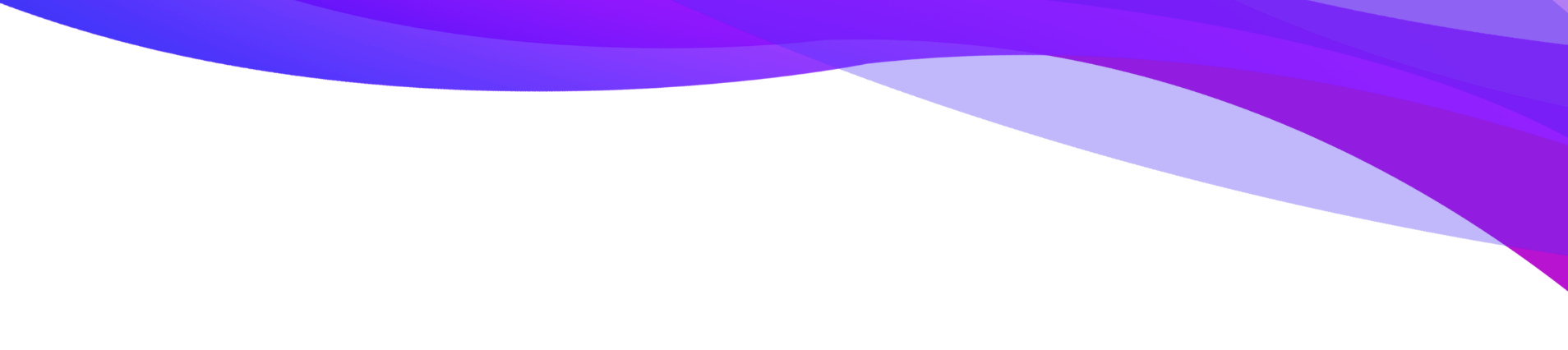
****

**DELHI TECHNOLOGICAL UNIVERSITY**

(Formerly Delhi College of Engineering)

Shahbad Daulatpur, Bawana Road, Delhi- 110042

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**Lab Code: CO-102**

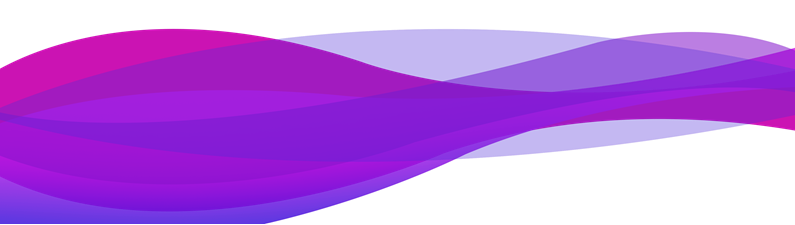
**DATA STRUCTURES LAB FILE**

**CO-201**

**Submitted To:**

**Dr. Ashish Girdhar**

**Department of COE**

****

**Submitted By:**

**Sandesh Shrestha**

**2K21/CO/417**

**A6 BATCH**

**INDEX: LIST OF PROGRAMS**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No.** | **Program Name** | **Date of Experiment** | **Remarks** |
| 1) | Write a program to Implement Linear Search in the C programming language | 23.08.2022 |  |
| 2) | Write a program to Implement Binary Search in the C programming language. Assume the list is already sorted. | 23.08.2022 |  |
| 3) | Write a program to insert an element at the mid-position in the One-dimensional array. | 23.08.2022 |  |
| 4) | Write a program to insert & delete a given row in a two-dimensional array. | 24.08.2022 |  |
| 5) | Write a program to implement a stack data structure and perform its operations - | 06.09.2022 |  |
| 6) | Write a program to implement two stacks using a single array. | 06.09.2022 |  |
| 7) | Write a program to implement Queue Data structure with its functions | 13.09.2022 |  |
| 8) | Write a program to find the minimum element of the stack in constant time using extra space. Assume that elements are being pushed onto the stack with user input, not with a pre-formed stack. | 13.09.2022 |  |
| 9) | Write a program to find the minimum element of the stack without using extra space. Assume that elements are being pushed onto the stack with user input, not with a pre-formed stack | 20.09.2022 |  |
| 10) | Write a program to reverse the first k elements of a given Queue | 27.09.2022 |  |
| 11) | Write a program to check whether the given string is Palindrome or not, using DEQUEUE | 27.09.2022 |  |
| 12) | Implement Tower of Hanoi Problem using Stack. | 11.10.2022 |  |
| 13) | Write a program to implement the Linked List Data structure and insert a new node at the beginning, and at a given position | 18.10.2022 |  |
| 14) | Write a program to split a given linked list into two sub-list as Front sub-list and back sub-list, if odd number of the element, then add the last element into the front list. | 18.10.2022 |  |
| 15) | Given a Sorted doubly linked list of positive integers and an integer, then finds all the pairs (sum of two nodes data part) that is equal to the given integer value. | 18.10.2022 |  |
| 16) | Write a program to implement the Binary Tree using linked list and perform In-order traversal. | 25.10.2022 |  |
| 17) | Write a Program to check whether the given tree is a Binary Search Tree or not. | 25.10.2022 |  |
| 18) | Write a program to implement insertion in the AVL tree. | 25.10.2022 |  |
| 19) | Write an Algorithm to count the number of leaf nodes in an AVL tree | 25.10.2022 |  |
| 20) | Write a program to Delete a key from the AVL tree | 25.10.2022 |  |
| 21) | Write a program to implement Stack Data Structure using Queue. | 01.11.2022 |  |
| 22) | Write a program to implement Queue Data Structure using Stack. | 01.11.2022 |  |
| 23) | Write a program to implement Graph Data Structure and Its traversal BFS and DFS. | 01.11.2022 |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |
|  |  |  |
|  |  |  |

**Program 1**

**Program Objective: Program to implement linear search in C programming language.**

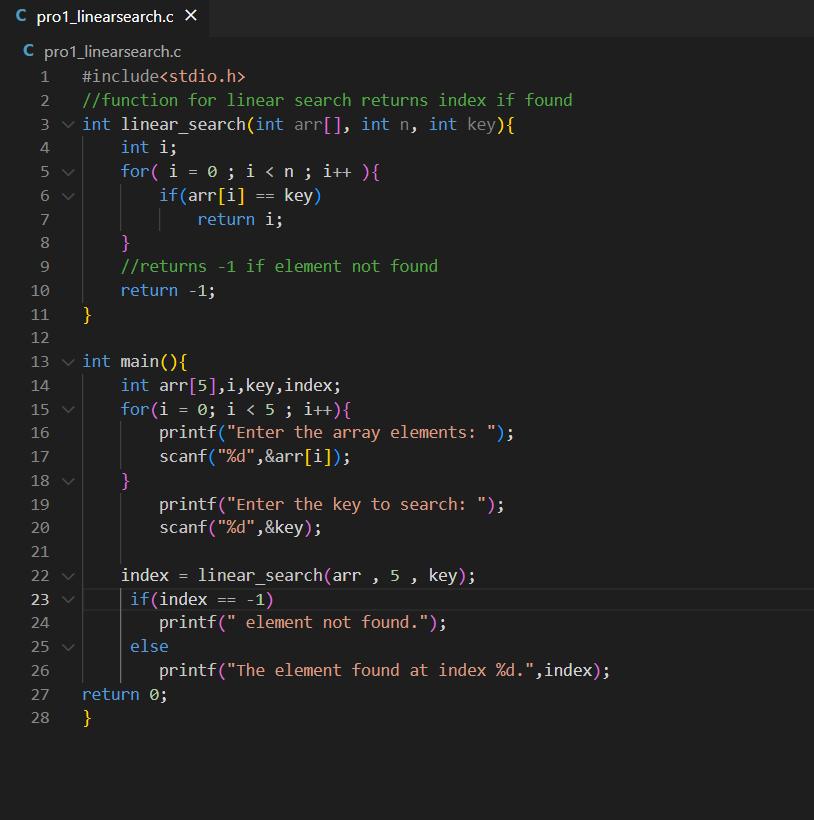
**Program theory**: A linear search, also known as a sequential search, is a method of finding an element within a list. It checks each element of the list sequentially until a match is found or the whole list has been searched.

**Algorithm**: A simple approach to implement a linear search is:

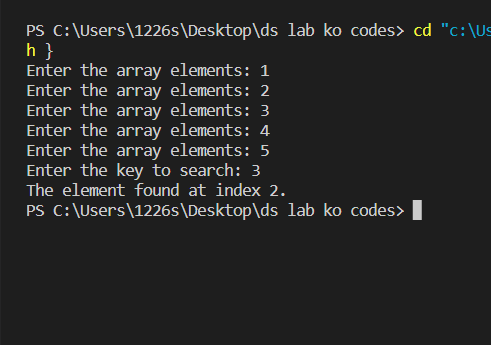
* Begin with the leftmost element of array and one by one compare x with each element.
* If x matches with an element, then return the index.
* If x does not match with any of the elements, then return -1.

The **time complexity** of a linear search is **O(n) in worst case and O(1) in best case.**

**Program Code:**

****

**Program Output:**

****

**Program 2**

**Program Objective: Write a program to Implement Binary Search in the C programming language. Assume the list is already sorted.**

**Program theory**: Binary search is a 'divide and conquer' algorithm which requires the initial array to be sorted before searching. It is called binary because it splits the array into two halves as part of the algorithm. Initially, a binary search will look at the item in the middle of the array and compare it to the search terms. Binary search can be implemented on sorted array elements. If the list elements are not arranged in a sorted manner, we have first to sort them.

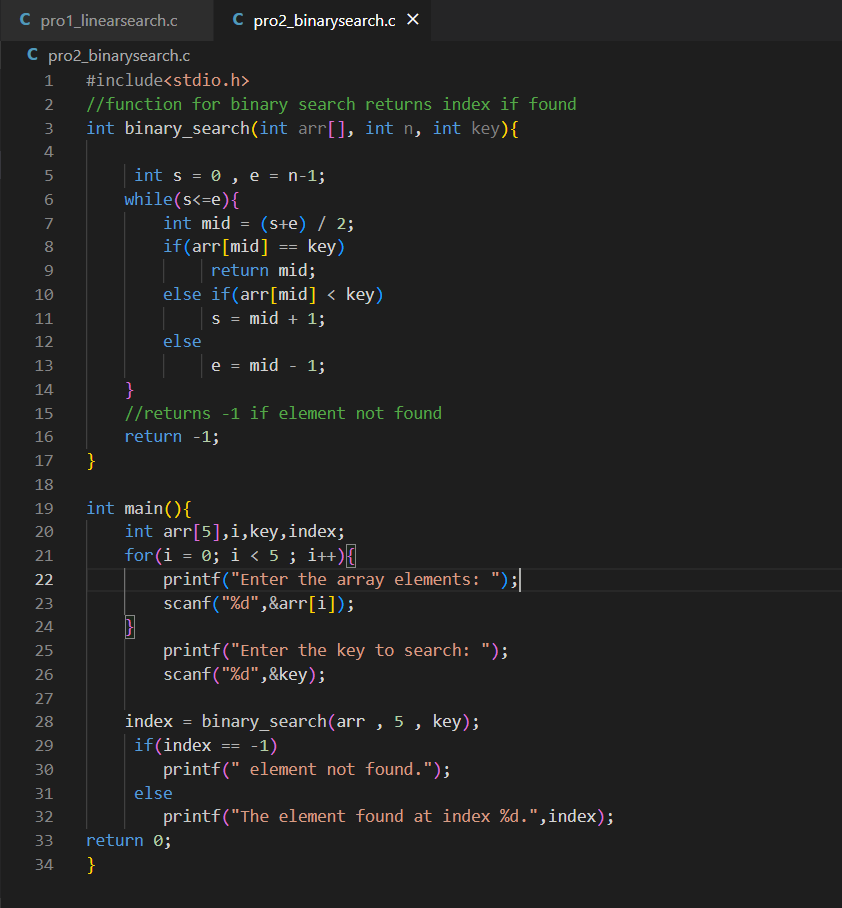
**Algorithm**: The basic steps to perform Binary Search are:

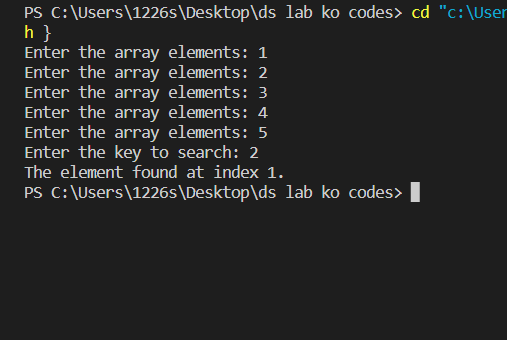
* Begin with the mid element of the whole array as a search key.
* If the value of the search key is equal to the item, then return an index of the search key.
* Or if the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
* Otherwise, narrow it to the upper half.
* Repeatedly check from the second point until the value is found or the interval is empty.

The **time complexity** of a binary search is:

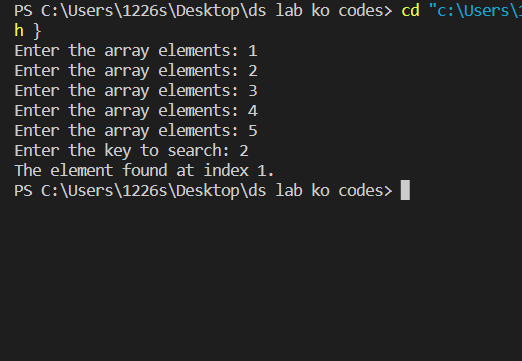
|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| Best Case | O(1) |
| Average Case | O(logn) |
| Worst Case | O(logn) |

**Program Code:**

****

****

**Program Output:**

****

**Program 3**

**Program Objective: Write a program to insert an element at the mid-position in the One-dimensional array in C programming language.**

**Program theory**: Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. In this scenario, we are given the exact location i.e., middle (index) of an array where a new data element (value) needs to be inserted. First, we shall check if the array is full, if it is not, then we shall move all data elements from middle location one step downward. This will make place for a new data element.

**Algorithm**: The basic steps to perform insertion operation are:

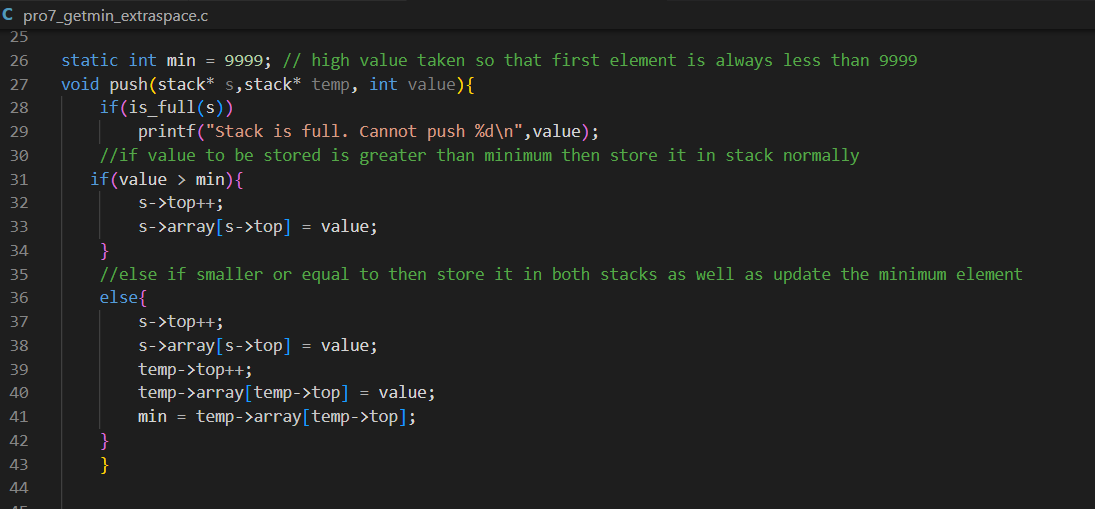
* First get the element to be inserted, say x
* Then shift the array elements from middle position to one position forward, and do this for all the other elements next to pos.
* Insert the element x now at the position mid, as this is now empty.

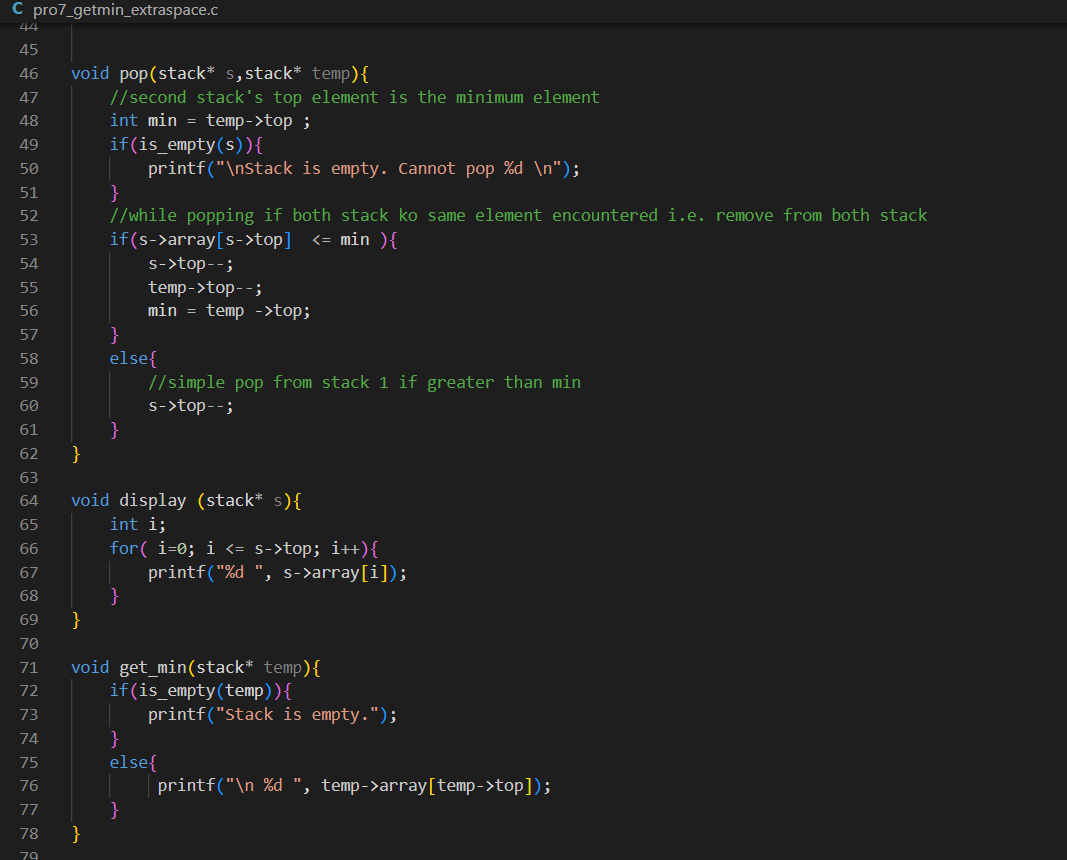
The **time complexity** of a binary search is:

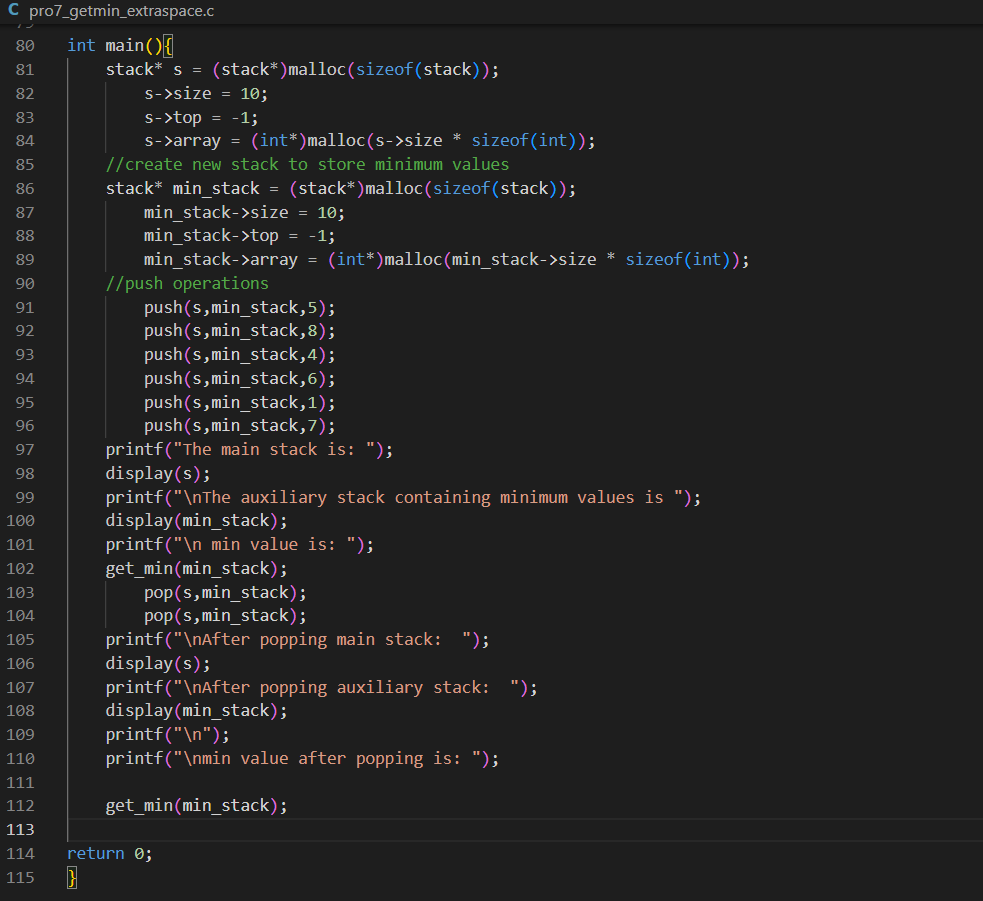
|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| Best Case | O(1) |
| Average Case | O(n) |
| Worst Case | O(n) |

**Program Code:**

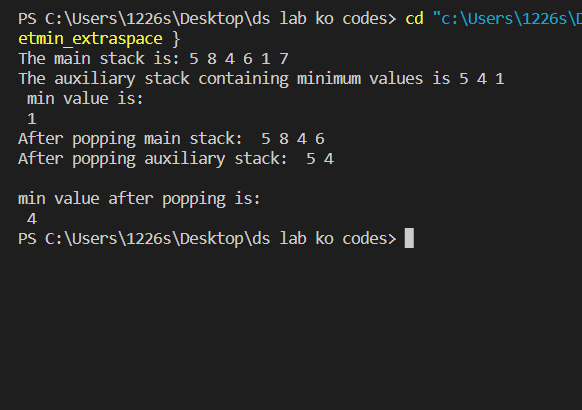








**Program Output:**



**Program 4**

**Program Objective: Write a program to insert & delete a given row in a two-dimensional array.**

**Program theory**: The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

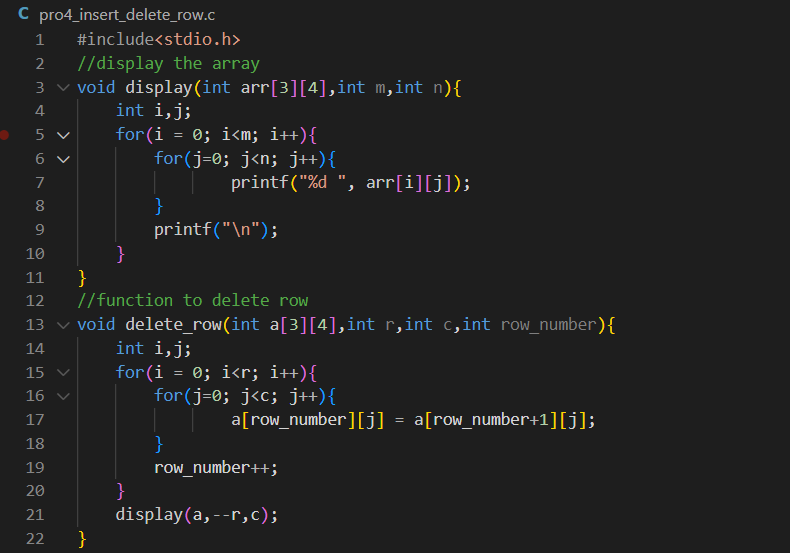
**Algorithm** **Insert Row**: Create a new array of one more element in row size and another one-dimensional array to store the row to insert. Then there are three cases: If row that we currently are on is less than the row index to insert in, copy everything as it is in new array. If row is equal to row index to insert, then insert the new row. Then after that, copy everything after that row from the input (shifted back one against the current row index).

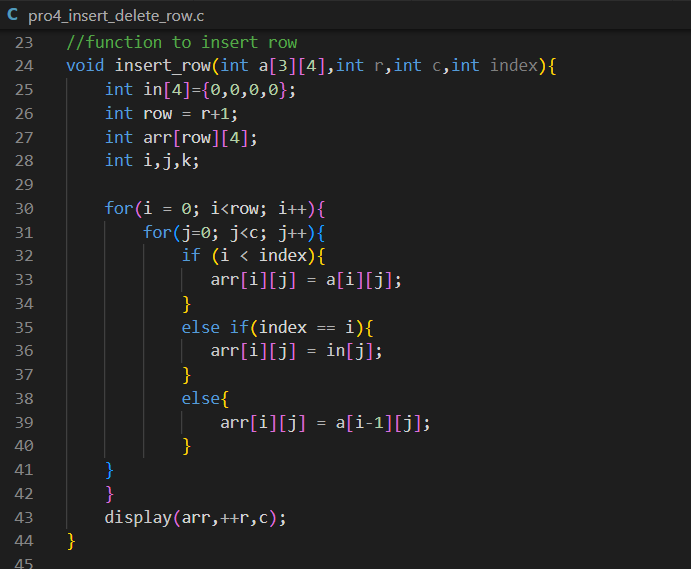
**Algorithm** **delete Row**: To delete a row, first we need to get the index of row to be deleted consider it as index. Then, we need to shift elements one index above from the deleted row index and then decrease the row number.

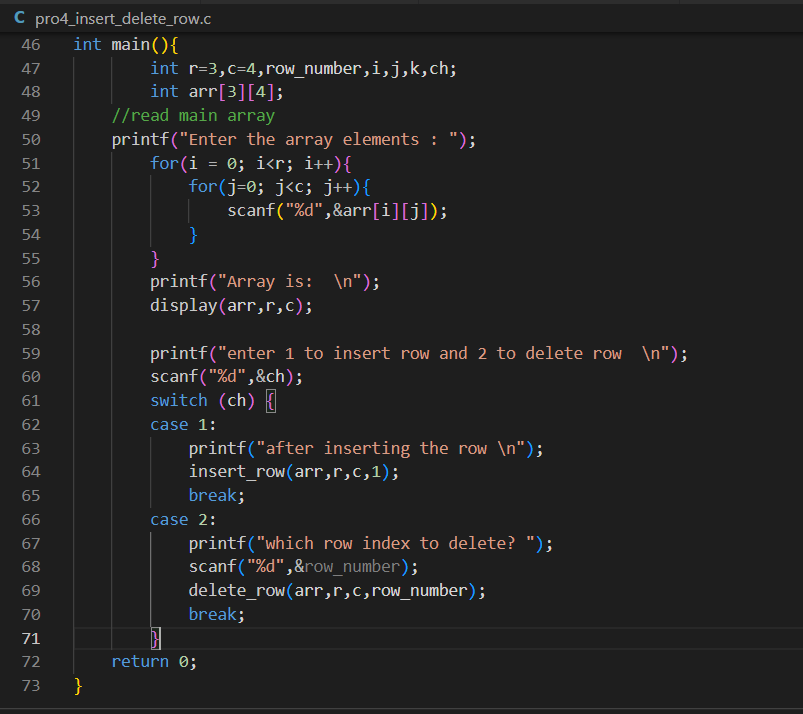
The **time complexity** is:

|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| Best Case | O(1) |
| Average Case | O(n) |
| Worst Case | O(n) |

**Program Code:**

****





**Program Output:**

****

**Program 5**

**Program Objective: Write a program to implement a stack data structure and perform its operations -**

1. **Push(): Inserts an element into the stack**
2. **Pop(): Removes an element from the stack**
3. **Top(): Returns the top element of the stack**
4. **Size(): Returns the size of stack**

**Program theory**: A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stack has one end, whereas the Queue has two ends (front and rear). It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack. The following are some common operations implemented on the stack:

* push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
* pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
* isEmpty(): It determines whether the stack is empty or not.
* isFull(): It determines whether the stack is full or not.'
* peek(): It returns the element at the given position.
* count(): It returns the total number of elements available in a stack.
* change(): It changes the element at the given position.
* display(): It prints all the elements available in the stack.

**Algorithm:**

The steps involved in the **PUSH** operation is given below:

* Before inserting an element in a stack, we check whether the stack is full.
* If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
* When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
* When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., top=top+1, and the element will be placed at the new position of the top.
* The elements will be inserted until we reach the max size of the stack.

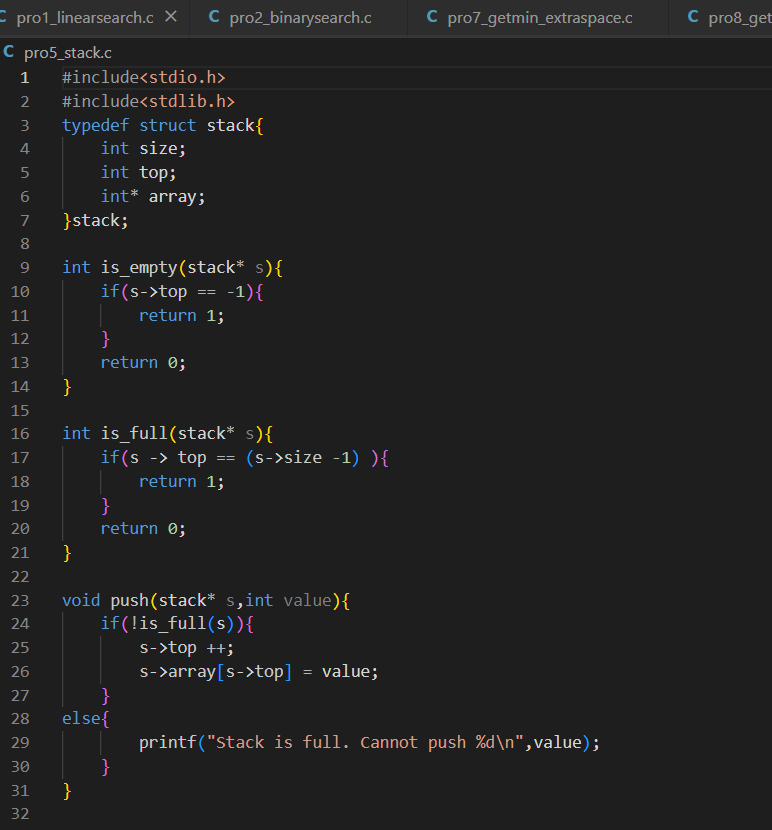
The steps involved in the **POP** operation is given below:

* Before deleting the element from the stack, we check whether the stack is empty.
* If we try to delete the element from the empty stack, then the underflow condition occurs.
* If the stack is not empty, we first access the element which is pointed by the top
* Once the pop operation is performed, the top is decremented by 1, i.e., top=top-1.

The **time complexity** is:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OPERATION** | **BEST TIME COMPLEXITY** | **WORST TIME COMPLEXITY** | **AVERAGE TIME COMPLEXITY** | **SPACE COMPLEXITY** |
| Push | O(1) | O(N) | O(1) | O(1) |
| Pop | O(1) | O(1) | O(1) | O(1) |
| Peek | O(1) | O(1) | O(1) | O(1) |

**Program Code:**





**Program Output:**

****

**Program 6**

**Program Objective: Write a program to implement two stacks using a single array.**

**Program theory**: The stack data structure is a linear data structure that accompanies a principle known as LIFO (Last In First Out) or FILO (First In Last Out). In this case, stack1 starts from 0 while stack2 starts from n-1. Both the stacks start from the extreme corners, i.e., Stack1 starts from the leftmost corner (at index 0), and Stack2 starts from the rightmost corner (at index n-1). Stack1 extends in the right direction, and stack2 extends in the left direction. In this case, the stack overflow condition occurs only when top1 + 1 = top2. This approach provides a space-efficient implementation means that when the array is full, then only it will show the overflow error. In contrast, the first approach shows the overflow error even if the array is not full.

**Following functions must be supported by twoStacks:**

push1(int x) –> pushes x to first stack

push2(int x) –> pushes x to second stack

pop1() –> pops an element from first stack and return the popped element

pop2() –> pops an element from second stack and return the popped element

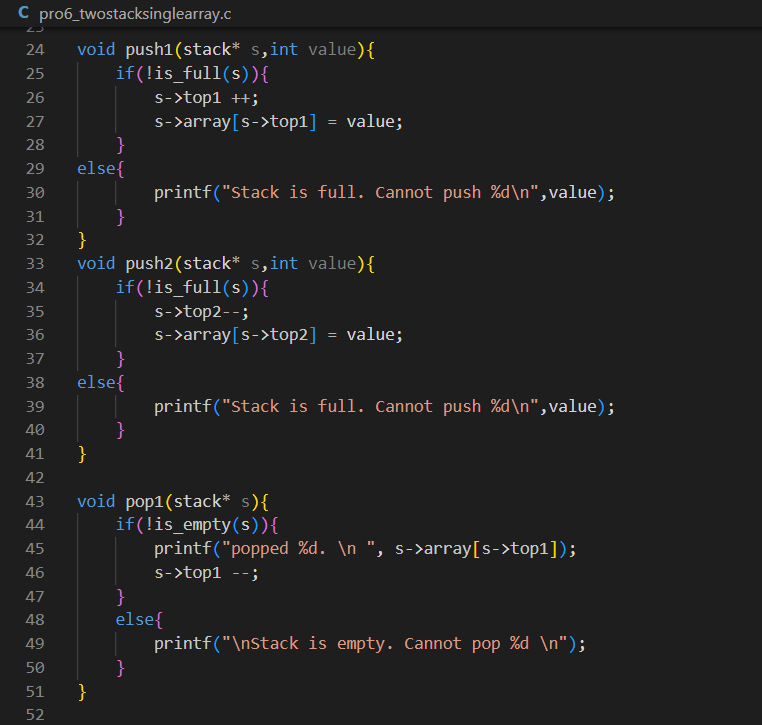
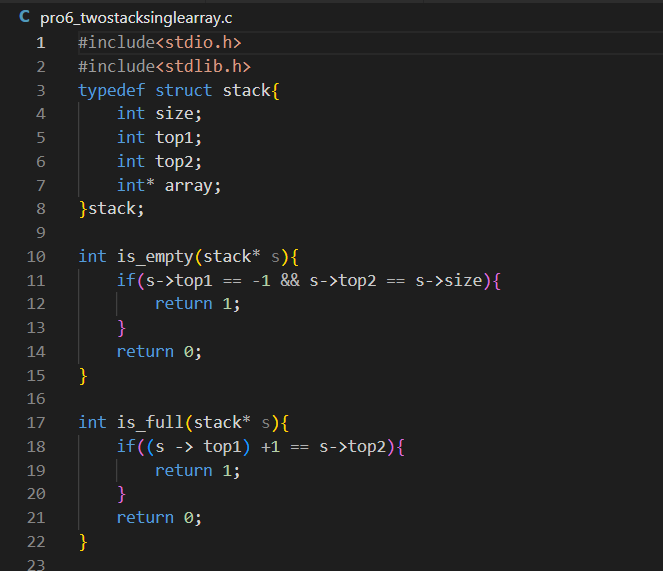
**Algorithm:**

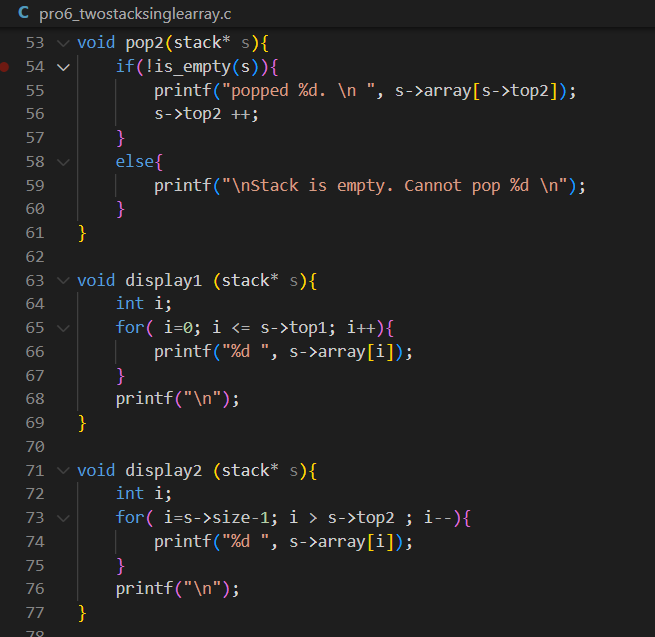
Follow the steps below to solve the problem:

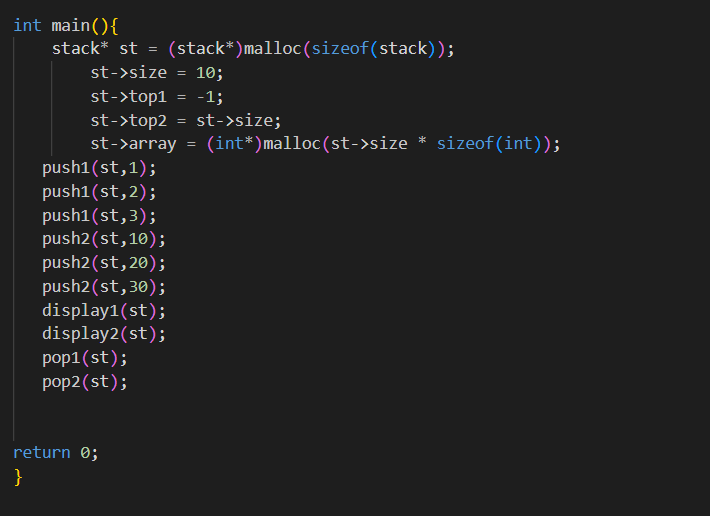
* Stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0.
* Stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1).
* Both stacks grow (or shrink) in opposite directions.
* To check for overflow, all we need to check is for space between top elements of both stacks. In this case, the stack overflow condition occurs only when top1 + 1 = top2.

The **time complexity** is:

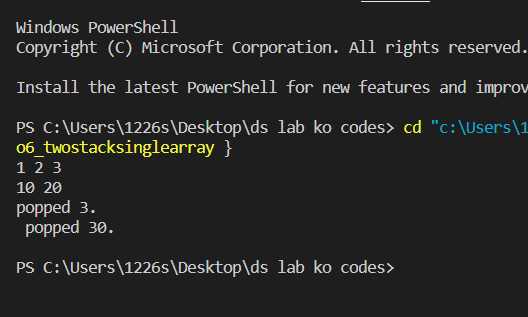
|  |  |
| --- | --- |
| **Case** | **Time Complexity** |
| Best Case | O(1) |
| Average Case | O(n) |
| Worst Case | O(n) |

**Program Code: **

****

****

**Program Output:**

****

**Program 7**

**Program Objective: Write a program to implement Queue Data structure with its functions - Enqueue, Dequeue, IsEmpty, IsFull as below:**

1. **Enqueue(): Inserts an element at the end of the queue i.e. at the rear end**
2. **Dequeue(): Removes and returns an element that is at the front end of the queue**
3. **IsEmpty(): Returns 1 if queue is empty & 0 if not**
4. **IsFull (): Returns 1 if queue is full & 0 if not**

**Program theory**: Queue follows the First In First Out (FIFO) rule - the item that goes in first is the item that comes out first. A queue can be defined as an ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.

A queue is an object (an abstract data structure - ADT) that allows the following operations:

* Enqueue: Add an element to the end of the queue
* Dequeue: Remove an element from the front of the queue
* IsEmpty: Check if the queue is empty
* IsFull: Check if the queue is full
* Peek: Get the value of the front of the queue without removing it

**Enqueue Operation**

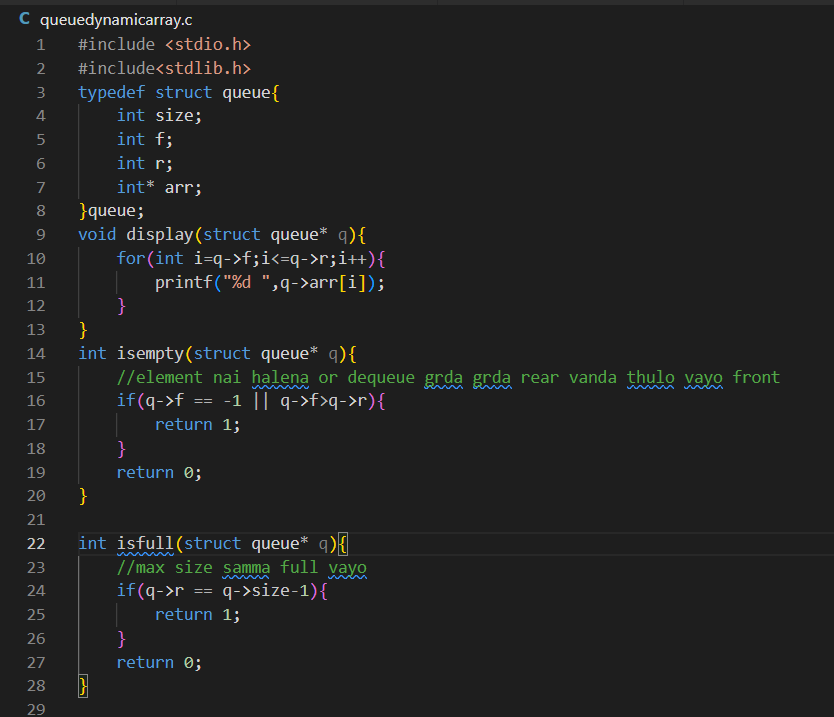
* check if the queue is full
* for the first element, set the value of FRONT to 0
* increase the REAR index by 1
* add the new element in the position pointed to by REAR

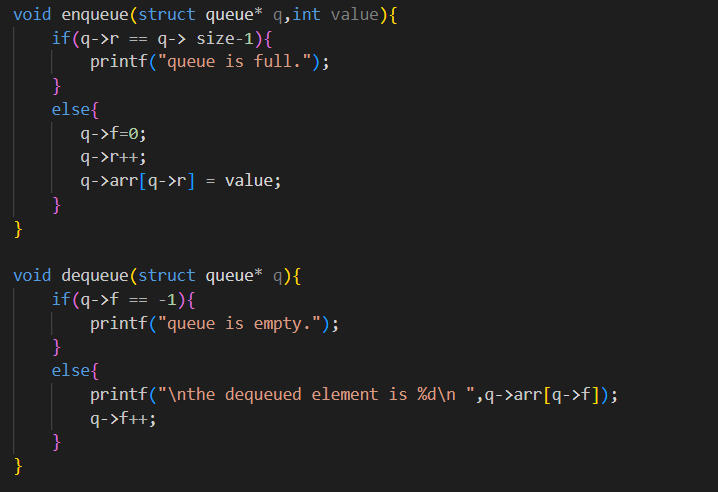
**Dequeue Operation**

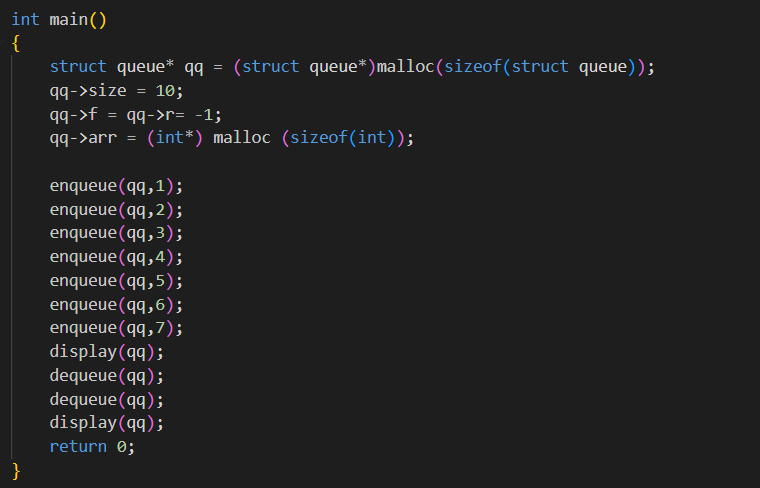
* check if the queue is empty
* return the value pointed by FRONT
* increase the FRONT index by 1
* for the last element, reset the values of FRONT and REAR to -1

The complexity of enqueue and dequeue operations in a queue using an array is O(1).

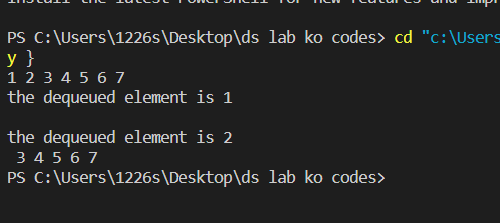
**Program Code:**

****

****

****

**Program Output:**

****

**Program 8**

**Program Objective: Write a program to find the minimum element of the stack in constant time using extra space. Assume that elements are being pushed onto the stack with user input, not with a pre-formed stack.**

**Program theory**: The correct approach uses two stacks – the main stack to store the actual stack elements and an auxiliary stack to store the determine the minimum value so that we can get\_min in constant time by popping the top of auxiliary stack. This implementation requires few changes in push and pop operations.

1. Push operation

The idea is to push the new element into the main stack and push it into the auxiliary stack only if the stack is empty or the new element is less than or equal to the current minimum or top element of the auxiliary stack.

2. Pop operation

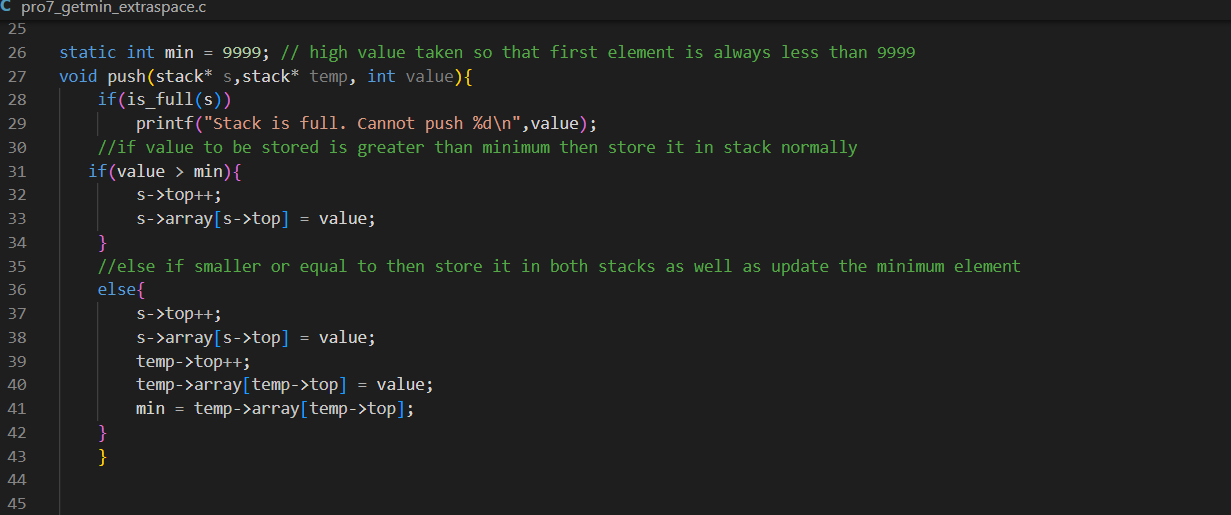
For pop operation, remove the top element from the main stack and remove it from the auxiliary stack only if it is equal to the current minimum element, i.e., a top element of both the main stack and the auxiliary stack is the same. After the minimum number is popped, the next minimum number appears on the top of the auxiliary stack.

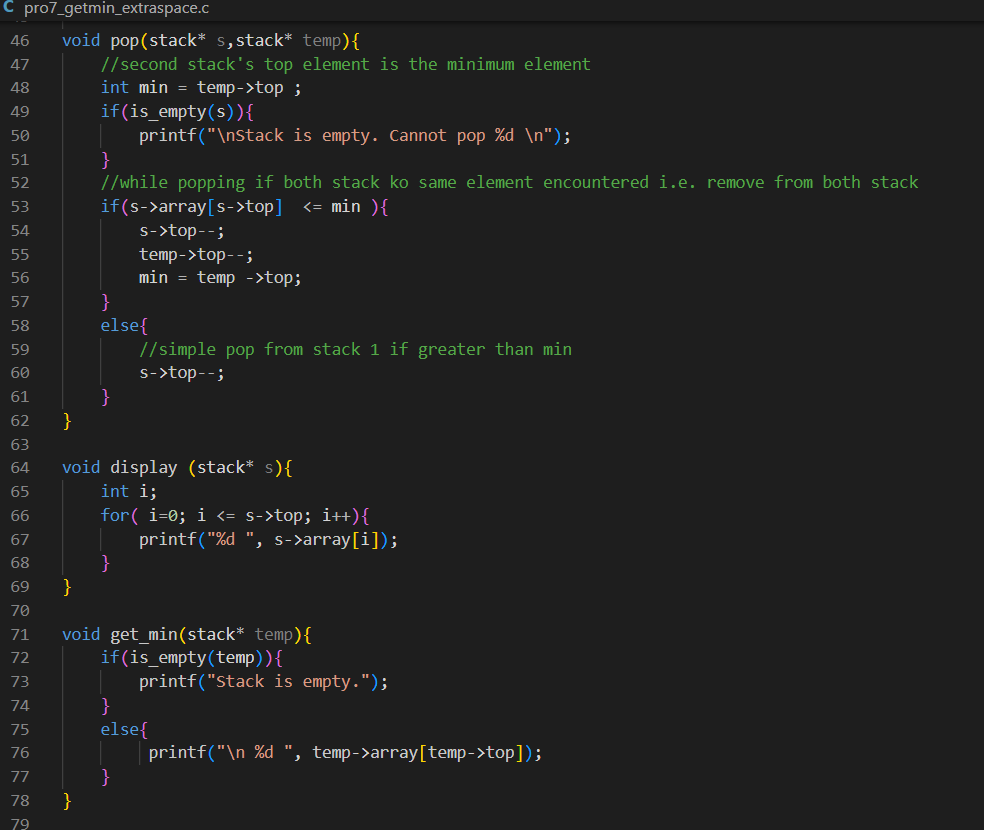
3. Min operation

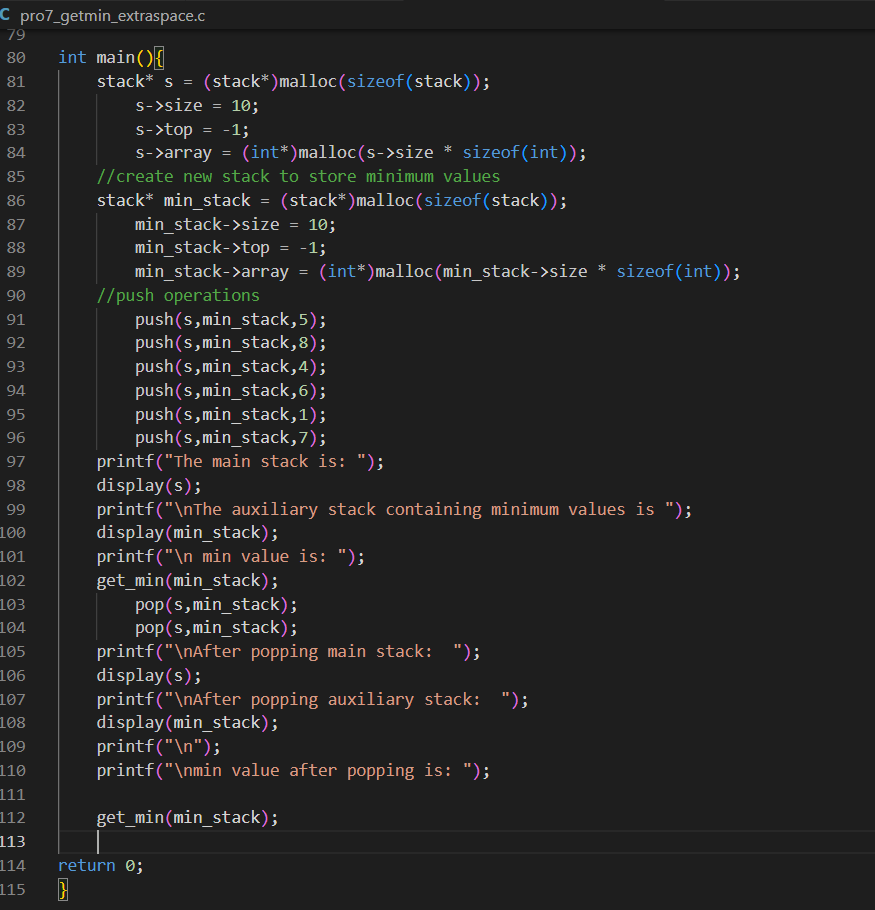
The top of the auxiliary stack always returns the minimum number since we are pushing the minimum number into the auxiliary stack and removing the minimum number from the auxiliary stack only if it is removed from the main stack.

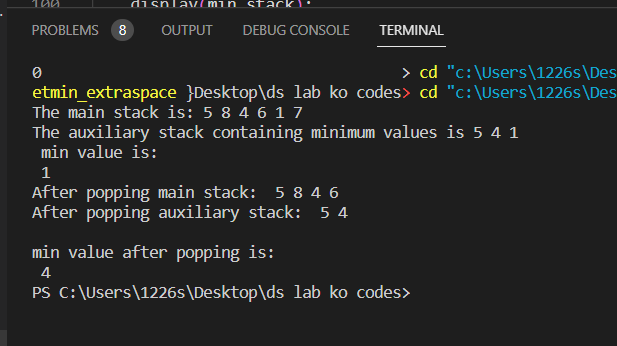
**Program Code:**

****

****

****

****

**Program Output:** ****

**Program 9**

**Program Objective: Write a program to find the minimum element of the stack without using extra space. Assume that elements are being pushed onto the stack with user input, not with a pre-formed stack**

**Program theory**: Suppose we will push a number value into a stack with a minimum number, min. If the value is greater than or equal to the min, it is pushed directly into the stack. If it is less than min, push 2×value-min, and update min as a value since a new minimum number is pushed. To pop, we pop it directly if the top of the stack (it is denoted as top) is greater than or equal to min. Otherwise, the number top is not the real pushed number. The real pushed number is stored as min. After the current minimum number is popped, we need to restore the previous minimum number, 2×min-top.

1. Push operation

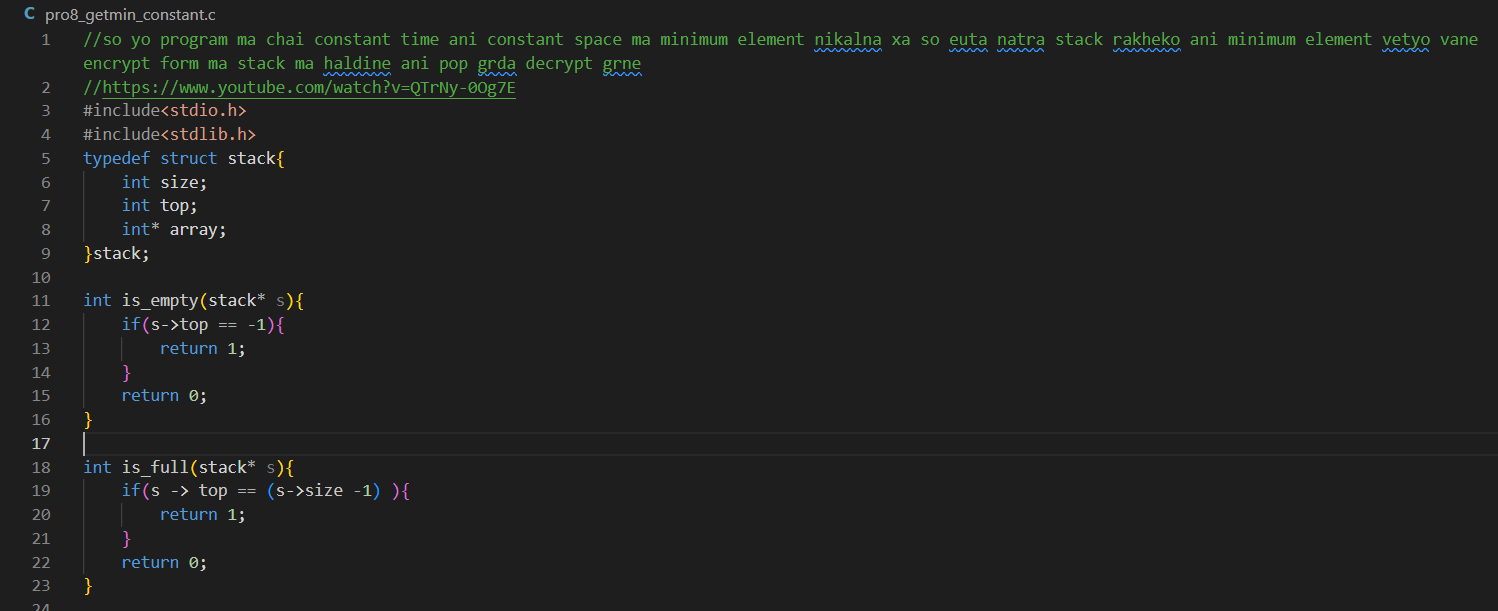
The idea is to push the new element into the stack and replace **static** integer minimum which is initially declared as 9999 as the first element in stack. Then after that, if the next element is greater than current min simply push it. Else if it is less than or equal to the minimum then push 2\*value – min into stack.

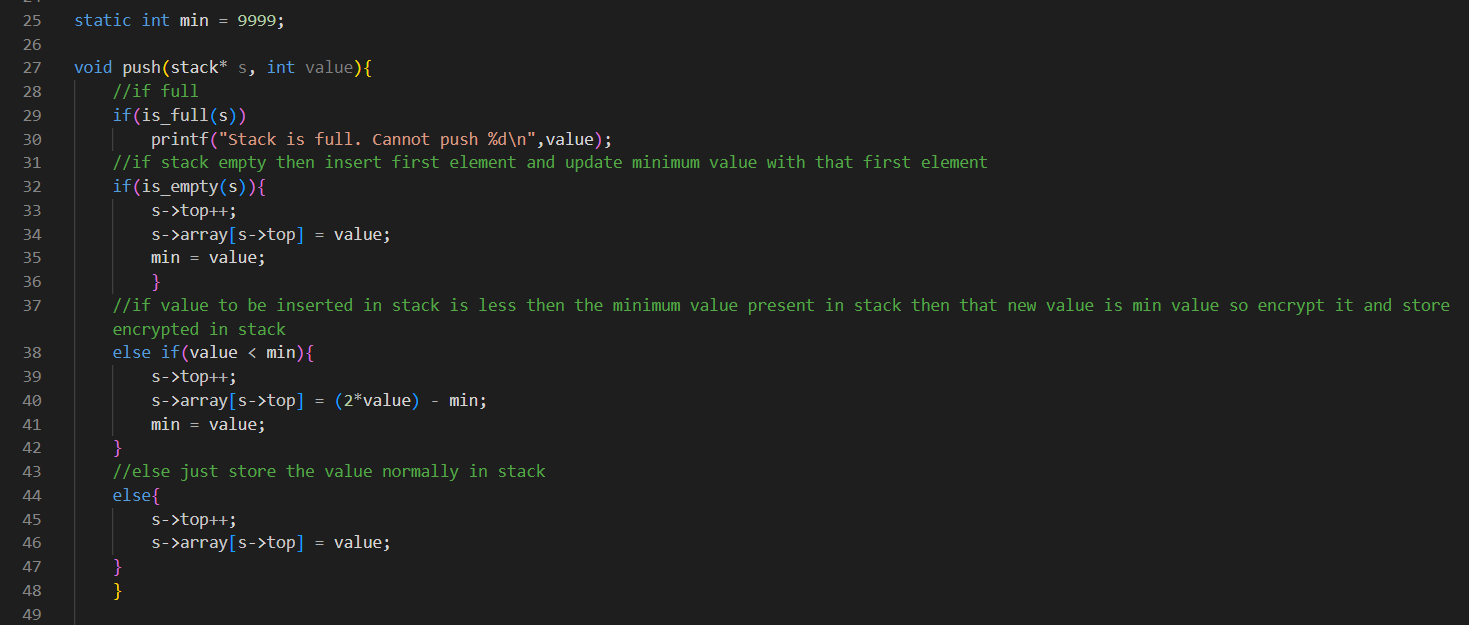
2. Pop operation

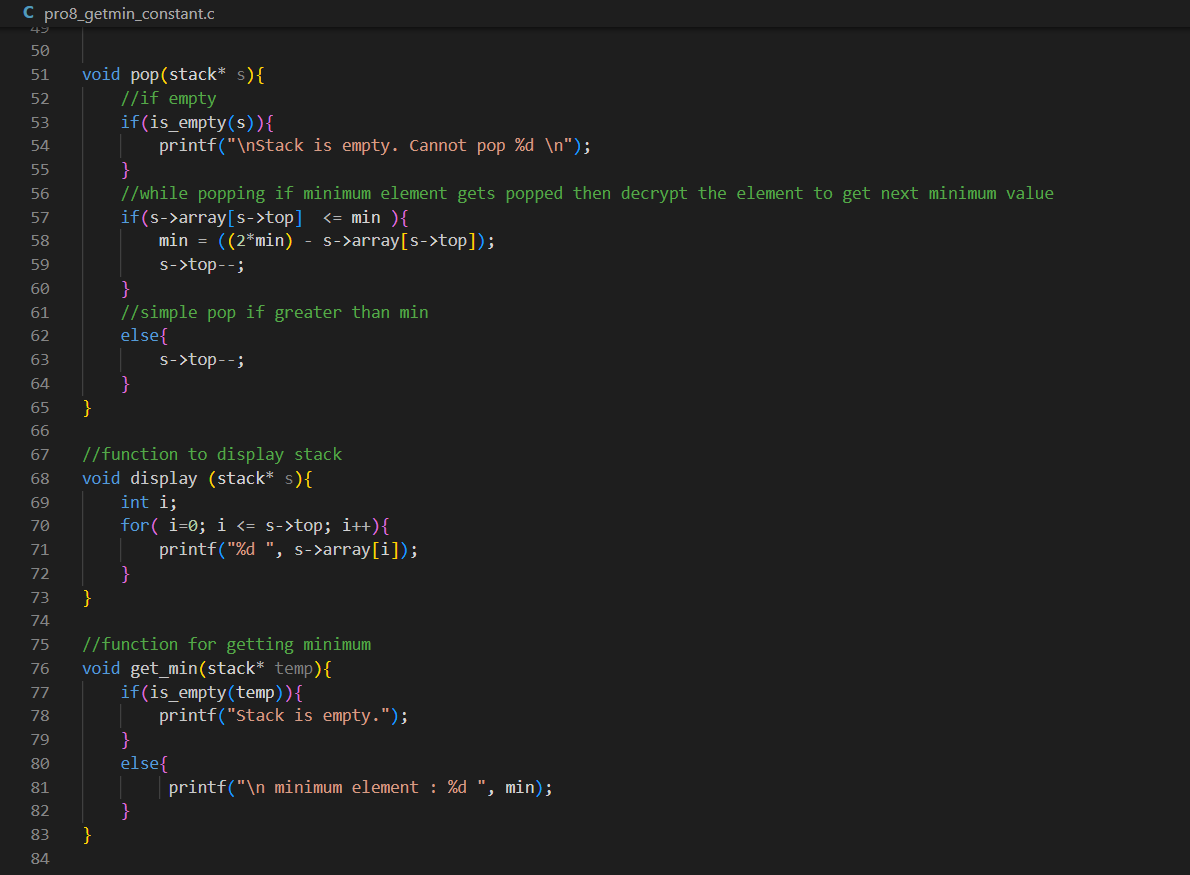
For pop operation, remove the top element from the stack if it is greater than min. Else if top element is smaller than the min then new min will be 2\*min – top element value. Then pop the top element. Done.

3. Min operation

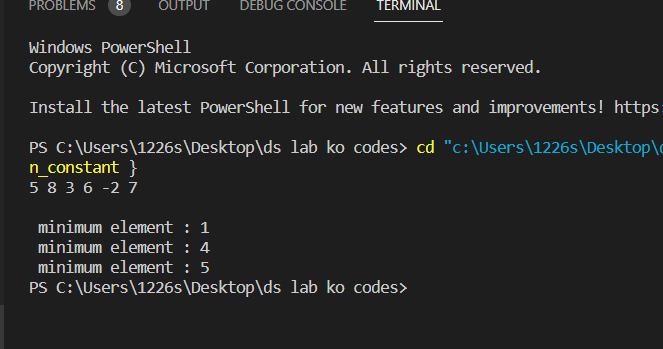
The static integer maintained will give the value of min value in stack. Simply print it out.

**Program Code: **







**Program Output: **

**Program 10**

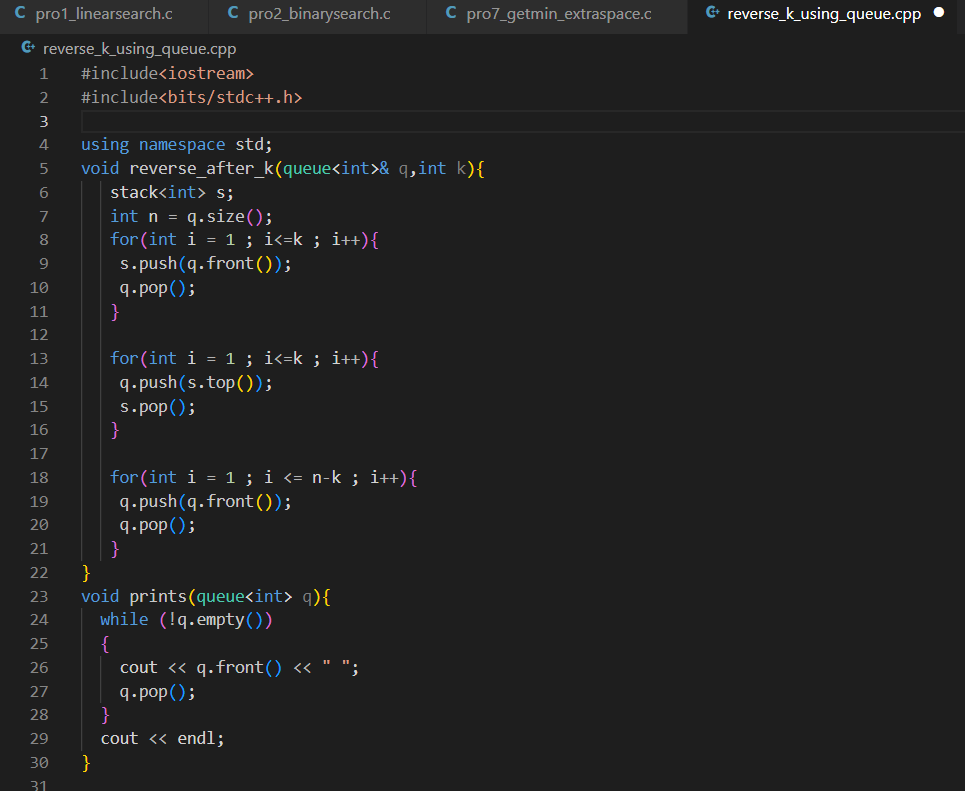
**Program Objective: Write a program to reverse the first k elements of a given Queue**

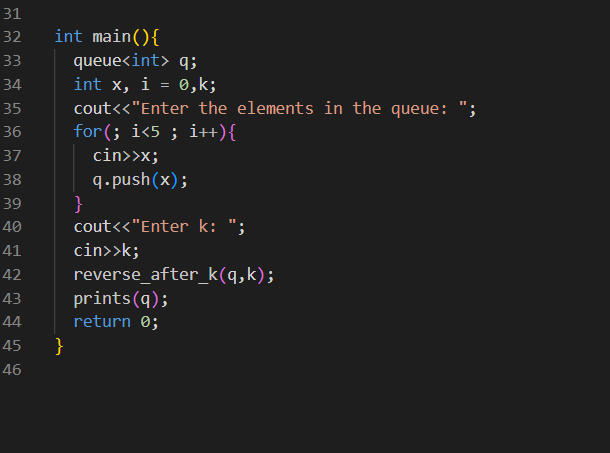
**Program theory**: The idea is to use an auxiliary stack. Store the first k elements of the queue in a stack and pop it from the queue, then push it back to the queue and perform pop operation for n-k times and again push the popped element.

**Algorithm**:

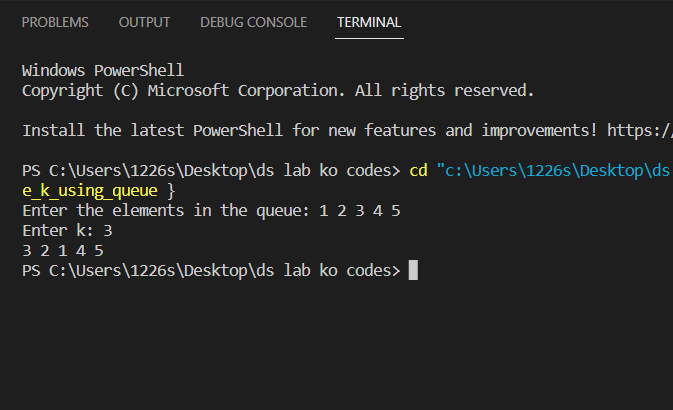
* Create an empty stack.
* One by one dequeue first K items from given queue and push the dequeued items to stack.
* Enqueue the contents of stack at the back of the queue
* Dequeue (size - k) i.e. remaining elements from the front and enqueue them one by one to the same queue from rear.
* Done.

**Program Code:**

****

****

**Program Output:**

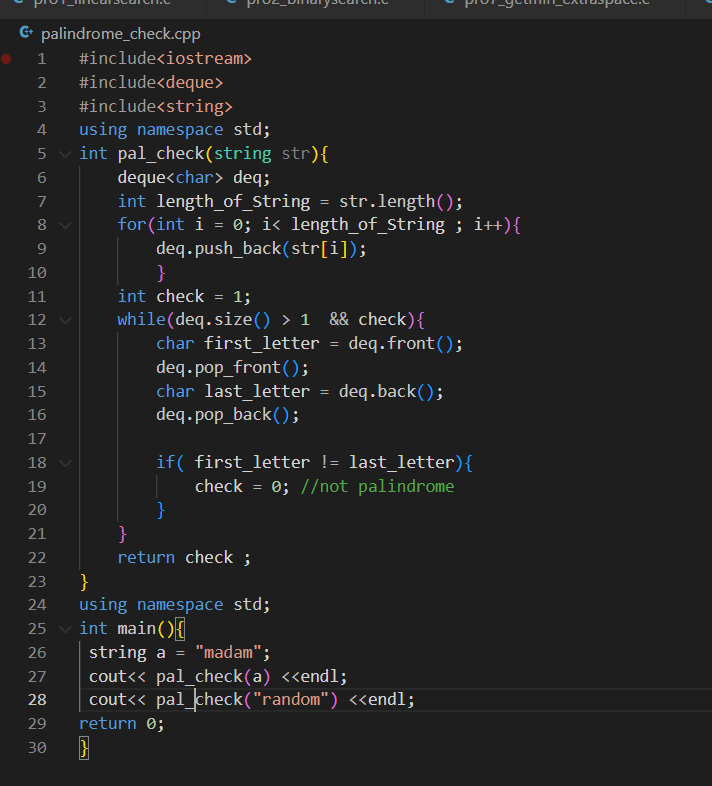
****

**Program 11**

**Program Objective: Write a program to check whether the given string is Palindrome or not, using DEQUEUE**

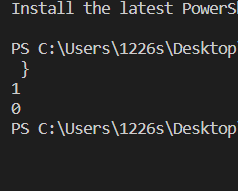
**Program theory:** A palindrome is a word, sentence, verse, or even number that reads the same backward or forward.

The solution to this problem will use a deque property. The front of the deque will hold the first character of the string and the rear of the deque will hold the last character. Since we can remove both of them directly, we can compare them and continue only if they match. If we can keep matching first and the last items, we will eventually either run out of characters or be left with a deque of size 1 depending on whether the length of the original string was even or odd. In either case, the string must be a palindrome.



**Program Code:**

**Program Output:**

****

**Program 12**

**Program Objective: Implement Tower of Hanoi Problem using Stack.**

**Program theory:** Tower of hanoi is a classic problem where you try to move all the disks from one peg to another peg using only three pegs. Initially, all of the disks are stacked on top of each other with larger disks under the smaller disks. You may move the disks to any of three pegs as you attempt to relocate all of the disks, but you cannot place the larger disks over smaller disks and only one disk can be transferred at a time.

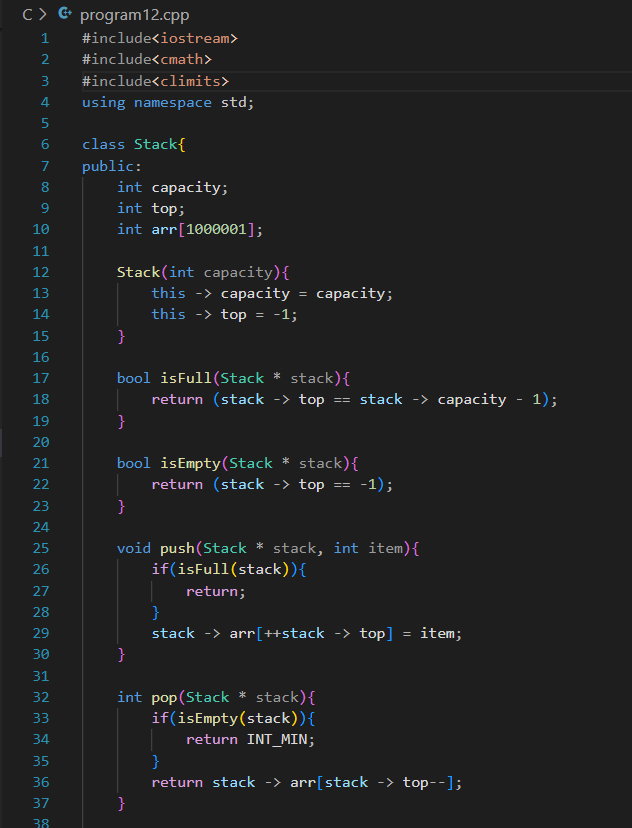
This problem can be easily solved by Divide & Conquer algorithm.

Let T (n) be the total time taken to move n disks from peg A to peg C

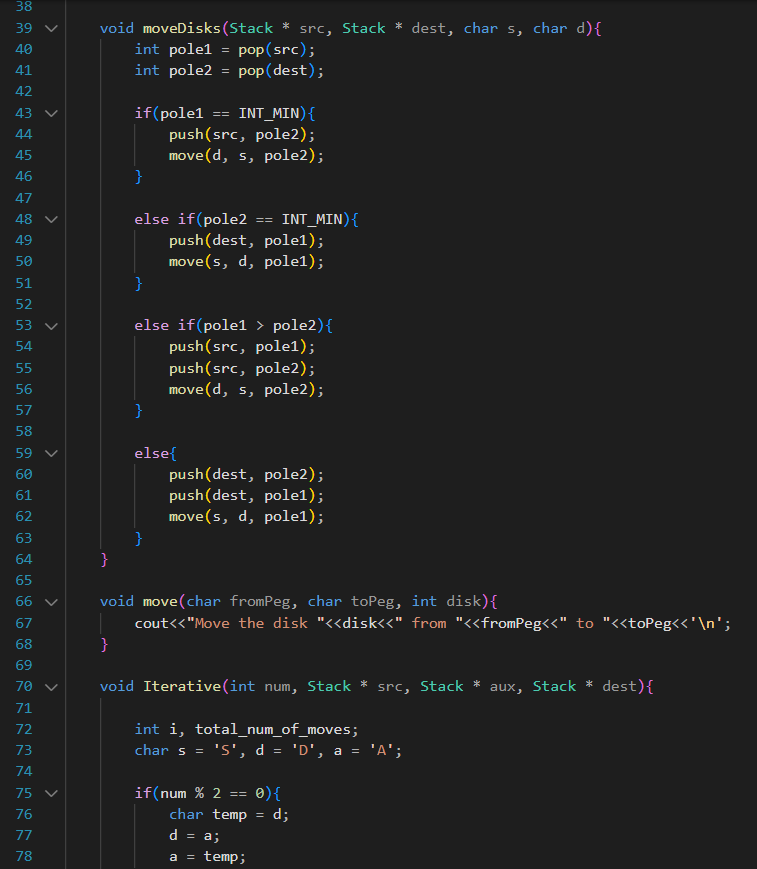
* Moving n-1 disks from the first peg to the second peg. This can be done in T (n-1) steps.
* Moving larger disks from the first peg to the third peg will require first one step.
* Recursively moving n-1 disks from the second peg to the third peg will require again T (n-1) step.

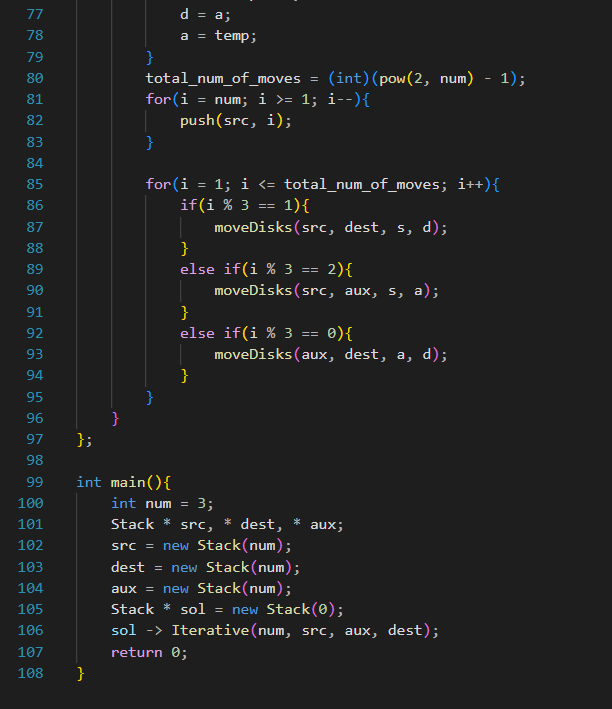
So, total time taken T (n) = T (n-1) +1+ T(n-1)

Relation formula for Tower of Hanoi is: T (n) = 2T (n-1) + 1

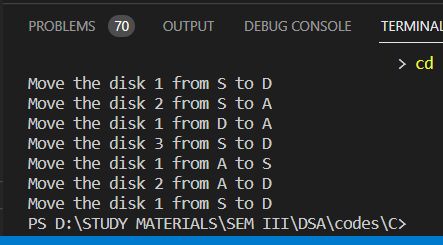
****

**Program Code:**

****

****

**Program Output:**



**Program 13**

**Program Objective**: **Write a program to implement the Linked List Data structure and insert a new node at the beginning, and at a given position.**

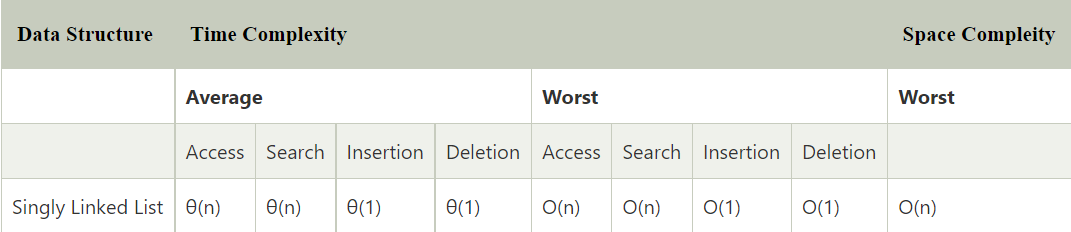
**Program theory**: Linked List can be defined as collection of objects called nodes that are randomly stored in the memory. A node contains two fields i.e., data stored at that particular address and the pointer which contains the address of the next node in the memory. The last node of the list contains pointer to the null.

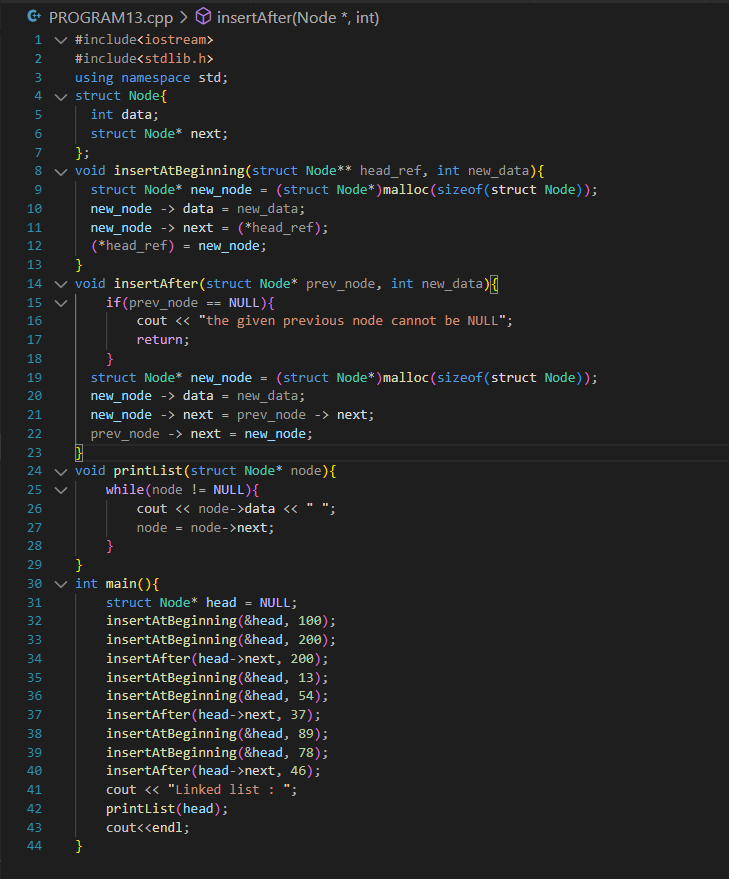
Array contains following limitations:

* The size of array must be known in advance before using it in the program.
* Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
* All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

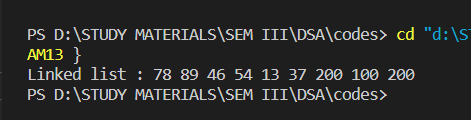
Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

* It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
* Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.



**Program Code: **

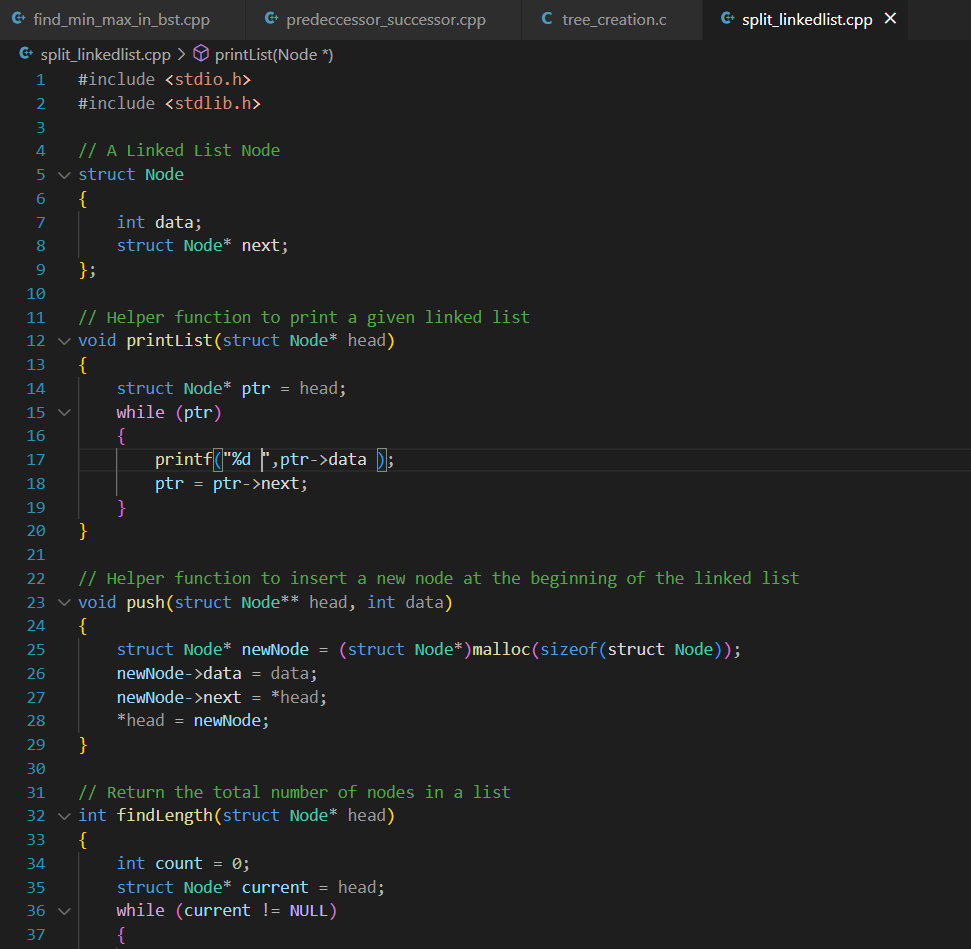
**Program Output:**

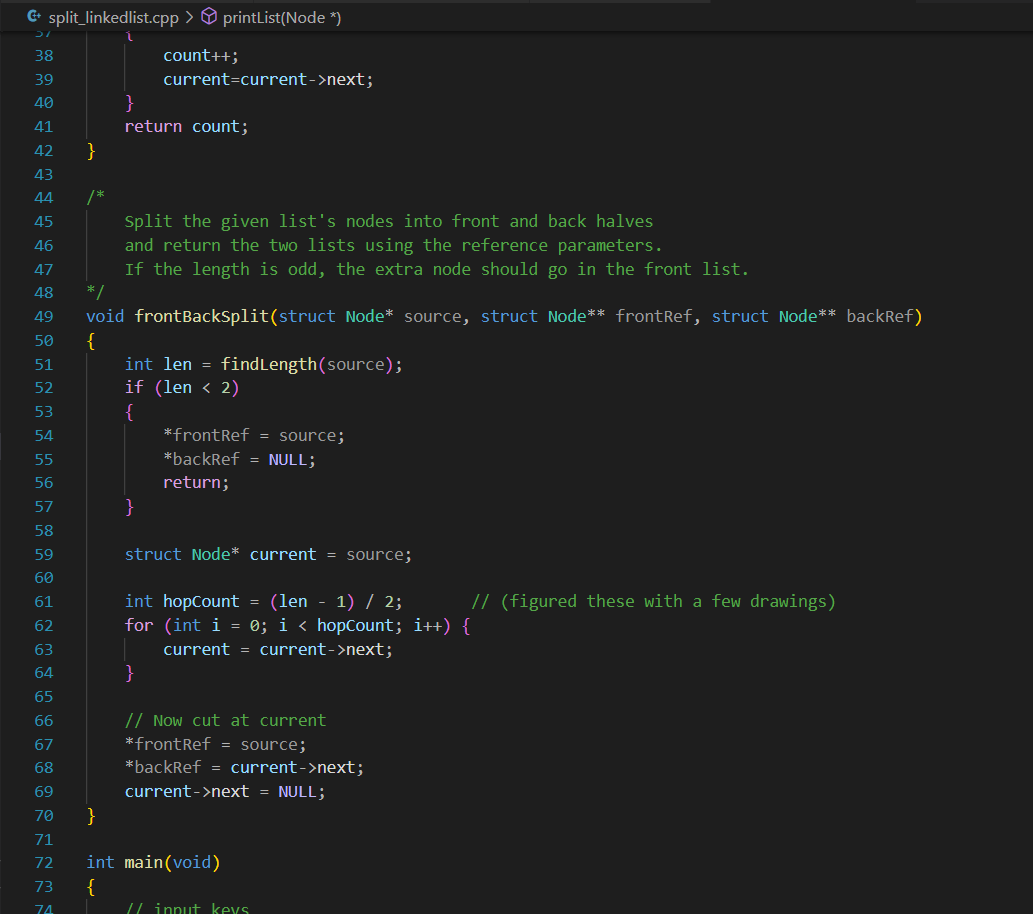


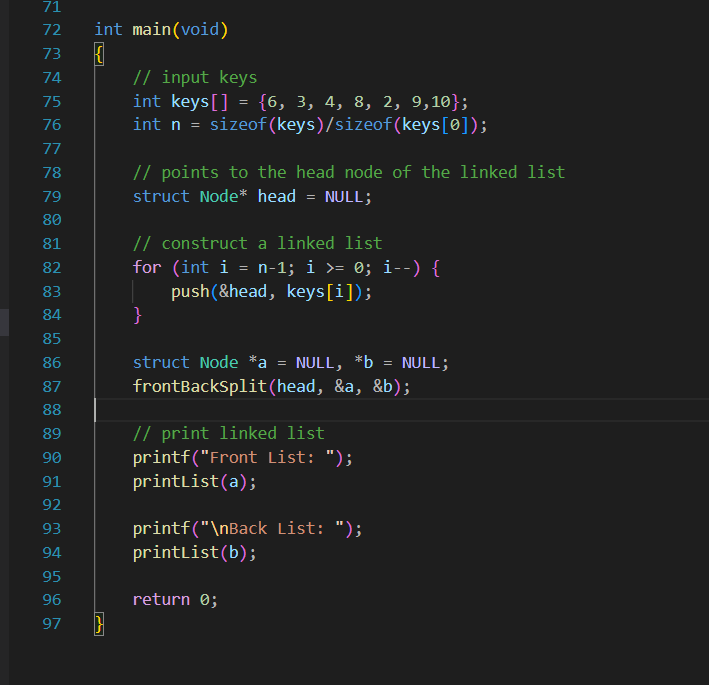
**Program 14**

**Program Objective: Write a program to split a given linked list into two sub-list as Front sub-list and back sub-list, if odd number of the element, then add the last element into the front list.**

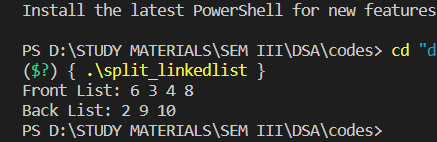
**Program theory**: Linked List can be defined as collection of objects called nodes that are randomly stored in the memory. A node contains two fields i.e., data stored at that particular address and the pointer which contains the address of the next node in the memory. The last node of the list contains pointer to the null. Probably the simplest strategy is to compute the length of the list, then use a for loop to hop over the right number of nodes to find the last node of the front half, and then cut the list at that point.

**Program Code: **

****

****

**Program Output:**

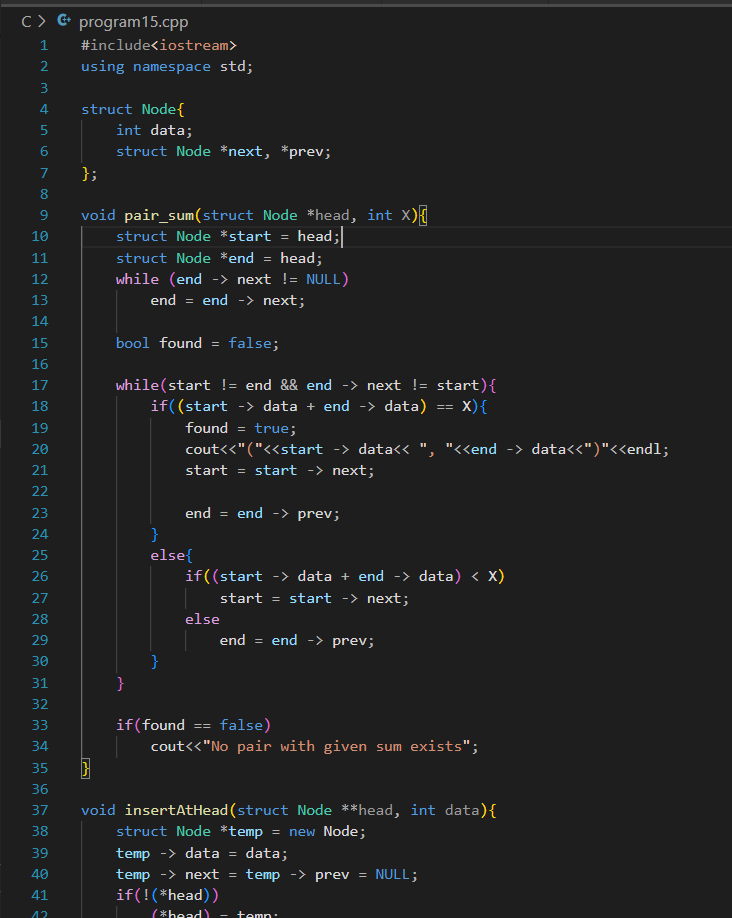
****

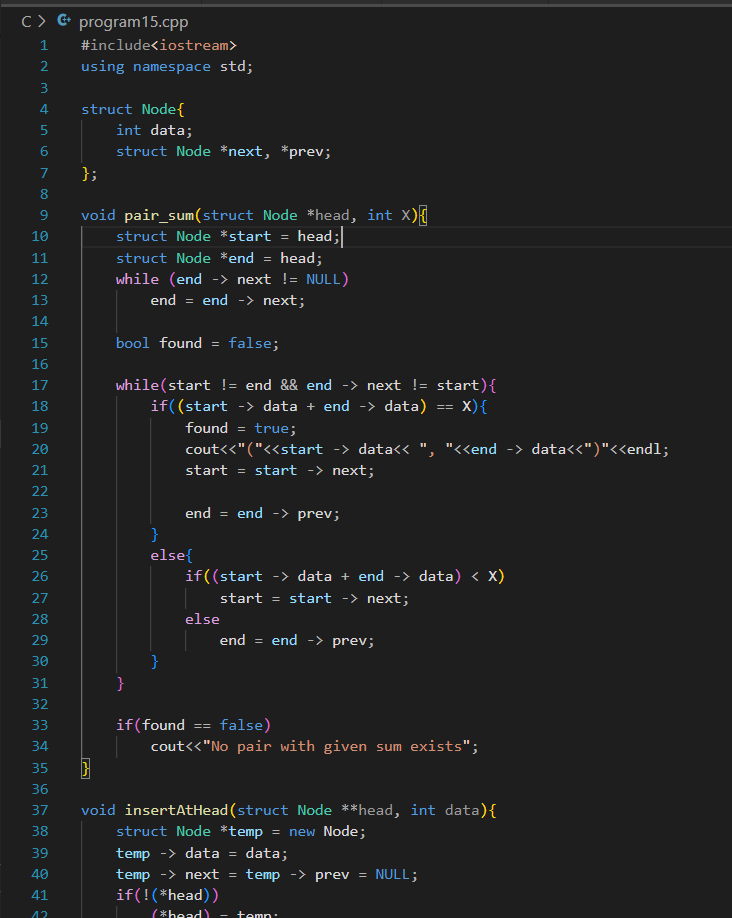
**Program 15**

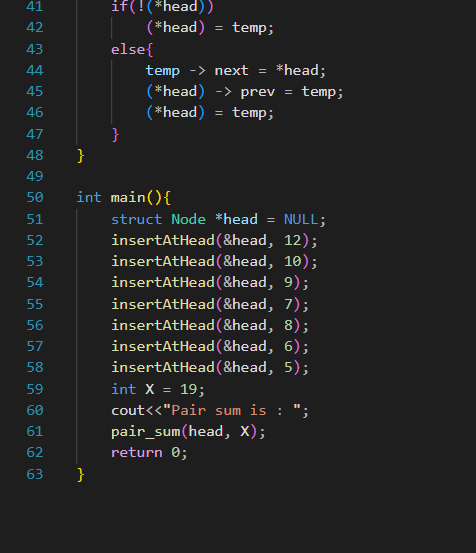
**Program Objective: Given a Sorted doubly linked list of positive integers and an integer, then finds all the pairs (sum of two nodes data part) that is equal to the given integer value. Example: Double Linked List 2, 5, 7, 8, 9, 10, 12, 16, 19, 25, and P=35 then pairs will be Pairs will be (10, 25), (16, 19).**

**Program theory**: Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward). A Doubly Linked List (DLL) contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

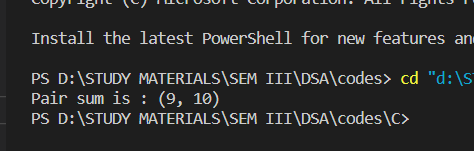
**Program Code:**

****

****

****

**Program Output:**



**Program 16**

**Program Objective: Write a program to implement the Binary Tree using linked list and perform In-order traversal.**

**Program theory**: Binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

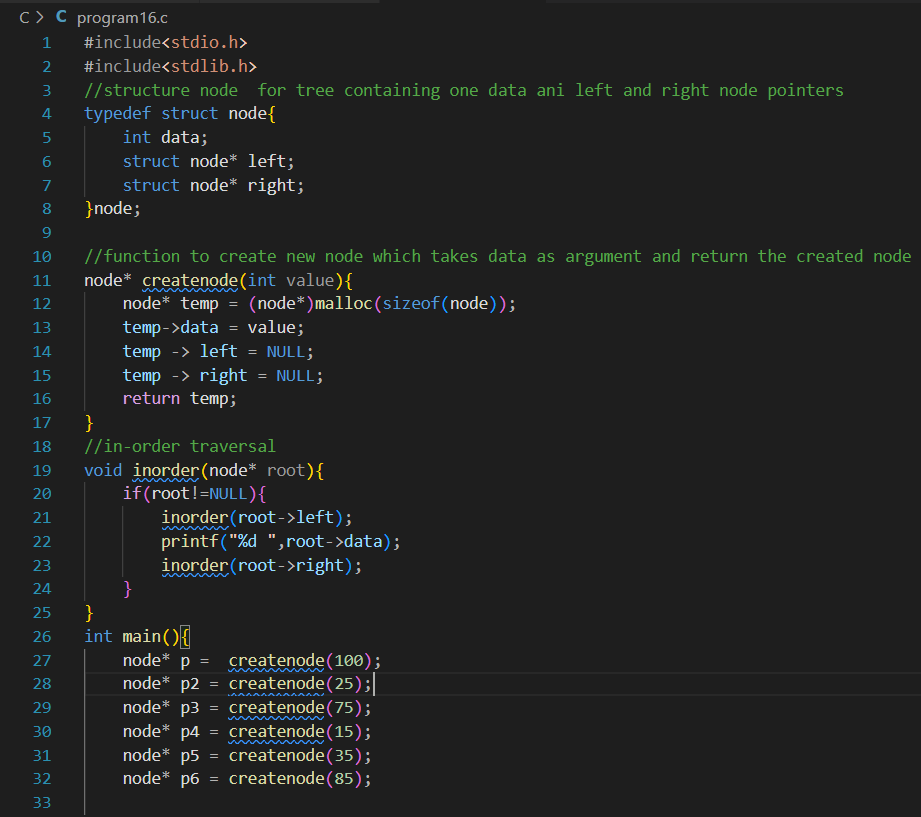
Properties of Binary Tree:

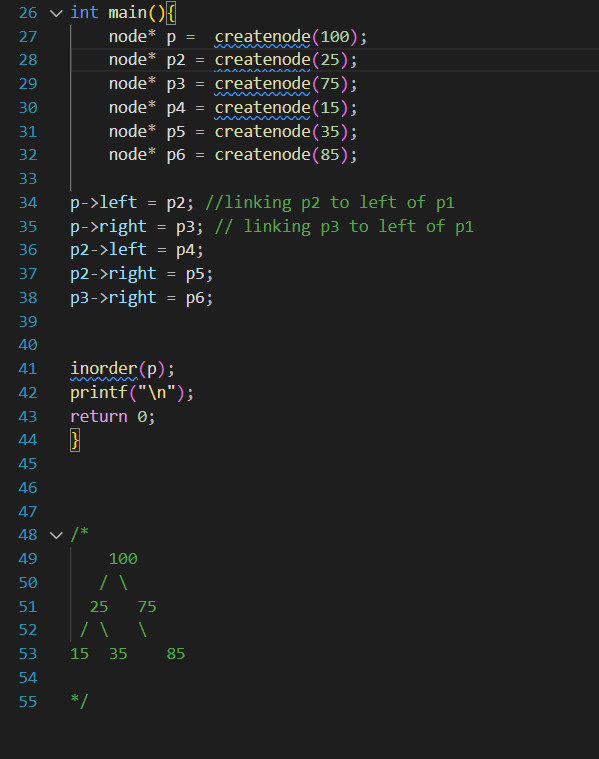
* At each level of i, the maximum number of nodes is 2i.
* The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is (20 + 21 + 22+….2h) = 2h+1 -1.
* The minimum number of nodes possible at height h is equal to h+1.
* If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

Types of Binary Tree

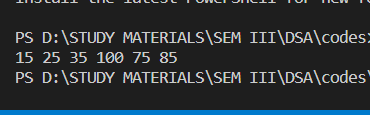
There are four types of Binary tree:

* Full/ proper/ strict Binary tree
* Complete Binary tree
* Perfect Binary tree
* Degenerate Binary tree
* Balanced Binary tree

**Program Code: **

****

**Program Output:**

****

**Program 17**

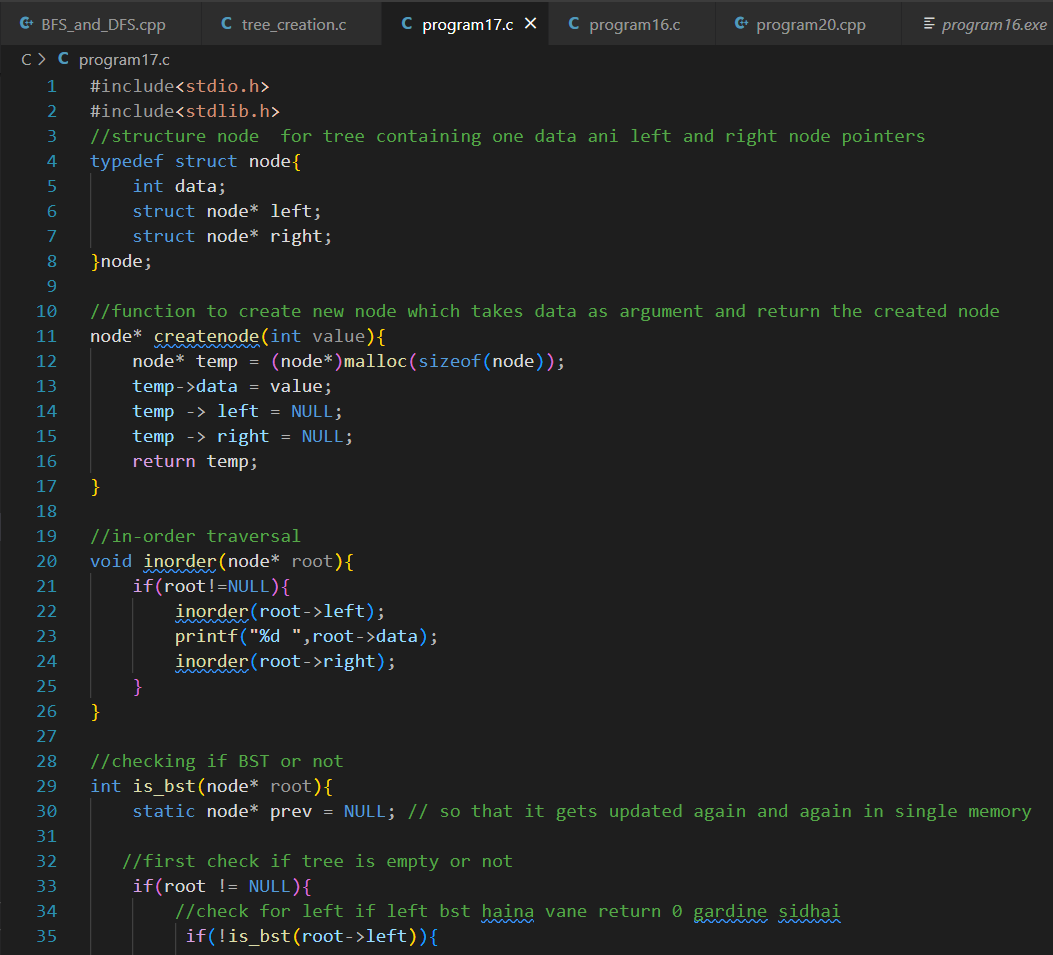
**Program Objective: Write a Program to check whether the given tree is a Binary Search Tree or not.**

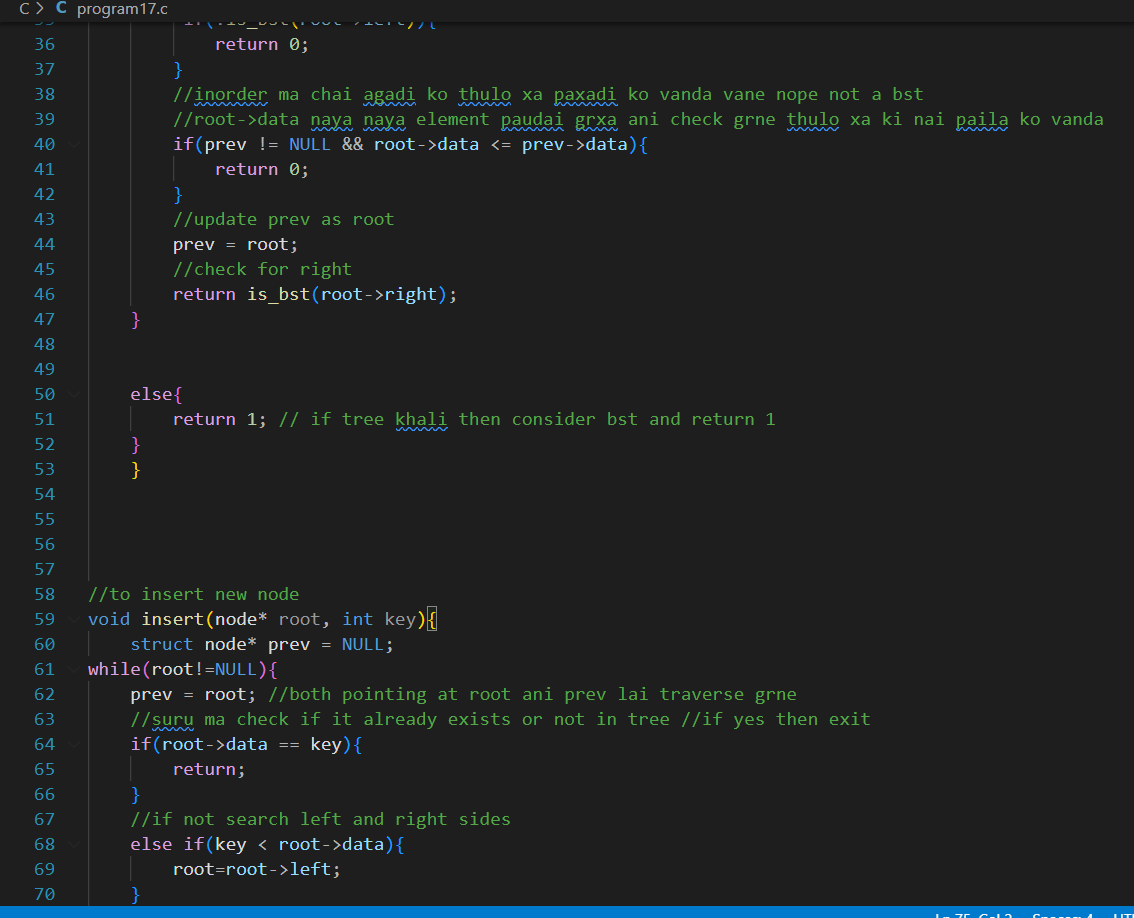
**Program theory**: A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

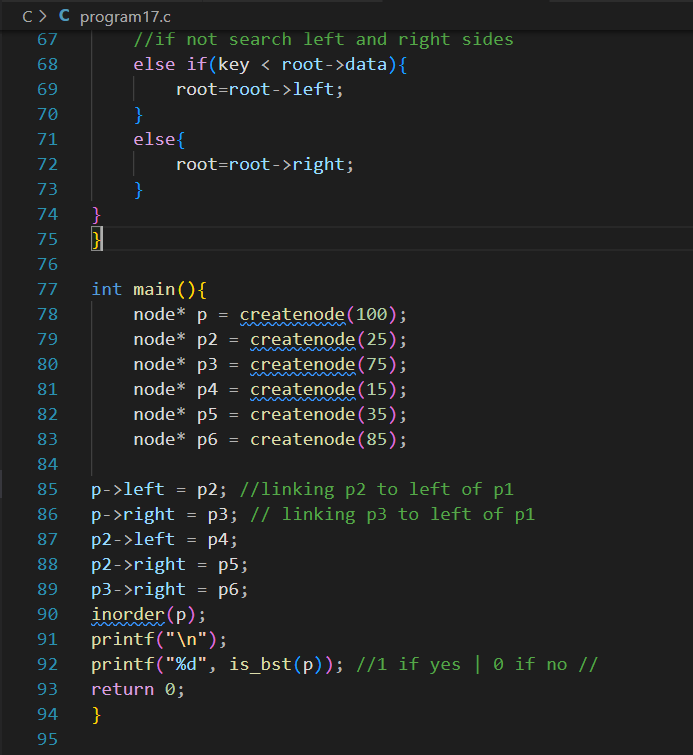
Binary Search Tree is a node-based binary tree data structure which has the following properties:

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.

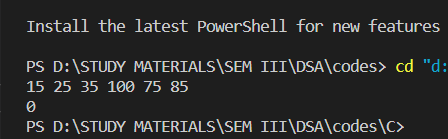
**Program Code:**

****

****

****

**Program Output:**



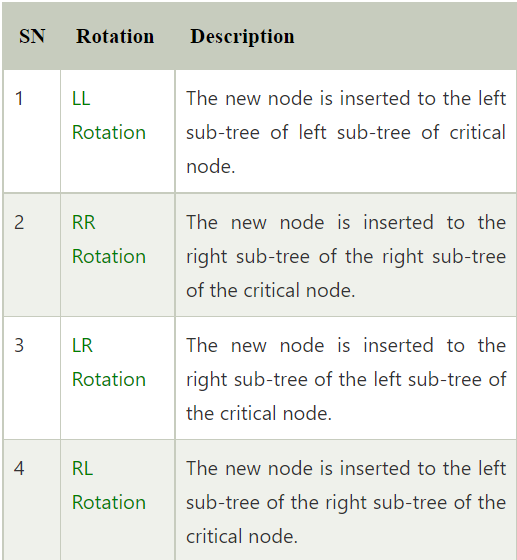
**Program 18**

**Program Objective: Write a program to implement insertion in the AVL tree.**

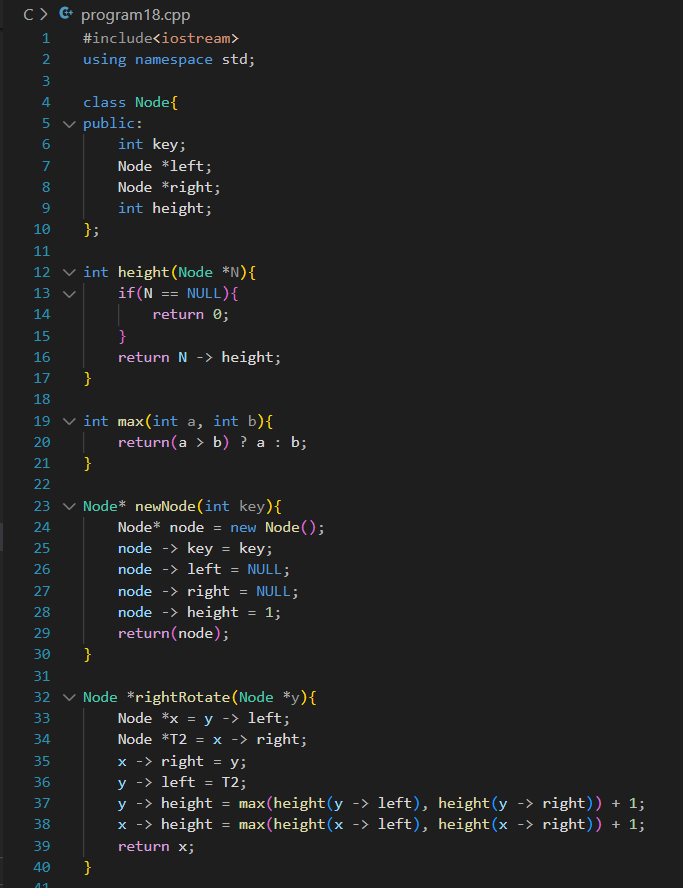
**Program theory**: AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.

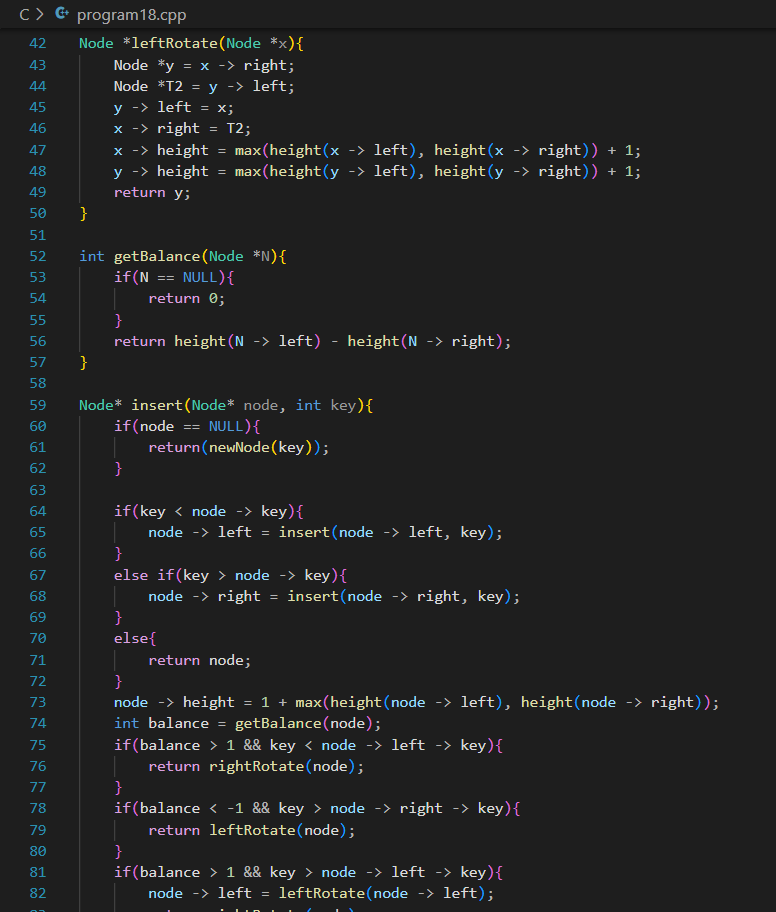
Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

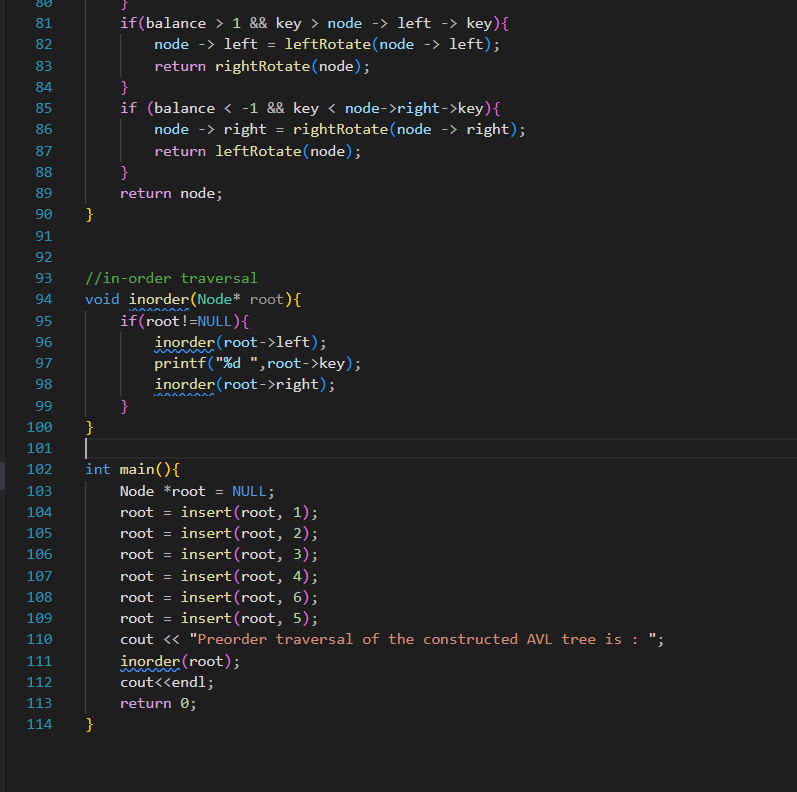
* Left Rotation
* Right Rotation



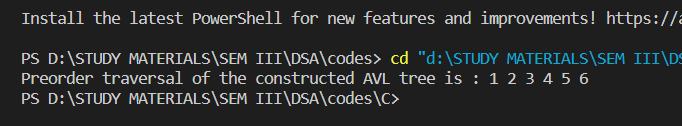
**Program Code:**

****

****

****

**Program Output:**



**Program 19**

**Program Objective: Write an Algorithm to count the number of leaf nodes in an AVL tree.**

**Program theory**: AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log(n)) after every insertion and deletion, then we can guarantee an upper bound of O(log(n)) for all these operations. The height of an AVL tree is always O(log(n)) where n is the number of nodes in the tree.

The logic is the same for the leaf node, any node whose left and right children are null is known as a leaf node in a binary tree. They are the nodes that reside in the last level of a binary tree and they don't have any children. In order to count the total number of leaf nodes in a binary tree, you need to traverse the tree and increase the count variable whenever you see a leaf node.

Here are the actual steps to follow:

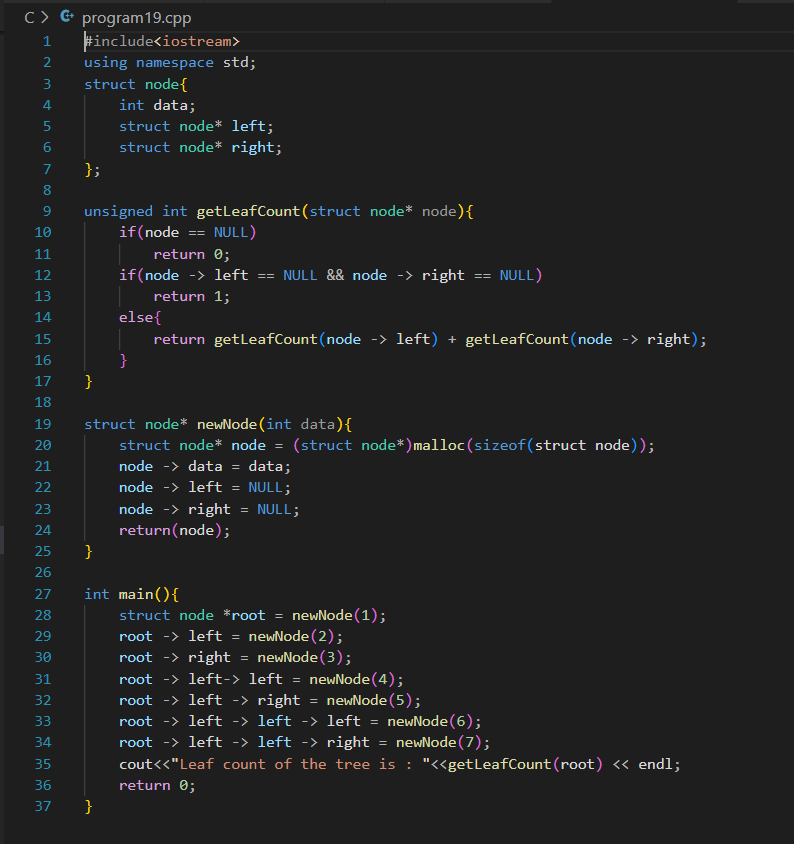
1) If the node is null return 0, this is also the base case of our recursive algorithm

2) If a leaf node is encountered then return 1

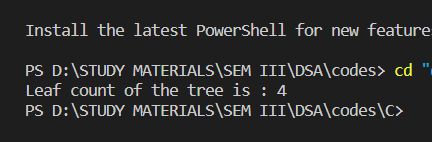
3) Repeat the process with left and right subtree

4) Return the sum of leaf nodes from both left and right subtree

**Program Code:**

****

**Program Output:**

****

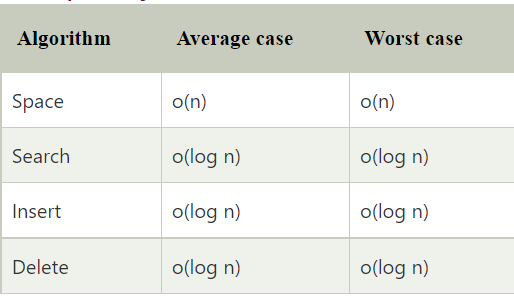
**Program 20**

**Program Objective: Write a program to Delete a key from the AVL tree.**

**Program theory**: AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honor of its inventors. AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree. Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

**Balance Factor (k) = height (left(k)) - height (right(k))**

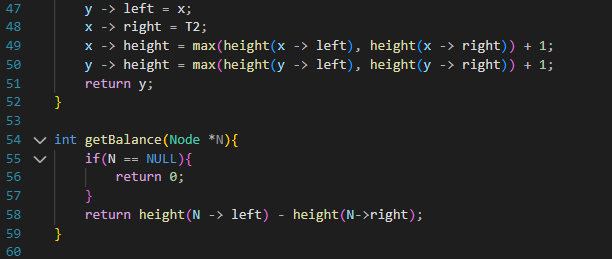
* If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
* If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
* If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

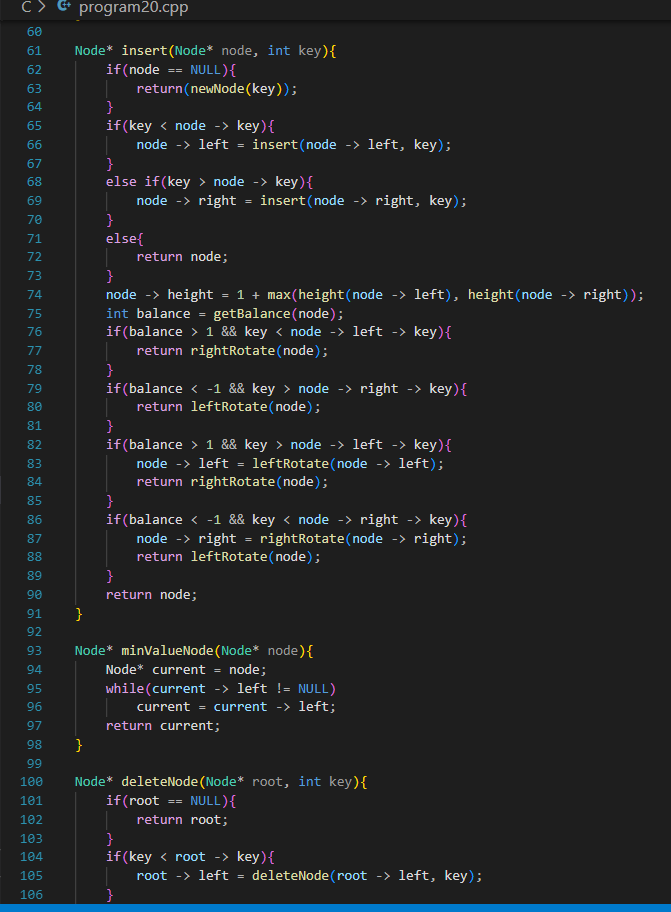


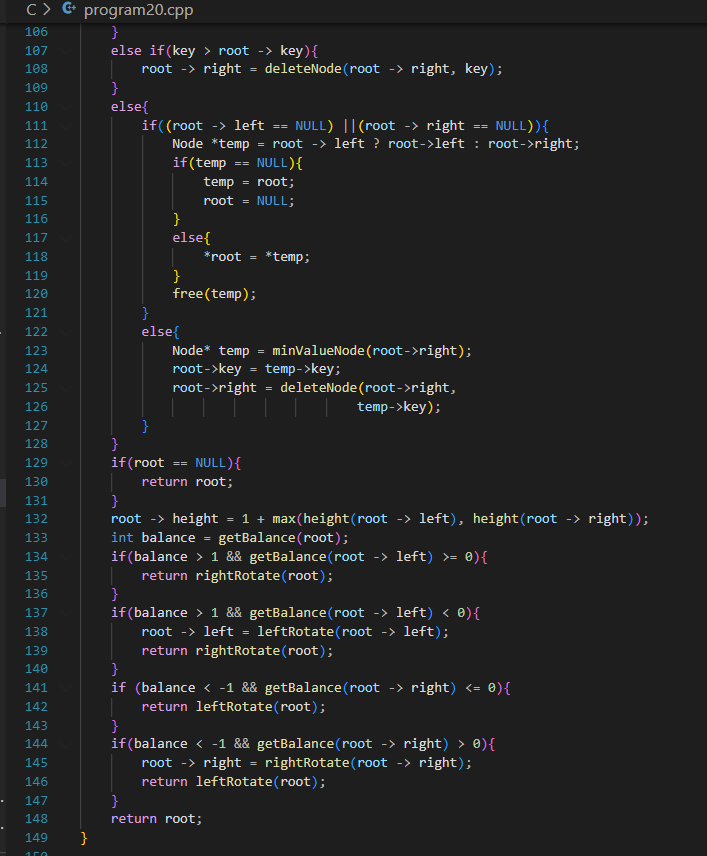
Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore; various types of rotations are used to rebalance the tree.

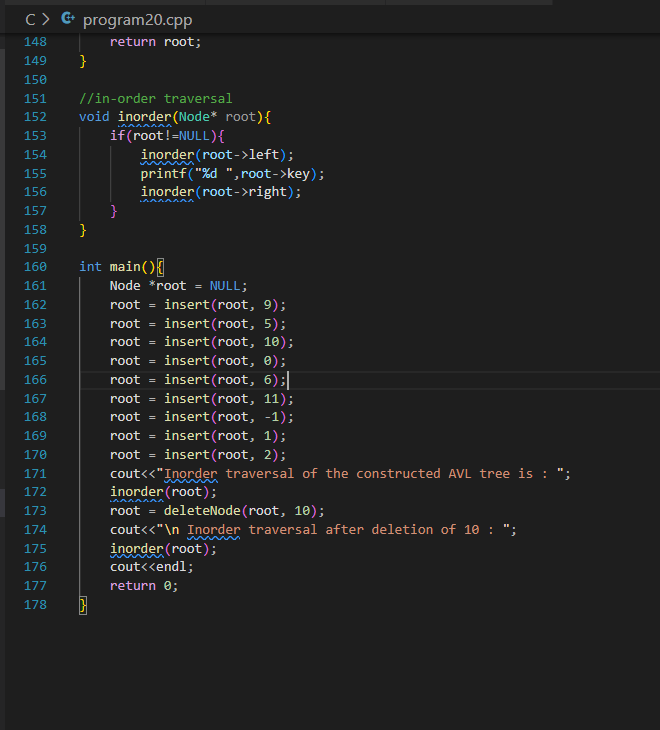
**Program Code:**



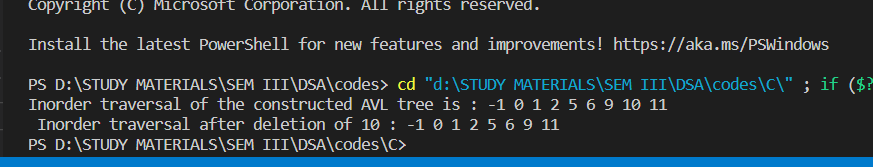


****

****

****

**Program Output:**



**Program 21**

**Program Objective: Write a program to implement Stack Data Structure using Queue.**

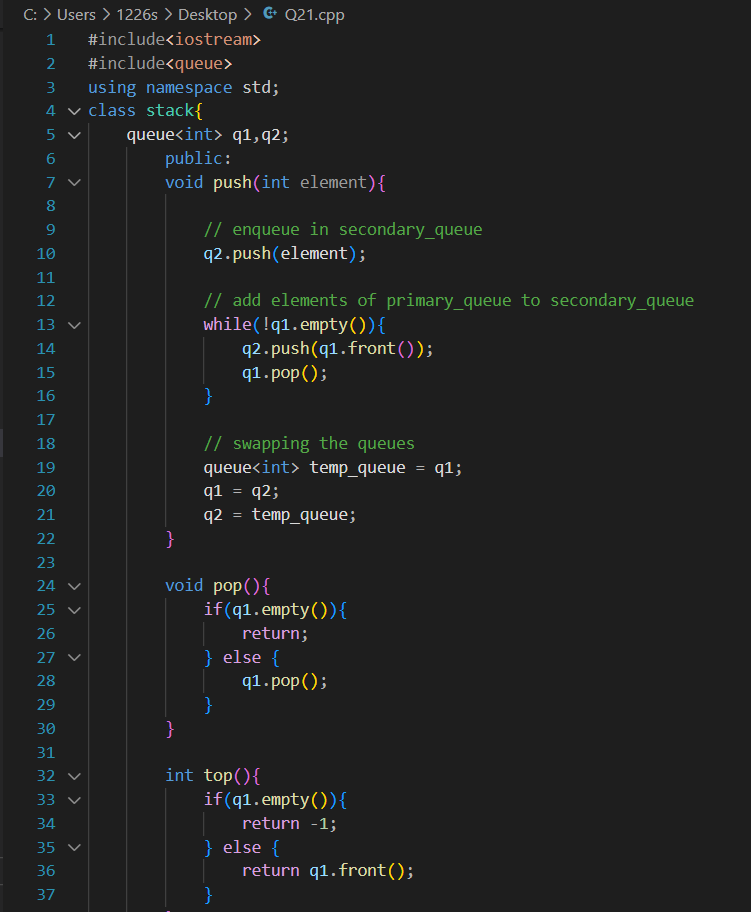
**Program theory**: Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A stack is a linear data structure that follows the principle of Last In First Out (LIFO). This means the last element inserted inside the stack is removed first.

Given a Queue data structure that supports standard operations like enqueue () and dequeue (). The task is to implement a Stack data structure using only instances of Queue and Queue operations allowed on the instances. A Stack can be implemented using two queues. Let Stack to be implemented be ‘s’ and queues used to implement are ‘q1’ and ‘q2’. Stack ‘s’ can be implemented in two ways:

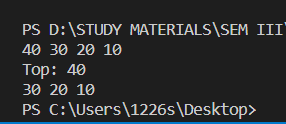
* Implement Stack using Queues By making push () operation costly*:* The idea is to keep newly entered element at the front of ‘q1’ so that pop operation dequeues from ‘q1’. ‘q2’ is used to put every new element in front of ‘q1’.
* Implement Stack using Queues by making pop () operation costly: The new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally, the last element is dequeued from q1 and returned.

**Program Code:**

****

****

**Program Output:**



**Program 22**

**Program Objective: Write a program to implement Queue Data Structure using Stack.**

**Program theory**: Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

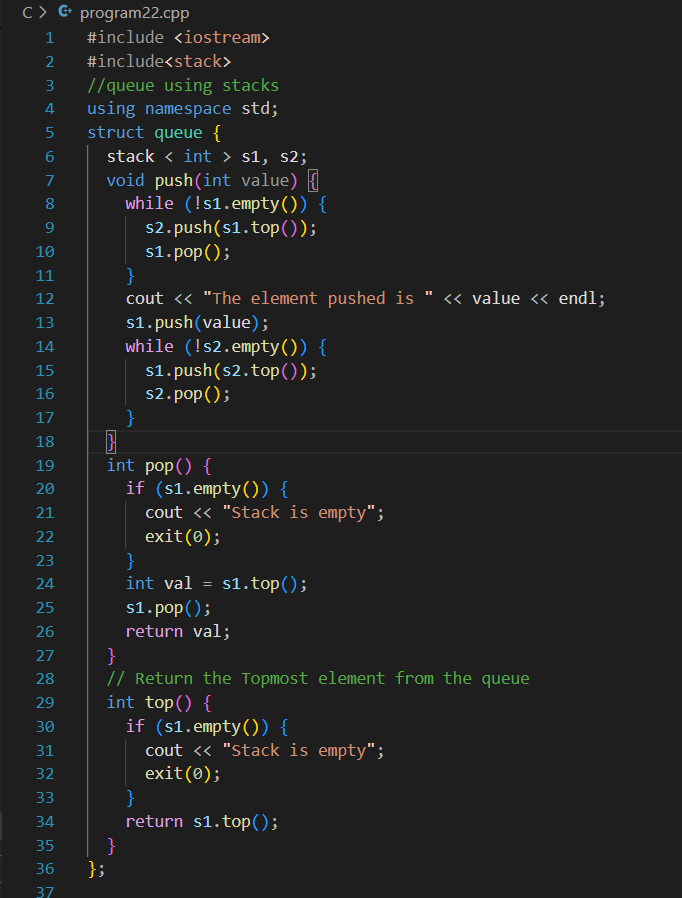
A stack is a linear data structure that follows the principle of Last In First Out (LIFO). This means the last element inserted inside the stack is removed first.

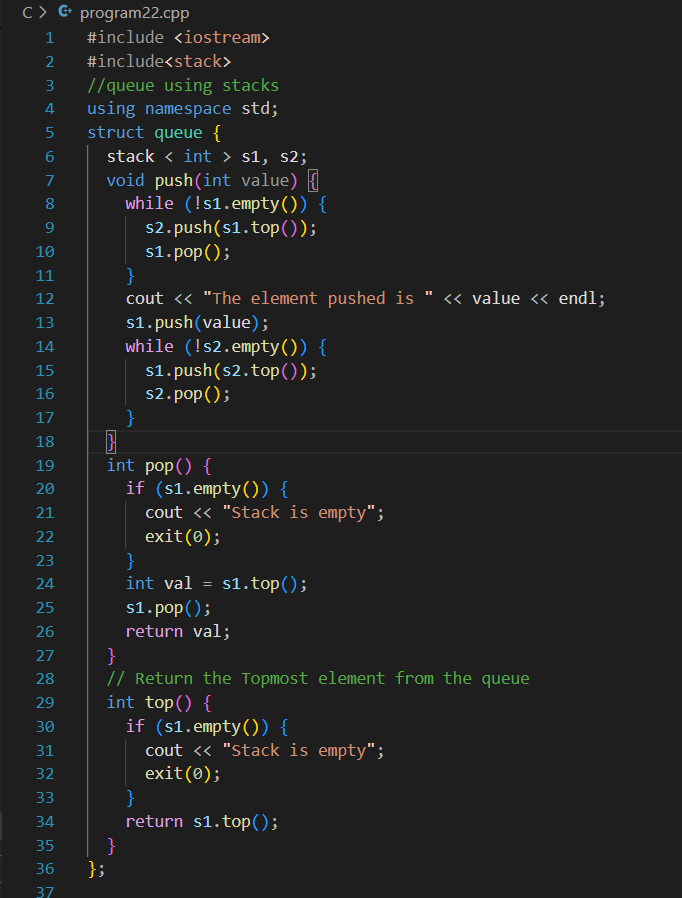
A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

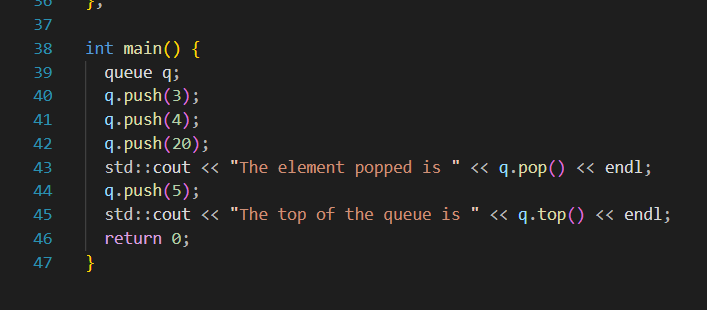
* Method 1 (By making enQueue operation costly)
* Method 2 (By making deQueue operation costly)

Method 2 is definitely better than method 1. Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty. So, the amortized complexity of the dequeue operation becomes Theta(1).

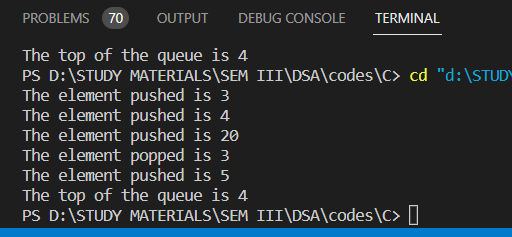
**Program Code:**

****

****

****

**Program Output:**



**Program 23**

**Program Objective: Write a program to implement Graph Data Structure and Its traversal BFS and DFS.**

**Program theory**: A graph data structure is a collection of nodes that have data and are connected to other nodes. A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices (V) and a set of edges (E). The graph is denoted by G (E, V).

Graph traversal is a technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

* DFS (Depth First Search)
* BFS (Breadth First Search)

**BFS traversal of a graph** produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

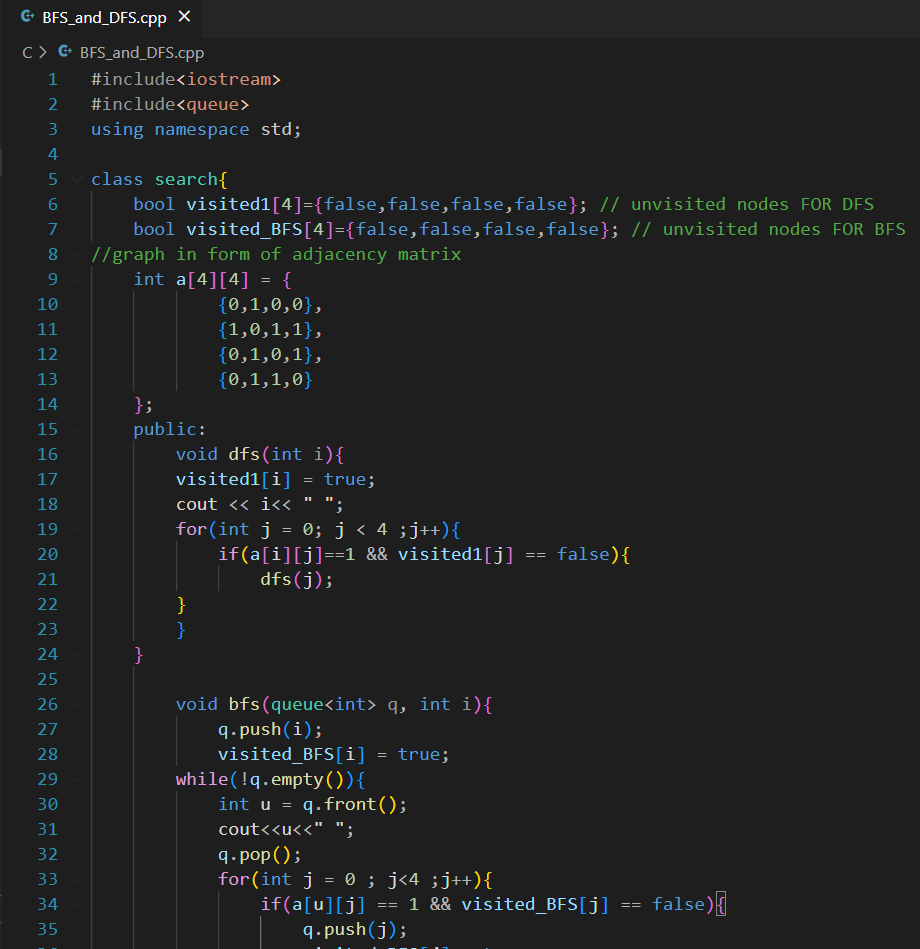
Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

**Depth-first search** is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

So, the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

* Create a recursive function that takes the index of the node and a visited array.
* Mark the current node as visited and print the node.
* Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

**Program Code:**

****

****

**Program Output:**

