**1.** **Grafo**

| ADT Grafo  {Nodes = <nodes>, Edges = <edges>}  { inv: Graph.Nodes ≠ null, Graph.Edges ≠ null } |
| --- |
| Primitive Operations: |
| ▪ CreateGraph: → Graph  ▪ AddNode: Graph x Node → Graph  ▪ AddEdge: Graph x Node x Node → Graph  ▪ RemoveNode: Graph x Node → Graph  ▪ RemoveEdge: Graph x Node x Node → Graph  ▪ IsEmpty: Graph → Boolean  Dijkstra  DFS |

| Create Graph -> Graph  “Creates a new graph with no vertices and no edges” |
| --- |
| {pre : TRUE} |
| {post: Graph = {nill, nill} } |

| AddNode(Graph, Node) -> Graph  “Adds a new node to the graph ” |
| --- |
| {pre : The node to be added must not exist in the graph (Adjacency list and adjacency matrix} |
| {post: The node is added to the graph} |

| AddEdge(Graph, Node, Node) -> Graph  “Adds a new Edge” |
| --- |
| {pre: The nodes of the edge must exist in the graph} |
| {post: The edge is added to the graph with the specified weight (if weighted)} |

| RemoveNode(Graph, Node) -> Graph  “Deletes a Vertex” |
| --- |
| {pre: The node to be deleted must exist in the graph} |
| {post: The node and all associated edges are removed from the graph} |

| RemoveEdge(Graph, Node, Node) -> Graph  “Deletes a Vertex” |
| --- |
| {pre: The edge to be deleted must exist in the graph} |
| {post: The edge are removed from the graph} |

| isEmpty(Graph) "Checks if the Graph is empty" |
| --- |
| { pre:TRUE } |
| { post: Returns true if the graph is empty(No edges,No nodes); otherwise, returns false {if graph.getNode() == 0 : true, else : false}; } |

**1.** **Priority Queue**

| ADT Priority Queue  {Elements = <elements>, Size = <size>, PriorityFunction = <priority\_function>}  { inv: PriorityQueue.Size >= 0, PriorityQueue.Elements ≠ null } |
| --- |
| Primitive Operations: |
| ▪ CreatePriorityQueue: → PriorityQueue  ▪ InsertIntoQueue: PriorityQueue x Element → PriorityQueue  ▪ GetNext: PriorityQueue → Element  ▪ RemoveNext: PriorityQueue → PriorityQueue  ▪ IsEmpty: PriorityQueue → Boolean |

| CreatePriorityQueue() "Creates a new empty priority queue" |
| --- |
| { pre: NIL } |
| { post: PriorityQueue.Size = 0, PriorityQueue.Elements is empty } |

| InsertIntoQueue(queue, element) "Inserts an element into the priority queue" |
| --- |
| { pre: {PriorityQueue = queue, element ≠ null} } |
| { post: PriorityQueue.Size = queue.Size + 1, element is in PriorityQueue.Elements, following the priority criteria } |

| GetMaxElement(queue) "Obtains the element of highest priority in the queue" |
| --- |
| { pre: {PriorityQueue = queue} } |
| { post: element = queue.maxMaxElement. The element of highest priority in the queue is returned without removing it } |

| DequeueMaxElement(queue) "Removes/dequeues the element of highest priority from the queue and returns it" |
| --- |
| { pre: {PriorityQueue = queue} } |
| { post: PriorityQueue.Size = queue.Size - 1, the element of highest priority is removed and returned from PriorityQueue.Elements } |

| IsEmpty(queue) "Checks if the priority queue is empty" |
| --- |
| { pre: {PriorityQueue = queue} } |
| { post: Returns true if the priority queue is empty; otherwise, returns false {if queue.getSize() == 0 : true, else : false}} |

**Dijkstra**

| Dijkstra(Graph, start, priorityQueue) "Finds the shortest path from a starting node to all other nodes in a weighted directed graph" |
| --- |
| { pre: {Graph = graph, start is a valid node in the graph, priorityQueue is a valid priority queue} } |
| { post: Returns a mapping of each node to the minimum distance from the start node } |

| CreateDijkstraDataStructure(Graph, start) "Creates the data structures needed for Dijkstra's Algorithm" |
| --- |
| { pre: {Graph = graph, start is a valid node in the graph} } |
| { post: Returns a tuple (distances, previous) where distances is a map initialized with infinity for all nodes except start (0), and previous is a map with null for all nodes } |

| CreateDijkstraDataStructure(Graph, start) "Creates the data structures needed for Dijkstra's Algorithm" |
| --- |
| { pre: {Graph = graph, start is a valid node in the graph} } |
| { post: Returns a tuple (distances, previous) where distances is a map initialized with infinity for all nodes except start (0), and previous is a map with null for all nodes } |

**DFS**

| DFS(vert) "Performs Depth-First Search starting from a given vertex" |
| --- |
| { pre: {vert is a valid vertex in the graph} } |
| { post: The DFS traversal starting from the given vertex is performed } |

| DFS(graph, vert, visited) "Recursive Depth-First Search" |
| --- |
| { pre: {graph is a valid graph, vert is a valid vertex, visited is a map initialized with all vertices marked as not visited} } |
| { post: The DFS traversal starting from the given vertex is performed } |