Introduction

In Hashing, we have seen that hashing takes a key as an input and returns us the memory address (index in hash-table) where the key is stored.

We also have discussed about a problem �.�.$i.e.$ *collision* which occurs when the hash function generates the same index for two different keys. To eliminate the difficulty we have seen about "Collision Resolution Techniques" viz. Open addressing and Closed addressing. Today we will briefly discuss **Open Addressing in hashing**.

The main concept of Open Addressing hashing is to keep all the data in the same hash table and hence a bigger Hash Table is needed. When using open addressing, a collision is resolved by probing (searching) alternative cells in the hash table until our target cell (empty cell while insertion, and cell with value �x while searching �x) is found. It is advisable to keep load factor (�α) below $0.50.5$, where �α is defined as �=�/�$\alpha=n/m$ where �n is the total number of entries in the hash table and �m is the size of the hash table. As explained above, since all the keys are stored in the same hash table so it's obvious that �$\leq 1 \alpha \leq 1$ because �$\leq$�$n \leq m$ always. If in case a collision happens then, alternative cells of the hash table are checked until the target cell is found. More formally,

- Cells $h0($�$),h1($�$),h2($�$)....h$�$($�$)h_0(x),h_1(x),h_2(x)....h_n(x)$ are tried consecutively until the target cell has been found in the hash table. Where h�$($�$)=(h$��$h($�$)+$�$($�$))\%$����$,h_i(x)=(hash(x)+f(i))\%Size$, keeping �$(0)=0f(0)=0$.
- The collision function �f is decided according to method resolution strategy.

There are three main Method Resolution Strategies --

1. Linear Probing
2. Quadratic Probing
3. Double Hashing

Linear Probing

In linear probing, collisions are resolved by searching the hash table consecutively (with wraparound) until an empty cell is found. The definition of collision function �f is quite simple in linear probing. As suggested by the name it is a linear function of �i or simply �$($�$)=$�$f(i)=i$. Operations in linear probing collision resolution technique -

- For inserting �x we search for the cells h��$h($�$)+0,h$��$h($�$)+1,...h$��$h($�$)+$�$hash(x)+0,hash(x)+1,...hash(x)+k$ until we found a empty cell to insert �x.

- For searching �x we again search for the cells h��h(�)+0,h��h(�)+1,...h��h(�)+�$hash(x)+0,hash(x)+1,...hash(x)+k$ until we found a cell with value �x. If we found a cell that has never been occupied it means �x is not present in the hash table.
- For deletion, we repeat the search process if a cell is found with value �x we replace the value �x with a predefined unique value (say $\infty\infty$) to denote that this cell has contained some value in past.

Example of linear probing -

Table Size = 77 Hash Function - h��h(���)=���%7$hash(key)=key\%7$ Collision Resoulution Strategy - �(�)=�$f(i)=i$

- Insert - $16,40,27,9,7516,40,27,9,75$
- Search - $75,2175,21$
- Delete - 4040

Steps involved are

- Step 1 - Make an empty hash table of size 7.

- Step 2 - Inserting 16,4016,40, and 2727.
  - ℎ��ℎ(16)=16%7=2ℎ𝑎𝑠ℎ(16)=16%7=2
  - ℎ��ℎ(40)=40%7=5ℎ𝑎𝑠ℎ(40)=40%7=5
  - ℎ��ℎ(27)=27%7=6ℎ𝑎𝑠ℎ(27)=27%7=6

As we do not get any collision we can easily insert values at their respective indexes generated by the hash function.
After inserting, the hash table will look like
-

- Step 3 - Inserting 9 and 75.
  - ℎ��ℎ(9)=9%7=2ℎ𝑎𝑠ℎ(9)=9%7=2 But at index 22 already 1616 is placed and hence collision occurs so as per linear probing we will search for consecutive cells till we find an empty cell.
    So we will probe for ℎ��ℎ(9)+1ℎ𝑎𝑠ℎ(9)+1 �.�.𝑖.𝑒. cell 3, since the next cell �.�. 3𝑖.𝑒. 3 is not occupied we place 9 in cell 33.
  - ℎ��ℎ(75)=75%7=5ℎ𝑎𝑠ℎ(75)=75%7=5 Again collision happens because 40 is already placed in cell 55. So will search for the consecutive cells, so we search for cell 66 which is also occupied then we will search for cell (ℎ��ℎ(75)+2)%7 �.�. 0(ℎ𝑎𝑠ℎ(75)+2)%7 𝑖.𝑒. 0 which is empty so we will place 75 there.

After inserting 99 and 7575 hash table will look like
-

- Step 4 - Search 7575 and 2121 -
  - *h*��*h*(75)=75%7=5*hash*(75)=75%7=5 But at index 5,755,75 is not present so we search for consecutive cells until we found an empty cell or a cell with a value of 7575. So we search in cell 66 but it does not contain 7575, so we search for 7575 in cell 00 and we stop our search here as we have found 7575.
  - *h*(21)=21%7=0*h*(21)=21%7=0 We will search for 2121 in cell 00 but it contains 75 so we will search in the next cell *h*��*h*(21)+1,*hash*(21)+1, �.�. 1*i.e.* 1 since it is found empty it is clear that 2121 do not exist in our table.
- Step 5 - Delete 4040
  - *h*(40)=40%7=5*h*(40)=40%7=5 Firstly we search for 4040 which results in a successful search as we get 4040 in cell 55 then we will remove 4040 from cell 55 and replace it with a unique value (say -ash∞∞).

After all these operations our hash table will look like
-

Algorithm of linear probing

- **Insert(�x) -**
    - Find the hash value, �k of �x from the hash function h��h(�)$hash(x)$.
    - Iterate consecutively in the table starting from the �,k, till you find a cell that is currently not occupied.
    - Place �x in that cell.
- **Search(�x) -**
    - Find the hash value, �k of �x from the hash function h��h(�)$hash(x)$.
    - Iterate consecutively in the table starting from the �,k, till you find a cell that contains �x or which is never been occupied.
    - If we found �x, then the search is successful and unsuccessful in the other case.
- **Delete(�x) -**
    - Repeat the steps of Search(�x).
    - If element �x does not exist in the table then we can't delete it.
    - If �x exists in the cell (say �k), put $\infty\infty$ in cell k to denote it has been occupied some time in the past, but now it is empty.

Pesudocode of Linear Probing

```
class Hashing:
    size, table[]
    Hash(x):
        return x%size

    Insert(x):
        k=Hash(x)
        while(table[k] is not empty):
            k=(k+1)%size
        table[k]=x

    Search(x):
        k=Hash(x)
        while(table[k] != x):
            if(table[k] has never been occupied):
                return false
            k=(k+1)%size
        return table[k]==x

    Delete(x):
        k=Hash(x)
        while(table[k]!=x):
            if(table[k] has never been occupied):
                return
            k=(k+1)%size
        if(table[k]==x):
            table[k] = -Infinity
```

Code of Linear Probing

- C/C++

```
#include<bits/stdc++.h>
using namespace std;
class Hashing{
    // Declaring Table and size.
    int *table;
    int size;
public:
    // Constructor
    Hashing(int Size){
        // Initializing size.
        size=Size;
        // Allocating memory to the table.
        table=new int[size];
        // Initializing all values of
        // table with minimum possible
        // value integer can hold.
        for(int i=0;i<size;i++)
            table[i]=INT_MIN;
    }
    // Hash Function
    int hash(int x){
        // returning value of modulus
        // of x taken with table size.
        return x%size;
    }
    // Insert function
    void insert(int x){
        // Finding the hash value of x.
        int k=hash(x);
        // Iterating till we find a cell
        // that is not occupied currently.
        while(table[k]!=INT_MIN&&table[k]!=INT_MAX)
            k=(k+1)%size;
        // Assigning x to cell k.
        table[k]=x;
    }
    // Search function
    bool search(int x){
        // Finding the hash value of x.
        int k=hash(x);
        // Iterating till we find a cell
        // containing x.
        while(table[k]!=x){
            // If the cell has never been
            // occupied we return false.
            if(table[k]==INT_MIN)
                return false;
            k=(k+1)%size;
        }
        // Checking if table[k] is x or not.
        return table[k]==x;
    }
    void Delete(int x){
        // Finding the hash value of x.
        int k=hash(x);
        // Iterating till we find a cell
        // containing x.
        while(table[k]!=x)
        {
            // If the cell has never been
            // occupied we return false.
            if(table[k]==INT_MIN)
                return;
            k=(k+1)%size;
        }
        // If x exists in table replacing
        // its value with a very large value.
        if(table[k]==x)
            table[k]=INT_MAX;
    }
};
int main(){
    Hashing h(7);
    h.insert(16);
    h.insert(40);
    h.insert(27);
    h.insert(9);
    h.insert(75);

    if(h.search(75))
        cout<<"75 found"<<endl;
    if(h.search(40))
        cout<<"40 found"<<endl;

    h.Delete(40);
    if(!h.search(40))
        cout<<"After deleting 40, 40 is not found";
    return 0;
}
```

- Java

```
import java.util.*;
class Hashing{
    // Declaring Table and size.
    int table[];
    int size;
    // Constructor
    Hashing(int size){
        // Initializing size.
        this.size=size;
        // Allocating memory to the table.
        table=new int[size];
        // Initializing all values of
        // table with minimum possible
        // value integer can hold.
        for(int i=0;i<size;i++)
            table[i]=Integer.MIN_VALUE;
    }
    // Hash Function
    int hash(int x){
        // returning value of modulus
        // of x taken with table size.
        return x%size;
    }
    // Insert function
    void insert(int x){
        // Finding the hash value of x.
        int k=hash(x);
```

```java
        // Iterating till we find a cell
        // that is not occupied currently.
        while(table[k]!=Integer.MIN_VALUE&&table[k]!=Integer.MAX_VALUE)
            k=(k+1)%size;
        // Assigning x to cell k.
        table[k]=x;
    }
    // Search function
    boolean search(int x){
        // Finding the hash value of x.
        int k=hash(x);
        // Iterating till we find a cell
        // containing x.
        while(table[k]!=x){
            // If the cell has never been
            // occupied we return false.
            if(table[k]==Integer.MIN_VALUE)
                return false;
            k=(k+1)%size;
        }
        // Checking if table[k] is x or not.
        return table[k]==x;
    }
    void delete(int x){
        // Finding the hash value of x.
        int k=hash(x);
        // Iterating till we find a cell
        // containing x.
        while(table[k]!=x)
        {
            // If the cell has never been
            // occupied we return false.
            if(table[k]==Integer.MIN_VALUE)
                return;
            k=(k+1)%size;
        }
        // If x exists in table replacing
        // its value with a very large value.
        if(table[k]==x)
            table[k]=Integer.MAX_VALUE;
    }
    public static void main(String args[]){
        Hashing h=new Hashing(7);
        h.insert(16);
        h.insert(40);
        h.insert(27);
        h.insert(9);
        h.insert(75);

        if(h.search(75))
            System.out.println("75 found");
        if(h.search(40))
            System.out.println("40 found");

        h.delete(40);
        if(!h.search(40))
            System.out.println("After deleting 40, 40 is not found");
    }
}
```

Output -

```
75 found
40 found
After deleting 40, 40 is not found
```

Problem With Linear Probing

Even though linear probing is intuitive and easy to implement but it suffers from a problem known as *Primary Clustering*. It occurs because the table is large enough therefore time to get an empty cell or to search for a key k is quite large. This happens mainly because consecutive elements form a group and then it takes a lot of time to find an element or an empty cell which ultimately makes the worst case time complexity of *searching*, *insertion* and *deletion* operations to be $O(n)$, where n is the size of the table.

Quadratic Probing

Quadratic probing eliminates the problem of "Primary Clustering" that occurs in Linear probing techniques. The working of quadratic probing involves taking the initial hash value and probing in the hash table by adding successive values of an arbitrary quadratic polynomial. As suggested by its name, quadratic probing uses a quadratic collision function f. One of the most common and reasonable choices for f is -

- $f(i)=i^2$Operations in quadratic probing collision resolution strategy are -

- For inserting x we search for the cells $hash(x)+0, hash(x)+1^2, hash(x)+2^2, \ldots$ until we find an empty cell to insert x.
- For searching x we again search for the cells $hash(x)+0, hash(x)+1^2, hash(x)+2^2, \ldots$ until we find a cell with value x. If we find an empty cell that has never been occupied it means x is not present in the hash table.
- For deletion, we repeat the search process if a cell is found with value x we replace the value x with a predefined unique value to denote that this cell has contained some value in past.

You can see that the only one change between linear and quadratic probing is that in case of collision we are not searching in cells consecutively, rather we are interested in probing the cells quadratically. Let us understand this by an example -

Example of Quadratic Probing

Table Size = 7 **Insert** = 15, 23, and 85. **Search & Delete** = 85 **Hash Function** - $Hash(x)=x\%7$ **Collision Resolution Strategy** - $f(i)=i^2$

- Step 1 - Create a table of size 77.

- Step 2 - Insert 1515 and 2323
    - *h*��*h*(15)=15%7=1*hash*(15)=15%7=1 Since the cell at index 1 is not occupied we can easily insert 15 at cell 1.
    - *h*��*h*(23)=23%7=2*hash*(23)=23%7=2 Again cell 2 is not occupied so place 23 in cell 2. After performing this step our hash table will look like
      -

- Step 3 - Inserting 8585
    - *h*��*h*(85)=85%7=1*hash*(85)=85%7=1 In our hash table cell 1 is already occupied so we will search for cell 1+12, �.�.1+12, *i.e.* cell 22. Again it is found occupied so we will search for cell 1+22, �.�.1+22, *i.e.* cell 55. It is not occupied so we will place 8585 in cell 55. After performing all these 33 insertions in our hash table it will look like

-

- Step 4 - Search and delete 8585 We will go through the same steps as in inserting 85 and when we find 85 our search is successful and to delete it we will replace it with some other unique value a good choice is to replace it with ∞∞.

Now as there is not much change in the approach of quadratic probing and linear probing. We are skipping algorithm and pseudocode of quadratic probing and directly jumping to its code.

Example of linear probing -

Table Size = 77 Hash Function - *h*��*h*(���)=���%7*hash*(*key*)=*key*%7 Collision Resoulution Strategy - �(�)=�*f*(*i*)=*i*

- Insert - 16,40,27,9,7516,40,27,9,75
- Search - 75,2175,21
- Delete - 4040

Steps involved are

- Step 1 - Make an empty hash table of size 7.

- Step 2 - Inserting 16,4016,40, and 2727.
    - *h*��*h*(16)=16%7=2*hash*(16)=16%7=2

- h��$h(40)=40\%7=5hash(40)=40\%7=5$
- h��$h(27)=27\%7=6hash(27)=27\%7=6$

As we do not get any collision we can easily insert values at their respective indexes generated by the hash function.
After inserting, the hash table will look like
-

- Step 3 - Inserting 9 and 75.
  - h��$h(9)=9\%7=2hash(9)=9\%7=2$ But at index 22 already 1616 is placed and hence collision occurs so as per linear probing we will search for consecutive cells till we find an empty cell.
    So we will probe for h��$h(9)+1hash(9)+1$ �.�.$i.e.$ cell 3, since the next cell �.�. $3i.e.\ 3$ is not occupied we place 9 in cell 33.
  - h��$h(75)=75\%7=5hash(75)=75\%7=5$ Again collision happens because 40 is already placed in cell 55. So will search for the consecutive cells, so we search for cell 66 which is also occupied then we will search for cell $(h$��$h(75)+2)\%7$ �.�. $0(hash(75)+2)\%7\ i.e.\ 0$ which is empty so we will place 75 there.

After inserting 99 and 7575 hash table will look like
-

- Step 4 - Search 7575 and 2121 -

    - h��h(75)=75%7=5hash(75)=75%7=5 But at index 5,755,75 is not present so we search for consecutive cells until we found an empty cell or a cell with a value of 7575. So we search in cell 66 but it does not contain 7575, so we search for 7575 in cell 00 and we stop our search here as we have found 7575.
    - h(21)=21%7=0h(21)=21%7=0 We will search for 2121 in cell 00 but it contains 75 so we will search in the next cell h��h(21)+1,hash(21)+1, �.�. 1i.e. 1 since it is found empty it is clear that 2121 do not exist in our table.
- Step 5 - Delete 4040
    - h(40)=40%7=5h(40)=40%7=5 Firstly we search for 4040 which results in a successful search as we get 4040 in cell 55 then we will remove 4040 from cell 55 and replace it with a unique value (say -ash∞∞).

After all these operations our hash table will look like
-

Algorithm of linear probing

- **Insert(�x) -**
    - Find the hash value, �k of �x from the hash function h��h(�)hash(x).
    - Iterate consecutively in the table starting from the �,k, till you find a cell that is currently not occupied.
    - Place �x in that cell.
- **Search(�x) -**
    - Find the hash value, �k of �x from the hash function h��h(�)hash(x).
    - Iterate consecutively in the table starting from the �,k, till you find a cell that contains �x or which is never been occupied.

    - If we found �x, then the search is successful and unsuccessful in the other case.
- **Delete(�x) -**
    - Repeat the steps of Search(�x).
    - If element �x does not exist in the table then we can't delete it.
    - If �x exists in the cell (say �k), put $\infty\infty$ in cell k to denote it has been occupied some time in the past, but now it is empty.

Pesudocode of Linear Probing

```
class Hashing:
    size, table[]
    Hash(x):
        return x%size

    Insert(x):
        k=Hash(x)
        while(table[k] is not empty):
            k=(k+1)%size
        table[k]=x

    Search(x):
        k=Hash(x)
        while(table[k] != x):
            if(table[k] has never been occupied):
                return false
            k=(k+1)%size
        return table[k]==x

    Delete(x):
        k=Hash(x)
        while(table[k]!=x):
            if(table[k] has never been occupied):
                return
            k=(k+1)%size
        if(table[k]==x):
            table[k] = -Infinity
```

Code of Linear Probing

- C/C++

```
#include<bits/stdc++.h>
using namespace std;
class Hashing{
    // Declaring Table and size.
    int *table;
    int size;
public:
    // Constructor
    Hashing(int Size){
        // Initializing size.
        size=Size;
        // Allocating memory to the table.
        table=new int[size];
        // Initializing all values of
        // table with minimum possible
        // value integer can hold.
        for(int i=0;i<size;i++)
            table[i]=INT_MIN;
    }
    // Hash Function
    int hash(int x){
        // returning value of modulus
        // of x taken with table size.
        return x%size;
    }
    // Insert function
    void insert(int x){
        // Finding the hash value of x.
        int k=hash(x);
        // Iterating till we find a cell
        // that is not occupied currently.
        while(table[k]!=INT_MIN&&table[k]!=INT_MAX)
            k=(k+1)%size;
        // Assigning x to cell k.
        table[k]=x;
    }
    // Search function
    bool search(int x){
        // Finding the hash value of x.
        int k=hash(x);
        // Iterating till we find a cell
        // containing x.
        while(table[k]!=x){
            // If the cell has never been
            // occupied we return false.
            if(table[k]==INT_MIN)
                return false;
            k=(k+1)%size;
        }
        // Checking if table[k] is x or not.
        return table[k]==x;
    }
    void Delete(int x){
        // Finding the hash value of x.
        int k=hash(x);
        // Iterating till we find a cell
        // containing x.
        while(table[k]!=x)
        {
            // If the cell has never been
            // occupied we return false.
            if(table[k]==INT_MIN)
                return;
            k=(k+1)%size;
        }
        // If x exists in table replacing
        // its value with a very large value.
        if(table[k]==x)
            table[k]=INT_MAX;
    }
};
int main(){
    Hashing h(7);
    h.insert(16);
    h.insert(40);
    h.insert(27);
    h.insert(9);
    h.insert(75);

    if(h.search(75))
        cout<<"75 found"<<endl;
    if(h.search(40))
        cout<<"40 found"<<endl;

    h.Delete(40);
    if(!h.search(40))
        cout<<"After deleting 40, 40 is not found";
    return 0;
}
```