**Experiment 1**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 10// *Define maximum array size*

int main() {

    int a[MAX\_SIZE];

    int element, i, loc, size, n = 0, j = 0;

    // *Input array size*

    printf("Enter the size of an array (Max %d): ", MAX\_SIZE);

    scanf("%d", &size);

    // *Validate array size*

    if (size > MAX\_SIZE || size <= 0) {

        printf("Invalid size! Please enter a value between 1 and %d.\n", MAX\_SIZE);

        return 1;

    }

    // *Input array elements*

    printf("Enter %d array elements: ", size);

    for (i = 0; i < size; i++) {

        scanf("%d", &a[i]);

    }

    // *Display list before insertion*

    printf("List before Insertion: ");

    for (i = 0; i < size; i++) {

        printf("%d ", a[i]);

    }

    // *Insertion operation*

    if (size < MAX\_SIZE) {

        printf("\nEnter an element to insert: ");

        scanf("%d", &element);

        printf("Enter a position to insert (0 to %d): ", size);

        scanf("%d", &loc);

        // *Validate position*

        if (loc < 0 || loc > size) {

            printf("Invalid position! Position should be between 0 and %d.\n", size);

        } else {

            // *Shift elements to the right*

            for (i = size; i > loc; i--) {

                a[i] = a[i - 1];

            }

            a[loc] = element;

            size++;// *Increase size after insertion*

        }

    } else {

        printf("\nArray is full! Cannot insert an element.\n");

    }

    // *Display list after insertion*

    printf("\nList after Insertion: ");

    for (i = 0; i < size; i++) {

        printf("%d ", a[i]);

    }

    // *Deletion operation*

    printf("\nEnter an element to delete: ");

    scanf("%d", &n);

    int found = 0;// *Flag to check if element is found*

    for (i = 0; i < size; i++) {

        if (a[i] == n) {

            found = 1;

            for (j = i; j < size - 1; j++) {

                a[j] = a[j + 1];

            }

            size--;// *Reduce size after deletion*

            break;// *Stop after deleting the first occurrence*

        }

    }

    if (!found) {

        printf("Element not found!\n");

    }

    // *Display list after deletion*

    printf("List after deletion: ");

    for (i = 0; i < size; i++) {

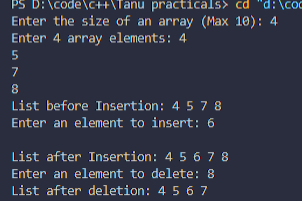
        printf("%d ", a[i]);

    }

    return 0;

}

**Output:**



**Experiment 2**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 5// *Define maximum size of stack*

// *Function to check if the stack is full*

int isFull(int top) {

    return top == MAX - 1;

}

// *Function to check if the stack is empty*

int isEmpty(int top) {

    return top == -1;

}

// *Function to push an element onto the stack*

void push(int arr[], int \*top, int value) {

    if (isFull(\*top)) {

        printf("Stack Overflow! Cannot push %d\n", value);

        return;

    }

    arr[++(\*top)] = value;

    printf("%d pushed to stack\n", value);

}

// *Function to pop an element from the stack*

int pop(int arr[], int \*top) {

    if (isEmpty(\*top)) {

        printf("Stack Underflow! No element to pop.\n");

        return -1;// *Return -1 if stack is empty*

    }

    int poppedValue = arr[\*top];

    (\*top)--;// *Decrease top pointer*

    return poppedValue;

}

// *Function to peek at the top element of the stack*

int peek(int arr[], int top) {

    if (isEmpty(top)) {

        printf("Stack is empty! No element to peek.\n");

        return -1;

    }

    return arr[top];

}

// *Function to display the stack elements*

void display(int arr[], int top) {

    if (isEmpty(top)) {

        printf("Stack is empty!\n");

        return;

    }

    printf("Stack elements are: ");

    for (int i = 0; i <= top; i++) {

        printf("%d ", arr[i]);

    }

    printf("\n");

}

int main() {

    int stack[MAX];// *Create an array for the stack*

    int top = -1;// *Initialize the stack as empty*

    // *Push elements onto the stack*

    push(stack, &top, 10);

    push(stack, &top, 20);

    push(stack, &top, 30);

    push(stack, &top, 40);

    push(stack, &top, 50);

    // *Display stack elements*

    display(stack, top);

    // *Push one more element (should trigger stack overflow)*

    push(stack, &top, 60);

    // *Pop an element from the stack*

    printf("%d popped from stack\n", pop(stack, &top));

    // *Peek at the top element of the stack*

    printf("Top element is %d\n", peek(stack, top));

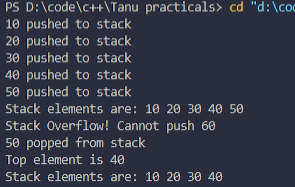
    // *Display stack elements after pop*

    display(stack, top);

    return 0;

}

**Output:**



**Experiment 3**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define SIZE 5// *Maximum size of the queue*

typedef struct {

    int items[SIZE];

    int front, rear;

} Queue;

// *Initialize the queue*

void initializeQueue(Queue \*q) {

    q->front = -1;

    q->rear = -1;

}

// *Check if the queue is empty*

int isEmpty(Queue \*q) {

    return q->front == -1;

}

// *Check if the queue is full*

int isFull(Queue \*q) {

    return q->rear == SIZE - 1;

}

// *Enqueue operation*

void enqueue(Queue \*q, int element) {

    if (isFull(q)) {

        printf("Queue is full! Cannot enqueue %d\n", element);

        return;

    }

    if (isEmpty(q)) {

        q->front = 0;// *Set front to 0 if queue was empty*

    }

    q->rear++;

    q->items[q->rear] = element;

    printf("Enqueued: %d\n", element);

}

// *Dequeue operation*

int dequeue(Queue \*q) {

    if (isEmpty(q)) {

        printf("Queue is empty! Cannot dequeue.\n");

        return -1;

    }

    int element = q->items[q->front];

    if (q->front == q->rear) {

        // *Reset queue when it becomes empty*

        q->front = q->rear = -1;

    } else {

        q->front++;

    }

    printf("Dequeued: %d\n", element);

    return element;

}

// *Display the queue*

void displayQueue(Queue \*q) {

    if (isEmpty(q)) {

        printf("Queue is empty!\n");

        return;

    }

    printf("Queue elements: ");

    for (int i = q->front; i <= q->rear; i++) {

        printf("%d ", q->items[i]);

    }

    printf("\n");

}

int main() {

    Queue q;

    initializeQueue(&q);

    enqueue(&q, 10);

    enqueue(&q, 20);

    enqueue(&q, 30);

    displayQueue(&q);

    dequeue(&q);

    displayQueue(&q);

    enqueue(&q, 40);

    enqueue(&q, 50);

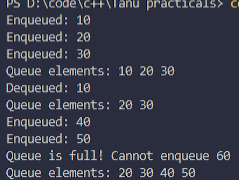
    enqueue(&q, 60);// *Queue is full*

    displayQueue(&q);

    return 0;

}

**Output:**



**Experiment 4**

**Code:**

#include <stdio.h>

#define SIZE 5// *Define the maximum size of the queue*

int queue[SIZE], front = -1, rear = -1;

// *Function to check if the queue is full*

int isFull() {

    return ((rear + 1) % SIZE == front);

}

// *Function to check if the queue is empty*

int isEmpty() {

    return (front == -1);

}

// *Function to insert an element into the queue*

void enqueue(int value) {

    if (isFull()) {

        printf("Queue is full!\n");

        return;

    }

    if (isEmpty()) {

        front = 0;

    }

    rear = (rear + 1) % SIZE;

    queue[rear] = value;

    printf("Inserted %d\n", value);

}

// *Function to remove an element from the queue*

void dequeue() {

    if (isEmpty()) {

        printf("Queue is empty!\n");

        return;

    }

    printf("Deleted %d\n", queue[front]);

    if (front == rear) {

        // *Reset queue when it becomes empty*

        front = rear = -1;

    } else {

        front = (front + 1) % SIZE;

    }

}

// *Function to display the queue*

void display() {

    if (isEmpty()) {

        printf("Queue is empty!\n");

        return;

    }

    printf("Queue elements: ");

    int i = front;

    while (1) {

        printf("%d ", queue[i]);

        if (i == rear)

            break;

        i = (i + 1) % SIZE;

    }

    printf("\n");

}

int main() {

    enqueue(10);

    enqueue(20);

    enqueue(30);

    enqueue(40);

    enqueue(50);

    enqueue(60);// *Queue is full*

    display();

    dequeue();

    dequeue();

    display();

    enqueue(60);

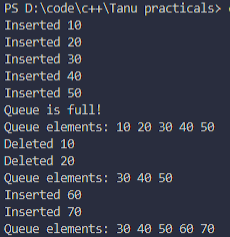
    enqueue(70);

    display();

    return 0;

}

**Output:**

****

**Experiment 5**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// *Define a node structure*

struct Node {

    int data;

    struct Node\* next;

};

struct Node\* head = NULL;// *Initialize head as NULL*

// *Function to insert a node at a specific position*

void insertAtPosition(int value, int position) {

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->data = value;

    newNode->next = NULL;

    if (position == 0 || head == NULL) {

        newNode->next = head;

        head = newNode;

        return;

    }

    struct Node\* temp = head;

    for (int i = 0; temp != NULL && i < position - 1; i++) {

        temp = temp->next;

    }

    if (temp == NULL) {

        printf("Invalid position!\n");

        free(newNode);

        return;

    }

    newNode->next = temp->next;

    temp->next = newNode;

}

// *Function to delete a node from a specific position*

void deleteAtPosition(int position) {

    if (head == NULL) {

        printf("List is empty!\n");

        return;

    }

    struct Node\* temp = head;

    if (position == 0) {

        head = temp->next;

        free(temp);

        return;

    }

    struct Node\* prev = NULL;

    for (int i = 0; temp != NULL && i < position; i++) {

        prev = temp;

        temp = temp->next;

    }

    if (temp == NULL) {

        printf("Invalid position!\n");

        return;

    }

    prev->next = temp->next;

    free(temp);

}

// *Function to display the linked list*

void display() {

    struct Node\* temp = head;

    if (temp == NULL) {

        printf("List is empty!\n");

        return;

    }

    printf("Linked List: ");

    while (temp != NULL) {

        printf("%d -> ", temp->data);

        temp = temp->next;

    }

    printf("NULL\n");

}

int main() {

    insertAtPosition(10, 0);

    insertAtPosition(20, 1);

    insertAtPosition(30, 2);

    display();

    deleteAtPosition(1);

    display();

    return 0;

}

**Output :**

****

**Experiment 6**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// *Define a node structure*

struct Node {

    int data;

    struct Node\* next;

    struct Node\* prev;

};

struct Node\* head = NULL;// *Initialize head as NULL*

// *Function to insert a node at a specific position*

void insertAtPosition(int value, int position) {

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->data = value;

    newNode->next = NULL;

    newNode->prev = NULL;

    if (position == 0 || head == NULL) {

        newNode->next = head;

        if (head != NULL) {

            head->prev = newNode;

        }

        head = newNode;

        return;

    }

    struct Node\* temp = head;

    for (int i = 0; temp != NULL && i < position - 1; i++) {

        temp = temp->next;

    }

    if (temp == NULL) {

        printf("Invalid position!\n");

        free(newNode);

        return;

    }

    newNode->next = temp->next;

    if (temp->next != NULL) {

        temp->next->prev = newNode;

    }

    temp->next = newNode;

    newNode->prev = temp;

}

// *Function to delete a node from a specific position*

void deleteAtPosition(int position) {

    if (head == NULL) {

        printf("List is empty!\n");

        return;

    }

    struct Node\* temp = head;

    if (position == 0) {

        head = temp->next;

        if (head != NULL) {

            head->prev = NULL;

        }

        free(temp);

        return;

    }

    for (int i = 0; temp != NULL && i < position; i++) {

        temp = temp->next;

    }

    if (temp == NULL) {

        printf("Invalid position!\n");

        return;

    }

    if (temp->next != NULL) {

        temp->next->prev = temp->prev;

    }

    if (temp->prev != NULL) {

        temp->prev->next = temp->next;

    }

    free(temp);

}

// *Function to display the doubly linked list*

void display() {

    struct Node\* temp = head;

    if (temp == NULL) {

        printf("List is empty!\n");

        return;

    }

    printf("Doubly Linked List: ");

    while (temp != NULL) {

        printf("%d <-> ", temp->data);

        temp = temp->next;

    }

    printf("NULL\n");

}

int main() {

    insertAtPosition(10, 0);

    insertAtPosition(20, 1);

    insertAtPosition(30, 2);

    display();

    deleteAtPosition(1);

    display();

    return 0;

}

**Output:**

****

**Experiment 7**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// *Define a node structure*

struct Node {

    int data;

    struct Node\* next;

    struct Node\* prev;

};

struct Node\* front = NULL;

struct Node\* rear = NULL;

// *Function to insert an element at the front*

void insertFront(int value) {

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->data = value;

    newNode->next = front;

    newNode->prev = NULL;

    if (front != NULL) {

        front->prev = newNode;

    }

    front = newNode;

    if (rear == NULL) {

        rear = newNode;

    }

}

// *Function to insert an element at the rear*

void insertRear(int value) {

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->data = value;

    newNode->next = NULL;

    newNode->prev = rear;

    if (rear != NULL) {

        rear->next = newNode;

    }

    rear = newNode;

    if (front == NULL) {

        front = newNode;

    }

}

// *Function to delete an element from the front*

void deleteFront() {

    if (front == NULL) {

        printf("Deque is empty!\n");

        return;

    }

    struct Node\* temp = front;

    front = front->next;

    if (front != NULL) {

        front->prev = NULL;

    } else {

        rear = NULL;

    }

    free(temp);

}

// *Function to delete an element from the rear*

void deleteRear() {

    if (rear == NULL) {

        printf("Deque is empty!\n");

        return;

    }

    struct Node\* temp = rear;

    rear = rear->prev;

    if (rear != NULL) {

        rear->next = NULL;

    } else {

        front = NULL;

    }

    free(temp);

}

// *Function to display the deque*

void display() {

    struct Node\* temp = front;

    if (temp == NULL) {

        printf("Deque is empty!\n");

        return;

    }

    printf("Deque: ");

    while (temp != NULL) {

        printf("%d <-> ", temp->data);

        temp = temp->next;

    }

    printf("NULL\n");

}

int main() {

    insertFront(10);

    insertRear(20);

    insertFront(5);

    insertRear(30);

    display();

    deleteFront();

    deleteRear();

    display();

    return 0;

}

**Output:**

****

**Experiment 8**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// *Define a structure for a BST node*

struct Node {

    int data;

    struct Node\* left;

    struct Node\* right;

};

// *Function to create a new node*

struct Node\* createNode(int value) {

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->data = value;

    newNode->left = newNode->right = NULL;

    return newNode;

}

// *Function to insert a node in BST*

struct Node\* insert(struct Node\* root, int value) {

    if (root == NULL) {

        return createNode(value);

    }

    if (value < root->data) {

        root->left = insert(root->left, value);

    } else {

        root->right = insert(root->right, value);

    }

    return root;

}

// *Function for Inorder Traversal*

void inorder(struct Node\* root) {

    if (root != NULL) {

        inorder(root->left);

        printf("%d ", root->data);

        inorder(root->right);

    }

}

// *Function for Preorder Traversal*

void preorder(struct Node\* root) {

    if (root != NULL) {

        printf("%d ", root->data);

        preorder(root->left);

        preorder(root->right);

    }

}

// *Function for Postorder Traversal*

void postorder(struct Node\* root) {

    if (root != NULL) {

        postorder(root->left);

        postorder(root->right);

        printf("%d ", root->data);

    }

}

int main() {

    struct Node\* root = NULL;

    root = insert(root, 50);

    insert(root, 30);

    insert(root, 70);

    insert(root, 20);

    insert(root, 40);

    insert(root, 60);

    insert(root, 80);

    printf("Inorder Traversal: ");

    inorder(root);

    printf("\n");

    printf("Preorder Traversal: ");

    preorder(root);

    printf("\n");

    printf("Postorder Traversal: ");

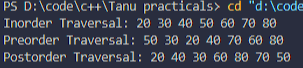
    postorder(root);

    printf("\n");

    return 0;

}

**Output:**

****

**Experiment 9**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define MAX 100

// *Stack structure*

struct Stack {

    char arr[MAX];

    int top;

};

// *Initialize stack*

void initStack(struct Stack\* s) {

    s->top = -1;

}

// *Check if stack is empty*

int isEmpty(struct Stack\* s) {

    return s->top == -1;

}

// *Push element onto stack*

void push(struct Stack\* s, char c) {

    if (s->top < MAX - 1) {

        s->arr[++s->top] = c;

    }

}

// *Pop element from stack*

char pop(struct Stack\* s) {

    return isEmpty(s) ? '\0' : s->arr[s->top--];

}

// *Get top element without popping*

char peek(struct Stack\* s) {

    return isEmpty(s) ? '\0' : s->arr[s->top];

}

// *Get precedence of an operator*

int precedence(char op) {

    switch (op) {

        case '^': return 3;

        case '\*':

        case '/': return 2;

        case '+':

        case '-': return 1;

        default: return 0;

    }

}

// *Convert infix to postfix*

void infixToPostfix(char\* infix, char\* postfix) {

    struct Stack s;

    initStack(&s);

    int j = 0;

    for (int i = 0; i < strlen(infix); i++) {

        char c = infix[i];

        // *If operand, add directly to output*

        if (isalnum(c)) {

            postfix[j++] = c;

        }

        // *If '(', push to stack*

        else if (c == '(') {

            push(&s, c);

        }

        // *If ')', pop and add to output until '(' is found*

        else if (c == ')') {

            while (!isEmpty(&s) && peek(&s) != '(') {

                postfix[j++] = pop(&s);

            }

            pop(&s);// *Remove '('*

        }

        // *If operator, pop higher precedence operators from stack*

        else {

            while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(c)) {

                postfix[j++] = pop(&s);

            }

            push(&s, c);

        }

    }

    // *Pop remaining operators from stack*

    while (!isEmpty(&s)) {

        postfix[j++] = pop(&s);

    }

    postfix[j] = '\0';// *Null terminate the string*

}

int main() {

    char infix[MAX], postfix[MAX];

    printf("Enter an infix expression: ");

    scanf("%s", infix);

    infixToPostfix(infix, postfix);

    printf("Postfix expression: %s\n", postfix);

    return 0;

}

**Output:**

****

**Experiment 10**

**Code:**

#include <stdio.h>

#include <stdlib.h>

// *Structure for a tree node*

struct Node {

    int data;

    struct Node\* left;

    struct Node\* right;

};

// *Function to create a new tree node*

struct Node\* createNode(int data) {

    struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

    newNode->data = data;

    newNode->left = newNode->right = NULL;

    return newNode;

}

// *Function to count the number of leaf nodes*

int countLeafNodes(struct Node\* root) {

    if (root == NULL) {

        return 0;

    }

    // *If it's a leaf node, return 1*

    if (root->left == NULL && root->right == NULL) {

        return 1;

    }

    // *Recursively count in left and right subtrees*

    return countLeafNodes(root->left) + countLeafNodes(root->right);

}

// *Function to find the largest leaf node*

int findLargestLeaf(struct Node\* root) {

    if (root == NULL) {

        return -1;// *Return a value indicating an empty tree*

    }

    // *If it's a leaf node, return its data*

    if (root->left == NULL && root->right == NULL) {

        return root->data;

    }

    // *Recursively find the largest leaf in left and right subtrees*

    int leftLargest = findLargestLeaf(root->left);

    int rightLargest = findLargestLeaf(root->right);

    // *Return the largest among the two*

    return (leftLargest > rightLargest) ? leftLargest : rightLargest;

}

int main() {

    // *Constructing a sample binary tree*

    struct Node\* root = createNode(10);

    root->left = createNode(20);

    root->right = createNode(30);

    root->left->left = createNode(40);

    root->left->right = createNode(50);

    root->right->right = createNode(60);

    // *Counting the number of leaf nodes*

    int leafCount = countLeafNodes(root);

    printf("Number of leaf nodes: %d\n", leafCount);

    // *Finding the largest leaf node*

    int largestLeaf = findLargestLeaf(root);

    printf("Largest leaf node: %d\n", largestLeaf);

    return 0;

}

**Output:**

****