Hash e tabelle hash

Materia	Algoritmi
Anno	Secondo Anno
Data	@October 29, 2024

Le **tabelle hash** servono per implementare una struttura astratta nota come **dizionario**, un insieme all'interno del quale si fa una operazione principale, ossia la **ricerca**. Sono ammesse anche altre operazioni, quali inserimento e cancellazione, ma queste operazioni sono in genere più rare rispetto alla ricerca.
Solitamente un dizionario si utilizza per verificare se un elemento è presente o meno, o si restituisce la posizione di un elemento se esso esiste.

La ricerca viene divisa quindi in due casi:

- ricerca **con successo**, che implica che stiamo cercando un elemento che **sicuramente** esiste;
- ricerca **senza successo** implica che non troviamo l'elemento che cerchiamo, quindi **sicuramente non** esiste.

Possiamo quindi immaginare una struttura dati che ci dice subito se un elemento esiste o meno, e una volta che sappiamo che esiste possiamo ricercare l'elemento nella struttura.

Possiamo implementare un dizionario nel seguente modo:

- **array ordinato**:
    - inserimento: $O(n)$, perché dobbiamo trovare lo spazio, e spostare gli elementi a destra;
    - cancellazione: $O(n)$, perché una volta eliminato l'elemento, tutti gli elementi vanno spostati da destra verso sinistra mantenendo l'ordine;
    - ricerca: $O(\log n)$ sfruttando la ricerca dicotomica;
- array non ordinato:
    - inserimento: $O(1)$;

- cancellazione: $O(1)$, perché non c'è bisogno di spostare tutti gli elementi a sinistra in quanto non ordinato;
- ricerca: $O(n)$, in quanto dobbiamo scorrere tutti gli elementi;

- lista ordinata:
  - inserimento: $O(n)$;
  - cancellazione: $O(1)$;
  - ricerca: $O(n)$, in quanto la ricerca binaria non è utilizzabile perché è impossibile accedere in tempo costate ad ogni elemento della lista;
- lista non ordinata:
  - inserimento: $O(1)$;
  - cancellazione: $O(1)$;
  - ricerca: $O(n)$;
- binary search tree bilanciato:
  - inserimento: $O(\log n)$;
  - cancellazione: $O(\log n)$;
  - ricerca: $O(\log n)$, grazie alle sue proprietà di bilanciamento.

La soluzione migliore per ora è l'albero binario di ricerca bilanciato. Ma noi possiamo anche usare una **tabella a indirizzamento diretto**. Sia U l'insieme di tutte le chiavi rappresentabili nel dizionario, e S le chiavi effettivamente presenti.

Allora le operazioni relative ad un elemento k nella tabella di indirizzamento diretto T basta:

```
insert(T, k):
    T[k] = 1

delete(T, k):
    T[k] = 0

search(T, k):
    if T[k] = 1 then
```

```
        return true
    return false
```

Nella tabella di indirizzamento diretto, quindi, **inserimento, cancellazione e ricerca sono operazioni a tempo costante**.

T di conseguenza ha cardinalità m dove m è l'elemento più grande di S. Tuttavia:

- se m è molto grande e S è **sparso**, allora T è uno **spreco di memoria**;
- l'insieme U può essere un insieme grande, quindi risulta impraticabile in quanto **manca lo spazio in memoria**.

Per risolvere, introduciamo una **black box** che ci aiuta a **indirizzare gli elementi nella tabella T**. Tale black box utilizza una funzione della **funzione di hashing**:

$$h : U \to \{\,0, 1, ..., n-1\,\}$$

Quindi le nostre funzioni diventano:

```
insert(T,k):
    T[h(k)] = 1

delete(T, k):
    T[h(k)] = 0

search(T, k):
    if T[h(k)] = 1 then
        return true
    return false
```

Tuttavia c'è un problema. Supponiamo di avere due chiavi $k_1, k_2 \in U$, tali che $h(k_1) = h(k_2)$. Tale fenomeno prende il nome di **collisione di due chiavi**, e non si può evitare. Questo è dovuto al fatto che la funzione di hashing va da un dominio grande ad un codominio piccolo.

Si possono risolvere le collisioni in due modi:

- risoluzione per **concatenazione**;
- risoluzione per **indirizzamento aperto**.

Risoluzione per concatenazione

Semplicemente se k_1, k_2 hanno la stessa immagine per h, basta concatenarli in una lista concatenata. Da un certo punto di vista complica molto le operazioni di inserimento: all'interno della tabella non abbiamo più procedure binarie.

```
insert(T,k):
    list-insert(T[h(k)], k)

delete(T, k):
    list-delete(T[h(k)], k)

search(T, k):
    return list-search(T[h(k)], k)
```

Le insert e il delete hanno entrambe complessità $O(1)$ (il delete solo se la lista è doppiamente concatenata). La ricerca, tuttavia, cambia.
Il problema di una ricerca all'interno di una lista prende un tempo proporzionale alla lunghezza della lista stessa. Il caso pessimo è che gli
$|S|$ elementi sono tutti indirizzati nella stessa posizione. Nel caso peggiore quindi la complessità è $O(n)$, praticamente una lista standard.

La tabella hash si comporta meglio nel caso medio. In particolare, una **tabella hash mantiene costante la ricerca nel caso medio**.

Hashing uniforme semplice

Per il caso medio è necessario che la funzione di hashing sia un **hashing uniforme semplice**.

> *Un hashing si dice **uniforme semplice** se soddisfa la seguente proprietà:*
>
> $$P_i\{\, h(x) = i \,\} = \frac{1}{m}$$
>
> *ossia la probabilità che x sia indirizzata ad una cella i è la stessa per ogni cella, con m celle. In altri termini:*

> $$\sum_{0 \leq i < m} P_i\{\, h(k) = i \,\} = 1$$

Introduciamo inoltre un fattore α noto come **fattore di carico**, uguale a:

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

Dove m è il numero di celle, n il numero di elementi. Quindi esprime in percentuale quanto è carica la tabella. Vorremmo che α sia più o meno uguale a 1, ossia:

- le posizioni **non sono sprecate**;
- non superi di troppo 1.

La dimensione media di una lista concatenata in una cella di T in media è uguale a α, poiché se $\frac{1}{m}$ è la probabilità che un elemento venga inserito nella cella i, allora per ogni cella i la sua lunghezza è la somma:

$$\sum_{k \in S} P(h(k) = i) = \frac{n}{m}$$

Quindi la ricerca nel caso medio è $O(\alpha)$.

Implementazioni di funzioni di hashing

I principali metodi di hashing sono due:

- metodo della **divisione**;
- metodo della **moltiplicazione**.

Metodo della divisione

Supponiamo che $k \in U$. Supponiamo sia rappresentabile come un numero intero (assunzione più che valida, dato che **anche i numeri reali sono rappresentati come interi con codifica diversa** nei pc). Allora possiamo dire che:

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

Ad esempio, se $m = 10$, basta che guardo l'ultima cifra e so la sua posizione. Tuttavia, m ha un valore tale che **numeri con cifra delle unità pari a 6 creano collisioni**. Quindi non è una funzione ottima.

Buona prassi, inoltre, è evitare di prendere $m = 2^p$. Se ciò accade, dividendo per 2^p spostiamo p bit a destra. I p bit spostati sono il resto della divisione, e l'hashing dipenderebbe dagli ultimi p bit dei valori.

Converrebbe prendere allora un **numero primo**.

Metodo della moltiplicazione

Consideriamo $0 < A < 1$ e imponiamo:

$$h(k) = (kA \mod 1) \cdot m$$

Dove $\mod 1$ naturalmente indica di prendere la parte decimale. Quindi:

$$0 \leq kA \mod 1 < 1 \implies 0 \leq m \cdot ka \mod 1 < m$$

Tuttavia, l'**insieme è continuo**, e a noi interessano i numeri interi. Quindi consideriamo il **floor**:

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

Tuttavia tale funzione può essere **dispendiosa**, ma possiamo migliorarla con le operazioni bitwise.

Sia w la **dimensione di una word**. Allora il numero A compreso nell'intervallo $(0, 1)$. Possiamo sicuramente rappresentare A come un **numero intero se spostiamo a sinistra i bit di A di w posti**, ossia calcoliamo:

$$S = A \cdot 2^w$$

S è un intero rappresentabile in w bit. Moltiplicare k per S porta ad un numero rappresentabile in $2 \cdot w$ bit. Per calcolare $kA \mod 1$ considero i bit appartenenti alla word di sinistra.
Se prendiamo
$m = 2^p$ il prodotto kA **spostare i bit a sinistra di p posti**.

Nel complesso, quindi, basta che consideriamo i primi p bit a sinistra della prima word a partire da destra del numero $S \cdot k$ quando considero $m = 2^p$.