Experiment No. 3

Aim:

Write a C program for telephone book database; consider the telephone book database of N clients. Make use of a hash table implementation to quickly look up a client's telephone number. Make use of linear probing, double hashing and quadratic collision handling techniques.

Objective:

- To understand and implement hash tables for efficient data retrieval.
- To store Hash keys by using collision techniques.
- Explain the concepts of Hashing and apply it to solve searching problems.
- To apply and compare different collision handling techniques: linear probing, quadratic probing, and double hashing.
- To provide quick insertion, search, and storage of client names and telephone numbers.

Problem Statement Write a program:

A telephone book database stores the contact details of N clients. Given the client's name, the system should quickly return the client's telephone number. Since direct addressing is inefficient for large data, we use hashing to index data. Collisions in hash tables occur when two keys hash to the same index. To resolve collisions, different probing techniques are used:

- **Linear Probing:** Check the next slot sequentially.
- **Quadratic Probing:** Check slots at increasing quadratic intervals.
- **Double Hashing:** Use a second hash function to calculate interval jumps.

Implement a hash table for storing clients' telephone numbers using these three collision resolution methods.

Theory:

Hashing:

Hashing is a technique to convert a given key (like a name or number) into an index in an array (called a hash table). This allows quick data retrieval because instead of searching the whole list, you directly jump to the position using the hash function.

Hash Table:

A data structure that maps keys (client names) to values (telephone numbers) using a hash function. The hash function converts the key into an index in the array.

Hash Function:

For simplicity, we will use a basic hash function that converts a string key into an integer index using modular arithmetic.

Collision Handling Techniques:

1. **Linear Probing:**
   - If a collision occurs at index h, try h+1, h+2, ..., h+i.
   - Simple but can lead to clustering.
2. **Quadratic Probing:**
   - If collision at h, try h + 1^2, h + 2^2, h + 3^2, ...
   - Helps reduce clustering.
3. **Double Hashing:**
   - Use two hash functions: h1 and h2.
   - On collision at h1(key), next index checked is (h1(key) + i * h2(key)) % table_size.
   - More effective in avoiding clustering.

Example:
Example: Linear Probing in Hashing

Scenario:

We have a hash table of size 7 to store integer keys:

Index	Content
0	
1	
2	
3	
4	

Index	Content
5	
6	

Hash Function:

hash(key) = key % 7

Insert keys: 10, 20, 15, 7, 32

Step-by-step insertion:

1. **Insert 10**

- hash(10) = 10 % 7 = 3
- Slot 3 is empty, insert 10 at index 3.

2. **Insert 20**

- hash(20) = 20 % 7 = 6
- Slot 6 is empty, insert 20 at index 6.

3. **Insert 15**

- hash(15) = 15 % 7 = 1
- Slot 1 is empty, insert 15 at index 1.

4. **Insert 7**

- hash(7) = 7 % 7 = 0
- Slot 0 is empty, insert 7 at index 0.

5. **Insert 32**

- hash(32) = 32 % 7 = 4
- Slot 4 is empty, insert 32 at index 4.

Now insert key 17

- hash(17) = 17 % 7 = 3
- Slot 3 is **occupied by 10** → collision!
- **Linear probing**: try next slot (3+1)%7 = 4

- Slot 4 is occupied by 32 → collision again.
- Try next slot (4+1)%7 = 5
- Slot 5 is empty → insert 17 at index 5.

Final Hash Table:

Index	**Key**
0	7
1	15
2	
3	10
4	32
5	17
6	20

When a collision occurs at an index, linear probing checks the next slot sequentially until it finds an empty slot. This technique is simple but can cause **primary clustering** where consecutive slots get filled.

Implementation of a Simple Hash Table:

A hash table is stored in an array that can be used to store data of any type. In this case, we will define a table .

```
#define TABLE_SIZE 7


int hashTable[TABLE_SIZE];


void initTable() {
   for (int i = 0; i < TABLE_SIZE; i++) {
      hashTable[i] = -1; // -1 indicates empty slot
   }
}
```

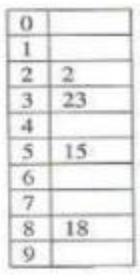
```
int hash(int key) {
    return key % TABLE_SIZE;
}


void insertLinear(int key) {
    int index = hash(key);
    int originalIndex = index;


    while (hashTable[index] != -1) { // while slot is occupied
        index = (index + 1) % TABLE_SIZE;
        if (index == originalIndex) {
            printf("Hash table is full\n");
            return;
        }
    }


    hashTable[index] = key;
    printf("Inserted key %d at index %d\n", key, index);
}
```

Collisions:

One problem with hashing is that it is possible that two strings can hash into the same location. This is called a collision. We can deal with collisions using many strategies, such as linear probing (looking for the next available location i+1, i+2, etc. from the hashed value i), quadratic probing (same as linear probing, except we look for available positions i+1 , i + 4, i + 9, etc from the hashed value i and separate chaining, the process of creating a linked list of values if they hashed into the same location.

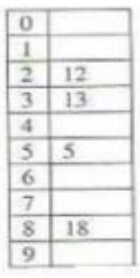
Example:

The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table?



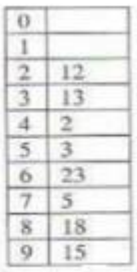
0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(A)



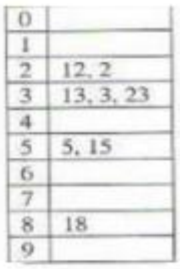
0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(B)



0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(C)



0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

(D)

(A) A

(B) B

(C) C

(D) D

Ans: (C)

Explanation: To get the idea of open addressing concept, you can go through below lines from .Open addressing, or closed hashing, is a method of collision resolution in hash tables. With this method a hash collision is resolved by probing, or searching through alternate locations in the array (the probe sequence) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. Well known probe sequences include:

1. linear probing in which the interval between probes is fixed–often at 1.

2. quadratic probing in which the interval between probes increases linearly (hence, the indices

are described by a quadratic function).

3. double hashing in which the interval between probes is fixed for each record but is

computed by another hash function.

Algorithm:

1: Create a hash table with maximum size.

2: Define Hash function.

3: Read the telephone no & user's information to insert it into hash table.

4: Insert telephone no value in the hash table.

5: If collision occurs, apply separate chaining method.

6: If user requires inserting another data, go to step4.

7: Display the all user's data.

8: Read the telephone no from user to be found.

9: find the data from hash table.

10: Display the data with minimum no. of comparisons.

Input:

- Number of clients, N
- For each client: Name (string), Telephone Number (string)

Note: Write input you are giving to your program

Output:

- Prompted insertion messages
- Search functionality to retrieve telephone number by client name
- Display of stored telephone book entries

Note: Write output of your program

Conclusion:

Hence we have studied successfully the use of a hash table & implementing it.