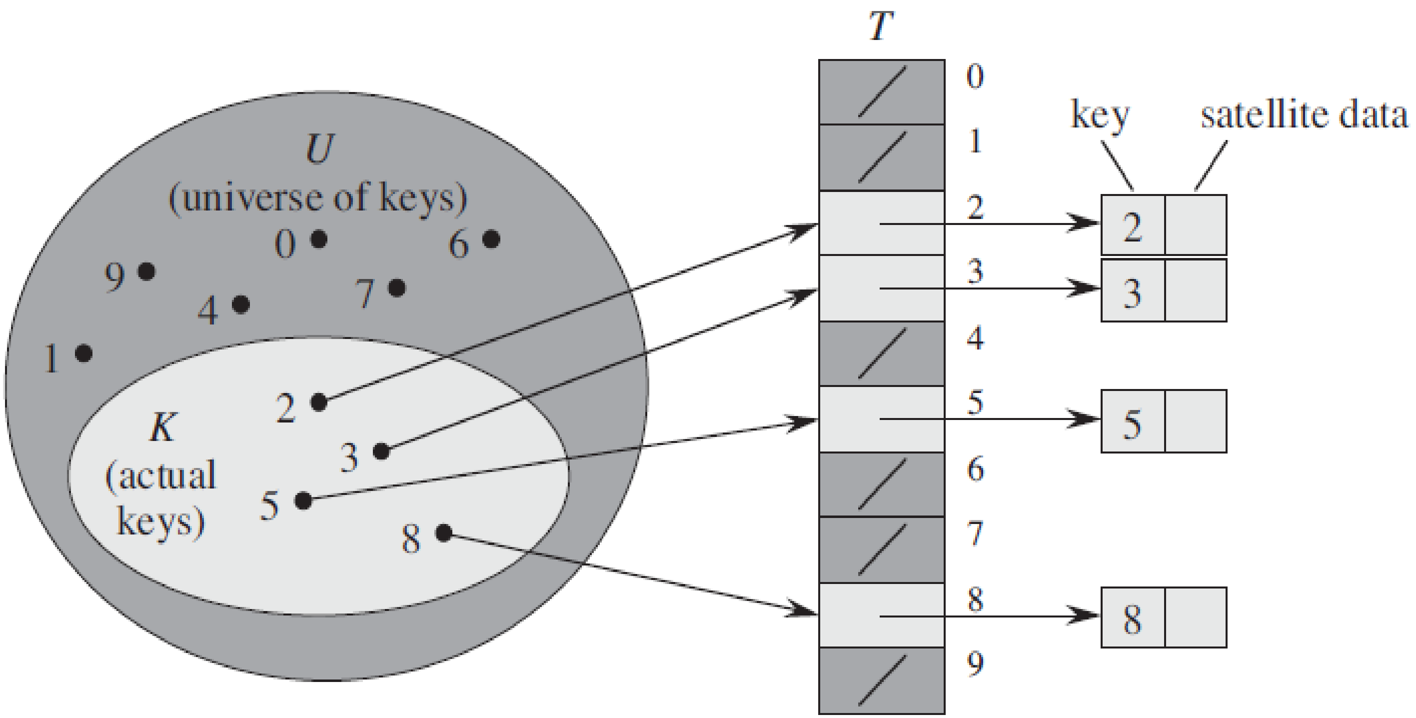
BBM371- Data Management

Lecture 6: Hash Tables

8.11.2018

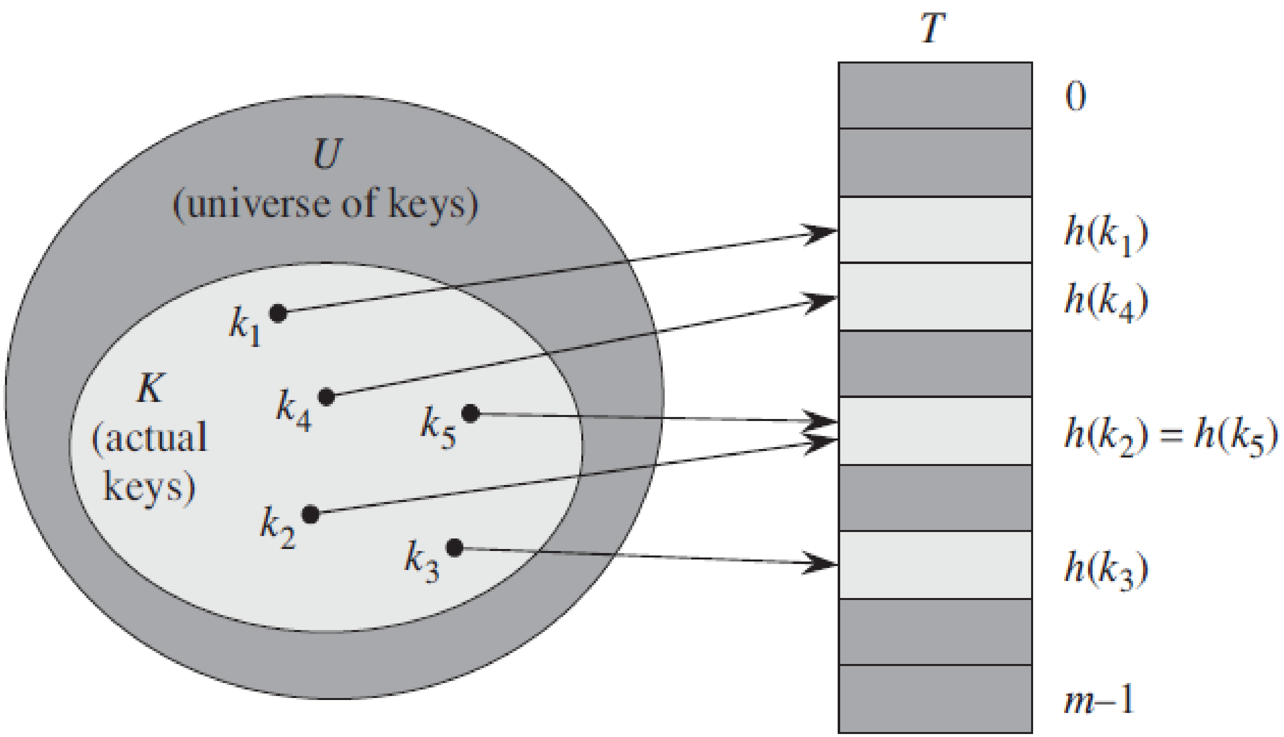
Purpose of using hashes

- A generalization of ordinary arrays:
  - Direct access to an array index is O(1), can we generalize direct access to any key
  - If U is small (e.g. 9) we can do it with a table T.
    - Think of ASCII table to map characters to arrays.



Hash tables

- Used when the set of potential keys U is large and actual used keys K is small
  - Consider storing 5 keys from potential keys between numbers 0 and 1 billion



Hashing

- An element with key k is transformed with a hash function to map to slot h(k) in hash table T.
- h(k) is the hash value of key k
- Hash function reduces the range of key values from |U| to |T|

Note: Now let's think address as record number in a file (or table)

Example

- Hash function : Convert an arbitrary string key in ASCII format and multiply first two character and use rightmost three digit.

H(Lowell) = 4

- Can consider all possible strings as infinite number of keys, we map it to small range.

Hashing with Files

Key → Apply Hash function → Physical Address

h(K) = K mod m
m = 70 - 90% of the expected number of records

Example:

Name	Department	Salary

h(James Adams) = (74+65) mod 17 = 139 mod 17 = 3

	Name	Department	Salary	Overflow Pointer
0				
1				
2				
3	James Adams			-1
15	Mary Jones			-1
16				
17	Henry Truman			-1

Records are mapped to blocks in a file!

Collisions

- Larger set U is mapped to a small set T
- Having multiple-keys mapped to the same slot is called as collision.
- If we add n keys larger than the number of slots we have we must have at least one collision.
- Even if $n < |T|$ and mapping is random we can have collisions.
  - Think of birthday paradox; with more than 23 people in a room probability of having the same birthdate (day, month) is 50%

Good Hash Function

- Generated record numbers should be uniformly and randomly distrubuted over the file (0 <= h(key)< |T|)
- The hashing function must minimize collisions (random distribution)
  - Worst case : Map all keys to slot 0. O(|T|)
- Should be easy to calculate

Load Factor

- Loading factor (LF), $\alpha = n / m$
  n: number of keys
  m: number of slots
- If uniform distribution (1/m) to get mapped to a slot, a slot will have an expectation of α elements.
- If m increases
  - Collision decreases
  - LF decreases
  - 0.5 > LF > 0.8 is unacceptable
  - Storage requirements increases.
- Reduce collisions while keeping storage requirements low.

Hashing Transformations

- Digit analysis

  Use specific digits from key; might not be random

- Division method

  $h_{ab}(k) = ((ak+b) \bmod p) \bmod m)$ p>m and p is prime

- Radix transformation

  $f(abc)= a * 11^2 + b * 11 + c$

Overflow Management Techniques

- Direct organisation of file by using Hashing
  - Open Addressing
    - Linear search
    - Nonlinear search
  - Chaining
- Hash based indexing
  - Extendible Hashing
  - Linear Hashing

Open Addressing

- All elements occupy the hash table itself. (i.e. No pointers or overflow buckets)
- When collision occurs, the new record will be inserted in the first available slot after h(k)
- When searching for available slot, hash table is probed.
- Depending on h(k) different probe sequences from all permutations of 0…m can be used for a slot.
- The probe sequence generation methods determine the performance of hash tables.

Insert Algorithm

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

- $h(k, i)$: returns the i^{th} element of key k's probe sequence.
- If all slots are probed ($i==m$), the hash table is full.

Search Algorithm

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

- Similar code for search.
- If key is not encountered before a NIL value, key is not in T
  - In an edge case all nodes in probe sequence is filled, must check i==m

Linear Probing

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

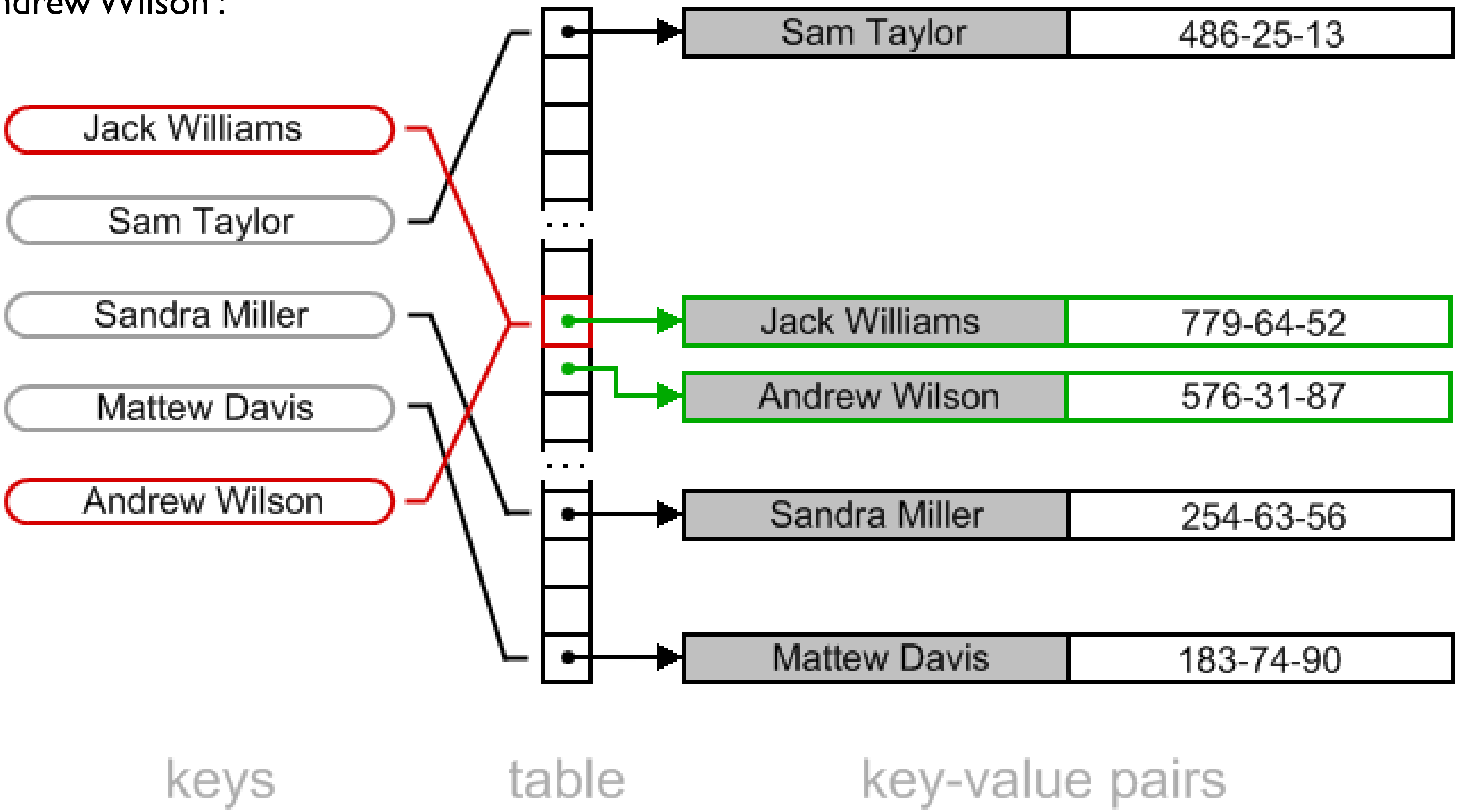
- Always check the next index
- Increments index linearly with respect to i.
- Clustering problem

hash(10) = 2
hash(5) = 5
hash(15) = 7

0	72	72	72	72
1				15
2	18	18	18	18
3	43	43	43	43
4	36	36	36	36
5		10	10	10
6	6	6	6	6
7			5	5

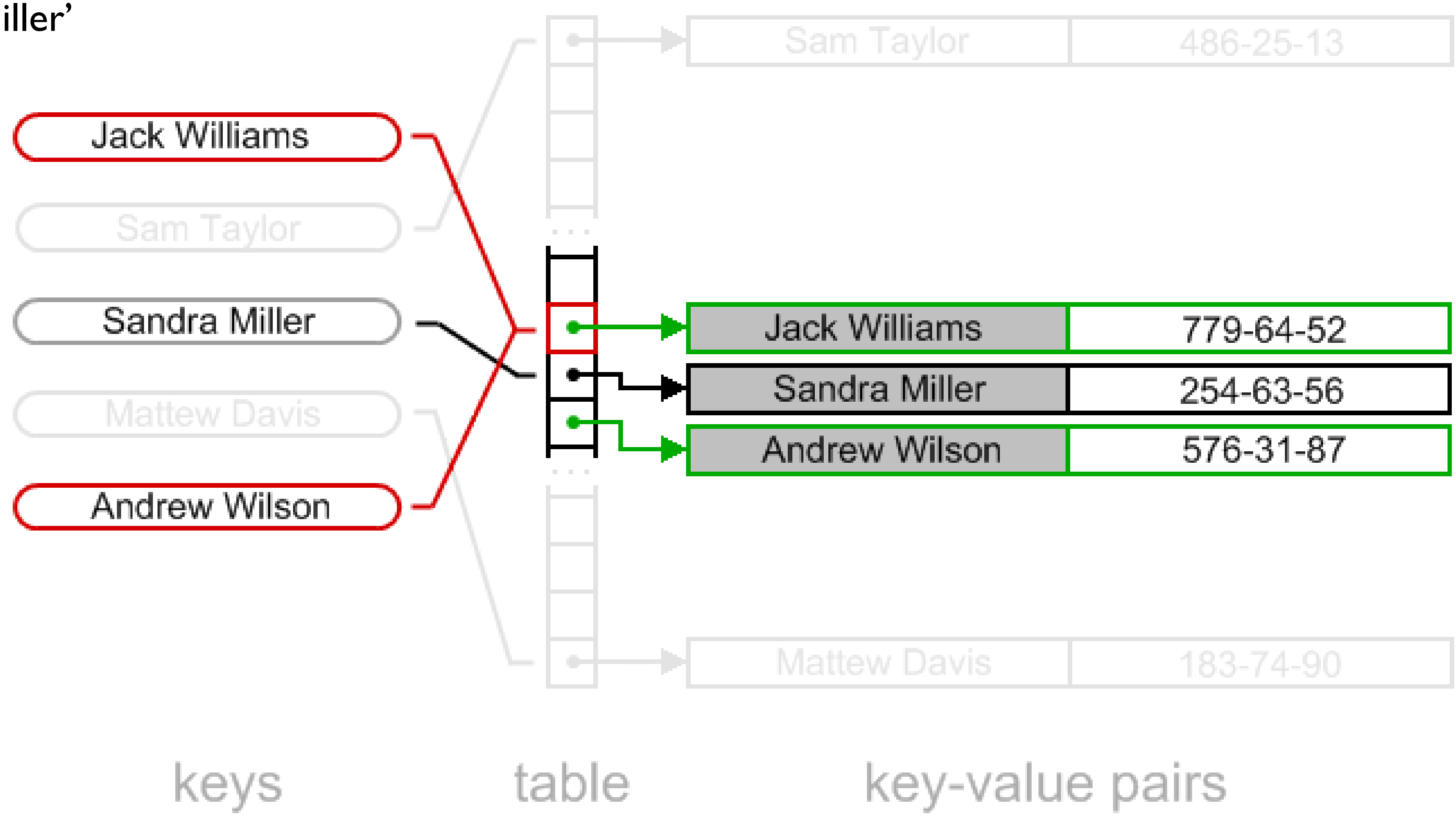
Linear Probing - insertion

Add 'Andrew Wilson':

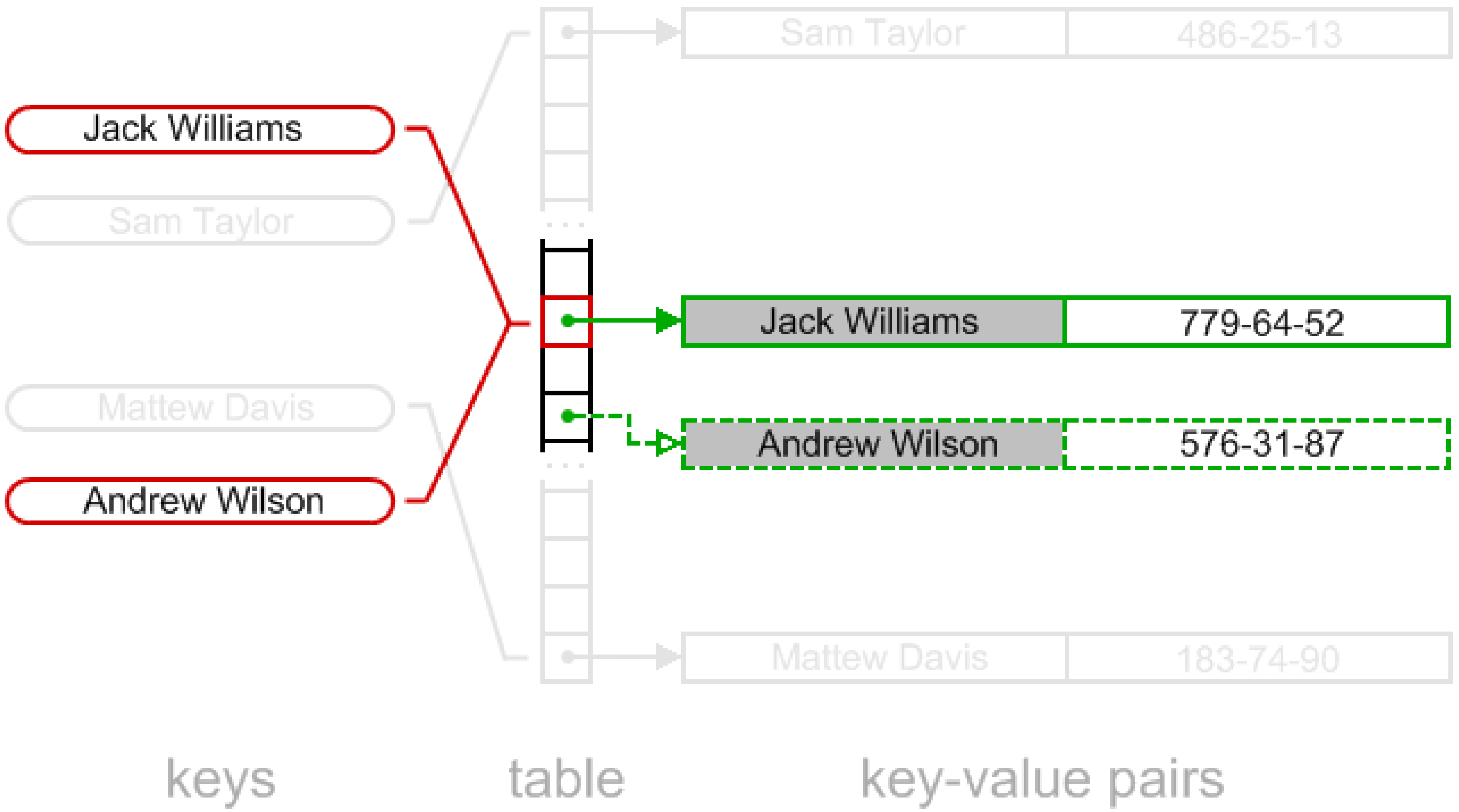


Linear search - deletion

Delete ‘Sandra Miller’



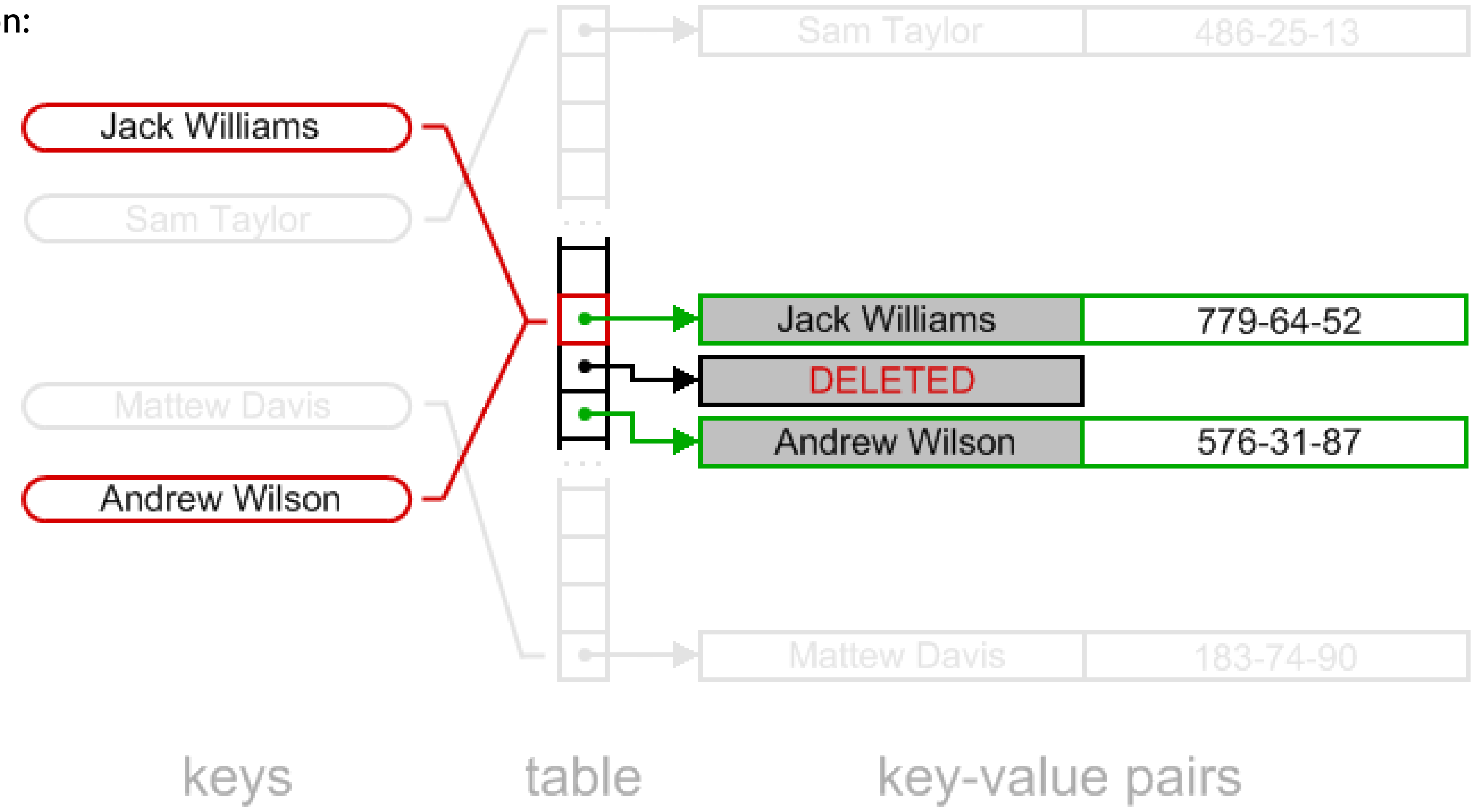
Linear search - deletion



How to find Andew Wilson?

Linear search - deletion

The solution:

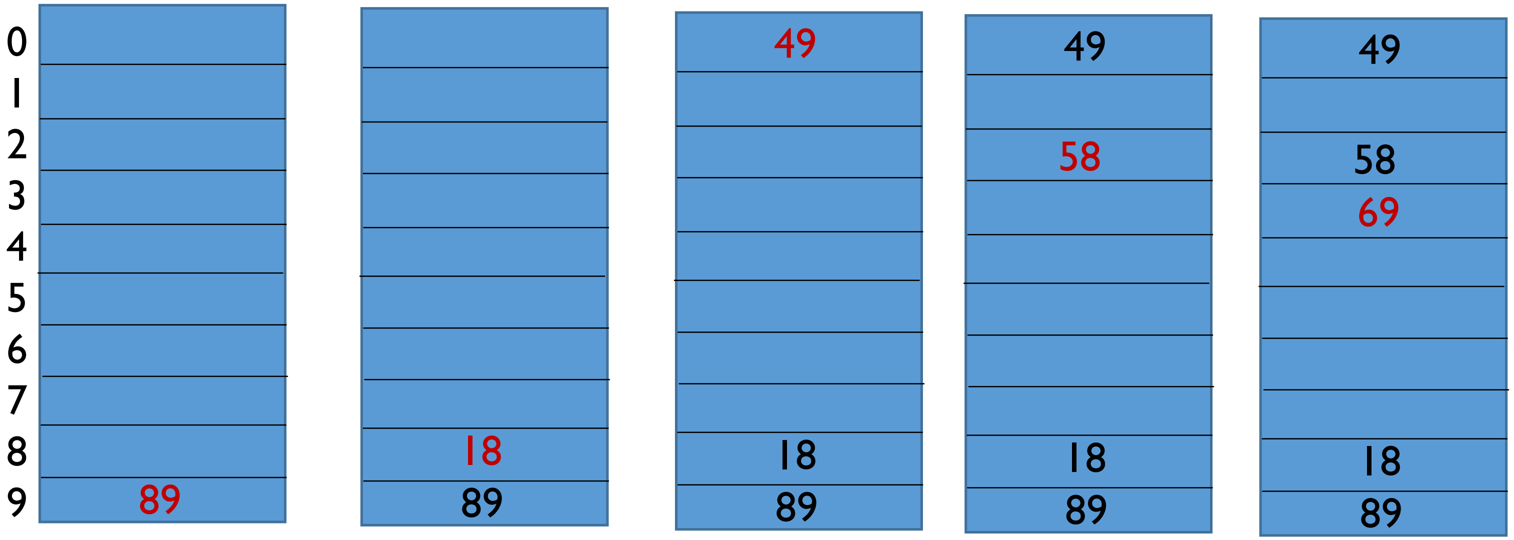


Open Addressing – Quadratic Probing

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

- Instead of moving by one, move i^2

$c_1=0$, $c_2=1$
hash(89)=9
hash(18)=8
hash(49)=9
hash(49, 1) = 0
hash(58) = 8
hash(58, 1) = 9
hash(58,2) = 2
hash(69) = 9
hash(69,1) = 0
hash(69,2) = 3



Insert Algorithm – Double Hashing

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$$

- Two hash functions
  - h_1 to find the initial position
  - h_2 to find offset from initial position
- Different from linear and quadratic, two keys mapping to same slot can now use different offsets $h_2(k)$

Example 1 - Double Hashing

- Example:
  - Table Size is 11 (0..10)
  - Hash Function:

    $h_1(x) = x \bmod 11$

    $h_2(x) = 7 - (x \bmod 7)$
  - Insert keys: 58, 14, 91
    - 58 mod 11 = 3
    - 14 mod 11 = 3 ➔ 3+7=10
    - 91 mod 11 = 3 ➔ 3+7, 3+2*7 mod 11=6

0	
1	
2	
3	58
4	
5	
6	91
7	
8	
9	
10	14

Example 2 – Double Hashing

	insert(76)	insert(93)	insert(40)	insert(47)	insert(10)	insert(55)
	76%7 = 6	93%7 = 2	40%7 = 5	47%7 = 5	10%7 = 3	55%7 = 6
				5 - (47%5) = 3		5 - (55%5) = 5
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
probes:	1	1	1	2	1	2

Dynamic Hashing Methods

- *As for any index, 2 alternatives for data entries* **k***:
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
  - Choice orthogonal to the *indexing technique*
- *Hash-based* indexes are best for *equality selections*. ***Cannot*** support range searches.

Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h(*k*) mod M** = bucket to which data entry with key *k* belongs. (M = # of buckets)