Batch: A1 **Roll No.: 16010123012**

Experiment No. 10

Grade: AA / AB / BB / BC / CC / CD /DD

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, returnthe 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:

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
    int ar[7];
    int hash[10] = {0};
    bool occupied[10] = {false};
    int i;

    printf("Enter 7 elements:\n");
    for (i = 0; i < 7; i++) {
        scanf("%d", &ar[i]);
    }

    for (i = 0; i < 7; i++) {
        int j = 0;
        bool flag = true;
```

```
        while (flag) {
            int index = (ar[i] + j) % 10;

            if (!occupied[index]) {
                hash[index] = ar[i];
                occupied[index] = true;
                printf("Inserted: %d at index %d\n", ar[i], index);
                flag = false;
            }
            j++;
        }
    }

    printf("Hash Table (Linear Probing):\n");
    for (i = 0; i < 10; i++) {
        if (occupied[i]) {
            printf("Index %d: %d\n", i, hash[i]);
        }
    }
    return 0;
}
```

```
#include <stdio.h>
#include <stdbool.h>

int main() {
    int ar[7];
    int hash[10] = {0};
    bool occupied[10] = {false};
    int i;

    printf("Enter 7 elements:\n");
    for (i = 0; i < 7; i++) {
        scanf("%d", &ar[i]);
    }

    for (i = 0; i < 7; i++) {
        int j = 0;
        bool flag = true;

        while (flag) {
            int index = (ar[i] + j * j) % 10;

            if (!occupied[index]) {
                hash[index] = ar[i];
```

```
                occupied[index] = true;
                printf("Inserted: %d at index %d\n", ar[i], index);
                flag = false;
            }
            j++;
        }
    }

    printf("Hash Table (Quadratic Probing):\n");
    for (i = 0; i < 10; i++) {
        if (occupied[i]) {
            printf("Index %d: %d\n", i, hash[i]);
        }
    }
    return 0;
}
```

Output:

```
Enter 7 elements:
3
41
134
45
76
34
2
Inserted: 3 at index 3
Inserted: 41 at index 1
Inserted: 134 at index 4
Inserted: 45 at index 5
Inserted: 76 at index 6
Inserted: 34 at index 7
Inserted: 2 at index 2
Hash Table (Linear Probing):
Index 1: 41
Index 2: 2
Index 3: 3
Index 4: 134
Index 5: 45
Index 6: 76
Index 7: 34
```

```
Enter 7 elements:
13
45
21
43
12
68
54
Inserted: 13 at index 3
Inserted: 45 at index 5
Inserted: 21 at index 1
Inserted: 43 at index 4
Inserted: 12 at index 2
Inserted: 68 at index 8
Inserted: 54 at index 0
Hash Table (Quadratic Probing):
Index 0: 54
Index 1: 21
Index 2: 12
Index 3: 13
Index 4: 43
Index 5: 45
Index 8: 68
```

Conclusion:-

We have learned about hashing, including how to insert data and use an array to store the information along with the corresponding hash mappings.

Post Lab Questions:

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

this method?
Linear hashing resolves collisions through open addressing, using a linear probing strategy where it sequentially checks the next buckets for an empty slot when a collision occurs. This method allows for dynamic resizing of the hash table as the number of keys increases.
Drawbacks are; it can lead to clustering, where occupied buckets group together, increasing the likelihood of collisions and degrading performance. Additionally, rehashing during table resizing incurs overhead, and maintaining an optimal load factor is crucial to avoid performance issues.

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 uses a quadratic function, checking positions with the formula index=(h(k)+i2)mod m\text{index} = (h(k) + i2) \mod mindex=(h(k)+i2)modm, where i is the attempt number. This results in increasingly larger gaps between probes, reducing clustering. In contrast, linear hashing uses a simple linear sequence, checking positions with index=(h(k)+i)mod m\text{index} = (h(k) + i) \mod mindex=(h(k)+i)modm, which leads to sequential probing and greater potential for clustering. The quadratic approach helps distribute keys more evenly across the hash table.

3) What are some challenges you encountered when implementing linear or quadratic hashing in your lab? How did you overcome them?
Common challenges in implementing linear or quadratic hashing include:
Clustering: In linear hashing, clustering can degrade performance. Using a better hash function or switching to quadratic probing can help reduce this.
Load Factor Management: Resizing the hash table can be complex. Setting a clear threshold for resizing and implementing efficient rehashing can mitigate this issue.
Collision Handling: Effective collision resolution is crucial. Employing systematic probing strategies or double hashing can improve distribution.
Memory Overhead: Resizing incurs overhead. Planning the initial table size and using dynamic resizing strategies can balance performance and memory use.
Testing and Debugging: Verifying correctness can be challenging. Comprehensive test cases covering various scenarios can ensure reliable implementation.

4) In what scenarios might you prefer one hashing technique over the other? Provide specific examples.
Scenario for Quadratic Hashing: In a large-scale database application where fast lookups are critical, quadratic hashing is preferred due to its better distribution of entries, which reduces clustering.
Example: A social media platform's user profile search feature benefits from quadratic hashing, as it helps maintain efficient access times even with a growing user base.