1. **Put 1367 and 984 into a hash table of size m=5 by using a hash function: h(k) = k mod m.**

1. Define the hash function:

*function hash(k):*

*return k mod 5*

2. Initialize the hash table:

*Let hashTable[0..4] be a new array of size 5 // Since m = 5, we have indices 0 to 4*

3. Insert the elements into the hash table:

For each element (k) in the input set (in this case, (k\_1 = 1367) and (k\_2 = 984)):

a. Compute the hash value (h(k)) for (k).

b. Insert (k) into the hash table at the index determined by (h(k)).

*index = hash(1367)*

*insert 1367 at hashTable[index]*

*index = hash(984)*

*insert 984 at hashTable[index]*

Example Calculation:

To better understand, let's calculate the hash for each key:

- For (k\_1 = 1367):

(h(1367) = 1367 mod 5 = 2)

- For (k\_2 = 984):

(h(984) = 984 mod 5 = 4)

So, 1367 should be inserted at index 2 of the hash table, and 984 should be inserted at index 4.

1. **Create a hash table with the following data and hash function, h(k) = k mod 7: 50, 700, 76, 85, 92, 73, 101 Collision resolution is solved by chaining**

1. Define the hash function:

*function hash(k):*

*return k mod 7*

2. Initialize the hash table:

*Let hashTable[0..6] be a new array of size 7*

*For i from 0 to 6:*

*hashTable[i] = new LinkedList() // Initialize each slot with a new LinkedList for chaining*

3. Pseudocode for the LinkedList Append Function

*function append(linkedList, value):*

*// Create a new node with the given value*

*newNode = new Node()*

*newNode.value = value*

*newNode.next = null*

*// If the linkedList is empty, then set the newNode as the head of the list*

*if linkedList is null:*

*linkedList = newNode*

*else:*

*// If the list is not empty, iterate to the end of the list*

*current = linkedList.head*

*while current.next is not null:*

*current = current.next*

*// Append the new node to the end of the list*

*current.next = newNode*

4. Insert the elements with chaining for collision resolution:

For each element (k) in the input list:

a. Compute the hash value (h(k)) for (k).

b. Append (k) to the list at `hashTable[h(k)]`.

*For each k in [50, 700, 76, 85, 92, 73, 101]:*

*index = hash(k)*

*append(hashTable[index], k) // Use the append function to add k to the linked list at hashTable[index]*

4. Handle collisions by chaining:

- If multiple items hash to the same index, they are stored in a list at that index. This way, each index of the hash table actually stores a list (or chain) of items that are hashed to the same index.

Example Insertion:

Let's calculate the hash for each key and perform the insertions:

- (h(50) = 50 mod 7 = 1)

- (h(700) = 700 mod 7 = 0)

- (h(76) = 76 mod 7 = 6)

- (h(85) = 85 mod 7 = 1)

- (h(92) = 92 mod 7 = 1)

- (h(73) = 73 mod 7 = 3)

- (h(101) = 101 mod 7 = 3)

Using these hash values, insert each item into the hash table at their respective indices and use chaining (a list) to handle collisions.

For example, for the keys that hash to 1 ((50, 85, 92)), you append each to the list at index 1. The resulting hash table structure would have lists of varying lengths at each index, depending on how many keys hash to the same value.

1. **Create a hash table with the following data and hash function, h(k) = k mod 9 with open indexing. example data: 19 blue, 27 red, 51 green, 13 blue, 37 brown, 31 pink, 25 orange**
   1. **open indexing**

1. Define the hash function:

*function hash(k):*

*return k mod 9*

2. Initialize the hash table:

*// Assume a larger array size for open addressing to reduce collision impact*

*Let hashTable[0..8] be a new array // Indices 0 to 8 for mod 9*

*// Initialize each slot of the hash table with a special value (e.g., null) indicating an empty slot.*

*For i from 0 to 8:*

*hashTable[i] = null*

3. Insert data into the hash table using linear probing for collision resolution:

For each key-value pair (k, v) in the data set:

a. Compute the initial hash index (i = hash(k)).

b. If `hashTable[i]` is empty (null), insert (k, v) there.

c. If `hashTable[i]` is occupied, probe sequentially (i.e., check `hashTable[(i + 1) mod tableSize]`, `hashTable[(i + 2) mod tableSize]`, ...) until an empty slot is found, then insert (k, v).

*data = [(19, "blue"), (27, "red"), (51, "green"), (13, "blue"), (37, "brown"), (31, "pink"), (25, "orange")]*

*For each (k, v) in data:*

*index = hash(k)*

*while hashTable[index] is not null:*

*index = (index + 1) mod 9 // Linear probing*

*hashTable[index] = (k, v) // Insert key-value pair*

Note on Linear Probing

- Linear probing is a simple and popular method for open addressing. However, it can lead to clustering, where consecutive slots get filled, increasing the chance of collisions and the time it takes to resolve them.

- Other forms of open addressing include quadratic probing and double hashing, each with its mechanisms to reduce clustering and improve the distribution of entries across the hash table.

Example Insertion Calculation

Let's consider the insertion process for a couple of the given data points using (h(k) = k mod 9):

- Insert (19, "blue"): (h(19) = 19 mod 9 = 1). Slot 1 is empty, so insert here.

- Insert (27, "red"): (h(27) = 27 mod 9 = 0). Slot 0 is empty, so insert here.

- Insert (51, "green"): (h(51) = 51 mod 9 = 6). Slot 6 is empty, so insert here.

- Insert (13, "blue"): (h(13) = 13 mod 9 = 4). Slot 4 is empty, so insert here.

...and so on for the rest of the data. If a collision occurs during this process (for example, if another key also had a hash result of 1), the algorithm would check the next slot (2, then 3, etc.) until an empty slot is found.

* 1. **Chaining**

1. Define the hash function:

*function hash(k):*

*return k mod 9*

2. Initialize the hash table:

*// Create an array of 9 elements to represent the hash table.*

*Let hashTable[0..8] be a new array // Indices 0 to 8 for mod 9*

*// Initialize each slot of the hash table with an empty list for chaining.*

*For i from 0 to 8:*

*hashTable[i] = new LinkedList()*

3. Function to append a value to a LinkedList:

*function append(linkedList, value):*

*newNode = new Node()*

*newNode.value = value*

*newNode.next = null*

*if linkedList.head is null:*

*linkedList.head = newNode*

*else:*

*current = linkedList.head*

*while current.next is not null:*

*current = current.next*

*current.next = newNode*

4. Insert keys into the hash table with chaining for collision resolution:

Assuming you have a set of keys you want to insert, you would iterate through each key, compute its hash value, and append it to the corresponding list in the hash table:

*// Example keys for illustration. Replace these with any set of keys you wish to insert.*

*keys = [your, keys, here] // Placeholder for actual keys*

*For each k in keys:*

*index = hash(k)*

*append(hashTable[index], k)*

1. **Create a hash table with the following data and hash function, h(k) = k mod 12, with all the 3 technique**

***A. Chaining***

Pseudocode for Creating a Hash Table with Chaining:

1. Define hash function and initialize hash table:

*function hash(k):*

*return k mod 12*

*Let hashTable[0..11] be a new array*

*For i from 0 to 11:*

*hashTable[i] = new LinkedList()*

2. Insert data:

*data = [(21, "blue"), (43, "red"), (55, "gray"), (8, "brown"), (71, "green"), (14, "blue"), (33, "yellow"), (27, "yellow")]*

*For each (k, v) in data:*

*index = hash(k)*

*hashTable[index].append((k, v))*

**B. Open Addressing (Linear Probing)**

Pseudocode for Creating a Hash Table with Open Addressing (Linear Probing):

1. Define hash function and initialize hash table:

*function hash(k, attempt):*

*return (k mod 12 + attempt) mod tableSize // tableSize is the size of the hashTable*

*Let hashTable[0..11] be a new array*

*For i from 0 to 11:*

*hashTable[i] = null*

2. Insert data:

*data = [(21, "blue"), (43, "red"), (55, "gray"), (8, "brown"), (71, "green"), (14, "blue"), (33, "yellow"), (27, "yellow")]*

*For each (k, v) in data:*

*attempt = 0*

*index = hash(k, attempt)*

*while hashTable[index] is not null:*

*attempt += 1*

*index = hash(k, attempt)*

*hashTable[index] = (k, v)*