**EXPERIMENT NO. 6**

| **Objective(s):**  To implement a single linked list data structure and perform operations such as creation, insertion, deletion, traversal, search, and reversal. |
| --- |
| **Outcome:**  Understanding how to implement and manipulate a single linked list, gaining proficiency in basic operations of linked lists. |
| **Problem Statement:**  Implement Single Linked lists and its operations(creation insertion deletion traversal search reverse). |
| **Background Study:**  **Definition**:   * A linked list is a linear data structure where each element is a separate object called a node. * Each node contains two parts: the data and a reference (link) to the next node in the sequence.   **Types of Linked Lists:**   * **Single** **Linked** **List**: Each node points to the next node in the sequence. * **Double** **Linked** **List**: Each node has two references, one to the next node and another to the previous node. * **Circular** **Linked** **List**: Last node points back to the first node.   **Operations on Single Linked Lists:**   1. **Creation of a Linked List:**  * Create an empty linked list by setting the head pointer to NULL.  1. **Insertion**:  * **Insert at the Beginning**: Create a new node, set its next to the current head, and update the head to point to the new node. * **Insert at the End**: Traverse to the end of the list, create a new node, and set the next of the last node to the new node. * **Insert at a Position**: Traverse to the desired position, adjust the next pointers to insert the new node.  1. **Deletion:**  * **Delete** **at the** **Beginning**: Update the head to point to the second node and free the memory of the first node. * **Delete** **at** **the End**: Traverse to the second last node, update its next to NULL, and free the memory of the last node. * **Delete at** **a** **Position:** Traverse to the node before the position, adjust the next pointers to skip the node to be deleted, and free its memory.  1. **Traversal:**  * Start from the head and move to the next node until the end (NULL) is reached, printing or processing each node.  1. **Search:**  * Traverse the list, comparing each node's data with the target value until the target is found or the end of the list is reached.  1. **Reverse:**  * Reverse the order of nodes in the linked list by adjusting the next pointers to point in theopposite direction.   **Advantages:**   * Dynamic size: Easily grow and shrink in size during runtime. * Efficient Insertions and Deletions: Insertions and deletions can be done in constant time, O(1), whenperformed at the beginning of the list.   **Disadvantages:**   * More memory overhead than arrays because of the storage used by pointers. * Sequential access is slow compared to arrays.   **Applications:**   * Implementation of stacks, queues, and hash tables. * Undo functionality in software applications. * Used in adjacency list representation of graphs. |

| **Algorithm (Student Work Area):**   1. **Creation**  * Step 1: Initialize the head pointer to null or `None`.  1. **Insertion** 2. **At the beginning:**    1. Step 1: Create a new node.    2. Step 2: Set the new node's next pointer to the current head.    3. Step 3: Update the head pointer to the new node. 3. **At the end:**    1. Step 1: Create a new node.    2. Step 2: If the list is empty, set the head pointer to the new node.    3. Step 3: Otherwise, traverse to the last node.    4. Step 4: Set the last node's next pointer to the new node. 4. **At a given position:**    1. Step 1: Create a new node.    2. Step 2: Traverse to the node currently at the given position.    3. Step 3: Adjust the new node's next pointer to point to the node at the given position.    4. Step 4: Adjust the previous node's next pointer to include the new node. 5. **Deletion** 6. **From the beginning:**    1. Step 1: If the list is empty, return.    2. Step 2: Update the head pointer to the next node. 7. **From the end:**    1. Step 1: If the list is empty, return.    2. Step 2: Traverse to the second-to-last node.    3. Step 3: Update the second-to-last node's next pointer to null. 8. **From a given position:**    1. Step 1: Traverse to the node just before the given position.    2. Step 2: Adjust the previous node's next pointer to exclude the node to be deleted. 9. **Traversal** 10. **Forward Traversal:**     1. Step 1: Start from the head.     2. Step 2: While the current node is not null, process the current node and move to the next node. 11. **Search** 12. Step 1: Start from the head. 13. Step 2: While the current node is not null, compare its value with the target value. 14. Step 3: If a match is found, return the node. 15. Step 4: Move to the next node. 16. Step 5: If the end of the list is reached without finding the value, return null or `None`. 17. **Reverse** 18. Step 1: Start from the head. 19. Step 2: Initialize a temporary variable to null to store the previous node. 20. Step 3: While the current node is not null:     1. Step 3a: Store the next node.     2. Step 3b: Set the current node's next pointer to the previous node.     3. Step 3c: Move the temporary variable to the current node.     4. Step 3d: Move the current node to the next node (stored in step 3a). 21. Step 4: After the loop, update the head pointer to the last processed node (stored in the temporary variable). |
| --- |

| **Code:** |
| --- |
| **OUTPUT :** |