Batch: E2 Roll No.: 16010123325

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**:

```
    o Compute the initial index: index = h(k).
    o If hash_table[index] is equal to k, return the index.
    o 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.
    o If you reach an empty slot, the key is not in the table.
```

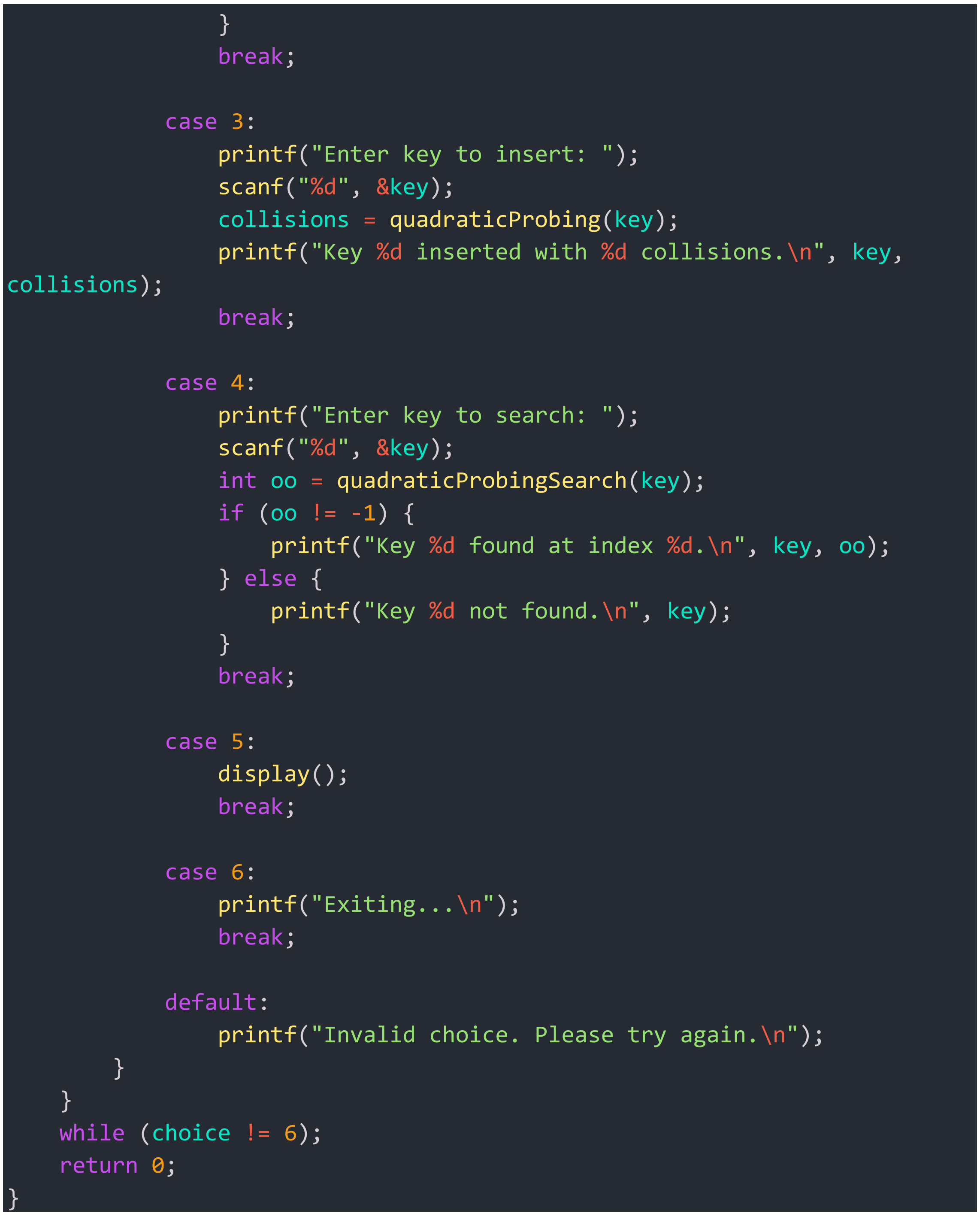
5. **Deletion**:

```
    o Compute the index using the search algorithm.
    o If found, mark the slot as deleted.
```

Program:

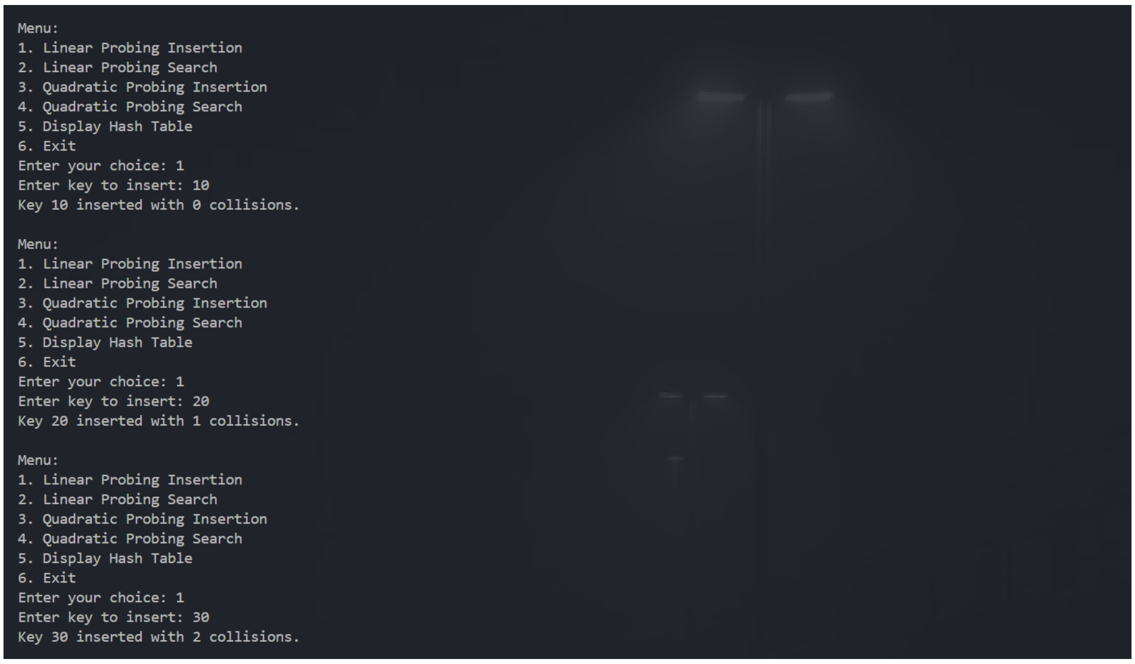
```c
#include <stdio.h>
#include <stdlib.h>
#define n 10

typedef struct {
    int key;
    int isOccupied; // 0 if empty, 1 if occupied
} hashEntry;

hashEntry hashTable[n];

int hash(int key) {
    return key % n;
}

int linearProbing(int key) {
    int index = hash(key);
    int collisions = 0;
    while (hashTable[index].isOccupied) {
        collisions++;
        index = (index + 1) % n;
    }
    hashTable[index].key = key;
    hashTable[index].isOccupied = 1;
    return collisions;

}

int linearProbingSearch(int key) {
    int index = hash(key);
    while (hashTable[index].isOccupied) {
        if (hashTable[index].key == key) return index;
        index = (index + 1) % n;
    }
    return -1;
}

int quadraticProbing(int key) {
    int index = hash(key);
    int collisions = 0;
    int i = 1;
    while (hashTable[index].isOccupied) {
        collisions++;
        i++;
        index = (index + i * i) % n;
    }
    hashTable[index].key = key;
    hashTable[index].isOccupied = 1;
    return collisions;
}

int quadraticProbingSearch(int key) {
    int index = hash(key);
    int i = 0;
    while (hashTable[index].isOccupied) {
        if (hashTable[index].key == key) return index;
        i++;
        index = (index + i * i) % n;
    }
    return -1;
}

void display() {
    printf("Hash Table: \n");
```

```c
    for (int i = 0; i < n; ++i) {
        if (hashTable[i].isOccupied) {
            printf("Index %d: %d\n", i, hashTable[i].key);
        } else {
            printf("Index %d: Empty\n", i);
        }
    }
}

int main() {
    int choice, key, collisions;

    for (int i = 0; i < n; ++i) {
        hashTable[i].isOccupied = 0;
    }

    do {
        printf("\nMenu:\n");
        printf("1. Linear Probing Insertion\n");
        printf("2. Linear Probing Search\n");
        printf("3. Quadratic Probing Insertion\n");
        printf("4. Quadratic Probing Search\n");
        printf("5. Display Hash Table\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                collisions = linearProbing(key);
                printf("Key %d inserted with %d collisions.\n", key,
collisions);
                break;

            case 2:
                printf("Enter key to search: ");
                scanf("%d", &key);
                int searchIndex = linearProbingSearch(key);
                if (searchIndex != -1) {
                    printf("Key %d found at index %d.\n", key,
searchIndex);
                } else {
                    printf("Key %d not found.\n", key);
```

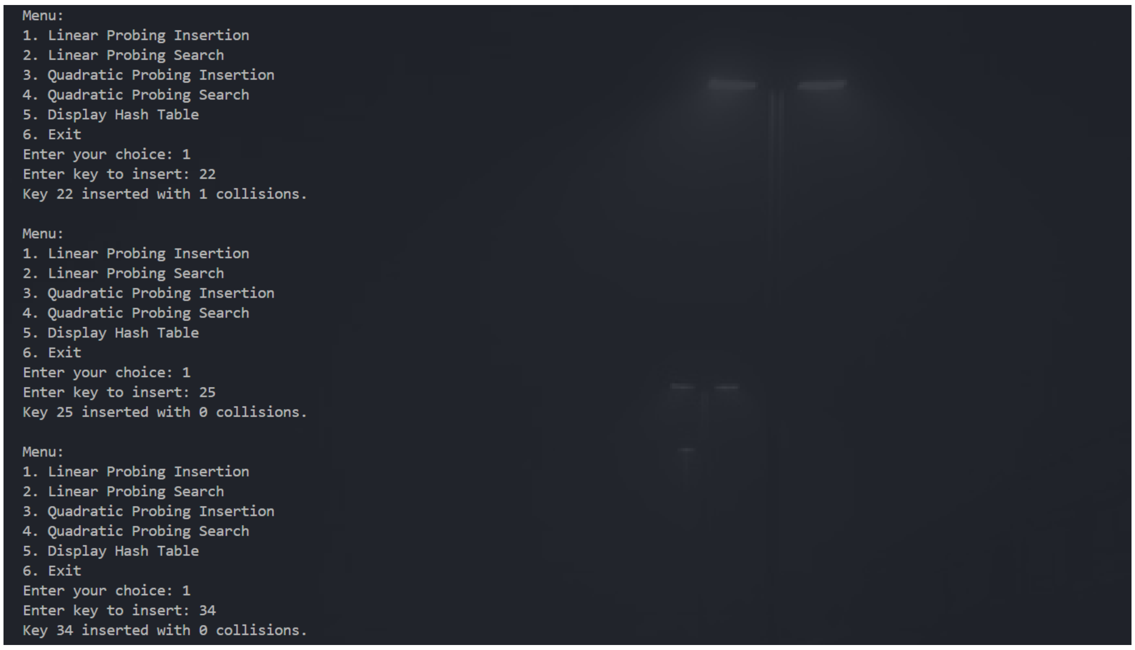


```
                }
                break;

            case 3:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                collisions = quadraticProbing(key);
                printf("Key %d inserted with %d collisions.\n", key,
collisions);
                break;

            case 4:
                printf("Enter key to search: ");
                scanf("%d", &key);
                int oo = quadraticProbingSearch(key);
                if (oo != -1) {
                    printf("Key %d found at index %d.\n", key, oo);
                } else {
                    printf("Key %d not found.\n", key);
                }
                break;

            case 5:
                display();
                break;

            case 6:
                printf("Exiting...\n");
                break;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
    while (choice != 6);
    return 0;
}
```

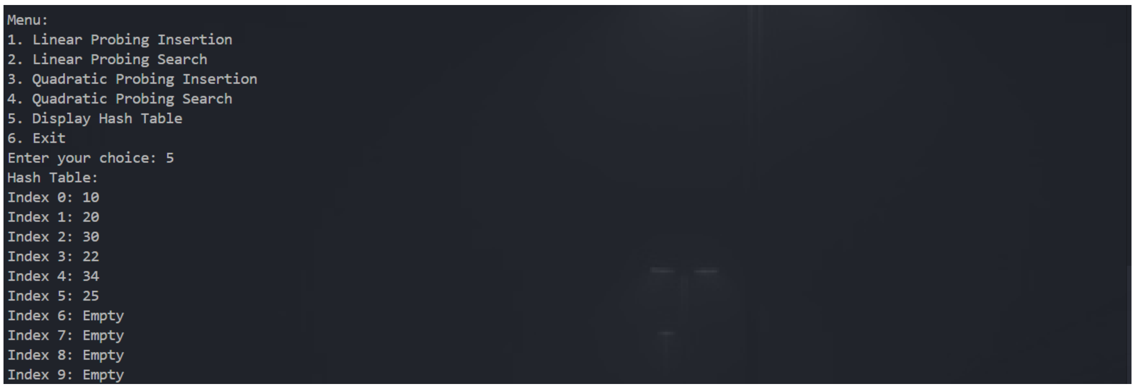
Output:



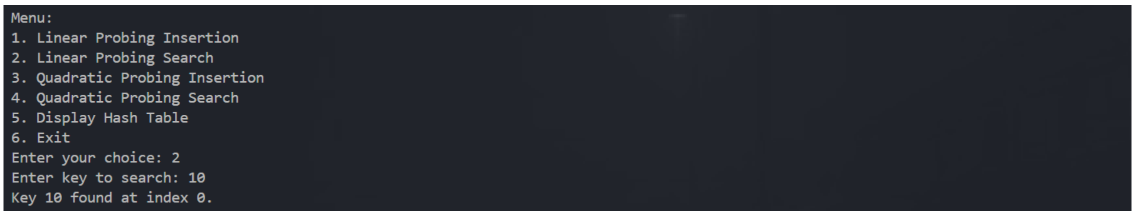
```
Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 1
Enter key to insert: 10
Key 10 inserted with 0 collisions.

Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 1
Enter key to insert: 20
Key 20 inserted with 1 collisions.

Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 1
Enter key to insert: 30
Key 30 inserted with 2 collisions.
```



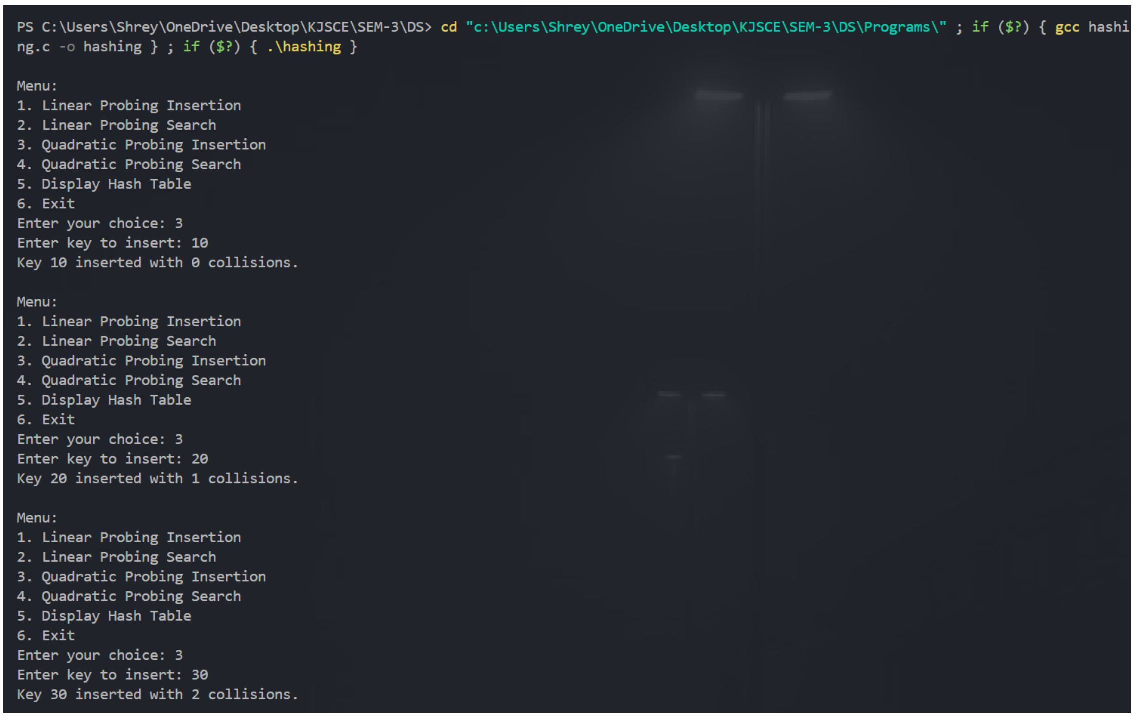
```
Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 1
Enter key to insert: 22
Key 22 inserted with 1 collisions.

Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 1
Enter key to insert: 25
Key 25 inserted with 0 collisions.

Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 1
Enter key to insert: 34
Key 34 inserted with 0 collisions.
```



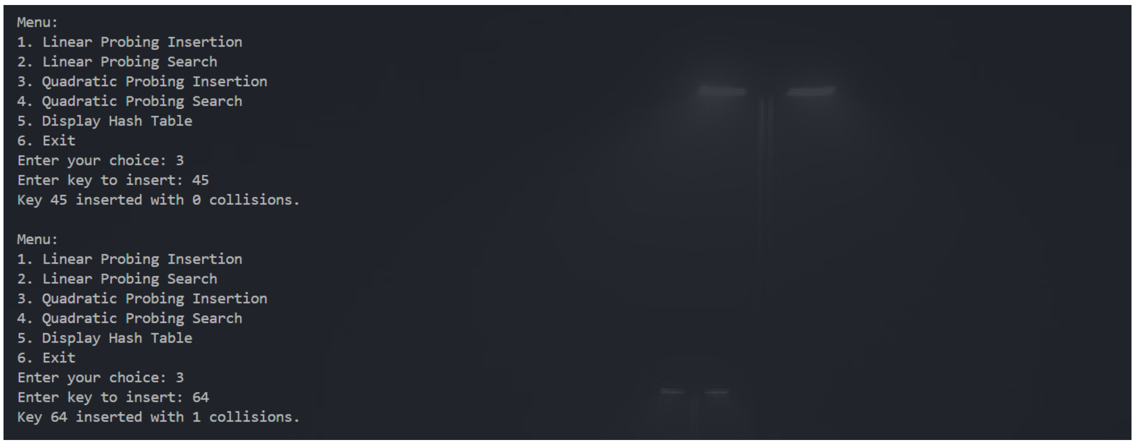
```
Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 5
Hash Table:
Index 0: 10
Index 1: 20
Index 2: 30
Index 3: 22
Index 4: 34
Index 5: 25
Index 6: Empty
Index 7: Empty
Index 8: Empty
Index 9: Empty
```



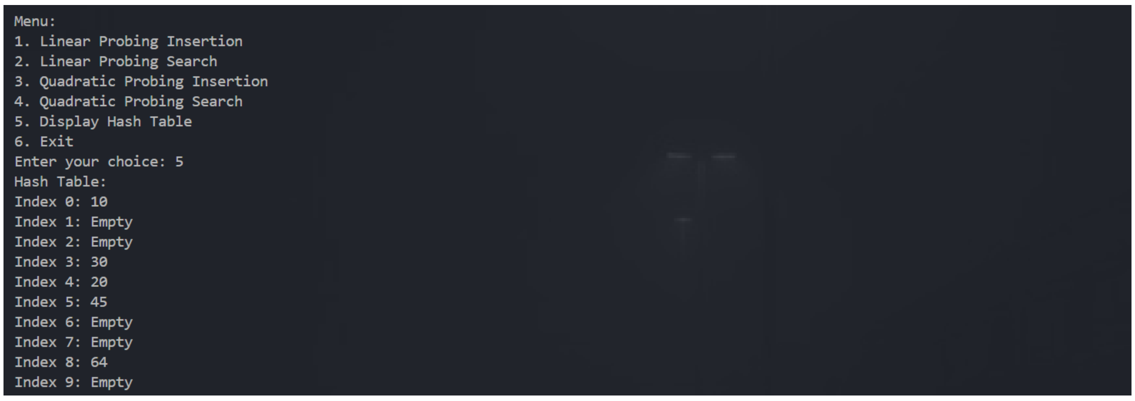
```
Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 2
Enter key to search: 10
Key 10 found at index 0.
```



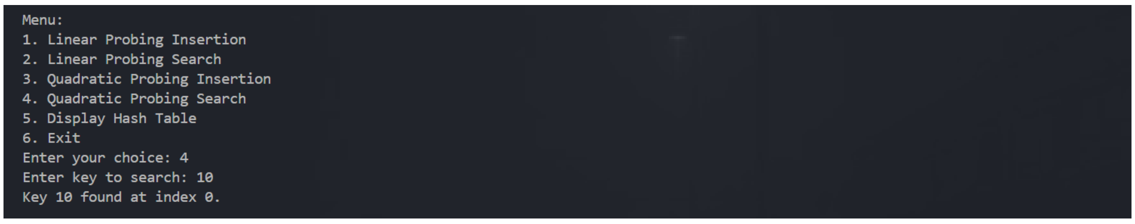
```
PS C:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-3\DS> cd "c:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-3\DS\Programs\" ; if ($?) { gcc hashi
ng.c -o hashing } ; if ($?) { .\hashing }

Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 3
Enter key to insert: 10
Key 10 inserted with 0 collisions.

Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 3
Enter key to insert: 20
Key 20 inserted with 1 collisions.

Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 3
Enter key to insert: 30
Key 30 inserted with 2 collisions.
```



```
Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 3
Enter key to insert: 45
Key 45 inserted with 0 collisions.

Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 3
Enter key to insert: 64
Key 64 inserted with 1 collisions.
```



```
Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 5
Hash Table:
Index 0: 10
Index 1: Empty
Index 2: Empty
Index 3: 30
Index 4: 20
Index 5: 45
Index 6: Empty
Index 7: Empty
Index 8: 64
Index 9: Empty
```



```
Menu:
1. Linear Probing Insertion
2. Linear Probing Search
3. Quadratic Probing Insertion
4. Quadratic Probing Search
5. Display Hash Table
6. Exit
Enter your choice: 4
Enter key to search: 10
Key 10 found at index 0.
```

Conclusion:-
This program effectively demonstrates the implementation of hash table operations using linear and quadratic probing, allowing users to insert, search, and display keys while tracking collisions.

Post Lab Questions:

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

Ans:

Linear hashing resolves collisions by probing the next slot in the hash table in a linear sequence. When a collision occurs, the algorithm checks the next slot, and if it's empty, the key is inserted. If the next slot is also occupied, the algorithm continues to probe subsequent slots until an empty slot is found. The potential drawbacks of this method are:

- Clustering: Linear hashing can lead to clustering, where groups of keys tend to congregate in certain areas of the hash table, reducing the overall performance.
- Cache performance: The linear probing sequence can result in poor cache performance, as the algorithm may access non-contiguous memory locations.
- Collision resolution: In the worst-case scenario, linear hashing can degenerate into a linear search, leading to poor performance.

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

Ans:

Quadratic hashing uses a probing sequence that is quadratic in nature. The sequence is generated using the formula hash(key) + i^2, where i is the probe number. This sequence differs from linear hashing in that it uses a non-linear sequence to probe the hash table. The quadratic probing sequence is designed to minimize clustering and reduce the likelihood of collisions.

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

Ans:

Some challenges encountered when implementing linear and quadratic hashing include:

- Handling edge cases: Ensuring that the probing sequence wraps around the hash table correctly when the end of the table is reached.
- Avoiding infinite loops: Implementing a mechanism to prevent infinite loops when the hash table is full or when a key is not found.
- Optimizing performance: Minimizing the number of probes required to resolve collisions and improving cache performance.

To overcome these challenges, careful consideration was given to the implementation details, including:

- Using modular arithmetic: To ensure that the probing sequence wraps around the hash table correctly.
- Implementing a maximum probe limit: To prevent infinite loops and ensure that the algorithm terminates.
- Optimizing the probing sequence: By using a quadratic probing sequence, which is designed to minimize clustering and reduce collisions.

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

Ans:

The choice between linear and quadratic hashing depends on the specific use case and requirements. Here are some scenarios where one technique might be preferred over the other:

- Linear Hashing:
    - When memory is limited, and a simple, cache-friendly implementation is required.

- When the hash table is relatively small, and the likelihood of collisions is low.

- Quadratic Hashing:
    - When the hash table is large, and collisions are more likely to occur.
    - When cache performance is critical, and a non-linear probing sequence is beneficial.
    - When the application requires a more robust and adaptable hashing technique.

Examples of scenarios where quadratic hashing might be preferred include:

- Database indexing: Where large amounts of data need to be stored and retrieved efficiently.
- Caching mechanisms: Where cache performance is critical, and a non-linear probing sequence can improve hit rates.
- High-performance computing: Where the application requires a robust and adaptable hashing technique to handle large amounts of data.