SIMATS
ENGINEERING

SIMATS
Saveetha Institute of Medical And Technical Sciences
(Declared as Deemed to be University under Section 3 of UGC Act 1956)

LAB REPORT

Submitted in the partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted by

J. Thejeswar Reddy (192210297)

Data Structures For Optimized Memory Usage – CSA0315

Under the Supervisor of

Dr. F. Mary Harin Fernandez
Dr. K. Sashi Rekha

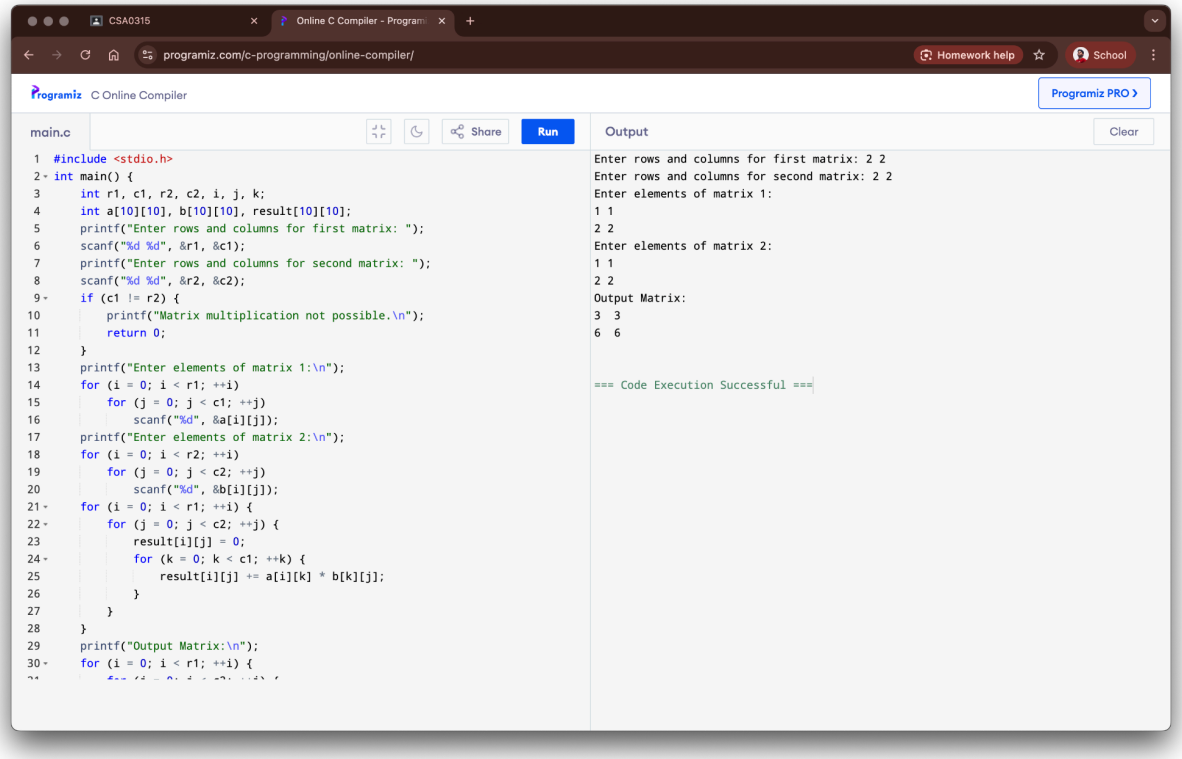
Sep - 2025

1. Matrix Multiplication

Aim:
To write a C program to perform the multiplication of two matrices.

Algorithm:

1. Start the program.
2. Read the order of the first matrix (r1, c1) and the second matrix (r2, c2).
3. Check if c1 is equal to r2. If not, multiplication is not possible.
4. Read the elements of both matrices from the user.
5. Multiply the matrices by iterating through rows and columns and store the result in a new matrix.



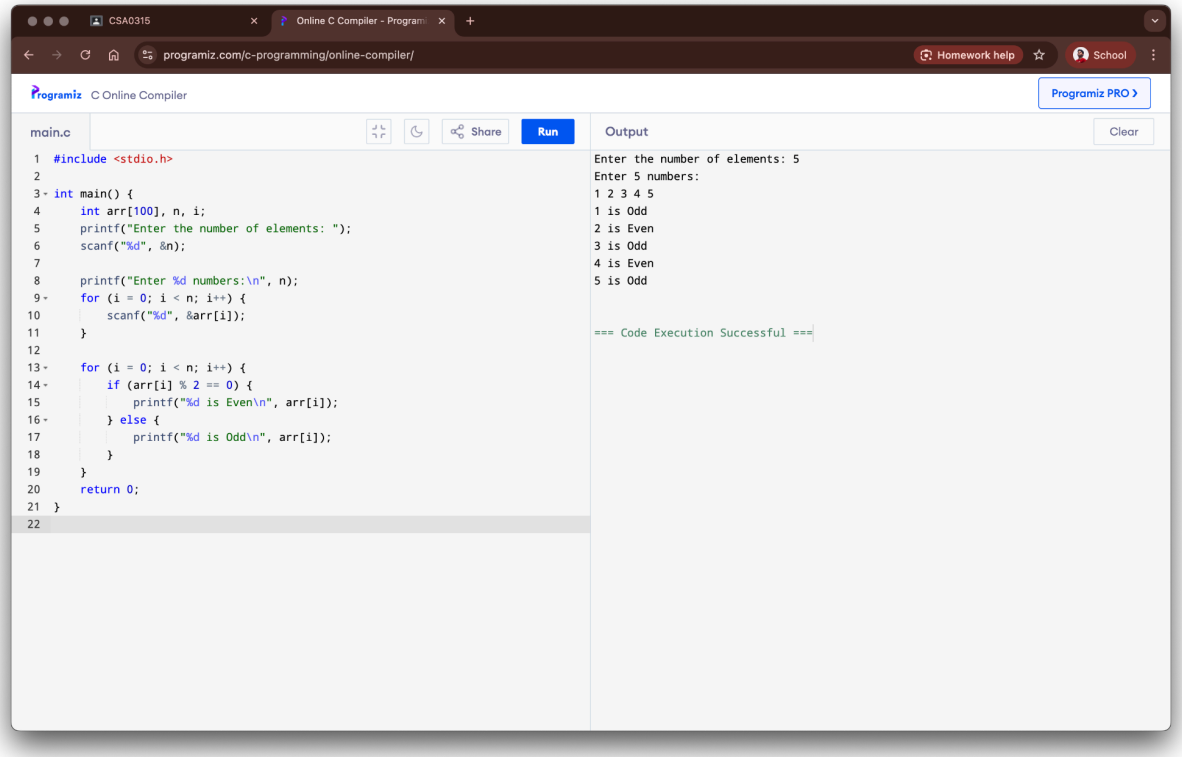
Result:
The program for matrix multiplication was successfully executed.

2. Find Odd or Even Number

Aim:
To write a C program to find whether a number is odd or even from a given set of numbers.

Algorithm:

1. Start the program.
2. Read the total number of elements (n).
3. Read n numbers from the user and store them in an array.
4. Iterate through each number in the array.
5. For each number, check if it is divisible by 2. If yes, it's even; otherwise, it's odd.



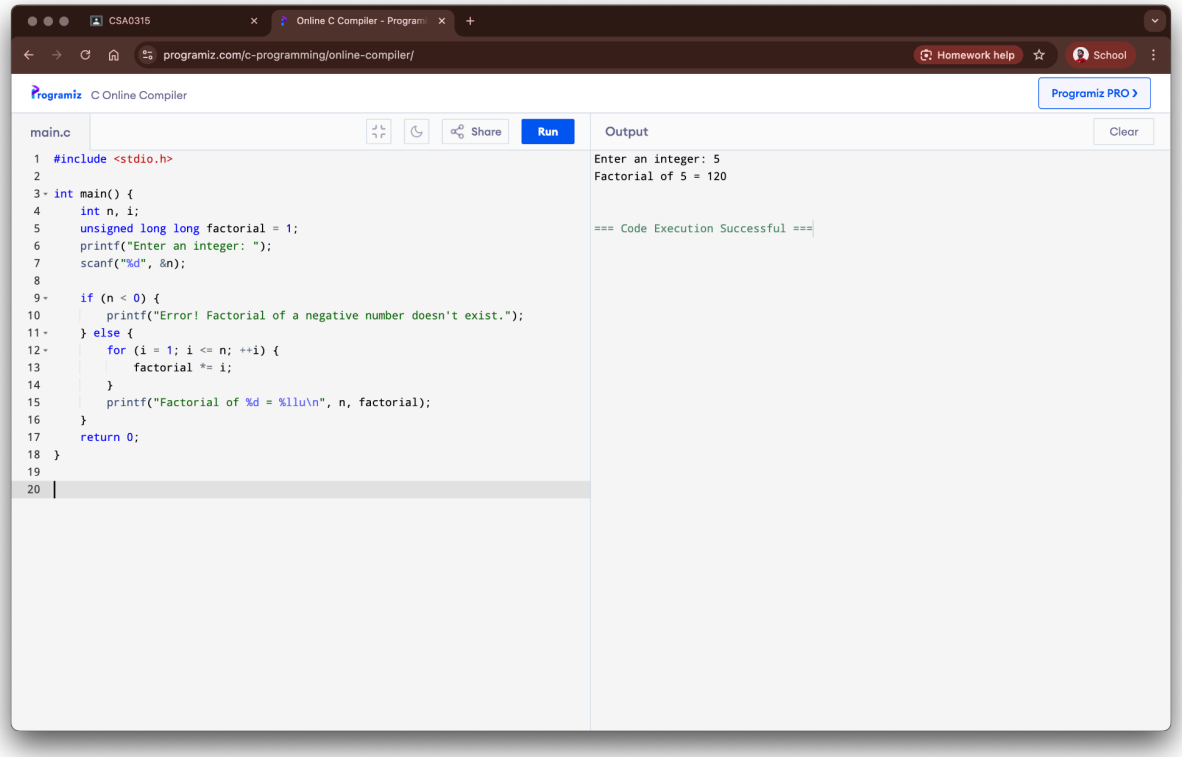
Result:
The program to find odd or even numbers from a set was successfully executed.

3. Factorial of a Given Number (without Recursion)

Aim:
To write a C program to find the factorial of a given number without using recursion.

Algorithm:

1. Start the program.
2. Read an integer 'n' from the user.
3. Initialize a variable 'factorial' to 1.
4. Use a loop to iterate from 1 to n.
5. In each iteration, multiply 'factorial' by the loop counter.



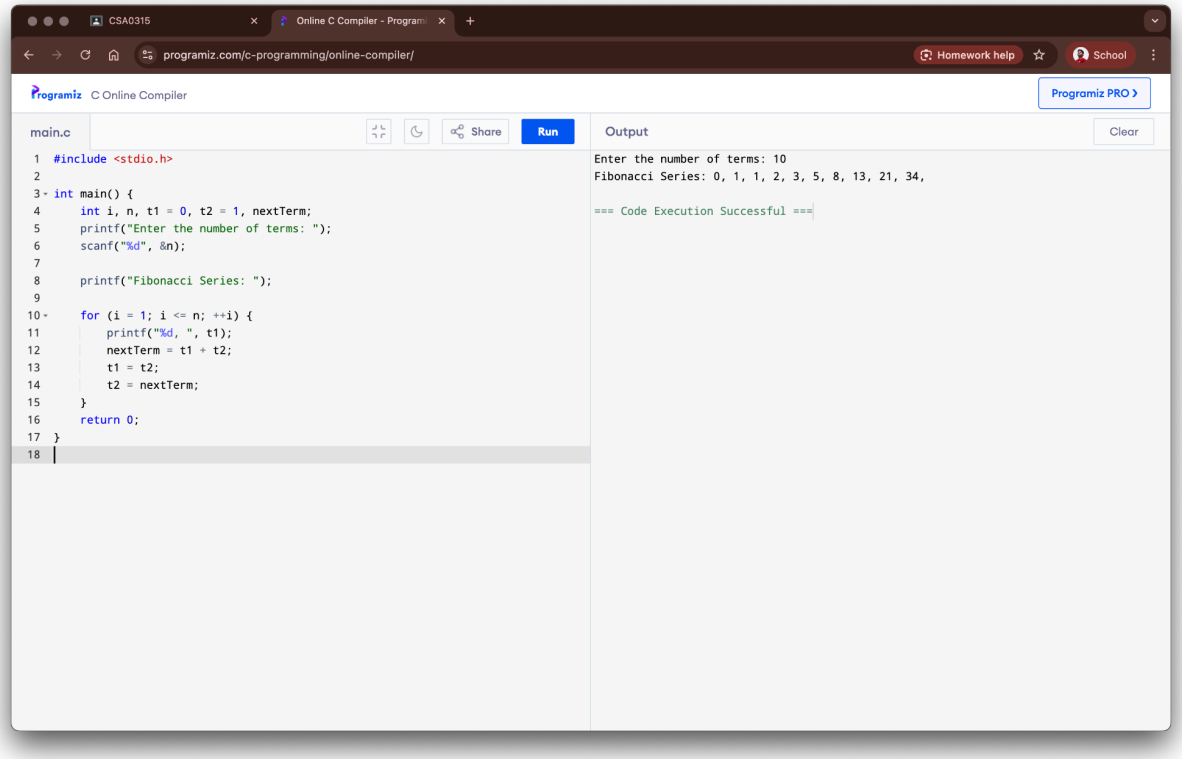
Result:
The program to find the factorial iteratively was successfully executed.

4. Fibonacci Series (without Recursion)

Aim:
To write a C program to generate the Fibonacci series up to a given number of terms without using recursion.

Algorithm:

1. Start the program.
2. Read the number of terms 'n' from the user.
3. Initialize the first two terms, t1 = 0 and t2 = 1.
4. Use a loop to iterate from 1 to n.
5. Print the current term and calculate the next term by summing the previous two.



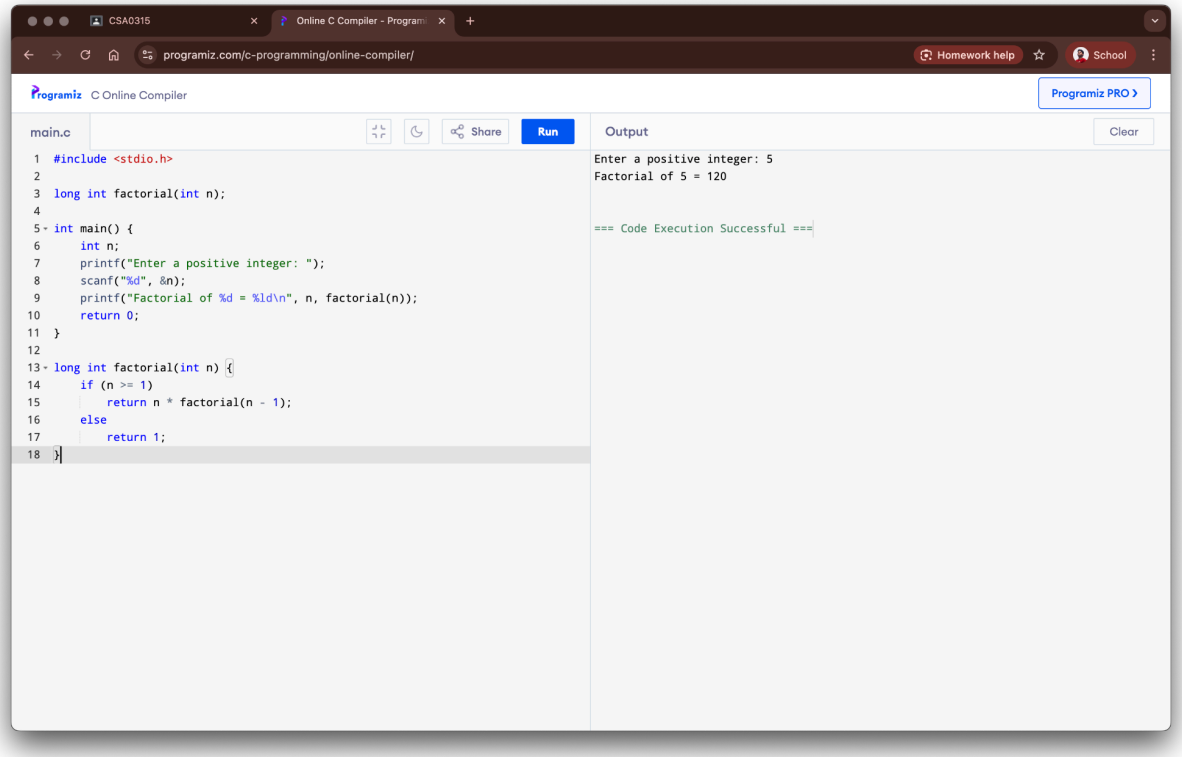
Result:
The program to generate the Fibonacci series iteratively was successfully executed.

5. Factorial of a Given Number (using Recursion)

Aim:
To write a C program to find the factorial of a given number using recursion.

Algorithm:

1. Start the program.
2. Read an integer 'n' from the user.
3. Define a recursive function factorial(num).
4. The base case for the recursion is when num is 0 or 1, return 1.
5. Otherwise, return num * factorial(num - 1).



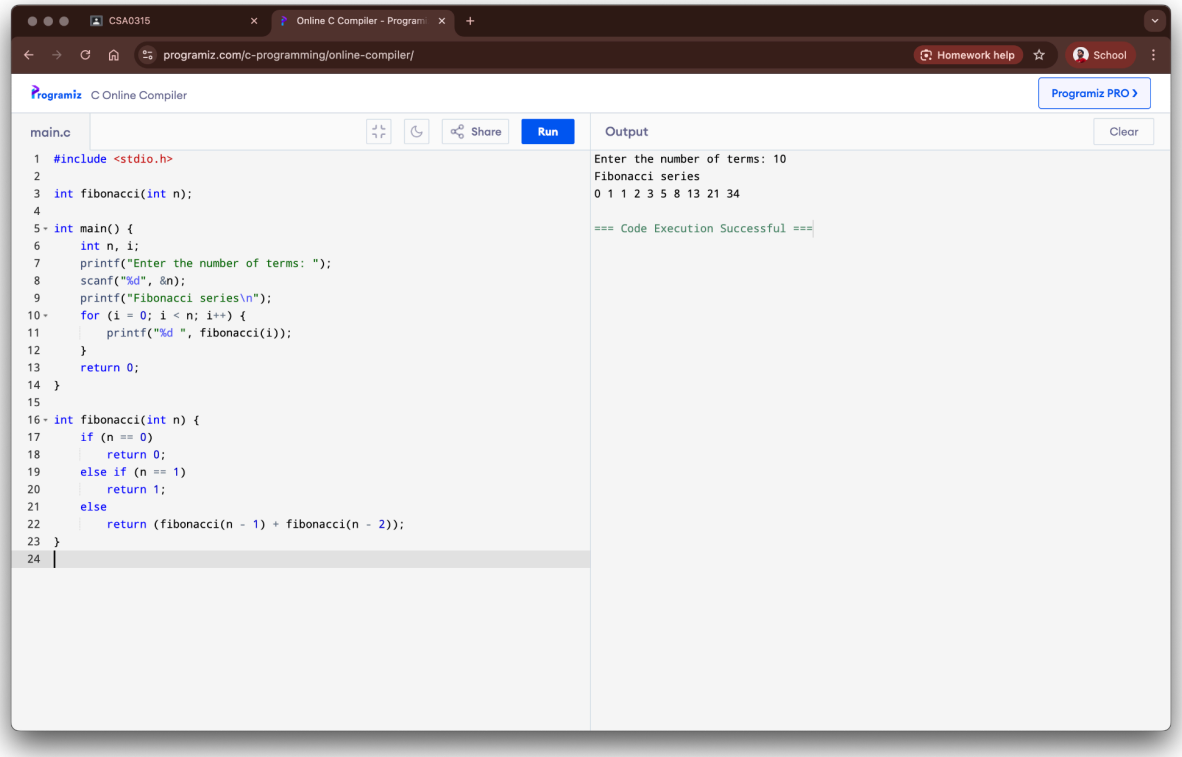
Result:
The program to find the factorial recursively was successfully executed.

6. Fibonacci Series (using Recursion)

Aim:
To write a C program to generate the Fibonacci series using recursion.

Algorithm:

1. Start the program.
2. Read the number of terms 'n' from the user.
3. Define a recursive function fibonacci(num).
4. The base cases are: if num is 0, return 0; if num is 1, return 1.
5. Otherwise, return fibonacci(num - 1) + fibonacci(num - 2).



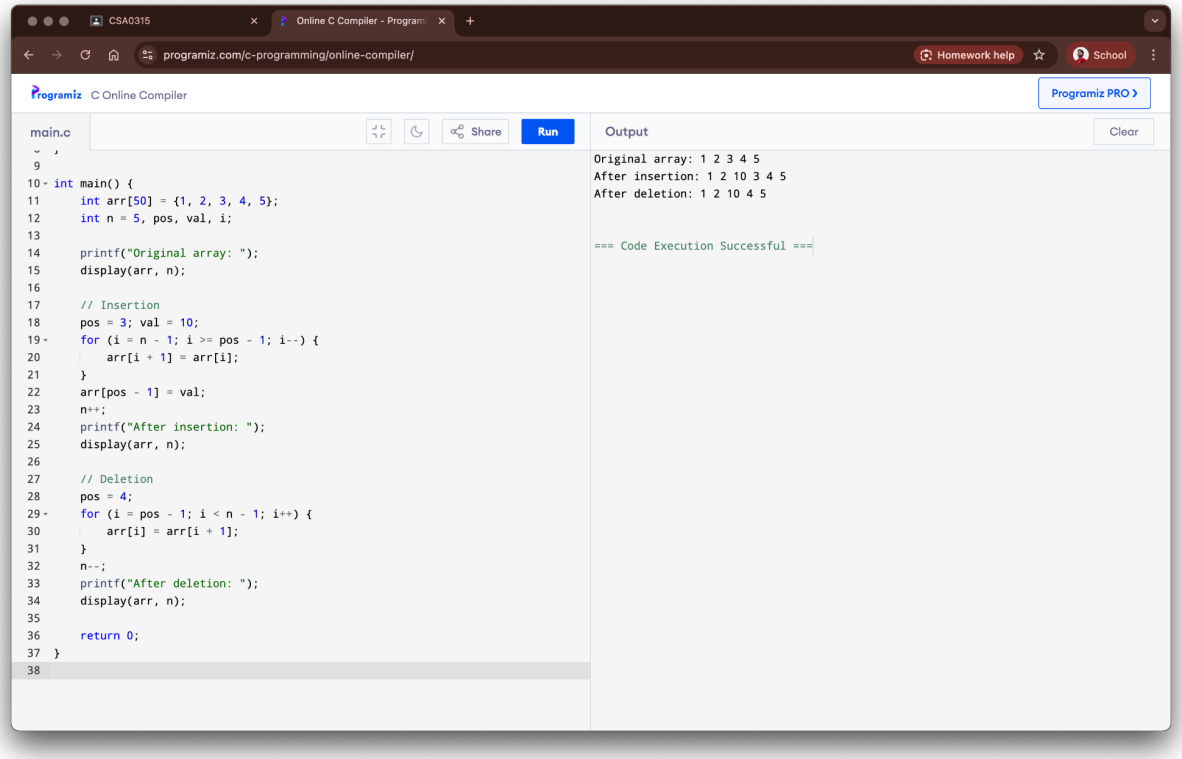
Result:
The program to generate the Fibonacci series recursively was successfully executed.

7. Array Operations (Insert, Delete, Display)

Aim:
To write a C program to perform insertion, deletion, and display operations on an array.

Algorithm:

1. Start the program.
2. Create an array and initialize its size.
3. For insertion, shift elements to the right from the given position and insert the new element.
4. For deletion, shift elements to the left from the position of the element to be deleted.
5. For display, iterate through the array and print all its elements.

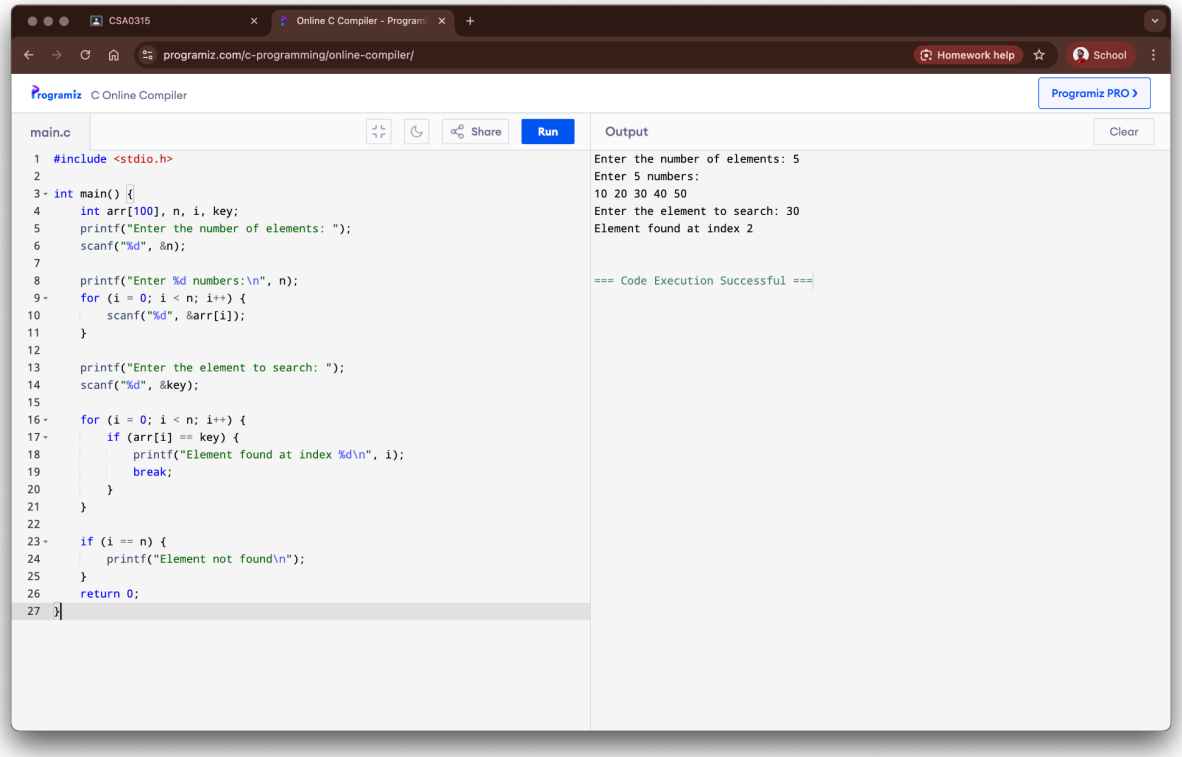


Result:
The program for array operations was successfully executed.

8. Linear Search

Aim:
To write a C program to search for an element in an array using the linear search method.
Algorithm:

1. Start the program.
2. Read the array elements and the search key.
3. Start from the first element of the array.
4. Compare the search key with each element of the array.
5. If a match is found, return the index; otherwise, continue until the end of the array.



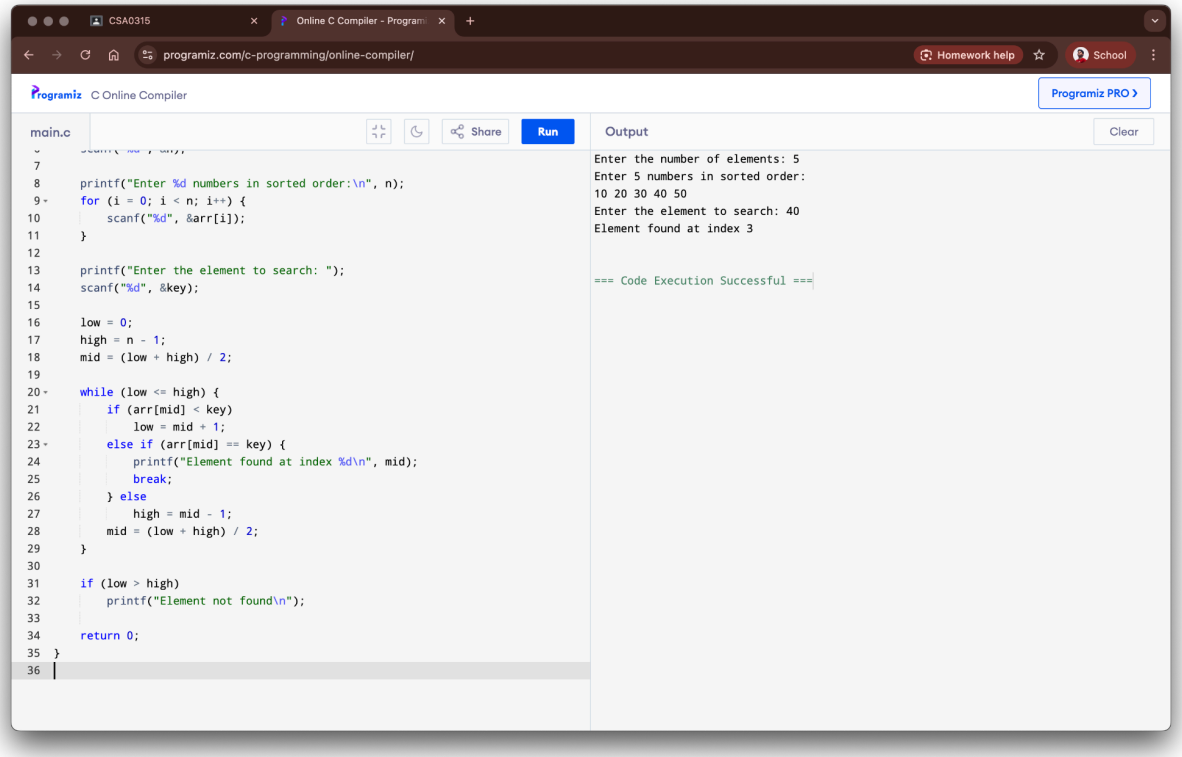
Result:
The program for linear search was successfully executed.

9. Binary Search

Aim:
To write a C program to search for an element in a sorted array using the binary search method.

Algorithm:

1. Start the program.
2. Read the sorted array elements and the search key.
3. Set low = 0 and high = n-1.
4. Find the middle element: mid = (low + high) / 2.
5. If key equals mid, the element is found. If key is less than mid, set high = mid - 1. If key is greater than mid, set low = mid + 1. Repeat until found or low > high.



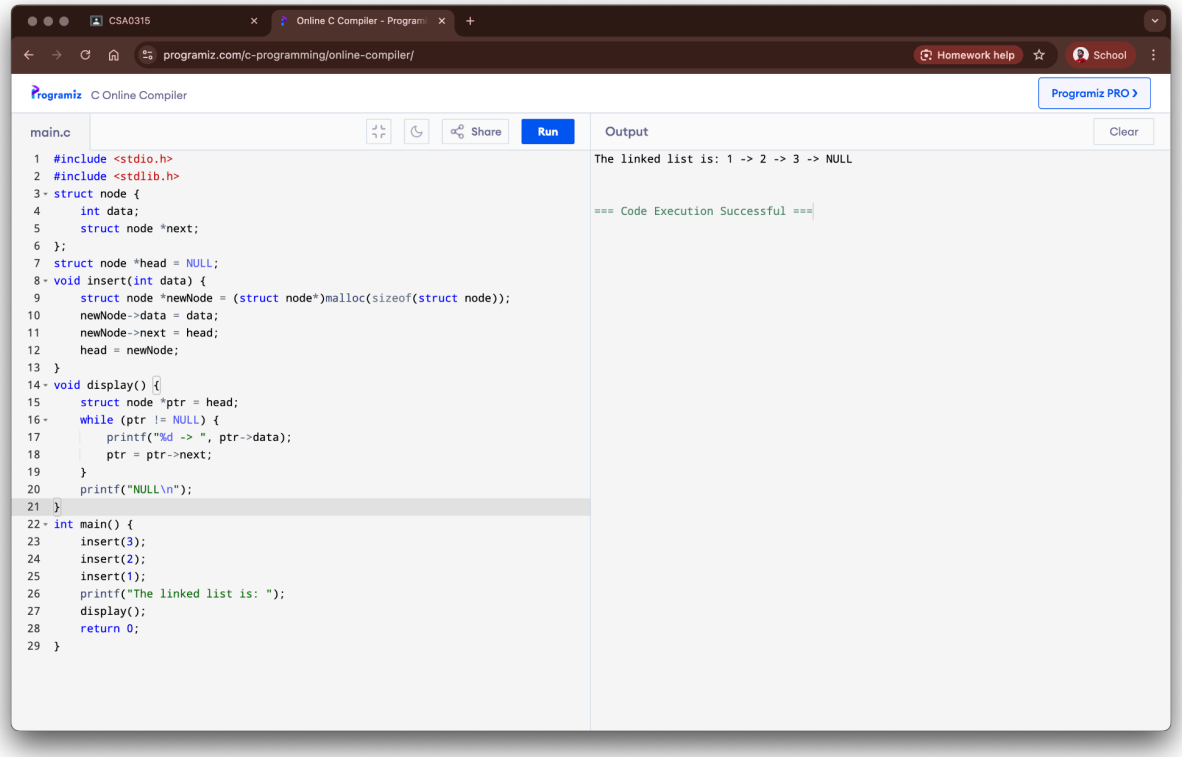
Result:
The program for binary search was successfully executed.

10. Linked List Operations

Aim:
To write a C program to implement basic operations (insert, delete, display) on a singly linked list.

Algorithm:

1. Define a structure for a node with data and a next pointer.
2. For insertion, create a new node, allocate memory, and adjust pointers to add it to the list.
3. For deletion, find the node to be deleted, adjust the pointers of the previous node, and free the memory.
4. For display, traverse the list from the head node and print the data of each node.
5. Implement a menu-driven program to perform these operations.



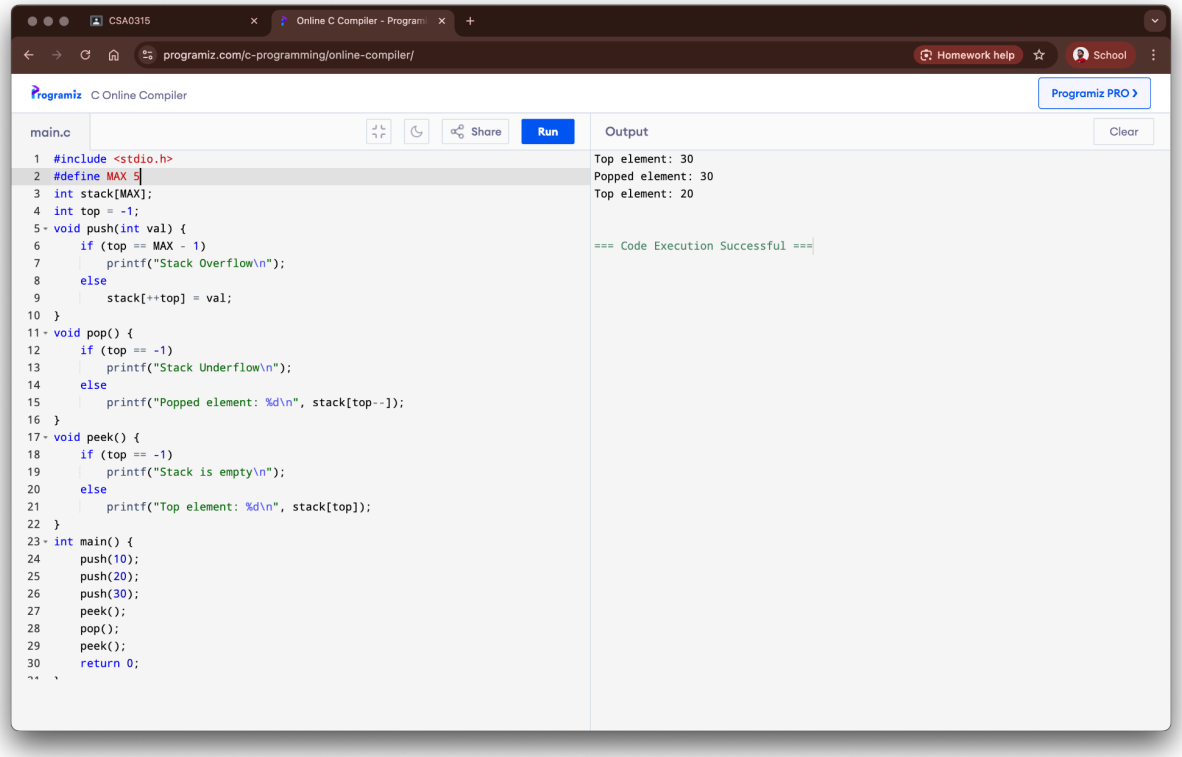
Result:
The program to implement linked list operations was successfully executed.

11. Stack Operations (PUSH, POP, PEEK)

Aim:
To write a C program to implement stack operations (PUSH, POP, PEEK) using an array.

Algorithm:

1. Declare a global array for the stack and a variable top initialized to -1.
2. For PUSH, check for overflow. If not, increment top and add the element.
3. For POP, check for underflow. If not, return the element at top and decrement top.
4. For PEEK, check for underflow. If not, return the element at top without decrementing.
5. Implement a menu-driven program to perform these operations.



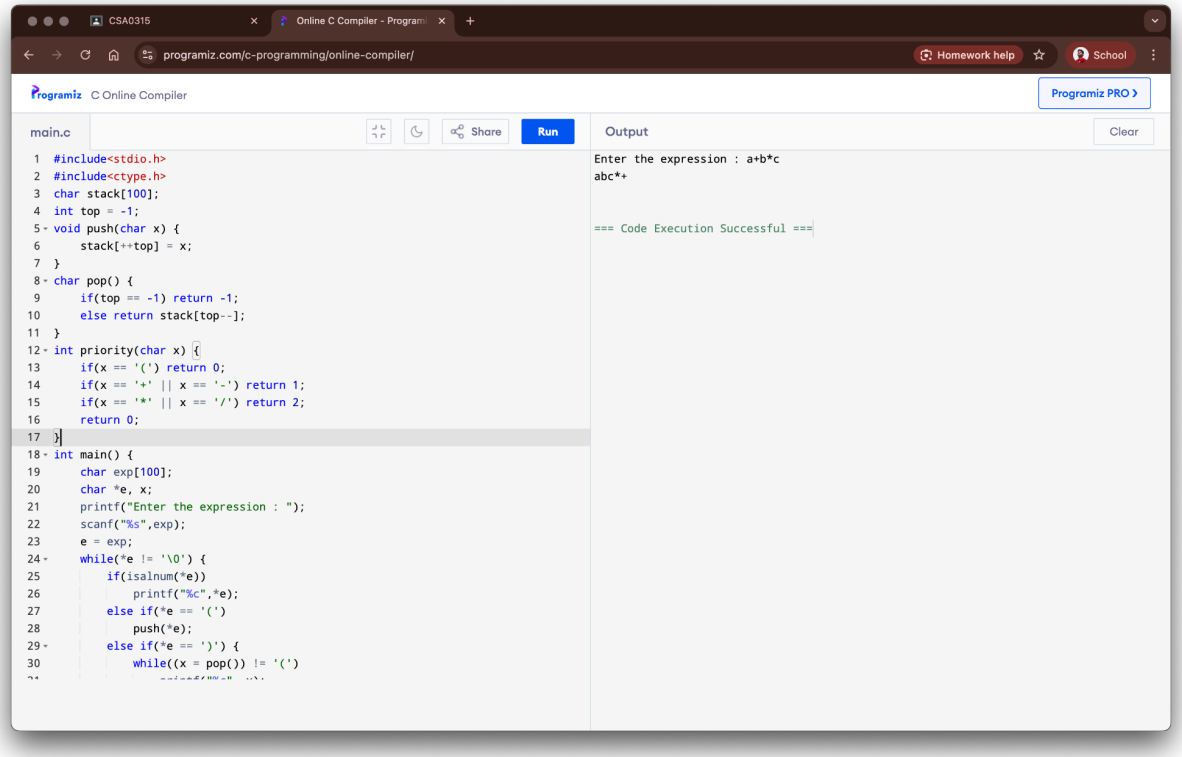
Result:
The program to implement stack operations was successfully executed.

12. Application of Stack (Infix to Postfix)

Aim:
To write a C program to convert an infix expression to a postfix expression.

Algorithm:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. If the scanned character is an operator, and stack is empty or contains a lower precedence operator, push it.
4. If the scanned character is an operator and has lower or equal precedence than the top of the stack, pop the stack and output. Then test the incoming operator against the new top.
5. After scanning all characters, pop and output any remaining operators from the stack.



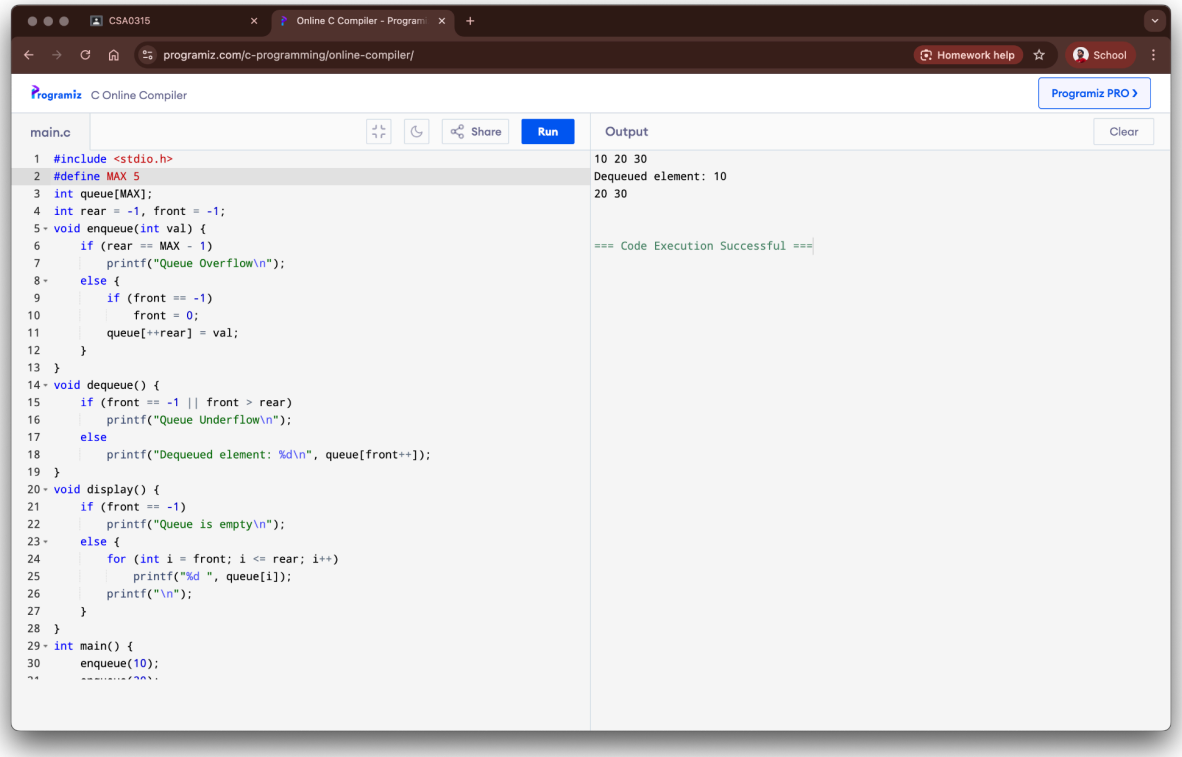
Result:
The program for infix to postfix conversion was successfully executed.

13. Queue Operations (ENQUEUE, DEQUEUE, Display)

Aim:
To write a C program to implement queue operations (ENQUEUE, DEQUEUE, Display) using an array.

Algorithm:

1. Declare a global array for the queue and variables front and rear initialized to -1.
2. For ENQUEUE, check for overflow. If not, increment rear and add the element. Set front to 0 if it's the first element.
3. For DEQUEUE, check for underflow. If not, return the element at front and increment front.
4. For Display, check if the queue is empty. If not, print elements from front to rear.
5. Implement a menu-driven program for these operations.



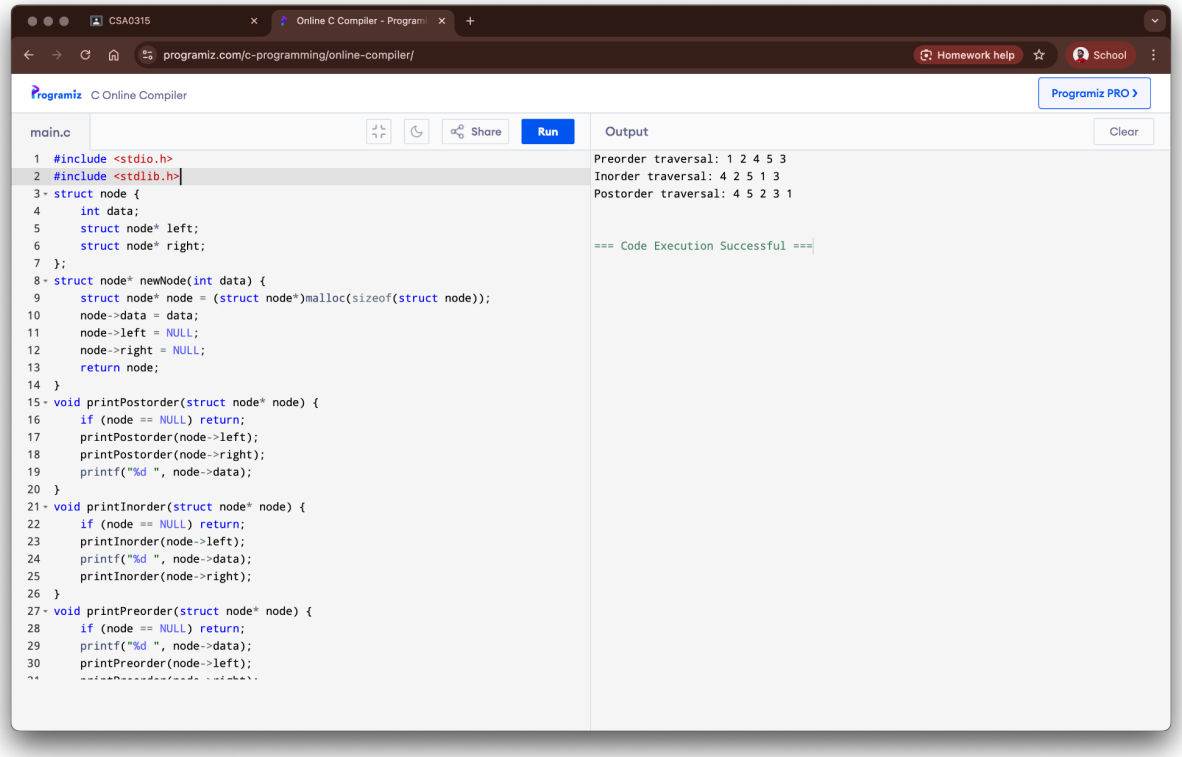
Result:
The program to implement queue operations was successfully executed.

14. Tree Traversals (Inorder, Preorder, Postorder)

Aim:
To write a C program to implement inorder, preorder, and postorder traversals of a binary tree.

Algorithm:

1. Define a structure for a tree node with data, left child, and right child pointers.
2. **Preorder:** Visit the root, traverse the left subtree, then traverse the right subtree (Root-Left-Right).
3. **Inorder:** Traverse the left subtree, visit the root, then traverse the right subtree (Left-Root-Right).
4. **Postorder:** Traverse the left subtree, traverse the right subtree, then visit the root (Left-Right-Root).
5. Create a sample binary tree and call the traversal functions.



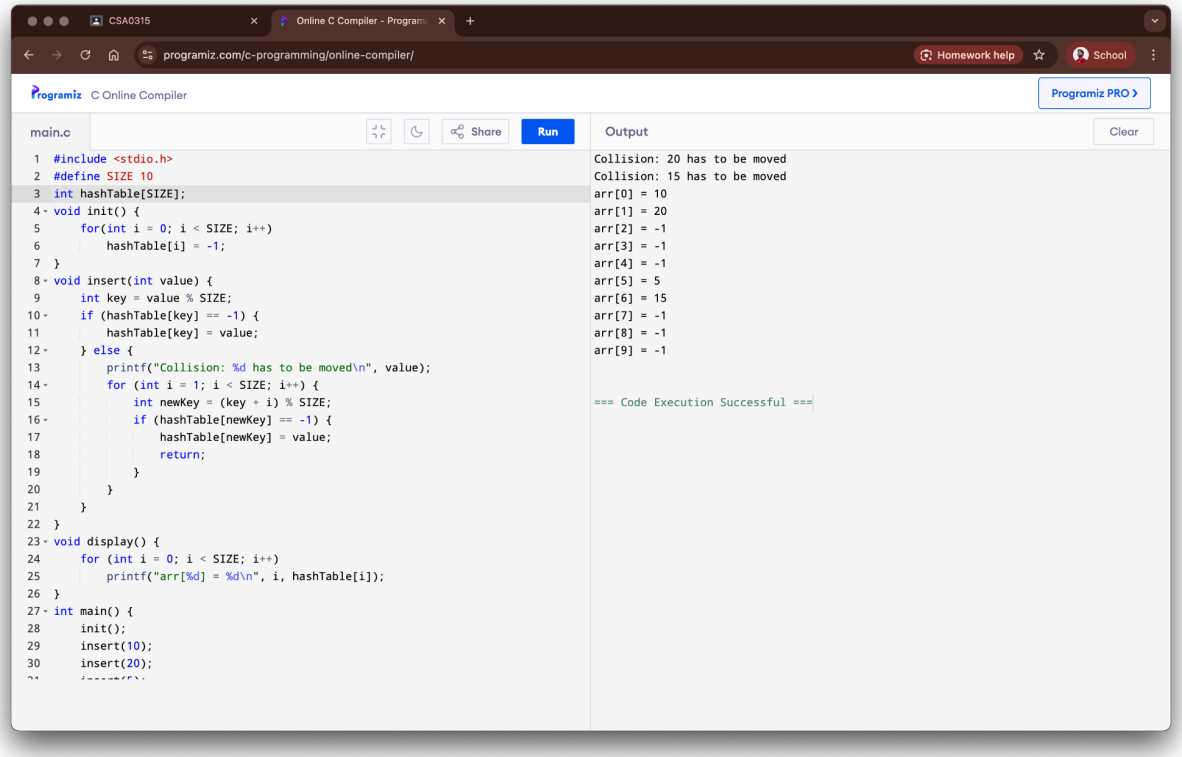
Result:
The program for tree traversals was successfully executed.

15. Hashing using Linear Probing

Aim:
To write a C program to implement hashing with the linear probing method for collision resolution.

Algorithm:

1. Create a hash table (an array) and initialize all its slots to a sentinel value (e.g., -1).
2. Define a hash function, h(key) = key % size.
3. To insert a key, compute the hash index. If the slot is empty, place the key.
4. If the slot is occupied (collision), probe subsequent slots linearly ((index + 1) % size) until an empty slot is found.
5. To search for a key, compute the hash index and probe linearly until the key is found or an empty slot is encountered.



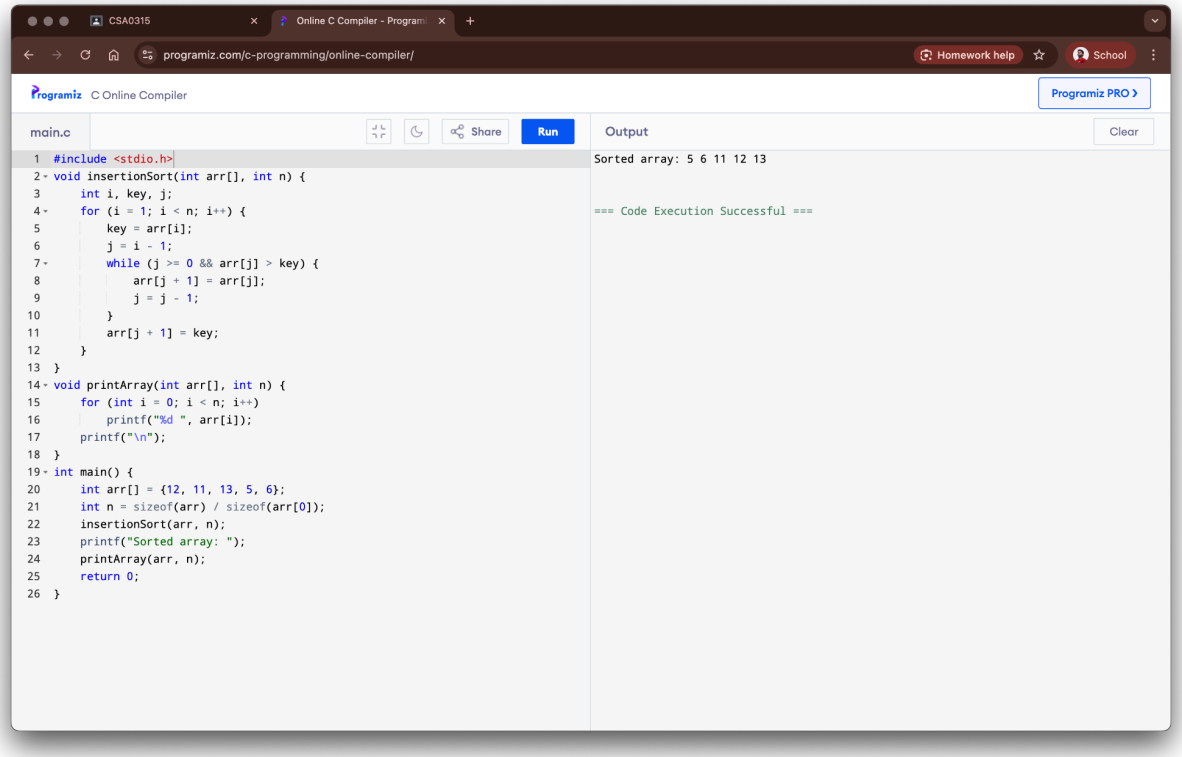
Result:
The program for hashing using linear probing was successfully executed.

16. Insertion Sort

Aim:
To write a C program to sort an array of numbers using the insertion sort algorithm.

Algorithm:

1. Start from the second element of the array.
2. Compare the current element (key) with its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before.
4. Move the greater elements one position up to make space for the swapped element.
5. Repeat this process for all elements in the array.



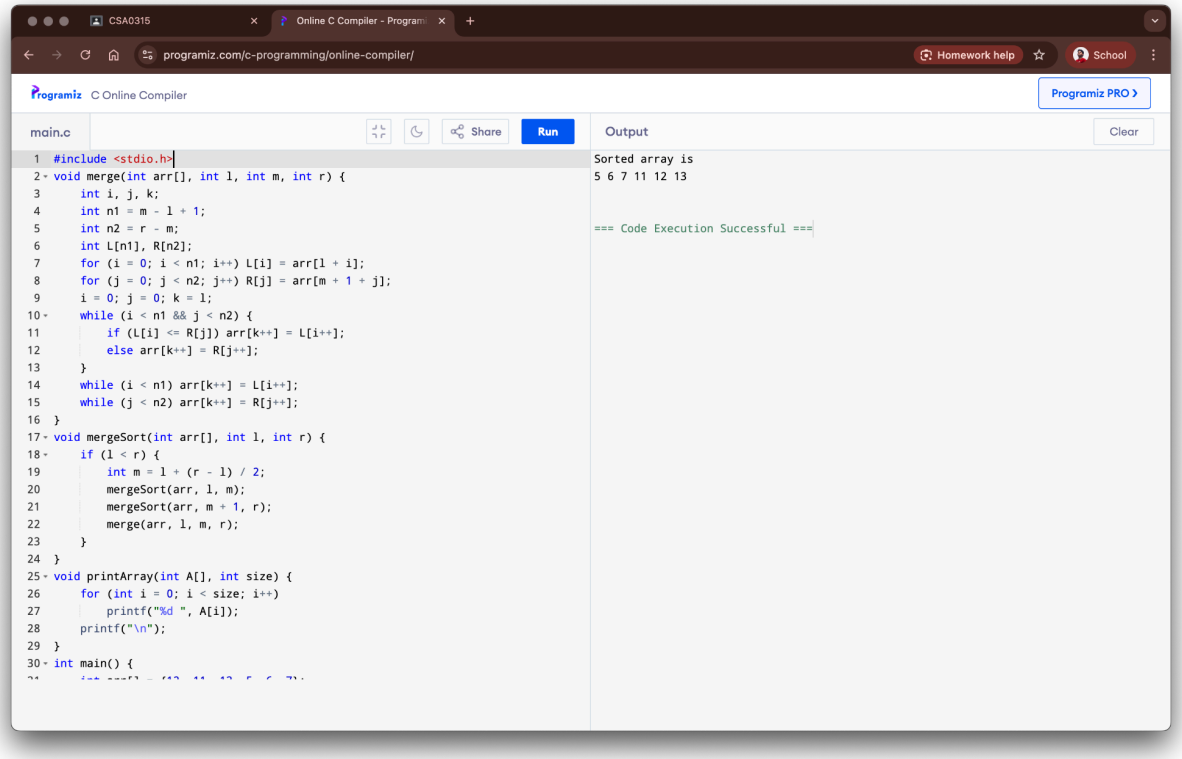
Result:
The program for insertion sort was successfully executed.

17. Merge Sort

Aim:
To write a C program to sort an array of numbers using the merge sort algorithm.

Algorithm:

1. Divide the unsorted list into n sublists, each containing one element.
2. Repeatedly merge sublists to produce new sorted sublists.
3. The merge step involves comparing elements from two sublists and placing the smaller one into the merged list.
4. Continue merging until there is only one sublist remaining.
5. This will be the sorted list.



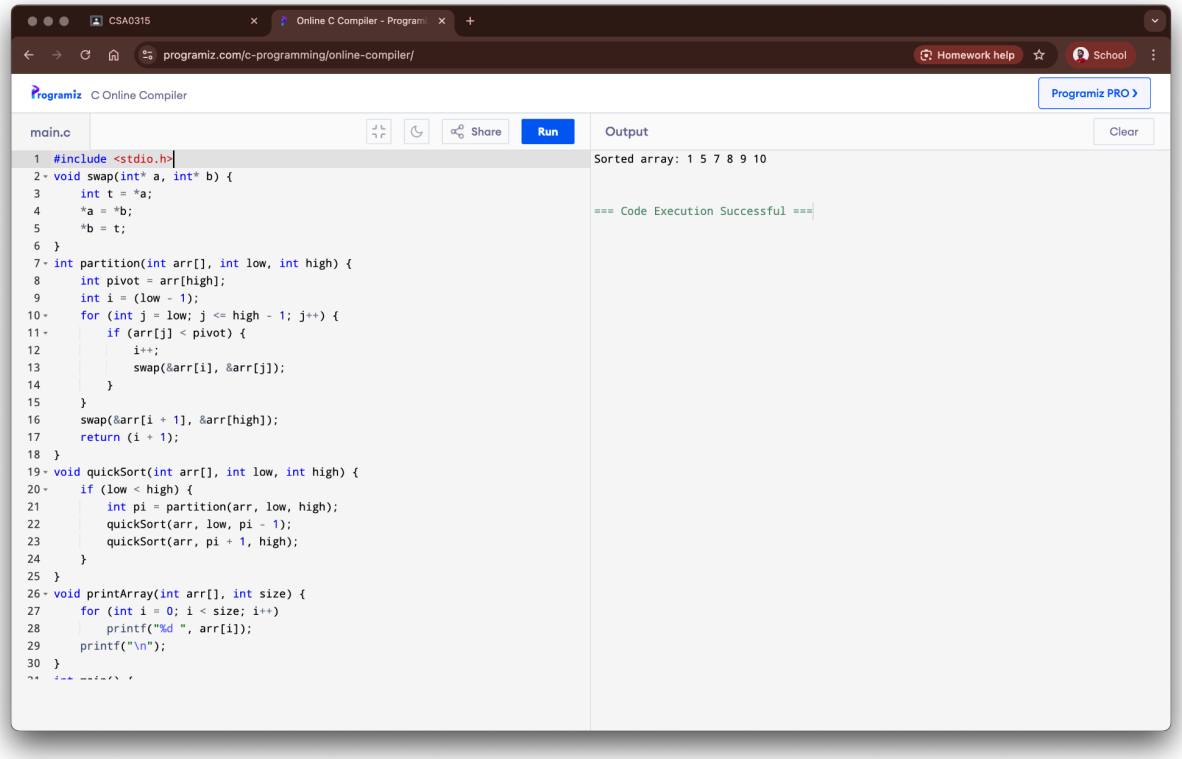
Result:
The program for merge sort was successfully executed.

18. Quick Sort

Aim:
To write a C program to sort an array of numbers using the quick sort algorithm.

Algorithm:

1. Pick an element as a pivot (e.g., the last element).
2. Partition the array around the pivot by placing all smaller elements before it and all greater elements after it.
3. The pivot is now in its final sorted position.
4. Recursively apply the above steps to the sub-array of elements with smaller values.
5. Recursively apply the steps to the sub-array of elements with greater values.



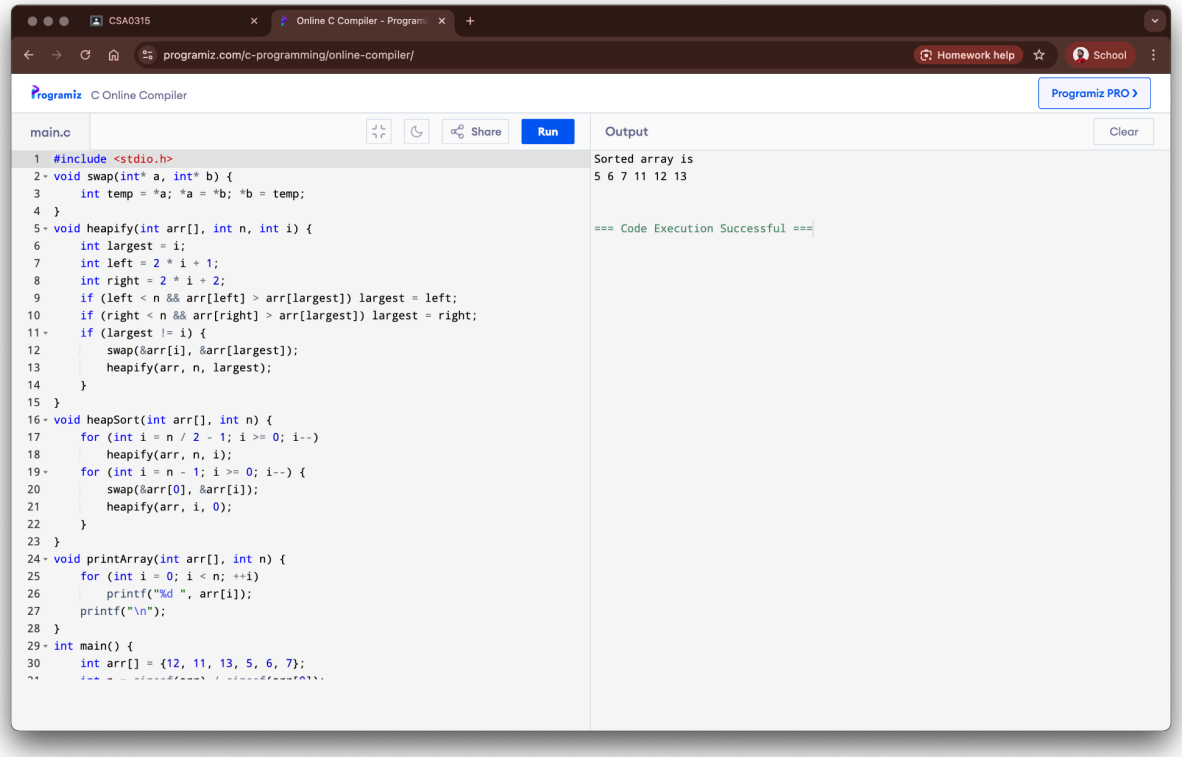
Result:
The program for quick sort was successfully executed.

19. Heap Sort

Aim:
To write a C program to sort an array of numbers using the heap sort algorithm.

Algorithm:

1. Build a max heap from the input data.
2. The largest item is stored at the root of the heap.
3. Replace the root with the last item of the heap and reduce the size of the heap by one.
4. Heapify the root of the tree to maintain the heap property.
5. Repeat steps 3 and 4 while the size of the heap is greater than 1.



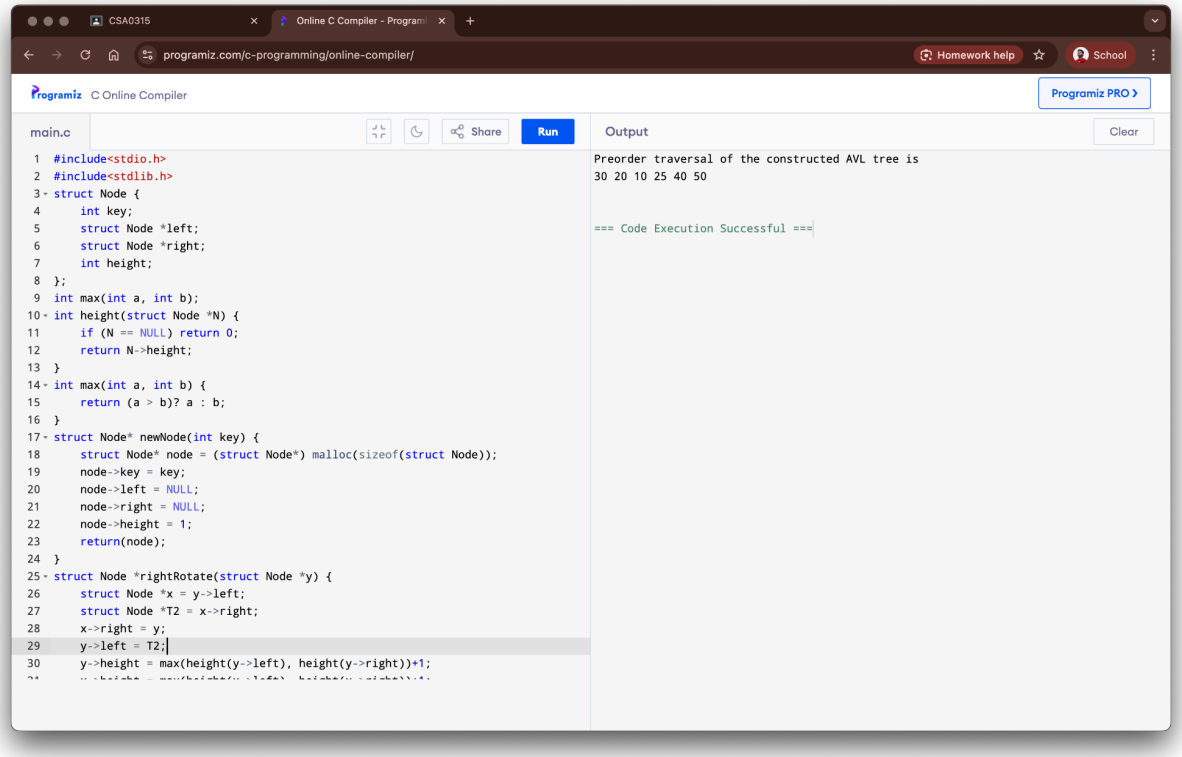
Result:
The program for heap sort was successfully executed.

20. AVL Tree Operations

Aim:
To write a C program to perform insertion, deletion, and searching operations on an AVL tree.

Algorithm:

1. Define a node structure with data, height, and left/right child pointers.
2. **Insertion:** Perform a standard BST insert. Then, trace back to the root, updating heights and performing rotations (LL, RR, LR, RL) to balance the tree if the balance factor is > 1 or < -1.
3. **Deletion:** Perform a standard BST delete. Then, trace back to the root, updating heights and performing rotations to rebalance the tree.
4. **Search:** Perform a standard BST search.
5. The balance factor of a node is the height of its left subtree minus the height of its right subtree.



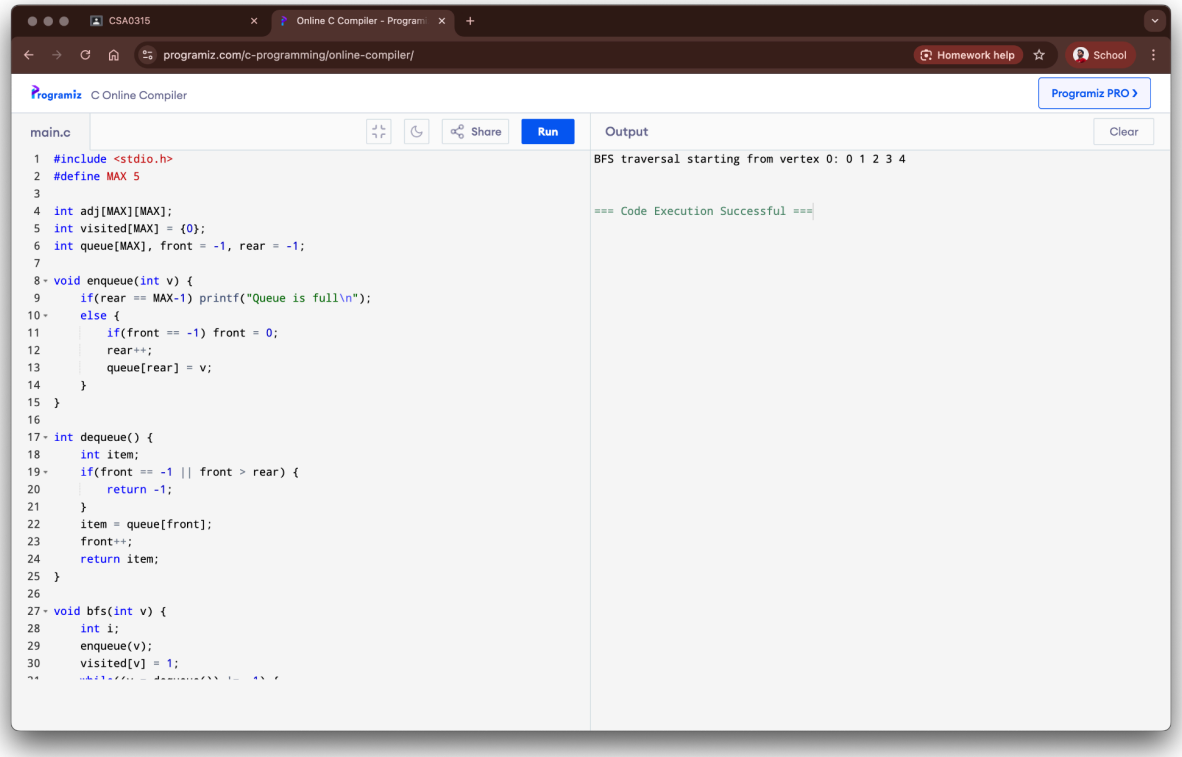
Result:
The program for AVL tree operations was successfully executed.

21. Graph Traversal using Breadth First Search (BFS)

Aim:
To write a C program to traverse a graph using the Breadth First Search (BFS) algorithm.

Algorithm:

1. Start at a source vertex and add it to a queue. Mark it as visited.
2. While the queue is not empty, dequeue a vertex.
3. Visit all its unvisited adjacent vertices.
4. Mark them as visited and enqueue them.
5. Repeat until the queue is empty.



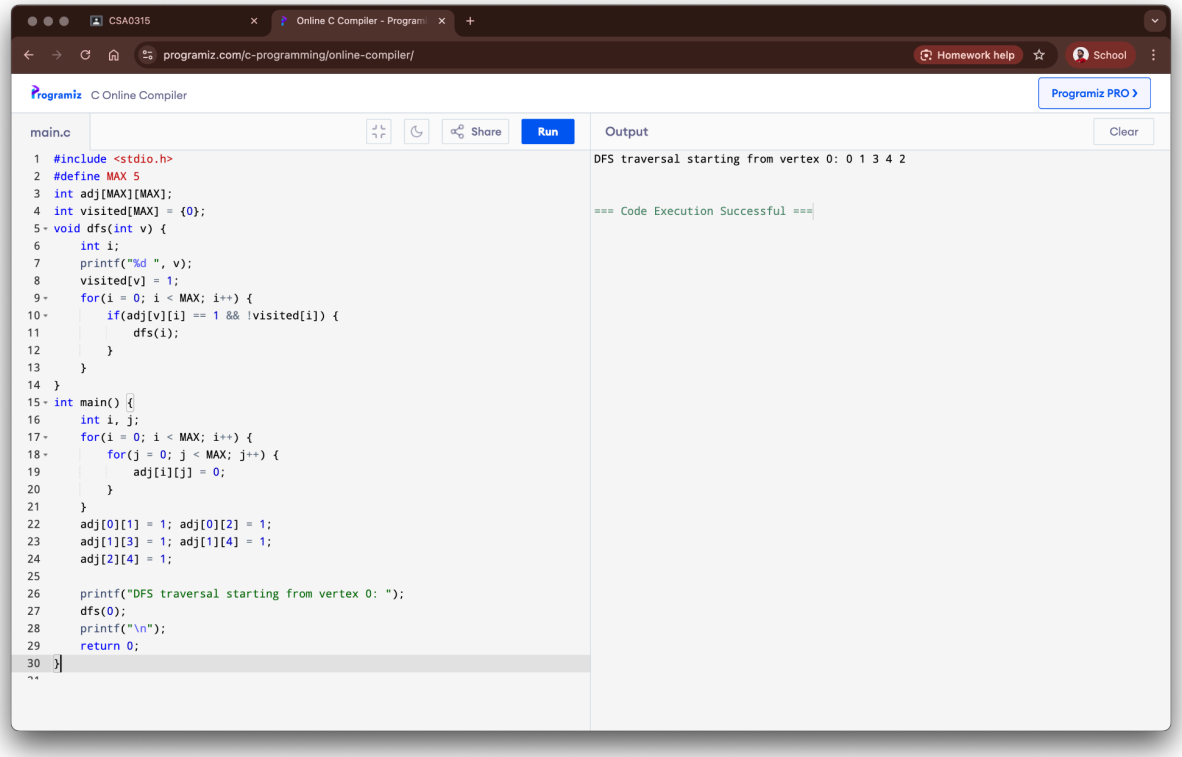
Result:
The program for BFS graph traversal was successfully executed.

22. Graph Traversal using Depth First Search (DFS)

Aim:
To write a C program to traverse a graph using the Depth First Search (DFS) algorithm.

Algorithm:

1. Start at a source vertex. Mark it as visited.
2. Explore as far as possible along each branch before backtracking.
3. Recursively visit an unvisited adjacent vertex of the current vertex.
4. Mark it as visited and repeat the process.
5. If there are no unvisited adjacent vertices, backtrack.



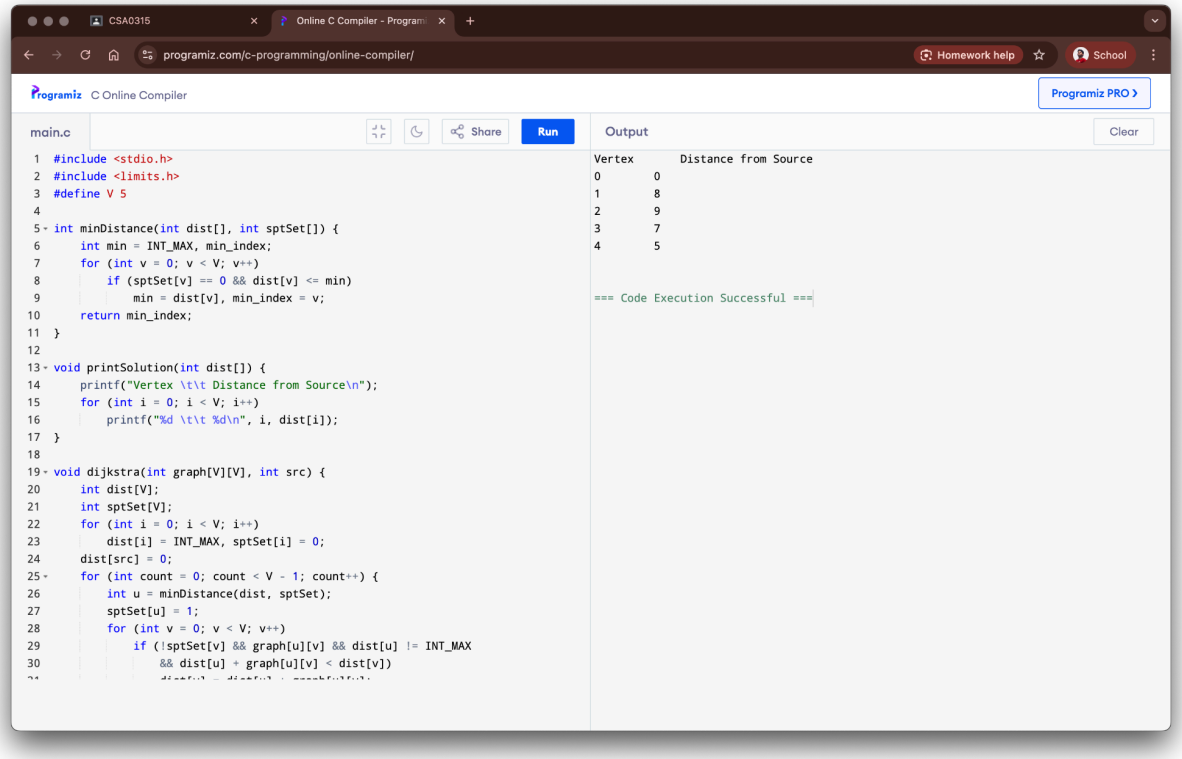
Result:
The program for DFS graph traversal was successfully executed.

23. Shortest Path using Dijkstra's Algorithm

Aim:
To implement Dijkstra's algorithm to find the shortest path from a source vertex to all other vertices in a weighted graph.

Algorithm:

1. Initialize distances to all vertices as infinite and the source vertex as 0.
2. Create a set of unvisited vertices.
3. While the unvisited set is not empty, select the vertex u with the smallest distance.
4. For each neighbor v of u, update its distance if the path through u is shorter.
5. Remove u from the unvisited set. Repeat until all vertices are visited.



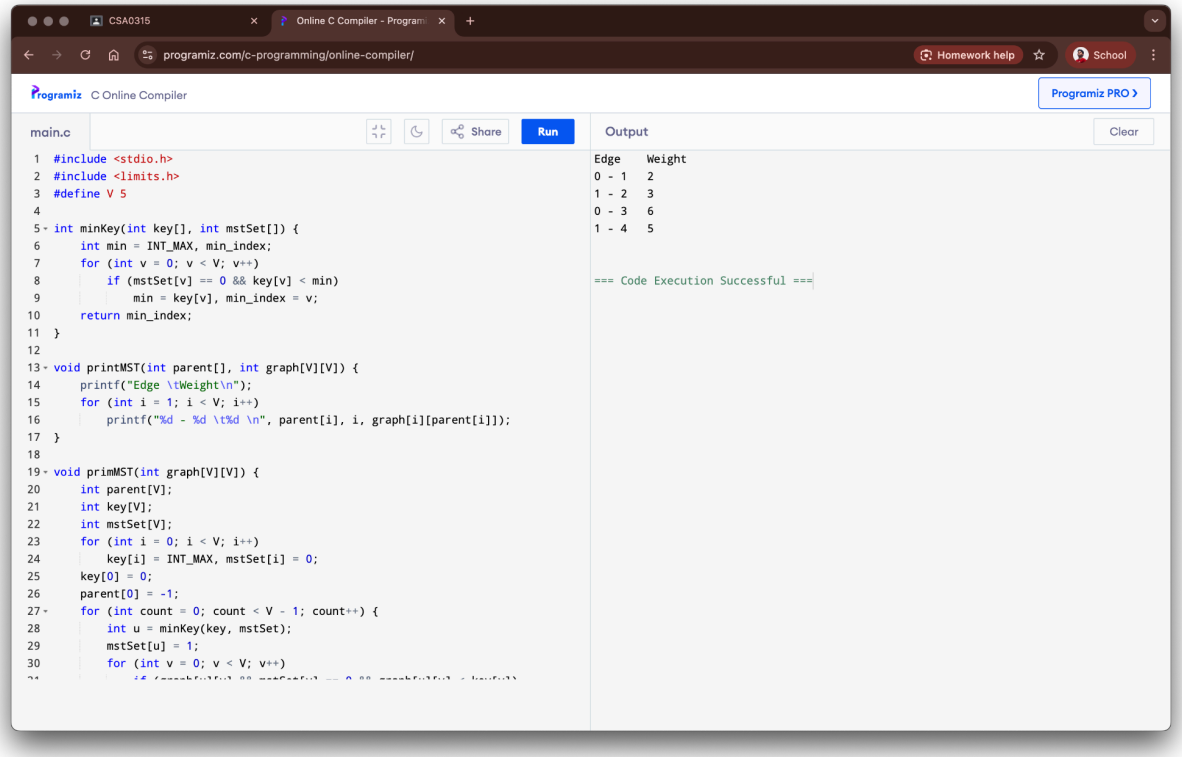
Result:
The program for Dijkstra's algorithm was successfully executed.

24. Minimum Spanning Tree using Prim's Algorithm

Aim:
To implement Prim's algorithm to find the Minimum Spanning Tree (MST) of a connected, undirected, and weighted graph.

Algorithm:

1. Initialize the MST with a single vertex (e.g., the first one).
2. Maintain a set of vertices already included in the MST.
3. In each step, find the edge with the minimum weight that connects a vertex in the MST to a vertex outside the MST.
4. Add this edge and the corresponding vertex to the MST.
5. Repeat until all vertices are included in the MST.



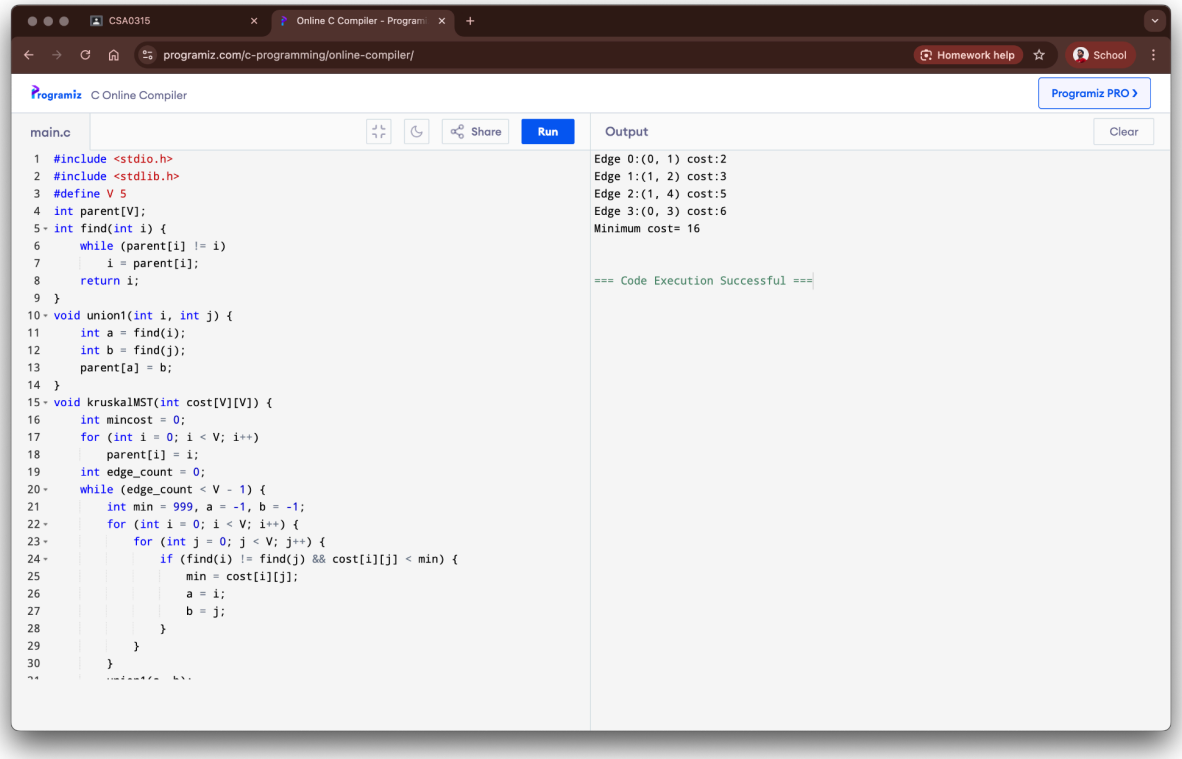
Result:
The program for Prim's algorithm was successfully executed.

25. Minimum Spanning Tree using Kruskal's Algorithm

Aim:
To implement Kruskal's algorithm to find the Minimum Spanning Tree (MST) of a connected, undirected, and weighted graph.

Algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.
3. If no cycle is formed, include this edge. Else, discard it.
4. Repeat step 2 and 3 until there are (V-1) edges in the spanning tree.
5. The 'find' and 'union' operations are used to detect cycles.



Result:
The program for Kruskal's algorithm was successfully executed.