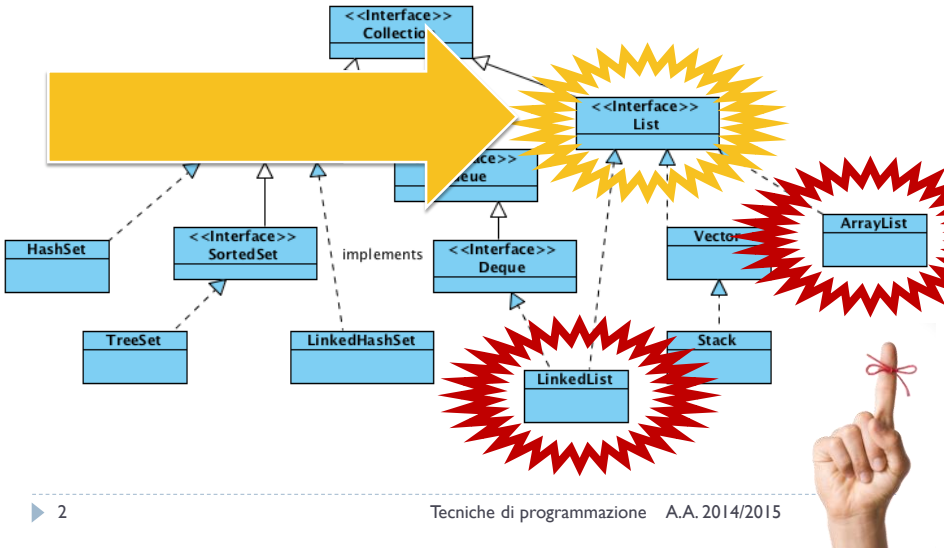
Sets

Collection that cannot contain duplicate elements.

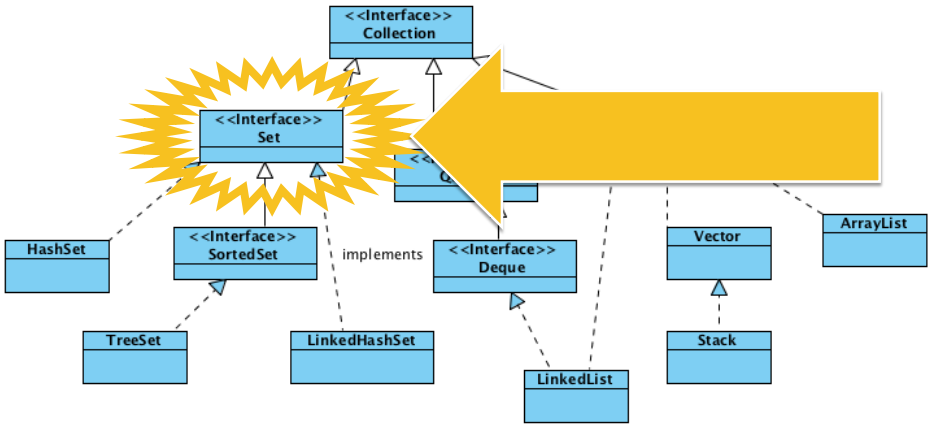
Collection Family Tree



ArrayList vs. LinkedList

	ArrayList	LinkedList
add(element)	IMMEDIATE	IMMEDIATE
remove(object)	SLUGGISH	~~IMMEDIATE~~
get(index)	IMMEDIATE	SLUGGISH
set(index, element)	IMMEDIATE	SLUGGISH
add(index, element)	SLUGGISH	SLUGGISH
remove(index)	SLUGGISH	SLUGGISH
contains(object)	SLUGGISH	SLUGGISH
indexOf(object)	SLUGGISH	SLUGGISH

Collection Family Tree

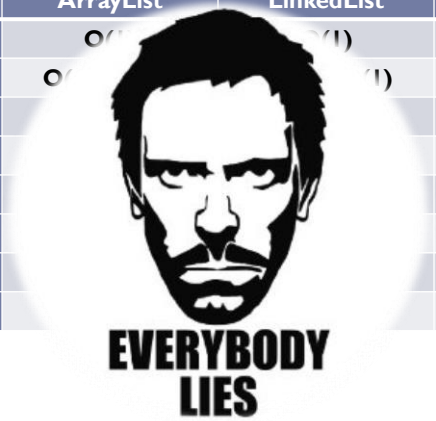


Set interface

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

Lists vs. Sets

	ArrayList	LinkedList	Set
add(element)	O(	(1)	O(1)
remove(object)	O(	(1)	O(1)
get(index)			n.a.
set(index, elem)			n.a.
add(index, elem)			n.a.
remove(index)			n.a.
contains(object)			O(1)
indexOf(object)			n.a.

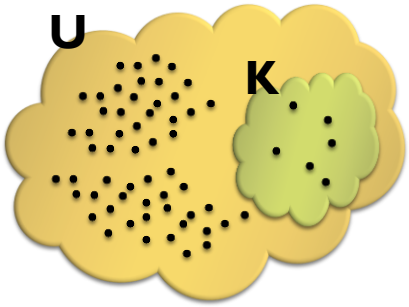


Hash Tables

A data structure implementing an associative array

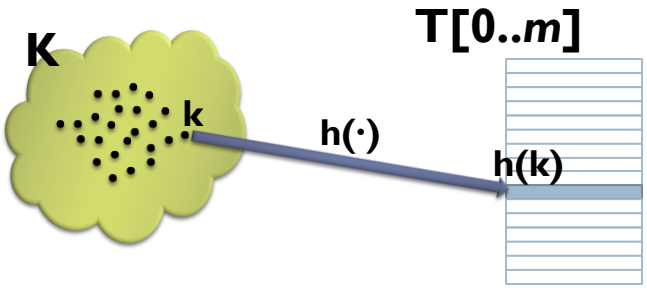
Notation

- 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

- 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
- "Any key should be equally likely to hash into any of the *m* slots, independent of where any other key hashes to" (Simple uniform hashing)
- Compression/expansion
  - $h_N : \mathbf{U} \rightarrow \mathbf{N}^+$
    $h(k) = h_N(k) \bmod m$
  - $h_R : \mathbf{U} \rightarrow [0, 1[\ \in \mathbf{R}$
    $h(k) = \lfloor h_R(k) \cdot m \rfloor$

Hash Function - Complexity

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

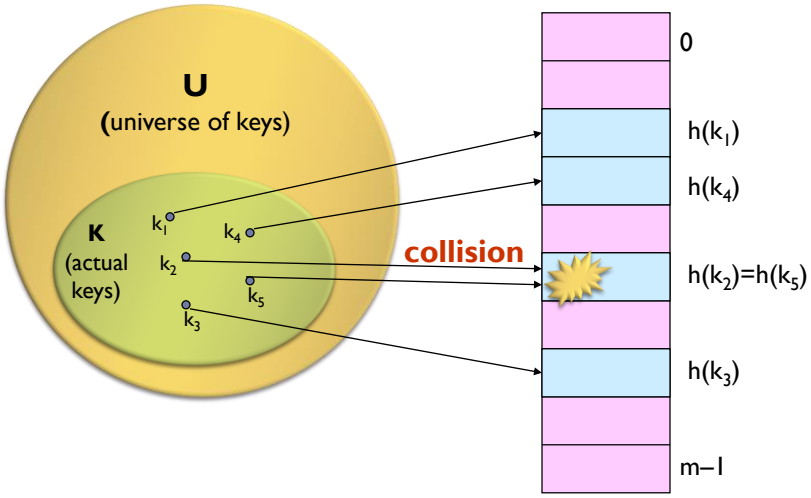
A simple hash function

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

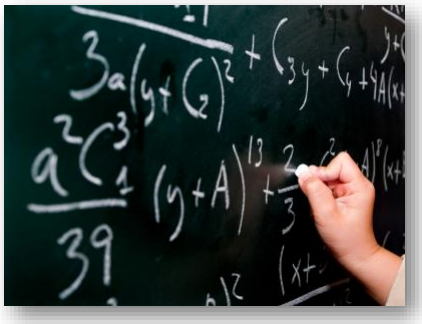


Collisions

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

Hash functions

- Simple uniform hashing
- Hash value should be independent of any patterns that might exist in the data
- No funneling



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

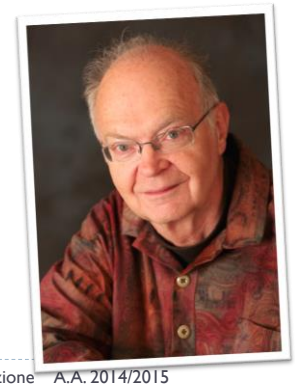
Natural numbers hashing

- Multiplication Method I

$$h_R(k) = < k \cdot A > = (k \cdot A - \lfloor k \cdot A \rfloor)$$

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

- Pros
  - Value of m is not critical (typically $m=2^p$)
- Cons
  - Value of A is critical
- Good choice for A (Donald Knuth)
  - $A = \frac{1}{\varphi} = \frac{\sqrt{5}+1}{2} - 1$

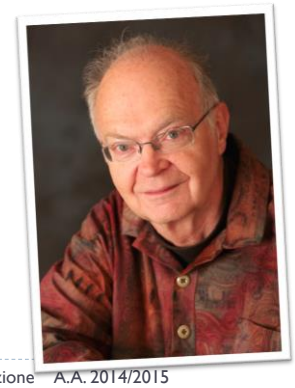


Natural numbers hashing

- Multiplication Method II

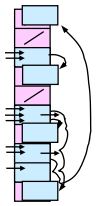
$$h(k) = k \cdot 2{,}654{,}435{,}761$$

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

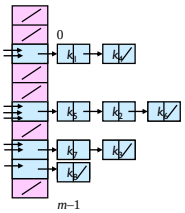


Resolution of collisions

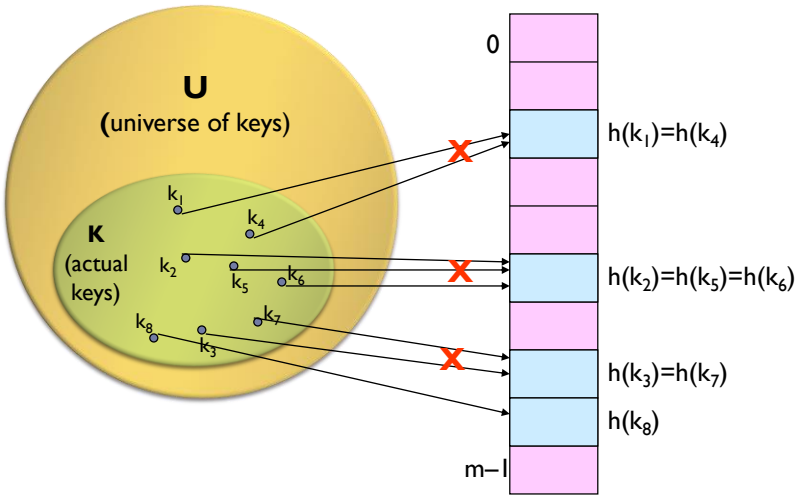
- Open Addressing
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table
  - "Double hashing", "linear probing", …



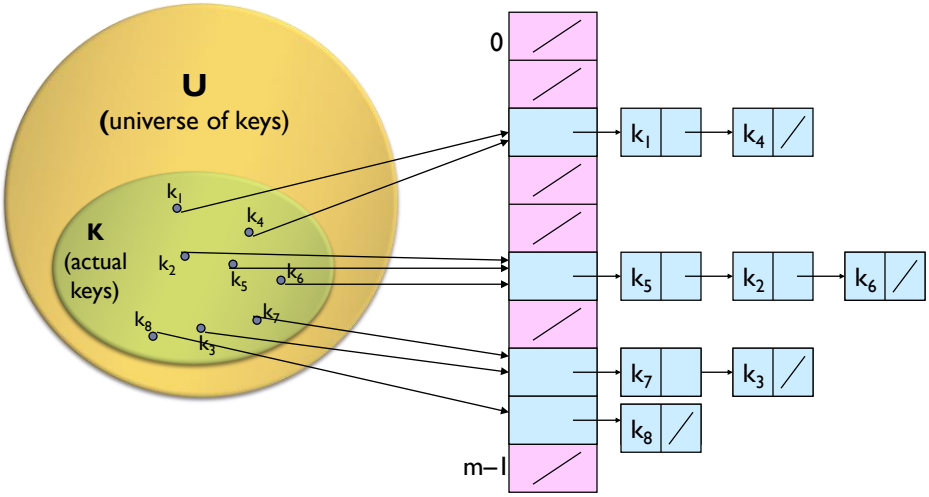
- Chaining
  - Store all elements that hash to the same slot in a linked list



Chaining



Chaining



Chaining (analysis)

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

Chaining (analysis)

- Worst-case complexity:

  $\Theta(n)$ (+ time to compute h(k))

Chaining (analysis)

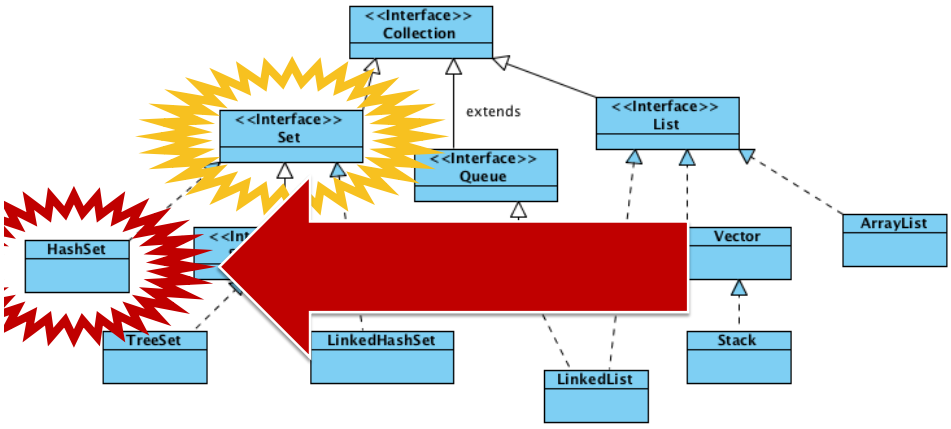
- Average depends on how h(·) distributes keys among m slots
- Let assume
  - Any key is equally likely to hash into any of the m slots
  - h(k) = O(1)
- Expected length of a linked list = load factor = α = n/m
- Search(x) = $O(\alpha) + O(1) \approx O(1)$

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

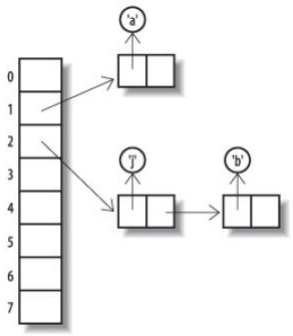
```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

Collection Family Tree

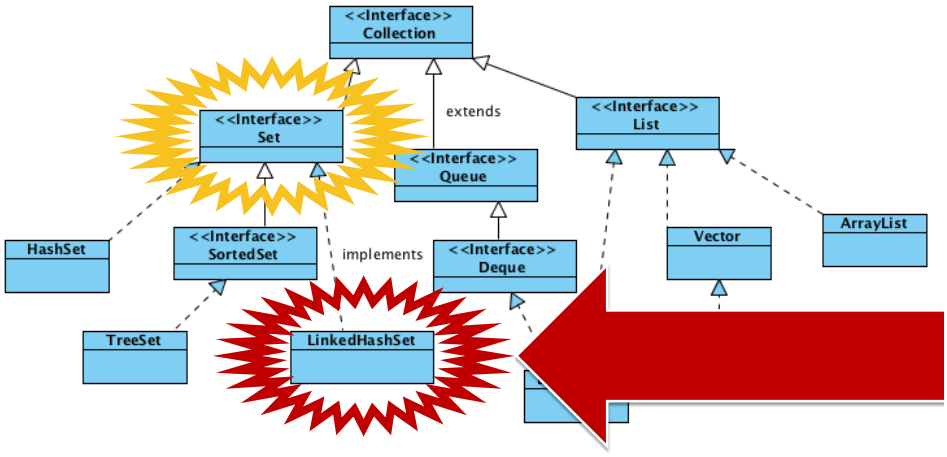


HashSet

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

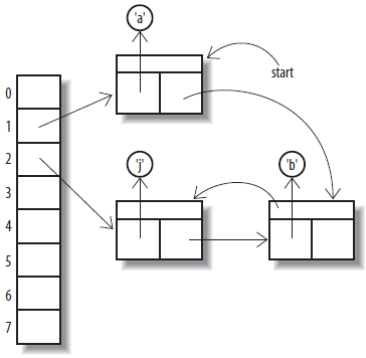


Collection Family Tree



LinkedHashSet

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

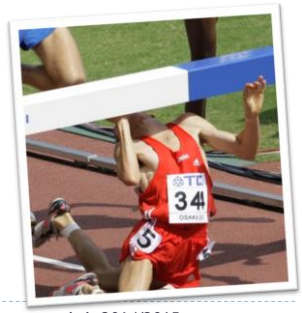
JCF's HashSet

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

Default hash function in Java

- In Java every class must provide a hashCode() method which digests the data stored in an instance of the class into a single 32-bit value
- In Java 1.2, Joshua Bloch implemented the java.lang.String hashCode() using a product sum over the entire text of the string

  $h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$
- But the basic Object's hashCode() is implemented by **converting the internal address of the object into an integer**!



Understanding hash in Java

```
public class MyData {
    public String name;
    public String surname;
    int age;
}
```

Understanding hash in Java

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

if(foo.hashCode() == bar.hashCode()) {
    System.out.println("FLICK");
} else {
    System.out.println("FLOCK");
}
```


Understanding hash in Java

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

foo.name = "Stephane";
foo.surname = "Hessel";
foo.age = 95;
bar.name = "Stephane";
bar.surname = "Hessel";
bar.age = 95;

if(foo.hashCode() == bar.hashCode()) {
    System.out.println("FLICK");
} else {
    System.out.println("FLOCK");
}
```

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 results

Hash functions in Java

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

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

public class MyData

```
public String name;
public String surname;
int age;

public MyData() { }

public MyData(String n, String s, int a) {
    name = n;
    surname = s;
    age = a;
}

[…]
```


public class MyData

```
@Override
public boolean equals(Object obj) {
    if (obj == this) {
       return true; // quite obvious ;-)
    }
    if (obj == null || obj instanceof MyData == false) {
       return false;   // not even comparable
    }
    // the real check!
    if(name.equals(((MyData)obj).name) == false ||
         surname.equals(((MyData)obj).surname) == false) {
       return false;
    }
    return true;
}

[…]
```

public class MyData

```
@Override
public b  lean equals(Object obj) {
    if (ob      this) {
        retu          // quite obvious ;-)
    }
                                              e) {
    }
        surname.equals(((MyData)obj).surname) == false) {
        return false;
    }
    return true;
}

[…]
```

The annotation **@Override** signals the compiler that overriding is expected,
and that it has to fail if an override does not occur

public class MyData

```
@Override
public int hashCode() {
    String tmp = name+":"+surname;
    return tmp.hashCode();
}
```

tmp will be "null:null" if MyData has not been initialized

Implementing your own hash functions

- Grab your hash function from a professional

Trivial Hash Function

This hash function helps creating predictable collisions (e.g., "ape" and "pea")

```
public long TrivialHash(String str)
{
    long hash = 0;

    for(int i = 0; i < str.length(); i++)
    {
       hash = hash + str.charAt(i);
    }

    return hash;
}
```

BKDR Hash Function

This hash function comes from Brian Kernighan and Dennis Ritchie's book "The C Programming Language". It is a simple hash function using a strange set of possible seeds which all constitute a pattern of 31....31...31 etc, it seems to be very similar to the DJB hash function.

```
public long BKDRHash(String str)
{
   long seed = 131; // 31 131 1313 13131 131313 etc..
   long hash = 0;

   for(int i = 0; i < str.length(); i++)
   {
      hash = (hash * seed) + str.charAt(i);
   }

   return hash;
}
```


RS Hash Function

A simple hash function from Robert Sedgwicks Algorithms in C book

```
public long RSHash(String str)
{
   int b     = 378551;
   int a     = 63689;
   long hash = 0;

   for(int i = 0; i < str.length(); i++)
   {
      hash = hash * a + str.charAt(i);
      a    = a * b;
   }

   return hash;
}
```

DJB Hash Function

An algorithm produced by Professor Daniel J. Bernstein and shown first to the world on the usenet newsgroup comp.lang.c. It is one of the most efficient hash functions ever published

```
public long DJBHash(String str)
{
   long hash = 5381;

   for(int i = 0; i < str.length(); i++)
   {
      hash = hash * 33 + str.charAt(i);
   }

   return hash;
}
```

JS Hash Function

A bitwise hash function written by Justin Sobel

```
public long JSHash(String str)
{
   long hash = 1315423911;

   for(int i = 0; i < str.length(); i++)
   {
      hash ^= ((hash << 5) + str.charAt(i) + (hash >> 2));
   }

   return hash;
}
```

SDBM Hash Function

This is the algorithm of choice which is used in the open source SDBM project. The hash function seems to have a good over-all distribution for many different data sets. It seems to work well in situations where there is a high variance in the MSBs of the elements in a data set.

```
public long SDBMHash(String str)
{
   long hash = 0;

   for(int i = 0; i < str.length(); i++)
   {
      hash = str.charAt(i) + (hash << 6) +
         (hash << 16) - hash;
   }

   return hash;
}
```


DEK Hash Function

An algorithm proposed by Donald E. Knuth in *The Art Of Computer Programming* (Volume 3), under the topic of sorting and search chapter 6.4.

```
public long DEKHash(String str)
{
   long hash = str.length();

   for(int i = 0; i < str.length(); i++)
   {
      hash = ((hash << 5) ^ (hash >> 27)) ^ str.charAt(i);
   }

   return hash;
}
```

DJB Hash Function

The algorithm by Professor Daniel J. Bernstein (alternative take)

```
public long DJBHash(String str)
{
   long hash = 5381;

   for(int i = 0; i < str.length(); i++)
   {
      hash = ((hash << 5) + hash) ^ str.charAt(i);
   }

   return hash;
}
```

PJW Hash Function

This hash algorithm is based on work by Peter J. Weinberger of AT&T Bell Labs. The book *Compilers (Principles, Techniques and Tools)* by Aho, Sethi and Ulman, recommends the use of hash functions that employ the hashing methodology found in this particular algorithm

```
public long PJWHash(String str)
{
   long BitsInUnsigned = (long)(4 * 8);
   long ThreeQuarters  = (long)((BitsInUnsigned *3) /4);
   long OneEighth      = (long)(BitsInUnsigned / 8);
   long HighBits       = (long)(0xFFFFFFFF) <<
                          (BitsInUnsigned - OneEighth);
   long hash           = 0;
   long test           = 0;

[…]
```

PJW Hash Function

```
[…]

  for(int i = 0; i < str.length(); i++)
   {
      hash = (hash << OneEighth) + str.charAt(i);

      if((test = hash & HighBits)  != 0)
      {
         hash = (( hash ^ (test >> ThreeQuarters)) &
                      (~HighBits));
      }
   }

   return hash;
}
```


ELF Hash Function

Similar to the PJW Hash function, but tweaked for 32-bit processors. Its the hash function widely used on most UNIX systems

```
public long ELFHash(String str)
{
   long hash = 0, x = 0;

   for(int i = 0; i < str.length(); i++)
   {
      hash = (hash << 4) + str.charAt(i);
      if((x = hash & 0xF0000000L) != 0)
         hash ^= (x >> 24);
      hash &= ~x;
   }
   return hash;
}
```

Maps

a.k.a, associative array, map, or dictionary

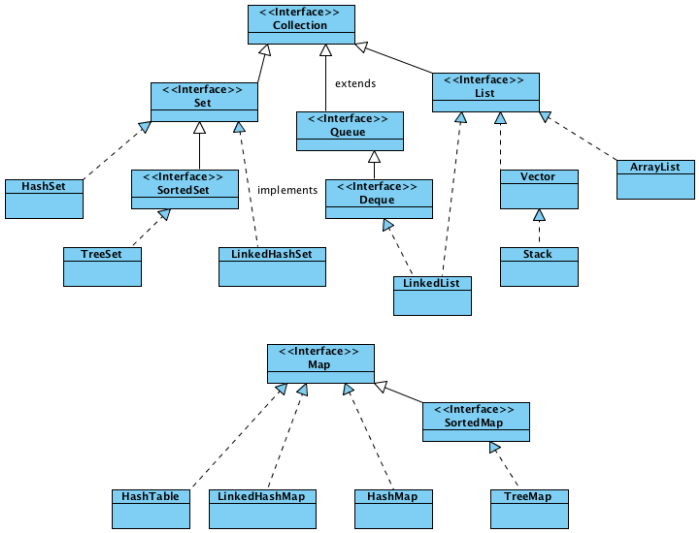
Definition

- In computer science, an **associative array**, **map**, or **dictionary** is an abstract data type composed of (key, value) pairs, such that each possible key appears at most once
- Native support in most modern languages (perl, python, ruby, go, …). E.g.,

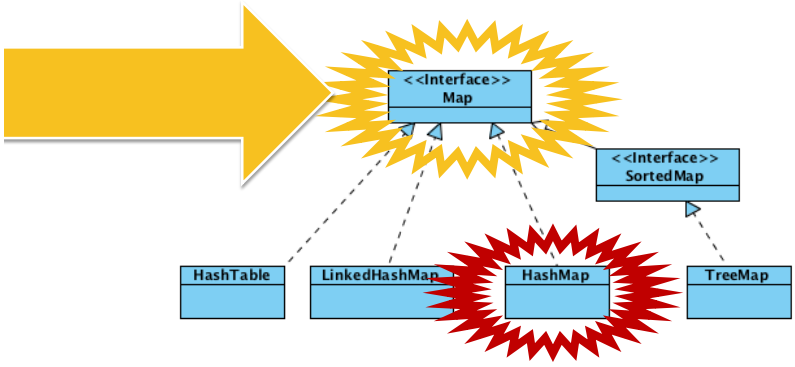
```
V1[42] = "h2g2"
V2["h2g2"] = 42
```

- Implemented through hash tables

Java Collection Framework



Collection Family Tree



Map interface

- Map<K,V>
  - K: the type of keys maintained by this map
  - V: the type of mapped values
- Add/remove elements
  - value **put**(key, value)
  - value **remove**(key)
- Search
  - boolean **containsKey**(key)
  - boolean **containsValue**(value)

Map interface

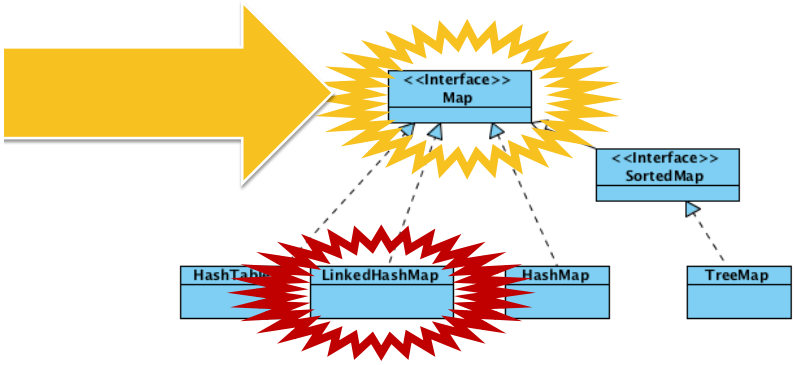
- Map<K
  - K: th
  - V: th
- Add/re
  - value **put**(key, value)
  - value **remove**(key)
- Search
  - boolean **containsKey**(key)
  - boolean **containsValue**(value)

containsValue() will probably require time *linear in the map size* for most implementations of the Map interface – ie. it is O(N)

Map interface (cont)

- Nested Class
  - Map.Entry<K,V>
  - A map entry (key-value pair).
- Set<Map.Entry<K,V>> entrySet()
  - Returns a **Set view** of the mappings contained in this map
- Set<K> keySet()
  - Returns a **Set view** of the keys contained in this map
- Collection<V> values()
  - Returns a **Collection** view of the values contained in this map

Collection Family Tree

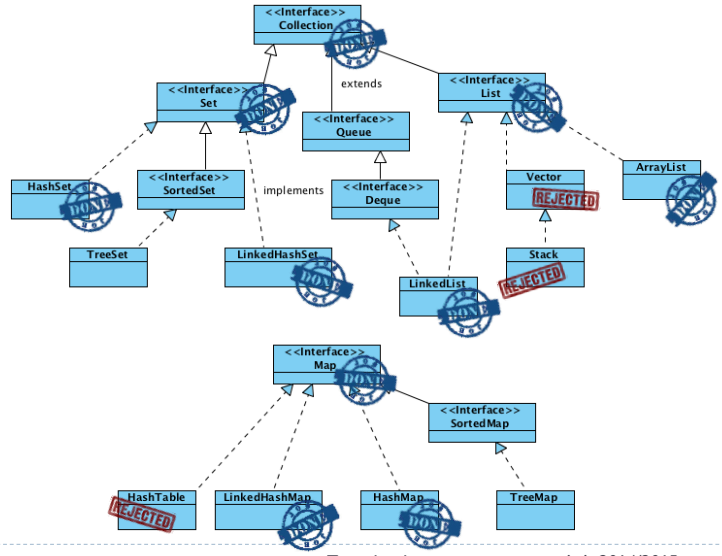


LinkedHashMap

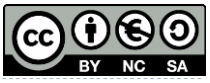
- Maintains a doubly-linked list running through all of its entries
- This linked list defines the iteration ordering (**normally** insertion-order)
- Insertion order is not affected if a key is re-inserted

Wrap-up

Java Collection Framework



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/