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

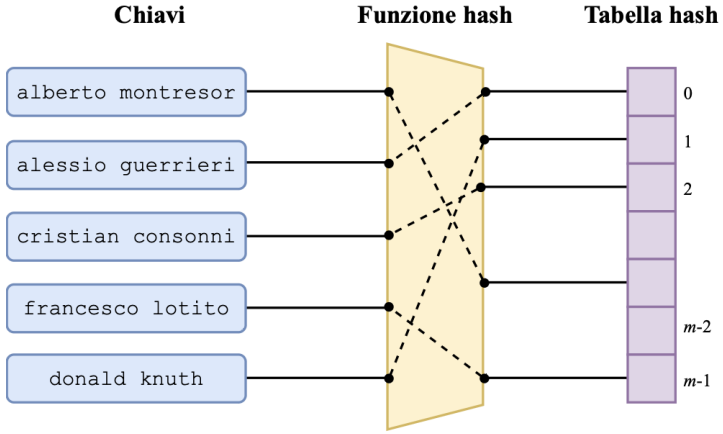
Quello che si vuole fare è memorizzare tutti i dati dell'insieme ***U*** in un vettore di dimensione (m) finita $T[0...m-1]$, ed avere un meccanismo per cui, data una chiave, trovare rapidamente la posizione in cui è memorizzata.

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.

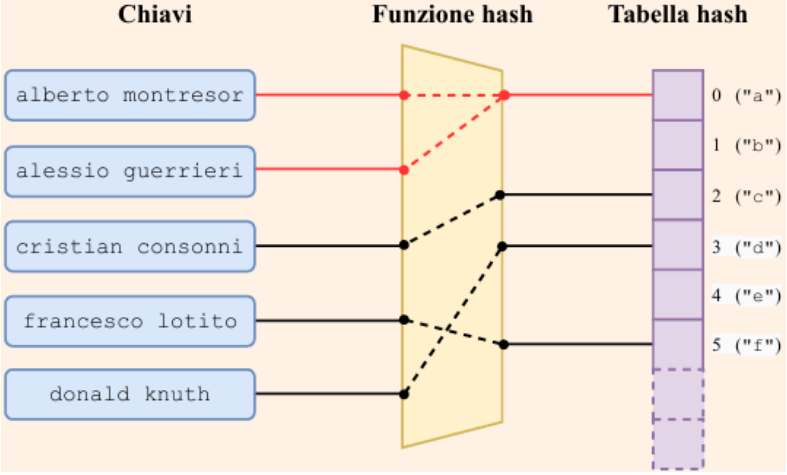
Quando si verifica una collisione?

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 · 256^2  +  79 · 256  +  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 ed efficienti 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.

Esempi di utilizzo
Si sceglie una tabella con $m = 2^p = 2^{16} = 65536$ celle.
Per indirizzarle tutte, **necessiteranno** di $p = 16$ bit estratti dalla chiave, ad esempio:

- Si può scegliere il **blocco dei** 16 **meno significatvi**

```
bin("Alberto") =  01000001 01101100 01100010 01100101 01110010 01110100
                  01101111
bin("Roberto") =  01010010 01101111 01100010 01100101 01110010 01110100
                  01101111
H("Alberto") = int(01110100 01101111) = 29.807
H("Roberto") = int(01110100 01101111) = 29.807
```

- Oppure si prendono 16 bit partendo da una **posizione casuale interna**

```
bin("Alberto") =  01000001 01101100 01100010 01100101 01110010 01110100
                  01101111
bin("Alessio") =  01000001 01101100 01100101 01110011 01110011 01101001
                  01101111
H("Alberto") = int(00010110 11000110) = 5.830
H("Alessio") = int(00010110 11000110) = 5.830
```

Come si può vedere dagli esempi, nonostante questa metodologia sia semplice ed efficiente, è **estremamente sensibile** alla scelta dei **bit selezionati per la chiave**, infatti, variazioni minime delle chiavi come anagrammi o suffissi simili produrranno **collisioni frequenti**.
Anche selezionare bit da altre parti della chiave può risultare inefficace, specialmente se le chiavi condividono prefissi o segmenti centrali comuni.
Proprio per questo motivo, **se non si conosce la distribuzione dei dati, questo approccio è sconsigliato**, in quanto scarta completamente l'informazione contenuta nei bit non selezionati.

1.3.2 Metodo dello XOR

Il metodo dello XOR nasce per risolvere il difetto del metodo dell'estrazione, nel quale, se prendi solo alcuni bit, **ignori completamente i primi**.
Quindi, se cambia un bit all'inzio della chiave, l'hash non cambia, mentre con lo XOR si vuole che **tutti i bit partecipino al calcolo dell'indice**.

Come funziona?

- Anche in questo caso si assume che la **dimensione della tabella** sia una **potenza di due**, ovvero $m = 2^p$;
- Si suddivide la rappresentazione binaria della chiave k in q sotto-blocchi, ciascuno di lunghezza pari a p bit (la dimensione dell'indirizzo della tabella);
- L'indice hash $h(k)$ è ottenuto effettuando lo XOR progressivo tra tutti i blocchi e **convertendo la somma finale in un numero** intero.

Esempio: si immagini che la chiave sia una striscia di carta lunga (tanti bit) e che la si voglia ridurre a un quadratino piccolo (l'indice p). Invece di tagliare un pezzo e buttare il resto (estrazione), si piega la striscia su se stessa sovrapponendo i bit.

Esempio di utilizzo
Utilizziamo sempre una tabella con $m = 2^p = 2^{16} = 65536$ celle, 16 bit di indirizzamento.
In questo caso, dovendo dividere la chiave in sotto-blocchi da 16 bit, risultano 5 gruppi, uno dei quali viene riempito con 8 zeri di "*padding*". Utilizzando il metodo XOR è come se facessimo tante addizioni in colonna per le quali non si considera il riporto.

```
bin("montresor") =
01101101 01101111 ⊕
01101110 01110100 ⊕
01110010 01100101 ⊕
01110011 01101111 ⊕
01110010 00000000

H("montresor") =
int(01110000 00010001) = 28.689
```

```
bin("sontremor") =
01110011 01101111 ⊕
01101110 01110100 ⊕
01110010 01100101 ⊕
01101101 01101111 ⊕
01110010 00000000

H("sontremor") =
int(01110000 00010001) = 28.689
```

Nonostante questo approccio permetta di utilizzare ogni singolo bit della chiave per influenzare il risultato finale, presenta alcune problematiche:

- **Vulnerabilità agli anagrammi**: due chiavi composte dagli stessi caratteri in ordine diverso (come nell'esempio sopra illustrato) possono generare lo stesso indice hash, e di conseguenza collisioni;
- L'**efficacia** dipende fortemente dalla suddivisione in blocchi e dall'eventuale padding applicato.

1.3.3 Metodo della divisione

I metodi basati sulla manipolazione dei bit visti ai capitoli 1.3.1 e 1.3.2 soffrono di problemi legati alla regolarità dei dati (ad esempio, suffissi identici) o alla commutatività (anagrammi). Questo perché, caratteri uguali in chiavi differenti, vengono rappresentati con la stessa codifica ASCII a 8 bit.
Il **metodo della divisione** cambia approccio, utilizzando una **logica aritmetica** al posto di una logica basata sui bit. Viene infatti utilizzato un sistema posizionale (come le unità, decine e centinaia), dove i caratteri più a sinistra valgono di più, ad esempio:

```
"AB"=ord("A")·256^1 + ord("B")·256^0
"BA"=ord("B")·256^1 + ord("A")·256^0
```

Come funziona?

Utilizzando la logica aritmetica si converte quindi la chiave in un **intero**, che grazie al sistema posizionale è **sempre diverso** (evitando così anagrammi), di cui si calcola il **resto della divisione per** m: il resto della divisione per m è **sempre un numero compreso tra** 0 **e** $m-1$.
In questo modo è possibile indirizzare correttamente tutte le celle della tabella hash senza "***uscire dai bordi***".

La scelta del numero di celle (m)

La scelta di m determina la **qualità della distribuzione**: è necessario che sia **un numero dispari**, preferibilmente un **numero primo** non troppo vicino a una potenza di due.

Le chiavi reali spesso presentano dei pattern o regolarità (ad esempio, indirizzi di memoria che sono multipli di 4 o 8). Se m avesse un divisore in comune con il "passo" o il pattern delle chiavi, la funzione hash mapperebbe le chiavi solo su un sottoinsieme delle celle disponibili, lasciandone molte vuote e causando collisioni. Un numero primo non possiede divisori (a parte 1 e se stesso), minimizzando la probabilità di interazione con i pattern dei dati e garantendo una distribuzione più uniforme ("sparpagliamento") su tutta la tabella.

Esempio di utilizzo
Si sceglie un numero di celle secondo le specifiche descritte prima, ad esempio $m = 383$

```
H("Alberto")  = 18.415.043.350.787.183    mod 383 = 221
H("Alessio")  = 18.415.056.470.632.815    mod 383 = 77
H("Cristian") = 4.860.062.892.481.405.294 mod 383 = 130
```

Dunque, il metodo della divisione è facile da imlementare ed efficace **solo se** m **viene scelto con cura**, altrimenti possono verificarsi le seguenti problematiche:

- Se si sceglie un numero di celle per la tabella hash pari a $m = 2^p$, l'operazione modulo della chiave considera solo gli ultimi p bit per la creazione dell'indice.
  In questo modo viene buttata via parte dell'informazione della chiave che poteva essere utilizzata per la creazione del suo indice all'interno della tabella hash, "regredendo" così al metodo dell'estrazione (capitolo 1.3.1);
- Se invece si sceglie un numero di celle pari a $m = 2^p - 1$, bisogna fare attenzione che la **base di rappresentazione dei dati** (es. 256) divisa per m **non dia resto 1**. Questo perché verrebbe a mancare il peso conferito a ciascun carattere tramite il sistema posizionale, facendo diventare la funzione hash una semplice somma, incapace di distinguere gli anagrammi.

  Esempio pratico (in base 10)
  Immaginamo che $m = 9$, con due numeri anagrammi: 12 e 21.
  Calcoliamo l'indice della tabella hash (il resto della divisione per 9):
  - $H(12)$: Normalmente è $1 \cdot 10 + 2$, ma col modulo 9, il 10 diventa un 1. Dunque: $1 \cdot 1 + 2 = 3$.
  - $H(21)$: Normalmente è $2 \cdot 10 + 1$, ma col modulo 9, il 10 diventa un 1. Dunque: $2 \cdot 1 + 1 = 3$.

  Come si può vedere, entrambe le chiavi vengono mappate con indice 3, creando collisione.

1.3.4 Metodo della moltiplicazione (Knuth)

A differenza del metodo della divisione (capitolo 1.3.3), nel quale ci si affidava alla scelta di un m primo per "rompere" i pattern delle chiavi, **il metodo della moltiplicazione** utilizza una costante C per mescolare i bit. Questo permette di ottenere indici ben distribuiti pur scegliendo un valore di m comodo per il calcolatore (ad esempio una potenza di 2).

Come funziona?

Il processo avviene in **due passaggi**:

- ***Mescolamento***: si moltiplica il valore numerico associato alla chiave k per una costante C (spesso **irrazionale**) compresa tra 0 e 1, e si estrae la parte frazionaria (decimale) del risultato. Questa operazione "distrugge" i pattern della chiave originale;
- ***Mappatura***: Si moltiplica questa parte frazionaria per m (il numero di celle) così da ottenere un indice valido tra 0 ed $m-1$.

La formula che esprime questi passaggi è la seguente: $h(k) = \lfloor m(kC \mod 1) \rfloor$

Perché conviene che C sia irrazionale?

Un numero razionale semplice (come ad esempio $0.5 = 1/2$) ha una rappresentazione binaria finita e regolare. Moltiplicare per esso equivale spesso a un semplice scorrimento di bit (shift), con il rischio di estrarre nuovamente pattern prevedibili.
Dunque è preferibile che C sia un **numero irrazionale**, cioè un numero non esprimibile tramite un rapporto di due numeri interi che ha una **rappresentazione decimale illimitata** (dopo la virgola le cifre continuano all'infinito) e **non periodica** (non formano una sequenza ripetitiva).

Nello specifico *Knuth* suggerì il valore $C = (\sqrt{5}-1)/2$, poiché presenta delle caratteristiche matematiche interessanti che permettono una distribuzione più uniforme ("spapagliata"), minimizzando le collisioni.

Esempio di utilizzo

Sia $m = 2^{16} = 65536$ celle e $C = (\sqrt{5}-1)/2 \approx 0.6180339887$.
Se moltiplico $C \cdot k$, dove k è il valore numerico associato a ciascuna chiave dopo la codifica ASCII e la trasformazione in intero utilizzando il sistema posizionale, e estraggo la parte decimale, ottengo una parte decimale che risulterà distribuita in modo **pseudo-casuale**.
Dopodiché moltiplico la parte decimale per m ottenendo un indice valido tra 0 e $m-1$.

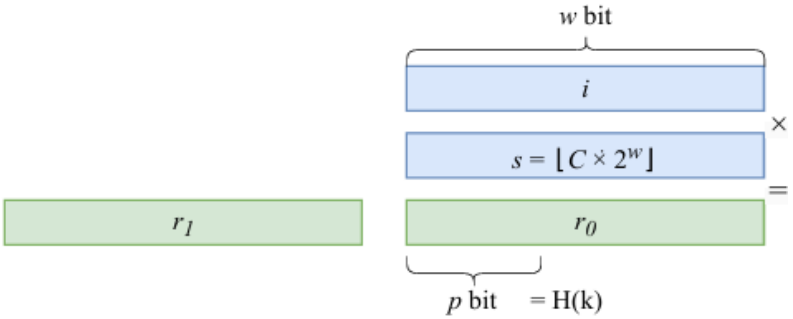
```
H("Alberto")  = 65.536 · 0.78732161432 = 51.598
H("Alessio")  = 65.536 · 0.51516739168 = 33.762
H("Cristian") = 65.536 · 0.72143641000 = 47.280
```

A questo punto è possibile effettuare alcune osservazioni:

- il metodo della moltiplicazione funziona per qualsiasi valore di m, ma è **particolarmente efficiente** se $m = 2^p$;
- Il **risultato dipende fortemente dal valore di** C, infatti come detto prima valori irrazionali sono preferibili per garantire una buona distribuzione;
- È **indipendente da particolarità delle chiavi** (prefisi, suffissi).

A differenza di come è stato spiegato a livello teorico, a **livello pratico** per un computer fare calcoli con la virgola rappresenta un **problema** che causa **lentezza** e **minor precisione**. Dunque, per utilizzare questo metodo, un calcolatore ha la necessità di eseguire lo stesso calcolo utilizzando **solo aritmetica intera**, e per farlo in modo performante sfrutta la **dimensione delle parole del processore**.

Una CPU ha dei contenitori fissi (i registri) la cui dimensione può essere espressa con w, che solitamente nei calcolatori moderni equivale a 64 bit. Usando un w più piccolo (< 64) sprecheremmo spazio nei contenitori (bit inutilizzati), mentre utilizzando un w più grande (> 64) avremmo bisogno di due contenitori e di calcoli più complessi.



Lavorando in questo modo, l'idea è quella di sostituire la costante reale C con una **costante intera** S, che contiene la moltiplicazione tra C $(0.618..)$ e 2^{64} in modo tale da spostare la virgola di 64 posizioni verso destra, ma siccome il registro del computer ha solo 64 posti tutto quello che "avanza" ancora dopo la virgola cade nel vuoto e viene perso. In questo modo la parte decimale (dopo la virgola) sparisce e all'interno di S rimane solo un numero intero gigantesco che "riempie" perfettamente la parola in memoria.

Come fatto prima a livello teorico, ora è necessario moltiplicare la costante S (intero) per k (anch'esso un intero rappresentativo del valore della chiave di grandezza $w = 64$ bit) in modo da ottenere il numero con la parte **decimale pseudo-casuale**.
Quando la CPU moltiplica due numeri interi a w bit viene prodotto un risultato **sempre grande il doppio** ($2w$ bit), e il processore salva il risultato in due registri separati automaticamente:
- Registro Alto (r_1): che contiene i primi w bit, che rappresentano la parte intera;
- Registro Basso (r_0): che contiene gli ultimi w bit, che rappresentano la parte frazionaria.

Di queste due parti ci interessa solo il registro basso (r_0), che rappresenta la **parte frazionaria pseudo-casuale**. Per ottenere l'indice finale tra 0 e $m - 1$ (assumendo $m = 2^p$), non serve eseguire un'altra moltiplicazione: è sufficiente estrarre i p bit più significativi di r_0. Questo equivale matematicamente a moltiplicare per m e troncare, ma a livello hardware è un semplice spostamento di bit (shift), istantaneo per la CPU.

1.4 Gestione delle collisioni

Come abbiamo già detto al capitolo 1.1.1, quando due chiavi vengono mappate dalla funzione hash nella stessa posizione della tabella, si verifica una **collisione**.

Idea generale per la gestione delle collisioni

- ***Inserimento***: se la posizione calcolata tramite la funzione hash risulta già occupata, è necessario trovare una **posizione alternativa** in cui memorizzare la chiave.
- ***Ricerca***: se durante la ricerca di una chiave, essa non si trova nella posizione attesa, bisogna **cercarla nelle posizoni alternative**, secondo la **strategia adottata** in fase di **inserimento**.

Esempio con analogia

- **Inserimento**: ho il biglietto per il posto auto numero 5, ma arrivo ed è occupato. Seguendo la regola del parcheggio, provo il posto successivo finché non ne trovo uno libero, ad esempio, 6 occupato e 7 libero;
- **Ricerca**: quando torno a riprendere l'auto il biglietto dice ancora 5, ma a quel parcheggio non c'è la mia auto. Allora devo agire **analogamente** all'inserimento, quindi se so che il 5 era occupato controllo prima il 6 e poi il 7.

Dato 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**.
Le posizioni alternative possono essere trovate all'**interno della tabella** o all'**esterno della tabella**, ciò evidenzia **due possibili approcci** per la gestione delle collisoni:

- Le posizioni alternative **esterne** alla tabella prendono il nome di **liste di trabocco**;
- Le posizioni alternative **interne** alla tabella prendono il nome di **indirizzamento aperto**;

A volte può succedere che quando si va a cercare una chiave la quale non si trova nel posto giusto, poi la si va a cercare in posti alternativi, e se non si trova, la si cerca ancora in altri posti alternativi finché:

- Si trova la chiave;
- Si trova una **cella vuota**. Se si trova un buco significa che la chiave non c'é (perché se ci fosse stata sarebbe dovuta essere li o dopo).

In una tabella hash ben progettata, dove le chiavi sono ben "sparpagliate", si calcola l'hash, si va alla cella e si trova il dato. Ci si trova quindi nel caso medio, dove il costo è rappresentato da $O(1)$ e non importa se gli elementi sono 10 o 10 milioni, proprio perché l'accesso al dato è immediato.
Invece, in alcuni casi, ad esempio in presenza di molte collisioni o di una funzione hash mal progettata, nel caso medio il tempo può degradare fino a $O(n)$.

1.4.1 Liste di trabocco (concatenamento o chaining)

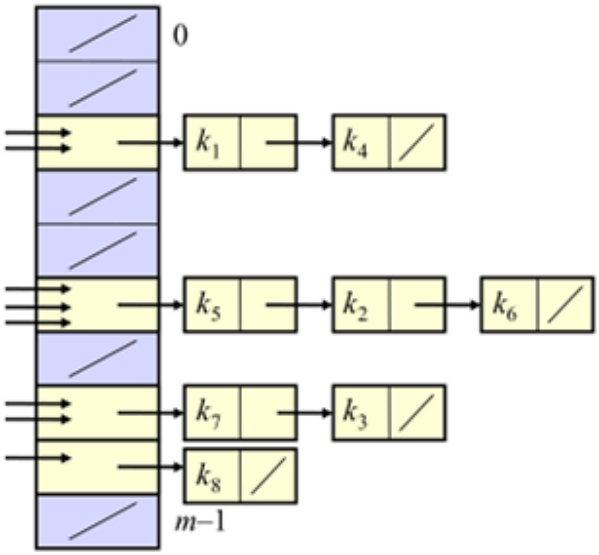
Le **liste di trabocco** rappresentano un approccio per la gestione delle collisioni che si basa sull'utilizzo di **liste monodirezionali**.

Idea generale

Ogni indice (slot) della tabella hash non contiene direttamente una chiave, ma un **puntatore alla testa di una struttura dinamica** (lista monodirezionale-vettore dinamico), in cui vengono memorizzate tutte le chiavi che **condividono lo stesso indice hash**.

Come struttura dinamica da utilizzare sono sufficienti le liste monodirezionali poiché o si trova o non si trova la chiave durante le operazioni di `insert()`, `lookup()`, `remove()`.
Facendo riferimento alla figura 20, nel momento in cui vengono inserite due chiavi nell'indice 2, tre nell'indice 5, e altre due nell'indice 7, esse sono aggiunte ad una lista (in corrispondenza dell'indice) che cresce in modo arbitrario.
Dunque, le operazioni effettuate sulle liste utilizzano la seguente logica:

Figure 20: liste di trabocco



- `insert()` **in coda**: viene utilizzato se devo inserire una chiave e verificare che questa non sia già stata inserita, devo scorrere tutta la lista. Se la chiave che sto inserendo è già presente mi fermo e sovrascrivo il valore, altrimenti arrivo in fondo, aggiungo l'elemento alla lista e lo memorizzo;
- `insert()` **in testa**: viene utilizzato quando la lista è vuota per inserire una chiave senza preoccuparsi che questa sia già presente;
- `lookup()`, `remove()`: in entrambi i casi bisogna scorrere tutta la lista per cercare la chiave e restituire il valore corrispondente/rimuovere la coppia chiave-valore nella lista.

1.4.2 Liste di trabocco: analisi della complessità

A questo punto si vuole fare un'**analisi della complessità** di questo meccanismo in particolare, e per fare ciò bisogna definire alcuni parametri:

- $n \rightarrow$ numero di chiavi memorizzate in tabella hash;
- $m \rightarrow$ capacità della tabella hash;
- $\alpha = n/m \rightarrow$ si riferisce al **fattore di carico** che nel caso delle liste di trabocco può essere:
  - $\alpha = 0$ se la tabella è vuota;
  - $0 < \alpha < 1$ se la tabella non è completamente occupata;
  - $\alpha > 1$ se la tabella contiene tanti elementi, non esiste un limite superiore.

Partendo da questi parametri, si ottengono poi due informazioni importanti:

- $I(\alpha) \rightarrow$ numero medio di accessi alla tabella per la ricerca di una chiave non presente all'interno di essa (**ricerca con insuccesso**);
- $S(\alpha) \rightarrow$ numero medio di accessi alla tabella per la ricerca di una chiave presente all'interno di essa (**ricerca con successo**).

Analisi del caso pessimo

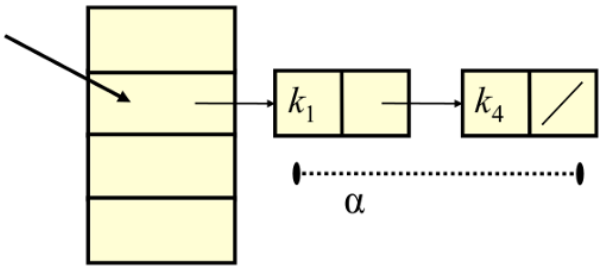
Il caso peggiore, si presenta nel momento in cui ci si ritrova ad avere dei **costi** che sono i **più elevati possibili**: questo avviene quando tutte le chiavi hanno lo **stesso hash**, ovvero tutte le chiavi vengono **collocate nella stessa lista**. Possiamo quindi dire che una struttura dati efficiente come le tabelle hash **si trasformano in una lista non ordinata**.

- `insert()` ha costo $\theta(1)$ se non si controlla (inserimento in testa), mentre ha costo $\theta(n)$ nel caso in cui si debba controllare che la chiave sia già stata inserita.
- `lookup()` e `remove()` hanno entrambi costo $\theta(n)$ poiché in entrambi i casi bisognerà scorrere la lista.

Analisi del caso medio

Per quanto riguarda il caso medio, è necessario fare prima alcune assunzioni:

- Dipende da come le chiavi vengono distribuite;
- Si assume che la funzione hash implementata abbia un hashing uniforme semplice, e che sia possibile calcolarla in $\theta(1)$.



Dopodiché, per lo studio del caso medio, bisogna **prendere in considerazione** la **lunghezza delle liste/vettori**. L'idea generale è la seguente:
Se ho $n = 1000$ chiavi e 100 celle nel vettore avrò un fattore carico di $\alpha = n/m = 1000/100 = 10$, ciò significa che il numero atteso di elementi in una lista (se come detto prima la distribuzione è uniforme) è 10. In sostanza, se il fattore di carico è tenuto abbastanza piccolo ($2, 3$ oppure 4), allora le **operazioni avranno un costo che deriva da** α, ad esempio:

- **Ricerca senza successo**: se sto effettuando un tipo di ricerca **senza succeso** dovrò guardare tutta la lista di lunghezza α, quindi pago il costo della funzione hash $\theta(1)$ + il costo di analizzare in media l'intera lista di lunghezza α;
- **Ricerca con successo**: se invece effettuo una ricerca con successo dove potrei trovare la chiave al primo elemento oppure all'ultimo, **in media la troverò a metà** e quindi il costo atteso risulta la somma tra il costo della funzione hash $\theta(1)$ + la metà della lunghezza della lista $\alpha/2$.

Dunque, mantenendo un valore di α abbastanza **piccolo** si garantisce il fatto che i **costi rimarranno costanti**, propriò perché il fattore di carico (α) **influenza il costo computazionale delle operazioni sulle tabelle hash**.

1.4.3 Indirizzamento aperto (memorizzazione interna)

Utilizzando un approccio a **indirizzamento aperto** viene evitato l'utilizzo strutture dati complesse per le quali si utilizzano liste e puntatori (capitolo 1.4.1), ma ci si limita a memorizzare tutte le chiavi direttamente nel vettore che costituisce la tabella hash.

Idea generale

In questo caso l'idea generale è molto semplice: ogni indice (slot) della tabella hash conterrà o una chiave, o un puntatore `null`.

- **Inserimento**: Se durante l'inserimento lo slot prescelto dalla funzione hash H è già occupato, cerco uno slot alternativo;
- **Ricerca**: dopo l'inserimento, vado a ricercare nello slot prescelto, ma, se anch'esso è occupato da una chiave che non è quella che sto cercando, cerco negli slot alternativi, uno dopo l'altro, fino a quando non trovo la chiave che sto cercando oppure trovo `null` a significare che ho trovato una casella veramente vuota e che quindi non posso più andare avanti.

Per verificare che all'interno di uno slot sia presente una chiave oppure `null` ci si basa sul concetto di **ispezione**, ovvero l'**esame di uno slot** durante la ricerca.
Per descrivere questa strategia, la funzione hash viene estesa aggiungendo un secondo parametro, quindi non prende più in ingresso solo l'insieme universo U delle chiavi, ma anche un indice (**indice di sipezione**) compreso fra 0 e $(m-1)$.

$$H : \mathcal{U} \times \overbrace{[0 \dots m-1]}^{\text{Numero ispezione}} \rightarrow \overbrace{[0 \dots m-1]}^{\text{Indice vettore}}$$

La funzione $H(k, i)$ rappresenta la posizione esaminata quando si cerca nella tabella hash la chiave k (una di quelle dell'insieme U) 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.

Ad esempio, si prenda in considerazione la seguente **sequenza di ispezione** generata.

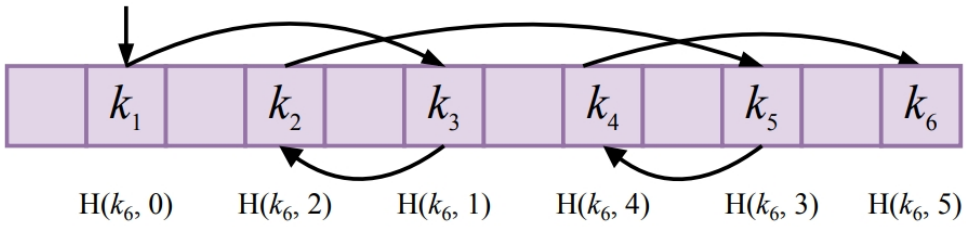


Figure 22: esempio di una sequenza di ispezione composta da 6 passi

Sequenza di ispezione

Una **sequenza di ispezione** $[H(k, 0), H(k, 1), ..., H(k, m-1)]$ è una permutazione degli indici $[0, ..., m-1]$, corrispondente all'ordine in cui vengono esaminati gli slot.
- Non si vuole esaminare lo stesso slot più di una volta;
- Potrebbe essere necessario esaminare tutti gli slot nella tabella.

In figura 22 si vede come, cercando k_6, al crescere dell'indice di ispezione si viene portati in posizioni differenti della tabella hash, poiché ad un determinato indice corrisponde una chiave. All'**ultima operazione** della sequenza di ispezione, se la **cella è vuota**:
- **Inserimento**: la chiave viene inserita;
- **Ricerca**: posso dire che k_n cercato non c'è.

Se invece la **cella è piena**:
- **Inserimento**: se la chiave nella cella è uguale a quella da inserire, decido se aggiornarla o dare errore (duplicato). Se è diversa (considerando che stimao parlando dell'ultima operazione della sequenza) la tabella è piena (Overflow/Errore).
- **Ricerca**: si deve controllare se la chiave in quella cella è quella che si cerca: se sì la chiava è stata trovata, altrimenti la chiave non esiste nella tabella.

1.4.4 Indirizzamento aperto: analisi della complessità

In un approccio ad indirizzamento aperto, il **fattore di carico** è un valore per tutta la tabella che varia da 0 a 1, il quale **indica quanto è satura**, ovvero la percentuale di **riempimento globale** della tabella hash. A differenza di come è stato visto al capitolo 1.4.2 il significato di α cambia, perché cambia la capienza fisica delle celle:
- **Liste di trabocco**: ogni cella è un puntatore ad una lista che può allungarsi all'infinito, quindi se $m = 10$, $n = 20$, avrò $\alpha = 2$ a significare che ***in media, ogni lista contiene 2 elementi***.
- **Indirizzamento aperto**: invece, in questo caso, ogni cella può contenere al massimo 1 elemento, quindi n (elementi) non potrà mai superare m celle, di conseguenza $0 \leqslant \alpha \leqslant 1$. Se $\alpha = 1$ la tabella è totalmente piena e si entra in **overflow**, facendo schizzare il costo delle operazioni a $O(n)$.

Esempio pratico: ipotizziamo dover trovare posto auto all'interno di un parcheggio.

- **Scenario A** ($\alpha = 0.1$): il parcheggio ha 100 posti e solo 10 macchine al suo interno. La probabilità che il posto assegnato con la funzione hash $h(k)$ sia occupato è bassissima (10%). Dunque, quasi sicuramente si troverà il posto auto al primo colpo, indicando ciò con $O(1)$.
- **Scenario B** ($\alpha = 0.9$): il parcheggio ha sempre 100 posti ma questa volta 90 macchine al suo interno. La funzione hash $h(k)$, molto probabilmente, calcolerà una posizione già occupata, per più volte, prima di trovare un posto auto libero all'interno del parcheggio. Ciò significa che bisogna girare tanto prima di trovare il 10% dei posti rimanenti.

In modo analogo all'esempio, il costo tende a $O(n)$ (cioè bisogna scorrere tutta la tabella) quando α si avvicina ad 1. Le collisioni diventano la norma, non l'eccezione, e le sequenze di ispezione si allungano a dismisura.

In un **contesto ideale**, per fare in modo che non si debba scorrere tutta la tabella e che quindi il costo non salga a $O(n)$, quello che si ricerca è una situazione di **hashing uniforme**.

Hashing uniforme ideale

Utilizando un **hashing uniforme** ogni chiave (k_n) dell'insieme U ha la stessa probabilità di avere come sequenza di ispezione $H(k_n, 0), H(k_n, 1), ..., H(k_n, m-1)$ una qualsiasi delle $m!$ permutazioni di [0,...,m-1].
Dunque la funzione hash non dà solo il punto di partenza, ma consegna l'intera lista delle celle da visitare, in ordine secondo una qualsiasi permutazione.

Esempio pratico

Prendiamo in considerazione due chiavi A e B, le quali per casualità, su una tabella da 5 posizioni partono dalla cella 2.

- Se si utilizzasse una regola fissa, ad esempio "***vai al successivo finché non trovi una cella vuota***", si creerebbe un "muro" (clustering) prima di poter inserire la chiave nell'indice:
  - A: $2 \to 3 \to 4 \to ...$
  - B: $2 \to 3 \to 4 \to ...$

  Questo fino a che una delle due trova per prima la cella vuota e subito dopo la rimanente va $+1$ avanti per essere inserita (**primary clustering - capitolo** 1.4.5).
- Se usiamo l'**Hashing uniforme**, il sistema assegna due itinerari completi totalmente diversi, pescati a caso tra tutte le combinazioni possibili ($m!$), in modo tale da non creare dei "muri". In questo modo ogni chiave può essere inserita il prima possibile.
  - A: $2 \to 0 \to 4 \to 1 \to 3$
  - B: $2 \to 3 \to 1 \to 0 \to 4$

Nella pratica, implementare un vero hashing uniforme è difficile.
Non possiamo garantire che il percorso di ricerca sia perfettamente casuale per ogni chiave (troppo costoso per il processore), ma **bisogna obbligatoriamente garantire** che il percorso non entri in un "ciclo vizioso". La funzione di ispezione deve essere progettata matematicamente affinché, al variare di i da 0 a $m-1$, vengano generati tutti gli indici della tabella esattamente una volta: dunque viene utilizzata **una singola permutazione**.
Per fare ciò vengono utilizzate tre tecniche principali:

- Ispezione lineare;
- Ispezione quadratica;
- Doppio hashing

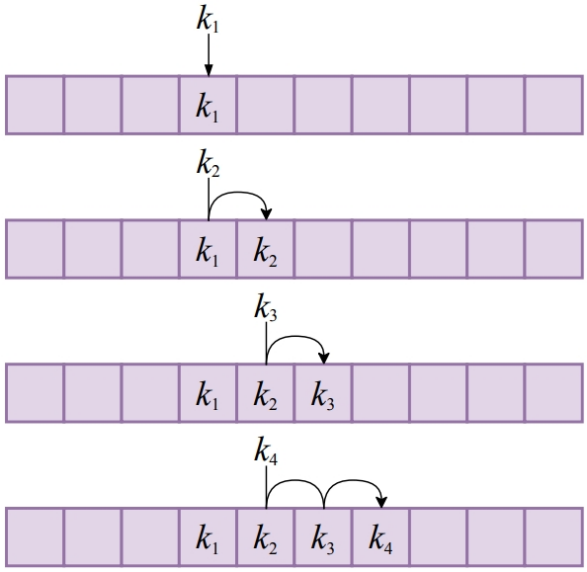
1.4.5 Indirizzamento aperto: ispezione lineare

Metodologia utilizzata

In questo metodo, si utilizza un parametro δ che rappresenta la distanza tra due posizioni successive esaminate durante l'ispezione: $H(k,i) = (H_1(k) + \delta \cdot i)\ mod\ m$

Dunque, si prende la posizione di partenza e poi si va avanti per un numero fisso di caselle δ. Quindi, quando vado a fare l'ispezione "*zeresima*", con $i = 0$, vado a guardare nella funzione $H_1(k)$, con $i = 1$ vado a guardare nella funzione $H_1(k)+\delta$, con $i = 2$ nella funzione $H_1(k)+2\cdot\delta$, e così via.
Dividere per il modulo di m garantisce di ottenere sempre un indice interno alla tabella, in modo tale che quando si supera l'ultima posizione della tabella, l'ispezione riprenda dall'inizio, rendendo il processo circolare.

Questo metodo è semplice, ma presenta un limite importante, quello dell'**agglomerazione primaria** (primary clustering): quando si vuole inserire una chiave all'interno di un determinato indice all'interno della tabella hash, essa collide con l'elemento già presente, e prosegue la sua ricerca fino a trovare la prima cella libera.



Dato che si sta utilizzando una metodologia a "***ispezione lineare***" (quindi si va avanti di una dimensione fissa), 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 quest'ultima.

Questo fenomeno porta alla formazione di sequenze sempre più lunghe di posizioni a distanza δ l'una dall'altra, il che rallenta le operazioni e i **tempi medi** di inserimento e cancellazione crescono. Dunque, uno slot vuoto che si trova dopo i slot già occupati nella stessa sequenza viene riempito con probabilità $(i + 1)/m$. L'ispezione lineare è quella "colpevole" dei muri, e il doppio hashing, come si vedrà al capitolo 1.4.7, è la "soluzione reale" al problema.

1.4.6 Indirizzamento aperto: ispezione quadratica

Metodologia utilizzata

L'unica differenza rispetto all'utilizzo di un ispezione lineare è che **la distanza** (offset) tra due posizioni successive nella sequenza di ispezione non è costante, ma **cresce quadraticamente** con i: $H(k,i) = (H_1(k) + \delta \cdot i^2)\ mod\ m$.

- con $\delta = 1$ e $i = 0$ si esamina prima la posizione $(H_1(k) + 0)$;
- con $\delta = 1$ e $i = 1$ si esamina la posizione $(H_1(k) + \delta\cdot)$;
- con $\delta = 1$ e $i = 2$ si esamina la posizone $(H_1(k) + \delta \cdot 4)$;
- con $\delta = 1$ e $i = 3$ si esamina la posizone $(H_1(k) + \delta \cdot 9)$, e così via;

Questa variazione di indice riduce l'effetto dell'**agglomerazione primaria**, tipico dell'ispezione lineare, poiché due chiavi con indirizzi iniziali differenti ($H_1(k) \neq H_1(k')$) tendono a generare sequenze di ispezione che divergono più rapidamente.
Tuttavia, se due chiavi condividono lo stesso indirizzo iniziale ($H_1(k) = H_1(k')$) le sequenze di ispezione coincidono completamente, e il problema si ripresenta in questo caso prendendo il nome di **agglomerazione secondaria**.
Un ulteriore limite, è che **la sequenza di ispezione** generata **non è una permutazione** dell'insieme $\{0, ..., m-1\}$: può quindi accadere che alcune posizioni libere non vengano mai raggiunte, impedendo l'inserimento di nuove chiavi anche se la tabella non è piena.

1.4.7 Indirizzamento aperto: doppio hashing

Il **doppio hashing** è il metodo che viene utilizzato maggiormente per risovere il problema dell'agglomerazione.

Metodologia utilizzata

Vengono utilizzate **due funzioni hash**, la prima indica la posizione iniziale e la seconda indica l'offset, che quindi cambia per ogni chiave: $H(k,i) = (H_1(k) + i \cdot H_2(k))\ mod\ m$.

In questo modo, anche se due chiavi k e k' hanno lo stesso indirizzo iniziale $H_1(k) = H_1(k')$, non è detto che abbiano lo stesso valore in H_2 (anzi, la probabilità è relativamente bassa), dunque tendono ad essere sparse in maniera più "*uniforme*" riducendo drasticamente la formazione di **agglomerati primari** e **secondari**.

Utilizzando questa metodologia si hanno al massimo m^2 sequenze di ispezione distinte:
- m scelte possibili per decidere da dove partire (cella $0, 1, ..., m-1$);
- m scelte possibili per decidere quanto lunghi fare i passi (offset di 1, 2, 3, ...)

Il numero totale di combinazioni unice (coppie partenza-passo) è quindi $m \cdot m = m^2$.
Però non basta avere tanti percorsi diversi (m^2), bisogna essere sicuri che **ogni percorso sia valido**, ovvero che continuando a saltare, **si debbano toccare tutte le celle della tabella** prima di ritornare al **punto di partenza**. Per garantire ciò si usano principalmente due metodi:
1. ***Metodo della potenza di*** 2
   - Si sceglie $m = 2^p$ (ad esempio, 16, 32, 64, ...), in modo tale che l'unico divisore di m sia 2;
   - Ci si assicura che $H_2(k)$ dia sempre un **numero dispari**.

   Funziona perché un numero dispari non è mai diivisibile per 2, quindi sono per forza primi tra loro.
2. ***Metodo del numero primo***:
   - Si sceglie m come numero primo (11, 13, 17, ...);
   - Ci si assicura che $H_2(k)$, dia un numero positivo minore di m ($0 < H_2 < m$);

   Funziona perché un numero primo non divisibile per nulla tranne che per se stesso.

1.4.8 Indirizzamento aperto: cancellazione

Durante la gestione delle collisioni, utlizzando un approccio ad indirizzamento aperto, se si vuole cancellare un elemento dalla tabella hash **non è possibile** semplicemente sostituire la chiave che si vuole cancellare con un valore `null`.
Questo perché si potrebbe "***rompere***" **una sequenza di ispezione**: infatti, come si può vedere

in figura 24, durante la ricerca di una chiave k_n, il raggiungimento di una cella `"null"` non garantisce che k_n non sia presente altrove nella tabella hash, poiché potrebbe essere semplicemente in una posizione differente sempre all'interno della stessa. Contrassegnare la cella come `"null"` **arresterebbe l'algoritmo** di ricerca, **spezzando il collegamento verso le chiavi successive** nella sequenza di ispezione.

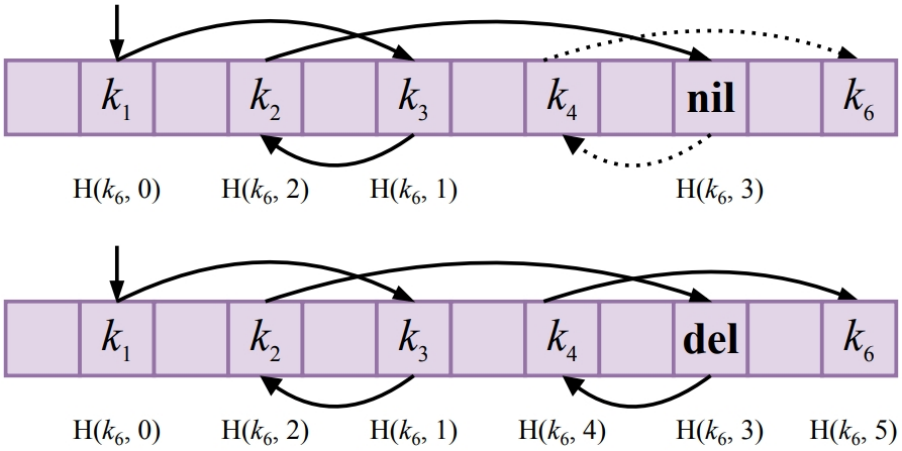


Figure 24: interruzione sequenza di ispezione

Proprio per questo motivo è necessario introdurre un **marcatore speciale** che indichi esplicitamente che la cella era occupata in passato ma è stata svuotata. L'elemento in questione è ***deleted (del)*** che permette di continuare l'ispezione fino a raggiungere la chiave ricercata (nel caso in cui sia effettivamente presente all'interno della tabella hash):

- Durante una **ricerca** il *deleted* viene trattato come uno slot pieno.
- Durante un **inserimento** il *deleted* viene trattato come uno slot vuoto, perché non ci interessa che sia stato cancellato in precedenza.

Tuttavia, aggiungere il marcatore *deleted* comporta lo svantaggio di aumentare i tempi di ricerca, che non dipendono più solamente dal fattore di carico α ($I(\alpha), S(\alpha)$ - capitolo 1.4.2), ma anche da quanti elementi sono stati cancellati.

1.5 Implementazione dell'hashing doppio

L'hashing doppo è indubbiamente la **tecnica migliore** da implementare **utilizzando** una gestione delle collisioni ad **indirizzamento aperto**, poiché permette di ridurre drasticamente clustering primari e secondari.

1.5.1 Definizione delle variabili

La tabella hash viene **memorizzata mediante due vettori paralleli**, a cui in seguito verrà assegnata la dimensione m (valore che detta la dimensione della tabella):

- key il quale memorizza le chiavi;
- $values$ che memorizza i valori ad esse associati.

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

Ovviamente, come spiegato al capitolo 1.4.3 e 1.4.8, l'implementazione dell'indirizzamento aperto prevede che ogni cella di del vettore key contenga una chiave valida oppure uno dei due valori speciali *null* o *deleted*, che indicano rispettivamente una posizione mai utilizzata e una posizione occupata in passato ma che è stata cancellata.

1.5.2 La funzione `hash`

Questa funzione prende in ingresso un parametro `"dim"` ad indicare la dimensione della tabella hash. Inizialmente viene definita la dimensione della tabella, dando un valore alla variabile `"m"` tramite il parametro `"dim"` che la funzione prende in ingresso.
Successivamente ci si occupa di inizializzare tutte le chiavi e i valori (in particolare solo le chiavi poiché i valori al momento non interessano) con il valore `null`.

```
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
```

1.5.3 La funzione `scan`

La funzione `scan` non fa parte dell'API della tabella hash, infatti non è un operazione del dizionario come `lookup()`, `insert()` e `remove`, ma serve solo per essere richiamata all'interno di esse. La funzione `scan` prende in input la chiave k che si vuole ricercare all'interno della tabella hash, e un valore booleano $insert$ che indica se si tratta di un operazione di inserimento oppure no, mentre **ritorna un indice intero** che **indica la posizione della chiave**, e non il valore associato ad essa.

```
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
```

- La variabile *fistDeleted* è un "trucco" molto intelligente per **ottimizzare le prestazioni** della tabella hash durante l'inserimento. Viene utilizzata per ricordare **la posizione della prima cella *deleted***.

  Durante una scansione per inserimento, dobbiamo obbligatoriamente arrivare fino in fondo (al `null`) per assicurarci che la chiave non esista già. Tuttavia, se lungo il percorso incon-

triamo delle celle *deleted*, la variabile *firstDeleted*, che inizialmente viene inizializzata a -1 (ad indicare "*per ora non ho visto celle cancellate*"), memorizza l'indice della prima di queste celle. Se alla fine della scansione la chiave non è stata trovata, l'algoritmo preferisce inserire il nuovo elemento nella posizione *firstDeleted* (riciclando lo spazio) piuttosto che in fondo alla sequenza (`null`). Questo mantiene la **tabella compatta** e **riduce i tempi di ricerca futuri**.

- La variabile i è l'indice del numero di ispezione (ispezione zeresima, ispezione uno, ispezione due, e così via...);
- La variabile j indica la posizione attualmente analizzata: viene inizializzata con il valore $H(k)$ dove H è la funzione hash di base, quella che ci dice il punto di partenza dove andare a cercare e k ovviamente è la chiave che stiamo cercando.

A questo punto inizia un ciclo che terminerà qualora:

- trovo la chiave che sto cercando;
- trovo una posizione `null` a significare che non è più possibile andare avanti, perché la chiave cercata non è stata trovata;
- Oppure, nessuna delle due condizioni precedenti si è verificata e il numero di ispezioni raggiunge m ($i = m$) a significare sostanzialmente che ho ispezionato tutta la tabella senza trovare quello che si stava cercando.

Invece, le due righe all'interno del ciclo while sono quelle che permettono di andare avanti:

- $i = i + 1$ somma $+1$ all'indice delle ispezioni;
- Per calcolare la prossima cella si utilizza una seconda funzione $H'(k)$ che è l'offest che si va a sommare alla posizione che si stava cercando, come prevede il doppio hashing (capitolo 1.4.7).

Terminato il ciclo while si presentano due possibilità:

- Se sono in **condizione di inserimento** e non ho trovato una chiave da sovrascrivere e ***firstDeleted*** non è più inizializzato con il valore iniziale (-1), a significare che ho riempito la prima cella *deleted*, allora restituisco proprio ***firstDeleted***, poiché l'inserimento è avvenuto lì dentro;
- Altrimenti, sono in **condizione di ricerca** e restituisco j, che rappresenta l'ultimo indice guardato e se la chiave non è stata trovata, può essere un valore fittizio.

1.5.4 Le funzioni `lookup` e `insert`

Nel caso della funzione `lookup` viene richiamata la funzione `scan` passando il valore *false* ad indicare che non si tratta di un operazione di inserimento ma solo di ricerca.
Nel caso della funzione `insert` viene richiamata la funzione `scan` passando il valore *true* ad indicare che si tratta di un operazione di inserimento.

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

(a)

```
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
```

(b)

In entrambi i casi ottengo un indice che viene restituito dalla funzione `scan`:

- Nel caso della `lookup` (a) si hanno due possibilità:
  - si ha trovato quello che si cercava e questo è rappresnetato dal fatto che la chiave nell'indice i è uguale alla chiave k che si sta cercando, e allora viene restituito il valore associato a quella chiave;
  - Altrimenti ritorno `null` a significare che l'oggetto non è stato trovato.
- Allo stesso modo, nel caso dell'`insert` (b), le possibilità sono:
  - se si ha trovato una **casella vuota** oppure una casella ***deleted*** oppure una casella che **già contiene la chiave** k, in tutti questi casi si va a sovrascrivere il valore k e il valore v sulle rispettive posizioni i-esime;
  - Altrimenti, la tabella hash è piena poiché non si sono trovate caselle a disposizione dove andare a scrivere il valore k e quindi bisognerà ritornare un qualche tipo di errore.
    Questo potrebbe essere anche verificato prima di fare una ricerca del genere se si mantenessero informazioni riguardanti il numero di chiavi effettivamente registrate nella tabella, allora a quel punto se il numero di chiavi ha raggiunto m si può subito ritornare che è impossibile effettuare un inseriemnto.

1.5.5 La funzione `remove`

Nuovamente la funzione `scan` viene passata con il valore *false*, poiché non si tratta di un operazione di inserimento.
In questo caso la condizione è ancora più semplice:

- Trovo l'elemento e lo cancello, segnando la chiave come ***deleted*** (per non rompere la sequenza di ispezione) mentre il valore associato a quella determinata chiave; viene impostato a ***null***;
- Non trovo l'elemento e non devo fare assolutamente nulla;

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