

Apex Institute of Technology

**Program Name: BE CSE (IOT)**

**LAB MANUAL**

Semester : 5th

Course Name : Advanced Programming (Lab)

Course Code : CSP-347

Course Coordinator : Mr. Rahul Rathore

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Scheme Version: 2021** | **Name of Course:** Advanced Programming (Lab) | **L** | **T** | **P** | **S** | **C** |
| **Programs:** BE-CSE (IoT) | **-** | **-** | **4** | **-** | **1** |
| **Total Marks: 100**  **Internal Marks: 40**  **External Marks: 60** | | | | | | |
|  | **Pre-requisite:** Basics of Data Structure, C++ programming | | **Total hours =30** | | | |
| **Course Objective** | | | | | |
| **1** | This subject aims to focuses on Advanced concept of C++ and advanced data structure to students. | | | | | |
| **2** | It focuses on advanced level analysis of algorithm and computational mathematics. | | | | | |
| **Course Outcomes** | | | | | | |
| **1** | Explain the data structure and OOPS concepts using C++. | | | | | |
| **2** | Apply the shortest path and minimum spanning algorithms in computer networks. | | | | | |
| **3** | Examine the complexity of searching and sorting algorithms, and optimization through arrays, linked structures, stacks, queues, trees, and graphs. | | | | | |
| **4.** | Decide and implement an appropriate graph algorithm and hashing function in computer networks for data security. | | | | | |
| **5.** | Construct security encryption and decryption algorithms using computational mathematics and graph algorithm. | | | | | |

**Lab Experiments with CO Mapping**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **S.NO.** | **Experiment** |  | Mapped CO |
|  | **Unit-I** | | | |
|  | 1 | WAP in C++ to find the Kth smallest/largest element in an array using template and C++ STL. |  | CO1 |
|  | 2 | A left rotation operation on a vector of size N shifts each of the array's elements 1 unit to the left. For example, if 2 left rotations are performed on array [1,2,3,4,5], then the array would become[3,4,5,1,2]. Given an vector of n integers and a number, d , perform d left rotations on the array. Then print the updated array as a single line of space-separated integers. Print a single line of n space-separated integers denoting the final state of the array after performing d left rotations |  | CO1 |
|  | 3 | You are given a string containing characters A and B only. Your task is to change it into a string such that there are no matching adjacent characters. To do this, you are allowed to delete zero or more characters in the string. Your task is to find the minimum number of required deletions. For example, given the string s=AABAAB, remove an A at positions 0 and 3 to make s=ABAB in 2 deletions. |  | CO1 |
|  | 4 | **Write a program to maintain a elementary database of employees using files.** |  |  |
|  | **Unit 2** | | | |
|  | 5 | 1. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra’s algorithm.   http://d1gjlxt8vb0knt.cloudfront.net/wp-content/uploads/Fig-11.jpg  b.Compute the transitive closure of a given directed graph using Warshall's algorithm. |  | CO2, CO3 |
|  | 6 | Obtain the Topological ordering of vertices in a given digraph.  http://d1gjlxt8vb0knt.cloudfront.net/wp-content/uploads/graph.png |  | CO3 |
|  | 7 | Implement N Queen's problem using Back Tracking. |  | CO3 |
|  | 8 | Implement Travelling Salesperson problem using Dynamic programming. |  | CO4 |
|  | **Unit 3** | | | |
|  | 9 | Design a quick sort with random pivoting using Lomuto partition scheme. |  | CO4 |
|  | 10 | Demonstrate insert, delete and search in Treap. |  | CO5 |

**MODE OF EVALUATION: The performance of students is evaluated as follows:**

|  |  |  |
| --- | --- | --- |
|  | **Practical** | |
| **Components** | **Continuous Internal Assessment (CAE)** | **Semester End**  **Examination (SEE)** |
| **Marks** | 60 | 40 |
| **Total Marks** | 100 | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Course Outcome** | **PO 1** | **PO 2** | **PO 3** | **PO 4** | **PO 5** | **PO 6** | **PO 7** | **PO 8** | **PO 9** | **PO 10** | **PO 11** | **PO 12** | **PS O 1** | **PS O 2** | **PSO 3** |
| CO1 | - | 3 | - | - | - | - | - | - | - | - | - | - | - | - | - |
| CO2 | - | 3 | - | - | - | - | - | - | - | - | - | - | - | - | - |
| CO3 | - | 3 | 3 | 3 | - | - | - | - | - | - | - | - | - | - | - |
| CO4 | - | 3 | 3 | 3 | - | - | - | - | - | - | - | - | - | - | - |
| CO5 | - | 3 | 3 | 3 | - | 2 | - | - | - | - | - | 2 | - | - | - |

**EXPERIMENT 1**

**Mapped Course Outcomes- CO1**

**CO1:** This subject aims to focuses on Advanced concept of C++ and advanced data structure to students.

**AIM:** Find the Kth smallest/largest element in an array using template

**EXPERIMENT 1-** WAP in C++ to find the Kth smallest/largest element in an array using template and C++ STL.

## Explanation

Given an array and a number k where k is smaller than size of array, we need to find the k’th smallest element in the given array.

Examples:

Input : arr[] = {7, 10, 4, 3, 20, 15}

k = 2

Output : 4

Smallest element is 3. Second smallest

is 4.

Input : arr[] = {7, 10, 4, 3, 3, 15}

k = 2

Output : 4

Even if there are more than one occurrences

of 3, answer should be 4.

Input :arr[] = {7, 10, 4, 3, 20, 15}

k = 4

Output : 10

**Program:** **STL based C++ program to find k-th smallest element.**

#include <bits/stdc++.h>

using namespace std;

int kthSmallest(int arr[], int n, int k)

{

    // Insert all elements into the set

    set<int> s;

    for (int i = 0; i < n; i++)

        s.insert(arr[i]);

      // Traverse set and print k-th element

    auto it = s.begin();

    for (int i = 0; i < k - 1; i++)

        it++;

    return \*it;

}

int main()

{

    int arr[] = { 12, 3, 5, 7, 3, 19 };

    int n = sizeof(arr) / sizeof(arr[0]), k = 2;

    cout << "K'th smallest element is "

         << kthSmallest(arr, n, k);

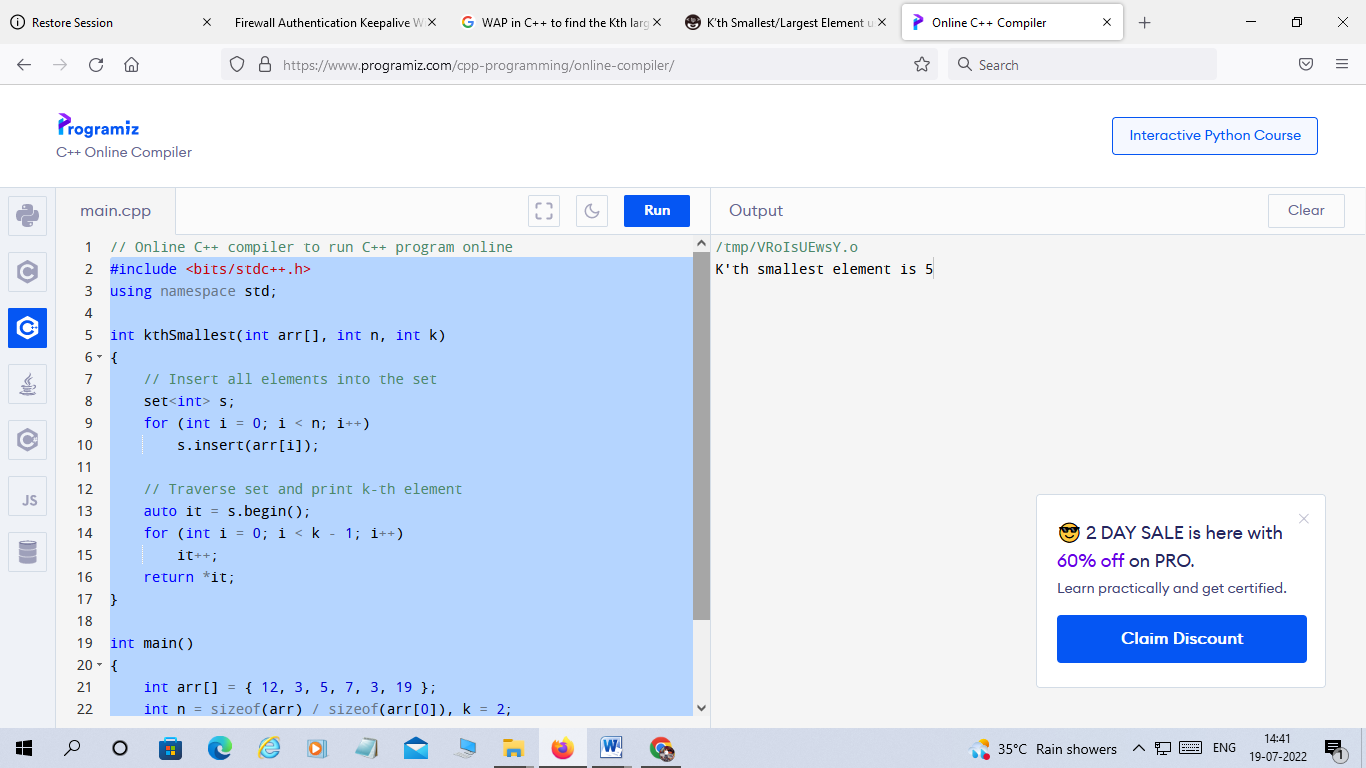
    return 0;

}

Output:

K'th smallest element is 5

**Implementation:-**



**EXPERIMENT -2**

**Mapped Course Outcomes-CO1**

**CO1:** Explain the data structure and OOPS concepts using C++.

**Aim:** A left rotation operation on a vector of size N shifts each of the array's elements 1 unit to the left. For example, if 2 left rotations are performed on array [1,2,3,4,5], then the array would become[3,4,5,1,2]. Given an vector of n integers and a number, d , perform d left rotations on the array. Then print the updated array as a single line of space-separated integers. Print a single line of n space-separated integers denoting the final state of the array after performing d left rotations

**Experiment -2 :**

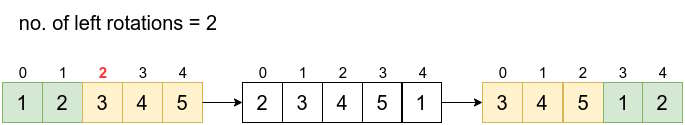
**Problem:**

A left rotation operation on an array of size *n* shifts each of the array’s elements *1* unit to the left. For example, if *2* left rotations are performed on array [1, 2, 3, 4, 5], then the array would become [3, 4, 5, 1, 2].

Given an array of *n* integers and a number, *d*, perform *d* left rotations on the array. Then print the updated array as a single line of space-separated integers.

Read the full problem here : [Left Rotation](https://www.hackerrank.com/challenges/array-left-rotation/copy-from/155235597)

**Solution:**

﻿﻿

This is a very easy problem. In the above figure we see that after two rotations, the element at index 2 is the first element in the new array. Thus after *d* rotations, the element at index *d* is the first element. And all elements after *d*th elements with follow.

for (std::size\_t i = rotations; i < arr.size(); ++i)

{

rotated\_array.push\_back(arr[i]);

}

Now, elements from 0 to *d*-1 indices are added to the new rotated array.

for (std::size\_t i = 0; i < rotations; ++i)

{

rotated\_array.push\_back(arr[i]);

}

**C++ Implementation**

#include <iostream>

#include <vector>

std::vector<int> left\_rotation(std::vector<int>& arr, int rotations)

{

std::vector<int> rotated\_array;

for (std::size\_t i = rotations; i < arr.size(); ++i)

{

rotated\_array.push\_back(arr[i]);

}

for (std::size\_t i = 0; i < rotations; ++i)

{

rotated\_array.push\_back(arr[i]);

}

return rotated\_array;

}

int main()

{

int num\_elements, num\_left\_rotations;

std::cin >> num\_elements;

std::cin >> num\_left\_rotations;

std::vector<int> input\_array(num\_elements);

for (int i = 0; i < num\_elements; ++i)

{

std::cin >> input\_array[i];

}

std::vector<int> result\_array = left\_rotation(input\_array, num\_left\_rotations);

for (int i = 0; i < num\_elements; ++i)

{

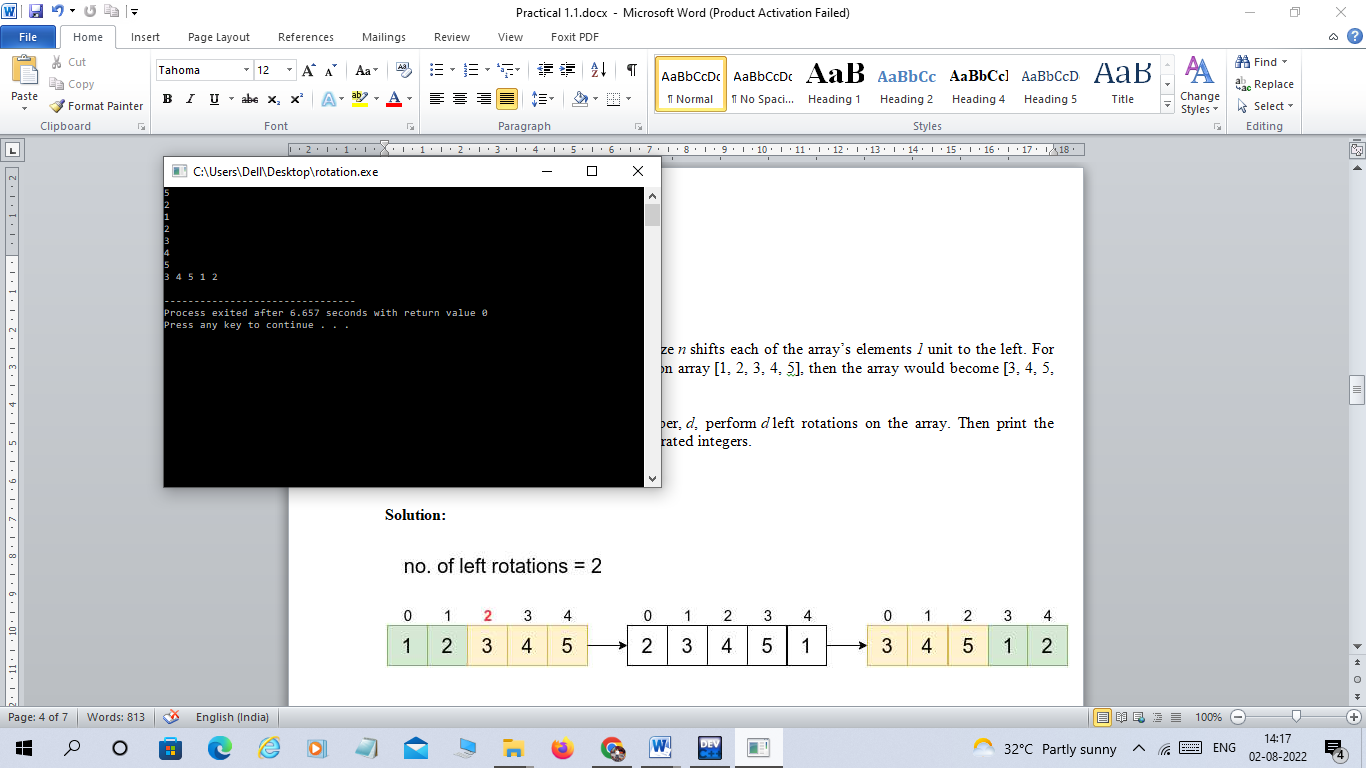
std::cout << result\_array[i] <<" ";

}

std::cout << "\n";

}

**Output**



**EXPERIMENT -3**

**Mapped Course Outcomes-CO1**

**CO1:** Explain the data structure and OOPS concepts using C++.

**Aim:** You are given a string containing characters A and B only. Your task is to change it into a string such that there are no matching adjacent characters. To do this, you are allowed to delete zero or more characters in the string. Your task is to find the minimum number of required deletions. For example, given the string s=AABAAB, remove an A at positions 0 and 3 to make s=ABAB in 2 deletions.

### Experiment -3

**Problem:**

You are given a string containing characters A and B only. Your task is to change it into a string such that there are no matching adjacent characters. To do this, you are allowed to delete zero or more characters in the string. Your task is to find the minimum number of required deletions.

**Example**

s = AABAAB

Remove an A at positions 0 and 3 to make s=ABAB in 2 deletions

**Function Description**

alternating Characters has the following parameter(s):

* string s: a string

**Returns**

* int: the minimum number of deletions required

**Input Format**

The first line contains an integer , the number of queries.

The next lines each contain a string to analyze.

**Constraints**

* 1<=q<=10
* 1<=length of s <= 10^5
* Each string s will consist only of characters A and B.

**Sample Input**

5

AAAA

BBBBB

ABABABAB

BABABA

AAABBB

**Sample Output**

3

4

0

0

4

**Code:**

#include <cmath>﻿

#include <cstdio>

#include <vector>

#include <iostream>

#include <algorithm>

using namespace std;

int main() {

int t;

cin>>t;

while(t--)

{

string s;int c=0,a=0;

cin>>s;

for(int i=1;s[i]!='\0';i++)

{

if((s[i]==65 && s[a]==66)||(s[i]==66 &&s[a]==65))

{

a=i;

// cout<<a<<endl;

}

else{

c++;

//cout<<c<<" "<<i<<endl;

}

}

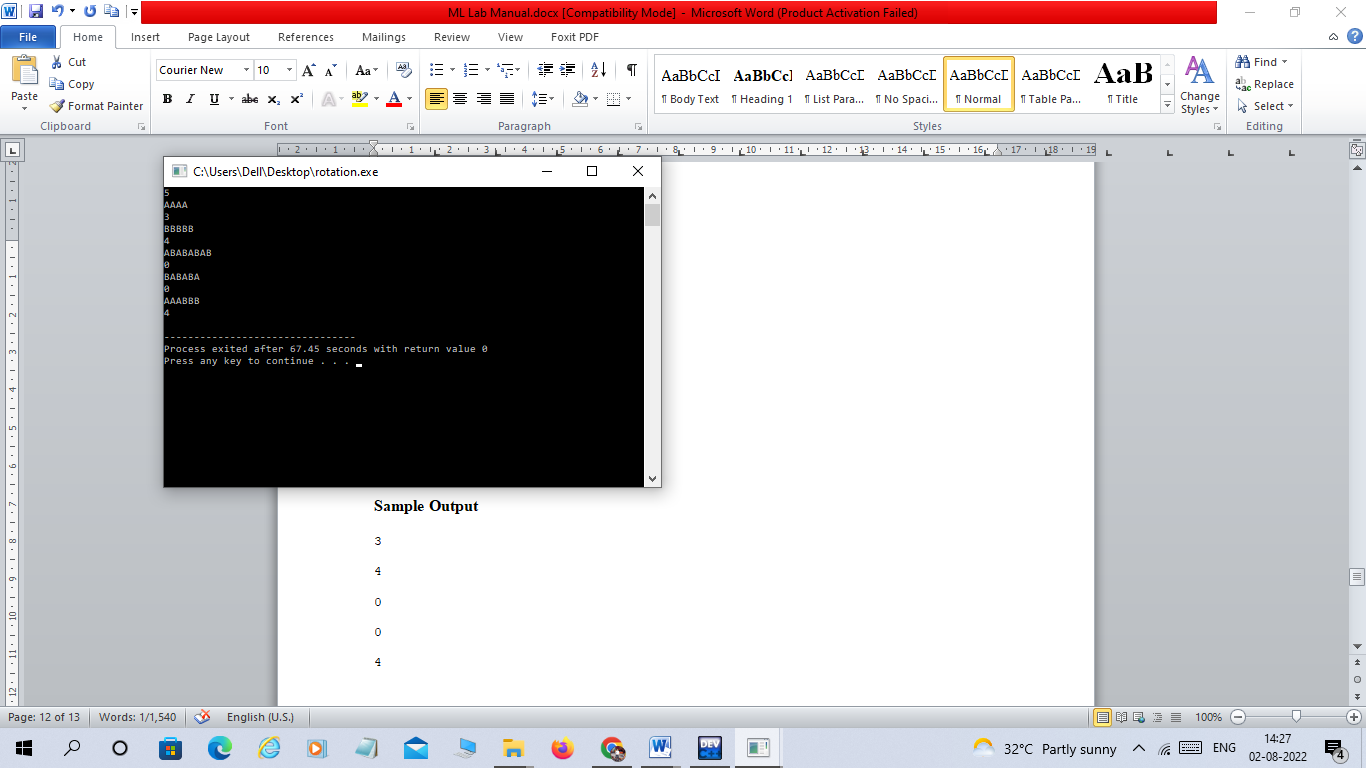
cout<<c<<endl;

}

return 0;

}

OUTPUT



**EXPERIMENT -4**

**Mapped Course Outcomes-CO1**

**CO1:** Explain the data structure and OOPS concepts using C++.

**Aim:** Write a program to maintain a elementary database of employees using files.

**Experiment -4**

**Program**

#include &lt;fstream&gt;

#include &lt;iostream&gt;

using namespace std;

int main () {

char data[100];

// open a file in write mode.

ofstream outfile;

outfile.open(&quot;afile.dat&quot;);

cout &lt;&lt; &quot;Writing to the file&quot; &lt;&lt; endl;

cout &lt;&lt; &quot;Enter your name: &quot;;

cin.getline(data, 100);

// write inputted data into the file.

outfile &lt;&lt; data &lt;&lt; endl;

cout &lt;&lt; &quot;Enter your age: &quot;;

cin &gt;&gt; data;

cin.ignore();

// again write inputted data into the file.

outfile &lt;&lt; data &lt;&lt; endl;

// close the opened file.

outfile.close();

// open a file in read mode.

ifstream infile;

infile.open(&quot;afile.dat&quot;);

cout &lt;&lt; &quot;Reading from the file&quot; &lt;&lt; endl;

infile &gt;&gt; data;

// write the data at the screen.

cout &lt;&lt; data &lt;&lt; endl;

// again read the data from the file and display it.

infile &gt;&gt; data;

cout &lt;&lt; data &lt;&lt; endl;

// close the opened file.

infile.close();

return 0;

}

**Output:**

Writing to the file

Enter your name: Shiv

Enter your age: 50

Reading from the file

Shiv

50

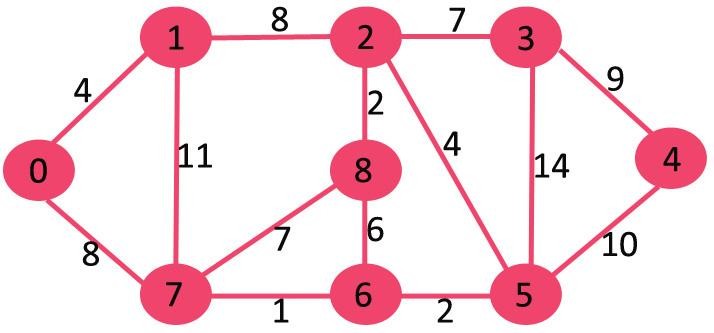
**EXPERIMENT -5**

**Mapped Course Outcomes- CO2 & CO3**

**CO2:** Apply the shortest path and minimum spanning algorithms in computer networks.

**CO3:** Examine the complexity of searching and sorting algorithms, and optimization through arrays, linked structures, stacks, queues, trees, and graphs.

**Aim: a)** From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra’s algorithm.



**Experiment -4(a)**

**Explanation**

Dijkstra’s algorithm is very similar to [Prim’s algorithm for minimum spanning tree](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/). Like Prim’s MST, we generate a SPT (shortest path tree) with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.  
Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm   
1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.   
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.   
3) While sptSet doesn’t include all vertices   
a) Pick a vertex u which is not there in sptSet and has a minimum distance value.   
b) Include u to sptSet.   
c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

**Program**

|  |
| --- |
| #include <iostream>  using namespace std;  #include <limits.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  void printSolution(int dist[])  {      cout <<"Vertex \t Distance from Source" << endl;      for (int i = 0; i < V; i++)          cout  << i << " \t\t"<<dist[i]<< endl;  }    // 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);  }    // 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;  } |

**Output:**

Vertex Distance from Source

0 0

1 4

2 12

3 19

4 21

5 11

6 9

7 8

8 14

**Experiment 5(b)**

**Aim:** Compute the transitive closure of a given directed graph using Warshall's algorithm.

#### Program

#include<iostream>

using namespace std;

const int n\_nodes = 20;

int main() {

int n\_nodes, k, n;

char i, j, res, c;

int adj[10][10], path[10][10];

cout << "\n\tMaximum number of nodes in the graph :";

cin >>n;

n\_nodes = n;

cout << "\nEnter 'y' for 'YES' and 'n' for 'NO' \n";

for (i = 97; i < 97 + n\_nodes; i++)

for (j = 97; j < 97 + n\_nodes; j++) {

cout << "\n\tIs there an edge from " << i << " to " << j << " ? ";

cin >>res;

if (res == 'y')

adj[i - 97][j - 97] = 1;

else

adj[i - 97][j - 97] = 0;

}

cout << "\nTransitive Closure of the Graph:\n";

cout << "\n\t\t\t ";

for (i = 0; i < n\_nodes; i++) {

c = 97 + i;

cout << c << " ";

}

cout << "\n\n";

for (int i = 0; i < n\_nodes; i++) {

c = 97 + i;

cout << "\t\t\t" << c << " ";

for (int j = 0; j < n\_nodes; j++)

cout << adj[i][j] << " ";

cout << "\n";

}

return 0;

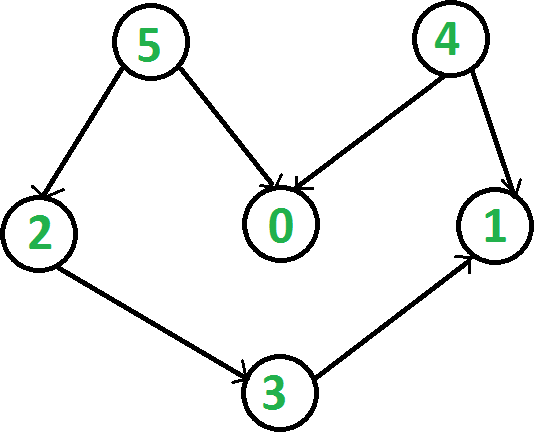
}

**EXPERIMENT -6**

**Mapped Course Outcomes-CO3**

**CO3:** Examine the complexity of searching and sorting algorithms, and optimization through arrays, linked structures, stacks, queues, trees, and graphs.

**Aim:** Obtain the Topological ordering of vertices in a given digraph.



**Experiment -4**

**Explanation:**

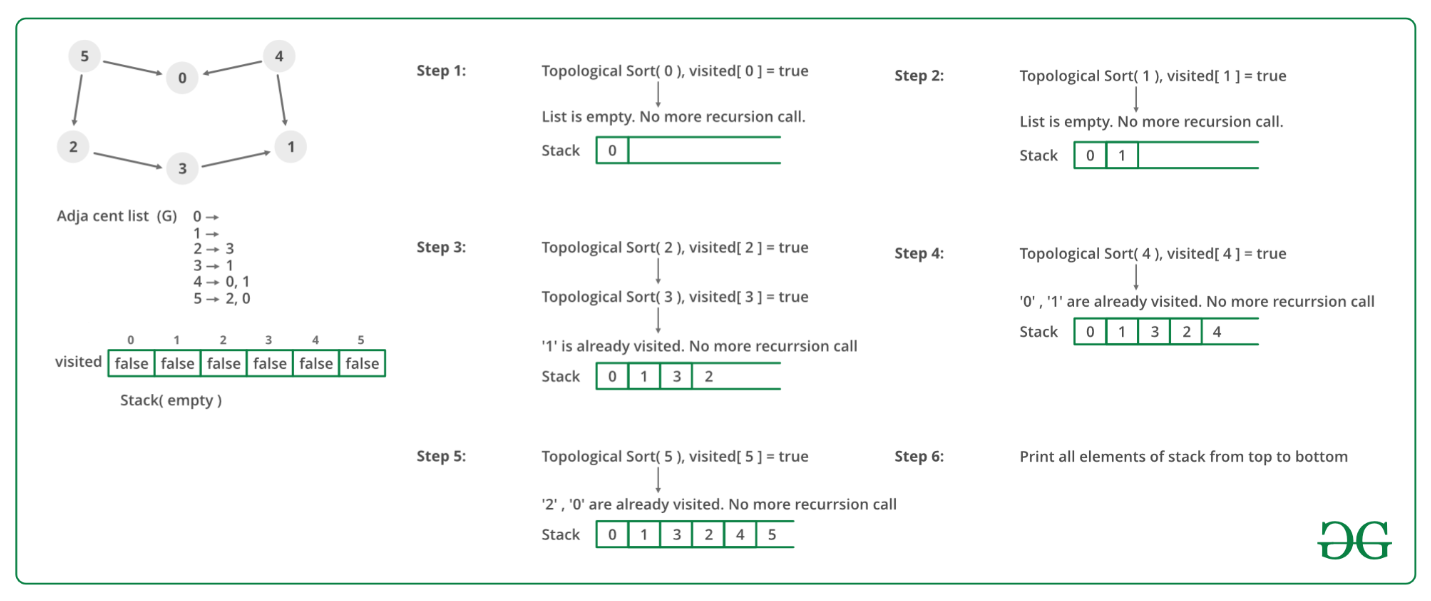
Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).

**Algorithm to find Topological Sorting:**

We recommend to first see the implementation of [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/). We can modify [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)to find Topological Sorting of a graph. In [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of the stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack.

Below image is an illustration of the above approach:



**Program**

#include <iostream>  
#include <list>  
#include <stack>  
using namespace std;  
   
// Class to represent a graph  
class Graph {  
    int V; // No. of vertices'  
   
    // Pointer to an array containing adjacency listsList  
    list<int>\* adj;  
   
    // A function used by topologicalSort  
    void topologicalSortUtil(int v, bool visited[], stack<int>& Stack);  
   
public:  
    Graph(int V); // Constructor  
   
    // function to add an edge to graph  
    void addEdge(int v, int w);  
   
    // prints a Topological Sort of the complete graph  
    void topologicalSort();  
};  
   
Graph::Graph(int V)  
{  
    this->V = V;  
    adj = new list<int>[V];  
}  
   
void Graph::addEdge(int v, int w)  
{  
    adj[v].push\_back(w); // Add w to v’s list.  
}  
   
// A recursive function used by topologicalSort  
void Graph::topologicalSortUtil(int v, bool visited[],  
                                stack<int>& Stack)  
{  
    // Mark the current node as visited.  
    visited[v] = true;  
   
    // 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])  
            topologicalSortUtil(\*i, visited, Stack);  
   
    // Push current vertex to stack which stores result  
    Stack.push(v);  
}  
   
// The function to do Topological Sort. It uses recursive  
// topologicalSortUtil()  
void Graph::topologicalSort()  
{  
    stack<int> Stack;  
   
    // Mark all the vertices as not visited  
    bool\* visited = new bool[V];  
    for (int i = 0; i < V; i++)  
        visited[i] = false;  
   
    // Call the recursive helper function to store Topological  
    // Sort starting from all vertices one by one  
    for (int i = 0; i < V; i++)  
        if (visited[i] == false)  
            topologicalSortUtil(i, visited, Stack);  
   
    // Print contents of stack  
    while (Stack.empty() == false) {  
        cout << Stack.top() << " ";  
        Stack.pop();  
    }  
}  
   
// Driver program to test above functions  
int main()  
{  
    // Create a graph given in the above diagram  
    Graph g(6);  
    g.addEdge(5, 2);  
    g.addEdge(5, 0);  
    g.addEdge(4, 0);  
    g.addEdge(4, 1);  
    g.addEdge(2, 3);  
    g.addEdge(3, 1);  
   
    cout << "Following is a Topological Sort of the given graph n";  
    g.topologicalSort();  
   
    return 0;  
}

**Output**

Following is a Topological Sort of the given graph

5 4 2 3 1 0

**EXPERIMENT -7**

**Mapped Course Outcomes- CO3**

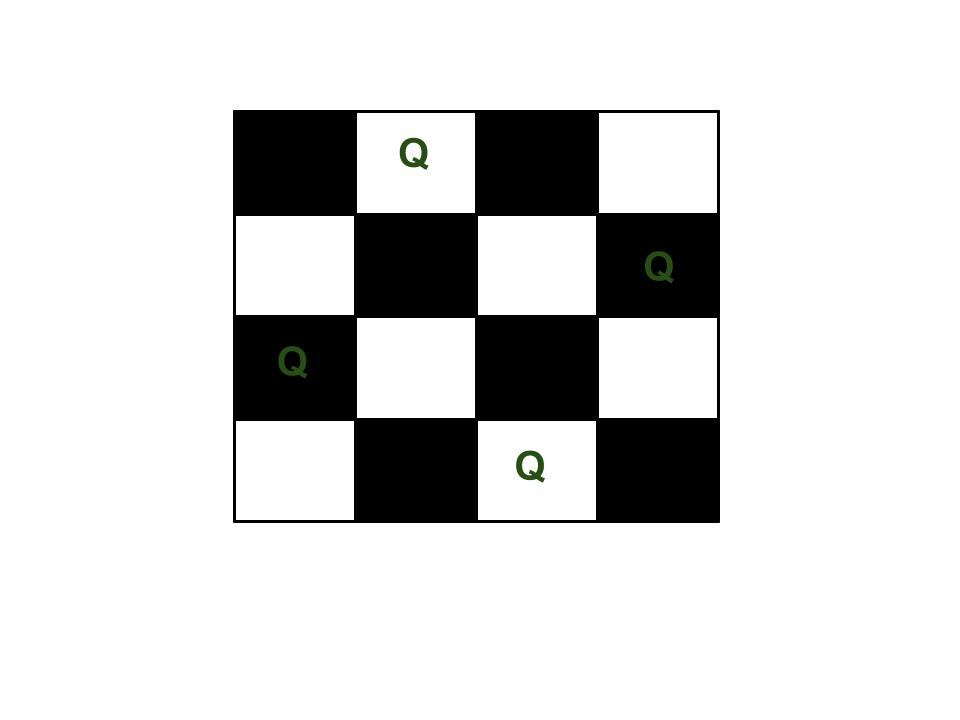
**CO3:** Examine the complexity of searching and sorting algorithms, and optimization through arrays, linked structures, stacks, queues, trees, and graphs.

**Aim:** Implement N Queen's problem using Back Tracking.

**Experiment 7:**

**Explanation:**

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, the following is a solution for the 4 Queen problem.



The expected output is a binary matrix that has 1s for the blocks where queens are placed.

**Backtracking Algorithm**

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.

1) Start in the leftmost column

2) If all queens are placed

return true

3) Try all rows in the current column.

Do following for every tried row.

a) If the queen can be placed safely in this row

then mark this [row, column] as part of the

solution and recursively check if placing

queen here leads to a solution.

b) If placing the queen in [row, column] leads to

a solution then return true.

c) If placing queen doesn't lead to a solution then

unmark this [row, column] (Backtrack) and go to

step (a) to try other rows.

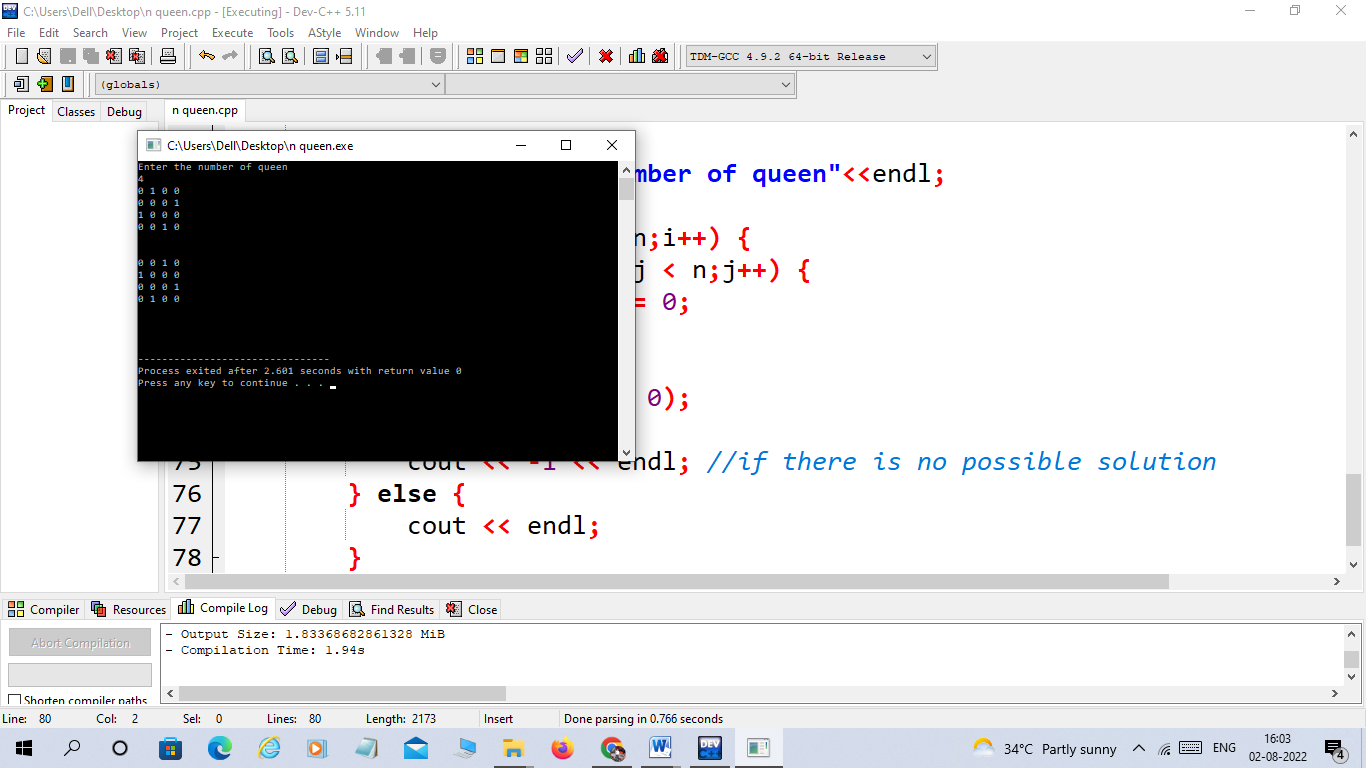
4) If all rows have been tried and nothing worked,

return false to trigger backtracking.

##### Program

#include<iostream>  
using namespace std;  
int grid[10][10];  
//print the solution  
void print(int n) {  
    for (int i = 0;i <= n-1; i++) {  
        for (int j = 0;j <= n-1; j++) {  
             
                cout <<grid[i][j]<< " ";  
             
        }  
        cout<<endl;  
    }  
    cout<<endl;  
    cout<<endl;  
}  
//function for check the position is safe or not  
//row is indicates the queen no. and col represents the possible positions  
bool isSafe(int col, int row, int n) {  
  //check for same column  
    for (int i = 0; i < row; i++) {  
        if (grid[i][col]) {  
            return false;  
        }  
    }  
    //check for upper left diagonal  
    for (int i = row,j = col;i >= 0 && j >= 0; i--,j--) {  
        if (grid[i][j]) {  
            return false;  
        }  
    }  
    //check for upper right diagonal  
    for (int i = row, j = col; i >= 0 && j < n; j++, i--) {  
        if (grid[i][j]) {  
            return false;  
        }  
    }  
    return true;  
}  
//function to find the position for each queen  
//row is indicates the queen no. and col represents the possible positions  
bool solve (int n, int row) {  
    if (n == row) {  
        print(n);  
        return true;  
    }  
    //variable res is use for possible backtracking  
    bool res = false;  
    for (int i = 0;i <=n-1;i++) {  
        if (isSafe(i, row, n)) {  
            grid[row][i] = 1;  
            //recursive call solve(n, row+1) for next queen (row+1)  
            res = solve(n, row+1) || res;//if res ==false then backtracking will occur  
            //by assigning the grid[row][i] = 0  
             
            grid[row][i] = 0;  
        }  
    }  
    return res;  
}  
int main()  
{  
  ios\_base::sync\_with\_stdio(false);  
    cin.tie(NULL);  
        int n;  
        cout<<"Enter the number of queen"<<endl;  
        cin >> n;  
        for (int i = 0;i < n;i++) {  
            for (int j = 0;j < n;j++) {  
                grid[i][j] = 0;  
            }  
        }  
        bool res = solve(n, 0);  
        if(res == false) {  
            cout << -1 << endl; //if there is no possible solution  
        } else {  
            cout << endl;  
        }  
  return 0;  
}

**Output:**



**EXPERIMENT -8**

**Mapped Course Outcomes- CO4**

**CO4:** Decide and implement an appropriate graph algorithm and hashing function in computer networks for data security.

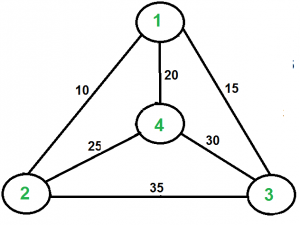
**Aim:** Implement Travelling Salesperson problem using Dynamic programming.

**Experiment 8:**

**Explanation:**

[Travelling Salesman Problem (TSP) :](https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/) Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.   
Note the difference between [Hamiltonian Cycle](https://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/) and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.  
The problem is a famous NP-hard problem. There is no polynomial-time known solution for this problem. 



**Program**

**#include<iostream>**

**using** **namespace** std;

**#define MAX 9999**

**int** n=4; // Number of the places want to visit

//Next distan array will give Minimum distance through all the position

**int** distan[10][10] = {

{0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0}

};

**int** completed\_visit = (1<<n) -1;

**int** DP[16][4];

**int** TSP(**int** mark,**int** position){

**if**(mark==completed\_visit){ // Initially checking whether all

// the places are visited or not

**return** distan[position][0];

}

**if**(DP[mark][position]!=-1){

**return** DP[mark][position];

}

//Here we will try to go to every other places to take the minimum

// answer

**int** answer = MAX;

//Visit rest of the unvisited cities and mark the . Later find the

//minimum shortest path

**for**(**int** city=0;city<n;city++){

**if**((mark&(1<<city))==0){

**int** newAnswer = distan[position][city] + TSP( mark|(1<<city),city);

answer = min(answer, newAnswer);

}

}

**return** DP[mark][position] = answer;

}

**int** main(){

/\* initialize the DP array \*/

**for**(**int** i=0;i<(1<<n);i++){

**for**(**int** j=0;j<n;j++){

DP[i][j] = -1;

}

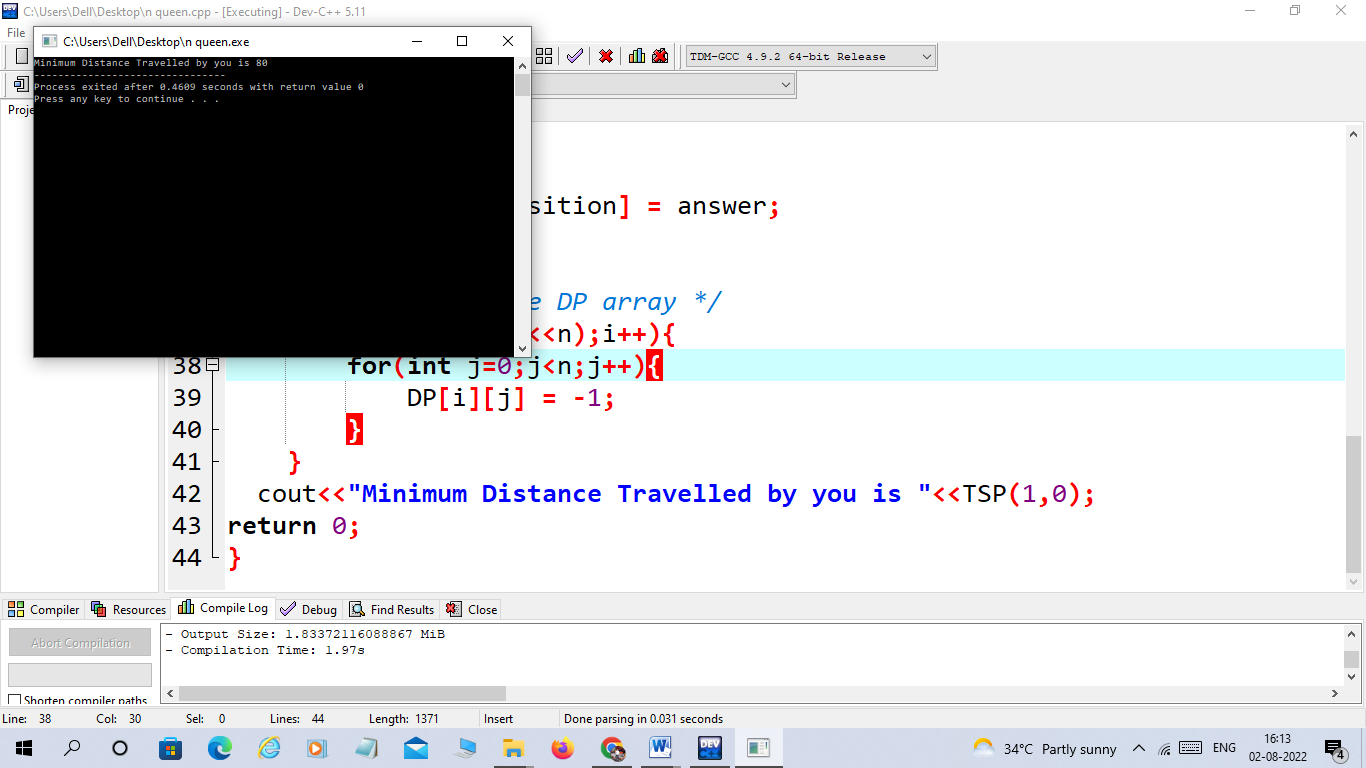
}

cout<<"Minimum Distance Travelled by you is "<<TSP(1,0);

**return** 0;

}

**Output:**



**EXPERIMENT -9**

**Mapped Course Outcomes- CO4**

**CO4:** Decide and implement an appropriate graph algorithm and hashing function in computer networks for data security.

**Aim:** Design a quick sort with random pivoting using Lomuto partition scheme.

**Experiment 9:**

**Explanation:**

In QuickSort we first partition the array in place such that all elements to the left of the pivot element are smaller, while all elements to the right of the pivot are greater than the pivot. Then we recursively call the same procedure for left and right subarrays.   
Unlike [merge sort](https://www.geeksforgeeks.org/merge-sort/), we don’t need to merge the two sorted arrays. Thus Quicksort requires lesser auxiliary space than Merge Sort, which is why it is often preferred to Merge Sort. Using a randomly generated pivot we can further improve the time complexity of QuickSort.

**Lomuto’s Partition Scheme:** This algorithm works by assuming the pivot element as the last element. If any other element is given as a pivot element then swap it first with the last element. Now initialize two variables i as low and j also low,  iterate over the array and increment i when arr[j] <= pivot and swap arr[i] with arr[j] otherwise increment only i. After coming out from the loop swap arr[i] with arr[hi]. This i stores the pivot element.

partition(arr[], lo, hi)

pivot = arr[hi]

i = lo // place for swapping

for j := lo to hi – 1 do

if arr[j] <= pivot then

i = i + 1

swap arr[i] with arr[j]

swap arr[i] with arr[hi]

return i

###### Algorithm for random pivoting using Lomuto Partitioning

partition(arr[], lo, hi)

pivot = arr[hi]

i = lo // place for swapping

for j := lo to hi – 1 do

if arr[j] <= pivot then

swap arr[i] with arr[j]

i = i + 1

swap arr[i] with arr[hi]

return i

partition\_r(arr[], lo, hi)

r = Random Number from lo to hi

Swap arr[r] and arr[hi]

return partition(arr, lo, hi)

quicksort(arr[], lo, hi)

if lo < hi

p = partition\_r(arr, lo, hi)

quicksort(arr, lo , p-1)

quicksort(arr, p+1, hi)

Program

#include <cstdlib>

#include <time.h>

#include <iostream>

using namespace std;

// This function takes last element

// as pivot, places

// the pivot element at its correct

// position in sorted array, and

// places all smaller (smaller than pivot)

// to left of pivot and all greater

// elements to right of pivot

int partition(int arr[], int low, int high)

{

    // pivot

    int pivot = arr[high];

    // Index of smaller element

    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)

    {

        // If current element is smaller

        // than or equal to pivot

        if (arr[j] <= pivot) {

            // increment index of

            // smaller element

            i++;

            swap(arr[i], arr[j]);

        }

    }

    swap(arr[i + 1], arr[high]);

    return (i + 1);

}

// Generates Random Pivot, swaps pivot with

// end element and calls the partition function

int partition\_r(int arr[], int low, int high)

{

    // Generate a random number in between

    // low .. high

    srand(time(NULL));

    int random = low + rand() % (high - low);

    // Swap A[random] with A[high]

    swap(arr[random], arr[high]);

    return partition(arr, low, high);

}

/\* The main function that implements

QuickSort

arr[] --> Array to be sorted,

low --> Starting index,

high --> Ending index \*/

void quickSort(int arr[], int low, int high)

{

    if (low < high) {

        /\* pi is partitioning index,

        arr[p] is now

        at right place \*/

        int pi = partition\_r(arr, low, high);

        // Separately sort elements before

        // partition and after partition

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        cout<<arr[i]<<" ";

}

// Driver Code

int main()

{

    int arr[] = { 10, 7, 8, 9, 1, 5 };

    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);

    printf("Sorted array: \n");

    printArray(arr, n);

    return 0;

}

**Output**

Sorted array:

1 5 7 8 9 10

**EXPERIMENT -10**

**Mapped Course Outcomes- CO4**

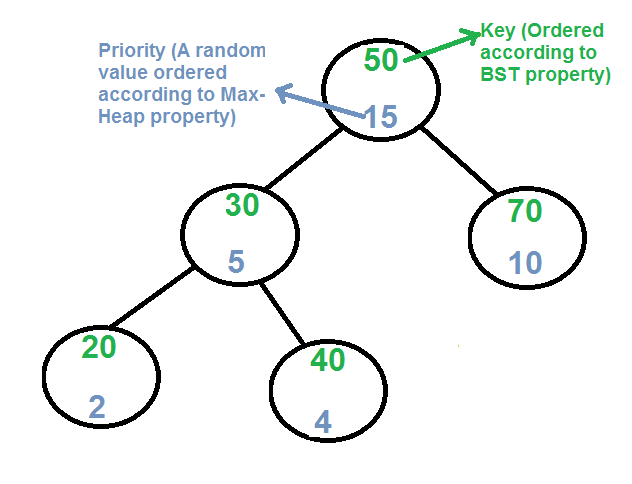
**CO4:** Decide and implement an appropriate graph algorithm and hashing function in computer networks for data security.

**Aim:** Demonstrate insert, delete and search in Treap.

**Experiment 10:**

**Explanation:**

**Treap** is a Balanced Binary Search Tree, but not guaranteed to have height as O(Log n). The idea is to use Randomization and Binary Heap property to maintain balance with high probability. The expected time complexity of search, insert and delete is O(Log n).

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/treap.png)

Every node of Treap maintains two values.  
1) **Key** Follows standard BST ordering (left is smaller and right is greater)  
2) **Priority** Randomly assigned value that follows Max-Heap property.

**Basic Operation on Treap:**  
Like other self-balancing Binary Search Trees, Treap uses rotations to maintain Max-Heap property during insertion and deletion.

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

or x (on right side)

y x

/ \ Right Rotation / \

x T3 – – – – – – – > T1 y

/ \ < - - - - - - - / \

T1 T2 Left Rotation T2 T3

Keys in both of the above trees follow the following order

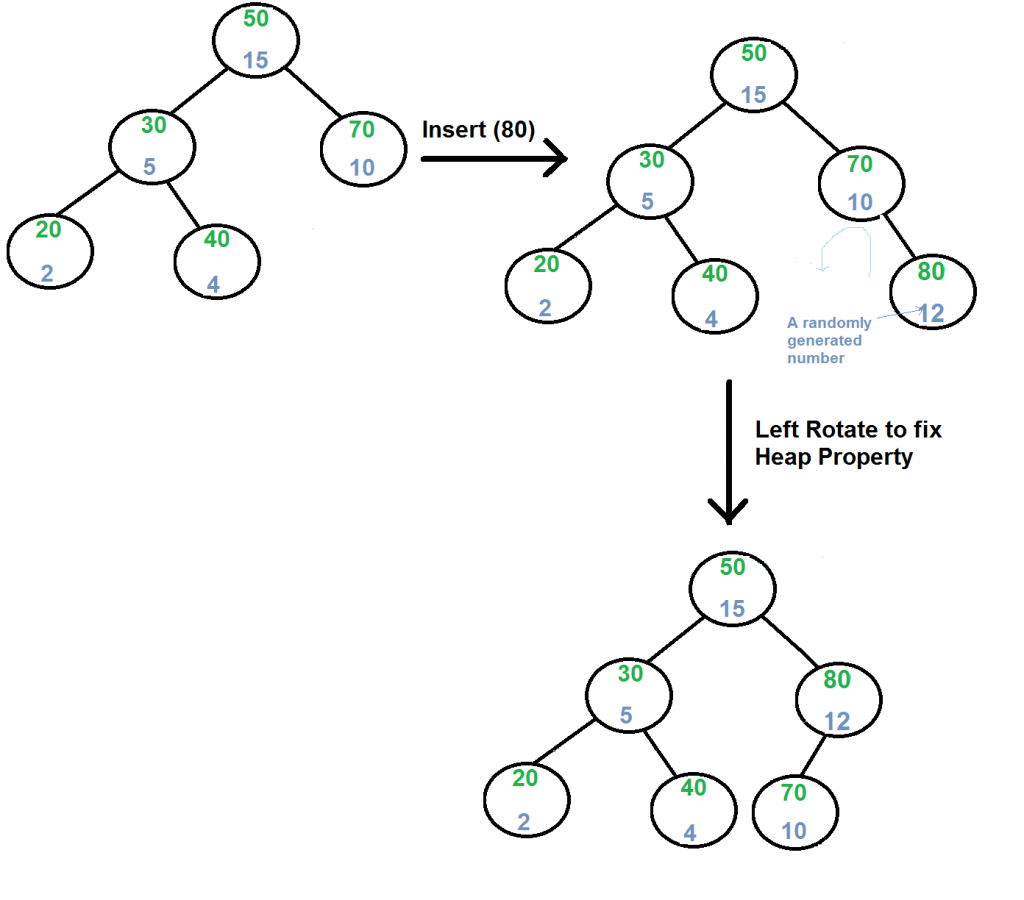
keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

So BST property is not violated anywhere.

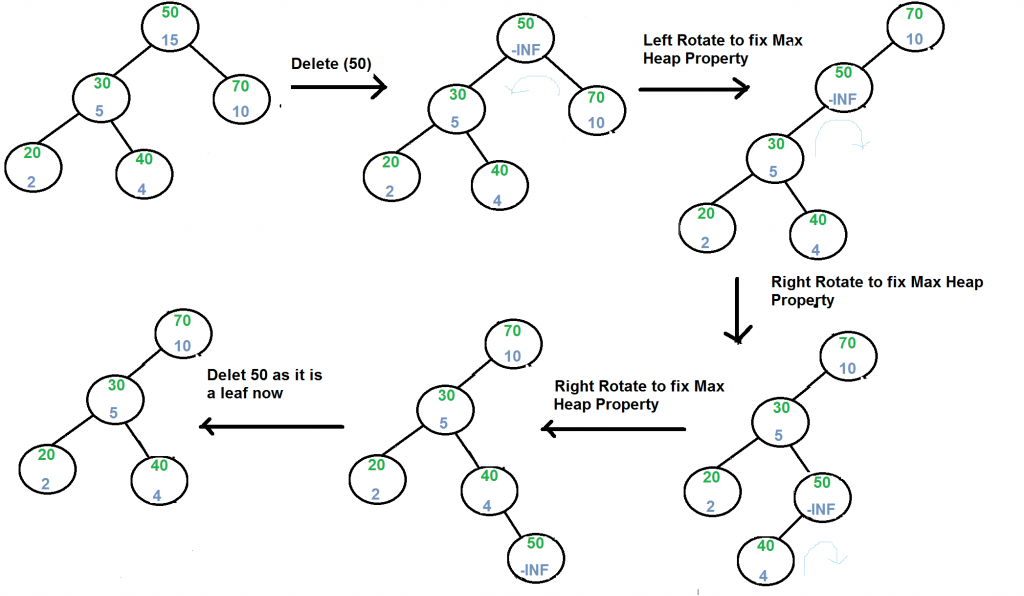
**search(x)**

Perform standard [BST Search](http://geeksquiz.com/binary-search-tree-set-1-search-and-insertion/) to find x.

**Insert(x):**  
1) Create new node with key equals to x and value equals to a random value.  
2) Perform standard [BST insert](http://geeksquiz.com/binary-search-tree-set-1-search-and-insertion/).  
3) Use rotations to make sure that inserted node's priority follows max heap property.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/treapInsert1.png)

**Delete(x):**  
1) If node to be deleted is a leaf, delete it.  
2) Else replace node's priority with minus infinite ( -INF ), and do appropriate rotations to bring the node down to a leaf.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/treapDelete.png)

**Program**

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;   // A Treap Node  struct TreapNode  {      int key, priority;      TreapNode \*left, \*right;  };    /\* T1, T2 and T3 are subtrees of the tree rooted with y    (on left side) or x (on right side)                  y                               x                 / \     Right Rotation          /  \                x   T3   – – – – – – – >        T1   y               / \       < - - - - - - -            / \              T1  T2     Left Rotation            T2  T3 \*/    // A utility function to right rotate subtree rooted with y  // See the diagram given above.  TreapNode \*rightRotate(TreapNode \*y)  {      TreapNode \*x = y->left,  \*T2 = x->right;        // Perform rotation      x->right = y;      y->left = T2;        // Return new root      return x;  }    // A utility function to left rotate subtree rooted with x  // See the diagram given above.  TreapNode \*leftRotate(TreapNode \*x)  {      TreapNode \*y = x->right, \*T2 = y->left;        // Perform rotation      y->left = x;      x->right = T2;        // Return new root      return y;  }    /\* Utility function to add a new key \*/  TreapNode\* newNode(int key)  {      TreapNode\* temp = new TreapNode;      temp->key = key;      temp->priority = rand()%100;      temp->left = temp->right = NULL;      return temp;  }    // C function to search a given key in a given BST  TreapNode\* search(TreapNode\* root, int key)  {      // Base Cases: root is null or key is present at root      if (root == NULL || root->key == key)         return root;        // Key is greater than root's key      if (root->key < key)         return search(root->right, key);        // Key is smaller than root's key      return search(root->left, key);  }    /\* Recursive implementation of insertion in Treap \*/  TreapNode\* insert(TreapNode\* root, int key)  {      // If root is NULL, create a new node and return it      if (!root)          return newNode(key);        // If key is smaller than root      if (key <= root->key)      {          // Insert in left subtree          root->left = insert(root->left, key);            // Fix Heap property if it is violated          if (root->left->priority > root->priority)              root = rightRotate(root);      }      else  // If key is greater      {          // Insert in right subtree          root->right = insert(root->right, key);            // Fix Heap property if it is violated          if (root->right->priority > root->priority)              root = leftRotate(root);      }      return root;  }    /\* Recursive implementation of Delete() \*/  TreapNode\* deleteNode(TreapNode\* root, int key)  {      if (root == NULL)          return root;        if (key < root->key)          root->left = deleteNode(root->left, key);      else if (key > root->key)          root->right = deleteNode(root->right, key);        // IF KEY IS AT ROOT        // If left is NULL      else if (root->left == NULL)      {          TreapNode \*temp = root->right;          delete(root);          root = temp;  // Make right child as root      }        // If Right is NULL      else if (root->right == NULL)      {          TreapNode \*temp = root->left;          delete(root);          root = temp;  // Make left child as root      }        // If key is at root and both left and right are not NULL      else if (root->left->priority < root->right->priority)      {          root = leftRotate(root);          root->left = deleteNode(root->left, key);      }      else      {          root = rightRotate(root);          root->right = deleteNode(root->right, key);      }        return root;  }    // A utility function to print tree  void inorder(TreapNode\* root)  {      if (root)      {          inorder(root->left);          cout << "key: "<< root->key << " | priority: %d "              << root->priority;          if (root->left)              cout << " | left child: " << root->left->key;          if (root->right)              cout << " | right child: " << root->right->key;          cout << endl;          inorder(root->right);      }  }      // Driver Program to test above functions  int main()  {      srand(time(NULL));        struct TreapNode \*root = NULL;      root = insert(root, 50);      root = insert(root, 30);      root = insert(root, 20);      root = insert(root, 40);      root = insert(root, 70);      root = insert(root, 60);      root = insert(root, 80);        cout << "Inorder traversal of the given tree \n";      inorder(root);        cout << "\nDelete 20\n";      root = deleteNode(root, 20);      cout << "Inorder traversal of the modified tree \n";      inorder(root);        cout << "\nDelete 30\n";      root = deleteNode(root, 30);      cout << "Inorder traversal of the modified tree \n";      inorder(root);        cout << "\nDelete 50\n";      root = deleteNode(root, 50);      cout << "Inorder traversal of the modified tree \n";      inorder(root);        TreapNode \*res = search(root, 50);      (res == NULL)? cout << "\n50 Not Found ":                     cout << "\n50 found";        return 0;  } |

Output:

Inorder traversal of the given tree

key: 20 | priority: %d 92 | right child: 50

key: 30 | priority: %d 48 | right child: 40

key: 40 | priority: %d 21

key: 50 | priority: %d 73 | left child: 30 | right child: 60

key: 60 | priority: %d 55 | right child: 70

key: 70 | priority: %d 50 | right child: 80

key: 80 | priority: %d 44

Delete 20

Inorder traversal of the modified tree

key: 30 | priority: %d 48 | right child: 40

key: 40 | priority: %d 21

key: 50 | priority: %d 73 | left child: 30 | right child: 60

key: 60 | priority: %d 55 | right child: 70

key: 70 | priority: %d 50 | right child: 80

key: 80 | priority: %d 44

Delete 30

Inorder traversal of the modified tree

key: 40 | priority: %d 21

key: 50 | priority: %d 73 | left child: 40 | right child: 60

key: 60 | priority: %d 55 | right child: 70

key: 70 | priority: %d 50 | right child: 80

key: 80 | priority: %d 44

Delete 50

Inorder traversal of the modified tree

key: 40 | priority: %d 21

key: 60 | priority: %d 55 | left child: 40 | right child: 70

key: 70 | priority: %d 50 | right child: 80

key: 80 | priority: %d 44

50 Not Found