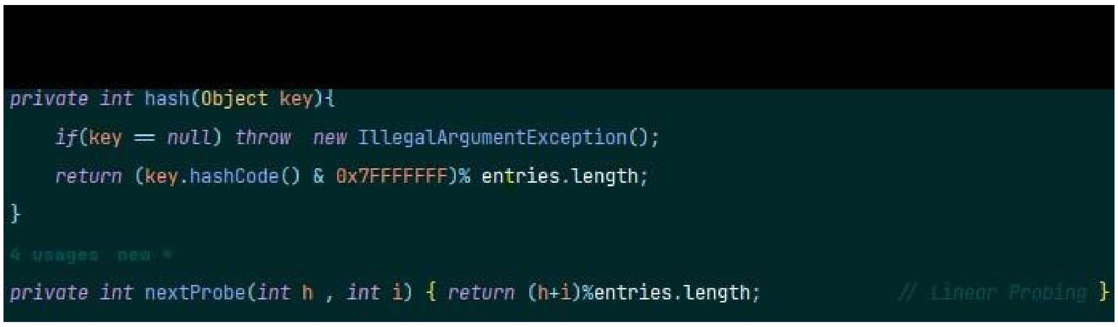
LINEAR PROBING VS QUADRATIC PROABING VS DOUBLE HASHING VS SEPARATE CHANNING

1. Linear Probing:

- ***How it works:*** In linear probing, if a collision occurs at position \(h\), the algorithm checks position \(h+1\), then \(h+2\), and so on until it finds an empty slot or reaches the end of the table.
- ***Example:***
  - Suppose we have a hash table with an array of length 7.
  - Insertion sequence: 25, 42, 36, 19, 13.
  - Resulting table:

```
Index:  0   1   2   3   4   5   6
Key:   42  36  19  13  25      36
```

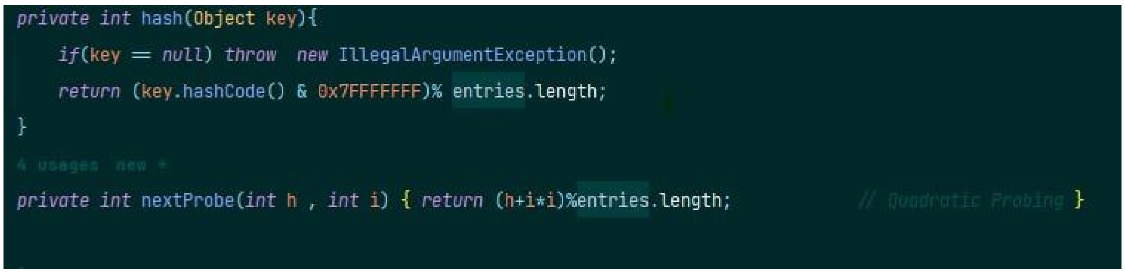
- ***Pros:*** Simple implementation, less computation overhead.
- ***Cons:*** Clustering, performance degrades with high load factors.



```java
private int hash(Object key){
    if(key == null) throw  new IllegalArgumentException();
    return (key.hashCode() & 0x7FFFFFFF)% entries.length;
}
4 usages  new *
private int nextProbe(int h , int i) { return (h+i)%entries.length;          // Linear Probing }
```

2. Quadratic Probing:

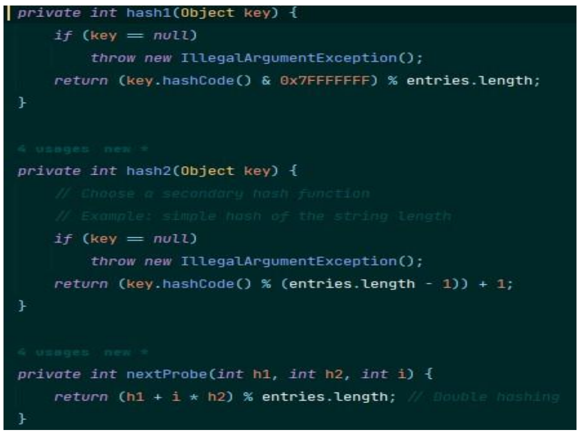
- ***How it works:*** In quadratic probing, the probe sequence is calculated using a quadratic function: \((h + i^2) \mod N \), where \(h\) is the initial hash, \(i\) is the probe number, and \(N\) is the size of the table.
- ***Example:***
  - Example same as before, with the quadratic probing probe sequence being \(h + i^2\) instead of linear increments.
- ***Pros:*** Reduced clustering compared to linear probing.
- ***Cons****:* Secondary clustering, as it suffers from secondary clustering around the original hash index.



```java
private int hash(Object key){
    if(key == null) throw  new IllegalArgumentException();
    return (key.hashCode() & 0x7FFFFFFF)% entries.length;
}
4 usages  new *
private int nextProbe(int h , int i) { return (h+i*i)%entries.length;          // Quadratic Probing }
```

3. Double Hashing:

- ***How it works:*** Double hashing uses two hash functions to calculate the probe sequence: \((h_1 + i \times h_2) \mod N \), where \(h_1\) is the primary hash, \(h_2\) is the secondary hash, \(i\) is the probe number, and \(N\) is the size of the table.
- ***Example:***
  - Example same as before, but with the double hashing probe sequence being \(h_1 + i \times h_2\).
- ***Pros:*** Reduced clustering, better distribution of keys.
- ***Cons:*** Slightly more complex implementation, additional overhead of computing two hash functions.



```java
private int hash1(Object key) {
    if (key == null)
        throw new IllegalArgumentException();
    return (key.hashCode() & 0x7FFFFFFF) % entries.length;
}

4 usages  new *
private int hash2(Object key) {
    // Choose a secondary hash function
    // Example: simple hash of the string length
    if (key == null)
        throw new IllegalArgumentException();
    return (key.hashCode() % (entries.length - 1)) + 1;
}

4 usages  new *
private int nextProbe(int h1, int h2, int i) {
    return (h1 + i * h2) % entries.length; // Double hashing
}
```

4. Separate Chaining:

- ***How it works:*** In Separate Chaining, each slot in the hash table holds a linked list (or another data structure) of collided keys. Collided keys are stored independently in these lists, avoiding clustering.
- ***Example:***
  - Suppose we have a hash table with an array of length 5 and separate chaining to handle collisions.
  - Insertion sequence: 25, 42, 36, 19, 13.
  - Resulting table:

```
Index:  0      1       2     3     4
Keys:   42     36 -> 25  19  13
```

- ***Pros:*** Minimal clustering, efficient for a wide range of load factors, maintains good performance even with high load factors.
- ***Cons:*** Requires additional memory for linked lists or other data structures, slightly more complex implementation compared to linear probing.

Clustering and it's types

Clustering in the context of hash tables refers to the phenomenon where consecutive insertions or collisions result in keys being placed in nearby slots in the hash table. This can lead to inefficient performance due to increased collision rates and longer probe sequences.

Clustering occurs due to the nature of collision resolution techniques, especially in scenarios where multiple keys hash to the same index or nearby indices. Here's why clustering occurs:

1. Primary Clustering:

- In linear probing, when a collision occurs, the algorithm checks the next slot sequentially until an empty slot is found. If multiple keys collide at the same index, they will be placed in consecutive slots, forming clusters.

- In quadratic probing, while the probe sequence spreads out more rapidly than linear probing, collisions still tend to cause clustering, especially if multiple keys have similar initial hash values.

2.Secondary Clustering:
- Secondary clustering occurs when different keys that hash to different indices end up colliding at the same probe sequence due to the secondary function used in techniques like double hashing.
- While double hashing reduces primary clustering, secondary clustering may still occur, particularly if the secondary hash function does not sufficiently spread out the probe sequence.

Clustering can lead to several performance issues:

- Increased probe sequence lengths: Longer probe sequences mean more iterations to find an empty slot or the desired key, impacting the time complexity of operations.
- Reduced cache performance: Clustering can lead to poor cache utilization, as accessing nearby memory locations may not benefit from cache locality.
- Uneven distribution: Clusters can cause uneven distribution of keys within the hash table, leading to inefficient use of memory and decreased look up performance.

To mitigate clustering, hash table implementations often use techniques like rehashing (re sizing the table and redistributing keys), dynamic re sizing strategies, or alternative collision resolution methods such as chaining (using linked lists or other data structures to store collided keys) in addition to the ones discussed earlier.

More On these Techniques

1. Linear Probing:
- ***Problem:*** Linear probing addresses the issue of collisions by sequentially searching for an empty slot when a collision occurs.
- ***Solution:*** It ensures that collided keys are placed in consecutive slots in the hash table, reducing the likelihood of clustering and minimizing memory overhead.
- ***Use Case:*** Linear probing is suitable for scenarios where simplicity of implementation and space efficiency are priorities, and where the load factor is expected to remain relatively low.

2. Quadratic Probing:
- ***Problem:*** Quadratic probing aims to reduce primary clustering compared to linear probing, where consecutive collisions result in keys being placed in nearby slots.
- ***Solution***: By using a quadratic function to calculate the probe sequence, quadratic probing spreads out collided keys more quickly, reducing the likelihood of primary clustering.
- ***Use Case:*** Quadratic probing is beneficial when better clustering avoidance than linear probing is desired, while still maintaining a relatively simple implementation.

3. Double Hashing:
- ***Problem:*** Double hashing addresses the challenge of clustering by using two hash functions to calculate the probe sequence, providing better key distribution and reducing clustering.
- ***Solution:*** By combining two hash functions, double hashing offers improved distribution of keys, leading to fewer collisions and better performance under higher load factors.
- ***Use Case:*** Double hashing is suitable for scenarios where avoiding clustering is critical for maintaining look up performance, and where a moderate level of complexity in implementation is acceptable.

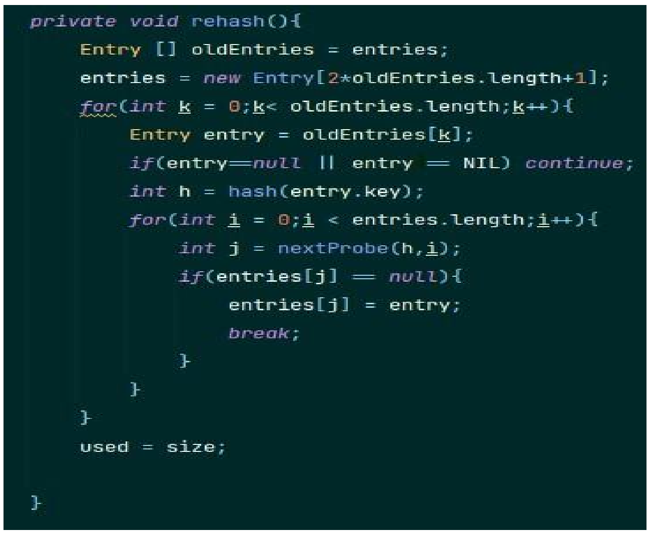
4. Separate Chaining:
- ***Problem:*** Separate chaining tackles clustering by storing collided keys in separate linked lists or arrays, preventing them from being placed in consecutive slots.
- ***Solution:*** Each slot in the hash table holds a reference to a linked list or array, allowing collided keys to be stored independently and avoiding clustering issues altogether.

- ***Use Case:*** Separate chaining is ideal when minimizing clustering is a top priority, and when efficient insertion and deletion operations are required even under high load factors.

Some Basic Concepts

1. Rehashing:

- ***Purpose:*** The `rehash` method is used to resize the hash table when the number of elements (or the load) exceeds a certain threshold. Rehashing involves creating a new, larger hash table, recalculating the hash values of all elements, and redistributing them across the new table.
- ***Use:*** Rehashing ensures that the hash table remains efficient by preventing excessive clustering and maintaining a suitable load factor.
- ***Implementation:*** Typically, rehashing is triggered when the load factor exceeds a predefined threshold (e.g., 0.75). When this happens, the hash table is re sized to approximately double its previous size, which helps distribute the elements more evenly and reduces the likelihood of collisions.



```java
private void rehash(){
    Entry [] oldEntries = entries;
    entries = new Entry[2*oldEntries.length+1];
    for(int k = 0;k< oldEntries.length;k++){
        Entry entry = oldEntries[k];
        if(entry==null || entry == NIL) continue;
        int h = hash(entry.key);
        for(int i = 0;i < entries.length;i++){
            int j = nextProbe(h,i);
            if(entries[j] == null){
                entries[j] = entry;
                break;
            }
        }
    }
    used = size;

}
```

2. Load Factor:

- ***Definition:*** The load factor of a hash table is the ratio of the number of elements stored in the table to the size of the table. It is calculated as \(\text{Load Factor} = \frac{\text{Number of Elements}}{\text{Size of Table}} \).
- ***Purpose:*** The load factor provides an indication of how full the hash table is. It helps determine when to trigger rehashing to maintain performance.
- ***Use:*** By monitoring the load factor, you can dynamically adjust the size of the hash table to ensure that it remains efficient. When the load factor exceeds a certain threshold (e.g., 0.75), rehashing is performed to resize the table and redistribute the elements, preventing excessive clustering and improving performance.
- ***Impact:*** A higher load factor means that the hash table is more densely populated, increasing the likelihood of collisions and potentially degrading performance. By keeping the load factor within an acceptable range, you can balance memory usage and look up efficiency.

3. Size: The `size` variable represents the number of elements currently stored in the hash table. It is incremented whenever an element is added and decremented whenever an element is removed. The size is used to calculate the load factor and determine when to trigger rehashing.

4. Capacity: The `capacity` variable represents the total number of slots available in the hash table. It may be equal to or greater than the current size of the table. The capacity is used to determine the initial size of the table and may be adjusted during rehashing to resize the table as needed.

5. Used: The `used` variable is often used in rehashing algorithms to track the number of elements that have been moved or reinserted during the re sizing process. It helps ensure that all elements are properly redistributed across the new hash table.