TAD Graph

Graph = {nodes: < $Node_1$, $Node_2$, $Node_3$, …, $Node_n$>}

- Where:

Node = {key: <key>, value: <value>, edges: <$Edge_1$, $Edge_2$, $Edge_3$, …, $Edge_n$>}

- Where:

$Edge_m$ = {weight: <$weight_m$>, to: <key_k>}

{ invariant: $Node_a$.key ≠ $Node_h$.key ∧ Edge.to ≠ NULL ∧ Edge.weight ≥ 0 }

Primitive functions:

CreateGraph:		→	Graph
addNode:	Graph X Value X Key	→	Graph
addEdge:	Graph X Key_1 X Key_2 X Integer	→	Graph
removeNode:	Graph X Key	→	Graph
removeEdge:	Graph X Key_1 X Key_2	→	Graph
getNeighbors:	Graph X Key	→	List<Edge>
getEdge:	Graph X Key_1 X Key_2	→	Edge
DFS	Graph X Key	→	List<Key>
BFS	Graph X Key	→	List<Key>
Dijkstra	Graph X Key	→	List<List<Key>>

CreateGraph()

"Creates a new Graph object with 0 elements"

{ pre: TRUE}

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

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, [ ])}} }

addEdge(graph, key_1, key_2, weight)

"Adds a new edge between the nodes with keys key_1 and key_2 in the graph with the given weight"

{ pre: key_1 ∈ {n.key | n ∈ graph.nodes} ∧ key_2 ∈ {n.key | n ∈ graph.nodes} ∧ weight ≥ 0 ∧ key_1 ≠ key_2 }

{ post: {n_1, n_2, e| n_1.key = key_1 ∧ n_2.key = key_2 ∧ e.weight = weight ∧ e.to = n_2.key ∧ e ∈ n_1.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, key_1, key_2)

“Removes the edge between the nodes with keys key_1 and key_2 from the graph”

{ pre: $key_1 \in \{n.key \mid n \in graph.nodes\} \land key_2 \in \{n.key \mid n \in graph.nodes\}$ }

{ post: $\exists\, n, e \,\{ n \in graph.nodes \land n.key = key_1 \land e \in n.edges \land e.to \neq key_2\}$ }

getNeighbors(graph, key)

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

{ pre: $key \in \{n.key \mid n \in graph.nodes\}$ }

{ post: $neighbors = \{e \mid (n_1, e, n_2) \in graph \land n_1.key = key\}$ }

getEdge(graph, key_1, key_2)

“Returns the edge between the nodes with keys key_1 and key_2 in the graph”

{ pre: $key_1 \in \{n.key \mid n \in graph.nodes\} \land key_2 \in \{n.key \mid n \in graph.nodes\}$ }

{ post: $e = \{e \mid (n_1, e, n_2) \in graph \land n_1.key = key_1 \land n_2.key = key_2\}$ }

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: $key \in \{n.key \mid n \in graph.nodes\}$ }

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

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: $key \in \{n.key \mid n \in graph.nodes\}$ }

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

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: $key \in \{node.key \mid node \in graph.nodes\}$ }

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