DATA STRUCTURE ALGORITHMS AND APPLICATIONS (CT- 159)

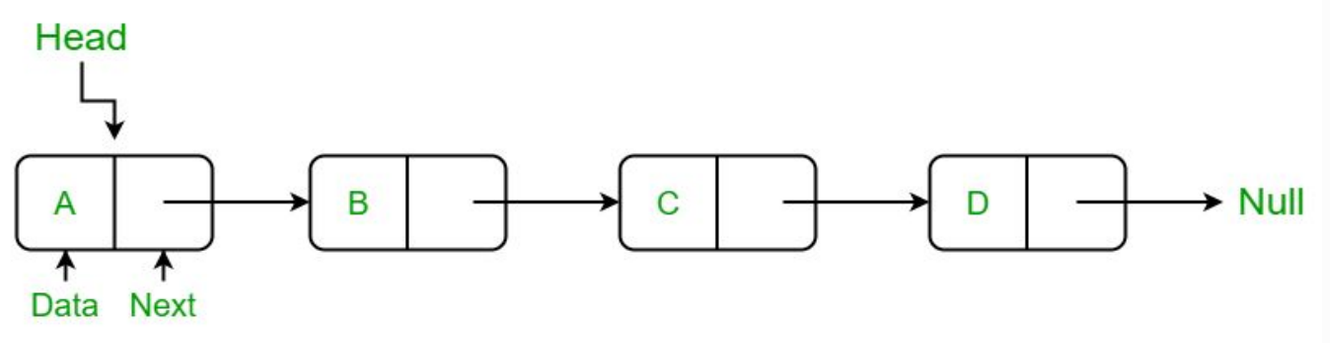
Lecture – 6, 7
Linked List, Doubly Linked List, Circular Linked List

Instructor: Engr. Nasr Kamal

Department of Computer Science and Information Technology

What is Linked List

- A **linked list** is a linear data structure where elements (called **nodes**) are stored in a sequence, but instead of using a contiguous block of memory (like an array), each node points to the next node in the sequence.
- Each node typically contains two things:
  Data: The value stored in the node.
  Pointer: A reference to the next node in the list.
- Unlike arrays, linked lists allow for efficient insertion and deletion of elements, as they don't require shifting elements around.
- However, searching for an element in a linked list is generally slower since you have to traverse the list node by node.



Pseudo Code for Creating a Linked List

```
struct Node {
    int data;
    Node* next;
};
```

CREATING NODE

```
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;  // No next
node initially
    return newNode;
}
```

Basic Operations in Linked List

- **Insertion**: Adding an element to the list. Efficient when adding nodes at the beginning or end of the list.

INSERT AT BEGINNING

```
Function InsertAtBeginning(Head, Data)
Create new Node Node.Data = Data
Node.Prev = NULL Node.Next = Head
If Head != NULL Head.Prev = Node
End If
Head = Node
End Function
```

Basic Operations in Linked List

- **Insertion**: Adding an element to the list. Efficient when adding nodes at the beginning or end of the list.

INSERT IN THE MIDDLE

```
Function InsertInMiddle(Head, Data, Position)
    Create new Node
    Node.Data = Data

    If Position == 1
        InsertAtBeginning(Head, Data)
    Else
        Temp = Head
        Count = 1
        While Count < Position - 1 AND Temp != NULL
            Temp = Temp.Next
            Count = Count + 1
        End While
```

```
If Temp == NULL
            Print "Position out of bounds"
        Else
            Node.Next = Temp.Next
            Node.Prev = Temp
            If Temp.Next != NULL
                Temp.Next.Prev = Node
            End If
            Temp.Next = Node
        End If
    End If
End Function
```

Basic Operations in Linked List

- **Insertion**: Adding an element to the list. Efficient when adding nodes at the beginning or end of the list.

INSERT AT THE END

```
Function InsertAtEnd(Head, Data)
    Create new Node
    Node.Data = Data
    Node.Next = NULL

    If Head == NULL
        Node.Prev = NULL
        Head = Node
Else
        Temp = Head
        While Temp.Next != NULL
            Temp = Temp.Next
        End While
        Temp.Next = Node
        Node.Prev = Temp
    End If
End Function
```

Basic Operations in Linked List

- **Deletion**: Removing an element from the list. Deleting a node is fast if the position is known, but it may require traversal for arbitrary positions.

DELETE AT THE BEGINNING

```
Function DeleteAtBeginning(Head)
    If Head == NULL
        Print "List is empty"
    Else
        Temp = Head
        Head = Head.Next
```

```
If Head != NULL
            Head.Prev = NULL
        End If

        Delete Temp
    End If
End Function
```

Basic Operations in Linked List

- **Deletion**: Removing an element from the list. Deleting a node is fast if the position is known, but it may require traversal for arbitrary positions.

DELETE AT THE END

```
Function DeleteAtEnd(Head)
    If Head == NULL
        Print "List is empty"
    Else If Head.Next == NULL
        Delete Head
        Head = NULL
Else
        Temp = Head
        While Temp.Next != NULL
            Temp = Temp.Next
        End While

        Temp.Prev.Next = NULL
        Delete Temp
    End If
End Function
```

Basic Operations in Linked List

- **Searching**: Finding an element in the list. Searching in a linked list requires linear time (O(n)) as you must traverse node by node

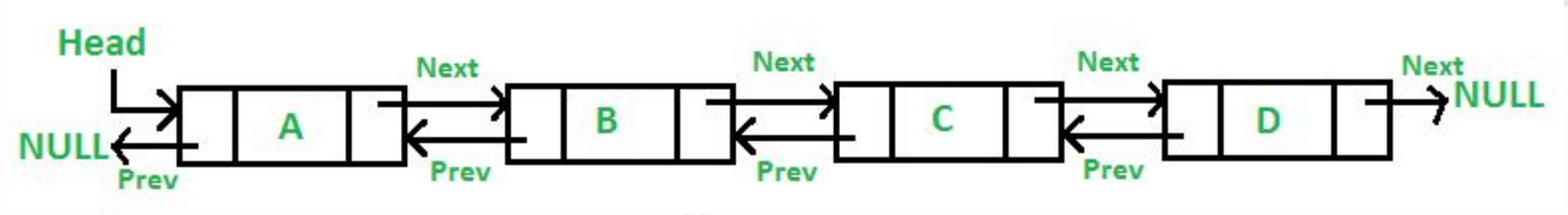
```
SEARCHING A NODE/ TRAVERSING
Function Search(Head, Data)
    Temp = Head
    Position = 1

    While Temp != NULL
        If Temp.Data == Data
            Print "Node found at position", Position
            Return
        End If
Temp = Temp.Next
        Position = Position + 1
    End While

    Print "Node not found"
End Function
```

Doubly Linked List

- A doubly linked list is similar to a singly linked list, but each node contains an additional pointer that points to the previous node.
- This allows for traversal in both directions: forward and backward.
- **Advantages:**
- Easier to implement deletion of a node, as you can directly access the previous node.
- You can traverse the list in both directions.



Pseudo Code for Doubly Linked List

```
struct Node {
    int data;
    Node* next;
    Node* prev;
};
```

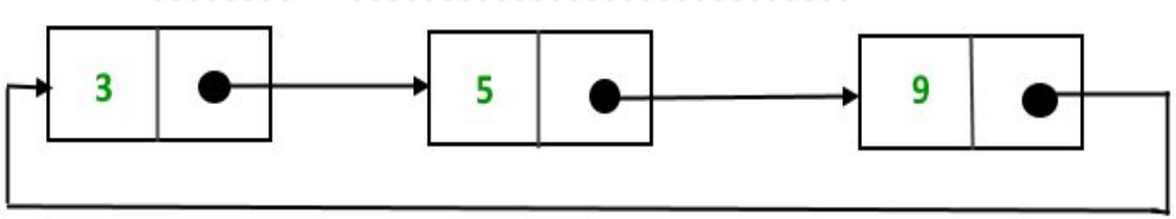
```
// Function to create a new node
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;
    newNode->prev = nullptr;  // Points to the previous
node
    return newNode;
}
```

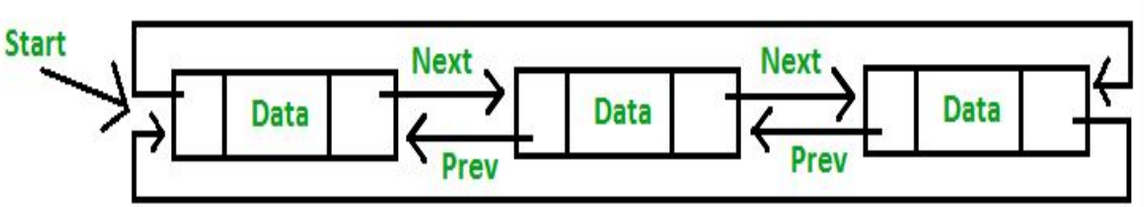
Circular Linked List

- In a circular linked list, the last node points back to the head, forming a circular structure.

There are two types:

- **Singly Circular Linked List**: The last node's next pointer points to the head.
- **Doubly Circular Linked List**: The last node's next pointer points to the head, and the head's previous pointer points to the last node.





Pseudo Code for Circular Linked List

```
// Node structure
struct Node {
    int data;
    Node* next;
};

// Function to create a new node
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;
    return newNode;
}
```

Comparison of Linked Lists

- **Singly Linked List:** Simple and easy to implement. But limited traversal (only forward).
- **Doubly Linked List**: More versatile, allowing forward and backward traversal.
  - Slightly more memory-intensive due to the additional pointer.
- **Circular Linked List:** Useful in applications where you need cyclic traversal (e.g., in round-robin scheduling).

Arrays VS Linked List

Feature	Linked List	Array
Size	Dynamic	Fixed (unless dynamically resized)
Memory Utilization	Efficient for variable sizes, but overhead for pointers	Efficient for fixed sizes, but wastes space if underused
Insertion/Deletion	Fast for middle and end operations	Slow due to shifting elements
Access Time	Sequential, O(n)	Random access, O(1)
Memory Allocation	Scattered memory, more flexible	Contiguous memory allocation