Lecture 11
Hashing

For Efficient Look-up Tables

Lecture Outline

- What is hashing?
- How to hash?
- What is collision?
- How to resolve collision?
  - Separate chaining
  - Linear probing
  - Quadratic probing
  - Double hashing
- Load factor
- Primary clustering and secondary clustering

What is Hashing?

- **Hashing** is an algorithm (via a **hash function**) that maps **large** data sets of **variable** length, called **keys**, to **smaller** data sets of a **fixed** length

- A **hash table** (or **hash map**) is a data structure that uses a hash function to efficiently map keys to values, for efficient search and retrieval

- Widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets

Table ADT

CS2010 stuff

Operations	Sorted Array	Balanced BST	Hashing
Insertion	O(*n*)	O(log *n*)	O(1) avg
Deletion	O(*n*)	O(log *n*)	O(1) avg
Retrieval	O(log *n*)	O(log *n*)	O(1) avg

- Hence, **hash table** supports the Table ADT in **constant time on average** for the above operations (terms and conditions apply…)

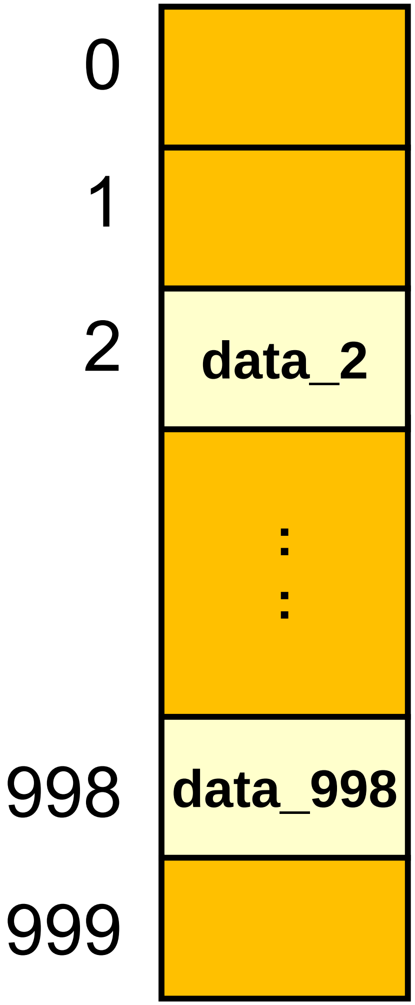
Direct Addressing Table

The easiest form of hashing

Example: SBS Bus Services

Now there are more bus operators in SG

- Operations
  - **Retrieval**: find(*num*)
    - Find the bus route of bus service number *num*
  - **Insertion**: insert(*num*)
    - Introduce a new bus service number *num*
  - **Deletion**: delete(*num*)
    - Remove bus service number *num*



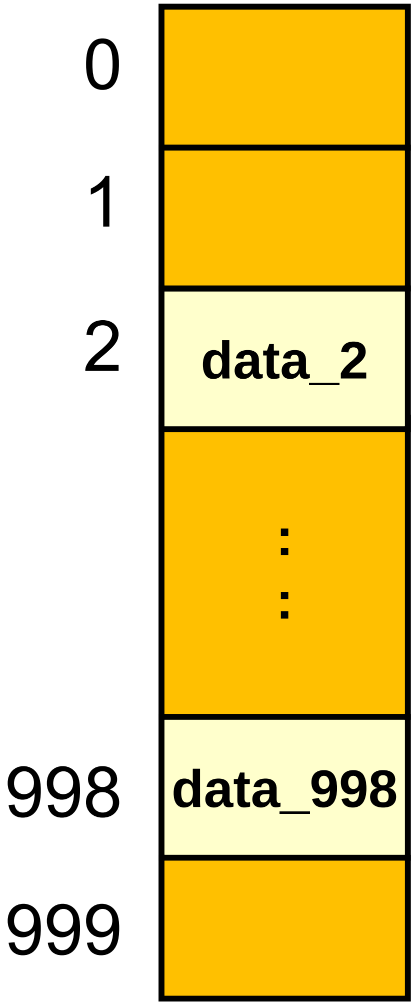
0
1
2 data_2
:
:
998 data_998
999

- If bus numbers are integers 0 – 999, we can use an array with 1000 entries

Of course for now we assume that bus numbers don’t have variants, like 96A, 96B…, etc

Example: SBS Bus Services

```
// a[] is an array (the table)

insert(key, data)
   a[key] = data

delete(key)
   a[key] = NULL

find(key)
   return a[key]
```



Direct Addressing Table: Limitations

- Keys must be non-negative integer values
  - What happen for key values 151A and NR10?
- Range of keys must be small
- Keys must be dense
  - i.e. not many gaps in the key values
- How to overcome these restrictions?

Hash Table

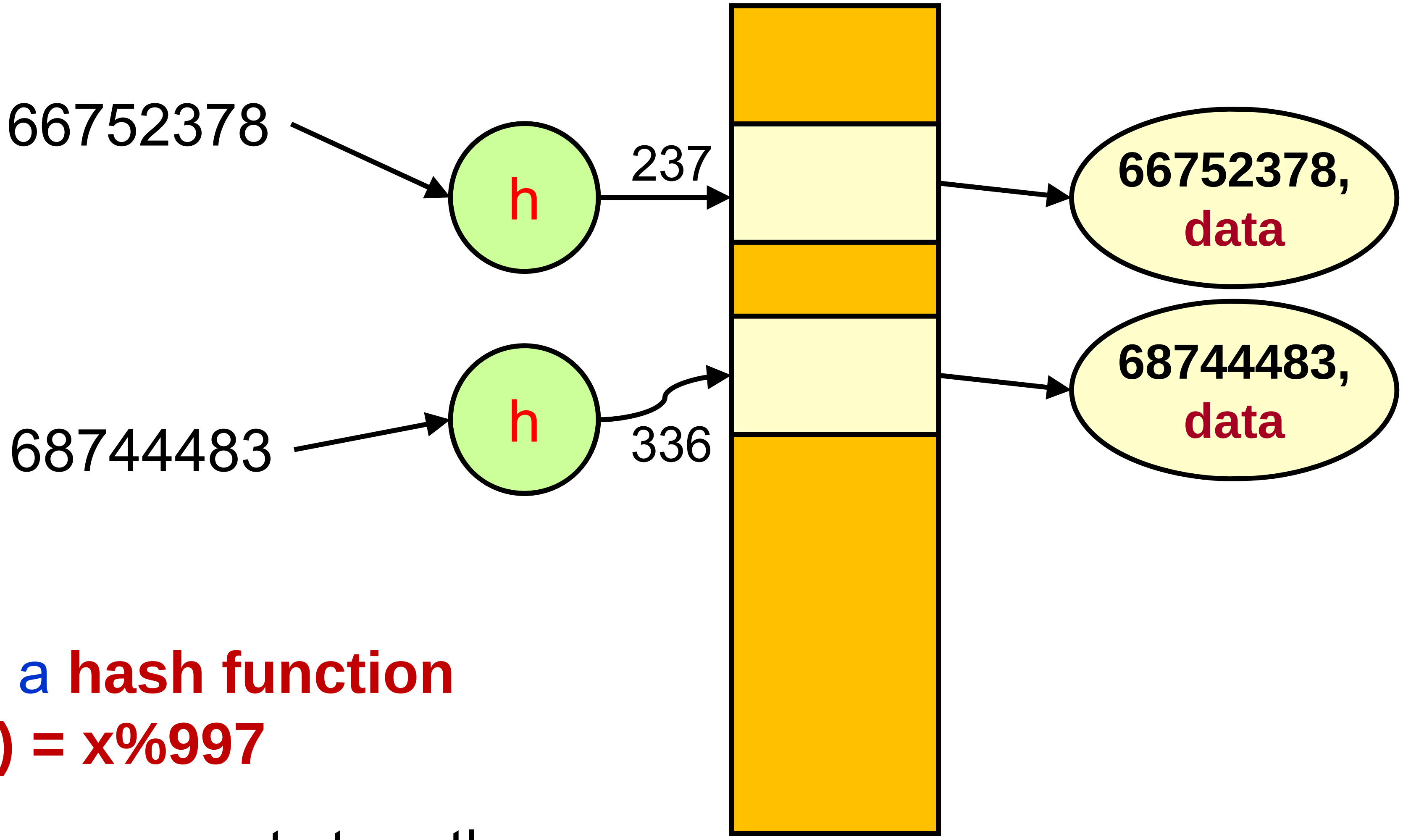
The true form of hashing…

Hashing: Ideas

- Map large integers to smaller integers

- Map non-integer keys to integers

Hash Table: Phone Numbers Example



h is a **hash function**

h(x) = x%997

Note: we must store the key values. Why?

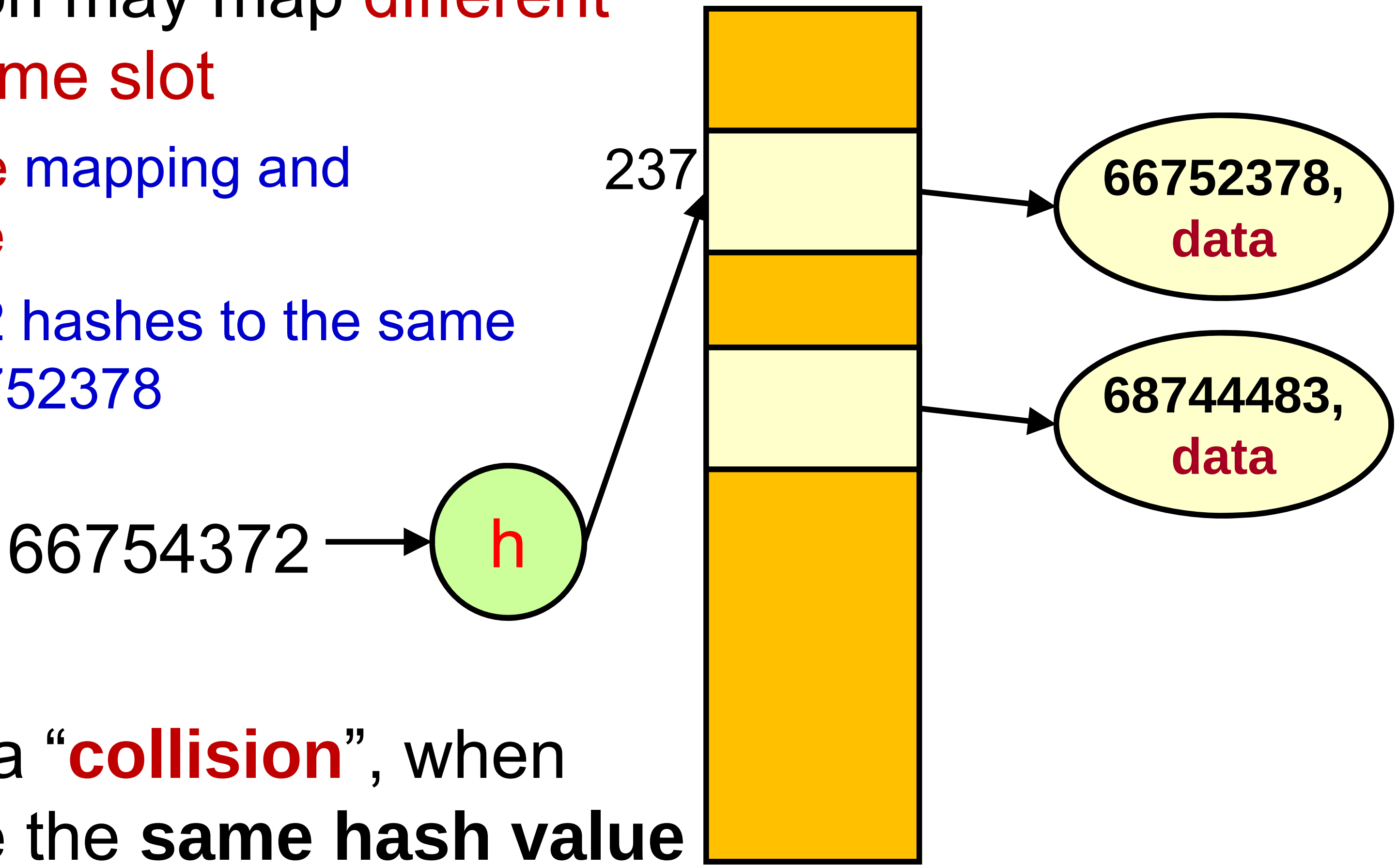
Hash Table: Operations

```
// a[] is an array (the table)
// h is a hash function

insert(key, data)
  a[h(key)] = data

delete(key)
  a[h(key)] = NULL

find(key)
  return a[h(key)]
```

However, this does **not** work for **all** cases! Why?

Hash Table: Collision

- A hash function may map different keys to the same slot
  - A many-to-one mapping and not one-to-one
  - E.g. 66754372 hashes to the same location of 66752378



- This is called a "**collision**", when two keys have the **same hash value**

Two Important Issues

- How to hash?
- How to resolve collisions?

Hash Functions

How to create a good one?

Hash Functions and Hash Values

- Suppose we have a **hash table** of size *N*
- **Keys** are used to identify the data
- A **hash function** is used to compute a **hash value**
- A hash value (hash code) is
  - Computed from the key with the use of a hash function to get a number in the range 0 to *N*–1
  - Used as the index (address) of the table entry for the data
  - Regarded as the "home address" of a key
- **Desire**: The addresses are different and spread evenly over the range
- When two keys have same hash value — **collision**

Good Hash Functions

- Fast to compute, O(**1**)
- Scatter keys evenly throughout the hash table
- Less collisions
- Need less slots (space)

Bad Hash Functions: Example

- Select Digits
  - e.g. choose the 4th and 8th digits of a phone number
  - hash(677**5**437**8**) = 58
  - hash(634**9**782**0**) = 90

- What happen when you hash Singapore’s house phone numbers by selecting the first three digits?

Perfect Hash Functions

- Perfect hash function is a **one-to-one** mapping between keys and hash values. So no collision occurs
- Possible if all keys are known
- Applications: compiler and interpreter search for reserved words; shell interpreter searches for built-in commands
- GNU gperf is a freely available perfect hash function generator written in C++ that automatically constructs perfect functions (a C++ program) from a user supplied list of keywords
- Minimal perfect hash function: The table size is the same as the number of keywords supplied

How to Define a Hash Function?

- Uniform hash function
- Division method
- Multiplication method
- Hashing of strings

Uniform Hash Functions

- Distributes keys uniformly in the hash table
- If keys are uniformly distributed in [0, *X*), we map them to a hash table of size *m* (*m* < *X*) using the hash function below

$$k \in [0, X)$$

$$hash(k) = \left\lfloor \frac{km}{X} \right\rfloor$$

k is the key value

[ ]: close interval

( ): open interval

Hence, $0 \le k < X$

⌊ ⌋ is the *floor* function

Division Method (mod operator)

- Map into a hash table of *m* slots
- Use the modulo operator (%) to map an integer to a value between 0 and *m*–1
  - *n* mod *m* = remainder of *n* divided by *m*, where *n* and *m* are positive integers

$$hash(k) = k \% m$$

- The most popular method

How to Pick ***m*** (table size)?

- If *m* is power of two, say 2^n, then (*key* mod *m*) is the same as extracting the last *n* bits of the key
- If *m* is 10^n, then the hash value is the last *n* digit of the key
- Both are not good, why?
- **Rule of thumb:** Pick a **prime number**, *close to a power of two*, to be *m*

Multiplication Method

1) Multiply key by a fraction *A* (between 0 and 1)
2) Extract the fractional part
3) Multiply by *m*, the hash table size

$$hash(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

- The reciprocal of the golden ratio = (sqrt(5) – 1) / 2 = 0.618033 seems to be a good choice for *A*

Hashing of Strings: Example

```
hash1(s) {  // s is a string
  sum = 0
  for each character c in s {
    sum += c
    // sum up the ASCII values of all characters
  }
  return sum % m  // m is the hash table size
}
```

Hashing of Strings: Example

```
hash1("Tan Ah Teck")
= ('T' + 'a' + 'n' + ' ' +
   'A' + 'h' + ' ' +
   'T' + 'e' + 'c' + 'k') % 11
   // hash table size is 11
= (84 + 97 + 110 + 32 +
   65 + 104 + 32 +
   84 + 101 + 99 + 107) % 11
= 825 % 11
= 0
```

Hashing of Strings: Example

- All 3 strings below have the same hash value. Why?
  - `"Lee Chin Tan"`
  - `"Chen Le Tian"`
  - `"Chan Tin Lee"`

- Problem: The hash value is independent of the positions of the characters

Improved Hashing of Strings

- Better to “shift” the sum before adding the next character, so that its position affects the hash code
  - Polynomial hash code

```
hash2(s) {
   sum = 0
   for each character c in s {
      sum = sum * 37 + c
   }
   return sum % m
}
```

Collision Resolution

Handling the inevitables…

Probability of Collision

- **von Mises Paradox** (**The Birthday Paradox**):
  "How many people must be in a room before the probability that some share a birthday, ignoring the year and leap days, becomes at least 50 percent?"

Q(*n*) = Probability of unique birthday for *n* people

$$= \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \cdots \frac{365-n+1}{365}$$

P(*n*) = Probability of collisions (same birthday) for *n* people

= 1 – Q(*n*)

P(**23**) = 0.507

Hence, you need only **23 people** in the room!

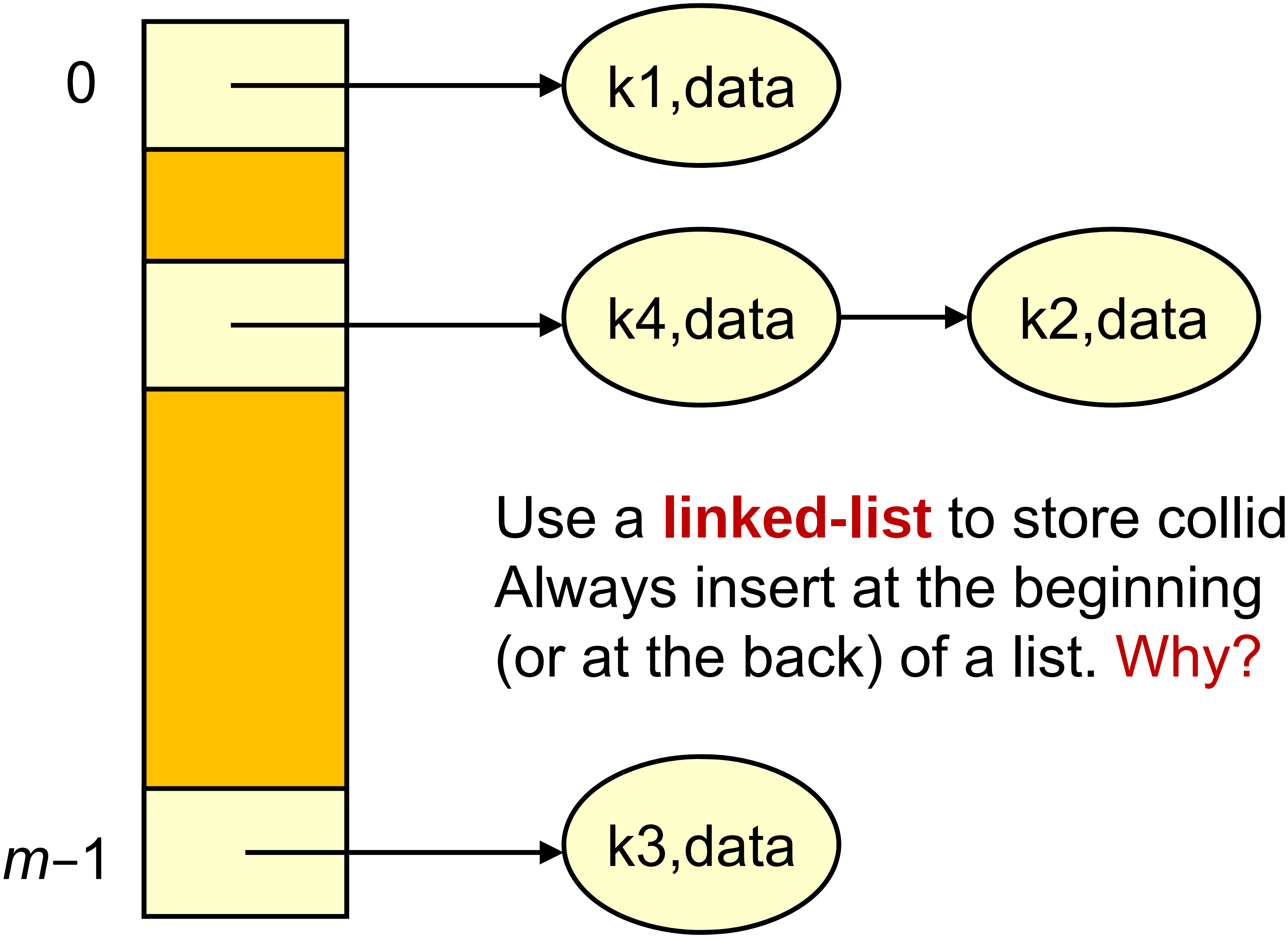
Probability of Collision

- This means that if there are 23 people in a room, the probability that some people share a birthday is 50.7%!
- In the hashing context, if we insert 23 keys into a table with 365 slots, more than half of the time we will get collisions! Such a result is counter-intuitive to many
- So, collision is very likely!

Collision Resolution Techniques

- Separate Chaining
- Linear Probing
- Quadratic Probing
- Double Hashing

Separate Chaining



Use a **linked-list** to store collided keys.
Always insert at the beginning
(or at the back) of a list. Why?

Load Factor

- *n*: number of keys in the hash table
- *m*: size of the hash tables — number of slots

- α: **load factor**
  - $\alpha = n / m$
  - Measures how full the hash table is.
  - In separate chaining, table size equals to the number of linked lists, so α is the average length of the linked lists

Separate Chaining: Performance

- Hash table operations
  - **insert** (key, data)
    - Insert data into the list a[h(key)]
    - Takes O(1) time
  - **find** (key)
    - Find key from the list a[h(key)]
    - Takes O(1+α) time on average
  - **delete** (key)
    - Delete data from the list a[h(key)]
    - Takes O(1+α) time on average

If α is bounded by some constant, then all three operations are O(**1**)

Open Addressing

- Separate chaining is a **close addressing** system as the address given to a key is fixed
- When the hash address given to a key is open (not fixed), the hashing is an **open addressing** system
- **Open addressing**
  - Hashed items are in a single array
  - Hash code gives the home address
  - Collision is resolved by checking multiple positions
  - Each check is called a **probe** into the table

Linear Probing

hash(*k*) = *k* mod 7

Here the table size *m* = 7

Note: 7 is a prime number.

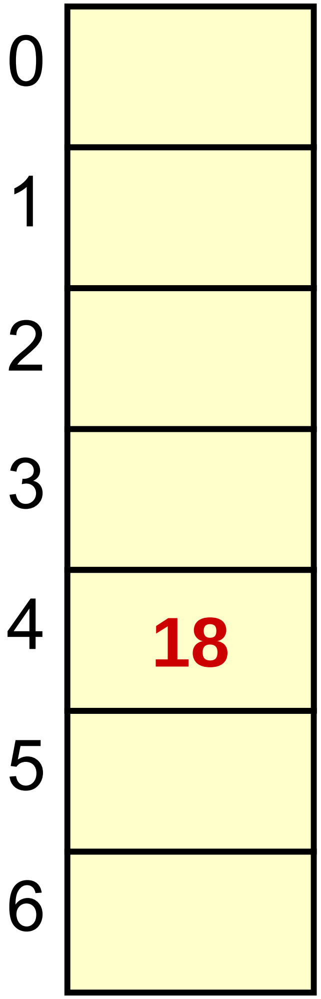
0	
1	
2	
3	
4	
5	
6	

In linear probing, when there is a collision, we scan forwards for the the next empty slot (wrapping around when we reach the last slot).

Linear Probing: Insert 18

hash(*k*) = *k* mod 7

hash(18)
= 18 mod 7
= 4



0	
1	
2	
3	
4	**18**
5	
6	

Linear Probing: Insert 14

hash(*k*) = *k* mod 7

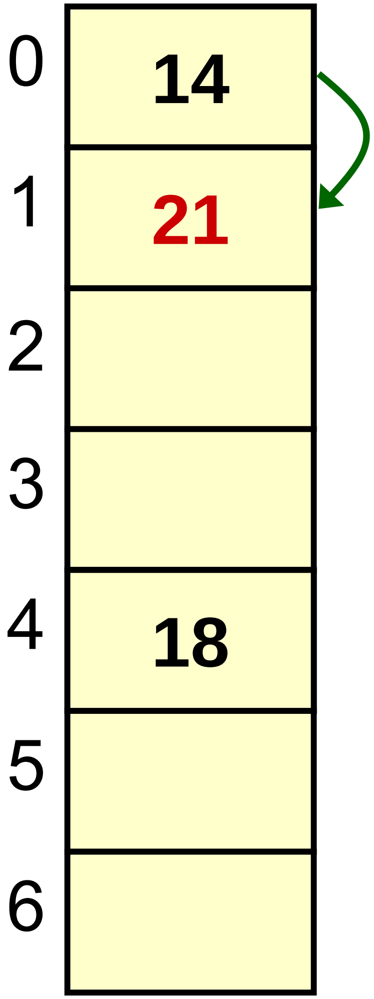
hash(14)
= 14 mod 7
= 0

0	**14**
1	
2	
3	
4	**18**
5	
6	

Linear Probing: Insert 21

hash(*k*) = *k* mod 7

hash(21)
= 21 mod 7
= 0



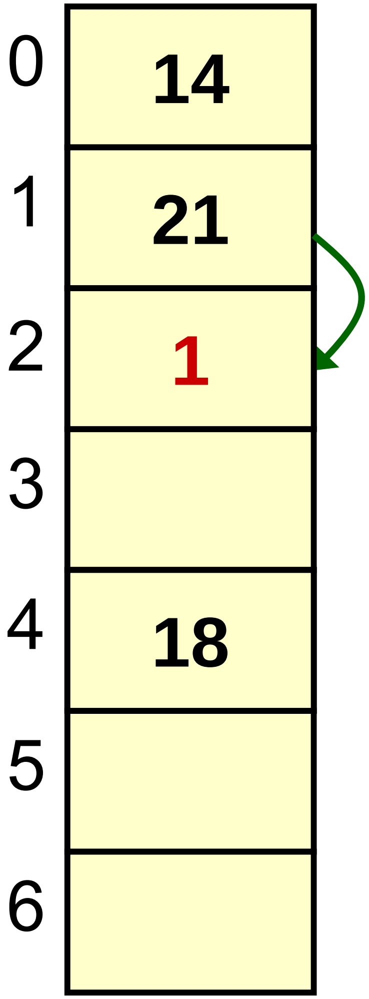
0	**14**
1	**21**
2	
3	
4	**18**
5	
6	

Collision occurs!
Look for next empty slot.

Linear Probing: Insert 1

hash(*k*) = *k* mod 7

hash(1)
= 1 mod 7
= 1



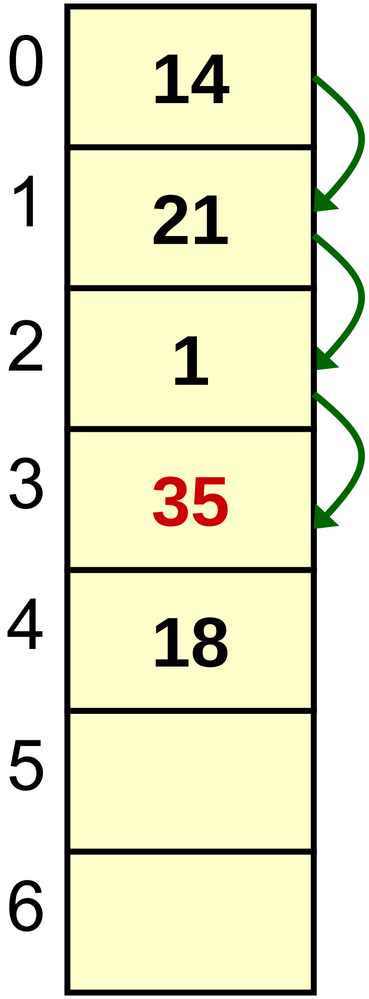
0	14
1	21
2	1
3	
4	18
5	
6	

Collides with 21 (hash value 0). Look for next empty slot.

Linear Probing: Insert 35

hash(*k*) = *k* mod 7

hash(35)
= 35 mod 7
= 0



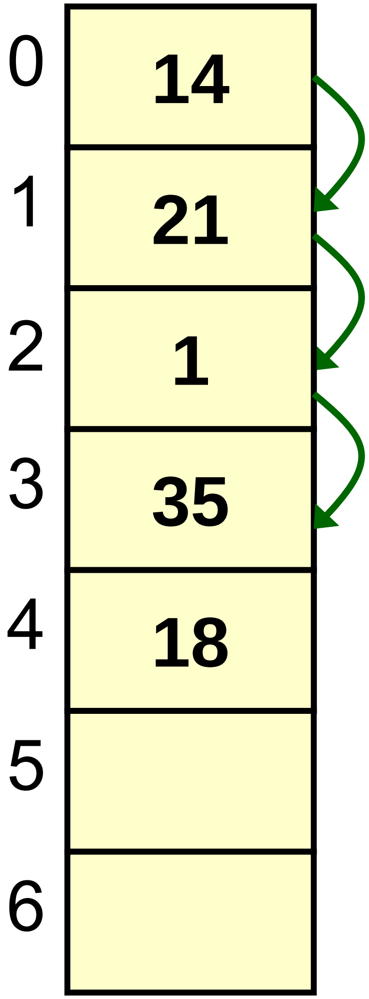
0	14
1	21
2	1
3	35
4	18
5	
6	

Collision, need to check next 3 slots.

Linear Probing: Find 35

hash(*k*) = *k* mod 7

hash(35)
= 35 mod 7
= 0



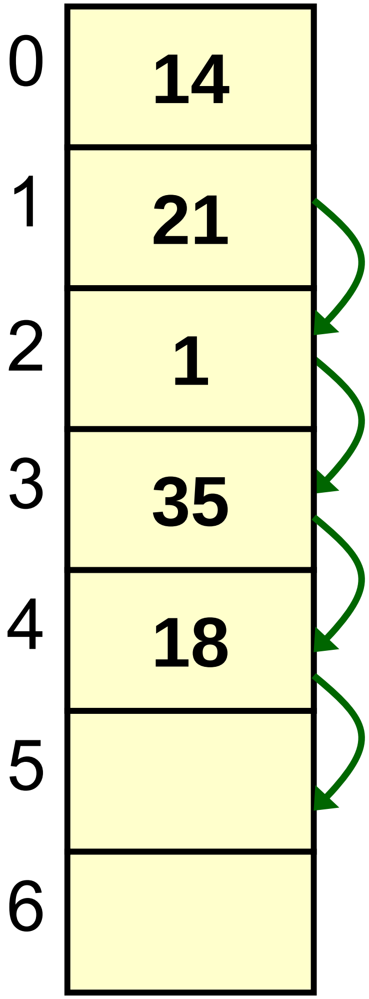
0	**14**
1	**21**
2	**1**
3	**35**
4	**18**
5	
6	

Found 35, after 4 probes.

Linear Probing: Find 8

hash(*k*) = *k* mod 7

hash(8)
= 8 mod 7
= 1



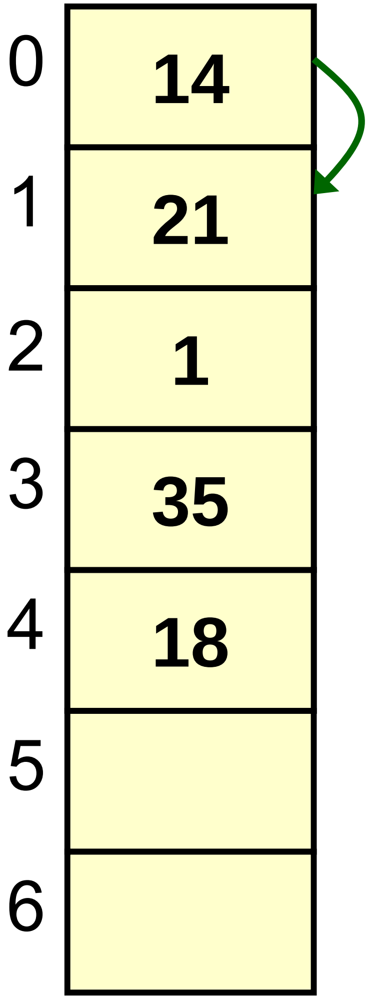
0	**14**
1	**21**
2	**1**
3	**35**
4	**18**
5	
6	

8 NOT found.
Need 5 probes!

Linear Probing: Delete 21

hash(*k*) = *k* mod 7

hash(21)
= 21 mod 7
= 0

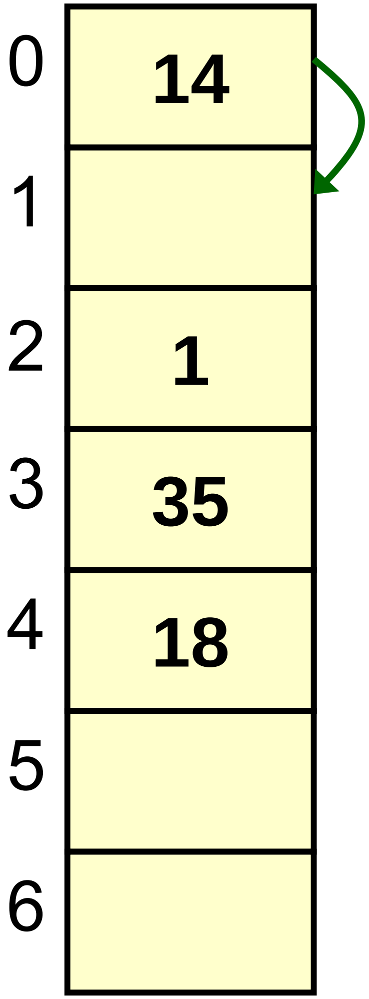


0	**14**
1	**21**
2	**1**
3	**35**
4	**18**
5	
6	

Linear Probing: Find 35

hash(*k*) = *k* mod 7

hash(35)
= 35 mod 7
= 0



0	**14**
1	
2	**1**
3	**35**
4	**18**
5	
6	

35 NOT found!
Incorrect!

We cannot simply remove a value, because it can affect find()!

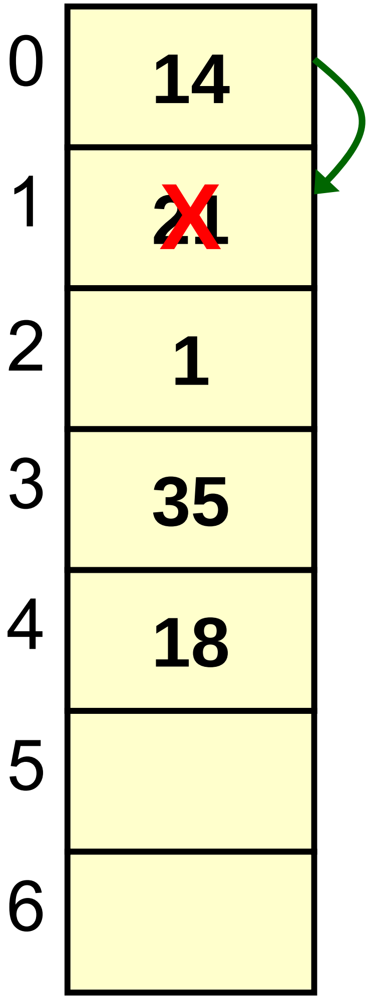
How to Delete?

- **Lazy Deletion**
- Use three different **states** at each slot
  - Occupied
  - Deleted
  - Empty
- When a value is removed from linear probed hash table, we just mark the status of the slot as "deleted", instead of emptying the slot
- Need to use a state array the same size as the hash table

Linear Probing: Delete 21

hash(*k*) = *k* mod 7

hash(21)
= 21 mod 7
= 0



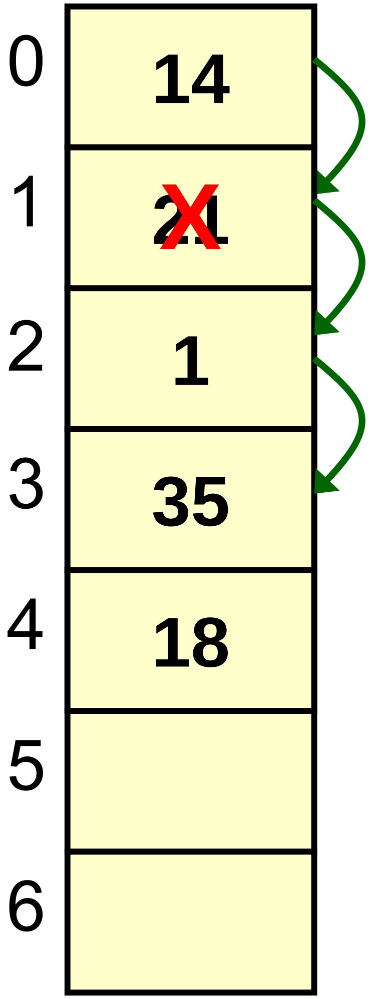
0	**14**
1	**~~21~~**
2	**1**
3	**35**
4	**18**
5	
6	

Slot 1 is occupied but now marked as deleted.

Linear Probing: Find 35

hash(*k*) = *k* mod 7

hash(35)
= 35 mod 7
= 0



0	**14**
1	**21** X
2	**1**
3	**35**
4	**18**
5	
6	

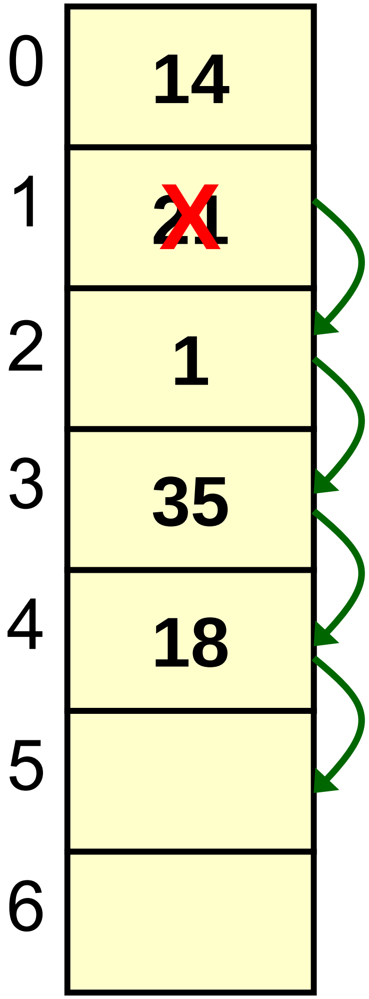
Found 35.

Now we can find 35.

Linear Probing: Insert 15 (1/2)

hash(*k*) = *k* mod 7

hash(15)
= 15 mod 7
= 1



Slot 1 is marked as deleted.

We continue to search for 15, and found that 15 is not in the hash table (total 5 probes).

So, we insert this new value 15 into the slot that has been marked as deleted (i.e. slot 1).

Linear Probing: Insert 15 (2/2)

hash(*k*) = *k* mod 7

hash(15)
= 15 mod 7
= 1

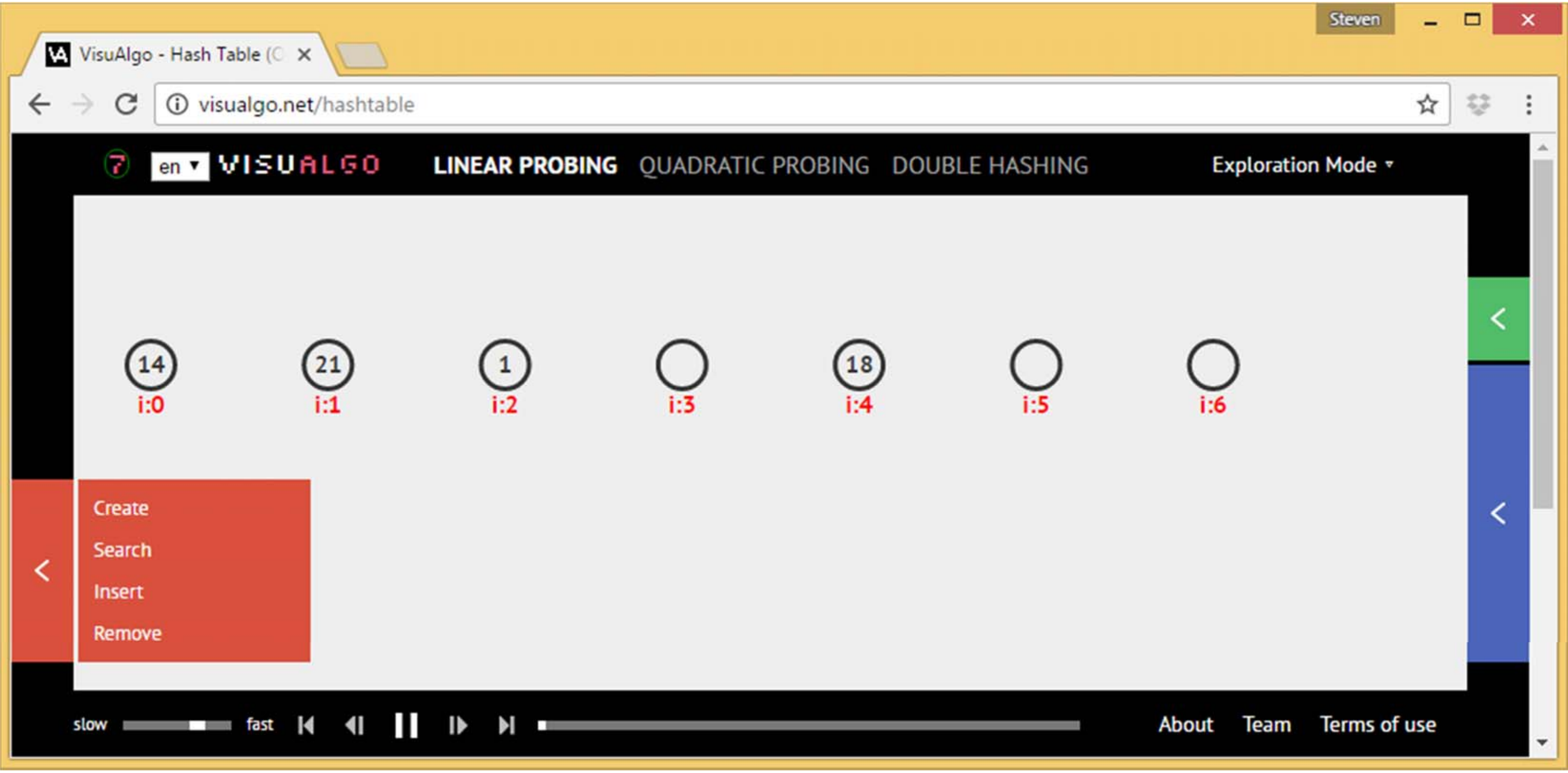
0	**14**
1	**15**
2	**1**
3	**35**
4	**18**
5	
6	

So, 15 is inserted into slot 1, which was marked as deleted.

Note: We should insert a new value in first available slot so that the find operation for this value will be the fastest.

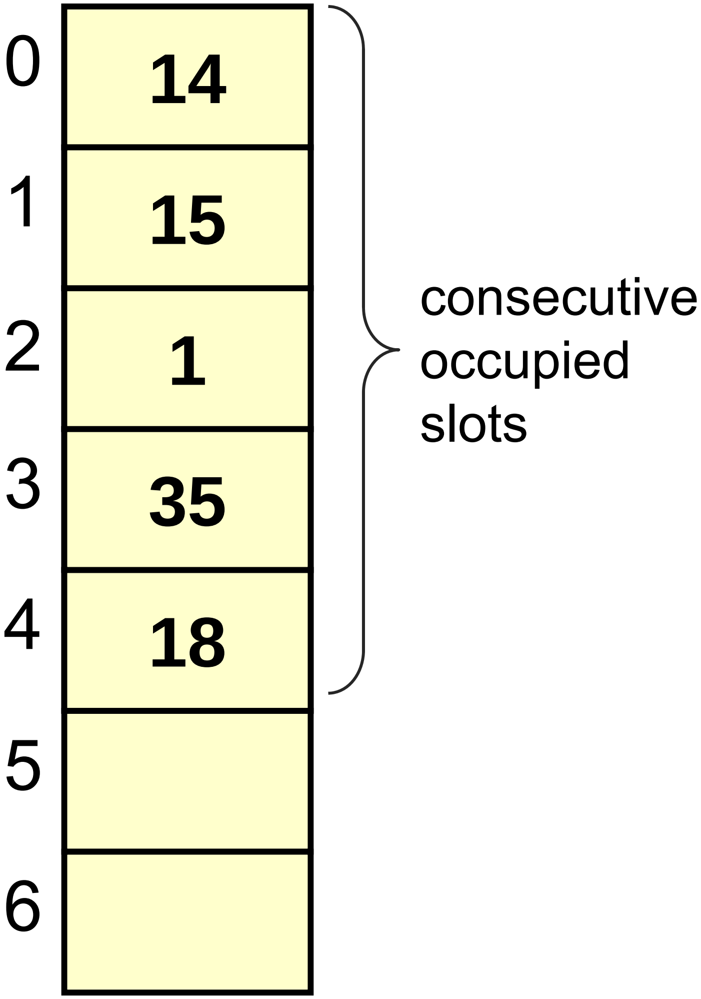
VisuAlgo (Part 1)

- Hash Table with linear probing collision resolution has been integrated in VisuAlgo (http://visualgo.net/hashtable)



Problem 1: Primary Clustering

- A **cluster** is a collection of consecutive occupied slots
- A cluster that covers the home address of a key is called the **primary cluster** of the key
- Linear probing can create large primary clusters that will increase the running time of find/insert/delete operations



0	**14**
1	**15**
2	**1**
3	**35**
4	**18**
5	
6	

Linear Probing: Probe Sequence

- The **probe sequence** of this linear probing is

  hash(key)

  (hash(key) + **1**) % *m*

  (hash(key) + **2**) % *m*

  (hash(key) + **3**) % *m*

  ⋮

- If there is an empty slot, we are sure to find it
- When an empty slot is found, conflict resolved, but the primary cluster of the key is expanded as a result
- The size of the resulting primary cluster may be very big due to the annexation of the neighboring cluster

Modified Linear Probing

- To reduce primary clustering, we can modify the probe sequence to

  hash(key)

  (hash(key) + **1** * ***d***) % *m*

  (hash(key) + **2** * ***d***) % *m*

  (hash(key) + **3** * ***d***) % *m*

  ⋮

  where ***d*** is some constant integer >1 and is co-prime to *m*

  - Since ***d*** and *m* are co-primes, the probe sequence covers all the slots in the hash table

Quadratic Probing

- The probe sequence of **quadratic probing** is

$$\text{hash(key)}$$

$$(\ \text{hash(key)} + \mathbf{1}\ )\ \%\ m$$

$$(\ \text{hash(key)} + \mathbf{4}\ )\ \%\ m$$

$$(\ \text{hash(key)} + \mathbf{9}\ )\ \%\ m$$

$$\vdots$$

$$(\ \text{hash(key)} + \boldsymbol{k}^{\mathbf{2}}\ )\ \%\ m$$

Quadratic Probing: Insert 18, 3

hash(*k*) = *k* mod 7

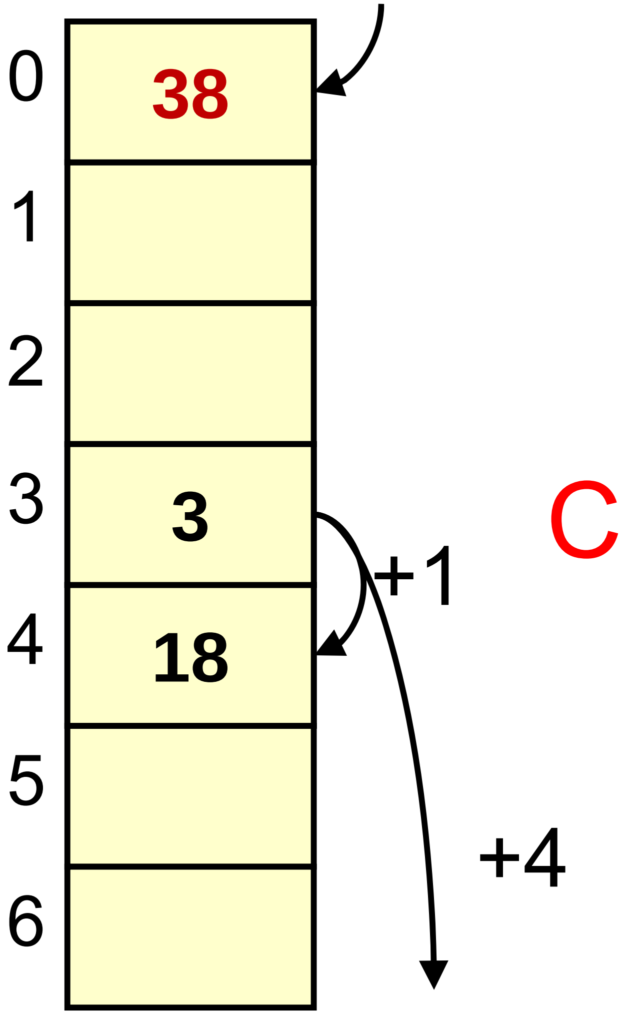
hash(18) = 4
hash(3) = 3

0	
1	
2	
3	**3**
4	**18**
5	
6	

Quadratic Probing: Insert 38

hash(*k*) = *k* mod 7

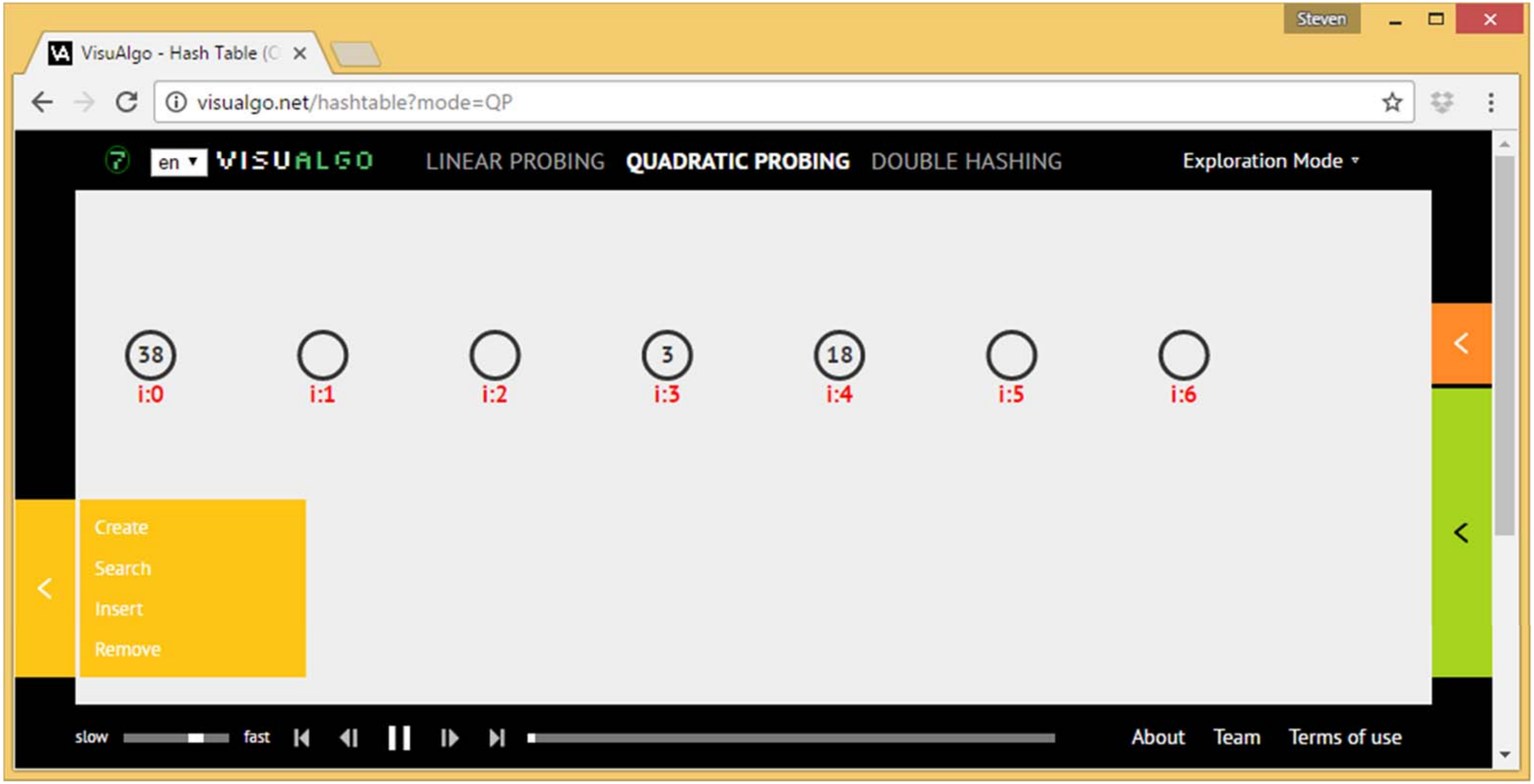
hash(38) = 3



Collision!

VisuAlgo (Part 2)

- Hash Table with quadratic probing collision resolution is also in VisuAlgo (http://visualgo.net/hashtable?mode=QP)



Theorem of Quadratic Probing

- How can we be sure that quadratic probing always terminates?
  - Insert 12 into the previous example, followed by 10. See what happen?
  - Try it on VisuAlgo directly

- **Theorem**: If $\alpha < 0.5$, and m is prime, then we can always find an empty slot
  - m is the table size and α is the load factor

Problem 2: Secondary Clustering

- In quadratic probing, clusters are formed along the path of probing, instead of around the home location
- These clusters are called **secondary clusters**
- Secondary clusters are formed as a result of using the same pattern in probing by all keys
  - If two keys have the same home location, their probe sequences are going to be the same
- But it is not as bad as primary clustering in linear probing

Double Hashing

- To reduce secondary clustering, we can use a second hash function to generate different probe sequences for different keys

  hash(key)

  (hash(key) + **1** * $\textbf{hash}_2$(key)) % *m*

  (hash(key) + **2** * $\textbf{hash}_2$(key)) % *m*

  (hash(key) + **3** * $\textbf{hash}_2$(key)) % *m*

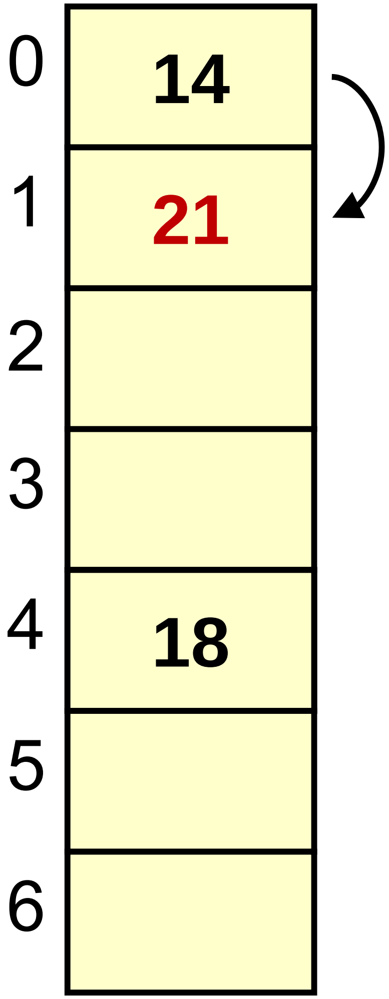
  ⋮

- $\textbf{hash}_2$ is called the **secondary hash function**
  - If $\text{hash}_2(\text{k}) = 1$, then it is the same as linear probing
  - If $\text{hash}_2(\text{k}) = d$, where *d* is a constant integer > 1, then it is the same as modified linear probing

Double Hashing: 14, 18 in, Insert 21

hash(*k*) = *k* mod 7
hash_2(*k*) = *k* mod 5

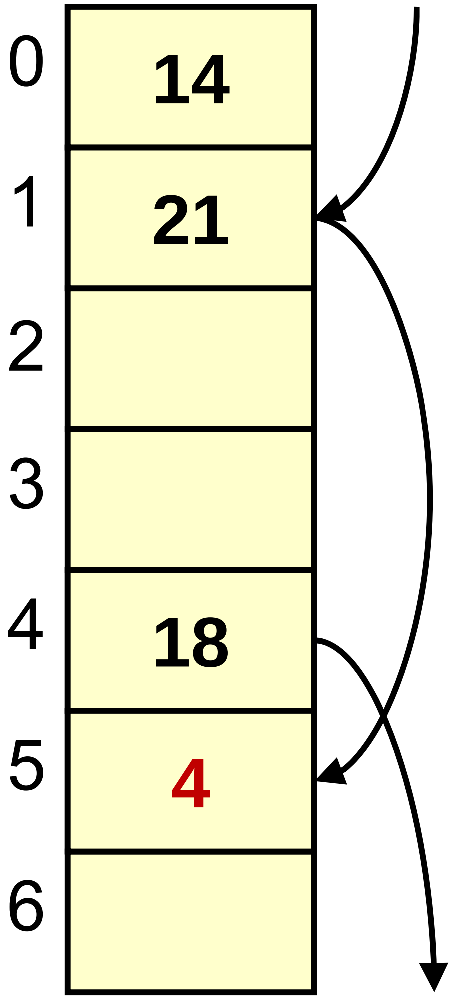
hash(21) = 0
$\text{hash}_2(21) = 1$



Double Hashing: Insert 4

hash(k) = k mod 7
hash$_2$(k) = k mod 5

hash(4) = 4
hash$_2$(4) = 4



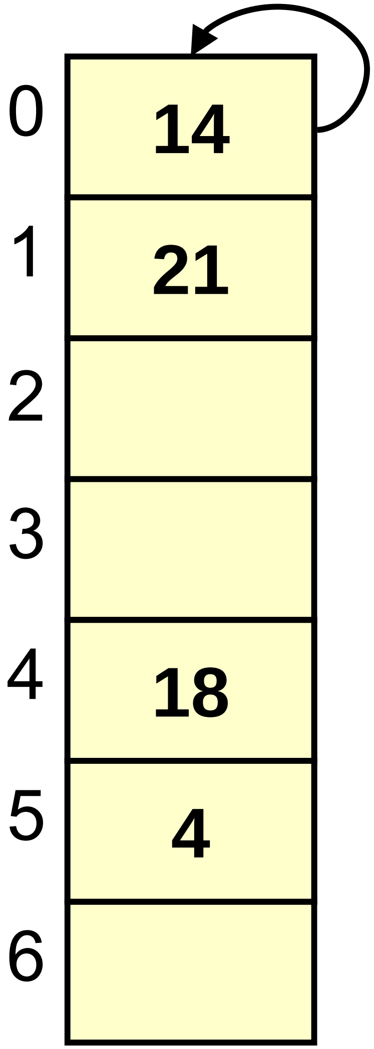
0	14
1	21
2	
3	
4	18
5	4
6	

If we insert 4, the probe sequence is 4 (home), 8, 12, …

Double Hashing: Insert 35

hash(*k*) = *k* mod 7
hash_2(*k*) = *k* mod 5

hash(35) = 0
hash_2(35) = 0



0	14
1	21
2	
3	
4	18
5	4
6	

But if we insert 35, the probe sequence is **0, 0, 0,** …

What is wrong?
Since hash_2(35)=**0**.
Not acceptable!

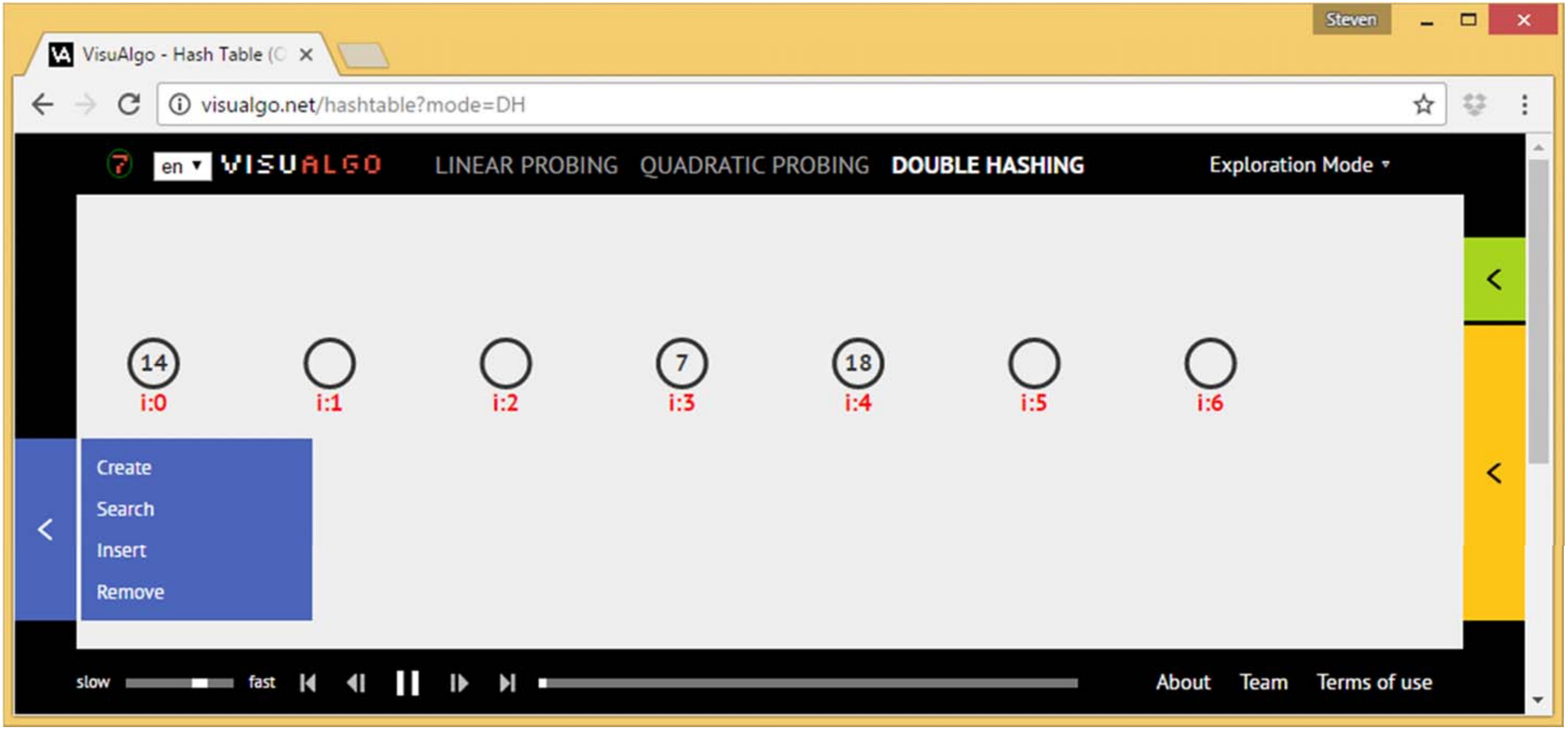
hash_2(key) must not be 0

- We can redefine hash_2(key) as
  - hash_2(key) = (key % s) + 1, or
  - hash_2(key) = s – (key % s)

- Note
  - The size of hash table must be a prime m
  - When defining hash_2(key) = (key % s) + 1
    - $s < m$ but s need not be a prime
    - Usually $s = m - 1$

VisuAlgo (Part 3)

- Hash Table with double hashing collision resolution is also in VisuAlgo (http://visualgo.net/hashtable?mode=DH)
- Currently, the secondary hash = 1+key%(HT_size-2)



Good Collision Resolution Method

- Minimize clustering
- Always find an empty slot if it exists
- Give different probe sequences when 2 keys collide (i.e. no secondary clustering)
- Fast, O(**1**)

Rehash

- Time to rehash
  - When the table is getting full, the operations are getting slow
  - For quadratic probing, insertions might fail when the table is more than half full
- Rehash operation
  - Build another table about twice as big with a new hash function
  - Scan the original table, for each key, compute the new hash value and insert the data into the new table
  - Delete the original table
- The load factor used to decide when to rehash
  - For open addressing: 0.5
  - For closed addressing: 1

Summary

- How to hash?
  - Criteria for good hash functions
- How to resolve collision?
  - Separate chaining
  - Linear probing
  - Quadratic probing
  - Double hashing
- Problem on deletions
- Primary clustering and secondary clustering