**EXPERIMENT NO. 12**

| **Objective(s):**  Implement Graph data structure using adjacency list and adjacency matrix representations, and perform basic operations including BFS and DFS traversals. |
| --- |
| **Outcome:**  Develop methods to efficiently store graph data, allowing traversal algorithms (BFS and DFS) to explore nodes and edges, facilitating various graph-related computations and analyses. |
| **Problem Statement:**  Implement Graphs and represent using adjaceny list and adjacency matrix and implement basic operations with traversals |
| **Background Study:** Graphs A **graph** is a non-linear data structure that consists of a set of **vertices** (also called nodes) and a set of **edges** that connect pairs of vertices. Graphs are widely used to represent networks of interconnected objects, such as social networks, road networks, computer networks, etc. Types of Graphs  1. **Undirected Graph:**    * A graph where edges have no direction. If there is an edge between vertex A and vertex B, it can be traversed in both directions. 2. **Directed Graph (Digraph):**    * A graph where edges have a direction. If there is a directed edge from vertex A to vertex B, it can only be traversed from A to B, not from B to A unless there is a separate edge in the opposite direction. 3. **Weighted Graph:**    * A graph where edges are assigned weights or costs. These weights can represent distances, capacities, costs, etc. 4. **Unweighted Graph:**    * A graph where all edges have the same weight, often represented simply by their existence.  Graph Representation Graphs can be represented in two primary ways:   1. **Adjacency Matrix:**    * A 2D array graph[V][V] where V is the number of vertices.    * graph[i][j] indicates the presence of an edge between vertex i and vertex j.    * For unweighted graphs, graph[i][j] is typically 1 if there is an edge, and 0 otherwise.    * For weighted graphs, graph[i][j] may contain the weight of the edge, or a special value (like INT\_MAX) to denote absence of an edge. 2. **Adjacency List:**    * An array of lists, where each element of the array represents a vertex.    * Each vertex v has a list of vertices adjacent to it (neighbors).    * This representation is more space-efficient for sparse graphs (where there are few edges compared to the number of vertices).    * Allows efficient insertion and removal of edges.  Basic Operations on Graphs  1. **Traversal:**    * **Depth-First Search (DFS):**      + Start from a vertex and explore as far as possible along each branch before backtracking.      + Uses a stack (or recursion) to maintain the sequence of visited vertices.      + Useful for finding connected components, topological sorting, and detecting cycles in graphs.    * **Breadth-First Search (BFS):**      + Start from a vertex and explore all its neighbors at the present depth level before moving on to vertices at the next depth level.      + Uses a queue to maintain the sequence of visited vertices.      + Useful for finding shortest paths in unweighted graphs, level-order traversal, and network broadcasting. 2. **Adding and Removing Vertices and Edges:**    * Vertices are typically added or removed straightforwardly by adjusting the adjacency matrix or list.    * Adding or removing edges involves updating the appropriate entry in the matrix or adjusting the adjacency list. 3. **Finding Shortest Paths:**    * Algorithms like Dijkstra's algorithm (for non-negative weights) or Bellman-Ford algorithm (for negative weights) are used to find the shortest path between two vertices in a weighted graph. 4. **Cycle Detection:**    * Techniques such as DFS can be used to detect cycles in directed and undirected graphs by maintaining a list of visited vertices and checking for back edges.  Applications of Graphs  * **Routing and Network Design:** Used in computer networks and telecommunications for routing packets and designing optimal network topologies. * **Social Network Analysis:** Analyzing relationships and connections between individuals or entities in social networks. * **Recommendation Systems:** Understanding user preferences and relationships to recommend items or content. * **Transportation and Logistics:** Optimizing routes for vehicles, scheduling deliveries, and managing traffic flow. |

| **Algorithm :**  **1. Graph Representation**   * **Adjacency Matrix:**   + Create a 2D array graph[V][V] where V is the number of vertices.   + Initialize all elements to 0 or INF (infinity) depending on whether the graph is weighted or unweighted.   + Set graph[i][j] to 1 or to the weight of the edge (i, j) for an unweighted or weighted graph respectively.   + Adjust entries based on adding or removing edges. * **Adjacency List:**   + Create an array of linked lists (or dynamic arrays) where each array index represents a vertex.   + For each vertex v, maintain a list of vertices adjacent to v.   + Insertion and deletion of edges involve appending or removing elements from these lists.  2. Basic Operations  * **Traversal:**    1. **Depth-First Search (DFS):**      + Start from a vertex v.      + Mark v as visited.      + Recursively visit all adjacent vertices not yet visited.      + Use a stack (or recursion) to maintain the traversal path.   2. **Breadth-First Search (BFS):**      + Start from a vertex v.      + Mark v as visited.      + Use a queue to store vertices at the current level.      + Dequeue a vertex, visit its neighbors, and enqueue them.      + Repeat until the queue is empty. * **Adding and Removing Vertices and Edges:**    1. **Adding a Vertex:**      + Increase the size of the adjacency list or matrix and initialize its entries.   2. **Removing a Vertex:**      + Remove the vertex from the adjacency list or matrix, along with its edges.   3. **Adding an Edge:**      + Update the corresponding entries in the adjacency list or matrix.   4. **Removing an Edge:**      + Set the corresponding entries in the adjacency list or matrix to 0 or INF. * **Finding Shortest Paths:**    1. Use algorithms like Dijkstra's (for non-negative weights) or Bellman-Ford (for negative weights) to find the shortest path between two vertices. * **Algorithm Steps:**    1. **Initialization:**       + Create a distance array dist[] where dist[s] = 0 and dist[v] = INF for all vertices v ≠ s, indicating the shortest known distance from the source vertex s to each vertex.      + Maintain a set of vertices S whose shortest path from the source vertex s has been found.   2. **Process:**       + While S does not include all vertices:        - Select the vertex u not in S with the smallest distance dist[u].        - Add u to S.        - Update the distances of all neighboring vertices v of u if dist[u] + weight(u, v) < dist[v].        - Continue until all vertices have been added to S.   3. **Output:**       + The array dist[] now contains the shortest path distances from the source vertex s to every other vertex in the graph. |
| --- |

| **Code:**  #include <stdio.h>  #include <stdlib.h>  // Structure to represent a node in adjacency list  struct AdjListNode {  int dest;  struct AdjListNode\* next;  };  // Structure to represent an adjacency list  struct AdjList {  struct AdjListNode\* head; // Pointer to head node of the list  };  // Structure to represent a graph  struct Graph {  int V; // Number of vertices  struct AdjList\* array; // Array of adjacency lists  };  // Function to create a new adjacency list node  struct AdjListNode\* newAdjListNode(int dest) {  struct AdjListNode\* newNode = (struct AdjListNode\*)malloc(sizeof(struct AdjListNode));  if (newNode == NULL) {  fprintf(stderr, "Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  newNode->dest = dest;  newNode->next = NULL;  return newNode;  }  // Function to create a graph with V vertices  struct Graph\* createGraph(int V) {  struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));  if (graph == NULL) {  fprintf(stderr, "Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  graph->V = V;  // Create an array of adjacency lists. Size of array will be V  graph->array = (struct AdjList\*)malloc(V \* sizeof(struct AdjList));  if (graph->array == NULL) {  fprintf(stderr, "Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  // Initialize each adjacency list as empty by making head as NULL  for (int i = 0; i < V; ++i) {  graph->array[i].head = NULL;  }  return graph;  }  // Function to add an edge to an undirected graph  void addEdge(struct Graph\* graph, int src, int dest) {  // Add an edge from src to dest  struct AdjListNode\* newNode = newAdjListNode(dest);  newNode->next = graph->array[src].head;  graph->array[src].head = newNode;  // Since graph is undirected, add an edge from dest to src also  newNode = newAdjListNode(src);  newNode->next = graph->array[dest].head;  graph->array[dest].head = newNode;  }  // Function to print the adjacency list representation of the graph  void printGraph(struct Graph\* graph) {  for (int v = 0; v < graph->V; ++v) {  struct AdjListNode\* temp = graph->array[v].head;  printf("Adjacency list of vertex %d\n head", v);  while (temp) {  printf(" -> %d", temp->dest);  temp = temp->next;  }  printf("\n");  }  }  // Structure for stack  struct Stack {  int top;  unsigned capacity;  int\* array;  };  // Function to create a stack of given capacity  struct Stack\* createStack(unsigned capacity) {  struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));  if (stack == NULL) {  fprintf(stderr, "Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  stack->capacity = capacity;  stack->top = -1;  stack->array = (int\*)malloc(stack->capacity \* sizeof(int));  if (stack->array == NULL) {  fprintf(stderr, "Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  return stack;  }  // Stack functions  int isStackEmpty(struct Stack\* stack) {  return stack->top == -1;  }  void push(struct Stack\* stack, int item) {  stack->array[++stack->top] = item;  }  int pop(struct Stack\* stack) {  if (!isStackEmpty(stack)) {  return stack->array[stack->top--];  }  return -1; // Stack is empty  }  // Function to perform Depth First Search (DFS) traversal  void DFS(struct Graph\* graph, int start) {  // Create a stack for DFS  struct Stack\* stack = createStack(graph->V);  // Array to keep track of visited vertices  int\* visited = (int\*)malloc(graph->V \* sizeof(int));  if (visited == NULL) {  fprintf(stderr, "Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  for (int i = 0; i < graph->V; ++i) {  visited[i] = 0;  }  // Push the start vertex onto the stack and mark it as visited  push(stack, start);  visited[start] = 1;  // Traverse while stack is not empty  while (!isStackEmpty(stack)) {  // Pop a vertex from stack and print it  int current = pop(stack);  printf("%d ", current);  // Get all adjacent vertices of the popped vertex current  // If an adjacent has not been visited, then mark it visited  // and push it to the stack  struct AdjListNode\* temp = graph->array[current].head;  while (temp) {  if (!visited[temp->dest]) {  push(stack, temp->dest);  visited[temp->dest] = 1;  }  temp = temp->next;  }  }  // Free allocated memory  free(stack->array);  free(stack);  free(visited);  }  // Structure for queue  struct Queue {  int front, rear, size;  unsigned capacity;  int\* array;  };  // Function to create a queue of given capacity  struct Queue\* createQueue(unsigned capacity) {  struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));  if (queue == NULL) {  fprintf(stderr, "Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  queue->capacity = capacity;  queue->front = queue->size = 0;  queue->rear = capacity - 1; // This is important, see the enqueue  queue->array = (int\*)malloc(queue->capacity \* sizeof(int));  if (queue->array == NULL) {  fprintf(stderr, "Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  return queue;  }  // Queue functions  int isQueueEmpty(struct Queue\* queue) {  return (queue->size == 0);  }  int isQueueFull(struct Queue\* queue) {  return (queue->size == queue->capacity);  }  void enqueue(struct Queue\* queue, int item) {  if (isQueueFull(queue))  return;  queue->rear = (queue->rear + 1) % queue->capacity;  queue->array[queue->rear] = item;  queue->size = queue->size + 1;  }  int dequeue(struct Queue\* queue) {  if (isQueueEmpty(queue))  return -1;  int item = queue->array[queue->front];  queue->front = (queue->front + 1) % queue->capacity;  queue->size = queue->size - 1;  return item;  }  // Function to perform Breadth First Search (BFS) traversal  void BFS(struct Graph\* graph, int start) {  // Create a queue for BFS  struct Queue\* queue = createQueue(graph->V);  // Array to keep track of visited vertices  int\* visited = (int\*)malloc(graph->V \* sizeof(int));  if (visited == NULL) {  fprintf(stderr, "Memory allocation failed\n");  exit(EXIT\_FAILURE);  }  for (int i = 0; i < graph->V; ++i) {  visited[i] = 0;  }  // Enqueue the start vertex and mark it as visited  enqueue(queue, start);  visited[start] = 1;  // Traverse while queue is not empty  while (!isQueueEmpty(queue)) {  // Dequeue a vertex from queue and print it  int current = dequeue(queue);  printf("%d ", current);  // Get all adjacent vertices of the dequeued vertex current  // If an adjacent has not been visited, then mark it visited  // and enqueue it  struct AdjListNode\* temp = graph->array[current].head;  while (temp) {  if (!visited[temp->dest]) {  enqueue(queue, temp->dest);  visited[temp->dest] = 1;  }  temp = temp->next;  }  }  // Free allocated memory  free(queue->array);  free(queue);  free(visited);  }  // Driver program to test above functions  int main() {  // Create a graph with 5 vertices  int V = 5;  struct Graph\* graph = createGraph(V);  // Add edges  addEdge(graph, 0, 1);  addEdge(graph, 0, 4);  addEdge(graph, 1, 2);  addEdge(graph, 1, 3);  addEdge(graph, 1, 4);  addEdge(graph, 2, 3);  addEdge(graph, 3, 4);  // Print the adjacency list representation of the graph  printf("Graph represented using adjacency list:\n");  printGraph(graph);  printf("\n");  // Perform BFS traversal starting from vertex 0  printf("BFS traversal starting from vertex 0:\n");  BFS(graph, 0);  printf("\n");  } |
| --- |
| **OUTPUT :** |