CS 106B, Lecture 17
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
Recap

• 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
  – Do you see why?
Add to Back

• Yesterday, we talked about how to add to the front of a linked list
• How would we add to the back of a Linked List?
• Should the front be passed by reference or by value?
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

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

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

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

• Assignment 4 is due on **Thursday** – please finish it before then
• You will get assignment 3 feedback on **Wednesday** (tomorrow)
• Please give feedback (if you have the next 30 minutes free): cs198.stanford.edu

• Exam logistics
  – Midterm review session **today**, from 7:00-8:30PM, in Gates B01, led by SL Peter
  – Midterm is on Wednesday (tomorrow), July 25, from 7:00-9:00PM in Hewlett 200
  – Complete assignment 4 before the midterm – backtracking will be tested
• 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?
  – Do we need a layer of indirection?
  – 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.
  – Thought exercise: how would you modify the solution if to handle shorter lists?
Remove Middle
Remove 0
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;
    }
}
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;
    }
}
• We also need to free memory. How would we do that?
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;
    }
}
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;
    }
}
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;
    }
}
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;
  }
}
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!
// 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
};
// (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;
}

...
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
    • Thought experiment: how?
  – 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.
- You'll see these on your next homework.

```c
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 (we've had a lot of these in class!)
• **Practice safe pointers:** always check for null before dereferencing!