**Merge sort**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* next;

} Node;

Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

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

Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

return;

}

Node\* current = \*head;

while (current->next != NULL)

current = current->next;

current->next = newNode;

}

void printList(Node\* head) {

while (head != NULL) {

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

head = head->next;

}

printf("NULL\n");

}

Node\* getMiddle(Node\* head) {

if (head == NULL) return head;

Node\* slow = head;

Node\* fast = head->next;

while (fast && fast->next) {

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

Node\* merge(Node\* left, Node\* right) {

if (!left) return right;

if (!right) return left;

Node\* result;

if (left->data <= right->data) {

result = left;

result->next = merge(left->next, right);

} else {

result = right;

result->next = merge(left, right->next);

}

return result;

}

Node\* mergeSort(Node\* head) {

if (!head || !head->next)

return head;

Node\* middle = getMiddle(head);

Node\* rightHalf = middle->next;

middle->next = NULL;

Node\* leftSorted = mergeSort(head);

Node\* rightSorted = mergeSort(rightHalf);

return merge(leftSorted, rightSorted);

}

int main() {

Node\* head = NULL;

insert(&head, 5);

insert(&head, 2);

insert(&head, 8);

insert(&head, 3);

insert(&head, 1);

printf("Original List: ");

printList(head);

head = mergeSort(head);

printf("Sorted List: ");

printList(head);

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Divide a linked list into two halves**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

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;

}

}

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

if (head == NULL) return;

struct Node\* slow = head;

struct Node\* fast = head;

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

slow = slow->next;

fast = fast->next->next;

}

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

fast = fast->next;

}

\*head1 = head;

if (head->next != head) {

\*head2 = slow->next;

}

slow->next = \*head1;

fast->next = \*head2;

}

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;

insertEnd(&head, 1);

insertEnd(&head, 2);

insertEnd(&head, 3);

insertEnd(&head, 4);

insertEnd(&head, 5);

printf("Original Circular Linked List: ");

printList(head);

splitCircularList(head, &head1, &head2);

printf("First Half: ");

printList(head1);

printf("Second Half: ");

printList(head2);

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int val;

struct TreeNode \*left, \*right;

};

struct TreeNode\* newNode(int val) {

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

node->val = val;

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

return node;

}

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);

}

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");

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Check whether BST Contains Dead End.**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <limits.h>**

**struct Node {**

**int data;**

**struct Node \*left, \*right;**

**};**

**struct Node\* newNode(int data) {**

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

**node->data = data;**

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

**return node;**

**}**

**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;**

**}**

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

**if (root == NULL)**

**return 0;**

**if (min == max)**

**return 1;**

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

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

**}**

**int containsDeadEnd(struct Node\* root) {**

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

**}**

**int main() {**

**struct Node\* root = NULL;**

**root = insert(root, 8);**

**root = insert(root, 5);**

**root = insert(root, 2);**

**root = insert(root, 3);**

**root = insert(root, 7);**

**root = insert(root, 11);**

**if (containsDeadEnd(root))**

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

**else**

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

**return 0;**

**}**