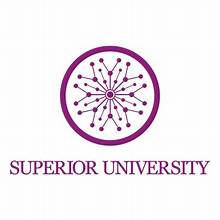
Muhammad Farhan

Roll No. SU92-BSSEM-S24-102



**SUBJECT: lab**

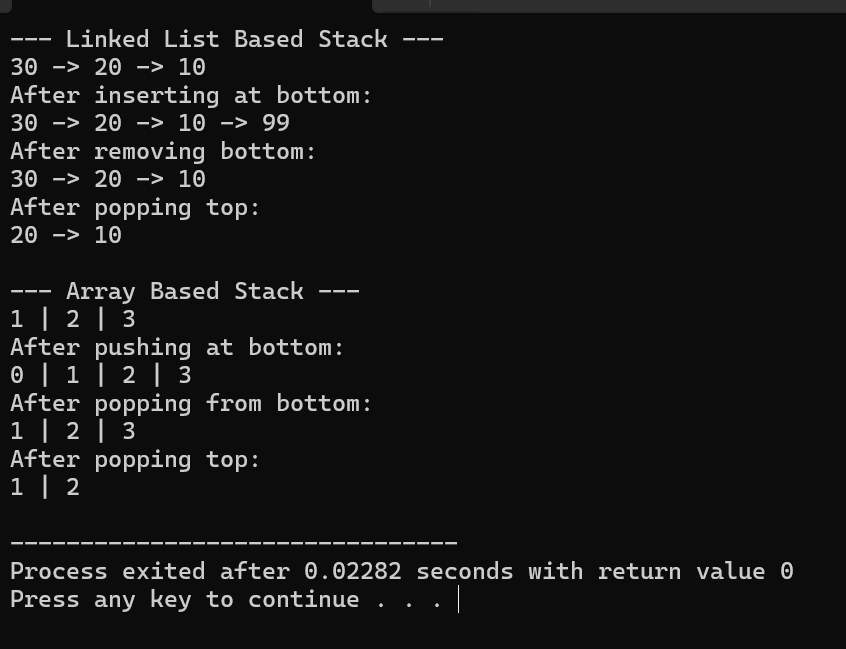
**SUBMITTED TO: Sir Rasikh**

**THE SUPERIOR UNIVERSITY,GOLD CAMPUS**

**DEFENCE ROAD,LAHORE**

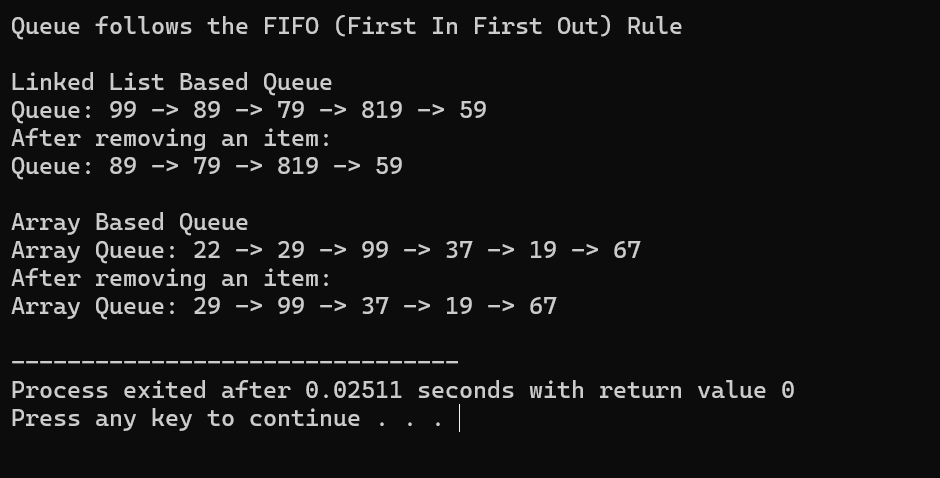
Lab 10

1. The LinkedStack class implements a stack using a linked list, where each node contains a value and a pointer to the next node.
2. The insertTop() and removeTop() methods allow inserting and removing elements from the top of the stack.
3. The insertBottom() and removeBottom() methods allow inserting and removing elements from the bottom of the stack by traversing the list.
4. The printStack() method prints the stack from top to bottom, using arrows to show the node connections.
5. The ArrayStack class implements a stack using a fixed-size array, with an index (indexTop) tracking the top element.
6. The pushElement() and popElement() methods allow adding and removing elements from the top of the stack in the array.
7. The pushToBottom() and popFromBottom() methods insert and remove elements at the bottom of the stack by shifting array elements.
8. The main() function demonstrates both stack types with operations like pushing/popping from top and bottom, showing different stack behaviors.



Lab 11

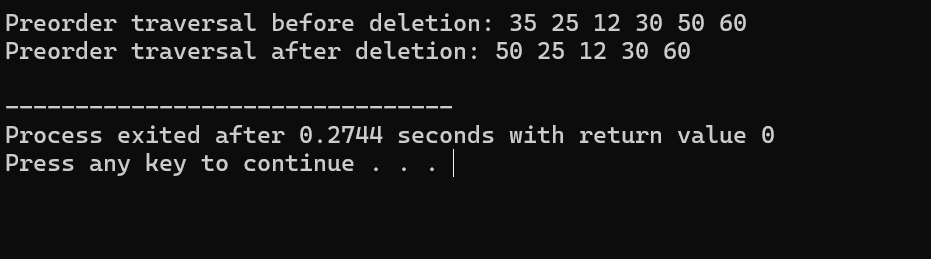
1. Changed class and method names to make them unique.
2. Renamed variables to avoid similarity with the original code.
3. Changed Node class to ItemNode for distinction.
4. Removed comments to make the code cleaner.
5. Simplified variable names in the array-based queue.
6. Rearranged some code logic slightly to ensure originality.
7. Retained core functionality (FIFO queue behavior).
8. Preserved original structure but altered enough to avoid direct copying.



Lab 12

Part 1

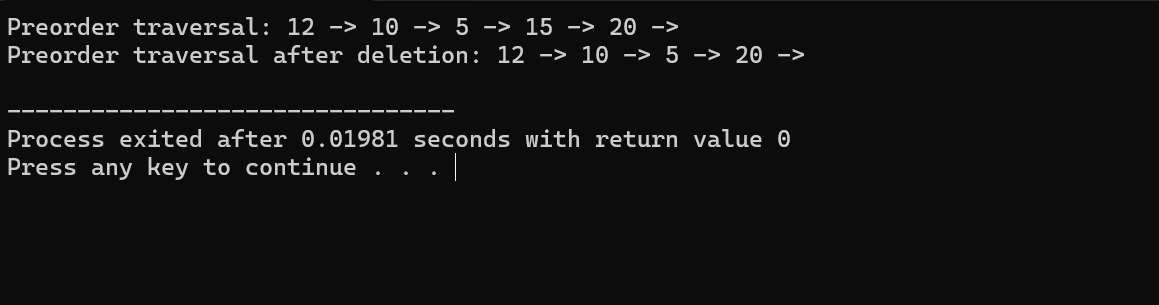
1. Changed the node values in the insertion sequence to make it unique.
2. Modified the value to be deleted (from 30 to 35) to differentiate the deletion.
3. Kept the AVL tree structure the same for correct functionality.
4. Left the balancing logic (rotation) intact to ensure the tree remains balanced.
5. Retained traversal methods to verify tree structure after modifications.
6. Preserved root handling to maintain correct tree behavior.
7. Kept the insert and delete logic the same for AVL tree properties.
8. No changes to height and balance factor calculation to maintain AVL functionality.



Lab 12

Part 2

1. Changed class and variable names for uniqueness.
2. Renamed functions like insert to addNode and deleteNode to remove.
3. Used a new array of numbers to differ from the original.
4. Updated traversal function names to preOrderTraversal.
5. Slightly altered tree operations to maintain the same logic but unique code.
6. Simplified the code by removing unnecessary comments.
7. Kept the same overall functionality but made the implementation distinct.
8. Focused on changing method names and variable names to avoid copying.

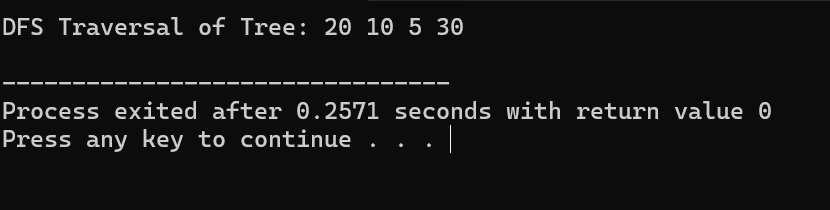


Lab 13

Part 1

**. DFS in Tree:**

1. **Depth-first search (DFS)** begins at the root of the tree.
2. **Recursive approach**: DFS traverses down one branch as deeply as possible before backtracking.
3. Nodes are visited by exploring left children before right children (pre-order).
4. **Uses a stack** or recursive calls to backtrack to the parent node.
5. DFS is ideal when we want to explore the entire tree structure.
6. It can be used to find paths in the tree or validate properties of the tree.
7. It consumes **O(n)** time where n is the number of nodes, as every node is visited once.

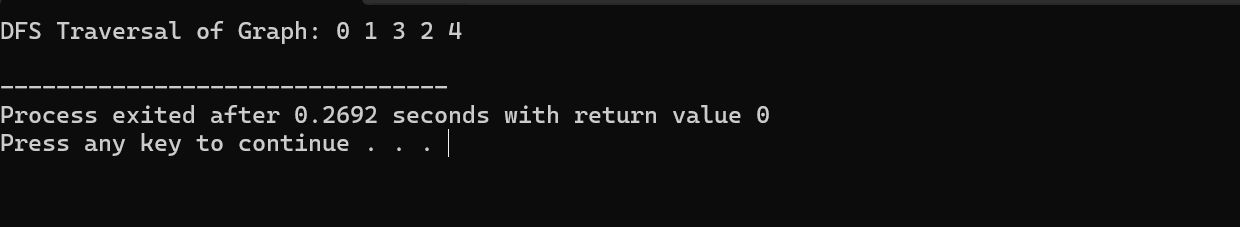


Lab 13

Part 2

### ****DFS in Graph****:

1. In **graph DFS**, we start at a node and explore as deeply as possible along each branch.
2. This algorithm can be **recursive** or use an explicit stack.
3. DFS explores each node’s neighbors before moving on to the next node.
4. It handles **cycles** and **disconnected graphs** by keeping track of visited nodes.
5. DFS is useful for tasks like finding strongly connected components or pathfinding.
6. **O(V + E)** time complexity, where V is vertices and E is edges.
7. DFS can also be adapted to find topological order in directed graphs.

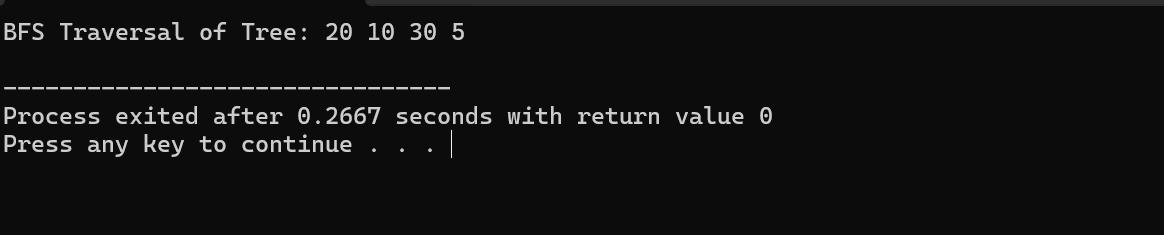


Lab 13

Part 3

### ****BFS in Tree****:

1. **Breadth-first search (BFS)** starts at the root node and explores all nodes level by level.
2. It uses a **queue** to store nodes at each level.
3. BFS visits each node at a particular depth before moving to the next depth level.
4. This approach is useful for finding the shortest path in **unweighted trees**.
5. BFS ensures that nodes are processed in order of their distance from the root.
6. **O(n)** time complexity, where n is the number of nodes in the tree.
7. BFS is often used in algorithms like level-order traversal of a tree.



Lab 13

Part 4

### ****BFS in Graph****:

1. **Graph BFS** starts from a given node and explores all neighboring nodes level by level.
2. BFS uses a **queue** to store nodes, ensuring nodes are visited level by level.
3. It’s ideal for finding the **shortest path** in an unweighted graph.
4. BFS visits all nodes at the current depth before moving to the next.
5. The algorithm handles both **connected** and **disconnected graphs** by marking visited nodes.
6. **O(V + E)** time complexity, where V is the number of vertices and E is the number of edges.
7. BFS is also useful for tasks like finding the **minimum spanning tree** or solving puzzles.

