DATA STRUCTURES

UNIT-4

Hashing

Dr G.Kalyani

Topics

- **Dictionaries**
- **Hashing**
- **Hash Functions**
- **Handling the collisions**
- **Closed Addressing**
- **Open Addressing**
- **Rehashing**
- **Extendable Hashing**

Dictionaries

- Dictionaries stores elements so that they can be located quickly using keys.
- For eg
  - A Dictionary may hold bank accounts.
  - In which key will be account number.
  - And each account may stores many additional information.

How to Implement a Dictionary?

- Different data structure to realize a key
  - Array ,
  - Linked list
  - Binary tree
  - Red/Black tree
  - AVL Tree
  - B-Tree
  - **Hash table**

Why Hashing??

- The sequential search algorithm takes time proportional to the data size, i.e, **O(n)**.
- Binary search improves on liner search reducing the search time to **O(log n)**.
- With a BST, an **O(log n)** search efficiency can be obtained; but the worst-case complexity is **O(n)**.
- To guarantee the **O(log n)** search time, BST height balancing is required (i.e., AVL trees).

Why Hashing?? (Cntd.)

- Suppose that we want to store 10,000 students records (each with a 5-digit ID) in a given container.
  - A linked list implementation would take **O(n)** time.
  - A height balanced tree would give **O(log n)** access time.
  - Using an array of size 100,000 would give **O(1)** access time but will lead to a lot of space wastage.
- Is there some way that we could get **O(1)** access without wasting a lot of space?
- **The answer is hashing**.

Hashing

- Another important and widely useful technique for implementing dictionaries
- Constant time per operation (on the average)
- Like an array, come up with a function to map the large range into one which we can manage.

Basic Idea of Hashing

- Use *hash function* to map keys into positions in a *hash table*
- Ideally
  - If Student A has Key '*k*' and '*h*' is hash function, then *A's Details* is stored in position '*h(k)*' of table
  - To search for A, compute *h(k)* to locate position.
  - If no element, dictionary does not contain *A*.

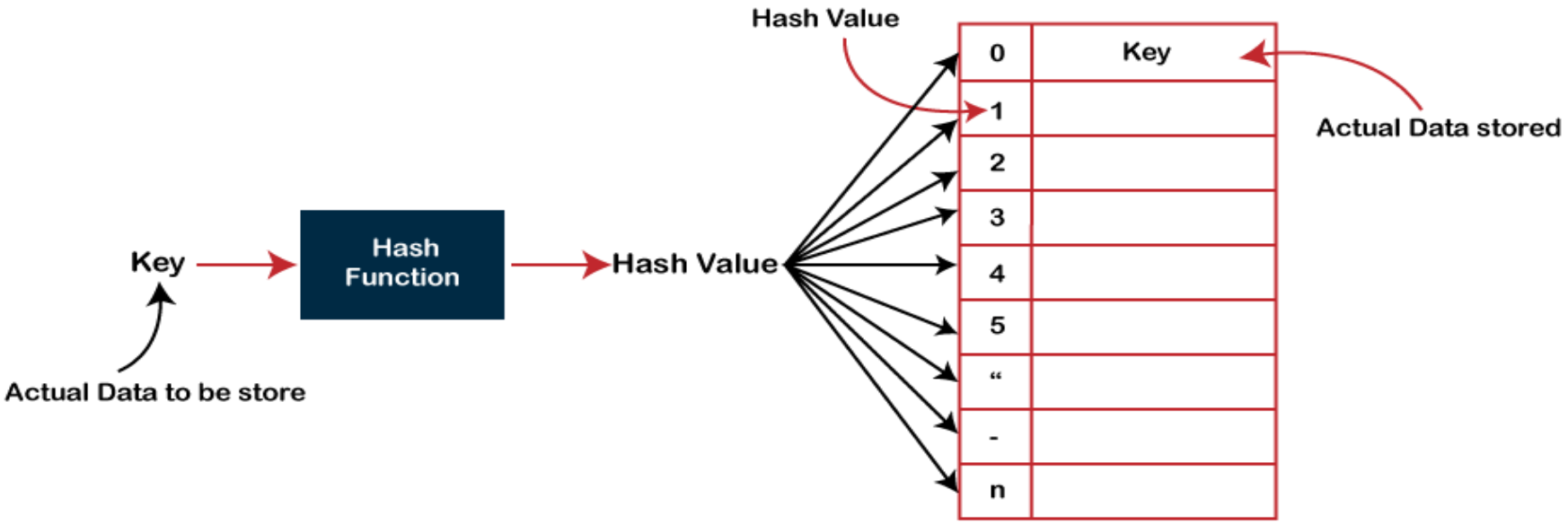
Hashing and Hash Table

- Hashing is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster.
- A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value.
- The hash table can be implemented with the help of an array.
- The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

Hashing and Hash Table

- In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.
- The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:

Index = hash(key)



Hash Functions

- **There are three ways of calculating the hash function:**
- **Division method**
- **Folding method**
- **Mid square method**

Division Hash Function

- In the division method, the hash function can be defined as:
  - **$h(k_i) = k_i \% m;$**
- where **m** is the size of the hash table.
- For example, if the key value is 6 and the size of the hash table is 10. When we apply the hash function to key 6 then the index would be:
- h(6) = 6%10 = 6
- The index is 6 at which the value is stored.

Mid-Square Hash Function.

- A good hash function for numerical values is the **mid-square** method.
- The mid-square method squares the key value, and then takes the middle *r* digits of the result.
- For example, consider records whose keys are 4-digit numbers in base 10. The goal is to hash these key values to a table of size 100 (i.e., a range of 0 to 99). This range is equivalent to two digits in base 10. That is, $r = 2$.
- If the input is the number 4567, squaring yields an 8-digit number, 20857489.
- The middle two digits of this result are 57.

Folding Hash Function

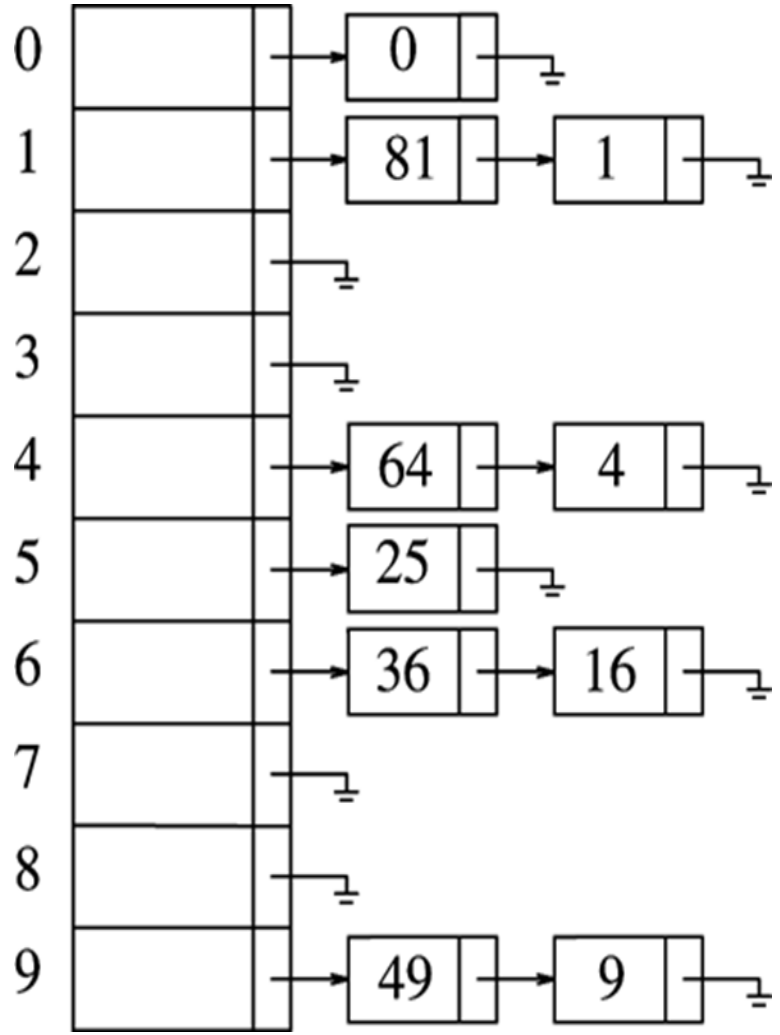
- The **folding method** for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size).
- These pieces are then added together to give the resulting hash value.
- For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43,65,55,46,01).
- After the addition, 43+65+55+46+0143+65+55+46+01, we get 210.
- If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder.
- In this case 210 % 11. 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1.
- Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get 43+56+55+64+01=219 which gives 219 % 11=10219 % 11=10.

Collision in Hashing

- **When the two different values have the same value, then the problem occurs between the two values, known as a collision.**
- For example, the value 16 is stored at index 6. If the key value is 26, then the index would be:
- h(26) = 26%10 = 6
- Therefore, two values are stored at the same index, i.e., 6, and this leads to the collision problem.
- To resolve these collisions, we have some techniques known as collision techniques.
- The following are the collision techniques:
  - **Closed addressing : It is also known as Open Hashing.**
  - **Open Addressing: It is also known as Closed Hashing.**

Closed Addressing: Separate Chaining

- Consider the data
  81,25,36,49,16, 0, 9, 64,1,4
- Hash function
  H(X)=X mod 10
- Each table entry stores a list of Items. So the problem of multiple keys mapped to same entry can be Solved.
- This need pointer, which makes algorithm slow.



Open Addressing

- If a collision occurs, alternative cells are tried until an empty cell is found.
- i.e., cells $h_0(X), h_1(X), \ldots$ are tried in sequence where $h_i(X) = (h(X) + F(i)) \% TS$ with $F(0) = 0$.

- Three techniques are used to resolve the collision:
  - Linear probing
  - Quadratic probing
  - Double Hashing technique

Open Addressing: Linear Probing

- In linear probing F(i)=I

  Insert 89, 18, 49, 58, 69

- $h_i(X)=(h(X)+F(i))\%TS$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Open Addressing: Linear Probing

- First collision occurs with 49
- It is placed in next available location that is 0 which is open
- 58 colloids with 18,89 and then 49
- 69 can also be handled similarly
- Any key that hashes into the cluster will require several attempts to resolve the collision. This is called **primary clustering**.

Open Addressing: Quadratic Probing

- In quadratic probing F(i)= i^2

  Insert 89, 18, 49, 58, 69

  $h_i(X)=(h(X)+F(i))\%TS$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Open Addressing: Quadratic Probing

- When 49 collides with 89, the next position attempted is one cell away. This cell is empty, so 49 is placed there.
- 58 collides at 8 in the next location(9) another collision occurs.
- 2^2=4 so 58 is placed in cell 2
- Same thing happened for 69
- Quadratic probing solves the problem of **primary clustering**.
- Limitation: at most half of the table can be used as alternative locations to resolve collisions.(**secondary clustering)**

Open Addressing: Double Hashing

- In double hashing F(i)= i. $hash_2(X)$ $h_i(X)=(h(X)+F(i))\%TS$
- $hash_2(X)$ =R-(X mod R) where R is prime number smaller than TS

Insert 89, 18, 49, 58, 69

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Open Addressing: Double Hashing

- First collision occurs when 49 is inserted

- *$hash_2(49) = 7 – (49 \% 7)=7$*
- *Hence h(49)= (*$9+7)\%10$
- *so 49 is inserted at position 6*

- *58 inserted at position 3*
- *$hash_2(69)=1$*
- *$hash_2(60)=3$*

- Double hashing is the most efficient collision technique, when the size of the table is prime number and it avoids clustering

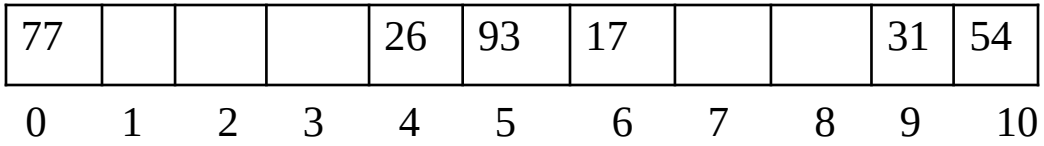
Rehashing

- Hash Table may get full
  - No more insertions possible
- Hash table may get *too* full
  - Insertions, deletions, search take longer time
- Solution: Rehash
  - Build another table that is twice as big and has a new hash function
  - Move all elements from smaller table to bigger table

Rehashing

- If table gets too full, the running time for the operations will take long time and insert may fail using open addressing.
- A solution is to build another table that is twice as big that original table and map data values using new hash function.
- This entire process is called rehashing.
- Rehashing is very expensive
- Rehash can be done in following cases
  - If insertion fails
  - When table is half full
  - Reaches certain load factor

Load factor



$$Load\ \ factor\ \lambda = \frac{Number\ of\ \ data\ values}{Table\ size}$$

$$Load\ \ factor\ \lambda = \frac{6}{11}$$

Rehashing

Data:
13,15,24
and 6
Table size:7
H(x)=xmod7

Original
Hash Table

0	6
1	15
2	
3	24
4	
5	
6	13

After 23 is Inserted :70% of table is full

0	6
1	15
2	23
3	24
4	
5	
6	13

First prime
number twice as
large than old
table size is 17

New has function
H(x)=xmod17

After Rehashing
table is like below

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Extendable hashing

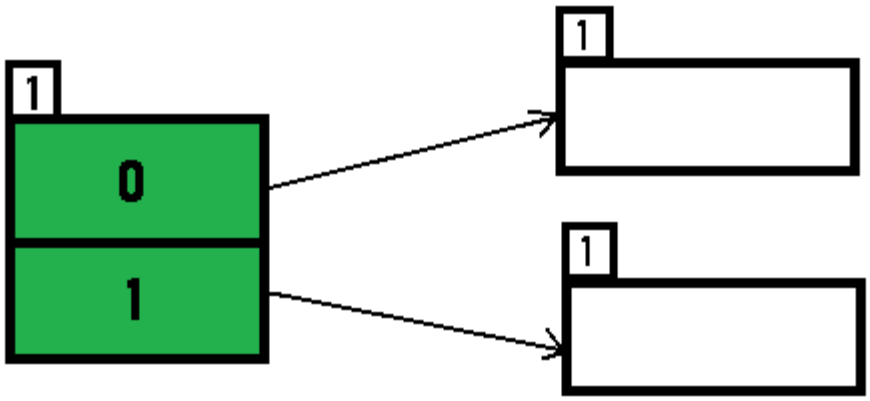
- If either open addressing or closed addressing is used , the major problem is collision.
- Several locations need to be examined .
- Furthermore when the table gets too full an extremely expensive rehashing must be performed.
- A clever alternative is known as extendible hashing.

Example for Extendible Hashing

- let us consider a prominent example of hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**
  Bucket Size: 3 (Assume)
- First, calculate the binary forms of each of the given numbers.
  16- 10000
  4- 00100
  6- 00110
  22- 10110
  24- 11000
  10- 01010
  31- 11111
  7- 00111
  9- 01001
  20- 10100
  26- 01101

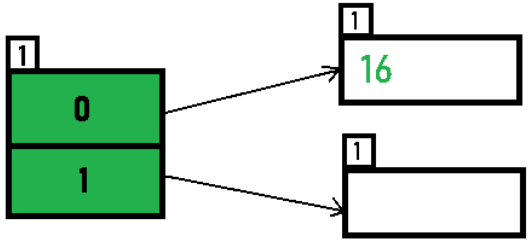
Example for Extendible Hashing

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



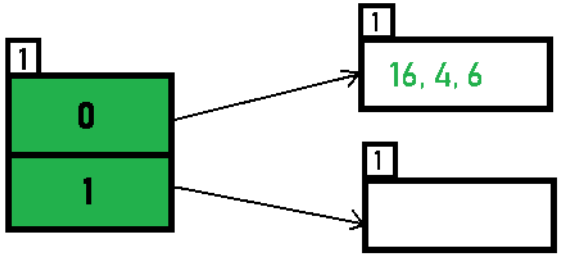
Example for Extendible Hashing

- **Inserting 16:**
  The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.



Hash(16)= 10000

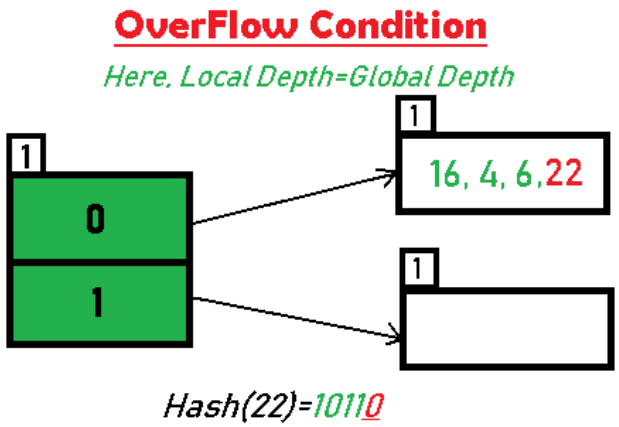
- **Inserting 4 and 6:**
  Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as follows:



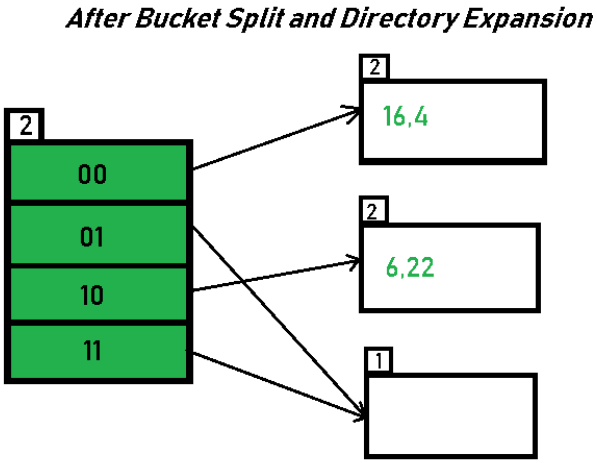
Hash(4)=100
Hash(6)=110

Example for Extendible Hashing

- **Inserting 22:** The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.

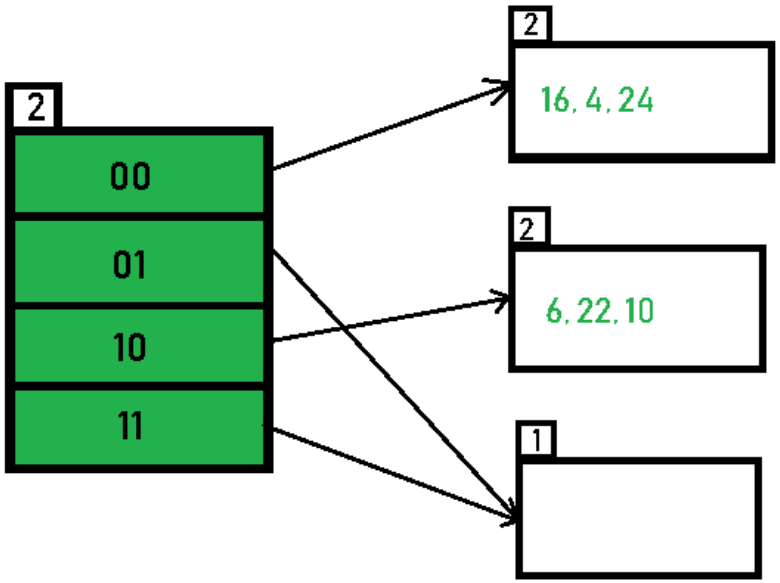


- Since Local Depth = Global Depth, the bucket splits and directory expansion takes place.



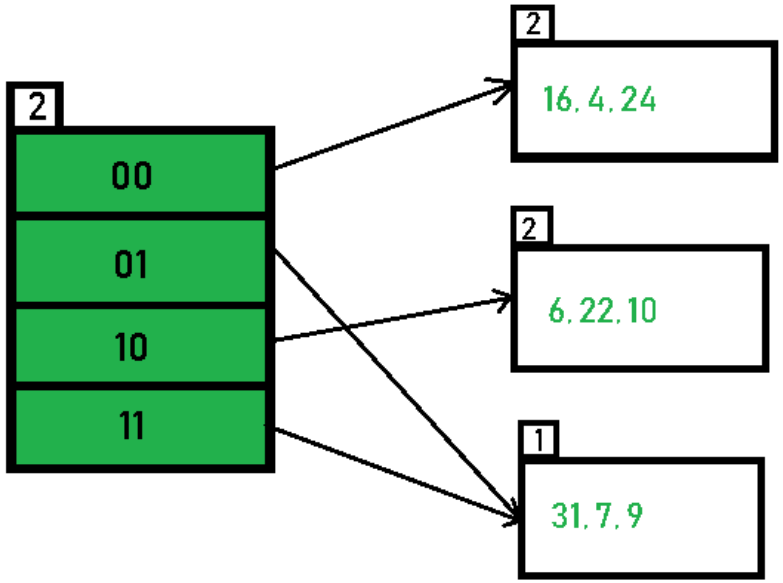
Example for Extendible Hashing

- **Inserting 24 and 10:** 24(11**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



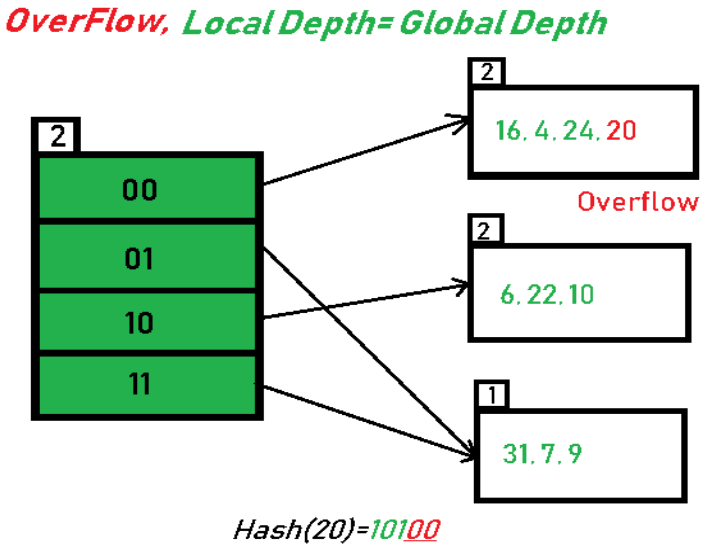
Example for Extendible Hashing

- **Inserting 31,7,9:** All of these elements[31(11**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. We do not encounter any overflow condition here.

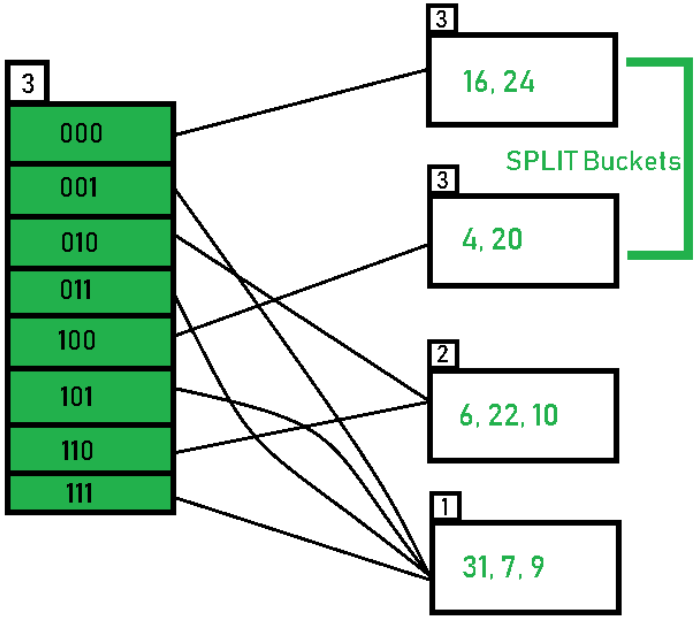


Example for Extendible Hashing

- **Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.

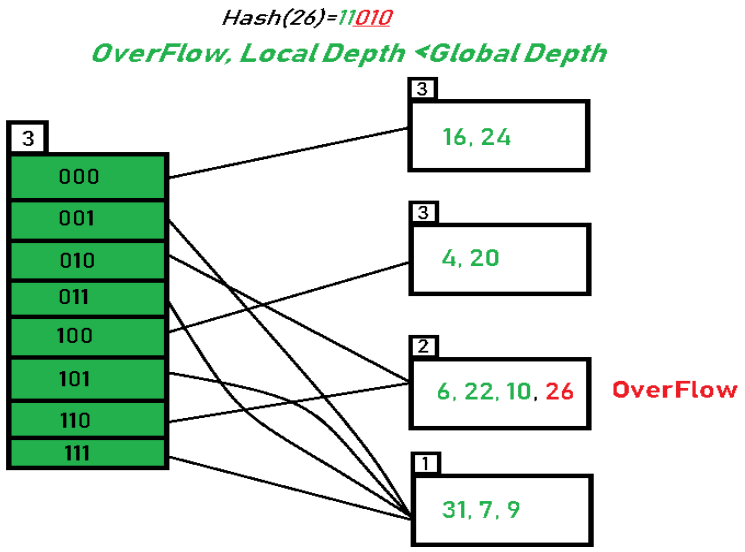


- since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:

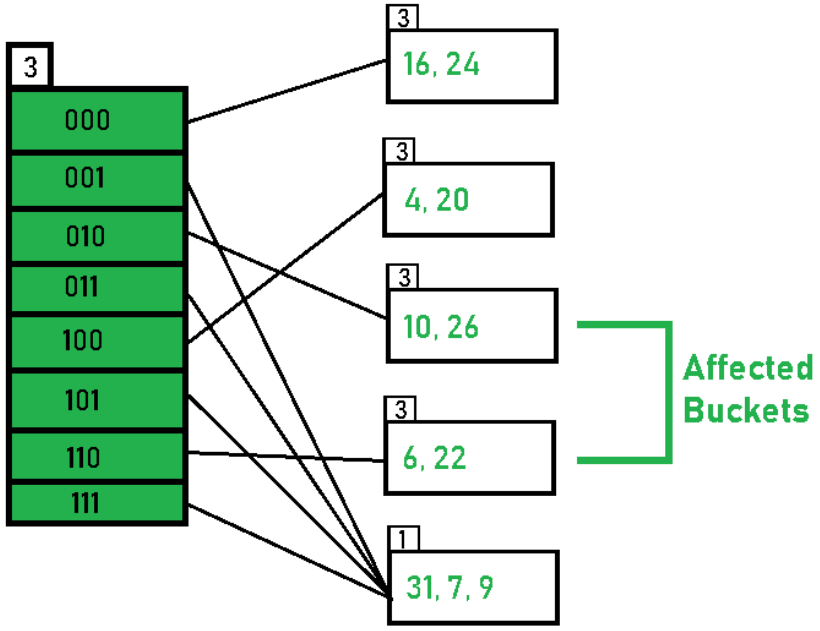


Example for Extendible Hashing

- **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.



- since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed.



Extendible Hashing

- **Limitations Of Extendible Hashing:**
- The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
- Size of every bucket is fixed.
- Memory is wasted in pointers when the global depth and local depth difference becomes drastic.
- This method is complicated to code.