Massimo Benerecetti

Tabelle Hash

Lezione n.#

Parole chiave:

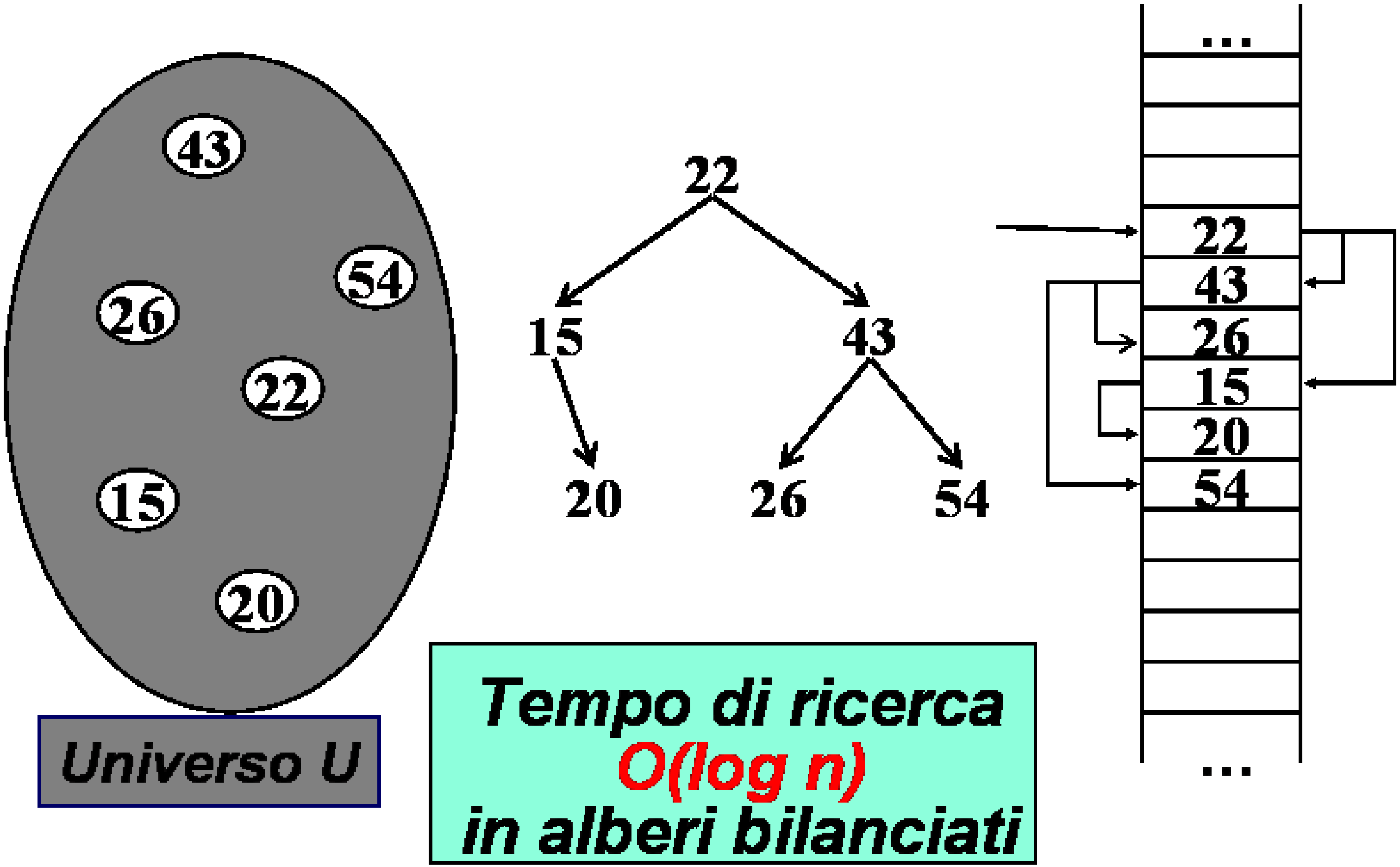
Corso di Laurea:
Informatica

Insegnamento:
Algoritmi e
Strutture Dati I

Email Docente:
bene@na.infn.it

A.A. 2009-2010

- Gli insiemi dinamici possono essere rappresentati con varie strutture dati, ciascuna con caratteristiche di flessibilità e di prestazioni differenti.
- Array, liste ed alberi sono tra le rappresentazioni più diffuse.
- Gli alberi binari di ricerca bilanciati offrono un buon compromesso tra flessibilità e prestazioni, garantendo tempi di ricerca logaritmici rispetto al numero di elementi.
- Rinunciando ad un po’ della flessibilità degli alberi, è possibile però ottenere strutture dati con migliori prestazioni per la ricerca degli elementi.

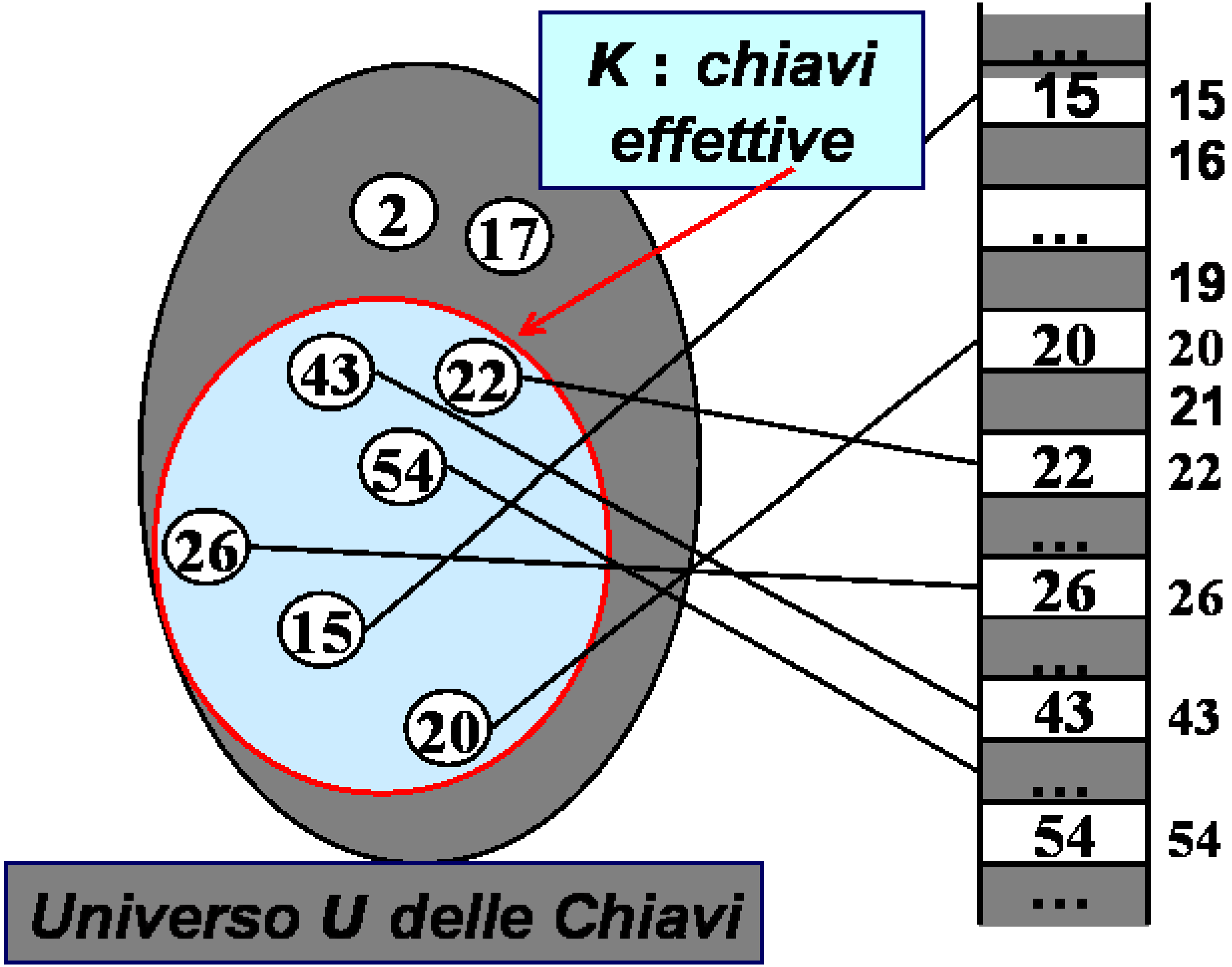


Rappresentazione ad ***albero*** di un insieme dinamico di chiavi prese da un universo **U**.

- Una ***tabella ad accesso diretto*** è una struttura dati che *suppotra SOLO* le operazioni di:
  - inserimento
  - ricerca
  - cancellazione

- in tempo che è ***O(1)***

- *Non supporta direttamente Minimo, Massimo, Successore, Predecessore (cioè gli Ordinamenti)*



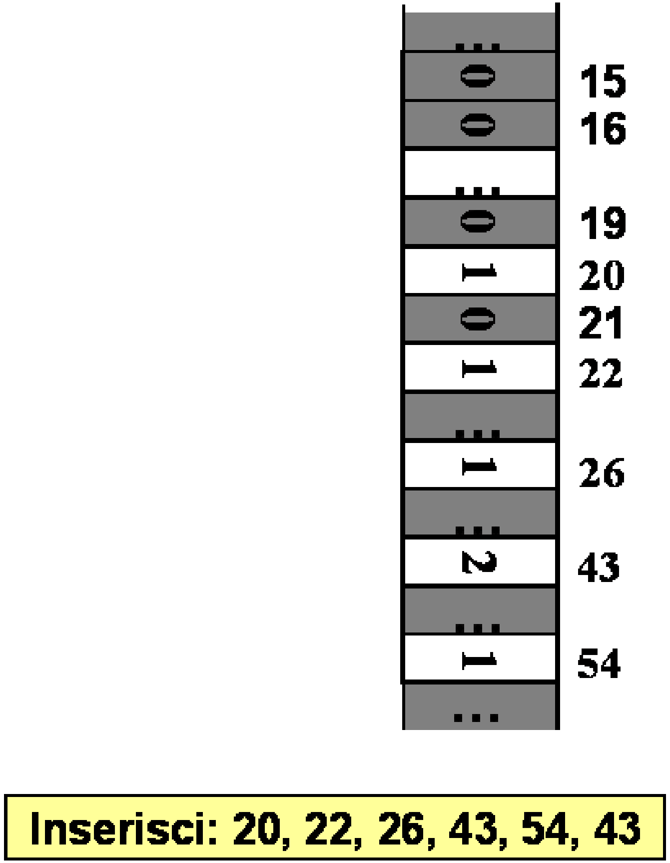
Rappresentazione con ***tabella ad accesso diretto*** per un insieme dinamico di chiavi prese da un universo **U**.

- La più *semplice* implementazione di una ***tabella ad accesso diretto*** è un ***array***.
- Per memorizzare gli interi a 16-bit possiamo utilizzare un array ***A*** di dimensione 2^{16}.
- Le operazioni potrebbero essere definite come segue:

– **inserisci(i): A[i] = A[i] + 1**

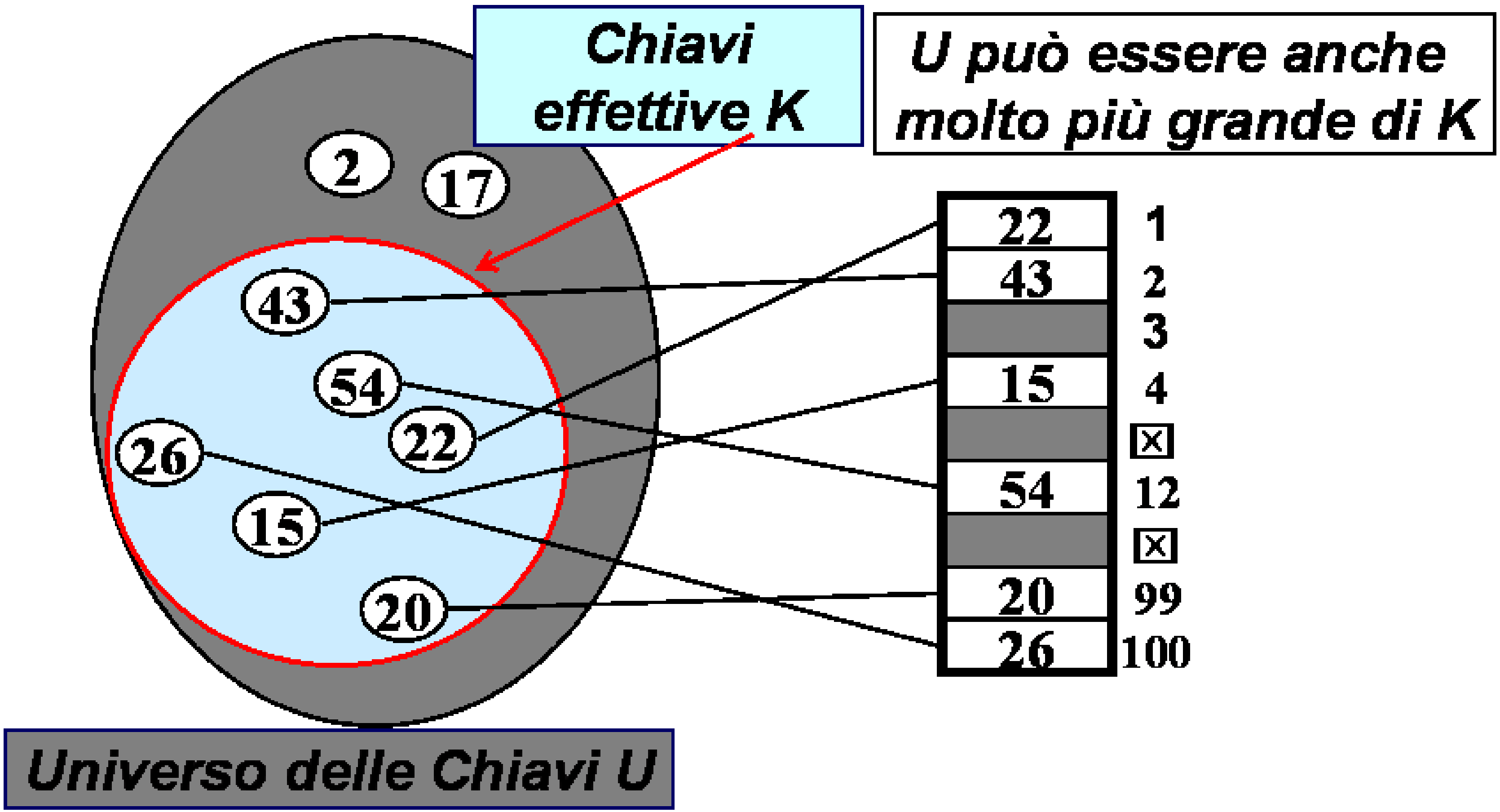
– **ricerca(i) : (A[i] > 0)?**

– **cancella(i) : A[i] = A[i] - 1**



Esempio di funzione indice e tabella ad accesso diretto.

- Se le ***chiavi*** sono ***stringhe di 8 lettere*** alfabetiche, ci sono **26^8** (o circa 200 miliadri) di possibili chiavi [circa 200 'giga' di chiavi].
- Quasi sempre solo una ***piccola frazione*** di queste chiavi verrà effetticamente ***impiegata***.
- Ne risulterebbe la necessità di un ***array molto grande***, ma con ***pochissime celle occupate***.
- Ci serve, quindi, una ***soluzione migliore!***



Rappresentazione di una ***tabella hash*** per un insieme dinamico di chiavi prese da un universo **U**.

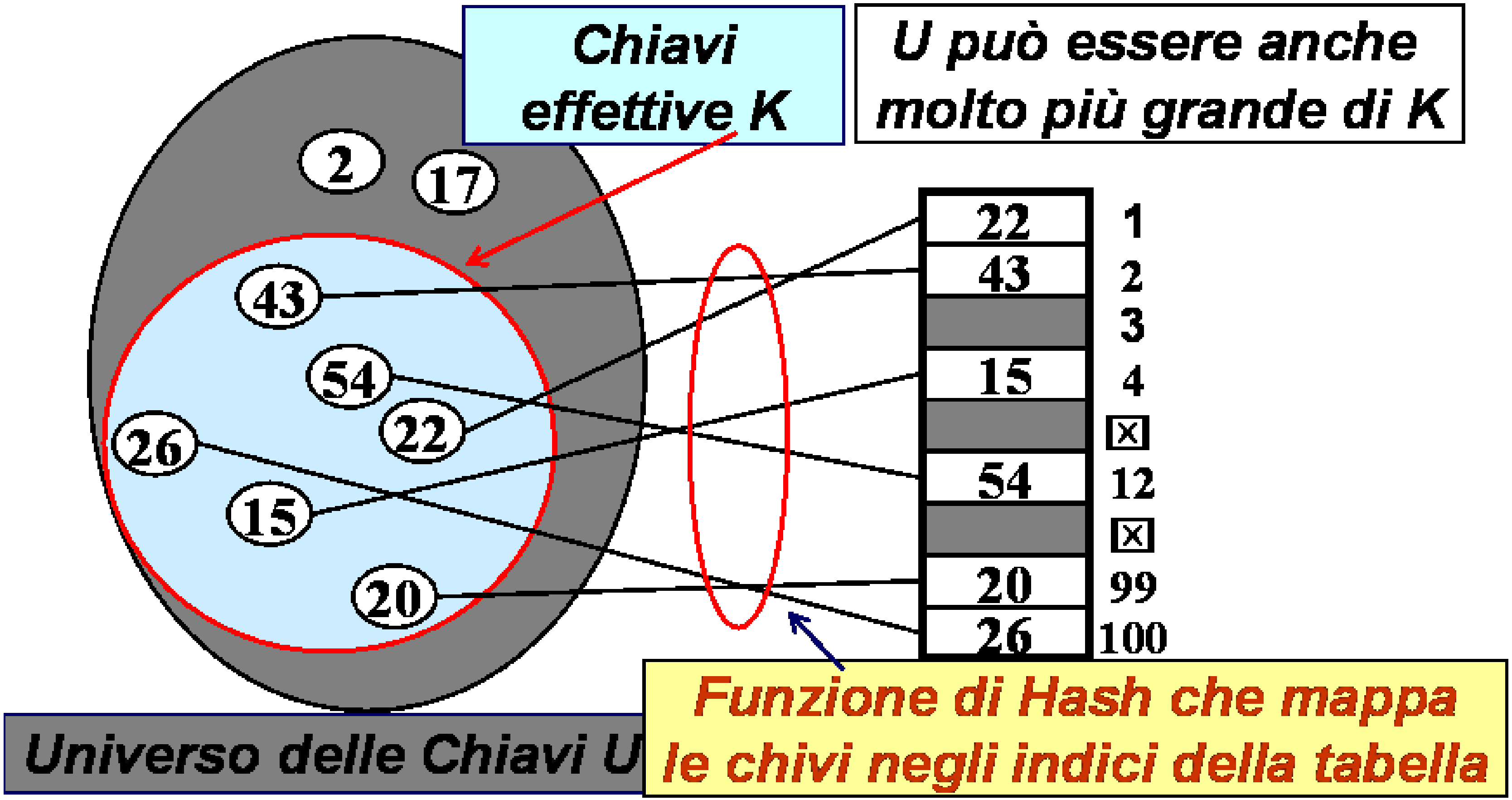


Illustrazione di ina ***tabella hash*** e della ***funzione di hashing***.

- Uno ***schema di hashing*** consiste di una ***tabella ad accesso diretto*** (la **tabella hash**) e di una ***funzione di hashing*** con dominio l'universo delle chiavi e codominio l'insieme degli indici della tabella.

- Una ***funzione hash*** prende in input una chiave e la "***mappa***" su qualche ***indice*** all'interno della tabella.

- Uno ***schema di hashing*** ammette che differenti chiavi possibili vengano "***mappate***" nella stessa locazione (la ***funzione di hash*** **NON** è ***iniettiva***). Quando ciò avviene, si parla di ***collisione*** tra chiavi.

- È quindi necessario definire dei meccanismi opportuni per la gestione delle ***collisioni***.

Data una funzione di hash ***hash(key)*** che ritorna un intero, l'approccio semplicistico potrebbe essere il seguente

- **inserisci(key): A[*hash*(*key*)] = oggetto da inserire**
- **ricerca(key) : l'oggetto è presente in A[*hash*(*key*)]?**
- **cancella(key) : A[*hash*(*key*)] = NULL**

Nella definizione di una ***tabella hash*** adeguata alle necessità applicative si deve scegliere:

- una opportuna ***funzione di hash*** che abbia buone proprietà di distribuzione uniforme delle chiavi sugli indici
- la ***dimensione della tabella*** che spesso dipende dal tipo di funzione hash scelto
- la *politica* di ***gestione*** *e soluzione* delle ***collisioni***

TSIZE = 10

0	10, 100
1	
2	2
3	
4	
5	5
6	
7	
8	18
9	9

hash(k) = k mod TSIZE

Semplice finzione di Hash su interi:

TSIZE: dimesione della tabella

mod: operazione di modulo

Esempio di tabella hash e funzione hash.

TSIZE = 10

0	**10, 100**
1	
2	**2, 12, 22**
3	
4	
5	**5, 15, 25**
6	
7	
8	**18**
9	**9**

hash(k) = k mod TSIZE

Le collisioni sono molto frequenti.

La dimensione della tabella è inadeguata

Si noti che **10 = 2·5**, e i multipli di **2** (pari) possono solo finire nelle **celle di indice pari**, mentre multipli di **5** solo nelle celle di **indice 0** e **5**.

hash(k) = k mod TSIZE

TSIZE = 10

0	10, 100
1	
2	2, 12, 22
3	
4	
5	5, 15, 25
6	
7	
8	18
9	9

TSIZE = 11

0	22
1	12, 100
2	2
3	25
4	15
5	5
6	
7	7, 18
8	
9	9
10	10

Si noti che **11** è primo e ora i multipli di **2** e di **5** coprono tutte le celle della tabella (minor possibilità di collisioni).

La ***dimensione della tabella*** *può* influire sulla ***frequenza delle collisioni***

- **TSIZE** Numero Composto
  - **10: 2*5**
  - **300: 2*2*3*5*5**
  - ***Scarsa uniformità*** della distribuzione della chiavi che determina *maggiori possibilità di collisioni*
- **TSIZE** Numero Primo
  - **11**
  - **10007**
  - ***Maggiore uniformità*** della distribuzione delle chiavi porta a *minori possibilità di collisioni*

Una proprietà importante di un meccanismo di hashing è quella di

- ***Hashing Uniforme Semplice***: ogni chiave ha la stessa probabilità di essere mappata in una delle ***n*** celle della tabella, indipendentemente dalla cella in cui è mappata ogni altra chiave.

Proprietà desiderabili di una "**buona**" ***funzione di hash*** sono:

- ***efficienza*** e facilità di calcolo
- ***distribuzione uniforme*** delle chiavi sul dominio degli indici
- ***minimizzazione*** delle ***collisioni***

1. ***Metodo della Divisione***

   Convertire la chiave in un intero e calcolare il modulo (**mod**) rispetto alla dimensione della tabella

2. ***Metodo de Moltiplicazione***

   Convertire la chiave in un intero utilizzando operazioni di moltiplicazione

3. ***Troncamento***

   Ignorare parte della chiave e usare la porzione che rimane come indice

4. ***Folding***

   Partizionare la chiave in parti differenti e combinare queste parti in modo da ottenere l'indice

Metodo della divisione

– e.g. Semplicemente calcolare il ***modulo*** rispetto alla dimensione della tabella:

$$\textbf{\textit{hash}}(62538194) = 62538194 \textbf{ mod } 1000 = 194$$

- Il metodo è sensibile al valore scelto per la dimensione della tabella.

– es. l'uso di una potenza di due per la dimensione può causare scarsa uniformità della funzione. Se $n = 2^p$, allora vengono considerati solo i p bit meno significativi della chiave.

- Nel caso dell'aritmetica modulare, la migliore scelta della ***dimensione della tabella hash*** è un ***numero primo***.
- Nel caso sopra, tabelle di dimensione 997 o 1009 (entrambi numeri primi) darebbero migliori prestazioni dal punto di vista della distribuzione delle chiavi sugli indici.

Metodo della motiplicazione

- Prima si moltiplica la chiave $\boldsymbol{k}$ per una costante $\boldsymbol{a}$ nell'intervallo $0 < \boldsymbol{a} < 1$.
- Poi si estrae la parte frazionaria del prodotto $\boldsymbol{k \cdot a}$, cioè il valore

$$\boldsymbol{k \cdot a} - \lfloor \boldsymbol{k \cdot a} \rfloor$$

- Infine si moltiplica il valore ottenuto per la dimensione $\boldsymbol{n}$ della tabella.

$$hash(\boldsymbol{k}) = \lfloor \boldsymbol{n} \cdot (\boldsymbol{k \cdot a} - \lfloor \boldsymbol{k \cdot a} \rfloor) \rfloor$$

- Chiaramente, $0 \leq (\boldsymbol{k \cdot a} - \lfloor \boldsymbol{k \cdot a} \rfloor) < 1$ e, quindi, $0 \leq hash(\boldsymbol{k}) < n$.
- Questo metodo non è particolarmente sensibile al valore $\boldsymbol{n}$ della dimensione della tabella.

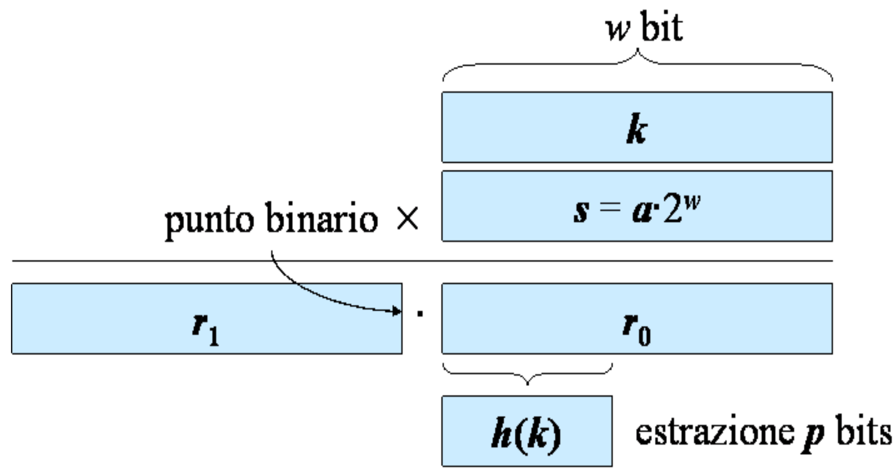
Il metodo della moltiplicazione può essere implementato facilmente e in ***modo efficiente***, scegliendo il valore n come una potenza di **2**.

- Sia $n = 2^p$ per qualche intero p.
- Sia w la dimensione della parola della macchina (numero di bit in una parola).
- Scegliamo $a = s/2^w$, con $0 < s < 2^w$ intero.
- Moltiplicando k per $s = a \cdot 2^w$. Il risultato sarà un valore di $2 \cdot w$ bit, della forma

$$k \cdot s = r_1 \cdot 2^w + r_0$$

dove r_1 è la parte più significativa del prodotto, e r_0 quella meno significativa.

- Il valore *hash*(k) è rappresentato dai p bit più significativi di r_0.



Implementazione del metodo della moltiplicazione.

Si noti che

$$k \cdot a - \lfloor k \cdot a \rfloor = frac\left(\frac{k \cdot s}{2^w}\right) = r_0$$

E che, posto $n = 2^p$, si ottiene

$$\lfloor n \cdot (k \cdot a - \lfloor k \cdot a \rfloor) \rfloor = \lfloor n \cdot r_0 \rfloor = 2^p \cdot r_0$$

Cioè i p bit più significativi di r_0.

Dati interi di 8 cifre e una tabella di dimensione 1000

Troncamento

– es. — usare congiuntamente solo la 4ª, 7ª e 8ª cifra per formare l’indice: *hash*(62538194) = 394

Folding

– es. — suddividere ogni chiave in gruppi di 3, 3, e 2 cifre, sommare le parti e troncatare se necessario:

hash(62538194) **=** (625+381+94) **mod** 1000 **=**

= 1100 **mod** 1000 **=** 100

Funzioni hash per chiavi a stringa

Definire una funzione che trasforma una stringa in un intero.

- Ad esempio, sommando il valore ASCII di tutti i caratteri:
  - ad esempio $\boldsymbol{hash1(K) = \sum_i K[i]}$
  - oppure, meglio, $\boldsymbol{hash2(K) = \sum_i d^i \cdot K[n-i]}$ (per ***d*** intero > 1)
- Problema ***hash1***(): quando le chiavi sono corte e la tabella è grande
  1. 8 caratteri, TSIZE = 10007, ma 8 · 256 = 2048. La tabella può, quindi, risultare sproporzionata, determinando spreco di spazio.
  2. tutte le permutazioni della stessa stringa collidono allo stesso valore hash.

Una possibile soluzione è quella di usare solo alcuni caratteri e ***moltiplicare tra loro i valori dei caratteri*** (*metodo del troncamento*)

- :
  - **C** a p o **V** e **r** d e
  - Numero di possibili valori di indice: 27*27*27 **=** 17576 > 10007
  - Può essere necessario, quindi, integrarlo con il *metodo della divisione*.
- Problema: le lingue non sono casuali
  - molte meno combinazioni effettivamente possibili di quelle permesse
  - rischio di spazio sprecato

Le seguenti funzioni di hash pesano diversamente ciascun carattere della stringa e impiegano il metodo della divisione (ipotizzando 27 diversi caratteri alfabetici) :

1. $hash_1(K)$ $= (\ldots + 27^2\, K[2] + 27\, K[1] + K[0] \ldots) \bmod TSIZE$
   $= ((\ldots + K[2])*27 + K[1])*27 + K[0])\ldots) \bmod TSIZE$

2. **$hash_2(K) = ((\ldots + K[2])*32 + K[1])*32 + K[0])\ldots) \bmod TSIZE$**

L'algoritmo sotto riportato calcola la seconda funzione di hash nell'esempio:

```
Hash2(K[])
  i = 1
  WHILE (K[i] ≠ '\0') DO
     hash  = (shift(hash,5)) + K[i] /* hash = hash * 2^5 + key[i] */
     i  = i + 1
  return (hash mod TSIZE)
```

Si noti che l'espressione **shift(*hash*,5)** corrisponde alla moltiplicazione del valore contenuto in ***hash*** per **32** (cioè per **2^5**).

$$hash_3(0) = 5381$$

$$hash_3(i) = hash_3(i - 1) * 33 + K[i]$$

```
Hash3(K[])
    hash = 5381
    i = 1
    WHILE (K[i] ≠ '\0') DO
        hash = ((shift(hash,5)) + hash) + K[i]
        i = i + 1
    return (hash mod TSIZE)
```

Questa funzione di hash ha mostrato prestazioni di uniformità particolarmente buone in pratica.

Un'altra possibilità è ***usare il folding***: elaborare la stringa 4 byte alla vota, convertendo ogni gruppo di 4 byte in un intero, usando uno dei metodi sopra descritti. I valori interi di ogni gruppo vengono poi sommati tra di loro. Infine, si converte il risultato in un intero tra 0 e TSIZE tramite operazione di modulo.

- Semplice somma dei valori numerici dei caratteri della stringa
  - Molto semplice da implementare.
  - Può impiegare molto tempo se le chiavi sono lunghe.
  - I primi caratteri possono non venir considerati.
  - Posso essere spostati (`shift`) fuori dal range.
- Una possibile soluzione può essere quella di adottare una variante del folding:
  - usare solo alcuni caratteri
  - e.g. Via Cintia 345, Napoli, I-81100
- Un'altra soluzione può essere quella utilizzare il metodo della divisione insieme alla somma (pesata) dei diversi caratteri.