1. Tabelle hash

Le *tabelle hash* sono un metodo per la realizzazione di dizionari e insiemi, alternativo agli alberi bilanciati visti nel capitolo precedente. L'idea di base è usare la chiave per calcolare direttamente la posizione in cui memorizzarla all'interno di un vettore. Con questo approccio, le operazioni principali (inserimento, ricerca e cancellazione) richiedono tempo costante $O(1)$ nel caso medio, anche se nel caso pessimo possono arrivare a tempo lineare $O(n)$.

Sia $\mathscr{U}$ l'*universo delle chiavi*, ovvero l'insieme di tutte le possibili chiavi distinte, e si supponga di voler memorizzare il dizionario in un vettore A di dimensione m.

Definizione Una *funzione hash* è una funzione $H : \mathscr{U} \to \{0, \ldots, m-1\}$ che mappa ciascuna chiave $k \in \mathscr{U}$ nell'*indice* $H(k)$ del vettore A destinata a contenere la coppia (k, v).

Idealmente, vorremmo disporre di una funzione hash *perfetta*, cioè una funzione iniettiva (per ogni coppia di chiavi distinte $k_1 \neq k_2$, risulti $H(k_1) \neq H(k_2)$). In questo modo, ciascuna chiave verrebbe memorizzata in una posizione distinta della tabella. Un esempio è mostrato in Figura 1.1.

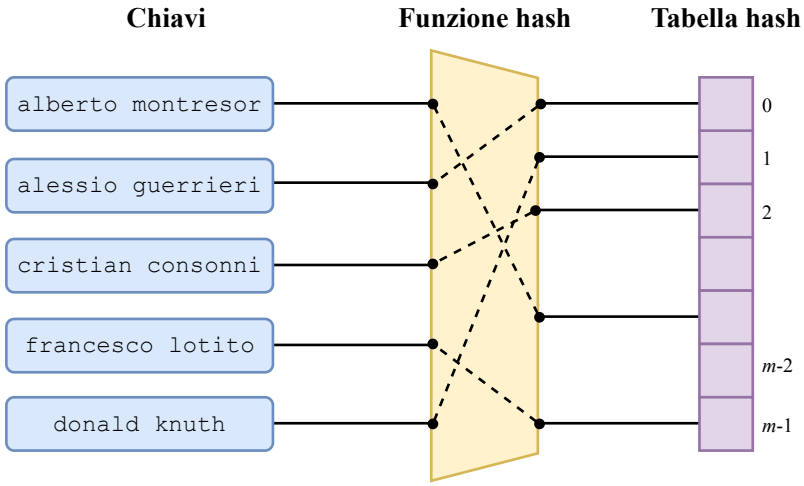


Figura 1.1: Un esempio di funzione hash perfetta.

Quando l'insieme $\mathscr{U}$ è un sottoinsieme numerico ristretto di $\mathbb{Z}_{\geq 0}$ (numeri interi positivi o nulli), è molto semplice realizzare una funzione hash perfetta: è sufficiente adottare la *funzione identità*, $H(k) = k$. In tal caso, si impiega un vettore A di dimensione $m = |\mathscr{U}|$, in cui la chiave k è memorizzata direttamente nella posizione $A[k]$. Questo approccio, noto come *tabella ad accesso diretto*, consente operazioni in tempo $O(1)$

Esempio (Ritardatreno) In una non precisata nazione, le ferrovie statali sono note per i loro ritardi. Invece di risolvere il problema, le ferrovie hanno deciso di mettere a disposizione degli utenti il sito `www.ritardatreno.it`, che dato un numero di treno, risponde con la sua posizione e il ritardo stimato. In questa nazione, i treni passeggeri sono numerati da 1 a 36,799, e ogni giorno viaggiano circa 10,000 treni. Visto che il sito è molto popolare, i progettisti hanno deciso di ottimizzare le ricerche mantenendo in memoria una tabella di dimensione $m = 36,799$, le cui posizioni sono riempite per poco meno di un terzo.

In molti casi pratici non è necessario utilizzare la funzione identità. È sufficiente impiegare una funzione relativamente semplice, ma che sia ancora iniettiva sull'insieme delle chiavi effettivamente usate. Se le chiavi occupano un intervallo numerico compatto, è possibile "traslare" l'intervallo per risparmiare memoria, senza perdere la proprietà di accesso diretto.

Esempio (Numeri telefonici) I numeri telefonici interni dell'Università di Trento appartengono all'intervallo $[28\,0000, 28\,9999]$. Per memorizzare informazioni associate a ciascun interno, è possibile utilizzare un vettore A di dimensione $m = 10,000$,utilizzando la funzione hash $H(k) = k - 280,000$ è iniettiva sull'intervallo considerato, e consente un accesso diretto efficiente, senza sprechi eccessivi di spazio.

Tuttavia, entrambe queste soluzioni presentano alcune limitazioni importanti:

- Se l'universo $\mathscr{U}$ è molto grande, non è possibile allocare un vettore di dimensione $|\mathscr{U}|$ per motivi di spazio di memoria;
- Anche se fosse possibile allocare sufficiente memoria per memorizzare $\mathscr{U}$, può succdere che numero effettivo di chiavi memorizzate nel dizionario sia molto inferiore a $|\mathscr{U}|$; in tal caso, la maggior parte delle posizioni del vettore resterebbe inutilizzata, con un conseguente spreco di memoria.

Esempio (Tabella dei simboli) I compilatori utilizzano un dizionario, detto *tabella dei simboli*, per memorizzare tutti gli identificatori (nomi di variabili, costanti, procedure, ecc.) definiti dall'utente all'interno del proprio codice. Questi identificatori vengono inseriti nel dizionario durante la compilazione delle "dichiarazioni", mentre la loro presenza viene verificata durante la compilazione delle "istruzioni eseguibili". In passato, molti compilatori ammettevano come simboli stringhe di al più 8 caratteri alfanumerici, senza distinzione tra maiuscole e minuscole, con il primo carattere alfabetico; in questo caso, $|\mathscr{U}| = 26 \cdot 36^7 > 2 \cdot 10^{12}$. È chiaramente impensabile utilizzare un vettore con oltre duemila miliardi di posizioni per un programma che conterrà, al più, qualche centinaio di identificatori! Inoltre, tutti i linguaggi moderni permettono identificatori di lunghezza illimitata, rendendo del tutto insensato l'uso di una tabella di dimensione $|\mathscr{U}|$.

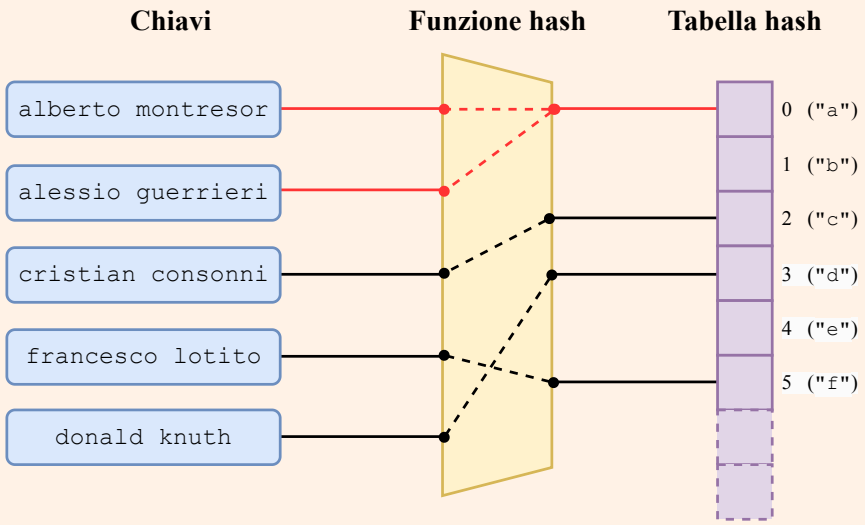
Per evitare inutili sprechi di memoria, la dimensione m del vettore non deve essere determinata sulla base della dimensione dell'intero universo $\mathscr{U}$, ma piuttosto in funzione del numero di chiavi *attese*, cioè la quantità $|K|$ di elementi che si prevede saranno effettivamente presenti nel dizionario in un dato momento.

Per contenere l'uso di memoria, è quindi necessario rinunciare all'iniettività della funzione hash H, ammettendo che chiavi distinte possano essere associate allo stesso indirizzo nel vettore: in altre parole, può accadere che $H(k_1) = H(k_2)$ anche se $k_1 \neq k_2$.

Definizione Si verifica una *collisione* quando due chiavi distinte $k_1 \neq k_2$ vengono mappate dalla funzione hash sulla stessa posizione del vettore, ovvero quando $H(k_1) = H(k_2)$.

Esempio (Collisione) Supponiamo di voler memorizzare i nomi di famosi informatici in una tabella hash di dimensione $m = 26$. Si consideri una funzione $H(k) = p$, dove p corrisponde alla posizione della prima lettera della chiave nell'alfabeto inglese, con numerazione a partire da 0. In altre parole, se la prima lettera è "`A`", allora $p = 0$; se è "`B`", allora $p = 1$; e così via fino a "`Z`", per cui $p = 25$.

La figura sottostante mostra un esempio di tale tabella, in cui due chiavi (evidenziate in rosso) sono mappate nella stessa posizione, provocando una collisione. Questo accade perché $H(\text{"}\texttt{albertoMontresor}\text{"}) = H(\text{"}\texttt{alessioGuerrieri}\text{"}) = 0$, in quanto entrambi i nomi iniziano con la stessa lettera.



L'esempio presentato mette in luce due aspetti fondamentali nella progettazione di una tabella hash:

- La *scelta della funzione di hash* è cruciale: una funzione mal progettata può distribuire le chiavi in modo molto sbilanciato, causando una concentrazione eccessiva di valori in alcune posizioni del vettore. Nell'esempio precedente, l'uso del solo carattere iniziale genera molte collisioni fra chiavi che iniziano con la stessa lettera (come la "`a`"), mentre altre posizioni rimangono quasi del tutto inutilizzate (come la "`x`").
- Anche scegliendo una buona funzione di hash, quando il numero di chiavi possibili supera il numero di posizioni disponibili nella tabella, le collisioni sono inevitabili. Per questa ragione, è essenziale progettare meccanismi efficienti per la *gestione delle collisioni*, che garantiscano comunque il corretto funzionamento delle operazioni di inserimento, ricerca e cancellazione.

Affronteremo questi due aspetti nelle prossime due sezioni.

1.1 Funzioni hash

Trovare una buona funzione hash non è semplice; essa deve ricavare l'output (un numero intero) che sia scorrelato dalla struttura dell'input (la chiave stessa). Da qui il termine "hash", che in inglese significa "tritare": si sminuzza la chiave in pezzettini che sono rimescolati in modo da formare un

intero che sia il più "casuale" possibile. Ovviamente l'algoritmo che calcola la funzione hash non è casuale, in quanto se si ricalcola più volte $H(k)$ per la stessa chiave k si ottiene sempre lo stesso indice, ma l'indirizzo hash si comporta da un punto di vista statistico come se fosse stato davvero prodotto con uno o più lanci casuali di una moneta.

> **Un po' di storia** Il termine "hash" deriva dal francese *hacher*, che significa "tritare" o "sminuzzare", a sua volta derivato da *hache*, "ascia". L'idea di utilizzare una funzione per "sminuzzare" le chiavi di un dizionario e distribuirle in modo uniforme in una tabella risale a un promemoria interno scritto da H. P. Luhn presso IBM nel gennaio 1953. Nonostante l'uso dell'idea si sia diffuso rapidamente in ambito industriale, il termine "hashing" non apparve con il significato attuale in pubblicazioni ufficiali fino alla fine degli anni Sessanta. Una delle prime trattazioni accademiche del tema si deve a Robert Morris, che nel 1968 pubblicò un influente articolo sull'argomento in *Communications of the ACM* [0].

La qualità di una funzione hash dipende in modo cruciale dalla sua capacità di distribuire uniformemente le chiavi all'interno della tabella. Una distribuzione sbilanciata porta a un aumento del numero di collisioni in alcune celle, con conseguente peggioramento delle prestazioni del dizionario. Per questo motivo, una buona funzione hash dovrebbe evitare di concentrare troppe chiavi in pochi indirizzi, favorendo invece una ripartizione il più possibile uniforme.

Uno dei criteri più semplici e diffusi per valutare la qualità di una funzione hash è l'*uniformità semplice*. Dato che le chiavi possono non avere tutte la stessa probabilità di comparire nel dizionario, si assume una distribuzione $P(k)$ su $\mathscr{U}$, dove $P(k)$ è la probabilità che la chiave k sia inserita nel dizionario. Da questa distribuzione, si deriva la probabilità $Q(i)$ che una chiave qualsiasi venga mappata nella posizione i della tabella.

Definizione Una funzione hash $H : \mathscr{U} \to \{0,\dots,m-1\}$ gode della proprietà di *uniformità semplice* se, per ogni posizione i della tabella, la probabilità che una chiave k venga mappata in i è uguale a $1/m$:

$$Q(i) = \sum_{k:H(k)=i} P(k) = \frac{1}{m}, \quad \forall i \in \{0,\dots,m-1\}$$

dove $P(k)$ è la probabilità che la chiave k venga inserita nel dizionario.

Nel resto della sezione presenteremo quattro funzioni hash di esempio, con l'obiettivo di evidenziare le principali difficoltà nella progettazione di funzioni hash realmente efficaci. Gli esempi sono volutamente semplici e didattici: la definizione di funzioni hash robuste richiede infatti competenze più avanzate e considerazioni legate al contesto applicativo. Per operare su chiavi generiche, ricondurremo ciascuna chiave k a una rappresentazione binaria, utilizzando le funzioni ausiliarie $bin()$, $ord()$ e $int()$ descritte in Tabella 1.1. Concluderemo infine con una breve panoramica sulle funzioni hash adottate nelle librerie dei principali linguaggi di programmazione.

Funzione	Descrizione
$bin(k)$	Rappresentazione binaria della chiave k; se k è una stringa, è ottenuta concatenando i bit di ciascun carattere
$ord(c)$	Restituisce il valore intero (codice ASCII o Unicode) del carattere c
$int(b)$	Restituisce il numero intero rappresentato dalla stringa binaria b

Tabella 1.1: Funzioni primitive utilizzate per la definizione di funzioni hash

1.1.1 Metodo dell'estrazione

Il metodo dell'estrazione è uno dei più semplici per definire una funzione hash su chiavi binarie. Si assume che la dimensione della tabella sia una potenza di due, ovvero $m = 2^p$, e si seleziona un blocco di p bit dalla rappresentazione binaria della chiave. L'indice hash è ottenuto convertendo questi bit in un intero.

$$H(k) = int(b), \quad \text{dove } b \subseteq bin(k),\ |b| = p$$

Esempio $m = 2^p = 2^{16} = 65536$; 16 bit meno significativi di $bin(k)$

$$\begin{aligned}
bin(\text{“Alberto”}) &= \texttt{01000001 01101100 01100010 01100101} \\
&\quad\ \texttt{01110010 01110100 01101111} \\
bin(\text{“Roberto”}) &= \texttt{01010010 01101111 01100010 01100101} \\
&\quad\ \texttt{01110010 01110100 01101111} \\
H(\text{“Alberto”}) &= int(\texttt{01110100 01101111}) = 29.807 \\
H(\text{“Roberto”}) &= int(\texttt{01110100 01101111}) = 29.807
\end{aligned}$$

Esempio $m = 2^p = 2^{16} = 65536$; 16 bit presi all'interno di $bin(k)$

$$\begin{aligned}
bin(\text{“Alberto”}) &= \texttt{01000001 01101100 01100010 01100101} \\
&\quad\ \texttt{01110010 01110100 01101111} \\
bin(\text{“Alessio”}) &= \texttt{01000001 01101100 01100101 01110011} \\
&\quad\ \texttt{01110011 01101001 01101111} \\
H(\text{“Alberto”}) &= int(\texttt{0001011011000110}) = 5.830 \\
H(\text{“Alessio”}) &= int(\texttt{0001011011000110}) = 5.830
\end{aligned}$$

Il metodo dell'estrazione è semplice ed efficiente, ma estremamente sensibile alla scelta dei bit selezionati:

- Se si estraggono i bit dal suffisso della chiave, chiavi che terminano allo stesso modo (come spesso accade con nomi propri o parole comuni) produrranno collisioni frequenti.
- Anche selezionare bit da altre parti della chiave può risultare inefficace, specialmente se le chiavi condividono prefissi o segmenti centrali comuni.
- Il metodo è poco robusto rispetto a variazioni minime delle chiavi, come anagrammi o suffissi simili.

1.1.2 Metodo dello XOR

Il metodo dello XOR sfrutta la somma bit a bit modulo 2 (operatore $\oplus$) tra sottoinsiemi della rappresentazione binaria della chiave. Anche in questo caso si assume che $m = 2^p$ e si costruisce una sequenza di p bit ottenuta combinando blocchi di $bin(k)$ tramite XOR. L'indice hash è quindi:

$$H(k) = int\left(\bigoplus_i b_i\right)$$

dove i blocchi b_i sono gruppi di p bit derivati dalla rappresentazione binaria della chiave, eventualmente completata con padding.

Esempio $m = 2^{16} = 65536$; 5 gruppi di 16 bit ottenuti con 8 zeri di "*padding*"

$bin(\text{“montresor”}) =$	$bin(\text{“sontremor”}) =$
01101101 01101111 $\oplus$	01110011 01101111 $\oplus$
01101110 01110100 $\oplus$	01101110 01110100 $\oplus$
01110010 01100101 $\oplus$	01110010 01100101 $\oplus$
01110011 01101111 $\oplus$	01101101 01101111 $\oplus$
01110010 00000000	01110010 00000000
$H(\text{“montresor”}) =$	$H(\text{“sontremor”}) =$
$int(0111000000010001) =$	$int(0111000000010001) =$
28.689	28.689

Il metodo dello XOR è semplice da implementare ed efficiente in termini di calcolo, ma presenta alcune criticità:

- È vulnerabile agli anagrammi: due chiavi composte dagli stessi caratteri in ordine diverso (es. `montresor` e `sontremor`) possono generare lo stesso indice hash, come visto nell'esempio.
- L'efficacia dipende fortemente dalla suddivisione in blocchi e dall'eventuale padding applicato.
- Non è adatto per chiavi molto corte o con caratteri ripetuti, dove l'operatore XOR può facilmente annullare parti significative dell'informazione.

1.1.3 Metodo della divisione

Il metodo della divisione è uno dei più diffusi per la costruzione di funzioni hash semplici ed efficaci. Consiste nel convertire la chiave in un intero e calcolare il resto della divisione per m:

$$H(k) = int(bin(k)) \bmod m$$

La scelta di m è fondamentale: si richiede che sia un numero dispari, preferibilmente un numero primo non troppo vicino a una potenza di due.

Esempio $m = 383$

$$\begin{aligned} H(\text{“Alberto”}) &= 18.415.043.350.787.183 \bmod 383 = 221 \\ H(\text{“Alessio”}) &= 18.415.056.470.632.815 \bmod 383 = 77 \\ H(\text{“Cristian”}) &= 4.860.062.892.481.405.294 \bmod 383 = 130 \end{aligned}$$

Il metodo della divisione è facile da implementare ed è efficace se m è scelto con cura. Alcune osservazioni:

- Se $m = 2^p$, solo i p bit meno significativi influenzano il risultato, trasformando il metodo della divisione nel metodo della divisione, rendendo così la funzone vulnerabile a collisioni su suffissi comuni.
- Se $m = 2^p - 1$ e le chiavi contengono caratteri in un alfabeto di dimensione 2^p, allora permutazioni degli stessi caratteri produrranno lo stesso hash.
- La scelta ideale è un numero primo lontano da potenze di 2 o di 10, come 383, 769, 1531, 12289, ...

Reality check Nel metodo della divisione è conveniente applicare la cosiddetta "regola di Horner" per la valutazione di un polinomio in un punto. Infatti, un qualsiasi polinomio

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

può essere riscritto come

$$p(x) = x(\cdots(x((a_k x) + a_{k-1})\cdots) + a_1) + a_0$$

Per valutare p nel punto x, si procede secondo tale parentesizzazione, dalla parentesi più interna verso quella più esterna. In questo modo il polinomio di grado q è valutato in x utilizzando soltanto q prodotti e q addizioni. La seguente procedura calcola la funzione hash per una chiave di ℓ caratteri utilizzando la regola di Horner ed effettuando l'operazione mod ad ogni iterazione, per evitare che b divenga troppo grande e provochi "overflow" (superamento della capacità di una parola di memoria).

int H(**char**[] k, **int** ℓ)

```
int b = ord(k[0])
for j = 1 to ℓ − 1 do
    b = ((b · 256) + ord(k[j])) mod m
return b
```

Ovviamente, per chiavi definite su alfabeti di dimensione diversa si userà una base diversa da 256. La complessità della funzione H è $O(\ell)$.

1.1.4 Metodo della moltiplicazione

Il metodo della moltiplicazione è un approccio efficace e generalizzabile per la costruzione di funzioni hash. Funziona per qualsiasi valore di m e si basa sulla seguente formula:

$$H(k) = \lfloor m \cdot (C \cdot i - \lfloor C \cdot i \rfloor) \rfloor$$

dove C è una costante reale fra 0 e 1 (spesso irrazionale), e $i = int(bin(k))$ è il valore intero associato alla chiave. ù La funzione $(C \cdot i - \lfloor C \cdot i \rfloor)$ corrisponde alla parte decimale di $C \cdot i$.

Esempio Sia $m = 2^{16} = 65536$ e $C = (\sqrt{5} - 1)/2 \approx 0.6180339887$

$$H(\text{“}\texttt{Alberto}\text{”}) = 65.536 \cdot 0.78732161432 = 51.598$$
$$H(\text{“}\texttt{Alessio}\text{”}) = 65.536 \cdot 0.51516739168 = 33.762$$
$$H(\text{“}\texttt{Cristian}\text{”}) = 65.536 \cdot 0.72143641000 = 47.280$$

Alcune osservazioni:

- Il metodo funziona per qualsiasi m, ma è particolarmente efficiente se $m = 2^p$.
- Il risultato dipende fortemente dalla scelta di C: valori irrazionali sono preferibili per garantire una buona distribuzione.
- È indipendente da particolarità della chiave (prefissi, suffissi), ed è meno vulnerabile a pattern ripetitivi.

Nel caso m sia una potenza di 2, è possibile calcolare i valori in maniera efficiente su architetture binarie utilizzando operazioni di shift e evitando operazioni in virgola mobile. Si supponga che la dimensione della tabella sia una potenza di due, $m = 2^p$, e che la chiave k sia stata trasformata in un intero $i = int(bin(k))$, con $i < 2^w$, dove w è la dimensione della parola di memoria (tipicamente

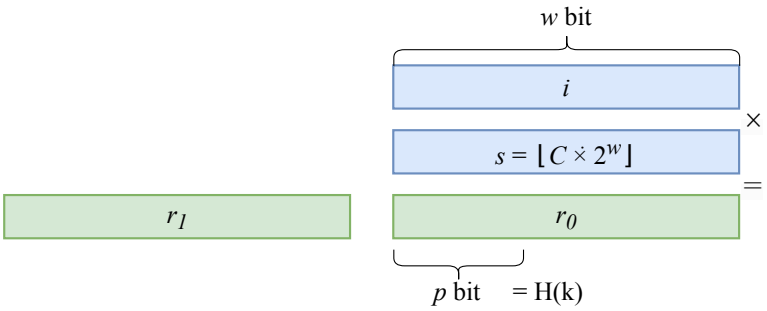


Figura 1.2: Metodo della moltiplicazioni implementato su architetture binarie.

$w = 32$ o $w = 64$). Si definisca il valore intero $s = \lfloor C \cdot 2^w \rfloor$ A questo punto si calcola il prodotto intero $i \cdot s$, che è un numero di $2w$ bit. Questo prodotto può essere diviso in due parti:

$$i \cdot s = r_1 \cdot 2^w + r_0$$

- r_1 contiene i bit più significativi, ovvero la parte intera del prodotto $i \cdot C$;
- r_0 contiene i bit meno significativi, cioè la parte frazionaria del prodotto $i \cdot C$;
- l'indice hash $H(k)$ è dato dai p bit più significativi di r_0.

Un po' di storia Il metodo della moltiplicazione è stato proposto da Donald E. Knuth in *The Art of Computer Programming*, vol. 3. La scelta della costante $C = (\sqrt{5}-1)/2$ è motivata dal fatto che questa quantità, nota anche come *coniugato della sezione aurea*, ha buone proprietà aritmetiche per la distribuzione dei bit.

Reality check Abbiamo descritto quattro metodi classici per costruire funzioni hash, scelti per la loro semplicità e per motivi didattici. Tuttavia, nella pratica le cose sono più complesse.

Per cominciare, anche metodi apparentemente robusti, come quello della moltiplicazione suggerito da Knuth, non garantiscono una distribuzione uniforme in tutti i contesti. In particolare, se le chiavi hanno una struttura regolare (come stringhe alfanumeriche), anche scelte "razionali" per i parametri possono portare a collisioni frequenti o schemi prevedibili.

Negli ultimi decenni, sono stati sviluppati test più sofisticati per valutare la qualità delle funzioni hash. Tra questi:

- *Avalanche effect*: una piccola modifica (ad esempio, un singolo bit) nella chiave dovrebbe cambiare in modo imprevedibile almeno la metà dei bit dell'indice hash. È un requisito fondamentale per evitare che chiavi simili vengano mappate in indirizzi simili.
- *Test statistici* come il *chi-square test* permettono di valutare quanto la distribuzione dei valori hash si discosta da quella uniforme, su insiemi di chiavi reali o sintetiche.

In contesti dove la sicurezza è fondamentale (ad esempio, firme digitali o identificatori non invertibili), si ricorre a *funzioni hash crittografiche*, come SHA-1 o SHA-256, progettate per rendere computazionalmente impossibile ricostruire la chiave originale a partire dall'hash o trovare due chiavi con lo stesso hash.

Infine, nell'uso quotidiano nei software moderni, si preferiscono funzioni hash **non crittografiche** ma robuste e veloci, come:

- FNV, semplice ed efficace per molte applicazioni.
- CityHash, progettato da Google per performance elevate su stringhe.
- FarmHash, il successore di CityHash, ancora più ottimizzato.

1.2 Gestione delle collisioni

Come abbiamo visto, quando due chiavi distinte vengono mappate dalla funzione hash nella stessa posizione della tabella, si verifica una *collisione*. Dal momento che le collisioni sono inevitabili, soprattutto quando l'universo delle chiavi è molto più ampio della dimensione della tabella, è fondamentale disporre di un meccanismo per gestirle in modo efficiente.

L'idea generale è la seguente: se la posizione calcolata tramite la funzione hash è già occupata, è necessario trovare una posizione alternativa in cui memorizzare la nuova chiave. Analogamente, durante una ricerca, se la chiave non si trova nella posizione attesa, bisogna cercarla nelle posizioni alternative secondo una strategia coerente con quella adottata in fase di inserimento.

Dal punto di vista computazionale, la gestione delle collisioni dovrebbe mantenere le operazioni di ricerca, inserimento e cancellazione in tempo $O(1)$ nel caso medio. Tuttavia, nel caso peggiore– ad esempio in presenza di molte collisioni o di una funzione hash mal progettata–il tempo può degradare fino a $O(n)$.

Nel seguito, presenteremo due approcci principali per la gestione delle collisioni:

- *Liste di trabocco*, o *memorizzazione esterna*, in cui ogni posizione della tabella mantiene una lista (o altra struttura dinamica) di elementi che vi sono stati mappati.
- *Indirizzamento aperto*, o *memorizzazione interna*, in cui tutte le chiavi vengono memorizzate direttamente nel vettore della tabella, seguendo una strategia per individuare una posizione alternativa libera.

1.2.1 Liste di trabocco

L'idea delle *liste di trabocco* (*chaining*) è semplice: ogni slot della tabella hash non contiene direttamente una chiave, ma un puntatore a una struttura dinamica (una lista collegata o un vettore dinamico) in cui vengono memorizzate tutte le chiavi che condividono lo stesso indice hash, come illustrato in Figura 1.3. Le operazioni principali vengono implementate come segue:

insert(k, v): la nuova chiave k viene inserita all'inizio della lista corrispondente all'indirizzo $H(k)$;

lookup(k): si esplora la lista in corrispondenza di $H(k)$ alla ricerca della chiave e si restituisce il valore corrispondente;

remove(k): si esplora la lista in corrispondenza di $H(k)$ e si rimuove la coppia chiave-valore memorizzata nella lista.

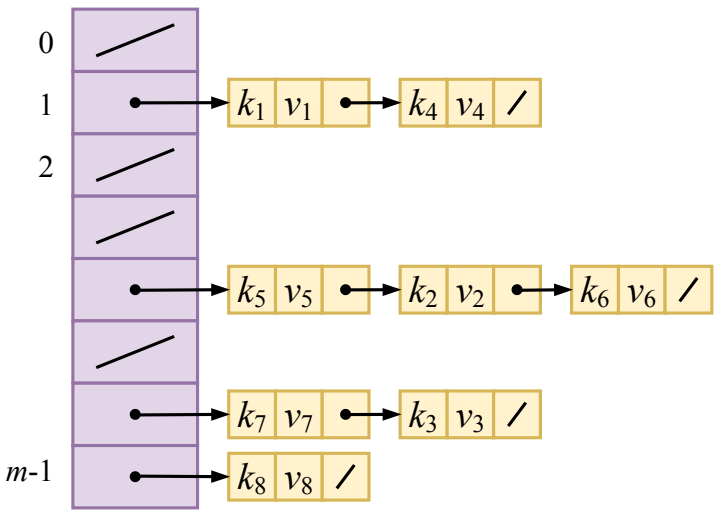


Figura 1.3: Le chiavi con lo stesso indice hash vengono memorizzate in una lista collegata.

1.2.2 Indirizzamento aperto

Nell'*indirizzamento aperto*, o *memorizzazione interna*, tutte le chiavi sono memorizzate direttamente nel vettore che costituisce la tabella hash. Quando la posizione $H(k)$ calcolata dalla funzione hash di base H è occupata da un'altra chiave, non si ricorre a strutture esterne, ma si prosegue la ricerca di una cella libera *all'interno* dello stesso vettore, secondo una sequenza di posizioni determinata da una strategia di *ispezione*.

Per descrivere questa strategia, estendiamo la funzione hash aggiungendo un secondo parametro, un indice i compreso fra 0 e $m-1$:

$$H : \mathscr{U} \times \overbrace{[0 \ldots m-1]}^{\text{Numero ispezione}} \to \overbrace{[0 \ldots m-1]}^{\text{Indice vettore}}$$

La funzione $H(k,i)$ rappresenta la posizione esaminata quando si cerca la chiave k alla i-esima ispezione, cioè dopo i tentativi falliti: durante un inserimento, i conta quante volte si è trovata una cella occupata; durante una ricerca, quante volte si è trovata una cella occupata da una chiave diversa da k. Un esempio di sequenza di ispezione è mostrato in Figura 1.4.

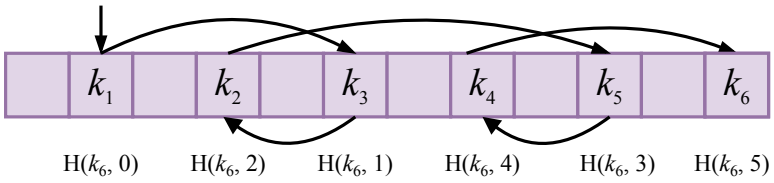


Figura 1.4: Esempio di una sequenza di ispezione composta da 6 passi.

La sequenza di ispezione $H(k,0), H(k,1), \ldots, H(k,m-1)$ deve visitare tutte le posizioni esattamente una volta; in altre parole, la sequenza deve costituire una permutazione di $\{0,\ldots,m-1\}$. Se il vettore è pieno e la chiave k non è presente, la ricerca può terminare solo dopo aver ricercato la chiave nell'intero vettore.

La situazione ideale prende il nome di *hashing uniforme*: ogni chiave ha la stessa probabilità di avere come sequenza di ispezione una qualsiasi delle $m!$ permutazioni di $\{0,\ldots,m-1\}$. Si tratta di una generalizzazione dell'*hashing uniforme semplice*, che richiede invece soltanto che la posizione iniziale $H(k)$ sia scelta uniformemente tra le m disponibili.

Nella pratica, implementare un vero hashing uniforme è difficile; ci si accontenta quindi di progettare la funzione di ispezione in modo che, per ogni chiave, la sequenza $H(k,0), H(k,1), \ldots$ sia almeno una permutazione dell'insieme delle posizioni disponibili.

Nel seguito analizzeremo tre metodi di ispezione che differiscono nel modo in cui viene costruita la sequenza di posizioni alternative a partire da $H(k)$.

Ispezione lineare: $H(k,i) = (H_0(k) + \delta \cdot i) \bmod m$
In questo metodo, si utilizza un parametro δ che rappresenta la distanza tra due posizioni successive esaminate durante l'ispezione. Affinché tutte le celle della tabella vengano visitate esattamente una volta prima di tornare a una posizione già considerata, è necessario che δ e m siano *coprimi* ($\gcd(\delta,m) = 1$). La divisione in modulo m garantisce che, quando si supera l'ultima posizione della tabella, l'ispezione riprenda dall'inizio, rendendo il processo *circolare*.

Questo metodo è semplice da implementare, ma presenta un limite importante: la tendenza a formare *agglomerati primari* di chiavi: quando una chiave collide con un elemento già presente, prosegue la ricerca fino a trovare la prima cella libera. Se la posizione iniziale della chiave cade all'interno di una sequenza di celle occupate, tutte queste verranno trovate già piene e la chiave finirà inevitabilmente per essere inserita immediatamente dopo la sequenza, allungando ulteriormente la sequenza.

La Figura 1.5 illustra questo fenomeno nel caso $\delta = 1$ (ispezione a passo unitario). Supponiamo di inserire quattro chiavi in una tabella vuota:

1. k_1 viene collocata nella posizione $H(k_1)$.
2. k_2 ha $H(k_2)$ già occupata da k_1, quindi viene inserita nella cella successiva.
3. k_3 trova $H(k_3)$ occupata da k_2 (che a sua volta non è nella sua posizione ideale) e viene spostata alla cella immediatamente seguente.
4. k_4 deve saltare due celle occupate prima di trovare uno slot libero.

Questo fenomeno prende il nome di *agglomerazione primaria* (*primary clustering*) e porta alla formazione di sequenze sempre più lunghe di posizioni a distanza δ l'una dall'altra, il che rallenta le operazioni: uno slot vuoto che si trova dopo i slot già occupati nella stessa sequenza viene riempito con probabilità $(i+1)/m$.

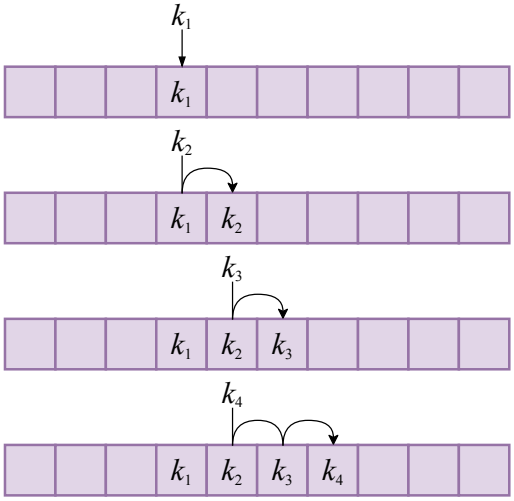


Figura 1.5: Esempio di formazione di un *agglomerato primario* con ispezione lineare a passo unitario ($\delta = 1$).

Ispezione quadratica: $H(k,i) = (H_0(k) + \delta \cdot i^2) \bmod m$
In questo metodo, m deve essere un numero primo e δ è un intero positivo. La distanza tra due posizioni successive nella sequenza di ispezione non è costante, ma cresce quadraticamente con i: dopo il primo passo si esamina la posizione a distanza $1h$, poi quella a distanza $4h$, poi $9h$, e così via. Questa variazione riduce l'effetto di *agglomerazione primaria* tipico dell'ispezione lineare, poiché due chiavi con indirizzi iniziali diversi ($H_0(k) \neq H_0(k')$) tendono a generare sequenze di ispezione che divergono più rapidamente.

Tuttavia, se due chiavi condividono lo stesso indirizzo iniziale ($H_0(k) = H_0(k')$), le sequenze di ispezione coincidono completamente, mantenendo il problema dell'*agglomerazione secondaria*. Un ulteriore limite è che la sequenza generata da $H(k,i)$ non è, in generale, una permutazione dell'insieme $\{0,\ldots,m-1\}$: può quindi accadere che alcune posizioni libere non vengano mai raggiunte, impedendo l'inserimento di nuove chiavi anche se la tabella non è piena. Questo svantaggio è meno rilevante quando m è sufficientemente grande rispetto al numero di elementi memorizzati.

Hashing doppio: $H(k,i) = (H_0(k) + H_\delta(k) \cdot i) \bmod m$
In questo metodo si impiegano due funzioni hash distinte: la funzione di base $H_0(k)$, che determina la posizione iniziale, e una seconda funzione $H_\delta(k)$, che definisce l'ampiezza del passo di ispezione. In questo modo, anche se due chiavi k e k' hanno lo stesso indirizzo iniziale ($H_0(k) = H_0(k')$), è molto probabile che generino passi di ispezione diversi ($H_\delta(k) \neq H_\delta(k')$). Questa caratteristica riduce drasticamente la probabilità di formazione di *agglomerati primari* e *secondari*.

Affinché la sequenza $H(k,0), H(k,1), \ldots, H(k,m-1)$ visiti tutte le posizioni della tabella senza ripetizioni, è necessario che il passo $H_\delta(k)$ sia *coprimo* con m ($\gcd(H_\delta(k), m) = 1$). Questa condizione può essere garantita, ad esempio:

- scegliendo $H'(k)$ sempre dispari, se m è una potenza di 2;
- imponendo che $H'(k)$ sia un intero positivo minore di m, se m è primo.

Cancellazione
Quando si utilizzano metodi di ispezione interna e si permette la cancellazione di chiavi, si presenta un ulteriore problema. Durante la ricerca di una chiave k, il raggiungimento di una cella "vuota" non garantisce che k non sia presente altrove nella tabella: quella posizione potrebbe infatti essere stata occupata al momento dell'inserimento di k e liberata solo in seguito, come mostrato in Figura 1.6(a).

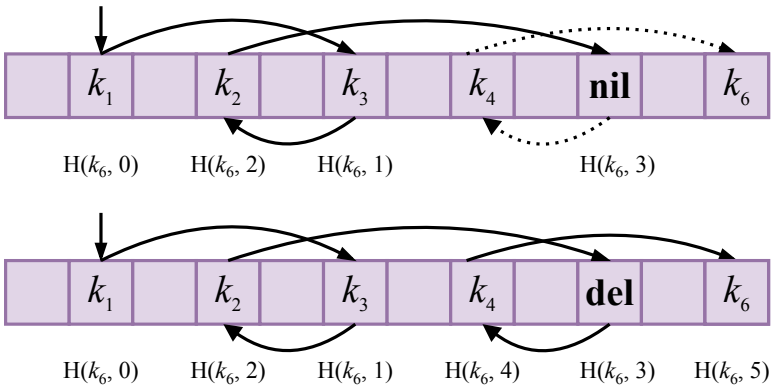


Figura 1.6: (a) Una sequenza di ispezione che viene interrotta dalla presenza di un elemento **nil**, nonostante la presenza della chiave k_5; (b) L'elemento **deleted** permette di continuare l'ispezione fino a raggiungere h_5

Per questo motivo, una posizione liberata da una cancellazione–pur essendo riutilizzabile per un successivo inserimento–non può essere trattata, durante l'ispezione, come una cella mai utilizzata. È necessario introdurre un *marcatore speciale* che indichi esplicitamente che la cella era occupata in passato ma è stata svuotata:

- le celle marcate possono essere riempite da nuovi inserimenti;
- durante una ricerca, tali celle non interrompono l'ispezione, che prosegue fino a trovare una cella mai usata o fino a tornare alla posizione iniziale, come mostrato in Figura 1.6(b).

L'uso di questo marcatore permette di preservare la correttezza della ricerca, ma comporta un aumento del tempo medio di ispezione, specialmente se il numero di celle cancellate cresce. Per applicazioni in cui siano previste frequenti cancellazioni, risulta in genere più conveniente adottare un metodo di gestione delle collisioni basato su liste di trabocco.

1.2.3 Implementazione

Siamo quindi pronti per presentare il codice di esempio di una tabella hash basata su indirizzamento aperto e hashing doppio, mostrato nella pagina successiva. La tabella è memorizzata mediante due vettori paralleli di dimensione m: *key*, che memorizza le chiavi, e *values*, che memorizza i valori associati. Ogni cella può contenere una chiave valida oppure uno dei due valori speciali **nil** e **deleted**, che indicano rispettivamente una posizione mai utilizzata e una posizione che era occupata in passato ma è stata cancellata.

La funzione principale scan$(k, insert)$ esplora la tabella alla ricerca della chiave k seguendo la sequenza determinata dall'hashing doppio. L'ispezione si interrompe se viene trovata la chiave, una cella vuota, oppure se sono state ispezionate tutte le posizioni. Durante la ricerca, se viene incontrata per la prima volta una cella cancellata, il suo indice viene salvato in *firstDeleted*. Se *insert* è **true** e la chiave non è stata trovata, l'indice restituito sarà quello della prima cella cancellata incontrata, se esiste; altrimenti, sarà quello della cella in cui l'ispezione si è arrestata.

Le operazioni del dizionario sono implementate come segue:

- lookup(k) chiama scan con $insert =$ **false** e restituisce il valore associato a k se presente, oppure **nil** in caso contrario.
- insert(k, v) chiama scan con $insert =$ **true** e, se la cella restituita è vuota, cancellata o contiene già k, memorizza la nuova coppia chiave–valore; in caso contrario, segnala che la tabella è piena.
- remove(k) cerca la chiave e, se presente, la sostituisce con il marcatore **deleted**.

```
HASH
item [ ] key                                              % Vettore delle chiavi
item [ ] values                                           % Vettore dei valori
int m                                                     % Dimensione della tabella

HASH Hash(int dim)
    HASH t = new HASH
    t.m = dim
    t.key = new item[0...dim-1] = {nil}
    t.values = new item[0...dim-1] = {nil}
    return t

int scan(item k, boolean insert)
    int firstDeleted = -1                                 % Prima posizione deleted
    int i = 0                                             % Numero di ispezione
    int j = H1(k)                                         % Posizione attuale
    while key[j] ≠ k and key[j] ≠ nil and i < m do
        if key[j] == deleted and firstDeleted == -1 then
            firstDeleted = j
        j = (j + H'(k)) mod m
        i = i + 1                                         % Prossima ispezione
    if insert and key[j] ≠ k and firstDeleted ≠ -1 then
        return firstDeleted                               % Riutilizza la prima cella cancellata trovata
    else
        return j                                          % Restituisci la cella trovata

item lookup(item k)
    int i = scan(k, false)
    if key[i] == k then
        return values[i]
    else
        return nil

insert(item k, item v)
    int i = scan(k, true)
    if key[i] == nil or key[i] == deleted or key[i] == k then
        key[i] = k
        values[i] = v
    else
        // Errore: tabella piena

remove(item k)
    int i = scan(k, false)
    if key[i] == k then
        key[i] = deleted                                  // Marca come cancellata
```

1.3 Complessità

Benché nel *caso pessimo* la ricerca di un elemento possa richiedere l'ispezione di tutte le posizioni della tabella, nella pratica il *caso medio* è molto più favorevole. Quando le chiavi sono distribuite in modo uniforme sulle posizioni della tabella, il tempo medio di ricerca dipende soltanto dal cosiddetto *fattore di carico*, ossia dal rapporto tra il numero di chiavi memorizzate e la dimensione della tabella stessa. In altri termini, a parità di distribuzione uniforme, tabelle di dimensioni diverse ma con la stessa percentuale di posizioni occupate presentano lo stesso tempo medio di ricerca.

Definizione Detto n il numero di chiavi memorizzate in una tabella e detto m la dimensione della tabella, il *fattore di carico* di una tabella hash è pari a $\alpha = n/m$.

Per esprimere il costo medio delle operazioni, definiamo due grandezze:

- $I(\alpha)$: numero medio di accessi alla tabella per una *ricerca con insuccesso*, cioè la ricerca di una chiave non presente.
- $S(\alpha)$: numero medio di accessi alla tabella per una *ricerca con successo*, cioè la ricerca di una chiave presente.

Queste grandezze sono direttamente collegate alla complessità media di lookup(), insert() e remove(): per inserire un elemento non presente si esegue una ricerca con insuccesso seguita dall'inserimento; per cancellare un elemento presente si esegue una ricerca con successo seguita dalla cancellazione. Pertanto, i tempi medi di inserzione e cancellazione dipendono, a meno di fattori costanti, da $I(\alpha)$ e $S(\alpha)$.

Tali considerazioni valgono però solo se l'hash distribuisce le chiavi in modo sufficientemente uniforme: una funzione hash scadente, che concentri molte chiavi nelle stesse posizioni, può far degenerare drasticamente le prestazioni, rendendo il tempo medio di ricerca paragonabile a quello del caso pessimo.

Assumendo quindi una distribuzione uniforme delle chiavi, le seguenti espressioni approssimano bene i valori medi di $I(\alpha)$ e $S(\alpha)$ per diversi metodi di ispezione:

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

1.3.1 Liste di trabocco

Il metodo delle liste di trabocco presenta due vantaggi principali: non pone limiti alla capacità del dizionario e previene la formazione di agglomerati. La tabella *key* contiene, per ciascun indice $\{0,\ldots,m-1\}$, un puntatore a una lista contenente le chiavi che hanno lo stesso indice hash.

Il costo medio di un'operazione è dato da un tempo costante $O(1)$ per l'accesso alla lista corrispondente, a cui si somma il costo medio della ricerca all'interno della lista stessa, pari a $I(\alpha)$ per una ricerca con insuccesso o $S(\alpha)$ per una ricerca con successo. Ad esempio, lookup(k) richiede l'accesso alla lista $key[H(k)]$ e l'ispezione della lista fino a trovare k o esaurire gli elementi; insert(k) aggiunge k in testa alla lista se non è già presente, mentre remove(k) la rimuove, se esiste. Un'implementazione efficiente può basarsi su un vettore di puntatori a liste collegate bidirezionali. La dimensione m della tabella va scelta in modo da evitare sia un eccesso di liste vuote (se m è troppo grande) sia liste troppo lunghe (se m è troppo piccolo).

1.3.2 Indirizzamento aperto

Il risultato per l'ispezione lineare è piuttosto complesso da dimostrare e non viene riportato in questa sede.

Per l'hashing doppio (e, indirettamente, anche per l'ispezione quadratica), la difficoltà principale sta nel dimostrare l'assenza di agglomerati. Assumiamo qui che tale proprietà sia verificata e consideriamo una ricerca con insuccesso.

Indichiamo con p_i la probabilità di ottenere esattamente i collisioni: il costo medio di una ricerca con insuccesso è

$$1 + \sum_{i=1}^{\infty} i \cdot p_i$$

dove il termine 1 corrisponde al costo per accedere all'unica posizione vuota che non genera collisione, e $p_i = 0$ per $i > n$, essendoci n chiavi memorizzate.

Per valutare la somma, si sfrutta la relazione

$$\sum_{i=1}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} q_i$$

dove q_i è la probabilità di avere almeno i collisioni. Si osservi che $q_1 = n/m = \alpha$, poiché si ha almeno una collisione se la prima posizione acceduta è occupata. Analogamente, $q_2 = (n/m) \cdot ((n-1)/(m-1))$, in quanto, dopo una prima collisione, si verifica una seconda collisione se la posizione successiva appartiene alle rimanenti $n-1$ occupate tra le $m-1$ ancora disponibili. Iterando il ragionamento si ottiene:

$$q_i = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1}.$$

Per m ed n grandi, si può maggiorare q_i con $(n/m)^i = \alpha^i$. Pertanto, per $0 \leq \alpha < 1$:

$$I(\alpha) \leq 1 + \sum_{i=1}^{\infty} \alpha^i = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

Il costo di una ricerca con successo corrisponde al costo della ricerca con insuccesso effettuata al momento in cui la chiave è stata inserita, poiché la sequenza di posizioni ispezionate è la stessa. All'atto dell'inserzione, però, il fattore di carico β era inferiore a quello corrente α. Facendo la media su tutti i possibili valori di β compresi tra 0 e α si ottiene:

$$S(\alpha) = \frac{1}{\alpha} \int_0^{\alpha} \frac{1}{1-\beta}\, d\beta = -\frac{1}{\alpha} \ln(1-\alpha).$$

Il confronto delle funzioni in Figura 1.7 evidenzia la netta superiorità del metodo delle liste di trabocco per $\alpha < 1$. Per $\alpha \geq 1$ questo metodo non è direttamente confrontabile con quelli a indirizzamento aperto, che operano solo per $\alpha < 1$.

Tra i metodi di ispezione, quello lineare è il meno efficiente, soprattutto quando α è vicino a 1. Per valori moderati di α (come $\alpha \leq 0.5$), anche l'ispezione lineare può risultare conveniente: in tal caso $I(\alpha) \leq 4$ e $S(\alpha) \leq 2$. È comunque evidente che, anche per gli altri metodi di ispezione, conviene mantenere il fattore di carico ben al di sotto di 1, poiché i tempi di ricerca crescono rapidamente con α. Inoltre, le cancellazioni non riducono il tempo di ricerca, poiché è comunque necessario esaminare le celle marcate con **deleted**. In presenza di frequenti cancellazioni, le liste di trabocco risultano in genere la scelta più efficiente.

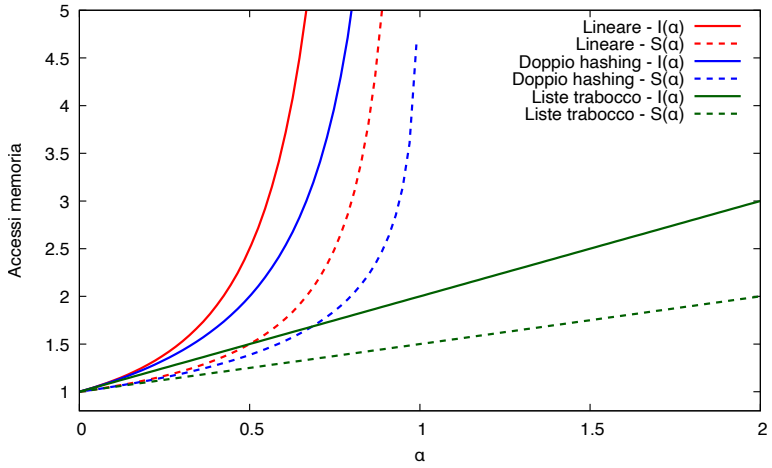


Figura 1.7: Numero medio di accessi per ricerche in tabelle hash.

1.4 Ristrutturazione

Utilizzando metodi di ispezione, è opportuno evitare che la tabella raggiunga un fattore di carico troppo elevato, poiché il tempo medio di ricerca cresce rapidamente all'aumentare di α. In pratica, quando in seguito a un'inserzione il fattore di carico supera una soglia prefissata (tipicamente $\alpha > 0.5$), conviene ristrutturare la tabella adottando le tecniche viste in precedenza.

La ristrutturazione consiste nel creare una nuova tabella di dimensione m pari al doppio della precedente, reinserendo tutte le chiavi presenti nella vecchia tabella (utilizzando una nuova funzione hash, poiché m è cambiato). In questo modo, il fattore di carico risulta al più dimezzato (non superiore a 0.25) e tutte le posizioni sono o libere o occupate: eventuali celle marcate come cancellate vengono eliminate.

Il costo di questa operazione è $O(m)$ nel caso pessimo, ma il *costo ammortizzato* delle operazioni lookup(), insert() e remove() rimane $O(1)$. Limitandosi per semplicità alle sole inserzioni e trascurando le cancellazioni, si può dimostrare che il costo complessivo di n inserimenti è limitato superiormente da $3n$, e quindi il costo ammortizzato è costante. La dimostrazione è analoga a quella riportata nella Capitolo 4.2.

Il principio si estende anche a sequenze di operazioni miste, comprendenti sia insert() sia remove(). In questo caso, oltre a espandere la tabella quando è troppo piena, può essere utile ridurla di dimensione (per esempio dimezzando m) quando α scende al di sotto di una soglia prefissata, come 0.25.

Ristrutturazioni analoghe, sia in espansione sia in contrazione, risultano convenienti anche per le tabelle con liste di trabocco; in questo caso, un criterio tipico è espandere la tabella quando α supera 2.

Reality check Finora abbiamo discusso metodi e complessità delle tabelle hash in termini astratti. Nelle librerie standard dei più comuni linguaggi, però, le implementazioni reali incorporano una serie di accorgimenti pratici per far fronte a problemi che emergono nella pratica.

In Java, tutte le classi ereditano i metodi `equals()` e `hashCode()` dalla classe `Object`. Se una classe non effettua l'override di `equals()`, il metodo `hashCode()` restituisce un valore calcolato a partire dall'indirizzo di memoria dell'oggetto, e due oggetti distinti hanno quasi certamente codici hash diversi.

Se invece una classe effettua l'override di `equals()` (come avviene, ad esempio, per `String`), *deve* anche ridefinire `hashCode()` in modo coerente. La documentazione ufficiale e la regola *Always override hashCode when you override equals* di Joshua Bloch (Effective Java, 2008) ribadiscono che, in assenza di questa coerenza, oggetti logicamente uguali possono finire in bucket diversi, compromettendo il corretto funzionamento della tabella hash.

Una cattiva implementazione di `hashCode()`, come quella seguente, porta tutte le chiavi nello stesso bucket, degradando il tempo di ricerca a $O(n)$.

```
public int hashCode() {
  return 0;
}
```

Questi problemi non sono esclusivi di Java: in C++, Python, C#, Go e in molti altri linguaggi, le prestazioni delle strutture dati hash dipendono in modo cruciale dalla qualità della funzione di hashing fornita per il tipo di chiave.

Reality check La tabella seguente riassume alcune implementazioni di tabelle hash in diversi linguaggi e librerie, evidenziando tecnica di gestione delle collisioni, fattore di carico tipico e note rilevanti.

Fino a Java 7, la classe `HashMap` usava liste di trabocco (`LinkedList`) per gestire le collisioni. Se la funzione di hashing era mal implementata, il caso pessimo degradava a $O(n)$ per operazione. Con Java 8, per ridurre l'impatto di funzioni hash di bassa qualità, le liste di trabocco vengono convertite in `TreeMap` (basate su alberi RB) quando superano una certa lunghezza. In questo modo, il caso pessimo passa da $O(n)$ a $O(\log n)$, a prezzo di un overhead di memoria più alto.

Linguaggio	**Tecnica**	t_α	**Note**
Java 7 `HashMap`	Liste di trabocco basate su `LinkedList`	0.75	$O(n)$ nel caso pessimo Overhead: $16n+4m$ byte
Java 8 `HashMap`	Liste di trabocco basate su RB Tree	0.75	$O(\log n)$ nel caso pessimo Overhead: $48n+4m$ byte
C++ `sparse_hash`	Ind. aperto, ispezione quadratica	?	Overhead: $2n$ bit
C++ `dense_hash`	Ind. aperto, ispezione quadratica	0.5	X byte per chiave-valore $\Rightarrow$ 2-3X overhead
C++ STL `unordered_map`	Liste di trabocco basate su liste	1.00	MurmurHash
Python	Indirizzamento aperto, ispezione quadratica	0.66	Implementazione ottimizzata per ridurre overhead di memoria

Esercizi

Esercizio 1.1. Si indichi il contenuto di una tabella hash di dimensione 11, inizialmente vuota, dopo l'inserzione, nell'ordine, delle chiavi: C, A, L, C, I, O, P, U, L, I, T, O. Si usi la funzione hash $H(k) = k \bmod 11$ per la k-esima lettera dell'alfabeto italiano e il metodo di ispezione lineare a passo unitario. Si indichi successivamente il contenuto della tabella dopo avervi cancellato, nell'ordine: G, A, L, L, I, A, N, I, e poi inserito, nell'ordine: R, I, V, E, R, A.

Esercizio 1.2. Si consideri una tabella hash di dimensione $m = 11$ inizialmente vuota. Si mostri il contenuto della tabella dopo aver inserito nell'ordine, i valori 33, 10, 24, 14, 16, 13, 23, 31, 18, 11, 7. Si assuma che le collisioni siano gestite mediante indirizzamento aperto utilizzando come funzione di hash $H(k)$, definita nel modo seguente:

$$H(k) = (h'(k) + 3i + i^2) \bmod m$$
$$h'(k) = k \bmod m$$

Mostrare il contenuto della tabella al termine degli inserimenti e calcolare il numero medio di accessi alla tabella per la ricerca di una chiave presente nella tabella.

Esercizio 1.3. Si definisca la realizzazione di un dizionario contenente in media 256 elementi, in cui la chiave richiede 8 bit, in modo da minimizzare il tempo di accesso.

Esercizio 1.4. Si forniscano procedure per realizzare i metodi di ispezione lineare e quadratico.

Esercizio 1.5. Un dizionario è realizzato mediante una tabella hash con liste di trabocco. Alcune chiavi sono ricercate molto più frequentemente delle altre, ma non si sa a priori quali siano tali chiavi. Quale accorgimento si può adottare per sperare di abbassare il tempo di accesso a queste chiavi?

Esercizio 1.6. Supponete di avere un albero di directory, contenenti un numero n di file, alcuni dei quali potrebbero essere replicati una o più volte, sotto nomi e percorsi diversi. Il vostro compito è elencare tutti i file replicati (ovvero, con lo stesso contenuto e la stessa lunghezza). Proponente un metodo efficiente ($O(n)$) per risolvere questo problema.

Esercizio 1.7. Supponete di cercare una chiave k^* in una lista concatenata di lunghezza n, dove ogni elemento contiene una chiave k ed un indice hash $H(k)$. Ogni chiave è una lunga stringa di caratteri. Come potreste trarre vantaggio dai valori hash quando ricercate nella lista un elemento con una data chiave? Proporre una procedura che effettui la ricerca di una chiave in maniera efficiente.

Esercizio 1.8. Utilizzando il metodo della divisione, si suggerisce di evitare valori vicini a potenze di 2. Si consideri una stringa di caratteri scelti da un alfabeto di dimensione 2^p, e si consideri una dimensione m per la divisione pari a $2^p - 1$. Il motivo è che, con questo alfabeto, due stringhe ottenute una da una permutazione dell'altra danno origine ad una collisione. Dimostrare come questo avvenga considerando lo scambio di due caratteri in una stringa; dallo scambio è poi possibile generalizzare al concetto di permutazione.

Esercizio 1.9. Progettare un algoritmo efficiente per identificare ed eliminare tutti gli elementi duplicati di un vettore non ordinato.

Esercizio 1.10. Supponiamo di mantenere ordinate le liste di trabocco. Quali sono i vantaggi e/o svantaggi di tale scelta in termini di efficienza?

Soluzioni

Soluzione Esercizio 1.1

0			
1	A	-	A
2	O	O	O
3	C	C	C
4	P	P	P
5			R
6			E
7	T	T	T
8	U	U	U
9	I	-	I
10	L	-	V

Soluzione Esercizio 1.3

Si osservi che con 8 bit si possono rappresentare in base 2 tutti i numeri compresi tra 0 e $2^8 - 1 = 255$. Ma allora ci si aspetta che nel dizionario siano presenti al massimo 256 chiavi. Utilizziamo quindi un vettore $A[0 \ldots 255]$ ad accesso diretto, dove la generica chiave k viene memorizzata direttamente in $A[int(bin(k))]$.

Soluzione Esercizio 1.5

Quando si accede ad una chiave, si può scambiarla di posto con quella che la precede nella lista di trabocco. In questo modo, le chiavi ricercate più di frequente tendono a portarsi verso la testa delle liste di trabocco, richiedendo un minor tempo di accesso.

Soluzione Esercizio 1.7

L'indice hash può essere utilizzato per evitare di dover confrontare tutte le chiavi con k^*, un'operazione pesante per via della lunghezza delle stringhe. Si confronta invece l'indice hash di k^* con l'indice hash di ogni chiave k incontrata; se $H(k) \neq H(k^*)$, k e k^* sono sicuramente diverse. Altrimenti, k è uguale a k^* oppure siamo in presenza di una collisione; è necessario quindi confrontare le chiavi.

Soluzione Esercizio 1.8

Si considerino le stringhe `ab` e `ba`; mostriamo che assegnando alle variabili a e b il valore numerico dei due caratteri, si dà origine ad una collisione.

$$
\begin{aligned}
& (2^p a + b) \bmod 2^p - 1 \\
= & ((2^p - 1)a + a + b) \bmod 2^p - 1 \\
= & (a + b) \bmod 2^p - 1 \\
= & ((2^p - 1)b + b + a) \bmod 2^p - 1 \\
= & (2^p b + a) \bmod 2^p - 1
\end{aligned}
$$