**Roll No…………….. Total No. of Pages:……**

**ST-5 (SET-II)**

**6th SEMESTER 2023-24**

**CS192- Advanced Data Structures**

**Time allowed: 90 Minutes Max. Marks: 40**

**General Instructions:**

* **Follow the instructions given in each section.**
* **Make sure that you attempt the questions in order.**

**SECTION-A (10\*1 mark=10 marks)**

***(All questions are compulsory)***

1. Which algorithm can be used to find the minimum spanning tree of a connected, undirected graph?
   1. Bellman-Ford algorithm
   2. Floyd-Warshall algorithm
   3. **Kruskal's algorithm**
   4. Dijkstra's algorithm
2. Topological sorting is only applicable to:
   1. **Directed acyclic graphs (DAGs)**
   2. Complete graphs
   3. Bipartite graphs
   4. Strongly connected components
3. Which algorithm can detect negative cycles in a graph?
   1. Kruskal's algorithm
   2. **Bellman-Ford algorithm**
   3. Dijkstra's algorithm
   4. Prim's algorithm
4. Prim's algorithm is used to find:
   1. Strongly Connected Components
   2. Shortest Path
   3. **Minimum Spanning Tree**
   4. Maximum Flow
5. What data structure is typically used to implement BFS?

a) Stack

**b) Queue**

c) Priority Queue

d) Hash Table

1. What is the main limitation of Dijkstra's algorithm when working with graphs containing negative weight edges?

a) It doesn't work with negative weights.

**b) It might enter an infinite loop.**

c) It will give incorrect results.

d) It becomes slower.

1. The Bellman-Ford algorithm works on graphs with:

**a) Negative weights**

b) Positive weights

c) No weights

d) Cycles

1. What is the maximum number of edges in a bipartite graph with 'n' vertices on one side and 'm' vertices on the other side?

a) n

b) m

c) n + m

**d) nm**

1. In a Trie, what is the purpose of a leaf node?
   1. To store the keys
   2. **To indicate the end of a valid key**
   3. To store metadata about the Trie
   4. To improve search performance
2. Which type of Trie optimizes space by merging nodes with a single child into a single node?
   1. **Compressed Trie**
   2. Radix Trie
   3. Suffix Trie
   4. Double Array Trie

**SECTION-B (5\*2 mark=10 marks)**

***(All questions are compulsory)***

11) #include <iostream>

#include <vector>

#include <stack>

using namespace std;

void DFS(vector<vector<int>>& graph, int start) {

stack<int> s;

s.push(start);

while (!s.empty()) {

int node = s.top();

s.pop();

cout << node << " ";

for (int neighbor : graph[node]) {

s.push(neighbor);

}

}

}

int main() {

int n = 5;

vector<vector<int>> graph(n + 1);

graph[1] = {2, 3};

graph[2] = {4, 5};

graph[3] = {};

graph[4] = {};

graph[5] = {};

DFS(graph, 1);

return 0;

}

What will be the output of the program?

a) 1 2 3 4 5

**b) 1 3 2 5 4**

c) 1 2 3

d) 1 2

12) What is the major missing in DFS snippet given below?

void Graph::DFS(int v)

{

cout << v << " ";

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFS(\*i);

}

a) Nothing

b) Recursive call

**c) Mark the visited vertex as true**

d) Check for unvisited vertex

13) #include <iostream>

#include <vector>

#include<climits>

using namespace std;

void FloydWarshall(vector<vector<int>>& graph) {

int n = graph.size();

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

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

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

if (graph[i][k] != INT\_MAX && graph[k][j] != INT\_MAX && graph[i][k] + graph[k][j] < graph[i][j]) {

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

}

}

}

}

}

int main() {

int n = 3;

vector<vector<int>> graph(n, vector<int>(n, INT\_MAX));

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

graph[i][i] = 0;

}

graph[0][1] = 3;

graph[1][2] = -2;

graph[2][0] = 7;

FloydWarshall(graph);

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

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

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

}

cout << endl;

}

return 0;

}

What will be the output of the program?

a) 0 3 1

7 0 -2

5 8 0

b) 0 3 -2

5 0 -4

7 10 0

**c) 0 3 1**

**5 0 -2**

**7 10 0**

d) 0 3 1

7 0 -2

5 10 0

14) Consider the following adjacency matrix for a directed graph:

a b c

a 0 ∞ -2

b 3 0 ∞

c ∞ 1 0

After applying the Floyd-Warshall algorithm, what will be the distance from vertex 'b' to vertex 'c'?

a) 0

b) -2

**c) 1**

d) ∞

15) The degree sequence of a simple graph is the sequence of the degrees of the nodes in the graph in decreasing order. Which of the following sequences can not be the degree sequence of any graph?

I. 7, 6, 5, 4, 4, 3, 2, 1

II. 6, 6, 6, 6, 3, 3, 2, 2

III. 7, 6, 6, 4, 4, 3, 2, 2

IV. 8, 7, 7, 6, 4, 2, 1, 1

a) I and II

b) III and IV

c) IV only

**d) II and IV**

**SECTION-C(Coding Question) (2x5 marks=5 marks)**

Q16) Given a directed graph, find out whether the graph is strongly connected or not. A directed graph is strongly connected if there is a path between any two pair of vertices. ((Kosaraju using DFS)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | graph=[ (0, 1),(1, 2),(2, 3),(3, 0),(2, 4),(4, 2) ] | graph=[(0, 1),(1, 2),(2, 3)] | graph=[(0, 1),(1, 2),(2, 0)] |
| **Output** | Yes | No | Yes |

Solution :

**#include <iostream>**

**#include <list>**

**#include <stack>**

**using namespace std;**

**class Graph**

**{**

**int V; // No. of vertices**

**list<int> \*adj; // An array of adjacency lists**

**// A recursive function to print DFS starting from v**

**void DFSUtil(int v, bool visited[]);**

**public:**

**// Constructor and Destructor**

**Graph(int V) { this->V = V; adj = new list<int>[V];}**

**~Graph() { delete [] adj; }**

**// Method to add an edge**

**void addEdge(int v, int w);**

**// The main function that returns true if the**

**// graph is strongly connected, otherwise false**

**bool isSC();**

**// Function that returns reverse (or transpose)**

**// of this graph**

**Graph getTranspose();**

**};**

**// A recursive function to print DFS starting from v**

**void Graph::DFSUtil(int v, bool visited[])**

**{**

**// Mark the current node as visited and print it**

**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])**

**DFSUtil(\*i, visited);**

**}**

**// Function that returns reverse (or transpose) of this graph**

**Graph Graph::getTranspose()**

**{**

**Graph g(V);**

**for (int v = 0; v < V; v++)**

**{**

**// Recur for all the vertices adjacent to this vertex**

**list<int>::iterator i;**

**for(i = adj[v].begin(); i != adj[v].end(); ++i)**

**{**

**g.adj[\*i].push\_back(v);**

**}**

**}**

**return g;**

**}**

**void Graph::addEdge(int v, int w)**

**{**

**adj[v].push\_back(w); // Add w to v’s list.**

**}**

**// The main function that returns true if graph is strongly connected**

**bool Graph::isSC()**

**{**

**// St1p 1: Mark all the vertices as not visited**

**// (For first DFS)**

**bool visited[V];**

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

**visited[i] = false;**

**// Step 2: Do DFS traversal starting from first vertex.**

**DFSUtil(0, visited);**

**// If DFS traversal doesn’t visit all vertices, then return false.**

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

**if (visited[i] == false)**

**return false;**

**// Step 3: Create a reversed graph**

**Graph gr = getTranspose();**

**// Step 4: Mark all the vertices as not visited (For second DFS)**

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

**visited[i] = false;**

**// Step 5: Do DFS for reversed graph starting from first vertex. Starting Vertex must be same starting point of first DFS**

**gr.DFSUtil(0, visited);**

**// If all vertices are not visited in second DFS, then return false**

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

**if (visited[i] == false)**

**return false;**

**return true;**

**}**

**int main()**

**{**

**// Create graphs given in the above diagrams**

**Graph g1(5);**

**g1.addEdge(0, 1);**

**g1.addEdge(1, 2);**

**g1.addEdge(2, 3);**

**g1.addEdge(3, 0);**

**g1.addEdge(2, 4);**

**g1.addEdge(4, 2);**

**g1.isSC()? cout << "Yes\n" : cout << "No\n";**

**Graph g2(4);**

**g2.addEdge(0, 1);**

**g2.addEdge(1, 2);**

**g2.addEdge(2, 3);**

**g2.isSC()? cout << "Yes\n" : cout << "No\n";**

**return 0;**

**}**

Q17) Write a C++ program for insert and search operations on Trie tree.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | keys[] = {"the", "a", "there","answer", "any", "by","bye", "their" }  searchkey= the | keys[] = {"the", "a", "there","answer", "any", "by","bye", "their" }  searchkey= these | keys[] = {"the", "a", "there","answer", "any", "by","bye", "their" }  searchkey= their |
| **Output** | the --- Present in trie | these --- Not present in trie | their --- Present in trie |

Solution :

**#include <bits/stdc++.h>**

**using namespace std;**

**const int ALPHABET\_SIZE = 26;**

**// trie node**

**struct TrieNode**

**{**

**struct TrieNode \*children[ALPHABET\_SIZE];**

**// isEndOfWord is true if the node represents**

**// end of a word**

**bool isEndOfWord;**

**};**

**// Returns new trie node (initialized to NULLs)**

**struct TrieNode \*getNode(void)**

**{**

**struct TrieNode \*pNode = new TrieNode;**

**pNode->isEndOfWord = false;**

**for (int i = 0; i < ALPHABET\_SIZE; i++)**

**pNode->children[i] = NULL;**

**return pNode;**

**}**

**// If not present, inserts key into trie**

**// If the key is prefix of trie node, just marks leaf node**

**void insert(struct TrieNode \*root, string key)**

**{**

**struct TrieNode \*pCrawl = root;**

**for (int i = 0; i < key.length(); i++)**

**{**

**int index = key[i] - 'a';**

**if (!pCrawl->children[index])**

**pCrawl->children[index] = getNode();**

**pCrawl = pCrawl->children[index];**

**}**

**// mark last node as leaf**

**pCrawl->isEndOfWord = true;**

**}**

**// Returns true if key presents in trie, else false**

**bool search(struct TrieNode \*root, string key)**

**{**

**struct TrieNode \*pCrawl = root;**

**for (int i = 0; i < key.length(); i++)**

**{**

**int index = key[i] - 'a';**

**if (!pCrawl->children[index])**

**return false;**

**pCrawl = pCrawl->children[index];**

**}**

**return (pCrawl->isEndOfWord);**

**}**

**int main()**

**{**

**// Input keys (use only 'a' through 'z' and lower case)**

**string keys[] = {"the", "a", "there",**

**"answer", "any", "by",**

**"bye", "their" };**

**int n = sizeof(keys)/sizeof(keys[0]);**

**struct TrieNode \*root = getNode();**

**// Construct trie**

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

**insert(root, keys[i]);**

**// Search for different keys**

**char output[][32] = {"Not present in trie", "Present in trie"};**

**// Search for different keys**

**cout<<"the"<<" --- "<<output[search(root, "the")]<<endl;**

**cout<<"these"<<" --- "<<output[search(root, "these")]<<endl;**

**cout<<"their"<<" --- "<<output[search(root, "their")]<<endl;**

**cout<<"thaw"<<" --- "<<output[search(root, "thaw")]<<endl;**

**return 0;**

**}**

**SECTION-D (Coding Question)(1x10 mark=10 mark)**

Q18) Given a graph and two nodes the task is to print the shortest path between two nodes using the Floyd Warshall algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | node1=1, node2=3 | node1=0, node2=2 | node1=3, node2=2 |
| **Output** | Shortest path from 1 to 3: 1 -> 2 -> 3 | Shortest path from 0 to 2: 0 -> 1 -> 2 | Shortest path from 3 to 2: 3 -> 0 -> 1 -> 2 |

Solution :

**#include <bits/stdc++.h>**

**using namespace std;**

**#define MAXN 100**

**// Infinite value for array**

**const int INF = 1e7;**

**int dis[MAXN][MAXN];**

**int Next[MAXN][MAXN];**

**// Initializing the distance and Next array**

**void initialise(int V, vector<vector<int> >& graph)**

**{**

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

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

**dis[i][j] = graph[i][j];**

**// No edge between node i and j**

**if (graph[i][j] == INF)**

**Next[i][j] = -1;**

**else**

**Next[i][j] = j;**

**}**

**}**

**}**

**// Function construct the shortest path between u and v**

**vector<int> constructPath(int u, int v)**

**{**

**// If there's no path between node u and v, simply return an empty array**

**if (Next[u][v] == -1)**

**return {};**

**// Storing the path in a vector**

**vector<int> path = { u };**

**while (u != v) {**

**u = Next[u][v];**

**path.push\_back(u);**

**}**

**return path;**

**}**

**// Standard Floyd Warshall Algorithm with little modification Now if we find**

**// that dis[i][j] > dis[i][k] + dis[k][j] then we modify next[i][j] = next[i][k]**

**void floydWarshall(int V)**

**{**

**for (int k = 0; k < V; k++) {**

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

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

**// We cannot travel through**

**// edge that doesn't exist**

**if (dis[i][k] == INF|| dis[k][j] == INF){**

**continue;**

**}**

**if (dis[i][j] > dis[i][k] + dis[k][j]) {**

**dis[i][j] = dis[i][k]+ dis[k][j];**

**Next[i][j] = Next[i][k];**

**}**

**}**

**}**

**}**

**}**

**// Print the shortest path**

**void printPath(vector<int>& path)**

**{**

**int n = path.size();**

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

**cout << path[i] << " -> ";**

**cout << path[n - 1] << endl;**

**}**

**int main()**

**{**

**int V = 4;**

**vector<vector<int> > graph**

**= { { 0, 3, INF, 7 },**

**{ 8, 0, 2, INF },**

**{ 5, INF, 0, 1 },**

**{ 2, INF, INF, 0 } };**

**// Function to initialise the distance and Next array**

**initialise(V, graph);**

**// Calling Floyd Warshall Algorithm, this will update the shortest distance as well as Next array**

**floydWarshall(V);**

**vector<int> path;**

**// Path from node 1 to 3**

**cout << "Shortest path from 1 to 3: ";**

**path = constructPath(1, 3);**

**printPath(path);**

**// Path from node 0 to 2**

**cout << "Shortest path from 0 to 2: ";**

**path = constructPath(0, 2);**

**printPath(path);**

**// path from node 3 to 2**

**cout << "Shortest path from 3 to 2: ";**

**path = constructPath(3, 2);**

**printPath(path);**

**return 0;**

**}**