lezione 8

Tabelle Hash

Tipo di dato Dizionario

Il tipo di dato Dizionario permette di gestire collezioni di elementi a cui sono associate chiavi prese da un dominio totalmente ordinato.
I dati ammessi sono quindi un insieme di coppie (elemento, chiave), mentre le operazioni tipiche sono inserimento, cancellazione e ricerca.

In generale non si fanno assunzioni su:
-unicità delle chiavi presenti nel dizionario
-ordinamento delle chiavi all'interno del dizionario
Questo tipo di assunzioni vengono delegate alle particolari realizzazioni del tipo di dato.

Vediamo in dettaglio la specifica…

Dizionario: specifica

tipo di dato: Dizionario
dati:
un insieme S di coppie (elem, chiave)
operazioni:
insert(elem el, chiave k)
aggiunge a S una nuova coppia (e,k)

delete(chiave k)
cancella da S la coppia con chiave k

elem search(chiave k)
se la chiave k è presente in S restituisce l'elemento e ad essa associato. Altrimenti restituisce null.

Possibili realizzazioni del tipo di dato Dizionario

Con array:
-inserimento O(n)
-cancellazione O(n)
-ricerca O(n)

Con liste concatenate:
-inserimento O(n)
-cancellazione O(n)
-ricerca O(n)

Con Tabelle Hash:
-inserimento ?
-cancellazione ?
-ricerca ?

Con alberi…

Tabelle ad accesso diretto

Vogliamo pensare ad una realizzazione efficiente del tipo di dato dizionario sapendo che le chiavi associate agli *n* elementi da memorizzare sono numeri interi compresi in [0…m-1], con n≤m. Supponiamo di ammettere solo chiavi distinte. Allora:
-dizionario realizzato con un array diz di dimensione m
-chiavi degli elementi usati come indici dell'array
-diz[k] = null se k non ha un elemento associato

Complessità delle operazioni del dizionario:
-insert(key, elem): diz[key] = elem O(1)
-search(key): return diz[key] O(1)
-delete(key): diz[key] = null O(1)

Realizzazione efficientissima! Tuttavia, se m>>n si ha un'enorme spreco di memoria

Fattore di carico

Misuriamo il grado di riempimento della tabella attraverso il *fattore di carico.*

Definiamo *fattore di carico* di una tabella il rapporto α= n/m tra il numero degli elementi memorizzati in tabella e la sua dimensione m.

Esempio: dizionario degli studenti che frequentano il corso di labasd, in modo da poterli cercare facilmente per numero di matricola. Assumendo un numero di matricola a 6 cifre abbiamo una tavola ad accesso diretto con fattore di carico α= n/m = 100/1000000 = 0,0001 = 0,01%. Troppo spreco di memoria!

Tabelle hash

Sia U l'universo delle chiavi associabili agli elementi del dizionario. Possiamo usare l'approccio appena visto se U non contiene interi in [0…m-1], o i numeri non sono interi, o non sono numeri?

Idea: non usiamo direttamente le chiavi come indici della tabella… usiamo una loro trasformazione tramite un'opportuna *funzione hash*.

Una *funzione hash* (= tritare, sminuzzare) è una funzione

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

che trasforma chiavi in indici di una tabella.
Chiamiamo tabella hash una tabella indicizzata tramite funzioni hash.

Funzioni hash perfette

Una funzione hash $h: U \rightarrow \{0\ldots m-1\}$ è *perfetta* se è *iniettiva*, cioè se per ogni $u,v \in U$, $u \neq v \Rightarrow h(u) \neq h(v)$.

Si noti che, affinchè una funzione hash sia perfetta, deve essere $|U| \leq m$, cioè ci deve essere spazio per tanti elementi quante sono le chiavi possibili. In tal caso una possibile realizzazione del dizionario è la seguente:

-insert(key,elem): diz[h(key)] = elem O(1)
-delete(key): diz[h(key)] = null O(1)
-search(key): return diz[h(key)] O(1)

Esempio: gli studenti che frequentano labasd hanno tutti matricola compresa tra 81000 e 82000, allora |U| = 1000 e il fattore di carico è α =n/m= 100/1000 = 10%. La funzione hash, dato il numero di matricola, toglie semplicemente il valore 81000.

Uso funzioni hash perfette?

In molti casi pratici l'universo delle chiavi è molto grande, e dunque l'uso di funzioni hash perfette richiede comunque un grande spreco di memoria. Se una funzione hash non è perfetta, allora esistono due o più chiavi diverse che hanno associato lo stesso indice, cioè abbiamo una *collisione*.
Usare tabelle hash non perfette rende quindi necessario applicare delle strategie di risoluzione delle collisioni.
Minori sono le probabilità di collisione e migliori saranno le prestazioni del dizionario.
Misuriamo la bontà di una funzione hash in base al numero atteso di collisioni che si avrebbero assumendo di inserire nel dizionario chiavi prese a caso dall'universo U.

Si noti comunque che, indipendentemente dalla funzione hash adottata, se sono possibili chiavi duplicate all'interno del dizionario le collisioni sono inevitabili.

Definizione di funzioni hash

Assumiamo di sapere che ciascuna chiave $k \in U$ abbia probabilità $p(k)$ di essere presente nel dizionario. Per funzionare bene una funzione hash deve distribuire uniformemente le chiavi nello spazio degli indici $\{0 \ldots m-1\}$.
Sia $Q(i) = \sum_{k:\, h(k)=i} p(k)$ la probabilità che, scegliendo una chiave a caso, questa finisca nella cella i. Una funzione hash h gode della proprietà di *uniformità semplice* se, per ciascun indice $i \in \{0 \ldots m-1\}$, $Q(i) = 1/m$.

In tal caso ogni cella ha la stessa probabilità di essere usata, cioè non ci sono fenomeni di *agglomerazione*, dove più chiavi tendono a collidere su alcune celle più che su altre.

Esempio: U insieme dei reali in [0,1] e ogni chiave ha la stessa probabilità di essere scelta. Allora si puo' dimostrare che
$h(k) = \lfloor km \rfloor$ soddisfa la proprietà di uniformità semplice.

Definizione di funzioni hash

Per definire una funzione hash che soddisfi l'uniformità semplice bisogna conoscere la distribuzione di probabilità P delle chiavi ma questo non è sempre possibile. Allora si assume l'equiprobabilità delle chiavi, cioè P uniforme e p(k)=1/|U|.

Da qui in avanti assumiamo:
- U con P uniforme;
- chiavi intere non negative.

Per trattare chiavi arbitrarie (non intere, negative, non numeriche) osserviamo che ci si può sempre ricondurre al caso di chiavi intere non negative.
Es: stringa --> rappresentazione binaria.

Definizione di funzioni hash

Metodo della divisione: **h(k) = k mod m** dove m è la dimensione della tabella. La bontà del metodo dipende dalla scelta di m.

Esempio: se le chiavi sono tutte multiple di m allora h(k) = 0 per ogni chiave h e quindi Q(0) = 1 e Q(i) = 0 per ogni $i \in \{1 \ldots m-1\}$.

In generale, bisogna evitare che m sia una potenza di due. Infatti, se $m = 2^p$ allora h(k) rappresenta solo i p bit meno significativi di k. Questo limita la casualità di h perché diventa funzione di una porzione della chiave.

La funzione hash dovrebbe essere funzione di tutti i bit della chiave; una buona scelta per m è un numero primo non troppo vicino ad una potenza di due.

Definizione di funzioni hash

Metodo del ripiegamento: supponiamo di suddividere la rappresentazione di una chiave k in parti $k_1,k_2,\ldots,k_l$ e di applicare **$h(k) = f(k_1,k_2,\ldots k_l)$** dove f è una opportuna funzione a più variabili e codominio {0,…,m-1}.

Esempio: supponiamo che le chiavi siano numeri di carte di credito k= 4772 6455 3393 2293. Allora possiamo ottenere k_1= 4772 k_2= 6455, k_3= 3393, k_4= 2293 e definire $h(k) = f(k_1,k_2,k_3,k_4) = (k_1+k_2+k_3+k_4) \bmod m$.

Risoluzione delle collisioni

Ci sono due metodi principali per risolvere le collisioni:
- liste di collisione
- indirizzamento aperto

Vediamoli nel dettaglio…

Risoluzione collisioni: liste di collisione

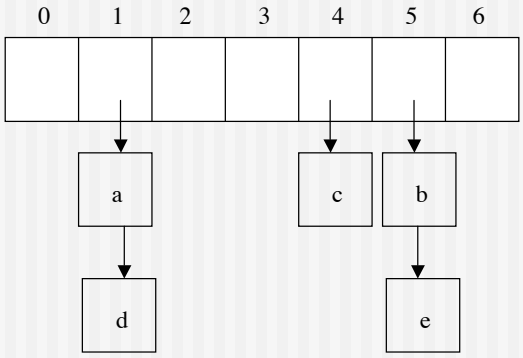
Il metodo consiste nell'associare ad ogni cella della tavola hash una lista di chiavi, detta lista di collisione, piuttosto che una singola chiave.

Quindi se due chiavi collidono sulla stessa cella verranno a trovarsi nella stessa lista di collisione.

Liste di collisione: esempio

Data una tabella vuota con 7 posizioni, effettuiamo i seguenti inserimenti (per semplicità si indicano solo le chiavi), mantenendo le liste di collisione ordinate rispetto alla chiave:

a --> hash(a) = 1
b --> hash(b) = 5
c --> hash(c) = 4
d --> hash(d) = 1 collisione
e --> hash(e) = 5 collisione!



Liste di collisione:osservazioni

Si osservi che:

- diversamente dalle tabelle ad accesso diretto e dalle tabelle hash con funzione hash perfetta, con le liste di collisione possiamo avere fattore di carico $\alpha>1$;

- assumendo la distribuzione uniforme delle chiavi, la lunghezza media delle liste di collisione sarà pari al fattore di carico $\alpha= n/m$

-se sono ammesse chiavi duplicate, si può mantenere la lista non ordinata e l'inserimento ha complessità costante;

-se non sono ammesse chiavi duplicate mediamente conviene tenere la lista ordinata in base alla chiave.

Liste di collisione: complessità

Vediamo la complessità delle operazioni del dizionario usando una tabella hash con liste di collisione. Supponiamo che la distribuzione di probabilità delle chiavi sia uniforme e che siano ammesse nel dizionario chiavi duplicate.

-insert(key) $T(n) = O(1)$ NB: avg = caso medio
-search(key) $T_{avg}(n) = O(1+n/m) =O(1+\alpha)$
-delete(key) $T_{avg}(n) = O(1+n/m) =O(1+\alpha)$

Quindi, per $n = O(m)$ abbiamo nel caso medio costo costante.

Questo tipo di tabelle forniscono un ottimo esempio di bilanciamento spazio-tempo: per m=1, tutte le n chiavi sono in una sola lista e la tabella diventa un'unica struttura a ricerca sequenziale che richiede tempo $T(n,m) = O(n/m) = O(n)$ per ogni ricerca e spazio $S(n) = O(n)$. Se invece siamo disposti ad usare molto spazio, possiamo usare una funzione hash perfetta ottenendo $T(n) = O(1)$ per la ricerca, ma spazio $S(n) = O(|U|)$.

Risoluzione collisioni: indirizzamento aperto

Questo metodo mantiene tutte le chiavi nelle celle della tavola, per cui si ha sempre $n \leq m$ e $\alpha \leq 1$.

Se h(k) corrisponde ad una cella già occupata la tecnica di indirizzamento aperto consiste nell'occupare un'altra cella vuota, anche se essa potrebbe spettare di diritto ad un'altra chiave. NOTA: una cella è vuota se il suo campo elem è null.

Esempi d'uso pratico di questa tecnica: posti prenotati al cinema, in treno etc..

Questa tecnica presuppone di non fare mai cancellazioni!

Indirizzamento aperto:insert

insert(key,elem): se diz[h(key)] è vuota si assegna la nuova coppia (key,elem), altrimenti si ispezionano le celle della tabella secondo una opportuna sequenza di indici

c(key,0)=h(key), c(key,1), …, c(key,m-1)

fermandosi alla prima cella vuota, a cui si assegnerà la coppia (key,elem). La sequenza c(key,i) considerata deve contenere tutti gli indici in {0,…,m-1} in modo da garantire che, se esiste una cella vuota, il metodo prima o poi la trova.

Indirizzamento aperto: search

search(key, elem): simile all'inserimento. Al primo null nella sequenza di indici si ferma.
La correttezza di search si basa sul fatto che se j è il più piccolo intero per cui diz[c(key,j)].elem=null allora diz[c(key,q)].chiave≠k per ogni j<q<m.
(si basa anche sul fatto che non si possono fare cancellazioni!)

Complessità delle operazioni: dipende dalle funzioni c(key,i) utilizzate. Vediamo quali funzioni si possono utilizzare…

Indirizzamento aperto: tecniche di scansione

Scansione lineare: la sequenza di scansione è definita da

$$c(key,i) = (h(key) + i) \bmod m \qquad \text{per } 0 \le i < m.$$

Le funzioni coprono tutti gli indici in {0,…,m-1}.

Ci possono essere casi di agglomerazione, che viene detta *agglomerazione primaria*. Ad esempio, k1≠k2 e h(k1)=h(k2) generano la stessa sequenza.

Si osservi inoltre che:
- la prima posizione esaminata determina l'intera sequenza di scansione. Quindi ci sono m sequenze di scansione distinte;
- il numero ottimo è m! ed è dato dall'ipotesi di uniformità della funzione hash: ognuna delle m! permutazioni di <0,…,m-1> sono equiprobabili.

Indirizzamento aperto & scansione lineare: esempio

Data una tabella vuota con 7 posizioni, effettuiamo i seguenti inserimenti (per semplicità si indicano solo le chiavi):

a --> hash(a) mod 7 = 1
b --> hash(b) mod 7 = 5
c --> hash(c) mod 7 = 4
d --> hash(d) mod 7 = 1 collisione!
(hash(d) + 1) mod 7 = 2 libera!
e --> hash(e) mod 7 = 5 collisione!
(hash(e) + 1) mod 7 = 6 libera!
f --> hash(f) mod 7 = 5 collisione!
(hash(f) + 1) mod 7 = 6 collisione!
(hash(f) + 2) mod 7 = 0 libera!

0	1	2	3	4	5	6
f	a	d		c	b	e

Indirizzamento aperto: tecniche di scansione

Scansione quadratica: la sequenza di scansione è definita da

$$c(key,i) = \lfloor h(key) + c_1 \cdot i + c_2 \cdot i^2 \rfloor \bmod m \quad \text{per } 0 \leq i < m.$$

Considera tutto lo spazio degli indici?
Ad esempio, si può dimostrare che per $c_1 = c_2 = 0{,}5$ e m potenza di due, la sequenza c(key,i) contiene tutti gli indici in {0,…,m-1}.

In questo modo si evita l'agglomerazione primaria ma ogni coppia di chiavi k1 e k2 tali che h(k1)=h(k2) continua a generare la stessa sequenza di scansione. Questo dà luogo alla cosiddetta *agglomerazione secondaria.*

Anche in questo caso la prima posizione esaminata determina l'intera sequenza di scansione.

Indirizzamento aperto: tecniche di scansione

Hashing doppio: facciamo in modo che anche il passo di incremento dipenda dalla chiave:

$$c(key,i) = (h1(key) + i \cdot h2(key)) \bmod m \qquad \text{per } 0 \le i < m.$$

dove h1 e h2 sono funzioni distinte.
In questo modo, anche se due chiavi collidono, la sequenza di scansione sarà diversa.

Affinchè c(key,i) copra tutti gli indici in {0,…,m-1}, è necessario che m e h2(key) siano primi tra loro.
Ad esempio, possiamo scegliere:
-m primo
-h1(key) = key mod m
-h2(key) = [key mod (m-1)] +1

Indirizzamento aperto: tecniche di scansione

Hashing doppio: osservazioni

- La prima posizione esaminata è h1(k) mod m; ogni posizione esaminata successivamente è distanziata dalla precedente di una quantità h2(k) mod m
- La sequenza di scansione dipende da k in due modi: a seconda della chiave, possono variare sia la posizione iniziale che il passo
- L'hashing doppio non soffre di fenomeni di agglomerazione perché il passo è casuale inoltre ...
- Ogni possibile coppia (h1(k), h2(k)) produce una sequenza di scansione distinta: ci sono quindi $O(m^2)$ sequenze di scansione distinte (miglioramento rispetto alla scansione lineare o quadratica!)

Indirizzamento aperto & cancellazione chiavi

Supponiamo di voler cancellare una chiave da una tabella hash con indirizzamento aperto.

Se marchiamo la cella corrispondente come vuota (mettendo il suo campo elem a null) perdiamo la correttezza dell'operazione search.

Infatti, la ricerca potrebbe interrompersi prima di trovare l'elemento cercato per via di un "buco" creato da una cancellazione.

Allora come si può fare?

Indirizzamento aperto & cancellazione chiavi

Una soluzione possibile è quella di marcare le celle che hanno subito una cancellazione con un valore speciale, ad esempio ponendo il campo elem a "canc".

Se sono ammesse chiavi duplicate, l'inserimento considererà una cella cancellata come vuota fermandosi su di essa, mentre la ricerca la oltrepasserà. Se non sono ammesse chiavi duplicate, anche l'inserimento non può fermarsi alla prima cella cancellata, ma deve continuare comunque la scansione.

Questa soluzione tende a rallentare le operazioni di inserimento (se non sono ammesse chiavi duplicate) e ricerca, poiché celle cancellate dovranno comunque essere esaminate senza contribuire in alcun modo al successo o all'insuccesso di una ricerca. In particolare le operazioni non dipendono più solamente dal fattore di carico della tabella hash.

Indirizzamento aperto: costo di scansione

La scansione per la ricerca di un elemento o di una cella vuota può richiedere tempo O(n) nel caso peggiore.
Nel caso medio le cose vanno meglio. Infatti, si può dimostrare che:

se le chiavi associate agli n elementi di una tavola hash di dimensione m sono prese dall'universo delle chiavi con probabilità uniforme, il numero medio di passi richiesto da una operazione search (contando in n anche le celle marcate canc) è:

Esito ricerca	sc. lineare	sc. quadratica/hashing doppio
chiave trovata	$1/2 + 1/2(1-\alpha)$	$(-1/\alpha)\ \log_e(1-\alpha)$
chiave non trovata	$1/2 + 1/2(1-\alpha)^2$	$1/(1-\alpha)$

Indirizzamento aperto: costo di scansione

Osservazioni:
-il costo di insert è al più quello di search senza successo.
-il costo di delete è dominato dalla ricerca della chiave da cancellare, e quindi coincide con il costo di search.

Allora, partendo dal caso di ricerca con insuccesso e usando la definizione $\alpha = n/m$, possiamo concludere che il tempo medio richiesto per le operazioni search, insert e delete è
-$O(m/(m-n)^2)$ nel caso di scansione lineare e
-$O(m/(m-n))$ nel caso di scansione quadratica e hashing doppio

Inoltre:
- se la tavola è piena a metà il numero medio di ispezioni per la ricerca senza successo è 1/(1-0,5) = 2
-se la tavola è piena al 90% il numero medio di ispezioni per la ricerca senza successo è 1/(1-0,9) = 10
Quindi più la tabella si riempie e più basse diventano le sue prestazioni

Tabelle Hash in Java

Java mette a disposizione una funzione hash nella classe Object, attraverso il metodo hashCode().
Come sempre, il metodo può essere riscritto dalle sottoclassi, che però dovrebbero dare garanzia della bontà della distribuzione delle chiavi ottenuta.

Esiste anche la classe java.util.Hashtable che realizza una tabella hash.
La classe Hashtable contiene un interessante metodo *rehash*, che incrementa la capacità della tabella hash e la riorganizza per poter memorizzare e accedere i suoi dati in modo più efficiente.