Data structure and algorithms.

Luong Hoang Hai

BH01797

1. A stack ADT, a concrete data structure for a First In First out (FIFO) queue.

- What is DSA?
- **Data Structures and Algorithms (DSA)**
- is a field of computer science that focuses on the organization and manipulation of data. Data structures are ways to store and organize data efficiently, while algorithms are procedures or formulas for solving problems. DSA is fundamental for writing efficient software and understanding how data is processed.

What is ADT?

- **Abstract Data Type (ADT)** is a mathematical model for data types where the data type is defined by its behavior from the point of view of a user. This includes the operations that can be performed on the data type and the properties of those operations. ADTs are implemented using concrete data structures.

How many ways are there to implement Stack and Queue?

- **Stack Implementations**

1. **Array-based Implementation**
    1. Uses a fixed-size array to store elements.
    2. Simple and efficient for a known maximum size.
    3. Requires resizing if the stack grows beyond the initial capacity.

1. **Linked List Implementation**
    1. Uses nodes where each node points to the next.
    2. Dynamic size, no pre-defined limit.
    3. Memory overhead due to node pointers.
2. **Dynamic Array Implementation**
    1. Similar to the array-based approach but automatically resizes (e.g., doubling the array size when full).
    2. Balances memory usage and access speed.

- **Queue Implementations**

1. **Array-based Implementation**
    1. Uses a circular array to efficiently utilize space.
    2. Front and rear indices wrap around to avoid shifting elements.
    3. May require resizing if the maximum capacity is reached.
2. **Linked List Implementation**
    1. Consists of nodes with pointers to the next node.
    2. Dynamic size and allows efficient enqueue and dequeue operations.

1. **Two Stacks Implementation**
    1. Uses two stacks to simulate queue operations.
    2. One stack for enqueue and another for dequeue, managing the order through stack operations.
2. **Dynamic Array Implementation**
    1. Similar to the array-based approach, with automatic resizing to manage capacity.

2. Two sorting algorithms.

- **Bubble Sort**
- **Description:** Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until no swaps are needed.
- **Time Complexity:**
  - **Best Case:** $O(n)$ (when the array is already sorted)
  - **Average Case:** $O(n^2)$
  - **Worst Case:** $O(n^2)$ (when the array is sorted in reverse order)
- **Space Complexity:** $O(1)$ (in-place sorting)

- **Heap Sort**
- **Description:** Heap Sort builds a binary heap from the input data and then repeatedly extracts the maximum (or minimum) element from the heap, reconstructing the heap until all elements are sorted.
- **Time Complexity:**
  - **Best Case:** O(n log n)
  - **Average Case:** O(n log n)
  - **Worst Case:** O(n log n)
- **Space Complexity:** O(1) (in-place sorting, though it may use additional space for the heap structure)

3. Two network shortest path algorithms.

- **1. Dijkstra's Algorithm**
- **Overview:** Dijkstra's Algorithm is used to find the shortest path from a single source node to all other nodes in a graph with non-negative edge weights.
- **How It Works:**
- **Initialization:** Set the distance to the source node to 0 and all other nodes to infinity. Use a priority queue to track nodes by their current shortest distance.
- **Processing Nodes:**
  - Extract the node with the smallest distance from the queue.
  - Update distances for its neighbors if a shorter path is found through the current node.

- **Repeat:** Continue until all nodes are processed or the queue is empty.
- **Time Complexity:**
- $O(V^2)$ with an array implementation.
- $O((V + E) \log V)$ with a binary heap or priority queue.
- **Limitations:**
- Only works with non-negative weights. It does not handle negative weight edges.

- **Bellman-Ford Algorithm**
- **Overview:** The Bellman-Ford Algorithm computes shortest paths from a single source node to all other nodes in a graph, accommodating graphs with negative edge weights.
- **How It Works:**
- **Initialization:** Set the distance to the source node to 0 and all other nodes to infinity.
- **Relaxation:** For each edge in the graph, update the distances of the target nodes:
  - Repeat this for V-1 iterations (where V is the number of vertices).

- **Check for Negative Cycles:** After V-1 iterations, check all edges again. If any distance can still be updated, a negative weight cycle exists.
- **Time Complexity:**
- O(VE), where V is the number of vertices and E is the number of edges.
- **Limitations:**
- Slower than Dijkstra's Algorithm for graphs without negative weights.