Appunti Algoritmi e Strutture Dati

Luca Seggiani

23 Aprile 2024

1 Hashing

Abbiamo già avuto a che fare con algoritmi di ricerca: i nostri approcci finora si limitavano a scorrimenti completi della lista (ricerca lineare) o ad approcci divide et impera che si basavano sul preordinamento (ricerca binaria). Presentiamo adesso un metodo più (sotto determinate condizioni) efficiente per fare ricerche su array, non basato sui confronti.

Funzione di hashing

Il concetto alla base dell'hashing è la cosiddetta funzione di hashing. La funzione di hashing stabilisce una corrispondenza quanto più iniettiva e surgettiva fra il tipo di dati su cui vogliamo effettuare ricerche, e un intero i che sarà indice di un'array. Se ammettiamo di allocare un'array di dimensione k, il suo risultato sarà un intero modulo k, ovvero un valore compreso fra $0 < n < k - 1$. L'implementazione della funzione di hashing consiste spesso in qualche "riarrangiamento" del valore iniziale (il verbo *to hash* in inglese significa spezzettare, confondere...), previa elevamenti a potenze particolari o altre operazioni molto divergenti. Un'esempio di algoritmo di ricerca effettuata con questo metodo potrebbe essere:

```
bool hashSearch(T* A, T x) {
  int i = h(x);
  if(A[i] == x) return true;
  else return false;
}
```

La funzione ci restituisce effettivamente se l'elemento x è stato immagazzinato nell'array attraverso la funzione di hashing h(x).

Indirizzamento aperto

Un'approccio usato spesso per implementare la funzione di hashing è quella di definire una funzione modulo la dimensione dell'array k:

$$h(x) = f(x) \ \% \ k$$

Notiamo che questo significa rilasciare l'iniettività della funzione: potranno esistere più di valori in entrata che corrisponderanno alla stessa chiave generata da h(x). Cercare di immagazzinare entrambi questi valori risulterà in una cosiddetta collisione.

Gestione delle collisioni

Come abbiamo visto, la funzione h(x) non potrà mai essere veramente iniettiva: scelto un k necessariamente finito, se il nostro range di dati in entrata è illimitato (o comunque più grande di k) sarà impossibile stabilire una funzione di hashing inettiva. Si renderà necessario allora un meccanismo di difesa contro le eventuali (e inevitabili) collisioni. Un'approccio usato per l'eliminazione delle collisioni è quello della scansione lineare: se non si trova l'elemento al primo posto restituito dalla funzione di hashing, si scorre l'array progressivamente, finchè non si riesce a trovare l'elemento o si torna sull'elemento di partenza. Una possibile implementazione di questo algoritmo potrà essere:

```
bool hashSearch(T* A, int k, T x) {
  int i = h(x);
  for(int j = 0; j < k; j++) {
    int pos = (i + j) % k;
    if(A[pos] == -1) return false;
    if(A[pos] == x) return true;
  }
  return false;
}
```

A questo punto potremo inserire con:

```
int hashInsert(T* A, int n, T x) {
  int i = h(x);
  for(int j = 0; j < n; j++) {
    int pos = (i + j) % n;
    if(A[pos] == -1) {
      A[pos] = x;
      return 1;
    }
  }
  return 0;
}
```

Scelta del k

Poniamo di voler immagazinare un'insieme di dati di dimensione minima n e massima N. Possiamo fare alcune considerazioni sulla scelta del valore k, dimensione dell'array, più adeguato. Innanzitutto, sarà necessario che k sia maggiore di n: in caso contrario, non potremo immagazzinare l'interezza dei valori nemmeno nel caso migliore. Sarà poi conveniente scegliere un valore maggiore, tenendo comunque a mente che finchè k è minore di N avrò la sicurezza di avere collisioni. Dobbiamo però aspettarci che con numeri k simili

a N avremo altà probabilità di collisione, e quindi la formazione di agglomerati, con successive ricerche inefficienti. Conviene quindi scegliere k più piccoli possibile, ma che non sacrifichino troppo le prestazioni della ricerca.

Inserzione dopo cancellazioni

C'è un'ottimizzazione possibile nel caso si vogliano effettuare cancellazioni: invece di assegnare al valore cancellato la chiave standard -1, possiamo assegnarvi una chiave diversa, sia -2. Nel caso vada a ricercare lo stesso elemento, una volta trovata la chiave -1 potrò direttamente arrendermi alla mia ricerca, in quanto sarò sicuro che l'elemento non sarà contenuto nella tabella. Possiamo quindi distinguere fra chiave:

- -1: elemento vuoto;
- -2: elemento disponibile.

E' esattamente per questo motivo che avevamo inserito la clausola:

```
1 if(A[x] == -1)
```

nella nostra funzione d'inserzione.

Scansioni

Esistono scansioni più efficienti della semplice scansione lineare. Vediamo una lista (non comprensiva):

- **Scansione lineare**: $h(x) + j \ \% \ k$;
- **Scansione quadratica**: $h(x) + j^2 \ \% \ k$, più efficiente (riduce gli agglomerati), ma rende nessario controllare di visitare tutte le posizioni dell'array, in modo da non fallire anche in caso di array non pieno.

Tempo medio di ricerca per l'indirizzamento aperto

Il tempo medio di ricerca (ovvero il numero di confronti) dipende da:

- **Fattore di carico**: il rapporto $\alpha = \frac{n}{k}$, compreso fra 0 e 1, che rappresenta il numero medio di elementi per ogni posizione;
- **Legge di scansione**: la scansione usata (meglio quadratica o ancora più sofisticata);
- **Uniformità della funzione hash**: la sua attitudine a generare indici con uguale probabilità.

Problemi con l'indirizzamento aperto

L'indirizzamento aperto non è sempre ottimale: molti inserimenti e cancellazioni degradano col tempo l'efficienza delle ricerche. Si rende necessaria una "rististemazione" periodica dell'array.

Hash con metodo di concatenazione

Vediamo un modo alternativo di implementare hash ovviando al problema delle collisioni. Possiamo associare ad ogni posizione dell'array una lista: eventuali collisioni risulteranno semplicemente in un inserzione in fondo alla lista della posizione trovata. Questo ci permette di scegliere un valore k minore di n (bound minimo di oggetti con cui avrò che fare), e evita del tutto gli agglomerati. Chiaramente, l'inefficienza è data dallo scorrimento lineare che dobbiamo fare quando andiamo a cercare valori che sono finiti in posizioni in fondo alla lista.