for a given array , write a program to print all of its subarrays

#include <stdio.h>

void printSubarrays(int arr[], int n) {

for (int i = 0; i < n; i++)

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

for (int k = i; k <= j; k++) printf("%d ", arr[k]);

printf("\n");

}

}

int main() {

int arr[] = {1, 2, 3}; // example array

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

printSubarrays(arr, n);

return 0;

}

for a given array, write a program to print all of its subsequences

#include <stdio.h>

void printSubsequences(int arr[], int n) {

int total = 1 << n;

for (int i = 1; i < total; i++) {

for (int j = 0; j < n; j++)

if (i & (1 << j)) printf("%d ", arr[j]);

printf("\n");

}

}

int main() {

int arr[] = {1, 2, 3}; // example array

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

printSubsequences(arr, n);

return 0;

}

Write a program to implement Binary search.

#include <stdio.h>

int binarySearch(int arr[], int size, int key) {

int start = 0;

int end = size - 1;

int mid = (start + end) / 2;

while (start <= end) {

if (arr[mid] == key) {

return mid;

}

if (key > arr[mid]) {

start = mid + 1;

} else {

end = mid - 1;

}

mid = (start + end) / 2;

}

return -1;

}

int main() {

int arr[5] = {2, 4, 6, 8, 10};

int key = 10, size = 5;

int var = binarySearch(arr, size, key);

printf("Key element present at index %d", var);

return 0;

}

A sorted array is rotated clockwise at arbitrary point. Ask user to enter a no which is need to be found. Apply binary search to find that number. If number is found return its index

#include <stdio.h>

int rotatedBinarySearch(int arr[], int size, int key) {

int start = 0;

int end = size - 1;

while (start <= end) {

int mid = (start + end) / 2;

// Check if the middle element is the key

if (arr[mid] == key) {

return mid;

}

// Determine which part is sorted

if (arr[start] <= arr[mid]) {

// Left half is sorted

if (key >= arr[start] && key < arr[mid]) {

end = mid - 1; // Search in the left sorted half

} else {

start = mid + 1; // Search in the right unsorted half

}

} else {

// Right half is sorted

if (key > arr[mid] && key <= arr[end]) {

start = mid + 1; // Search in the right sorted half

} else {

end = mid - 1; // Search in the left unsorted half

}

}

}

return -1; // Key not found

}

int main() {

// Example of a rotated sorted array

int arr[6] = {8, 10, 12, 2, 4, 6};

int key = 4;

int size = sizeof(arr) / sizeof(arr[0]);

int result = rotatedBinarySearch(arr, size, key);

if (result != -1) {

printf("Key element is present at index %d\n", result);

} else {

printf("Key element not found\n");

}

return 0;

}

Write a program to implement bubble sort.

#include <stdio.h>

void bubbleSort(int arr[], int n) {

for (int i = 0; i < n - 1; i++)

for (int j = 0; j < n - i - 1; j++)

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

int main() {

int arr[] = {5, 2, 9, 1, 5, 6}; // example array

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

bubbleSort(arr, n);

printf("Sorted array: ");

for (int i = 0; i < n; i++) printf("%d ", arr[i]);

printf("\n");

return 0;

}

Write a program to implement Selection sort.

#include <stdio.h>

void selectionSort(int arr[], int n) {

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

int minIndex = i;

for (int j = i + 1; j < n; j++)

if (arr[j] < arr[minIndex])

minIndex = j;

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

int main() {

int arr[] = {29, 10, 14, 37, 13}; // example array

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

selectionSort(arr, n);

printf("Sorted array: ");

for (int i = 0; i < n; i++) printf("%d ", arr[i]);

printf("\n");

return 0;

}

Write a program to implement Insertion sort.

#include <stdio.h>

void insertionSort(int arr[], int n) {

for (int i = 1; i < n; i++) {

int key = arr[i], j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6}; // example array

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

insertionSort(arr, n);

printf("Sorted array: ");

for (int i = 0; i < n; i++) printf("%d ", arr[i]);

printf("\n");

return 0;

}

Bub insertion and selection -menudriven

#include <stdio.h>

void bubbleSort(int arr[], int n) {

int i, j, temp;

for (i = 0; i < n - 1; i++) {

for (j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

void selectionSort(int arr[], int n) {

int i, j, minIndex, temp;

for (i = 0; i < n - 1; i++) {

minIndex = i;

for (j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

//After finding the smallest element, it is swapped with the first element of the unsorted portion (arr[i])

}

}

void insertionSort(int arr[], int n) {

// i for looping, key to hold the value of the current element being inserted, and j to track where to insert the element in the sorted part.

int i, key, j;

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

// Move elements that are greater than key

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

void displayArray(int arr[], int n) {

int i;

for (i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int n, i, choice;

printf("Enter the number of elements in the array: ");

scanf("%d", &n);

int arr[n], original[n];

printf("Enter the elements of the array:\n");

for (i = 0; i < n; i++) {

scanf("%d", &arr[i]);

original[i] = arr[i];

}

printf("\nChoose the sorting method:\n");

printf("1. Bubble Sort\n");

printf("2. Selection Sort\n");

printf("3. Insertion Sort\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

bubbleSort(arr, n);

printf("Array after Bubble Sort: ");

break;

case 2:

selectionSort(arr, n);

printf("Array after Selection Sort: ");

break;

case 3:

insertionSort(arr, n);

printf("Array after Insertion Sort: ");

break;

default:

printf("Invalid choice.\n");

return 0;

}

displayArray(arr, n);

return 0;

}

Write a Program to implement merge sort

#include <stdio.h>

//merge func takes 3 para start,end,arr.

void merge(int arr[], int start, int end) {

int mid = (start + end) >> 1;

//mid=(start+end)/2

int length1 = mid - start + 1;//1st half(including mid pt)

int length2 = end - mid;//2nd half

int first[length1];

int second[length2];

//2 arrs l1 and l2 created

int index1 = 0, index2 = 0, index = start;

//copy the elements from the original array into the first and second arrays.

for (int i = 0; i < length1; i++) {

first[i] = arr[start + i];

}

for (int i = 0; i < length2; i++) {

second[i] = arr[mid + 1 + i];

}

//while merges above 2 arrs back to og arr[]

while (index1 < length1 && index2 < length2) {

if (first[index1] < second[index2]) { //compares elements from both halves and places smaller in correct pos.

arr[index] = first[index1];

index++;

index1++;

} else {

arr[index] = second[index2];

index++;

index2++;

}

}

//merges the two sorted subarrays by comparing elements from arr first and second

//the remaining elem(if any) are copied to the og arr

while (index1 < length1) {

arr[index] = first[index1];

index++;

index1++;

}

while (index2 < length2) {

arr[index] = second[index2];

index++;

index2++;

}

}

//The sort function is a recursive function that divides the array into two halves until it reaches a base case where start >= end, meaning the subarray has one or no elements (already sorted).

//elements(already sorted)

void sort(int arr[], int start, int end) {

if (start >= end) {

return;

}

int mid = (start + end) >> 1;

//to split arr

// for left sort

sort(arr, start, mid);

// for right sort

sort(arr, mid + 1, end);

// merge array

merge(arr, start, end);

}

int main() {

int arr[5] = {8, 2, 6, 1, 9};

int n = 5;

int start = 0;

int end = n - 1;

sort(arr, start, end);

//sorted arr is printed

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

printf("%d ", arr[i]);

}

return 0;

}

Write a program to implement singly linked list and perform the following operations a) Insert at beginning b) Insert at end c) Insert after specified node d) delete at beginning e) delete at end f) delete after specified node g) display h) search an element

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* head = NULL;

void insertAtBeginning(int data) {

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

newNode->data = data;

newNode->next = head;

head = newNode;

}

void insertAtEnd(int data) {

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

newNode->data = data;

newNode->next = NULL;

if (head == NULL) {

head = newNode;

} else {

struct Node\* temp = head;

while (temp->next) temp = temp->next;

temp->next = newNode;

}

}

void insertAfter(int key, int data) {

struct Node\* temp = head;

while (temp && temp->data != key) temp = temp->next;

if (temp) {

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

newNode->data = data;

newNode->next = temp->next;

temp->next = newNode;

}

}

void deleteAtBeginning() {

if (head) {

struct Node\* temp = head;

head = head->next;

free(temp);

}

}

void deleteAtEnd() {

if (head) {

if (!head->next) {

free(head);

head = NULL;

} else {

struct Node\* temp = head;

while (temp->next->next) temp = temp->next;

free(temp->next);

temp->next = NULL;

}

}

}

void deleteAfter(int key) {

struct Node\* temp = head;

while (temp && temp->data != key) temp = temp->next;

if (temp && temp->next) {

struct Node\* toDelete = temp->next;

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

free(toDelete);

}

}

void display() {

struct Node\* temp = head;

if (temp == NULL) {

printf("NULL\n");

return;

}

while (temp) {

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

temp = temp->next;

}

printf("NULL\n");

}

void search(int key) {

struct Node\* temp = head;

int position = 1;

while (temp) {

if (temp->data == key) {

printf("Element %d found at position %d\n", key, position);

return;

}

temp = temp->next;

position++;

}

printf("Element %d not found\n", key);

}

int main() {

insertAtBeginning(10);

insertAtEnd(20);

insertAfter(10, 15);

display(); // Output: 10 -> 15 -> 20 -> NULL

deleteAtEnd();

deleteAfter(10);

deleteAtBeginning();

display(); // Output: NULL

insertAtEnd(30);

insertAtEnd(40);

search(30); // Output: Element 30 found at position 1

search(50); // Output: Element 50 not found

display(); // Output: 30 -> 40 -> NULL

return 0;

}

Write a program to generate a linked list. Using the head pointer of generated linked list, reverse it and update the head pointer accordingly.

Print such reversed linked list just by using head.

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* head = NULL;

void insert(int data) {

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

newNode->data = data;

newNode->next = head;

head = newNode;

}

void reverse() {

struct Node\* prev = NULL;

struct Node\* current = head;

struct Node\* next = NULL;

while (current) {

next = current->next;

current->next = prev;

prev = current;

current = next;

}

head = prev;

}

void display() {

struct Node\* temp = head;

while (temp) {

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

temp = temp->next;

}

printf("NULL\n");

}

int main() {

insert(10);

insert(20);

insert(30);

insert(40);

printf("Original list: ");

display(); // Output: 40 -> 30 -> 20 -> 10 -> NULL

reverse();

printf("Reversed list: ");

display(); // Output: 10 -> 20 -> 30 -> 40 -> NULL

return 0;

}

Write a program to implement doubly linked list and perform the following operations a) Insert at beginning b) Insert at end c)Insert after specified node d) delete at beginning e) delete at end f) delete after specified node g)display h)search an element

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* prev;

struct Node\* next;

};

struct Node\* head = NULL;

void insertAtBeginning(int data) {

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

newNode->data = data;

newNode->prev = NULL;

newNode->next = head;

if (head) head->prev = newNode;

head = newNode;

}

void insertAtEnd(int data) {

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

newNode->data = data;

newNode->next = NULL;

if (head == NULL) {

newNode->prev = NULL;

head = newNode;

} else {

struct Node\* temp = head;

while (temp->next) temp = temp->next;

temp->next = newNode;

newNode->prev = temp;

}

}

void insertAfter(int key, int data) {

struct Node\* temp = head;

while (temp && temp->data != key) temp = temp->next;

if (temp) {

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

newNode->data = data;

newNode->next = temp->next;

newNode->prev = temp;

if (temp->next) temp->next->prev = newNode;

temp->next = newNode;

}

}

void deleteAtBeginning() {

if (head) {

struct Node\* temp = head;

head = head->next;

if (head) head->prev = NULL;

free(temp);

}

}

void deleteAtEnd() {

if (head) {

if (!head->next) { // Only one element

free(head);

head = NULL;

} else {

struct Node\* temp = head;

while (temp->next) temp = temp->next;

temp->prev->next = NULL;

free(temp);

}

}

}

void deleteAfter(int key) {

struct Node\* temp = head;

while (temp && temp->data != key) temp = temp->next;

if (temp && temp->next) {

struct Node\* toDelete = temp->next;

temp->next = toDelete->next;

if (toDelete->next) toDelete->next->prev = temp;

free(toDelete);

}

}

void display() {

struct Node\* temp = head;

while (temp) {

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

temp = temp->next;

}

printf("NULL\n");

}

void search(int key) {

struct Node\* temp = head;

int position = 1; // Starting from position 1

while (temp) {

if (temp->data == key) {

printf("Element %d found at position %d\n", key, position);

return;

}

temp = temp->next;

position++;

}

printf("Element %d not found\n", key);

}

int main() {

insertAtBeginning(10);

insertAtEnd(20);

insertAtEnd(30);

insertAtBeginning(5);

insertAfter(20, 25);

printf("Doubly Linked List: ");

display(); // Output: 5 <-> 10 <-> 20 <-> 25 <-> 30 <-> NULL

deleteAtBeginning();

deleteAtEnd();

deleteAfter(20);

printf("Doubly Linked List after deletions: ");

display(); // Output: 10 <-> 20 <-> NULL

search(20); // Output: Element 20 found at position 2

search(40); // Output: Element 40 not found

return 0;

}

Write a program to implement circular singly linked list and perform the following operations a) Insert at beginning b) Insert at end c) Insert after specified node d) delete at beginning e) delete at end f) delete after specified node g) display h) search an element

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Node\* head = NULL;

void insertAtBeginning(int data) {

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

newNode->data = data;

if (head == NULL) {

newNode->next = newNode; // Point to itself if list is empty

head = newNode;

} else {

struct Node\* temp = head;

while (temp->next != head) temp = temp->next;

temp->next = newNode;

newNode->next = head;

head = newNode;

}

}

void insertAtEnd(int data) {

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

newNode->data = data;

if (head == NULL) {

newNode->next = newNode; // Point to itself if list is empty

head = newNode;

} else {

struct Node\* temp = head;

while (temp->next != head) temp = temp->next;

temp->next = newNode;

newNode->next = head;

}

}

void insertAfter(int key, int data) {

struct Node\* temp = head;

while (temp && temp->data != key) temp = temp->next;

if (temp) {

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

newNode->data = data;

newNode->next = temp->next;

temp->next = newNode;

}

}

void deleteAtBeginning() {

if (head) {

struct Node\* temp = head;

if (head->next == head) {

free(head);

head = NULL;

} else {

struct Node\* last = head;

while (last->next != head) last = last->next;

head = head->next;

last->next = head;

free(temp);

}

}

}

void deleteAtEnd() {

if (head) {

struct Node\* temp = head;

if (head->next == head) {

free(head);

head = NULL;

} else {

struct Node\* prev = NULL;

while (temp->next != head) {

prev = temp;

temp = temp->next;

}

prev->next = head;

free(temp);

}

}

}

void deleteAfter(int key) {

struct Node\* temp = head;

while (temp && temp->data != key) temp = temp->next;

if (temp && temp->next != head) {

struct Node\* toDelete = temp->next;

temp->next = toDelete->next;

free(toDelete);

}

}

void display() {

if (head) {

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("(head)\n");

} else {

printf("List is empty\n");

}

}

void search(int key) {

struct Node\* temp = head;

int position = 1; // Start counting positions from 1

if (temp) {

do {

if (temp->data == key) {

printf("Element %d found at position %d\n", key, position);

return;

}

temp = temp->next;

position++;

} while (temp != head);

}

printf("Element %d not found\n", key);

}

int main() {

insertAtBeginning(10);

insertAtEnd(20);

insertAtEnd(30);

insertAtBeginning(5);

insertAfter(20, 25);

printf("Circular Singly Linked List: ");

display(); // Output: 5 -> 10 -> 20 -> 25 -> 30 -> (head)

deleteAtBeginning();

deleteAtEnd();

deleteAfter(20);

printf("Circular Singly Linked List after deletions: ");

display(); // Output: 10 -> 20 -> (head)

search(20); // Output: Element 20 found at position 2

search(40); // Output: Element 40 not found

return 0;

}

Write a program to implement circular doubly linked list and perform the following operations a) Insert at beginning b) Insert at end c) Insert after specified node d) delete at beginning e) delete at end f) delete after specified node g) display h) search an element

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

struct Node\* head = NULL;

void insertAtBeginning(int data) {

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

newNode->data = data;

if (head == NULL) {

newNode->next = newNode;

newNode->prev = newNode;

head = newNode;

} else {

struct Node\* temp = head;

while (temp->next != head) temp = temp->next; // Traverse to the last node

temp->next = newNode;

newNode->prev = temp;

newNode->next = head;

head->prev = newNode;

head = newNode; // Update head to the new node

}

}

void insertAtEnd(int data) {

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

newNode->data = data;

if (head == NULL) {

newNode->next = newNode;

newNode->prev = newNode;

head = newNode;

} else {

struct Node\* temp = head;

while (temp->next != head) temp = temp->next; // Traverse to the last node

temp->next = newNode;

newNode->prev = temp;

newNode->next = head;

head->prev = newNode;

}

}

void insertAfter(int key, int data) {

struct Node\* temp = head;

while (temp && temp->data != key) temp = temp->next;

if (temp) {

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

newNode->data = data;

newNode->next = temp->next;

newNode->prev = temp;

if (temp->next) temp->next->prev = newNode;

temp->next = newNode;

}

}

void deleteAtBeginning() {

if (head) {

struct Node\* temp = head;

if (head->next == head) {

free(head);

head = NULL;

} else {

struct Node\* last = head->prev;

head = head->next;

last->next = head;

head->prev = last;

free(temp);

}

}

}

void deleteAtEnd() {

if (head) {

struct Node\* temp = head;

if (head->next == head) {

free(head);

head = NULL;

} else {

while (temp->next != head) temp = temp->next;

temp->prev->next = head;

head->prev = temp->prev;

free(temp);

}

}

}

void deleteAfter(int key) {

struct Node\* temp = head;

while (temp && temp->data != key) temp = temp->next;

if (temp && temp->next != head) {

struct Node\* toDelete = temp->next;

temp->next = toDelete->next;

toDelete->next->prev = temp;

free(toDelete);

}

}

void display() {

if (head) {

struct Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

printf("(head)\n");

} else {

printf("List is empty\n");

}

}

void search(int key) {

struct Node\* temp = head;

int position = 1; // Start counting positions from 1

if (temp) {

do {

if (temp->data == key) {

printf("Element %d found at position %d\n", key, position);

return;

}

temp = temp->next;

position++;

} while (temp != head);

}

printf("Element %d not found\n", key);

}

int main() {

insertAtBeginning(10);

insertAtEnd(20);

insertAtEnd(30);

insertAtBeginning(5);

insertAfter(20, 25);

printf("Circular Doubly Linked List: ");

display(); // Output: 5 <-> 10 <-> 20 <-> 25 <-> 30 <-> (head)

deleteAtBeginning();

deleteAtEnd();

deleteAfter(20);

printf("Circular Doubly Linked List after deletions: ");

display(); // Output: 10 <-> 20 <-> (head)

search(20); // Output: Element 20 found at position 2

search(40); // Output: Element 40 not found

return 0;

}

WAP to Create A GLL of type A=(a,b, (c,d),e, (f,g)….). Perform COPY operation on it.

#include <stdio.h>

#include <stdlib.h>

// Define a structure for a Generalized Linked List node

struct GLLNode {

int isAtom; // 1 if it's an atom (element), 0 if it's a sublist

union {

char data; // Used if it's an atomic element (character)

struct GLLNode\* sublist; // Used if it's a sublist

};

struct GLLNode\* next; // Pointer to the next element in the list

};

// Function to create a new atomic node (single element)

struct GLLNode\* createAtomNode(char data) {

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

newNode->isAtom = 1;

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to create a new sublist node (pointing to another GLL)

struct GLLNode\* createSublistNode(struct GLLNode\* sublist) {

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

newNode->isAtom = 0;

newNode->sublist = sublist;

newNode->next = NULL;

return newNode;

}

// Function to copy a GLL

struct GLLNode\* copyGLL(struct GLLNode\* head) {

if (head == NULL) return NULL;

struct GLLNode\* newHead = NULL;

struct GLLNode\* temp = head;

struct GLLNode\* prev = NULL;

while (temp != NULL) {

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

if (temp->isAtom) {

newNode->isAtom = 1;

newNode->data = temp->data;

} else {

newNode->isAtom = 0;

newNode->sublist = copyGLL(temp->sublist); // Recursively copy the sublist

}

newNode->next = NULL;

if (prev == NULL) {

newHead = newNode;

} else {

prev->next = newNode;

}

prev = newNode;

temp = temp->next;

}

return newHead;

}

// Function to print the GLL

void printGLL(struct GLLNode\* head) {

if (head == NULL) {

printf("Empty GLL\n");

return;

}

struct GLLNode\* temp = head;

int first = 1; // To handle spacing between elements

while (temp != NULL) {

if (!first) {

printf(" "); // Print space between elements

}

first = 0; // No space before the first element

if (temp->isAtom) {

printf("%c", temp->data); // Print atom

} else {

printf("(");

printGLL(temp->sublist); // Recursively print sublist

printf(")");

}

temp = temp->next;

}

printf(""); // Remove the newline to prevent extra line breaks

}

int main() {

// Create a GLL (A = (a, b, (c, d), e, (f, g)) ...)

struct GLLNode\* a = createAtomNode('a');

struct GLLNode\* b = createAtomNode('b');

struct GLLNode\* c = createAtomNode('c');

struct GLLNode\* d = createAtomNode('d');

struct GLLNode\* e = createAtomNode('e');

struct GLLNode\* f = createAtomNode('f');

struct GLLNode\* g = createAtomNode('g');

// Create sublists

struct GLLNode\* sublist1 = createSublistNode(c);

sublist1->next = createSublistNode(d);

struct GLLNode\* sublist2 = createSublistNode(f);

sublist2->next = createSublistNode(g);

// Create main list (A = (a, b, (c, d), e, (f, g)) ...)

a->next = b;

b->next = createSublistNode(sublist1);

b->next->next = e;

e->next = createSublistNode(sublist2);

// Print the original GLL

printf("Original GLL: ");

printGLL(a);

printf("\n");

// Perform copy operation on the GLL

struct GLLNode\* copiedList = copyGLL(a);

// Print the copied GLL

printf("Copied GLL: ");

printGLL(copiedList);

printf("\n");

return 0;

}

Write a program to implement Linear queue using array and perform the

following operations a)Insert b)delete c)peek d)queue full() e)queue empty()

#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Define the maximum size of the queue

struct Queue {

int front, rear;

int arr[MAX];

};

// Function to initialize the queue

void initializeQueue(struct Queue\* q) {

q->front = -1;

q->rear = -1;

}

// Function to check if the queue is full

int isFull(struct Queue\* q) {

if (q->rear == MAX - 1)

return 1; // Queue is full

return 0; // Queue is not full

}

// Function to check if the queue is empty

int isEmpty(struct Queue\* q) {

if (q->front == -1 || q->front > q->rear)

return 1; // Queue is empty

return 0; // Queue is not empty

}

// Function to insert an element into the queue

void insert(struct Queue\* q, int value) {

if (isFull(q)) {

printf("Queue is full. Cannot insert %d\n", value);

return;

}

if (q->front == -1) // If queue is initially empty

q->front = 0;

q->rear++;

q->arr[q->rear] = value;

printf("%d inserted into queue\n", value);

}

// Function to delete an element from the queue

void delete(struct Queue\* q) {

if (isEmpty(q)) {

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

return;

}

printf("%d deleted from queue\n", q->arr[q->front]);

q->front++;

}

// Function to peek the front element of the queue

int peek(struct Queue\* q) {

if (isEmpty(q)) {

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

return -1;

}

return q->arr[q->front];

}

// Function to display the current elements in the queue

void display(struct Queue\* q) {

if (isEmpty(q)) {

printf("Queue is empty\n");

return;

}

printf("Queue elements: ");

for (int i = q->front; i <= q->rear; i++) {

printf("%d ", q->arr[i]);

}

printf("\n");

}

int main() {

struct Queue q;

initializeQueue(&q); // Initialize the queue

// Insert elements into the queue

insert(&q, 10);

insert(&q, 20);

insert(&q, 30);

insert(&q, 40);

insert(&q, 50);

// Display the queue

display(&q);

// Try inserting when the queue is full

insert(&q, 60); // This should display an error message

// Peek the front element

printf("Peek: %d\n", peek(&q));

// Delete elements from the queue

delete(&q);

delete(&q);

// Display the queue again

display(&q);

// Check if the queue is empty

if (isEmpty(&q)) {

printf("Queue is empty\n");

}

else {

printf("Queue not empty\n");

}

// Delete all remaining elements

delete(&q);

delete(&q);

delete(&q);

// Check if the queue is empty

if (isEmpty(&q)) {

printf("Queue is empty\n");

}

return 0;

}

Write a program to implement Circular queue using array and perform the following operations a) Insert b) delete c) display rear d) display front d) queue full() e)queue empty()

#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Define the maximum size of the queue

struct CircularQueue {

int front, rear;

int arr[MAX];

};

// Function to initialize the queue

void initializeQueue(struct CircularQueue\* q) {

q->front = q->rear = -1;

}

// Function to check if the queue is full

int isFull(struct CircularQueue\* q) {

return (q->rear + 1) % MAX == q->front;

}

// Function to check if the queue is empty

int isEmpty(struct CircularQueue\* q) {

return q->front == -1;

}

// Function to insert an element into the queue

void insert(struct CircularQueue\* q, int value) {

if (isFull(q)) {

printf("Queue is full. Cannot insert %d\n", value);

return;

}

if (q->front == -1) // If queue is empty

q->front = 0;

q->rear = (q->rear + 1) % MAX;

q->arr[q->rear] = value;

printf("%d inserted into queue\n", value);

}

// Function to delete an element from the queue

void delete(struct CircularQueue\* q) {

if (isEmpty(q)) {

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

return;

}

printf("%d deleted from queue\n", q->arr[q->front]);

if (q->front == q->rear) // If there is only one element

q->front = q->rear = -1;

else

q->front = (q->front + 1) % MAX;

}

// Function to display the front element

void displayFront(struct CircularQueue\* q) {

if (isEmpty(q)) {

printf("Queue is empty\n");

return;

}

printf("Front element: %d\n", q->arr[q->front]);

}

// Function to display the rear element

void displayRear(struct CircularQueue\* q) {

if (isEmpty(q)) {

printf("Queue is empty\n");

return;

}

printf("Rear element: %d\n", q->arr[q->rear]);

}

// Function to display all elements in the queue

void display(struct CircularQueue\* q) {

if (isEmpty(q)) {

printf("Queue is empty\n");

return;

}

int i = q->front;

while (i != q->rear) {

printf("%d ", q->arr[i]);

i = (i + 1) % MAX;

}

printf("%d\n", q->arr[q->rear]); // Print the rear element

}

int main() {

struct CircularQueue q;

initializeQueue(&q); // Initialize the queue

// Insert elements into the queue

insert(&q, 10);

insert(&q, 20);

insert(&q, 30);

insert(&q, 40);

insert(&q, 50);

// Display the queue

display(&q);

// Insert when the queue is full

insert(&q, 60); // This should display an error message

// Display front and rear

displayFront(&q);

displayRear(&q);

// Delete elements from the queue

delete(&q);

delete(&q);

// Display the queue again

display(&q);

// Delete remaining elements

delete(&q);

delete(&q);

delete(&q);

if (isEmpty(&q)) {

printf("Queue is empty\n");

}

return 0;

}

Write a program to implement Doubly ended queue using array and perform the following operations a) Insert front b) Insert rear c) delete rear d) delete front

#include <stdio.h>

#include <stdlib.h>

#define MAX 5 // Define the maximum size of the deque

struct Deque {

int front, rear;

int arr[MAX];

};

// Function to initialize the deque

void initializeDeque(struct Deque\* dq) {

dq->front = dq->rear = -1;

}

// Function to check if the deque is full

int isFull(struct Deque\* dq) {

return (dq->rear + 1) % MAX == dq->front;

}

// Function to check if the deque is empty

int isEmpty(struct Deque\* dq) {

return dq->front == -1;

}

// Function to insert an element at the front of the deque

void insertFront(struct Deque\* dq, int value) {

if (isFull(dq)) {

printf("Deque is full. Cannot insert %d at front\n", value);

return;

}

if (dq->front == -1) { // If deque is empty

dq->front = dq->rear = 0;

} else {

dq->front = (dq->front - 1 + MAX) % MAX;

}

dq->arr[dq->front] = value;

printf("%d inserted at front\n", value);

}

// Function to insert an element at the rear of the deque

void insertRear(struct Deque\* dq, int value) {

if (isFull(dq)) {

printf("Deque is full. Cannot insert %d at rear\n", value);

return;

}

if (dq->front == -1) { // If deque is empty

dq->front = dq->rear = 0;

} else {

dq->rear = (dq->rear + 1) % MAX;

}

dq->arr[dq->rear] = value;

printf("%d inserted at rear\n", value);

}

// Function to delete an element from the front of the deque

void deleteFront(struct Deque\* dq) {

if (isEmpty(dq)) {

printf("Deque is empty. Cannot delete from front\n");

return;

}

printf("%d deleted from front\n", dq->arr[dq->front]);

if (dq->front == dq->rear) { // Only one element

dq->front = dq->rear = -1;

} else {

dq->front = (dq->front + 1) % MAX;

}

}

// Function to delete an element from the rear of the deque

void deleteRear(struct Deque\* dq) {

if (isEmpty(dq)) {

printf("Deque is empty. Cannot delete from rear\n");

return;

}

printf("%d deleted from rear\n", dq->arr[dq->rear]);

if (dq->front == dq->rear) { // Only one element

dq->front = dq->rear = -1;

} else {

dq->rear = (dq->rear - 1 + MAX) % MAX;

}

}

// Function to display the current elements in the deque

void display(struct Deque\* dq) {

if (isEmpty(dq)) {

printf("Deque is empty\n");

return;

}

int i = dq->front;

printf("Deque elements: ");

while (i != dq->rear) {

printf("%d ", dq->arr[i]);

i = (i + 1) % MAX;

}

printf("%d\n", dq->arr[dq->rear]); // Print the rear element

}

int main() {

struct Deque dq;

initializeDeque(&dq); // Initialize the deque

// Perform operations on the deque

insertRear(&dq, 10);

insertRear(&dq, 20);

insertFront(&dq, 5);

insertRear(&dq, 30);

insertFront(&dq, 2);

// Display the deque

display(&dq);

// Delete elements from the deque

deleteFront(&dq);

deleteRear(&dq);

// Display the deque again

display(&dq);

// Delete remaining elements

deleteFront(&dq);

deleteRear(&dq);

deleteFront(&dq);

// Check if the deque is empty

if (isEmpty(&dq)) {

printf("Deque is empty\n");

}

return 0;

}

Write a menu-driven program that maintains a queue of passengers waiting to see a ticket agent. The program user should be able to insert a new passenger at the rear of the queue, display the passenger at the front of the queue, or remove the passenger at the front of the queue. The program will display the number of passengers left in the queue just before it terminates.

#include <stdio.h>

#include <stdlib.h>

#define MAX 5

struct Queue {

int front, rear;

int arr[MAX];

};

void initializeQueue(struct Queue\* q) {

q->front = q->rear = -1;

}

int isFull(struct Queue\* q) {

return (q->rear + 1) % MAX == q->front;

}

int isEmpty(struct Queue\* q) {

return q->front == -1;

}

void insert(struct Queue\* q, int value) {

if (isFull(q)) {

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

return;

}

if (q->front == -1) {

q->front = 0;

}

q->rear = (q->rear + 1) % MAX;

q->arr[q->rear] = value;

printf("Passenger %d added to the queue.\n", value);

}

void removeFront(struct Queue\* q) {

if (isEmpty(q)) {

printf("Queue is empty. No passenger to remove.\n");

return;

}

printf("Passenger %d removed from the queue.\n", q->arr[q->front]);

if (q->front == q->rear) {

q->front = q->rear = -1;

} else {

q->front = (q->front + 1) % MAX;

}

}

void displayFront(struct Queue\* q) {

if (isEmpty(q)) {

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

return;

}

printf("Passenger at the front: %d\n", q->arr[q->front]);

}

int main() {

struct Queue q;

initializeQueue(&q);

int choice, passengerID;

while (1) {

printf("\nMenu:\n");

printf("1. Insert Passenger at rear\n");

printf("2. Display Passenger at front\n");

printf("3. Remove Passenger from front\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter Passenger ID to add: ");

scanf("%d", &passengerID);

insert(&q, passengerID);

break;

case 2:

displayFront(&q);

break;

case 3:

removeFront(&q);

break;

case 4:

printf("Number of passengers left in queue: ");

if (isEmpty(&q)) {

printf("0\n");

} else {

int count = (q.rear - q.front + MAX) % MAX + 1;

printf("%d\n", count);

}

printf("Exiting program.\n");

return 0;

default:

printf("Invalid choice. Please try again.\n");

}

}

}

There are n people in a line queuing to buy tickets, where the 0th person is at the front of the line and the (n - 1)th person is at the back of the line.

You are given a 0-indexed integer array tickets of length n where the number of tickets that the ith person would like to buy is tickets[i].

Each person takes exactly 1 second to buy a ticket. A person can only buy 1 ticket at a time and has to go back to the end of the line (which happens instantaneously) in order to buy more tickets. If a person does not have any tickets left to buy, the person will leave the line.

Return the time taken for the person initially at position k (0-indexed) to finish buying tickets. Input: tickets = [2,3,2], k = 2

Output: 6

#include <stdio.h>

int timeToBuyTickets(int\* tickets, int ticketsSize, int k) {

int time = 0;

while (tickets[k] > 0) {

// Process each person in the queue

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

if (tickets[i] > 0) {

tickets[i]--; // This person buys a ticket

time++; // Increment time for each ticket bought

}

}

}

return time;

}

int main() {

int tickets[] = {2, 3, 2};

int k = 2;

int ticketsSize = sizeof(tickets) / sizeof(tickets[0]);

int result = timeToBuyTickets(tickets, ticketsSize, k);

printf("Time taken for person %d to finish buying tickets: %d\n", k, result);

return 0;

}

Write a program to push (), pop (), display (), peek (), stack full () and stack empty () operations on stack using array.

#include <stdio.h>

#define MAX 5

int stack[MAX];

int top = -1;

void push(int val) {

if (top == MAX - 1) printf("Stack is full\n");

else stack[++top] = val;

}

void pop() {

if (top == -1) printf("Stack is empty\n");

else top--;

}

void display() {

if (top == -1) printf("Stack is empty\n");

else for (int i = 0; i <= top; i++) printf("%d ", stack[i]);

printf("\n");

}

int peek() {

if (top == -1) {

printf("Stack is empty\n");

return -1;

}

return stack[top];

}

int stackFull() {

return top == MAX - 1;

}

int stackEmpty() {

return top == -1;

}

int main() {

push(10);

push(20);

push(30);

push(40);

push(50);

display(); // Output: 10 20 30 40 50

printf("Stack full: %d\n", stackFull()); // Output: 1

printf("Peek: %d\n", peek()); // Output: 50

pop();

display(); // Output: 10 20 30 40

printf("Stack full: %d\n", stackFull()); // Output: 0

printf("Stack empty: %d\n", stackEmpty()); // Output: 0

pop();

pop();

pop();

pop();

printf("Stack empty: %d\n", stackEmpty()); // Output: 1

return 0;

}

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

MinStack() initializes the stack object.

void push(int val) pushes the element val onto the stack. void pop() removes the element on the top of the stack.

int top() gets the top element of the stack. int getMin() retrieves the minimum element in the stack.

#include <stdio.h>

#include <limits.h>

#define MAX\_SIZE 1000

// Structure for MinStack

typedef struct {

int stack[MAX\_SIZE];

int minStack[MAX\_SIZE];

int top;

} MinStack;

// Function to initialize the stack

void init(MinStack\* minStack) {

minStack->top = -1;

}

// Function to push an element onto the stack

void push(MinStack\* minStack, int val) {

if (minStack->top == MAX\_SIZE - 1) {

printf("Stack overflow\n");

return;

}

minStack->stack[++(minStack->top)] = val;

// Push to minStack if it's empty or new value is smaller or equal to current minimum

if (minStack->top == 0 || val <= minStack->minStack[minStack->top - 1]) {

minStack->minStack[minStack->top] = val;

} else {

minStack->minStack[minStack->top] = minStack->minStack[minStack->top - 1];

}

}

// Function to pop the top element from the stack

void pop(MinStack\* minStack) {

if (minStack->top == -1) {

printf("Stack underflow\n");

return;

}

minStack->top--;

}

// Function to get the top element of the stack

int top(MinStack\* minStack) {

if (minStack->top == -1) {

printf("Stack is empty\n");

return INT\_MIN; // Return a sentinel value if the stack is empty

}

return minStack->stack[minStack->top];

}

// Function to get the minimum element from the stack

int getMin(MinStack\* minStack) {

if (minStack->top == -1) {

printf("Stack is empty\n");

return INT\_MIN; // Return a sentinel value if the stack is empty

}

return minStack->minStack[minStack->top];

}

int main() {

MinStack minStack;

init(&minStack);

push(&minStack, 3);

push(&minStack, 1);

push(&minStack, 2);

printf("Top: %d\n", top(&minStack)); // Output: 2

printf("Min: %d\n", getMin(&minStack)); // Output: 1

pop(&minStack);

printf("Top: %d\n", top(&minStack)); // Output: 1

printf("Min: %d\n", getMin(&minStack)); // Output: 1

pop(&minStack);

printf("Top: %d\n", top(&minStack)); // Output: 3

printf("Min: %d\n", getMin(&minStack)); // Output: 3

return 0;

}

Write a Program to convert Infix to postfix expression Using Stack.

#include <stdio.h>

#include <ctype.h> // For isdigit()

#include <string.h> // For string manipulation

#define MAX 100

// Stack for operators

char stack[MAX];

int top = -1;

// Function to return precedence of operators

int precedence(char c) {

if (c == '^') return 3;

if (c == '\*' || c == '/') return 2;

if (c == '+' || c == '-') return 1;

return 0;

}

// Function to push elements onto the stack

void push(char c) {

if (top < MAX - 1) {

stack[++top] = c;

}

}

// Function to pop elements from the stack

char pop() {

if (top != -1) {

return stack[top--];

}

return -1; // Stack is empty

}

// Function to convert infix to postfix

void infixToPostfix(char\* infix, char\* postfix) {

int i = 0, j = 0;

while (infix[i] != '\0') {

if (isalnum(infix[i])) {

// If operand, add it to the postfix expression

postfix[j++] = infix[i++];

} else if (infix[i] == '(') {

// If '(', push it onto the stack

push(infix[i++]);

} else if (infix[i] == ')') {

// If ')', pop from the stack until '(' is found

while (top != -1 && stack[top] != '(') {

postfix[j++] = pop();

}

pop(); // Discard the '('

i++;

} else {

// If operator, pop from the stack based on precedence and push current operator

while (top != -1 && precedence(stack[top]) >= precedence(infix[i])) {

postfix[j++] = pop();

}

push(infix[i++]);

}

}

// Pop remaining operators from the stack

while (top != -1) {

postfix[j++] = pop();

}

postfix[j] = '\0'; // Null-terminate the postfix expression

}

int main() {

// Hardcoded Infix expression

char infix[MAX] = "(A+B)\*(C-D)";

char postfix[MAX];

// Convert Infix to Postfix

infixToPostfix(infix, postfix);

// Output Postfix expression

printf("Infix expression: %s\n", infix);

printf("Postfix expression: %s\n", postfix);

return 0;

}

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>

// Structure of a tree node

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 for non-recursive Preorder traversal (using stack)

void preorder(struct Node\* root) {

if (!root) return;

struct Node\* stack[100];

int top = -1;

stack[++top] = root;

while (top >= 0) {

struct Node\* node = stack[top--];

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

if (node->right) stack[++top] = node->right;

if (node->left) stack[++top] = node->left;

}

}

// Function for non-recursive Postorder traversal (using two stacks)

void postorder(struct Node\* root) {

if (!root) return;

struct Node\* stack1[100], \*stack2[100];

int top1 = -1, top2 = -1;

stack1[++top1] = root;

while (top1 >= 0) {

struct Node\* node = stack1[top1--];

stack2[++top2] = node;

if (node->left) stack1[++top1] = node->left;

if (node->right) stack1[++top1] = node->right;

}

// Printing elements from stack2, which gives the postorder traversal

while (top2 >= 0) {

printf("%d ", stack2[top2--]->data);

}

}

// Function to count the total number of nodes

int countNodes(struct Node\* root) {

if (!root) return 0;

struct Node\* stack[100];

int top = -1, count = 0;

stack[++top] = root;

while (top >= 0) {

struct Node\* node = stack[top--];

count++;

if (node->right) stack[++top] = node->right;

if (node->left) stack[++top] = node->left;

}

return count;

}

// Function to display height of the tree

int height(struct Node\* root) {

if (!root) return 0;

struct Node\* stack[100];

int height[100], top = -1, maxHeight = 0;

stack[++top] = root;

height[top] = 1;

while (top >= 0) {

struct Node\* node = stack[top];

int currentHeight = height[top--];

if (node->left) {

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

height[top] = currentHeight + 1;

}

if (node->right) {

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

height[top] = currentHeight + 1;

}

if (currentHeight > maxHeight) maxHeight = currentHeight;

}

return maxHeight;

}

int main() {

// Creating a sample tree

struct Node\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

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

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

printf("Preorder Traversal: ");

preorder(root);

printf("\n");

printf("Postorder Traversal: ");

postorder(root);

printf("\n");

printf("Total Nodes: %d\n", countNodes(root));

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

return 0;

}

Write a program to illustrate operations on a BST holding numeric keys.The menu must include: • Insert • Delete • Find • Show

#include <stdio.h>

#include <stdlib.h>

// Structure for BST node

struct Node {

int data;

struct Node\* left;

struct Node\* 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 in BST

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

if (root == NULL) return newNode(data);

if (data < root->data)

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

else

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

return root;

}

// Function to find the minimum value node in BST

struct Node\* minValueNode(struct Node\* root) {

struct Node\* current = root;

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

current = current->left;

return current;

}

// Function to delete a node from BST

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

if (root == NULL) return root;

if (data < root->data)

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

else if (data > root->data)

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

else {

if (root->left == NULL) {

struct Node\* temp = root->right;

free(root);

return temp;

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

struct Node\* temp = root->left;

free(root);

return temp;

}

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

root->data = temp->data;

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

}

return root;

}

// Function to find a node in BST

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

if (root == NULL) return 0;

if (root->data == data) return 1;

if (data < root->data) return find(root->left, data);

return find(root->right, data);

}

// Function to perform inorder traversal (show the BST)

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

int main() {

struct Node\* root = NULL;

int choice, data;

while (1) {

printf("\nMenu:\n1. Insert\n2. Delete\n3. Find\n4. Show (Inorder)\n5. Exit\n");

printf("Enter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter value to insert: ");

scanf("%d", &data);

root = insert(root, data);

break;

case 2:

printf("Enter value to delete: ");

scanf("%d", &data);

root = deleteNode(root, data);

break;

case 3:

printf("Enter value to find: ");

scanf("%d", &data);

if (find(root, data))

printf("%d found in BST\n", data);

else

printf("%d not found in BST\n", data);

break;

case 4:

printf("Inorder traversal: ");

inorder(root);

printf("\n");

break;

case 5:

exit(0);

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

Write a program to generate a binary tree. Then print the height of given tree by referring the root node

#include <stdio.h>

#include <stdlib.h>

// Definition of a Binary Tree Node

struct Node {

int data;

struct Node\* left;

struct Node\* 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 calculate the height of the binary tree

int height(struct Node\* root) {

if (root == NULL) {

return 0; // Height of an empty tree is 0

}

// Recursively calculate the height of the left and right subtrees

int leftHeight = height(root->left);

int rightHeight = height(root->right);

// The height of the current node is 1 + the maximum height of its left and right subtrees

if (leftHeight > rightHeight) {

return leftHeight + 1;

} else {

return rightHeight + 1;

}

}

// Main function

int main() {

struct Node\* root = NULL;

// Creating the binary tree

root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

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

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

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

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

// Printing the height of the tree

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

return 0;

}

Write a program to to generate a binary tree and then print the nodes levelwise

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a tree node

struct Node {

int data;

struct Node \*left, \*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 print nodes level by level

void printLevelOrder(struct Node\* root) {

if (root == NULL) return;

struct Node\* queue[100];

int front = 0, rear = 0;

queue[rear++] = root;

while (front < rear) {

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

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

if (current->left) queue[rear++] = current->left;

if (current->right) queue[rear++] = current->right;

}

}

// Main function to test the program

int main() {

struct 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);

printf("Level order traversal of binary tree:\n");

printLevelOrder(root);

return 0;

}

Write a Program to implement Dijkstra’s algorithm to find shortest distance between two nodes of a user defined graph. Use Adjacency List to represent a graph.

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Definition of an adjacency list node

struct AdjListNode {

int dest;

int weight;

struct AdjListNode\* next;

};

// Definition of an adjacency list

struct AdjList {

struct AdjListNode\* head;

};

// Definition of the graph

struct Graph {

int V; // Number of vertices

struct AdjList\* array;

};

// Create a new adjacency list node

struct AdjListNode\* newAdjListNode(int dest, int weight) {

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

newNode->dest = dest;

newNode->weight = weight;

newNode->next = NULL;

return newNode;

}

// Create a graph of V vertices

struct Graph\* createGraph(int V) {

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

graph->V = V;

graph->array = (struct AdjList\*)malloc(V \* sizeof(struct AdjList));

// Initialize the adjacency list

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

graph->array[i].head = NULL;

}

return graph;

}

// Add an edge to the graph

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

// Add an edge from src to dest

struct AdjListNode\* newNode = newAdjListNode(dest, weight);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

// Add an edge from dest to src (since the graph is undirected)

newNode = newAdjListNode(src, weight);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

// A utility function to find the vertex with the minimum distance value

int minDistance(int dist[], int sptSet[], int V) {

int min = INT\_MAX, min\_index;

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

if (sptSet[v] == 0 && dist[v] <= min) {

min = dist[v], min\_index = v;

}

}

return min\_index;

}

// Implement Dijkstra's algorithm to find the shortest path from source to destination

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

int V = graph->V;

int dist[V];

int sptSet[V]; // Shortest Path Tree Set

// Initialize all distances as infinite and sptSet[] as false

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

dist[i] = INT\_MAX;

sptSet[i] = 0;

}

// Distance to the source vertex is 0

dist[src] = 0;

// Find the shortest path for all vertices

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

// Pick the minimum distance vertex from the set of vertices not yet processed

int u = minDistance(dist, sptSet, V);

// Mark the picked vertex as processed

sptSet[u] = 1;

// Update dist value of the adjacent vertices of the picked vertex

struct AdjListNode\* pCrawl = graph->array[u].head;

while (pCrawl != NULL) {

int v = pCrawl->dest;

// Update dist[v] if and only if the new distance is smaller

if (sptSet[v] == 0 && dist[u] != INT\_MAX && dist[u] + pCrawl->weight < dist[v]) {

dist[v] = dist[u] + pCrawl->weight;

}

pCrawl = pCrawl->next;

}

}

// Print the shortest distance from source to destination

if (dist[dest] != INT\_MAX) {

printf("Shortest distance from node %d to node %d is %d\n", src, dest, dist[dest]);

} else {

printf("No path exists from node %d to node %d\n", src, dest);

}

}

// Main function

int main() {

int V, E, src, dest, weight;

// Input the number of vertices and edges

printf("Enter number of vertices: ");

scanf("%d", &V);

struct Graph\* graph = createGraph(V);

// Input the edges

printf("Enter number of edges: ");

scanf("%d", &E);

printf("Enter the edges (src, dest, weight):\n");

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

scanf("%d %d %d", &src, &dest, &weight);

addEdge(graph, src, dest, weight);

}

// Input source and destination for Dijkstra's algorithm

printf("Enter the source node: ");

scanf("%d", &src);

printf("Enter the destination node: ");

scanf("%d", &dest);

// Call Dijkstra's algorithm to find the shortest distance

dijkstra(graph, src, dest);

return 0;

}

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>

#define MAX\_VERTICES 10

// Function to print the adjacency matrix

void printAdjacencyMatrix(int graph[MAX\_VERTICES][MAX\_VERTICES], int numVertices) {

printf("Adjacency Matrix:\n");

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

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

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

}

printf("\n");

}

}

// Function to perform BFS traversal on a graph

void BFS(int graph[MAX\_VERTICES][MAX\_VERTICES], int startVertex, int numVertices) {

int visited[MAX\_VERTICES] = {0}; // Array to keep track of visited nodes

int queue[MAX\_VERTICES], front = 0, rear = 0;

// Start with the given start vertex

visited[startVertex] = 1;

queue[rear++] = startVertex;

printf("BFS Traversal: ");

while (front < rear) {

int vertex = queue[front++];

printf("%d ", vertex);

// Visit all adjacent vertices of the dequeued vertex

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

if (graph[vertex][i] == 1 && !visited[i]) {

visited[i] = 1;

queue[rear++] = i;

}

}

}

printf("\n");

}

// Function to perform DFS traversal on a graph

void DFS(int graph[MAX\_VERTICES][MAX\_VERTICES], int vertex, int visited[MAX\_VERTICES], int numVertices) {

visited[vertex] = 1;

printf("%d ", vertex);

// Visit all adjacent vertices of the current vertex

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

if (graph[vertex][i] == 1 && !visited[i]) {

DFS(graph, i, visited, numVertices);

}

}

}

// Main function to input the graph and perform traversals

int main() {

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

int numVertices, numEdges, u, v;

// Input the number of vertices and edges

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

scanf("%d", &numVertices);

// Initialize the adjacency matrix with 0

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

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

graph[i][j] = 0;

}

}

// Input the edges

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

scanf("%d", &numEdges);

printf("Enter the edges (u v) where u and v are vertex numbers (0-indexed):\n");

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

scanf("%d %d", &u, &v);

graph[u][v] = 1;

graph[v][u] = 1; // For undirected graph

}

// Display the adjacency matrix

printAdjacencyMatrix(graph, numVertices);

// Perform BFS traversal

int startVertex;

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

scanf("%d", &startVertex);

BFS(graph, startVertex, numVertices);

// Perform DFS traversal

int visited[MAX\_VERTICES] = {0}; // Reset visited array for DFS

printf("DFS Traversal starting from vertex %d: ", startVertex);

DFS(graph, startVertex, visited, numVertices);

printf("\n");

return 0;

}

Write a Program to implement Kruskal’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>

#define MAX\_VERTICES 10

// Structure to represent an edge

typedef struct {

int u, v, weight;

} Edge;

// Function to compare two edges (used by qsort)

int compareEdges(const void\* a, const void\* b) {

return ((Edge\*)a)->weight - ((Edge\*)b)->weight;

}

// Function to find the set of an element using path compression

int find(int parent[], int i) {

if (parent[i] == -1) {

return i;

}

return parent[i] = find(parent, parent[i]); // Path compression

}

// Function to do union of two subsets

void unionSets(int parent[], int rank[], int x, int y) {

int rootX = find(parent, x);

int rootY = find(parent, y);

if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

}

// Kruskal’s Algorithm to find the MST

void kruskal(int graph[MAX\_VERTICES][MAX\_VERTICES], int numVertices) {

Edge edges[MAX\_VERTICES \* (MAX\_VERTICES - 1) / 2]; // Store all edges

int edgeCount = 0;

// Step 1: Convert the adjacency matrix to a list of edges

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

for (int j = i + 1; j < numVertices; j++) {

if (graph[i][j] != 0) { // There is an edge

edges[edgeCount].u = i;

edges[edgeCount].v = j;

edges[edgeCount].weight = graph[i][j];

edgeCount++;

}

}

}

// Step 2: Sort the edges based on their weight

qsort(edges, edgeCount, sizeof(Edge), compareEdges);

int parent[numVertices], rank[numVertices];

// Initialize parent and rank arrays

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

parent[i] = -1;

rank[i] = 0;

}

// Step 3: Iterate through sorted edges and apply union-find

printf("Edges in the Minimum Spanning Tree (MST):\n");

int mstWeight = 0;

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

int u = edges[i].u;

int v = edges[i].v;

int weight = edges[i].weight;

// Find the roots of the sets for u and v

int rootU = find(parent, u);

int rootV = find(parent, v);

// If including this edge does not cause a cycle

if (rootU != rootV) {

printf("Edge (%d, %d) with weight %d\n", u, v, weight);

mstWeight += weight;

unionSets(parent, rank, rootU, rootV);

}

}

printf("Total weight of MST: %d\n", mstWeight);

}

int main() {

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

int numVertices, numEdges;

// Input number of vertices and edges

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

scanf("%d", &numVertices);

// Initialize the graph (adjacency matrix)

printf("Enter the adjacency matrix (use 0 for no edge, and the weight for edges):\n");

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

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

scanf("%d", &graph[i][j]);

}

}

// Call Kruskal's Algorithm

kruskal(graph, numVertices);

return 0;

}

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

#include <stdio.h>

#include <limits.h>

#define MAX\_VERTICES 10

// Function to find the vertex with the minimum key value, from the set of vertices that are not yet included in the MST

int minKey(int key[], int mstSet[], int numVertices) {

int min = INT\_MAX, minIndex;

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

if (mstSet[v] == 0 && key[v] < min) {

min = key[v];

minIndex = v;

}

}

return minIndex;

}

// Function to implement Prim's Algorithm to find MST

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

int parent[numVertices]; // Array to store the MST

int key[numVertices]; // Key values used to pick minimum weight edge

int mstSet[numVertices]; // Set of vertices included in the MST

// Initialize all keys to infinity and mstSet[] as false

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

key[i] = INT\_MAX;

mstSet[i] = 0;

}

// Start with the first vertex

key[0] = 0; // Start from vertex 0

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

// The MST will have numVertices vertices

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

// Pick the vertex with the minimum key value that is not yet included in the MST

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

// Include the picked vertex in the MST

mstSet[u] = 1;

// Update the key and parent values of the adjacent vertices

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

// If there is an edge and v is not included in MST, and weight of u-v is smaller than the key[v]

if (graph[u][v] != 0 && mstSet[v] == 0 && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

// Print the constructed MST

printf("Edges in the Minimum Spanning Tree (MST):\n");

int totalWeight = 0;

for (int i = 1; i < numVertices; i++) {

printf("Edge (%d, %d) with weight %d\n", parent[i], i, graph[i][parent[i]]);

totalWeight += graph[i][parent[i]];

}

printf("Total weight of MST: %d\n", totalWeight);

}

int main() {

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

int numVertices;

// Input number of vertices

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

scanf("%d", &numVertices);

// Initialize the graph (adjacency matrix)

printf("Enter the adjacency matrix (use 0 for no edge, and the weight for edges):\n");

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

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

scanf("%d", &graph[i][j]);

}

}

// Call Prim's Algorithm

primMST(graph, numVertices);

return 0;

}