WEEK-3

Lab Practice Assignment Instructions

❖ Part A _ Circular & Doubly Linked List mastery

➢ A1 _ CSLL: tail-to-head wrap Vs manual reset

Temp	1st	2nd	3rd	4th	5th	average
CSLL with tail->next = head and a single loop of n steps						
	8500	8400	8450	8420	8600	8450
Non-circular SLL that restarts at head whenever you hit nullptr						
	12500	12600	12400	12550	12560	12520

CSLL is the faster Because

- CSLL don't need branch misprediction penalties.
- CSLL is better cache locality from circular access.
- Efficient pipeline utilization.

➢ A2 _ CSLL: delete with and without predecessor

Temp	1st	2nd	3rd	4th	5th	average
Delete a given node when you also have its predecessor (middle node)						
	1100	2400	2400	1600	1500	1800
Delete the same node when you only have the node pointer (head)						
	1400	1300	2200	2100	1200	1640
Delete the same node when you only have the node pointer (tail)						
	1900	4800	1200	2500	1100	2300
Delete a given node when you also have its predecessor (last node)						
	2000	1900	1300	2100	2000	1860

+ **Predict cost different**

  The reason that cost different is with predecessor its' time complexity is 0 (1) (constant time per deletion) and without predecessor its' time complexity is 0 (n) (it needs 0(n) time to find the predecessor.

+ **Explain curve**

With predecessor: linear growth (fix cost per element and Don't dependency on list size).

Without predecessor: Quadratic growth (dependency on the list size and need nested loops effect)

- **A3_ Rotate-k on CSLL Vs SLL**

CSLL Win all case Because:

- If K is small both need to traverse, but SLL has extra pointer operation.
- If K = n/2 both have to traversal n/2 node, so both are nearly the same.
- If K= n-1 the reason is similar to K is small.

k	CSLL	SLL	CSLL Time (O)	SLL Time (O)	Winner	Reason
1 (small)	Pointer moves (adjust head/tail pointers)	Find-break-relink	O(k) → O(1)	O(n)	**CSLL**	CSLL only adjusts pointers; SLL must traverse list.
n/2	Pointer moves (iterate k nodes)	Find kth, break, relink	O(k) → O(n/2)	O(n)	**CSLL**	Fewer pointer changes, no need to reconnect tail manually.
n−1	Pointer moves (almost full traversal)	Find-break-relink	O(n)	O(n)	**Tie**	Both traverse ~n nodes, but CSLL avoids tail relinking overhead.

- **A4 _ DLL Vs SLL: erase-given-node**

temp	1st	2nd	3rd	4th	5th	Average Nano second
DLL						
	51800	77300	29400	107400	42100	61600
SLL with predecessor						

	119100	161400	57600	220800	84200	128600
SLL without predecessor						
	128700	179000	65800	239500	91200	140840

So the fastest is using DLL.

+ **Explain**

Structure Condition Erase Time Why

DLL Given node pointer O(1) Direct prev/next access

SLL With known predecessor O(1) Predecessor known

SLL Without predecessor O(n) Must traverse to find prev

Constant-factor hits

Even though both DLL and SLL (with predecessor) are O(1), DLL operations are slightly slower in practice (larger constant factor) because:

Each node stores an extra pointer (prev).

More pointer updates (two links instead of one).

Slightly higher memory usage and cache misses.

Conclusion

DLL: Erasing a given node is always O(1).

SLL:

O(1) if you know the previous node.

O(n) if you don't.

DLL is faster asymptotically, but has constant-factor overhead**.**

- **A5 _ Push and Pop ends: head-only Vs head + tail**

+ **Benchmark Example (Theory)**

If you test with 70% push_back and 30% pop_front, here's what happens:

SLL with head only: push_back is slow (O(n)), since you must traverse the whole list to find the last node.pop_front is fast (O(1)).

Result: Poor throughput when push_back dominates.

SLL with head + tail: push_back becomes fast (O(1)), because the tail pointer directly points to the last node. pop_front remains O(1).

Result: Much better performance; tail avoids traversal.

DLL with head + tail:All four operations (push_front, pop_front, push_back, pop_back) are O(1).

Result: Best performance if your workload mixes both front and back operations.

+ **Explain "Why Tail Changes the Story"**

The tail pointer gives direct access to the end of the list. Without it, you must traverse the entire list to reach the last node, which costs O(n) time. So: With head only, operations at the back are slow. Adding a tail makes back operations constant-time. In a Doubly Linked List, you can also move backward easily, so even pop_back becomes O(1).That's why the "story changes" the tail pointer transforms linear-time operations into constant-time ones.

- **A6 _ Memory overhead audit**

As we know int = 4bytes, and ptr = 8bytes

	Int	ptr	Bytes per node	Number of Node	Total bytes
SLL	4 bytes	8 bytes	12 bytes	n	12 x n
CSLL	4 bytes	8 bytes	12 bytes	n	12 x n
DLL	4 bytes	16 bytes	20 bytes	n	20x n

SLL and CSLL's ptr = 8 bytes because it only contains next ptr.
DLL's ptr = 16 bytes because it contains both next and prev ptr.
For delete node in the middle (if we have too many node) we should use DLL(spend less time because it can access both from beginning and the end) because it spend less time then CSLL and SLL, because when we use CSLL and SLL have to traversal until the middle (spend a lot of time) . But (if we don't have too many node) we should use CSLL or SLL.

❖ Part B _ Real – world – use cases

- **B1 _ Recent Items Tray (add/remove at the same end)**

- Predict which is faster

Aspect	Singly Linked Lists	Double Linked List
addFront	O(1), 2 pointer operations	O(1), 3-4 per ops
removeFront	O(1), 2 pointers ops	O(1), 3-4 per ops
Pointer count	1 per node	2 per node
Memory usage	Lower	Higher (+8 bytes/node on 64-bit)
Cache efficiency	Faster (better)	slower

temp	1st	2nd	3rd	4th	5th	average
DLL	1792	1801	1795	1799	1794	1796
SLL	1195	1201	1198	1203	1197	1198

The faster is SLL Because:

- Better cache performance
- Lower memory overhead
- Bidirectional capability not needed for front only operation

- **B2 _ Editor Undo History**
  - Predict

Aspect	Sll(front-add/remove)	Dynamic array (front-add/remove)	Better
Add (push)	O(1) but slow new heap node each time	Amortized O(1) very fast contiguous memory	Array

Undo(pop)	O(1)	O(1)	Tie
Memory behavior	Steady but scattered(1 pointer per node)	Occasional realloc (temporary spike)	Trade off
Cache efficiency	Poor (nodes all over memory)	Excellent (contiguous data)	Array
Throughput (80% add, 20% undo)	Slower (heap overhead)	Much faster overall	Array
Implementation	Manual memory handling	Simple with std::vector	Array

- Measure

Case	SLL	ARRAY
Best case	980000	1100000
Average	950000	1500000
Worst case	920000	50000

- The design we suggest is SLL Because:
  - No memory spikes
  - No Throughput spikes
  - Better worth case