Hashing
Hashing

• Linear search has a running time proportional to $O(n)$, while binary search takes time proportional to $O(\log n)$, where $n$ is the number of elements in the array.

• Binary search and binary search trees are efficient algorithms to search for an element. But what if we want to perform the search operation in time proportional to $O(1)$?

• In other words, is there a way to search an array in constant time, irrespective of its size?
<table>
<thead>
<tr>
<th>Key</th>
<th>Array of Employees’ Records</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key 0</td>
<td>Employee record with Emp_ID 0</td>
</tr>
<tr>
<td>Key 1</td>
<td>Employee record with Emp_ID 1</td>
</tr>
<tr>
<td>Key 2</td>
<td>Employee record with Emp_ID 2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Key 98</td>
<td>Employee record with Emp_ID 98</td>
</tr>
<tr>
<td>Key 99</td>
<td>Employee record with Emp_ID 99</td>
</tr>
<tr>
<td>Key</td>
<td>Array of Employees’ Records</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>Key 00000 → [0]</td>
<td>Employee record with Emp_ID 00000</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Key n → [n]</td>
<td>Employee record with Emp_ID n</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Key 99998 → [99998]</td>
<td>Employee record with Emp_ID 99998</td>
</tr>
<tr>
<td>Key 99999 → [99999]</td>
<td>Employee record with Emp_ID 99999</td>
</tr>
</tbody>
</table>
In order to keep the array size down to the size that we will actually be using (100 elements), another good option is to use just the last two digits of the key to identify each employee. For example, the employee with `Emp_ID 79439` will be stored in the element of the array with index 39. Similarly, the employee with `Emp_ID 12345` will have his record stored in the array at the 45th location.

In the second solution, the elements are not stored according to the value of the key. So in this case, we need a way to convert a five-digit key number to a two-digit array index. We need a function which will do the transformation.

In this case, we will use the term `hash table for an array` and the function that will carry out the transformation will be called a `hash function`. 
Hash table is a data structure in which keys are mapped to array positions by a hash function.

In the example we discussed, we will use a hash function that extracts the last two digits of the key. Therefore, we map the keys to array locations or array indices.

A value stored in a hash table can be searched in O(1) time by using a hash function which generates an address from the key (by producing the index of the array where the value is stored).
Figure shows a direct correspondence between the keys and the indices of the array. This concept is useful when the total universe of keys is small and when most of the keys are actually used from the whole set of keys. This is equivalent to our first example, where there are 100 keys for 100 employees.
In a hash table, an element with key k is stored at index h(k) and not k. It means a hash function h is used to calculate the index at which the element with key k will be stored. **This process of mapping the keys to appropriate locations (or indices) in a hash table is called hashing.**
Figure shows a hash table in which each key from the set $K$ is mapped to locations generated by using a hash function. Note that keys $k_2$ and $k_6$ point to the same memory location. **This is known as collision.**

*That is, when two or more keys map to the same memory location, a collision is said to occur.* Similarly, keys $k_5$ and $k_7$ also collide. The main goal of using a hash function is to reduce the range of array indices that have to be handled. Thus, instead of having $U$ values, we just need $K$ values, thereby reducing the amount of storage space required.
HASH FUNCTIONS

• **A hash function** is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table.

• The main aim of a hash function is that elements should be relatively, randomly, and uniformly distributed.

• It produces a unique set of integers within some suitable range in order to reduce the number of collisions. In practice, there is no hash function that eliminates collisions completely.

• A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array.
HASH FUNCTIONS

Properties of a Good Hash Function

Low cost: The cost of executing a hash function must be small, so that using the hashing technique becomes preferable over other approaches.

Determinism: A hash procedure must be deterministic. This means that the same hash value must be generated for a given input value.

Uniformity: A good hash function must map the keys as evenly as possible over its output range. This means that the probability of generating every hash value in the output range should roughly be the same. The property of uniformity also minimizes the number of collisions.
HASH FUNCTIONS

DIFFERENT HASH FUNCTIONS

• Division Method
• Multiplication Method
• Mid-Square Method
• Folding Method

We will see the hash functions which use numeric keys. However, there can be cases in real-world applications where we can have alphanumeric keys rather than simple numeric keys. In such cases, the ASCII value of the character can be used to transform it into its equivalent numeric key. Once this transformation is done, any of the hash functions given above can be applied to generate the hash value.
HASH FUNCTIONS

DIFFERENT HASH FUNCTIONS

1. Division Method

It is the most simple method of hashing an integer x. This method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as

\[ h(x) = x \mod M \]

The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for M.
Division Method

The division method is extremely simple to implement. The following code segment illustrates how to do this:

```c
def(h(x))
{ return (x % M); }
```

A potential drawback of the division method is that while using this method, consecutive keys map to consecutive hash values. On one hand, this is good as it ensures that consecutive keys do not collide, but on the other, it also means that consecutive array locations will be occupied. This may lead to degradation in performance.
HASH FUNCTIONS

DIFFERENT HASH FUNCTIONS

• Division Method

Example: Calculate the hash values of keys 1234 and 5462.

Solution: Setting $M = 97$, hash values can be calculated as:

\[
\begin{align*}
    h(1234) &= 1234 \mod 97 = 70 \\
    h(5642) &= 5642 \mod 97 = 16
\end{align*}
\]
2. Multiplication Method

The steps involved in the multiplication method are as follows:

**Step 1:** Choose a constant $A$ such that $0 < A < 1$.

**Step 2:** Multiply the key $k$ by $A$.

**Step 3:** Extract the fractional part of $kA$.

**Step 4:** Multiply the result of Step 3 by the size of hash table ($m$).

Hence, the hash function can be given as:

$$h(k) = m \times (kA \mod 1)$$

where $(kA \mod 1)$ gives the fractional part of $kA$ and $m$ is the total number of indices in the hash table.
HASH FUNCTIONS

DIFFERENT HASH FUNCTIONS

• Multiplication Method

The greatest advantage of this method is that it works practically with any value of A. Although the algorithm works better with some values, the optimal choice depends on the characteristics of the data being hashed.

Knuth has suggested that the best choice of A is

" (sqrt5 – 1) /2 = 0.6180339887"
HASH FUNCTIONS

DIFFERENT HASH FUNCTIONS

• Multiplication Method

Example: Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table.

Solution: We will use \( A = 0.618033, \ m = 1000, \ \text{and} \ k = 12345 \)

\[
\begin{align*}
    h(12345) &= 1000 (12345 \times 0.618033 \mod 1) \\
    h(12345) &= 1000 (7629.617385 \mod 1) \\
    h(12345) &= 1000 (0.617385) \\
    h(12345) &= 617.385 \\
    h(12345) &= 617
\end{align*}
\]
3. Mid-Square Method

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find \( k^2 \).

Step 2: Extract the middle \( r \) digits of the result obtained in Step 1.

The algorithm works well because most or all digits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.
HASH FUNCTIONS

DIFFERENT HASH FUNCTIONS

• Mid-Square Method

In the mid-square method, the same \( r \) digits must be chosen from all the keys. Therefore, the hash function can be given as:

\[ h(k) = s \]

where \( s \) is obtained by selecting \( r \) digits from \( k^2 \).
HASH FUNCTIONS

DIFFERENT HASH FUNCTIONS

• Mid-Square Method

Example: Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.

Solution: Note that the hash table has 100 memory locations whose indices vary from 0 to 99.

This means that only two digits are needed to map the key to a location in the hash table, so r = 2.

When k = 1234, k^2 = 1522756, h (1234) = 27
When k = 5642, k^2 = 31832164, h (5642) = 21
Observe that the 3rd and 4th digits starting from the right are chosen.

Prepared By: Kaushik Vaghani
4. Folding Method

The folding method works in the following two steps:

*Step 1: Divide the key value into a number of parts. That is, divide $k$ into parts $k_1$, $k_2$, ..., $k_n$, where each part has the same number of digits except the last part which may have lesser digits than the other parts.*

*Step 2: Add the individual parts. That is, obtain the sum of $k_1 + k_2 + ... + k_n$. The hash value is produced by ignoring the last carry, if any.*
HASH FUNCTIONS

DIFFERENT HASH FUNCTIONS

• Folding Method

The folding method works in the following two steps:

Note that the number of digits in each part of the key will vary depending upon the size of the hash table.

For example, if the hash table has a size of 1000, then there are 1000 locations in the hash table. To address these 1000 locations, we need at least three digits; therefore, each part of the key must have three digits except the last part which may have lesser digits.
HASH FUNCTIONS

DIFFERENT HASH FUNCTIONS

• Folding Method

Example: Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.

Solution:
Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

<table>
<thead>
<tr>
<th>key</th>
<th>5678</th>
<th>321</th>
<th>34567</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parts</td>
<td>56 and 78</td>
<td>32 and 1</td>
<td>34, 56 and 7</td>
</tr>
<tr>
<td>Sum</td>
<td>134</td>
<td>33</td>
<td>97</td>
</tr>
<tr>
<td>Hash value</td>
<td>34 (ignore the last carry)</td>
<td>33</td>
<td>97</td>
</tr>
</tbody>
</table>
• collisions occur when the hash function maps two different keys to the same location.

• Obviously, two records cannot be stored in the same location.

• Therefore, a method used to solve the problem of collision, also called collision resolution technique, is applied.

• The two most popular methods of resolving collisions are:

  1. Open addressing

  2. Chaining
Collision Resolution by Open Addressing

Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position.

In this technique, all the values are stored in the hash table.

The hash table contains two types of values: sentinel values (e.g., −1) and data values.

The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value.
Collision Resolution by Open Addressing

When a key is mapped to a particular memory location, then the value it holds is checked.

If it contains a sentinel value, then the location is free and the data value can be stored in it. However, if the location already has some data value stored in it, then other slots are examined systematically in the forward direction to find a free slot.

If even a single free location is not found, then we have an OVERFLOW condition.

The process of examining memory locations in the hash table is called *probing*. *Open addressing* technique can be implemented using 1)linear probing (primary clustering), 2)quadratic probing (secondary clustering), 3)double hashing, and 4)rehashing.
COLLISIONS

Collision Resolution by Open Addressing: 1) Linear Probing

The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by \( h(k) \), then the following hash function is used to resolve the collision:

\[
h(k, i) = [h'(k) + i] \mod m
\]

Where \( m \) is the size of the hash table, \( h'(k) = (k \mod m) \), and \( i \) is the probe number that varies from 0 to \( m-1 \).

Therefore, for a given key \( k \), first the location generated by \( [h'(k) \mod m] \) is probed because for the first time \( i=0 \). If the location is free, the value is stored in it, else the second probe generates the address of the location given by \( [h'(k) + 1] \mod m \). Similarly, if the location is occupied, then subsequent probes generate the address as \( [h'(k) + 2] \mod m \), \( [h'(k) + 3] \mod m \), \( [h'(k) + 4] \mod m \), \( [h'(k) + 5] \mod m \), and so on, until a free location is found.
COLLISIONS

Collision Resolution by Open Addressing: Linear Probing

Example:
Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.

Solution:
Let $h'(k) = k \mod m$, $m = 10$
Initially, the hash table can be given as:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Step 1   Key = 72
$h(72, 0) = (72 \mod 10 + 0) \mod 10$
= $(2) \mod 10$
= 2
Since $T[2]$ is vacant, insert key 72 at this location.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Collision Resolution by Open Addressing: Linear Probing

Step 2  Key = 27

\[ h(27, 0) = (27 \mod 10 + 0) \mod 10 \]
\[ = (7) \mod 10 \]
\[ = 7 \]

Since \(T[7]\) is vacant, insert key 27 at this location.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Step 3  Key = 36

\[ h(36, 0) = (36 \mod 10 + 0) \mod 10 \]
\[ = (6) \mod 10 \]
\[ = 6 \]

Since \(T[6]\) is vacant, insert key 36 at this location.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>36</td>
<td>27</td>
<td>-1</td>
</tr>
</tbody>
</table>
Collision Resolution by Open Addressing: Linear Probing

**Step 4**  
Key = 24  
\[ h(24, 0) = (24 \mod 10 + 0) \mod 10 \]  
= (4) \mod 10  
= 4  

Since \( T[4] \) is vacant, insert key 24 at this location.

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>-1</td>
<td>24</td>
<td>-1</td>
<td>36</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
```

**Step 5**  
Key = 63  
\[ h(63, 0) = (63 \mod 10 + 0) \mod 10 \]  
= (3) \mod 10  
= 3  

Since \( T[3] \) is vacant, insert key 63 at this location.

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>63</td>
<td>24</td>
<td>-1</td>
<td>36</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
```
Step 6

Key = 81
\[ h(81, 0) = (81 \mod 10 + 0) \mod 10 \]
\[ = (1) \mod 10 \]
\[ = 1 \]

Since \( \tau[1] \) is vacant, insert key 81 at this location.

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
0 & 81 & 72 & 63 & 24 & -1 & 36 & 27 & -1 & -1 \\
\end{array}
\]

Step 7

Key = 92
\[ h(92, 0) = (92 \mod 10 + 0) \mod 10 \]
\[ = (2) \mod 10 \]
\[ = 2 \]

Now \( \tau[2] \) is occupied, so we cannot store the key 92 in \( \tau[2] \). Therefore, try again for the next location. Thus probe, \( i = 1 \), this time.

Key = 92
\[ h(92, 1) = (92 \mod 10 + 1) \mod 10 \]
\[ = (2 + 1) \mod 10 \]
\[ = 3 \]

Now \( \tau[3] \) is occupied, so we cannot store the key 92 in \( \tau[3] \). Therefore, try again for the next location. Thus probe, \( i = 2 \), this time.
COLLISIONS

Key = 92
\[ h(92, 2) = (92 \mod 10 + 2) \mod 10 \]
\[ = (2 + 2) \mod 10 \]
\[ = 4 \]

Now \( t[4] \) is occupied, so we cannot store the key 92 in \( t[4] \). Therefore, try again for the next location. Thus probe, \( i = 3 \), this time.

Key = 92
\[ h(92, 3) = (92 \mod 10 + 3) \mod 10 \]
\[ = (2 + 3) \mod 10 \]
\[ = 5 \]

Since \( t[5] \) is vacant, insert key 92 at this location.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>81</td>
<td>72</td>
<td>63</td>
<td>24</td>
<td>92</td>
<td>36</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Step 8  Key = 101
\[ h(101, 0) = (101 \mod 10 + 0) \mod 10 \]
\[ = (1) \mod 10 \]
\[ = 1 \]

Now \( \tau[1] \) is occupied, so we cannot store the key 101 in \( \tau[1] \). Therefore, try again for the next location. Thus probe, \( i = 1 \), this time.

Key = 101
\[ h(101, 1) = (101 \mod 10 + 1) \mod 10 \]
\[ = (1 + 1) \mod 10 \]
\[ = 2 \]

\( \tau[2] \) is also occupied, so we cannot store the key in this location. The procedure will be repeated until the hash function generates the address of location 8 which is vacant and can be used to store the value in it.
COLLISIONS

Collision Resolution by Open Addressing: Linear Probing

**Pros and Cons**

Linear probing finds an empty location by doing a linear search in the array beginning from position $h(k)$. Although the algorithm provides good memory caching through good locality of reference, the drawback of this algorithm is that it results in clustering, and thus there is a higher risk of more collisions where one collision has already taken place. The performance of linear probing is sensitive to the distribution of input values.

As the hash table fills, clusters of consecutive cells are formed and the time required for a search increases with the size of the cluster. In addition to this, when a new value has to be inserted into the table at a position which is already occupied, that value is inserted at the end of the cluster, which again increases the length of the cluster. Generally, an insertion is made between two clusters that are separated by one vacant location. But with linear probing, there are more chances that subsequent insertions will also end up in one of the clusters, thereby potentially increasing the cluster length by an amount much greater than one. More the number of collisions, higher the probes that are required to find a free location and lesser is the performance. This phenomenon is called **primary clustering**. To avoid primary clustering, other techniques such as quadratic probing and double hashing are used.
Collision Resolution by Open Addressing: 2) Quadratic Probing

In this technique, if a value is already stored at a location generated by \( h(k) \), then the following hash function is used to resolve the collision:

\[
h(k, i) = [h'(k) + c_1 i + c_2 i^2] \mod m
\]

where \( m \) is the size of the hash table, \( h'(k) = (k \mod m) \), \( i \) is the probe number that varies from 0 to \( m-1 \), and \( c_1 \) and \( c_2 \) are constants such that \( c_1 \) and \( c_2 \neq 0 \).

Quadratic probing eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search.
Example: Consider a hash table of size 10. Using **quadratic probing**, insert the keys 72, 27, 36, 24, 63, 81, and 101 into the table. Take $c_1 = 1$ and $c_2 = 3$.

**Solution**

Let $h'(k) = k \mod m$, $m = 10$

Initially, the hash table can be given as:

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
-1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
\end{array}
\]

We have,

\[
h(k, i) = [h'(k) + c_1 i + c_2 i^2] \mod m
\]

**Step 1**

Key = 72

\[
h(72, 0) = [72 \mod 10 + 1 \times 0 + 3 \times 0] \mod 10
\]

\[
= [72 \mod 10] \mod 10
\]

\[
= 2 \mod 10
\]

\[
= 2
\]

Since $τ[2]$ is vacant, insert the key 72 in $τ[2]$. The hash table now becomes:

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
-1 & -1 & 72 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
\end{array}
\]
COLLISIONS

Step 2  
Key = 27

\[ h(27, 0) = [27 \mod 10 + 1 \times 0 + 3 \times 0] \mod 10 \]
\[ = [27 \mod 10] \mod 10 \]
\[ = 7 \mod 10 \]
\[ = 7 \]

Since \( \tau[7] \) is vacant, insert the key 27 in \( \tau[7] \). The hash table now becomes:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Step 3  
Key = 36

\[ h(36, 0) = [36 \mod 10 + 1 \times 0 + 3 \times 0] \mod 10 \]
\[ = [36 \mod 10] \mod 10 \]
\[ = 6 \mod 10 \]
\[ = 6 \]

Since \( \tau[6] \) is vacant, insert the key 36 in \( \tau[6] \). The hash table now becomes:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>36</td>
<td>27</td>
<td>-1</td>
</tr>
</tbody>
</table>
** COLLISIONS **

**Step 4**

Key = 24

\[ h(24, 0) = [24 \mod 10 + 1 \times 0 + 3 \times 0] \mod 10 \]
\[ = [24 \mod 10] \mod 10 \]
\[ = 4 \mod 10 \]
\[ = 4 \]

Since \( \tau[4] \) is vacant, insert the key 24 in \( \tau[4] \). The hash table now becomes:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>-1</td>
<td>24</td>
<td>-1</td>
<td>36</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

**Step 5**

Key = 63

\[ h(63, 0) = [63 \mod 10 + 1 \times 0 + 3 \times 0] \mod 10 \]
\[ = [63 \mod 10] \mod 10 \]
\[ = 3 \mod 10 \]
\[ = 3 \]

Since \( \tau[3] \) is vacant, insert the key 63 in \( \tau[3] \). The hash table now becomes:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>63</td>
<td>24</td>
<td>-1</td>
<td>36</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
COLLISIONS

Step 6  
Key = 81  
\[ h(81, 0) = [81 \mod 10 + 1 \times 0 + 3 \times 0] \mod 10 \]
\[ = [81 \mod 10] \mod 10 \]
\[ = 81 \mod 10 \]
\[ = 1 \]

Since \( \tau[1] \) is vacant, insert the key 81 in \( \tau[1] \). The hash table now becomes:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>81</td>
<td>72</td>
<td>63</td>
<td>24</td>
<td>-1</td>
<td>36</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Step 7  
Key = 101  
\[ h(101, 0) = [101 \mod 10 + 1 \times 0 + 3 \times 0] \mod 10 \]
\[ = [101 \mod 10 + 0] \mod 10 \]
\[ = 1 \mod 10 \]
\[ = 1 \]

Since \( \tau[1] \) is already occupied, the key 101 cannot be stored in \( \tau[1] \). Therefore, try again for next location. Thus probe, \( i = 1 \), this time.

Key = 101  
\[ h(101, 0) = [101 \mod 10 + 1 \times 1 + 3 \times 1] \mod 10 \]
COLLISIONS

\[= [101 \mod 10 + 1 + 3] \mod 10\]
\[= [101 \mod 10 + 4] \mod 10\]
\[= [1 + 4] \mod 10\]
\[= 5 \mod 10\]
\[= 5\]

Since \(τ[5]\) is vacant, insert the key 101 in \(τ[5]\). The hash table now becomes:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>81</td>
<td>72</td>
<td>63</td>
<td>24</td>
<td>101</td>
<td>36</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
COLLISIONS

Although quadratic probing is free from primary clustering, it is still liable to what is known as secondary clustering. It means that if there is a collision between two keys, then the same probe sequence will be followed for both. With quadratic probing, the probability for multiple collisions increases as the table becomes full. This situation is usually encountered when the hash table is more than full.

Quadratic probing is widely applied in the Berkeley Fast File System to allocate free blocks.
COLLISIONS

Collision Resolution by Open Addressing: 3) Double Hashing

Double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function, hence the name double hashing. In double hashing, we use two hash functions rather than a single function. The hash function in the case of double hashing can be given as:

\[ h(k, i) = [h_1(k) + ih_2(k)] \mod m \]

where \( m \) is the size of the hash table, \( h_1(k) \) and \( h_2(k) \) are two hash functions given as \( h_1(k) = k \mod m \), \( h_2(k) = k \mod m' \), \( i \) is the probe number that varies from 0 to \( m-1 \), and \( m' \) is chosen to be less than \( m \). We can choose \( m' = m-1 \) or \( m-2 \).

**Pros and Cons**

Double hashing minimizes repeated collisions and the effects of clustering. That is, double hashing is free from problems associated with primary clustering as well as secondary clustering.
Collision Resolution by Open Addressing: **Double Hashing**

**Example:** Consider a hash table of size $m = 10$. Using double hashing, insert the keys $72, 27, 36, 24, 63, 81, 92,$ and $101$ into the table. Take $h_1 = (k \mod 10)$ and $h_2 = (k \mod 8)$.

**Solution**

Let $m = 10$

Initially, the hash table can be given as:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

We have,

$$h(k, i) = [h_1(k) + ih_2(k)] \mod m$$
COLLISIONS

Step 1

Key = 72

\[ h(72, 0) = \left[ 72 \ mod \ 10 + (0 \times 72 \ mod \ 8) \right] \ mod \ 10 \]
\[ = \left[ 2 + (0 \times 0) \right] \ mod \ 10 \]
\[ = 2 \ mod \ 10 \]
\[ = 2 \]

Since \( \tau[2] \) is vacant, insert the key 72 in \( \tau[2] \). The hash table now becomes:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Step 2

Key = 27

\[ h(27, 0) = \left[ 27 \ mod \ 10 + (0 \times 27 \ mod \ 8) \right] \ mod \ 10 \]
\[ = \left[ 7 + (0 \times 3) \right] \ mod \ 10 \]
\[ = 7 \ mod \ 10 \]
\[ = 7 \]

Since \( \tau[7] \) is vacant, insert the key 27 in \( \tau[7] \). The hash table now becomes:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
COLLISIONS

Step 3

Key = 36

\[ h(36, 0) = [(36 \mod 10 + (0 \times 36 \mod 8))] \mod 10 \]

\[ = [6 + (0 \times 4)] \mod 10 \]

\[ = 6 \mod 10 \]

\[ = 6 \]

Since \( \tau[6] \) is vacant, insert the key 36 in \( \tau[6] \). The hash table now becomes:

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
-1 & -1 & 72 & -1 & -1 & -1 & 36 & 27 & -1 & -1 \\
\end{array}
\]

Step 4

Key = 24

\[ h(24, 0) = [(24 \mod 10 + (0 \times 24 \mod 8))] \mod 10 \]

\[ = [4 + (0 \times 0)] \mod 10 \]

\[ = 4 \mod 10 \]

\[ = 4 \]

Since \( \tau[4] \) is vacant, insert the key 24 in \( \tau[4] \). The hash table now becomes:

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
-1 & -1 & 72 & -1 & 24 & -1 & 36 & 27 & -1 & -1 \\
\end{array}
\]
COLLISIONS

Step 5  
Key = 63
\[ h(63, 0) = [63 \mod 10 + (0 \times 63 \mod 8)] \mod 10 \]
\[ = [3 + (0 \times 7)] \mod 10 \]
\[ = 3 \mod 10 \]
\[ = 3 \]
Since \( t[3] \) is vacant, insert the key 63 in \( t[3] \). The hash table now becomes:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>-1</td>
<td>72</td>
<td>63</td>
<td>24</td>
<td>-1</td>
<td>36</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Step 6  
Key = 81
\[ h(81, 0) = [81 \mod 10 + (0 \times 81 \mod 8)] \mod 10 \]
\[ = [1 + (0 \times 1)] \mod 10 \]
\[ = 1 \mod 10 \]
\[ = 1 \]
Since \( t[1] \) is vacant, insert the key 81 in \( t[1] \). The hash table now becomes:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>81</td>
<td>72</td>
<td>63</td>
<td>24</td>
<td>-1</td>
<td>36</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
COLLISIONS

Step 7

Key = 92

\[ h(92, 0) = [92 \mod 10 + (0 \times 92 \mod 8)] \mod 10 \]
\[ = [2 + (0 \times 4)] \mod 10 \]
\[ = 2 \mod 10 \]
\[ = 2 \]

Now \( \tau[2] \) is occupied, so we cannot store the key 92 in \( \tau[2] \). Therefore, try again for the next location. Thus probe, \( i = 1 \), this time.

Key = 92

\[ h(92, 1) = [92 \mod 10 + (1 \times 92 \mod 8)] \mod 10 \]

\[ = [2 + (1 \times 4)] \mod 10 \]
\[ = (2 + 4) \mod 10 \]
\[ = 6 \mod 10 \]
\[ = 6 \]

Now \( \tau[6] \) is occupied, so we cannot store the key 92 in \( \tau[6] \). Therefore, try again for the next location. Thus probe, \( i = 2 \), this time.
Key = 92

\[ h(92, 2) = \left[ 92 \mod 10 + (2 \times 92 \mod 8) \right] \mod 10 \]
\[ = \left[ 2 + (2 \times 4) \right] \mod 10 \]
\[ = \left[ 2 + 8 \right] \mod 10 \]
\[ = 10 \mod 10 \]
\[ = 0 \]

Since \( \tau[0] \) is vacant, insert the key 92 in \( \tau[0] \). The hash table now becomes:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>92</td>
<td>81</td>
<td>72</td>
<td>63</td>
<td>24</td>
<td>-1</td>
<td>36</td>
<td>27</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Step 8

Key = 101
\[ h(101, 0) = [101 \mod 10 + (0 \times 101 \mod 8)] \mod 10 \]
\[ = [1 + (0 \times 5)] \mod 10 \]
\[ = 1 \mod 10 \]
\[ = 1 \]

Now \( T[1] \) is occupied, so we cannot store the key 101 in \( T[1] \). Therefore, try again for the next location. Thus probe, \( i = 1 \), this time.

Key = 101
\[ h(101, 1) = [101 \mod 10 + (1 \times 101 \mod 8)] \mod 10 \]
\[ = [1 + (1 \times 5)] \mod 10 \]
\[ = [1 + 5] \mod 10 \]
\[ = 6 \]

Now \( T[6] \) is occupied, so we cannot store the key 101 in \( T[6] \). Therefore, try again for the next location with probe \( i = 2 \). Repeat the entire process until a vacant location is found. You will see that we have to probe many times to insert the key 101 in the hash table. Although double hashing is a very efficient algorithm, it always requires \( m \) to be a prime number. In our case \( m=10 \), which is not a prime number, hence, the degradation in performance. Had \( m \) been equal to 11, the algorithm would have worked very efficiently. Thus, we can say that the performance of the technique is sensitive to the value of \( m \).
When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table.

All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table.

Though rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently. Consider the hash table of size 5 given below. The hash function used is $h(x) = x \% 5$. Rehash the entries into to a new hash table.
Collision Resolution by Open Addressing: *Rehashing*

Note that the new hash table is of 10 locations, double the size of the original table.

Now, rehash the key values from the old hash table into the new one using hash function—\( h(x) = x \% 10. \)
Collision Resolution by **Chaining**

In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location. That is, location 1 in the hash table points to the head of the linked list of all the key values that hashed to 1. However, if no key value hashes to 1, then location 1 in the hash table contains `NULL`. Below figure shows how the key values are mapped to a location in the hash table and stored in a linked list that corresponds to that location.
APPLICATIONS OF HASHING

• Hash tables are widely used in situations where enormous amounts of data have to be accessed to quickly search and retrieve information. A few typical examples where hashing is used are given here.

• Hashing is used for database indexing. Some database management systems store a separate file known as the index file. When data has to be retrieved from a file, the key information is first searched in the appropriate index file which references the exact record location of the data in the database file. This key information in the index file is often stored as a hashed value.

• In many database systems, file and directory hashing is used in high-performance file systems. Such systems use two complementary techniques to improve the performance of file access. While one of these techniques is caching which saves information in the memory, the other is hashing which makes looking up the file location in the memory much quicker than most other methods.

• Hashing technique is used to implement compiler symbol tables in C++. The compiler uses a symbol table to keep a record of all the user-defined symbols in a C++ program. Hashing facilitates the compiler to quickly look up variable names and other attributes associated with symbols. Hashing is also widely used for Internet search engines.
File Systems
INTRODUCTION

• Nowadays, most organizations use data collection applications which collect large amounts of data in one form or other. For example, when we seek admission in a college, a lot of data such as our name, address, phone number, the course in which we want to seek admission, aggregate of marks obtained in the last examination, and so on, are collected. Similarly, to open a bank account, we need to provide a lot of input. All these data were traditionally stored on paper documents, but handling these documents had always been a chaotic and difficult task.

• Similarly, scientific experiments and satellites also generate enormous amounts of data. Therefore, in order to efficiently analyse all the data that has been collected from different sources, it has become a necessity to store the data in computers in the form of files.
INTRODUCTION

• In computer terminology, a file is a block of useful data which is available to a computer program and is usually stored on a persistent storage medium. Storing a file on a persistent storage medium like hard disk ensures the availability of the file for future use. These days, files stored on computers are a good alternative to paper documents that were once stored in offices and libraries.

• Every file contains data which can be organized in a hierarchy to present a systematic organization. The data hierarchy includes data items such as fields, records, files, and database.
TERMINOLOGY

• A **data field** is an elementary unit that stores a single fact. A data field is usually characterized by its type and size. For example, student’s name is a data field that stores the name of students. This field is of type *character* and its size can be set to a maximum of 20 or 30 characters depending on the requirement.

• A **record** is a collection of related data fields which is seen as a single unit from the application point of view. For example, the student’s record may contain data fields such as name, address, phone number, roll number, marks obtained, and so on.

• A **file** is a collection of related records. For example, if there are 60 students in a class, then there are 60 records. All these related records are stored in a file. Similarly, we can have a file of all the employees working in an organization, a file of all the customers of a company, a file of all the suppliers, so on and so forth.
TERMINOLOGY

• A directory stores information of related files. A directory organizes information so that users can find it easily. For example, consider Fig. below that shows how multiple related files are stored in a student directory.
FILE ORGANIZATION

- The following considerations should be kept in mind before selecting an appropriate file organization method:
  1. Rapid access to one or more records
  2. Ease of inserting/updating/deleting one or more records without disrupting the speed of accessing record(s)
  3. Efficient storage of records
  4. Using redundancy to ensure data integrity

- Although one may find that these requirements are in contradiction with each other, it is the designer’s job to find a good compromise among them to get an adequate solution for the problem at hand. For example, the ease of addition of records can be compromised to get fast access to data.
FILE ORGANIZATION

TECHNIQUES:

1. Sequential Organization
2. Relative File Organization
3. Indexed Sequential File Organization

NOTE: Study all these techniques in detail.
INDEXING

- There are several indexing techniques and each technique works well for a particular application. For a particular situation at hand, we analyse the indexing technique based on factors such as access type, access time, insertion time, deletion time, and space overhead involved. There are two kinds of indices:

1. **Ordered indices** that are sorted based on one or more key values

2. **Hash indices** that are based on the values generated by applying a hash function

**NOTE:** Study all these techniques in detail.