**Introduction**

This document outlines the implementation of a Graph Abstract Data Type (ADT) through the **ArrayGraph** class in Java, detailing the underlying structure, methodology, and complexity analysis of its operations. This implementation adheres to the constraints and specifications provided, including the use of arrays to store vertices and edges in sorted ascending order and enforcing a maximum capacity of 20 vertices and 50 edges.

**Class Structure**

Vertex and Edge Classes

**Vertex<F extends Comparable<F>> implements Comparable<Vertex<F>>**

* **Instance Variable**: **private F value;** - stores the value of the vertex.
* **Methods**:
  + **Vertex(F value)**: Constructs a vertex with the specified value.
  + **F getValue()**: Returns the value of the vertex.
  + **void setValue(F value)**: Sets the vertex's value.
  + **boolean equals(Object other)**: Checks equality based on vertex value.
  + **int hashCode()**: Generates hash code based on value.
  + **int compareTo(Vertex<F> other)**: Compares vertices based on their value.
  + **String toString()**: Returns string representation of the vertex's value.

**Edge<F extends Comparable<F>> implements Comparable<Edge<F>>**

* **Instance Variables**:
  + **private final Vertex<F> v1;** - Start vertex (label is smaller).
  + **private final Vertex<F> v2;** - End vertex (label is larger).
* **Methods**:
  + **Edge(Vertex<F> v1, Vertex<F> v2)**: Constructs an edge, ensuring vertex order.
  + **Vertex<F> getV1()** and **Vertex<F> getV2()**: Return start and end vertices.
  + **boolean equals(Object other)**: Checks equality based on start and end vertices.
  + **int hashCode()**: Generates hash code considering both vertices.
  + **int compareTo(Edge<F> other)**: Compares edges based on start vertex.
  + **String toString()**: Returns a formatted string representing the edge.

ArrayGraph Class

Implements the **Graph** interface with generic types, using arrays for storing vertices (**Vertex<F>[] vertices**) and edges (**Edge<F>[] edges**), and integer counters (**numVertices**, **numEdges**) for tracking their current counts.

**Core Methods and Complexity Analysis**

addVertex(Vertex<F> v)

* **Purpose**: Adds a vertex to the graph if it does not already exist.
* **Underlying Algorithm**:
  + **Existence and Index Check**: Binary search (**findVertexIndex**), to efficiently determine if the vertex already exists within the sorted array of vertices, and then use the index returned by that method for insertion.
  + **Insertion**: If the vertex does not exist, the return value is inverted, giving the insertion point which is then passed on to the **insertElementInSortedOrder** method. This method then inserts the vertex and shifts elements to maintain the order.
* **Complexity**:
  + **Existence and Index Check**: **O(log n)** where **n** is the number of vertices, due to binary search.
  + **Insertion**: The shifting of elements for insertion can take up to **O(n)** in the worst case. Therefore, the overall complexity is dominated by the shifting operation, making it **O(n)**.

addEdge(Edge<F> e)

* **Purpose**: Adds an edge to the graph if both vertices exist and the edge does not already exist.
* **Underlying Algorithm**:
  + **Vertex Existence Checks**: Two binary searches to check if each vertex of the edge exists in the graph.
  + **Edge Existence and Index Check**: Binary search (**findEdgeIndex**) to efficiently determine if the edge already exists within the sorted array of edge, and then use the index returned by that method for insertion.
  + **Insertion**: If the edge does not exist, the return value is inverted, giving the insertion point which is then passed on to the **insertElementInSortedOrder** method. This method then inserts the edge and shifts elements to maintain the order.
* **Complexity**:
  + **Vertex Checks**: **2 \* O(log n)** simplifies to **O(log n)**.
  + **Edge Existence and Index Check**: **O(log m)** where **m** is the number of edges.
  + **Insertion**: Shifting elements can be **O(m)** in the worst case. Thus, the total complexity is **O(m)** due to the shifting operation for insertion.

deleteVertex(Vertex<F> v)

* **Purpose**: Deletes a vertex and any edges connected to it.
* **Underlying Algorithm**: Binary search for vertex, array manipulation for deletion, and linear search for edge deletion.
* **Complexity**: **O(n + m)** due to shifting vertices and removing edges, where **n** is the number of vertices and **m** is the number of edges.

deleteEdge(Edge<F> e)

* **Purpose**: Deletes an edge if it exists.
* **Underlying Algorithm**: Binary search for edge and array manipulation for deletion.
* **Complexity**: **O(m)** for edge search and deletion, where **m** is the number of edges.

vertexSet() and edgeSet()

* **Purpose**: Returns a set of all vertices or edges.
* **Underlying Algorithm**: Conversion from array to List to Set.
* **Complexity**: **O(n)** for vertices and **O(m)** for edges, where **n** is the number of vertices and **m** is the number of edges.

**Helper Methods**

* **containsVertex(Vertex<F> v)**: Utilizes binary search to check for a vertex's existence. Complexity: **O(log n)**.
* **containsEdge(Edge<F> e)**: Utilizes binary search to check for an edge's existence. Complexity: **O(log m)**.
* **insertSorted(T[] array, T element, int count)**: Finds insertion point using binary search and inserts the element, shifting subsequent elements. Complexity: **O(n)** for shifting.

**Conclusion**

The **ArrayGraph** class offers a compact and efficient representation of a graph with restricted capacities, employing sorted arrays to facilitate rapid searches and orderly storage. This document has presented the design and complexity analysis of its key operations, underscoring the effectiveness and limitations of using arrays as the underlying data structure for graph representation. Given the scope and requirements, this implementation serves as a robust foundation for representing and manipulating graphs with defined maximum capacities.