**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**:
  + **public Vertex(F value)**: Constructs a vertex with the specified value.
  + **public F getValue()**: Returns the value of the vertex.
  + **public boolean equals(Object other)**: Checks equality based on vertex value.
  + **public int hashCode()**: Generates hash code based on value.
  + **public int compareTo(Vertex<F> other)**: Compares vertices based on their value.
  + **public 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**:
  + **public Edge(Vertex<F> v1, Vertex<F> v2)**: Constructs an edge, ensuring vertex order.
  + **public Vertex<F> getV1()** and **Vertex<F> getV2()**: Return start and end vertices.
  + **public boolean equals(Object other)**: Checks equality based on start and end vertices.
  + **public int hashCode()**: Generates hash code considering both vertices.
  + **public int compareTo(Edge<F> other)**: Compares edges based on start vertex.
  + **public String toString()**: Returns a formatted string in the form of "(startVertex -> endVertex)" 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. Its methods are described and analysed below:

**public boolean 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 a helper method **(findVertexIndex)** to efficiently determine a vertex's presence. If absent, this method returns a negative value. The result can then be converted into an appropriate insertion point for the vertex by negating and decrementing the returned value**.**
  + **Insertion:** Upon confirmation of the vertex's absence, the insertion point is calculated as described above, and passed into a second helper method **(insertElementInSortedOrder)** which 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.

**public boolean 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 returns a value that can later be user to identify 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, again using the **insertElementInSortedOrder** helper method.
* **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), again dominated by the necessity to shift elements for edge placement.

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

* **Purpose**: This method is designed to remove a specific vertex from the graph, along with all edges that are connected to it, ensuring the integrity of the graph's structure.
* **Underlying Algorithm**:
  + The process begins with a binary search to swiftly locate the vertex within the graph. Once found, we employ array manipulation techniques to remove the vertex from its storage array, effectively excising it from the graph.
  + Following the vertex's removal, we conduct a linear scan through the edges array to identify and remove all edges that are linked to the vertex. This is achieved in a single pass to streamline the process and maintain the graph's accuracy without redundant edges.
* **Complexity Analysis**:
  + The combined operation of removing a vertex and its associated edges is performed with a complexity of O(n + m). This reflects the steps involved in shifting vertices (O(n)) and pruning edges (O(m)), where 'n' represents the total number of vertices and 'm' the total number of edges in the graph.

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

* **Purpose**: Aimed at eliminating a specific edge from the graph, this method ensures that the graph's connectivity is accurately maintained by removing only the connections that are no longer needed or valid.
* **Underlying Algorithm**:
  + Initially, a binary search is utilized to precisely locate the edge in question within the graph's edge array. Upon locating the edge, we apply array manipulation strategies to remove it, ensuring the remaining edges are correctly repositioned to keep the array's order intact. This meticulous approach guarantees that the graph's structure remains coherent and searchable.
* **Complexity Analysis**:
  + The process of deleting an edge operates within O(m) time complexity, where 'm' stands for the edge count. This primarily accounts for the search and the subsequent array adjustments required to remove the edge and reorganize the remaining elements, preserving the graph's orderly structure.

**public Set<Vertex<F>> 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.

**public Set<Edge<F>> 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):** This method is essentially just a direct call to **Arrays.binarySearch()**, specifying the search to only consider the portion of the array currently in use (**0** to **numVertices**). It simply serves to abstract away the binary search operation to improve readability. Its function is to execute 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)**: Like its vertex equivalent, this method is a direct call to **Arrays.binarySearch()**. It 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 as described above.  
  **Complexity Analysis**: The operation's time complexity is O(log m), with m representing the current count of edges. This reflects 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.