Batch: A1	**Roll No.: 16010123012**
Experiment No. 6	
Grade: AA / AB / BB / BC / CC / CD /DD	

Title: Implementation of various types of LL- doubly LL, circular LL, circular doubly LL

Objective: To understand the use of linked lists as data structures for various applications.

Expected Outcome of Experiment:

CO	Outcome
CO 2	Apply linear and non-linear data structure in application development.

Books/ Journals/ Websites referred:

Introduction:

A **Linked List** is a linear data structure where each element, called a node, points to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory blocks. They are dynamic in nature and can grow or shrink during runtime. Linked lists are mainly used when the size of the data is not known in advance or frequent insertion/deletion operations are needed.

Types of linked list:

- **Singly Linked List (SLL):** Each node contains data and a pointer to the next node.
- **Doubly Linked List (DLL):** Each node contains data, a pointer to the next node, and a pointer to the previous node.

- **Circular Linked List (CLL):** In a CLL, the last node points back to the first node, making a circular chain.

- **Circular Doubly Linked List (CDLL):** A combination of DLL and CLL, where both next and previous pointers are used, and the list forms a circle.

Algorithm for creation, insertion, deletion, traversal and searching an element in assigned linked list type:

Creation of a Doubly Linked List:

- Create a node structure with data, previous pointer, and next pointer.
- Set head to NULL.

Insertion at the beginning:

- Create a new node.
- Update the new node's next pointer to the current head.
- If the head is not NULL, update the current head's prev pointer to the new node.
- Set the new node as the new head.

Deletion at the beginning:

- Check if the list is empty.
- Move the head to the next node.
- Update the new head’s prev pointer to NULL.

Traversal (Forward):

- Start from the head.
- Print the data of each node while moving to the next node.

Searching for an element:

- Traverse through the list, comparing each node’s data with the target element.
- If found, return the position or node; otherwise, return "Not Found".

Program:

1. Singly Linked List (SLL) Implementation

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

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

struct Node* head = NULL;

void insertEnd(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
}

void insertFront(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
    head = newNode;
}

void deleteFront() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = head;
    head = head->next;
    free(temp);
}
```

```
void deleteEnd() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = head;
    if (temp->next == NULL) {
        free(temp);
        head = NULL;
    } else {
        while (temp->next->next != NULL)
            temp = temp->next;
        free(temp->next);
        temp->next = NULL;
    }
}

void display() {
    struct Node* temp = head;
    if (temp == NULL) {
        printf("List is empty!\n");
    } else {
        while (temp != NULL) {
            printf("%d\t", temp->data);
            temp = temp->next;
        }
    }
}

int main() {
    int t = -1, value;

    while (t != 6) {
        printf("\n1. Insert End\n2. Insert Front\n3. Delete Front\n4.
Delete End\n5. View\n6. Exit\n");
        scanf("%d", &t);

        switch (t) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insertEnd(value);
                break;
            case 2:
                printf("Enter value: ");
```

```
                scanf("%d", &value);
                insertFront(value);
                break;
            case 3:
                deleteFront();
                break;
            case 4:
                deleteEnd();
                break;
            case 5:
                display();
                break;
            case 6:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid option! Please choose a valid
action.\n");
        }
    }

    return 0;
}
```

2. Doubly Linked List (DLL) Implementation

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

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

struct Node* head = NULL;

void insertFront(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
    newNode->prev = NULL;
    if (head != NULL)
        head->prev = newNode;
    head = newNode;
}
```

```c
void insertEnd(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (head == NULL) {
        newNode->prev = NULL;
        head = newNode;
        return;
    }
    struct Node* temp = head;
    while (temp->next != NULL)
        temp = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
}

void deleteFront() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = head;
    head = head->next;
    if (head != NULL)
        head->prev = NULL;
    free(temp);
}

void deleteEnd() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = head;
    if (temp->next == NULL) {
        head = NULL;
        free(temp);
        return;
    }
    while (temp->next != NULL)
        temp = temp->next;
    temp->prev->next = NULL;
    free(temp);
}
```

```
void display() {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d\t", temp->data);
        temp = temp->next;
    }
}

int main() {
    int t = -1, value;

    while (t != 6) {
        printf("\n1. Insert Front\n2. Insert End\n3. Delete Front\n4.
Delete End\n5. View\n6. Exit\n");
        scanf("%d", &t);

        switch (t) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insertFront(value);
                break;
            case 2:
                printf("Enter value: ");
                scanf("%d", &value);
                insertEnd(value);
                break;
            case 3:
                deleteFront();
                break;
            case 4:
                deleteEnd();
                break;
            case 5:
                display();
                break;
            case 6:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid option! Please choose a valid
action.\n");
        }
    }
    return 0;
}
```

3. Circular Linked List:

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

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

struct Node* head = NULL;

void insertEnd(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        struct Node* temp = head;
        while (temp->next != head)
            temp = temp->next;
        temp->next = newNode;
    }
}

void deleteFront() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = head;
    if (head->next == head) {
        head = NULL;
        free(temp);
        return;
    }
    struct Node* last = head;
    while (last->next != head)
        last = last->next;
    head = head->next;
    last->next = head;
    free(temp);
}
```

```
void display() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node* temp = head;
    do {
        printf("%d\t", temp->data);
        temp = temp->next;
    } while (temp != head);
}

int main() {
    int t = -1, value;

    while (t != 4) {
        printf("\n1. Insert End\n2. Delete Front\n3. View\n4. Exit\n");
        scanf("%d", &t);

        switch (t) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insertEnd(value);
                break;
            case 2:
                deleteFront();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid option! Please choose a valid
action.\n");
        }
    }
    return 0;
}
```

4. Circular Doubly Linked List:

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

```c
#include <stdlib.h>

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

struct Node* head = NULL;

void insertEnd(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if (head == NULL) {
        newNode->next = newNode->prev = newNode;
        head = newNode;
    } else {
        struct Node* last = head->prev;
        newNode->next = head;
        head->prev = newNode;
        newNode->prev = last;
        last->next = newNode;
    }
}

void deleteFront() {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }
    if (head->next == head) {
        free(head);
        head = NULL;
    } else {
        struct Node* last = head->prev;
        struct Node* temp = head;
        head = head->next;
        last->next = head;
        head->prev = last;
        free(temp);
    }
}

void display() {
    if (head == NULL) {
        printf("List is empty!\n");
```

```
        return;
    }
    struct Node* temp = head;
    do {
        printf("%d\t", temp->data);
        temp = temp->next;
    } while (temp != head);
}

int main() {
    int t = -1, value;

    while (t != 4) {
        printf("\n1. Insert End\n2. Delete Front\n3. View\n4. Exit\n");
        scanf("%d", &t);

        switch (t) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insertEnd(value);
                break;
            case 2:
                deleteFront();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid option! Please choose a valid
action.\n");
        }
    }
    return 0;
}
```

Output:-

1.

```
1. Insert End
2. Insert Front
3. Delete Front
4. Delete End
5. View
6. Exit
1
Enter value: 23

1. Insert End
2. Insert Front
3. Delete Front
4. Delete End
5. View
6. Exit
1
Enter value: 243

1. Insert End
2. Insert Front
3. Delete Front
4. Delete End
5. View
6. Exit
1
Enter value: 234

1. Insert End
2. Insert Front
3. Delete Front
4. Delete End
5. View
6. Exit
5
23  243 234
1. Insert End
2. Insert Front
3. Delete Front
4. Delete End
5. View
6. Exit
```

2.

```
1. Insert Front
2. Insert End
3. Delete Front
4. Delete End
5. View
6. Exit
1
Enter value: 753

1. Insert Front
2. Insert End
3. Delete Front
4. Delete End
5. View
6. Exit
1
Enter value: 357

1. Insert Front
2. Insert End
3. Delete Front
4. Delete End
5. View
6. Exit
5
357 753
1. Insert Front
2. Insert End
3. Delete Front
4. Delete End
5. View
6. Exit
```

3.
```
1. Insert End
2. Delete Front
3. View
4. Exit
1
Enter value: 64

1. Insert End
2. Delete Front
3. View
4. Exit
1
Enter value: 65

1. Insert End
2. Delete Front
3. View
4. Exit
2

1. Insert End
2. Delete Front
3. View
4. Exit
1
Enter value: 86

1. Insert End
2. Delete Front
3. View
4. Exit
3
65  86
```

4.
```
1. Insert End
2. Delete Front
3. View
4. Exit
2
List is empty!

1. Insert End
2. Delete Front
3. View
4. Exit
1
Enter value: 432

1. Insert End
2. Delete Front
3. View
4. Exit
1
Enter value: 768

1. Insert End
2. Delete Front
3. View
4. Exit
3
432 768
1. Insert End
2. Delete Front
3. View
4. Exit
4
Exiting...
```

Conclusion:-

In this experiment, we implemented various types of linked lists, including doubly linked lists (DLL), circular linked lists (CLL), and circular doubly linked lists (CDLL). We performed fundamental operations such as insertion, deletion, and traversal, showcasing their dynamic structure and flexibility in memory management.

Post lab questions:

1. Compare and contrast SLL and DLL

Feature	Singly Linked List (SLL)	Doubly Linked List (DLL)
Memory	Requires less memory (only next pointer)	Requires more memory (next and prev pointers)
Traversal	Can only traverse in one direction	Can traverse both forward and backward
Deletion	Requires traversal from the head to delete a node	Deletion can be done efficiently with direct access to the previous node

2. Priority Queue
   A Priority Queue is a type of data structure where each element is assigned a priority, and elements are dequeued based on their priority rather than their insertion order. The element with the highest priority is dequeued first. Priority queues can be implemented using heaps or linked lists.