TECNICO SUPERIORE WEB DEVELOPER FULL STACK

#8 - Hashing

Ripasso

Un dizionario è una struttura dati utilizzata per memorizzare insiemi **dinamici** di coppie (chiave, valore):

- Le coppie sono indicizzate in base alla chiave;
- Il valore è un dato satellite, ossia una volta trovata la chiave abbiamo anche il valore.

Item lookup (Item k)	Restituisce il valore associato alla chiave k se presente, **nil** altrimenti
insert (Item k, Item v)	Associa il valore v alla chiave k
remove (Item k)	Rimuove l'associazione della chiave k

Dizionario - Ripasso della specifica

Possibili implementazioni				
Struttura dati	lookup	insert	remove	foreach
Vettore non ordinato	O(n)	O(1)*	O(1)*	O(n)
Vettore ordinato	O(logn)	O(n)	O(n)	O(n)
Lista non ordinata	O(n)	O(1)*	O(1)*	O(n)
Alberi binari di ricerca	O(log n)	O(log n)	O(log n)	O(n)
Struttura dati ideale	O(1)	O(1)	O(1)	O(n)

Una possibile struttura dati che si avvicina a quella ideale è la Tabella Hash (da “hacher”, “tagliuzzare” in francese)

Tabella Hash - Definizioni

Una funzione hash è definita come $h: U \rightarrow \{1, ..., m\}$ dove U è l'insieme delle chiavi possibili. In una coppia chiave-valore (k,v) il valore v viene memorizzato in una cella indirizzata da un puntatore nella posizione $h(k)$.

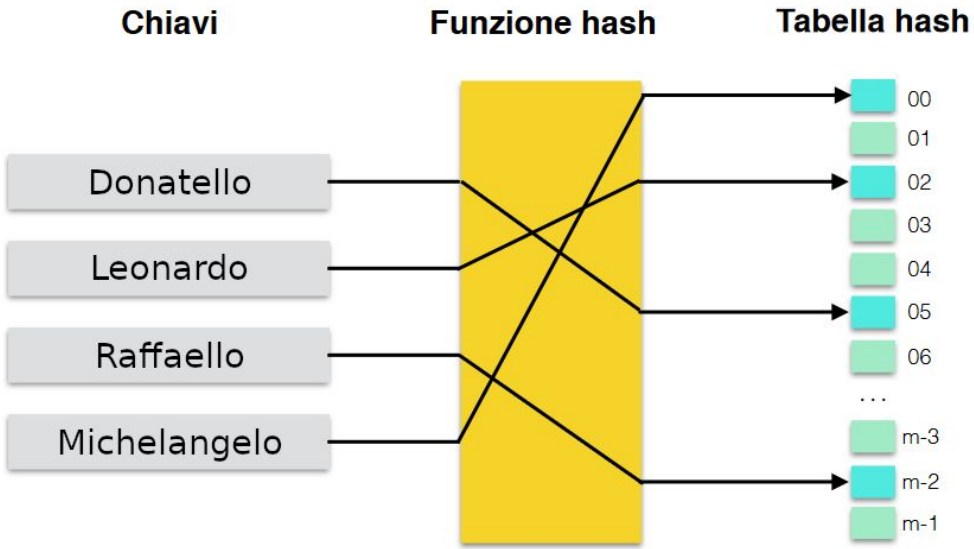


Tabella Hash - Definizioni

Quando due o più chiavi nel dizionario hanno lo stesso valore hash,diciamo che è avvenuta una collisione. Idealmente, vogliamo funzioni hash senza collisioni.

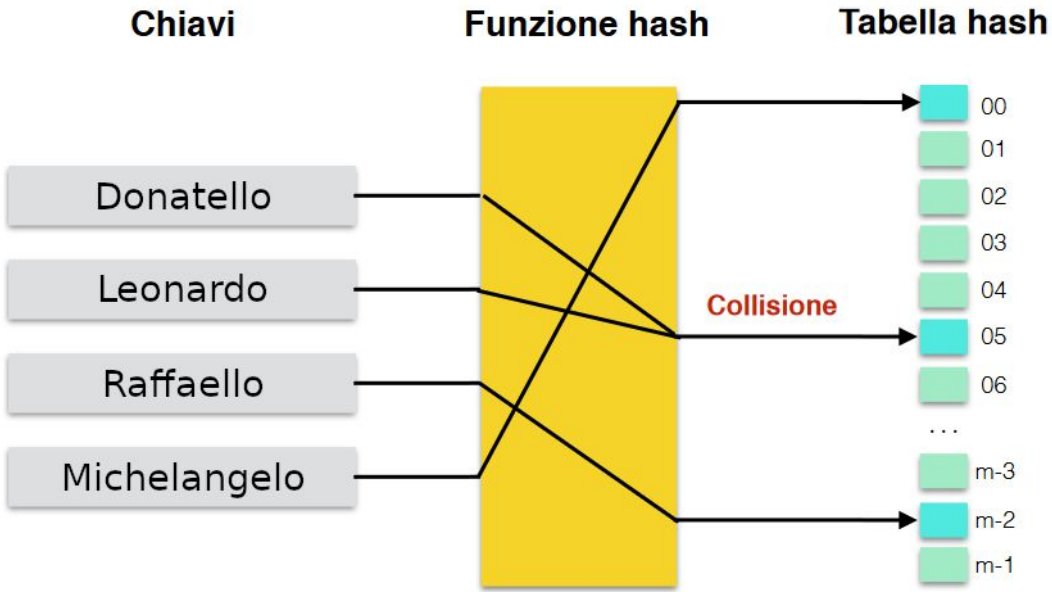


Tabelle ad accesso diretto

Quando l'insieme U è già un sottoinsieme (piccolo) di Z^+ (i numeri interi positivi), come i giorni dell'anno (numerati da 1 a 366) o l'insieme delle puntate di una serie TV, è più semplice utilizzare la funzione hash identità $h(k) = k$, con dimensione m pari al numero di valori in U.

Problemi:

- Se U è molto grande, l'approccio non è praticabile (creeremmo un vettore enorme);
- Se U non è grande ma il numero di chiavi effettivamente registrate è molto minore di U, si spreca memoria.

Considerazioni senza matematica sull'hash

Una funzione hash h si dice **perfetta** se non dà origine a collisioni, ma spesso l'insieme delle chiavi è molto più grande del numero , sparso, e non conosciamo tutte le sue caratteristiche (ad es. l'insieme dei nomi e cognomi).

Dunque è impraticabile ottenere una funzione hash realmente perfetta.

Probabilità

Un insieme di eventi $E = \{e_1, e_2, ..., e_k\}$ si dice **distribuito uniformemente** se la probabilità P che uno degli eventi si verifichi è pari a 1/k, ossia se ogni evento ha la stessa probabilità degli altri di verificarsi. La somma delle probabilità di tutti gli eventi in E è 1.

Ad esempio, in un dado non truccato ogni faccia ha probabilità di uscire pari ad 1/6.

Uniformità semplice

Se non possiamo evitare le collisioni, almeno cerchiamo di minimizzare il loro numero attraverso funzioni che distribuiscano **uniformemente** le chiavi negli indici [1...m] della tabella hash, ossia che ogni indice [1...m] abbia la stessa probabilità degli altri di uscire come risultato della funzione hash.

Una funzione h gode dell'uniformità semplice se vale che

$$\forall i \in [1,...,m] : P(i) = 1/m$$

Dove $P(i)$ è la probabilità che $h(k) = i$.

Problema

Per poter ottenere una funzione hash con uniformità semplice, devo conoscere la *distribuzione delle probabilità* di uscita delle chiavi, cioè devo sapere quanto è probabile che una certa chiave esista.

Tuttavia, per insiemi U molto grandi essa non è nota, e dunque si applicano tecniche di approssimazione, ossia manipolazioni che diano una distribuzione “sufficientemente” uniforme.

L’idea generale

L’idea di una funzione hash è dunque quella di “spezzettare” la chiave (o meglio, la sua rappresentazione binaria) in frammenti che, manipolati in qualche modo, permettano di ottenere un valore quanto più possibile uniformemente distribuito nell’intervallo [1...m].

(. . .)

Tra queste due parentesi c’è tutta la parte matematica necessaria per creare una funzione hash, che non vediamo.

(https://en.wikipedia.org/wiki/List_of_hash_functions)

Nella realtà...

Non è così semplice creare funzioni hash, ed esistono dei test per valutare la bontà delle funzioni:

- **Avalanche effect (Effetto valanga)**: se si cambia un bit nella chiave, deve cambiare almeno la metà dei bit del valore hash.
- **Test statistici (Chi-quadrato)**: verifica quanto i risultati della funzione si discostano da quelli attesi (1/m).

Complessità della funzione hash

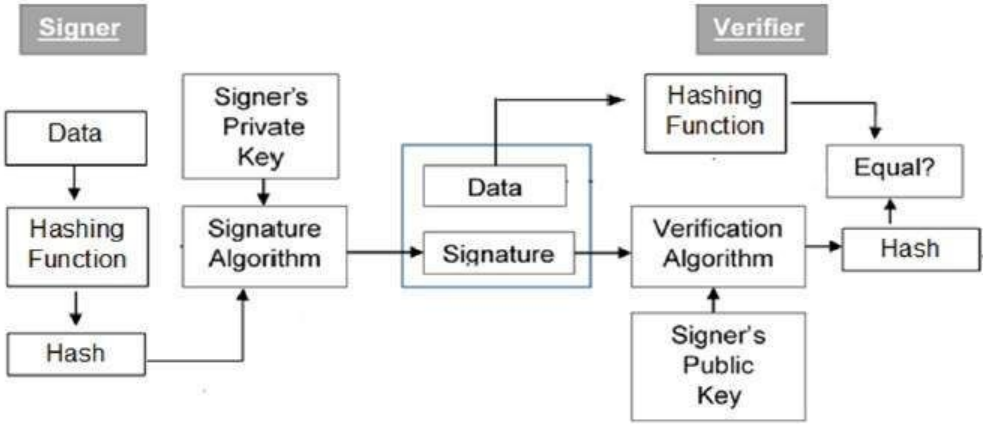
Il costo computazionale della funzione hash è $\Theta(1)$, ossia costante, perché:

- possiamo rappresentare le chiavi in una “forma standard”, ad esempio una sequenza binaria con un numero di bit “comodo”, tipo una potenza di 2;
- le operazioni di calcolo effettuate dalle funzioni hash possono quindi essere effettuate attraverso istruzioni in codice macchina.

Hash e crittografia

Il principio delle funzioni hash è usato anche in crittografia, ma in quel caso la funzione deve essere non invertibile, cioè deve essere molto difficile o quasi impossibile:

- risalire al testo che ha portato ad un certo valore hash;
- generare un testo che produca un valore hash specifico.



Collisioni

Una collisione avviene quando due o più chiavi nel dizionario hanno lo stesso valore hash. Visto che è impossibile evitarle, tanto vale trovare un modo per gestire le collisioni in modo efficiente.

Gestire le collisioni

In pratica, quando la posizione che dovrebbe occupare una chiave è già occupata da un valore (diverso) precedente, dobbiamo trovare una posizione alternativa.

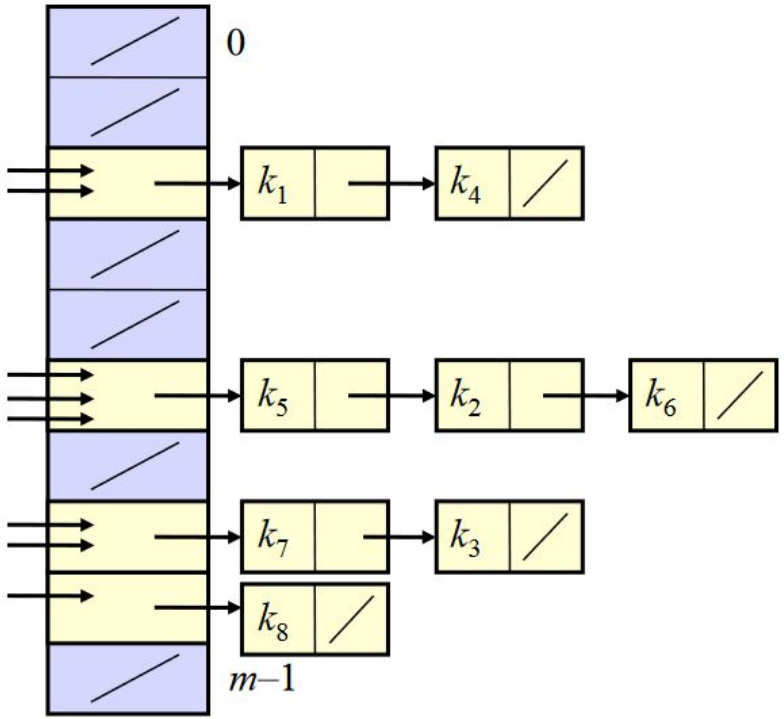
Inoltre, se una chiave non si trova nella posizione attesa, bisogna cercarla nelle posizioni alternative.

Questa ricerca dovrebbe costare $O(1)$ nel caso medio, ma può arrivare a costare $O(n)$ nel caso pessimo

Liste/vettori di trabocco

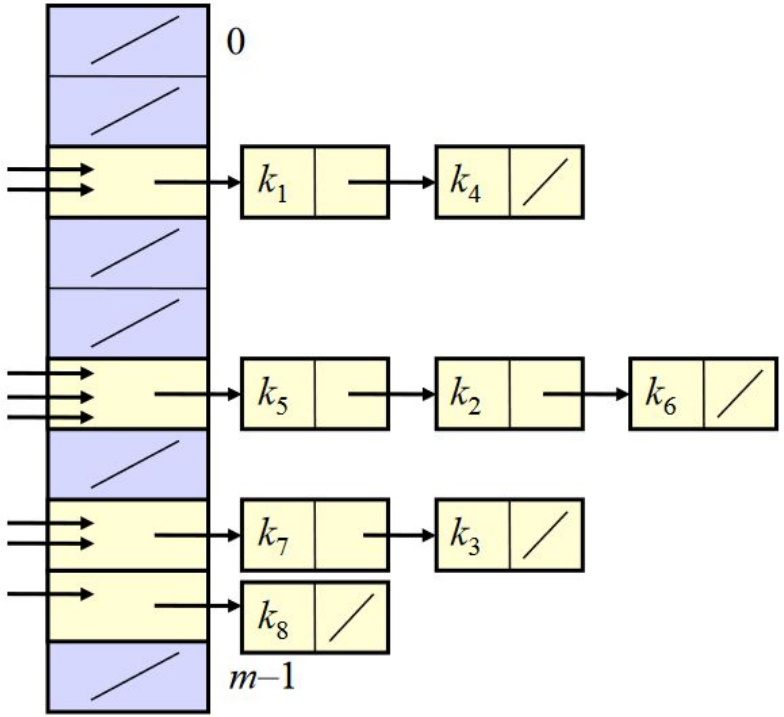
Con questo metodo le chiavi con lo stesso valore hash vengono memorizzate in una lista monodirezionale (o un vettore dinamico).

Si memorizza un puntatore alla testa della lista (o al vettore) nello slot H(k)-esimo della tabella hash.



Operazioni

- insert():
  inserisce il nuovo elemento in testa o in coda
- lookup(), remove():
  scansione della lista per cercare la chiave (e rimuoverla)



Complessità delle liste di trabocco

Utilizziamo le seguenti definizioni:

n	Numero di chiavi memorizzati in tabella hash
m	Capacità totale della tabella hash
$\alpha = n/m$	Fattore di carico (“quanta tabella è occupata”)
$I(\alpha)$	Numero medio di accessi alla tabella per la ricerca di una chiave non presente nella tabella (ricerca senza successo o miss)
$S(\alpha)$	Numero medio di accessi alla tabella per la ricerca di una chiave presente nella tabella (ricerca con successo o hit)

Caso pessimo

Nel caso pessimo, tutte le chiavi sono collocate in unica lista.

- insert(): $\Theta(1)$ - inserimento in testa
  $\Theta(n)$ - inserimento in coda

- lookup(), remove(): $\Theta(n)$

Avete appena trasformato una bella struttura dati in una lista non ordinata.

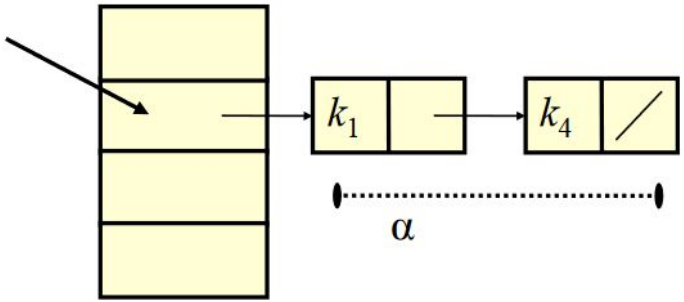
Caso medio

Ipotesi

- Assumiamo hashing uniforme semplice
- Costo calcolo funzione di hashing: $\Theta(1)$

Quanto sono lunghe le liste di trabocco nel caso medio?

Il valore atteso della lista di trabocco è $\alpha = n/m$.



Caso medio: costo effettivo

Ricerca senza successo

Una ricerca senza successo tocca tutte le chiavi nella lista corrispondente:

Costo atteso: $\Theta(1) + \alpha$

Ricerca con successo

Una ricerca con successo tocca in media metà delle chiavi nella lista corrispondente (alcuni hit saranno all'inizio della lista, altri alla fine):

Costo atteso: $\Theta(1) + \alpha/2$

L’importanza del fattore di carico

Il fattore di carico α influenza il costo computazionale delle operazioni sulle tabelle hash.

$$\text{Se } n = O(m) \text{ allora } \alpha = n/m = O(m)/m = O(1).$$

L’obiettivo è quindi quello di mantenere il fattore di carico sufficientemente piccolo da rendere costanti tutte le operazioni (potenzialmente potrebbe salire oltre 1).

Indirizzamento aperto

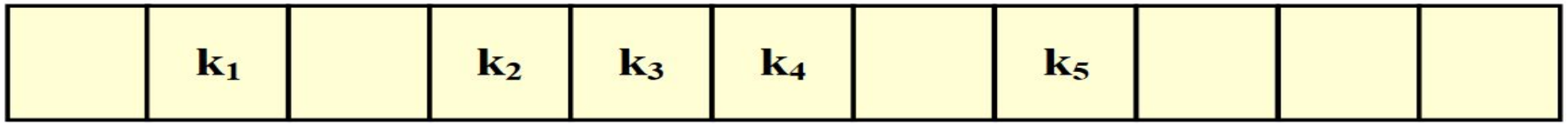
Problema: le liste di trabocco sono strutture dati complesse, con liste, puntatori, etc.

La soluzione di gestione alternativa è l'**indirizzamento aperto**, dove memorizzando tutte le chiavi direttamente nel vettore stesso, senza ulteriori puntatori (risparmio di memoria).

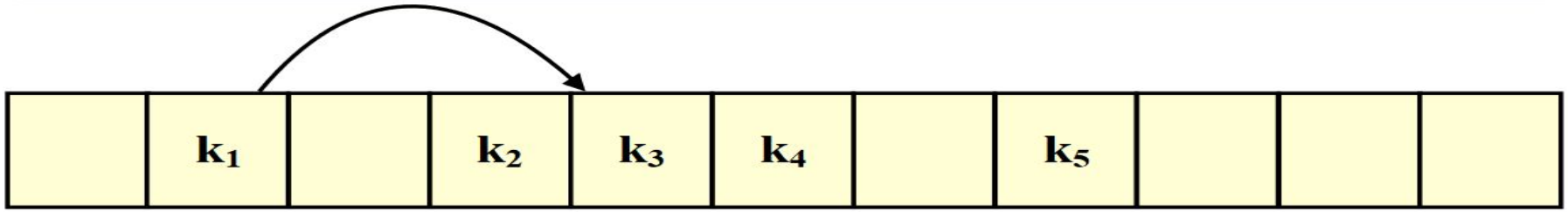
Ogni slot del vettore contiene una chiave oppure il valore nil.

- **Inserimento**: Se lo slot prescelto è utilizzato,si cerca uno slot "alternativo"
- **Ricerca**: Si cerca nello slot prescelto, e poi negli slot "alternativi" fino a quando non si trova la chiave oppure nil.

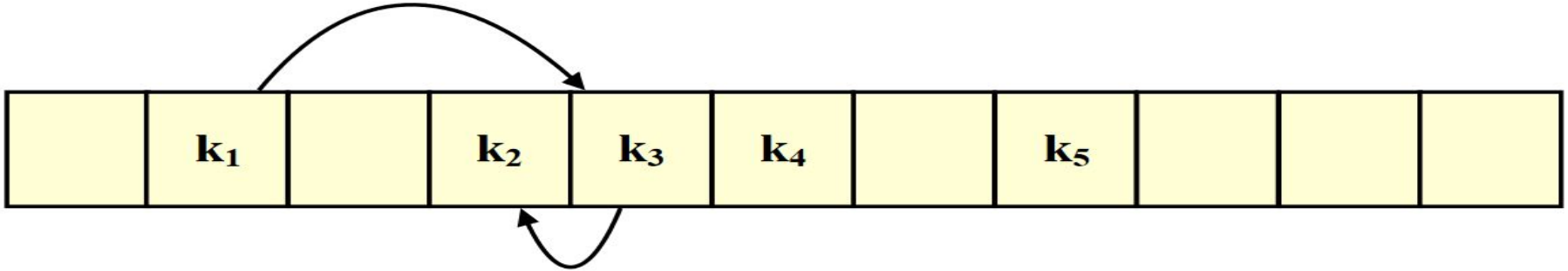
Ispezione



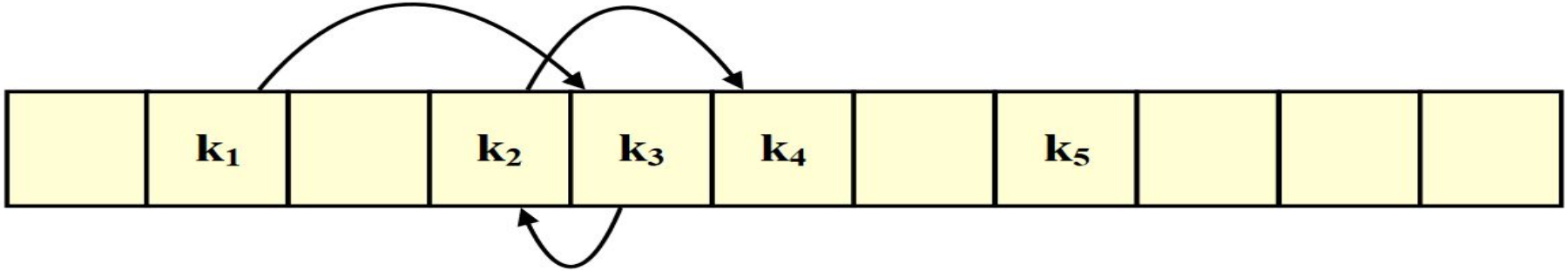
Ispezione



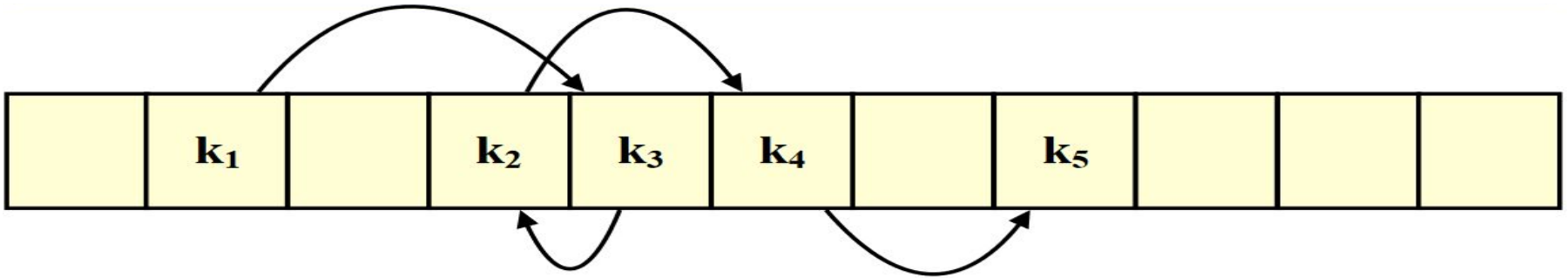
Ispezione



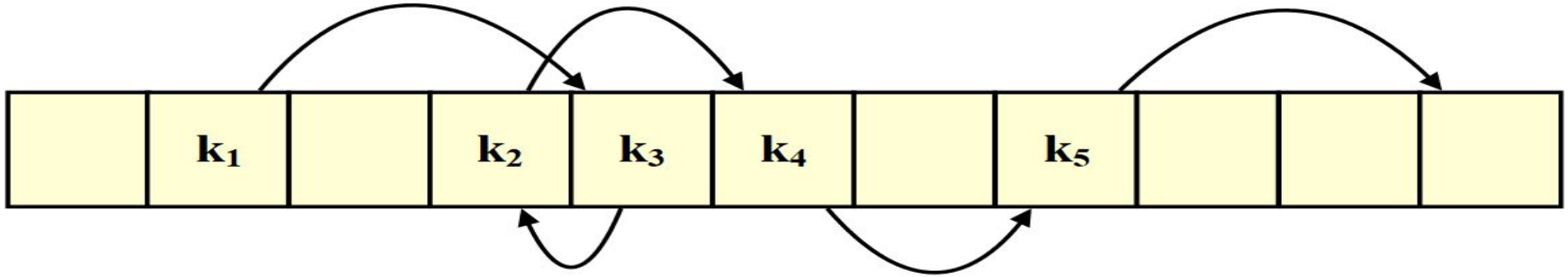
Ispezione



Ispezione



Ispezione



Come cambia il fattore di carico?

Il fattore di carico rimane sempre compreso tra 0 e 1, ma all'aumentare delle celle occupate tendiamo ad una ricerca lineare, perché l'ispezione deve sempre cercare "da un'altra parte".

Tecniche di ispezione

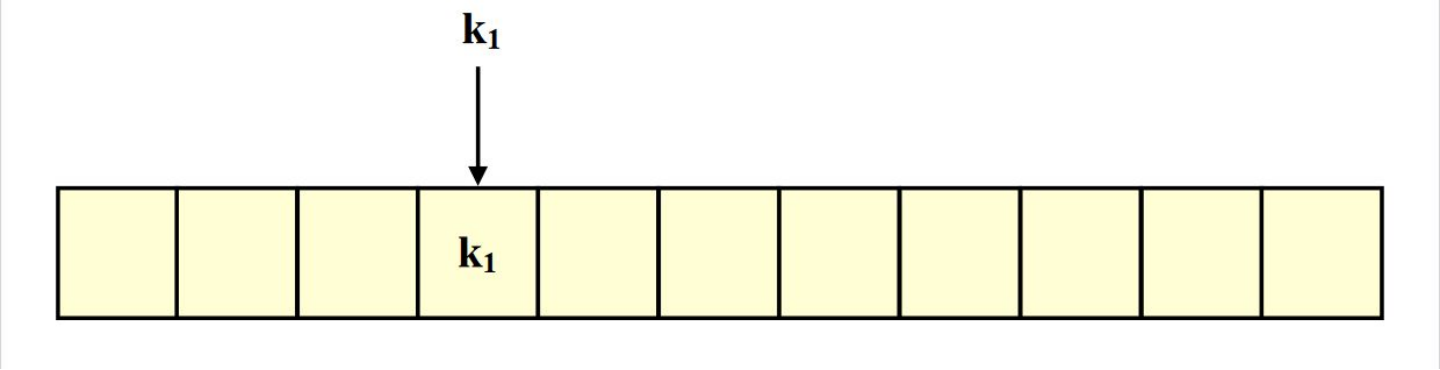
Senza entrare nei dettagli matematici delle varie tecniche, ne distinguiamo principalmente tre:

- Ispezione lineare
- Ispezione quadratica
- Doppio hashing

Ispezione lineare

Trova la casella di partenza, poi spostati di un numero fisso di caselle: ad es. $h(k)+0$, $h(k)+1$, $h(k)+2$, $h(k)+3\ldots$

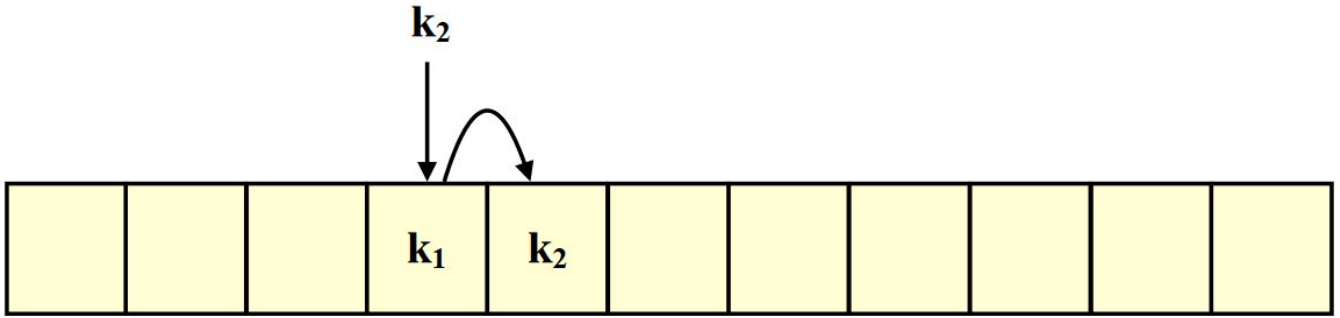
Problema: Agglomerazione primaria



Ispezione lineare

Trova la casella di partenza, poi spostati di un numero fisso di caselle: ad es. $h(k)+0$, $h(k)+1$, $h(k)+2$, $h(k)+3\ldots$

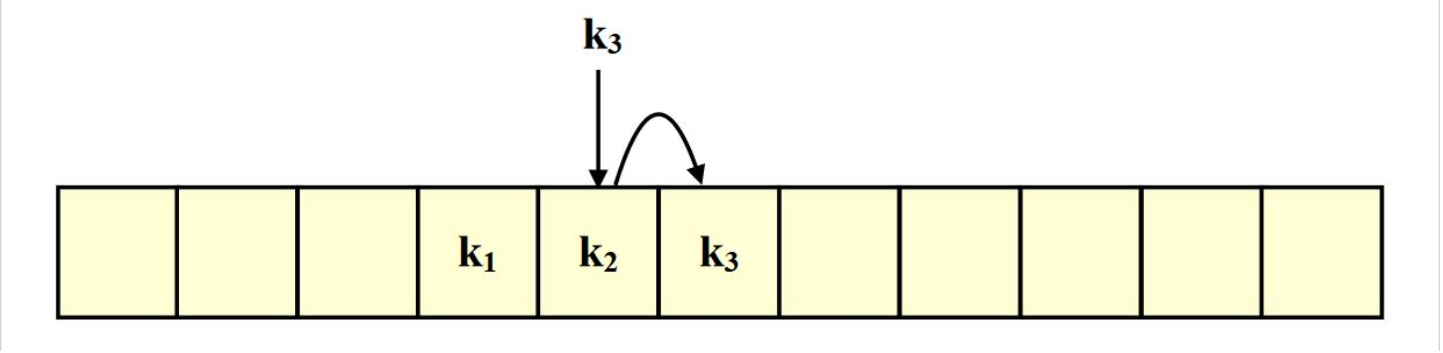
Problema: Agglomerazione primaria



Ispezione lineare

Trova la casella di partenza, poi spostati di un numero fisso di caselle: ad es. $h(k)+0$, $h(k)+1$, $h(k)+2$, $h(k)+3\ldots$

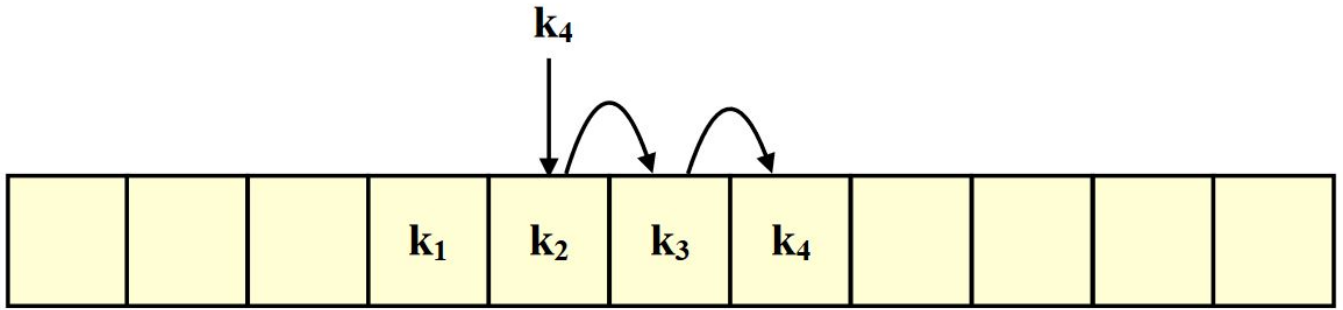
Problema: Agglomerazione primaria



Ispezione lineare

Trova la casella di partenza, poi spostati di un numero fisso di caselle: ad es. $h(k)+0$, $h(k)+1$, $h(k)+2$, $h(k)+3\ldots$

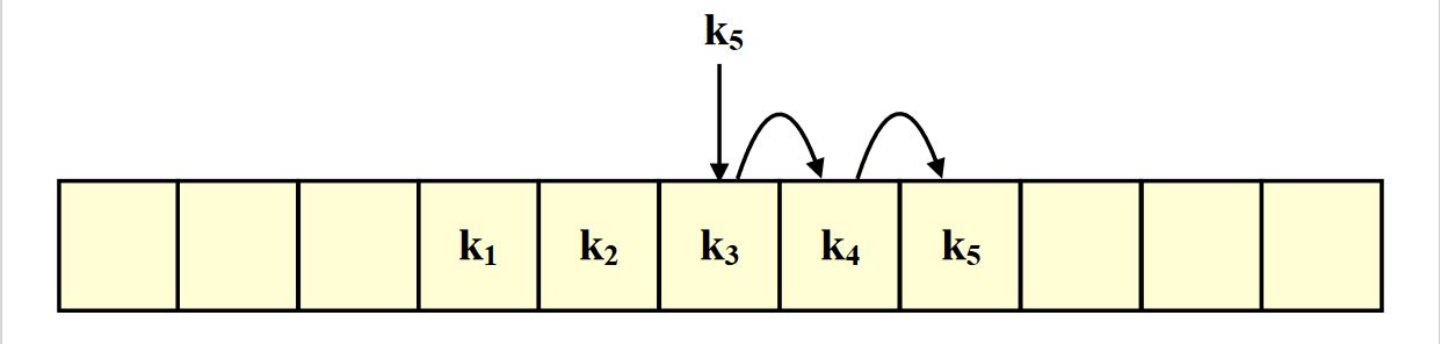
Problema: Agglomerazione primaria



Ispezione lineare

Trova la casella di partenza, poi spostati di un numero fisso di caselle: ad es. $h(k)+0$, $h(k)+1$, $h(k)+2$, $h(k)+3\ldots$

Problema: Agglomerazione primaria



Ispezione quadratica

Trova la casella di partenza, poi spostati di un numero di caselle proporzionale ai “salti”: ad es. $h(k)+0$, $h(k)+1$, $h(k)+4$, $h(k)+9\ldots$

Problema: Agglomerazione secondaria

Se due chiavi hanno lo stesso hash, le loro sequenze sono identiche. Inoltre, alcune caselle potrebbero essere ripetute (non provo ogni casella solo una volta).

Doppio Hashing

Trova la casella di partenza, poi spostati di un numero di caselle pari al valore di una seconda funzione di hash:

$$h_1(k)+0 \cdot h_2(k),\ h_1(k)+1 \cdot h_2(k),\ h_1(k)+2 \cdot h_2(k),\ h_1(k)+3 \cdot h_2(k)\ldots$$

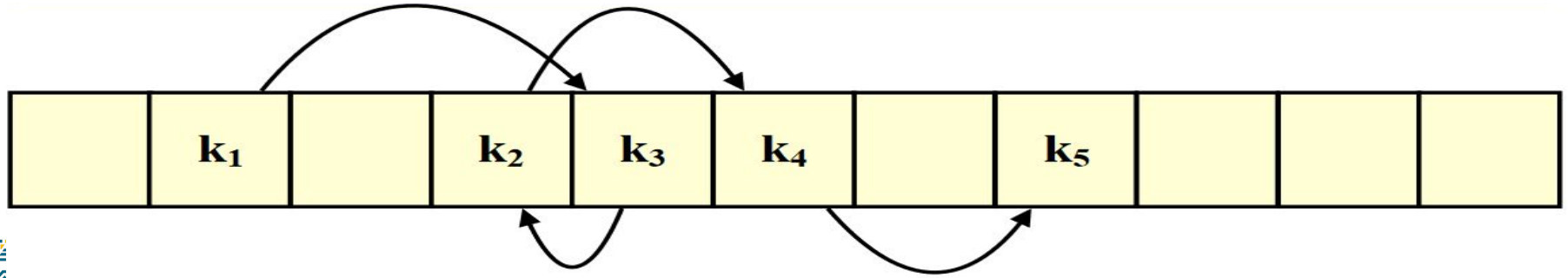
Usiamo due funzioni ausiliarie: h_1 fornisce la prima ispezione e h_2 fornisce l'offset delle successive.

Cancellazione nell’indirizzamento aperto

Non possiamo semplicemente sostituire la chiave che vogliamo cancellare con un nil. Perché?

Cancellazione nell’indirizzamento aperto

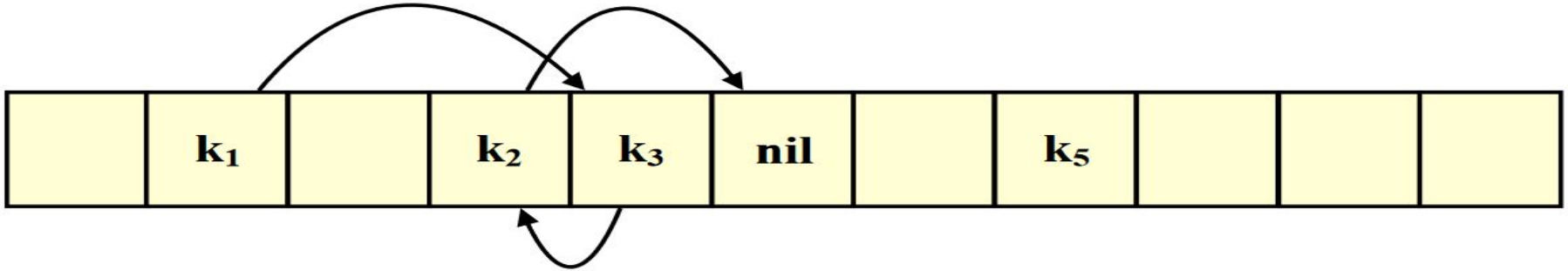
Non possiamo semplicemente sostituire la chiave che vogliamo cancellare con un nil. Perché?



Cancellazione nell'indirizzamento aperto

Non possiamo semplicemente sostituire la chiave che vogliamo cancellare con un nil. Perché?

Potremmo "rompere" una catena di ispezione, facendo risultare mancante un elemento che in realtà c'è.

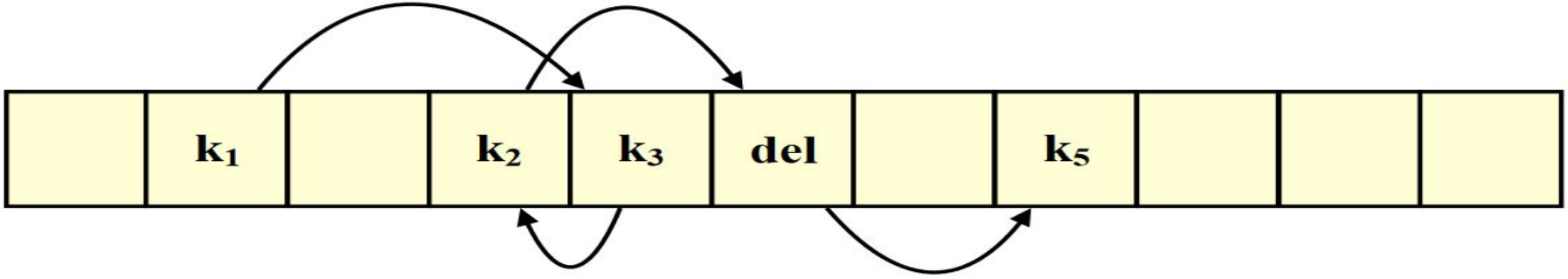


Cancellazione nell'indirizzamento aperto

Utilizziamo un speciale valore **del** al posto di **nil** per marcare uno slot come vuoto dopo la cancellazione

- Ricerca: del trattati come slot pieni
- Inserimento: del trattati come slot vuoti

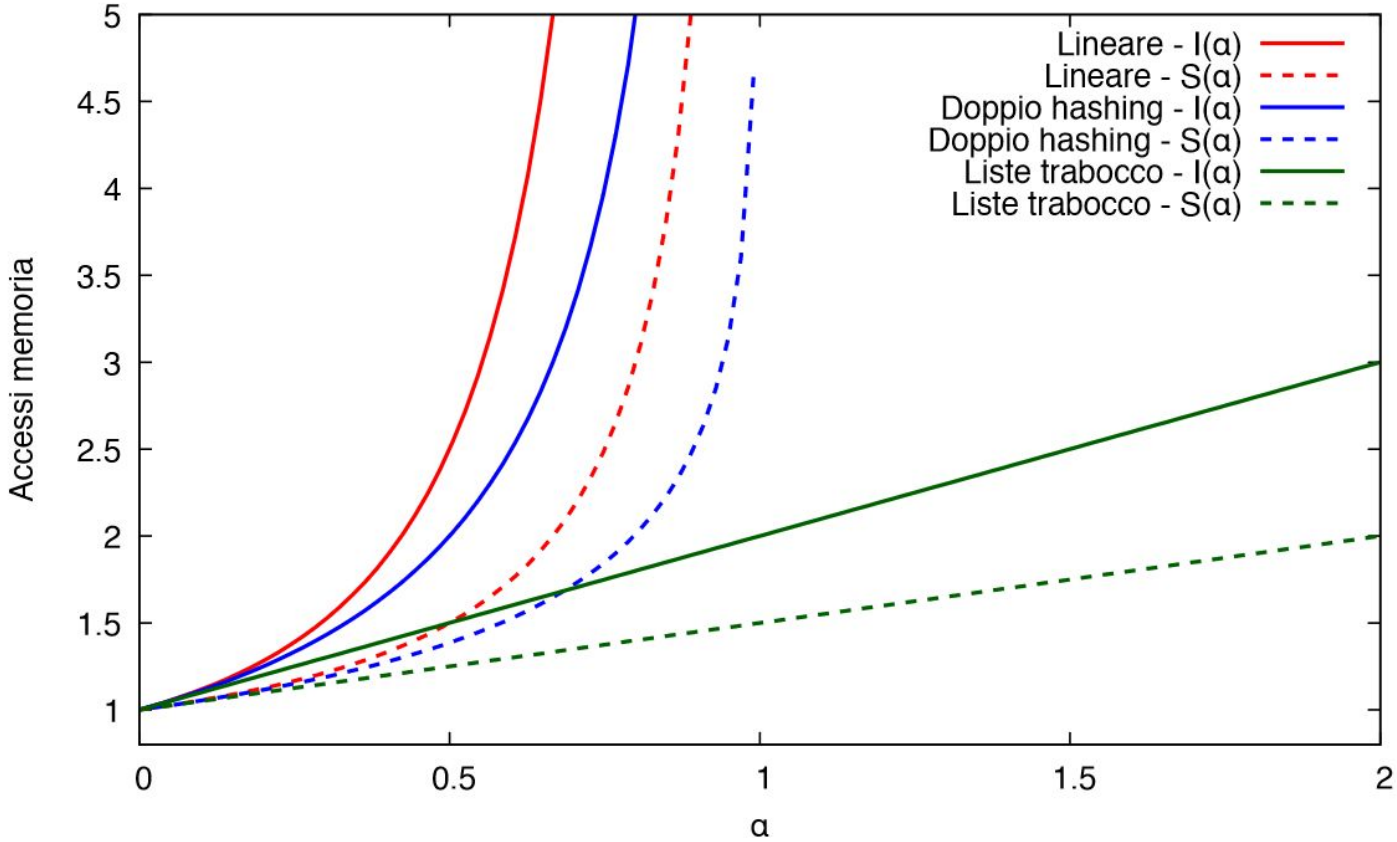
Svantaggio: il tempo di ricerca non dipende più da α (non so più quali caselle sono piene davvero e quali no)



Complessità

Metodo	α	$I(\alpha)$	$S(\alpha)$
Lineare	$0 \leq \alpha < 1$	$\frac{(1-\alpha)^2+1}{2(1-\alpha)^2}$	$\frac{1-\alpha/2}{1-\alpha}$
Hashing doppio	$0 \leq \alpha < 1$	$\frac{1}{1-\alpha}$	$-\frac{1}{\alpha}\ln(1-\alpha)$
Liste di trabocco	$\alpha \geq 0$	$1+\alpha$	$1+\alpha/2$

Complessità



Dalla documentazione di Object (C#)

A hash code is a numeric value that is used to insert and identify an object in a hash-based collection such as the Dictionary<TKey,TValue> class, the Hashtable class, or a type derived from the DictionaryBase class. The *GetHashCode* method provides this hash code for algorithms that need quick checks of object equality.

Two objects that are equal return hash codes that are equal. However, the reverse is not true: equal hash codes do not imply object equality, because different (unequal) objects can have identical hash codes. Furthermore, .NET does not guarantee the default implementation of the GetHashCode method, and the value this method returns may differ between .NET implementations, such as different versions of .NET Framework and .NET Core, and platforms, such as 32-bit and 64-bit platforms. **For these reasons, do not use the default implementation of this method as a unique object identifier for hashing purposes**. Two consequences follow from this:

- You should not assume that equal hash codes imply object equality.
- You should never persist or use a hash code outside the application domain in which it was created, because the same object may hash across application domains, processes, and platforms.

Dalla documentazione di Object (C#)

A hash code is intended for efficient insertion and lookup in collections that are based on a hash table. **A hash code is not a permanent value**. For this reason:

- Do not serialize hash code values or store them in databases.
- Do not use the hash code as the key to retrieve an object from a keyed collection.
- Do not send hash codes across application domains or processes. In some cases, hash codes may be computed on a per-process or per-application domain basis.
- Do not use the hash code instead of a value returned by a cryptographic hashing function if you need a cryptographically strong hash.
- Do not test for equality of hash codes to determine whether two objects are equal. (Unequal objects can have identical hash codes.) To test for equality, call the ReferenceEquals or Equals method.

If you override the GetHashCode method, you should also override Equals, and vice versa. If your overridden Equals method returns true when two objects are tested for equality, your overridden GetHashCode method must return the same value for the two objects.

Assolutamente no!

```
public override int GetHashCode()
{
    return 0;
}
```

Perchè?

Conclusioni

Problemi con le tabelle di hash

- Scarsa “locality of reference”: continuando a “saltare” da un punto all’altro della tabella, posso avere molti cache miss.
- Non è possibile ottenere le chiavi in ordine. Tipo mai. E’ tutto disperso nello sminuzzamento fatto dalla funzione.

Applicazioni delle funzioni di hash

- Protezioni dati con hash crittografici (MD5)
- Data deduplication (DropBox, Google Drive, ecc.)