1. ASSEMBLY LINE SCHEDULING AND OPTIMAL PATH

**THEORY:**

A factory has two assembly lines, each with n stations. A station is denoted by Si, j where i is either 1 or 2 and indicates the assembly line the station is on, and j indicates the number of the station. The time taken per station is denoted by A i,j. Each station is dedicated to some sort of work like engine fitting, body fitting, painting and so on. So, a chassis must pass through each of the n stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station Si,j it will continue to station Si,j+1 unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line i at station j – 1 to station j on the other line takes time tI,j. Each assembly line takes an entry time and exit time xi which may be different for the two lines.

Code:

#include<bits/stdc++.h>

using namespace std;

int main(){

int n;

cout<<"no of stages\n";

scanf("%d",&n);

int time[2][n+4];//0 1st line 1 2nd line

int e1,e2,x1,x2;

int shift[2][n+4];

int dp[2][n+4];

int path[n+3];

cout<<"entry time e1 and e2\n";cin>>e1>>e2;cout<<"time for line 1\n";

for(int i=0;i<n;i++)

cin>>time[0][i];cout<<"time for line 2\n";

for(int i=0;i<n;i++)

cin>>time[1][i];cout<<"time for shift line 1to 2\n";

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

cin>>shift[0][i];cout<<"time for shift line 2to 1\n";

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

cin>>shift[1][i];cout<<"exit time x1,x2\n";

cin>>x1>>x2;

dp[0][0]=e1+time[0][0];

dp[1][0]=e2+time[1][0];

// optimal sub problem

for(int i=1;i<n;i++)

{if(dp[0][i-1]<=dp[1][i-1]+shift[1][i-1]){

dp[0][i]= dp[0][i-1]+time[0][i];

}else

{dp[0][i]= dp[1][i-1]+time[0][i]+shift[1][i-1];

}

if(dp[1][i-1]<=dp[0][i-1]+shift[0][i-1]){

dp[1][i]= dp[1][i-1]+time[1][i];

}

else {

dp[1][i]= dp[0][i-1]+time[1][i]+shift[0][i-1];

}

}

dp[0][n-1]+=x1;

dp[1][n-1]+=x2;

// for path

int m=min(dp[0][n-1],dp[1][n-1]);

if(dp[0][n-1]>dp[1][n-1])

path[n-1]=2;

else

path[n-1]=1;

dp[0][n-1]-=x1;

dp[1][n-1]-=x2;

for(int i=n-2;i>=0;i--){

if(path[i+1]==1){

if(dp[0][i]==dp[0][i+1]-time[0][i+1])

path[i]=1;

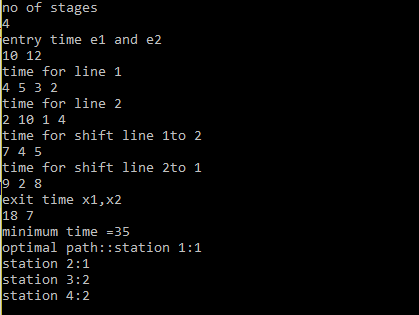
else

path[i]=2;

}

else if (dp[1][i]==dp[1][i+1]-time[1][i+1])

path[i]=2; Output:

else

path[i]=1;

}

}

cout<<"minimum time ="<<m<<endl;

cout<<"optimal path"<<"::";

for(int i=0;i<n;i++)

cout<<path[i];

cout<<endl;

}

2. Matrix Chain Multiplication

**THEORY**: A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parentheses in n-1 ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 ways to place first set of parenthesis outer side: (A)(BCD), (AB)(CD) and (ABC)(D). So when we place a set of parenthesis, we divide the problem into sub problems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

This problem has Overlapping Sub problems property. So we use a dynamic programming problem. Re-computations of same sub problems can be avoided by constructing a temporary array m[][] in bottom up manner.

**CODE**:

#include<bits/stdc++.h>

using namespace std;

int main(){

int arr[100]; //to store size of matrix

int m[100][100];// dp

int i,j,n,p;

while(1){

cin>>n;//no of matries -1

if(n==0)// breaking condition

return 0;

for(i=0;i<n;i++)

cin>>arr[i];

for(i=0;i<n;i++)

m[i][i]=0;//product of matrix it self is zero

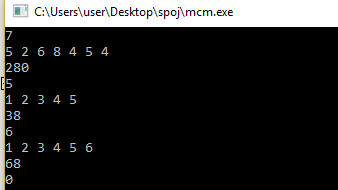
for(int len=2;len<n;len++){

for(i=1;i<n-len+1;i++){j=i+len-1;m[i][j]=INT\_MAX;

for(int k=i;k<j;k++){

**OUTPUT:**

p=m[i][k]+m[k+1][j]+arr[i-1]\*arr[k]\*arr[j];

**** if(p<m[i][j]) m[i][j]=p;

} } }

cout<<m[1][n-1];

}}

# 3. Topological Sorting

**THEORY:**

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

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 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 stack.

**CODE:**

#include<iostream>

#include<bits/stdc++.h>

using namespace std;

int m[105][105];

void dfs(vector<bool>& visited, int s) {

visited[s]=1;

int n=visited.size();

char c= 'A'+s;

cout<<c<<" ";

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

if(m[s][i]&& !visited[i])

dfs(visited,i);

}

}

int main() {

int v,e,i;

char x,y;

cin>>v>>e;

for(i=0;i<e;i++) {

cin>>x>>y;m[x-'A'][y-'A']=1;

}

vector<bool> visited(v,0);

char src;

cin>>src;

dfs(visited,src-'A');

for(i=0;i<v;i++) {

if(!visited[i]){dfs(visited,i);

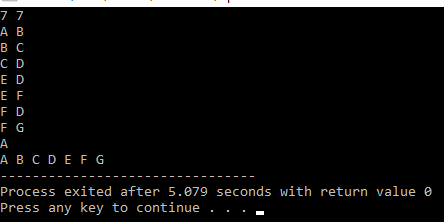
}

}

return 0;

}

**OUTPUT**:



**4. Strongly Connected Components**

**THEORY:** A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph.

We can find all strongly connected components in O(V+E) time using Kosaraju’s algorithm. Following is detailed Kosaraju’s algorithm:

1. Create an empty stack ‘S’ and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.
2. Reverse directions of all arcs to obtain the transpose graph.
3. One by one pop a vertex from S while S is not empty. Let the popped vertex be ‘v’. Take v as source and do DFS (call DFSUtil(v)). The DFS starting from v prints strongly connected component of v.

**CODE:**

#include<bits/stdc++.h>

#define MAXV 1000

using namespace std;

typedef vector <int> vi;

vi G[MAXV], Grev[MAXV];

bool explored[MAXV];

int leader[MAXV], finish[MAXV], order[MAXV], t = -1, parent = 0, V, E;

void dfs\_reverse(int i) {

explored[i] = true;

for(vi::iterator it = Grev[i].begin(); it != Grev[i].end(); it++)

if(!explored[\*it])

dfs\_reverse(\*it);

t++;

finish[i] = t;

}

void dfs(int i) {

explored[i] = true;

cout<<i+1<<" ";

leader[i] = parent;

for(vi::iterator it = G[i].begin(); it != G[i].end(); it++)

if(!explored[\*it])

dfs(\*it);

}

int main() {

int i, u, v, countCC, Q;

scanf("%d %d", &V, &E);

for(i=0; i<E; i++) {

scanf("%d %d", &u, &v);

G[u].push\_back(v);

Grev[v].push\_back(u);

}

memset(explored, false, V\*sizeof(bool));

for(i=0; i<V; i++) {

if(!explored[i])

dfs\_reverse(i);

order[finish[i]] = i;

}

memset(explored, false, V\*sizeof(bool));

countCC = 0; cout<<endl;

for(i=V-1; i>=0; i--)

if(!explored[order[i]]) {

cout<<countCC+1<<": ";

parent = order[i];

dfs(order[i]);

countCC++;

cout<<endl;

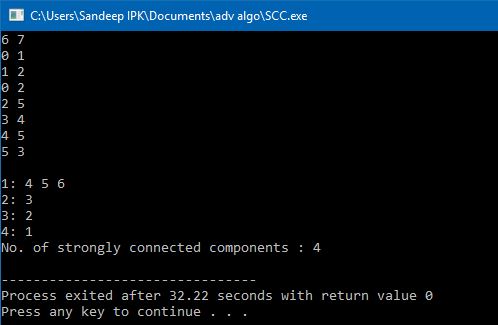
}

printf("No. of strongly connected components : %d\n", countCC);

return 0;

}

**OUTPUT:**



**5. Finding all paths from vertices:**

**THEORY:**

Given a directed graph, a source vertex ‘s’ and a destination vertex ‘d’, print all paths from given ‘s’ to ‘d’. The idea is to do Depth First Traversal of given directed graph. Start the traversal from source. Keep storing the visited vertices in an array say ‘path[]’. If we reach the destination vertex, print contents of path[]. The important thing is to mark current vertices in path[] as visited also, so that the traversal doesn’t go in a cycle.

**CODE:**

#include <vector>

#include <iostream>

using namespace std;

class Node

{

public:

void AddLink(int id)

{

next.push\_back(id);

}

public:

vector <int> next;

};

void FindAllPathsAt(vector <Node> &all\_nodes, int id, vector < vector<int> > &all\_paths, vector <int> tmp)

{

tmp.push\_back(id);

if(all\_nodes[id].next.size() == 0) {

all\_paths.push\_back(tmp);

return;

}

for(size\_t i=0; i < all\_nodes[id].next.size(); i++) {

vector <int> tmp2(tmp);

FindAllPathsAt(all\_nodes, all\_nodes[id].next[i], all\_paths, tmp2);

}

}

void PrintPaths(const vector < vector<int> > &all\_paths)

{

for(size\_t i=0; i < all\_paths.size(); i++) {

if(all\_paths[i].size() == 1) {

continue;

}

cout << all\_paths[i][0];

for(size\_t j=1; j < all\_paths[i].size(); j++) {

cout << " -- > " << all\_paths[i][j];

}

cout << endl;

}

}

int main()

{

vector <Node> all\_nodes(8);

all\_nodes[0].AddLink(4);

all\_nodes[1].AddLink(5);

all\_nodes[2].AddLink(4);

all\_nodes[2].AddLink(5);

all\_nodes[3].AddLink(4);

all\_nodes[3].AddLink(5);

all\_nodes[4].AddLink(6);

all\_nodes[4].AddLink(7);

vector <int> tmp;

for(size\_t i=0; i < all\_nodes.size(); i++) {

vector < vector<int> > all\_paths;

FindAllPathsAt(all\_nodes, i, all\_paths, tmp);

cout << "All paths at node " << i << endl;

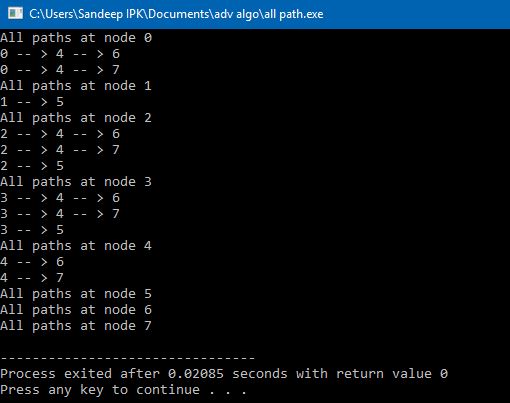
PrintPaths(all\_paths);

}

return 0;

}

**OUTPUT:**



**6. Floyd -Warshall’s Algorithm**

**THEORY:**

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

**CODE:**

#include<iostream>

using namespace std;

#define V 5

#define INF 99999

void printSolution(int dist[][V]);

void floydWarshall (int graph[][V])

{

int dist[V][V], i, j, k;

for (i = 0; i < V; i++)

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

dist[i][j] = graph[i][j];

for (k = 0; k < V; k++)

{

for (i = 0; i < V; i++)

{

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

{

if (dist[i][k] + dist[k][j] < dist[i][j])

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

printSolution(dist);

}

void printSolution(int dist[][V])

{

printf ("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)

printf("%7s", "INF");

else

printf ("%7d", dist[i][j]);

}

printf("\n");

}

}

int main()

{

int graph[V][V] = { {0, 3, 8, INF, -4},

{INF, 0, 3, 1, 7},

{INF, 4, 0, INF, INF},

{2, INF, -5, 0, INF},

{INF, INF, INF, 6, 0}

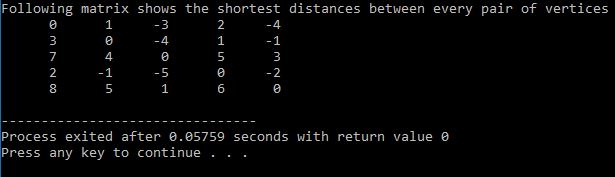
};

floydWarshall(graph);

return 0;

}

**OUTPUT:**



**7. Dijkstra’s Algorithm**

**THEORY:**

Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

**CODE:**

#include <iostream>

#include <limits.h>

using namespace std;

#define V 9

int minDistance(int dist[], bool sptSet[])

{

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;

}

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]);

}

void dijkstra(int graph[V][V], int src)

{

int dist[V];

bool sptSet[V];

for (int i = 0; i < V; i++)

dist[i] = INT\_MAX, sptSet[i] = false;

dist[src] = 0;

for (int count = 0; count < V-1; count++)

{

int u = minDistance(dist, sptSet);

sptSet[u] = true;

for (int v = 0; v < V; 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];

}

printSolution(dist, V);

}

int main()

{

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}

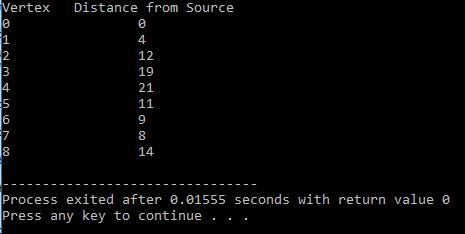
};

dijkstra(graph, 0);

return 0;

}

**OUTPUT:**



**8. Bellman-Ford** Algorithm

**THEORY:**

The algorithm calculates shortest paths in bottom-up manner. It first calculates the shortest distances for the shortest paths which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the ith iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

struct Edge

{

int src, dest, weight;

};

struct Graph

{

int V, E;

struct Edge\* edge;

};

struct Graph\* createGraph(int V, int E)

{

struct Graph\* graph =

(struct Graph\*) malloc( sizeof(struct Graph) );

graph->V = V;

graph->E = E;

graph->edge =

(struct Edge\*) malloc( graph->E \* sizeof( struct Edge ) );

return graph;

}

void printArr(int dist[], int n)

{

printf("\nVertex Distance from Source\n");

for (int i = 0; i < n; ++i)

printf("%d \t\t %d\n", i, dist[i]);

}

void BellmanFord(struct Graph\* graph, int src)

{

int V = graph->V;

int E = graph->E;

int dist[V];

for (int i = 0; i < V; i++)

dist[i] = INT\_MAX;

dist[src] = 0;

for (int i = 1; i <= V-1; i++)

{

for (int j = 0; j < E; j++)

{

int u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

dist[v] = dist[u] + weight;

}

}

for (int i = 0; i < E; i++)

{

int u = graph->edge[i].src;

int v = graph->edge[i].dest;

int weight = graph->edge[i].weight;

}

printArr(dist, V);

return;

}

int main()

{

int V = 5;

int E = 10;

struct Graph\* graph = createGraph(V, E);

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = 6;

graph->edge[1].src = 0;

graph->edge[1].dest = 3;

graph->edge[1].weight = 7;

graph->edge[2].src = 1;

graph->edge[2].dest = 2;

graph->edge[2].weight = 5;

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 8;

graph->edge[4].src = 1;

graph->edge[4].dest = 4;

graph->edge[4].weight = -4;

graph->edge[5].src = 2;

graph->edge[5].dest = 1;

graph->edge[5].weight = -2;

graph->edge[6].src = 3;

graph->edge[6].dest = 2;

graph->edge[6].weight = -3;

graph->edge[7].src = 3;

graph->edge[7].dest = 4;

graph->edge[7].weight = 9;

graph->edge[8].src = 4;

graph->edge[8].dest = 1;

graph->edge[8].weight = 2;

graph->edge[9].src = 4;

graph->edge[9].dest = 2;

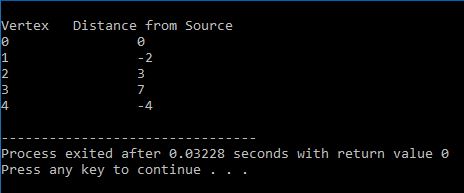
graph->edge[9].weight = 7;

BellmanFord(graph, 0);

return 0;

}

**OUTPUT:**



**9. Transitive Closure of a Graph**

**THEORY:**

Transitive closure of a graph. Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j. The reach-ability matrix is called transitive closure of a graph.

Floyd Warshall Algorithm can be used, we can calculate the distance matrix dist[V][V] using Floyd Warshall, if dist[i][j] is infinite, then j is not reachable from i, otherwise j is reachable and value of dist[i][j] will be less than V.

**CODE:**

#include<stdio.h>

#define V 4

void printSolution(int reach[][V]);

void transitiveClosure(int graph[][V])

{

int reach[V][V], i, j, k;

for (i = 0; i < V; i++)

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

reach[i][j] = graph[i][j];

for (k = 0; k < V; k++)

{

for (i = 0; i < V; i++)

{

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

{

reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);

}

}

}

printSolution(reach);

}

void printSolution(int reach[][V])

{

printf ("Following matrix is transitive closure of the given graph\n");

for (int i = 0; i < V; i++)

{

for (int j = 0; j < V; j++)

printf ("%d ", reach[i][j]);

printf("\n");

}

}

int main()

{

int graph[V][V] = { {1, 0, 0, 0},

{0, 1, 1, 1},

{0, 1, 1, 0},

{1, 0, 1, 1}

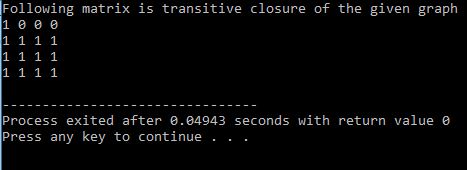
};

transitiveClosure(graph);

return 0;

}

**OUTPUT:**



1. **Rabin-Karp** string matching Algorithm.

**THEORY:**

Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So, Rabin Karp algorithm needs to calculate hash values for following strings.

1 Pattern itself .

2 All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function properly.Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say hash(txt[s+1 .. s+m]) must be efficiently computable from hash(txt[s .. s+m-1]) and txt[s+m] i.e., hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1]) and rehash must be O(1) operation.

hash( txt[s+1 .. s+m] ) = d ( hash( txt[s .. s+m-1]) – txt[s]\*h ) + txt[s + m] ) mod q

hash( txt[s .. s+m-1] ) : Hash value at shift s.

hash( txt[s+1 .. s+m] ) : Hash value at next shift (or shift s+1)

d: Number of characters in the alphabet

q: A prime number

h: d^(m-1)

**CODE:**

#include<stdio.h>

#include<string.h>

#define d 256

int main()

{char txt[80],pat[10];

printf("\n \*\* Rabinkarp Pattern Matcher \*\*\n\n");

printf("Enter the text string: ");

gets(txt);

printf("\nEnter the pattern to be searched: ");

gets(pat);

int q = 101;

int M = strlen(pat);

int N = strlen(txt);

int i, j;

int p = 0;

int t = 0;

int h = 1;

for (i = 0; i < M-1; i++)

h = (h\*d)%q;

for (i = 0; i < M; i++)

{

p = (d\*p + pat[i])%q;

t = (d\*t + txt[i])%q;

}

for (i = 0; i <= N - M; i++)

{

if ( p == t )

{

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

{

if (txt[i+j] != pat[j])

break;

}

if (j == M)

{

printf("pattern matches at shift=%d \n", i);

}

else{

printf(“spurious hit at shift=%d\n, i ”)

}

}

if ( i < N-M )

{

t = (d\*(t - txt[i]\*h) + txt[i+M])%q;

if(t < 0)

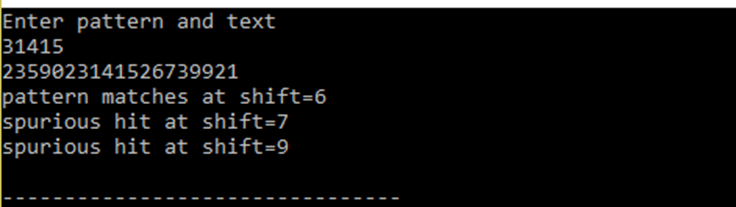
t = (t + q);

}

}

return 0;}

**OUTPUT:**

****

1. **String matching with Finite Automata**

**THEORY:**

In FA based algorithm, we preprocess the pattern and build a 2D array that represents a Finite Automata. Construction of the FA is the main tricky part of this algorithm. Once the FA is built, the searching is simple.

In search, we simply need to start from the first state of the automata and the first character of the text. At every step, we consider next character of text, look for the next state in the built FA and move to a new state.

If we reach the final state, then the pattern is found in the text. The time complexity of the search process is O(n).

**CODE:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

char t[100];

char p[100];

int m,n,i,j=0,k,x,q,d[100][26],l;

void compute\_transition\_function()

{

m=strlen(p);

for(q=0;q<=m;q++)

{

for(i=0;i<l;i++)

{

if(m<q+1)

{k=m;x=1;}

else {k=q+1;

x=0;}

while(k!=0){

j=0;

while(p[j]==p[x+j]&&j<k-1)

j++;

if(p[j]==i+97)

j++;

if(j==k)

break;

else {k--;x++;}

}

d[q][i]=k;

}

}

}

void finite\_automaton\_matcher(){

n=strlen(t);

m=strlen(p);

q=0;

for(i=0;i<n;i++)

{

q=d[q][t[i]-97];

if(q==m)

printf("pattern occur with shift %d\n",i-m+1);

}

}

int main(){

printf("Enter the text: ");

gets(t);

printf("Enter pattern: ");

gets(p);

printf("Enter no. of chars: ");

scanf("%d",&l);

compute\_transition\_function();

finite\_automaton\_matcher();

printf("\nTransition Function :\n\n");

for(i=0;i<=m;i++){

for(j=0;j<l;j++){

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

}

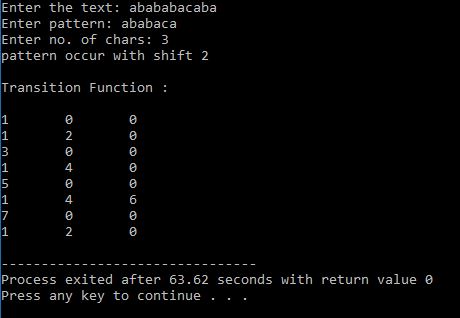
printf("\n");

}

return 0;

}

**OUTPUT:**



1. **LUP decomposition of a matrix**

**THEORY:**

L U decomposition of a matrix is the factorization of a given square matrix into two triangular matrices, one upper triangular matrix and one lower triangular matrix, such that the product of these two matrices gives the original matrix. It was introduced by Alan Turing in 1948, who also created the Turing machine.

This method of factorizing a matrix as a product of two triangular matrices has various applications such as solution of a system of equations, which itself is an integral part of many applications such as finding current in a circuit and solution of discrete dynamical system problems; finding the inverse of a matrix and finding the determinant of the matrix.

**CODE:**

#include<stdio.h>

float a[4][4]={{2,0,2,0.6},

{3,3,4,-2},

{5,5,4,2},

{-1,-2,3.4,-1}

};

int p[4][4]={{1},{2},{3},{4}};

float l[4][4];

float u[4][4];

int main()

{

int n=4; //no. of rows in A

int i,k,j,p2;

float p1;

float temp;

for(k=0;k<n;k++)

{

p1=0;

for(i=k;i<n;i++) //choosing pivot

{

if(a[i][k]<0)

{

temp=-1\*a[i][k];

}

else

{

temp=a[i][k];

}

if(temp>p1)

{

p1=temp;

p2=i;

}

}

if(p1==0)

{

printf("\n error");

}

printf("\n pivot is :%f", p1);

temp=p[k][0];

p[k][0]=p[p2][0];

p[p2][0]=temp;

for(i=0;i<n;i++)

{

temp=a[k][i];

a[k][i]=a[p2][i];

a[p2][i]=temp;

}

for(i=k+1;i<n;i++)

{

a[i][k]=a[i][k]/a[k][k];

for(j=k+1;j<n;j++)

{

a[i][j]=a[i][j]-a[i][k]\*a[k][j];

}

}

printf("\n A MATRIX: \n ");

for(i=0;i<4;i++)

{

printf("\n");

for(j=0;j<4;j++)

{

printf(" %0.02f ",a[i][j]);

}

}

}

printf("\n P MATRIX: \n ");

for(i=0;i<4;i++)

{

printf("\n");

for(j=0;j<4;j++)

{

printf(" %d ",p[i][j]);

}

}

printf("\n P MATRIX: \n ");

for(i=0;i<4;i++)

{

j=p[i][0];

j--;

for(k=0;k<4;k++)

{

if(k==j)

{

p[i][k]=1;

}

else

{

p[i][k]=0;

}

}

}

printf("\n final Permutation MATRIX: \n ");

for(i=0;i<4;i++)

{

printf("\n");

for(j=0;j<4;j++)

{

printf(" %d ",p[i][j]);

}

}

for(i=0;i<4;i++)

{

for(j=0;j<4;j++)

{

if(i==j)

{

l[i][j]=1;

u[i][j]=a[i][j];

}

else if(i>j)

{

l[i][j]=a[i][j];

u[i][j]=0;

}

else

{

l[i][j]=0;

u[i][j]=a[i][j];

}

}

}

printf("\n final L MATRIX: \n ");

for(i=0;i<4;i++)

{

printf("\n");

for(j=0;j<4;j++)

{

printf(" %0.02f ",l[i][j]);

}

}

printf("\n final U MATRIX: \n ");

for(i=0;i<4;i++)

{

printf("\n");

for(j=0;j<4;j++)

{

printf(" %0.02f ",u[i][j]);

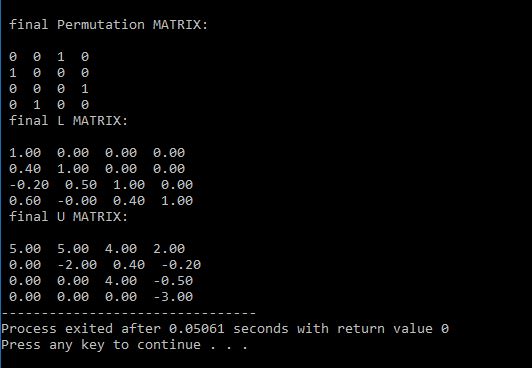
}

}

return 0;

}

**OUTPUT:**



1. **Johnson’s Algorithm**

**THEORY:**

The problem is to find shortest paths between every pair of vertices in a given weighted directed Graph and weights may be negative. We have discussed Floyd Warshall Algorithm for this problem. Time complexity of Floyd Warshall Algorithm is Θ(V3). Using Johnson’s algorithm, we can find all pair shortest paths in O(V2log V + VE) time. Johnson’s algorithm uses both Dijkstra and Bellman-Ford as subroutines.

If we apply Dijkstra’s Single Source shortest path algorithm for every vertex, considering every vertex as source, we can find all pair shortest paths in O(V\*VLogV) time. So using Dijkstra’s single source shortest path seems to be a better option than Floyd Warshell, but the problem with Dijkstra’s algorithm is, it doesn’t work for negative weight edge.

The idea of Johnson’s algorithm is to re-weight all edges and make them all positive, then apply Dijkstra’s algorithm for every vertex.

**CODE:**

#include<stdio.h>

int graph[4][4];

int d[4];

int dist[3];

int parent[3];

int finish[3];

int relax(int u)

{

int i=0;

for(i=0;i<=3;i++)

{

if(graph[u][i]!=1000)

{

if(d[i]>d[u]+graph[u][i])

{

d[i]=d[u]+graph[u][i];

}

}

}

return 0;

}

int relax\_to\_detect\_cycle(int u)

{

int i=0;

int cycle=0;

for(i=0;i<=3;i++)

{

if(graph[u][i]!=1000)

{

if(d[i]>d[u]+graph[u][i])

{

d[i]=d[u]+graph[u][i];

cycle=1;

break;

}

}

}

return cycle;

}

int main()

{

int i=0,k;

for(i=0;i<=3;i++)

{

d[i]=1000;

}

int j;

for(i=0;i<4;i++)

{

for(j=0;j<4;j++)

{

graph[i][j]=1000;

}

}

int e;

do{

printf("\n enter edge cost fron node:i to node:k (enter -1 to stop)\n");

printf(" Source node i: \n");

scanf("%d",&i);

printf(" Destination node k: \n");

scanf("%d",&k);

printf(" edge cost: \n");

scanf("%d",&e);

if(i!=-1||k!=-1)

graph[i][k]=e;

}while(i!=-1&&k!=-1);

i=3;

for(k=0;k<=2;k++){

graph[i][k]=0;

}

printf("\n Graph: \n");

for(i=0;i<4;i++)

{

printf("|");

for(j=0;j<4;j++)

{

printf(" %d ",graph[i][j]);

}

printf("|\n");

}

d[3]=0;

for(i=0;i<=2;i++)

{

for(k=3;k>=0;k--)

{

relax(k);

}

}

int cycle;

for(k=3;k>=0;k--)

{

cycle=relax\_to\_detect\_cycle(k);

if(cycle==1)

{

break;

}

}

if(cycle==1)

{

printf("\n there is negative edge cycle, minimum distance is not possibe \n");

}

for(i=0;i<3;i++)

{

graph[3][i]=d[i];

}

int w[3];

for(i=0;i<3;i++)

{

for(j=0;j<3;j++)

{

if(graph[i][j]!=1000)

{

graph[i][j]=graph[i][j]+d[i]-d[j];

}

}

}

printf("\n Modified Graph: \n");

for(i=0;i<4;i++)

{

printf("|");

for(j=0;j<4;j++)

{

printf(" %d ",graph[i][j]);

}

printf("|\n");

}

int nd=0;

while(nd<3)

{

for(i=0;i<3;i++)

{

parent[i]=nd;

dist[i]=1000;

}

for(i=0;i<3;i++)

{

finish[i]=0;

}

dist[nd]=0;

parent[nd]=-1;

int u=nd;

int round;

int min\_node=-1;

min\_node=u;

k=0;

while(k<3)

{

round=k+1;

for(i=0;i<3;i++)

{

if(graph[min\_node][i]!=1000)

{

relax\_djkstra(min\_node,i);

}

}

finish[min\_node]=1;

int temp=0;

for(i=0;i<3;i++)

{

if(dist[i]!=1000 && finish[i]!=1)

{

if(temp==0)

{

min\_node=i;

temp=1;

}

if(dist[min\_node]>dist[i])

{

min\_node=i;

}

}

}

k++;

}

printf("\n The final distance matrix from source node:%d ---- \n",u);

for(i=0;i<4;i++)

{

printf("%d ",dist[i]);

}

printf("\n The parent matrix from source node:%d ---- \n",u);

for(i=0;i<4;i++)

{

printf("%d ",parent[i]);

}

nd++;

}

return 0;

}

int relax\_djkstra(u,v)

{

if(dist[v]>dist[u]+graph[u][v])

{

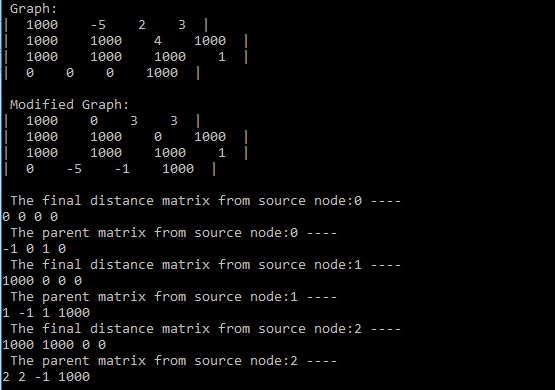
dist[v]=dist[u]+graph[u][v];

parent[v]=u;

}

}

**OUTPUT:**



1. **Multiple Line segment Intersection**

**THEORY:**

Given two-line segments (p1, q1) and (p2, q2), find if the given line segments intersect with each other. Before we discuss solution, let us define notion of orientation. Orientation of an ordered triplet of points in the plane can be:  
–counter clockwise  
–clockwise  
–colinear

A naive solution to solve this problem is to check every pair of lines and check if the pair intersects or not. We can check two line segments in O(1) time. Therefore, this approach takes O(n2).

**CODE:**

#include<stdio.h>

#include<malloc.h>

struct point{

char flag;

int x;

int y;

};

struct line{

struct point \*p1;

struct point \*p2;

};

int Direction(struct point \*p1,struct point \*p2,struct point \*p3)

{

return ((p1->x-p3->x)\*(p2->y-p3->y) - (p2->x-p3->x)\*(p1->y-p3->y));

}

int Onsegment(struct point \*p1,struct point \*p2,struct point \*p3)

{

int minx, maxx, miny, maxy;

if(p1->x >= p2->x){

minx=p2->x;

maxx=p1->x;

}

else{

minx=p1->x;

maxx=p2->x;

}

if(p1->y >= p2->y){

miny=p2->y;

maxy=p1->y;

}

else{

miny=p1->y;

maxy=p2->y;

}

if((minx<=p3->x) && (p3->x<=maxx) && (miny<=p3->y) && (p3->y<=maxy))

return 1;

else

return 0;

}

int Intersect(struct point \*p1,struct point \*p2,struct point \*p3,struct point \*p4)

{

int d1,d2,d3,d4;

d1 = Direction(p3,p4,p1);

d2 = Direction(p3,p4,p2);

d3 = Direction(p1,p2,p3);

d4 = Direction(p1,p2,p4);

if(((d1>0&&d2<0) || (d1<0&&d2>0)) && ((d3>0&&d4<0) || (d3<0&&d4>0))){

return 1;

}

else if((d1==0 && Onsegment(p3,p4,p1)==1) || (d2==0 && Onsegment(p3,p4,p2)==1) || (d3==0 && Onsegment (p1,p2,p3)==1) || (d4==0 && Onsegment(p1,p2,p4)==1)){

return 1;

}

else{

return 0;

}

}

void main()

{

struct point \*p1,\*p2,\*p3,\*p4,\*p5,\*p6;

struct line \*li[3];

int r,i,j;

p1 = (struct point\*)malloc(sizeof(struct point));

p2 = (struct point\*)malloc(sizeof(struct point));

p3 = (struct point\*)malloc(sizeof(struct point));

p4 = (struct point\*)malloc(sizeof(struct point));

p5 = (struct point\*)malloc(sizeof(struct point));

p6 = (struct point\*)malloc(sizeof(struct point));

printf("\nEnter first point.\n");

scanf("(%d,%d)",&p1->x,&p1->y);

printf("\nEnter second point.");

scanf(" (%d,%d)",&p2->x,&p2->y);

printf("\nEnter third point.");

scanf(" (%d,%d)",&p3->x,&p3->y);

printf("\nEnter fourth point.");

scanf(" (%d,%d)",&p4->x,&p4->y);

printf("\nEnter fifth point.");

scanf(" (%d,%d)",&p5->x,&p5->y);

printf("\nEnter sixth point.");

scanf(" (%d,%d)",&p6->x,&p6->y);

li[0] = (struct line\*)malloc(sizeof(struct line));

li[1] = (struct line\*)malloc(sizeof(struct line));

li[2] = (struct line\*)malloc(sizeof(struct line));

li[0]->p1 = p1;

li[0]->p2 = p2;

li[1]->p1 = p3;

li[1]->p2 = p4;

li[2]->p1 = p5;

li[2]->p2 = p6;

for(i=0;i<3;i++){

for(j=i+1;j<3;j++){

r = Intersect(li[i]->p1,li[i]->p2,li[j]->p1,li[j]->p2);

if(r==1)

printf("(%d,%d) Both line intersects.\n",i,j);

else

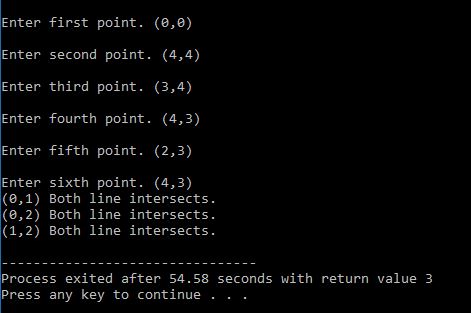
printf("(%d,%d) line doesn't intersect.\n",i,j);

}

}

}

**OUTPUT:**



1. **Modular Exponentiation**

**THEORY:**

Modular exponentiation is a type of exponentiation performed over a modulus. It is useful in computer science, especially in the field of public-key cryptography.

The operation of modular exponentiation calculates the remainder when an integer b (the base) raised to the eth power (the exponent), be, is divided by a positive integer m (the modulus). In symbols, given base b, exponent e, and modulus m, the modular exponentiation c is:

c ≡ b^e (mod m).

**CODE:**

#include <iostream>

#define ll long long

using namespace std;

ll modular\_pow(ll base, ll exponent, int modulus)

{

ll result = 1;

while (exponent > 0)

{

if (exponent % 2 == 1)

result = (result \* base) % modulus;

exponent = exponent >> 1;

base = (base \* base) % modulus;

}

return result;

}

int main()

{

ll x, y;

int mod;

cout<<"Enter Base Value: ";

cin>>x;

cout<<"Enter Exponent: ";

cin>>y;

cout<<"Enter Modular Value: ";

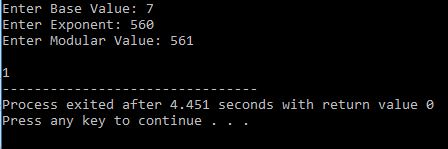
cin>>mod;

cout<<endl<<modular\_pow(x, y , mod);

return 0;

}

**OUTPUT:**



1. **KMP Pattern matching**

CODE:

#include<bits/stdc++.h>

using namespace std;

int main(){

string T,P;

cout<< "\nEnter The Text\n";

cin>>T;

cout<< "\nEnter The Pattern\n";

cin>>P;

int n=T.size(),m=P.size();

int pi[m];

pi[0]=0;

int k=0,i;

cout<<" pi :"<<pi[0];

for(i=1;i<m;i++){

while(k>0&&P[k]!=P[i])

k=pi[k-1];

if(P[k]==P[i])

k++; pi[i]=k;

cout<<pi[i];

}

cout<< endl;

int q=0;

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

while(q>0&&P[q]!=T[i])

q=pi[q-1];

if(P[q]==T[i])

q++;

if(q==m){

cout<<"\nPattern at shift : "<<i-m+1;

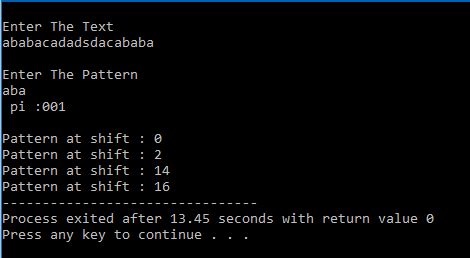
q=pi[q-1];

}

}

return 0;}

OUTPUT:



1. **Fast fourier transform**

CODE:

#include<bits/stdc++.h>

using namespace std;

typedef complex<double> Complex;

const double PI = 3.141592653589793238460;

int reverseBits(int num)

{

unsigned int count = sizeof(num) \* 8 - 1;

unsigned int reverse\_num = num;

num >>= 1;

while(num)

{

reverse\_num <<= 1;

reverse\_num |= num & 1;

num >>= 1;

count--;

}

reverse\_num <<= count;

return reverse\_num;

}

valarray<Complex> ifft(valarray<Complex> primal,int absP)

{

int i,j,k,p,n = primal.size(),offset;

valarray<Complex> dual(n);

for(i = 0;i<n;i++)

{

dual[i] = primal[reverseBits(i)>>(32-absP)];

}

for(p = 1;p<=absP;p++)

{

int unityStep = (1<<p);

double theta = 2\*M\_PI/unityStep;

Complex unityRoot(cos(theta),sin(theta));

for(offset = 0;offset<n;offset += unityStep)

{

Complex omega = 1;

for(k = 0;k<unityStep/2;k++)

{

Complex u = dual[offset+k];

Complex t = omega\*dual[offset+k+unityStep/2];

omega = omega\*unityRoot;

dual[offset+k] = u+t;

dual[offset+k+unityStep/2] = u-t;

}

}

}

return dual;

}

int main()

{

double n,i,j,k;

cout<<"Enter no. of coefficients\n";

cin>>n;

valarray<Complex> primal(n);

for(i = 0;i<n;i++)

{

primal[i] = 1+rand()%100;

}

int p = log2(n);

if(pow(2,p) != n)

{

cout<<"n should be power of 2\n";

exit(1);

}

valarray<Complex> y = ifft(primal,p);

for(i = 0;i<n;i++)

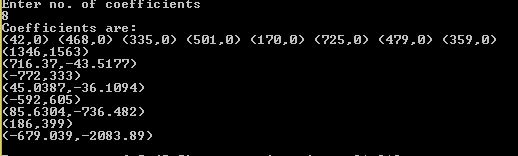
{

cout<<y[i]<<"\n";

}

}

OUTPUT:

****

1. **AVL Tree:**

CODE:

#include<stdio.h>

#include<stdlib.h>

#include<bits/stdc++.h>

using namespace std;

struct Node

{

int value;

struct Node \*left,\*right;

int height;

};

int height(struct Node \*N)

{

if (N == NULL)

return 0;

return N->height;

}

int max(int a, int b)

{

return (a > b)? a : b;

}

struct Node\* newNode(int key)

{

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->value = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return(node);

}

struct Node \* minValueNode(struct Node\* node)

{

struct Node\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

void preOrder(struct Node \*root)

{

if(root != NULL)

{

printf("%d ", root->value);

preOrder(root->left);

preOrder(root->right);

}

}

struct Node \*rightRotate(struct Node \*y)

{

struct Node \*x = y->left;

struct Node \*T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right))+1;

x->height = max(height(x->left), height(x->right))+1;

return x;

}

struct Node \*leftRotate(struct Node \*x)

{

struct Node \*y = x->right;

struct Node \*T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right))+1;

y->height = max(height(y->left), height(y->right))+1;

return y;

}

int getBalance(struct Node \*N)

{

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

struct Node\* insert(struct Node\* node, int key)

{

if (node == NULL)

return(newNode(key));

if (key < node->value)

node->left = insert(node->left, key);

else if (key > node->value)

node->right = insert(node->right, key);

else

return node;

node->height = 1 + max(height(node->left),height(node->right));

int balance = getBalance(node);

if (balance > 1 && key < node->left->value)

return rightRotate(node);

if (balance < -1 && key > node->right->value)

return leftRotate(node);

if (balance > 1 && key > node->left->value)

{

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && key < node->right->value)

{

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

struct Node\* deleteNode(struct Node\* root, int key)

{

if (root == NULL)

return root;

if ( key < root->value )

root->left = deleteNode(root->left, key);

else if( key > root->value )

root->right = deleteNode(root->right, key);

else

{

if( (root->left == NULL) || (root->right == NULL) )

{

struct Node \*temp = root->left ? root->left :root->right;

if (temp == NULL)

{

temp = root;

root = NULL;

}

else

\*root = \*temp;

free(temp);

}

else

{

struct Node\* temp = minValueNode(root->right);

root->value = temp->value;

root->right = deleteNode(root->right, temp->value);

}

}

if (root == NULL)

return root;

root->height = 1 + max(height(root->left),height(root->right));

int balance = getBalance(root);

if (balance > 1 && getBalance(root->left) >= 0)

return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0)

{

root->left = leftRotate(root->left);

return rightRotate(root);

}

if (balance < -1 && getBalance(root->right) <= 0)

return leftRotate(root);

if (balance < -1 && getBalance(root->right) > 0)

{

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

struct Node\* search(struct Node\* root, int key)

{

struct Node\* current=NULL;

if(root==NULL)

return NULL;

if(root->value==key)

return root;

else if(key < root->value)

current=search(root->left, key);

else if(key > root->value)

current=search(root->right, key);

return current;

}

int main()

{

struct Node \*root = NULL,\*temp;

int n,x,check=1; cout<<"Enter the nuber of nodes for the tree\n";

cin>>n; cout<<"\nEnter the nodes\n";

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

cin>>x;

root = insert(root, x);

}

printf("Preorder traversal of the constructed AVL tree is \n");

preOrder(root);

do{

cout<<"\nEnter the operation to perform: 1=Insert 2=Delete 3=Search: ";

cin>>x;

int y;

switch(x){

case 1:

cout<<"\n Enter the Number to insert: ";

cin>>y;

root = insert(root, y);

printf("Preorder traversal of the AVL tree after insertion is \n");

preOrder(root);

break;

case 2:

cout<<"\n Enter the Number to delete: ";

cin>>y;

root = deleteNode(root, y);

printf("\nPreorder traversal after deletion of %d is \n",y);

preOrder(root);

break;

case 3:

cout<<"\n Enter the Number to Search: ";

cin>>y;

temp=search(root, y);

if(temp)

cout<<"\nNode found at height: "<<(floor(log2(n))+1-height(temp));

else

cout<<"\nNode not found";

break;

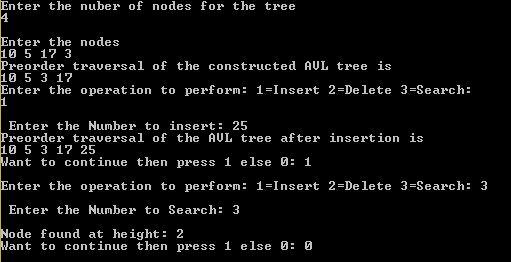
}

cout<<"\nWant to continue then press 1 else 0: ";

cin>>check;

}while(check);

return 0;}

****

1. **Binomial Heap**

CODE:

#include<stdio.h>

#include<malloc.h>

struct node

{

int n;

int degree;

struct node\* parent;

struct node\* child;

struct node\* sibling;

};

struct node\* MAKE\_bin\_HEAP();

int bin\_LINK(struct node\*,struct node\*);

struct node\* CREATE\_NODE(int);

struct node\* bin\_HEAP\_UNION(struct node\*,struct node\*);

struct node\* bin\_HEAP\_INSERT(struct node\*,struct node\*);

struct node\* bin\_HEAP\_MERGE(struct node\*,struct node\*);

struct node\* bin\_HEAP\_EXTRACT\_MIN(struct node\*);

int REVERT\_LIST(struct node\*);

int DISPLAY(struct node\*);

struct node\* FIND\_NODE(struct node\*,int);

int bin\_HEAP\_DECREASE\_KEY(struct node\*,int,int);

int bin\_HEAP\_DELETE(struct node\*,int);

int count=1;

struct node\* MAKE\_bin\_HEAP()

{

struct node\* np;

np=NULL;

return np;

}

struct node \* H=NULL;struct node \*Hr=NULL;

int bin\_LINK(struct node\* y,struct node\* z)

{

y->parent=z;

y->sibling=z->child;

z->child=y;

z->degree=z->degree+1;

}

struct node\* CREATE\_NODE(int k)

{

struct node\* p;//new node;

p=(struct node\*)malloc(sizeof(struct node));

p->n=k;

return p;

}

struct node\* bin\_HEAP\_UNION(struct node\* H1,struct node\* H2)

{

struct node\* prev\_x;

struct node\* next\_x;

struct node\* x;

struct node\* H=MAKE\_bin\_HEAP();

H=bin\_HEAP\_MERGE(H1,H2);

if(H==NULL)

return H;

prev\_x=NULL;

x=H;

next\_x=x->sibling;

while(next\_x!=NULL)

{

if((x->degree!=next\_x->degree)||((next\_x->sibling!=NULL)&&(next\_x->sibling)->degree==x->degree))

{

prev\_x=x;

x=next\_x;

}

else

{

if(x->n<=next\_x->n)

{

x->sibling=next\_x->sibling;

bin\_LINK(next\_x,x);

}

else

{

if(prev\_x==NULL)

H=next\_x;

else

prev\_x->sibling=next\_x;

bin\_LINK(x,next\_x);

x=next\_x;

}

}

next\_x=x->sibling;

}

return H;

}

struct node\* bin\_HEAP\_INSERT(struct node\* H,struct node\* x)

{

struct node\* H1=MAKE\_bin\_HEAP();

x->parent=NULL;

x->child=NULL;

x->sibling=NULL;

x->degree=0;

H1=x;

H=bin\_HEAP\_UNION(H,H1);

return H;

}

struct node\* bin\_HEAP\_MERGE(struct node\* H1,struct node\* H2)

{

struct node\* H=MAKE\_bin\_HEAP();

struct node\* y;

struct node\* z;

struct node\* a;

struct node\* b;

y=H1;

z=H2;

if(y!=NULL)

{

if(z!=NULL&&y->degree<=z->degree)

H=y;

else if(z!=NULL&&y->degree>z->degree)//need some modificationss here;the first and the else conditions can be merged together!!!!

H=z;

else

H=y;

}

else

H=z;

while(y!=NULL&&z!=NULL)

{

if(y->degree<z->degree)

{

y=y->sibling;

}

else if(y->degree==z->degree)

{

a=y->sibling;

y->sibling=z;

y=a;

}

else

{

b=z->sibling;

z->sibling=y;

z=b;

}

}

return H;

}

int DISPLAY(struct node\* H)

{

struct node\* p;

if(H==NULL)

{

printf("\nHEAP EMPTY");

return 0;

}

printf("\nTHE ROOT NODES ARE:-\n");

p=H;

while(p!=NULL)

{

printf("%d",p->n);

if(p->sibling!=NULL)

printf("-->");p=p->sibling;

}

printf("\n");

}

struct node\* bin\_HEAP\_EXTRACT\_MIN(struct node\* H1)

{

int min;

struct node\* t=NULL;

struct node\* x=H1;

struct node \*Hr;

struct node\* p;

Hr=NULL;

if(x==NULL)

{

printf("\nNOTHING TO EXTRACT");

return x;

}

// int min=x->n;

p=x;

while(p->sibling!=NULL)

{

if((p->sibling)->n<min)

{

min=(p->sibling)->n;

t=p;

x=p->sibling;

}

p=p->sibling;

}

if(t==NULL&&x->sibling==NULL)

H1=NULL;

else if(t==NULL)

H1=x->sibling;

else if(t->sibling==NULL)

t=NULL;

else

t->sibling=x->sibling;

if(x->child!=NULL)

{

REVERT\_LIST(x->child);

(x->child)->sibling=NULL;

}

H=bin\_HEAP\_UNION(H1,Hr);

return x;

}

int REVERT\_LIST(struct node\* y)

{

if(y->sibling!=NULL)

{

REVERT\_LIST(y->sibling);

(y->sibling)->sibling=y;

}

else

{

Hr=y;

}

}

struct node\* FIND\_NODE(struct node\* H,int k)

{

struct node\* x=H;

struct node\* p=NULL;

if(x->n==k)

{

p=x;

return p;

}

if(x->child!=NULL&&p==NULL)

{

p=FIND\_NODE(x->child,k);

}

if(x->sibling!=NULL&&p==NULL)

{

p=FIND\_NODE(x->sibling,k);

}

return p;

}

int bin\_HEAP\_DECREASE\_KEY(struct node\* H,int i,int k)

{

int temp;

struct node\* p;

struct node\* y;

struct node\* z;

p=FIND\_NODE(H,i);

if(p==NULL)

{

printf("\nINVALID CHOICE OF KEY TO BE REDUCED");

return 0;

}

if(k>p->n)

{

printf("\nSORY!THE NEW KEY IS GREATER THAN CURRENT ONE");

return 0;

}

p->n=k;

y=p;

z=p->parent;

while(z!=NULL&&y->n<z->n)

{

temp=y->n;

y->n=z->n;

z->n=temp;

y=z;

z=z->parent;

}

printf("\nKEY REDUCED SUCCESSFULLY!");

}

int bin\_HEAP\_DELETE(struct node\* H,int k)

{

struct node\* np;

if(H==NULL)

{

printf("\nHEAP EMPTY");

return 0;

}

bin\_HEAP\_DECREASE\_KEY(H,k,-1000);

np=bin\_HEAP\_EXTRACT\_MIN(H);

if(np!=NULL)

printf("\nNODE DELETED SUCCESSFULLY");

}

int main()

{

int i,n,m,l;

struct node\* p;

struct node\* np;

//struct node \*H;

char ch;

printf("\nENTER THE NUMBER OF ELEMENTS:");

scanf("%d",&n);

printf("\nENTER THE ELEMENTS:\n");

for(i=1;i<=n;i++)

{

scanf("%d",&m);

np=CREATE\_NODE(m);

H=bin\_HEAP\_INSERT(H,np);

}

DISPLAY(H);

do

{

printf("\nMENU:-\n");

printf("\n1)INSERT AN ELEMENT\n2)EXTRACT THE MINIMUM KEY NODE\n3)DECREASE A NODE KEY\n4)DELETE A NODE\n5)QUIT\n");

scanf("%d",&l);

switch(l)

{

case 1:do

{

printf("\nENTER THE ELEMENT TO BE INSERTED:");

scanf("%d",&m);

p=CREATE\_NODE(m);

H=bin\_HEAP\_INSERT(H,p);

printf("\nNOW THE HEAP IS:\n");

DISPLAY(H);

printf("\nINSERT MORE(y/Y)= \n");

fflush(stdin);

scanf("%c",&ch);

}while(ch=='Y'||ch=='y');

break;

case 2:do

{

printf("\nEXTRACTING THE MINIMUM KEY NODE");

p=bin\_HEAP\_EXTRACT\_MIN(H);

if(p!=NULL)

printf("\nTHE EXTRACTED NODE IS %d",p->n);

printf("\nNOW THE HEAP IS:\n");

DISPLAY(H);

printf("\nEXTRACT MORE(y/Y)\n");

fflush(stdin);

scanf("%c",&ch);

}while(ch=='Y'||ch=='y');

break;

case 3:do

{

printf("\nENTER THE KEY OF THE NODE TO BE DECREASED:");

scanf("%d",&m);

printf("\nENTER THE NEW KEY : ");

scanf("%d",&l);

bin\_HEAP\_DECREASE\_KEY(H,m,l);

printf("\nNOW THE HEAP IS:\n");

DISPLAY(H);

printf("\nDECREASE MORE(y/Y)\n");

fflush(stdin);

scanf("%c",&ch);

}while(ch=='Y'||ch=='y');

break;

case 4:do

{

printf("\nENTER THE KEY TO BE DELETED: ");

scanf("%d",&m);

bin\_HEAP\_DELETE(H,m);

printf("\nDELETE MORE(y/Y)\n");

fflush(stdin);

scanf("%c",&ch);

}while(ch=='y'||ch=='Y');

break;

case 5:printf("\nTHANK U SIR\n");break;

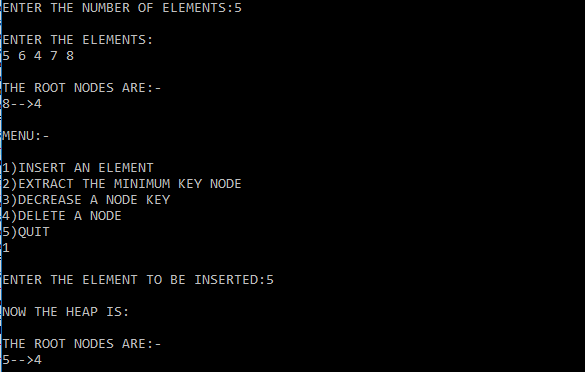
default :printf("\nINVALID ENTRY...TRY AGAIN....\n");

}

}while(l!=5);

}

OUTPUT:



1. **Fibonacci Heap**

CODE:

#include <iostream>

#include <cmath>

#include <cstdlib>

using namespace std;

/\*

\* Node Declaration

\*/

struct node

{

int n;

int degree;

node\* parent;

node\* child;

node\* left;

node\* right;

char mark;

char C;

};

/\*

\* Class Declaration

\*/

class FibonacciHeap

{

private:

int nH;

node \*H;

public:

node\* InitializeHeap();

int Fibonnaci\_link(node\*, node\*, node\*);

node \*Create\_node(int);

node \*Insert(node \*, node \*);

node \*Union(node \*, node \*);

node \*Extract\_Min(node \*);

int Consolidate(node \*);

int Display(node \*);

node \*Find(node \*, int);

int Decrease\_key(node \*, int, int);

int Delete\_key(node \*,int);

int Cut(node \*, node \*, node \*);

int Cascase\_cut(node \*, node \*);

FibonacciHeap()

{

H = InitializeHeap();

}

};

/\*

\* Initialize Heap

\*/

node\* FibonacciHeap::InitializeHeap()

{

node\* np;

np = NULL;

return np;

}

/\*

\* Create Node

\*/

node\* FibonacciHeap::Create\_node(int value)

{

node\* x = new node;

x->n = value;

return x;

}

/\*

\* Insert Node

\*/

node\* FibonacciHeap::Insert(node\* H, node\* x)

{

x->degree = 0;

x->parent = NULL;

x->child = NULL;

x->left = x;

x->right = x;

x->mark = 'F';

x->C = 'N';

if (H != NULL)

{

(H->left)->right = x;

x->right = H;

x->left = H->left;

H->left = x;

if (x->n < H->n)

H = x;

}

else

{

H = x;

}

nH = nH + 1;

return H;

}

/\*

\* Link Nodes in Fibonnaci Heap

\*/

int FibonacciHeap::Fibonnaci\_link(node\* H1, node\* y, node\* z)

{

(y->left)->right = y->right;

(y->right)->left = y->left;

if (z->right == z)

H1 = z;

y->left = y;

y->right = y;

y->parent = z;

if (z->child == NULL)

z->child = y;

y->right = z->child;

y->left = (z->child)->left;

((z->child)->left)->right = y;

(z->child)->left = y;

if (y->n < (z->child)->n)

z->child = y;

z->degree++;

}

/\*

\* Union Nodes in Fibonnaci Heap

\*/

node\* FibonacciHeap::Union(node\* H1, node\* H2)

{

node\* np;

node\* H = InitializeHeap();

H = H1;

(H->left)->right = H2;

(H2->left)->right = H;

np = H->left;

H->left = H2->left;

H2->left = np;

return H;

}

/\*

\* Display Fibonnaci Heap

\*/

int FibonacciHeap::Display(node\* H)

{

node\* p = H;

if (p == NULL)

{

cout<<"The Heap is Empty"<<endl;

return 0;

}

cout<<"The root nodes of Heap are: "<<endl;

do

{

cout<<p->n;

p = p->right;

if (p != H)

{

cout<<"-->";

}

}

while (p != H && p->right != NULL);

cout<<endl;

}

/\*

\* Extract Min Node in Fibonnaci Heap

\*/

node\* FibonacciHeap::Extract\_Min(node\* H1)

{

node\* p;

node\* ptr;

node\* z = H1;

p = z;

ptr = z;

if (z == NULL)

return z;

node\* x;

node\* np;

x = NULL;

if (z->child != NULL)

x = z->child;

if (x != NULL)

{

ptr = x;

do

{

np = x->right;

(H1->left)->right = x;

x->right = H1;

x->left = H1->left;

H1->left = x;

if (x->n < H1->n)

H1 = x;

x->parent = NULL;

x = np;

}

while (np != ptr);

}

(z->left)->right = z->right;

(z->right)->left = z->left;

H1 = z->right;

if (z == z->right && z->child == NULL)

H = NULL;

else

{

H1 = z->right;

Consolidate(H1);

}

nH = nH - 1;

return p;

}

/\*

\* Consolidate Node in Fibonnaci Heap

\*/

int FibonacciHeap::Consolidate(node\* H1)

{

int d, i;

float f = (log(nH)) / (log(2));

int D = f;

node\* A[D];

for (i = 0; i <= D; i++)

A[i] = NULL;

node\* x = H1;

node\* y;

node\* np;

node\* pt = x;

do

{

pt = pt->right;

d = x->degree;

while (A[d] != NULL)

{

y = A[d];

if (x->n > y->n)

{

np = x;

x = y;

y = np;

}

if (y == H1)

H1 = x;

Fibonnaci\_link(H1, y, x);

if (x->right == x)

H1 = x;

A[d] = NULL;

d = d + 1;

}

A[d] = x;

x = x->right;

}

while (x != H1);

H = NULL;

for (int j = 0; j <= D; j++)

{

if (A[j] != NULL)

{

A[j]->left = A[j];

A[j]->right =A[j];

if (H != NULL)

{

(H->left)->right = A[j];

A[j]->right = H;

A[j]->left = H->left;

H->left = A[j];

if (A[j]->n < H->n)

H = A[j];

}

else

{

H = A[j];

}

if(H == NULL)

H = A[j];

else if (A[j]->n < H->n)

H = A[j];

}

}

}

/\*

\* Decrease key of Nodes in Fibonnaci Heap

\*/

int FibonacciHeap::Decrease\_key(node\*H1, int x, int k)

{

node\* y;

if (H1 == NULL)

{

cout<<"The Heap is Empty"<<endl;

return 0;

}

node\* ptr = Find(H1, x);

if (ptr == NULL)

{

cout<<"Node not found in the Heap"<<endl;

return 1;

}

if (ptr->n < k)

{

cout<<"Entered key greater than current key"<<endl;

return 0;

}

ptr->n = k;

y = ptr->parent;

if (y != NULL && ptr->n < y->n)

{

Cut(H1, ptr, y);

Cascase\_cut(H1, y);

}

if (ptr->n < H->n)

H = ptr;

return 0;

}

/\*

\* Cut Nodes in Fibonnaci Heap

\*/

int FibonacciHeap::Cut(node\* H1, node\* x, node\* y)

{

if (x == x->right)

y->child = NULL;

(x->left)->right = x->right;

(x->right)->left = x->left;

if (x == y->child)

y->child = x->right;

y->degree = y->degree - 1;

x->right = x;

x->left = x;

(H1->left)->right = x;

x->right = H1;

x->left = H1->left;

H1->left = x;

x->parent = NULL;

x->mark = 'F';

}

/\*

\* Cascade Cutting in Fibonnaci Heap

\*/

int FibonacciHeap::Cascase\_cut(node\* H1, node\* y)

{

node\* z = y->parent;

if (z != NULL)

{

if (y->mark == 'F')

{

y->mark = 'T';

}

else

{

Cut(H1, y, z);

Cascase\_cut(H1, z);

}

}

}

/\*

\* Find Nodes in Fibonnaci Heap

\*/

node\* FibonacciHeap::Find(node\* H, int k)

{

node\* x = H;

x->C = 'Y';

node\* p = NULL;

if (x->n == k)

{

p = x;

x->C = 'N';

return p;

}

if (p == NULL)

{

if (x->child != NULL )

p = Find(x->child, k);

if ((x->right)->C != 'Y' )

p = Find(x->right, k);

}

x->C = 'N';

return p;

}

/\*

\* Delete Nodes in Fibonnaci Heap

\*/

int FibonacciHeap::Delete\_key(node\* H1, int k)

{

node\* np = NULL;

int t;

t = Decrease\_key(H1, k, -5000);

if (!t)

np = Extract\_Min(H);

if (np != NULL)

cout<<"Key Deleted"<<endl;

else

cout<<"Key not Deleted"<<endl;

return 0;

}

/\*

\* Main Contains Menu

\*/

int main()

{

int n, m, l;

FibonacciHeap fh;

node\* p;

node\* H;

H = fh.InitializeHeap();

while (1)

{

cout<<"----------------------------"<<endl;

cout<<"Operations on Binomial heap"<<endl;

cout<<"----------------------------"<<endl;

cout<<"1)Insert Element in the heap"<<endl;

cout<<"2)Extract Minimum key node"<<endl;

cout<<"3)Decrease key of a node"<<endl;

cout<<"4)Delete a node"<<endl;

cout<<"5)Display Heap"<<endl;

cout<<"6)Exit"<<endl;

cout<<"Enter Your Choice: ";

cin>>l;

switch(l)

{

case 1:

cout<<"Enter the element to be inserted: ";

cin>>m;

p = fh.Create\_node(m);

H = fh.Insert(H, p);

break;

case 2:

p = fh.Extract\_Min(H);

if (p != NULL)

cout<<"The node with minimum key: "<<p->n<<endl;

else

cout<<"Heap is empty"<<endl;

break;

case 3:

cout<<"Enter the key to be decreased: ";

cin>>m;

cout<<"Enter new key value: ";

cin>>l;

fh.Decrease\_key(H, m, l);

break;

case 4:

cout<<"Enter the key to be deleted: ";

cin>>m;

fh.Delete\_key(H, m);

break;

case 5:

cout<<"The Heap is: "<<endl;

fh.Display(H);

break;

case 6:

exit(1);

default:

cout<<"Wrong Choice"<<endl;

}

}

return 0;

}

OUTPUT:

