Batch: B2	**Roll No.: 16010124107**
Experiment / assignment / tutorial No. 9	
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:

```
#include <bits/stdc++.h>

using namespace std;


const int s = 31;
```

```
int hash_table[s];


int hashfunc(int k){
    return k%s;
}


void insertion(){
    cout << "Enter -1 to exit" << endl;
    while(true){
        cout << "Enter value to insert\n";
        int x;
        cin >> x;
        if(x==-1) return;
        int index = hashfunc(x);
        if(hash_table[index]==0){
            hash_table[index]=x;
        }
        else{
            int i=1;
            while(hash_table[(index + i) % s]!=0){
                i++;
            }
            hash_table[(index+i)%s]=x;
        }
```

```
    }
}


int searchKey(){
    cout << "Enter element to search\n";
    int k;
    cin >> k;
    int index = hashfunc(k);
    if(hash_table[index]==k){
        return index;
    }
    else{
        int i=1;
        while(hash_table[(index + i) % s]!=0){
            if(hash_table[(index + i) % s]==k) {
                return (index + i) % s;
            }
            i++;
        }
    }
    return -1;
}


void deletion(){
```

```
    cout << "Enter element to delete\n";
    int k;
    cin >> k;
    int index = hashfunc(k);
    if(hash_table[index]==k){
        hash_table[index]=-1;
    }
    else{
        int i=1;
        while(hash_table[(index + i) % s]!=0){
            if(hash_table[(index + i) % s]==k) {
                hash_table[(index + i) % s] = -1;
                cout << "Element deleted!\n";
                return;
            }
            i++;
        }
    }
    cout << "Element not found\n";
}

int main() {
    int dummy;
    memset(hash_table, 0, sizeof(hash_table));
```

```
    while(true){
        cout << "Menu options: \n1. Insertion\n2. Search\n3. Deletion\n4.
Exit\n";
        int choice;
        cin >> choice;
        switch(choice){
            case 1:
            insertion();
            break;
            case 2:
            dummy = searchKey();
            if(dummy>-1)
            cout << "Index of element is " << dummy << endl;
            else cout << "Element not found\n";
            break;
            case 3:
            deletion();
            break;
            case 4:
            cout << "Bye\n";
            return 0;
            default:
            cout << "Invalid option. Choose again.\n";
        }
```

```
    }

}
```

Output:

```
g++ exp9.cpp -o exp9 } ; if ($?) { .\ex
Menu options:
1. Insertion
2. Search
3. Deletion
4. Exit
1
Enter -1 to exit
Enter value to insert
2
Enter value to insert
5
Enter value to insert
54
Enter value to insert
34
Enter value to insert
23
Enter value to insert
56
Enter value to insert
34
Enter value to insert
12
Enter value to insert
-1
Menu options:
```

```
-1
Menu options:
1. Insertion
2. Search
3. Deletion
4. Exit
2
Enter element to search
2
Index of element is 2
Menu options:
1. Insertion
2. Search
3. Deletion
4. Exit
2
Enter element to search
34
Index of element is 3
Menu options:
1. Insertion
2. Search
3. Deletion
4. Exit
3
Enter element to delete
54
Element not found
Menu options:
1. Insertion
2. Search
```

```
g++ exp9.cpp -o exp9 } ; I
Enter element to delete
54
Element not found
Menu options:
1. Insertion
2. Search
3. Deletion
4. Exit
3
Enter element to delete
23
Element deleted!
Menu options:
1. Insertion
2. Search
3. Deletion
4. Exit
4
Bye
```

Conclusion:-

Hashing is a technique for efficiently storing and retrieving data by mapping keys to specific indices within an array, known as a hash table. This method allows for fast access to data based on its corresponding key.

Post Lab Questions:

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

In linear hashing, when a collision is encountered, the element is pushed to the next available slot in the table. This effectively avoids two different elements having the same key.

The potential drawbacks of linear hashing include the problem of primary clustering, performance degradation with high load factors, and potentially inefficient storage utilization compared to some other methods

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

In quadratic hashing, we have two functions, h and h'. while H returns the modulo of the value to the size, h' returns the key using a special formula: $h(k)+i^2$. if this is also occupied, i increments till we find a free space.

This sequence is different from linear hashing as it doesn't map things sequentially but base it on a formula around the modulo operator. It helps keep the hash table free from clusters.

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

The function calling in quadratic hashing is a little tricker to implement. As we go on and practice the same code repeatedly, it gets getting clearer.

Another issue I encountered was the size of the hash table which I eventually had to hardcode. This is a constant value because we initialize the hashtable as a class variable in the very beginning. And a constant value cannot be taken as input before the program starts. In order to keep the code menu driven and readable, I had to hardcode the table size which is a bad practice.

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

Linear hashing is a better choice for situations where simplicity, memory efficiency, and good cache performance are critical. In practice, linear hashing is used to maintain commercial databases.

Quadratic hashing reduces clustering. Therefore, whenever our load factor is at a higher value, it's better to switch to quadratic hashing. For example, compilers and interpreters use symbol tables to store and quickly look up identifiers and their associated information.