CS2040 Data Structures and Algorithms
Lecture Note #7

Map ADT – HashTable

For efficient look-up in a table

Objectives

1

- To know Map ADT and one efficient implementation – hashing/hashtable

2

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

3

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

Outline

1 Map ADT and Hashing

An ADT to map values to keys

1. Map ADT and Hashing

- Map ADT
  - An abstract data type that contains a collection of <key,value> pairs/mappings
  - It associates a key to a value in a one-to-one or many-to-one relation

    but each can still have multiple values by putting them in an array
  - There cannot be duplicate keys in the map
  - There are 3 basic operations
    - Retrieval – retrieve the value using the given key
    - Insertion – insert/replace a value using the given key
    - Deletion – delete the <key,value> pair using the given key

1. Map ADT and Hashing

- Hashing which is performed via a hash function is one concrete way for Map ADT to map a key to its value.
- 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 and caches.

1. Map ADT Operations

	Sorted List (Array impl. By sorting key)	**Balanced BST**	**HashTable**
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 (bBST) will be covered in later lectures.

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

2 Direct Addressing Table

A simplified version of hash table

2 SBS Transit Problem

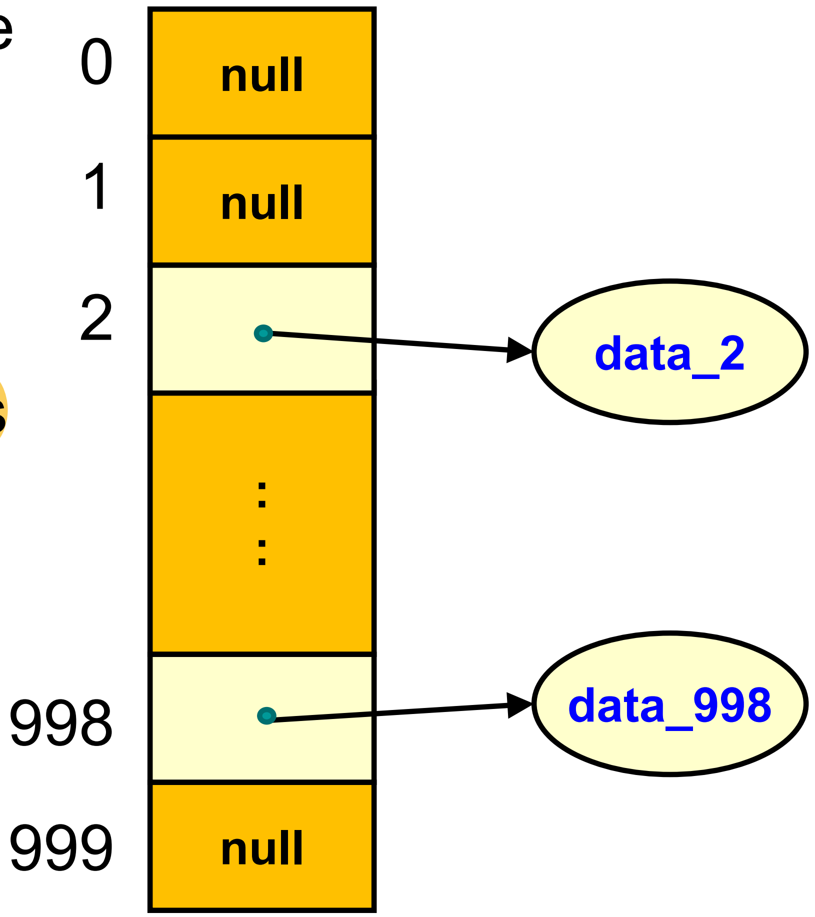
- Retrieval: find(*num*)
    - Find the bus route of bus service number *num*
- Insertion: insert(*num*,*data*)
    - Introduce a new bus service number *num*
- Deletion: delete(*num*)
    - Remove bus service number *num*

2 Direct Addressing Table

Assume that bus numbers are integers between 0 and 999, we can create an array of 1000 slots, each is a **reference** to an object which contains the details of the bus route (one-to-one mapping).

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

this is DAT because we use the key(bus number) as the indexto the array to get the data



2 Direct Addressing Table: Operations

```
insert (key, data)    O(1)
    a[key] = data        // where a[] is an array – the table

delete (key)    O(1)
    a[key] = null

find (key)    O(1)
    return a[key]
```

2 Direct Addressing Table: Restrictions

- Keys must be non-negative integer values
  - What happens for key values 151A and NR10? cannot directly use as index
- Range of keys must be small will run out of memory
- Keys must be dense, i.e. not many gaps in the key values.
- How to overcome these restrictions?

using general hash table

3 Hash Table

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

3 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.

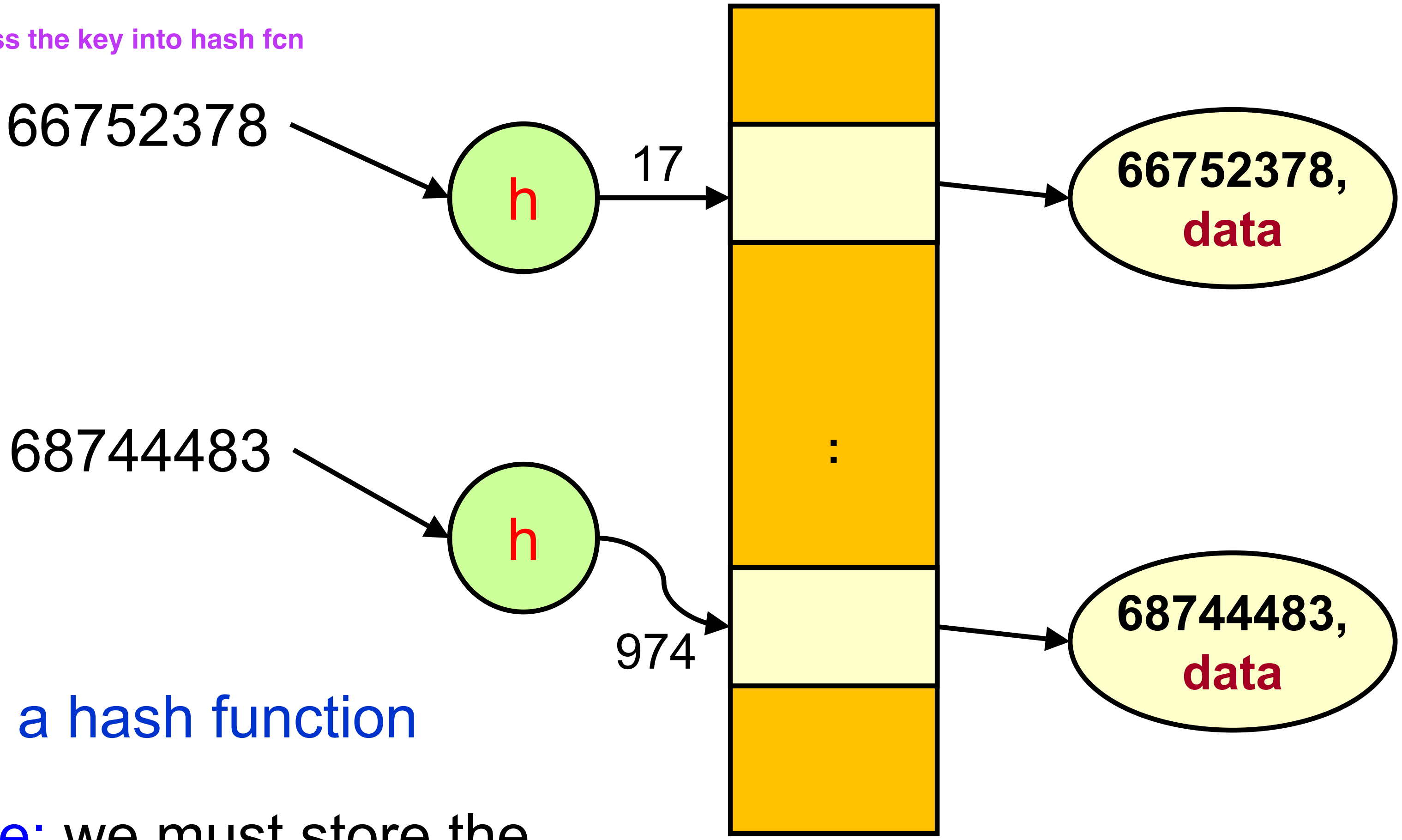
3 Ideas

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

HASHING

3 Hash Table

pass the key into hash fcn



h is a hash function

Note: we must store the key values. Why?

values of keys might be very large but the keys might not be dense enough(very spread out in the array) in DAT

3 Hash Table: Operations

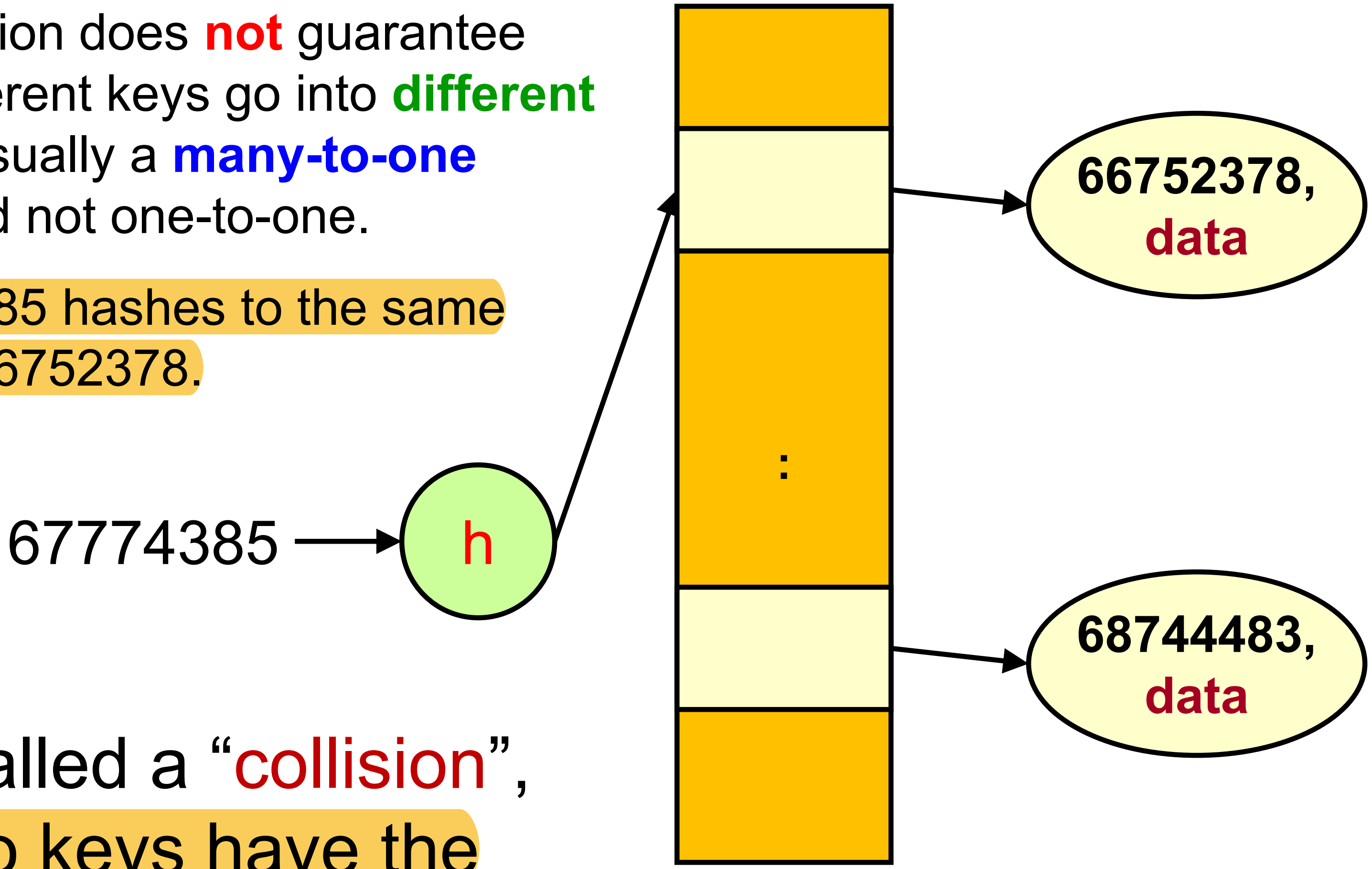
```
insert (key, data)
    a[h(key)] = data  // h is a hash function and a[] is an array
delete (key)
    a[h(key)] = null
find (key)
    return a[h(key)]
```

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

3 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.

3 Two Important Issues

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

4 Hash Functions

4 Criteria of Good Hash Functions

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

4 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?

4 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 beforehand.
- 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.

4 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, $0 \le k < X$

$\lfloor \ \rfloor$ is the *floor* function

4 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.

4 Multiplication method

1. Multiply by a constant real number $\mathbf{A}$ 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 $\mathbf{A}$ (recommended by Knuth).

4 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:
  - In general, pick a prime number for *m*

4 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
}
```

4 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

4 Hashing of strings: Examples (3/4)

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

4 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
}
```

31 because it is a prime no.

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

5 Collision Resolution

5 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!

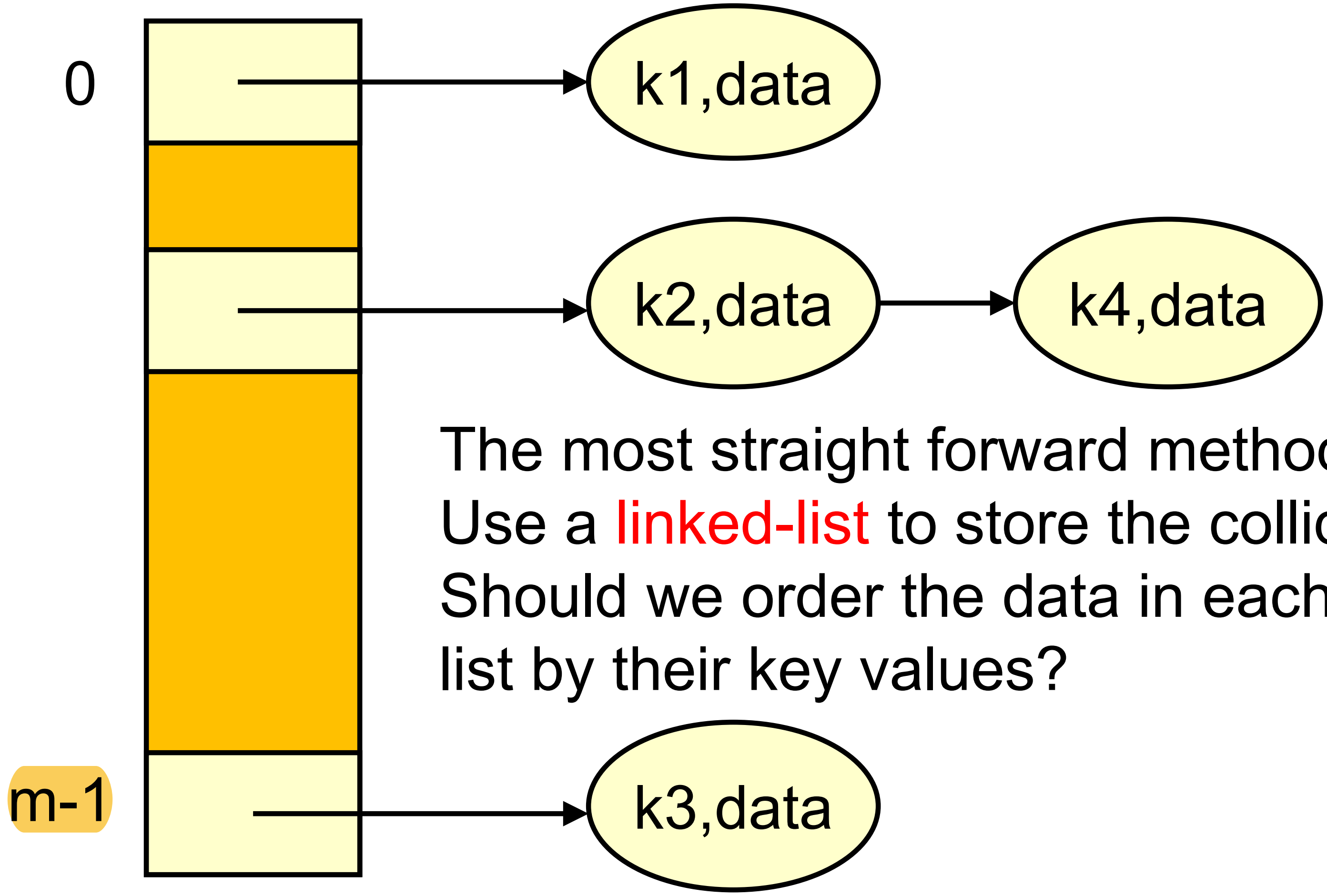
5 Probability of Collision (2/2)

- 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!

5 Collision Resolution Techniques

- Separate Chaining
- Open Addressing
  - Linear Probing + Modified Linear Probing
  - Quadratic Probing
  - Double Hashing

5.1 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?

5.1 Hash operations

insert (key, data)

add data to the back of the list at a[h(key)]

Takes worst case O(n) time, where *n* is length of the chain (**why?**) **need to check if the key exists first**

find (key)

Find key from the list at a[h(key)]

Takes worst case O(*n*) time

delete (key)

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

Takes worst case O(*n*) time **need to search for key**

need to check if the key exists first

5.1 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.
- Using a linked list for the chains also means separate chaining is not cache friendly

5.1 Reconstructing Hash Table

- To keep α bounded, we may need to reconstruct the whole table when the load factor exceeds the bound.

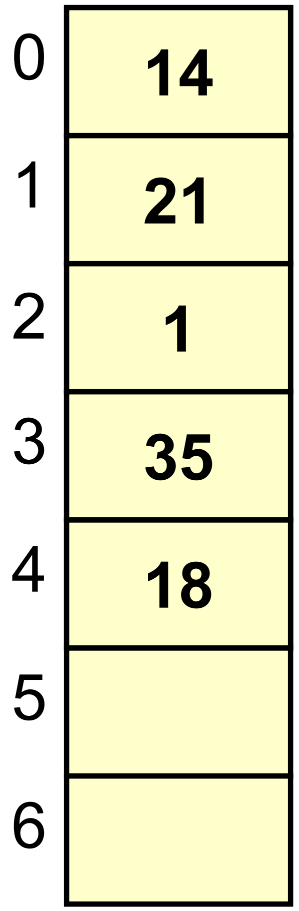
- Whenever the load factor exceeds the bound, we need to rehash all keys into a bigger table (increase *m* to reduce α), say a prime close to double the table size *m*.

5.2 Linear Probing - Find

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

Here the table size m=7

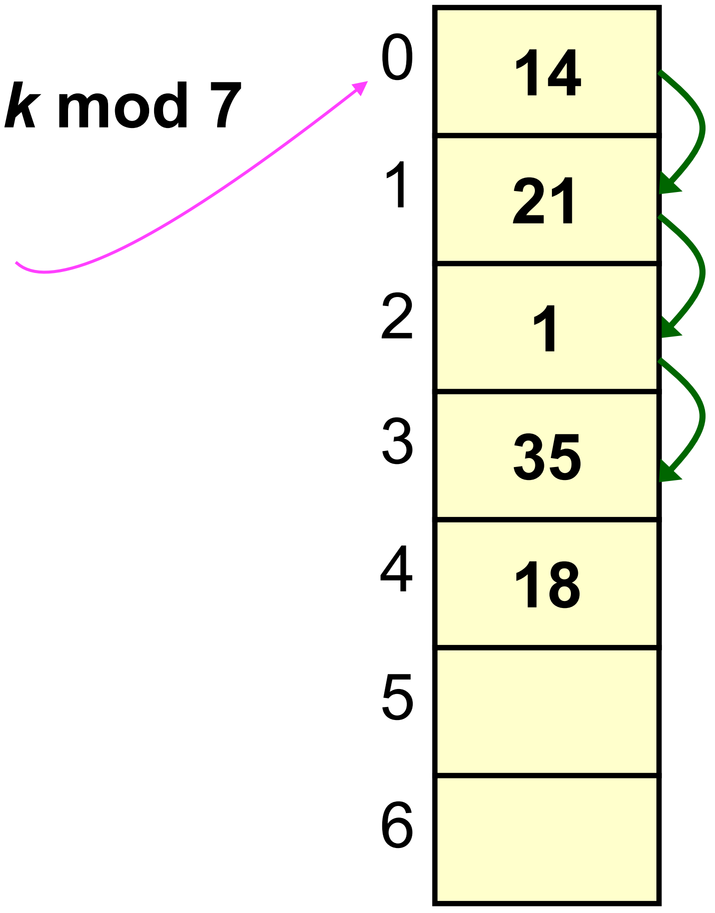
Note: 7 is a prime number.



5.2 Linear Probing: Find 35

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

hash(35) = 0

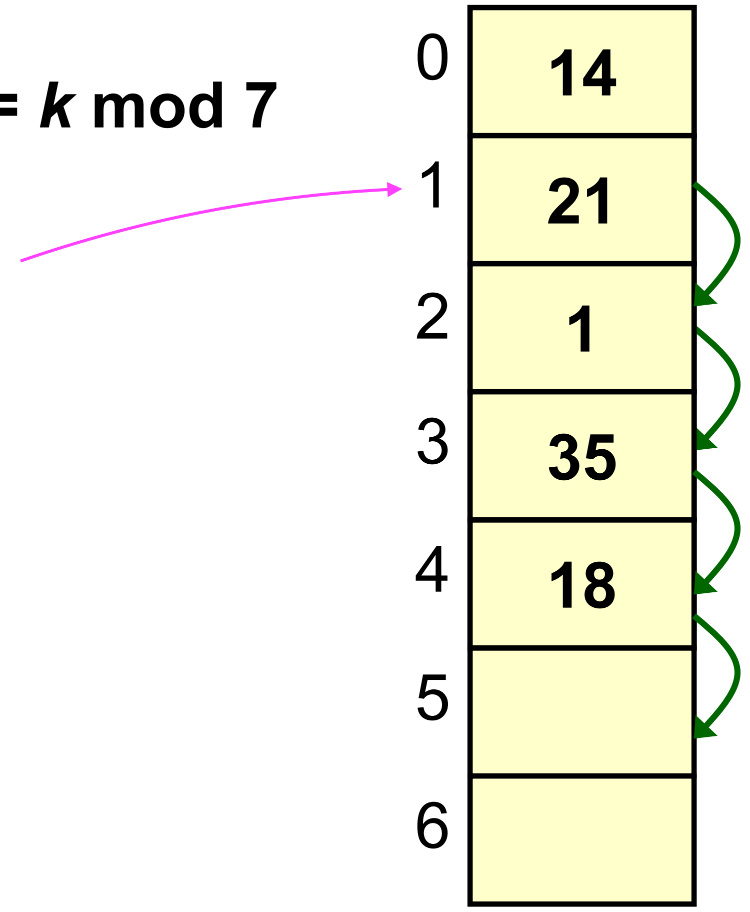


Found 35, after 4 probes.

5.2 Linear Probing: Find 8

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

hash(8) = 1

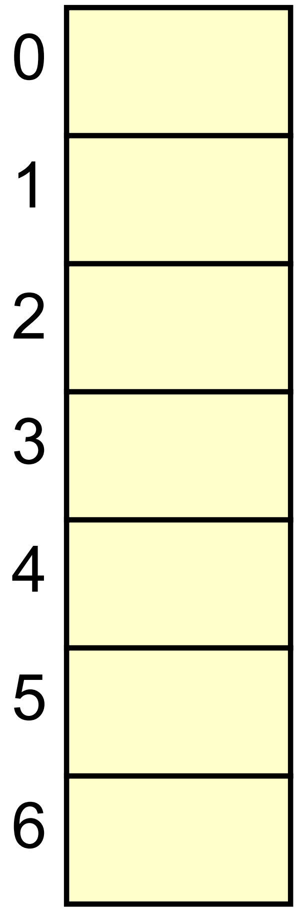


8 NOT found.
Need 5 probes!

if index 5 and 6 have values, wrap around probe at index 0 (or until we reach the index we started at)

5.2 Linear Probing - Insertion

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

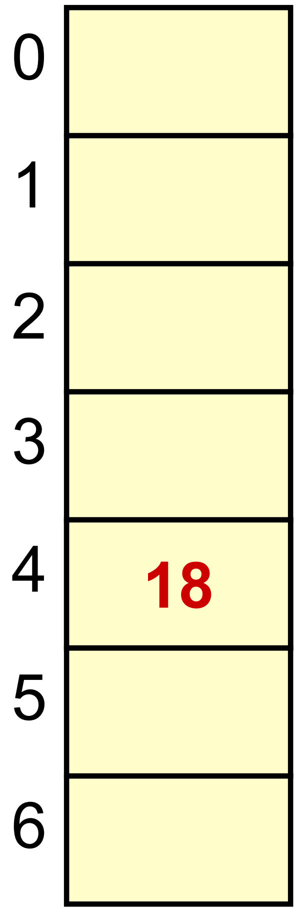


Start with an empty hash table

5.2 Linear Probing: Insert 18

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

hash(18) = 18 mod 7 = 4



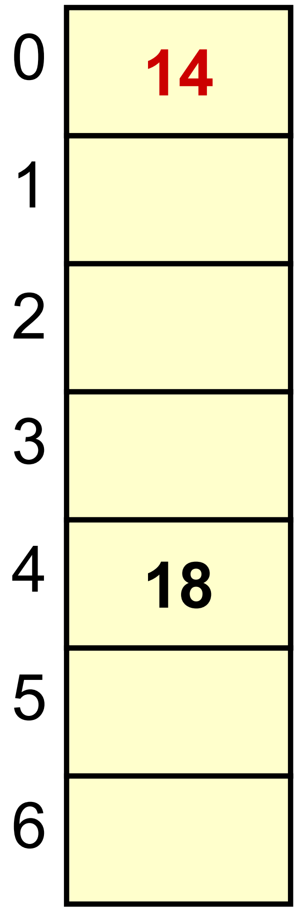
1. Find 18
2. If 18 not found insert at 1st empty slot
3. If 18 found update value associated with 18 if needed (there are no duplicate keys in the hash table !)

5.2 Linear Probing: Insert 14

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

hash(18) = 18 mod 7 = 4

hash(14) = 14 mod 7 = 0



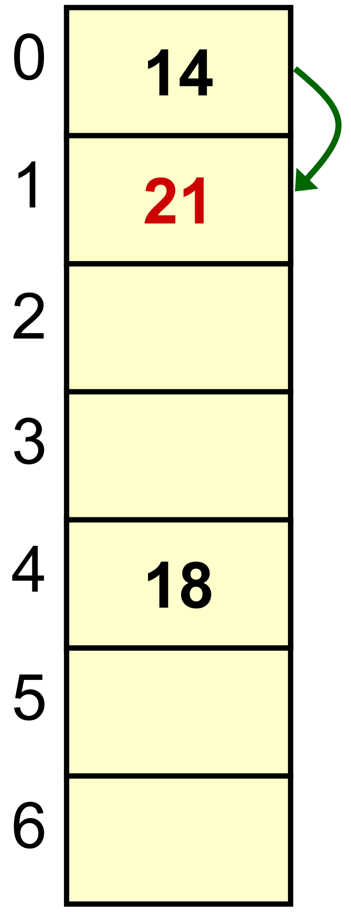
5.2 Linear Probing: Insert 21

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!

Continue until 21 is found or the next empty slot is reached

(wrapping around when we reach the last slot).

5.2 Linear Probing: Insert 1

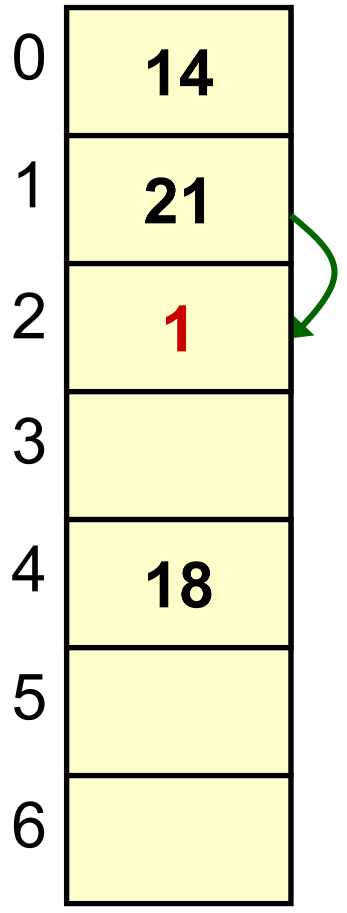
hash(*k*) = *k* mod 7

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~~ 1).

What should we do?

Look for next empty slot.

5.2 Linear Probing: Insert 35

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

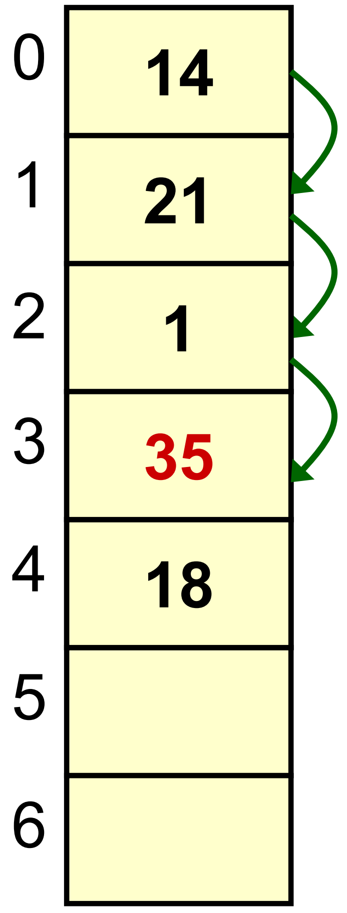
hash(18) = 18 mod 7 = 4

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.

5.2 Linear Probing - Deletion

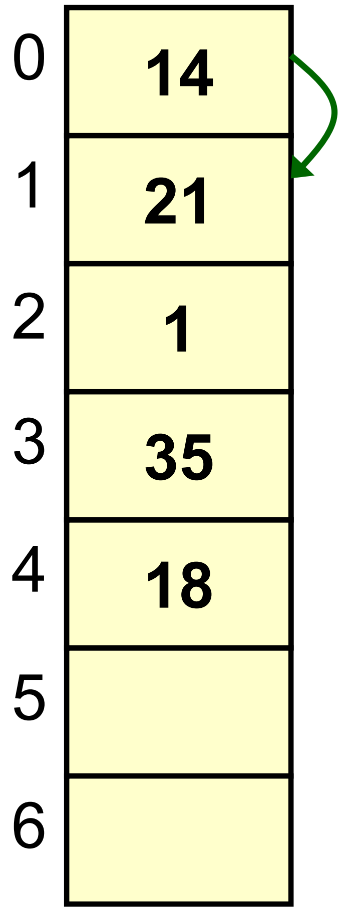
hash(k) = k mod 7

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

5.2 Linear Probing: Delete 21

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

hash(21) = 0



1. Find 21
2. If 21 not found do nothing
3. If 21 found ~~delete 21~~

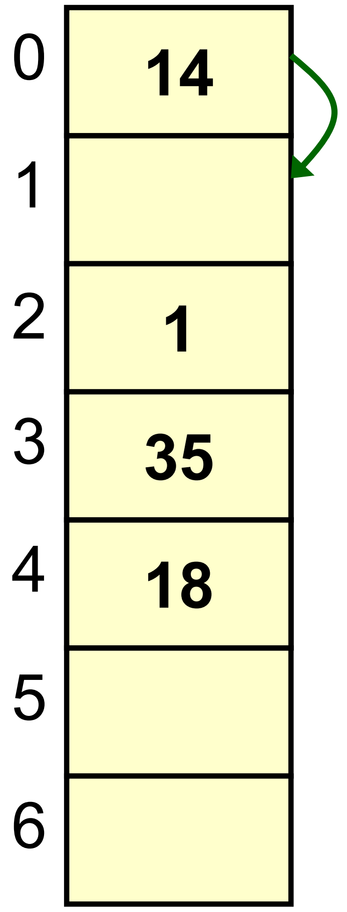
We cannot simply remove a value, because it can affect find()!

5.2 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!

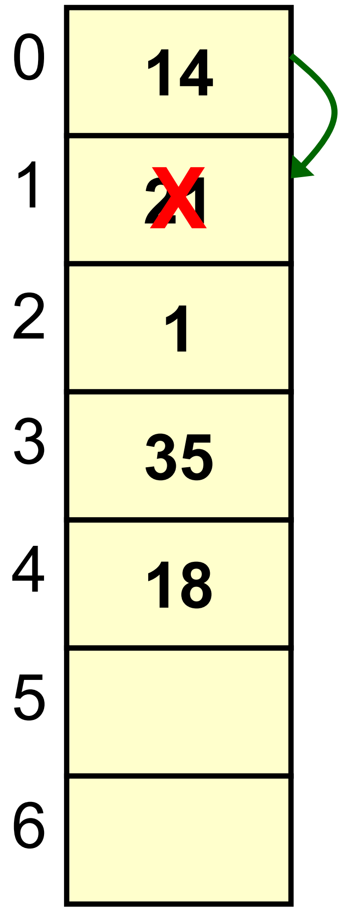
5.2 How to delete?

- 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.

5.2 Linear Probing: Delete 21

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

hash(21) = 0

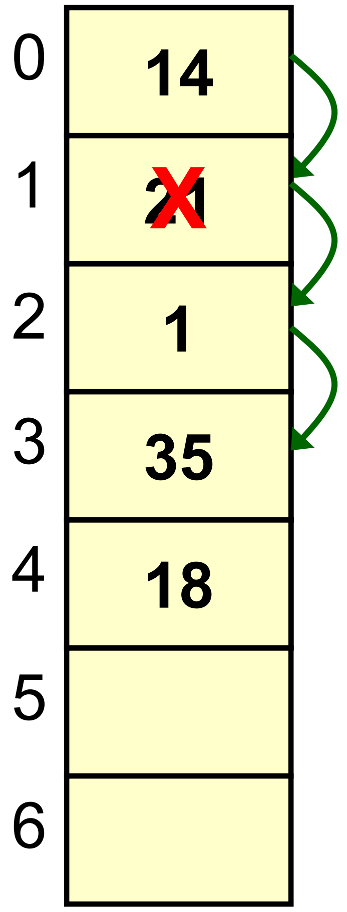


Slot 1 is occupied but now marked as deleted.

5.2 Linear Probing: Find 35

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

hash(35) = 0



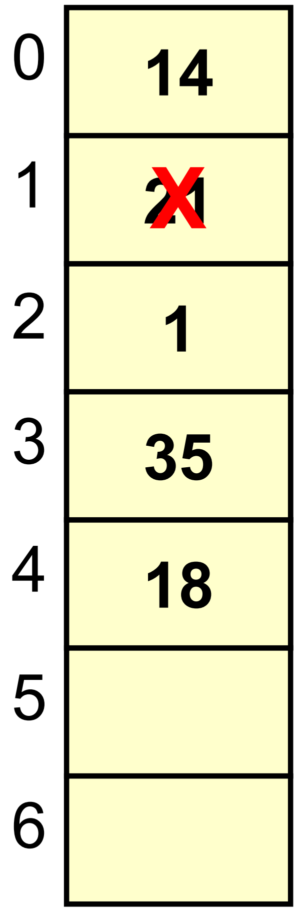
Found 35

Now we can find 35

5.2 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).

next time we want to find 15, faster to find -> we want the insertion to be as close as possible to the original hash value

5.2 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 the first available slot so that the find operation for this value will be the fastest.

5.2 Linear Probing

The probe sequence of this linear probing is:

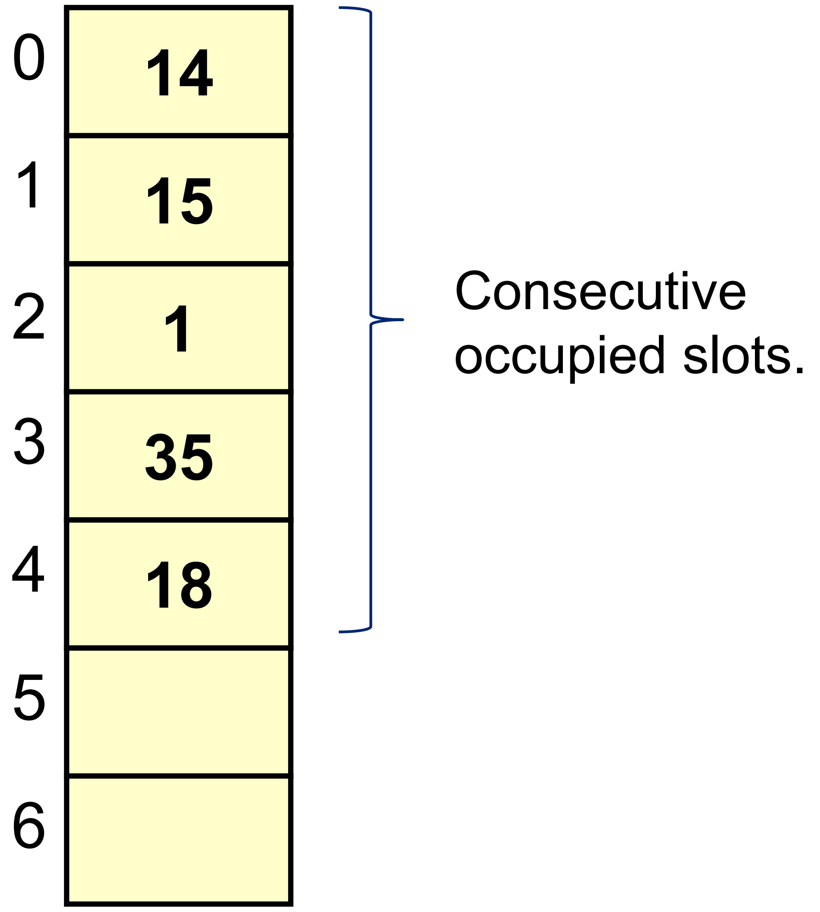
%m so that we can wrap
ard when we hit last index

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

5.2 Problem of Linear Probing

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

This is called Primary Clustering



5.2 Modified Linear Probing

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

We can modify the probe sequence as follows:

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

prevents overlapping of keys

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.

coprime: the largest integer that can divide both numbers is 1

5.3 Quadratic Probing

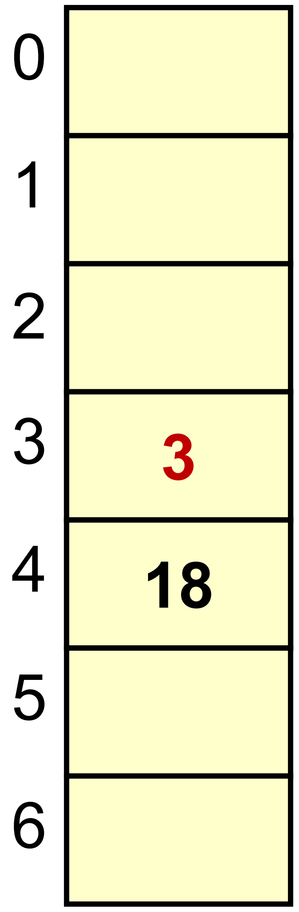
For quadratic probing, the probe sequence is:

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

5.3 Quadratic Probing: Insert 3

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

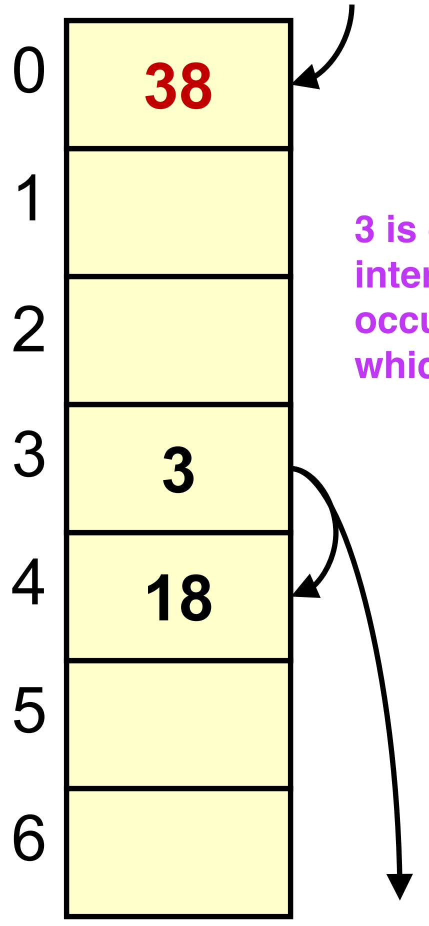
hash(3) = 3



5.3 Quadratic Probing: Insert 38

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

hash(38) = 3



3 is occupied so then we probe in odd intervals i.e probe at 4(+1) which is occupied by 18, so then probe again(+3) which brings us to 0

5.3 Theorem of Quadratic Probing

once it is >= 0.5, there is no guarantee Q.P. can find an empty slot even if there is one

- 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. if exceeds 0.5 we should resize the table so that we keep using less than half of it to successfully find a slot
- Q: How can we be sure that quadratic probing always terminates? when we come back to starting position
- Insert 12 into the previous example, followed by 10. See what happen?

5.3 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.

5.4 Double Hashing

Use 2 hash functions:

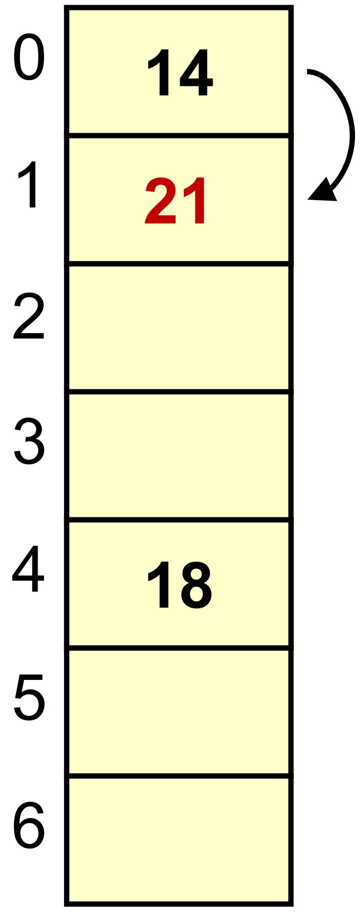
$$\text{hash(key)}$$
$$(\ \text{hash(key)} + 1 * \text{hash}_2\text{(key)}\ )\ \%\ m$$
$$(\ \text{hash(key)} + 2 * \text{hash}_2\text{(key)}\ )\ \%\ m$$
$$(\ \text{hash(key)} + 3 * \text{hash}_2\text{(key)}\ )\ \%\ m$$
$$\vdots$$

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

5.4 Double Hashing: Insert 21

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

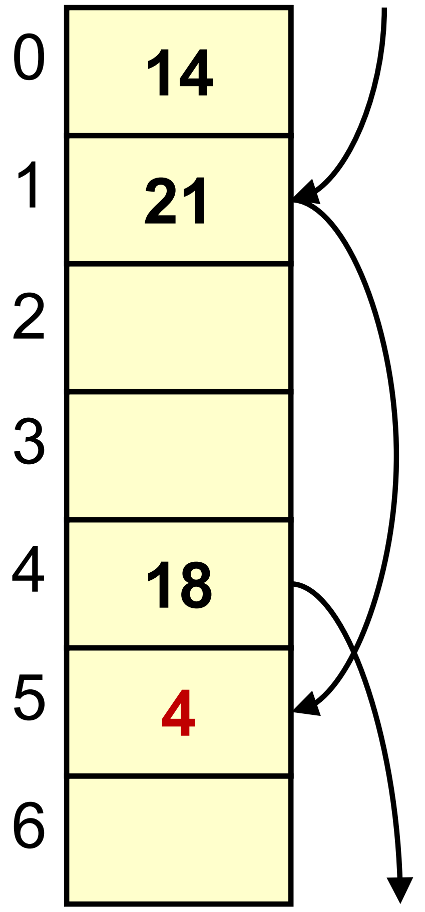
hash(21) = 0
$\text{hash}_2(21) = 1$



5.4 Double Hashing: Insert 4

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

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



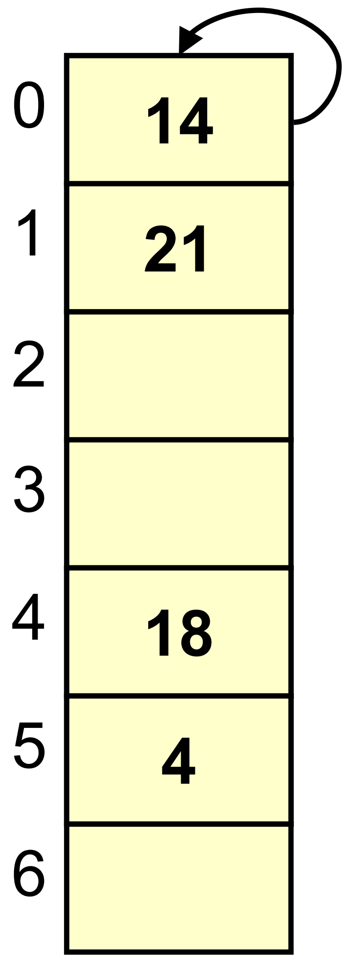
If we insert 4, the probe sequence is 4, 8, 12, …

5.4 Double Hashing: Insert 35

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

hash(35) = 0
hash$_2$(35) = 0

secondary hash value should never evaluate to 0



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

What is wrong?
Since hash$_2$(35)=**0**.
Not acceptable!

5.4 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 integer > 1, then it is the same as modified linear probing.

5.5 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

5.6 Worst case performance of hashing

- For separate chaining
  - O(n) time for find/insert/delete
  - meaning the single index holds all the items
  - Such a case is when all items hash to one index so you have a linked list of size n

- For open addressing
  - O(n) time for find/insert/delete
  - Such a case is when you have to probe every slot in the table to determine you cannot insert or what is to be found/deleted is not in the table

5.6 Average case performance of hashing: Using Separate Chaining

- α (load factor) is the average size of the linked list for each slot
- Thus if α is bounded, find/insert/delete will take O(1) time on average

5.6 Average case performance of hashing: Using linear probing

uniform

- Average case = average number of probes

- Assuming each probe location is generated randomly and independently *(such collision resolution cannot be used in practice !)* so that clustering does not happen

- For each probe

load factor in open addressing is how full our table is, i.e
if LF = 1/3, it means the table is 1/3 full and 2/3 empty

$$\text{Probability of finding empty slot} = 1 - \propto$$

$$\text{Probablity of finding non-empty slot} = \propto$$

5.6 Average case performance of hashing: Using linear probing

- For unsuccessful find and successful insert (need to hit an empty slot)

$$\text{expected number of probes} = 1 + \frac{1}{(1-\propto)^2}$$

- For successful find and successful delete

$$\text{expected number of probes} = 1 + \frac{1}{1-\propto}$$

- Again if α is bounded, find/insert/delete will take O(1) time on average (as α grows, performance will degrade severely)

6 Set ADT

- A set as you have learned in high school is simply a unordered collection of items with no duplicates (with duplicates it’s a multi-set)
  - E.g {1,2,3} is a set of 3 integers and this set is the same set as {3,1,2}, since order does not matter
- Some simple Set operations include the following
  - find(x) – retrieve x from the set if it exist in the set
  - insert(x) – insert x into the set if it doesn’t already exist in set
  - remove(x) – remove x from the set if it exist in the set
  - union(s) – return union of this set with another set s
  - intersect(s) – return intersection of this set with another set s

6 Using Hashtable for simple Set

- HashTable can easily and efficiently implement a Set ADT If we do not need complex operations like set intersection and union

Set Operations	HashTable implementation	Time complexity
find(x)	find(x)	Average O(1)
Insert(x)	insert(x,x) – make <key,value> pair the same	Average O(1)
remove(x)	remove(x)	Average O(1)

7 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.

8 Java HashMap Class

8 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.

8 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.

8 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.
- …

8 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