Data Structures and Algorithms I

Hashing

For efficient look-up in a table

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy, and Dr. Low Kok Lim for kindly sharing these materials.

Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Recording of modifications

- Course website address is changed to http://sakai.it.tdt.edu.vn
- Course codes cs1010, cs1020, cs2010 are placed by 501042, 501043, 502043 respectively.

Objectives

1

- To understand how **hashing** is used to accelerate table lookup

2

- To study the issue of **collision** and techniques to resolve it

References

Book

- **Chapter 13**, section 13.2, pages 761 to 787.
- Visualgo: http://visualgo.net

IT-TDT Sakai → 501043 website → Lessons

- http://sakai.it.tdt.edu.vn

Outline

1. Direct Addressing Table
2. Hash Table
3. Hash Functions
   - Good/bad/perfect/uniform hash function
4. Collision Resolution
   - Separate Chaining
   - Linear Probing
   - Quadratic Probing
   - Double Hashing
5. Summary
6. Java HashMap Class

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.

ADT Table 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

Note: Balanced Binary Search Tree (BST) will be covered in 502043 Data Structures and Algorithms II.

- Hence, hash table supports the table ADT in constant time on average for the above operations. It has many applications.

1 Direct Addressing Table

A simplified version of hash table

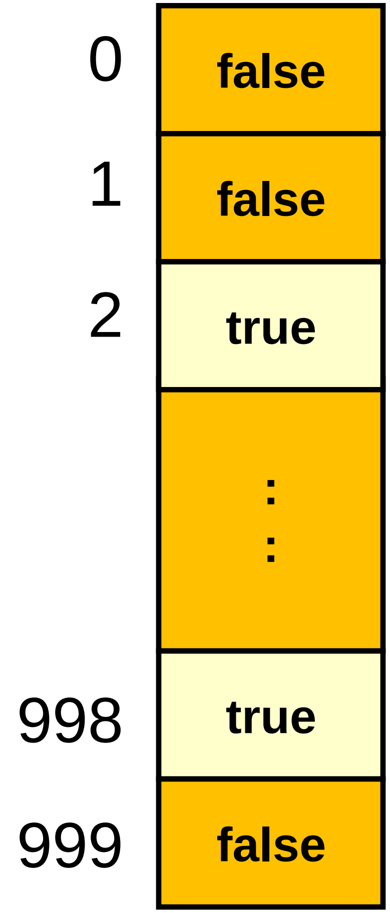
1 SBS Transit Problem

- 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*

1 SBS Transit Problem

Assume that bus numbers are integers between 0 and 999, we can create an array with 1000 Boolean values.

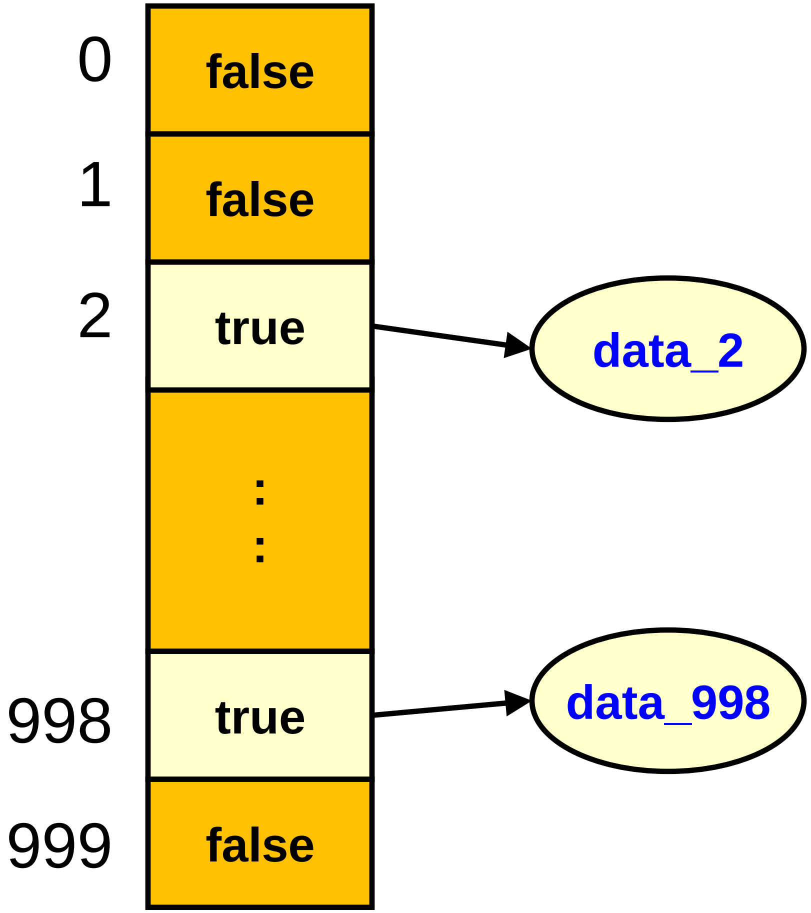
If bus service *num* exists, just set position *num* to true.



1 Direct Addressing Table (1/2)

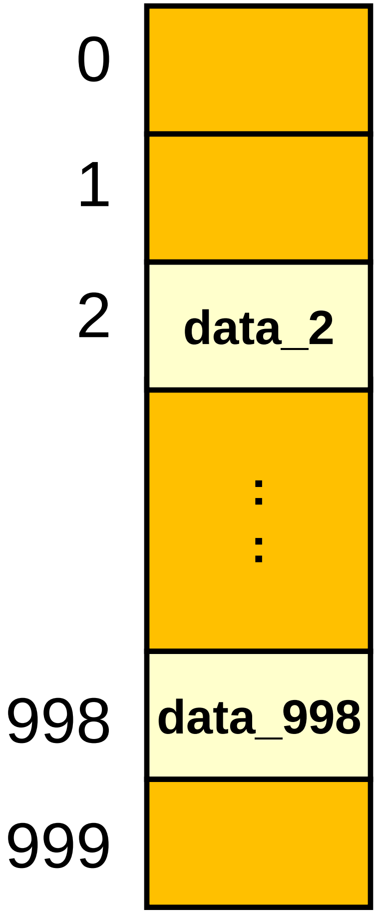
If we want to maintain additional data about a bus, use an array of 1000 slots, each can **reference** to an object which contains the details of the bus route.

Note: You may want to store the key values, i.e. bus numbers, also.



1 Direct Addressing Table (2/2)

Alternatively, we can store the data **directly in the table slots** also.



Q: What are the advantages and disadvantages of these 2 approaches?

1 Direct Addressing Table: Operations

insert (key, data)

```
a[key] = data        // where a[] is an array – the table
```

delete (key)

```
a[key] = null
```

find (key)

```
return a[key]
```

1 Direct Addressing Table: Restrictions

- Keys must be non-negative integer values
  - What happens 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?

2 Hash Table

Hash Table is a generalization of direct addressing table, to remove the latter’s restrictions.

2 Origins of the term Hash

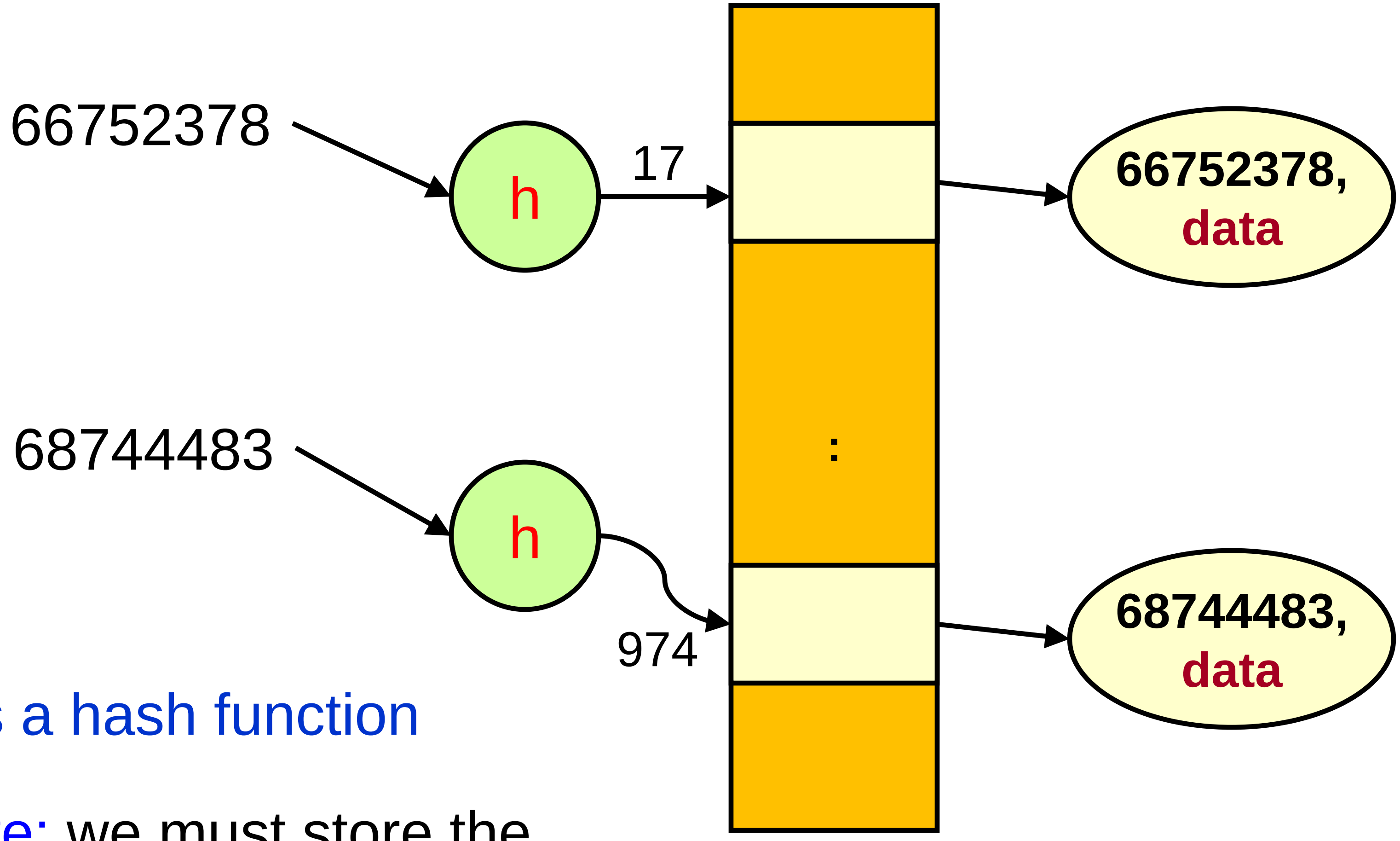
- The term "hash" comes by way of analogy with its standard meaning in the physical world, to "chop and mix".
- Indeed, typical hash functions, like the **mod** operation, “chop” the input domain into many sub-domains that get “mixed” into the output range.
- Donald Knuth notes that Hans Peter Luhn of IBM appears to have been the first to use the concept, in a memo dated January 1953, and that Robert Morris used the term in a survey paper in CACM which elevated the term from technical jargon to formal terminology.

2 Ideas

- Map large integers to smaller integers
- Map non-integer keys to integers

HASHING

2 Hash Table



h is a hash function

Note: we must store the key values. Why?

2 Hash Table: Operations

insert (key, data)

a[h(key)] = data // h is a hash function and a[] is an array

delete (key)

a[h(key)] = null

find (key)

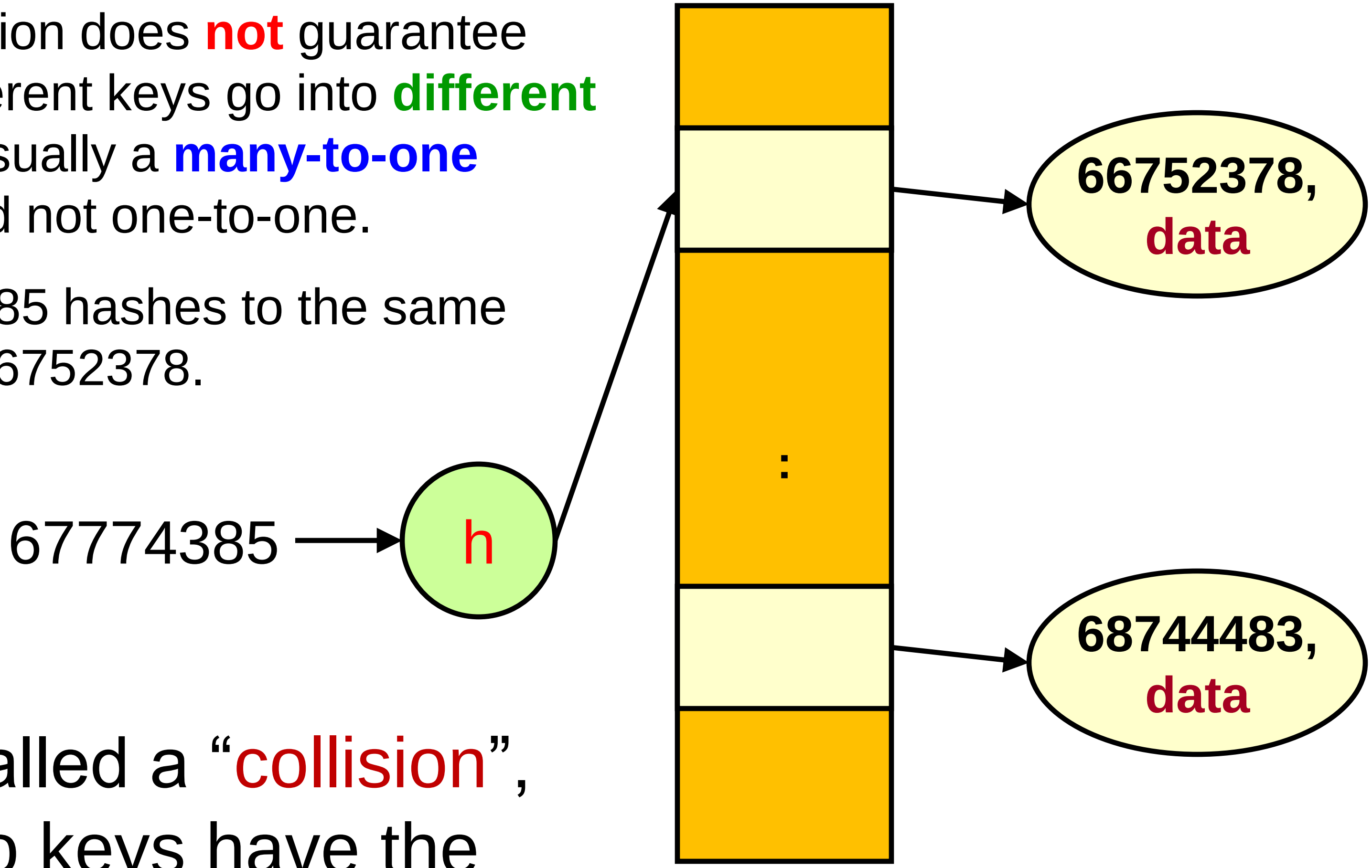
return a[h(key)]

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

2 Hash Table: Collision

A hash function does **not** guarantee that two different keys go into **different** slots! It is usually a **many-to-one** mapping and not one-to-one.

E.g. 67774385 hashes to the same location of 66752378.



This is called a "collision", when two keys have the <u>same hash value</u>.

2 Two Important Issues

- How to hash?
- How to resolve collisions?
- These are important issues that can affect the efficiency of hashing

3 Hash Functions

3 Criteria of Good Hash Functions

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

3 Example of Bad Hash Function

- Select Digits – e.g. choose the 4th and 8th digits of a phone number
  - hash(67754378) = 58
  - hash(63497820) = 90
- What happen when you hash Singapore's house phone numbers by selecting the first three digits?

3 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.

3 Uniform Hash Functions

- Distributes keys evenly in the hash table
- Example
  - If k integers are uniformly distributed among 0 and X-1, we can map the values 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, $0 \leq k < X$
$\lfloor \; \rfloor$ is the *floor* function

3 Division method (mod operator)

- Map into a hash table of ***m*** slots.
- Use the modulo operator (**%** in Java) 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.

3 How to pick *m*?

- The choice of *m* (or hash table size) is important. If *m* is power of two, say 2^n, then key modulo of *m* is the same as extracting the last *n* bits of the key.
- If *m* is 10^n, then our hash values is the last *n* digit of keys.
- Both are no good.
- Rule of thumb:
  - Pick a prime number close to a power of two to be *m*.

3 Multiplication method

1. Multiply by a constant real number **A** between 0 and 1
2. Extract the fractional part
3. Multiply by *m*, the hash table size

$$hash(k) = \lfloor m(k\mathbf{A} - \lfloor k\mathbf{A} \rfloor) \rfloor$$

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

3 Hashing of strings (1/4)

- An example hash function for strings:

```
hash(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
}
```

3 Hashing of strings: Examples (2/4)

hash("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

3 Hashing of strings: Examples (3/4)

- All 3 strings below have the same hash value! Why?
  - Lee Chin Tan
  - Chen Le Tian
  - Chan Tin Lee
- Problem: This hash function value does not depend on positions of characters! – Bad

3 Hashing of strings (4/4)

- A better hash function for strings is to “shift” the sum after each character, so that the positions of the characters affect the hash value.

```
hash(s)
    sum = 0
    for each character c in s {
        sum = sum*31 + c
    }
    return sum % m      // m is the hash table size
```

Java’s String.hashCode() uses *31 as well.

4 Collision Resolution

4 Probability of Collision (1/2)

- **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!

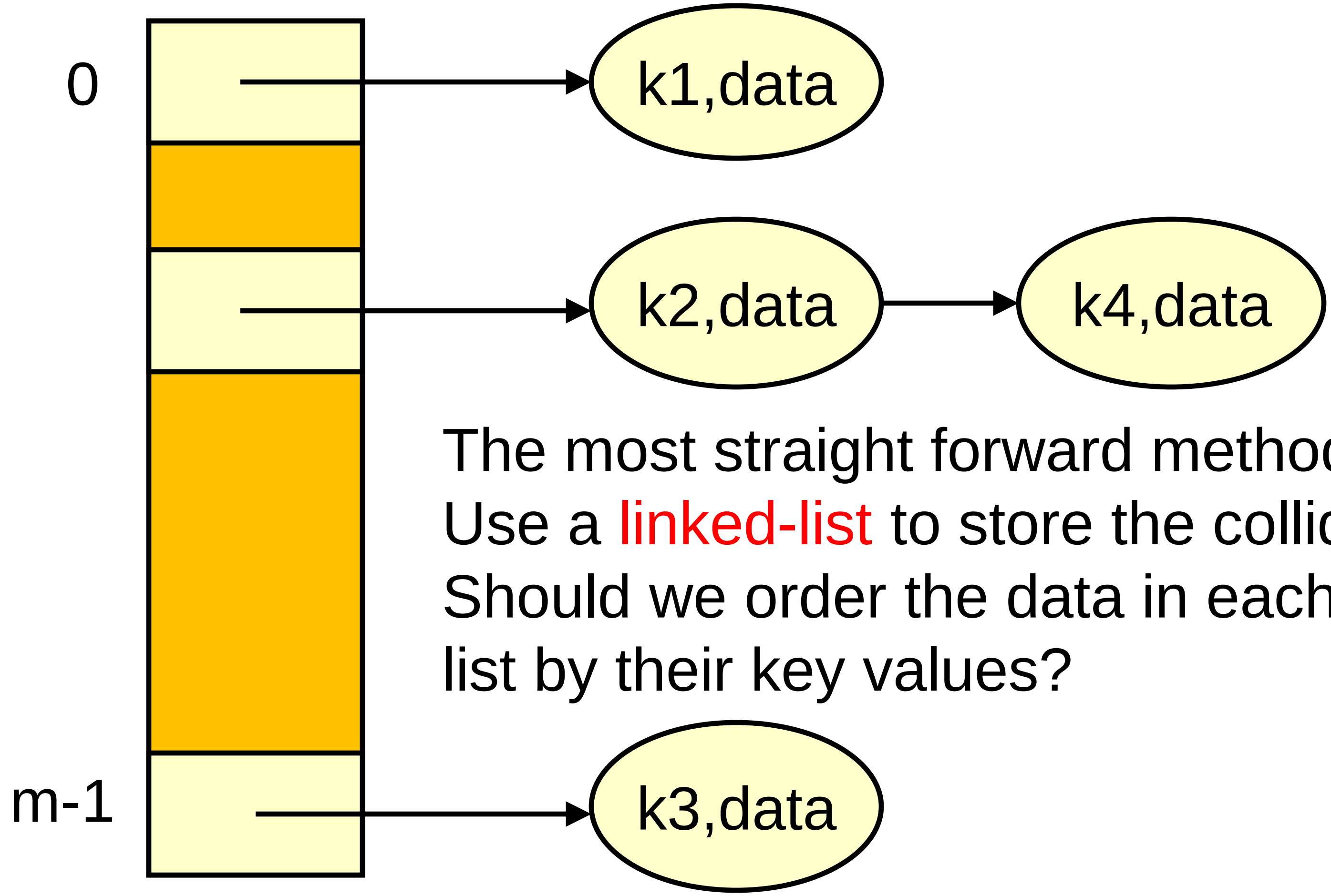
4 Probability of Collision (2/2)

- 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!

4 Collision Resolution Techniques

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

4.1 Separate Chaining



The most straight forward method.
Use a linked-list to store the collided keys.
Should we order the data in each linked list by their key values?

4.1 Hash 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(*n*) time, where *n* is length of the chain

delete (key)

Delete data from the list a[h(key)]

Takes O(*n*) time, where *n* is length of the chain

4.1 Analysis: Performance of Hash Table

- *n*: number of keys in the hash table
- *m*: size of the hash tables – number of slots
- α: load factor

$$\alpha = n/m$$

a measure of how full the hash table is. If table size is the number of linked lists, then α is the average length of the linked lists.

4.1 Reconstructing Hash Table

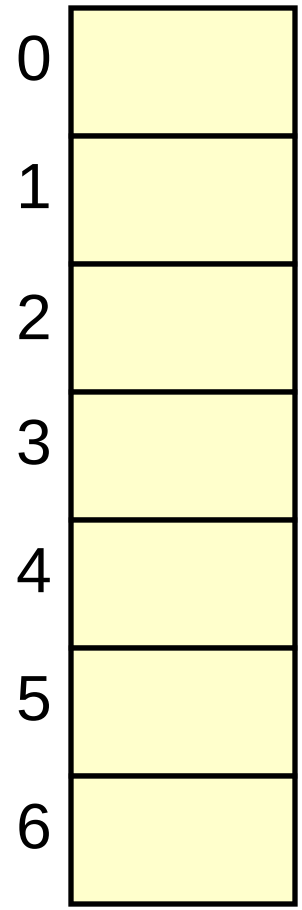
- To keep α bounded, we may need to reconstruct the whole table when the load factor exceeds the bound.
- Whenever the load factor exceeds the bound, we need to rehash all keys into a bigger table (increase *m* to reduce α), say double the table size *m*.

4.2 Linear Probing

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

Here the table size m=7

Note: 7 is a prime number.



In linear probing, when we get a collision, we scan through the table looking for the next empty slot (wrapping around when we reach the last slot).

4.2 Linear Probing: Insert 18

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

hash(18) = 18 mod 7 = 4

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

4.2 Linear Probing: Insert 14

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

hash(18) = 18 mod 7 = 4

hash(14) = 14 mod 7 = 0

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

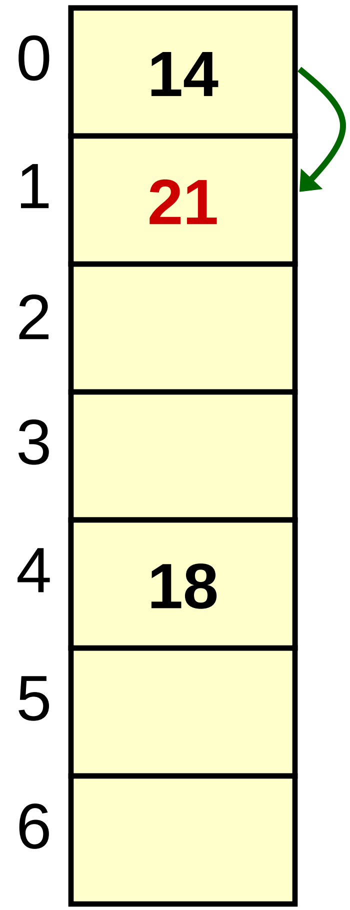
4.2 Linear Probing: Insert 21

hash(k) = k mod 7

hash(18) = 18 mod 7 = 4

hash(14) = 14 mod 7 = 0

hash(21) = 21 mod 7 = 0



Collision occurs!

Look for next empty slot.

4.2 Linear Probing: Insert 1

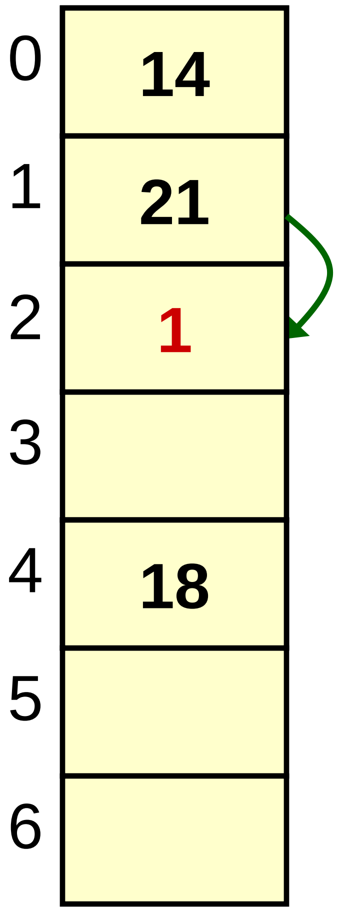
hash(*k*) = *k* mod 7

hash(18) = 18 mod 7 = 4

hash(14) = 14 mod 7 = 0

hash(21) = 21 mod 7 = 0

hash(1) = 1 mod 7 = 1



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

4.2 Linear Probing: Insert 35

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

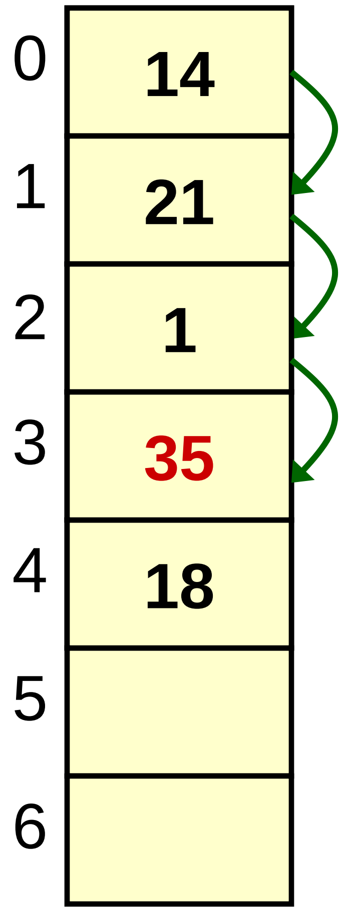
hash(18) = 18 mod 7 = 4

hash(14) = 14 mod 7 = 0

hash(21) = 21 mod 7 = 0

hash(1) = 1 mod 7 = 1

hash(35) = 35 mod 7 = 0

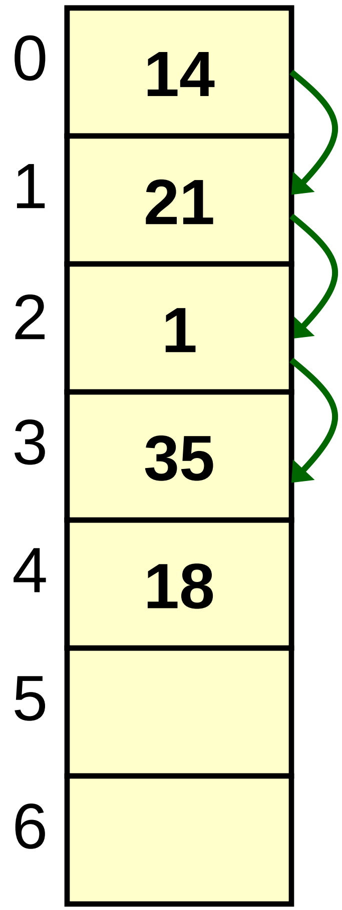


Collision, need to check next 3 slots.

4.2 Linear Probing: Find 35

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

hash(35) = 0

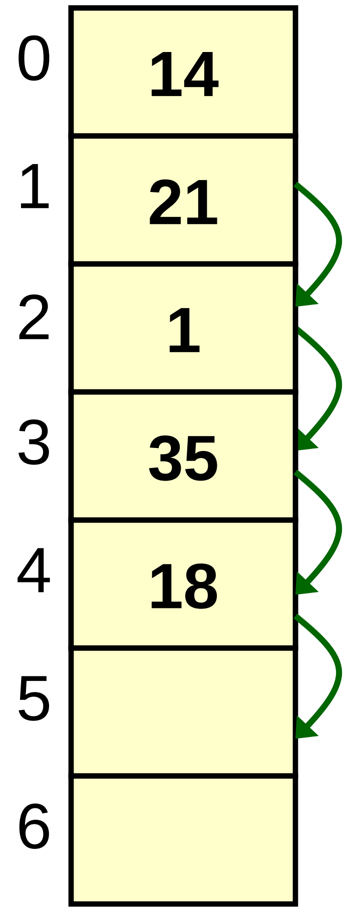


Found 35, after 4 probes.

4.2 Linear Probing: Find 8

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

hash(8) = 1



8 NOT found.
Need 5 probes!

4.2 Linear Probing: Delete 21

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

hash(21) = 0

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

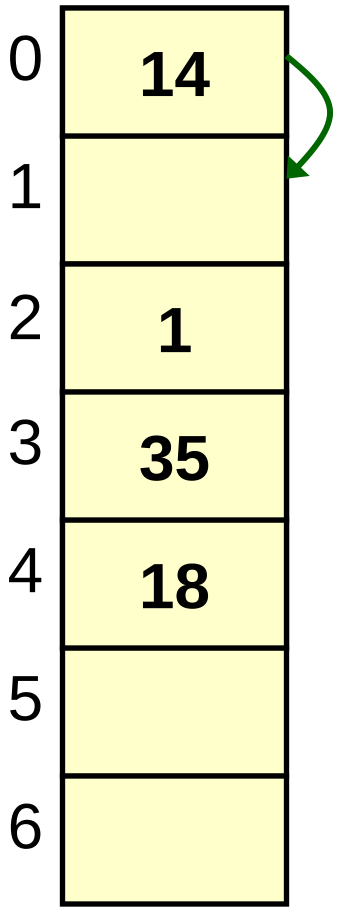
We cannot simply remove a value, because it can affect find()!

4.2 Linear Probing: Find 35

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

hash(35) = 0

Hence for deletion, cannot simply remove the key value!



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

35 NOT found!
Incorrect!

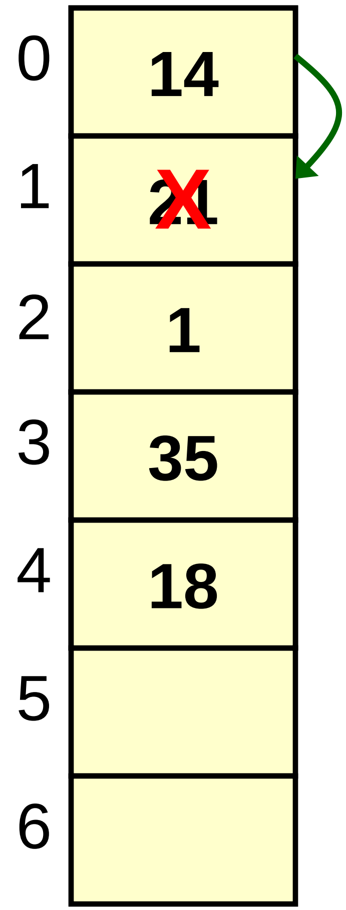
4.2 How to delete?

- Lazy Deletion
- Use three different states of a slot
  - Occupied
  - Occupied but mark as 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.

4.2 Linear Probing: Delete 21

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

hash(21) = 0

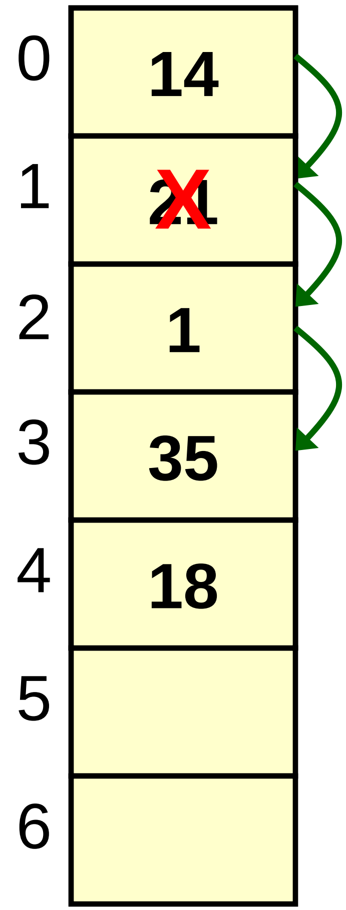


Slot 1 is occupied but now marked as deleted.

4.2 Linear Probing: Find 35

hash(k) = k mod 7

hash(35) = 0



Found 35

Now we can find 35

4.2 Linear Probing: Insert 15 (1/2)

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

hash(15) = 1

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

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).

4.2 Linear Probing: Insert 15 (2/2)

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

hash(15) = 1

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

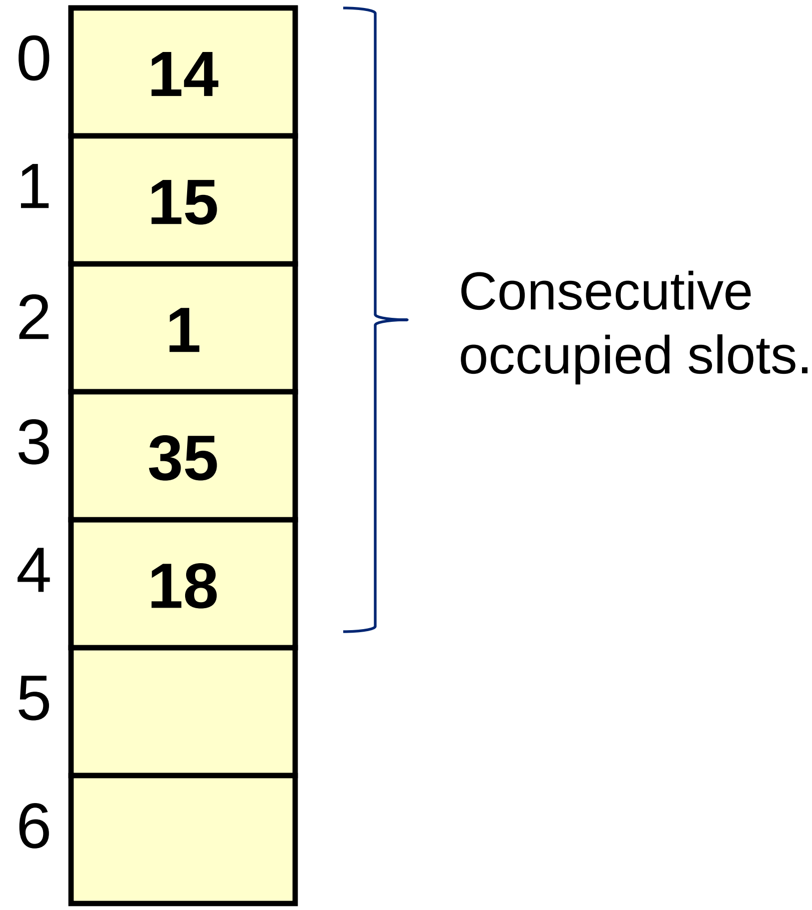
So, 15 is inserted into slot 1, which was marked as deleted.

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

4.2 Problem of Linear Probing

It can create many **consecutive occupied slots**, increasing the running time of find/insert/delete.

This is called Primary Clustering



4.2 Linear Probing

The probe sequence of this linear probing is:

hash(key)
(hash(key) + **1**) % *m*
(hash(key) + **2**) % *m*
(hash(key) + **3**) % *m*
:

4.2 Modified Linear Probing

Q: How to modify linear probing to avoid primary clustering?

We can modify the probe sequence as follows:

$$\text{hash(key)}$$
$$(\ \text{hash(key)} + \mathbf{1} * \boldsymbol{d}\ )\ \%\ m$$
$$(\ \text{hash(key)} + \mathbf{2} * \boldsymbol{d}\ )\ \%\ m$$
$$(\ \text{hash(key)} + \mathbf{3} * \boldsymbol{d}\ )\ \%\ m$$
$$\vdots$$

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

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

4.3 Quadratic Probing

For quadratic probing, the probe sequence is:

$$\text{hash(key)}$$
$$(\ \text{hash(key)} + 1\ )\ \%\ m$$
$$(\ \text{hash(key)} + 4\ )\ \%\ m$$
$$(\ \text{hash(key)} + 9\ )\ \%\ m$$
$$\vdots$$
$$(\ \text{hash(key)} + k^2\ )\ \%\ m$$

4.3 Quadratic Probing: Insert 3

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

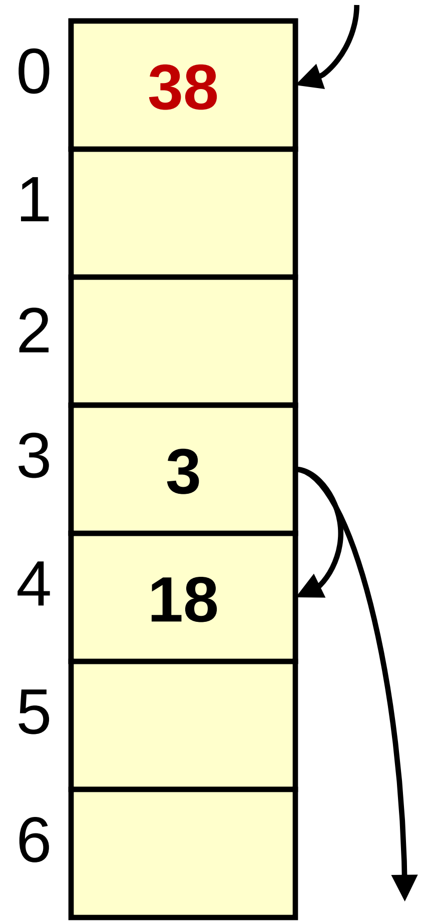
hash(3) = 3

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

4.3 Quadratic Probing: Insert 38

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

hash(38) = 3



4.3 Theorem of Quadratic Probing

- 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)
- Note: $\alpha < 0.5$ means the hash table is less than half full.
- Q: How can we be sure that quadratic probing always terminates?
- Insert 12 into the previous example, followed by 10. See what happen?

4.3 Problem of Quadratic Probing

- If two keys have the same initial position, their probe sequences are the same.
- This is called secondary clustering.
- But it is not as bad as linear probing.

4.4 Double Hashing

Use 2 hash functions:

$$\text{hash(key)}$$
$$(\ \text{hash(key)} + 1*\text{hash}_2\text{(key)}\ )\ \%\ m$$
$$(\ \text{hash(key)} + 2*\text{hash}_2\text{(key)}\ )\ \%\ m$$
$$(\ \text{hash(key)} + 3*\text{hash}_2\text{(key)}\ )\ \%\ m$$
$$\vdots$$

hash_2 is called the secondary hash function, the number of slots to jump each time a collision occurs.

4.4 Double Hashing: Insert 21

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

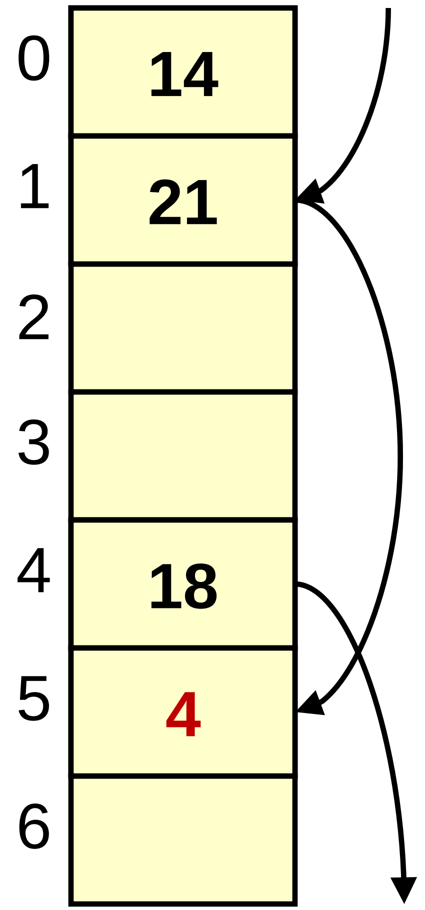
hash(21) = 0
hash$_2$(21) = 1

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

4.4 Double Hashing: Insert 4

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

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

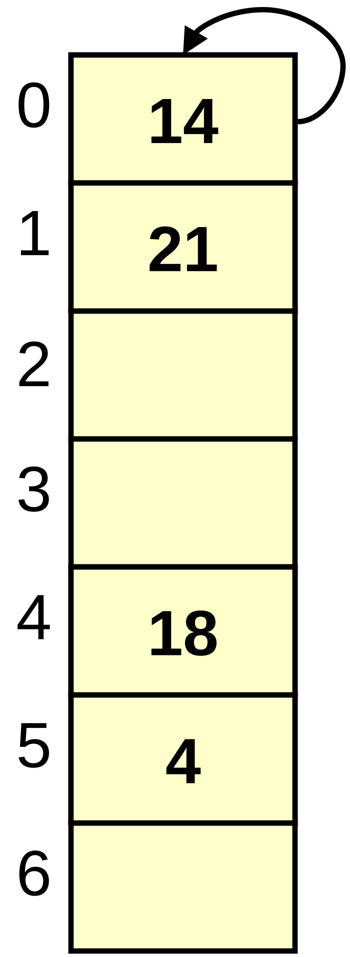


If we insert 4, the probe sequence is 4, 8, 12, …

4.4 Double Hashing: Insert 35

hash(*k*) = *k* mod 7
$hash_2(k) = k \bmod 5$

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



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

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

4.4 Warning

- **Secondary hash function must not evaluate to 0!**
- To solve this problem, simply change $hash_2(key)$ in the above example to:

$$hash_2(key) = 5 - (key \% 5)$$

Note:

- If $hash_2(k) = 1$, then it is the same as linear probing.
- If $hash_2(k) = d$, where d is a constant integer > 1, then it is the same as modified linear probing.

4.5 Criteria of Good Collision Resolution Method

- Minimize clustering
- Always find an empty slot if it exists
- Give different probe sequences when 2 initial probes are the same (i.e. no secondary clustering)
- Fast

ADT Table 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

Note: Balanced Binary Search Tree (BST) will be covered in 502043 Data Structures and Algorithms II.

5 Summary

- How to hash? Criteria for good hash functions?
- How to resolve collision?

  Collision resolution techniques:
  - separate chaining
  - linear probing
  - quadratic probing
  - double hashing
- Problem on deletions
- Primary clustering and secondary clustering.

6 Java HashMap Class

6 Class HashMap <K, V>

```
public class HashMap<K,V>
  extends AbstractMap<K,V>
  implements Map<K,V>, Cloneable, Serializable
```

- This class implements a hash map, which maps keys to values. Any non-null object can be used as a key or as a value.
  e.g. We can create a hash map that maps people names to their ages. It uses the names as keys, and the ages as the values.
- The AbstractMap is an abstract class that provides a skeletal implementation of the Map interface.
- Generally, the default load factor (0.75) offers a good tradeoff between time and space costs.
- The default HashMap capacity is 16.

6 Class HashMap <K, V>

- **Constructors** summary
  - `HashMap()`

    Constructs an empty HashMap with a default initial capacity (16) and the default load factor of 0.75.
  - `HashMap(int initialCapacity)`

    Constructs an empty HashMap with the specified initial capacity and the default load factor of 0.75.
  - `HashMap(int initialCapacity, float loadFactor)`

    Constructs an empty HashMap with the specified initial capacity and load factor.
  - `HashMap(Map<? extends K, ? extends V> m)`

    Constructs a new HashMap with the same mappings as the specified Map.

6 Class HashMap <K, V>

Some methods

- `void clear()`

  Removes all of the mappings from this map.
- `boolean containsKey(Object key)`

  Returns true if this map contains a mapping for the specified key.
- `boolean containsValue(Object value)`

  Returns true if this map maps one or more keys to the specified value.
- `V get(Object key)`

  Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
- `V put(K key, V value)`

  Associates the specified value with the specified key in this map.
- …

6 Example

- Example: Create a hashmap that maps people names to their ages. It uses names as key, and the ages as their values.

```
HashMap<String, Integer> hm = new HashMap<String, Integer>();
// placing items into the hashmap
hm.put("Mike", 52);
hm.put("Janet", 46);
hm.put("Jack", 46);
// retrieving item from the hashmap
System.out.println("Janet => " + hm.get("Janet"));
```

TestHash.java

The output of the above code is:

```
Janet => 46
```

End of file