Module III:
Linked Lists

1. Introduction to Linked Lists

This initial section serves as a formal introduction to **Module III**, which is entirely dedicated to the exploration of **Linked Lists**. It clearly states the module's topic, immediately informing the audience about the subject matter. Essential contextual information, such as the presenter's name, course code, and institution, is provided to maintain a professional and academic tone. This opening section is crucial for setting the academic stage for a deep dive into **linked list data structures**. It marks a transition from previous topics (like arrays, stacks, and queues) to a distinct category of **non-contiguous linear data structures**. A well-presented introduction not only offers clarity but also prepares learners for the specific conceptual, operational, and application-based knowledge of linked lists that will be covered. It's designed to capture attention and direct focus toward the fascinating world of dynamically allocated data collections.

2. Learning Objectives for Linked Lists

This part of the module explicitly lays out the **learning objectives**, providing a clear roadmap of what participants are expected to achieve and understand.

The first objective, "**Understand types of linked lists**," indicates that the module will differentiate between various forms of linked lists, including **singly, doubly, and circular linked lists**. This involves grasping their distinct structures, how nodes are connected, and the implications of these structures on operations. The focus is on recognizing the architectural variations that define each type.

The second objective, "**Implement linked list operations**," emphasizes the practical, hands-on aspect. Learners will delve into the core functionalities that make linked lists useful. This includes understanding and potentially writing code for operations such as **creating a linked list**, **inserting new nodes** (at the beginning, end, or specific positions), **deleting existing nodes**, and **traversing the list** to access elements. This objective centers on the procedural knowledge required to effectively manipulate linked lists.

Finally, "**Explore applications**" highlights the real-world relevance of linked lists. This objective aims to showcase how these data structures are employed in various computational problems and software systems. Examples might range from **managing dynamic memory** and **implementing other data structures** (like stacks and queues) to maintaining browser history, managing music playlists, or handling circular buffers in operating systems. By the end of the module, attendees should not only comprehend the mechanics of linked lists but also be able to identify scenarios where they offer optimal solutions compared to other data structures, thereby fostering a practical and applied understanding of the topic.

3. What is a Linked List?

A **Linked List** is a fundamental **dynamic data structure** that significantly differs from arrays in its memory allocation and flexibility. Unlike arrays, which store elements in contiguous memory locations, linked lists store data in discrete units called **nodes**. Each node typically contains two parts: the **actual data (or value)** and a **pointer (or reference)** to the next node in the sequence. This pointer

is key to connecting nodes, forming a chain. The entire list is accessed via a **head pointer**, which points to the very first node. If the head pointer is NULL, the list is empty.

The defining characteristic of a linked list as a "dynamic data structure" means its **size is not fixed** at creation. It can **grow or shrink dynamically** at runtime as elements are added or removed, limited only by available memory. This contrasts sharply with arrays, which are typically fixed in size. When an array needs to grow, a new, larger array must be allocated, and all existing elements copied over, an expensive **O(N) operation**. Linked lists, by adjusting pointers, can insert or delete elements efficiently (often **O(1)** or **O(N)** depending on position, but without large-scale data shifts).

Comparing Linked Lists with Arrays:

- **Memory Allocation**: Arrays require **contiguous memory**; linked lists use **non-contiguous memory**, connected by pointers, which makes them flexible for memory usage.
- **Size**: Arrays are generally **fixed-size (static)** unless explicitly resized; linked lists are **dynamic**, easily growing or shrinking.
- **Insertion/Deletion**: Arrays are **costly (O(N))** for insertions/deletions in the middle as elements need shifting. Linked lists can perform these operations **efficiently (O(1)** at ends/known pointer, O(N) for searching in the middle).
- **Random Access**: Arrays offer **O(1) random access** by index (e.g., arr[5]). Linked lists require **sequential traversal** from the head to reach a specific element (O(N) in the worst case).
- **Memory Overhead**: Linked lists have **higher memory overhead** per element due to storing pointers; arrays typically only store data.

In summary, linked lists excel where **dynamic size**, **frequent insertions/deletions**, and **non-contiguous memory** are beneficial, while arrays are preferred for fixed-size collections with a strong need for **fast random access**.

4. Structure of a Node

The fundamental building block of any linked list is the **node**. Understanding the structure of a node is paramount to grasping how linked lists operate. Conceptually, each node in a linked list is a self-contained unit designed to hold two distinct pieces of information:

- **Data (or Value)**: This part of the node stores the actual information or element that the linked list is meant to manage. This could be any data type, such as an integer, a string, an object, or a more complex custom data structure. This is the **payload** of the node.
- **Pointer to Next (or Reference to Next)**: This is the crucial component that defines the "**linked**" aspect of a linked list. It stores the **memory address (or a reference)** of the subsequent node in the sequence. This pointer acts as the connection, forming the chain of nodes. For the last node in a singly linked list, this pointer will typically be NULL (or None in Python) to signify the end of the list. In a doubly linked list, there would be an additional pointer to the previous node.

Visual Representation (Box & Arrow Diagram):

To truly visualize the concept, a "box and arrow diagram" is incredibly helpful. Imagine each node as a rectangular box:

- The **left half** of the box represents the **data field**, where the actual value is written.
- The **right half** of the box represents the **pointer to next field**. From this half, an "arrow" originates, pointing to the next rectangular box in the sequence.

The entire linked list begins with a special **head pointer** (often depicted as an arrow pointing from outside the list to the first node). If the head pointer points to NULL, it means the list is empty. This visual model clearly illustrates how individual, non-contiguous nodes are logically connected to form an ordered sequence, allowing traversal from the head through each next pointer until the NULL pointer marks the end of the list. This simple yet powerful structure enables the dynamic nature and efficient element manipulation that define linked lists.

5. Types of Linked Lists

Linked lists, while sharing the core concept of interconnected nodes, come in several variations, each optimized for different use cases and offering distinct operational capabilities. The three primary types are: **Singly Linked Lists**, **Doubly Linked Lists**, and **Circular Linked Lists**.

- **Singly Linked Lists (SLL)**: This is the most basic form, where each node contains **data** and a **pointer that points only to the next node** in the sequence. Traversal is strictly **unidirectional**, from the head to the tail. The last node's pointer points to NULL, indicating the end of the list. They are simple to implement and require less memory per node (just one pointer). However, they cannot be traversed backward, and deleting a node requires knowledge of its preceding node, often necessitating a traversal from the head.
- **Doubly Linked Lists (DLL)**: Building upon the singly linked list, each node in a doubly linked list contains **data and two pointers**: one pointing to the next node and another pointing to the previous node. This **bidirectional linkage** allows traversal in both forward and backward directions. The previous pointer of the first node and the next pointer of the last node typically point to NULL. The advantages include much easier and more efficient deletion (if you have a pointer to the node to be deleted, you can update both its next and previous neighbors directly) and backward traversal. The trade-off is increased memory overhead per node (due to the extra pointer) and slightly more complex insertion/deletion logic to manage both next and previous pointers.
- **Circular Linked Lists (CLL)**: In a circular linked list, the last node's pointer **does not point to NULL**. Instead, it points back to the first node of the list, forming a **continuous loop**. This can apply to both singly and doubly linked lists.
    - **Singly Circular Linked List**: The next pointer of the last node points to the head node.
    - Doubly Circular Linked List: The next pointer of the last node points to the head, and the previous pointer of the head node points to the last node.

The primary advantage of circular linked lists is that any node can serve as a starting point for traversal, and the entire list can be traversed from any node. They are useful for applications that require repetitive cycling through elements, like round-robin scheduling. The drawback is that checking for the end of the list (or an empty list) requires more careful logic, as there's no NULL terminator.

Each type offers specific strengths and weaknesses, making the choice dependent on the particular requirements for data manipulation and traversal in an application.

6. Singly Linked List

A **Singly Linked List (SLL)** is the simplest and most fundamental type of linked list. Its core concept revolves around a **unidirectional chain of interconnected nodes**. Each node in a singly linked list is composed of two distinct fields:

- **Data Field**: This part holds the actual value or piece of information that the node is storing. This data can be of any type—an integer, a string, an object, etc.
- **Next Pointer (or Reference)**: This is the crucial link. It stores the memory address or a reference to the next node in the sequence. This pointer is what establishes the logical connection between consecutive nodes.

The entire singly linked list is accessed and managed through a special pointer known as the **Head Pointer**. This head pointer always points to the very first node of the list. It serves as the entry point; to access any element in the list, you must start from the head and follow the next pointers sequentially. If the head pointer is NULL (or None in Python), it signifies that the linked list is currently empty, meaning it contains no nodes.

The connection between nodes is **strictly one-directional**. From any given node, you can only move forward to the next node in the sequence by following its next pointer. There is no direct way to move backward to the previous node without restarting traversal from the head. The last node in a singly linked list is identified by its next pointer, which points to NULL, indicating that there are no further nodes in the chain. This simplicity in structure leads to straightforward implementation for basic operations but also introduces limitations, particularly concerning backward traversal and certain deletion scenarios. Despite these limitations, the singly linked list forms the bedrock for understanding more complex linked list variations and is highly efficient for many dynamic data management tasks where sequential forward access is sufficient.

Singly Linked Lists, despite their simple structure, support a comprehensive set of operations essential for dynamic data management.

- **Creation**: An empty singly linked list is typically created by initializing its **head pointer to NULL (or None)**. To add the first element, a new node is created, and the head pointer is made to point to this new node.
- **Insertion**:
    - **Insertion at the Beginning**: This is the most efficient insertion (**O(1)**). A new node is created. Its next pointer is set to point to the current head node. Then, the head pointer is updated to point to the newly created node.
    - **Insertion at the End**: This typically requires traversing the entire list from the head to find the last node (the one whose next pointer is NULL). Once the last node is found, its next pointer is updated to point to the new node. This operation is **O(N)** in the worst case unless a separate tail pointer is maintained, which can reduce it to **O(1)**.
    - **Insertion in the Middle (After a specific node)**: This involves finding the node after which the new node needs to be inserted. Once found, the new node's next pointer is set to the found node's next (effectively linking it to the rest of the list), and the found node's next pointer is updated to point to the new node. This is **O(N)** due to the traversal to find the position.

- **Deletion**:
    - **Deletion from the Beginning**: The head pointer is simply moved to point to the second node in the list (head = head.next). The old first node is then eligible for garbage collection. This is **O(1)**.
    - **Deletion from the End**: This requires traversing the list to find the second-to-last node. Its next pointer is then set to NULL. This is an **O(N)** operation.
    - **Deletion from the Middle (a specific node)**: To delete a specific node, you must have access to its preceding node. The next pointer of the preceding node is then updated to skip the node to be deleted and point directly to the deleted node's next. Finding the preceding node typically requires **O(N)** traversal.
- **Traversal**: This operation involves visiting each node in the list, usually to print its data or perform some other action. It starts from the head and repeatedly moves to the next node by following its next pointer until a NULL pointer is encountered, signifying the end of the list. This is an **O(N)** operation, where N is the number of nodes.

These operations collectively enable singly linked lists to be dynamic and flexible for many data management tasks, particularly those involving frequent insertions or deletions at the beginning.

7. Code Example: Singly Linked List

This section presents a clear, conceptual code example for implementing basic operations on a Singly Linked List, typically in Python due to its readability, or C for direct memory manipulation. The code illustrates the **Node structure** and core linked list functionalities.

The Python code structure would likely include:

Python

```
# Node class definition
class Node:
    def __init__(self, data):
        self.data = data  # Data stored in the node
        self.next = None  # Pointer to the next node, initialized to None

# Singly Linked List class definition
class LinkedList:
    def __init__(self):
        self.head = None  # Head of the list, initially empty

    # --- Basic Operations ---
    # 1. Insertion at the Beginning (push)
    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head  # New node points to current head
        self.head = new_node       # Head now points to new node
        print(f"Inserted {data} at beginning.")

    # 2. Insertion at the End (append)
    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
```

```
            self.head = new_node  # If list is empty, new node is the head
            print(f"Inserted {data} at end (list was empty).")
            return
        current = self.head
        while current.next:       # Traverse to the last node
            current = current.next
        current.next = new_node   # Last node points to new node
        print(f"Inserted {data} at end.")

    # 3. Traversal (print all elements)
    def traverse(self):
        if self.head is None:
            print("List is empty.")
            return
        current = self.head
        elements = []
        while current:
            elements.append(current.data)
            current = current.next
        print("Linked List:", " -> ".join(map(str, elements)))

    # 4. Deletion of a specific value (simplified, often by position or value)
    def delete_node(self, key):
        current = self.head
        prev = None

        # If head node itself holds the key to be deleted
        if current and current.data == key:
            self.head = current.next
            current = None # For garbage collection
            print(f"Deleted {key} from beginning.")
            return

        # Search for the key to be deleted, keep track of previous node
        while current and current.data != key:
            prev = current
            current = current.next

        # If key was not present in linked list
        if current is None:
            print(f"{key} not found in list.")
            return

        # Unlink the node from linked list
        prev.next = current.next
        current = None # For garbage collection
        print(f"Deleted {key}.")

# --- Example Usage (Commented out for notes, but would be demonstrated) ---
# my_list = LinkedList()
# my_list.insert_at_beginning(10)
# my_list.insert_at_end(20)
# my_list.insert_at_beginning(5)
# my_list.traverse() # Expected: 5 -> 10 -> 20
# my_list.delete_node(10)
```

```
# my_list.traverse() # Expected: 5 -> 20
```

This code illustrates the fundamental structure of a Node and a LinkedList class. It provides methods for creating a list, adding elements at both the beginning and end, traversing the list to print its contents, and deleting a node by its value. Comments within the code explain each step, making it easy to follow the logic. This example concretely demonstrates how **pointers (self.next and self.head)** are manipulated to manage the dynamic chain of elements in a singly linked list.

8. Doubly Linked List – Concept

A **Doubly Linked List (DLL)** is an extension of the singly linked list that directly addresses a key limitation: the inability to traverse backward. In a DLL, each node is enhanced to contain three distinct parts:

- **prev Pointer (Previous Pointer)**: This pointer stores the memory address or reference to the node **immediately preceding** it in the sequence. For the first node (the head) in the list, this prev pointer will be NULL (or None).
- **data (or Value)**: This field holds the actual information stored in the node, identical to the data field in a singly linked list.
- **next Pointer (Next Pointer)**: This pointer stores the memory address or reference to the node **immediately succeeding** it in the sequence, just like in a singly linked list. For the last node (the tail) in the list, this next pointer will be NULL.

This **dual-pointer structure** (previous and next) establishes **bidirectional links** between nodes.

DLL vs. SLL (Key Differences and Implications):

- **Traversal**:
  - **SLL**: Only **forward traversal** is possible from the head (unidirectional).
  - **DLL**: Both **forward** (using the next pointer) and **backward** (using the prev pointer) traversal are possible. This significantly enhances flexibility for operations requiring movement in both directions.
- **Memory Usage**:
  - **SLL**: Requires **less memory per node**, as each node stores only one pointer (next).
  - **DLL**: Requires **more memory per node**, as each node stores two pointers (prev and next). For applications handling a very large number of small data items, this increased overhead can be a consideration.
- **Complexity of Operations**:
  - **SLL**: Insertion and deletion logic (especially in the middle or at the end) can sometimes be simpler to conceptualize but are less efficient, as finding the preceding node often requires a full traversal.
  - **DLL**: Insertion and deletion logic is slightly more complex than SLLs because two pointers (prev and next) need to be updated for both the new/deleted node and its neighbors. However, once the node to be operated on is found, **deletion is O(1)**

because both its neighbors can be directly accessed. This is a major advantage over SLLs. For instance, deleting a specific node X in an SLL requires finding the node before X. In a DLL, if you have a pointer to X, you can directly update X.prev.next and X.next.prev.

In essence, DLLs offer enhanced functionality and efficiency for certain operations (especially deletions and backward traversals) at the cost of increased memory footprint per node and slightly more intricate pointer management during updates.

9. Doubly Linked List – Operations

Doubly Linked Lists provide more robust and flexible operations compared to singly linked lists due to their bidirectional pointers.

- **Creation**: An empty DLL is initialized by setting both its **head and tail (or last) pointers to NULL**. The very first node inserted becomes both the head and tail.
- **Insertion**: Inserting a node in a DLL generally involves updating four pointers (two for the new node, two for its neighbors).
    - **Insertion at the Beginning (addFront)**:
        - Create new_node.
        - Set new_node.next = head.
        - Set new_node.prev = NULL.
        - If the list is not empty, set head.prev = new_node.
        - Update head = new_node.
        - If the list was initially empty, tail = new_node.
        - This is an **O(1) operation**.
    - **Insertion at the End (addRear)**:
        - Create new_node.
        - Set new_node.prev = tail.
        - Set new_node.next = NULL.
        - If the list is not empty, set tail.next = new_node.
        - Update tail = new_node.
        - If the list was initially empty, head = new_node.
        - This is an **O(1) operation** (assuming a tail pointer is maintained).
    - **Insertion in the Middle (after a specific node)**:
        - First, **find the existing_node** after which to insert (this step usually takes **O(N)** time due to traversal).
        - Create new_node.

        - new_node.next = existing_node.next.
        - new_node.prev = existing_node.
        - If existing_node.next is not NULL (i.e., not inserting at the very end), set existing_node.next.prev = new_node.
        - Finally, set existing_node.next = new_node.
        - While finding the node is O(N), the actual pointer manipulation once the existing_node is found is **O(1)**.
- **Deletion**: Deletion is one of the biggest advantages of DLLs, especially when a pointer to the node to be deleted is already known. This operation involves updating two next and two prev pointers.
    - **Deletion from the Beginning (removeFront)**:
        - Update head = head.next.
        - If the new head is not NULL, set head.prev = NULL.
        - If the list becomes empty, tail = NULL.
        - This is **O(1)**.
    - **Deletion from the End (removeRear)**:
        - Update tail = tail.prev.
        - If the new tail is not NULL, set tail.next = NULL.
        - If the list becomes empty, head = NULL.
        - This is **O(1)** (assuming a tail pointer).
    - **Deletion of a specific node (by pointer or value)**:
        - **Find the node_to_delete** (this step takes **O(N)** to find the node if only its value is known).
        - If node_to_delete.prev is not NULL, set node_to_delete.prev.next = node_to_delete.next.
        - If node_to_delete.next is not NULL, set node_to_delete.next.prev = node_to_delete.prev.
        - Handle edge cases: if deleting the head or tail, update those pointers accordingly.
        - While finding the node is O(N), the actual unlinking once the node_to_delete is found is **O(1)**.
- **Traversal**:
    - **Forward Traversal**: Start from the head and follow next pointers. This is an **O(N)** operation.

- **Backward Traversal**: Start from the tail and follow prev pointers. This is also an **O(N)** operation. This bidirectional traversal is a key strength, allowing efficient movement in either direction without restarting from an end.

These operations collectively highlight how DLLs provide robust and efficient data manipulation, particularly when bidirectional movement or efficient deletion (given a node pointer) is required.