▼ 7.0 - Tabelle Hash

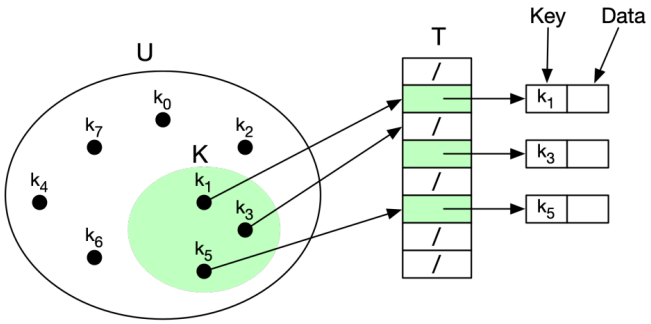
> Le **tabelle Hash** sono strutture dati che consentono di implementare dizionari con operazioni basilari (Search, Insert e Delete) estremamente efficienti. Tali operazioni effettuate su un dizionario implementato tramite tabella Hash possono avere un costo pessimo lineare, ma in media le prestazioni computazionali sono efficienti, addirittura sotto ragionevoli assunzioni probabilistiche le operazioni basilari hanno un costo costante.

Tale tipologia di implementazione non viene scelta universalmente, ma in base al dominio di applicazione. Analizzeremo tale scelta mettendola a confronto con l'implementazione tramite tabelle ad indirizzamento diretto, tenendo a mente che indicheremo con:

- U: universo di tutte le chiavi possibili.
- K: insieme di tutte le chiavi effettivamente utilizzate.

▼ 7.1 - Tabelle ad indirizzamento diretto

Tale implementazione è basata sull'utilizzo di un array di dimensione U in cui ogni chiave k è memorizzata all'interno di esso in posizione k.



Le operazioni basilari per un dizionario implementato tramite tabella ad indirizzamento diretto hanno il seguente **pseudocodice**:

```
Data search(HashTab T, Key k) {
  if (T[k] == null) return null
  else return T[k].data
}

void insert(HashTab T, Key k, Data d) {
  T[k].key = new node(k, d)
}

void delete(HashTab T, Key k) {
  T[k] = null
}
```

Il **costo computazionale** di una tale tabella è dunque il seguente:

- Costo in termini di **tempo** (tutte le operazioni hanno lo stesso costo): $O(1)$.
- Costo in termini di **memoria**: $O(U)$.

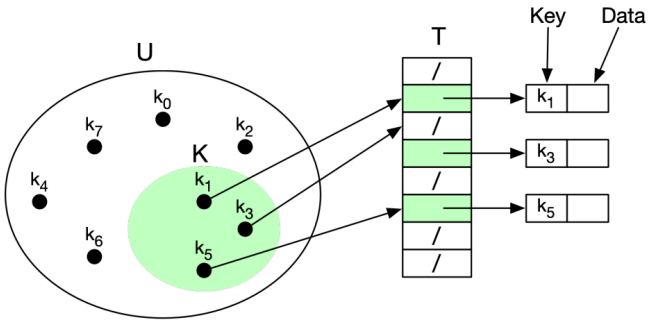
Notiamo dunque che il costo in termini di tempo è molto conveniente per tutte le operazioni basilari, ma la scelta di una tale implementazione risiede nella dimensione di K rispetto a U; se $K \approx U$, allora tale soluzione è accettabile, altrimenti, se $K \ll U$, c'è un grande spreco di memoria e tale soluzione risulta non accettabile.

Inoltre, nel caso in cui U è molto grande di per sè, può risultare impossibile memorizzare un tale array all'interno di un elaboratore. Ad esempio, se occorre utilizzare un dizionario in cui si utilizzano degli identificatori che contengono lettere e numeri e possono avere una lunghezza massima di 20 caratteri, l'universo U risulta grande $36^{20} \approx 10^{31}$, e nel caso in cui l'array utilizzato per implementare la struttura dati contiene dei puntatori con 4 bytes ognuno, allora la memoria totale utilizzata da un tale array è di circa $10^{31} \cdot 4$ bytes $> 10^{19}$ terabytes, impossibile da memorizzare all'interno degli elaboratori oggi in circolazione.

▼ 7.2 - Tabelle Hash

La tipologia di implementazione a tabella Hash viene dal problema che molte volte la dimensione dell'insieme K è molto minore rispetto a quello dell'insieme U. L'idea è quella di utilizzare un array T di dimensione m, con $m = \Theta(K)$, e una funzione Hash $h : U \to [0, \dots, m-1]$ che fornito un elemento da memorizzare restituisca il valore Hash $h(k)$ di tale elemento, il quale corrisponde all'indice da utilizzare per memorizzate l'elemento in posizione $T[h(k)]$.

Il problema di tale implementazione è il fatto che, siccome $U > m$, allora due o più elementi dell'insieme U possono avere lo stesso valore Hash, ottenendo così una **collisione**. Idealmente si vorrebbe utilizzare funzioni Hash che evitino collisioni, ma visto che queste sono inevitabili, occorre almeno minimizzarle.



Funzione Hash

Una buona funzione Hash deve soddisfare la proprietà di **uniformità semplice**, ovvero deve distribuire in maniera equiprobabile le chiavi negli m indici dell'array T. Infatti nel caso in cui certi indici vengono scelti con maggiore probabilità allora si otterrebbe un numero maggiore di collisioni.

Per costruire una funzione Hash che soddisfi tale proprietà occorre conoscere la **distribuzione di probabilità** con cui le chiavi sono estratte da U, purtroppo però conoscere tale distribuzione probabilistica è spesso irrealistico.

Assunzioni

Per discutere dell'implementazione di una tabella Hash assumiamo che tutte le chiavi hanno la stessa probabilità di essere estratte da U, in quanto è l'unico modo per proporre un meccanismo generale. Assumiamo inoltre, per analizzare i costi computazionali, che la funzione Hash possa essere calcolata in un tempo $O(1)$. Nel caso in cui invece una funzione Hash non sia $O(1)$, allora tale costo dominerebbe quello di tutte le operazioni basilari.

Inoltre sappiamo che è sempre possibile trasformare in un qualche modo una chiave k in un intero positivo al fine di ottenere un indice da utilizzare per indirizzare nell'array T. Un esempio banale può essere quello di utilizzare il numero decimale ottenuto dalla rappresentazione binaria ~~di k~~.

Funzione Hash: metodo della divisione

$$h(k) = k \bmod m$$

Esempi:

- $m = 12, k = 100 \implies h(k) = 4$.
- $m = 10, k = 101 \implies h(k) = 1$.

Il **vantaggio** di tale approccio è quello di utilizzare una funzione molto efficiente, in quanto composta da una singola operazione.

Lo **svantaggio** invece è quella di essere suscettibile a certi valori di m. Ad esempio infatti se $m = 10$, allora $h(k) =$ ultima cifra di k, oppure se $m = 2^p$, allora $h(k)$ dipende unicamente dai p bit meno significativi di k e non da tutti i bit. La soluzione a questo problema è quella di preferire un numero m diverso da una potenza di 2 e di 10, dunque sarebbe meglio un numero primo non troppo vicino a una potenza esatta di 2 e di 10.

Funzione Hash: metodo della moltiplicazione

$$h(k) = \lfloor m(kC - \lfloor kC \rfloor) \rfloor$$

Il metodo della moltiplicazione consiste nello scegliere una costante $0 < C < 1$, moltiplicare tale costante per la chiave e prendere la parte frazionaria, a questo punto moltiplicare quest'ultima per m e tenere la parte intera.

Il **vantaggio** è quello di poter scegliere un qualunque valore di m in quanto questo non è critico.

Lo **svantaggio** invece è il fatto che la costante C influenza la proprietà di uniformità della funzione. Per ottenere una buona proprietà di uniformità in tutti i casi viene suggerita una costante $C = (\sqrt{5} - 1)/2$.

Funzione hash: metodo della codifica algebrica

$$h(k) = (k_n x^n + k_{n-1} x^{n-1} + \cdots + k_1 x + k_0) \bmod m$$

dove:

- k_i è l'i-esimo bit della rappresentazione binaria di k, oppure l'i-esima cifra della rappresentazione decimale di k, o anche il codice ascii dell'i-esimo carattere.
- x è un valore costante.

Esempio:

- $k = 234, x = 3, m = 12 \implies h(k) = (2 \cdot 3^2 + 3 \cdot 3^1 + 4) \bmod 12 = 7$.

Il **vantaggio** di tale approccio è quello di far dipendere il risultato da tutti i bit/caratteri della chiave.

Lo **svantaggio** è invece il fatto di risultare in una funzione abbastanza costosa da calcolare, in quanto richiedere n addizioni e $n \cdot (n+1)/2$ prodotti, dove $n = \Theta(\log k)$.

È possibile comunque ovviare a questo problema utilizzando una regola algebrica, la regola di Horner, la quale afferma che un polinomio di grado n del tipo $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots +$

$a_1x + a_0$ può essere riscritto nel seguente modo $p(x) = a0 + x(a_1 + x(a_2 + x(\cdots + x(a_{n-1} + a_nx))))$.

Utilizzando tale regola è possibile abbassare il costo computazionale del calcolo della funzione Hash da quadratico a lineare sul numero di cifre della chiave k. Dato inoltre che il numero di cifre di una chiave è tipicamente un numero molto piccolo, possiamo assumere un costo **costante** per il calcolo della funzione Hash tramite il metodo della codifica algebrica.

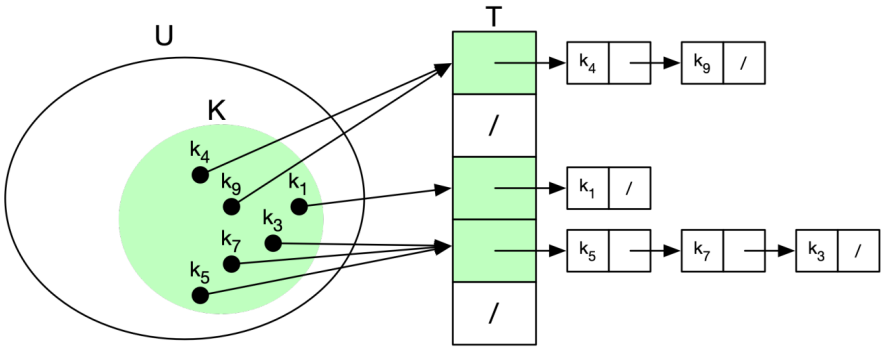
Problema delle collisioni

Abbiamo visto delle funzioni Hash che rispettano l'uniformità semplice, la quale riduce le collisioni ma non le elimina. È possibile inoltre notare che anche assumendo hashing uniforme semplice la probabilità che ci sia collisione tra due chiavi è molto alta (vedi teorema del compleanno).

Occorre dunque trovare delle tecniche di gestione di tali collisioni, noi vedremo quella del concatenamento e quella dell'indirizzamento aperto.

Concatenamento

La tecnica del concatenamento consiste nell'utilizzare la tabella Hash come un array di puntatori che puntano alla testa di liste contenenti tutte chiavi con lo stesso valore hash.



I **costi computazionali** delle principali operazioni su una lista Hash con metodo del concatenamento sono i seguenti ($L =$ lunghezza della lista di collisione più lunga, $\alpha = n/m$):

- **search** (ricerca lineare su lista concatenata)
  - Caso **ottimo**: $O(1)$.
  - Caso **pessimo**: $O(L)$.
  - Caso **medio**: $\Theta(1 + \alpha)$.

    Dimostrazione

    Sotto l'assunzione di hashing uniforme semplice ogni slot della tabella ha mediamente α chiavi.

    Una ricerca senza successo in una tabella hash con concatenamento ha dunque costo medio $\Theta(1 + \alpha)$, in quanto se k non compare nella tabella la ricerca visita tutte le chiavi nella lista concatenata $T[h(k)]$, che ha in media α chiavi. Quindi il costo computazionale è uguale al costo del calcolo della funzione hash, che possiamo assumere costante, sommato al tempo di visita della lista $T[h(k)]$, ovvero α: $\Theta(1 + \alpha)$.

    Una ricerca con successo in una tabella hash con concatenamento ha dunque costo medio $\Theta(1 + \alpha)$, in quanto se k compare nella tabella la ricerca visita in mediala metà delle chiavi nella lista concatenata $T[h(k)]$, che ha in media α chiavi. Quindi il costo computazionale è

uguale al costo del calcolo della funzione hash, che possiamo assumere costante, sommato al tempo di visita della lista $T[h(k)]$, ovvero $\alpha/2$: $\Theta(1 + \alpha/2) = \Theta(1 + \alpha)$.

- **insert** (inserimento in coda su lista concatenata)
  - Caso **ottimo**: $O(1)$.
  - Caso **pessimo**: $O(L)$.
  - Caso **medio**: $\Theta(1 + \alpha)$.
- **delete** (rimozione su lista concatenata)
  - Caso **ottimo**: $O(1)$.
  - Caso **pessimo**: $O(L)$.
  - Caso **medio**: $\Theta(1 + \alpha)$.

Indirizzamento aperto

L'idea nell'utilizzo dell'**indirizzamento aperto** è quella di, data una chiave k, se uno slot $T[h(k)]$ è già occupato, ispezionare altre celle della tabella alla ricerca di uno slot libero. Per decidere quali slot ispezionare viene estesa la funzione hash in modo che prenda in input anche il parametro "passo di ispezione":

$$h : U \times [0, \dots, m-1] \to [0, \dots, m-1]$$

Siccome tramite la sequenza di ispezione vogliamo visitare ogni slot della tabella solo una volta, tale sequenza deve fornire una permutazione degli indici della tabella.

Per questo motivo possiamo inoltre capire come il costo computazionale nel caso **pessimo** delle funzioni search, insert e delete è $O(m)$, mentre ciò che cambia a seconda della strategia di ispezione adottata è il costo medio.

Analizziamo 3 strategie di ispezione:

- Ispezione lineare
- Ispezione quadratica
- Doppio hashing

Ispezione lineare

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

dove $h'(k)$ è una funzione hash ausiliaria.

Il problema di questa strategia di ispezione è il **clustering primario**, ovvero il fatto di avere lunghe sotto-sequenze occupate, che diventano sempre più lunghe. I tempi medi di inserimento e cancellazione infatti crescono con il riempimento della tabella, e assumendo hashing uniforme semplice, uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$, quindi la probabilità di creare sotto-sequenze è molto più alta rispetto a quella di occupato uno slot vuoto preceduto da un altro slot vuoto.

Ispezione quadratica

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

dove $h'(k)$ è una funzione hash ausiliaria e $c_1 \neq c_2$.

Questa strategia di ispezione richiede che vengano scelte le costanti c_1 e c_2 in modo che venga garantita una permutazione degli indici della tabella.

Il problema di questa strategia è il **clustering secondario**, ovvero se due chiavi hanno la stessa ispezione iniziale, allora le loro sequenze di ispezione saranno identiche.

Doppio hashing

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

dove $h_1(k)$ e $h_2(k)$ sono la funzione hash primaria e secondaria. Occorre scegliere la funzione hash h_2 in modo da non dare mai in output il valore 0 e in modo da permettere di iterare su tutta la tabella.

Quando avviene una collisione si usa anche la funzione secondaria per determinare il successivo slot da ispezionare. Notiamo che se $h_1 \neq h_2$ è meno probabile che per una coppia di chiavi $a \neq b$, $h_1(a) = h_1(b)$ e $h_2(a) = h_2(b)$, dunque tramite questa strategia di ispezione viene evitato il clustering primario e secondario.

Indirizzamento aperto: analisi del numero di ispezioni della funzione search

I numeri di ispezioni della funzione search in una tabella hash con indirizzamento aperto nei diversi casi sono i seguenti ($\alpha = n/m$):

- Sequenze di ispezione **equiprobabili**

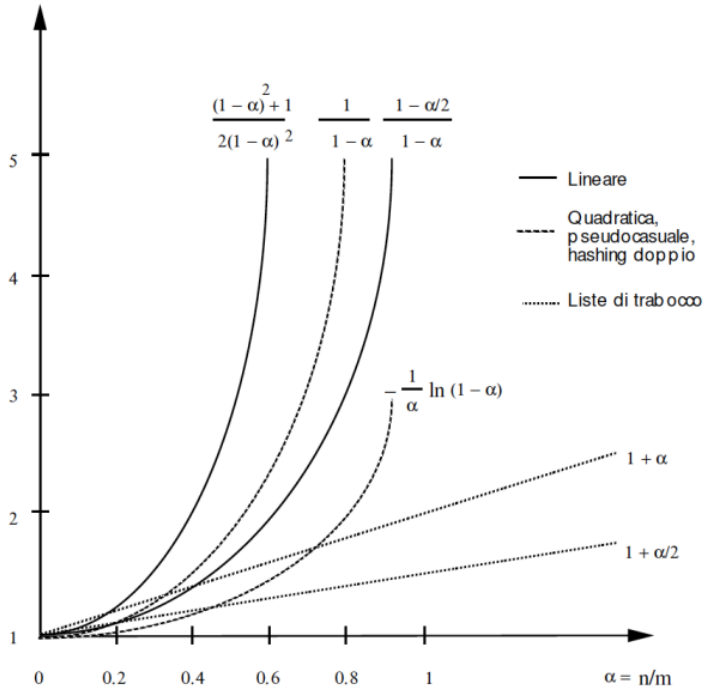
  Ricadono in questa categoria l'ispezione quadratica e il doppio hashing.

  - Caso **pessimo** di una ricerca **senza successo**: $O(\frac{1}{1-\alpha})$
  - Caso **pessimo** di una ricerca **con successo**: $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

- Sequenze di ispezione **non equiprobabili**

  Ricade in questa categoria l'ispezione lineare.

  - Caso **medio** di una ricerca **senza successo**: $O(\frac{(1-\alpha)^2+1}{2(1-\alpha)^2})$
  - Caso **medio** di una ricerca con **successo**: $O(\frac{1-\alpha/2}{2(1-\alpha)})$



Riepilogo numero di ispezioni della funzione search in una tabella hash con indirizzamento aperto.

Notiamo inoltre che le prestazioni delle tabelle hash con indirizzamento aperto dipendono dal fattore di carico α. La strategia per migliorare le prestazione è dunque quella di mantenere un fattore di carico basso (un fattore di carico $\alpha < 0.75$ è considerato ottimale), dunque è utile ridimensionare la tabella quando il fattore di carico supera una certa soglia critica.

Dizionario con tabella hash

Riassumiamo il costo delle operazioni basilari di un dizionario implementato tramite tabella hash confrontati con le altre tipologie di implementazione.

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista concatenata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Albero Binario di Ricerca	$O(\bar{h})$	$O(h)$	$O(\bar{h})$	$O(h)$	$O(\bar{h})$	$O(h)$
Albero AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Tabelle Hash	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$

Riassunto dei costi per le operazioni basilari delle diverse implementazioni di un dizionario.

Nonostante i dizionari implementati tramite tabelle hash abbiano costi pessimi lineari, sotto ragionevoli assunzioni probabilistiche hanno un costo costante e nella pratica sono molto efficienti, infatti le implementazioni di strutture dati di tipo dizionario fanno tipicamente uso di tabelle hash.

Ad esempio in java la struttura dati **HashMap** fa completo utilizzo di tabelle hash con liste concatenate che vengono convertite in alberi AVL quando le liste concatenate diventano troppo grandi.