1 Hashing

L'**hashing** è una **tecnica alternativa** agli alberi binari di ricerca (capitolo 4.4) o agli array associativi, utilizzata per realizzare i dizionari.

A differenza dell'implementazione tramite alberi, nella quale si riusciva a mantenere la stessa complessità nei casi di `insert()`, `lookup()` e `remove()`, nel caso ideale, la cosa migliore sarebbe quella di **mantenere una complessità costante** per tutti i tipi di operazioni, che sia inoltre **inferiore** a quella delle strutture ad albero.
Questa implementazione ideale prende il nome di **tabelle di hash**.

	Array non ordinato	**Array ordinato**	**Lista**	**Alberi RB**	~~**Impl. ideale**~~ **Hash table**
insert()	$O(1), O(n)$	$O(n)$	$O(1), O(n)$	$O(\log n)$	$O(1)$
lookup()	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
remove()	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
foreach	$O(n)$	$O(n)$	$O(n)$	$O(n)$	~~$O(n)$~~$O(m)$

Per comprendere il funzionamento delle tabelle hash dobbiamo prendere in considerazione il concetto di **insieme universo *U***, ovvero un insieme di tutte le possibili chiavi, la cui grandezza dell'insieme varia arbitrariamente in base ai dati che si vogliono contenere.

Idea di base

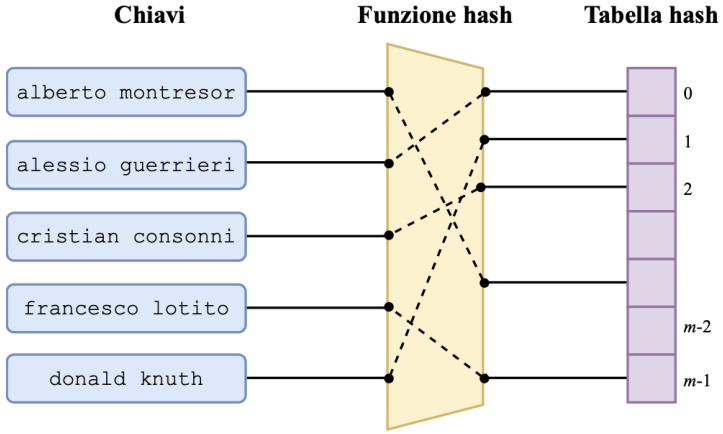
Quello che si vuole fare è memorizzare tutti i dati dell'insieme ***U*** in un vettore di dimensione (m) finita $T[0...m-1]$, ed avere un meccanismo per cui, data una chiave, trovare rapidamente la posizione in cui è memorizzata.

Le chiavi possono essere delle stringhe, degli oggetti o dei numeri, e il compito delle tabelle hash è quello di trasformarle in un indice all'interno di esse. Per fare ciò viengono utilizzate le **funzioni hash**.

Cos'è una funzione hash?

Una **funzione hash** è una funzione H che mappa ciascuna chiave k appartenente all'insieme universo U ($k \in U$) nell'indice $H(k)$ di un vettore A, destinato a contenere la coppia (k, v). Viene definita come $H : U \to \{0, 1, ..., m-1\}$.

Figure 17: Esempio di funzione hash



Come si può vedere dall'immagine, viene presa una chiave (in questo caso una stringa) che verrà poi trasformata in qualche modo, tramite la funzione hash (che può essere anche una black box), in un indice.

A questo punto sorge un **problema**: l'insieme delle chiavi è potenzialmente infinito, ma non si vuole che sia lo stesso anche per la tabella hash, dal momento in cui si potrebbe non disporre

dello spazio in memoria necessario per qualunque coppia chiave-valore. Dunque, quello che succede è che avvengono delle **collisioni**.

1.1 Tabelle ad accesso diretto

Prima di affrontare il problema delle collisioni, è utile analizzare un **caso particolare** in cui esse possono essere **completamente evitate**: le **tabelle ad accesso diretto**.

Questo approccio è applicabile solo quando l'insieme universo U delle chiavi è limitato e di dimensione ridotta, tale da poter essere rappresentato direttamente in memoria senza **sprechi** eccessivi. Utilizzando un approccio del genere è possibile utilizzare un vettore A della stessa dimensione dell'insieme U, quindi $m = |U|$, nel quale ogni chiave k che appartiene all'insieme U viene memorizzata direttamente nella posizione $A[k]$.
La funzione hash utilizzata è quindi la **funzione identità** $h(k) = k$, in questo caso una **funzione hash perfetta**, come quella illustrata in figura 17, che garantisce l'esecuzione delle operazioni di `insert()`, `lookup()` e `remove()` in tempo $O(1)$ nel caso peggiore.
In questo modo ogni chiave verrebbe memorizzata in una posizione distinta della tabella.

1.1.1 Funzioni hash perfette

Funzioni hash perfette

Una funzione hash h si dice **perfetta** se è **iniettiva**, ovvero se non da origine a collisioni:
$$\forall k_1, k_2 \in U : k_1 \neq k_2 \Rightarrow H(k_1) \neq H(k_2)$$

Tuttavia, questo metodo presenta un **limite fondamentale**:
oltre al fatto che se l'insieme universo U è molto grande, l'approccio non è praticabile, nel caso in cui il numero di chiavi memorizzate nel dizionario sia molto inferiore al massimo disponibile, quindi $m = |U|$, si incorrerebbe in uno **spreco di meoria** poiché molte posizioni del vettore verrebbero lasciate inutilizzate.
Per questo motivo, le **tabelle ad accesso diretto** sono utilizzabili solo in **contesti specifici** e rappresentano principalmente un modello teorico di riferimento per le tabelle hash.

Per evitare inutili sprechi di memoria, la dimensione m del vettore non deve essere determinata sulla base dell'intero universo U, ma piuttosto in funzione del **numero di chiavi attese**, ovvero la **quantità** k **di elementi** che si prevede saranno **effettivamente presenti nel dizionario** in un determinato momento.

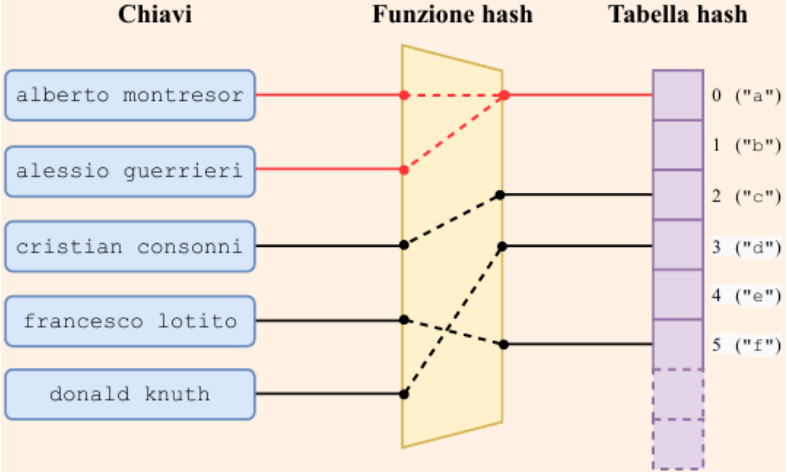
Quando si verifica una collisione?

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 $H(k_1) = H(k_2)$.

Le collisioni mettono quindi in luce due **aspetti fondamentali** nella progettazione di una tabella hash:

- La **scelta della funzione hash** è cruciale, infatti una funzione mal progettata può distribuire le chiavi in modo sbilanciato, causando una concentrazione eccessiva in alcune posizioni del vettore, lasciando quasi del tutto inutilizzate altre.

- È essenziale progettare dei **meccanismi** efficienti per la **gestione delle collisioni**, poiché 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.
Questi meccanismi permettono quindi di **garantire il corretto funzionamento** delle operazioni di inserimento, ricerca e cancellazione.



1.2 Minimizzare le collisoni

Se **non è possibile evitare** le collisioni, si cerca di **minimizzarne il numero**.
La **qualità di una funzione hash** dipende in modo cruciale dalla **sua capacità** di distribuire le chiavi dell'universo U negli indici $[0...m-1]$ della tabella hash **in modo uniforme**.
Una distribuzione sbilanciata porta a un aumento del numero di collisioni in alcune celle, con conseguente peggioramento delle prestazioni del dizionario.

Uno dei criteri più diffusi per valutare la qualità di una funzione hash è l'**uniformità semplice**:
- Sia $P(k)$ la probabilità che una qualunque chiave k sia inserita nel dizionario;
- Sia $Q(i)$ la probabilità che una qualunque chiave finisca nella cella i-esima della tabella.

Uniformità semplice

Una funzione hash h gode di **uniformità semplice** se, per ogni posizione i della tabella, la probabilità che una chiave k venga mappata in i è uguale a $1/m$, quindi $Q(i) = 1/m$.

Quindi, l'evento che interessa è: *"la chiave k_n scelta finisce nella cella i"*, e ciò accade se la funzione hash, per qualche motivo, mappa una determinata chiave per quella cella: $h(k_1) = i$, oppure $h(k_2) = i$, e così via... definendo eventi tra loro mutualmente esclusivi.

$$Q(i) = \sum_{k \in U : h(k) = i} P(k)$$

Quando un evento può accadere tramite alternative mutualmente esclusive, la probabilità totale è la somma delle probabilità alternative.
Proprio per questo motivo ottengo $Q(i)$ sommando le probabilità che le chiavi k_n hanno di finire in i: se questa poi risulta uguale a $1/m$ la funzione hash gode di uniformità semplice.

Problema fondamentale

Per poter ottenere una **funzione hash con uniformità semplice**, la **distribuzione delle probabilità** P deve essere **nota**.
Ad esempio, si supponga di avere $m = 10$ celle nella tabella.
- 90% delle chiavi che iniziano con "*A*";

- 10% delle chiavi che iniziano con "*Z*".

Per costruire una funzione hash con uniformità semplice bisognerebbe fare in modo che le celle che iniziano con "*A*" occupino il 90% delle celle $(0...8)$, mentre le chiavi che iniziano con "*Z*" occupino il 10% delle celle (cella 9) in modo che la probabilità che una chiave cada nelle celle $0...8$ sia dello 0.9 e la probabilità che cada nella cella 9 sia dello 0.1.

Nella realtà però non sappiamo quali chiavi verranno inserite né con quale frequenza compariranno, quindi **non è possibile costruire una funzione hash perfettamente uniforme**. In sostanza la distribuzione esatta $P(k)$ non può essere nota e si va ad utilizzare tecniche **euristiche**.

Le tecniche euristiche sono delle funzioni hash che usano **formule matematiche** per distribuire bene le chiavi anche se non si conosce $P(k)$, sperando che nella pratica funzionino bene.

1.3 Come realizzare una funzione hash

Dal momento che per utilizzare le tecniche euristiche **si ha bisogno di numeri**, si presenta la necessità di **codificare le chiavi** in numeri naturali. Dunque, se la chiave non è un numero (ad esempio una stringa "ciao") bisogna prima convertirla.

Assunzione

Si assume che le chiavi possano essere tradotte in valori **numerici non negativi**, eventualmente interpretando la loro rappresentazione in memoria come un numero.

Esempio: trasformazione di stringhe

Consideriamo una chiave k costituita da una stringa di caratteri, la trasformazione in intero avviene seguendo dei passaggi logici:

1. $ord(c) \rightarrow$ **Codifica dei caratteri**: Si prende il valore ordinale binario del carattere c secondo una codifica standard (ad esempio ASCII);
2. $bin(k) \rightarrow$ **Rappresentazione binaria**: si concatenano i valori binari dei singoli caratteri che compongono la chiave k.
3. $int(b) \rightarrow$ **Interpretazione intera**: la sequenza di bit risultante viene interpretata come un unico grande numero intero.

```
Utilizzando una codifica ASCII a 8 bit, e concatenando i risultati di
ogni lettera, si ottiene un numero a 24 bit
bin("DOG") = ord("D")  ord("O")  ord("G")
           = 01000100  01001111  01000111
int("DOG") =  68 · 256  +  79 · 256^2  +  71  = 4.476.743
```

Come detto alla fine del capitolo 1.2, le chiavi vanno ben distribuite proprio perché si ha il problema che il numero ottenuto è solitamente molto grande (nell'esempio precedente > 4 milioni), e la tabella hash non ha 4 milioni di righe, magari solo $m = 100$.

I metodi che seguiranno sono esattamente quelle tecniche euristiche che utilizzano formule matematiche progettate per **distribuire** (o "*sparpagliare*") uniformemente questi grandi numeri all'interno delle poche celle disponibili.

1.3.1 Metodo dell'estrazione

Il metodo dell'estrazione è uno dei più semplici per definire una funzione hash su chiavi binarie.

Come funziona?

- Si assume che la **dimensione della tabella** sia una **potenza di due**, ovvero $m = 2^p$;
- Si seleziona un **blocco di** p **bit** dalla rappresentazione binaria della chiave;
- L'indice hash è ottenuto **convertendo questi bit in un numero** intero.

Esempio di utilizzo