Stacks

1. Stack using arrays:

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

#define RED "\x1B[31m"
#define GREEN "\x1B[32m"
#define YELLOW "\x1B[33m"
#define RESET "\x1B[0m"

struct Stack
{
    int top;
    int* arr;
    int max;
};
struct Stack creatStack(int size);
void push(struct Stack* a, int data);
int pop(struct Stack* a);
void peek(struct Stack* a);
void display(struct Stack* a);


int main(void)
{
    printf("Enter the size of the stack: ");
    int size;
    scanf("%d", &size);
    struct Stack stack = creatStack(size);
    struct Stack* stackPTR = &stack;

    // while loop with switch case for menu until user exits
    int choice;
    while(1)
    {
        printf("\nEnter 1 to push, 2 to pop, 3 to peek, 4 display, 5 to to
exit: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter the element to push: ");
            int data;
            scanf("%d", &data);
            push(stackPTR, data);
            break;
```

```
            case 2:
                printf("The popped element is: ");
                printf(GREEN "%d\n" RESET, pop(stackPTR));
                break;
            case 3:
                peek(stackPTR);
                break;
            case 4:
                display(stackPTR);
                break;
            case 5:
                printf("Exiting...\n");
                return 0;
                break;
            default:
                printf(YELLOW "Invalid choice!!\n" RESET);
                break;
            }
        }

}

// using arrays

void push(struct Stack* a, int data)
{
    if (a->top == a->max-1)
    {
        printf(RED "Overflow!!\n" RESET);
        return;
    }
    else
    {
        a->top = a->top + 1;
        a->arr[a->top] = data;
        return;
    }
}

int pop(struct Stack* a)
{
    if (a->top == -1)
    {
        printf(RED "Underflow!!\n" RESET);
        //return -1;
    }
    else
    {
        int z =  a->arr[a->top];
        a->top = a->top - 1;
        return z;
    }
}
```

```
void peek(struct Stack* a)
{
    if (a->top == -1)
    {
        printf(RED "Underflow!!\n" RESET);
    }
    else
    {
        printf("The top element is: ");
        printf(GREEN "%d\n" RESET, a->arr[a->top]);
    }
}

void display(struct Stack* a)
{
    if (a->top == -1)
    {
        printf(RED "Underflow!!\n" RESET);
        return;
    }
    else
    {
        for (int i = a->top; i >= 0; i--)
        {
            printf(GREEN "| %d |\n" RESET, a->arr[i]);
        }
        printf(GREEN "|___|\n" RESET);
        return;
    }
}
// create stack
struct Stack creatStack(int size)
{
    struct Stack a;
    a.arr = malloc(sizeof(int) * size);
    a.top = -1;
    a.max = size;
    return a;
}
```

2. Infix to postfix:

```
char* infixToPostfix(struct Stack* stack, char* infix) {
    char* postfix = (char*)malloc(sizeof(char) * (2 * strlen(infix) + 1));
    int postfixIndex = 0;

    for (int i = 0; infix[i] != '\0'; ++i) {
        if (isalnum(infix[i])) {
            postfix[postfixIndex++] = infix[i];
            postfix[postfixIndex++] = ' ';
```

```
        } else if (infix[i] == '(') {
            push(stack, infix[i]);
        } else if (infix[i] == ')') {
            while (stack->top != -1 && peek(stack) != '(') {
                postfix[postfixIndex++] = pop(stack);
                postfix[postfixIndex++] = ' ';
            }
            if (stack->top != -1 && peek(stack) != '(') {
                printf(RED "Invalid expression\n" RESET);
                free(postfix);
                return NULL;
            } else {
                pop(stack);
            }
        } else if (isOperator(infix[i])) {
            while (stack->top != -1 && precedence(infix[i]) <=
precedence(peek(stack))) {
                postfix[postfixIndex++] = pop(stack);
                postfix[postfixIndex++] = ' ';
            }
            push(stack, infix[i]);
        }
    }

    while (stack->top != -1) {
        postfix[postfixIndex++] = pop(stack);
        postfix[postfixIndex++] = ' ';
    }

    postfix[postfixIndex] = '\0';

    return postfix;
}
```

3. Evaluation of postfix:

```
int evaluatePostfix(struct Stack* stack, char* postfix) {
    for (int i = 0; postfix[i] != '\0'; ++i) {
        if (isalnum(postfix[i])) {
            push(stack, postfix[i] - '0');
        } else {
            int operand2 = pop(stack);
            int operand1 = pop(stack);
            int result = performOperation(postfix[i], operand1, operand2);
            push(stack, result);
        }
    }
    return pop(stack);
}

int performOperation(char operator, int operand1, int operand2) {
```

```
    switch (operator) {
        case '+':
            return operand1 + operand2;
        case '-':
            return operand1 - operand2;
        case '*':
            return operand1 * operand2;
        case '/':
            return operand1 / operand2;
        case '^':
            return operand1 ^ operand2;
        default:
            printf(RED "Invalid operator\n" RESET);
            return -1;
    }
}
```

4. Queue using stack:

```
struct Queue {
    struct Stack* stack1;
    struct Stack* stack2;
};

struct Queue* createQueue(int capacity) {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->stack1 = createStack(capacity);
    queue->stack2 = createStack(capacity);
    return queue;
}

void enqueue(struct Queue* queue, int item) {
    while (queue->stack1->top != queue->stack1->capacity - 1) {
        push(queue->stack1, pop(queue->stack2));
    }
    push(queue->stack1, item);
}

int dequeue(struct Queue* queue) {
    while (queue->stack1->top != -1) {
        push(queue->stack2, pop(queue->stack1));
    }
    if (queue->stack2->top == -1) {
        printf("Queue is empty\n");
        return -1;
    }
    return pop(queue->stack2);
}
```

5. Stacks using linked list:

```
struct StackNode {
    int data;
    struct StackNode* next;
};

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

int isEmpty(struct StackNode* root) {
    return !root;
}

struct StackNode* push(struct StackNode* root, int data) {
    struct StackNode* stackNode = newNode(data);
    stackNode->next = root;
    return stackNode;
}

struct StackNode* pop(struct StackNode* root, int* popped) {
    if (isEmpty(root)) {
        printf("Stack underflow\n");
        return NULL;
    }
    struct StackNode* temp = root;
    *popped = temp->data;
    root = root->next;
    free(temp);
    return root;
}

int peek(struct StackNode* root) {
    if (isEmpty(root)) {
        printf("Stack underflow\n");
        return -1;
    }
    return root->data;
}
```

Queues

1. Queue using arrays (not circular):

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

#define RED "\x1B[31m"
#define GREEN "\x1B[32m"
#define YELLOW "\x1B[33m"
#define RESET "\x1B[0m"

struct queue
{
    int* arr;
    int front;
    int rear;
    int max;
};

bool isEmpty(struct queue* q) {
    return (q->front == -1 && q->rear == -1);
}

bool isFull(struct queue* q) {
    return ((q->rear + 1) % q->max == q->front);
}

void enqueue(struct queue* q, int data)
{
    if (isFull(q))
    {
        printf(RED "Overflow\n" RESET);
        return;
    }
    else if (isEmpty(q))
    {
        q->front = 0;
        q->rear = 0;
    }
    else
    {
        q->rear = (q->rear + 1) % q->max;
    }

    q->arr[q->rear] = data;
}

int dequeue(struct queue* q)
{
    if (isEmpty(q))
    {
        printf(RED "Underflow\n" RESET);
        return -1;
    }
```

```
    else if (q->front == q->rear)
    {
        int data = q->arr[q->front];
        q->front = -1;
        q->rear = -1;
        return data;
    }
    else
    {
        int data = q->arr[q->front];
        q->front = (q->front + 1) % q->max;
        return data;
    }
}

void display(struct queue* q)
{
    if (isEmpty(q))
    {
        printf(RED "Underflow\n" RESET);
        return;
    }
    else
    {
        int front = q->front;
        int rear = q->rear;

        do
        {
            printf(GREEN "| %d |\n" RESET, q->arr[front]);
            front = (front + 1) % q->max;
        } while (front != (rear + 1) % q->max);

        printf(GREEN "|___|\n" RESET);
    }
}

struct queue createQueue(int size)
{
    struct queue q;
    q.arr = (int*)malloc(sizeof(int) * size);
    q.front = -1;
    q.rear = -1;
    q.max = size;
    return q;
}
```

---

2. Queue using linked list:

```c
#define RED "\x1B[31m"
#define GREEN "\x1B[32m"
#define YELLOW "\x1B[33m"
#define RESET "\x1B[0m"

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

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

bool isEmpty(struct Queue* q) {
    return (q->front == NULL && q->rear == NULL);
}

void enqueue(struct Queue* q, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    if (isEmpty(q)) {
        q->front = newNode;
        q->rear = newNode;
        newNode->next = newNode;
    } else {
        q->rear->next = newNode;
        newNode->next = q->front;
        q->rear = newNode;
    }
}

int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf(RED "Underflow\n" RESET);
        return -1;
    }

    int data;
    struct Node* temp = q->front;

    if (q->front == q->rear) {
        data = temp->data;
        free(temp);
        q->front = NULL;
        q->rear = NULL;
    } else {
        data = temp->data;
```

```
        q->front = temp->next;
        q->rear->next = q->front;
        free(temp);
    }

    return data;
}

void display(struct Queue* q) {
    if (isEmpty(q)) {
        printf(RED "Underflow\n" RESET);
        return;
    }

    struct Node* current = q->front;

    do {
        printf(GREEN "| %d |\n" RESET, current->data);
        current = current->next;
    } while (current != q->front);
}
```

---

3. Priority Queue:

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

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

int peek(struct Node* head) {
    return head->data;
}

struct Node* pop(struct Node* head) {
    if (head == NULL) {
        printf("Underflow\n");
        return NULL;
    }

    struct Node* temp = head;
    head = head->next;
```

```
    free(temp);
    return head;
}

struct Node* push(struct Node* head, int data, int priority) {
    struct Node* temp = newNode(data, priority);

    if (head == NULL || head->priority > priority) {
        temp->next = head;
        head = temp;
    } else {
        struct Node* start = head;
        while (start->next != NULL && start->next->priority < priority) {
            start = start->next;
        }
        temp->next = start->next;
        start->next = temp;
    }

    return head;
}

int isEmpty(struct Node* head) {
    return head == NULL;
}
```

---

4. Deque:

```
struct Deque {
    int* arr;
    int front;
    int rear;
    int max;
};

struct Deque* createDeque(int size) {
    struct Deque* deque = (struct Deque*)malloc(sizeof(struct Deque));
    deque->arr = (int*)malloc(sizeof(int) * size);
    deque->front = -1;
    deque->rear = 0;
    deque->max = size;
    return deque;
}

bool isEmpty(struct Deque* deque) {
    return (deque->front == -1);
}

bool isFull(struct Deque* deque) {
    return ((deque->front == 0 && deque->rear == deque->max - 1) || deque-
```

```
>front == deque->rear + 1);
}

void insertFront(struct Deque* deque, int data) {
    if (isFull(deque)) {
        printf("Overflow\n");
        return;
    }

    if (deque->front == -1) {
        deque->front = 0;
        deque->rear = 0;
    } else if (deque->front == 0) {
        deque->front = deque->max - 1;
    } else {
        deque->front = deque->front - 1;
    }

    deque->arr[deque->front] = data;
}

void insertRear(struct Deque* deque, int data) {
    if (isFull(deque)) {
        printf("Overflow\n");
        return;
    }

    if (deque->front == -1) {
        deque->front = 0;
        deque->rear = 0;
    } else if (deque->rear == deque->max - 1) {
        deque->rear = 0;
    } else {
        deque->rear = deque->rear + 1;
    }

    deque->arr[deque->rear] = data;
}

void deleteFront(struct Deque* deque) {
    if (isEmpty(deque)) {
        printf("Underflow\n");
        return;
    }

    if (deque->front == deque->rear) {
        deque->front = -1;
        deque->rear = -1;
    } else if (deque->front == deque->max - 1) {
        deque->front = 0;
    } else {
        deque->front = deque->front + 1;
    }
```

```
}

void deleteRear(struct Deque* deque) {
    if (isEmpty(deque)) {
        printf("Underflow\n");
        return;
    }

    if (deque->front == deque->rear) {
        deque->front = -1;
        deque->rear = -1;
    } else if (deque->rear == 0) {
        deque->rear = deque->max - 1;
    } else {
        deque->rear = deque->rear - 1;
    }
}

int getFront(struct Deque* deque) {
    if (isEmpty(deque)) {
        printf("Underflow\n");
        return -1;
    }

    return deque->arr[deque->front];
}

int getRear(struct Deque* deque) {
    if (isEmpty(deque) || deque->rear < 0) {
        printf("Underflow\n");
        return -1;
    }

    return deque->arr[deque->rear];
}
```

---

5. Stack using Queue:

```
struct QueueNode {
    int data;
    struct QueueNode* next;
};

struct Queue {
    struct QueueNode* front;
    struct QueueNode* rear;
};

struct Stack {
    struct Queue* q1;
```

```
    struct Queue* q2;
};

struct QueueNode* newQueueNode(int data) {
    struct QueueNode* temp = (struct QueueNode*)malloc(sizeof(struct
QueueNode));
    temp->data = data;
    temp->next = NULL;
    return temp;
}

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 data) {
    struct QueueNode* temp = newQueueNode(data);

    if (q->rear == NULL) {
        q->front = q->rear = temp;
        return;
    }

    q->rear->next = temp;
    q->rear = temp;
}

int dequeue(struct Queue* q) {
    if (q->front == NULL) {
        printf("Underflow\n");
        return -1;
    }

    struct QueueNode* temp = q->front;
    int data = temp->data;

    q->front = temp->next;

    if (q->front == NULL) {
        q->rear = NULL;
    }

    free(temp);
    return data;
}

struct Stack* createStack() {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->q1 = createQueue();
    stack->q2 = createQueue();
    return stack;
```

```
}

void push(struct Stack* stack, int data) {
    enqueue(stack->q1, data);

    while (stack->q2->front != NULL) {
        enqueue(stack->q1, dequeue(stack->q2));
    }

    struct Queue* temp = stack->q1;
    stack->q1 = stack->q2;
    stack->q2 = temp;
}

int pop(struct Stack* stack) {
    if (stack->q2->front == NULL) {
        printf("Underflow\n");
        return -1;
    }

    return dequeue(stack->q2);
}
```

Trees:

1. Binary Tree:

```
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}

struct Node* constructTree() {
    int data;
    printf("Enter data: ");
    scanf("%d", &data);

    if (data == -1) {
        return NULL;
```

```
    }

    struct Node* root = newNode(data);

    printf("Enter left child of %d\n", data);
    root->left = constructTree();

    printf("Enter right child of %d\n", data);
    root->right = constructTree();

    return root;
}

void printTree(struct Node* root) {
    if (root == NULL) {
        return;
    }

    printf("%d: ", root->data);

    if (root->left != NULL) {
        printf("L %d ", root->left->data);
    }

    if (root->right != NULL) {
        printf("R %d", root->right->data);
    }

    printf("\n");

    printTree(root->left);
    printTree(root->right);
}
```

---

2. Traversals:

```
void preorder(struct Node* root) {
    if (root == NULL) {
        return;
    }

    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

void inorder(struct Node* root) {
    if (root == NULL) {
        return;
    }
```

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

void postorder(struct Node* root) {
    if (root == NULL) {
        return;
    }

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

---

3. Binary Search Tree (using arrays):

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

struct node {
    int* arr;
    int levels;
    int size;
};

struct node* insertNode(struct node* tree, int data) {
    if (tree->arr[0] == 0) {
        tree->arr[0] = data;
        return tree;
    }

    int i = 0;
    while (i < tree->size) {
        if (data < tree->arr[i]) {
            if (2*i+1 > tree->size) {
                i = 2*i+1;
                break;
            }
            if (tree->arr[2*i+1] == 0) {
                tree->arr[2*i+1] = data;
                return tree;
            }
            else {
                i = 2*i+1;
            }
        }
        else {
```

```
            if (2*i+2 > tree->size) {
                i = 2*i+2;
                break;
            }
            if (tree->arr[2*i+2] == 0) {
                tree->arr[2*i+2] = data;
                return tree;
            }
            else {
                i = 2*i+2;
            }
        }
    }

    int oldSize = tree->size;
    tree->size = pow(2, ++(tree->levels)) - 1;
    int* tmp = realloc(tree->arr, (tree->size)*sizeof(int));
    if (tmp == NULL) {
        printf("Reallocation Error!\n");
        exit(1);
    }

    for (int j = 0; j < oldSize; j++) {
        tmp[j] = tree->arr[j];
    }
    for (int j = oldSize; j < tree->size; j++) {
        tmp[j] = 0;
    }
    tree->arr = tmp;

    tree->arr[i] = data;
    return tree;

}

struct node* initialise() {
    struct node* new = malloc(sizeof(struct node));
    new->levels = 1;
    new->size = pow(2, new->levels) - 1;
    new->arr = calloc(new->size, sizeof(int));
    return new;
}

struct node* newTree() {

    struct node* tree = initialise();

    printf("Enter number of elements:");
    int val, num;
    scanf("%d", &num);

    for (int n = 0; n < num; n++)
    {
```

```
        printf("Enter value: ");
        scanf("%d", &val);
        tree = insertNode(tree, val);
    }

    return tree;
}

void printLevelOrder(struct node* tree) {
    if (tree->size == 0) {
        printf("Tree is empty.\n");
        return;
    }

    printf("Tree Structure:\n ");
    int lastSize = pow(2, tree->levels - 1);
    int i = 0, levelSize = 1, level = 1;
    int space = (lastSize - pow(2, level - 1)) / 2;
    for (int j = 0; j < space; j++) printf("  ");


    while (i < tree->size) {
        printf("%d ", tree->arr[i++]);
        if (i == levelSize) {
            printf("\n");
            levelSize = pow(2, ++level) - 1;
            int space = (lastSize - pow(2, level - 1)) / 2;
            for (int j = 0; j < space; j++) printf("  ");
        }
    }
}

void printInorder(struct node* tree, int node) {
    if (node < tree->size && tree->arr[node] != 0) {
        printInorder(tree, 2 * node + 1);
        printf("%d ", tree->arr[node]);
        printInorder(tree, 2 * node + 2);
    }
}

int delIndex(struct node* tree, int data, int i){
    if (i > tree->size) return -1;
    if (tree->arr[i] == 0) return -1;
    if (data == tree->arr[i]) return i;

    if (data < tree->arr[i]) return delIndex(tree, data, 2*i+1);
    if (data > tree->arr[i]) return delIndex(tree, data, 2*i+2);
}

void fixTree(struct node* tree, int i) {
    tree->arr[(i-1)/2] = tree->arr[i];
    tree->arr[i] = 0;
```

```
    if (2*i+1 < tree->size && tree->arr[2*i+1] != 0) fixTree(tree, 2*i+1);
    if (2*i+2 < tree->size && tree->arr[2*i+2] != 0) fixTree(tree, 2*i+2);
}

int inOrderSuccessor(struct node* tree, int delPos) {
    int succ = 2*delPos+2;

    while (2*succ+1 < tree->size && tree->arr[2*succ+1] != 0) {
        succ = 2*succ+1;
    }
    return succ;
}

struct node* delete(struct node* tree, int data) {
    int delPos = delIndex(tree, data, 0);

    // doesn't exist
    if (delPos == -1) return tree;

    // no children
    if (2*delPos+1>tree->size && 2*delPos+2>tree->size) {
        tree->arr[delPos] = 0;
        return tree;
    }
    if (tree->arr[2*delPos+1] == 0 && tree->arr[2*delPos+2] == 0) {
        tree->arr[delPos] = 0;
        return tree;
    }

    // right child
    if (tree->arr[2*delPos+1] == 0 && tree->arr[2*delPos+2] != 0) {
        fixTree(tree, 2*delPos+2);
        return tree;
    }
    // left child
    if (tree->arr[2*delPos+1] != 0 && tree->arr[2*delPos+2] == 0) {
        fixTree(tree, 2*delPos+1);
        return tree;
    }

    // two children
    if (tree->arr[2*delPos+1] != 0 && tree->arr[2*delPos+2] != 0) {
        int IOSuccPos = inOrderSuccessor(tree, delPos);
        int IOSuccVal = tree->arr[IOSuccPos];
        tree = delete(tree, IOSuccVal);
        tree->arr[delPos] = IOSuccVal;
        return tree;
    }
    return tree;
}

void printMenu() {
    printf("\nMenu:\n");
```

```
    printf("1. Create a new tree\n");
    printf("2. Insert a node\n");
    printf("3. Delete a node\n");
    printf("4. Print tree\n");
    printf("5. Print menu\n");
    printf("6. Exit\n");
}

int main(void) {
    struct node* tree = NULL; // Initialize tree to NULL

    int option;
    printMenu();
    do {
        printf("Enter your choice: ");
        scanf("%d", &option);

        switch (option) {
            case 1:
                // Create a new tree
                tree = newTree();
                break;

            case 2:
                // Insert a node
                if (tree == NULL) {
                    printf("Error: Create a new tree first.\n");
                }
                else {
                    int val;
                    printf("Enter value to insert: ");
                    scanf("%d", &val);
                    tree = insertNode(tree, val);
                    printf("Node %d inserted.\n", val);
                }
                break;

            case 3:
                // Delete a node
                if (tree == NULL) {
                    printf("Error: Create a new tree first.\n");
                } else {
                    int val;
                    printf("Enter value to delete: ");
                    scanf("%d", &val);
                    tree = delete(tree, val);
                    printf("Node %d deleted.\n", val);
                }
                break;

            case 4:
                // Print tree
                if (tree == NULL) {
```

```
                    printf("Error: Create a new tree first.\n");
                } else {
                    printf("\nArray in memory:\n");
                    for (int i = 0; i < tree->size; i++) printf("%d", tree-
>arr[i]);
                    printf("\n\nInorder: ");
                    printInorder(tree, 0);
                    printf("\n\nTree Structure: ");
                    printLevelOrder(tree);
                }
                break;

            case 5:
                // Print menu
                printMenu();
                break;

            case 6:
                // Exit
                printf("Exiting program.\n");
                break;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }

    } while (option != 6);

    return 0;
}
```

---

4. Binary Search Tree (using linked list):

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

struct node {
    int data;
    struct node * left;
    struct node * right;
};

struct node* getNode(int data) {
    struct node* new = malloc(sizeof(struct node));
    new->data = data;
    new->left = NULL;
    new->right = NULL;
    return new;
}
```

```c
struct node* insertNode(struct node* root, int data) {
    if (root == NULL) {
        root = getNode(data);
        return root;
    }

    struct node* iter = root;

    while (iter) {
        if (data < iter->data) {
            if (iter->left) {
                iter = iter->left;
            }
            else {
                iter->left = getNode(data);
                break;
            }
        }
        else {
            if (iter->right) {
                iter = iter->right;
            }
            else {
                iter->right = getNode(data);
                break;
            }
        }
    }
    return root;
}

struct node* newTree() {
    int num, data = 0;
    printf("How many elemnets do you want in your tree? ");
    scanf("%d", &num);

    struct node* tree = NULL;
    for (int i = 0; i < num; i++) {
        printf("Enter the value: ");
        scanf("%d", &data);
        tree = insertNode(tree, data);
    }
    return tree;
}

void printBST(struct node* root) {
    if (root != NULL) {
        printBST(root->left);
        printf("%d ", root->data);
        printBST(root->right);
    }
}
```

```c
void printTree(struct node* root, int space) {
    int i;

    // Base case: if the node is null, return
    if (root == NULL) return;

    // Increase the space count for each level
    space += 10;

    // Recursively print the right subtree (on top)
    printTree(root->right, space);

    // Print the current node
    printf("\n");
    for (i = 2; i < space; i++) {
        printf(" ");
    }
    printf("%d\n", root->data);

    // Recursively print the left subtree (on bottom)
    printTree(root->left, space);
}

struct node* inorderSuccessor(struct node* node)
{
    struct node* curr = node -> right;

    while(curr -> left) {
        curr = curr -> left;
    }
    return curr;
}

struct node* findParent(struct node* node, int val, int* child) {
    struct node* parent = NULL;
    while (node) {
        if (node->data == val) return parent;
        parent = node;
        if (val < node->data) {
            *child = 2;
            node = node->left;
        }
        else {
            *child = 1;
            node = node->right;
        }
    }
    return parent;
}

struct node* deleteNode(struct node* root, int data) {
    int child = 0;
    struct node* parent = findParent(root, data, &child);
```

```
    struct node* TBDel = NULL;
    struct node* tmp = NULL;
    if (parent && child == 2) TBDel = parent->left;
    else if (parent && child == 1) TBDel = parent->right;
    else if (parent == NULL) TBDel = root;

    // case 1 left no right no
    if (!(TBDel->left) && !(TBDel->right)) {
        if (parent == NULL) {
            return NULL;
        }
        if (child == 2) {
           tmp = parent->left;
           parent->left = NULL;
        }
        else if (child == 1) {
            tmp = parent->right;
            parent->right = NULL;
        }
    }

    // case 2.1 left yes right no
    else if ((TBDel->left) && !(TBDel->right)) {
        if (parent == NULL) {
            return TBDel->left;
        }
        if (child == 2) {
            tmp = parent->left;
            parent->left = parent->left->left;
        }
        else if (child == 1) {
            tmp = parent->right;
            parent->right = parent->right->left;
        }
    }

    // case 2.2 left no right yes
    else if (!(TBDel->left) && (TBDel->right)) {
        if (parent == NULL) {
            return TBDel->right;
        }
        if (child == 2) {
            tmp = parent->left;
            parent->left = parent->left->right;
        }
        else if (child == 1) {
            tmp = parent->right;
            parent->right = parent->right->right;
        }
    }

    // case 3 left yes right yes
    else if ((TBDel->left) && (TBDel->right)) {
```

```
        struct node* IOSucc = inorderSuccessor(TBDel);
        int tmp = IOSucc->data;
        root = deleteNode(root, tmp);
        TBDel->data = tmp;
    }

    if (tmp) free(tmp);
    return root;
}

void displayMenu() {
    printf("\nMenu:\n");
    printf("1. New Tree\n");
    printf("2. Add New Element\n");
    printf("3. Delete Element\n");
    printf("4. Print BST\n");
    printf("5. Menu\n");
    printf("6. Exit\n");
}

int main() {
    struct node* tree = NULL;

    int choice;
    int element;

    displayMenu();
    do {
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                tree = newTree();
                printf("New tree created.\n");
                break;

            case 2:
                printf("Enter the element to add: ");
                scanf("%d", &element);
                tree = insertNode(tree, element);
                printf("Element %d added to the tree.\n", element);
                break;

            case 3:
                printf("Enter the element to delete: ");
                scanf("%d", &element);
                tree = deleteNode(tree, element);
                printf("Element %d deleted from the tree.\n", element);
                break;

            case 4:
                printf("BST in-order traversal:\n");
```

```
                printBST(tree);
                printTree(tree, 0);
                break;

            case 5:
                displayMenu();
                break;

            case 6:
                printf("Exiting the program.\n");
                break;

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

    } while (choice != 6);

    return 0;
}
```

---

5. Leetcode:

116. Populating Next Right Pointers in Each Node

```
class Solution(object):
    def connect(self, root):
        """
        :type root: Node
        :rtype: Node
        """

        queue = [root]
        levels = [root]

        while queue:

            curr = queue.pop(0)

            if curr:
                if curr.left:
                    queue.append(curr.left)
                    levels.append(curr.left)
                if curr.right:
                    queue.append(curr.right)
                    levels.append(curr.right)

        n = 0
        while levels:
            level = levels[0:2**n]
```

```
            del levels[0:2**n]
            n+=1

            for i in range(len(level)):
                if level[i]:
                    if i == 0:
                        prev = level[i]
                    else:
                        prev.next = level[i]
                        prev = level[i]

        return root
```

AVL and Expression Trees:

1. AVL Tree:

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

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    int height;
};

int height(struct Node* node) {
    if (node == NULL)
        return 0;
    return node->height;
}

int getBalance(struct Node* node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

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

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

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

    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) :
height(y->right));
    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) :
height(x->right));

    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 = 1 + (height(x->left) > height(x->right) ? height(x->left) :
height(x->right));
    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) :
height(y->right));

    return y;
}

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

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

    node->height = 1 + (height(node->left) > height(node->right) ? height(node-
>left) : height(node->right));

    int balance = getBalance(node);

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

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

    if (balance > 1 && key > node->left->data) {
```

```
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

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

    return node;
}

struct Node* InOrderSuccessor(struct Node* node) {
    struct Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL)
        return root;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        if (root->left == NULL || root->right == NULL) {
            struct Node* temp = (root->left) ? root->left : root->right;

            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = *temp;

            free(temp);
        } else {
            struct Node* temp = InOrderSuccessor(root->right);

            root->data = temp->data;

            root->right = deleteNode(root->right, temp->data);
        }
    }

    if (root == NULL)
        return root;

    root->height = 1 + (height(root->left) > height(root->right) ? height(root-
>left) : height(root->right));
```

```
    int balance = getBalance(root);

    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

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

void freeTree(struct Node* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}

int main() {
    struct Node* root = NULL;

    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    printf("In-order traversal of the AVL tree: ");
    inOrder(root);
    printf("\n");

    root = deleteNode(root, 30);
```

```
    printf("In-order traversal after deleting 30: ");
    inOrder(root);
    printf("\n");

    freeTree(root);

    return 0;
}
```

---

2. Expression Tree:

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

struct TreeNode {
    char data;
    struct TreeNode* left;
    struct TreeNode* right;
};

struct TreeNode* createNode(char data) {
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}

int isOperand(char ch) {
    return (ch >= '0' && ch <= '9');
}

struct TreeNode* constructExpressionTree(char postfix[]) {
    struct TreeNode* stack[100];
    int top = -1;

    for (int i = 0; postfix[i] != '\0'; ++i) {
        struct TreeNode* newNode = createNode(postfix[i]);

        if (isOperand(postfix[i])) {
            stack[++top] = newNode;
        } else {
            newNode->right = stack[top--];
            newNode->left = stack[top--];
            stack[++top] = newNode;
        }
    }

    return stack[top];
}
```

```c
void inOrderTraversal(struct TreeNode* root) {
    if (root != NULL) {
        if (isOperand(root->data))
            printf("%c ", root->data);
        else {
            printf("( ");
            inOrderTraversal(root->left);
            printf("%c ", root->data);
            inOrderTraversal(root->right);
            printf(") ");
        }
    }
}

int evaluateExpressionTree(struct TreeNode* root) {
    if (root == NULL)
        return 0;

    if (isOperand(root->data))
        return root->data - '0';

    int leftValue = evaluateExpressionTree(root->left);
    int rightValue = evaluateExpressionTree(root->right);

    switch (root->data) {
        case '+':
            return leftValue + rightValue;
        case '-':
            return leftValue - rightValue;
        case '*':
            return leftValue * rightValue;
        case '/':
            return leftValue / rightValue;
        default:
            return 0;
    }
}

int main() {
    char postfix[] = "34*5+";

    struct TreeNode* root = constructExpressionTree(postfix);

    printf("In-order traversal of the expression tree: ");
    inOrderTraversal(root);
    printf("\n");

    int result = evaluateExpressionTree(root);
    printf("Result of the expression: %d\n", result);

    return 0;
}
```

Heaps:

1. Min Heap:

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

struct Heap {
    int* arr;
    int size;
};

void propagateUp(struct Heap* mHeap, int i) {
    if (i == 0) return;
    int tmp = 0;
    int* heap = mHeap->arr;
    while ((*(heap + (i-1)/2) > *(heap + i)) && (i != 0)) {
        tmp = *(heap + (i-1)/2);
        *(heap + (i-1)/2) = *(heap + i);
        *(heap + i) = tmp;

        i = (i-1)/2;
    }
    return;
}

struct Heap* createHeap() {
    struct Heap* heap = malloc(sizeof(struct Heap));
    printf("Enter the number of elements:");
    scanf("%d", &(heap -> size));
    heap->arr = malloc(heap -> size * sizeof(int));

    int val;
    for (int i = 0; i < heap->size; i++) {
        printf("Enter a number:");
        scanf("%d", &val);
        heap->arr[i] = val;
        propagateUp(heap, i);
    }
    return heap;
}

int extractMin(struct Heap* h) {
    return h->arr[0];
}

void heapify(struct Heap* h, int i) {
```

```
    // curr is out of bounds
    if (i > h->size) return;

    // children are out of bounds
    if (((2*i+1) >= h->size) || ((2*i+2) >= h->size)) return;

    // both children are greater than curr
    if ((*(h->arr + 2*i + 1) > *(h->arr + i)) && (*(h->arr + 2*i + 2) > *(h-
>arr + i) )) return;

    // parent is greater than both children
    if ((*(h->arr + 2*i + 1) < *(h->arr + i)) && (*(h->arr + 2*i + 2) < *(h-
>arr + i) )) {

        // swapping curr with the smaller child
        if (*(h->arr + 2*i + 1) <  *(h->arr + 2*i + 2)) {
            int tmp = *(h->arr + 2*i + 1);
            *(h->arr + 2*i + 1) = *(h->arr + i);
            *(h->arr + i) = tmp;
            i = 2*i + 1;
        }
        else {
            int tmp = *(h->arr + 2*i + 2);
            *(h->arr + 2*i + 2) = *(h->arr + i);
            *(h->arr + i) = tmp;
            i = 2*i + 2;
        }
        heapify(h, i);
        return;
    }

    // left child is smaller, right is greater
    if (*(h->arr + 2*i + 1) <  *(h->arr + i)) {
        int tmp = *(h->arr + 2*i + 1);
        *(h->arr + 2*i + 1) = *(h->arr + i);
        *(h->arr + i) = tmp;
        i = 2*i + 1;
        heapify(h, i);
        return;
    }

    // left child is greater, right is smaller
    if (*(h->arr + 2*i + 2) <  *(h->arr + i)) {
        int tmp = *(h->arr + 2*i + 2);
        *(h->arr + 2*i + 2) = *(h->arr + i);
        *(h->arr + i) = tmp;
        i = 2*i + 2;
        heapify(h, i);
        return;
    }
}

void deleteMin(struct Heap* h) {
```

```
    h->arr[0] = h->arr[--h->size];
    for (int i = 0; i < h->size; i++) printf("%d, ", h->arr[i]);
    printf("\n");
    heapify(h, 0);
    return;
}

int main(void) {
    struct Heap* minHeap = createHeap();
    for (int i = 0; i < minHeap->size; i++) printf("%d, ", minHeap->arr[i]);
    printf("\nRoot: %d\n", extractMin(minHeap));
    deleteMin(minHeap);
    for (int i = 0; i < minHeap->size; i++) printf("%d, ", minHeap->arr[i]);
    return 0;
}
```

---

2. Max Heap:

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

struct Heap {
    int* arr;
    int size;
};

void propagateUp(struct Heap* mHeap, int i) {
    if (i == 0)
        return;

    int tmp = 0;
    int* heap = mHeap->arr;

    while ((*(heap + (i - 1) / 2) < *(heap + i)) && (i != 0)) {
        tmp = *(heap + (i - 1) / 2);
        *(heap + (i - 1) / 2) = *(heap + i);
        *(heap + i) = tmp;

        i = (i - 1) / 2;
    }
    return;
}

struct Heap* createHeap() {
    struct Heap* heap = malloc(sizeof(struct Heap));
    printf("Enter the number of elements:");
    scanf("%d", &(heap->size));
    heap->arr = malloc(heap->size * sizeof(int));

    int val;
```

```
    for (int i = 0; i < heap->size; i++) {
        printf("Enter a number:");
        scanf("%d", &val);
        heap->arr[i] = val;
        propagateUp(heap, i);
    }
    return heap;
}

int extractMax(struct Heap* h) {
    return h->arr[0];
}

void heapify(struct Heap* h, int i) {
    // curr is out of bounds
    if (i >= h->size)
        return;

    // children are out of bounds
    if (((2 * i + 1) >= h->size) || ((2 * i + 2) >= h->size))
        return;

    // both children are smaller than curr
    if ((*(h->arr + 2 * i + 1) < *(h->arr + i)) && (*(h->arr + 2 * i + 2) < *
(h->arr + i)))
        return;

    // parent is smaller than both children
    if ((*(h->arr + 2 * i + 1) > *(h->arr + i)) && (*(h->arr + 2 * i + 2) > *
(h->arr + i))) {

        // swapping curr with the larger child
        if (*(h->arr + 2 * i + 1) > *(h->arr + 2 * i + 2)) {
            int tmp = *(h->arr + 2 * i + 1);
            *(h->arr + 2 * i + 1) = *(h->arr + i);
            *(h->arr + i) = tmp;
            i = 2 * i + 1;
        } else {
            int tmp = *(h->arr + 2 * i + 2);
            *(h->arr + 2 * i + 2) = *(h->arr + i);
            *(h->arr + i) = tmp;
            i = 2 * i + 2;
        }
        heapify(h, i);
        return;
    }

    // left child is larger, right is smaller
    if (*(h->arr + 2 * i + 1) > *(h->arr + i)) {
        int tmp = *(h->arr + 2 * i + 1);
        *(h->arr + 2 * i + 1) = *(h->arr + i);
        *(h->arr + i) = tmp;
        i = 2 * i + 1;
```

```
        heapify(h, i);
        return;
    }

    // left child is smaller, right is larger
    if (*(h->arr + 2 * i + 2) > *(h->arr + i)) {
        int tmp = *(h->arr + 2 * i + 2);
        *(h->arr + 2 * i + 2) = *(h->arr + i);
        *(h->arr + i) = tmp;
        i = 2 * i + 2;
        heapify(h, i);
        return;
    }
}

void deleteMax(struct Heap* h) {
    h->arr[0] = h->arr[--h->size];
    for (int i = 0; i < h->size; i++)
        printf("%d, ", h->arr[i]);
    printf("\n");
    heapify(h, 0);
    return;
}

int main(void) {
    struct Heap* maxHeap = createHeap();
    for (int i = 0; i < maxHeap->size; i++)
        printf("%d, ", maxHeap->arr[i]);
    printf("\nRoot: %d\n", extractMax(maxHeap));
    deleteMax(maxHeap);
    for (int i = 0; i < maxHeap->size; i++)
        printf("%d, ", maxHeap->arr[i]);
    return 0;
}
```

---

3. Heap sort:

```
#include <stdio.h>

void heapify(int arr[], int n, int i, int isMaxHeap) {
    int largestOrSmallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (isMaxHeap) {
        // For max-heap, compare with left and right children
        if (left < n && arr[left] > arr[largestOrSmallest])
            largestOrSmallest = left;

        if (right < n && arr[right] > arr[largestOrSmallest])
```

```
            largestOrSmallest = right;
        } else {
            // For min-heap, compare with left and right children
            if (left < n && arr[left] < arr[largestOrSmallest])
                largestOrSmallest = left;

            if (right < n && arr[right] < arr[largestOrSmallest])
                largestOrSmallest = right;
        }

        // If the largest/smallest is not the root, swap them and recursively
heapify the affected sub-tree
        if (largestOrSmallest != i) {
            int temp = arr[i];
            arr[i] = arr[largestOrSmallest];
            arr[largestOrSmallest] = temp;

            // Recursively heapify the affected sub-tree
            heapify(arr, n, largestOrSmallest, isMaxHeap);
        }
    }

    void heapSort(int arr[], int n, int isMaxHeap) {
        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i, isMaxHeap);

        // One by one extract an element from the heap
        for (int i = n - 1; i >= 0; i--) {
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // Call heapify on the reduced heap
            heapify(arr, i, 0, isMaxHeap);
        }
    }

    void printArray(int arr[], int size) {
        for (int i = 0; i < size; i++)
            printf("%d ", arr[i]);
        printf("\n");
    }

    int main() {
        int arr[] = {12, 11, 13, 5, 6, 7};
        int n = sizeof(arr) / sizeof(arr[0]);

        printf("Original array: \n");
        printArray(arr, n);

        // Sort as Max Heap (Ascending Order)
```

```
    heapSort(arr, n, 1);
    printf("Sorted array (Max Heap): \n");
    printArray(arr, n);

    // Sort as Min Heap (Descending Order)
    heapSort(arr, n, 0);
    printf("Sorted array (Min Heap): \n");
    printArray(arr, n);

    return 0;
}
```

Graphs:

1. Adjacency Matrix:

```
int** initializeGraph(int numVertices) {
    int** graph = malloc(sizeof(int*) * numVertices);
    for (int i = 0; i < numVertices; i++) {
        graph[i] = malloc(sizeof(int) * numVertices);
        for (int j = 0; j < numVertices; j++) {
            graph[i][j] = 0;
        }
    }

    return graph;
}

void addEdge(int** graph, int startVertex, int endVertex, int weight) {
    graph[startVertex][endVertex] = weight;
}

void printGraph(int** graph, int numVertices) {
    printf("Adjacency Matrix:\n");
    for (int i = 0; i < numVertices; ++i) {
        for (int j = 0; j < numVertices; ++j) {
            printf("%d\t", graph[i][j]);
        }
        printf("\n");
    }
}
```

2. Adjacency List:

```c
struct Node {
    int vertex;
    struct Node* next;
};

struct Graph {
    int numVertices;
    struct Node* adjacencyList[MAX_VERTICES];
};

struct Graph* initializeGraph(int numVertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = numVertices;

    // Initialize the adjacency list for each vertex
    for (int i = 0; i < numVertices; ++i) {
        graph->adjacencyList[i] = NULL;
    }

    return graph;
}

void addEdge(struct Graph* graph, int startVertex, int endVertex) {
    // Create a new node for the end vertex
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = endVertex;
    newNode->next = graph->adjacencyList[startVertex];

    // Update the adjacency list for the start vertex
    graph->adjacencyList[startVertex] = newNode;
}

void printGraph(struct Graph* graph) {
    printf("Adjacency List:\n");
    for (int i = 0; i < graph->numVertices; ++i) {
        struct Node* current = graph->adjacencyList[i];
        printf("Vertex %d: ", i);
        while (current != NULL) {
            printf("%d -> ", current->vertex);
            current = current->next;
        }
        printf("NULL\n");
    }
}
```

---

3. Breadth First Search:

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

```c
#define MAX_VERTICES 50

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

struct Graph {
    int numVertices;
    struct Node* adjacencyList[MAX_VERTICES];
};

struct Graph* initializeGraph(int numVertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = numVertices;

    // Initialize the adjacency list for each vertex
    for (int i = 0; i < numVertices; ++i) {
        graph->adjacencyList[i] = NULL;
    }

    return graph;
}

void addEdge(struct Graph* graph, int startVertex, int endVertex) {
    // Create a new node for the end vertex
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = endVertex;
    newNode->next = graph->adjacencyList[startVertex];

    // Update the adjacency list for the start vertex
    graph->adjacencyList[startVertex] = newNode;
}

void printGraph(struct Graph* graph) {
    printf("Adjacency List:\n");
    for (int i = 0; i < graph->numVertices; ++i) {
        struct Node* current = graph->adjacencyList[i];
        printf("Vertex %d: ", i);
        while (current != NULL) {
            printf("%d -> ", current->vertex);
            current = current->next;
        }
        printf("NULL\n");
    }
}

struct Queue {
    int items[MAX_VERTICES];
    int front;
    int rear;
};
```

```c
struct Queue* createQueue() {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->front = -1;
    queue->rear = -1;
    return queue;
}

int isEmpty(struct Queue* queue) {
    if (queue->rear == -1)
        return 1;
    else
        return 0;
}

void enqueue(struct Queue* queue, int value) {
    if (queue->rear == MAX_VERTICES - 1)
        printf("\nQueue is Full!!");
    else {
        if (queue->front == -1)
            queue->front = 0;
        queue->rear++;
        queue->items[queue->rear] = value;
    }
}

int dequeue(struct Queue* queue) {
    int item;
    if (isEmpty(queue)) {
        printf("Queue is empty");
        item = -1;
    } else {
        item = queue->items[queue->front];
        queue->front++;
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
    }
    return item;
}

void deleteQueue(struct Queue* queue) {
    if (queue) {
        if (queue->items)
            free(queue->items);
        free(queue);
    }
}

void bfs(struct Graph* graph, int startVertex) {
    struct Queue* queue = createQueue();

    int* visited = malloc(graph->numVertices * sizeof(int));
```

```
    for (int i = 0; i < graph->numVertices; i++)
        visited[i] = 0;

    visited[startVertex] = 1; // just to keep track of visited nodea
    enqueue(queue, startVertex);

    printf("BFS: ");
    while (!isEmpty(queue)) {
        // Print the visited node
        int currentVertex = dequeue(queue);
        printf("%d", currentVertex);

        // Find the adjacent vertices and add them to the queue
        struct Node* temp = graph->adjacencyList[currentVertex];

        while (temp) {
            int adjVertex = temp->vertex;

            if (visited[adjVertex] == 0) {
                visited[adjVertex] = 1;
                enqueue(queue, adjVertex);
            }
            temp = temp->next;
        }
    }
}
```

---

4. Depth First Search:

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

#define MAX_VERTICES 50

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

struct Graph {
    int numVertices;
    struct Node* adjacencyList[MAX_VERTICES];
};

struct Graph* initializeGraph(int numVertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = numVertices;

    // Initialize the adjacency list for each vertex
    for (int i = 0; i < numVertices; ++i) {
```

```
        graph->adjacencyList[i] = NULL;
    }

    return graph;
}

void addEdge(struct Graph* graph, int startVertex, int endVertex) {
    // Create a new node for the end vertex
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = endVertex;
    newNode->next = graph->adjacencyList[startVertex];

    // Update the adjacency list for the start vertex
    graph->adjacencyList[startVertex] = newNode;
}


void printGraph(struct Graph* graph) {
    printf("Adjacency List:\n");
    for (int i = 0; i < graph->numVertices; ++i) {
        struct Node* current = graph->adjacencyList[i];
        printf("Vertex %d: ", i);
        while (current != NULL) {
            printf("%d -> ", current->vertex);
            current = current->next;
        }
        printf("NULL\n");
    }
}

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

struct Stack* createStack() {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->top = -1;
    return stack;
}

int isEmpty(struct Stack* stack) {
    if (stack->top == -1)
        return 1;
    else
        return 0;
}

void push(struct Stack* stack, int value) {
    stack->items[++stack->top] = value;
}

int pop(struct Stack* stack) {
```

```
    return stack->items[stack->top--];
}

void deleteStack(struct Stack* stack) {
    if (stack) {
        free(stack);
    }
}

void dfs(struct Graph* graph, int startVertex) {
    struct Stack* stack = createStack();

    int* visited = malloc(graph->numVertices * sizeof(int));
    for (int i = 0; i < graph->numVertices; i++)
        visited[i] = 0;

    visited[startVertex] = 1;
    push(stack, startVertex);

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

        struct Node* temp = graph->adjacencyList[currentVertex];

        while (temp) {
            int adjVertex = temp->vertex;

            if (visited[adjVertex] == 0) {
                visited[adjVertex] = 1;
                push(stack, adjVertex);
            }
            temp = temp->next;
        }
    }
}
```

---

5. Topological Sort:

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

#define MAX_VERTICES 100

// Node structure for the adjacency list
struct Node {
    int vertex;
    struct Node* next;
};
```

```c
// Graph structure containing an array of adjacency lists
struct Graph {
    int numVertices;
    struct Node* adjacencyList[MAX_VERTICES];
    int inDegree[MAX_VERTICES];
};

// Function to initialize the graph
struct Graph* initializeGraph(int numVertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = numVertices;

    // Initialize the adjacency list for each vertex
    for (int i = 0; i < numVertices; ++i) {
        graph->adjacencyList[i] = NULL;
        graph->inDegree[i] = 0;
    }

    return graph;
}

// Function to add a directed edge to the graph
void addDirectedEdge(struct Graph* graph, int startVertex, int endVertex) {
    // Create a new node for the end vertex
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = endVertex;
    newNode->next = graph->adjacencyList[startVertex];

    // Update the adjacency list for the start vertex
    graph->adjacencyList[startVertex] = newNode;

    // Increment in-degree for the end vertex
    graph->inDegree[endVertex]++;
}

// Function to perform topological sorting using a queue
void topologicalSort(struct Graph* graph) {
    int queue[MAX_VERTICES];
    int front = 0, rear = -1;

    // Enqueue vertices with in-degree 0
    for (int i = 0; i < graph->numVertices; ++i) {
        if (graph->inDegree[i] == 0) {
            queue[++rear] = i;
        }
    }

    // Perform topological sorting
    while (front <= rear) {
        int currentVertex = queue[front++];
        printf("%d ", currentVertex);
```

```
        // Decrease in-degree of adjacent vertices and enqueue if in-degree
becomes 0
        struct Node* temp = graph->adjacencyList[currentVertex];
        while (temp != NULL) {
            int adjacentVertex = temp->vertex;
            graph->inDegree[adjacentVertex]--;
            if (graph->inDegree[adjacentVertex] == 0) {
                queue[++rear] = adjacentVertex;
            }
            temp = temp->next;
        }
    }
}
```
