DSA-LAB-Week3

Group 6: Seng Dina, Sor Channorakpitou, Chay SengHab, Ly SrongChhay, Roth SoraYuth

PART A : Circular & Doubly LinkedList mastery

A1: CSLL : tail to head wrap vs manual reset

Test	CSLL	SLL
1	128	161
2	40	123
3	39	136
4	40	134
5	48	193
Average	59 micro s	149.4 Micro s

Traversal 100 Node for Both CSLL and SLL

Predict: The CSLL traversal is faster

Branching: in CSLL the CPU know exactly where we should stop, while SLL need to predict where is nullptr it may have misprediction.

Explanation:

The circular linked list (CSLL) is faster because it doesn't need to check for `nullptr` every time it moves to the next node. It just keeps looping smoothly. This makes it easier for the computer processor to handle since there are fewer checks, and the data is accessed more predictably.

A2: CSLL: deletion with / without predecessor

Test	Without Predecessor	With Predecessor
1	12900 NS	1000 NS
2	12500 NS	800 NS
3	11900 NS	600 NS
4	20400 NS	600 NS
5	11800 NS	400 NS
Average	13900 nano seconds	680 nano seconds

Explanation:

- With Prodecessor: (10000 node delete at node number 5000)

- O(1) just update pointer
- Flat horizontal line - constant time regardless of list size

- Withoud Prodecessor: (10000 node delete at node number 5000)

- O(n) must search for predecessor
- Upward sloping line - time increases linearly with list size

A3: Rotate-K on CSLL vs SLL

Test	Time in CSLL	Time in SLL
1	1	1
2	5	4
3	2	4
4	2	4
5	3	5
Average	2.6 micro sec	3.6 micro sec

Explanation: CSLL is much faster than SLL. But when k is small CSLL is slightly slower than SLL, it depends on exact implementation, but CSLL is avoid relinking while SLL always need to traversal and relinking making it slower than CSLL for rotating k times

So, CSLL win when k is large (close to size) and SLL is always O(n)

A4: DLL vs SLL : erase-given-node

Test	SLL Without Predecessor	SLL With Predecessor	DLL
1	12400	800	900
2	12200	500	500
3	12300	600	600
4	12000	400	400
5	12100	300	400
Average	12200 nano seconds	520 nanoseconds	560 nano seconds

Explanation:

(10000 node delete at node number 5000)

- DLL and SLL with predecessor O(1) speed is similer because we can access the previous node
- SLL without predecessor : O(n) is too slow since we need to traverse from the head until a node before the deleting node .

A5: Push/Pop ends: head-only vs (head + tail)

+ push front

Test	SLL head only	SLL head + tail	DLL Head + Tail
1	200 ns	200 ns	200 ns
2	1300 ns	1100 ns	1700 ns
3	11200 ns	500 ns	800 ns
4	1000 ns	500 ns	1000 ns
5	900 ns	500 ns	600 ns
Average	2920ns	560ns	860ns

+ push back

Test	SLL head only	SLL head + tail	DLL Head + Tail
1	58 ms	300 ns	200 ns
2	89 ms	1000 ns	900 ns
3	80 ms	900 ns	1100 ns
4	56 ms	900 ns	1200 ns
5	66 ms	2100 ns	800 ns
Average	70ms	1040ns	840ns

+ pop front

Test	SLL head only	SLL head + tail	DLL Head + Tail
1	700 ns	700 ns	600 ns
2	900 ns	700 ns	600 ns
3	700 ns	600 ns	400 ns
4	600 ns	400 ns	300 ns
5	700 ns	300 ns	300 ns
Average	720ns	540ns	440ns

+ pop back

Test	SLL head only	SLL head + tail	DLL Head + Tail
1	56 ms	40 ms	600 ns
2	91 ms	67 ms	1100 ns
3	73 ms	57 ms	700 ns
4	47 ms	46 ms	500 ns
5	40 ms	46 ms	400 ns
Average	61.4 ms	51ms	660ns

Explanation:

Operation	SLL head only	SLL head + tail	DLL Head + Tail
Push front	**O(1)**	**O(1)**	**O(1)**
Pop front	**O(1)**	**O(1)**	**O(1)**
Push back	**O(n)**	**O(1)**	**O(1)**
Pop back	**O(n)**	**O(n)**	**O(1)**

SLL with head only:

- push_back requires traversing entire list → O(n)
- pop_back requires finding second-to-last node → O(n)

SLL with head+tail:

- push_back becomes O(1) - direct access via tail
- But pop_back remains O(n) - still need to find predecessor

DLL with head+tail:

- All four operations become O(1)
- pop_back can access predecessor directly via prev pointers

A6: Memory Overhead audit

Test	SLL	CSLL	DLL
Byte per node	**16**	**16**	**24**
Total byte	**16*n**	**16*n**	**24*n**
Allocation time1	**3905 microsecond(s)**	**4204 microsecond(s)**	**5479 microsecond(s)**
Allocation time2	**6197 microsecond(s)**	**7015microsecond(s)**	**7756 microsecond(s)**
Allocation time3	**3754 microsecond(s)**	**5081 microsecond(s)**	**5523 microsecond(s)**
Allocation time4	**4707 microsecond(s)**	**5436 microsecond(s)**	**6685 microsecond(s)**
Allocation time5	**4509 microsecond(s)**	**5258 microsecond(s)**	**6002 microsecond(s)**

Discuss: For frequent middle deletions, I would choose DLL because it has previous that can access previous without traverse like SSL or CSLL, making it constant time O(1) in this task.

PART B: Real-World use cases

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

Implement with Singly linked nodes and doubly linked nodes

Prediction: Singly linked nodes will perform faster than doubly in this case.

Reason: Inserting / Deleting at the front is only work at the front so if we use doubly linked nodes, 2 pointers must rewrite and spend more space for more pointers but if we use singly linked nodes only a pointer must rewrite and maintain and spend less pointer. The most recently added item is always the next one removed using LIFO concept same as stack.

Real time experiment:

Times	SLL	DLL
1	2000ns	1900ns
2	700ns	1100ns
3	800ns	800ns
4	900ns	800ns
5	400ns	1000ns
Average	960ns	1120ns

Assumption: This instruction using the singly linked list is a better choice than doubly one, we can handle it well and save memory as well. Singly faster than doubly linked list as expected.

B2: Editor/Undo history

Implement with Singly linked nodes and Dynamic Array

Prediction: We expect SIngly Linked List to perform faster for Insert/Delete Front operations because these actions only modify pointers and do not require shifting element.

Real time experiment:

Times	SLL (insert)	SSL (delete)	Dynamic Array
1	2100 ns	100 ns	1400ns
2	2400 ns	100 ns	200ns
3	3100 ns	100 ns	100ns
4	3700 ns	100 ns	100ns
5	2800ns	100 ns	100ns
Average	2820 Ns	100 ns	380ns

Assumption: While running this experiment, We assumed that both programs were tasted under the same conditions. We made sure no other heavy applications were running in the background that could affect the timing. Also, we only measure the time for the insert and delete functions not printing or user input, So the results reflect only the data structure's performent.