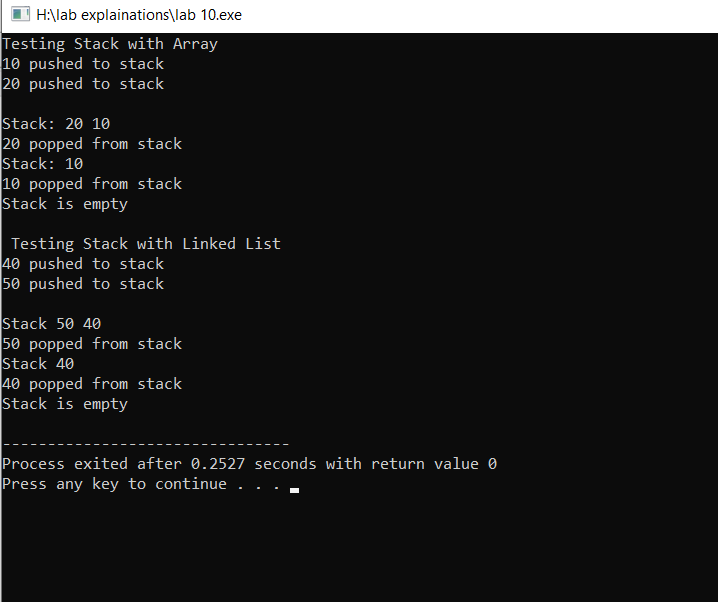
**LAB 10:**

**Explanation**

* Two stack implementations are shown: one using an array (Array class), the other using a linked list (LinkedList class).
* push(val) adds an element to the top of the stack.
* pop() removes the top element from the stack.
* display() prints all stack elements from top to bottom.
* The main() function tests both implementations with sample operations.

**OUTPUT:**

****

**LAB 11:**

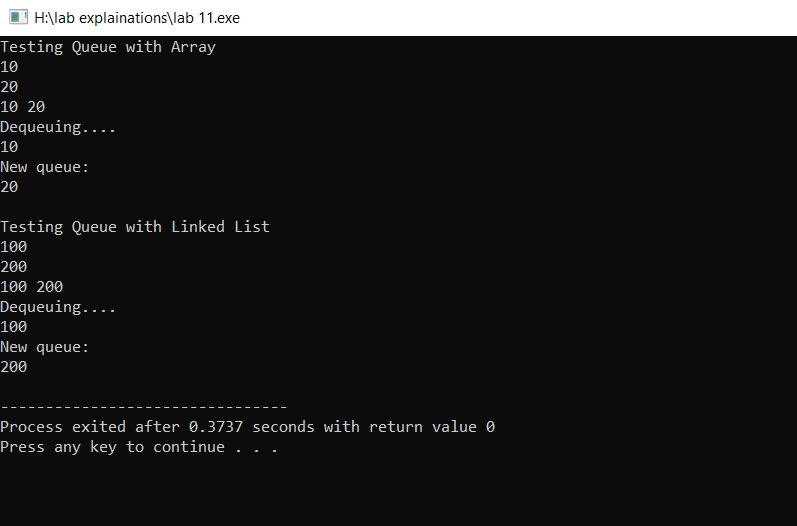
**Explanation**

* This code demonstrates **two queue implementations**: one with an **array** (Array class) and one with a **linked list** (LinkedList class).
* **Queue Operations**:
  + enqueue(val): Adds an element to the rear.
  + dequeue(): Removes an element from the front.
  + display(): Prints all elements in the queue.

**Key Points:**

* **Array**: Uses front and rear indices to manage the queue.
* **Linked List**: Uses front and rear pointers to link nodes for the queue.
* **main()**: Tests both queue implementations with basic operations.

**OUTPUT:**

****

**LAB 12:**

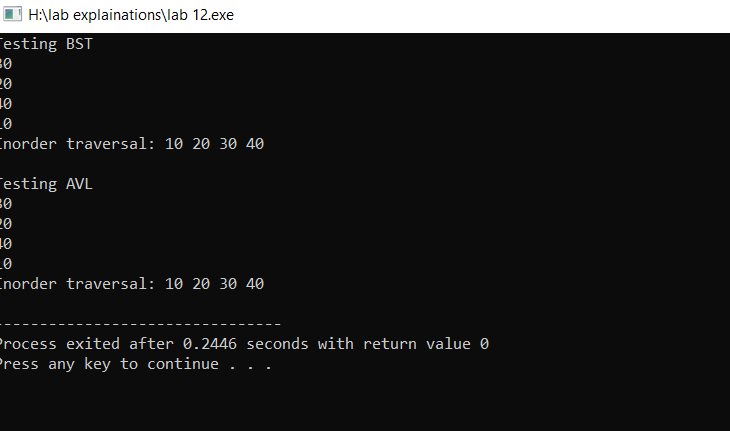
**Explanation**

* This code demonstrates the implementation of **two types of binary trees**: a **Binary Search Tree (BST)** and an **AVL Tree**.

**Key Points:**

* **BST (Binary Search Tree)**:
  + Each node has a data, left (left child), and right (right child).
  + Insertion follows the BST property where values less than the node's data go to the left, and values greater go to the right.
  + **Inorder Traversal**: Prints values in ascending order (left, root, right).
* **AVL (Adelson-Velsky and Landis Tree)**:
  + An **self-balancing binary search tree** where the difference in heights of the left and right subtrees (balance factor) is kept at most 1.
  + Rotations (left and right) are used to maintain balance after insertion:
    - **Left Rotation**: Done when the left subtree is heavier.
    - **Right Rotation**: Done when the right subtree is heavier.
    - **Left-Right and Right-Left Rotations**: Double rotations used for balancing when necessary.
  + **Inorder Traversal**: Prints values in ascending order.

**OUPUT:**

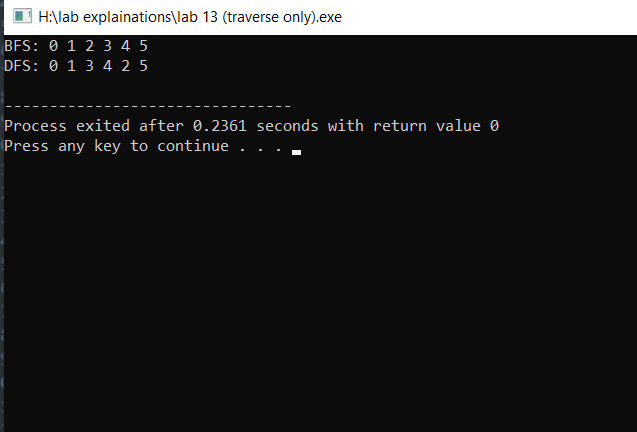
****

**LAB 13(TRAVERSE ONLY):**

**Explanation**

* The graph is represented using an adjacency list (list<int> adj[V]).
* addEdge connects two nodes in an undirected graph.
* bfs(start) uses a queue to visit nodes level by level.
* dfs(node) uses recursion to explore as deep as possible before backtracking.
* The visited[] array prevents revisiting nodes.
* In main(), a sample graph is created, and both BFS and DFS are performed from node 0.

**OUTPUT:**

****