

UNIT - V

Hashing-Hash table, Hash table representations, hash functions, collision resolution techniques- separate chaining, open addressing-linear probing, quadratic probing, double hashing, Re hashing, Extendible hashing,

Pattern matching : Introduction, Brute force, the Boyer –Moore algorithm, Knuth-Morris-Pratt algorithm.

Hashing: Hashing is a technique to convert a range of key values into a range of indexes of an array. Hashing techniques is implemented using hash function and hash table.

Hash Table: It is data structure which contains index and associated data.

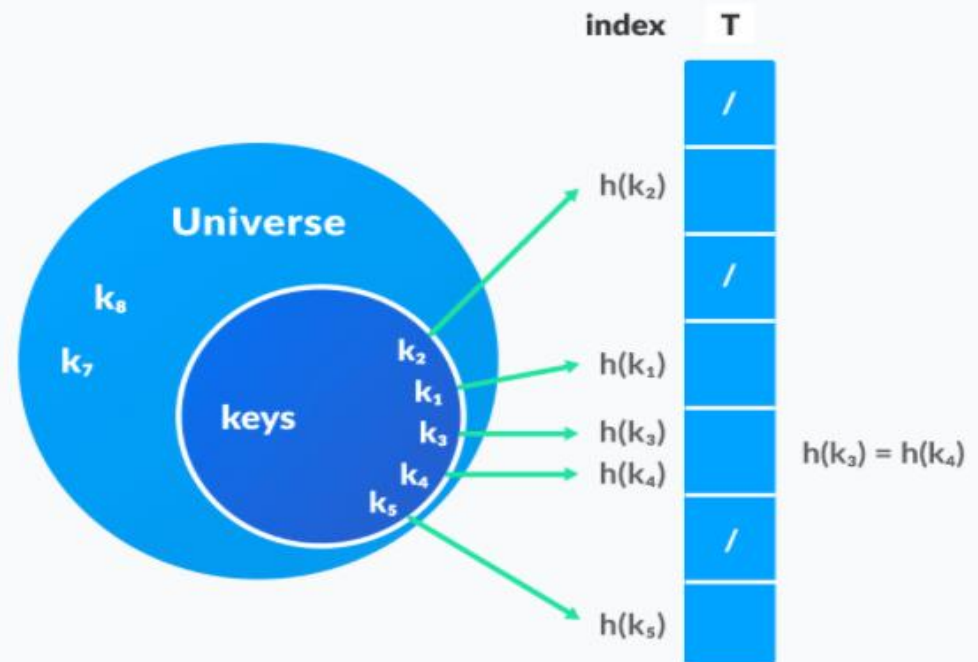
Access of data becomes very fast if we know the index of the desired data.

Hash Function: It is a function which is used to map the key to a hash value.

It is represented as $h(x)$.

Collision: If the same index or hash value is produced by the hash function for multiple keys then, conflict arises.

This situation is called collision.



Hashing...

Types of hash functions:

➤ 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 as hash value. In this case, the hash function can be given as

$$h(x) = x \bmod m.$$

It requires only a single division operation, therefore this method works very fast.

Example:

calculate the hash values of keys 1234 and 5462, where $m=97$.

$$h(1234) = 1234 \% 97 = 70, \quad h(5642) = 5642 \% 97 = 16$$

➤ 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:

$$h(k) = m (ka \bmod 1) \quad \text{where, } (ka \bmod 1) \text{ gives the fractional part of } ka \text{ and } m \text{ is the total number of indices in the hash table.}$$

Hash Function Types...

Example:

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

we will use $a = 0.618033$, $m = 1000$, and $k = 12345$

$$h(12345) = 1000 (12345 * 0.618033 \bmod 1)$$

$$= 1000 (7629.617385 \bmod 1)$$

$$= 1000 (0.617385)$$

$$= 617.385$$

$$= 617$$

➤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.

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 Function Types...

Example:

calculate the hash value for keys 1234 and 5642 using the mid-square method.

The hash table has 100 memory locations.

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.

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, \dots, 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 $k_1 + k_2 + k_3 + \dots + k_n$. This hash value produced by ignoring the last carry, if any.

Hash FunctionTypes...

Example:

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

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:

Key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

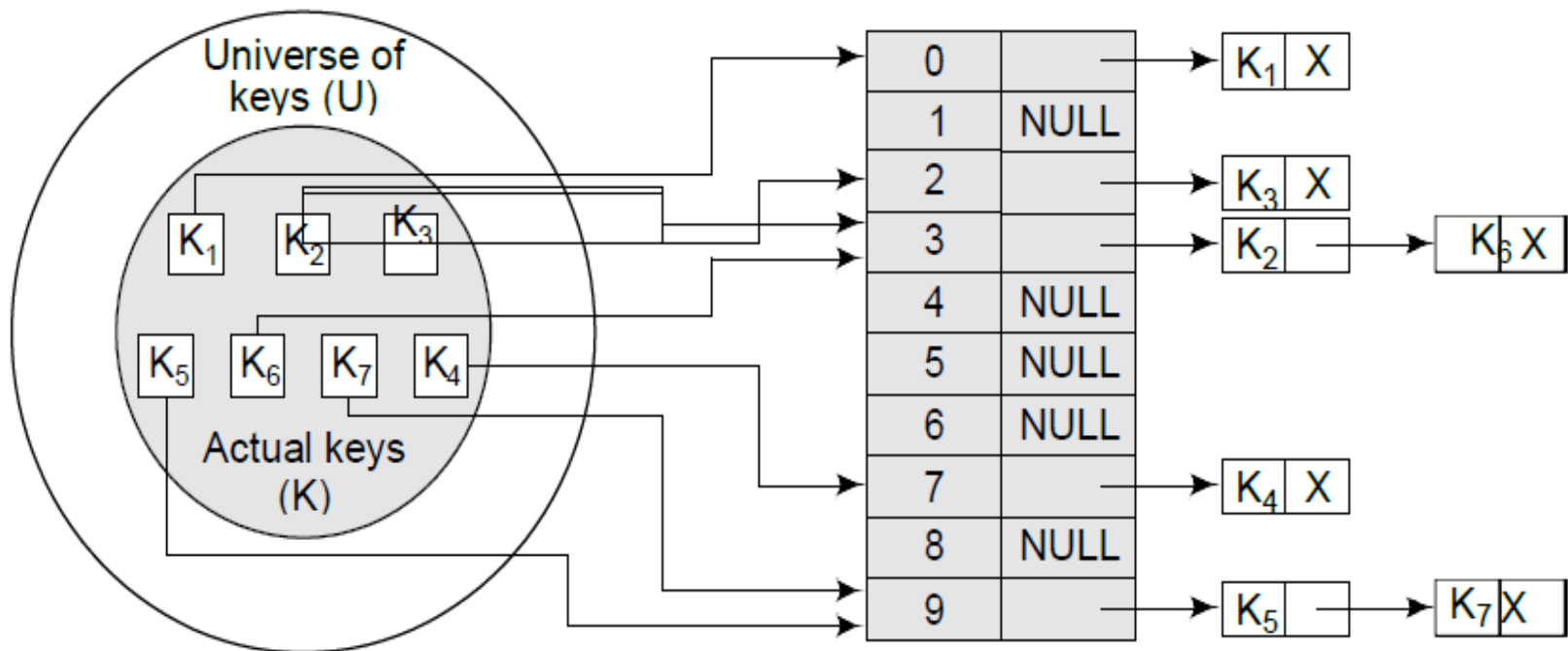
Collision Resolution Techniques

The collision resolution techniques are :

1. Separate chaining
2. Open addressing

Separate chaining:

- In this technique, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a linked list.
- if no element is hashed to a particular index then it will contain NULL.



Collision Resolution Techniques...

Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use hash function $h(k) = k \bmod m$.

In this case, $m=9$.

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL

Insert 7

Key = 7
 $H(k) = 7 \bmod 9$
 $= 7$

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	→ [7 X]
8	NULL

Insert 24

Key = 24
 $H(k) = 24 \bmod 9$
 $= 6$

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	→ [24 X]
7	→ [7 X]
8	NULL

Insert 18

key = 18
 $H(k) = 18 \bmod 9 = 0$

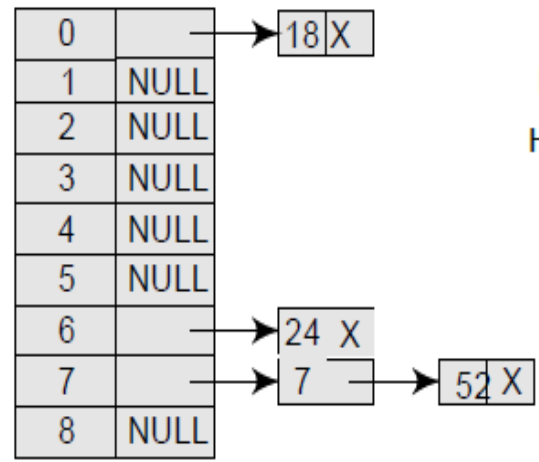
0	→ [18 X]
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	→ [24 X]
7	→ [7 X]
8	NULL

Collision Resolution Techniques...

Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use hash function $h(k) = k \bmod m$.

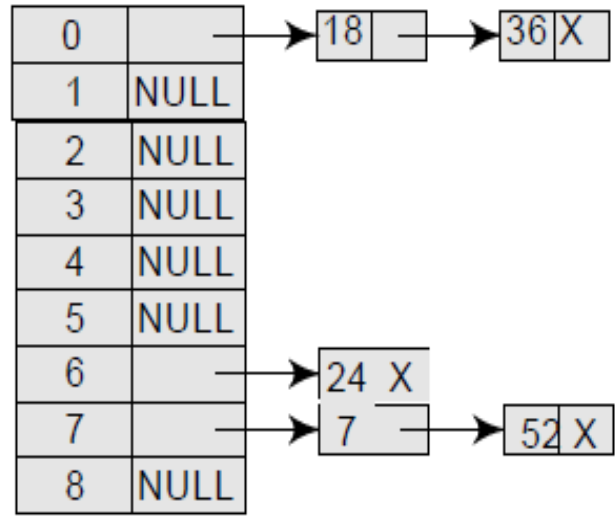
Insert 52

key = 52
 $h(k) = 52 \bmod 9 = 7$

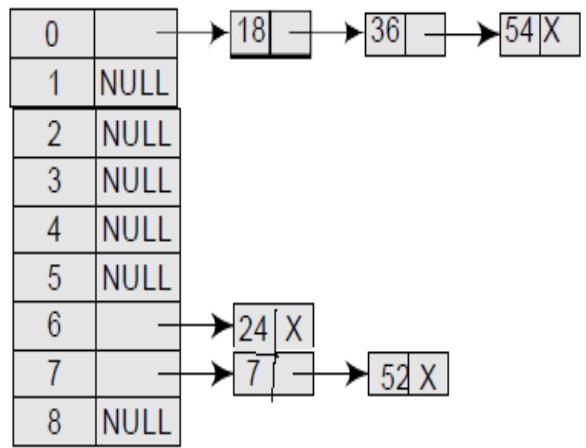


Insert 36

key = 36
 $H(k) = 36 \bmod 9 = 0$

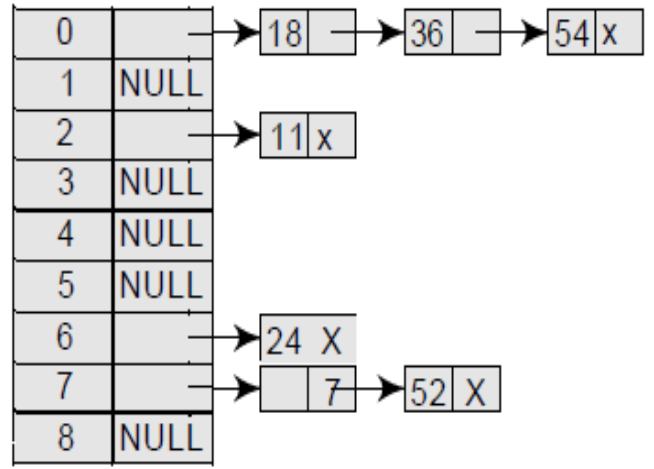


Insert 54



Insert 11

key = 11
 $H(k) = 11 \bmod 9 = 2$

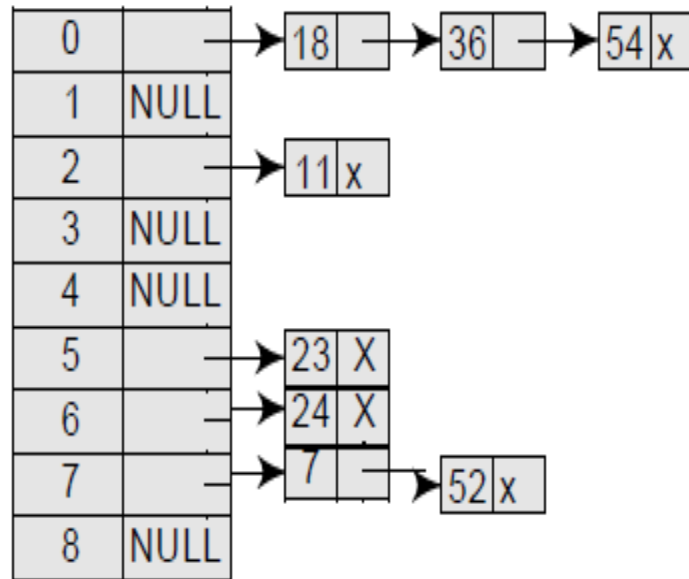


Collision Resolution Techniques...

Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use hash function $h(k) = k \bmod m$.

Insert 23

key = 23
 $h(k) = 23 \bmod 9 = 5$



Collision Resolution Techniques...

Open Addressing:

- In this technique, 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.
- 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.
- 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**
 2. **Quadratic probing**
 3. **Double hashing**
 4. **Rehashing.**

Collision Resolution Techniques...

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:

$$H(k, i) = [h(k) + i] \bmod m$$

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

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

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

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Step 1 key = 72

$$\begin{aligned} H(72, 0) &= (72 \bmod 10 + 0) \bmod 10 \\ &= (2) \bmod 10 \\ &= 2 \end{aligned}$$

Since $t[2]$ is vacant, insert key 72 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Linear Probing...

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

Let $h(k) = k \bmod m$, $m = 10$

Step 2 key = 27

$$\begin{aligned} H(27, 0) &= (27 \bmod 10 + 0) \bmod 10 \\ &= (7) \bmod 10 \\ &= 7 \end{aligned}$$

Since $t[7]$ is vacant, insert key 27 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step 3 key = 36

$$\begin{aligned} H(36, 0) &= (36 \bmod 10 + 0) \bmod 10 \\ &= (6) \bmod 10 \\ &= 6 \end{aligned}$$

Since $t[6]$ is vacant, insert key 36 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Linear Probing...

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

Let $h(k) = k \bmod m$, $m = 10$

Step 4 key = 24

$$\begin{aligned} H(24, 0) &= (24 \bmod 10 + 0) \bmod 10 \\ &= (4) \bmod 10 \\ &= 4 \end{aligned}$$

Since $t[4]$ is vacant, insert key 24 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step 5 key = 63

$$\begin{aligned} H(63, 0) &= (63 \bmod 10 + 0) \bmod 10 \\ &= (3) \bmod 10 \\ &= 3 \end{aligned}$$

Since $t[3]$ is vacant, insert key 63 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Linear Probing...

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

Let $h(k) = k \bmod m$, $m = 10$

Step 6 key = 81

$$\begin{aligned} H(81, 0) &= (81 \bmod 10 + 0) \bmod 10 \\ &= (1) \bmod 10 \\ &= 1 \end{aligned}$$

Since $t[1]$ is vacant, insert key 81 at this location.

0	1	2	3	4	5	6	7	8	9
0	81	72	63	24	-1	36	27	-1	-1

Step 7 key = 92

$$\begin{aligned} H(92, 0) &= (92 \bmod 10 + 0) \bmod 10 \\ &= (2) \bmod 10 \\ &= 2 \end{aligned}$$

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

Key = 92

$$\begin{aligned} H(92, 1) &= (92 \bmod 10 + 1) \bmod 10 \\ &= (2 + 1) \bmod 10 \\ &= 3 \end{aligned}$$

Linear Probing...

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

Let $h(k) = k \bmod m$, $m = 10$

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

Key = 92

$$\begin{aligned} H(92, 2) &= (92 \bmod 10 + 2) \bmod 10 \\ &= (2 + 2) \bmod 10 \\ &= 4 \end{aligned}$$

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

$$\begin{aligned} H(92, 3) &= (92 \bmod 10 + 3) \bmod 10 \\ &= (2 + 3) \bmod 10 \\ &= 5 \end{aligned}$$

Since $t[5]$ is vacant, insert key 92 at this location.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	92	36	27	-1	-1

Linear Probing...

Advantages:

- Easy to compute.

Disadvantages:

- Table must be big enough to get a free cell.
- Time to get a free cell may be quite large.
- Primary Clustering: Any key that hashes into the cluster will require several attempts to resolve the collision.

Quadratic probing

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:

$$H(k, i) = [h(k) + c_1 i + c_2 i^2] \bmod m$$

Where,

m is the size of the hash table,

$h(k) = (k \bmod m)$,

i is the probe number that varies from 0 to $M-1$, and

c_1 and c_2 are constants.

- 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 $c_2 = 3$.

Let $h(k) = k \bmod m$, $m = 10$

Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Quadratic probing...

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 $c_2 = 3$.

We have,

$$H(k, i) = [h(k) + c_1 i + c_2 i^2] \bmod m$$

Step 1 key = 72

$$\begin{aligned} H(72, 0) &= [72 \bmod 10 + 1 * 0 + 3 * 0] \bmod 10 \\ &= [72 \bmod 10] \bmod 10 \\ &= 2 \bmod 10 \\ &= 2 \end{aligned}$$

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

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step 2 key = 27

$$\begin{aligned} H(27, 0) &= [27 \bmod 10 + 1 * 0 + 3 * 0] \bmod 10 \\ &= [27 \bmod 10] \bmod 10 \\ &= 7 \bmod 10 \\ &= 7 \end{aligned}$$

Since $t[7]$ is vacant, insert the key 27 in $t[7]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Quadratic probing...

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 $c = 1$ and $c = 3$.

Step 3 key = 36

$$\begin{aligned} H(36, 0) &= [36 \bmod 10 + 1 * 0 + 3 * 0] \bmod 10 \\ &= [36 \bmod 10] \bmod 10 \\ &= 6 \bmod 10 \\ &= 6 \end{aligned}$$

Since $t[6]$ is vacant, insert the key 36 in $t[6]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step 4 key = 24

$$\begin{aligned} H(24, 0) &= [24 \bmod 10 + 1 * 0 + 3 * 0] \bmod 10 \\ &= [24 \bmod 10] \bmod 10 \\ &= 4 \bmod 10 \\ &= 4 \end{aligned}$$

Since $t[4]$ is vacant, insert the key 24 in $t[4]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Quadratic probing...

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 $c = 1$ and $c = 3$.

Step 5 key = 63

$$\begin{aligned} H(63, 0) &= [63 \bmod 10 + 1 * 0 + 3 * 0] \bmod 10 \\ &= [63 \bmod 10] \bmod 10 \\ &= 3 \bmod 10 \\ &= 3 \end{aligned}$$

Since $t[3]$ is vacant, insert the key 63 in $t[3]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step 6 key = 81

$$\begin{aligned} H(81, 0) &= [81 \bmod 10 + 1 * 0 + 3 * 0] \bmod 10 \\ &= [81 \bmod 10] \bmod 10 \\ &= 81 \bmod 10 \\ &= 1 \end{aligned}$$

Since $t[1]$ is vacant, insert the key 81 in $t[1]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Quadratic probing...

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 $c = 1$ and $c = 3$.

Step 7 key = 101

$$\begin{aligned} H(101, 0) &= [101 \bmod 10 + 1 * 0 + 3 * 0] \bmod 10 \\ &= [101 \bmod 10 + 0] \bmod 10 \\ &= 1 \bmod 10 \\ &= 1 \end{aligned}$$

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

Key = 101

$$\begin{aligned} H(101, 0) &= [101 \bmod 10 + 1 * 1 + 3 * 1] \bmod 10 \\ &= [101 \bmod 10 + 1 + 3] \bmod 10 \\ &= [101 \bmod 10 + 4] \bmod 10 \\ &= [1 + 4] \bmod 10 \\ &= 5 \bmod 10 \\ &= 5 \end{aligned}$$

Since $t[5]$ is vacant, insert the key 101 in $t[5]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	101	36	27	-1	-1

Quadratic probing...

Advantage:

- Eliminates Primary Clustering problem.

Disadvantage:

- Secondary Clustering: Elements that hash to the same position will probe the same alternative cells.

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:

$$H(k, i) = [h_1(k) + i h_2(k)] \bmod m$$

- Where m is the size of the hash table, $h_1(k)$ and $h_2(k)$ are two hash functions given as $h_1(k) = k \bmod m$ and $h_2(k) = k \bmod m'$, 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 $m-2$.

Advantage:

- 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.

Disadvantage:

- Implementation is difficult as it uses two hash functions.

Double Hashing...

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

Let $m = 10$

Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

We have,

$$H(k, i) = [h_1(k) + ih_2(k)] \bmod m$$

Step 1 key = 72

$$\begin{aligned} H(72, 0) &= [72 \bmod 10 + (0 * 72 \bmod 8)] \bmod 10 \\ &= [2 + (0 * 0)] \bmod 10 \\ &= 2 \bmod 10 \\ &= 2 \end{aligned}$$

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

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step 2 key = 27

$$\begin{aligned} H(27, 0) &= [27 \bmod 10 + (0 * 27 \bmod 8)] \bmod 10 \\ &= [7 + (0 * 3)] \bmod 10 \\ &= 7 \bmod 10 \\ &= 7 \end{aligned}$$

Since $t[7]$ is vacant, insert the key 27 in $t[7]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Double Hashing...

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

Step 3

key = 36

$$\begin{aligned} H(36, 0) &= [36 \bmod 10 + (0 * 36 \bmod 8)] \bmod 10 \\ &= [6 + (0 * 4)] \bmod 10 \\ &= 6 \bmod 10 \\ &= 6 \end{aligned}$$

Since $t[6]$ is vacant, insert the key 36 in $t[6]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step 4

key = 24

$$\begin{aligned} H(24, 0) &= [24 \bmod 10 + (0 * 24 \bmod 8)] \bmod 10 \\ &= [4 + (0 * 0)] \bmod 10 \\ &= 4 \bmod 10 \\ &= 4 \end{aligned}$$

Since $t[4]$ is vacant, insert the key 24 in $t[4]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Double Hashing...

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

Step 5 key = 63

$$\begin{aligned} H(63, 0) &= [63 \bmod 10 + (0 * 63 \bmod 8)] \bmod 10 \\ &= [3 + (0 * 7)] \bmod 10 \\ &= 3 \bmod 10 \\ &= 3 \end{aligned}$$

Since $t[3]$ is vacant, insert the key 63 in $t[3]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step 6 key = 81

$$\begin{aligned} H(81, 0) &= [81 \bmod 10 + (0 * 81 \bmod 8)] \bmod 10 \\ &= [1 + (0 * 1)] \bmod 10 \\ &= 1 \bmod 10 \\ &= 1 \end{aligned}$$

Since $t[1]$ is vacant, insert the key 81 in $t[1]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Double Hashing...

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

Step 7

key = 92

$$\begin{aligned} H(92, 0) &= [92 \bmod 10 + (0 * 92 \bmod 8)] \bmod 10 \\ &= [2 + (0 * 4)] \bmod 10 \\ &= 2 \bmod 10 \\ &= 2 \end{aligned}$$

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

Key = 92

$$\begin{aligned} H(92, 1) &= [92 \bmod 10 + (1 * 92 \bmod 8)] \bmod 10 \\ &= [2 + (1 * 4)] \bmod 10 \\ &= (2 + 4) \bmod 10 \\ &= 6 \bmod 10 \\ &= 6 \end{aligned}$$

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

Key = 92

$$\begin{aligned} H(92, 2) &= [92 \bmod 10 + (2 * 92 \bmod 8)] \bmod 10 \\ &= [2 + (2 * 4)] \bmod 10 \\ &= [2 + 8] \bmod 10 \\ &= 10 \bmod 10 \\ &= 0 \end{aligned}$$

Since $t[0]$ is vacant, insert the key 92 in $t[0]$. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24	-1	36	27	-1	-1

Rehashing

Rehashing :

- 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.

Example : 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.

0	1	2	3	4
	26	31	43	17

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

0	1	2	3	4	5	6	7	8	9

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

0	1	2	3	4	5	6	7	8	9
	31		43			26	17		

Extendible Hashing

Extendible Hashing :

- It is a dynamic hashing method in which directories and buckets are used to hash data.
- It is an aggressively flexible method in which the hash function also experiences dynamic changes.

Terminology used in Extendible hashing:

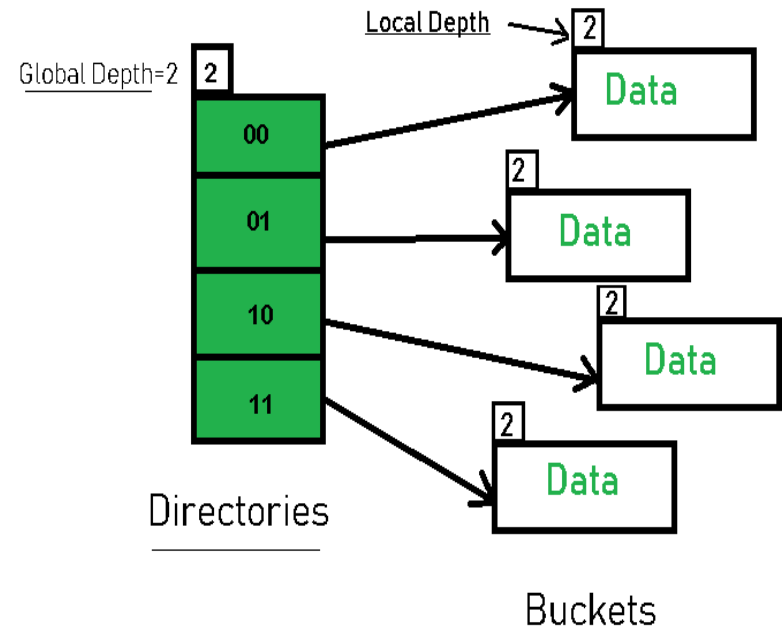
Directories: The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.

Buckets: The buckets are used to hash the actual data.

Global Depth: It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys.

Global Depth = Number of bits in directory id.

Local Depth: It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.



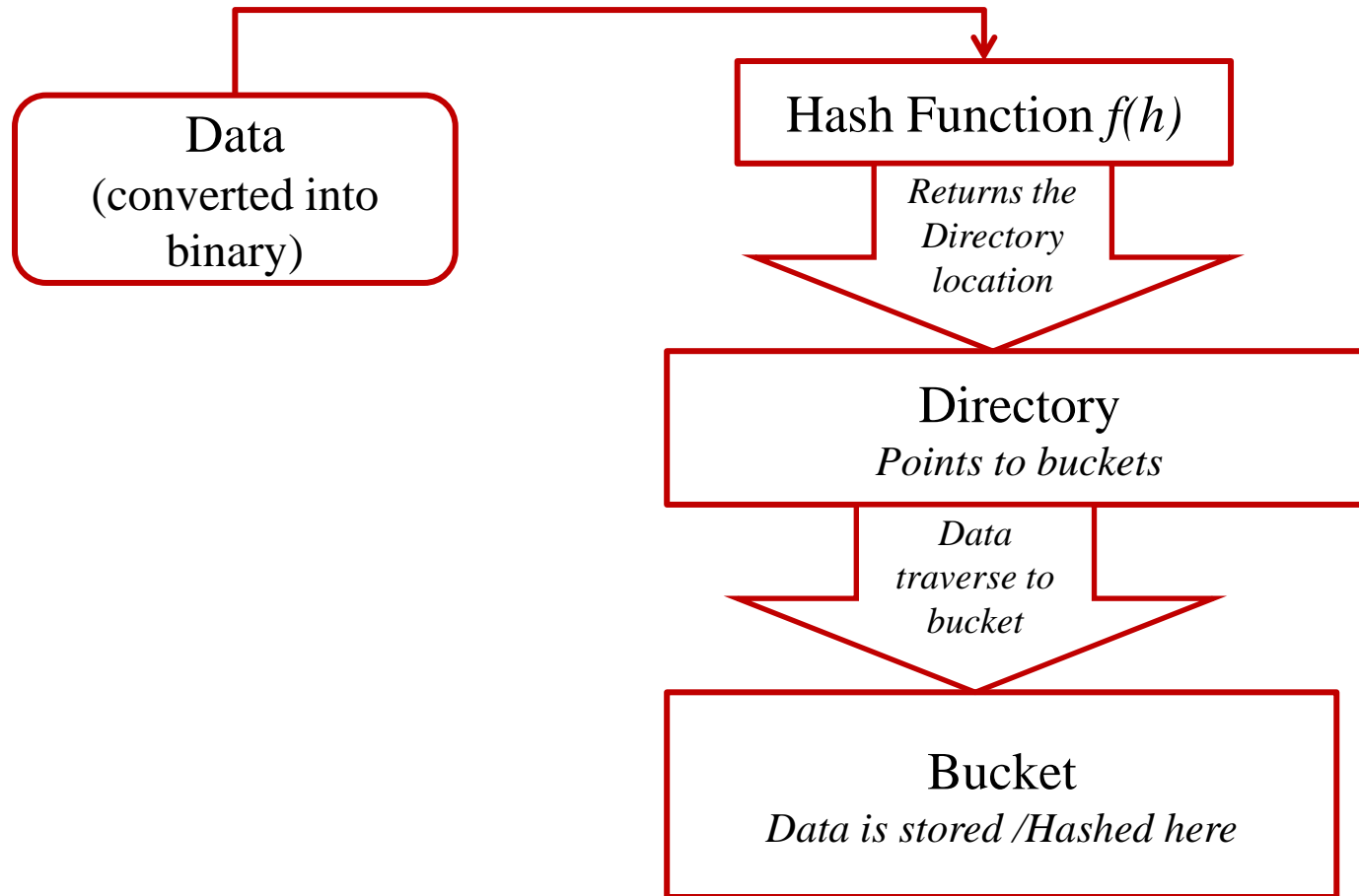
Extendible Hashing

Extendible Hashing...

Bucket Splitting: When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.

Directory Expansion: Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.

Basic Working of Extendible Hashing:



Extendible Hashing...

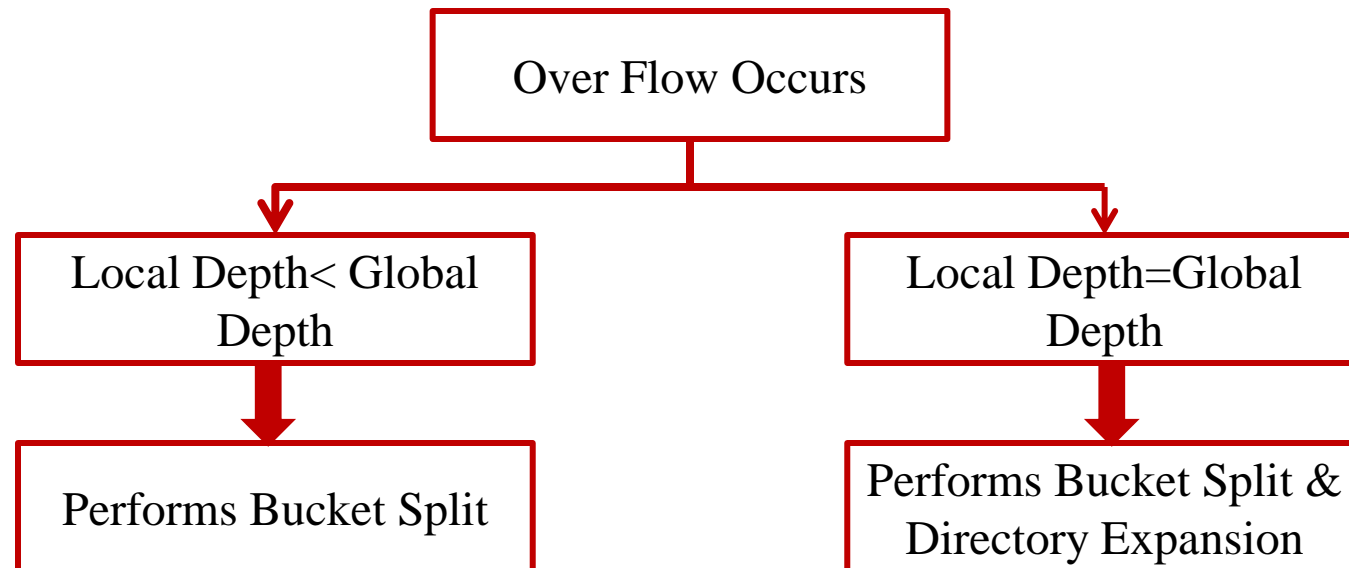
Tackling Over Flow Condition during Data Insertion:

- Many times, while inserting data in the buckets, it might happen that the Bucket overflows. In such cases, we need to follow an appropriate procedure to avoid mishandling of data.
- First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.

Case1: If the local depth of the overflowing Bucket is equal to the global depth, then Directory Expansion, as well as Bucket Split, needs to be performed. Then increment the global depth and the local depth value by 1. And, assign appropriate pointers.

Directory expansion will double the number of directories present in the hash structure.

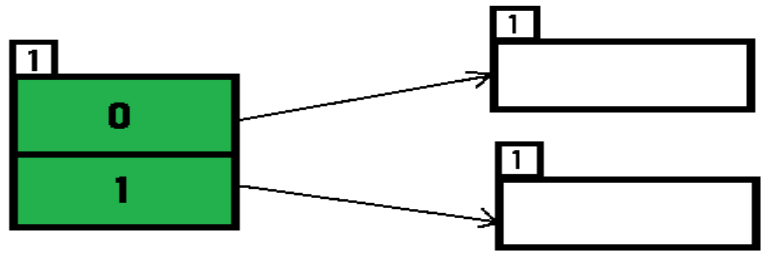
Case2: In case the local depth is less than the global depth, then only Bucket Split takes place. Then increment only the local depth value by 1. And, assign appropriate pointers.



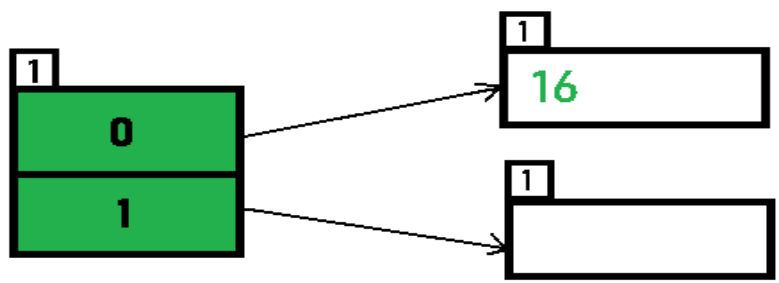
Extendible Hashing...

Example :Hash the following elements: 16,4,6,22,24,10,31,7,9,20,26 using Extendible hashing where bucket size is 3 and Hash Function: Suppose the global depth is X ,then the Hash Function returns X LSBs.

➤Initially, the global-depth and local-depth is always 1.
Thus, the hashing frame looks like this:



➤**Insert 16:**
The binary format of 16 is 10000 and global-depth is 1.
The hash function returns 1 LSB of 10000 which is 0.
Hence, 16 is mapped to the directory with id=0.



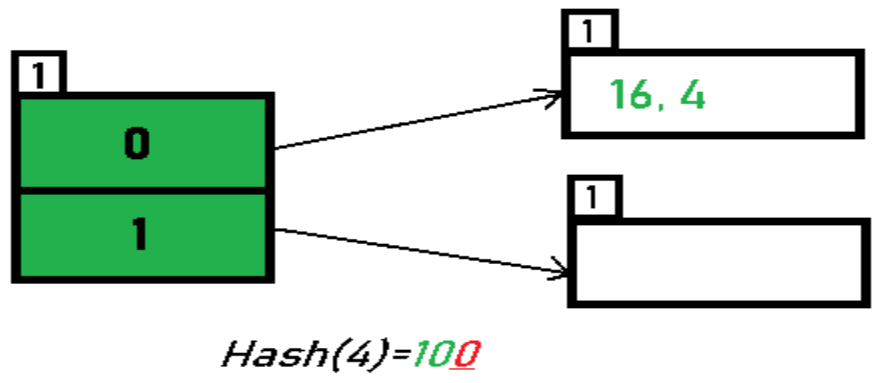
Hash(16)= 10000

Key	Binary Representation
16	10000
4	00100
6	00110
22	10110
24	11000
10	01010
31	11111
7	00111
9	01001
20	10100
26	01101

Extendible Hashing...

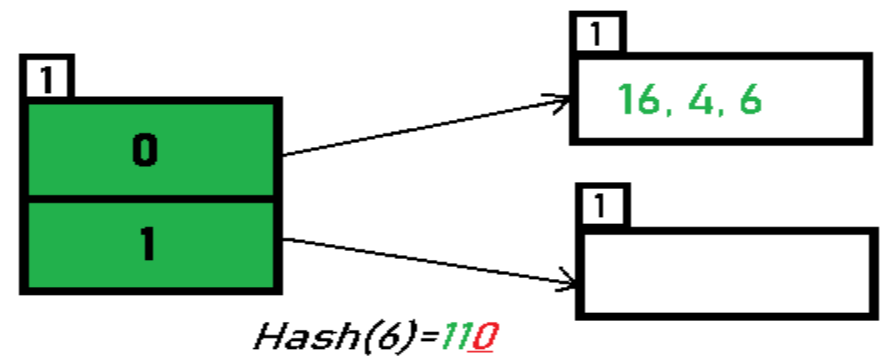
➤Insert 4:

The binary format of 4 is 00100 and global-depth is 1.
The hash function returns 1 LSB of 00100 which is 0.
Hence, 4 is mapped to the directory with id=0.



➤Insert 6:

$Hash(6)=00110$

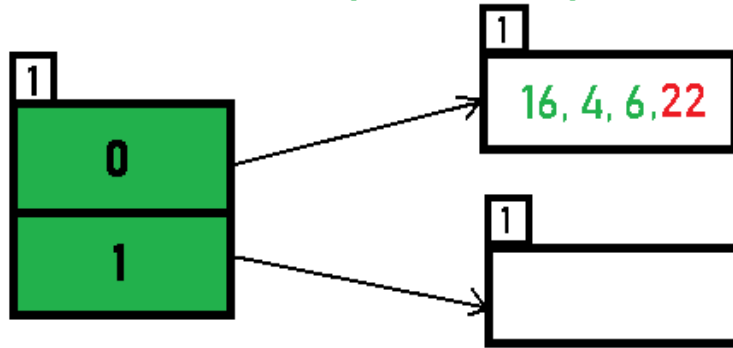


Key	Binary Representation
16	10000
4	00100
6	00110
22	10110
24	11000
10	01010
31	11111
7	00111
9	01001
20	10100
26	01101

Extendible Hashing...

➤ Insert 22

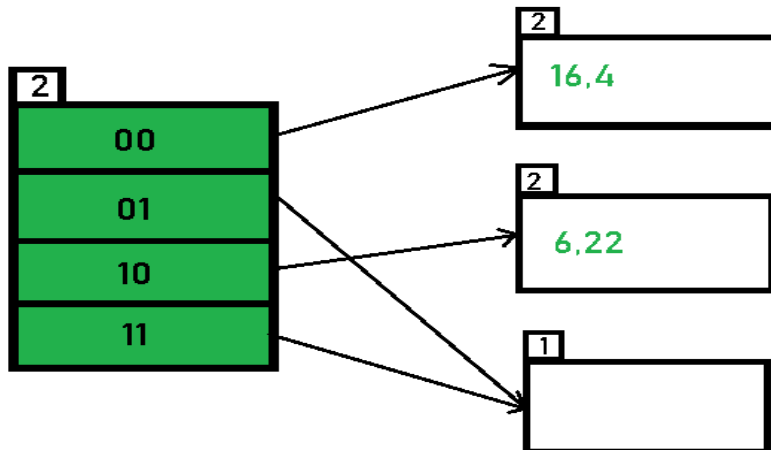
Hash(22)=10110, The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.



Hash(22)=10110

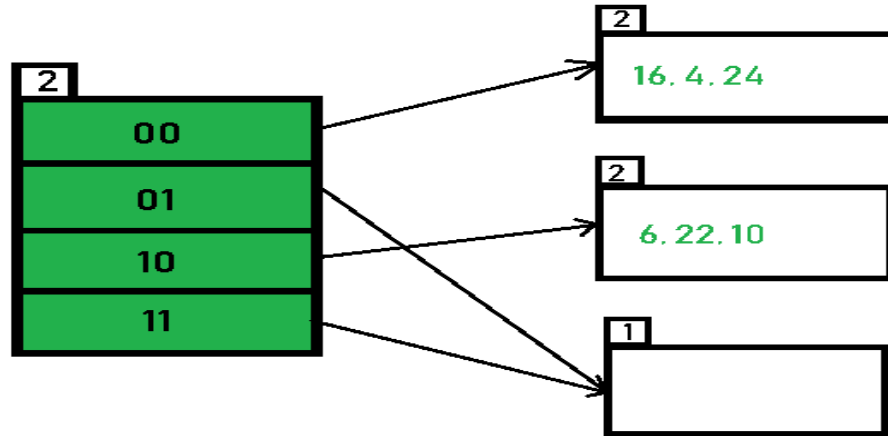
Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now, the global depth is 2. Hence, 16, 4, 6, 22 are now rehashed w.r.t 2 LSBs. [16(10000), 4(100), 6(110), 22(10110)]

After Bucket Split and Directory Expansion



Extendible Hashing...

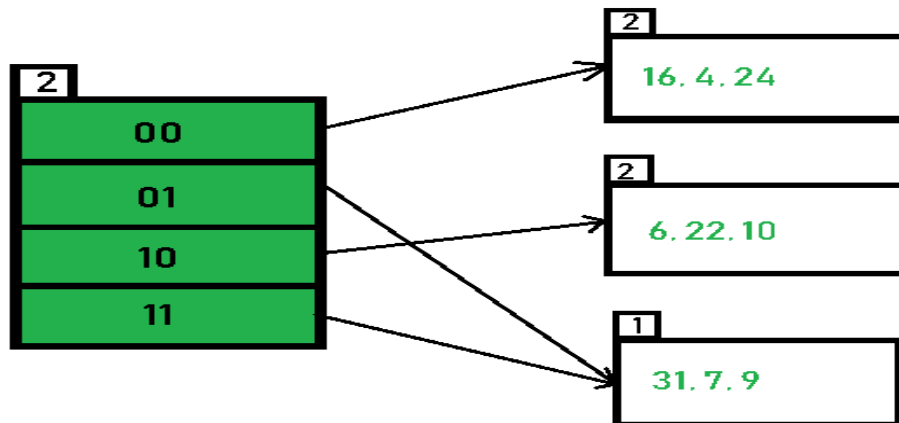
- **Inserting 24 and 10:** 24(110**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10



*Hash(24) = 110**00***

*Hash(10) = 10**10***

- **Inserting 31,7,9:** All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11.



*Hash(31) = 111**11***

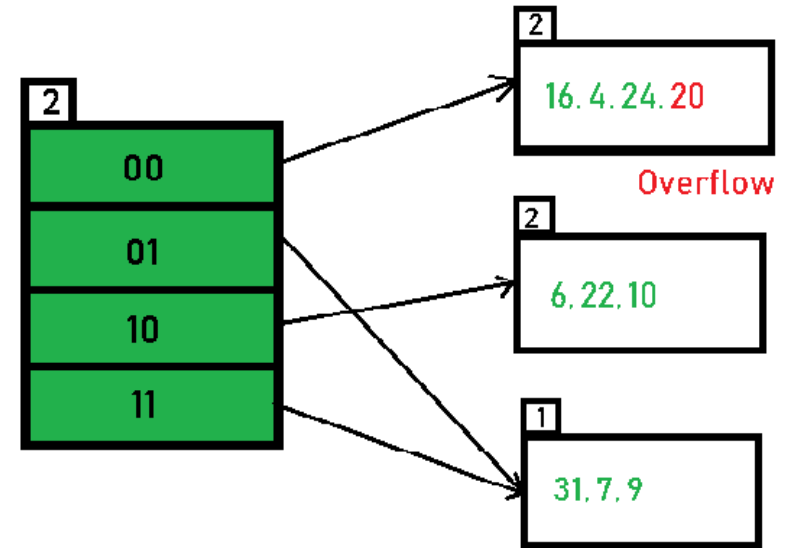
*Hash(7) = 1**11***

*Hash(9) = 10**01***

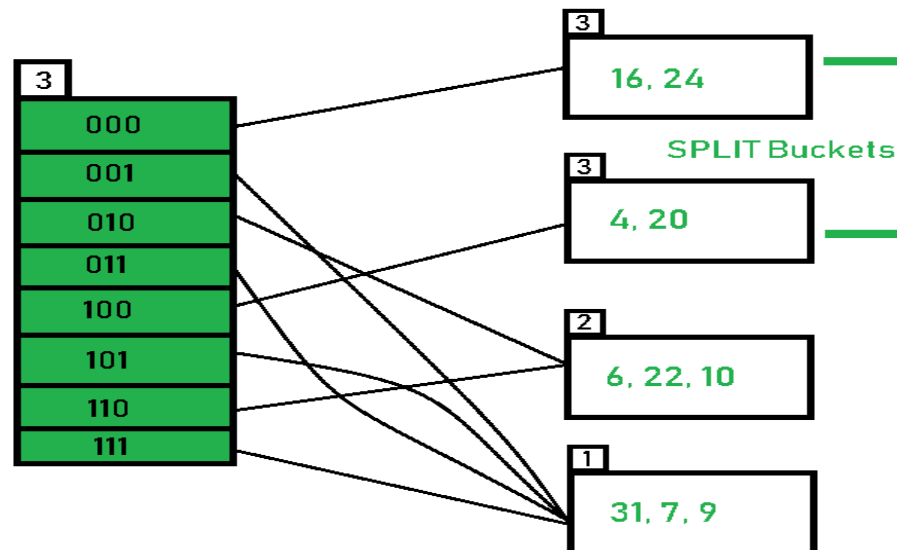
Extendible Hashing...

- **Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.
- 20 is inserted in bucket pointed out by 00. since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth..

Overflow, Local Depth=Global Depth

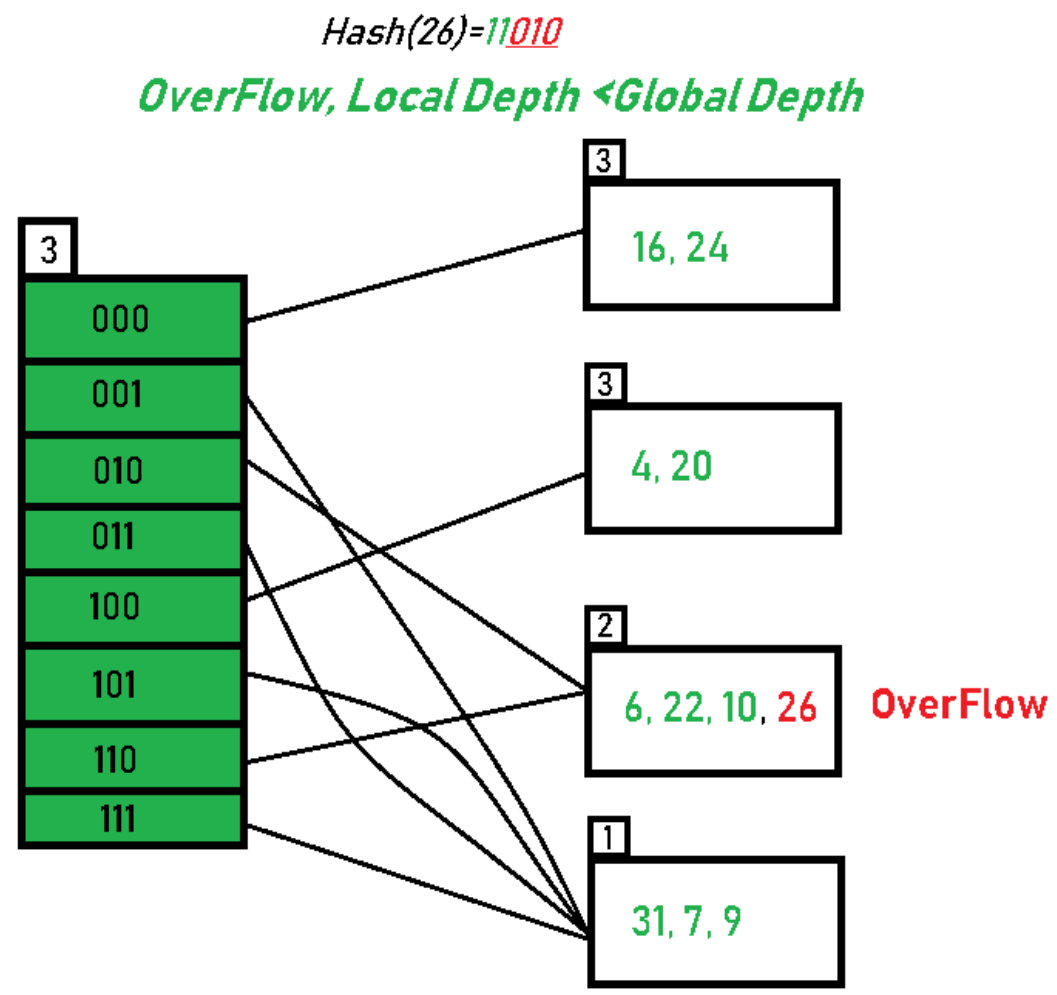


*Hash(20)=101**00***



Extendible Hashing...

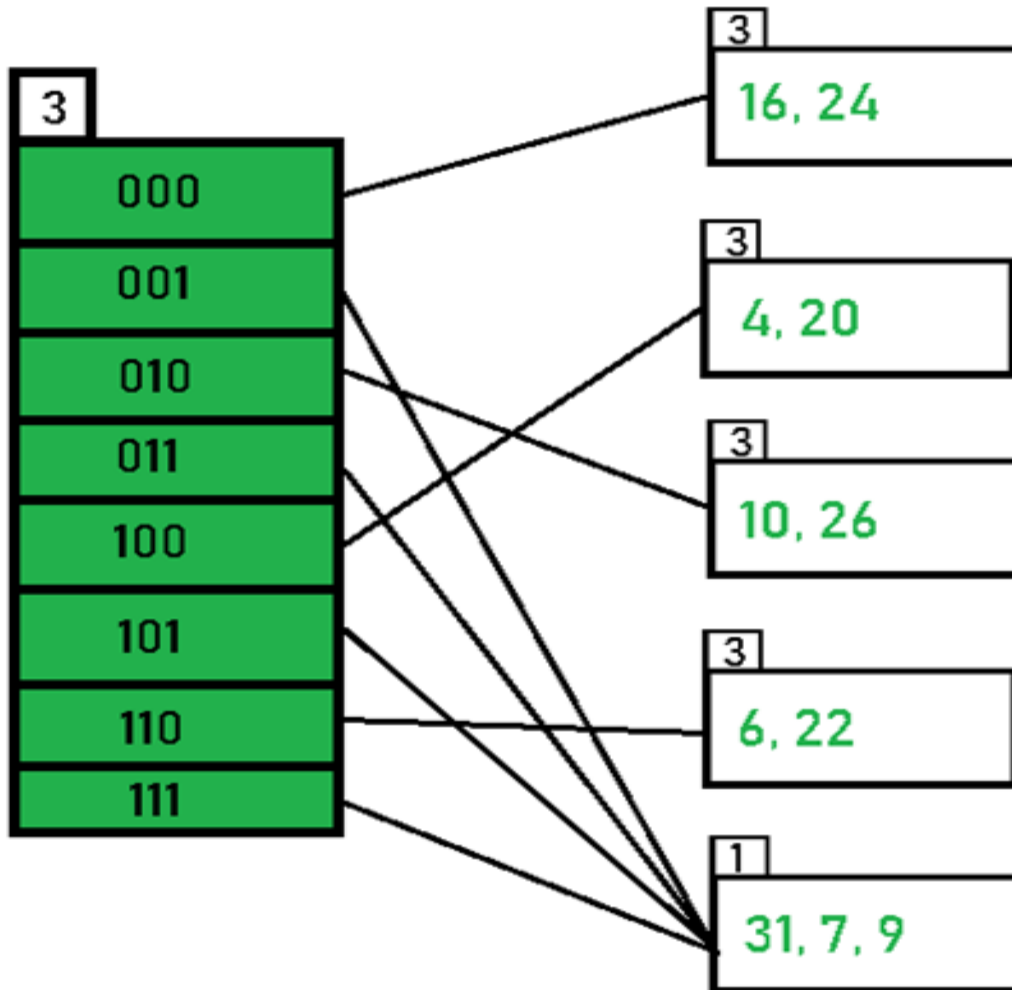
➤ **Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



Key	Binary Representation
16	10000
4	00100
6	00110
22	10110
24	11000
10	01010
31	11111
7	00111
9	01001
20	10100
26	01101

Extendible Hashing...

➤ The bucket overflows, and the **local depth of bucket** < **Global depth** ($2 < 3$), directories are not doubled but, only the bucket is split and elements are rehashed. Finally, the output of hashing the given list of numbers is obtained.



Key	Binary Representation
16	10000
4	00100
6	00110
22	10110
24	11000
10	01010
31	11111
7	00111
9	01001
20	10100
26	01101

Pattern Matching

Pattern Matching:

- The Pattern Matching algorithms are also referred as String Searching Algorithms
- These algorithms are useful in the case of searching a pattern within the text or searching a substring within a String.

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A

A	A	B	A							A	A	B	A		
A	A	B	A	A	C	A	A	D	A	A	B	A	A	B	A
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
												A	A	B	A

Pattern Found at 0, 9 and 12

Pattern Matching Algorithms:

1. Brute force algorithm
2. Boyer –Moore algorithm
3. Knuth-Morris-Pratt algorithm

Brute Force Algorithm

Brute force algorithm:

- The Brute Force algorithm compares the pattern to the text, one character at a time, until unmatching characters are found.
- If unmatched characters are found move the pattern down the text by one character.
- The algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching the end of the text.

Algorithm BruteForceMatch(T,P):

Input: Strings T with n characters and P with m characters

Output: String index of the first substring of T matching P, or an indication that P is not a substring of T

for i:=0 to n-m do //for each candidate index in T do //

{ j:=0

while (j<m and T[i+j]=P[j]) do j:=j+1

if j=m then return i

}

return “ there is no substring of T matching P.”

Brute Force Algorithm...

Example:

Sequence

C	T	C	A	C	T	G	C	C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Pattern

C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---

Sequence

C	T	C	A	C	T	G	C	C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Pattern

C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---

Sequence

C	T	C	A	C	T	G	C	C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Pattern

C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---

Sequence

C	T	C	A	C	T	G	C	C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Pattern

C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---

Sequence

C	T	C	A	C	T	G	C	C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Pattern

C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---

Sequence

C	T	C	A	C	T	G	C	C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Pattern

C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---

Sequence

C	T	C	A	C	T	G	C	C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Pattern

C	T	G	C	C	T	A	G
---	---	---	---	---	---	---	---

KMP Algorithm

- KMP Algorithm is one of the most popular patterns matching algorithms. KMP stands for **Knuth Morris Pratt**.
- KMP algorithm was invented by **Donald Knuth** and **Vaughan Pratt** together and independently by **James H Morris** in the year 1970. In the year 1977, all the three jointly published KMP Algorithm.
- KMP algorithm is used to find a "**Pattern**" in a "**Text**". This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a pre-processed table called "**Prefix Table**" to skip characters comparison while matching.
- Prefix table is also known as **LPS Table**. Here LPS stands for "**Longest proper Prefix which is also Suffix**".

Steps for Creating LPS Table (Prefix Table)

Step 1 : Define a one dimensional array with the size equal to the length of the Pattern.
(LPS[size])

Step 2 : Define variables **i & j**. Set $i = 0$, $j = 1$ and $LPS[0] = 0$.

Step 3 : Compare the characters at **Pattern[i]** and **Pattern[j]**.

Step 4 : If both are matched then set $LPS[j] = i+1$ and increment both i & j values by one.
Goto to Step 3.

Step 5 : If both are not matched then check the value of variable ' i '. If it is '0' then set $LPS[j] = 0$ and increment ' j ' value by one, if it is not '0' then set $i = LPS[i-1]$. Goto Step 3.

Step 6: Repeat above steps until all the values of LPS[] are filled.

KMP Algorithm...

Example for creating KMP Algorithm's LPS Table (Prefix Table)

Consider the following Pattern

Pattern :

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Let us define LPS[] table with size 7 which is equal to length of the Pattern

LPS

0	1	2	3	4	5	6

Step 1 - Define variables i & j. Set $i = 0$, $j = 1$ and $LPS[0] = 0$.

LPS

0	1	2	3	4	5	6
0						

$i = 0$ and $j = 1$

Step 2 - Compare Pattern[i] with Pattern[j] ==> A with B.

Since both are not matching and also " $i = 0$ ", we need to set $LPS[j] = 0$ and increment 'j' value by one.

LPS

0	1	2	3	4	5	6
0	0					

$i = 0$ and $j = 2$

KMP Algorithm...

Pattern :

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Step 3 - Compare Pattern[i] with Pattern[j] ==> A with C.

Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

0 1 2 3 4 5 6
LPS

0	0	0				
---	---	---	--	--	--	--

i = 0 and j = 3

Step 4 - Compare Pattern[i] with Pattern[j] ==> A with D.

Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

0 1 2 3 4 5 6
LPS

0	0	0	0			
---	---	---	---	--	--	--

i = 0 and j = 4

Step 5 - Compare Pattern[i] with Pattern[j] ==> A with A.

Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

0 1 2 3 4 5 6
LPS

0	0	0	0	1		
---	---	---	---	---	--	--

i = 1 and j = 5

KMP Algorithm...

Pattern :

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Step 6 - Compare Pattern[i] with Pattern[j] ==> B with B.
Since both are matching set $LPS[j] = i+1$ and increment both i & j value by one.

LPS

0	1	2	3	4	5	6
0	0	0	0	1	2	

i = 2 and j = 6

Step 7 - Compare Pattern[i] with Pattern[j] ==> C with D.
Since both are not matching and $i \neq 0$, we need to set $i = LPS[i-1]$
==> $i = LPS[2-1] = LPS[1] = 0$.

LPS

0	1	2	3	4	5	6
0	0	0	0	1	2	

i = 0 and j = 6

Step 8 - Compare Pattern[i] with Pattern[j] ==> A with D.
Since both are not matching and also " $i = 0$ ", we need to set $LPS[j] = 0$ and increment 'j' value by one.

LPS

0	1	2	3	4	5	6
0	0	0	0	1	2	0

Here LPS[] is filled with all values so we stop the process. The final LPS[] table is as follows...

LPS

0	1	2	3	4	5	6
0	0	0	0	1	2	0

KMP Algorithm...

How to use LPS Table:

- We use the LPS table to decide how many characters are to be skipped for comparison when a mismatch has occurred.
- When a mismatch occurs, check the LPS value of the previous character of the mismatched character in the pattern. If it is '0' then start comparing the first character of the pattern with the next character to the mismatched character in the text.
- If it is not '0' then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in pattern with the mismatched character in the Text.
- **Example :** Apply the KMP for the following

Text : ABC ABCDAB ABCDABCDABDE
Pattern : ABCDABD

LPS[] table for the above pattern is as follows...

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

KMP Algorithm...

Text : ABC ABCDAB ABCDABCDABDE

Pattern : ABCDABD

LPS[] table for the above pattern is as follows...

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Step 1 - Start comparing first character of Pattern with first character of Text from left to right

Text	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
	0	1	2	3	4	5	6																
Pattern	A	B	C	D	A	B	D																

Here mismatch occurred at Pattern[3], so we need to consider LPS[2] value. Since LPS[2] value is '0' we must compare first character in Pattern with next character in Text.

KMP Algorithm...

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Step 2 - Start comparing first character of Pattern with next character of Text.

Text	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
Pattern					0	1	2	3	4	5	6												
					A	B	C	D	A	B	D												

Here mismatch occurred at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 3 - Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values

Text	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
Pattern									0	1	2	3	4	5	6								
									A	B	C	D	A	B	D								

Here mismatch occurred at Pattern[2], so we need to consider LPS[1] value. Since LPS[1] value is '0' we must compare first character in Pattern with next character in Text.

KMP Algorithm...

LPS 0 1 2 3 4 5 6

0	0	0	0	1	2	0
---	---	---	---	---	---	---

Step 4 - Compare Pattern[0] with next character in Text.

Text A B C A B C D A B A B C D A B C D A B D E

Pattern

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Here mismatch occurred at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 5 - Compare Pattern[2] with mismatched character in Text.

Text A B C A B C D A B A B C D A B C D A B D E

Pattern

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Here all the characters of Pattern matched with a substring in Text which is starting from index value 15. So we conclude that given Pattern found at index 15 in Text.

Boyer Moore Algorithm

- It was developed by Robert S. Boyern and J Strother Moore in 1977.
- The Boyer-Moore algorithm is consider the most efficient string-matching algorithm.
- It works the fastest when the Text is moderately sized and the pattern is relatively long.
- Boyer Moore is a combination of following two approaches:
 - 1) Bad Character Heuristic
 - 2) Good Suffix Heuristic
- Both of the above heuristics can also be used independently to search a pattern in a text.
- It processes the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by the max of the slides suggested by the two heuristics. So it uses best of the two heuristics at every step.
- Boyer Moore algorithm starts matching from the last character of the pattern.


Bad Character Heuristic:

- The character of the text which doesn't match with the current character of the pattern is called the **Bad Character**.
- Upon mismatch, we shift the pattern until –
 - 1) The mismatch becomes a match.
 - 2) Pattern P move past the mismatched character.

Boyer Moore Algorithm

Case 1 : Mismatch become match We will lookup the position of last occurrence of mismatching character in pattern and if mismatching character exist in pattern then we'll shift the pattern such that it get aligned to the mismatching character in text T.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	C	C
T	A	T	G	T	G											



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	C	C
		T	A	T	G	T	G									

Case 2: Pattern move past the mismatch character We'll lookup the position of last occurrence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	C	C
			T	A	T	G	T	G								



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
G	C	A	A	T	G	C	C	T	A	T	G	T	G	A	C	C
								T	A	T	G	T	G			