1.Reversing a 32 bit signed intergers:

#include <stdio.h>

int reverse(int x) {

int reversed = 0

while (x != 0) {

int digit = x % 10;

x /= 10;

if (reversed > INT\_MAX/10 || (reversed == INT\_MAX/10 && digit > 7)) return 0;

if (reversed < INT\_MIN/10 || (reversed == INT\_MIN/10 && digit < -8)) return 0;

reversed = reversed \* 10 + digit;

}

return reversed;

}

int main() {

int num = -123456789;

int reversed\_num = reverse(num);

printf("Reversed number of %d is: %d\n", num, reversed\_num);

return 0;

}

2.Check for a valid String:

#include <stdio.h>

#include <string.h>

int isValidString(char \*str) {

for (int i = 0; i < strlen(str); i++) {

if (!(str[i] >= 'a' && str[i] <= 'z') && !(str[i] >= 'A' && str[i] <= 'Z')) {

return 0;

}

}

return 1;

}

int main() {

char str[] = "ValidString";

if (isValidString(str)) {

printf("The string is valid.\n");

} else {

printf("The string is not valid.\n");

}

return 0;

}

3. Merging two Arrays:

#include <stdio.h>

int main() {

int arr1[] = {1, 2, 3, 4, 5};

int arr2[] = {6, 7, 8, 9, 10};

int merged[10];

int i, j;

for (i = 0; i < 5; i++) {

merged[i] = arr1[i];

}

for (j = 0; j < 5; j++) {

merged[i + j] = arr2[j];

}

printf("Merged Array: ");

for (i = 0; i < 10; i++) {

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

}

return 0;

}

4. Given an array finding duplication values:

#include <stdio.h>

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

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

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

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

printf("Duplicate: %d\n", arr[j]);

}

}

}

}

int main() {

int arr[] = {4, 2, 7, 3, 2, 7, 8, 4};

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

findDuplicates(arr, size);

return 0;

}

5. Merging of list:

#include <stdio.h>

#include <stdlib.h>

int main() {

int list1[] = {1, 3, 5, 7, 9};

int list2[] = {2, 4, 6, 8, 10};

int mergedList[10];

int i, j, k;

for (i = 0; i < 5; i++) {

mergedList[i] = list1[i];

}

for (j = 0; j < 5; j++) {

mergedList[i + j] = list2[j];

}

printf("Merged List: ");

for (k = 0; k < 10; k++) {

printf("%d ", mergedList[k]);

}

return 0;

}

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

#include <stdio.h>

int main() {

int regNos[] = {123, 456, 789, 1011, 1213};

int searchRegNo = 1011;

int found = 0;

for (int i = 0; i < 5; i++) {

if (regNos[i] == searchRegNo) {

found = 1;

break;

}

}

if (found) {

printf("Registration number %d found in the array.", searchRegNo);

} else {

printf("Registration number %d not found in the array.", searchRegNo);

}

return 0;

}

7. Identify location of element in given array:

#include <stdio.h>

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

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

if (arr[i] == key) {

return i;

}

}

return -1;

}

int main() {

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

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

int key = 30;

int index = findElement(arr, n, key);

if (index != -1) {

printf("Element found at index: %d", index);

} else {

printf("Element not found in the array");

}

return 0;

}

8. Given array print odd and even values:

#include <stdio.h>

int main() {

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

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

printf("Odd numbers: ");

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

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

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

}

}

printf("\nEven numbers: ");

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

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

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

}

}

return 0;

}

9. sum of Fibonacci Series:

#include <stdio.h>

int fibonacci(int n) {

if (n <= 1)

return n;

return fibonacci(n - 1) + fibonacci(n - 2);

}

int main() {

int n = 10; // Number of terms in the series

int sum = 0;

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

sum += fibonacci(i);

}

printf("Sum of the first %d terms of the Fibonacci series is: %d\n", n, sum);

return 0;

}

10. Finding factorial of a number:

#include <stdio.h>

int main() {

int number, i;

unsigned long long factorial = 1;

printf("Enter a positive integer: ");

scanf("%d", &number);

if (number < 0)

printf("Error! Factorial of a negative number doesn't exist.");

else {

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

factorial \*= i;

}

printf("Factorial of %d = %llu", number, factorial);

}

return 0;

}

11. 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);

int height(struct Node \*N);

struct Node \*newNode(int key);

struct Node \*rightRotate(struct Node \*y);

struct Node \*leftRotate(struct Node \*x);

int getBalance(struct Node \*N);

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

void preOrder(struct Node \*root);

int max(int a, int b) {

return (a > b) ? a : b;

}

int height(struct Node \*N) {

if (N == NULL)

return 0;

return N->height;

}

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;

return node;

}

struct Node \*rightRotate(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 \*leftRotate(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;

}

int getBalance(struct Node \*N) {

if (N == NULL)

return 0;

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

}

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

return node;

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

int balance = getBalance(node);

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

return rightRotate(node);

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

return leftRotate(node);

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

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

return rightRotate(node);

}

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

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

return leftRotate(node);

}

return node;

}

void preOrder(struct Node \*root) {

if (root != NULL) {

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

preOrder(root->left);

preOrder(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("Preorder traversal of the constructed AVL tree is \n");

preOrder(root);

return 0;

}

12. Valid stack:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

typedef struct {

int items[MAX\_SIZE];

int top;

} Stack;

void initialize(Stack \*stack) {

stack->top = -1;

}

int isEmpty(Stack \*stack) {

return stack->top == -1;

}

int isFull(Stack \*stack) {

return stack->top == MAX\_SIZE - 1;

}

void push(Stack \*stack, int value) {

if (isFull(stack)) {

printf("Stack Overflow\n");

return;

}

stack->items[++stack->top] = value;

}

int pop(Stack \*stack) {

if (isEmpty(stack)) {

printf("Stack Underflow\n");

return -1;

}

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

}

int peek(Stack \*stack) {

if (isEmpty(stack)) {

printf("Stack is empty\n");

return -1;

}

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

}

int main() {

Stack stack;

initialize(&stack);

push(&stack, 1);

push(&stack, 2);

push(&stack, 3);

printf("Top element: %d\n", peek(&stack));

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

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

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

printf("Is the stack empty? %s\n", isEmpty(&stack) ? "Yes" : "No");

return 0;

}

13. Graph - shortest path:

#include <stdio.h>

#include <limits.h>

#define V 9

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

int min = INT\_MAX, min\_index;

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

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

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

void printSolution(int dist[]) {

printf("Vertex \t Distance from Source\n");

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

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

}

}

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

int dist[V];

bool sptSet[V];

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

dist[i] = INT\_MAX;

sptSet[i] = false;

}

dist[src] = 0;

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

int u = minDistance(dist, sptSet);

sptSet[u] = true;

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

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]) {

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

}

}

}

printSolution(dist);

}

int main() {

int graph[V][V] = {

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

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

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

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

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}

};

dijkstra(graph, 0);

return 0;

}

14. Traveling Salesman Problem:

#include <stdio.h>

#define V 4

int tsp(int graph[][V], int s)

{

int vertex[V];

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

{

if (i != s)

vertex[i] = i;

}

int min\_path = INT\_MAX;

do

{

int current\_pathweight = 0;

int k = s;

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

{

current\_pathweight += graph[k][vertex[i]];

k = vertex[i];

}

current\_pathweight += graph[k][s];

min\_path = (min\_path > current\_pathweight) ? current\_pathweight : min\_path;

} while (next\_permutation(vertex, vertex + V));

return min\_path;

}

int main()

{

int graph[][V] = { {0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0} };

int s = 0;

printf("Minimum cost: %d\n", tsp(graph, s));

return 0;

}

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

struct Node\* search(struct Node\* root, int key) {

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

return root;

if (root->data < key)

return search(root->right, key);

return search(root->left, key);

}

// Find the minimum element in a Binary Search Tree

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

struct Node\* current = node;

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

current = current->left;

return current;

}

// Find the maximum element in a Binary Search Tree

struct Node\* maxValueNode(struct Node\* node) {

struct Node\* current = node;

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

current = current->right;

return current;

}