

## EXPERIMENT NO 1.1:

### TITTLE: TO PERFORM VARIOUS OPERATION ON SINGLE LINKED LIST(SLL).

#### OBJECTIVE:

- To understand and implement various operations on a single linked list in the C programming language.

#### THEORY:

A linked list is a fundamental data structure in computer science and is commonly used to implement other data structures like stacks, queues, and hash tables. It consists of a sequence of elements, called nodes, where each node contains both data and a reference (or link) to the next node in the sequence. The last node typically points to a null reference to signify the end of the list.

Some key aspects and operations related to linked lists:

- **Nodes:** Each node in a linked list contains two parts:
- **Data:** This holds the value or information associated with the node.
- **Next pointer/reference:** This is a reference to the next node in the sequence. In a singly linked list, each node only points to the next node, whereas in a doubly linked list, each node points to both the next and the previous node.
- **Head Pointer:** In many implementations, a linked list is managed via a reference to the first node called the head pointer. This head pointer allows traversal through the list starting from the first node.

#### Operations:

Insertion: Adding a new node to the linked list.

Deletion: Removing a node from the linked list.

Traversal: Iterating through the list to access or manipulate nodes.

Searching: Finding a specific element within the list.

Reversal: Changing the order of the nodes in the list

#### IMPLEMENTATION WITH C:

**Program 1:** Perform various operation on Single Linked List.

##### Source code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node{ // structure
    int data;
    struct Node *next;
};
struct Node *head = NULL;
void insertAtBeginning(int value){ //insert at beginning
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node)); //size allocation for a node.
    if(head==NULL){
        head=newnode;}else{
        newNode->data = value;
        newNode->next = head;
        head = newNode;}}
//Insertion
```

```

void insertAtEnd(int value){ //insert at end
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
        return; }
    struct Node *temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;}

void insertFromSpecificPosition(int value, int position){ //insert at specific position
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    if (position == 1){
        newNode->next = head;
        head = newNode;
        return; }
    struct Node *temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++){
        temp = temp->next;
    }
    if (temp == NULL){
        printf("Invalid position. Insertion failed.\n");
        free(newNode);
    }
    newNode->next = temp->next;
    temp->next = newNode;}

//Deletion
void deleteFromBeginning(){
    if (head == NULL){
        printf("Linked list is empty. Nothing to delete.\n");
    }
    struct Node *temp = head;
    head = head->next;
    free(temp);}

void deleteFromEnd(){//Delete from end
    if (head == NULL){
        printf("Linked list is empty. Nothing to delete.\n");
        return;
    } if (head->next == NULL){
        free(head);
        head = NULL;
    }
    struct Node *temp = head;
    while (temp->next->next != NULL){
        temp = temp->next; }
    free(temp->next);
    temp->next = NULL;
}

void deleteFromSpecificPosition(int position){
    if (head == NULL){
        printf("Linked list is empty. Nothing to delete.\n");
    }
    if (position == 1){
        struct Node *temp = head;
        head = head->next;
        free(temp); }
    struct Node *temp = head;
    struct Node *prev = NULL;
    for (int i = 1; i < position && temp != NULL; i++){
        prev = temp;

```

```

    temp = temp->next; }
if (temp == NULL){
    printf("Invalid position. Deletion failed.\n");
    return; }
prev->next = temp->next;
free(temp);}
void printLinkedList(){
    struct Node *temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
}
int main(){
    int choice, value, position;
    while (1){
        printf("\n\n----- Single Linked List Operations ----- \n\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Insert at specific position\n");
        printf("4. Delete from beginning\n");
        printf("5. Delete from end\n");
        printf("6. Delete from specific position\n");
        printf("7. Print linked list\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice){
            case 1:
                printf("Enter value to insert at the beginning: ");
                scanf("%d", &value);
                insertAtBeginning(value);
                break;
            case 2:
                printf("Enter value to insert at the end: ");
                scanf("%d", &value);
                insertAtEnd(value);
                break;
            case 3:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                printf("Enter position to insert at: ");
                scanf("%d", &position);
                insertFromSpecificPosition(value, position);
                break;
            case 4:
                deleteFromBeginning();
                break;
            case 5:
                deleteFromEnd();
                break;
            case 6:
                printf("Enter position to delete from: ");
                scanf("%d", &position);
                deleteFromSpecificPosition(position);
                break;
            case 7:
                printf("Linked list: ");
                printLinkedList();
                break;
            case 8:
                printf("Exiting program.\n");
                exit(0);
            default:
                printf("Invalid choice! Please enter a valid option.\n");} }
    return 0;}

```

## OUTPUT:

### 1) Insertion

```
----- Single Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 1
Enter value to insert at the beginning: 1

----- Single Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position

----- Single Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 3
Enter value to insert: 3
Enter position to insert at: 3

----- Single Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 2
Enter value to insert at the end: 2

----- Single Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 7
Linked list: 1 -> 2 -> 3 -> NULL
```

### 2) Deletion

```
----- Single Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 4

----- Single Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 5

----- Single Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 6
Enter position to delete from: 2
Invalid position. Deletion failed.

----- Single Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 7
Linked list: 2 -> NULL
```

## DISCUSSION:

To understand the single linked list, we execute two fundamental operations: insertion and deletion, covering various scenarios using C programming. We create separate functions for different cases and utilize a switch statement for user-driven selection. User can choose his/her choice according to the given option provided in above program. All cases are handle according to the given instruction.

## CONCLUSION:

The above experiment in C programming, we have understood the concept of single linked list along with different cases and how it works.

## EXPERIMENT NO 1.2:

### TITTLE: TO PERFORM VARIOUS OPERATION ON DOUBLY LINKED LIST(DLL).

#### OBJECTIVE:

- To understand and implement various operations on a doubly linked list in the C programming language.

#### THEORY:

A doubly linked list is a type of linked list data structure where each node contains two references (pointers): one pointing to the next node in the sequence and another pointing to the previous node. This bidirectional linkage allows for more efficient traversal in both directions, unlike singly linked lists that only support forward traversal. The first node in a doubly linked list is called the head, while the last node points to null, indicating the end of the list. This structure enables easy insertion and deletion operations at both ends and at any position within the list, as it only requires updating the pointers of adjacent nodes. While doubly linked lists offer advantages such as bidirectional traversal and better flexibility in insertion and deletion, they require more memory compared to singly linked lists due to the extra pointer in each node. Doubly linked lists find application in scenarios where frequent traversal in both directions and dynamic data modifications are essential, making them useful for implementing data structures like queues, stacks, and linked hash tables.

#### IMPLEMENTATION WITH C:

**Program 1:** Perform various operation on Double Linked List.

##### Source code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};
struct Node* head = NULL;
void insertAtBeginning(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = head;
    if (head != NULL)
        head->prev = newNode;
    head = newNode;
}
//Insertion
void insertAtEnd(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;
    }
    struct Node* temp = head;
    while (temp->next != NULL)
        temp = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
}

void insertFromSpecificPosition(int value, int position) {
    if (position < 1) {
        printf("Invalid position. Insertion failed.\n");
    }
    if (position == 1) {
        insertAtBeginning(value);
    }
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    struct Node* temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++)
        temp = temp->next;
    if (temp == NULL) {
        printf("Invalid position. Insertion failed.\n");
    }
    newNode->next = temp->next;
    newNode->prev = temp;
    if (temp->next != NULL)
        temp->next->prev = newNode;
    temp->next = newNode;
}

//Deletion
void deleteFromBeginning() {
    if (head == NULL) {
        printf("Linked list is empty. Nothing to delete.\n");
    }
    struct Node* temp = head;
    head = head->next;
    if (head != NULL)
        head->prev = NULL;
    free(temp);
}

void deleteFromEnd() {
    if (head == NULL) {
        printf("Linked list is empty. Nothing to delete.\n");
    }
    struct Node* temp = head;
    while (temp->next != NULL)
        temp = temp->next;
    if (temp->prev != NULL)
        temp->prev->next = NULL;
    else
        head = NULL;
    free(temp);
}

void deleteFromSpecificPosition(int position) {
    if (position < 1 || head == NULL) {
        printf("Invalid position or empty list. Deletion failed.\n");
    }
}

```

```

    struct Node* temp = head;
    if (position == 1) {
        head = head->next;
        if (head != NULL)
            head->prev = NULL;
        free(temp);
    }
    for (int i = 1; temp != NULL && i < position; i++)
        temp = temp->next;
    if (temp == NULL) {
        printf("Invalid position. Deletion failed.\n");
    }
    if (temp->next != NULL)
        temp->next->prev = temp->prev;
    temp->prev->next = temp->next;
    free(temp);
}

void printData() {
    struct Node* temp = head;
    printf("Forward: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    int choice, value, position;
    while (1) {
        printf("\n\n----- Double Linked List Operations ----- \n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Insert at specific position\n");
        printf("4. Delete from beginning\n");
        printf("5. Delete from end\n");
        printf("6. Delete from specific position\n");
        printf("7. Print linked list \n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to insert at the beginning: ");
                scanf("%d", &value);
                insertAtBeginning(value);
                break;
            case 2:
                printf("Enter value to insert at the end: ");
                scanf("%d", &value);
                insertAtEnd(value);
                break;
            case 3:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                printf("Enter position to insert at: ");
                scanf("%d", &position);
                insertFromSpecificPosition(value, position);
                break;
            case 4:
                deleteFromBeginning();
                break;
            case 5:
                deleteFromEnd();
                break;
        }
    }
}

```

```

case 6:
    printf("Enter position to delete from: ");
    scanf("%d", &position);
    deleteFromSpecificPosition(position);
    break;
case 7:
    printData();
    break;
case 8:
    printf("Exiting program.\n");
    exit(0);
default:
    printf("Invalid choice! Please enter a valid option.\n");
} }return 0;}

```

## OUTPUT:

### 1) Insertion

```

----- Double Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 1
Enter value to insert at the beginning: 1

```

```

----- Double Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 2
Enter value to insert at the end: 2

```

```

----- Double Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 3
Enter value to insert: 3
Enter position to insert at: 3

```

```

----- Double Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 7
Forward: 1 -> 2 -> 3 -> NULL

```

### 2) Deletion

```

----- Double Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 4

```

```

----- Double Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 5

```

```

----- Double Linked List Operations -----
1. Insert at beginning
2. Insert at end
3. Insert at specific position
4. Delete from beginning
5. Delete from end
6. Delete from specific position
7. Print linked list
8. Exit
Enter your choice: 7
Forward: 2 -> NULL

```

## DISCUSSION:

To understand the doubly linked list, we execute two fundamental operations: insertion and deletion, covering various scenarios using C programming. We create separate functions for different cases and utilize a switch statement for user-driven selection. User can choose his/her choice according to the given option provided in above program. All cases are handle according to the given instruction.

## CONCLUSION:

The above experiment in C programming, we have understood the concept of double linked list along with different cases and how it works.



## EXPERIMENT NO 1.3:

### TITTLE: TO PERFORM VARIOUS OPERATION ON SINGLE CIRCULAR LINKED LIST(SCLL).

#### OBJECTIVE:

- To understand and implement various operations on a single circular linked list in the C programming language.

#### THEORY:

A circular linked list is a variation of the linked list data structure where the last node points back to the first node, forming a circular loop. Instead of having the last node pointing to null, like in traditional linked lists, its "next" pointer is connected to the first node, creating a circular chain. This circular arrangement enables seamless traversal of the list from any node to any other node. Each node in the list contains data and a pointer to the next node.

A Single Circular Linked List (SCLL) is a type of linked list where each node has two parts: data and a pointer to the next node. What makes SCLL unique is that the last node in the list points back to the first node, forming a circular loop. This loop allows for seamless traversal through the list from any node to any other node. SCLLs are commonly used in situations where continuous iteration or rotation through the list is needed, and they offer advantages such as efficient storage and manipulation of data. However, they also require careful handling to prevent infinite loops and other errors during operations like insertion and deletion.

#### IMPLEMENTATION WITH C:

**Program 1:** Perform various operation on Single Circular Linked List.

##### Source code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;}

//insertion
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        newNode->next = newNode; // Pointing to itself
        *head = newNode;
    } else {
        struct Node* current = *head;
        while (current->next != *head) {
            current = current->next;
        }
        current->next = newNode;
        newNode->next = *head;
    }
}
```

```

*head = newNode; }}

void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        newNode->next = newNode; // Pointing to itself
        *head = newNode;
    } else {
        struct Node* current = *head;
        while (current->next != *head) {
            current = current->next;
        }
        current->next = newNode;
        newNode->next = *head;
    }
}

void insertAtPosition(struct Node** head, int data, int position) {
    if (position <= 0) {
        printf("Invalid position.\n");
        return; }
    if (*head == NULL || position == 1) {
        insertAtBeginning(head, data);
        return;
    }
    struct Node* newNode = createNode(data);
    struct Node* current = *head;
    int currentPosition = 1;
    while (current->next != *head && currentPosition < position - 1) {
        current = current->next;
        currentPosition++;
    }
    newNode->next = current->next;
    current->next = newNode;
}

//deletion
void deleteNode(struct Node** head, int key) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return; }
    struct Node *temp = *head, *prev = NULL;
    if (temp->data == key) {
        while (temp->next != *head)
            temp = temp->next;
        temp->next = (*head)->next;
        free(*head);
        *head = temp->next;
        return;
    }
    while (temp->next != *head && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }
    if (temp->data != key) {
        printf("Key not found in the list.\n");
        return;
    }
    prev->next = temp->next;
    free(temp);}

void displayData(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
}

```

```

    struct Node* current = head;
    do {
        printf("%d ", current->data);
        current = current->next;
    } while (current != head);
    printf("\n");
}

int main() {

    struct Node* head = NULL;
    int choice, data, position;

    do {

        printf("\nSingly Circular Linked List Operations:\n");
        printf("1. Insert at the beginning\n");
        printf("2. Insert at the end\n");
        printf("3. Insert at a specific position\n");
        printf("4. Delete a node\n");
        printf("5. Display the list\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the data to insert at the beginning: ");
                scanf("%d", &data);
                insertAtBeginning(&head, data);
                break;
            case 2:
                printf("Enter the data to insert at the end: ");
                scanf("%d", &data);
                insertAtEnd(&head, data);
                break;
            case 3:
                printf("Enter the data to insert: ");
                scanf("%d", &data);
                printf("Enter the position to insert at: ");
                scanf("%d", &position);
                insertAtPosition(&head, data, position);
                break;
            case 4:
                printf("Enter the data of the node to delete: ");
                scanf("%d", &data);
                deleteNode(&head, data);
                break;
            case 5:
                printf("Singly Circular linked list: ");
                displayData(head);
                break;
            case 6:
                printf("Exiting the program.\n");
                break;
            default:
                printf("Invalid choice, please try again.\n"); }
        } while (choice != 6);
    return 0
}

```

## OUTPUT:

### 1) Insertion

<pre>Singly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 1 Enter the data to insert at the beginning: 3</pre>	<pre>Singly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 1 Enter the data to insert at the beginning: 2</pre>
<pre>Singly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 1 Enter the data to insert at the beginning: 1</pre>	<pre>Singly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 5 Singly Circular linked list: 1 2 3</pre>

### 2) Deletion

<pre>Singly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 4 Enter the data of the node to delete: 2</pre>	<pre>Singly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 5 Singly Circular linked list: 1 3</pre>
---	--

## DISCUSSION:

To understand the single circular linked list we perform 2 operations i.e. insertion and deletion along with their different cases using c programming. First we can create a different function of different cases then use the switch statement to call the different cases. User can choose his/her choice according to the given option provided in above program. All cases are handle according to the given instruction.

## CONCLUSION:

The above experiment in C programming, we have understood the concept of single circular linked list along with different cases and how it works.

## EXPERIMENT NO 1.4:

### TITTLE: TO PERFORM VARIOUS OPERATION ON DOUBLY CIRCULAR LINKED LIST(DCLL).

#### OBJECTIVE:

- To understand and implement various operations on a doubly circular linked list in the C programming language.

#### THEORY:

A circular linked list is a variation of the linked list data structure where the last node points back to the first node, forming a circular loop. Instead of having the last node pointing to null, like in traditional linked lists, its "next" pointer is connected to the first node, creating a circular chain.

The circular linked list can be either singly circular linked list, where each node has a single reference to the next node, or doubly circular linked list, where each node has both "next" and "previous" pointers, forming a bidirectional circular structure.

The circular linked list offers some unique advantages. For example, it simplifies certain operations, such as traversing a list from the last node back to the first node without needing to maintain extra variables. Additionally, it ensures that there is always a valid "next" node for any node, even if there is only one node in the list.

#### IMPLEMENTATION WITH C:

**Program 1:** Perform various operation on Doubly Circular Linked List.

##### Source code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

//insertion
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        newNode->next = newNode;
        newNode->prev = newNode;
    } else {
        newNode->next = *head;
        newNode->prev = (*head)->prev;
        (*head)->prev->next = newNode;
        (*head)->prev = newNode;
    }
    *head = newNode;
}
```

```

void insertAtEnd(struct Node** head, int data) {
    if (*head == NULL) {
        insertAtBeginning(head, data);
        return;
    }
    struct Node* newNode = createNode(data);
    newNode->next = *head;
    newNode->prev = (*head)->prev;
    (*head)->prev->next = newNode;
    (*head)->prev = newNode;
}

void insertAtPosition(struct Node** head, int data, int position) {
    if (position <= 0) {
        insertAtBeginning(head, data);
        return;
    }
    struct Node* newNode = createNode(data);
    struct Node* current = *head;
    int currentPosition = 1;

    while (current != NULL && currentPosition < position) {
        current = current->next;
        currentPosition++;
    }
    if (current == NULL) {
        printf("Position is out of range.\n");
        return;
    }
    newNode->prev = current;
    newNode->next = current->next;
    if (current->next != NULL) {
        current->next->prev = newNode;
    }
    current->next = newNode;}

//Deletion
void deleteNode(struct Node** head, int key) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = *head;
    while (temp->data != key) {
        temp = temp->next;
        if (temp == *head) {
            printf("Element not found in the list.\n");
            return; } }
    if (temp->next == temp) {
        *head = NULL;
    } else {
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
        if (temp == *head) {
            *head = temp->next;
        }
    }
    free(temp);}

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

```

```

    printf("\n");
}
int main() {
    struct Node* head = NULL;
    int choice, data, position;
    do {
        printf("\nDoubly Circular Linked List Operations:\n");
        printf("1. Insert at the beginning\n");
        printf("2. Insert at the end\n");
        printf("3. Insert at a specific position\n");
        printf("4. Delete a node\n");
        printf("5. Display the list\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the data to insert at the beginning: ");
                scanf("%d", &data);
                insertAtBeginning(&head, data);
                break;
            case 2:
                printf("Enter the data to insert at the end: ");
                scanf("%d", &data);
                insertAtEnd(&head, data);
                break;
            case 3:
                printf("Enter the data to insert: ");
                scanf("%d", &data);
                printf("Enter the position to insert at: ");
                scanf("%d", &position);
                insertAtPosition(&head, data, position);
                break;
            case 4:
                printf("Enter the data of the node to delete: ");
                scanf("%d", &data);
                deleteNode(&head, data);
                break;
            case 5:
                printf("Doubly Circular linked list: ");
                displayData(head);
                break;
            case 6:
                printf("Exiting the program.\n");
                break;
            default:
                printf("Invalid choice, please try again.\n");
        } } while (choice != 6);
    return 0;
}
;
}

```

## OUTPUT:

### 1) Insertion

Doubly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 1 Enter the data to insert at the beginning: 3	Doubly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 1 Enter the data to insert at the beginning: 2	Doubly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 1 Enter the data to insert at the beginning: 1
---	---	---

Doubly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 5 Doubly Circular linked list: 1 2 3
---

### 2) Deletion

Doubly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 4 Enter the data of the node to delete: 2	Doubly Circular Linked List Operations: 1. Insert at the beginning 2. Insert at the end 3. Insert at a specific position 4. Delete a node 5. Display the list 6. Exit Enter your choice: 5 Doubly Circular linked list: 1 3
--	---

## DISCUSSION:

Understanding doubly circular linked lists involves mastering two key operations: insertion and deletion. By creating separate functions to handle different cases, such as insertion at the beginning, end, or specific positions, and deletion of nodes with various scenarios, users can seamlessly manipulate the linked list structure. Utilizing a switch statement enhances user interaction, allowing for easy selection of operations. This approach simplifies navigation through different functionalities, promoting a clear and intuitive user experience in C programming.

## CONCLUSION:

The above experiment in C programming, we have understood the concept of Doubly Circular linked list along with different cases and how it works.



## EXPERIMENT NO 2.1:

### TITLE: TO IMPLEMENT STACK OPERATION WITH ARRAY.

#### OBJECTIVE:

- To implement stack operations using arrays in C programming language.

#### THEORY:

A stack is a linear data structure that follows the Last in, First Out (LIFO) principle, where elements are added and removed from the same end, known as the top of the stack. Stacks are used to manage function calls and local variables in programming languages, track execution context, and implement undo-redo functionality. Stacks facilitate efficient storage and retrieval, making them useful for managing structured data and controlling program flow.

#### Operations:

**Push:** Adds an element to the top of the stack.

**Pop:** Removes the top element from the stack.

**isFull:** Checks if the stack is full.

**isEmpty:** Checks if the stack is empty.

The stack can be implemented using various data structures such as arrays, linked lists, or dynamic arrays. In this experiment, we will implement stack operations using an array.

#### IMPLEMENTATION WITH C:

**Program 1:** Perform various operation on Stack using array.

##### Source code:

```
#include<stdio.h>
#include<stdbool.h>
#define max 10
int stack[max];
int tos = -1;

bool isEmpty() {
    return (tos == -1);
}

bool isFull() {
    return (tos == max - 1);
}

void push(int data) {
    if (tos == max - 1) {
        printf("Stack Overflow");
    } else {
        tos++;
        stack[tos] = data;
    }
}

int pop() {
```

```

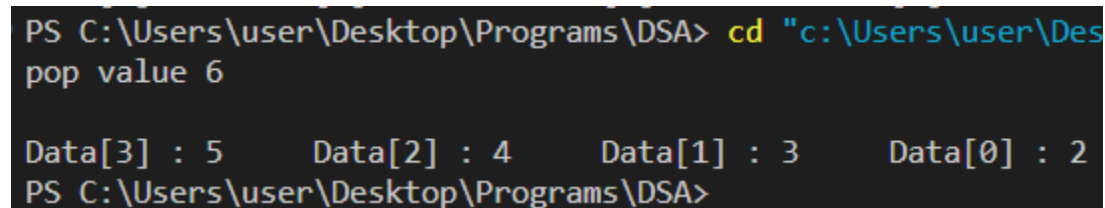
    if (tos == -1) {
        printf("Stack UnderFlow");
    } else {
        int popValue = stack[tos];
        tos--;
        return popValue;
    }
}

void display() {
    if (tos == -1) {
        printf("Stack has no value");
    } else {
        for (int i = tos; i >= 0; i--) {
            printf("Data[%d] : %d\t", i, stack[i]);
        }
    }
}

int main() {
    push(2);
    push(3);
    push(4);
    push(5);
    push(6);
    int data = pop();
    printf("Pop value %d \n\n", data);
    display();
    return 0;}

```

## OUTPUT:



```

PS C:\Users\user\Desktop\Programs\DSA> cd "c:\Users\user\Desktop\Programs\DSA"
pop value 6

Data[3] : 5    Data[2] : 4    Data[1] : 3    Data[0] : 2
PS C:\Users\user\Desktop\Programs\DSA>

```

## DISCUSSION:

In this experiment, we implemented a stack data structure using an array in the C programming language. We defined functions to perform stack operations such as push, pop, isFull, and isEmpty. We initialized the stack with a given size and used a simple integer array as the underlying data structure. During the demonstration, we pushed multiple elements onto the stack, printed the stack, and displayed the top element. After popping an element from the stack, we printed the modified stack.

## CONCLUSION:

Through this experiment, we successfully implemented stack operations using an array. We observed how the LIFO principle applies to stack operations and how arrays can be used as the underlying data structure for implementing a stack.

## EXPERIMENT NO 2.2:

### TITTLE: TO IMPLEMENT STACK OPERATION WITH LINKED LIST.

#### OBJECTIVE:

- To implement stack operations using Linked List in C programming language.

#### THEORY:

A stack is a linear data structure that follows the Last in, First Out (LIFO) principle, where elements are added and removed from the same end, known as the top of the stack. Stacks are used to manage function calls and local variables in programming languages, track execution context, and implement undo-redo functionality. Stacks facilitate efficient storage and retrieval, making them useful for managing structured data and controlling program flow. Linked list is a data structure consisting of a collection of nodes where each node contains a data field and a reference (link) to the next node in the sequence. Using linked lists for implementing a stack allows for dynamic memory allocation and efficient insertion and deletion of elements.

#### Operations:

**Push:** Adds an element to the top of the stack.

**Pop:** Removes the top element from the stack.

**Peek:** returns the value of the top element without removing it.

**isEmpty:** Checks if the stack is empty.

#### IMPLEMENTATION WITH C:

**Program 1:** Perform various operation on Stack using Linked List.

##### Source code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* top = NULL;
void push(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = top;
    top = newNode;
}
int pop() {
    if (top == NULL) {
        printf("Stack is empty. Cannot pop.\n");
        return -1; }
    int poppedData = top->data;
    struct Node* temp = top;
    top = top->next;
    return poppedData;}
int peek() {
    if (top == NULL) {
        printf("Stack is empty.\n");
        return -1; }
```

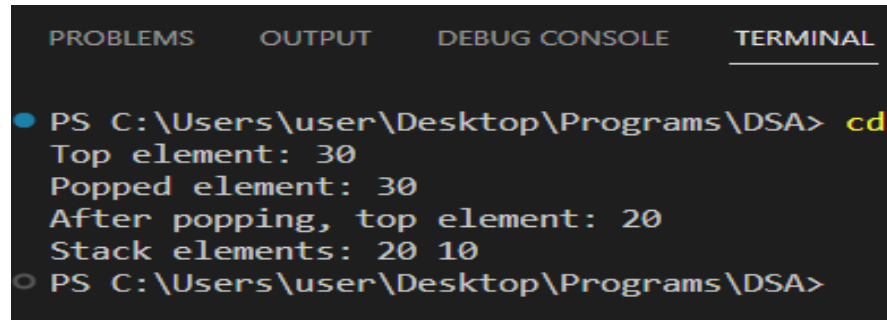
```

    return top->data;}
int isEmpty() {
    return top == NULL;
}
void display() {
    struct Node* current = top;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    } printf("\n");}

int main() {
    push(10);
    push(20);
    push(30);
    printf("Top element: %d\n", peek());
    int poppedData = pop();
    printf("Popped element: %d\n", poppedData);
    printf("After popping, top element: %d\n", peek());
    printf("Stack elements: ");
    display();
    return 0;}

```

## OUTPUT:



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\user\Desktop\Programs\DSA> cd
Top element: 30
Popped element: 30
After popping, top element: 20
Stack elements: 20 10
PS C:\Users\user\Desktop\Programs\DSA>

```

## DISCUSSION:

In this experiment, we implemented a stack data structure using a linked list in the C programming language. We defined functions to perform stack operations such as push, pop, peek, and isEmpty. Instead of using an integer array as the underlying data structure, we utilized a linked list composed of nodes, where each node contains an integer value and a reference to the next node.

During the demonstration, we pushed multiple elements onto the stack, printed the stack contents, and displayed the top element using the peek function. After popping an element from the stack, we printed the modified stack to reflect the updated state.

## CONCLUSION:

Through this experiment, we successfully implemented stack operations using an linked list. We observed how the LIFO principle applies to stack operations and how arrays can be used as the underlying data structure for implementing a stack.

## EXPERIMENT NO :3.1

### TITTLE: TO IMPLEMENT QUEUE OPERATIONS WITH ARRAY

#### OBJECTIVE:

- To implement the basic operations of a queue data structure using an array.

#### THEORY:

A queue is a linear data structure that follows the First In First Out (FIFO) principle, meaning that the element inserted first is the one that is removed first. Queues can be implemented using arrays or linked lists. In this experiment, we focus on implementing a queue using an array.

#### Operations:

The basic operations of a queue are as follows:

**Enqueue:** Add an element to the rear of the queue.

**Dequeue:** Remove an element from the front of the queue.

**isEmpty:** Check if the queue is empty.

**Display:** Display the elements of the queue.

For implementing these operations with an array, we need to maintain two pointers: 'front' and 'rear'. 'Front' points to the first element of the queue, and 'rear' points to the last element.

#### IMPLEMENTATION WITH C:

##### Program 1: Source code:

```
#include <stdio.h>
#define SIZE 5
int queue[SIZE];
int front = -1, rear = -1;
void enqueue(int value) {
    if (rear == SIZE - 1) {
        printf("Queue is full. Cannot enqueue.\n");
        return;
    }
    if (front == -1)
        front = 0;
    rear++;
    queue[rear] = value;}
int dequeue() {
    int value;
    if (front == -1 || front > rear) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }
    value = queue[front];
    front++;
    return value;}
int isEmpty() {
    if (front == -1 || front > rear)
        return 1;
    else
        return 0;}
int isFull() {
    if (rear == SIZE - 1)
        return 1;
    else
        return 0;}
void display() {
    if (isEmpty()) {
```

```

        printf("Queue is empty.\n");
        return; }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++)
        printf("%d ", queue[i]);
    printf("\n");}
int main() {
    enqueue(1);
    enqueue(2);
    enqueue(3);
    display();
    printf("Dequeued element: %d\n", dequeue());
    display();
    printf("Is queue empty? %s\n", isEmpty() ? "Yes" : "No");
    printf("Is queue full? %s\n", isFull() ? "Yes" : "No");
    return 0;}

```

## OUTPUT:

```

Output

/tmp/BtGeXcNO10.o
Queue elements: 1 2 3
Dequeued element: 1
Queue elements: 2 3
Is queue empty? No
Is queue full? No

=== Code Execution Successful ===

```

## DISCUSSION:

This experiment involved implementing Queue operations using an array in C. Through functions like enqueue, dequeue, isEmpty, and isFull, we managed to maintain a FIFO structure. By tracking the front and rear ends with pointers, we ensured proper element addition and removal. This approach demonstrated the efficiency of arrays in managing sequential data structures like queues.

## CONCLUSION:

Through this experiment, we successfully implemented basic Queue operations using an array in the C programming language. We observed the FIFO principle, where elements are added to the rear end and removed from the front end of the queue.

## EXPERIMENT NO :3.2

### TITLE: TO IMPLEMENT CIRCULAR QUEUE OPERATIONS WITH ARRAY OBJECTIVE:

- To implement circular queue operations using an array in the C programming language.

### THEORY:

Circular Queue is a linear data structure in which operations are performed based on the FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. The key operations of Circular Queue are:

#### Operations:

**enqueue():** Adds an element to the rear of the queue.

**dequeue():** Removes an element from the front of the queue.

**isFull():** Checks if the queue is full.

**isEmpty():** Checks if the queue is empty.

**display():** Displays the elements of the queue.

### IMPLEMENTATION WITH C:

#### Program 1: Source code:

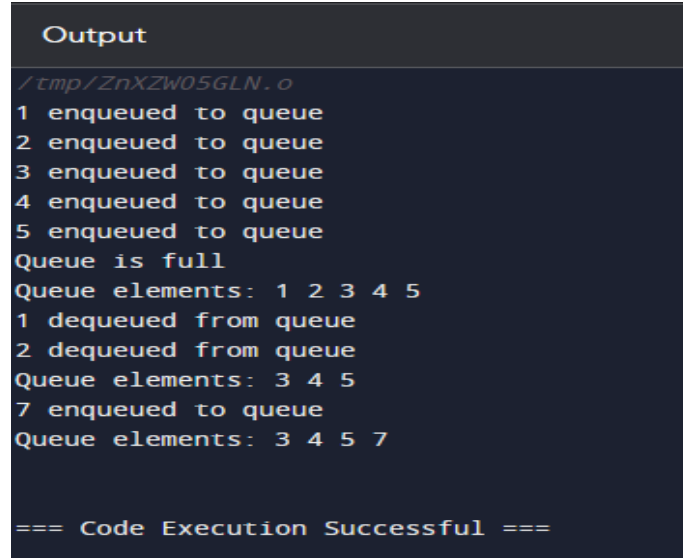
```
#include <stdio.h>
#define SIZE 5
int queue[SIZE];
int front = -1, rear = -1;
void enqueue(int value) {
    if ((rear + 1) % SIZE == front) {
        printf("Queue is full\n");
    } else {
        if (front == -1)
            front = 0;
        rear = (rear + 1) % SIZE;
        queue[rear] = value;
        printf("%d enqueued to queue\n", value);
    }
}
void dequeue() {
    if (front == -1) {
        printf("Queue is empty\n");
    } else {
        printf("%d dequeued from queue\n", queue[front]);
        if (front == rear) {
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % SIZE;
        }
    }
}
void display() {
    if (front == -1)
        printf("Queue is empty\n");
    else {
        int i = front;
        printf("Queue elements: ");
        do {
            printf("%d ", queue[i]);
            i = (i + 1) % SIZE;
        } while (i != front);
        printf("\n");
    }
}
```

```

    } while (i != (rear + 1) % SIZE);
    printf("\n");}}
int main() {
    enqueue(1);
    enqueue(2);
    enqueue(3);
    enqueue(4);
    enqueue(5);
    enqueue(6); // Trying to enqueue when queue is full
    display();
    dequeue();
    dequeue();
    display();
    enqueue(7);
    display();
    return 0;
}

```

## OUTPUT:



```

Output
/tmp/ZnXZW05GLN.o
1 enqueued to queue
2 enqueued to queue
3 enqueued to queue
4 enqueued to queue
5 enqueued to queue
Queue is full
Queue elements: 1 2 3 4 5
1 dequeued from queue
2 dequeued from queue
Queue elements: 3 4 5
7 enqueued to queue
Queue elements: 3 4 5 7

=== Code Execution Successful ===

```

## DISCUSSION:

The lab experiment implemented circular queue operations using an array in C, efficiently handling the circular nature of the queue with modulo arithmetic. Key operations like enqueue, dequeue, and display were demonstrated, along with handling queue full or empty scenarios. Circular queues offer efficient memory usage and sequential data processing. This experiment provided practical insight into their implementation in a concise manner.

## CONCLUSION:

The lab experiment successfully implemented circular queue operations in C, showcasing their efficient handling of memory and sequential data processing. Through key operations like enqueue and dequeue, along with appropriate handling of edge cases, the experiment provided practical understanding of circular queues' implementation.



## EXPERIMENT NO :3.3

### TITTLE:

### OBJECTIVE:

- To implement priority queue operations using an array-based data structure.

### THEORY:

Priority queue is a variation of a queue where each element has a priority associated with it. The element with the highest priority is dequeued first. Priority queues are commonly implemented using heaps, but for simplicity, we'll implement it using arrays. The basic operations of a priority queue are:

Insertion: Add an element with a specified priority.

Deletion: Remove the element with the highest priority.

### IMPLEMENTATION WITH C:

#### Program 1: Source code:

```
#include <stdio.h>
#define MAX_SIZE 100
// Structure to represent a priority queue element
struct PriorityQueueElement {
    int data;
    int priority;
};
// Structure to represent a priority queue
struct PriorityQueue {
    struct PriorityQueueElement elements[MAX_SIZE];
    int size;
};
// Function to initialize the priority queue
void initPriorityQueue(struct PriorityQueue *pq) {
    pq->size = 0;
}
// Function to insert an element into the priority queue
void insert(struct PriorityQueue *pq, int data, int priority) {
    if (pq->size == MAX_SIZE) {
        printf("Priority Queue Overflow\n");
        return;
    }
    int i = pq->size - 1;
    while (i >= 0 && pq->elements[i].priority < priority) {
        pq->elements[i + 1] = pq->elements[i];
        i--;
    }
    pq->elements[i + 1].data = data;
    pq->elements[i + 1].priority = priority;
    pq->size++;
}
// Function to delete the element with the highest priority from the priority queue
int deleteMax(struct PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Priority Queue Underflow\n");
        return -1; // return an error value
    }
    int maxData = pq->elements[0].data;
    for (int i = 1; i < pq->size; i++) {
        pq->elements[i - 1] = pq->elements[i];
    }
    pq->size--;
    return maxData;
}
```

```
// Function to print the elements of the priority queue
void printPriorityQueue(struct PriorityQueue *pq) {
    printf("Priority Queue: ");
    for (int i = 0; i < pq->size; i++) {
        printf("(%d, %d) ", pq->elements[i].data, pq->elements[i].priority);
    }
    printf("\n");
}

// Main function to demonstrate priority queue operations
int main() {
    struct PriorityQueue pq;
    initPriorityQueue(&pq);
    insert(&pq, 10, 3);
    insert(&pq, 20, 2);
    insert(&pq, 30, 5);
    insert(&pq, 40, 1);
    printf("After Insertions:\n");
    printPriorityQueue(&pq);
    int maxElement = deleteMax(&pq);
    printf("Deleted Element with highest priority: %d\n", maxElement);
    printf("After Deletion:\n");
    printPriorityQueue(&pq);
    return 0;
}
```

## OUTPUT:

```
Output

/tmp/63oleP2t1L.o
After Insertions:
Priority Queue: (30, 5) (10, 3) (20, 2) (40, 1)
Deleted Element with highest priority: 30
After Deletion:
Priority Queue: (10, 3) (20, 2) (40, 1)

=== Code Execution Successful ===
```

## DISCUSSION:

In this implementation, we used an array to represent the priority queue. When an element is inserted, it is placed in the appropriate position based on its priority. Elements with higher priority are placed closer to the beginning of the array. When an element needs to be deleted, the element with the highest priority (located at the beginning of the array) is removed.

## CONCLUSION:

In conclusion, we have successfully implemented priority queue operations using an array-based data structure in C language.

## EXPERIMENT NO : 4.1

### TITTLE: TO IMPLEMENT VARIOUS ITERATIVE SORTING ALGORITHMS

#### OBJECTIVE:

- To study the concept of Iterative Sorting and implement the Insertion Sort algorithm.

#### THEORY:

Iterative Sorting is a class of sorting algorithms that sort data by repeatedly stepping through the list, comparing adjacent elements and swapping them if they are in the wrong order. Insertion Sort is an iterative sorting algorithm that builds the final sorted array one element at a time. It works by iterating through the array, removing one element at a time, and inserting it into the correct position in the already-sorted portion of the array.

#### Algorithm:

1. Start with the second element (key) in the array.
  2. Compare the key with the elements to its left, moving from right to left.
  3. If the key is smaller than the current element being compared, shift the current element to the right.
  4. Repeat step iii until the key is greater than or equal to the current element or until reaching the beginning of the array.
  5. Insert the key into the correct position in the sorted sequence.
- Move to the next element in the array and repeat steps ii to v until all elements are sorted.

#### IMPLEMENTATION WITH C:

##### Program 1: Source code:

```
#include <stdio.h>
void insertionSort(int arr[], int n) {
    int i, j, key;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int n, i;
    int arr[n];
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Invalid array size. Exiting...\n");
        return 0;
    }
    printf("Enter the elements of the array:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    insertionSort(arr, n);
}
```

```
printf("The sorted array is:\n");
for (i = 0; i < n; i++) {
    printf("%d ", arr[i]);
}
printf("\n");
return 0;
}
```

## OUTPUT:

```
Output
/tmp/mEzBhppVX8.o
Enter the size of the array: 5
Enter the elements of the array:
10 2 9 18 34
The sorted array is:
2 9 10 18 34

=== Code Execution Successful ===
```

## DISCUSSION:

In this lab, we've successfully implemented an insertion sort to display the iterative sorting using the C programming language. In the above output, first user was asked to enter the size of array and after that a certain no of elements were given(10,2,9,18,34). After that, the array was sorted using the logic of Insertion Sort algorithm and display the result. The program effectively demonstrates the logic of iterative sorting algorithm using Insertion sort.

## CONCLUSION:

Through this experiment, we have gained a comprehensive understanding of Iterative Sorting and the Insertion Sort algorithm.

## EXPERIMENT NO : 4.2

### TITLE: TO IMPLEMENT VARIOUS RECURSIVE SORTING ALGORITHMS

#### OBJECTIVE:

- To study the concept of Recursive Sorting and implement the Merge Sort algorithm.

#### THEORY:

Recursive sorting algorithms are methods that break down an initial array into smaller subarrays, sort these subarrays, and then recombine them to produce a fully sorted array. A notable example of such an algorithm is Merge Sort. Merge Sort operates through three primary steps:

1. Divide: The initial array is divided recursively into smaller halves until each subarray contains only one element.
2. Sort: The two halves are sorted individually through recursive calls, ensuring that each subarray is ordered correctly.
3. Merge: The sorted subarrays are then merged together by comparing elements and arranging them in the correct order. This merging process continues until the entire array is sorted.

Merge Sort's efficiency stems from its ability to efficiently divide, sort, and merge subarrays, resulting in a time complexity of  $O(n \log n)$ , making it particularly effective for sorting large datasets.

#### IMPLEMENTATION WITH C:

##### Program 1: Source code:

```
#include <stdio.h>
#include <stdlib.h>
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];
}
```

```

        j++;
    }
    k++;
}
while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}
while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
}
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
void printArray(int A[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
int main() {
    int n, i;
    printf("Enter the total number of elements:\n");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    printf("Given array is:\n");
    printArray(arr, arr_size);
    mergeSort(arr, 0, arr_size - 1);
    printf("\nSorted array is:\n");
    printArray(arr, arr_size);
    return 0;
}

```

**OUTPUT:**

## Output

```
/tmp/jgn3lqPl0Y.o
Enter the total number of elements:
5
Enter the elements:
10 20 19 21 9
Given array is:
10 20 19 21 9

Sorted array is:
9 10 19 20 21
```

## DISCUSSION:

Merge Sort, a Recursive Sorting algorithm, showcases the effectiveness of the divide-and-conquer approach. It achieves efficient sorting by recursively breaking down the input array into smaller subarrays, sorting them, and then merging them back together. The implementation includes robust error handling for edge cases such as invalid array sizes and empty arrays. With a time complexity of  $O(n \log n)$  in both average and worst cases, Merge Sort proves to be a superior choice for sorting larger datasets compared to methods like Insertion Sort.

## CONCLUSION:

Through this experiment, we have gained a comprehensive understanding of Recursive Sorting and the Merge Sort algorithm.

## EXPERIMENT NO : 4.3

### TITTLE: TO IMPLEMENT LINEAR AND BINARY SEARCHING ALGORITHMS

#### OBJECTIVE:

- To study the concept of Linear and Binary Searching Algorithms and implement them.

#### THEORY:

Linear Search is a straightforward searching algorithm that sequentially examines each element in the list until the target element is found or until the entire list has been checked.

Binary Search, on the other hand, is an efficient searching algorithm designed for sorted arrays. It repeatedly divides the search interval in half until the target element is found or the search space is exhausted.

#### Algorithm for Linear Search:

1. Start: Begin at the first element of the list.
2. Loop: Repeat the following steps until the end of the list is reached:
  - Compare the current element with the target value.
  - If the current element matches the target value, return its index.
  - If the current element does not match the target value, move to the next element.
3. End: If the end of the list is reached without finding a match, return a special value (e.g., -1) to indicate that the target value is not present in the list.

#### Algorithm for Binary Search:

1. Start: Begin with the entire sorted array.
2. Initialize: Set the low and high indices to the first and last elements of the array, respectively.
3. Loop: Repeat the following steps until the low index is less than or equal to the high index:
  - Compute the middle index as the average of the low and high indices.
  - Compare the target value with the middle element of the array.
  - If the target value matches the middle element, return its index.
  - If the target value is less than the middle element, set the high index to one less than the middle index.
  - If the target value is greater than the middle element, set the low index to one more than the middle index.
4. End: If the low index becomes greater than the high index, the target value is not present in the array.

The primary distinction between the two algorithms lies in their time complexity. Linear Search has a time complexity of  $O(n)$  in the average and worst cases, while Binary Search has a time complexity of  $O(\log n)$  in the average and worst cases, making it more efficient for larger datasets.

### IMPLEMENTATION WITH C:

#### Program 1: Source code:

To Implement the linear search algorithm.

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

```
int linearSearch(int array[], int size, int target) {  
    for (int i = 0; i < size; i++) {  
        if (array[i] == target) {  
            return i;  
        }  
    }  
}
```



```

    }}
    return -1;
}

int main() {
    int index, target, size = 5;
    int array[5] = {10, 20, 30, 40, 50};

    printf("Array elements: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\nEnter the value to search: ");
    scanf("%d", &target);

    index = linearSearch(array, size, target);
    if (index != -1) {
        printf("Target value %d found at index %d.\n", target, index+1);
    } else {
        printf("Target value %d not found in the array.\n", target);
    }
    return 0;
}

```

## OUTPUT:

```

Output

/tmp/iEqUUUo8E4.o
Array elements: 10 20 30 40 50
Enter the value to search: 20
Target value 20 found at index 2.

```

## Program 2: Source code:

To Implement the binary search algorithm.

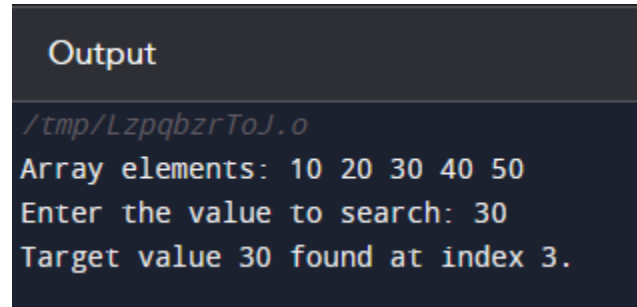
```

#include <stdio.h>
int binarySearch(int array[], int size, int target) {
    int low = 0;
    int high = size - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (array[mid] == target) {
            return mid;
        } else if (array[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1; }
int main() {
    int target, index, size = 5;
    int array[5] = {10, 20, 30, 40, 50};
    printf("Array elements: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
}

```

```
}  
printf("\nEnter the value to search: ");  
scanf("%d", &target);  
index = binarySearch(array, size, target)  
if (index != -1) {  
    printf("Target value %d found at index %d.\n", target, index+1); } else {  
    printf("Target value %d not found in the array.\n", target); }  
return 0;}
```

## OUTPUT

A terminal window with a dark background. The title bar reads "Output". The terminal shows the execution of a program. It starts with a prompt character, followed by the file path "/tmp/LzpqbzrToJ.o". The program prints "Array elements: 10 20 30 40 50". It then prompts "Enter the value to search: 30". Finally, it prints "Target value 30 found at index 3.".

```
Output  
/tmp/LzpqbzrToJ.o  
Array elements: 10 20 30 40 50  
Enter the value to search: 30  
Target value 30 found at index 3.
```

## DISCUSSION:

The Linear Search algorithm, although simple and straightforward to implement, exhibits a time complexity of  $O(n)$ , rendering it less efficient when dealing with larger datasets. In contrast, the Binary Search algorithm stands out as a more efficient searching technique with a time complexity of  $O(\log n)$ , particularly suited for sizable sorted datasets.

Both programs are equipped to handle edge cases, such as scenarios where the target element is not found within the array, ensuring comprehensive error handling. Employing distinct programs for Linear Search and Binary Search enables a lucid comparison, facilitating a deeper understanding of the disparities between the two algorithms.

## CONCLUSION:

Through this experiment, we have gained a comprehensive understanding of linear and binary search algorithm. The successful implementation of linear and binary search algorithm in the provided C program demonstrates our ability to apply the theoretical concepts to practical programming tasks