Marking Rubric

| Criteria | Maximum Points |
| --- | --- |
| 1. **Hash Function** |  |
| 1.1 Correct computation of hash index | 4 points |
| 1.2 Handling short keys with given character | 3 points |
| 2 .**Insert Method** |  |
| 2.1 Correct insertion for new keys | 3 points |
| 2.1 Correct insertion for new keys | 4 points |
| 3. General Functionality & Code Structure | 1 points |

Solution

| Python | Java |
| --- | --- |
| class Node:  def \_\_init\_\_(self, key, value, next=None):  self.key = key  self.value = value  self.next = next  class HashTable:  def \_\_init\_\_(self, size):  self.size = size  self.table = [None] \* size  def hash\_function(self, key):  # Compute hash index based on the sum of ASCII values of the first three characters  ascii\_sum = 0  for i in range(3):  if i < len(key):  ascii\_sum += ord(key[i])  else:  ascii\_sum += ord('X') # Add 'X' if the key is shorter than three characters  return ascii\_sum % self.size  def insert(self, key, value):  index = self.hash\_function(key)  new\_node = Node(key, value)  if self.table[index] is None:  # No collision, directly insert  self.table[index] = new\_node  else:  # Collision resolution with linked list  current = self.table[index]  while current:  if current.key == key:  # Update the value if key already exists  current.value = value  return  if current.next is None:  # Add new node to the end of the list  current.next = new\_node  return  current = current.next    # Example usage  ht = HashTable(10)  # Insertions  ht.insert("PKG123", "In Transit")  ht.insert("AB", "Delivered")  ht.insert("PKG456", "Returned")  print("\nHash table after insertions:")  ht.display()  # Update existing key  ht.insert("PKG123", "Delivered")  print("\nHash table after updates:")  ht.display() | class Node {  String key;  String value;  Node next;  public Node(String key, String value) {  this.key = key;  this.value = value;  this.next = null;  }  }  class HashTable {  private Node[] table;  private int size;  public HashTable(int size) {  this.size = size;  this.table = new Node[size];  }  public int hashFunction(String key) {  // Compute hash index based on the sum of ASCII values of the first three characters  int asciiSum = 0;  for (int i = 0; i < 3; i++) {  if (i < key.length()) {  asciiSum += key.charAt(i);  } else {  asciiSum += 'X'; // Add 'X' if the key is shorter than three characters  }  }  return asciiSum % size;  }  public void insert(String key, String value) {  int index = hashFunction(key);  Node newNode = new Node(key, value);  if (table[index] == null) {  // No collision, directly insert  table[index] = newNode;  } else {  // Collision resolution with linked list  Node current = table[index];  while (current != null) {  if (current.key.equals(key)) {  // Update the value if key already exists  current.value = value;  return;  }  if (current.next == null) {  // Add new node to the end of the list  current.next = newNode;  return;  }  current = current.next;  }  }  }    public static void main(String[] args) {  // Example usage  HashTable ht = new HashTable(10);  // Insertions  ht.insert("PKG123", "In Transit");  ht.insert("AB", "Delivered");  ht.insert("PKG456", "Returned");  System.out.println("\nHash table after insertions:");  ht.display();  // Update existing key  ht.insert("PKG123", "Delivered");  System.out.println("\nHash table after updates:");  ht.display();  }  } |

SET-B

| Python | Java |
| --- | --- |
| class Node:  def \_\_init\_\_(self, key, value, next=None):  self.key = key  self.value = value  self.next = next  class HashTable:  def \_\_init\_\_(self, size):  self.size = size  self.table = [None] \* size  def hash\_function(self, key):  # Compute the hash index based on the sum of ASCII values of the first three characters  ascii\_sum = 0  for i in range(3):  if i < len(key):  ascii\_sum += ord(key[i])  else:  ascii\_sum += ord('0') # Add '0' if the key is shorter than three characters  return ascii\_sum % self.size  def insert(self, key, value):  index = self.hash\_function(key)  new\_node = Node(key, value)  if self.table[index] is None:  # No collision, directly insert  self.table[index] = new\_node  else:  # Collision resolution with linked list  current = self.table[index]  while current:  if current.key == key:  # Update the value by adding the new price to the existing one  current.value += value  return  if current.next is None:  # Add new node to the end of the list  current.next = new\_node  return  current = current.next  def display(self):  for i, node in enumerate(self.table):  if node is None:  continue # Skip empty indices  print(f"Index {i}:")  current = node  while current:  print(f" {current.key} ({current.value:.2f})")  current = current.next  # Example usage  ht = HashTable(10)  # Insertions  ht.insert("P123", 19.99)  ht.insert("AB", 15.50)  ht.insert("P456", 25.75)  print("\nHash table after insertions:")  ht.display()  # Update existing key  ht.insert("P123", 21.99)  print("\nHash table after updates:")  ht.display() | class Node {  String key;  double value;  Node next; // Pointer to the next node in case of a collision  // Constructor for the Node class  public Node(String key, double value, Node next) {  this.key = key;  this.value = value;  this.next = next;  }  }  class HashTable {  private Node[] table; // Array of Node references to store the data  private int size; // Size of the hash table  // Constructor to initialize the table with the given size  public HashTable(int size) {  this.size = size;  this.table = new Node[size];  }    public int hash\_function(String key) {  int sum = 0;  // Sum the ASCII values of the first three characters  for (int i = 0; i < 3; i++) {  if (i < key.length()) {  sum += key.charAt(i); } else {  sum += '0'; // If key length is less than 3, add ASCII value of '0'  }  }  return sum % size; // Return the index based on the table size  }    public void insert(String key, double value) {  int index = hash\_function(key); // Get the index using the hash function  Node newNode = new Node(key, value, null);  // If there's no node at the index, insert the new node  if (table[index] == null) {  table[index] = newNode;  } else {  // If collision occurs, traverse the linked list  Node current = table[index];  while (current != null) {  // If the key already exists, update the value by adding the new price  if (current.key.equals(key)) {  current.value += value;  return;  }  if (current.next == null) {  // If we've reached the end of the list, add the new node  current.next = newNode;  return;  }  current = current.next;  }  }  }  // Main method to test the hash table functionality  public static void main(String[] args) {  // Create a new hash table with size 10  HashTable ht = new HashTable(10);  // Insert some sample key-value pairs  ht.insert("P123", 19.99);  ht.insert("AB", 15.50);  ht.insert("P456", 25.75);  System.out.println("\nHash table after insertions:");  ht.display(); // Display the hash table after insertions  // Update the price for P123 by adding the new price  ht.insert("P123", 21.99);  System.out.println("\nHash table after update:");  ht.display(); // Display the hash table after update  }  } |