|  |  |  |
| --- | --- | --- |
| TAD Graph | | |
| Graph = {nodes: < Node₁, Node₂, Node₃, …, Nodeₙ>}   * Where:   Node = {key: <key>, value: <value>, edges: <Edge₁, Edge₂, Edge₃, …, Edgeₙ>}   * Where:   Edgeₘ = {weight: <weightₘ>, to: <keyₖ>} | | |
| { invariant: Nodeₐ.key ≠ Nodeₕ.key ∧ Edge.to ≠ NULL ∧ Edge.weight ≥ 0 } | | |
| Primitive functions: | | |
| * CreateGraph: |  | * Graph |
| * addNode: | Graph X Value X Key | * Graph |
| * addEdge: | Graph X Key₁ X Key₂ X Integer | * Graph |
| * removeNode: | Graph X Key | * Graph |
| * removeEdge: | Graph X Key₁ X Key₂ | * Graph |
| * getNeighbors: | Graph X Key | * List<Edge> |
| * getEdge: | Graph X Key₁ X Key₂ | * Edge |
| * DFS | Graph X Key | * List<Key> |
| * BFS | Graph X Key | * List<Key> |
| * Dijkstra | Graph X Key | * List<List<Key>> |

addNode(graph, value, key)

“Adds a new node with the given key and value to the graph”

{ pre: key ∉ {n.key | n ∈ graph.nodes} }

{ post: graph = {nodes: graph.nodes ∪ {Node(key, value, [ ])}} }

CreateGraph()

“Creates a new Graph object with 0 elements”

{ pre: TRUE}

{ post: graph = {nodes: [ ] } }

addEdge(graph, key₁, key₂, weight)

“Adds a new edge between the nodes with keys key₁ and key₂ in the graph with the given weight”

{ pre: key₁ ∈ {n.key | n ∈ graph.nodes} ∧ key₂ ∈ {n.key | n ∈ graph.nodes} ∧ weight ≥ 0 ∧ key₁ ≠ key₂ }

{ post: {n₁, n₂, e| n₁.key = key₁ ∧ n₂.key = key₂ ∧ e.weight = weight ∧ e.to = n₂.key ∧ e ∈ n₁.edges } }

removeNode(graph, key)

“Removes the node with the given key from the graph”

{ pre: key ∈ {n.key | n ∈ graph.nodes} }

{ post: ∃n {n ∈ graph.nodes ∧ n.key ≠ key ∃e { e ∈ n.edges ∧ e.to ≠ key } } }

removeEdge(graph, key₁, key₂)

“Removes the edge between the nodes with keys key₁ and key₂ from the graph”

{ pre: key₁ ∈ {n.key | n ∈ graph.nodes} ∧ key₂ ∈ {n.key | n ∈ graph.nodes} }

{ post: ∃ n, e { n ∈ graph.nodes ∧ n.key = key₁ ∧ e ∈ n.edges ∧ e.to ≠ key₂} }

getNeighbors(graph, key)

“Returns a list of edges representing the neighbors of the node with the given key in the graph”

{ pre: key ∈ {n.key | n ∈ graph.nodes} }

{ post: neighbors = {e | (n₁, e, n₂) ∈ graph ∧ n₁.key = key} }

Dijkstra(graph, key)

"Calculates the shortest paths from the node with the given key to all other nodes in the graph using Dijkstra's algorithm"

{ pre: key ∈ {node.key | node ∈ graph.nodes} }

{ post: shortest\_paths = {n: path | n ∈ graph.nodes ∧ path is the shortest path from the start node to n} }

BFS(graph, key)

“Returns a list of nodes representing the result of a breadth-first search starting from the node with the given key in the graph”

{ pre: key ∈ {n.key | n ∈ graph.nodes} }

{ post: result = [n | n ∈ graph.nodes ∧ there exists a path from the node with key to n in the graph] }

getEdge(graph, key₁, key₂)

“Returns the edge between the nodes with keys key₁ and key₂ in the graph”

{ pre: key₁ ∈ {n.key | n ∈ graph.nodes} ∧ key₂ ∈ {n.key | n ∈ graph.nodes} }

{ post: e = {e | (n₁, e, n₂) ∈ graph ∧ n₁.key = key₁ ∧ n₂.key = key₂} }

DFS(graph, key)

“Returns a list of nodes representing the result of a depth-first search starting from the node with the given key in the graph”

{ pre: key ∈ {n.key | n ∈ graph.nodes} }

{ post: result = [n | n ∈ graph.nodes ∧ there exists a path from the node with key to n in the graph] }