Week 1 Lab Notes: Linked Lists

Slides: https://drive.google.com/file/d/13aDyloExvuxH6W-fn_QeNqgBxSZ-0xH5/view

Linked Lists Patterns

- Multiple passes (i.e. getting the length, reversing a list)
    - Used when you need to compute a value before running through the main algorithm
    - Helpful because linked lists only give you a pointer to the head of the list and no other info
    - Simple helper methods using multiple passes probably won't add a significant amount of time to runtime complexity
    - *Aside: recursive vs. iterative—good to master both; recursive may cause stack overflow and is worse on space complexity*
- Two-pointer technique (e.g. using a "slow" and "fast" pointer)
    - Used for cycle detection, figuring out structure of linked list- does it have a cycle or does it end somewhere?
- Dummy head technique
    - Never truly necessary, BUT useful for saving time
    - Helps address edge cases that you might not catch
    - Simplifies solution & makes it easier to work with

Linked Lists Interview Strategies

- Pointer bookkeeping = visualization technique in which you draw out a linked list and pointers and update the pointers visually as you work through your solution
    - Keep track of pointers on a whiteboard and the order in which they're updated
- Helper functions
    - Can write out function signature, and then ask interviewer if it's safe to assume certain basic functions are already implemented
        - E.g. ask interviewer: "I need to calculate the length of this linked list in this algorithm. I'm going to assume the function for doing so already exists. Is that okay?"
        - Communicate with your interviewer about this
    - Useful because working on the core algorithm is more important than implementing these little helper functions
- UMPIRE: Understand, Match, Plan/Pseudocode, Implement, Reflect and verify, Evaluate performance
    - U and M can be quick
    - P and I will take more time & communication with interviewer
    - R and E should be quick too

Linked List Demo

Given a linked list, remove the n-th node from the end of the list and return its head.

Example: Given linked lists 1->2->3->4->5, and n = 2. After removing the second node from the end, the linked list becomes 1->2->3->5.

My attempt: wasn't able to finish within the three minutes given in class.

- U: Some test cases—
  - 1>2>3>4>5, n = 1 => 1>2>3>4
  - 1>2>3, n = 3 => 2>3
  - 1, n = 1
  - 1>2, n = 3??
- M: dummy head, pointer bookkeeping, multi pass
- P:
  - First, reverse list. Then proceed with code below.
    - To reverse list: if head == nullptr, return nullptr.
    - Then, ???
  - If head == nullptr: return nullptr
  - Create dummy head(-1, head).
  - Create pointer node_before = dummy head.
  - For(int I = 0; I < n, i++):
    - If node_before->next != nullptr: Node_before = node_before -> next.
    - Else: return head.
  - If node_before->next != nullptr:
    - Pointer to_delete = node_before->next
    - node_before->next = node_before->next->next
    - delete to_delete
  - Return dummy->next.

- I: No code to reverse list, but here's how to delete the nth element from the start of the list.

```
Node* deleteNode(Node* head, int n)
{
	// case: null list
	if (head == nullptr)
	{
		return nullptr;
	}

	// create dummy head and pointer to iterate through list
	Node* dummy = new Node(-1, head);
	Node* node_before = dummy;

	// iterate to node before the one to be deleted
	for(int i = 0; i < n, i++)
	{
		node_before = node_before->next;
		if(node_before->next == nullptr)
		{
			return head;
		}
	}

	// delete the given node
	Node* to_delete = node_before->next;
	node_before->next = node_before->next->next;
	delete to_delete;

	return dummy->next;
}
```

Iris's demonstration:

- Comes up with her own test cases (e.g. given test case had odd # of nodes, she makes a list with even #
- Does her thinking in comment block above code
- What if n is invalid (negative, greater than the list length)? Do I have to consider those cases?
    - Consider more obvious edge cases first, more subtle ones later. Ask your interviewer about which ones you need to consider.
- Usually, you'll have to implement class yourself.
- Solicit feedback from interviewer in "plan" step.

'''

Given a linked list, remove the n-th node from the end of list and return its head.

Example:

Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5.

1. Understand

1 -> 2 -> 3 -> 4, n = 1

1 -> 2 -> 3

2. Match

Multipass? Possibly

Two Pointer? Possibly

3. Plan

move end pointer n steps

create the curr pointer

move both pointers until the end reaches list end

remove node

* curr.next = curr.next.next

```
     c              e
1 -> 2 -> 3 -> 4 -> 5 n = 3
```

'''

```
# Setting up Linked List class
class Node:
	def __init__(self, val, next=None):
		self.val = val
		self.next = next

	# Helper method to simplify debugging
	def __str__(self):
		return f"{self.val} > {self.next}"

def remove_nth_node(node, n):
	dummy = Node("dummy", node)
	# Move the end pointer N steps
	end = dummy
	for _ in range(n+1):
		end = end.next

	# Move both pointers until the end pointer hits the end
	curr = dummy
	while end:
		end = end.next
		curr = curr.next

	# Remove the node
	curr.next = curr.next.next
	return dummy.next

list1 = Node(1, Node(2, Node(3)))
print(list1)
```

Worked on practicing the UMPIRE method with two programming problems—see .cpp files in folder.

Before the next session

- Do Hackerrank assessment
- Do Week 2 warm up problems
- Read stacks and queues, heaps, hash tables guides
- Watch the Heaps Interview Question Walkthrough