哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

数据结构
Data Structures

Chapter 5 Linear List

Prof. Yitian Shao
School of Computer Science and Technology

Linear List

Course Overview

- Introduction of (Singly-) Linked List
- Create a Linked List
- Iterate through a Linked List
- Modify Linked Lists:
  - Implementing **add()** and **delete()** from a Linked List
- Common Mistakes and Tips
- Doubly-Linked Lists

Linear Data Structures

- Linear Data Structures are a type of data structure in computer science where **data elements are arranged sequentially**, one after the other. Each element has a unique predecessor (except for the first element) and a unique successor (except for the last element)
- **Arrays**: A collection of elements stored in contiguous memory locations.
- **Linked Lists**: A collection of nodes, each containing an element and a reference to the next node.
- **Stacks**: A collection of elements with Last-In-First-Out (LIFO) order.
- **Queues**: A collection of elements with First-In-First-Out (FIFO) order

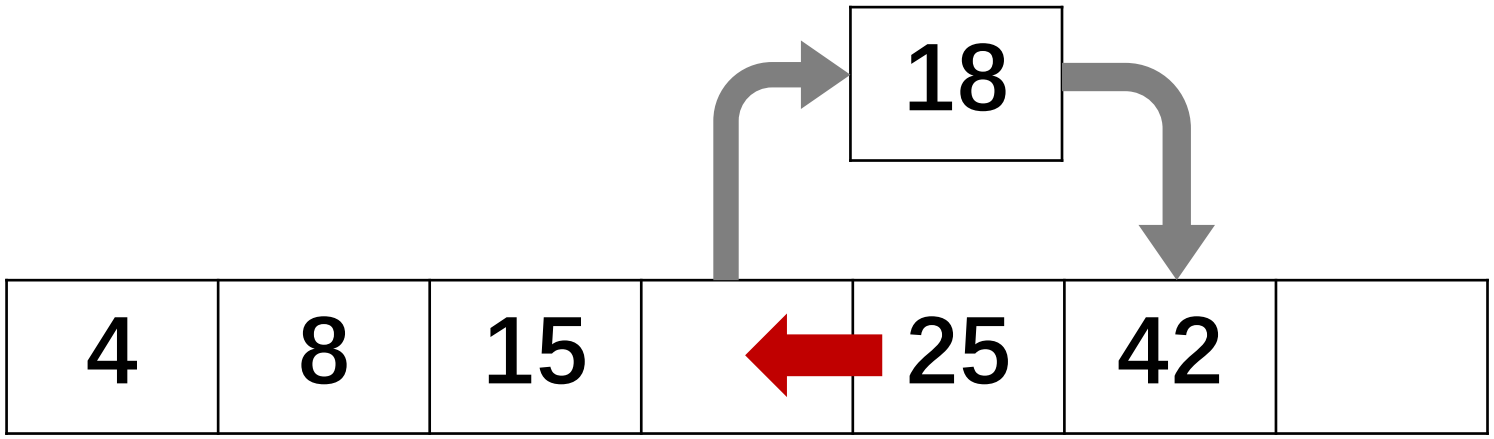
Flaws with Arrays

- Some adds are very costly (when we have to resize)– Adding just one element requires copying all the elements
- Imagine if everything were like that?
  - Instead of just grabbing a new sheet of paper, re-copy all notes to a bigger sheet when you run out of space
- Idea: what if we could just add the amount of memory we need?

4	8	15	18	25	42	

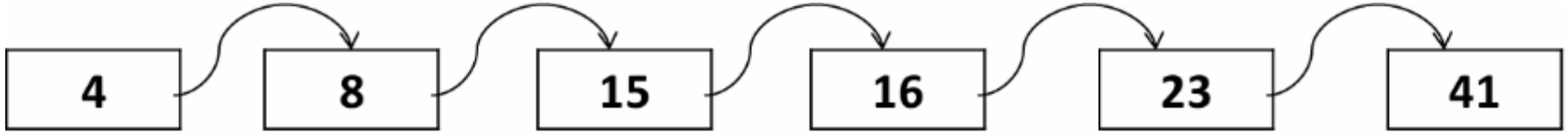
Vectors and Arrays

- Inserting into an array involves shifting all the elements over
  - That's O(N)
- What if we were able to easily insert?



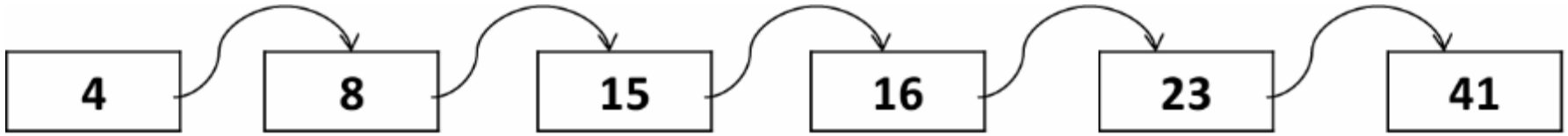
Linked List

- Main idea: Store every element in its own block of memory
- Then we can just add one block of memory!
- Then we can efficiently insert into the middle (or front)!
- A Linked List is good for storing elements in an order (similar to Vector)
- Elements are chained together in a sequence
- Each element is allocated on the heap



Parts of a Linked List

- What does each part of a Linked List need to store?
  - element
  - pointer to the next element
  - the last node points to a **nullptr**

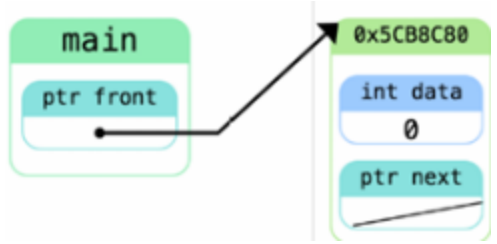


The ListNode struct ()

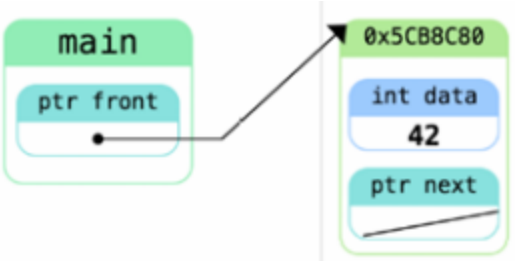
```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;             //value of the element data
 *     ListNode* next;      //pointer to the next element
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
```

Creating a Linked List

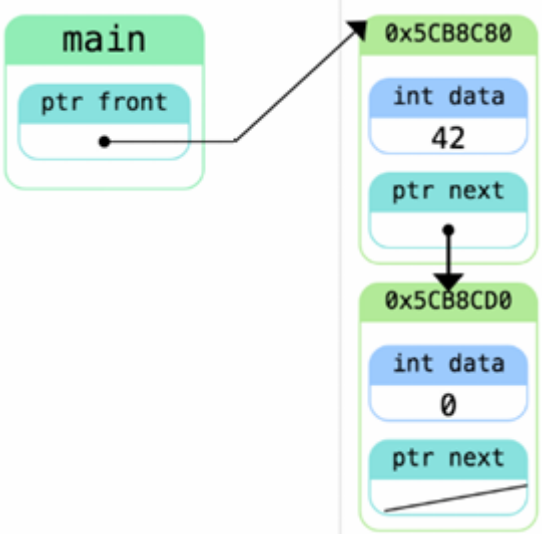
```
#(inside the main function)
ListNode* front = new ListNode();
```




```
front->val = 42; // value (int data)
```

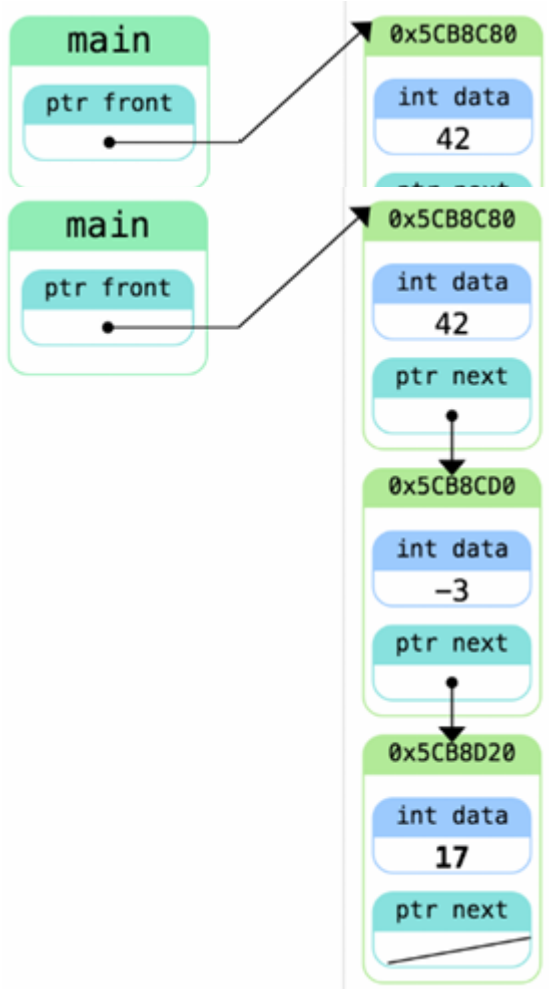



```
front->next = new ListNode();
```



Creating a Linked List (Continue)

```
ListNode* front = new ListNode();
front->val = 42;
front->next = new ListNode();
front->next->val = -3;
front->next->next = new ListNode();
front->next->next->val = 17
front->next->next->next = nullptr;
```



Linked List Iteration

- Idea: travel each `ListNode` one at a time
- No easy way to "index in" like with Vector. Why?
- General syntax:

```
for (ListNode* ptr = list; ptr != nullptr; ptr = ptr->next) {
      /* … use ptr … */
}
```

Linked List Iteration

Initialize ptr to the first node in (front node of) the linked list

```
for (ListNode* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```

Linked List Iteration

Updated in each loop: Move ptr to point to the next node of the list

```
for (ListNode* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```

Linked List Iteration

Continue doing this until we hit the end of the list

```
for (ListNode* ptr = list; ptr != nullptr; ptr = ptr->next) {
    /* … use ptr … */
}
```

Linked List Iteration Exercise

- Write a function that takes in the pointer to the front of a Linked List and **prints out** all the elements of a Linked List

```
void printList(ListNode *front) {
    // Complete the code below


}
```

Linked List Iteration Exercise

- Write a function that takes in the pointer to the front of a Linked List and **prints out** all the elements of a Linked List

```
void printList(ListNode *front) {
for (ListNode* ptr = front; ptr != nullptr; ptr = ptr->next)
{
      cout << ptr->val << endl;
}
}
```

Alternative Iteration using While Loop

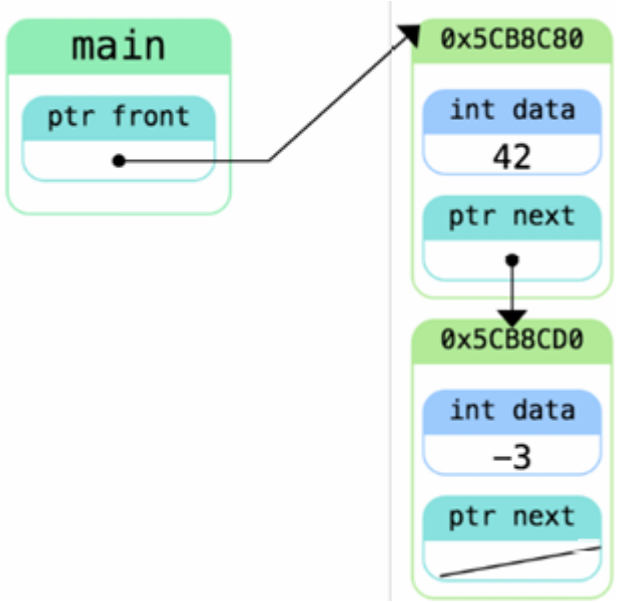
```
for (ListNode* ptr = front; ptr != nullptr; ptr = ptr->next)
{
        // do something with ptr
}
```

is equivalent to

```
ListNode *ptr = front;
while (ptr != nullptr) // or while (ptr)
{
        // do something with ptr
        ptr = ptr->next;
}
```

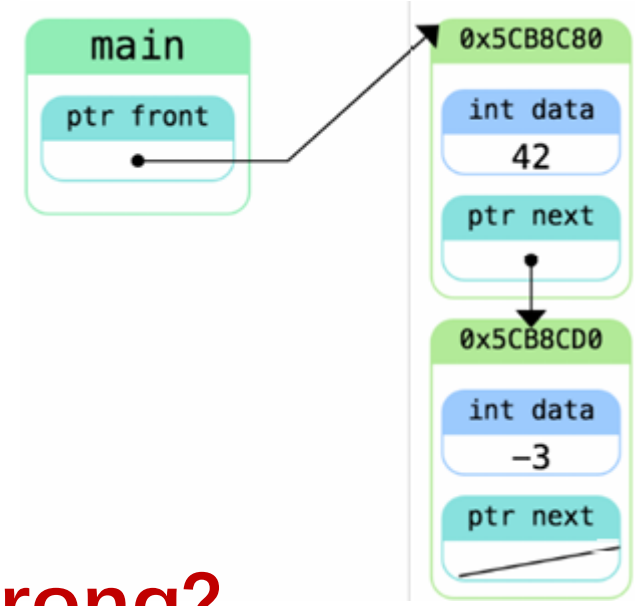
Questionable Solution

```
int main() {
    ListNode* front = new ListNode();
    front->val = 42;
    front->next = new ListNode();
    front->next->val = -3;
    front->next->next = nullptr;
    while (front != nullptr) {
      cout << front->val << " ";
      front = front->next;
    }
  return 0;
}
```



Questionable Solution

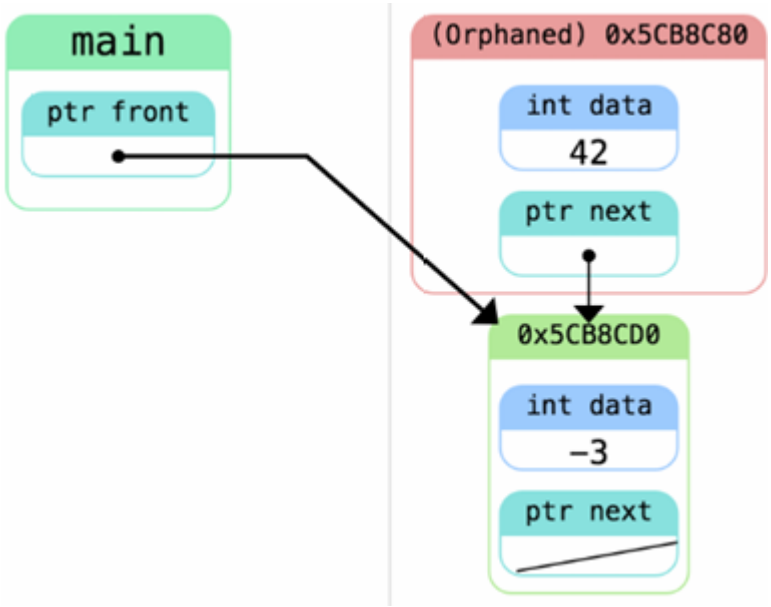
```
int main() {
    ListNode* front = new ListNode();
    front->val = 42;
    front->next = new ListNode();
    front->next->val = -3;
    front->next->next = nullptr;
    while (front != nullptr) {
      cout << front->val << " ";
      front = front->next;
    }
  return 0;
}
```



What's wrong?

Questionable Solution

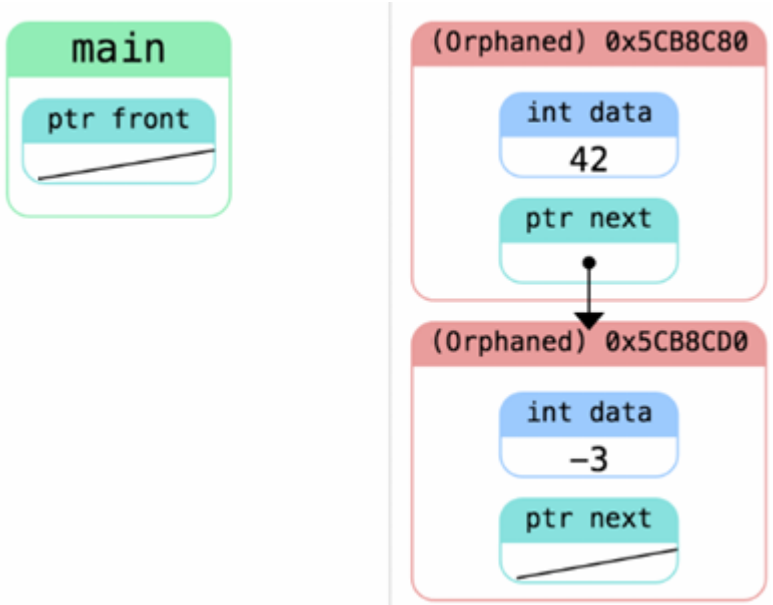
```
int main() {
    ListNode* front = new ListNode();
    front->val = 42;
    front->next = new ListNode();
    front->next->val = -3;
    front->next->next = nullptr;
    while (front != nullptr) {
      cout << front->val << " ";
      front = front->next;
    }
  return 0;
}
```



What's wrong?

Questionable Solution

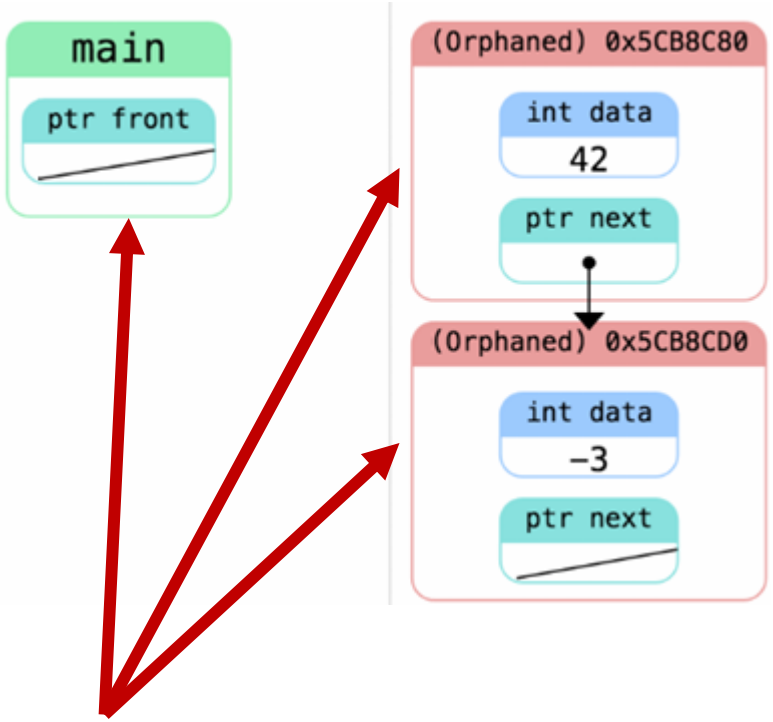
```
int main() {
    ListNode* front = new ListNode();
    front->val = 42;
    front->next = new ListNode();
    front->next->val = -3;
    front->next->next = nullptr;
    while (front != nullptr) {
      cout << front->val << " ";
      front = front->next;
    }
  return 0;
}
```



What's wrong?

Questionable Solution

```
int main() {
    ListNode* front = new ListNode();
    front->val = 42;
    front->next = new ListNode();
    front->next->val = -3;
    front->next->next = nullptr;
    while (front != nullptr) {
      cout << front->val << " ";
      front = front->next;
    }
  return 0;
}
```

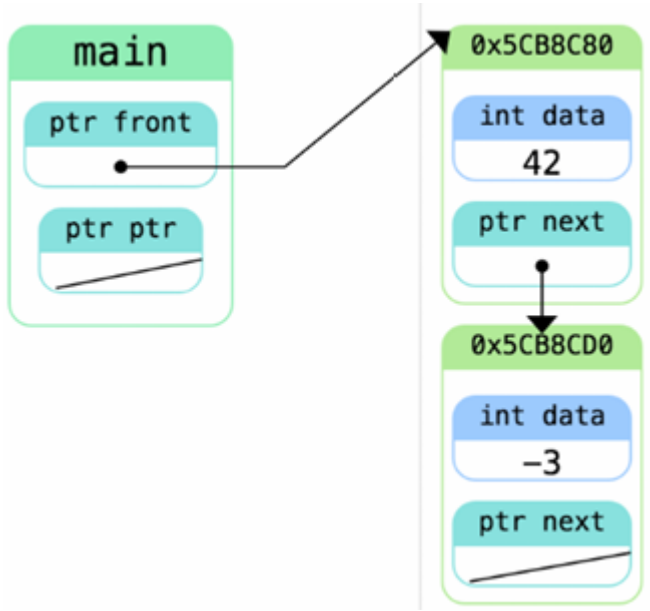


Orphaned memory!

How to correct it?

Correct Version

```
int main() {
    ListNode* front = new ListNode();
    front->val = 42;
    front->next = new ListNode();
    front->next->val = -3;
    front->next->next = nullptr;
    ListNode *ptr = front;
    while (ptr != nullptr) {
      cout << ptr->val << " ";
       ptr = ptr->next;
    }
  return 0;
}
```



There is still a pointer to the front node of the linked list

Homework

- Complete **exercise 5.1**, **5.2**, **5.3** (at the end of the lecture slides)

Linked Lists

- So far, we learned how to create and traverse a linked list from front to end
- **How can we add to the front of a Linked List?**
- **Should the front be passed by reference or by value?**

Add to Front

- When **modifying** the list, pass the front ptr by **reference**
- When simply iterating through the list, the front ptr can be passed by value (to avoid producing orphaned memory)

```
void addToFront(int elem, ListNode* &front) {
    ListNode* newNode = new ListNode(elem, front);
    front = newNode;
}
```

Add to Front

	Pointer	Reference
Declare	int* myPtr1 Float* myPtr2 char* myPtr3	void myFunction(int& myVariable){} void swap_vals(float& val1, float& val2){}

Access	*myPtr1
	*myPtr2
	*myPtr3
		int x; float y;
		myPtr1 = &x;
		myPtr2 = &y;

A ListNode-type pointer that is passing by reference to the function

```
void addToFront(int elem, ListNode*& front) {
    ListNode* newNode = new ListNode(elem, front);
    front = newNode;
}
```

Add to Back

- How would we add to the back of a Linked List?
- Should the front be passed by reference or by value?

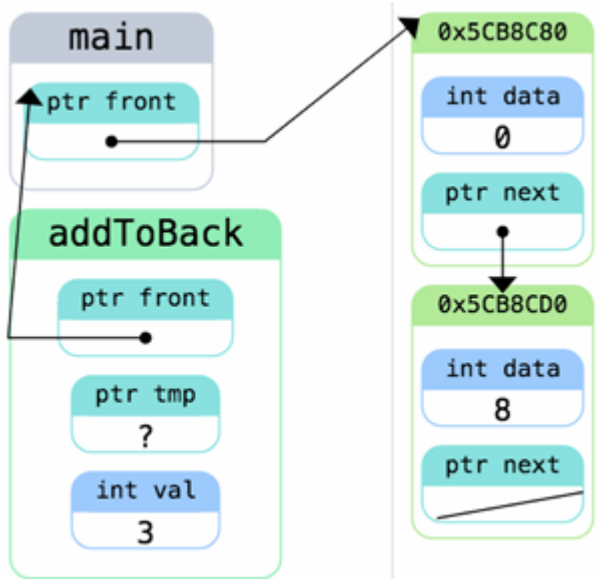
Add to Back

```
void addToBack(ListNode*& front, int value) {
    ListNode* tmp = front;
    while (tmp != nullptr) {
      tmp = tmp->next;
    }
    tmp = new ListNode();
    tmp->val = value;
    tmp->next = nullptr;
 }
```

Is it correct?

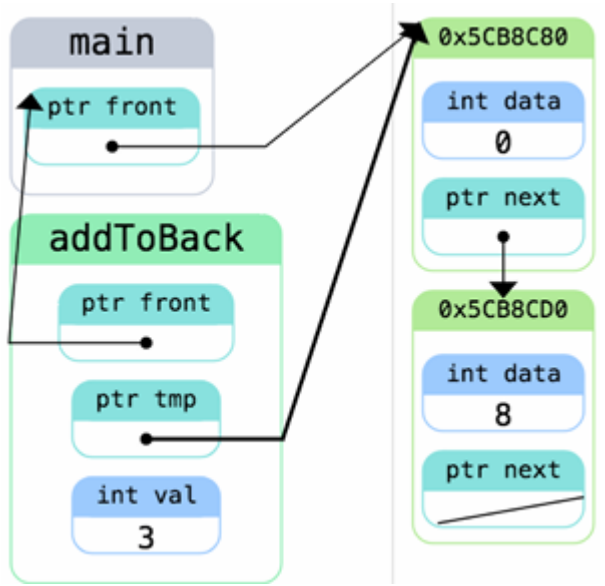
Add to Back

```
void addToBack(ListNode*& front, int value) {
    ListNode* tmp = front;
    while (tmp != nullptr) {
      tmp = tmp->next;
    }
    tmp = new ListNode();
    tmp->val = value;
    tmp->next = nullptr;
}
```



Add to Back

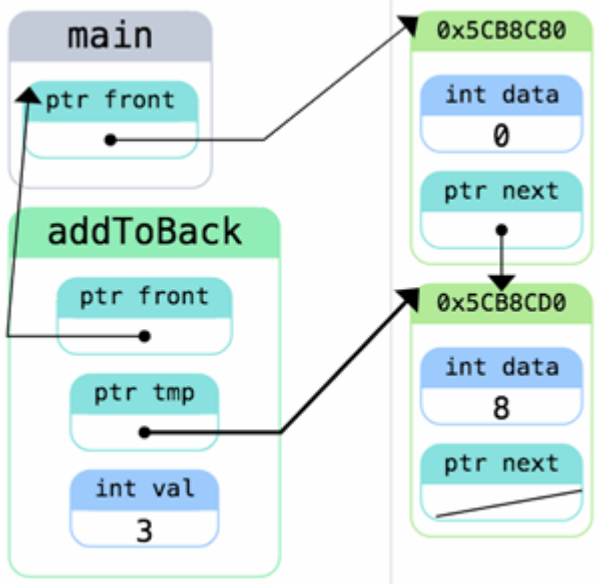
```
void addToBack(ListNode*& front, int value) {
    ListNode* tmp = front;
    while (tmp != nullptr) {
      tmp = tmp->next;
    }
    tmp = new ListNode();
    tmp->val = value;
    tmp->next = nullptr;
}
```



Add to Back

```
void addToBack(ListNode*& front, int value) {
    ListNode* tmp = front;
    while (tmp != nullptr) {
      tmp = tmp->next;
    }
    tmp = new ListNode();
    tmp->val = value;
    tmp->next = nullptr;
}
```

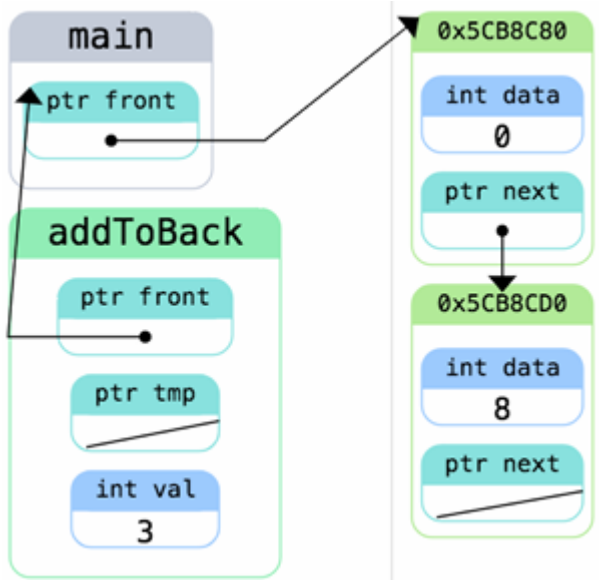
(1st loop)



Add to Back

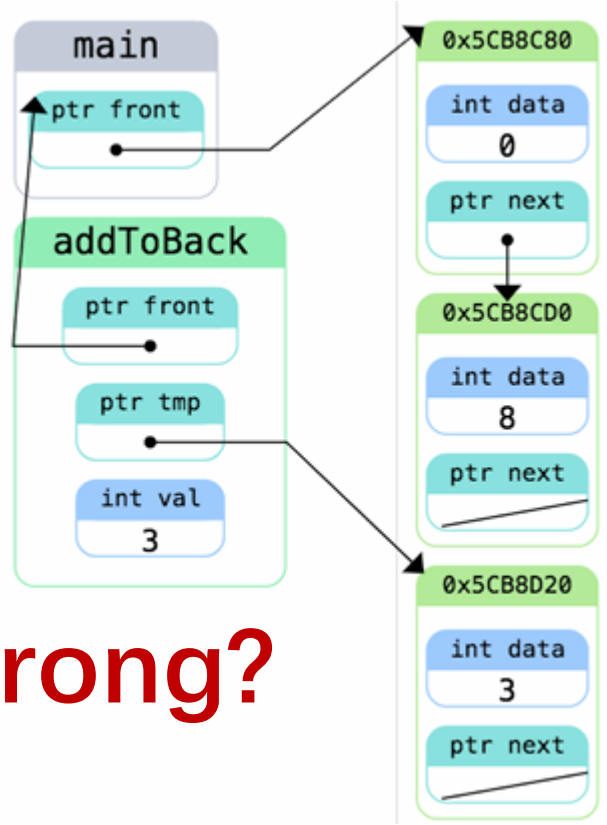
```
void addToBack(ListNode*& front, int value) {
    ListNode* tmp = front;
    while (tmp != nullptr) {
      tmp = tmp->next;
    }
    tmp = new ListNode();
    tmp->val = value;
    tmp->next = nullptr;
}
```

(2nd loop)



Add to Back

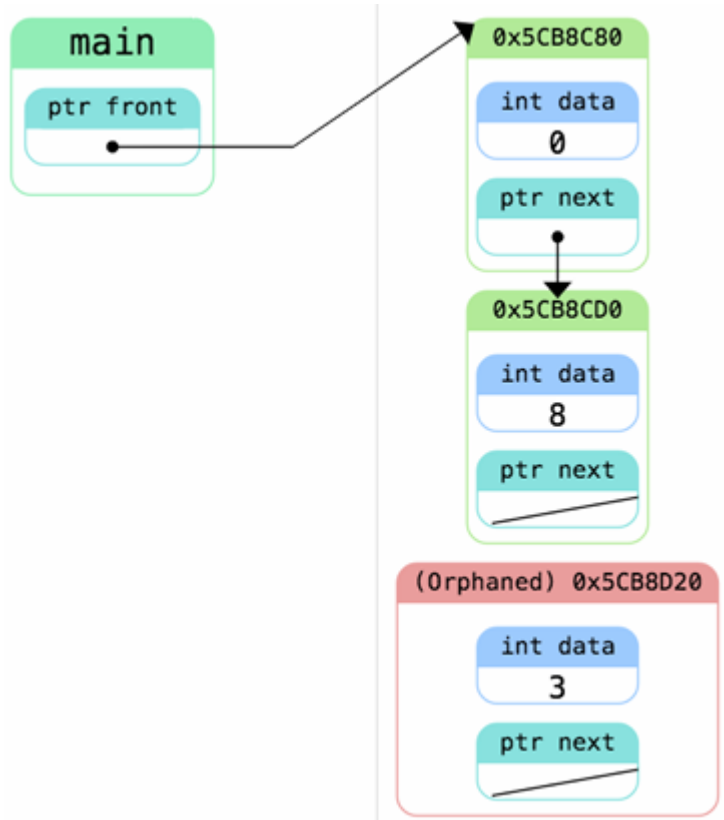
```
void addToBack(ListNode*& front, int value) {
    ListNode* tmp = front;
    while (tmp != nullptr) {
      tmp = tmp->next;
    }
    tmp = new ListNode();
    tmp->val = value;
    tmp->next = nullptr;
}
```



What's Wrong?

Add to Back

```
void addToBack(ListNode*& front, int value) {
    ListNode* tmp = front;
    while (tmp != nullptr) {
      tmp = tmp->next;
    }
    tmp = new ListNode();
    tmp->val = value;
    tmp->next = nullptr;
}
```



After exiting **addToBack** function

Add to Back: Key Point

- When modifying (adding to or removing from) a linked list, we need to be **one node away from the node we want to impact** (**one level of indirection**)
- How do we add the node after our current node then?

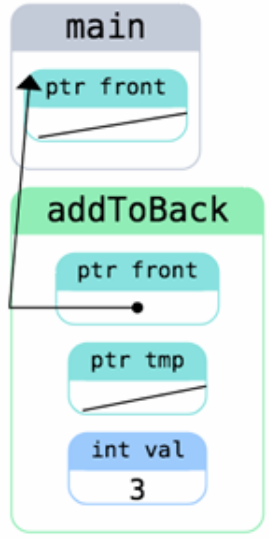
Add to Back

```
void addToBack(ListNode*& front, int value) {
    ListNode* tmp = front;
    while (tmp->next != nullptr) {
      tmp = tmp->next;
    }
    tmp->next = new ListNode();
    tmp->next->val = value;
    tmp->next->next = nullptr;
 }
```

Is It Good Enough?

Add to Back

```
void addToBack(ListNode*& front, int value) {
    ListNode* tmp = front;
    while (tmp->next != nullptr) {
      tmp = tmp->next;
    }
    tmp->next = new ListNode();
    tmp->next->val = value;
    tmp->next->next = nullptr;
}
```



What if we pass in an empty list?

Add to Back

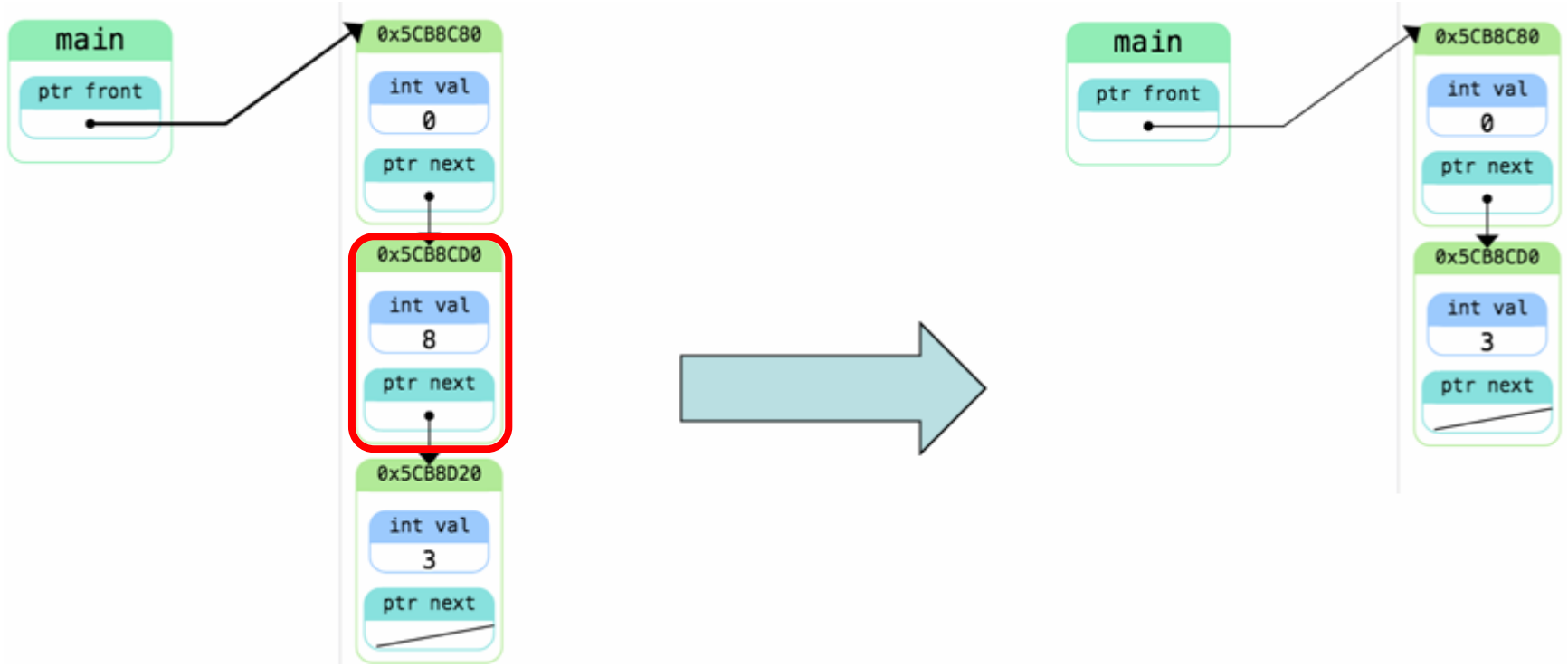
```
void addToBack(ListNode*& front, int value) {
    ListNode* tmp = front;
    if (front == nullptr) {
        front = new ListNode{value, nullptr};
        return;
    }
    while (tmp->next != nullptr) {
      tmp = tmp->next;
    }
    tmp->next = new ListNode();
    tmp->next->val = value;
    tmp->next->next = nullptr;
}
```

Must take care of the edge case when front is an empty list

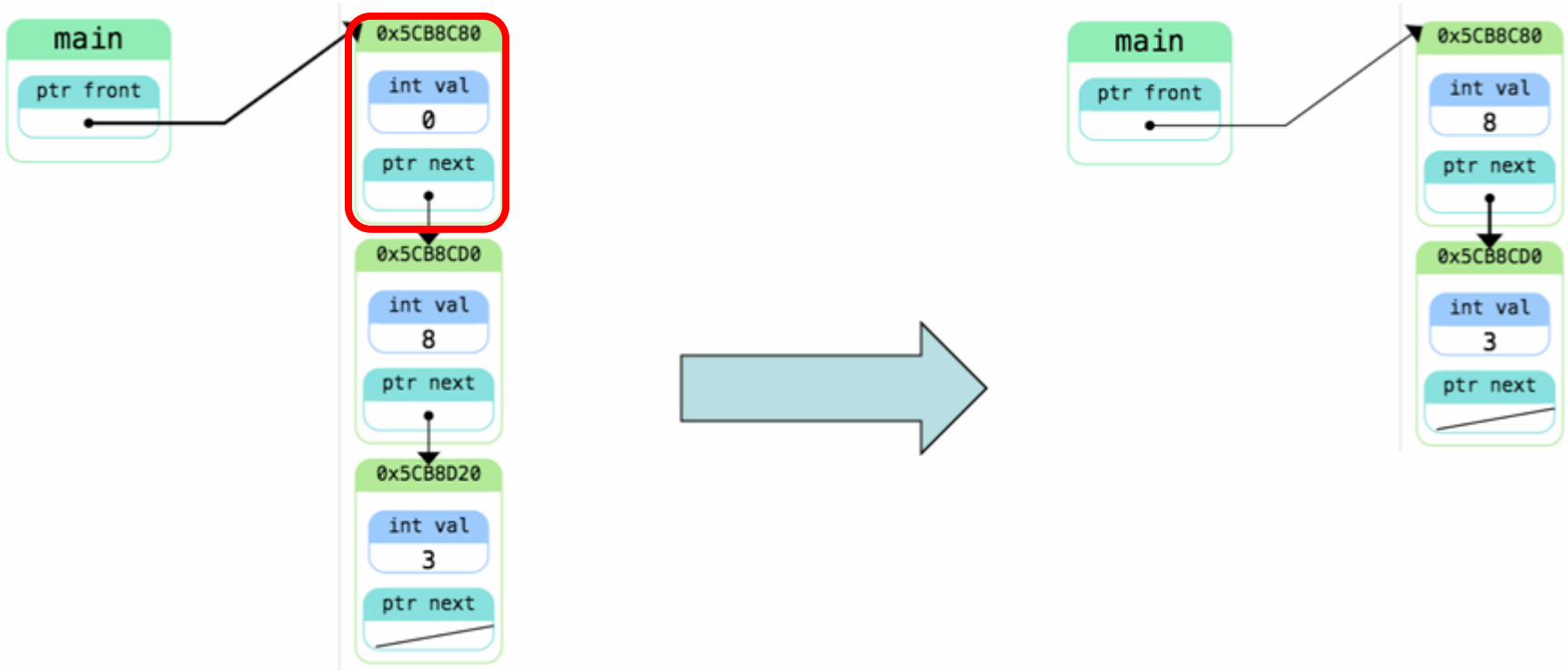
Remove Node via Indexing

- We've seen how to add to a Linked List
- How would we **remove an element from a specific index** in the linked list?
- How do we want to rewire the pointers?
- Should we pass by value or by reference?
- What edge cases should we consider?
  - **Empty list**
  - **Removing from the front**
  - **Removing from the back**
- Assume for now that the list has an element in that index.

Remove Middle Node

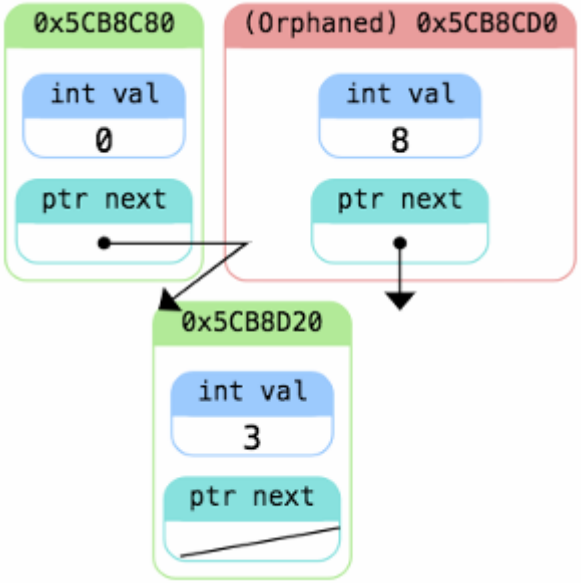


Remove Front Node



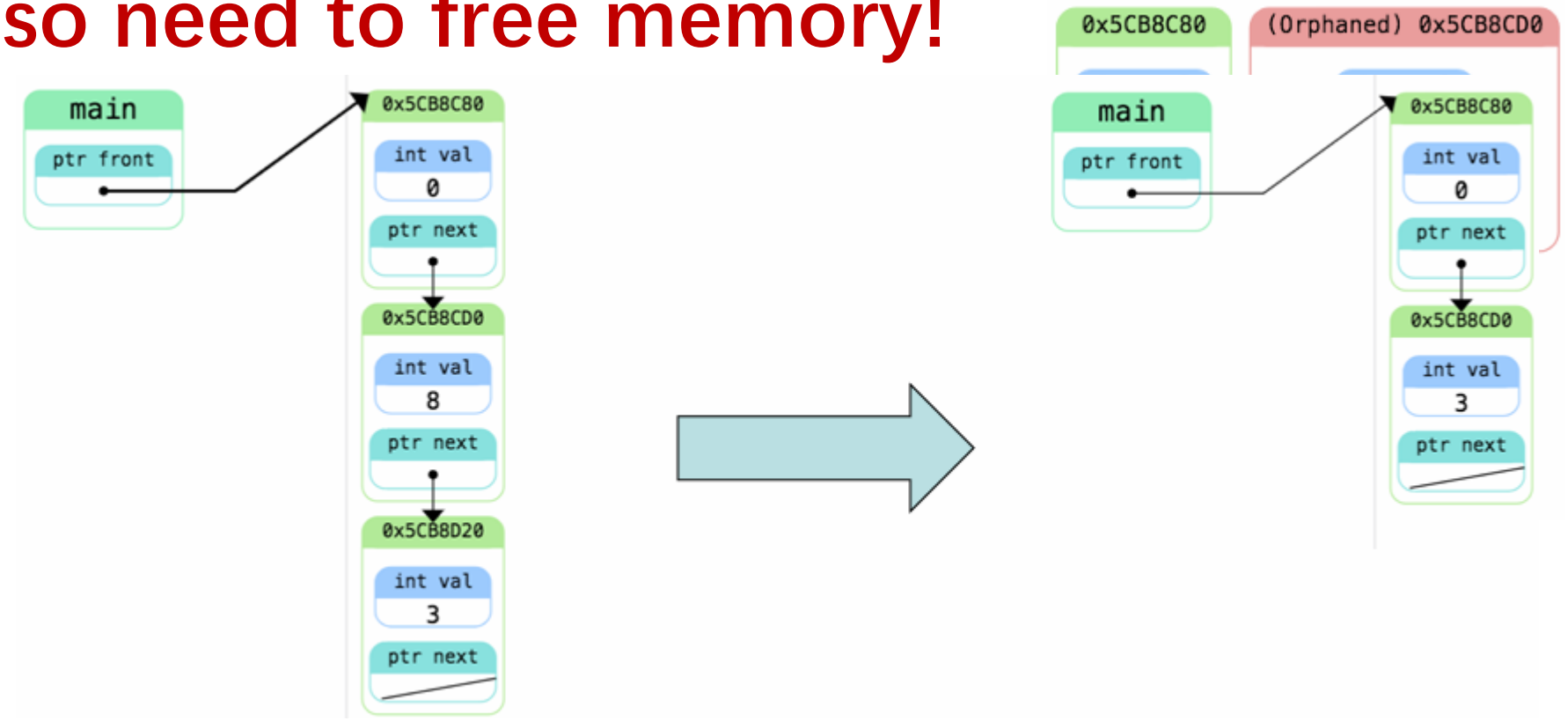
Remove Node via Indexing

```
void removeIndex(ListNode*& front, int index) {
    if (index == 0) {
      front = front->next;
      return;
    }
    ListNode *tmp = front;
    for (int i = 0; i < index-1; i++) {
            tmp = tmp->next;
    }
    tmp->next = tmp->next->next;
}
```



Remove Node via Indexing

We also need to free memory!

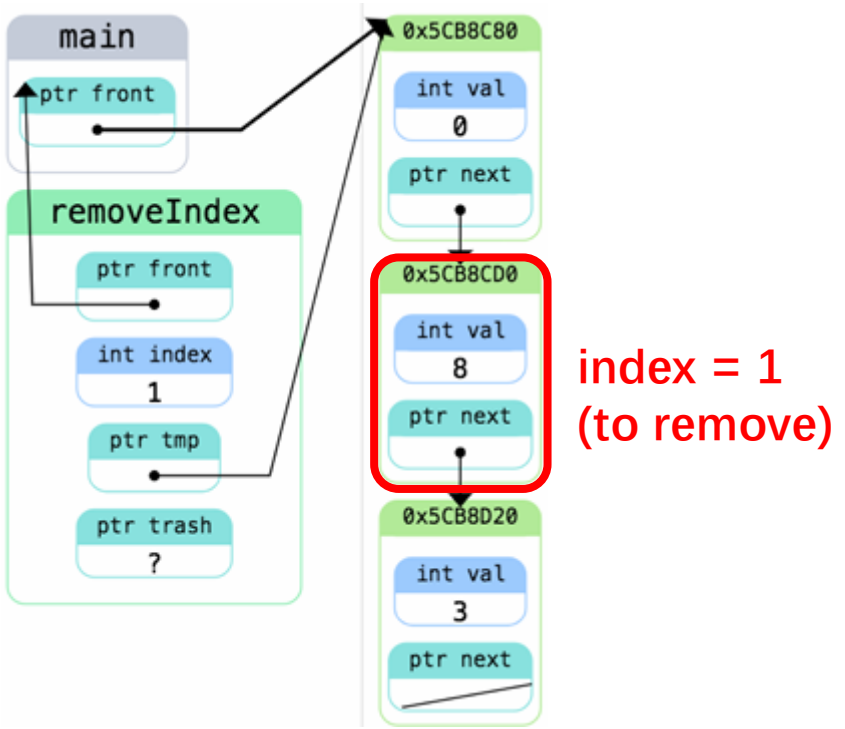


Remove Node via Indexing

```
void removeIndex(ListNode*& front, int index) {
    if (index == 0) {
      ListNode* trash = front;
      front = front->next;
      delete trash;
      return;
    }
    ListNode *tmp = front;
    for (int i = 0; i < index-1; i++) {
            tmp = tmp->next;
    }
    ListNode* trash = tmp->next;
    tmp->next = tmp->next->next;
    delete trash;
}
```

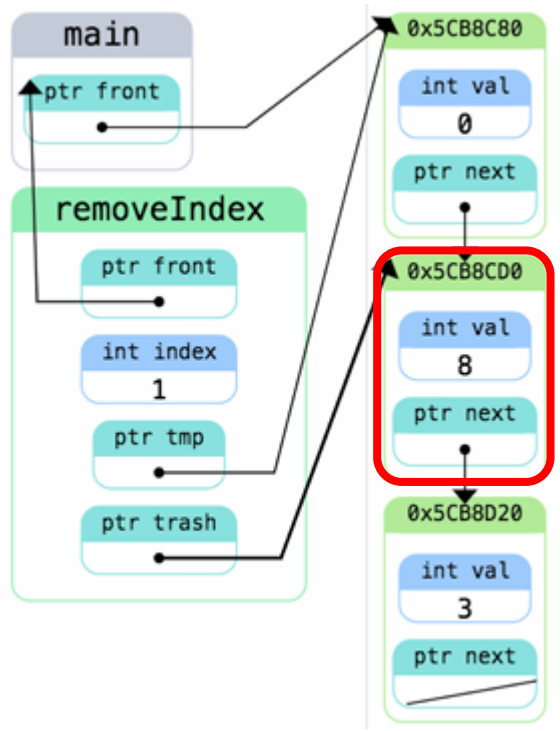
Remove Node via Indexing

```
void removeIndex(ListNode*& front, int index) {
    if (index == 0) {
      ListNode* trash = front;
      front = front->next;
      delete trash;
      return;
    }
    ListNode *tmp = front;
    for (int i = 0; i < index-1; i++) {
            tmp = tmp->next;
    }
    ListNode* trash = tmp->next;
    tmp->next = tmp->next->next;
    delete trash;
}
```



Remove Node via Indexing

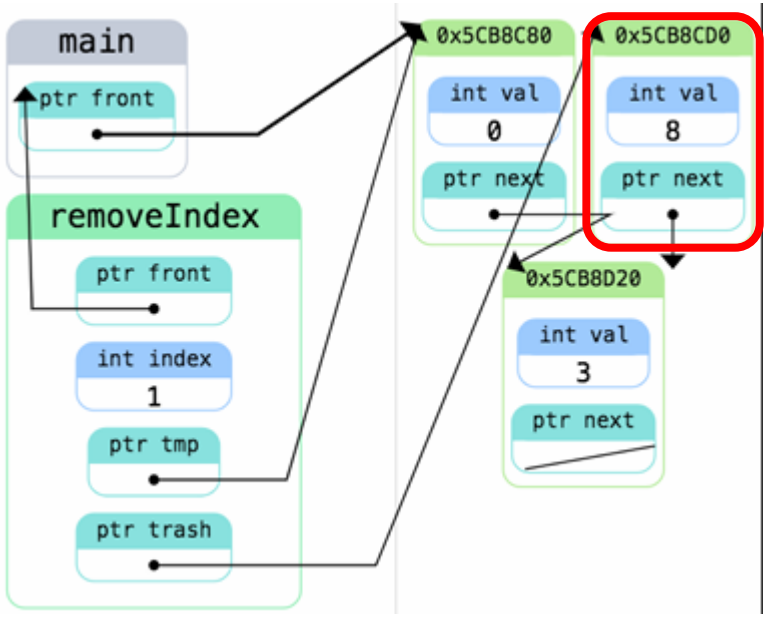
```
void removeIndex(ListNode*& front, int index) {
    if (index == 0) {
      ListNode* trash = front;
      front = front->next;
      delete trash;
      return;
    }
    ListNode *tmp = front;
    for (int i = 0; i < index-1; i++) {
            tmp = tmp->next;
    }
    ListNode* trash = tmp->next;
    tmp->next = tmp->next->next;
    delete trash;
}
```



index = 1
(to remove)

Remove Node via Indexing

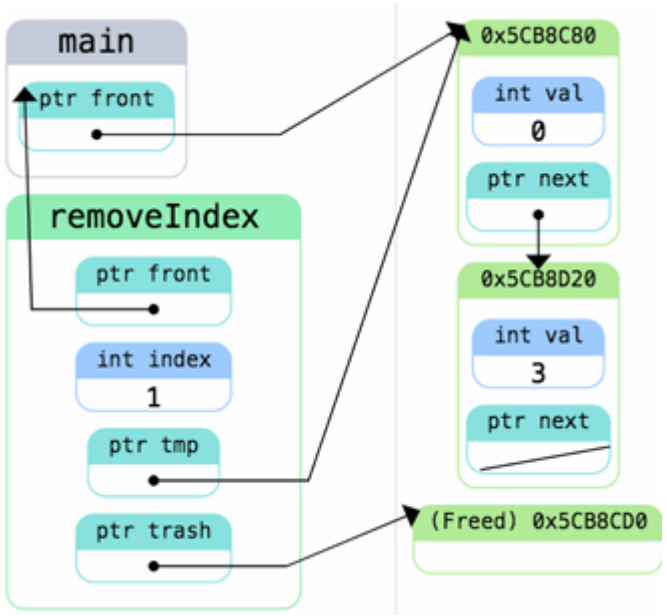
```
void removeIndex(ListNode*& front, int index) {
    if (index == 0) {
      ListNode* trash = front;
      front = front->next;
      delete trash;
      return;
    }
    ListNode *tmp = front;
    for (int i = 0; i < index-1; i++) {
            tmp = tmp->next;
    }
    ListNode* trash = tmp->next;
    tmp->next = tmp->next->next;
    delete trash;
}
```



(to remove)

Remove Node via Indexing

```
void removeIndex(ListNode*& front, int index) {
    if (index == 0) {
      ListNode* trash = front;
      front = front->next;
      delete trash;
      return;
    }
    ListNode *tmp = front;
    for (int i = 0; i < index-1; i++) {
            tmp = tmp->next;
    }
    ListNode* trash = tmp->next;
    tmp->next = tmp->next->next;
    delete trash;
}
```



Linked List: Summary

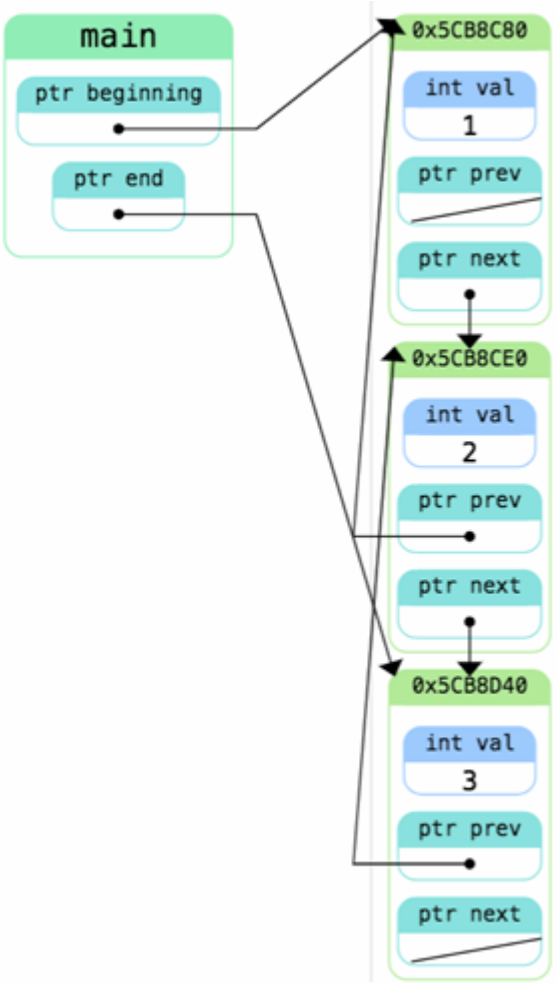
- Every element in a Linked List is stored in its own block, which we call a ListNode - Can only access an element by visiting every element before it
- When **modifying** the list, pass the front ListNode by **reference**
- **Edge cases**: Test your code with a Linked List of size 0, 1, 2, and 3, and with operations on the beginning, middle, and end
- When in doubt, draw out a memory diagram
- **Practice safe pointers**: always check for null before dereferencing!

Linked List: Pros and Cons

- Pros:
  - Fast to add/remove near the front of the list
    - Great for queues, especially if we keep a pointer to the end
  - Can merge or concatenate two linked lists without allocating any more memory
  - Nodes are stored wherever there is free space in memory, the nodes do not have to be stored contiguously
- Cons:
  - Slow to "index" into the list
  - Slow to add/remove in the middle or near the end of the list
  - Can only iterate one way

Doubly-Linked List

- Have each node point to the **next node** in the list and the **previous node** in the list
- Generally store pointer to the front and back
- Advantages:
  - easy to add to the front and the back of the list
  - **don't need a level of indirection** for adding or removing nodes



Exercise 5.1

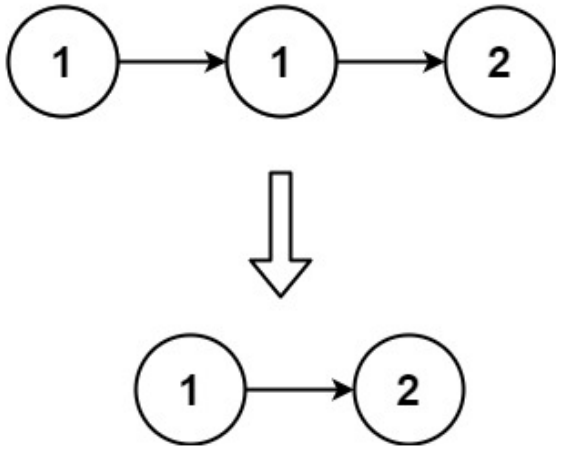
- Complete LeetCode 83

83. Remove Duplicates from Sorted List

Easy | Topics | Companies

Given the `head` of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list* ***sorted*** *as well*.

Example 1:




```
Input: head = [1,1,2]
Output: [1,2]
```

Exercise 5.2

- Complete LeetCode 21

21. Merge Two Sorted Lists

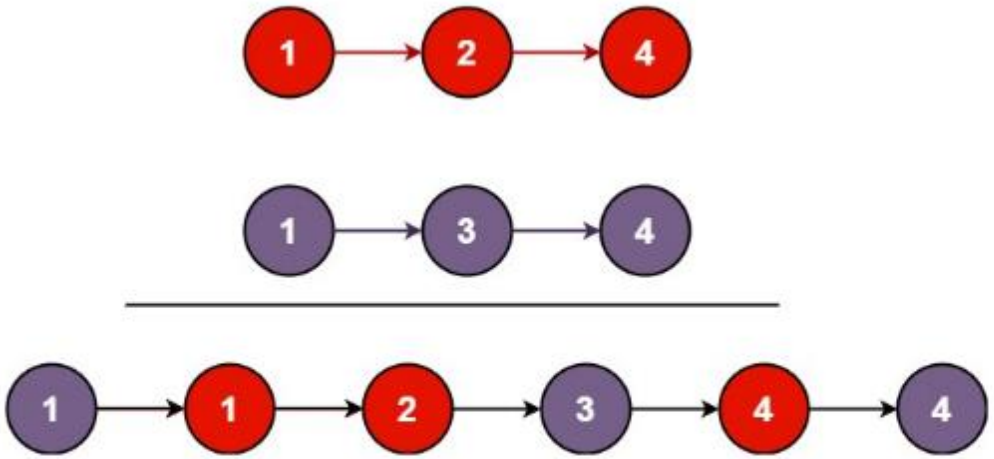
Easy | Topics | Companies

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list.*

Example 1:




```
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

Exercise 5.3

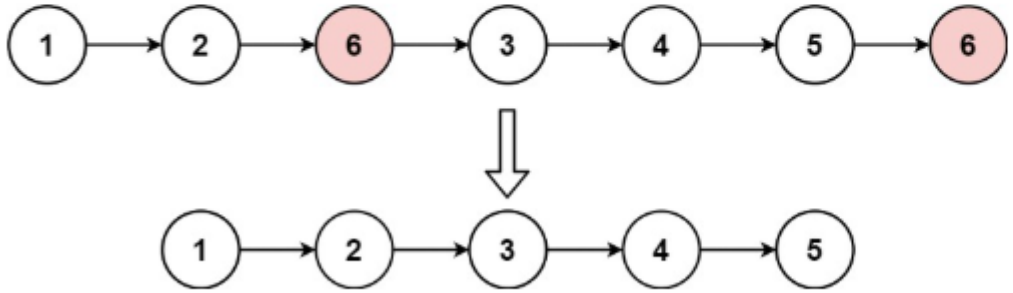
- Complete LeetCode 203

203. Remove Linked List Elements

Easy | Topics | Companies

Given the `head` of a linked list and an integer `val`, remove all the nodes of the linked list that has `Node.val == val`, and return *the new head*.

Example 1:




```
Input: head = [1,2,6,3,4,5,6], val = 6
Output: [1,2,3,4,5]
```

Exercise 5.4

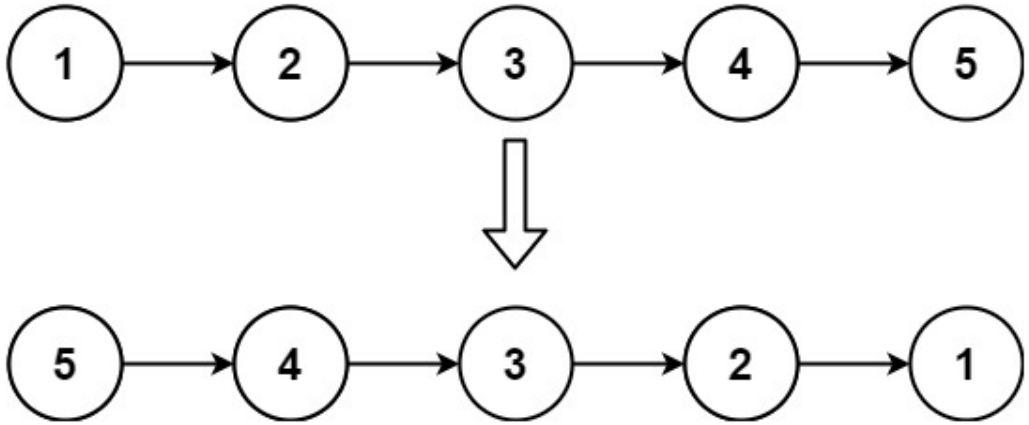
- Complete LeetCode 206

206. Reverse Linked List

Easy | Topics | Companies

Given the `head` of a singly linked list, reverse the list, and return *the reversed list*.

Example 1:




```
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
```

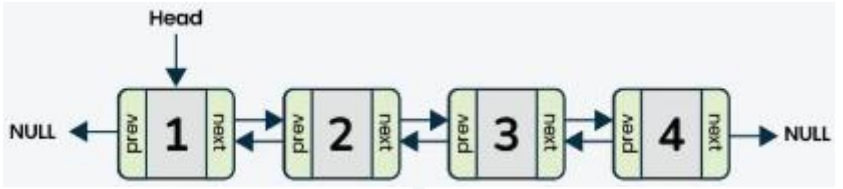
Exercise 5.5

- Reverse a doubly linked List
  - Define a doubly linked list

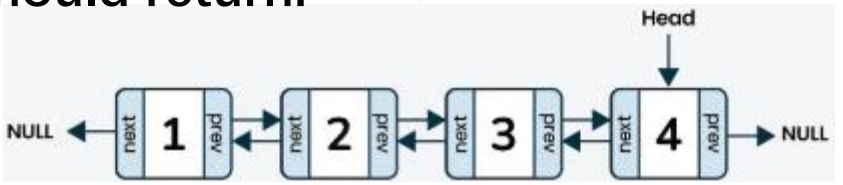
```
#include <iostream>
using namespace std;

struct ListNode {
    int val;
    ListNode*next, *prev;
    ListNode(int x) : val(x), next(nullptr), prev(nullptr){}
};
```

For example, given an input doubly linked list containing four nodes:



Your program should return:



Exercise 5.5 (Continue)

```
ListNode* Reverse(ListNode* head) {
    // (Take care of the corner case) If the list is empty or has only one node, return the head as is
    if (head == nullptr || head->next == nullptr)
        return head;

    ListNode* prevNode = NULL;
    ListNode* currNode = head;

    // Traverse the list and reverse the links
    while (currNode != nullptr) {
        // Swap the next and prev pointers (complete the code)

        // Move to the next node in the original list, or say, previous node in the reversed list (complete the code)

    }

    // The final node in the original list becomes the new head after reversal
    return prevNode->prev;
}
```

Exercise 5.5 (Continue)

Use the following code to test your solutions，check whether you get 4, 3 ,2, 1 after reversing：

```
void printList(ListNode* node) {
    while (node != nullptr) {
        cout << node->val << " ";
        node = node->next;
    }
    cout << endl;
}

int main() {
    ListNode *head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->prev = head;
    head->next->next = new ListNode(3);
    head->next->next->prev = head->next;
    head->next->next->next = new ListNode(4);
    head->next->next->next->prev = head->next->next;
    cout << "Original Doubly Linked List" << endl; printList(head);
    head = Reverse(head);
    cout << "Reversed Doubly Linked List" << endl; printList(head);
    return 0;
}
```

(You can also test other doubly linked lists by changing the node construction here)