Types of Hash Functions

1. Division Method

- **Description**: Uses the modulus operation to determine the hash value. Simple but can lead to clustering if `m` is not chosen properly.
- **Formula**: `h(k) = k mod m`
- **Java Code**:

```
public int divisionHash(int key, int tableSize) {
        return key % tableSize;
    }
```

2. Multiplication Method

- **Description**: Uses multiplication and fractional part extraction to determine the hash value. Often leads to better distribution.
- **Formula**: `h(k) = floor(m * (k * A mod 1))`
- **Java Code**:

```
public int multiplicationHash(int key, int tableSize) {
    double A = 0.6180339887; // (sqrt(5) - 1) / 2
    return (int) Math.floor(tableSize * (key * A % 1));
}
```

3. Mid-Square Method

- **Description**: Squares the key and uses the middle part of the result to determine the hash value.
- **Java Code**:

```
public int midSquareHash(int key, int tableSize) {
    int squared = key * key;
    String squaredStr = Integer.toString(squared);
    int midLength = squaredStr.length() / 2;
    String midStr = squaredStr.substring(midLength / 2, midLength / 2 +
midLength);
```

```
        int midValue = Integer.parseInt(midStr);
        return midValue % tableSize;
    }
```

4. Folding Method

- **Description**: Divides the key into equal-sized parts, adds them together to obtain the hash value.
- **Java Code**:

```
public int foldingHash(String key, int tableSize) {
    int sum = 0;
    for (int i = 0; i < key.length(); i++) {
        sum += key.charAt(i);
    }
    return sum % tableSize;
}
```

5. Radix Transformation Method

- **Description**: Transforms keys using a certain base and then applies the hash function.
- **Java Code**:

```
public int radixHash(String key, int base, int tableSize) {
    int hashValue = 0;
    for (int i = 0; i < key.length(); i++) {
        hashValue = (base * hashValue + key.charAt(i)) % tableSize;
    }
    return hashValue;
}
```

6. Universal Hashing

- **Description**: Uses a randomly chosen hash function from a set of hash functions. Reduces the probability of collisions.
- **Java Code**:

```
import java.util.Random;

public class UniversalHash {
```

```
    private int a, b, p, m;

    public UniversalHash(int m) {
        this.m = m;
        Random rand = new Random();
        p = 2147483647; // A large prime number
        a = rand.nextInt(p - 1) + 1; // 1 <= a < p
        b = rand.nextInt(p); // 0 <= b < p
    }

    public int hash(int key) {
        return ((a * key + b) % p) % m;
    }
}
```

7. DJB2 Hash

- **Description**: Created by Daniel J. Bernstein. Simple and efficient, often used in practice.
- **Java Code**:

```
public int djb2Hash(String key, int tableSize) {
    long hash = 5381;
    for (int i = 0; i < key.length(); i++) {
        hash = ((hash << 5) + hash) + key.charAt(i); // hash * 33 + c
    }
    return (int) (hash % tableSize);
}
```

8. MurmurHash

- **Description**: A non-cryptographic hash function known for its good distribution properties and performance.
- **Java Code**:

```
import java.nio.ByteBuffer;
import java.nio.ByteOrder;

public int murmurHash(byte[] data, int tableSize) {
    int seed = 0x9747b28c;
    int m = 0x5bd1e995;
    int r = 24;
    int len = data.length;

    int h = seed ^ len;
```

```
        int i = 0;
        while (len >= 4) {
            int k = ByteBuffer.wrap(data, i,
    4).order(ByteOrder.LITTLE_ENDIAN).getInt();
            k *= m;
            k ^= k >>> r;
            k *= m;
            h *= m;
            h ^= k;
            i += 4;
            len -= 4;
        }

        switch (len) {
            case 3:
                h ^= (data[i + 2] & 0xFF) << 16;
            case 2:
                h ^= (data[i + 1] & 0xFF) << 8;
            case 1:
                h ^= (data[i] & 0xFF);
                h *= m;
        }

        h ^= h >>> 13;
        h *= m;
        h ^= h >>> 15;

        return h % tableSize;
    }
```