**Introduction**

This document outlines the implementation of a Graph Abstract Data Type (ADT) through the **ArrayGraph** class, 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**

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

* **Instance Variable**:
  + **private final 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.
  + **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.

**public class 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.

**public class ArrayGraph<F extends Comparable<F>> implements Graph<F>**

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 unique vertex to the graph, ensuring no duplicates.
* **Underlying Algorithm:**
  + **Existence and Index Check:** Utilizes binary search via **findVertexIndex** to efficiently determine a vertex's presence. If absent, the method calculates the precise insertion index.
  + **Insertion:** Upon confirmation of the vertex's absence, its intended position in the array is pinpointed, ensuring vertices remain sorted. The **insertElementInSortedOrder** method efficiently places the new vertex into this spot, adjusting subsequent elements to preserve order.
* **Complexity Analysis:**
  + **Existence and Index Check:** Executes in O(log n), with 'n' representing the total vertex count, attributable to binary search's logarithmic performance.
  + **Insertion Process:** Element shifting, necessary for maintaining array order during insertion, peaks at O(n). Consequently, insertion's linear time complexity, O(n), prevails as the operation's dominant factor.

**addEdge(Edge<F> e)**

* **Purpose:** Incorporates a new edge between two existing vertices, barring duplicates.
* **Underlying Algorithm:**
  + **Vertex Existence Checks:** A pair of binary searches validate the presence of both vertices within the graph.
  + **Edge Existence and Index Check:** Another binary search, **findEdgeIndex**, ascertains the edge's non-existence and identifies the appropriate insertion point.
  + **Insertion:** Absence confirmed, the edge is methodically inserted at the derived position, with subsequent elements shifted to maintain the array's sorted state.
* **Complexity Analysis:**
  + **Vertex Verification:** The dual binary searches, despite being two, collectively approximate to O(log n) complexity.
  + **Edge Check and Insertion:** With 'm' denoting the edge count, the search and subsequent insertion operations culminate in a complexity of O(m), heavily influenced by the necessity to shift elements for edge placement.

**deleteVertex(Vertex<F> v)**

* **Purpose:** Excises a specified vertex and its associated edges from the graph.
* **Underlying Algorithm:**
  + A binary search initially locates the vertex. Following its discovery, array manipulation techniques facilitate its removal.
  + Edges linked to this vertex undergo a streamlined deletion process, identified through a linear scan and excised in a single operation to enhance efficiency.
* **Complexity Analysis:**
  + The amalgamation of vertex removal and associated edge deletions executes in O(n + m), where 'n' is the vertex tally and 'm' encapsulates the edges, reflecting the composite steps of vertex shifting and edge pruning.

**deleteEdge(Edge<F> e)**

* **Purpose:** Removes an existing edge from the graph.
* **Underlying Algorithm:** A binary search pinpoints the target edge. Success leads to the edge's extraction from the array, invoking array manipulation techniques to ensure continuity and order.
* **Complexity Analysis:** Edge deletion is completed in O(m) time, predominantly due to the array manipulation required post-identification of the edge, with 'm' representing the total edge count.

**vertexSet()**

* **Purpose**: Retrieves a set containing all the vertices present in the graph. This method is instrumental for operations requiring iteration over all vertices, such as graph traversal or visualization.
* **Underlying Algorithm**: The method initiates by converting the portion of the array containing vertices up to **numVertices** into a list. Subsequently, this list is transformed into a HashSet, ensuring unique elements and providing efficient lookup times. This process eliminates any null values that could exist beyond the last vertex in the array due to the fixed size of the vertices array.
* **Complexity Analysis**: The overall time complexity is O(n), correlating with the number of vertices (n). This complexity arises from iterating over the array to create the list and then constructing a HashSet from this list. The direct array-to-list conversion is linear with respect to the number of vertices, and so is the list-to-set conversion, as it requires iterating over the list to add elements to the set.

**edgeSet()**

* **Purpose**: Returns a set comprising all the edges in the graph. This functionality is crucial for algorithms that need to access every edge, such as those calculating the minimum spanning tree or detecting cycles.
* **Underlying Algorithm**: Similar to **vertexSet()**, this method converts the segment of the edge array up to **numEdges** into a list, which is then converted into a HashSet. This ensures all edges in the returned set are unique and allows for efficient access. The conversion process effectively ignores any null entries beyond the last edge in the array, maintaining the integrity of the edge set.
* **Complexity Analysis**: The method exhibits a time complexity of O(m), where m is the number of edges. This is due to the linear nature of the array-to-list and then list-to-set conversions, each of which operates in a time proportional to the number of edges in the graph.

**Helper Methods**

* **private int findVertexIndex(Vertex<F> v):** Executes a binary search to identify the position of a specified vertex within the vertices array. If the vertex is not found, the method returns a negative value. This result can then be converted into an appropriate insertion point for the vertex by negating and decrementing the returned value. This ensures efficient insertion while keeping the array sorted.  
  **Complexity Analysis**: Maintains a time complexity of O(log n), where n denotes the total number of vertices. This efficiency is attributable to the binary search's methodical halving of the search interval, minimizing the required comparisons to either find a vertex or deduce where it should be inserted.
* **private int findEdgeIndex(Edge<F> e)**: Searches for the index of a specified edge within the array of edges using binary search. A potential negative result can be converted into the insertion point.  
  **Complexity Analysis**: The operation's time complexity is O(log m), with m representing the current count of edges. This mirrors the binary search's ability to efficiently narrow down the target space, facilitating either the discovery of an edge or the pinpointing of an insertion slot.
* **private <T> void insertElementInSortedOrder(T[] array, T element, int count, int insertionPoint)**: This method is tasked with inserting a new element into the sorted array at the designated insertion point. It manages this by shifting all elements positioned after the insertion point one space towards the end of the array to accommodate the new element. This ensures the array's order is maintained after the insertion, crucial for both vertex and edge additions. It was introduced as an application of the DRY principle of software development.  
  **Complexity Analysis**: The shifting operation dictates a time complexity of O(n) for vertices and O(m) for edges, dependent on the array's size before insertion. This worst-case scenario arises when inserting at the array's start, requiring a shift of all subsequent elements.

**Conclusion**

My **ArrayGraph** class attempts to not only encapsulate the required functionalities as described in the specification, but also to do in the most efficient way possible, under the specified constraints. It represents 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, which ultimately serve to emphasise both the effectiveness and the limitations of using arrays as the underlying data structure for graph representation. Given the scope and requirements, I believe this implementation serves as a robust foundation for representing and manipulating graphs with defined maximum capacities.