

CS1102: Lecture 11



Hashing

Chapter 13: pages 695 - 719

Hashing



ADT table operations

	Sorted Array	Balanced BST	
Insertion	$O(n)$	$O(\log n)$	
Deletion	$O(n)$	$O(\log n)$	
Retrieval	$O(\log n)$	$O(\log n)$	

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

Q: What is the meaning of $O(1)$?

Direct Addressing Table

- a simplified version of hash table

SBS bus problem

- Retrieval **find**(N)
 - Find the bus route of bus service no. N
- Insertion **insert**(N)
 - Introduce a new bus service no. N
- Deletion **delete**(N)
 - Remove bus service no. N

SBS bus problem

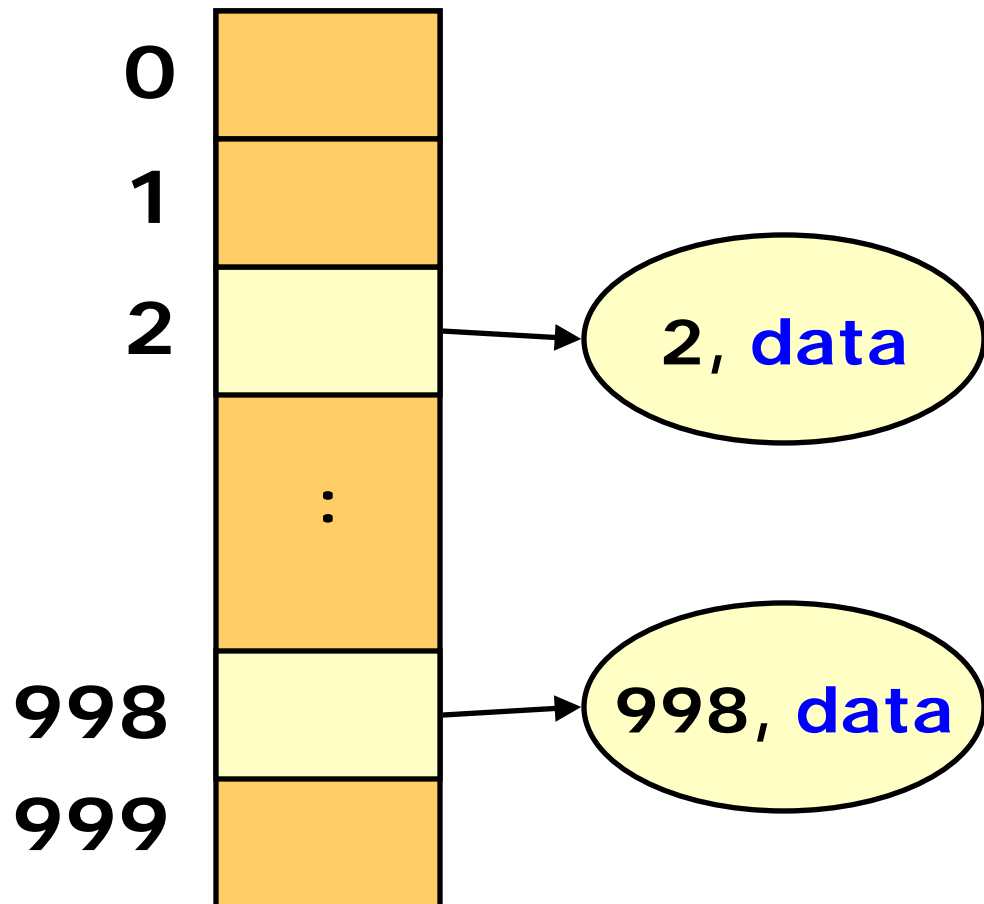
Assume that bus numbers are integers between **1 to 999**, we can create an array with 1000 booleans.

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

0	false
1	false
2	true
	:
	:
998	true
999	false

Direct addressing table

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.



Direct addressing table

Alternatively, we can store the record (key value and other data) **directly in the table slots** also.

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

0	
1	
2	2, data
	:
998	998, data
999	

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]

Restrictions of direct addressing table

- 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?

Hash Table



Hash Table is a **generalization** of direct addressing table, to remove these restrictions

Origins of the term **Hash**

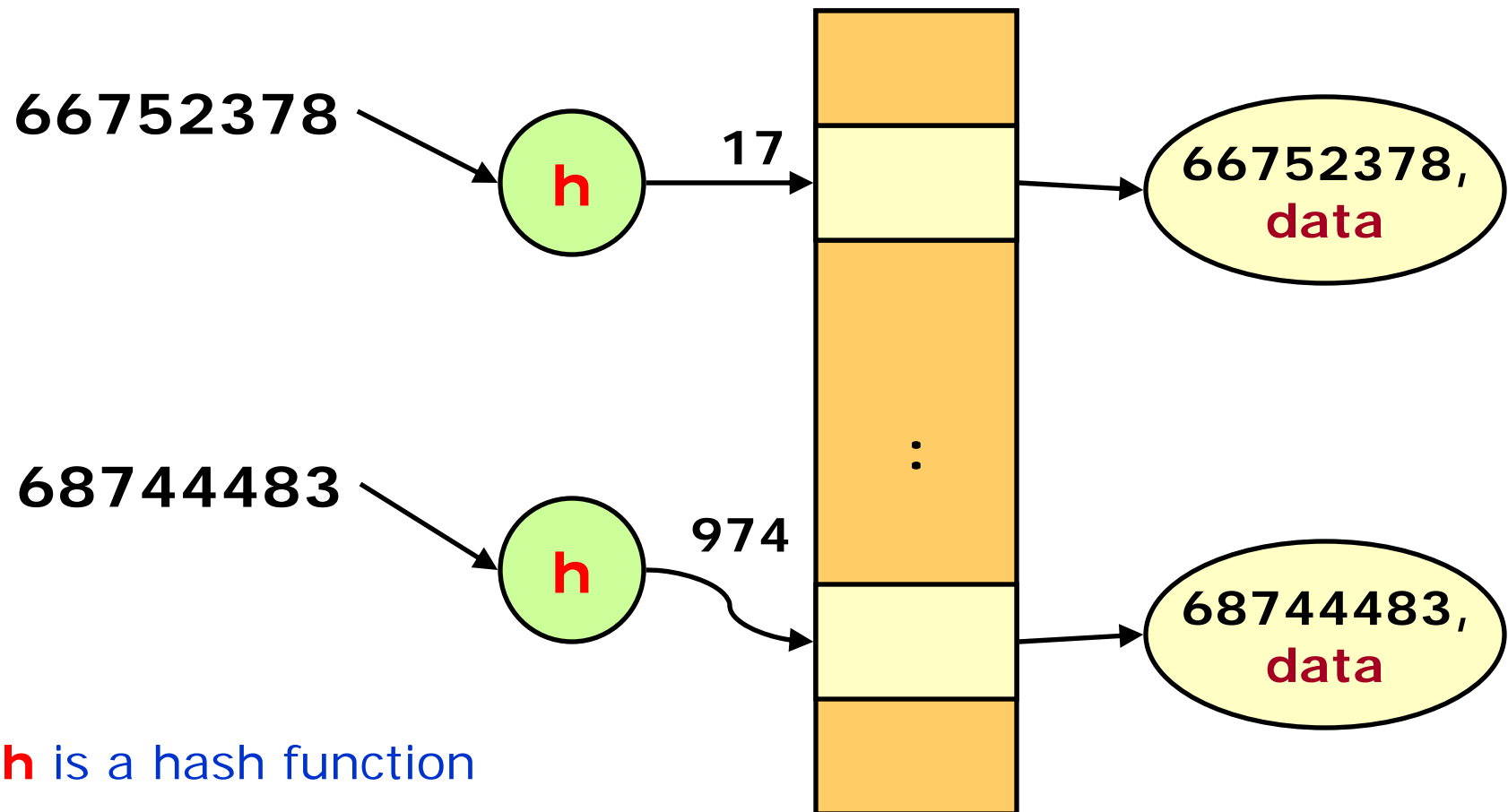
- ❑ The term "hash" comes by way of analogy with its standard meaning in the physical world, to "**chop and mix**".
- ❑ Indeed, typical hash functions, like the mod operation, "chop" the input domain into many sub-domains that get "mixed" into the output range.
- ❑ Donald Knuth notes that Hans Peter 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.

I deas

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

HASHING

Hash table



Hash table **operations**

insert (key, data)

$a[h(\text{key})] = \text{data}$ // h is a **hash function** and $a[]$ is an array

delete (key)

$a[h(\text{key})] = \text{null}$

find (key)

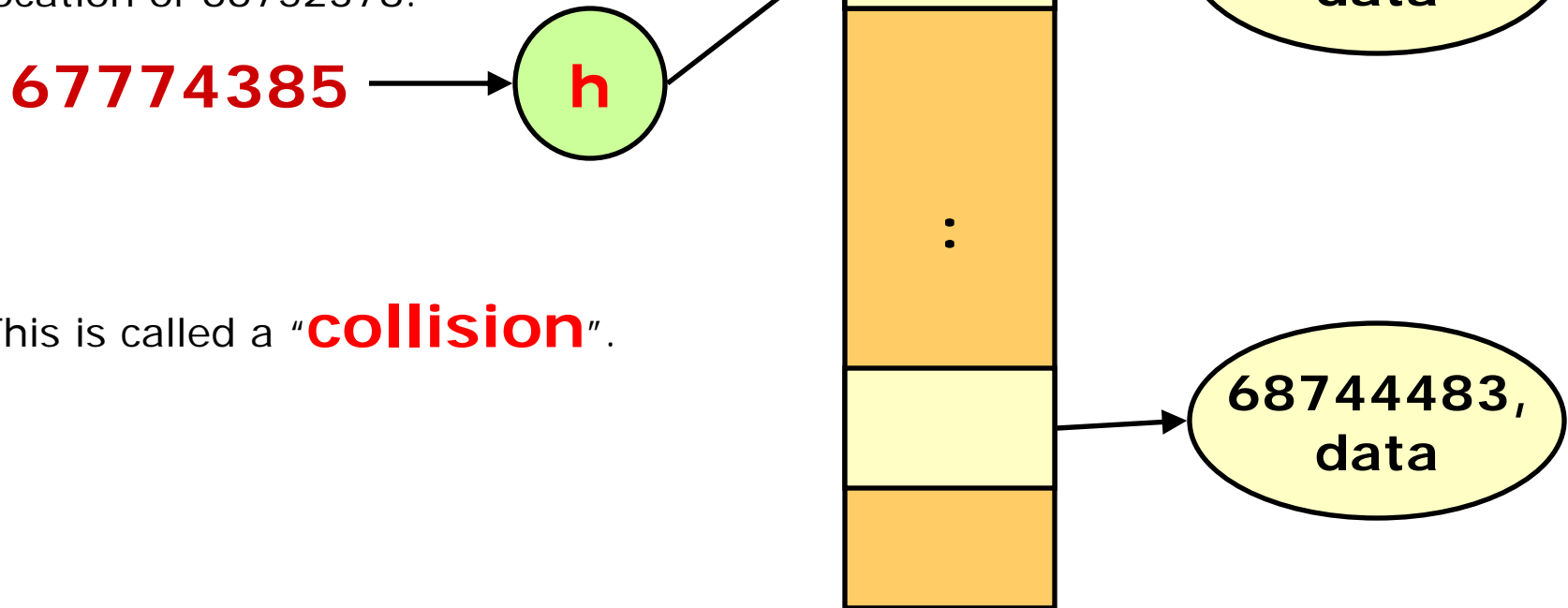
return $a[h(\text{key})]$

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

Hash table

A hash function does **not** guarantee that two different keys goes 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**".

Problem

- Two keys can have the **same** hash value

collision occurs!

Two important issues

- How to hash?
- How to resolve collisions?

Hash Functions



Criteria of Good Hash Functions

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

Examples of

Bad Hash Functions

- Select Digits - e.g. choose the 4th and 8th digits of a phone no.
 - $\text{hash}(67754378) = 58$
 - $\text{hash}(63497820) = 90$

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

Perfect Hash Function

- ❑ 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.

Uniform Hash Function

- Distributes keys **evenly** in the hash table
- Example
 - if k are integers **uniformly** distributed among 0 and $X-1$,
we can map the values to a hash table of size **m** ($m < X$)
using the below hash function

$k \in [0, X)$ // k is the key value

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

// the **floor**

Q: What are the meanings of “[” and “)””? **Closed** and **open** intervals.

Hash function:

Division method

- Map into a hash table of m slots.
- Use the modulo operator ($\%$ in Java) to map an integer to a value between 0 and $m-1$.

$$\textit{hash}(k) = k \% m$$

The most popular method.

mod operator

- $n \bmod m = \text{remainder}$ of n divided by m , where n and m are positive integers.

How to pick **m**?

- $m = 100$
- $m = 16$
- $m = 73$
- 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 of 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 m to be a **prime number** close to a power of two.
- **Q:** What is the table size using mod function? Same, smaller, or bigger than m ?

Hash function:

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

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

The golden ratio = $(\sqrt{5} - 1)/2$
seems to be a good choice for A .

Hash function:

Hashing of strings

An example hash function for strings:

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

Example:

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

Hashing of strings (cont.)

- Lee Chin Tan
- Chen Le Tian
- Chan Tin Lee

All the 3 strings have the same hash value! Why?

Problem: This hash function value does not depend on positions of characters!

Bad!

Hashing of strings (cont.)

A **better hash function** for strings:

hash(s)

sum = 0

foreach character c in s

sum = **sum***37 + c

return sum % m // m is the hash table size

A better way is to “**shift**” the sum every time, so that the positions of characters affect the hash value.

(**Note**: Java’s String.hashCode() uses 31 instead of 37)

Collision Resolution



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?"

Probability of collision (cont.)

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

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

$P(n)$ = Probability of **collisions** (same birthday) for n people
 $= 1 - Q(n)$

$$P(\mathbf{23}) = \mathbf{0.507}$$

This means if there are **23** people in a room, the probability that some people share a birthday is **50.7%**.

So, if we insert **23** keys into a table with 365 slots, more than half of the time we get collisions. Such a result is counter-intuitive to many.

Probability of collision (cont.)

So collision is very likely!

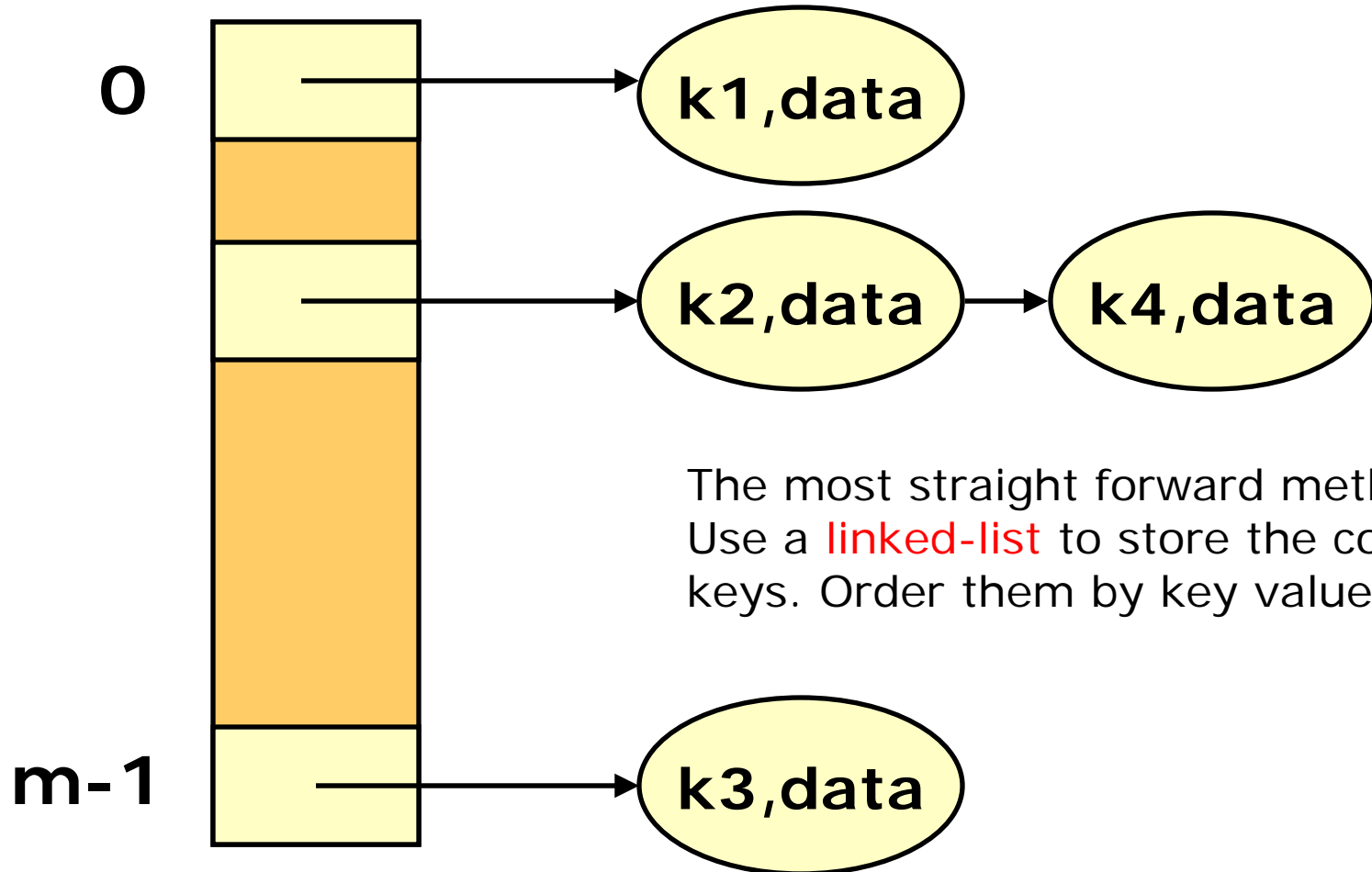
Collision resolution techniques

How to resolve the collision problem?

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

Collision resolution technique:

Separate chaining



The most straight forward method.
Use a **linked-list** to store the collided
keys. Order them by key values.

Separate chaining

Hash table

insert (key, data)

insert data into the **list** $a[h(\text{key})]$

delete (key)

delete data from the **list** $a[h(\text{key})]$

find (key)

find key from the **list** $a[h(\text{key})]$

Analysis the performance of Hash Table

- **n**: number of keys in the hash table
- **m**: size of the hash tables – no of slots
- **α** : load factor

$$\alpha = n/m$$

a measure of how full the hash table is.

Average running time

- Find $O(1 + \alpha)$ where α is the load factor
- Insert $O(1)$
- Delete $O(1 + \alpha)$

- If α is bounded by some constant, then all three operations are $O(1)$.
- We can bound the length of the chain by a constant. How?

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.

Collision resolution technique:

Linear probing

$$\text{hash}(k) = k \bmod 7$$

Here the table size $m=7$

Note: 7 is a prime number.

0	
1	
2	
3	
4	
5	
6	

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).

Linear probing

Insert 18

$\text{hash}(k) = k \bmod 7$

$\text{hash}(18) = 18 \bmod 7 = 4$

0	
1	
2	
3	
4	18
5	
6	

Linear probing

Insert 14

$\text{hash}(k) = k \bmod 7$

0	14
1	
2	
3	
4	18
5	
6	

$\text{hash}(14) = 14 \bmod 7 = 0$

Linear probing

Insert 21

$\text{hash}(k) = k \bmod 7$

0	14
1	21
2	
3	
4	18
5	
6	

$\text{hash}(21) = 21 \bmod 7 = 0$

Collision occurs!

Look for **next empty slot**.

Insert 1

$\text{hash}(k) = k \bmod 7$

0	14
1	21
2	1
3	
4	18
5	
6	

$\text{hash}(1) = 1 \bmod 7 = 1$
Collides with 21 (hash value 0),
look for an empty slot.

Linear probing

Insert 35

$$\text{hash}(k) = k \bmod 7$$

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

$\text{hash}(35) = 35 \bmod 7 = 0$.
Collision, need to check next
3 slots.

Linear probing

Find 35

$\text{hash}(k) = k \bmod 7$

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

$\text{Hash}(35) = 0$

FOUND 35
But need **4** probes

Linear probing

Find 8

$\text{hash}(k) = k \bmod 7$

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

$\text{Hash}(8) = 1$

8 NOT FOUND

Need **5** probes!

Delete 21

$$\text{hash}(k) = k \bmod 7$$

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

$$\text{Hash}(21) = 0$$

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

Find 35

$$\text{hash}(k) = k \bmod 7$$

0	14
1	
2	1
3	35
4	18
5	
6	

$$\text{Hash}(35) = 0$$

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

35 NOT FOUND!
Incorrect!

Problem on deletion

**Cannot simply remove the
key value!**

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.

Linear probing

Delete 21

$\text{hash}(k) = k \bmod 7$

Delete 21

$\text{Hash}(21) = 0$

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

Slot 1 is occupied
but **marked as deleted**.

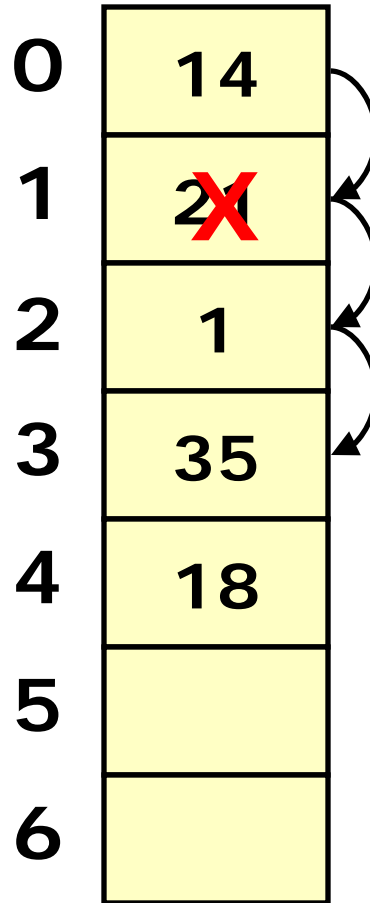
Linear probing

Find 35

$\text{hash}(k) = k \bmod 7$

$\text{Hash}(35) = 0$

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



FOUND 35

Now we can find 35

Insert 15

$\text{hash}(k) = k \bmod 7$

Insert 15

$\text{hash}(15) = 1$

0	14
1	21
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.

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

Insert 15

$\text{hash}(k) = k \bmod 7$

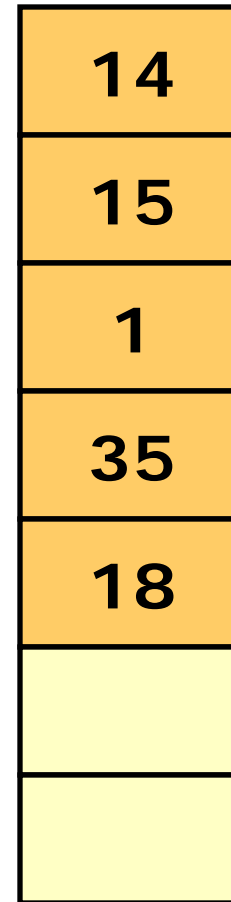
$\text{Hash}(15) = 1$

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

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

Problem of linear probing

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



Problem of linear probing

This is called

Primary Clustering

Linear probing

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

$$\begin{aligned} &\text{hash(key)} \\ &(\text{hash(key)} + \mathbf{1}) \% m \\ &(\text{hash(key)} + \mathbf{2}) \% m \\ &(\text{hash(key)} + \mathbf{3}) \% m \\ &\vdots \end{aligned}$$

Modified Linear probing

Q: How to Modify linear probing to **avoid** primary clustering?

We can **modify** the probe sequence as below:

$$\begin{aligned} & \text{hash(key)} \\ & (\text{hash(key)} + \mathbf{1 * d}) \% m \\ & (\text{hash(key)} + \mathbf{2 * d}) \% m \\ & (\text{hash(key)} + \mathbf{3 * d}) \% m \\ & \quad \vdots \end{aligned}$$

where **d** is some constant integer and **d > 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.

Collision resolution technique:

Quadratic probing

For quadratic probing, the probe sequence is

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

$(\text{hash}(\text{key}) + 1) \% m$ jump 1^2

$(\text{hash}(\text{key}) + 4) \% m$ jump 2^2

$(\text{hash}(\text{key}) + 9) \% m$ jump 3^2

:

Quadratic Probing

Insert 3

$\text{hash}(k) = k \bmod 7$

$\text{Hash}(3) = 3$

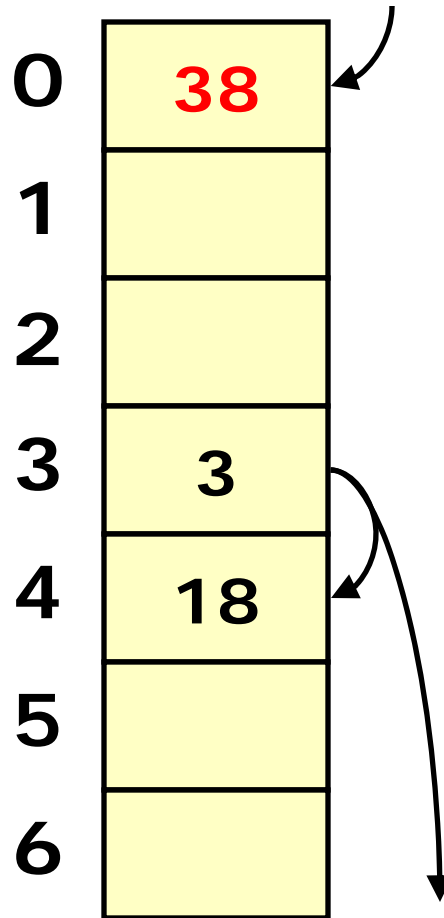
0	
1	
2	
3	3
4	18
5	
6	

Quadratic Probing

Insert 38

$$\text{hash}(k) = k \bmod 7$$

$$\text{hash}(38) = 3$$



Theorem of quadratic probing

- If $\alpha < 0.5$, and m is prime, then we can always find an empty slot.

where 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, follow by 10.
See what happen?

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.

Collision resolution technique:

Double hashing – use 2 hash functions

Use 2 hash functions

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

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

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

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

:

hash_2 is called the secondary hash function, the no of slots to jump each time a collision occurs.

Double Hashing

Insert 21

$$\text{hash}(k) = k \bmod 7$$

$$\text{hash}_2(k) = k \bmod 5$$

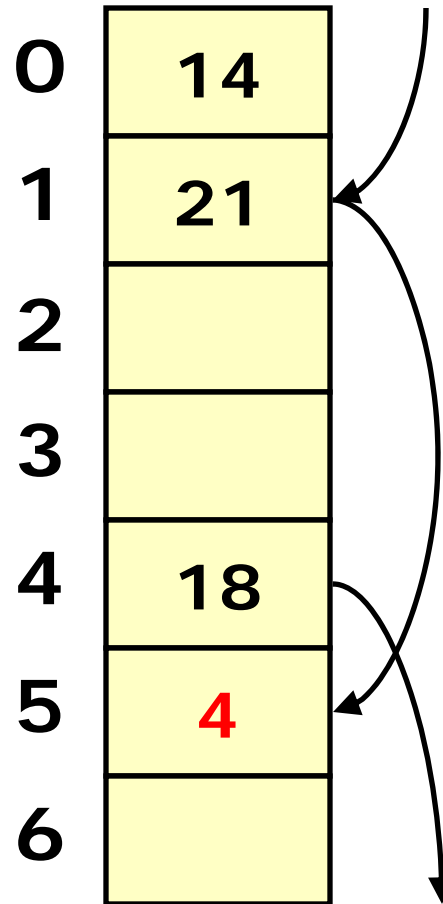
0	14
1	21
2	
3	
4	18
5	
6	

$$\begin{aligned}\text{hash}(21) &= 21 \bmod 7 = 0 \\ \text{hash}_2(21) &= 21 \bmod 5 = 1\end{aligned}$$

Insert 4

$$\text{hash}(k) = k \bmod 7$$

$$\text{hash}_2(k) = k \bmod 5$$



$$\text{Hash}(4) = 4$$

$$\text{hash}_2(4) = 4$$

If we insert 4, the probe sequence is **4, 8, 12 ...**

Double Hashing

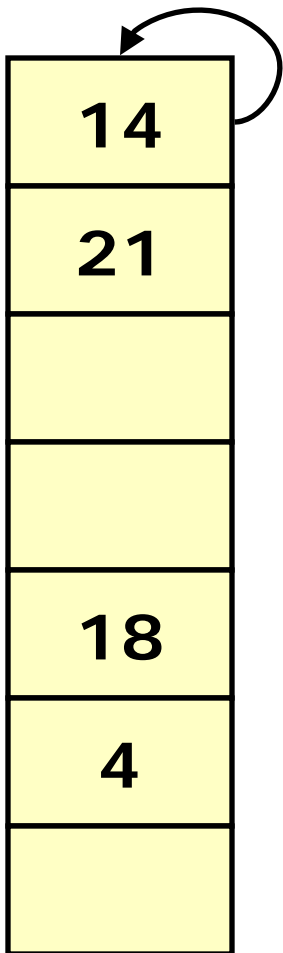
Insert 35

$$\text{hash}(k) = k \bmod 7$$

$$\text{hash}_2(k) = k \bmod 5$$

$$\text{Hash}(35) = 0$$

$$\text{hash}_2(35) = 0$$



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 $\text{hash}_2(35) = \mathbf{0}$.
Not acceptable!

Warning

- ❑ Secondary hash function must **not** evaluate to **0** !
- ❑ To solve this problem, simply change $\text{hash}_2(\text{key})$ in the above example to:

$$\text{hash}_2(\text{key}) = 5 - (\text{key} \% 5)$$

Note: If $\text{hash}_2(k) = 1$, then it is the same as linear probing.

If $\text{hash}_2(k) = d$, where d is a constant and $d > 1$, then it is the same as modified linear probing.

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

Compare BST and hashing

- ❑ Balanced BST has $O(\log n)$ operations in the **worst** case; hashing has $O(1)$ operations in the **average** case.
- ❑ **Ordered traversal** of all items
 - $O(n)$ time for BST
 - $O(n \log n)$ for hashing, **bad!**
Key values are **not ordered** in the hash tables.
- ❑ How about search for **top/bottom value, range search** (between values X and Y , where $X < Y$)?
 - Same as ordered traversal of all items, hashing technique is **no good**.
 - Complexity?
- ❑ When do we use hashing technique?

Summary of This Lecture

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

Collision resolution techniques:

- **separate chaining**
- **linear probing**
- **quadratic probing**
- **double hashing**
- **Preliminary** clustering and **secondary** clustering. Techniques?

Class Hashtable<K,V>

```
public class Hashtable<K,V>
```

```
extends Dictionary<K,V>
```

```
implements Map<K,V>, Cloneable, Serializable
```

- This class implements a hashtable, 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 table that maps people names to their ages. It uses the names as keys, and the ages as the values.
- The **Dictionary** class is the abstract parent of any class, such as Hashtable, which maps keys to values.
- Generally, the default **load factor** (**0.75**) offers a good tradeoff between time and space costs.
- The default hash table size is **11**.

Class Hashtable<K,V> (cont.)

□ Constructor Summary

■ Hashtable()

Constructs a new, empty hashtable with a default initial capacity (**11**) and load factor, which is **0.75**.

■ Hashtable(int initialCapacity)

Constructs a new, empty hashtable with the specified initial capacity and default load factor, which is **0.75**.

■ Hashtable(int initialCapacity, float loadFactor)

Constructs a new, empty hashtable with the specified initial capacity and the specified load factor.

Class Hashtable<K,V> (cont.)

Some Methods

- ❑ void **clear()**
Clears this hashtable so that it contains no keys.
- ❑ boolean **contains**(Object **value**)
Tests if some key maps into the specified value in this hashtable.
- ❑ boolean **containsKey**(Object **key**)
Tests if the specified object is a key in this hashtable.
- ❑ boolean **containsValue**(Object **value**)
Returns true if this Hashtable maps one or more keys to this value.
Note that this method is identical in functionality to the contains method.
- ❑ V **get**(Object key)
Returns the value to which the specified key is mapped in this hashtable.
- ❑ V **put**(K key, V value)
Maps the specified key to the specified value in this hashtable.
- ❑ ...

Example

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

```
Hashtable<string, Integer> ht = new Hashtable<String, Integer>();
```

```
// placing items into the hash table
```

```
ht.put("Mike", 52);
```

```
ht.put("Janet", 46);
```

```
ht.put("Jack", 46);
```

```
// retrieving item from the hash table
```

```
System.out.println("Janet => " + ht.get("Janet"));
```

The output of the above code is:

Janet => 46