Algoritmi e Strutture Dati

Tabelle Hash

Basato su materiale di C. Demetrescu, I. Finocchi, G.F. Italiano

Implementazioni Dizionario

- Liste	O(n)
- Alberi di ricerca non bilanciati	O(n)
- Alberi di ricerca bilanciati	O(log n)
- Tabelle hash	O(1)

Tabelle ad accesso diretto

Sono dizionari basati sulla proprietà di accesso diretto alle celle di un array

Idea:

- dizionario memorizzato in array v di m celle
- a ciascun elemento è associata una chiave intera nell'intervallo [0,m-1]
- elemento con chiave k contenuto in v[k]
- al più $n \leq m$ elementi nel dizionario

Implementazione

classe `TavolaAccessoDiretto` **implementa** `Dizionario`:
dati: $S(m) = \Theta(m)$
un array v di dimensione $m \geq n$ in cui $v[k] = elem$ se c'è un elemento $elem$ con chiave k nel dizionario, e $v[k] =$ `null` altrimenti. Le chiavi k devono essere interi nell'intervallo $[0,\ m-1]$.

operazioni:

`insert`($elem\ e, chiave\ k$) $T(n) = O(1)$
$v[k] \leftarrow e$

`delete`($chiave\ k$) $T(n) = O(1)$
$v[k] \leftarrow$ `null`

`search`($chiave\ k$) $\rightarrow elem$ $T(n) = O(1)$
return $v[k]$

Fattore di carico

Misura il grado di riempimento di una tabella:

$$\alpha = \frac{n}{m}$$

Esempio:
Tabella con i dati di 100 studenti, indicizzati da matricole a 6 cifre

n=100

$m=10^6$ **$\alpha = 0{,}0001 = 0{,}01\%$**

Grande spreco di memoria!

Pregi e difetti

Pregi:

– Tutte le operazioni richiedono tempo O(1)

Difetti:

– Le chiavi sono necessariamente interi in [0, *m*-1]
– Lo spazio usato è proporzionale ad *m*, non al numero *n* di elementi: può esserci grande spreco di memoria!

Tabelle hash

Se le chiavi non sono numeri interi si usa un'estensione delle tabelle ad accesso diretto: le **tabelle hash**

Idea:

– Chiavi prese da un universo totalmente ordinato *U*
(possono non essere numeri)
– **Funzione hash**: $h: U \rightarrow [0, m-1]$
(funzione che trasforma chiavi in indici)
– Elemento con chiave *k* in posizione *v[h(k)]*

Collisioni

Le tabelle hash sono soggette al fenomeno delle collisioni.

Si ha una collisione quando si deve inserire nella tabella hash un elemento con chiave *u*, e nella tabella esiste già un elemento con chiave *v* tale che **h(u)=h(v)**:

il nuovo elemento andrebbe a sovrascrivere il vecchio!

Funzioni hash perfette

Un modo per evitare il fenomeno delle collisioni è usare funzioni hash perfette.

Una funzione hash si dice **perfetta** se è iniettiva, cioè per ogni u,v ∈ U:

$$u \neq v \quad \Rightarrow \quad h(u) \neq h(v)$$

Ciò implica che **|U| ≤ m**

Implementazione

classe `TavolaHashPerfetta` **implementa** `Dizionario`:
dati: $S(m) = \Theta(m)$
un array v di dimensione $m \geq n$ in cui $v[h(k)] = e$ se c'è un elemento e con chiave $k \in U$ nel dizionario, e $v[h(k)] =$ `null` altrimenti. La funzione $h : U \to \{0, \ldots, m-1\}$ è una funzione hash perfetta calcolabile in tempo $O(1)$.

operazioni:

`insert`($elem\ e, chiave\ k$) — $T(n) = O(1)$
$v[h(k)] \leftarrow e$

`delete`($chiave\ k$) — $T(n) = O(1)$
$v[h(k)] \leftarrow$ `null`

`search`($chiave\ k$) $\rightarrow elem$ — $T(n) = O(1)$
return $v[h(k)]$

Funzione hash perfetta: Esempio

Tabella hash con nomi di studenti aventi come chiavi numeri di matricola nell'insieme **U**=[234717, 235717]

Funzione hash perfetta: ***h(k) = k - 234717***

n=100 m=1000 α = 0,1 = 10%

L'assunzione $|U| \leq m$ necessaria per avere una funzione hash perfetta è raramente conveniente (o possibile)…

Funzione hash non perfetta: Esempio

Tabella hash con elementi aventi come chiavi lettere dell'alfabeto **U={A,B,C,…}**

Funzione hash non perfetta (ma efficace se *m* è primo):

h(*c*) = ascii(*c*) mod *m*

per *m*=11: h('C') = h('N') ⇒ se inseriti entrambi si ha una collisione!

L'idea si può applicare a qualunque tipo di chiave

h(*k*) = *integer*(*k*) mod *m*

Uniformità delle funzioni hash

Per ridurre la probabilità di collisioni, una buona funzione hash distribuisce in modo uniforme le chiavi nello spazio degli indici della tabella

Questo si ha, ad esempio, se la funzione hash gode della proprietà di **uniformità semplice …**

Uniformità semplice

Sia $P(k)$ la probabilità che la chiave k sia presente nel dizionario (spesso si può assumere $P(k) = 1/|U|$)

Se i valori delle chiavi sono indipendenti, la **probabilità che la cella *i* sia occupata** è:

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

Una funzione hash **h** gode dell'**uniformità semplice** se:

$$Q(i) = \frac{1}{m}$$

Esempio

Se U è l'insieme dei numeri reali in [0,1] e ogni chiave ha la stessa probabilità di essere usata, allora si può dimostrare che la funzione hash:

$$h(k) = \lfloor km \rfloor$$

soddisfa la proprietà di uniformità semplice

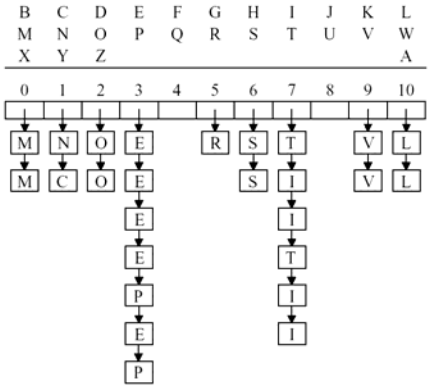
Gestione delle collisioni

Nel caso in cui non si possano evitare le collisioni, dobbiamo trovare un modo per gestirle.

Due metodi classici sono i seguenti:

1. **Liste di collisione**. Gli elementi sono contenuti in liste esterne alla tabella:
   v[i] punta alla lista degli elementi tali che *h(k)=i*
2. **Indirizzamento aperto**. Tutti gli elementi sono contenuti nella tabella:
   se una cella è occupata, se ne cerca un'altra libera

Liste di collisione



Esempio di tabella hash basata su liste di collisione contenente le lettere della parola:

PRECIPITEVOLISS IMEVOLMENTE

Lunghezza attesa delle liste = *n/m*

Implementazione

classe `TavolaHashListeColl` **implementa** `Dizionario`:
dati: $S(m,n) = \Theta(m+n)$
un array v di dimensione m in cui ogni cella contiene un puntatore a una lista di coppie $(elem, chiave)$. Un elemento e con chiave $k \in U$ è nel dizionario se e solo se (e,k) è nella lista puntata da $v[h(k)]$, con $h : U \to \{0,\ldots,m-1\}$ funzione hash con uniformità semplice calcolabile in tempo $O(1)$.

operazioni:
`insert`$(elem\ e, chiave\ k)$ $T(n) = O(1)$
aggiungi la coppia (e,k) alla lista puntata da $v[h(k)]$.

`delete`$(chiave\ k)$ $T_{avg}(n) = O(1+n/m)$
rimuovi la coppia (e,k) nella lista puntata da $v[h(k)]$.

`search`$(chiave\ k) \rightarrow elem$ $T_{avg}(n) = O(1+n/m)$
se (e,k) è nella lista puntata da $v[h(k)]$, allora restituisci e, altrimenti restituisci `null`.

Indirizzamento aperto

Supponiamo di voler inserire un elemento con chiave *k* e la sua posizione "naturale" *h(k)* sia già occupata.

L'indirizzamento aperto consiste nell'occupare un'altra cella, che potrebbe spettare di diritto ad un'altra chiave.

Cerchiamo la cella vuota (se c'è) scandendo tutte le celle secondo una sequenza di indici:

c(k,0), c(k,1), c(k,2),…c(k,m-1)

Implementazione

classe `TavolaHashAperta` **implementa** `Dizionario`:
dati: $S(m) = \Theta(m)$
un array v di dimensione m in cui ogni cella contiene una coppia $(elem, chiave)$.

operazioni:
`insert`$(elem\ e, chiave\ k)$
1. **for** $i = 0$ **to** $m-1$ **do**
2. **if** $(v[c(k,i)].elem = \texttt{null})$ **then**
3. $v[c(k,i)] \leftarrow (e,k)$
4. **return**
5. **errore** tavola piena

`delete`$(chiave\ k)$
errore operazione non supportata

`search`$(chiave\ k) \rightarrow elem$
1. **for** $i = 0$ **to** $m-1$ **do**
2. **if** $(v[c(k,i)].elem = \texttt{null})$ **then**
3. **return** `null`
4. **if** $(v[c(k,i)].chiave = k)$ **then**
5. **return** $v[c(k,i)].elem$
6. **return** `null`

Metodi di scansione

Scansione lineare:

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

per $0 \le i < m$

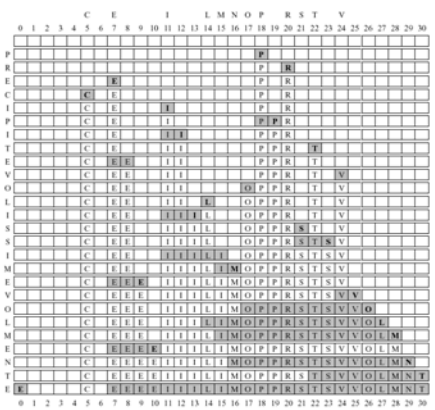
Scansione quadratica:

$$c(k,i) = (h(k) + 0.5\ i + 0.5\ i^2) \bmod m$$

per $0 \le i < m$

Agglomerazione primaria: chiavi collidenti sono associate alla stessa sequenza di indici)

Esempio



Inserimenti in tabella hash basata su indirizzamento aperto con scansione lineare delle lettere della parola:

PRECIPITEVOLISSIMEVOLMENTE

4,8 celle scandite in media per inserimento

Metodi di scansione: hashing doppio

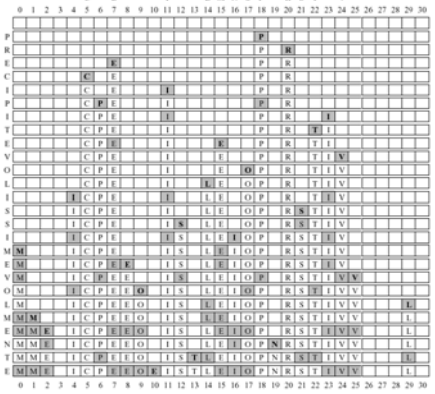
La scansione lineare provoca effetti di **agglomerazione secondaria**, cioè lunghi gruppi di celle consecutive occupate che rallentano la scansione

L'hashing doppio riduce il problema:

$$c(k,i) = \lfloor h_1(k) + i \cdot h_2(k) \rfloor \bmod m$$

per $0 \le i < m$, h_1 e h_2 funzioni hash

Esempio



Inserimenti in tabella hash basata su indirizzamento aperto con hashing doppio delle lettere della parola:

PRECIPITEVOLISSIMEVOLMENTE

3,1 celle scandite in media per inserimento

Analisi del costo di scansione

Tempo richiesto in media da un'operazione di ricerca di una chiave, assumendo che le chiavi siano prese con probabilità uniforme da U:

esito ricerca	*sc. lineare*	*hashing doppio*
chiave trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)}$	$-\frac{1}{\alpha}\log_e(1-\alpha)$
chiave non trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$	$\frac{1}{1-\alpha}$

dove α=n/m (fattore di carico)

Cancellazione con indir. aperto

classe `TavolaHashApertaBis` **implementa** `Dizionario`:
dati: $S(m) = \Theta(m)$
un array v di dimensione m in cui ogni cella contiene una coppia $(elem, chiave)$.

operazioni:

`insert`$(elem\ e, chiave\ k)$
1. **for** $i = 0$ **to** $m-1$ **do**
2. **if** $(v[c(k,i)].elem =$ `null` **or** $v[c(k,i)].elem =$ `canc`$)$ **then**
3. $v[c(k,i)] \leftarrow (e,k)$
4. **return**
5. **errore** tavola piena

`delete`$(chiave\ k)$
1. **for** $i = 0$ **to** $m-1$ **do**
2. **if** $(v[c(k,i)].elem =$ `null`$)$ **then**
3. **errore** chiave non in dizionario
4. **if** $(v[c(k,i)].chiave = k$ **and** $v[c(k,i)].elem \neq$ `canc`$)$ **then**
5. $v[c(k,i)].elem \leftarrow$ `canc`
6. **errore** chiave non in dizionario

`search`$(chiave\ k) \rightarrow elem$
1. **for** $i = 0$ **to** $m-1$ **do**
2. **if** $(v[c(k,i)].elem =$ `null`$)$ **then**
3. **return** `null`
4. **if** $(v[c(k,i)].chiave = k$ **and** $v[c(k,i)].elem \neq$ `canc`$)$ **then**
5. **return** $v[c(k,i)].elem$
6. **return** `null`

Riepilogo

- Tabelle ad accesso diretto consentono di realizzare **dizionari con operazioni in tempo O(1)**
  - elementi indicizzati con le loro stesse chiavi (purché intere)
  - La dimensione dell'array è pari al numero di chiavi distinte!
- F**unzioni hash** trasformano chiavi (anche non numeriche) in indici
  - Usando funzioni hash possono aversi **collisioni**
- Tecniche classiche per risolvere le collisioni sono **liste di collisione** e **indirizzamento aperto**