Week03 - Lab Practice Report

Group 6:
- Chea Kuyhong
- Chhuong Sophakvatey
- Ean Mana
- Hout Mengleang
- Ly Sokunnita
- Nget Sokunkanha
- Nuttsotea Tireachh

Part A – CIRCULAR & DOUBLY LINKED LIST MASTERY

*Each test used n = 10,000 nodes

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

- The Circular Singly Linked List (CSLL) is faster for traversal because it avoids branch checks and benefits from predictable memory access.
- Measure result:

Structure	Steps	Time (ms)	Branches per loop
CSLL	50,000	0.42	0
SLL	50,000	0.61	1

Both are O(n) in time complexity, but CSLL has lower constant overhead.

- Code explanation:
  *CSLL:
  - if (head == nullptr) return; -> prevents dereferencing null when list is empty.
  - Node* current = head -> starts traversal at the beginning of the circular list.
  - for (int i = 0; i < steps; i++) ->fixed-length traversal; no nullptr condition since the list is circular.
  - current = current->next; -> moves forward safely because in CSLL, tail->next points back tif (head == nullptr) return;
    o head.
  - This loop is predictable (no branching) -> CPU cache friendly.

  *SLL:
  - In SLL, the traversal ends when current == nullptr.
  - To simulate circular behavior, we manually reset the head.
  - if (current == nullptr) introduces branching, which slows the loop slightly.
  - Each cycle incurs a small CPU penalty due to conditional evaluation.

- Circular traversal removes if-null checks → better for repetitive looping (e.g., round-robin scheduling).
- Non-circular lists are simpler but slightly slower for repeated passes.

A2. CSLL deletion with/without predecessor:

- Case 1 : pointer re-linking is constant time (O(1)). No searching required.
- Case 2 : an O(n) search is required to locate the previous node.
- Result :
- On average, deletion without a predecessor takes ~5x longer for 10,000 nodes

Case	**Time (ms)**	**Time Complexity**	**curve**
1	0.004	O(1)	flat
2	0.021	O(n)	Rising linear

- Curve explanation :
- Case 1 is a flat line because the cost is constant because we already know the node's predecessor. The operation only involves updating one link and freeing memory, both O(1) actions.Therefore, the measured time remains nearly constant for all node positions.
- Case 2 is a rising linear curve because the algorithm must traverse the list from the head to locate the node's predecessor before deletion. The traversal time increases linearly with the node's position, making the cost proportional to the list length.
- Code explanation :

  *case 1:
  - The traversal keeps track of prev
  - Once cur reaches target, you re-link prev->next and delete cur.
  - prev->next = nullptr ensures proper disconnection before freeing memory.

  *case 2:
  - When the predecessor is missing, we **c**opy the next node's data into the target node.
  - Then delete the next node instead — a common trick for O(1) deletion.
  - Limitation: fails if target is the last node (no next node).

A3. Rotate-k CSLL vs SLL :

- Time table :
  * CSLL rotate right performance :

k value	Run1	Run2	Run3	Run4	Run5	Avg (ns)
1	380ns	180ns	210ns	200ns	240ns	242ns
5	120ns	60ns	60ns	70ns	70ns	76ns
9	80ns	40ns	60ns	40ns	40ns	52ns

  * SLL rotate right performance :

k value	Run1	Run2	Run3	Run4	Run5	Avg (ns)
1	240ns	350ns	240ns	310ns	330ns	294ns
5	120ns	100ns	140ns	130ns	91ns	114ns
9	70ns	90ns	70ns	80ns	80ns	78ns

- For all tested k values (1, 5, 9), CSLL performs faster. Reason: CSLL rotation only moves pointers (O(k)), while SLL requires finding the tail and relinking nodes (O(n)). The difference becomes more noticeable as the list size grows.
- Code explanation :

  * CSLL code :

    - Uses modular arithmetic (k % n) to prevent redundant rotations.
    - In CSLL, rotation only needs head adjustment because tail->next already loops.
    - No breaking/reconnecting — just move pointer references.

  * SLL code :

    - SLL must explicitly rewire the end and start nodes.
    - Finding both tail and newTail requires two traversals, O(n) cost.
    - Once found, we “cut and paste” the list to form the rotated version.

A4. DLL vs SLL: erase given-node :

- DLL erase-given-node: In a doubly linked list, each node has both prev and next pointers. Therefore, when a node pointer is given, we can remove it directly by updating its neighbors. This operation runs in O(1) time.
- SLL erase with known predecessor: If we have the pointer to the predecessor node, deletion is also O(1) — we just re-link pred->next = target->next. The cost is similar to the DLL case, differing only by a small constant factor.
- SLL erase without predecessor: Without a predecessor, we must first find it by traversing the list from the head, making this an O(n) operation. The time increases linearly with list size, as seen in the timing results.

n	**DLL O(1)**	**SLL O(1)**	**SLL O(n) no preed**
1000	0.4	0.5	20.3
10000	0.5	0.6	190.7
100000	0.6	0.7	1900.2

- Code explanation :
  *DLL code :
  - Handles all cases: deleting first, last, or middle node.
  - No traversal is required since the target has both prev and next.
  - Bidirectional links make this an O(1) operation.
  - DLL's prev pointer eliminates dependency on predecessor knowledge.

  *SLL code :
  - if you already know the predecessor, re-link and delete in O(1).
  - If not, you must find it first (O(n) traversal).
  - SLL sacrifices flexibility for a smaller memory footprint.

A5. Push/pops ends: head-only vs head+tail :

- SLL head-only: Must traverse the entire list to find tail → O(n).
- SLL head+tail: Directly access tail pointer → O(1).
- DLL head+tail : Both ends accessible with prev/next pointers → O(1).
- Measure result :

Structure	**Push_back Complexity**	**Pop_up Complexity**	**Time (ms)**	**Average per op (ns)**
SLL head-only	O(n)	O(n)	1,180	11,800
SLL head+tail	O(1)	O(n)	440	4,400

DLL head+tail	O(1)	O(1)	180	1,800

- Tail changes story because:
  - With tail :
    - push_back becomes instantaneous, since we can directly access the last node.
    - Even though pop_back in SLL HeadTail still needs a traversal (to find a new tail), the majority of cost in push_back is eliminated, drastically lowering total time.
  - Without tail :
    - Each push_back requires a full traversal from head to tail.
    - Time grows linearly with list length.
    - Repeating this thousands of times -> massive accumulated cost.
- Code explanation :
- SLL head-only : must traverse from head to tail for every insertion at back.
- SLL head+tail : uses a stored tail pointer for constant-time append. Removes need for traversal.
- DLL head+TAIL : maintains prev and next, so push and pop from both ends are O(1). Used for deque and bidirectional lists.

A6. Memory overhead audit :

- Bytes per node :

Structure	**Pointers**	**Data size**	**Pointers size**	**Total bytes Per node**
SLL	1 (next)	4 bytes	8 bytes	12 bytes
CSLL	1 (next)	4 bytes	8 bytes	12 bytes
DLL	2 (next + prev)	4 bytes	16 bytes	20 bytes

- Total bytes for n = 10,000 nodes : *Total bytes = bytes per node x n*

Structure	**Bytes per node**	**Total bytes**	**Approx. in KB**
SLL	12 bytes	120,000 bytes	117.2 KB
CSLL	12 bytes	120,000 bytes	117.2 KB
DLL	20 bytes	200,000 bytes	195.3 KB

1KB = 1024 bytes

- Measured time allocation :

Structure	Time allocation (ms)	Average per node (ms)
SLL	0.72 ms	0.072 ms
CSLL	0.75 ms	0.075 ms
DLL	1.10 ms	0.110 ms

- In frequent middle deletion O(1) for middle deletion is critical in systems where **li**sts are large or deletions occur frequently.
  O(1) ensures: Constant-time performance (independent of list length) with much faster execution for large dataset, more predictable latency and better scalability in real applications.
- Even though a DLL costs slightly more memory, the time savings from O(1) middle deletion make it a superior choice in most performance-sensitive use cases.

PART B – REAL WORLD USE CASES

n = 10,000 ops

B1. Recent items tray (add/remove at the same end) :

- Measure result :

Structure	Total time(ms)	Avg time per Ops (ms)	Memory per node	Pointers writes
SLL	1.9 ms	0.19 ms	12 bytes	1
DLL	2.3 ms	0.23 ms	20 bytes	2

- For a recent items tray where all actions are at the front, both SLL and DLL is very fast since there's no traversal. DLL is slightly slower due to the extra prev pointer updates, so SLL is the best choice because:
  - Fewer pointer updates
  - Less memory per node
  - same time complexity (O(1)) as DLL
  - Simpler code
  - DLL adds unnecessary pointer rewiring and memory cost without any performance gain.
- Second pointers doesn't help here because :
- All operations are at the front.

- We don’t need to go backward, so prev isn’t used functionally.
- It only adds memory overhead and slight extra cost per operation.

B2. Editor undo history :

- Predict :

  Throughput:

  - Vector is expected to be faster on average due to contiguous memory and better cache locality.
  - SLL will be slightly slower due to pointer dereferencing and frequent allocations.

  Memory behavior:

  - Vector will grow in chunks, doubling capacity and causing temporary spikes in memory use and latency during reallocation.
  - SLL grows node by node, leading to flat, predictable memory usage.

  Latency:

  - SLL: Strict O(1) per operation, no spikes.
  - Vector: Amortized O(1), but with short spikes during capacity growth.
- Result :

Structure	**Total time (ms)**	**Avg time/op (ms)**	**Peak memory (KB)**
SLL	3.5 ms	0.35 ms	~120 KB
Vector	2.7 ms	0.27	~190 KB

- The design i’d ship :

Both the singly linked list (SLL) and dynamic array (std::vector) achieve O(1) push and pop on average but differ in performance characteristics.

SLL offers stable, predictable latency with no spikes, making it suitable for unbounded or real-time workloads. However, it has lower cache efficiency and higher allocation overhead.std::vector provides higher throughput and better memory locality, but normally introduces occasional reallocation spikes. Since the workload size is fixed, using reserve() eliminates these spikes entirely, resulting in fast and stable performance.

I would ship the std::vector implementation because it delivers higher throughput, efficient memory use, and stable performance after pre-allocation. SLL would only be preferable if the history size were unbounded or required strict real-time guarantee.