

***Name : Arbab javed***

***Roll NO : 053***

***Subject : DSA (Lab)***

***LAB TASKS***

**Lab 10**

**Stack with array:**

**What is a stack?**

A stack is a container that follows the Last-In-First-Out (LIFO) principle, meaning the last item added is the first one to be removed.

**Code explanation:**

1. We create a Stack class with an array arr to store the stack elements, and an integer top to keep track of the top element's index.

2. The push function adds an element to the top of the stack.

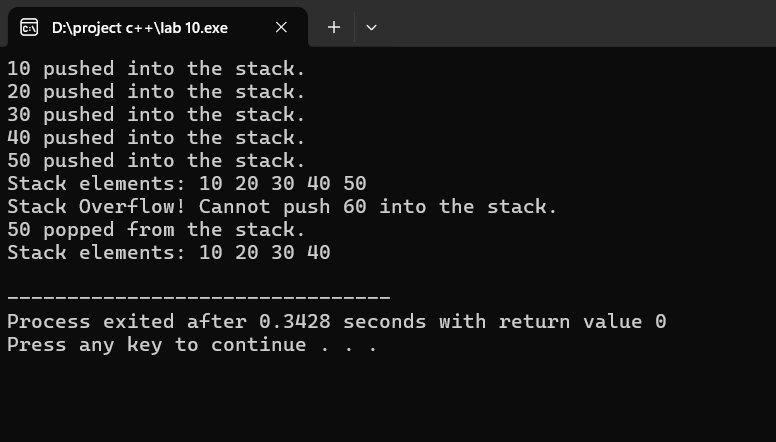
3. The pop function removes the top element from the stack.

4. The peek function returns the value of the top element without removing it.

**How it works:**

1. When you call push(5), the element 5 is added to the top of the stack, and top is incremented.

2. When you call pop(), the top element is removed, and top is decremented.



**Stack with linked list:**

**Code explanation**:

1. We create a Node class to represent each plate (element) in the stack.

2. The Stack class has a top pointer that points to the top plate.

3. push adds a new plate on top.

4. pop removes the top plate.

**How it works:**

1. push(1): adds plate 1 on top.

2. push(2): adds plate 2 on top of plate 1.

3. pop(): removes plate 2.

4. peek(): shows what's on the top plate (plate 1).

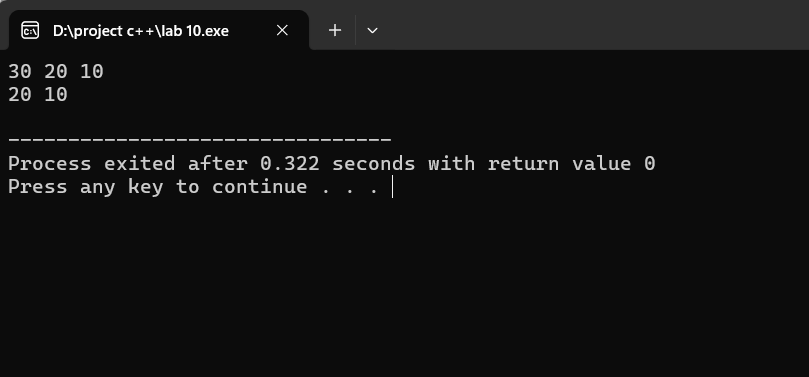
**Why use a stack?**

A stack is useful when you need to follow the Last-In-First-Out (LIFO) principle, like

- Undo/Redo functionality

- Evaluating postfix expressions

- Function call stacks



**LAB 11**

**Queue with array:**

**Queue Code:**

enqueue: Add item to end

dequeue: Remove item from front

peek: Look at front item

**How it works:**

1. Add items: 1, 2, 3

2. Queue: 1, 2, 3

3. dequeue: Remove 1, queue: 2, 3

4. peek: Look at front item (2)

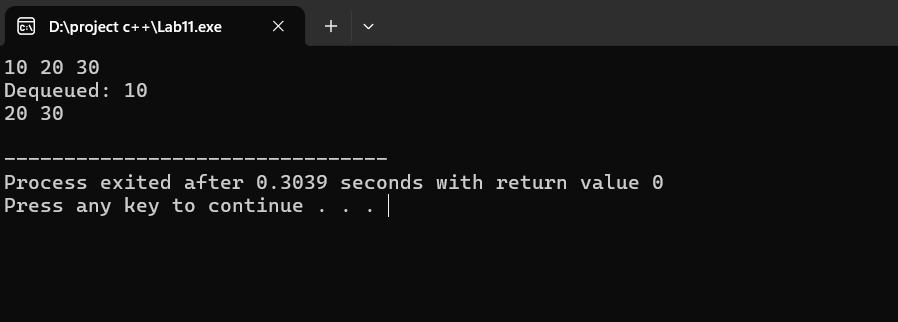
**Why use a queue?**

Useful for

- Job scheduling

- Print queues

- Network requests



**Queue with linked list :**

**Queue Code:**

enqueue: Add node to end

dequeue: Remove node from front

**How it works:**

Create nodes: 1, 2, 3

Queue: 1 -> 2 -> 3

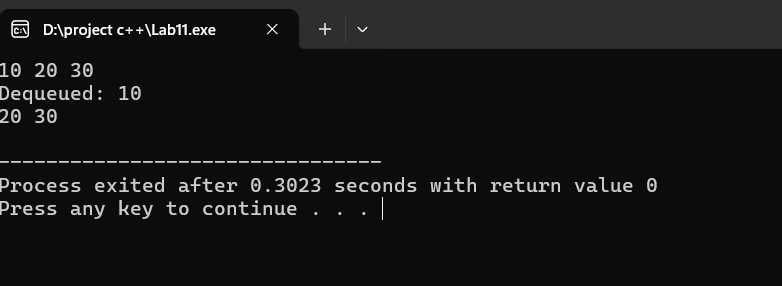
dequeue: Remove 1, queue: 2 -> 3

enqueue(4): Add 4 to end, queue: 2 -> 3 -> 4

**Why use a linked list for a queue?**

Dynamic size: Can grow or shrink as needed

Efficient insertion and removal: Can add or remove nodes at any time



**LAB 12**

**Insert and traverse BST :**

**Binary Search Tree (BST) Code:**

insert: Add node to tree

traverse: Visit nodes in order (inorder, preorder, postorder)

**How it works:**

1. Create root node: 5

2. insert(3): Add 3 to left of 5

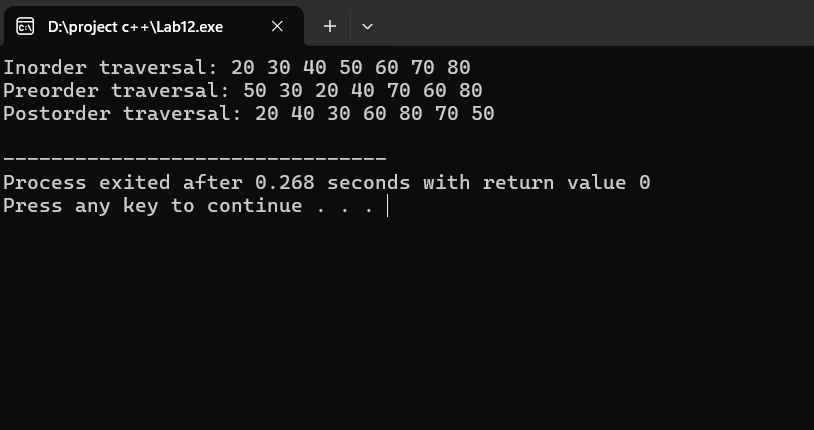
3. insert(7): Add 7 to right of 5

4. inorder traverse: Visit nodes in order: 3, 5, 7

**Why use a BST**?

Fast search: Find nodes quickly (O(log n) time)

Efficient insertion and deletion: Add or remove nodes efficiently



**Insert and Traverse for AVL:**

**AVL Tree Code:**

- insert: Add node to tree, balance if needed

- traverse: Visit nodes in order (inorder)

**How it works:**

Create root node: 5

insert(3): Add 3 to left of 5

insert(7): Add 7 to right of 5

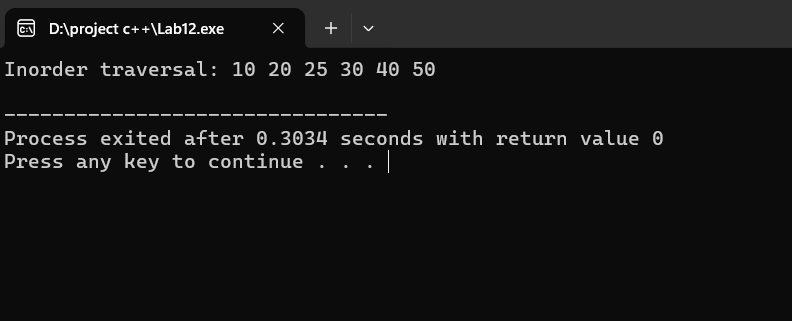
If tree becomes unbalanced, rotate nodes to balance

**Why use an AVL tree?**

- Keeps tree height relatively small

- Find nodes quickly (O(log n) time)

- Add or remove nodes efficiently



**LAB 13**

**Traverse DFS in Tree:**

**types of DFS:**

1. Preorder: Root, Left, Right

2. Inorder: Left, Root, Right

3. Postorder: Left, Right, Root

**How it works:**

1. Start at root node

2. Explore as far as possible along each branch

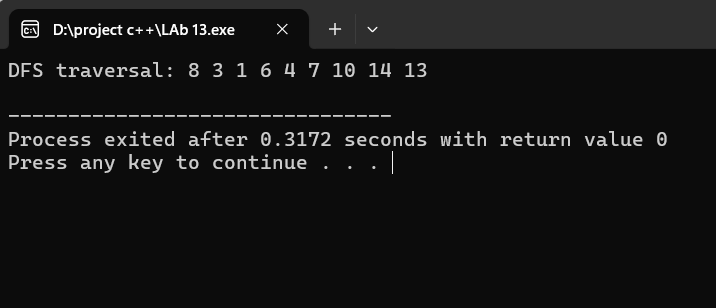
3. Backtrack when reaching a dead end

**Why use DFS?**

- Find a path between two nodes

- Topological sorting

- Finding connected components



**Traverse DFS in Graph:**

**DFS (Depth-First Search) in Graph:**

- Traverse graph depth-first: Explore as far as possible along each edge

**How it works:**

1. Choose a starting node (vertex)

2. Explore neighboring nodes

3. Mark visited nodes to avoid revisiting

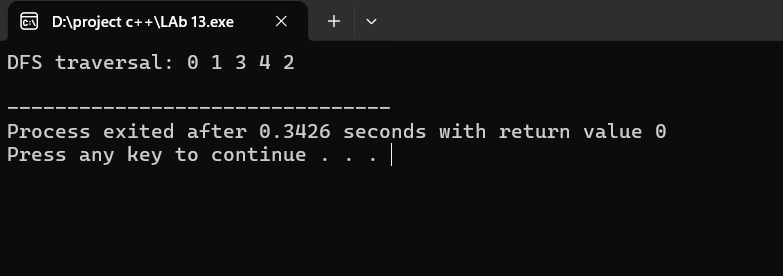
4. Backtrack when reaching a dead end

**Why use DFS?**

- Find connected components

- Topological sorting

- Finding paths between nodes



**Traverse BFS in Tree:**

**BFS (Breadth-First Search) in Tree:**

- Traverse tree level by level, starting from root

**How it works:**

1. Start at root node

2. Visit all nodes at current level

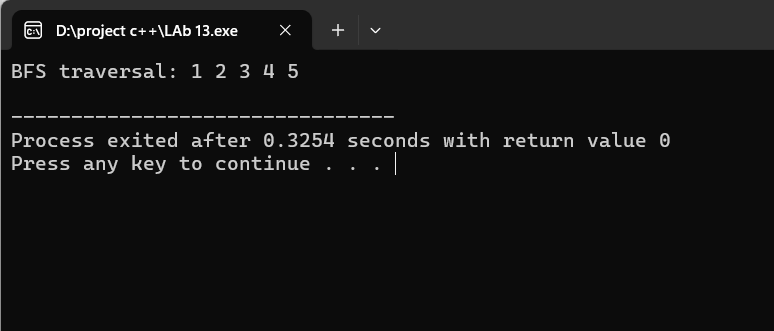
3. Move to next level and repeat

**Why use BFS?**

- find shortest path between nodes

- web pages level by level

- Find friends within a certain distance



**Traverse BFS in Graph :**

**BFS (Breadth-First Search) in** Graph:

- Traverse graph level by level, starting from a node

**How it works:**

1. Choose a starting node (vertex)

2. Visit all neighboring nodes

3. Mark visited nodes to avoid revisiting

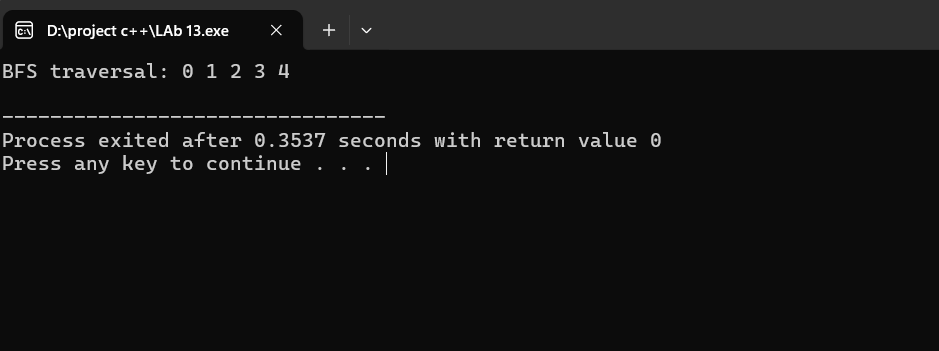
4. Move to next level and repeat

**Why use BFS?**

- Find shortest path between nodes

- Network topology discovery

- Social network analysis



**LaB 14**

**Project : Sudoko solver game**

**Important topics DSA used in project:**

Linked list

Backtracking Algorithm

Recursion

2D Array

