HASHING

HASH TABLE ADT

HASHING

A technique used for performing insertions, deletions, and finds in constant average time.

HASHING

Not supported:

- find_min
- find_max
- print_sorted

HASH TABLE

An array of fixed size, containing the keys.

HASH TABLE

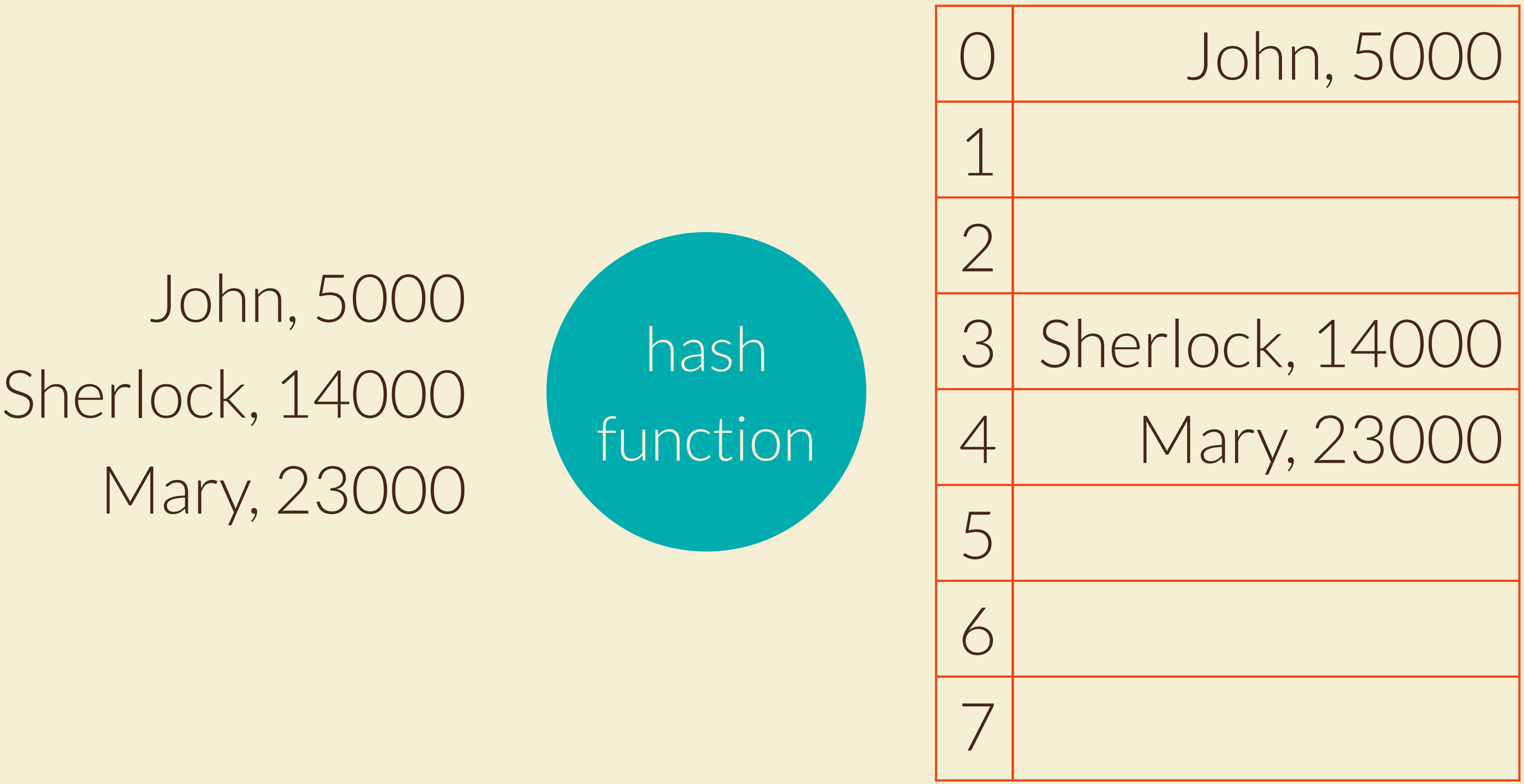
0	
1	
2	
3	
4	
5	
…	
hSize-1	

HASH TABLE

H[0]	
H[1]	
H[2]	
H[3]	
H[4]	
H[5]	
…	
H[hSize]	

HASHING

Map the input keys to the indices of hash table using a hash function.


John, 5000
Sherlock, 14000
Mary, 23000
hash
function
0 John, 5000
1
2
3 Sherlock, 14000
4 Mary, 23000
5
6
7

HASH FUNCTION

hTable[hFunction(key)] = key

HASH FUNCTION

- Should be simple to compute.
- Should ensure that two distinct keys get different cells.

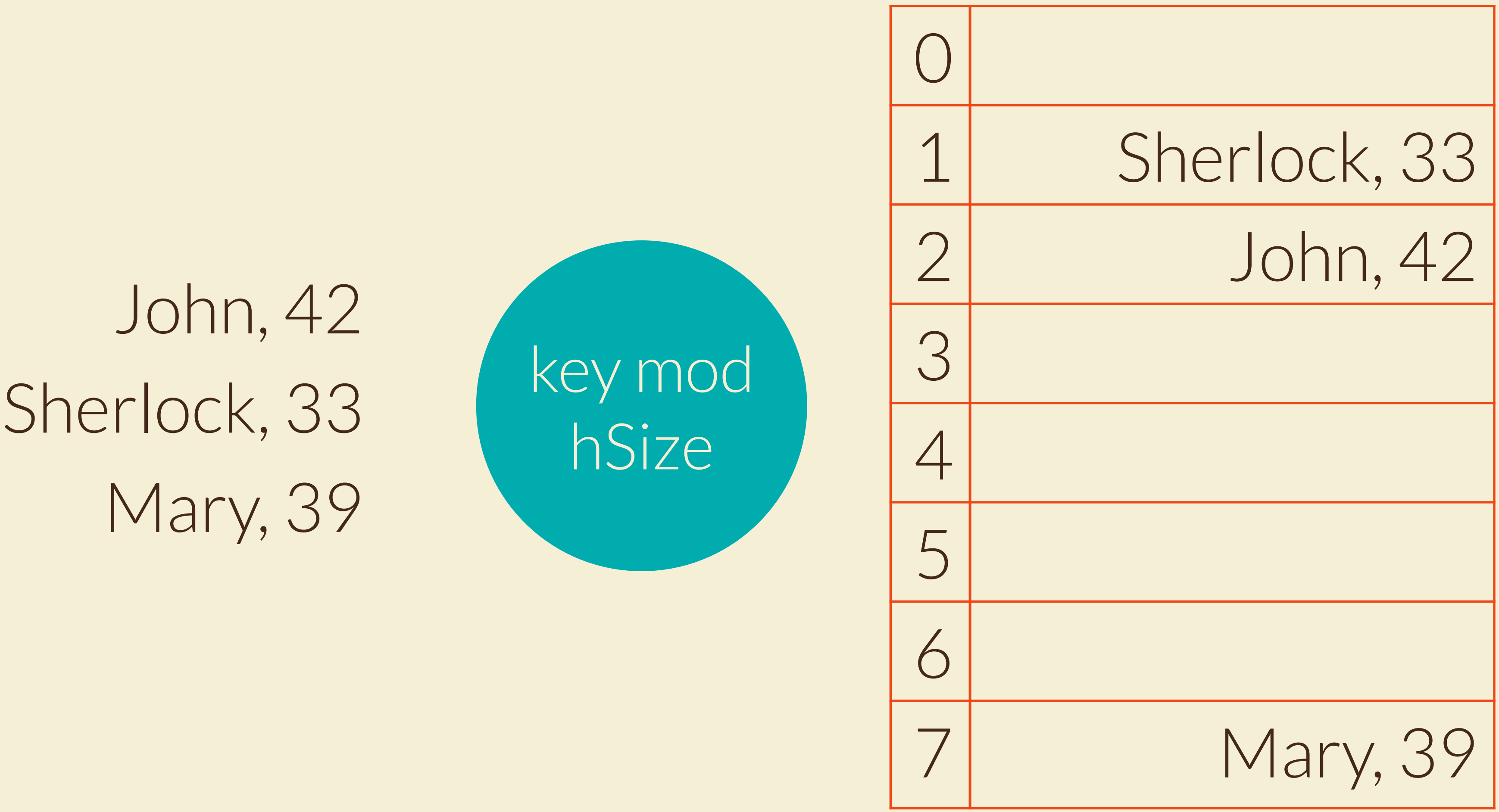
HASH FUNCTION

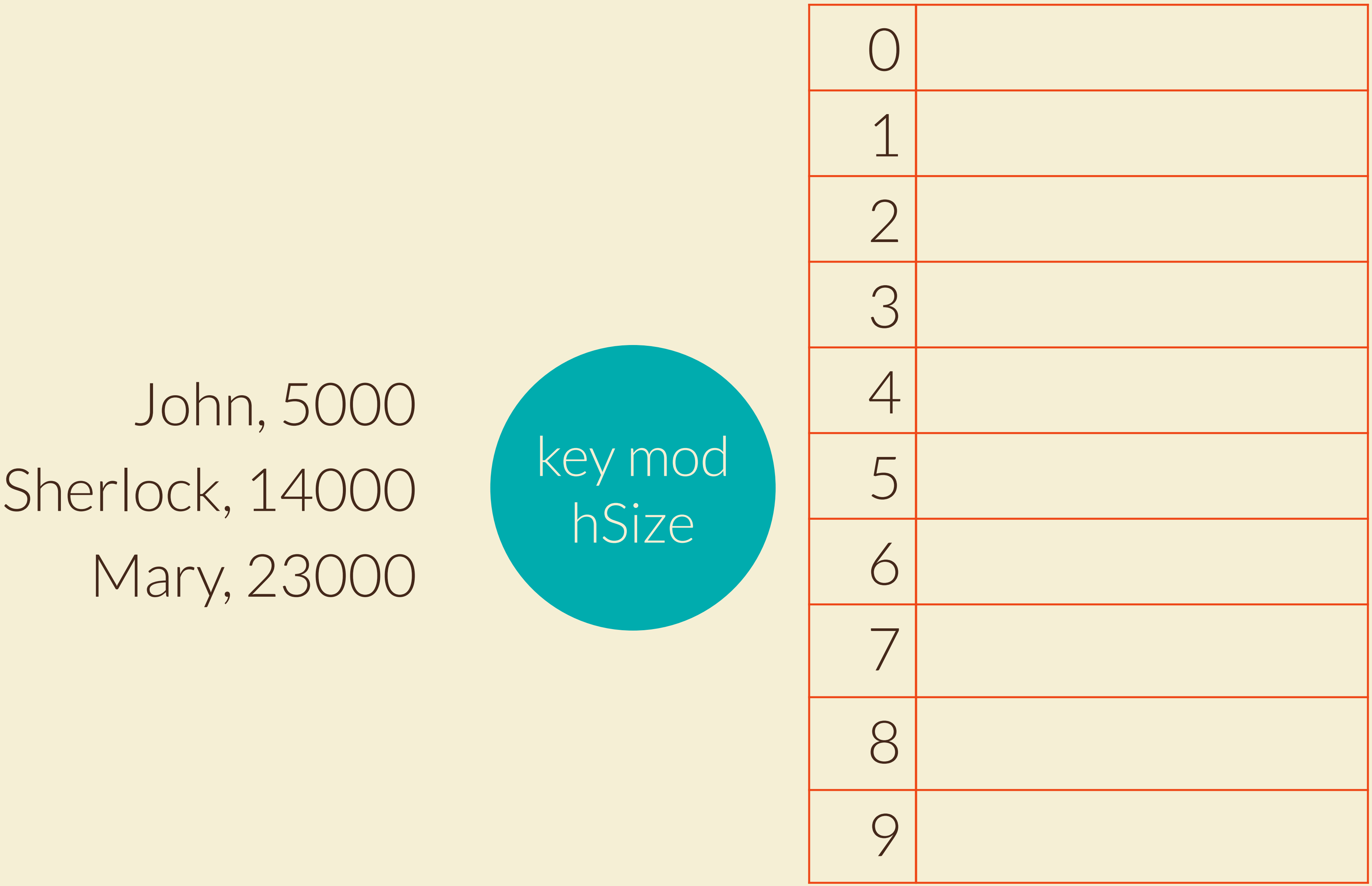
- Choosing a hash function
- Deciding what to do when two keys hash to the same value (collision).
- Table size.

choosing a good
HASH FUNCTION

If the input keys are integers:

hFunction(key) = key mod hSize

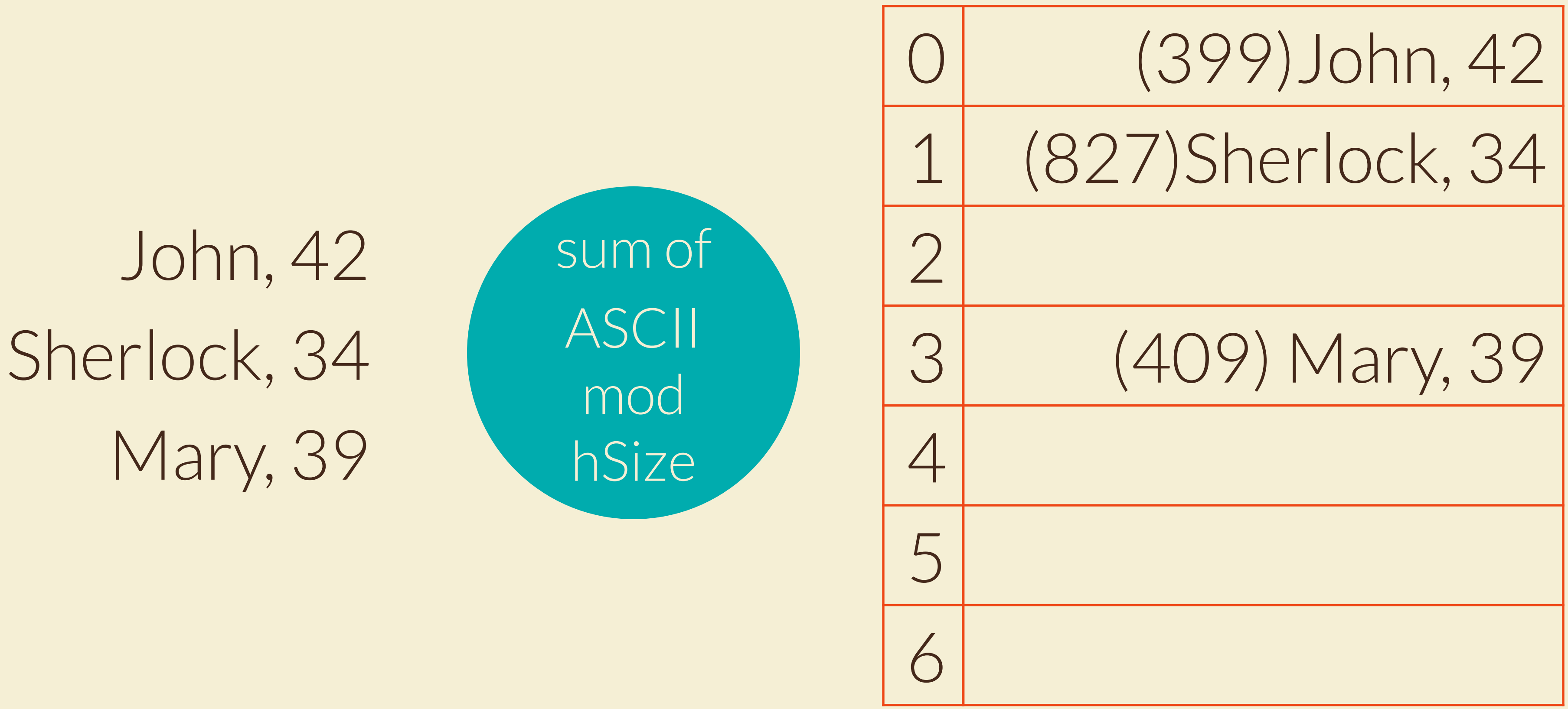

John, 42
Sherlock, 33
Mary, 39
key mod hSize
0
1
Sherlock, 33
2
John, 42
3
4
5
6
7
Mary, 39


John, 5000
Sherlock, 14000
Mary, 23000
key mod hSize
0
1
2
3
4
5
6
7
8
9

If the input keys are strings:

hFunction(key) = add up the ASCII values of the characters in the string.

```
int hash(char *key, int hSize){
  int hValue;
  while(*key!='\0')
    hValue += *key++;
  return (hValue % hSize);
}
```


John, 42
Sherlock, 34
Mary, 39
sum of
ASCII
mod
hSize
0
(399)John, 42
1
(827)Sherlock, 34
2
3
(409) Mary, 39
4
5
6

If the input keys are strings:

```
hFunction(key)
    =( ( key[0] + 27*key[1] + 729*key[2] ) % hSize );
```

```
int hash(char *key, int hSize){
  int hValue;
  while(*key!='\0')
    hValue = ( hValue << 5 ) + *key++;
  return (hValue % hSize);
}
```

COLLISION RESOLUTION

DIRECT
ADDRESSING

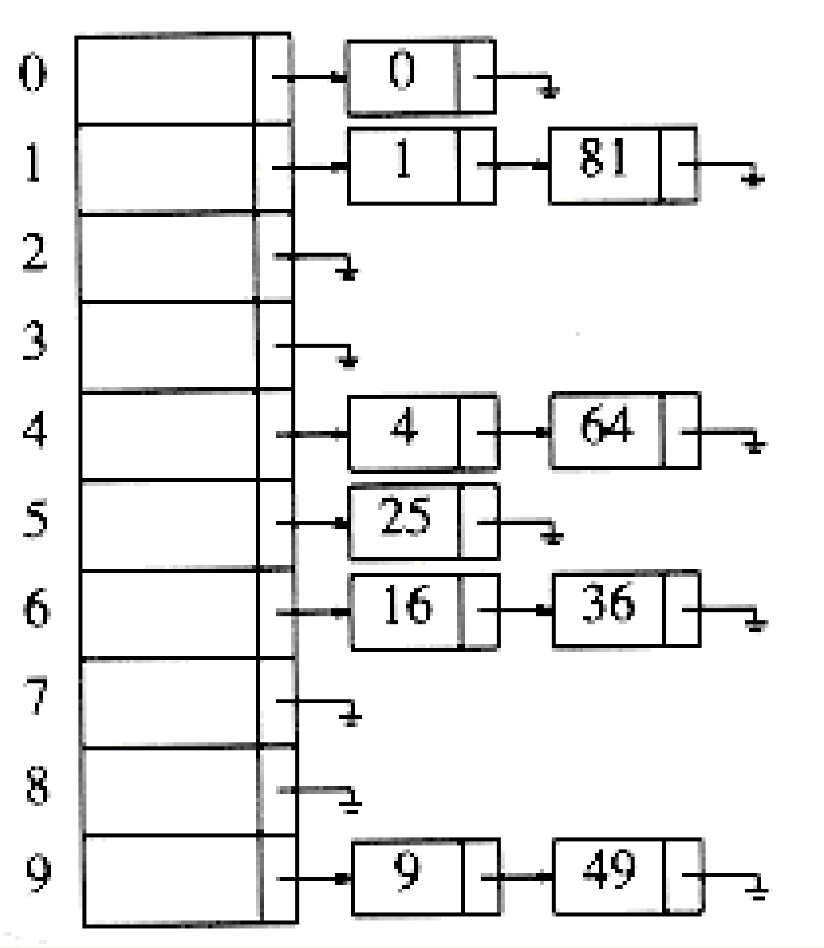
DIRECT ADDRESSING

0	
1	
...	
4	4
...	
22	
23	23
...	
39	39
40	
41	

OPEN CLOSED HASHING

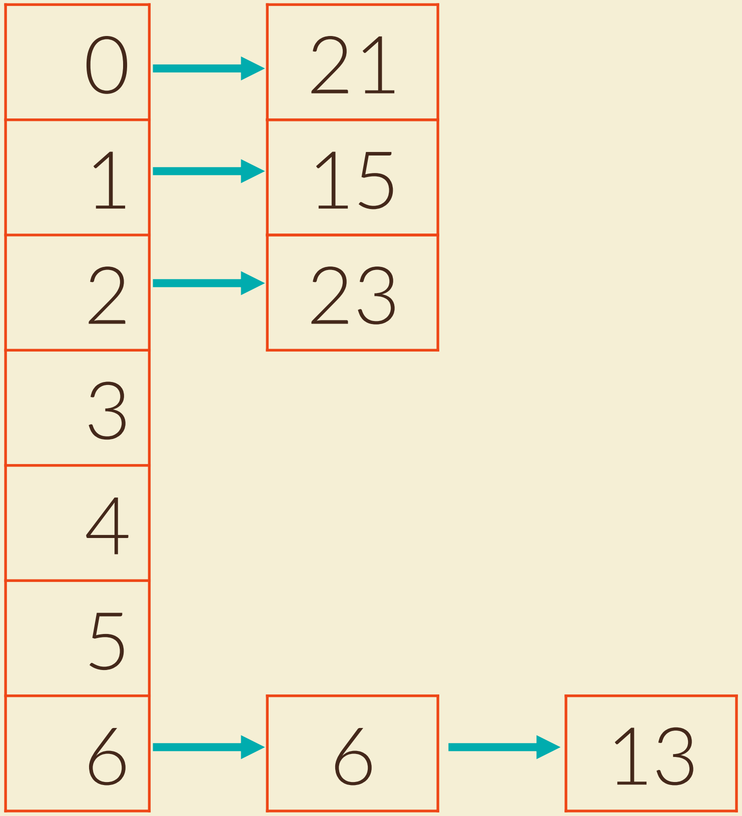
OPEN HASHING

Keep a list of all elements that hash to the same value.


0
1
2
3
4
5
6
7
8
9
0
1
81
4
64
25
16
36
9
49

Insert 6, 15, 23, 21 and 13 to a hash table of size m = 7

key mod hSize



CLOSED HASHING

If a collision occurs, alternate cells are tried until an empty cell is found.

CLOSED HASHING

$h_0(x), h_1(x), \ldots$ are tried in succession where

$$h_i(x) = (hash(x) + f(i)) \bmod hSize$$

CLOSED HASHING

Linear Probing

Quadrating Probing

Double Hashing

LINEAR PROBING

$$(hFunction(x) + f(i))\%hSize$$

f is a linear function of i.

$$f(i) = i$$

i = number of collisions

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key mod hSize) + i) mod hSize

i = number of collisions

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key mod hSize) + i) mod hSize

i = number of collisions

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key mod hSize) + i) mod hSize

i = number of collisions

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key mod hSize) + i) mod hSize

i = number of collisions

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key mod hSize) + i) mod hSize

i = number of collisions

0	49
1	58
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key mod hSize) + i) mod hSize

i = number of collisions

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

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key mod hSize) + i) mod hSize

i = number of collisions

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

LINEAR PROBING

Primary Clustering

Any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

QUADRATIC PROBING

f is a quadratic function of i.

$$f(i) = c_1{}^{*}i + c_2 i^2$$

QUADRATIC PROBING

f is a quadratic function of i.

$$f(i) = i^2$$

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key mod hSize) + i^2) mod hSize

i = number of collisions

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

$$((\text{key mod hSize}) + i^2) \text{ mod hSize}$$

i = number of collisions

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

$$((key \bmod hSize) + i^2) \bmod hSize$$

i = number of collisions

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

$$((\text{key mod hSize}) + i^2) \text{ mod hSize}$$

i = number of collisions

0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size $m = 10$

$((\text{key mod hSize}) + i^2) \text{ mod hSize}$

i = number of collisions

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

QUADRATIC PROBING

Secondary Clustering

Elements that hash to the same position will probe the same alternate cells.

DOUBLE HASHING

If a collision occurs, apply a second hash function to x.

DOUBLE HASHING

$f(i) = i*hash2(x)$

i = number of collisions

DOUBLE HASHING

hash2(x) = R - (x%R)

R = prime smaller than hSize.

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key%hSize) + i*h2(key))%hSize

h2(key) = 7 - (key%7)

i = number of collisions

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key%hSize) + i*h2(key))%hSize

h2(key) = 7 - (key%7)

i = number of collisions

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key%hSize) + i*h2(key))%hSize

h2(key) = 7 - (key%7)

i = number of collisions

0	
1	
2	
3	
4	
5	
6	49
7	
8	18
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key%hSize) + i*h2(key))%hSize

h2(key) = 7 - (key%7)

i = number of collisions

0	
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

Insert 89, 18, 49, 58 and 69 to a hash table of size m = 10

((key%hSize) + i*h2(key))%hSize

h2(key) = 7 - (key%7)

i = number of collisions

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

REHASHING

REHASHING

Build another table that is about twice as big (with associated new hash function).

REHASHING

Scan down the entire original hash table, computing the new hash value for each element and inserting it in the new table.

key % 7

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

Insert 13, 15, 6, 24, and 23

key % 7

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

key % 17 + i

0	
...	
5	
6	6
7	23
8	24
...	
12	
13	13
14	
15	15
16	

Rehash 6, 15, 23, 24 and 13