Strutture Dati, Algoritmi e Complessità

TABELLE DI HASH

AA 2023-2024

I Dizionari

Una struttura dati molto comoda ed efficiente per supportare diverse applicazioni è quelle del dizionario

- consente di rappresenta il concetto matematico di relazione univoca R : D → C, o *associazione chiave-valore*.
  - Insieme D è il dominio (elementi detti chiavi)
  - Insieme C è il codominio (elementi detti valori)
  - Esempio: rubrica telefonica
- deve supportare in modo efficiente le operazioni:
  - *Insert*: aggiunge un nuovo valore
  - Search: restituisce il valore associato alla chiave *key*
  - Delete: elimina il valore dalla struttura

esistono diversi modi per realizzare un dizionario tra cui

- *tabelle ad indirizzamento diretto*
- *tabelle hash*

Tabelle ad indirizzamento diretto

Il dizionario si gestisce in maniera semplice sfruttando un '*array di puntatori*'

- Dato l'insieme delle chiavi U={k_1, k_2, … k_m} possiamo usare un array *T* in cui associo ad ogni posizione (cella) una chiave k_i
- *T*[*k*] è un puntatore al valore associato alla chiave *k*
  - *il valore può essere di diversi tipi e lo chiameremo genericamente record*
- Se la tabella non contiene un record con chiave *k* allora *T*[*k*] = *nil.*

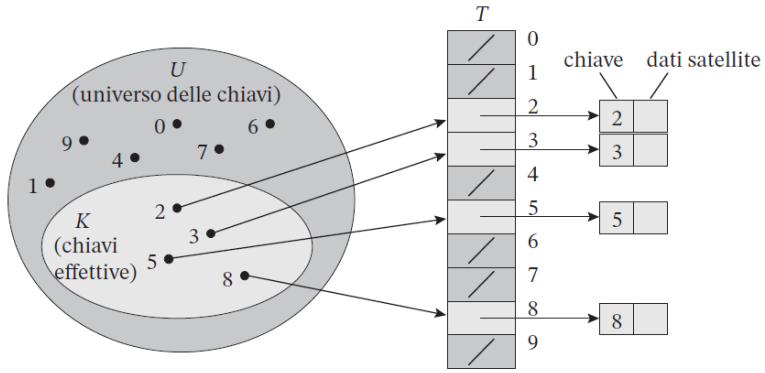


Tabelle ad indirizzamento diretto

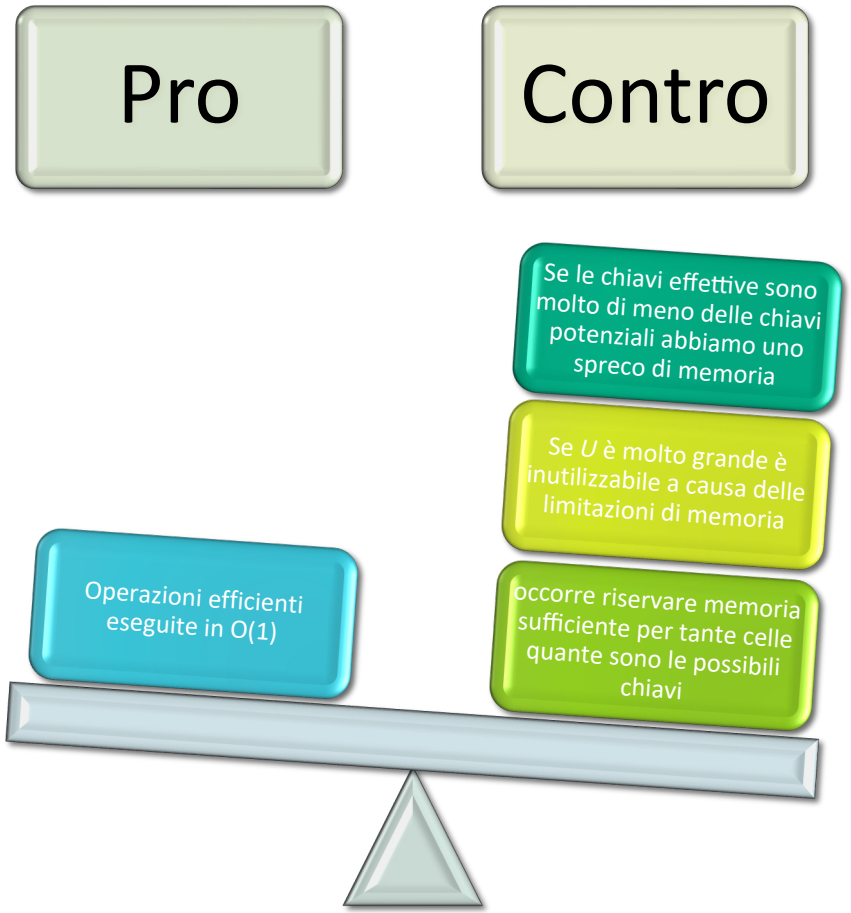
Le operazioni caratteristiche della struttura dati possono essere realizzate in modo molto semplice utilizzando gli indici dell'array ed accedendo direttamente

```
Search(T,k)
  return T[k]
```

```
Insert(T,x)
  T[x.key] = x
```

```
Delete(T,x)
  T[x.key] = nil
```

Tabelle ad indirizzamento diretto



ESEMPIO

Misuriamo il grado di riempimento di una tabella introducendo il fattore di carico:

$$\alpha = n/m$$

dove

- $m = |U|$ (dimensione della tabella) e
- $n = |K|$ (numero di chiavi effettivamente utilizzate)

Se nella tabella voglio memorizzare nomi di studenti indicizzati da numeri di matricola a 6 cifre ho

- $n = 100$,
- $m = 10^6$,
- $\alpha = 0{,}0001 = 0{,}01\%$

Grande spreco di memoria!

Tabelle Hash

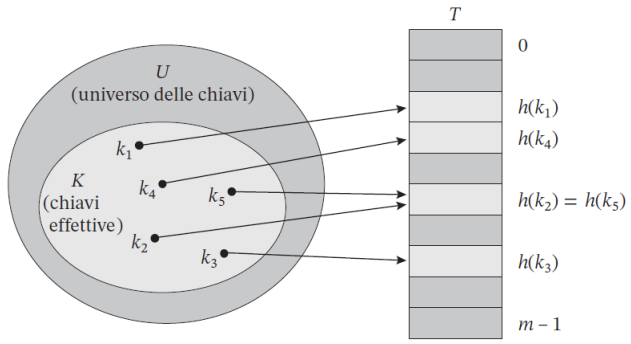
Concettualmente si realizza con gli stessi principi utilizzati per le tabelle ad indirizzamento diretto ma…

… l'associazione *chiave-cella di memoria* è definita tramite una funzione e non in maniera statica

In una *tabella hash* di m celle ogni chiave k viene memorizzata nella cella $h(k)$ usando una funzione

$$h : U \rightarrow \{0..m\text{-}1\}$$

detta *funzione di hash*.



VANTAGGIO - richiede memoria proporzionale al numero massimo di chiavi presenti nel dizionario indipendentemente dalla cardinalità dell'insieme U di tutte le possibili chiavi

Tabelle Hash- Collisioni

IL PRINCIPIO DELLA PICCIONAIA (Pigeonhol Principle)
Se *p* piccioni devono trovare posto in *c* caselle e ci sono più piccioni che caselle (*p*>*c*) allora in qualche casella entreranno almeno due piccioni.

Siccome $|U| > m$, esisteranno coppie di chiavi diverse $k_1 \neq k_2$ tali che $h(k_1) = h(k_2)$

- Diremo in questo caso che vi è una ***collisione*** tra le due chiavi k_1 e k_2.

Nessuna funzione di hash può evitare le collisioni… Quindi

- dobbiamo identificare funzioni di hash che minimizzino la probabilità delle collisioni
- dobbiamo prevedere un meccanismo di gestione delle collisioni

Risoluzione delle collisioni

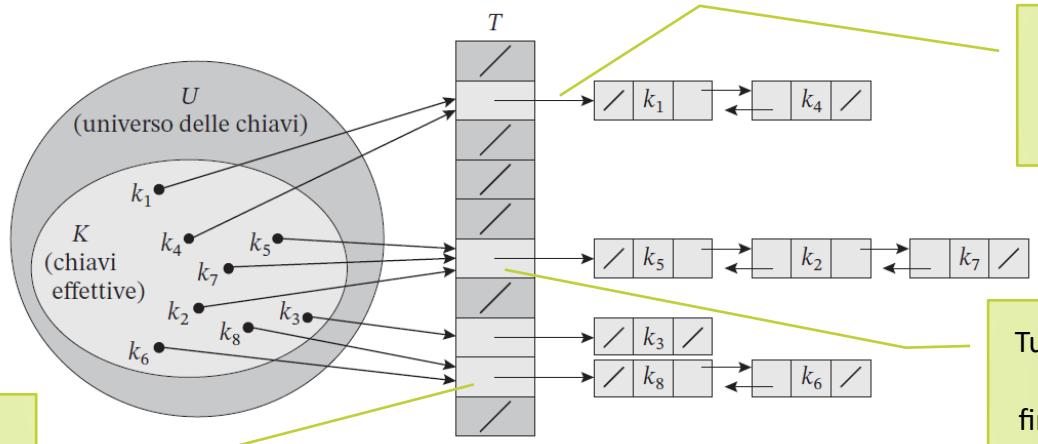
METODO DEL CONCATENAMENTO (O CON LISTE DI TRABOCCO)

Risoluzione delle collisioni

METODO DEL CONCATENAMENTO (O CON LISTE DI TRABOCCO)

IDEA

- L'insieme di input di *n* elementi viene diviso casualmente in *m* sottoinsiemi (ciascuno di cardinalità *n/m*)
- Una funzione di hash *h()* determina a quale sottoinsieme appartiene un elemento (cioè qual è la sua chiave)
- Ogni sottoinsieme (chiave) è gestito in modo indipendente tramite una lista



La cella j ha un puntatore alla testa della lista

Tutti gli elementi che la funzione h() manda nella stessa cella finiscono nella lista concatenata

Ogni cella non vuota punta a una lista concatenata

Risoluzione delle collisioni

METODO DEL CONCATENAMENTO (O CON LISTE DI TRABOCCO)

REALIZZAZIONE

Search(*T*,*k*)
  "cerca nella lista *T*[*h*(*k*)]
un elemento *x*
    tale che *x*.*key* == *k*"
  return *x*

Insert(*T*,*x*)
  "aggiungi *x* alla lista
T[*h*(*x*.*key*)]"

Delete(*T*,*x*)
  "togli *x* dalla lista
T[*h*(*x*.*key*)]"

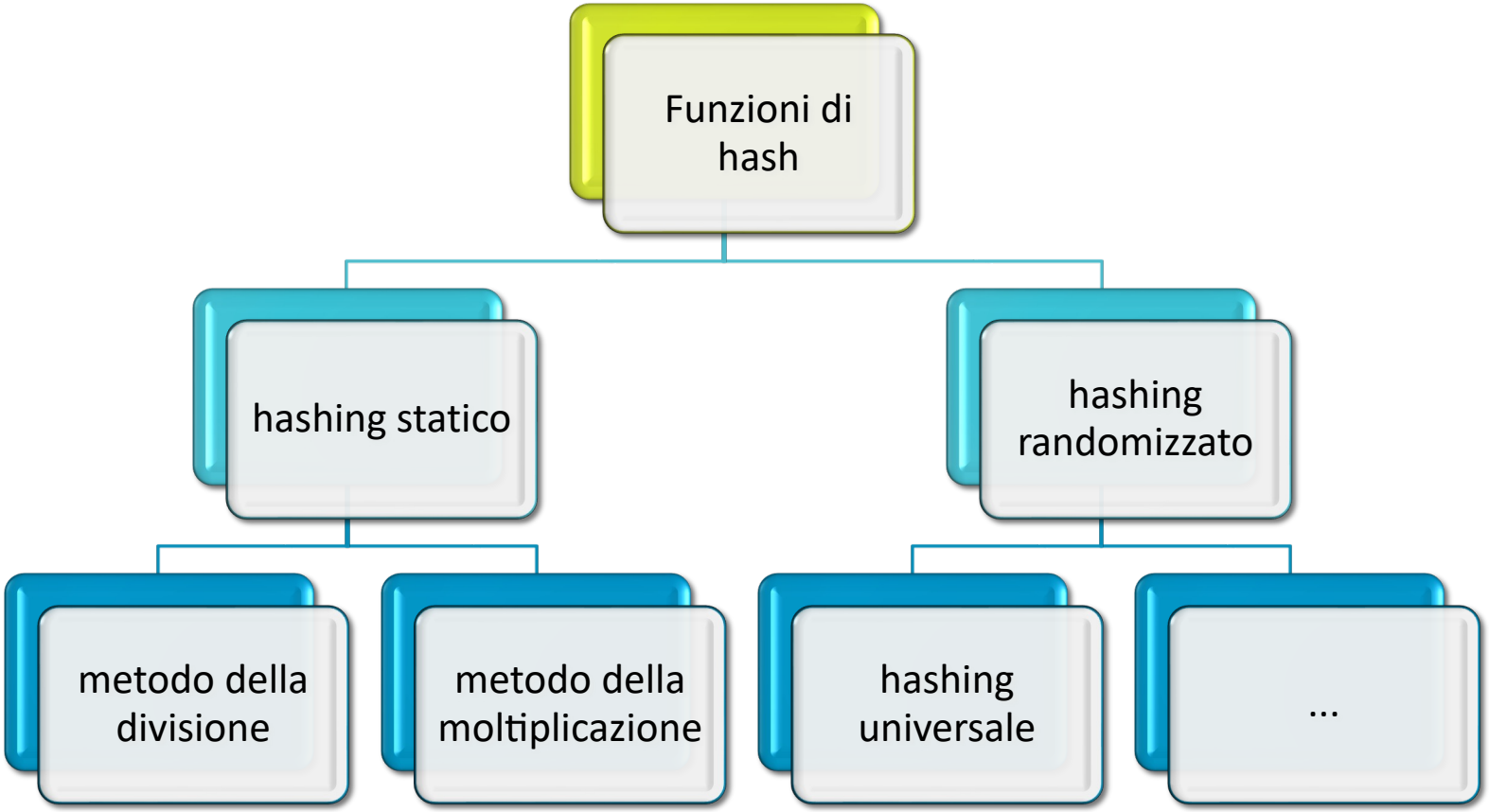
Analisi dell'hashing con concatenamento

CASO PEGGIORE

- tutte le *n* chiavi sono in conflitto e puntano tutte alla stessa cella
  - La tabella collassa su una lista di lunghezza n

Search (T, k)	Insert (T, k)	Delete (T, k)
Θ(n) + tempo di calcolo della funzione di hash	• O(1) sotto l'assunzione che x non sia già nel dizionario. • Se questa assunzione non vale allora pago il costo di una precedente search	• O(1) se le liste sono doppiamente concatenate • altrimenti costo di una search

Progettare funzioni di hash



Caratteristiche di una buona funzione di hash

Una buona funzione di hash dovrebbe soddisfare l'ipotesi di *hash uniforme indipendente*

> *ogni chiave ha la stessa probabilità 1/m di essere mandata in una qualsiasi delle m celle, indipendentemente dalle chiavi inserite precedentemente*

ESEMPIO

Consideriamo chiavi rappresentate da numeri reali x estratti casualmente e indipendentemente da una distribuzione di probabilità uniforme in $0 \leq x < 1$

$h(x) = \lfloor mx \rfloor$ soddisfa l'ipotesi di hash uniforme indipendente

Caratteristiche di una buona funzione di hash

SVANTAGGIO

- l'ipotesi di hash uniforme indipendente dipende dalle probabilità con cui vengono estratti gli elementi da inserire…
- …questa probabilità però in genere non è nota

Progettare una funzione di hash

ASSUNZIONE
le chiavi che considereremo nelle prossime slide sono rappresentate degli interi non negativi piccoli

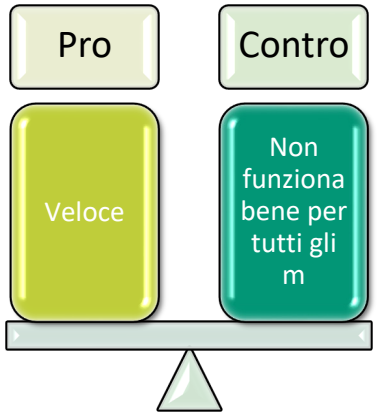
Questa assunzione non è restrittiva, in quanto ogni tipo di chiave è rappresentato nel calcolatore con una sequenza di bit e ogni sequenza di bit si può interpretare come un intero non negativo

Hashing Statico

METODO DELLA DIVISIONE

Associa ad una chiave *k* una delle *m* celle prendendo il resto della divisione intera tra *k* e *m*

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



- valori di *m* prossimi ad una potenza di 2 non funzionano bene
- una buona scelta per m è un numero primo non troppo vicino ad una potenza di 2
  - Ad esempio m = 701

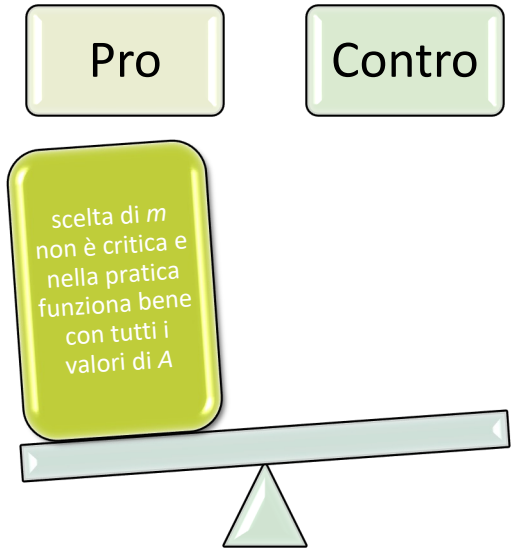
Hashing Statico

METODO DELLA MOLTIPLICAZIONE

1. moltiplica la chiave k per una costante A (con $0 < A < 1$) ed estrai la parte frazionaria di kA
2. moltiplica il valore ottenuto per m e prendi la parte intera inferiore

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

$x \bmod 1 = x - \lfloor x \rfloor$ è la parte frazionaria



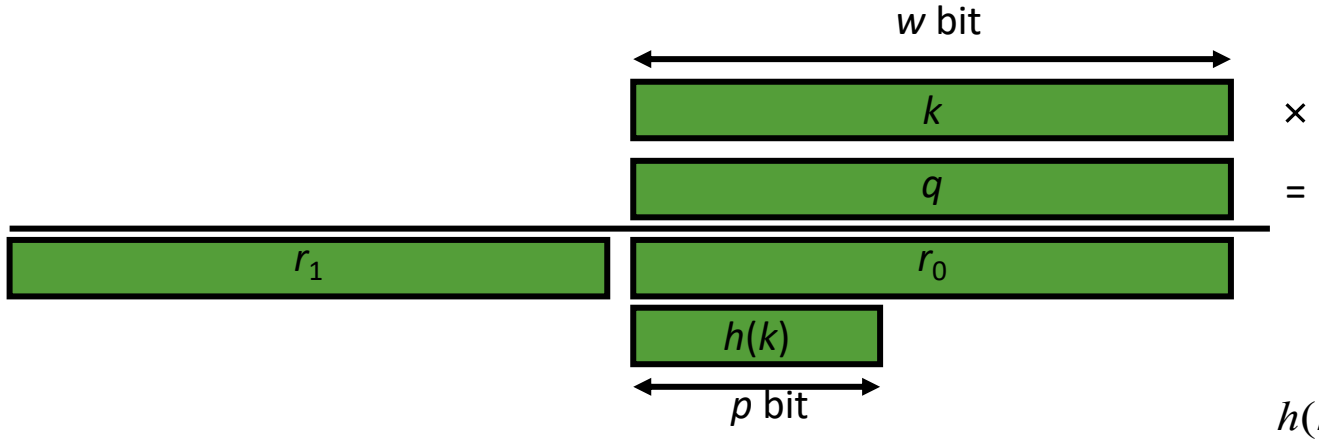
Hashing Statico

METODO DELLA MOLTIPLICAZIONE

Molto efficiente se m è una potenza di 2 (metodo moltiplica e scorri)

- $m = 2^p$
- $A = q/2^w$ (con $0 < q < 2^w$ dove w è la lunghezza di una parola di memoria)

Assumiamo che ogni chiave possa essere rappresentata da una sola parola di w bit



$$h(k) = \left\lfloor 2^p \frac{r_0}{2^w} \right\rfloor = \left\lfloor 2^p \left(\frac{kq}{2^w} \bmod 1 \right) \right\rfloor$$
$$= \lfloor m(kA \bmod 1) \rfloor$$

Hashing Statico

WARNING

Nessuna funzione di hash può evitare che un avversario malevolo inserisca nella tabella una sequenza di valori che vadano a finire tutti nella stessa lista

- Peggioramento delle prestazioni a causa della search

Più in generale, per ogni funzione di hash si possono trovare delle distribuzioni di probabilità degli input per le quali la funzione non ripartisce bene le chiavi tra le varie liste della tabella hash

Hashing Randomizzato

Possiamo usare la randomizzazione per rendere il comportamento della tabella hash indipendente dall'input.

L'idea è quella di usare una funzione di hash scelta casualmente in un insieme "*universale*" di funzioni di hash.

Questo approccio viene detto *hashing universale*.

Hashing Universale

Un insieme H di funzioni di hash che mandano un insieme U di chiavi nell'insieme $\{0,1,\ldots,m-1\}$ degli indici della tabella hash si dice ***universale*** se per ogni coppia di chiavi diverse j e k vi sono al più $|H|/m$ funzioni di hash in H tali che $h(j) = h(k)$

Scegliendo casualmente la funzione di hash in un insieme universale H

- la probabilità che due chiavi qualsiasi j e k collidano è $1/m$
- questa probabilità è la stessa che si avrebbe scegliendo casualmente le due celle in cui mandare j e k *(hashing universale indipendente)*

Progettare una famiglia universale di funzioni di hash

METODO BASATO SULLA TEORIA DEI NUMERI

1. Scegliamo un numero primo *p* maggiore di ogni possibile chiave *k*.
2. Per ogni coppia di interi (*a*,*b*) tali che $1 \le a < p$ e $0 \le b < p$ definiamo una funzione di hash:

$$h_{a,b}(k) = [(ak + b) \bmod p] \bmod m$$

Risoluzione delle collisioni

METODO DELL'INDIRIZZAMENTO APERTO

Risoluzione delle Collisioni

METODO DELL'INDIRIZZAMENTO APERTO

Con la tecnica di ***indirizzamento aperto*** tutti gli elementi stanno nella tabella

- non utilizziamo puntatori a zone esterne di memoria ma tutto risiede nella tabella
- se la tabella è piena potrei non riuscire a completare un inserimento

IDEA

- Quando voglio inserire un elemento, provo a metterlo nella sua *posizione primaria* (restituita dalla funzione hash)
- se non riesco, provo a inserirlo nella posizione secondaria, se non riesco nella terziaria e così via…

Ne segue che

- La funzione di hash non individua una singola cella ma ma deve definire un ordine in cui ispezionare tutte le celle.
- L'inserimento di un elemento avviene nella prima cella libera che si incontra nell'ordine di ispezione.

Risoluzione delle Collisioni

METODO DELL'INDIRIZZAMENTO APERTO

In questo caso, la funzione di hash è una funzione ***h*(*k*,*i*)** che al variare di ***i*** tra **0** ed ***m*-1** fornisce, per ciascuna chiave ***k***, una sequenza di indici

$$h(k,0),\ h(k,1),\ldots,\ h(k,m-1)$$

che rappresenta l'ordine di ispezione

Siccome vogliamo poter ispezionare *tutte* le celle, la sequenza deve essere una permutazione dell'insieme degli indici **0,1,..., *m*-1** della tabella.

Risoluzione delle Collisioni

METODO DELL'INDIRIZZAMENTO APERTO

```
Insert(T, k)
 i = 0
 repeat
    q = h(k, i)
   if T[ q ] == nil
      T[ q ] = k
      return  q
   else i = i +1
 until i == m
 return  "tabella piena"
```

```
Search(T, k)
  i = 0
  repeat
     q = h(k, i)
     if T[ q ] == k
       return  q
     i = i +1
  until i == m or T[ q ] == nil
  return nil
```

```
Delete(T, k)
 %più complicata e
richiede piccoli
accorgimenti
```

Risoluzione delle Collisioni

METODO DELL'INDIRIZZAMENTO APERTO - DELETE

WARNING

- Non possiamo infatti limitarci a porre *nil* nella cella, perché altrimenti interromperemmo delle sequenze di ispezione

Delete assegna alla chiave dell'elemento da togliere un particolare valore diverso da ogni possibile chiave:

```
Delete(T, i)
  T[ i ] = DELETED
```

Devo modificare la Insert

```
Insert(T, k)
 i = 0
 repeat
   j = h(k, i)
   if T[ j ] == nil or T[ j ] == DELETED
     T[ j ] = k
     return  j
   i = i+1
 until i == m
 return  "tabella piena"
```

Risoluzione delle Collisioni

METODO DELL'INDIRIZZAMENTO APERTO

OSSERVAZIONE

- Con l'indirizzamento aperto, la funzione di hash fornisce una sequenza di ispezione

Si modifica l'ipotesi di hashing indipendente e uniforme e si parla di *hashing con permutazioni indipendenti e uniformi* (più comunemente noto come hashing uniforme)

ogni chiave ha la stessa probabilità **1/*m*!** di generare una qualsiasi delle ***m*!** possibili sequenze di ispezione

Risoluzione delle Collisioni

METODO DELL'INDIRIZZAMENTO APERTO

Ci sono tre tecniche comunemente usate per determinare l'ordine di ispezione:

1. Ispezione lineare
2. Ispezione quadratica
3. Doppio hashing

Nessuna delle tre genera tutte le $m!$ sequenze di ispezione

- Le prime due ne generano soltanto m e l'ultima ne genera m^2

Ispezione Lineare

La funzione di hash ***h*(*k*,*i*)** si ottiene da una funzione di hash ordinaria ***h*'(*k*)** ponendo

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

L'esplorazione inizia dalla cella ***h*(*k*,0) = *h*'(*k*)** e continua con le celle ***h*'(*k*)+1**, ***h*'(*k*)+2**, ecc. fino ad arrivare alla cella ***m*-1**, dopo di che si continua con le celle **0**,**1**,ecc. fino ad aver percorso circolarmente tutta la tabella.

Ispezione Lineare

L’ispezione lineare è facile da implementare ma soffre del problema del clustering primario:

- i nuovi elementi inseriti nella tabella tendono ad addensarsi (formare un cluster) attorno agli elementi già presenti

Ispezione Quadratica

La funzione di hash ***h*(*k*, *i*)** si ottiene da una funzione di hash ordinaria ***h'*(*k*)** ponendo

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

dove c_1 e c_2 sono due costanti con $c_2 \neq 0$.

I valori di ***m***, c_1 e c_2 non possono essere qualsiasi ma debbono essere scelti opportunamente in modo che la sequenza di ispezione percorra tutta la tabella

Ispezione con Doppio Hashing

La funzione di hash $h(k,i)$ si ottiene da due funzioni di hash ordinarie $h_1(k)$ ed $h_2(k)$ ponendo

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

Perché la sequenza di ispezione percorra tutta la tabella, il valore di $h_2(k)$ deve essere relativamente primo con m.

Possiamo soddisfare questa condizione in diversi modi.

Ispezione con Doppio Hashing

Possiamo scegliere

- $m = 2^p$ potenza di 2
- $h_2(k) = 2\,h'(k) + 1$

con $h'(k)$ funzione di hash qualsiasi per una tabella di dimensione $m' = m/2 = 2^{p-1}$.

Possiamo scegliere m primo e $h_2(k)$ che ritorna sempre un valore minore di m

Esempio è:

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

dove m' è minore di m

- di solito $m' = m\text{-}1$

Analisi della Complessità

Recap - Analisi dell'hashing con concatenamento

CASO PEGGIORE

- tutte le *n* chiavi sono in conflitto e puntano tutte alla stessa cella
  - La tabella collassa su una lista di lunghezza n

Search (T, k)	Insert (T, k)	Delete (T, k)
Θ(n) + tempo di calcolo della funzione di hash	• O(1) sotto l'assunzione che x non sia già nel dizionario. • Se questa assunzione non vale allora pago il costo di una precedente search	• O(1) se le liste sono doppiamente concatenate • altrimenti costo di una search

Analisi dell'hashing con concatenamento

CASO MEDIO

- le prestazioni dipendono dal modo in cui la funzione di hash distribuisce le chiavi tra le celle

- Valutiamo la complessità media di Search in funzione del *fattore di carico* $\alpha = n/m$.

- OSSERVAZIONE: n è compreso tra 0 e $|U|$ -> $0 \leq \alpha \leq |U|/m$

Analisi dell'hashing con concatenamento

CASO MEDIO

- Valutiamo le prestazioni sotto l'ipotesi che la funzione hash h(k) sia uniforme e indipendente
  - i.e., $h(k)$ distribuisce in modo uniforme le n chiavi tra le m liste.

1. ogni elemento in input ha la stessa probabilità di essere mandato in una qualsiasi delle m celle

2. l'immagine tramite la funzione di hash di ogni elemento è indipendente da quella di ogni altro elemento

Analisi dell'hashing con concatenamento

CASO MEDIO

- Siano $n_0, n_1, \ldots, n_{m-1}$ le lunghezze delle m liste. La lunghezza attesa di una lista è

$$\mathrm{E}[n_j] = \frac{1}{m}\sum_{i=0}^{m-1} n_i = \frac{n}{m} = \alpha$$

TEOREMA 1

In una tabella hash le cui collisioni sono risolte tramite la tecnica del concatenamento, una ricerca senza successo richiede in media un tempo $\Theta(1 + \alpha)$, nell'ipotesi di hashing uniforme e indipendente.

Analisi dell'hashing con concatenamento

CASO MEDIO – DIMOSTRAZIONE TEOREMA 1

- Consideriamo che il calcolo della funzione h(k) sia O(1)
- *Search* calcola $j = h(k)$ (in tempo $\Theta(1)$) e poi controlla tutti gli n_j elementi della lista $T[j]$ (in tempo $\Theta(n_j)$)
- Nell'ipotesi di hash uniforme indipendente $E[n_j] = \boldsymbol{\alpha}$ e quindi l'algoritmo richiede un tempo $\Theta(1+\boldsymbol{\alpha})$ in media.

Analisi dell'hashing con concatenamento

TEOREMA 2
In una tabella hash le cui collisioni sono risolte tramite la tecnica del concatenamento, una ricerca con successo richiede in media un tempo $\Theta(1 + \alpha)$, nell'ipotesi di hashing uniforme e indipendente.

DIMOSTRAZIONE TEOREMA 2

- Assumiamo che ogni chiave presente nella tabella abbia la stessa probabilità di essere la chiave cercata
- Una ricerca di un una chiave *k* presente nella tabella richiede
  - il calcolo dell'indice $j = h(k)$,
  - il test sulle chiavi che precedono *k* nella lista $T[j]$
  - infine il test su *k*
  - -> numero operazioni = 2 + numero elementi che precedono *k* nella lista

Analisi dell'hashing con concatenamento

DIMOSTRAZIONE TEOREMA 2 (cont)

- Le chiavi che precedono *k* nella lista *T*[*j*] sono quelle che sono state inserite dopo di *k*
  - Supponiamo che $k = k_i$ sia l'*i*-esima chiave inserita nella lista
- Per $j = i+1,\ldots,n$ sia $X_{i,j}$ la variabile casuale che vale 1 se k_j viene inserita nella stessa lista di *k* e 0 altrimenti

$$X_{i,j} = \begin{cases} 0 & \text{se } h(k_j) \neq h(k_i) \\ 1 & \text{se } h(k_j) = h(k_i) \end{cases}$$

- Nell'ipotesi di hashing uniforme indipendente: $E(X_{i,j}) = 1/m$.

Analisi dell'hashing con concatenamento

DIMOSTRAZIONE TEOREMA 2 (cont)

- Il valore atteso del numero medio di operazioni eseguite è

$$\mathrm{E}[\frac{1}{n}\sum_{i=1}^{n}(2+\sum_{j=i+1}^{n}X_{i,j})]=\frac{1}{n}\sum_{i=1}^{n}(2+\sum_{j=i+1}^{n}\mathrm{E}[X_{i,j}])$$

$$=\frac{1}{n}\sum_{i=1}^{n}(2+\sum_{j=i+1}^{n}\frac{1}{m})=2+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=2+\frac{1}{nm}\sum_{t=0}^{n-1}t=2+\frac{1}{nm}\frac{n(n-1)}{2}$$

$$=2+\frac{n-1}{2m}=2+\frac{\alpha}{2}-\frac{1}{2m}=\Theta(1+\alpha)$$

Take away da Teorema 1 e Teorema 2

Se il numero n di elementi nella tabella è proporzionale al numero di celle m (ossia n=cm per qualche costante c)

- $\alpha = n/m = cm/m$ -> costante

- Il costo della ricerca è $\Theta(1+\alpha) = \Theta(1)$.

- RECALL
  - l'inserimento e la cancellazione con liste doppiamente collegate richiedono un tempo O(1)

  - La tabella si comporta particolarmente bene

Hashing Randomizzato

COROLLARIO

Utilizzando l'hashing universale e risolvendo le collisioni tramite concatenamento in una tabella inizialmente vuota di m celle, il tempo atteso per gestire qualsiasi sequenza di s operazioni INSERT, SEARCH e DELETE contenente n = O(m) operazioni INSERT è Θ(s).

DIMOSTRAZIONE

- Segue dal TEOREMA 2 osservando che usando una funzione di hash universale ho una probabilità di collisione tra due chiavi di al più 1/m che mi porta ad avere operazioni di costo medio costante

Progettare una famiglia universale di funzioni di hash

METODO BASATO SULLA TEORIA DEI NUMERI

Teorema

La famiglia di funzioni

$$H_{pm} = \{h_{a,b} : 1 \le a < p, 0 \le b < p\} \text{ con p numero primo}$$

$$h_{a,b}(k) = ((ak+b) \bmod p) \bmod m$$

è universale.

Progettare una famiglia universale di funzioni di hash

*Teorema**

la famiglia di funzioni

$$H = \{h_a : a \text{ è dispari}, 1 < q < m\} \text{ dove}$$

$h_a(k) = (ka \bmod 2^w)$ ottenuta tramite metodo moltiplica e scorri

è 2/m-universale.

* Di questo teorema omettiamo la dimostrazione

Tabella hash Adattiva

Indipendentemente dalla strategia per la gestione delle collisioni, non conviene far crescere troppo α

Possiamo sostituire l'array con un «array dinamico»

- Sopra una certa soglia di carico, si raddoppia la dimensione dell'array
- Si reinseriscono tutte le chiavi
- Nel caso di indirizzamento aperto, in fase di nuovo inserimento si rimuovono anche tutti gli elementi marcati DELETED

Il costo ammortizzato è O(1)

Esercizi

Esercizio 1

Consideriamo una funzione hash h: U → {0, 1, ..., m-1} . h si dice *perfetta* se per ogni coppia di chiavi distinte x e y, si ha h(x) ≠ h(y).

1. Scrivere un algoritmo per il calcolo di una funzione hash perfetta in cui l'insieme universo sia costituito da sequenze di cinque caratteri scelti tra i seguenti: 'A', 'C', 'T', 'G' e il valore di m sia minimo possibile.

2. Determinare il costo computazionale dell'algoritmo di cui al punto precedente.

Esercizio 2

Sia dato l'insieme di chiavi K = {25, 83, 57, 26, 13, 60, 93, 4}, e sia m=10.

Si consideri una tabella hash (inizialmente vuota) di dimensione m.

Mostrare come cambia la struttura della tabella ogni volta che si inserisce una delle chiavi di K, nell'ordine indicato

- La funzione hash utilizzata è h(k) = k mod m.
- La tabella utilizza il metodo delle liste collegate per gestire le collisioni
  - assumere che gli elementi vengano sempre inseriti in fondo alle liste.

Esercizio 3

Si consideri una tabella hash memorizzata in un vettore A[0..m - 1] con m = 10 slot in cui le collisioni sono gestite mediante indirizzamento aperto.

L'insieme delle chiavi è costituito da interi non negativi.

La funzione hash adottata è h(k, i) = ((k mod m) + i) mod m, essendo k il valore della chiave e i il contatore del “tentativo di inserimento”.

Mostrare il contenuto del vettore A dopo ciascuna delle operazioni seguenti:

1. INSERT(7)
2. INSERT(19)
3. INSERT(37)
4. DELETE(7)
5. INSERT(21)
6. DELETE(37)
7. INSERT(17)
8. INSERT(11)
9. DELETE(21)