JAIN (DEEMED-TO-BE UNIVERSITY), KOCHI

COURSE CODE: 23MCAC101L

COURSE TITLE: DATA STRUCTURES LAB

NAME:

USN:

PROGRAMME:

SEMESTER: FIRST

DATE OF SUBMISSION: <to be entered by faculty during final submission>

CERTIFICATE

Certified that this is the bonafide record work done in the ……………….Laboratory by …………… ………………………., of First Semester …………Degree Course under School of ………………..in JAIN Deemed to be University Kochi, having University Register No JUK………………… during the academic year……………...

Faculty In-Charge **Head of the Department**

TABLE OF CONTENTS

Sl No	Name of Program	Date	Page No	Remarks

EXPERIMENT NO: 1 **DATE:**

SINGLY LINKED LIST

AIM:

Write a C program that uses functions to perform the following:

a. Create a singly linked list of integers

b. Delete a given integer from above linked list

c. Display the contents from the above linked list after deletion

ALGORITHM:

Step 1: START
Step 2: Initialize the head pointer of the linked list as `NULL`.
Step 3: Display a menu to the user:

- 1: Insert Integer
- 2: Delete Integer
- 3: Display List
- 4: Exit

Step 4: Based on the user's choice, perform the following:

Insertion Operation:

Step 4.1: If the choice is 1 (Insert Integer):
Step 4.1.1: Read the integer to insert.
Step 4.1.2: Create a new node with the given integer.
Step 4.1.3: If the list is empty, set the head pointer to the new node.
Step 4.1.4: Otherwise, traverse the list to the last node.
Step 4.1.5: Update the `next` pointer of the last node to point to the new node.
Step 4.1.6: Return to Step 3.

Deletion Operation:

Step 4.2: If the choice is 2 (Delete Integer):
Step 4.2.1: Read the integer to delete.
Step 4.2.2: Check if the head node contains the given integer:
- **Step 4.2.2.1**: If yes, update the head pointer to point to the next node and free the old head node.
Step 4.2.3: Otherwise, traverse the list to find the node with the given integer:
- **Step 4.2.3.1**: If found, unlink the node from the list and free it.

- **Step 4.2.3.2**: If not found, display a "Not Found" message.
Step 4.2.4: Return to Step 3.

Display Operation:

Step 4.3: If the choice is 3 (Display List):
Step 4.3.1: If the list is empty, display "List is empty".
Step 4.3.2: Otherwise, traverse the list from the head, printing each node's data until reaching the end (`NULL`).
Step 4.3.3: Return to Step 3.

Exit Operation:
Step 4.4: If the choice is 4 (Exit):
Step 4.4.1: TERMINATE the program.

Step 5: END

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;

}


void insertNode(struct Node** head, int data) {

    struct Node* newNode = createNode(data);
```

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

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

void displayList(struct Node* head) {
    if (!head) {
```

```
        printf("List is empty.\n");
        return;
    }
    printf("Linked List: ");
    while (head) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;
    int choice, data;

    while (1) {
        printf("\n1. Insert\n2. Delete\n3. Display\n4. Exit\nChoice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: printf("Enter value: "); scanf("%d", &data); insertNode(&head, data); break;
            case 2: printf("Enter value: "); scanf("%d", &data); deleteNode(&head, data); break;
            case 3: displayList(head); break;
            case 4: exit(0);
            default: printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

OUTPUT:

1. Insert

2. Delete

3. Display

4. Exit

Choice: 1

Enter value: 5

1. Insert

2. Delete

3. Display

4. Exit

Choice: 1

Enter value: 6

1. Insert

2. Delete

3. Display

4. Exit

Choice: 1

Enter value: 7

1. Insert

2. Delete

3. Display

4. Exit

Choice: 3

Linked List: 5 -> 6 -> 7 -> NULL

1. Insert

2. Delete

3. Display

4. Exit

Choice: 2

Enter value: 6

1. Insert

2. Delete

3. Display

4. Exit

Choice: 3

Linked List: 5 -> 7 -> NULL

1. Insert

2. Delete

3. Display

4. Exit

Choice: 4

RESULT:

The program is run and the result is verified.

EXPERIMENT NO: 2 **DATE:**

DOUBLY LINKED LIST

AIM:

Write a C program that uses functions to perform the following:

a. Create a doubly linked list of integers

b. Delete a given integer from above doubly linked list

c. Display the contents of above after deletion

ALGORITHM:

Step 1: Create a struct Node to represent a node in the doubly linked list, containing data, prev, and next fields.

Step 2: Initialize a head pointer to NULL to represent an empty list.

Step 3: Insert elements into the doubly linked list:

Step 4: Call insertEnd multiple times to insert desired values.

Print the original list:

Step 5: Call displayList to display the contents of the doubly linked list.

Delete a node:

Step 6: Call deleteNode with the desired value to remove the corresponding node.

Print the updated list:

Step 7: Call displayList again to display the modified doubly linked list.

Free memory:

Step 8: Iterate through the doubly linked list, freeing each node using a temporary pointer.

SOURCE CODE:

```
#include <stdio.h>

#include <stdlib.h>
```

```c
struct Node {
    int data;
    struct Node *prev, *next;
};

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

void insertEnd(struct Node **head, int data) {
    struct Node *newNode = createNode(data), *temp = *head;
    if (!*head) {
        *head = newNode;
        return;
    }
    while (temp->next) temp = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
}

void deleteNode(struct Node **head, int key) {
    struct Node *temp = *head;
```

```
    if (!temp) {
        printf("List is empty!\n");
        return;
    }
    while (temp && temp->data != key) temp = temp->next;
    if (!temp) {
        printf("Node with value %d not found!\n", key);
        return;
    }
    if (*head == temp) *head = temp->next;
    if (temp->prev) temp->prev->next = temp->next;
    if (temp->next) temp->next->prev = temp->prev;
    free(temp);
    printf("Node with value %d deleted!\n", key);
}

void displayList(struct Node *head) {
    if (!head) {
        printf("List is empty!\n");
        return;
    }
    printf("Doubly linked list: ");
    while (head) {
        printf("%d ", head->data);
        head = head->next;
    }
```

```
    printf("\n");
}


int main() {
    struct Node *head = NULL;
    int choice, value;


    while (1) {
        printf("\n1. Insert at End\n2. Delete by Value\n3. Display List\n4. Exit\nEnter your
choice: ");
        scanf("%d", &choice);


        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insertEnd(&head, value);
                break;
            case 2:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                deleteNode(&head, value);
                break;
            case 3:
                displayList(head);
                break;
```

```
            case 4:
                printf("Exiting program.\n");
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

OUTPUT:

1. Insert at End

2. Delete by Value

3. Display List

4. Exit

Enter your choice: 1

Enter value to insert: 4

1. Insert at End

2. Delete by Value

3. Display List

4. Exit

Enter your choice: 1

Enter value to insert: 9

1. Insert at End

2. Delete by Value

3. Display List

4. Exit

Enter your choice: 1

Enter value to insert: 7

1. Insert at End

2. Delete by Value

3. Display List

4. Exit

Enter your choice: 3

Doubly linked list: 4 9 7

1. Insert at End

2. Delete by Value

3. Display List

4. Exit

Enter your choice: 2

Enter value to delete: 4

Node with value 4 deleted!

1. Insert at End

2. Delete by Value

3. Display List

4. Exit

Enter your choice: 3

Doubly linked list: 9 7

1. Insert at End

2. Delete by Value

3. Display List

4. Exit

Enter your choice: 4

Exiting program.

RESULT:

The program is run and the result is verified.

EXPERIMENT NO: 3 **DATE:**

STACK USING ARRAY AND LINKED LIST

AIM:

Write C programs to implement a Stack ADT using
i) Array and
ii) Linked list respectively.

ALGORITHM:

Using Array:

Step 1: START

Step 2: Define a Stack structure with an array of items and an integer top initialized to -1.

Step 3: Create utility functions:

- Initialize Stack: Set top = -1.
- Check if Stack is Empty: Return true if top == -1.
- Check if Stack is Full: Return true if top == MAX - 1.
- Push Element:
  - If the stack is full, print "Stack is full".
  - Otherwise, increment top and insert the element at items[top].
- Pop Element:
  - If the stack is empty, print "Stack is empty" and return -1.
  - Otherwise, return the element at items[top] and decrement top.
- Show Stack:
  - If the stack is empty, print "Stack is empty".
  - Otherwise, print elements from top to 0.

Step 4: Create a menu-driven program in main to:

- Push an element onto the stack.
- Pop an element from the stack.
- Show all elements in the stack.
- Exit the program.

Step 5: Based on user input, call the appropriate function:

- If 1, call push and pass the user-provided value.
- If 2, call pop and display the returned value.
- If 3, call show to display stack contents.
- If 4, terminate the program.

Step 6: END

Using Linked List:

Step 1: START
Step 2: Define a Node structure with data (integer) and next (pointer to the next node).
Step 3: Implement utility functions:

- **Push**:
    1. Allocate memory for a new node.
    2. If memory allocation fails, print "Stack overflow" and return.
    3. Assign value to the new node's data field.
    4. Make the new node's next point to the current top of the stack.
    5. Update the top pointer to the new node.
- **Pop**:
    1. If the stack is empty, print "Stack underflow" and return -1.
    2. Save the top node's data to a variable.
    3. Update the top pointer to the next node in the stack.
    4. Free the memory of the old top node.
    5. Return the saved data.
- **Display**:
    1. If the stack is empty, print "Stack is empty" and return.
    2. Traverse the stack from the top node, printing each node's data.

Step 4: In the main function:

- Initialize stack (pointer to Node) as NULL.
- Display a menu with options:
    - Push an element.

    - Pop an element.
    - Display all elements.
    - Exit the program.
- Take the user's choice and call the respective function:
    - For 1, take input and call push.
    - For 2, call pop and display the returned value.
    - For 3, call display.
    - For 4, terminate the program.

Step 5: Repeat until the user chooses to exit.

Step 6: END

SOURCE CODE:

Using array:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 100


struct Stack {

    int items[MAX];

    int top;

};


void initialize(struct Stack *s) {

    s->top = -1;

}


int isEmpty(struct Stack *s) {
```

```c
    return s->top == -1;
}

int isFull(struct Stack *s) {
    return s->top == MAX - 1;
}

void push(struct Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack is full. Cannot push %d.\n", value);
    } else {
        s->items[++s->top] = value;
        printf("Pushed %d onto the stack.\n", value);
    }
}

int pop(struct Stack *s) {
    return isEmpty(s) ? (printf("Stack is empty. Cannot pop.\n"), -1) : s->items[s->top--];
}

void show(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
    } else {
        printf("Stack elements:\n");
        for (int i = s->top; i >= 0; i--)
```

```
            printf("%d\n", s->items[i]);
    }
}

int main() {
    struct Stack s;
    int choice, data;
    initialize(&s);

    while (1) {
        printf("\n1. Push\n2. Pop\n3. Show Stack\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &data);
                push(&s, data);
                break;
            case 2:
                printf("Popped element: %d\n", pop(&s));
                break;
            case 3:
                show(&s);
                break;
            case 4:
```

```
            printf("Exiting program.\n");
            exit(0);
          default:
            printf("Invalid choice! Try again.\n");
        }
    }

    return 0;
}
```

Using linked list:

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

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

void push(struct Node **top, int value) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Stack overflow!\n");
        return;
    }
    newNode->data = value;
```

```
    newNode->next = *top;
    *top = newNode;
    printf("Pushed %d onto the stack.\n", value);
}

int pop(struct Node **top) {
    if (!*top) {
        printf("Stack underflow!\n");
        return -1;
    }
    struct Node *temp = *top;
    int value = temp->data;
    *top = (*top)->next;
    free(temp);
    return value;
}

void display(struct Node *top) {
    if (!top) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack elements:\n");
    while (top) {
        printf("%d\n", top->data);
        top = top->next;
```

```
    }
}

int main() {
    struct Node *stack = NULL;
    int choice, data;

    while (1) {
        printf("\n1. Push\n2. Pop\n3. Display Stack\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter element to push: ");
                scanf("%d", &data);
                push(&stack, data);
                break;
            case 2:
                printf("Popped element: %d\n", pop(&stack));
                break;
            case 3:
                display(stack);
                break;
            case 4:
                printf("Exiting program.\n");
                exit(0);
```

```
        default:
          printf("Invalid choice! Try again.\n");
      }
    }

    return 0;
}
```

OUTPUT:

```
1. Push
2. Pop
3. Display Stack
4. Exit
Enter your choice: 1
Enter element to push: 6
Pushed 6 onto the stack.

1. Push
2. Pop
3. Display Stack
4. Exit
Enter your choice: 1
Enter element to push: 5
Pushed 5 onto the stack.

1. Push
```

2. Pop

3. Display Stack

4. Exit

Enter your choice: 1

Enter element to push: 7

Pushed 7 onto the stack.

1. Push

2. Pop

3. Display Stack

4. Exit

Enter your choice: 3

Stack elements:

7

5

6

1. Push

2. Pop

3. Display Stack

4. Exit

Enter your choice: 2

Popped element: 7

1. Push

2. Pop

3. Display Stack

4. Exit

Enter your choice: 3

Stack elements:

5

6

1. Push

2. Pop

3. Display Stack

4. Exit

Enter your choice: 4

Exiting program.

RESULT:

The program is run and the result is verified.

EXPERIMENT NO: 4 **DATE:**

INFIX TO POSTFIX

AIM:

Write a C program that uses stack operations to convert a given infix expression into its postfix equivalent.

ALGORITHM:

Step 1: START

Step 2: Define a stack array of size 100 and a top variable initialized to -1.

Step 3: Implement utility functions:

- **Push**: Increment top and insert the element at stack[top].
- **Pop**: If the stack is empty, return -1; otherwise, return stack[top] and decrement top.
- **Priority**:
  - Return 0 for '('.
  - Return 1 for '+' or '-'.
  - Return 2 for '*' or '/'.
  - Default priority is 0.

Step 4: Read the infix expression into a character array exp.

Step 5: For each character in the expression:

- If the character is an operand (isalnum), print it.
- If the character is '(', push it onto the stack.
- If the character is ')', pop and print characters from the stack until '(' is encountered.
- If the character is an operator:
  - Pop and print operators from the stack while the stack is not empty, and the priority of the top operator is greater than or equal to the current operator's priority.
  - Push the current operator onto the stack.

Step 6: After processing the expression, pop and print all remaining operators from the stack.

Step 7: END

SOURCE CODE:

```
#include <stdio.h>

#include <ctype.h>


char stack[100];

int top = -1;


void push(char x) {

    stack[++top] = x;

}


char pop() {

    return (top == -1) ? -1 : stack[top--];

}


int priority(char x) {

    return (x == '(') ? 0 : (x == '+' || x == '-') ? 1 : (x == '*' || x == '/') ? 2 : 0;

}


int main() {

    char exp[100], *e, x;

    printf("Enter the expression: ");

    scanf("%s", exp);


    for (e = exp; *e != '\0'; e++) {

        if (isalnum(*e)) {
```

```
        printf("%c", *e);
      } else if (*e == '(') {
        push(*e);
      } else if (*e == ')') {
        while ((x = pop()) != '(') {
          printf("%c", x);
        }
      } else {
        while (top != -1 && priority(stack[top]) >= priority(*e)) {
          printf("%c", pop());
        }
        push(*e);
      }
    }

    while (top != -1) {
      printf("%c", pop());
    }

    return 0;
}
```

OUTPUT:

Enter the expression (without spaces): A+B-C/D*E

AB+CD/E* -

Press any key to exit...

RESULT:

The program is run and the result is verified.

EXPERIMENT NO: 5 **DATE:**

QUEUE ADT USING STACK AND LINKED LIST

AIM:

Write C programs to implement a queue ADT using

i) Array and

ii) Linked list respectively.

ALGORITHM :

using Array

Step 1: START
Step 2: Define an array queue of size MAX and two integer variables front and rear initialized to -1.
Step 3: Implement utility functions:

- **Enqueue:**
  - Check if the queue is full, If yes, print "Queue Overflow" and return.
  - If the queue is empty, set front = 0.
  - Increment rear by 1 and insert the element at queue[rear].
- **Dequeue:**
  - Check if the queue is empty, print "Queue Underflow" and return -1.
  - Retrieve the element at queue[front] and increment front by 1.
  - If front > rear after dequeuing, reset both front and rear to -1.
- **Display:**
  - If the queue is empty, print "Queue is Empty".
  - Otherwise, iterate from front to rear and print all elements.

Step 4: Allow user input to perform Enqueue, Dequeue, or Display operations.
Step 5: Repeat until the user decides to exit.
Step 6: END

using Linked List

Step 1: START
Step 2: Define a Node structure with data (integer) and next (pointer to the next node).
Define two pointers, front and rear, both initialized to NULL.

Step 3: Implement utility functions:

- **Enqueue:**
  - Create a new node and assign the data to it.
  - Set the next pointer of the new node to NULL.
  - If the queue is empty (front == NULL and rear == NULL):
    - Set both front and rear to point to the new node.
  - Otherwise, set rear->next to the new node and update rear to point to the new node.
- **Dequeue:**
  - Check if the queue is empty, print "Queue Underflow" and return -1.
  - Retrieve the data from the front node.
  - Move front to the next node.
  - If front becomes NULL, set rear to NULL (queue becomes empty).
  - Free the memory of the dequeued node.
- **Display:**
  - If the queue is empty (front == NULL), print "Queue is Empty".
  - Otherwise, traverse from front to rear and print the data of each node.

Step 4: Allow user input to perform Enqueue, Dequeue, or Display operations.
Step 5: Repeat until the user decides to exit.
Step 6: END

SOURCE CODE:

Using array:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 5


struct Queue {

        int items[MAX];

        int front, rear;

};


struct Queue* createQueue() {
```

```
    struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
    q->front = 0;
    q->rear = -1;
    return q;
}

void enqueue(struct Queue* q, int value) {
    if (q->rear == MAX - 1) {
  printf("Queue Overflow.\n");
    return;
    }
 q->items[++q->rear] = value;
 printf("Enqueued: %d\n", value);
}

void dequeue(struct Queue* q) {
    if (q->front > q->rear) {
  printf("Queue Underflow.\n");
    return;
    }
 printf("Dequeued: %d\n", q->items[q->front++]);
}

void displayQueue(struct Queue* q) {
    if (q->front > q->rear) {
  printf("Queue is empty.\n");
```

```
        return;
        }
    printf("Queue: ");
        for (int i = q->front; i <= q->rear; i++) {
      printf("%d ", q->items[i]);
        }
    printf("\n");
}

int main() {
        struct Queue* q = createQueue();
        int choice, value;

        do {
      printf("\n1. Enqueue\n2. Dequeue\n3. Display Queue\n4. Exit\nEnter your choice: ");
      scanf("%d", &choice);
        switch (choice) {
        case 1:
          printf("Enter value to enqueue: ");
          scanf("%d", &value);
          enqueue(q, value);
          break;
        case 2:
          dequeue(q);
          break;
        case 3:
```

```
            displayQueue(q);
            break;
        case 4:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice.\n");
        }
    } while (choice != 4);
    return 0;
}
```

Using linked list:

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

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

struct Queue {
    struct Node* front;
    struct Node* rear;
};
```

```
struct Queue* createQueue() {
    struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
    q->front = q->rear = NULL;
    return q;
}

void enqueue(struct Queue* q, int value) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = value;
    temp->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = temp;
        return;
    }
    q->rear->next = temp;
    q->rear = temp;
    printf("Enqueued: %d\n", value);
}

void dequeue(struct Queue* q) {
    if (q->front == NULL) {
        printf("Queue is empty.\n");
        return;
    }
    struct Node* temp = q->front;
    q->front = q->front->next;
```

```
    if (q->front == NULL) {
        q->rear = NULL;
    }
    printf("Dequeued: %d\n", temp->data);
    free(temp);
}

void displayQueue(struct Queue* q) {
    if (q->front == NULL) {
        printf("Queue is empty.\n");
        return;
    }
    struct Node* temp = q->front;
    printf("Queue: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    struct Queue* q = createQueue();
    int choice, value;

    do {
```

```
        printf("\n1. Enqueue\n2. Dequeue\n3. Display Queue\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
        case 1:
            printf("Enter value to enqueue: ");
            scanf("%d", &value);
            enqueue(q, value);
            break;
        case 2:
            dequeue(q);
            break;
        case 3:
            displayQueue(q);
            break;
        case 4:
          printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice.\n");
        }
    } while (choice != 4);

    return 0;
}
```

OUTPUT:

1. Enqueue

2. Dequeue

3. Display Queue

4. Exit

Enter your choice: 1

Enter value to enqueue: 10

Enqueued: 10

1. Enqueue

2. Dequeue

3. Display Queue

4. Exit

Enter your choice: 1

Enter value to enqueue: 20

Enqueued: 20

1. Enqueue

2. Dequeue

3. Display Queue

4. Exit

Enter your choice: 1

Enter value to enqueue: 30

Enqueued: 30

1. Enqueue

2. Dequeue

3. Display Queue

4. Exit

Enter your choice: 1

Enter value to enqueue: 40

Enqueued: 40

1. Enqueue

2. Dequeue

3. Display Queue

4. Exit

Enter your choice: 3

Queue: 10 20 30 40

1. Enqueue

2. Dequeue

3. Display Queue

4. Exit

Enter your choice: 2

Dequeued: 10

1. Enqueue

2. Dequeue

3. Display Queue

4. Exit

Enter your choice: 3

Queue: 20 30 40

1. Enqueue

2. Dequeue

3. Display Queue

4. Exit

Enter your choice: 4

Exiting...

RESULT:

The program is run and the result is verified.

EXPERIMENT NO: 6 **DATE:**

CIRCULAR QUEUE

AIM:

Write a C program to implement a circular queue along with different operations using linked lists.

ALGORITHM :

SOURCE CODE:

OUTPUT:

RESULT:

EXPERIMENT NO: 7 **DATE:**

BINARY TREE

AIM:

Write a C program that uses functions to perform the following:

1) Create a Binary tree of numbers
2) Define functions to perform Inorder, Postorder and Preorder traversals on the above tree.

ALGORITHM :

SOURCE CODE:

OUTPUT:

RESULT:

EXPERIMENT NO: 8 **DATE:……**

BINARY SEARCH TREE

AIM:

Write a C program that uses functions to perform the following:

1. Create a Binary Search Tree (BST) of integers.
2. Define a function to search for a given key in the BST.

ALGORITHM:

SOURCE CODE:

OUTPUT:

RESULT:

EXPERIMENT NO: 9 **DATE:**

AVL TREE

AIM:

Write a C program to demonstrate an AVL Tree (Insertion and Rotations).

ALGORITHM:

Step 1: Create a struct Node to represent a node in the AVL tree, containing fields for the key, left and right child nodes, and the height.

Step 2: Create a helper function getHeight to return the height of a given node.

Step 3: Create a helper function max to return the maximum of two integers.

Step 4: Create a helper function getBalanceFactor to compute and return the balance factor of a node. This is calculated as the height difference between the left and right subtrees.

Step 5: Create a helper function createNode to create a new node with a given key. This function initializes the key, sets both left and right to NULL, and sets the height of the new node to 1.

Step 6: Create a rightRotate function to perform a right rotation on a given node. Update the heights of the involved nodes and return the new root node after rotation.

Step 7: Create a leftRotate function to perform a left rotation on a given node. Update the heights of the involved nodes and return the new root node after rotation.

Step 8: Create an insert function to insert a key into the AVL tree:

- If the tree is empty, create a new node.
- If the key is less than the current node's key, insert it into the left subtree.
- If the key is greater than the current node's key, insert it into the right subtree.
- If the key already exists, do nothing.
- After insertion, update the height of the current node.
- Calculate the balance factor and check for imbalances:

    - If the balance factor is greater than 1 (left-heavy), check the direction of imbalance and perform the appropriate rotation (right rotate, left-right rotate).
    - If the balance factor is less than -1 (right-heavy), check the direction of imbalance and perform the appropriate rotation (left rotate, right-left rotate).

Step 9: Create a preOrder function to perform a preorder traversal of the AVL tree and print the keys.

Step 10: In the main function, initialize the root of the AVL tree to NULL.

- Enter a loop to repeatedly ask for a key to insert into the AVL tree.
- Insert the key using the insert function.
- Display the preorder traversal of the AVL tree after each insertion.
- Check the balance factor of the root node and prompt the user to choose a rotation type if the tree is unbalanced:
    - If the tree is unbalanced, offer options for performing left, right, left-right, or right-left rotations.
- After performing the rotation, print the updated tree and continue the loop.

Step 11: Exit the loop when the user chooses to stop by inputting -1.

SOURCE CODE:

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

    int key;

    struct Node* left;

    struct Node* right;

    int height;

};
```

```c
int getHeight(struct Node* n) {
    if (n == NULL)
        return 0;
    return n->height;
}

struct Node* createNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

int getBalanceFactor(struct Node* n) {
    if (n == NULL)
        return 0;
    return getHeight(n->left) - getHeight(n->right);
}
```

```c
struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

    return x;
}

struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

    return y;
}
```

```
struct Node* insert(struct Node* node, int key) {
    if (node == NULL)
        return createNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;

    node->height = 1 + max(getHeight(node->left), getHeight(node->right));

    int balance = getBalanceFactor(node);

    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
```

```
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Main function
int main() {
    struct Node* root = NULL;
    int choice, key;

    while (1) {
        printf("\nEnter a key to insert (or -1 to exit): ");
        scanf("%d", &key);
        if (key == - 1) {
            break;
```

```
}
root = insert(root, key);
printf("Preorder traversal of the AVL tree is: ");
preOrder(root);
printf("\n");

int balance = getBalanceFactor(root);
if (balance > 1 || balance < -1) {
  printf("The tree is unbalanced. Balance factor: %d\n", balance);
  printf("Choose rotation type (1: Left, 2: Right, 3: Left-Right, 4: Right-Left): ");
  scanf("%d", &choice);
  switch (choice) {
    case 1:
      root = leftRotate(root);
      printf("Performed Left Rotation.\n");
      break;
    case 2:
      root = rightRotate(root);
      printf("Performed Right Rotation.\n");
      break;
    case 3:
      root = leftRotate(root->left);
      root = rightRotate(root);
      printf("Performed Left-Right Rotation.\n");
      break;
    case 4:
```

```
                root = rightRotate(root->right);
                root = leftRotate(root);
                printf("Performed Right-Left Rotation.\n");
                break;
            default:
                printf("Invalid choice. No rotation performed.\n");
        }
    } else {
        printf("The tree is balanced. Balance factor: %d\n", balance);
    }
  }

  return 0;
}
```

OUTPUT:

Enter a key to insert (or -1 to exit): 1

Preorder traversal of the AVL tree is: 1

The tree is balanced. Balance factor: 0

Enter a key to insert (or -1 to exit): 2

Preorder traversal of the AVL tree is: 1 2

The tree is balanced. Balance factor: -1

Enter a key to insert (or -1 to exit): 3

Preorder traversal of the AVL tree is: 2 1 3

The tree is balanced. Balance factor: 0

Enter a key to insert (or -1 to exit): 7

Preorder traversal of the AVL tree is: 2 1 3 7

The tree is balanced. Balance factor: -1

Enter a key to insert (or -1 to exit):

6

Preorder traversal of the AVL tree is: 2 1 6 3 7

The tree is balanced. Balance factor: -1

Enter a key to insert (or -1 to exit): -1

RESULT:

The program is run and the result is verified.

EXPERIMENT NO: 10 **DATE:**

BFS TRAVERSAL

AIM:

Write C program for implementing the Breadth first graph traversal technique.

ALGORITHM:

Step 1: Define data structures.

- Create a 2D array graph[MAX][MAX] to represent the adjacency matrix of the graph.
- Create an array visited[MAX] to keep track of visited vertices, initializing all elements to 0.

Step 2: Define a Queue structure.

- Create a struct Queue with fields:
  - items[MAX]: Array to hold queue elements.
  - front and rear: Pointers to manage the queue.
- Initialize front = -1 and rear = -1.

Step 3: Implement enqueue and dequeue functions.

- **Enqueue**: Add an element to the queue and update the rear pointer.
- **Dequeue**: Remove an element from the queue, return it, and update the front pointer.

Step 4: Input the graph.

- Ask the user for the number of vertices (numVertices) and edges (numEdges).
- For each edge, set graph[u][v] = graph[v][u] = 1 to create an undirected graph.

Step 5: Initialize BFS traversal.

- Start from a given vertex startVertex:
  - Mark it as visited: visited[startVertex] = 1.
  - Enqueue the vertex.

Step 6: Perform BFS traversal.

- While the queue is not empty:
  - Dequeue the front element (currentVertex) and print it.
  - For each unvisited neighbor of currentVertex:
    - Mark it as visited.
    - Enqueue the neighbor.

Step 7: End traversal.

- Print the traversal order of vertices.

Step 8: Exit.

- Terminate the program after completing the BFS traversal.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 20

int graph[MAX][MAX] = {0}, visited[MAX] = {0}; // Adjacency matrix and visited array

struct Queue {
   int items[MAX], front, rear;
} q = {.front = -1, .rear = -1}; // Initialize the queue structure

void enqueue(int value) {
   if (q.rear == MAX - 1) return; // Queue overflow
   if (q.front == -1) q.front = 0;
   q.items[++q.rear] = value;
}

int dequeue() {
   if (q.front == -1) return -1; // Queue underflow
   int item = q.items[q.front++];
   if (q.front > q.rear) q.front = q.rear = -1;
   return item;
}

void BFS(int startVertex, int numVertices) {
   printf("BFS Traversal starting from vertex %d: ", startVertex);
```

```
    enqueue(startVertex);
    visited[startVertex] = 1;

    while (q.front != -1) {
        int curr = dequeue();
        printf("%d ", curr);

        for (int i = 0; i < numVertices; i++) {
            if (graph[curr][i] && !visited[i]) {
                visited[i] = 1;
                enqueue(i);
            }
        }
    }
    printf("\n");
}

int main() {
    int numVertices, numEdges, u, v;

    printf("Enter number of vertices: ");
    scanf("%d", &numVertices);

    printf("Enter number of edges: ");
    scanf("%d", &numEdges);

    printf("Enter edges (u v):\n");
    for (int i = 0; i < numEdges; i++) {
        scanf("%d %d", &u, &v);
        graph[u][v] = graph[v][u] = 1;
```

```
    }

    BFS(0, numVertices);
    return 0;
}
```

OUTPUT:

Enter number of vertices: 5

Enter number of edges: 4

Enter edges (u v):

0 1

0 2

1 3

0 4

BFS Traversal starting from vertex 0: 0 1 2 4 3

RESULT:

The program is run and the result is verified.

EXPERIMENT NO: 11 **DATE:**

DFS TRAVERSAL

AIM:

Write C program for implementing the Depth first graph traversal technique.

ALGORITHM :

Step 1: Define data structures.

- Create a 2D array graph[MAX][MAX] to represent the adjacency matrix of the graph.
- Create an array visited[MAX] to keep track of visited vertices, initializing all elements to 0.

Step 2: Define a Queue structure.

- Create a struct Queue with fields:
  - items[MAX]: Array to hold queue elements.
  - front and rear: Pointers to manage the queue.
- Initialize front = -1 and rear = -1.

Step 3: Implement enqueue and dequeue functions.

- Enqueue: Add an element to the queue and update the rear pointer.
- Dequeue: Remove an element from the queue, return it, and update the front pointer.

Step 4: Input the graph.

- Ask the user for the number of vertices (numVertices) and edges (numEdges).
- For each edge, set graph[u][v] = graph[v][u] = 1 to create an undirected graph.

Step 5: Initialize BFS traversal.

- Start from a given vertex startVertex:
  - Mark it as visited: visited[startVertex] = 1.
  - Enqueue the vertex.

Step 6: Perform BFS traversal.

- While the queue is not empty:
  - Dequeue the front element (currentVertex) and print it.
  - For each unvisited neighbor of currentVertex:
    - Mark it as visited.
    - Enqueue the neighbor.

Step 7: End traversal.

- Print the traversal order of vertices.

Step 8: Exit.

SOURCE CODE:

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

#define MAX 20

int graph[MAX][MAX];
int visited[MAX];

struct Stack {
    int items[MAX];
    int top;
};

void initStack(struct Stack *s) {
    s->top = -1;
}

int isEmpty(struct Stack *s) {
    return s->top == -1;
}

void push(struct Stack *s, int value) {
    if (s->top == MAX - 1) {
        printf("Stack Overflow\n");
```

```
    } else {
        s->items[++(s->top)] = value;
    }
}

int pop(struct Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow\n");
        return -1;
    } else {
        return s->items[(s->top)--];
    }
}

void DFS(int startVertex, int numVertices) {
    int i;
    struct Stack stack;
    initStack(&stack);

    push(&stack, startVertex);

    visited[startVertex] = 1;

    printf("DFS Traversal starting from vertex %d: ", startVertex);

    while (!isEmpty(&stack)) {
        int currentVertex = pop(&stack);
        printf("%d ", currentVertex);

for (i = 0; i < numVertices; i++) {
```

```
            if (graph[currentVertex][i] == 1 && !visited[i]) {
                push(&stack, i);
                visited[i] = 1;
            }
        }
    }
    printf("\n");
}

int main() {
    int numVertices, numEdges, u, v, i, j;
    printf("Enter the number of vertices: ");
    scanf("%d", &numVertices);

    for (i = 0; i < numVertices; i++) {
for (j = 0; j < numVertices; j++) {
            graph[i][j] = 0;
        }
    }
    printf("Enter the number of edges: ");
    scanf("%d", &numEdges);

    printf("Enter the edges (u v):\n");
    for (i = 0; i < numEdges; i++) {
        scanf("%d %d", &u, &v);
        graph[u][v] = 1;
        graph[v][u] = 1;
    }

    for (int i = 0; i < numVertices; i++) {
```

```
        visited[i] = 0;
    }

    DFS(0, numVertices);
    printf("\nPress any key to exit...");
    return 0;
}
```

OUTPUT:

```
Enter the number of vertices: 4 3
Enter the number of edges: Enter the edges (u v):
0 1
0 2
1 3
DFS Traversal starting from vertex 0: 0 2 1 3

Press any key to exit...
```

RESULT:

The program is run and the result is verified.

EXPERIMENT NO: 12 **DATE:**

HASHING

AIM:

Write C program for demonstrating Hashing technique with Searching operation.

ALGORITHM:

Step 1: Define data structures.

- Create a HashTable struct with an array data[TABLE_SIZE] to store the table elements.

Step 2: Initialize the hash table.

- Initialize each element of the data array to -1 to indicate empty slots.

Step 3: Define the hash function.

- The hashFunction computes the index by applying modulo operation on the key: key % TABLE_SIZE.

Step 4: Define the insert function.

- Compute the index using the hash function.
- Use linear probing to find the next available slot if the computed index is occupied.
- Insert the key at the available slot.

Step 5: Define the search function.

- Compute the index using the hash function.
- Search for the key in the table using linear probing.
- Return the index of the key if found, otherwise return -1.

Step 6: Provide user interaction.

- Present options to the user for inserting or searching for a key in the hash table.
- Based on the user input, either insert a key, search for a key, or exit the program.

SOURCE CODE:

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

#define TABLE_SIZE 10

typedef struct {
    int data[TABLE_SIZE];
} HashTable;

void initializeHashTable(HashTable *ht) {
    for (int i = 0; i < TABLE_SIZE; i++) ht->data[i] = -1;
}

int hashFunction(int key) {
    return key % TABLE_SIZE;
}

void insert(HashTable *ht, int key) {
    int index = hashFunction(key);
    while (ht->data[index] != -1) index = (index + 1) % TABLE_SIZE;
    ht->data[index] = key;
    printf("Inserted %d at index %d\n", key, index);
}

int search(HashTable *ht, int key) {
    int index = hashFunction(key), startIndex = index;
    while (ht->data[index] != -1) {
        if (ht->data[index] == key) return index;
        index = (index + 1) % TABLE_SIZE;
```

```
        if (index == startIndex) break;
    }
    return -1;
}

int main() {
    int choice, key, index;
    HashTable ht;
    initializeHashTable(&ht);
    while (1) {
        printf("\n1. Insert key\n2. Search key\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter key to insert: "); scanf("%d", &key);
                insert(&ht, key); break;
            case 2:
                printf("Enter key to search: "); scanf("%d", &key);
                index = search(&ht, key);
                printf(index != -1 ? "Key %d found at index %d\n" : "Key %d not found\n", key,
index);
                break;
            case 3:
                printf("Exiting...\n"); return 0;
            default: printf("Invalid choice\n");
        }
    }
}
```

OUTPUT:

1. Insert key

2. Search key

3. Exit

Enter choice: 1

Enter key to insert: 6

Inserted 6 at index 6

1. Insert key

2. Search key

3. Exit

Enter choice: 1

Enter key to insert: 8

Inserted 8 at index 8

1. Insert key

2. Search key

3. Exit

Enter choice: 1

Enter key to insert: 18

Inserted 18 at index 9

1. Insert key

2. Search key

3. Exit

Enter choice: 2

Enter key to search: 8

Key 8 found at index 8

1. Insert key

2. Search key

3. Exit

Enter choice: 2

Enter key to search: 3

Key 3 not found

1. Insert key

2. Search key

3. Exit

Enter choice: 3

Exiting...

RESULT:

The program is run and the result is verified.