

Assignment 11.2

Name : J. Naveen kumar

HT.NO : 2303A510F5

B.NO : 16

Task Description -1 – (Stack Using AI Guidance)

- Task: With the help of AI, design and implement a Stack data structure supporting basic stack operations.

Expected Output:

- A Python Stack class supporting push, pop, peek, and empty-check operations with proper documentation.

The screenshot shows a VS Code interface with the following details:

- File Explorer:** Shows files task1.py, task2.py, task3.py, task4.py, and task5.py.
- Editor:** task1.py is open, displaying code for a Stack class. The code includes docstrings for push, pop, peek, and is_empty methods, and example usage at the bottom.
- Terminal:** Shows command-line interactions for pushing (10, 20, 30), popping, peeking, and checking if the stack is empty.
- Chat Panel:** A sidebar titled "DESIGNING A STACK DATA STRUCTURE IN PYTHON" contains a tip about using Plan mode, requirements (Create a Stack class, Implement push(), pop(), peek(), and is_empty() methods, Include proper comments explaining each method, Add example usage with sample output, Also explain time complexity of each operation), and a link to lab1.py.
- Status Bar:** Shows file paths, line numbers (Ln 10, Col 64), tab sizes, encoding, Python version (3.13.1), and other development tools.

Implemented a Stack class backed by a Python list.

Methods:

- `push(item)`: adds an element to the top.
 - `pop()`: removes and returns the top element, raises `IndexError` if empty.
 - `peek()`: returns (without removing) the top element, raises `IndexError` if empty.
 - `is_empty()`: returns `True` if there are no elements.

Task Description -2 – (Queue Design)

- Task: Use AI assistance to create a Queue data structure following FIFO principles

Expected Output:

- A complete Queue implementation including enqueue, dequeue, front element access, and size calculation

The screenshot shows a browser-based development environment with several tabs open:

- task1.py
- task2.py (highlighted)
- task3.py
- task4.py
- task5.py

The task2.py tab contains Python code for implementing a Queue class using collections.deque. The code includes comments explaining the implementation and example usage.

```
AI-ASSISTED_CODING > task2.py > ...
1     """Queue implementation in Python following FIFO (First-In, First-Out) principle.
2
3     This module defines a Queue class with the following operations:
4     - enqueue(item)
5     - dequeue()
6     - front()
7     - size()
8
9     It also includes example usage at the bottom.
10    """
11
12    from collections import deque
13
14
15    class Queue:
16        """A simple FIFO (First-In, First-Out) queue data structure.
17
18        Internally this uses collections.deque for efficient O(1) operations
19        at both ends.
20        """
21
22        def __init__(self):
23            """Initialize an empty queue.
24
```

The terminal tab shows the command `python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task2.py"`. The output indicates that the queue is empty and shows the enqueueing of elements 'A', 'B', and 'C'.

```
C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task2.py"
Is queue empty? True

Enqueuing elements: 'A', 'B', 'C'
Current queue: Queue(['A', 'B', 'C'])

Front element: A

Dequeuing elements (FIFO order):
Dequeued: A
Dequeued: B
Current queue: Queue(['C'])

Queue size: 1
```

The right side of the interface features a sidebar with icons for file operations, a search bar, and a "CHAT" section. The "CHAT" section displays a user message about creating a Queue data structure, followed by an AI-generated response with bullet points and explanatory text. Below the AI response, there are several checkmarks indicating tasks completed.

CHAT

DESIGNING A STACK DATA STRUCTURE IN PYTHON

Restore Checkpoint

Act as a data structures instructor.
Create a Queue data structure in Python following FIFO principles.
Requirements:

- Create a Queue class, implement enqueue(), dequeue(), front(), and size() methods. Add proper comments. Provide example usage with output. Explain how FIFO works in this implementation. Mention time complexity.

Click to Edit

task2.py

✓ Implementing queue in task2.py

I'll open your task2.py and then implement a commented Queue class with example usage and time complexity notes.

✓ Reviewed task2.py

Now I'll write the Queue implementation, comments, explanation, and example usage into task2.py.

✓ Updated task2.py

Here's your Queue implementation and how it satisfies each requirement:

- Implemented a Queue class in task2.py using collections.deque for efficient operations.
- Methods:
 - enqueue(item): adds an element to the rear of the queue (FIFO).

task5.py

GPT-5.1

Describe what to build next

GPT-5.1

Python 3.13 (64-bit)

Go Live

Windsurf: ...

Prettier

- Implemented a Queue class using collections.deque for efficient operations.
 - Methods:

`enqueue(item)`: adds an element to the rear of the queue (FIFO).

`dequeue()`: removes and returns the front element; raises `IndexError` if empty.

`front()`: returns (without removing) the front element; raises `IndexError` if empty.

`size()`: returns the current number of elements.

`is_empty()`: helper to check if the queue has no elements.

Task Description -3 – (Singly Linked List Construction)

- Task: Utilize AI to build a singly linked list supporting insertion and traversal.

Expected Output:

- Correctly functioning linked list with node creation, insertion logic, and display functionality.

- Added Node and LinkedList classes.
 - Node holds data and a next reference that points to the next node (or None for the last node).
 - LinkedList:

- Maintains head (first node).
- `insert_at_end(data)`: creates a new node and links it at the end by walking from head to the last node and updating its next.
- `traverse()`: walks from head via next, collects data values into a Python list, and returns it.
- `display()`: prints the list in the form 10 -> 20 -> 30 -> None.

Task Description -4 – (Binary Search Tree Operations)

- Task: Implement a Binary Search Tree with AI support focusing on insertion and traversal.

Expected Output:

- BST program with correct node insertion and in-order traversal output.

The screenshot shows a code editor with multiple tabs open: task1.py, task2.py, task3.py, task4.py (the active tab), and task5.py. The code in task4.py implements a Binary Search Tree (BST) with Node and BinarySearchTree classes. The Node class has an __init__ method that initializes data, left, and right pointers. The BinarySearchTree class has an __init__ method that initializes root to None, and an insert method that performs recursive insertion. The code includes docstrings and comments explaining the implementation.

On the right side of the screen, there is a Chat interface with the following content:

DESIGNING A STACK DATA STRUCTURE IN PYTHON

Act as a data structures professor.
Help me implement a Binary Search Tree in Python.
Requirements:

- Create a Node class
- Implement BST insertion method
- Implement in-order traversal
- Add proper comments explaining recursion
- Provide example with sample values
- Show expected traversal output
- Explain why in-order traversal gives sorted output

task4.py

✓ Implementing BST in Python
I'll open task4.py (currently empty) and then add a Binary Search Tree implementation with insertion, in-order traversal, comments, and example usage.

✓ Reviewed task4.py
Now I'll write the BST Node and tree classes, plus example and explanations, into task4.py.

✓ Updated task4.py
Here's what I've implemented for your Binary Search Tree:

- Added `Node` and `BinarySearchTree` classes in `AI-ASSISTED_CODING/task4.py`.
- `Node` holds `data`, `left`, and `right` references.
- `BinarySearchTree`:

task5.py task4.py
Describe what to build next
GPT-5.1

Added Node and BinarySearchTree classes in AI-ASSISTED_CODING/task4.py.

Node holds data, left, and right references.

BinarySearchTree:

`insert(value)`: public insert method using `_insert_recursive(node, value)`:

Base case: if node is None, create and return a new Node.

Recursive case: go left if value < node.data, right if value > node.data, then return node to maintain links.

`inorder traversal()`: returns a list of values; uses `_inorder recursive(node, result)`:

Base case: node is None → return.

Recursive case: traverse left, visit node (append data), traverse right.

Comments in both recursive helpers explain base/recursive cases clearly.

Task Description -5 – (Hash Table Implementation)

- Task: Create a hash table using AI with collision handling

Expected Output:

- Hash table supporting insert, search, and delete using chaining or open

The screenshot shows a code editor with multiple tabs open, including task1.py, task2.py, task3.py, task4.py, and task5.py. The task5.py tab is active, displaying Python code for a HashTable class. The code includes comments explaining collision handling and provides sample test cases for insertion, search, and deletion.

```

AI-ASSISTED_CODING > task5.py ...
113     def _example_usage():
114         """Run sample test cases to demonstrate chaining and basic operations.
115
116         Collision handling explanation:
117         - Different keys can produce the same hash index (after modulo).
118         - All such keys are stored in the same bucket list.
119         - When we search or delete, we restrict our scan to that bucket only.
120
121     # Small capacity to increase chances of collision for demonstration
122     ht = HashTable(capacity=5)
123
124     print("Inserting key-value pairs:")
125     ht.insert("apple", 1)
126     ht.insert("banana", 2)
127     ht.insert("grape", 3)
128     ht.insert("grape", 3)
129
130     # Insert keys that are likely to collide by design (Integers)
131     ht.insert(10, "ten")
132     ht.insert(15, "fifteen") # 10 and 15 collide when capacity=5 (10% == 15%)
133
134     print("Hash table internal state (buckets):")
135     print(ht)
136
137     print("Searching for keys:")
138     print(search("apple")) ->, ht.search("apple")
139     print(search("banana")) ->, ht.search("banana")
140     print(search("grape")) ->, ht.search("grape")
141     print(search("grape")) ->, ht.search("grape")
142     print(search(15)) ->, ht.search(15)
143     print(search("unknown")) ->, ht.search("unknown") # Not present
144
145     print("Deleting keys:")
146     print(delete("banana")) ->, ht.delete("banana")
147     print(delete(10)) ->, ht.delete(10)
148     print(delete("unknown")) ->, ht.delete("unknown") # Not present
149
150     print("Hash table after deletions:")
151     print(ht)
152
153     # Expected sample output (exact bucket order may differ due to hash values):
154     # Inserting key-value pairs:
155     # Hash table internal state (buckets):
156     # HashTable([(('apple', 1),), (('banana', 2)), [(_('grape', 3)), ((10, 'ten'), (15, 'fifteen'))], []])
157     #
158     # Searching for keys:
159     # search('apple') -> 1
160     # search('banana') -> 2
161     # search('grape') -> 3
162     # search(10) -> ten
163     # search(15) -> fifteen
164     # search('unknown') -> None
165
166     # Deleting keys:
167     # delete('banana') -> True
168     # delete(10) -> True
169     # delete('unknown') -> False

```

On the right side of the screen, there is a Chat interface titled "DESIGNING A STACK DATA STRUCTURE IN PYTHON". It contains a message from an AI expert asking for help in implementing a Hash Table in Python with collision handling using chaining. Below the message, there are requirements listed:

- Create a HashTable class
- Implement insert(), search(), and delete() methods
- Use chaining (list of lists)
- Add detailed comments
- Provide sample test cases with output
- Explain how collision handling works
- Mention average and worst-case time complexity

The Chat interface also shows a note about planning the hash table implementation and a review of the task5.py code.

- HashTable class with:

- `__init__(capacity=10)`: creates capacity buckets, each an empty list.
- `_hash(key)`: uses Python `hash(key) % capacity` to pick a bucket index.
- `insert(key, value)`: updates existing key or appends (key, value) into the bucket.
- `search(key)`: scans the bucket for key, returns the value or `None`.

- `delete(key)`: removes (key, value) from the bucket, returns True/False.
- Chaining:
 - `self.table` is a list of lists (buckets).
 - Each bucket stores multiple (key, value) pairs that share the same index → this is collision handling by chaining.