1. **Write a Program to create a Binary Tree and perform following nonrecursive operations on it. a. Preorder Traversal, b. Postorder Traversal, c. Count total no. of nodes, d. Display height of a tree.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

    int data;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct TreeNode\* createNode(int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

void preorderTraversal(struct TreeNode\* current) {

    if (current != NULL) {

        printf("%d ", current->data);

        preorderTraversal(current->left);

        preorderTraversal(current->right);

    }

}

void postorderTraversal(struct TreeNode\* current) {

    if (current != NULL) {

        postorderTraversal(current->left);

        postorderTraversal(current->right);

        printf("%d ", current->data);

    }

}

int countNodes(struct TreeNode\* root) {

    if (root == NULL)

        return 0;

    return 1 + countNodes(root->left) + countNodes(root->right);

}

int max(int a, int b) {

    return (a > b) ? a : b;

}

int height(struct TreeNode\* root) {

    if (root == NULL)

        return 0;

    int leftHeight = height(root->left);

    int rightHeight = height(root->right);

    return 1 + max(leftHeight, rightHeight);

}

int main() {

    struct TreeNode\* root = NULL;

    char choice;

    do {

        int data;

        struct TreeNode\* newNode;

        printf("Enter the data for the new node: ");

        scanf("%d", &data);

        newNode = createNode(data);

        if (root == NULL) {

            root = newNode;

        } else {

            struct TreeNode\* current = root;

            while (1) {

                printf("Do you want to insert '%d' to the left or right of '%d' (l/r): ", data, current->data);

                scanf(" %c", &choice);

                if (choice == 'l' || choice == 'L') {

                    if (current->left == NULL) {

                        current->left = newNode;

                        break;

                    } else {

                        current = current->left;

                    }

                } else if (choice == 'r' || choice == 'R') {

                    if (current->right == NULL) {

                        current->right = newNode;

                        break;

                    } else {

                        current = current->right;

                    }

                } else {

                    printf("Invalid choice! Please enter 'L' or 'R'.\n");

                }

            }

        }

        printf("Do you want to insert another node? (Y/N): ");

        scanf(" %c", &choice);

    } while (choice == 'Y' || choice == 'y');

    printf("\nPreorder Traversal: ");

    preorderTraversal(root);

    printf("\nPostorder Traversal: ");

    postorderTraversal(root);

    printf("\nTotal number of nodes: %d\n", countNodes(root));

    printf("Height of the tree: %d\n", height(root));

    return 0;

}

1. **Write a Program to create a Binary Tree and perform following nonrecursive operations on it. a. inorder Traversal; b. Count no. of nodes on longest path; c. display tree levelwise; d. Display height of a tree.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

    int data;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct TreeNode\* createNode(int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

void inorderTraversal(struct TreeNode\* root) {

    struct TreeNode\* stack[100];

    int top = -1;

    struct TreeNode\* current = root;

    while (current != NULL || top != -1) {

        while (current != NULL) {

            stack[++top] = current;

            current = current->left;

        }

        current = stack[top--];

        printf("%d ", current->data);

        current = current->right;

    }

}

int countNodesLongestPath(struct TreeNode\* root) {

    if (root == NULL)

        return 0;

    int leftHeight = height(root->left);

    int rightHeight = height(root->right);

    return 1 + leftHeight + rightHeight;

}

void displayLevelWise(struct TreeNode\* root) {

    if (root == NULL)

        return;

    struct TreeNode\* queue[100];

    int front = -1, rear = -1;

    queue[++rear] = root;

    while (front != rear) {

        struct TreeNode\* current = queue[++front];

        printf("%d ", current->data);

        if (current->left != NULL)

            queue[++rear] = current->left;

        if (current->right != NULL)

            queue[++rear] = current->right;

    }

}

int max(int a, int b) {

    return (a > b) ? a : b;

}

int height(struct TreeNode\* root) {

    if (root == NULL)

        return 0;

    int leftHeight = height(root->left);

    int rightHeight = height(root->right);

    return 1 + max(leftHeight, rightHeight);

}

int main() {

    struct TreeNode\* root = NULL;

    char choice;

    do {

        int data;

        struct TreeNode\* newNode;

        printf("Enter the data for the new node: ");

        scanf("%d", &data);

        newNode = createNode(data);

        if (root == NULL) {

            root = newNode;

        } else {

            struct TreeNode\* current = root;

            while (1) {

                printf("Do you want to insert '%d' to the left or right of '%d' (l/r): ", data, current->data);

                scanf(" %c", &choice);

                if (choice == 'l' || choice == 'L') {

                    if (current->left == NULL) {

                        current->left = newNode;

                        break;

                    } else {

                        current = current->left;

                    }

                } else if (choice == 'r' || choice == 'R') {

                    if (current->right == NULL) {

                        current->right = newNode;

                        break;

                    } else {

                        current = current->right;

                    }

                } else {

                    printf("Invalid choice! Please enter 'L' or 'R'.\n");

                }

            }

        }

        printf("Do you want to insert another node? (Y/N): ");

        scanf(" %c", &choice);

    } while (choice == 'Y' || choice == 'y');

    printf("\nInorder Traversal: ");

    inorderTraversal(root);

    printf("\nNumber of nodes on longest path: %d\n", countNodesLongestPath(root));

    printf("Tree levelwise: ");

    displayLevelWise(root);

    printf("\nHeight of the tree: %d\n", height(root));

    return 0;

}

1. **Write a Program to create a Binary Search Tree holding numeric keys and perform following nonrecursive operations on it. a. Levelwise display, b. Mirror image, c. Display height of a tree, d. Find**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

struct TreeNode {

    int data;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct TreeNode\* createNode(int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

void insertNode(struct TreeNode\*\* root, int data) {

    struct TreeNode\* newNode = createNode(data);

    if (\*root == NULL) {

        \*root = newNode;

    } else {

        struct TreeNode\* current = \*root;

        while (1) {

            if (data < current->data) {

                if (current->left == NULL) {

                    current->left = newNode;

                    break;

                } else {

                    current = current->left;

                }

            } else if (data > current->data) {

                if (current->right == NULL) {

                    current->right = newNode;

                    break;

                } else {

                    current = current->right;

                }

            } else {

                printf("Duplicate keys are not allowed\n");

                free(newNode);

                break;

            }

        }

    }

}

void levelwiseDisplay(struct TreeNode\* root) {

    if (root == NULL) {

        printf("Tree is empty\n");

        return;

    }

    struct TreeNode\* queue[100];

    int front = -1, rear = -1;

    queue[++rear] = root;

    while (front != rear) {

        int nodesInLevel = rear - front;

        while (nodesInLevel > 0) {

            struct TreeNode\* current = queue[++front];

            printf("%d ", current->data);

            if (current->left != NULL)

                queue[++rear] = current->left;

            if (current->right != NULL)

                queue[++rear] = current->right;

            nodesInLevel--;

        }

        printf("\n");

    }

}

void mirrorImage(struct TreeNode\* root) {

    if (root == NULL)

        return;

    struct TreeNode\* temp = root->left;

    root->left = root->right;

    root->right = temp;

    mirrorImage(root->left);

    mirrorImage(root->right);

}

int height(struct TreeNode\* root) {

    if (root == NULL)

        return 0;

    int leftHeight = height(root->left);

    int rightHeight = height(root->right);

    return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);

}

struct TreeNode\* search(struct TreeNode\* root, int key) {

    while (root != NULL && root->data != key) {

        if (key < root->data)

            root = root->left;

        else

            root = root->right;

    }

    return root;

}

int main() {

    struct TreeNode\* root = NULL;

    int choice, data, key;

    do {

        printf("\n1. Insert\n2. Levelwise Display\n3. Mirror Image\n4. Display Height\n5. Search\n6. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter data to insert: ");

                scanf("%d", &data);

                insertNode(&root, data);

                break;

            case 2:

                printf("Levelwise Display:\n");

                levelwiseDisplay(root);

                break;

            case 3:

                printf("Mirror Image:\n");

                mirrorImage(root);

                levelwiseDisplay(root);

                break;

            case 4:

                printf("Height of the tree: %d\n", height(root));

                break;

            case 5:

                printf("Enter key to search: ");

                scanf("%d", &key);

                if (search(root, key) != NULL)

                    printf("%d found in the tree.\n", key);

                else

                    printf("%d not found in the tree.\n", key);

                break;

            case 6:

                printf("Exiting...\n");

                break;

            default:

                printf("Invalid choice!\n");

        }

    } while (choice != 6);

    return 0;

}

1. **Write a program to illustrate operations on a BST holding numeric keys. The menu must include: • Insert • Delete • Find • display in Inorder way**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

    int key;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct TreeNode\* createNode(int key) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->key = key;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int key) {

    if (root == NULL)

        return createNode(key);

    if (key < root->key)

        root->left = insertNode(root->left, key);

    else if (key > root->key)

        root->right = insertNode(root->right, key);

    else

        printf("Key already exists in the tree.\n");

    return root;

}

struct TreeNode\* minValueNode(struct TreeNode\* node) {

    struct TreeNode\* current = node;

    while (current && current->left != NULL)

        current = current->left;

    return current;

}

struct TreeNode\* deleteNode(struct TreeNode\* root, int key) {

    if (root == NULL)

        return root;

    if (key < root->key)

        root->left = deleteNode(root->left, key);

    else if (key > root->key)

        root->right = deleteNode(root->right, key);

    else {

        if (root->left == NULL) {

            struct TreeNode\* temp = root->right;

            free(root);

            return temp;

        } else if (root->right == NULL) {

            struct TreeNode\* temp = root->left;

            free(root);

            return temp;

        }

        struct TreeNode\* temp = minValueNode(root->right);

        root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }

    return root;

}

struct TreeNode\* searchNode(struct TreeNode\* root, int key) {

    if (root == NULL || root->key == key)

        return root;

    if (key < root->key)

        return searchNode(root->left, key);

    return searchNode(root->right, key);

}

void inorderTraversal(struct TreeNode\* root) {

    if (root == NULL)

        return;

    inorderTraversal(root->left);

    printf("%d ", root->key);

    inorderTraversal(root->right);

}

void displayMenu() {

    printf("\nMenu:\n");

    printf("1. Insert\n");

    printf("2. Delete\n");

    printf("3. Find\n");

    printf("4. Display Inorder\n");

    printf("5. Exit\n");

}

int main() {

    struct TreeNode\* root = NULL;

    int choice, key;

    struct TreeNode\* found;

    do {

        displayMenu();

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter the key to insert: ");

                scanf("%d", &key);

                root = insertNode(root, key);

                break;

            case 2:

                printf("Enter the key to delete: ");

                scanf("%d", &key);

                root = deleteNode(root, key);

                break;

            case 3:

                printf("Enter the key to find: ");

                scanf("%d", &key);

                found = searchNode(root, key);

                if (found != NULL)

                    printf("Key %d found in the tree.\n", key);

                else

                    printf("Key %d not found in the tree.\n", key);

                break;

            case 4:

                printf("Inorder Traversal: ");

                inorderTraversal(root);

                printf("\n");

                break;

            case 5:

                printf("Exiting the program.\n");

                break;

            default:

                printf("Invalid choice! Please enter a valid option.\n");

        }

    } while (choice != 5);

    return 0;

}

1. **Write a program to illustrate operations on a BST holding numeric keys. The menu must include: • Insert • Mirror Image • Find • Post order (nonrecursive)**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

    int key;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct TreeNode\* createNode(int key) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->key = key;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int key) {

    if (root == NULL)

        return createNode(key);

    if (key < root->key)

        root->left = insertNode(root->left, key);

    else if (key > root->key)

        root->right = insertNode(root->right, key);

    else

        printf("Key already exists in the tree.\n");

    return root;

}

struct TreeNode\* minValueNode(struct TreeNode\* node) {

    struct TreeNode\* current = node;

    while (current && current->left != NULL)

        current = current->left;

    return current;

}

struct TreeNode\* deleteNode(struct TreeNode\* root, int key) {

    if (root == NULL)

        return root;

    if (key < root->key)

        root->left = deleteNode(root->left, key);

    else if (key > root->key)

        root->right = deleteNode(root->right, key);

    else {

        if (root->left == NULL) {

            struct TreeNode\* temp = root->right;

            free(root);

            return temp;

        } else if (root->right == NULL) {

            struct TreeNode\* temp = root->left;

            free(root);

            return temp;

        }

        struct TreeNode\* temp = minValueNode(root->right);

        root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }

    return root;

}

struct TreeNode\* searchNode(struct TreeNode\* root, int key) {

    if (root == NULL || root->key == key)

        return root;

    if (key < root->key)

        return searchNode(root->left, key);

    return searchNode(root->right, key);

}

void inorderTraversal(struct TreeNode\* root) {

    if (root == NULL)

        return;

    inorderTraversal(root->left);

    printf("%d ", root->key);

    inorderTraversal(root->right);

}

void displayMenu() {

    printf("\nMenu:\n");

    printf("1. Insert\n");

    printf("2. Delete\n");

    printf("3. Find\n");

    printf("4. Display Inorder\n");

    printf("5. Exit\n");

}

int main() {

    struct TreeNode\* root = NULL;

    int choice, key;

    struct TreeNode\* found;

    do {

        displayMenu();

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter the key to insert: ");

                scanf("%d", &key);

                root = insertNode(root, key);

                break;

            case 2:

                printf("Enter the key to delete: ");

                scanf("%d", &key);

                root = deleteNode(root, key);

                break;

            case 3:

                printf("Enter the key to find: ");

                scanf("%d", &key);

                found = searchNode(root, key);

                if (found != NULL)

                    printf("Key %d found in the tree.\n", key);

                else

                    printf("Key %d not found in the tree.\n", key);

                break;

            case 4:

                printf("Inorder Traversal: ");

                inorderTraversal(root);

                printf("\n");

                break;

            case 5:

                printf("Exiting the program.\n");

                break;

            default:

                printf("Invalid choice! Please enter a valid option.\n");

        }

    } while (choice != 5);

    return 0;

}

1. **Write a Program to create a Binary Search Tree and perform following nonrecursive operations on it. a. Preorder Traversal b. Inorder Traversal c. Display Number of Leaf Nodes d. Mirror Image.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

    int data;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct Queue {

    struct TreeNode\* array[100];

    int front, rear;

};

struct Queue\* createQueue() {

    struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

    queue->front = queue->rear = -1;

    return queue;

}

int isEmpty(struct Queue\* queue) {

    return queue->front == -1;

}

void enqueue(struct Queue\* queue, struct TreeNode\* item) {

    if (queue->rear == 99) {

        printf("Queue is full. Enqueue operation failed.\n");

        return;

    }

    if (isEmpty(queue))

        queue->front = 0;

    queue->rear++;

    queue->array[queue->rear] = item;

}

struct TreeNode\* dequeue(struct Queue\* queue) {

    if (isEmpty(queue)) {

        printf("Queue is empty. Dequeue operation failed.\n");

        return NULL;

    }

    struct TreeNode\* item = queue->array[queue->front];

    if (queue->front == queue->rear)

        queue->front = queue->rear = -1;

    else

        queue->front++;

    return item;

}

struct TreeNode\* createNode(int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

void insert(struct TreeNode\*\* root, int data) {

    if (\*root == NULL) {

        \*root = createNode(data);

    } else {

        struct TreeNode\* current = \*root;

        struct TreeNode\* parent = NULL;

        while (current != NULL) {

            parent = current;

            if (data < current->data)

                current = current->left;

            else

                current = current->right;

        }

        if (data < parent->data)

            parent->left = createNode(data);

        else

            parent->right = createNode(data);

    }

}

void levelwiseDisplay(struct TreeNode\* root) {

    if (root == NULL)

        return;

    struct Queue\* queue = createQueue();

    enqueue(queue, root);

    enqueue(queue, NULL);  // Using NULL as a delimiter for levels

    while (!isEmpty(queue)) {

        struct TreeNode\* current = dequeue(queue);

        if (current == NULL) {

            printf("\n");

            if (!isEmpty(queue))  // Add delimiter for next level if queue is not empty

                enqueue(queue, NULL);

        } else {

            printf("%d ", current->data);

            if (current->left != NULL)

                enqueue(queue, current->left);

            if (current->right != NULL)

                enqueue(queue, current->right);

        }

    }

    free(queue);

}

void mirrorImage(struct TreeNode\* root) {

    if (root == NULL)

        return;

    struct TreeNode\* temp = root->left;

    root->left = root->right;

    root->right = temp;

    mirrorImage(root->left);

    mirrorImage(root->right);

}

int main() {

    struct TreeNode\* root = NULL;

    int choice, data;

    do {

        printf("\nMenu:\n1. Insert\n2. Display Level-wise Pyramid\n3. Display Mirror Image\n4. Exit\nEnter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter the key to insert: ");

                scanf("%d", &data);

                insert(&root, data);

                break;

            case 2:

                printf("\nLevel-wise Pyramid Display:\n");

                levelwiseDisplay(root);

                break;

            case 3:

                printf("\nMirror Image Display:\n");

                mirrorImage(root);

                levelwiseDisplay(root);

                mirrorImage(root);  // Restore original tree after printing mirror image

                break;

            case 4:

                printf("Exiting the program.\n");

                break;

            default:

                printf("Invalid choice! Please enter a valid option.\n");

        }

    } while (choice != 4);

    return 0;

}

1. **Write a Program to create a Binary Search Tree and perform following nonrecursive operations on it. a. Preorder Traversal b. Postorder Traversal c. Display total Number of Nodes d. Display Leaf nodes.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

    int key;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct TreeNode\* createNode(int key) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->key = key;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int key) {

    if (root == NULL)

        return createNode(key);

    struct TreeNode\* current = root;

    struct TreeNode\* parent = NULL;

    while (current != NULL) {

        parent = current;

        if (key < current->key)

            current = current->left;

        else if (key > current->key)

            current = current->right;

        else {

            printf("Key already exists in the tree.\n");

            return root;

        }

    }

    if (key < parent->key)

        parent->left = createNode(key);

    else

        parent->right = createNode(key);

    return root;

}

void preorderTraversal(struct TreeNode\* root) {

    struct TreeNode\* stack[100];

    int top = -1;

    if (root == NULL)

        return;

    stack[++top] = root;

    while (top >= 0) {

        struct TreeNode\* current = stack[top--];

        printf("%d ", current->key);

        if (current->right != NULL)

            stack[++top] = current->right;

        if (current->left != NULL)

            stack[++top] = current->left;

    }

}

void postorderTraversal(struct TreeNode\* root) {

    struct TreeNode\* stack1[100];

    struct TreeNode\* stack2[100];

    int top1 = -1, top2 = -1;

    if (root == NULL)

        return;

    stack1[++top1] = root;

    while (top1 >= 0) {

        struct TreeNode\* current = stack1[top1--];

        stack2[++top2] = current;

        if (current->left != NULL)

            stack1[++top1] = current->left;

        if (current->right != NULL)

            stack1[++top1] = current->right;

    }

    while (top2 >= 0) {

        struct TreeNode\* current = stack2[top2--];

        printf("%d ", current->key);

    }

}

int countNodes(struct TreeNode\* root) {

    struct TreeNode\* stack[100];

    int top = -1;

    int count = 0;

    if (root == NULL)

        return 0;

    stack[++top] = root;

    while (top >= 0) {

        struct TreeNode\* current = stack[top--];

        count++;

        if (current->right != NULL)

            stack[++top] = current->right;

        if (current->left != NULL)

            stack[++top] = current->left;

    }

    return count;

}

void displayLeafNodes(struct TreeNode\* root) {

    struct TreeNode\* stack[100];

    int top = -1;

    if (root == NULL)

        return;

    stack[++top] = root;

    while (top >= 0) {

        struct TreeNode\* current = stack[top--];

        if (current->left == NULL && current->right == NULL)

            printf("%d ", current->key);

        if (current->right != NULL)

            stack[++top] = current->right;

        if (current->left != NULL)

            stack[++top] = current->left;

    }

}

void displayMenu() {

    printf("\nMenu:\n");

    printf("1. Insert\n");

    printf("2. Preorder Traversal\n");

    printf("3. Postorder Traversal\n");

    printf("4. Display total Number of Nodes\n");

    printf("5. Display Leaf nodes\n");

    printf("6. Exit\n");

}

int main() {

    struct TreeNode\* root = NULL;

    int choice, key, totalNodes;

    do {

        displayMenu();

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter the key to insert: ");

                scanf("%d", &key);

                root = insertNode(root, key);

                break;

            case 2:

                printf("Preorder Traversal: ");

                preorderTraversal(root);

                printf("\n");

                break;

            case 3:

                printf("Postorder Traversal: ");

                postorderTraversal(root);

                printf("\n");

                break;

            case 4:

                totalNodes = countNodes(root);

                printf("Total Number of Nodes: %d\n", totalNodes);

                break;

            case 5:

                printf("Leaf nodes: ");

                displayLeafNodes(root);

                printf("\n");

                break;

            case 6:

                printf("Exiting the program.\n");

                break;

            default:

                printf("Invalid choice! Please enter a valid option.\n");

        }

    } while (choice != 6);

    return 0;

}

1. **Write a Program to create a Binary Search Tree and perform deletion of a node from it. Also display the tree in nonrecursive postorder way.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

    int key;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct TreeNode\* createNode(int key) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->key = key;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int key) {

    if (root == NULL)

        return createNode(key);

    struct TreeNode\* current = root;

    struct TreeNode\* parent = NULL;

    while (current != NULL) {

        parent = current;

        if (key < current->key)

            current = current->left;

        else if (key > current->key)

            current = current->right;

        else {

            printf("Key already exists in the tree.\n");

            return root;

        }

    }

    if (key < parent->key)

        parent->left = createNode(key);

    else

        parent->right = createNode(key);

    return root;

}

struct TreeNode\* findMinNode(struct TreeNode\* node) {

    while (node->left != NULL)

        node = node->left;

    return node;

}

struct TreeNode\* deleteNode(struct TreeNode\* root, int key) {

    if (root == NULL)

        return root;

    if (key < root->key)

        root->left = deleteNode(root->left, key);

    else if (key > root->key)

        root->right = deleteNode(root->right, key);

    else {

        if (root->left == NULL) {

            struct TreeNode\* temp = root->right;

            free(root);

            return temp;

        } else if (root->right == NULL) {

            struct TreeNode\* temp = root->left;

            free(root);

            return temp;

        }

        struct TreeNode\* temp = findMinNode(root->right);

        root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }

    return root;

}

void postorderTraversal(struct TreeNode\* root) {

    struct TreeNode\* stack[100];

    int top = -1;

    if (root == NULL)

        return;

    struct TreeNode\* current = root;

    do {

        while (current != NULL) {

            if (current->right != NULL)

                stack[++top] = current->right;

            stack[++top] = current;

            current = current->left;

        }

        current = stack[top--];

        if (current->right != NULL && stack[top] == current->right) {

            stack[top--];

            stack[++top] = current;

            current = current->right;

        } else {

            printf("%d ", current->key);

            current = NULL;

        }

    } while (top >= 0);

}

void displayMenu() {

    printf("\nMenu:\n");

    printf("1. Insert\n");

    printf("2. Delete\n");

    printf("3. Display Tree (Postorder non-recursive)\n");

    printf("4. Exit\n");

}

int main() {

    struct TreeNode\* root = NULL;

    int choice, key;

    do {

        displayMenu();

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter the key to insert: ");

                scanf("%d", &key);

                root = insertNode(root, key);

                break;

            case 2:

                printf("Enter the key to delete: ");

                scanf("%d", &key);

                root = deleteNode(root, key);

                break;

            case 3:

                printf("Postorder Traversal (non-recursive): ");

                postorderTraversal(root);

                printf("\n");

                break;

            case 4:

                printf("Exiting the program.\n");

                break;

            default:

                printf("Invalid choice! Please enter a valid option.\n");

        }

    } while (choice != 4);

    return 0;

}

1. **Write a Program to create a Binary Search Tree and display it levelwise. Also perform deletion of a node from it.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

    int key;

    struct TreeNode\* left;

    struct TreeNode\* right;

};

struct TreeNode\* createNode(int key) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->key = key;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* root, int key) {

    if (root == NULL)

        return createNode(key);

    struct TreeNode\* current = root;

    struct TreeNode\* parent = NULL;

    while (current != NULL) {

        parent = current;

        if (key < current->key)

            current = current->left;

        else if (key > current->key)

            current = current->right;

        else {

            printf("Key already exists in the tree.\n");

            return root;

        }

    }

    if (key < parent->key)

        parent->left = createNode(key);

    else

        parent->right = createNode(key);

    return root;

}

struct TreeNode\* findMinNode(struct TreeNode\* node) {

    while (node->left != NULL)

        node = node->left;

    return node;

}

struct TreeNode\* deleteNode(struct TreeNode\* root, int key) {

    if (root == NULL)

        return root;

    if (key < root->key)

        root->left = deleteNode(root->left, key);

    else if (key > root->key)

        root->right = deleteNode(root->right, key);

    else {

        if (root->left == NULL) {

            struct TreeNode\* temp = root->right;

            free(root);

            return temp;

        } else if (root->right == NULL) {

            struct TreeNode\* temp = root->left;

            free(root);

            return temp;

        }

        struct TreeNode\* temp = findMinNode(root->right);

        root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);

    }

    return root;

}

void levelOrderTraversal(struct TreeNode\* root) {

    if (root == NULL)

        return;

    struct TreeNode\* queue[100];

    int front = 0, rear = -1;

    queue[++rear] = root;

    while (front <= rear) {

        struct TreeNode\* current = queue[front++];

        printf("%d ", current->key);

        if (current->left != NULL)

            queue[++rear] = current->left;

        if (current->right != NULL)

            queue[++rear] = current->right;

    }

}

void displayMenu() {

    printf("\nMenu:\n");

    printf("1. Insert\n");

    printf("2. Delete\n");

    printf("3. Display Tree (Level Order)\n");

    printf("4. Exit\n");

}

int main() {

    struct TreeNode\* root = NULL;

    int choice, key;

    do {

        displayMenu();

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter the key to insert: ");

                scanf("%d", &key);

                root = insertNode(root, key);

                break;

            case 2:

                printf("Enter the key to delete: ");

                scanf("%d", &key);

                root = deleteNode(root, key);

                break;

            case 3:

                printf("Level Order Traversal: ");

                levelOrderTraversal(root);

                printf("\n");

                break;

            case 4:

                printf("Exiting the program.\n");

                break;

            default:

                printf("Invalid choice! Please enter a valid option.\n");

        }

    } while (choice != 4);

    return 0;

}

1. **Write a Program to create a Binary Search Tree and display its mirror image with and without disturbing the original tree. Also display height of a tree using nonrecursion.**

#include <stdio.h>

#include <stdlib.h>

// Node structure for the Binary Search Tree

struct Node {

    int data;

    struct Node\* left;

    struct Node\* right;

};

// Function to create a new 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 insert a new node into BST

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

    if (root == NULL) {

        root = createNode(data);

        return root;

    }

    if (data <= root->data) {

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

    } else {

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

    }

    return root;

}

// Function to display the tree in inorder traversal

void inorderTraversal(struct Node\* root) {

    if (root != NULL) {

        inorderTraversal(root->left);

        printf("%d ", root->data);

        inorderTraversal(root->right);

    }

}

// Function to calculate the height of the tree non-recursively

int heightNonRecursive(struct Node\* root) {

    if (root == NULL)

        return 0;

    int height = 0;

    struct Node\* queue[1000]; // Assuming a maximum of 1000 nodes

    int front = 0, rear = 0;

    queue[rear++] = root;

    while (front < rear) {

        int size = rear - front;

        height++;

        for (int i = 0; i < size; i++) {

            struct Node\* node = queue[front++];

            if (node->left)

                queue[rear++] = node->left;

            if (node->right)

                queue[rear++] = node->right;

        }

    }

    return height;

}

// Function to swap left and right child nodes to create a mirror image of the tree

void mirror(struct Node\* root) {

    if (root == NULL)

        return;

    // Swap left and right children

    struct Node\* temp = root->left;

    root->left = root->right;

    root->right = temp;

    // Recursively mirror left and right subtrees

    mirror(root->left);

    mirror(root->right);

}

// Example usage

int main() {

    struct Node\* root = NULL;

    int values[] = {5, 3, 8, 1, 4, 7, 9};

    int n = sizeof(values) / sizeof(values[0]);

    // Insert nodes into BST

    for (int i = 0; i < n; i++) {

        root = insert(root, values[i]);

    }

    // Display original tree (inorder traversal)

    printf("Original Tree (Inorder Traversal): ");

    inorderTraversal(root);

    printf("\n");

    // Calculate height of the tree

    printf("Height of the tree: %d\n", heightNonRecursive(root));

    // Display mirror image of the tree

    mirror(root);

    printf("Mirror Image (Inorder Traversal): ");

    inorderTraversal(root);

    printf("\n");

    return 0;

}

1. **Write a program to efficiently search a particular employee record by using Tree data structure. Also sort the data on emp-id in ascending order.**

#include <stdio.h>

#include <stdlib.h>

// Define structure for an employee record

struct Employee {

    int emp\_id;

    char name[50];

    // Add other fields as needed

    struct Employee\* left;

    struct Employee\* right;

};

// Function to create a new employee record

struct Employee\* createEmployee(int emp\_id, char name[]) {

    struct Employee\* newEmployee = (struct Employee\*)malloc(sizeof(struct Employee));

    if (newEmployee == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newEmployee->emp\_id = emp\_id;

    strcpy(newEmployee->name, name);

    newEmployee->left = NULL;

    newEmployee->right = NULL;

    return newEmployee;

}

// Function to insert an employee record into BST

struct Employee\* insertEmployee(struct Employee\* root, int emp\_id, char name[]) {

    if (root == NULL)

        return createEmployee(emp\_id, name);

    if (emp\_id < root->emp\_id)

        root->left = insertEmployee(root->left, emp\_id, name);

    else if (emp\_id > root->emp\_id)

        root->right = insertEmployee(root->right, emp\_id, name);

    return root;

}

// Function to search for an employee record by emp\_id

struct Employee\* searchEmployee(struct Employee\* root, int emp\_id) {

    if (root == NULL || root->emp\_id == emp\_id)

        return root;

    if (emp\_id < root->emp\_id)

        return searchEmployee(root->left, emp\_id);

    return searchEmployee(root->right, emp\_id);

}

// Function to perform inorder traversal and display employee records

void inorderTraversal(struct Employee\* root) {

    if (root != NULL) {

        inorderTraversal(root->left);

        printf("Employee ID: %d, Name: %s\n", root->emp\_id, root->name);

        inorderTraversal(root->right);

    }

}

int main() {

    // Create a sample employee records BST

    struct Employee\* root = NULL;

    root = insertEmployee(root, 101, "John");

    insertEmployee(root, 102, "Alice");

    insertEmployee(root, 105, "Bob");

    insertEmployee(root, 103, "Emma");

    insertEmployee(root, 104, "Michael");

    // Search for an employee record

    int emp\_id\_to\_search = 103;

    struct Employee\* result = searchEmployee(root, emp\_id\_to\_search);

    if (result != NULL) {

        printf("Employee found - ID: %d, Name: %s\n", result->emp\_id, result->name);

    } else {

        printf("Employee with ID %d not found\n", emp\_id\_to\_search);

    }

    // Display employee records in ascending order of emp\_id

    printf("\nEmployee Records (Sorted by Employee ID):\n");

    inorderTraversal(root);

    return 0;

}

1. **Write a Program to create Inorder Threaded Binary Tree and Traverse it in Preorder way.**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Define structure for a threaded binary tree node

struct TreeNode {

    int data;

    struct TreeNode\* left;

    bool isThreadedLeft; // Indicates if left pointer is threaded

    struct TreeNode\* right;

    bool isThreadedRight; // Indicates if right pointer is threaded

};

// Function to create a new node

struct TreeNode\* createNode(int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->data = data;

    newNode->left = NULL;

    newNode->isThreadedLeft = false;

    newNode->right = NULL;

    newNode->isThreadedRight = false;

    return newNode;

}

// Function to perform inorder threading of the tree

struct TreeNode\* createInorderThreadedTree(struct TreeNode\* root) {

    if (root == NULL)

        return NULL;

    struct TreeNode\* prev = NULL;

    struct TreeNode\* current = root;

    // Perform inorder traversal

    while (current != NULL) {

        if (current->left == NULL) {

            // If left child is NULL, thread it to its inorder predecessor

            current->left = prev;

            current->isThreadedLeft = true;

        }

        if (prev != NULL && prev->right == NULL) {

            // If right child of predecessor is NULL, thread it to the current node

            prev->right = current;

            prev->isThreadedRight = true;

        }

        prev = current;

        if (current->isThreadedLeft)

            current = current->right; // Move to inorder successor

        else

            current = current->left; // Move to left child

    }

    return root;

}

// Function to traverse the threaded binary tree in preorder

void preorderTraversal(struct TreeNode\* root) {

    if (root == NULL)

        return;

    struct TreeNode\* current = root;

    while (current != NULL) {

        printf("%d ", current->data);

        // If left child is not threaded, move to the left child

        if (!current->isThreadedLeft)

            current = current->left;

        else {

            // Otherwise, move to the right child

            current = current->right;

            // If right child is threaded, move to its inorder successor

            while (current != NULL && current->isThreadedRight) {

                printf("%d ", current->data);

                current = current->right;

            }

        }

    }

}

int main() {

    // Create a sample threaded binary tree

    struct TreeNode\* root = createNode(1);

    root->left = createNode(2);

    root->right = createNode(3);

    root->left->left = createNode(4);

    root->left->right = createNode(5);

    root->right->left = createNode(6);

    root->right->right = createNode(7);

    // Perform inorder threading of the tree

    struct TreeNode\* threadedRoot = createInorderThreadedTree(root);

    // Traverse the threaded binary tree in preorder

    printf("Preorder Traversal of the Threaded Binary Tree: ");

    preorderTraversal(threadedRoot);

    printf("\n");

    return 0;

}

1. **Write a Program to create Inorder Threaded Binary Tree and Traverse it in Inorder way.**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Define structure for a threaded binary tree node

struct TreeNode {

    int data;

    struct TreeNode\* left;

    bool isThreadedLeft; // Indicates if left pointer is threaded

    struct TreeNode\* right;

    bool isThreadedRight; // Indicates if right pointer is threaded

};

// Function to create a new node

struct TreeNode\* createNode(int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->data = data;

    newNode->left = NULL;

    newNode->isThreadedLeft = false;

    newNode->right = NULL;

    newNode->isThreadedRight = false;

    return newNode;

}

// Function to perform inorder threading of the tree

struct TreeNode\* createInorderThreadedTree(struct TreeNode\* root) {

    if (root == NULL)

        return NULL;

    struct TreeNode\* prev = NULL;

    struct TreeNode\* current = root;

    // Perform inorder traversal

    while (current != NULL) {

        if (current->left == NULL) {

            // If left child is NULL, thread it to its inorder predecessor

            current->left = prev;

            current->isThreadedLeft = true;

        }

        if (prev != NULL && prev->right == NULL) {

            // If right child of predecessor is NULL, thread it to the current node

            prev->right = current;

            prev->isThreadedRight = true;

        }

        prev = current;

        if (current->isThreadedLeft)

            current = current->right; // Move to inorder successor

        else

            current = current->left; // Move to left child

    }

    return root;

}

// Function to traverse the threaded binary tree in inorder

void inorderTraversal(struct TreeNode\* root) {

    if (root == NULL)

        return;

    struct TreeNode\* current = root;

    while (current->left != NULL)

        current = current->left; // Move to the leftmost node

    while (current != NULL) {

        printf("%d ", current->data);

        // If right child is threaded, move to its inorder successor

        if (current->isThreadedRight)

            current = current->right;

        else {

            // Otherwise, move to the leftmost node of the right subtree

            current = current->right;

            while (current != NULL && !current->isThreadedLeft)

                current = current->left;

        }

    }

}

int main() {

    // Create a sample binary tree

    struct TreeNode\* root = createNode(6);

    root->left = createNode(3);

    root->right = createNode(8);

    root->left->left = createNode(1);

    root->left->right = createNode(5);

    root->right->left = createNode(7);

    root->right->right = createNode(9);

    // Perform inorder threading of the tree

    struct TreeNode\* threadedRoot = createInorderThreadedTree(root);

    // Traverse the threaded binary tree in inorder

    printf("Inorder Traversal of the Threaded Binary Tree: ");

    inorderTraversal(threadedRoot);

    printf("\n");

    return 0;

}

1. **Write a Program to implement AVL tree and perform different rotations on it and display it in sorted manner.**

#include <stdio.h>

#include <stdlib.h>

// Define structure for a tree node

struct TreeNode {

    int data;

    struct TreeNode\* left;

    struct TreeNode\* right;

    int height; // Height of the subtree rooted at this node

};

// Function to create a new node

struct TreeNode\* createNode(int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    newNode->height = 1;

    return newNode;

}

// Function to get the height of a node

int height(struct TreeNode\* node) {

    if (node == NULL)

        return 0;

    return node->height;

}

// Function to get the balance factor of a node

int balanceFactor(struct TreeNode\* node) {

    if (node == NULL)

        return 0;

    return height(node->left) - height(node->right);

}

// Function to update the height of a node

void updateHeight(struct TreeNode\* node) {

    if (node == NULL)

        return;

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

}

// Function to perform a right rotation

struct TreeNode\* rightRotate(struct TreeNode\* y) {

    struct TreeNode\* x = y->left;

    struct TreeNode\* T2 = x->right;

    // Perform rotation

    x->right = y;

    y->left = T2;

    // Update heights

    updateHeight(y);

    updateHeight(x);

    return x;

}

// Function to perform a left rotation

struct TreeNode\* leftRotate(struct TreeNode\* x) {

    struct TreeNode\* y = x->right;

    struct TreeNode\* T2 = y->left;

    // Perform rotation

    y->left = x;

    x->right = T2;

    // Update heights

    updateHeight(x);

    updateHeight(y);

    return y;

}

// Function to insert a node into AVL tree

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

    if (root == NULL)

        return createNode(data);

    if (data < root->data)

        root->left = insertNode(root->left, data);

    else if (data > root->data)

        root->right = insertNode(root->right, data);

    else // Duplicate keys are not allowed in AVL trees

        return root;

    // Update height of the current node

    updateHeight(root);

    // Get the balance factor of this node to check if it became unbalanced

    int balance = balanceFactor(root);

    // Perform rotations if necessary to balance the tree

    if (balance > 1 && data < root->left->data) // Left Left case

        return rightRotate(root);

    if (balance < -1 && data > root->right->data) // Right Right case

        return leftRotate(root);

    if (balance > 1 && data > root->left->data) { // Left Right case

        root->left = leftRotate(root->left);

        return rightRotate(root);

    }

    if (balance < -1 && data < root->right->data) { // Right Left case

        root->right = rightRotate(root->right);

        return leftRotate(root);

    }

    return root;

}

// Function to perform inorder traversal and display the tree in sorted order

void inorderTraversal(struct TreeNode\* root) {

    if (root != NULL) {

        inorderTraversal(root->left);

        printf("%d ", root->data);

        inorderTraversal(root->right);

    }

}

int main() {

    struct TreeNode\* root = NULL;

    root = insertNode(root, 10);

    root = insertNode(root, 20);

    root = insertNode(root, 30);

    root = insertNode(root, 40);

    root = insertNode(root, 50);

    root = insertNode(root, 25); // This insertion will cause rotations

    // Display the AVL tree in sorted order

    printf("AVL Tree in sorted order: ");

    inorderTraversal(root);

    printf("\n");

    return 0;

}

1. **Write a Program to implement AVL tree and perform deletion on it and display it in sorted manner.**

#include <stdio.h>

#include <stdlib.h>

// Define structure for a tree node

struct TreeNode {

    int data;

    struct TreeNode\* left;

    struct TreeNode\* right;

    int height; // Height of the subtree rooted at this node

};

// Function to create a new node

struct TreeNode\* createNode(int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(1);

    }

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    newNode->height = 1;

    return newNode;

}

// Function to get the height of a node

int height(struct TreeNode\* node) {

    if (node == NULL)

        return 0;

    return node->height;

}

// Function to update the height of a node

void updateHeight(struct TreeNode\* node) {

    if (node == NULL)

        return;

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

}

// Function to perform a right rotation

struct TreeNode\* rightRotate(struct TreeNode\* y) {

    struct TreeNode\* x = y->left;

    struct TreeNode\* T2 = x->right;

    // Perform rotation

    x->right = y;

    y->left = T2;

    // Update heights

    updateHeight(y);

    updateHeight(x);

    return x;

}

// Function to perform a left rotation

struct TreeNode\* leftRotate(struct TreeNode\* x) {

    struct TreeNode\* y = x->right;

    struct TreeNode\* T2 = y->left;

    // Perform rotation

    y->left = x;

    x->right = T2;

    // Update heights

    updateHeight(x);

    updateHeight(y);

    return y;

}

// Function to get the balance factor of a node

int balanceFactor(struct TreeNode\* node) {

    if (node == NULL)

        return 0;

    return height(node->left) - height(node->right);

}

// Function to perform insertion into AVL tree

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

    if (root == NULL)

        return createNode(data);

    if (data < root->data)

        root->left = insertNode(root->left, data);

    else if (data > root->data)

        root->right = insertNode(root->right, data);

    else // Duplicate keys are not allowed in AVL trees

        return root;

    // Update height of the current node

    updateHeight(root);

    // Get the balance factor of this node to check if it became unbalanced

    int balance = balanceFactor(root);

    // Perform rotations if necessary to balance the tree

    if (balance > 1 && data < root->left->data) // Left Left case

        return rightRotate(root);

    if (balance < -1 && data > root->right->data) // Right Right case

        return leftRotate(root);

    if (balance > 1 && data > root->left->data) { // Left Right case

        root->left = leftRotate(root->left);

        return rightRotate(root);

    }

    if (balance < -1 && data < root->right->data) { // Right Left case

        root->right = rightRotate(root->right);

        return leftRotate(root);

    }

    return root;

}

// Function to find the inorder successor (minimum value node) in a subtree

struct TreeNode\* minValueNode(struct TreeNode\* node) {

    struct TreeNode\* current = node;

    while (current->left != NULL)

        current = current->left;

    return current;

}

// Function to perform deletion in AVL tree

struct TreeNode\* deleteNode(struct TreeNode\* root, int data) {

    if (root == NULL)

        return root;

    // Perform standard BST delete

    if (data < root->data)

        root->left = deleteNode(root->left, data);

    else if (data > root->data)

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

    else {

        // Node to delete found

        // Node with only one child or no child

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

            struct TreeNode\* temp = root->left ? root->left : root->right;

            // No child case

            if (temp == NULL) {

                temp = root;

                root = NULL;

            } else // One child case

                \*root = \*temp; // Copy the contents of the non-empty child

            free(temp);

        } else {

            // Node with two children: Get the inorder successor (smallest in the right subtree)

            struct TreeNode\* temp = minValueNode(root->right);

            // Copy the inorder successor's data to this node

            root->data = temp->data;

            // Delete the inorder successor

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

        }

    }

    // If the tree had only one node then return

    if (root == NULL)

        return root;

    // Update height of the current node

    updateHeight(root);

    // Get the balance factor of this node to check if it became unbalanced

    int balance = balanceFactor(root);

    // Perform rotations if necessary to balance the tree

    if (balance > 1 && balanceFactor(root->left) >= 0) // Left Left case

        return rightRotate(root);

    if (balance > 1 && balanceFactor(root->left) < 0) { // Left Right case

        root->left = leftRotate(root->left);

        return rightRotate(root);

    }

    if (balance < -1 && balanceFactor(root->right) <= 0) // Right Right case

        return leftRotate(root);

    if (balance < -1 && balanceFactor(root->right) > 0) { // Right Left case

        root->right = rightRotate(root->right);

        return leftRotate(root);

    }

    return root;

}

// Function to perform inorder traversal and display the tree in sorted order

void inorderTraversal(struct TreeNode\* root) {

    if (root != NULL) {

        inorderTraversal(root->left);

        printf("%d ", root->data);

        inorderTraversal(root->right);

    }

}

int main() {

    struct TreeNode\* root = NULL;

    root = insertNode(root, 10);

    root = insertNode(root, 20);

    root = insertNode(root, 30);

    root = insertNode(root, 40);

    root = insertNode(root, 50);

    // Display the AVL tree in sorted order after insertions

    printf("AVL Tree in sorted order after insertions: ");

    inorderTraversal(root);

    printf("\n");

    // Delete a node and display the AVL tree in sorted order after deletion

    root = deleteNode(root, 20);

    printf("AVL Tree in sorted order after deletion of 20: ");

    inorderTraversal(root);

    printf("\n");

    return 0;

}

1. **Write a Program to accept a graph from user and represent it with Adjacency Matrix and perform BFS and DFS traversals on it.**

#include <stdio.h>

#include <stdlib.h>

struct Graph {

    int numVertices;

    int\*\* adjMatrix;

};

struct Graph\* createGraph(int numVertices) {

    struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

    graph->numVertices = numVertices;

    graph->adjMatrix = (int\*\*)malloc(numVertices \* sizeof(int\*));

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

        graph->adjMatrix[i] = (int\*)calloc(numVertices, sizeof(int));

    }

    return graph;

}

void addEdge(struct Graph\* graph, int src, int dest) {

    graph->adjMatrix[src][dest] = 1;

    graph->adjMatrix[dest][src] = 1; // If undirected graph

}

void displayAdjMatrix(struct Graph\* graph) {

    printf("Adjacency Matrix:\n");

    for (int i = 0; i < graph->numVertices; ++i) {

        for (int j = 0; j < graph->numVertices; ++j) {

            printf("%d ", graph->adjMatrix[i][j]);

        }

        printf("\n");

    }

}

void BFS(struct Graph\* graph, int startVertex) {

    int\* visited = (int\*)calloc(graph->numVertices, sizeof(int));

    int\* queue = (int\*)malloc(graph->numVertices \* sizeof(int));

    int front = 0, rear = 0;

    printf("Breadth-First Search Traversal: ");

    queue[rear++] = startVertex;

    visited[startVertex] = 1;

    while (front < rear) {

        int current = queue[front++];

        printf("%d ", current);

        for (int i = 0; i < graph->numVertices; ++i) {

            if (graph->adjMatrix[current][i] && !visited[i]) {

                queue[rear++] = i;

                visited[i] = 1;

            }

        }

    }

    printf("\n");

    free(visited);

    free(queue);

}

void DFS(struct Graph\* graph, int startVertex) {

    int\* visited = (int\*)calloc(graph->numVertices, sizeof(int));

    int\* stack = (int\*)malloc(graph->numVertices \* sizeof(int));

    int top = -1;

    printf("Depth-First Search Traversal: ");

    stack[++top] = startVertex;

    visited[startVertex] = 1;

    while (top != -1) {

        int current = stack[top--];

        printf("%d ", current);

        for (int i = 0; i < graph->numVertices; ++i) {

            if (graph->adjMatrix[current][i] && !visited[i]) {

                stack[++top] = i;

                visited[i] = 1;

            }

        }

    }

    printf("\n");

    free(visited);

    free(stack);

}

int main() {

    int numVertices, numEdges;

    printf("Enter the number of vertices in the graph: ");

    scanf("%d", &numVertices);

    printf("Enter the number of edges in the graph: ");

    scanf("%d", &numEdges);

    struct Graph\* graph = createGraph(numVertices);

    printf("Enter the edges (source and destination vertices) in the graph:\n");

    for (int i = 0; i < numEdges; ++i) {

        int src, dest;

        scanf("%d %d", &src, &dest);

        addEdge(graph, src, dest);

    }

    displayAdjMatrix(graph);

    int startVertex;

    printf("Enter the starting vertex for BFS and DFS traversals: ");

    scanf("%d", &startVertex);

    BFS(graph, startVertex);

    DFS(graph, startVertex);

    // Free allocated memory

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

        free(graph->adjMatrix[i]);

    }

    free(graph->adjMatrix);

    free(graph);

    return 0;

}

1. **Write a Program to accept a graph from user and represent it with Adjacency List and perform BFS and DFS traversals on it.**

#include <stdio.h>

#include <stdlib.h>

struct Node {

    int vertex;

    struct Node\* next;

};

struct Graph {

    int numVertices;

    struct Node\*\* adjList;

};

struct Graph\* createGraph(int numVertices) {

    struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

    graph->numVertices = numVertices;

    graph->adjList = (struct Node\*\*)malloc(numVertices \* sizeof(struct Node\*));

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

        graph->adjList[i] = NULL;

    }

    return graph;

}

void addEdge(struct Graph\* graph, int src, int dest) {

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

    newNode->vertex = dest;

    newNode->next = graph->adjList[src];

    graph->adjList[src] = newNode;

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

    newNode->vertex = src;

    newNode->next = graph->adjList[dest];

    graph->adjList[dest] = newNode;

}

void displayAdjList(struct Graph\* graph) {

    printf("Adjacency List:\n");

    for (int i = 0; i < graph->numVertices; ++i) {

        printf("%d -> ", i);

        struct Node\* temp = graph->adjList[i];

        while (temp) {

            printf("%d ", temp->vertex);

            temp = temp->next;

        }

        printf("\n");

    }

}

void BFS(struct Graph\* graph, int startVertex) {

    int\* visited = (int\*)calloc(graph->numVertices, sizeof(int));

    int\* queue = (int\*)malloc(graph->numVertices \* sizeof(int));

    int front = 0, rear = 0;

    printf("Breadth-First Search Traversal: ");

    queue[rear++] = startVertex;

    visited[startVertex] = 1;

    while (front < rear) {

        int current = queue[front++];

        printf("%d ", current);

        struct Node\* temp = graph->adjList[current];

        while (temp) {

            int neighbor = temp->vertex;

            if (!visited[neighbor]) {

                queue[rear++] = neighbor;

                visited[neighbor] = 1;

            }

            temp = temp->next;

        }

    }

    printf("\n");

    free(visited);

    free(queue);

}

void DFSUtil(struct Graph\* graph, int vertex, int\* visited) {

    visited[vertex] = 1;

    printf("%d ", vertex);

    struct Node\* temp = graph->adjList[vertex];

    while (temp) {

        int neighbor = temp->vertex;

        if (!visited[neighbor]) {

            DFSUtil(graph, neighbor, visited);

        }

        temp = temp->next;

    }

}

void DFS(struct Graph\* graph, int startVertex) {

    int\* visited = (int\*)calloc(graph->numVertices, sizeof(int));

    printf("Depth-First Search Traversal: ");

    DFSUtil(graph, startVertex, visited);

    printf("\n");

    free(visited);

}

int main() {

    int numVertices, numEdges;

    printf("Enter the number of vertices in the graph: ");

    scanf("%d", &numVertices);

    printf("Enter the number of edges in the graph: ");

    scanf("%d", &numEdges);

    struct Graph\* graph = createGraph(numVertices);

    printf("Enter the edges (source and destination vertices) in the graph:\n");

    for (int i = 0; i < numEdges; ++i) {

        int src, dest;

        scanf("%d %d", &src, &dest);

        addEdge(graph, src, dest);

    }

    displayAdjList(graph);

    int startVertex;

    printf("Enter the starting vertex for BFS and DFS traversals: ");

    scanf("%d", &startVertex);

    BFS(graph, startVertex);

    DFS(graph, startVertex);

    // Free allocated memory

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

        struct Node\* temp = graph->adjList[i];

        while (temp) {

            struct Node\* prev = temp;

            temp = temp->next;

            free(prev);

        }

    }

    free(graph->adjList);

    free(graph);

    return 0;

}

1. **Write a Program to implement Prim’s algorithm to find minimum spanning tree of a user defined graph. Use Adjacency Matrix to represent a graph.**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

#define MAX\_VERTICES 100

// Function to find the vertex with minimum key value

int minKey(int key[], bool mstSet[], int vertices) {

    int min = INT\_MAX, minIndex;

    for (int v = 0; v < vertices; v++) {

        if (mstSet[v] == false && key[v] < min) {

            min = key[v];

            minIndex = v;

        }

    }

    return minIndex;

}

// Function to print the constructed MST stored in parent[]

void printMST(int parent[], int graph[MAX\_VERTICES][MAX\_VERTICES], int vertices) {

    printf("Edge \tWeight\n");

    for (int i = 1; i < vertices; i++)

        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);

}

// Function to implement Prim's algorithm to find MST of a graph

void primMST(int graph[MAX\_VERTICES][MAX\_VERTICES], int vertices) {

    int parent[vertices]; // Array to store constructed MST

    int key[vertices];    // Key values used to pick minimum weight edge in cut

    bool mstSet[vertices]; // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE

    for (int i = 0; i < vertices; i++) {

        key[i] = INT\_MAX;

        mstSet[i] = false;

    }

    // Always include first 0th vertex in MST.

    key[0] = 0; // Make key 0 so that this vertex is picked as first vertex

    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices

    for (int count = 0; count < vertices - 1; count++) {

        // Pick the minimum key vertex from the set of vertices not yet included in MST

        int u = minKey(key, mstSet, vertices);

        // Add the picked vertex to the MST Set

        mstSet[u] = true;

        // Update key value and parent index of the adjacent vertices of the picked vertex.

        // Consider only those vertices which are not yet included in MST

        for (int v = 0; v < vertices; v++) {

            // graph[u][v] is non zero only for adjacent vertices of m

            // mstSet[v] is false for vertices not yet included in MST

            // Update the key only if graph[u][v] is smaller than key[v]

            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {

                parent[v] = u;

                key[v] = graph[u][v];

            }

        }

    }

    // Print the constructed MST

    printMST(parent, graph, vertices);

}

int main() {

    int vertices, edges;

    int graph[MAX\_VERTICES][MAX\_VERTICES];

    // Accept number of vertices and edges from user

    printf("Enter the number of vertices: ");

    scanf("%d", &vertices);

    printf("Enter the number of edges: ");

    scanf("%d", &edges);

    // Initialize graph with all 0s

    for (int i = 0; i < MAX\_VERTICES; i++) {

        for (int j = 0; j < MAX\_VERTICES; j++) {

            graph[i][j] = 0;

        }

    }

    // Accept edges from user and their weights

    printf("Enter edges (source destination weight):\n");

    for (int i = 0; i < edges; i++) {

        int source, destination, weight;

        scanf("%d %d %d", &source, &destination, &weight);

        graph[source][destination] = weight;

        graph[destination][source] = weight; // For undirected graph

    }

    // Print the Minimum Spanning Tree (MST) using Prim's algorithm

    printf("Minimum Spanning Tree using Prim's algorithm:\n");

    primMST(graph, vertices);

    return 0;

}

1. **WAP to implement heap sort on 1D array of Student structure (contains student\_name, student\_roll\_no, total\_marks), with key as student\_roll\_no. And count the number of swap performed.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Structure to represent a Student

struct Student {

    char student\_name[50];

    int student\_roll\_no;

    int total\_marks;

};

// Function to merge two subarrays arr[l..m] and arr[m+1..r]

// and count the number of swaps performed

void merge(struct Student arr[], int l, int m, int r, int \*swapCount) {

    int n1 = m - l + 1;

    int n2 = r - m;

    // Create temporary arrays

    struct Student L[n1], R[n2];

    // Copy data to temporary arrays L[] and R[]

    for (int i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (int j = 0; j < n2; j++)

        R[j] = arr[m + 1 + j];

    // Merge the temporary arrays back into arr[l..r]

    int i = 0, j = 0, k = l;

    while (i < n1 && j < n2) {

        if (L[i].student\_roll\_no <= R[j].student\_roll\_no) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

            // Count the number of swaps performed

            \*swapCount += n1 - i;

        }

        k++;

    }

    // Copy the remaining elements of L[], if any

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    // Copy the remaining elements of R[], if any

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

// Main function to implement merge sort on the student array

void mergeSort(struct Student arr[], int l, int r, int \*swapCount) {

    if (l < r) {

        // Same as (l+r)/2, but avoids overflow for large l and r

        int m = l + (r - l) / 2;

        // Sort first and second halves

        mergeSort(arr, l, m, swapCount);

        mergeSort(arr, m + 1, r, swapCount);

        // Merge the sorted halves

        merge(arr, l, m, r, swapCount);

    }

}

int main() {

    int n;

    printf("Enter the number of students: ");

    scanf("%d", &n);

    struct Student \*students = (struct Student \*)malloc(n \* sizeof(struct Student));

    // Accept student details from the user

    printf("Enter student details:\n");

    for (int i = 0; i < n; i++) {

        printf("Enter details for student %d:\n", i + 1);

        printf("Name: ");

        scanf("%s", students[i].student\_name);

        printf("Roll No.: ");

        scanf("%d", &students[i].student\_roll\_no);

        printf("Total Marks: ");

        scanf("%d", &students[i].total\_marks);

    }

    // Perform merge sort on the student array based on student\_roll\_no

    int swapCount = 0;

    mergeSort(students, 0, n - 1, &swapCount);

    // Display the sorted array

    printf("\nSorted student details based on roll number:\n");

    printf("Name\tRoll No.\tTotal Marks\n");

    for (int i = 0; i < n; i++) {

        printf("%s\t%d\t\t%d\n", students[i].student\_name, students[i].student\_roll\_no, students[i].total\_marks);

    }

    // Display the number of swaps performed

    printf("\nNumber of swaps performed: %d\n", swapCount);

    free(students);

    return 0;

}

1. **WAP to convert a given Infix expression into its equivalent Postfix expression and evaluate it using stack.\**

1. **WAP to implement stack using a singly linked list and perform following operations on it. A. PUSH, B. POP, C. StackeEmpty D. Display Stack.**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Structure to represent a node in the singly linked list

struct Node {

    int data;

    struct Node\* next;

};

// Structure to represent the stack

struct Stack {

    struct Node\* top;

};

// Function to create an empty stack

struct Stack\* createStack() {

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    stack->top = NULL;

    return stack;

}

// Function to check if the stack is empty

bool StackEmpty(struct Stack\* stack) {

    return stack->top == NULL;

}

// Function to push an element onto the stack

void push(struct Stack\* stack, int data) {

    // Create a new node

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

    if (newNode == NULL) {

        printf("Memory allocation failed.\n");

        return;

    }

    // Assign data to the new node

    newNode->data = data;

    // Link the new node to the current top of the stack

    newNode->next = stack->top;

    // Make the new node the new top of the stack

    stack->top = newNode;

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

    if (StackEmpty(stack)) {

        printf("Stack is empty.\n");

        return -1; // Return -1 indicating stack underflow

    }

    // Store the data of the top node

    int data = stack->top->data;

    // Move the top to the next node

    struct Node\* temp = stack->top;

    stack->top = stack->top->next;

    // Free the memory of the popped node

    free(temp);

    // Return the popped data

    return data;

}

// Function to display the stack

void displayStack(struct Stack\* stack) {

    if (StackEmpty(stack)) {

        printf("Stack is empty.\n");

        return;

    }

    printf("Stack: ");

    struct Node\* current = stack->top;

    while (current != NULL) {

        printf("%d ", current->data);

        current = current->next;

    }

    printf("\n");

}

int main() {

    struct Stack\* stack = createStack();

    // Push elements onto the stack

    push(stack, 10);

    push(stack, 20);

    push(stack, 30);

    // Display the stack

    displayStack(stack);

    // Pop an element from the stack

    int popped = pop(stack);

    if (popped != -1)

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

    // Display the stack after popping

    displayStack(stack);

    // Pop all elements from the stack

    while (!StackEmpty(stack)) {

        popped = pop(stack);

        if (popped != -1)

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

    }

    // Display the stack after popping all elements

    displayStack(stack);

    return 0;

}

1. **WAP to implement following by using stack. A. Factorial of a given number B. Generation of Fibonacci series.**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the stack

struct Node {

    int data;

    struct Node\* next;

};

// Structure to represent the stack

struct Stack {

    struct Node\* top;

};

// Function to create an empty stack

struct Stack\* createStack() {

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    stack->top = NULL;

    return stack;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

    return stack->top == NULL;

}

// Function to push an element onto the stack

void push(struct Stack\* stack, int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed.\n");

        return;

    }

    newNode->data = data;

    newNode->next = stack->top;

    stack->top = newNode;

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

    if (isEmpty(stack)) {

        printf("Stack is empty.\n");

        return -1; // Return -1 indicating stack underflow

    }

    int data = stack->top->data;

    struct Node\* temp = stack->top;

    stack->top = stack->top->next;

    free(temp);

    return data;

}

// Function to calculate factorial of a number using stack

int factorial(int n) {

    struct Stack\* stack = createStack();

    int fact = 1;

    for (int i = 1; i <= n; i++) {

        push(stack, i);

    }

    while (!isEmpty(stack)) {

        fact \*= pop(stack);

    }

    return fact;

}

// Function to generate Fibonacci series using stack

void fibonacci(int n) {

    struct Stack\* stack = createStack();

    push(stack, 0);

    push(stack, 1);

    printf("Fibonacci series up to %d terms:\n", n);

    if (n >= 1) {

        printf("0 ");

    }

    if (n >= 2) {

        printf("1 ");

    }

    for (int i = 3; i <= n; i++) {

        int second = pop(stack);

        int first = pop(stack);

        int next = first + second;

        push(stack, second);

        push(stack, next);

        printf("%d ", next);

    }

    printf("\n");

}

int main() {

    int choice, n;

    do {

        printf("Select an option:\n");

        printf("1. Calculate factorial of a number\n");

        printf("2. Generate Fibonacci series\n");

        printf("3. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter a number to calculate factorial: ");

                scanf("%d", &n);

                printf("Factorial of %d is %d\n", n, factorial(n));

                break;

            case 2:

                printf("Enter the number of terms in Fibonacci series: ");

                scanf("%d", &n);

                fibonacci(n);

                break;

            case 3:

                printf("Exiting...\n");

                break;

            default:

                printf("Invalid choice! Please enter a valid option.\n");

        }

    } while (choice != 3);

    return 0;

}

24

1. **WAP to implement a linear queue using a singly linked list and perform following operations on it. A. enqueue, B. dequeue, C. QueueEmpty, D. Display queue, E. Display Front element, F. Display Rear element**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Structure to represent a node in the queue

struct Node {

    int data;

    struct Node\* next;

};

// Structure to represent the queue

struct Queue {

    struct Node\* front;

    struct Node\* rear;

};

// Function to create an empty queue

struct Queue\* createQueue() {

    struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

    queue->front = NULL;

    queue->rear = NULL;

    return queue;

}

// Function to check if the queue is empty

bool QueueEmpty(struct Queue\* queue) {

    return queue->front == NULL;

}

// Function to enqueue an element into the queue

void enqueue(struct Queue\* queue, int data) {

    // Create a new node

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

    if (newNode == NULL) {

        printf("Memory allocation failed.\n");

        return;

    }

    newNode->data = data;

    newNode->next = NULL;

    // If the queue is empty, make the new node front and rear

    if (QueueEmpty(queue)) {

        queue->front = newNode;

        queue->rear = newNode;

    } else {

        // Link the new node to the current rear and update rear

        queue->rear->next = newNode;

        queue->rear = newNode;

    }

}

// Function to dequeue an element from the queue

int dequeue(struct Queue\* queue) {

    if (QueueEmpty(queue)) {

        printf("Queue is empty.\n");

        return -1; // Return -1 indicating queue underflow

    }

    // Store the data of the front node

    int data = queue->front->data;

    // Move the front to the next node

    struct Node\* temp = queue->front;

    queue->front = queue->front->next;

    // If the queue becomes empty after dequeuing, update rear to NULL

    if (queue->front == NULL)

        queue->rear = NULL;

    // Free the memory of the dequeued node

    free(temp);

    // Return the dequeued data

    return data;

}

// Function to display the queue

void displayQueue(struct Queue\* queue) {

    if (QueueEmpty(queue)) {

        printf("Queue is empty.\n");

        return;

    }

    printf("Queue: ");

    struct Node\* current = queue->front;

    while (current != NULL) {

        printf("%d ", current->data);

        current = current->next;

    }

    printf("\n");

}

// Function to display the front element of the queue

void displayFront(struct Queue\* queue) {

    if (QueueEmpty(queue)) {

        printf("Queue is empty.\n");

        return;

    }

    printf("Front element: %d\n", queue->front->data);

}

// Function to display the rear element of the queue

void displayRear(struct Queue\* queue) {

    if (QueueEmpty(queue)) {

        printf("Queue is empty.\n");

        return;

    }

    printf("Rear element: %d\n", queue->rear->data);

}

int main() {

    struct Queue\* queue = createQueue();

    // Enqueue elements into the queue

    enqueue(queue, 10);

    enqueue(queue, 20);

    enqueue(queue, 30);

    // Display the queue

    displayQueue(queue);

    // Display the front and rear elements of the queue

    displayFront(queue);

    displayRear(queue);

    // Dequeue an element from the queue

    int dequeued = dequeue(queue);

    if (dequeued != -1)

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

    // Display the queue after dequeue operation

    displayQueue(queue);

    return 0;

}

25

1. **      Write a Program to implement circular queue where user can add and remove the elements from rear and front end of the queue**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 5 // Maximum size of the circular queue

// Structure to represent a circular queue

struct CircularQueue {

    int\* array;

    int front, rear, size;

    unsigned capacity;

};

// Function to create an empty circular queue

struct CircularQueue\* createCircularQueue(unsigned capacity) {

    struct CircularQueue\* queue = (struct CircularQueue\*)malloc(sizeof(struct CircularQueue));

    queue->capacity = capacity;

    queue->size = 0;

    queue->front = 0;

    queue->rear = -1; // -1 indicates an empty queue

    queue->array = (int\*)malloc(queue->capacity \* sizeof(int));

    return queue;

}

// Function to check if the circular queue is empty

bool isEmpty(struct CircularQueue\* queue) {

    return queue->size == 0;

}

// Function to check if the circular queue is full

bool isFull(struct CircularQueue\* queue) {

    return queue->size == queue->capacity;

}

// Function to add an element to the rear end of the circular queue

void enqueueRear(struct CircularQueue\* queue, int item) {

    if (isFull(queue)) {

        printf("Queue is full. Cannot enqueue.\n");

        return;

    }

    queue->rear = (queue->rear + 1) % queue->capacity; // Circular increment

    queue->array[queue->rear] = item;

    queue->size++;

    printf("%d enqueued to rear end of the queue.\n", item);

}

// Function to add an element to the front end of the circular queue

void enqueueFront(struct CircularQueue\* queue, int item) {

    if (isFull(queue)) {

        printf("Queue is full. Cannot enqueue.\n");

        return;

    }

    queue->front = (queue->front - 1 + queue->capacity) % queue->capacity; // Circular decrement

    queue->array[queue->front] = item;

    queue->size++;

    printf("%d enqueued to front end of the queue.\n", item);

}

// Function to remove an element from the rear end of the circular queue

int dequeueRear(struct CircularQueue\* queue) {

    if (isEmpty(queue)) {

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

        return -1; // Return -1 indicating queue underflow

    }

    int item = queue->array[queue->rear];

    queue->rear = (queue->rear - 1 + queue->capacity) % queue->capacity; // Circular decrement

    queue->size--;

    return item;

}

// Function to remove an element from the front end of the circular queue

int dequeueFront(struct CircularQueue\* queue) {

    if (isEmpty(queue)) {

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

        return -1; // Return -1 indicating queue underflow

    }

    int item = queue->array[queue->front];

    queue->front = (queue->front + 1) % queue->capacity; // Circular increment

    queue->size--;

    return item;

}

// Function to display the circular queue

void displayQueue(struct CircularQueue\* queue) {

    if (isEmpty(queue)) {

        printf("Queue is empty.\n");

        return;

    }

    printf("Circular Queue: ");

    int i;

    for (i = queue->front; i != queue->rear; i = (i + 1) % queue->capacity)

        printf("%d ", queue->array[i]);

    printf("%d\n", queue->array[i]);

}

int main() {

    struct CircularQueue\* queue = createCircularQueue(MAX\_SIZE);

    enqueueRear(queue, 10);

    enqueueRear(queue, 20);

    enqueueRear(queue, 30);

    displayQueue(queue);

    enqueueFront(queue, 40);

    enqueueFront(queue, 50);

    displayQueue(queue);

    printf("Dequeued from rear end: %d\n", dequeueRear(queue));

    printf("Dequeued from front end: %d\n", dequeueFront(queue));

    displayQueue(queue);

    return 0;

}

26

1. **      WAP to perform addition of two polynomials using singly linked list.**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a term in a polynomial

struct Term {

    int coefficient;

    int exponent;

    struct Term\* next;

};

// Function to create a new term

struct Term\* createTerm(int coefficient, int exponent) {

    struct Term\* newTerm = (struct Term\*)malloc(sizeof(struct Term));

    if (newTerm == NULL) {

        printf("Memory allocation failed.\n");

        exit(EXIT\_FAILURE);

    }

    newTerm->coefficient = coefficient;

    newTerm->exponent = exponent;

    newTerm->next = NULL;

    return newTerm;

}

// Function to insert a term at the end of a polynomial

void insertTerm(struct Term\*\* poly, int coefficient, int exponent) {

    struct Term\* newTerm = createTerm(coefficient, exponent);

    if (\*poly == NULL) {

        \*poly = newTerm;

    } else {

        struct Term\* current = \*poly;

        while (current->next != NULL)

            current = current->next;

        current->next = newTerm;

    }

}

// Function to add two polynomials

struct Term\* addPolynomials(struct Term\* poly1, struct Term\* poly2) {

    struct Term\* result = NULL;

    while (poly1 != NULL && poly2 != NULL) {

        if (poly1->exponent > poly2->exponent) {

            insertTerm(&result, poly1->coefficient, poly1->exponent);

            poly1 = poly1->next;

        } else if (poly1->exponent < poly2->exponent) {

            insertTerm(&result, poly2->coefficient, poly2->exponent);

            poly2 = poly2->next;

        } else {

            int sum = poly1->coefficient + poly2->coefficient;

            if (sum != 0)

                insertTerm(&result, sum, poly1->exponent);

            poly1 = poly1->next;

            poly2 = poly2->next;

        }

    }

    // If there are remaining terms in either polynomial, add them to the result

    while (poly1 != NULL) {

        insertTerm(&result, poly1->coefficient, poly1->exponent);

        poly1 = poly1->next;

    }

    while (poly2 != NULL) {

        insertTerm(&result, poly2->coefficient, poly2->exponent);

        poly2 = poly2->next;

    }

    return result;

}

// Function to display a polynomial

void displayPolynomial(struct Term\* poly) {

    if (poly == NULL) {

        printf("0\n");

        return;

    }

    while (poly != NULL) {

        printf("%dx^%d", poly->coefficient, poly->exponent);

        poly = poly->next;

        if (poly != NULL)

            printf(" + ");

    }

    printf("\n");

}

int main() {

    // Create first polynomial: 3x^3 + 4x^2 + 2x^1

    struct Term\* poly1 = NULL;

    insertTerm(&poly1, 3, 3);

    insertTerm(&poly1, 4, 2);

    insertTerm(&poly1, 2, 1);

    printf("First polynomial: ");

    displayPolynomial(poly1);

    // Create second polynomial: 5x^4 + 2x^2 + 1x^0

    struct Term\* poly2 = NULL;

    insertTerm(&poly2, 5, 4);

    insertTerm(&poly2, 2, 2);

    insertTerm(&poly2, 1, 0);

    printf("Second polynomial: ");

    displayPolynomial(poly2);

    // Add the two polynomials

    struct Term\* sum = addPolynomials(poly1, poly2);

    printf("Sum of polynomials: ");

    displayPolynomial(sum);

    return 0;

}

27

1. **      Write a Reverse() function that reverses a Singly linked list and display the list**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the singly linked list

struct Node {

    int data;

    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));

    if (newNode == NULL) {

        printf("Memory allocation failed.\n");

        exit(EXIT\_FAILURE);

    }

    newNode->data = data;

    newNode->next = NULL;

    return newNode;

}

// Function to insert a node at the end of the linked list

void insertEnd(struct Node\*\* head, int data) {

    struct Node\* newNode = createNode(data);

    if (\*head == NULL) {

        \*head = newNode;

    } else {

        struct Node\* current = \*head;

        while (current->next != NULL)

            current = current->next;

        current->next = newNode;

    }

}

// Function to reverse a singly linked list

struct Node\* reverse(struct Node\* head) {

    struct Node\* prev = NULL;

    struct Node\* current = head;

    struct Node\* next = NULL;

    while (current != NULL) {

        next = current->next; // Store next node

        current->next = prev; // Reverse current node's pointer

        prev = current; // Move pointers one position ahead

        current = next;

    }

    head = prev; // Update head to the last node

    return head;

}

// Function to display the elements of a singly linked list

void displayList(struct Node\* head) {

    if (head == NULL) {

        printf("List is empty.\n");

        return;

    }

    printf("List: ");

    while (head != NULL) {

        printf("%d ", head->data);

        head = head->next;

    }

    printf("\n");

}

int main() {

    // Create a singly linked list: 1 -> 2 -> 3 -> 4 -> 5

    struct Node\* head = NULL;

    insertEnd(&head, 1);

    insertEnd(&head, 2);

    insertEnd(&head, 3);

    insertEnd(&head, 4);

    insertEnd(&head, 5);

    printf("Original ");

    displayList(head);

    // Reverse the list

    head = reverse(head);

    printf("Reversed ");

    displayList(head);

    return 0;

}

28

1. **      WAP to create doubly linked list and perform following operations on it. A) Insert (all cases) 2. Delete (all cases).**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent 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));

    if (newNode == NULL) {

        printf("Memory allocation failed.\n");

        exit(EXIT\_FAILURE);

    }

    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 insertBegin(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 insertEnd(struct Node\*\* head, int data) {

    struct Node\* newNode = createNode(data);

    if (\*head == NULL) {

        \*head = newNode;

    } else {

        struct Node\* current = \*head;

        while (current->next != NULL)

            current = current->next;

        current->next = newNode;

        newNode->prev = current;

    }

}

// Function to insert a node after a given node in the doubly linked list

void insertAfter(struct Node\* prevNode, int data) {

    if (prevNode == NULL) {

        printf("Previous node cannot be NULL.\n");

        return;

    }

    struct Node\* newNode = createNode(data);

    newNode->next = prevNode->next;

    if (prevNode->next != NULL)

        prevNode->next->prev = newNode;

    prevNode->next = newNode;

    newNode->prev = prevNode;

}

// Function to delete the first occurrence of a node with given key in the doubly linked list

void deleteNode(struct Node\*\* head, int key) {

    if (\*head == NULL) {

        printf("List is empty. Cannot delete.\n");

        return;

    }

    struct Node\* current = \*head;

    // If the key is found in the first node itself

    if (current->data == key) {

        \*head = current->next;

        if (\*head != NULL)

            (\*head)->prev = NULL;

        free(current);

        return;

    }

    // Search for the node with the key

    while (current != NULL && current->data != key)

        current = current->next;

    // If the key is not found in the list

    if (current == NULL) {

        printf("Key not found in the list.\n");

        return;

    }

    // Remove the node

    if (current->prev != NULL)

        current->prev->next = current->next;

    if (current->next != NULL)

        current->next->prev = current->prev;

    free(current);

}

// Function to display the doubly linked list in forward direction

void displayForward(struct Node\* head) {

    if (head == NULL) {

        printf("List is empty.\n");

        return;

    }

    printf("Forward List: ");

    while (head != NULL) {

        printf("%d ", head->data);

        head = head->next;

    }

    printf("\n");

}

// Function to display the doubly linked list in reverse direction

void displayBackward(struct Node\* head) {

    if (head == NULL) {

        printf("List is empty.\n");

        return;

    }

    // Move to the last node

    while (head->next != NULL)

        head = head->next;

    printf("Backward List: ");

    // Traverse backwards and print the data

    while (head != NULL) {

        printf("%d ", head->data);

        head = head->prev;

    }

    printf("\n");

}

int main() {

    struct Node\* head = NULL;

    // Insert at the beginning

    insertBegin(&head, 10);

    insertBegin(&head, 20);

    insertBegin(&head, 30);

    displayForward(head);

    displayBackward(head);

    // Insert at the end

    insertEnd(&head, 40);

    insertEnd(&head, 50);

    displayForward(head);

    displayBackward(head);

    // Insert after a specific node

    insertAfter(head->next, 35);

    displayForward(head);

    displayBackward(head);

    // Delete a node

    deleteNode(&head, 30);

    displayForward(head);

    displayBackward(head);

    return 0;

}

29

1. **  WAP to merge two sorted Doubly linked lists and display their result.**

#include <stdlib.h>

// Node structure for the doubly linked list

struct Node {

    int data;

    struct Node\* prev;

    struct Node\* next;

};

// Function to create a new node

struct Node\* getNode(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 end

void insertAtEnd(struct Node\*\* head\_ref, int data) {

    struct Node\* newNode = getNode(data);

    if (\*head\_ref == NULL) {

        \*head\_ref = newNode;

        return;

    }

    struct Node\* temp = \*head\_ref;

    while (temp->next != NULL) {

        temp = temp->next;

    }

    temp->next = newNode;

    newNode->prev = temp;

}

// Function to merge two sorted doubly linked lists

struct Node\* mergeSortedLists(struct Node\* head1, struct Node\* head2) {

    if (head1 == NULL) return head2;

    if (head2 == NULL) return head1;

    if (head1->data < head2->data) {

        head1->next = mergeSortedLists(head1->next, head2);

        head1->next->prev = head1;

        head1->prev = NULL;

        return head1;

    } else {

        head2->next = mergeSortedLists(head1, head2->next);

        head2->next->prev = head2;

        head2->prev = NULL;

        return head2;

    }

}

// Function to print the doubly linked list

void printList(struct Node\* node) {

    while (node != NULL) {

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

        node = node->next;

    }

    printf("NULL\n");

}

int main() {

    struct Node\* head1 = NULL;

    struct Node\* head2 = NULL;

    // Create first sorted doubly linked list: 1 <-> 3 <-> 5 <-> 7

    insertAtEnd(&head1, 1);

    insertAtEnd(&head1, 3);

    insertAtEnd(&head1, 5);

    insertAtEnd(&head1, 7);

    // Create second sorted doubly linked list: 2 <-> 4 <-> 6 <-> 8

    insertAtEnd(&head2, 2);

    insertAtEnd(&head2, 4);

    insertAtEnd(&head2, 6);

    insertAtEnd(&head2, 8);

    printf("First Sorted Doubly Linked List:\n");

    printList(head1);

    printf("Second Sorted Doubly Linked List:\n");

    printList(head2);

    // Merge the two sorted doubly linked lists

    struct Node\* mergedHead = mergeSortedLists(head1, head2);

    printf("Merged Sorted Doubly Linked List:\n");

    printList(mergedHead);

    return 0;

}

1. **Write a Program to create Inorder Threaded Binary Tree and Traverse it in Postorder way.**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

    struct Node \*left, \*right;

    int data;

    int rightThread;

} Node;

Node \*createNode(int item) {

    Node \*node = (Node \*)malloc(sizeof(Node));

    node->left = node->right = NULL;

    node->data = item;

    node->rightThread = 0;

    return node;

}

void createThreaded(Node \*root) {

    static Node \*prev = NULL;

    if (root == NULL)

        return;

    createThreaded(root->left);

    if (prev != NULL && prev->right == NULL) {

        prev->right = root;

        prev->rightThread = 1;

    }

    if (root->left == NULL)

        root->left = prev;

    prev = root;

    createThreaded(root->right);

}

// Function to find the rightmost node in a subtree

Node\* rightmost(Node\* node) {

    while (node != NULL && node->rightThread == 0 && node->right != NULL)

        node = node->right;

    return node;

}

// Function to find the postorder predecessor of a node

Node\* postorderPredecessor(Node\* node) {

    // If the node has a left child, the predecessor is the rightmost node of the left child

    if (node->left != NULL && node->left->right != node) {

        return rightmost(node->left);

    }

    // Otherwise, return the left child itself

    return node->left;

}

// Function to perform postorder traversal of the threaded binary tree

void postOrderTraversal(Node\* root) {

    Node\* curr = rightmost(root);

    while (curr != NULL) {

        printf("%d ", curr->data);

        curr = postorderPredecessor(curr);

    }

}

int main() {

    Node \*root = createNode(1);

    root->left = createNode(2);

    root->right = createNode(3);

    root->left->left = createNode(4);

    root->left->right = createNode(5);

    root->right->left = createNode(6);

    root->right->right = createNode(7);

    createThreaded(root);

    printf("Postorder traversal of the created threaded binary tree is: \n");

    postOrderTraversal(root);

    return 0;

}