

## Lab Practice: Linked List

### Challenge 1 — Insert at the Front

**Q: Insert a new node at the start of a linked list. What is the complexity?**

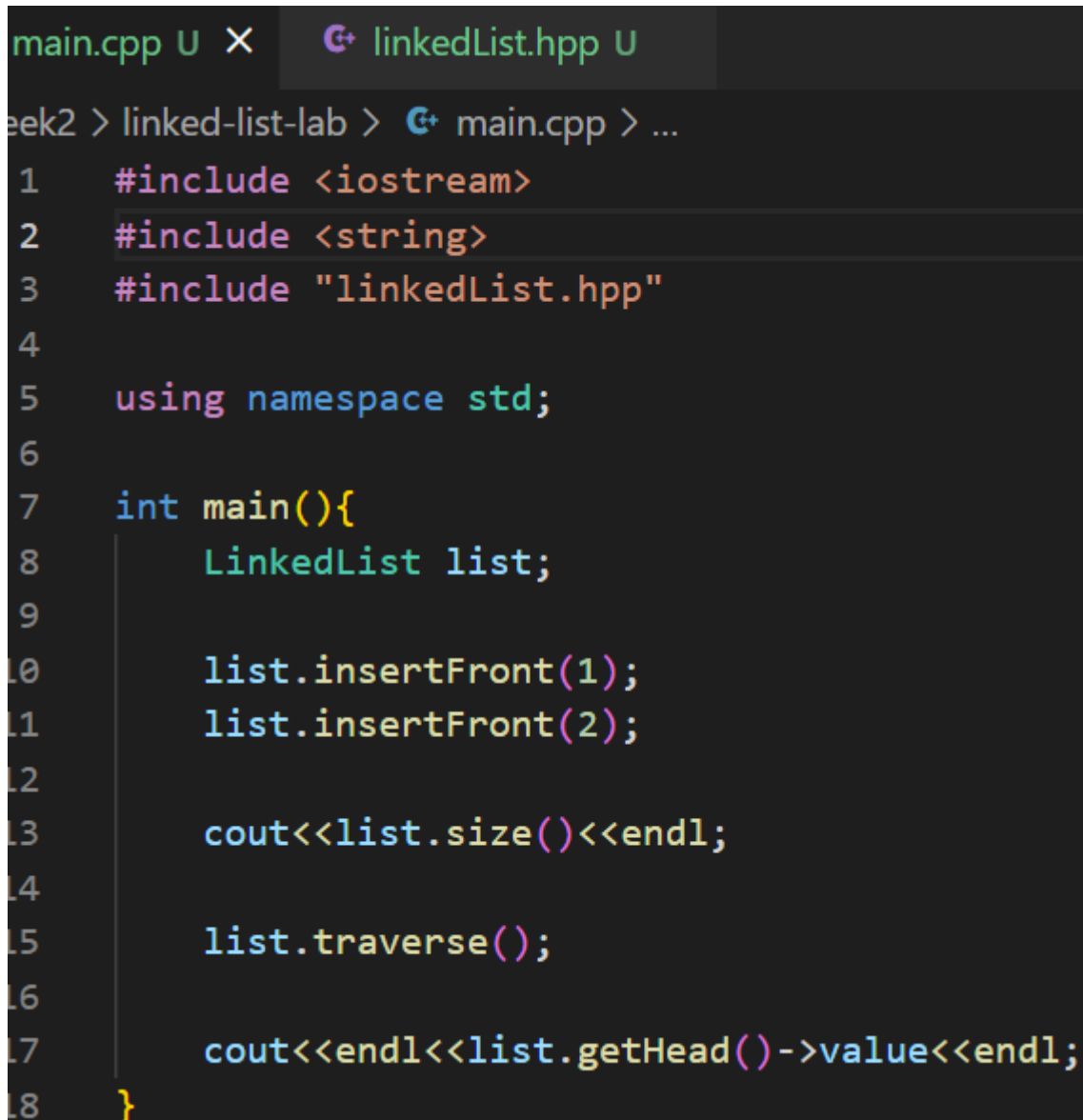
A: The complexity of inserting a new node at the start of a linked list is  $O(1)$ . Because we don't have to do shifting.

**Discuss: How is this easier compared to inserting at index 0 in an array?**

- ➔ It is easier compared to inserting at index 0 in an array because Linked list we just need to point `newNode->next` to head and change head to point to `newNode->next`. Which is faster [ $O(1)$ ] compared to shifting elements one by one,  $n$  times [ $O(n)$ ] to make one space in front of arr.

Code and Explanation:

main.cpp



```
main.cpp U X  G+ linkedList.hpp U
week2 > linked-list-lab > G+ main.cpp > ...
1  #include <iostream>
2  #include <string>
3  #include "linkedList.hpp"
4
5  using namespace std;
6
7  int main(){
8      LinkedList list;
9
10     list.insertFront(1);
11     list.insertFront(2);
12
13     cout<<list.size()<<endl;
14
15     list.traverse();
16
17     cout<<endl<<list.getHead()->value<<endl;
18 }
```

- ➔ In main func, we insert 2 elements to the Linked List (value: 1 and 2).

linkedList.hpp

```

Week2 > linked-list-lab > linkedList.hpp > LinkedList > getHead()
5
6  struct Node{
7      int value;
8      Node* next;
9  };
10
11 class LinkedList{
12     //properties = var
13     private:
14         Node *head, *cur;
15         int n;
16
17     //constructor = special method
18     public:
19     LinkedList(){
20         head = nullptr;
21         cur = nullptr;
22         n = 0;
23     }
24
25     //methods = func
26     int size(){return n;};
27
28     void traverse(){
29         if (n == 0) {
30             cout<<"(empty!)"<<endl;
31             return;
32         }
33         cur = head;
34         while(cur != nullptr){
35             cout<<cur->value<<"->";
36             cur = cur->next;
37         }
38     }
39
40     Node* getHead(){return head;};
41
42     void insertFront(int val){
43         Node* newNode = new Node{val, nullptr};
44         newNode->next = head;
45         head = newNode;
46         n++;
47     }
48 };

```

The **insertFront(int val)** method adds a new node to the very beginning of the linked list. It starts by **creating a new node** using the input value `val` and setting its next pointer to nothing yet. The core action is then performing two pointer updates: first, the **new node's next pointer** is made to point to the current head (which links the new node to the rest of the existing list); second, the list's main **head pointer** is updated to point to the newly created node, making it the new start of the list. Finally, the total node count, `n`, is increased by one.

## Challenge 2 — Insert at the End

**Q: Append a new node to the end of a linked list. What is the complexity?**

A: Insert at the end is to append a new node to the end of a singly linked list, requires traversing the entire list from the head to find the very last node. The complexity of this operation is  $O(n)$  (linear time), where  $n$  is the number of nodes in the list.

**Discuss: Do we need to traverse the entire list? How does this differ from arrays?**

- Yes we need to in order for us to append to new node to the end of the linked list because linked list doesn't have index like array. It is different from array because array stores elements contiguously, and the size/capacity metadata tells it exactly where to insert. So, array can do it immediately at  $O(1)$ .

Code and Explanation:

main.cpp

```
int main(){
    LinkedList list;

    // list.insertFront(1);
    // list.insertFront(2);

    list.insertEnd(10);
    list.insertEnd(15);
    list.insertEnd(30);
    list.insertEnd(100);

    cout<<"Size n = "<<list.size()<<endl;

    list.traverse();

    cout<<endl<<"Head Value: "<<list.getHead()->value<<endl;
}
```

- In my main func, I inserted 4 elements one by one at the end.

linkedList.hpp

```

void insertEnd(int val){
    if(n==0){
        insertFront(val);
        return;
    }
    cur = head;
    for(int i=0;i<n-1;i++){
        cur = cur->next;
    }
    // while(cur->next != nullptr){
    //     cur = cur->next;
    // }
    Node* newNode = new Node{val, nullptr};
    cur->next = newNode;
    n++;
}

```

In my insertEnd func, I check first, if n is 0 I just insert at the front. If not, I will use for loop to travel to the end of the list and insert the element there. Plus, increment the number of the nodes.

### Challenge 3 — Insert in the Middle

**Q:** Insert a node between two existing nodes.

**A:** Inserting a node in the middle of a singly linked list requires changing two pointers.

**Discuss:** Which two arrows (pointers) need to be changed? Compare to shifting in arrays.

- ➔ When inserting a new node (newNode) between two existing nodes (preNode & nextNode). That two pointers must be updated to :
  - newNode->next = nextNode
  - preNode->next = newNode
- ➔ If compare to shifting array, the difference is that, linked lists perform insertion by only rewiring pointers, which is a fast operation[ $O(1)$ ] once we've known the spot. However, arrays must move every elements after the insertion point to make space, resulting in a much slower operation[ $O(n)$ ].

## THANG SAOLY G3 - Lab Practice: Linked List

Code and Explanation:

main.cpp

```
int main(){
    LinkedList list;

    // list.insertFront(1);
    // list.insertFront(2);

    list.insertEnd(10);
    list.insertEnd(15);
    list.insertEnd(14);

    list.insertAt(12,1);

    cout<<"Size n = "<<list.size()<<endl;

    list.traverse();

    cout<<endl<<"Head Value: "<<list.getHead()->value<<endl;
}
```

→ In my main func, I insert 12 at position 1 (between 10 and 15).

linkedList.hpp

```
void insertAt(int val, int pos){ // Between two Nodes
    if (pos > n){
        cout << "Out of node range!\n";
        return;
    }
    if (pos == 0){
        insertFront(val);
        return;
    }
    if (pos == n){
        insertEnd(val);
        return;
    }
    cur = head;
    for (int i = 0; i < pos - 1; i++){
        cur = cur->next;
    }
    Node *newNode = new Node{val, nullptr};
    newNode->next = cur->next;
    cur->next = newNode;
    n++;
}
```

The *insertAt* method puts a new node with the given *val* at a specific position (*pos*). It first checks for edge cases to simplify the process: if the position is out of the list's range, it stops. If *pos* is **0**, it uses *insertFront*. If *pos* equals the total node count (*n*), it uses *insertEnd*. If the position is in the middle, it starts a *for* loop to move the pointer *cur* until it stops at the node just before the insertion point (position *pos - 1*). A new node is created. Then, two simple pointer changes occur: *newNode->next* is linked to *cur->next*, and *cur->next* is linked to the new node, forming it into the list. Finally, the node count (*n*) is increased by one.

## Challenge 4 — Delete from the Front

**Q: Remove the first node.**

A: We create a temporary node to store the front node (tmp). We change the head pointer to next element and they just delete the temp node.

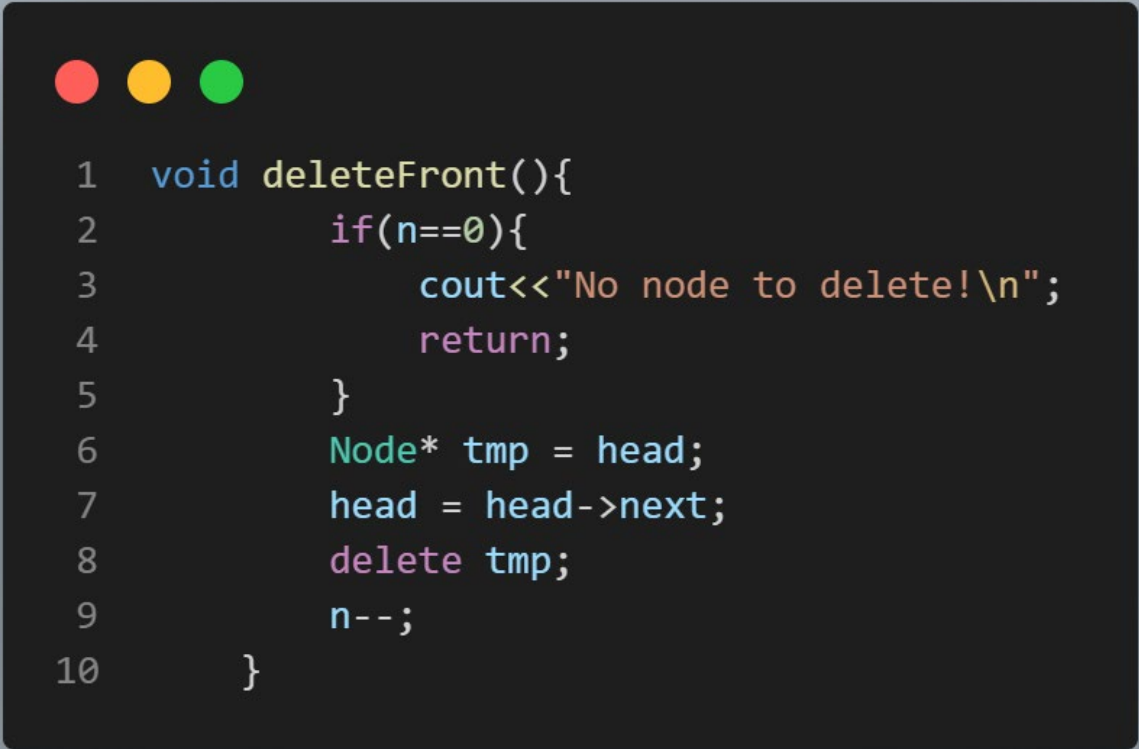
**Discuss: What happens to the head pointer? What about the deleted node's memory?**

- Head pointer moves from the deleted node to the next node. The deleted node that store in tmp now get deleted and free up space.

In **main.cpp**, we just need one line of this code to perform deletion operation.

```
list.deleteFront();
```

In **linkedList.hpp**,



```
1 void deleteFront(){
2     if(n==0){
3         cout<<"No node to delete!\n";
4         return;
5     }
6     Node* tmp = head;
7     head = head->next;
8     delete tmp;
9     n--;
10 }
```

- If n is 0, nothing to delete. But if n is greater or equal to 1 we create tmp Node to store head and point head to head->next , delete the tmp node and decrease the number of node (n--);

## Challenge 5 — Delete from the End

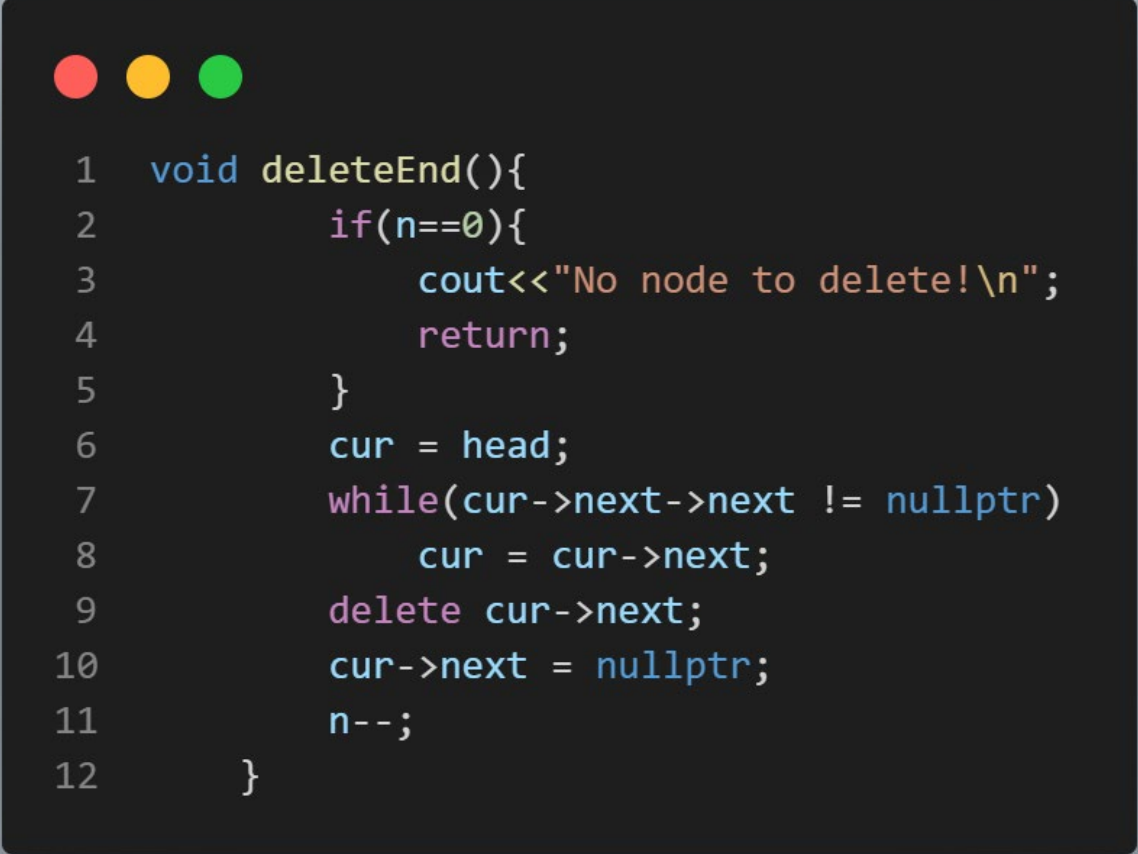
**Q: Remove the last node.**

**Discuss: How do we find the node before the last one?**

- ➔ To delete the last node in a singly linked list, you must find the **second-to-last node**, find it by traversing the list with a pointer (cur) until the loop condition `cur->next->next != nullptr` is met. It's an  **$O(n)$**  complexity for `deleteEnd()`.

Code & Explanation:

In main func, we just call `list.deleteEnd()`. In hpp file, there is a `deleteEnd` func that it work by traversal to the second-to-last node and delete the node after it.



```
1 void deleteEnd(){
2     if(n==0){
3         cout<<"No node to delete!\n";
4         return;
5     }
6     cur = head;
7     while(cur->next->next != nullptr)
8         cur = cur->next;
9     delete cur->next;
10    cur->next = nullptr;
11    n--;
12 }
```

## Challenge 6 — Delete from the Middle

**Q: Remove a node between two others.**

**Discuss: Which arrow changes? What happens if we forget to free memory?**

- The single arrow that changes belongs to the previous node's next pointer (`cur->next`). It is rewired to point to the node after the one being deleted (`cur->next = cur->next->next`).

## THANG SAOLY G3 - Lab Practice: Linked List

- If you forget to free the memory of the deleted node (by calling delete tmp after saving a temporary pointer to it), it leads to a memory leak. The memory remains allocated but is no longer reachable, which can slow down or crash a program over time.

Code & Explanation:

In main.cpp we need this one line: `list.deleteAt(1);`

```
1 void deleteAt(int pos){// Between two Nodes
2     if (pos > n-1){
3         cout << "Out of node range!\n";
4         return;
5     }
6     if (pos == 0){
7         deleteFront();
8         return;
9     }
10    if (pos == n-1){
11        deleteEnd();
12        return;
13    }
14    cur = head;
15    for (int i = 0; i < pos -1; i++){
16        cur= cur->next;
17    }
18    Node* tmp = cur->next;
19    cur->next = cur->next->next;
20    delete tmp;
21 }
```

In linkedList.cpp we have deleteAt() that do just like insertAt() by first **checks for edge cases** to simplify the process: if the position is out of the list's range (i.e., pos > n-1), it stops and prints an error. If **pos is 0**, it uses deleteFront(). If **pos equals the last valid index (n-1)**, it uses deleteEnd(). If the position is in the middle, it starts a **for loop** to move the pointer **cur** until it stops at the node **just before the node to be deleted** (position pos - 1). The node to be removed is temporarily stored in a pointer, **tmp** (i.e., tmp = cur->next). Then, a single pointer change occurs: cur->next is linked to the node **after tmp** (i.e., cur->next = cur->next->next). The memory of the deleted node is **freed** using delete tmp. Finally, the node count (**n**) is **decreased by one**.

## Challenge 7 — Traverse the List

**Q:** Print all elements in the linked list.

**Discuss:** How does traversal differ from direct arr[i] access?

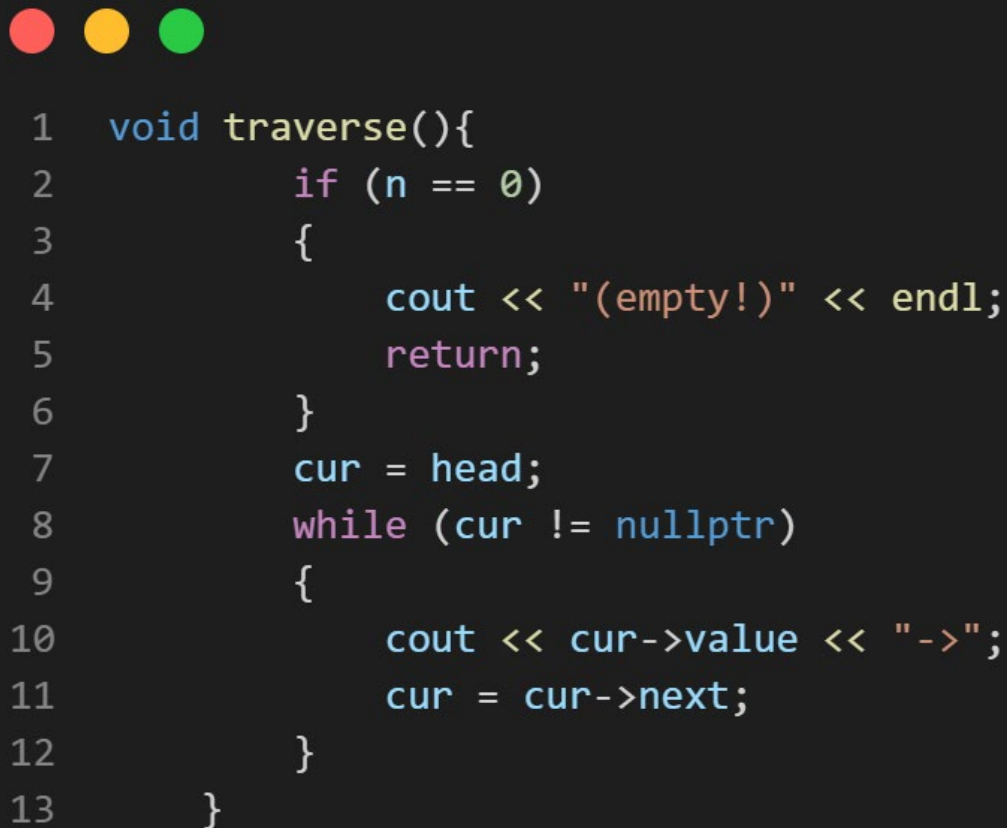
- ➔ Traversal in a Linked List requires starting at the head and sequentially following the next pointer from node to node until a nullptr is reached. The complexity is O(n) because you must visit every node.
- ➔ Direct Array Access (arr[i]) allows you to jump immediately to the memory location of the -th element using a simple calculation (base address + \* element size). This is a random access operation with a complexity of O(1).

Code & Explanation:

In main.cpp we need this, `list.traverse();`

In header file, we set cur = head and put it in the while loop to make it go through the linked list.





```
1 void traverse(){
2     if (n == 0)
3     {
4         cout << "(empty!)" << endl;
5         return;
6     }
7     cur = head;
8     while (cur != nullptr)
9     {
10        cout << cur->value << "->";
11        cur = cur->next;
12    }
13 }
```

## Challenge 8 — Swap Two Nodes

**Q: Swap two nodes in the list (not just their values).**

**Discuss: Is it easier to swap values or swap links? Why?**

- It is generally **much easier to swap values**. Swapping values is a simple operation once the nodes are found.
- **Swapping links** (the nodes themselves) is far more complex because it requires re-wiring up to four different pointers: the next pointers of the two nodes *before* the ones being swapped, and the next pointers of the two nodes being swapped.

Code & Explanation:

In main.cpp, This main function creates a linked list named list and inserts four integer values (1, 2, 3, 4) at the end. It then prints the size of the list, traverses and prints the elements, swaps the nodes at positions 1 and 2, traverses and prints the elements again, and finally prints the value of the new head node.

```
1  int main(){
2      LinkedList list;
3
4      list.insertEnd(1);
5      list.insertEnd(2);
6      list.insertEnd(3);
7      list.insertEnd(4);
8
9      cout<<"Size n = "<<list.size()<<endl;
10
11     list.traverse();
12     list.swap2Nodes(1,2);
13     cout<<endl;
14     list.traverse();
15
16     cout<<endl<<"Head Value: "<<list.getHead()->value<<endl;
17 }
```

In Headerfile, The swap2Nodes function swaps two nodes in the linked list based on their positions, pA and pB, by changing their **pointers** instead of just their values. It first checks for errors like having too few nodes or invalid positions. To perform the swap, it traverses the list to find the two nodes and the nodes directly preceding them. Finally, it **re-wires the pointers** of the previous nodes to point to the swapped nodes and updates the next pointers of the swapped nodes themselves, correctly linking them back into the list.

## THANG SAOLY G3 - Lab Practice: Linked List

```
1 void swap2Nodes(int pA, int pB) {
2     // check the number of nodes must have at least 2
3     if (n < 2) {
4         cout << "Need at least 2 nodes to swap (currently have " << n << ")\n";
5         return;
6     }
7
8     // check range of the valid position
9     if (pA < 0 || pB < 0 || pA >= n || pB >= n) {
10        cout << "Error: Position out of range [0, " << n-1 << "]\n";
11        return;
12    }
13
14    //the same position, no need to swap.
15    if (pA == pB) return;
16
17    //guarantee that pA is always less than pB for simplicity
18    if (pA > pB) {
19        swap(pA, pB);
20    }
21
22    // Find four required pointers (A, B, and their previous)
23    Node* pA_prev = nullptr;
24    Node* pB_prev = nullptr;
25    Node* currA = head;
26    Node* currB = head;
27
28    Node* temp_curr = head;
29    Node* temp_prev = nullptr;
30    int count = 0;
31
32    while (temp_curr != nullptr) {
33        if (count == pA) {
34            pA_prev = temp_prev;
35            currA = temp_curr;
36        }
37        if (count == pB) {
38            pB_prev = temp_prev;
39            currB = temp_curr;
40            break;
41        }
42
43        temp_prev = temp_curr;
44        temp_curr = temp_curr->next;
45        count++;
46    }
47
48    // Re-wire pointers
49
50    // Connect node before A to B
51    if (pA_prev != nullptr) {
52        pA_prev->next = currB;
53    } else {
54        head = currB;
55    }
56
57    // Connect node before B to A
58    if (pB_prev != nullptr) {
59        pB_prev->next = currA;
60    } else {
61        head = currA;
62    }
63
64    // Swap 'next' pointers of nodes themselves (currA and currB)
65    Node* tempNext = currB->next;
66    currB->next = currA->next;
67    currA->next = tempNext;
68 }
```

## Challenge 9 — Search in Linked List

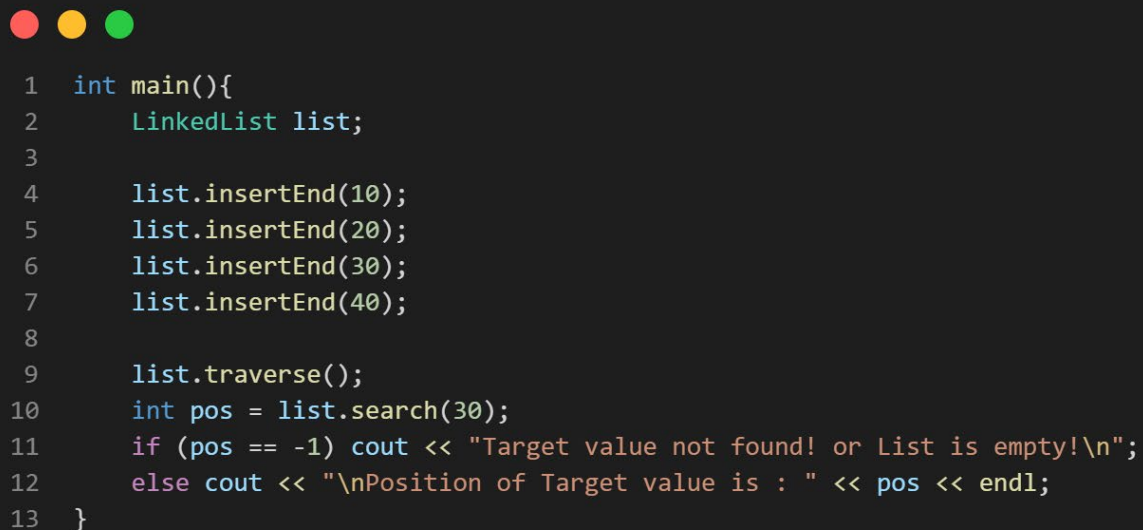
**Q: Search for a value in a linked list.**

**Discuss: How is this similar to linear search in arrays? Which one is faster for random access?**

- **Similarity to Linear Search:** Searching in a linked list is the same as a linear search in an array. Both start at the beginning and check each element one by one until the target is found or the end is reached. The complexity for both is  $O(n)$  in the worst case.
- **Random Access:** Arrays are much faster for random access (). Linked lists cannot perform random access and always require a linear traversal () to reach the  $n$ -th element.

Code & Explanation:

In main.cpp, we give the parameter of the target value.



```
1  int main(){
2      LinkedList list;
3
4      list.insertEnd(10);
5      list.insertEnd(20);
6      list.insertEnd(30);
7      list.insertEnd(40);
8
9      list.traverse();
10     int pos = list.search(30);
11     if (pos == -1) cout << "Target value not found! or List is empty!\n";
12     else cout << "\nPosition of Target value is : " << pos << endl;
13 }
```

In header file, The search function looks for a specific `target_val` in the linked list. It first checks if the list is empty. If not empty, it starts a pointer (current) at the head and traverses the list, checking the value of each node against the target and keeping track of the current position. If the target value is found, the function immediately returns its position; otherwise, it reaches the end and returns -1, indicating the value wasn't found.

```

1  int search(int target_val) {
2      if (head == nullptr) {
3          cout<<"List is empty!\n";
4          return -1;
5      }
6
7      Node* current = head;
8      int position = 0;
9
10     // Traverse the list until the end (nullptr)
11     while (current != nullptr) {
12         if (current->value == target_val) {
13             return position; // Value found
14         }
15
16         // Move to next node, increment position counter
17         current = current->next;
18         position++;
19     }
20     cout<<"Target value not found!\n";
21     return -1;
22 }

```

### Challenge 10 — Compare with Arrays

**Q:** For each operation (insert, delete, access), write the complexity for arrays vs. linked lists.

Operation	Linked List Complexity		Array Complexity	
Insert at Front		O(1)		O(n) (requires shifting)
Insert at End		O(n) (requires traversal)		O(1) (if capacity allows)
Insert in Middle	O(n) (to find position)	O(n) (to find position and shift)		
Delete from Front		O(1)	O(n) (requires shifting)	
Delete from End	O(n) (requires traversal)	O(1) (easy to reduce size)		
Access/Search by Index (arr[i])	O(n) (requires traversal)		O(1) (random access)	

**Discuss:** In what situations is a linked list clearly better?

- A linked list is clearly better when:
  1. **Frequent Insertions/Deletions at the Front:** Operations at the head of a linked list are , which is significantly faster than the shifting required for arrays.

2. **Memory Efficiency:** When you need a highly dynamic structure that can grow and shrink without pre-allocating a large, contiguous block of memory.
3. **Arbitrary Insertions/Deletions (Middle):** While you still have to search for the spot (), the actual insertion/deletion is a fast pointer change, unlike the shifting required by arrays.
- ### Reflection Prompts

1. **Which operations were in linked lists but in arrays?**

○ **Insertion at the Front** (insertFront).

○ **Deletion from the Front** (deleteFront).

○ *The actual insertion/deletion (rewiring pointers) in the middle is , but finding the location first makes the overall operation .*

2. **Which operation is clearly faster in arrays than in linked lists?**

○ **Accessing an element by index ()** (Random Access). This is in arrays but in linked lists.

3. **Why must we manage memory carefully in linked lists?**

○ Nodes in a linked list are created dynamically using new (or malloc). This memory is allocated on the heap and **must be manually freed** using delete (or free) when a node is removed (like in deleteFront or deleteEnd). Failure to do so results in **memory leaks**.

4. **What does the head pointer represent?**

○ The **head pointer** is a crucial class member variable that stores the memory address of the **very first node** in the linked list. It serves as the single entry point to the entire list.

5. **What happens if we lose the head pointer?**

• If you lose the head pointer, you lose all access to the memory allocated for the rest of the list. The entire list becomes inaccessible, resulting in a **memory leak** because that memory cannot be used or freed.
- Scenario Analysis: Choose Array or Linked List
- | Scenario  | Choice             | Justification  |
|---|--------------------|--|
| 1. Real-time scoreboard (Add at end, remove from front)         | <b>Linked List</b> | Optimized for frequent <b>O(1) deletions from the front</b> (deleteFront) and <b>O(n) additions to the end</b> (since array shifting is slow). |
| 2. Undo/Redo feature (Frequent additions/removals at the front) | <b>Linked List</b> | This is the classic use case for the <b>insertion and deletion at the front</b> that linked lists excel at.                                    |

### THANG SAOLY G3 - Lab Practice: Linked List

3. Music playlist (Add and remove songs anywhere)	<b>Linked List</b>	While finding a song is in both, once the position is found, the <b>insertion/deletion is an pointer change</b> for a linked list, avoiding the shifting () an array would require.
4. Large dataset search (Random access by index needed often)	<b>Array</b>	Arrays are essential when <b>random access by index is frequent</b> due to their lookup time. Linked lists would be too slow ().
5. Simulation of a queue (Join at end, leave at front)	<b>Linked List</b>	This structure (a queue) is best implemented with a linked list, as adding and removing from the ends are its most efficient operations ( for front deletion/insertion, or for both if a tail pointer is used).
6. Inventory system (Always know the item's index, need quick lookups)	<b>Array</b>	Knowing the index and needing a quick lookup means <b>random access</b> is the priority, which is in an array.
7. Polynomial addition program (Terms inserted and deleted dynamically)	<b>Linked List</b>	Since the number of terms is highly variable and operations involve frequent dynamic insertions and deletions, a linked list is preferred for its memory flexibility and ease of <i>rewiring</i> links once the term's location is known.
8. Student roll-call system (Fixed order, access by index is frequent)	<b>Array</b>	The fixed order and need for frequent <b>access by index</b> (like ) makes the random access of an array the clear choice.