Revision - Linked List

Introduction

A linked list is a data structure that consists of a sequence of elements called nodes, where each node contains a value and a reference to the next node in the sequence. Unlike arrays, the nodes are not stored in contiguous memory locations, but rather scattered throughout the computer's memory.

Example Representation

```
struct node
{
    int value;
    struct node *next;
};
```

Searching in Linked List

To search for an element in a linked list, we need to traverse the list from the beginning until we find the node containing the desired element or reach the end of the list. Here's the algorithm:

- Set the current node to the head of the list.
- While the current node is not NULL and the value of the current node is not equal to the desired value, move to the next node in the list.
- If the current node is NULL, the element is not in the list. Otherwise, the element is in the list and its node is the current node.

Code

```
function search(head, value):
    current = head
    while current is not null and current.value is not equal to value:
        current = current.next
    return current
```

The time complexity in the best case is O(1), in the worst case it is O(N) and in the average case also it is O(N). Here N indicates the size of the LL. The best case is obtained when the value of the head of the LL is equal to the target value. The worst case is when the desired node is absent from the LL.

Insertion in LL

If the position is 0, we insert the new node at the beginning of the list by setting the next pointer of the new node to the current head of the list, and then setting the head pointer to point to the new node.
If the position is greater than 0, we iterate through the list with a current pointer until we reach the node just before the position where the new node needs to be inserted. If the current pointer becomes null before reaching the desired position, we return without making any changes to the list. Once we find the correct position, we insert the new node by setting its next pointer to point to the node that follows current, and then setting current’s next pointer to point to the new node.
Finally, we return the head of the list.

Code

```
function insertAtPosition(head, value, position):
    new_node = new Node(value)
    if position is equal to 0:
        new_node.next = head
        head = new_node
    else:
        current = head
        for i from 0 to position - 2:
            if current is null:
                return // position is out of range
            current = current.next
        new_node.next = current.next
        current.next = new_node
    return head
```

The time complexity of inserting a node at a specific position in a linked list depends on the position of the node. In the worst case scenario, where the node needs to be inserted at the end of the list, the time complexity is O(n), where n is the number of nodes in the list. In the best case scenario, where the node needs to be inserted at the beginning of the list, the time complexity is O(1).

Deletion in LL

We traverse the list with two pointers - current and previous - until we find a node that has a value equal to target value, or until it reaches the end of the list.
If the target value is found, we remove the node from the list by updating the next pointer of the previous node to point to the node that follows the current node (current.next). If the previous pointer is null, that means the node being deleted is the head of the list, so the head pointer is updated to point to the next node (current.next).

Code

```
function deleteNode(head, value):
    current = head
    previous = null
    while current is not null and current.value is not equal to value:
        previous = current
        current = current.next
    if current is null:
        return // element not found
    if previous is null:
        head = current.next
    else:
        previous.next = current.next
    delete current
```

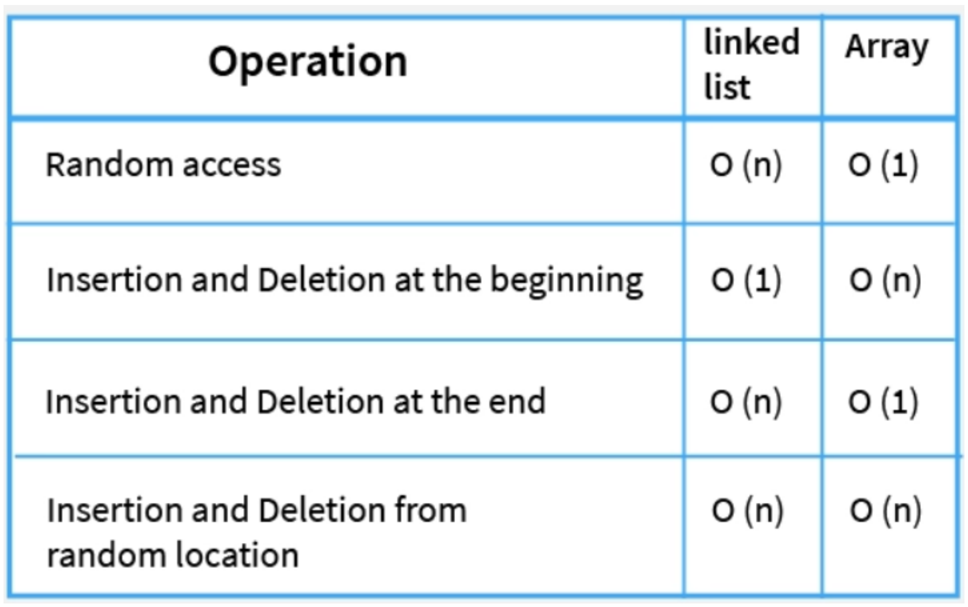
Reversing a LL

To reverse a linked list, we need to traverse the list while changing the direction of the links between nodes. We can initialize three pointers - previous, current, and next, and then traverse the list with the current pointer, updating the links between nodes as we go.
For each node, we set next to the node that follows current, then update the next pointer of current to point to previous instead of next, and finally update previous to current and current to next.
We repeat these steps until the current reaches the end of the list (null). We then return previous as the new head of the reversed list.

Code

```
function reverse(head):
    previous = null
    current = head
    while current is not null:
        next = current.next
        current.next = previous
        previous = current
        current = next
    return previous
```

Linked Lists vs Arrays



Operation	linked list	Array
Random access	O (n)	O (1)
Insertion and Deletion at the beginning	O (1)	O (n)
Insertion and Deletion at the end	O (n)	O (1)
Insertion and Deletion from random location	O (n)	O (n)

Removing Loop in a LL

A loop in a linked list occurs when one or more nodes in the list point to a node that has already been visited, resulting in an infinite loop. Removing a loop from a linked list involves detecting the loop and then breaking the loop by redirecting the last node in the loop to point to null.

Example

```
Input 1:

1 -> 2
^    |
| - -

Output 1:

1 -> 2 -> NULL


Input 2:

3 -> 2 -> 4 -> 5 -> 6
          ^         |
          |         |
          - - - - - -


Output 2:

 3 -> 2 -> 4 -> 5 -> 6 -> NULL
```

Code

```
function detectAndRemoveLoop(head):
    slow = head
    fast = head
    while slow and fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            break
    if slow != fast:
        return // no loop found
    slow = head
    while slow.next != fast.next:
        slow = slow.next
        fast = fast.next
    fast.next = null
```

The algorithm works as follows:

- Initialize two pointers, slow and fast, to the head of the list.
- Traverse the list with the slow and fast pointers, with slow moving one node at a time and fast moving two nodes at a time.
- If slow and fast meet at a node (i.e., slow and fast point to the same node), then there is a loop in the list. Otherwise, there is no loop, and the function returns.
- To remove the loop, set slow back to the head of the list, and move both slow and fast one node at a time until they meet again at the last node in the loop.
- Set the next pointer of the last node in the loop (pointed to by fast) to null, thus breaking the loop.
- Return the head of the list.

The time complexity of this algorithm is O(n), where n is the number of nodes in the list.