Batch: D3	**Roll No.: 16010123294**
Experiment / assignment / tutorial No. 10	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title: Implementation of Hashing - Linear and quadratic hashing

Objective: To Understand and Implement Linear and Quadratic Hashing

Expected Outcome of Experiment:

CO	Outcome
4	Demonstrate sorting and searching methods.

Books/ Journals/ Websites referred:

1. *Fundamentals Of Data Structures In C* – Ellis Horowitz, Satraj Sahni, Susan Anderson-Fred
2. *An Introduction to data structures with applications* – Jean Paul Tremblay, Paul G. Sorenson
3. *Data Structures A Pseudo Approach with C* – Richard F. Gilberg & Behrouz A. Forouzan

Abstract:

Linear and quadratic hashing are two methods used in hash table implementations to manage collisions—situations where two keys hash to the same index in a table. These techniques are crucial for ensuring efficient data retrieval, storage, and overall performance of hash-based data structures.

Linear Hashing

In linear hashing, when a collision occurs, the algorithm searches for the next available slot in a sequential manner. This means that if a key hashes to a position that is already occupied, the algorithm checks the next index, and continues this process until an empty slot is found. This approach is simple and easy to implement, but it can lead to a phenomenon known as "clustering," where groups of filled slots form. This clustering can degrade performance, especially as the load factor (the ratio of filled slots to total slots) increases, resulting in longer search times.

Quadratic Hashing

Quadratic hashing offers a solution to the clustering problem associated with linear hashing. Instead of searching for the next available slot linearly, quadratic hashing uses a quadratic function to probe for an open slot. For instance, if a collision occurs at index h(k)h(k)h(k), the algorithm will check h(k)+12,h(k)+22,h(k)+32h(k) + 1^2, h(k) + 2^2, h(k) + 3^2h(k)+12,h(k)+22,h(k)+32, and so on, effectively spreading out the probe sequence. This reduces the chances of clustering and generally leads to better performance, particularly in scenarios with higher load factors.

Comparison

Both methods have their advantages and disadvantages. Linear hashing is straightforward and can be easier to implement, while quadratic hashing generally provides better performance due to reduced clustering. However, quadratic hashing can complicate the search process and requires careful consideration of the probing sequence to ensure that all entries can be accessed effectively.

Applications

These hashing techniques are widely used in databases, caching mechanisms, and data structures where efficient access to elements is critical. Understanding the strengths and weaknesses of each approach allows developers to choose the most suitable method based on the specific requirements of their applications, such as load characteristics and expected usage patterns.

Algorithm for Implementation:

Linear Hashing Algorithm:

1. **Initialization**:
    - Create an array (hash table) of size m.
    - Set all entries to null or a sentinel value (e.g., `None`).
2. **Hash Function**:
    - Define a hash function h(k)=k mod m to compute the initial index for a key k.
3. **Insertion**:
    - Compute the initial index: `index = h(k)`.
    - If `hash_table[index]` is empty:
        - Insert the key k at `hash_table[index]`.
    - If `hash_table[index]` is occupied:
        - Use a linear probe:
            - Set `i = 1`.
            - While `hash_table[(index + i) % m]` is occupied:
                - Increment `i`.
            - Insert the key k at `hash_table[(index + i) % m]`.
4. **Search**:
    - Compute the initial index: `index = h(k)`.
    - If `hash_table[index]` is equal to k, return the index.
    - If not, use linear probing:
        - Set `i = 1`.
        - While `hash_table[(index + i) % m]` is not empty:
            - If `hash_table[(index + i) % m]` is equal to k, return the index.
            - Increment `i`.
    - If you reach an empty slot, the key is not in the table.
5. **Deletion**:
    - Compute the index using the search algorithm.
    - If found, mark the slot as deleted (using a special marker or simply null).

Quadratic Hashing Algorithm:

1. **Initialization**:
    - Create an array (hash table) of size m.
    - Set all entries to null or a sentinel value.
2. **Hash Function**:
    - Define a hash function h(k)=k mod m
3. **Insertion**:
    - Compute the initial index: `index = h(k)`.
    - If `hash_table[index]` is empty:
        - Insert the key kkk at `hash_table[index]`.
    - If `hash_table[index]` is occupied:

- Use quadratic probing:
  - Set `i = 1`.
  - While `hash_table[(index + i^2) % m]` is occupied:
    - Increment `i`.
  - Insert the key k at `hash_table[(index + i^2) % m]`.

4. **Search**:
   - Compute the initial index: `index = h(k)`.
   - If `hash_table[index]` is equal to k, return the index.
   - If not, use quadratic probing:
     - Set `i = 1`.
     - While `hash_table[(index + i^2) % m]` is not empty:
       - If `hash_table[(index + i^2) % m]` is equal to k, return the index.
       - Increment `i`.
   - If you reach an empty slot, the key is not in the table.
5. **Deletion**:
   - Compute the index using the search algorithm.
   - If found, mark the slot as deleted.

Program:

Linear Hashing:

Quadratic Hashing:

```
#include <stdio.h>

#define TABLE_SIZE 10

int hashTable[TABLE_SIZE];

void initializeTable() {
```

```
    for (int i = 0; i < TABLE_SIZE; i++)
        hashTable[i] = -1;  // Initialize all slots as empty
}

int hashFunction(int key) {
    return key % TABLE_SIZE;
}

void displayTable() {
    printf("Hash Table (Quadratic Probing):\n");
    for (int i = 0; i < TABLE_SIZE; i++)
        printf("Index %d: %d\n", i, hashTable[i]);
    printf("\n");
}

void insert(int key) {
    int hashIndex = hashFunction(key);
    int i = 0;

    while (hashTable[hashIndex] != -1) {  // Collision occurs
        i++;
        hashIndex = (hashFunction(key) + i * i) % TABLE_SIZE;  // Quadratic Probing
    }
    hashTable[hashIndex] = key;
```

```
    // Display the table after each insertion
    printf("After inserting key %d:\n", key);
    displayTable();
}

int main() {
    int numKeys, key;

    initializeTable();

    printf("Enter the number of keys to insert: ");
    scanf("%d", &numKeys);

    for (int i = 0; i < numKeys; i++) {
        printf("Enter key %d: ", i + 1);
        scanf("%d", &key);
        insert(key);
    }

    return 0;
}
```

Output:

Linear Hashing:

```
Enter the number of keys to insert: 5
Enter key 1: 3
After inserting key 3:
Hash Table (Linear Probing):
Index 0: -1
Index 1: -1
Index 2: -1
Index 3: 3
Index 4: -1
Index 5: -1
Index 6: -1
Index 7: -1
Index 8: -1
Index 9: -1
```

```
Enter key 2: 12
After inserting key 12:
Hash Table (Linear Probing):
Index 0: -1
Index 1: -1
Index 2: 12
Index 3: 3
Index 4: -1
Index 5: -1
Index 6: -1
Index 7: -1
Index 8: -1
Index 9: -1
```

```
Enter key 3: 7
After inserting key 7:
Hash Table (Linear Probing):
Index 0: -1
Index 1: -1
Index 2: 12
Index 3: 3
Index 4: -1
Index 5: -1
Index 6: -1
Index 7: 7
Index 8: -1
Index 9: -1
```

```
Enter key 4: 13
After inserting key 13:
Hash Table (Linear Probing):
Index 0: -1
Index 1: -1
Index 2: 12
Index 3: 3
Index 4: 13
Index 5: -1
Index 6: -1
Index 7: 7
Index 8: -1
Index 9: -1
```

```
Enter key 5: 94
After inserting key 94:
Hash Table (Linear Probing):
Index 0: -1
Index 1: -1
Index 2: 12
Index 3: 3
Index 4: 13
Index 5: 94
Index 6: -1
Index 7: 7
Index 8: -1
Index 9: -1
```

Quadratic Hashing :

```
Enter the number of keys to insert: 5
Enter key 1: 10
After inserting key 10:
Hash Table (Quadratic Probing):
Index 0: 10
Index 1: -1
Index 2: -1
Index 3: -1
Index 4: -1
Index 5: -1
Index 6: -1
Index 7: -1
Index 8: -1
Index 9: -1
```

```
Enter key 2: 56
After inserting key 56:
Hash Table (Quadratic Probing):
Index 0: 10
Index 1: -1
Index 2: -1
Index 3: -1
Index 4: -1
Index 5: -1
Index 6: 56
Index 7: -1
Index 8: -1
Index 9: -1
```

```
Enter key 3: 24
After inserting key 24:
Hash Table (Quadratic Probing):
Index 0: 10
Index 1: -1
Index 2: -1
Index 3: -1
Index 4: 24
Index 5: -1
Index 6: 56
Index 7: -1
Index 8: -1
Index 9: -1
```

```
Enter key 4: 77
After inserting key 77:
Hash Table (Quadratic Probing):
Index 0: 10
Index 1: -1
Index 2: -1
Index 3: -1
Index 4: 24
Index 5: -1
Index 6: 56
Index 7: 77
Index 8: -1
Index 9: -1
```

```
Enter key 5: 63
After inserting key 63:
Hash Table (Quadratic Probing):
Index 0: 10
Index 1: -1
Index 2: -1
Index 3: 63
Index 4: 24
Index 5: -1
Index 6: 56
Index 7: 77
Index 8: -1
Index 9: -1
```

Conclusion:-
We learned of Linear and Quadratic Hashing and implemented it in our code.

Post Lab Questions:

1) Explain how linear hashing resolves collisions. What are the potential drawbacks of this method?

Linear hashing resolves collisions by sequentially checking the next slot in the hash table when a collision occurs. If the calculated slot is occupied, it moves one position forward (wrapping to the start if needed) until an empty slot is found. This is called linear probing.

Drawbacks:

Clustering: Consecutive occupied slots, or "primary clustering," can lead to slower insertions and searches as the table fills.

Limited Space Utilization: As the table becomes fuller, more probes are often needed to find an empty slot, reducing efficiency.

2) Describe the probing sequence used in quadratic hashing. How does this sequence differ from that of linear hashing?

In quadratic hashing, the probing sequence for resolving collisions follows a quadratic function. If a collision occurs, the hash function recalculates the next position as:

where iii is the number of probes attempted (1, 2, 3, ...). This means that after each collision, the next slot is checked further from the original hash position in a non-linear sequence.

Difference from Linear Hashing

In **linear hashing**, the probing sequence moves one slot at a time, creating a consecutive search pattern (e.g., hashIndex+1,hashIndex+2,...\text{hashIndex} + 1, \text{hashIndex} + 2, ...hashIndex+1,hashIndex+2,...), leading to linear progression. In

contrast, **quadratic hashing** increases the interval between slots as probes continue (e.g., hashIndex+12,hashIndex+22,...\text{hashIndex} + 1^2, \text{hashIndex} + 2^2, ...hashIndex+12,hashIndex+22,...), spreading out the probes and reducing primary clustering. However, this can make quadratic hashing complex and less predictable in finding open slots.

3) What are some challenges you encountered when implementing linear or quadratic hashing in your lab? How did you overcome them?

Collision Handling: Ensuring the probing sequence wraps around correctly at the table's end. This was managed by using modulo operations to stay within bounds.

Secondary Clustering (in Linear Probing): Consecutive occupied slots slow down insertion and lookup. To reduce this, we tried using a larger table size relative to the number of keys.

Unreachable Slots (in Quadratic Probing): Not all slots may be reachable with quadratic probing, especially in non-prime table sizes. Using a prime table size and careful probing logic helped minimize this issue.

4) In what scenarios might you prefer one hashing technique over the other? Provide specific examples.

Linear Hashing: Suitable for cases where simplicity and memory efficiency are prioritized. For example, **caching systems** with smaller datasets and frequent inserts can benefit from linear hashing, as it has predictable, sequential probing.

Quadratic Hashing: Preferred when clustering needs to be minimized and the hash table load is high. For instance, in **database indexing**, where large datasets require quick lookups with fewer collisions, quadratic probing reduces the chance of long collision chains, improving search performance.

Generally, **linear hashing** is simpler and faster for lightly loaded tables, while **quadratic hashing** is better for denser tables where clustering can significantly affect performance.