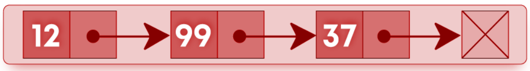
Linked Lists

Introduction to linked lists

A linked list is a collection of nodes in **non-contiguous** memory locations where every node contains some data and a pointer to the next node of the same data type. In other words, the node stores the address of the next node in the sequence. **A singly linked list allows traversal of data only in one way.**



Following are the terms used in Linked Lists :

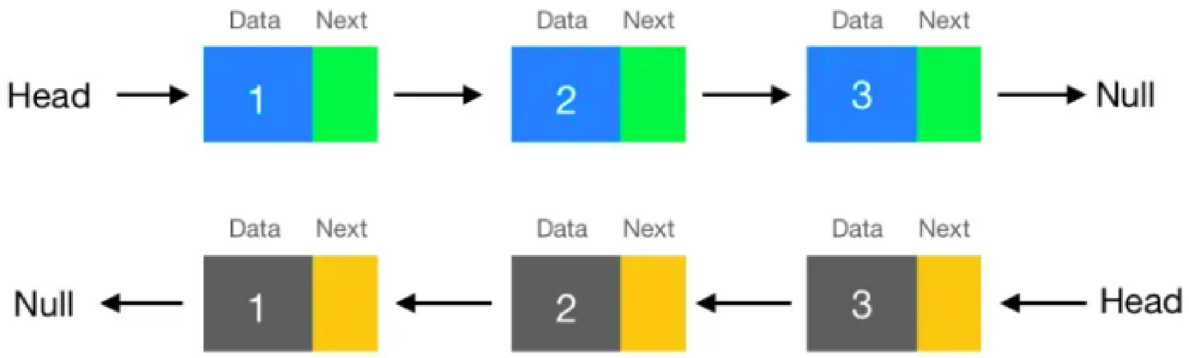
- **Node:** A node in a singly linked list contains two fields -
    - **Data** field which stores the data at the node
    - **A pointer** that contains the address of the next node in the sequence.
- **Head:** The first node in a linked list is called the head. The head is always used as a reference to traverse the list.
- **Tail:** The last node in a linked list is called the tail. It always contains a pointer to NULL (since the next node is NULL), denoting the end of a linked list.

Properties of Linked Lists

- A linked list is a **dynamic** data structure, which means the list can grow or shrink easily as the nodes are stored in memory in a non-contiguous fashion.
- The size of a linked list is limited to the size of memory, and the size need not be declared in advance.
- **Note:** We must never lose the address of the head pointer as it references the starting address of the linked list and if lost, would lead to loss of the list.

Linked List Reversal

We first look at the standard algorithm for reversing a Linked list. Given a Linked List we want to reverse it and then return the pointer to the first node of the reversed list.



Recursive Approach

Algorithm:

1. We divide the linked list of **N** nodes into two parts. i.e head and rest of the Linked List with (**N-1)** nodes.
2. Now recursively reverse the (**N**-1) nodes of Linked List and return the head of this part i.e **rest.** After the reversal, the next node of the head will be the last node of the reversed Linked List and the head will be pointing to this node.
3. But for the complete reversal of the Linked List, the head should be the last node. So, we do the following:
    1. head.next.next = head, where head.next is the last node of the reverse Linked List.
    2. head.next = NULL
4. Return the head pointer of the reversed Linked List i.e. return **rest.**

```
function reverseLL(head)

        //  Base condition
        if head is null or head.next is null
                //  Return the last node.
                return head

        //  Reverse the rest of Linked List
        rest = reverseLinkedList(head->next)

        //  Changing the reference of next node next to itself
        head->next->next = head

        //  Assign current node next to NULL.
        head->next = NULL

        //  Return the reverse Linked List.
        return rest
```

Time Complexity: O(N), where N is the number of nodes in the Linked List. In the worst case, we are traversing the whole Linked List O(N) using recursion, Hence, the overall complexity will be O(N).

Space Complexity: O(N), where N is the total number of nodes in the Linked List. In the worst case, O(N) extra space is required for the recursion stack.

Iterative Approach

Algorithm:

1. Initially, we will take three-pointers, **current** that points to the head of Linked List, **prev,** and **nextNode**, both pointing to null.
2. Then we will iterate over the linked list until the **current** is not equal to NULL and do the following update in every step of the iteration:
    1. nextNode = current.next
    2. current.next = prev
    3. prev = current
    4. current = nextNode
3. Now return the **prev** pointer which is now the head of reverse Linked List.

```
function reverseLL(head)

        //  Creating node for remembering the previous node in the Linked List.
        prev = NULL

        //  Creating temporary node.
        current = head

        while current is not null
                nextNode = current->next
                current->next = prev
                prev = current
                current = nextNode
```

```
//   Return reverse Linked List.
return prev
```

Time Complexity: O(N), where N is the number of nodes in the linked list.
In the worst-case, we are iterating the whole linked list O(N). Hence, the overall complexity will be O(N).

Space Complexity: O(1), as we are using constant extra space.
Now we move to an application-based problem that uses the concept of reversal in linked lists.