Example Problem for Linear Probing in Hashing

Problem Statement

You are given a hash table of size **10** and the following sequence of integer keys to insert using **linear probing** (open addressing):

Keys to insert:
25, 36, 45, 20, 30, 12, 52, 44, 60, 85

The hash function to determine the initial position is:

h(k)=kmod 10h(k) = k \mod 10

If a collision occurs, resolve it using **linear probing** (i.e., move to the next available slot).

Step-by-Step Solution

Step 1: Compute Initial Hash Positions

Using the hash function:

h(k)=kmod 10h(k) = k \mod 10

Key	Hash kmod 10k \mod 10	Initial Position
25	25 mod 10 = 5	5
36	36 mod 10 = 6	6
45	45 mod 10 = 5	Collision at 5
20	20 mod 10 = 0	0
30	30 mod 10 = 0	Collision at 0
12	12 mod 10 = 2	2
52	52 mod 10 = 2	Collision at 2
44	44 mod 10 = 4	4
60	60 mod 10 = 0	Collision at 0
85	85 mod 10 = 5	Collision at 5

Step 2: Resolve Collisions Using Linear Probing

- **25 → Position 5** ✅ (No collision)
- **36 → Position 6** ✅ (No collision)

- **45 → Position 5 (Collision), check next → Position 7** ✅
- **20 → Position 0** ✅ (No collision)
- **30 → Position 0 (Collision), check next → Position 1** ✅
- **12 → Position 2** ✅ (No collision)
- **52 → Position 2 (Collision), check next → Position 3** ✅
- **44 → Position 4** ✅ (No collision)
- **60 → Position 0 (Collision), check next → Position 1 (Collision), check next → Position 8** ✅
- **85 → Position 5 (Collision), check next → Position 7 (Collision), check next → Position 9** ✅

Final Hash Table

Index	Key
0	20
1	30
2	12
3	52
4	44
5	25
6	36
7	45
8	60
9	85

Python Code for Linear Probing Implementation

Here's a Python implementation to insert values using **linear probing**:

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size
```

```
    def hash_function(self, key):
        return key % self.size

    def insert(self, key):
        index = self.hash_function(key)

        # Linear probing
        while self.table[index] is not None:
            index = (index + 1) % self.size  # Move to the next slot

        self.table[index] = key  # Place key in the found slot

    def display(self):
        for i in range(self.size):
            print(f"Index {i}: {self.table[i]}")

# Example Usage
keys = [25, 36, 45, 20, 30, 12, 52, 44, 60, 85]
hash_table = HashTable(10)

for key in keys:
    hash_table.insert(key)

hash_table.display()
```

Example Problem for Linear Probing in Hashing

Problem Statement

You have a **hash table of size 7** and need to insert the following keys using **linear probing** (open addressing):

Keys to insert:
19, 27, 36, 10, 64, 29, 42

The hash function to determine the initial position is:

h(k)=kmod 7h(k) = k \mod 7

If a collision occurs, resolve it using **linear probing** (move to the next available slot).

Step-by-Step Solution

Step 1: Compute Initial Hash Positions

Using the hash function:

h(k)=kmod 7h(k) = k \mod 7

Key	Hash kmod 7k \mod 7	Initial Position
19	19 mod 7 = 5	5
27	27 mod 7 = 6	6
36	36 mod 7 = 1	1
10	10 mod 7 = 3	3
64	64 mod 7 = 1	Collision at 1
29	29 mod 7 = 1	Collision at 1, next slot also taken
42	42 mod 7 = 0	0

Step 2: Resolve Collisions Using Linear Probing

- **19 → Position 5** ✅ (No collision)
- **27 → Position 6** ✅ (No collision)
- **36 → Position 1** ✅ (No collision)
- **10 → Position 3** ✅ (No collision)
- **64 → Position 1 (Collision), check next → Position 2** ✅
- **29 → Position 1 (Collision), check next → Position 2 (Collision), check next → Position 4** ✅
- **42 → Position 0** ✅ (No collision)

Final Hash Table

Index	**Key**
0	42
1	36
2	64
3	10
4	29
5	19
6	27

Python Code for Linear Probing Implementation

Here’s a Python implementation to insert values using **linear probing**:

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def hash_function(self, key):
        return key % self.size

    def insert(self, key):
        index = self.hash_function(key)

        # Linear probing to resolve collisions
        while self.table[index] is not None:
            index = (index + 1) % self.size  # Move to the next slot
```

```
        self.table[index] = key  # Place key in the found slot

    def display(self):
        for i in range(self.size):
            print(f"Index {i}: {self.table[i]}")

# Example Usage
keys = [19, 27, 36, 10, 64, 29, 42]
hash_table = HashTable(7)

for key in keys:
    hash_table.insert(key)

hash_table.display()
```

Example Problem for Linear Probing in Hashing

Problem Statement

You have a **hash table of size 9** and need to insert the following keys using **linear probing** (open addressing):

Keys to insert:
23, 45, 12, 67, 89, 19, 34, 56, 78

The hash function to determine the initial position is:

h(k)=kmod 9h(k) = k \mod 9

If a collision occurs, resolve it using **linear probing** (move to the next available slot).

Step-by-Step Solution

Step 1: Compute Initial Hash Positions

Using the hash function:

h(k)=kmod 9h(k) = k \mod 9

Key	Hash kmod 9k \mod 9	Initial Position
23	23 mod 9 = 5	5
45	45 mod 9 = 0	0
12	12 mod 9 = 3	3
67	67 mod 9 = 4	4
89	89 mod 9 = 8	8
19	19 mod 9 = 1	1
34	34 mod 9 = 7	7
56	56 mod 9 = 2	2
78	78 mod 9 = 6	6

Step 2: Insert Keys into the Hash Table

Since there are **no collisions**, all keys are placed in their computed positions.

Final Hash Table

Index	Key
0	45
1	19
2	56
3	12
4	67
5	23
6	78
7	34
8	89

Another Case: Handling Collisions

Let's change the dataset slightly to **introduce collisions**.

Keys to insert (with collision):
23, 45, 12, 67, 89, 19, 34, 56, 78, 91

- The new key **91** hashes to:

91mod 9=191 \mod 9 = 1

- **Collision at index 1 (19 is already there)** → Move to the next available slot **(index 2)**.
- **Collision at index 2 (56 is already there)** → Move to **index 3**.
- **Collision at index 3 (12 is already there)** → Move to **index 4**.
- **Collision at index 4 (67 is already there)** → Move to **index 5**.
- **Collision at index 5 (23 is already there)** → Move to **index 6**.
- **Collision at index 6 (78 is already there)** → Move to **index 7**.
- **Collision at index 7 (34 is already there)** → Move to **index 8**.
- **Collision at index 8 (89 is already there)** → Move to **index 0**.
- **Collision at index 0 (45 is already there)** → Move to **index 1 (again, full loop completed)** → The table is **full**, and we cannot insert 91.

Python Code for Linear Probing Implementation

Here's a Python implementation to insert values using **linear probing**:

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def hash_function(self, key):
        return key % self.size

    def insert(self, key):
```

```
        index = self.hash_function(key)
        start_index = index  # Save initial index to detect full loop

        # Linear probing to resolve collisions
        while self.table[index] is not None:
            index = (index + 1) % self.size  # Move to the next slot
            if index == start_index:  # Full cycle completed, table is full
                print(f"Table full! Could not insert {key}")
                return

        self.table[index] = key  # Place key in the found slot

    def display(self):
        for i in range(self.size):
            print(f"Index {i}: {self.table[i]}")

# Example Usage
keys = [23, 45, 12, 67, 89, 19, 34, 56, 78, 91]  # Includes collision case
hash_table = HashTable(9)

for key in keys:
    hash_table.insert(key)

hash_table.display()
```

Example Problem for Quadratic Probing in Hashing

Problem Statement

You have a **hash table of size 10** and need to insert the following keys using **quadratic probing**:

Keys to insert:
27, 43, 36, 17, 98, 19, 50, 29

The hash function to determine the initial position is:

h(k)=kmod 10h(k) = k \mod 10

If a collision occurs, resolve it using **quadratic probing**, which uses the probing function:

h′(k,i)=(h(k)+i2)mod 10h'(k, i) = (h(k) + i^2) \mod 10

where ii is the number of attempts to find an open slot.

Step-by-Step Solution

Step 1: Compute Initial Hash Positions

Using the hash function:

h(k)=kmod 10h(k) = k \mod 10

Key	Hash kmod 10k \mod 10	Initial Position
27	27 mod 10 = 7	7
43	43 mod 10 = 3	3
36	36 mod 10 = 6	6
17	17 mod 10 = 7	Collision at 7
98	98 mod 10 = 8	8
19	19 mod 10 = 9	9
50	50 mod 10 = 0	0
29	29 mod 10 = 9	Collision at 9

Step 2: Resolve Collisions Using Quadratic Probing

- **27 → Position 7** ✅ (No collision)
- **43 → Position 3** ✅ (No collision)
- **36 → Position 6** ✅ (No collision)
- **17 → Position 7 (Collision)**
  - Try i=1i = 1: (7+12)mod 10=8(7 + 1^2) \mod 10 = 8 (Occupied)

- Try i=2i = 2: (7+22)mod 10=1(7 + 2^2) \mod 10 = 1 ✅ **Place 17 at index 1**

- **98 → Position 8** ✅ (No collision)
- **19 → Position 9** ✅ (No collision)
- **50 → Position 0** ✅ (No collision)
- **29 → Position 9 (Collision)**
  - Try i=1i = 1: (9+12)mod 10=0(9 + 1^2) \mod 10 = 0 (Occupied)
  - Try i=2i = 2: (9+22)mod 10=3(9 + 2^2) \mod 10 = 3 (Occupied)
  - Try i=3i = 3: (9+32)mod 10=8(9 + 3^2) \mod 10 = 8 (Occupied)
  - Try i=4i = 4: (9+42)mod 10=5(9 + 4^2) \mod 10 = 5 ✅ **Place 29 at index 5**

Final Hash Table

Index	Key
0	50
1	17
2	None
3	43
4	None
5	29
6	36
7	27
8	98
9	19

Python Code for Quadratic Probing Implementation

Here's a Python implementation to insert values using **quadratic probing**:

```
class HashTable:
    def __init__(self, size):
        self.size = size
```

```python
        self.table = [None] * size

    def hash_function(self, key):
        return key % self.size

    def insert(self, key):
        index = self.hash_function(key)
        i = 0  # Quadratic probing counter

        while self.table[index] is not None:
            i += 1
            index = (self.hash_function(key) + i ** 2) % self.size

            if i == self.size:  # If all slots are checked, table is full
                print(f"Table full! Could not insert {key}")
                return

        self.table[index] = key  # Place key in the found slot

    def display(self):
        for i in range(self.size):
            print(f"Index {i}: {self.table[i]}")

# Example Usage
keys = [27, 43, 36, 17, 98, 19, 50, 29]
hash_table = HashTable(10)

for key in keys:
    hash_table.insert(key)
```

hash_table.display()

Example Problem for Quadratic Probing in Hashing

Problem Statement

You have a **hash table of size 11** and need to insert the following keys using **quadratic probing**:

Keys to insert:
20, 34, 45, 65, 12, 88, 75, 32

The hash function to determine the initial position is:

h(k)=kmod 11h(k) = k \mod 11

If a collision occurs, resolve it using **quadratic probing**, which uses the formula:

h′(k,i)=(h(k)+i2)mod 11h'(k, i) = (h(k) + i^2) \mod 11

where ii is the number of attempts to find an open slot.

Step-by-Step Solution

Step 1: Compute Initial Hash Positions

Using the hash function:

h(k)=kmod 11h(k) = k \mod 11

Key	Hash kmod 11k \mod 11	Initial Position
20	20 mod 11 = 9	9
34	34 mod 11 = 1	1
45	45 mod 11 = 1	Collision at 1
65	65 mod 11 = 10	10
12	12 mod 11 = 1	Collision at 1
88	88 mod 11 = 0	0
75	75 mod 11 = 9	Collision at 9
32	32 mod 11 = 10	Collision at 10

Step 2: Resolve Collisions Using Quadratic Probing

- **20 → Position 9** ✅ (No collision)
- **34 → Position 1** ✅ (No collision)
- **45 → Position 1 (Collision)**
  - Try i=1i = 1: (1+12)mod 11=2(1 + 1^2) \mod 11 = 2 ✅ **Place 45 at index 2**
- **65 → Position 10** ✅ (No collision)
- **12 → Position 1 (Collision)**
  - Try i=1i = 1: (1+12)mod 11=2(1 + 1^2) \mod 11 = 2 (Occupied)
  - Try i=2i = 2: (1+22)mod 11=5(1 + 2^2) \mod 11 = 5 ✅ **Place 12 at index 5**
- **88 → Position 0** ✅ (No collision)
- **75 → Position 9 (Collision)**
  - Try i=1i = 1: (9+12)mod 11=10(9 + 1^2) \mod 11 = 10 (Occupied)
  - Try i=2i = 2: (9+22)mod 11=2(9 + 2^2) \mod 11 = 2 (Occupied)
  - Try i=3i = 3: (9+32)mod 11=6(9 + 3^2) \mod 11 = 6 ✅ **Place 75 at index 6**
- **32 → Position 10 (Collision)**
  - Try i=1i = 1: (10+12)mod 11=0(10 + 1^2) \mod 11 = 0 (Occupied)
  - Try i=2i = 2: (10+22)mod 11=4(10 + 2^2) \mod 11 = 4 ✅ **Place 32 at index 4**

Final Hash Table

Index	**Key**
0	88
1	34
2	45
3	None
4	32
5	12
6	75

Index	Key
7	None
8	None
9	20
10	65

Python Code for Quadratic Probing Implementation

Here's a Python implementation to insert values using **quadratic probing**:

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def hash_function(self, key):
        return key % self.size

    def insert(self, key):
        index = self.hash_function(key)
        i = 0  # Quadratic probing counter

        while self.table[index] is not None:
            i += 1
            index = (self.hash_function(key) + i ** 2) % self.size

            if i == self.size:  # If all slots are checked, table is full
                print(f"Table full! Could not insert {key}")
                return
```

```python
        self.table[index] = key  # Place key in the found slot

    def display(self):
        for i in range(self.size):
            print(f"Index {i}: {self.table[i]}")

# Example Usage
keys = [20, 34, 45, 65, 12, 88, 75, 32]
hash_table = HashTable(11)

for key in keys:
    hash_table.insert(key)

hash_table.display()
```
