

# Assignment11.2

Name: T. Lokesh

HT.NO:2303A51079

B.NO : 16

## TaskDescription-1-(StackUsingAIGuidance)

- 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, containing code for a Stack class. The code includes docstrings for the class and its methods, and example usage at the bottom.
- Terminal:** Displays command-line interactions with the Stack class, showing pushing elements (10, 20, 30), popping elements, peeking at the top element (30), and checking if the stack is empty (False).
- Chat Panel:** A GPT-5.1 AI assistant is providing guidance. It suggests creating a Stack class with methods push(), pop(), peek(), and is\_empty(). It also asks for proper comments, example usage with output, and time complexity analysis.
- Status Bar:** Shows file path (AI-ASSISTED\_CODING > task1.py > Stack), line 10, column 64, tab size 4, encoding UTF-8, CRLF, Python 3.13.1, Python 3.13 (64-bit), Go Live, Windurf, and 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

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

- File Explorer:** Shows files task1.py, task2.py (marked as modified), task3.py, task4.py, and task5.py.
- Editor:** task2.py is open, displaying code for a Queue class. The code uses the collections.deque module for efficient O(1) operations at both ends. It includes methods for enqueue, dequeue, front, and size, along with example usage and time complexity notes.
- Terminal:** Shows command-line output of the Python script running task2.py, demonstrating the implementation's functionality.
- AI Assistant:** A sidebar titled "DESIGNING A STACK DATA STRUCTURE IN PYTHON" provides requirements and tasks for the Queue implementation.
- Status Bar:** Shows file path (C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED\_CODING\task2.py"), line count (Ln 129, Col 1), and other development tools like GPT-5.1 and 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.

## **TaskDescription-3-(SinglyLinkedListConstruction)**

- Task: Utilize AItobuild a singly linked list supporting insertion and traversal.

## **ExpectedOutput:**

- 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.

## **TaskDescription-4-(BinarySearchTreeOperations)**

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

## **ExpectedOutput:**

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

The screenshot shows a Microsoft Visual Studio Code interface with the following details:

- File Explorer:** Shows files task1.py, task2.py, task3.py, task4.py (current tab), and task5.py.
- Code Editor:** Displays the content of task4.py, which implements a Binary Search Tree (BST) with recursive insertion and in-order traversal. The code includes comments explaining the Node class, BST class, and the insert method.
- Terminal:** Shows the command `python -u "c:\Users\akhil\OneDrive\Documents\Devops\task4.py"`, followed by the output: "Inserting values into BST: [50, 30, 70, 20, 40, 60, 80]" and the result of an in-order traversal: "[20, 30, 40, 50, 60, 70, 80]".
- Output Panel:** Shows the message "(c) Microsoft Corporation. All rights reserved."
- Status Bar:** Shows "Ln 145, Col 1" and "Python 3.13 (64-bit)".
- Right Sidebar:** A "CHAT" panel titled "DESIGNING A STACK DATA STRUCTURE IN PYTHON" contains a list of requirements for implementing a BST. Below it, a "task4.py" panel shows a checklist:
  - ✓ Implementing BST in Python
  - ✓ Reviewed task4.py
  - Now I'll write the BST Node and tree classes, plus example and explanations, into task4.py.
  - ✓ Updated task4.py
- Bottom Status:** Shows "Describe what to build next" and "GPT-5.1" with a progress bar.

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):

Basecase: 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 helper explain base/recursive cases clearly.

## **TaskDescription-5-(HashTableImplementation)**

- Task: Create a hashtable using AI with collision handling

## **Expected Output:**

- Hashtables supporting insert, search, and delete using chaining or open

File Edit Selection View Go Run Terminal Help ↗ | Q Devops

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

AI-ASSISTED CODING > task5.py > ...

```
2  """Hash table implementation in Python using chaining for collision handling!"""
3  This module defines a HashTable class that supports:
4  - insert(key, value)
5  - search(key)
6  - delete(key)
7
8  Collision handling is done using "chaining": each bucket is a list of
9  key-value pairs that share the same hash index.
10
11 class HashTable:
12     """Simple hash table with separate chaining collision handling.
13
14     Internally:
15         - We maintain a fixed-size Python list called 'table'.
16         - Each entry in the table is a bucket, implemented as a Python list.
17         - Each bucket stores (key, value) pairs.
18
19     When two different keys hash to the same index, they are stored in
20     the same bucket list. This is called "chaining".
21
22     def __init__(self, capacity=10):
23         """Initialize the hash table with a given number of buckets.
24
25         Args:
26             capacity: Number of buckets (slots) in the hash table.
27
28         Time Complexity: O(capacity) to allocate buckets (once at start).
29
30         self.capacity = capacity
31         # Create a list of empty lists (buckets) for chaining.
32         self.table = [ [] for _ in range(capacity) ]
33
34     def hash(self, key):
35         """Compute hash index for a given key.
36
37         We use Python's built-in hash() and then take modulo capacity
38         to map the hash value into the bucket range [0, capacity-1].
39
40         return hash(key) % self.capacity
41
42     def insert(self, key, value):
43         """Insert or update a key-value pair in the hash table.
44
45         Steps:
46             1. Compute the bucket index using the hash function.
47             2. Look through the bucket to see if the key already exists.
48                 - If found, update its value.
49                 - If not found, append a new (key, value) pair to the bucket.
50
51         Average Time Complexity: O(1)
52         Worst-case Time Complexity: O(n) when many keys collide in one bucket.
53
54         Index = self.hash(key)
55         bucket = self.table[index]
56
57         # Check if key already exists in this bucket; if so, update it.
58         for (k, v) in enumerate(bucket):
59             if k == key:
60                 bucket[enum...
```

CHAT DESIGNING A STACK DATA STRUCTURE IN PYTHON

Act as a Python programming expert.  
Help me implement a Hash Table in Python with collision handling using chaining.

Requirements:

- 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

task5.py

✓ Planning hash table implementation  
I'll open task5.py and then implement a chained HashTable class with insert/search/delete, comments, example tests, and complexity notes.

✓ Reviewed and updated task5.py  
I've implemented your chained hash table and example tests in AI-ASSISTED\_CODING/task5.py.

What's implemented

- 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.

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

Ln 1, Col 78 Spaces: 4 UTF-8 CRLF () Python 3.13.1 Python 3.13 (64-bit) I-I Go Live Windsurf (...) Prettier

The screenshot shows a code editor with multiple tabs open. The active tab is `task5.py`, which contains Python code for a HashTable class. The code includes methods for inserting key-value pairs, searching for keys, and deleting keys. It also includes a `def example_usage()` function and some test cases. The code editor has a dark theme with syntax highlighting.

```

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         - Keys such as 'apple' and 'grape' will both map to bucket 1.
119         - When no search or delete, or restrict our view to that bucket only.
120
121         # Small capacity to increase chances of collision for demonstration
122
123         ht = HashTable(capacity=5)
124
125         print("Inserting key-value pairs:")
126         ht.insert("apple", 1)
127         ht.insert("banana", 2)
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("\nSearching 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(10) ->", ht.search(10))
142         print("search(15) ->", ht.search(15))
143         print("search('unknown') ->", ht.search("unknown")) # Not present
144
145         print("\nDeleting 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("\nHash 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([[], [{}], [[('banana', 2)]], [[('grape', 3)]], [[('apple', 1)], [(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 an AI-assisted coding interface. It shows a sidebar titled "DESIGNING A STACK DATA STRUCTURE IN PYTHON" with requirements like "Create a HashTable class" and "Implement insert(), search(), and delete() methods". Below the sidebar, there is a "CHAT" section with a message from the AI: "Act as a Python programming expert. Help me implement a Hash Table in Python with collision handling using chaining." There is also a "Planner" section with tasks like "Planning hash table implementation" and "Reviewed and updated task5.py". At the bottom right, there is a terminal window showing the command `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.