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

**ST-5 (SET-IV)**

**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. What is the maximum number of edges in a connected acyclic undirected graph with n vertices?
   1. **n - 1**
   2. n
   3. 2n - 2
   4. (n \* (n - 1)) / 2
2. A graph in which every vertex is connected to every other vertex is called a:
   1. **Complete graph**
   2. Bipartite graph
   3. Cycle graph
   4. Tree
3. Which of the following is NOT a graph traversal algorithm?
   1. Breadth-First Search (BFS)
   2. Depth-First Search (DFS)
   3. **Dijkstra's algorithm**
   4. Depth-Limited Search (DLS)
4. In DFS, the data structure typically used for maintaining the vertices is:
   1. **Stack**
   2. Queue
   3. Linked List
   4. Array
5. In DFS, which vertex is explored first in each iteration?

a) The vertex with the lowest index.

b) The vertex with the highest index.

**c) Any unvisited vertex adjacent to the current one.**

d) The vertex with the maximum degree.

1. What is the time complexity of Dijkstra's algorithm when implemented using a binary heap priority queue?

a) O(V + E)

b) O(V log V)

**c) O(E log V)**

d) O(V^2)

1. The Floyd-Warshall algorithm is used to:

a) Find the shortest path in a weighted directed graph

b) Find the longest path in an unweighted graph

c) Detect cycles in a directed graph

**d) Find all-pairs shortest paths in a weighted graph**

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 the root node?
   1. To store the first character of all keys
   2. To improve search performance
   3. To indicate the end of a valid key
   4. **To serve as the entry point of the Trie**
2. What is the time complexity for deleting a key from a Trie with 'k' characters?
   1. **O(k)**
   2. O(log k)
   3. O(n)
   4. O(1)

**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 = 4;

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

graph[1] = {2, 3};

graph[2] = {4};

graph[3] = {};

graph[4] = {3};

DFS(graph, 1);

return 0;

}

What will be the output of the program?

a) 1 2 3 4

**b) 1 3 2 4**

c) 1 2 3

d) 1 2

12) #include <iostream>

#include <vector>

#include <queue>

using namespace std;

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

int n = graph.size();

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

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

dist[start] = 0;

pq.push({0, start});

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

for (auto edge : graph[u]) {

int v = edge.first;

int weight = edge.second;

if (dist[u] + weight < dist[v]) {

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

pq.push({dist[v], v});

}

}

}

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

cout << dist[i] << " ";

}

}

int main() {

int n = 5;

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

graph[1].push\_back({2, 10});

graph[2].push\_back({3, 5});

graph[1].push\_back({3, 3});

graph[3].push\_back({4, 2});

graph[2].push\_back({4, 7});

graph[3].push\_back({5, 15});

graph[4].push\_back({5, 10});

Dijkstra(graph, 1);

return 0;

}

What will be the output of the program?

**a) 0 10 3 5 15**

b) 0 10 5 7 15

c) 0 5 3 2 10

d) 0 10 3 2 10

13) #include <iostream>

#include <vector>

#include <climits>

using namespace std;

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

int n = graph.size();

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

dist[start] = 0;

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

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

for (auto edge : graph[u]) {

int v = edge.first;

int weight = edge.second;

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

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

}

}

}

}

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

cout << dist[i] << " ";

}

}

int main() {

int n = 5;

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

graph[1].push\_back({2, 6});

graph[1].push\_back({3, 5});

graph[2].push\_back({4, -2});

graph[3].push\_back({2, -2});

graph[2].push\_back({3, 7});

graph[3].push\_back({5, 3});

graph[4].push\_back({5, 1});

BellmanFord(graph, 1);

return 0;

}

What will be the output of the program?

a) 0 3 5 -2 0

b) 0 6 5 -2 0

**c) 0 3 5 1 2**

d) 0 6 7 2 5

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

a b c

a 0 -1 ∞

b 3 0 ∞

c 2 ∞ 0

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

a) 0

**b) -1**

c) 2

d) ∞

15) Which of the following statements is/are TRUE for an undirected graph?

P: The number of odd-degree vertices is even

Q: Sum of degrees of all vertices is even

a) P Only

b) Q Only

**c) Both P and Q**

d) Neither P nor Q

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

Q16) Given an undirected graph, The task is to check if there is a cycle in the given graph.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | N = 4, E = 3 , 0 1, 1 2, 2 3 | N = 4, E = 4 , 0 1, 1 2, 2 3 ,0 2 | N = 3, E = 2 , 0 1, 1 2 |
| **Output** | No | Yes | No |

Solution :

**#include <iostream>**

**#include <limits.h>**

**#include <list>**

**using namespace std;**

**// Class for an undirected graph**

**class Graph {**

**// No. of vertices**

**int V;**

**// Pointer to an array containing adjacency lists**

**list<int>\* adj;**

**bool isCyclicUtil(int v, bool visited[], int parent);**

**public:**

**// Constructor**

**Graph(int V);**

**// To add an edge to graph**

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

**// Returns true if there is a cycle**

**bool isCyclic();**

**};**

**Graph::Graph(int V)**

**{**

**this->V = V;**

**adj = new list<int>[V];**

**}**

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

**{**

**// Add w to v’s list.**

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

**// Add v to w’s list.**

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

**}**

**// A recursive function that uses visited[] and parent to detect**

**// cycle in subgraph reachable from vertex v.**

**bool Graph::isCyclicUtil(int v, bool visited[], int parent)**

**{**

**// 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 an adjacent vertex is not visited,**

**// then recur for that adjacent**

**if (!visited[\*i]) {**

**if (isCyclicUtil(\*i, visited, v))**

**return true;**

**}**

**// If an adjacent vertex is visited and is not parent of current vertex,**

**// then there exists a cycle in the graph.**

**else if (\*i != parent)**

**return true;**

**}**

**return false;**

**}**

**// Returns true if the graph contains a cycle, else false.**

**bool Graph::isCyclic()**

**{**

**// Mark all the vertices as not visited and not part of recursion stack**

**bool\* visited = new bool[V];**

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

**visited[i] = false;**

**// Call the recursive helper function to detect cycle in different DFS trees**

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

**// Don't recur for u if it is already visited**

**if (!visited[u])**

**if (isCyclicUtil(u, visited, -1))**

**return true;**

**}**

**return false;**

**}**

**int main()**

**{**

**Graph g1(5);**

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

**g1.addEdge(0, 2);**

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

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

**g1.addEdge(3, 4);**

**g1.isCyclic() ? cout << "Graph contains cycle\n": cout << "Graph doesn't contain cycle\n";**

**Graph g2(3);**

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

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

**g2.isCyclic() ? cout << "Graph contains cycle\n": cout << "Graph doesn't contain cycle\n";**

**return 0;**

**}**

Q17) Given a string, count all distinct substrings of the given string using trie tree.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | abcd | Aaa | pixel |
| **Output** | Count of Distinct Substrings: 10 | Count of Distinct Substrings: 3 | Count of Distinct Substrings: 21 |

Solution :

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

**using namespace std;**

**class TrieNode {**

**public:**

**bool isWord;**

**TrieNode\* child[26];**

**TrieNode()**

**{**

**isWord = 0;**

**for (int i = 0; i < 26; i++) {**

**child[i] = 0;**

**}**

**}**

**};**

**int countDistinctSubstring(string str)**

**{**

**TrieNode\* head = new TrieNode();**

**// will hold the count of unique substrings**

**int count = 0;**

**// included count of substr " "**

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

**TrieNode\* temp = head;**

**for (int j = i; j < str.length(); j++) {**

**// when char not present add it to the trie**

**if (temp->child[str[j] - 'a'] == NULL) {**

**temp->child[str[j] - 'a'] = new TrieNode();**

**temp->isWord = 1;**

**count++;**

**}**

**// move on to the next char**

**temp = temp->child[str[j] - 'a'];**

**}**

**}**

**return count;**

**}**

**int main()**

**{**

**int count = countDistinctSubstring("aaabc");**

**cout << "Count of Distinct Substrings: " << count**

**<< endl;**

**return 0;**

**}**

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

Q18) Write a C++ program to find articulation points in an undirected graph

Note: A vertex v is an articulation point (also called cut vertex) if removing v increases the number of connected components.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | Graph= (1, 2),( 2, 3),(1, 3),(3, 4),(4, 5) | Graph= (1,0),(0, 2),(2, 1),(0, 3),(3, 4) | Graph= (1, 2),( 2, 3),(1, 2) |
| **Output** | Articulation points in the graph 3 4 | Articulation points in the graph 0 3 | Articulation points in the graph 1 2 |

Solution :

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

**using namespace std;**

**/\* A recursive function that find articulation points using DFS traversal**

**adj[] --> Adjacency List representation of the graph u --> The vertex to be visited next**

**visited[] --> keeps track of visited vertices**

**disc[] --> Stores discovery times of visited vertices**

**low[] -- >> earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with current vertex**

**parent --> Stores the parent vertex in DFS tree**

**isAP[] --> Stores articulation points**

**\*/**

**void APUtil(vector<int> adj[], int u, bool visited[], int disc[], int low[], int& time, int parent,bool isAP[])**

**{**

**// Count of children in DFS Tree**

**int children = 0;**

**// Mark the current node as visited**

**visited[u] = true;**

**// Initialize discovery time and low value**

**disc[u] = low[u] = ++time;**

**// Go through all vertices adjacent to this**

**for (auto v : adj[u]) {**

**// If v is not visited yet, then make it a child of u in DFS tree and recur for it**

**if (!visited[v]) {**

**children++;**

**APUtil(adj, v, visited, disc, low, time, u, isAP);**

**// Check if the subtree rooted with v has a connection to one of the ancestors of u**

**low[u] = min(low[u], low[v]);**

**// If u is not root and low value of one of its child is more than discovery value of u.**

**if (parent != -1 && low[v] >= disc[u])**

**isAP[u] = true;**

**}**

**// Update low value of u for parent function calls.**

**else if (v != parent)**

**low[u] = min(low[u], disc[v]);**

**}**

**// If u is root of DFS tree and has two or more children.**

**if (parent == -1 && children > 1)**

**isAP[u] = true;**

**}**

**void AP(vector<int> adj[], int V)**

**{**

**int disc[V] = { 0 };**

**int low[V];**

**bool visited[V] = { false };**

**bool isAP[V] = { false };**

**int time = 0, par = -1;**

**// Adding this loop so that the code works even if we are given disconnected graph**

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

**if (!visited[u])**

**APUtil(adj, u, visited, disc, low,**

**time, par, isAP);**

**// Printing the APs**

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

**if (isAP[u] == true)**

**cout << u << " ";**

**}**

**// Utility function to add an edge**

**void addEdge(vector<int> adj[], int u, int v)**

**{**

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

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

**}**

**int main()**

**{**

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

**cout << "Articulation points in first graph \n";**

**int V = 5;**

**vector<int> adj1[V];**

**addEdge(adj1, 1, 0);**

**addEdge(adj1, 0, 2);**

**addEdge(adj1, 2, 1);**

**addEdge(adj1, 0, 3);**

**addEdge(adj1, 3, 4);**

**AP(adj1, V);**

**cout << "\nArticulation points in second graph \n";**

**V = 4;**

**vector<int> adj2[V];**

**addEdge(adj2, 0, 1);**

**addEdge(adj2, 1, 2);**

**addEdge(adj2, 2, 3);**

**AP(adj2, V);**

**cout << "\nArticulation points in third graph \n";**

**V = 7;**

**vector<int> adj3[V];**

**addEdge(adj3, 0, 1);**

**addEdge(adj3, 1, 2);**

**addEdge(adj3, 2, 0);**

**addEdge(adj3, 1, 3);**

**addEdge(adj3, 1, 4);**

**addEdge(adj3, 1, 6);**

**addEdge(adj3, 3, 5);**

**addEdge(adj3, 4, 5);**

**AP(adj3, V);**

**return 0;**

**}**