Introduction to Linked list

Lesson Plan

PW SKILLS

Today's Checklist:

- Limitations of Arrays
- Introduction to Linked List
- Implementing Linked List
- Displaying
- Insert in Linked List
- Limitation
- Delete in Linked List

Limitations of Arrays

- **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.

- **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.

- **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.

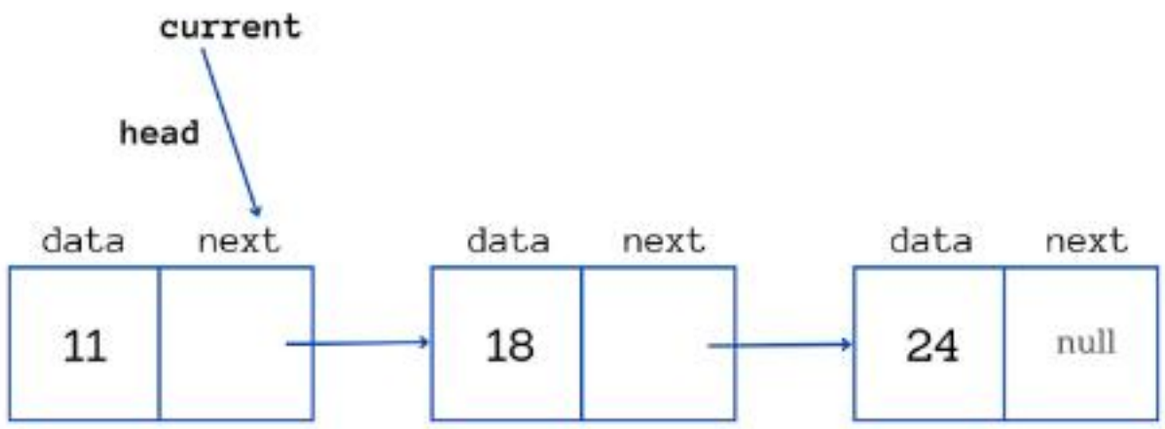
- **Wastage of Memory:** If you allocate more space than needed for an array, you may end up wasting memory. This is particularly problematic when the array size is predetermined to accommodate the worst-case scenario.

- **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.

- **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?

- Dynamic Data structure: The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.

- 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.

- 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.

- 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:

- **Node Class:**

Create a class for the linked list node with data and a pointer to the next node.

```
class Node:
    def __init__(self, value):
        self.data = value
        self.next = None
```

- **LinkedList Class:**

Create a class to represent the linked list.
Include a pointer to the head of the list.

```
class LinkedList:
    def __init__(self):
        self.head = None
```

- **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.

```
def add_node(self, value):
    new_node = Node(value)
    if self.head is None:
        self.head = new_node
    else:
        current = self.head
        while current.next is not None:
            current = current.next
        current.next = new_node
```

- **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.

```
class Node:
    def __init__(self, value):
        self.data = value
        self.next = None

def display_linked_list_iterative(head):
    while head is not None:
        print(head.data, end=" ")
        head = head.next
    print()

def display_linked_list_recursive(head):
    if head is None:
        return
    print(head.data, end=" ")
    display_linked_list_recursive(head.next)

if __name__ == "__main__":
    head = Node(1)
    head.next = Node(2)
    head.next.next = Node(3)
    head.next.next.next = Node(4)

    print("Iterative Display: ", end="")
    display_linked_list_iterative(head)

    print("Recursive Display: ", end="")
    display_linked_list_recursive(head)
    print()
```

Explanation:

Iterative Display:

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

Recursive Display:

- Base case: If the current node is null, return.
- Print the data of the current node.
- Make a recursive call with the next node.
- The recursion unwinds, printing nodes in sequential order.
- 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.

Length of Linked List

```
def find_length_iterative(head):
    length = 0
    while head is not None:
        length += 1
        head = head.next
    return length

# Test the iterative length function
if __name__ == "__main__":
    head = Node(1)
    head.next = Node(2)
    head.next.next = Node(3)
    head.next.next.next = Node(4)
    print("Length (Iterative):", find_length_iterative(head))  #
Output: 4


def find_length_recursive(head):
    if head is None:
        return 0
    return 1 + find_length_recursive(head.next)

# Test the recursive length function
if __name__ == "__main__":
    head = Node(1)
    head.next = Node(2)
    head.next.next = Node(3)
    head.next.next.next = Node(4)
    print("Length (Recursive):", find_length_recursive(head))  #
Output: 4
```

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...
- Inserting At the Beginning of the list
- Inserting At End of the list
- Inserting At Specific location in the list

Code:

```
def insert_at_beginning(head, value):
    new_node = Node(value)
    new_node.next = head
    return new_node

# Test insert at beginning
if __name__ == "__main__":
    head = None
    head = insert_at_beginning(head, 1)
    head = insert_at_beginning(head, 2)
    display(head)  # Output: 2 1

def insert_at_end(head, value):
    new_node = Node(value)
    if head is None:
        return new_node
    current = head
    while current.next is not None:
        current = current.next
    current.next = new_node
    return head

# Test insert at end
if __name__ == "__main__":
    head = None
    head = insert_at_end(head, 1)
    head = insert_at_end(head, 2)
    display(head)  # Output: 1 2

def insert_at_location(head, value, position):
    new_node = Node(value)
    if position == 1:
        new_node.next = head
        return new_node
    current = head
    for i in range(1, position - 1):
        if current is None:
            print("Invalid position.")
            return head
        current = current.next
    if current is None:
        print("Invalid position.")
        return head
    new_node.next = current.next
    current.next = new_node
    return head

# Test insert at specific location
if __name__ == "__main__":
    head = None
    head = insert_at_end(head, 1)
    head = insert_at_end(head, 3)
    head = insert_at_location(head, 2, 2)
    display(head)  # Output: 1 2 3
```

Insert at the Beginning:

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

Insert at the End:

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

Insert at Specific Location:

- Create a New Node: Allocate memory for a new node and set its data.
- Handle Special Case (Insert at the Beginning): If the position is 1, link the new node to the current head and update the head.
- Traverse to the Previous Node: Iterate through the list to the node preceding the desired position.
- 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:

- Deleting from the Beginning of the list
- Deleting from the End of the list
- Deleting a Specific Node

Code:

```
def delete_from_beginning(head):
    if head is None:
        print("List is empty. Cannot delete.")
        return None
    new_head = head.next
    head.next = None
    return new_head

# Test delete from beginning
if __name__ == "__main__":
    head = None
    head = insert_at_end(head, 1)
    head = insert_at_end(head, 2)
    head = delete_from_beginning(head)
    display(head)  # Output: 2

def delete_from_end(head):
    if head is None:
        print("List is empty. Cannot delete.")
        return None
    if head.next is None:
        return None
    current = head
    while current.next.next is not None:
        current = current.next
    current.next = None
    return head

# Test delete from end
if __name__ == "__main__":
    head = None
    head = insert_at_end(head, 1)
    head = insert_at_end(head, 2)
    head = delete_from_end(head)
    display(head)  # Output: 1

def delete_node(head, value):
    if head is None:
        print("List is empty. Cannot delete.")
        return None
    if head.data == value:
        new_head = head.next
        head.next = None
        return new_head
    current = head
    while current.next is not None and current.next.data ≠
value:
        current = current.next
    if current.next is None:
        print(f"Node with value {value} not found.")
        return head
    temp = current.next
    current.next = current.next.next
    temp.next = None
    return head

# Test delete specific node
if __name__ == "__main__":
    head = None
    head = insert_at_end(head, 1)
    head = insert_at_end(head, 2)
    head = insert_at_end(head, 3)
    head = delete_node(head, 2)
    display(head)  # Output: 1 3
```

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)