CS2040 Lecture Note #8: **Hashing**

For efficient look-up in a table

Lecture Note #10: Hashing

- **Objectives:**
  - To understand how hashing is used to accelerate table lookup
  - To study the issue of collision and techniques to resolve it

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 Hashtable 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

1 Direct Addressing Table

A simplified version of hash table

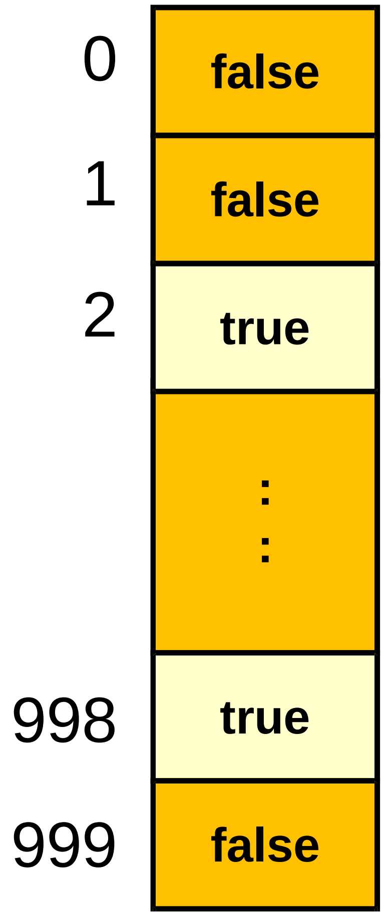
1 SBS Transit Problem

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

1 SBS Transit Problem

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

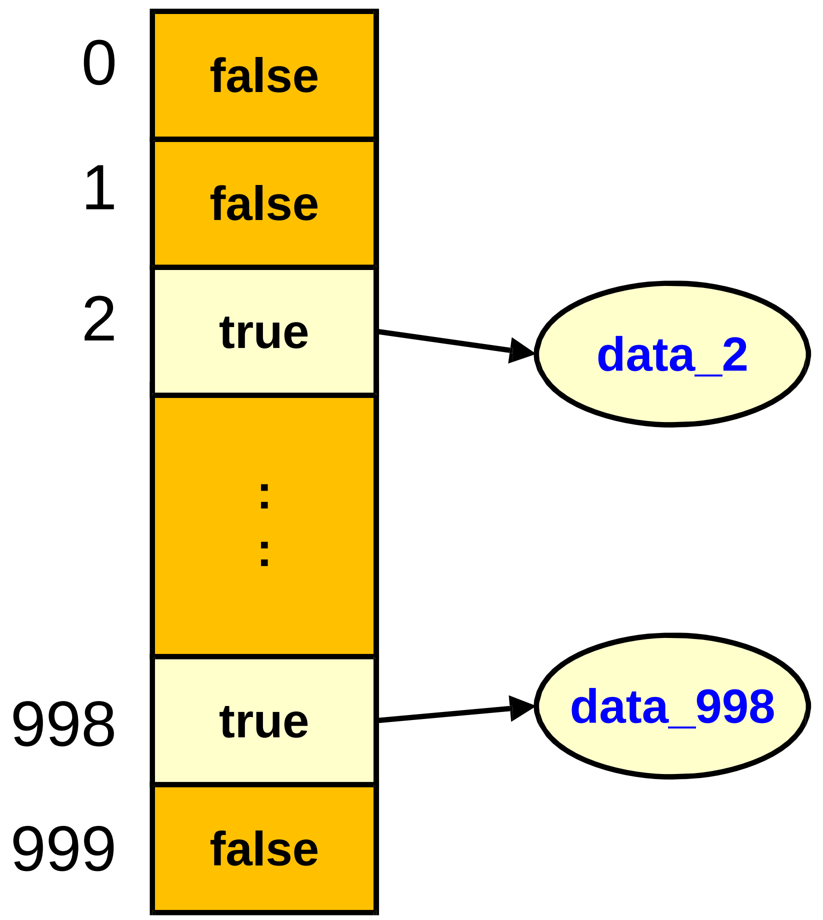
If bus service N exists, just set position N to true.



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: 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

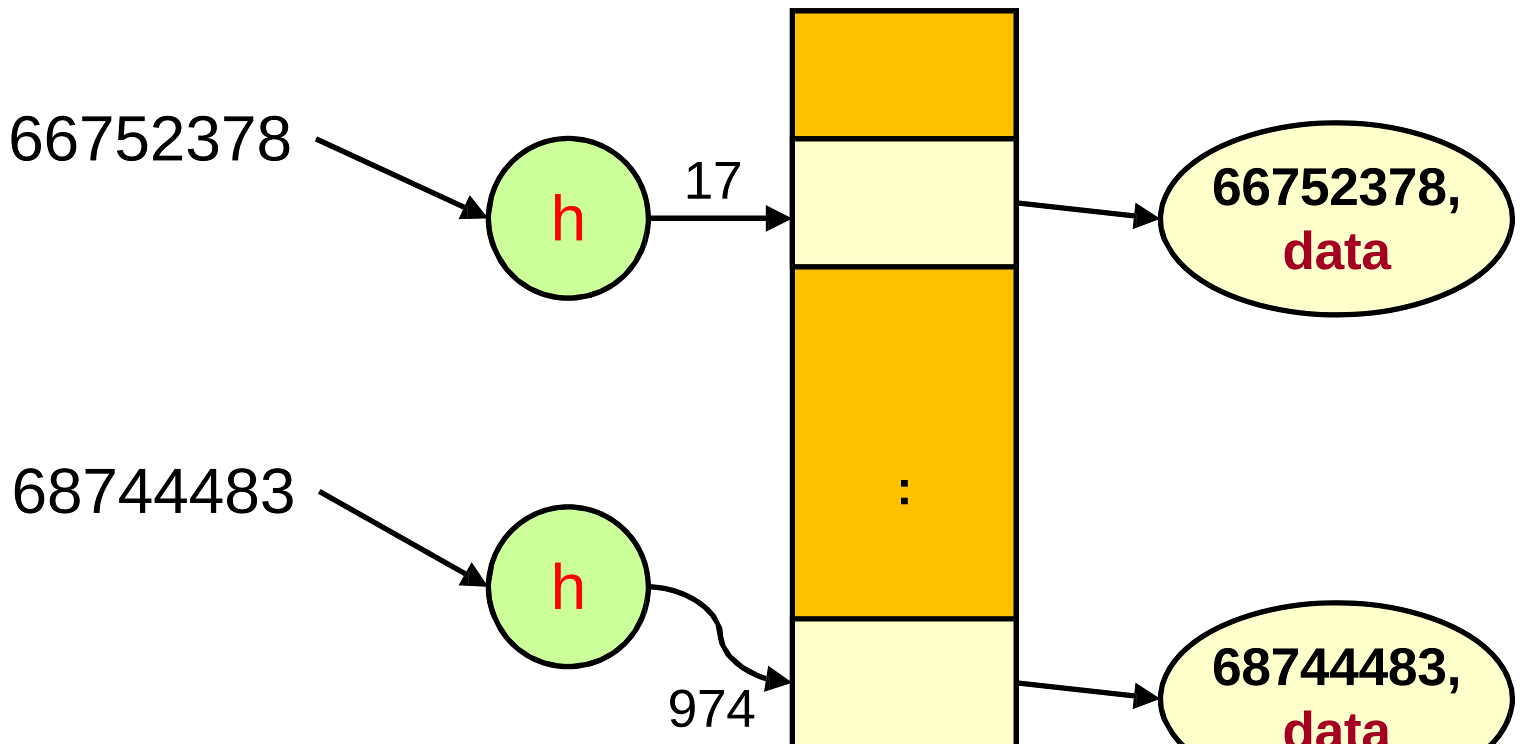
Hash Table is a generalization of direct addressing table, to remove the latter’s restrictions.

2 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)

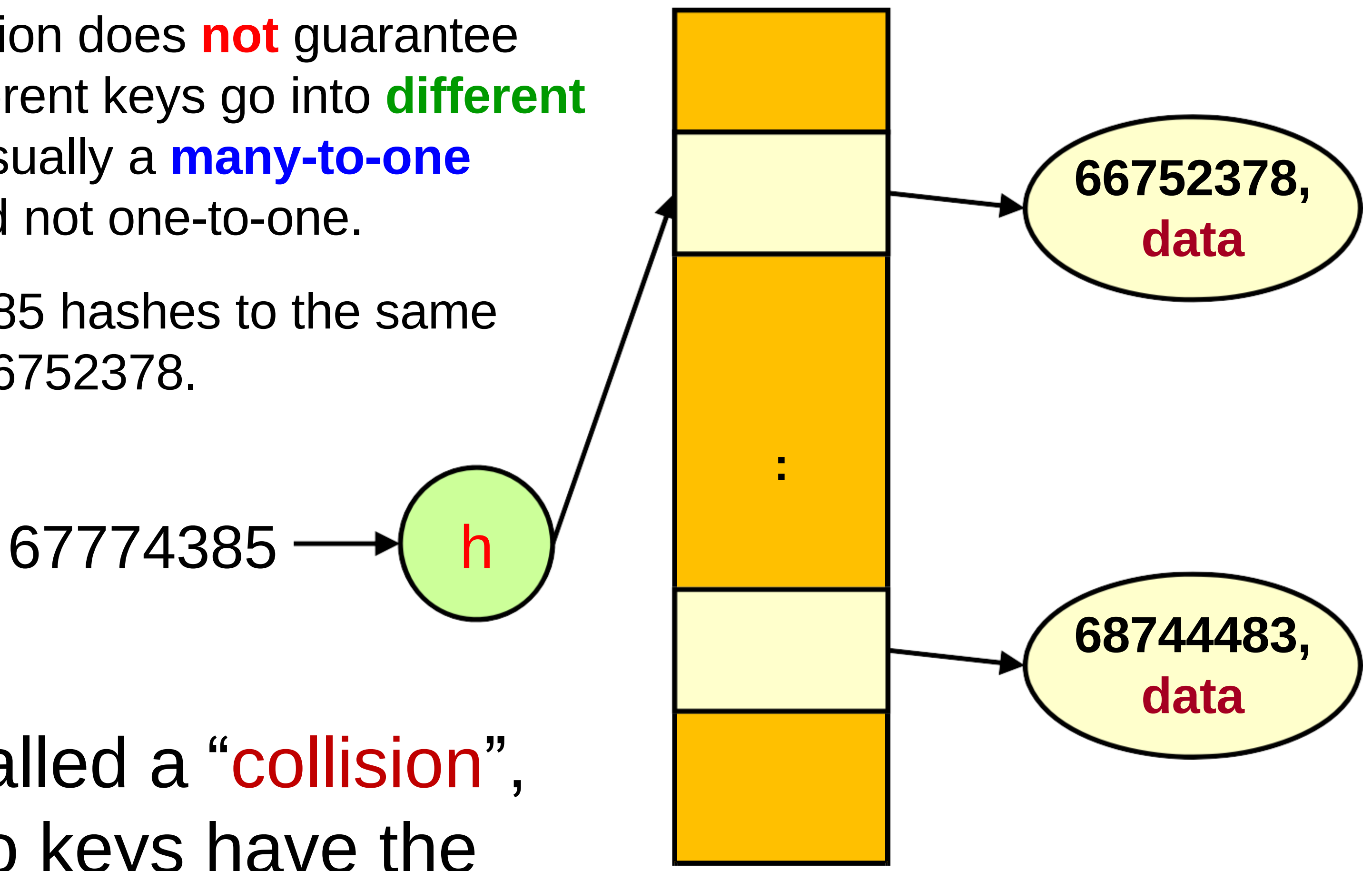
return a[h(key)]

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 same hash value.

2 Two Important Issues

- How to hash?
- How to resolve collisions?

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 Examples of Bad Hash Functions

- 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

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

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

Q: What are the meanings of “[” and “**)**” in [0, *X*)?

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 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?

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 has function value does not depend on positions of characters! – Bad

3 Hashing of strings (4/4)

- A better hash function for strings:

```
hash(s)
        sum = 0
        for each character c in s
                sum = sum*31 + 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.

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

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

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

= 1 – Q(n)

P(**23**) = 0.507

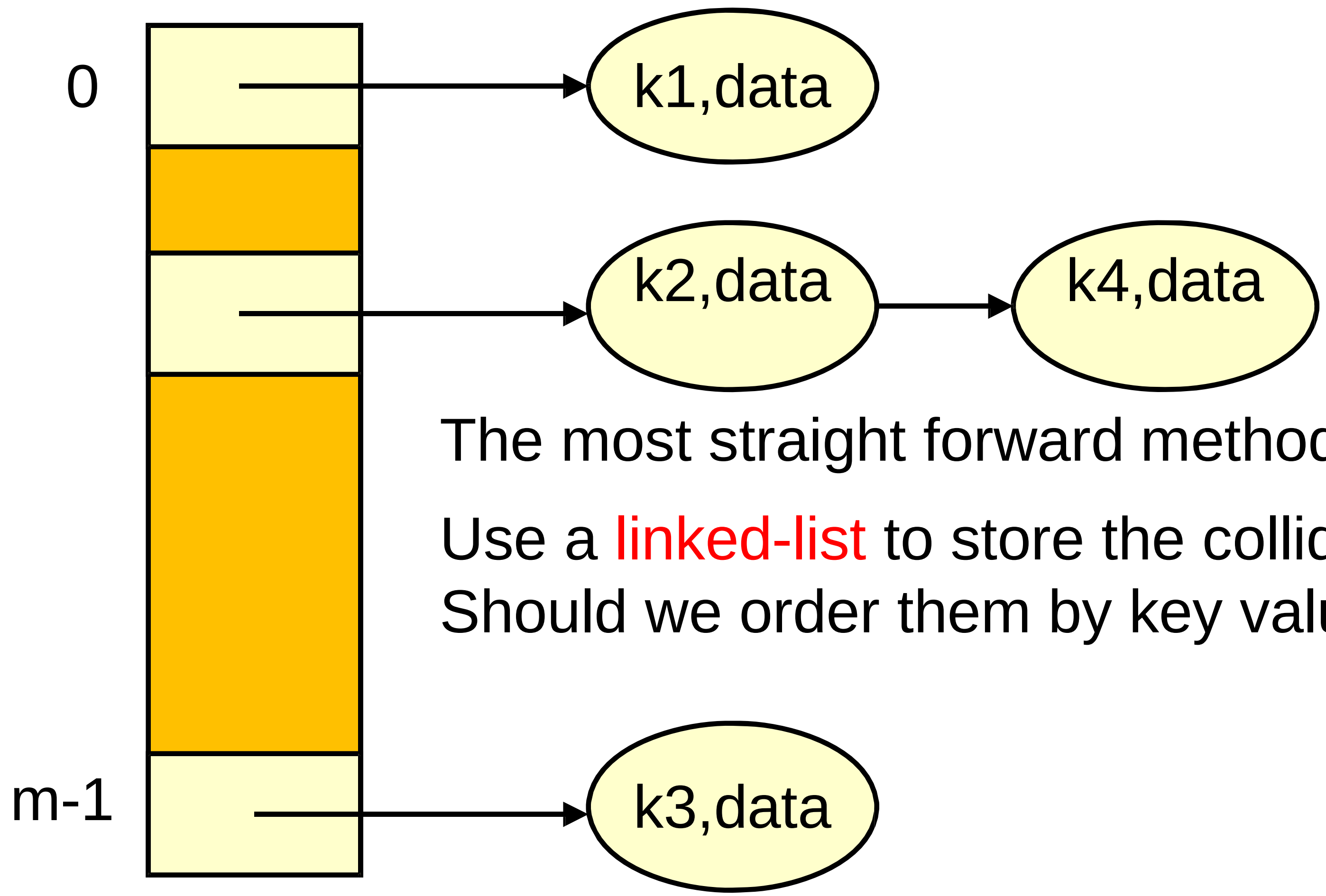
4 Probability of Collision (2/2)

- This means that if there are 23 people in a room, the probability that some people share a birthday is 50.7%.
- So, if we insert 23 keys in to a table with 365 slots, more than half of the time we 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 them by key values?

4.1 Hash table

insert (key, data)

insert data into the list a[h(key)]

delete (key)

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

find (key)

find key from the list a[h(key)]

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 Average Running Time

Find	$1 + \alpha/2$	for successful search
	$1 + \alpha$	for unsuccessful search (linked lists are not ordered)
Insert	$1 + \alpha$	for successful insertion
Delete	$1 + \alpha/2$	for successful deletion

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

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.

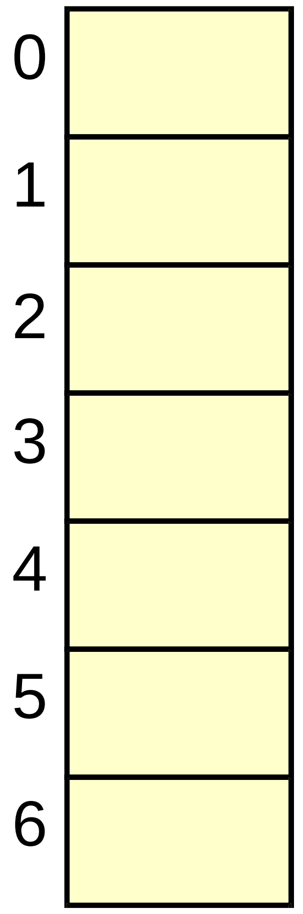
Collision resolution technique

4.2 Linear Probing

hash(k) = k mod 7

Here the table size m=7

Note: 7 is a prime number.

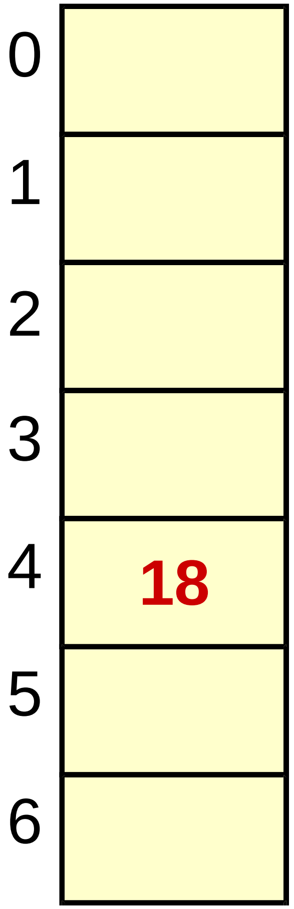


In linear probing, when we get a collision, we scan through the table looking for the next empty slot (wrapping around when we reach the last slot).

4.2 Linear Probing: Insert 18

hash(k) = k mod 7

hash(18) = 18 mod 7 = 4

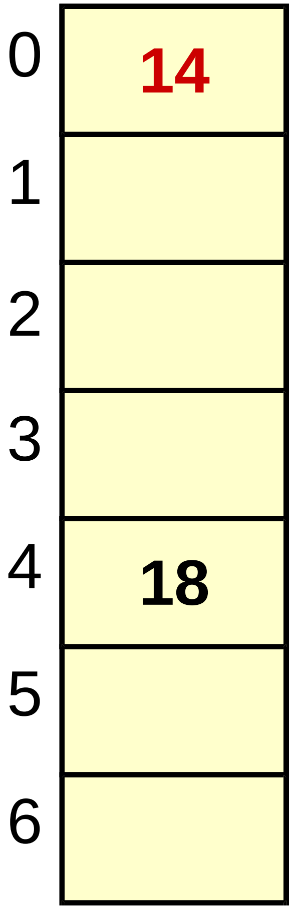


4.2 Linear Probing: Insert 14

hash(k) = k mod 7

hash(18) = 18 mod 7 = 4

hash(14) = 14 mod 7 = 0



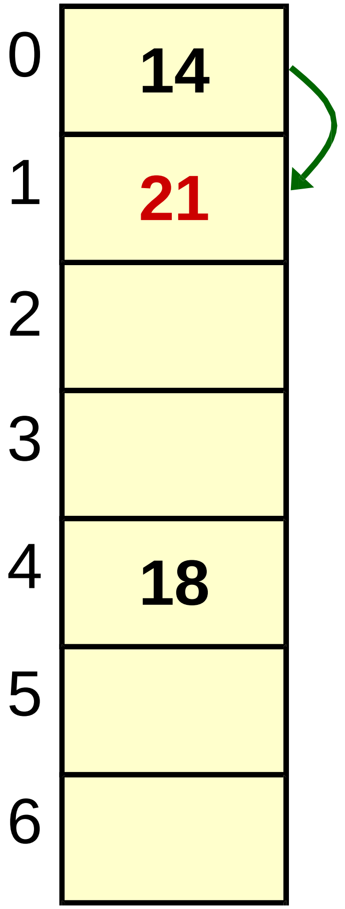
4.2 Linear Probing: Insert 21

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

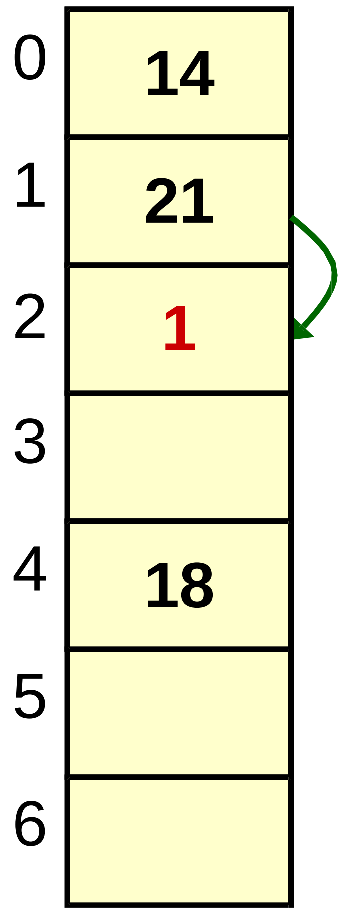
hash(k) = k mod 7

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

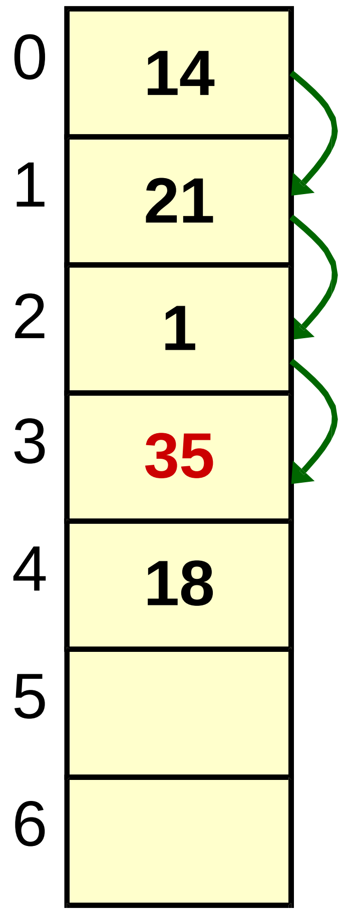
hash(18) = 18 mod 7 = 4

hash(14) = 14 mod 7 = 0

hash(21) = 21 mod 7 = 0

hash(1) = 1 mod 7 = 1

hash(35) = 35 mod 7 = 0

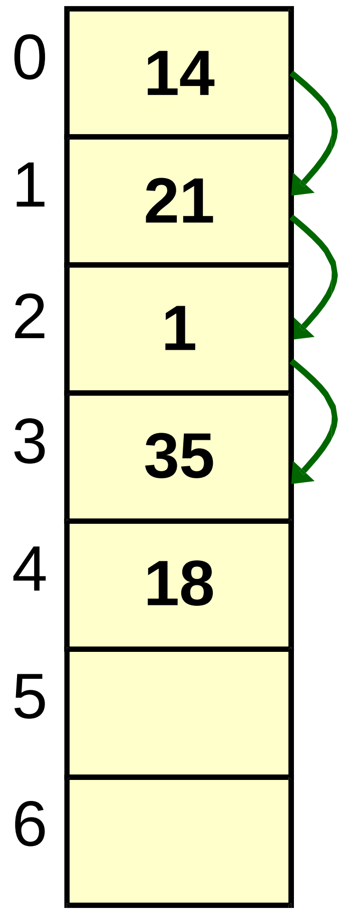


Collision, need to check next 3 slots.

4.2 Linear Probing: Find 35

hash(k) = k mod 7

hash(35) = 0

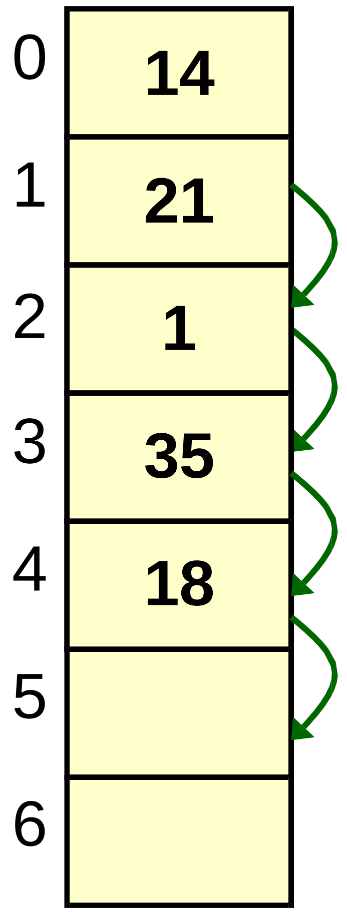


Found 35, after 4 probes.

4.2 Linear Probing: Find 8

hash(k) = k mod 7

hash(8) = 1

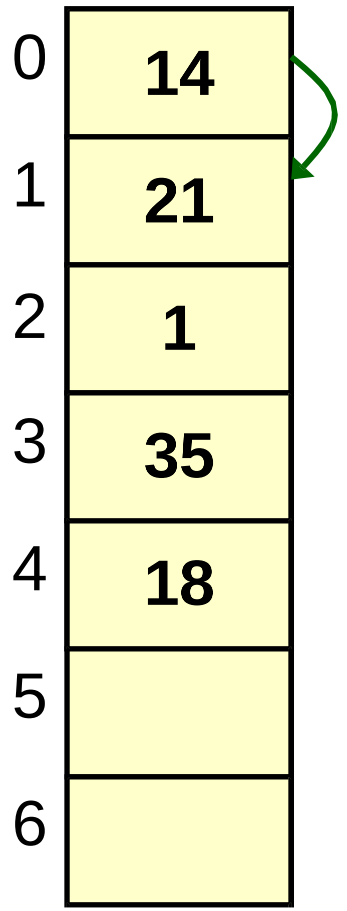


8 NOT found.
Need 5 probes!

4.2 Linear Probing: Delete 21

hash(k) = k mod 7

hash(21) = 0



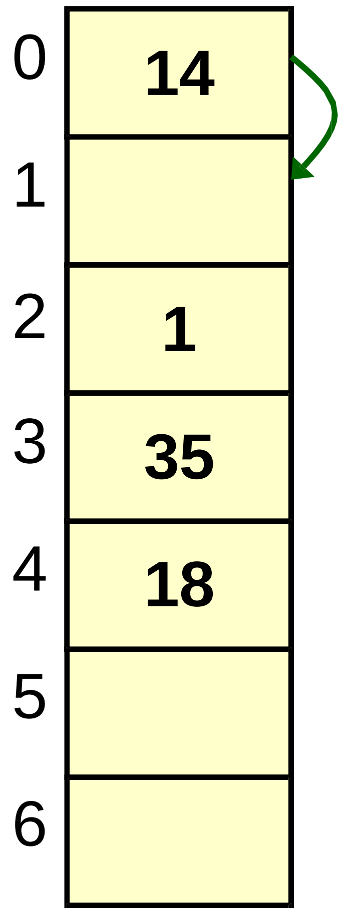
We cannot simply remove a value, because it can affect find()!

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!

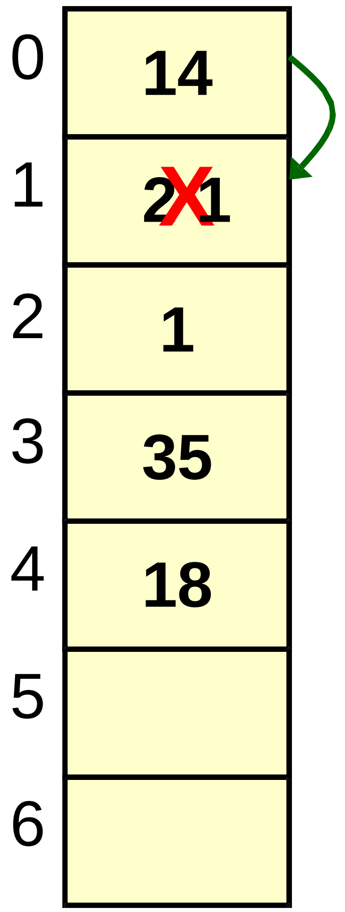
4.2 How to delete?

- **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

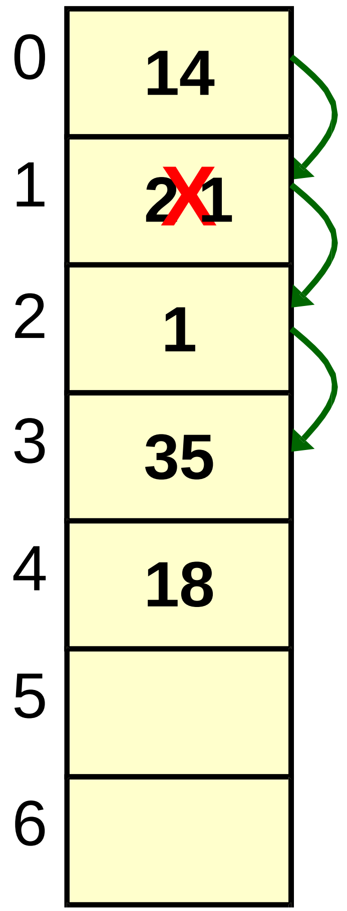


Slot 1 is occupied but now marked as deleted.

4.2 Linear Probing: Find 35

hash(k) = k mod 7

hash(35) = 0



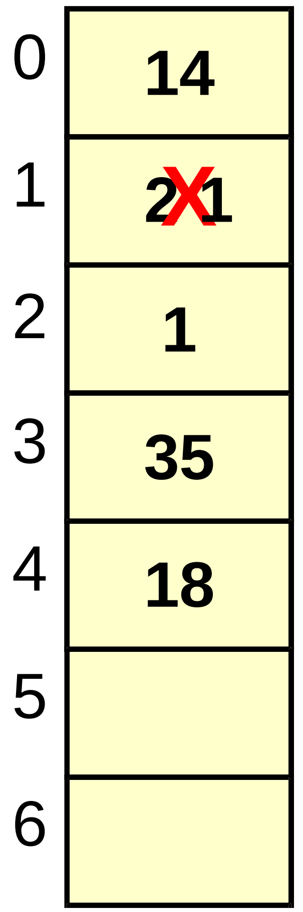
Found 35

Now we can find 35

4.2 Linear Probing: Insert 15 (1/2)

hash(k) = k mod 7

hash(15) = 1



Slot 1 is marked as deleted.

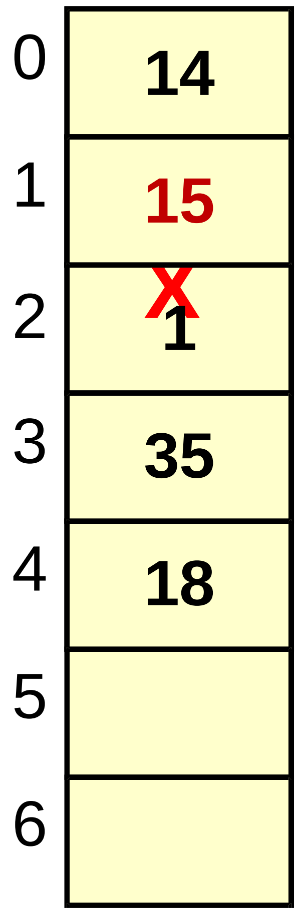
We continue to search for 15, and found that 15 is not in the hash table (total 5 probes).

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



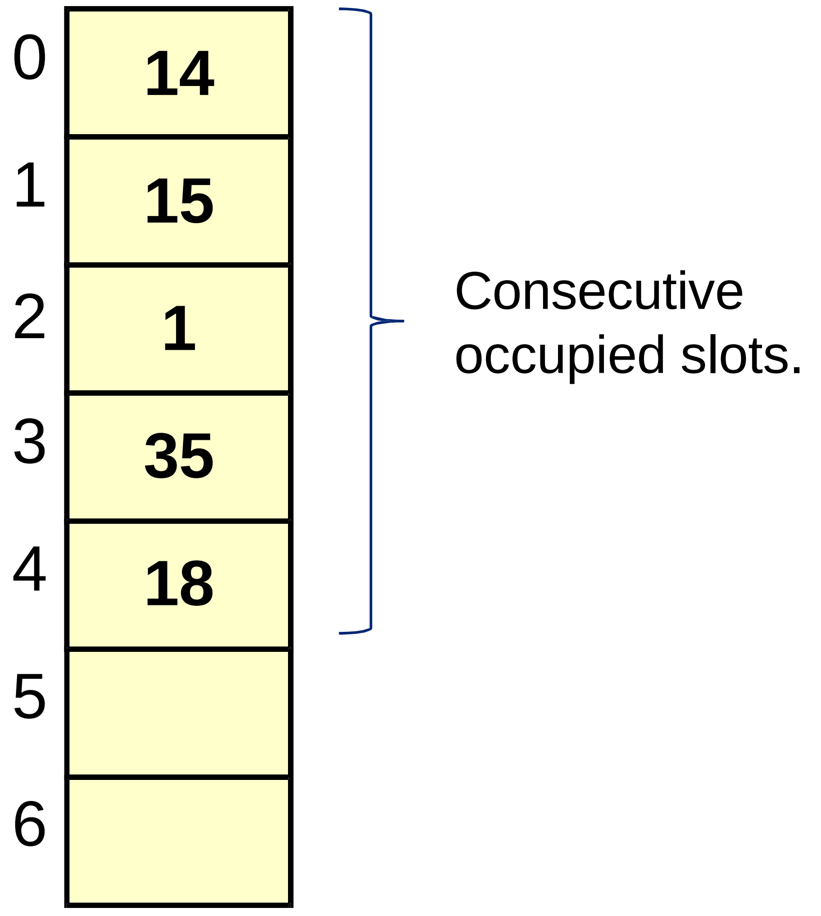
So, 15 is inserted into slot 1, which was marked as deleted.

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



Collision resolution technique

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

:

Collision resolution technique

4.2 Modified Linear Probing

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

We can modify the probe sequence as follows:

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

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.

Collision resolution technique

4.3 Quadratic Probing

For quadratic probing, the probe sequence is:

hash(key)

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

(hash(key) + **4**) % m

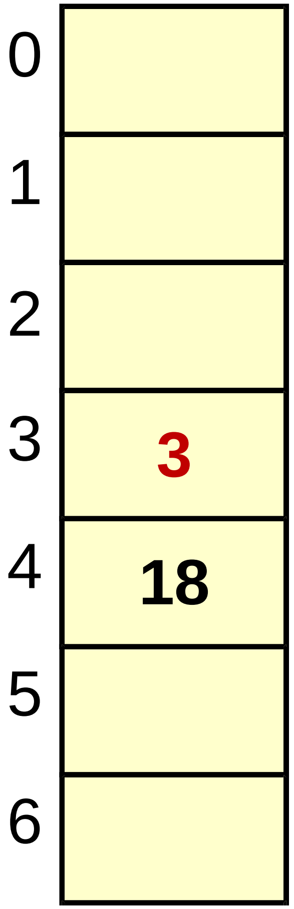
(hash(key) + **9**) % m

:

4.3 Quadratic Probing: Insert 3

hash(k) = k mod 7

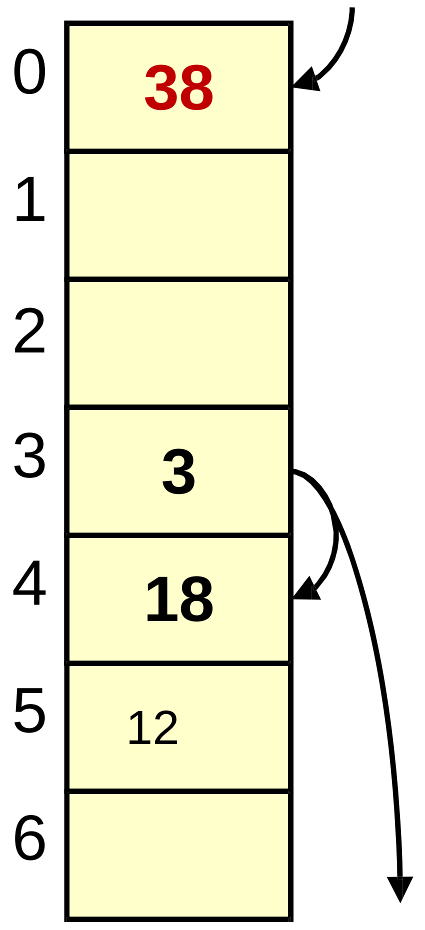
hash(3) = 3



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.

Collision resolution technique

4.4 Double Hashing

Use 2 hash functions:

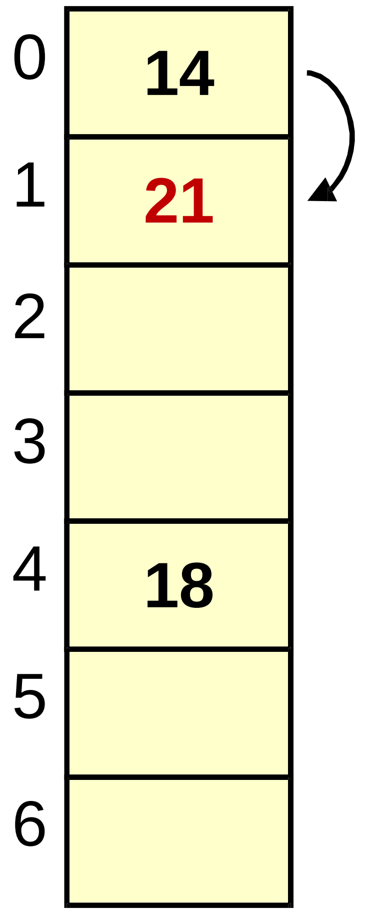
$$\begin{array}{l} hash(key) \\ (\ hash(key) + 1 * hash_2(key)\ )\ \%\ m \\ (\ hash(key) + 2 * hash_2(key)\ )\ \%\ m \\ (\ hash(key) + 3 * hash_2(key)\ )\ \%\ m \\ \quad\quad\quad\quad\vdots \end{array}$$

$hash_2$ 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

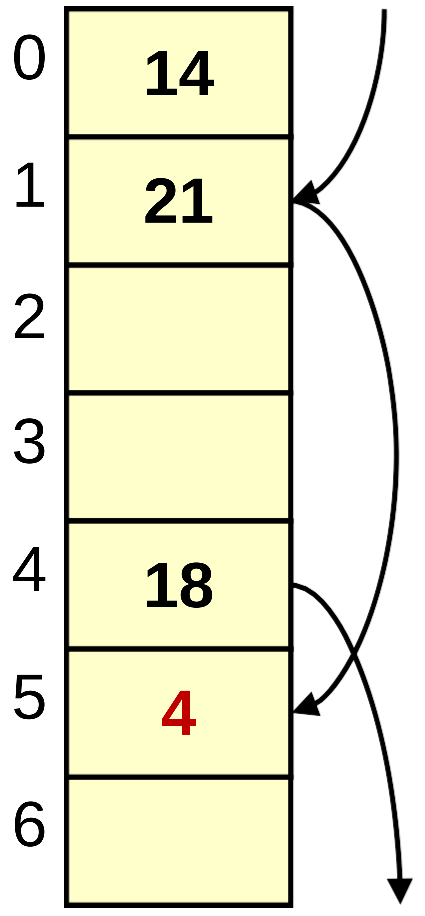
hash(21) = 0
hash_2(21) = 1



4.4 Double Hashing: Insert 4

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

hash(4) = 4
$hash_2$(4) = 4

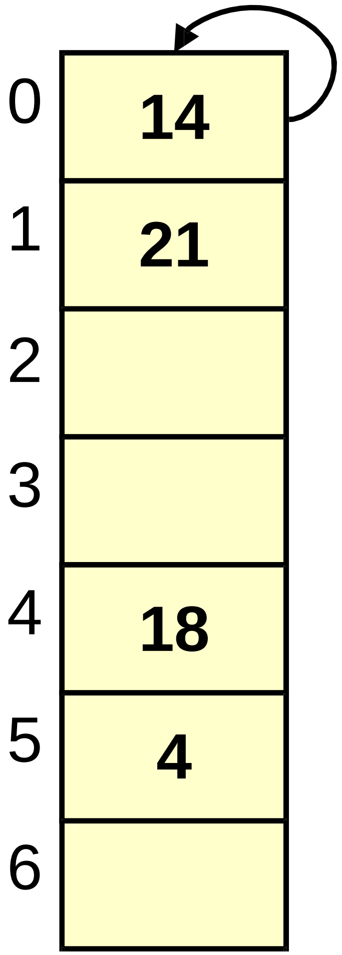


If we insert 4, the probe sequence is 4, 8, 12, …

4.4 Double Hashing: Insert 35

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

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



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

What is wrong?
Since $\text{hash}_2(35)=$**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

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 Hashtable Class

java.util

6 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.

java.util

6 Class Hashtable <K, V>

- **Constructor** summary
  - Hashtable()
    Constructs a new, empty hashtable with a default initial capacity (11) and load factor of 0.75.
  - Hashtable(int initialCapacity)
    Constructs a new, empty hashtable with the specified initial capacity and default load factor of 0.75.
  - Hashtable(int initialCapacity, float loadFactor)
    Constructs a new, empty hashtable with the specified initial capacity and the specified load factor.

- **Note:** A new version of hash table class is called HashMap. HashMap allows null values for key and value whereas Hashtable doesn’t allow. Hashtable is synchronized but HashMap is unsynchronized so it provides better performance if synchronization is not needed.

6 Class Hashtable <K, V>

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.
- ...

6 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
```

End of file