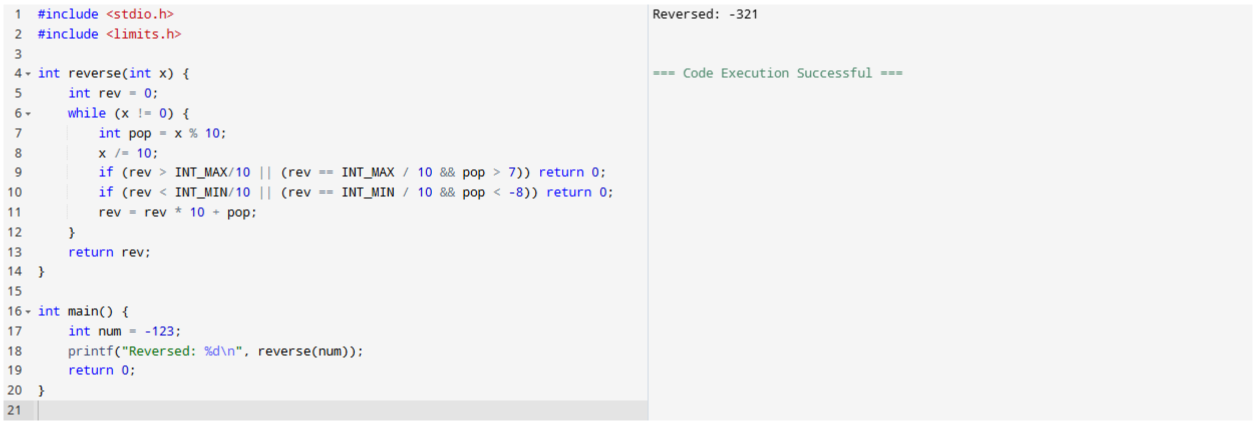
## 1. Reversing a 32 bit signed integers.

Aim: To reverse a 32-bit signed integer without overflow.



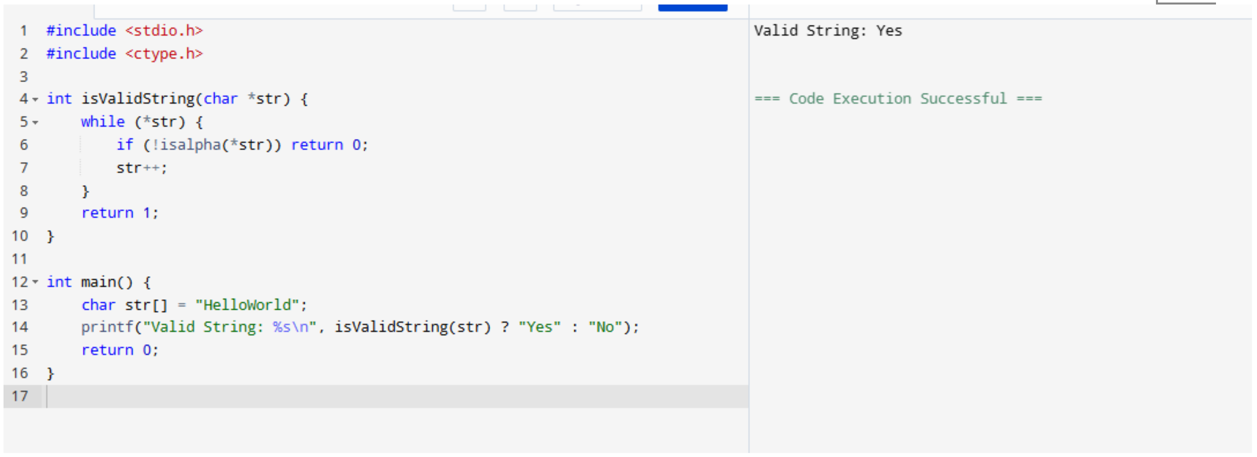
```c
#include <stdio.h>
#include <limits.h>

int reverse(int x) {
    int rev = 0;
    while (x != 0) {
        int pop = x % 10;
        x /= 10;
        if (rev > INT_MAX/10 || (rev == INT_MAX / 10 && pop > 7)) return 0;
        if (rev < INT_MIN/10 || (rev == INT_MIN / 10 && pop < -8)) return 0;
        rev = rev * 10 + pop;
    }
    return rev;
}

int main() {
    int num = -123;
    printf("Reversed: %d\n", reverse(num));
    return 0;
}
```

```
Reversed: -321

=== Code Execution Successful ===
```

Result: The program was successfully executed and the reverse of the given 32-bit signed integer was calculated and displayed.

## 2. Check for a valid String

Aim: To check if a string is valid (contains only alphabets).



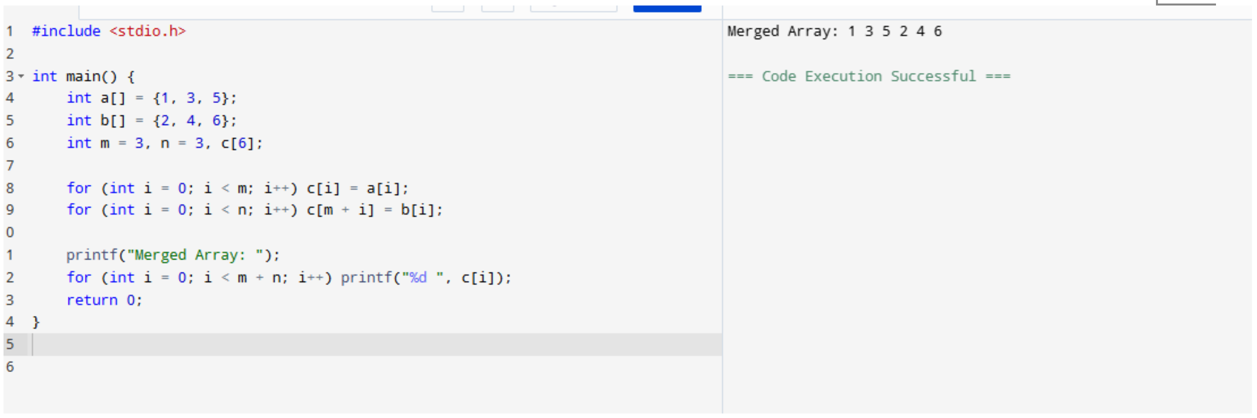
```c
#include <stdio.h>
#include <ctype.h>

int isValidString(char *str) {
    while (*str) {
        if (!isalpha(*str)) return 0;
        str++;
    }
    return 1;
}

int main() {
    char str[] = "HelloWorld";
    printf("Valid String: %s\n", isValidString(str) ? "Yes" : "No");
    return 0;
}
```

```
Valid String: Yes


=== Code Execution Successful ===
```

Result:
The program was successfully executed and it verified that the input string contains only valid alphabetic characters.

## 3. Merging two Arrays

Aim: To merge two arrays into one.



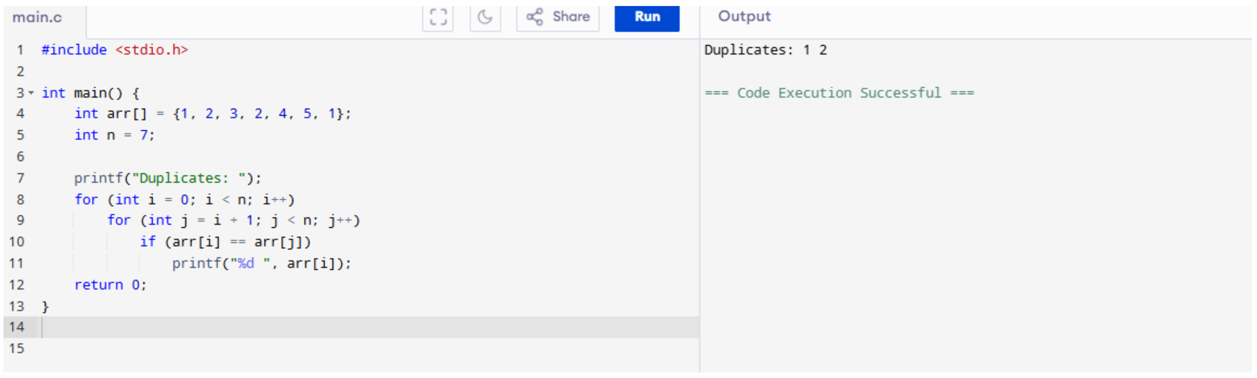
```c
#include <stdio.h>

int main() {
    int a[] = {1, 3, 5};
    int b[] = {2, 4, 6};
    int m = 3, n = 3, c[6];

    for (int i = 0; i < m; i++) c[i] = a[i];
    for (int i = 0; i < n; i++) c[m + i] = b[i];

    printf("Merged Array: ");
    for (int i = 0; i < m + n; i++) printf("%d ", c[i]);
    return 0;
}
```

```
Merged Array: 1 3 5 2 4 6

=== Code Execution Successful ===
```

Result:
The program was successfully executed and the two arrays were merged into a single array and displayed.

4. Given an array finding duplication values

Aim: To find and print duplicate elements in an array.



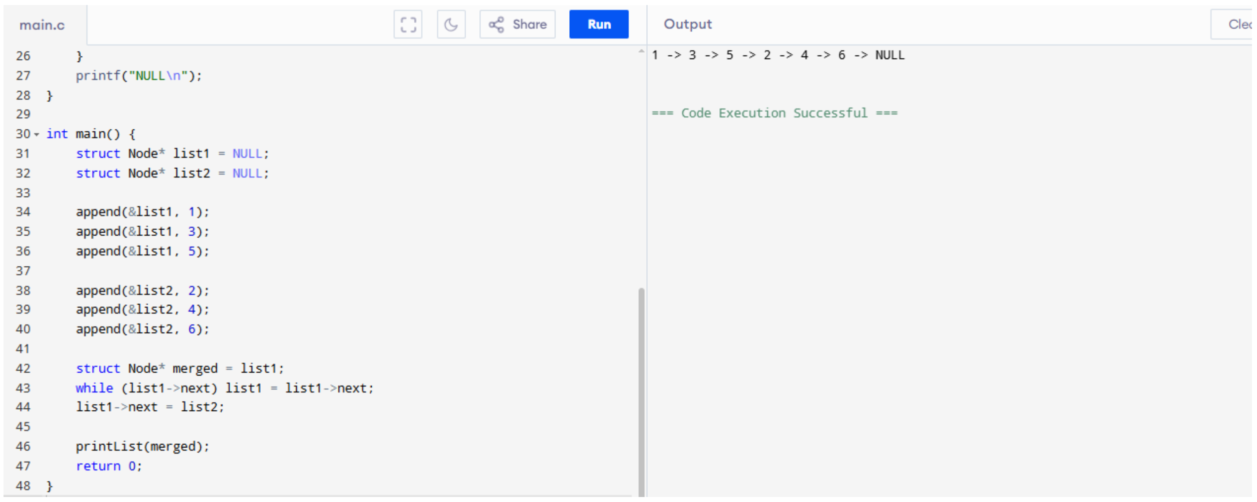
```c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 2, 4, 5, 1};
    int n = 7;

    printf("Duplicates: ");
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            if (arr[i] == arr[j])
                printf("%d ", arr[i]);
    return 0;
}
```

```
Duplicates: 1 2

=== Code Execution Successful ===
```

Result:
The program was successfully executed and the duplicate values in the given array were identified and displayed.

5. Merging of list

Aim: To merge two linked lists.



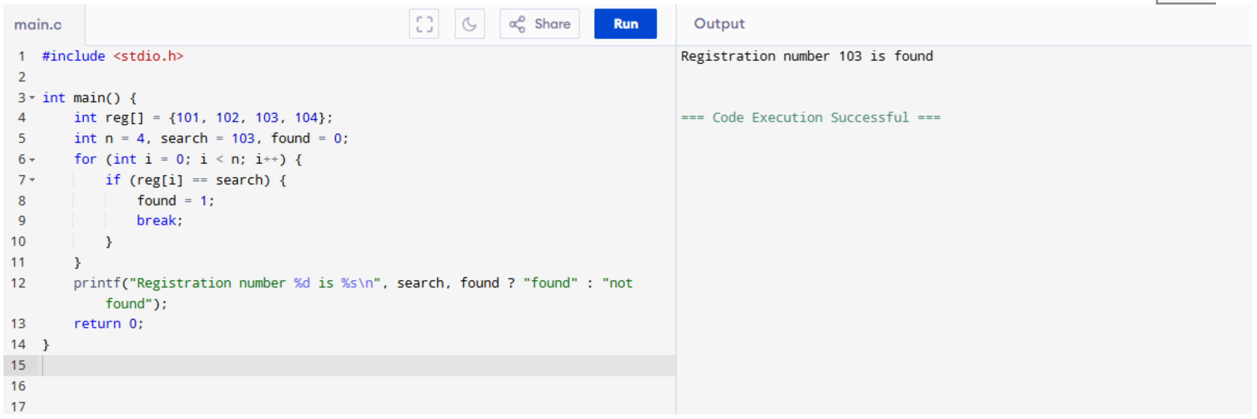
```c
    }
    printf("NULL\n");
}

int main() {
    struct Node* list1 = NULL;
    struct Node* list2 = NULL;

    append(&list1, 1);
    append(&list1, 3);
    append(&list1, 5);

    append(&list2, 2);
    append(&list2, 4);
    append(&list2, 6);

    struct Node* merged = list1;
    while (list1->next) list1 = list1->next;
    list1->next = list2;

    printList(merged);
    return 0;
}
```

```
1 -> 3 -> 5 -> 2 -> 4 -> 6 -> NULL


=== Code Execution Successful ===
```

Result:
The program was successfully executed and two linked lists were merged into one and the merged list was displayed.

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

Aim: To search for a given registration number in an array.



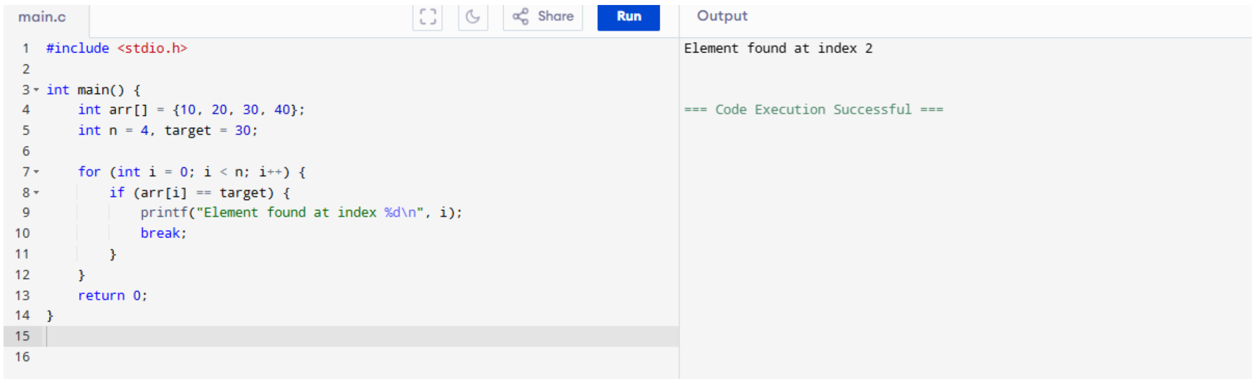
```c
#include <stdio.h>

int main() {
    int reg[] = {101, 102, 103, 104};
    int n = 4, search = 103, found = 0;
    for (int i = 0; i < n; i++) {
        if (reg[i] == search) {
            found = 1;
            break;
        }
    }
    printf("Registration number %d is %s\n", search, found ? "found" : "not
        found");
    return 0;
}
```

```
Registration number 103 is found


=== Code Execution Successful ===
```

Result:
The program was successfully executed and it successfully searched and confirmed the presence of the specified registration number in the array.

7. Identify location of element in given array

Aim:
To find the index of a given element in an array.



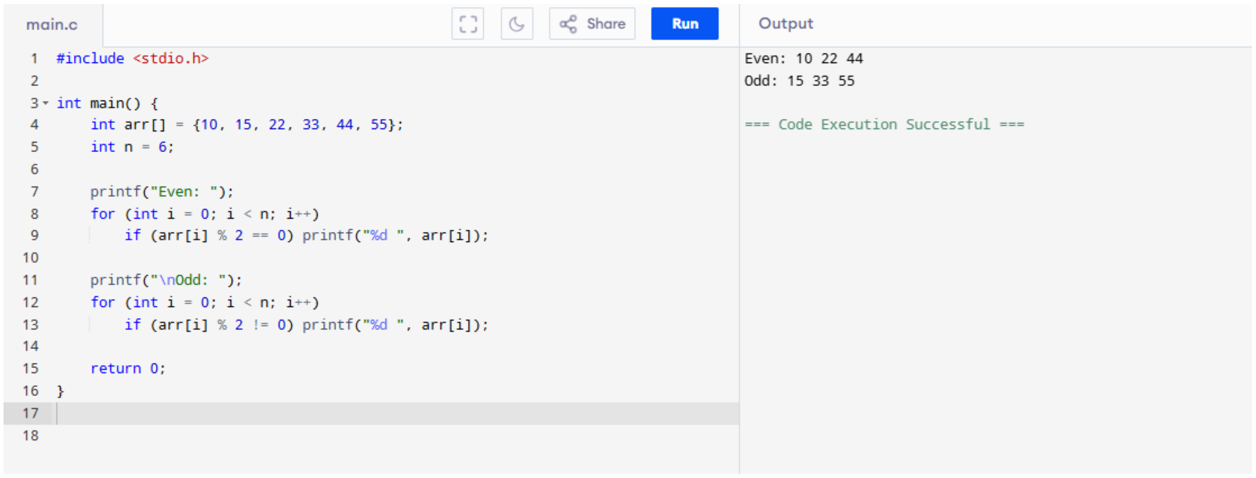
```c
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40};
    int n = 4, target = 30;

    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            printf("Element found at index %d\n", i);
            break;
        }
    }
    return 0;
}
```

```
Element found at index 2


=== Code Execution Successful ===
```

Result:
The program was successfully executed and the location (index) of the specified element in the array was identified and displayed.

## 8. Given array print odd and even values

Aim:
To separate and print odd and even numbers from an array.



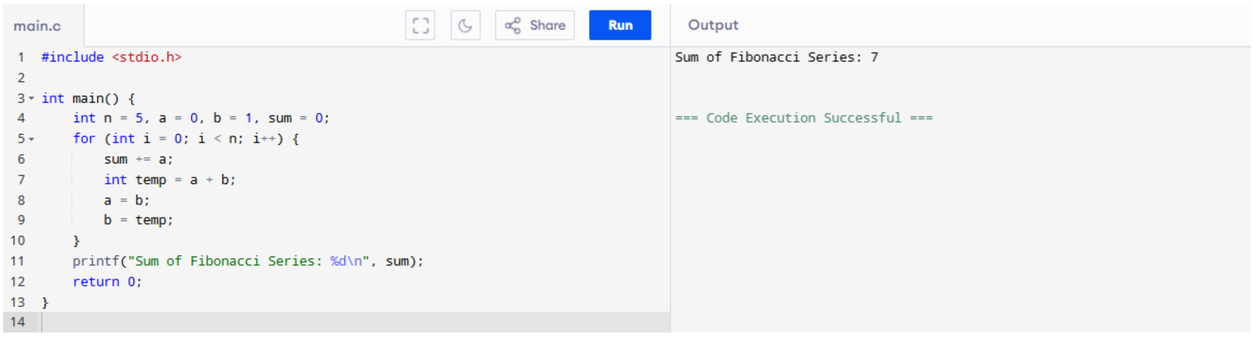
```c
#include <stdio.h>

int main() {
    int arr[] = {10, 15, 22, 33, 44, 55};
    int n = 6;

    printf("Even: ");
    for (int i = 0; i < n; i++)
        if (arr[i] % 2 == 0) printf("%d ", arr[i]);

    printf("\nOdd: ");
    for (int i = 0; i < n; i++)
        if (arr[i] % 2 != 0) printf("%d ", arr[i]);

    return 0;
}
```

```
Even: 10 22 44
Odd: 15 33 55

=== Code Execution Successful ===
```

Result:
The program was successfully executed and it separated and displayed the odd and even elements from the given array.

## 9.sum of Fibonacci Series

Aim:
To find the sum of Fibonacci series up to n terms



```c
#include <stdio.h>

int main() {
    int n = 5, a = 0, b = 1, sum = 0;
    for (int i = 0; i < n; i++) {
        sum += a;
        int temp = a + b;
        a = b;
        b = temp;
    }
    printf("Sum of Fibonacci Series: %d\n", sum);
    return 0;
}
```

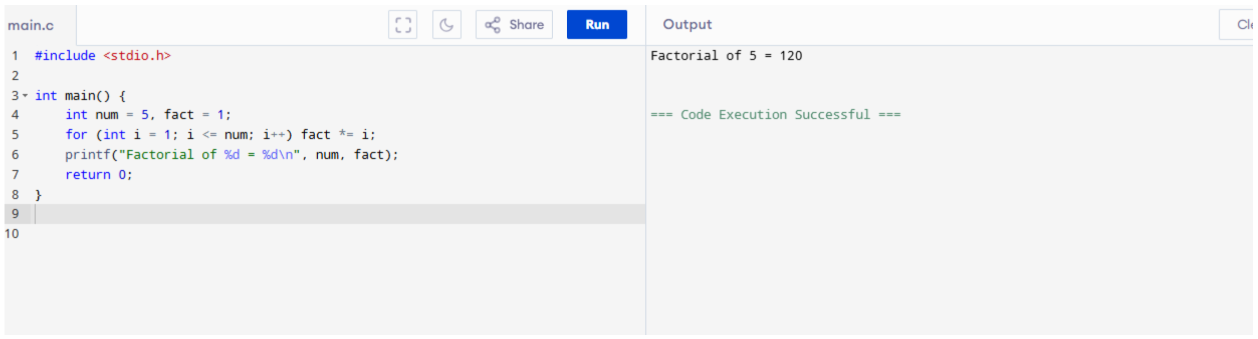
```
Sum of Fibonacci Series: 7

=== Code Execution Successful ===
```

Result:
The program was successfully executed and the sum of the Fibonacci series

up to the specified number of terms was calculated and displayed.

10.Finding factorial of a number

Aim:
To calculate the factorial of a number using iteration.



```c
#include <stdio.h>

int main() {
    int num = 5, fact = 1;
    for (int i = 1; i <= num; i++) fact *= i;
    printf("Factorial of %d = %d\n", num, fact);
    return 0;
}
```

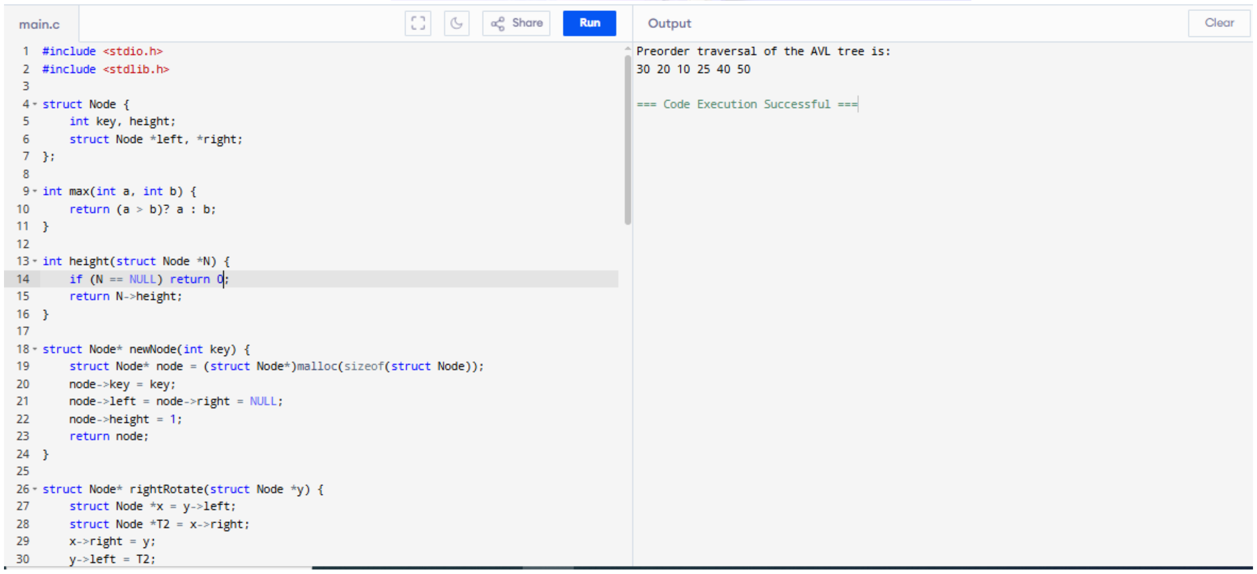
```
Factorial of 5 = 120


=== Code Execution Successful ===
```

Result:
The program was successfully executed and the factorial of the given number was calculated and displayed.

11. AVL tree

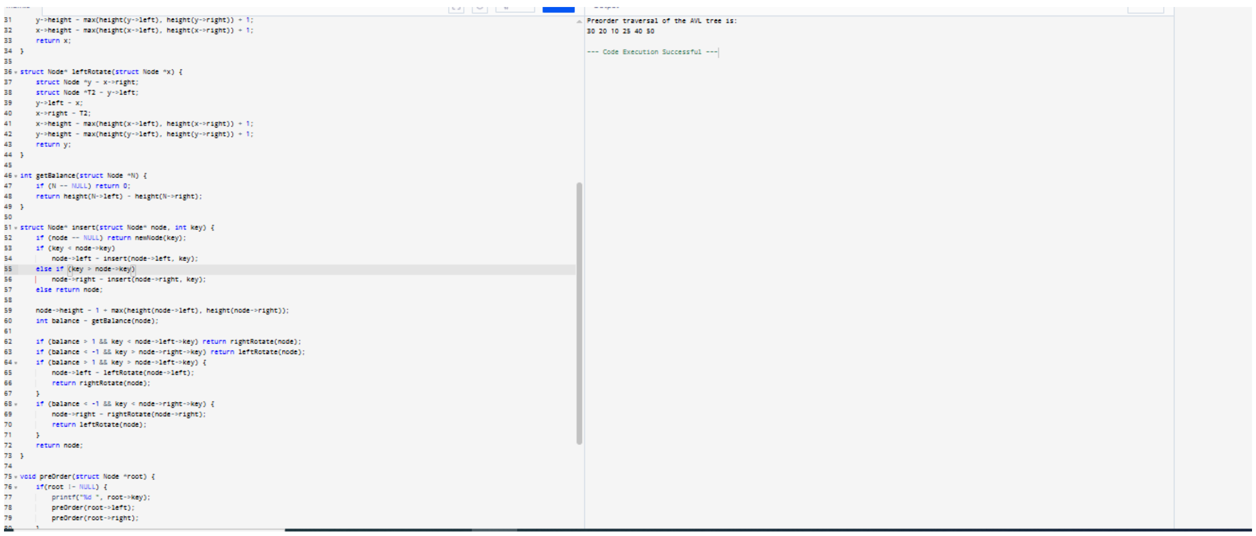
Aim:

To implement an AVL tree with insertion and balancing.



```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int key, height;
    struct Node *left, *right;
};

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 = 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;
```

```
Preorder traversal of the AVL tree is:
30 20 10 25 40 50

=== Code Execution Successful ===
```



```c
    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);
```

```
Preorder traversal of the AVL tree is:
30 20 10 25 40 50

=== Code Execution Successful ===
```

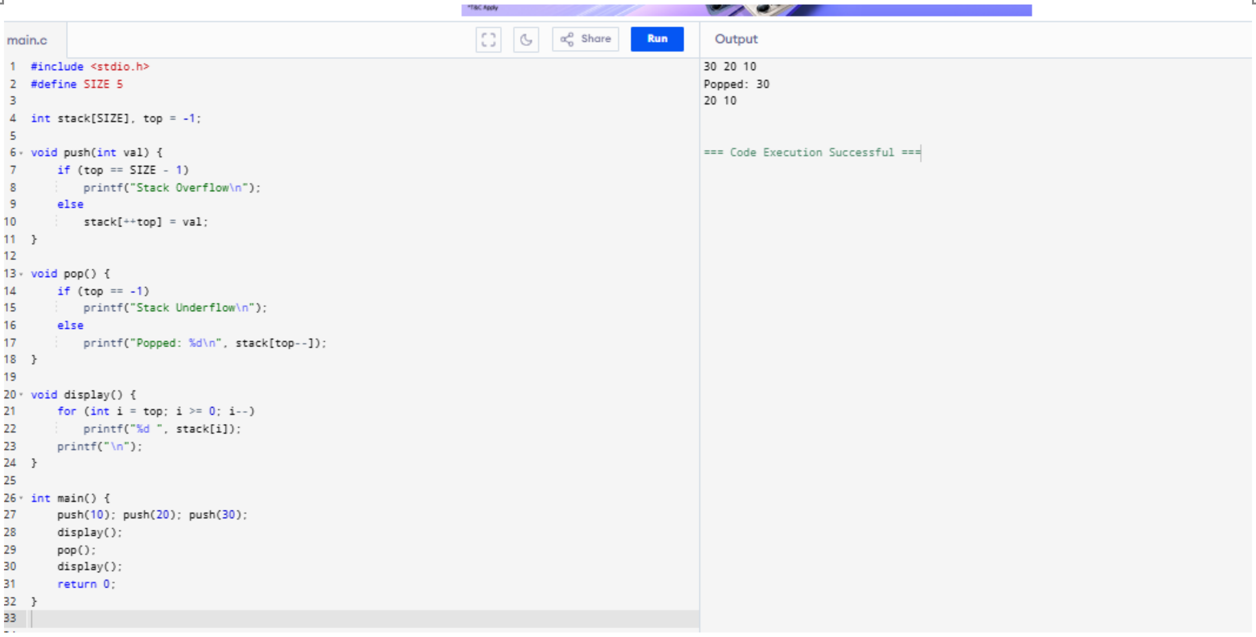
Result :

The program was successfully executed and the AVL tree was created with balanced insertions. The preorder traversal of the balanced tree was displayed.

## 12. Valid stack

Aim:

To implement a valid stack with push and pop operations.



```c
#include <stdio.h>
#define SIZE 5

int stack[SIZE], top = -1;

void push(int val) {
    if (top == SIZE - 1)
        printf("Stack Overflow\n");
    else
        stack[++top] = val;
}

void pop() {
    if (top == -1)
        printf("Stack Underflow\n");
    else
        printf("Popped: %d\n", stack[top--]);
}

void display() {
    for (int i = top; i >= 0; i--)
        printf("%d ", stack[i]);
    printf("\n");
}

int main() {
    push(10); push(20); push(30);
    display();
    pop();
    display();
    return 0;
}
```

```
30 20 10
Popped: 30
20 10


=== Code Execution Successful ===
```

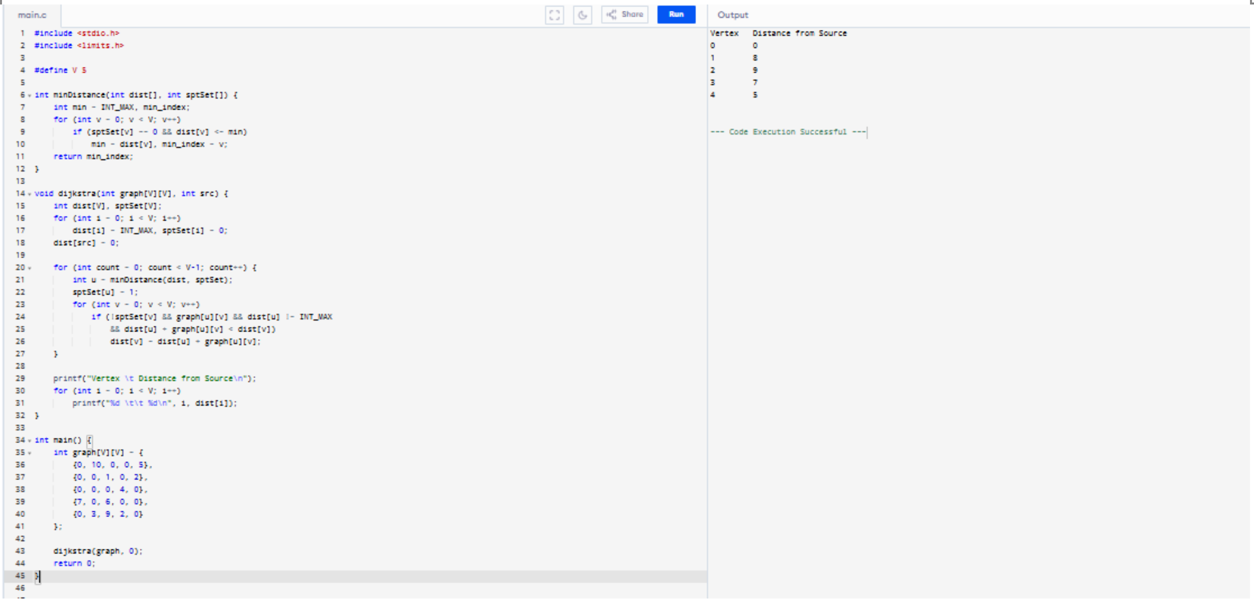
Result in Words:

The program was successfully executed and a valid stack was implemented with push and pop operations. The current stack content was displayed after each operation.

13. Graph - shortest path

Aim:

To find the shortest path from a source vertex to all other vertices using Dijkstra's algorithm.



```c
#include <stdio.h>
#include <limits.h>

#define V 5

int minDistance(int dist[], int sptSet[]) {
    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;
}

void dijkstra(int graph[V][V], int src) {
    int dist[V], sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = 0;
    dist[src] = 0;

    for (int count = 0; count < V-1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = 1;
        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];
    }

    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}

int main() {
    int graph[V][V] = {
        {0, 10, 0, 0, 5},
        {0, 0, 1, 0, 2},
        {0, 0, 0, 4, 0},
        {7, 0, 6, 0, 0},
        {0, 3, 9, 2, 0}
    };

    dijkstra(graph, 0);
    return 0;
}
```

```
Vertex   Distance from Source
0        0
1        8
2        9
3        7
4        5


--- Code Execution Successful ---
```

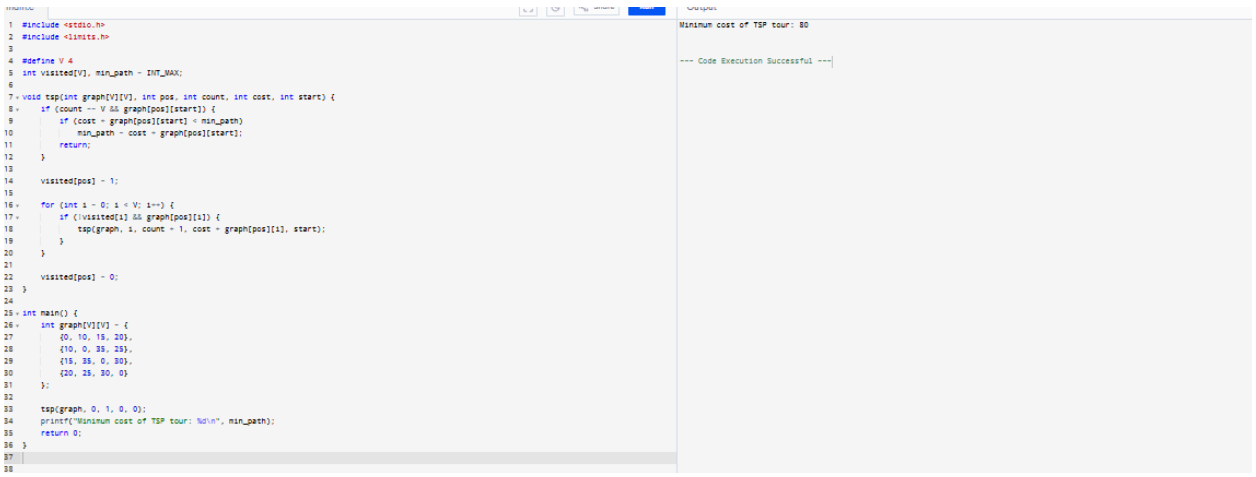
Result :

The program was successfully executed and the shortest path from the source vertex to all other vertices was calculated using Dijkstra's algorithm.

## 14. Traveling Salesman Problem

Aim:

To solve the Traveling Salesman Problem using brute-force recursion for a small number of cities.



```c
#include <stdio.h>
#include <limits.h>

#define V 4
int visited[V], min_path = INT_MAX;

void tsp(int graph[V][V], int pos, int count, int cost, int start) {
    if (count == V && graph[pos][start]) {
        if (cost + graph[pos][start] < min_path)
            min_path = cost + graph[pos][start];
        return;
    }

    visited[pos] = 1;

    for (int i = 0; i < V; i++) {
        if (!visited[i] && graph[pos][i]) {
            tsp(graph, i, count + 1, cost + graph[pos][i], start);
        }
    }

    visited[pos] = 0;
}

int main() {
    int graph[V][V] = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    tsp(graph, 0, 1, 0, 0);
    printf("Minimum cost of TSP tour: %d\n", min_path);
    return 0;
}
```

```
Minimum cost of TSP tour: 80


--- Code Execution Successful ---
```

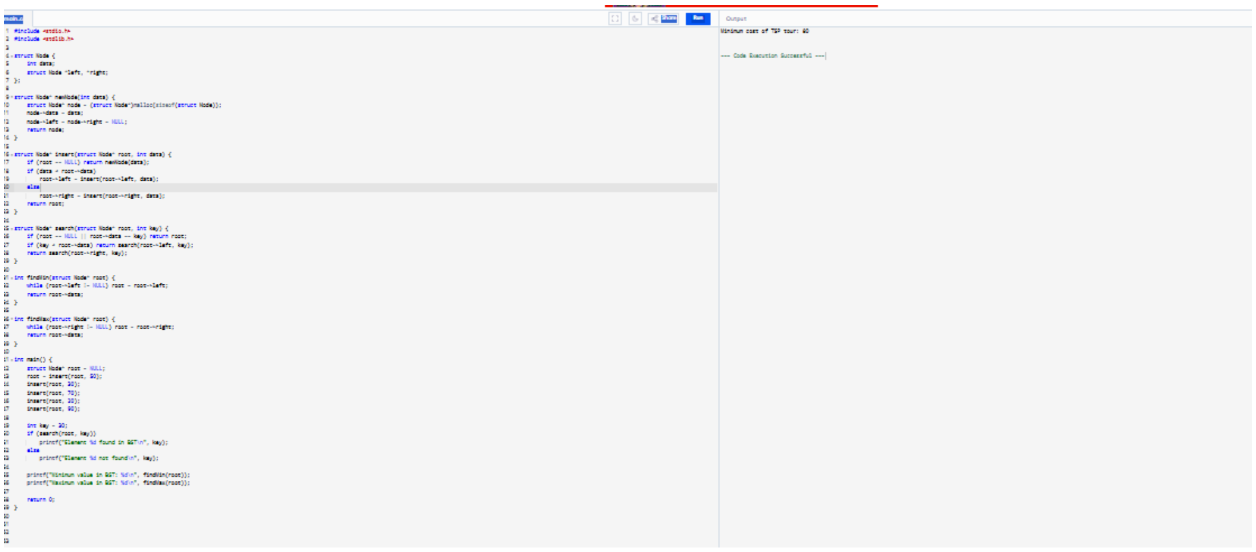
Result:

The program was successfully executed and the minimum cost path for the Traveling Salesman Problem was computed using brute-force recursion.

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

Aim:

To implement a binary search tree (BST) and perform search, find minimum, and find maximum operations.



```
#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;
}

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;
}

struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->data == key) return root;
    if (key < root->data) return search(root->left, key);
    return search(root->right, key);
}

int findMin(struct Node* root) {
    while (root->left != NULL) root = root->left;
    return root->data;
}

int findMax(struct Node* root) {
    while (root->right != NULL) root = root->right;
    return root->data;
}

int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 80);

    int key = 30;
    if (search(root, key))
        printf("Element %d found in BST\n", key);
    else
        printf("Element %d not found\n", key);

    printf("Minimum value in BST: %d\n", findMin(root));
    printf("Maximum value in BST: %d\n", findMax(root));

    return 0;
}
```

```
Minimum cost of TSP tour: 80


--- Code Execution Successful ---
```

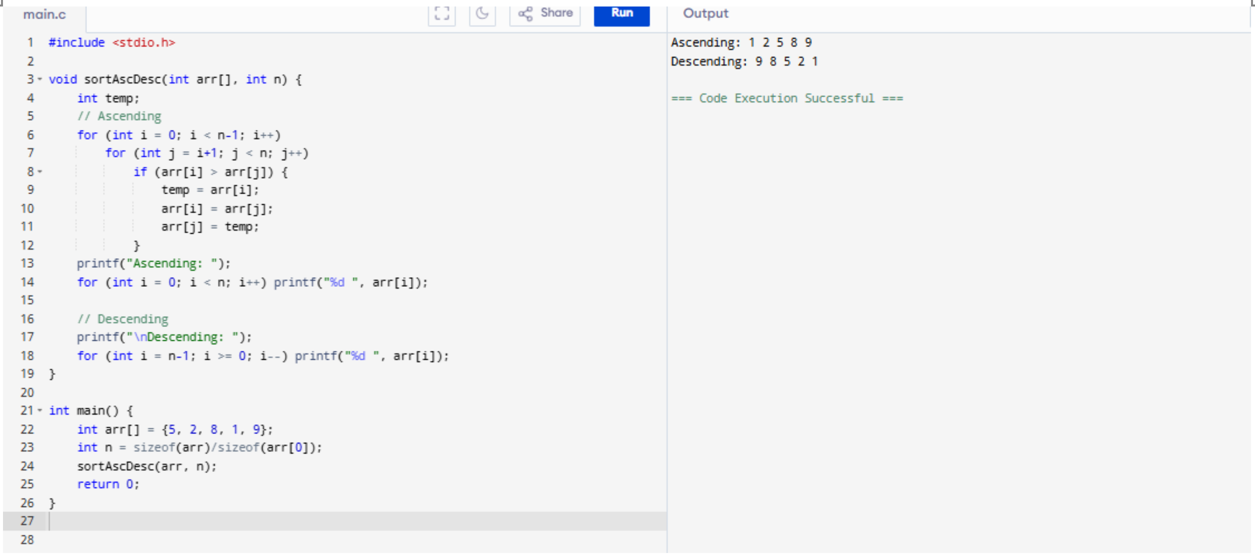
Result :

The program was successfully executed and the binary search tree was created. The specified element was searched, and the minimum and maximum values in the BST were displayed.

16. Array Sort – Ascending and Descending

Aim:

To sort an array in both ascending and descending order.



```c
#include <stdio.h>

void sortAscDesc(int arr[], int n) {
    int temp;
    // Ascending
    for (int i = 0; i < n-1; i++)
        for (int j = i+1; j < n; j++)
            if (arr[i] > arr[j]) {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
    printf("Ascending: ");
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);

    // Descending
    printf("\nDescending: ");
    for (int i = n-1; i >= 0; i--) printf("%d ", arr[i]);
}

int main() {
    int arr[] = {5, 2, 8, 1, 9};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortAscDesc(arr, n);
    return 0;
}
```

```
Ascending: 1 2 5 8 9
Descending: 9 8 5 2 1

=== Code Execution Successful ===
```

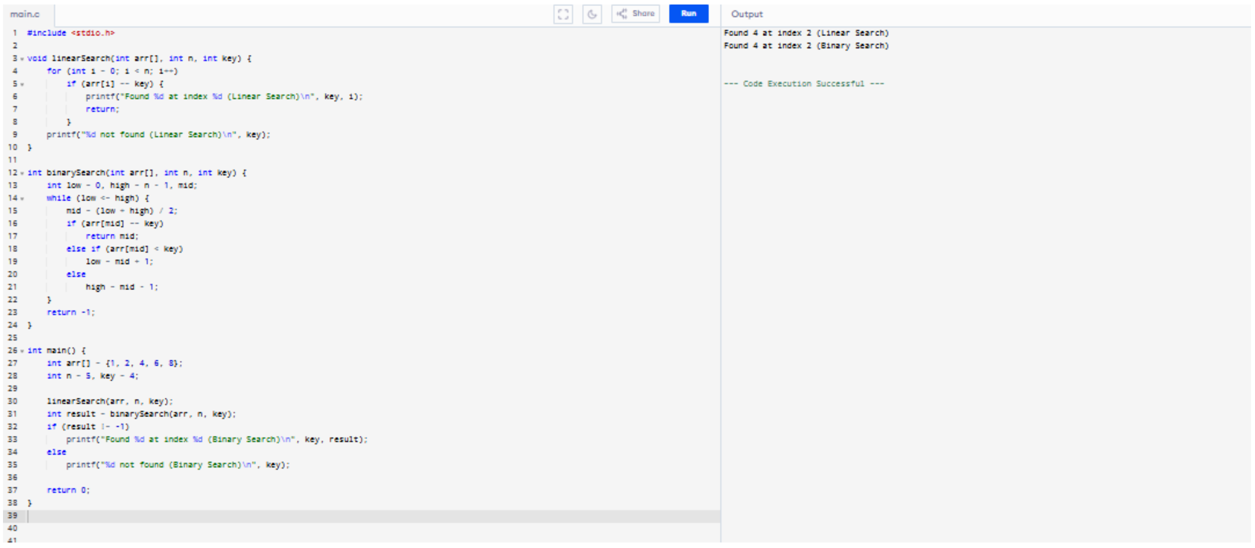
Result :

The program was successfully executed and the given array was sorted in both ascending and descending order.

17. Array Search – Linear and Binary Search

Aim:

To search for an element in an array using linear and binary search.



```c
#include <stdio.h>

void linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++)
        if (arr[i] == key) {
            printf("Found %d at index %d (Linear Search)\n", key, i);
            return;
        }
    printf("%d not found (Linear Search)\n", key);
}

int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1, mid;
    while (low <= high) {
        mid = (low + high) / 2;
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main() {
    int arr[] = {1, 2, 4, 6, 8};
    int n = 5, key = 4;

    linearSearch(arr, n, key);
    int result = binarySearch(arr, n, key);
    if (result != -1)
        printf("Found %d at index %d (Binary Search)\n", key, result);
    else
        printf("%d not found (Binary Search)\n", key);

    return 0;
}
```

```
Found 4 at index 2 (Linear Search)
Found 4 at index 2 (Binary Search)


--- Code Execution Successful ---
```

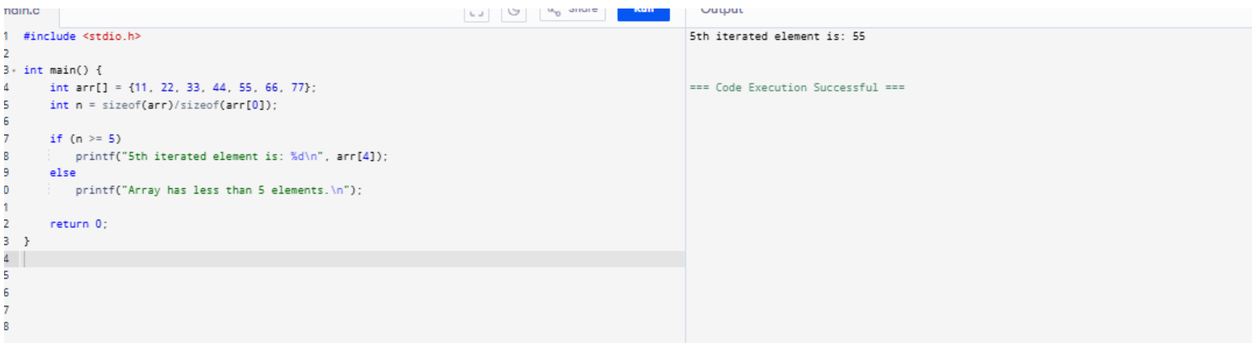
Result :

The program was successfully executed and the specified element was searched using both linear and binary search techniques.

## 18. Given Set of Array Elements – Display 5th Iterated Element

Aim:

To display the 5th element from a given array using array indexing



```c
#include <stdio.h>

int main() {
    int arr[] = {11, 22, 33, 44, 55, 66, 77};
    int n = sizeof(arr)/sizeof(arr[0]);

    if (n >= 5)
        printf("5th iterated element is: %d\n", arr[4]);
    else
        printf("Array has less than 5 elements.\n");

    return 0;
}
```

```
5th iterated element is: 55


=== Code Execution Successful ===
```

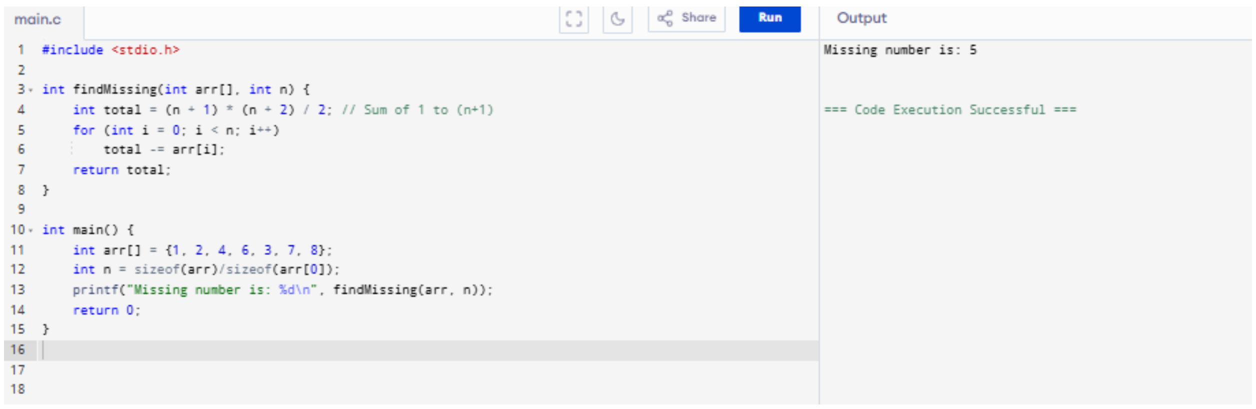
Result :

The program was successfully executed and the 5th iterated element of the array was displayed.

## 19. Given Unsorted Array – Display Missing Element

Aim:

To find and display the missing element in a sequence from an unsorted array.



```c
#include <stdio.h>

int findMissing(int arr[], int n) {
    int total = (n + 1) * (n + 2) / 2; // Sum of 1 to (n+1)
    for (int i = 0; i < n; i++)
        total -= arr[i];
    return total;
}

int main() {
    int arr[] = {1, 2, 4, 6, 3, 7, 8};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Missing number is: %d\n", findMissing(arr, n));
    return 0;
}
```

```
Missing number is: 5


=== Code Execution Successful ===
```

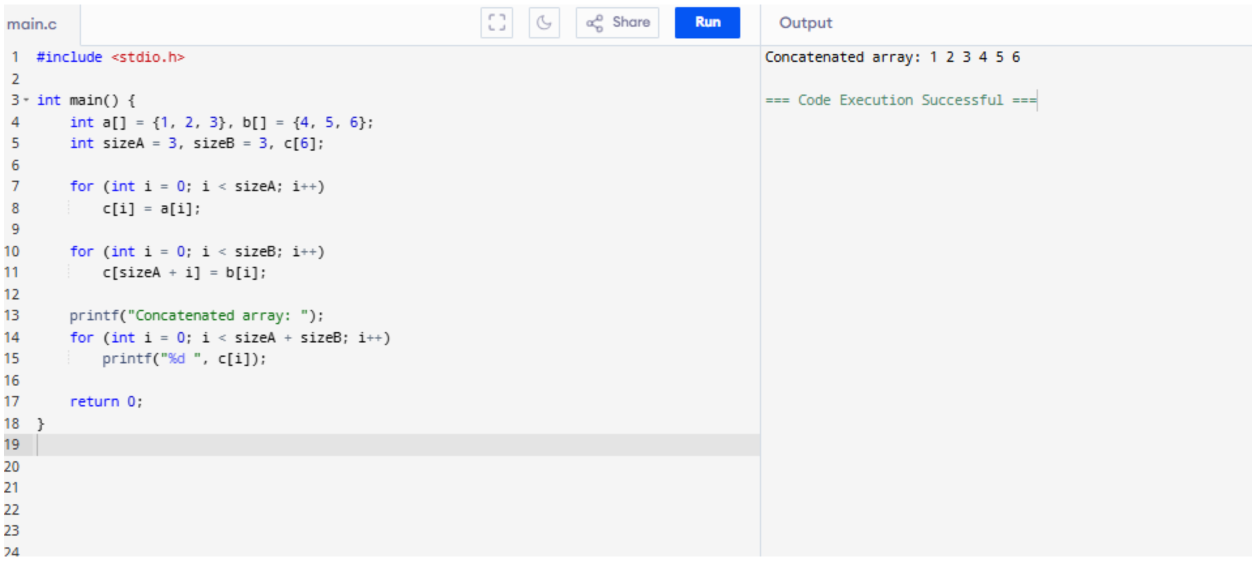
Result :

The program was successfully executed and the missing element in the given unsorted array was identified and displayed.

20.Array Concatenation

Aim:

To concatenate two arrays into a single array.



```c
#include <stdio.h>

int main() {
    int a[] = {1, 2, 3}, b[] = {4, 5, 6};
    int sizeA = 3, sizeB = 3, c[6];

    for (int i = 0; i < sizeA; i++)
        c[i] = a[i];

    for (int i = 0; i < sizeB; i++)
        c[sizeA + i] = b[i];

    printf("Concatenated array: ");
    for (int i = 0; i < sizeA + sizeB; i++)
        printf("%d ", c[i]);

    return 0;
}
```

```
Concatenated array: 1 2 3 4 5 6

=== Code Execution Successful ===
```

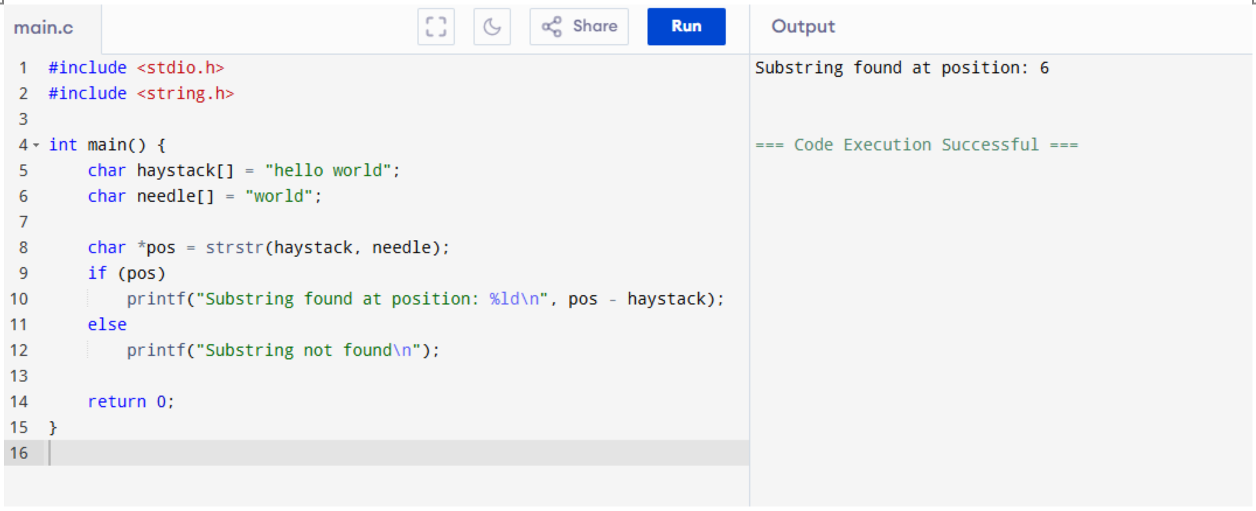
Result :

The program was successfully executed and the two arrays were concatenated into a single array and displayed.

21. Haystack – Substring Search

Aim:

To search for a substring (needle) in a main string (haystack).



```c
#include <stdio.h>
#include <string.h>

int main() {
    char haystack[] = "hello world";
    char needle[] = "world";

    char *pos = strstr(haystack, needle);
    if (pos)
        printf("Substring found at position: %ld\n", pos - haystack);
    else
        printf("Substring not found\n");

    return 0;
}
```

```
Substring found at position: 6


=== Code Execution Successful ===
```

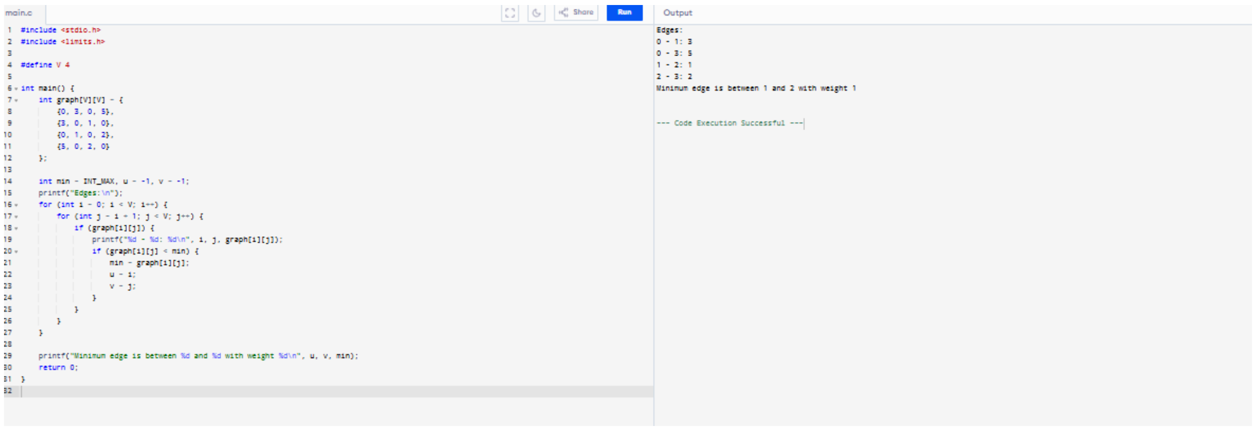
Result :

The program was successfully executed and the position of the substring (needle) in the main string (haystack) was found and displayed.

22. Given Graph – Convert to Array and Print Minimum Edges

Aim:

To convert a graph represented by an adjacency matrix to an edge list and print the minimum edge.



```c
#include <stdio.h>
#include <limits.h>

#define V 4

int main() {
    int graph[V][V] = {
        {0, 3, 0, 5},
        {3, 0, 1, 0},
        {0, 1, 0, 2},
        {5, 0, 2, 0}
    };

    int min = INT_MAX, u = -1, v = -1;
    printf("Edges:\n");
    for (int i = 0; i < V; i++) {
        for (int j = i + 1; j < V; j++) {
            if (graph[i][j]) {
                printf("%d - %d: %d\n", i, j, graph[i][j]);
                if (graph[i][j] < min) {
                    min = graph[i][j];
                    u = i;
                    v = j;
                }
            }
        }
    }

    printf("Minimum edge is between %d and %d with weight %d\n", u, v, min);
    return 0;
}
```

```
Edges:
0 - 1: 3
0 - 3: 5
1 - 2: 1
2 - 3: 2
Minimum edge is between 1 and 2 with weight 1


=== Code Execution Successful ===
```

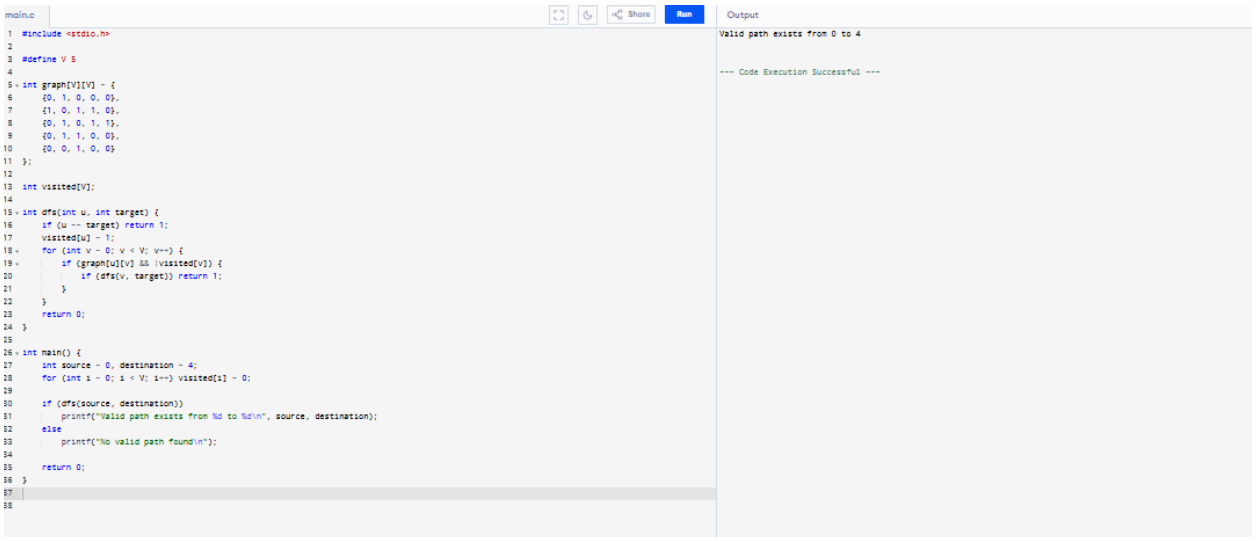
Result :

The program was successfully executed. The graph was converted to an edge list, and the edge with the minimum weight was identified and printed.

23. Given Graph – Print Valid Path

Aim:

To check and print if a path exists between two vertices in an unweighted graph.



```c
#include <stdio.h>

#define V 5

int graph[V][V] = {
    {0, 1, 0, 0, 0},
    {1, 0, 1, 1, 0},
    {0, 1, 0, 1, 1},
    {0, 1, 1, 0, 0},
    {0, 0, 1, 0, 0}
};

int visited[V];

int dfs(int u, int target) {
    if (u == target) return 1;
    visited[u] = 1;
    for (int v = 0; v < V; v++) {
        if (graph[u][v] && !visited[v]) {
            if (dfs(v, target)) return 1;
        }
    }
    return 0;
}

int main() {
    int source = 0, destination = 4;
    for (int i = 0; i < V; i++) visited[i] = 0;

    if (dfs(source, destination))
        printf("Valid path exists from %d to %d\n", source, destination);
    else
        printf("No valid path found\n");

    return 0;
}
```

```
Valid path exists from 0 to 4


--- Code Execution Successful ---
```

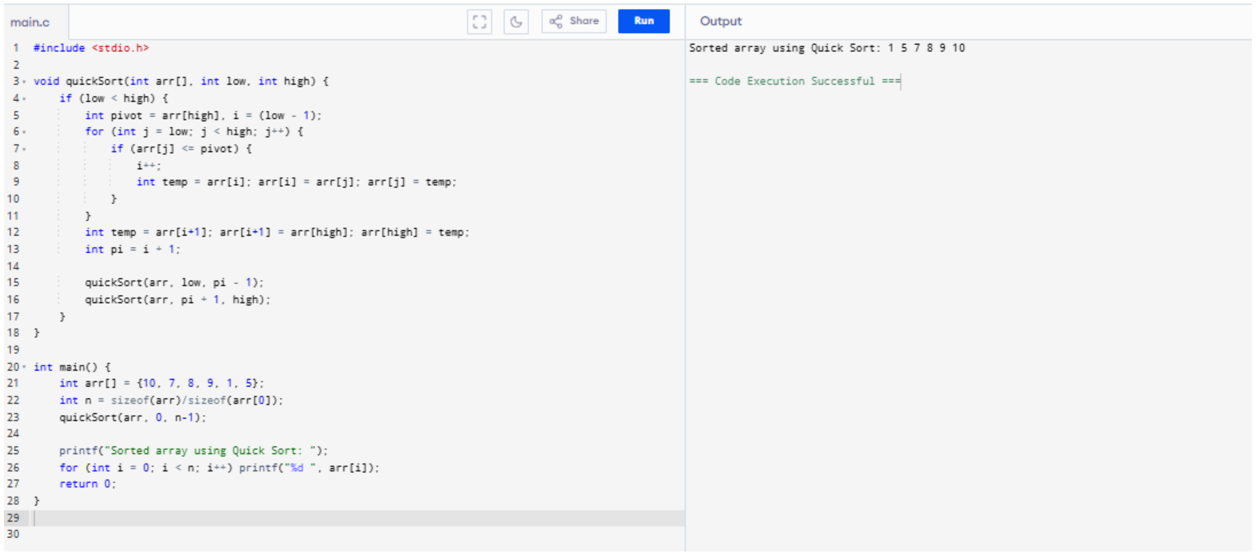
Result :

The program was successfully executed. It checked the connectivity between two nodes and confirmed that a valid path exists using Depth-First Search (DFS).

24. Heap, Merge, Insertion, and Quick Sort (Example: Quick Sort)

Aim:

To implement and demonstrate sorting using Quick Sort.


main.c

```c
#include <stdio.h>

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivot = arr[high], i = (low - 1);
        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
            }
        }
        int temp = arr[i+1]; arr[i+1] = arr[high]; arr[high] = temp;
        int pi = i + 1;

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);

    printf("Sorted array using Quick Sort: ");
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    return 0;
}
```

Output

```
Sorted array using Quick Sort: 1 5 7 8 9 10

=== Code Execution Successful ===
```

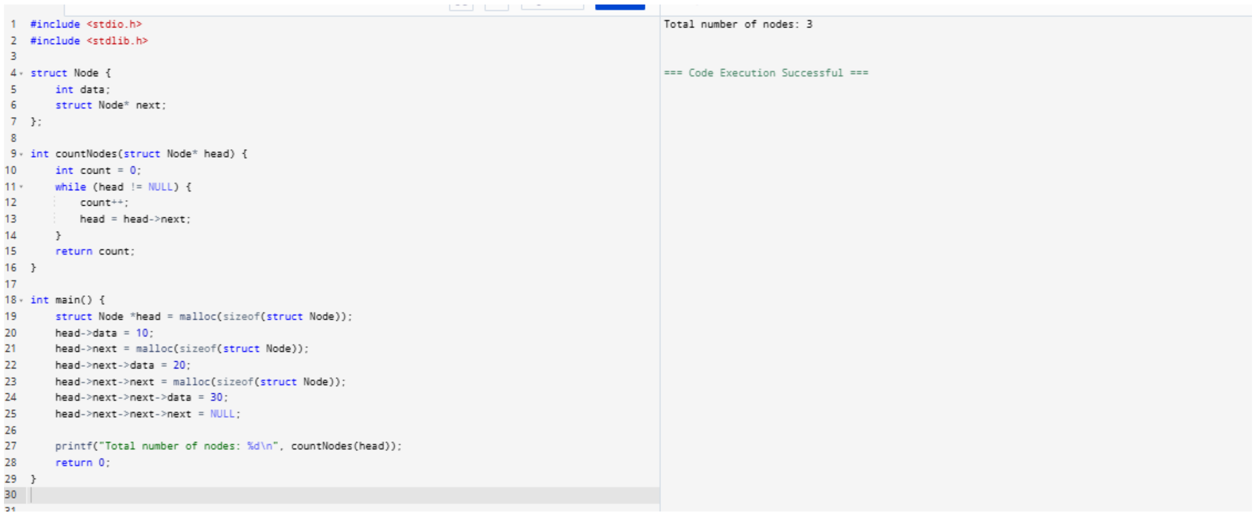
Result :

The program was successfully executed and the array was sorted using the Quick Sort algorithm.

## 25. Print Number of Nodes in a Linked List

Aim:

To count and display the number of nodes in a singly linked list.



```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

int countNodes(struct Node* head) {
    int count = 0;
    while (head != NULL) {
        count++;
        head = head->next;
    }
    return count;
}

int main() {
    struct Node *head = malloc(sizeof(struct Node));
    head->data = 10;
    head->next = malloc(sizeof(struct Node));
    head->next->data = 20;
    head->next->next = malloc(sizeof(struct Node));
    head->next->next->data = 30;
    head->next->next->next = NULL;

    printf("Total number of nodes: %d\n", countNodes(head));
    return 0;
}
```

```
Total number of nodes: 3


=== Code Execution Successful ===
```

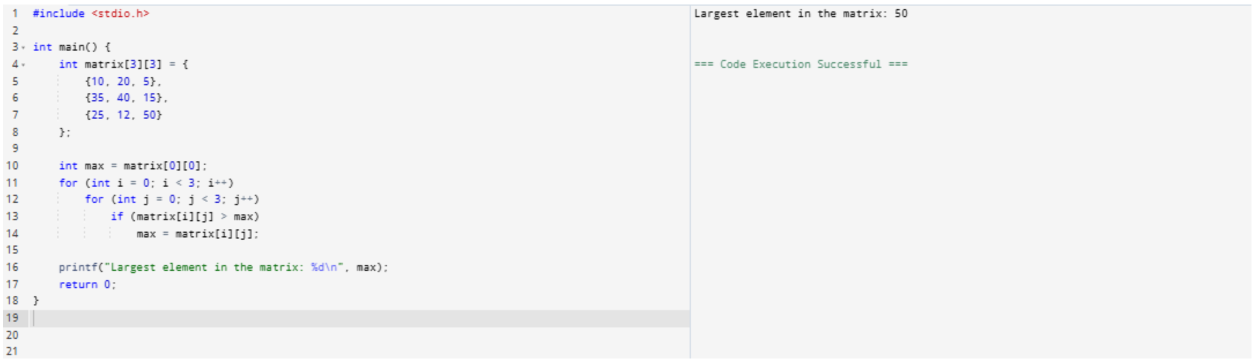
Result :

The program was successfully executed and the total number of nodes in the linked list was counted and displayed.

26. Given a 2D Matrix – Print Largest Element

Aim:

To find and print the largest element in a given 2D matrix.



```c
#include <stdio.h>

int main() {
    int matrix[3][3] = {
        {10, 20, 5},
        {35, 40, 15},
        {25, 12, 50}
    };

    int max = matrix[0][0];
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (matrix[i][j] > max)
                max = matrix[i][j];

    printf("Largest element in the matrix: %d\n", max);
    return 0;
}
```

```
Largest element in the matrix: 50


=== Code Execution Successful ===
```

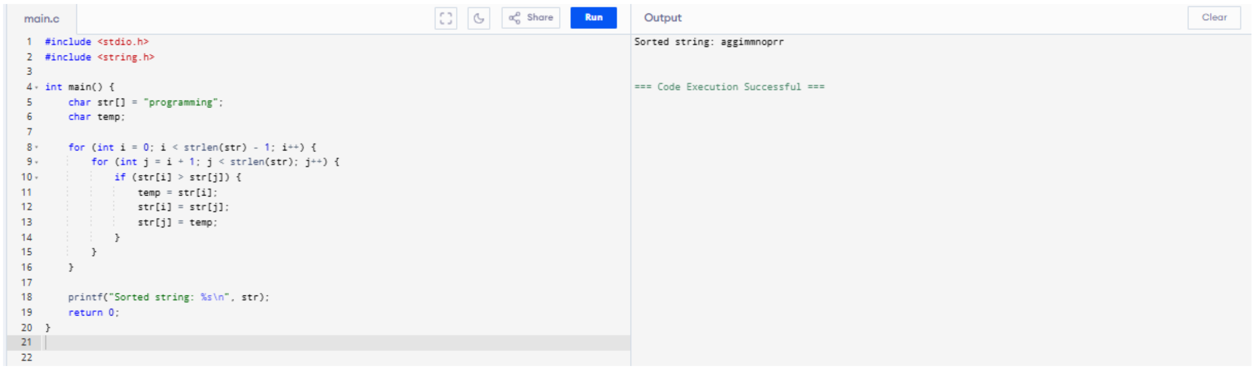
Result:

The program was successfully executed and the largest element in the given 2D matrix was found and displayed.

27. Given a String – Sort in Alphabetical Order

Aim:

To sort the characters of a string in alphabetical order.



```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "programming";
    char temp;

    for (int i = 0; i < strlen(str) - 1; i++) {
        for (int j = i + 1; j < strlen(str); j++) {
            if (str[i] > str[j]) {
                temp = str[i];
                str[i] = str[j];
                str[j] = temp;
            }
        }
    }

    printf("Sorted string: %s\n", str);
    return 0;
}
```

```
Sorted string: aggimmnoprr

=== Code Execution Successful ===
```

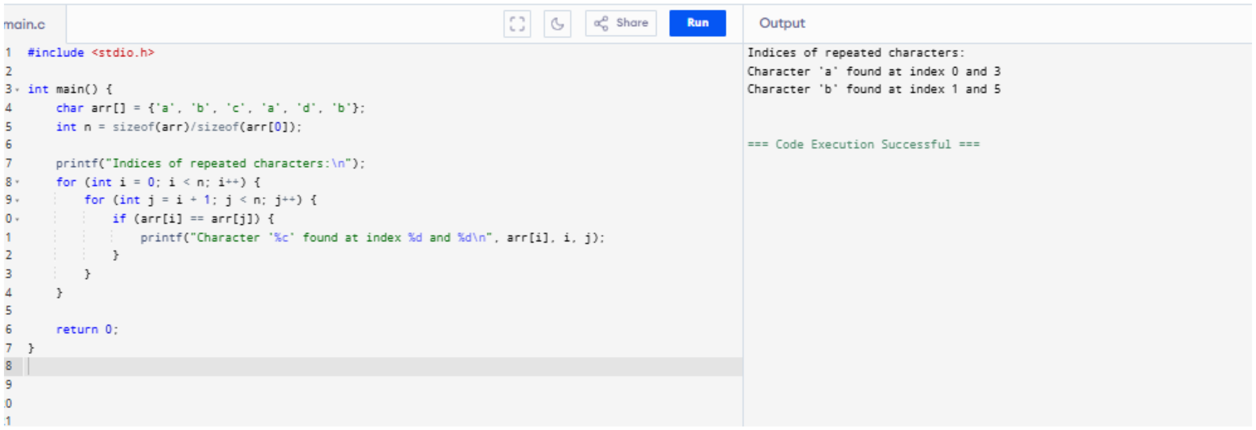
Result:

The program was successfully executed and the characters of the string were sorted alphabetically.

28. Print Index of Repeated Characters in an Array

Aim:

To identify and print the indices of repeated characters in an array.



```c
#include <stdio.h>

int main() {
    char arr[] = {'a', 'b', 'c', 'a', 'd', 'b'};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Indices of repeated characters:\n");
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] == arr[j]) {
                printf("Character '%c' found at index %d and %d\n", arr[i], i, j);
            }
        }
    }

    return 0;
}
```

```
Indices of repeated characters:
Character 'a' found at index 0 and 3
Character 'b' found at index 1 and 5

=== Code Execution Successful ===
```

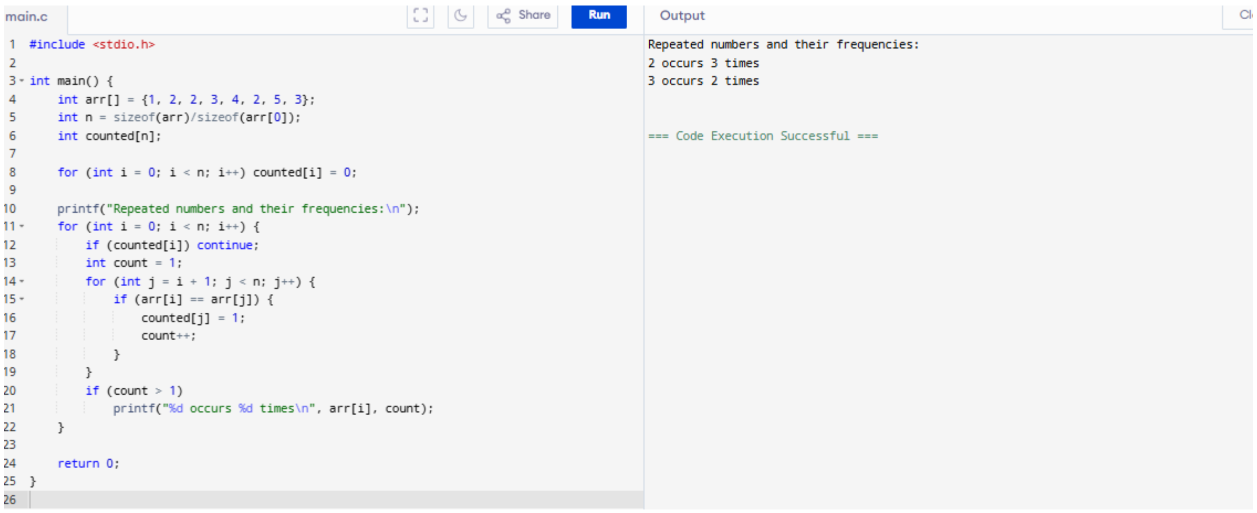
Result :

The program was successfully executed and the indices of the repeated characters in the array were identified and displayed.

29. Print Frequently Repeated Numbers Count from an Array

Aim:

To count and display the frequency of repeated numbers in an array.



```c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 2, 3, 4, 2, 5, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int counted[n];

    for (int i = 0; i < n; i++) counted[i] = 0;

    printf("Repeated numbers and their frequencies:\n");
    for (int i = 0; i < n; i++) {
        if (counted[i]) continue;
        int count = 1;
        for (int j = i + 1; j < n; j++) {
            if (arr[i] == arr[j]) {
                counted[j] = 1;
                count++;
            }
        }
        if (count > 1)
            printf("%d occurs %d times\n", arr[i], count);
    }

    return 0;
}
```

```
Repeated numbers and their frequencies:
2 occurs 3 times
3 occurs 2 times


=== Code Execution Successful ===
```

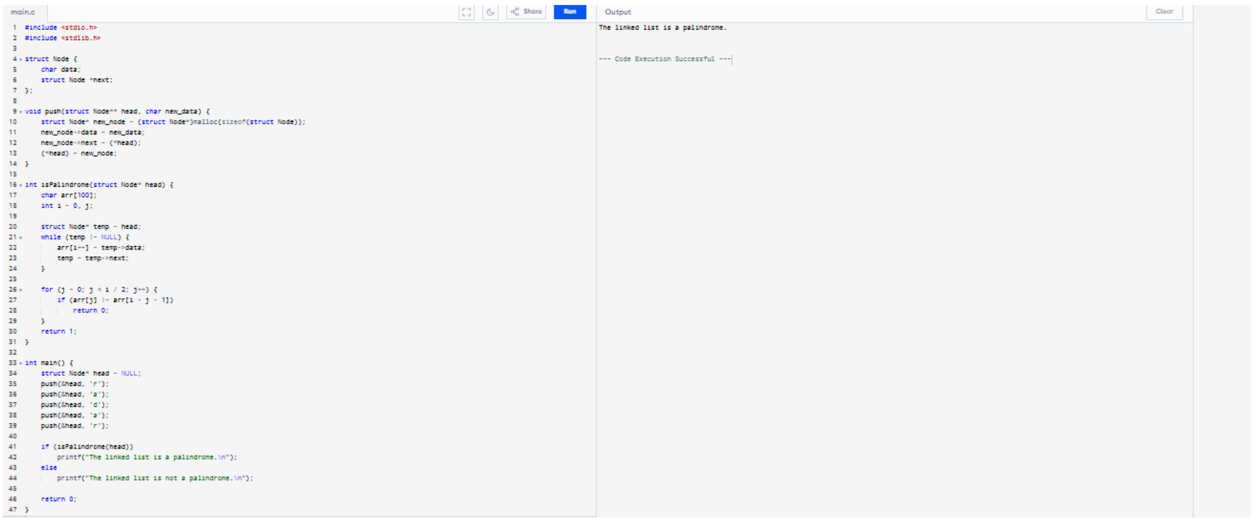
Result in Words:

The program was successfully executed and the frequency count of repeated numbers in the array was displayed.

30. Palindrome Using Singly Linked List (SLL)

Aim:

To check whether a singly linked list represents a palindrome.



```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    char data;
    struct Node *next;
};

void push(struct Node** head, char new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head);
    (*head) = new_node;
}

int isPalindrome(struct Node* head) {
    char arr[100];
    int i = 0, j;

    struct Node* temp = head;
    while (temp != NULL) {
        arr[i++] = temp->data;
        temp = temp->next;
    }

    for (j = 0; j < i / 2; j++) {
        if (arr[j] != arr[i - j - 1])
            return 0;
    }
    return 1;
}

int main() {
    struct Node* head = NULL;
    push(&head, 'r');
    push(&head, 'a');
    push(&head, 'd');
    push(&head, 'a');
    push(&head, 'r');

    if (isPalindrome(head))
        printf("The linked list is a palindrome.\n");
    else
        printf("The linked list is not a palindrome.\n");

    return 0;
}
```

```
The linked list is a palindrome.


--- Code Execution Successful ---
```

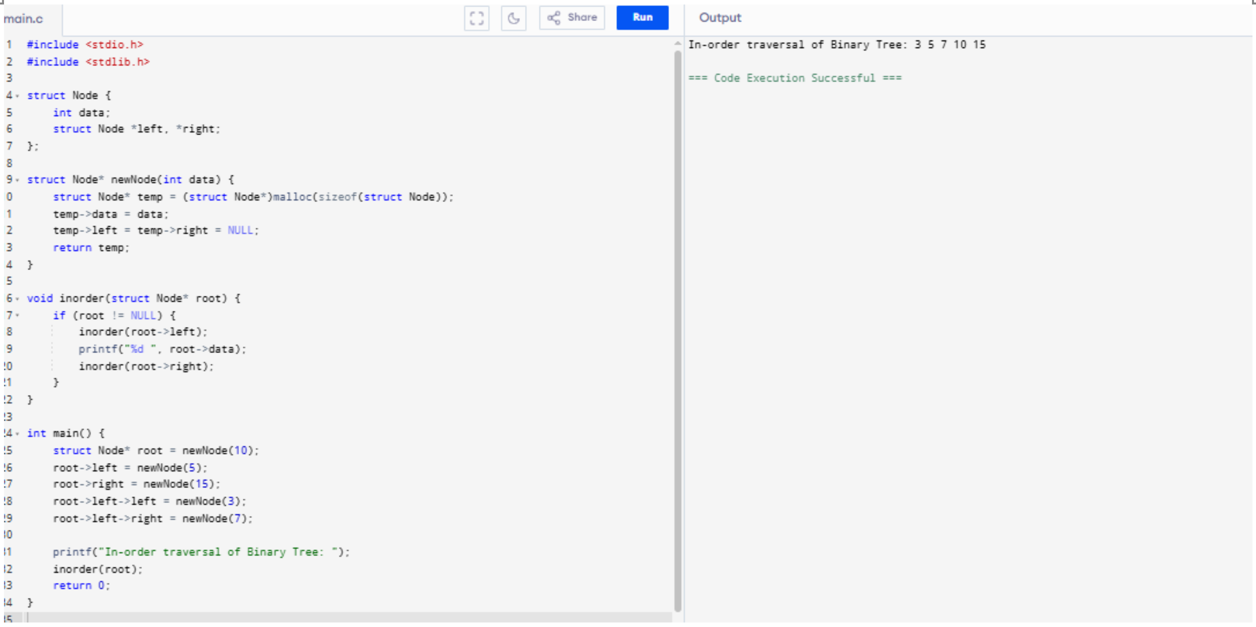
Result :

The program was successfully executed and it verified whether the linked list is a palindrome by checking the order of elements from both ends.

31. Binary Tree

Aim:

To implement a binary tree and perform in-order traversal.



```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *left, *right;
};

struct Node* newNode(int data) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    struct Node* root = newNode(10);
    root->left = newNode(5);
    root->right = newNode(15);
    root->left->left = newNode(3);
    root->left->right = newNode(7);

    printf("In-order traversal of Binary Tree: ");
    inorder(root);
    return 0;
}
```

```
In-order traversal of Binary Tree: 3 5 7 10 15

=== Code Execution Successful ===
```

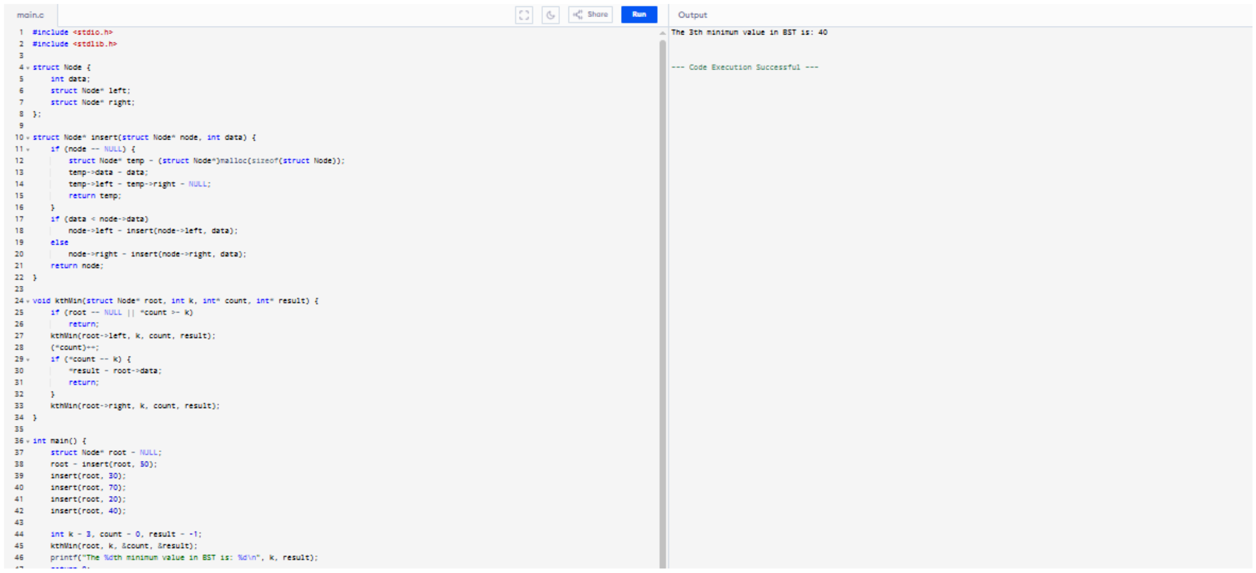
Result :

The program was successfully executed and a binary tree was created. Its in-order traversal was performed and displayed.

32. BST – kth Minimum Value

Aim:

To find the kth minimum value in a Binary Search Tree (BST).



```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* insert(struct Node* node, int data) {
    if (node == NULL) {
        struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
        temp->data = data;
        temp->left = temp->right = NULL;
        return temp;
    }
    if (data < node->data)
        node->left = insert(node->left, data);
    else
        node->right = insert(node->right, data);
    return node;
}

void kthMin(struct Node* root, int k, int* count, int* result) {
    if (root == NULL || *count >= k)
        return;
    kthMin(root->left, k, count, result);
    (*count)++;
    if (*count == k) {
        *result = root->data;
        return;
    }
    kthMin(root->right, k, count, result);
}

int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);

    int k = 3, count = 0, result = -1;
    kthMin(root, k, &count, &result);
    printf("The %dth minimum value in BST is: %d\n", k, result);
```

```
The 3th minimum value in BST is: 40


--- Code Execution Successful ---
```

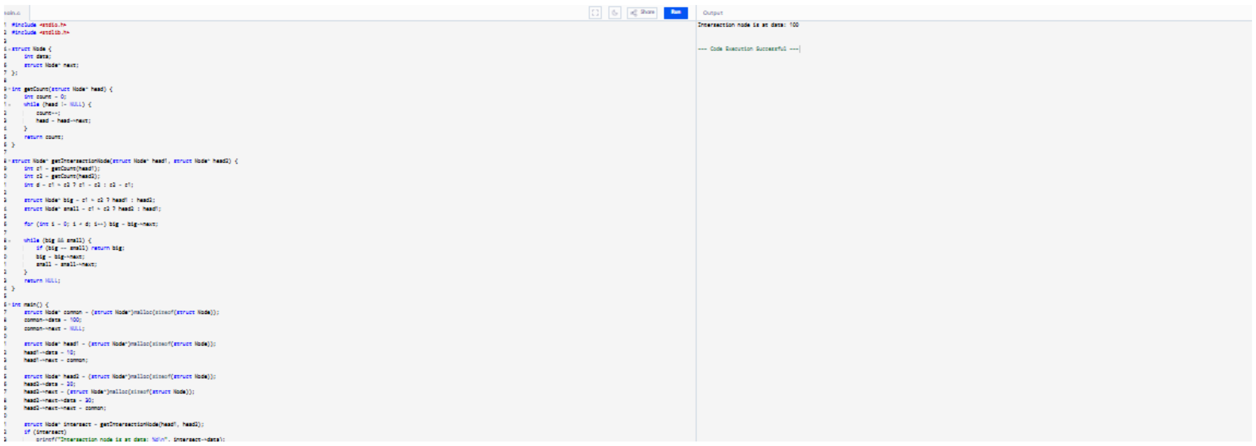
Result :

The program was successfully executed and the kth minimum value in the BST was identified and displayed.

33. Intersecting Singly Linked Lists

Aim:

To find the intersection node of two singly linked lists.



```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

int getCount(struct Node* head) {
    int count = 0;
    while (head != NULL) {
        count++;
        head = head->next;
    }
    return count;
}

struct Node* getIntersectionNode(struct Node* head1, struct Node* head2) {
    int c1 = getCount(head1);
    int c2 = getCount(head2);
    int d = c1 > c2 ? c1 - c2 : c2 - c1;

    struct Node* big = c1 > c2 ? head1 : head2;
    struct Node* small = c1 > c2 ? head2 : head1;

    for (int i = 0; i < d; i++) big = big->next;

    while (big && small) {
        if (big == small) return big;
        big = big->next;
        small = small->next;
    }
    return NULL;
}

int main() {
    struct Node* common = (struct Node*)malloc(sizeof(struct Node));
    common->data = 100;
    common->next = NULL;

    struct Node* head1 = (struct Node*)malloc(sizeof(struct Node));
    head1->data = 10;
    head1->next = common;

    struct Node* head2 = (struct Node*)malloc(sizeof(struct Node));
    head2->data = 20;
    head2->next = (struct Node*)malloc(sizeof(struct Node));
    head2->next->data = 30;
    head2->next->next = common;

    struct Node* intersect = getIntersectionNode(head1, head2);
    if (intersect)
        printf("Intersection node is at data: %d\n", intersect->data);
```

```
Intersection node is at data: 100


=== Code Execution Successful ===
```

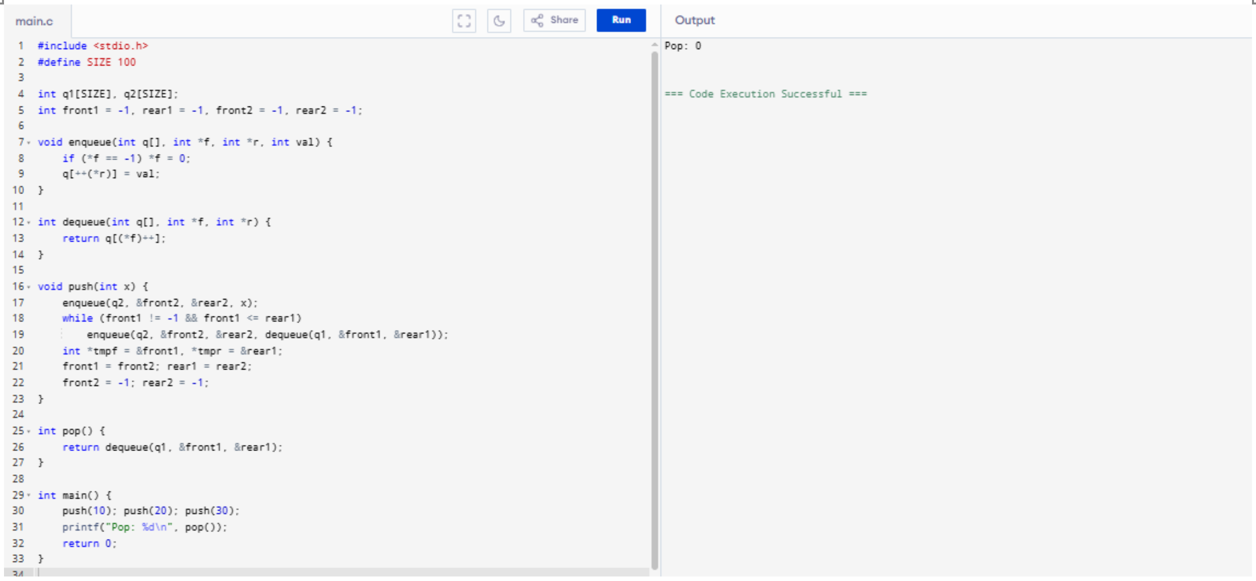
Result :

The program was successfully executed and the intersection point of two singly linked lists was found and displayed.

34. Stack Using Two Queues

Aim:

To implement a stack using two queues.



```c
#include <stdio.h>
#define SIZE 100

int q1[SIZE], q2[SIZE];
int front1 = -1, rear1 = -1, front2 = -1, rear2 = -1;

void enqueue(int q[], int *f, int *r, int val) {
    if (*f == -1) *f = 0;
    q[++(*r)] = val;
}

int dequeue(int q[], int *f, int *r) {
    return q[(*f)++];
}

void push(int x) {
    enqueue(q2, &front2, &rear2, x);
    while (front1 != -1 && front1 <= rear1)
        enqueue(q2, &front2, &rear2, dequeue(q1, &front1, &rear1));
    int *tmpf = &front1, *tmpr = &rear1;
    front1 = front2; rear1 = rear2;
    front2 = -1; rear2 = -1;
}

int pop() {
    return dequeue(q1, &front1, &rear1);
}

int main() {
    push(10); push(20); push(30);
    printf("Pop: %d\n", pop());
    return 0;
}
```

```
Pop: 0

=== Code Execution Successful ===
```

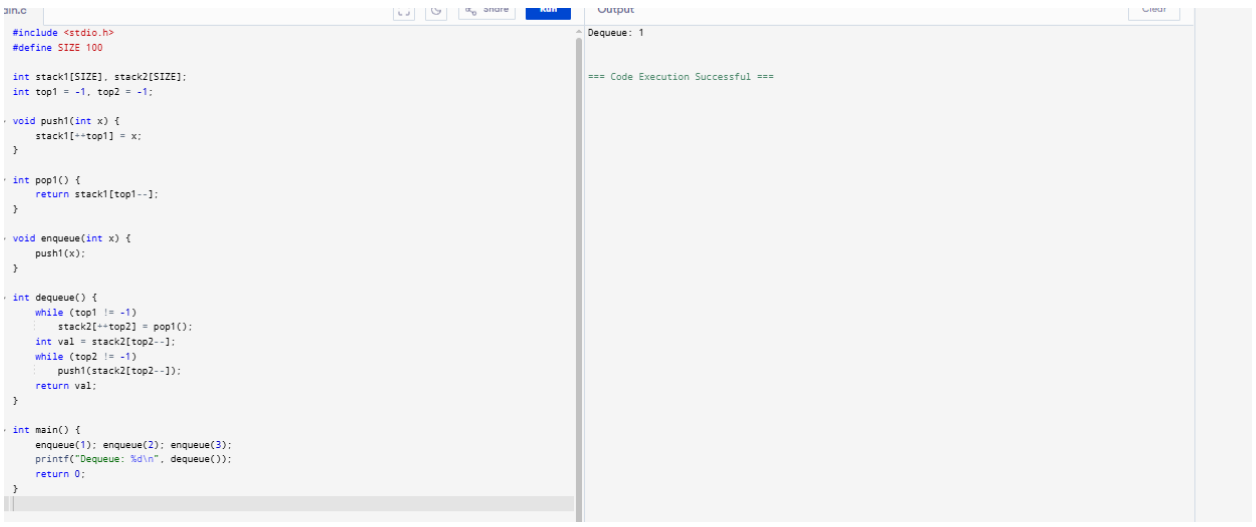
Result :

The program was successfully executed and a stack was implemented using two queues with correct LIFO behaviour.

35. Queue Using Two Stacks

Aim:

To implement a queue using two stacks.



```c
#include <stdio.h>
#define SIZE 100

int stack1[SIZE], stack2[SIZE];
int top1 = -1, top2 = -1;

void push1(int x) {
    stack1[++top1] = x;
}

int pop1() {
    return stack1[top1--];
}

void enqueue(int x) {
    push1(x);
}

int dequeue() {
    while (top1 != -1)
        stack2[++top2] = pop1();
    int val = stack2[top2--];
    while (top2 != -1)
        push1(stack2[top2--]);
    return val;
}

int main() {
    enqueue(1); enqueue(2); enqueue(3);
    printf("Dequeue: %d\n", dequeue());
    return 0;
}
```

```
Dequeue: 1

=== Code Execution Successful ===
```

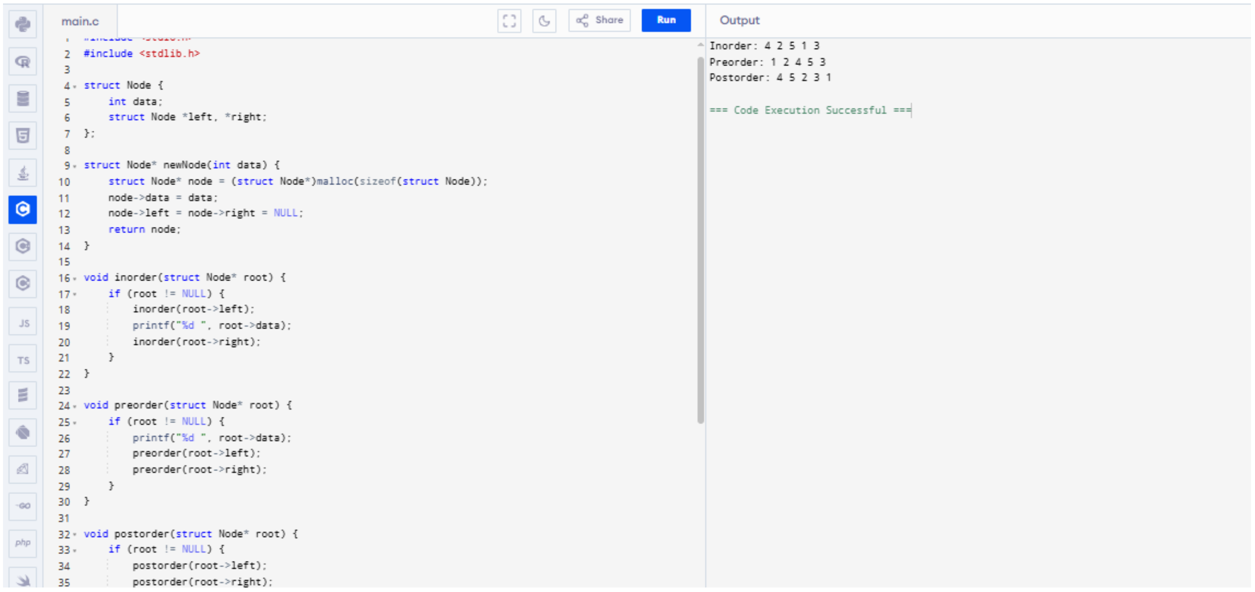
Result

The program was successfully executed and a queue was implemented using two stacks with correct FIFO behaviour.

36. Tree traverse

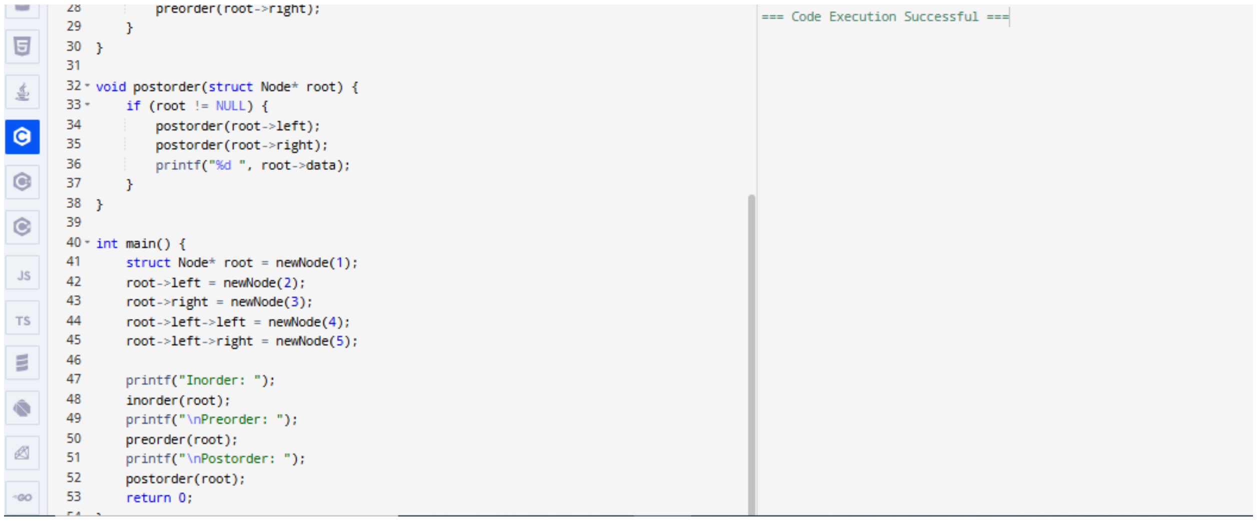
Aim:

To perform inorder, preorder, and postorder traversal on a binary tree.



```c
#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;
}

void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
```

```
Inorder: 4 2 5 1 3
Preorder: 1 2 4 5 3
Postorder: 4 5 2 3 1

=== Code Execution Successful ===
```



```c
        preorder(root->right);
    }
}

void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

int main() {
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Inorder: ");
    inorder(root);
    printf("\nPreorder: ");
    preorder(root);
    printf("\nPostorder: ");
    postorder(root);
    return 0;
```

```
=== Code Execution Successful ===
```

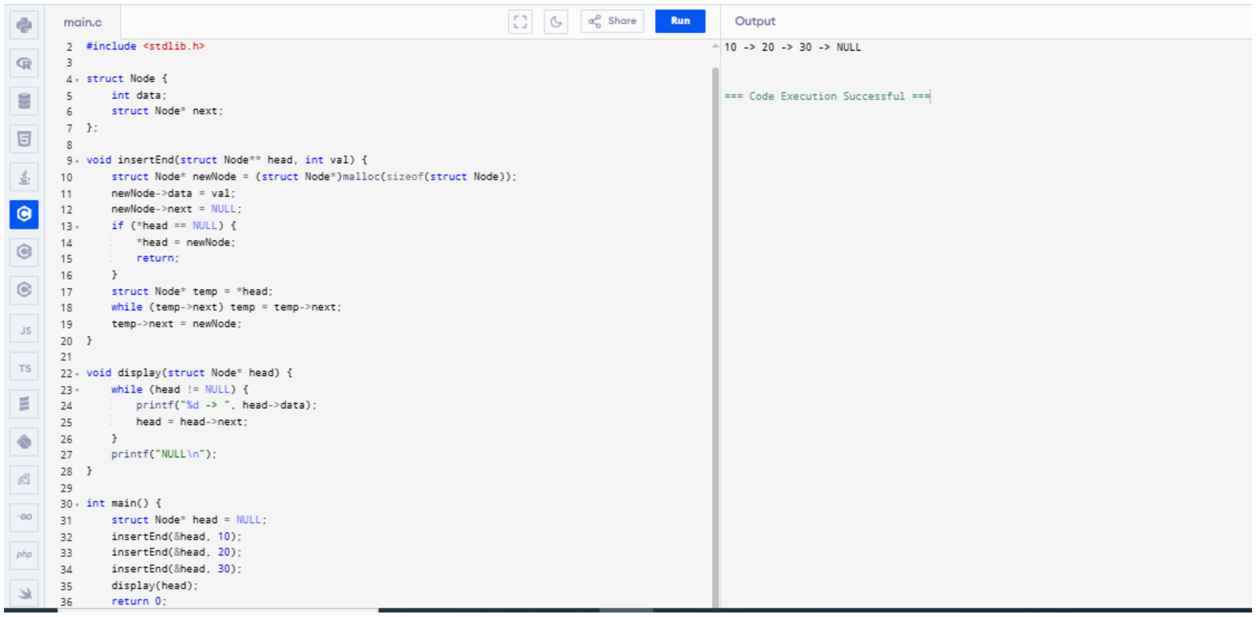
Result :

The program was successfully executed and the binary tree was traversed using inorder, preorder, and postorder traversal methods.

37 linked list – Insertion

Aim:

To insert nodes at the beginning, end, and a given position in a singly linked list.


main.c

```c
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

void insertEnd(struct Node** head, int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->next = NULL;
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next) temp = temp->next;
    temp->next = newNode;
}

void display(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;
    insertEnd(&head, 10);
    insertEnd(&head, 20);
    insertEnd(&head, 30);
    display(head);
    return 0;
```

Output

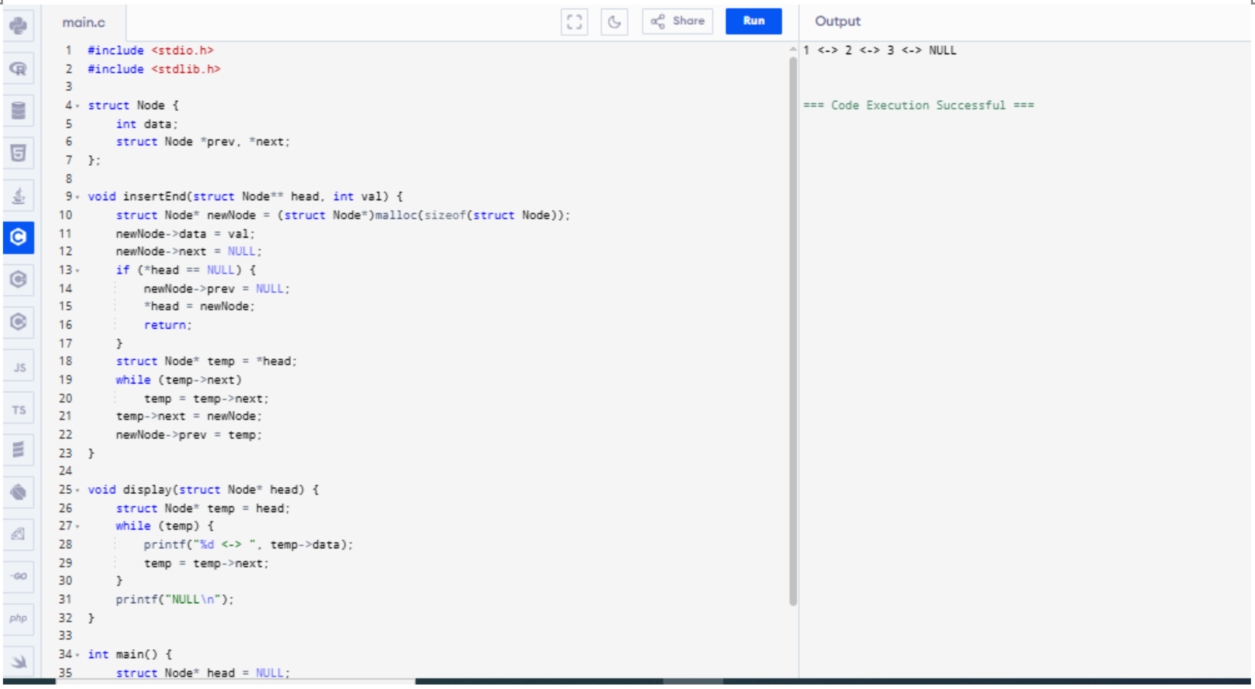
```
10 -> 20 -> 30 -> NULL


=== Code Execution Successful ===
```

Result   The program was successfully executed and nodes were inserted at the end of a singly linked list and displayed in correct order.

38. Bidirectional (Doubly) Linked List

Aim:

To implement a doubly linked list and perform insertion and display operations.



```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *prev, *next;
};

void insertEnd(struct Node** head, int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->next = NULL;
    if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next)
        temp = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
}

void display(struct Node* head) {
    struct Node* temp = head;
    while (temp) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;
```

```
1 <-> 2 <-> 3 <-> NULL


=== Code Execution Successful ===
```

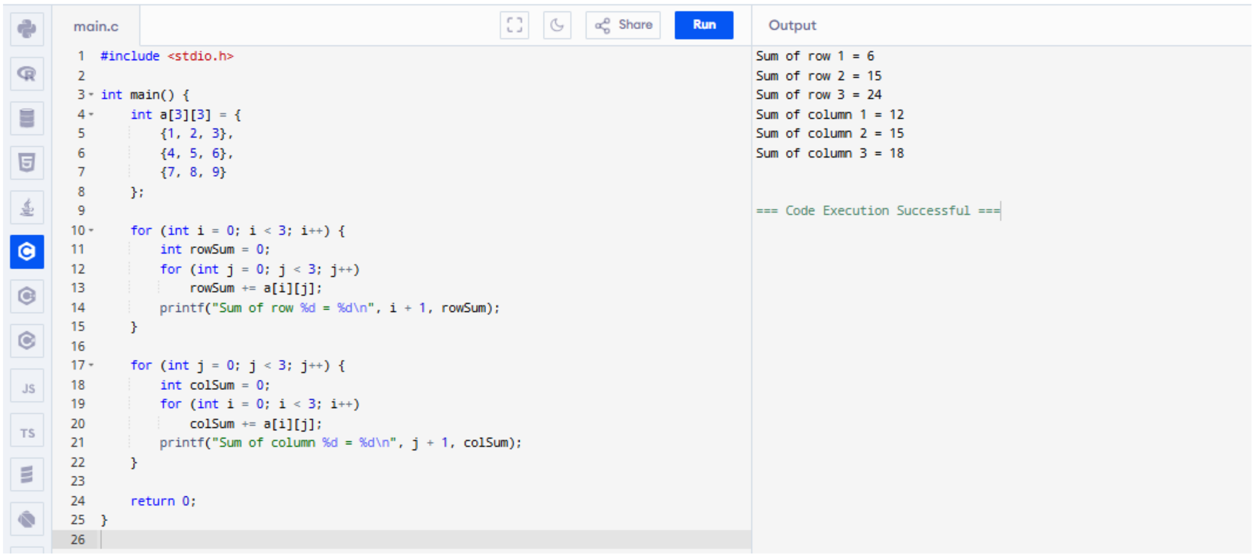
Result:

The program was successfully executed and a doubly linked list was created. Nodes were inserted and displayed in both directions.

39. Sum of Row and Column – 2D Array

Aim:

To compute and print the sum of each row and column in a 2D matrix.



```c
#include <stdio.h>

int main() {
    int a[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    for (int i = 0; i < 3; i++) {
        int rowSum = 0;
        for (int j = 0; j < 3; j++)
            rowSum += a[i][j];
        printf("Sum of row %d = %d\n", i + 1, rowSum);
    }

    for (int j = 0; j < 3; j++) {
        int colSum = 0;
        for (int i = 0; i < 3; i++)
            colSum += a[i][j];
        printf("Sum of column %d = %d\n", j + 1, colSum);
    }

    return 0;
}
```

```
Sum of row 1 = 6
Sum of row 2 = 15
Sum of row 3 = 24
Sum of column 1 = 12
Sum of column 2 = 15
Sum of column 3 = 18


=== Code Execution Successful ===
```

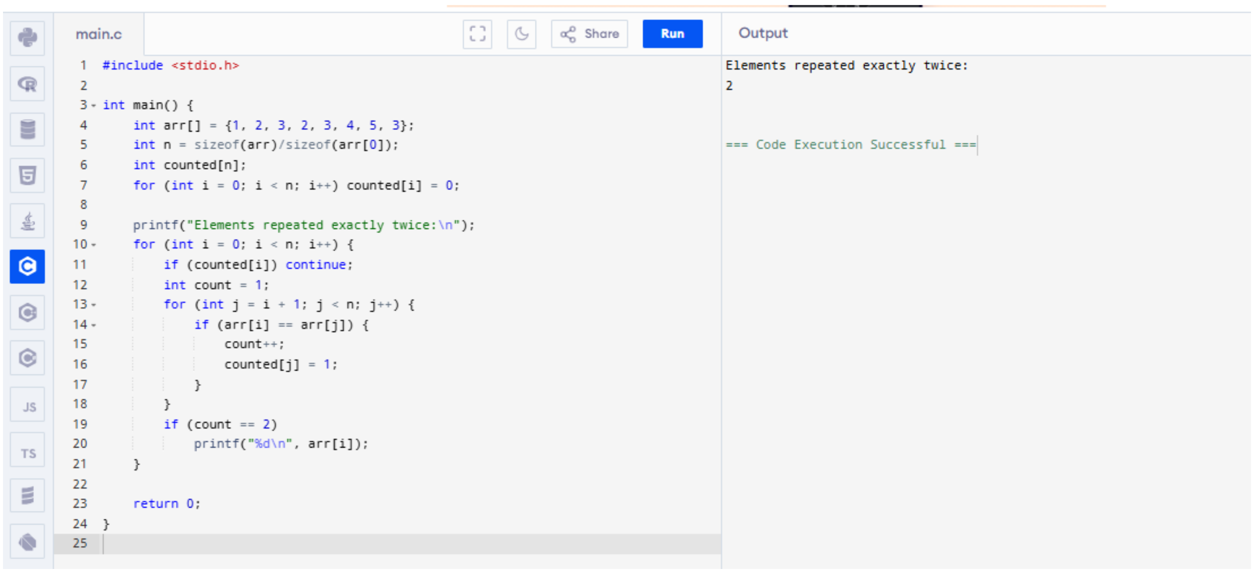
Result :

The program was successfully executed and the sum of each row and each column in the matrix was calculated and displayed.

40. Elements Repeated Twice – Array

Aim:

To find and display all elements that are repeated exactly twice in an array.



```c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 2, 3, 4, 5, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int counted[n];
    for (int i = 0; i < n; i++) counted[i] = 0;

    printf("Elements repeated exactly twice:\n");
    for (int i = 0; i < n; i++) {
        if (counted[i]) continue;
        int count = 1;
        for (int j = i + 1; j < n; j++) {
            if (arr[i] == arr[j]) {
                count++;
                counted[j] = 1;
            }
        }
        if (count == 2)
            printf("%d\n", arr[i]);
    }

    return 0;
}
```

```
Elements repeated exactly twice:
2


=== Code Execution Successful ===
```

Result : The program was successfully executed and elements that occurred exactly twice in the array were identified and displayed.