1)Array.

```
#include <iostream>
using namespace std;
int main()
{
int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
//traversing array
for (int i = 0; i < 5; i++)
{
cout<<arr[i]<<"\n";
}
}
```

#array delete

```
#include <iostream>

const int MAX_SIZE = 100; // Maximum size of the array

int main() {
   int arr[MAX_SIZE] = {1, 2, 3, 4, 5};
   int currentSize = 5; // Number of elements in the array

   int deleteIndex = 2; // Index of the element to delete

   // Shift elements to the left to remove the specified element
   for (int i = deleteIndex; i < currentSize - 1; i++) {
      arr[i] = arr[i + 1];
   }

   // Update the size of the array
   currentSize--;

   // Display the updated array
   for (int i = 0; i < currentSize; i++) {
      std::cout << arr[i] << " ";
   }
   std::cout << std::endl;

   return 0;
}
```

2) Multidimensional Arrays, Matrices.

//Multidimensional Arrays

```
#include <iostream>
using namespace std;
int main()
{
int test[3][3]; //declaration of 2D array
test[0][0]=5; //initialization
test[0][1]=10;
test[1][1]=15;
test[1][2]=20;
test[2][0]=30;
test[2][2]=10;
//traversal
for(int i = 0; i < 3; ++i)
{
for(int j = 0; j < 3; ++j)
{
cout<< test[i][j]<<" ";
}
cout<<"\n"; //new line at each row
}
return 0;
}
```

//Matrices.

```
#include <iostream>
using namespace std; int main()
{
int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;
cout<<"enter the number of row=";
cin>>r;
cout<<"enter the number of column=";
cin>>c;
cout<<"enter the first matrix element=\n";
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
cin>>a[i][j];
}
}
cout<<"enter the second matrix element=\n";
for(i=0;i<r;i++)
```

```
{

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

{
cin>>b[i][j];
}
}
cout<<"multiply of the matrix=\n";
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
mul[i][j]=0;
for(k=0;k<c;k++)
{
mul[i][j]+=a[i][k]*b[k][j];
}
}
}
//for printing result
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{ cout<<mul[i][j]<<" ";
}
cout<<"\n";
}
return 0;
}
```

3) stack operation

```
#include <iostream>
using namespace std;
int stack[100], n=100, top=-1;
void push(int val) {
  if(top>=n-1)
  cout<<"Stack Overflow"<<endl;
  else {
    top++;
    stack[top]=val;
  }
}
void pop() {
  if(top<=-1)
  cout<<"Stack Underflow"<<endl;
  else {
    cout<<"The popped element is "<< stack[top] <<endl;
    top--;
  }
}
void display() {
  if(top>=0) {
    cout<<"Stack elements are:";
    for(int i=top; i>=0; i--)
    cout<<stack[i]<<" ";
    cout<<endl;
  } else
  cout<<"Stack is empty";
}
int main() {
  int ch, val;
  cout<<"1) Push in stack"<<endl;
  cout<<"2) Pop from stack"<<endl;
  cout<<"3) Display stack"<<endl;
  cout<<"4) Exit"<<endl;
  do {
    cout<<"Enter choice: "<<endl;
    cin>>ch;
    switch(ch) {
```

```
 case 1: {
           cout<<"Enter value to be pushed:"<<endl;
           cin>>val;
           push(val);
           break;
        }
        case 2: {
           pop();
           break;
        }
        case 3: {
           display();
           break;
        }
        case 4: {
           cout<<"Exit"<<endl;
           break;
        }
        default: {
           cout<<"Invalid Choice"<<endl;
        }
     }
  }while(ch!=4);
  return 0;
}

//polish notation
#include <iostream>
#include <stack>
using namespace std;
int priority (char alpha){
 if(alpha == '+' || alpha =='-')
 return 1;
 if(alpha == '*' || alpha =='/')
 return 2;
 if(alpha == '^')
 return 3;
 return 0;
}
```

```
string convert(string infix)
{
 int i = 0;
 string postfix = "";
 // using inbuilt stack< > from C++ stack library
 stack <int>s;
 while(infix[i]!='\0')
 {
 // if operand add to the postfix expression
 if(infix[i]>='a' && infix[i]<='z'|| infix[i]>='A'&& infix[i]<='Z')

 {
 postfix += infix[i];
 i++;
 }
 // if opening bracket then push the stack
 else if(infix[i]=='(')
 {
 s.push(infix[i]);
 i++;
 }
 // if closing bracket encounted then keep popping from stack until
 // closing a pair opening bracket is not encountered
 else if(infix[i]==')')
 {
 while(s.top()!='('){
 postfix += s.top();
 s.pop();
 }
 s.pop();
 i++;
 }
 else
 {
 while (!s.empty() && priority(infix[i]) <= priority(s.top())){
 postfix += s.top();
 s.pop();
 }
 s.push(infix[i]);
 i++;
 }
```

```
}

while(!s.empty()){
postfix += s.top();
s.pop();
}
cout << "Postfix is : " << postfix; //it will print postfix conversion
return postfix;
}
int main()
{
string infix = "((a+(b*c))-d)";
string postfix;
postfix = convert(infix);

return 0;
}
```

4) queues operation

```
#include <iostream>
using namespace std;
int queue[100], n = 100, front = - 1, rear = - 1;
void Insert() {
int val;
if (rear == n - 1)
cout<<"Queue Overflow"<<endl;
else {
if (front == - 1)
front = 0;
cout<<"Insert the element in queue : "<<endl;
cin>>val;
rear++;
queue[rear] = val;
} }
void Delete() {
if (front == - 1 || front > rear) {
cout<<"Queue Underflow ";
return ;
} else {
cout<<"Element deleted from queue is : "<< queue[front] <<endl;
front++;;
```

```
 } }

void Display() {
 if (front == - 1)
 cout<<"Queue is empty"<<endl;
 else {
 cout<<"Queue elements are : ";
 for (int i = front; i <= rear; i++)
 cout<<queue[i]<<" ";
 cout<<endl;
 } }
int main() {
 int ch;
 cout<<"1) Insert element to queue"<<endl;
 cout<<"2) Delete element from queue"<<endl;
 cout<<"3) Display all the elements of queue"<<endl;
 cout<<"4) Exit"<<endl;
 do {
 cout<<"Enter your choice : "<<endl;
 cin>>ch;
 switch (ch) {
 case 1: Insert();
 break;
 case 2: Delete();
 break;
 case 3: Display();
 break;
 case 4: cout<<"Exit"<<endl;
 break;
 default: cout<<"Invalid choice"<<endl;
 }
 } while(ch!=4);
 return 0;
}
```

5) Deque

```
#include <deque>
#include <iostream>
using namespace std;
void showdq(deque<int> g)
{
```

```
deque<int>::iterator it;

for (it = g.begin(); it != g.end(); ++it)
cout << '\t' << *it;
cout << '\n';
}
int main()
{
deque<int> gquiz;
gquiz.push_back(10);
gquiz.push_front(20);
gquiz.push_back(30);
gquiz.push_front(15);
cout << "The deque gquiz is : ";
showdq(gquiz);
cout << "\ngquiz.size() : " << gquiz.size();
cout << "\ngquiz.max_size() : " << gquiz.max_size();
cout << "\ngquiz.at(2) : " << gquiz.at(2);
cout << "\ngquiz.front() : " << gquiz.front();
cout << "\ngquiz.back() : " << gquiz.back();
cout << "\ngquiz.pop_front() : ";
gquiz.pop_front();
showdq(gquiz);
cout << "\ngquiz.pop_back() : ";
gquiz.pop_back();
showdq(gquiz);
return 0;
}
```

6) //Linked list

```
#include <iostream>
using namespace std;

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

struct Node* head = NULL;

void insert(int new_data) {
```

```
  struct Node* new_node = new Node(); // Use 'new' to allocate memory for a
new node
  new_node->data = new_data;
  new_node->next = head;
  head = new_node;
}

void display() {
  struct Node* ptr = head;
  while (ptr != NULL) {
    cout << ptr->data << " ";
    ptr = ptr->next;
  }
}

int main() {
  insert(3);
  insert(1);
  insert(7);
  insert(2);
  insert(9);

  cout << "The linked list is: ";
  display();

  return 0;
}

//circular linked list
#include <iostream>
using namespace std;

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

struct Node* head = NULL;

void insert(int newdata) {
   struct Node* newnode = new Node();
```

```
    newnode->data = newdata;
    if (head == NULL) {
        newnode->next = newnode; // Point to itself in case of the first node
        head = newnode;
    } else {
        struct Node* last = head;
        while (last->next != head)
            last = last->next;
        last->next = newnode;
        newnode->next = head; // Point back to the head to make it circular
    }
}

void display() {
    if (head == NULL) {
        cout << "The circular linked list is empty.\n";
        return;
    }
    struct Node* ptr = head;
    do {
        cout << ptr->data << " ";
        ptr = ptr->next;
    } while (ptr != head);
    cout << endl;
}

int main() {
    insert(3);
    insert(1);
    insert(7);
    insert(2);
    insert(9);

    cout << "The circular linked list is: ";
    display();

    return 0;
}

//doubly linked list
#include <iostream>
```

```
using namespace std;

struct Node {
   int data;
   struct Node* prev;
   struct Node* next;
};

struct Node* head = NULL;

void insert(int newdata) {
   struct Node* newnode = new struct Node; // Use 'new' for memory allocation
   newnode->data = newdata;
   newnode->prev = NULL;
   newnode->next = head;
   if (head != NULL)
      head->prev = newnode;
   head = newnode;
}

void display() {
   struct Node* ptr = head;
   while (ptr != NULL) {
      cout << ptr->data << " ";
      ptr = ptr->next;
   }
}

void freeMemory() {
   struct Node* current = head;
   while (current != NULL) {
      struct Node* next = current->next;
      delete current; // Use 'delete' to free memory
      current = next;
   }
}

int main() {
   insert(3);
   insert(1);
   insert(7);
```

```
    insert(2);

    insert(9);
    cout << "The doubly linked list is: ";
    display();

    freeMemory(); // Free the allocated memory before exiting

    return 0;
}
```

7)**Polynomial Addition/Subtraction.**

```
#include <iostream>
using namespace std;
// Node class
class Node {
public:
int coeff, power;
Node* next;
// Constructor of Node
Node(int coeff, int power)
{
this->coeff = coeff;
this->power = power;
this->next = NULL;
}
};
// Function to add polynomials
void addPolynomials(Node* head1, Node* head2)
{
// Checking if our list is empty
if (head1 == NULL && head2 == NULL)
return;
// List contains elmements
else if (head1->power == head2->power) {
cout << " " << head1->coeff + head2->coeff << "x^"
<< head1->power << " ";
addPolynomials(head1->next, head2->next);
}
else if (head1->power > head2->power) {
cout << " " << head1->coeff << "x^" << head1->power
```

```
<< " ";
addPolynomials(head1->next, head2);
}
else {
cout << " " << head2->coeff << "x^" << head2->power
<< " ";
addPolynomials(head1, head2->next);
} }
void insert(Node* head, int coeff, int power)
{
Node* new_node = new Node(coeff, power);
while (head->next != NULL) {
head = head->next;
}
head->next = new_node;
}
void printList(Node* head)
{
cout << "Linked List" << endl;
while (head != NULL) {
cout << " " << head->coeff << "x"
<< "^" << head->power;
head = head->next;
} }
// Main function
int main()
{
Node* head = new Node(5, 2);
insert(head, 4, 1);
Node* head2 = new Node(6, 2);
insert(head2, 4, 1);
printList(head);
cout << endl;
printList(head2);
cout << endl << "Addition:" << endl;
addPolynomials(head, head2);
return 0;
}
```

8)Binary Search Tree.

```
#include <stdio.h>
#include <stdlib.h>
struct node {
int key;
struct node *left, *right;
};
// A utility function to create a new BST node
struct node* newNode(int item)
{
struct node* temp
= (struct node*)malloc(sizeof(struct node));
temp->key = item;
temp->left = temp->right = NULL;
return temp;
}
// A utility function to do inorder traversal of BST
void inorder(struct node* root)
{
if (root != NULL) {
inorder(root->left);
printf("%d ", root->key);
inorder(root->right);
} }
// A utility function to insert
// a new node with given key in BST
struct node* insert(struct node* node, int key)
{
// If the tree is empty, return a new node
if (node == NULL)
return newNode(key);
// Otherwise, recur down the tree
if (key < node->key)
node->left = insert(node->left, key);
else if (key > node->key)
node->right = insert(node->right, key);
// Return the (unchanged) node pointer
return node;
}
```

```
// Driver Code
int main()
{
struct node* root = NULL;
root = insert(root, 50);
insert(root, 30);
insert(root, 20);
insert(root, 40);
insert(root, 70);
insert(root, 60);
insert(root, 80);
// Print inoder traversal of the BST
inorder(root);
return 0;
}
```

9) //in-order traversal

```
// C++ program for different tree traversals
#include <bits/stdc++.h>
using namespace std;
/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node {
int data;
struct Node *left, *right;
};
// Utility function to create a new tree node
Node* newNode(int data)
{
Node* temp = new Node;
temp->data = data;
temp->left = temp->right = NULL;
return temp;
}
/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node)
{
if (node == NULL)
return;
/* first recur on left child */
printInorder(node->left);
```

```
/* then print the data of node */
cout << node->data << " ";
/* now recur on right child */
printInorder(node->right);
}
/* Driver code*/
int main()
{
struct Node* root = newNode(1);
root->left = newNode(2);
root->right = newNode(3);
root->left->left = newNode(4);
root->left->right = newNode(5);
// Function call
cout << "\nInorder traversal of binary tree is \n";
printInorder(root);
return 0;
}
```

//post-order

```
// C++ program for different tree traversals
#include <bits/stdc++.h>
using namespace std;
/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node {
int data;
struct Node *left, *right;
};
// Utility function to create a new tree node
Node* newNode(int data)
{
Node* temp = new Node;
temp->data = data;
temp->left = temp->right = NULL;
return temp;
}
/* Given a binary tree, print its nodes according to the
"bottom-up" postorder traversal. */
void printPostorder(struct Node* node)
{
```

```
if (node == NULL)
return;
// first recur on left subtree
printPostorder(node->left);
// then recur on right subtree
printPostorder(node->right);
// now deal with the node
cout << node->data << " ";
}
/* Driver code*/
int main()
{
struct Node* root = newNode(1);
root->left = newNode(2);
root->right = newNode(3);
root->left->left = newNode(4);
root->left->right = newNode(5);
// Function call
cout << "\nPostorder traversal of binary tree is \n";
printPostorder(root);
return 0;
}
```

//Pre-order:

```
// C++ program for different tree traversals
#include <bits/stdc++.h>
using namespace std;
/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node {
int data;
struct Node *left, *right;
};
// Utility function to create a new tree node
Node* newNode(int data)
{
Node* temp = new Node;
temp->data = data;
temp->left = temp->right = NULL;
return temp;
}
```

```
/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
if (node == NULL)
return;
/* first print data of node */
cout << node->data << " ";
/* then recur on left subtree */
printPreorder(node->left);
/* now recur on right subtree */
printPreorder(node->right);
}
/* Driver code*/
int main()
{
struct Node* root = newNode(1);
root->left = newNode(2);
root->right = newNode(3);
root->left->left = newNode(4);
root->left->right = newNode(5);
// Function call
cout << "\nPreorder traversal of binary tree is \n";
printPreorder(root);
return 0;
}
```

10) heap tree

```
#include <iostream>
#include <conio.h>
using namespace std;
void min_heap(int *a, int m, int n){
 int j, t;
 t= a[m];
 j = 2 * m;
 while (j <= n) {
 if (j < n && a[j+1] < a[j])
 j = j + 1;
 if (t < a[j])
 break;
 else if (t >= a[j]) {
 a[j/2] = a[j];
```

```
 j = 2 * j;
 }
 }
 a[j/2] = t;
 return;
}
void build_minheap(int *a, int n) {
 int k;
 for(k = n/2; k >= 1; k--) {
 min_heap(a,k,n);
 } }
int main() {
 int n, i;
 cout<<"enter no of elements of array\n";
 cin>>n;
 int a[30];
 for (i = 1; i <= n; i++) {
 cout<<"enter element"<<" "<<(i)<<endl;
 cin>>a[i];
 }
 build_minheap(a, n);
 cout<<"Min Heap\n";
 for (i = 1; i <= n; i++) {
 cout<<a[i]<<endl;
 }
 getch();
}
```

11) depth first search

```
// C++ program to print DFS traversal from
// a given vertex in a given graph
#include <bits/stdc++.h>
using namespace std;
// Graph class represents a directed graph
// using adjacency list representation
class Graph {
public:
map<int, bool> visited;
map<int, list<int> > adj;
// function to add an edge to graph
void addEdge(int v, int w);
// DFS traversal of the vertices
// reachable from v
void DFS(int v);
```

```
};
void Graph::addEdge(int v, int w)
{
adj[v].push_back(w); // Add w to v’s list.
}
void Graph::DFS(int v)
{
// Mark the current node as visited and
// print it
visited[v] = true;
cout << v << " ";// Recur for all the vertices adjacent
// to this vertex
list<int>::iterator i;
for (i = adj[v].begin(); i != adj[v].end(); ++i)
if (!visited[*i])
DFS(*i);
}
// Driver's code
int main()
{
// Create a graph given in the above diagram
Graph g;
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
g.addEdge(2, 0);
g.addEdge(2, 3);
g.addEdge(3, 3);
cout << "Following is Depth First Traversal"
" (starting from vertex 2) \n";
// Function call
g.DFS(2);
return 0;
}
```

12)   : Breadth First Traversal.

```
// BFS algorithm in C++

#include <iostream>
#include <list>

using namespace std;

class Graph {
  int numVertices;
  list<int>* adjLists;
  bool* visited;

   public:
```

```
  Graph(int vertices);
  void addEdge(int src, int dest);
  void BFS(int startVertex);
};

// Create a graph with given vertices,
// and maintain an adjacency list
Graph::Graph(int vertices) {
  numVertices = vertices;
  adjLists = new list<int>[vertices];
}

// Add edges to the graph
void Graph::addEdge(int src, int dest) {
  adjLists[src].push_back(dest);
  adjLists[dest].push_back(src);
}

// BFS algorithm
void Graph::BFS(int startVertex) {
  visited = new bool[numVertices];
  for (int i = 0; i < numVertices; i++)
    visited[i] = false;

  list<int> queue;

  visited[startVertex] = true;
  queue.push_back(startVertex);

  list<int>::iterator i;

  while (!queue.empty()) {
    int currVertex = queue.front();
    cout << "Visited " << currVertex << " ";
    queue.pop_front();

    for (i = adjLists[currVertex].begin(); i != adjLists[currVertex].end(); ++i) {
      int adjVertex = *i;
      if (!visited[adjVertex]) {
        visited[adjVertex] = true;
        queue.push_back(adjVertex);
      }
    }
  }
}
```

```
int main() {
  Graph g(4);
  g.addEdge(0, 1);
  g.addEdge(0, 2);
  g.addEdge(1, 2);
  g.addEdge(2, 0);
  g.addEdge(2, 3);
  g.addEdge(3, 3);

  g.BFS(2);

  return 0;
}
```

13) Obtaining shortest Path(Dijkstra and Floyd-Warshall).

//Dijkstra

```
// A C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
#include <limits.h>
#include <stdio.h>
// Number of vertices in the graph
#define V 9
// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
// Initialize min value
int min = INT_MAX, min_index;
for (int v = 0; v < V; v++)
if (sptSet[v] == false && dist[v] <= min)
min = dist[v], min_index = v;
return min_index;
}
// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
printf("Vertex Distance from Source\n");
for (int i = 0; i < V; i++)
printf("%d \t\t %d\n", i, dist[i]);
}// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
int dist[V]; // The output array. dist[i] will hold the shortest
// distance from src to i
bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest
// path tree or shortest distance from src to i is finalized
// Initialize all distances as INFINITE and stpSet[] as false
```

```
for (int i = 0; i < V; i++)
dist[i] = INT_MAX, sptSet[i] = false;
// Distance of source vertex from itself is always 0
dist[src] = 0;
// Find shortest path for all vertices
for (int count = 0; count < V - 1; count++) {
// Pick the minimum distance vertex from the set of vertices not
// yet processed. u is always equal to src in the first iteration.
int u = minDistance(dist, sptSet);
// Mark the picked vertex as processed
sptSet[u] = true;
// Update dist value of the adjacent vertices of the picked vertex.
for (int v = 0; v < V; v++)
// Update dist[v] only if is not in sptSet, there is an edge from
// u to v, and total weight of path from src to v through u is
// smaller than current value of dist[v]
if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
&& dist[u] + graph[u][v] < dist[v])
dist[v] = dist[u] + graph[u][v];
}
// print the constructed distance array
printSolution(dist, V);
}// driver program to test above function
int main()
{
/* Let us create the example graph discussed above */
int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
{ 4, 0, 8, 0, 0, 0, 0, 11, 0 },
{ 0, 8, 0, 7, 0, 4, 0, 0, 2 },
{ 0, 0, 7, 0, 9, 14, 0, 0, 0 },
{ 0, 0, 0, 9, 0, 10, 0, 0, 0 },
{ 0, 0, 4, 14, 10, 0, 2, 0, 0 },
{ 0, 0, 0, 0, 0, 2, 0, 1, 6 },
{ 8, 11, 0, 0, 0, 0, 1, 0, 7 },
{ 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
dijkstra(graph, 0);
return 0;
}
```

//Floyd-Warshall:

```
// C++ Program for Floyd Warshall Algorithm
#include <bits/stdc++.h>
using namespace std;
// Number of vertices in the graph
#define V 4
/* Define Infinite as a large enough
value.This value will be used for
vertices not connected to each other */
#define INF 99999
```

```
// A function to print the solution matrix
void printSolution(int dist[][V]);
// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm
void floydWarshall(int dist[][V])
{
int i, j, k;
/* Add all vertices one by one to
the set of intermediate vertices.
---> Before start of an iteration,
we have shortest distances between all
pairs of vertices such that the
shortest distances consider only the
vertices in set {0, 1, 2, .. k-1} as
intermediate vertices.
----> After the end of an iteration,
vertex no. k is added to the set of
intermediate vertices and the set becomes {0, 1, 2, ..
k} */
for (k = 0; k < V; k++) {
// Pick all vertices as source one by one
for (i = 0; i < V; i++) {
// Pick all vertices as destination for the
// above picked source
for (j = 0; j < V; j++) {
// If vertex k is on the shortest path from
// i to j, then update the value of
// dist[i][j]
if (dist[i][j] > (dist[i][k] + dist[k][j])
&& (dist[k][j] != INF
&& dist[i][k] != INF))
dist[i][j] = dist[i][k] + dist[k][j];
} } }
// Print the shortest distance matrix
printSolution(dist);
}
/* A utility function to print solution */
void printSolution(int dist[][V])
{
cout << "The following matrix shows the shortest "
"distances"
" between every pair of vertices \n";
for (int i = 0; i < V; i++) {
for (int j = 0; j < V; j++) {
if (dist[i][j] == INF)
```

```
cout << "INF"
<< " ";
else
cout << dist[i][j] << " ";
}
cout << endl;
} }
// Driver's code
int main()
{
/* Let us create the following weighted graph
10
(0)------->(3)
| /|\
5 | || | 1
\|/ |
(1)------->(2)
3 */
int graph[V][V] = { { 0, 5, INF, 10 },
{ INF, 0, 3, INF },
{ INF, INF, 0, 1 },
{ INF, INF, INF, 0 } };
// Function call
floydWarshall(graph);
return 0;
}
```

14)Minimum spanning tree(Kruskal and Prim).

```
//Kruskal:
// C++ program for Kruskal's algorithm to find Minimum
// Spanning Tree of a given connected, undirected and
// weighted graph
#include<bits/stdc++.h>
using namespace std;
// Creating shortcut for an integer pair
typedef pair<int, int> iPair;
// Structure to represent a graph
struct Graph
{
int V, E;
vector< pair<int, iPair> > edges;
// Constructor
Graph(int V, int E)
```

```
{
this->V = V;
this->E = E;
}
// Utility function to add an edge
void addEdge(int u, int v, int w)
{
edges.push_back({w, {u, v}});
}
// Function to find MST using Kruskal's
// MST algorithm
int kruskalMST();
};
// To represent Disjoint Sets
struct DisjointSets
{
int *parent, *rnk;
int n;
// Constructor.
DisjointSets(int n)
{
// Allocate memory
this->n = n;
parent = new int[n+1];
rnk = new int[n+1];
// Initially, all vertices are in
// different sets and have rank 0.
for (int i = 0; i <= n; i++)
{
rnk[i] = 0;
//every element is parent of itself
parent[i] = i;
} }
// Find the parent of a node 'u'
// Path Compression
int find(int u)
{
/* Make the parent of the nodes in the path
from u--> parent[u] point to parent[u] */
if (u != parent[u])
parent[u] = find(parent[u]);
```

```
return parent[u];
}
// Union by rank
void merge(int x, int y)
{
x = find(x), y = find(y);
/* Make tree with smaller height
a subtree of the other tree */
if (rnk[x] > rnk[y])
parent[y] = x;
else // If rnk[x] <= rnk[y]
parent[x] = y;
if (rnk[x] == rnk[y])
rnk[y]++;
}
};
/* Functions returns weight of the MST*/
int Graph::kruskalMST()
{
int mst_wt = 0; // Initialize result
// Sort edges in increasing order on basis of cost
sort(edges.begin(), edges.end());
// Create disjoint sets
DisjointSets ds(V);
// Iterate through all sorted edges
vector< pair<int, iPair> >::iterator it;
for (it=edges.begin(); it!=edges.end(); it++)
{
int u = it->second.first;
int v = it->second.second;
int set_u = ds.find(u);
int set_v = ds.find(v);
// Check if the selected edge is creating
// a cycle or not (Cycle is created if u
// and v belong to same set)
if (set_u != set_v)
{
// Current edge will be in the MST
// so print it
cout << u << " - " << v << endl;
// Update MST weight
```

```
mst_wt += it->first;
// Merge two sets
ds.merge(set_u, set_v);
} }
return mst_wt;
}
// Driver program to test above functions
int main()
{
/* Let us create above shown weighted
and undirected graph */
int V = 9, E = 14;
Graph g(V, E);
// making above shown graph
g.addEdge(0, 1, 4);
g.addEdge(0, 7, 8);
g.addEdge(1, 2, 8);
g.addEdge(1, 7, 11);
g.addEdge(2, 3, 7);
g.addEdge(2, 8, 2);
g.addEdge(2, 5, 4);
g.addEdge(3, 4, 9);
g.addEdge(3, 5, 14);
g.addEdge(4, 5, 10);
g.addEdge(5, 6, 2);
g.addEdge(6, 7, 1);
g.addEdge(6, 8, 6);
g.addEdge(7, 8, 7);
cout << "Edges of MST are \n";
int mst_wt = g.kruskalMST();
cout << "\nWeight of MST is " << mst_wt;
return 0;
}
```

//Prim

```
// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <bits/stdc++.h>
using namespace std;
// Number of vertices in the graph
```

```
#define V 5
// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
// Initialize min value
int min = INT_MAX, min_index;
for (int v = 0; v < V; v++)
if (mstSet[v] == false && key[v] < min)
min = key[v], min_index = v;
return min_index;
}
// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
cout << "Edge \tWeight\n";
for (int i = 1; i < V; i++)
cout << parent[i] << " - " << i << " \t"
<< graph[i][parent[i]] << " \n";
}
// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
// Array to store constructed MST
int parent[V];
// Key values used to pick minimum weight edge in cut
int key[V];
// To represent set of vertices included in MST
bool mstSet[V];
// Initialize all keys as INFINITE
for (int i = 0; i < V; i++)
key[i] = INT_MAX, mstSet[i] = false;
// Always include first 1st vertex in MST.
// Make key 0 so that this vertex is picked as first
// vertex.
key[0] = 0;
// First node is always root of MST
```

```
parent[0] = -1;
// The MST will have V vertices
for (int count = 0; count < V - 1; count++) {
// Pick the minimum key vertex from the
// set of vertices not yet included in MST
int u = minKey(key, mstSet);
// Add the picked vertex to the MST Set
mstSet[u] = true;
// Update key value and parent index of
// the adjacent vertices of the picked vertex.
// Consider only those vertices which are not
// yet included in MST
for (int v = 0; v < V; v++)
// graph[u][v] is non zero only for adjacent
// vertices of m mstSet[v] is false for vertices
// not yet included in MST Update the key only
// if graph[u][v] is smaller than key[v]
if (graph[u][v] && mstSet[v] == false
&& graph[u][v] < key[v])
parent[v] = u, key[v] = graph[u][v];
}
// Print the constructed MST
printMST(parent, graph);
}
// Driver's code
int main()
{
int graph[V][V] = { { 0, 2, 0, 6, 0 },
{ 2, 0, 3, 8, 5 },
{ 0, 3, 0, 0, 7 },
{ 6, 8, 0, 0, 9 },
{ 0, 5, 7, 9, 0 } };
// Print the solution
primMST(graph);
return 0;
}
```

15)Hash Table.

```
#include<iostream>
#include<cstdlib>
#include<string>
```

```
#include<cstdio>
using namespace std;
const int T_S = 200;
class HashTableEntry {
  public:
    int k;
    int v;
    HashTableEntry(int k, int v) {
      this->k= k;
      this->v = v;
    }
};
class HashMapTable {
  private:
    HashTableEntry **t;
  public:
    HashMapTable() {
      t = new HashTableEntry * [T_S];
      for (int i = 0; i< T_S; i++) {
        t[i] = NULL;
      }
    }
    int HashFunc(int k) {
      return k % T_S;
    }
    void Insert(int k, int v) {
      int h = HashFunc(k);
      while (t[h] != NULL && t[h]->k != k) {
        h = HashFunc(h + 1);
      }
      if (t[h] != NULL)
        delete t[h];
      t[h] = new HashTableEntry(k, v);
    }
    int SearchKey(int k) {
      int h = HashFunc(k);
      while (t[h] != NULL && t[h]->k != k) {
        h = HashFunc(h + 1);
      }
      if (t[h] == NULL)
        return -1;
```

```
        else
          return t[h]->v;
      }
      void Remove(int k) {
        int h = HashFunc(k);
        while (t[h] != NULL) {
          if (t[h]->k == k)
            break;
          h = HashFunc(h + 1);
        }
        if (t[h] == NULL) {
          cout<<"No Element found at key "<<k<<endl;
          return;
        } else {
          delete t[h];
        }
        cout<<"Element Deleted"<<endl;
      }
      ~HashMapTable() {
        for (int i = 0; i < T_S; i++) {
          if (t[i] != NULL)
            delete t[i];
            delete[] t;
        }
      }
};
int main() {
  HashMapTable hash;
  int k, v;
  int c;
  while (1) {
    cout<<"1.Insert element into the table"<<endl;
    cout<<"2.Search element from the key"<<endl;
    cout<<"3.Delete element at a key"<<endl;
    cout<<"4.Exit"<<endl;
    cout<<"Enter your choice: ";
    cin>>c;
    switch(c) {
      case 1:
        cout<<"Enter element to be inserted: ";
        cin>>v;
```

```
        cout<<"Enter key at which element to be inserted: ";
        cin>>k;
        hash.Insert(k, v);
      break;
      case 2:
        cout<<"Enter key of the element to be searched: ";
        cin>>k;
        if (hash.SearchKey(k) == -1) {
          cout<<"No element found at key "<<k<<endl;
          continue;
        } else {
          cout<<"Element at key "<<k<<" : ";
          cout<<hash.SearchKey(k)<<endl;
        }
      break;
      case 3:
        cout<<"Enter key of the element to be deleted: ";
        cin>>k;
        hash.Remove(k);
      break;
      case 4:
        exit(1);
      default:
        cout<<"\nEnter correct option\n";
    }
  }
  return 0;
}
```

16)//linear search

```
#include<iostream>
using namespace std;
int main()
{
 int arr[10], i, num, index;
 cout<<"Enter 10 Numbers: ";
 for(i=0; i<10; i++)
 cin>>arr[i];
 cout<<"\nEnter a Number to Search: ";
 cin>>num;
 for(i=0; i<10; i++)
```

```
{
if(arr[i]==num)
{
index = i;
break;
}
}
cout<<"\nFound at Index No."<<index;
cout<<endl;
return 0;
}
```

///Binary search

```
#include<iostream>
using namespace std;
int main()
{
int i, arr[10], num, first, last, middle;
cout<<"Enter 10 Elements (in ascending order): ";
for(i=0; i<10; i++)
cin>>arr[i];
cout<<"\nEnter Element to be Search: ";
cin>>num;
first = 0;
last = 9;
middle = (first+last)/2;
while(first <= last)
{
if(arr[middle]<num)
first = middle+1;
else if(arr[middle]==num)
{
cout<<"\nThe number, "<<num<<" found at Position "<<middle+1;
break;
}
else
last = middle-1;
middle = (first+last)/2;
}
if(first>last)
cout<<"\nThe number, "<<num<<" is not found in given Array";
```

```
 cout<<endl;
 return 0;
}
```

17) bubble sort

```
#include<iostream>
using namespace std;
void swapping(int &a, int &b) { //swap the content of a and b
 int temp;
 temp = a;
 a = b;
 b = temp;
}
void display(int *array, int size) {
 for(int i = 0; i<size; i++)
 cout << array[i] << " ";
 cout << endl;
}
void bubbleSort(int *array, int size) {
 for(int i = 0; i<size; i++) {
 int swaps = 0; //flag to detect any swap is there or not
 for(int j = 0; j<size-i-1; j++) {
 if(array[j] > array[j+1]) { //when the current item is bigger than next
 swapping(array[j], array[j+1]);
 swaps = 1; //set swap flag
 }
 }
 if(!swaps)
 break; // No swap in this pass, so array is sorted
 } }
int main() {
 int n;
 cout << "Enter the number of elements: ";
 cin >> n;
 int arr[n]; //create an array with given number of elements
 cout << "Enter elements:" << endl;
 for(int i = 0; i<n; i++) {
 cin >> arr[i];
 }
 cout << "Array before Sorting: ";
 display(arr, n);
```

```
 bubbleSort(arr, n);
 cout << "Array after Sorting: ";
 display(arr, n);
}
```

18)Selection Sort.

```
#include<iostream>
using namespace std;
void swapping(int &a, int &b) { //swap the content of a and b
 int temp;
 temp = a;
 a = b;
 b = temp;
}
void display(int *array, int size) {
 for(int i = 0; i<size; i++)
 cout << array[i] << " ";
 cout << endl;
}
void selectionSort(int *array, int size) {
 int i, j, imin;
 for(i = 0; i<size-1; i++) {
 imin = i; //get index of minimum data
 for(j = i+1; j<size; j++)
 if(array[j] < array[imin])
 imin = j;
 //placing in correct position
 swap(array[i], array[imin]);
 } }
int main() {
 int n;
 cout << "Enter the number of elements: ";
 cin >> n;
 int arr[n]; //create an array with given number of elements
 cout << "Enter elements:" << endl;
 for(int i = 0; i<n; i++) {
 cin >> arr[i];
 }
 cout << "Array before Sorting: ";
 display(arr, n);
 selectionSort(arr, n);
```

```
 cout << "Array after Sorting: ";
 display(arr, n);
}
```

19)Insertion Sort

```
// C++ program for insertion sort
#include <bits/stdc++.h>
using namespace std;
// Function to sort an array using
// insertion sort
void insertionSort(int arr[], int n)
{
int i, key, j;
for (i = 1; i < n; i++) {
key = arr[i];
j = i - 1;
// Move elements of arr[0..i-1],
// that are greater than key,
// to one position ahead of their
// current position
while (j >= 0 && arr[j] > key) {
arr[j + 1] = arr[j];
j = j - 1;
}
arr[j + 1] = key;
} }
// A utility function to print an array
// of size n
void printArray(int arr[], int n)
{
int i;
for (i = 0; i < n; i++)
cout << arr[i] << " ";
cout << endl;
}
// Driver code
int main()
{
int arr[] = { 12, 11, 13, 5, 6 };
int N = sizeof(arr) / sizeof(arr[0]);
insertionSort(arr, N);
```

```
printArray(arr, N);
return 0;
}
```

20)Radix Sort

```
#include <iostream>
using namespace std;

// Function to get the maximum value from the array.
int getMax(int arr[], int n) {
   int max = arr[0];
   for (int i = 1; i < n; i++)
      if (arr[i] > max)
         max = arr[i];
   return max;
}

// Function to perform Count Sort for the specified digit position (exp).
void countSort(int arr[], int n, int exp) {
   // Output array to store sorted elements.
   int output[n];

   // Count array will keep track of the number of occurrences of each digit (0 to
9).
   int count[10] = {0};

   // Count the occurrences of digits at the specified position (exp) in the input
array.
   for (int i = 0; i < n; i++)
      count[(arr[i] / exp) % 10]++;

   // Calculate the cumulative count for each digit in the count array.
   for (int i = 1; i < 10; i++)
      count[i] += count[i - 1];

   // Place the elements into the output array based on their digit (exp).
   // Decrement the count for each element to handle duplicate elements.
   for (int i = n - 1; i >= 0; i--) {
      output[count[(arr[i] / exp) % 10] - 1] = arr[i];
      count[(arr[i] / exp) % 10]--;
   }
```

```
    // Copy the sorted elements back to the original array.
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

// Radix Sort function to sort the array.
void radixsort(int arr[], int n) {
    // Get the maximum element in the array to determine the number of digits in
it.
    int max = getMax(arr, n);

    // Perform counting sort for each digit position, starting from the least
significant digit (exp=1)
    // and going up to the most significant digit.
    for (int exp = 1; max / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

int main() {
    int n, i;
    cout << "Enter the number of data elements to be sorted: ";
    cin >> n;

    int arr[n];
    for (i = 0; i < n; i++) {
        cout << "Enter element " << i + 1 << ": ";
        cin >> arr[i];
    }

    radixsort(arr, n);

    // Printing the sorted data.
    cout << "Sorted Data: ";
    for (i = 0; i < n; i++)
        cout << "->" << arr[i];

    return 0;
}
```

21)Quick Sort.

```
// C++ Implementation of the Quick Sort Algorithm.
#include <iostream>
using namespace std;
int partition(int arr[], int start, int end)
{
int pivot = arr[start];
int count = 0;
for (int i = start + 1; i <= end; i++) {
if (arr[i] <= pivot)
count++;
}
// Giving pivot element its correct position
int pivotIndex = start + count;
swap(arr[pivotIndex], arr[start]);
// Sorting left and right parts of the pivot element
int i = start, j = end;
while (i < pivotIndex && j > pivotIndex) {
while (arr[i] <= pivot) {
i++;
}
while (arr[j] > pivot) {
j--; }
if (i < pivotIndex && j > pivotIndex) {
swap(arr[i++], arr[j--]);
} }
return pivotIndex;
}
void quickSort(int arr[], int start, int end)
{
// base case
if (start >= end)
return;
// partitioning the array
int p = partition(arr, start, end);
// Sorting the left part
quickSort(arr, start, p - 1);
// Sorting the right part
quickSort(arr, p + 1, end);
}
```

```
int main()
{
int arr[] = { 9, 3, 4, 2, 1, 8 };
int n = 6;
quickSort(arr, 0, n - 1);
for (int i = 0; i < n; i++) {
cout << arr[i] << " ";
}
return 0;
}
```

22)Merge Sort.

```
#include<iostream>
using namespace std;
void swapping(int &a, int &b) { //swap the content of a and b
 int temp;
 temp = a;
 a = b;
 b = temp;
}
void display(int *array, int size) {
 for(int i = 0; i<size; i++)
 cout << array[i] << " ";
 cout << endl;
}
void merge(int *array, int l, int m, int r) {
 int i, j, k, nl, nr;
 //size of left and right sub-arrays
 nl = m-l+1; nr = r-m;
 int larr[nl], rarr[nr];
 //fill left and right sub-arrays
 for(i = 0; i<nl; i++)
 larr[i] = array[l+i];
 for(j = 0; j<nr; j++)
 rarr[j] = array[m+1+j];
 i = 0; j = 0; k = l;
 //marge temp arrays to real array
 while(i < nl && j<nr) {
 if(larr[i] <= rarr[j]) {
 array[k] = larr[i];
 i++;
```

```
}else{
array[k] = rarr[j];
j++;
}
k++;
}
while(i<nl) { //extra element in left array
array[k] = larr[i];
i++; k++;
}
while(j<nr) { //extra element in right array
array[k] = rarr[j];
j++; k++;
} }
void mergeSort(int *array, int l, int r) {
int m;
if(l < r) {
int m = l+(r-l)/2;
// Sort first and second arrays
mergeSort(array, l, m);
mergeSort(array, m+1, r);
merge(array, l, m, r);
} }
int main() {
int n;
cout << "Enter the number of elements: ";
cin >> n;
int arr[n]; //create an array with given number of elements
cout << "Enter elements:" << endl;
for(int i = 0; i<n; i++) {
cin >> arr[i];
}
cout << "Array before Sorting: ";
display(arr, n);
mergeSort(arr, 0, n-1); //(n-1) for last index
cout << "Array after Sorting: ";
display(arr, n);
}
```

23)Heap Sort.

```
#include <iostream>
```

```
using namespace std;

void heapify(int arr[], int n, int i) {
   int temp;
   int largest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;

   if (l < n && arr[l] > arr[largest])
       largest = l;

   if (r < n && arr[r] > arr[largest])
       largest = r;

   if (largest != i) {
       temp = arr[i];
       arr[i] = arr[largest];
       arr[largest] = temp;
       heapify(arr, n, largest);
   }
}

void heapSort(int arr[], int n) {
   int temp;
   for (int i = n / 2 - 1; i >= 0; i--)
       heapify(arr, n, i);

   for (int i = n - 1; i >= 0; i--) {
       temp = arr[0];
       arr[0] = arr[i];
       arr[i] = temp;
       heapify(arr, i, 0);
   }
}

int main() {
   int arr[] = { 20, 7, 1, 54, 10, 15, 90, 23, 77, 25 };
   int n = 10;

   cout << "Given array is: " << endl;
   for (int i = 0; i < n; i++)
```

```
        cout << arr[i] << " ";

    cout << endl;
    heapSort(arr, n);

    cout << "\nSorted array is: " << endl;
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";

    return 0;
}
```