Linked Lists

Detailed Notes

Hiike

Introduction to Linked Lists

A linked list is a linear data structure where each element (often called a node) contains a data part and a reference (or link) to the next node in the sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations, allowing for efficient insertion and deletion operations.

Types of Linked Lists

1. **Singly Linked List:** Each node contains a single link to the next node.
2. **Doubly Linked List:** Each node contains two links, one to the next node and another to the previous node.
3. **Circular Linked List:** The last node points back to the first node, forming a circle.

Singly Linked List

Definition and Structure:

In a singly linked list, each node has two parts: data and a pointer to the next node.

Pseudocode and Time Complexity for Singly Linked List

Insertion:

1. **At the Beginning:**

```
function insertAtBeginning(head, newData):
    newNode = Node(newData)
    newNode.next = head
    head = newNode
```

Time Complexity: O(1)

2. **At the End:**

```
function insertAtEnd(head, newData):
    newNode = Node(newData)
    if head is NULL:
        head = newNode
        return
    temp = head
    while temp.next is not NULL:
        temp = temp.next
    temp.next = newNode
```

Time Complexity: O(n)

3. **After a Given Node:**

```
function insertAfter(prevNode, newData):
    if prevNode is NULL:
        print "Previous node cannot be NULL"
        return
    newNode = Node(newData)
    newNode.next = prevNode.next
    prevNode.next = newNode
```

Time Complexity: O(1)

Deletion:

1. **Deleting a Node:**

```
function deleteNode(head, key):
    temp = head
    prev = NULL

    if temp is not NULL and temp.data == key:
        head = temp.next
        delete temp
        return
    while temp is not NULL and temp.data != key:
        prev = temp
        temp = temp.next

    if temp is NULL:
        return

    prev.next = temp.next
    delete temp
```

Time Complexity: O(n)

2. **Deleting at a Given Position:**

```
function deleteAtPosition(head, position):
    if head is NULL:
        return

    temp = head

    if position == 0:
        head = temp.next
        delete temp
        return

    for i = 0 to position-1:
        temp = temp.next

    if temp is NULL or temp.next is NULL:
        return

    next = temp.next.next
    delete temp.next
    temp.next = next
```

Time Complexity: O(n)

Traversal:

1. **Iterative Approach:**

```
function printList(node):
    while node is not NULL:
        print node.data
        node = node.next
```

Time Complexity: O(n)

2. **Recursive Approach:**

```
function printListRecursively(node):
    if node is NULL:
        return
    print node.data
    printListRecursively(node.next)
```

Time Complexity: O(n)

Doubly Linked List

Definition and Structure:

In a doubly linked list, each node has three parts: data, a pointer to the next node, and a pointer to the previous node.

Pseudocode and Time Complexity for Doubly Linked List

Insertion:

1. **At the Beginning:**

```
function insertAtBeginning(head, newData):
    newNode = Node(newData)
    newNode.next = head
    if head is not NULL:
        head.prev = newNode
    head = newNode
```

Time Complexity: O(1)

2. **At the End:**

```
function insertAtEnd(head, newData):
    newNode = Node(newData)
    temp = head

    if head is NULL:
        head = newNode
        return

    while temp.next is not NULL:
        temp = temp.next

    temp.next = newNode
    newNode.prev = temp
```

Time Complexity: O(n)

3. **After a Given Node:**

```
function insertAfter(prevNode, newData):
    if prevNode is NULL:
        print "Previous node cannot be NULL"
        return

    newNode = Node(newData)
    newNode.next = prevNode.next
    newNode.prev = prevNode
    prevNode.next = newNode

    if newNode.next is not NULL:
        newNode.next.prev = newNode
```

Time Complexity: O(1)

Deletion:

1. Deleting a Node:

```
function deleteNode(head, del):
    if head == NULL or del == NULL:
        return

    if head == del:
        head = del.next

    if del.next != NULL:
        del.next.prev = del.prev

    if del.prev != NULL:
        del.prev.next = del.next

    delete del
```

Time Complexity: O(1)

Traversal:

1. Forward Traversal:

```
function printList(node):
    Node last
    while node is not NULL:
        print node.data
        last = node
        node = node.next
```

Time Complexity: O(n)

2. Backward Traversal:

```
function printListReverse(node):
    Node last
    while node is not NULL:
        last = node
        node = node.next
    while last is not NULL:
        print last.data
        last = last.prev
```

Time Complexity: O(n)

Circular Linked List

Definition and Structure:

In a circular linked list, the last node points back to the first node, forming a circle.

Pseudocode and Time Complexity for Circular Linked List

Insertion:

1. **At the Beginning:**

```
function insertAtBeginning(head, newData):
    newNode = Node(newData)
    temp = head

    if head == NULL:
        head = newNode
        newNode.next = head
        return

    while temp.next != head:
        temp = temp.next

    temp.next = newNode
    newNode.next = head
    head = newNode
```

Time Complexity: O(n)

2. **At the End:**

```
function insertAtEnd(head, newData):
    newNode = Node(newData)
    temp = head

    if head == NULL:
        head = newNode
        newNode.next = head
        return

    while temp.next != head:
        temp = temp.next

    temp.next = newNode
    newNode.next = head
```

Time Complexity: O(n)

Deletion:

1. **Deleting a Node:**

```
function deleteNode(head, key):
    if head == NULL:
        return

    Node temp = head, prev = NULL

    while temp.data != key:
        if temp.next == head:
            print "Node not found"
            return
        prev = temp
        temp = temp.next

    if temp.next == head and prev == NULL:
        head = NULL
        delete temp
        return
    if temp == head:
        prev = head
        while prev.next != head:
            prev = prev.next
        head = temp.next
        prev.next = head
        delete temp
    else if temp.next == head:
        prev.next = head
        delete temp
    else:
        prev.next = temp.next
        delete temp
```

Time Complexity: O(n)

Traversal:

```
function printList(head):
    Node temp = head
    if head != NULL:
        do:
            print temp.data
            temp = temp.next
        while temp != head
```

Time Complexity: O(n)

Key Points

1. **Linked Lists** are dynamic in size and efficient for insertions and deletions.
2. **Singly Linked List** has nodes with data and a pointer to the next node.
3. **Doubly Linked List** has nodes with data, a pointer to the next node, and a pointer to the previous node.
4. **Circular Linked List** has nodes where the last node points back to the first node, forming a circle.

5. **Operations** like insertion, deletion, and traversal can be done efficiently in linked lists.

Here are ten LeetCode coding questions that will help you test your knowledge on various linked list concepts and operations:

Reverse Linked List

- Problem: Reverse a singly linked list.
- Link: Reverse Linked List

Merge Two Sorted Lists

- Problem: Merge two sorted linked lists and return it as a new sorted list.
- Link: Merge Two Sorted Lists

Remove Nth Node From End of List

- Problem: Remove the n-th node from the end of list and return its head.
- Link: Remove Nth Node From End of List

Linked List Cycle

- Problem: Determine if a linked list has a cycle in it.
- Link: Linked List Cycle

Palindrome Linked List

- Problem: Check if a singly linked list is a palindrome.
- Link: Palindrome Linked List

Add Two Numbers

- Problem: Add two numbers represented by linked lists.
- Link: Add Two Numbers

Intersection of Two Linked Lists

- Problem: Find the node at which the intersection of two singly linked lists begins.

- Link: Intersection of Two Linked Lists

Flatten a Multilevel Doubly Linked List

- Problem: Flatten a multilevel doubly linked list.
- Link: Flatten a Multilevel Doubly Linked List

Rotate List

- Problem: Rotate the list to the right by k places, where k is non-negative.
- Link: Rotate List

Remove Duplicates from Sorted List

- Problem: Remove duplicates from a sorted linked list.
- Link: Remove Duplicates from Sorted List