

Link List Report

Challenge 1: Insert at the front

Inserting at the front of a linked list is $O(1)$ because we only change the head pointer to the new node. It's fast and doesn't depend on the list size. In an array, inserting at index 0 takes $O(n)$ because all elements must move, so linked lists are easier for this case.



```
4
5  int main(){
6      LinkedList List;
7
8      // InsertFront
9      List.insertFront(2);
10     List.insertFront(5);
11
12     // InsertEnd
13     // List.insertEnd(10);
14     // List.insertEnd(15);
15     // List.insertEnd(20);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS D:\Documents\Y2 Semester1\C++> cd "d:\Documents\Y2 Semester1\
main } ; if ($?) { .\main }
● Value: 2
Value: 5
5->2->
Head: 5
○ PS D:\Documents\Y2 Semester1\C++\LinkedList>
```

Challenge 2: Insert at the end

Inserting at the end of linked list is $O(n)$ because we must go through the list to find the last node. If we have a tail pointer, it is $O(1)$. Arrays can add at the end quickly if have big size, but linked list need traversal first.

LinkedList >  main.cpp >  main()

```
4
5  int main(){
6      LinkedList List;
7
8      // InsertFront
9      List.insertFront(2);
10     List.insertFront(5);
11
12     // InsertEnd
13     List.insertEnd(10);
14     List.insertEnd(15);
15     List.insertEnd(20);
16     // List.insertAt(12,1);
17
18     List.traverse();
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

```
PS D:\Documents\Y2 Semester1\C++> cd "d:\Documents\Y2 Semester1\C++\LinkedList" & gcc main.cpp -o main } ; if ($?) { .\main }
```

● Value: 2

Value: 5

5->2->

Head: 5

● PS D:\Documents\Y2 Semester1\C++\LinkedList> cd "d:\Documents\Y2 Semester1\C++\LinkedList" & gcc main.cpp -o main } ; if (\$?) { .\main }

Value: 2

Value: 5

5->2->10->15->20->

Head: 5

○ PS D:\Documents\Y2 Semester1\C++\LinkedList>

Challenge 3: Insert at the middle

Inserting in the middle of a linked list is $O(n)$ because we must find the right position first. We change two pointers: the previous node's next and the new node's next. In arrays, inserting in the middle means shifting many elements, which is slower.

```
11
12 // InsertEnd
13 List.insertEnd(10);
14 List.insertEnd(15);
15 List.insertEnd(20);
16
17 // Insert at position
18 List.insertAt(12,1);
19
20 List.traverse();
21 // List.deleteFront();
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
● PS D:\Documents\Y2 Semester1\C++> cd "d:\Documents\Y2
main } ; if ($?) { .\main }
Value: 2
Value: 5
5->12->2->10->15->20->
Head: 5
○ PS D:\Documents\Y2 Semester1\C++\LinkedList>
```

Challenge 4: Delete at the front

Deleting the first node is $O(1)$. We just move the head pointer to the next node. The deleted node's memory should be freed to avoid memory leaks.

```

17      // Insert at position
18      List.insertAt(12,1);
19
20      List.traverse();
21      // Delete both front and end
22      List.deleteFront();
23      // List.deleteEnd();
24      List.traverse();
25
26      // DeleteAt
27      // List.deleteAt(2);

```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

```

PS D:\Documents\Y2 Semster1\C++> cd "d:\Documents\Y2
) { .\main }

```

● Value: 2

Value: 5

5->12->2->10->15->20->

12->2->10->15->20->

Head: 12

○ PS D:\Documents\Y2 Semster1\C++\LinkedList>

Challenge 5: Delete at the end

Deleting the last node is $O(n)$ because we must find the second-to-last node to update its next to NULL. Then we free the last node's memory.


```

20     List.traverse();
21     // Delete both front and end
22     List.deleteFront();
23     List.deleteEnd();
24     List.traverse();
25
26     // DeleteAt
27     // List.deleteAt(2);
28
29     // Swap Two Node

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

PS D:\Documents\Y2 Semster1\C++> cd "d:\Documents\Y2 Se
) { .\main }

```

● Value: 2

Value: 5

5->12->2->10->15->20->

12->2->10->15->20->

Head: 12

● PS D:\Documents\Y2 Semster1\C++\LinkedList> cd "d:\Docu
} ; if (\$?) { .\main }

Value: 2

Value: 5

5->12->2->10->15->20->

12->2->10->15->

Head: 12

○ PS D:\Documents\Y2 Semster1\C++\LinkedList>

Challenge 6: Delete at the middle

Deleting a middle node is $O(n)$ since we must find it first. We change one pointer the previous node's next to skip the deleted node. If we don't free the memory, it causes a memory leak.

```

26 // DELETE
27 List.deleteAt(2);
28 List.traverse();
29
30 // Swap Two Node
31 // List.swapTwoNode(10,20);
32 // List.swapTwoNode(15,12);
33 // cout<<"After Swapped: \n";
34
35 // Search in Linked List
36 // List.search(15);
37 // List.search(6);
38 // List.traverse();
39 cout<<"Head: "<<List.getHead()-

```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

```

PS D:\Documents\Y2 Semster1\C++> cd "d:\Documents\Y2 Semster1\C++\LinkedList" & .\main
) { .\main }
● Value: 2
Value: 5
5->12->2->10->15->20->
5->12->10->15->20->
Head: 5
○ PS D:\Documents\Y2 Semster1\C++\LinkedList>

```

Challenge 7: Traverse the list

To print all elements in a linked list, we visit each node one by one using a loop. The time complexity is $O(n)$. In an array, we can directly access any element using `arr[i]` ($O(1)$), but linked lists must move through each node step by step.

Challenge 8: Swap Two Nodes

Swapping two nodes (it isn't just its values) is $O(n)$ because we need to find both nodes first. It is harder to swap links than values since we must carefully change the pointers of the previous and next nodes. Swapping values is simpler.

```
30      // Swap Two Node
31      List.swapTwoNode(10,20);
32      List.swapTwoNode(15,12);
33      cout<<"After Swapped: \n";
34      List.traverse();
35
36      // Search in Linked List
37      // List.search(15);
38      // List.search(6);
39      // List.traverse();
40      cout<<"Head: "<<List.getHead()->va
41
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

```
● PS D:\Documents\Y2 Semster1\C++> cd "d:\Documents\Y2 Semster
) { .\main }
Value: 2
Value: 5
5->12->2->10->15->20->
After Swapped:
5->15->2->20->12->10->
Head: 5
○ PS D:\Documents\Y2 Semster1\C++\LinkedList>
```

Challenge 9: Search Linked List

Searching in a linked list is like a linear search in arrays — we check each element one by one. The time complexity is $O(n)$. Arrays are faster for random access because we can use an index, while linked lists must move node by node.

```

32 // List.swapTwoNode(15,12);
33 // cout<<"After Swapped: \n";
34 // List.traverse();
35
36 // Search in Linked List
37 List.search(15);
38 List.search(6);
39 cout<<"Head: "<<List.getHead()->value<<endl;
40
41 return 0;
42 }

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

PS D:\Documents\Y2 Semster1\C++> cd "d:\Documents\Y2 Semster1\C++\LinkedList"
) { .\main }
● Value: 2
Value: 5
5->12->2->10->15->20->
Value 15 found at position 4
Value 6 not found in the list.
Head: 5
○ PS D:\Documents\Y2 Semster1\C++\LinkedList>

```

Challenge 10: Compare with arrays

A linked list is better when doing many insertions or deletions, especially at the front, because it doesn't need to shift elements. Arrays are better for fast random access or when the data size is fixed.

Array vs Linked List

Operation	Array	Linked List
Insert at front	$O(n)$	$O(1)$
Insert at end	$O(1)$	$O(n)/O(1)$
Insert at middle	$O(n)$	$O(n)$
Delete at front	$O(n)$	$O(1)$
Delete at end	$O(1)$	$O(n)$
Delete at middle	$O(n)$	$O(n)$
Access by index	$O(1)$	$O(n)$

Reflection Prompts

1. Operations were $O(1)$ in linked lists but $O(n)$ in arrays are inserting or deleting at the front.
2. Operation is faster in array is accessing elements by index.

3. Because each node uses dynamic memory; if not freed, it causes memory leaks.
4. It's the first node the start of the list.
5. If we lose the head pointer, we lose access to the whole list and its data.

Scenario Analysis: Choose array or linked list

Nº	Scenario	Choice	Reason
1	Real-time scoreboard	Linked List	Easier to insert at end and delete from front quickly.
2	Undo/Redo feature	Linked List	Fast add/remove at the front.
3	Music playlist	Linked List	Flexible insertions and deletions anywhere.
4	Large dataset search	Array	Direct access by index is faster.
5	Simulation of queue at bank	Linked List	Easily add and remove nodes at both ends
6	Inventory system	Array	Know index so it fast to lookup is $O(1)$
7	Polynomial addition program	Linked List	Handles frequent insertions and deletions easily.
8	Student roll-call system	Array	Fixed order and quick access by index.