**EXPERIMENT NO. 7**

| **Objective(s):**  Design and implement a doubly linked list data structure supporting creation, insertion, deletion, traversal, search, and reversal operations. |
| --- |
| **Outcome:**  Efficient data organization with ability to add, remove, find, and iterate elements in both forward and backward directions. |
| **Problem Statement:**  Implement Double Linked lists and its operations(creation insertion deletion traversal search reverse) |
| **Background Study:**  **Doubly Linked Lists**  A doubly linked list is a linear data structure where each element (node) contains data and references to two other nodes:   * + **Next:** Points to the next node in the list.   + **Previous:** Points to the previous node in the list.   This enables traversal and modification in both forward and backward directions, unlike singly linked lists which only allow forward traversal.   **Theory:**   * + **Nodes:** Each node consists of three parts:     - **Data:** The actual information stored in the node (integer, string, etc.).     - **Next:** A pointer to the next node in the list.     - **Previous:** A pointer to the previous node in the list.   + **Head and Tail:**     - **Head:** A pointer to the first node in the list. If the list is empty, the head points to null.     - **Tail:** A pointer to the last node in the list. If the list is empty, the tail points to null.   + **Operations:**     - **Creation:** Allocate memory for a new node, initialize its data, and set its next and previous pointers to null. The head and tail are also set to this new node if the list is empty.     - **Insertion:**       * **At the beginning:** Update the new node's next pointer to point to the current head, update the current head's previous pointer to point to the new node, and finally update the head to point to the new node.       * **At the end:** Update the new node's previous pointer to point to the current tail, update the current tail's next pointer to point to the new node, and finally update the tail to point to the new node.       * **In the middle:** Traverse to the desired position, update pointers of the new node, its neighbors, and the surrounding nodes.     - **Deletion:**       * **First node:** Update the head to point to the second node (if it exists) and set the second node's previous pointer to null.       * **Last node:** Update the tail to point to the second-last node (if it exists) and set the second-last node's next pointer to null.       * **Middle node:** Traverse to the node, update pointers of its neighbors to bypass it.     - **Traversal:** Start from the head and follow the next pointers until a null pointer is encountered. Print the data of each node during traversal.     - **Search:** Start from the head and traverse the list, comparing the data of each node with the search key. Return the node's pointer if found, otherwise return null.     - **Reversal:** Reverse the direction of the next and previous pointers for each node in the list. The head becomes the tail and vice versa.   **Benefits of Doubly Linked Lists:**   * + Efficient insertion and deletion at any position compared to arrays.   + No need to shift elements during insertion/deletion in the middle.   + Support for forward and backward traversal.   **Drawbacks of Doubly Linked Lists:**   * + More memory overhead compared to arrays due to the extra pointer (previous) in each node.   + Random access (accessing a specific element by index) is inefficient as it requires traversal from the beginning.   **Use Cases**   * + *Navigation Systems*: Where you need to go back and forth through the list of locations or steps.   + *Undo/Redo Functionality*: In applications like text editors, where you can move back and forth between states.   + *Complex Data Structures*: Forms the basis for more advanced data structures like balanced trees and certain types of heaps. |

| **Algorithm (Student Work Area):**  **Algorithms for Doubly Linked List Operations**  **1. Creation**  **Objective:** Initialize an empty doubly linked list.   * + Step 1: Initialize the head pointer to null or None.   **2. Insertion**  **Objective:** Add a new node to the list at a specified position.   * **a. At the beginning:**    + Step 1: Create a new node.   + Step 2: Set the new node's next pointer to the current head.   + Step 3: If the list is not empty, set the current head's previous pointer to the new node.   + Step 4: Update the head pointer to the new node. * **b. At the end:**    + Step 1: Create a new node.   + Step 2: If the list is empty, set the head pointer to the new node.   + Step 3: Otherwise, traverse to the last node.   + Step 4: Set the last node's next pointer to the new node.   + Step 5: Set the new node's previous pointer to the last node. * **c. At a given position:**    + Step 1: Create a new node.   + Step 2: Traverse to the node currently at the given position.   + Step 3: Adjust the new node's next and previous pointers to point to the surrounding nodes.   + Step 4: Adjust the surrounding nodes' pointers to include the new node.   **3. Deletion**  **Objective:** Remove a node from the list.   * **a. From the beginning:**    + Step 1: If the list is empty, return.   + Step 2: Update the head pointer to the next node.   + Step 3: If the new head is not null, set its previous pointer to null. * **b. From the end:**    + Step 1: If the list is empty, return.   + Step 2: Traverse to the last node.   + Step 3: Update the second-to-last node's next pointer to null. * **c. From a given position:**    + Step 1: Traverse to the node at the given position.   + Step 2: Adjust the surrounding nodes' pointers to exclude the node to be deleted.   **4. Traversal**  **Objective:** Visit each node in the list.   * **a. Forward Traversal:**    + Step 1: Start from the head.   + Step 2: While the current node is not null, process the current node and move to the next node. * **b. Backward Traversal:**    + Step 1: Start from the tail (requires maintaining a tail pointer or traversing to the end).   + Step 2: While the current node is not null, process the current node and move to the previous node.   **5. Search**  **Objective:** Find a node with a specified value.   * Step 1: Start from the head. * Step 2: While the current node is not null, compare its value with the target value. * Step 3: If a match is found, return the node. * Step 4: Move to the next node. * Step 5: If the end of the list is reached without finding the value, return null or None.   **6. Reverse**  **Objective:** Reverse the order of nodes in the list.   * Step 1: Start from the head. * Step 2: Initialize a temporary variable to null to store the previous node. * Step 3: While the current node is not null:   + Step 3a: Swap the current node's next and previous pointers.   + Step 3b: Move the temporary variable to the current node.   + Step 3c: Move the current node to its new next node (previously the previous node). * Step 4: After the loop, update the head pointer to the last processed node (stored in the temporary variable). |
| --- |

| **Code:**  **#include <stdio.h>**  **#include <stdlib.h>**  **// Define the structure of a node in a doubly linked list**  **typedef struct Node {**  **int data;**  **struct Node\* prev;**  **struct Node\* next;**  **} Node;**  **// Function to create a new node**  **Node\* createNode(int data) {**  **Node\* newNode = (Node\*)malloc(sizeof(Node));**  **newNode->data = data;**  **newNode->prev = NULL;**  **newNode->next = NULL;**  **return newNode;**  **}**  **// Function to insert a node at the beginning**  **void insertAtBeginning(Node\*\* head, int data) {**  **Node\* newNode = createNode(data);**  **if (\*head == NULL) {**  **\*head = newNode;**  **} else {**  **newNode->next = \*head;**  **(\*head)->prev = newNode;**  **\*head = newNode;**  **}**  **}**  **// Function to insert a node at the end**  **void insertAtEnd(Node\*\* head, int data) {**  **Node\* newNode = createNode(data);**  **if (\*head == NULL) {**  **\*head = newNode;**  **} else {**  **Node\* temp = \*head;**  **while (temp->next != NULL) {**  **temp = temp->next;**  **}**  **temp->next = newNode;**  **newNode->prev = temp;**  **}**  **}**  **// Function to insert a node at a given position**  **void insertAtPosition(Node\*\* head, int data, int position) {**  **if (position == 1) {**  **insertAtBeginning(head, data);**  **return;**  **}**  **Node\* newNode = createNode(data);**  **Node\* temp = \*head;**  **for (int i = 1; i < position - 1 && temp != NULL; i++) {**  **temp = temp->next;**  **}**  **if (temp == NULL) {**  **printf("Position out of range\n");**  **free(newNode);**  **return;**  **}**  **newNode->next = temp->next;**  **if (temp->next != NULL) {**  **temp->next->prev = newNode;**  **}**  **temp->next = newNode;**  **newNode->prev = temp;**  **}**  **// Function to delete a node from the beginning**  **void deleteFromBeginning(Node\*\* head) {**  **if (\*head == NULL) {**  **printf("List is empty\n");**  **return;**  **}**  **Node\* temp = \*head;**  **\*head = (\*head)->next;**  **if (\*head != NULL) {**  **(\*head)->prev = NULL;**  **}**  **free(temp);**  **}**  **// Function to delete a node from the end**  **void deleteFromEnd(Node\*\* head) {**  **if (\*head == NULL) {**  **printf("List is empty\n");**  **return;**  **}**  **Node\* temp = \*head;**  **while (temp->next != NULL) {**  **temp = temp->next;**  **}**  **if (temp->prev != NULL) {**  **temp->prev->next = NULL;**  **} else {**  **\*head = NULL;**  **}**  **free(temp);**  **}**  **// Function to delete a node from a given position**  **void deleteFromPosition(Node\*\* head, int position) {**  **if (\*head == NULL) {**  **printf("List is empty\n");**  **return;**  **}**  **if (position == 1) {**  **deleteFromBeginning(head);**  **return;**  **}**  **Node\* temp = \*head;**  **for (int i = 1; i < position && temp != NULL; i++) {**  **temp = temp->next;**  **}**  **if (temp == NULL) {**  **printf("Position out of range\n");**  **return;**  **}**  **if (temp->next != NULL) {**  **temp->next->prev = temp->prev;**  **}**  **if (temp->prev != NULL) {**  **temp->prev->next = temp->next;**  **}**  **free(temp);**  **}**  **// Function to traverse the list in forward direction**  **void traverseForward(Node\* head) {**  **Node\* temp = head;**  **while (temp != NULL) {**  **printf("%d ", temp->data);**  **temp = temp->next;**  **}**  **printf("\n");**  **}**  **// Function to traverse the list in backward direction**  **void traverseBackward(Node\* head) {**  **if (head == NULL) {**  **return;**  **}**  **Node\* temp = head;**  **while (temp->next != NULL) {**  **temp = temp->next;**  **}**  **while (temp != NULL) {**  **printf("%d ", temp->data);**  **temp = temp->prev;**  **}**  **printf("\n");**  **}**  **// Function to search for a node with a given value**  **Node\* search(Node\* head, int data) {**  **Node\* temp = head;**  **while (temp != NULL) {**  **if (temp->data == data) {**  **return temp;**  **}**  **temp = temp->next;**  **}**  **return NULL;**  **}**  **// Function to reverse the list**  **void reverseList(Node\*\* head) {**  **if (\*head == NULL) {**  **return;**  **}**  **Node\* current = \*head;**  **Node\* temp = NULL;**  **while (current != NULL) {**  **temp = current->prev;**  **current->prev = current->next;**  **current->next = temp;**  **current = current->prev;**  **}**  **if (temp != NULL) {**  **\*head = temp->prev;**  **}**  **}**  **// Main function to demonstrate the operations**  **int main() {**  **Node\* head = NULL;**    **// Insert nodes**  **insertAtEnd(&head, 1);**  **insertAtEnd(&head, 2);**  **insertAtEnd(&head, 3);**  **insertAtBeginning(&head, 0);**  **insertAtPosition(&head, 5, 3);**    **// Traverse the list**  **printf("List in forward direction: ");**  **traverseForward(head);**    **printf("List in backward direction: ");**  **traverseBackward(head);**    **// Search for a node**  **int searchValue = 2;**  **Node\* searchResult = search(head, searchValue);**  **if (searchResult != NULL) {**  **printf("Node with value %d found.\n", searchValue);**  **} else {**  **printf("Node with value %d not found.\n", searchValue);**  **}**    **// Delete nodes**  **deleteFromPosition(&head, 3);**  **deleteFromEnd(&head);**  **deleteFromBeginning(&head);**    **// Traverse the list again**  **printf("List after deletions: ");**  **traverseForward(head);**    **// Reverse the list**  **reverseList(&head);**  **printf("Reversed list: ");**  **traverseForward(head);**    **return 0;**  **}** |
| --- |
| **OUTPUT :** |