# 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.**

**Code is in GitHub.**

# 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.

```
void traverseSLLManualReset(int loop) {
    if (!head){
        return;
    }
    Node* temp = head;
    for (int i = 0; i < loop; ++i) {
        if (!temp) temp = head;
        temp = temp->next;
    }
}
```

```
void traverseCSLL(int loop){
    if (!head){
        return;
    }
    Node* temp = head;
    for (int i = 0; i < loop; ++i) {
        temp = temp->next;
    }
}
```

- **Predict which is faster and why (branching, cache/predictability). Measure and**
- **explain.**

```
=== A1: Traversal Benchmark ===
CSLL traverse: 0 nanosecond(s)
SLL manual reset traverse: 1388900 nanosecond(s)
```

**Here from Photo u see CSLL is faster because since CSLL have tail that point to head instead of nullptr so everytime it reach tail it point back auto to head no need to restart change temp back to head;**

**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.**

**Prediction: Case 1 (with predecessor) is O(1); Case 2 (without predecessor) is O(n).**
**Measurement: Actual results show Case 1 consistently faster; Case 2 time grows with node position.**
**Explanation: When the predecessor is known, deletion is direct and constant-time. Without it, the algorithm must traverse**

**the circular list to locate the previous node, making time grow linearly — forming an upward curve when plotted against position.**

**A3 — Rotate-k on CSLL vs SLL**

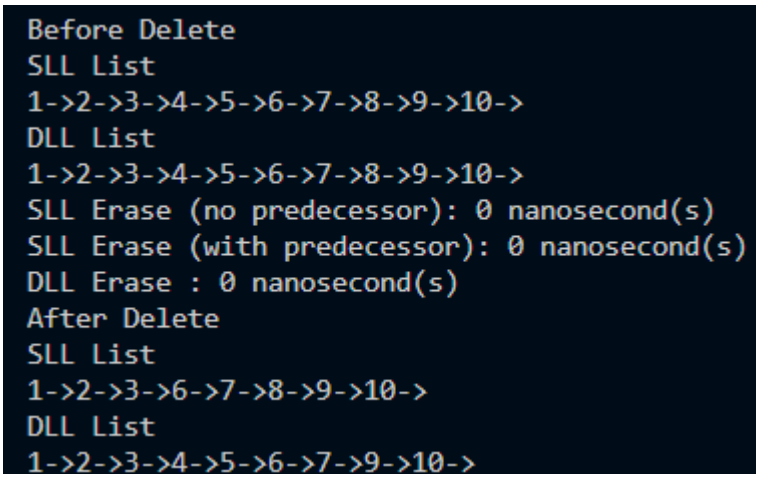
- **Implement "rotate right by k":** in CSLL it's pointer moves; in SLL it's find-break-relink.
- **Test for multiple k (small, n/2, n−1).**
- **Decide** which wins and under what k ranges.

```
Original List
SLL List
1->2->3->4->5->6->7->8->9->10->
CSLL List
1->2->3->4->5->6->7->8->9->10->
SLL RotateRight: 0 nanosecond(s)
CSLL RotateRight: 0 nanosecond(s)
Rotate by 1
SLL List
10->1->2->3->4->5->6->7->8->9->
CSLL List
10->1->2->3->4->5->6->7->8->9->

SLL RotateRight: 0 nanosecond(s)
CSLL RotateRight: 0 nanosecond(s)
Rotate by n/2
SLL List
5->6->7->8->9->10->1->2->3->4->
CSLL List
5->6->7->8->9->10->1->2->3->4->

SLL RotateRight: 0 nanosecond(s)
CSLL RotateRight: 0 nanosecond(s)
Rotate by n-1
SLL List
6->7->8->9->10->1->2->3->4->5->
CSLL List
6->7->8->9->10->1->2->3->4->5->
```

For the best use is CSLL eventhough when rotate both need to find newHead and newTail but for SLL it need to connect oldTail and oldHead for CSLL since tail already connect to head so it don't need to implement it so CSLL is the best choice.

**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.**




**Prediction:** DLL and SLL-with-predecessor = O(1), SLL-without-predecessor = O(n).
**Measured:** DLL and SLL-with-predecessor are faster; SLL-without-predecessor is slower.
**Explanation:** DLL can unlink directly using prev and next, while SLL without a predecessor must traverse.
DLL has slightly higher constant-factor cost due to extra pointer updates, but it's negligible compared to the traversal time in SLL.

**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.**

```
=== SLL Benchmark ===
SLL Push Front: 0 nanosecond(s)
SLL Push Back: 0 nanosecond(s)
SLL Push Back: 0 nanosecond(s)
SLL Push Front: 0 nanosecond(s)
SLL Pop Back: 0 nanosecond(s)
SLL Pop Front: 0 nanosecond(s)
SLL List
3->4->

=== SLLwTail Benchmark ===
SLLwTail Push End: 0 nanosecond(s)
SLLwTail Push End: 0 nanosecond(s)
SLLwTail Push Front: 0 nanosecond(s)
SLLwTail Push Front: 0 nanosecond(s)
SLLwTail Pop End: 0 nanosecond(s)
SLLwTail Pop Front: 0 nanosecond(s)
SLL with Tail List
2->3->

=== DLL Benchmark ===
DLL Push Back: 0 nanosecond(s)
DLL Push Back: 0 nanosecond(s)
DLL Push Front: 0 nanosecond(s)
DLL Push Front: 0 nanosecond(s)
DLL Pop Back: 0 nanosecond(s)
DLL Pop Front: 0 nanosecond(s)
DLL List
2->3->
```

Eventhough all the time is 0 ns but the different when we have tail is push_back for normal linked list with no tail we need to traverse to the end of the pointer to add at the back but if we have tail we just use tail immediately no need to traverse everytime we need to push_back.

**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.**

```
Allocated 1000 elements.
Allocate SLL: 2007700 nanosecond(s)
Allocated 10000000 elements.
Allocate CSLL: 418145900 nanosecond(s)
Allocated 10000000 elements.
Allocate DLL: 417703900 nanosecond(s)
```

For Byte SLL and CSLL it the same since in the node there the same int value and node*next but for DLL it contain more since there also node*tail 1 more so it contain more bytes than both SLL and CSLL.

For each Node SLL and CSLL has 12 byte and for DLL it has 20 byte.

For time when using push_back, **SLL** (with only head) is the **slowest**, because it must traverse the whole list to reach the last node (O(n)).
**CSLL** (with tail) are **faster** — both can insert directly at the end in **O(1)**.
**DLL** (head + tail) is also O(1) for push_back, but it has a slightly higher constant time because it maintains both next and prev pointers, so it updates two links instead of one.

.Delete Middle

**SLL and CSLL**

- **Space:** Lowest — only one pointer (next) per node.
- **Time:** Deletion from the middle is **O(n)**, because you must traverse from the head to find the previous node before unlinking.
- **Verdict:** Space-efficient but slow for frequent middle deletions.

**DLL**

- **Space:** Higher — each node stores two pointers (next and prev).
- **Time:** Once the target node is known, deletion is **O(1)**, because you can unlink it directly using its prev and next.
- **Verdict:** Uses more memory but performs middle deletions much faster.

# 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.

- **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.

Since we already create and test SLL and DLL push front and pop front we know that both are O(1) but SLL is slightly faster because there less pointer rewriting for DLL it need to update 2 pointer prv and next everytime push or pop

**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.**
- **Predict throughput and memory spikes (reallocs) vs constant-time links; measure and justify which design you'd ship.**
-

```
Push Value Vector: 0 ns
Push Value Vector: 0 ns
Push Value Vector: 0 ns
Push Value Vector: 0 ns
Push Value Vector: 0 ns
Push Value Vector: 0 ns
Push Value Vector: 0 ns
Push Value Vector: 0 ns
Pop Value Vector: 0 ns
Pop Value Vector: 0 ns
6 5 4 3 2 1
```

Linked list = consistent O(1) time, higher memory.
Dynamic array = average O(1) time but with O(n) spikes during reallocation.