

Algoritmi e Strutture Dati

Capitolo 7 Tavole hash

Implementazioni Dizionario

Tempo richiesto dall'operazione più costosa:

- Liste e array $O(n)$
- Alberi binari di ricerca $O(n)$
- Alberi AVL $O(\log n)$
- **Tavole hash** (H. P. Luhn, IBM, 1953) $O(1)$

...ma solo sotto certe condizioni!

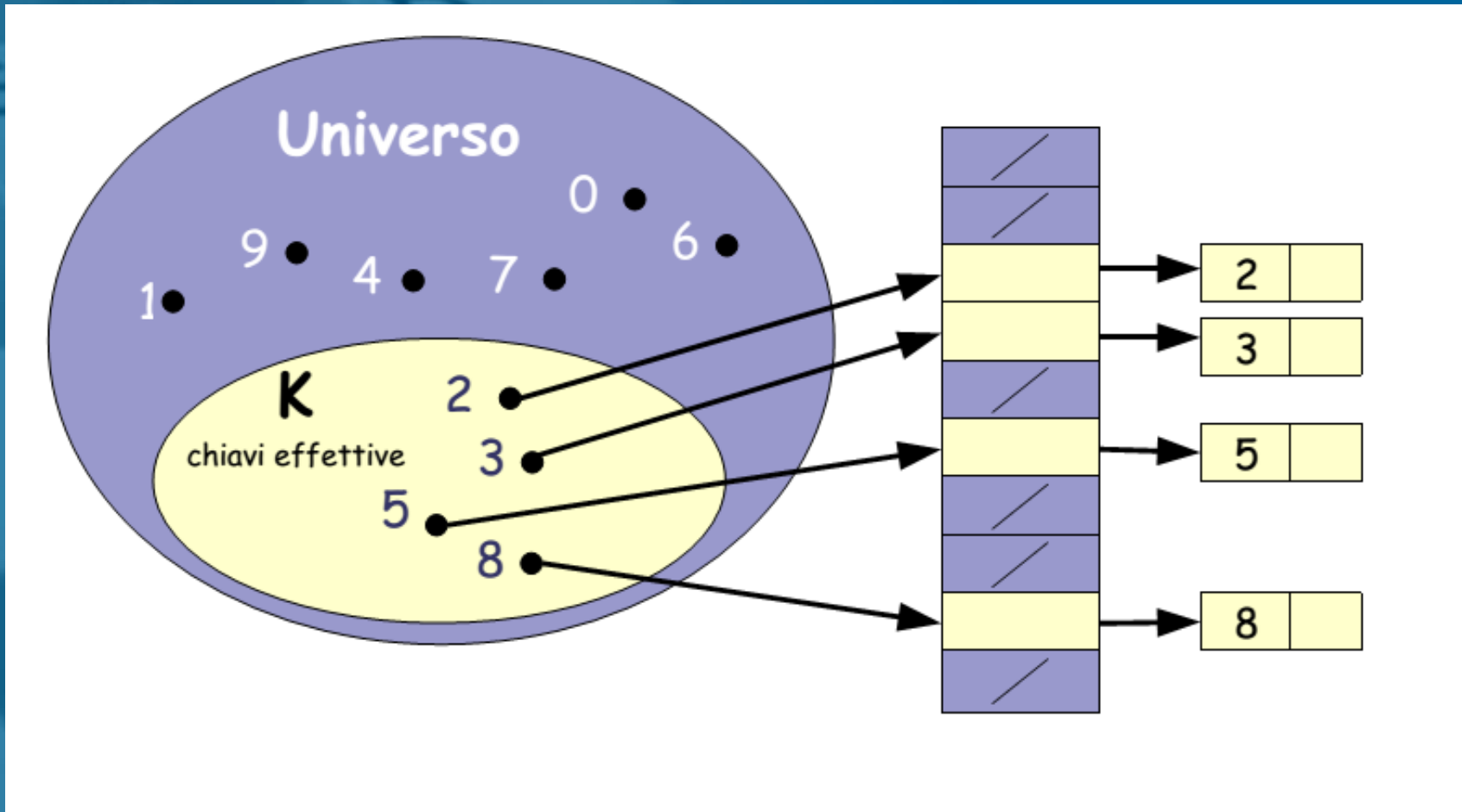
Preambolo: tavole ad accesso diretto

Sono dizionari basati sulla proprietà di accesso diretto alle celle di un array

Idea (simile all'Integer Sort):

- Denotiamo con n il numero corrente di elementi contenuti nel dizionario, e supponiamo che a ciascun elemento e sia associata una **chiave intera** k nell'intervallo $[0, m-1]$, con $m \geq n$ (unicità della chiave)
- Il dizionario viene memorizzato in un array v di m celle
- L'elemento con chiave k è contenuto in $v[k]$

Esempio



Implementazione

classe TavolaAccessoDiretto **implementa** Dizionario:

dati: $S(m) = \Theta(m)$
 un array v di dimensione $m \geq n$ in cui $v[k] = elem$ se c'è un elemento $elem$ con chiave k nel dizionario, e $v[k] = \text{null}$ altrimenti. Le chiavi k devono essere interi nell'intervallo $[0, m - 1]$.

operazioni:

$\text{insert}(elem\ e, chiave\ k)$ $T(n) = O(1)$
 $v[k] \leftarrow e$

$\text{delete}(chiave\ k)$ $T(n) = O(1)$
 $v[k] \leftarrow \text{null}$

$\text{search}(chiave\ k) \rightarrow elem$ $T(n) = O(1)$
return $v[k]$

Fattore di carico

- Analogamente all'Integer Sort, avrò che lo spazio utilizzato sarà proporzionale al valore massimo **m** che una chiave può assumere, e non al numero **n** di elementi effettivamente contenuti nel dizionario!
- Misuriamo il grado di riempimento di una tavola ad accesso diretto usando il **fattore di carico**

$$\alpha = \frac{n}{m}$$

Esempio: tavola con i nomi di **n=100** studenti indicizzati da numeri di matricola a 6 cifre:

$$n=100 \quad m=10^6 \quad \alpha = 0,0001 = 0,01\%$$

⇒ grande spreco di memoria!

Pregi e difetti

Pregi:

- Tutte le operazioni richiedono **tempo $O(1)$**

Difetti:

- Le chiavi devono essere necessariamente **interi** in $[0, m-1]$ (non possiamo accogliere un elemento con chiave $\geq m$, oppure chiavi definite su altri domini, ad esempio chiavi alfanumeriche)
- Lo spazio utilizzato è **proporzionale** alla chiave più grande **m** , e non al numero **n** di elementi effettivamente contenuti nel dizionario : può esserci grande spreco di memoria!

Tavole hash

Per ovviare agli inconvenienti delle tavole ad accesso diretto ne consideriamo un'estensione: le **tavole hash**. Nel seguito, denoteremo con **m** la dimensione della tavola hash.

Idea:

- Chiavi prese da un universo totalmente ordinato **U** (possono non essere numeri interi)
- **Funzione hash** (*to hash*, letteralmente, significa *tritare*): $h: U \rightarrow [0, m-1]$ (funzione che trasforma **chiavi** in **interi**, ovvero negli **indici** dell'array **v** di **m** celle che conterrà il dizionario)
- L'elemento con chiave $k \in U$ è contenuto in $v[h(k)]$

Collisioni

Le tavole hash possono soffrire del fenomeno delle **collisioni**:

Si ha una collisione quando si deve inserire nella tavola hash un elemento con chiave **u**, e nella tavola esiste già un elemento con chiave **v** tale che **$h(u)=h(v)$** : il nuovo elemento andrebbe a sovrascrivere il vecchio!

Funzioni hash perfette

Un modo per evitare il fenomeno delle collisioni è usare **funzioni hash perfette**:

Una funzione hash $h: U \rightarrow [0, m-1]$ si dice **perfetta** se è **iniettiva**, cioè per ogni $u, v \in U$:

$$u \neq v \implies h(u) \neq h(v)$$

NOTA: Ovviamente, deve essere $|U| \leq m$

Implementazione

classe TavolaHashPerfetta **implementa** Dizionario:

dati: $S(m) = \Theta(m)$

un array v di dimensione $m \geq n$ in cui $v[h(k)] = e$ se c'è un elemento e con chiave $k \in U$ nel dizionario, e $v[h(k)] = \text{null}$ altrimenti. La funzione $h : U \rightarrow \{0, \dots, m - 1\}$ è una funzione hash perfetta calcolabile in tempo $O(1)$.

operazioni:

$\text{insert}(\text{elem } e, \text{chiave } k)$ $T(n) = O(1)$
 $v[h(k)] \leftarrow e$

$\text{delete}(\text{chiave } k)$ $T(n) = O(1)$
 $v[h(k)] \leftarrow \text{null}$

$\text{search}(\text{chiave } k) \rightarrow \text{elem}$ $T(n) = O(1)$
return $v[h(k)]$

Esempio

Tavola hash con i nomi di $n=100$ studenti aventi come chiavi numeri di matricola nell'insieme $U=[234717, 235717] \Rightarrow |U|=1001$

Funzione hash perfetta: $h(k) = k - 234717$, cioè $h[k] \in [0, 1000] \Rightarrow$ alloco un array di $m=1001$ celle

$$n=100 \quad |U|=m=1001 \quad \alpha = n/m = 0,1 = 10\%$$

...ma il vincolo $m \geq |U|$ necessario per avere una funzione hash perfetta è raramente conveniente (o possibile)...

Funzioni hash **non perfette**

Una funzione hash si dice **non perfetta** se **non è iniettiva**, cioè esistono $u, v \in U$ per cui:

$$u \neq v \not\Rightarrow h(u) \neq h(v)$$

ovvero, $h(u) = h(v) \not\Rightarrow u = v$, e quindi si potrebbero avere delle **collisioni**.

Esempio

Sia $U=\{A,B,C,\dots,Z\}$ l'insieme delle 26 lettere dell'alfabeto inglese, e supponiamo di voler costruire una tavola hash per accogliere $m<26$ elementi aventi chiave in U

Funzione hash non perfetta:

$$h(k) = \text{ascii}(k) \bmod m \quad \text{con } m < 26$$

Ad esempio, per $m=11$:

$$h('C') = 67 \bmod 11 = 1$$

$$h('N') = 78 \bmod 11 = 1 \Rightarrow h('C') = h('N')$$

\Rightarrow se volessimo inserire nella tavola sia 'C' che 'N' avremmo una collisione!

Ridurre al minimo le collisioni

Per ridurre la probabilità di collisioni, una buona funzione hash dovrebbe essere in grado di distribuire in modo quanto più uniforme possibile le chiavi nello spazio degli indici della tavola

Questo accade, ad esempio, se la funzione hash gode della proprietà di **uniformità semplice**

Uniformità semplice

Sia $P(k)$ la probabilità che l'elemento con chiave k venga inserito nella tavola, e sia

$$Q(i) = \sum_{k \text{ in } U : h(k)=i} P(k)$$

il **numero atteso** di elementi che verranno indirizzati dalla funzione hash nella cella i . Allora, La funzione hash h gode dell'uniformità semplice se, per ogni i, j in $[0, m-1]$, si ha che $Q(i)=Q(j)$. In altre parole, ogni chiave deve avere la stessa probabilità di vedersi assegnata una qualsiasi posizione ammissibile, indipendentemente da altri valori hash già assegnati, e quindi la probabilità che 2 chiavi collidano è al più $1/m$.

Esempio

Se U è l'insieme dei **numeri reali** in $[0,1)$ e ogni chiave ha la stessa probabilità di essere scelta, allora è semplice dimostrare che la funzione hash:

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

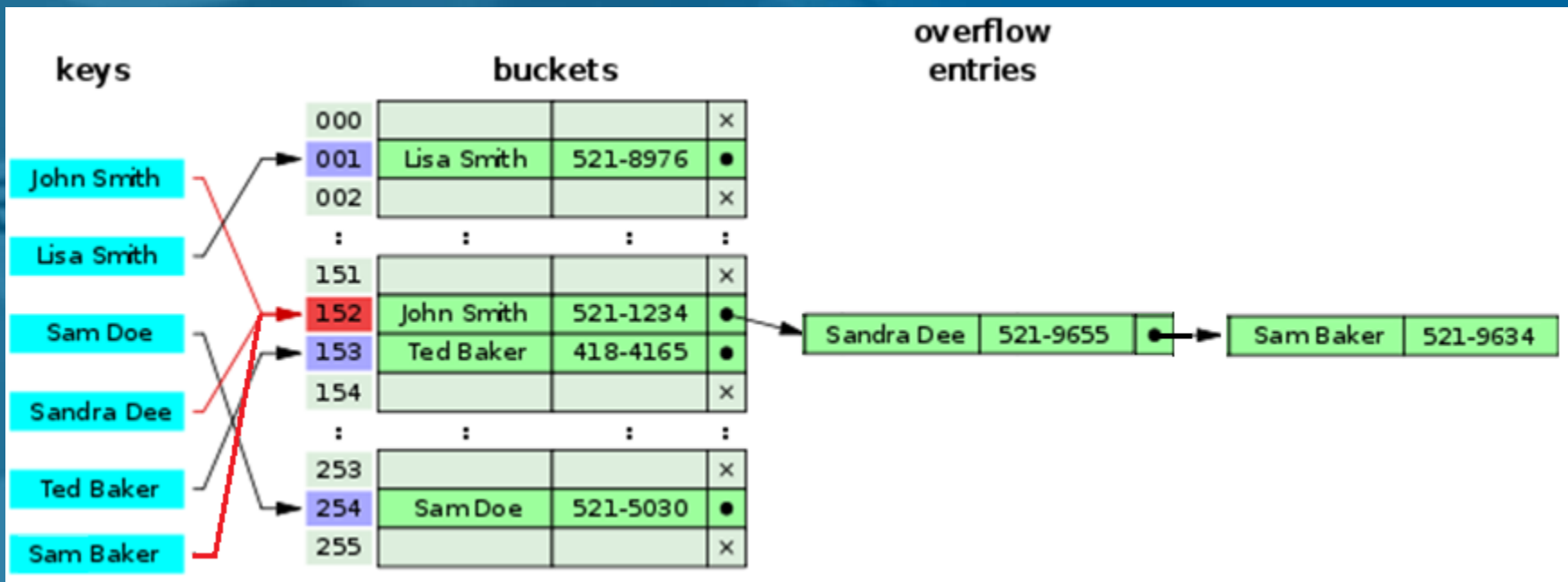
che in sostanza raggruppa le chiavi nell'intervallo $[0,1)$ in m sottintervalli uguali di dimensione $1/m$, soddisfa la proprietà di **uniformità semplice**

Risoluzione delle collisioni

Due metodi classici:

1. **Liste di collisione ($n \geq m, \alpha \geq 1$)**. Gli elementi collidenti sono contenuti in liste esterne alla tabella: $v[i]$ contiene il **primo** elemento inserito in v avente chiave k tale che $h(k)=i$, e un puntatore ad una lista di elementi con chiave k_1, k_2, \dots, k_p tali che $h(k_1)=h(k_2)=\dots=h(k_p)=i$
2. **Indirizzamento aperto ($n \leq m, \alpha \leq 1$)**. Tutti gli elementi sono contenuti nella tabella: se una cella è occupata, se ne cerca un'altra libera

1. Liste di collisione



Esempio di tabella hash con liste di collisione per la gestione di una rubrica telefonica. Si noti che in questo caso U è l'insieme di tutte le possibili stringhe (**cognome, nome**), e quindi $|U|$ cresce come 26^k , con k che denota la lunghezza massima di una stringa, che può ragionevolmente arrivare ad una ventina di caratteri! In tal caso ovviamente $|U| > m$, ed è inevitabile generare collisioni

Analisi del costo di una ricerca

- Nel caso **migliore**, $O(1)$
- Nel caso **peggiore**, $O(n)$ (devo scandire una lista di trabocco che contiene tutti gli elementi del dizionario)
- Nel caso **medio**, se la funzione hash gode dell'uniformità semplice, allora $T_{AVG}(n,m)=O(n/m)$, in quanto le liste di trabocco si equipartiscono gli elementi.

2. Indirizzamento aperto

- Supponiamo di voler inserire un elemento con chiave k e la sua posizione “naturale” $h(k)$ sia già occupata
- L’indirizzamento aperto consiste nell’occupare un’altra cella, anche se potrebbe spettare di diritto a un’altra chiave
- Cerchiamo la prima cella vuota disponibile **scandendo** le celle secondo una sequenza di **indici**:

$$c(k,0)=h(k), c(k,1), c(k,2), \dots c(k,m-1)$$

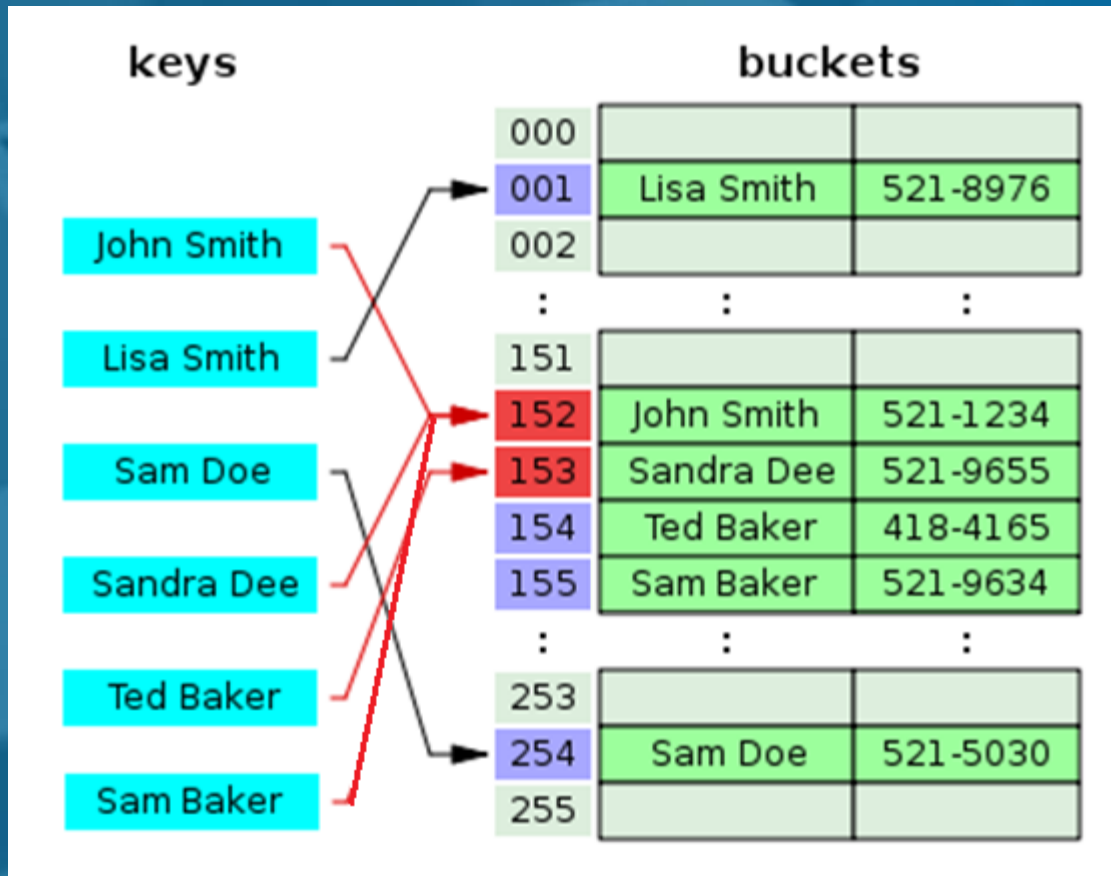
Metodi di scansione: scansione lineare

Scansione lineare: gli elementi che confliggono vengono messi l'uno dopo l'altro nella tabella (vengono cioè scandite celle **contigue**):

$$c(k,i) = (h(k) + i) \bmod m$$

$$\text{per } 0 \leq i < m$$

Esempio



Esempio di tabella hash con indirizzamento aperto a scansione lineare per la gestione di una rubrica telefonica: si noti la collisione “indiretta” tra **Ted Baker** e **Sandra Dee**, che in realtà avrebbero un’allocazione diversa secondo la funzione hash. Si noti anche come **Sam Baker** debba essere appeso in coda a **Ted Baker**, nonostante la sua collisione sia con **John Smith** (fenomeno dell’agglomerazione primaria)

Il problema dell'agglomerazione primaria

- La scansione lineare provoca effetti di **agglomerazione primaria**, cioè lunghi gruppi di celle consecutive occupate che rallentano la scansione: infatti, più cresce la dimensione di un gruppo di celle contigue occupate, e più tale insieme di celle tenderà a crescere (perché sempre più elementi collideranno e si accoderanno al gruppo)!

Metodi di scansione: scansione quadratica

Scansione quadratica: risolve il problema dell'agglomerazione primaria, scandendo celle **non contigue**.

$$c(k,i) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$$

per $0 \leq i < m$

Si può dimostrare che per $c_1=c_2=0.5$ e m potenza di 2 viene scandita tutta la tavola

Metodi di scansione: hashing doppio

- La scansione quadratica risolve il problema dell'agglomerazione primaria, ma provoca invece **agglomerazione secondaria**: coppie di chiavi collidenti generano la **stessa** sequenza di scansione: $h(k_1)=h(k_2) \Rightarrow c(k_1,i)=c(k_2,i)$

L'**hashing doppio** risolve il problema:

$$c(k,i) = \lfloor h_1(k) + i \cdot h_2(k) \rfloor \bmod m$$

per $0 \leq i < m$, h_1 e h_2 funzioni hash, m e $h_2(k)$ primi tra loro (così da scandire tutta la tabella: infatti se $\text{MCD}(h_2(k), m) = d > 1$ per qualche k , allora la ricerca di una posizione per tale chiave andrebbe ad esaminare solo una porzione di dimensione $1/d$ della tabella)

Analisi del costo di una ricerca

- Nel caso **migliore** $O(1)$, in quello **peggiore** $O(m)$
- Nel caso **medio**, un'operazione di ricerca di una chiave, assumendo che le chiavi siano prese con probabilità uniforme da U , costa:

<i>esito ricerca</i>	<i>sc. lineare</i>	<i>hashing doppio</i>
chiave trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)}$	$-\frac{1}{\alpha} \log_e(1 - \alpha)$
chiave non trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$	$\frac{1}{1-\alpha}$

dove $\alpha = n/m \leq 1$ è il fattore di carico. Ad esempio,

per $\alpha = 1/2$

1,5	1,38
2,5	2

mentre per $\alpha = 0,1$

1,05	1,05
1,11	1,11

Esercizio di approfondimento

Si supponga di inserire le chiavi 10, 22, 31, 4, 15, 28, 17, 88, 59 (in quest'ordine) in una tavola hash di lunghezza $m=11$ (con indici in $[0,10]$) utilizzando l'indirizzamento aperto con la funzione hash $h(k)=k \bmod m$. Illustrare il risultato dell'inserimento di queste chiavi utilizzando la scansione lineare, la scansione quadratica con $c_1=1$ e $c_2=3$, e l'hashing doppio con

$$h_2(k)=1+(k \bmod (m-1)).$$

Soluzione (scansione lineare)

- Scansione lineare

$$h(10) \rightarrow [10]$$

$$h(22) \rightarrow [0]$$

$$h(31) \rightarrow [9]$$

$$h(4) \rightarrow [4]$$

$$h(15) \rightarrow [4] \rightarrow [5]$$

$$h(28) \rightarrow [6]$$

$$h(17) \rightarrow [6] \rightarrow [7]$$

$$h(88) \rightarrow [0] \rightarrow [1]$$

$$h(59) \rightarrow [4] \rightarrow [5] \rightarrow [6] \rightarrow [7] \rightarrow [8]$$

22	88			4	15	28	17	59	31	10
----	----	--	--	---	----	----	----	----	----	----

Soluzione (scansione quadratica)

- Scansione quadratica

$$h(10) \rightarrow [10]$$

$$h(22) \rightarrow [0]$$

$$h(31) \rightarrow [9]$$

$$h(4) \rightarrow [4]$$

$$h(15) \rightarrow [4] \rightarrow [8]$$

$$h(28) \rightarrow [6]$$

$$h(17) \rightarrow [6] \rightarrow [10] \rightarrow [9] \rightarrow [3]$$

$$h(88) \rightarrow [0] \rightarrow [4] \rightarrow [3] \rightarrow [8] \rightarrow [8] \rightarrow [3] \rightarrow [4] \rightarrow [0] \rightarrow [2]$$

$$h(59) \rightarrow [4] \rightarrow [8] \rightarrow [7]$$

22		88	17	4		28	59	15	31	10
----	--	----	----	---	--	----	----	----	----	----

Soluzione (hashing doppio)

- Hashing doppio

$$h(10) \rightarrow [10]$$

$$h(22) \rightarrow [0]$$

$$h(31) \rightarrow [9]$$

$$h(4) \rightarrow [4]$$

$$h(15) \rightarrow [4] \rightarrow [10] \rightarrow [5]$$

$$h(28) \rightarrow [6]$$

$$h(17) \rightarrow [6] \rightarrow [3]$$

$$h(88) \rightarrow [0] \rightarrow [9] \rightarrow [7]$$

$$h(59) \rightarrow [4] \rightarrow [3] \rightarrow [2]$$

22		59	17	4	15	28	88		31	10
----	--	----	----	---	----	----	----	--	----	----