Reverse a 32-bit number :

#include <stdio.h>

int reverse 32BitSigned(int num) {

int result = 0, i;

for (i = 0; i < 32; i++) {

result <<= 1;

result |= num & 1;

num >>= 1;

}

return result;

}

int main() {

int num = 32768, i;

printf("Original: %d\n", num);

int reversed = reverse32BitSigned(num);

printf("Reversed: %d\n", reversed);

return 0;

}

Array operation :

#include<stdio.h>

int main()

{

int n=0;

int a[n],i,j,ch,ele,ind;

printf("\nenter the no of elements : ");

scanf("%d",&n);

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

printf("\nenter the num %d : ",i+1);

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

}

while(ch!=4){

printf("\n1. insert \n2. delete \n3. display \n4. exit \nenter your choice : ");

scanf ("%d",&ch);

if(ch==1)

{

printf("\nenter the element to be added : ");

scanf("%d",&ele);

a[n]=ele;

printf("\nsuccessfully added");

n++;

}

else if(ch==2)

{

printf("\nenter the element to be delete : ");

scanf("%d",&ele);

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

if(a[i]==ele)

ind=i;

}

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

if(i<n-1)

a[i]=a[i+1];

else

a[i]=0;

}

}

else if(ch==3){

printf("\n elements : ");

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

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

}

}

}

}

Binary search :

#include<stdio.h>

int main()

{

int a[10],i,n,low=0,high,mid,s,temp,j,f=0;

printf("\nenter the no of elements : ");

scanf("%d",&n);

high=n-1;

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

{

printf("\nenter the num %d : ",i+1);

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

}

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

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

if(a[i]>a[j]){

temp=a[i];

a[i]=a[j];

a[j]=temp;

}

}

}

printf("\nenter the search element : ");

scanf("%d",&s);

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

{

mid=(low+high)/2;

if(a[mid]==s)

{

printf("\nthe element is present ");

f=1;

exit(0);

}

else if(a[mid]<s){

low=mid+1;

}

else if(a[mid]>s){

high=mid;

}

}

if(f==0)

printf("\nelement is not present");

}

Binary tree :

#include<stdio.h>

struct node {

int data;

struct node \*left,\*right;

};

struct node \*new(int a)

{

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

tmp->data=a;

tmp->left = tmp->right = NULL;

return tmp;

}

struct node \*insert(struct node \*node , int b)

{

if(node==NULL)

return new(b);

if(b<node->data)

node->left=insert(node->left,b);

else

node->right=insert(node->right,b);

return node;

}

void inorder(struct node \*root)

{

if(root!=NULL){

inorder(root->left);

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

inorder(root->right);

}

}

int main()

{

struct node \*root=NULL,\*tmp;

root=insert(root,40);

root=insert(root,50);

root=insert(root,30);

root=insert(root,35);

root=insert(root,37);

inorder(root);

}

Array concatenation :

#include <stdio.h>

int main() {

int array[10];

int even[5] = {0, 2, 4, 6, 8};

int odd[5] = {1, 3, 5, 7, 9};

int loop, index, e\_len, o\_len;

e\_len = o\_len = 5;

index = 0;

for(loop = 0; loop < e\_len; loop++) {

array[index] = even[loop];

index++;

}

for(loop = 0; loop < o\_len; loop++) {

array[index] = odd[loop];

index++;

}

printf("\nEven -> ");

for(loop = 0; loop < e\_len; loop++)

printf(" %d", even[loop]);

printf("\nOdd -> ");

for(loop = 0; loop < o\_len; loop++)

printf(" %d", odd[loop]);

printf("\nConcat -> ");

for(loop = 0; loop < 10; loop++)

printf(" %d", array[loop]);

return 0;

}

Sorting an array :

#include <stdio.h>

int main() {

int array[10];

int even[5] = {0, 2, 4, 6, 8};

int odd[5] = {1, 3, 5, 7, 9};

int loop, index, e\_len, o\_len;

e\_len = o\_len = 5;

index = 0;

for(loop = 0; loop < e\_len; loop++) {

array[index] = even[loop];

index++;

}

for(loop = 0; loop < o\_len; loop++) {

array[index] = odd[loop];

index++;

}

printf("\nEven -> ");

for(loop = 0; loop < e\_len; loop++)

printf(" %d", even[loop]);

printf("\nOdd -> ");

for(loop = 0; loop < o\_len; loop++)

printf(" %d", odd[loop]);

printf("\nConcat -> ");

for(loop = 0; loop < 10; loop++)

printf(" %d", array[loop]);

return 0;

}

Bubble sort :

#include <stdio.h>

void bubble\_sort(int arr[], int n) {

int i, j;

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

for (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[] = {64, 34, 25, 12, 22, 11, 90};

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

bubble\_sort(arr, n);

printf("Sorted array: ");

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

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

}

return 0;

}

Djikstra algorithm :

#include <stdio.h>

#define INFINITY 9999

#define MAX 10

void Dijkstra(int Graph[MAX][MAX], int n, int start);

void Dijkstra(int Graph[MAX][MAX], int n, int start) {

int cost[MAX][MAX], distance[MAX], pred[MAX];

int visited[MAX], count, mindistance, nextnode, i, j;

// Creating cost matrix

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

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

if (Graph[i][j] == 0)

cost[i][j] = INFINITY;

else

cost[i][j] = Graph[i][j];

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

distance[i] = cost[start][i];

pred[i] = start;

visited[i] = 0;

}

distance[start] = 0;

visited[start] = 1;

count = 1;

while (count < n - 1) {

mindistance = INFINITY;

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

if (distance[i] < mindistance && !visited[i]) {

mindistance = distance[i];

nextnode = i;

}

visited[nextnode] = 1;

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

if (!visited[i])

if (mindistance + cost[nextnode][i] < distance[i]) {

distance[i] = mindistance + cost[nextnode][i];

pred[i] = nextnode;

}

count++;

}

// Printing the distance

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

if (i != start) {

printf("\nDistance from source to %d: %d", i, distance[i]);

}

}

int main() {

int Graph[MAX][MAX], i, j, n, u;

n = 7;

Graph[0][0] = 0;

Graph[0][1] = 0;

Graph[0][2] = 1;

Graph[0][3] = 2;

Graph[0][4] = 0;

Graph[0][5] = 0;

Graph[0][6] = 0;

Graph[1][0] = 0;

Graph[1][1] = 0;

Graph[1][2] = 2;

Graph[1][3] = 0;

Graph[1][4] = 0;

Graph[1][5] = 3;

Graph[1][6] = 0;

Graph[2][0] = 1;

Graph[2][1] = 2;

Graph[2][2] = 0;

Graph[2][3] = 1;

Graph[2][4] = 3;

Graph[2][5] = 0;

Graph[2][6] = 0;

Graph[3][0] = 2;

Graph[3][1] = 0;

Graph[3][2] = 1;

Graph[3][3] = 0;

Graph[3][4] = 0;

Graph[3][5] = 0;

Graph[3][6] = 1;

Graph[4][0] = 0;

Graph[4][1] = 0;

Graph[4][2] = 3;

Graph[4][3] = 0;

Graph[4][4] = 0;

Graph[4][5] = 2;

Graph[4][6] = 0;

Graph[5][0] = 0;

Graph[5][1] = 3;

Graph[5][2] = 0;

Graph[5][3] = 0;

Graph[5][4] = 2;

Graph[5][5] = 0;

Graph[5][6] = 1;

Graph[6][0] = 0;

Graph[6][1] = 0;

Graph[6][2] = 0;

Graph[6][3] = 1;

Graph[6][4] = 0;

Graph[6][5] = 1;

Graph[6][6] = 0;

u = 0;

Dijkstra(Graph, n, u);

return 0;

}

Finding an element in an array :

#include <stdio.h>

int main()

{

int array[100], search, c, n;

printf("Enter number of elements in array\n");

scanf("%d", &n);

printf("Enter %d integer(s)\n", n);

for (c = 0; c < n; c++)

scanf("%d", &array[c]);

printf("Enter a number to search\n");

scanf("%d", &search);

for (c = 0; c < n; c++)

{

if (array[c] == search) /\* If required element is found \*/

{

printf("%d is present at location %d.\n", search, c+1);

break;

}

}

if (c == n)

printf("%d isn't present in the array.\n", search);

return 0;

}

Heap sort :

#include <stdio.h>

void swap(int \*a, int \*b) {

int tempvar = \*a;

\*a = \*b;

\*b = tempvar;

}

void heapify(int arr[], int n, int i) {

int greatest = i;

int leftSide = 2 \* i + 1;

int rightSide = 2 \* i + 2;

if (leftSide < n && arr[leftSide] > arr[greatest])

greatest = leftSide;

if (rightSide < n && arr[rightSide] > arr[greatest])

greatest = rightSide;

if (greatest != i) {

swap(&arr[i], &arr[greatest]);

heapify(arr, n, greatest);

}

}

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

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

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

swap(&arr[0], &arr[i]);

heapify(arr, i, 0);

}

}

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

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

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

printf("\n");

}

int main() {

int arr[] = {1, 12, 9, 5, 6, 10};

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

heapSort(arr, n);

printf("Sorted array is \n");

printArray(arr, n);

}

Insertion sort :

#include <math.h>

#include <stdio.h>

void insertionSort(int arr[], int n)

{

int i, key, j;

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

{

key = arr[i];

j = i - 1;

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

{

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

j = j - 1;

}

arr[j + 1] = key;

}

}

void printArray(int arr[], int n)

{

int i;

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

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

printf("\n");

}

int main()

{

int n,i;

printf("Enter no of element :");

scanf("%d",&n);

int arr[n];

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

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

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

}

insertionSort(arr, n);

printArray(arr, n);

return 0;

}

Linear search :

#include <stdio.h>

int main()

{

int array[100], search, c, n;

printf("Enter number of elements in array\n");

scanf("%d", &n);

printf("Enter %d integer(s)\n", n);

for (c = 0; c < n; c++)

scanf("%d", &array[c]);

printf("Enter a number to search\n");

scanf("%d", &search);

for (c = 0; c < n; c++)

{

if (array[c] == search) /\* If required element is found \*/

{

printf("%d is present at location %d.\n", search, c+1);

break;

}

}

if (c == n)

printf("%d isn't present in the array.\n", search);

return 0;

}

Merging two linked list :

#include<stdio.h>

struct node{

int data;

struct node \*next;

};

struct node \*start1, \*start2;

int main()

{

int n=2,i,j,val;

struct node \*tmp1 ,\*tmp2,\*fnnode1,\*fnnode2 ,\*tmp3,\*tmp4;

start1=(struct node\* )malloc(sizeof(struct node\*));

start2=(struct node\* )malloc(sizeof(struct node\*));

printf("\nenter the value of start 1: ");

scanf("%d",&val);

start1->data=val;

start1->next=NULL;

tmp1=start1;

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

{

fnnode1=(struct node\* )malloc(sizeof(struct node\*));

printf("\nenter the data of ll1 : ");

scanf("%d",&val);

fnnode1->data=val;

fnnode1->next=NULL;

tmp1->next=fnnode1;

tmp1=fnnode1;

}

printf("\nenter the value of start 2: ");

scanf("%d",&val);

start2->data=val;

start2->next=NULL;

tmp2=start2;

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

{

fnnode2=(struct node\* )malloc(sizeof(struct node\*));

printf("\nenter the data of ll2 : ");

scanf("%d",&val);

fnnode2->data=val;

fnnode2->next=NULL;

tmp2->next=fnnode2;

tmp2=fnnode2;

}

tmp1=start1;

tmp2=start2;

i=0;

while(tmp1!=NULL)

{

tmp3=tmp1->next;

tmp4=tmp2->next;

tmp1->next=tmp2;

tmp2->next=tmp3;

tmp2=tmp4;

tmp1=tmp3;

i++;

}

tmp1=start1;

while(tmp1!=NULL)

{

printf("\n%d",tmp1->data);

tmp1=tmp1->next;

}

}

Missed element in an array :

#include <stdio.h>

void main()

{

int n, i, j, c, t, b;

printf("Enter size of array : ");

scanf("%d", &n);

int array[n - 1]; /\* array size-1 \*/

printf("Enter elements into array : \n");

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

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

b = array[0];

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

b = b ^ array[i];

for (i = 2, c = 1; i <= n; i++)

c = c ^ i;

c = c ^ b;

printf("Missing element is : %d \n", c);

}

Repeatation of element :

#include<stdio.h>

int main()

{

int arr[] = {20, 30, 10, 2, 10, 20, 30, 11};

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

int visited[n];

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

// only if unvisited

if(visited[i] == 0){

int count = 1;

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

// if appears again in the array

if(arr[i] == arr[j])

{ // increase count & mark index visited

count++;

visited[j] = 1;

}

} //

if(count > 1)

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

}

}

return 0;

}

Travelling salesman problem :

#include<stdio.h>

int a[10][10],v[10],n,k1=1,sum=0;

int tra(int b){

int i,j,k=0,min,fl,nod=0,v1=0;

k=k1;

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

if(i!=b){

min=a[b][i];

nod=i;

break;

}

}

// printf("\nmin%d",nod);

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

{

fl=0;

if(a[b][i]<min & i!=b)

{

min=a[b][i];

nod=i;

}

for(j=0;j<k;j++)

{

if(v[j]==nod)

{

fl=1;

}

}

if(fl!=1){

v[k]=nod;

// printf("\n%d ",nod);

}

else {

min=a[b][i+1];

nod=i+1;

v1++;

}

}

if(v1==n)

{

v[k]=1;

min=a[b][1];

}

k++;

k1=k;

sum+=min;

if(v[k-1]==1 ){

return v[k-1];

}

else {

return tra(v[k-1]);

}

}

int main()

{

int i,j,t;

printf("\nenter the no of cities : ");

scanf("%d",&n);

v[0]=1;

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

{

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

{

printf("\nenter the cost from %d to %d : ",i,j);

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

}

}

t=tra(1);

for(i=0;i<k1;i++)

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

printf("\ntotal cost : %d",sum);

}

Armstrong number :

#include<stdio.h>

int main()

{

int n,i,rem,sum=0,k;

printf("\nenter the num : ");

scanf("%d",&n);

k=n;

while(n!=0)

{

rem=n%10;

sum+=(rem)\*(rem)\*(rem);

n/=10;

}

if(sum==k)

printf("\nit is armstrong ");

else

printf("\nit is not ");

}

Factorial of a number :

#include<stdio.h>

int main()

{

int n,f=1,i;

printf("\nenter the num : ");

scanf("%d",&n);

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

{

f=f\*i;

}

printf("%d",f);

}

Fibonacci series :

#include<stdio.h>

int main()

{

int a=0,b=1,i,n,c;

printf("\nenter the num : ");

scanf("%d",&n);

printf("%d %d ",a,b);

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

{

c=a+b;

printf("%d ",c);

a=b;

b=c;

}

}

Min and max in a tree :

#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node \*left, \*right;

};

struct Node\* newNode(int data)

{

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

node->data = data;

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

return (node);

}

int findMax(struct Node\* root)

{

if (root == NULL)

return INT\_MIN;

int res = root->data;

int lres = findMax(root->left);

int rres = findMax(root->right);

if (lres > res)

res = lres;

if (rres > res)

res = rres;

return res;

}

int findMin(struct Node\* root)

{

if (root == NULL)

return INT\_MAX;

int res = root->data;

int lres = findMin(root->left);

int rres = findMin(root->right);

if (lres < res)

res = lres;

if (rres < res)

res = rres;

return res;

}

int main(void)

{

struct Node\* NewRoot = NULL;

struct Node\* root = newNode(12);

root->left = newNode(72);

root->right = newNode(32);

root->left->right = newNode(94);

root->left->right->left = newNode(30);

root->left->right->right = newNode(45);

root->right->right = newNode(34);

root->right->right->left= newNode(1);

printf("Maximum element is %d \n", findMax(root));

printf("Minimum element is %d \n", findMin(root));

return 0;

}

Prime number :

#include<stdio.h>

int main()

{

int i,n,p=0;

printf("\nenter the num : ");

scanf("%d",&n);

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

{

if(n%i==0)

p++;

}

if(p==1)

printf("\nit is prime %d");

else

printf("\nit is not ");

}

Sum of digits :

#include<stdio.h>

int main()

{

int n,rem,sum=0,i;

printf("\nenter the num : ");

scanf("%d",&n);

while(n!=0)

{

rem=n%10;

sum+=rem;

n/=10;

}

printf("%d",sum);

}

Sum of rows and column :

#include <stdio.h>

int main() {

int rows, cols;

// Input the number of rows and columns

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

scanf("%d", &rows);

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

scanf("%d", &cols);

int matrix[rows][cols];

// Input the matrix elements

printf("Enter the matrix elements:\n");

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

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

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

}

}

// Calculate and display the sum of rows

printf("Sum of rows:\n");

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

int rowSum = 0;

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

rowSum += matrix[i][j];

}

printf("Row %d: %d\n", i + 1, rowSum);

}

// Calculate and display the sum of columns

printf("Sum of columns:\n");

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

int colSum = 0;

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

colSum += matrix[i][j];

}

printf("Column %d: %d\n", j + 1, colSum);

}

return 0;

}

Tree traverse :

#include <stdio.h>

#include <stdlib.h>

// Definition of a tree node

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new tree node

struct TreeNode\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// In-order traversal

void inOrder(struct TreeNode\* root) {

if (root == NULL) return;

inOrder(root->left);

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

inOrder(root->right);

}

// Pre-order traversal

void preOrder(struct TreeNode\* root) {

if (root == NULL) return;

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

preOrder(root->left);

preOrder(root->right);

}

// Post-order traversal

void postOrder(struct TreeNode\* root) {

if (root == NULL) return;

postOrder(root->left);

postOrder(root->right);

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

}

int main() {

struct TreeNode\* root = createNode(1);

root->left = createNode(2);

root->right = createNode(3);

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

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

printf("In-order traversal: ");

inOrder(root);

printf("\n");

printf("Pre-order traversal: ");

preOrder(root);

printf("\n");

printf("Post-order traversal: ");

postOrder(root);

printf("\n");

return 0;

}

Stack using two queues :

#include <stdio.h>

#include <stdlib.h>

// Structure for a Queue Node

struct QueueNode {

int data;

struct QueueNode\* next;

};

// Structure for a Queue

struct Queue {

struct QueueNode\* front;

struct QueueNode\* rear;

};

// Create a new QueueNode

struct QueueNode\* createQueueNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Initialize a new Queue

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->front = NULL;

queue->rear = NULL;

return queue;

}

// Enqueue an element into the Queue

void enqueue(struct Queue\* queue, int data) {

struct QueueNode\* newNode = createQueueNode(data);

if (queue->rear == NULL) {

queue->front = newNode;

queue->rear = newNode;

} else {

queue->rear->next = newNode;

queue->rear = newNode;

}

}

// Dequeue an element from the Queue

int dequeue(struct Queue\* queue) {

if (queue->front == NULL) {

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

return -1; // You can choose a different value to represent an empty queue

}

int data = queue->front->data;

struct QueueNode\* temp = queue->front;

queue->front = queue->front->next;

free(temp);

if (queue->front == NULL) {

queue->rear = NULL;

}

return data;

}

// Stack structure using two queues

struct Stack {

struct Queue\* q1;

struct Queue\* q2;

};

// Initialize a new stack

struct Stack\* createStack() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->q1 = createQueue();

stack->q2 = createQueue();

return stack;

}

// Push an element onto the stack

void push(struct Stack\* stack, int data) {

enqueue(stack->q1, data);

}

// Pop an element from the stack

int pop(struct Stack\* stack) {

if (stack->q1->front == NULL) {

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

return -1; // You can choose a different value to represent an empty stack

}

// Move elements from q1 to q2, leaving the last element in q1

while (stack->q1->front->next != NULL) {

enqueue(stack->q2, dequeue(stack->q1));

}

// Pop the last element from q1

int data = dequeue(stack->q1);

// Swap q1 and q2 to maintain the original state

struct Queue\* temp = stack->q1;

stack->q1 = stack->q2;

stack->q2 = temp;

return data;

}

int main() {

struct Stack\* stack = createStack();

push(stack, 1);

push(stack, 2);

push(stack, 3);

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

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

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

printf("Popped: %d\n", pop(stack)); // Stack is empty now

return 0;

}

Queue using two stack :

#include <stdio.h>

#include <stdlib.h>

// Structure for a Stack Node

struct StackNode {

int data;

struct StackNode\* next;

};

// Structure for a Stack

struct Stack {

struct StackNode\* top;

};

// Create a new StackNode

struct StackNode\* createStackNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Initialize a new Stack

struct Stack\* createStack() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->top = NULL;

return stack;

}

// Push an element onto the stack

void push(struct Stack\* stack, int data) {

struct StackNode\* newNode = createStackNode(data);

newNode->next = stack->top;

stack->top = newNode;

}

// Pop an element from the stack

int pop(struct Stack\* stack) {

if (stack->top == NULL) {

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

return -1; // You can choose a different value to represent an empty stack

}

int data = stack->top->data;

struct StackNode\* temp = stack->top;

stack->top = stack->top->next;

free(temp);

return data;

}

// Structure for a Queue using two stacks

struct Queue {

struct Stack\* stack1; // For enqueue operation

struct Stack\* stack2; // For dequeue operation

};

// Initialize a new queue

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->stack1 = createStack();

queue->stack2 = createStack();

return queue;

}

// Enqueue an element into the queue

void enqueue(struct Queue\* queue, int data) {

// Push all elements from stack2 to stack1

while (queue->stack2->top != NULL) {

push(queue->stack1, pop(queue->stack2));

}

// Push the new element onto stack1

push(queue->stack1, data);

}

// Dequeue an element from the queue

int dequeue(struct Queue\* queue) {

if (queue->stack1->top == NULL && queue->stack2->top == NULL) {

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

return -1; // You can choose a different value to represent an empty queue

}

// Push all elements from stack1 to stack2

while (queue->stack1->top != NULL) {

push(queue->stack2, pop(queue->stack1));

}

// Pop and return the front element from stack2

return pop(queue->stack2);

}

int main() {

struct Queue\* queue = createQueue();

enqueue(queue, 1);

enqueue(queue, 2);

enqueue(queue, 3);

printf("Dequeued: %d\n", dequeue(queue));

printf("Dequeued: %d\n", dequeue(queue));

printf("Dequeued: %d\n", dequeue(queue));

printf("Dequeued: %d\n", dequeue(queue)); // Queue is empty now

return 0;

}

Print no of nodes in a given linked list :

#include <stdio.h>

#include <stdlib.h>

// Structure for 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 count the number of nodes in the linked list

int countNodes(struct Node\* head) {

int count = 0;

struct Node\* current = head;

while (current != NULL) {

count++;

current = current->next;

}

return count;

}

int main() {

// Create a linked list

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);

// Count the number of nodes

int count = countNodes(head);

printf("Number of nodes in the linked list: %d\n", count);

return 0;

}

Merge sort :

#include <stdio.h>

#include <stdlib.h>

// Merge two sorted subarrays into a single sorted subarray

void merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[left + i];

for (int j = 0; j < n2; j++)

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

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

// Merge Sort function

void mergeSort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

// Function to print an array

void printArray(int arr[], int size) {

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

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

printf("\n");

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int arr\_size = sizeof(arr) / sizeof(arr[0]);

printf("Original array: \n");

printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

printf("Sorted array: \n");

printArray(arr, arr\_size);

return 0;

}

Given 2 D matrix print largest element :

#include <stdio.h>

int main() {

int rows, cols;

// Input the number of rows and columns

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

scanf("%d", &rows);

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

scanf("%d", &cols);

int matrix[rows][cols];

// Input the matrix elements

printf("Enter the matrix elements:\n");

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

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

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

}

}

// Find the largest element

int largest = matrix[0][0];

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

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

if (matrix[i][j] > largest) {

largest = matrix[i][j];

}

}

}

// Print the largest element

printf("The largest element in the matrix is: %d\n", largest);

return 0;

}

Given a string - sort in alphabetical order :

#include <stdio.h>

int main() {

int rows, cols;

// Input the number of rows and columns

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

scanf("%d", &rows);

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

scanf("%d", &cols);

int matrix[rows][cols];

// Input the matrix elements

printf("Enter the matrix elements:\n");

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

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

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

}

}

// Find the largest element

int largest = matrix[0][0];

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

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

if (matrix[i][j] > largest) {

largest = matrix[i][j];

}

}

}

// Print the largest element

printf("The largest element in the matrix is: %d\n", largest);

return 0;

}

Print the index of repeated characters given in an array :

#include <stdio.h>

#include <stdbool.h>

int main() {

int size;

// Input the size of the array

printf("Enter the size of the array: ");

scanf("%d", &size);

char array[size];

// Input the array elements

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

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

scanf(" %c", &array[i]); // Add space before %c to consume newline

}

bool found[256] = {false}; // Assuming ASCII characters

printf("Repeated characters and their indices:\n");

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

if (found[array[i]]) {

printf("Character '%c' at index %d\n", array[i], i);

} else {

found[array[i]] = true;

}

}

return 0;

}

Given a string - sort in alphabetical order :

#include <stdio.h>

#include <string.h>

int main() {

char str[100];

// Input the string

printf("Enter a string: ");

scanf("%s", str);

int n = strlen(str);

// Bubble Sort

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

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

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

char temp = str[j];

str[j] = str[j + 1];

str[j + 1] = temp;

}

}

}

// Print the sorted string

printf("Sorted string in alphabetical order: %s\n", str);

return 0;

}

Avl tree :

#include <stdio.h>

#include <stdlib.h>

struct Node {

int key;

struct Node\* left;

struct Node\* right;

int height;

};

int max(int a, int b) {

return (a > b) ? a : b;

}

int height(struct Node\* node) {

if (node == NULL)

return 0;

return node->height;

}

int getBalance(struct Node\* node) {

if (node == NULL)

return 0;

return height(node->left) - height(node->right);

}

struct Node\* newNode(int key) {

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

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1; // New node is initially added at leaf

return node;

}

struct Node\* rotateRight(struct Node\* y) {

struct Node\* x = y->left;

struct Node\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

struct Node\* rotateLeft(struct Node\* x) {

struct Node\* y = x->right;

struct Node\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

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

if (node == NULL)

return newNode(key);

if (key < node->key)

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

else if (key > node->key)

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

else // Duplicate keys are not allowed

return node;

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

int balance = getBalance(node);

// Left Left Case

if (balance > 1 && key < node->left->key)

return rotateRight(node);

// Right Right Case

if (balance < -1 && key > node->right->key)

return rotateLeft(node);

// Left Right Case

if (balance > 1 && key > node->left->key) {

node->left = rotateLeft(node->left);

return rotateRight(node);

}

// Right Left Case

if (balance < -1 && key < node->right->key) {

node->right = rotateRight(node->right);

return rotateLeft(node);

}

return node;

}

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

root = insert(root, 25);

printf("Inorder traversal of the AVL tree: ");

inorder(root);

printf("\n");

return 0;

}

Element twice repeated :

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

// Function to find elements repeated twice in an array

void findElementsRepeatedTwice(int arr[], int size) {

int hash[MAX\_SIZE] = {0};

printf("Elements repeated twice in the array:\n");

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

hash[arr[i]]++;

if (hash[arr[i]] == 2) {

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

}

}

printf("\n");

}

int main() {

int size;

// Input the size of the array

printf("Enter the size of the array: ");

scanf("%d", &size);

int arr[MAX\_SIZE];

// Input the array elements

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

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

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

}

findElementsRepeatedTwice(arr, size);

return 0;

}

Given array of reg nos need to search for particular reg no :

#include <stdio.h>

#define MAX\_REGNOS 100

int searchRegNo(int regNos[], int size, int target) {

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

if (regNos[i] == target) {

return i; // Return the index where the reg no was found

}

}

return -1; // Return -1 if the reg no was not found

}

int main() {

int size, target;

// Input the size of the array

printf("Enter the number of registration numbers: ");

scanf("%d", &size);

int regNos[MAX\_REGNOS];

// Input the registration numbers

printf("Enter the registration numbers:\n");

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

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

}

// Input the target registration number to search for

printf("Enter the registration number to search for: ");

scanf("%d", &target);

int index = searchRegNo(regNos, size, target);

if (index != -1) {

printf("Registration number %d found at index %d\n", target, index);

} else {

printf("Registration number %d not found\n", target);

}

return 0;

}

Kruskal’s algorithm :

#include <stdio.h>

#include <stdlib.h>

// Structure to represent an edge

struct Edge {

int src, dest, weight;

};

// Structure to represent a subset for union-find

struct Subset {

int parent, rank;

};

// Function to compare two edges for sorting

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

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

}

// Find the subset of an element using path compression

int find(struct Subset subsets[], int i) {

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// Perform union of two subsets using union by rank

void unionSets(struct Subset subsets[], int x, int y) {

int xroot = find(subsets, x);

int yroot = find(subsets, y);

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

else {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

// Kruskal's algorithm to find minimum spanning tree

void kruskalMST(struct Edge edges[], int V, int E) {

struct Subset\* subsets = (struct Subset\*)malloc(V \* sizeof(struct Subset));

struct Edge\* result = (struct Edge\*)malloc((V - 1) \* sizeof(struct Edge));

qsort(edges, E, sizeof(edges[0]), compareEdges);

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

subsets[i].parent = i;

subsets[i].rank = 0;

}

int mstSize = 0, i = 0;

while (mstSize < V - 1) {

struct Edge nextEdge = edges[i++];

int x = find(subsets, nextEdge.src);

int y = find(subsets, nextEdge.dest);

if (x != y) {

result[mstSize++] = nextEdge;

unionSets(subsets, x, y);

}

}

printf("Minimum Spanning Tree edges:\n");

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

printf("%d - %d: %d\n", result[i].src, result[i].dest, result[i].weight);

}

free(subsets);

free(result);

}

int main() {

int V, E;

// Input the number of vertices and edges

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

scanf("%d", &V);

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

scanf("%d", &E);

struct Edge\* edges = (struct Edge\*)malloc(E \* sizeof(struct Edge));

// Input the edges and their weights

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

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

scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);

}

kruskalMST(edges, V, E);

free(edges);

return 0;

}

Prims algorithm :

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#include <limits.h>

#define V 5 // Number of vertices

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

int min = INT\_MAX, min\_index;

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

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

min = key[v];

min\_index = v;

}

}

return min\_index;

}

void printMST(int parent[], int graph[V][V]) {

printf("Minimum Spanning Tree edges:\n");

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

printf("%d - %d: %d\n", parent[i], i, graph[i][parent[i]]);

}

}

void primMST(int graph[V][V]) {

int parent[V]; // Array to store the constructed MST

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

bool mstSet[V]; // To represent set of vertices included in MST

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

key[i] = INT\_MAX;

mstSet[i] = false;

}

key[0] = 0; // Starting vertex

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

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

int u = minKey(key, mstSet);

mstSet[u] = true;

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

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

parent[v] = u;

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

}

}

}

printMST(parent, graph);

}

int main() {

int graph[V][V] = {

{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0}

};

primMST(graph);

return 0;

}