### 1. Reversing a 32 bit signed integers.

#include <stdio.h>

int reverse(int x) {

int result = 0;

while(x != 0) {

result = result \* 10 + x % 10;

x /= 10;

}

return result;

}

int main() {

int x = -12345;

printf("%d\n", reverse(x));

return 0;

}

### 2. Check for a valid String.

#include <stdio.h>

int isValidString(char str[]) {

int i;

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

return 0;

}

}

return 1;

}

int main() {

char str[100];

printf("Enter a string: ");

scanf("%[^\n]%\*c", str);

if(isValidString(str)) {

printf("Valid string\n");

} else {

printf("Invalid string\n");

}

return 0;

}

### 3. Merging two Arrays.

#include <stdio.h>

void mergeArrays(int arr1[], int m, int arr2[], int n, int arr3[]) {

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

while (i < m && j < n) {

if (arr1[i] < arr2[j]) {

arr3[k++] = arr1[i++];

} else {

arr3[k++] = arr2[j++];

}

}

while (i < m) {

arr3[k++] = arr1[i++];

}

while (j < n) {

arr3[k++] = arr2[j++];

}

}

int main() {

int arr1[] = {1, 3, 5, 7};

int m = sizeof(arr1) / sizeof(arr1[0]);

int arr2[] = {2, 4, 6, 8};

int n = sizeof(arr2) / sizeof(arr2[0]);

int arr3[m + n];

mergeArrays(arr1, m, arr2, n, arr3);

printf("Merged array: ");

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

printf("%d ", arr3[i]);

}

return 0;

}

### 4. Given an array finding duplication values.

#include <stdio.h>

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

int i, j;

printf("Duplicate elements: ");

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

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

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

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

break;

}

}

}

}

int main() {

int arr[] = {1, 2, 3, 4, 2, 3, 5, 6, 7, 8, 9, 5};

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

findDuplicates(arr, n);

return 0;

}

### 5. Merging of list.

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* next;

} Node;

Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

Node\* mergeLists(Node\* head1, Node\* head2) {

Node\* dummyNode = createNode(0);

Node\* current = dummyNode;

while (head1 != NULL && head2 != NULL) {

if (head1->data < head2->data) {

current->next = head1;

head1 = head1->next;

} else {

current->next = head2;

head2 = head2->next;

}

current = current->next;

}

if (head1 != NULL) {

current->next = head1;

} else {

current->next = head2;

}

return dummyNode->next;

}

void printList(Node\* head) {

while (head != NULL) {

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

head = head->next;

}

printf("NULL\n");

}

int main() {

Node\* head1 = createNode(1);

head1->next = createNode(3);

head1->next->next = createNode(5);

Node\* head2 = createNode(2);

head2->next = createNode(4);

head2->next->next = createNode(6);

printf("Linked List 1: ");

printList(head1);

printf("Linked List 2: ");

printList(head2);

Node\* mergedHead = mergeLists(head1, head2);

printf("Merged Linked List: ");

printList(mergedHead);

return 0;

}

### 6. Given array of reg nos need to search for particular reg no.

#include <stdio.h>

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

int i;

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

if (regNos[i] == target) {

return i;

}

}

return -1;

}

int main() {

int regNos[] = {123, 456, 789, 101, 202};

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

int target = 789;

int result = searchRegNo(regNos, n, target);

if (result != -1) {

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

} else {

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

}

return 0;

}

### 7. Identify location of element in given array.

#include <stdio.h>

int findElement(int arr[], int n, int target) {

int i;

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

if (arr[i] == target) {

return i;

}

}

return -1;

}

int main() {

int arr[] = {10, 20, 30, 40, 50};

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

int target = 30;

int result = findElement(arr, n, target);

if (result != -1) {

printf("Element %d found at index %d\n", target, result);

} else {

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

}

return 0;

}

### 8. Given array print odd and even values.

#include <stdio.h>

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

printf("Odd values: ");

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

if (arr[i] % 2 != 0) {

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

}

}

printf("\nEven values: ");

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

if (arr[i] % 2 == 0) {

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

}

}

}

int main() {

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

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

printOddEven(arr, n);

return 0;

}

### 9.sum of Fibonacci Series.

#include <stdio.h>

int fibonacci(int n) {

int a = 0, b = 1, sum = 0;

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

sum += a;

int temp = a;

a = b;

b = temp + b;

}

return sum;

}

int main() {

int n;

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

scanf("%d", &n);

printf("Sum of Fibonacci Series: %d\n", fibonacci(n));

return 0;

}

### 10. Finding factorial of a number.

#include <stdio.h>

long long factorial(int n) {

long long fact = 1;

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

fact \*= i;

}

return fact;

}

int main() {

int n;

printf("Enter a number: ");

scanf("%d", &n);

printf("Factorial of %d: %lld\n", n, factorial(n));

return 0;

}

### 11. AVL tree.

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int key;

struct Node\* left;

struct Node\* right;

int height;

} Node;

Node\* createNode(int key) {

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

newNode->key = key;

newNode->left = NULL;

newNode->right = NULL;

newNode->height = 1;

return newNode;

}

int getHeight(Node\* node) {

if (node == NULL) {

return 0;

}

return node->height;

}

void updateHeight(Node\* node) {

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

}

int getBalance(Node\* node) {

if (node == NULL) {

return 0;

}

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

}

Node\* leftRotate(Node\* node) {

Node\* temp = node->right;

node->right = temp->left;

temp->left = node;

updateHeight(node);

updateHeight(temp);

return temp;

}

Node\* rightRotate(Node\* node) {

Node\* temp = node->left;

node->left = temp->right;

temp->right = node;

updateHeight(node);

updateHeight(temp);

return temp;

}

Node\* rebalance(Node\* node) {

int balance = getBalance(node);

if (balance > 1) {

if (getHeight(node->left->left) >= getHeight(node->left->right)) {

node = rightRotate(node);

} else {

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

node = rightRotate(node);

}

} else if (balance < -1) {

if (getHeight(node->right->right) >= getHeight(node->right->left)) {

node = leftRotate(node);

} else {

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

node = leftRotate(node);

}

}

return node;

}

Node\* insertNode(Node\* node, int key) {

if (node == NULL) {

return createNode(key);

}

if (key < node->key) {

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

} else if (key > node->key) {

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

} else {

return node;

}

updateHeight(node);

return rebalance(node);

}

Node\* deleteNode(Node\* node, int key) {

if (node == NULL) {

return node;

}

if (key < node->key) {

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

} else if (key > node->key) {

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

} else {

if (node->left == NULL) {

Node\* temp = node->right;

free(node);

return temp;

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

Node\* temp = node->left;

free(node);

return temp;

}

Node\* temp = node->right;

while (temp->left != NULL) {

temp = temp->left;

}

node->key = temp->key;

node->right = deleteNode(node->right, temp->key);

}

updateHeight(node);

return rebalance(node);

}

Node\* searchNode(Node\* node, int key) {

if (node == NULL || node->key == key) {

return node;

}

if (key < node->key) {

return searchNode(node->left, key);

}

return searchNode(node->right, key);

}

void printTree(Node\* node) {

if (node == NULL) {

return;

}

printTree(node->left);

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

printTree(node->right);

}

int main() {

Node\* root = NULL;

root = insertNode(root, 10);

root = insertNode(root, 20);

root = insertNode(root, 30);

root = insertNode(root, 40);

root = insertNode(root, 50);

printTree(root);

return 0;

}

### 12. Valid stack.

#include <stdio.h>

#include <stdlib.h>

struct Stack {

int top;

int capacity;

int\* array;

};

struct Stack\* createStack(int capacity) {

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

stack->capacity = capacity;

stack->top = -1;

stack->array = (int\*)malloc(stack->capacity \* sizeof(int));

return stack;

}

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

int isFull(struct Stack\* stack) {

return stack->top == stack->capacity - 1;

}

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

if (isFull(stack)) {

printf("Stack is full\n");

return;

}

stack->array[++stack->top] = item;

}

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty\n");

return -1;

}

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

}

int isValidStack(struct Stack\* stack) {

int prev = pop(stack);

while (!isEmpty(stack)) {

int curr = pop(stack);

if (curr > prev) {

return 0;

}

prev = curr;

}

return 1;

}

int main() {

struct Stack\* stack = createStack(5);

push(stack, 1);

push(stack, 2);

push(stack, 3);

push(stack, 4);

push(stack, 5);

printf("%d\n", isValidStack(stack));

return 0;

}

### 13. Graph - shortest path.

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define V 6

int minDistance(int dist[], int visited[]) {

int min = INT\_MAX, min\_index;

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

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

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

void printPath(int parent[], int j) {

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

printf("%d ", j);

return;

}

printPath(parent, parent[j]);

printf("%d ", j);

}

void dijkstra(int graph[V][V], int src) {

int dist[V];

int visited[V];

int parent[V];

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

dist[i] = INT\_MAX;

visited[i] = 0;

parent[i] = -1;

}

dist[src] = 0;

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

int u = minDistance(dist, visited);

visited[u] = 1;

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

dist[v] = dist[u] + graph[u][v];

parent[v] = u;

}

}

}

printf("Vertex\tDistance\tPath\n");

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

printf("%d\t%d\t", i, dist[i]);

printPath(parent, i);

printf("\n");

}

}

int main() {

int graph[V][V] = {

{0, 4, 0, 0, 0, 0},

{4, 0, 8, 0, 0, 0},

{0, 8, 0, 7, 0, 4},

{0, 0, 7, 0, 9, 14},

{0, 0, 0, 9, 0, 10},

{0, 0, 4, 14, 10, 0}

};

dijkstra(graph, 0);

return 0;

}

### 14. Traveling Salesman Problem.

### The Traveling Salesman Problem (TSP) is an NP-hard problem in combinatorial optimization and operations research that is important in theoretical computer science and operations research. Here is a C code to solve TSP using the Nearest Neighbor algorithm:

#include <stdio.h>

#include <stdlib.h>

#define V 5

int distance[V][V] = {

{0, 10, 15, 20, 25},

{10, 0, 35, 30, 20},

{15, 35, 0, 25, 18},

{20, 30, 25, 0, 22},

{25, 20, 18, 22, 0}

};

int nearestNeighbor(int start) {

int visited[V];

int current = start;

int totalDistance = 0;

int i;

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

visited[i] = 0;

}

visited[current] = 1;

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

int minDistance = INT\_MAX;

int next;

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

if (!visited[j] && distance[current][j] < minDistance) {

minDistance = distance[current][j];

next = j;

}

}

totalDistance += minDistance;

current = next;

visited[current] = 1;

}

totalDistance += distance[current][start];

return totalDistance;

}

int main() {

int start = 0;

printf("Total distance: %d\n", nearestNeighbor(start));

return 0;

}

### 15.! Binary search tree - search for a element, min element and Max element.

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

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

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

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

} else if (data > root->data) {

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

}

return root;

}

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

if (root == NULL || root->data == data) {

return root;

}

if (data < root->data) {

return searchNode(root->left, data);

} else {

return searchNode(root->right, data);

}

}

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

while (root->left != NULL) {

root = root->left;

}

return root;

}

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

while (root->right != NULL) {

root = root->right;

}

return root;

}

int main() {

struct Node\* root = NULL;

root = insertNode(root, 50);

root = insertNode(root, 30);

root = insertNode(root, 20);

root = insertNode(root, 40);

root = insertNode(root, 70);

root = insertNode(root, 60);

root = insertNode(root, 80);

struct Node\* searchedNode = searchNode(root, 40);

if (searchedNode != NULL) {

printf("Element found: %d\n", searchedNode->data);

} else {

printf("Element not found\n");

}

struct Node\* minNode = findMinNode(root);

printf("Minimum element: %d\n", minNode->data);

struct Node\* maxNode = findMaxNode(root);

printf("Maximum element: %d\n", maxNode->data);

return 0;

}