# Practical No. 1 (A)

**Title:** Write a program to calculate factorial of number without recursion.

**Problem Definition:** Here problem is to find factorial of number which is read from the user using iterative approach.

**Software Requirement:** Turbo C

**Algorithm:**

Step 1: Input a number **n**

Step 2: set variable **final** as 1

Step 3: final <= final * n

Step 4: decrease **n**

Step 5: check if **n** is equal to 0

Step 6: if **n is** equal to zero, goto step 8 (break out of loop)

Step 7: else goto step 3

Step 8: print the result **final**

**Code:**

```
#include<stdio.h>

int main(){
  int i=1,f=1,num;

  printf("Enter a number: ");
  scanf("%d",&num);

  while(i<=num){
     f=f*i;
     i++;
  }

  printf("Factorial of %d is: %d",num,f);
```

```
  return 0;
}
```

**Input Specification:** Enter a number: 5

**Output Specification:** Factorial of 5 is: 120

**Conclusion:** Thus we have successfully implemented factorial of n number without using recursion.

# Practical No. 1 (B)

**Title:** Write a program to calculate factorial of number with recursion.

**Problem Definition:** Here problem is to find factorial of number which is read from the user using recursive approach.

**Software Requirement: Turbo C**

**Algorithm:**

integer factorial(integer n) {
**if** (n < 0) {
error: "Factorials cannot be defined for negative numbers"
}
**if** (n == 0) {
**return** 1;
}
**return** n * factorial(n - 1);
}

**Code:**

```
#include<stdio.h>

int findFactorial(int);
int main(){
  int i,factorial,num;

  printf("Enter a number: ");
  scanf("%d",&num);

  factorial = findFactorial(num);
  printf("Factorial of %d is: %d",num,factorial);
```

```
  return 0;
}

int findFactorial(int num){
    int i,f=1;

    for(i=1;i<=num;i++)
      f=f*i;

     return f;
}
```

**Input Specification:** Enter a number: 8

**Output Specification:** Factorial of 8 is: 40320

**Conclusion:** Thus we have successfully implemented factorial of n number using recursion.

# Practical No. 2 (A)

**Title:** Write a program to print Fibonacci Series without recursion.

**Problem Definition:** Here is to display Fibonacci series which prints 0 1 1 2 3 5 8 ….. using iterative approach.

**Software Requirement: Turbo C**

**Algorithm:**

Start

Declare variables i, a,b , show

Initialize the variables, a=0, b=1, and show =0

Enter the number of terms of Fibonacci series to be printed

Print First two terms of series

Use loop for the following steps
-> show=a+b
-> a=b
-> b=show
-> increase value of i each time by 1
-> print the value of show

End

**Code:**

```
#include<stdio.h>

int main()
{
```

```
  int n, first = 0, second = 1, next, c;

  printf("Enter the number of terms\n");
  scanf("%d",&n);

  printf("First %d terms of Fibonacci series are :-\n",n);

  for ( c = 0 ; c < n ; c++ )
  {
    if ( c <= 1 )
      next = c;
    else
    {
      next = first + second;
      first = second;
      second = next;
    }
    printf("%d\n",next);
  }

  return 0;
}
```

**Input Specification:** Enter the number of terms: 10

**Output Specification:** First 10 terms of Fibonacci series are:

0 1 1 2 3 5 8 13 21 44

**Conclusion:** Thus we have successfully implemented Fibonacci series without recursion.

# Practical No. 2 (B)

**Title:** Write a program to print Fibonacci Series using recursion.

**Problem Definition:** Here is to display Fibonacci series which prints 0 1 1 2 3 5 8 ….. using recursive approach.

**Software Requirement: Turbo C**

**Algorithm:**

```
int factorial(int n) {
   if (n == 0)
      return 1;
   else
      return (n * factorial(n-1));
}
```

**Code:**

```
#include<stdio.h>
int Fibonacci(int);
main()
{
  int n, i = 0, c;

printf("Enter the number of terms\n");
 scanf("%d",&n);
 printf("Fibonacci series\n");
 for ( c = 1 ; c <= n ; c++ )
  {
    printf("%d\n", Fibonacci(i));
    i++;
  }
 return 0;
}
```

```
 int Fibonacci(int n)
{
  if ( n == 0 )
    return 0;
  else if ( n == 1 )
    return 1;
  else
    return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

**Input Specification:** Enter the number of terms: 10

**Output Specification:** Fibonacci series :

0 1 1 2 3 5 8 13 21 44

**Conclusion:** Thus we have successfully implemented Fibonacci series using recursion.

# Practical No. 3

**Title:** Write a program to implement Binary search using Divide & Conquer approach.

**Problem Definition:** We have to implement searching operating using binary search algorithm which uses divide and conquer method to find out the location of searching element which us entered by user in the array list.

**Software Requirement: Turbo C**

**Algorithm:**

```
// initially called with low = 0, high = N – 1
  BinarySearch_Right(A[0..N-1], value, low, high) {
     // invariants: value >= A[i] for all i < low
                    value < A[i] for all i > high
     if (high < low)
        return low
     mid = low +((high – low) / 2) // THIS IS AN IMPORTANT STEP TO AVOID BUGS
     if (A[mid] > value)
        return BinarySearch_Right(A, value, low, mid-1)
     else
        return BinarySearch_Right(A, value, mid+1, high)
  }
```

**Code:**

```
#include<stdio.h>
int main(){

   int a[10],i,n,m,c,l,u;
```

```
   printf("Enter the size of an array: ");
   scanf("%d",&n);

   printf("Enter the elements of the array: " );
   for(i=0;i<n;i++){
       scanf("%d",&a[i]);
   }

   printf("Enter the number to be search: ");
   scanf("%d",&m);

   l=0,u=n-1;
   c=binary(a,n,m,l,u);
   if(c==0)
       printf("Number is not found.");
   else
       printf("Number is found at % d: , mid+1");

   return 0;
 }

int binary(int a[],int n,int m,int l,int u){

    int mid,c=0;

    if(l<=u){
        mid=(l+u)/2;
        if(m==a[mid]){
           c=1;
        }
        else if(m<a[mid]){
           return binary(a,n,m,l,mid-1);
        }
        else
           return binary(a,n,m,mid+1,u);
    }
    else
      return c;
}
```

**Input Specification:** Enter the size of an array: 5

Enter the elements of the array: 8 9 10 11 12
Enter the number to be search: 11

**Output Specification:** Number is found at 4

**Conclusion:** Thus we have successfully implemented Binary search using Divide & Conquer approach.

# Practical No. 4

**Title:** Write a program to implement Merge Sort.

**Problem Definition:** We have to implement sorting operating using merge sort algorithm which will break the array in number of subparts and then solve each sub-part individually and combine the solution.

**Software Requirement: Turbo C**

**Algorithm:**

**function** merge_sort(m)

**if** length(m) ≤ 1

**return** m

**var** *list* left, right, result

**var** *integer* middle = length(m) / 2

**for each** x **in** m **up to** middle

add x to left

**for each** x **in** m **after** middle

add x to right

left = merge_sort(left)

right = merge_sort(right)

result = merge(left, right)

**return** result

**Code:**

```
#include<stdio.h>
void merge(int [],int ,int ,int );
void part(int [],int ,int );
int main()
```

```
{
intarr[30];
inti,size;
printf("\n\t------- Merge sorting method -------\n\n");
printf("Enter total no. of elements : ");
scanf("%d",&size);
for(i=0; i<size; i++)
{
  printf("Enter %d element : ",i+1);
  scanf("%d",&arr[i]);
}
 part(arr,0,size-1);
printf("\n\t------- Merge sorted elements -------\n\n");
for(i=0; i<size; i++)
printf("%d ",arr[i]);

 }


void part(intarr[],intmin,int max)
{
 int mid;
 if(min<max)
 {
   mid=(min+max)/2;
   part(arr,min,mid);
   part(arr,mid+1,max);
   merge(arr,min,mid,max);
}
}


void merge(intarr[],intmin,intmid,int max)
{
  inttmp[30];
  inti,j,k,m;
  j=min;
  m=mid+1;
  for(i=min; j<=mid && m<=max ; i++)
 {
   if(arr[j]<=arr[m])
```

```
 {
   tmp[i]=arr[j];
   j++;
 }
 else
  {
   tmp[i]=arr[m];
   m++;
  }
 }
 if(j>mid)
 {
   for(k=m; k<=max; k++)
  {
   tmp[i]=arr[k];
   i++;
  }
}



 else
  {
   for(k=j; k<=mid; k++)
  {
    tmp[i]=arr[k];
    i++;
  }
 }
 for(k=min; k<=max; k++)
   arr[k]=tmp[k];
}
```

**Input Specification:**

------- Merge sorting method -------

Enter total  no. of elements : 9

Enter 1 element : 12

Enter 2 element : 23

Enter 3 element : 34

Enter 4 element : 11

Enter 5 element : 33

Enter 6 element : 55

Enter 7 element : 66

Enter 8 element : 54

Enter 9 element : 22

**Output Specification:**

------- Merge sorted elements -------

11 12 22 23 33 34 54 55 66

**Conclusion:** Thus We have successfully implemented Merge sort.

# Practical No. 5

**Title:** Write a program to implement Strassen's Multiplication of 2*2 matrixes.

**Problem Definition:** We have to perform multiplication of 2*2 matrices using strassen's algorithm.

**Software Requirement: Turbo C**

**Algorithm:**

1. Partition A, B and and C into 4 equal parts:

$$A = \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix}$$

$$B = \begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{matrix}$$

$$C = \begin{matrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{matrix}$$

2. Evaluate the intermediate matrices:

$M_1 = (A_{11} + A_{22}) (B_{11} + B_{22})$
$M_2 = (A_{21} + A_{22}) B_{11}$
$M_3 = A_{11} (B_{12} - B_{22})$
$M_4 = A_{22} (B_{21} - B_{11})$
$M_5 = (A_{11} + A_{12}) B_{22}$
$M_6 = (A_{21} - A_{11}) (B_{11} + B_{12})$
$M_7 = (A_{12} - A_{22}) (B_{21} + B_{22})$

3. Construct C using the intermediate matrices:

$C_{11} = M_1 + M_4 - M_5 + M_7$
$C_{12} = M_3 + M_5$
$C_{21} = M_2 + M_4$
$C_{22} = M_1 - M_2 + M_3 + M_6$

**Code:**

```
#include<stdio.h>
int main(){
  int a[2][2],b[2][2],c[2][2],i,j;
  int m1,m2,m3,m4,m5,m6,m7;

  printf("Enter the 4 elements of first matrix: ");
  for(i=0;i<2;i++)
     for(j=0;j<2;j++)
         scanf("%d",&a[i][j]);

  printf("Enter the 4 elements of second matrix: ");
  for(i=0;i<2;i++)
     for(j=0;j<2;j++)
         scanf("%d",&b[i][j]);

  printf("\nThe first matrix is\n");
  for(i=0;i<2;i++){
     printf("\n");
     for(j=0;j<2;j++)
         printf("%d\t",a[i][j]);
  }

  printf("\nThe second matrix is\n");
  for(i=0;i<2;i++){
     printf("\n");
     for(j=0;j<2;j++)
         printf("%d\t",b[i][j]);
  }

  m1= (a[0][0] + a[1][1])*(b[0][0]+b[1][1]);
  m2= (a[1][0]+a[1][1])*b[0][0];
  m3= a[0][0]*(b[0][1]-b[1][1]);
  m4= a[1][1]*(b[1][0]-b[0][0]);
  m5= (a[0][0]+a[0][1])*b[1][1];
  m6= (a[1][0]-a[0][0])*(b[0][0]+b[0][1]);
  m7= (a[0][1]-a[1][1])*(b[1][0]+b[1][1]);

  c[0][0]=m1+m4-m5+m7;
  c[0][1]=m3+m5;
  c[1][0]=m2+m4;
  c[1][1]=m1-m2+m3+m6;

   printf("\nAfter multiplication using \n");
   for(i=0;i<2;i++){
     printf("\n");
     for(j=0;j<2;j++)
```

```
        printf("%d\t",c[i][j]);
  }

  return 0;
}
```

**Input Specification:**

Enter the 4 elements of first matrix: 1
2
3
4
Enter the 4 elements of second matrix: 5
6
7
8

The first matrix is

1 2
3 4
The second matrix is

5 6
7 8

**Output Specification:**

After multiplication using

19 22
43 50

**Conclusion:**

Thus we have successfully implement strassen's multiplication of 2*2 matrices.

# Practical No. 6

**Title:** Write a program to implement Knapsack algorithm.

**Problem Definition:** The problem is we have to fulfill a knapsack (bag) in such way that it either contain maximum profit or less weight which is the nothing but capacity of that bag.

**Software Requirement:** Turbo C

**Algorithm:**

```
// Input:
 2 // Values (stored in array v)
 3 // Weights (stored in array w)
 4 // Number of distinct items (n)
 5 // Knapsack capacity (W)
 6
 7 for j from 0 to W do:
 8     m[0, j] := 0
 9
10 for i from 1 to n do:
11     for j from 0 to W do:
12         if w[i] > j then:
13             m[i, j] := m[i-1, j]
14         else:
15             m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

**Code:**

```
# include<stdio.h>

void knapsack(int n, float weight[], float profit[], float capacity) {
  float x[20], tp = 0;
  int i, j, u;
  u = capacity;

  for (i = 0; i < n; i++)
    x[i] = 0.0;

  for (i = 0; i < n; i++) {
    if (weight[i] > u)
```

```
      break;
    else {
      x[i] = 1.0;
      tp = tp + profit[i];
      u = u - weight[i];
    }
  }

  if (i < n)
    x[i] = u / weight[i];

  tp = tp + (x[i] * profit[i]);

  printf("\nThe result vector is:- ");
  for (i = 0; i < n; i++)
    printf("%f\t", x[i]);

  printf("\nMaximum profit is:- %f", tp);

}

int main() {
  float weight[20], profit[20], capacity;
  int num, i, j;
  float ratio[20], temp;

  printf("\nEnter the no. of objects:- ");
  scanf("%d", &num);

  printf("\nEnter the wts and profits of each object:- ");
  for (i = 0; i < num; i++) {
    scanf("%f %f", &weight[i], &profit[i]);
  }

  printf("\nEnter the capacityacity of knapsack:- ");
  scanf("%f", &capacity);

  for (i = 0; i < num; i++) {
    ratio[i] = profit[i] / weight[i];
  }

  for (i = 0; i < num; i++) {
    for (j = i + 1; j < num; j++) {
```

```
        if (ratio[i] < ratio[j]) {
          temp = ratio[j];
          ratio[j] = ratio[i];
          ratio[i] = temp;

          temp = weight[j];
          weight[j] = weight[i];
          weight[i] = temp;

          temp = profit[j];
          profit[j] = profit[i];
          profit[i] = temp;
        }
      }
    }

  knapsack(num, weight, profit, capacity);
  return(0);
}
```

**Input Specification:**

Enter the no. of objects:- 7

Enter the wts and profits of each object:-
2 10
3 5
5 15
7 7
1 6
4 18
1 3

Enter the capacity of knapsack:- 15

**Output Specification:**

The result vector is:- 1.000000 1.000000 1.000000 1.000000
1.000000 0.666667 0.000000

Maximum profit is:- 55.333332

**Conclusion:** Thus we have successfully implement knapsack Problem using greedy approach.

# Practical No. 7

**Title:** Write a program to implement Kruskal's algorithm.

**Problem Definition:** The problem is to find minimum spanning tree using Kruskal's algorithm in such way that all the vertices should be visited once and cost of visited edge is less as possible.

**Software Requirement:** Turbo C

**Algorithm:**

**Function Kruskal(G=<N,A>:graph, length: A→$R^+$): set of edges**

{Initialization}

Sort A by increasing length

n← the number of nodes in N

T←Ǿ {will contain the edges of the minimum spanning tree}

Initialize n sets, each containing a different element of N

{Greedy loop}

repeat

e←{u, v}← shortest edge not yet considered

ucomp← find(u)

vcomp← find (v)

if ucomp≠vcomp then

merge(ucomp,vcomp)

T←T U {e}

until T contails n-1 edges

return T

**Code:**

```
#include<stdio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
printf("\n\n\tImplementation of Kruskal's algorithm\n\n");
printf("\nEnter the no. of vertices\n");
scanf("%d",&n);
printf("\nEnter the cost adjacency matrix\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
scanf("%d",&cost[i][j]);
if(cost[i][j]==0)
cost[i][j]=999;
}
}
printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
while(ne<n)
```

```
{
for(i=1,min=999;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if(cost[i][j]<min)
{
min=cost[i][j];
a=u=i;
b=v=j;
}
}
}
u=find(u);
v=find(v);
if(uni(u,v))
{
printf("\n%d edge (%d,%d) =%d\n",ne++,a,b,min);
mincost +=min;
}
cost[a][b]=cost[b][a]=999;
}
printf("\n\tMinimum cost = %d\n",mincost);
getch();
}
```

```
int find(int i)
{
while(parent[i])
i=parent[i];
return i;
}
int uni(int i,int j)
{
if(i!=j)
{
parent[j]=i;
return 1;
}
return 0;
}
```

**Input Specification:**

Implementation of Kruskal's algorithm

Enter the no of vertices

4

Enter the cost adjacency matrx

0 20 10 50

20 0 60 999

10 60 0 40

50 999 40 0

**Output Specification:**

The edges of Minimum Cost Spanning Tree are

1 edge <1,3> = 10

2 edge <1,2> = 20

3 edge <3,4> = 40

Minimum cost = 70

**Conclusion: -**

Thus we have studied and implement Kruskal's algorithm**.** In this algorithm no initial node is selected and the edge is selected as per the criteria means it must give the minimum cost and no cycle will form by that edges. So all the edges from the given graph is arranged in the descending order of the cost and then select the edge one after another till all the nodes are visited from graph to form a spanning tree.The Kruskal's algorithm is used for finding the minimum cost spanning tree.

# Practical No. 8

**Title:** Write a program to implement Prims algorithm.

**Problem Definition:** The problem is to find minimum spanning tree using Prims's algorithm in such way that all the vertices should be visited once and cost of visited edge is less as possible.

**Software Requirement: Turbo C**

**Algorithm:**

**function Prim(G=<N,A>:graph, length: A→$R^+$): set of edges**

{Initialization}

T← Ǿ

B← {an arbitrary member of N}

while B≠N do

find e={u, v} of minimum length such that

u ε B and v ε N/B

T←T U {e}

B←B U {v}

return T

**Code:**

```
#include<stdio.h>
int a,b,u,v,n,i,j,ne=1;
int visited[10]={0},min,mincost=0,cost[10][10];
void main()
{
```

```
clrscr();
printf("\n Enter the number of nodes:");
scanf("%d",&n);
printf("\n Enter the adjacency matrix:\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
{
scanf("%d",&cost[i][j]);
if(cost[i][j]==0)
cost[i][j]=999;
}
visited[1]=1;
printf("\n");
while(ne<n)
{
for(i=1,min=999;i<=n;i++)
for(j=1;j<=n;j++)
if(cost[i][j]<min)
if(visited[i]!=0)
{
min=cost[i][j];
a=u=i;
b=v=j;
}
if(visited[u]==0 || visited[v]==0)
```

```
{
printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);
mincost+=min;
visited[b]=1;
}
cost[a][b]=cost[b][a]=999;
}
printf("\n Minimun cost=%d",mincost);
}
```

**Input Specification:**

Enter the number of nodes: 4

Enter the adjacency matrix:

0 20 10 50

20 0 60 999

10 60 0 40

50 999 40 0

**Output Specification:**

Edge 1: <1 3> cost: 10

Edge 2: <1 2> cost: 20

Edge 3: <3 4> cost: 40

Minimum cost=70

**Conclusion: -**

The prim's algorithm is used for finding the minimum cost spanning tree. In prim's algorithm, on the other hand, the minimum spanning tree grows in a natural way, starting from an arbitrary root. At each stage we add a new branch to the tree already constructed; the algorithm stops when all the nodes have been reached.

# Practical No. 9

**Title:** Write a program to study & implement concept Backtracking using Depth First Search.

**Problem Definition:** We have to visit each node of the tree by selecting depth node as far as possible before taking backtracking.

**Software Requirement: Turbo C**

**Algorithm:**

**procedure dfsearch(G)**

for each v ε N do mark[v]← not visited

for each v ε N do

if mark [v] ≠ visited then dfs(v)

**procedure dfs(v)**

{node vhas not previously been visited}

mark[v]← visited

for each node w adjacent to v do

if mark[v]≠ vosited then dfs(w)

**Code:**

```
#include<stdio.h>
int a[20][20],reach[20],n;
void dfs(int v)
{
int i;
reach[v]=1;
for(i=1;i<=n;i++)
```

```
if(a[v][i] && !reach[i])
{
printf("\n %d->%d",v,i);
dfs(i);
}
}
void main()
{
int i,j,count=0;
printf("\n Enter number of vertices:");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
reach[i]=0;
for(j=1;j<=n;j++)
a[i][j]=0;
}
printf("\n Enter the adjacency matrix:\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
scanf("%d",&a[i][j]);
dfs(1);
printf("\n");
for(i=1;i<=n;i++)
{
```

```
if(reach[i])
count++;
}
if(count==n)
printf("\n Graph is connected");
else
printf("\n Graph is not connected");
}
```

**Input Specification:** Enter number of vertices: 4

Enter the adjacency matrix:

```
0   1  1  1
0   0  0  1
0   0  0  0
0     0 1  0
```

1-> 2

2-> 4

4 -> 3

**Output Specification:**

Graph is connected.

**Conclusion:**

Thus we have studied and implement concept backtracking using Depth First Search. Where in Depth First Search algorithm after selecting the root node it will always go for the left child first and then the child of that particular node and if any child is no available means we reach

to the leaf node then it will backtrack to its parent node and then it will again check for any other child. And likewise all the nodes will get traversed.

# Practical No. 10

**Title:** Write a program to study & implement concept Backtracking using Breadth First Search.

**Problem Definition:** We have to visit each node of the tree and their neighbor child node as far as possible before taking backtracking.

**Software Requirement: Turbo C**

**Algorithm:**

Breadth-First-Search(Graph, root):

```
    for each node n in Graph:
        n.distance = INFINITY
        n.parent = NIL

    create empty queue Q

    root.distance = 0
    Q.enqueue(root)

    while Q is not empty:

        current = Q.dequeue()

        for each node n that is adjacent to current:
            if n.distance == INFINITY:
                n.distance = current.distance + 1
                n.parent = current
             Q.enqueue(n)
```

**Code:**

```
#include<stdio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
void bfs(int v)
{
 for(i=1;i<=n;i++)
  if(a[v][i] && !visited[i])
```

```
   q[++r]=i;
 if(f<=r)
 {
  visited[q[f]]=1;
  bfs(q[f++]);
 }
}
void main()
{
 int v;
 printf("\n Enter the number of vertices:");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
  q[i]=0;
  visited[i]=0;
 }
 printf("\n Enter graph data in matrix form:\n");
 for(i=1;i<=n;i++)
  for(j=1;j<=n;j++)
   {
scanf("%d",&a[i][j]);
}
 printf("\n Enter the starting vertex:");
 scanf("%d",&v);
 bfs(v);
 printf("\n The node which are reachable are:\n");
 for(i=1;i<=n;i++)
  if(visited[i])
   printf("%d\t",i);
  else
   printf("\n Bfs is not possible");

}
```

**Input Specification:**

Enter number of vertices: 4

Enter the adjacency matrix:

0 1 1 1

1 0 1 1

1 1 0 1

1 1 1 0

**Output Specification:**

The nodes are reachable are:

1 2 3 4

**Conclusion: -**

Breadth First Search algorithm selects the root node first and it will always go for the left child first and then the neighbor child of that particular node and then its neighbor child with same level and then to the next level. If neighboring node is not available means we reach to the leaf node then it will backtrack to its parent node and then it will again check for any other child. And likewise all the nodes will get traversed.