PALINDROME\_RECURSION

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

/\* Link list node \*/

struct node {

char data;

struct node\* next;

};

// Initial parameters to this function are &head and head

bool isPalindromeUtil(struct node\*\* left, struct node\* right)

{

/\* stop recursion when right becomes NULL \*/

if (right == NULL)

return true;

/\* If sub-list is not palindrome then no need to

check for current left and right, return false \*/

bool isp = isPalindromeUtil(left, right->next);

if (isp == false)

return false;

/\* Check values at current left and right \*/

bool isp1 = (right->data == (\*left)->data);

/\* Move left to next node \*/

\*left = (\*left)->next;

return isp1;

}

// A wrapper over isPalindromeUtil()

bool isPalindrome(struct node\* head)

{

isPalindromeUtil(&head, head);

}

/\* Push a node to linked list. Note that this function

changes the head \*/

void push(struct node\*\* head\_ref, char new\_data)

{

/\* allocate node \*/

struct node\* new\_node = (struct node\*)malloc(sizeof(struct node));

/\* put in the data \*/

new\_node->data = new\_data;

/\* link the old list off the new node \*/

new\_node->next = (\*head\_ref);

/\* move the head to pochar to the new node \*/

(\*head\_ref) = new\_node;

}

// A utility function to print a given linked list

void printList(struct node\* ptr)

{

while (ptr != NULL) {

printf("%c->", ptr->data);

ptr = ptr->next;

}

printf("NULL\n");

}

/\* Driver program to test above function\*/

int main()

{

/\* Start with the empty list \*/

struct node\* head = NULL;

char str[] = "abacaba";

int i;

for (i = 0; str[i] != '\0'; i++) {

push(&head, str[i]);

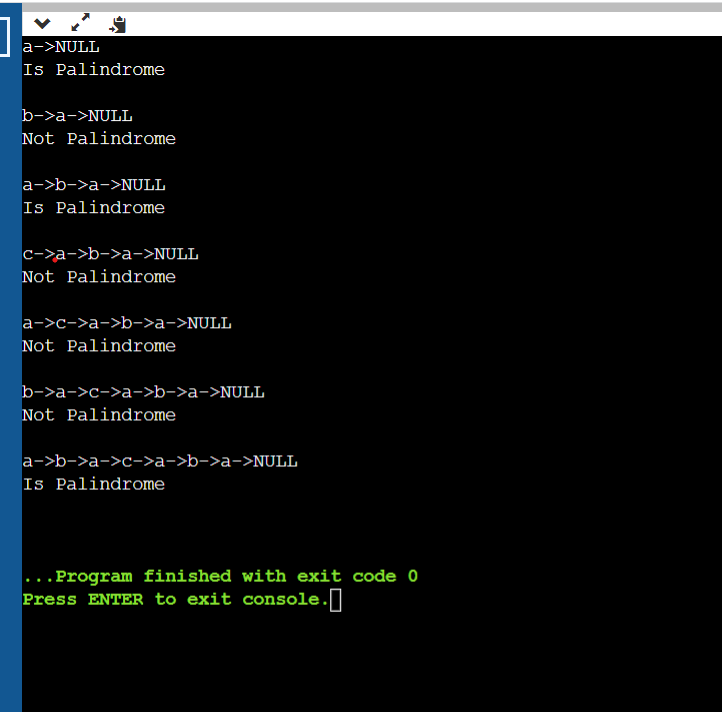
printList(head);

isPalindrome(head) ? printf("Is Palindrome\n\n") : printf("Not Palindrome\n\n");

}

return 0;

}



ALTERNATING HIGH\_LOW VALUES

#include <stdio.h>

#include <stdlib.h>

// A Linked List Node

struct Node

{

int data;

struct Node\* next;

};

// Helper function to return new linked list node from the heap

struct Node\* newNode(int key, struct Node \*ptr)

{

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

node->data = key;

node->next = ptr;

return node;

}

// Helper function to create a new node with the given data and

// pushes it onto the list's front

void push(struct Node\*\* head, int data)

{

// create a new linked list node from the heap

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

newNode->data = data;

newNode->next = \*head;

\*head = newNode;

}

// Helper function to print a given linked list

void printList(struct Node\* head)

{

struct Node\* ptr = head;

while (ptr)

{

printf("%d —> ", ptr->data);

ptr = ptr->next;

}

printf("null");

}

void swap(struct Node \*first, struct Node \*second)

{

int temp = first->data;

first->data = second->data;

second->data = temp;

}

// Rearrange the linked list so that it has alternating high, low values

void rearrange(struct Node \*head)

{

// empty list

if (head == NULL) {

return;

}

struct Node\* prev = head;

struct Node\* curr = head->next;

// start from the second node

while (curr)

{

// if the previous node is greater than the current node, swap their values

if (prev->data > curr->data) {

swap(prev, curr);

}

// if the next node is greater than the current node, swap their values

if (curr->next && curr->next->data > curr->data) {

swap(curr->next, curr);

}

// update `prev` and `curr` node

prev = curr->next;

if (!curr->next) {

break;

}

curr = curr->next->next;

}

}

int main(void)

{

// input keys

int keys[] = { 1, 2, 3, 4, 5, 6, 7, 8, 6 };

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

struct Node\* head = NULL;

for (int i = n - 1; i >= 0; i--) {

push(&head, keys[i]);

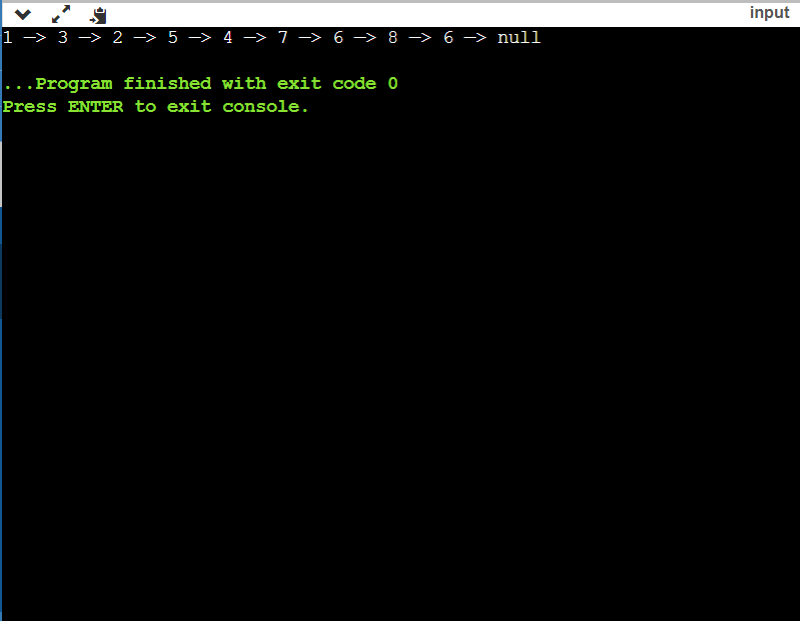
}

rearrange(head);

printList(head);

return 0;

}



HEIGHT OF A BINARY TREE:-

#include <stdio.h>

#include <stdlib.h>

// Utility function to find the maximum of two integers

int max(int x, int y) {

return (x > y) ? x : y;

}

// Data structure to store a binary tree node

struct Node

{

int data;

struct Node \*left, \*right;

};

// Utility function to allocate memory for a binary tree node

struct Node\* allocateNode(int data)

{

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

node->data = data;

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

return node;

}

// Recursive function to calculate the height of a binary tree with

// leaf nodes forming a circular doubly linked list

int height(struct Node\* node)

{

// base case: if the node is NULL

if (node == NULL) {

return 0;

}

// node is a leaf if its left's right and right's left

// are pointing to the node itself

if ((node->left && node->left->right == node) &&

(node->right && node->right->left == node)) {

return 1;

}

// recur for the left and right subtree and consider maximum depth

return 1 + max(height(node->left), height(node->right));

}

int main(void)

{

struct Node\* root = NULL;

// construct the tree

root = allocateNode(1);

root->left = allocateNode(2);

root->right = allocateNode(3);

root->left->left = allocateNode(4);

root->left->right = allocateNode(5); // leaf node

root->right->right = allocateNode(6); // leaf node

root->left->left->left = allocateNode(7); // leaf node

// construct a circular doubly linked list from leaves

struct Node\* first = root->left->left->left;

struct Node\* second = root->left->right;

struct Node\* third = root->right->right;

// set previous and next pointers of the linked list

// (left and right pointer of a binary tree node, respectively)

first->left = third;

first->right = second;

second->left = first;

second->right = third;

third->left = second;

third->right = first;

printf("The height of the binary tree is %d", height(root));

return 0;

}

