Complexity Analysis

1. Linked List Choice

Chosen structure: **Doubly Linked List (DLL)**.

Reasoning:

- **Insertion/Deletion:** O(1) at head/tail or after a known node, because both `next` and `prev` are stored. Singly Linked List (SLL) needs O(n) to delete a middle node.
- **Bidirectional traversal:** DLL allows forward and backward iteration for debugging, dependency tracing, or cleanup.
- **Iteration:** O(n) sequential traversal is simple; overhead from extra pointer is negligible for small/medium lists.
- **Circular Linked List (CLL):** adds wrap-around complexity and complicates termination checks, not useful for task scheduling.
- **Vector:** good for random access but poor for frequent insert/delete (O(n) shifts).

Conclusion: DLL balances flexibility and simplicity for dynamic task management.

2. Time and Space Complexity Analysis

Component	Operation	Time	Space	Reason
LinkedList	`insertTask`	O(1)	O(1)	Insert at tail
	`deleteTask`	O(n)	O(1)	Linear search by ID
	`findTask`	O(n)	O(1)	Sequential traversal
Iterator (TaskIterator)	`hasNext`	O(1)	O(1)	Single pointer check
	`next`	O(1)	O(1)	Advance pointer
PriorityQueue (sorted DLL)	`enqueue`	O(n)	O(1)	Find correct position by priority
	`dequeue`	O(1)	O(1)	Remove head (highest priority)
Stack (ExpressionEvaluator)	`push` / `pop`	O(1)	O(1)	Linked-list node ops
	Infix→Postfix conversion	O(n)	O(n)	Each token processed once

Component	Operation	Time	Space	Reason
	Postfix evaluation	O(n)	O(n)	Each operator/operand handled once
Recursive dependency check	`hasCycleHelper`	O(n)	O(n)	DFS over task list using recursion stack

3. Trade-offs Discussion

- **DLL vs SLL:** DLL uses slightly more memory per node but enables backward traversal and constant-time deletion with known node pointers.
- **DLL vs CLL:** CLL avoids null checks but complicates iterator logic and cycle detection; DLL simpler and safer.
- **LinkedList vs Vector:** Vector provides random access (O(1)) but expensive insertions (O(n))—not ideal for dynamic scheduling.
- **Priority Queue design:**
  - **Sorted list:** `enqueue` O(n), `dequeue` O(1). Efficient when dequeuing is frequent (as in schedulers).
  - **Unsorted list:** `enqueue` O(1), `dequeue` O(n). Better if many inserts before few dequeues.
  - Chosen: **sorted list**, since scheduling repeatedly extracts the next high-priority task.

This structure mix minimizes complexity while keeping clarity and recursive operations manageable.