Unity 5 Suman Pant ①

Imp

Hashing

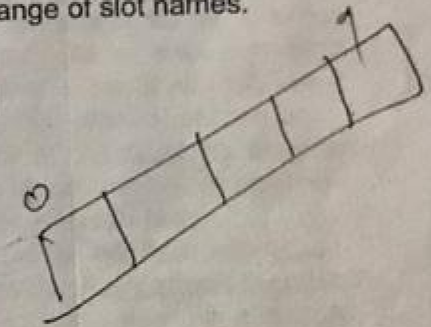
If the array is *not* sorted, a search requires O(n) time.
If the array is sorted, a binary search requires O(log n) time
If the array is organized using *hashing* then it is possible to have constant time search: O(1).

- A **hash table** is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table is called a **slot**, can hold an item and is named by an integer value starting at 0.
- For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty.
- Figure shows a hash table of size m=11. In other words, there are *m* slots in the table, named 0 through 10.

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

- The mapping between an item and the slot where that item belongs in the hash table is called the **hash function**. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and *m*-1. Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31.
- First hash function, known as the "remainder method," simply takes an item and divides it by the table size, returning the remainder as its hash value (h(item)=item%11h(item)=item%11). Below table gives all of the hash values for our example items. Note that this remainder method (modulo arithmetic) will typically be present in some form in all hash functions, since the result must be in the range of slot names.

Key%10



Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

- Once the hash values have been computed, we can insert each item into the hash table at the designated position as shown in below figure. Note that 6 of the 11 slots are now occupied. This is referred to as the **load factor**, and is commonly denoted by λ=numberofitems/tablesize
For this example, λ=6/11.

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

- Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present. This searching operation is O(1)O(1), since a constant amount of time is required to compute the hash value and then index the hash table at that location. If everything is where it should be, we have found a constant time search algorithm.
- You can probably already see that this technique is going to work only if each item maps to a unique location in the hash table. For example, if the item 44 had been the next item in our collection, it would have a hash value of 0 (44%11==0). Since 77 also had a hash value of 0, we would have a problem. According to the hash function, two or more items would need to be in the same slot. This is referred to as a **collision** (it may also be called a "clash"). Collisions create a problem for the hashing technique..

Hash Functions-

- Given a collection of items, a hash function that maps each item into a unique slot is referred to as a **perfect hash function**. If we know the items and the collection will never change, then it is possible to construct a perfect hash
- One way to always have a perfect hash function is to increase the size of the hash table so that each possible value in the item range can be accommodated. This guarantees that each item will have a unique slot. Although this is practical for small numbers of items, it is not feasible when the number of possible items is large. For example, if the items were nine-digit Social Security numbers, this method would require almost one billion slots. If we only want to store data for a class of 25 students, we will be wasting an enormous amount of memory.
- Our goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the items in the hash table. There are a number of common ways to extend the simple remainder method. We will consider a few of them here.
- 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+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 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=10.
- Another numerical technique for constructing a hash function is called the **mid-square method**. We first square the item, and then extract some portion of the resulting digits. For example, if the item were 44, we would first compute $44^2=1,936$ .By extracting the middle two digits, 93, and performing the remainder step, we get 5 (93 % 11).

Below table shows items under both the remainder method and the mid-square method.

Item	Remainder	Mid-Square
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

A good hash function should:

1. Minimize collisions.
2. Be easy and quick to compute.
3. Distribute key values evenly in the hash table.
4. Use all the information provided in the key.

Imp

Collision Resolution

When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called **collision resolution**. If the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing.

One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty. Note that we may need to go back to the first slot (circularly) to cover the entire hash table. This collision resolution process is referred to as **open addressing** in that it tries to find the next open slot or address in the hash table. By systematically visiting each slot one at a time, we are performing an open addressing technique called **linear probing**.

Below Figure shows an extended set of integer items under the simple remainder method hash function (54,26,93,17,77,31,44,55,20). Table above shows the hash values for the original items. Below Figure shows the original contents. When we attempt to place 44 into slot 0, a collision occurs. Under linear probing, we look sequentially, slot by slot, until we find an open position. In this case, we find slot 1.

Again, 55 should go in slot 0 but must be placed in slot 2 since it is the next open position. The final value of 20 hashes to slot 9. Since slot 9 is full, we begin to do linear probing. We visit slots 10, 0, 1, and 2, and finally find an empty slot at position 3.

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

Once we have built a hash table using open addressing and linear probing, it is essential that we utilize the same methods to search for items. Assume we want to look up the item 93. When we compute the hash value, we get 5. Looking in slot 5 reveals 93, and we can return `True`. What if we are looking for 20? Now the hash value is 9, and slot 9 is currently holding 31. We cannot simply return `False` since we know that there could have been collisions. We are now forced to do a sequential search, starting at position 10, looking until either we find the item 20 or we find an empty slot.

A disadvantage to linear probing is the tendency for **clustering**; items become clustered in the table. This means that if many collisions occur at the same hash value, a number of surrounding slots will be filled by the linear probing resolution. This will have an impact on other items that are being inserted, as we saw when we tried to add the item 20 above. A cluster of values hashing to 0 had to be skipped to finally find an open position. This cluster is shown in below figure.

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

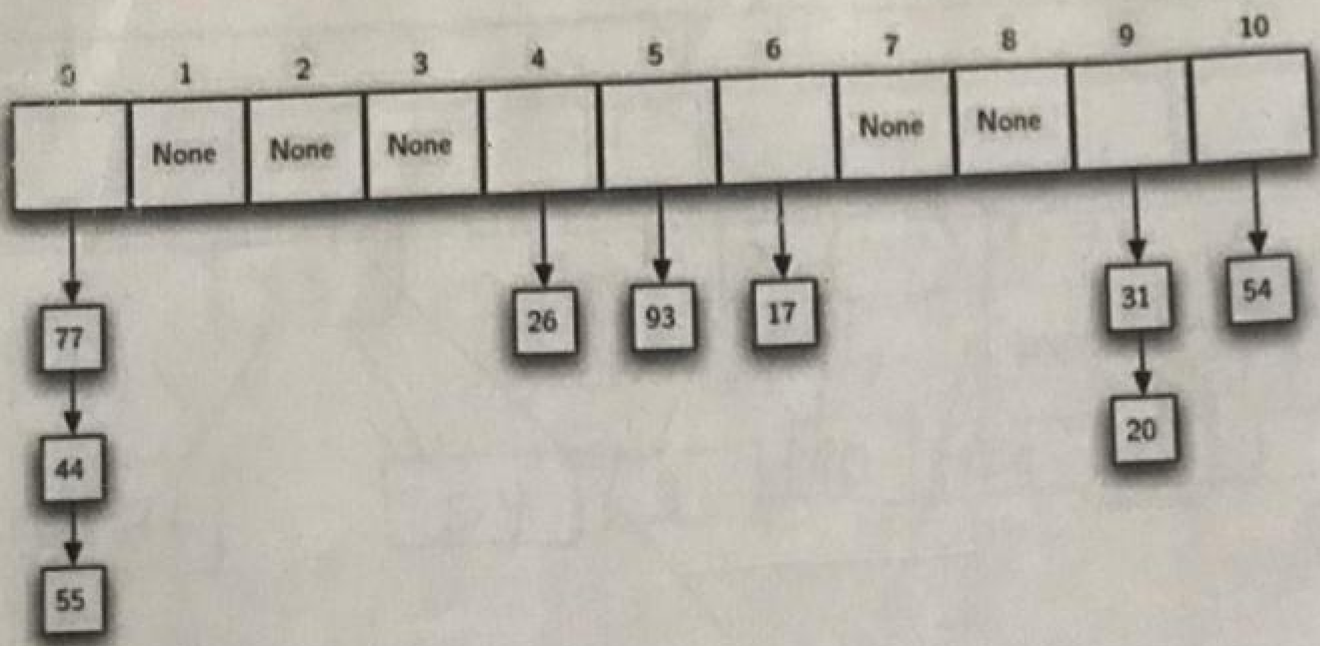
One way to deal with clustering is to extend the linear probing technique so that instead of looking sequentially for the next open slot, we skip slots, thereby more evenly distributing the items that have caused collisions. This will potentially reduce the clustering that occurs. Below figure shows the items when collision resolution is done with a "plus 3" probe. This means that once a collision occurs, we will look at every third slot until we find one that is empty.

0	1	2	3	4	5	6	7	8	9	10
77	55	None	44	26	93	17	20	None	31	54

A variation of the linear probing idea is called **quadratic probing**. Instead of using a constant "skip" value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9, and so on. This means that if the first hash value is *h*, the successive values are $h+1$, $h+4$, $h+9$, $h+16$, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares. Figure shows our example values after they are placed using this technique.

0	1	2	3	4	5	6	7	8	9	10
77	44	20	55	26	93	17	None	None	31	54

An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items. **Chaining** allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. As more and more items hash to the same location, the difficulty of searching for the item in the collection increases. Below figure shows the items as they are added to a hash table that uses chaining to resolve collisions.



When we want to search for an item, we use the hash function to generate the slot where it should reside. Since each slot holds a collection, we use a searching technique to decide whether the item is present. The advantage is that on the average there are likely to be many fewer items in each slot, so the search is perhaps more efficient.

Hashing

Different hashing functions to generate the value are as follows:

1) **Division method:**

Hash function can be given as
$$h(x) = x \bmod M$$

Q1) Calculate the hash values of keys 1234 & 5462

Sol Setting $m = 97$ (not too close to the exact powers of 2)

hash values can be calculated as

$h(1234) = 1234 \% 97 = 70$.

$h(5642) = 5642 \% 97 = 16$

2) ~~Multiplication~~ **Mid-square method:**

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

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

Q1) Calculate the hash value for keys 1234 & 5642 using the mid-square method. The hash table has 100 memory locations.

Sol: Memory locations = 100
$\therefore$ 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 $k=1234$, $k^2 = 15\,227\,56$ $r=2$ $227/10$.
227 but $r \neq 2$
$\therefore$ $227 \% 100 = 27$
When $k=5642$, $k^2 = 318\,32\,164$.
$\therefore$ $h(5642) = 32$.

Or choose 3rd & 4th digits starting from the right
then $h(1234) = 27$
and $h(5642) = 32$

(iii) Folding Method:
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 $k_1 + k_2 + — + k_n$.

(8) The hash value is produced by ignoring the last carry, if any.

(Q1) Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321 & 34567.

Sol Memory locations = 100
Memory : 0 – 99 (so each part of key must have 2 digits except the last part)

(a) Key ⇒ 5678
Parts ⇒ 56, 78
Sum ⇒ 56 + 78 = 134
Hash value ⇒ Ignore the last carry (34)

(b) Key = 321
Parts = 32, 1
Sum = 32 + 1
= 33
Hash Value = 33

(c) Key = 34567
Parts = 34, 56, 7
Sum = 34 + 56 + 7
= 97
Hash value = 97

Collisions :-

Collision Resolution techniques.

→ Open Addressing

→ Chaining

Once a collision takes place, open addressing finds new positions using a probe sequence & the next record is stored in that position.

Hash table contains two types of value

→ Sentinel values (-1) → means that location contains no data value at present & can be used to hold a value when collision occurs.

→ Data Values.

Imp

The process of examining memory locations in the hash table is called probing.

Open Addressing can be implemented using linear probing, quadratic probing, double hashing & rehashing.

Linear Probing :- If a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision.

(10)

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

$m \downarrow$ size of hash table

Where $h'(k) = (k \bmod m)$

i = probe number that varies from 0 to $m-1$.

<u>Ex</u> Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92 in to the table.

<u>Sol</u>

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

<u>Step1</u> ÷ key = 72

$h(72, 0) = [72 \bmod 10 + 0] \bmod 10$

$= [2 + 0] \bmod 10$

$= 2$

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

<u>Step2</u> ÷ key = 27

$h(27, 0) = (27 \bmod 10 + 0) \bmod 10$

$= (7) \bmod 10$

$= 7$

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

Step 3 ÷ key = 36
$h(36, 0) = (36 \bmod 10 + 0) \bmod 10$
$6 \bmod 10$
$= 6$

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

Step 4 ÷ key = 24
$h(24, 0) = (24 \bmod 10 + 0) \bmod 10$
$= (4+0) \bmod 10$
$= 4$

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

Steps: key = 63
$h(63, 0) = (63 \bmod 10 + 0) \bmod 10$
$= 3$

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

Step 6 ÷ key = 81
$h(81, 0) = (81 \bmod 10 + 0) \bmod 10$
$= (1) \bmod 10$
$= 1$

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

Step 7: key = 92
$h(92, 0) = (92 \bmod 10 + 0) \bmod 10$
$= 2$

but (2) location is occupied, so find next empty location, by taking $i = 1$

$h(92, 1) = (92 \bmod 10 + 1) \bmod 10$
$= (2+1) \bmod 10 = 3$

but 3rd location is also occupied then find next empty location by taking $i = 2$

Q1) Consider a hash table of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81 & 101

Taking $c_1 = 1$, $c_2 = 3$

Sol

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

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

Step 1

key = 72

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

$$= 2 \bmod 10 = 2$$

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

Step 2 : key = 27

$$h(27, 0) = [27 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$$

$$= 7 \bmod 10 = 7$$

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

Step 3 : key = 36

$$h(36, 0) = [36 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$$

$$= 6 \bmod 10 = 6$$

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

Step 4 : key = 24

$$h(24, 0) = [24 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$$

$$= [24 \bmod 10] \bmod 10 = 4 \bmod 10 = 4$$

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

Step5 : Key = 63

$h(63,0) = [63 \bmod 10 + 1\times0 + 3\times0] \bmod 10$

$= 3 \bmod 10 = 3$

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

Step6 Key = 81

$h(81,0) = [81 \bmod 10 + 1\times0 + 3\times0] \bmod 10$

$= [81 \bmod 10] \bmod 10 = 1$

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

Step7 : key = 101

$h(101,0) = [101 \bmod 10 + 1\times0 + 3\times0] \bmod 10$

$= 1 \bmod 10 = 1$

1 location is already occupied, then find next empty location. by taking $i=1$

Key = 101

$h(101,1) = [101 \bmod 10 + 1\times1 + 3\times1] \bmod 10$

$= [1+1+3] \bmod 10 = 5 \% 10$

$= 5$

5^{th} location is empty

∴ 101 is inserted at 5^{th} location.

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

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 locations.

Q1 Insert the keys 7, 24, 18, 52, in Chained hash tabl of 9 memory locations. Use $h(k) = k \bmod m$ +10

K mod m +10

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

Sol

$m = 9$

0 to 8

Step 1 :- 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	

Step 2 :- 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	

Step 3 :- 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	

Step 4 :- Key = 52 $h(k) = 52 \bmod$
$= 7$

0		→ [18 | X]
1	NULL	
2	NULL	
3	NULL	
4	NULL	
5	NULL	
6		→ [24 | X]
7		→ [7 | ~~X~~] → [52 | X]
8	NULL	

(12)

$h(92,2) = (92 \bmod 10 + 2) \bmod 10$
$= 4$

But 4th location is again occupied, find next by taking $i = 3$

$h(92,3) = (92 \bmod 10 + 3) \bmod 10$
$= (2+3) \bmod 10$
$= 5$

Now 5th location is empty so 92 can be inserted at 5th location

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

Ans

(13)

Quadratic Probing :- to remove the problem of clustering (where there is a higher risk of more collisions where one collision has already taken place) quadratic probing is used.

Hash function used :-

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

where $h'(k) = (k \bmod m)$

$i = 0$ to $m-1$

c_1 & c_2 are constants & not equal to 0

Q1) Consider a hash table of size 16. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81 & 101 Taking $c_1 = 1$, $c_2 = 3$

Sol

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

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

Step 1

key = 72

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

$= 2 \bmod 10 = 2$

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

Step 2 : key = 27

$h(27, 0) = [27 \bmod 10 + 1\times 0 + 3\times 0] \bmod 10$

$= 7 \bmod 10 = 7$

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

Step 3 : key = 36

$h(36, 0) = [36 \bmod 10 + 1\times 0 + 3\times 0] \bmod 10$

$= 6 \bmod 10 = 6$

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

Step 4 : key = 24

$h(24, 0) = [24 \bmod 10 + 1\times 0 + 3\times 0] \bmod 10$

$= [24 \bmod 10] \bmod 10 = 4 \bmod 10 = 4$

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

(17)

Step 5 :- Key = 63

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

$= 3 \bmod 10 = 3$

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

Step 6 Key = 81

$h(81, 0) = [81 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$

$= [81 \bmod 10] \bmod 10 = 1$

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

Step 7 :- key = 101

$h(101, 0) = [101 \bmod 10 + 1 \times 0 + 3 \times 0] \bmod 10$

$= 1 \bmod 10 = 1$

1 location is already occupied, then find next empty location by taking i = 1

Key = 101

$h(101, 1) = [101 \bmod 10 + 1 \times 1 + 3 \times 1] \bmod 10$

$= [1 + 1 + 3] \bmod 10 = 5 \% 10$

$= 5$

5^{th} location is empty

∴ 101 is inserted at 5^{th} location.

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

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 locations.

Q1 Insert the keys 7, 24, 18, 52 in Chained hash tabl of 9 memory locations. Use $h(k) = k \bmod m$ +10

K mod m +10

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

Sol

$m = 9$

0 to 8

Step 1 ÷ 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

Step 2 ÷ 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

Step 3 ÷ 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

Step 4 ÷ Key = 52 $h(k) = 52 \bmod$
$= 7$

0	→ [18 | X]
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	→ [24 | X]
7	→ [7 | ~~X~~] → [52 | X]
8	NULL