DAA LAB FILE



NAME-HARSHIT GUPTA

ROLL NO.-2021UCS1621

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,Quick Sort,Insertion Sort,Selection Sort,Bubble Sort.

**Theory** :

**Merge Sort**-

It is a sorting technique based on divide and conquer approach. In this technique we recursively break the array into two halves to make a smallest sub problem then these are merged in sorted manner. **Time Complexity-O(nlogn)**

**Quick Sort**-

It is another sorting technique based on divide and conquer approach in this technique we assign a pivot and iterate using two variable in starting and ending till an element comes to its actual sorted position and now we divide the left and right half of sorted element.and repeat the process. **c-O(nlogn)**

**Insertion Sort**–

Iit is a sorting technique in which we we divide the array in to sorted and unsorted array and an element form unsorted array is placed to the correct position in sorted array. **Time Complexity-(n\*n)**

**Selection Sort**–

In this technique we find the minimum element from the unsorted array and place it in 0th index repeatedly. **Time Complexity-(n\*n)**

**Bubble Sort**-

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high. **Time Complexity-(n\*n)**

**Algorithm** :

*Merge Sort :*

1.stat

2.declare array and left, right, mid variables

3.perform merge function.

if left > right

return

mid= (left+right)/2

mergesort(arr, left, mid)

mergesort(arr, mid+1, right)

merge(arr, left, mid, right)

*Quick Sort*-

QUICKSORT (array A, start, end)

{

**if** (start < end)

  {

 PIVOT = DIVIDE(A, start, end)

QUICK (A, start, PIVOT- 1)

QUICK (A, PIVOT + 1, end)

}

}

*Insertion Sort* -

Begin

   for i := 1 to size-1 do

      key := arr[i]

      j := i

      while j > 0 AND arr[j-1] > key do

         arr[j] := arr[j-1];

         j := j – 1

      done

      arr[j] := key

   done

End

*Selection Sort*-

Follow the below steps to solve the problem:

1.Initialize minimum value(min\_idx) to location 0.

2.Traverse the array to find the minimum element in the array.

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

4.Then, increment min\_idx to point to the next element.

5.Repeat until the array is sorted.

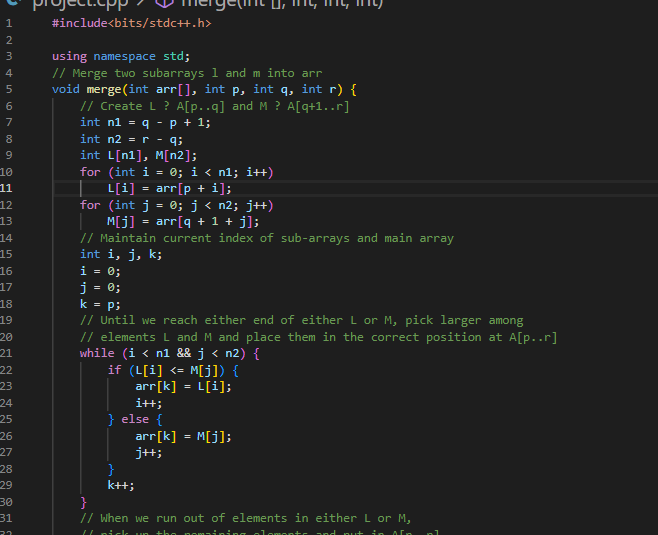
*Bubble sort* -

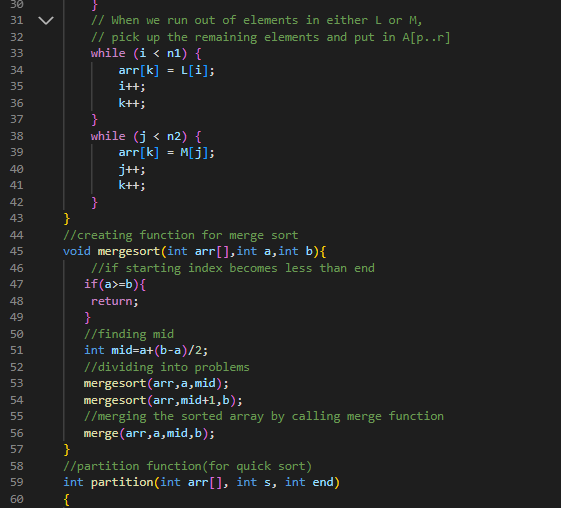
1.run a nested for loop using variable I and j such that outer iterator 0 ≤ i < n-1 and outer iterator 0 ≤ j < n-i-1

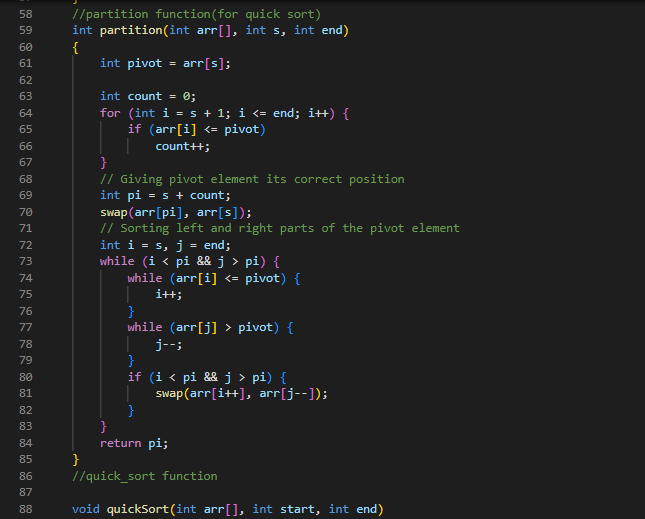
2.If arr[j] >arr[j+1] swap these adjacent elements, else continue and

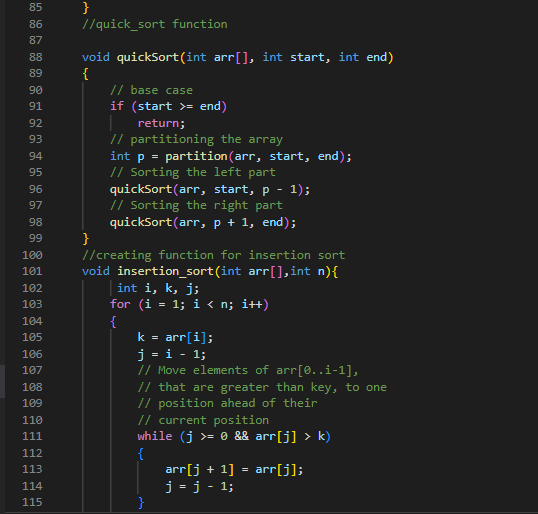
Print the sorted array

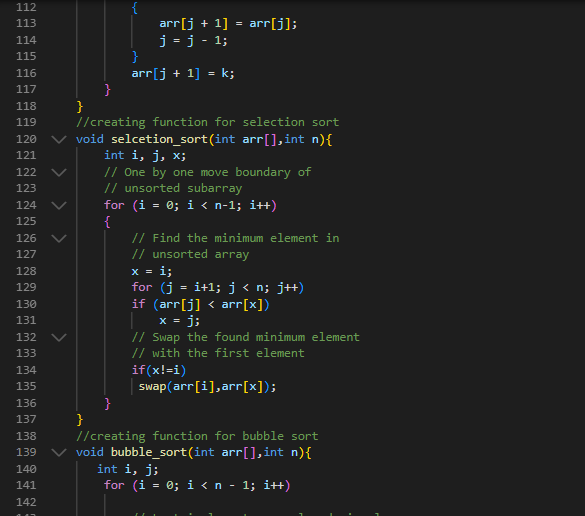
**CODE:-**

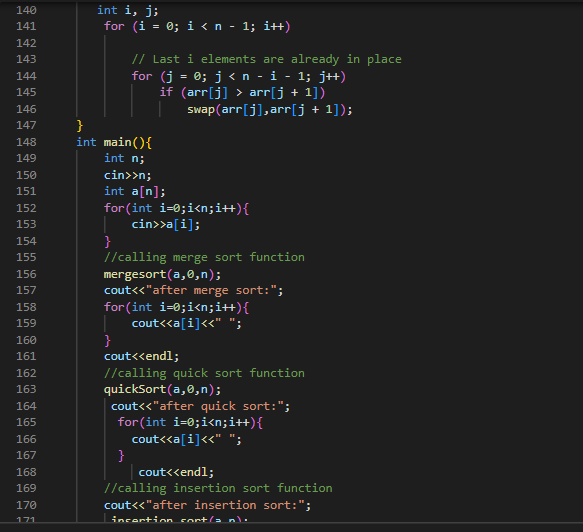


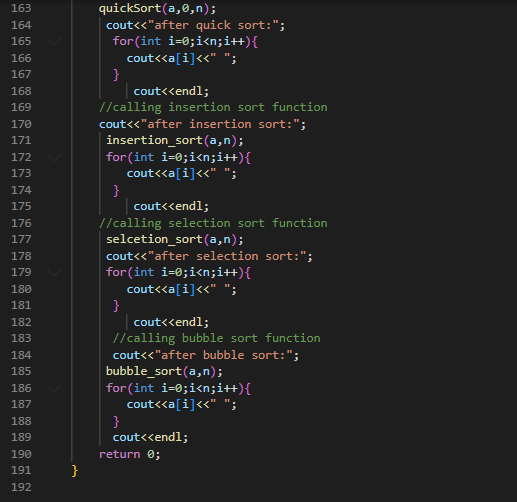




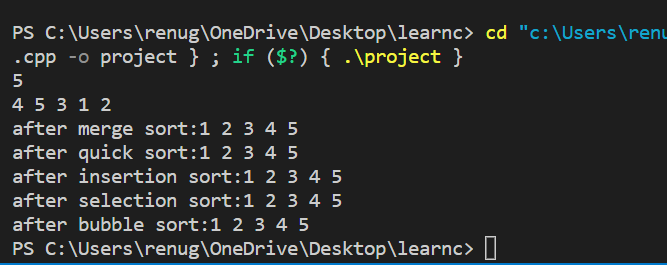








OUTPUT:-



Experiment No.2

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

Theory:- Radix sort: idea behind the radix sort is to digit by digit or character by character sort starting from least significant digit to most significant digit.it is uses counting sort as a subroutine to sort.

**Algorithm:**

Radixsort(a,n){

Int max=getmax(a,n)

For(pass=1,max/pass,pos\*10){

Countsort(a,n,pass)}

}

Time complexity: best-case scenario occurs When all elements have the same number of digits, the. O(a(n+k)) is the best-case time complexity. If b equals O(n), the time complexity is O. (a\*n).

Count Sort: Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values. Then do some arithmetic to calculate the position of each object in the output sequence.

**Algorithm:**

COUNTSORT(arr, size)  
max = largest element in arr  
DECLARE ALL COUNT arr as 0  
for j : 0 to size  
find count of all unique element and store it in jth index of count arr  
for i : 1 to max  
find the sum and store in array  
for j : size down to 1  
restore the elements to arr  
for each restored element decrease the count by 1.

Bucket sort: in this sortingtechnique objects are first classified in an a particular group than these are internally sorted using count sort generally.

**Algorithm:**

Begin

for i := 0 to size-1 do

insert array[i] into the bucket index (size \* array[i])

done

for i := 0 to size-1 do

sort bucket[i]

done

for i := 0 to size -1 do

gather items of bucket[i] and put in array

done

End

Time complexity- O(n+k) .

Shell sort:

It is a sorting technique derived fron insertion sort in which we first take the gap between the element n and keep on decreasing by half.

**Algorithm:**

* **Decalre the variable gap**
* **Initialise it with the value size where size is the length of the arrays**
* **Apply insertion sort**
* **On each pass go on decreasing the gap by the factor of 2.**

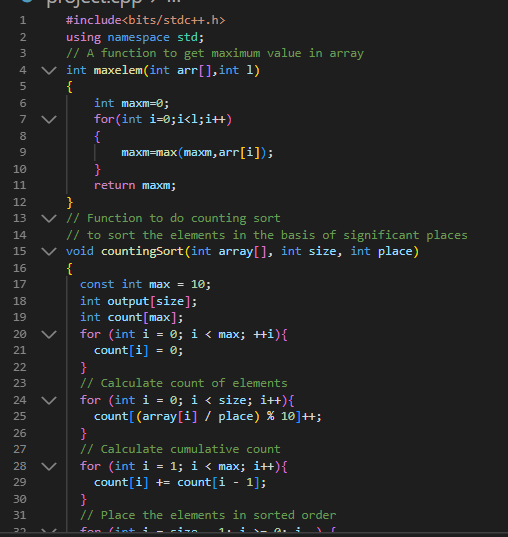
**Time complexity:**

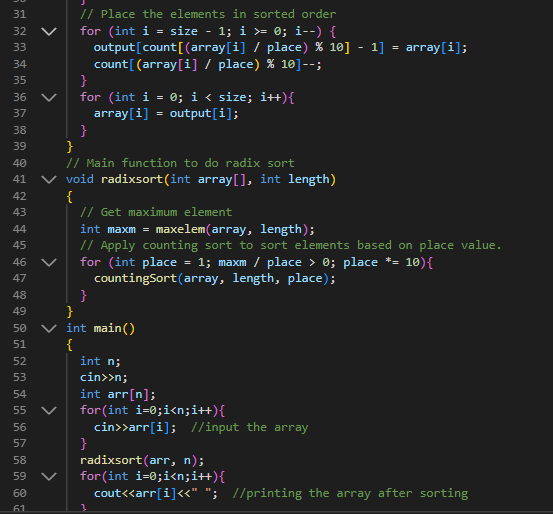
best case complexity is O(n logn).

worst-case complexity for shell sort is  O(n\*n).

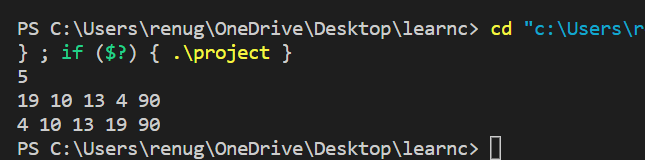
**CODE:-**

**RADIX SORT**

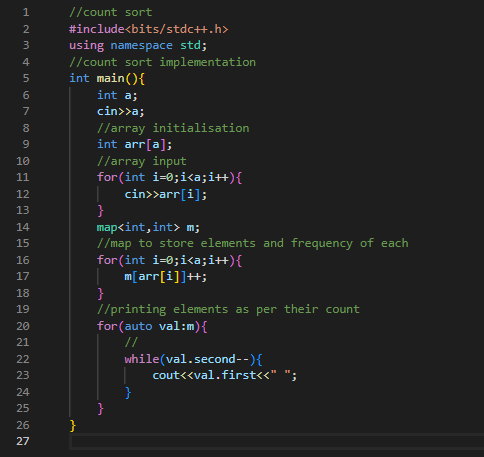




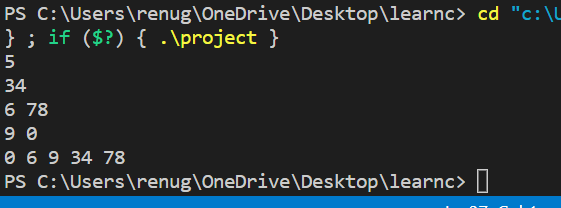
OUTPUT:-



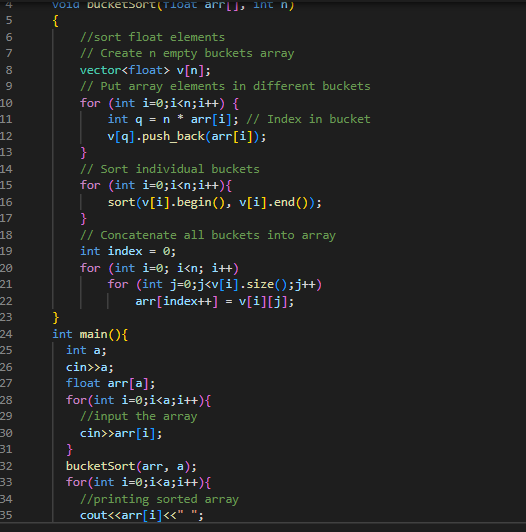
COUNT SORT:



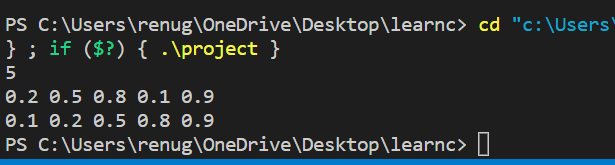
Output:



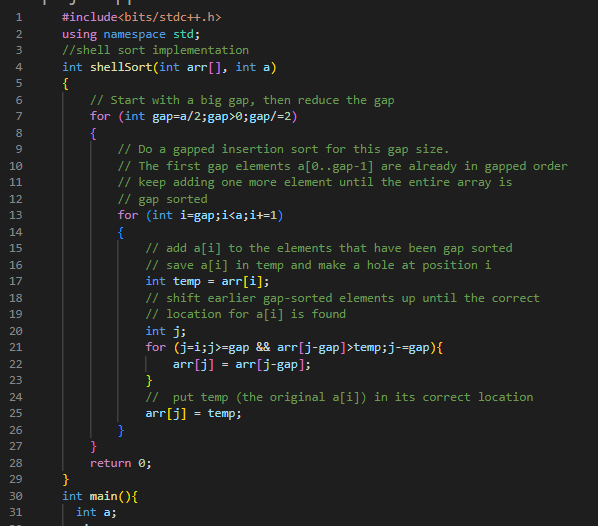
BUCKET SORT:

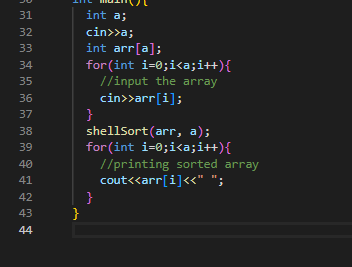


Output:

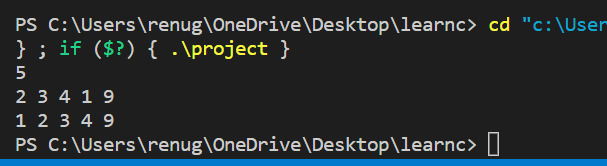


SHELL SORT:





OUTPUT:



Experiment No. 3

Objective:- Perform Linear and Binary search operations.

Theory:-

Linear search is the most basic searching technique in which we traverse the array completely time complexity

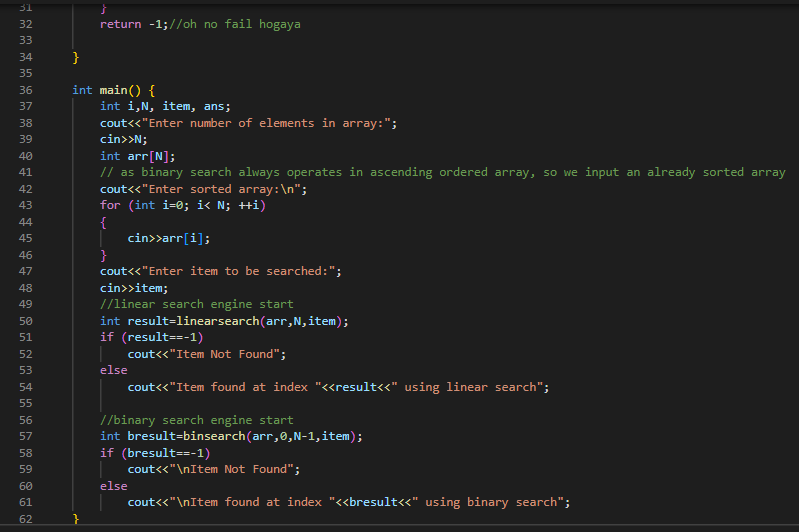
Best-O(1)

Worst-O(n)

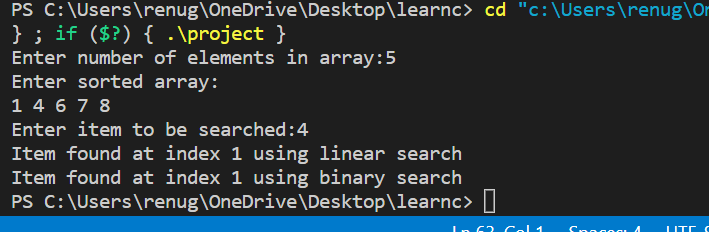
Binary search is based on divide and conquer approach in this sorting technique we search the element always in the middle odf two parameter start and end and keep on reducing the parameter gap,array must be sorted for that algorithm.time complexity –O(logn)

**Code:-**





OUTPUT:



Experiment No. 4

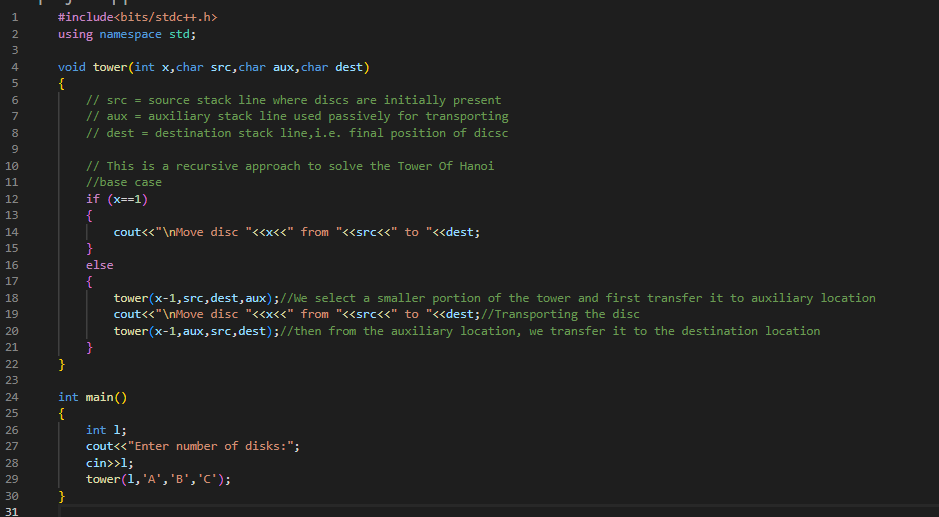
Objective:- Solving standard Tower of Hanoi.

Theory:- Tower of Hanoi is a recursion based puzzle and thus, we will follow a recursive approach to solve it.it consists three imaginary pillars called source ,auxillary and destination

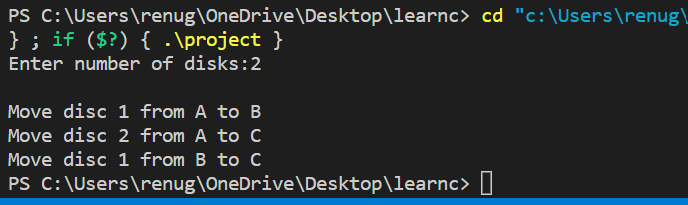
 The idea is to use the helper node to reach the destination using recursion. Below is the pattern for this problem:

**1:** Move n-1 disks from source to auxiliary   
**2:** Move the nth disk from source to destination  
**3:** Move n-1 th disks from aux to dest

**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**

It is a binary tree in which left subtree is always less than the root node element and right subtree is always grater than the root element.time complexity of insertion- O(H)

**AVL TREE :—**AVL tree is a binary search tree in which height of every node is always either 0,1 or -1. time complexity of insertion- O(logn)

We insert the data in AVL tree same as we do it in a BST but here when any node get height ather than 0,1 or -1 we do appropriate rotations to balance the height like

1. Ll rotation
2. Rr rotation
3. Rl rotation
4. Ir rotaion

**RED- BLACK TREE :—** it is a kind of avl tree in which each properties of red black tree are as follows;

1. Root node is always black.
2. Two adjacent node cannot be red
3. Leaf node i.e. nil should be black
4. For each path to leaf node there must be same no. of black node

**ALGORITHM:**

**Binary Search Tree:**

InsertBST (c )  
 b = NIL  
 a = root  
 **while** a!= NIL  
 b = a  
 **if** c.key < c.key  
 a = a.left  
 **else** a = a.right  
 c.parent = b  
 **if** b == NIL  
 root = c  
 **elseif** c.key < b.key  
 b.left = c  
 **else** b.right = c

**AVL Tree:**

* Let ‘A’ be a node.
* Once a is added according to bst insertion find unbalanced node
* Let z be the unbalanced node found
* Node y is the child of z and x be the grandchild of z.

Then there will be four possibilities

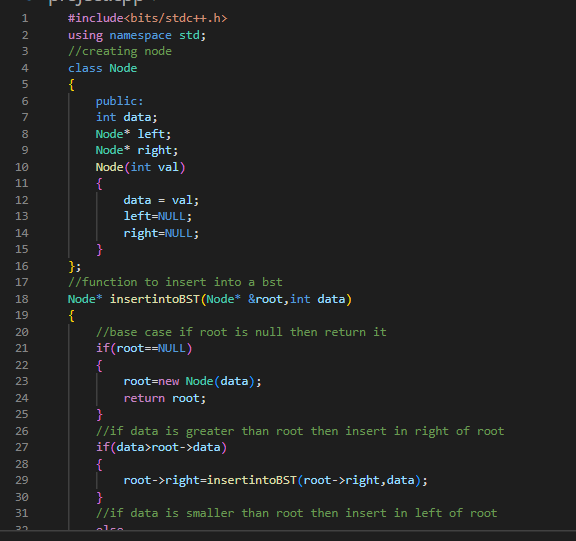
1. **Left-Left rotation** : – x is the left child of y and y is the left child of z.
2. **Left-Right rotaion :** – x is the right child of y and y is the left child of z.
3. **Right-Left rotation :** – x is the left child of y and y is the right child of z.
4. **Right-Right rotation** : – x is the right child of y and y is the right child of z.

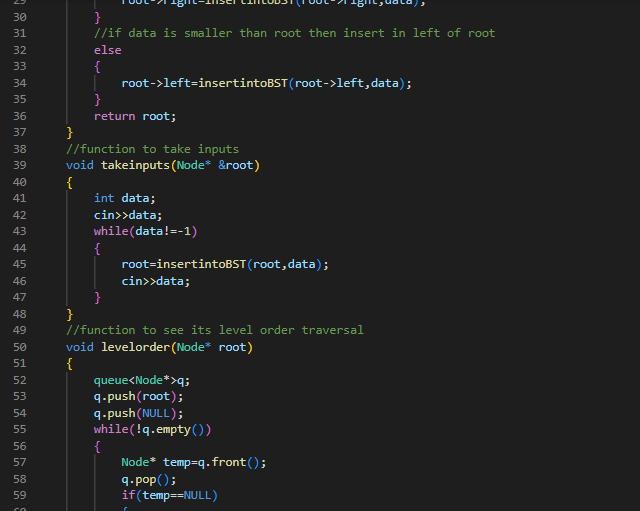
**Red-Black Tree:**

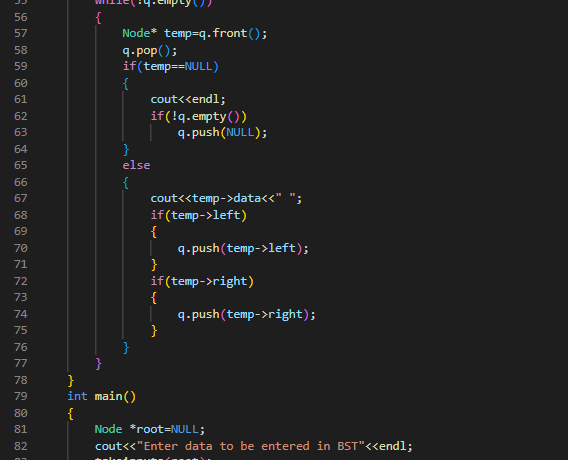
* **If tree is empty create a node and color it black**
* **If not empty create a node and color it red**
* **Exit if parent is black**
* **If parent of newnode is red than check color of parent sibling if it is black or null than do suitable rotation and recolor it**
* **Else recolor and check if parent’s parent of new node is not root node than recolor and recheck**

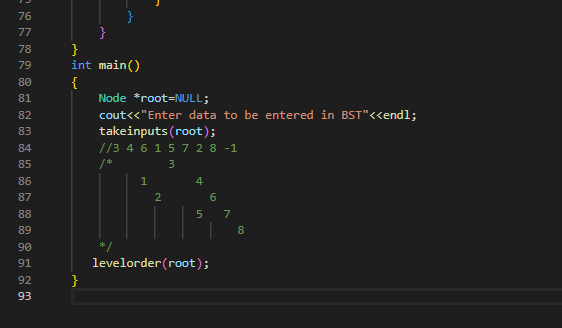
**SOURCE CODE:--**

**Binary Search Tree:--**

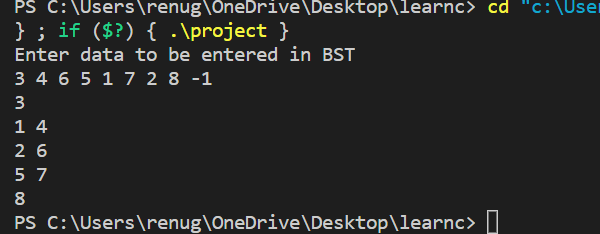






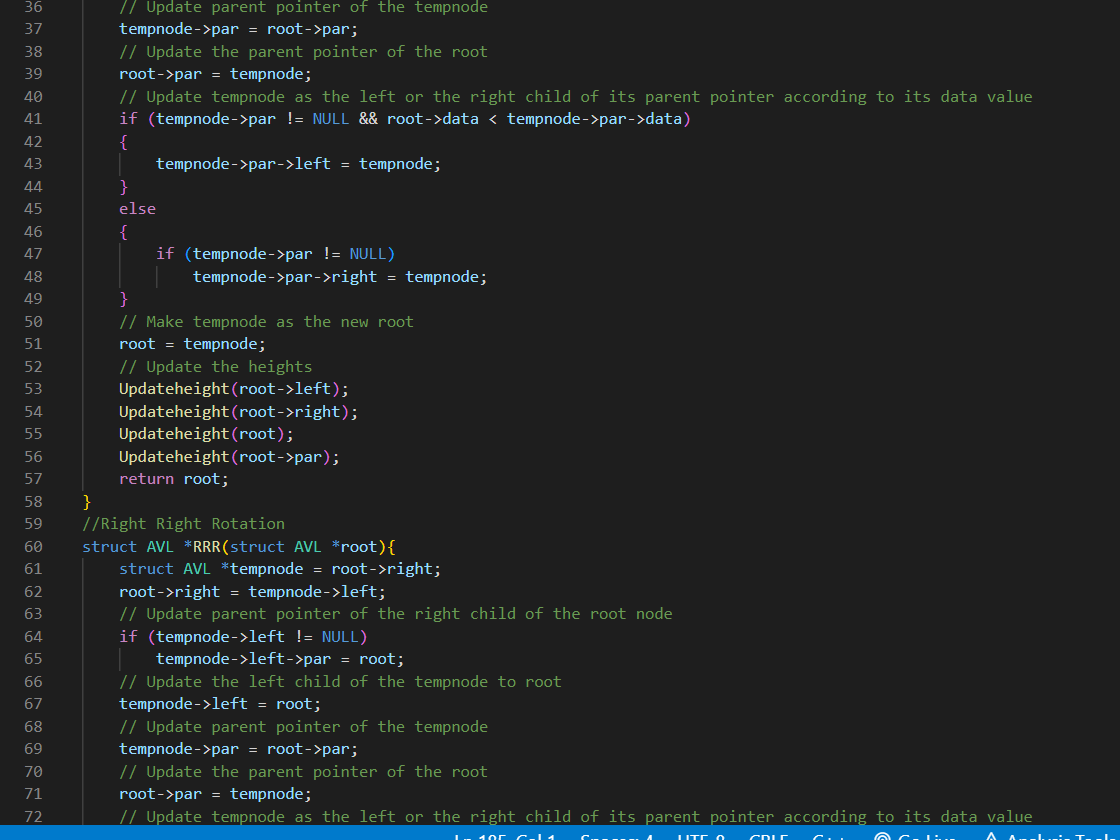


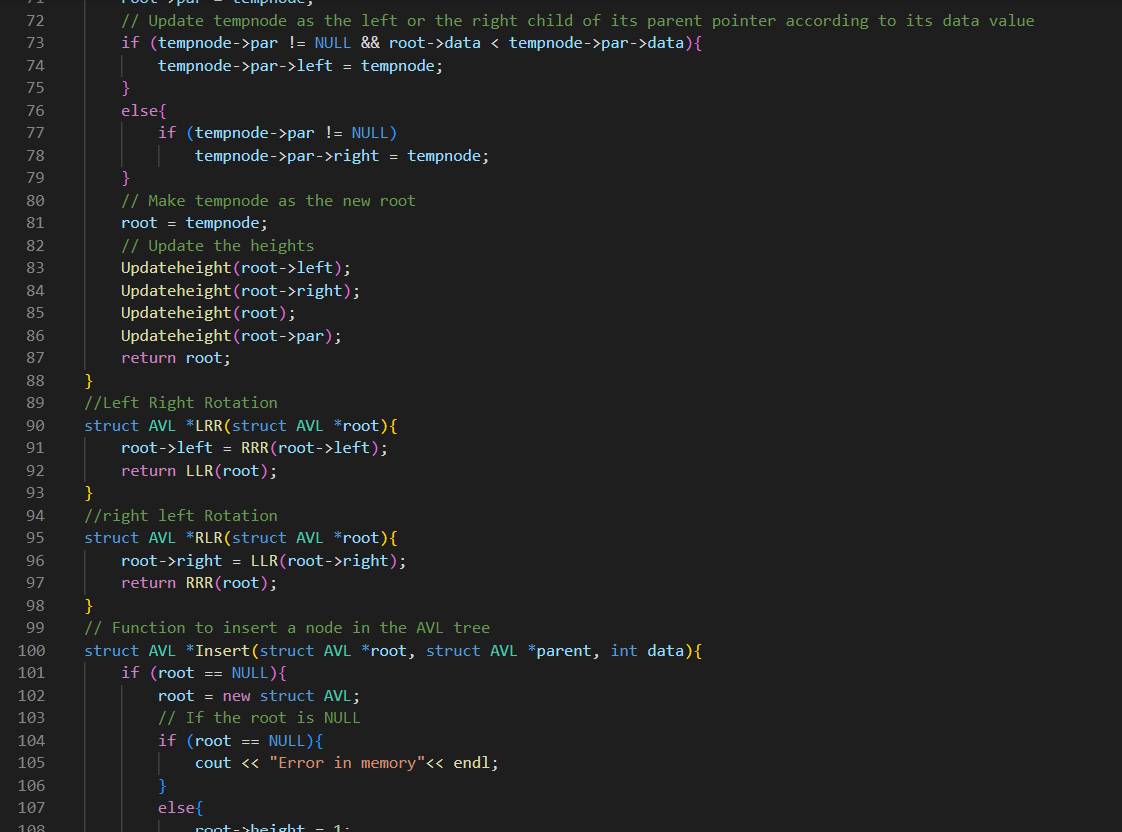
OUTPUT:



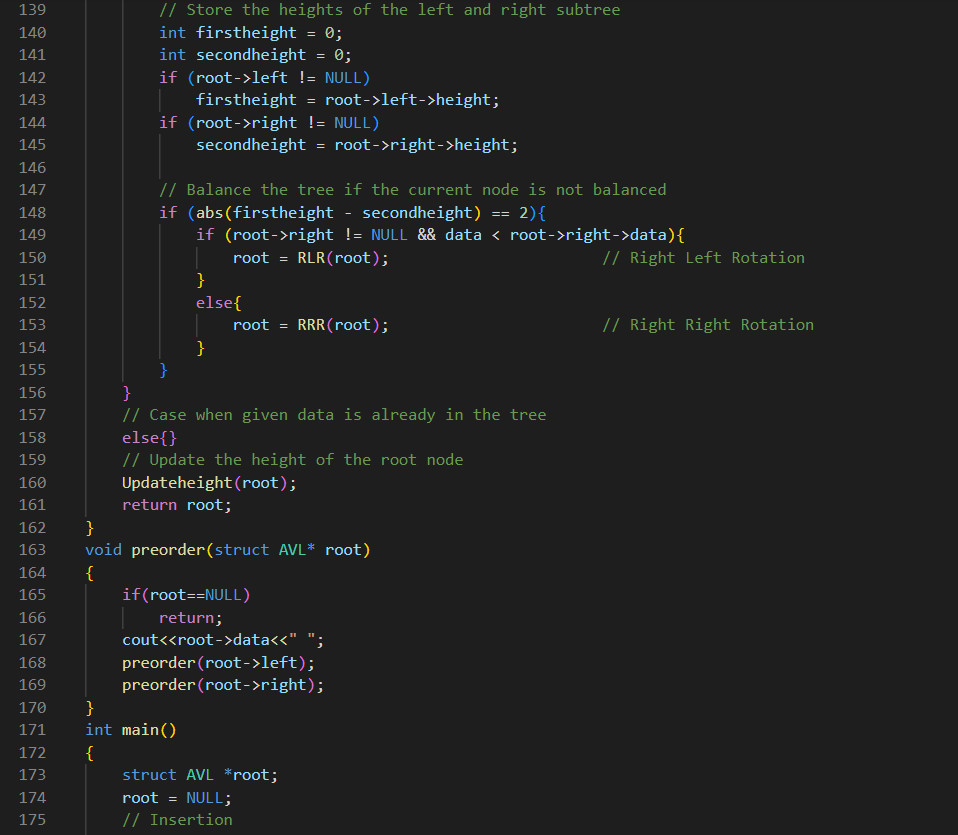
AVL TREE:

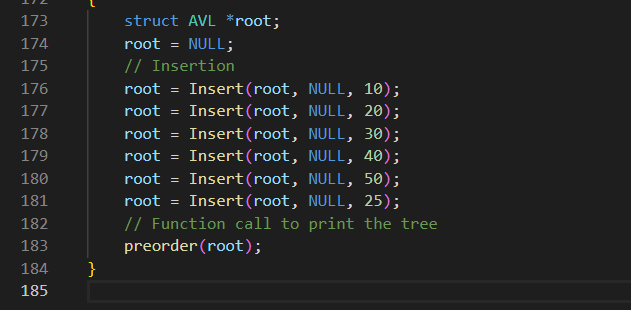




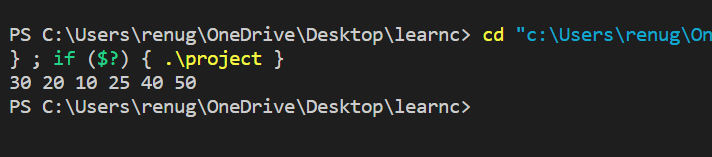








**OUTPUT:**

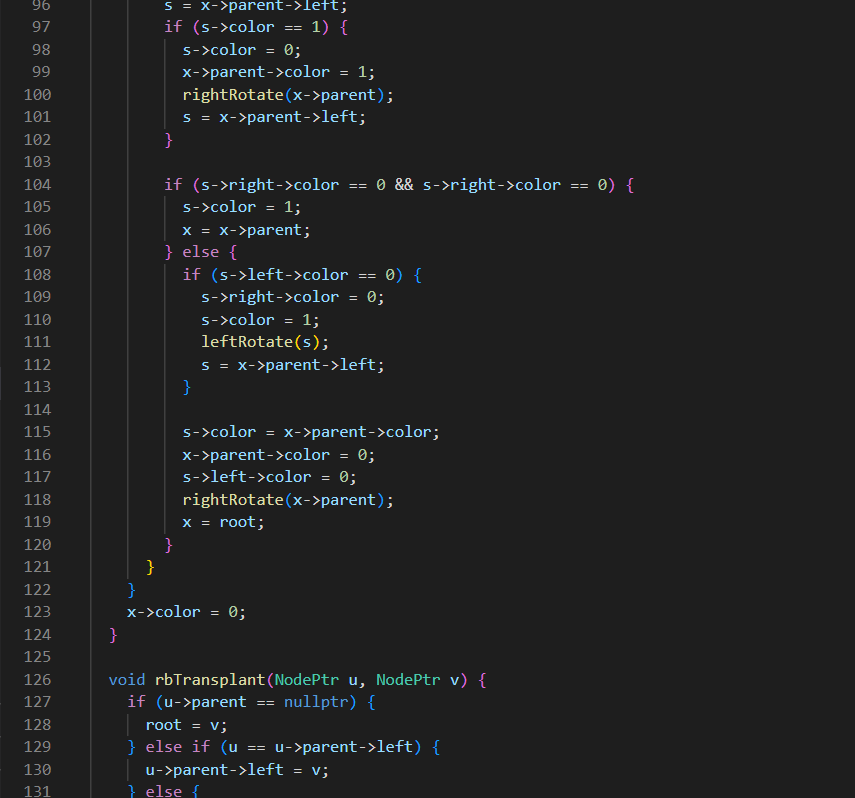


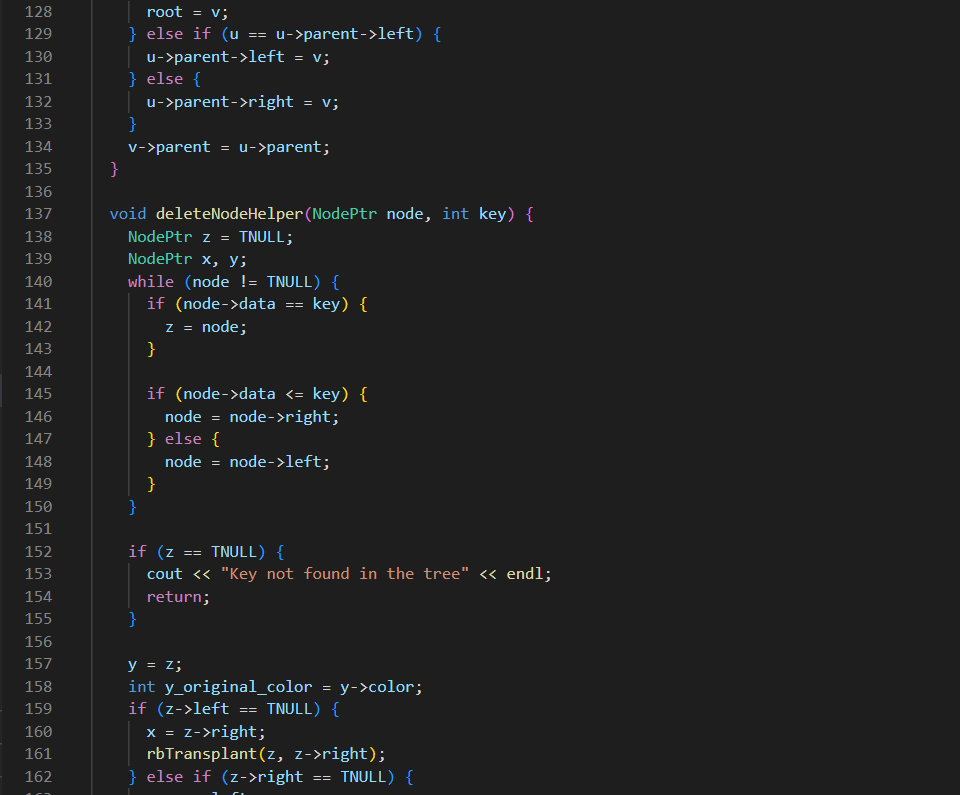
**RED BLACK TREE:**

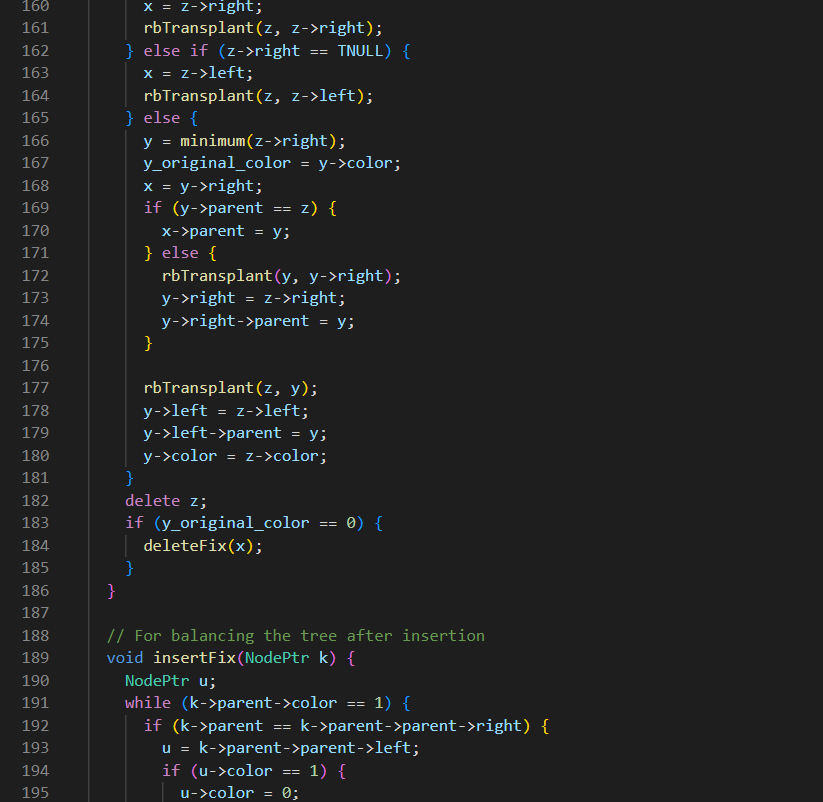








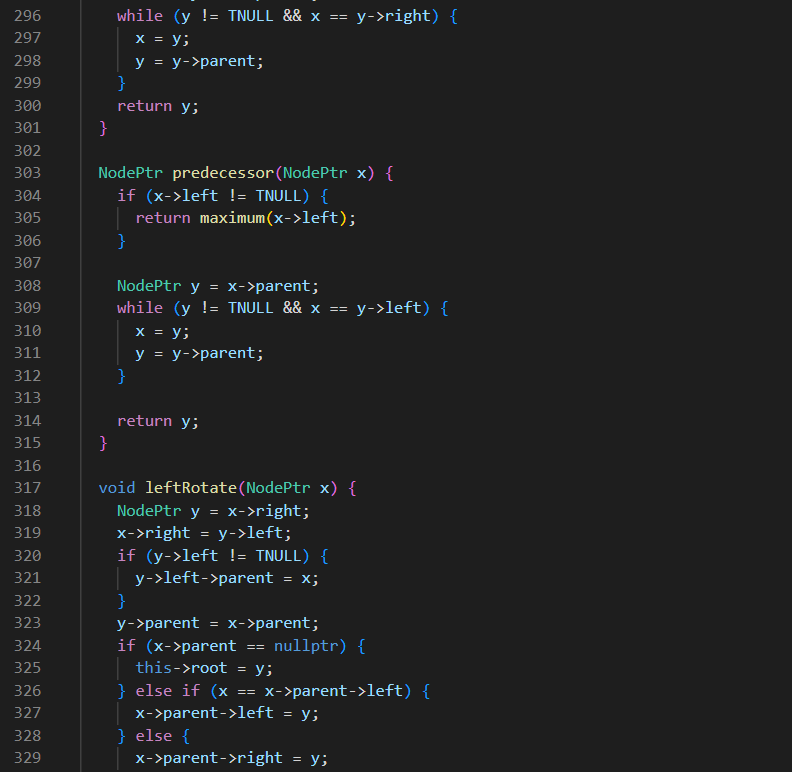




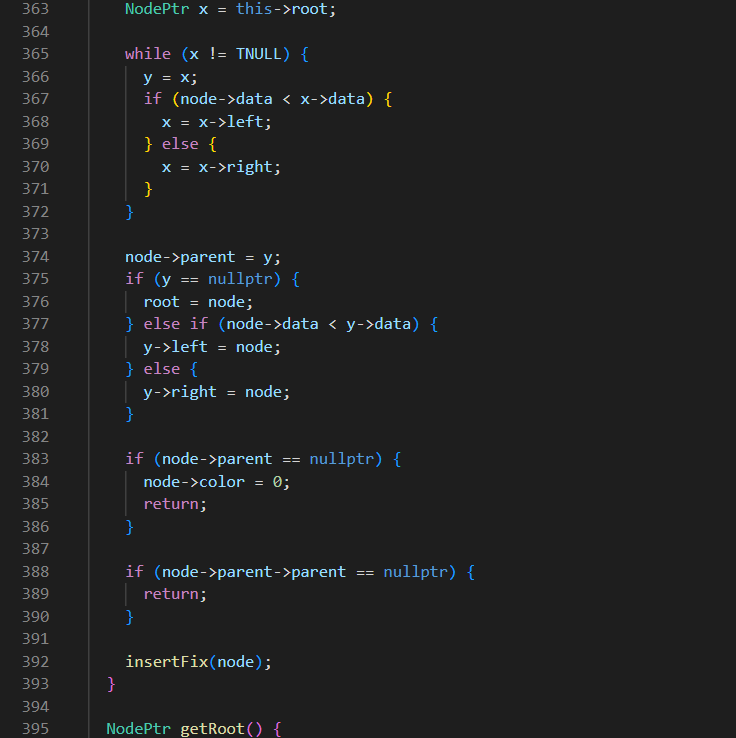


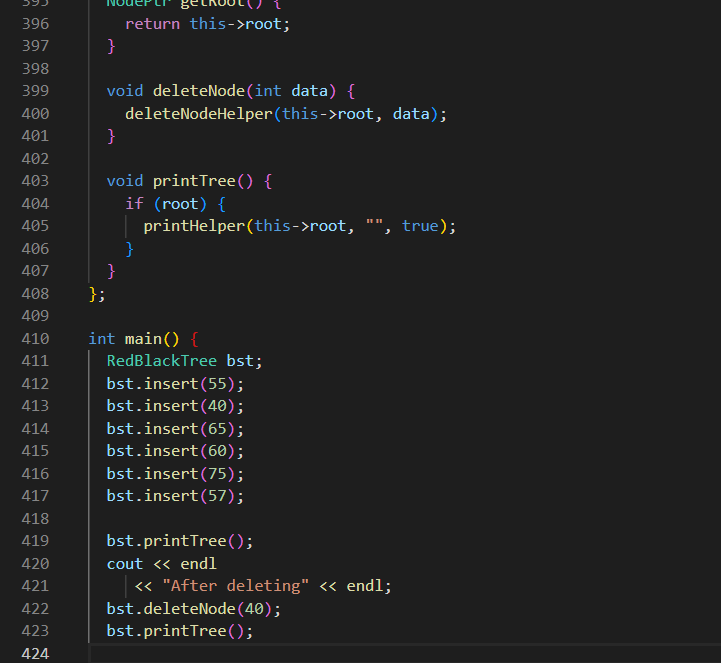




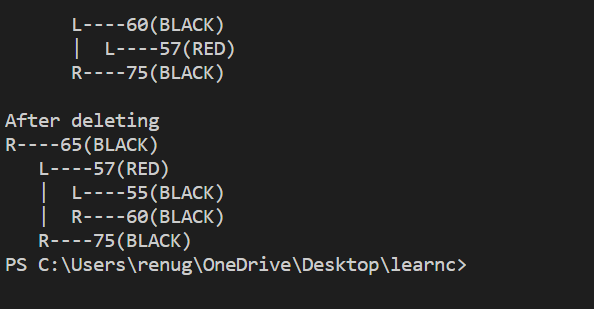








OUTPUT:



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**

It is a binary tree in which left subtree is always less than the root node element and right subtree is always grater than the root element.time complexity of insertion- O(H)

**AVL TREE :—**AVL tree is a binary search tree in which height of every node is always either 0,1 or -1. time complexity of insertion- O(logn)

We insert the data in AVL tree same as we do it in a BST but here when any node get height ather than 0,1 or -1 we do appropriate rotations to balance the height like

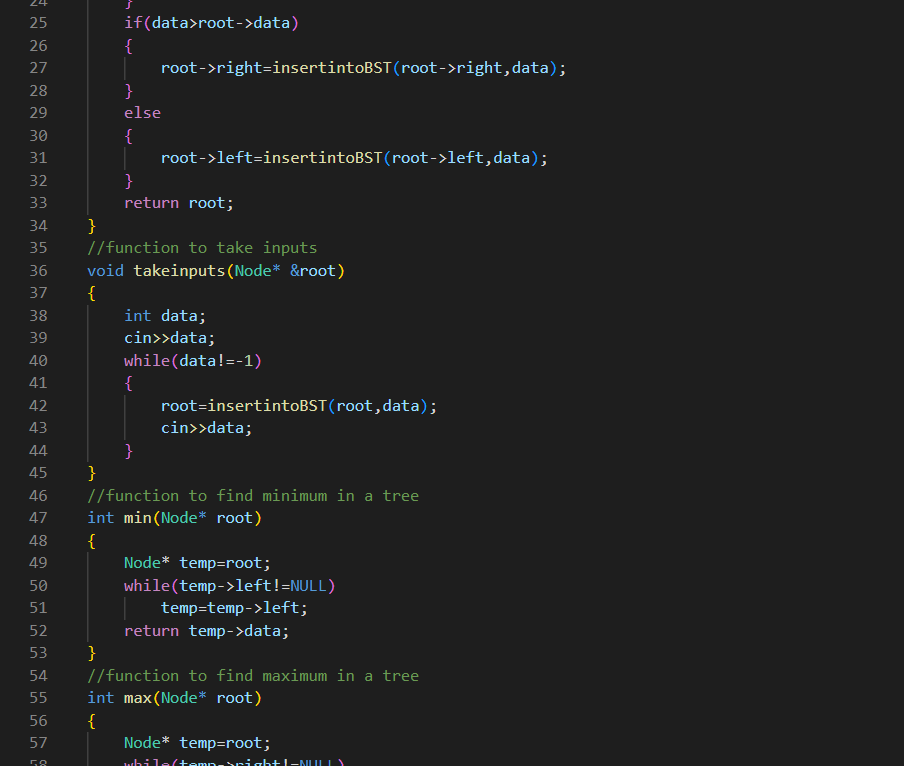
* Ll rotation
* Rr rotation
* Rl rotation
* Ir rotaion

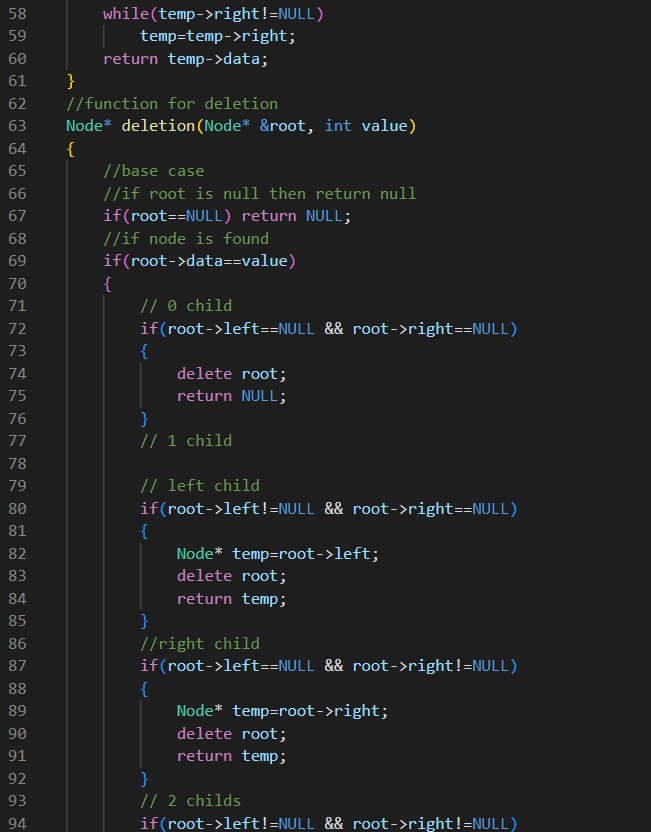
**RED- BLACK TREE :—** it is a kind of avl tree in which each properties of red black tree are as follows;

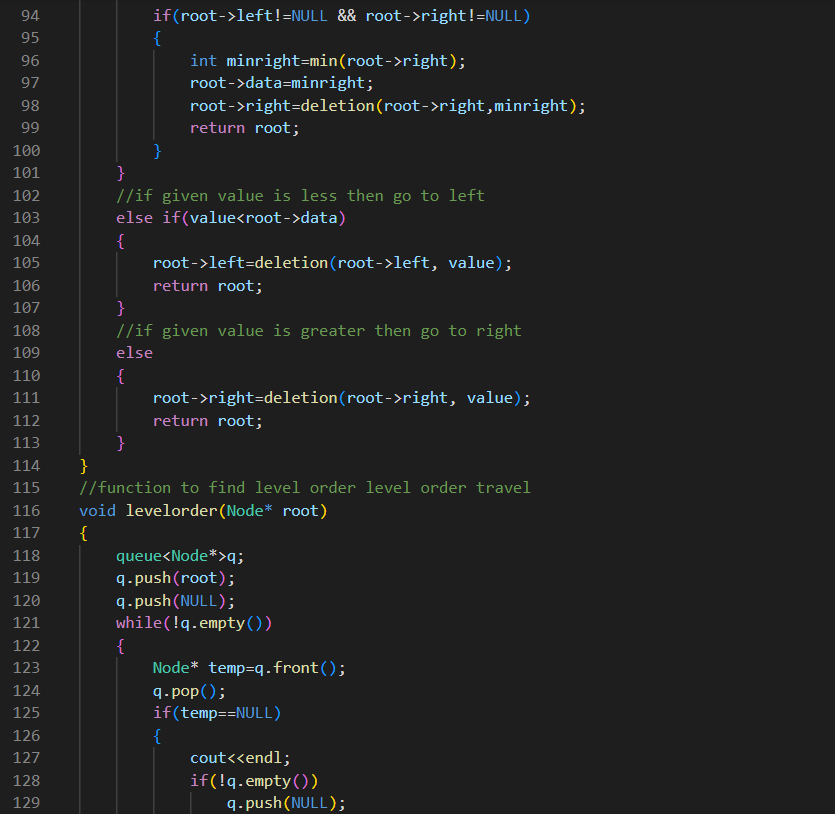
1. Root node is always black.
2. Two adjacent node cannot be red
3. Leaf node i.e. nil should be black
4. For each path to leaf node there must be same no. of black node  
   **CODE:-**

**Binary search tree:**



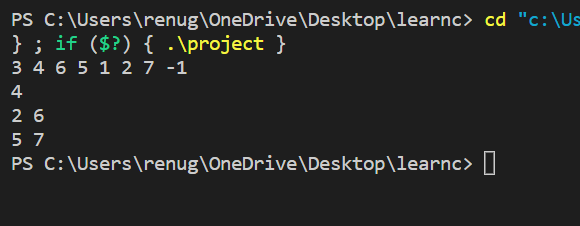




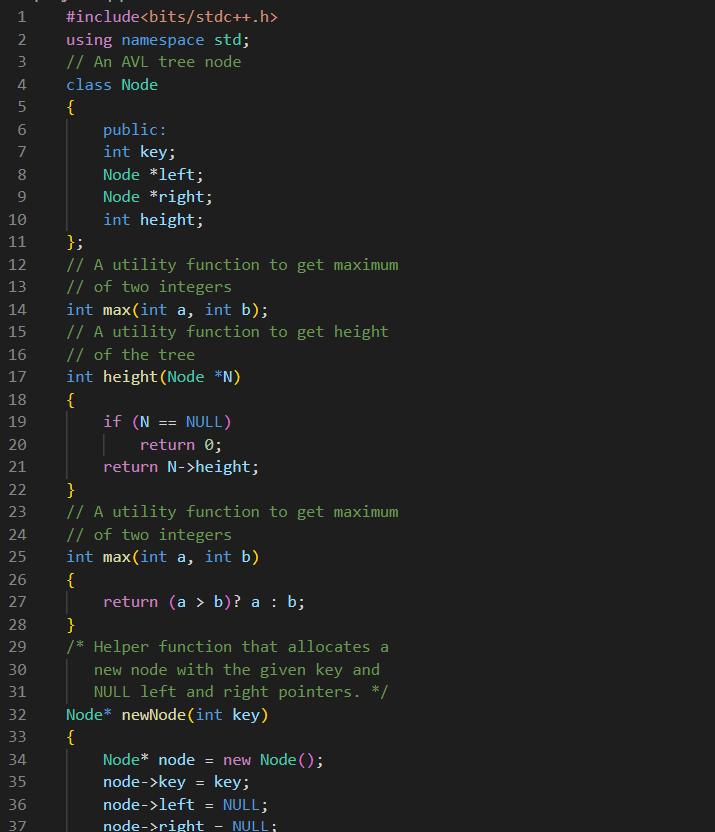


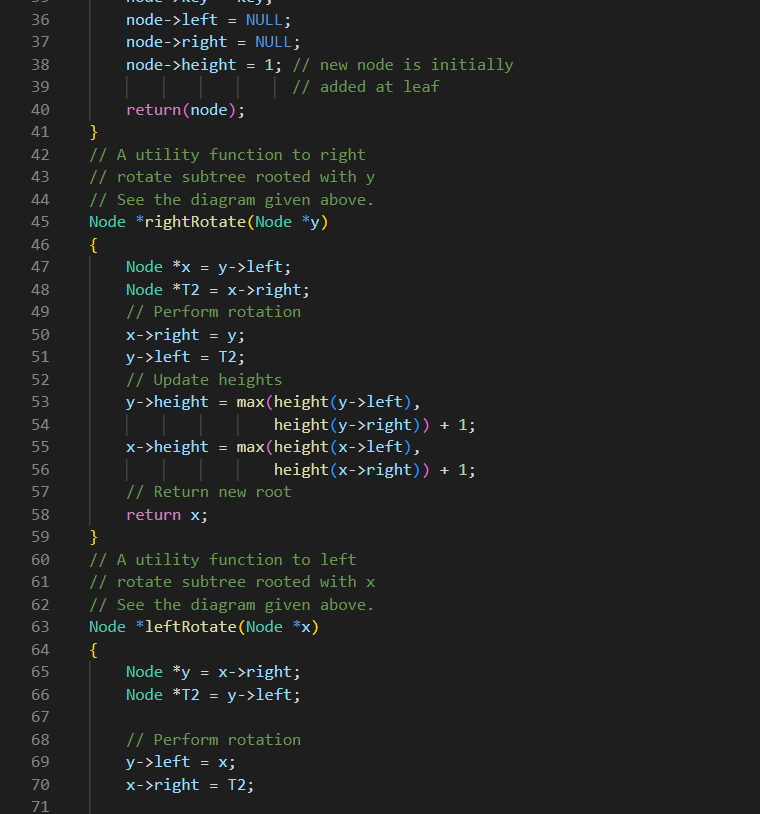


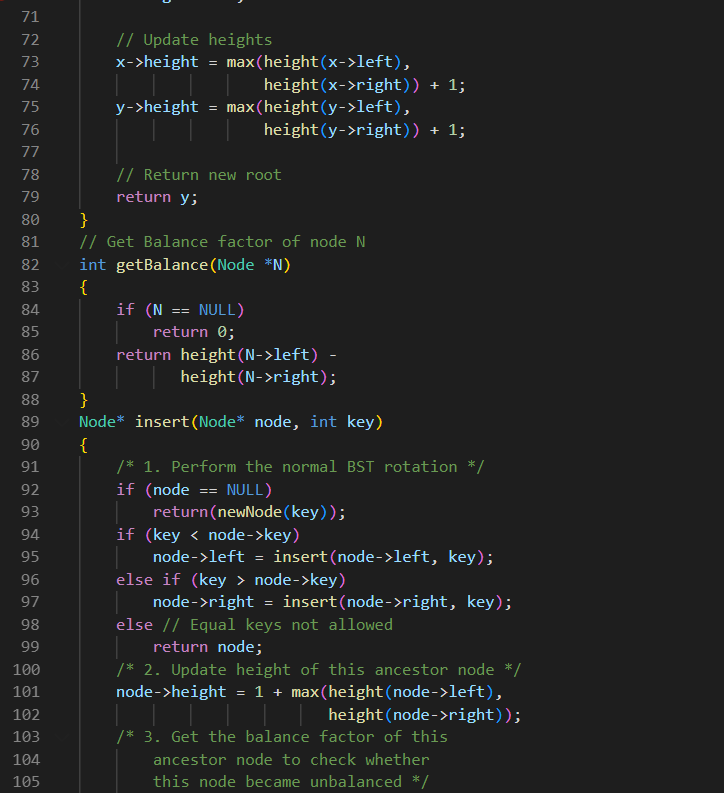
OUTPUT:

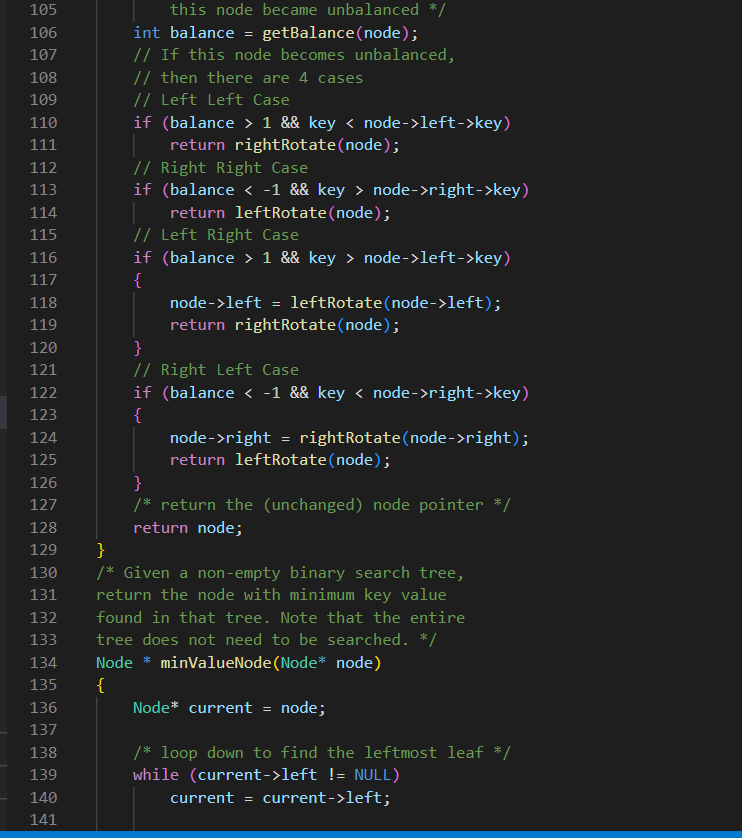


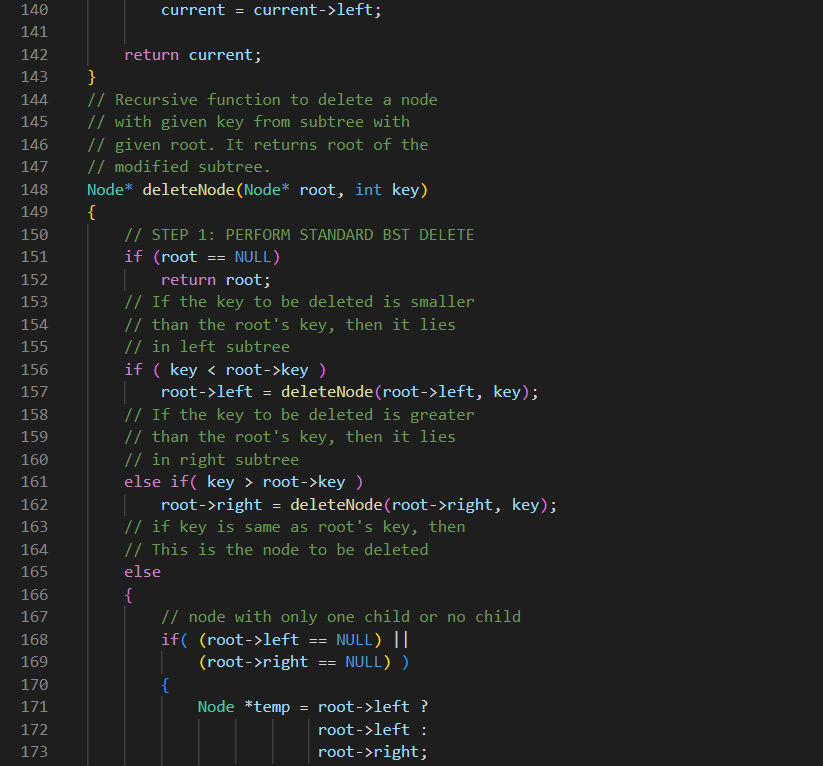
**AVL TREE:**

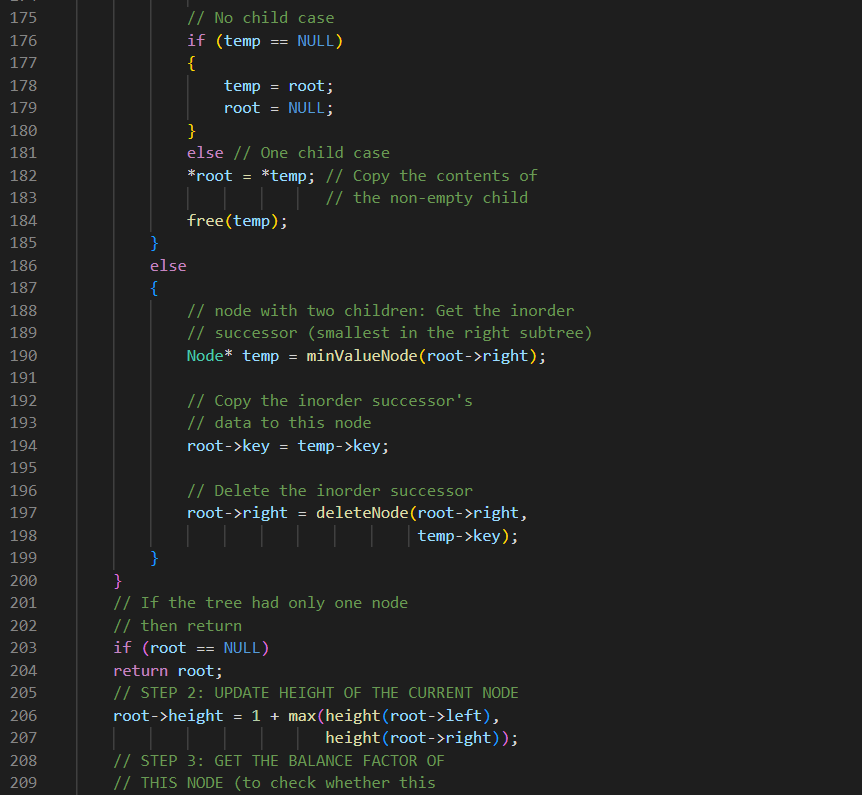




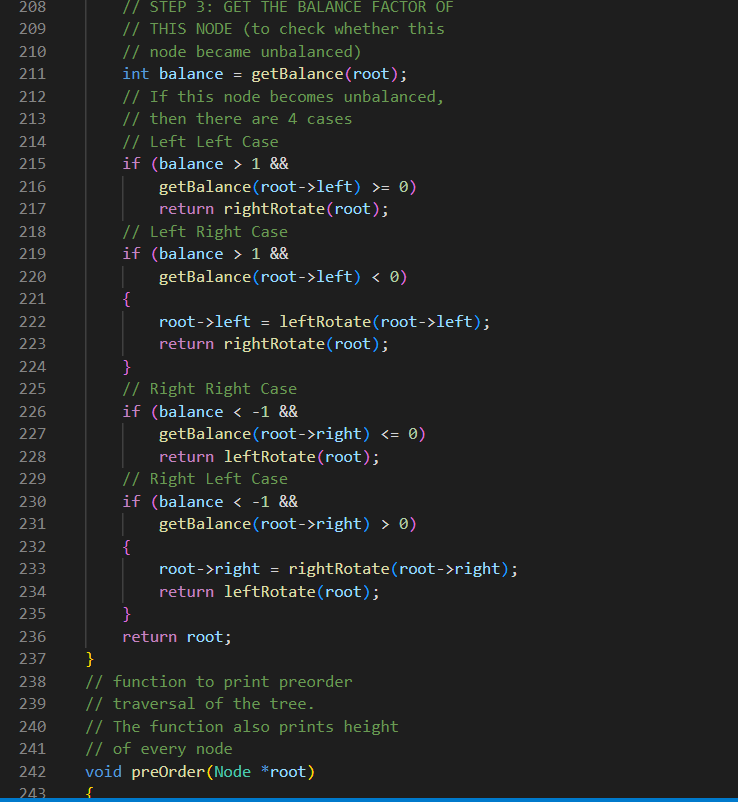


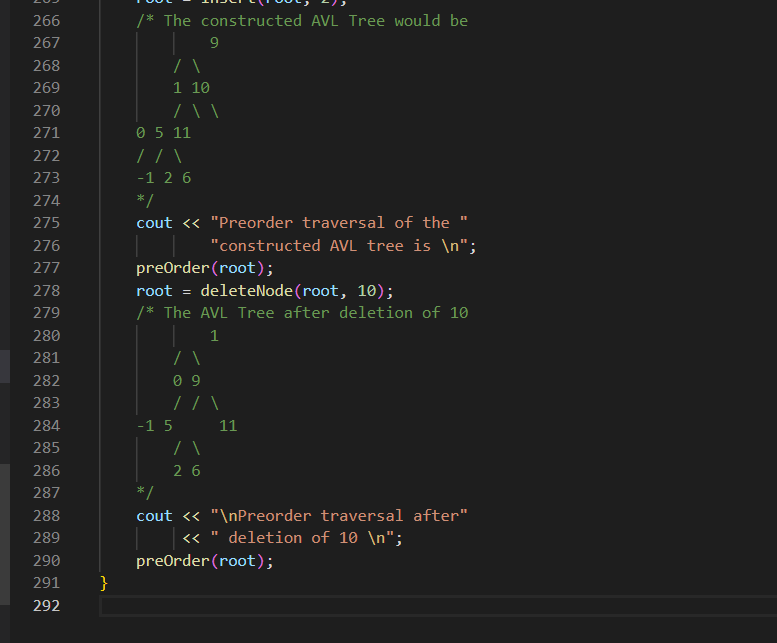




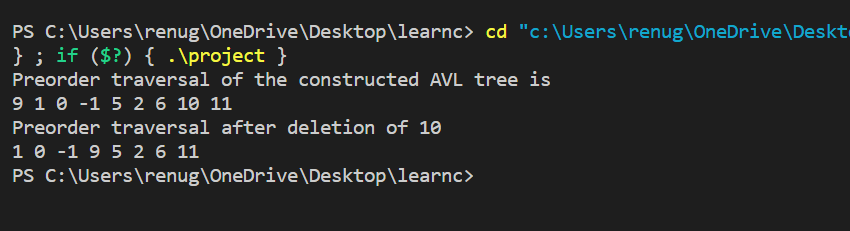








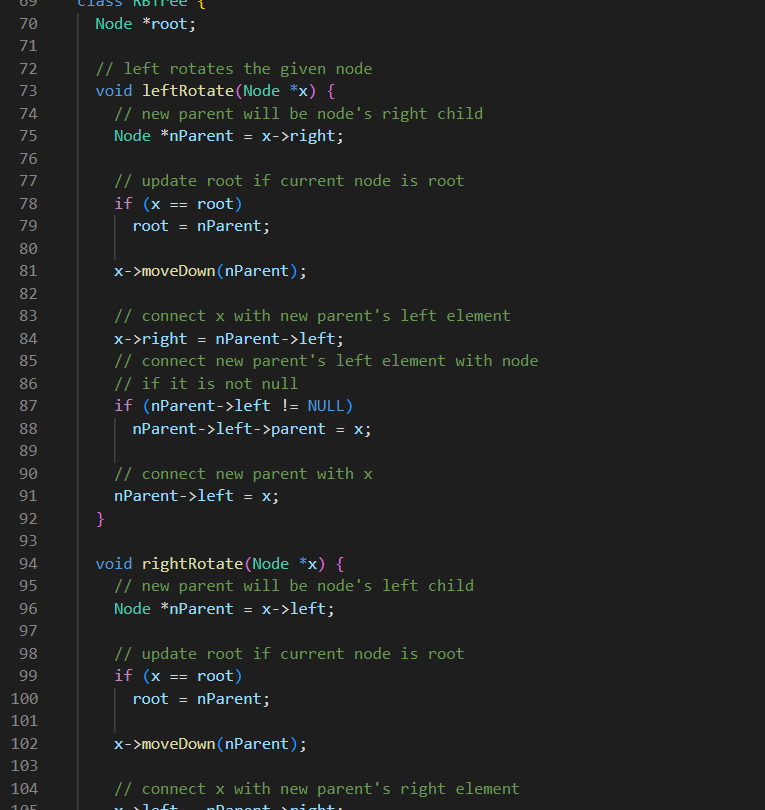
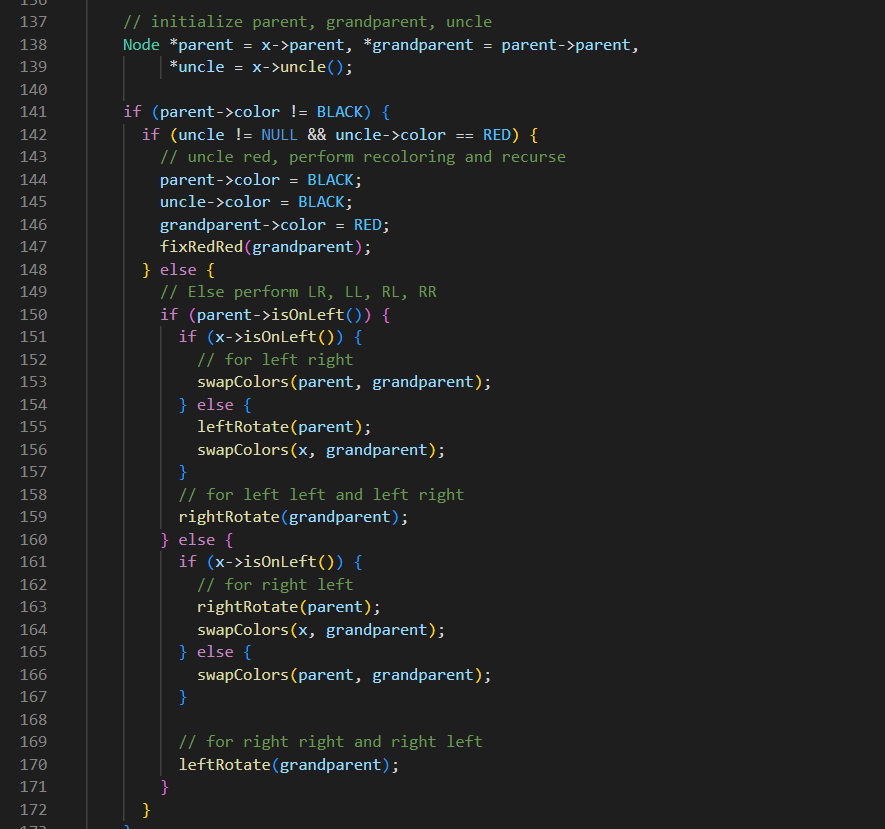
**OUTPUT:**

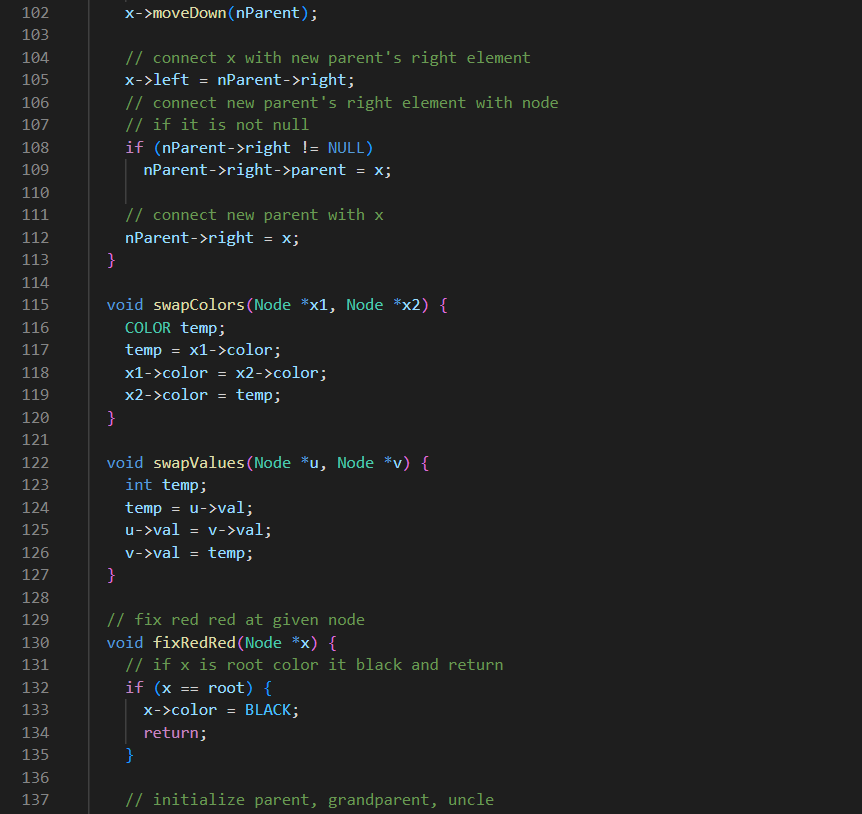


RED BLACK TREE:

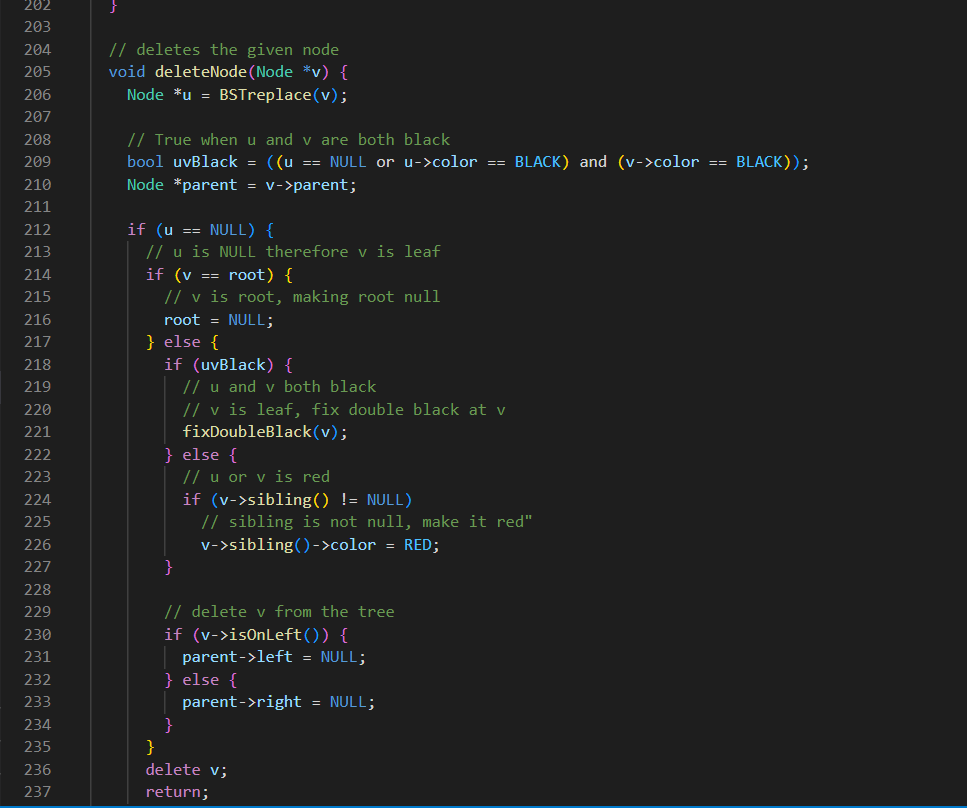


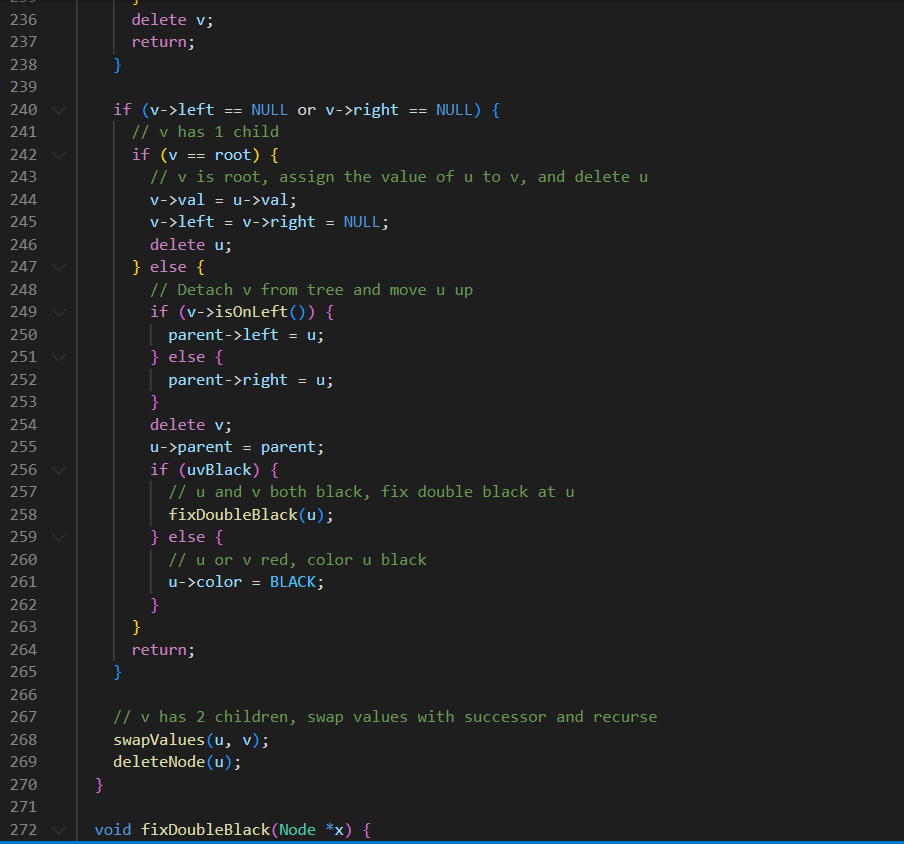


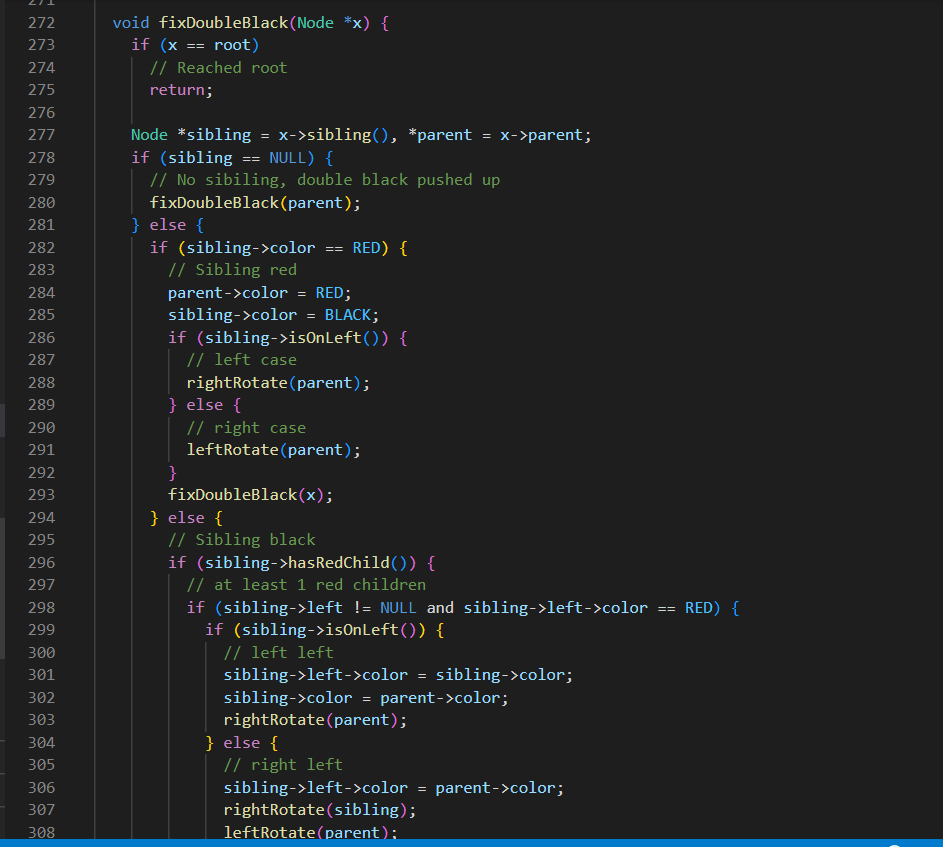


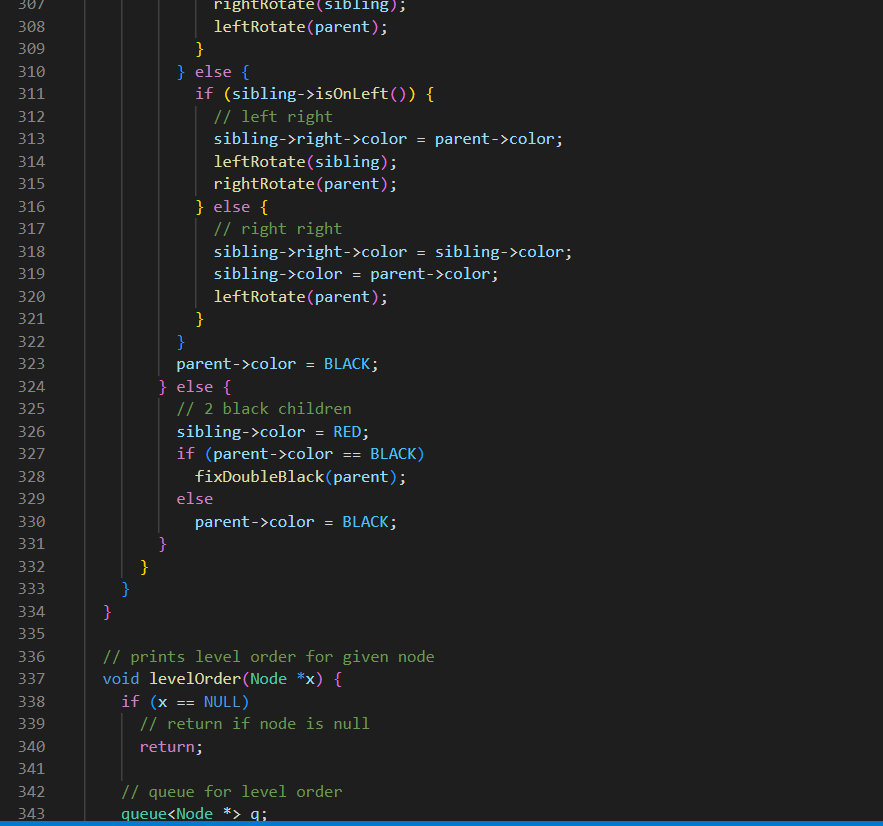




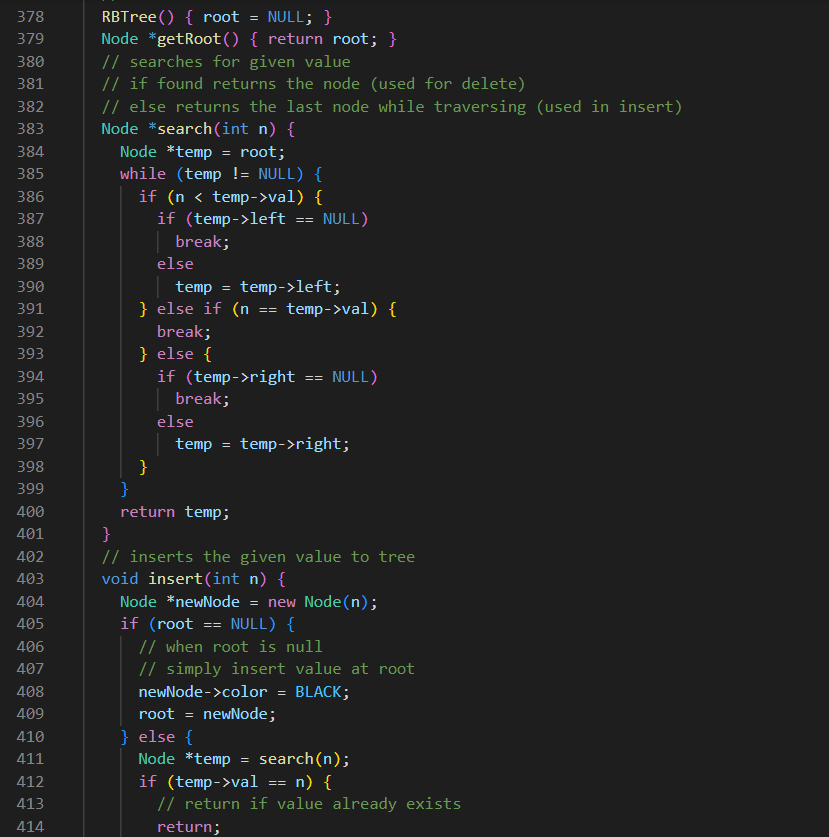






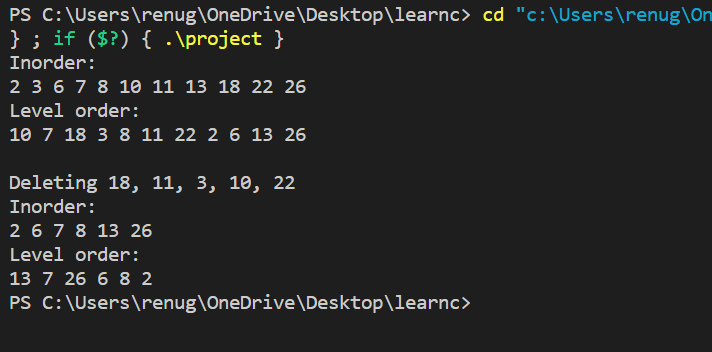








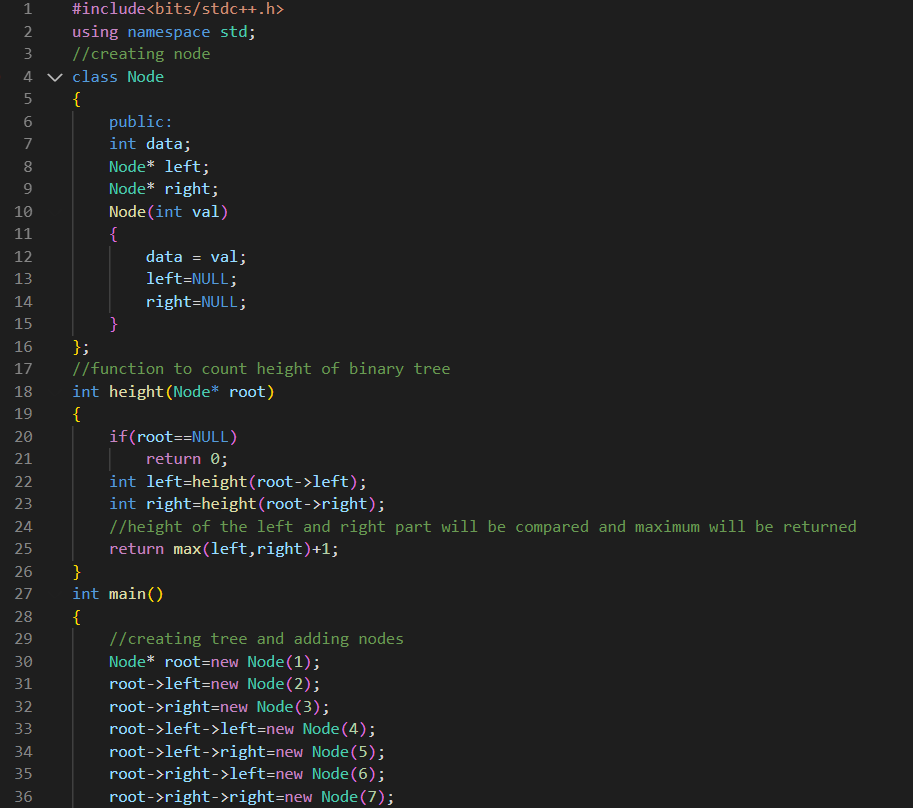
OUTPUT:

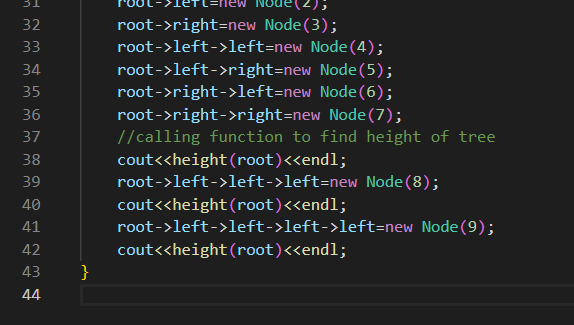


Experiment No. 7

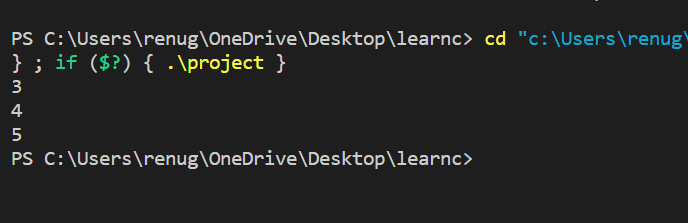
Objective:- Given the root of a binary tree, return the maximum height of the tree.

Code:-





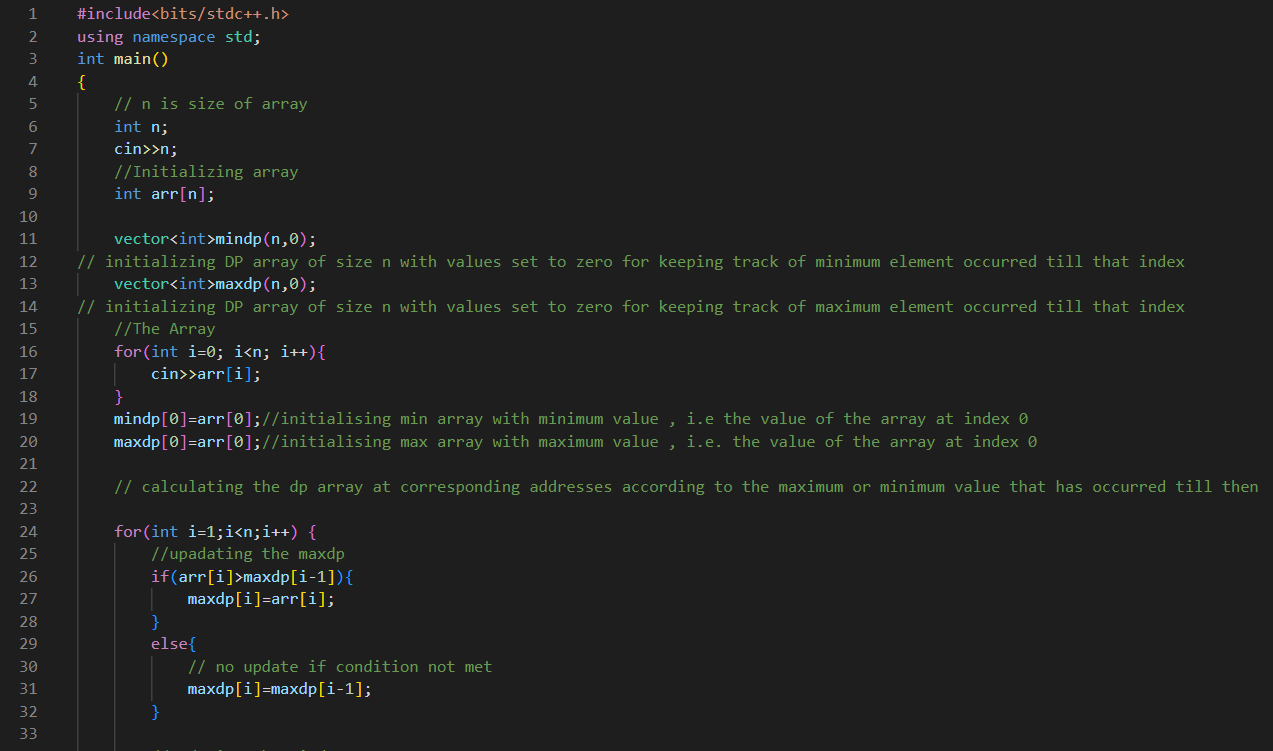
**OUTPUT:**

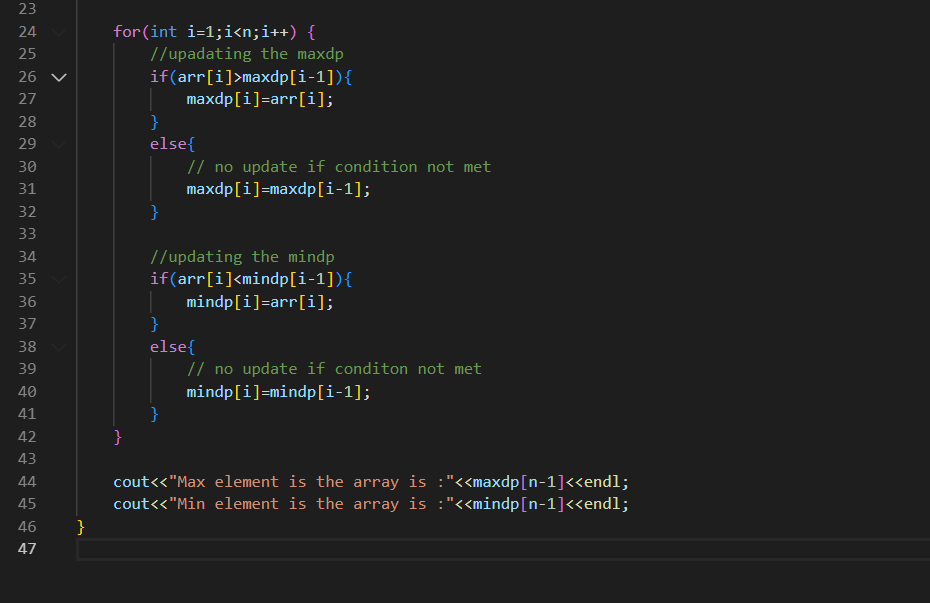


Experiment No. 8

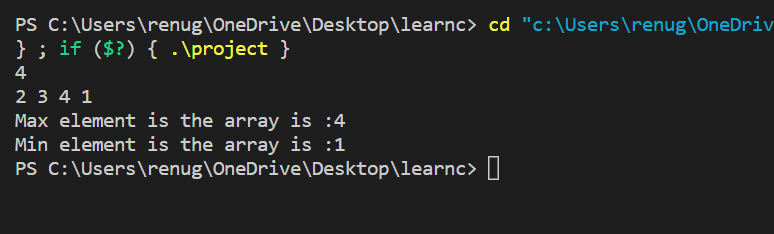
Objective:- Find maximum and minimum of array using the dynamic programming.

Code:-





**OUTPUT:**



Experiment No. 9

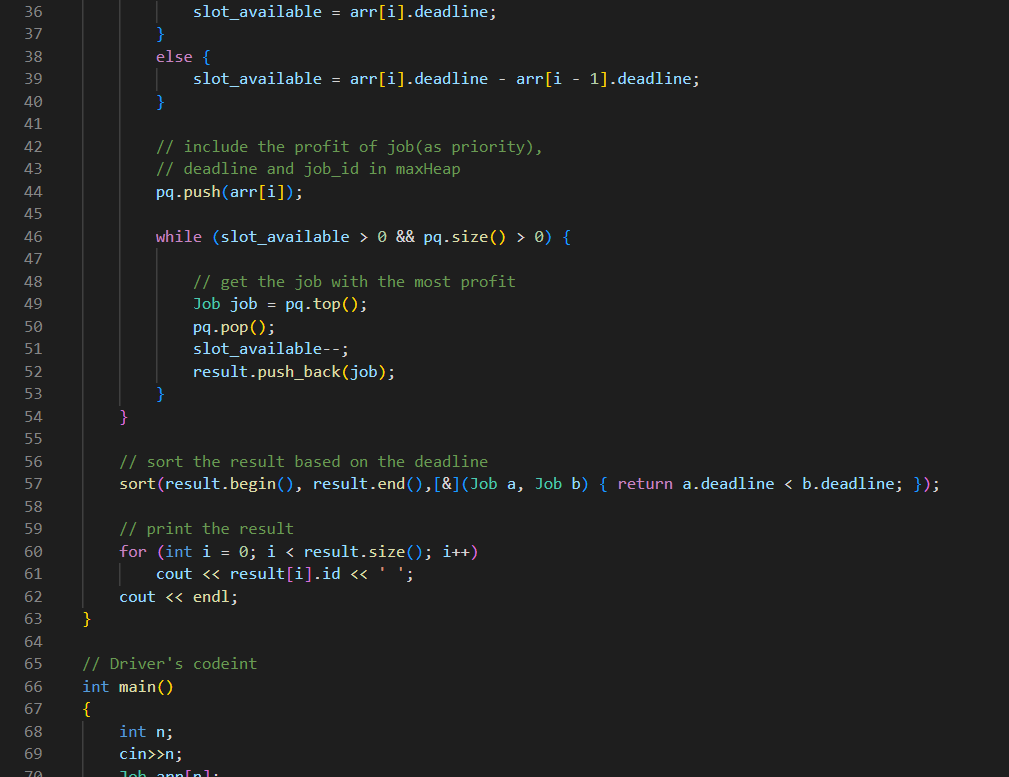
Objective:- Given a set of N jobs where each job i has a deadline and profit associated with it. Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit associated with the job if and only if the job is completed by its deadline. Find the number of jobs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

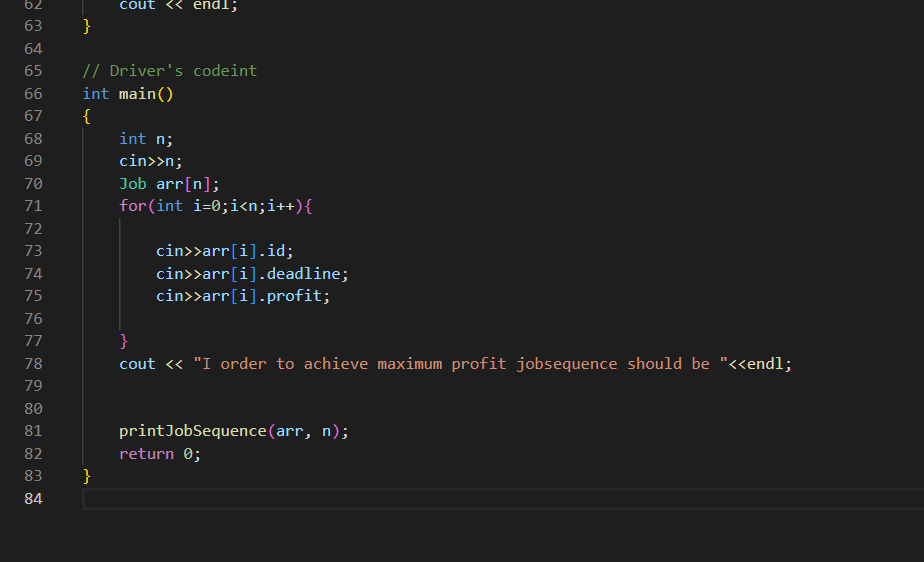
Theory:- This is a optimisation problem in which we are supposed to apply greedy approach i.e. we try to get optimal soluton in each step and the end result gives us global optimal solution.

Algorithm:-

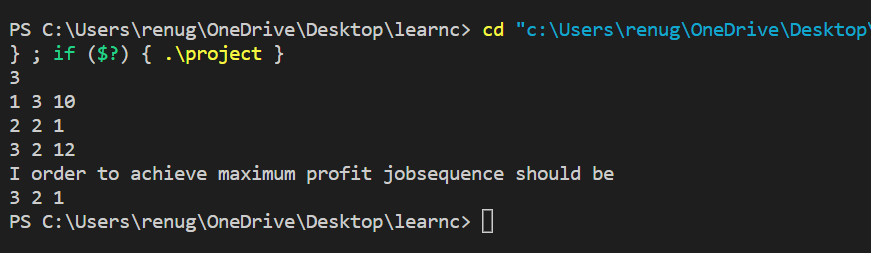
1) Sort all the jobs by decreasing order of profit.  
2) Initializing the result sequence as the first job in sorted the jobs.  
3) Do following for the remaining n-1 jobs.  
a) If the current job can fit in the current slot without missing the deadline, add the current job to the result. Else reject the current job.

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.

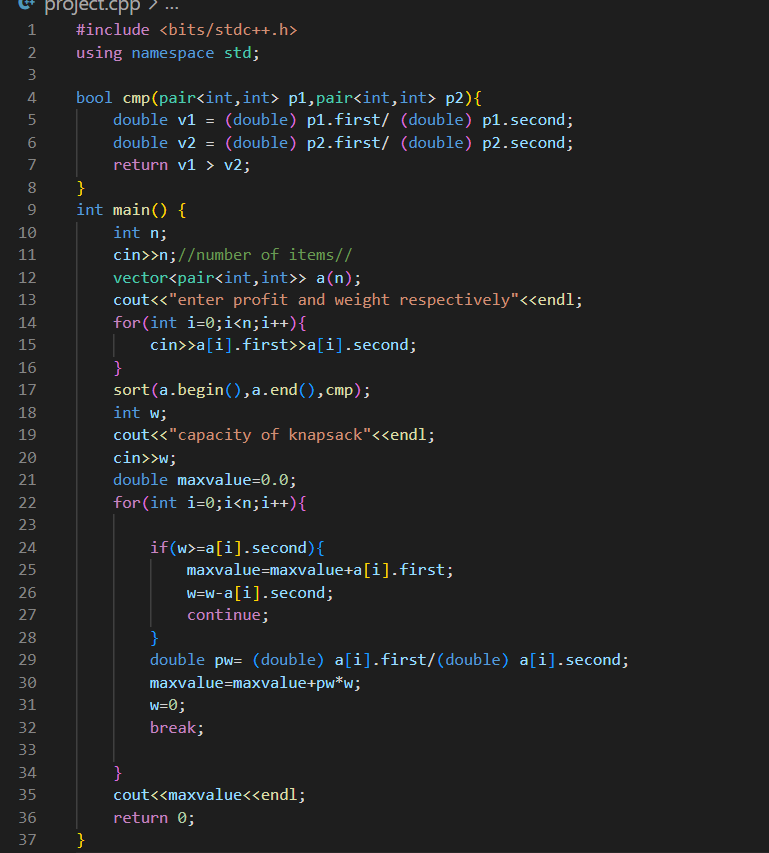
Theory:- The elementary idea of fractional knapsack is to sort elements on the bases of there profit/weight value and than

Fill the knapsack with the items having higher profit/weight value till the maximum capacity is reached.

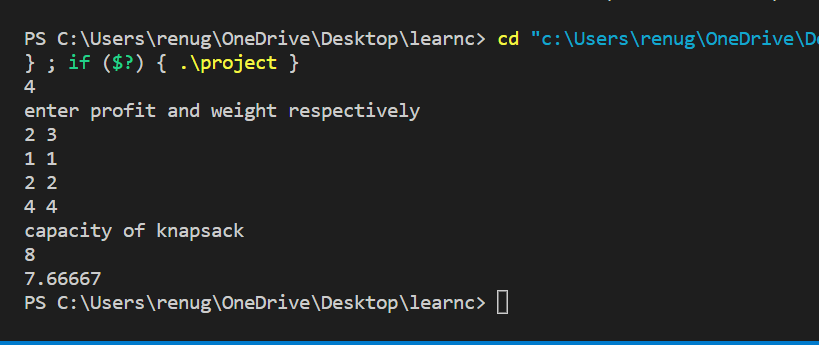
Algorithm:-

1. First sort the elements on the bases of their profit/weight value.
2. Take first element if its weight is less than total capacity of knapsack include the element fully and decrease the total capacity of knapsack by the weight of that item.
3. Else include the item partially and set the total capacity to be 0.
4. Repeat this process n-1 times.

Code:-



**OUTPUT:**



Experiment no 11

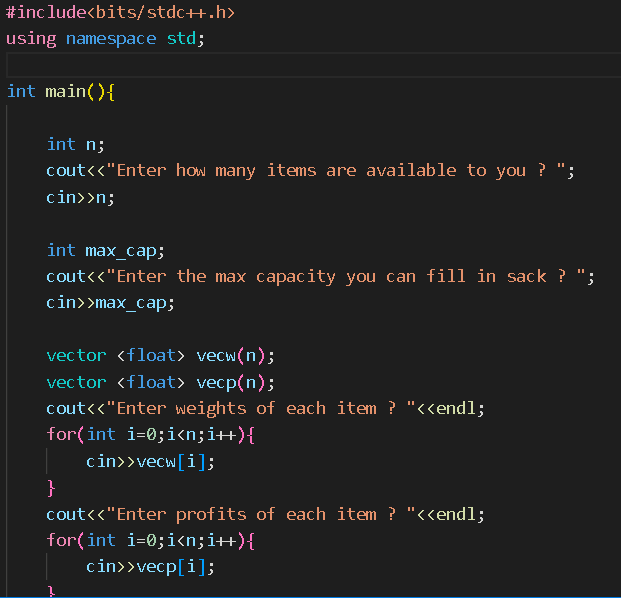
Fractional knapsack

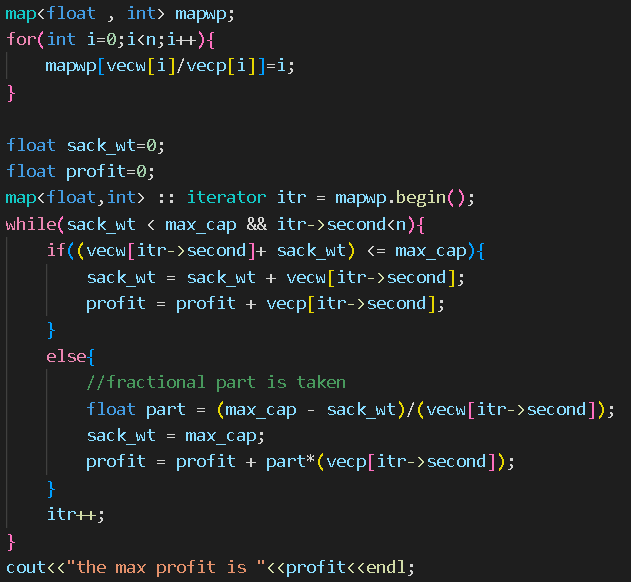
Algo:

1.sort in decreasing order of profit by weight

2. if sack is not full fill it fully or partially accordingly.

SOURCE CODE:





**OUTPUT**



Part 2

Objective : 0-1 knapsack

**Theory**: unlike fractional knapsack 0/1 knapsack is the the problem in which we are not allowed to contain any element fractionally.

**Algorithm**

for i=0 to i=n

for w=0 to w<=m

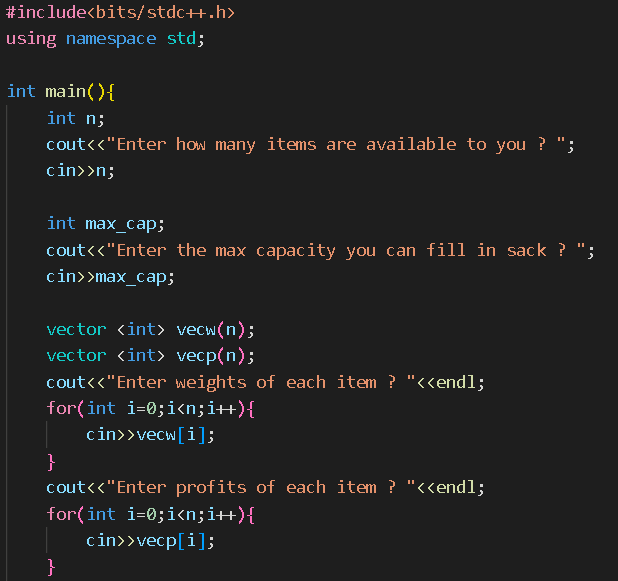
if(i==0||w==0) k[i][w]=0

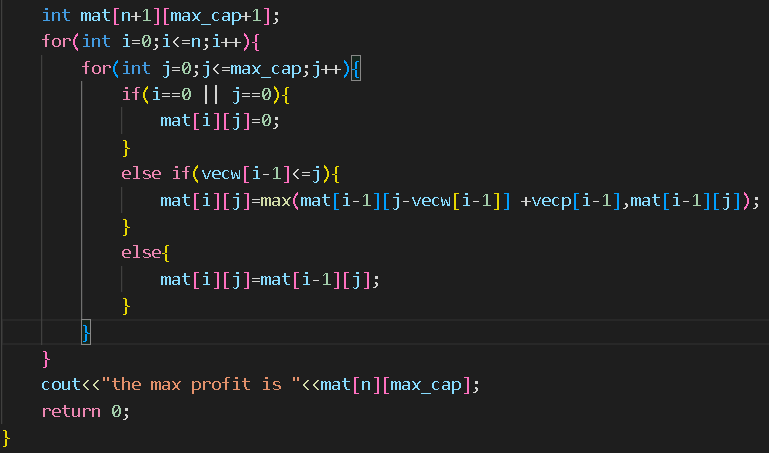
else if(w[i]<=w) k[i][w]=max(p[i]+k[i-1][w-w[i]],k[i-1][w]);

else k[i][w]=k[i-1][w];

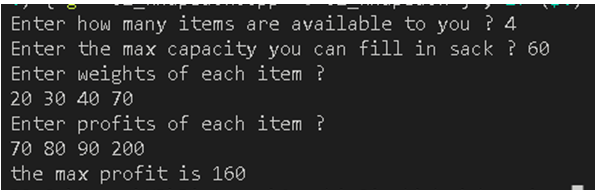
print k[n][w]

CODE:





OUTPUT:



Experiment No. 12

Part 1

Objective: to make a mst using prims algorithim

Algorithm:

* Mstset should be created that keeps track of included elements in mst
* Set all vertices to be infinite and starting node to be zero
* While mstSet doesn’t include all vertices
  + Pick u that ha s minimum value and not been included in mstSet and include it in mstSet.
  + Keep on updating all the nodes if its value is grater than u-v.

Part 3

Dfs traversal topological sort

Algorithm:

* Take any nod eas starting point
* Mark it as visited node now visite its neighbour if any
* Repeat the above step till null node is reached.
* Now go to previous node visit its neghbours as well

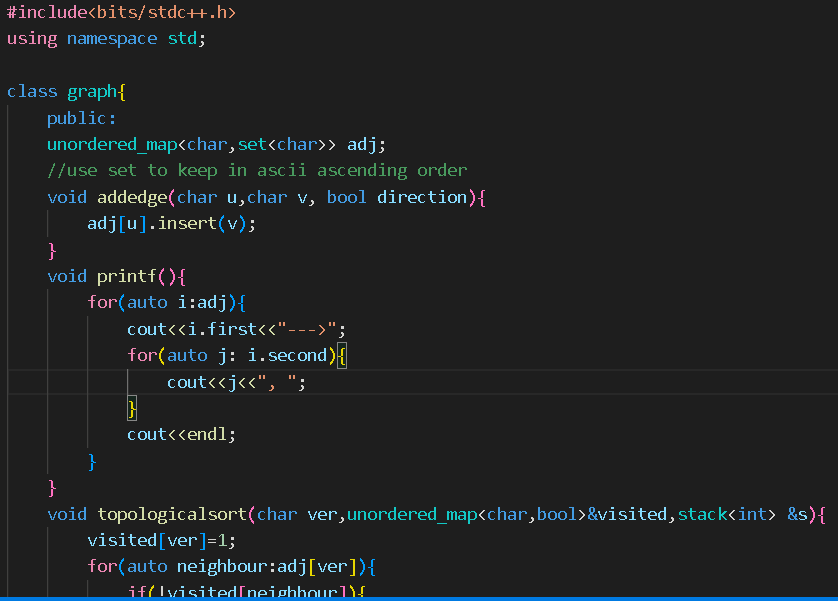
Time complexity analysis:

O(V+E)

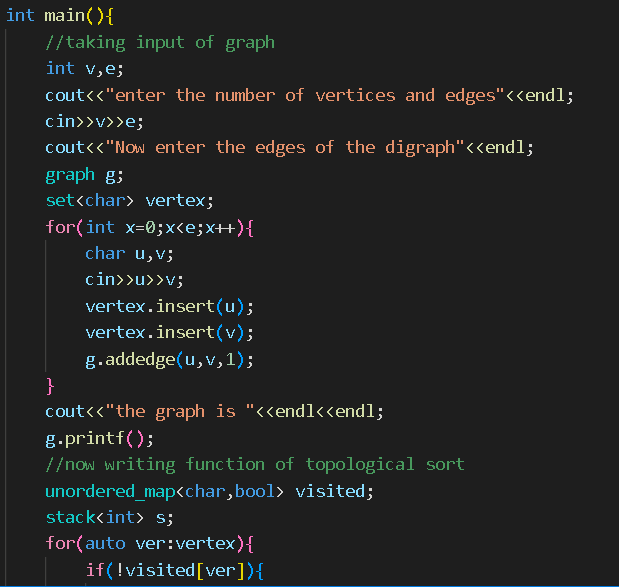
Space complexity analysis:

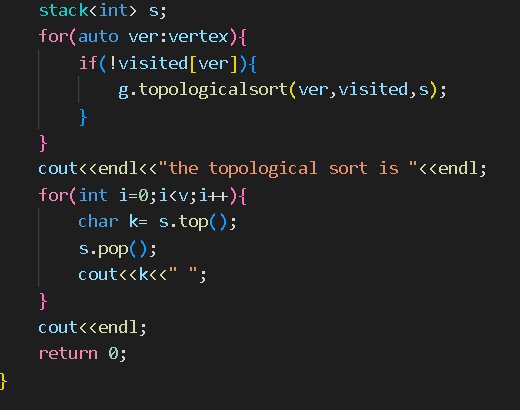
O(V+E)

**CODE:**









OUTPUT:



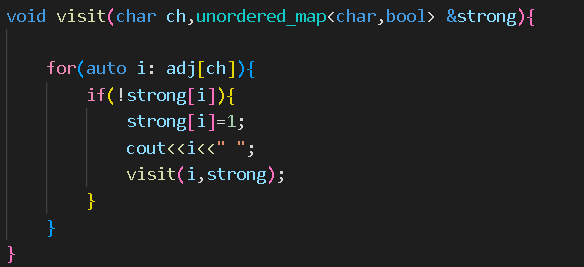
Part 4

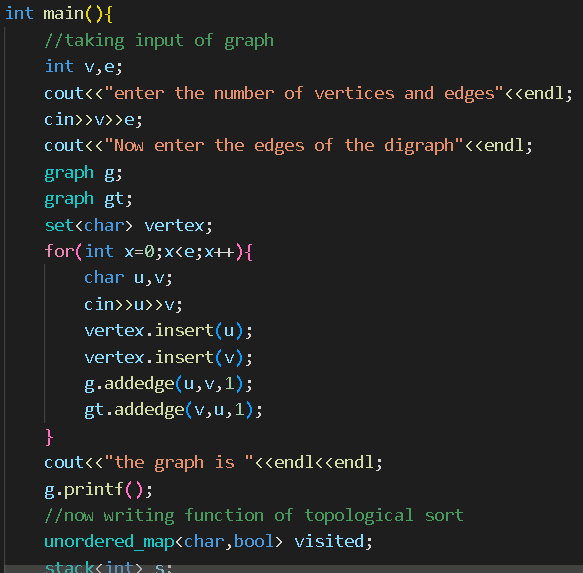
Objective :In a Directed Graph identify all strongly connected component.

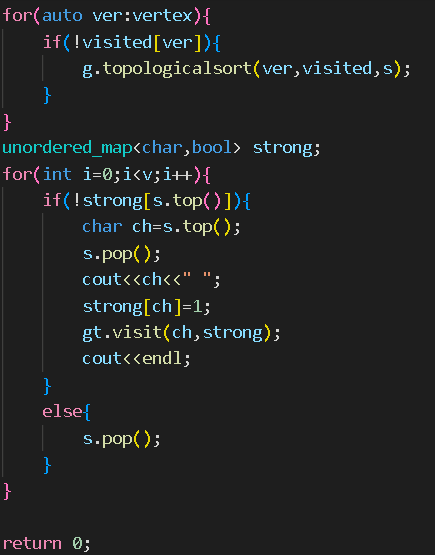
Algo

1. sort all the nodes on the bases of finishing time
2. Transpose the graph.

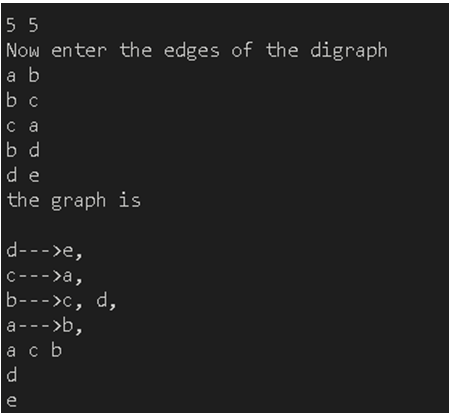
3.do the dfs traversa on the bases of finishing time and print.







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 matrices are given in an array arr[] of size N (such that N = number of matrices + 1) where the ith matrix has the dimensions (arr[i-1] x arr[i]).

**Theory**: these type of problems are called matrix chain multiplication

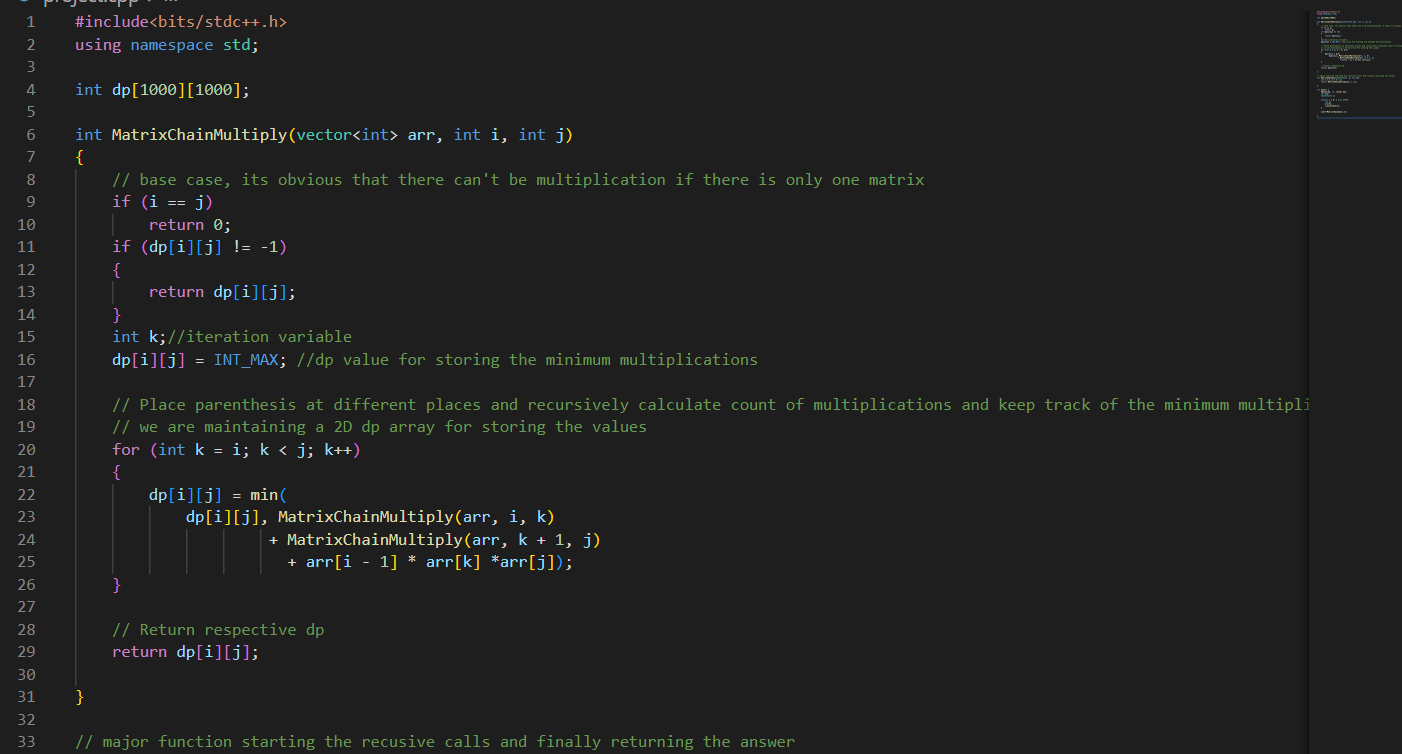
1.take parameters i and j to define the range.

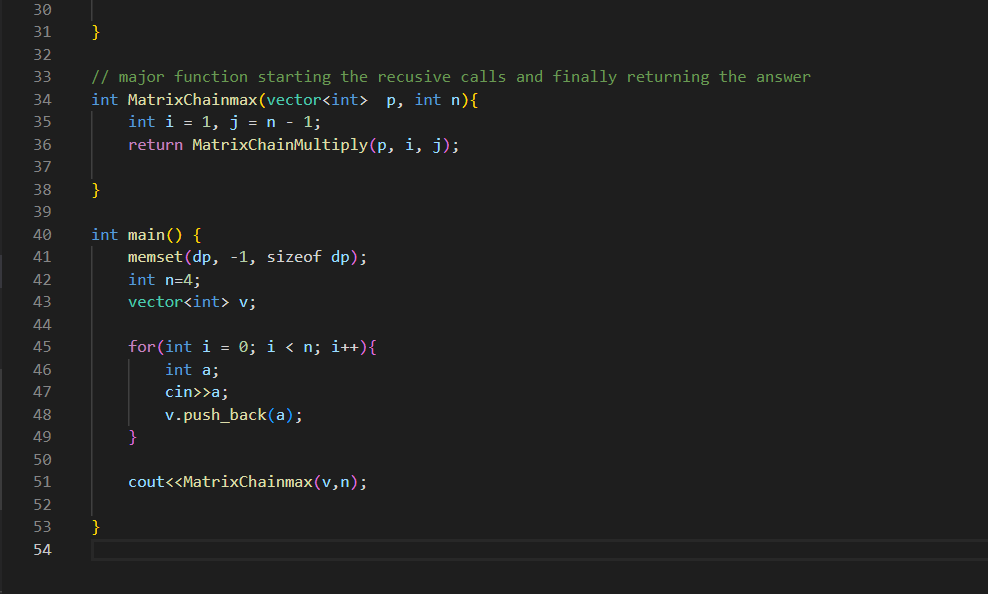
2.declare k=i to j .m[I,j]=min{m[i,j]+m[k+1]+di\*dk\*dj}

5.thre minimum calue fro 0 to n-1 is the ans

Time complexity =O(n\*n\*n)

Code:-





OUTPUT:



Experiment No. 14

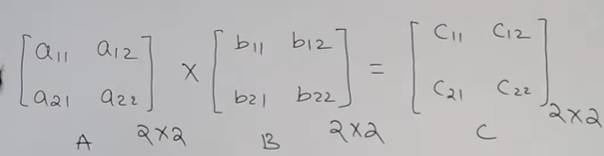
1. **Strassen’s Matrix Multiplication**

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

**Algorithm:-**

Strassen’s Matrix multiplication can be performed only on **square matrices** where **n** is a **power of 2**. Order of both of the matrices are **n × n**.

Divide **X**, **Y** and **Z** into four (n/2)×(n/2) matrices as represented below −



Using Strassen’s Algorithm compute the following −

Begin:

For i : 0 to n

For j : 0 to n

c[i][j]=0

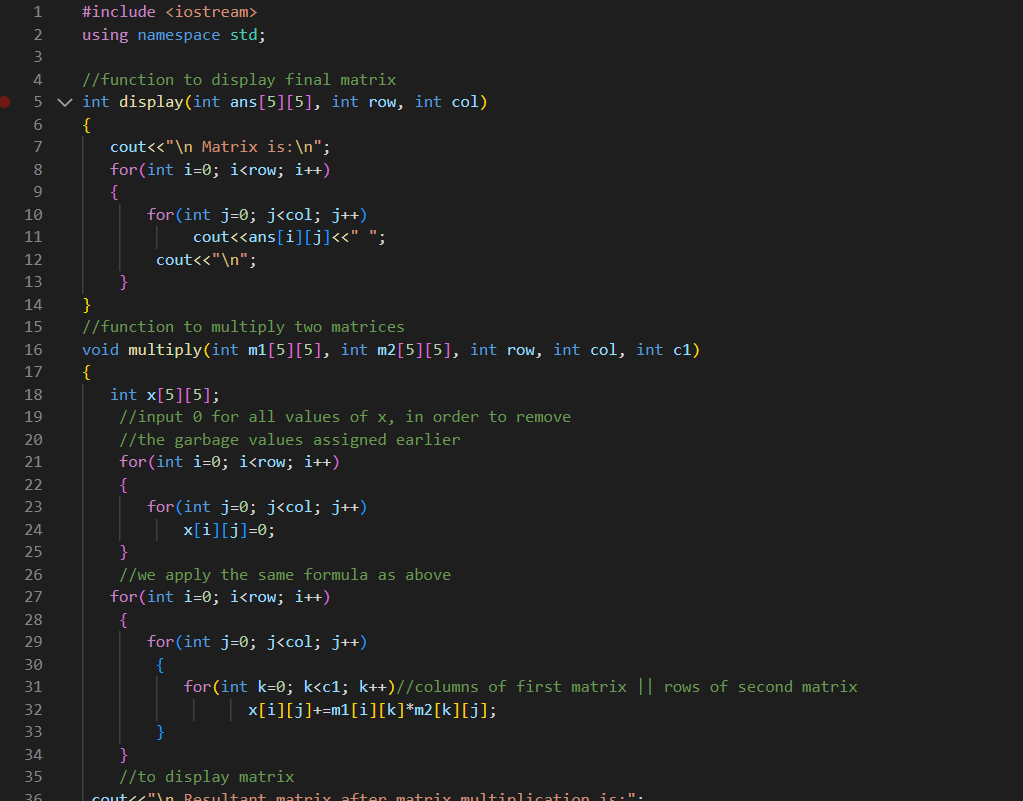
for k : 0 to n

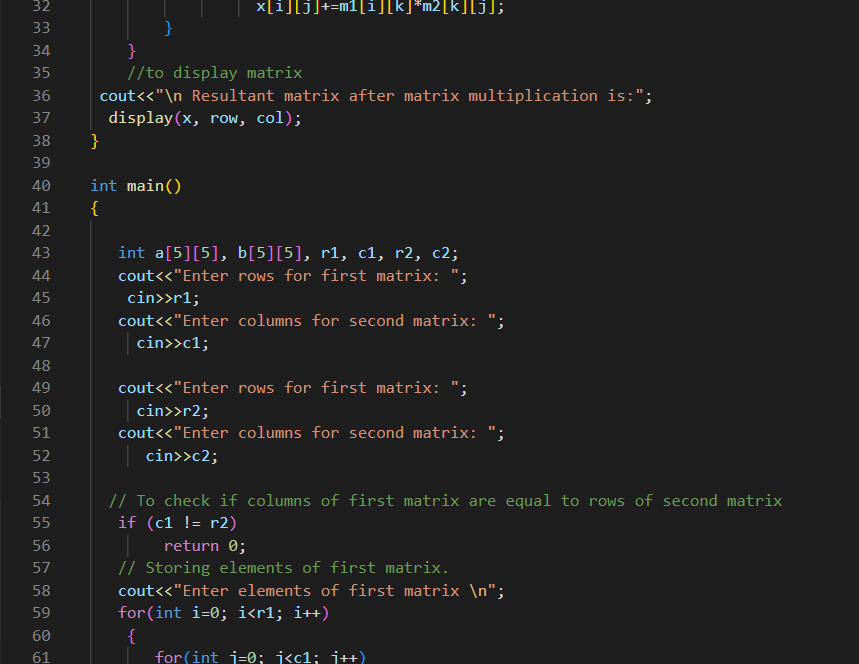
c[i][j]= c[i][j]+A[i][k]+ B[i][k]

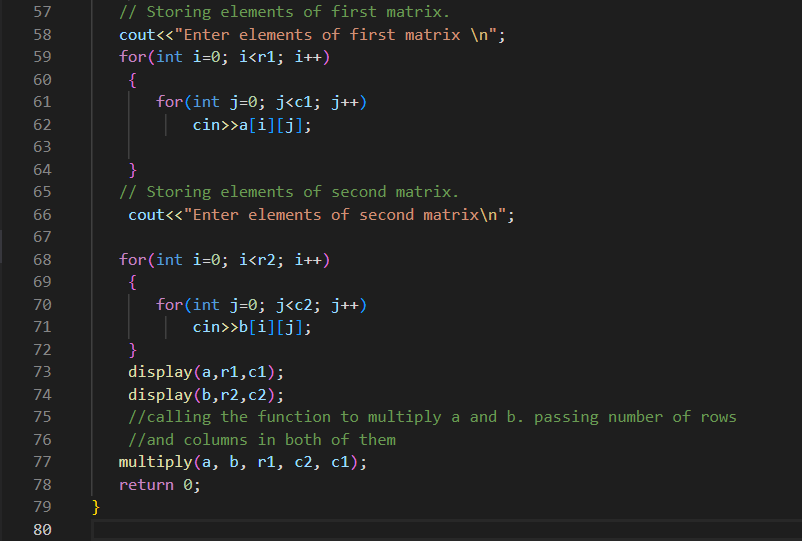
End:

Time Complexity:-  O(N2.8074).

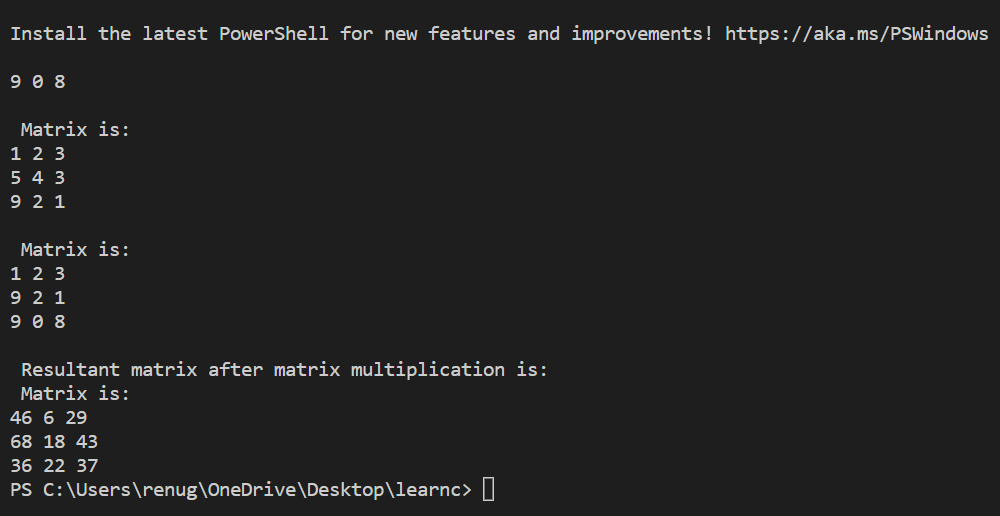
Code:-







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 (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that to find the most efficient way to multiply a given sequence of matrices. The problem is not actually to perform the multiplications but merely to decide the sequence of the matrix multiplications involved.

It is done by Dynamic Programming.

Algorithm: -

1. Put parenthesis in all feasible manner now calculate cost of every parenthesis.
2. Position parenthesis in n-1 ways .
3. Divide the problem in further sub problems.
4. The least number of placements are required to multiply

Time Complexity: - O(N3)

**Experiment No. 15**

Longest Common Subsequence

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

Theory: -

In this problrm we are required to find the longest common sequence of characters in two given strings. We use recursion and memoization in such problems.

Algorithm: -

1. Two strings a and b are created .
2. Ans(arr,n,a,b){

If(arr[n]!=-1) return arr[n]

If(arr[n]==-1&&a[i]==b[j]) {

arr[n]=ans(a,b,i+1,j+1)

Return arr[n]

}

Arr[n]=max(ans(a,b,i,j+1), ans(a,b,i+1,j))

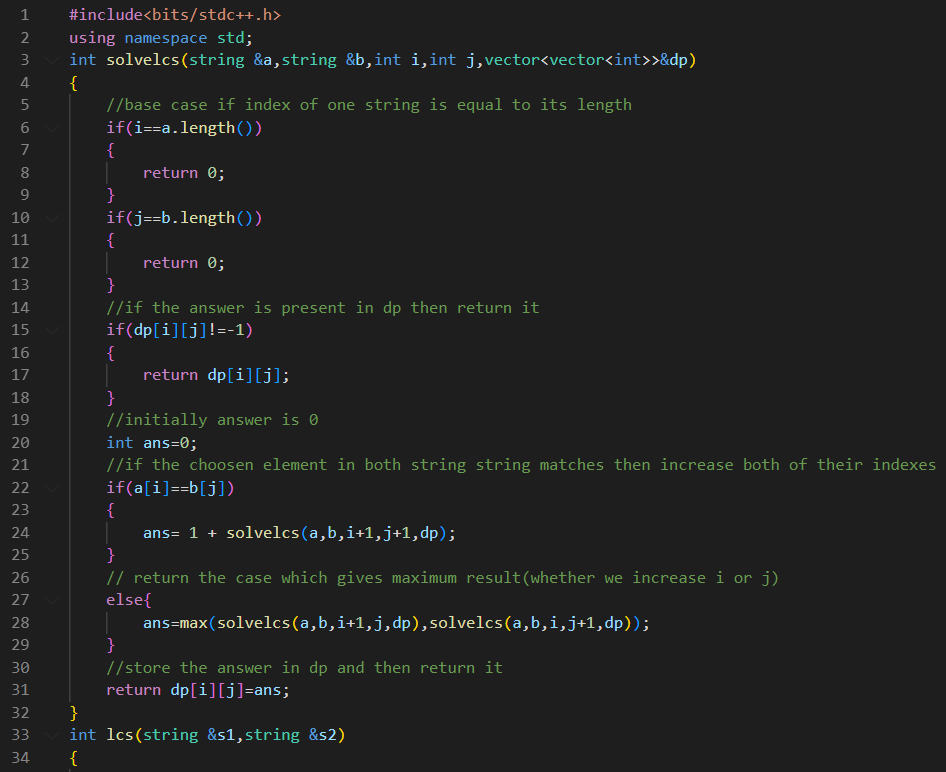
Return arr[n];

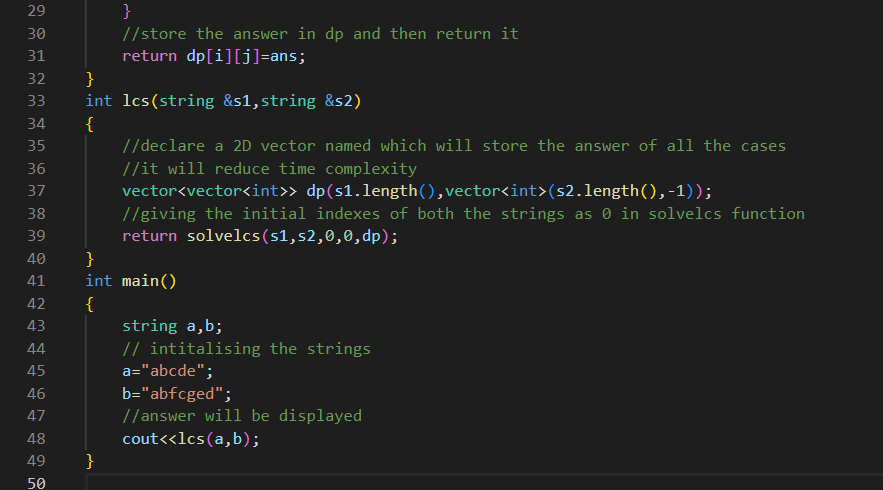
}

Time Complexity:

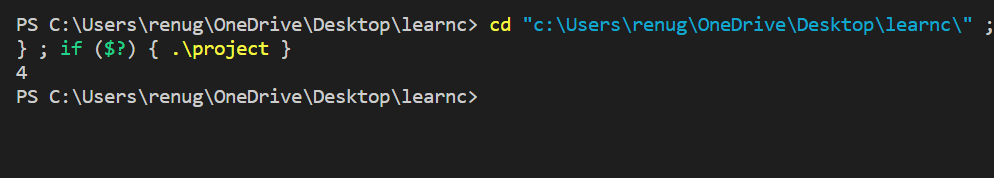
O(n\*m) where n and m are string lengths of 1st and 2nd strings.

**Code:-**





OUTPUT:



Experiment No. 16

Travelling Salesman Problem

Objective:- Implement Travelling Salesman Problem.

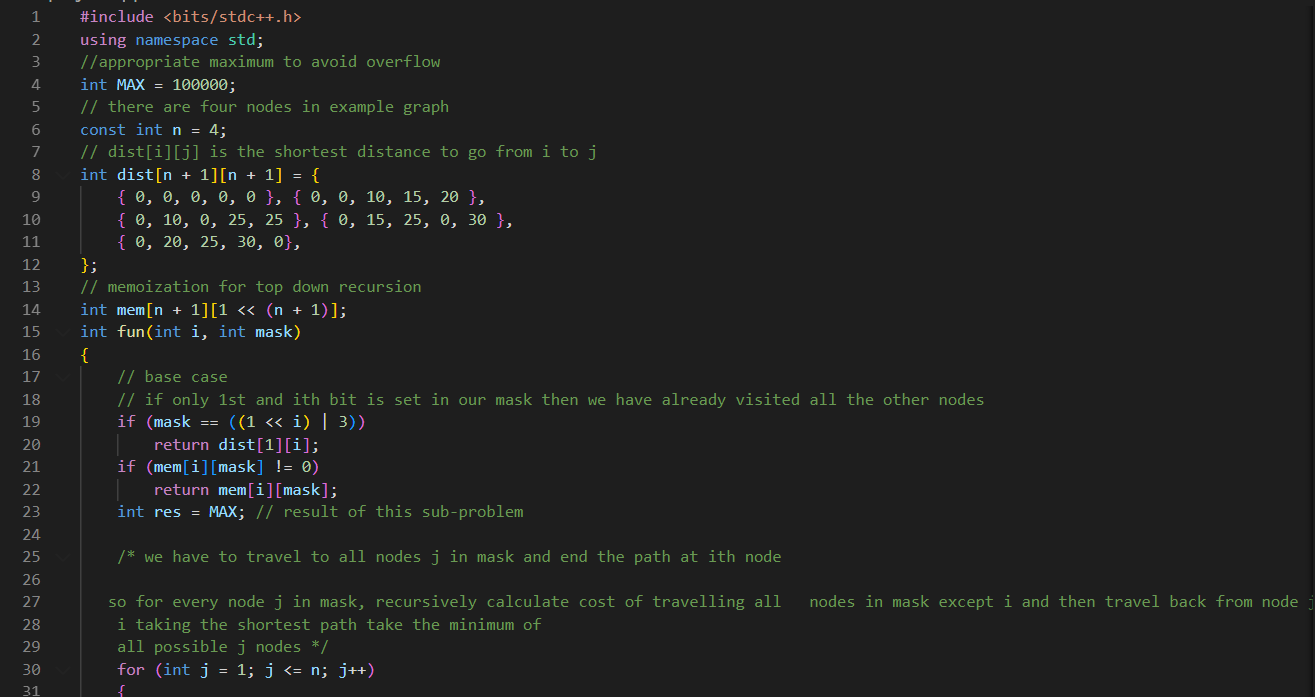
Theory:- it is a graph based problem in which we are required to visit all the nodes of graph such that cost for visiting all the nodes is minimum hence it is a optimisation problem so we will use greedy method.

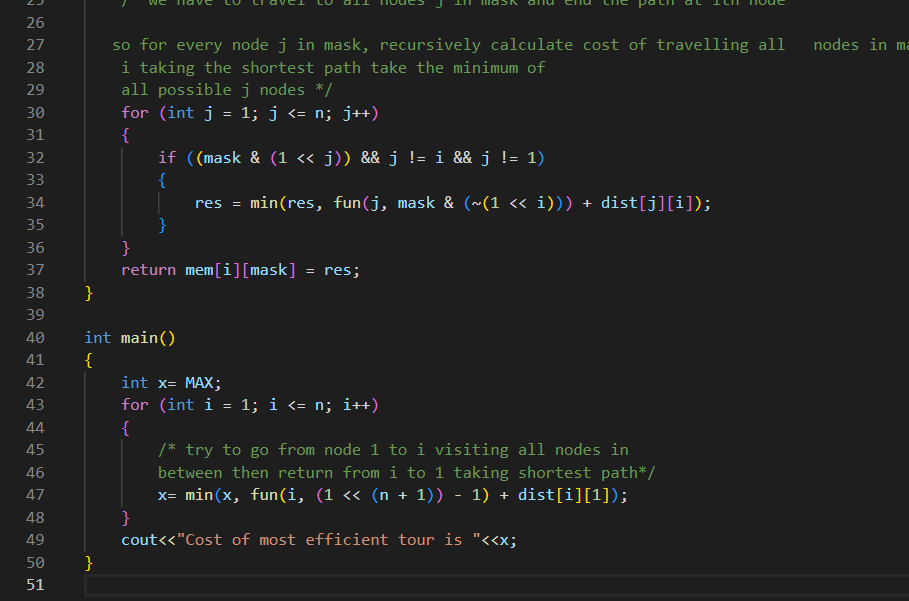
Algorithm:-

* Create two arrays
* Holding city index and distance between the the cities or cost of tour betwee two cities.
* Create a 2d matrix tsp[][]
* if the cost of the reaching any city from current city is less than current cost the update the cost.
* Generate the minimum path cycle using the above step and return their minimum cost

Time Complexity: - There are greater than or equal to O(n\*2n) subproblems, and each takes linear time to solve. The total running time is therefore O(n2\*2n). The time complexity is much less than O(n!) but still exponential.

Code: -





OUTPUT:

