

# Insegnamento di Laboratorio di algoritmi e strutture dati

## Hash Table

Roberto Posenato

ver. 0.7.1, 07/03/2009

# Introduzione

## Definizione 1 (Dizionario)

ADT in cui le informazioni sono individuate in modo univoco tramite *chiavi* e in cui le operazioni fondamentali sono: *inserimento*, *cancellazione* e *ricerca*.

## Esempi

- Il dizionario classico! La chiave è il lemma e l'informazione è la spiegazione dello stesso.
- L'elenco telefonico. La chiave è il nome dell'utente. L'informazione è il numero di telefono.
- ...

# Introduzione

## Dizionari implementati come tabelle

- Struttura dati è una *tabella*:

Chiave	Informazione
3	Oggetto <sub>3</sub>
7	Oggetto <sub>7</sub>
...	...

- Chiave  $\in U = \{0, 1, \dots, u - 1\}$  (*Universo delle chiavi*) è il campo che individua in modo univoco l'oggetto.
- Le operazioni fondamentali:
  - `ricerca(chiave)`: restituisce l'oggetto con chiave *chiave*;
  - `inserisci(chiave, oggetto)`: inserisce *oggetto* associandolo alla chiave *chiave*;
  - `cancella(chiave)`: cancella l'oggetto con chiave *chiave*.

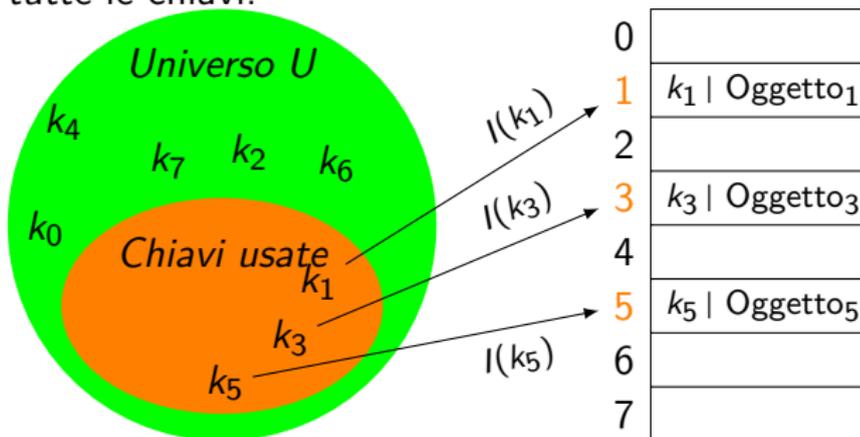
### Nota!

È opportuno determinare un'implementazione dei dizionari che esegua le operazioni di inserimento e ricerca in modo efficiente!

# Introduzione

## Tabella a indirizzamento diretto

- Se  $|U|$  è limitato, allora si può usare una tabella che contenga tutte le chiavi.



- Costo operazioni è  $O(1)$ !



# Introduzione

## Tabella a indirizzamento diretto

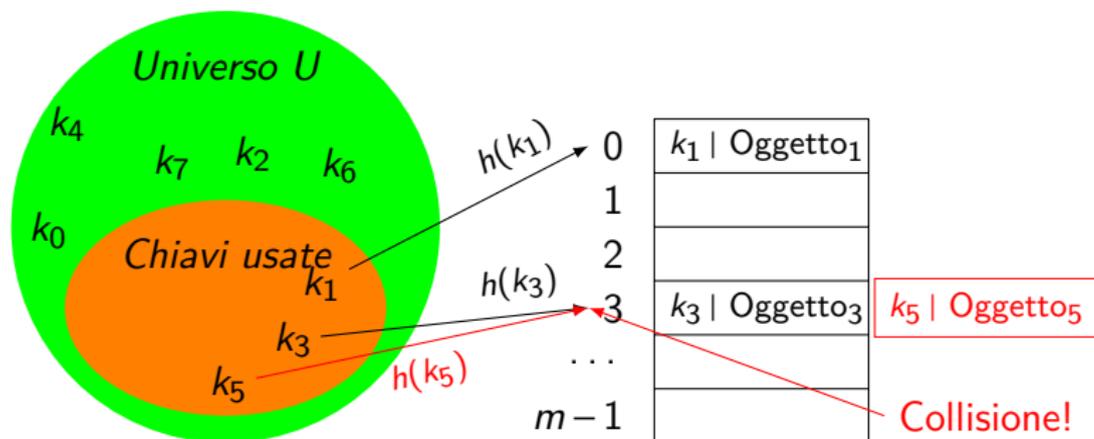
- Se  $|U|$  è troppo grande, la tabella non è realizzabile;
- Se  $n$  (numero chiavi realmente usate) è piccolo, si ha uno spreco di memoria;

### Possibile strategia alternativa

- si utilizza tabella di dimensioni  $m \approx n$ ;
- si utilizza una funzione che trasforma la chiave per trovare una posizione di memorizzazione nella tabella:  
 $h()$  - funzione hash

# Tabella Hash

## Schema generale



- $m \ll |U|$
- Funzione hash  $h: U \rightarrow \{0, \dots, m-1\}$ 
  - disperdere in modo uniforme le chiavi;
  - suriettiva;  $\Leftarrow$  **Possibili collisioni!**
  - se è biunivoca quando si considerano solo le chiavi usate, allora è una funzione **hash perfetta**.

# Tabella Hash

## Funzione Hash: come scegliere?

Funzione hash  $h : U \rightarrow \{0, \dots, m-1\}$

- funzione *perfetta*: distribuisce le chiavi in modo casuale uniforme senza collisioni (biunivoca);
- funzioni hash perfette sono poche e non sempre facili da determinare:

Dati  $n$  dati e una tabella con  $m$  celle ( $n \leq m$ ):

- # funzioni hash possibili =  $m^n$  (disposizioni con rip.);
- # funzioni hash perfette =  $\frac{m!}{(m-n)!}$  (disposizioni senza rip.);
- Esempio:  $n = 50$  e  $m = 100$ , una funzione su un milione è perfetta!



# Tabella Hash

## Funzione Hash: come scegliere?

Funzione hash  $h : U \rightarrow \{0, \dots, m-1\}$

- una *buona* funzione hash distribuisce le chiavi in modo casuale minimizzando le collisioni;
- esistono diversi metodi che producono funzioni di buona qualità:
  - metodo della *divisione*;
  - metodo della *moltiplicazione*;
  - metodo del *ripiegamento (folding)*;
- tutte hanno dominio = `int`
- le chiavi diverse da `int` devono essere quindi convertite prima in `int`



# Tabella Hash

Funzione Hash: conversione verso gli interi

## Conversione chiave alfanumerica

- Sia  $k = c_n c_{n-1} \dots c_0$  dove  $c_i$  è un char;
- Se si vuole determinare un intero in corrispondenza biunivoca con  $k$ , allora la conversione è  $k' = \sum (c_i 65536^i)$  dove la base è 65536 perché in Java un char è di 16 bit!!!
- Se si ha la certezza che i caratteri sono del sottoinsieme Latin1 (set ASCII più alcune lettere accentate di vario tipo e segni grafici), allora la base può essere 256.
- Se si fanno ulteriori restrizioni sul set di caratteri possibili e se si riassegna gli ordinali ai caratteri possibili a partire da 0, allora la base può essere posta pari alla cardinalità dell'insieme.



# Tabella Hash

## Funzione Hash: metodo divisione

- Sia  $m$  dimensione tabella;
- Dato che  $h()$  deve restituire sempre un indice valido  $[0, \dots, m-1]$ , la più semplice funzione definibile è  $h(k) = k \% m$
- Avvertenze:
  - $m$  è critico: meglio scegliere numero primo non vicino a potenze di 2;
  - altrimenti, miglior  $m$  è un numero con fattori primi  $> 20$  (Lum, 1971);
  - altrimenti, se  $r =$  base conversione chiavi,  $m \neq ri \pm a$  (con  $i$  e  $a$  interi)

# Tabella Hash

Funzione Hash: metodo divisione

## Esempi **negativi** di $h()$ basati sul metodo della divisione

- se  $m = 2^p$ , dove  $p$  è un numero primo, allora  $h(k) =$  primi  $p$  bit meno significativi di  $k$ ;
- se  $m = ri \pm a$  (con  $i$  e  $a$  interi), allora le chiavi sono distribuite su uno spazio  $[0, \dots, r]$ :
  - $r = 256$ , scegliendo  $m = 65537$  che è pari a  $256^2 + 1$  (primo!), si dimostra che  $h(c_2c_1c_0) = (\text{ord}(c_1)\text{ord}(c_0) - \text{ord}(c_2))_{256}$



# Tabella Hash

## Funzione Hash: metodo moltiplicazione

- Sia  $m$  dimensione tabella e  $0 < A < 1$  una costante;
- Si fissa  $h(k) = \lceil m(kA \% 1) \rceil$
- Avvertenze:
  - $kA$  è un valore decimale  $< k$ ;
  - Non essendo  $kA$  un int,  $kA \% 1$  è la parte decimale del valore  $kA$ , quindi è  $< 1$ ;
  - $m$  NON è critico. Valore tipico  $m = 2^p$ , dove  $p$  è un numero primo
  - D. Knuth suggerisce  $A \approx \frac{\sqrt{5}-1}{2} \approx 0.6180$  (sezione aurea)

# Tabella Hash

## Funzione Hash: metodo folding

Diversi metodi di folding:

- Più semplice:
  - chiave divisa in due o più parti  $k_i$ ;
  - $h(k) = (\sum k_i) \% m$  dove  $m$  è dimensione tabella;

### Metodo folding semplice

$$k = 013402122$$

$$k_2 = 013, k_1 = 402, k_0 = 122$$

$$h(k) = 013 + 402 + 122 = 537$$



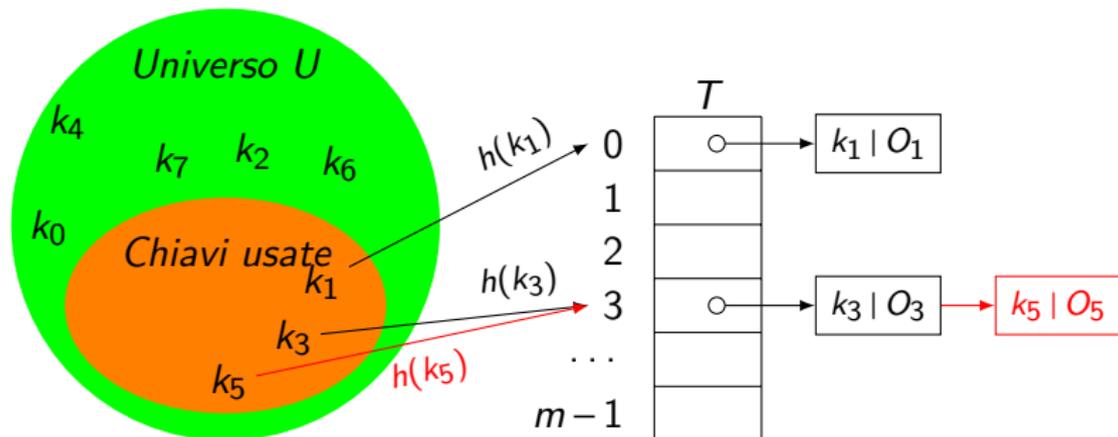
# Tabella Hash

## Funzione Hash: collisioni

- Con funzioni hash non perfette più di una chiave può essere assegnata alla stessa posizione (**collisione**)
- Diverse tecniche per gestire le collisioni:
  - concatenamento (chaining)
  - indirizzamento aperto (open addressing)
    - reindirizzamento lineare (linear probing)
    - doppio hashing

# Tabella Hash

Funzione Hash: concatenamento



- Tutti gli elementi con medesima chiave hash sono posti su una lista;
- `inserisci( $k, o$ )`: inserisci  $o$  nella lista  $T[h(k)]$ ;
- `ricerca( $k$ )`: restituisci l'oggetto con chiave  $k$  nella lista  $T[h(k)]$ ;
- `cancella( $k$ )`: cancella l'oggetto con chiave  $k$  nella lista  $T[h(k)]$ .



# Tabella Hash

## Funzione Hash: concatenamento

### Prestazioni

Operazione	Caso peggiore	Caso medio
inserisci	$\Theta(n)$	$\Theta(\alpha + 1)$
ricerca	$\Theta(n)$	$\Theta(\alpha + 1)$
cancella	$\Theta(n)$	$\Theta(\alpha + 1)$

- $\alpha = \frac{n}{m}$  è indice di carico,
- $n$  = celle dell'array occupate,
- 1 è dato dalla computazione di  $h()$ .
- Si ricorda che l'operazione `inserisci()` deve sovrascrivere l'elemento di chiave  $k$  se esiste o aggiungerne uno nuovo altrimenti. Quindi prima si deve ricercare l'elemento. Da cui la complessità nel caso peggiore.

# Tabella Hash

## Funzione Hash: indirizzamento aperto

- In caso di conflitto, si usano altre celle della stessa tabella per memorizzare il nuovo oggetto.

$$h(k_1) = 0$$

$$h(k_3) = h(k_5) = 3$$

0	$k_1   \dots$
1	$k_5   \dots$
2	
3	$k_3   \dots$ $k_5   \dots$
...	
$m-1$	

- Il tipo di selezione delle celle determina tipo di reindirizzamento:
  - lineare;
  - doppio hashing.

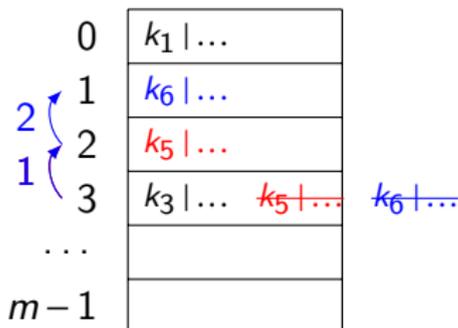


# Tabella Hash

## Funzione Hash: indirizzamento aperto lineare

- In inserimento, in caso di conflitto, si ricerca la prima cella libera, scandendo le celle a una a una, a partire da quella adiacente a quello corretta:

- Si supponga che si inserisca, in ordine,  $k_1$ ,  $k_3$ ,  $k_5$  e  $k_6$  e che  $h(k_1) = 0$ ,  $h(k_3) = h(k_5) = h(k_6) = 3$ ;
- L'inserimento di  $k_5$  richiede la ricerca di 1 cella libera: sufficiente 1 salto;
- L'inserimento di  $k_6$  richiede la ricerca di 1 cella libera: necessari 2 salti;





# Tabella Hash

## Funzione Hash: indirizzamento aperto lineare

Dettaglio operazioni:

- `inserisci(k, o)`: inserisci `o` in prima cella libera di  $T[h'(k, i)]$ , dove  $h'(k, i) = (h(k) + i) \% m$  e  $i = [0, \dots, m - 1]$ ;  
**Nota: la ricerca si fa nelle celle successive anziché in quelle precedenti. Cambia qualcosa?**
- `ricerca(k)`: ricerca oggetto con chiave `k` nella sequenza  $T[h'(k, i)]$ . Se `k` non c'è, la fine ricerca è data da cella vuota o  $i = m - 1$ ;
- `cancella(k)`: cancella oggetto con chiave `k` nella sequenza  $T[h'(k)]$  **senza creare buchi!**

### Nota!

Il metodo soffre del cosiddetto *Problema del clustering primario*:

- cluster: sequenza di celle adiacenti occupate;
- un cluster tende a crescere velocemente, vanificando la funzione hash.



# Tabella Hash

Funzione Hash: indirizzamento aperto lineare

## Prestazioni

- tutte le operazioni sono dipendenti dalla ricerca;
- D. Knuth (1998) ha sviluppato delle formule che approssimano il # di confronti necessari:

Operazione	Caso medio
ricerca con successo	$\frac{1}{2} \left(1 + \frac{1}{1-\alpha}\right)$
ricerca con insuccesso	$\frac{1}{2} \left(1 + \frac{1}{1-\alpha}\right)^2$

- $\alpha = \frac{n}{m}$  è indice di carico.



# Tabella Hash

## Funzione Hash: indirizzamento aperto doppio

- In inserimento, in caso di conflitto, si ricerca una cella libera mediante l'uso di un'altra funzione hash.
- Dettaglio operazioni:
  - `inserisci(k, o)`: inserisci `o` in prima cella libera di  $T[h'(k, i)]$ , dove

$$h'(k, i) = (h(k) + i \cdot p(k)) \% m$$

$p(k)$  è un'altra funzione hash (**funzione probe**) e  $i = [0, \dots, m - 1]$ ;

**Nota:**  $p(k)$  deve garantire di poter accedere a tutte le celle in  $m$  chiamate.

- `cancella(k)` e `ricerca(k)` sono analoghe.



# Tabella Hash

Funzione Hash: indirizzamento aperto doppio

## Nota!

- problema clustering primario non presente;
- $p()$  deve garantire la copertura di tutto lo spazio array
  - condizione sufficiente è che  $\text{mcd}(p(k), m) = 1$ ;  
Esempio:  $m$  primo e  $0 < p(k) < m$ .
  - altra condizione sufficiente è che se  $m = 2^c$ , allora  $0 < p(k) < m$  sia dispari.



# Tabella Hash

Funzione Hash: indirizzamento aperto doppio

## Prestazioni

- tutte le operazioni sono dipendenti dalla ricerca;
- D. Knuth (1998) ha sviluppato delle formule che approssimano il # di confronti necessari:

Operazione	Caso medio
ricerca con successo	$\frac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right)$
ricerca con insuccesso	$\frac{1}{1-\alpha}$

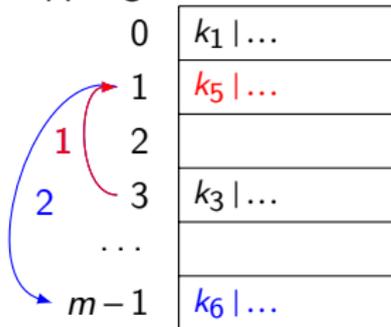
- $\alpha = \frac{n}{m}$  è indice di carico.

# Tabella Hash

## Cancellazione

L'operazione di cancellazione è delicata:

- Concatenazione: semplice eliminazione elemento dalla lista;
- Indirizzamento aperto:
  - maggior cautela!
  - Una semplice cancellazione può determinare dei buchi che violano la struttura dell'indirizzamento aperto.
  - Si supponga di voler cancellare la chiave  $k_5$ :



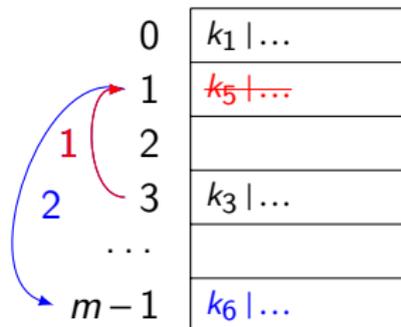
- Se non si compatta la lista dei salti, il vuoto lasciato dalla cancellazione di  $k_5$  rende impossibile la ricerca della chiave  $k_6$ .

# Tabella Hash

## Cancellazione

Possibile soluzione:

- NON rimuovere le chiavi ma *marcarle* come *cancellate*;



- ricerche tornano a funzionare;
- inserimenti successivi sovrascrivono chiavi *cancellate*.



# Implementazione

## Interfaccia Dizionario

### Sintesi interfaccia Dizionario

```
/** Javadoc... */
public interface Map {
    /** Internal interface to represent a map entry.
     */
    interface Entry {
        /** ... */
        String getKey();
        /** ... */
        Object getValue();
        /** ... */
        void setValue(Object o);
        /** ... */
        boolean isEmpty();
    }
}
```

# Implementazione

## Interfaccia Dizionario

### continua interfaccia Dizionario

```
/** Tests if this map has no keys.  
*  
* @return true if this map contains no key-value  
* mappings, false otherwise.  
*/  
boolean isEmpty();  
/**  
* Returns the value to which the specified key is  
* mapped in this map.  
*  
* @param key a key in the map.  
* @return the value to which the key is mapped in  
* this map; null otherwise.  
* @exception NullPointerException if the key is null.  
*/  
Object get(String key);
```

# Implementazione

## Interfaccia Dizionario

### continua interfaccia Dizionario

```

/**
 * Associates the specified key to the specified
 * value in this map. Neither the key nor
 * the value can be null.
 *
 * @param key the key.
 * @param value the value.
 * @return the previous value of the specified key in
 * this map, or null if it did not have one.
 * @exception NullPointerException if the key or value
 * is null.
 * @see Object#equals(Object)
 */
Object put(String key, Object value);

```

# Implementazione

## Interfaccia Dizionario

### continua interfaccia Dizionario

```
/**  
 * Removes the key (and its corresponding value) from  
 * this map. This method does nothing if the key is  
 * not present.  
 *  
 * @param key the key that needs to be removed.  
 * @return the value to which the key had been mapped  
 *         in this map, or null if the key did not  
 *         have a mapping.  
 */  
Object remove(String key);  
}
```



# Implementazione

## Hash Table con chaining

### Sintesi Hash Table con chaining

```
public class HashChain implements Map {
    /** Hash table data */
    private Entry table[];

    /** Total number of entries in the hash table */
    private int count;

    /** The table is rehashed when its size exceeds
     * this threshold. (The value of this field is
     * (int)(capacity * loadFactor).)
     */
    private int threshold;

    /** The load factor for the hashtable. */
    private float loadFactor;
}
```



# Implementazione

## Hash Table con chaining

### Sintesi Hash Table con chaining: la classe interna

```
static public class Entry implements Map.Entry {
    /** ... */
    Object value;
    /** ... */
    final String key;
    /**
     * This implementation adopts the chaining addressing.
     * next represents the next element with
     * the same hash code of the key.
     */
    Entry next;

    ...metodi di Entry...
}
```

# Implementazione

## Hash Table con chaining

### Sintesi Hash Table con chaining: metodo get

```
/** ... */
public Object get(String key) {
    //Check parameter
    if (key == null)
        throw new NullPointerException(
            "key cannot be null!");

    for (Entry e=table[hash(key)]; e!=null;
         e=e.next) {
        if (e.key.equals(key)) return e.value;
    }
    return null;
}
```



# Implementazione

## Hash Table con chaining

### Sintesi Hash Table con chaining: metodo put

```
if (count >= threshold) {
    // Rehash the table if the threshold is exceeded
    rehash();
    tab = table;
    index = hash(key);
}
// Insert the new entry.
Entry e = new Entry(key, value, tab[index]);
tab[index] = e;
count++;
return null;
}
```

# Implementazione

## Hash Table con chaining

### Sintesi Hash Table con chaining: metodo rehash

```
protected void rehash() {
    int oldCapacity = table.length;
    Entry oldMap[] = table;

    int newCapacity = oldCapacity * 2 + 1;
    Entry newMap[] = new Entry[newCapacity];

    threshold = (int) (newCapacity*loadFactor);
    table = newMap;

    // Rehash all entries
    int index;

    ...
}
```



# Implementazione

## Hash Table con chaining

### Sintesi Hash Table con chaining: metodo rehash

```
for (int i = oldCapacity; i-- > 0;) {
    for (Entry old = oldMap[i]; old!=null; ) {
        Entry e = old;
        old = old.next;
        index = hash(e.key);
        e.next = newMap[index];
        newMap[index] = e;
    }
}
```

# Librerie standard

- `java.util.Dictionary`, interfaccia (**deprecata**)
- `java.util.Hashtable` è un'implementazione (**deprecata**)
- `java.util.Map` è una nuova interfaccia
- `java.util.HashMap` è un'implementazione di `Map`
- `java.util.HashSet` è un'implementazione dell'ADT *set* con chiavi hash

# Esercizi

## Esercizio 1

- 1 Completare la classe `HashChain`. Ci sono circa 13 metodi da aggiungere. Si raccomanda il Javadoc!
- 2 Implementare `Map` con una classe (`HashOpen`) che usi l'open addressing con doppio hashing. Attenzione al metodo `remove()`!
- 3 Eseguire il seguente test:
  - Fissare  $m = 7$ ;
  - inserire ("ad", "AAA"), ("ba", "BBB"), ("c", "CCC"), ("e", "EEE"), ("bc", "BCB")
  - cancellare "ad"
  - stampare il contenuto finale della tabella della classe `HashOpen`. Per ogni cella della tabella deve essere stampata se è vuota o no e, in caso di presenza di un elemento, (*key*, *code(key)*, *hash(key)*, *value*).