# ASSIGNMENT-11.5

B.VAISHNAVI

2403A52408

Batch-15

# Task 1: Smart Contact Manager

# Part A: Using Arrays (Python List)

### Prompts:

Write Python code for a simple Contact Manager using a list. It should support add, search, and delete functions.

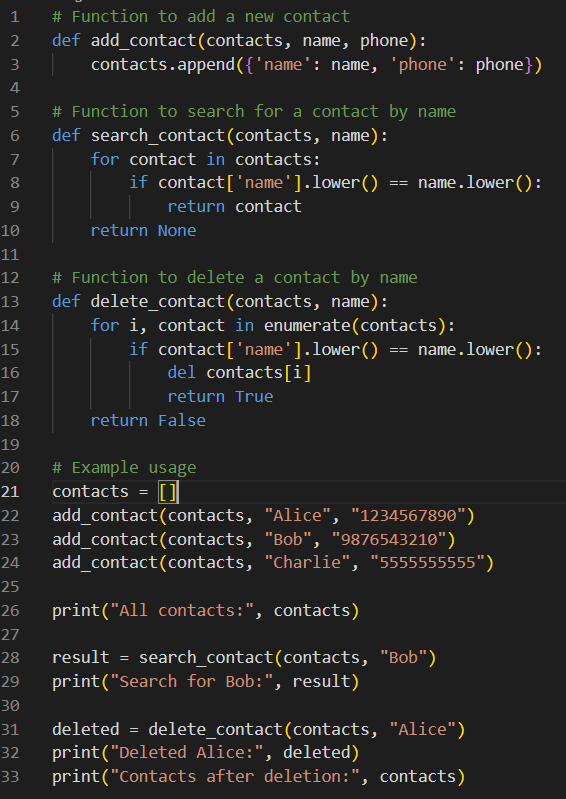
Generate a function add\_contact(name, phone, contacts) to add a contact to the list.

Generate a function search\_contact(name, contacts) to search for a contact by name.

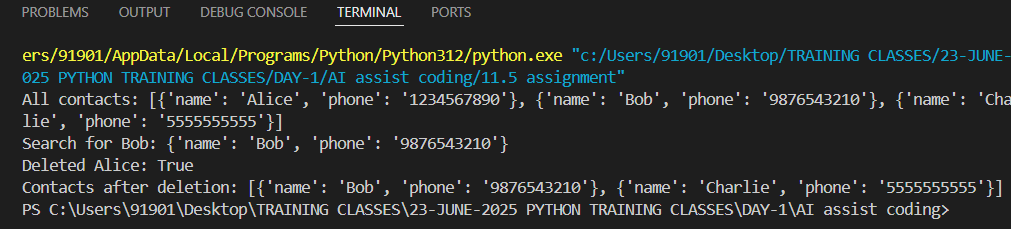
Generate a function delete\_contact(name, contacts) to delete a contact by name.

Show example usage with 3 contacts, including add, search, and delete operations.

CODE:



OUTPUT:



## Observations (Comparison):

**Arrays (Lists in Python):**

Easy to implement.

Search = O(n) (need to scan list).

Delete = O(n) (shifting elements after removal).

Good when size is small and predictable.

# Task 2: Emergency Help Desk (Stack Implementation)

## Prompt:

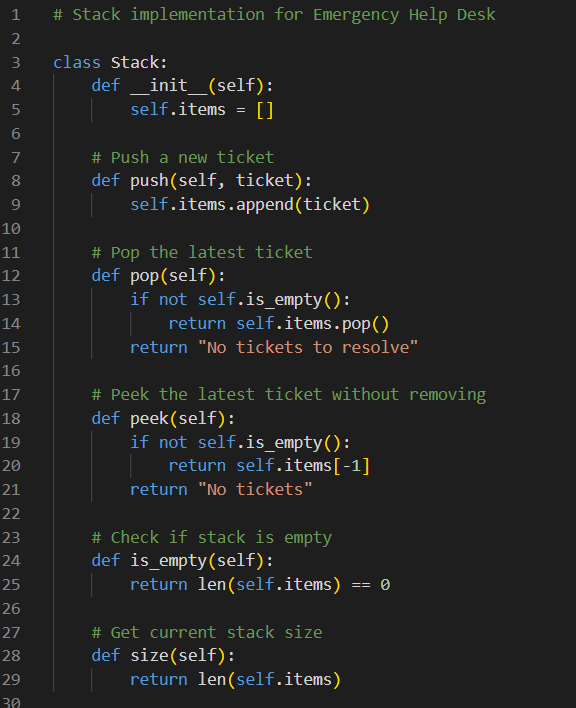
Write Python code to implement a stack class with push, pop, and peek methods.

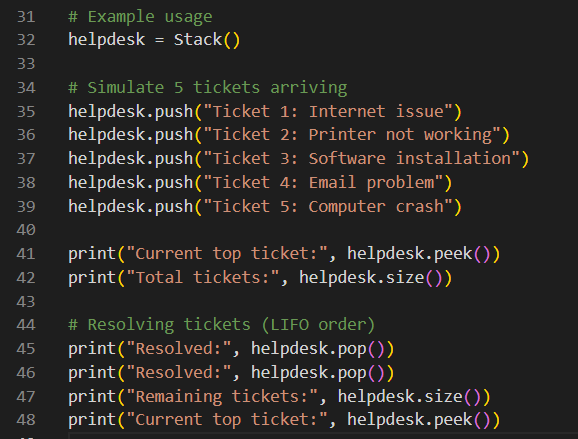
Show how to simulate 5 IT help desk tickets being pushed and popped from the stack.

Add extra stack methods like is\_empty() and size().

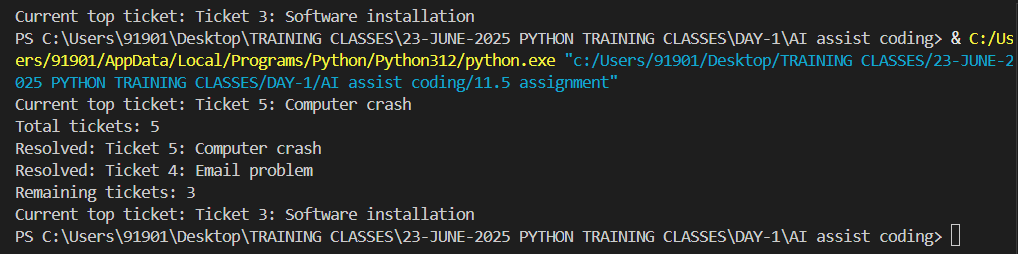
Show example output after adding and resolving tickets.

CODE:





OUTPUT:



OBSERVATION:

Stack follows LIFO (Last In, First Out):

The last ticket received is the first to be resolved.

Operations:

push() → O(1) (just add to end).

pop() → O(1) (remove last).

peek() → O(1) (check last).

Advantages:

Easy to manage urgent issues in reverse order.

Very efficient for insert/remove operations.

Limitation:

You cannot directly access middle tickets without popping others.

# Task 3: Library Book Search (Queues & Priority Queues)

## Prompt:

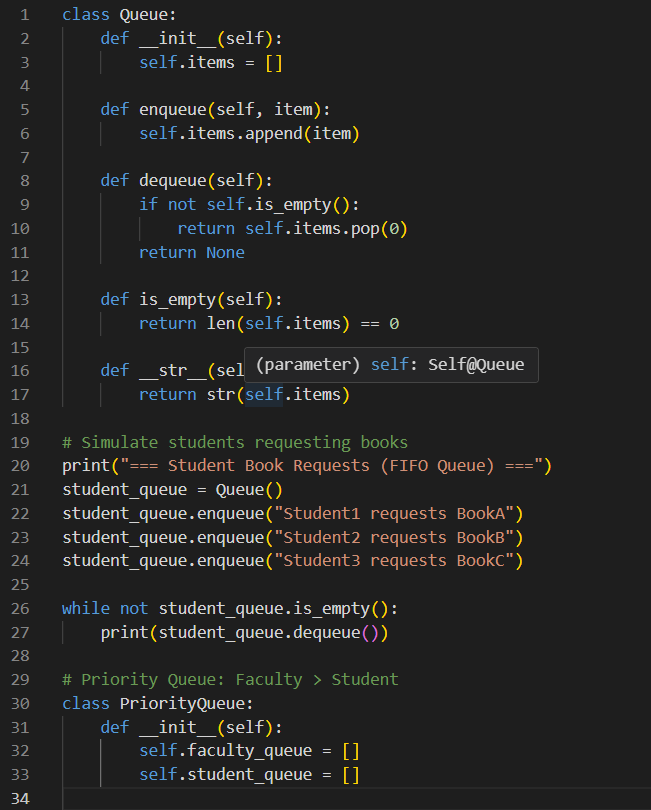
Write Python code for a queue class with enqueue and dequeue methods (FIFO).

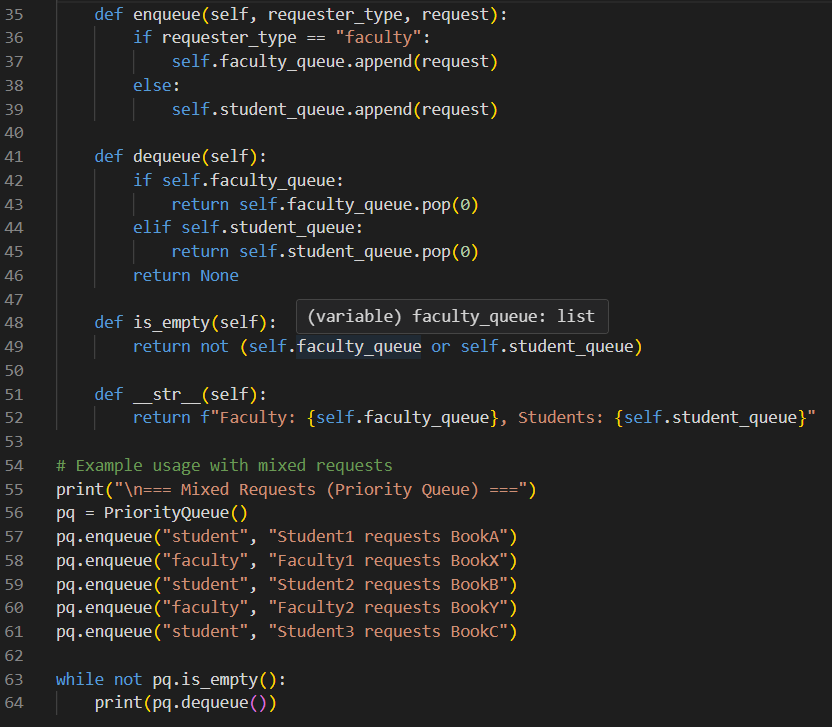
Simulate students requesting books using a queue.

Extend the queue into a priority queue where faculty requests get higher priority than student requests.

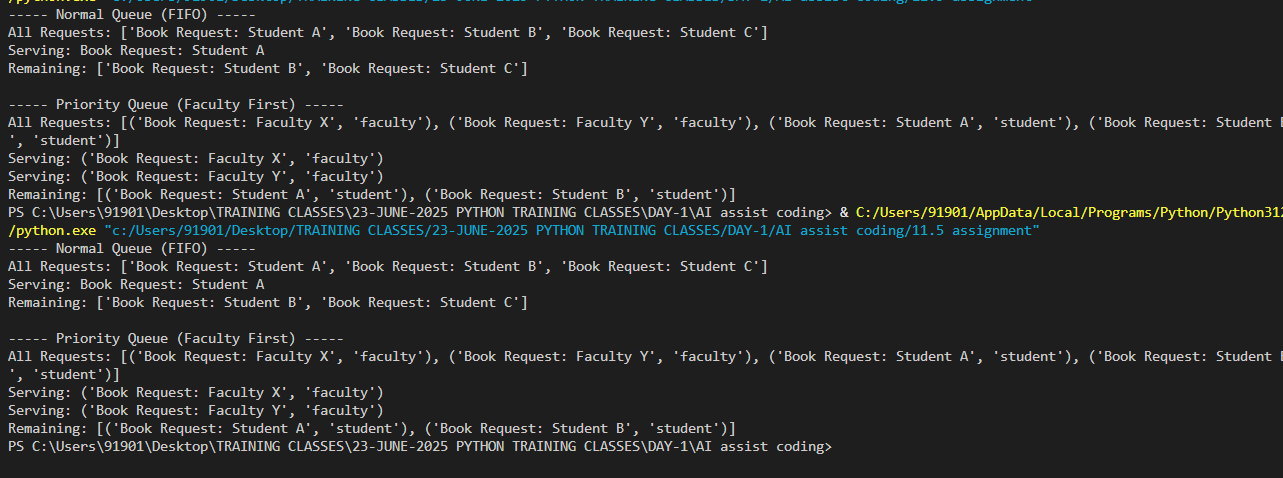
Show example usage with a mix of student and faculty requests.

CODE:





OUTPUT:



OBSERVATION:

Queue (FIFO):

First student who requested → gets served first.

Works like a line/queue in real life.

Operations:

enqueue() = O(1)

dequeue() = O(n) in Python list (because it shifts elements).

Priority Queue:

Faculty requests served before students, no matter order.

Useful for real-world scenarios where priority matters.

Slightly more complex to manage but ensures fairness for priority groups.

Conclusion:

Normal Queue → simple, first-come-first-serve.

Priority Queue → faculty first, then students. Better for library system.

# Task 4: Navigation Assistant (Trees & Graphs)

## Prompt:

Write Python code to implement a Binary Search Tree (BST) with insert, search, and traversal (inorder, preorder, postorder).

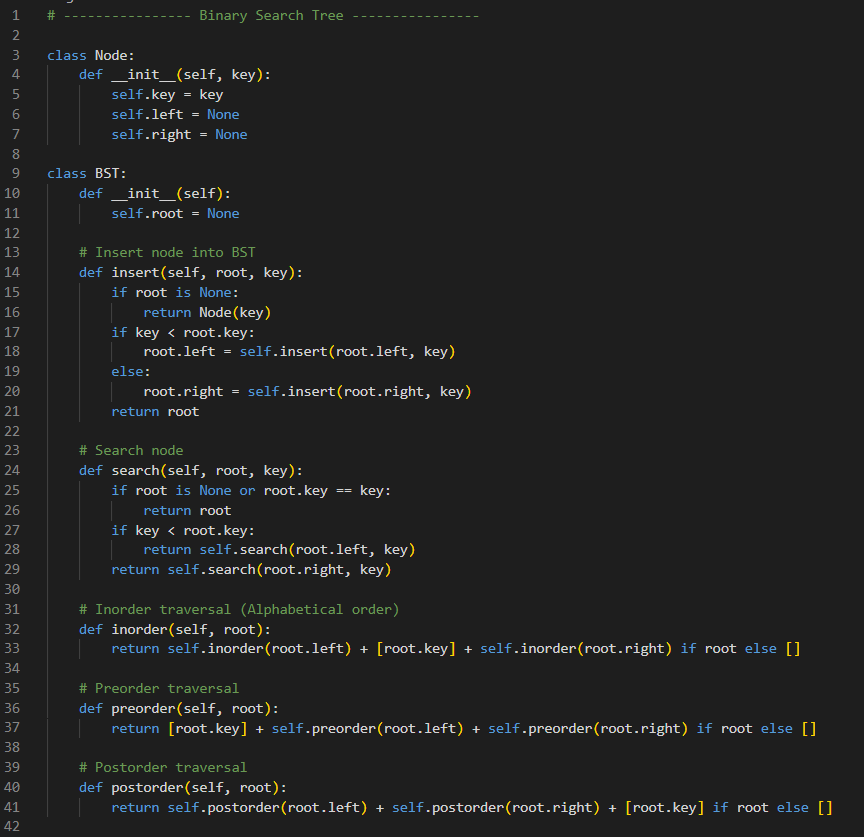
Show how to store building names in alphabetical order using BST and display traversals.

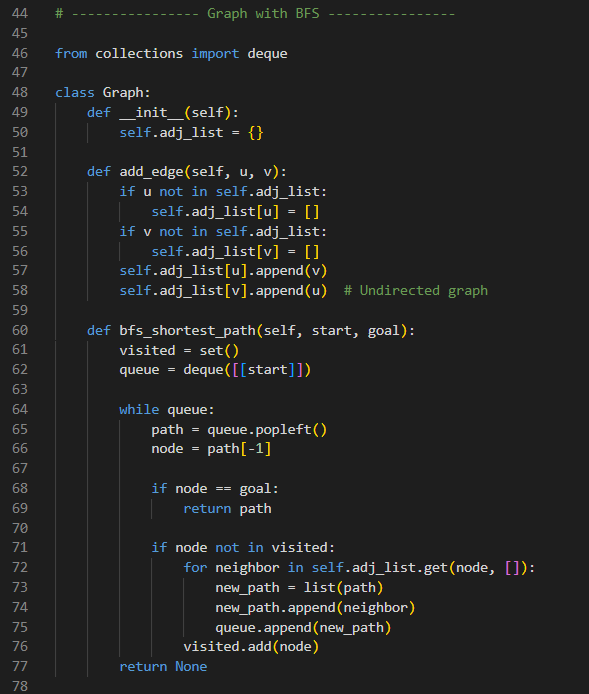
Write Python code to implement a graph with adjacency list representation.

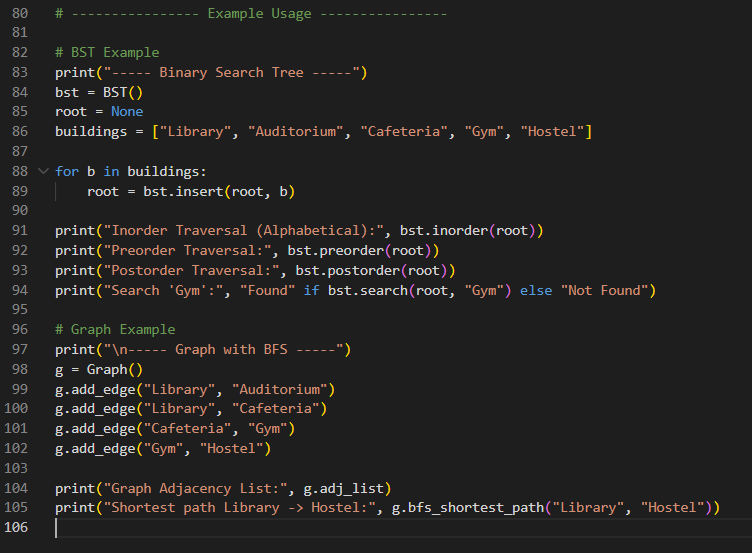
Add a method to find the shortest path between two nodes using BFS.

Demonstrate graph navigation between buildings/rooms.

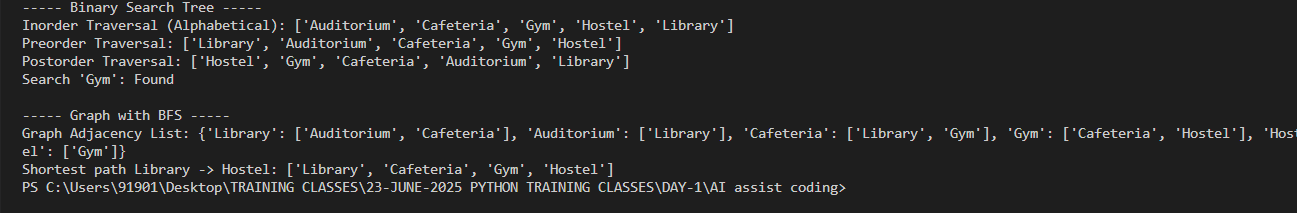
CODE:







OUTPUT:



Observations:

Binary Search Tree (BST):

Stores buildings in sorted order.

Traversals:

Inorder → Alphabetical order.

Preorder → Root first, then left-right.

Postorder → Children first, then root.

Search is efficient (O(log n) average).

Graph with BFS:

Models buildings/rooms as nodes and paths as edges.

BFS finds the shortest path (minimum steps).

Example: Shortest path from Library to Hostel goes via Cafeteria and Gym.

Conclusion:

BST is good for storing data in sorted order (easy lookup).

Graph with BFS is best for navigation problems (finding routes).