

# Assignment 11.2

Name :N.Akhil

HT.NO : 2303A51065

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 Visual Studio Code (VS Code) interface with the following details:

- File Explorer:** Shows files task1.py, task2.py, task3.py, task4.py, and task5.py.
- Editor:** The active file is task1.py, containing code for a Stack class. The code includes docstrings and comments explaining the implementation.
- Terminal:** The terminal tab is selected, showing command-line interactions with the code. It includes commands like "Pushing elements: 10, 20, 30", "Current stack: Stack([10, 20, 30])", "Top element (peek): 30", "Popping elements: Popped: 30", "Popped: 20", "Current stack: Stack([10])", "Is stack empty? False", and "Stack size: 1".
- Chat Panel:** The right-hand panel shows AI assistance for "DESIGNING A STACK DATA STRUCTURE IN PYTHON". It includes 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 summary of changes made to lab1.py.
- Status Bar:** The status bar at the bottom shows "Ln 10, Col 64 Tab Size: 4 UTF-8 CRLF () Python 3.13.1 Python 3.13 (64-bit) Go Live Windsurf: (...) Prettier".

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

File Edit Selection View Go Run Terminal Help ← →

AI-ASSISTED\_CODING > task2.py > ...

```
task1.py task2.py x task3.py task4.py task5.py
```

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

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

Ln 129, Col 1 Spaces: 4 UTF-8 CRLF { Python task5.py task2.py

Describe what to build next

GPT-5.1 ↻

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).

- 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.

The screenshot shows a Microsoft DevOps interface with several tabs open. The active tab is 'task3.py' under the 'AI-ASSISTED\_CODING' workspace. The code implements a singly linked list with methods for insertion at the end and traversal.

```
task1.py task2.py task3.py x task4.py task5.py
```

```
AI-ASSISTED_CODING > task3.py > ...
```

```
1  """Simple Implementation of a singly Linked List in Python.
2
3     This file defines:
4         - Node class: represents a single node in the list
5         - LinkedList class: manages the list of nodes
6
7     Features implemented:
8         - Insertion at the end of the list
9         - Traversal / display of all elements
10
11    It also includes example usage at the bottom.
12
13    Requirements:
14        - Create a Node class
15        - Create a LinkedList class
16        - Implement insertion at end
17        - Implement traversal/display method
18        - Add detailed comments explaining logic
19        - Provide sample input and output
20        - Explain how nodes are connected internally
21
22    class Node:
23        """A single node of a singly linked list.
24
25            Each node stores:
26                - data: the value we want to keep
27                - next: a reference (link) to the next node in the list, or None
28
29            __init__(self, data):
30                self.data = data
31                self.next = data
32                # next will point to the next node in the list.
33                # If the list is empty, the next node is not yet connected. It is None.
34                self.next = None
35
36    class LinkedList:
37        """Singly Linked List Implementation.
38
39        Internally, the list is represented by a chain of Node objects.
40        This class maintains the head of the list, a first node, called 'head'.
41        Each node points to the next one, forming a chain:
42
43            head -> [data | next] -> [data | next] -> ... -> None
44
45        def __init__(self):
46            # head points to the first node in the list.
47            # If the list is empty, the head is None.
48            self.head = None
49
50        def insert_at_end(self, data):
51            """Insert a new node with the given data at the end of the list.
52
53            Steps:
54            1. Create a new node.
55            2. If the list is empty, (head is None), make this node the head.
56            3. Otherwise, traverse to the last node and link its 'next' to the new node.
57
58            new_node = Node(data)
59
60            # Case 1: The list is empty; the new node becomes the first node (head).
61            if self.head is None:
62                self.head = new_node
63                return
64
65            # Case 2: The list is not empty; find the last node.
66            current = self.head
67
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

(c) Microsoft Corporation. All rights reserved.

```
C:\Users\akhil\OneDrive\Documents\Devops\python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task3.py"
Inserting elements at the end: 10, 20, 30

Linked list (display):
10 -> 20 -> 30 -> None

Linked list as Python list (traverse):
[10, 20, 30]

C:\Users\akhil\OneDrive\Documents\Devops>
```

Ln 129, Col 21 Spaces:4 UTF-8 CRLF [ ] Python 3.13.1 Python 3.13 (64-bit) (i) Go Live Windsurf: (...) Prettier

- 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 task4.py tab contains the following Python code:

```

AI-ASSISTED_CODING > task4.py > ...
1  """Binary Search Tree (BST) implementation in Python.
2
3 This file defines:
4 - Node class: represents a single node in the BST
5 - BinarySearchTree class: supports insertion and in-order traversal
6
7 Features implemented:
8 - Insert a value into the BST
9 - In-order traversal (left, Root, Right)
10
11 Example usage is provided at the bottom.
12 """
13
14
15 class Node:
16     """A single node in a Binary Search Tree.
17
18     Each node stores:
19         - data: the value stored at this node
20         - left: reference to the left child (values < data)
21         - right: reference to the right child (values > data)
22     """
23
24     def __init__(self, data):
25         self.data = data
26         self.left = None
27         self.right = None
28
29
30 class BinarySearchTree:
31     """Binary Search Tree (BST) with recursive insertion and in-order traversal."""
32
33     def __init__(self):
34         # Root of the BST. Initially, the tree is empty so root is None.
35         self.root = None
36
37     def insert(self, value):
38         """Insert a value into the BST.

```

The terminal below shows the code being run and the resulting in-order traversal output:

```

(c) Microsoft Corporation. All rights reserved.

C:\Users\akhil\OneDrive\Documents\Devops\xpython -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task4.py"
Inserting values into BST: [50, 30, 70, 20, 40, 60, 80]

In-order traversal result:
[20, 30, 40, 50, 60, 70, 80]

Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]

```

The status bar at the bottom indicates the code is running in Python 3.13 (64-bit) with GPT-5.1 AI support.

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")) #>
147     print(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.

At the bottom of the terminal window, the command used to run the script is shown: C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED\_CODING\task5.py"

- 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.