Shikha Omkar Singh
Roll no: 25

Assignment 4 DSA_2

Q1. What do you mean by hashing and hash functions in Data Structures?
Ans:

Hashing is a technique used in data structures to efficiently store and retrieve data. It involves transforming a given input (key) into a fixed-size numerical value called a **hash value** or **hash code**, which is used to index into an array (hash table) where the data is stored. This allows for fast access to the data based on its key.

Key terms:

1. **Hash Function**: A hash function is a function that takes an input (or key) and returns a fixed-size string of bytes. The output (hash value) is typically a number that corresponds to the location (index) in the hash table where the data associated with that key will be stored. A good hash function minimizes collisions, which occur when two keys hash to the same index.
2. **Hash Table**: A hash table is a data structure that stores key-value pairs. It uses the hash function to determine the index (or "bucket") where the value associated with a key will be stored.

Example:

If you want to store student records, and you use their student ID as the key, a hash function will convert the student ID into an index in the hash table where the student's record will be stored.

Benefits:

- **Fast Lookups**: Hashing provides constant-time complexity (O(1)) for lookups in the best case.
- **Efficient Space Usage**: A hash table can store large amounts of data with minimal space wastage.

Example of a simple hash function:

```
int hashFunction(int key) {
    return key % 10;  // Returns a value between 0 and 9
}
```

This hash function divides the key by 10 and uses the remainder as the index in the hash table.

Q2. What are the types of collision resolution techniques in Data
Structures?
Ans:

In hashing, a **collision** occurs when two different keys hash to the same index in a hash table. To resolve collisions, several techniques can be used. These techniques are called **collision resolution techniques**.

Types of Collision Resolution Techniques:

1. **Open Addressing**: In this method, when a collision occurs, the algorithm searches for the next available slot in the hash table to place the collided key. There are three common strategies for finding the next available slot:
    - **Linear Probing**: When a collision occurs, the algorithm checks the next slot (index + 1) in a sequential manner until an empty slot is found.
        - Formula: `new_index = (hash_index + 1) % table_size`

Example:
cpp
Copy code
```
int hashFunction(int key) {
    return key % 10;  // hash function
}

// If collision happens at index 5, try index 6, 7, etc.
```

-
- **Quadratic Probing**: When a collision occurs, it searches for an empty slot using a quadratic function (i.e., checking slots further away each time, like 1^2, 2^2, 3^2, etc.).
    - Formula: `new_index = (hash_index + i^2) % table_size`, where `i` is the number of attempts.
- **Double Hashing**: This technique uses a second hash function to calculate the next slot to check if a collision occurs.
    - Formula: `new_index = (hash_index + i * hash2(key)) % table_size`, where `hash2(key)` is another hash function.

2. **Chaining (Separate Chaining)**: In this method, each slot in the hash table holds a **linked list** or **chain** of all the keys that hash to that index. When a collision occurs, the new key is simply added to the linked list at that index.

    - Advantage: It handles collisions efficiently and avoids clustering.
    - Example: If two keys map to the same index, both are stored in a linked list at that index.

Example in C++ using a linked list for chaining:
```
struct Node {
    int key;
    Node* next;
};

// If collision happens, add to the linked list at the same index
```

3.
4. **Bucket Addressing**: This technique uses a **bucket** (an array or list) at each index, capable of holding multiple elements. When a collision occurs, all the elements are stored in the same bucket, similar to chaining but without linked lists.
5. **Rehashing**: When the hash table becomes too full and collisions become frequent, the hash table can be resized and rehashed. This means a larger hash table is created, and all elements are hashed again into the new table, reducing the chances of collisions.

Summary of Techniques:

- **Open Addressing**:
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
- **Chaining (Separate Chaining)**
- **Bucket Addressing**
- **Rehashing**

Each technique has its trade-offs, and the best one to use depends on the specific requirements of the application, such as speed, memory usage, and the likelihood of collisions.

Q3. A hash function f defined as f(key)=key mod 7, with linear probing
insert the keys
37, 38, 72, 48, 98, 11, 56
Into a table indexed from 0, in which location the key 11 will be stored
(count table index 0 as 0th location)?
Ans:

Let's go step by step and use the given hash function `f(key) = key % 7` with linear probing to insert the keys.

1. **Hash function**: `f(key) = key % 7`
2. **Table size**: 7 (because we're using modulo 7)

Inserting the keys:

1. **Key 37**:
   - Hash value: `37 % 7 = 2`
   - Insert 37 at index 2.
2. **Key 38**:
   - Hash value: `38 % 7 = 3`
   - Insert 38 at index 3.
3. **Key 72**:
   - Hash value: `72 % 7 = 2`
   - Index 2 is already occupied (collision with 37).
   - Use linear probing: Check next index (3), but it's also occupied by 38.
   - Move to index 4. Insert 72 at index 4.
4. **Key 48**:
   - Hash value: `48 % 7 = 6`
   - Insert 48 at index 6.
5. **Key 98**:
   - Hash value: `98 % 7 = 0`
   - Insert 98 at index 0.
6. **Key 11**:
   - Hash value: `11 % 7 = 4`
   - Index 4 is already occupied (collision with 72).
   - Use linear probing: Check the next index (5). It's empty.
   - Insert 11 at index 5.
7. **Key 56**:
   - Hash value: `56 % 7 = 0`
   - Index 0 is already occupied (collision with 98).
   - Use linear probing: Check the next index (1). It's empty.
   - Insert 56 at index 1.

Final Table:

Index	Value
0	98
1	56

2	37
3	38
4	72
5	11
6	48

Answer:

The key **11** will be stored at index **5**.

Q4. Explain the chaining method with an appropriate example?
Ans:

Chaining Method in Hashing

Chaining is a **collision resolution technique** used in hash tables where each index of the hash table stores a **linked list** (or any other data structure like an array or list) of all the keys that hash to the same index. When multiple keys hash to the same index (i.e., a collision occurs), they are added to the linked list at that index.

How Chaining Works:

1. The hash function maps a key to an index in the hash table.
2. If the index is empty, the key is inserted there.
3. If the index is already occupied (collision occurs), the new key is added to the linked list (chain) at that index.
4. Search, insert, and delete operations need to traverse the linked list at that index to locate or manipulate data.

Advantages of Chaining:

- **Efficient Collision Handling**: It handles collisions efficiently by allowing multiple keys to be stored at the same index.
- **Dynamic Size**: The linked list can grow dynamically as more elements are added.
- **Simple Implementation**: Chaining is relatively simple to implement and doesn't require resizing the hash table.

Example of Chaining:

Let's consider a hash function:

- `f(key) = key % 5` (modulus operator with table size 5)
- We'll insert the keys: **10, 15, 20, 25, 30, 12**

Step-by-step insertion:

1. **Key 10**:
   - Hash value: `10 % 5 = 0`
   - Insert 10 at index 0.
2. **Key 15**:
   - Hash value: `15 % 5 = 0`
   - Collision at index 0 (already occupied by 10).
   - Add 15 to the linked list at index 0.
3. **Key 20**:
   - Hash value: `20 % 5 = 0`
   - Collision at index 0 (already occupied by 10 and 15).
   - Add 20 to the linked list at index 0.
4. **Key 25**:
   - Hash value: `25 % 5 = 0`
   - Collision at index 0 (already occupied by 10, 15, and 20).
   - Add 25 to the linked list at index 0.
5. **Key 30**:
   - Hash value: `30 % 5 = 0`
   - Collision at index 0 (already occupied by 10, 15, 20, and 25).
   - Add 30 to the linked list at index 0.
6. **Key 12**:
   - Hash value: `12 % 5 = 2`
   - Insert 12 at index 2 (no collision here).

Final Hash Table with Chaining:

Index	**Linked List (Chain)**
0	10 → 15 → 20 → 25 → 30
1	Empty
2	12
3	Empty
4	Empty

- **Index 0** contains a chain (linked list) of elements: **10 → 15 → 20 → 25 → 30**.

- **Index 2** contains a single element: **12**.
- Other indices are empty.

Visual Representation of Chaining:

```
Index 0: 10 → 15 → 20 → 25 → 30
Index 1: (empty)
Index 2: 12
Index 3: (empty)
Index 4: (empty)
```

Search Example:

If we want to search for **20**, the hash function will return index 0 (`20 % 5 = 0`). At index 0, we traverse the linked list **10** → **15** → **20** and find the key **20**.

Conclusion:

The chaining method provides an efficient way to handle collisions, especially when the table size is small or the hash function leads to many collisions. It can store multiple keys at the same index without requiring extra space for probing.

Q5. What is the primary purpose of linear probing in hashing?
Ans:

The primary purpose of **linear probing** in hashing is to resolve **collisions** when two or more keys hash to the same index in a hash table. Linear probing finds the next available slot (or bucket) by sequentially checking the subsequent indices (in a linear manner) until an empty slot is found.

Key Features of Linear Probing:

1. **Collision Resolution**: When a collision occurs at a given index, linear probing searches for the next open index in a sequential manner, wrapping around the hash table if necessary.
2. **Simple Mechanism**: If index `i` is occupied, it checks index `i + 1`, then `i + 2`, and so on, until an empty slot is found. This approach is easy to implement and understand.
3. **Single Array Structure**: Unlike other methods (such as chaining), linear probing doesn't require additional data structures like linked lists. It only uses the hash table's array itself to handle collisions.

How Linear Probing Works:

- **Hash Function**: A key is hashed to an index using a hash function.
- **Collision**: If that index is already occupied, linear probing checks the next slot (index + 1).
- **Wrap Around**: If the end of the array is reached, the search continues from the beginning of the array (circular manner).

Example:

Assume the hash table size is 7, and the hash function is `f(key) = key % 7`. The keys to be inserted are **18, 41, 22, 44, 59**.

1. **Insert 18**:
   - Hash value: `18 % 7 = 4`
   - Insert 18 at index 4.
2. **Insert 41**:
   - Hash value: `41 % 7 = 6`
   - Insert 41 at index 6.
3. **Insert 22**:
   - Hash value: `22 % 7 = 1`
   - Insert 22 at index 1.
4. **Insert 44**:
   - Hash value: `44 % 7 = 2`
   - Insert 44 at index 2.
5. **Insert 59**:
   - Hash value: `59 % 7 = 3`
   - Insert 59 at index 3.

If a new key (like 81) results in a collision at index 2:

- Linear probing would check the next index (3), and continue until it finds an empty spot.

Conclusion:

Linear probing's primary purpose is to ensure that collisions in hash tables are handled efficiently by finding the next available slot using a simple and systematic approach. It is widely used because of its simplicity, but it can suffer from clustering, which is the accumulation of consecutive filled slots.

Q6. How does linear probing handle collision?
Ans:

Linear probing handles collisions in a hash table by systematically searching for the next available empty slot when the original hash index is already occupied by another key. It resolves the collision by following a **linear sequence** of probing (checking the next slots) until an empty slot is found.

Steps of Collision Resolution with Linear Probing:

1. **Hashing**: When a key is inserted, the hash function calculates its initial index (let's call this `hash_index`).
2. **Collision Detection**: If the slot at `hash_index` is already occupied by another key, a **collision** has occurred.
3. **Probing**: Linear probing then checks the **next slot** in the array (index `hash_index + 1`). If this slot is also occupied, it moves to the next slot (index `hash_index + 2`), and so on.
4. **Wrapping Around**: If the end of the table is reached, the search wraps around to the beginning of the table. This continues until an empty slot is found, where the key is inserted.

Example:

Let's take a hash function `f(key) = key % 7` and a hash table of size 7.

Insert the following keys using linear probing: **10, 20, 30, 15, 35**.

1. **Insert 10**:
   - Hash value: `10 % 7 = 3`
   - Insert 10 at index 3.
2. **Insert 20**:
   - Hash value: `20 % 7 = 6`
   - Insert 20 at index 6.
3. **Insert 30**:
   - Hash value: `30 % 7 = 2`
   - Insert 30 at index 2.
4. **Insert 15**:
   - Hash value: `15 % 7 = 1`
   - Insert 15 at index 1.
5. **Insert 35**:
   - Hash value: `35 % 7 = 0`
   - Insert 35 at index 0.

If we now try to insert **40**:

- **Hash value**: `40 % 7 = 5`

- **Index 5** is empty, so we insert **40** there.

Example with Collision:

If we want to insert **17**:

- **Hash value**: `17 % 7 = 3` → **collision** at index 3 (occupied by 10).
- **Probing**: Check index 4 (empty).
- Insert **17** at index 4.

Final Hash Table:

Index	Value
0	35
1	15
2	30
3	10
4	17
5	40
6	20

How Linear Probing Handles Collision:

- If two keys hash to the same index (collision), it linearly checks the next available slots.
- The search continues until an empty slot is found, and the key is inserted there.
- This approach ensures that even when collisions happen, all keys can still be inserted into the hash table.

Challenges with Linear Probing:

- **Primary Clustering**: As more collisions occur, consecutive slots in the table can get filled, forming clusters. This can increase the time it takes to find an empty slot during insertions or searches.
- **Wraparound**: The probing wraps around to the beginning of the array when the end of the hash table is reached.

Conclusion:

Linear probing handles collisions by checking the next consecutive slots in a hash table, using a simple linear process to find the next available empty spot. This method is easy to implement but can suffer from clustering issues in dense tables.

Q7. Explain quadratic probing method of hashing?
Ans:

Quadratic Probing in Hashing

Quadratic probing is a collision resolution technique used in open addressing within a hash table. Like **linear probing**, it finds the next available slot when a collision occurs, but instead of checking the next slot in a linear sequence, it uses a **quadratic function** to determine how far to move from the original hash index.

How Quadratic Probing Works:

When a collision occurs, instead of checking the next slot, it checks slots at intervals that increase quadratically. The probe sequence for an initial hash index `i` follows the pattern:

```
new_index = (hash_index + 1^2) % table_size
new_index = (hash_index + 2^2) % table_size
new_index = (hash_index + 3^2) % table_size
...
```

This means the distance from the original index grows as the square of the number of attempts (`i^2`). This reduces the clustering problem (large groups of consecutive filled slots) that occurs in **linear probing**.

Formula:

For a key `key` with initial hash index `hash_index` calculated by the hash function `f(key)`, the index for insertion using quadratic probing is calculated as:

```
new_index = (hash_index + i^2) % table_size
```

Where `i` starts at 1 and increases each time a collision occurs.

Example:

Let’s consider a hash function `f(key) = key % 7` and a hash table of size 7. We'll insert the keys **10, 20, 30, 40, 50** using **quadratic probing**.

1. **Insert 10**:
   - Hash value: `10 % 7 = 3`
   - Insert 10 at index 3.
2. **Insert 20**:
   - Hash value: `20 % 7 = 6`
   - Insert 20 at index 6.
3. **Insert 30**:
   - Hash value: `30 % 7 = 2`
   - Insert 30 at index 2.
4. **Insert 40**:
   - Hash value: `40 % 7 = 5`
   - Insert 40 at index 5.
5. **Insert 50**:
   - Hash value: `50 % 7 = 1`
   - Insert 50 at index 1.

Now, if we try to insert **60**:

- **Hash value**: `60 % 7 = 4`
- Index 4 is empty, so we insert **60** at index 4.

Example with Collision:

Let's now try to insert **70**:

- **Hash value**: `70 % 7 = 0`
- **Index 0** is empty, so we insert **70** at index 0.

Now, let's try inserting **80**:

- **Hash value**: `80 % 7 = 3` (collision at index 3, already occupied by 10).
- **Quadratic Probing**:
  - First probe: `(3 + 1^2) % 7 = 4` (collision at index 4, already occupied by 60).
  - Second probe: `(3 + 2^2) % 7 = 0` (collision at index 0, already occupied by 70).
  - Third probe: `(3 + 3^2) % 7 = 2` (collision at index 2, already occupied by 30).
  - Fourth probe: `(3 + 4^2) % 7 = 3` (collision again, but index is back to 3).

- **If the table is full or this probe sequence continues, rehashing would be required.**

Advantages of Quadratic Probing:

1. **Reduces Clustering**: Unlike linear probing, quadratic probing reduces the occurrence of **primary clustering** (when consecutive slots get filled and create clusters), making it more efficient in certain situations.
2. **Simple to Implement**: It's easy to implement since it follows a predictable sequence based on the square of the number of collisions.

Disadvantages of Quadratic Probing:

1. **Secondary Clustering**: While it reduces primary clustering, it still suffers from **secondary clustering**, where keys that hash to the same initial index follow the same probing sequence.
2. **Table Size Constraints**: Quadratic probing works best when the hash table size is a prime number. Otherwise, certain indices may not be probed at all, leaving some slots unreachable.
3. **Increased Probing Time**: As the number of collisions increases, the quadratic function can cause larger jumps in the table, which might take longer to find an available slot compared to linear probing.

Conclusion:

Quadratic probing is a collision resolution technique that uses a quadratic function to avoid clustering and handle collisions. It is more efficient than linear probing in some cases, especially when there are frequent collisions. However, it still has some limitations, such as secondary clustering and the need for careful table size selection to avoid unreachable slots.

Q8. What is the difference between linear probing and quadratic
probing methods?
Ans:

Differences Between Linear Probing and Quadratic Probing

Both **linear probing** and **quadratic probing** are open addressing techniques used to resolve collisions in a hash table. The key difference lies in how they probe (search for the next available slot) after a collision occurs.

Here's a detailed comparison between the two methods:

Aspect	Linear Probing	Quadratic Probing
Probing Function	Probes linearly: `new_index = (hash_index + i) % table_size`, where `i` is the probe number (1, 2, 3, ...)	Probes using a quadratic function: `new_index = (hash_index + i^2) % table_size`, where `i` is the probe number (1, 2, 3, ...)
Probing Sequence	Sequential, checking the next slot (index + 1, index + 2, etc.).	Grows quadratically (index + 1, index + 4, index + 9, etc.).
Clustering	Prone to **primary clustering**: Consecutive elements tend to fill up, causing long chains of filled slots.	Reduces primary clustering but can suffer from **secondary clustering**: Multiple keys hashing to the same index follow the same probing pattern.
Efficiency	Can become inefficient with frequent collisions due to clustering.	More efficient than linear probing in avoiding clusters, but probing gaps can become larger as collisions increase.
Collision Handling	Handles collisions by checking the very next slot until it finds an empty one.	Handles collisions by checking slots that are farther apart (based on the square of the attempt number).
Performance Impact	Performance degrades as the table gets fuller, especially when many collisions occur, due to primary clustering.	Better performance than linear probing in highly filled tables due to reduced clustering, but large gaps can affect insertion time.
Wrap Around	Wraps around to the beginning of the table when the end is reached.	Also wraps around to the beginning of the table when the end is reached.
Implementation Complexity	Simpler to implement as it just checks the next slot in sequence.	Slightly more complex to implement due to the quadratic probing formula.
Space Utilization	Can utilize every available slot in the hash table.	Depending on the table size, some slots may remain unreachable without rehashing (non-prime table sizes can cause issues).
Best Used For	Suitable for cases where the table is sparsely filled and collisions are rare.	Better suited for densely filled tables with frequent collisions.

Summary:

- **Linear Probing**: Simple and easy to implement, but suffers from **primary clustering** (long chains of consecutive filled slots), which can degrade performance as the table fills up.
- **Quadratic Probing**: Reduces primary clustering by spreading out the probe sequence, but it introduces **secondary clustering** and might leave some slots unreachable depending on the table size. It is more efficient when there are frequent collisions, but its gaps between probes increase as the number of collisions grows.

Both techniques have trade-offs, and the choice between them depends on the specific use case, hash table size, and expected collision frequency.

Q9. Apply Open Hashing Chaining method and insert the keys into the
hash table
Keys: 42,19,10,12
K mod 5
Table size-5
Ans:

To insert the keys **42, 19, 10, 12** into a hash table using **Open Hashing (Chaining)** with the hash function `K mod 5` and a table size of 5, we will follow these steps:

Steps:

1. **Hash function**: `f(key) = key % 5`
2. **Table size**: 5 (indices 0 to 4)
3. **Open Hashing (Chaining)**: In this method, each index in the hash table points to a linked list (or chain) where we can store multiple keys that hash to the same index.

Inserting the keys:

1. **Insert 42**:
   - Hash value: `42 % 5 = 2`
   - Insert 42 in index 2.
2. **Insert 19**:
   - Hash value: `19 % 5 = 4`
   - Insert 19 in index 4.
3. **Insert 10**:
   - Hash value: `10 % 5 = 0`
   - Insert 10 in index 0.
4. **Insert 12**:

    - Hash value: `12 % 5 = 2` (collision with 42)
    - Since index 2 is already occupied, we use chaining to add 12 to the linked list at index 2.

Final Hash Table with Chaining:

Index	Chain (Linked List)
0	10
1	(empty)
2	42 → 12
3	(empty)
4	19

Explanation:

- **Index 0** stores **10** because `10 % 5 = 0`.
- **Index 2** stores a chain (linked list) with **42** and **12**, since both keys hashed to index 2 (`42 % 5 = 2` and `12 % 5 = 2`).
- **Index 4** stores **19** because `19 % 5 = 4`.
- Indices 1 and 3 remain empty because no keys hashed to those locations.

In **Open Hashing (Chaining)**, collisions are resolved by creating a linked list at each index, allowing multiple keys to be stored at the same hash index. This way, even if multiple keys hash to the same value, they are stored in a list at that index.

Q10. Insert keys at the first free location from (u+(i*i))%m where i=0 to
(m-1)
A=3,2,9,6,11,13,7,12
h(k)=2k+3
m=10
Use Division Method and Quadratic Probing to store these values?

Ans:

To insert the keys **3, 2, 9, 6, 11, 13, 7, 12** using the formula `(u + (i*i)) % m` with **quadratic probing** and the **division method**, follow these steps:

Given:

- **Keys**: 3, 2, 9, 6, 11, 13, 7, 12
- **Hash function**: `h(k) = 2k + 3`
- **Table size**: `m = 10`
- **Quadratic probing**: We will check the quadratic probing sequence: `(hash_value + i²) % 10` where `i` is the attempt number (starting from 0).

Steps to Insert Keys Using Quadratic Probing:

1. **Insert 3**:
   - Calculate `h(3) = 2(3) + 3 = 6 + 3 = 9`.
   - Try inserting at index `9 % 10 = 9`. Index 9 is free, so insert **3** at index 9.
2. **Insert 2**:
   - Calculate `h(2) = 2(2) + 3 = 4 + 3 = 7`.
   - Try inserting at index `7 % 10 = 7`. Index 7 is free, so insert **2** at index 7.
3. **Insert 9**:
   - Calculate `h(9) = 2(9) + 3 = 18 + 3 = 21`.
   - Try inserting at index `21 % 10 = 1`. Index 1 is free, so insert **9** at index 1.
4. **Insert 6**:
   - Calculate `h(6) = 2(6) + 3 = 12 + 3 = 15`.
   - Try inserting at index `15 % 10 = 5`. Index 5 is free, so insert **6** at index 5.
5. **Insert 11**:
   - Calculate `h(11) = 2(11) + 3 = 22 + 3 = 25`.
   - Try inserting at index `25 % 10 = 5`. Index 5 is occupied by 6 (collision).
   - Use quadratic probing: `(5 + 1²) % 10 = 6`. Index 6 is free, so insert **11** at index 6.
6. **Insert 13**:
   - Calculate `h(13) = 2(13) + 3 = 26 + 3 = 29`.
   - Try inserting at index `29 % 10 = 9`. Index 9 is occupied by 3 (collision).
   - Use quadratic probing: `(9 + 1²) % 10 = 0`. Index 0 is free, so insert **13** at index 0.
7. **Insert 7**:
   - Calculate `h(7) = 2(7) + 3 = 14 + 3 = 17`.
   - Try inserting at index `17 % 10 = 7`. Index 7 is occupied by 2 (collision).
   - Use quadratic probing: `(7 + 1²) % 10 = 8`. Index 8 is free, so insert **7** at index 8.
8. **Insert 12**:
   - Calculate `h(12) = 2(12) + 3 = 24 + 3 = 27`.
   - Try inserting at index `27 % 10 = 7`. Index 7 is occupied by 2 (collision).
   - Use quadratic probing: `(7 + 1²) % 10 = 8`. Index 8 is occupied by 7 (collision).
   - Use quadratic probing again: `(7 + 2²) % 10 = 11`. Index 1 is occupied by 1