**list of Linked List questions**

**Basic Linked List Questions**

1. Write a program to create a singly linked list and display its elements.

Solution:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

void displayList(Node\* head) {

Node\* temp = head;

while (temp) {

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

temp = temp->next;

}

printf("NULL\n");

}

2. Insert a node at the beginning, middle, and end of a singly linked list.

// Function to insert at the beginning

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

Node\* newNode = createNode(data);

newNode->next = \*head;

\*head = newNode;

}

// Function to insert at the end

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

Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

return;

}

Node\* temp = \*head;

while (temp->next) {

temp = temp->next;

}

temp->next = newNode;

}

// Function to insert at a specific position

void insertAtPosition(Node\*\* head, int data, int position) {

if (position == 1) {

insertAtBeginning(head, data);

return;

}

Node\* newNode = createNode(data);

Node\* temp = \*head;

for (int i = 1; i < position - 1 && temp; i++) {

temp = temp->next;

}

if (temp == NULL) {

printf("Position out of bounds.\n");

return;

}

newNode->next = temp->next;

temp->next = newNode;

}

3 .Delete a node from the beginning, middle, and end of a singly linked list.

// Function to delete from the beginning

void deleteFromBeginning(Node\*\* head) {

if (\*head == NULL) {

printf("List is empty.\n");

return;

}

Node\* temp = \*head;

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

free(temp);

}

// Function to delete from the end

void deleteFromEnd(Node\*\* head) {

if (\*head == NULL) {

printf("List is empty.\n");

return;

}

if ((\*head)->next == NULL) {

free(\*head);

\*head = NULL;

return;

}

Node\* temp = \*head;

while (temp->next->next) {

temp = temp->next;

}

free(temp->next);

temp->next = NULL;

}

// Function to delete from a specific position

void deleteFromPosition(Node\*\* head, int position) {

if (position == 1) {

deleteFromBeginning(head);

return;

}

Node\* temp = \*head;

for (int i = 1; i < position - 1 && temp; i++) {

temp = temp->next;

}

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

printf("Position out of bounds.\n");

return;

}

Node\* nodeToDelete = temp->next;

temp->next = nodeToDelete->next;

free(nodeToDelete);

}

4. Count the number of nodes in a linked list.

int countNodes(Node\* head) {

int count = 0;

Node\* temp = head;

while (temp) {

count++;

temp = temp->next;

}

return count;

}

5. Search for a given value in a linked list.

int searchValue(Node\* head, int value) {

Node\* temp = head;

int position = 1;

while (temp) {

if (temp->data == value) {

return position;

}

temp = temp->next;

position++;

}

return -1;

}

6. Reverse a singly linked list.

void reverseList(Node\*\* head) {

Node \*prev = NULL, \*current = \*head, \*next = NULL;

while (current) {

next = current->next;

current->next = prev;

prev = current;

current = next;

}

\*head = prev;

}

7. Merge two sorted linked lists into a single sorted list.

Node\* mergeSortedLists(Node\* l1, Node\* l2) {

Node dummy;

Node\* tail = &dummy;

dummy.next = NULL;

while (l1 && l2) {

if (l1->data < l2->data) {

tail->next = l1;

l1 = l1->next;

} else {

tail->next = l2;

l2 = l2->next;

}

tail = tail->next;

}

tail->next = (l1) ? l1 : l2;

return dummy.next;

}

// Main function to test all operations

int main() {

Node\* head = NULL;

Node\* list2 = NULL;

// Insertions

insertAtEnd(&head, 10);

insertAtEnd(&head, 20);

insertAtEnd(&head, 30);

printf("List after insertions: ");

displayList(head);

// Insert at beginning, middle, and end

insertAtBeginning(&head, 5);

insertAtPosition(&head, 15, 3);

insertAtEnd(&head, 35);

printf("List after specific insertions: ");

displayList(head);

// Deletions

deleteFromBeginning(&head);

deleteFromPosition(&head, 3);

deleteFromEnd(&head);

printf("List after deletions: ");

displayList(head);

// Count nodes

printf("Number of nodes: %d\n", countNodes(head));

// Search for a value

int value = 20;

int position = searchValue(head, value);

if (position != -1) {

printf("Value %d found at position %d\n", value, position);

} else {

printf("Value %d not found in the list\n", value);

}

// Reverse the list

reverseList(&head);

printf("Reversed list: ");

displayList(head);

// Merge two sorted lists

insertAtEnd(&list2, 25);

insertAtEnd(&list2, 35);

insertAtEnd(&list2, 45);

Node\* mergedList = mergeSortedLists(head, list2);

printf("Merged sorted list: ");

displayList(mergedList);

return 0;

}

Med:

Question-1

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a linked list node

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 append a node at the end

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

return;

}

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

// Function to find the nth node from the end

void findNthFromEnd(struct Node\* head, int n) {

struct Node\* first = head;

struct Node\* second = head;

int count = 0;

// Move the first pointer n steps ahead

while (count < n) {

if (first == NULL) {

printf("The list has fewer than %d nodes.\n", n);

return;

}

first = first->next;

count++;

}

// Move both pointers one step at a time until first reaches the end

while (first != NULL) {

first = first->next;

second = second->next;

}

// The second pointer now points to the nth node from the end

printf("The %dth node from the end is: %d\n", n, second->data);

}

// // Function to display the linked list

// void displayList(struct Node\* head) {

// struct Node\* temp = head;

// while (temp != NULL) {

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

// temp = temp->next;

// }

// printf("NULL\n");

// }

// Main function

int main() {

struct Node\* head = NULL;

int n;

// Create a linked list

appendNode(&head, 10);

appendNode(&head, 20);

appendNode(&head, 30);

appendNode(&head, 40);

appendNode(&head, 50);

// Find the nth node from the end

printf("Enter the position from the end: ");

scanf("%d", &n);

findNthFromEnd(head, n);

return 0;

}

med -2

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a linked list node

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 append a node at the end

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

return;

}

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

// Function to detect a loop in the linked list

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

struct Node \*slow = head, \*fast = head;

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

slow = slow->next;

fast = fast->next->next;

if (slow == fast) { // Loop detected

return slow;

}

}

return NULL; // No loop

}

// Function to remove a loop from the linked list

void removeLoop(struct Node\* head, struct Node\* loopNode) {

struct Node\* ptr1 = head;

struct Node\* ptr2;

// Find the node where the loop starts

while (1) {

ptr2 = loopNode;

while (ptr2->next != loopNode && ptr2->next != ptr1) {

ptr2 = ptr2->next;

}

if (ptr2->next == ptr1) {

break; // Found the start of the loop

}

ptr1 = ptr1->next;

}

// Break the loop

ptr2->next = NULL;

}

// Function to display the linked list

void displayList(struct Node\* head) {

struct Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Main function

int main() {

struct Node\* head = NULL;

// Create a linked list

appendNode(&head, 10);

appendNode(&head, 20);

appendNode(&head, 30);

appendNode(&head, 40);

appendNode(&head, 50);

// Manually create a loop for testing

head->next->next->next->next->next = head->next; // Creates a loop at node 20

// Detect and remove the loop

struct Node\* loopNode = detectLoop(head);

if (loopNode) {

printf("Loop detected.\n");

removeLoop(head, loopNode);

printf("Loop removed.\n");

} else {

printf("No loop detected.\n");

}

// Display the linked list after removing the loop

printf("Linked List after loop removal: ");

displayList(head);

return 0;

}

med-3 (palindrome)

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Define the structure for a linked list node

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 append a node at the end

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

return;

}

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

// Function to reverse a linked list

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

struct Node\* prev = NULL;

struct Node\* current = head;

struct Node\* next = NULL;

while (current != NULL) {

next = current->next;

current->next = prev;

prev = current;

current = next;

}

return prev;

}

// Function to check if a linked list is a palindrome

bool isPalindrome(struct Node\* head) {

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

return true; // A single node or empty list is a palindrome

}

// Find the middle of the list

struct Node \*slow = head, \*fast = head;

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

slow = slow->next;

fast = fast->next->next;

}

// Reverse the second half of the list

struct Node\* secondHalf = reverseList(slow);

struct Node\* firstHalf = head;

struct Node\* secondHalfCopy = secondHalf; // Keep a copy for restoration

// Compare the first and second halves

bool isPalin = true;

while (secondHalf != NULL) {

if (firstHalf->data != secondHalf->data) {

isPalin = false;

break;

}

firstHalf = firstHalf->next;

secondHalf = secondHalf->next;

}

// Restore the list to its original form

reverseList(secondHalfCopy);

return isPalin;

}

// Function to display the linked list

void displayList(struct Node\* head) {

struct Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Main function

int main() {

struct Node\* head = NULL;

// Create a linked list

appendNode(&head, 1);

appendNode(&head, 2);

appendNode(&head, 3);

appendNode(&head, 2);

appendNode(&head, 1);

printf("Linked List: ");

displayList(head);

// Check if the list is a palindrome

if (isPalindrome(head)) {

printf("The linked list is a palindrome.\n");

} else {

printf("The linked list is not a palindrome.\n");

}

return 0;

}

med-4 (remove duplicates)

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define HASH\_SIZE 1000

// Define the structure for a linked list node

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 append a node at the end

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

return;

}

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

// Hashing utility: Initialize a boolean array to track visited nodes

bool hashTable[HASH\_SIZE];

// Function to remove duplicate nodes

void removeDuplicates(struct Node\* head) {

struct Node\* current = head;

struct Node\* prev = NULL;

// Initialize the hash table

for (int i = 0; i < HASH\_SIZE; i++) {

hashTable[i] = false;

}

// Traverse the linked list

while (current != NULL) {

int value = current->data;

// Check if the value is already in the hash table

if (hashTable[value]) {

// Duplicate found, remove the current node

prev->next = current->next;

free(current);

} else {

// Mark the value as visited

hashTable[value] = true;

prev = current;

}

current = prev->next;

}

}

// Function to display the linked list

void displayList(struct Node\* head) {

struct Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Main function

int main() {

struct Node\* head = NULL;

// Create a linked list

appendNode(&head, 10);

appendNode(&head, 20);

appendNode(&head, 10);

appendNode(&head, 30);

appendNode(&head, 20);

appendNode(&head, 40);

printf("Original Linked List: ");

displayList(head);

// Remove duplicates

removeDuplicates(head);

printf("Linked List after removing duplicates: ");

displayList(head);

return 0;

}

med -5

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a linked list node

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 get the length of a linked list

int getLength(struct Node\* head) {

int length = 0;

while (head != NULL) {

length++;

head = head->next;

}

return length;

}

// Function to find the intersection point of two linked lists

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

int len1 = getLength(head1);

int len2 = getLength(head2);

// Align the starts of both lists

struct Node\* longer = len1 > len2 ? head1 : head2;

struct Node\* shorter = len1 > len2 ? head2 : head1;

int diff = abs(len1 - len2);

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

longer = longer->next;

}

// Traverse both lists to find the intersection

while (longer != NULL && shorter != NULL) {

if (longer == shorter) {

return longer; // Intersection point found

}

longer = longer->next;

shorter = shorter->next;

}

return NULL; // No intersection

}

// Function to display a linked list

void displayList(struct Node\* head) {

while (head != NULL) {

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

head = head->next;

}

printf("NULL\n");

}

// Main function

int main() {

// Create two linked lists with an intersection

struct Node\* head1 = createNode(10);

head1->next = createNode(20);

head1->next->next = createNode(30);

struct Node\* head2 = createNode(5);

head2->next = createNode(15);

head2->next->next = head1->next->next; // Intersection at node with data 30

printf("List 1: ");

displayList(head1);

printf("List 2: ");

displayList(head2);

// Find the intersection point

struct Node\* intersection = findIntersection(head1, head2);

if (intersection) {

printf("Intersection point is at node with data: %d\n", intersection->data);

} else {

printf("No intersection point.\n");

}

return 0;

}

Med – last -3

#include <stdio.h>

#include <stdlib.h>

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 display a circular linked list

void displayCircularList(struct Node\* head) {

if (!head) return;

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("(back to head)\n");

}

// Function to make a linked list circular

void makeCircular(struct Node\* head) {

struct Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = head;

}

// 1. Split a circular linked list into two halves

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

struct Node \*slow = head, \*fast = head;

// Find the midpoint using slow and fast pointers

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

slow = slow->next;

fast = fast->next->next;

}

// Set heads for the two halves

\*head1 = head;

\*head2 = slow->next;

// Break the first half's circular connection

slow->next = \*head1;

// Find the end of the second half and close the loop

struct Node\* temp = \*head2;

while (temp->next != head) {

temp = temp->next;

}

temp->next = \*head2;

}

// 2. Rotate a linked list by k positions (left or right)

void rotateList(struct Node\*\* head, int k, int isLeft) {

if (!\*head || k == 0) return;

struct Node\* current = \*head;

int count = 1;

// Find the kth node

while (count < k && current->next) {

current = current->next;

count++;

}

// If k >= length of the list, no rotation is needed

if (!current->next) return;

struct Node\* kthNode = current;

// Find the last node

while (current->next) {

current = current->next;

}

// Perform rotation

if (isLeft) {

// Left rotation

current->next = \*head;

\*head = kthNode->next;

kthNode->next = NULL;

} else {

// Right rotation

struct Node\* temp = \*head;

while (temp->next != kthNode) {

temp = temp->next;

}

temp->next = NULL;

current->next = \*head;

\*head = kthNode;

}

}

// 3. Swap two nodes in a linked list without swapping data

void swapNodes(struct Node\*\* head, int x, int y) {

if (x == y) return;

struct Node \*prevX = NULL, \*currX = \*head;

while (currX && currX->data != x) {

prevX = currX;

currX = currX->next;

}

struct Node \*prevY = NULL, \*currY = \*head;

while (currY && currY->data != y) {

prevY = currY;

currY = currY->next;

}

// If either x or y is not present, do nothing

if (!currX || !currY) return;

// Update previous nodes' next pointers

if (prevX) prevX->next = currY;

else \*head = currY;

if (prevY) prevY->next = currX;

else \*head = currX;

// Swap the next pointers

struct Node\* temp = currY->next;

currY->next = currX->next;

currX->next = temp;

}

// Main function for testing

int main() {

// Create a circular linked list

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = createNode(4);

head->next

right rotation:

#include <stdio.h>

#include <stdlib.h>

// Definition of a Node

struct Node {

int data;

struct Node\* next;

};

// Function to rotate the list

void rotateList(struct Node\*\* head, int k, int isLeft) {

if (!\*head || k == 0) return; // No rotation needed if list is empty or k is 0

struct Node\* current = \*head;

int length = 1;

// Find the length of the list

while (current->next) {

current = current->next;

length++;

}

// Normalize k (handle cases where k >= length)

k = k % length;

if (k == 0) return; // No rotation needed if k is a multiple of the list's length

// Find the kth node for left or right rotation

if (!isLeft) {

k = length - k; // Convert right rotation to equivalent left rotation

}

// Reset current to the head

current = \*head;

int count = 1;

// Traverse to the kth node

while (count < k && current->next) {

current = current->next;

count++;

}

struct Node\* kthNode = current;

// Find the last node

while (current->next) {

current = current->next;

}

// Perform the rotation

current->next = \*head; // Connect the last node to the original head

\*head = kthNode->next; // Update head to the node after kthNode

kthNode->next = NULL; // Detach the first part of the list

}

// Utility functions

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

void printList(struct Node\* head) {

while (head) {

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

head = head->next;

}

printf("NULL\n");

}

// Main function

int main() {

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

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = createNode(4);

head->next->next->next->next = createNode(5);

printf("Original List: ");

printList(head);

// Rotate right by 2

rotateList(&head, 2, 0);

printf("After Right Rotation (k = 2): ");

printList(head);

return 0;

}

Adv – 2 (clone )

#include <stdio.h>

#include <stdlib.h>

// Definition of a Node

struct Node {

int data;

struct Node\* next;

struct Node\* random;

};

// 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;

newNode->random = NULL;

return newNode;

}

// Utility function to print a linked list

void printInterleavedList(struct Node\* head) {

struct Node\* current = head;

while (current) {

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

current = current->next;

}

printf("\n");

}

// Function to clone nodes and insert them next to the original nodes

void cloneNodes(struct Node\* head) {

struct Node\* current = head;

while (current) {

struct Node\* clone = createNode(current->data);

clone->next = current->next;

current->next = clone;

current = clone->next;

}

}

int main() {

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

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = createNode(4);

printf("Original List:\n");

printInterleavedList(head);

// Step 1: Clone nodes and insert them next to the original nodes

cloneNodes(head);

printf("After Adding Cloned Nodes:\n");

printInterleavedList(head);

return 0;

}

Adv- 1:

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a doubly linked list node

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

struct Node\* child;

};

// 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;

newNode->prev = NULL;

newNode->child = NULL;

return newNode;

}

// Function to flatten the multilevel doubly linked list

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

struct Node\* current = head;

while (current) {

if (current->child) {

// Flatten the child list

struct Node\* child = flatten(current->child);

// Save the next node

struct Node\* next = current->next;

// Attach the child list to the current node

current->next = child;

child->prev = current;

current->child = NULL;

// Traverse to the end of the flattened child list

struct Node\* tail = child;

while (tail->next) {

tail = tail->next;

}

// Reconnect the rest of the original list

if (next) {

tail->next = next;

next->prev = tail;

}

}

current = current->next;

}

return head;

}

// Function to display the doubly linked list

void displayList(struct Node\* head) {

while (head) {

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

if (head->child) {

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

}

head = head->next;

}

printf("\n");

}

// Main function

int main() {

// Create nodes for the list

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->prev = head;

head->next->next = createNode(3);

head->next->next->prev = head->next;

// Create a child list for the second node

head->next->child = createNode(4);

head->next->child->next = createNode(5);

head->next->child->next->prev = head->next->child;

// Display the original list

printf("Original list:\n");

displayList(head);

// Flatten the list

head = flatten(head);

// Display the flattened list

printf("Flattened list:\n");

displayList(head);

return 0;

}

Adv -3 :

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a linked list node

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 find the middle of the linked list

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

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

return head;

}

struct Node\* slow = head;

struct Node\* fast = head->next;

while (fast && fast->next) {

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

// Function to merge two sorted linked lists

struct Node\* mergeSortedLists(struct Node\* left, struct Node\* right) {

if (!left) return right;

if (!right) return left;

struct Node\* result = NULL;

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

result = left;

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

} else {

result = right;

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

}

return result;

}

// Function to sort the linked list using merge sort

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

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

return head;

}

// Split the list into two halves

struct Node\* middle = getMiddle(head);

struct Node\* nextHalf = middle->next;

middle->next = NULL;

// Recursively sort both halves

struct Node\* left = mergeSort(head);

struct Node\* right = mergeSort(nextHalf);

// Merge the two sorted halves

return mergeSortedLists(left, right);

}

// Function to print the linked list

void printList(struct Node\* head) {

while (head) {

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

head = head->next;

}

printf("NULL\n");

}

// Main function to demonstrate merge sort on a linked list

int main() {

// Create a sample linked list

struct Node\* head = createNode(4);

head->next = createNode(2);

head->next->next = createNode(1);

head->next->next->next = createNode(3);

printf("Original list:\n");

printList(head);

// Sort the linked list

head = mergeSort(head);

printf("Sorted list:\n");

printList(head);

return 0;

}

Adv -4 :

adv -4

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a linked list node

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 add a new node at the end of the list

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

return;

}

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

// Function to print the linked list

void printList(struct Node\* head) {

while (head) {

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

head = head->next;

}

printf("NULL\n");

}

// Function to add two numbers represented by linked lists

struct Node\* addTwoLists(struct Node\* l1, struct Node\* l2) {

struct Node\* result = NULL; // Resultant list

struct Node\*\* current = &result;

int carry = 0;

// Traverse both lists

while (l1 || l2 || carry) {

int sum = carry;

if (l1) {

sum += l1->data;

l1 = l1->next;

}

if (l2) {

sum += l2->data;

l2 = l2->next;

}

// Calculate the digit to store and the carry

int digit = sum % 10;

carry = sum / 10;

// Append the digit to the result list

\*current = createNode(digit);

current = &((\*current)->next);

}

return result;

}

// Main function to demonstrate the addition of two linked lists

int main() {

struct Node\* num1 = NULL;

struct Node\* num2 = NULL;

// Create first number: 342 (represented as 2 -> 4 -> 3)

appendNode(&num1, 2);

appendNode(&num1, 4);

appendNode(&num1, 3);

// Create second number: 465 (represented as 5 -> 6 -> 4)

appendNode(&num2, 5);

appendNode(&num2, 6);

appendNode(&num2, 4);

printf("First number: ");

printList(num1);

printf("Second number: ");

printList(num2);

// Add the two numbers

struct Node\* result = addTwoLists(num1, num2);

printf("Sum: ");

printList(result);

return 0;

}

Adv-5 :

#include <stdio.h>

#include <stdlib.h>

// Definition of a Node

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 reverse `k` nodes in a linked list

struct Node\* reverseKGroup(struct Node\* head, int k) {

if (!head || k <= 1) return head;

struct Node \*current = head, \*prev = NULL, \*next = NULL;

int count = 0;

// Count the number of nodes in the current group

struct Node\* temp = head;

for (count = 0; temp && count < k; count++) {

temp = temp->next;

}

// If the number of nodes is less than `k`, return as is

if (count < k) return head;

// Reverse the first `k` nodes

count = 0;

while (current && count < k) {

next = current->next;

current->next = prev;

prev = current;

current = next;

count++;

}

// `head` is now the last node of the reversed group, connect it to the next group

if (next) {

head->next = reverseKGroup(next, k);

}

// Return the new head of the reversed group

return prev;

}

// Utility function to print a linked list

void printList(struct Node\* head) {

while (head) {

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

head = head->next;

}

printf("NULL\n");

}

// Function to create a linked list using a loop

struct Node\* createList(int n) {

struct Node\* head = createNode(1);

struct Node\* current = head;

for (int i = 2; i <= n; i++) {

current->next = createNode(i);

current = current->next;

}

return head;

}

int main() {

int n = 7; // Number of nodes

struct Node\* head = createList(n);

printf("Original List:\n");

printList(head);

int k = 3;

head = reverseKGroup(head, k);

printf("Reversed in Groups of %d:\n", k);

printList(head);

return 0;

}

Adv -6:

#include <stdio.h>

#include <stdlib.h>

// Definition for a singly linked list node.

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 detect a loop in the linked list and return the length of the loop

int detectLoopAndFindLength(struct Node\* head) {

if (!head) return 0;

struct Node\* slow = head;

struct Node\* fast = head;

// Step 1: Detect the loop using Floyd’s Cycle-Finding Algorithm

while (fast && fast->next) {

slow = slow->next; // Move slow by 1 step

fast = fast->next->next; // Move fast by 2 steps

if (slow == fast) {

// Loop detected

break;

}

}

// If no loop is found

if (slow != fast) {

return 0; // No loop

}

// Step 2: Find the length of the loop

int loopLength = 1;

struct Node\* temp = slow->next;

while (temp != slow) {

temp = temp->next;

loopLength++;

}

return loopLength;

}

// Function to create a loop for testing

void createLoop(struct Node\* head, int loopStartIndex) {

struct Node\* temp = head;

struct Node\* loopStartNode = NULL;

int index = 0;

while (temp->next) {

if (index == loopStartIndex) {

loopStartNode = temp;

}

temp = temp->next;

index++;

}

// Create the loop by linking the last node to the loopStartNode

if (loopStartNode) {

temp->next = loopStartNode;

}

}

// Function to print the linked list (for testing purposes)

void printList(struct Node\* head) {

struct Node\* temp = head;

while (temp) {

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

temp = temp->next;

}

printf("NULL\n");

}

int main() {

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

struct Node\* head = createNode(1);

head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = createNode(4);

head->next->next->next->next = createNode(5);

// Create a loop: 5 -> 3 (index 2)

createLoop(head, 2);

// Detect and find the length of the loop

int loopLength = detectLoopAndFindLength(head);

if (loopLength > 0) {

printf("Loop detected. Loop length: %d\n", loopLength);

} else {

printf("No loop detected.\n");

}

return 0;

}

DLL OPERATIONS:

#include <stdio.h>

#include <stdlib.h>

// Definition of the doubly linked list node

struct DllNode {

int data;

struct DllNode\* prev;

struct DllNode\* next;

};

// Definition of a binary tree node

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new node in the doubly linked list

struct DllNode\* createDllNode(int data) {

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

newNode->data = data;

newNode->prev = newNode->next = NULL;

return newNode;

}

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

void append(struct DllNode\*\* head, int data) {

struct DllNode\* newNode = createDllNode(data);

if (\*head == NULL) {

\*head = newNode;

return;

}

struct DllNode\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

// Function to delete a node from the doubly linked list

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

struct DllNode\* temp = \*head;

// If the head node itself contains the key

if (temp != NULL && temp->data == key) {

\*head = temp->next;

if (\*head != NULL) {

(\*head)->prev = NULL;

}

free(temp);

return;

}

// Search for the node to be deleted

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

temp = temp->next;

}

// If key not found in the list

if (temp == NULL) return;

// Unlink the node from the list

if (temp->next != NULL) {

temp->next->prev = temp->prev;

}

if (temp->prev != NULL) {

temp->prev->next = temp->next;

}

free(temp);

}

// Function to print the doubly linked list

void printDllList(struct DllNode\* head) {

struct DllNode\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Function to reverse a doubly linked list

void reverseDllList(struct DllNode\*\* head) {

if (\*head == NULL) return;

struct DllNode\* temp = NULL;

struct DllNode\* current = \*head;

// Reverse the list

while (current != NULL) {

temp = current->prev;

current->prev = current->next;

current->next = temp;

current = current->prev;

}

// If the list is not empty, update head to the new first node

if (temp != NULL) {

\*head = temp->prev;

}

}

// Function to find pairs in a doubly linked list whose sum equals a given value

void findPairsWithSum(struct DllNode\* head, int sum) {

struct DllNode\* start = head;

struct DllNode\* end = head;

// Move 'end' to the last node

while (end != NULL && end->next != NULL) {

end = end->next;

}

// Find pairs

while (start != end && end != NULL && start != NULL && start->prev != end) {

if (start->data + end->data == sum) {

printf("Pair: (%d, %d)\n", start->data, end->data);

start = start->next;

end = end->prev;

} else if (start->data + end->data < sum) {

start = start->next;

} else {

end = end->prev;

}

}

}

// Function to create a new binary tree node

struct TreeNode\* createTreeNode(int data) {

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

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// Helper function to convert binary tree to doubly linked list

void treeToDll(struct TreeNode\* root, struct DllNode\*\* head, struct DllNode\*\* tail) {

if (root == NULL) return;

// Convert the left subtree

treeToDll(root->left, head, tail);

// Convert the current tree node to a doubly linked list node

struct DllNode\* newNode = createDllNode(root->data);

// If the doubly linked list is empty, make it the head

if (\*head == NULL) {

\*head = newNode;

\*tail = newNode;

} else {

// Add to the end of the doubly linked list

(\*tail)->next = newNode;

newNode->prev = \*tail;

\*tail = newNode;

}

// Convert the right subtree

treeToDll(root->right, head, tail);

}

// Main function to test all functionalities

int main() {

struct DllNode\* head = NULL;

// Doubly Linked List operations

append(&head, 1);

append(&head, 2);

append(&head, 3);

append(&head, 4);

append(&head, 5);

printf("Original Doubly Linked List: ");

printDllList(head);

printf("After Reversing: ");

reverseDllList(&head);

printDllList(head);

printf("Finding pairs with sum 6: \n");

findPairsWithSum(head, 6);

// Binary Tree to Doubly Linked List

struct TreeNode\* root = createTreeNode(10);

root->left = createTreeNode(5);

root->right = createTreeNode(15);

root->left->left = createTreeNode(3);

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

struct DllNode\* treeHead = NULL;

struct DllNode\* treeTail = NULL;

treeToDll(root, &treeHead, &treeTail);

printf("Doubly Linked List from Binary Tree: ");

printDllList(treeHead);

// Deleting a node from Doubly Linked List

deleteNode(&head, 3);

printf("After Deleting 3: ");

printDllList(head);

return 0;

}

problem optimation

#include <stdio.h>

#include <stdlib.h>

// Definition for a node in a linked list

struct Node {

int data;

struct Node\* next;

};

// Stack and Queue Node definitions

struct StackNode {

int data;

struct StackNode\* next;

};

struct QueueNode {

int data;

struct QueueNode\* next;

};

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

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

}

// Function to print the linked list

void printList(struct Node\* head) {

struct Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Function to find the middle node of a linked list in a single traversal

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

if (head == NULL) return NULL;

struct Node \*slow = head, \*fast = head;

// Traverse the list with two pointers: slow and fast

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

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

// Function to delete all occurrences of a given key from the linked list

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

struct Node \*temp = \*head, \*prev = NULL;

// Check if the head node itself contains the key

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

\*head = temp->next;

free(temp);

temp = \*head;

}

// Traverse the list and delete nodes containing the key

while (temp != NULL) {

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

prev = temp;

temp = temp->next;

}

if (temp == NULL) return;

prev->next = temp->next;

free(temp);

temp = prev->next;

}

}

// Stack implementation using linked list

void push(struct StackNode\*\* top, int data) {

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

newNode->data = data;

newNode->next = \*top;

\*top = newNode;

}

int pop(struct StackNode\*\* top) {

if (\*top == NULL) return -1;

struct StackNode\* temp = \*top;

int data = temp->data;

\*top = temp->next;

free(temp);

return data;

}

// Queue implementation using linked list

void enqueue(struct QueueNode\*\* front, struct QueueNode\*\* rear, int data) {

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

newNode->data = data;

newNode->next = NULL;

if (\*rear == NULL) {

\*front = \*rear = newNode;

} else {

(\*rear)->next = newNode;

\*rear = newNode;

}

}

int dequeue(struct QueueNode\*\* front, struct QueueNode\*\* rear) {

if (\*front == NULL) return -1;

struct QueueNode\* temp = \*front;

int data = temp->data;

\*front = temp->next;

if (\*front == NULL) \*rear = NULL;

free(temp);

return data;

}

// Function to convert a singly linked list into a balanced BST

struct BSTNode {

int data;

struct BSTNode\* left;

struct BSTNode\* right;

};

struct BSTNode\* newBSTNode(int data) {

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

node->data = data;

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

return node;

}

// Function to find the length of the linked list

int getLength(struct Node\* head) {

int length = 0;

while (head != NULL) {

length++;

head = head->next;

}

return length;

}

// Function to convert linked list to a balanced BST

struct BSTNode\* sortedListToBSTHelper(struct Node\*\* head, int n) {

if (n <= 0) return NULL;

struct BSTNode\* left = sortedListToBSTHelper(head, n / 2);

struct BSTNode\* root = newBSTNode((\*head)->data);

root->left = left;

\*head = (\*head)->next; // Move to the next node

root->right = sortedListToBSTHelper(head, n - n / 2 - 1);

return root;

}

// Main function

int main() {

struct Node\* head = NULL;

// Creating the linked list: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8

append(&head, 1);

append(&head, 2);

append(&head, 3);

append(&head, 4);

append(&head, 5);

append(&head, 6);

append(&head, 7);

append(&head, 8);

printf("Original List: ");

printList(head);

// Find the middle node

struct Node\* middle = findMiddle(head);

if (middle != NULL) {

printf("Middle Node: %d\n", middle->data);

}

// Delete all occurrences of key 3

printf("Deleting all occurrences of 3...\n");

deleteKey(&head, 3);

printList(head);

// Stack operations

struct StackNode\* stackTop = NULL;

push(&stackTop, 10);

push(&stackTop, 20);

printf("Popped from stack: %d\n", pop(&stackTop));

// Queue operations

struct QueueNode\* front = NULL, \*rear = NULL;

enqueue(&front, &rear, 30);

enqueue(&front, &rear, 40);

printf("Dequeued from queue: %d\n", dequeue(&front, &rear));

// Convert linked list to a balanced BST

int len = getLength(head);

struct BSTNode\* bstRoot = sortedListToBSTHelper(&head, len);

printf("Balanced BST root data: %d\n", bstRoot->data);

return 0;

}

coding challenge -1

#include <stdio.h>

#include <stdlib.h>

// Definition for a node in a 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 append a new node to the list

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

}

// Function to find the length of the linked list

int getLength(struct Node\* head) {

int length = 0;

struct Node\* temp = head;

while (temp != NULL) {

length++;

temp = temp->next;

}

return length;

}

// Function to split the linked list into k parts

struct Node\*\* splitList(struct Node\* head, int k, int\* returnSize) {

int length = getLength(head);

int partSize = length / k; // base size of each part

int extraNodes = length % k; // number of parts that will have one extra node

struct Node\*\* result = (struct Node\*)malloc(k \* sizeof(struct Node));

struct Node\* current = head;

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

result[i] = current;

// Determine the size of this part (either partSize or partSize+1)

int currentPartSize = partSize + (i < extraNodes ? 1 : 0);

// Traverse to the last node of this part

for (int j = 1; j < currentPartSize; j++) {

if (current != NULL) {

current = current->next;

}

}

// If there is a node in this part, cut the link to the next part

if (current != NULL) {

struct Node\* temp = current->next;

current->next = NULL;

current = temp;

}

}

\*returnSize = k;

return result;

}

// Function to print the list of linked list parts

void printParts(struct Node\*\* parts, int k) {

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

printf("Part %d: ", i + 1);

struct Node\* temp = parts[i];

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

}

// Main function to test the program

int main() {

struct Node\* head = NULL;

int k = 3; // Number of parts to split the list into

int returnSize = 0;

// Creating a linked list: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9

append(&head, 1);

append(&head, 2);

append(&head, 3);

append(&head, 4);

append(&head, 5);

append(&head, 6);

append(&head, 7);

append(&head, 8);

append(&head, 9);

// Splitting the list into k parts

struct Node\*\* parts = splitList(head, k, &returnSize);

// Printing the resulting parts

printParts(parts, returnSize);

// Free allocated memory (clean-up)

free(parts);

return 0;

}

CLL OPRATIONS:

#include <stdio.h>

#include <stdlib.h>

// Definition of the circular linked list node

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 = newNode; // Next points to itself (for circular structure)

return newNode;

}

// Function to append a new node to the circular linked list

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

struct Node\* newNode = createNode(data);

if (\*head == NULL) {

\*head = newNode; // If list is empty, new node becomes the head

} else {

struct Node\* temp = \*head;

// Traverse to the last node

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

temp = temp->next;

}

temp->next = newNode; // Point last node to the new node

newNode->next = \*head; // New node points to the head

}

}

// Function to print the circular linked list

void printCircularList(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");

}

// Function to check if a list is circular

int isCircular(struct Node\* head) {

if (head == NULL) return 0;

struct Node\* temp = head;

while (temp->next != NULL && temp->next != head) {

temp = temp->next;

}

return (temp->next == head);

}

// Function to delete the kth node from the circular linked list

void deleteKthNode(struct Node\*\* head, int k) {

if (\*head == NULL || k <= 0) return;

struct Node\* temp = \*head;

struct Node\* prev = NULL;

// If the node to delete is the head node

if (k == 1) {

// If there's only one node in the list

if (temp->next == \*head) {

free(temp);

\*head = NULL;

return;

}

// Traverse to the last node and update its next pointer

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

temp = temp->next;

}

struct Node\* toDelete = \*head;

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

temp->next = \*head; // Last node points to the new head

free(toDelete);

return;

}

// Traverse the list to find the kth node

for (int i = 1; temp != NULL && i < k; i++) {

prev = temp;

temp = temp->next;

}

// If k is greater than the number of nodes

if (temp == NULL) return;

prev->next = temp->next; // Remove the kth node

free(temp);

}

// Main function to test all functionalities

int main() {

struct Node\* head = NULL;

// Circular Linked List Operations

append(&head, 1);

append(&head, 2);

append(&head, 3);

append(&head, 4);

append(&head, 5);

printf("Original Circular Linked List: ");

printCircularList(head);

// Check if the list is circular

if (isCircular(head)) {

printf("The list is circular.\n");

} else {

printf("The list is not circular.\n");

}

// Delete the 3rd node

deleteKthNode(&head, 3);

printf("After deleting 3rd node: ");

printCircularList(head);

// Check again if the list is circular

if (isCircular(head)) {

printf("The list is circular.\n");

} else {

printf("The list is not circular.\n");

}

return 0;

}