Data Abstraction and Object Orientation: Tutorial 17-24th Sept Plan

Name: Soham Dambalkar **ID: 2023A7PS0343G**

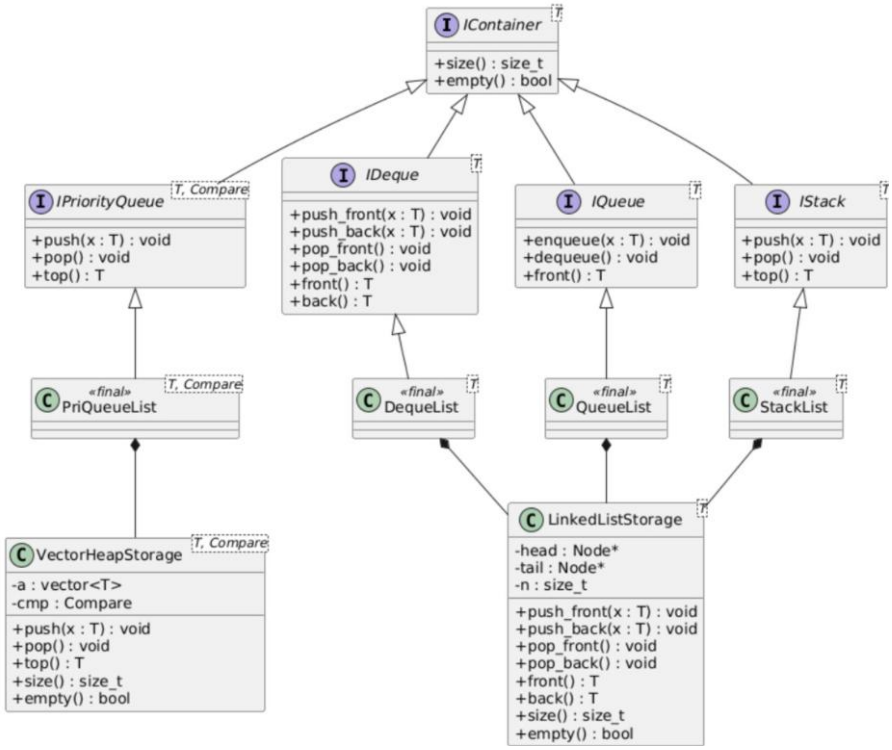
Introduction and Motivation

For this assignment, we're designing a set of list-like data structures—like stacks, queues, deques, and priority queues. The main idea is to make these different containers easy to use, while keeping their internal details hidden from users. We're following object-oriented principles, especially separating what a data structure does (its interface) from how it actually works (its implementation).

We want users to interact only with clear and safe interfaces. So, you should only be able to do stack things with a stack, queue things with a queue, and so on. This helps avoid mistakes, like accidentally trying to use one type as another. We're taking ideas from the Bjarne Stroustrup reading, focusing on abstract classes and keeping code flexible for future changes

Main Plan and Design

- We start with a very basic common interface called IContainer. This only lets you check the size of a container or if it's empty. Nothing else—so you can't mix up behaviors by mistake.
- Each container type gets its own interface:
    - Stacks (IStack) let you push, pop, and look at the top.
    - Queues (IQueue) let you add to the end, remove from the front, or see what's at the front.
    - Deques (IDeque) let you add/remove from both ends.
    - Priority queues (IPriorityQueue) let you add items by priority and always get the highest (or lowest) one out first.
- Behind each container, there's a "storage engine" that does the actual work (either as a linked list, or a vector/heap).
- The storage details ("how" it works) are private—you just use the public API ("what" it does).



User-Facing Interfaces and Classes (Detailed API “Promise”)

---

1. IContainer (Interface)

- **size() : size_t**
    - Returns the current number of items in the container.
- **empty() : bool**
    - Returns true if there are no items, false otherwise.

---

2. IStack<T> (Interface)

- **push(x : T) : void**
    - Puts the item x at the top of the stack.
- **pop() : void**
    - Removes the top item.
    - If the stack is empty, usually throws an exception or error.
- **top() : T**

- Returns the item at the top without removing it.
- If the stack is empty, throws an error.

---

3. IQueue<T> (Interface)

- **enqueue(x : T) : void**
  - Adds the item x to the rear (end) of the queue.
- **dequeue() : void**
  - Removes the front (oldest) item from the queue.
  - Throws error if queue is empty.
- **front() : T**
  - Returns the item at the front, without removing it.
  - Throws error if queue is empty.

---

4. IDeque<T> (Interface)

- **push_front(x : T) : void**
  - Adds an item at the front (head).
- **push_back(x : T) : void**
  - Adds an item at the back (tail).
- **pop_front() : void**
  - Removes item from the front.
  - Throws error if deque is empty.
- **pop_back() : void**
  - Removes item from the back.
  - Throws error if deque is empty.
- **front() : T**
  - Returns the front item, without removing.
  - Throws if deque is empty.
- **back() : T**
  - Returns the back item, without removing.
  - Throws if deque is empty.

---

5. IPriorityQueue<T, Compare> (Interface)

- **push(x : T) : void**
    - Adds item x into the priority queue, uses a given comparator.
- **pop() : void**
    - Removes the item with the highest (or lowest) priority.
    - Throws if empty.
- **top() : T**
    - Returns the item with the highest (or lowest) priority, without removing it.
    - Throws if empty.

Concrete Classes

Each of these implements only one behavior interface and uses a specific storage engine:

---

6. StackList<T> (Final Concrete Class)

- Implements **IStack<T>**.
- Internally uses a linked list (through composition with LinkedListStorage).
- All stack interface methods (push, pop, top, size, empty).

---

7. QueueList<T> (Final Concrete Class)

- Implements **IQueue<T>**.
- Also composed with a linked list.
- Supports enqueue, dequeue, front, size, empty.

---

8. DequeList<T> (Final Concrete Class)

- Implements **IDeque<T>**.
- Internally uses doubly-linked list for efficient add/remove at both ends.
- Supports all deque
  methods: push_front, push_back, pop_front, pop_back, front, back, size, empty.

---

9. PriQueueList<T, Compare> (Final Concrete Class)

- Implements **IPriorityQueue<T, Compare>**.

- Composes with a vector heap for efficient priority ordering.
- Supports push, pop, top, size, empty.

Internal Storage Engines (for completeness)

These classes are not used directly by users, but are critical for performance and design clarity:

---

10. LinkedListStorage<T>

- Internally manages nodes with head and tail pointers, keeps count with n.
- Provides fast operations for all front/back manipulations.
- Methods:
    - push_front(x), push_back(x)
    - pop_front(), pop_back()
    - front(), back()
    - size(), empty()

---

11. VectorHeapStorage<T, Compare>

- Maintains a vector (dynamic array) and a comparator.
- Used in the implementation of priority queues.
- Methods:
    - push(x), pop(), top()
    - size(), empty()

Note:

- All error conditions (like popping or accessing from empty structure) are to be handled with clear errors or exceptions.
- Users can only access the behaviors via interfaces—not by manipulating the storage engines or changing how the data is kept internally.

Key Promises of This API Design

- **Safety:**
  Each type of interface (stack, queue, etc.) only gives you the right set of operations, so you can never mix up stack and queue behaviors by accident.
- **Abstraction:**
  As a user, you just use the functions provided by the interface. You don't need to know (or worry) about how the data is actually stored inside.
- **Easy to Extend:**
  If we want to add new types of containers in the future (like a circular buffer or a more advanced priority queue), we can do this without changing or breaking the classes and interfaces we already have.
- **Works for Anything:**
  All the data structures are generic, so you can use them with any kind of element type, as long as you provide the needed comparison function if required (like in a priority queue).
- **Clear on Errors:**
  Every operation clearly says what happens if you call it when you shouldn't—for example, trying to pop from an empty queue gives you an error right away, so bugs are easier to spot.

Conclusion

- This design keeps each container (stack, queue, etc.) safe by only exposing the right set of functions, so you can't mix up behaviors.
- By hiding the storage details and using interfaces, we make it easy to update or change how items are stored without affecting user code.
- Using generics/templates lets our containers work for any data type and prepares us for future cases like mixed-type lists.
- If we want to add new data structures or improve performance later, it's simple to do without breaking anything that already works.
- The code can be implemented with abstract interfaces and separate classes for each behavior, using private storage like linked lists or heaps.