Hashing e Tabelle Hash

Alessandro Pellegrini
pellegrini@diag.uniroma1.it

Università di Roma Tor Vergata

Dizionari: Abstract Data Type

- Un dizionario è una struttura dati utilizzata per memorizzare *insiemi dinamici* di coppie $\langle chiave, valore\rangle$.
  - Le coppie sono indicizzate in base alla chiave
  - Il valore è un dato satellite
- Si tratta quindi di un'altra implementazione di una struttura dati di tipo *insieme*
- Le chiavi appartengono ad un certo *universo* $\mathfrak{U}$.
  - È un insieme *totalmente ordinato*
- Operazioni da supportare:
  - INSERT(key, element):
  - DELETE(key)
  - SEARCH(key) $\longrightarrow$ element

Esempio di utilizzo dei dizionari

```
>>> v = {}
>>> v[10] = 5
>>> v["10"] = 42
>>> print(v[10]+v["10"])
47
```

Possibili implementazioni

	Array non ordinato	Array ordinato	Lista	Alberi RB	Ideale
INSERT()	$O(n), O(1)$	$O(n)$	$O(n), O(1)$	$O(\log n)$	$O(1)$
SEARCH()	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
DELETE()	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
Iterazione	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Funzioni hash

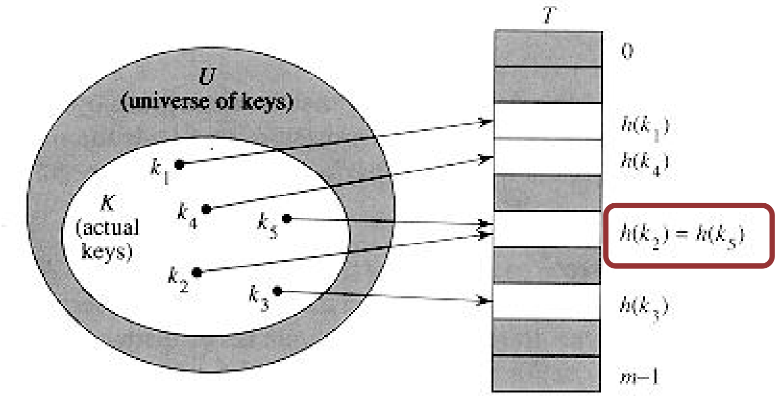
- Una funzione *hash* h mappa una chiave dall'universo $\mathfrak{U}$ in un intero
  - Dal francese *hacher* (“fare a pezzi”), dall'antico francese *hache* (ascia)
- La coppia $\langle chiave, valore \rangle$ viene memorizzata in un array in posizione $h(chiave)$
- Questo vettore viene denominato *tabella hash*

- Altre applicazioni delle funzioni hash (tipicament con hash crittografico):
  - Digest e firma digitale
  - Verifica di password
  - Proof of work (blockchain)
  - Identificazione di file (git)
  - Identificazione della posizione di file (magnet link)
  - Deduplica dei dati

Tabella hash

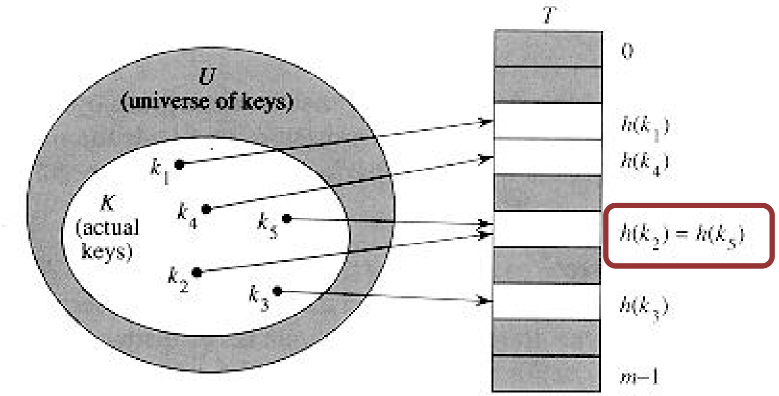
- La tabella hash è un vettore di dimensione N
- Le funzioni hash convertono le chiavi (dall'universo $\mathfrak{U}$) in un intero delle *pseudochiavi*—l'insieme delle pseudochiavi è di taglia prefissata N:

$$h: \mathfrak{U} \mapsto \{0, \dots, N-1\}$$



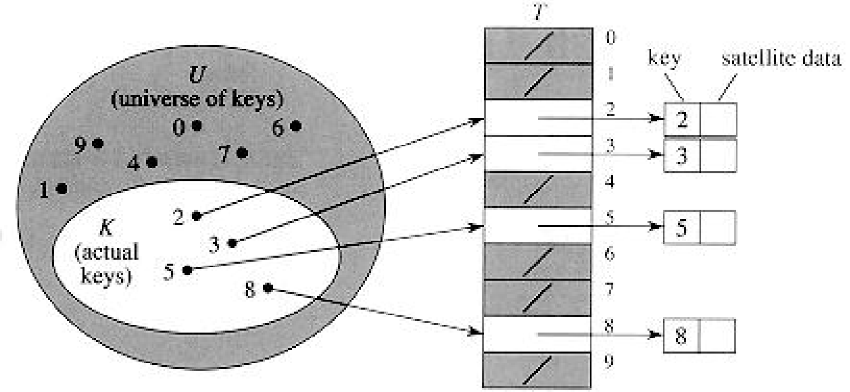
Collisioni

- Due o più chiavi appartenenti ad $\mathfrak{U}$ possono avere lo stesso valore hash
  - In questo caso si parla di *collisione*
- Idealmente, vorremmo funzioni hash senza collisioni



Funzione hash banale: tabelle ad accesso diretto

- Funzione hash *identità*: $h(k) = k$, tabella di dimensione $N = |\mathfrak{U}|$
  - La chiave diventa l'indice della tabella in cui si memorizza l'elemento
  - L'accesso diretto funziona bene se l'universo $\mathfrak{U}$ è ragionevolmente limitato
- L'assunzione dietro questa strategia è che due elementi differenti non saranno mai associati alla stessa chiave
- Se l'universo $\mathfrak{U}$ è troppo grande, non è una strategia praticabile
  - La tabella esploderebbe



Funzioni Hash

Funzioni hash perfette

- Idealmente, vorremmo funzioni hash senza collisioni
- Una funzione hash h si dice *perfetta* se è *iniettiva*, ovvero:

$$\forall k_1, k_2 \in \mathfrak{U}:\ k_1 \neq k_2 \Longrightarrow h(k_1) \neq h(k_2)$$

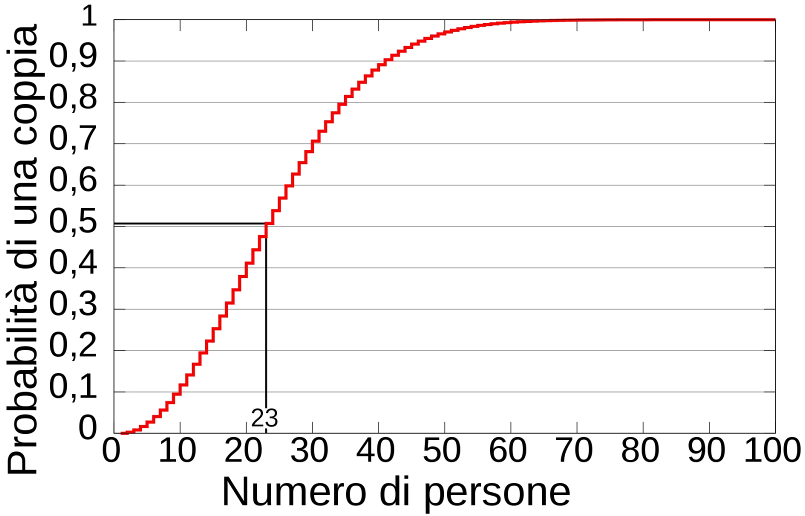
- Problemi:
  - Lo spazio delle chiavi è spesso molto grande, sparso e non conosciuto
  - È spesso impraticabile ottenere una funzione hash perfetta
- Conseguenza:
  - $k_1 \neq k_2$, ma $h(k_1) = h(k_2)$, ovvero collisione

Uniformità semplice

- Se non è possibile evitare le collisioni, cerchiamo almeno di ridurle!
  - una funzione hash deve quindi *distribuire uniformemente* le chiavi negli indici $[0, N-1]$ della tabella hash
- Principio di *uniformità semplice*:
  - Sia $P(k)$ la probabilità che una chiave k sia inserita in tabella
  - Sia $Q(i)$ la probabilità che una chiave finisca nell'elemento i-esimo della tabella: $Q(i) = \sum_{k \in \mathfrak{U} : h(k)=i} P(k)$
  - Si ha uniformità semplice se $\forall i \in [0, N-1] : \ Q(i) = \frac{1}{N}$
- Per poter realizzare una tale funzione hash, la distribuzione di probabilità P deve essere nota
  - È dunque difficile costruire funzioni che soddisfino questa proprietà in *ogni circostanza*

Paradosso del compleanno

- Scelti 23 individui a caso, la probabilità che due persone siano nate nello stesso giorno è del 51%
- Con 30 persone, la probabilità supera il 70%
- $P_1(p) = \frac{364}{365} \cdot \frac{363}{365} \cdot \ldots \cdot \frac{365-p+1}{365} = \frac{364!}{365^{p-1}(365-p)!}$



Come realizzare una funzione di hash

- Le chiavi possono essere di qualsiasi tipo e possono avere una lunghezza arbitraria
  - È necessario poter interpretare la loro rappresentazione in memoria come un numero

- Alcuni esempi, rispetto alla manipolazione di stringhe:
  - ord(c): valore binario del carattere c, in qualche codifica
  - bin(k): rappresentazione binaria della chiave k, concatenando i valori binari dei caratteri che la compongono
  - int(b): valore numerico associato al numero binario b
  - int(k)=int(bin(k))

Alcuni esempi

- bin(“ale”) = ord(“a”) ord(“l”) ord(“e”)
  01100001 01101100 01100101

- int(“ale”) = $97 \cdot 256^2 + 108 \cdot 256 + 101 = 6.384.741$

Funzione hash: estrazione

- L'estrazione consiste nel prendere una certa parte meno significativa della rappresentazione binaria della chiave come risultato della funzione hash
  - $N = 2^p$
  - $h(k) = int(b)$, dove b è un sottoinsieme di p bit presi presi da $bin(k)$
- Esempio:
  - $N = 2^{16} = 65536$, prendo i 16 bit meno significativi di $bin(k)$
  - $bin(\text{“}Alessandro\text{”}) =$ 01000001 01101100 01100101 01110011 01110011 01100001 01101110 01100100 **01110010** **01101111**
  - $h(\text{“}Alessandro\text{”}) = int($01110010 01101111$) = 29.295$

Funzione hash: estrazione

- Problema: la probabilità di collisione può essere molto elevata!
  - $h(\text{“}oleandro\text{”}) = int(\texttt{01110010 01101111}) = 29.295$

- La situazione non cambia effettuando un'estrazione in altri punti della stringa di bit

Funzione hash: XOR

- $N = 2^p$
- $h(k) = int(b)$, dove b è dato dalla somma modulo 2, effettuata bit a bit, di sottoinsiemi di p bit di $bin(k)$
- Può essere necessario *padding*
- Esempio con $\mathrm{N} = 2^{16}$

- bin(“accertare”) =

01100001	01100011	$\oplus$
01100011	01100101	$\oplus$
01110010	01110100	$\oplus$
01100001	01110010	$\oplus$
01100101	**00000000**	=
01110100	00000000	

- $h(\text{“}accertare\text{”}) =$ $int(0111010000000000)$ =29.696

- bin(“carcerate”) =

01100011	01100001	$\oplus$
01110010	01100011	$\oplus$
01100101	01110010	$\oplus$
01100001	01110100	$\oplus$
01100101	**00000000**	=
01110100	00000000	

- $h(\text{“}carcerate\text{”})=$ $int(0111010000000000)$ =29.696

Funzione hash: metodo della divisione

- N dispari, meglio se numero primo e distante da una potenza di 2
- $h(k) = int(b) \bmod N$

- Esempio con $N = 571$
- $h(\text{“}roma\text{”}) =$ 1.919.905.121 mod 571 = 416
- $h(\text{“}ramo\text{”}) =$ 1.918.987.631 mod 571 = 523
- $h(\text{“}orma\text{”}) =$ 1.869.770.081 mod 571 = 318
- $h(\text{“}orme\text{”}) =$ 1.869.770.085 mod 571 = 322

- NON vanno bene:
  - $N = 2^p$: vengono realmente utilizzati solo i p bit meno significativi
  - $N = 2^p - 1$: permutazioni di sottostringhe di lunghezza 2^p hanno lo stesso valore di hash

Gestione delle collisioni

Come gestire le collisioni

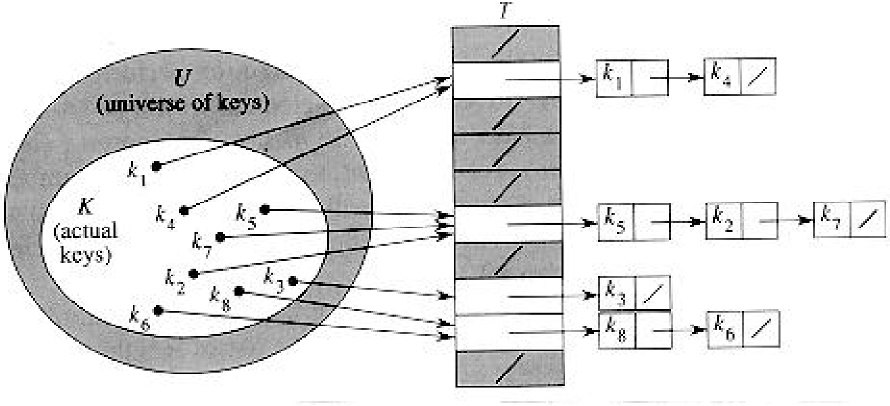
- L'hashing perfetto non è realizzabile in pratica
- È necessario quindi prevedere la possibilità di collisioni

- Cosa fare in caso di una collisione?
  - In caso di inserimento, bisogna trovare una posizione alternativa per gli elementi
  - In caso di ricerca, se non troviamo la chiave cercata, occorre cercarla altrove

- Queste operazioni aggiuntive dovrebbero avere un costo $O(1)$ nel caso medio

Liste di trabocco

- Le chiavi con lo stesso valore hash vengono organizzate in una *lista singolarmente collegata* (o vettore dinamico)
- Le operazioni fondamentali vengono realizzate tramite:
  - INSERT(): inserimento in testa
  - DELETE(): scansione
  - SEARCH(): scansione



Liste di trabocco: analisi della complessità

- Per analizzare la complessità, consideriamo il fattore di carico $\alpha = n/N$, dove n è il numero di elementi inseriti all'interno della tabella hash
- Assumiamo che il costo del calcolo di $h(\cdot) = \Theta(1)$

- **Caso pessimo**: tutte le chiavi collidono in un'unica lista
  - INSERT(): $\Theta(1)$
  - DELETE()/SEARCH(): $\Theta(n)$

Liste di trabocco: analisi della complessità

- **Caso medio**: La complessità dipende dalla distribuzione delle chiavi
  - Assumiamo hashing uniforme
  - La lunghezza di una lista è pari ad $\alpha = n/N$
  - INSERT() è sempre $\Theta(1)$

- Due possibilità per la ricerca:
  - Ricerca con successo: in media tocca metà delle chiavi in una lista. Il costo è $\Theta(1) + \frac{\alpha}{2}$
  - Ricerca con insuccesso: scandisce tutta la lista di trabocco. Il costo è $\Theta(1) + \alpha$

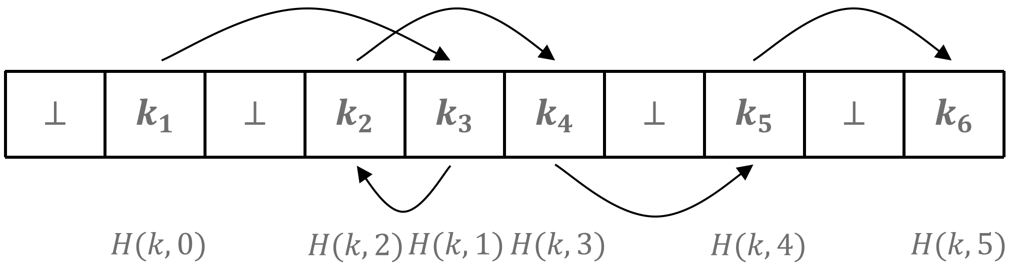
Liste di trabocco: analisi della complessità

- La complessità dipende quindi da α
- Se $n = O(N)$, allora $\alpha = O(1)$

- Con un appropriato dimensionamento della tabella hash, è possibile avere nel caso medio un costo $O(1)$ per tutte le operazioni
  - Intuitivamente, le liste di trabocco sono molto corte (mediamente, composte da un solo elemento)

Gestione delle collisioni: indirizzamento aperto

- Si vuole evitare l'utilizzo di strutture collegate per risolvere le collisioni
  - Tutte le collisioni devono essere risolte all'interno della tabella hash
- Si utilizza una *funzione hash espansa* $H(k, i)$, che calcola la posizione dell'elemento al "tentativo" i-esimo
  - Si genera una *sequenza di ispezione*
  - La tabella può andare in overflow (con $\alpha = 1$)



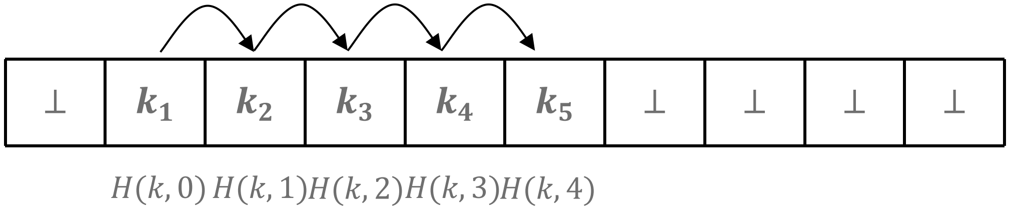
Sequenze di ispezione

- Le sequenze di ispezione determinano l'ordine con cui vengono “ispezionate” le posizioni successive nella tabella hash
- La funzione $H(k, i)$ deve permettere di ispezionare tutte le posizioni nella tabella
- Idealmente, una posizione nella tabella dovrebbe essere ispezionata una volta soltanto per chiave k
- Differenti tecniche:
  - Scansione lineare
  - Scansione quadratica
  - Hashing doppio

Scansione lineare

- La sequenza di ispezione è determinata dalla chiave dell'elemento che si vuole inserire, si cerca poi linearmente nelle posizioni adiacenti:

$$H(k,i) = (h(k) + c \cdot i) \bmod N$$



- Problema dell'*agglomerazione primaria* (primary clustering): si generano lunghe sottosequenze occupate
- I tempi medi per le operazioni che richiedono di effettuare una ispezione crescono molto

Scansione quadratica

- La sequenza di ispezione è determinata dalla chiave dell'elemento che si vuole inserire, si cerca poi linearmente nelle posizioni adiacenti:
$$H(k,i) = (h(k) + c \cdot i^2) \bmod N$$

- Sono possibili al più N sequenze di ispezione
- Problema dell'*agglomerazione secondaria* (secondary clustering): se $h(k_1) = h(k_2)$, le sequenze di ispezione per le due chiavi saranno le medesime

Hashing doppio

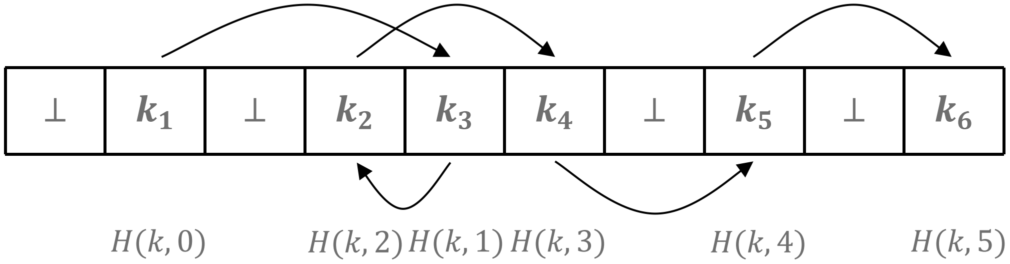
- Si utilizzano due funzioni hash differenti: una per determinare la posizione iniziale, una per generare la sequenza di ispezione:

$$H(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod N$$

- Sono possibili al più N^2 sequenze di ispezione
- Problema: è necessario scegliere $h_2(\cdot)$ in maniera tale da fornire un'ispezione completa
- Questo si ottiene se $h_2(k)$ ed N sono *coprimi* (ovverosia, il loro MCD è 1):
  - $N = 2^p$ e $h_2(k)$ restituisce sempre numeri dispari
  - N primo e $h_2 \colon \mathfrak{U} \longmapsto [0, N-1]$

Indirizzamento aperto: cancellazione

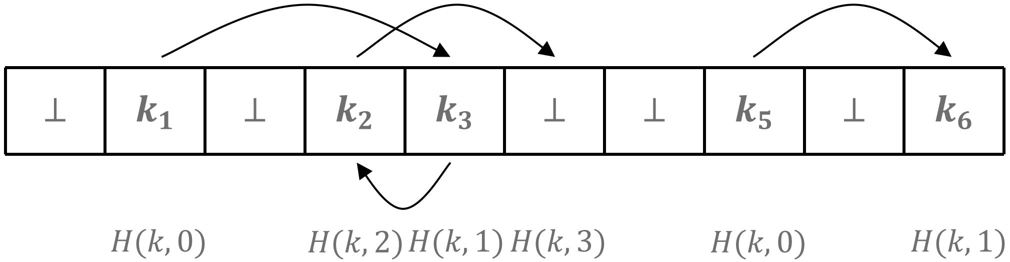
- La cancellazione deve essere gestita in maniera particolare nel caso dell'indirizzamento aperto



- Sequenza di operazioni:
  - SEARCH(k_2)

Indirizzamento aperto: cancellazione

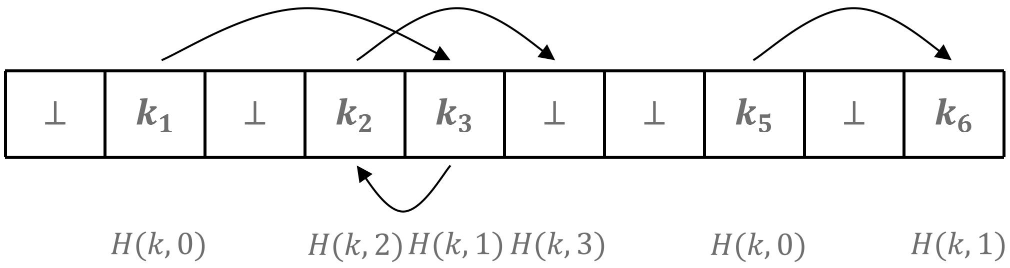
- La cancellazione deve essere gestita in maniera particolare nel caso dell'indirizzamento aperto



- Sequenza di operazioni:
  - SEARCH(k_2)
  - DELETE(k_4)

Indirizzamento aperto: cancellazione

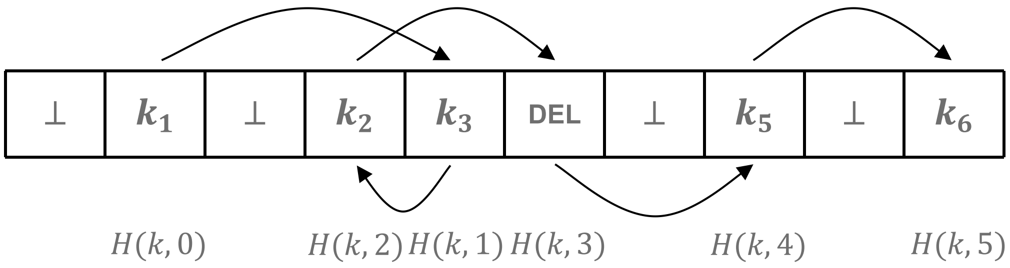
- La cancellazione deve essere gestita in maniera particolare nel caso dell'indirizzamento aperto



- Sequenza di operazioni:
  - SEARCH(k_2)
  - DELETE(k_4)
  - SEARCH(k_6): incontra un ⊥ e non trova l'elemento!

Indirizzamento aperto: cancellazione

- Occorre introdurre un elemento speciale DEL, per indicare che un elemento è stato cancellato



- Questo elemento va trattato con un elemento qualsiasi, anche se non deve mai essere restituito

Complessità

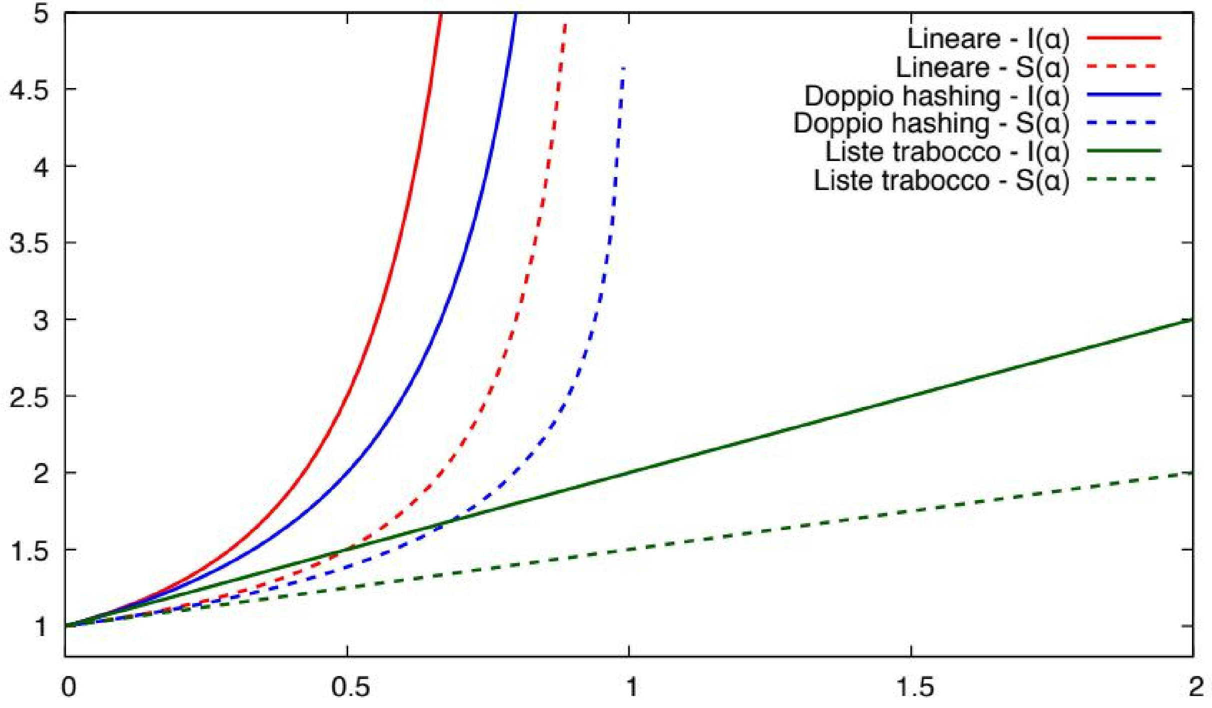


Tabella hash adattativa

- Indipendentemente dalla strategia per la gestione delle collisioni, non conviene far crescere troppo α
- Si può sostituire l'array con un array dinamico
  - Sopra un certa soglia di carico, si raddoppia la dimensione dell'array
  - Si reinseriscono tutte le chiavi
- Il costo è $O(1)$ ammortizzato
- Questa operazione permette di rimuovere anche tutti gli elementi DEL nel caso di indirizzamento aperto