Member name:

Sokdany Monyreach
Mey Chansopheaktra,
Thang Saoly,
Keav Sokhamuny,

Rules for every challenge

- **Do all setup (filling, seeding indices, printing) outside the timed block.**
- **Put only the target work between t0 and t1.**
- **Repeat ≥5 trials and average. Try multiple n (e.g., 1e3, 1e4, 1e5) where it makes sense.**
- **Hand in: your code, a small table of times, and a 2–4 sentence explanation per part.**

Part A — Circular & Doubly Linked List mastery

A1 — CSLL: tail-to-head wrap vs manual reset

- **Implement traversal of n nodes in two ways:**
  (i) CSLL with tail->next = head and a single loop of n steps;
  (ii) Non-circular SLL that restarts at head whenever you hit nullptr.
- **Predict which is faster and why (branching, cache/predictability).**
- **A cache miss is typically much slower than a branch misprediction because the penalty of waiting for an external resource (RAM) is far greater than the penalty of a small pipeline rollback.**
- **Measure and explain.**

The key difference comes down to the source of the delay:

- **Cache Miss $\to$ RAM: Delay is caused by data unavailability from a slow, distant source (RAM). The penalty is in the hundreds of cycles.**
- **Branch Misprediction: Delay is caused by incorrect control flow requiring a pipeline reset, which is a fast, localized operation. The penalty is in the tens of cycles.**

Therefore, although both are significant performance concerns, the cache miss to main memory is much slower because it forces the CPU to wait for a fundamental resource (data) from the slowest part of the memory hierarchy.

-

A2 — CSLL deletion with/without predecessor

- **Case 1:** Delete a given node when you also have its predecessor (O(1)).
- **Case 2:** Delete the same node when you only have the node pointer (must find predecessor).
- **Predict cost difference; measure for random positions; explain the curve.**

A3 — Rotate-k on CSLL vs SLL

- **Implement "rotate right by k":** in CSLL it's pointer moves; in SLL it's find-break-relink.
  1. In CSLL:
     - The list forms a circle because the last node points back to the head.
     - To rotate right by k, compute **k = k % n** for effective rotations.
     - Move the head pointer forward by **(n - k)** steps.

- ❖ Since it's circular, no nodes need to be detached or relinked.
- ❖ Just update the head pointer to the new node at **(n - k)** steps ahead.

2. In SLL:
   - ❖ The last node points to nullptr, so the list is linear.
   - ❖ Compute **k = k % n**.
   - ❖ Traverse to the **(n - k)**th node (this will be the new tail).
   - ❖ Find the last node (original tail).
   - ❖ Break the link after the new tail by setting its **next** to **nullptr**.
   - ❖ Link the original last node's **next** to the original head.
   - ❖ Set the new head pointer to the node after the new tail.

- **Test for multiple k (small, n/2, n−1).**
  - When k is small, both methods are fast.
  - When k = n/2:
    - CSLL rotation still takes O(k) pointer moves.
    - SLL rotation requires traversal and relinking, which is O(n).
  - When k = n - 1:
    - CSLL still just moves pointer.
    - SLL requires nearly full list traversal and relinking.
- **Decide** which wins and under what k ranges.
  - CSLL wins for all k because rotation is just pointer moves require no relinking. But
  - SLL rotation is requiring traversal and relinking.

A4 — DLL vs SLL: erase-given-node

- **Build DLL with prev.** Given a pointer to any node, erase it.
- **Compare to SLL** erase with known predecessor and SLL erase without predecessor.
- **Predict → measure → explain O(1) vs O(n) and constant-factor hits.**

A5 — Push/pop ends: head-only vs head+tail

- Implement push_front/pop_front and push_back/pop_back on:
  (i) SLL with head only, (ii) SLL with head+tail, (iii) DLL with head+tail.
- **Benchmark each op for random mixes (e.g., 70% push_back, 30% pop_front).**
- **Explain why tail changes the story.**

A6 — Memory overhead audit

- **For the same logical dataset size n, allocate SLL, CSLL, and DLL nodes.**
- **Report bytes per node (pointer count), total bytes, and measured allocation time.**
- **Discuss the time–space trade-off you'd choose for frequent middle deletions.**

Part B — Real-world use cases

B1 — Recent Items Tray (add/remove at the same end)

- **Build a "recent items"** tray where the most recently added item is always the next one removed.
  ->This is created by putting what's add/remove on record, creating another and always putting it to the head pointer when inserting new or put into what just removed.
- **Implement with:**

A) Singly linked nodes adding/removing at the front.
B) Doubly linked nodes adding/removing at the front

- **Predict which is faster** (pointer count vs rewiring), measure adds/removes (mixed workload), explain whether the second pointer helps here.
  ->Total pointer ops (mixed add/remove workload):
  If you do 50% adds + 50% removes:

  Singly: ~4 pointer writes per full add/remove cycle

  Doubly: ~7 pointer writes per cycle
  So doubly linked list performs **~75% more pointer updates** for the same logical workload.

B2 — Editor Undo History

- **Implement an Undo history where each new action is "placed on top," and Undo removes the last action added.**
- **Version A: Singly linked front-add/front-remove.**
- **Version B: Dynamic array (grow by doubling).**
- **Workload: 80% add actions, 20% undo actions.**
  ->

Metric	Singly Linked List	Dynamic Array
Add / Undo time	O(1) always	Amortized O(1), but realloc spikes
Memory locality	Poor (heap scattered)	Excellent (contiguous)
Memory overhead	1 pointer per node	Some unused capacity after growth
Realloc spikes	None	Yes (occasional copies)
Cache performance	Low	High
Predictability	Constant	Occasional stalls

- **Predict throughput and memory spikes (reallocs) vs constant-time links; measure and justify which design you'd ship.**
  ->Ship Version B (Dynamic Array)
  Because 80% adds, 20% undos → append/pop pattern fits a stack perfectly, and amortized O(1) realloc cost is negligible compared to improved cache performance.