TAD Graph<K,V>

Graph<K,V>: HashTable<K,V>, K = vertexKey, V = Vertex<T> ∧ Vertex<T> = {Edges = {E_1 , $E_{2,}$ …. E_n} } ∧ adjacencyMatrix[ ][ ] = ΣE for Vertex<T>$_I$ ∧ Edges.weight = F(Vertex<T>$_I$, Vertex<T>$_j$) → R ∧ weightMatrix[ ][ ]$_{ij}$ = E.weight

{Inv.:

- **weightMatrix$_{if}$ = Edge.weight$_{ij}$, if {$V_{i,}$ V_J} ∈ Edges, Else, ∞.**
- **adjacencyMatrix$_{ij}$ = 0 if Vertex<T> if Edges = { ∅ }**
- **HashTable<K,V>.size = Constant.**
- **if Graph<T>.Edge Origin and destination is irrelevant, then graphGrade = 2 x Edges = Σ Vertex<T>.grade. Vertex<T>.grade = n**
  }

Primitive Operations:

- **addVertex: HashTable<K,V> x Vertex<T> → HashTable <K,V>. (Modifier)**
- **addEdge: Vertex<T> x Vertex<T> x Weight → Edge. (Modifier)**
- **deleteEdge: Vertex<T> x Vertex<T> → Vertex<T>. (Modifier)**
- **bfs: Vertex<T> → String (Analyzer)**
- **dfs: HashTable<K,V> x Vertex → String (Analyzer)**
- **dijkstraShortestPath : HashTable<K,V> -> Vertex<T> → String (Analyzer)**
- **resetVisitedStatus: HashTable<K,V> → HashTable<K,V>(Modifier)**
- **getAdjacencyList: Vertex<T> → List<Vertex<T>>(Analyzer)**
- **getAdjacencyMatrix: HashTable<K,V> → int[][]adjacency Matrix**
- **getIndexForVertex: Vertex<T> → int**

addVertex(K key, T data)

“Add a new vertex to the hash table.”

{pre: Vertex<T>.data $\neq$ null }

{post: HashTable<K, V> $\rightarrow$ newHashTable<K, V>}

addEdge(T originKey, T destinyKey, int weight)

“Creates a new edge that connects two vertexes in the hash table.”

{pre: Vertex<T>$_i$ $\wedge$ Vertex<T>j $\in$ HashTable<K,V>
Edge.weight > = 0
}

{post: edge = new Edge(V_i,V_J) }

deleteEdge(T originKey, T destinyKey)

“This method receives the keys associated to the vertexes that may have an edge between them, and if they do, it is deleted from the list of edges of the origin vertex, and if it is undirected, then removes the edge in both vertexes.”

{pre: Vertex<T>$_i$ $\wedge$ Vertex<T>j $\in$ HashTable<K,V>
Edge.weight > = 0
}

{post: removedEdge $\notin$ Vertex<T>.EdgesList }

bfs(T sourceKey)

“This method explores the graph in a breadth-first manner by receiving the key of the vertex, ensuring that vertices closer to the starting vertex are visited first, followed by vertices at increasing distances. ”

{pre: Vertex<T> ∈ HashTable<K,V>

    If Vertex<T>$_{i+1}$ is adjacent to Vertex<T>$_{I}$, then,
    List.add(Vertex<T>),
    }

{post: returns String containing a list with possible visited in List<Vertex<T>>, }

dfs(T sourceKey)

"Performs a Depth-First Search (DFS) traversal on the graph represented by the hash table, starting from a given vertex, and return a list of vertices in the order they are visited during the traversal."

{pre: Vertex<T> ∈ HashTable<K,V>,

    If Vertex<T>$_{i+1}$ is adjacent to Vertex<T>$_{I}$, then.
    List.add(Vertex<T>)
    }

{post: String containing a list with possible visited in returns List<Vertex<T>>,
    }

dijkstraShortestPath(T sourceKey)

" This algorithm is executed on the graph starting from the given vertex. The algorithm maintains a priority queue of vertices to explore, initializing the distance from the starting vertex to itself as 0 and the distances to all other vertices as infinity."

{pre: Vertex<T> ∈ HashTable<K,V>,

    Vertex<T>.Edges.weight ≠ null
    }

{post: String containing a list with possible visited in List<Vertex<T>> (for Vertex<T> in List, Weight is minimum
    }

getAdjacencyList()

“ This method returns a list that contains the vertexes that an origin vertex is directed. This method is only implemented in the adjacency list graph. ”

```
{pre: Vertex<T> ∈ HashTable<K,V>
        If Vertex<T>.destination ∈ Vertex<T>.origin.Edge
        }
```

{post: Returns List<Vertex<T>>}

resetVisitedStatus()

“ This method resets the visited status for every of the vertexes, to help the routes to do their task.”

{pre: HashTable<K,V>.size() ≠ 0}

```
{post: Vertex<T> ∈ HashTable<K,V>
        Vertex<T>.isVisited == False;
        }
```

geeAdjacencyMatrix()

“This method provides a compact and structured way to represent the relationships and edge weights in a graph using a two-dimensional array, making it useful for various graph-related algorithms and analyses. This method is only implemented in the adjacency matrix graph ”

{pre: HashTable<K,V>.size() ≠ 0}

{post: Returns int[ ][ ] representing the edges between the Vertexes} getIndexForVertex()

“ Finds the index of a given Vertex<T> within a list of vertices stored in the HashTable<K,V>. It is primarily used when generating an adjacency matrix for the graph.”

{pre: Vertex<T> ∈ HashTable<K,V>
}

{post: returns integer value Referring to the Vertex<T> }