## ADDITIONAL PROGRAMS – MODULE 3 AND 5

1. **Merge Sort algorithm for linked lists:**

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a linked list node

typedef struct Node {

int data;

struct Node\* next;

} Node;

// Function to create a new linked list node

Node\* createNode(int data) {

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

if (!newNode) {

printf("Memory error\n");

return NULL;

}

newNode->data = data;

newNode->next = NULL;

return newNode;

}

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

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

Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

return;

}

Node\* lastNode = \*head;

while (lastNode->next) {

lastNode = lastNode->next;

}

lastNode->next = newNode;

}

// Function to print the linked list

void printList(Node\* head) {

while (head) {

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

head = head->next;

}

printf("NULL\n");

}

// Function to get the middle node of the linked list

Node\* getMiddle(Node\* head) {

if (head == NULL) {

return head;

}

Node\* slow = head;

Node\* fast = head;

while (fast->next && fast->next->next) {

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

// Function to merge two sorted linked lists

Node\* merge(Node\* head1, Node\* head2) {

if (head1 == NULL) {

return head2;

}

if (head2 == NULL) {

return head1;

}

if (head1->data <= head2->data) {

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

return head1;

} else {

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

return head2;

}

}

// Function to perform merge sort on the linked list

Node\* mergeSort(Node\* head) {

if (head == NULL || head->next == NULL) {

return head;

}

Node\* mid = getMiddle(head);

Node\* midNext = mid->next;

mid->next = NULL;

Node\* left = mergeSort(head);

Node\* right = mergeSort(midNext);

Node\* sortedList = merge(left, right);

return sortedList;

}

int main() {

Node\* head = NULL;

insertNode(&head, 5);

insertNode(&head, 2);

insertNode(&head, 8);

insertNode(&head, 3);

insertNode(&head, 1);

insertNode(&head, 6);

insertNode(&head, 4);

printf("Original Linked List: ");

printList(head);

head = mergeSort(head);

printf("Sorted Linked List: ");

printList(head);

    return 0;

}

1. **Divide a linked list into two halves:**

#include <stdio.h>

#include <stdlib.h>

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

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

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

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

newNode->next = \*head;

} else {

struct Node\* temp = \*head;

while (temp->next != \*head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

}

}

// Function to split the circular linked list into two halves

void splitCircularList(struct Node\* head, struct Node\*\* head1, struct Node\*\* head2) {

if (head == NULL) return;

// Use slow and fast pointers to find the middle

struct Node\* slow = head;

struct Node\* fast = head;

while (fast->next != head && fast->next->next != head) {

slow = slow->next;

fast = fast->next->next;

}

// If the list has an even number of nodes, move fast to the last node

if (fast->next->next == head) {

fast = fast->next;

}

// Set the head of the first half

\*head1 = head;

// Set the head of the second half

if (head->next != head) {

\*head2 = slow->next;

}

// Make the first half circular

slow->next = \*head1;

// Make the second half circular

fast->next = \*head2;

}

// Function to print a circular linked list

void printList(struct Node\* head) {

if (head == NULL) return;

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("HEAD\n");

}

int main() {

struct Node\* head = NULL;

struct Node\* head1 = NULL;

struct Node\* head2 = NULL;

// Create a circular linked list

insertEnd(&head, 1);

insertEnd(&head, 2);

insertEnd(&head, 3);

insertEnd(&head, 4);

insertEnd(&head, 5);

printf("Original Circular Linked List: ");

printList(head);

// Split the list into two halves

splitCircularList(head, &head1, &head2);

printf("First Half: ");

printList(head1);

printf("Second Half: ");

printList(head2);

    return 0;

}

1. **Check if two trees are mirrors of each other**

#include <stdio.h>

#include <stdlib.h>

// Structure for a tree node

struct TreeNode {

int val;

struct TreeNode \*left, \*right;

};

// Function to create a new node

struct TreeNode\* newNode(int val) {

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

node->val = val;

node->left = node->right = NULL;

return node;

}

// Function to check if two trees are mirrors

int areMirror(struct TreeNode\* root1, struct TreeNode\* root2) {

if (root1 == NULL && root2 == NULL) return 1;

if (root1 == NULL || root2 == NULL) return 0;

return (root1->val == root2->val) &&

areMirror(root1->left, root2->right) &&

areMirror(root1->right, root2->left);

}

// Driver code

int main() {

struct TreeNode\* root1 = newNode(1);

root1->left = newNode(2);

root1->right = newNode(3);

struct TreeNode\* root2 = newNode(1);

root2->left = newNode(3);

root2->right = newNode(2);

printf("%s\n", areMirror(root1, root2) ? "true" : "false"); // Output: true

    return 0;

}

1. **Check whether BST Contains Dead End.**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h> // For INT\_MAX

// Structure for BST Node

struct Node {

int data;

struct Node \*left, \*right;

};

// Function to create a new node

struct Node\* newNode(int data) {

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

node->data = data;

node->left = node->right = NULL;

return node;

}

// Function to insert a node into BST

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

if (root == NULL)

return newNode(key);

if (key < root->data)

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

else

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

return root;

}

// Function to check if BST contains a dead end

int checkDeadEnd(struct Node\* root, int min, int max) {

if (root == NULL)

return 0;

// If min == max, a dead end is found

if (min == max)

return 1;

// Recursively check left and right subtrees

return checkDeadEnd(root->left, min, root->data - 1) ||

checkDeadEnd(root->right, root->data + 1, max);

}

// Wrapper function to check for dead end

int containsDeadEnd(struct Node\* root) {

return checkDeadEnd(root, 1, INT\_MAX);

}

// Driver Code

int main() {

struct Node\* root = NULL;

// Insert nodes into BST

root = insert(root, 8);

root = insert(root, 5);

root = insert(root, 2);

root = insert(root, 3);

root = insert(root, 7);

root = insert(root, 11);

// Check if BST contains a dead end

if (containsDeadEnd(root))

printf("BST contains a dead end\n");

else

printf("BST does not contain a dead end\n");

return 0;

}