Lesson Plan

Linked List-1

Today’s Checklist

1. Limitations of Arrays
2. Introduction to Linked List
3. Implementing Linked List
4. Displaying
5. Insert in Linked List
6. Limitation
7. Delete in Linked List

Limitations of Arrays

1. **Fixed Size:** Most arrays have a fixed size, meaning you need to know the number of elements in advance. This can be a limitation when the size of the data is dynamic and unknown beforehand.

2. **Contiguous Memory Allocation:** Elements in an array are stored in contiguous memory locations. This can lead to fragmentation and might make it difficult to find a large enough block of memory for the array.

3. **Inefficient Insertions and Deletions:** Inserting or deleting elements in the middle of an array requires shifting all subsequent elements, which can be inefficient. The time complexity for these operations is O(n), where n is the number of elements.

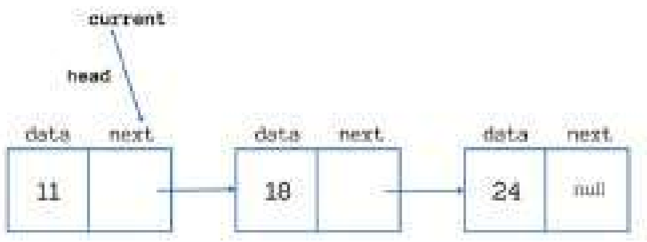
4. **Wastage of Memory:** If you allocate more space than needed for an array, you may end up wasting memory. This is particularly problematic when the array size is predetermined to accommodate the worst-case scenario.

5. **Homogeneous Data Types:** Arrays typically store elements of the same data type. This can be limiting when you need to store elements of different types.

6. **Memory Fragmentation:** The contiguous memory allocation can lead to memory fragmentation, making it challenging to allocate a large contiguous block of memory for a new array.

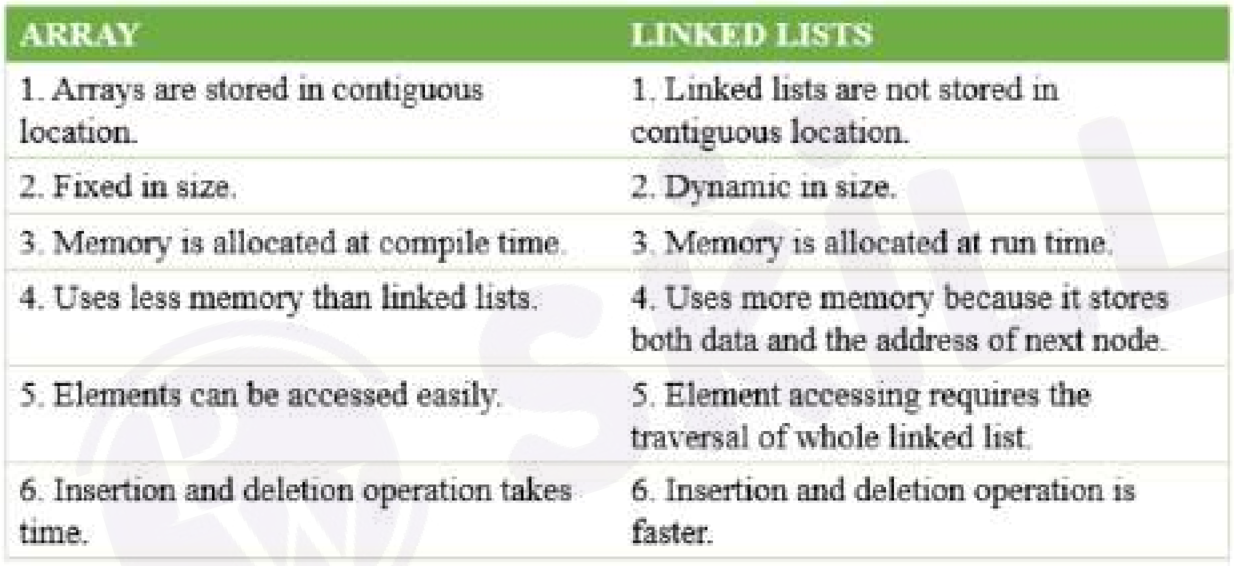
Introduction to Linked List

It is basically chains of nodes, each node contains information such as data and a pointer to the next node in the chain. In the linked list there is a head pointer, which points to the first element of the linked list, and if the list is empty then it simply points to null or nothing.



Why linked list data structure needed?

1. **Dynamic Data structure:** The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
2. **Ease of Insertion/Deletion:** The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
3. **Efficient Memory Utilization:** As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
4. **Implementation:** Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.



ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Implementing Linked List

A linked list can be implemented in various ways, but the basic structure involves nodes, where each node contains data and a reference (or link) to the next node in the sequence.

A step-by-step explanation of how to implement a simple singly linked list:

1. **Node Class:** Create a class for the linked list node with data and a pointer to the next node.

```
public class Node {
    public int data;
    public Node next;

    public Node(int value) {
        this.data = value;
        this.next = null;
    }
}
```

2. **LinkedList Class:** Create a class to represent the linked list. Include a pointer to the head of the list.

```
public class LinkedList {
    private Node head;

    public LinkedList() {
        this.head = null;
    }
}
```

3. **Add Nodes:**

- Implement a method to add nodes to the linked list.
- If the list is empty, create a new node and set it as the head.
- Otherwise, traverse to the end of the list and add a new node.

```
public void addNode(int value) {
    Node newNode = new Node(value);
    if (head == null) {
        head = newNode;
    } else {
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
}
```

Displaying:

Once, we have created the linked list, we would like to see the elements inside it. Displaying a linked list involves iterating through its nodes and printing their data. This can also be done recursively as shown in the code below.

Code:

```java
class Node {
    public int data;
    public Node next;

    public Node(int value) {
        this.data = value;
        this.next = null;
    }
}

public class Main {
    public static void displayLinkedListIterative(Node head) {
        while (head != null) {
            System.out.print(head.data + " ");
            head = head.next;
        }
        System.out.println();
    }

    public static void displayLinkedListRecursive(Node head) {
        if (head == null)
            return;

        System.out.print(head.data + " ");
        displayLinkedListRecursive(head.next);
    }

    public static void main(String[] args) {
        Node head = new Node(1);
        head.next = new Node(2);
        head.next.next = new Node(3);
        head.next.next.next = new Node(4);

        System.out.print("Iterative Display: ");
        displayLinkedListIterative(head);

        System.out.print("Recursive Display: ");
        displayLinkedListRecursive(head);
        System.out.println();
    }
}
```

Explanation:

Iterative Display:

1. Initialize a pointer to the head of the linked list.
2. Use a while loop to traverse the list.
3. Print the data of each node and move the pointer to the next node.
4. Repeat until the end of the list is reached.

Recursive Display:

1. Base case: If the current node is null, return.
2. Print the data of the current node.
3. Make a recursive call with the next node.
4. The recursion unwinds, printing nodes in sequential order.
5. Base case ensures termination when the end of the list is reached.

Iterative Display:

Time Complexity: O(n) - where 'n' is the number of nodes in the linked list. The algorithm iterates through each node once.
Space Complexity: O(1) - uses a constant amount of extra space, regardless of the size of the linked list.

Recursive Display:

Time Complexity: O(n) - where 'n' is the number of nodes in the linked list. Similar to the iterative approach, each node is visited once, but the recursive call stack contributes to the time complexity.
Space Complexity: O(n) - due to the recursive call stack. The maximum depth of the recursion is 'n', corresponding to the length of the linked list.

MCQ: What will this function do?

```
public void display(Node head) {
    if (head == null)
        return;
    display(head.next);
    System.out.print(head.data + " ");
}
```

A. Print all the elements of the linked list.

B. Print all the elements except last one.

C. Print alternate nodes of linked list

D. Print all the nodes in reverse order

Ans: Print all the nodes in reverse order.

Explanation:

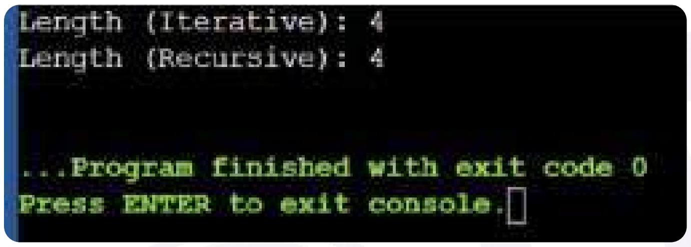
1. The given function is a recursive function that traverses the linked list using a recursive call (display(head->next)) before printing the value of the current node (cout<<head.val<<" ").
2. The base case (if(head == null) return;) ensures that the recursion stops when the end of the linked list is reached.
3. As the recursion unfolds, the values of the nodes are printed in reverse order, starting from the last node and moving towards the head of the linked list.

Length of Linked List

Code:

```
public class Node {
    public int data;
    public Node next;

    public Node(int value) {
        this.data = value;
        this.next = null;
    }
}

public class Main {
    public static int findLengthIterative(Node head) {
        int length = 0;
        while (head != null) {
            length++;
            head = head.next;
        }
        return length;
    }

    public static int findLengthRecursive(Node head) {
        if (head == null)
```

```
            return 0;
        return 1 + findLengthRecursive(head.next);
    }

    public static void main(String[] args) {
        Node head = new Node(1);
        head.next = new Node(2);
        head.next.next = new Node(3);
        head.next.next.next = new Node(4);

        System.out.println("Length (Iterative): " +
findLengthIterative(head));
        System.out.println("Length (Recursive): " +
findLengthRecursive(head));
    }
}
```



```
Length (Iterative): 4
Length (Recursive): 4


...Program finished with exit code 0
Press ENTER to exit console.
```

Iterative Method:
Time Complexity: O(n) - where 'n' is the number of nodes in the linked list. In the worst case, it needs to traverse all nodes once.

Space Complexity: O(1) - uses a constant amount of extra space.

Recursive Method:
Time Complexity: O(n) - where 'n' is the number of nodes in the linked list. Similar to the iterative approach, it needs to visit each node once.

Space Complexity: O(n) - due to the recursive call stack. The maximum depth of the recursion is 'n', corresponding to the length of the linked list.

Insert in Linked List

The insertion operation can be performed in three ways. They are as follows...

1. Inserting At the Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Code:

```
public class Node {
    public int data;
    public Node next;

    public Node(int value) {
        this.data = value;
        this.next = null;
    }
}

public class Main {
    // Insert at the Beginning of the list
    public static Node insertAtBeginning(Node head, int value) {
        Node newNode = new Node(value);
        newNode.next = head;
        return newNode;
    }

    // Insert at End of the list
    public static Node insertAtEnd(Node head, int value) {
        Node newNode = new Node(value);
        if (head == null) {
            return newNode;
        }
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
        return head;
    }

    // Insert at Specific location in the list
    public static Node insertAtLocation(Node head, int value, int 
position) {
        Node newNode = new Node(value);
        if (position == 1) {
            newNode.next = head;
            return newNode;
        }
        Node current = head;
        for (int i = 1; i < position - 1 && current != null; ++i) 
{
            current = current.next;
        }
        if (current == null) {
            System.out.println("Invalid position.");
            return head;
        }
        newNode.next = current.next;
        current.next = newNode;
        return head;
    }

    // Display the linked list
    public static void displayLinkedList(Node head) {
        while (head ≠ null) {
            System.out.print(head.data + " ");
            head = head.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Node head = null;

        // Insert at the Beginning
        head = insertAtBeginning(head, 1);
        displayLinkedList(head);

        // Insert at the End
        head = insertAtEnd(head, 3);
        displayLinkedList(head);

        // Insert at Specific location
        head = insertAtLocation(head, 2, 2);
        displayLinkedList(head);
    }
}
```

Insert at the Beginning:

1. **Create a New Node:** Allocate memory for a new node and set its data.
2. **Link to Current Head:** Set the next pointer of the new node to the current head.
3. **Update Head:** Set the new node as the new head of the linked list.

Insert at the End:

1. **Create a New Node: Allocate memory for a new node and set its data.**
2. **Traverse to the Last Node: Iterate through the linked list until the last node is reached.**
3. **Link to Last Node: Set the next pointer of the last node to the new node.**

Insert at Specific Location:

1. **Create a New Node:** Allocate memory for a new node and set its data.
2. **Handle Special Case (Insert at the Beginning):** If the position is 1, link the new node to the current head and update the head.
3. **Traverse to the Previous Node:** Iterate through the list to the node preceding the desired position.
4. **Link the New Node:** Set the next pointer of the new node to the next node of the previous node, and set the next pointer of the previous node to the new node.

Time and Space Complexity:

1. **Insert at the Beginning:**

   Time Complexity: O(1)
   Space Complexity: O(1)

2. **Insert at the End:**

   Time Complexity: O(n) - in the worst case
   Space Complexity: O(1)

3. **Insert at Specific Location:**

   Time Complexity: O(position) - in the worst case
   Space Complexity: O(1)

Delete in Linked List

The deletion operation can be performed in three ways. They are as follows:

1. Deleting from the Beginning of the list
2. Deleting from the End of the list
3. Deleting a Specific Node

Code:

```
public class Node {
    public int data;
    public Node next;

    public Node(int value) {
        this.data = value;
        this.next = null;
    }
}
```

```
public class Main {
    // Delete from the Beginning of the list
    public static Node deleteFromBeginning(Node head) {
        if (head == null) {
            System.out.println("List is empty. Cannot delete.");
            return null;
        }
        Node newHead = head.next;
        head.next = null; // Disconnect the deleted node from the
rest of the list
        delete head;
        return newHead;
    }

    // Delete from the End of the list
    public static Node deleteFromEnd(Node head) {
        if (head == null) {
            System.out.println("List is empty. Cannot delete.");
            return null;
        }
        if (head.next == null) {
            delete head;
            return null;
        }
        Node current = head;
        while (current.next.next != null) {
            current = current.next;
        }
        delete current.next;
        current.next = null;
        return head;
    }

    // Delete a Specific Node
    public static Node deleteNode(Node head, int value) {
        if (head == null) {
            System.out.println("List is empty. Cannot delete.");
            return null;
        }
        if (head.data == value) {
            Node newHead = head.next;
            head.next = null; // Disconnect the deleted node from
the rest of the list
            delete head;
            return newHead;
        }
        Node current = head;
        while (current.next != null && current.next.data !=
value) {
```

```
            current = current.next;
        }
        if (current.next == null) {
            System.out.println("Node with value " + value + " not 
found.");
            return head;
        }
        Node temp = current.next;
        current.next = current.next.next;
        temp.next = null; // Disconnect the deleted node from the 
rest of the list
        delete temp;
        return head;
    }

    // Display the linked list
    public static void displayLinkedList(Node head) {
        while (head != null) {
            System.out.print(head.data + " ");
            head = head.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Node head = null;

        // Delete from the Beginning
        head = deleteFromBeginning(head);
        displayLinkedList(head);

        // Delete from the End
        head = deleteFromEnd(head);
        displayLinkedList(head);

        // Delete a Specific Node
        head = deleteNode(head, 2);
        displayLinkedList(head);
    }
}
```

Deletion operations in Linked List

```
public class Node {
    public int data;
    public Node next;
    public Node(int value) {
        this.data = value;
        this.next = null;
    }
}

public class Main {
    // Delete from the Beginning of the list
    public static Node deleteFromBeginning(Node head) {
        if (head == null) {
            System.out.println("List is empty. Cannot delete.");
            return null;
        }
        Node newHead = head.next;
        head.next = null; // Disconnect the deleted node from the
rest of the list
        delete head;
        return newHead;
    }

    // Delete from the End of the list
    public static Node deleteFromEnd(Node head) {
        if (head == null) {
            System.out.println("List is empty. Cannot delete.");
            return null;
        }
        if (head.next == null) {
            delete head;
            return null;
        }
        Node current = head;
        while (current.next.next != null) {
            current = current.next;
        }
        delete current.next;
        current.next = null;
        return head;
    }

    // Delete a Specific Node
    public static Node deleteNode(Node head, int value) {
        if (head == null) {
            System.out.println("List is empty. Cannot delete.");
            return null;
        }
        if (head.data == value) {
            Node newHead = head.next;
            head.next = null; // Disconnect the deleted node from
the rest of the list
```

```
            delete head;
            return newHead;
        }
        Node current = head;
        while (current.next ≠ null && current.next.data ≠
value) {
            current = current.next;
        }
        if (current.next == null) {
            System.out.println("Node with value " + value + " not
found.");
            return head;
        }
        Node temp = current.next;
        current.next = current.next.next;
        temp.next = null; // Disconnect the deleted node from the
rest of the list
        delete temp;
        return head;
    }

    // Display the linked list
    public static void displayLinkedList(Node head) {
        while (head ≠ null) {
            System.out.print(head.data + " ");
            head = head.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Node head = null;

        // Delete from the Beginning
        head = deleteFromBeginning(head);
        displayLinkedList(head);

        // Delete from the End
        head = deleteFromEnd(head);
        displayLinkedList(head);

        // Delete a Specific Node
        head = deleteNode(head, 2);
        displayLinkedList(head);
    }
}
```

Explanation:

Delete from the Beginning: Check if the list is empty. If not, delete the current head and set the next node as the new head.

Delete from the End: Check if the list is empty or has only one node. If not, traverse to the second-to-last node, delete the last node, and set the next pointer of the second-to-last node to null.

Delete a Specific Node: Check if the list is empty. If not, traverse the list to find the node with the specified value. Delete the node by adjusting pointers.

Time and Space Complexity:

1. **Delete from the Beginning:**

Time Complexity: O(1)
Space Complexity: O(1)

2. **Delete from the End:**

Time Complexity: O(n) - in the worst case
Space Complexity: O(1)

3. **Delete a Specific Node:**

Time Complexity: O(n) - in the worst case
Space Complexity: O(1)

PW SKILLS

THANK YOU !