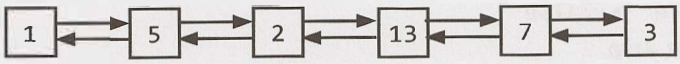
2

Linked Lists

A linked list is a data structure that represents a sequence of nodes. In a singly linked list, each node points to the next node in the linked list. A doubly linked list gives each node pointers to both the next node and the previous node.

The following diagram depicts a doubly linked list:



Unlike an array, a linked list does not provide constant time access to a particular "index" within the list. This means that if you'd like to find the Kth element in the list, you will need to iterate through K elements.

The benefit of a linked list is that you can add and remove items from the beginning of the list in constant time. For specific applications, this can be useful.

Creating a Linked List

The code below implements a very basic singly linked list.

```
1  class Node {
2    Node next = null;
3    int data;
4
5    public Node(int d) {
6      data = d;
7    }
8
9    void appendToTail(int d) {
10     Node end = new Node(d);
11     Node n = this;
12     while (n.next != null) {
13       n = n.next;
14     }
15     n.next = end;
16   }
17 }
```

In this implementation, we don't have a `LinkedList` data structure. We access the linked list through a reference to the head `Node` of the linked list. When you implement the linked list this way, you need to be a bit careful. What if multiple objects need a reference to the linked list, and then the head of the linked list changes? Some objects might still be pointing to the old head.

We could, if we chose, implement a `LinkedList` class that wraps the `Node` class. This would essentially just have a single member variable: the head `Node`. This would largely resolve the earlier issue.

Remember that when you're discussing a linked list in an interview, you must understand whether it is a singly linked list or a doubly linked list.

Deleting a Node from a Singly Linked List

Deleting a node from a linked list is fairly straightforward. Given a node n, we find the previous node `prev` and set `prev.next` equal to `n.next`. If the list is doubly linked, we must also update `n.next` to set `n.next.prev` equal to `n.prev`. The important things to remember are (1) to check for the null pointer and (2) to update the head or tail pointer as necessary.

Additionally, if you implement this code in C, C++ or another language that requires the developer to do memory management, you should consider if the removed node should be deallocated.

```
1  Node deleteNode(Node head, int d) {
2    Node n = head;
3
4    if (n.data == d) {
5      return head.next; / *moved head */
6    }
7
8    while (n.next != null) {
9      if (n.next.data == d) {
10       n.next = n.next.next;
11       return head; / *head didn't change */
12     }
13     n = n.next;
14   }
15   return head;
16 }
```

The "Runner" Technique

The "runner" (or second pointer) technique is used in many linked list problems. The runner technique means that you iterate through the linked list with two pointers simultaneously, with one ahead of the other. The "fast" node might be ahead by a fixed amount, or it might be hopping multiple nodes for each one node that the "slow" node iterates through.

For example, suppose you had a linked list $a_1->a_2->...->a_n->b_1->b_2->...->b_n$ and you wanted to rearrange it into $a_1->b_1->a_2->b_2->...->a_n->b_n$. You do not know the length of the linked list (but you do know that the length is an even number).

You could have one pointer p1 (the fast pointer) move every two elements for every one move that p2 makes. When p1 hits the end of the linked list, p2 will be at the midpoint. Then, move p1 back to the front and begin "weaving" the elements. On each iteration, p2 selects an element and inserts it after p1.

Recursive Problems

A number of linked list problems rely on recursion. If you're having trouble solving a linked list problem, you should explore if a recursive approach will work. We won't go into depth on recursion here, since a later chapter is devoted to it.

However, you should remember that recursive algorithms take at least O(n) space, where n is the depth of the recursive call. All recursive algorithms *can* be implemented iteratively, although they may be much more complex.

Interview Questions

2.1 **Remove Dups:** Write code to remove duplicates from an unsorted linked list.

FOLLOW UP

How would you solve this problem if a temporary buffer is not allowed?

Hints: #9, #40

pg 208

2.2 **Return Kth to Last:** Implement an algorithm to find the kth to last element of a singly linked list.

Hints: #8, #25, #41, #67, #126

pg 209

2.3 **Delete Middle Node:** Implement an algorithm to delete a node in the middle (i.e., any node but the first and last node, not necessarily the exact middle) of a singly linked list, given only access to that node.

EXAMPLE

Input: the node c from the linked list a->b->c->d->e->f

Result: nothing is returned, but the new linked list looks like a->b->d->e->f

Hints: #72

pg 211

2.4 **Partition:** Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x. If x is contained within the list, the values of x only need to be after the elements less than x (see below). The partition element x can appear anywhere in the "right partition"; it does not need to appear between the left and right partitions.

EXAMPLE

Input: 3 -> 5 -> 8 -> 5 -> 10 -> 2 -> 1 [partition = 5]

Output: 3 -> 1 -> 2 -> 10 -> 5 -> 5 -> 8

Hints: #3, #24

pg 212