Algoritmi e strutture dati

Roberto Cordone

A. A. 2015-16

Capitolo 2

Implementazione dei dizionari: tabelle hash

Nota: queste dispense sono un rapido riassunto delle lezioni svolte nel dicembre 2015 e gennaio 2016. In buona parte ripetono gli argomenti delle dispense del prof. Goldwurm, con alcune aggiunte, qualche piccola modifica nella presentazione, qualche esempio. Chi notasse errori, incoerenze, oscurità o avesse dubbi è invitato a contattarmi per segnalarli.

2.1 Una possibile applicazione: il gioco del tris

- Si vuole contare le posizioni possibili per il gioco del tris
- Idea di algoritmo:
  1. partire con la posizione iniziale (scacchiera 3×3 vuota),
  2. generare dalla posizione corrente tutte le posizioni ottenibili aggiungendo una pedina del tipo giusto (cerchio o croce);
  3. aggiungere le posizioni figlie a una struttura (coda delle posizioni)
  4. estrarne una come posizione corrente e tornare al punto 2, oppure terminare (se la coda è vuota)

  L'uso della coda garantisce una visita in ampiezza dell'albero delle posizioni.
- L'idea pare buona perché certamente produce tutte le posizioni raggiungibili e non torna mai indietro
- L'idea è cattiva perché ci sono doppioni (vedi Figura 2.1)

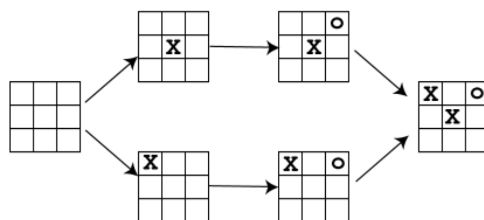

Figura 2.1: Nel gioco del tris, posizioni diverse possono dar luogo alla stessa posizione

- Idea di algoritmo modificato:
  1. partire con la posizione iniziale (scacchiera 3×3 vuota),
  2. generare dalla posizione corrente tutte le posizioni ottenibili aggiungendo una pedina del tipo giusto (cerchio o croce);
  3. per ogni posizione figlia
     (a) verificare se appartenga al dizionario delle posizioni già generate
     (b) se è nuova, aggiungerla alla coda delle posizioni da esplorare
  4. estrarre la posizione corrente dalla coda e tornare al punto 2, oppure terminare (se la coda è vuota)
- Il dizionario ha funzioni MEMBER e INSERT che consentono di eseguire questo algoritmo. Come si implementano in pratica?
- Implementazione banale: conserviamo una lista delle posizioni generate e usiamo le funzioni LIST_SEARCH e LIST_INSERT
- Problema: LIST_INSERT richiede tempo costante (inserimento in cima alla lista), ma LIST_SEARCH ha complessita' lineare nel numero di elementi della lista, per cui MEMBER sarebbe inefficiente

2.2 Tavola a indirizzamento diretto (look-up table)

- si definisce una funzione $k : A \to U$, dove $U = \{0, \ldots, |A| - 1\}$, che fa corrispondere ogni elemento di $a \in A$ a un numero naturale $k(a)$
- per brevità chiameremo *chiave* sia la funzione sia il suo valore
- ipotizziamo che $k(a)$ sia calcolabile in tempo costante: nel caso del tris, potrebbe esser la codifica della scacchiera per righe dall'alfabeto {' ','0','X'} come numero naturale in base 3, come nell'esempio della Figura 2.2 ($|A| = |U| = 19\,683$)

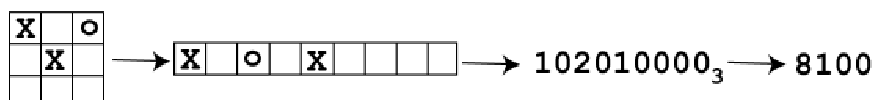

Figura 2.2: Codifica di una posizione del gioco del tris come numero intero

- Si definisce un vettore T di lunghezza $|A|$ che contiene in posizione k o l'elemento che ha chiave k oppure una marca a indicare che tale elemento non sta nel dizionario. Per esempio:

  ```
  element *T[|A|];
  ```

  dove `T[k]` punta l'elemento se questo fa parte del dizionario, ed è uguale a `NULL` se non ne fa parte.
- Si definisce la funzione costante $\emptyset$ come `for (i = 0; i < |A|; i++) T[i] = NULL;`.
- Si definisce `DIRECT_SEARCH(T,a)` come `return T[KEY(a)];`.

- Si definisce `DIRECT_INSERT(T,a)` come `T[KEY(a)] = &a;`[1].
- Si definisce `DIRECT_DELETE(T,a)` come `T[KEY(a)] = NULL;`.
- Vantaggi: tempo costante per tutte le operazioni
- Svantaggi: se la dimensione potenziale $|A|$ è grande
  1. occupazione in memoria enorme
  2. inizializzazione lenta

  anche per dizionari relativamente piccoli:
  - le parole italiane sono potenzialmente $26^2 1 = 2.4 \cdot 10^{34}$, ma in pratica circa 10^5;
  - le posizioni del tris sono potenzialmente 19 683, ma quelle realizzabili sono meno di 7 000, e molte meno se si considerano le simmetrie (almeno 2^4 volte meno).
- Nel caso di dizionari relativamente molto più piccoli del loro universo potenziale, basterebbe compattare i buchi, ma questo richiede una chiave più sofisticata, che (ammesso si riesca a trovarne una) potrebbe essere lenta da calcolare.

2.3 Tabella hash

- Si definisce una funzione *hash* $h : U \rightarrow\in \{0, \dots, m\}$, che ad ogni valore di chiave $k \in U$ fa corrispondere un valore $h(k) \in \{0, \dots, m\}$
- "Hash" viene da "ascia" perché tronca il valore della chiave.
- Si definisce un vettore H di lunghezza m. Per esempio:

  ```
  element *H[|A|];
  ```

  dove `H[k]` consente di accedere agli elementi con valore di hash h se fanno parte del dizionario, ed è uguale a `NULL` se nessun elemento del dizionario ha tale valore.
- Si fissa il valore m approssimativamente pari al numero di elementi del dizionario (spesso tale numero non è noto, ma solo stimato come ordine di grandezza).
- Vantaggi: l'uso della memoria e l'inizializzazione sono efficienti
- Svantaggi: può capitare che due chiavi abbiano uguale valore di hash (*collisione*).
- Il *paradosso del compleanno* ci dice che è anzi molto probabile.
- Non c'è modo sicuro per evitare le collisioni: occorrerebbe un hashing perfetto, che attribuisca valori tutti diversi agli elementi che vengono inseriti nel dizionario. Vediamo alcune soluzioni al problema di trattare le collisioni.

[1]Se si sta scrivendo `DIRECT_INSERT(T,a)` come funzione C, il passaggio per valore qui crea un problema: alla funzione bisogna passare l'indirizzo `&a`, che per chiarezza rappresenteremo come `pa`, e dentro la funzione si scriverà `T[KEY(*pa)] = pa;`

2.3.1 Chaining (liste di trabocco)

- Siccome gli elementi di H sono puntatori, li possiamo usare come teste di altrettante liste, i cui elementi contengono una posizione di gioco, la chiave (per poter fare una ricerca usando il valore della chiave anziché l'intera posizione) e un puntatore all'elemento successivo.

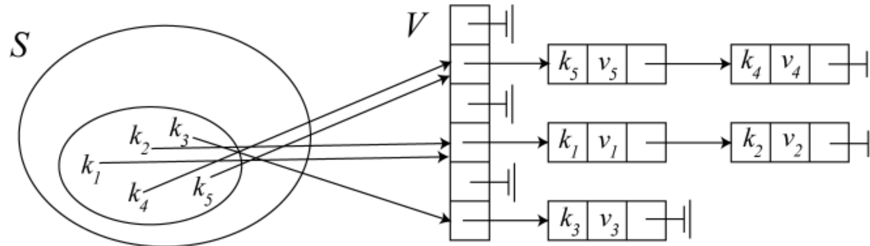

Figura 2.3: Gestione di una tabella hash con le liste di trabocco

- Si definisce `HASH_SEARCH(H,a)` come `return LIST_SEARCH(H,h(KEY(a)));`
- Si definisce `HASH_INSERT(H,a)` come `LIST_INSERT(H[h(KEY(a))],a);`
- Si definisce `HASH_DELETE(H,a)` come
```
pH = H[h(KEY(a))];
LIST_DELETE(pH,LIST_SEARCH(pH,KEY(a)));
```
- Esempio: si costruisca il dizionario dei quadrati perfetti fra 1 e 100 (esempio un po' assurdo: è abbastanza facile determinare se un numero è quadrato senza dover gestire una tabella di quadrati)
  - la chiave k di ogni elemento a coincide con il valore di a;
  - scegliamo $m = 11$, cioè un numero primo vicino al numero di elementi presunto per il dizionario (i quadrati perfetti fra 1 e 100 sono 10);
  - la funzione di hash sarà $h(k) = k \bmod m$;
  - la funzione di hash scelta non è molto efficace, dato che gli elementi si distribuiscono piuttosto disomogeneamente, su sole cinque caselle, tutte con liste di trabocco

h	0	1	2	3	4	5	6	7	8	9	10
L_h		100 1		36 25	81 4	49 16				64 9	

  Si osservi come, inserendo i numeri in ordine crescente, in ogni lista di trabocco il numero più alto sia in testa, dato che viene inserito per ultimo.

- La funzione `HASH_INSERT(H,a)` si esegue in tempo costante, inserendo ogni nuovo elemento in cima alla lista corrispondente.
- Le funzioni `HASH_SEARCH` e `HASH_DELETE` richiedono tempo lineare nella lunghezza della lista.
- Valutazione di complessità per `HASH_SEARCH`:
  - nel caso pessimo, tutti gli elementi finiscono nella stessa lista, e quindi la ricerca comporta tempo $O(n)$, dove n è il numero di elementi corrente del dizionario

- per valutare il caso medio, occorre un'ipotesi di distribuzione dei valori della funzione hash degli elementi del dizionario: l'ipotesi di *funzione di hash uniforme semplice* assume che ogni valore fra 0 e m sia equiprobabile ($\pi = 1/m$). È l'ipotesi più favorevole.
- la validità dell'ipotesi dipende dalla distribuzione dei valori delle chiavi, che ovviamente è diversa per ogni applicazione; nel caso del tris, le scacchiere con molti 0 e molte X sono vietate, per cui i corrispondenti numeri restano vuoti; se i loro valori di hash si concentrano in certe zone, le altre zone tendono invece a riempirsi.
- Definiamo *fattore di carico* della tabella hash il rapporto $\alpha = n/m$, dove n è la dimensione del dizionario e m quella del vettore usato per rappresentarlo.
- Il significato intuitivo di α è media aritmetica della lunghezza delle liste di trabocco:
$$\frac{\sum_{j=0}^{m-1} n_j}{m} = \frac{n}{m} = \alpha$$
dove n_j è la lunghezza della lista $H[j]$ ed è una variabile aleatoria.
- Se vale l'ipotesi di hash uniforme semplice, α è anche il valore atteso della lunghezza di ciascuna lista. Detta X_{ij} la variabile aleatoria binaria che indica se l'elemento i-esimo inserito nel dizionario ha hash j:
$$E[n_j] = E\left[\sum_{i=1}^{n} X_{ij}\right] = \sum_{i=1}^{n} \pi_j = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m} = \alpha$$
- Per un elemento non presente, la ricerca richiede nel caso medio tempo $O(1+\alpha)$ (1 per il calcolo della chiave e dell'hash, α per lo scorrimento di tutta la lista)
- Per un elemento presente, la ricerca richiede calcolo della chiave e dell'hash e scorrimento degli elementi che stanno prima nella lista, cioè son stati inseriti dopo. Per l'elemento i-esimo, contano solo gli elementi successivi $j = i+1, \ldots, n$:
$$E\left[\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{j=i+1}^{n} \frac{1}{m} = \frac{n-i}{m}$$
Quindi l'operazione è lenta per i primi elementi, veloce per gli ultimi. Mediamente
$$\frac{\sum_{i=1}^{n} \frac{n-i}{m}}{n} = \frac{n^2 - \sum_{i=1}^{n} i}{mn} = \frac{n^2 - n\frac{n+1}{2}}{mn} = \frac{n - \frac{n+1}{2}}{m} = \frac{n-1}{2m} = \frac{\alpha}{2} - \frac{1}{2m}$$
a cui vanno sommate la generazione della chiave e il confronto con l'elemento stesso.
- Come fare a generare una buona funzione di hash? L'idea naturale è $h(k) = k \bmod m$. Non va bene per valori di m prossimi a una potenza di un numero naturale, perché in tal caso l'hash diventa pari alle ultime cifre della codifica della chiave nella base m. Questo spesso equivale a guardare alcuni dati e ignorare gli altri, con la conseguenza che tutti gli elementi che contengono quei dati hanno chiavi identiche, dunque collisioni frequenti. Per esempio, nel caso del tris $m = 3^l$ fa collidere tutte le posizioni con le ultime l caselle identiche. I valori migliori sono numeri primi lontani dalle potenze dei numeri che svolgono un ruolo importante nella funzione chiave.

2.3.2 Probing (indirizzamento aperto)

- L'idea è registrare nella tabella tutti gli elementi
- Quindi lo spazio occupato in memoria è $O(m)$ e si rischia che possa non bastare.
- La funzione hash non indica un solo elemento, ma una permutazione delle celle della tabella: $h : U \times \{0, \dots, m\} \to \{0, \dots, m\}$

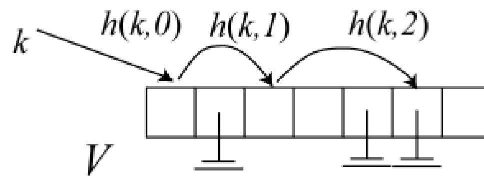

Figura 2.4: Gestione di una tabella hash con l'indirizzamento aperto

- La funzione HASH_SEARCH(H,a) percorre tutta la sequenza finché trova l'elemento o trova l'indicazione di fine della sequenza (una cella vuota);

```
k = KEY(a);
i = 0;
while ( (i < m) && (T[h(k,i)] != NULL) ) do
  if (KEY(T[h(k,i)]) = k) then
    return T[h(k,i)];
  i++;
return NULL ;
```

- La funzione HASH_DELETE(H,a) percorre tutta la sequenza finché trova l'elemento, poi marca la cella non come vuota (perderebbe il resto della sequenza!), ma come cancellata

```
k = KEY(a);
i = 0;
while ( (i < m) && (T[h(k,i)] != NULL) ) do
  if (KEY(T[h(k,i)]) = k) then
    T[h(k,i)]->deleted = TRUE; return;
  i++;
return NULL ;
```

- La funzione HASH_INSERT(H,a) percorre tutta la sequenza finché trova una cella vuota oppure cancellata in cui inserire l'elemento nuovo;

```
k = KEY(a);
i = 0;
while ( (i < m) && (T[h(k,i)] != NULL)
                && (!T[h(k,i)]->deleted) ) do
  i++;
if (i < m) then
  T[h(k,i)] = &a;
else
  exit(TABELLA_PIENA);
```

- Valutazione di complessità per `HASH_SEARCH`:
  - nel caso pessimo, è $O(n)$, esattamente come per le liste di trabocco
  - nel caso medio, dipende dalla distribuzione delle chiavi fra le sequenze di scansione
  - l'ipotesi di uniformità richiede che *ogni permutazione possibile abbia ugual probabilità* (ipotesi molto più forte che nel caso delle liste di trabocco)
- La funzione di hash più semplice è quella di *probing lineare*: si prende una funzione di hash base $h'(k)$ e si costruisce

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

  Questa funzione ha due grossi problemi:

  1. Siccome ci sono solo m possibili valori di hash h' per ogni k, questa famiglia contiene solo m funzioni di hash a indirizzamento aperto per ogni k, cioè m sequenze sul totale teorico di $m!$: l'ipotesi di uniformità è ben lontana
  2. se la funzione di partenza non rispetta l'uniformità, ma concentra i valori degli hash in certe zone, le sequenze prodotte finiscono subito per sovrapporsi
- Esempio: si costruisca il dizionario dei quadrati perfetti fra 1 e 100 con il probing lineare
  - la chiave k di ogni elemento a coincide con il valore di a;
  - scegliamo $m = 11$;
  - conserviamo la funzione di hash fondamentale $h'(k) = k \bmod m$
  - la funzione di hash sarà $h(k,i) = (h'(k) + i) \bmod m$;
  - gli inserimenti sono: 1 in 1, 4 in 4, 9 in 9, 16 in 5, 25 in 3, 36 in 3-4-5-6, 49 in 5-6-7, 64 in 9-10, 81 in 4-5-6-7-8 e 100 in 1-2, per un totale di 21 tentativi, col seguente risultato

h	0	1	2	3	4	5	6	7	8	9	10
L_h	0	1	100	25	4	16	36	49	81	9	64

  Si osservi come, inserendo i numeri in ordine crescente, i numeri bassi arrivano per primi e occupano il posto, mentre gli altri si adattano altrove. Inoltre, quasi subito si crea un blocco di caselle occupate che dà luogo a collisioni non solo per valori di uguale hash, ma anche per valori con hash simili.
- Esempio: per sperimentare con la `DELETE`:
  1. si cancelli 16 dalla casella 5;
  2. si cerchi 49 (per trovarlo bisogna non fermarsi in 16);
  3. si aggiunga 27 (viene inserito nella casella 5, disponibile anche se non vuota).
- la funzione di *probing quadratico* affronta il problema dell'agglomerazione, generando sequenze di valori non consecutivi

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

  dove c_1 e c_2 sono costanti opportune, che garantiscano insieme a m di scorrere l'intera tabella prima di tornare a un valore già visitato (per esempio, $c_1 = c_2 = 1/2$).
  Questa funzione:

1. risolve il problema di produrre lunghe sequenze di celle occupate
2. anch'essa costruisce sugli m possibili valori di hash h' per ogni k, e quindi produce solo sequenze sul totale teorico di $m!$

- Esempio: si costruisca il dizionario dei quadrati perfetti fra 1 e 100 con il probing lineare
  - la chiave k di ogni elemento a coincide con il valore di a;
  - scegliamo $m = 11$;
  - conserviamo la funzione di hash fondamentale $h'(k) = k \bmod m$
  - la funzione di hash sarà $h(k,i) = \left(h'(k) + i/2 + i^2/2\right) \bmod m$;
  - gli inserimenti sono: 1 in 1, 4 in 4, 9 in 9, 16 in 5, 25 in 3, 36 in 3-4-6, 49 in 5-6-8, 64 in 9-10, 81 in 4-5-7 e 100 in 1-2, per un totale di 18 tentativi, col seguente risultato

h	0	1	2	3	4	5	6	7	8	9	10
L_h	0	1	100	25	4	16	36	81	49	9	64

  I primi tentativi sono identici al caso di probing lineare (fino al secondo salto, con $i = 2$, non cambia nulla). Abbiamo risparmiato alcuni tentativi grazie al fatto di saltare più in lungo. Inoltre, il blocco di caselle occupate si crea più tardi. Comunque, la sequenza di salti è la stessa per tutte le chiavi: un passo avanti al primo tentativo, due passi al secondo, tre al terzo, ecc...

- per avvicinarsi alle $m!$ sequenze, si può usare il *doppio hashing*

$$h(k,i) = \left(h'_1(k) + ih'_2(k)\right) \mathrm{mod} m$$

  che usa due funzioni di hash base, avendo l'accortezza di sceglierle in modo che $h'_2(k)$ produca per ogni k valori primi rispetto a m. Questa combinazione produce per ogni k un insieme di $O\left(m^2\right)$ valori iniziali, e quindi sequenze diverse.

- Esempio di doppio hashing con
  - $h_1(k) = k \bmod m$,
  - $h_2(k) = 1 + k \bmod m'$, con $m' = 12$ (vicino a m, ma privo di fattori comuni)
  - $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$
  - gli inserimenti sono 1 in 1, 4 in 4, 9 in 9, 16 in 5, 25 in 3, poi le cose si complicano:[2] per un totale di X tentativi col seguente risultato

h	0	1	2	3	4	5	6	7	8	9	10
L_h											

  Si osservi come ogni chiave abbia la sua specifica sequenza di tentativi (con salti corti o lunghi, in avanti o all'indietro). Per alcune chiavi (36 e 100) la sequenza è la stessa, ma ha punti di partenza distinti.

[2] DA COMPLETARE: i miei appunti sono errati.

2.3.3 Gestione dinamica della tabella

- Che si fa se la tabella esplode, cioè non c'è spazio per nuovi valori? Si rialloca il vettore più largo e si reinseriscono gli elementi nella nuova tabella, ricalcolando le chiavi col nuovo m

- La strategia è di raddoppiare ogni volta la dimensione. Il procedimento di copia consiste ogni volta nel chiamare m volte INSERT. Se alla fine ci sono n elementi, il numero di riallocazioni r è determinato dal minimo valore $m_r \geq n$. Se si parte con una tabella da $m_0 = 1$ elemento (caso pessimo) e si fanno r riallocazioni, e il costo totale è

$$\sum_{s=1}^{r} m_s O(1) = \sum_{s=1}^{r} O(2^s) = O(2^r) = O(n)$$

  che è accettabile, essendo diviso su n inserimenti (i primi oggetti, che sono pochi, sono stati trasferiti parecchie volte, ma gli ultimi, che sono tanti, sono stati trasferiti poche volte: mediamente, ogni oggetto è stato trasferito un numero costante di volte).