Course Code	**:**	**BCSL-033**
Course Title	**:**	**Data and File Structures Lab**
Assignment Number	**:**	**BCA(III)/L-033/Assignment/2024-25**
Maximum Marks	**:**	**100**
Weightage	**:**	**25%**
Last Dates for Submission	**:**	**31st October,2024(for July Session)** **30th April, 2025(for January Session)**

There are 8 questions of 10 marks each in this assignment carrying a total of 80 marks. Rest 20 marks are for viva voce. Please go through the guidelines regarding assignments given in the Programme Guide for the format of the presentation. Write all the programs in 'C' language.

Q1. Write a program to take input of a Matrix using array and display its transpose. **(10 Marks)**

Q2. Write a program in `C' Language to accept 10 strings as input and print them inlexicographic order **(10 Marks)**

Q3. Write a program to implement singly linked list for user inputs and perform the following operations on it:

(i) Reverse the linked list and display it.

(ii) Sort the nodes in ascending order and display them. **(10 Marks)**

Q4. Write a program using linked list that accepts two polynomials as input and displays the resultant polynomial after performing the multiplication of the input polynomials. **(10 Marks)**

Q5. Write a program to implement doubly linked list of integers as user inputs and perform the following operations:

(i) Calculate and display the sum of all the even integers of the doubly linked list

(ii) Insert new elements at the beginning, in the middle and at the end of the linked list **(10 Marks)**

Q6. Write a program in C to sort user input data using bubble sort method. Also, print the number of swaps and comparison operations performed for sorting the given data set. **(10 Marks)**

Q7. Write a program to convert an infix expression to a prefix expression. Use appropriate data structure. **(10 Marks)**

Q8. Write a program in 'C' language for the creation of a Red Black tree. Also,implement insertion and deletion operations. **(10 Marks)**

Q1. Write a program to take input of a Matrix using array and display its transpose.

Ans

Here's a C program that takes a matrix as input and displays its transpose:

c code

```
#include <stdio.h>

#define MAX_SIZE 100 // Maximum size of the matrix

// Function to display the matrix
void displayMatrix(int matrix[MAX_SIZE][MAX_SIZE], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

// Function to find the transpose of a matrix
void transposeMatrix(int matrix[MAX_SIZE][MAX_SIZE], int
transposed[MAX_SIZE][MAX_SIZE], int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            transposed[j][i] = matrix[i][j]; // Swap indices
        }
    }
}

int main() {
```

```
    int matrix[MAX_SIZE][MAX_SIZE], transposed[MAX_SIZE][MAX_SIZE];
    int rows, cols;

    // Take the number of rows and columns as input
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    printf("Enter the number of columns: ");
    scanf("%d", &cols);

    // Initialize the matrix with user input
    printf("Enter the elements row-wise:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }

    printf("\nOriginal Matrix:\n");
    displayMatrix(matrix, rows, cols);

    // Get the transpose of the matrix
    transposeMatrix(matrix, transposed, rows, cols);

    printf("\nTransposed Matrix:\n");
    displayMatrix(transposed, cols, rows);

    return 0;
}
```

How the Program Works:

1. **Input Matrix:** The program prompts the user for the number of rows and columns, followed by the elements of the matrix.
2. **Transpose Calculation:** It computes the transpose by swapping rows and columns.
3. **Display:** Both the original matrix and its transpose are printed.

Example Usage:

```
PS D:\.vscode> cd "d:\.vscode\c language\" ; if ($?) { gcc c1.c -o c1 } ; if ($?) { .\c1 }
Enter the number of rows: 2
Enter the number of columns: 3
Enter the elements row-wise:
1 2 3
4 5 6

Original Matrix:
1 2 3
4 5 6

Transposed Matrix:
1 4
2 5
3 6
PS D:\.vscode\c language>
```

Q2. Write a program in `C' Language to accept 10 strings as input and print them inlexicographic order (10 Marks)

Ans

Here's a C program that accepts 10 strings as input and prints them in lexicographic order:

c code

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

#define MAX_STRINGS 10
#define MAX_LENGTH 100

// Function to sort strings using bubble sort
void sortStrings(char strings[MAX_STRINGS][MAX_LENGTH], int count) {
    char temp[MAX_LENGTH];
    for (int i = 0; i < count - 1; i++) {
        for (int j = i + 1; j < count; j++) {
            // Compare strings
```

```
            if (strcmp(strings[i], strings[j]) > 0) {
                // Swap if they are in the wrong order
                strcpy(temp, strings[i]);
                strcpy(strings[i], strings[j]);
                strcpy(strings[j], temp);
            }
        }
    }
}

int main() {
    char strings[MAX_STRINGS][MAX_LENGTH];

    // Accept 10 strings as input
    printf("Enter 10 strings:\n");
    for (int i = 0; i < MAX_STRINGS; i++) {
        printf("String %d: ", i + 1);
        fgets(strings[i], MAX_LENGTH, stdin);

        // Remove newline character from input
        strings[i][strcspn(strings[i], "\n")] = '\0';
    }

    // Sort the strings
    sortStrings(strings, MAX_STRINGS);

    // Print the strings in lexicographic order
    printf("\nStrings in lexicographic order:\n");
    for (int i = 0; i < MAX_STRINGS; i++) {
```

```
        printf("%s\n", strings[i]);
    }

    return 0;
}
```

How the Program Works:

1. **Input Strings:** The program prompts the user to enter 10 strings. It uses fgets to read each string and removes the newline character.
2. **Sorting:** The sortStrings function implements a bubble sort algorithm to sort the strings in lexicographic order using the strcmp function for comparison.
3. **Display:** The sorted strings are printed to the console.

Example Usage:

```
PS D:\.vscode> cd "d:\.vscode\c language\" ; if ($?) { gcc c1.c -o c1 } ; if ($?) { .\c1 }
Enter 10 strings:
String 1: banana
String 2: apple
String 3: cherry
String 4: date
String 5: elderberry
String 6: fig
String 7: grape
String 8: kiwi
String 9: lemon
String 10: mango

Strings in lexicographic order:
apple
banana
cherry
date
elderberry
fig
grape
kiwi
lemon
mango
PS D:\.vscode\c language>
```

Q3. Write a program to implement singly linked list for user inputs and perform the following operations on it: (i) Reverse the linked list and display it. (ii) Sort the nodes in ascending order and display them

Ans

Here's a C program that implements a singly linked list, allows the user to input values, and performs the following operations: reversing the linked list and sorting the nodes in ascending order.

c code

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

// Define the structure for a linked list node
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the end of the linked list
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 != NULL) {
            temp = temp->next;
```

```
        }
        temp->next = newNode;
    }
}

// Function to reverse the linked list
void reverseList(struct Node** head) {
    struct Node* prev = NULL;
    struct Node* current = *head;
    struct Node* next = NULL;

    while (current != NULL) {
        next = current->next; // Store the next node
        current->next = prev; // Reverse the link
        prev = current;       // Move prev one step ahead
        current = next;       // Move current one step ahead
    }
    *head = prev; // Update head to the new front
}

// Function to sort the linked list using bubble sort
void sortList(struct Node** head) {
    if (*head == NULL) return;

    struct Node* current;
    struct Node* next;
    int temp;

    for (current = *head; current != NULL; current = current->next) {
```

```
        for (next = current->next; next != NULL; next = next->next) {
            if (current->data > next->data) {
                // Swap data
                temp = current->data;
                current->data = next->data;
                next->data = temp;
            }
        }
    }
}

// Function to display the linked list
void displayList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function
int main() {
    struct Node* head = NULL;
    int n, data;

    // Accept user input for the linked list
    printf("Enter the number of nodes: ");
    scanf("%d", &n);
```

```
    for (int i = 0; i < n; i++) {
        printf("Enter data for node %d: ", i + 1);
        scanf("%d", &data);
        insertNode(&head, data);
    }

    printf("\nOriginal Linked List:\n");
    displayList(head);

    // Reverse the linked list
    reverseList(&head);
    printf("\nReversed Linked List:\n");
    displayList(head);

    // Sort the linked list
    sortList(&head);
    printf("\nSorted Linked List:\n");
    displayList(head);

    return 0;
}
```

How the Program Works:

1. **Node Structure:** A structure Node is defined to represent each node in the linked list, containing an integer data and a pointer to the next node.
2. **Insertion:** The insertNode function allows users to add nodes to the end of the linked list.
3. **Reversing the List:** The reverseList function reverses the linked list by adjusting the next pointers of each node.

4. **Sorting the List:** The sortList function sorts the linked list in ascending order using a bubble sort algorithm.
5. **Display:** The displayList function prints the elements of the linked list in order.

Example Usage:

Enter the number of nodes: 5

Enter data for node 1: 5

Enter data for node 2: 3

Enter data for node 3: 8

Enter data for node 4: 1

Enter data for node 5: 7

Original Linked List:

5 -> 3 -> 8 -> 1 -> 7 -> NULL

Reversed Linked List:

7 -> 1 -> 8 -> 3 -> 5 -> NULL

Sorted Linked List:

1 -> 3 -> 5 -> 7 -> 8 -> NULL

Q4. Write a program using linked list that accepts two polynomials as input and displays the resultant polynomial after performing the multiplication of the input polynomials.

Ans

C Program to Multiply Two Polynomials Using Linked Lists

c code

```
#include <stdio.h>

#include <stdlib.h>


// Node structure representing a term of the polynomial
```

```
struct Node {
    int coeff; // Coefficient of the term
    int exp;   // Exponent of the term
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int coeff, int exp) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->coeff = coeff;
    newNode->exp = exp;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a term into the polynomial linked list
void insertTerm(struct Node** poly, int coeff, int exp) {
    struct Node* newNode = createNode(coeff, exp);
    if (*poly == NULL) {
        *poly = newNode;
    } else {
        struct Node* temp = *poly;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}
```

```
// Function to multiply two polynomials
struct Node* multiplyPolynomials(struct Node* poly1, struct Node* poly2) {
    struct Node* result = NULL;

    // Traverse through the first polynomial
    for (struct Node* ptr1 = poly1; ptr1 != NULL; ptr1 = ptr1->next) {
        // For each term in the first polynomial, traverse through the second polynomial
        for (struct Node* ptr2 = poly2; ptr2 != NULL; ptr2 = ptr2->next) {
            // Multiply the coefficients and add the exponents
            int coeff = ptr1->coeff * ptr2->coeff;
            int exp = ptr1->exp + ptr2->exp;
            insertTerm(&result, coeff, exp);
        }
    }

    return result;
}

// Function to combine terms with the same exponent in the polynomial
void simplifyPolynomial(struct Node** poly) {
    struct Node* ptr1 = *poly;
    while (ptr1 != NULL && ptr1->next != NULL) {
        struct Node* ptr2 = ptr1;
        while (ptr2->next != NULL) {
            if (ptr1->exp == ptr2->next->exp) {
                ptr1->coeff += ptr2->next->coeff;
                struct Node* temp = ptr2->next;
                ptr2->next = ptr2->next->next;
                free(temp);
```

```
            } else {
                ptr2 = ptr2->next;
            }
        }
        ptr1 = ptr1->next;
    }
}

// Function to print the polynomial
void printPolynomial(struct Node* poly) {
    struct Node* temp = poly;
    while (temp != NULL) {
        printf("%dx^%d", temp->coeff, temp->exp);
        if (temp->next != NULL) {
            printf(" + ");
        }
        temp = temp->next;
    }
    printf("\n");
}

// Main function
int main() {
    struct Node* poly1 = NULL;
    struct Node* poly2 = NULL;
    struct Node* result = NULL;

    // Example input for polynomial 1: 3x^2 + 5x^1 + 6
    insertTerm(&poly1, 3, 2);
```

```
    insertTerm(&poly1, 5, 1);
    insertTerm(&poly1, 6, 0);

    // Example input for polynomial 2: 6x^1 + 4
    insertTerm(&poly2, 6, 1);
    insertTerm(&poly2, 4, 0);

    // Multiply the two polynomials
    result = multiplyPolynomials(poly1, poly2);

    // Simplify the result by combining like terms
    simplifyPolynomial(&result);

    // Print the resulting polynomial
    printf("Resultant polynomial after multiplication: ");
    printPolynomial(result);

    return 0;
}
```

Explanation:

1. **Node Structure:**
    - Each node of the linked list represents a term of the polynomial with a coefficient (coeff) and an exponent (exp).
2. **Insertion Function:**
    - insertTerm: This function inserts a term at the end of the polynomial linked list.
3. **Multiplication Function:**
    - multiplyPolynomials: This function multiplies each term of the first polynomial with every term of the second polynomial and stores the result in a new linked list.

4. **Simplification Function:**
    - simplifyPolynomial: This function combines the terms in the resultant polynomial that have the same exponent by summing their coefficients.
5. **Print Function:**
    - printPolynomial: This function prints the polynomial in a readable format.
6. **Main Function:**
    - The main function initializes two example polynomials, multiplies them, simplifies the result, and then prints the resultant polynomial.

Example:

For the polynomials:

- Polynomial 1: 3x2+5x+63x^2 + 5x + 63x2+5x+6
- Polynomial 2: 6x+46x + 46x+4

The output will be:

Resultant polynomial after multiplication: 18x^3 + 38x^2 + 39x^1 + 24

This C program allows you to multiply two polynomials using linked lists effectively.

Q5. Write a program to implement doubly linked list of integers as user inputs and perform the following operations:

(i) Calculate and display the sum of all the even integers of the doubly linked list
(ii) Insert new elements at the beginning, in the middle and at the end of the linked list

Ans

C Program for Doubly Linked List Operations

C code

```
#include <stdio.h>

#include <stdlib.h>


// Define the structure for a node in the doubly linked list

struct Node {

  int data;
```

```
    struct Node* prev;
    struct Node* next;
};

// Function to create a new node with the given data
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;
}

// Function to insert a node at the beginning of the doubly linked list
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        newNode->next = *head;
        (*head)->prev = newNode;
        *head = newNode;
    }
}

// Function to insert a node at the end of the doubly linked list
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
```

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

// Function to insert a node in the middle of the doubly linked list
void insertInMiddle(struct Node** head, int data, int position) {
    struct Node* newNode = createNode(data);
    struct Node* temp = *head;

    if (position == 1) {
        insertAtBeginning(head, data);
        return;
    }

    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }

    if (temp == NULL || temp->next == NULL) {
        insertAtEnd(head, data);
```

```
    } else {
        newNode->next = temp->next;
        temp->next->prev = newNode;
        newNode->prev = temp;
        temp->next = newNode;
    }
}

// Function to calculate and display the sum of all even integers in the doubly linked list
void sumOfEvenIntegers(struct Node* head) {
    int sum = 0;
    struct Node* temp = head;
    while (temp != NULL) {
        if (temp->data % 2 == 0) {
            sum += temp->data;
        }
        temp = temp->next;
    }
    printf("Sum of all even integers: %d\n", sum);
}

// Function to display the contents of the doubly linked list
void displayList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
```

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

// Main function to demonstrate the operations
int main() {
    struct Node* head = NULL;
    int choice, data, position;

    while (1) {
        printf("\nDoubly Linked List Operations:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert in Middle\n");
        printf("3. Insert at End\n");
        printf("4. Calculate Sum of Even Integers\n");
        printf("5. Display List\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert at the beginning: ");
                scanf("%d", &data);
                insertAtBeginning(&head, data);
                break;

            case 2:
                printf("Enter data to insert in the middle: ");
```

```
            scanf("%d", &data);
            printf("Enter the position to insert the data: ");
            scanf("%d", &position);
            insertInMiddle(&head, data, position);
            break;

        case 3:
            printf("Enter data to insert at the end: ");
            scanf("%d", &data);
            insertAtEnd(&head, data);
            break;

        case 4:
            sumOfEvenIntegers(head);
            break;

        case 5:
            printf("Doubly Linked List: ");
            displayList(head);
            break;

        case 6:
            exit(0);

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

```
    return 0;
}
```

Explanation:

1. **Node Structure:**
    - The Node structure defines a node in the doubly linked list with three members: data, prev (pointer to the previous node), and next (pointer to the next node).
2. **Insertion Functions:**
    - **insertAtBeginning**: Inserts a new node at the beginning of the list.
    - **insertAtEnd**: Inserts a new node at the end of the list.
    - **insertInMiddle**: Inserts a new node at a specified position in the list. If the position is invalid or beyond the list's length, it inserts at the end.
3. **Sum of Even Integers:**
    - **sumOfEvenIntegers**: Traverses the list, sums up all even integers, and displays the sum.
4. **Display Function:**
    - **displayList**: Traverses and prints all the elements of the doubly linked list.
5. **Main Function:**
    - The main function provides a menu-driven interface for the user to perform operations like inserting elements at different positions, calculating the sum of even integers, and displaying the list.

How to Use:

- **Insertions**: The user can insert elements at the beginning, in the middle at a specific position, or at the end of the list.
- **Sum Calculation**: The user can calculate and display the sum of all even integers in the list.
- **Display List**: The user can display the current state of the list.

This program is a complete implementation of a doubly linked list in C, capable of performing the required operations.

Q6. Write a program in C to sort user input data using bubble sort method. Also, print the number of swaps and comparison operations performed for sorting the given data set.

Ans

C Program for Bubble Sort with Count of Swaps and Comparisons

c

Copy code

```
#include <stdio.h>


// Function to perform Bubble Sort
void bubbleSort(int arr[], int n, int* swaps, int* comparisons) {
  int temp;
  *swaps = 0;       // Initialize swap count
  *comparisons = 0;  // Initialize comparison count

  // Outer loop for each pass
  for (int i = 0; i < n-1; i++) {
    // Inner loop for comparing adjacent elements
    for (int j = 0; j < n-i-1; j++) {
      (*comparisons)++;  // Increment comparison count
      if (arr[j] > arr[j+1]) {
        // Swap if elements are in the wrong order
        temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
        (*swaps)++;  // Increment swap count
      }
    }
  }
```

```
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int n;
    int swaps, comparisons;

    // Take input from the user
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Perform Bubble Sort
    bubbleSort(arr, n, &swaps, &comparisons);
```

```
    // Print the sorted array
    printf("Sorted array: ");
    printArray(arr, n);

    // Print the number of swaps and comparisons
    printf("Number of swaps performed: %d\n", swaps);
    printf("Number of comparisons performed: %d\n", comparisons);

    return 0;
}
```

Explanation:

1. **Bubble Sort Function:**
    - The bubbleSort function takes an array arr, its size n, and pointers to swaps and comparisons as parameters.
    - The outer loop controls the number of passes needed.
    - The inner loop compares adjacent elements and swaps them if they are in the wrong order.
    - The swaps counter is incremented each time a swap is performed.
    - The comparisons counter is incremented each time a comparison between two elements is made.
2. **Print Array Function:**
    - The printArray function prints the elements of the array in order.
3. **Main Function:**
    - The program prompts the user to input the number of elements and the elements themselves.
    - It then calls the bubbleSort function to sort the array.
    - After sorting, the program prints the sorted array, the number of swaps, and the number of comparisons.

Example Run:

Enter the number of elements: 5

Enter 5 elements: 64 34 25 12 22

Sorted array: 12 22 25 34 64

Number of swaps performed: 8

Number of comparisons performed: 10

How the Program Works:

- **User Input**: The program takes an array of integers as input from the user.
- **Sorting**: The array is sorted using the Bubble Sort algorithm.
- **Output**: The program prints the sorted array, the total number of swaps performed, and the total number of comparisons made during the sorting process.

This program provides a simple yet effective demonstration of the Bubble Sort algorithm, along with tracking important metrics like the number of swaps and comparisons.

Q7. Write a program to convert an infix expression to a prefix expression. Use appropriate data structure.

Ans

C Program to Convert Infix Expression to Prefix Expression

c code

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>


#define MAX 100


// Stack structure

struct Stack {
```

```
    int top;
    char items[MAX];
};

// Function to initialize the stack
void initStack(struct Stack* s) {
    s->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* s) {
    return s->top == -1;
}

// Function to check if the stack is full
int isFull(struct Stack* s) {
    return s->top == MAX - 1;
}

// Function to push an element onto the stack
void push(struct Stack* s, char value) {
    if (isFull(s)) {
        printf("Stack Overflow\n");
    } else {
        s->items[++(s->top)] = value;
    }
}
```

```
// Function to pop an element from the stack
char pop(struct Stack* s) {
    if (isEmpty(s)) {
        printf("Stack Underflow\n");
        return '\0';
    } else {
        return s->items[(s->top)--];
    }
}

// Function to get the top element of the stack
char peek(struct Stack* s) {
    if (isEmpty(s)) {
        return '\0';
    } else {
        return s->items[s->top];
    }
}

// Function to check if a character is an operator
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^');
}

// Function to return precedence of operators
int precedence(char c) {
    switch (c) {
        case '+':
```

```
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
    }
    return 0;
}

// Function to reverse a string
void reverse(char* exp) {
    int length = strlen(exp);
    for (int i = 0; i < length / 2; i++) {
        char temp = exp[i];
        exp[i] = exp[length - i - 1];
        exp[length - i - 1] = temp;
    }
}

// Function to convert infix to postfix expression
void infixToPostfix(char* infix, char* postfix) {
    struct Stack s;
    initStack(&s);
    int i, j = 0;
    char c;
```

```
    for (i = 0; infix[i] != '\0'; i++) {
        c = infix[i];

        if (isalnum(c)) {
            postfix[j++] = c;  // Add operands to the postfix expression
        } else if (c == '(') {
            push(&s, c);  // Push '(' to the stack
        } else if (c == ')') {
            while (!isEmpty(&s) && peek(&s) != '(') {
                postfix[j++] = pop(&s);  // Pop until '(' is found
            }
            pop(&s);  // Discard the '(' from the stack
        } else if (isOperator(c)) {
            while (!isEmpty(&s) && precedence(c) <= precedence(peek(&s))) {
                postfix[j++] = pop(&s);  // Pop higher precedence operators
            }
            push(&s, c);  // Push the current operator
        }
    }

    while (!isEmpty(&s)) {
        postfix[j++] = pop(&s);  // Pop all the remaining operators
    }
    postfix[j] = '\0';  // Null-terminate the postfix expression
}

// Function to convert infix to prefix expression
void infixToPrefix(char* infix, char* prefix) {
```

```
    // Reverse the infix expression
    reverse(infix);

    // Replace '(' with ')' and vice versa
    for (int i = 0; infix[i] != '\0'; i++) {
        if (infix[i] == '(') {
            infix[i] = ')';
        } else if (infix[i] == ')') {
            infix[i] = '(';
        }
    }

    // Convert the modified infix to postfix
    char postfix[MAX];
    infixToPostfix(infix, postfix);

    // Reverse the postfix expression to get the prefix expression
    reverse(postfix);

    // Copy the postfix to prefix
    strcpy(prefix, postfix);
}

// Main function
int main() {
    char infix[MAX], prefix[MAX];

    // Input the infix expression
```

```
    printf("Enter an infix expression: ");
    gets(infix);

    // Convert infix to prefix
    infixToPrefix(infix, prefix);

    // Output the prefix expression
    printf("Prefix expression: %s\n", prefix);

    return 0;
}
```

Explanation:

1. **Stack Structure:**
    - The Stack structure and its associated functions manage the stack operations required for the conversion process.
2. **Reverse Function:**
    - reverse: Reverses the string to facilitate the infix-to-postfix conversion process needed for obtaining the prefix expression.
3. **Infix to Postfix Conversion:**
    - infixToPostfix: Converts the given infix expression to postfix by utilizing the stack.
4. **Infix to Prefix Conversion:**
    - infixToPrefix: Converts an infix expression to a prefix by first reversing the infix expression, converting it to postfix, and then reversing the resulting postfix expression.
5. **Main Function:**
    - The user inputs the infix expression, and the program converts it to a prefix expression, which is then printed.

Example Run:

mathematica

Copy code

Enter an infix expression: (A-B/C)*(A/K-L)

Prefix expression: *-A/BC-/AKL

Steps to Convert:

1. **Reverse Infix**: The infix expression (A-B/C)*(A/K-L) is reversed to become)(L-K/A(*)(C/B-A.
2. **Infix to Postfix**: Convert this reversed infix expression to postfix, which yields LKA/-CB/-A*.
3. **Reverse Postfix**: Reverse the postfix expression to get the final prefix expression: *-A/BC-/AKL.

This C program effectively demonstrates the conversion of infix to prefix using stacks and string manipulation techniques.

Q8. Write a program in 'C' language for the creation of a Red Black tree. Also,implement insertion and deletion operations.

Ans

C Program for Red-Black Tree with Insertion and Deletion

c code

```
#include <stdio.h>

#include <stdlib.h>


// Enum for color

enum Color { RED, BLACK };


// Structure of the Red-Black Tree Node

struct Node {

  int data;

  enum Color color;

  struct Node *left, *right, *parent;

};
```

```
// Function to create a new node
struct Node* createNode(int data) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->data = data;
   node->color = RED;
   node->left = node->right = node->parent = NULL;
   return node;
}

// Function to perform a left rotate
void leftRotate(struct Node **root, struct Node *x) {
   struct Node *y = x->right;
   x->right = y->left;

   if (y->left != NULL)
      y->left->parent = x;

   y->parent = x->parent;

   if (x->parent == NULL)
      *root = y;
   else if (x == x->parent->left)
      x->parent->left = y;
   else
      x->parent->right = y;

   y->left = x;
```

```
    x->parent = y;
}

// Function to perform a right rotate
void rightRotate(struct Node **root, struct Node *y) {
    struct Node *x = y->left;
    y->left = x->right;

    if (x->right != NULL)
        x->right->parent = y;

    x->parent = y->parent;

    if (y->parent == NULL)
        *root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;

    x->right = y;
    y->parent = x;
}

// Function to fix violations after insertion
void fixInsert(struct Node **root, struct Node *z) {
    while (z != *root && z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
```

```
    struct Node *y = z->parent->parent->right;

    if (y != NULL && y->color == RED) {
      z->parent->color = BLACK;
      y->color = BLACK;
      z->parent->parent->color = RED;
      z = z->parent->parent;
    } else {
      if (z == z->parent->right) {
        z = z->parent;
        leftRotate(root, z);
      }
      z->parent->color = BLACK;
      z->parent->parent->color = RED;
      rightRotate(root, z->parent->parent);
    }
  } else {
    struct Node *y = z->parent->parent->left;

    if (y != NULL && y->color == RED) {
      z->parent->color = BLACK;
      y->color = BLACK;
      z->parent->parent->color = RED;
      z = z->parent->parent;
    } else {
      if (z == z->parent->left) {
        z = z->parent;
        rightRotate(root, z);
```

```
            }
            z->parent->color = BLACK;
            z->parent->parent->color = RED;
            leftRotate(root, z->parent->parent);
          }
        }
    }
    (*root)->color = BLACK;
}

// Function to insert a new node
void insert(struct Node **root, int data) {
    struct Node *z = createNode(data);
    struct Node *y = NULL;
    struct Node *x = *root;

    while (x != NULL) {
        y = x;
        if (z->data < x->data)
            x = x->left;
        else
            x = x->right;
    }

    z->parent = y;

    if (y == NULL)
        *root = z;
```

```
    else if (z->data < y->data)
        y->left = z;
    else
        y->right = z;

    fixInsert(root, z);
}

// Function to find the node with minimum value (used in deletion)
struct Node* minimum(struct Node *node) {
    while (node->left != NULL)
        node = node->left;
    return node;
}

// Function to fix violations after deletion
void fixDelete(struct Node **root, struct Node *x) {
    while (x != *root && x->color == BLACK) {
        if (x == x->parent->left) {
            struct Node *w = x->parent->right;

            if (w->color == RED) {
                w->color = BLACK;
                x->parent->color = RED;
                leftRotate(root, x->parent);
                w = x->parent->right;
            }
```

```
    if (w->left->color == BLACK && w->right->color == BLACK) {
      w->color = RED;
      x = x->parent;
    } else {
      if (w->right->color == BLACK) {
        w->left->color = BLACK;
        w->color = RED;
        rightRotate(root, w);
        w = x->parent->right;
      }

      w->color = x->parent->color;
      x->parent->color = BLACK;
      w->right->color = BLACK;
      leftRotate(root, x->parent);
      x = *root;
    }
  } else {
    struct Node *w = x->parent->left;

    if (w->color == RED) {
      w->color = BLACK;
      x->parent->color = RED;
      rightRotate(root, x->parent);
      w = x->parent->left;
    }

    if (w->right->color == BLACK && w->left->color == BLACK) {
```

```
            w->color = RED;
            x = x->parent;
        } else {
            if (w->left->color == BLACK) {
                w->right->color = BLACK;
                w->color = RED;
                leftRotate(root, w);
                w = x->parent->left;
            }

            w->color = x->parent->color;
            x->parent->color = BLACK;
            w->left->color = BLACK;
            rightRotate(root, x->parent);
            x = *root;
        }
    }
  }
  x->color = BLACK;
}

// Function to replace two nodes (used in deletion)
void replaceNode(struct Node **root, struct Node *u, struct Node *v) {
  if (u->parent == NULL)
    *root = v;
  else if (u == u->parent->left)
    u->parent->left = v;
  else
```

```
        u->parent->right = v;

    if (v != NULL)
        v->parent = u->parent;
}

// Function to delete a node
void deleteNode(struct Node **root, struct Node *z) {
    struct Node *y = z;
    struct Node *x;
    enum Color y_original_color = y->color;

    if (z->left == NULL) {
        x = z->right;
        replaceNode(root, z, z->right);
    } else if (z->right == NULL) {
        x = z->left;
        replaceNode(root, z, z->left);
    } else {
        y = minimum(z->right);
        y_original_color = y->color;
        x = y->right;

        if (y->parent == z)
            x->parent = y;
        else {
            replaceNode(root, y, y->right);
            y->right = z->right;
```

```
            y->right->parent = y;
        }

        replaceNode(root, z, y);
        y->left = z->left;
        y->left->parent = y;
        y->color = z->color;
    }

    if (y_original_color == BLACK)
        fixDelete(root, x);
}

// Function to search for a node
struct Node* search(struct Node* root, int data) {
    if (root == NULL || root->data == data)
        return root;

    if (data < root->data)
        return search(root->left, data);

    return search(root->right, data);
}

// Function to print the tree (In-order traversal)
void inOrder(struct Node *root) {
    if (root != NULL) {
        inOrder(root->left);
```

```
        printf("%d ", root->data);
        inOrder(root->right);
    }
}

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

    while (1) {
        printf("\nRed-Black Tree Operations:\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Search\n");
        printf("4. Display (In-order)\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                insert(&root, data);
                break;

            case 2:
```

```
    printf("Enter data to delete: ");
    scanf("%d", &data);
    struct Node *node = search(root, data);
    if (node != NULL)
      deleteNode(&root, node);
    else
      printf("Data not found!\n");
    break;

  case 3:
    printf("Enter data to search: ");
    scanf("%d", &data);
    if (search(root, data) != NULL)
      printf("Data found!\n");
    else
      printf("Data not found!\n");
    break;

  case 4:
    printf("In-order display: ");
    inOrder(root);
    printf("\n");
    break;

  case 5:
    exit(0);

  default:
```

```
            printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

Explanation:

1. **Node Structure:**
    - Each node contains a data field, color (RED/BLACK), and pointers to its left child, right child, and parent.
2. **Rotation Functions:**
    - leftRotate and rightRotate are used to maintain the Red-Black Tree properties during insertion and deletion.
3. **Insertion:**
    - The insert function adds a node and then calls fixInsert to ensure the Red-Black Tree properties are maintained.
4. **Deletion:**
    - The deleteNode function removes a node and then calls fixDelete to ensure the tree remains balanced.
5. **Search Function:**
    - search allows checking if a particular value exists in the tree.
6. **In-Order Traversal:**
    - inOrder displays the tree elements in sorted order.
7. **Main Menu:**
    - A simple menu-driven interface allows the user to perform insertions, deletions, searches, and display the tree in order.

Example Run:

Red-Black Tree Operations:

1. Insert

2. Delete

3. Search

4. Display (In-order)

5. Exit

Enter your choice: 1

Enter data to insert: 10

Red-Black Tree Operations:

1. Insert

2. Delete

3. Search

4. Display (In-order)

5. Exit

Enter your choice: 1

Enter data to insert: 20

Red-Black Tree Operations:

1. Insert

2. Delete

3. Search

4. Display (In-order)

5. Exit

Enter your choice: 4

In-order display: 10 20

This program demonstrates the creation, insertion, deletion, and traversal operations of a Red-Black Tree. The key balancing mechanisms (rotations and color adjustments) ensure that the tree remains balanced during insertions and deletions.