Tabelle Hash

Pietro Di Lena

Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna

Algoritmi e Strutture di Dati
Anno Accademico 2021/2022

Introduzione

- Molte applicazioni richiedono una struttura dati di tipo Dizionario che supporti in maniera estremamente efficiente unicamente le operazioni basilari SEARCH, INSERT, DELETE
  - Esempio: i compilatori utilizzano un Dizionario per memorizzare ed etichettare gli identificatori (chiavi) nel programma
- La Tabella Hash implementa efficientemente la struttura dati Dizionario
  - Idea: generalizzare l'indicizzazione in un array ordinario
- Per quanto le operazioni su una Tabella Hash possano avere un costo pessimo lineare, in media le prestazioni computazionali sono efficienti
  - Sotto ragionevoli assunzioni probabilistiche le operazioni SEARCH, INSERT, DELETE hanno un costo medio $O(1)$

Nozioni preliminari

- Indichiamo con
  - U = Universo di tutte le chiavi possibili
  - K = Insieme di tutte le chiavi effettivamente utilizzate
- Scelte implementative: dipendono dal dominio di applicazione
- Esempi:
  - $U = \{0, 1, \cdots, m-1\}$ con m *piccolo*, $|K| \sim |U|$
    - Usiamo tabelle ad indirizzamento diretto
  - U è un insieme generico molto grande, $|K| \ll |U|$
    - Usiamo Tabelle Hash

Tabelle ad indirizzamento diretto

- Implementazione basata su array T di dimensione $|U|$
- La chiave k è memorizzata nella posizione k dell'array
- Ricordiamo che tutte le chiavi sono distinte

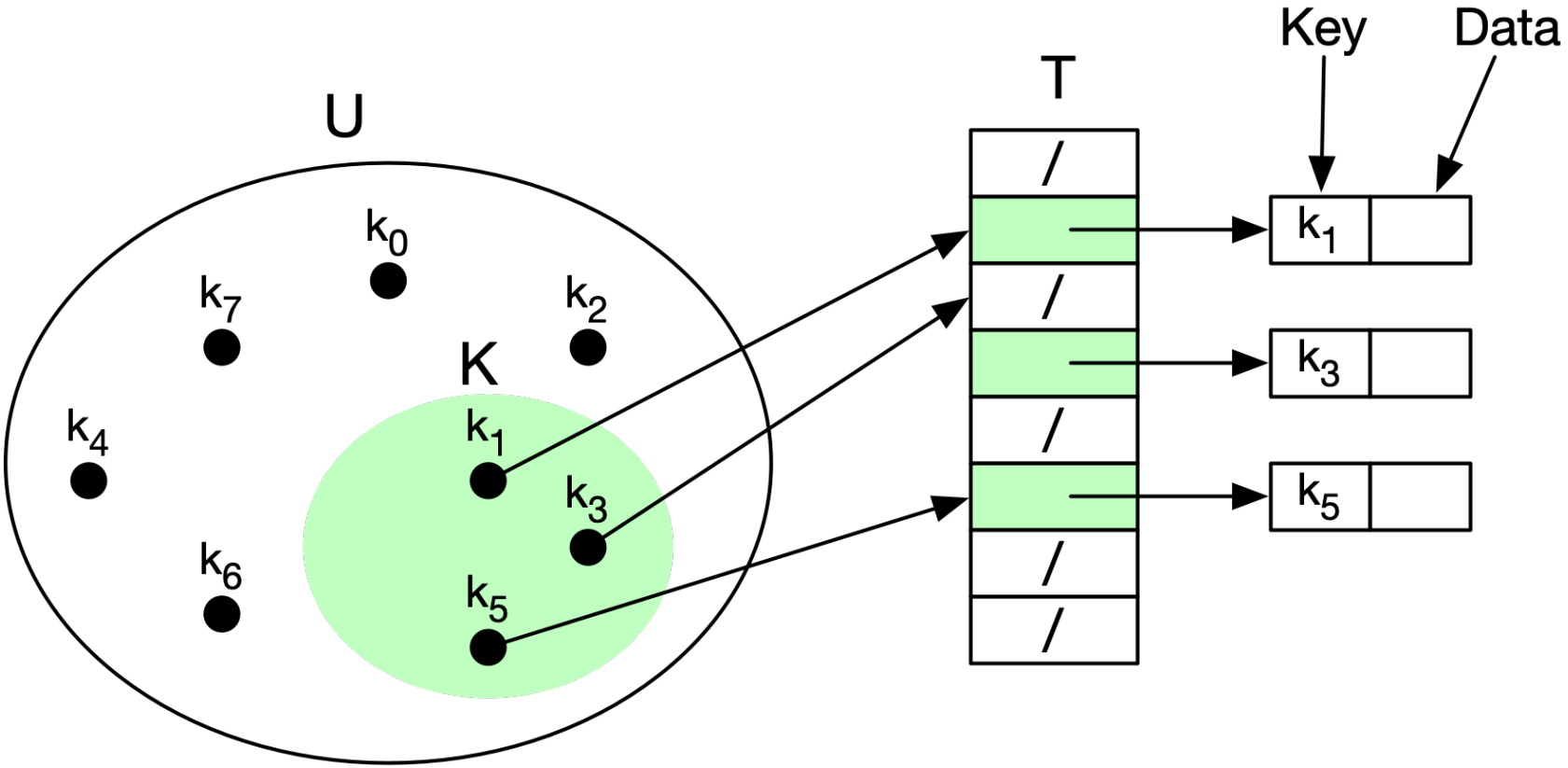


Tabelle ad indirizzamento diretto

```
1: function SEARCH(HASHTAB T, KEY k) → DATA
2:     return T[k].data
3:
4: function INSERT(HASHTAB T, KEY k, DATA d)
5:     T[k] = new NODE(k, d)
6:
7: function DELETE(HASHTAB T, KEY k)
8:     ERASE(T[k])
9:     T[k] = NIL
```

- Costo computazionale in termini di tempo: $O(1)$
- Costo computazionale in termini di memoria: $\Theta(|T|) = \Theta(|U|)$
  - Se $|K| \sim |U|$ soluzione accettabile
  - Se $|K| \ll |U|$ soluzione non accettabile
    - Esempio: $U =$ identificatori lunghi massimo 20 caratteri

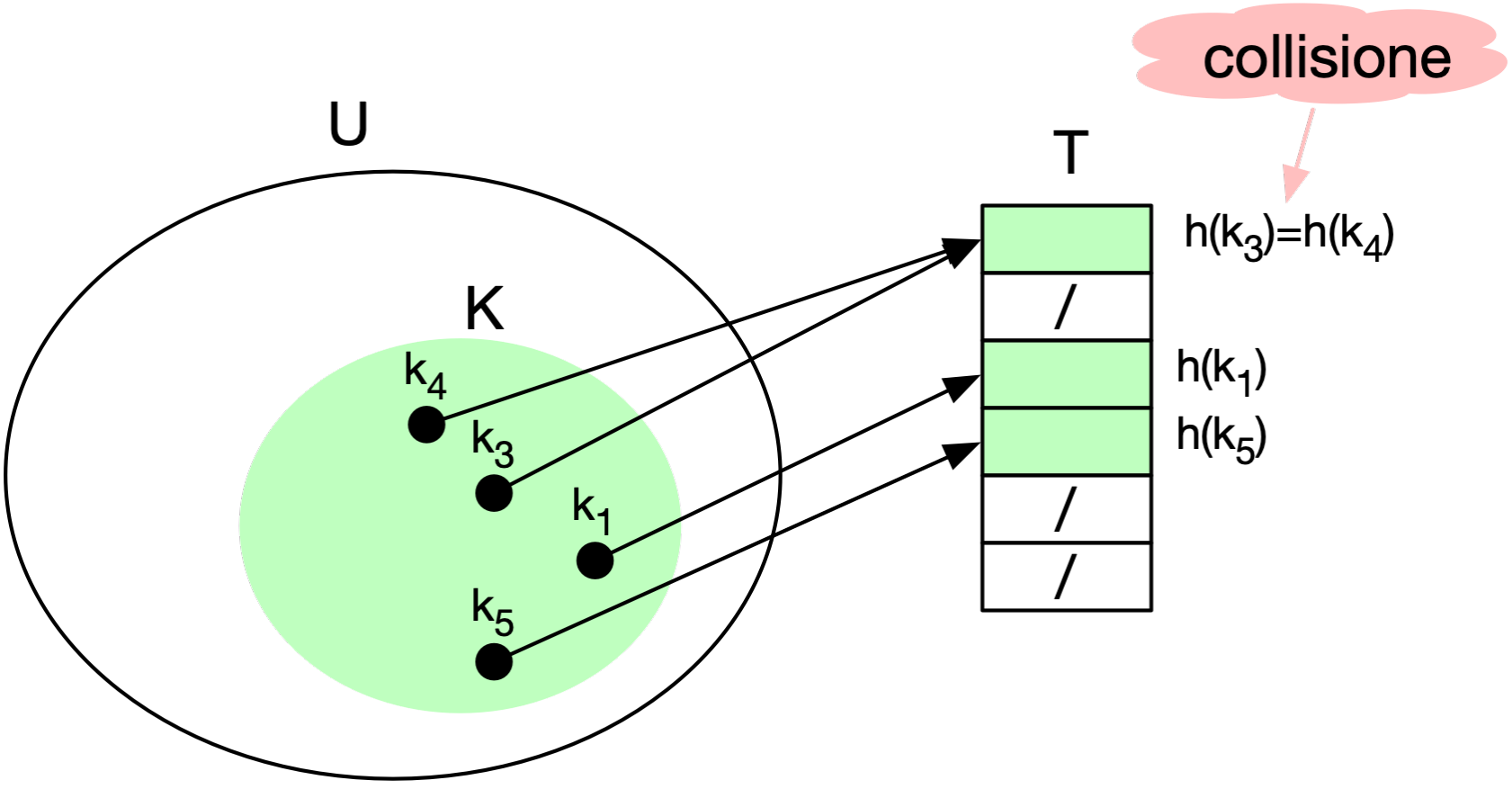
$$|U| = 26 * (26 + 10)^{19} \approx 10^{31} \Rightarrow |T| \approx 10^{31} * 4\text{bytes} > 10^{19}\text{Terabytes}$$

Tabelle Hash

- Un array di dimensione $\Theta(|U|)$ richiede troppa memoria se U è grande
- Generalmente l'insieme di chiavi K è molto più piccolo rispetto ad U
- Soluzione: Tabelle Hash
  - Usiamo un array $T[0, \cdots, m-1]$ di dimensione $m = \Theta(|K|)$
  - Usiamo una funzione hash $h : U \to [0, \cdots, m-1]$
- Indirizzamento hash
  - Diciamo che $h(k)$ è il valore hash della chiave k
  - La funzione h trasforma una chiave k in un indice dell'array T
  - La chiave k viene *mappata* nello slot $T[h(k)]$
  - Se due chiavi hanno lo stesso valore hash abbiamo una collisione
- Problema: evitare e gestire le collisioni hash
  - Idealmente vorremmo funzioni hash che evitino sempre collisioni
  - Non possiamo evitarle, dobbiamo almeno minimizzarle

Tabelle Hash

- Implementazione basata su array T di dimensione $\Theta(|K|)$
- La chiave k e i dati sono memorizzati nella posizione $h(k)$ dell'array
- Evitare le collisioni è impossibile anche con buone funzioni hash



Ricapitolando

- Per implementare una Tabella Hash efficiente abbiamo bisogno di
  1. Una funzione hash
     - Deve poter essere calcolata velocemente
     - Deve garantire una buona distribuzione delle chiavi su T
     - Una buona distribuzione minimizza il rischio di collisioni
  2. Un metodo per gestire le collisioni
     - Le collisioni sono inevitabili
     - Quando non riusciamo ad evitarle dobbiamo gestirle
  3. Un vettore $T[0, \cdots, m-1]$ di dimensione $m = \Theta(|K|)$
     - Normalmente possiamo solo stimare m
     - Non sappiamo a priori quante chiavi andremo a memorizzare
     - Potrebbe essere necessario ridimensionare T
     - La scelta migliore per la dimensione m dipende dalla funzione hash e dal metodo utilizzato per gestire le collisioni

Funzioni hash

- Una buona funzione hash soddisfa (approssimativamente) la proprietà di uniformità semplice (*hashing uniforme semplice*)
  - Una funzione hash h deve distribuire uniformemente le chiavi negli indici $[0, \cdots, m-1]$ della tabella T
  - Ogni indice $i = h(k)$ deve essere generato con probabilità $1/m$
  - Se alcuni indici in $[0, \cdots, m-1]$ sono *scelti* con maggiore probabilità da h allora avremo un numero maggiore di collisioni
- Per soddisfare la proprietà di uniformità semplice bisogna conoscere la distribuzione di probabilità con cui le chiavi sono *estratte* da U
  - Conoscere tale distribuzione di probabilità è spesso irrealistico
  - Solo in casi specifici tale distribuzione è nota
  - Esempio: le chiavi k sono estratte a caso dall'intervallo $U = [0, 1)$ (tutte le chiavi in U sono equiprobabili). Allora

    $$h(k) = \lfloor mk \rfloor$$

    soddisfa la proprietà di uniformità semplice

Funzioni hash: assunzioni

1. Tutte le chiavi sono equiprobabili
   - Tutte le chiavi hanno la stessa probabilità di essere estratte da U
   - Non è sempre vero (es. identificatori in un programma)
   - Semplificazione necessaria per proporre un meccanismo generale
2. La funzione hash può essere calcolata in tempo $O(1)$
   - Una codifica hash non $O(1)$ domina il costo delle operazioni
   - Es. costa più calcolare il valore hash che effettuare una ricerca
   - In realtà, ci accontentiamo di hashing sufficientemente veloci
3. Tutte le chiavi sono valori interi non-negativi
   - E' sempre possibile trasformare una qualsiasi chiave k in un intero
   - Es. numero decimale ottenuto dalla rappresentazione binaria di k

Esempio: da stringa ad intero positivo

- Vogliamo trasformare una chiave di tipo *stringa* in intero
- Idea: trasformiamo i caratteri in un codice binario
- Assumiamo di associare i seguenti codici alle lettere dell'alfabeto

$$a = 1, b = 2, c = 3, d = 4, e = 5 \cdots, r = 18, \cdots z = 26$$

in binario sono sufficienti 6 bit per carattere

- Codifica ottenuta concatenando i codici binari

$$\text{bin}(\text{"beer"}) = 000010\ 000101\ 000101\ 010010$$

che rappresenta il numero 545.106 in base 10

Funzione hash: metodo della divisione

- Metodo della divisione: $h(k) = k \mod m$
- Esempi:
  - Se $m = 12, k = 100 \Rightarrow h(k) = 4$
  - Se $m = 10, k = 101 \Rightarrow h(k) = 1$
- Vantaggi:
  - Molto efficiente (richiede solo una divisione intera)
- Svantaggi: tante collisioni, ma inevitabile
  - Suscettibile a specifici valori di m (potrebbe non usare tutto k)
  - Esempio 1: se $m = 10$ allora $h(k) =$ ultima cifra di k
  - Esempio 2: se $m = 2^p$ allora $h(k)$ dipende unicamente dai p bit meno significativi di k e non da tutti i bit di k
  - Soluzione: scegliere m come numero primo distante da potenze di 2 (e di 10)

Funzione hash: metodo della moltiplicazione

- Metodo della moltiplicazione: $h(k) = \lfloor m(kC - \lfloor kC \rfloor) \rfloor$
  - Sia C una costante $0 < C < 1$
  - Moltiplichiamo k per C e prendiamo la parte frazionaria
  - Moltiplichiamo quest'ultima per m e prendiamo la parte intera
- Esempi:
  - Se $m = 12, k = 101, C = 0.8 \Rightarrow h(k) = 9$
  - Se $m = 1000, k = 124, C = (\sqrt{5} - 1)/2 \approx 0.618 \Rightarrow h(k) = 18$
- Svantaggi:
  - La costante C influenza la proprietà di uniformità di h
  - $C = (\sqrt{5} - 1)/2$ suggerito da Knuth (*The Art of Computer Programming, Vol 3*)
- Vantaggi:
  - Il valore di m non è critico

Funzione hash: metodo della codifica algebrica

- Metodo della codifica algebrica:

$$h(k) = (k_n x^n + k_{n-1} x^{n-1} + \cdots + k_1 x + k_0) \mod m$$

  dove
  - $k = k_n k_{n-1} \cdots k_1 k_0$ e k_i è l'i-esimo bit della rappresentazione binaria di k, oppure l'i-esima cifra della rappresentazione decimale di k, o anche il codice ascii dell'i-esimo carattere
  - x è un valore costante
- Esempio: usando la rappresentazione decimale
  - Se $m = 12, k = 234, x = 3 \Rightarrow h(k) = (2 \times 3^2 + 3 \times 3 + 4) \mod 12 = 7$
- Vantaggi:
  - Dipende da tutti i bit/caratteri della chiave
- Svantaggi:
  - Costoso da calcolare
  - Richiede n addizioni e $n * (n+1)/2$ prodotti ($n = \Theta(\log k)$)

Regola di Horner

- Valutazione di un polinomio in un punto
- Un polinomio di grado n

  $$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

  può essere riscritto nel seguente modo

  $$p(x) = a_0 + x(a_1 + x(a_2 + x(\cdots x(a_{n-1} + a_n x))))$$

  che richiede n addizioni ed n moltiplicazioni
- La regola di Horner permette di abbassare il costo (da quadratico a lineare) del calcolo della funzione hash basata sul metodo della codifica algebrica

java.lang.String.hashCode()

```
/**
* Returns a hash code for this string. The hash code for a String
* object is computed as
*
* s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
*
* using int arithmetic, where s[i] is the ith character of the
* string, n is the length of the string, and ^ indicates
* exponentiation. (The hash value of the empty string is zero.)
*/
public int hashCode() {
        int h = hash;
        if (h == 0 && value.length > 0) {
            char val[] = value;

            for (int i = 0; i < value.length; i++)
                h = 31 * h + val[i];
            hash = h;
        }
        return h;
}
```

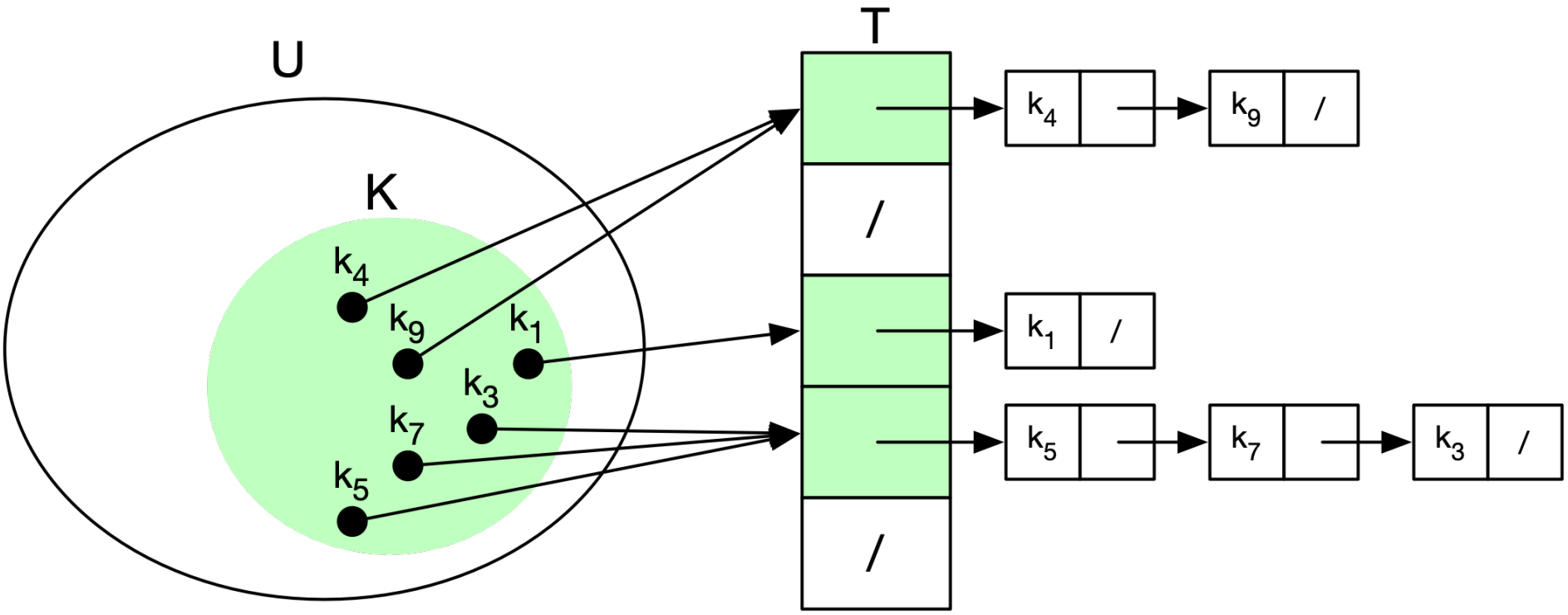
- Funzione hash di libreria Java della classe String
- Basata sul metodo della codifica algebrica
  - Utilizza i codici ascii dei caratteri
  - Calcolata con il metodo di Horner
  - La costante x è il numero primo 31

Problema delle collisioni

- Hashing uniforme semplice riduce ma non elimina le collisioni
- Anche assumendo hashing uniforme semplice, la probabilità che ci sia collisione tra due chiavi è (sorprendentemente) molto alta
  - Problema del compleanno: date n persone scelte a caso qual è la probabilità che due tra esse compiano gli anni nello stesso giorno?
  - Paradosso del compleanno: in un gruppo di 23 persone tale probabilità è maggiore del 50%
- Come gestire le eventuali collisioni?
  - Dobbiamo trovare collocazioni alternative per le chiavi
  - Se una chiave non si trova nella posizione attesa, bisogna andare a cercare nelle posizioni alternative
  - Le operazioni diventano costose nel caso pessimo
- Vediamo due possibili tecniche
  - Concatenamento (o anche scansione esterna)
  - Indirizzamento aperto (o anche scansione interna)

Risoluzione collisioni: concatenamento

- Concatenamento (chaining)
  - Le chiavi k con lo stesso valore hash $h(k) = i$ sono memorizzate in una lista concatenata (lista di trabocco)
  - Lo slot $T[i]$ contiene il puntatore alla testa della lista contenente tutte le chiavi k con hash $h(k) = i$



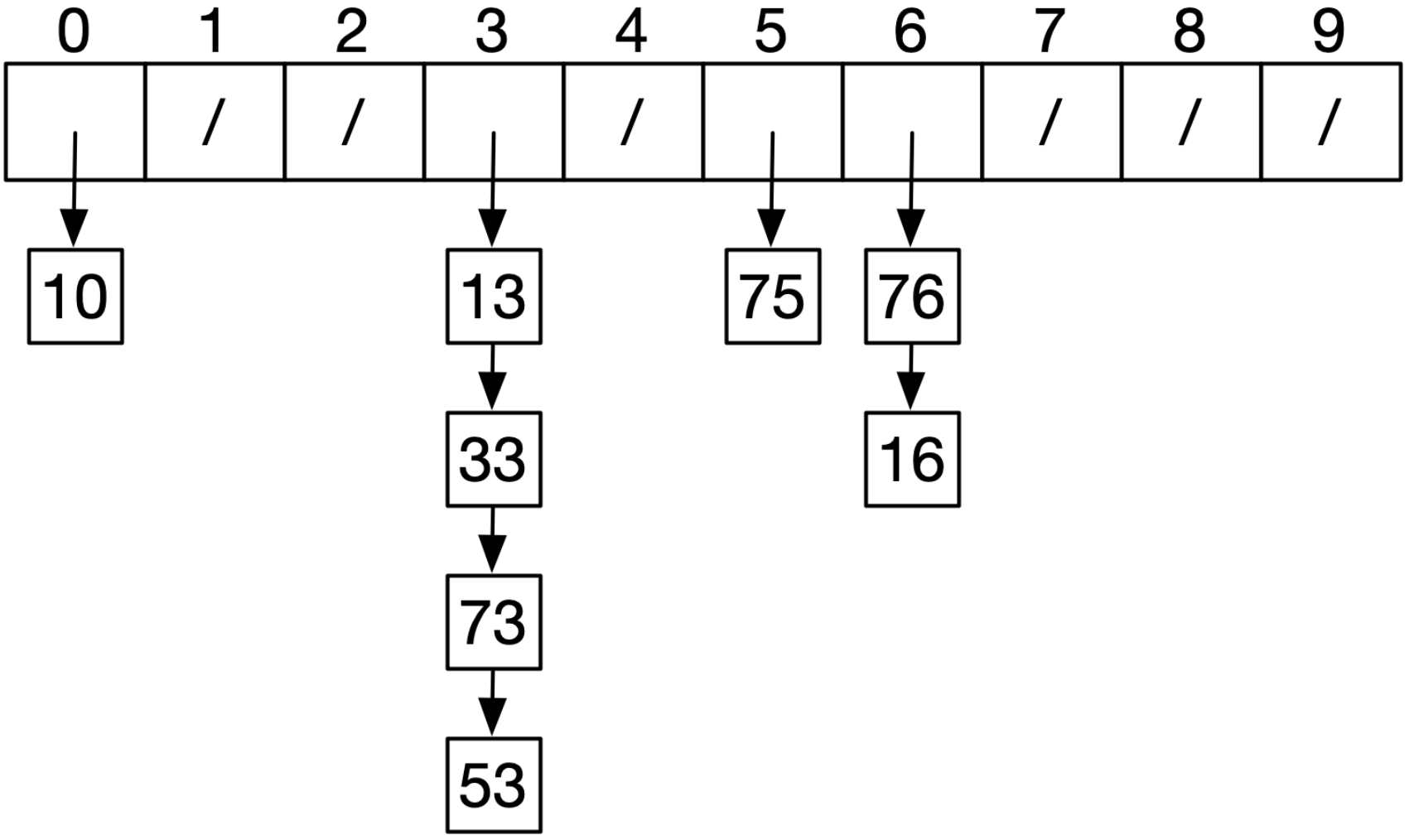
Pseudocodice: concatenamento

```
 1: function SEARCH(HASHTAB T, KEY k) → DATA
 2:     tmp =LINKED-LIST-SEARCH(T[h(k)], k)
 3:     if tmp ≠ NIL then
 4:         return tmp.data
 5:     else
 6:         return NIL
 7:
 8: function INSERT(HASHTAB T, KEY k, DATA d)
 9:     LINKED-LIST-INSERT(T[h(k)], k, d)
10:
11: function DELETE(HASHTAB T, KEY k)
12:     T[h(k)] = LINKED-LIST-DELETE(T[h(k)], k)
```

- SEARCH esegue una ricerca lineare su lista concatenata
- INSERT esegue un inserimento in testa in una lista concatenata
- DELETE esegue una rimozione su una lista concatenata

Esempio: concatenamento

- Funzione hash $h(k) = k \mod 10$
- Inserimenti nel seguente ordine: $53, 75, 16, 73, 10, 33, 13, 76$



Analisi del metodo di concatenamento

- Dimensione della tabella
  - Non impone vincoli sulla dimensione del vettore $T[0, \cdots, m-1]$
    - Vincoli eventualmente imposti dalla funzione hash
  - Se m troppo grande, rischio di sprecare spazio
  - Se m troppo piccolo, liste di collisione lunghe $\Rightarrow$ nel caso pessimo operazione di ricerca di una chiave ha un costo lineare sulla dimensione della lista
- Quanto costano le operazioni SEARCH, INSERT, DELETE?
  - Sia L la lunghezza della lista di collisione più lunga
  - SEARCH: costo nel caso pessimo $O(L)$, caso ottimo $O(1)$
  - INSERT: costo nel caso pessimo e ottimo $O(1)$
  - DELETE: costo nel caso pessimo $O(L)$, caso ottimo $O(1)$
  - N.B. $L = O(n)$, dove $n =$ numero di elementi nella tabella
  - N.B. Il costo pessimo non dipende da m ma dal numero di elementi n
  - Riusciamo a analizzare il caso medio?

Concatenamento: analisi del caso medio

- Il costo nel caso medio dipende dal numero medio di accessi per cercare (con successo o insuccesso) una chiave
  - Il costo della ricerca di una chiave incide su SEARCH e DELETE
  - Il numero medio di accessi dipende da come vengono distribuite le chiavi dalla funzione hash

- Chiamamo fattore di carico $\alpha = m/n$ il rapporto tra il numero di elementi e la dimensione di una Tabella Hash
  - $n =$ numero di elementi nella Tabella Hash
  - $m =$ numero di slot nella Tabella Hash

- Sotto l'assunzione di hashing uniforme semplice ogni slot della tabella ha mediamente α chiavi
  - Ricordiamo che hashing uniforme semplice $\Rightarrow$ la funzione hash distribuisce le chiavi uniformemente in $T[0, \cdots, m-1]$)

Analisi del caso medio: ricerca con insuccesso

Teorema

Sotto l'assunzione di hashing uniforme semplice, una ricerca senza successo in una tabella hash con concatenamento ha costo medio $\Theta(1+\alpha)$

- Dimostrazione
  - Sotto l'assunzione di hashing uniforme semplice, data una chiave k non presente nella tabella, gli m slot di T sono tutti ugualmente probabili per la codifica hash $h(k)$
  - Se k non compare nella tabella (ricerca con insuccesso), la ricerca visita tutte le chiavi nella lista $T[h(k)]$, che ha in media α chiavi
  - Costo medio: tempo di hashing $h(k)$ (costo medio 1) + tempo di visita della lista $T[h(k)]$ (costo medio α) $\Rightarrow \Theta(1+\alpha)$

Analisi del caso medio: ricerca con successo

Teorema

Sotto l'assunzione di hashing uniforme semplice, una ricerca con successo in una tabella hash con concatenamento ha costo medio $\Theta(1+\alpha)$

- Dimostrazione
  - Sotto l'assunzione di hashing uniforme semplice, data una chiave k non presente nella tabella, gli m slot di T sono tutti ugualmente probabili per la codifica hash $h(k)$
  - Se k compare nella tabella (con insuccesso), la ricerca visita in media all'incirca metà delle chiavi nella lista $T[h(k)]$ (l'analisi completa è più complessa), che ha in media α chiavi
  - Costo medio: tempo di hashing $h(k)$ (costo medio 1) + tempo di visita della lista $T[h(k)]$ (costo medio $\alpha/2$) $\Rightarrow$

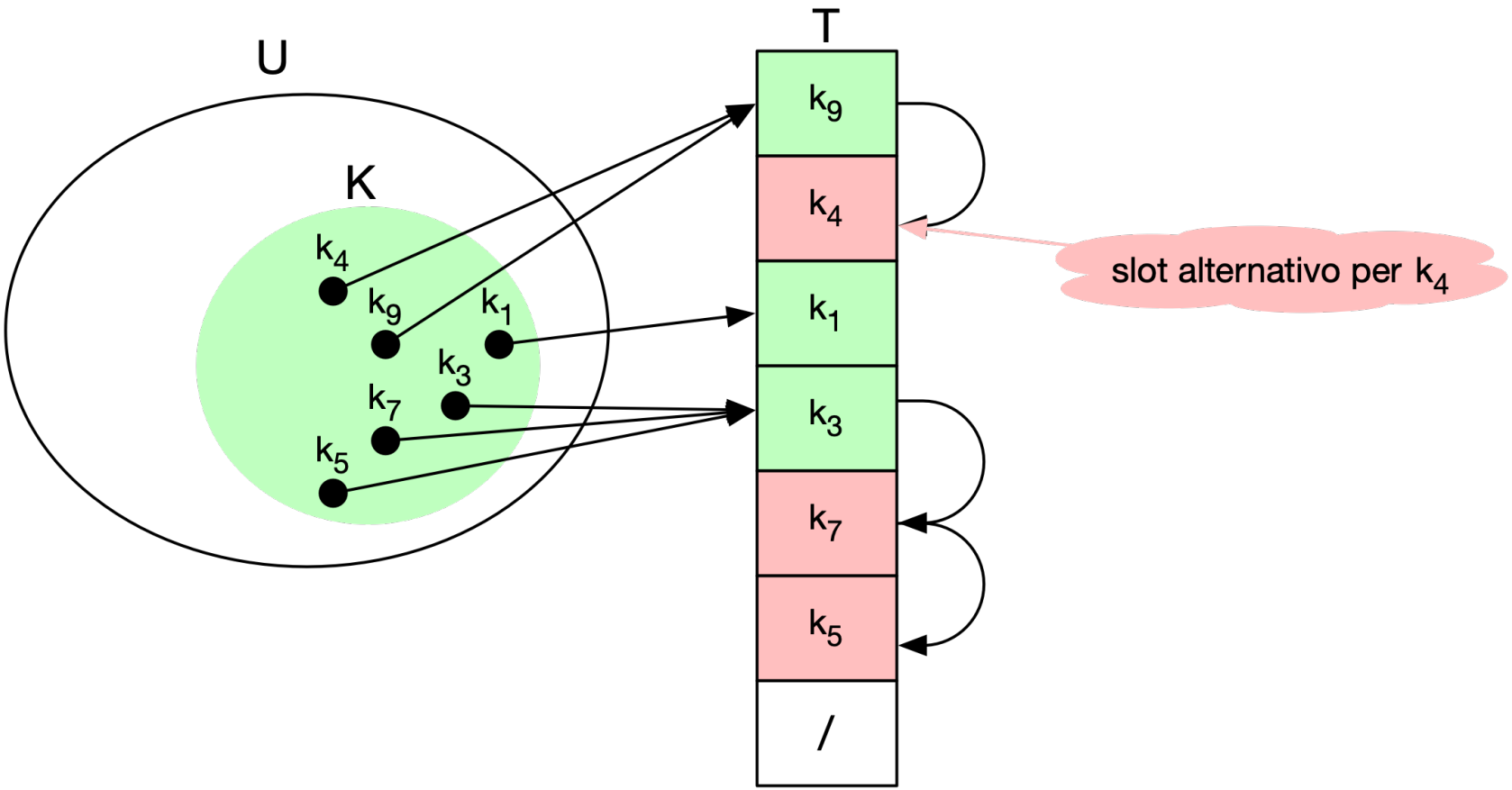
$$\Theta(1+\alpha/2) = \Theta(1+\alpha)$$

Riassumendo: analisi del caso medio

- Abbiamo dimostrato che su una Tabella Hash in cui le collisioni siano risolte con concatenamento, sotto l'assunzione di hashing uniforme semplice, la ricerca ha un costo medio $\Theta(1+\alpha)$
  - $n =$ numero di elementi nella tabella
  - $m =$ numero di slot nella tabella
  - fattore di carico $\alpha = n/m$
- Il fattore di carico influenza quindi il costo delle operazioni
  - Se $n = O(m)$ allora $\alpha = O(1) \Rightarrow$ costo medio della ricerca $O(1)$
  - Quindi SEARCH, INSERT, DELETE hanno costo medio $O(1)$

Risoluzione collisioni: indirizzamento aperto

- Indirizzamento aperto (open addressing)
  - Tutte le chiavi sono memorizzate nella stessa tabella
  - Ogni slot contiene una chiave oppure NIL
  - Se uno slot è occupato, se ne cerca uno alternativo nella tabella



Indirizzamento aperto: ispezioni

- Idea: data una chiave k, se uno slot $T[h(k)]$ è già occupato allora ispezioniamo la tabella alla ricerca di uno slot libero

- Per determinare quale slot ispezionare estendiamo la funziona hash in modo che abbia come parametro anche il passo di ispezione

$$h : U \times [0, \cdots, m-1] \rightarrow [0, \cdots, m-1]$$

- La sequenza di ispezione

$$h(k, 0), h(k, 1), \cdots, h(k, m-1)$$

deve fornire una permutazione degli indici della tabella

  - Vogliamo visitare ogni slot solo una volta
  - Potrebbe essere necessario visitare tutti gli m slot

Pseudocodice: indirizzamento aperto

```
1: function SEARCH(HASHTAB T, KEY k) → DATA
2:     i = 0
3:     repeat
4:         j = h(k, i)                    ▷ hash value at step i
5:         if A[j].key == k then
6:             return A[j].data
7:         i = i + 1
8:     until A[j] == NIL or i == A.size
9:     return NIL
```

```
1: function INSERT(HASHTAB T, KEY k, DATA d)
2:     i = 0
3:     repeat
4:         j = h(k, i)                    ▷ hash value at step i
5:         if A[j] == NIL then
6:             A[j] = (k, d)
7:             return
8:         i = i + 1
9:     until i == A.size
10:    error "overflow"
```

Attenzione: non funziona correttamente

Pseudocodice delete: indirizzamento aperto

- Non possiamo sostituire la chiave che vogliamo cancellare con NIL
  - SEARCH si ferma se trova NIL mentre la chiave cercata potrebbe essere presente e verrebbe trovata nelle ispezioni successive
- Soluzione: utilizziamo il valore DELETED invece di NIL per marcare uno slot vuoto dopo la cancellazione
  - SEARCH/DELETE: DELETED trattati come slot pieni
  - INSERT: DELETED trattati come slot vuoti

```
1:  function DELETE(HashTab T, Key k)
2:      i = 0
3:      repeat
4:          j = h(k, i)                    ▷ hash value at step i
5:          if A[j].key == k then
6:              ERASE(A[j])
7:              A[j] = DELETED
8:              return
9:          i = i + 1
10:     until A[j] == NIL or  i == A.size
```

Pseudocodice: indirizzamento aperto

```
1: function SEARCH(HASHTAB T, KEY k) → DATA
2:     i = 0
3:     repeat
4:         j = h(k, i)                          ▷ hash value at step i
5:         if A[j].key == k then
6:             return A[j].data
7:         i = i + 1
8:     until A[j] == NIL or i == A.size
9:     return NIL
```

```
1: function INSERT(HASHTAB T, KEY k, DATA d)
2:     i = 0
3:     repeat
4:         j = h(k, i)                          ▷ hash value at step i
5:         if A[j] == NIL or A[j] == DELETED then
6:             A[j] = (k, d)
7:             return
8:         i = i + 1
9:     until i == A.size
10:    error "overflow"
```

Analisi del metodo di indirizzamento aperto

- Nel caso pessimo SEARCH, INSERT, DELETE costano $O(m)$
  - $m =$ dimensione della tabella
  - Nel caso pessimo ispezioniamo l'intera tabella
- Quanto costano le operazioni nel caso medio?
  - Costo medio influenzato dalla strategia di ispezione
  - Anche sotto l'assunzione di hashing uniforme semplice
- Vediamo tre strategie di ispezione
  - ispezione lineare
  - ispezione quadratica
  - doppio hashing

Strategie di ispezione: ispezione lineare

- Funzione di ispezione (m = dimensione della tabella)

$$h(k, i) = (h'(k) + i) \mod m$$

  dove $h'(k)$ è una funzione hash ausiliaria

- Quando si ha una collisione, si ispeziona l'indice successivo
  - Il primo indice $h'(k)$ determina l'intera sequenza

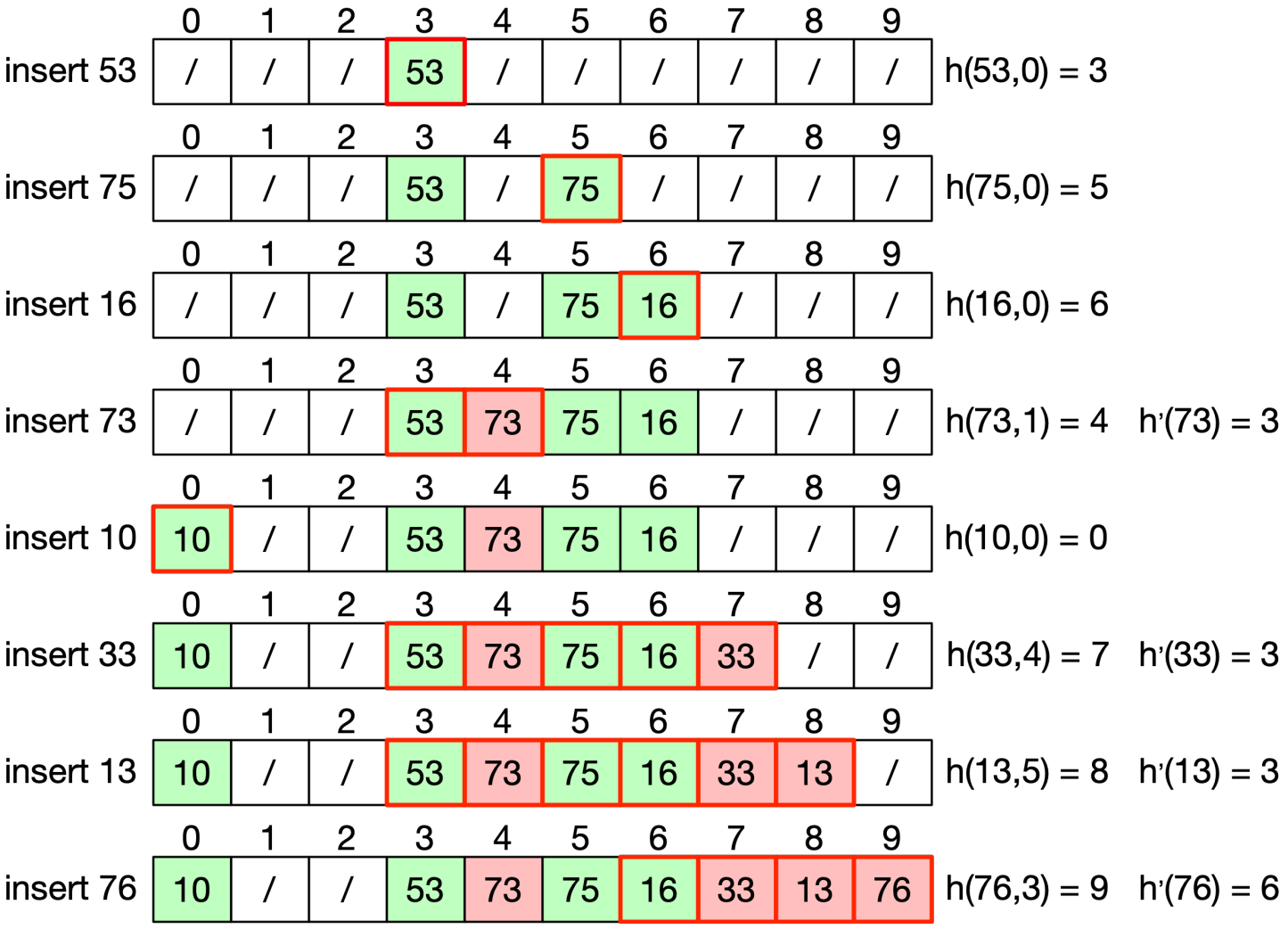
$$h'(k), h'(k) + 1, \cdots, m - 1, 0, 1, \cdots h'(k) - 1$$

  - Sono possibili solo m sequenze distinte di ispezione
  - Ogni slot è ispezionato una sola volta

- Problema: clustering primario
  - Lunghe sotto-sequenze occupate, che diventano sempre più lunghe
  - Assumendo hashing uniforme semplice, uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$
  - I tempi medi di inserimento e cancellazione crescono

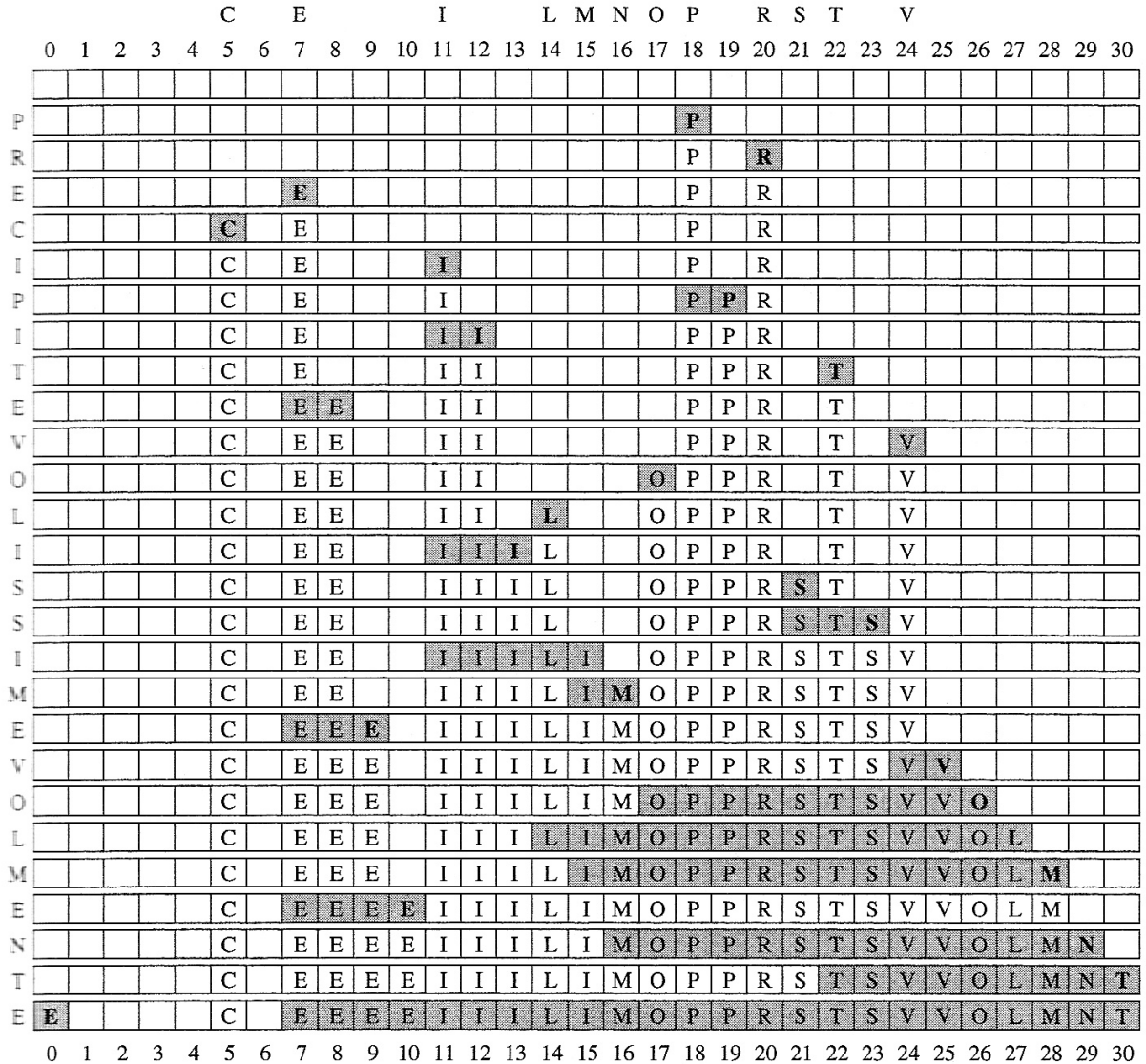
Esempio: ispezione lineare

- Funzione hash $h(k,i) = (h'(k) + i) \mod 10$ dove $h'(k) = k \mod 10$
- Inserimenti nel seguente ordine: $53, 75, 16, 73, 10, 33, 13, 76$



Esempio: ispezione lineare

$h(k,i) = (h'(k)+i) \mod 31$ dove $h'(k) = ascii(k) \mod 31$ $(ascii(A)=65)$



Strategie di ispezione: ispezione quadratica

- Funzione di ispezione (m = dimensione della tabella)

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m \quad \text{(con costanti } c_1 \neq c_2\text{)}$$

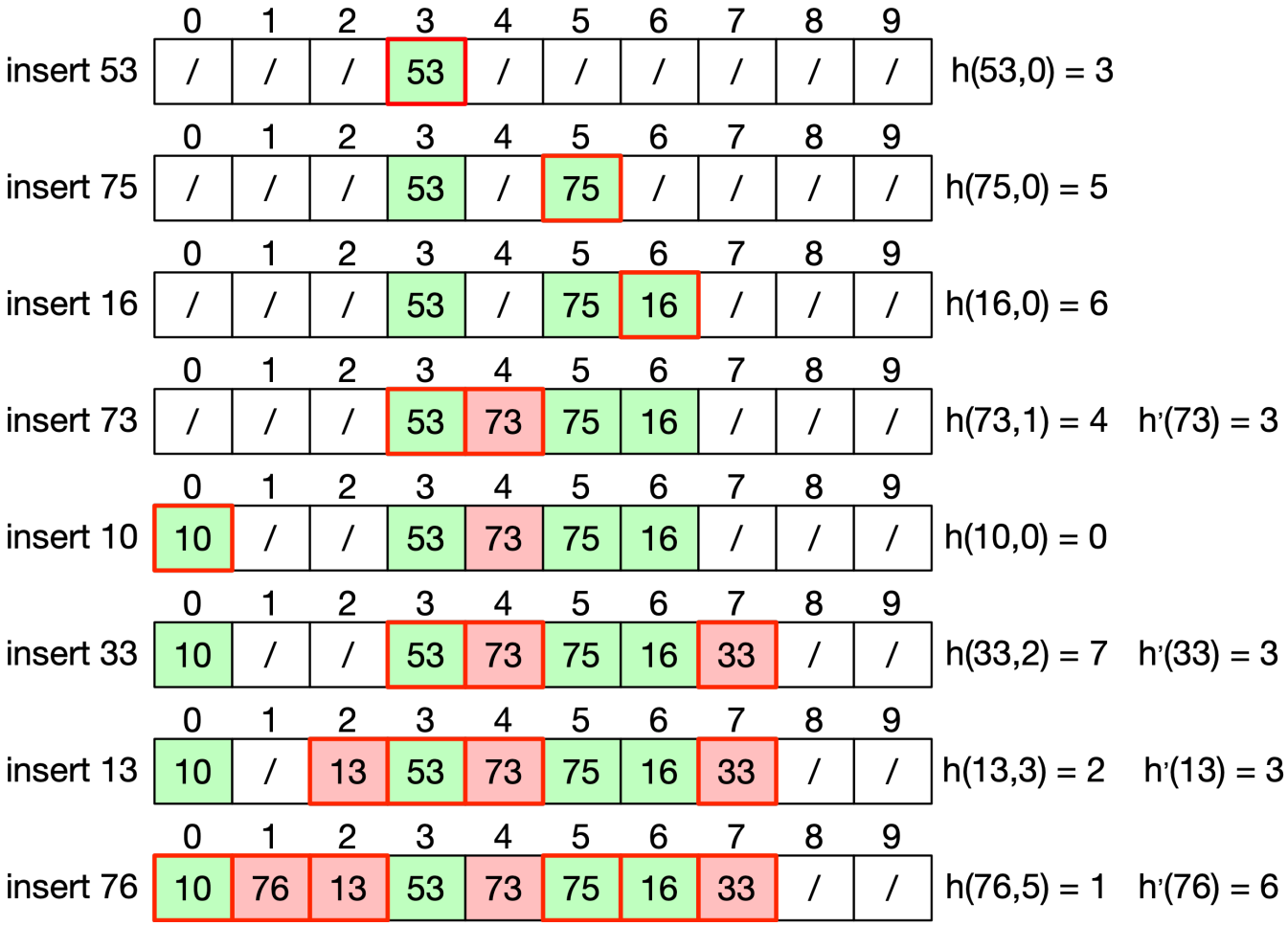
  dove $h'(k)$ è una funzione hash ausiliaria

- Quando si ha una collisione, si usa un passo quadratico
  - Il primo indice $h'(k)$ determina l'intera sequenza
  - Le ispezione successive hanno un offset che dipende da una funzione quadratica nel numero di ispezione i
  - Sono possibili solo m sequenze distinte di ispezione
  - c_1, c_2 devono garantire una permutazione di $[0, \cdots, m-1]$

- Problema: clustering secondario
  - Se due chiavi hanno la stessa ispezione iniziale, allora le loro sequenze di ispezione sono identiche

Esempio: ispezione quadratica

- Funzione hash $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod 10$ dove
  $h'(k) = k \mod 10$ e $c_1 = 0, c_2 = 1$
- Inserimenti nel seguente ordine: $53, 75, 16, 73, 10, 33, 13, 76$



Strategie di ispezione: doppio hashing

- Funzione di ispezione (m = dimensione della tabella)

$$h(k, i) = (h_1(k) + ih_2(k)) \mod m$$

  dove $h_1(k)$ e $h_2(k)$ sono la funzione hash primaria e secondaria

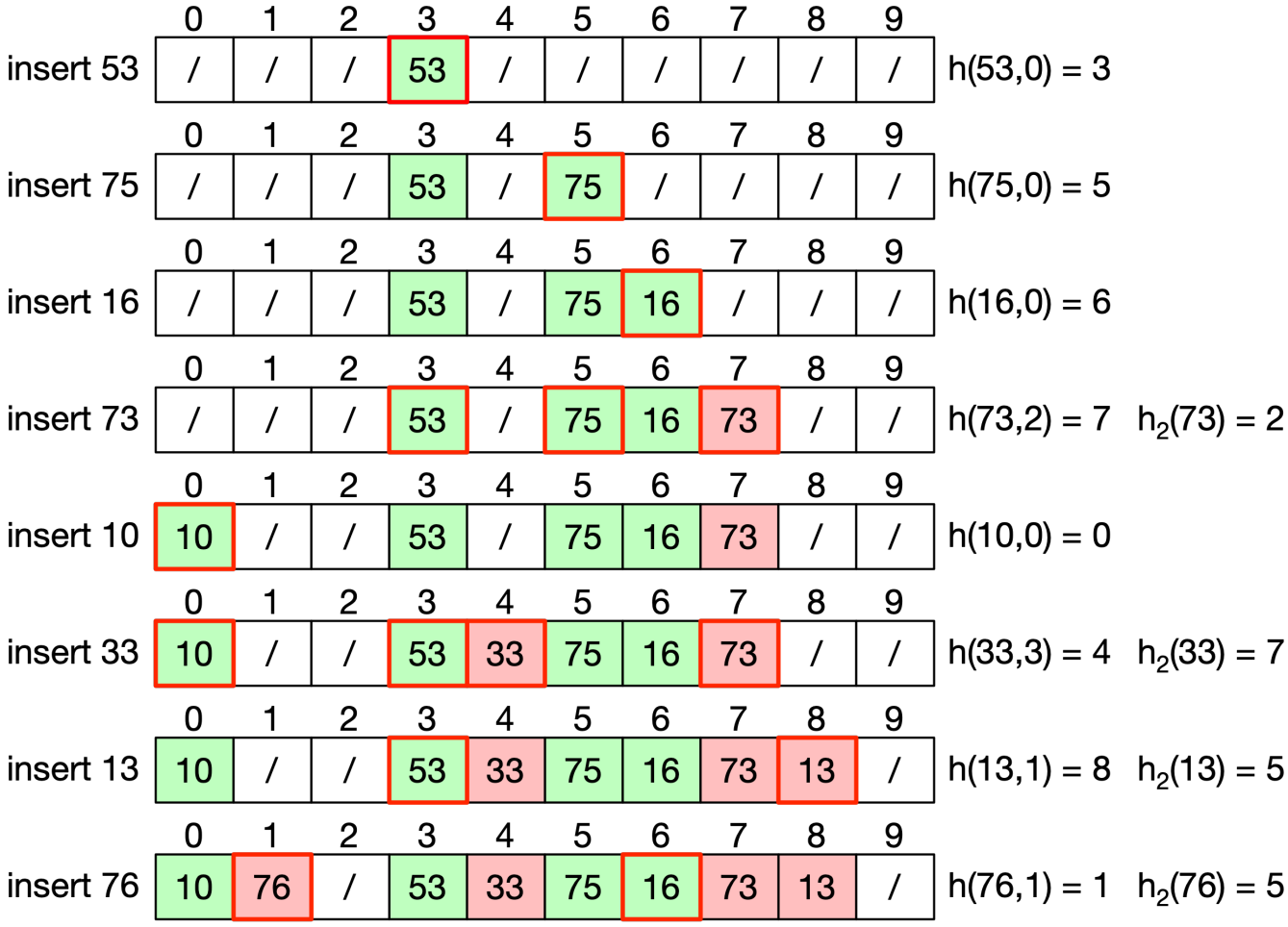
- Quando si ha una collisione, si usa la funzione secondaria e l'indice di ispezione per determinare il successivo slot da ispezionare
  - Evita il clustering primario e secondario
  - Se $h_1 \neq h_2$ è meno probabile che per una coppia di chiavi $a \neq b$

$$h_1(a) = h_1(b) \text{ e } h_2(a) = h_2(b)$$

- Vincoli sulla funzione hash secondaria h_2
  - Non deve mai dare il valore hash 0
  - Deve permettere di iterare su tutta la tabella

Esempio: doppio hashing

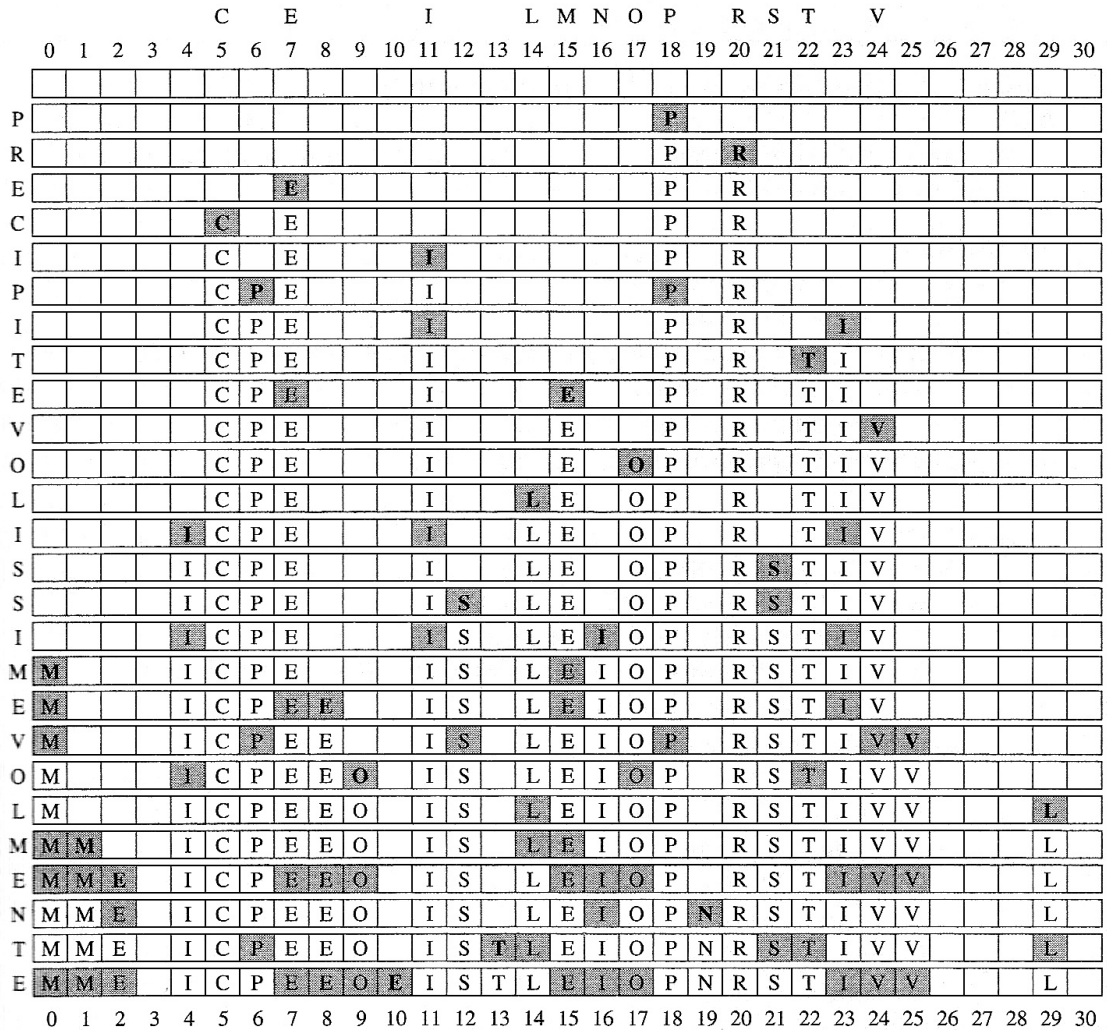
- Funzione hash $h(k,i) = (h_1(k) + ih_2(k)) \mod 10$ dove
  $h_1(k) = k \mod 10$ e $h_2(k) = (k \mod 9) + 1$
- Inserimenti nel seguente ordine: $53, 75, 16, 73, 10, 33, 13, 76$



Esempio: doppio hashing

$h(k,i) = (h_1(k) + ih_2(k)) \mod 31$ dove

$h_1(k) = ascii(k) \mod 31$ e $h_2(k) = (h_1(k) \mod 30) + 1$



C. Demetrescu, I. Finocchi, G. F. Italiano, "Algoritmi e strutture dati"

Indirizzamento aperto: analisi del caso medio

- Utilizziamo il fattore di carico $\alpha = n/m$ anche per l'analisi del costo medio col metodo di indirizzamento aperto
  - In questo caso, poiché $n \leq m$, abbiamo che $\alpha < 1$
- Assumiamo hashing uniforme semplice
- Assumiamo inoltre che le permutazioni degli indici $[0, \cdots, m-1]$ determinate dalle sequenze di ispezione

$$h(k,0), h(k,1), \cdots, h(k,m-1)$$

siano tutte equiprobabili

  - Ogni chiave k ha un'unica sequenza di ispezione associata
  - Ogni sequenza di ispezioni è ugualmente probabile
  - Questo dipende dalla strategia di ispezione: l'ispezione lineare non soddisfa tale proprietà, mentre è soddisfatta dall'ispezione quadratica e doppio hashing

Indirizzamento aperto: analisi del caso medio

Teorema (ricerca senza successo)

Sotto l'assunzione di hashing uniforme semplice, il numero medio di ispezioni di una ricerca senza successo in una tabella hash con indirizzamento aperto e fattore di carico $\alpha < 1$ è al massimo $1/(1-\alpha)$

Teorema (ricerca con successo)

Sotto l'assunzione di hashing uniforme semplice, il numero medio di ispezioni di una ricerca con successo in una tabella hash con indirizzamento aperto e fattore di carico $\alpha < 1$ è al massimo $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

- In entrambi i casi, se α è costante il tempo di accesso è $O(1)$
- Se la tabella è piena al 50%, la ricerca senza successo richiede in media al massimo due ispezioni, la ricerca con successo meno di due
- Se la tabella è piena al 90%, la ricerca senza successo richiede in media al massimo dieci ispezioni, la ricerca con successo meno di tre

Analisi del caso medio: ispezione lineare

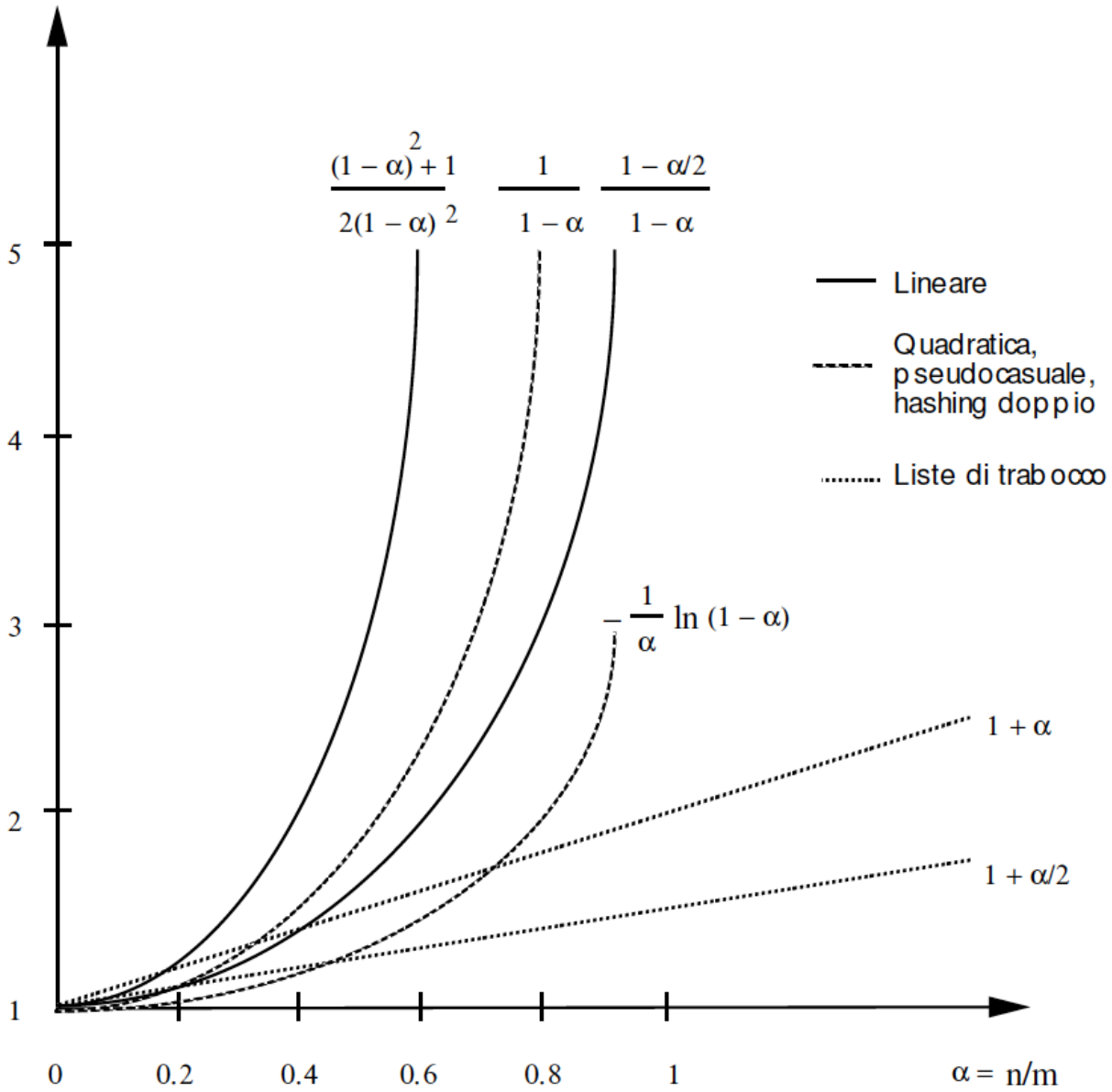
- L'ispezione lineare non assicura che le sequenze di ispezione siano tutte equiprobabili (effetto del clustering primario)
- Il costo medio non è caratterizzato dai due teoremi precedenti
  - Il costo medio nel caso di ricerca senza successo al massimo

$$\frac{(1-\alpha)^2+1}{2(1-\alpha)^2}$$

  - Il costo medio nel caso di ricerca con successo al massimo

$$\frac{(1-\alpha/2)}{2(1-\alpha)}$$

Confronto costi medi di ispezione



A. Bertossi, A. Montresor, "Algoritmi e strutture di dati"

Commenti generali: ruolo del fattore di carico

- Le prestazioni delle tabelle hash sono legate al fattore di carico α
- Secondo il paradosso del compleanno, le collisioni sono molto probabili
  - Le collisioni sono praticamente inevitabili anche su sottoinsiemi relativamente piccoli di possibili chiavi
- Strategia: mantenere il fattore di carico basso
  - Un fattore di carico $\alpha < 0.75$ è considerato ottimale
  - Ridimensioniamo la tabella quando il fattore di carico supera una certa soglia critica

Tabelle Hash in Java

- java.util.HashMap
  - classe equivalente a Hashtable (che è sincronizzata)
  - Rappresentazione con concatenamento
  - Java 7: liste di trabocco con liste concatenate
  - Java 8: liste di trabocco con RB alberi (alberi bilanciati di ricerca)
    - Costo pessimo logaritmico delle operazioni di ricerca, inserimento, rimozione
- Parametri fondamentali:
  - Fattore di carico (default 0.75)
  - Capacità iniziale (default 16)
  - Quando il numero di elementi eccede il prodotto tra fattore di carico e la capacità della Tabella Hash, questa viene ridimensionata (ricostruita completamente da capo) in modo che la nuova capacità diventi approssimativamente il doppio della precedente
  - Suggerimenti: evitare il più possibile i ridimensionamenti settando una capacità iniziale opportuna

Riassunto

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Liste concatenate	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Alberi BST	$O(\overline{h})$	$O(h)$	$O(\overline{h})$	$O(h)$	$O(\overline{h})$	$O(h)$
Alberi AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Tabelle Hash	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$

- Nota: $h = O(n)$ è l'altezza dell'albero
- $\overline{h}$ = altezza media dell'albero