In my implementation of Dijkstra's Algorithm, I utilized several fundamental data structures to balance efficiency and simplicity. I chose an adjacency list representation for the graph, implemented as a Map<Integer, List<Edge>>. This approach is memory-efficient and provides quick access to the neighbors of each vertex, which is crucial for Dijkstra's algorithm. Since the graphs typically encountered in Dijkstra's problems are sparse, this representation is particularly suitable. Additionally, the adjacency list structure simplifies both the construction and traversal of the graph, making it a practical choice.

To efficiently retrieve the vertex with the smallest distance, I used a PriorityQueue<Vertex>. Being heap-based, the priority queue optimizes the removal of the smallest element, a key operation in Dijkstra’s algorithm. This choice minimizes the time required for these operations, while keeping the implementation simple. I also employed fixed-size arrays to store the shortest distances (distances[]) and predecessors (parents[]) for each vertex. Arrays provide O(1) access time and are straightforward to manage, making them an effective choice for handling vertex-related data.

The worst-case and average-case runtime complexity of my implementation is O((V + E) log V), where V is the number of vertices and E is the number of edges. Constructing the adjacency list from the input takes O(V + E) time. Priority queue operations, such as extracting the minimum element, take O(log V) time per operation. Edge relaxations also involve O(log V) time as they update the priority queue. Since these operations dominate the overall runtime, they determine the algorithm’s complexity. While this runtime is efficient for most practical cases, replacing the standard priority queue with a Fibonacci heap could theoretically improve the complexity to O(V log V + E). However, implementing a Fibonacci heap would significantly increase code complexity without offering substantial practical benefits, especially for small or sparse graphs.

The memory requirements for my implementation are straightforward. Representing the graph using an adjacency list requires O(V + E) space. Arrays for distances[] and parents[] each require O(V) space. The priority queue, which stores up to V elements, adds another O(V). Therefore, the overall memory complexity is O(V + E) in both the worst and average cases. If necessary, memory usage could be optimized by compressing the adjacency list into a Compressed Sparse Row format, which would save space for large graphs. However, this would complicate adjacency traversal and might not align well with Dijkstra's algorithm requirements.

Alternatively, using dynamic, variable-sized structures could reduce memory usage when not all vertices are used. However, this approach would likely increase runtime complexity. Overall, the data structures I selected strike a balance between simplicity and efficiency. They allowed for a manageable yet robust implementation suitable for the problem constraints. While improvements such as Fibonacci heaps or graph compression could theoretically enhance performance, they are unnecessary for this assignment, as the current design achieves a good balance between runtime and memory usage.