ASSIGNMENT - 11

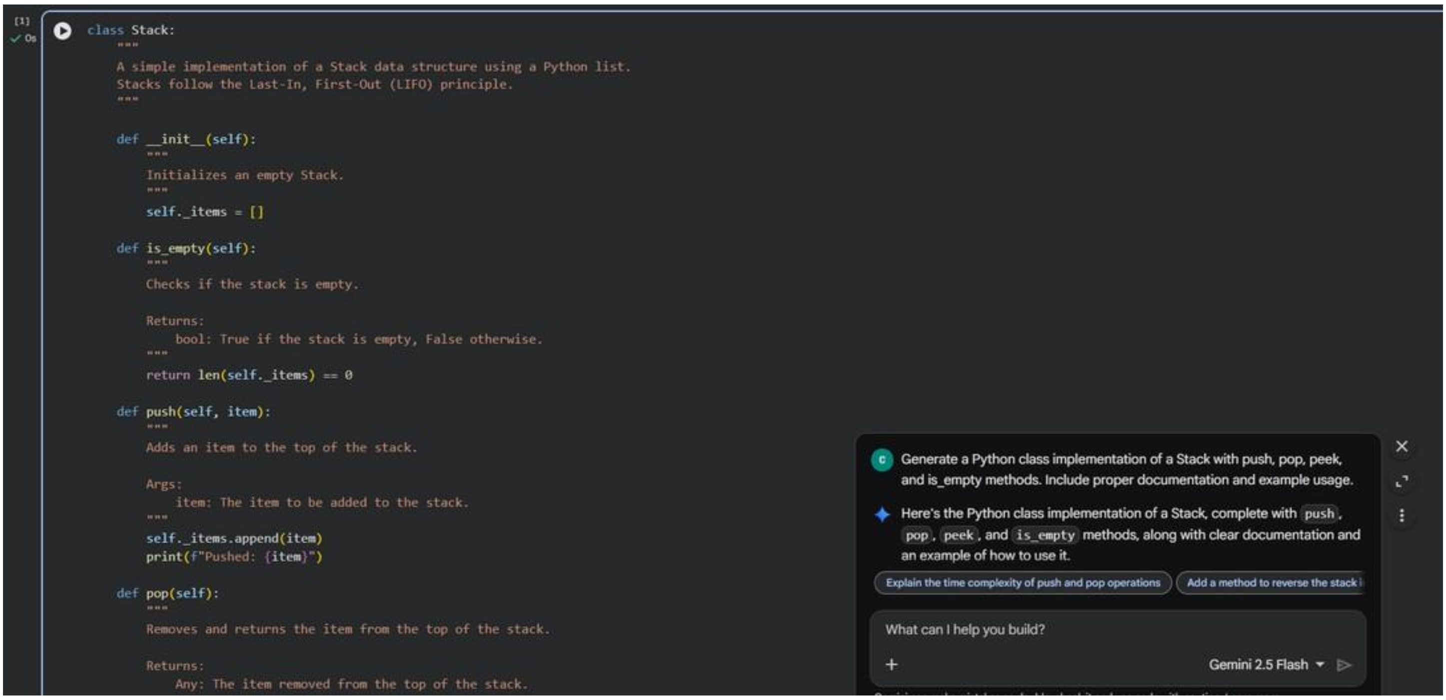
Thumma Hasini | 2303A52076 | Batch - 37

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.

Prompt Used: Generate a Python class implementation of a Stack with push, pop, peek, and is empty methods. Include proper documentation and example usage.



[1]
✓ 0s

```python
        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        popped_item = self._items.pop()
        print(f"Popped: {popped_item}")
        return popped_item

    def peek(self):
        """
        Returns the item at the top of the stack without removing it.

        Returns:
            Any: The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]

    def size(self):
        """
        Returns the number of items in the stack.

        Returns:
            int: The number of items in the stack.
        """
        return len(self._items)

    def __str__(self):
        """
        Returns a string representation of the stack.
        """
        return f"Stack: {self._items}"
```

The Stack class implementation was successfully executed. The output demonstrates the core functionalities: `push`, `pop`, `peek`, `is_empty`, and `size`. It also correctly shows the error handling for attempting to `pop` or `peek` from an empty stack.

Explain the time complexity of push and pop operations | Add a method to reverse the stack i

What can I help you build?

\+ Gemini 2.5 Flash

[1]
✓ 0s

```python
# --- Example Usage ---
print("\n--- Stack Example Usage ---")
my_stack = Stack()

print(f"Is stack empty? {my_stack.is_empty()}")
print(f"Stack size: {my_stack.size()}")
print(my_stack)

my_stack.push(10)
my_stack.push("hello")
my_stack.push([1, 2, 3])

print(f"Is stack empty? {my_stack.is_empty()}")
print(f"Stack size: {my_stack.size()}")
print(my_stack)

print(f"Top element (peek): {my_stack.peek()}")
print(my_stack)

my_stack.pop()
print(f"Stack size: {my_stack.size()}")
print(my_stack)

my_stack.pop()
print(f"Stack size: {my_stack.size()}")
print(my_stack)

my_stack.pop()
print(f"Stack size: {my_stack.size()}")
print(my_stack)

print(f"Is stack empty? {my_stack.is_empty()}")

# Demonstrating error handling for pop/peek on an empty stack
print("\nAttempting to pop from an empty stack:")
try:
    my_stack.pop()
except IndexError as e:
    print(e)
```

The Stack class implementation was successfully executed. The output demonstrates the core functionalities: `push`, `pop`, `peek`, `is_empty`, and `size`. It also correctly shows the error handling for attempting to `pop` or `peek` from an empty stack.

Explain the time complexity of push and pop operations | Add a method to reverse the stack i

What can I help you build?

\+ Gemini 2.5 Flash

```python
print("Attempting to peek from an empty stack:")
try:
    my_stack.peek()
except IndexError as e:
    print(e)
```

```
--- Stack Example Usage ---
Is stack empty? True
Stack size: 0
Stack: []
Pushed: 10
Pushed: hello
Pushed: [1, 2, 3]
Is stack empty? False
Stack size: 3
Stack: [10, 'hello', [1, 2, 3]]
Top element (peek): [1, 2, 3]
Stack: [10, 'hello', [1, 2, 3]]
Popped: [1, 2, 3]
Stack size: 2
Stack: [10, 'hello']
Popped: hello
Stack size: 1
Stack: [10]
Popped: 10
Stack size: 0
Stack: []
Is stack empty? True

Attempting to pop from an empty stack:
pop from empty stack
Attempting to peek from an empty stack:
peek from empty stack
```

Explanation: Stack follows LIFO principle. AI helped in structuring the class properly with documentation and edge-case handling.

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

Prompt Used: Create a Python Queue class implementing FIFO principle with enqueue, dequeue, front, and size methods.

```python
class Queue:
    """
    A simple implementation of a Queue data structure using a Python list.
    Queues follow the First-In, First-Out (FIFO) principle.
    """

    def __init__(self):
        """
        Initializes an empty Queue.
        """
        self._items = []

    def is_empty(self):
        """
        Checks if the queue is empty.

        Returns:
            bool: True if the queue is empty, False otherwise.
        """
        return len(self._items) == 0

    def enqueue(self, item):
        """
        Adds an item to the rear of the queue.

        Args:
            item: The item to be added to the queue.
        """
        self._items.append(item)
        print(f"Enqueued: {item}")
```

Create a Python Queue class implementing FIFO principle with enqueue, dequeue, front, and size methods.

Certainly! Here's the Python class implementation for a Queue, following the First-In, First-Out (FIFO) principle. It includes `enqueue`, `dequeue`, `front`, and `size` methods, along with comprehensive documentation and an example of its usage.

The Queue class implementation was successfully executed. The output clearly shows the First-In, First-Out (FIFO) behavior of the queue, demonstrating `enqueue`, `dequeue`, `front`, `is_empty`, and `size` methods. It also correctly handles attempts to `dequeue` or `front` from an empty queue by raising an `IndexError`, which is caught and printed.

Compare the performance of Stack and Queue implementations

Implement a linked list version of the Queue class

Write a function to demonstrate using both Stack and Queue

What can I help you build?

Gemini 2.5 Flash

```python
        Returns:
            Any: The item removed from the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        dequeued_item = self._items.pop(0) # Remove from the beginning (front)
        print(f"Dequeued: {dequeued_item}")
        return dequeued_item

    def front(self):
        """
        Returns the item at the front of the queue without removing it.

        Returns:
            Any: The item at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("front from empty queue")
        return self._items[0]

    def size(self):
        """
        Returns the number of items in the queue.

        Returns:
            int: The number of items in the queue.
        """
        return len(self._items)
```

[2]
✓ 0s

```python
    def __str__(self):
        """
        Returns a string representation of the queue.
        """
        return f"Queue: {self._items}"


# --- Example Usage ---
print("\n--- Queue Example Usage ---")
my_queue = Queue()

print(f"Is queue empty? {my_queue.is_empty()}")
print(f"Queue size: {my_queue.size()}")
print(my_queue)

my_queue.enqueue(10)
my_queue.enqueue("hello")
my_queue.enqueue([1, 2, 3])

print(f"Is queue empty? {my_queue.is_empty()}")
print(f"Queue size: {my_queue.size()}")
print(my_queue)

print(f"Front element: {my_queue.front()}")
print(my_queue)

my_queue.dequeue()
print(f"Queue size: {my_queue.size()}")
print(my_queue)

my_queue.dequeue()
print(f"Queue size: {my_queue.size()}")
print(my_queue)
```

```python
    my_queue.dequeue()
    print(f"Queue size: {my_queue.size()}")
    print(my_queue)

    print(f"Is queue empty? {my_queue.is_empty()}")

    # Demonstrating error handling for dequeue/front on an empty queue
    print("\nAttempting to dequeue from an empty queue:")
    try:
        my_queue.dequeue()
    except IndexError as e:
        print(e)

    print("Attempting to get front from an empty queue:")
    try:
        my_queue.front()
    except IndexError as e:
        print(e)
```

```
--- Queue Example Usage ---
Is queue empty? True
Queue size: 0
Queue: []
Enqueued: 10
Enqueued: hello
Enqueued: [1, 2, 3]
Is queue empty? False
Queue size: 3
Queue: [10, 'hello', [1, 2, 3]]
Front element: 10
Queue: [10, 'hello', [1, 2, 3]]
Dequeued: 10
Queue size: 2
Queue: ['hello', [1, 2, 3]]
Dequeued: hello
Queue size: 1
Queue: [[1, 2, 3]]
Dequeued: [1, 2, 3]
Queue size: 0
Queue: []
Is queue empty? True

Attempting to dequeue from an empty queue:
dequeue from empty queue
Attempting to get front from an empty queue:
front from empty queue
```

Explanation: Queue works on FIFO principle. AI improved method clarity and documentation.

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.

Prompt Used: Generate Python code for a singly linked list with node creation, insertion at end, and traversal display.

```python
class Node:
    """
    Represents a node in a singly linked list.
    Each node contains data and a reference to the next node.
    """
    def __init__(self, data):
        self.data = data
        self.next = None  # Initialize next as null

class SinglyLinkedList:
    """
    A simple implementation of a singly linked list.
    """
    def __init__(self):
        """
        Initializes an empty singly linked list.
        """
        self.head = None  # The head points to the first node in the list

    def insert_at_end(self, data):
        """
        Inserts a new node with the given data at the end of the linked list.

        Args:
            data: The data to be stored in the new node.
        """
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            print(f"Inserted '{data}' at the beginning (list was empty).")
            return

        current = self.head
        while current.next:
            current = current.next
```

Generate Python code for a singly linked list with node creation, insertion at end, and traversal display.

Here's the Python code for a singly linked list implementation, covering node creation, insertion at the end, and displaying the list's contents.

The Singly Linked List implementation was successfully executed. The

Add 'insert_at_beginning' and 'delete' methods to SinglyLinkedList | Explain the time complex

What can I help you build?

Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. Learn more

```python
        current.next = new_node
        print(f"Inserted '{data}' at the end.")

    def display(self):
        """
        Traverses the linked list and prints the data of each node.
        """
        if self.head is None:
            print("Linked List is empty.")
            return

        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        print("Linked List: " + " -> ".join(elements))

# --- Example Usage ---
print("\n--- Singly Linked List Example Usage ---")
my_linked_list = SinglyLinkedList()

my_linked_list.display()

my_linked_list.insert_at_end(10)
my_linked_list.insert_at_end(20)
my_linked_list.insert_at_end("hello")
my_linked_list.insert_at_end(30)

my_linked_list.display()

my_linked_list.insert_at_end([1, 2, 3])
my_linked_list.display()
```

The Singly Linked List implementation was successfully executed. The output demonstrates the creation of an empty list, subsequent insertions of various data types at the end of the list, and proper traversal and display of the list's elements at different stages.

Add 'insert_at_beginning' and 'delete' methods to SinglyLinkedList | Explain the time complex

What can I help you build?

Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. Learn more

```
[3]
✓ 0s
    my_linked_list.display()

...
    --- Singly Linked List Example Usage ---
    Linked List is empty.
    Inserted '10' at the beginning (list was empty).
    Inserted '20' at the end.
    Inserted 'hello' at the end.
    Inserted '30' at the end.
    Linked List: 10 -> 20 -> hello -> 30
    Inserted '[1, 2, 3]' at the end.
    Linked List: 10 -> 20 -> hello -> 30 -> [1, 2, 3]
```

Explanation: Linked list uses nodes connected via pointers. AI helped simplify insertion logic.

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.

Prompt Used: Write a Python program for Binary Search Tree with insertion and inorder traversal

```
[4]
✓ 0s
class Node:
    """
    Represents a node in a Binary Search Tree.
    Each node contains a key, and references to its left and right children.
    """
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BinarySearchTree:
    """
    A simple implementation of a Binary Search Tree.
    Supports insertion and inorder traversal.
    """
    def __init__(self):
        """
        Initializes an empty Binary Search Tree.
        """
        self.root = None

    def insert(self, key):
        """
        Inserts a new node with the given key into the BST.

        Args:
            key: The value to be inserted into the tree.
        """
        self.root = self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        """
        Helper function for recursive insertion.
        """
```

C Write a Python program for Binary Search Tree with insertion and inorder traversal

Certainly! Here's a Python implementation of a Binary Search Tree (BST) with methods for inserting new nodes and performing an inorder traversal, along with documentation and example usage.

Implement a search function for the Binary Search Tree | Add a deletion method to the Binary

What can I help you build?

+ Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. Learn more

[4]
✓ 0s

```python
        Helper function for recursive insertion.
        """
        if node is None:
            print(f"Inserted {key}")
            return Node(key)
        if key < node.key:
            node.left = self._insert_recursive(node.left, key)
        elif key > node.key:
            node.right = self._insert_recursive(node.right, key)
        else:
            # Key already exists, optionally handle duplicates
            print(f"Key {key} already exists, skipping insertion.")
        return node

    def inorder_traversal(self):
        """
        Performs an inorder traversal of the BST and prints the keys.
        Inorder traversal visits nodes in ascending order of their keys.
        """
        elements = []
        self._inorder_recursive(self.root, elements)
        if not elements:
            print("BST is empty.")
        else:
            print("Inorder Traversal: " + " -> ".join(map(str, elements)))

    def _inorder_recursive(self, node, elements):
        """
        Helper function for recursive inorder traversal.
        """
        if node:
            self._inorder_recursive(node.left, elements)
            elements.append(node.key)
            self._inorder_recursive(node.right, elements)
```

The Binary Search Tree implementation was successfully executed. The output clearly shows the insertion of elements, the handling of duplicate keys, and the correct inorder traversal of the tree, which prints elements in ascending order. It also demonstrates how an empty BST is handled during traversal.

Implement a search function for the Binary Search Tree | Add a deletion method to the Binary

What can I help you build?

+ Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. Learn more

[4]
✓ 0s

```python
        """
        if node:
            self._inorder_recursive(node.left, elements)
            elements.append(node.key)
            self._inorder_recursive(node.right, elements)

# --- Example Usage ---
print("\n--- Binary Search Tree Example Usage ---")
bst = BinarySearchTree()

print("Inserting elements:")
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)
bst.insert(60)
bst.insert(80)
bst.insert(30) # Attempt to insert a duplicate

print("\nPerforming inorder traversal:")
bst.inorder_traversal()

print("\nInserting more elements:")
bst.insert(10)
bst.insert(55)

print("\nPerforming inorder traversal again:")
bst.inorder_traversal()

bst_empty = BinarySearchTree()
print("\nTesting inorder traversal on an empty BST:")
bst_empty.inorder_traversal()
```

The Binary Search Tree implementation was successfully executed. The output clearly shows the insertion of elements, the handling of duplicate keys, and the correct inorder traversal of the tree, which prints elements in ascending order. It also demonstrates how an empty BST is handled during traversal.

Implement a search function for the Binary Search Tree | Add a deletion method to the Binar

What can I help you build?

+ Gemini 2.5 Flash

```
--- Binary Search Tree Example Usage ---
Inserting elements:
Inserted 50
Inserted 30
Inserted 70
Inserted 20
Inserted 40
Inserted 60
Inserted 80
Key 30 already exists, skipping insertion.

Performing inorder traversal:
Inorder Traversal: 20 -> 30 -> 40 -> 50 -> 60 -> 70 -> 80

Inserting more elements:
Inserted 10
Inserted 55

Performing inorder traversal again:
Inorder Traversal: 10 -> 20 -> 30 -> 40 -> 50 -> 55 -> 60 -> 70 -> 80

Testing inorder traversal on an empty BST:
BST is empty.
```

Explanation: Binary Search Tree maintains sorted order. Inorder traversal gives sorted output.

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

Prompt Used: Implement a hash table in Python using chaining for collision handling

```
[5]
✓ 0s
class HashTable:
    """
    A simple implementation of a Hash Table using chaining for collision handling.
    """

    def __init__(self, size=10):
        """
        Initializes the Hash Table with a given size.
        Each bucket in the table is a list to handle collisions (chaining).

        Args:
            size (int): The initial number of buckets in the hash table.
        """
        self.size = size
        self.table = [[] for _ in range(self.size)]
        print(f"Initialized Hash Table with {self.size} buckets.")

    def _hash(self, key):
        """
        Private method to compute the hash value for a given key.
        It sums the ASCII values of the characters in the key and
        takes the modulo of the table size.

        Args:
            key: The key to be hashed.

        Returns:
            int: The index in the hash table where the key should be stored.
        """
        return hash(key) % self.size
```

Implement a hash table in Python using chaining for collision handling

Okay, here's a Python implementation of a Hash Table that uses chaining to handle collisions, along with detailed explanations and examples of how to use it for insertion, retrieval, and deletion.

Explain the time complexity of the HashTable operations | Implement a function to rehash the

What can I help you build?

Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. Learn more

[5]
✓ 0s

```python
    def insert(self, key, value):
        """
        Inserts a key-value pair into the hash table.
        If the key already exists, its value will be updated.

        Args:
            key: The key to be inserted.
            value: The value associated with the key.
        """
        index = self._hash(key)
        bucket = self.table[index]

        # Check if key already exists in the bucket (for updating value)
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value) # Update value
                print(f"Updated key '{key}' at index {index}.")
                return

        # If key does not exist, add new key-value pair
        bucket.append((key, value))
        print(f"Inserted ('{key}', '{value}') at index {index}.")

    def get(self, key):
        """
        Retrieves the value associated with a given key.

        Args:
            key: The key to search for.
```

The Hash Table implementation was successfully executed. The output demonstrates the initialization of the hash table, insertion of key-value pairs (including handling of collisions and updates to existing keys), successful retrieval of values, and deletion of items. It also shows the behavior when attempting to retrieve or delete a key that does not exist. The print statements clearly illustrate the state of the buckets after various

Explain the time complexity of the HashTable operations | Implement a function to rehash the

What can I help you build?

+ Gemini 2.5 Flash

[5]
✓ 0s

```python
        Returns:
            Any: The value associated with the key, or None if the key is not found.
        """
        index = self._hash(key)
        bucket = self.table[index]

        for k, v in bucket:
            if k == key:
                print(f"Retrieved value '{v}' for key '{key}' from index {index}.")
                return v

        print(f"Key '{key}' not found in the hash table.")
        return None

    def delete(self, key):
        """
        Deletes a key-value pair from the hash table.

        Args:
            key: The key of the item to be deleted.

        Returns:
            bool: True if the item was deleted, False otherwise (key not found)
        """
        index = self._hash(key)
        bucket = self.table[index]

        for i, (k, v) in enumerate(bucket):
            if k == key:
```

retrieval of values, and deletion of items. It also shows the behavior when attempting to retrieve or delete a key that does not exist. The print statements clearly illustrate the state of the buckets after various operations, confirming the chaining mechanism.

Explain the time complexity of the HashTable operations | Implement a function to rehash t

What can I help you build?

+ Gemini 2.5 Flash

[5]
✓ 0s

```python
                del bucket[i]
                print(f"Deleted key '{key}' from index {index}.")
                return True

        print(f"Key '{key}' not found for deletion.")
        return False

    def __str__(self):
        """
        Returns a string representation of the hash table.
        """
        s = "-- Hash Table Contents --\n"
        for i, bucket in enumerate(self.table):
            s += f"Bucket {i}: {bucket}\n"
        return s


# --- Example Usage ---
print("\n--- Hash Table Example Usage ---")
hash_table = HashTable(size=5) # Small size to demonstrate collisions

# Insertion
hash_table.insert("apple", 10)
hash_table.insert("banana", 20)
hash_table.insert("cherry", 30)
hash_table.insert("date", 40)
hash_table.insert("elderberry", 50)

# Demonstrate collision (e.g., if 'grape' hashes to the same index as 'apple')
# Note: actual hash values depend on Python's hash() function and table size
```

[5]
✓ 0s

```python
hash_table.insert("grape", 60) # This might cause a collision with 'apple' or 'date'
hash_table.insert("fig", 70)
hash_table.insert("apple", 100) # Update existing key

print(hash_table)

# Retrieval
hash_table.get("cherry")
hash_table.get("apple")
hash_table.get("grape")
hash_table.get("pineapple") # Key not found

# Deletion
hash_table.delete("banana")
hash_table.delete("date")
hash_table.delete("orange") # Key not found for deletion

print(hash_table)

hash_table.get("banana") # Should not be found
hash_table.get("apple")  # Should still be found with updated value
```

...

```
--- Hash Table Example Usage ---
Initialized Hash Table with 5 buckets.
Inserted ('apple', '10') at index 2.
Inserted ('banana', '20') at index 4.
Inserted ('cherry', '30') at index 4.
Inserted ('date', '40') at index 0.
Inserted ('elderberry', '50') at index 0.
Inserted ('grape', '60') at index 4.
Inserted ('fig', '70') at index 3.
Updated key 'apple' at index 2.
```

```
-- Hash Table Contents --
Bucket 0: [('date', 40), ('elderberry', 50)]
Bucket 1: []
Bucket 2: [('apple', 100)]
Bucket 3: [('fig', 70)]
Bucket 4: [('banana', 20), ('cherry', 30), ('grape', 60)]

Retrieved value '30' for key 'cherry' from index 4.
Retrieved value '100' for key 'apple' from index 2.
Retrieved value '60' for key 'grape' from index 4.
Key 'pineapple' not found in the hash table.
Deleted key 'banana' from index 4.
Deleted key 'date' from index 0.
Key 'orange' not found for deletion.
-- Hash Table Contents --
Bucket 0: [('elderberry', 50)]
Bucket 1: []
Bucket 2: [('apple', 100)]
Bucket 3: [('fig', 70)]
Bucket 4: [('cherry', 30), ('grape', 60)]

Key 'banana' not found in the hash table.
Retrieved value '100' for key 'apple' from index 2.
100
```

Explanation: Hash table uses hashing for fast access. Chaining prevents collision issues.