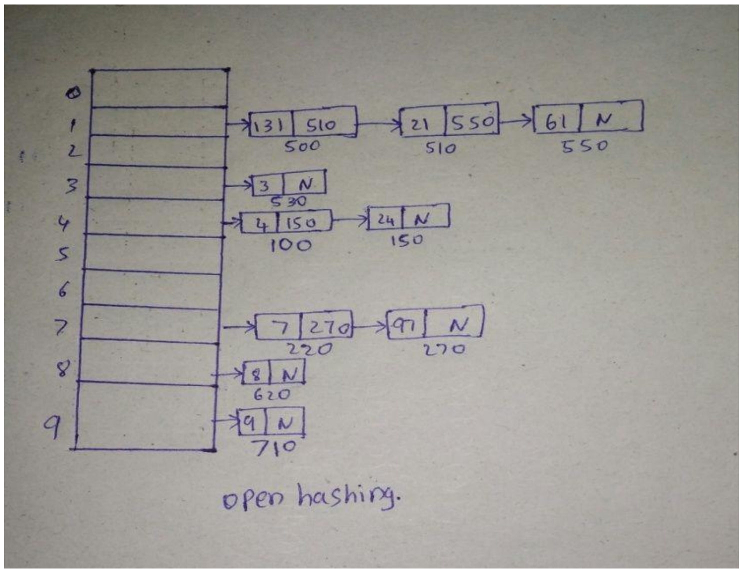
If collision occurs then it should be handled by applying some techniques. Such techniques are called collision resolution technique.

The goal of collision resolution techniques is to minimize collisions. There are two methods of handling collisions.

1. Open hashing or Separate Chain hashing
2. Closed hashing or Open addressing

The difference between open hashing and closed hashing is that in Open hashing the collision are stored outside table and in Closed hashing the collisions are stored in the same table at some another slot.

Open hashing



In collision handling method chaining is a concept which introduces an additional fields with data i.e., chain. A separate chain table is maintained for colliding data when collision occurs then linked list is maintained at home bucket.

Example:-

Consider the keys to be placed in the in their home buckets are 131, 3, 4, 21, 61, 24, 7, 97, 8, 9.

A chain is maintained for colliding elements. For distance 131 has a home bucket index 1. Similarly keys 21 and 61 demand for home bucket index 1. Hence a chain is maintained at index 1. Similarly the chain at index 4 and 7 is maintained.

Closed hashing

Closed hashing collision resolution strategy or technique which users following technique.

1. Linear probing
2. Quadratic probing
3. Double probing or Double hashing

Linear probing

This is the easiest method of handling collision. When collision occurs i.e., when two records demand for the same home bucket in the hash table then collision can be solved by placing the second record linearly down whenever the empty bucket is found. When use linear probing the hash table is represented as a one dimensional array with indices that range from 0 to desired table size-1.

Example:-

Consider that following keys are to be inserted in the hash table 131, 4, 8, 7, 21, 5, 31, 61, 9, 29.The hash table size is 10.

Initially we will put the following keys in the hash table 131, 4, 8, 7.

We will use division hash function. That means that keys are placed using formula.

H (Key) = key % table size

For instance the element 131 can be placed at H (Key) = 131 % 10 =1.

Index 1 will be the home bucket for 131. Continuing in the fashion we will place 4,8,7.

0	Null
1	131
2	Null
3	Null
4	4
5	Null
6	Null
7	7
8	8
9	Null

Now the next to be inserted is 21. According to hash function H (Key) = 21 % 10 =1.

But the index 1 location already occupied with 131 i.e., collision occurs. To resolve this collision we will linearly move down from 1 to empty location is found. Therefore 21 will be placed at index 2. If the next element is 5 then we get home bucket for 5 as index 5 this bucket is empty so, we will put the

element 5 at index 5.

0	Null
1	131
2	21
3	Null
4	4
5	5
6	Null
7	7
8	8
9	Null

After placing record keys 31, 61 the hash table will be

0	Null
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	Null

The next record key that comes is 9. According to decision as function it demands for the home bucket 9. Hence we will place 9 at index 9.

0	Null
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	9

Now the next final record key is 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs to handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0^{th} index.

0	29
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	9

Quadratic probing

Quadratic probing operates by taking original hash value and adding successive values of quadratic polynomial to the stating value.

This method uses following formula.

H(Key) = (H (Key) = i^2) % m

Where ‘m’ can be table size or any prime number.

Example: - If we have insert following elements in the hash table with table size 10.

37, 19, 55, 22, 17, 49, 87.

Initially we will put following keys into hash table.

37,19,55,22.

0	
1	
2	22
3	
4	
5	55
6	
7	37
8	
9	

Now, if you want to place 17 a collision will be occurs 17. 17 % 10 = 7, but bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

H (Key) = (H (Key) = i^2) % m

Consider I =0

H (key) = (17+0^2) % 10 = 17 % 10 = 7.

Then i=1

H (Key) = $(17 + 1^2)$ % 10 =18 % 10 = 8.

The bucket 8 is empty. Hence we will place the element of the index 8.

Now if you want to place 49 a collision will be occur 49 % 10 = 9 and bucket 9 as already occupied with 19. Hence we will applying quadratic probing to insert this record in the hash table.

H_i (Key) = (H (Key) + i^2) % m

I =0

= (49 + 0) % 10 = 49 % 10 = 9

I = 1

= $(49 + 1^2)$ % 10 = 50 % 10 = 0

0	49
1	
2	22
3	
4	
5	55
6	
7	37
8	17
9	19

The bucket 0 is empty.

Hence the value 49 is inserted at a 0th position.

Now to place 87 we will use quadratic probing.

H(Key) = $(87 + 0^2)$ % 10 = 87 % 10 = 7

H(Key) = $(87 + 1^2)$ % 10 = 88 % 10 = 8

H(Key) = $(87 + 2^2)$ % 10 = 91 % 10 = 1

0	49
1	87
2	22
3	
4	
5	55
6	
7	37
8	17
9	19

Double probing or double hashing

Double hashing is a technique in which a second hash function is applied to key when a collision occur by applying the second has function we will get number of positions from the point of Collision inserted.

By using following formulas we can find out the double hashing.

$$H_1(\text{key}) = k \,\%\, \text{table size}$$

$$H_2(\text{key}) = M - (K \,\%\, M)$$

Where M is prime number smaller than the size of the table.

Example: consider the following elements to be placed in the Hash table of size 10.

37, 90, 45, 22, 17, 49, 55.

Inside Initially the elements using the formula for H_1 (key). Insert 37, 90, 45, 22.

37 % 10 = 7

90 % 10 = 0

45 % 10 = 5

22 % 10 = 2

0	90
1	
2	22
3	
4	
5	45
6	
7	37
8	
9	

Now if 17 is to be inserted then

H_1 (17) = 17 % 10 = 7

Here collision will be occur because 7th position already occupied with element 37 or record 37. So we can apply second hash function to key.

$$H_2(\text{key}) = M-(K\%M)$$

Here M is prime number smaller than the size of the table.

Let us prime number is M = 7

H_2 (17) =7-(17%7)

= 7 - 3 = 4

That means we have to

insert the elements on 10 at 4 places from value 37 or 7th position.

0	90
1	17
2	22
3	
4	
5	45
6	
7	37
8	
9	

17 will be placed at index 1.

Now to insert number 49 at location 9th position that is 49 % 10 = 9.

0	90
1	17
2	22
3	
4	
5	45
6	
7	37
8	
9	49

Now to insert number 55.

H_1 (55) = 55 % 10 = 5 that is collision will be occur. Because the location 5 already occupied with 45. So, we can apply second hash function.

$$H_2 (55) = 7 - (55 \% 7) = 7 - 6 = 1$$

That means we have to take one jump from index 5 to place 55.

0	90
1	17
2	22
3	
4	
5	45
6	55
7	37
8	
9	49

Rehashing

Rehashing is a technique in which table is resized that is the size of table is double by creating a new table. It is preferable if the total size of new table is a prime number. There are situation in which rehashing is required.

i) When the table size is completely full.

ii) With Quadratic probing when the table is filled half.

iii) When insertion fail due to over flow.

In such situations, we have to transfer entries from old table to new table.

Example:

Consider we have to insert the elements 37, 90, 55, 22, 17, 49 and 87 the table size is 10 and will use hash function.

H (key) = K % Table size

Initially insert following elements 37, 90, 55, 22

0	90
1	
2	22
3	
4	
5	55
6	
7	37
8	
9	

Now you can insert 17 into hash table. Here collision will be occur. Because the 7th location already occupied with 37. So, by using linear probing the element 17 is insert at 8th position.

0	90
1	
2	22
3	
4	
5	55
6	
7	37
8	17
9	

Now this table is almost full. So, next element 87 is not inserted into hash table because the hash table is overflow. Hence we will rehashing by double the size for new table that becomes 20. But 20 is not prime number we will prefer to make table size as 23 and new hash function will be

H (key) = k % 23

37 % 23 = 14

90 % 23 = 21

55 % 23 = 9

22 % 23 = 22

17 % 23 = 17

49 % 23 = 3

87 % 23 = 18

0	
1	
2	
3	49
4	
5	
6	
7	
8	
9	55
10	
11	
12	
13	
14	37
15	
16	
17	17
18	87
19	
20	
21	90
22	22