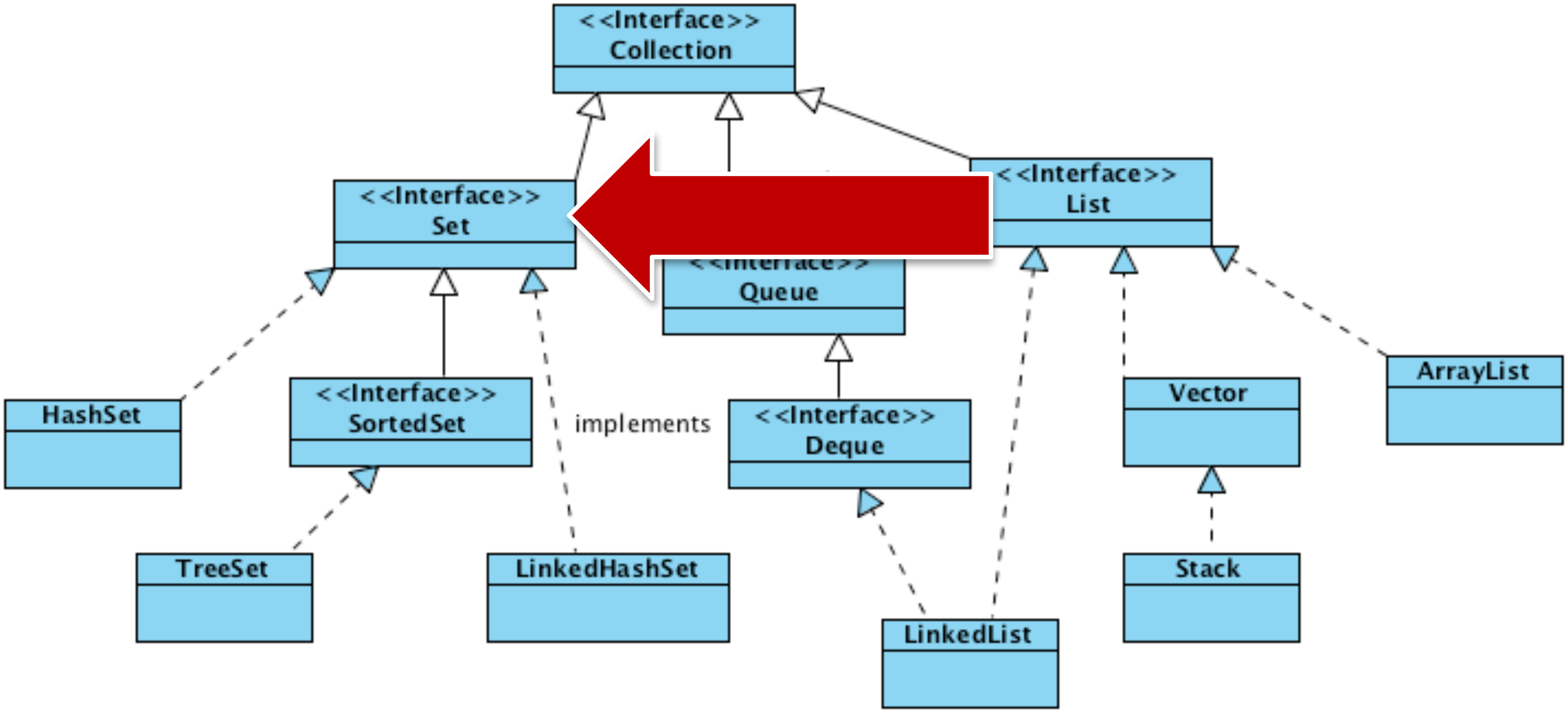
Sets

Collection that cannot contain duplicate elements.

Collection Family Tree



Set interface

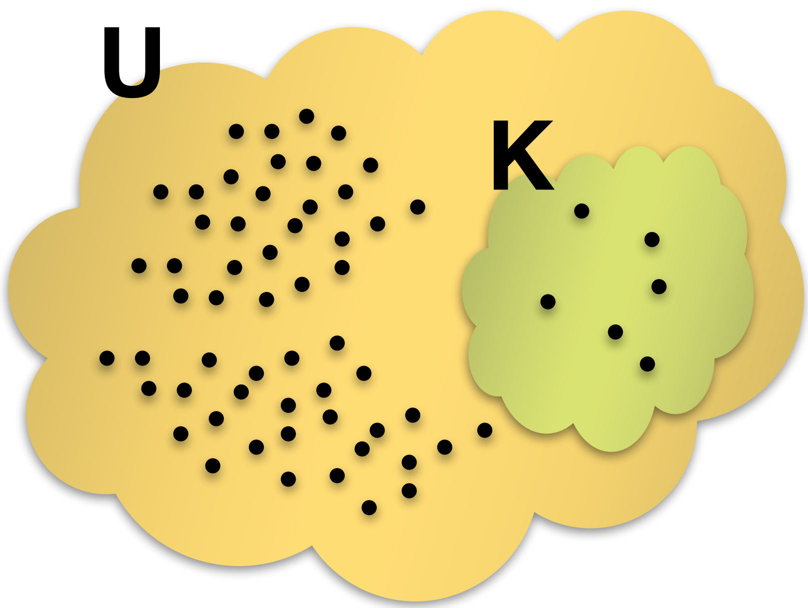
- Add/remove elements
  - boolean **add**(element)
  - boolean **remove**(object)
- Search
  - boolean **contains**(object)
- No duplicates
- No positional Access!

Hash Tables

A data structure implementing an associative array

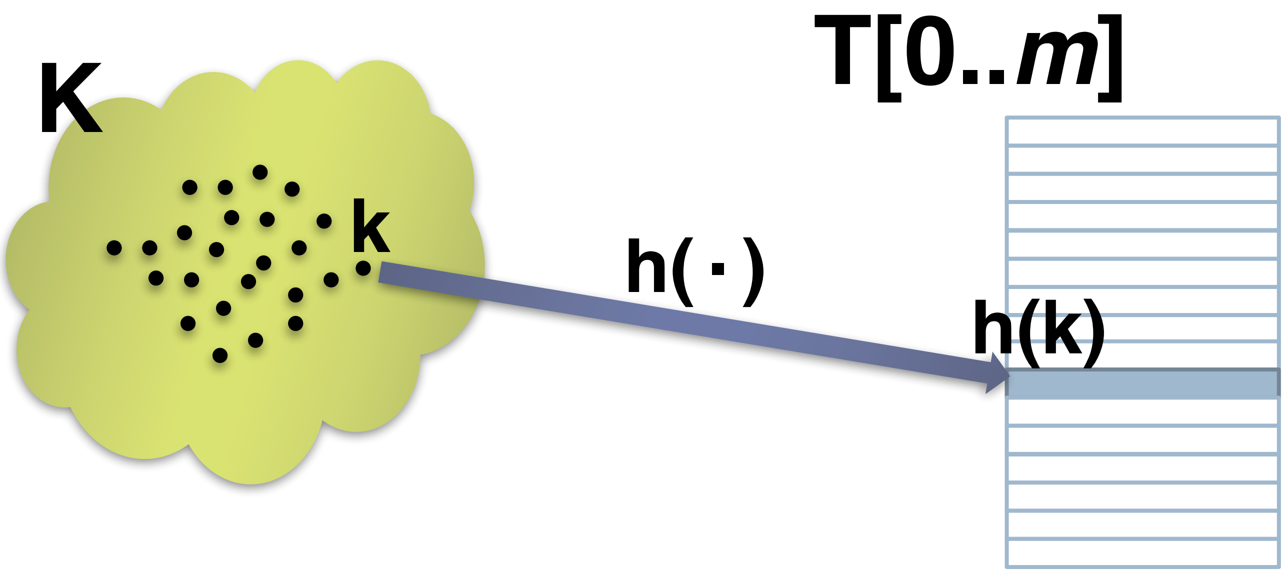
Notation

- A set stores *keys*
- U – Universe of all possible keys
- K – Set of keys actually stored



Hash Table

- Devise a function to transform each *key* into an index
- Use an array



Hash Function

- Is a function that maps data of arbitrary size to data of fixed size
- Mapping from **U** to the slots of a hash table T[0…m–1]

$$h : \mathbf{U} \rightarrow \{0,1,\ldots, m{-}1\}$$

- h(k) is the "hash value" of key k
- Main application:
  - Hash table
  - Cryptographic hash function
    - Authentication
    - Ensure file integrity (to avoid tampering)
    - Calculate digest for digital signature
    - *Used by Git too.*

Hash Function

- Main properties:
  - Hash table
    - Deterministic: same key, same hash value
    - Uniform: “Any key should be equally likely to hash into any of the *m* slots, independent of where any other key hashes to”
    - Defined range
  - Cryptographic hash function
    - Collision resistance (large hash value) e.g. SHA-1 160 bit
    - Non invertible: it is not possible to reconstruct *k* from *h(k)*

Hash Function

- Compression
  - $h_N : \mathbf{U} \rightarrow \mathbf{N}^+$

    $h(k) = h_N(k) \bmod m$
- Expansion
  - $h_R : \mathbf{U} \rightarrow [0, 1[\ \in \mathbf{R}$

    $h(k) = \lfloor h_R(k) \cdot m \rfloor$

Hash Function - Complexity

- Usually, h(k) = O(*length*(k))
  - *length*(k) « N → h(k) = O(1)

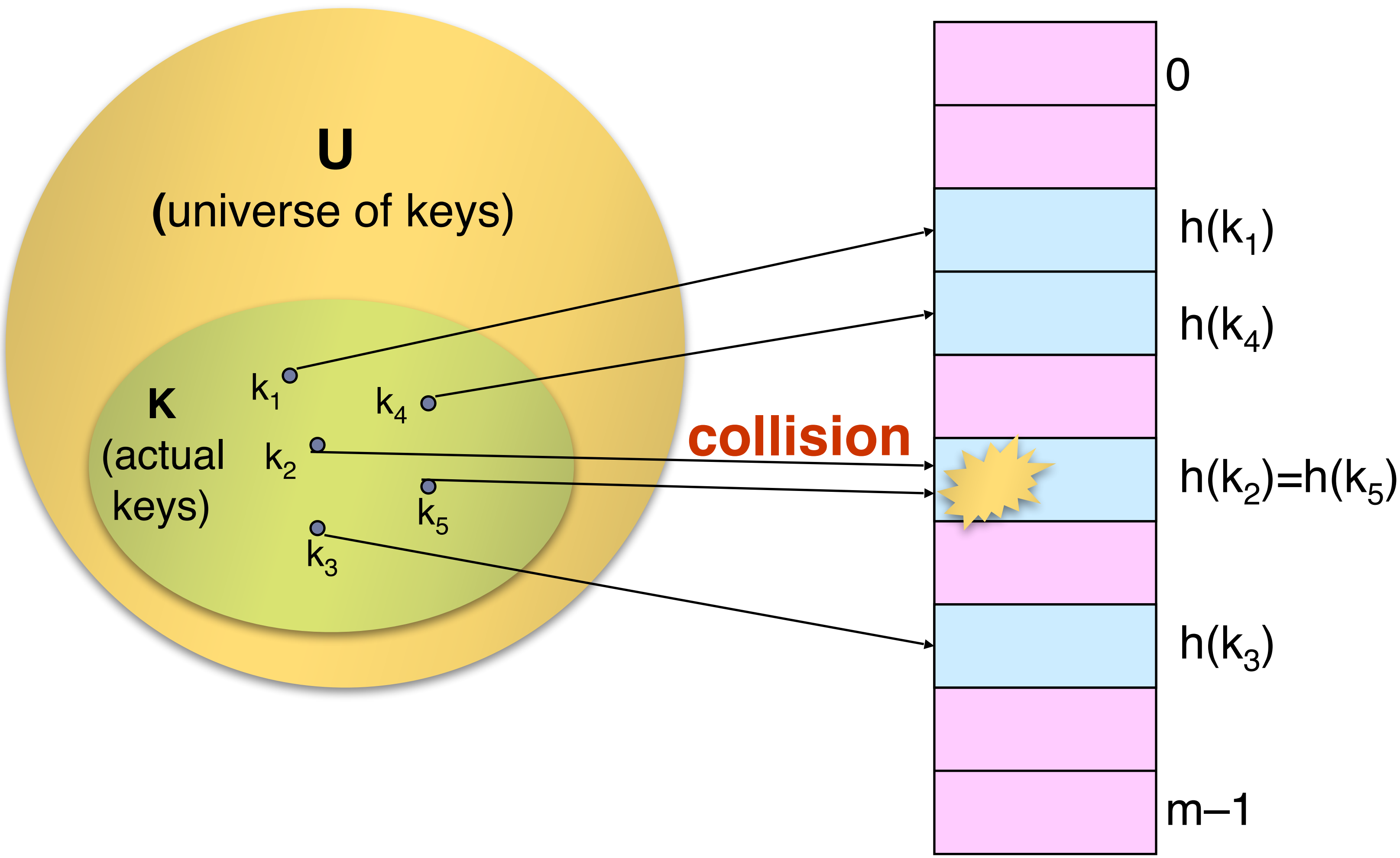
A simple hash function

- h : A ⊆ N+ → [0, m-1]
- Split the key into its "component", then sum their integer representation
- $h_N(k) = h_N(x_0 x_1 x_2 \dots x_n) = \sum_{i=0}^{n} x_i$
- h(k) = h_N(k) % m

A simple hash (problems)

- Problems
  - h_N("NOTE") = 78+79+84+69 = 310
  - h_N("TONE") = 310
  - h_N("STOP") = 83+84+79+80 = 326
  - h_N("SPOT") = 326
- Problems (m = 173)
  - h(74,778) = 42
  - h(16,823) = 42
  - h(1,611,883) = 42

Collisions

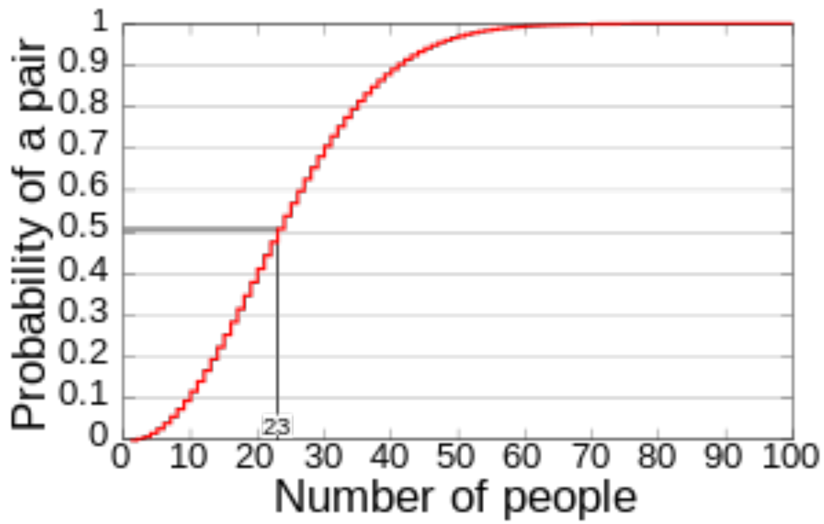

U
(universe of keys)
K
(actual
keys)
k1
k4
k2
k5
k3
collision
0
h(k1)
h(k4)
h(k2)=h(k5)
h(k3)
m–1

Collisions

- Collisions are possible!
- Multiple keys can hash to the same slot
  - Design hash functions such that collisions are minimized
- But avoiding collisions is impossible.
  - Birthday paradox
  - Design collision-resolution techniques
- Search will cost O(n) time in the worst case
- Hash value is an hint about where to start to search
- However, usually all operations can be made to have an expected complexity of O(1).

Birthday paradox

- Let's use birthday as hash function.
  - 365 slot in the array
  - Let's consider the probability of a collision



Natural numbers

- An hash function may assume that the keys are natural numbers
- When they are not, have to “interpret” them as natural numbers

Natural numbers hashing

- Division Method (compression)

$$h(k) = k \bmod m$$

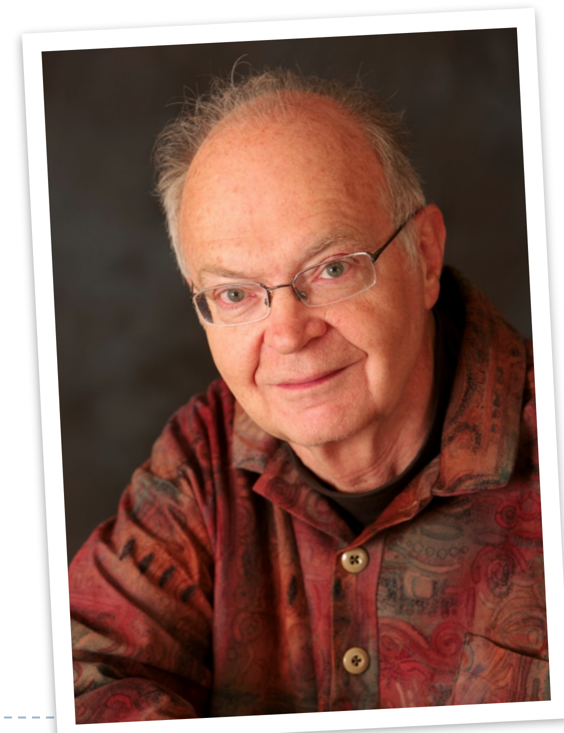
- Pros
  - Fast, since requires just one division operation
- Cons
  - Have to avoid certain values of m
- Good choice for *m* (recipe)
  - Prime
  - Not “too close” to powers of 2
  - Not “too close” to powers of 10

Natural numbers hashing

- Multiplication Method II

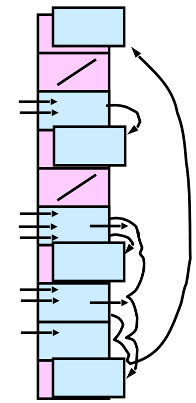
$$h(k) = k \cdot 2{,}654{,}435{,}761$$

- Pros
  - Works well for addresses
- Caveat (Donald Knuth)
  - $2{,}654{,}435{,}761 = \frac{2\hat{}32}{golden\ ratio}$

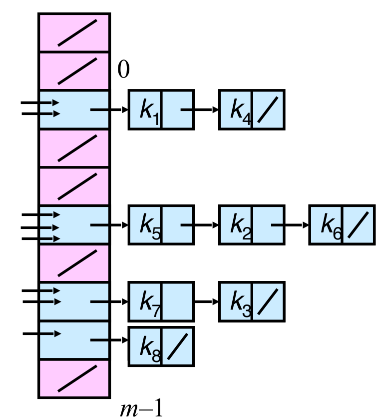


Resolution of collisions

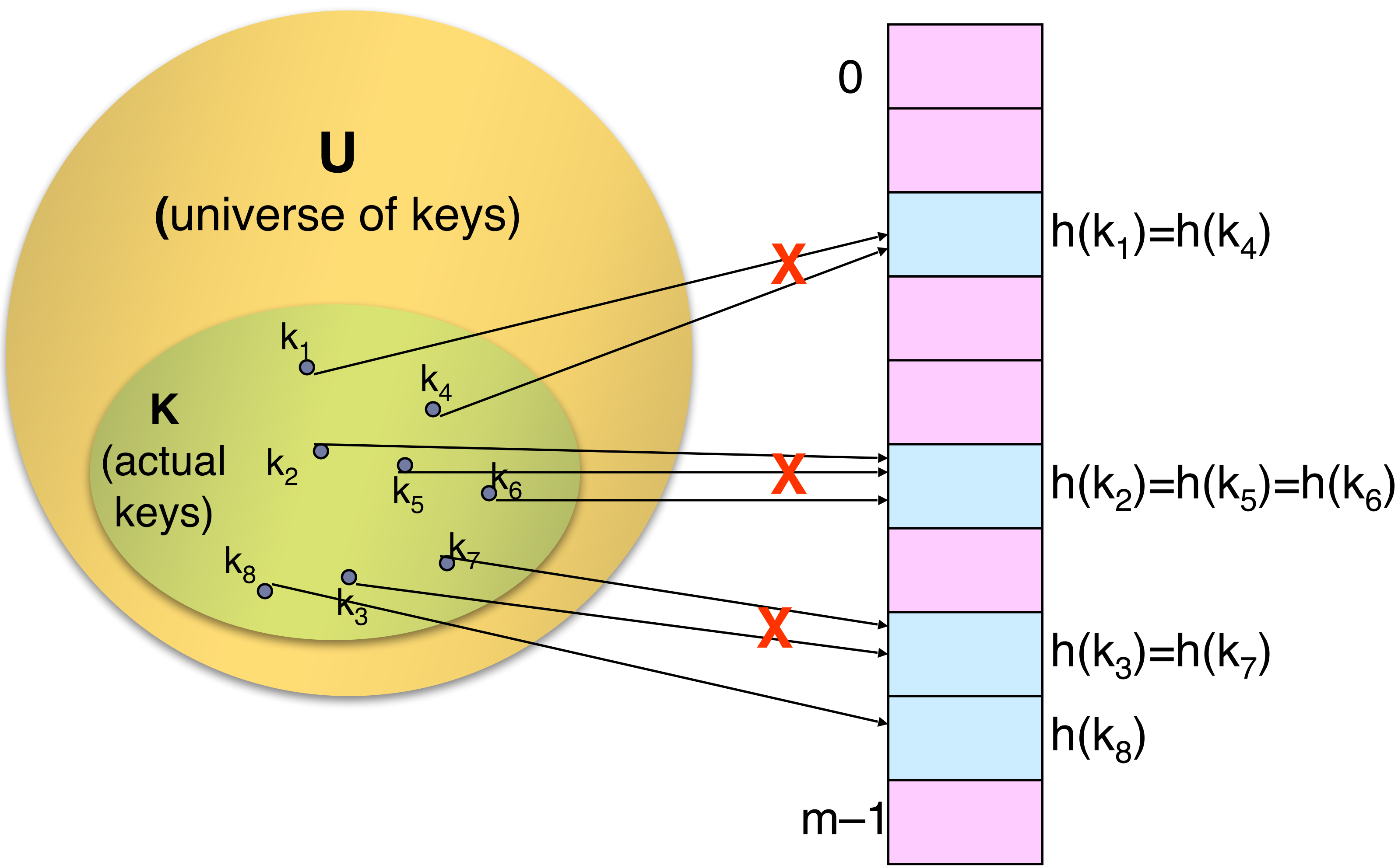
- Open Addressing
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table
  - “Double hashing”, “linear probing”, …



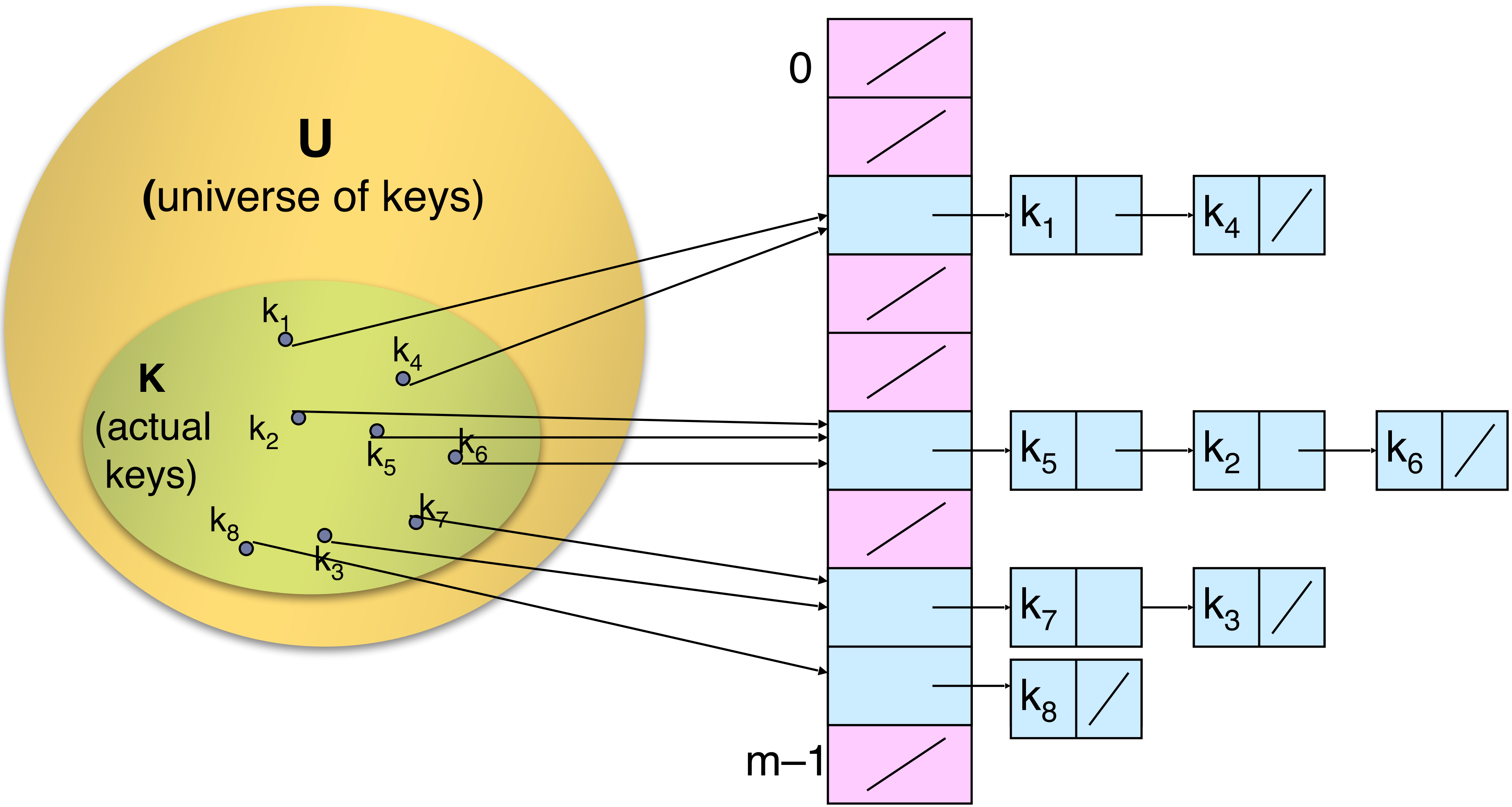
- Chaining
  - Store all elements that hash to the same slot in a linked list



Chaining


U
(universe of keys)
K
(actual
keys)
k1
k2
k3
k4
k5
k6
k7
k8
0
m–1
X
X
X
h(k1)=h(k4)
h(k2)=h(k5)=h(k6)
h(k3)=h(k7)
h(k8)

Chaining



U
(universe of keys)
K
(actual keys)
k1
k4
k2
k5
k6
k7
k8
k3
0
m–1
k1
k4
k5
k2
k6
k7
k3
k8

Chaining (analysis)

- Load factor $\alpha = n/m$ = average keys per slot
  - n – number of elements stored in the hash table
  - m – number of slots

  - If $n < m$, very few slots should have more than one entry
  - Even if $n < m$, collision occurs (birthday paradox)

Chaining (analysis)

- Worst-case complexity:

  O(n) (+ time to compute h(k))

Chaining (analysis)

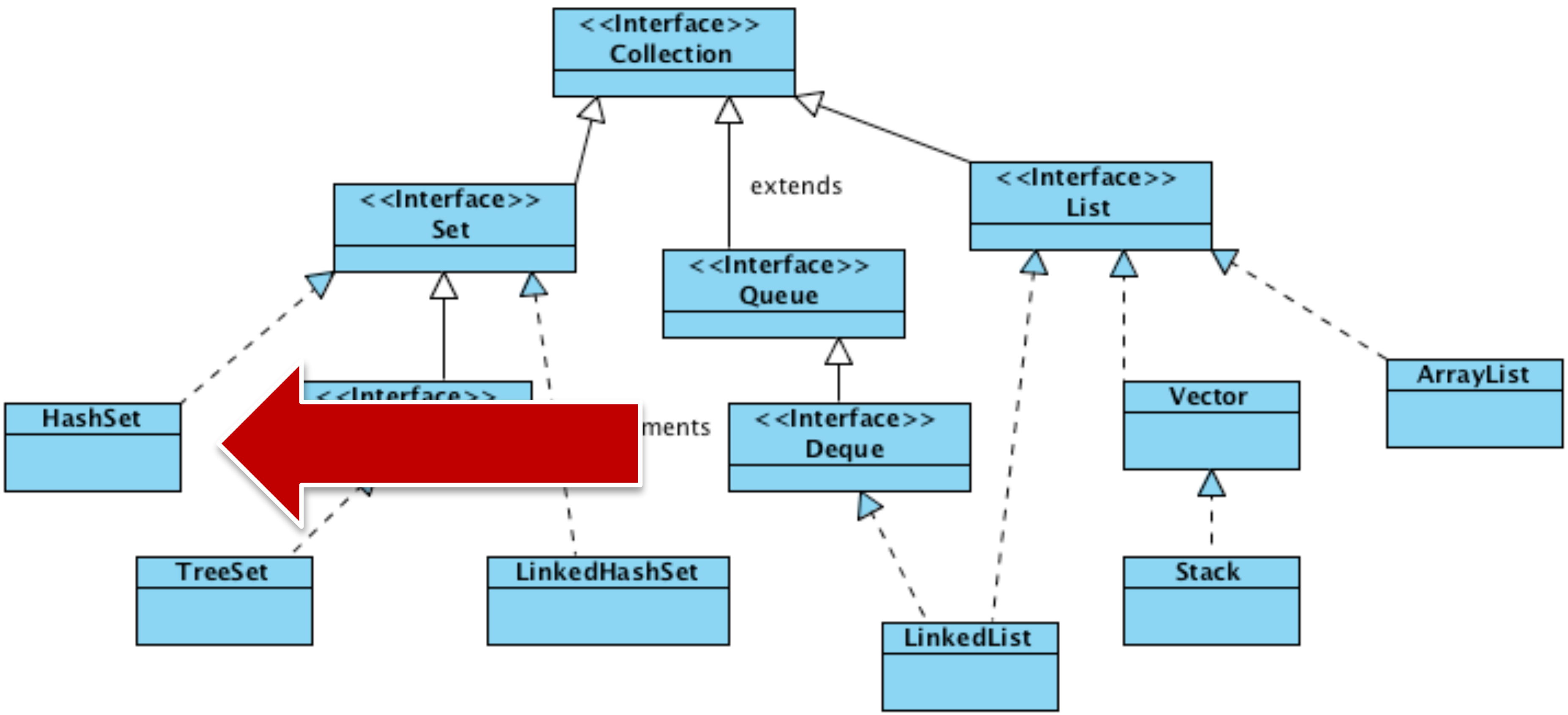
- Average depends on how $h(\cdot)$ distributes keys among m slots
- Let assume
  - Any key is equally likely to hash into any of the m slots
  - $h(k) = O(1)$
- Expected length of a linked list = load factor = $\alpha = n/m$
- Search(x) = $O(\alpha) + O(1) \approx O(1)$

A note on iterators

- **Collection** extends **Iterable**
- An **Iterator** is an object that enables you to traverse through a collection (and to remove elements from the collection selectively)
- You get an Iterator for a collection by calling its iterator() method

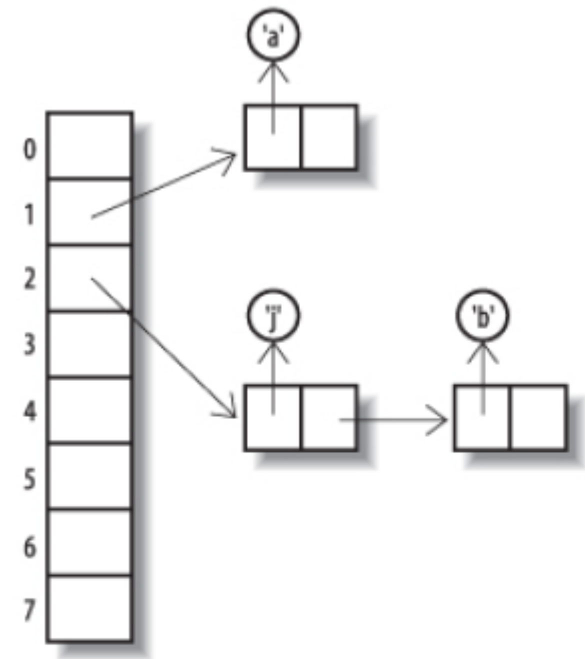
```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

Collection Family Tree

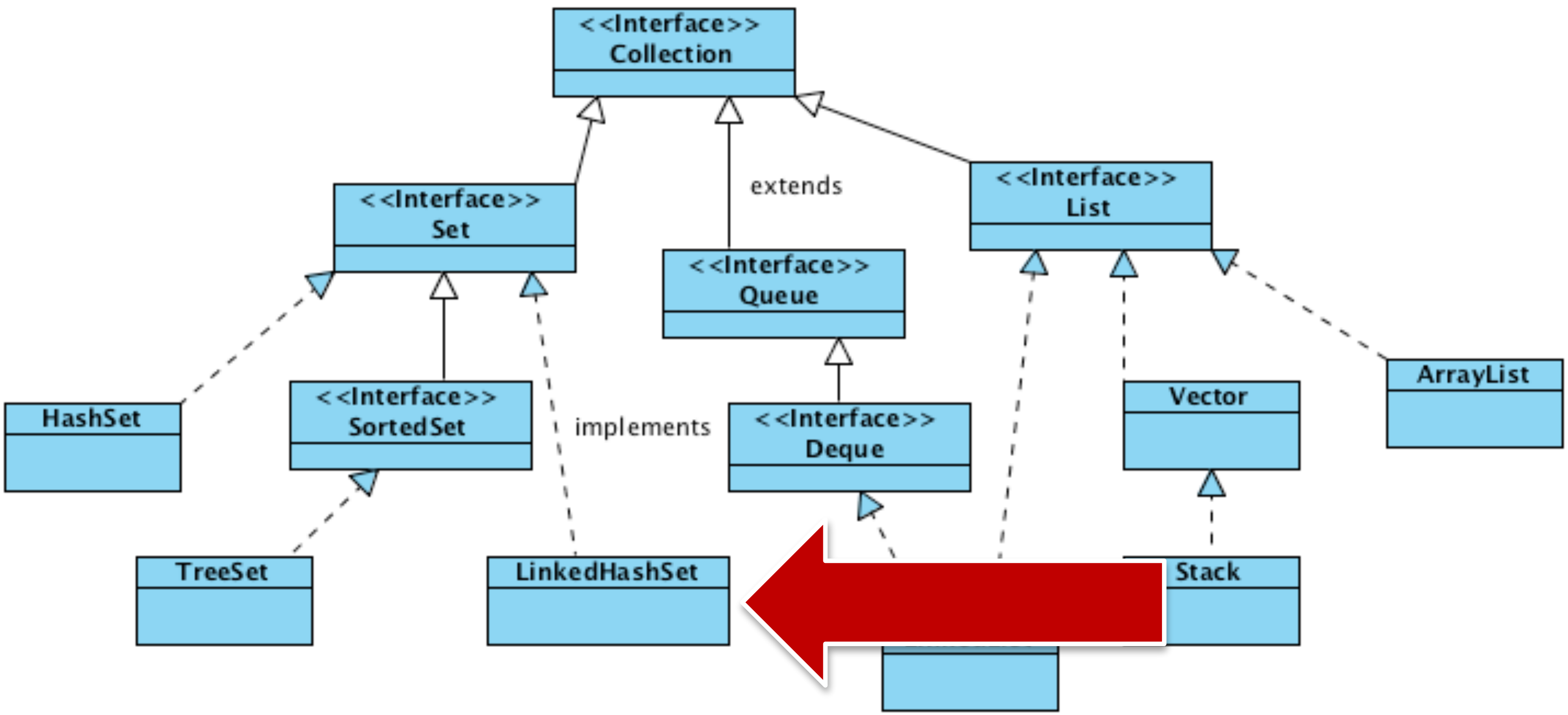


HashSet

- Add/remove elements
  - boolean **add**(element)
  - boolean **remove**(object)
- Search
  - boolean **contains**(object)
- No duplicates
- No positional Access
- Unpredictable iteration order!

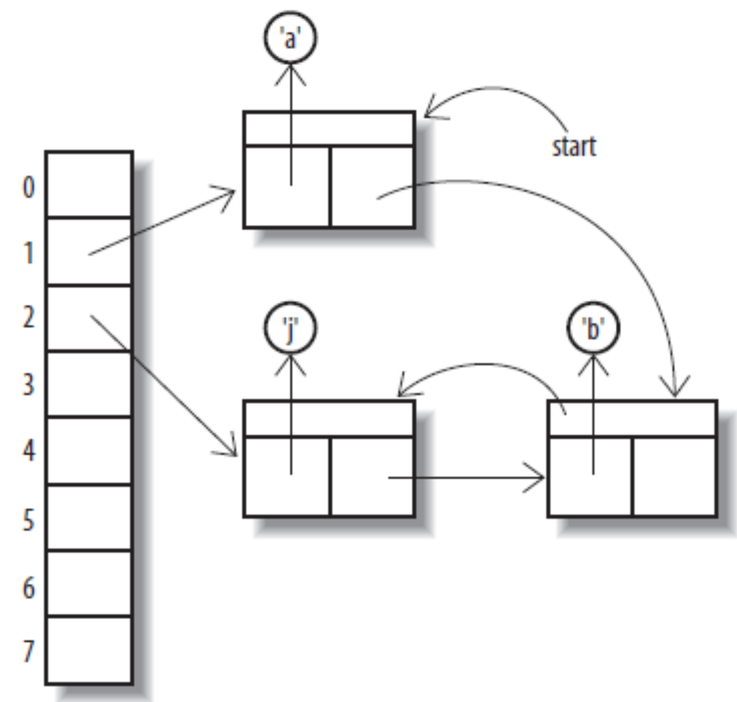


Collection Family Tree



LinkedHashSet

- Add/remove elements
  - boolean **add**(element)
  - boolean **remove**(object)
- Search
  - boolean **contains**(object)
- No duplicates
- No positional Access
- **<u>Predictable</u>** iteration order



Costructors

- public HashSet()
- public HashSet(Collection<? extends E> c)
- HashSet(int initialCapacity)
- HashSet(int initialCapacity, float loadFactor)

Costructors

- public HashSet()
- public HashSet(Collection<? extends E> c)
- HashSet(int initialCapacity)
- HashSet(int initialCapacity, float loadFactor)

16

16

75%

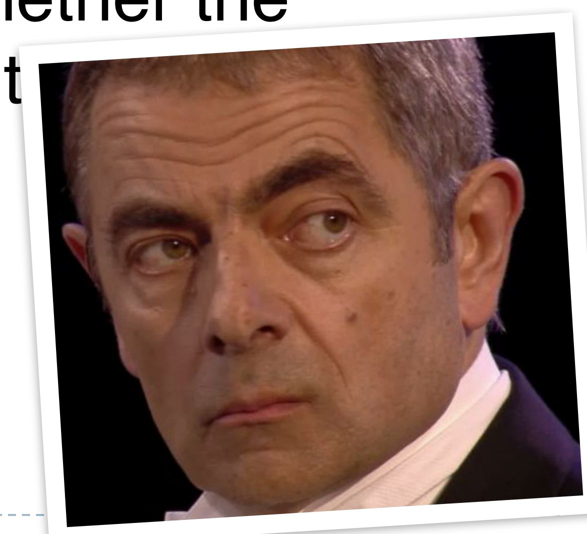
JCF's HashSet

- Built-in hash function
- Dynamic hash table resize
- Smoothly handles collisions (chaining)
- O(1) operations (well, usually)
- Take it easy!

Default hash function in Java

```
public boolean equals(Object obj);
public int hashCode();
```

- If two objects **are equal** according to the equals() method, then hashCode() must produce the same result
- If two objects **are not equal** according to the equals() method, performances are better whether the hashCode() produces different result



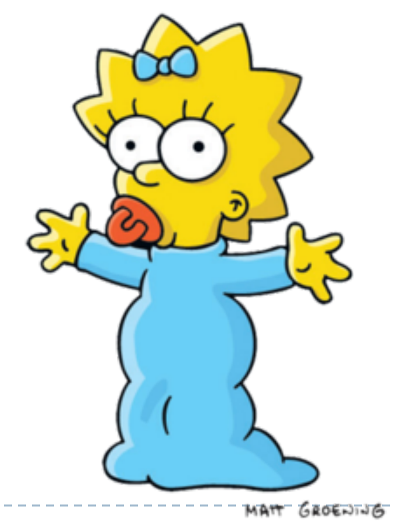
Hash functions in Java

```
public boolean equals(Object obj);
public int hashCode();
```

hashCode() and **equals()** should always be defined together

Hash functions in Java

- public int hashCode()
  - returns a 32-bit signed integer
    - 32-bit Float or 32-bit Integer could be used directly
    - Perfect hash function: map each input to a different hash value
  - Eclipse provides a convenient method to automatically generate equals() and hashCode() implementation



Recap

- == or !=
  - Used to compare the references of two objects

```
MyData foo = new MyData();
MyData bar = new MyData();

if(foo != bar) {
    System.out.println("References are different");
}

if(foo == bar){
    System.out.println("References are equal");
}
```

Recap

- equals()
  - Used to give **equality** information about the objects

```
MyData foo = new MyData();
MyData bar = new MyData();

if(foo.equals(bar)) {
    System.out.println("Objects have the same values");
} else {
    System.out.println("Objects have different values");
}
```

Recap

- hashCode()
  - Return the hash value of an object
  - Must behave in a way consistent with the same object equals() method

```
MyData foo = new MyData();
MyData bar = new MyData();

if(foo.equals(bar)) {
    if(foo.hashCode() == bar.hashCode()) {
        System.out.println("Hash code must be equal")
    }
}
```

Recap

- compareTo()
  - Gives the ordering of objects
  - Must be used only if need to order the object in a collection

```
MyData foo = new MyData();
MyData bar = new MyData();

if (foo.compareTo(bar) == 0){
    // WRONG!!
}
```

Implementing your own hash functions

- Grab your hash function from a professional

Licenza d'uso

- Queste diapositive sono distribuite con licenza Creative Commons "Attribuzione - Non commerciale - Condividi allo stesso modo (CC BY-NC-SA)"
- Sei libero:
  - di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera
  - di modificare quest'opera
- Alle seguenti condizioni:
  - **Attribuzione** — Devi attribuire la paternità dell'opera agli autori originali e in modo tale da non suggerire che essi avallino te o il modo in cui tu usi l'opera.
  - **Non commerciale** — Non puoi usare quest'opera per fini commerciali.
  - **Condividi allo stesso modo** — Se alteri o trasformi quest'opera, o se la usi per crearne un'altra, puoi distribuire l'opera risultante solo con una licenza identica o equivalente a questa.
- http://creativecommons.org/licenses/by-nc-sa/3.0/