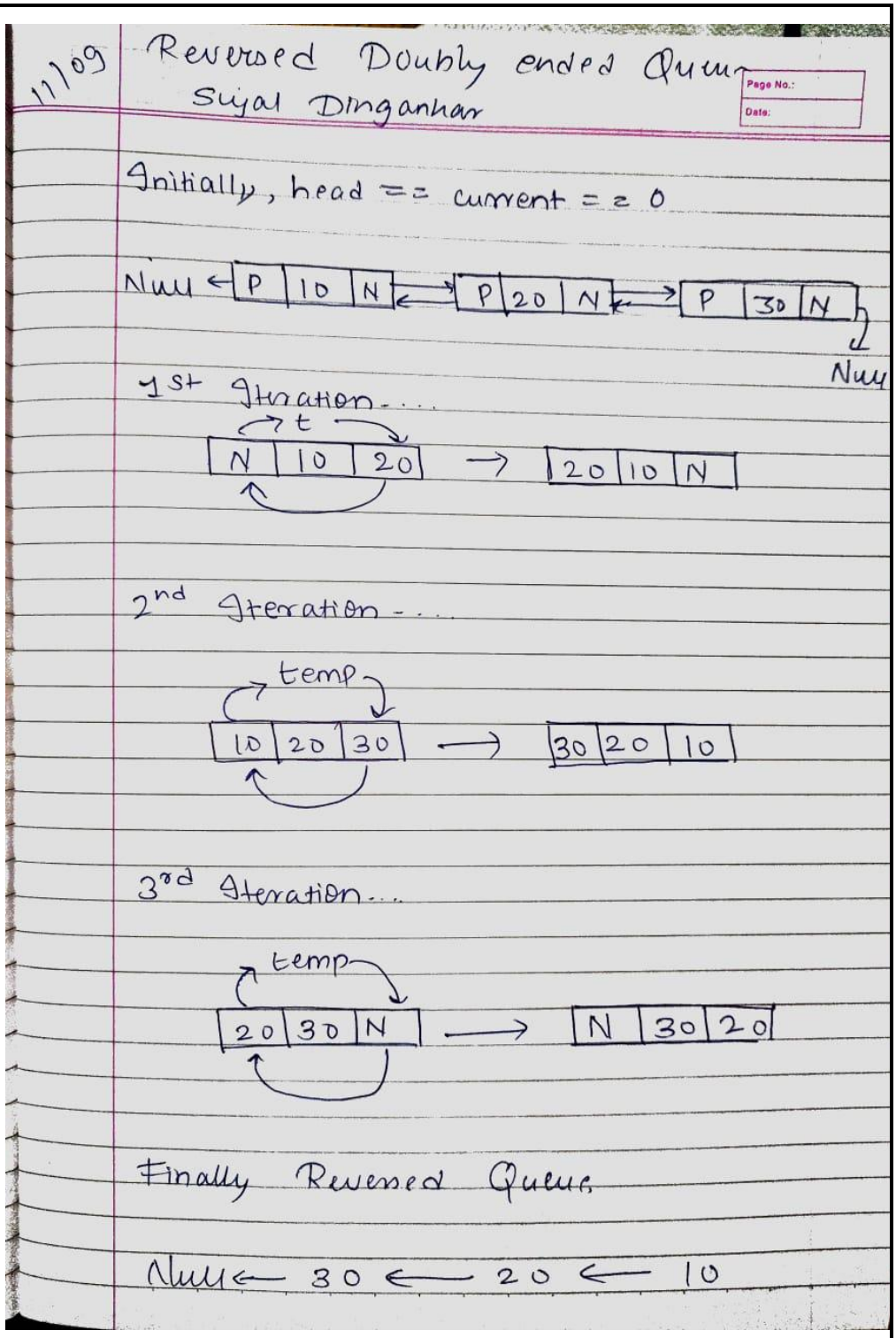
Name	Sujal Dingankar
UID no.	DSE24100720
Experiment No.	4

AIM:	Reverse the doubly linked list.
THEORY:	**Doubly Linked List** A doubly linked list is a linear data structure where each element (node) contains a reference to both the next node and the previous node in the sequence. This allows traversal in both forward and backward directions. Unlike a singly linked list, where each node only points to the next node, a doubly linked list provides more flexibility in navigation and manipulation of the list. Doubly Linked Lists (DLLs) are used in scenarios where bidirectional traversal of elements is needed, as each node contains references to both its previous and next nodes. This allows efficient insertion and deletion of nodes from both ends of the list and simplifies operations that require navigating in both directions, such as implementing certain data structures like deques. DLLs are particularly useful in applications such as navigation systems, undo functionality in software, and managing complex data structures where bidirectional traversal enhances performance and flexibility. **Operations on Doubly Linked List:** 1. **Insertion Operation:** o **At the Beginning:** Adds a new node at the start of the list. The new node's next pointer points to the previous head, and the previous pointer of the previous head points back to the new node. o **At the End:** Adds a new node at the end of the list. The new node's previous pointer points to the current tail, and the next pointer of the current tail points to the new node. o **At a Specific Position:** Inserts a node at a specific position within the list. This involves adjusting the pointers of the adjacent nodes to include the new node. 2. **Deletion Operation:** o **From the Beginning:** Removes the node at the start of the list. The head pointer is updated to point to the second node, and the new head's previous pointer is set to null.

	- **From the End:** Removes the node at the end of the list. The tail pointer is updated to point to the second-to-last node, and the new tail's next pointer is set to null. - **From a Specific Position:** Removes a node from a specific position. The pointers of the adjacent nodes are updated to bypass the removed node. 3. **Traversal Operation:** - **Forward Traversal:** Starts from the head and moves through the list using the next pointers until the end is reached. - **Backward Traversal:** Starts from the tail and moves through the list using the previous pointers until the beginning is reached. 4. **Search Operation:** - Searches for a specific value within the list by traversing either forward or backward until the value is found or the end of the list is reached. 5. **IsEmpty Operation:** - Checks if the list is empty, returning true if there are no nodes and false otherwise. **Applications of Doubly Linked List:** 1. **Navigation Systems:** Useful in applications requiring bidirectional traversal, such as navigation systems or undo/redo functionality. 2. **Deque Implementation:** Can be used to implement double-ended queues (deques) where insertion and deletion operations can occur at both ends efficiently. 3. **Memory Management:** Allows efficient memory management where the list structure needs to be rearranged or manipulated frequently. 4. **Browser History:** Maintains a history of web pages, allowing navigation both forward and backward through the history. 5. **LRU Caching:** Useful in implementing least recently used (LRU) caches where nodes can be efficiently added and removed from both ends.

ALGORITHM:	1. **Input the number of nodes and data:** • Ask the user for the number of nodes in the doubly linked list. • For each node, read its data and create a new node. • Link each node to form the list. 2. **Initialize the doubly linked list:** • Set the head pointer to the first node. • Link all subsequent nodes by setting their next and prev pointers accordingly. 3. **Print the doubly linked list before reversing:** • Traverse the list from the head and print the data of each node to show the original order. 4. **Define the reverse function (reverseDoublyLL):** • Create a function to reverse the doubly linked list. • Initialize two pointers: current to traverse the list and temp to temporarily hold pointers. 5. **Reverse the doubly linked list:** • Traverse the list using the current pointer. o For each node: ▪ Swap the next and prev pointers. ▪ Move to the next node using the previous pointer (which was the old next). • After the traversal, update the head pointer to point to the new head of the reversed list. 6. **Edge case handling:** • Ensure the list has nodes before attempting to reverse. • Update the head pointer correctly to avoid null or incorrect head pointers. 7. **Print the doubly linked list after reversing:** • Traverse the reversed list from the new head and print the data of each node to show the reversed order. 8. **End of program:** • The list has been reversed and displayed in its new order. • Free the allocated memory for the nodes (not included in the provided code but good practice).

PROBLEM SOLVING:



PROGRAM :-

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    struct Node *prev;
    int data;
    struct Node *next;
};

struct Node *reverseDoublyLL(struct Node *head){
    struct Node *current = head;
    struct 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;
    }

    return head;
};

void traversedDDL(struct Node* head){
    struct Node *temp = head;
    printf("Traversed DDL...\n");
    while (temp != NULL)
    {
        printf("%d\n",temp->data);
        temp=temp->next;
    }
};

int main(){
    int noOfNode, value;
    struct Node *head = NULL;
    struct Node *temp = NULL;
    struct Node *newNode = NULL;

    printf("Enter No of node for DLL :\n");
    scanf("%d",&noOfNode);

    for(int i=0;i<noOfNode;i++){
        newNode = (struct Node*)malloc(sizeof(struct Node));

        printf("Enter data for newNode :\n");
        scanf("%d",&value);
        newNode->data = value;
        newNode->prev = NULL;
        newNode->next = NULL;

        if(head == NULL){
            head = newNode;
        }else{
            temp->next = newNode;
            newNode->prev = temp;
        }

        temp = newNode;
    }
```

```
    //Traversed ddl...
    traversedDDL(head);

    //Reverse doubly linked list..
    head = reverseDoublyLL(head);
    traversedDDL(head);

    return 0;
}
```

RESULT

```
Enter No of node for DLL :
5
Enter data for newNode :
10
Enter data for newNode :
20
Enter data for newNode :
30
Enter data for newNode :
40
Enter data for newNode :
50
Traversed DDL...
10
20
30
40
50
Traversed DDL...
50
40
30
20
10


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

CONCLUSION : From this experiment with linked lists and adjacent pair swapping, I gained practical experience in manipulating linked lists and understanding their node-based structure. By implementing functions to create, traverse, and modify linked lists, including swapping adjacent pairs, I learned how to manage pointers and dynamically allocated memory effectively. The use of a swap function to interchange node values highlighted the importance of pointer manipulation and linked list traversal techniques. This exercise improved my ability to manage linked list operations and will be valuable for solving problems involving dynamic data structures.