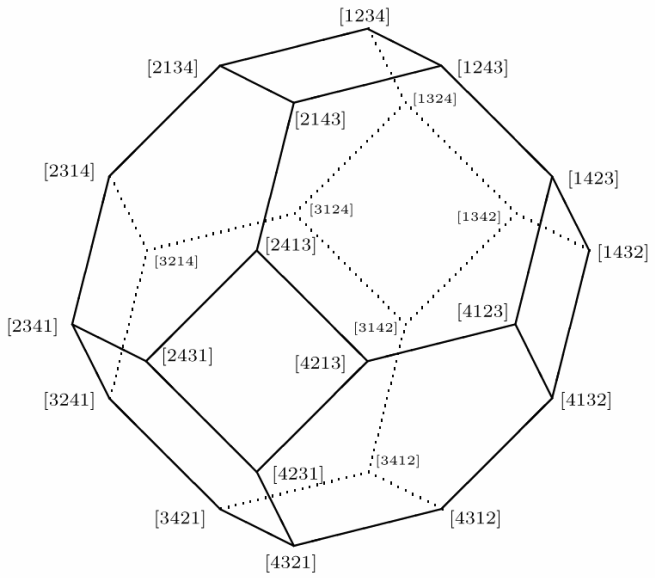
CS3334 Data Structures

Lecture 8: Hash Tables



Chee Wei Tan

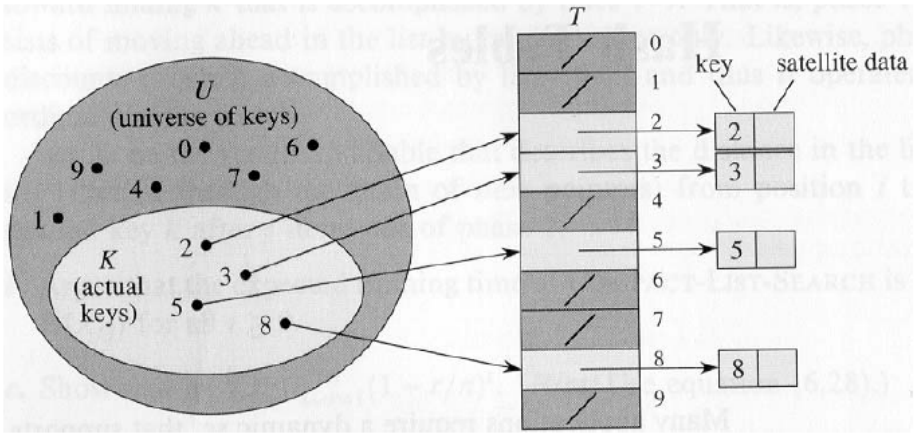
Let's Play a Game

First display of three piles

Story thus far ….
Algorithms of order statistics place special care on positions in order by leveraging array resource (e.g., bucket sorting) and array index encoding (heaps).
What if you can combine partial ideas to optimize storage and time complexity?

Introduction

- Data structures discussed so far:
  - Arrays, linked lists, stacks, and queues
  - Access of items based on “positions” (absolute or relative) in the data structure
- Hash tables
  - Data structures for a set and access items using “keys”



(insert/delete in O(1) time)

Hash Tables

- Three major operations
  - Query(x): return true if item x is in table
  - Insert(x): add item x into table
  - Delete(x): remove item x from table
- General ideas
  - Have a table T[0..M-1]
  - Store item x in T[h(x)] where h() is called a *hash* function
  - To query x, check if T[h(x)] contains x

Hash Table – Insertion

- Assume keys are non-negative integers
- Choose M=7, h(x) = x mod 7
- Initialize all cells to -1 (**for the sake of simplicity, an "empty" cell means that the cell value is "-1"**)

- x mod y = x, if x <y
- x mod y = the remainder of x/y, otherwise

Initialization:

0	
1	
2	
3	
4	
5	
6	

Insert(17):

0	
1	
2	
3	17
4	
5	
6	

h(17)=17 mod 7=3

Insert(9):

0	
1	
2	9
3	17
4	
5	
6	

h(9)=9 mod 7=2

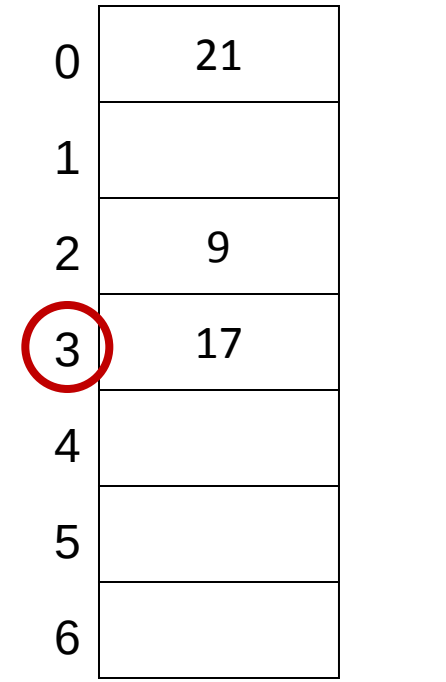
Insert(21):

0	21
1	
2	9
3	17
4	
5	
6	

h(21)=21 mod 7=0

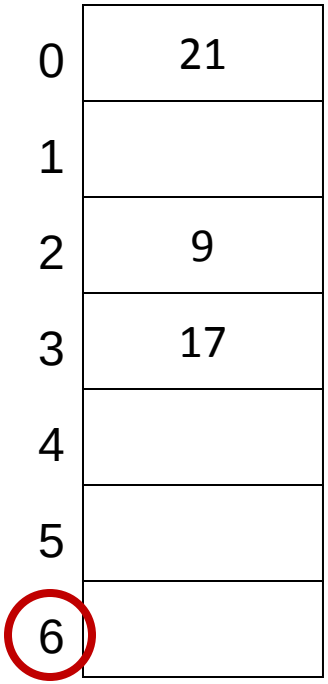
Hash Table – Query/Delete

Query(17): Yes



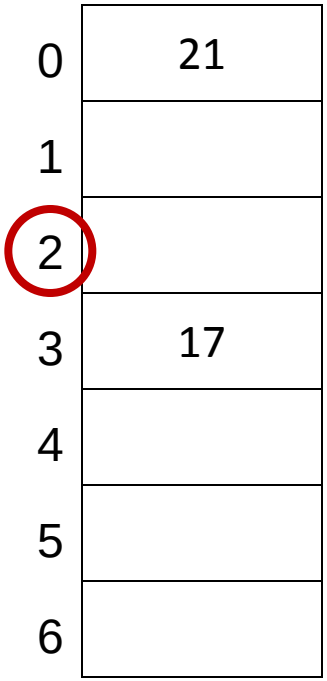
h(17)=17 mod 7=3

Query(27): No



h(27)=27 mod 7=6

Delete (9):



h(9)=9 mod 7=2

Practical Considerations

- Requirements of hash function
  - Distribute items evenly
  - Involve all bits of the items
  - Efficient to compute
- Hashing integers:
  - h(x) = x mod M (i.e., M is a prime number)
  - h(x) = ax mod M

    (no common factor between a and M)

Birthday Paradox in Hashing

- To appreciate the subtlety of hashing, first consider a puzzle: the birthday paradox.

- Suppose birth days are chance events:
  - –date of birth is purely random
  - –any day of the year just as likely as another

Birthday Paradox in Hashing

- What are the chances that in a group of 30 people, at least two have the same birthday?
- How many people will be needed to have at least 50% chance of same birthday?
- It's called a paradox because the answer appears to be counter-intuitive.
- There are 365 different birthdays, so for 50% chance, you expect at least 182 people.

Birthday Paradox in Hashing

- Suppose 2 people in the room.
- What is the prob. that they have the same birthday?
- Answer is 1/365.

  –All birthdays are equally likely, so B's birthday falls on A's birthday 1 in 365 times.

- Now suppose there are k people in the room.
- It's more convenient to calculate the prob. X that no two have the same birthday.

- Our answer will be the (1 – X)

Birthday Paradox in Hashing

- Define P_i = prob. that first i all have distinct birthdays
- For convenience, define p = 1/365
  - $P_1 = 1.$
  - $P_2 = (1 – p)$
  - $P_3 = (1 – p) * (1 – 2p)$
  - $P_k = (1 – p) * (1 – 2p) * …. * (1 – (k-1)p)$

- You can now verify that for k=23, $P_k <= 0.4999$

- That is, with just 23 people in the room, there is more than 50% chance that two have the same birthday

Birthday Paradox in Hashing

- Use $1 - x <= e^{-x}$, for all x
- Therefore, $1 - j*p <= e^{-jp}$
- Also, $e^x + e^y = e^{x+y}$

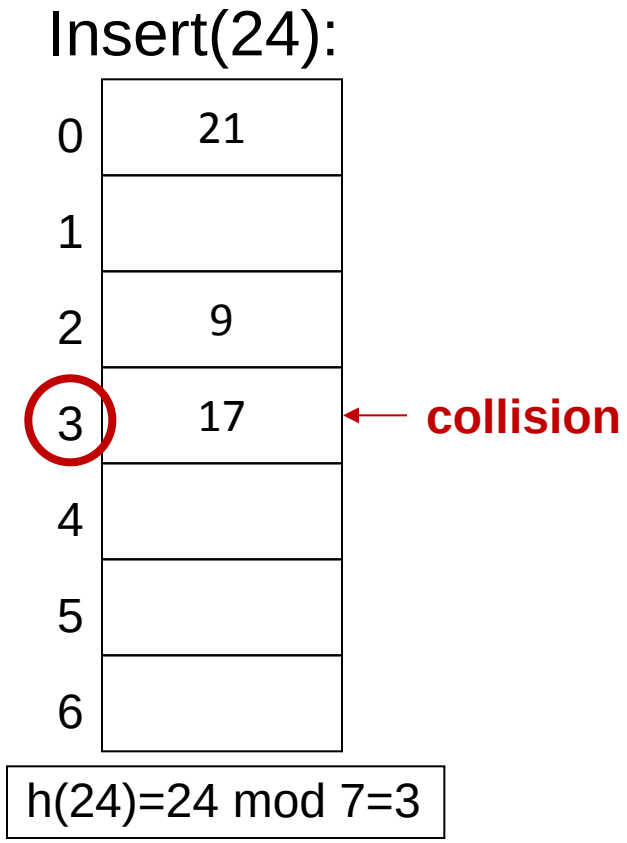
- Therefore, $P_k <= e^{(-p\ -2p\ -3p\ \ldots\ -(k-1)p)}$
- $P_k <= e^{-k(k-1)p/2}$

- For k = 23, we have k(k-1)/2*365 = 0.69
- $e^{-0.69} <= 0.4999$

- Connection to Hashing:
  - –Suppose n = 23, and hash table has size M = 365.
  - –50% chance that 2 keys will land in the same bucket
  - → Collision

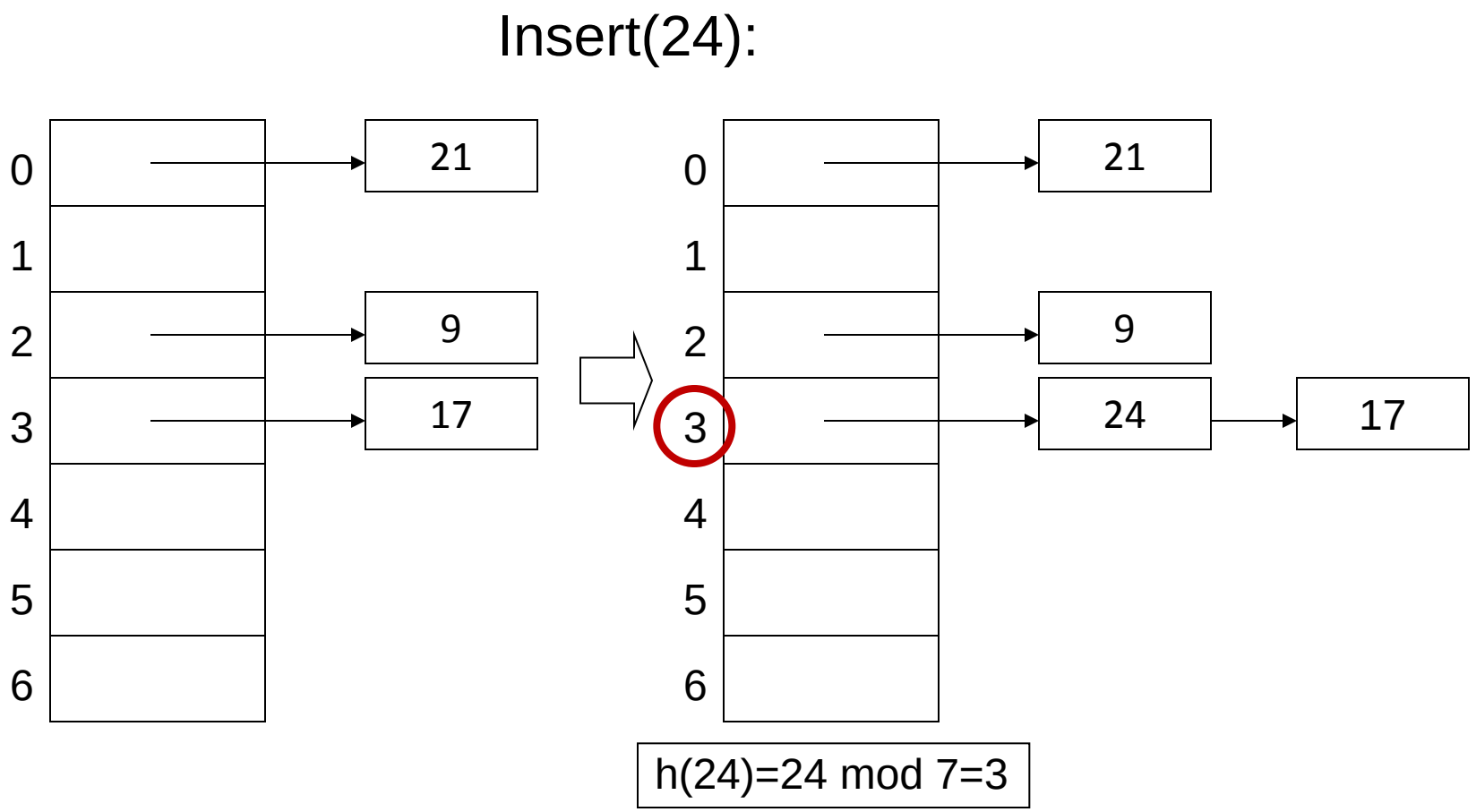
Collision

- Ideally, distinct keys hashed to distinct cells
- Collision: when two keys mapped to the same cell
- Some collision handling strategies:
  - Separate chaining
  - Linear/ quadratic probing
  - Double hashing



(1) Separate Chaining

- Each cell has a linked list



Performance

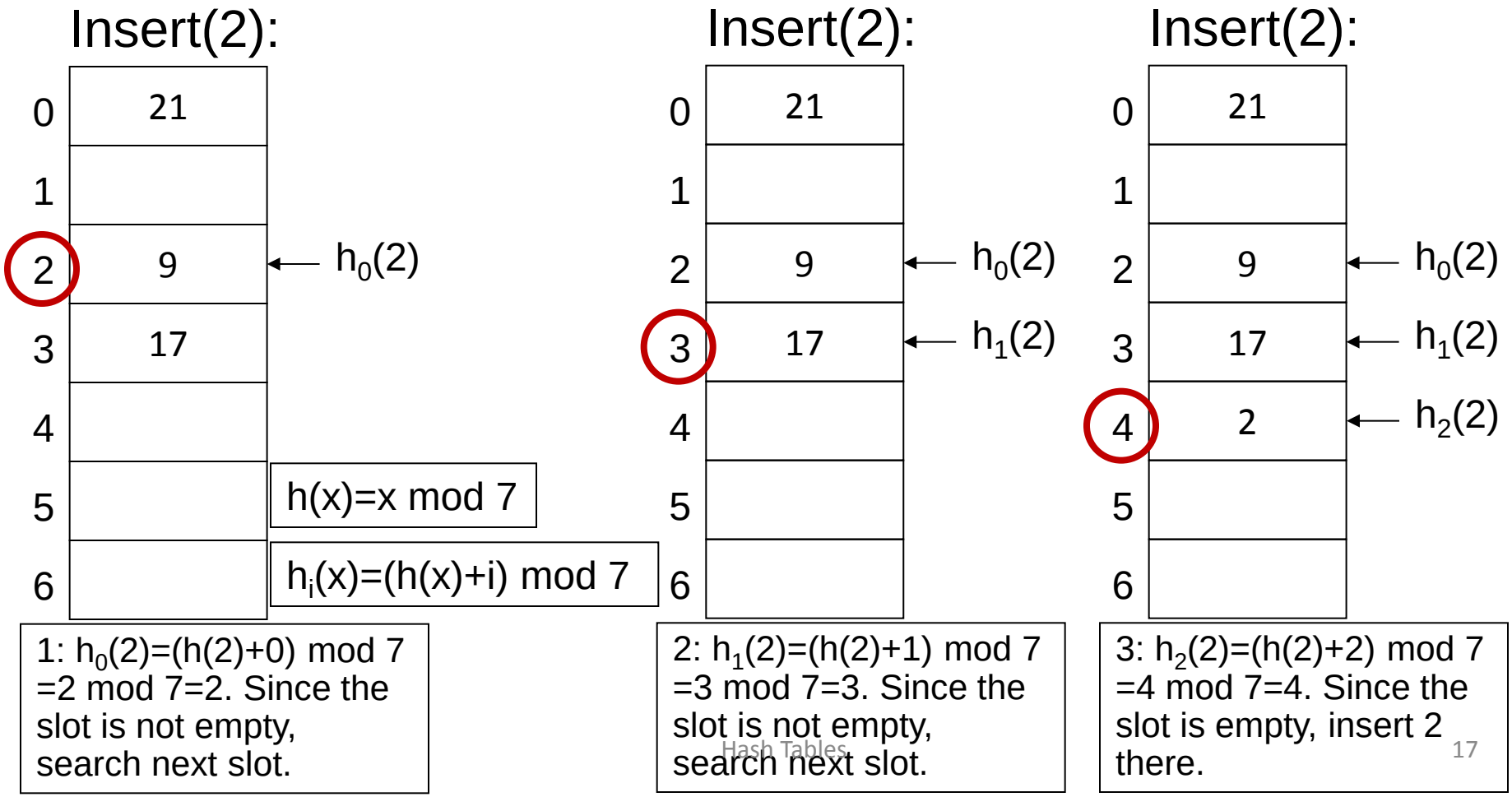
- Assume h() hashes items evenly
- Let n=number of items, and M=table size
- Define ***load factor λ = n/M***
- Average length of a linked list: n/M = λ
- Average time for query(x):
  - Constant time to compute h(x) and locate the head of linked list
  - O(λ) time to traverse
    - λ elements in the linked list if unsuccessful
    - λ/2 elements in the linked list if successful
- Time for insertion: treat as unsuccessful query => ensure no duplicate

(2) Linear/Quadratic Probing

- No linked lists. Require $\lambda < 1$.
- Try alternative cells $h_0(x)$, $h_1(x)$, … until an empty cell found
- $h_i(x) = (h(x) + f(i)) \bmod M$ where $f(0)=0$
- A check on a cell $T[h_i(x)]$ is called a "probe"
- Linear probing: f(i) linear in i
  - E.g.: $f(i) = i$
- Quadratic probing: f(i) quadratic in i
  - E.g.: $f(i) = i^2$

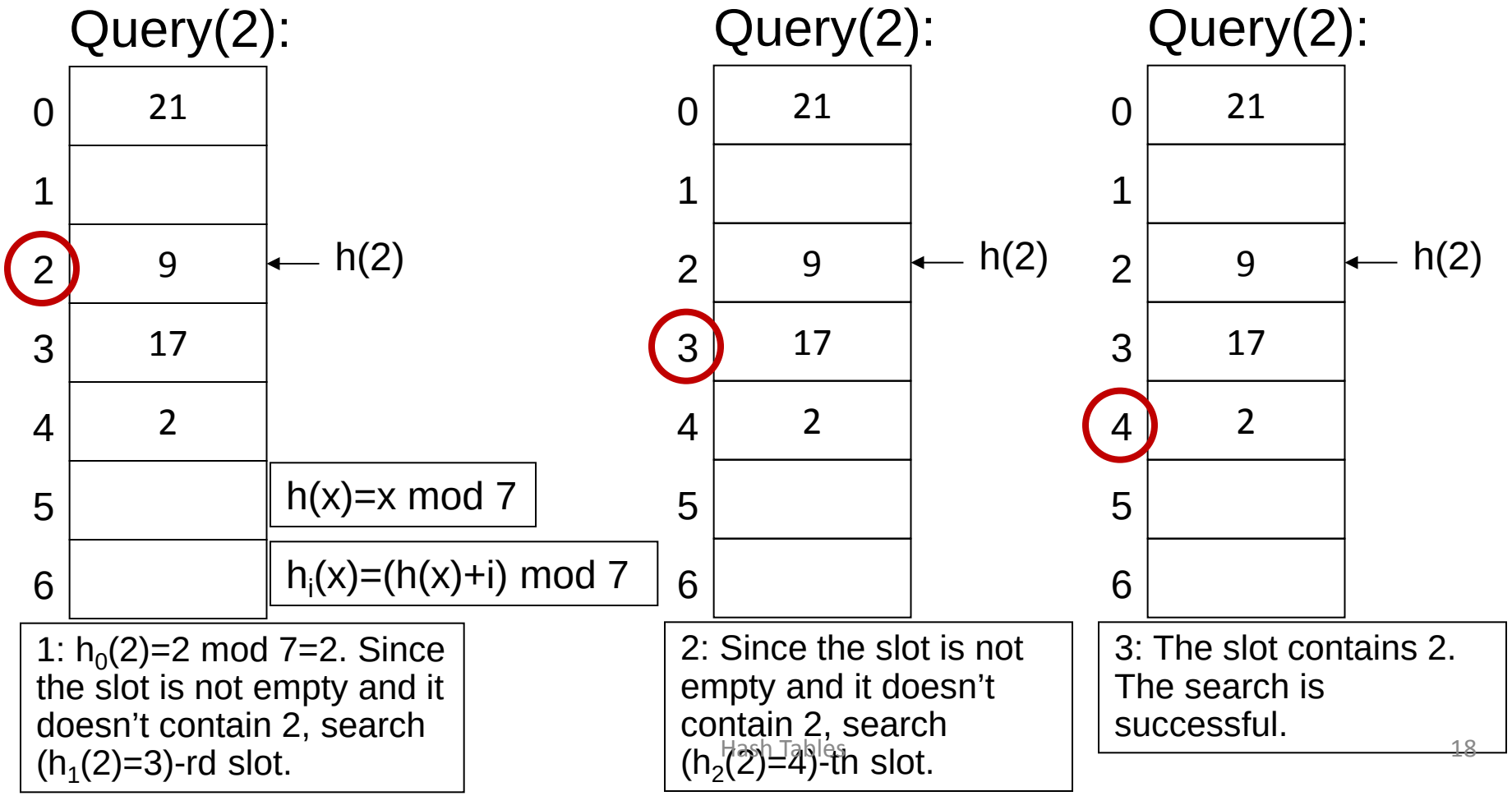
Linear Probing: Insertion

- First go to the slot which the hash function returned for an element X. Then, search sequentially until
  - An "*empty*" slot is encountered => put the element there; or
  - A slot with the same value (X) is encountered => duplicate, no insertion



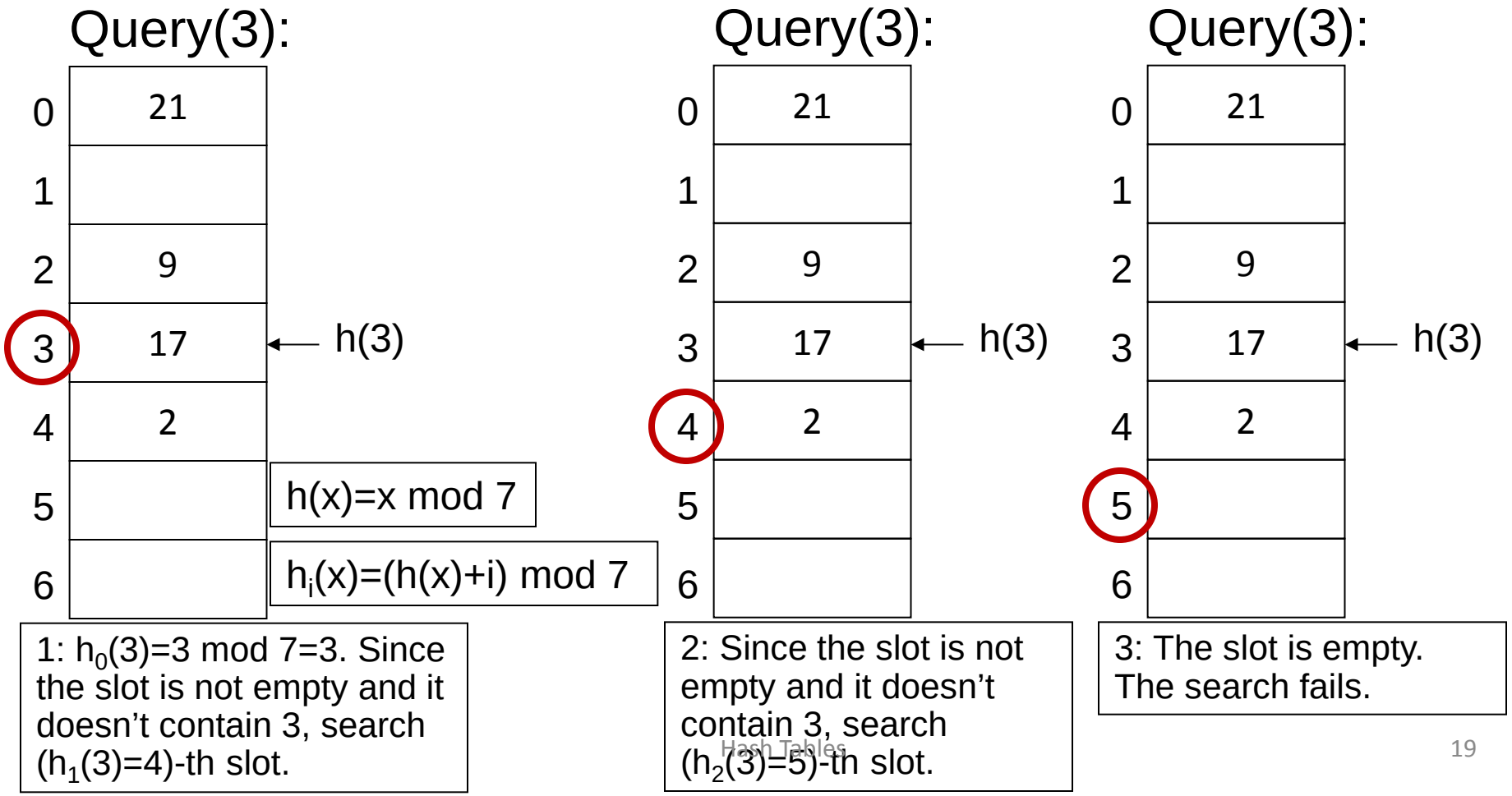
Linear Probing: Search (1/2)

- First go to the slot specified by the hash function. Then, do linear search until
  - The element to search is found (success); or
  - An "empty" slot is encountered (fail)



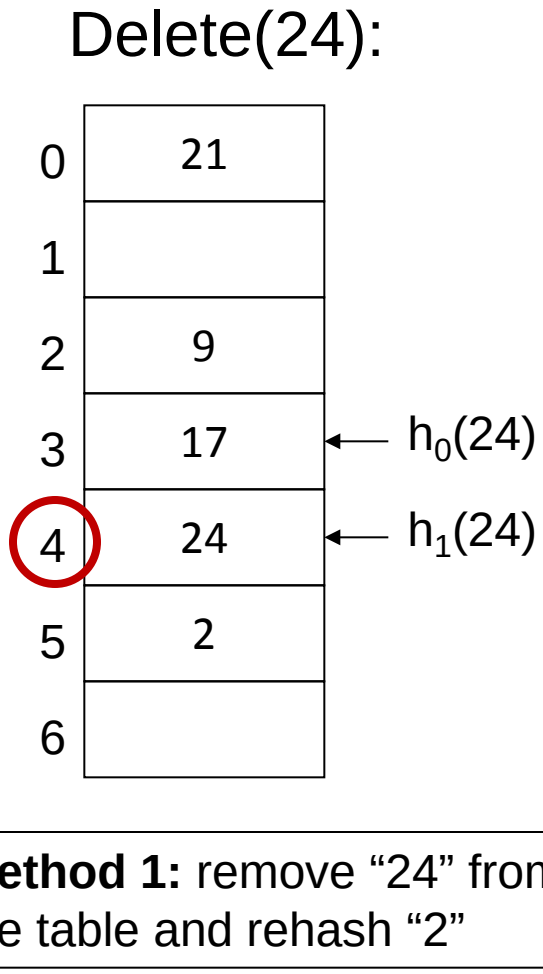
Linear Probing: Search (2/2)

- First go to the slot specified by the hash function. Then, do linear search until
  - The element to search is found (success); or
  - An "empty" slot is encountered (fail)



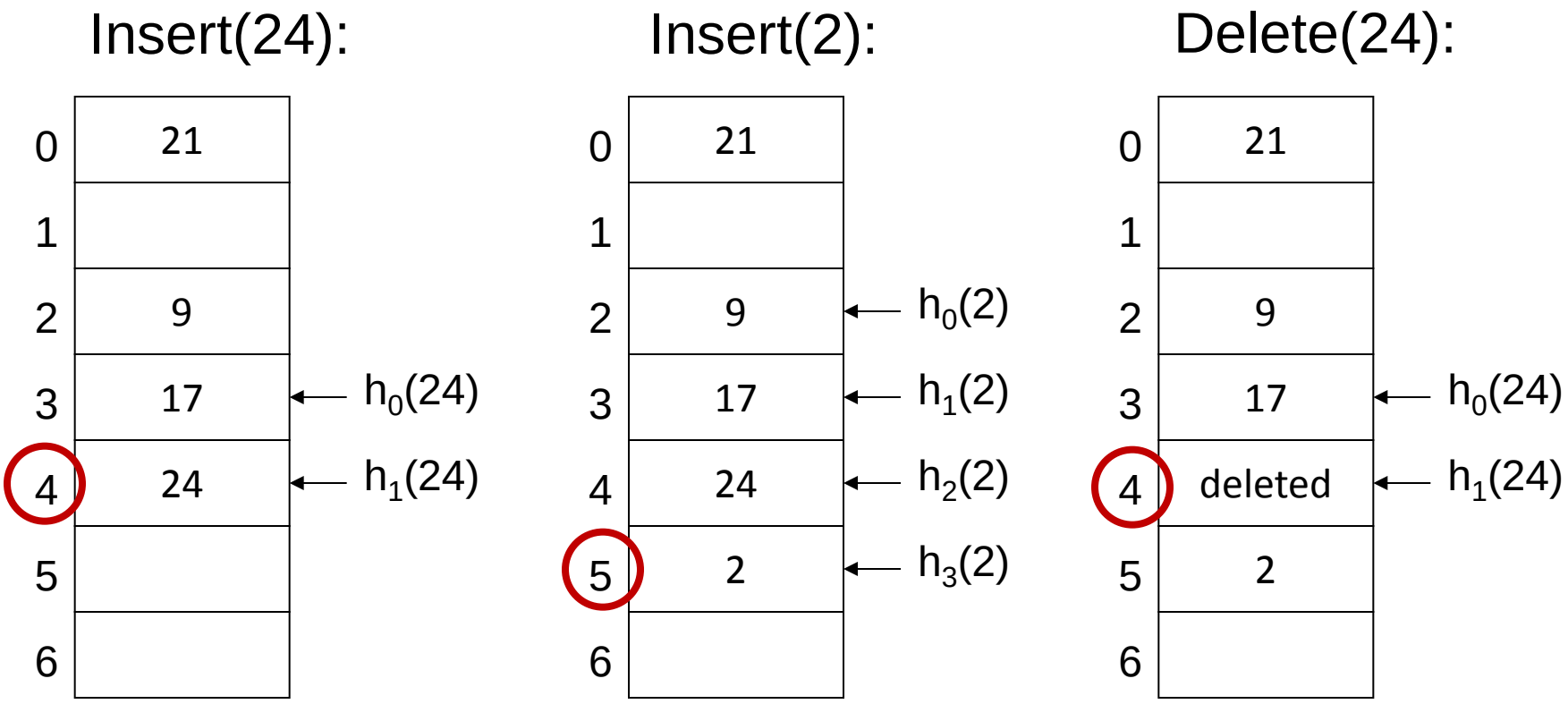
Linear Probing: Deletion

- Cannot just delete an item x
  - Items may have been inserted after insertion of x.
- Two solutions:
  - **Method 1:** Rehash all items in the chain of cells following the deleted item, i.e., *from the deleted item to next "empty" cell*, (for linear probing): good if λ is small
  - **Method 2:** Lazy deletion: replace x by a special value "deleted"



Method 1: remove "24" from the table and rehash "2"

Method 2: Lazy Deletion



- Insertion and search treat a “deleted” cell differently
  - Insertion treats it as an “empty” cell
  - Search treats it as a “non-Empty” cell

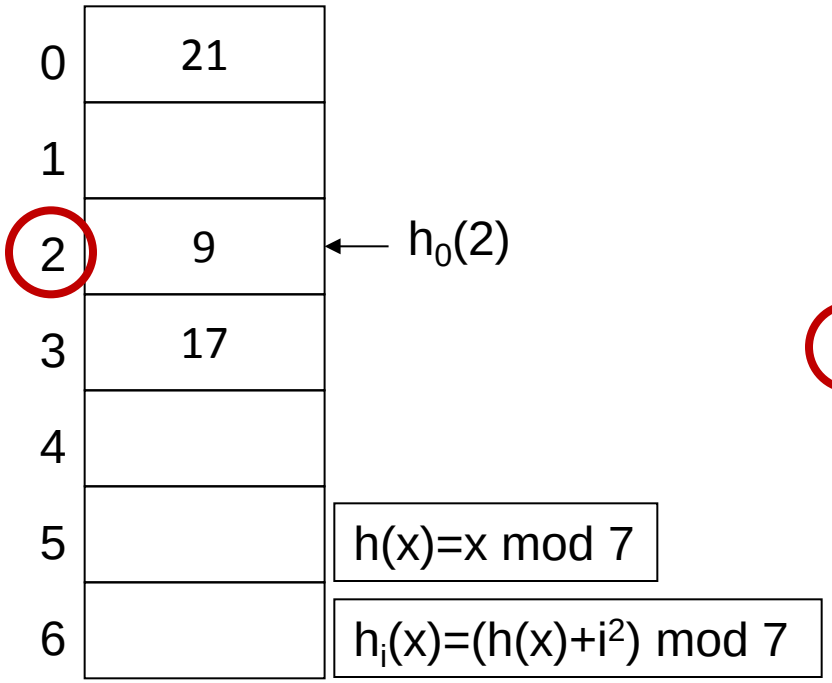
Linear Probing & Lazy Deletion

```
const int EMPTY=-1, DELETED=-2;
void insert(item x)   // assume x >=0
{
   int p = hash(x, M);  // hash() returns value in {0,…,M-1}
   while (T[p]!=EMPTY && T[p]!=x) p=(p+1)%M;
   if (T[p]==EMPTY) T[p]=x;
   else  // duplication ...
}

void delete(item x)
{
   int p = hash(x, M);
   while (T[p]!=EMPTY && T[p]!=x) p=(p+1)%M;
   if (T[p]==x) T[p]=DELETED;
   else  // not found ...
}
```

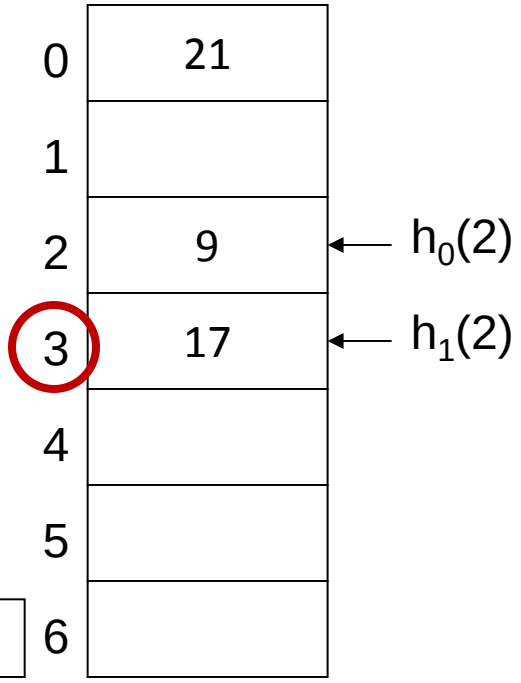
Quadratic Probing: Insertion

Insert(2):



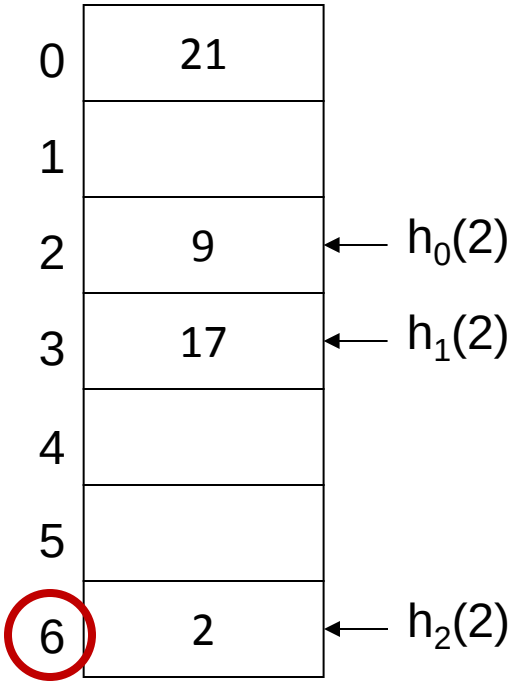
1: $h_0(2)=(h(2)+\underline{\mathbf{0^2}})$ mod 7 =2 mod 7=2. Since the slot is not empty, search next slot.

Insert(2):



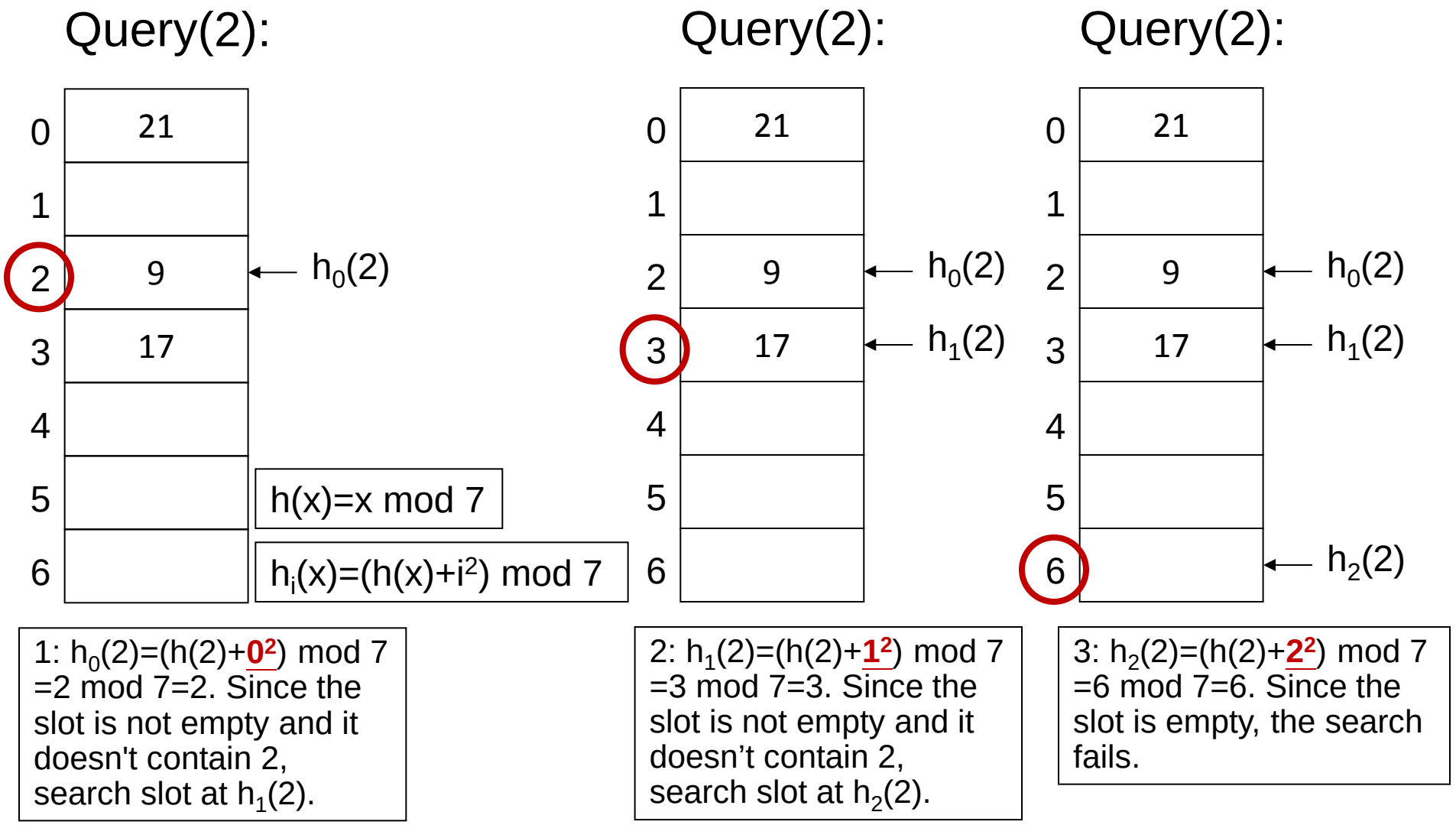
2: $h_1(2)=(h(2)+\underline{\mathbf{1^2}})$ mod 7 =3 mod 7=3. Since the slot is not empty, search next slot.

Insert(2):



3: $h_2(2)=(h(2)+\underline{\mathbf{2^2}})$ mod 7 =6 mod 7=6. Since the slot is empty, insert 2 there.

Quadratic Probing: Search



1: $h_0(2)=(h(2)+\underline{0^2})$ mod 7 =2 mod 7=2. Since the slot is not empty and it doesn't contain 2, search slot at $h_1(2)$.

2: $h_1(2)=(h(2)+\underline{1^2})$ mod 7 =3 mod 7=3. Since the slot is not empty and it doesn't contain 2, search slot at $h_2(2)$.

3: $h_2(2)=(h(2)+\underline{2^2})$ mod 7 =6 mod 7=6. Since the slot is empty, the search fails.

(3) Double hashing

- $h_i(x) = (h(x) + i * hash_2(x)) \bmod M$
- To be effective, $h(x)$ and $hash_2(x)$ have to be chosen with care, e.g.,
  - $hash_2(x) \neq 0$
  - $hash_2(x)$ should have no common factor with M; otherwise, $h_i(x)$ may not be able to find an empty cell even if there is some

Programming Game

- **Counting quadruples**
- Given four sorted arrays each of size **n** of distinct elements. Given a value **x**. The problem is to count all **quadruples**(group of four numbers) from all the four arrays whose sum is equal to **x**.
- **Note:** The quadruple has an element from each of the four arrays.

- What is the worst case time complexity of the algorithm you can come up with?

Programming Game

- Counting quadruples
- Given four sorted arrays each of size **n** of distinct elements. Given a value **x**. The problem is to count all **quadruples**(group of four numbers) from all the four arrays whose sum is equal to **x**.
- **Note:** The quadruple has an element from each of the four arrays.

```
Input : arr1 = {1, 4, 5, 6},
        arr2 = {2, 3, 7, 8},
        arr3 = {1, 4, 6, 10},
        arr4 = {2, 4, 7, 8}
```

Consider this example of four arrays. When x=30, output is 4. When x=25, output is 14.

https://www.geeksforgeeks.org/count-quadruples-four-sorted-arrays-whose-sum-equal-given-value-x

- Hashes play the role of summarizing statistics!

Let's Play a Game

- Four Fours
  - Find the ***simplest*** mathematical expression for ***every whole number*** from 0 to some maximum, using only common mathematical symbols and the digit ***four*** (no other digit is allowed).

Knowledge: An Illustrated Magazine of Science (1881)

This problem and its generalizations may be solved by a simple algorithm. The basic ingredients are hash tables that map rationals to strings. In these tables, the keys are the numbers being represented by some admissible combination of operators and the chosen digit *d*, e.g. four, and the values are strings that contain the actual formula.

https://en.wikipedia.org/wiki/Four_fours