

SR UNIVERSITY
AI ASSISTED CODING

Name:B.Hasini
2503A51L13

Lab 11 – Data Structures with AI: Implementing Fundamental Structures

Lab Objectives

- Use AI to assist in designing and implementing fundamental data structures in Python.
- Learn how to prompt AI for structure creation, optimization, and documentation.
- Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, Graphs, and Hash Tables.
- Enhance code quality with AI-generated comments and performance suggestions.

Task 1: Implementing a Stack (LIFO)

- **Task:** Use AI to help implement a **Stack** class in Python with the following operations: push(), pop(), peek(), and is_empty().

CODE:


```
5     class Stack:
6         def peek(self):
7             if self.is_empty():
8                 raise IndexError("peek from empty stack")
9             return self._items[-1]
10
11         def is_empty(self):
12             """Check if the stack is empty.
13
14             Returns:
15                 bool: True if the stack is empty, False otherwise.
16             """
17             return len(self._items) == 0
18
19         def __repr__(self):
20             """Return a string representation of the stack."""
21             return f"Stack({self._items})"
22
23
24     class DequeStack:
25         """Stack implementation using collections.deque for optimized performance."""
26
27         def __init__(self):
28             """Initialize an empty deque-based stack."""
29             self._items = deque()
30
31         def push(self, item):
32             """Push an item onto the stack."""
33             self._items.append(item)
34
35         def pop(self):
36             """Remove and return the top item from the stack."""
37             if self.is_empty():
38                 raise IndexError("pop from empty stack")
39             return self._items.pop()
40
41         def peek(self):
42             """Return the top item without removing it."""
43             if self.is_empty():
44                 raise IndexError("peek from empty stack")
45             return self._items[-1]
46
47         def is_empty(self):
48             """Check if the stack is empty."""
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
```

```

7>
80     def is_empty(self):
81         """Check if the stack is empty."""
82         return not self._items
83
84     def __repr__(self):
85         """Return a string representation of the stack."""
86         return f"DequeStack({list(self._items)})"
87 if __name__ == "__main__":
88     print("Testing Stack with list:")
89     s = Stack()
90     s.push(1)
91     s.push(2)
92     s.push(3)
93     print("Stack after pushes:", s)
94     print("Peek:", s.peek())
95     print("Pop:", s.pop())
96     print("Stack after pop:", s)
97     print("Is empty?", s.is_empty())
98     s.pop()
99     s.pop()
100    print("Is empty after popping all?", s.is_empty())
101
102   print("\nTesting DequeStack:")
103   ds = DequeStack()
104   ds.push('a')
105   ds.push('b')
106   ds.push('c')
107   print("DequeStack after pushes:", ds)
108   print("Peek:", ds.peek())
109   print("Pop:", ds.pop())
110   print("DequeStack after pop:", ds)
111   print("Is empty?", ds.is_empty())
112   ds.pop()
113   ds.pop()
114   print("Is empty after popping all?", ds.is_empty())

```

OUTPUT:

```

PS C:\Users\HASINI\OneDrive\Desktop\ai code> & "C:/Users/HASINI/OneDrive/Desktop/ai code/venv/Scripts\python.exe" stack.py
Testing Stack with list:
Stack after pushes: Stack([1, 2, 3])
Peek: 3
Pop: 3
Stack after pop: Stack([1, 2])
Is empty? False
Is empty after popping all? True

Testing DequeStack:
DequeStack after pushes: DequeStack(['a', 'b', 'c'])
Peek: c
Pop: c
DequeStack after pop: DequeStack(['a', 'b'])
Is empty? False
Is empty after popping all? True
PS C:\Users\HASINI\OneDrive\Desktop\ai code>

```

Observations:

Task 2: Queue Implementation with Performance Review

- **Task:** Implement a **Queue** with enqueue(), dequeue(), and is_empty() methods.

CODE:

```
1.py > ...
  from collections import deque

  class Queue:
      """A simple Queue (FIFO) implementation using a Python list."""

      def __init__(self):
          """Initialize an empty queue."""
          self._items = []

      def enqueue(self, item):
          """Add an item to the end of the queue.

          Args:
              item: The item to be added.
          """
          self._items.append(item)

      def dequeue(self):
          """Remove and return the item from the front of the queue.

          Returns:
              The item at the front of the queue.
          """
          Raises:
              IndexError: If the queue is empty.
          """
          if self.is_empty():

36          """
37          return len(self._items) == 0
38
39      def __repr__(self):
40          """Return a string representation of the queue."""
41          return f"Queue({self._items})"
42
43
44  class DequeQueue:
45      """Optimized Queue implementation using collections.deque."""
46
47      def __init__(self):
48          """Initialize an empty deque-based queue."""
49          self._items = deque()
50
51      def enqueue(self, item):
52          """Add an item to the end of the queue."""
53          self._items.append(item)
54
55      def dequeue(self):
56          """Remove and return the item from the front of the queue."""
57          if self.is_empty():
58              raise IndexError("dequeue from empty queue")
59          return self._items.popleft() # O(1) operation
```

```

44     class DequeQueue:
45
46         def dequeue(self):
47             """Remove and return the item from the front of the queue."""
48             if self.is_empty():
49                 raise IndexError("dequeue from empty queue")
50             return self._items.popleft() # O(1) operation
51
52         def is_empty(self):
53             """Check if the queue is empty."""
54             return not self._items
55
56         def __repr__(self):
57             """Return a string representation of the queue."""
58             return f"DequeQueue({list(self._items)})"
59
60
61     # 🧪 Test both implementations
62     if __name__ == "__main__":
63         print("Testing Queue with list:")
64         q = Queue()
65         q.enqueue(1)
66         q.enqueue(2)
67         q.enqueue(3)
68         print("Queue after enqueues:", q)
69         print("Dequeue:", q.dequeue())
70         print("Queue after dequeue:", q)
71         print("Is empty?", q.is_empty())
72         q.dequeue()
73         q.dequeue()
74         print("Is empty after all dequeues?", q.is_empty())
75
76
77     print("\nTesting DequeQueue:")
78     dq = DequeQueue()
79     dq.enqueue('a')
80     dq.enqueue('b')
81     dq.enqueue('c')
82     print("DequeQueue after enqueues:", dq)
83     print("Dequeue:", dq.dequeue())
84     print("DequeQueue after dequeue:", dq)
85     print("Is empty?", dq.is_empty())
86     dq.dequeue()
87     dq.dequeue()
88     print("Is empty after all dequeues?", dq.is_empty())

```

OUTPUT:

```

PS C:\Users\HASINI\OneDrive\Desktop\ai code> & "C:/Users/HASINI/OneDrive/Desktop/ai code/venv/Scripts/python.exe" "c:/U
Testing Queue with list:
Queue after enqueues: Queue([1, 2, 3])
Dequeue: 1
Queue after dequeue: Queue([2, 3])
Is empty? False
Is empty after all dequeues? True

Testing DequeQueue:
DequeQueue after enqueues: DequeQueue(['a', 'b', 'c'])
Dequeue: a
DequeQueue after dequeue: DequeQueue(['b', 'c'])
Is empty? False
Is empty after all dequeues? True
Testing Queue with list:
Queue after enqueues: Queue([1, 2, 3])
Dequeue: 1
Queue after dequeue: Queue([2, 3])
Is empty? False
Is empty after all dequeues? True

```

Observations:

Task 3: Singly Linked List with Traversal

- **Task:** Implement a **Singly Linked List** with operations: `insert_at_end()`, `delete_value()`, and `traverse()`.

CODE:

```
__pycache__
├── app.py
├── flask.py
└── venv
    ├── Include
    └── Lib
        ├── Scripts
        └── .gitignore
    └── pyenv.cfg
        ├── 11.py
        ├── 16.3.py
        ├── 17t1.py
        ├── 18.py
        ├── 19.2.java
        ├── app.py
        ├── portfolio.html
        └── restaurent.html
    └── UTLINE
    └── Node
```

```
1  class Node:
2      """A node in a singly linked list."""
3
4      def __init__(self, data):
5          """Initialize a node with data and next pointer.
6
7          Args:
8              data: The value to store in the node.
9
10         self.data = data
11         self.next = None
12
13
14 class SinglyLinkedList:
15     """A singly linked list with basic operations."""
16
17     def __init__(self):
18         """Initialize an empty linked list."""
19         self.head = None
20
21     def insert_at_end(self, data):
22         """Insert a new node with the given data at the end of the list.
23
24         Args:
25             data: The value to insert.
26
27         new_node = Node(data)
28         if not self.head:
29             self.head = new_node
30             return
31
32         current = self.head
33         while current.next:
34             current = current.next
```

```

36
37     def delete_value(self, value):
38         """Delete the first node with the specified value.
39
40         Args:
41             value: The value to delete.
42
43         Raises:
44             ValueError: If the value is not found in the list.
45
46         current = self.head
47         prev = None
48
49         while current:
50             if current.data == value:
51                 if prev:
52                     prev.next = current.next
53                 else:
54                     self.head = current.next
55                 return
56             prev = current
57             current = current.next
58
59         raise ValueError(f"Value {value} not found in the list.")
60
61     def traverse(self):
62         """Traverse the list and return a list of node values.
63
64         Returns:
65             List of node data values.
66
67         result = []
68         current = self.head
69         while current:
70             result.append(current.data)
71             current = current.next
72         return result
73
74     def __repr__(self):
75         """Return a string representation of the list."""
76
77
78     # Sample Test Cases
79     if __name__ == "__main__":
80         ll = SinglyLinkedList()
81         print("Initial list:", ll)
82
83         ll.insert_at_end(10)
84         ll.insert_at_end(20)
85         ll.insert_at_end(30)
86         print("After inserting 10, 20, 30:", ll)
87
88         ll.delete_value(20)
89         print("After deleting 20:", ll)
90
91     try:
92         ll.delete_value(99)
93     except ValueError as e:
94         print("Delete error:", e)
95
96     print("Traverse result:", ll.traverse())

```

OUTPUT:

```
PS C:\Users\HASINI\OneDrive\Desktop\ai code> & "C:/Users/HASINI/OneDrive/Desktop/ai code/venv/Scripts/py
Initial list: Empty List
After inserting 10, 20, 30: 10->20->30
After deleting 20: 10->30
Delete error: Value 99 not found in the list.
Initial list: Empty List
After inserting 10, 20, 30: 10->20->30
After deleting 20: 10->30
Delete error: Value 99 not found in the list.
After inserting 10, 20, 30: 10->20->30
After deleting 20: 10->30
Delete error: Value 99 not found in the list.
After deleting 20: 10->30
Delete error: Value 99 not found in the list.
Delete error: Value 99 not found in the list.
Traverse result: [10, 30]
PS C:\Users\HASINI\OneDrive\Desktop\ai code>
```

Observations:

Task 4: Binary Search Tree (BST)

- Task: Implement a **Binary Search Tree** with methods for `insert()`, `search()`, and `inorder traversal()`.

CODE:

```
thon-313.pyd
yton-313.pyc
re
cfg
.html
nhtml
ode
SearchTree

1      A node in the binary search tree.
2
3
4      def __init__(self, value):
5          """Initialize a BST node.
6
7          Args:
8              value: The value to store in the node.
9          """
10         self.value = value
11         self.left = None
12         self.right = None
13
14
15     class BinarySearchTree:
16         """Binary Search Tree with insert, search, and inorder traversal."""
17
18         def __init__(self):
19             """Initialize an empty BST."""
20             self.root = None
21
22         def insert(self, value):
23             """Insert a value into the BST.
24
25             Args:
26                 value: The value to insert.
27             """
28             self.root = self._insert_recursive(self.root, value)
29
30         def _insert_recursive(self, node, value):
31             """Helper method to insert recursively."""
32             if node is None:
33                 return BSTNode(value)
34             if value < node.value:
35                 node.left = self._insert_recursive(node.left, value)
36             elif value > node.value:
37                 node.right = self._insert_recursive(node.right, value)
38             # Duplicate values are ignored
```

```

File: 11.py
> Scripts 43
> .gitignore 44
> pyvenv.cfg 45
11.py 46
16.3.py 47
17t1.py 48
18.py 49
19.2.java 50
app.py 51
portfolio.html 52
restaurent.html 53
restaurent.html 54
restaurent.html 55
restaurent.html 56
restaurent.html 57
restaurent.html 58
restaurent.html 59
restaurent.html 60
restaurent.html 61
restaurent.html 62
restaurent.html 63
restaurent.html 64
restaurent.html 65
restaurent.html 66
restaurent.html 67
restaurent.html 68
restaurent.html 69
restaurent.html 70
restaurent.html 71
restaurent.html 72
restaurent.html 73
restaurent.html 74

    Args:
        value: The value to search for.

    Returns:
        bool: True if found, False otherwise.
    """
    return self._search_recursive(self.root, value)

def _search_recursive(self, node, value):
    """Helper method to search recursively."""
    if node is None:
        return False
    if value == node.value:
        return True
    if value < node.value:
        return self._search_recursive(node.left, value)
    else:
        return self._search_recursive(node.right, value)

def inorder_traversal(self):
    """Perform an inorder traversal of the BST.

    Returns:
        List of values in sorted order.
    """
    result = []
    self._inorder_recursive(self.root, result)
    return result

def _inorder_recursive(self, node, result):
    """Helper method for inorder traversal."""
    if node:
        self._inorder_recursive(node.left, result)
        result.append(node.value)
        self._inorder_recursive(node.right, result)

def __repr__(self):
    """Return a string representation of the BST (inorder)."""
    return "BST: " + " -> ".join(map(str, self.inorder_traversal())) or "Empty Tree"

# Sample Test Cases
if __name__ == "__main__":
    bst = BinarySearchTree()
    for val in [50, 30, 70, 20, 40, 60, 80]:
        bst.insert(val)

    print("BST after insertions:", bst)
    print("Inorder traversal:", bst.inorder_traversal())
    print("Search 40:", bst.search(40)) # True
    print("Search 25:", bst.search(25)) # False

```

OUTPUT:

```

PS C:\Users\HASINI\OneDrive\Desktop\ai code> & C:/Users/HASINI/OneDrive/Desktop/ai code/venv/Scripts/python.
PS C:\Users\HASINI\OneDrive\Desktop\ai code> & "C:/Users/HASINI/OneDrive/Desktop/ai code/venv/Scripts/python.
BST after insertions: BST: 20 -> 30 -> 40 -> 50 -> 60 -> 70 -> 80
Inorder traversal: [20, 30, 40, 50, 60, 70, 80]
Search 40: True
Search 25: False
PS C:\Users\HASINI\OneDrive\Desktop\ai code>

```

Observations:

Task 5: Graph Representation and BFS/DFS Traversal

- **Task:** Implement a **Graph** using an adjacency list, with traversal methods **BFS()** and **DFS()**.

CODE:

```

2
3     class Graph:
4         """Graph represented using an adjacency list."""
5
6     def __init__(self):
7         """Initialize an empty graph."""
8         self.adj_list = {}
9
10    def add_edge(self, src, dest):
11        """Add an edge from src to dest (undirected by default).
12
13        Args:
14            src: Source node.
15            dest: Destination node.
16        """
17        if src not in self.adj_list:
18            self.adj_list[src] = []
19        if dest not in self.adj_list:
20            self.adj_list[dest] = []
21        self.adj_list[src].append(dest)
22        self.adj_list[dest].append(src) # Remove this line for directed graph
23
24    def bfs(self, start):
25        """Perform Breadth-First Search (BFS) from the start node.
26
27        Args:
28            start: The starting node.
29
30        Returns:
31            List of nodes in BFS order.
32        """
33        visited = set()
34        queue = deque([start])
35        result = []
36
37        while queue:
38            node = queue.popleft()
39
40            for neighbor in self.adj_list[node]:
41                if neighbor not in visited:
42                    visited.add(neighbor)
43                    queue.append(neighbor)
44
45    def dfs(self, start):
46        """Perform Depth-First Search (DFS) from the start node.
47
48        Args:
49            start: The starting node.
50
51        Returns:
52            List of nodes in DFS order.
53        """
54        visited = set()
55        result = []
56
57        def dfs_recursive(node):
58            if node not in visited:
59                visited.add(node)
60                result.append(node)
61                for neighbor in self.adj_list.get(node, []):
62                    dfs_recursive(neighbor)
63
64        dfs_recursive(start)
65        return result
66
67
68    def __repr__(self):
69        """Return a string representation of the graph."""
70        return "\n".join(f"{node}: {neighbors}" for node, neighbors in self.adj_list.items())
71
72
73 # Sample Test Cases
74 if __name__ == "__main__":
75     g = Graph()
76     edges = [
77         ('A', 'B'), ('A', 'C'), ('B', 'D'),
78         ('C', 'E'), ('D', 'E'), ('E', 'F')
79     ]
80     for src, dest in edges:
81         g.add_edge(src, dest)
82
83     print("Graph adjacency list:")
84     print(g)
85
86     print("\nBFS from A:", g.bfs('A'))
87     print("DFS from A:", g.dfs('A'))

```

OUTPUT:

```

Graph adjacency list:
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'E']
D: ['B', 'E']
Graph adjacency list:
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'E']
D: ['B', 'E']
A: ['B', 'C']
B: ['A', 'D']
C: ['A', 'E']
D: ['B', 'E']
C: ['A', 'E']
D: ['B', 'E']
E: ['C', 'D', 'F']
F: ['E']

BFS from A: ['A', 'B', 'C', 'D', 'E', 'F']
DFS from A: ['A', 'B', 'D', 'E', 'C', 'F']

```

Observation:

- Uses dict for adjacency list — efficient and readable
- BFS uses deque for O(1) pops
- DFS uses recursion — elegant for small graphs
- For large graphs, consider iterative DFS to avoid recursion depth issues