DAA FILE



NAME-SHREYANSH AGRAWAL

ROLL NO.-2021UCS1612

Semester :- 3rd

Subject:- Design and Analysis of Algorithm

Course Code:- COCSC06

"I have done this assignment on my own. I have not copied any code from another student or any online source. I understand if my code is found similar to somebody else's code, my case can be sent to the Disciplinary committee of the institute for appropriate action."

Experiment No. 1

**Objective** : Perform the following sorting algorithms-Merge Sort,QuickSort,InsertionSort,SelectionSort,Bubble Sort.

**Theory** :

*Merge Sort* - The Merge Sort algorithm is Divide and Conquer.The array is initially divided into two equal halves and then they are combined in a sorted manner to get the final array.

*Quick Sort*- It is a Divide and Conquer based algorithm. It picks an element as a pivot and then accordingly swaps elements as per the two pointers.

*Insertion Sort* -Insertion sort is a simple sorting algorithm in programming that works similar to the way you sort playing cards in your hands.

*Selection Sort* -The selection sort algorithm sorts an array by repeatedly finding the minimum element in the array(considering ascending order) from the unsorted part and putting it at the beginning and repeating the steps.

*Bubble Sort*-Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements in the array if they are in the wrong order.

**Algorithm** :

*Merge Sort :*

step 1: start the code

step 2: declare array and left, right, mid variable in array

step 3: perform merge function.

if left > right

return back

mid= (left+right)/2

//mid element found out

mergesort(array, left, mid)

//dividing array in two parts

mergesort(array, mid+1, right)

merge(array, left, mid, right)

//combining the array

step 4: Stop code

*Quick Sort*-

QUICKSORT (array A, start, end)

{

if (start < end)

{

p = partition(A, start, end)

//divide and conquer approach

QUICKSORT (A, start, p - 1)

QUICKSORT (A, p + 1, end)

}

}

*Insertion Sort* -

1.Iterate from arr[1] to arr[N] over the array with loop.

2.Compare the current element to its predecessor in array.

3.If the key element is smaller than its predecessor in array compare it to the elements before.

4.Move the greater elements one position for the swapped element in array.

*Selection Sort*-

1.Initialize minimum value(min) to location 0 from array.

2.Traverse the array to find the min in the array.

3.While traversing if any element smaller than min is found then swap both the values accordingly.

4.Then, increment min/ to point to the next element.

5.Repeat until the array is sorted to get final answer.

*Bubble sort* -

1.Run a nested for loop to traverse the input array using two variables i and j,

2.If arr[j] is greater than arr[j+1] then swap change these/ elements, else move on

Print the sorted array./

**Complexity** :

1. Merge sort- O(nlogn)

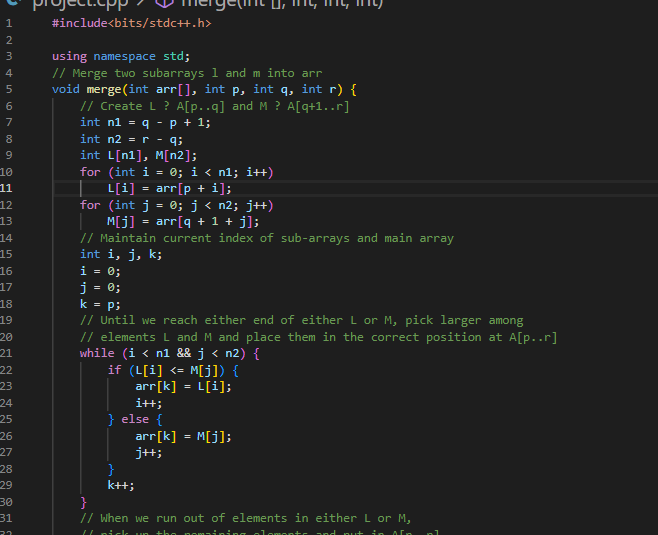
2.Quick sort-O(nlogn)

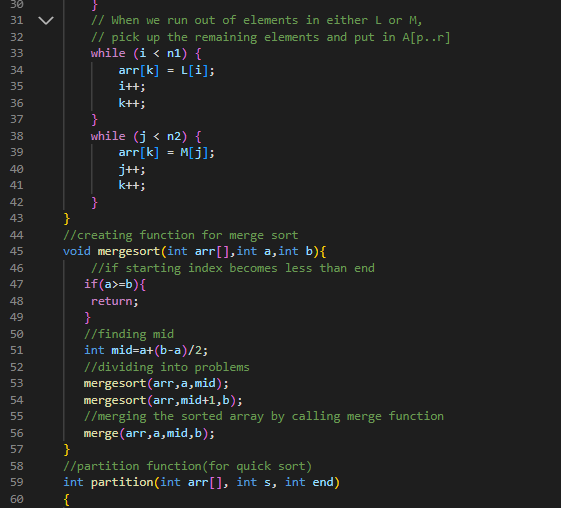
3.Selection sort-O(n\*n)

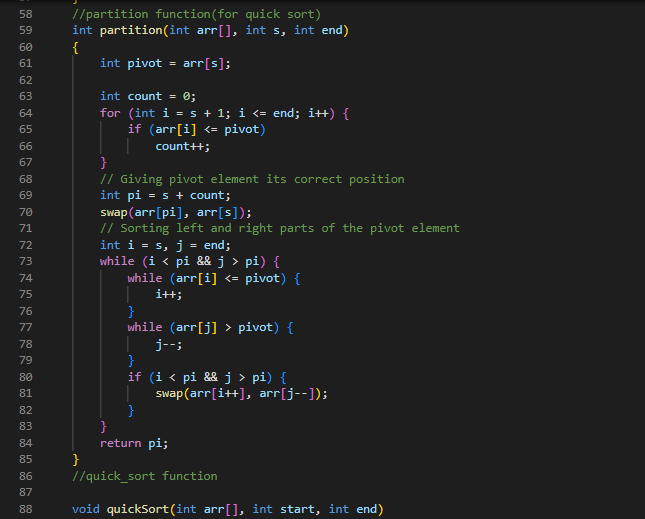
4.Insertion sort-O(n\*n)

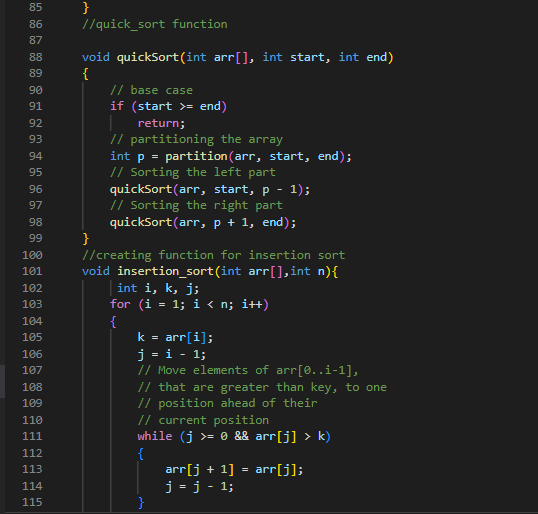
5.Bubble sort-O(n\*n)

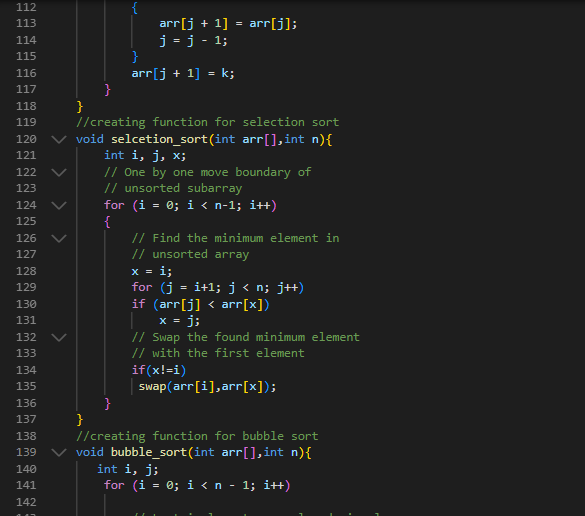
**SOURCE CODE:-**

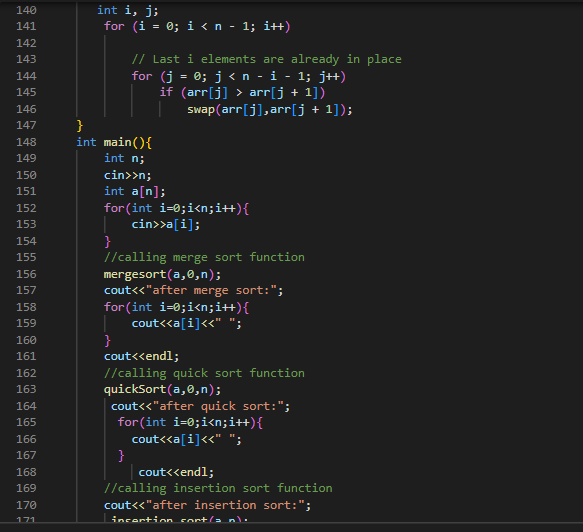


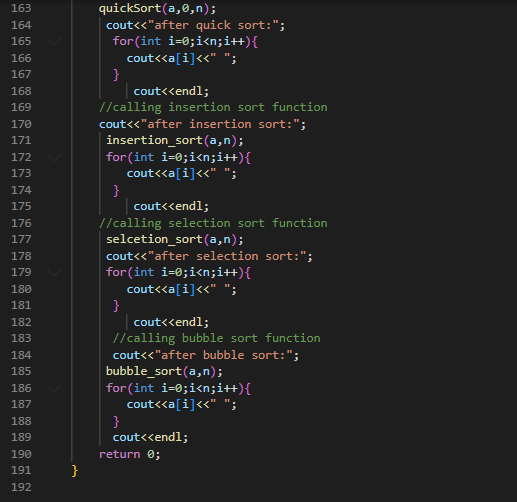




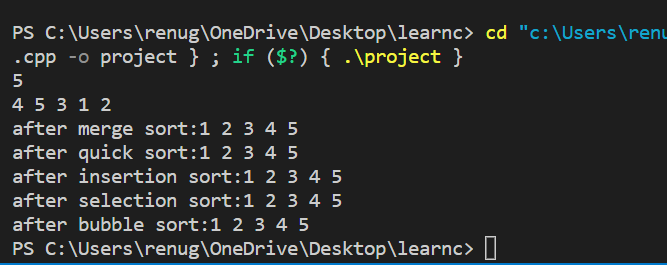








OUTPUT:-



Experiment No.2

Objective:- Perform the following sorting algorithm-Radix sort, Count sort, Bucket sort and Shell sort.

Theory:-Radix sort: The idea of radix sort is to SORT digit by digit in array sort starting from less significant digit to highest significant digit./Radix sort uses counting sort to sort its elements.

Algo:

radixSort(arr)

maxi= largest element in given array.

d = number of digits in largest element of array ( maxi)

Now, create d buckets of the size 0 - 9

for i -> 0 to d  in array

sort given array elements/ using counting sort according to the digits at the i place.

Time complexity: O(a(n+b)) is the best-case time complexity here. If b equals O(n) complexity is O.. (a\*n).

Count Sort: Counting sort works by counting the number of objects having distinct key values In array. Then do some arithmetic calculations to calculate the position of each object.

Algorithm:

countingSort(array, n) // 'n' is the size of array

maxi = find maximum element here

create count array with size of maximum + 1

Initialize count array with all the 0's in array

for i = 0 to n  loop

find the count of every unique element in array and

store that count at ith position in the array .

for j = 1 to maxi loop

Now, find the total sum and store it in count array

for i = n to 1  loop;

Restore the array given.

Decrease the count of every restored elemen/t by 1 in array

end countingSort

complexity:

Time complexity of count sort

O(n+k) where n is the total number of elements in given array.

Bucket sort: Bucket sort is algorithm that separate the elements in array into multiple groups said to be buckets for sorting elements between 0 and 1. Algo:

Bucket Sort(A[])

 Let B[0....n-1] be a new array

 n=length[A]

 for i=0 to n-1

 make B[i] an empty list

 for i=1 to n

 do insert A[i] into list B[n a[i]]

 for i=0 to n-1

 do sort list B[i] with insertion-sort

 Concatenate lists B[0], B[1],........, B[n-1] together in order

End

Time complexity: O(n + k), The best-case time complexity of bucket sort is /O(n + k).

Shell sort:

This algorithm first sorts the elements that are far away in array from each other, then it subsequently reduces the gap between them.

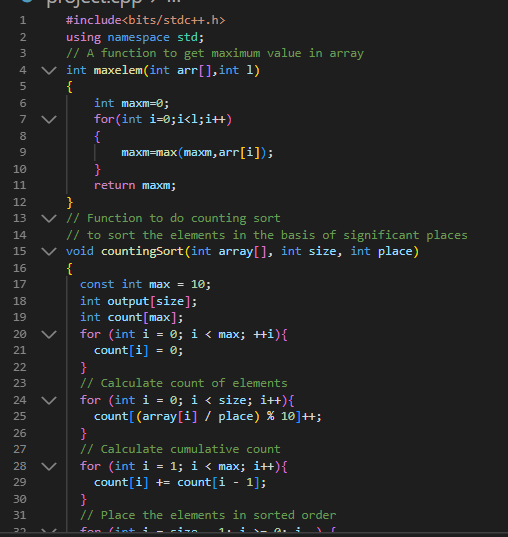
Time complexity:

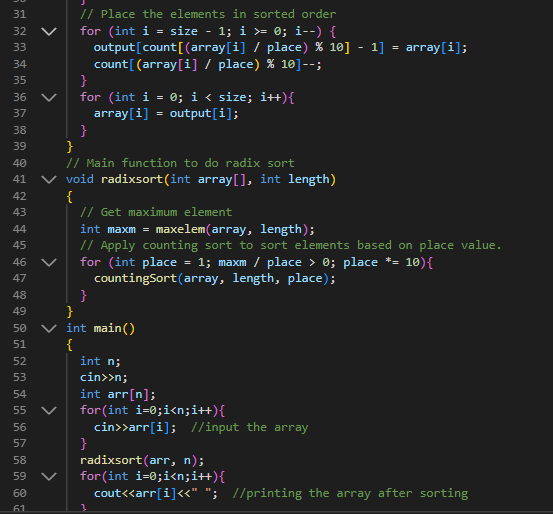
The best case complexity is O(n log(n)).

The worst-case complexity for shell sort is  O(n^2).

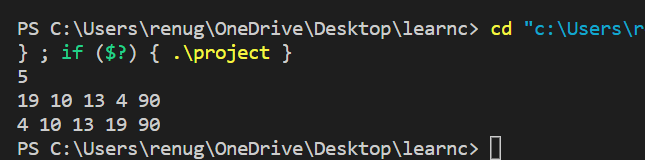
**SOURCE CODE:-**

**RADIX SORT**

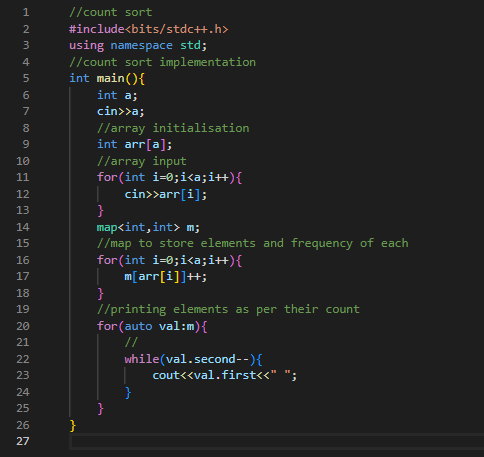




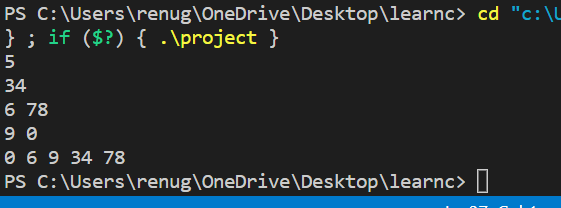
OUTPUT:-



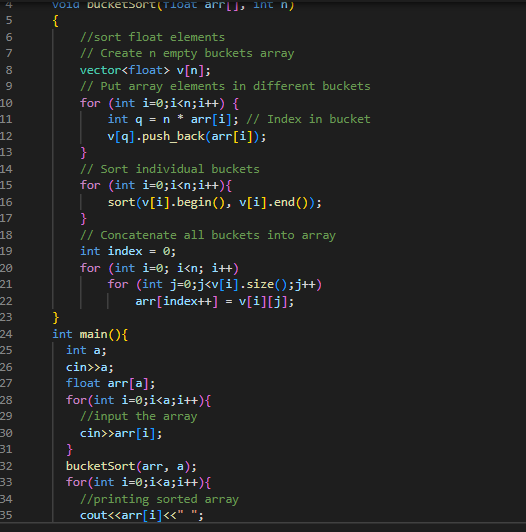
COUNT SORT:



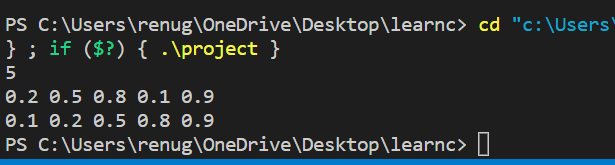
Output:



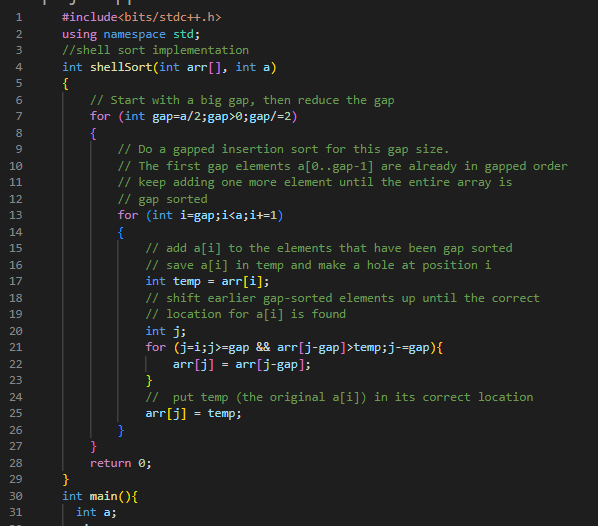
BUCKET SORT:

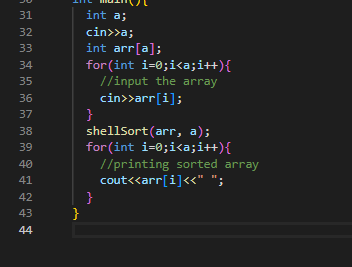


Output:

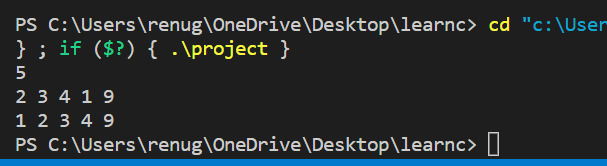


SHELL SORT:





OUTPUT:



Experiment No. 3

Objective:- Perform Linear and Binary search operations.

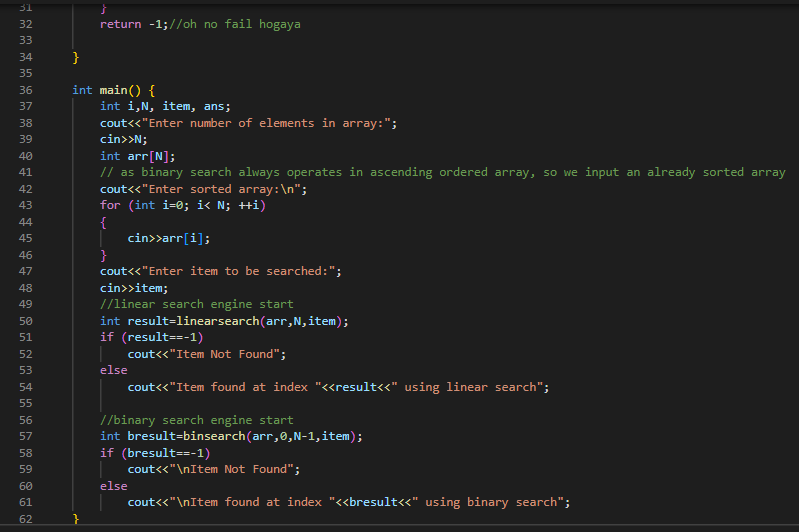
Theory:-

Linear Search is a search algorithm in which we search in the given array one by one to check the required element.

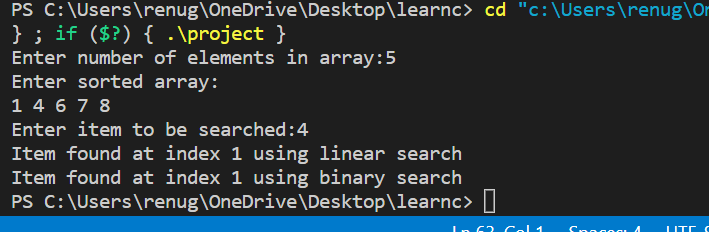
Binary Search is a search algorithm in which we continuously divide the array in parts to find the required element.

**Code:-**





OUTPUT:

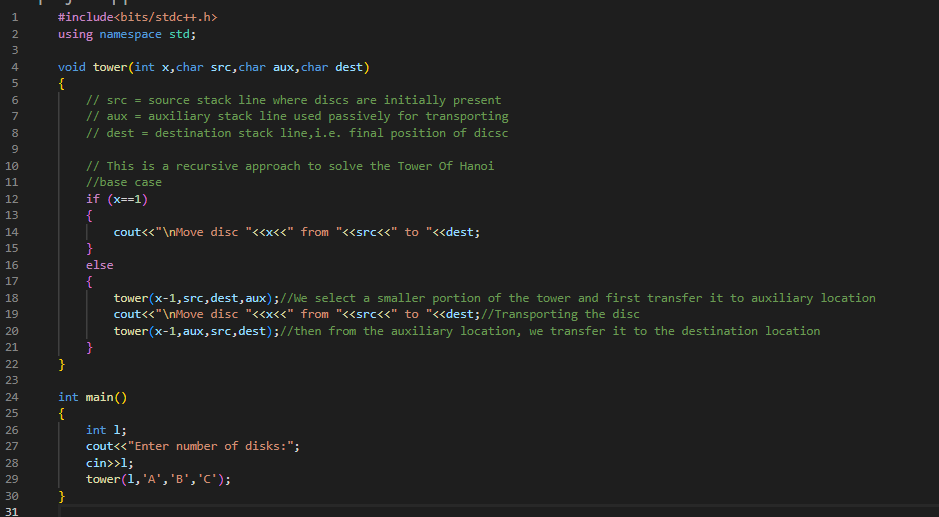


Experiment No. 4

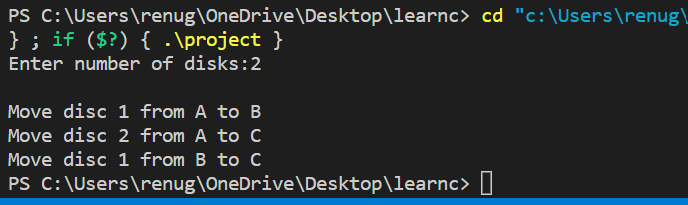
Objective:- Solving standard Tower of Hanoi.

Theory:- It is a beginner level problem based on Recursion in c++. We transfer all the discs from source to destination in this code by first doing the transfer operation for just one disc and then generalizing the method for all discs to complete the process.

**Code:-**



OUTPUT:



**Experiment - 5**

**Objective:**Write a program for inserting elements in:

* + - 1. Binary Search tree
      2. AVL tree
      3. Red-Black tree

**THEORY/DESCRIPTION:**

**BINARY SEARCH TREE :—** A binary Search Tree is a binary tree data structure having different properties:

* + 1. The left subtree of a node contains only nodes in tree n/.
    2. The right subtree of a node contains only nodes in tree with keys greater than the key.

**AVL TREE :—**AVL tree is a self-balancing Binary Search Tree (BST) where elements are inserted one by one and appropriate rotations take place.

There are 4 types of rotations.

* + - 1. LL
      2. RR
      3. LR
      4. RL

**RED- BLACK TREE :—** Red-Black tree is a self-balancing binary search tree in which each node has a color either red or black having different properties.

1.Root Property: The root node is always black.

2.Red node Property: If a red node has children then, the children are always black in color.

3.Depth Property: The depth should always be balanced.

**COMPLEXITY:**

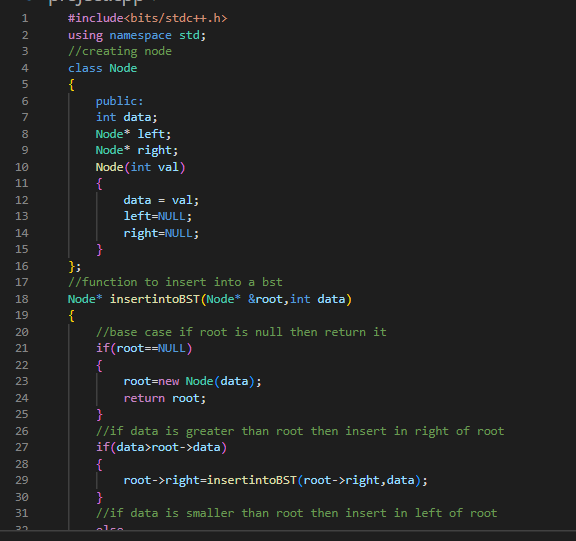
Binary Search Tree: O (H) H=height of the tree

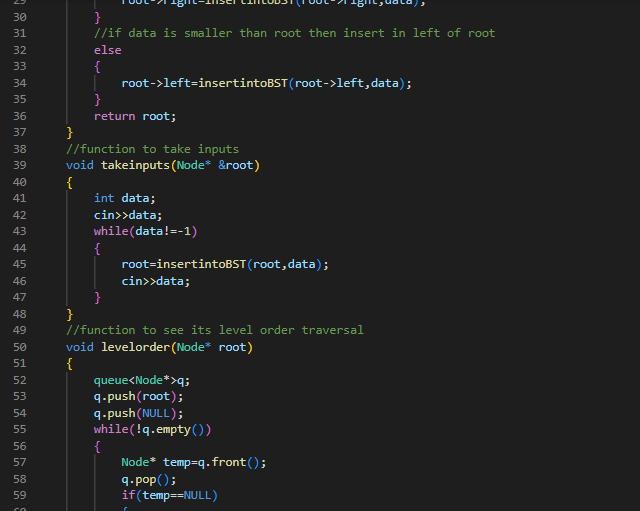
AVL tree: O( log n)

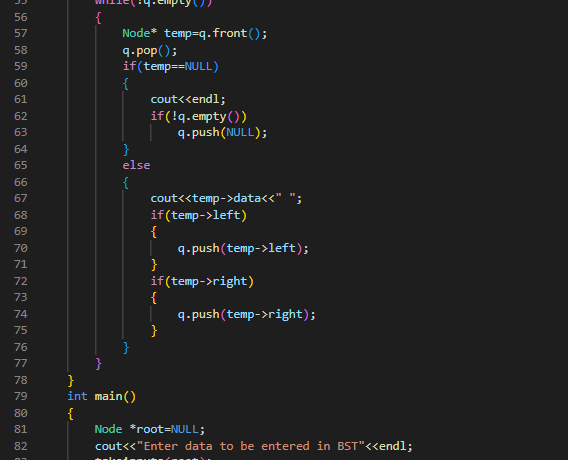
Red-black Tree. : O( log n)

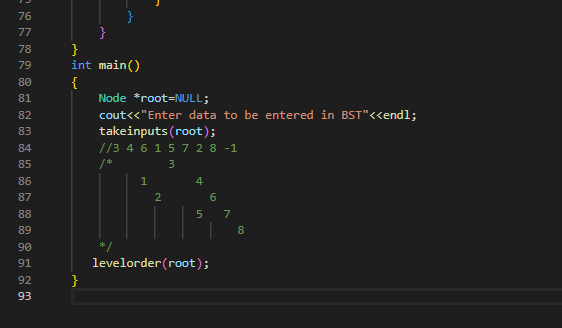
**SOURCE CODE:--**

**Binary Search Tree:--**

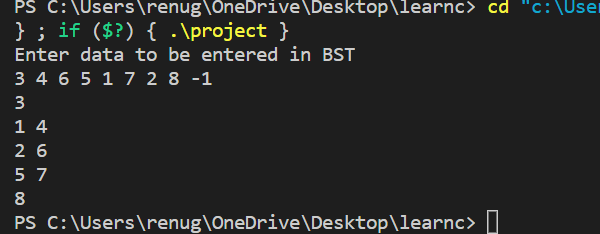






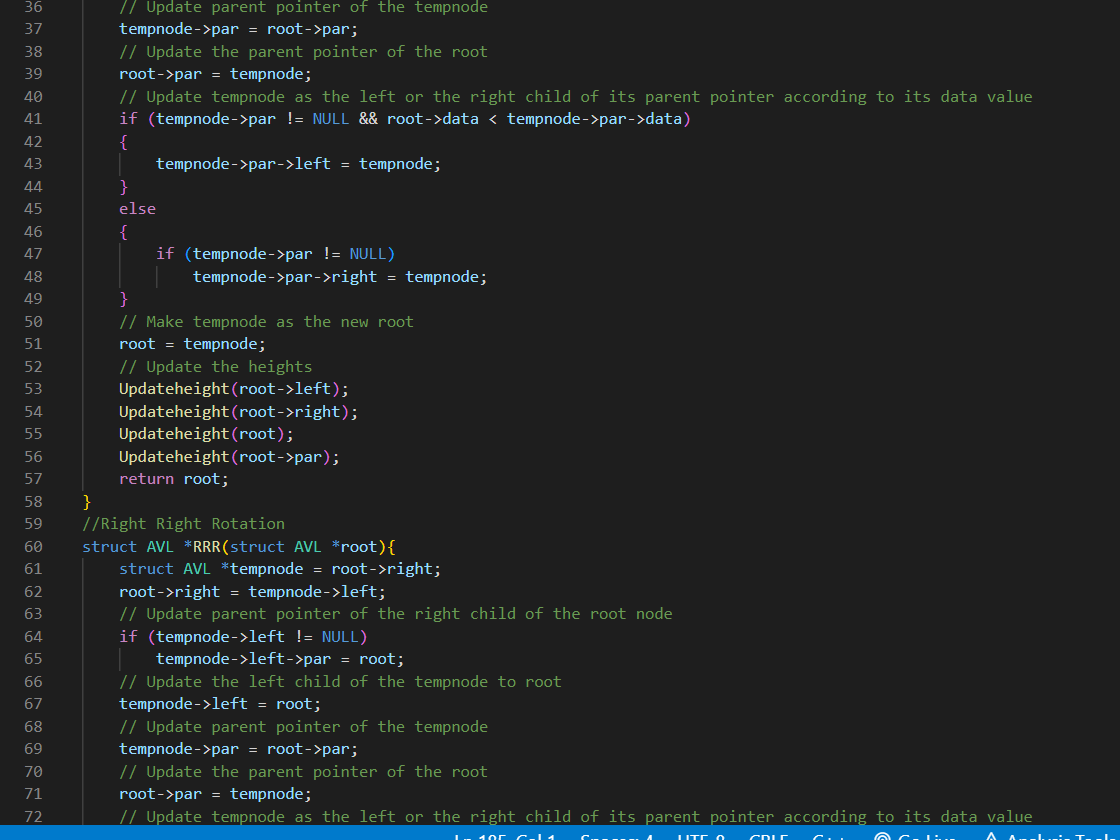


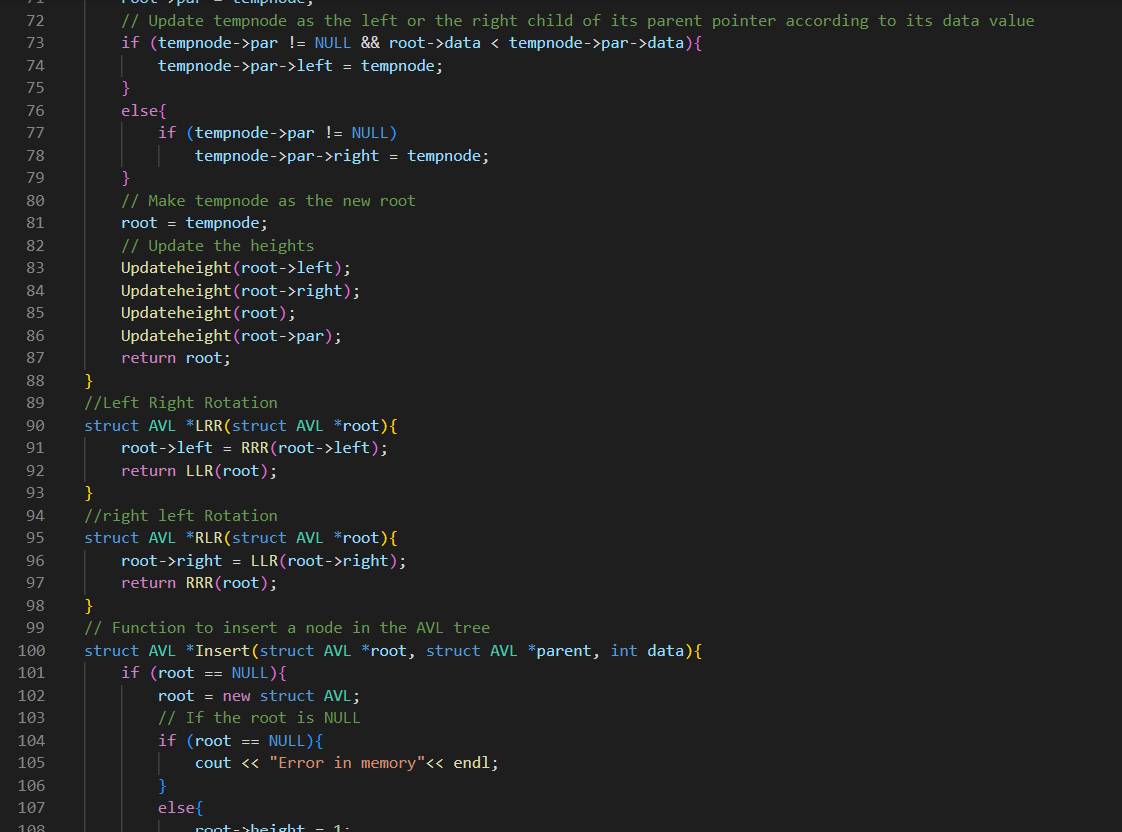
OUTPUT:



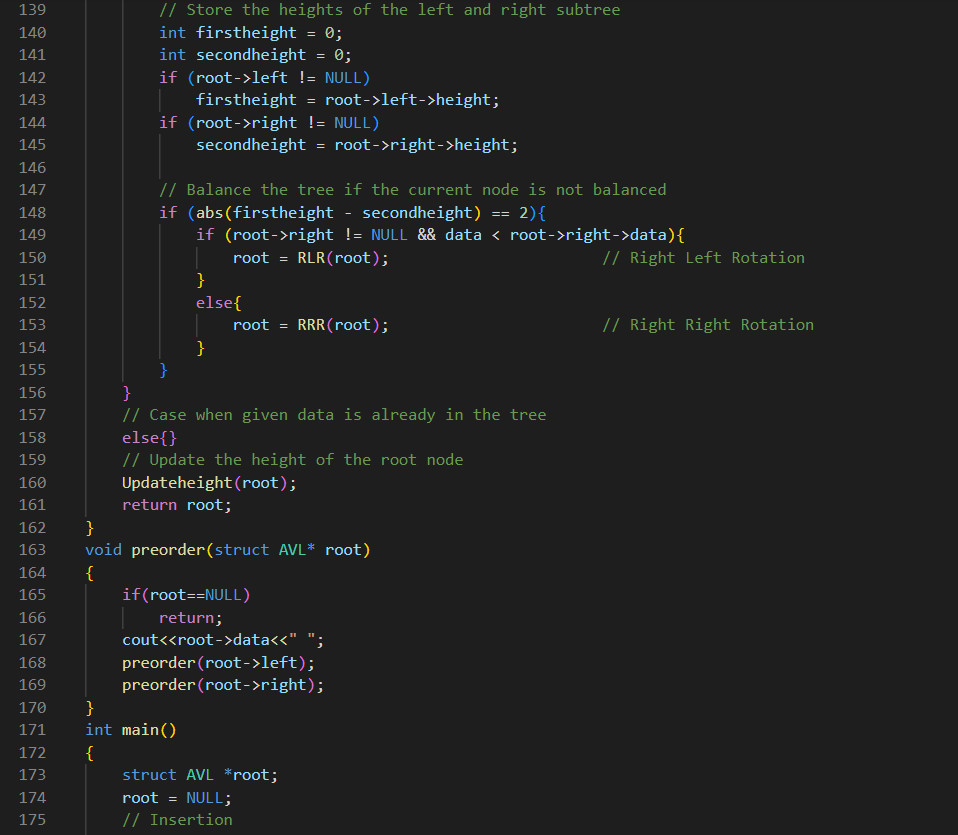
AVL TREE:

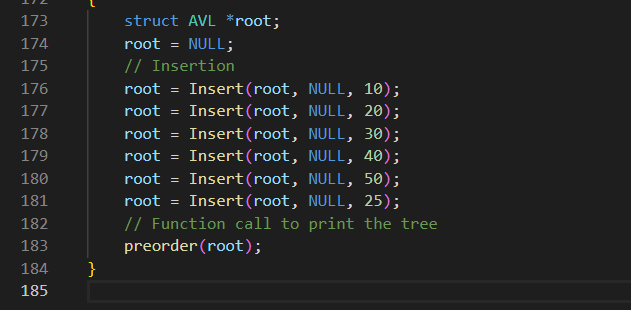




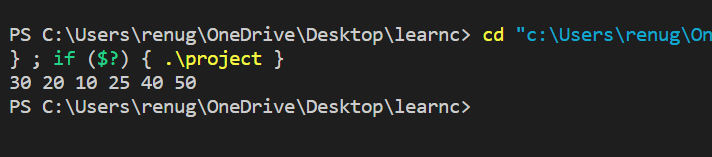








**OUTPUT:**

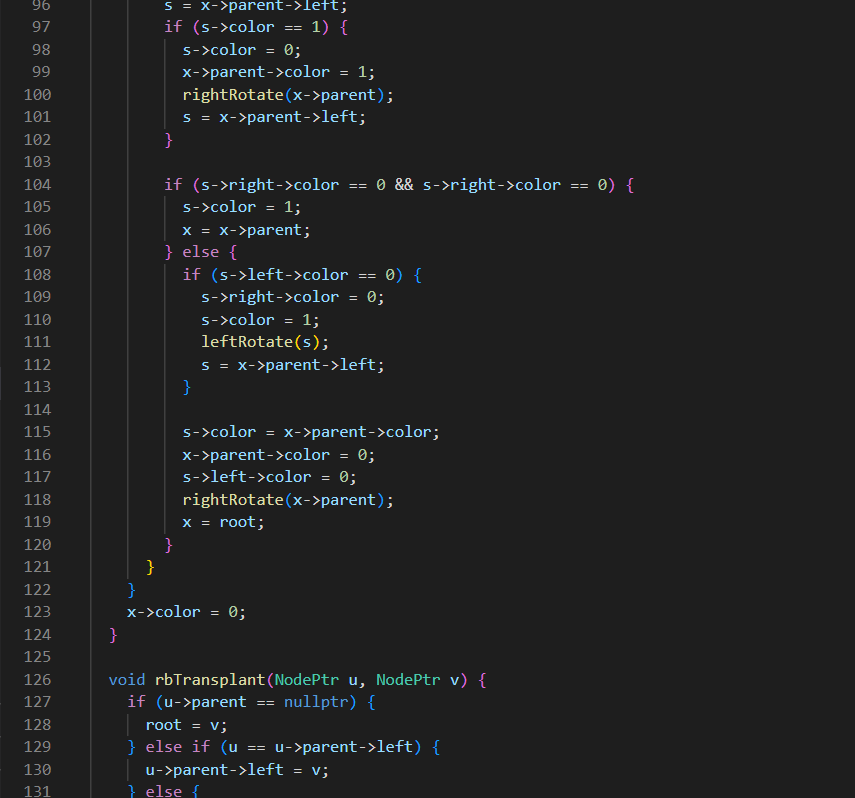


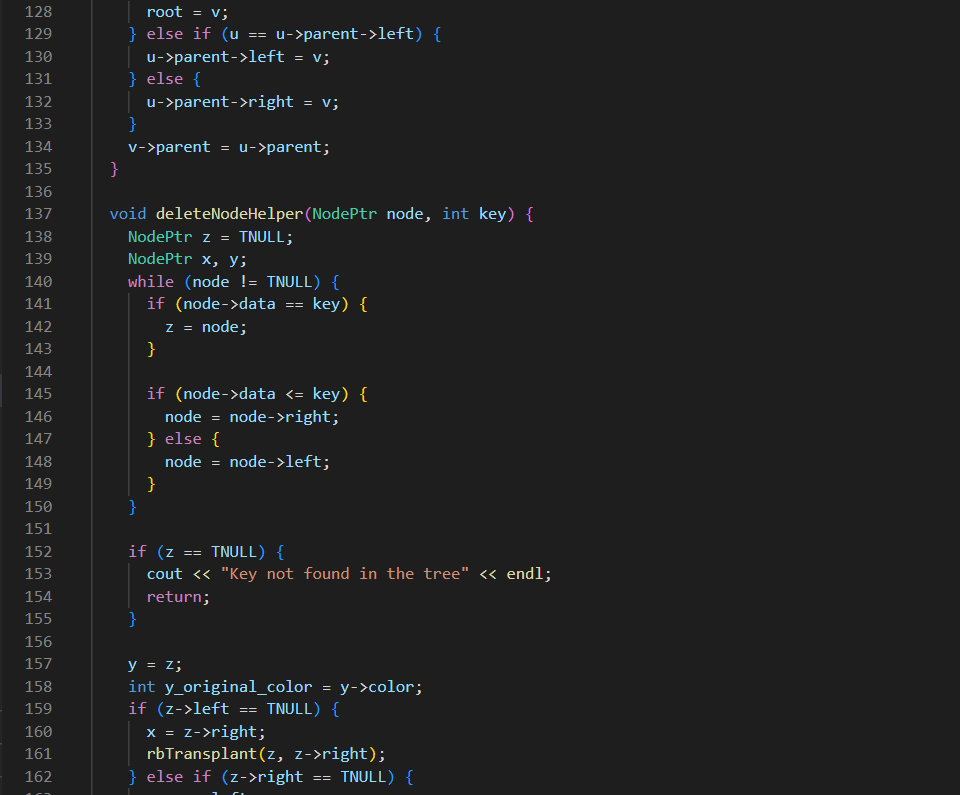
**RED BLACK TREE:**

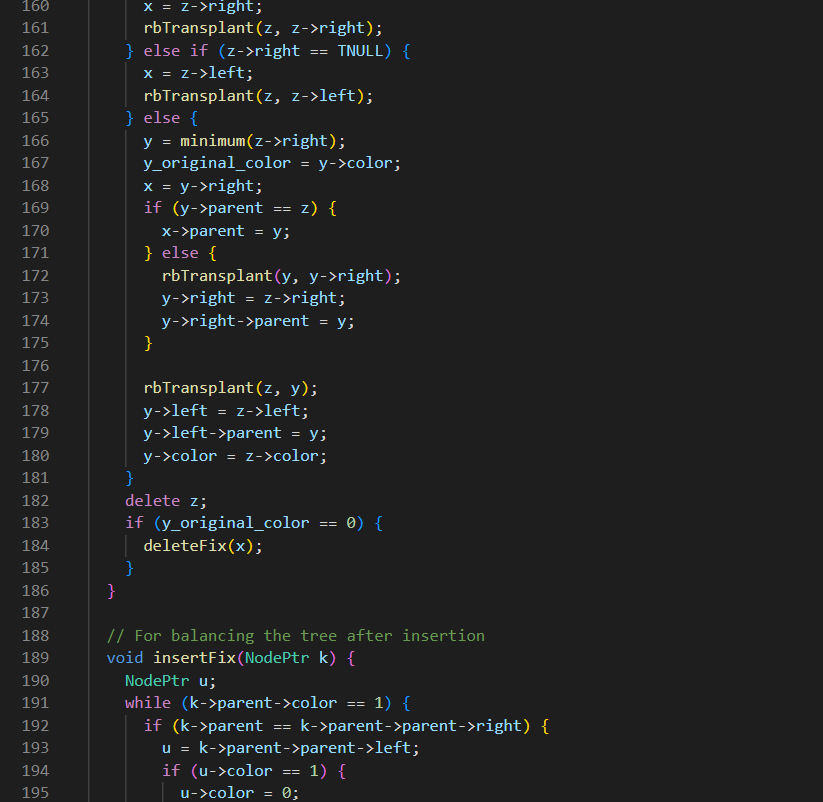








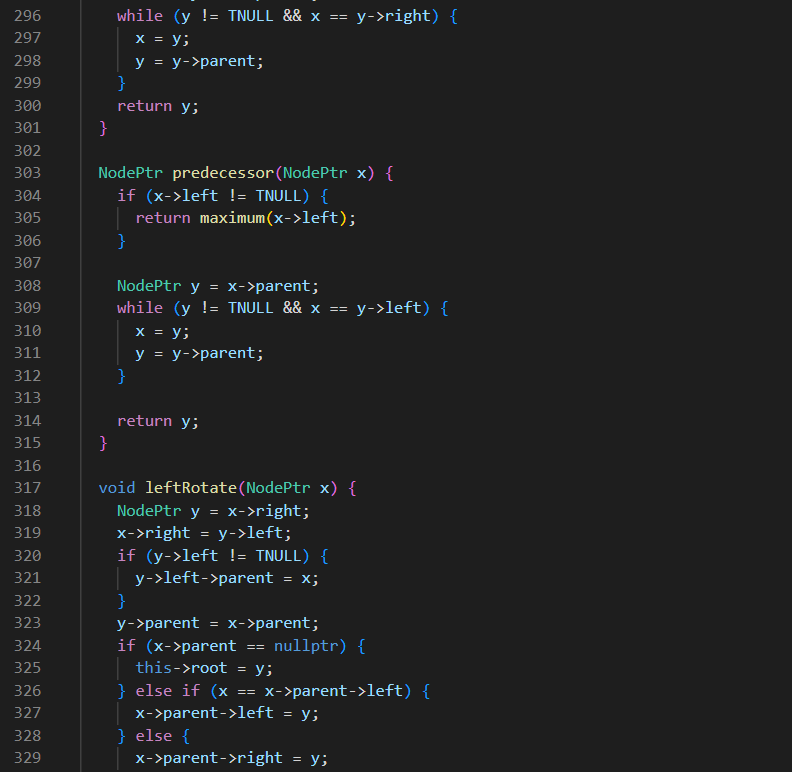




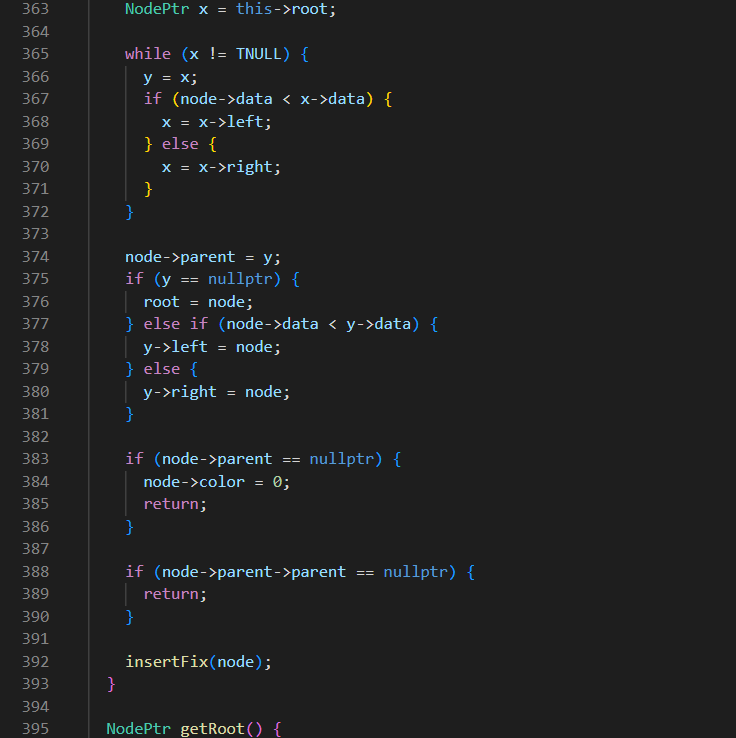


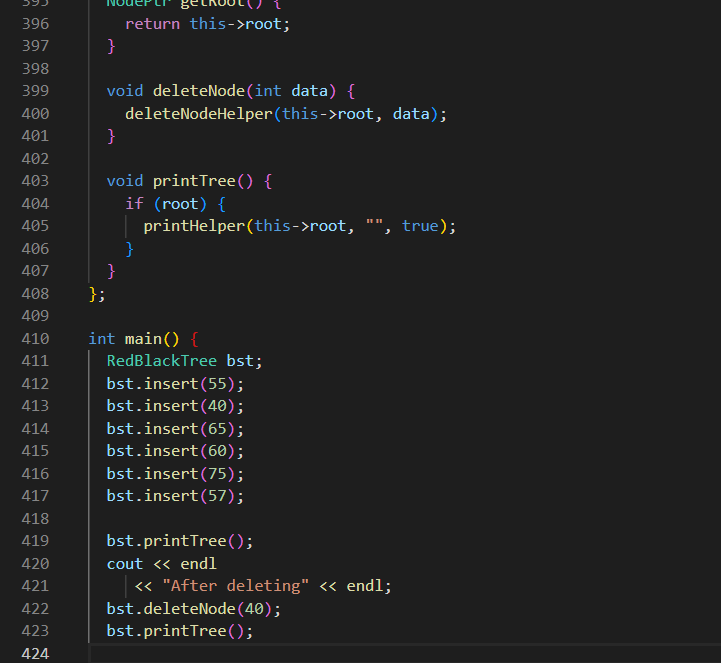




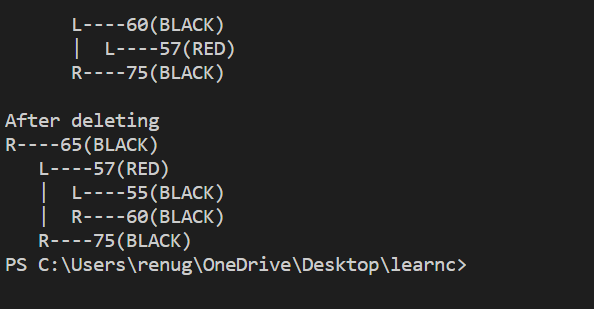








OUTPUT:



Experiment No. 6

Objective: Write a program for deleting elements in:

* + - 1. Binary Search tree
      2. AVL tree
      3. Red-Black tree

THEORY/DESCRIPTION:

**BINARY SEARCH TREE :—** A binary Search Tree is a binary tree data structure having different properties:

* + 1. The left subtree of a node contains only nodes in tree n/.
    2. The right subtree of a node contains only nodes in tree with keys greater than the key.

**AVL TREE :—**AVL tree is a self-balancing Binary Search Tree (BST) where elements are inserted one by one and appropriate rotations take place.

There are 4 types of rotations.

* + - 1. LL
      2. RR
      3. LR
      4. RL

**RED- BLACK TREE :—** Red-Black tree is a self-balancing binary search tree in which each node has a color either red or black having different properties.

1.Root Property: The root node is always black.

2.Red node Property: If a red node has children then, the children are always black in color.

3.Depth Property: The depth should always be balanced.

**COMPLEXITY:**

Binary Search Tree: O (H) H=height of the tree

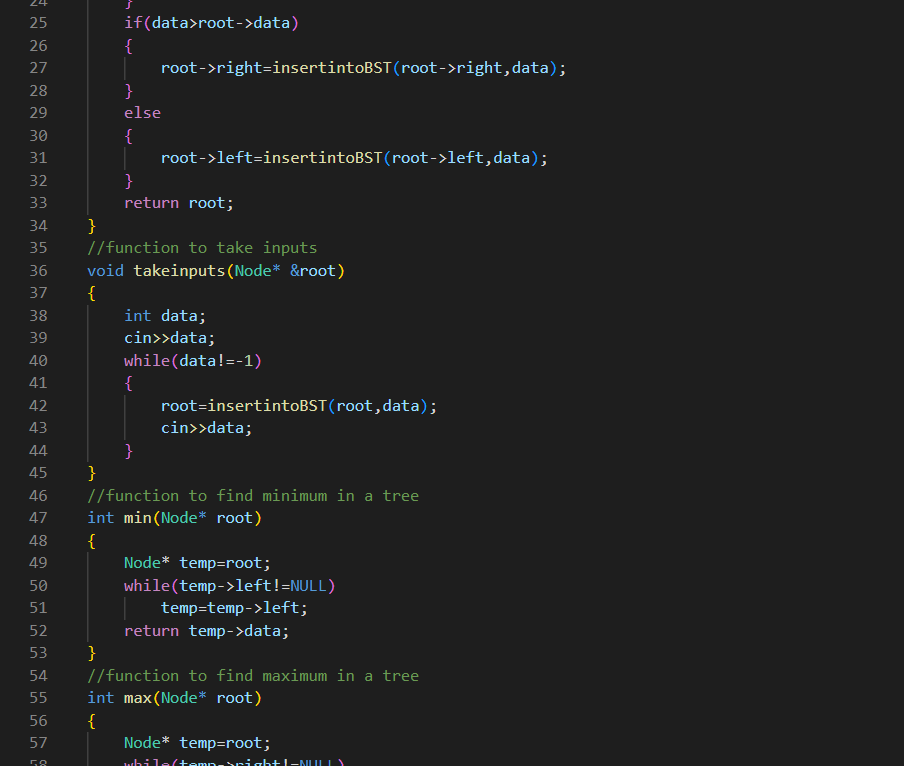
AVL tree: O( log n)

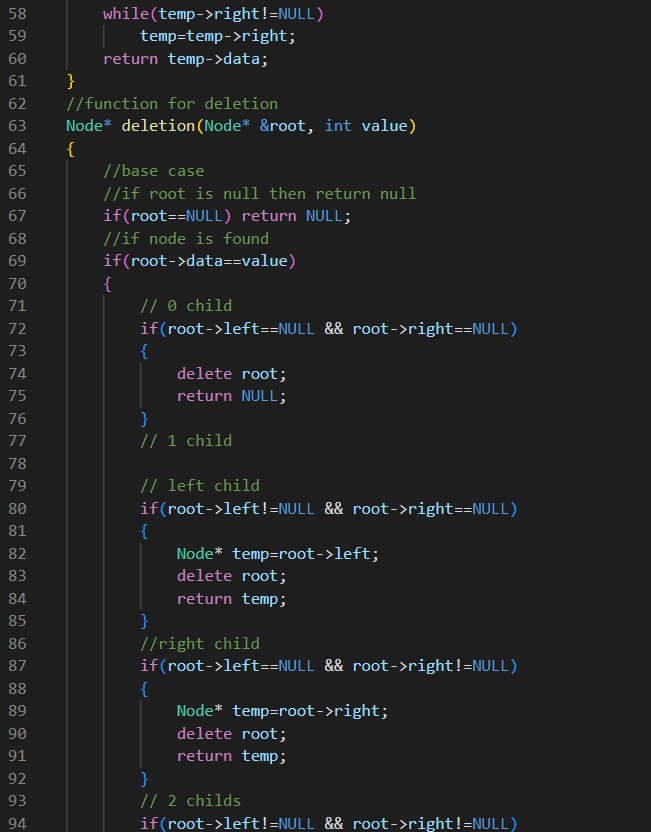
Red-black Tree. : O( log n)

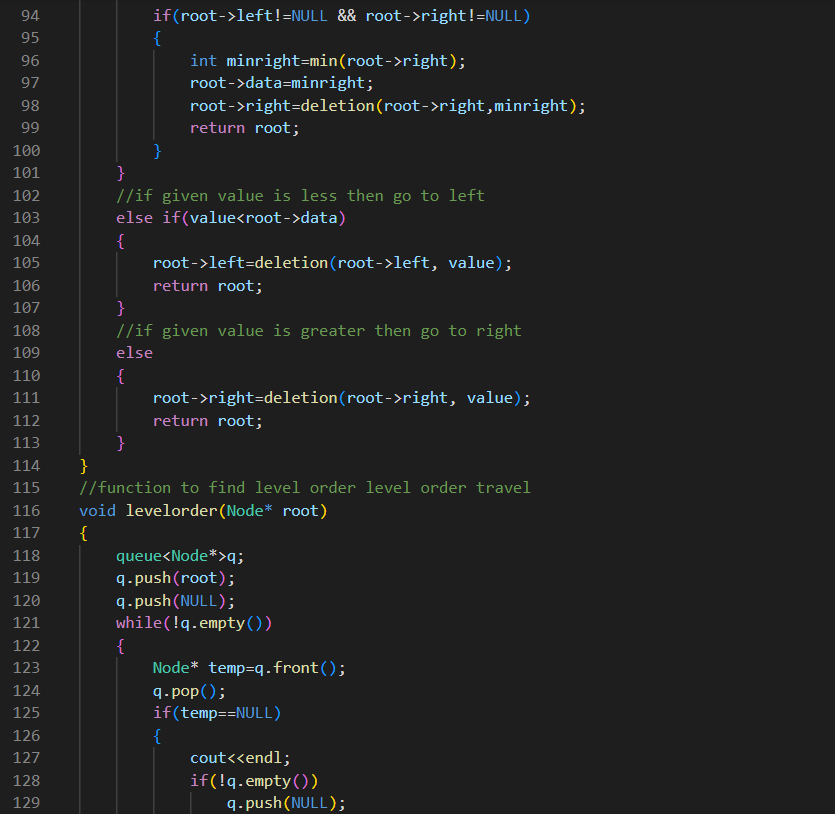
**SOURCE CODE:-**

**Binary search tree:**



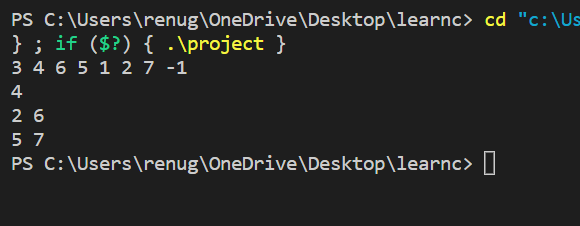




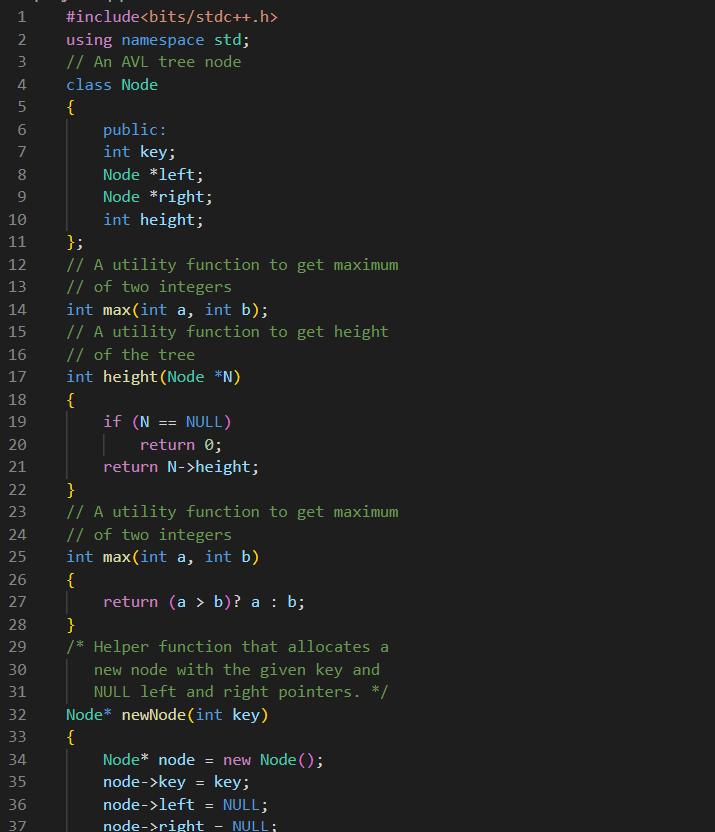


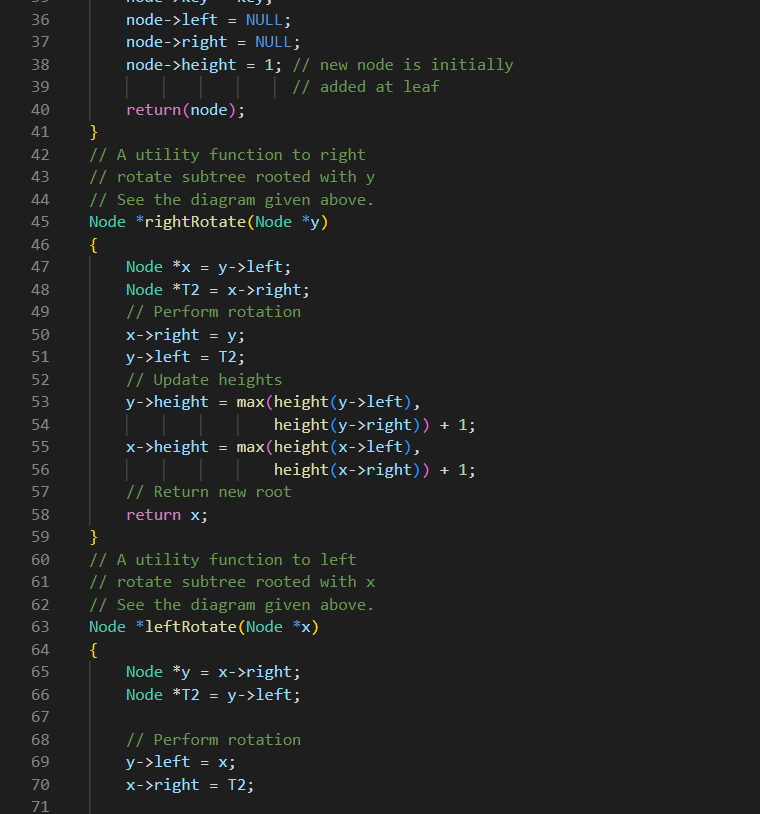


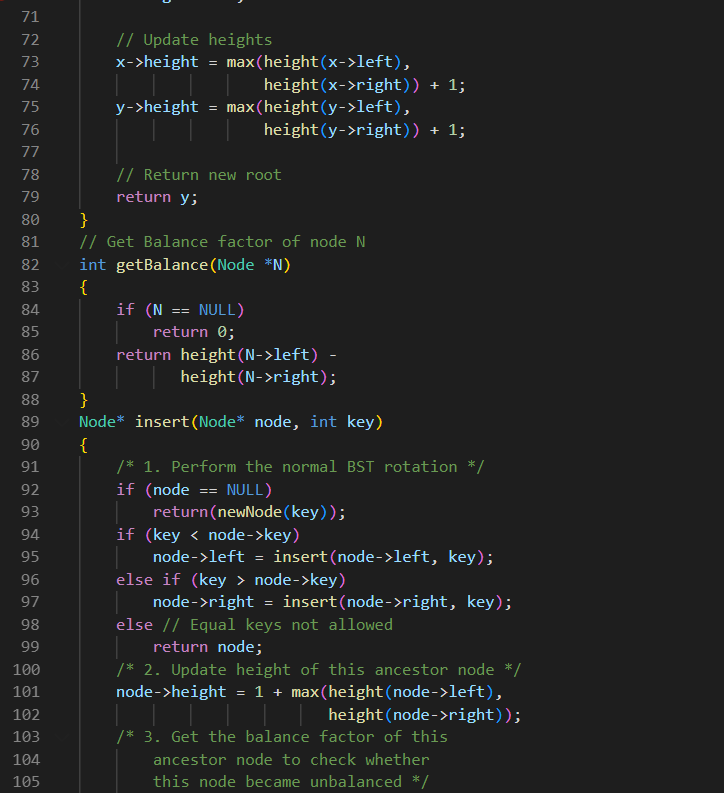
OUTPUT:

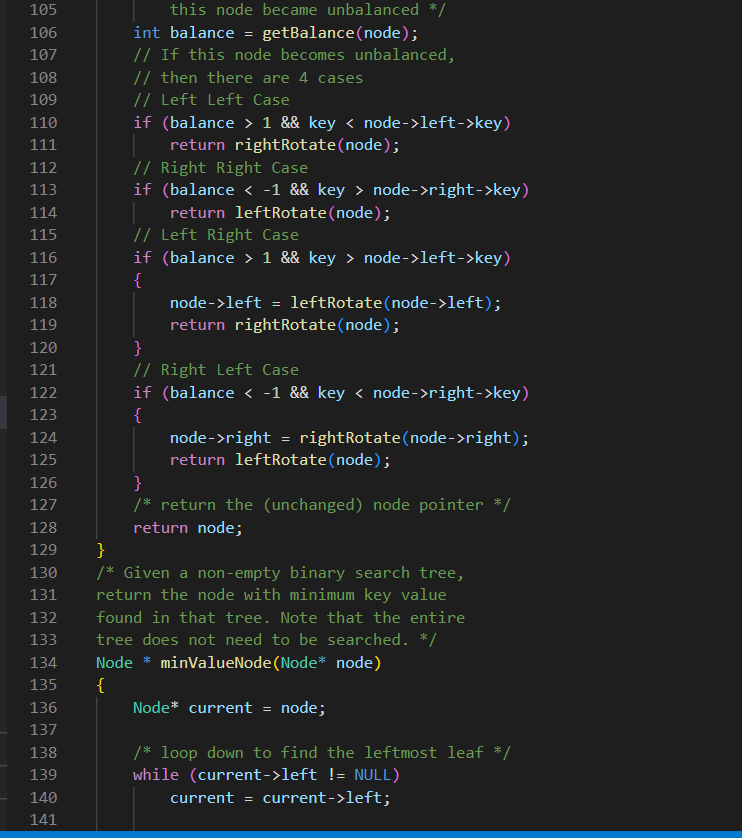


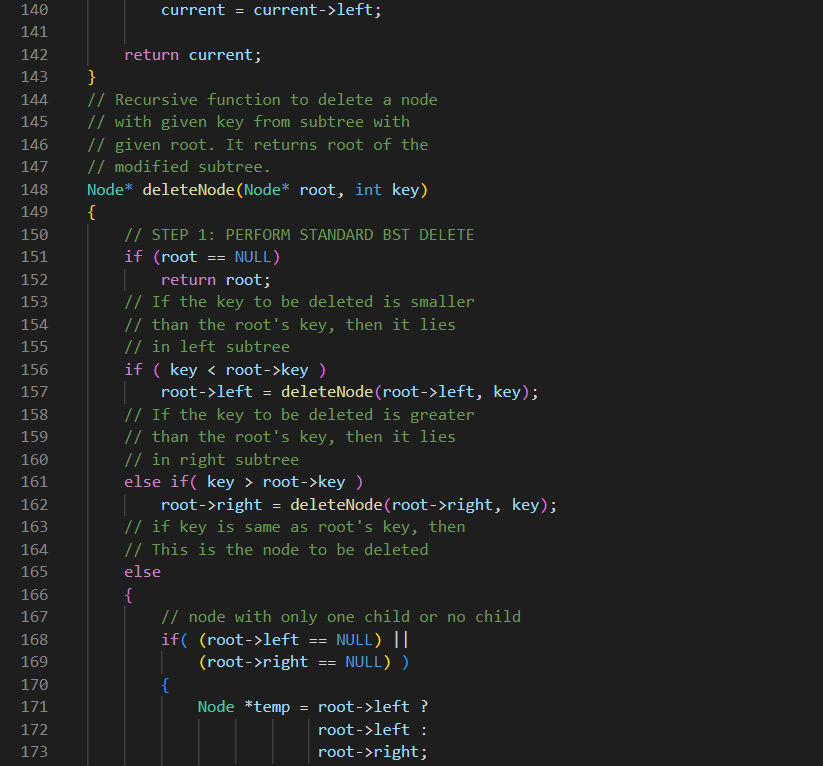
**AVL TREE:**

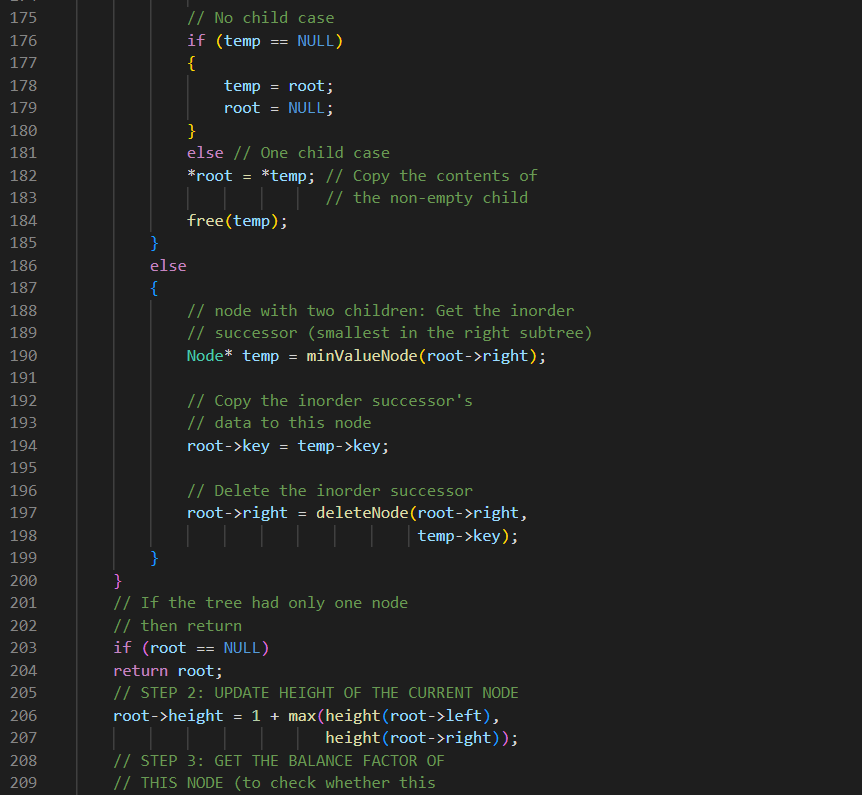


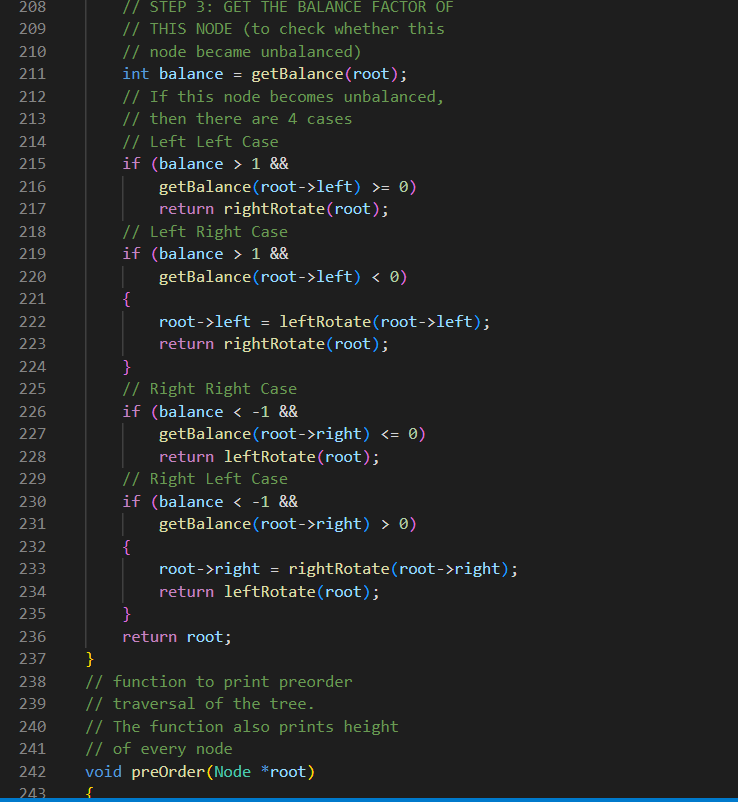




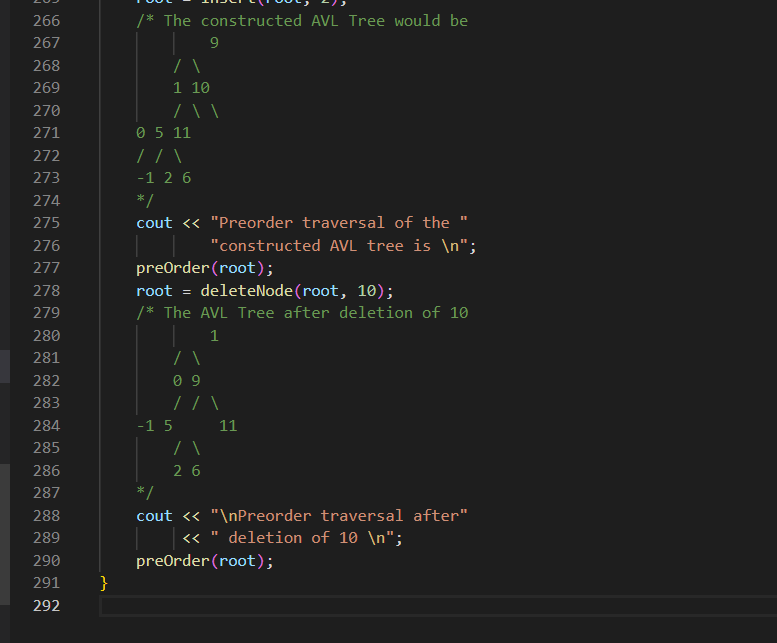




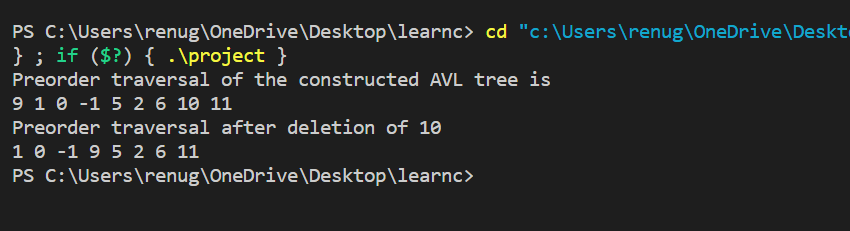








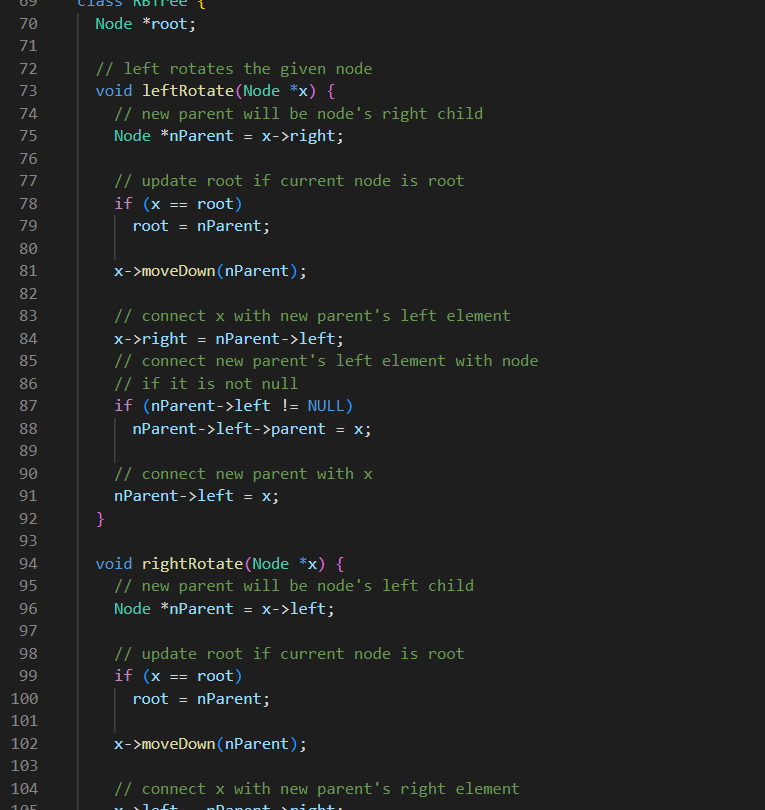
**OUTPUT:**

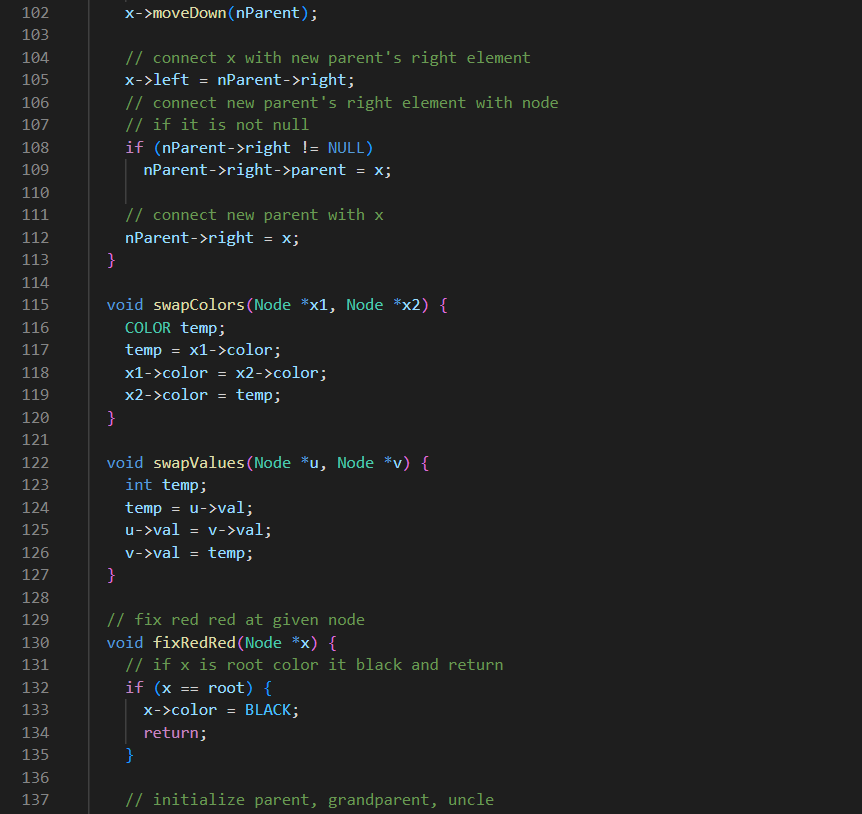


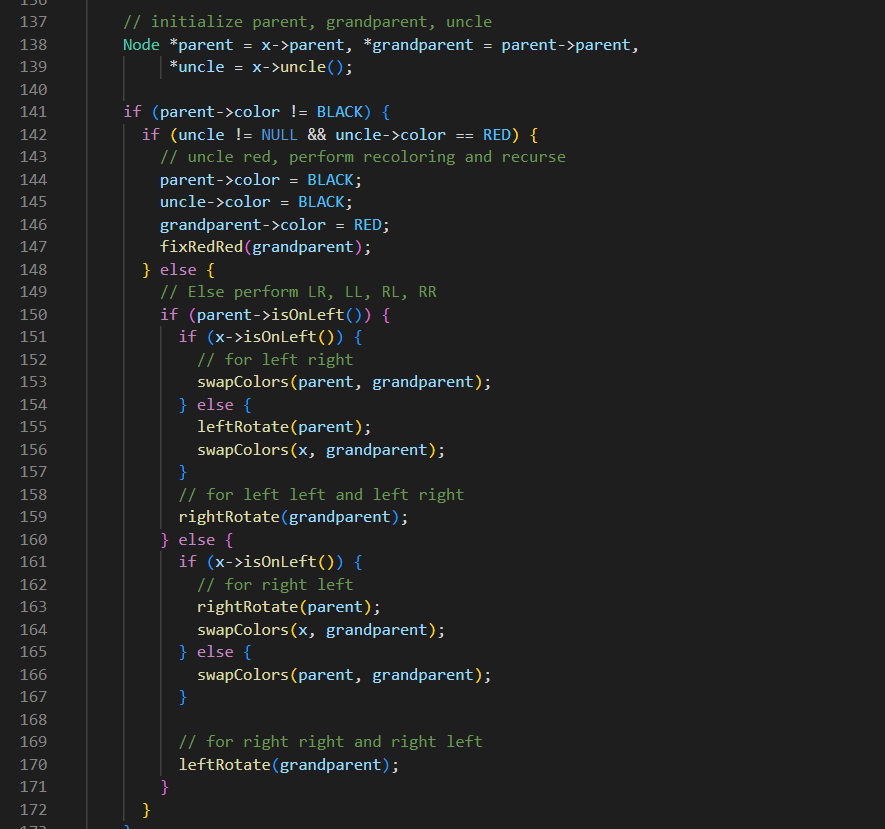
RED BLACK TREE:



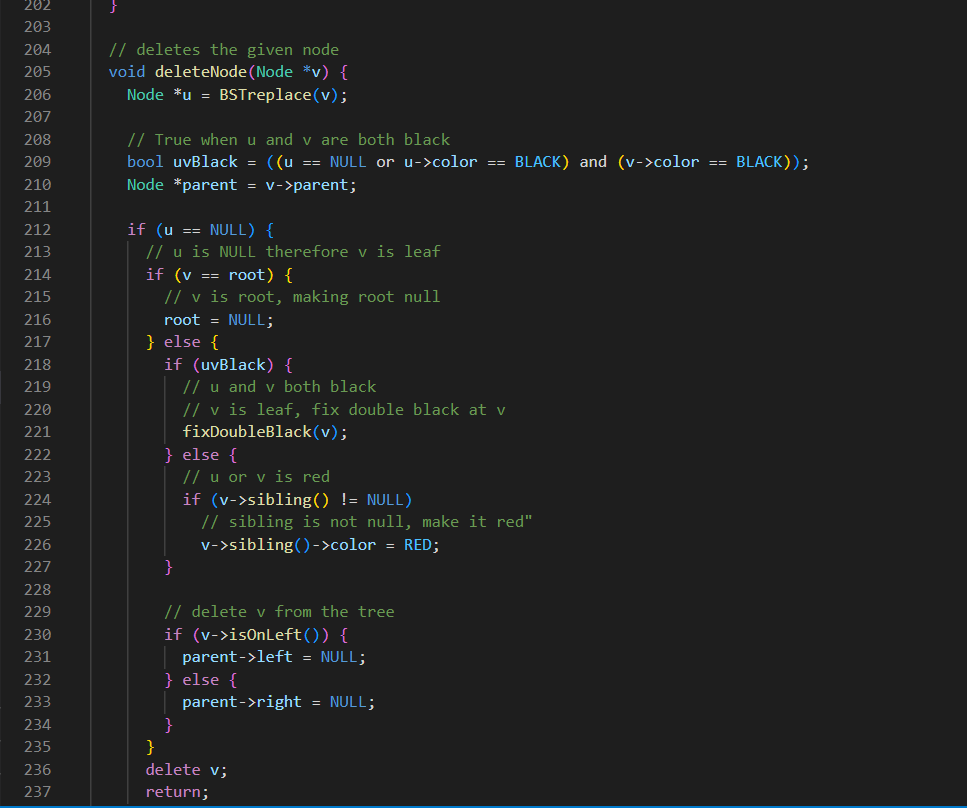


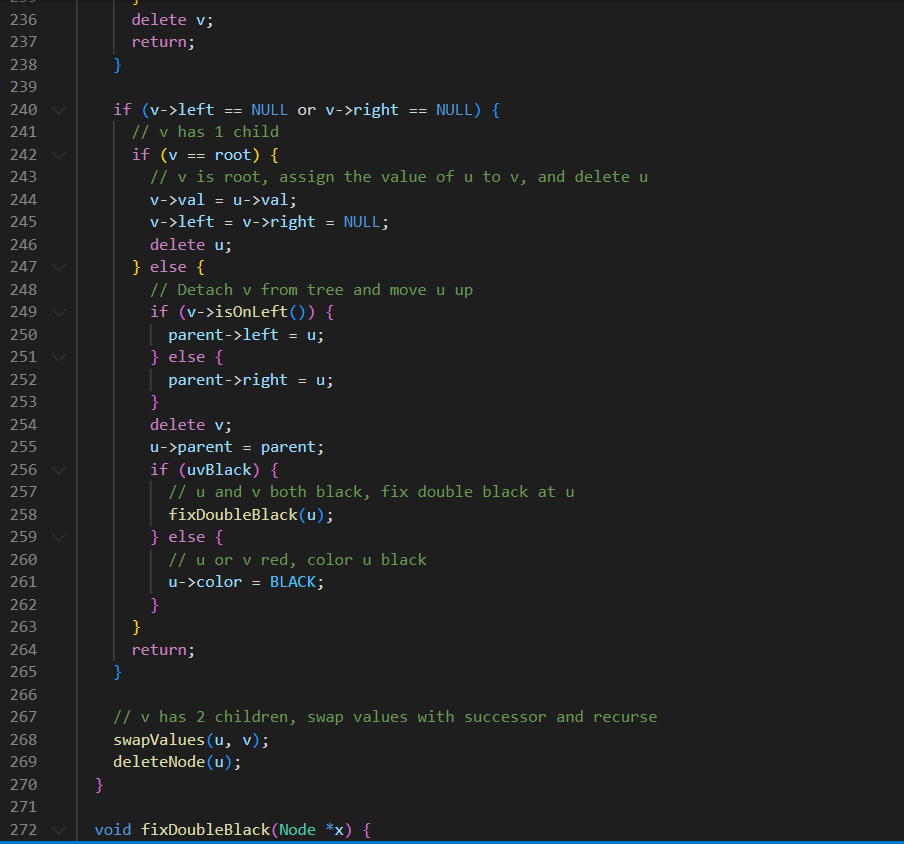


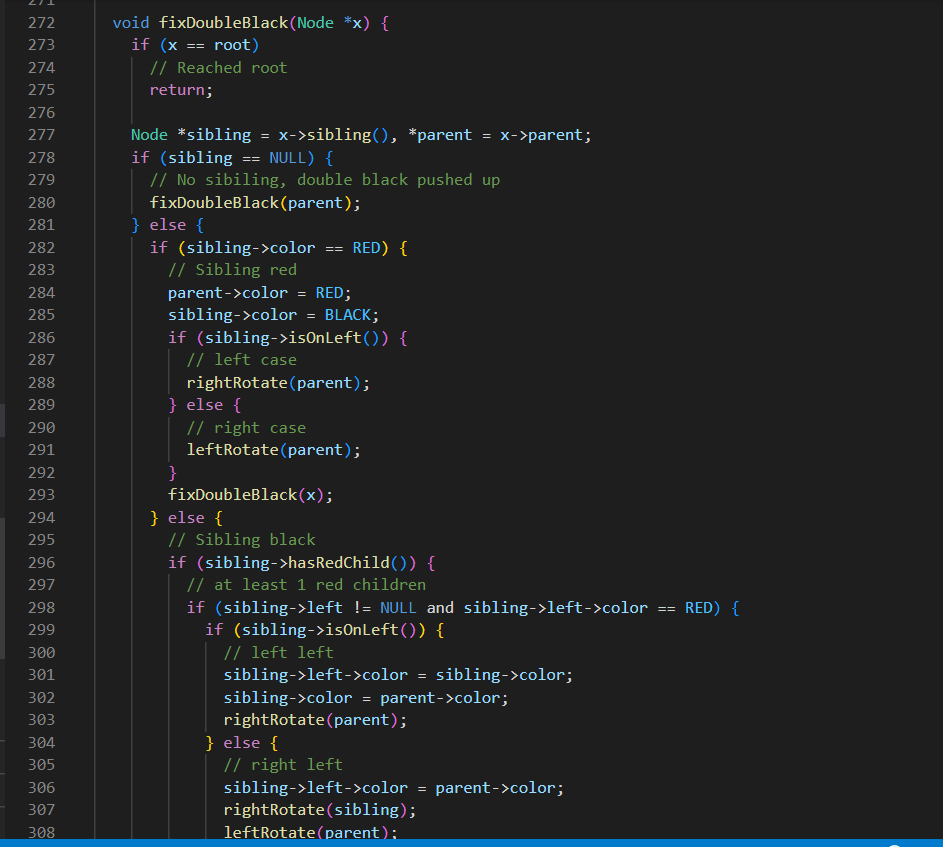


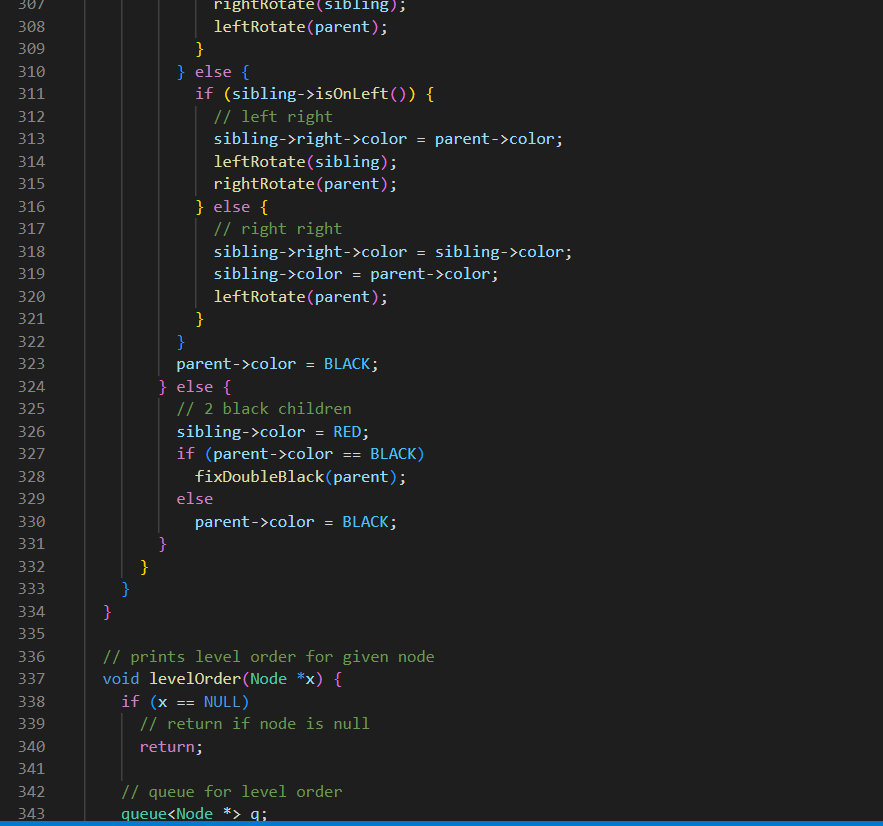




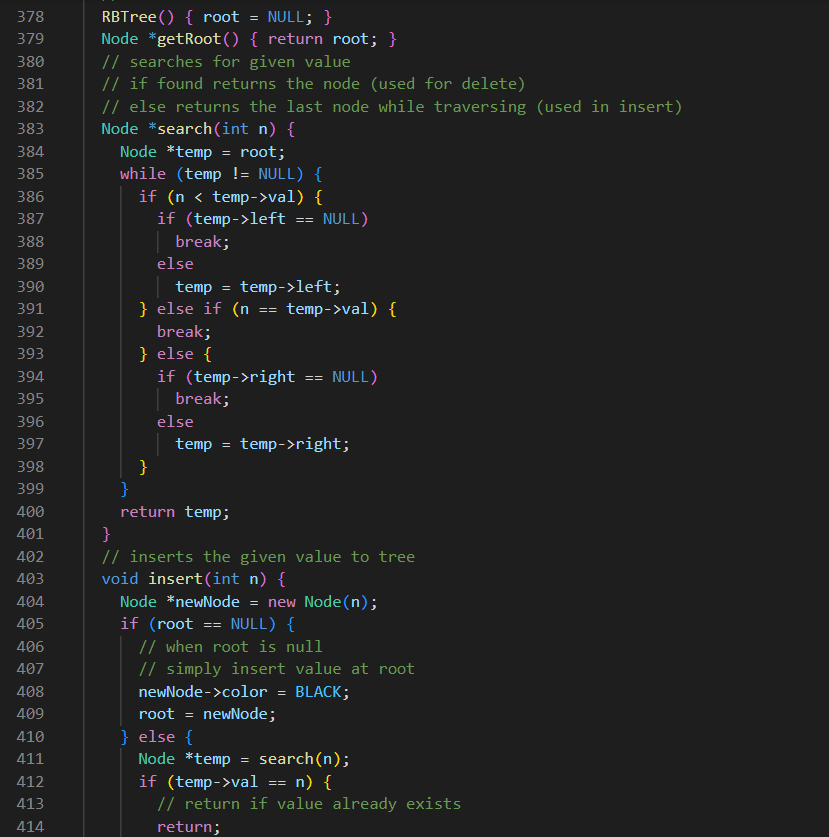






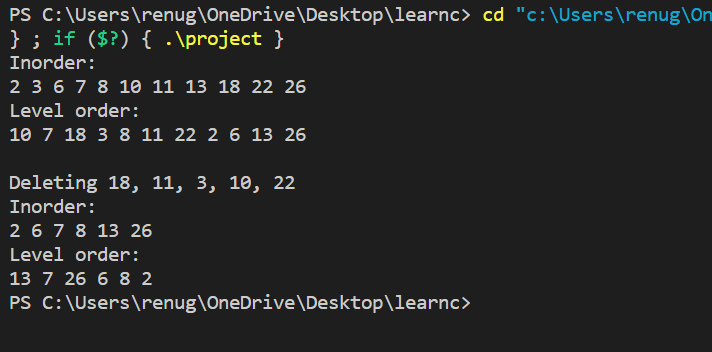








OUTPUT:



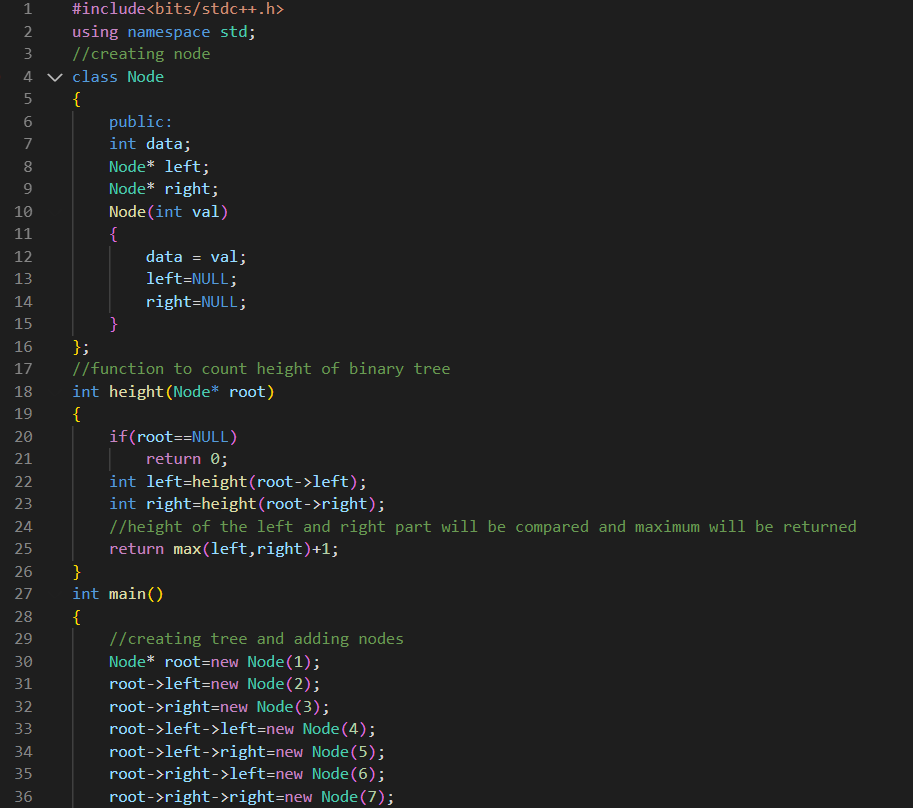
Experiment No. 7

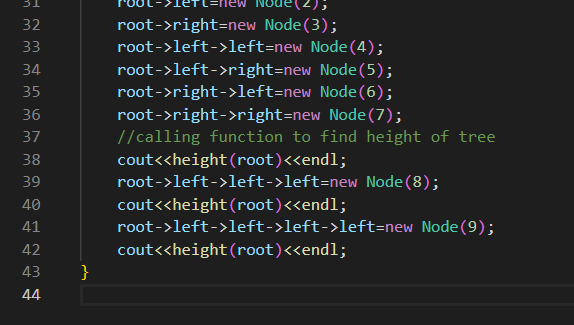
Objective:- Given the root of a binary tree, return the maximum height of the tree.

Theory:- The height of tree is calculated by calculating maximum of height of left subtree and the right subtree using a recursive and then adding 1 to it for the root element.

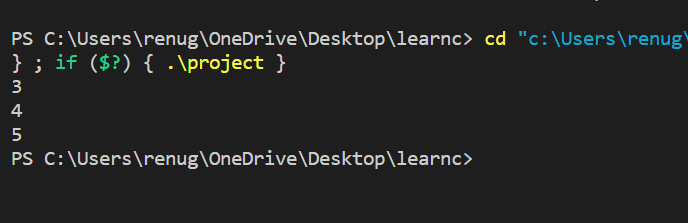
Height=max(Height of left subtree, Height of right subtree) + 1

Code:-





**OUTPUT:**



Experiment No. 8

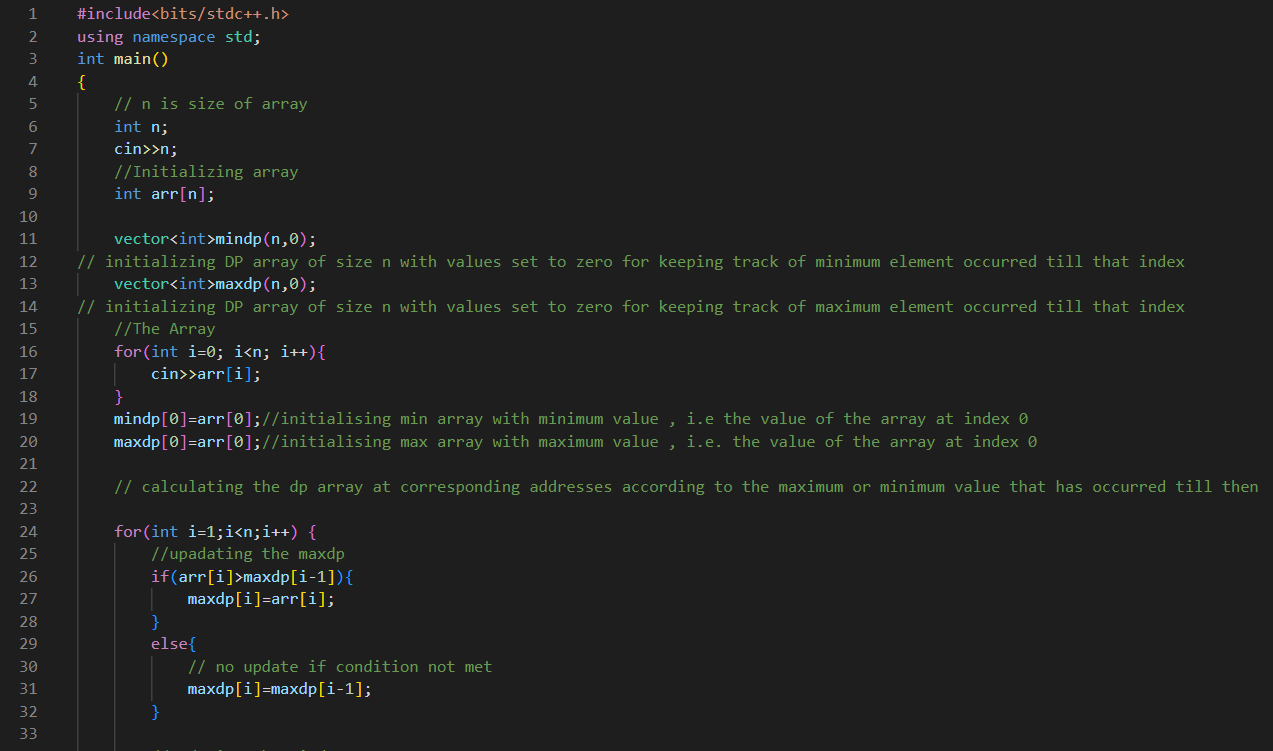
Objective:- Find maximum and minimum of array using the dynamic programming.

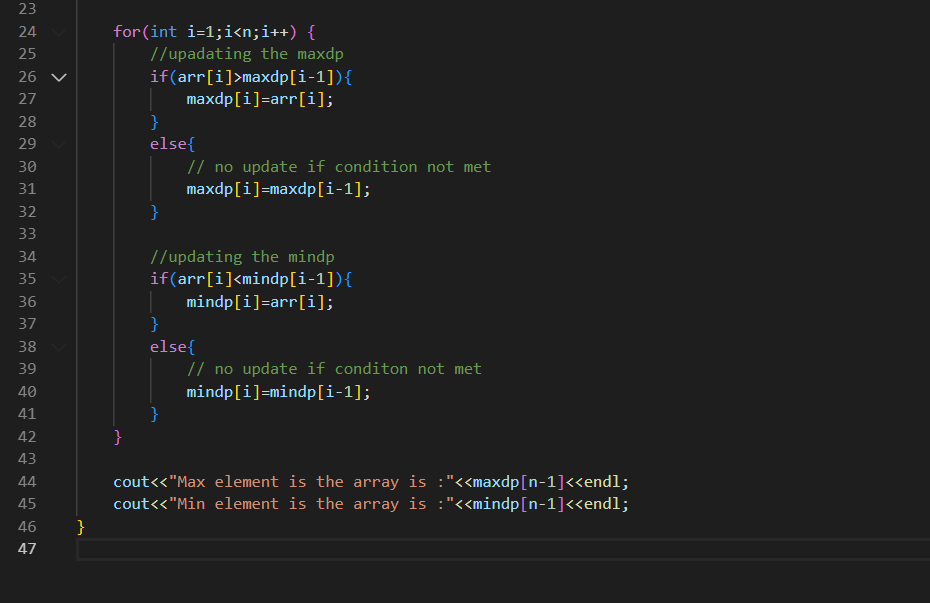
Theory:-

DP optimizes the solution over recursion. The principle is that it stores the result of subproblems so that it need not be calculated again and time complexity decreases.

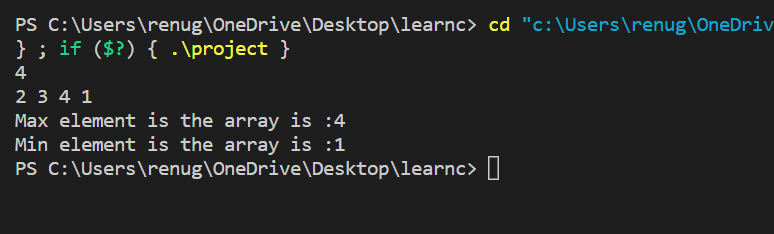
1. Assuming a 2d dp array.
2. Here dp[i][0] is max value of array till ith position if last operation is that of addition.
3. Here dp[i][1] is max value of array till ith position if last position is multiplication.

Code:-





**OUTPUT:**



Experiment No. 9

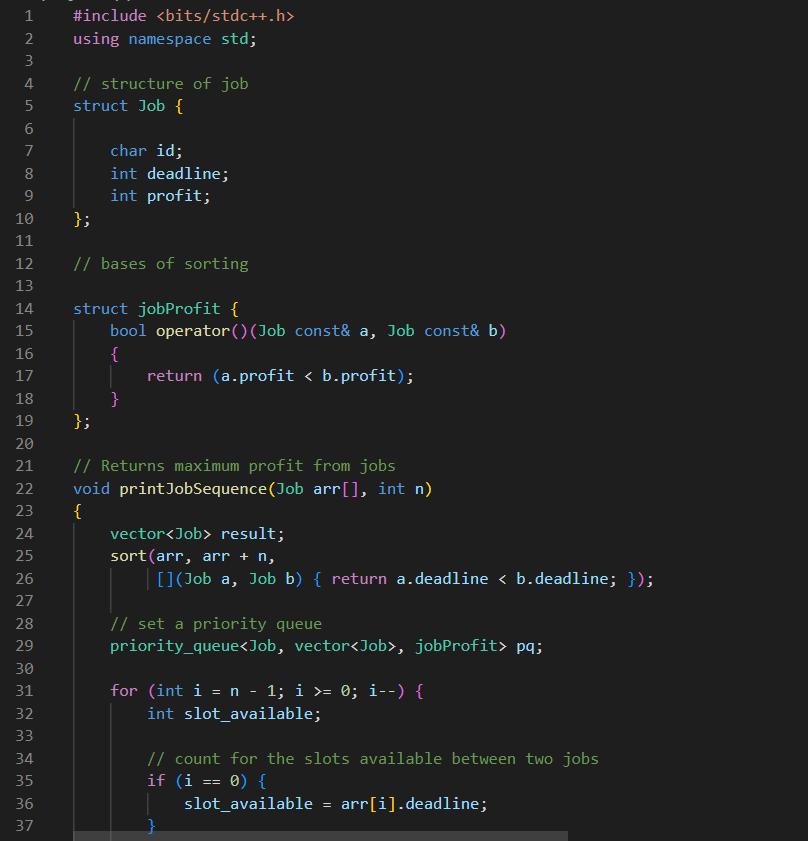
Objective:- Given a set of N jobs where each job i has a deadline and profit associated with it. Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit associated with the job if and only if the job is completed by its deadline. Find the number of jobs done and the maximum profit.

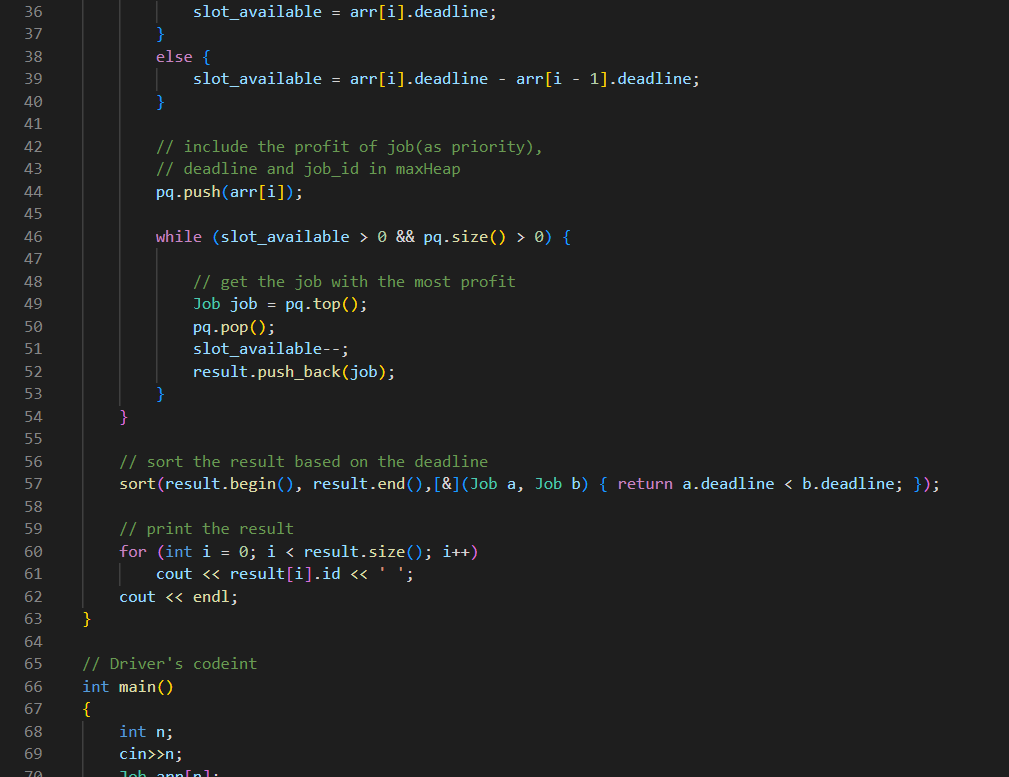
Theory:- This is a optimisation problem in which we apply greedy approach to find the optimal solution . The best of all the optimal approaches will result into final solution to give the best result and output.

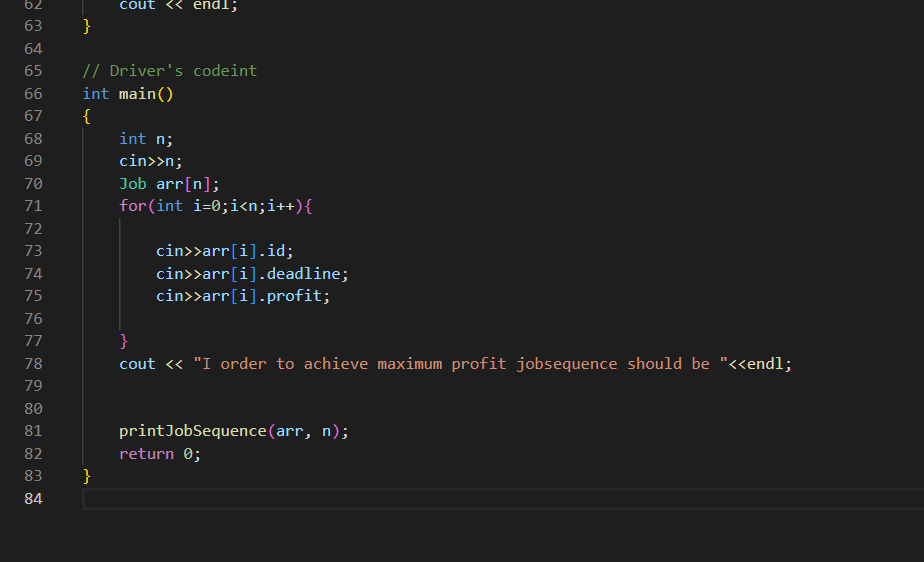
Algorithm:-

1) Sort all the jobs by decreasing order of profit using appropriate function.  
2) Initializing ,the result sequence as the first job in sorted the jobs/ given.  
3) Do following for the remaining n-1 jobs to complete all jobs.

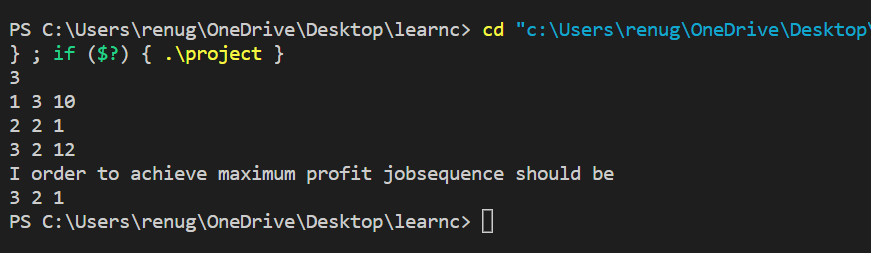
Code:-







**OUTPUT:**



Experiment No. 10

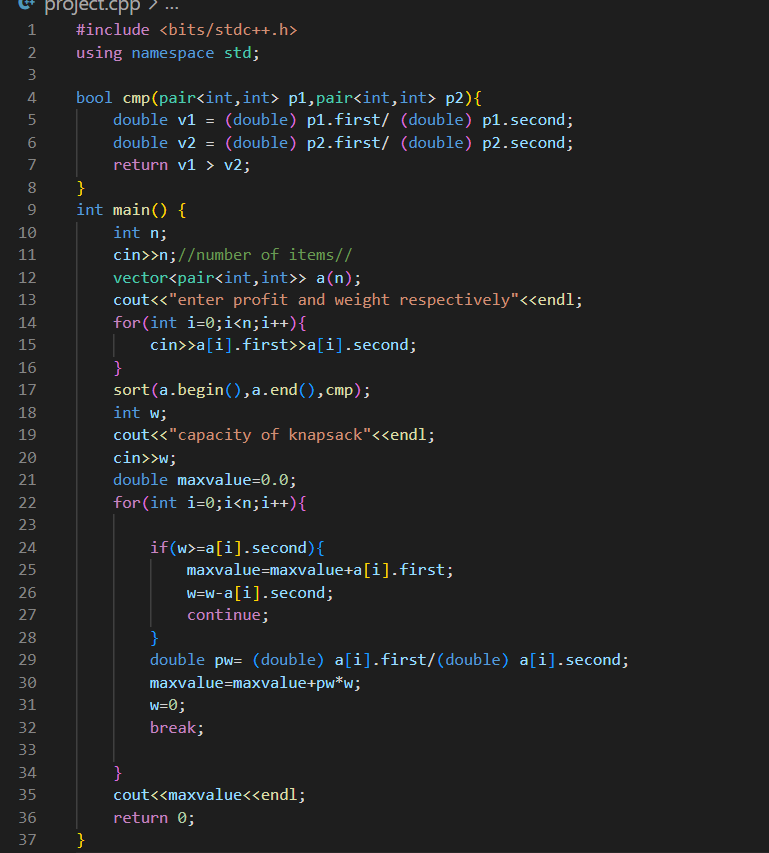
Objective:-Given weights and values of N items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack. Note: Unlike 0/1 knapsack, you are allowed to break the item.

Theory:- The elementary idea of fractional knapsack is to sort elements on the basis of profit by weight ratio. The one having high priority will be done as per priority.

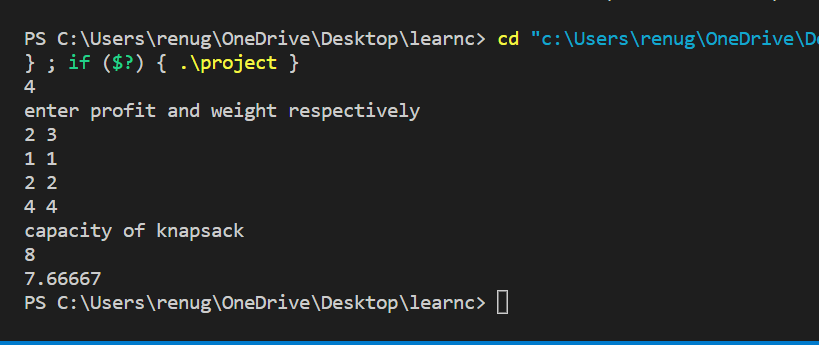
Algorithm:-

1. Calculate n/p ratio and sort them according to n/p ratio.
2. Take elements as per high n/p ratio fully as per capacity.
3. Else include the item partially as per requirement till the total capacity is 0.
4. Repeat this process n-1 times to cover all elements.

Code:-



**OUTPUT:**



Experiment no 11

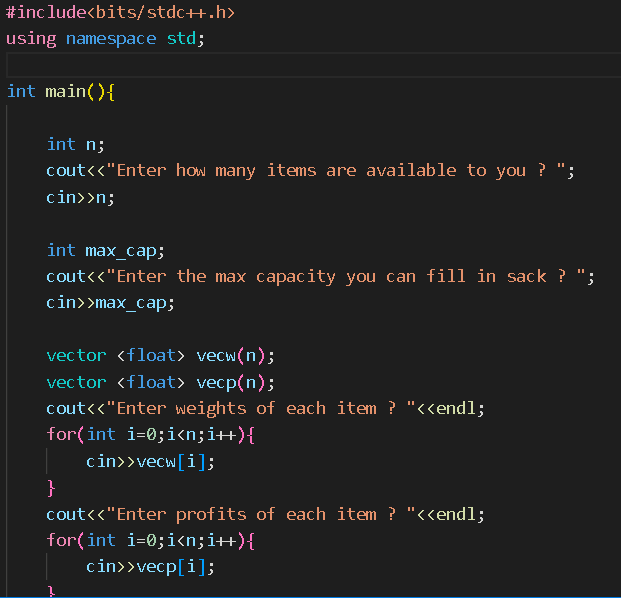
Fractional knapsack

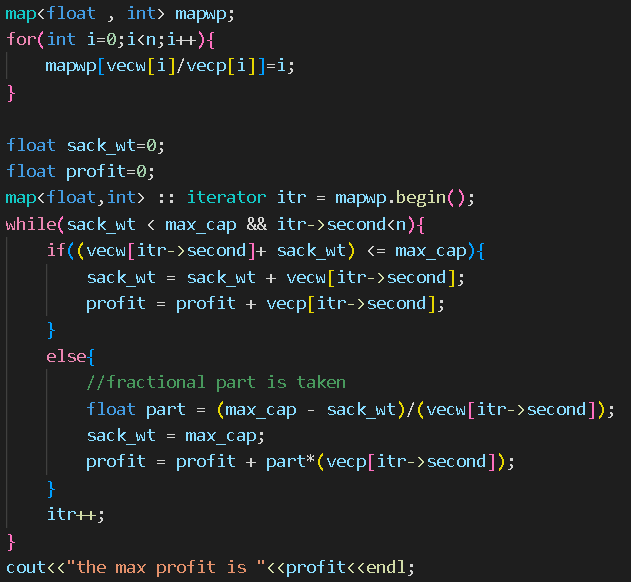
Algo:

1.sort the array in decreasing order of p/w ratio.

2. If partial capacity is left reject it.

SOURCE CODE:





**OUTPUT**

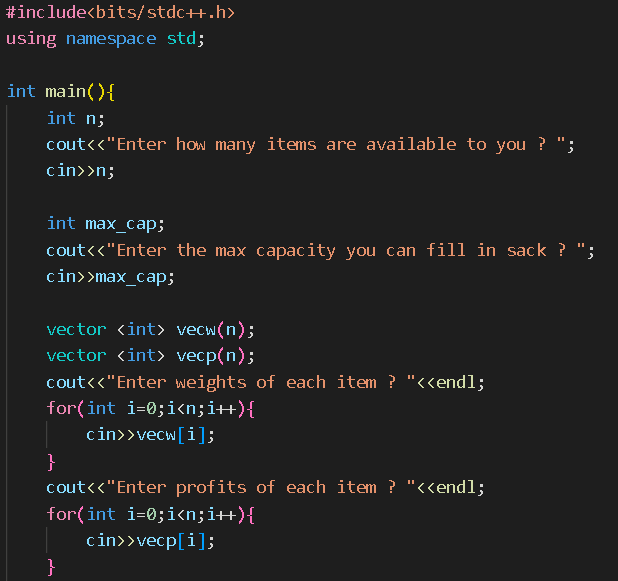


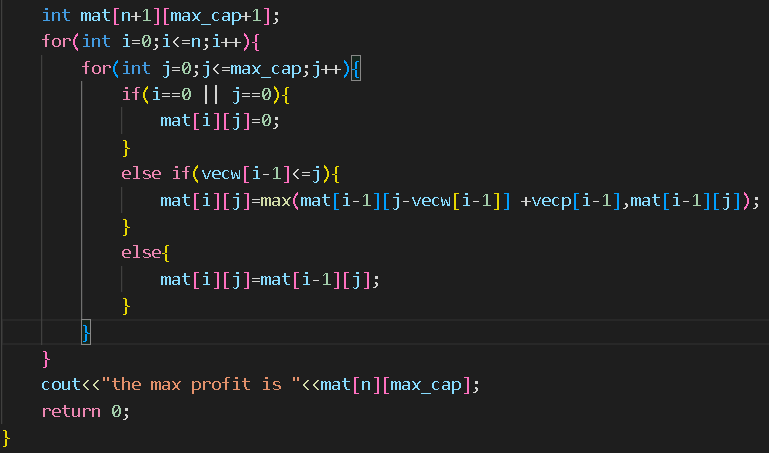
Part 2

Objective : 0-1 knapsack

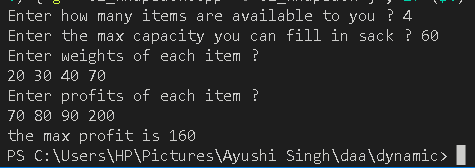
Theory:

SOURCE CODE:





OUTPUT:



Experiment No. 12

Part 1

Objective: to make a mst using prims

Theory:

It is used to create a minimum spanning tree using a given complete graph using prims algorithm.

Part 3

Topological sort using DFS traversal.

Algo:

1. Take the node from complete graph.
2. Mark it already visited.
3. If neighbours not visited recursively call topological sort as per function.
4. At the end of call push the node into the stack accordingly.

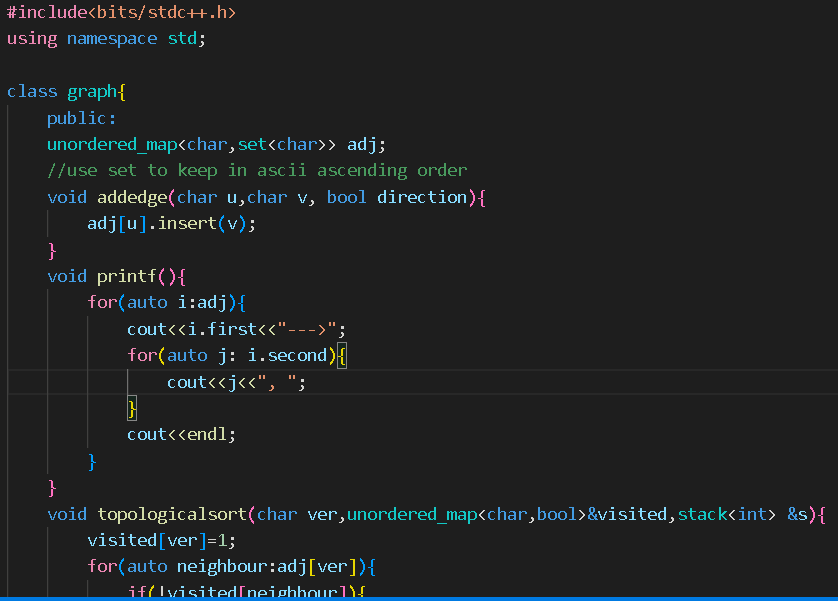
Time complexity analysis of traversal:

O(V+E)

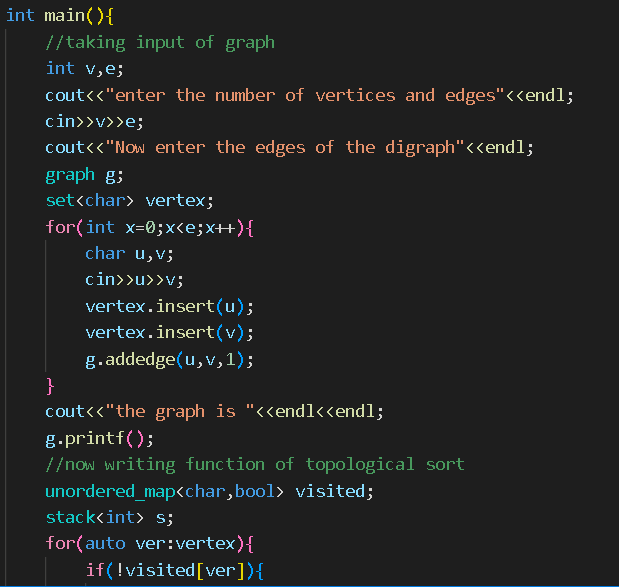
Space complexity analysis of traversal:

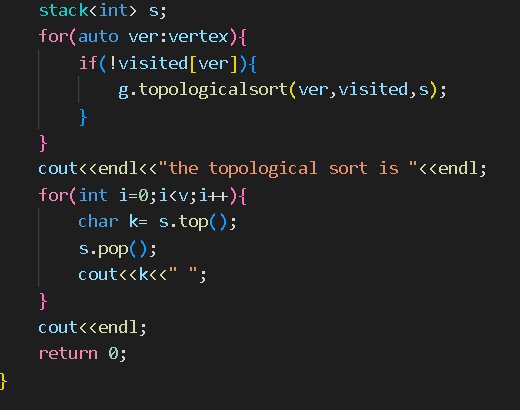
O(V+E)

SOURCE CODE:

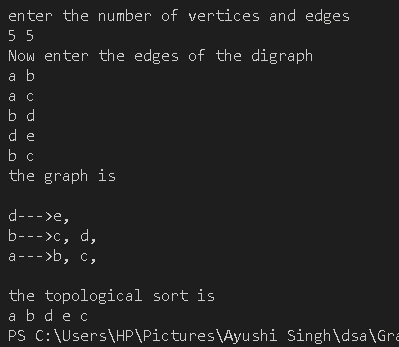






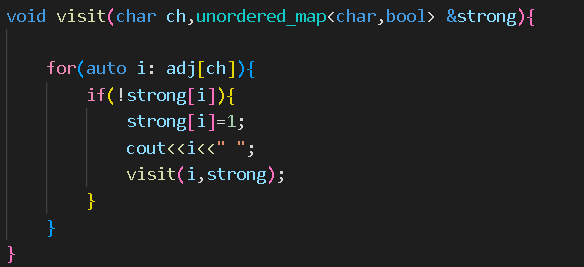


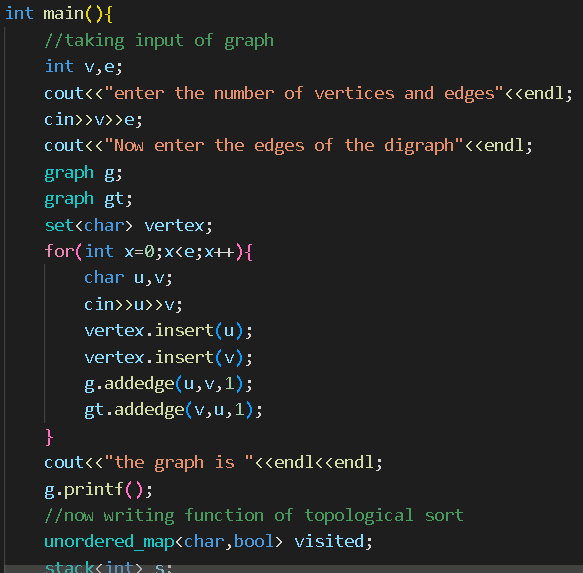
OUTPUT:

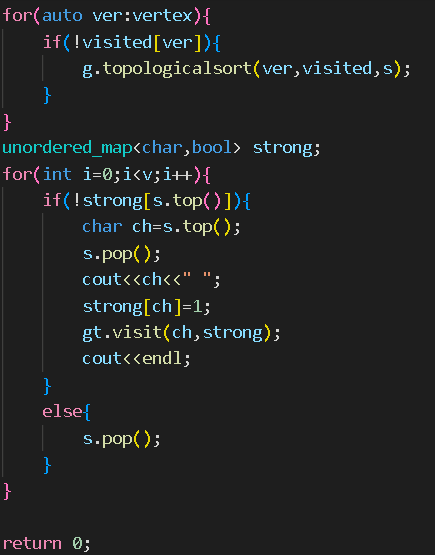


Part 4

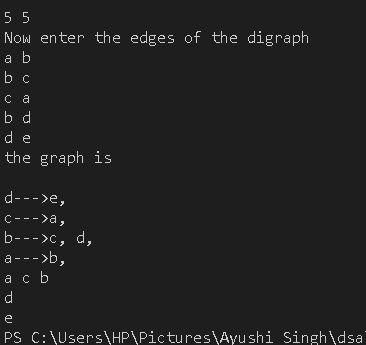
Objective :In a Directed Graph identify all strongly connected component.







OUTPUT:



Experiment No. 13

Objective:- Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications.

The dimensions of the matrix are given in an array arr of size N (where N = number of matrices + 1) .

Theory:- It is basically a Dynamic programming problem, in which we make use of recursion and store the values simultaneously so that we don't have to solve the same overlapping problems again and again and also time complexity is reduced. The algorithm applied is

1.Create a recursive function that uses i and j as two parameters variables.

2.Iterate using loop from k = i to j to partition the given range into two groups.

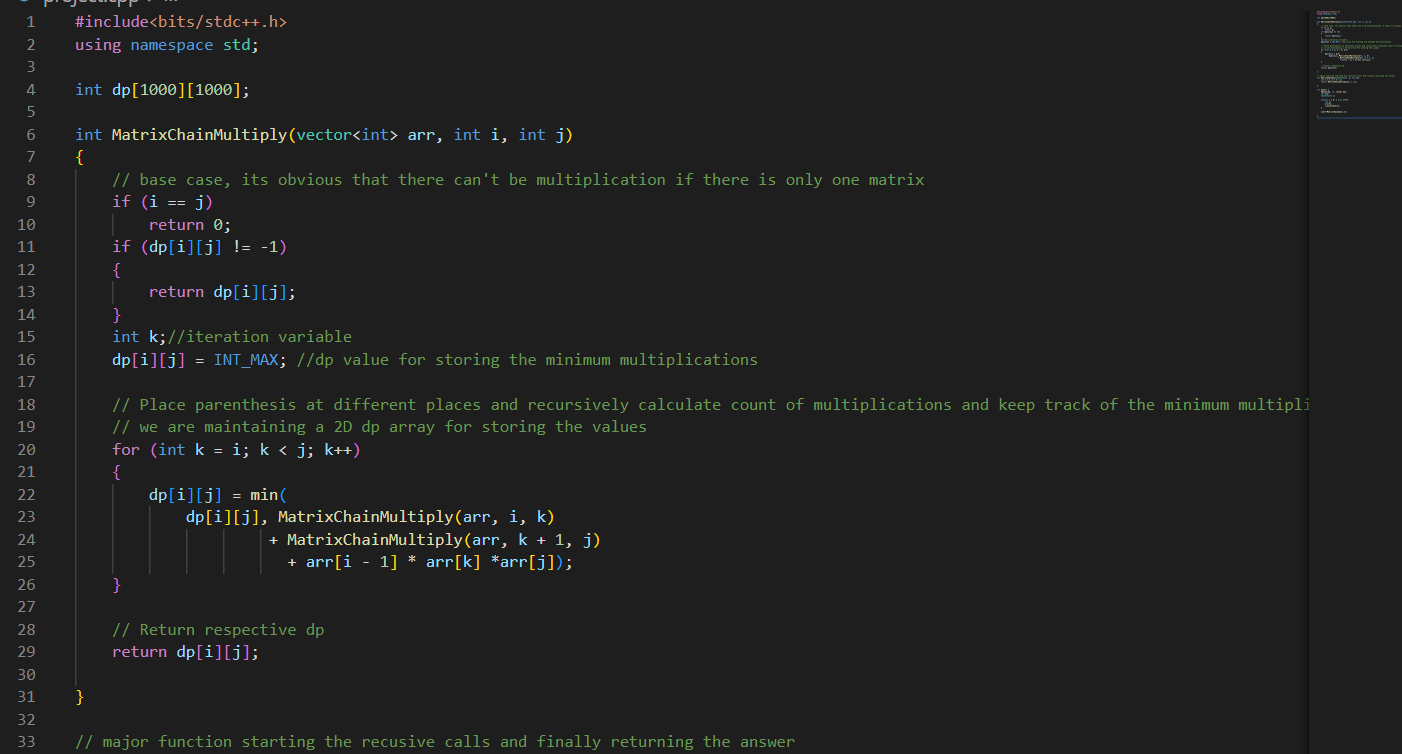
3.Call the respective recursive functions for these groups.

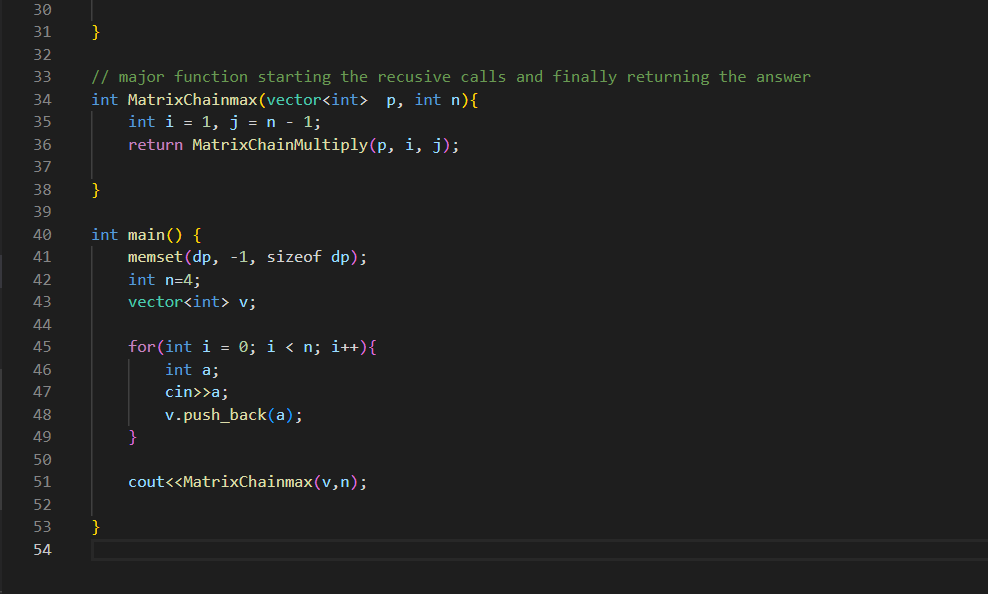
4.Return the minimum value found as the required minimum number of the multiplications here to multiply all the matrices of this group.

5.The minimum value returned here for the range 0 to N-1 is the required answer to the problem.

The complexity of this algorithm is O(N3) and space complexity is O(N2).

Code:-





OUTPUT:



Experiment No. 14

1. **Strassen’s Matrix Multiplication**

Objective :- Write a program to implement Strassen’s Matrix Multiplication.

Input: 6 (Number of matrices, followed by matrix size) 2 4 4 3 3 6 6 5 5 2 2 1

Output: 23XX22 (Number of operations) (A1 ((A2 A3) (A4 (A5 A6))))

Theory:-Strassen gives a method to find matrix multiplication by brute force algorithm using divide and conquer. For multiplying the two matrices Strassen used some formulas in which there are seven multiplications and eighteen additions, subtraction, and in the brute force method, there is eight multiplications and four addition hence increasing time complexity. Algorithm:-

Begin

If n = the threshold then compute

C = a \* b is the conventional matrix.

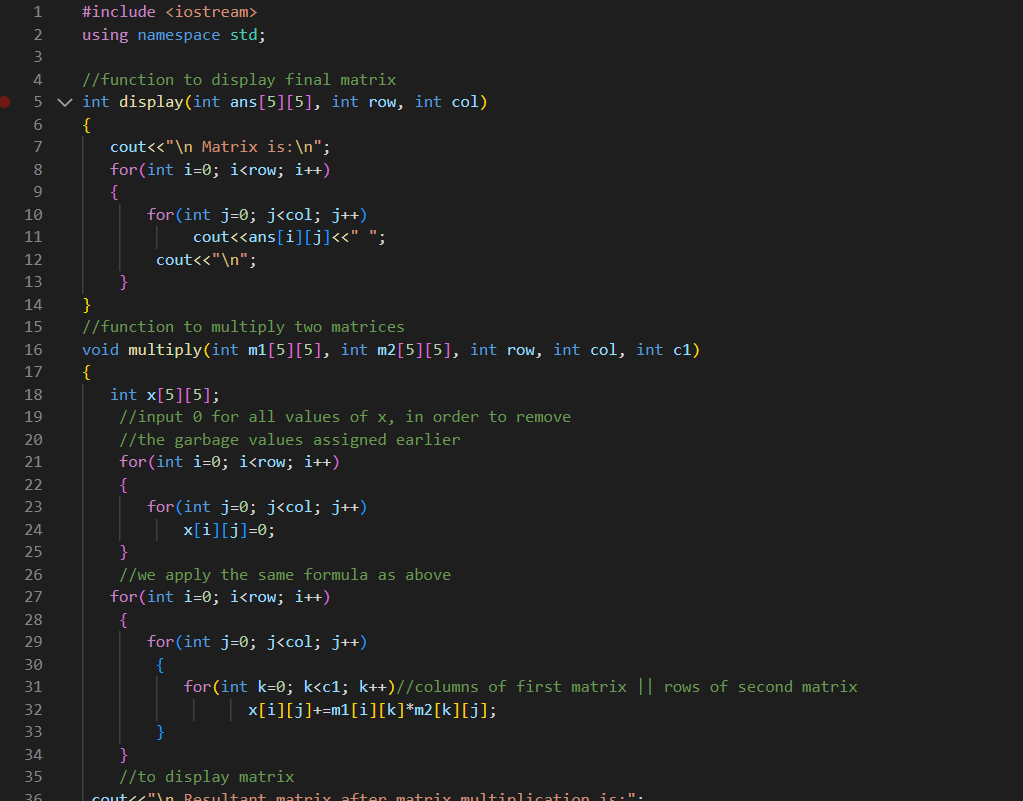
Else

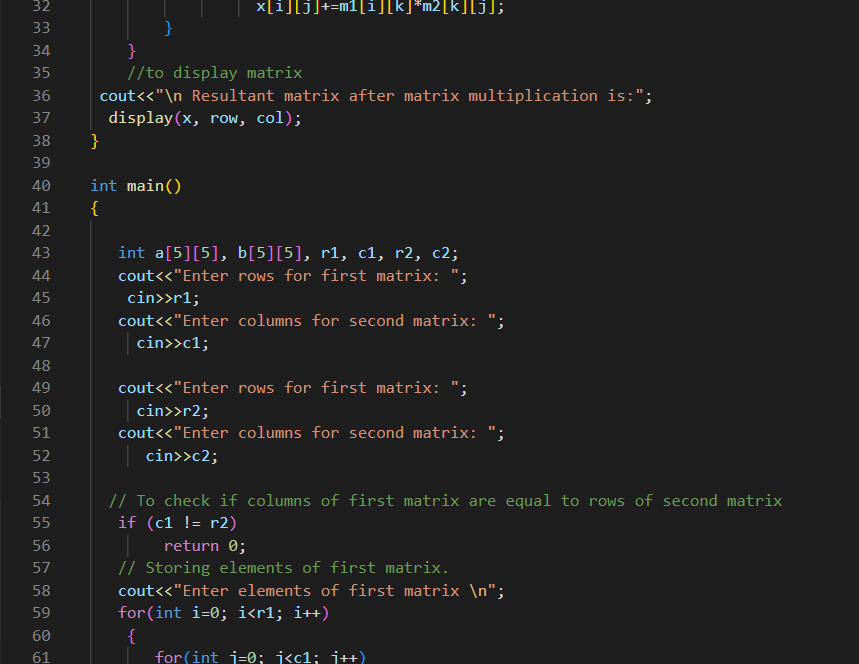
Partition a into four possible sub matrices given as a11, a12, a21, a22

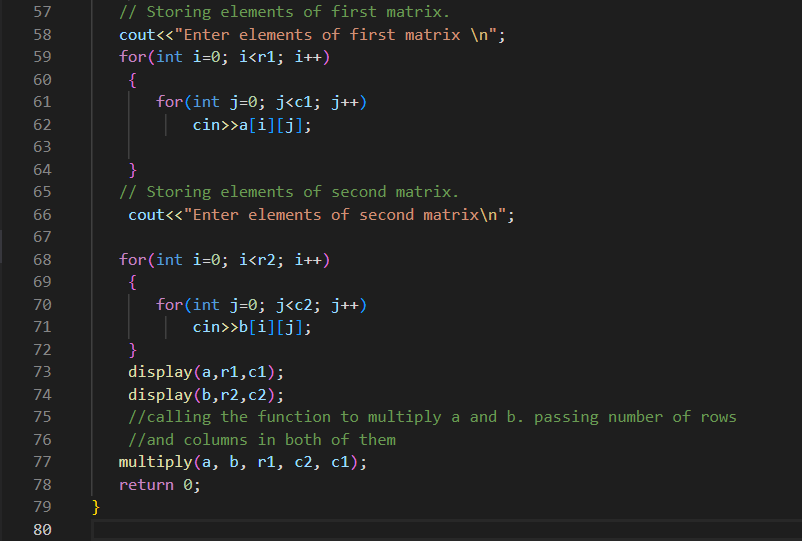
Partition b into four possible sub matrices given as b11, b12, b21, b22.

Complexity:-The time complexity is O( N2.80)i.e. O( N^log7 base 2)

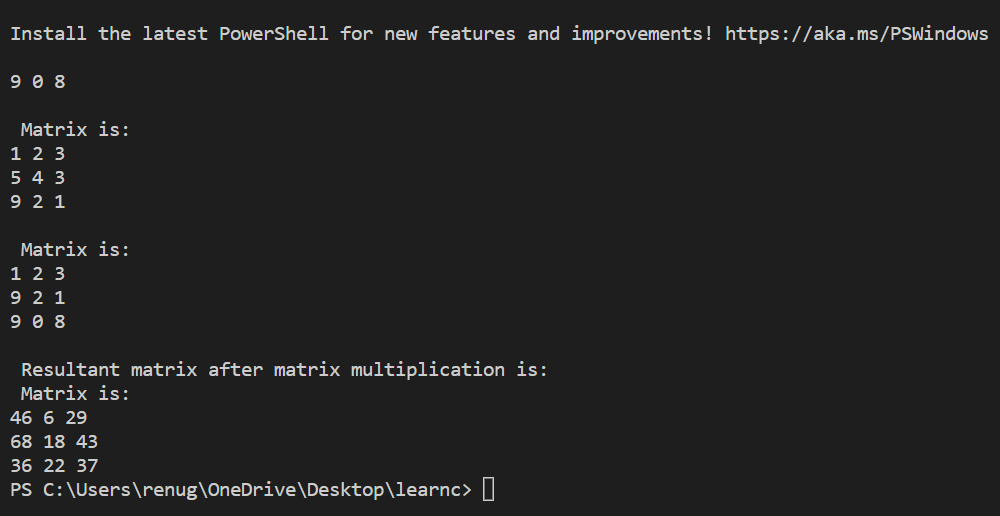
Code:-







OUTPUT:



1. **Matrix Chain Multiplication**

Objective:- Implement Matrix chain multiplication (MCM) using dynamic programming, you need to estimate the minimum number of operations and assign the parentheses for multiplying multiple matrices.

Input: 6 (Number of matrices, followed by matrix size) 2 4 4 3 3 6 6 5 5 2 2 1

Output: 23XX22 (Number of operations) (A1 ((A2 A3) (A4 (A5 A6))))

Theory:-Matrix chain multiplication is an optimization problem with the aim to find the most efficient way to multiply a given sequences of matrices.

It is done by Dynamic Programming to find the best possible optimal approach.

Time Complexity:- O(N^3)

Experiment No. 15

Longest Common Subsequence

Objective:- Write a program to implement Longest Common Subsequence.

Theory:-

In LCS problem we need to find the Longest common possible subsequences of all the possible subsequences.

A subsequence is a pattern that appears in same relative order.

This problem is done with the help of Dynamic Programming.

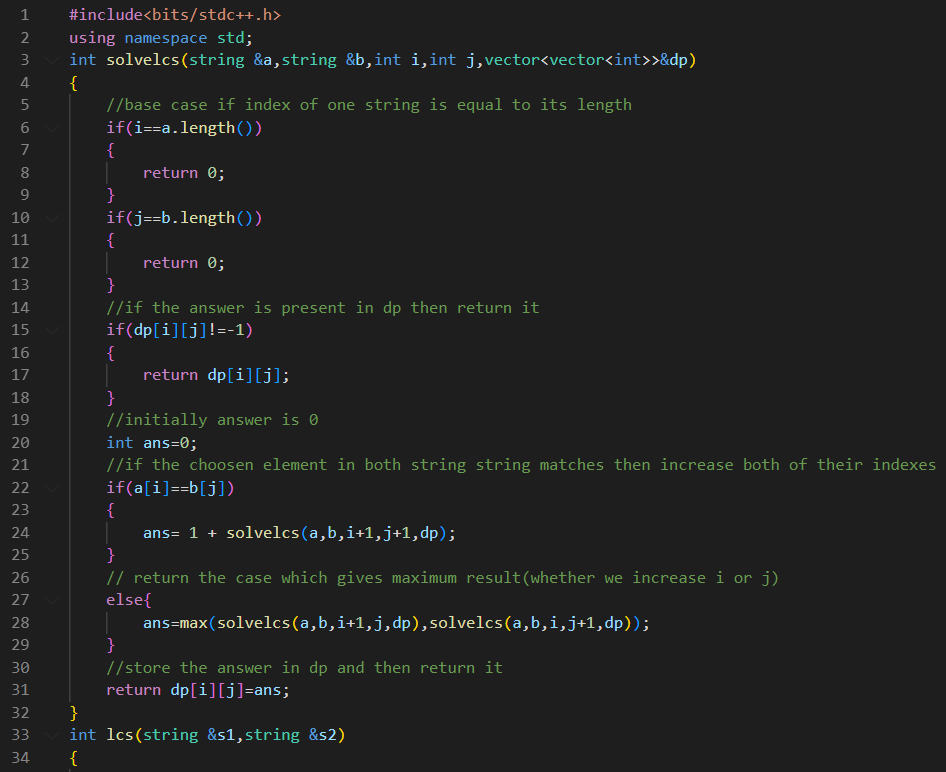
Algorithm:-

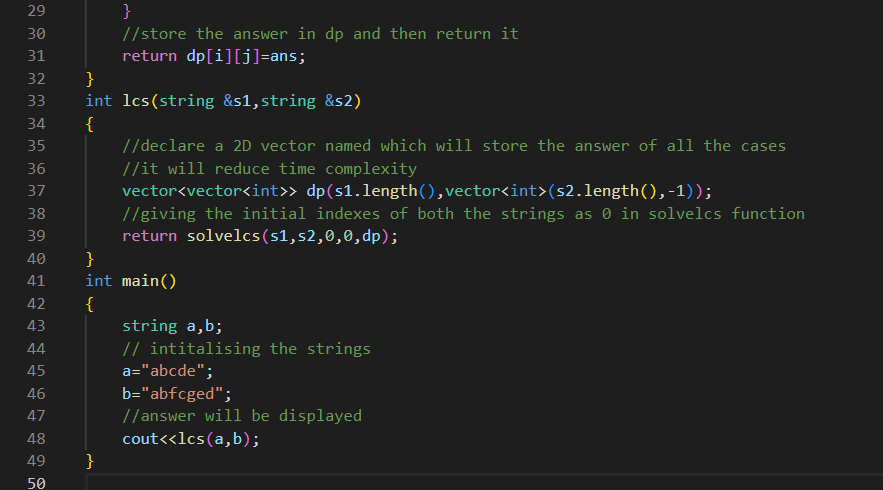
1. A 2D vector will be created to store all the results of given problems(n+1,m+1).
2. Fill the first row and column of vector with zeroes.
3. If the character corresponding to the given current row and current column are matching, add one to the diagonal element.
4. Else take the max value from the previous made columns and previous row element for filling the current cell. Repeat again the previous Step 2.
5. The value in the last row and the last column is the length of the longest common subsequence.
6. In order to find the longest common subsequence start from the last element of string and follow the direction of the arrow using variables pointers. The elements corresponding to the () symbol form the longest common subsequence in the array.

Time Complexity:

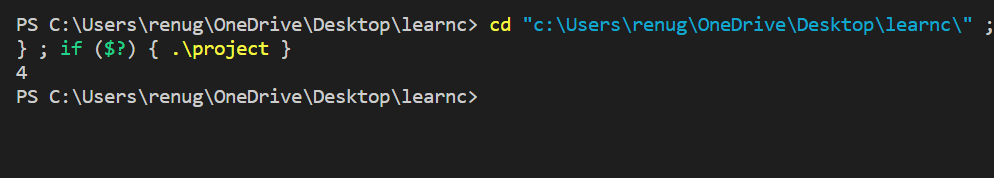
The complexity of the problem is O(n\*m) where n and m are the length of the two given strings.

Source Code:-





OUTPUT:



Experiment No. 16

Travelling Salesman Problem

Objective:- Implement Travelling Salesman Problem.

Theory:- Given a set of cities and the distance between every pair of cities, we need to find the minimum cost of travelling from one city to another city.

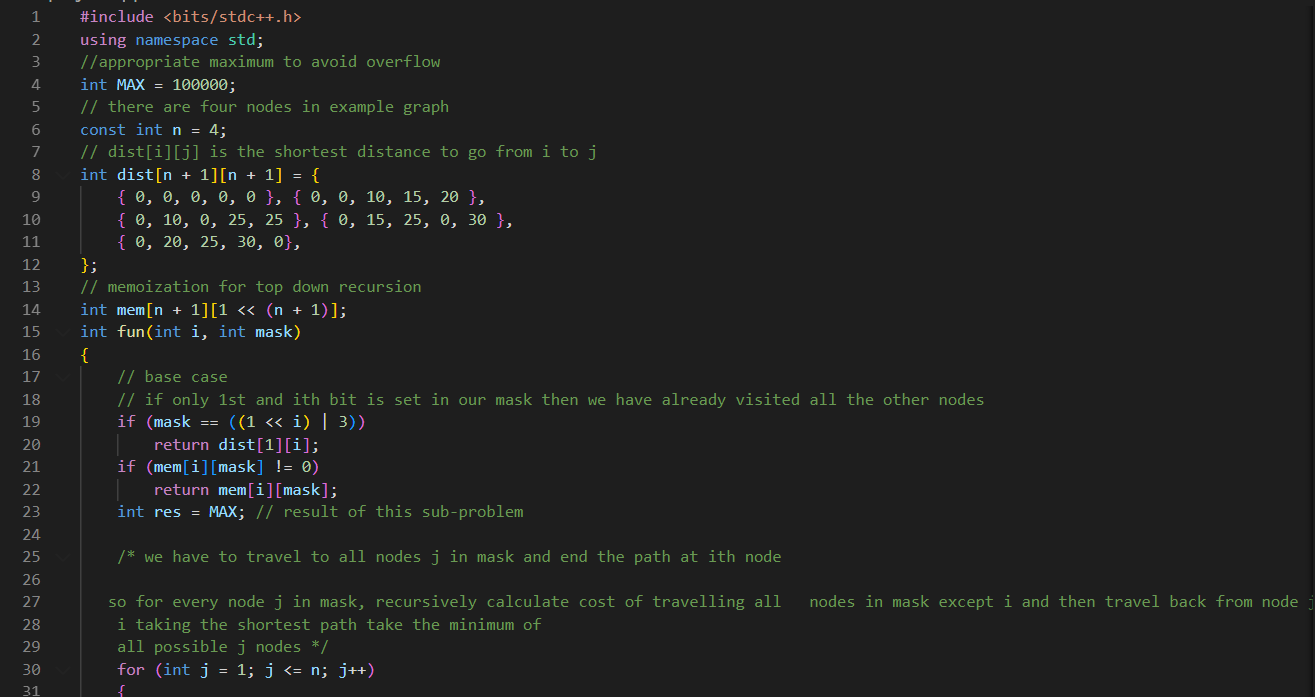
Algorithm:- Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 vertex as our starting and ending point of output. For every other vertex (other than starting vertex), we find the minimum cost path of travel with 1 as the starting point, I as the ending point of journey, and all vertices occuring exactly once.

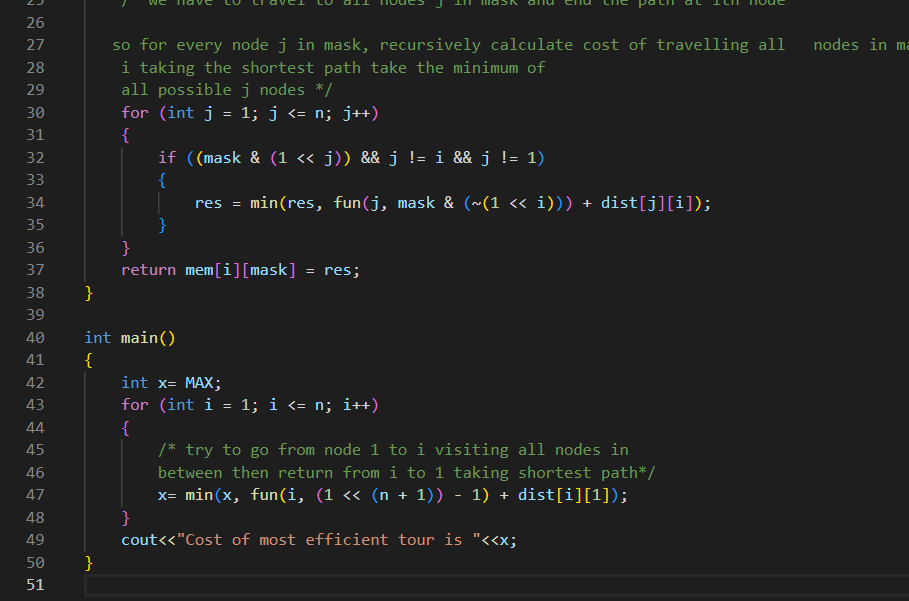
Let the cost of this given path cost (i), and the cost of the corresponding Cycle here would cost (i) + dist(i, 1) where dist(i, 1) is the distance from i to 1.

Finally, we return the minimum value of all [cost(i) + dist(i, 1)] values. To calculate the cost(i) of the problem using Dynamic Programming, we need to use recursive relations in terms of sub-problems.

Time Complexity:- The total running time i.e. complexity is hence O(n2\*2n). The time complexity is exponential.

Source Code:-





OUTPUT:

