

Corso di laurea in Informatica

Introduzione agli Algoritmi

Dizionari: Introduzione

Angelo Monti



SAPIENZA
UNIVERSITÀ DI ROMA

Dizionari

Un ***dizionario*** è una struttura dati che permette di gestire un insieme dinamico di dati, tramite queste tre operazioni:

- ***insert***: si inserisce un elemento;
- ***search***: si ricerca un elemento;
- ***delete***: si elimina un elemento.

Dizionari

Fra le strutture dati che abbiamo descritto, quelle che supportano in modo semplice (anche se non efficiente) tutte queste tre operazioni sono i vettori, le liste concatenate e gli alberi.

- le code (con o senza priorità, inclusi gli heap) e le pile non consentono né la ricerca né l'eliminazione di un elemento arbitrario.
- realizzare ad esempio un dizionario con un vettore ordinato permetterà di ricercare in tempo $O(\log n)$ e inserire e cancellare in tempo $O(n)$.
- realizzare un dizionario con alberi binari di ricerca **bilanciati** permetterà di ricercare inserire e cancellare in tempo $O(\log n)$

Quando l'esigenza è quella di realizzare un dizionario, si ricorre quindi a soluzioni specifiche.

Una **tabella hash** è una struttura di dati efficiente per implementare dizionari.

Sotto ragionevoli assunzioni, le tabelle hash implementano le tre operazioni di inserimento, cancellazione e ricerca con **complessità media costante** $\Theta(1)$.

Dunque, nel caso medio, implementare dizionari tramite tabelle hash risulta più efficiente che farlo su alberi binari di ricerca.

Qui di seguito illustreremo due diverse implementazioni di queste tabelle:

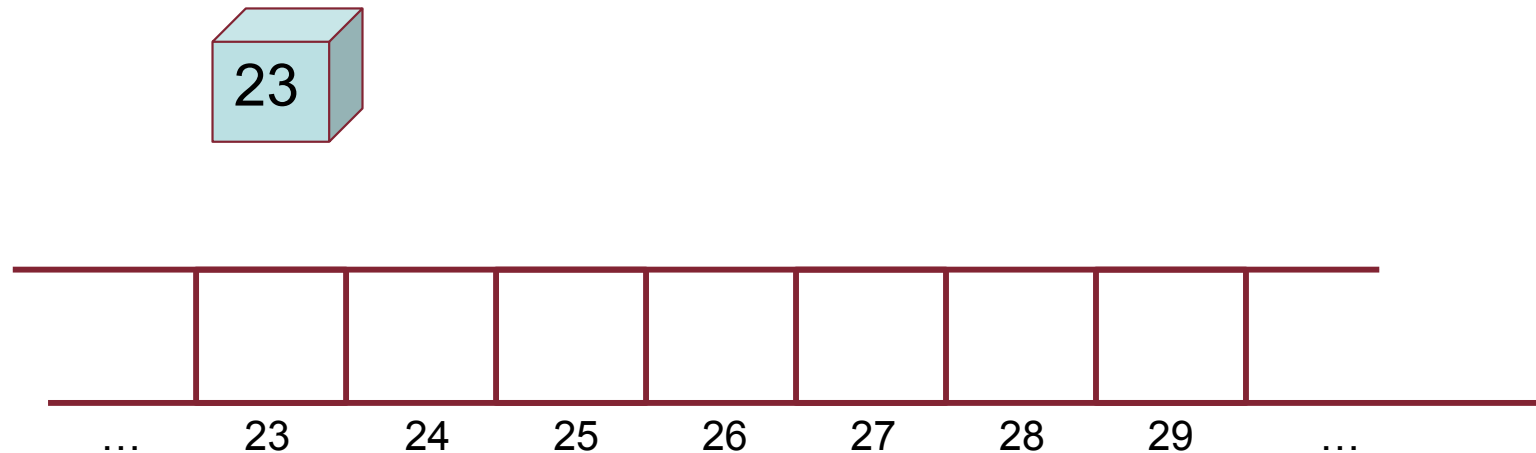
- **tabelle hash chiuso**
- **tabelle hash aperto**

Assunzioni e nomenclatura:

- Supponiamo di dover gestire un insieme dinamico i cui elementi abbiano chiavi (distinte) intere in $U = \{0, 1, \dots, m - 1\}$ detto insieme universo.

Tabelle ad indirizzamento diretto

Sotto le ipotesi appena viste viene naturale rappresentare il dizionario tramite un vettore ad indirizzamento diretto $V[0, 1, \dots, m-1]$ in cui ogni posizione corrisponde ad una chiave dell'universo U .



Abbiamo così una corrispondenza biunivoca tra le chiavi e le posizioni del vettore.

Tabelle ad indirizzamento diretto

E' un'implementazione naturale e di grande efficienza. Infatti, con questa scelta tutte e tre le operazioni hanno costo computazionale $\Theta(1)$:

```
def Insert_ID(V, k):  
     $V[k] =$  dati dell'elemento di chiave k
```

```
def Search_ID(V, k):  
    return  $V[k]$ 
```

```
def Delete_ID(V, k):  
     $V[k] = None$ 
```

Tabelle ad indirizzamento diretto

Purtroppo le cose non sono così semplici nel caso dei problemi reali, poiché:

- l'insieme U può essere tanto grande da rendere impraticabile l'allocazione in memoria di un vettore V di sufficiente capienza;
- il numero delle chiavi effettivamente utilizzate può essere molto più piccolo di $|U|$: in tal caso vi è un rilevante spreco di memoria, in quanto la maggioranza delle posizioni del vettore V resta inutilizzata.

Tabelle ad indirizzamento diretto

Ad esempio, si pensi al caso dei Codici Fiscali.

Un CF è costituito da 8 lettere (più una lettera di controllo) e 7 cifre. Considerando un alfabeto di 26 lettere si hanno:

$$|U| = 26^8 \cdot 10^7 > > 2 \cdot 10^{17}$$

Mentre ci sono meno di $n = 6 \cdot 10^7$ cittadini. Anche considerando che non tutte le lettere e le cifre vengono usate in ogni posizione, la sproporzione è enorme.

In sostanza:

L'indirizzamento diretto è una tecnica semplice che funziona bene quando l'universo U delle chiavi è ragionevolmente piccolo ed il numero n di elementi da memorizzare è vicino al numero m delle possibili chiavi.

Perciò, si ricorre spesso a differenti implementazioni dei dizionari.

Tabelle ad indirizzamento diretto

Lo spazio allocato è indipendente dal numero di elementi effettivamente memorizzati e se le chiavi effettive sono molte di meno delle chiavi potenziali c'è un notevole spreco di spazio.

Possiamo misurare il grado di riempimento di una tabella introducendo il fattore di carico

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

Dove m è la dimensione della tabella mentre n è la parte effettivamente utilizzata dalle chiavi. Ad esempio, si pensi al caso di 100 studenti memorizzati tramite la loro matricola a 6 cifre si avrebbe

$$\alpha = \frac{100}{10^6} = 0,0001.$$

con enorme spreco di spazio.

L'uso di tabelle hash consente di ottenere un buon compromesso tra tempo necessario alle operazioni e memoria:

memoria richiesta: $\Theta(n)$

tempo di ricerca: $\Theta(1)$ **ma nel caso medio e non in quello pessimo**

Tabelle hash

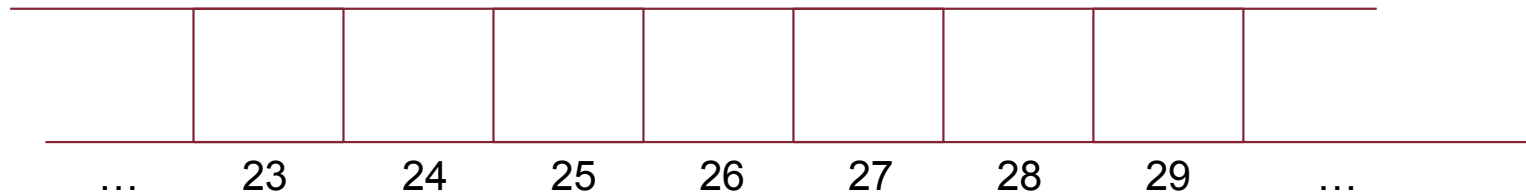
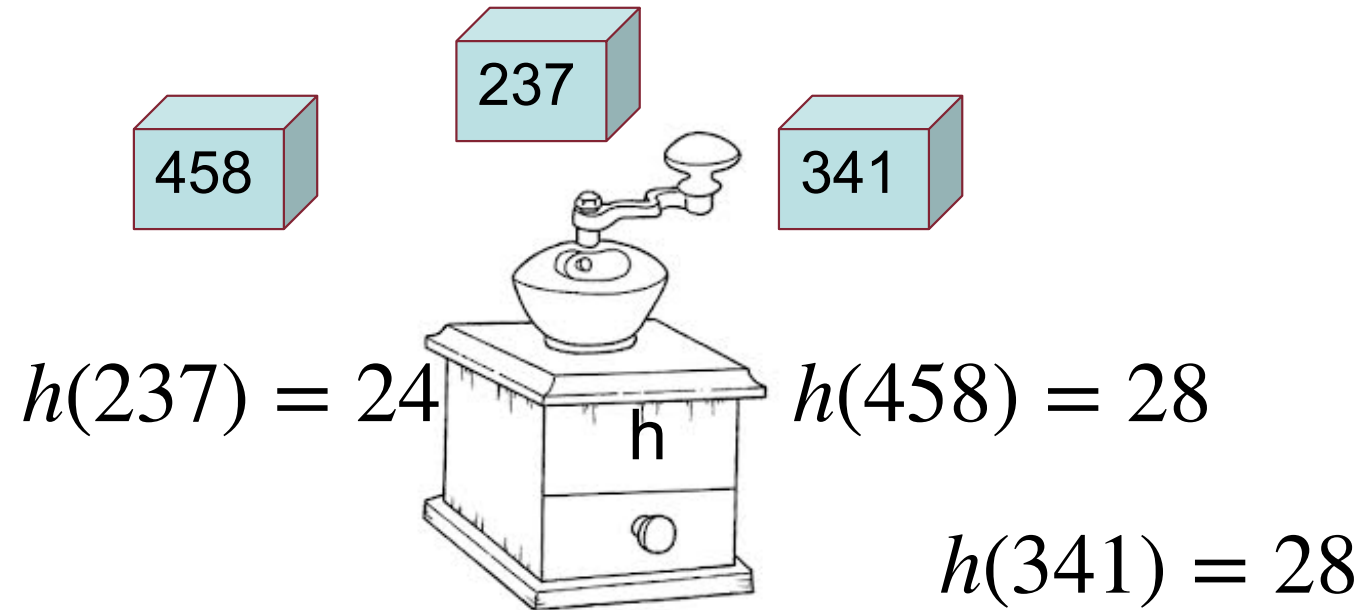
Dimensionare la tabella in base al numero di elementi **attesi** (che indicheremo con m) ed utilizzare una speciale funzione (funzione hash) per indicizzare la tabella

Una **funzione hash** è una funzione che data una chiave x restituisce la posizione della tabella in cui l'elemento con chiave x viene memorizzato:

$$h(x) \in \{0, 1, \dots, m - 1\}$$

la dimensione m della tabella può non coincidere con la dimensione dell'universo, anzi in generale si ha $m \ll |U|$

L'idea è quella di definire una funzione d'accesso che permetta di ottenere la posizione di un elemento data la sua chiave



Vantaggi:

- riduciamo lo spazio necessario per memorizzare la tabella

Svantaggi:

- perdiamo la corrispondenza tra chiavi e posizioni in tabella
- le tabelle hash soffrono del fenomeno delle **collisioni**

Funzioni hash e collisioni

Quando l'insieme U dei valori che le chiavi possono assumere è molto grande e la cardinalità m degli indici disponibili è invece molto più piccolo di U il verificarsi di collisioni è inevitabile

Nota poi che: anche se le chiavi da memorizzare sono meno di m , non si può escludere che due chiavi $x \neq y$ siano tali per cui $h(x) = h(y)$, ossia che la funzione hash restituisca lo stesso valore per entrambe le chiavi, che quindi andrebbero memorizzate nella stessa posizione della tabella.

Stando così le cose le **collisioni**, vanno evitate il più possibile e, altrimenti, risolte.

Le funzioni hash sono dunque funzioni matematiche che prendono in input un dato (come una stringa o un numero) e producono in output un valore hash, che è tipicamente un numero intero che fa riferimento ad un indirizzo all'interno di una tabella (tabella hash). L'obiettivo principale di una funzione hash è quello di distribuire uniformemente i dati in modo casuale all'interno di un range di valori possibili onde evitare il più possibile il verificarsi di collisioni.

supponiamo di avere una tabella di dimensioni m in cui vogliamo inserire record le cui chiavi siano stringhe. Una possibile funzione hash in questo caso sarebbe:

```
def mio_hash(s, size):  
    ''' Funzione hash personalizzata per le stringhe. '''  
    h = 0  
    for c in s:  
        h += ord(c) # Somma il valore ASCII di ogni carattere  
    return h % size
```

```
>>> mio_hash('Angelo Monti', 20)  
9
```

la funzione **mio_hash()** prende una stringa in input e calcola il suo hash sommando i valori ASCII di tutti i caratteri della stringa e poi calcolando il resto della divisione con la dimensione della tabella (quest'ultima operazione assicura che il valore restituito sia un indice all'interno della tabella).

Questa è solo una semplice illustrazione di come una funzione hash potrebbe funzionare ma potrebbe non essere adatta per tutti gli scopi.

Alcuni dei suoi svantaggi includono:

Collisioni: La funzione somma semplicemente i valori ASCII di tutti i caratteri nella stringa. Questo può portare a collisioni, ovvero situazioni in cui due stringhe diverse producono lo stesso valore hash. In applicazioni reali, è fondamentale ridurre al minimo le collisioni per garantire le prestazioni ottimali delle strutture dati che utilizzano le funzioni hash.

Distribuzione non uniforme: La somma dei valori ASCII non distribuisce uniformemente i dati. Alcune combinazioni di caratteri potrebbero produrre valori hash molto simili, mentre altre combinazioni potrebbero produrre valori hash molto diversi. Una buona funzione hash dovrebbe distribuire i valori in modo uniforme all'interno dello spazio degli indirizzi per minimizzare le collisioni e massimizzare l'efficienza delle strutture dati che li utilizzano.

Efficienza: La semplice somma dei valori ASCII di tutti i caratteri nella stringa può essere inefficiente per stringhe molto lunghe. Una funzione hash più efficiente dovrebbe essere in grado di calcolare l'hash in modo rapido, indipendentemente dalla lunghezza della stringa.

Tabelle hash

Una buona funzione hash deve essere in grado di distribuire uniformemente le chiavi nello spazio degli indici $\{0, 1, \dots, m - 1\}$.

La situazione ideale è quella in cui ciascuna delle m posizioni della tabella è scelta con la stessa probabilità, ipotesi che viene detta ***uniformità semplice della funzione hash***.

Usando una funzione hash che gode dell'uniformità semplice ogni cella ha la stessa probabilità di essere usata cioè non ci sono fenomeni di agglomerazione dove più chiavi tendono a collidere su alcune celle più che su altre.

Non vogliamo discutere qui come ottenere una funzione hash con l'ipotesi di uniformità semplice.

Partiamo, invece, dalla constatazione che le collisioni possono avvenire (anche se cerchiamo di renderle il più improbabili possibile!).

Infatti, per quanto bene sia progettata la funzione hash, è impossibile evitare del tutto le collisioni perché se $|U| > m$ è inevitabile che esistano chiavi diverse che producono una collisione.

Dobbiamo, quindi, risolverle.

Tabelle hash e collisioni.

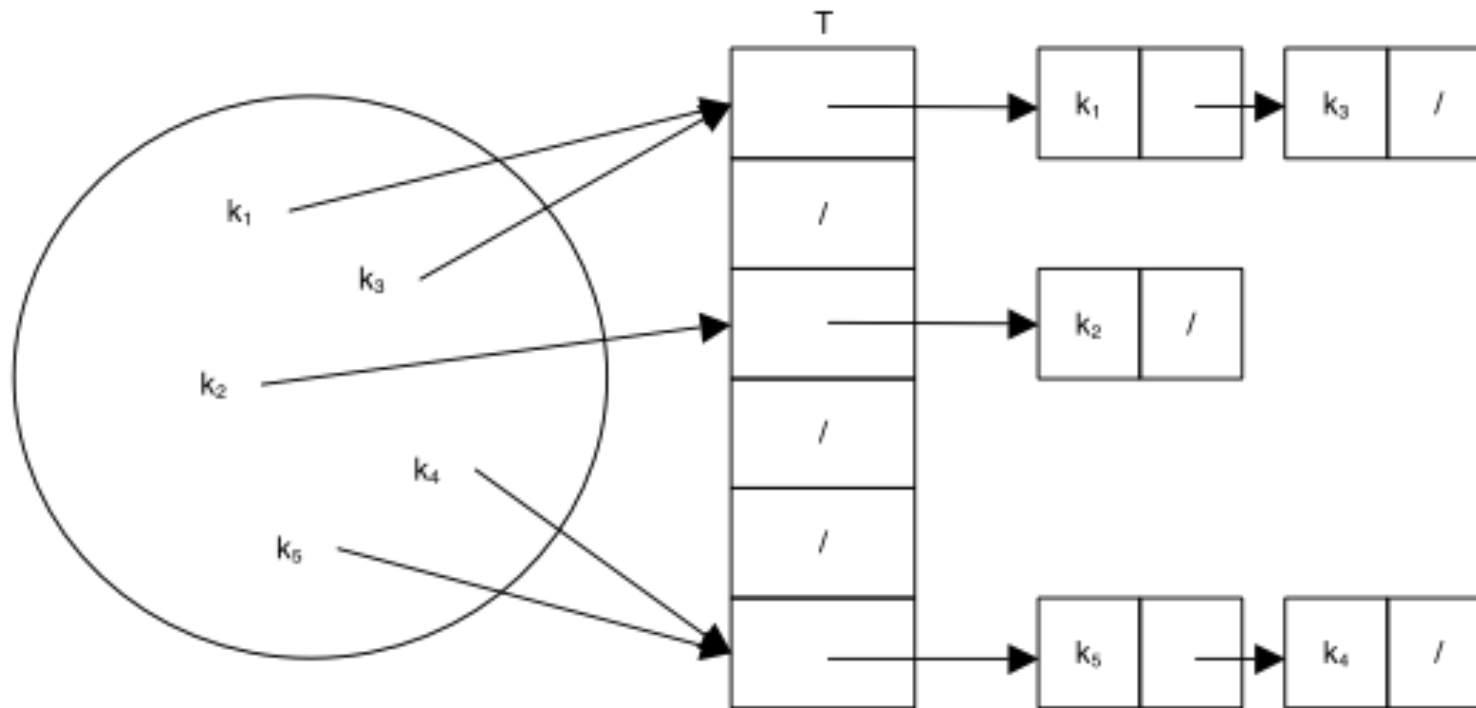
Esistono diverse tecniche per risolvere le collisioni in una tabella hash.

Nei lucidi che seguono illustreremo i due metodi più diffusi per la gestione delle collisioni:

- **indirizzamento chiuso (liste di trabocco)**
- **indirizzamento aperto**

Liste di trabocco

ad ogni cella della tabella hash si fa corrispondere invece di un elemento, una **Lista** (solitamente una **lista concatenata**) detta **lista di trabocco**. In questo modo un elemento che collide viene aggiunto alla lista corrispondente all'indice ottenuto.



Liste di trabocco, operazioni elementari (INSERIMENTO)

```
def Insert_LT(T, x):  
    #inserisci x nella lista puntata da T[h(chiave(x))]
```

Costo computazionale: $\Theta(1)$, anche nel caso peggiore, con l'inserzione in testa alla lista.

Liste di trabocco, operazioni elementari (RICERCA)

```
def Search_LT(T, k):  
    ricerca k nella lista di trabocco puntata da T[h(k)]  
    if k è presente:  
        return puntatore all'elemento contenente k  
    else:  
        return None
```

Costo computazionale: $O(\text{lunghezza della lista puntata da } T[h(k)])$

che, nel **caso peggiore**, diviene $O(n)$ quando tutti gli n elementi memorizzati nella tabella hash mappano nella medesima posizione, ma nel **caso medio** (quando la funzione hash gode di uniformità semplice) è $O\left(1 + \frac{n}{m}\right) = O(1 + \alpha)$

dove con α indichiamo il **fattore di carico** della tabella.

Nota che le tabelle hash con liste di trabocco sono un ottimo esempio di **bilanciamento spazio-tempo**:

- **per** $m = 1$ tutte le n chiavi sono in una sola lista e la tabella diventa un'unica lista a ricerca sequenziale che richiede tempo $T(n) = O(n)$ e spazio $O(n)$
- **per** $m = |U|$ possiamo usare una funzione hash perfetta ottenendo $T(n) = O(1)$ ma spazio $O(|U|)$.
- **per** $1 \leq m \leq |U|$, se la funzione hash gode di uniformità semplice allora abbiamo $T(n) = O\left(1 + \frac{n}{m}\right)$ e spazio $O(m + n)$.

Liste di trabocco, operazioni elementari (CANCELLAZIONE)

def Delete_LT (T, x)

cancella x dalla lista puntata da T[h(chiave(x))]

Costo computazionale: dipende dall'implementazione delle liste di trabocco e valgono, pertanto, tutte le osservazioni fatte per il costo dell'operazione di cancellazione nelle liste.

con liste semplici richiede lo stesso tempo della ricerca

Indirizzamento aperto

Questa tecnica prevede di inserire tutti gli elementi direttamente nella tabella, senza far uso di strutture dati aggiuntive.

Essa è applicabile quando:

$n \leq m$ (quindi per il fattore di carico si ha sempre $\alpha \leq 1$)

Supponiamo di voler inserire una chiave k e che la sua posizione "naturale" $h(k)$ sia già occupata. L'indirizzamento aperto consiste nell'occupare un'altra cella vuota, anche se essa potrebbe spettare di diritto ad un'altra chiave

Nota che in questo modo si usa meno memoria perché non c'è bisogno di puntatori.

Indirizzamento aperto

Idea: invece di seguire dei puntatori, calcoliamo (nei modi che vedremo fra breve) la sequenza delle posizioni da esaminare.

La funzione hash H dipende ora da 2 parametri: la chiave k e il numero di collisioni già trovate.

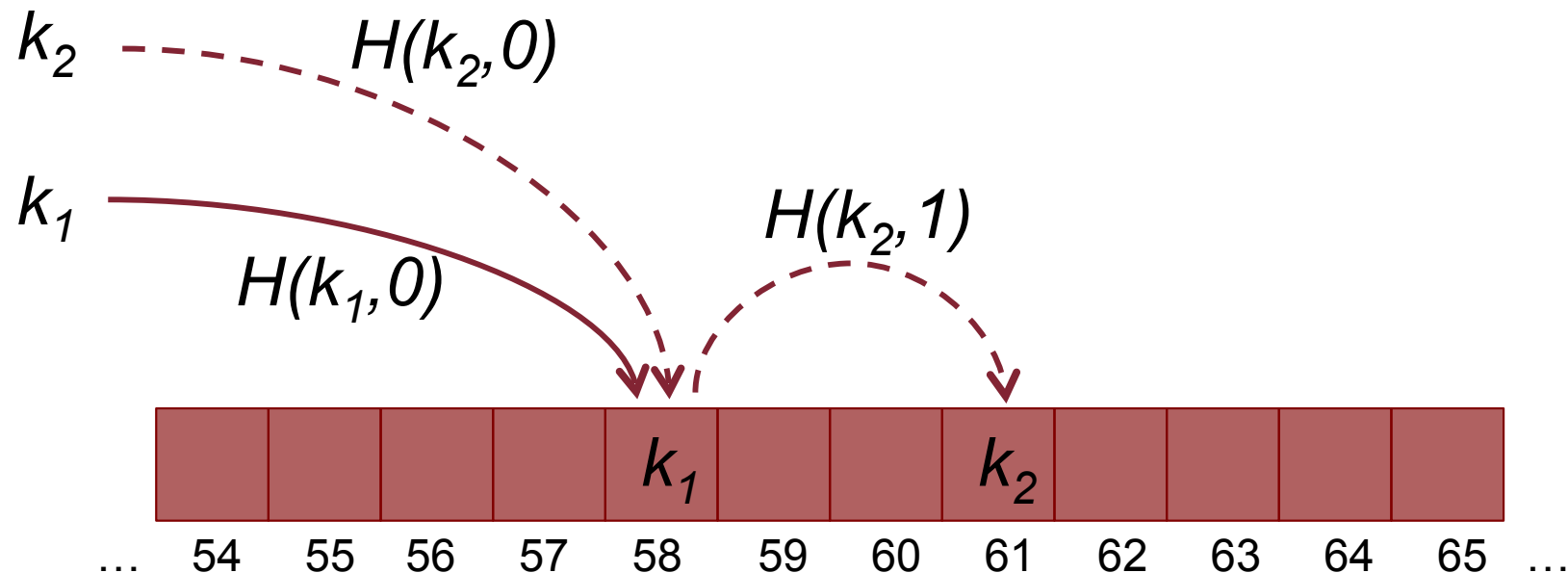
$$H : (U, i) \rightarrow \{0, m - 1\} \quad i = 0, 1, \dots, m - 1$$

Nota che la sequenza prodotta da H sui valori $0 \leq i < m$ deve contenere tutti gli indici $\{0, m - 1\}$ in modo da garantire che, se esiste una cella vuota, il metodo prima o poi la considera.

Indirizzamento aperto, operazioni elementari (Inserimento)

Se la posizione iniziale relativa alla chiave k è occupata, si scandisce la tabella fino a trovare una posizione libera nella quale l'elemento con chiave k può essere memorizzato.

La scansione è guidata dalla sequenza ben determinata: $H(k,0), H(k,1), \dots, H(k,m-1)$.



Indirizzamento aperto, operazioni elementari (Inserimento)

esistono diverse tecniche di scansione. Le più utilizzate sono ispezione lineare, ispezione quadratica e doppio hashing,

Ispezione lineare:

$$H(k, i) = (h(k) + i) \bmod m \quad i = 0, 1, \dots, m - 1$$

in pratica quando si incontra una collisione non si fa altro che utilizzare l'indice successivo a quello che collide sino a che non si trovi una casella libera.

Doppio hashing: (si utilizzano due funzioni hash)

$$H(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m, \quad i = 0, 1, \dots, m - 1$$

Per assicurare che la sequenza $H(k, i)$ contenga tutti gli indici è necessario che m e $h_2(k)$ siano primi tra loro. Ad esempio possiamo scegliere:

- m primo,
- $h_1(k) = k \bmod m$
- $h_2(k) = (k \bmod (m - 1)) + 1$

La bontà di questo metodo risiede nel fatto che, se le due funzioni sono ben progettate, è estremamente improbabile che due chiavi $k_1 \neq k_2$ producano una collisione su *entrambe* le funzioni hash (nel qual caso le due chiavi scandirebbero l'intera tabella nello stesso modo, situazione indesiderabile).

Indirizzamento aperto, operazioni elementari (ricerca)

L'operazione di **ricerca** è simile a quella di inserimento.

Se durante la scansione delle celle viene trovata una cella che contiene la chiave l'operazione restituisce l'elemento trovato. Se invece si arriva ad una cella vuota o si è scandita senza successo tutta la tabella si restituisce *None*.

Indirizzamento aperto, operazioni elementari (Cancellazione)

La prima cosa che verrebbe in mente è quella di marcare con *None* la cella dell'array che contiene la chiave. **Così facendo la ricerca non sarebbe più corretta** infatti la ricerca potrebbe interrompersi prima di trovare l'elemento cercato per via di un "buco" creato dalla cancellazione (in pratica potrebbe accadere che un elemento presente in tabella non venga trovato).

Idea: marchiamo le celle che hanno subito cancellazione con un valore speciale *canc*. L'inserimento considererà le celle marchate *canc* come vuote mentre una ricerca le oltrepasserà.

Operazione di cancellazione in definitiva è del tutto simile alla ricerca, l'unica differenza è che una volta trovata la cella da liberare la cella viene posta al valore speciale *canc*.

Analisi del costo di scansione

Sebbene usando l'indirizzamento aperto la scansione per la ricerca di un elemento ha tempo $O(n)$ nel caso peggiore, nel caso medio le cose vanno molto meglio.

Se le chiavi associate agli n elementi della tabella hash di m celle sono prese dall'universo U con probabilità uniforme, allora il numero medio di passi richiesto dall'operazione di ricerca (contando in n anche le celle marcate *canc*) è:

Esito ricerca	Scansione lineare	Hashing doppio / Scansione quadratica
Chiave trovata	$\frac{1}{2} + \frac{1}{2 - 2\alpha}$	$\frac{-\log_e(1 - \alpha)}{\alpha}$
Chiave non trovata	$\frac{1}{2} + \frac{1}{(2 - 2\alpha)^2}$	$\frac{1}{1 - \alpha}$

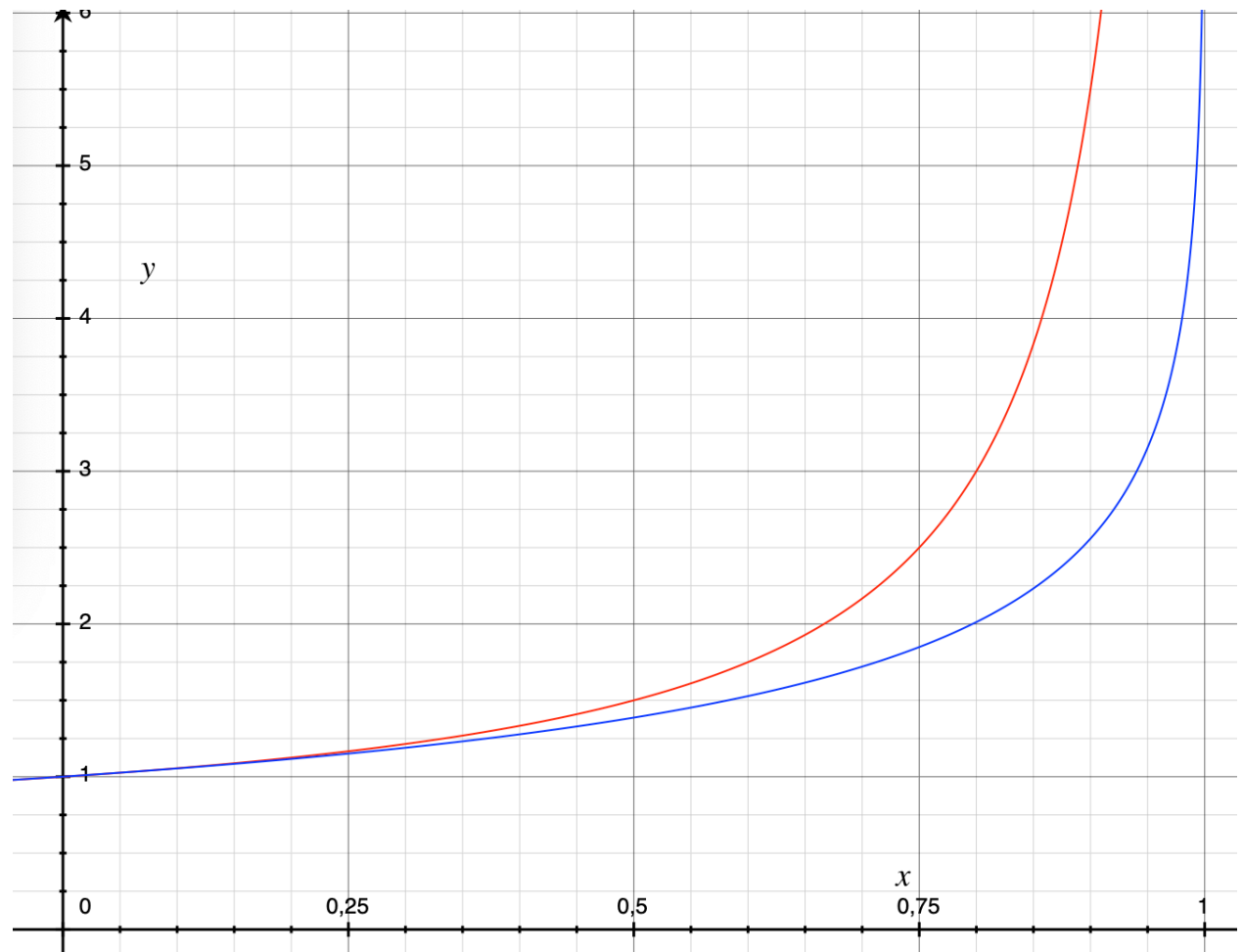
ricorda che è il fattore di carico della tabella: $\alpha = \frac{n}{m} < 1$

Nota che se solo una frazione della tabella è riempita e quindi $0 \leq \alpha < 1$ abbiamo che nel caso medio la ricerca richiede tempo costante.

Ricerca con successo:

$$y = \frac{1}{2} + \frac{1}{2(1-x)} \text{ scansione lineare}$$

$$y = -\frac{\ln(1-x)}{x} \text{ hashing doppio}$$



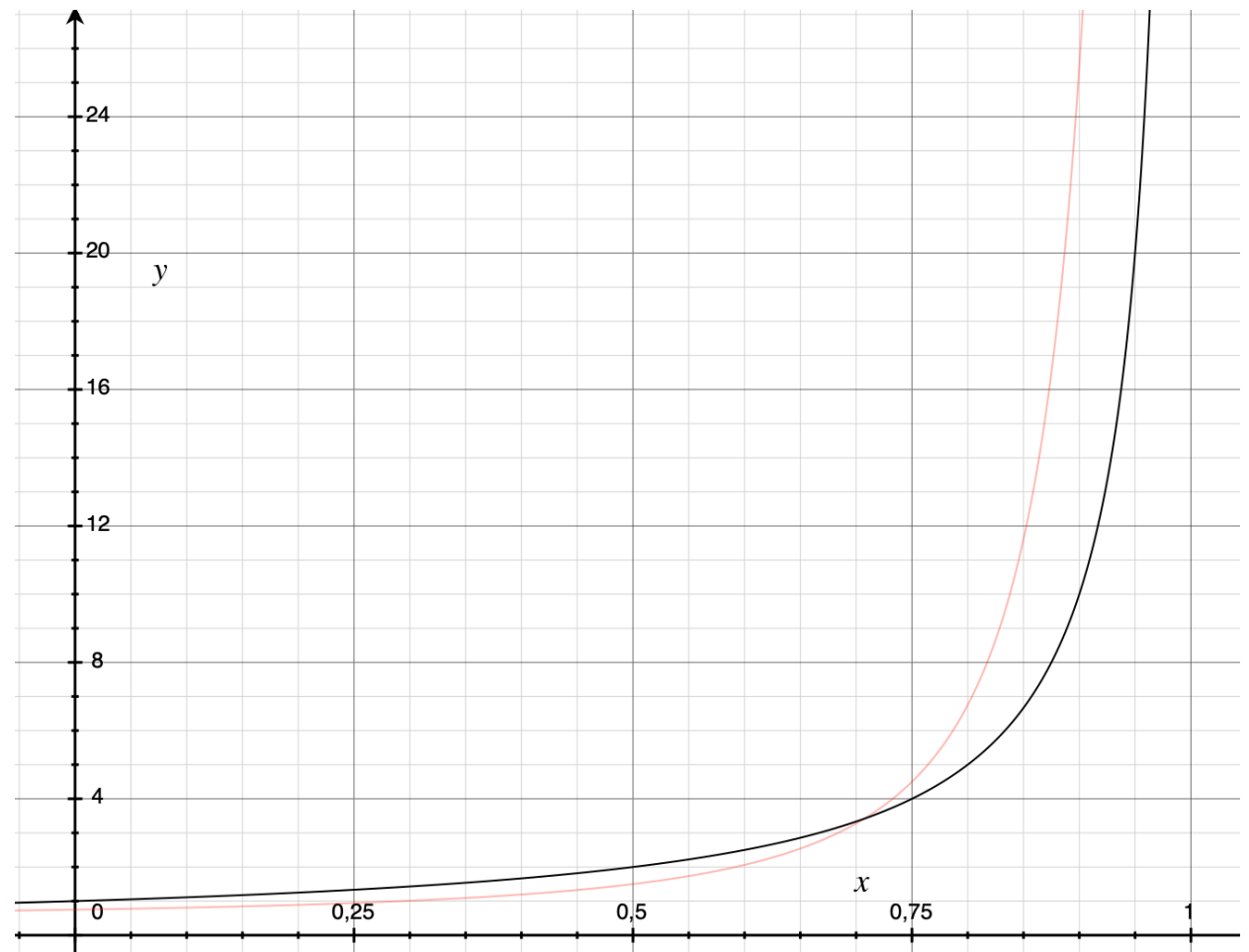
Ad esempio:

- se la tabella è piena al 50% il numero di passi atteso per una ricerca con successo è 1.5 per la scansione lineare e 1.3 per l'hashing doppio
- se la tabella è piena al 90% il numero di passi atteso per una ricerca con successo è 5.5 per la scansione lineare e 2.55 per l'hashing doppio.

Ricerca con insuccesso:

$$y = \frac{1}{2} + \frac{1}{4(1-x)^2} \text{ scansione lineare}$$

$$y = \frac{1}{1-x} \text{ hashing doppio}$$



Ad esempio:

- se la tabella è piena al 50% il numero di passi atteso per una ricerca con insuccesso è 1.5 per la scansione lineare e 2 per l'hashing doppio
- se la tabella è piena al 90% il numero di passi atteso per una ricerca con insuccesso è 26 per la scansione lineare e 10.5 per l'hashing doppio.

Analisi dei costi delle tre operazioni:

Esito ricerca	Scansione lineare	Hashing doppio / Scansione quadratica
Chiave trovata	$\frac{1}{2} + \frac{1}{2 - 2\alpha}$	$\frac{-\log_e(1 - \alpha)}{\alpha}$
Chiave non trovata	$\frac{1}{2} + \frac{1}{(2 - 2\alpha)^2}$	$\frac{1}{1 - \alpha}$

- Il costo dell'inserimento è al più quello di una ricerca nel caso di insuccesso.
- Il costo della cancellazione è dominato dal costo della ricerca della chiave da cancellare ed ha quindi lo stesso costo di una ricerca.

Possiamo concludere che: se solo una frazione della tabella è riempita, il tempo medio richiesto da ogni operazione di ricerca, cancellazione o inserimento è $O(1)$.

- In Python, l'implementazione dei dizionari utilizza una combinazione di indirizzamento aperto e liste di trabocco.
- **Indirizzamento Aperto:** Quando viene effettuata l'hash di una chiave e si verifica una collisione (cioè più di una chiave mappa allo stesso indice della tabella hash), Python non utilizza direttamente una lista di trabocco per risolvere il problema. Invece, cerca altri slot nella tabella hash fino a trovare uno vuoto o finché non esaurisce tutti gli slot possibili (questo è noto come "indirizzamento aperto lineare"). Questo approccio permette di evitare l'overhead delle liste di trabocco e di mantenere una maggiore compatezza della struttura dati.
- **Liste di Trabocco:** Tuttavia, se la tabella hash diventa troppo piena o ci sono troppe collisioni per una determinata chiave, Python passa automaticamente all'altra tecnica in cui ogni slot della tabella hash contiene un puntatore a una lista di tutte le coppie chiave-valore che hanno lo stesso hash. In questo modo, le collisioni sono gestite tramite una struttura di dati simile a una lista collegata.
- Questa combinazione di tecniche consente ai dizionari Python di bilanciare l'efficienza di accesso con la gestione efficiente delle collisioni, garantendo che l'accesso ai dati sia generalmente molto rapido, mantenendo allo stesso tempo una buona scalabilità.