| TAD Grafo  {Nodes = <nodes>, Edges = <edges>}  { inv: Graph.Nodes ≠ null, Graph.Edges ≠ null,vertex connected by edge } |
| --- |
| Primitive Operations: |
| ▪ CreateGraph: → Graph  ▪ AddNode: Graph x Node → Graph  ▪ AddEdge: Graph x Node x Node → Graph  Delete Node: node →boolean  getNode: Graph x Node → Node  DFS(start)  BFS(start)  Dijkstra(Vertice, toVertice, weight) -> Arraylist  FloydWarshall() ->double[][]  PrimL() ->GraphAdjacencyList []  PrimM() ->GraphAdjacencyMatrix []  KruskalL() -> GraphAdjacencyList []  KruskalM() -> GraphAdjacencyMatrix [] |

| CreateGraph: → Graph |
| --- |
| “Creates an empty graph” |
| { post:{graph} } |

| addNode(graph x vertex): |
| --- |
| “Adds a new vertex, with the given id, to the graph” |
| pre{vertex ∉ V} |
| { post{V={v\_0,v\_1,v\_2,,.... vert}} } |

| addEdge(graph x intialVertex x finalVertex x weight): |
| --- |
| “Adds a connection,with the given weight between the initial vertex and the final vertex” |
| pre{intialVertex ∈V ∧ finalVertex ∈ V ∧ weight>=0} |
| { post{E={(v\_0,v\_1,w ),(v\_1,v\_2, w\_2)....(initialVertex,finalVertex,weight)}} } |

| DeleteNode(vert) -> boolean |
| --- |
| “Deletes a node” |
| {pre: The node to be deleted must exist in the graph} |
| {post: The node and all associated edges are removed from the graph} |

| getNode(vert) -> Node |
| --- |
| “Finds the node in the Graph” |
| {pre: The node must exists} |
| {post: If the vertex node in the graph, the node is returned; otherwise, a null value is returned} |

| BFS(start) |
| --- |
| “Breadth-First Search” |
| {pre: The start node must exist in the graph} |
| {post: A list of nodes in BFS order is returned} |

| DFS(start) |
| --- |
| “Depth-First Search” |
| {pre: The start node must exist in the graph} |
| {post: A list of nodes in DFS order is returned |

| Dijkstra(Vertice, toVertice, weight) -> ArrayList |
| --- |
| “Dijkstra's Shortest Path Algorithm” |
| {pre: The graph must be weighted and connected} |
| {post: The shortest distances from the start node to all other nodes are returned} |

| FloydWarshall() -> double[][] |
| --- |
| “Floyd's All Pairs Shortest Path Algorithm” |
| {pre: The graph must be weighted} |
| {post: The matrix of minimum distances between all pairs of nodes is returned} |

| PrimL() -> GraphAdjacencyList  “Prim's Minimum Spanning Tree Algorithm [Modifier]”  {pre: The graph must be weighted and connected, represented as an adjacency list}  {post: The minimum spanning tree of the graph is returned as an adjacency list} |
| --- |

| PrimM() -> GraphAdjacencyMatrix  “Prim's Minimum Spanning Tree Algorithm []”  {pre: The graph must be weighted and connected, represented as an adjacency matrix}  {post: The minimum spanning tree of the graph is returned as an adjacency matrix} |
| --- |

| KruskalL() -> GraphAdjacencyList  “Kruskal's Minimum Spanning Tree Algorithm []”  {pre: The graph must be weighted, represented as an adjacency list}  {post: The minimum spanning tree of the graph is returned as an adjacency list} |
| --- |

| KruskalM() -> GraphAdjacencyMatrix  “Kruskal's Minimum Spanning Tree Algorithm []”  {pre: The graph must be weighted, represented as an adjacency matrix}  {post: The minimum spanning tree of the graph is returned as an adjacency matrix} |
| --- |