CS 106B, Lecture 19
Linked Lists II

Plan for Today

- Modifying linked lists: Implementing add and delete from a Linked List
- Common Linked Lists gotchas and Linked List tips
- Doubly-Linked Lists
- Linked List as a class

Add to Front

- How would 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

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

Add to Back

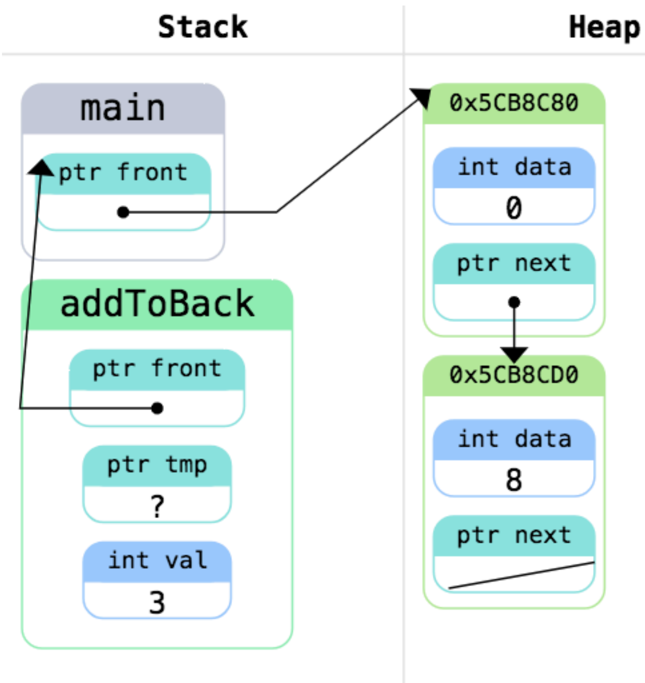
- How would we add to the back of a Linked List?
- Should the front be passed by **reference** or by **value**?

Add to Back: First Try

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

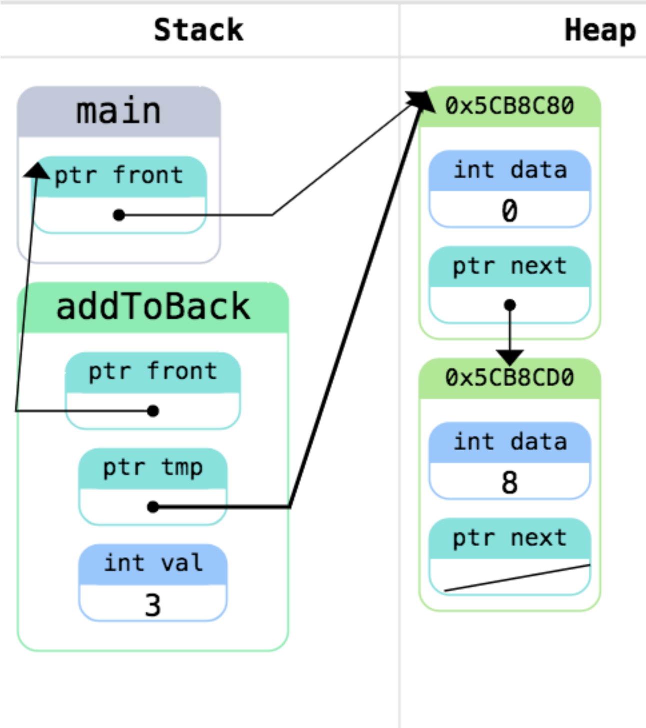
Add to Back: First Try

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



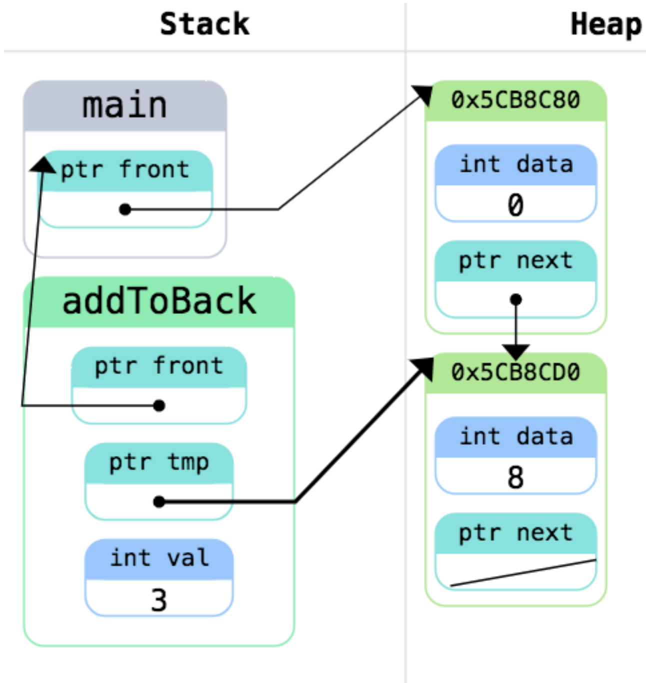
Add to Back: First Try

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



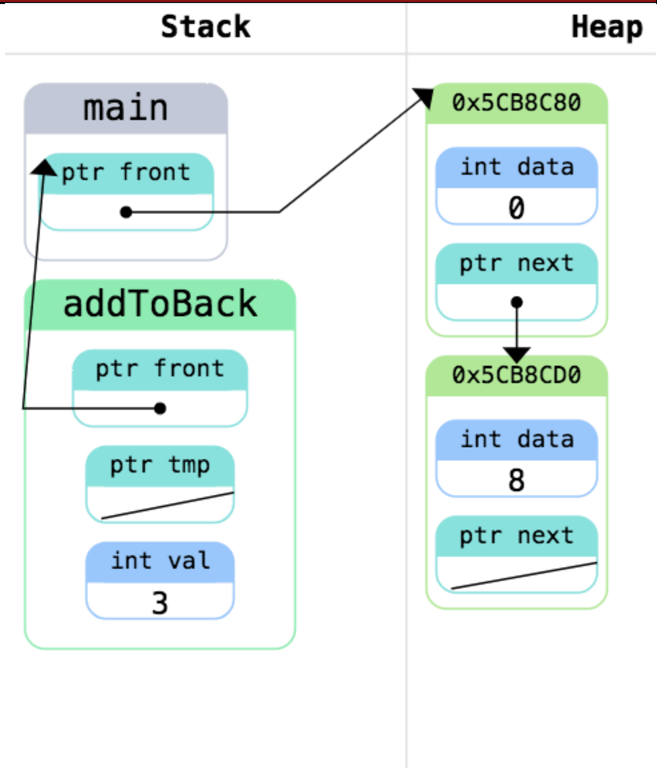
Add to Back: First Try

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



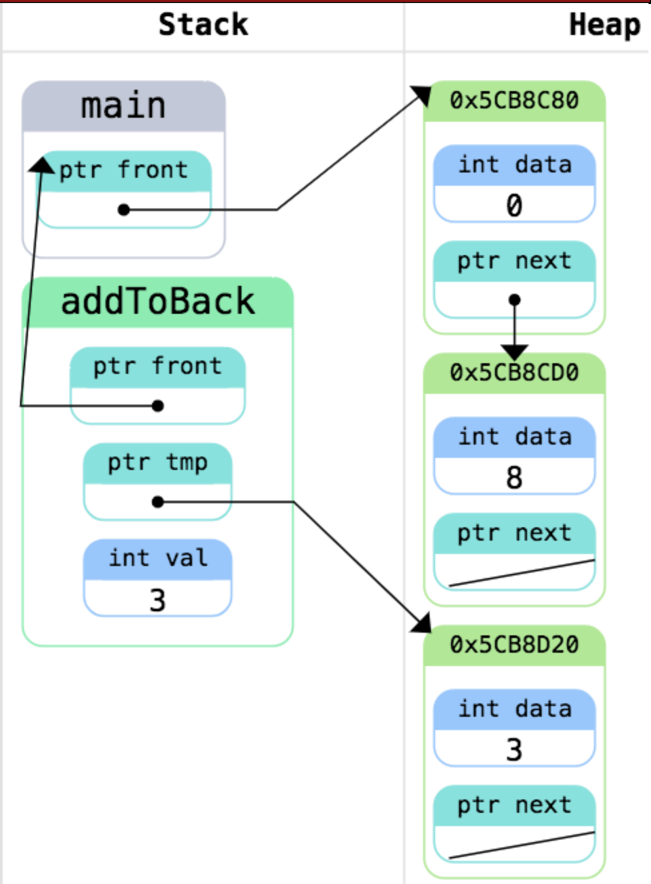
Add to Back: First Try

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



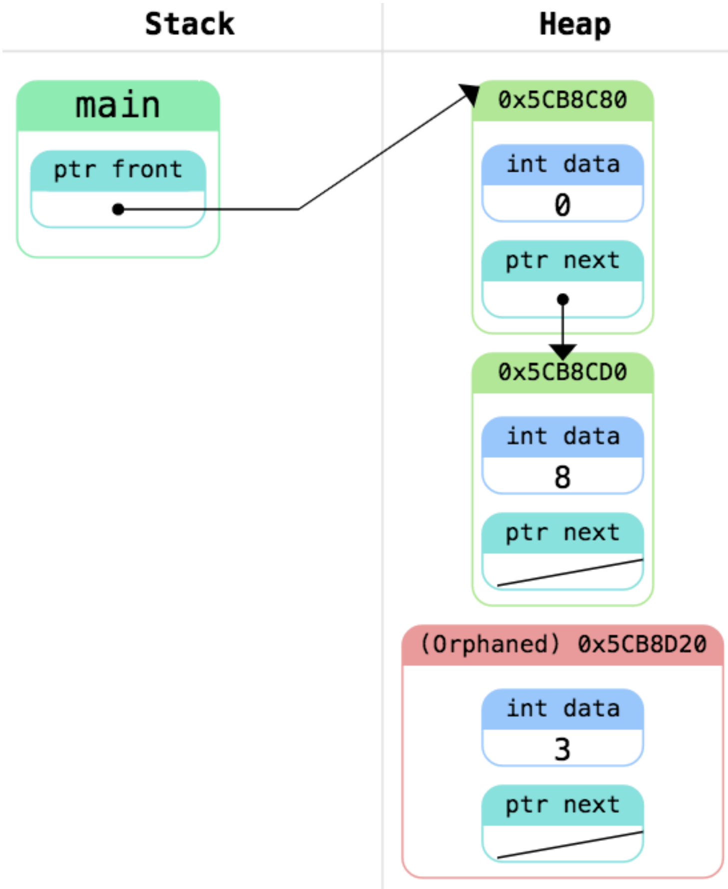
Add to Back: First Try

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



Add to Back: First Try

```
void addToBack(ListNode *&front,
               int val) {
    ListNode *tmp = front;
    while (tmp != nullptr) {
        tmp = tmp->next;
    }
    tmp = new ListNode();
    tmp->data = val;
    tmp->next = nullptr;
}
// in main after call to addToBack
```

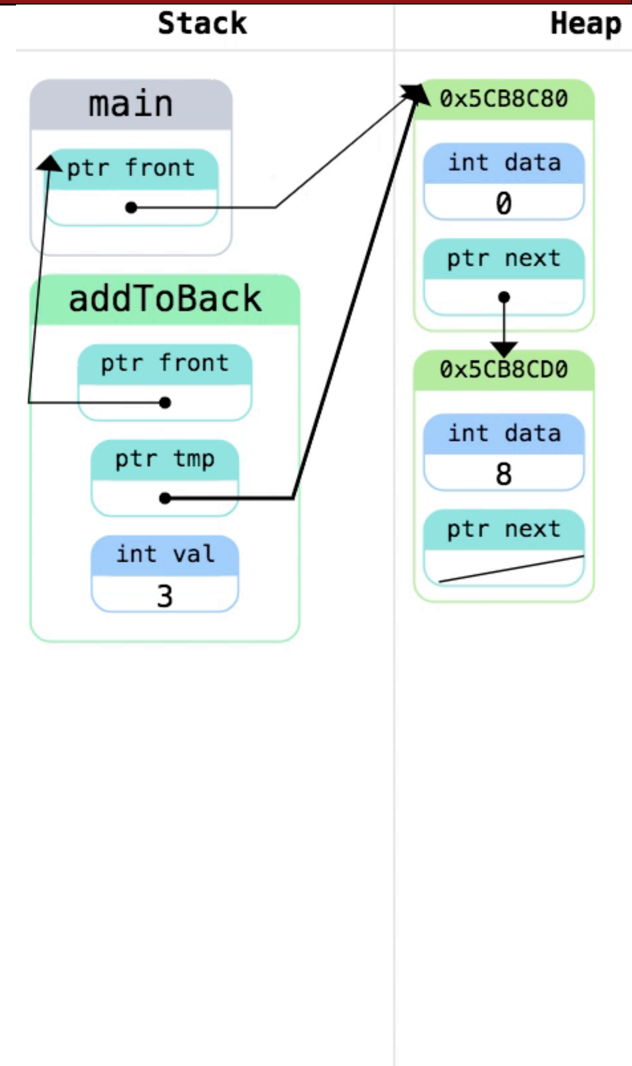


Add to Back: Key Point

- When modifying (adding to or removing from) a linked list, we need to be **one node away** from the node we want to impact (**layer of indirection)**
  - In this case, we need to add the node **after our current node** – how could we do that?

Add to Back: Second Try

```
void addToBack(ListNode *&front,
               int val) {
    ListNode *tmp = front;
    while (tmp->next != nullptr) {
        tmp = tmp->next;
    }
    tmp->next = new ListNode();
    tmp->next->data = val;
    tmp->next->next = nullptr;
}
// in main after call to addToBack
```

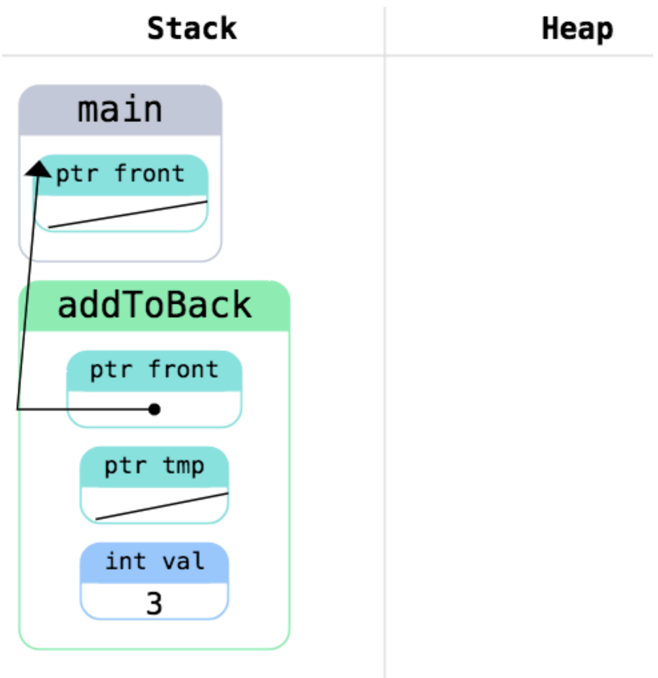


Add to Back: Second Try

```
// what if we pass in an empty list?
void addToBack(ListNode *&front,
               int val) {
    ListNode *tmp = front;
    while (tmp->next != nullptr) {
        tmp = tmp->next;
    }
    tmp->next = new ListNode;
    tmp->next->data = val;
    tmp->next->next = nullptr;
}
```

Add to Back: Second Try

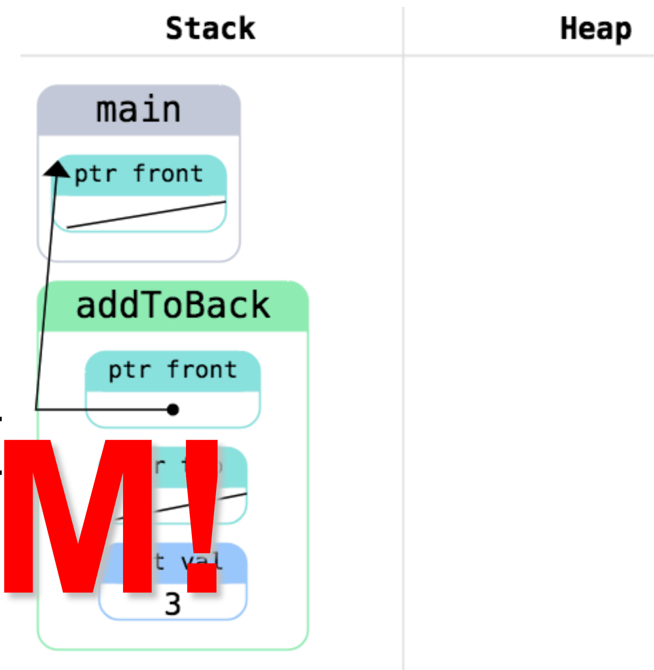
```
// good edge case: empty list
void addToBack(ListNode *&front,
               int val) {
    ListNode *tmp = front;
    while (tmp->next != nullptr) {
        tmp = tmp->next;
    }
    tmp->next = new ListNode;
    tmp->next->data = val;
    tmp->next->next = nullptr;
}
// in main after call to addToBack
```



Add to Back: Second Try

```
// good edge case: empty list
void addToBack(ListNode *&front,
                int val) {
    ListNode *tmp = front;
    while (tmp->next != nullptr) {
        tmp = tmp->next;
    }
    tmp->next = new ListNode;
    tmp->next->data = val;
    tmp->next->next = nullptr;
}
// in main after call to addToBack
```

KABOOM!

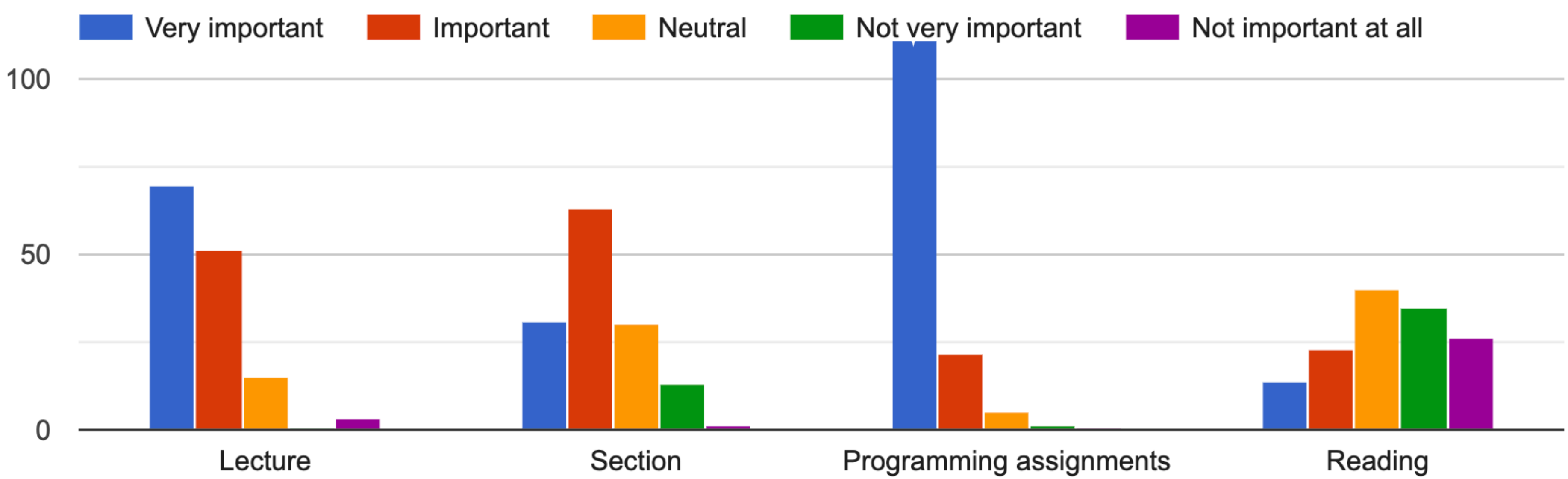


Add to Back: Solution

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

Announcements

- Class Survey
  - Thank you to everyone who participated in the class survey.
  - It remains open. So feel free to add any feedback.
  - Currently at ~73%. I will lower it to 80% for a free late day for everyone! You must finish it by the end of day Wednesday



Announcements

- Doing Well
  - "Very good job explaining concepts, the examples help a lot."
  - "Being serious in class"
- To Improve Upon
  - "Sometimes he's super serious when answering questions"
  - "Sometimes he speaks a little too fast but that is only a problem if you watch lecture on 1.5x speed"
  - "He speaks really rapid-fire, then takes a long break…"
  - "Choice of songs"
- One thing
  - "switch the playlist plz!"
  - "Having an assignment due one day after the midterm was a little brutal. …but having the assignments back earlier than the day the next assignment is due would help us incorporate feedback."

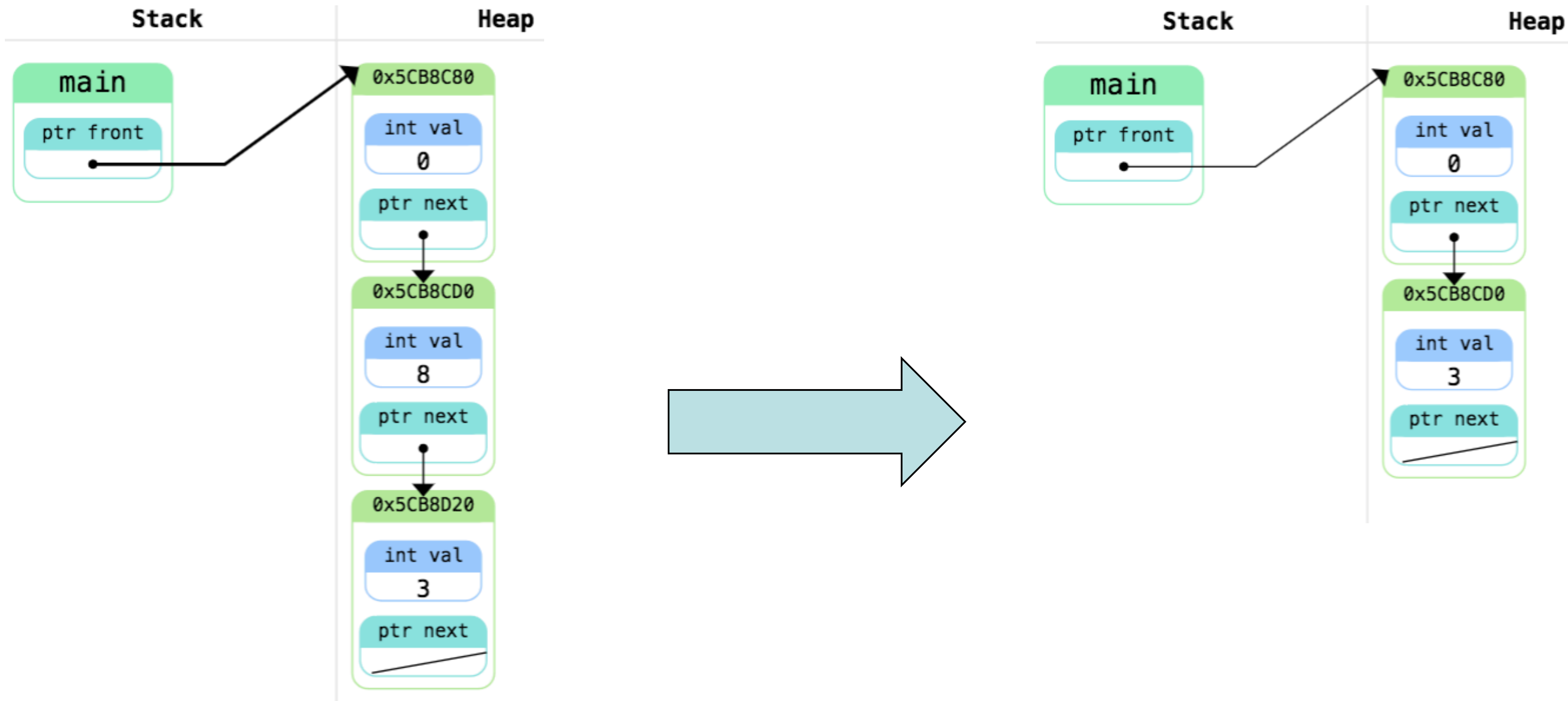
Announcements

- Playlist Link
  - https://open.spotify.com/user/122062784/playlist/4hlXo8uRQjiOPplhQbxtpQ?si=eIKa8qv0Qj-raqwBDtTuvQ

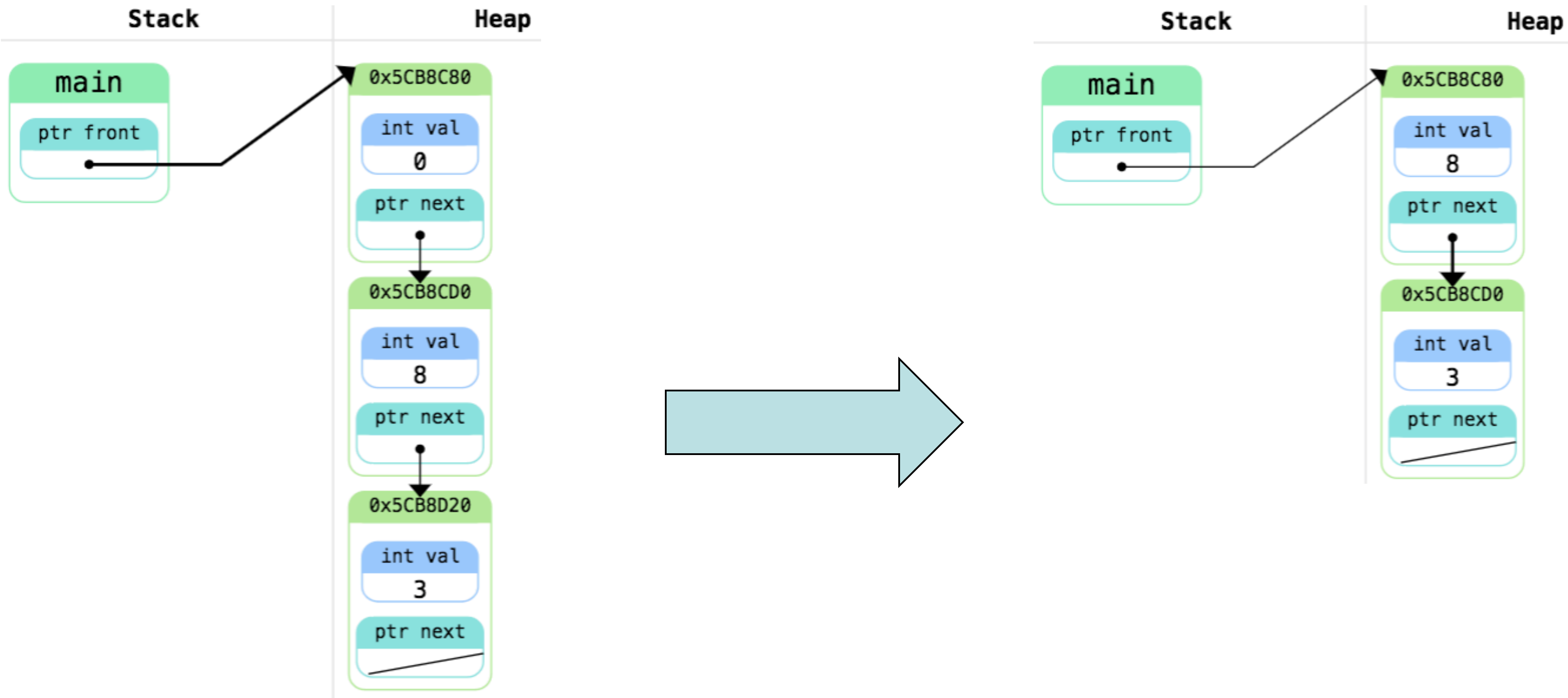
Remove Index

- 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



Remove 0

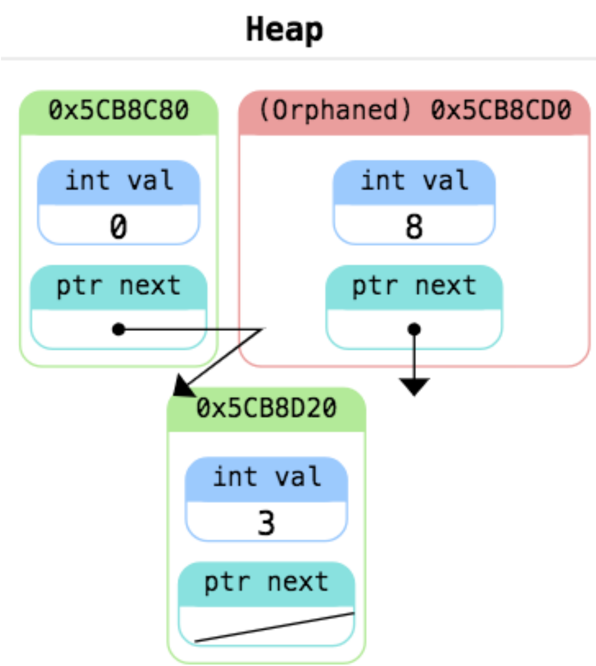

Stack
Heap
main
ptr front
0x5CB8C80
int val
0
ptr next
0x5CB8CD0
int val
8
ptr next
0x5CB8D20
int val
3
ptr next
Stack
Heap
main
ptr front
0x5CB8C80
int val
8
ptr next
0x5CB8CD0
int val
3
ptr next

Remove Index: First Try

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

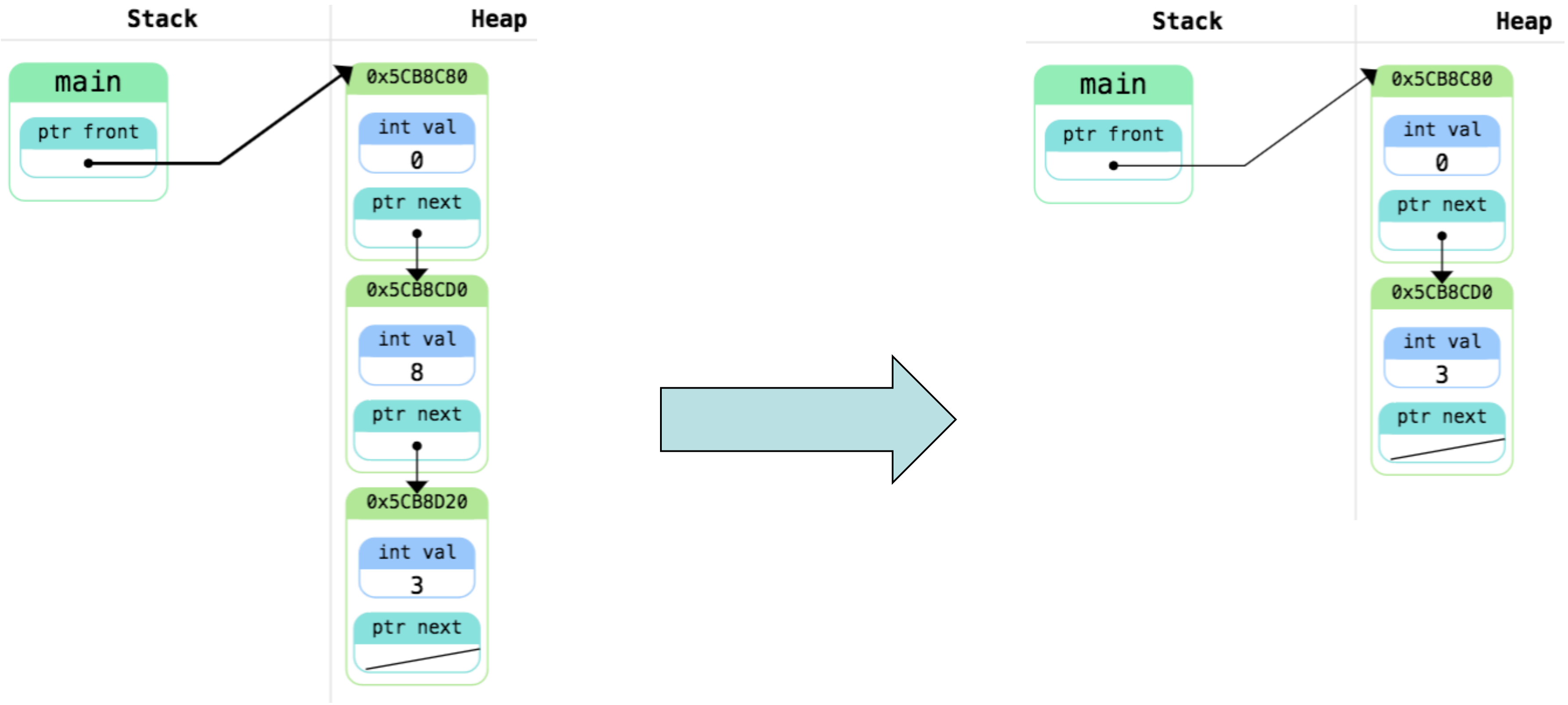
Remove Index: First Try

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



Remove Index

- We also need to free memory. How would we do that?

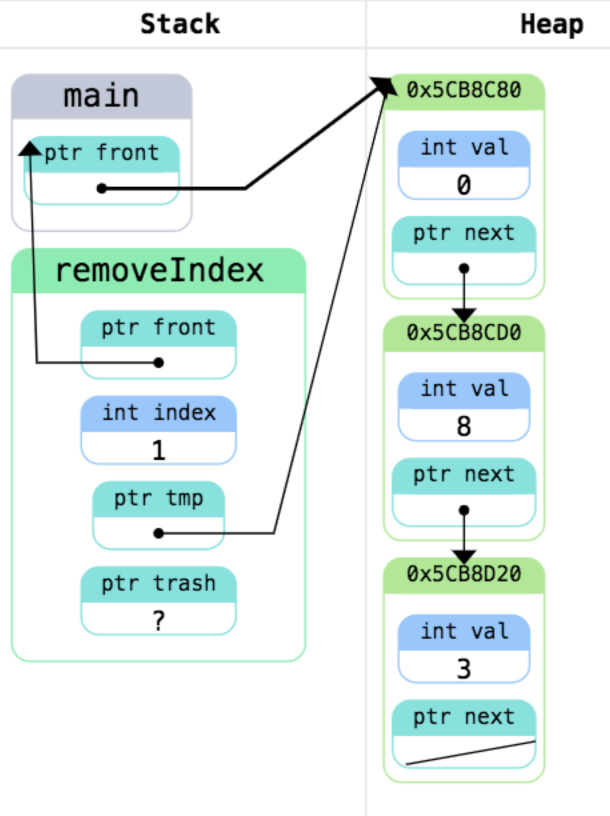


Remove Index: Solution

```
void removeIndex(ListNode *&front, int index) {
  if (index == 0) {
    ListNode *trash = front;
    front = front->next;
    delete trash;
  } else {
    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;
  }
}
```

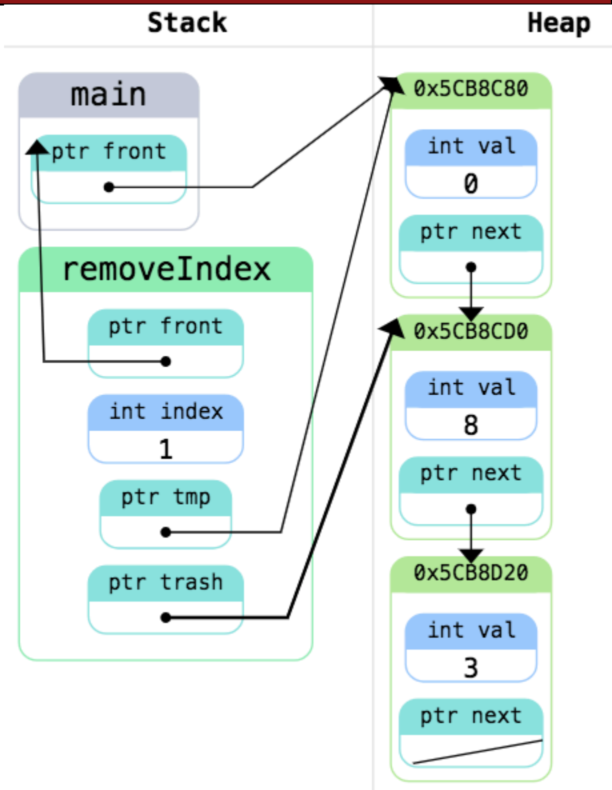
Remove Index: Solution

```
void removeIndex(ListNode *&front, int index) {
  if (index == 0) {
    ListNode *trash = front;
    front = front->next;
    delete trash;
  } else {
    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;
  }
}
```



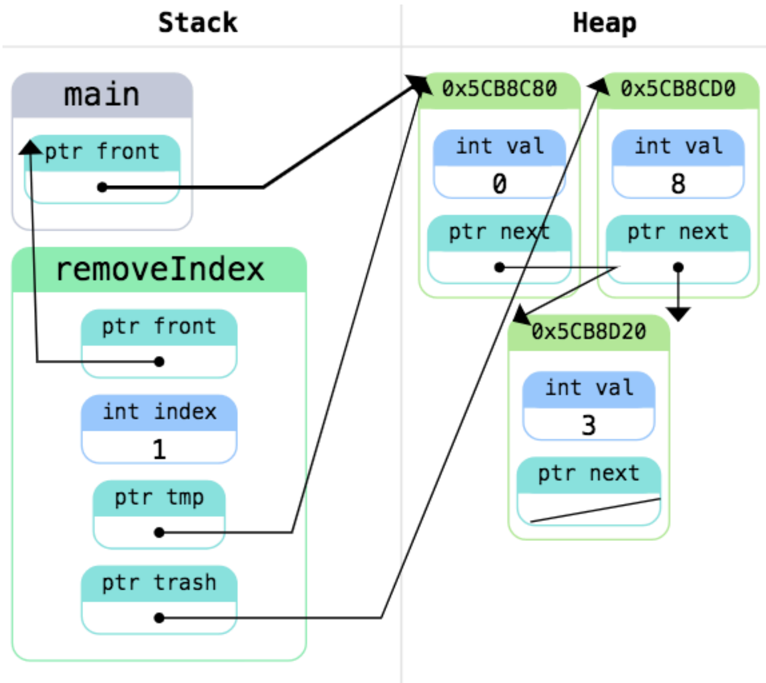
Remove Index: Solution

```
void removeIndex(ListNode *&front, int index) {
  if (index == 0) {
    ListNode *trash = front;
    front = front->next;
    delete trash;
  } else {
    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;
  }
}
```



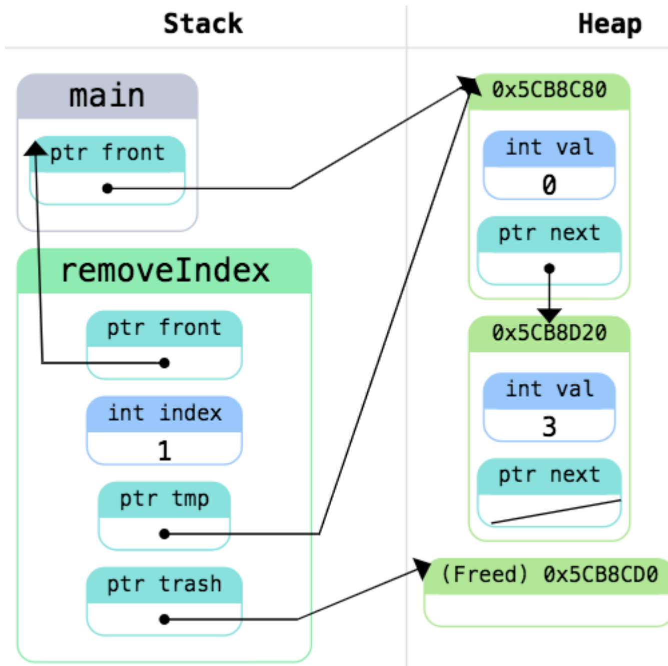
Remove Index: Solution

```
void removeIndex(ListNode *&front, int index) {
  if (index == 0) {
    ListNode *trash = front;
    front = front->next;
    delete trash;
  } else {
    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 Index: Solution

```
void removeIndex(ListNode *&front, int index) {
  if (index == 0) {
    ListNode *trash = front;
    front = front->next;
    delete trash;
  } else {
    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 as a Class

- What instance variables (fields) do we need?
- What should the constructor do? The destructor?
- Idea: instead of passing in front explicitly, store it as an instance variable!

LinkedIntList.h

```
// Represents a linked list of integers.
class LinkedIntList {
public:
    LinkedIntList();
    ~LinkedIntList();
    void addBack(int value);
    void addFront(int value);
    void deleteList();
    void print() const;
    bool isEmpty() const;
    ...
private:
    ListNode* front;   // nullptr if empty
};
```

LinkedIntList.cpp

```
// (partial)
#include "LinkedIntList.h"
LinkedIntList::LinkedIntList() {
    front = nullptr;
}

bool LinkedIntList::isEmpty() {
    return front == nullptr;
}

void LinkedIntList::addFront(int value) {
    ListNode* newNode = new ListNode(value);
    newNode->next = front;
    front = newNode;
}
...
```

Delete Linked List

- How do we delete our linked list?

Delete Linked List

```
void deleteList(ListNode *& front) {
    if (front == nullptr) {
        return;
    }
    deleteList(front->next);
    delete front;
}
```

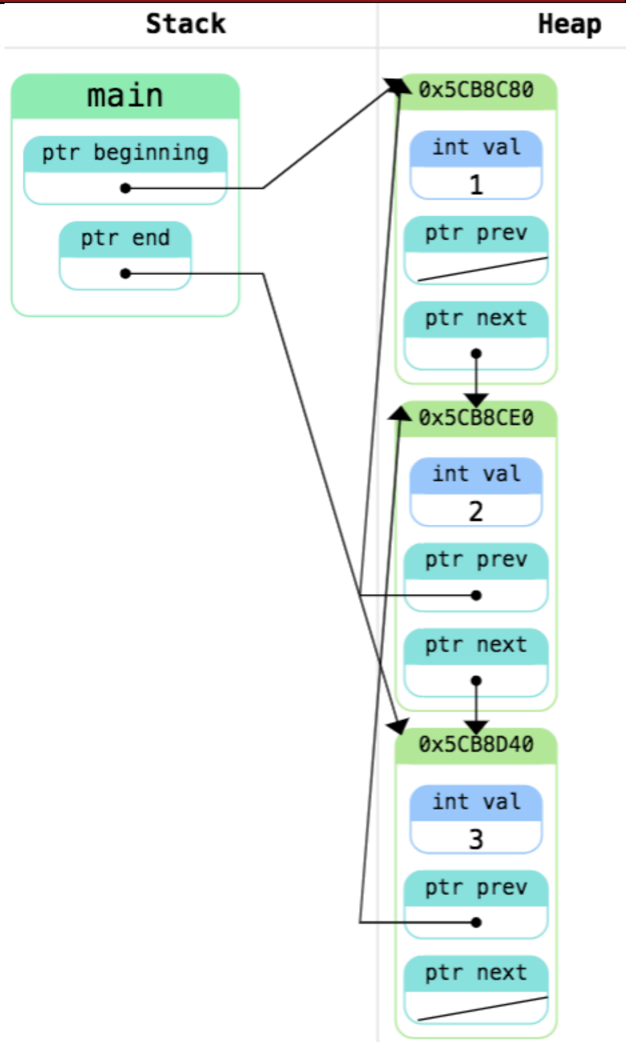
Linked List: Pros and Cons

- Pros:
  - Fast to add/remove near the front of the list
    - Great for queues, especially if we keep a pointer to the end of the LL
  - Can merge or concatenate two linked lists without allocating any more memory
  - Only uses the memory to store the number of elements in the list
- 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/removing nodes

```
struct DoublyListNode {
    int data;
    ListNode *prev;
    ListNode *next;
};
```



Final Thoughts on LL

- 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
- When simply **iterating** through the list, the front ListNode can be passed by value
- **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!**