ListContainer — Implementation Plan

Goals

- Provide a single abstract list container (`ListContainer<T>`) that captures common storage and behaviour.
- Implement concrete list specializations that enforce different logical access policies: `Stack`, `Queue`, `Deque`, `PriorityQueue`, `GenericList`.
- Provide generic element handling so lists accept any `T` (with optional `Comparable<T>` or `Comparator<T>` for ordering).
- Maintain clear time/space complexity expectations and strict pre/postconditions for every API.

Core invariants (must always hold)

- `size` equals number of logical elements stored in `elements`.
- `elements` order reflects the logical order used by the concrete container (e.g., top of stack at end or front consistently).
- All public methods leave containers in a valid state or throw a documented exception; no partial corruption.
- Methods that return references to stored elements should not expose internal mutable structure in an unsafe way (documented for the implementation language).

2) Abstract base: `ListContainer<T>` (<>)

Fields

- `protected List<T> elements` — backing collection (default `ArrayList<T>` or `LinkedList<T>` depending on subclass performance needs).
- `protected int size` — number of stored elements (kept consistent with `elements.size()`).

Constructor

- `ListContainer()`
  Creates an object.

API (behavioural contract)

- `+isEmpty(): boolean`
  - Returns `true` iff `size == 0`.
  - Complexity: `O(1)`.

- `+size(): int`
  - Returns `size`. Complexity: `O(1)`.
- `+clear(): void`
  - Removes all elements, sets `size` to `0` and clears `elements`. Complexity: `O(n)` depending on backing structure.
- `+update(index: int, value: T): boolean`
  - Replaces element at logical `index` with `value`.
  - Index is 0-based. Precondition: `0 <= index < size`.
  - Returns `true` on success, `false` if index out of range. Postcondition: element at `index` equals `value`.
  - Complexity: `O(1)` for random-access backing lists (`ArrayList`), `O(n)` for linked structures.
- `+contains(value: T): boolean`
  - Returns `true` when an element equal to `value` exists (uses `equals` or comparator equality).
  - Complexity: `O(n)`.
- `+toString(): String`
  - Returns a readable representation of the container: shows class name, size, and logical order of elements.
  - Complexity: `O(n)`.
- `+add(element: T): <abstract return>`
  - Abstract method. Semantics: insert element according to concrete container policy (e.g., `Stack` pushes, `Queue` enqueues, `GenericList` appends).
  - The exact return type and additional behavior is defined by subclasses.
  - Complexity: documented in subclass.
- `+remove(): T`
  - Abstract method.
  - Semantics: remove and return an element according to concrete container policy. If the container is empty, either return `null` (or `Optional`) or throw `NoSuchElementException` — the subclass must document the chosen error strategy.
  - Complexity: documented in subclass.
- `+peek(): T`
  - Abstract method.
  - Semantics: return but do not remove the element that would be returned by `remove()`. If empty, returns `null` or throws; documented by subclass.
  - Complexity: documented in subclass.

Design notes

`ListContainer` supplies common helper utilities (`contains`, `update`, `clear`, `toString`). Concrete classes override `add`/`remove`/`peek`.

---

3) Concrete containers

`Stack<T>`

Concept: LIFO (Last In First Out).
Inheritance: `Stack` extends `ListContainer<T>`.

APIs

- `+push(element: T): void`
  - Pushes element onto the top.
  - Postcondition: `size` increments by `1`.
  - Complexity: `O(1)` amortized with array-backed implementation.
- `+pop(): T`
  - Removes and returns the top element.
  - Precondition: `size > 0`. Postcondition: `size` decremented.
  - Complexity: `O(1)` amortized.
- `+peek(): T`
  - Returns top element without removing.
  - Precondition: `size > 0`.
  - Complexity: `O(1)`.
- `+add(element: T): void`
  - Adds to the top (equivalent to `push`).
- `+remove(): T`
  - Removes top (equivalent to `pop`).

Error behaviour

Empty `pop()` or `peek()` throws `NoSuchElementException` or returns `null`

Complexity summary

- `push/add`: `O(1)` amortized.
- `pop/remove/peek`: `O(1)`.

`Queue<T>`

Concept: FIFO (First In First Out).

APIs

- `+enqueue(element: T): void`
    - Adds element to the tail.
    - Postcondition: element placed at logical end.
    - Complexity: `O(1)` amortized.
- `+dequeue(): T`
    - Removes and returns element from head.
    - Precondition: `size > 0`.
    - Complexity: `O(1)` for linked implementation, `O(n)` for naive array remove-from-front;
- `+peek(): T`
    - Returns head without removing.
    - Precondition: `size > 0`.
    - Complexity: `O(1)`.
- `+add(element: T): void`
    - Appends to tail.
- `+remove(): T`
    - Removes from head.

Complexity summary

- `enqueue/add`: `O(1)`.
- `dequeue/remove/peek`: `O(1)` when using `LinkedList` or circular buffer.

`Deque<T>`

Concept: Double-ended queue — insert/delete at both ends.

APIs

- `+addFirst(element: T): void`
    - insert at front.
    - Complexity: `O(1)`.
- `+addLast(element: T): void`
    - insert at back.
    - Complexity: `O(1)`.
- `+removeFirst(): T`
    - remove and return front element.
    - Precondition: `size > 0`.
    - Complexity: `O(1)`.
- `+removeLast(): T`
    - remove and return back element.
    - Precondition: `size > 0`.
    - Complexity: `O(1)`.

- `+peekFirst(): T`
  - return front element.
  - Complexity: `O(1)`.
- `+peekLast(): T`
  - return back element.
  - Complexity: `O(1)`.

`PriorityQueue<T>`

Concept: Elements ordered by priority. Backed by a binary heap (`ArrayList<T> heap`).

Fields

- `-ArrayList<T> heap` — binary heap array representation.
- `-Comparator<T> comparator` — comparator used for order. If `null`, require `T` implements `Comparable<T>`.

APIs

- `+add(element: T): void`
  - Insert element into heap, `bubbleUp` to restore heap property.
  - Postcondition: heap property preserved.
  - Complexity: `O(log n)` amortized.
- `+remove(): T`
  - Remove and return element with highest priority (root). Replace root with last element and `bubbleDown(0)`.
  - Precondition: `size > 0`.
  - Complexity: `O(log n)`.
- `+peek(): T`
  - Return root element without removing.
  - Precondition: `size > 0`.
  - Complexity: `O(1)`.
- `-bubbleUp(index: int): void`
  - Helper to move element at `index` up until heap property satisfied.
  - Complexity: `O(log n)`.
- `-bubbleDown(index: int): void`
  - Helper to move element at `index` down until heap property satisfied.
  - Complexity: `O(log n)`.

Pre/postconditions and ordering

- Comparator must be consistent with `equals`. If no comparator provided, `T` must implement `Comparable<T>`.

Heap invariant: for min-heap, `parent <= children` according to comparator; for max-heap, comparator reversed. Document chosen convention (prefer min-heap or allow configurable).

Complexity summary

- `add`: `O(log n)`.
- `remove`: `O(log n)`.
- `peek`: `O(1)`.

Edge cases
Support bulk construction (`PriorityQueue(Collection<T>, Comparator<T>)`) using heapify in `O(n)` time .

`GenericList<T>`

Concept: Full-featured list with index-based access and sorting.

APIs

- `+addLast(element: T): void`
  - append at end.
  - Complexity: `O(1)` amortized.
- `+remove(): T`
  - remove from end (or `removeAt` is primary).
  - Complexity: `O(1)` amortized for end removal.
- `+peek(): T`
  - peek last element.
  - Complexity: `O(1)`.
- `+addFirst(element: T): void`
  - insert at head.
  - Complexity: `O(1)` for linked list, `O(n)` for array list.
- `+addAt(index: int, element: T): void`
  - insert at index shifting subsequent elements.
  - Precondition: `0 <= index <= size`.
  - Complexity: `O(n)` worst-case.
- `+removeFirst(): T`
  - remove from head.
  - Precondition: `size > 0`.
  - Complexity: `O(1)` for linked list, `O(n)` for array list.
- `+removeLast(): T`
  - remove from tail.
  - Complexity: `O(1)`.
- `+removeAt(index: int): T`
  - remove and return element at index.

  - Precondition: `0 <= index < size`.
  - Complexity: `O(n)` worst-case.

- `+get(index: int): T`
  - return element at index.
  - Precondition: `0 <= index < size`.
  - Complexity: `O(1)` for array-backed, `O(n)` for linked.

- `+sort(comparator: Comparator<T>): void`
  - stable sort using merge sort algorithm.
  - Postcondition: elements ordered according to comparator.
  - Complexity: `O(n log n)` time, `O(n)` auxiliary.

- `-mergeSort(left: int, right: int, comparator: Comparator<T>): void`
  - recursive helper.
  - Complexity: `O(n log n)` aggregate.

- `-merge(l: int, m: int, r: int, comparator: Comparator<T>): void`
  - merge helper.

- `-getMiddleIndex(l: int, r: int): int`
  - compute middle index for merges.

Stability
Merge sort implementation must be stable; preserve equal-element order.

Complexity notes
Many operations depend on chosen backing (array vs linked).We will choose array-backed for random-access performance and implement `addAt`/`removeAt` with shifts.

`Comparable<T>` (<>)

API

- `+compareTo(other: T): int`
  - Returns negative if `this < other`, zero if equal, positive if `this > other`.
  - Implementation classes must define natural ordering.

Contract

- Consistent with `equals` when used in `PriorityQueue` or `GenericList.sort` when comparator absent.

`Comparator<T>` (<>)

API

- `+compare(a: T, b: T): int`
  - Returns negative if `a < b`, zero if equal, positive if `a > b`.

Usage

- Passed into `PriorityQueue` and `GenericList.sort` to define ordering. Implementations must be transitive and consistent.

---

4) Cross-cutting design decisions

Error handling policy
Choosing one consistent policy for empty-container operations: either throw `NoSuchElementException` for `remove`/`peek`, or return `null`/`Optional<T>`.

Nulls
Define whether `null` is an allowed element. If allowed, document how `contains`, `equals`, and comparators handle `null`. Recommend disallowing `null` for simplicity or require `Objects.requireNonNull(element)` in `add`.

Thread-safety
Implementations are not thread-safe by default. If thread-safety is required, a synchronized wrapper or explicit concurrency control must be added.

Iteration
An iterator interface for each container that iterates in the container's logical order. Iterator must implement fail-fast behaviour (detect structural modification).

Serialization / persistence
`toArray()` and `fromArray()` utility methods if persistence or testing harness needs snapshots.

Generics & type constraints
For ordering operations, accept either `Comparator<T>` or require `T extends Comparable<T>`. Document both code paths.

---

5) Implementation roadmap

Phase A — Foundations

1. Defining project-wide policies: error handling, null policy, backing-collection choices, naming conventions, test harness framework.
2. Implement `ListContainer<T>` with `elements`, `size`, `isEmpty`, `size`, `clear`, `update`, `contains`, `toString`.
3. Implement a robust `Node<T>` internal type only if using linked-list backing.

Phase B — Core concrete containers

4. Implement `Stack<T>` using `ListContainer` helpers; test `push`/`pop`/`peek`.
5. Implement `Queue<T>` using a linked or circular buffer backing; implement `enqueue`/`dequeue`/`peek`.
6. Implement `Deque<T>` with `O(1)` both-ends operations.
7. Implement `GenericList<T>`:

- Index-based API: `addAt`, `removeAt`, `get`, `addFirst`/`removeFirst`, `addLast`/`removeLast`.
- Implement merge sort and merge helpers; ensure stability.

8. Implement `PriorityQueue<T>`:
    - heap array list, comparator support, `bubbleUp`, `bubbleDown`, `swap`.
    - Optional: bulk heapify constructor.

Phase C — Polishing

9. Implement iterators and `toArray()` snapshots.
10. Implement proper `toString()` formatting across classes.
11. Add input validation, `Objects.requireNonNull` checks, and detailed error messages.
12. Add a comprehensive Javadoc (or language-equivalent) for every API: pre/postconditions, complexity, exceptions.

Phase D — Testing, optimization, and docs

13. Build test harness and run full test suite (see testing plan).
14. Create example usage docs and small demo CLI/test program illustrating each container.

---

6) Testing plan

Testing is grouped into unit tests, integration/regression tests, property tests, and performance/scale tests. Each test suite identifies exact inputs and expected outputs.

Test framework

Using a unit test framework. Since we will be implementing in Java, we will use JUnit. Tests must run in CI.

Unit tests (for every class)

Empty container tests

- `isEmpty()` on new container -> `true`.
- `size()` on new container -> `0`.
- `peek()` and `remove()` behavior on empty container -> documented exception or null.

Single-element tests

- Add one element, verify `size == 1`, `peek()` returns that element, `remove()` returns and leaves empty.

Multiple-element correctness

- Add sequence of distinct elements and verify `remove()`/`pop()`/`dequeue()` order matches semantics (LIFO for `Stack`, FIFO for `Queue`, both ends for `Deque`, priority order for `PriorityQueue`).

Update/contains

- `update(index, value)` on valid index updates element; invalid index returns false (or throws) per API.
- `contains(value)` true for present values, false otherwise; test with duplicates.

GenericList index operations

- `addAt`, `removeAt`, `get` for first, middle, last indices; test out-of-bounds behaviour.

PriorityQueue ordering

- Use primitive comparator and custom comparator to test min-heap/max-heap behaviour, and verify correctness with duplicates.

Sorting tests

- For `GenericList.sort`, create cases: random, sorted, reverse-sorted, duplicate keys. Verify final order and stability (equal elements retain original relative order).

Iterator tests

- Iterator over each container yields elements in logical order.
- Concurrent modification detection (fail-fast) if implemented.

Edge-case tests

- Very large lists (e.g., `N = 100k` or larger depending on environment).
- Duplicate elements, negative values, extreme comparator behaviour.
- `null` insertion policy tests (if `null` allowed or rejected).
- Comparator that violates transitivity — check defined behaviour; tests should confirm appropriate exception or undefined behaviour documented.

Property-based tests

Use property testing to verify invariants over many random sequences:

- For `Stack`: push sequence then pop sequence yields reversed order.
- For `Queue`: enqueue sequence then dequeue yields same order.
- For `PriorityQueue`: for random sequences, successive `remove()` returns non-decreasing (or non-increasing) sequence per comparator.

Integration tests

- Compose containers (e.g., use `Deque` operations then feed into `PriorityQueue`), verify interplay and correctness.
- Serialize to array -> reconstruct container -> compare element sequences.

Performance & stress tests

- Benchmark `add`, `remove`, `get`, `sort`, and `heapify` with increasing sizes (`N = 1k, 10k, 100k, 1M` as environment permits). Record time and memory.
- Validate `sort` is `O(n log n)` empirically and `PriorityQueue.add` is `O(log n)`.

Correctness matrices / acceptance criteria

- All unit tests pass in CI.
- No structural invariants broken after any public API call under test.
- `sort` results match reference library sort for many randomized inputs.
- Memory usage within expected bounds; no leaks in languages that require manual memory management.

Test data examples

- Sequence tests: `[1,2,3,4,5]`, `["a","b","b","c"]`, random ints, objects with custom comparator (e.g., by key).
- Sorting stability tests: objects with key and original index; after sort with equal keys, original index order must be preserved.

---

7) Example test cases (short list)

Stack sequence

```
push(1), push(2), push(3) -> peek() == 3 -> pop() == 3 -> pop() == 2 ->
pop() == 1 -> isEmpty true.
```

Queue sequence

```
enqueue("a"), enqueue("b"), enqueue("c") -> dequeue() == "a" -> peek() ==
"b".
```

Deque sequence

```
addFirst(1), addLast(2), addFirst(0) => removeLast() == 2, removeFirst() ==
0.
```

PriorityQueue (min-heap) with comparator natural int order

```
add(5), add(1), add(3), remove() yields 1, then 3, then 5.
```

GenericList.sort stability

```
elements: [(key=1, id=0),(key=1,id=1),(key=0,id=2)] after sort: (0,id=2),
(1,id=0),(1,id=1).
```

Update & bounds

- add 3 items, update index 1 to new value -> `get(1) == new value`.
- `update(-1)` or `update(3)` -> returns false or throws `IndexOutOfBoundsException` depending on policy.

---

9) Risk analysis & mitigation

Risk: Wrong backing structure choice causes performance pathological cases (e.g., array-backed queue with `O(n)` dequeues).
Mitigation: Choose backing per container: `Stack`: array list; `Queue`: linked list or circular buffer; `Deque`: `ArrayDeque` or doubly-linked list; `GenericList`: array-backed for random access or provide two implementations if needed.

Risk: Comparator inconsistencies break priority queue.
Mitigation: Validate comparator (non-null where required) and document that comparators must be transitive; add runtime checks for comparator exceptions.

Risk: Sorting instability.
Mitigation: Implement merge sort carefully to maintain stability.