# Superior University, Lahore

**Name: Wardah Shoaib**

**Roll No: BSSEM-S24-095**

**DSA LAB TASKS 10-13**

**Submitted to: Sir Rasikh Ali**

# LAB 10: Stack with LinkedList and Array

## Stack with Array

**How it works:**

- Uses a fixed-size array (int arr[5]) to store elements.
- A variable top keeps track of the top element's index.
- **Push** adds an element at top + 1.
- **Pop** removes the element at top and decrements top.
- **Display** prints elements from top to bottom.

**Why use array:**

- Simple and fast (O(1) for push/pop).
- Easy to implement for small, fixed-size stacks.
- But limited: size is fixed, and it can overflow.

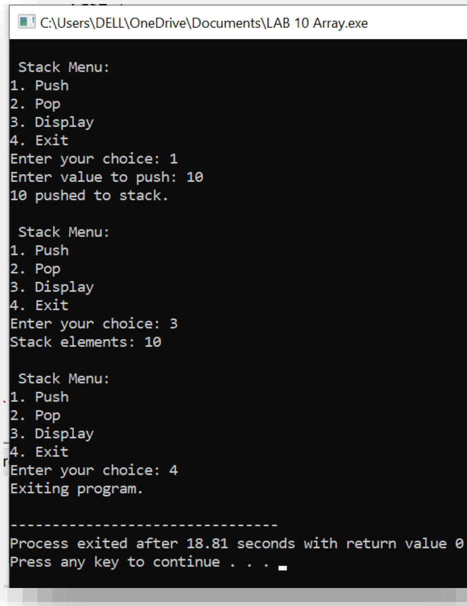
## Stack with Linked List

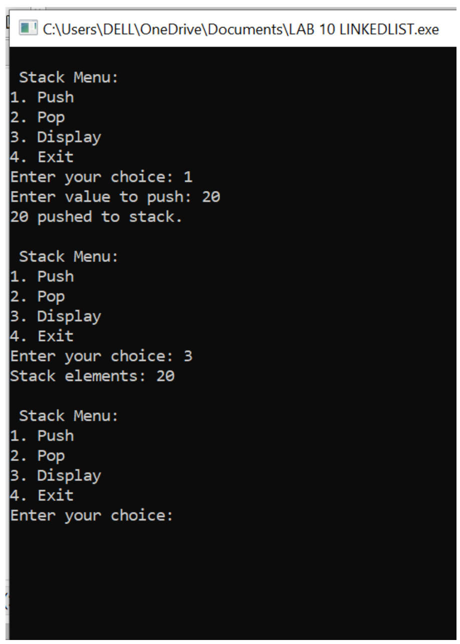
**How it works:**

- Each element is a Node (a class with data and next pointer).
- The top pointer points to the last pushed node.
- **Push** creates a new node and links it to the current top.
- **Pop** deletes the top node and updates top.
- **Display** traverses from top node to bottom.

**Why use linked list:**

- No size limit (grows as needed).
- No overflow unless memory is full.
- Useful when dynamic size is needed.




# LAB 11: Queue with LinkedList and Array

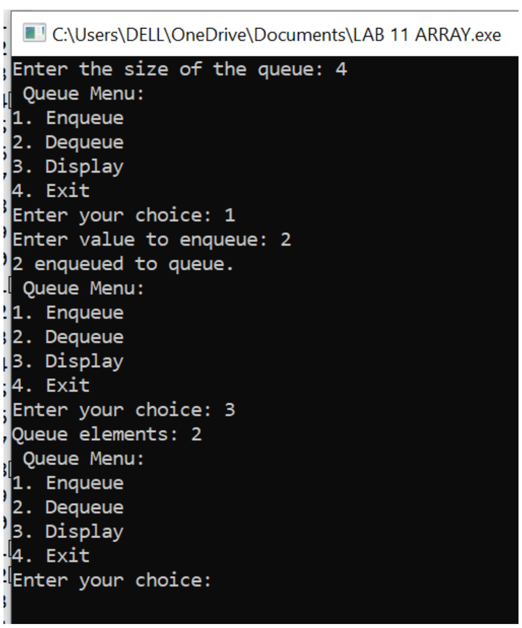
## Queue with Array:

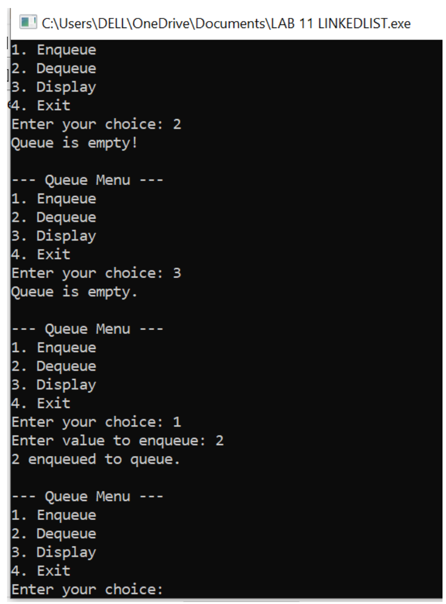
- **How**: This code implements a queue using a fixed-size array. It defines a Queue class with two key pointers: front and rear, which help manage the queue's elements.
  - **Enqueue**: Adds an element at the rear (end) if there is space.

   - **Dequeue**: Removes an element from the front (beginning) of the queue.
   - **Display**: Prints the elements from front to rear.
- **Why**: The array-based implementation provides a straightforward way to manage the queue, but its size is fixed at the time of creation, meaning it can't dynamically grow or shrink.

### Queue with Linked List:

- **How**: This code uses a **linked list** for the queue. A Node class represents each element, containing data and a next pointer to the next node. The Queue class has front and rear pointers to manage the queue.
   - **Enqueue**: Creates a new node and adds it to the rear.
   - **Dequeue**: Removes the node at the front and updates the front pointer.
   - **Display**: Traverses and prints the queue from front to rear.
- **Why**: The linked list implementation allows dynamic memory allocation, meaning the queue size can grow or shrink as needed.




# LAB 12: BST and AVL

### BST (Binary Search Tree)

**How**:

- Uses a Node class with data, left, and right pointers.

- The insert function places values by comparing:
  - Smaller → go left
  - Greater → go right
- inorder traversal visits nodes in sorted (ascending) order.

**Why**:

- BST is simple and efficient for sorted data access.
- But: **Not self-balancing**, can become skewed (like a linked list), making operations slow (O(n)).

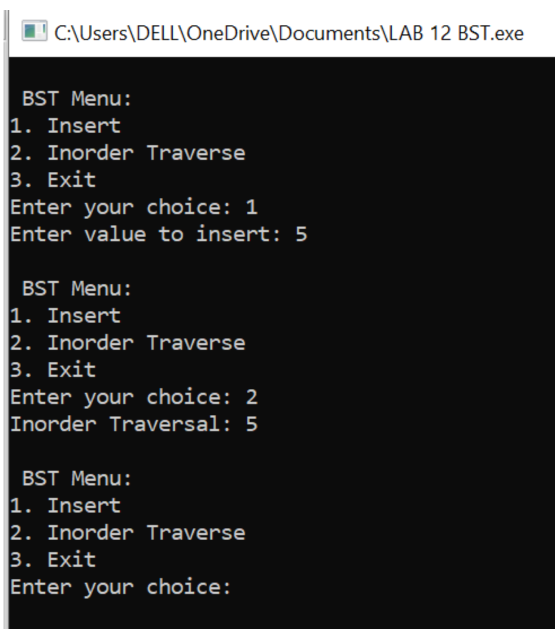
## AVL Tree

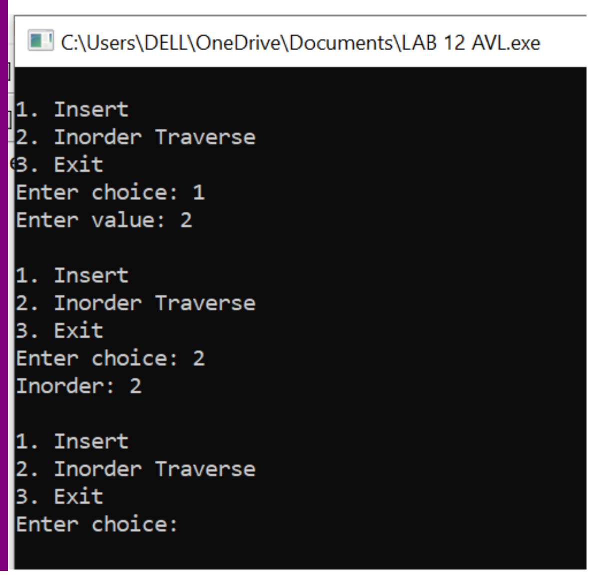
**How**:

- Same structure as BST, but tracks **height** and **balance factor** at each node.
- After insertion, the tree checks for imbalance:
  - If unbalanced, it uses **rotations (left/right)** to rebalance.
- inorder traversal still gives sorted output.

**Why**:

- AVL is a **self-balancing BST**.
- Maintains **O(log n)** time for insert/search by keeping the tree height minimal



# LAB 13: DFS and BFS

## BST (Binary Search Tree) Code:

- **Insertion**: A node is inserted by comparing the value with the current node. If the value is less, move to the left; if more, move to the right. This ensures the BST property (left subtree < node < right subtree).
- **Traversal**: Preorder, inorder, and postorder traversals are implemented using recursion. Inorder traversal gives nodes in sorted order.

## Why BST?

- It provides efficient searching, insertion, and deletion (average O(log n) time for balanced trees).
- Traversals can be used for different tree-processing tasks (e.g., printing in sorted order).

## 2. AVL (Balanced Binary Search Tree) Code:

- **Insertion**: Similar to BST but with an additional step to maintain balance after insertion. After inserting a node, the height difference between left and right subtrees is checked.
- **Balancing**: If the balance factor (height difference) of any node becomes more than 1 or less than -1, rotations are performed to restore balance.
  - **Left Rotation**: Used when the right subtree is too tall.
  - **Right Rotation**: Used when the left subtree is too tall.

## Why AVL?

- AVL trees are self-balancing, ensuring O(log n) time for operations even in the worst case. They help maintain balanced height to prevent degenerate trees, which can degrade performance in standard BST.

C:\Users\DELL\OneDrive\Documents\LAB 13 DFS IN TREE.exe

```
Preorder: 1 2 4 5 3
Inorder: 4 2 5 1 3
Postorder: 4 5 2 3 1

--------------------------------
Process exited after 5.836 seconds with return value 0
Press any key to continue . . .
```

C:\Users\DELL\OneDrive\Documents\LAB 13 DFS IN GRAPH.exe

```
DFS: 0 1 3 4 2
--------------------------------
Process exited after 6.031 seconds with return value 0
Press any key to continue . . .
```

C:\Users\DELL\OneDrive\Documents\LAB 13 BFS IN TREE.exe

```
BFS Traversal: 1 2 3 4 5

--------------------------------
Process exited after 6.093 seconds with return value 0
Press any key to continue . . .
```

C:\Users\DELL\OneDrive\Documents\LAB 13 BFS IN GRAPH.exe

```
BFS starting from node 0: 0 1 2 3 4

--------------------------------
Process exited after 5.835 seconds with return value 0
Press any key to continue . . .
```