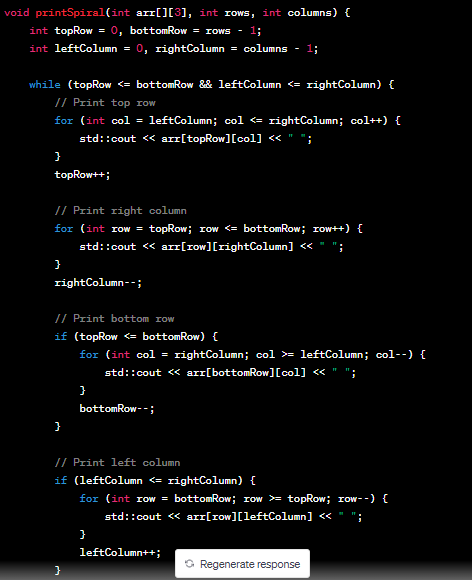
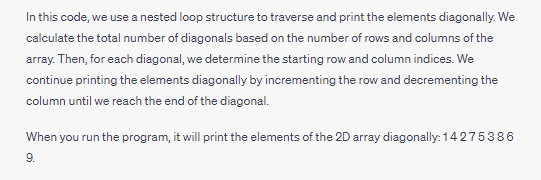
**Program to print 2D array element in spiral order**



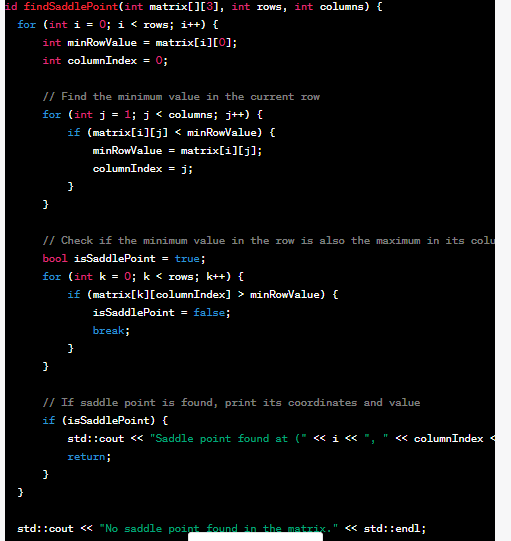
https://www.youtube.com/watch?v=SV5ao\_ITBjI

**Program to Print 2D array elements diagonally**

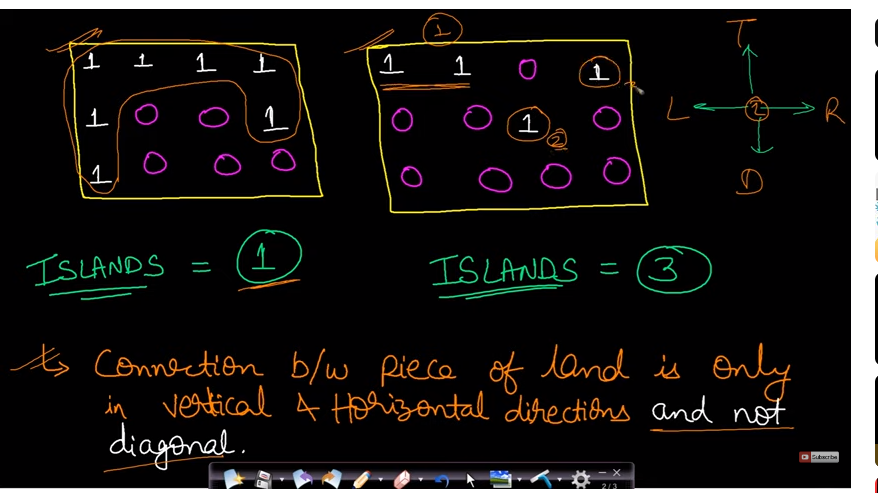




**Program to find 2D Matrix Saddle point**

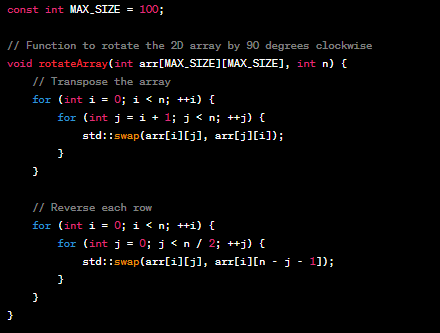


**Program for Number of islands**

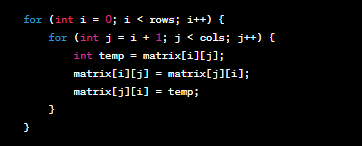


https://www.youtube.com/watch?v=\_\_98uL6wst8

**Program to rotate 2D array by 90 Degree**

https://www.youtube.com/watch?v=SoxrXQbhCPI

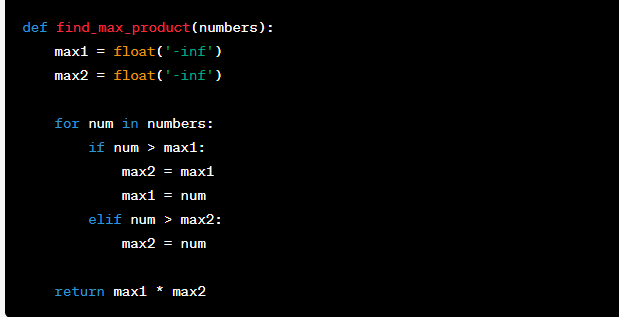
**Program to take transpose of matrix**



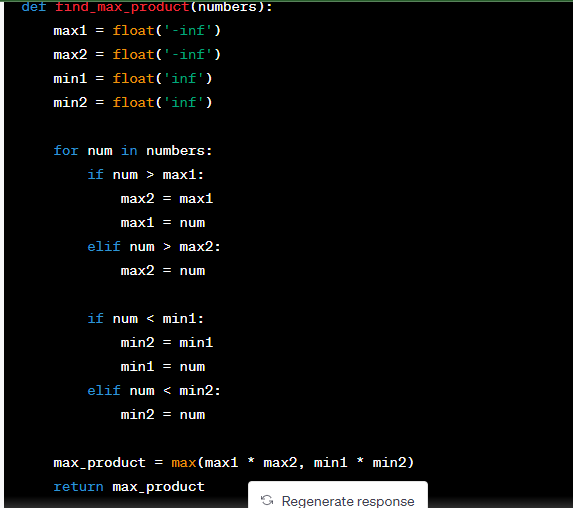
**Write a code to store total number of object created of a class**



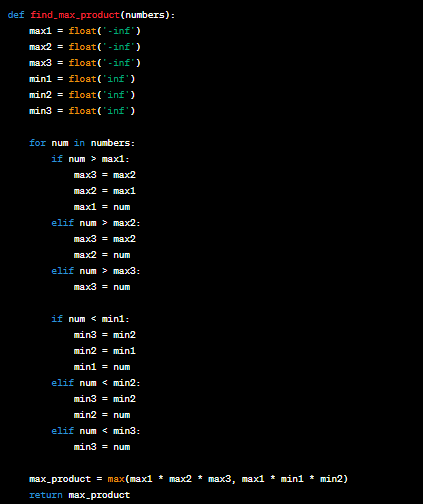
**Program to get two +ve numbers having highest product**



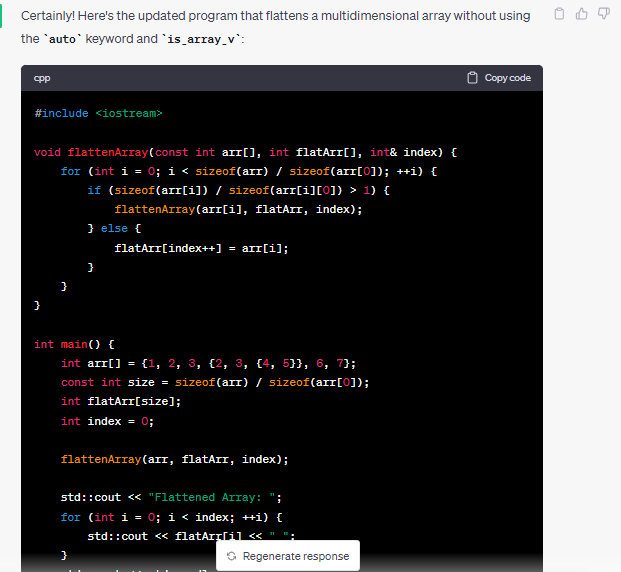
**Program to get two integers having highest product**



**Program to find three integers having highest product**

**s**

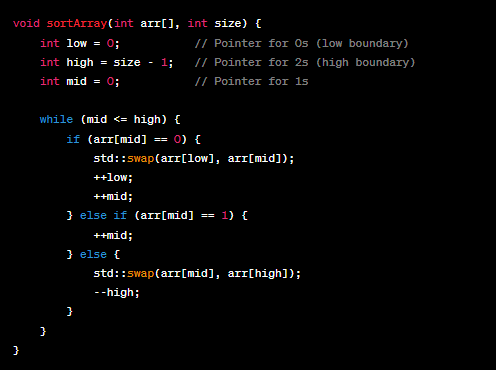
**Program to convert multidimentional array into flat 1D Array**

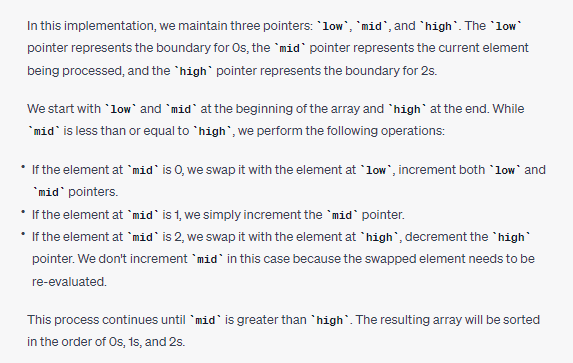


**Check if the string is anagram**



# Sort 0,1,2 array





https://www.youtube.com/watch?v=sEQk8xgjx64

# Rearrange array alternately

# We want this output

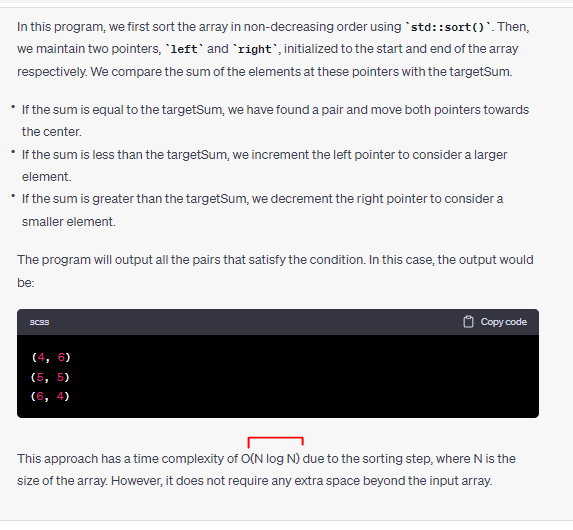
# 

# 

# 

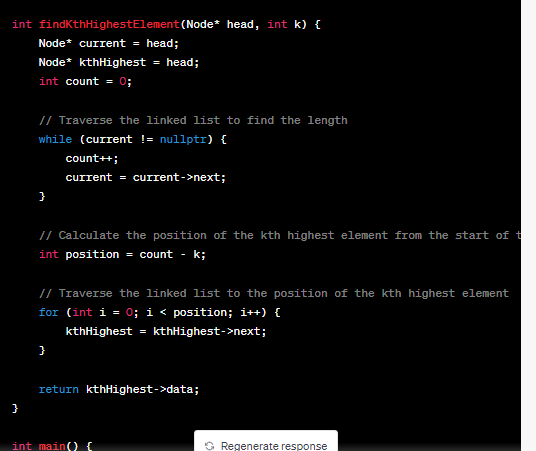
# Find all pairs whose sum is X(sorted array)

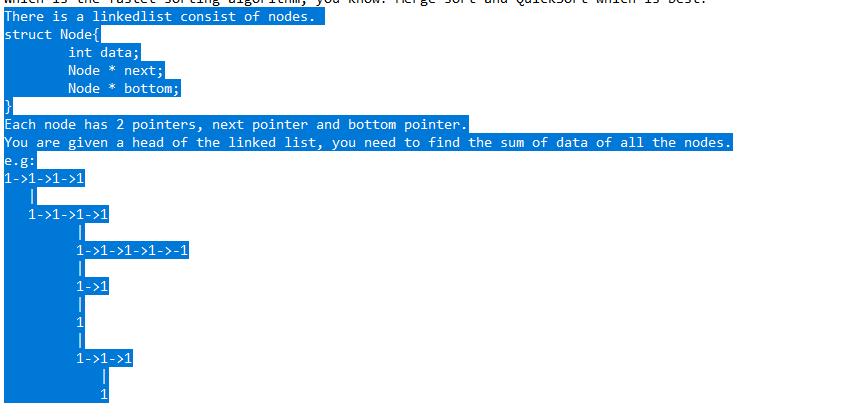
# 

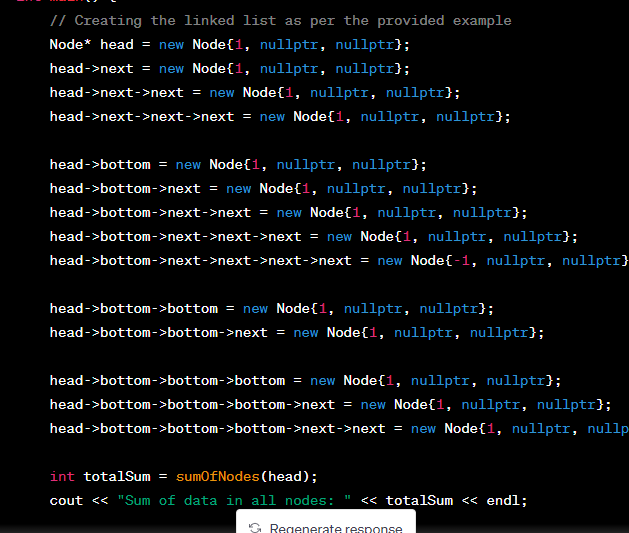
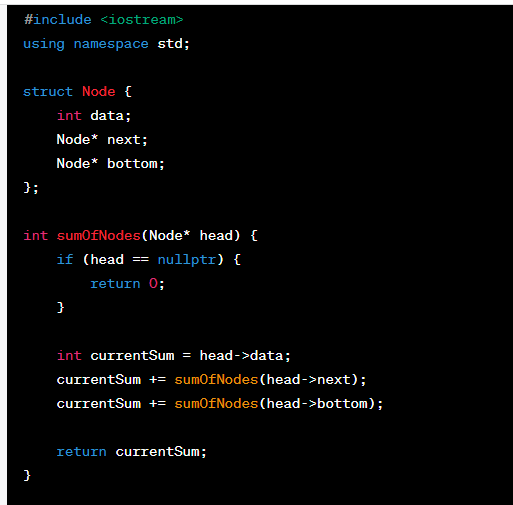


**Given k, find kth element from last in linked list**

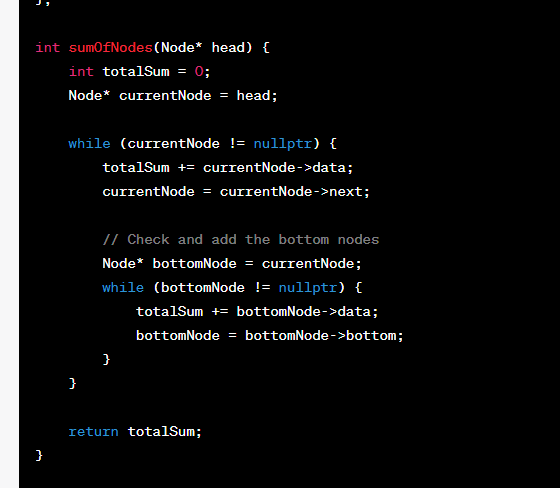
**Method1:if we know size**



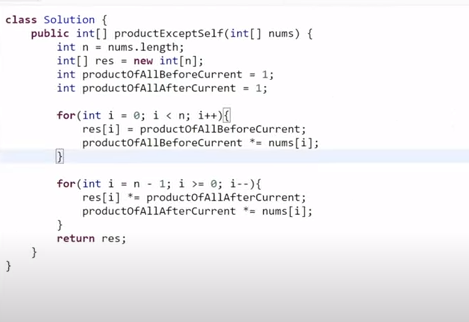




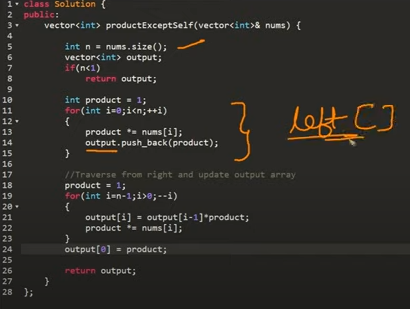
**Non recursive method**



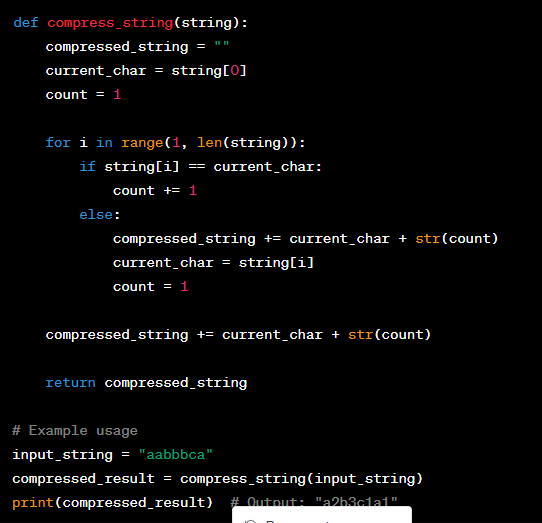
# Product of Array except itself

https://www.youtube.com/watch?v=0TIUlVdvVJo

[**https://www.youtube.com/watch?v=gREVHiZjXeQ**](https://www.youtube.com/watch?v=gREVHiZjXeQ)



**Convert aabbbca to a2b3c1a1 in single loop**



**Write a function f(A\*, B\*, size) which receive two arrays and size of array,**

**you need to populate the Array A in B in such a way that, B element contain the product of all the elements of A exluding ith element itself.**

**Time complexity = bigO(N)**

**if A = {1,3,2,5}, then B = {30, 10, 15, 6}**

**Note:**

~~When we take any odd nuber & with 1 then answer will be 1.i.e~~

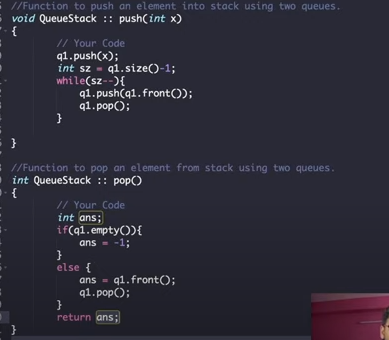
~~5&1=1 , 7&1=1~~

~~But if we take even number then answer will be 0~~

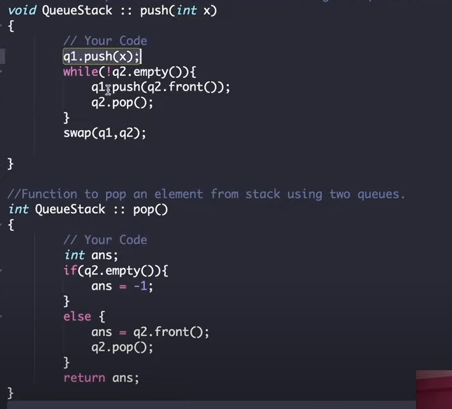
~~4&1=0 , 2&0=0~~

**Q Implement stack using queue**

**Approach1:Using 1Queue**



**Approach2:Using 2Queues**

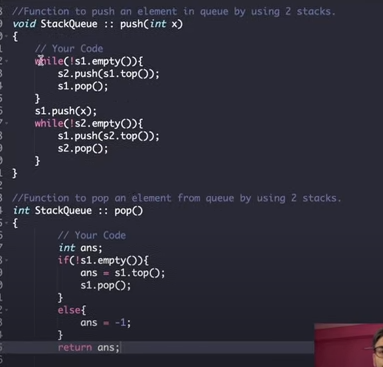


https://www.youtube.com/watch?v=SgQ0VV3eM7Q

**Q Implement queue using stack**



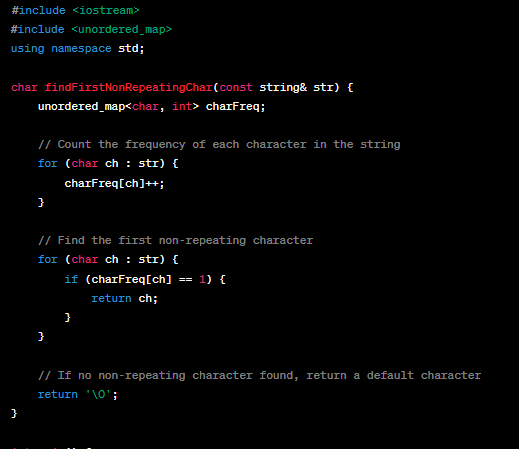
**Approach1:using two stacks**



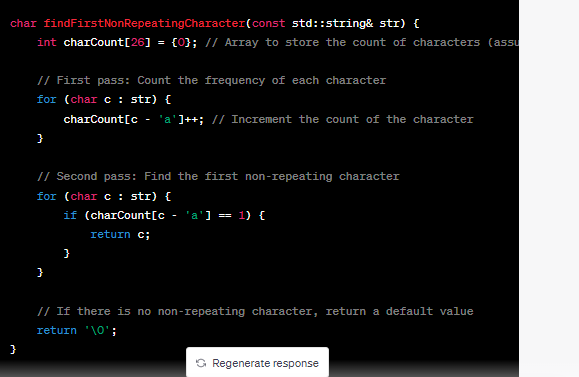
**Approach2:**

https://www.youtube.com/watch?v=BJE3VEQWtIs

**Q return non repeating characters in string**

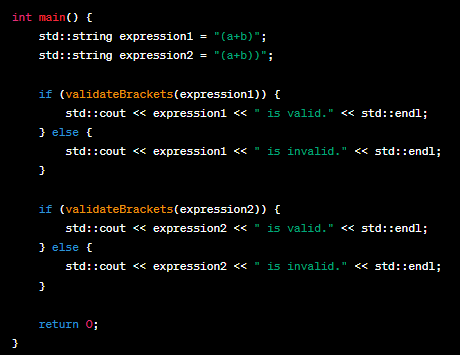


**Method2**

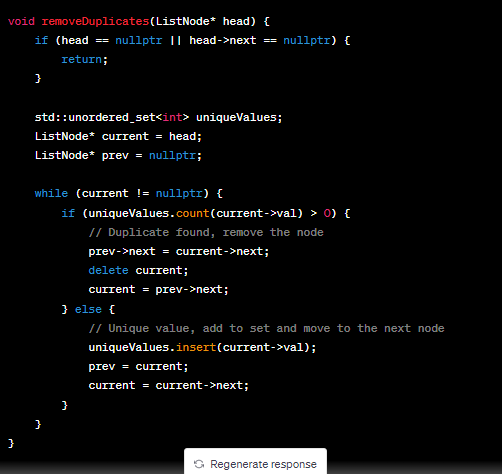


In this updated implementation, we use an array **charCount** of size 26 to represent the count of characters, assuming lowercase alphabets. The index of each character in the array corresponds to its ASCII value minus 'a'.

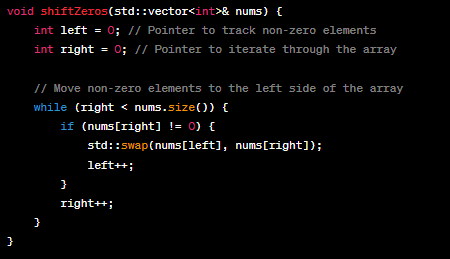
**Q Validate the bracket from an expression... Like (a+b) is valid, (a+b)) is invalid**



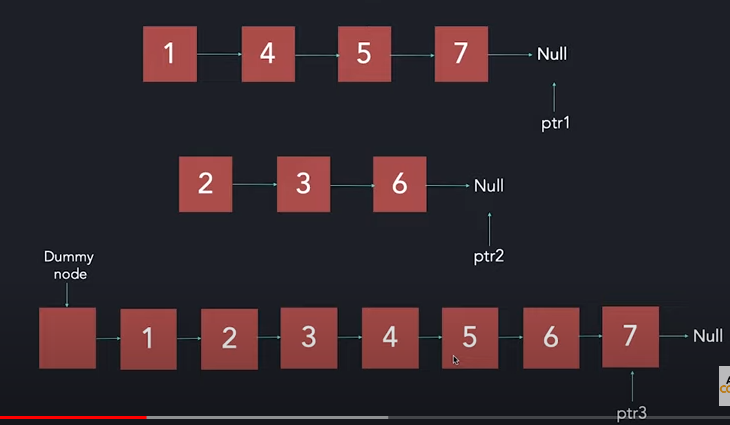
**Q remove duplicates from unordered linkedlist**

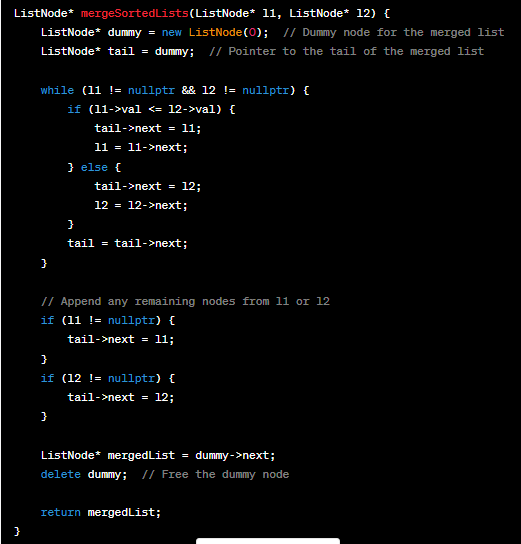


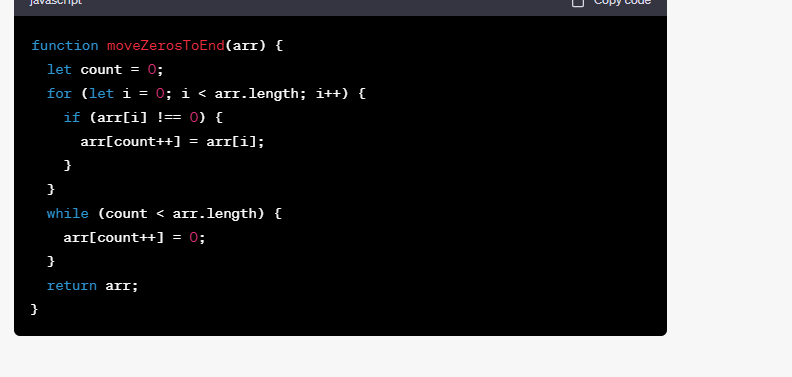
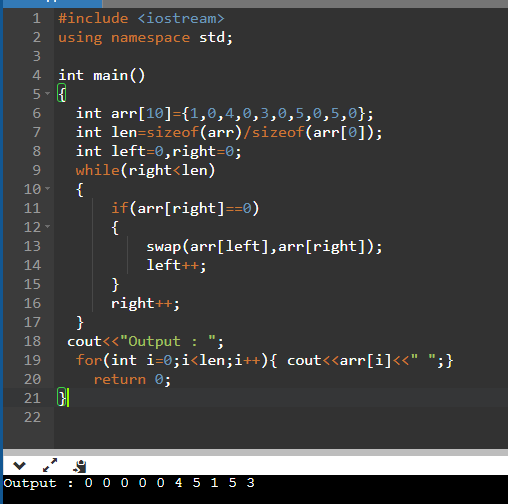
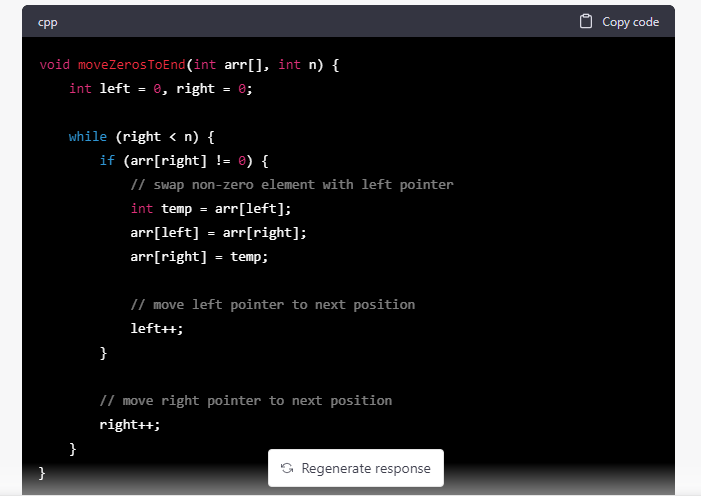
**Q shift zeros in array**

**1 2 0 5 0**

**Q Merge Two sorted linkedlist**



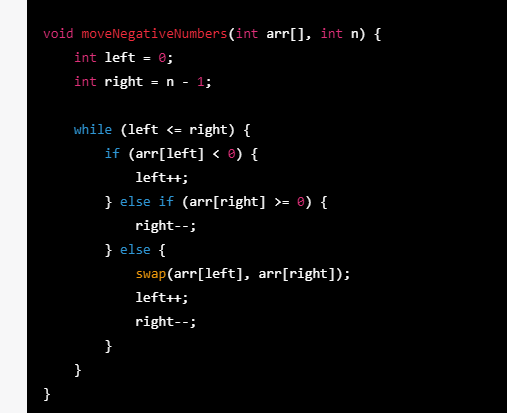


**1.Move Zeros To The end of Array** 

**2.Segregate positive and negative numbers in Array**

Negative number ek side pr aur positive number ek side pr

Best optimal method:



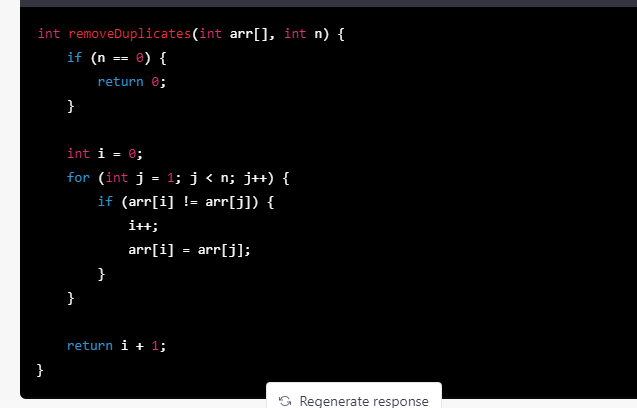
**Method1**



**Method2**



**3.Remove Duplicates from sorted Array**



**4. Remove leading and trailing spaces only from the string**

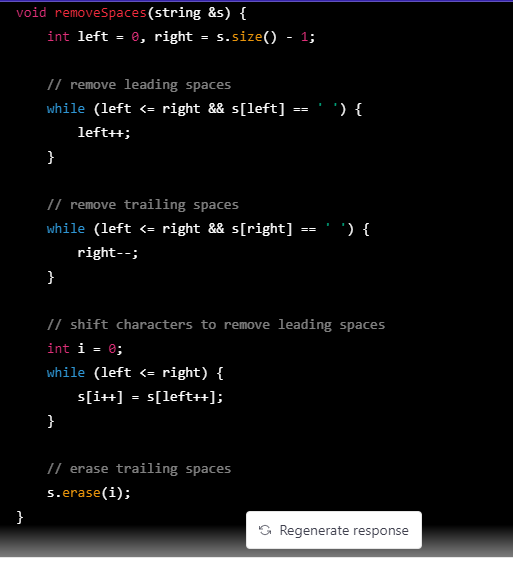
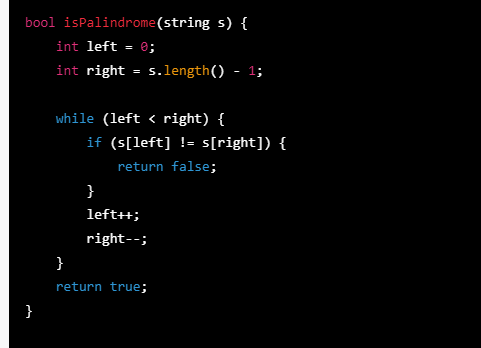


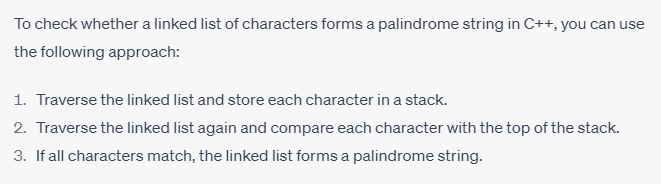
Figure original string changes

**Q5: Palindrome String**

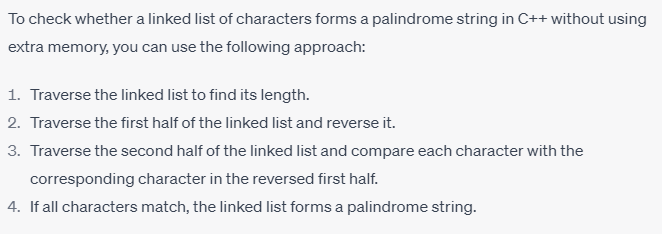


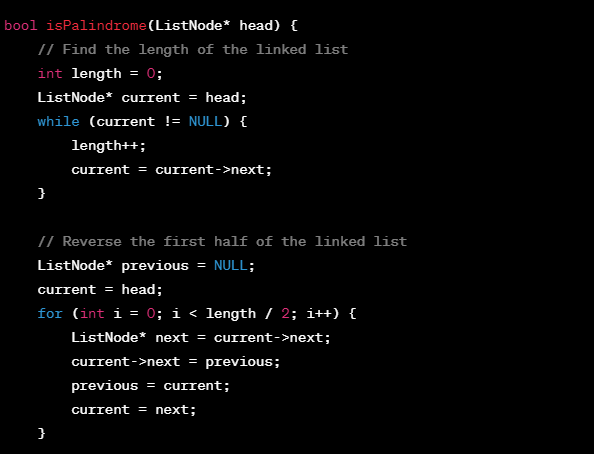
**Q6 Palindrome String linkedlist**

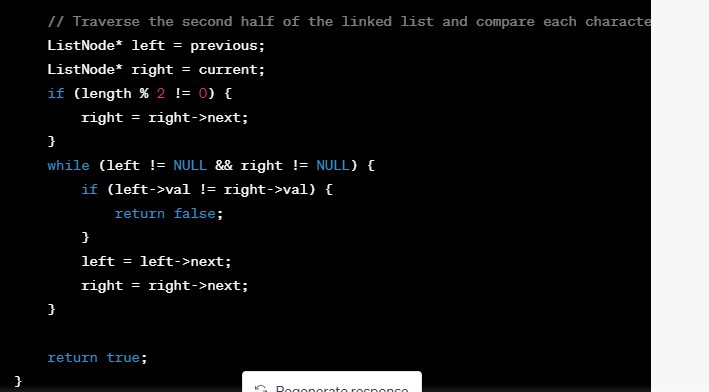
**Method1:**



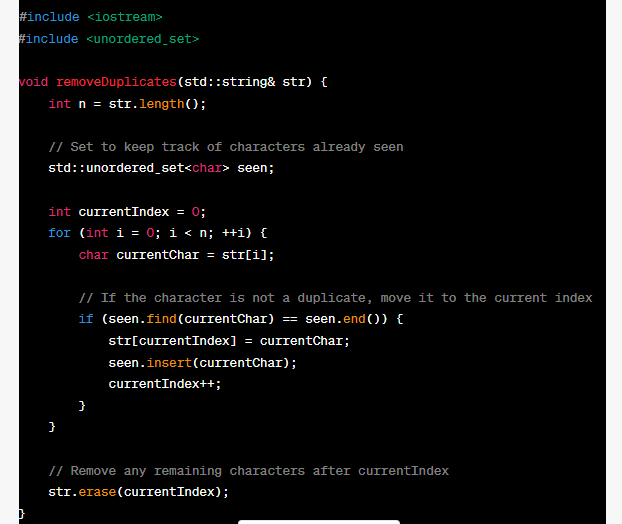
**Method2:**







**Q Remove duplicates from string**

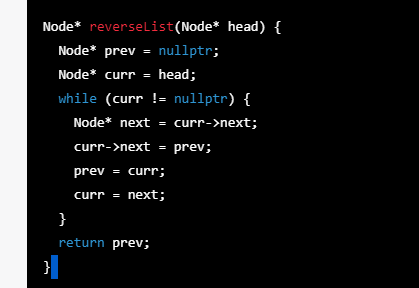
 In this implementation, the **removeDuplicates** function takes a string **str** as input and modifies it in-place to remove duplicate characters.

The function initializes an unordered set, **seen**, to keep track of characters already encountered. The **currentIndex** variable is used to store the current position where non-duplicate characters will be placed.

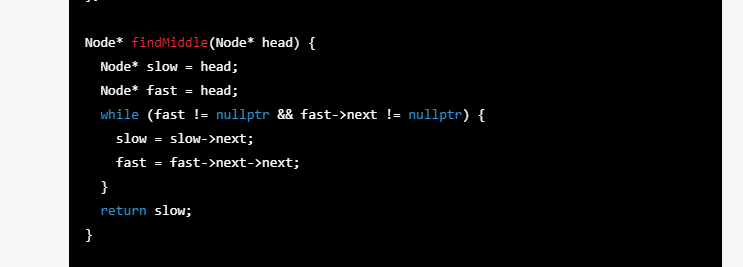
The function then iterates through each character of the string. If the character is not found in the **seen** set (indicating it is not a duplicate), it is moved to the current index, and the **seen** set is updated accordingly. The **currentIndex** is incremented to prepare for the next non-duplicate character.

Finally, any remaining characters after **currentIndex** are removed from the string using the **erase** function.

**Q7 Reverse Linkedlist**



**Q8 Find Middle of linkedlist**



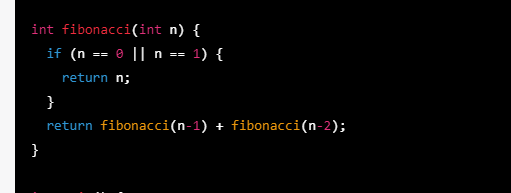
**Q9 Factorial print**



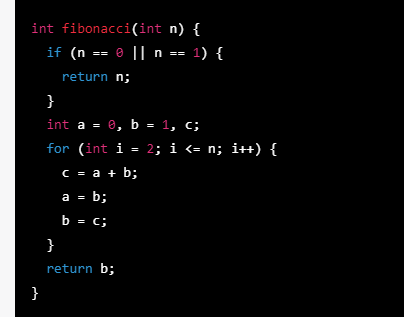
**Q10Fabonacii series print**

|  |
| --- |
|  |
|  |

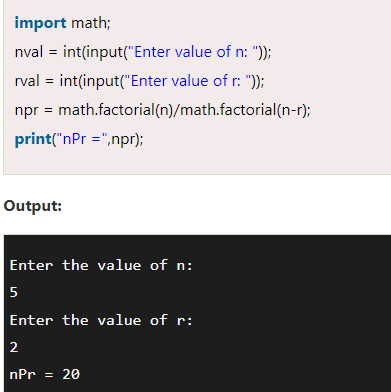
**Q11 Print nth fabonaci number**



**Non recursive:**



**Q12:Print Permutation & Combination of number nPr & nCr**

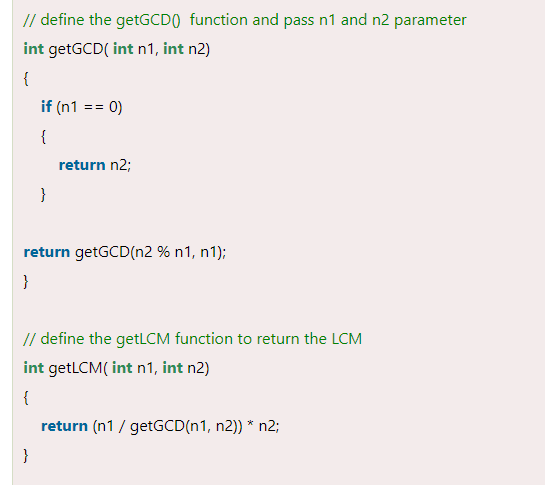


**Formula for permutation = n!/(n-r)!**

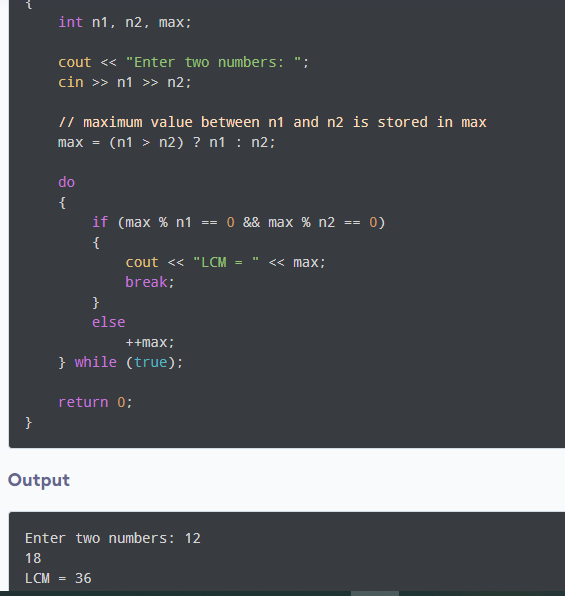
**Formula for combination =n!/r! \* (n-r)!**

**Q12 : Print LCM of numbers**

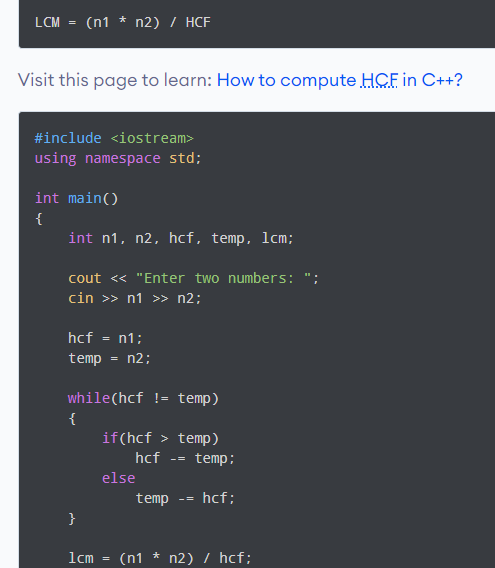
**Method1:Using gcd**



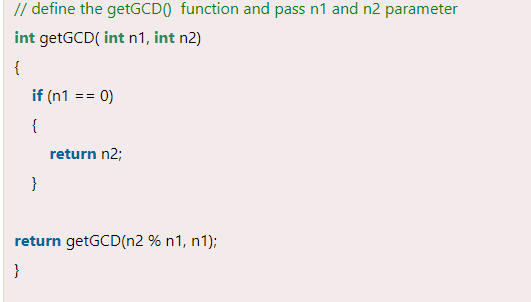
**Method2:Without gcd**



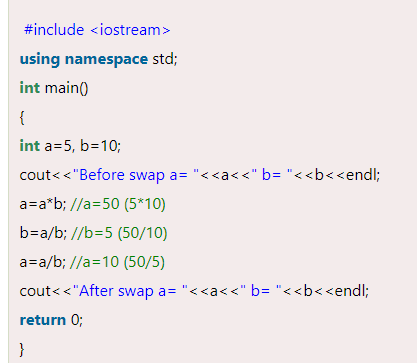
**Method4**



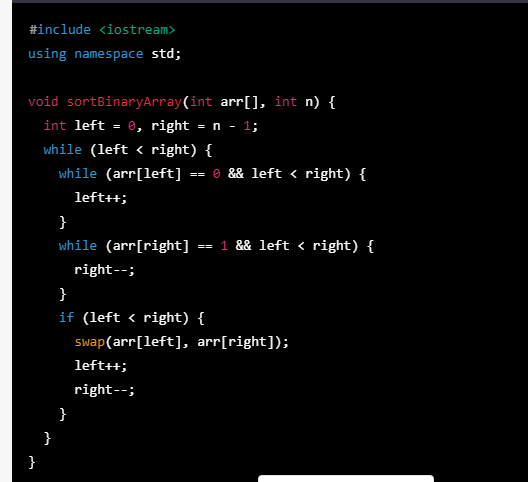
**Q13:find gcd of two numbr**



**Q15 Swap two numbers**

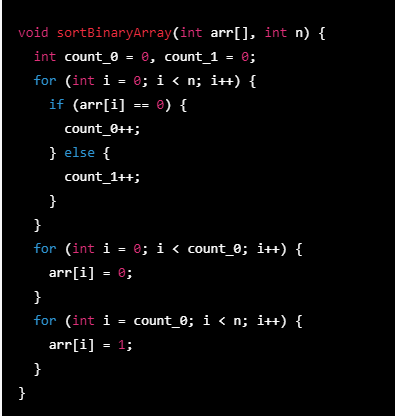


**Q16 Sort an array of binary numbers (0s and 1s)**



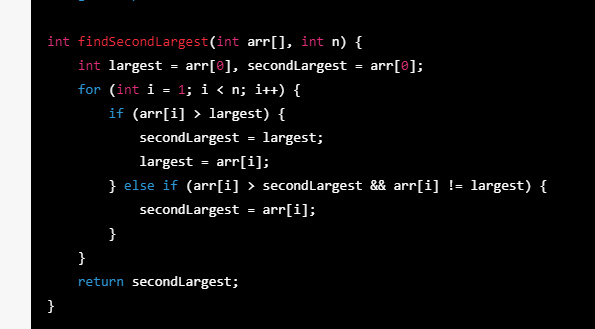
In this implementation, we first define a sortBinaryArray function that takes an array of integers arr and its length n as input. We initialize two pointers left and right to the beginning and end of the array, respectively. We then loop until left is less than right, and move left and right towards each other. We use two inner loops to move left past all the 0s and right past all the 1s. If left is still less than right, we swap the elements at left and right, and move both pointers towards each other. After the loop completes, all the 0s will be to the left of the array and all the 1s will be to the right.

**Method2: easy**



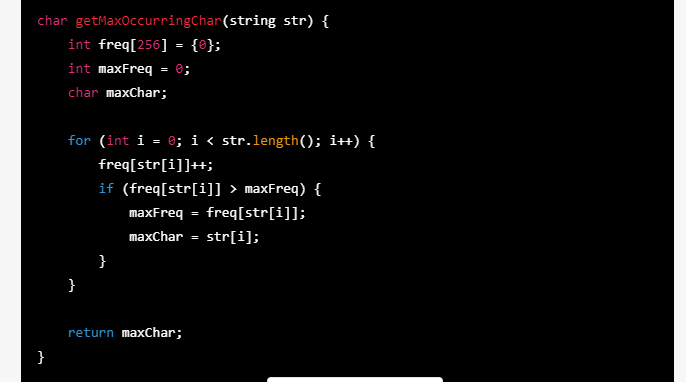
**Q17**

**Find 2nd largest number in array**



**Method2: sorting**

**Q18:** **find frequency of characters in a string**



<https://www.youtube.com/watch?v=FvssDsoahcU>

Ye mthod sirf capital letters contained string ka liay work kry ga

int findMaxFrequency(char arr[], int n) {

int fre[26]={0};

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

{

int nu=int(arr[i]);//convert character to digit

fre[nu-65]++;

}

for(int i=0;i<27;i++)

{

cout<<static\_cast<char>(i+65)<<fre[i]<<" ";

}

int max=0;

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

{

if(fre[i]>max){max=fre[i];}

}

return max;

}

int main() {

char arr[] = "HELLO WORLD";

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

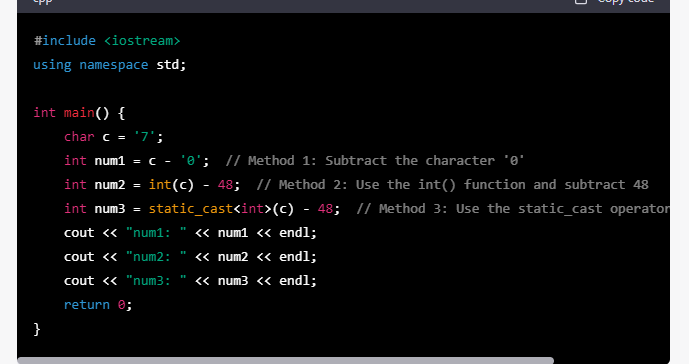
int maxFrequency = findMaxFrequency(arr, n);

cout << endl<<"Frequency of maximum occurring character in "<<arr<<" is "<<maxFrequency<< endl;

return 0;

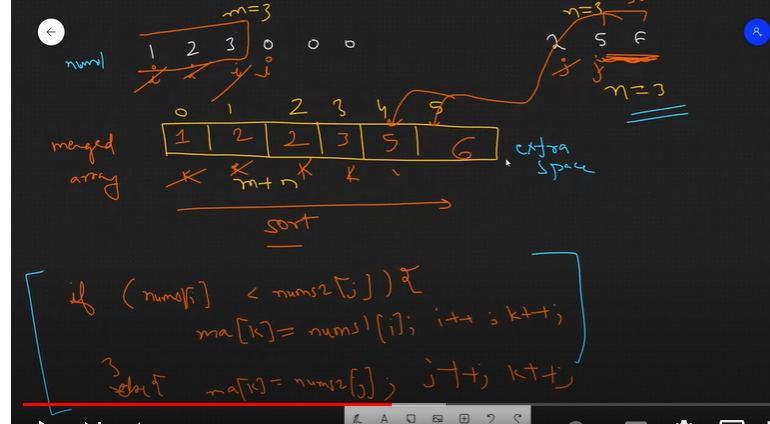
}

**Q19:Convert character to int**



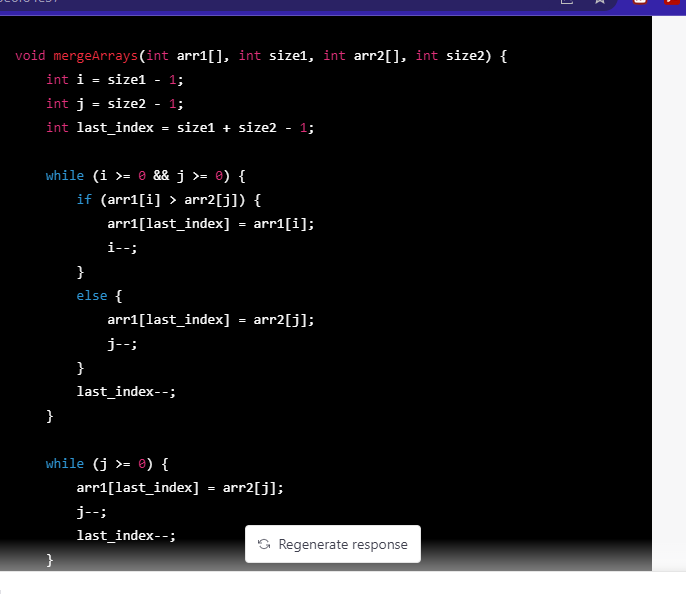
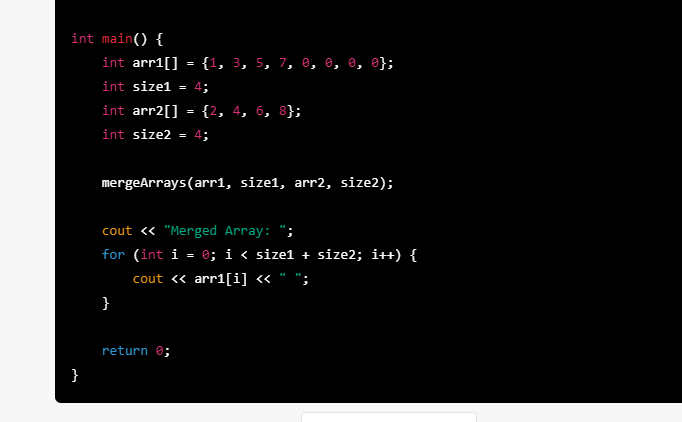
**Q20: Merge 2 sorted arrays without extra space**

**Method 1: with extra space**



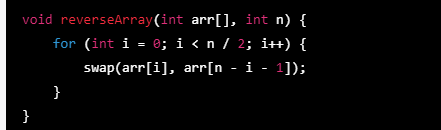
Donon arrays sa pkr kr tesri array ma dalty jaien

**Method2:**

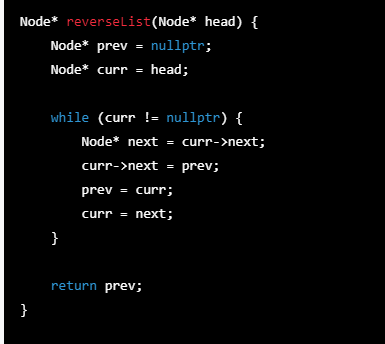
 

**Method3:**

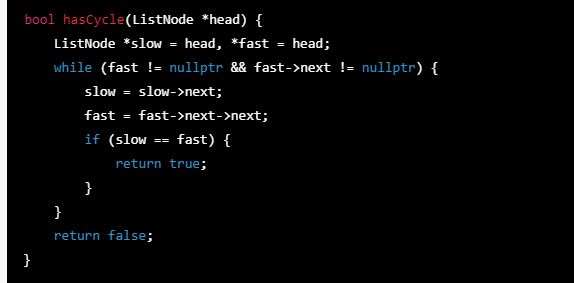
**Q21Reverse an array**



**Q22 Reverse linkedlist**



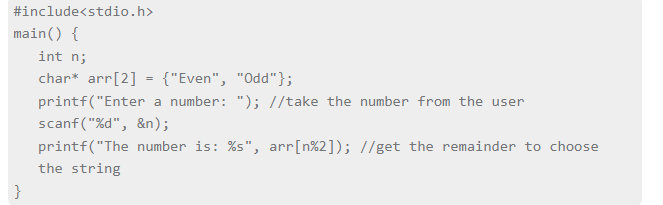
**Q23 Find circle in linkedlist**



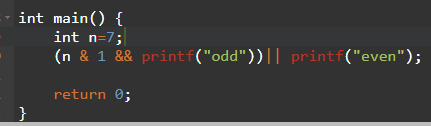
In this code, we first check if the list is empty or has only one node. Then, we initialize two pointers **slow** and **fast**. **slow** starts at the head node, and **fast** starts at the second node. We then loop through the list, moving **slow** one step and **fast** two steps at a time. If there is no cycle, **fast** will eventually reach the end of the list, and we return **false**. If there is a cycle, **fast** will eventually catch up with **slow**, and we return **true**.

# Q24  print “Even” or “Odd” without using Conditional statement

# Method1



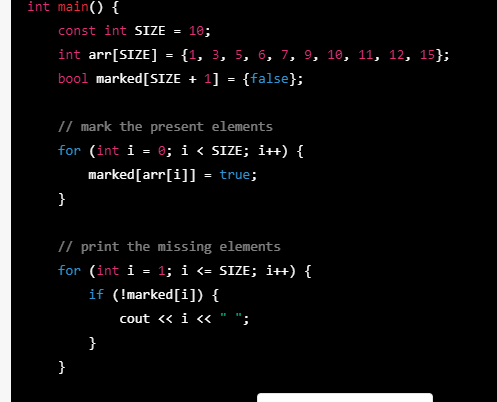
**Method2**



**Q25:**

**Print missing values in array**

**Method1: With extra space**



**Method2: without extra space:**

**Q26: Rotate the array**

**Method1:**

**Using reverse method**

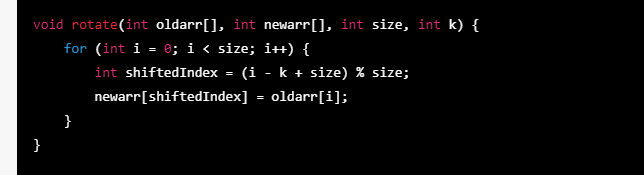
**Left Rotate :** The reversal algorithm is a common technique used to rotate an array. It involves reversing the elements of the array in two steps, followed by reversing the entire array to get the desired rotation.

Here's the algorithm to rotate an array in O(n) time and O(1) space using the reversal technique with the kth element as the pivot:

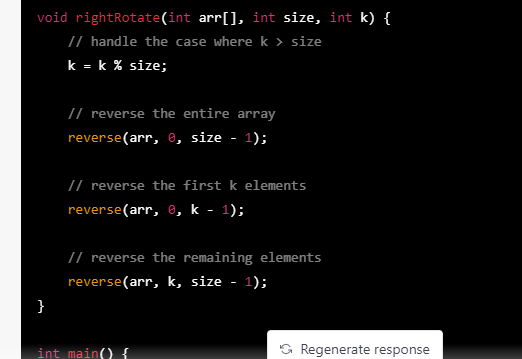
1. Reverse the first k elements of the array.
2. Reverse the remaining n-k elements of the array.
3. Reverse the entire array.



**Method for right rotate (using new array):**



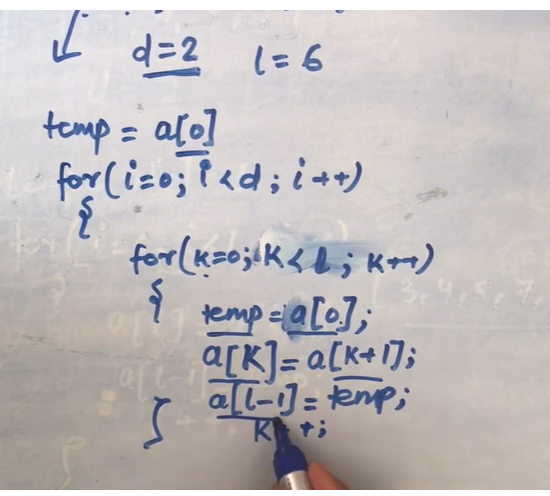
**Methof for right rotate(without extra space)**



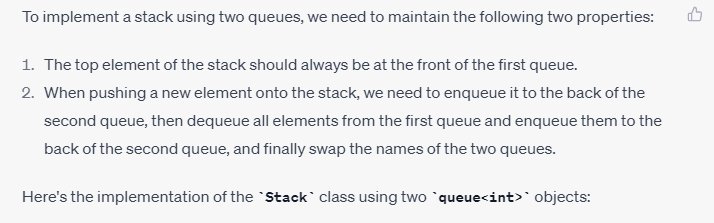
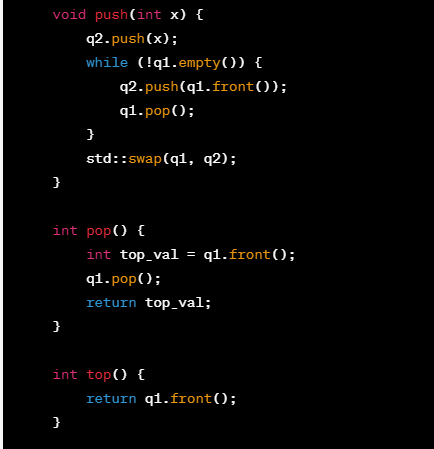
**Conclusion**

|  |  |
| --- | --- |
| **Right Rotate** | **Left rotate** |
|  |  |

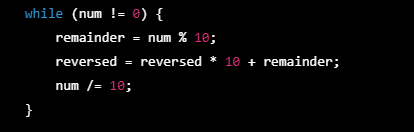
**Worst method**



**Q27 Implement stack**

**Q28 Reverse number**



**Q29 Remove Nth Node From End of List**

**Method1:**

Reverse the linkedlist and delete that nth node

**Method2:**

1.Count total length of linkedlist n.

2. use formula to calculate the position of desired node from the left side because we only know the position of that desired node from the right side.

Position of that node from left side=Size-n+1

3.Delete that nth node.

**Method3:**

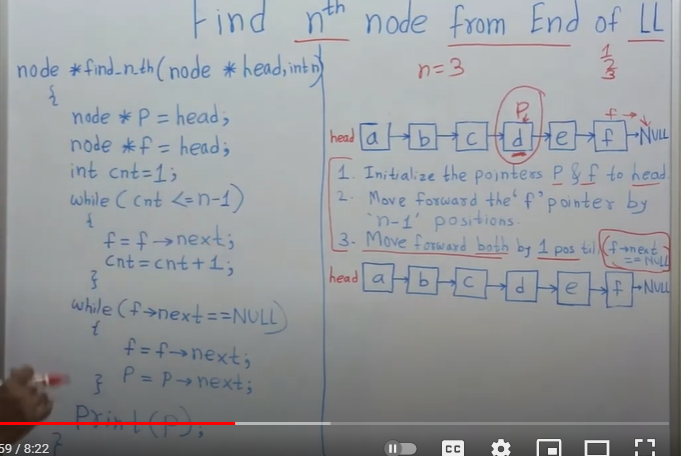
Two pointer approach.

1.ek fast pointer aur ek slow pointer donon ko head sa initialize krwa dia.

2.Ek loop li aur usmy condition rkhi ka isko N tk chalao yani fast pointer ko N tk chala.

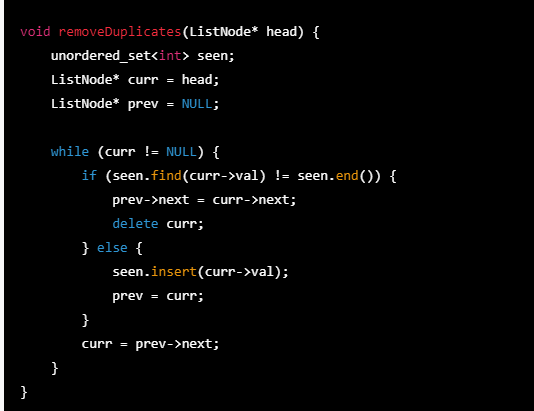
3.ab ek aur loop lagai ka fast jb tk null end of linkedlist na pohanch jay ye loop chly.

Ismy jb slow=slow->next aur fast=fast->next aur loop ki condition ka fast end tk na pohanch jay tb tk ye loop chalao.

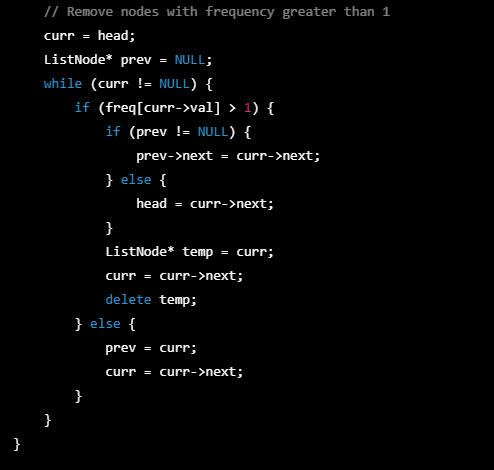
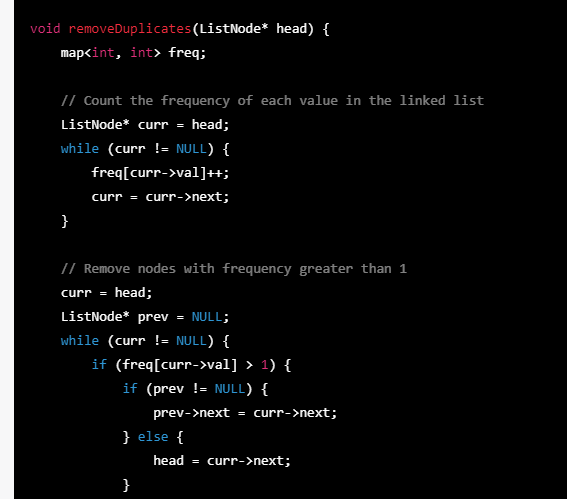


<https://www.youtube.com/watch?v=5BpQ5m0K_t4>

**Q30** **Remove duplicates from an unsorted linked list**



**Method2: using map data structure**



The **removeDuplicates** function takes the head of the linked list as input and removes any duplicates from the list. We start by creating a **map** called **freq** to store the frequency of each value in the linked list. We then iterate through the linked list and update the frequency of each value in the **map**.

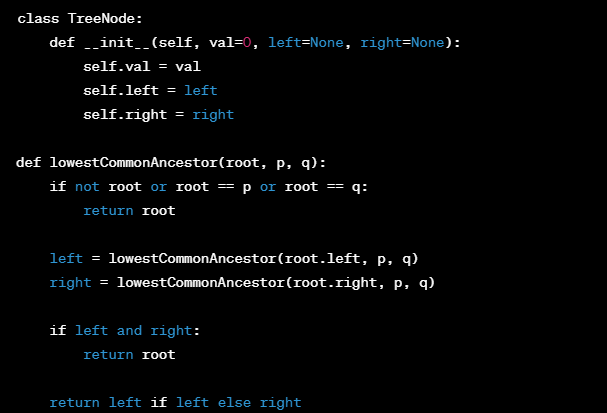
After counting the frequency of each value, we iterate through the linked list again and remove any nodes with a frequency greater than 1. To do this, we keep track of the previous node (**prev**) as well as the current node (**curr**). If the frequency of the current node's value is greater than 1, we skip over it and update the **next** pointer of the previous node to point to the next node. We then delete the current node and move on to the next node. If the frequency of the current node's value is 1, we update **prev** and **curr** and continue iterating through the linked list.

# Lowest common ancestor of two nodes in Binary Tree Algorithm

To find the lowest common ancestor of two nodes in a binary tree, you can use the following algorithm:

1. Start from the root node of the tree.
2. If either of the two nodes is the root node, return the root node.
3. Recursively search the left subtree of the root for the first node. If the first node is found in the left subtree, return the result of the recursive call for the left subtree.
4. Recursively search the right subtree of the root for the first node. If the first node is found in the right subtree, return the result of the recursive call for the right subtree.
5. If the first node is not found in either subtree, it must be in the other subtree, and the root node is the lowest common ancestor. Return the root node.
6. If the first node is found in one subtree and the second node is found in the other subtree, the root node is the lowest common ancestor. Return the root node.

Here is a sample code implementation of this algorithm in Python:

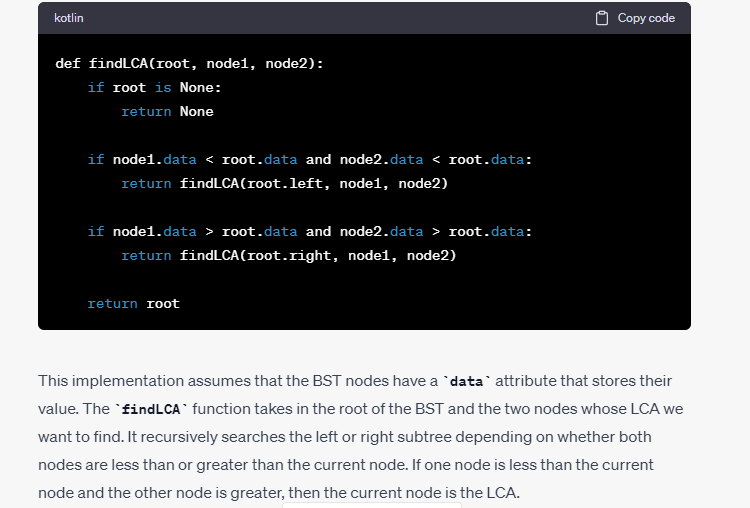
In this implementation, **root** is the root node of the binary tree, and **p** and **q** are the two nodes whose lowest common ancestor we want to find. The function returns the lowest common ancestor of **p** and **q** in the binary tree.

**Lowest common ancestor of two nodes in Binary search Tree Algorithm**

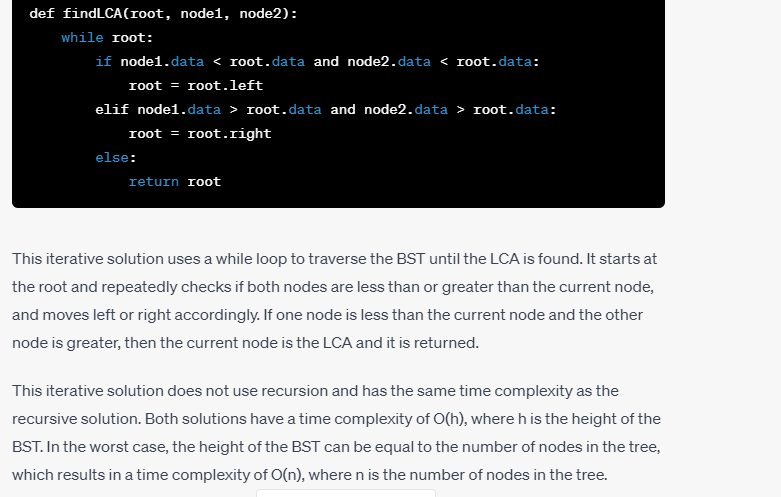
To find the lowest common ancestor (LCA) of two nodes in a binary search tree (BST), you can use the following algorithm:

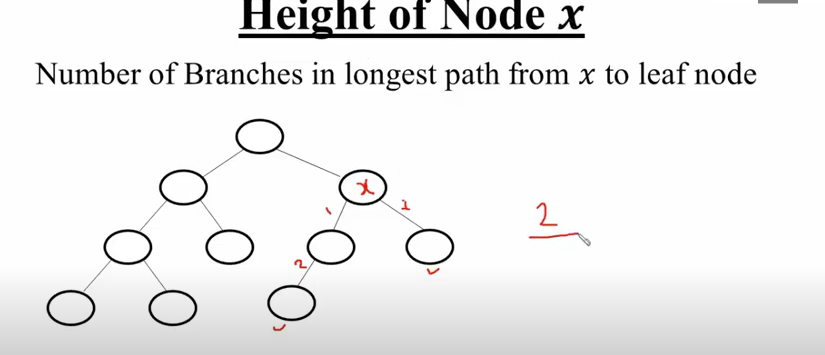
1. Start at the root of the BST.
2. If both nodes are less than the current node, move to the left subtree.
3. If both nodes are greater than the current node, move to the right subtree.
4. If one node is less than the current node and the other node is greater than the current node, then the current node is the LCA.
5. If one of the nodes is equal to the current node, then the current node is the LCA.
6. Repeat steps 2-5 until the LCA is found.

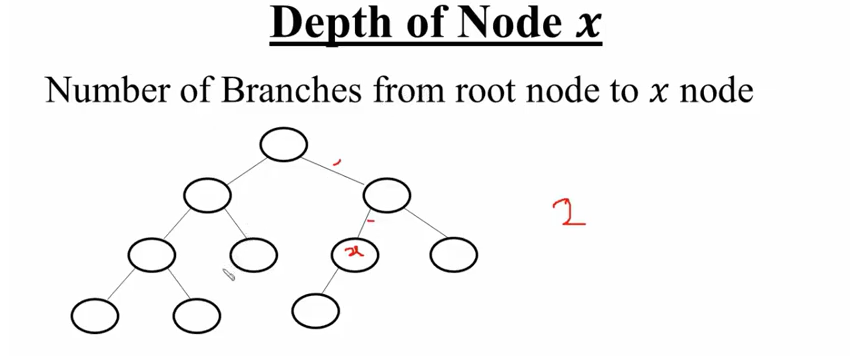
Here's an implementation of this algorithm in Python:



**2nd Method**







1. What is a tree data structure?

A tree is a hierarchical data structure that consists of nodes connected by edges. Each node in a tree has a parent node (except for the root node, which has no parent) and zero or more child nodes. Trees are commonly used to represent hierarchical relationships between data.

For example, consider a file system on a computer. The root directory would be the root node of the tree, with subdirectories and files as its children. Each subdirectory may have its own children, and so on.

1. What is the difference between a binary tree and a binary search tree?

A binary tree is a tree in which each node has at most two children, known as the left child and the right child. A binary search tree (BST) is a binary tree in which each node has a key (or value), and the keys in the left subtree are less than the key in the node, and the keys in the right subtree are greater than the key in the node.

For example, consider the following binary search tree:

markdownCopy code

5 / \ 3 7 / \ \ 1 4 9

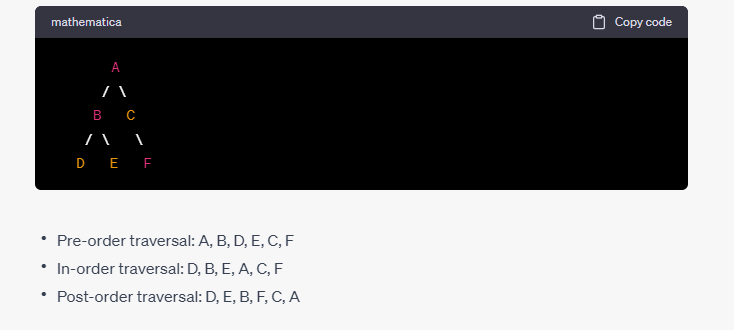
In this tree, the key in each node is represented by the number inside the node. The left subtree of the node with key 5 contains keys 1, 3, and 4, which are all less than 5. The right subtree contains keys 7 and 9, which are greater than 5.

1. How do you traverse a tree depth-first?

Depth-first traversal of a tree involves visiting all the nodes in the tree by exploring as far as possible along each branch before backtracking. There are three commonly used methods for depth-first traversal:

* Pre-order traversal: Visit the current node, then visit the left subtree recursively, then visit the right subtree recursively.
* In-order traversal: Visit the left subtree recursively, then visit the current node, then visit the right subtree recursively. For binary search trees, an in-order traversal will visit the nodes in ascending order.
* Post-order traversal: Visit the left subtree recursively, then visit the right subtree recursively, then visit the current node.

For example, consider the binary tree:



Sure, here are detailed answers and examples for some commonly asked questions about trees in job interviews:

1. What is a tree data structure?

A tree is a hierarchical data structure that consists of nodes connected by edges. Each node in a tree has a parent node (except for the root node, which has no parent) and zero or more child nodes. Trees are commonly used to represent hierarchical relationships between data.

For example, consider a file system on a computer. The root directory would be the root node of the tree, with subdirectories and files as its children. Each subdirectory may have its own children, and so on.

1. What is the difference between a binary tree and a binary search tree?

A binary tree is a tree in which each node has at most two children, known as the left child and the right child. A binary search tree (BST) is a binary tree in which each node has a key (or value), and the keys in the left subtree are less than the key in the node, and the keys in the right subtree are greater than the key in the node.

For example, consider the following binary search tree:

markdownCopy code

5 / \ 3 7 / \ \ 1 4 9

In this tree, the key in each node is represented by the number inside the node. The left subtree of the node with key 5 contains keys 1, 3, and 4, which are all less than 5. The right subtree contains keys 7 and 9, which are greater than 5.

1. How do you traverse a tree depth-first?

Depth-first traversal of a tree involves visiting all the nodes in the tree by exploring as far as possible along each branch before backtracking. There are three commonly used methods for depth-first traversal:

* Pre-order traversal: Visit the current node, then visit the left subtree recursively, then visit the right subtree recursively.
* In-order traversal: Visit the left subtree recursively, then visit the current node, then visit the right subtree recursively. For binary search trees, an in-order traversal will visit the nodes in ascending order.
* Post-order traversal: Visit the left subtree recursively, then visit the right subtree recursively, then visit the current node.

For example, consider the binary tree:

mathematicaCopy code

A / \ B C / \ \ D E F

* Pre-order traversal: A, B, D, E, C, F
* In-order traversal: D, B, E, A, C, F
* Post-order traversal: D, E, B, F, C, A

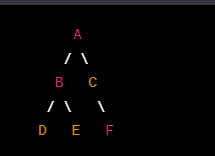
1. How do you traverse a tree breadth-first?

Breadth-first traversal of a tree involves visiting all the nodes in the tree level by level, starting from the root node and moving down to each level in turn. This is also known as level-order traversal.

To perform a breadth-first traversal of a tree, you can use a queue data structure. Start by enqueueing the root node, and then dequeue the next node from the queue, visit its children (if any), and enqueue them. Repeat this process until the queue is empty.

For example, consider the binary tree:

mathematicaCopy code



A breadth-first traversal of this tree would visit the nodes in the following order: A, B, C, D, E, F.

1. What is a self-balancing tree and how does it work?

A self-balancing tree is a binary search tree that automatically keeps itself balanced, meaning that the height of the left

1. What is a self-balancing tree and how does it work? (continued)

A self-balancing tree is a binary search tree that automatically keeps itself balanced, meaning that the height of the left and right subtrees differ by at most one. This is important because an unbalanced binary search tree can lead to poor performance in operations such as search, insert, and delete.

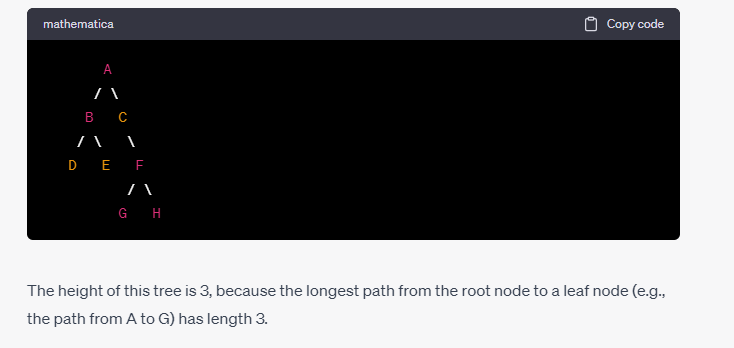
There are several types of self-balancing trees, including AVL trees, red-black trees, and B-trees. Each of these trees has its own balancing criteria and methods for rebalancing the tree.

For example, in an AVL tree, each node is assigned a balance factor that is the difference between the heights of its left and right subtrees. If the balance factor of a node is greater than 1 or less than -1, the tree is rebalanced by performing rotations on the affected subtree(s) to restore balance.

1. How do you find the height of a tree?

The height of a tree is the length of the longest path from the root node to any leaf node. One way to find the height of a tree is to perform a depth-first traversal of the tree and keep track of the maximum depth seen so far. Another way is to perform a breadth-first traversal of the tree and keep track of the current level.

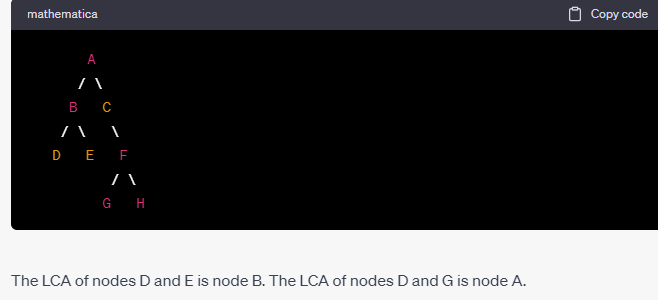
For example, consider the binary tree:

1. How do you find the lowest common ancestor of two nodes in a binary tree?

The lowest common ancestor (LCA) of two nodes in a binary tree is the deepest node that is a common ancestor of both nodes. One way to find the LCA of two nodes is to perform a depth-first traversal of the tree, keeping track of the path from the root node to each of the two nodes. The LCA is then the last node in the two paths that is common to both.

Another way to find the LCA of two nodes is to use recursion. Starting from the root node, recursively search for the two nodes in the left and right subtrees. If one node is found in the left subtree and the other is found in the right subtree, then the root node is the LCA. Otherwise, continue the search in the appropriate subtree.

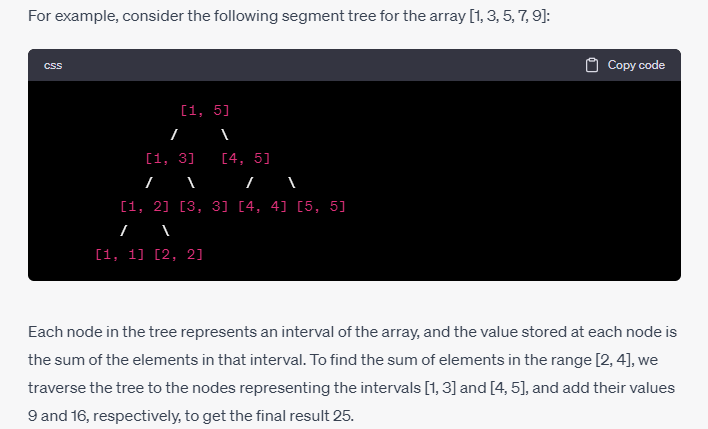
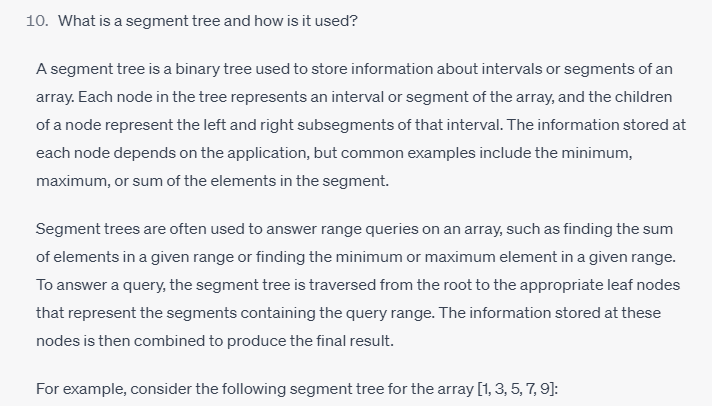
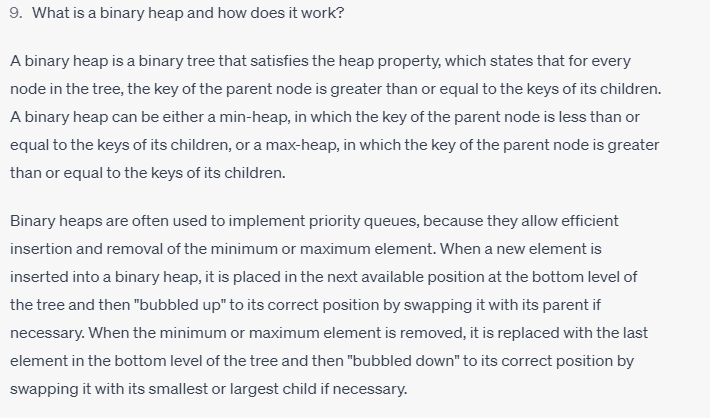
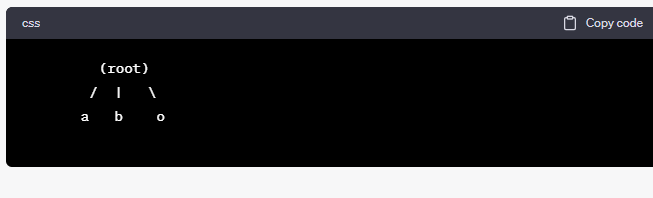
For example, consider the binary tree:

1. What is a trie and how is it used?

A trie (pronounced "try") is a tree-like data structure used to store a collection of strings. Each node in a trie represents a prefix or a complete word, and the children of a node represent the possible next characters in the string.

Tries are often used in applications such as spell-checking, autocompletion, and IP routing. They are particularly efficient for searching for strings with a common prefix, because all strings with the same prefix share the same initial path in the trie.

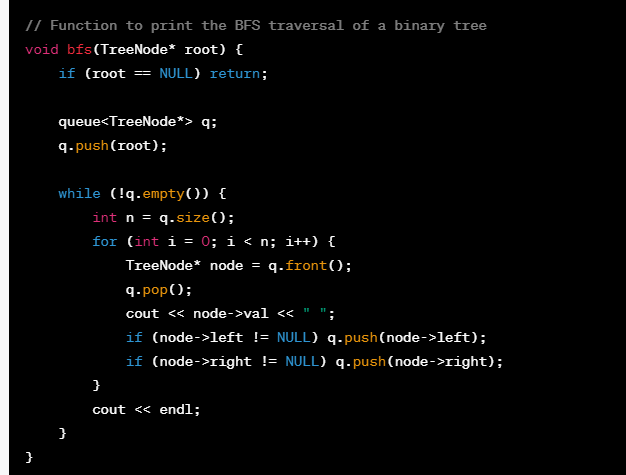
For example, consider the following trie that stores the words "apple", "banana", "orange", and "pear":

Top of Form

# Height of a Binary Tree / Maximum depth of a binary tree Algorithm



**Print Breath First Search**



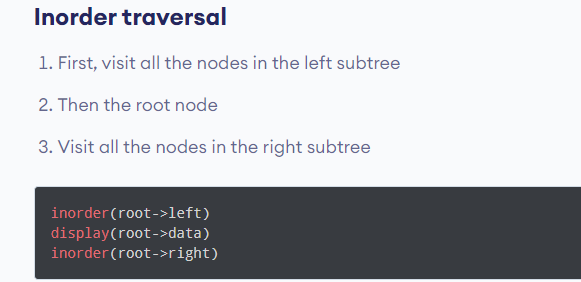
**Print Depth First Search**

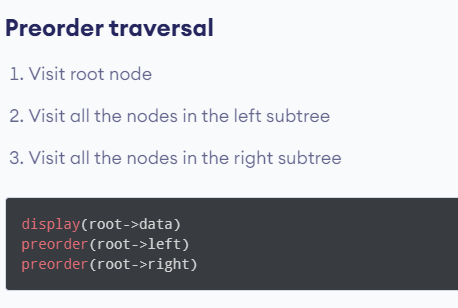
<https://www.programiz.com/dsa/graph-dfs>

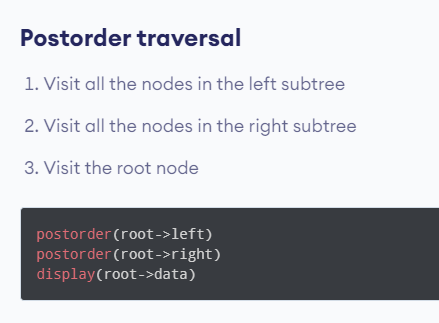
depth first search ma stack use hota ha

breath first search ma queue use hota ha.ye level order traversal hota ha

**Print Tree Traversal**





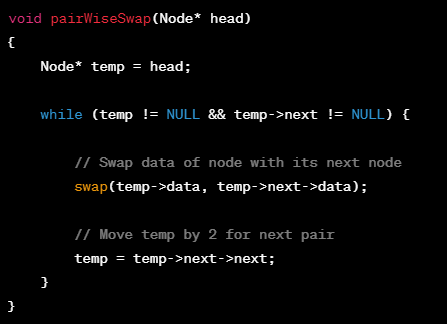


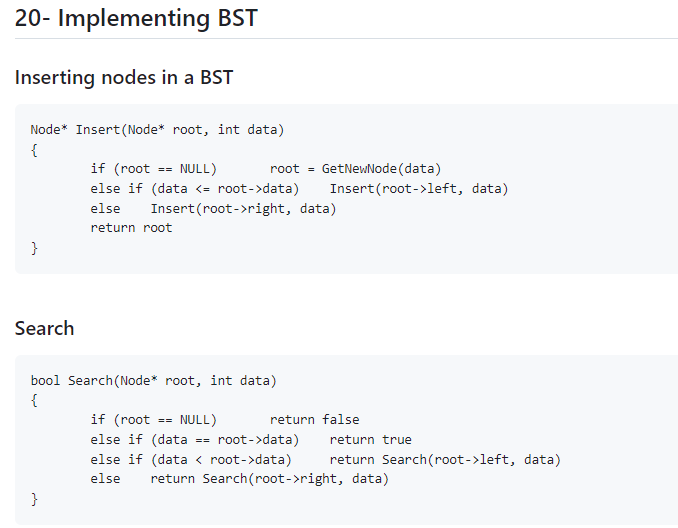
## **Making a queue using two stacks**

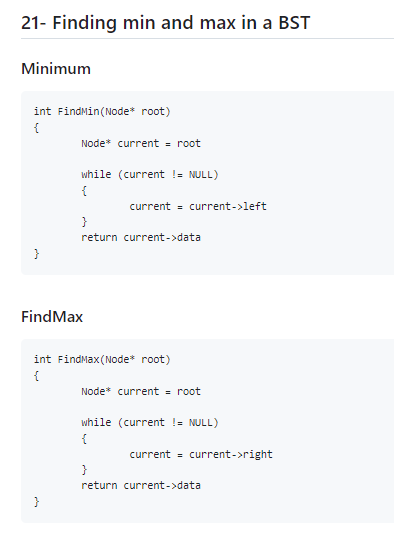
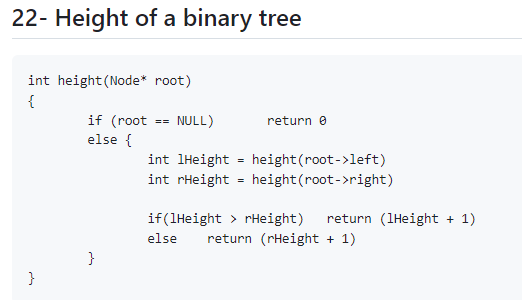
## **Pair wise swaping a linked list**

In this code, the **swap()** function is used to swap the data of the nodes, and the loop iterates over the linked list by moving the **temp** pointer by 2 nodes at a time. The **temp** pointer is initialized to the **head** of the linked list, and the loop continues until **temp** becomes **NULL** or **temp->next** becomes **NULL**.

Within the loop, the **swap()** function is used to swap the data of the current node with its next node. Then, the **temp** pointer is moved by 2 nodes to point to the next pair of nodes for swapping.



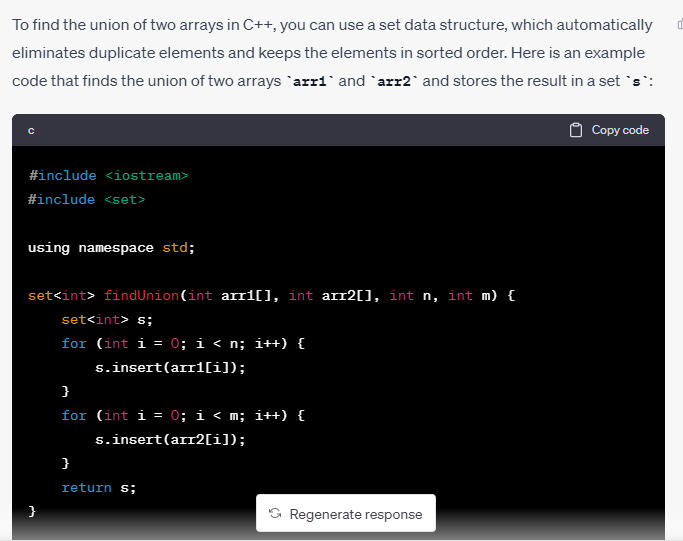


## **Check for pairs in an array with sum as x.**

## **Given a function that generates a random number between 20-50, write a function that generates a random number between 10-70.**

## **Finding union between 2 arrays**

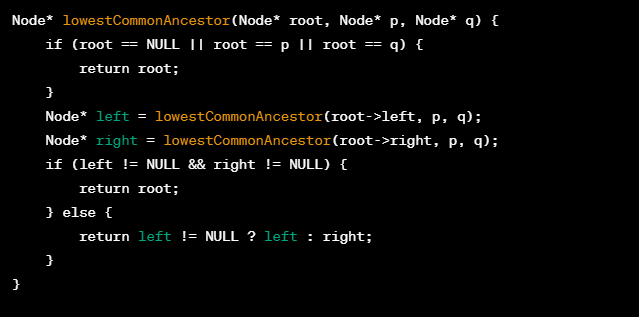


**Find the lca of two nodes in binary tree**

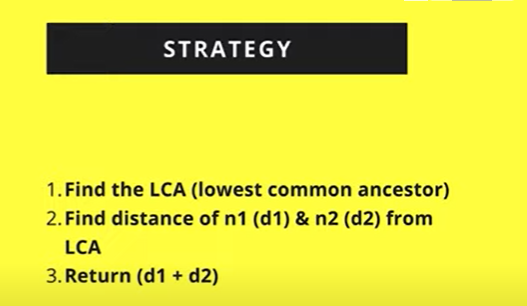
To find the lowest common ancestor (LCA) of two nodes in a binary tree, you can follow these steps:

1. Start at the root of the tree.
2. Check if the root is NULL or equal to either of the two nodes. If it is, return the root.
3. Recursively search the left subtree for one of the two nodes. If the node is found, return it.
4. Recursively search the right subtree for one of the two nodes. If the node is found, return it.
5. If both nodes were found in different subtrees, return the root.

Here's the C++ code to find the LCA of two nodes in a binary tree:



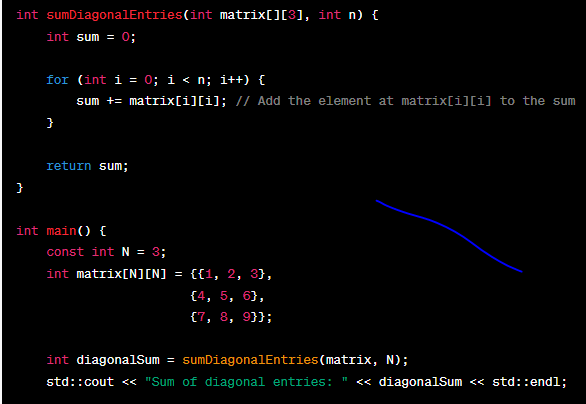
**Find the shortest distance between two nodes in tree**



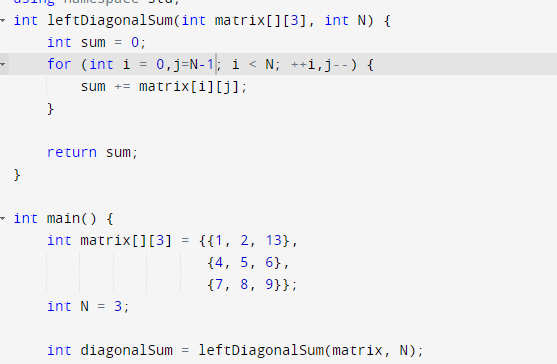
**Q count total number of leaf nodes in a tree**



**Q print sum of right diagonal enteries**



**Q print sum of left diagonal enteries**



**Q Longest Sub-Array with Sum K**