**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.

public class Node

{

Node next = null;

int data;

public Node(int d)

{

data = d;

}

void appendToTail(int d)

{

Node end = new Node(d);

Node n = this;

while (n.next != null)

{

n = n.next;

}

n.next = end;

}

}

In this implementation, we don't have a Linked List 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 Linked List 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 ton. 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.

Node deleteNode(Node head, int d)

{

Node n = head;

if(n.data == d)

{

return head.next; // move head

}

while (n.next != null)

{

if(n.next.data == d)

{

n.next = n.next.next;

return head;// head didnt change

}

n = n.next;

}

return head;

}

**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 a1 ->a2 -> ••• ->an ->b1 ->b2 -> ••• ->bn and you wanted to rearrange it into a1 ->b1 ->a2 ->b2 -> ••• ->an ->bn. 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 pl (the fast pointer) move every two elements for every one move that p2 makes. When pl hits the end of the linked list, p2 will be at the midpoint. Then, move pl back to the front and begin "weaving" the elements. On each iteration, p2 selects an element and inserts it after pl.

**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.