Tabelle di hash

Anna Corazza

aa 2023/24

Dove studiare

- Sha'13, 9.4

Sha'13 Clifford A. Shaffer, Data Structures & Algorithm Analysis in C++, (edition 3.2), 2013
https://people.cs.vt.edu/shaffer/Book/C++3elatest.pdf

Problema da risolvere

Problema Ricerca di un oggetto in un insieme di oggetti.

Tabella di hash Gli oggetti vengono sistemati all'interno di una tabella **HT**.

Soluzione Accesso diretto basato su una **chiave**.

Prerequisiti Non si usa nessun ordine particolare per sistemare gli oggetti all'interno della tabella.

- In particolare, gli oggetti non sono ordinati.

Hashing La posizione nella tabella viene calcolata a partire dalla chiave.

Soluzione

- Data la **funzione hash h** che calcola la posizione a partire dalla chiave.
- Il numero di slots all'interno della tabella **HT** sia M (le posizioni sono $0, 1, \ldots, M-1$)
- Quindi in un sistema basato su hashing, occorre che:
  - ogni chiave K venga associata ad una posizione $i = h(K)$ dalla funzione di hash $h()$, con $0 \leq h(K) < M$;
  - il record che si trova nella posizione i della tabella abbia chiave K.

Non va bene se . . .

- Ovviamente non si tratta di una soluzione valida in tutti i contesti.
- In particolare, non va bene se:
  - sono ammessi più record con la stessa chiave;
  - se cerchiamo valori della chiave all'interno di un certo range anziché un valore specifico;
  - vogliamo trovare il record con chiave massima o minima, o visitare tutti i record per chiave crescente (o decrescente);

Si applica se . . .

- La richiesta da soddisfare è “restituire, se esiste, l’oggetto con chiave *K*”.
- In questo caso, con le **corrette scelte implementative**, si ottiene una ottima efficienza.
- Ma se le scelte sono sbagliate, si ottiene una soluzione molto inefficiente.
- Si adatta sia a ricerche all’interno della memoria che su disco.
- Insieme ai B-tree è la scelta implementativa più usata per grandi basi di dati.

Semplice esempio

Irrealistico

- Supponiamo di avere M record, tutti con chiave diversa nell'intervallo $[0, M-1]$.
- In questo caso, il record con chiave k verrà inserito nello slot k della tabella di hash HT,
- e la funzione di hash sarà $h(k) = k$.
- Ma non succede mai così! di solito i record hanno chiavi in un range molto più ampio del numero di slot della tabella di hash.

Esempio più realistico

- Supponiamo di avere valori della chiave in $[0, 2^{16} - 1 = 65535]$,
- e di aspettarci che HT contenga circa 1000 elementi per volta.
- Una tabella con 65536 slots? la maggior parte (circa 65536) degli slots resterebbe vuota: troppo spreco!
- Vogliamo utilizzare una tabella molto più piccola.
- Conclusione: il numero di slots è minore del numero di possibili valori della chiave.
- Quindi non possiamo avere uno slot per chiave, ma ad uno slot (almeno per alcuni degli slot) dovranno corrispondere più chiavi.

Collisioni

- Più formalmente: $\exists k_1 \neq k_2 : h(k_1) = h(k_2) = \beta$, dove β corrisponde ad uno slot della tabella.
- Quando questo succede, diciamo che k_1 e k_2 presentano una **collisione** sullo slot β sotto la funzione di hash $h()$.
- In conclusione, per trovare il record con chiave K in una base di dati organizzata col metodo di hashing servono due passi:
  1. Calcolo della funzione di hash $h(K)$ per identificare la posizione nella tabella in cui dovrebbe trovarsi il record che ci interessa.
  2. A partire dallo slot $h(K)$ cercare il record con chiave K applicando se necessario una **politica di risoluzione delle collisioni**.

Collisioni

continua

- Le collisioni sono praticamente inevitabili, a parte qualche caso particolare che vedremo dopo.
- Si cerca almeno di ridurne il numero.
- Qual è la probabilità che in una classe ci siano due studenti con lo stesso compleanno? con 23 studenti supera il 50% e aumenta all'aumentare del numero di studenti (i calcoli sono un po' complicati)
- Quindi se applicassimo il compleanno come hash function, avremmo una probabilità di collisione $> 50\%$ già con 23 chiavi e 365 slots, e la probabilità aumenta all'aumentare del numero delle chiavi.

Hashing perfetto

- In un sistema di hashing perfetto non ci sono collisioni.
- La funzione di hash **dipende** dallo specifico insieme di chiavi che vanno inserite nella tabella di hash, che quindi deve essere completamente disponibile **prima** che venga definita la funzione di hash.
- Efficienza ottima, perché mi serve esattamente un accesso per trovare una qualunque delle chiavi considerate.
- Ovviamente lo paghiamo col costo necessario per costruire la funzione di hash, ma può comunque essere accettabile se l'efficienza di retrieval del record è cruciale.
- Esempio: ricerca di dati in un CD disponibile in sola lettura.
- In questo caso la base di dati non cambia mai, ogni accesso è costoso e chi prepara la base di dati può contestualmente preparare la funzione di hash ottima.

In pratica

- La scelta del numero degli slot nella tabella di hash dipende ovviamente da quanto la tabella verrà riempita.
- Deve essere tale che:
  - Non venga sprecato troppo spazio: riempita a metà sembra ragionevole.
- In questa situazione le collisioni sono molto probabili ($p \approx 0.5$): vanno evitate il più possibile.
- Cruciale la differenza nelle prestazioni di un sistema di hashing tra buona e cattiva funzione di hash.
- **Indifferente**: va bene qualsiasi funzione che prende una chiave e restituisce uno slot.
- **Pessimo**: per qualsiasi chiave restituisce lo stesso slot: la tabella di hash non ci aiuta nella ricerca.
- **Ottimo**: ogni slot ha la stessa probabilità di venir colpito.

Distribuzione delle chiavi

- Per poter controllare la distribuzione degli slot, dovremmo conoscere la distribuzione delle chiavi.
- Di solito conosciamo il range di valori che le chiavi possono assumere.
- A volte la distribuzione delle chiavi all'interno del range ci è favorevole:

  Esempio distribuzione uniforme delle chiavi: basta dividere il range in parti uguali.
- Purtroppo la fortuna è rara, e in genere le chiavi risultano ammucchiate in alcune parti e rare altrove.
- In questi casi la scelta della funzione di hash è critica,
- ma dovrebbe partire dalla conoscenza della distribuzione delle chiavi.

Non è solo sfortuna

Buone ragioni per cattive distribuzioni

1. Distribuzione di Zipf (Es: città grosse e piccole).
2. La raccolta dei dati introduce bias (Es: arrotondamenti dei numeri).
3. La prima lettera delle parole inglesi (e italiane e . . .) non è uniforme.

Esempi di funzioni di hash

Semplice, ma non un granché

1. 16 slots:

```
int h(int x) {
  return x % 16;
}
```

- Lo slot dipende solo dai quattro bit meno significativi, che possono avere una cattiva distribuzione.
- Esempio di una scelta tipica: $\%M$: la scelta della dimensione della tabella di hash ha spesso un effetto importante sulle prestazioni.

Esempi di funzioni di hash

Molto meglio

2. Metodo **mid-square**:
   - 2.1 Fai il quadrato della chiave.
   - 2.2 Prendi gli r bit centrali.
   - 2.3 Ottieni uno slot in $[0, 2^r - 1]$

   Esempio
   - Chiavi: numeri decimali di 4 cifre.
   - M=100 (2 cifre decimali $\Rightarrow r = 2$)
   - Chiave: 4567, al quadrato: 208**57**489
   - Tutte le cifre contribuiscono a 57 (fa' i conti)

Esempi di funzioni di hash

Stringhe per chiavi

3. Hash function per una stringa di caratteri:

```
int h(char* x) {
  int i, sum;
  for (sum=0, i=0; x[i] != '\0'; i++)
    sum += (int) x[i];
  return sum % M;
}
```

- Stesso peso a tutti i caratteri
- OK, purché *M* non sia troppo grande: la somma deve essere molto più grande di *M*
- Esempio di **folding**: si può fare anche con gli interi.

Esempi di funzioni di hash

Più complicata, ma migliore

```
// Use folding on a string, summed 4 bytes at
    a time
int sfold(char* key) {
  unsigned int *lkey = (unsigned int *)key;
  int intlength = strlen(key)/4;
  unsigned int sum = 0;
  for(int i=0; i<intlength; i++)
     sum += lkey[i];
  // Now deal with the extra chars at the end
  int extra = strlen(key) - intlength*4;
  char temp[4];
  lkey = (unsigned int *)temp;
  lkey[0] = 0;
  for(int i=0; i<extra; i++)
     temp[i] = key[intlength*4+i];
  sum += lkey[0]; // overflow is not a problem
  return sum % M; // no negative numbers
}
```

Gestione delle collisioni

- In pratica, le collisioni non possono venir escluse a priori, quindi vanno gestite.
- Due classi principali, con hashing

  Aperto o con catene separate: le collisioni vengono immagazzinate **fuori** della tabella di hash.

  Chiuso : **dentro** la tabella di hash, in un altro slot

Hashing aperto

- Liste linkate per ogni slot con più records.
- Possono venir ordinate secondo l'ordine di inserimento, ordinate per chiave, o per frequenza di accesso.
- L'ordine per chiave evita di dover scorrere tutta la lista se la chiave non si trova.
- Questo tipo di hashing aperto è più adatto quando la tabella di hash viene tenuta in memoria.
- Su disco, gli elementi della lista possono venir memorizzati in blocchi diversi e quindi l'accesso diventa costoso.

Hashing chiuso

- **Tutti** gli elementi vengono memorizzati all'interno della tabella.
- La funzione di hash restituisce per ogni record R la sua posizione **home**.
- Se però tale posizione è già occupata da un altro record, deve essere possibile trovare un'altra posizione all'interno della tabella in cui sistemare R.
- Compito della **politica di risoluzione dei conflitti**, da cui dipendono le diverse strategie utilizzate.
- La stessa identica politica deve essere utilizzata sia per l'inserimento che per la ricerca.

Bucket hashing: inserimento

- Gli slot della tabella di hash sono raggruppati in buckets (secchi).
- M slots, B "secchi" $\Rightarrow M/B$ slots per ogni secchio.
- La funzione di hash assegna ogni record al primo slot di un bucket;
- se lo slot è pieno, il bucket viene scorso verso il basso fino a che non si trova uno slot libero;
- se si arriva in fondo al bucket (che quindi è completamente pieno) si passa ad un bucket di **overflow**, comune a tutti e di **capacità infinita**.
- Ovviamente questo caso andrebbe scongiurato il più possibile.

Bucket hashing: ricerca

- La funzione di ricerca indica il bucket in cui cercare il record.
- Se il record non si trova nel bucket e questo non è pieno, allora si può concludere che la hash table non contiene il record.
- Se invece il bucket è pieno, bisogna cercare il record nello slot di overflow.
- Se il bucket di overflow contiene molti record, questa ricerca può essere costosa.

Bucket hashing: esempio

- 10 slots divisi in 5 buckets di due slots ciascuno + bucket di overflow.
- Funzione di hash: $h(k) = k\%5$
- Inserisci questi sette valori: 9877, 2007, 1000, 9530, 3013, 9879, 1057.

Bucket hashing: variante

- La hash function non tiene conto della suddivisione in buckets.
- Se la posizione è già occupata, si comincia a scorrere verso il basso fino a raggiungere il fondo del bucket.
- Se tutte le posizioni fin qui sono occupate, si continua dalla cima del bucket fino ad arrivare sopra la posizione originaria.
- Solo a questo punto si passa al bucket di overflow.

Bucket hashing: variante

Esempio

- Come prima: 10 slots divisi in 5 buckets di due slots ciascuno + bucket di overflow.
- Funzione di hash: $h(k) = k\%10$
- Inserisci questi sette valori: 9877, 2007, 1000, 9530, 3013, 9879, 1057.

Bucket hashing

Discussione

- Funziona bene per hash table memorizzate su disco.
- Ad ogni accesso (sia per inserimento che per ricerca), si carica in memoria l'intero bucket.
- Quindi la ricerca all'interno di ogni bucket avviene in memoria (poco costosa).
- Se però il bucket è pieno e occorre passare al bucket di overflow, anche questo dovrà essere caricato in memoria, con un altro accesso a disco.
- Anche qui, è bene ridurre il più possibile il rischio di overflow.

Probing

- Nel bucket hashing la hash table viene divisa in diverse parti separate.
- Il record viene sistemato all'interno del bucket considerando le diverse posizioni in un certo ordine.
- Solo se questo non è possibile, si passa al bucket di overflow.
- L'idea del probing, invece, è che si parte sempre dalla home, ma poi viene generata una sequenza di posizioni in cui cercare la prima posizione vuota.
- Si tratta della forma di hashing più utilizzata, perché la politica di risoluzione delle collisioni può potenzialmente usare ogni slot nella tabella.

Sequenza di probe

- Quindi a partire da una chiave, viene generata non una sola posizione (funzione di hash), ma una sequenza di posizioni (sequenza di probe):
  - il primo elemento della sequenza è la home, data dalla funzione di hash;
  - se la home è occupata, si coinsidera la sequenza di probe: la prima posizione libera verrà occupata dal record da inserire;
  - la sequenza di probe viene generata dalla funzione di probe $p(k, i)$ restituisce lo slot in posizione i nella sequenza che si ottiene per la chiave k;
  - $p(\cdot, \cdot)$ genera un offset rispetto a home.

Probing: inserimento

```
// Insert e into hash table HT
template <typename Key, typename E>
void hashdict<Key, E>::
hashInsert(const Key& k, const E& e) {
  int home; // Home position for e
  int pos = home = h(k); // Init probe sequence
  for (int i=1; EMPTYKEY != (HT[pos]).key(); i++) {
    pos = (home + p(k, i)) % M; // probe
    Assert(k != (HT[pos]).key(), "Duplicates not
        allowed");
  }
  KVpair<Key,E> temp(k, e);
  HT[pos] = temp;
}
```

Probing: ricerca

```
// Search for the record with Key K
template <typename Key, typename E>
E hashdict<Key, E>::
hashSearch(const Key& k) const {
 int home; // Home position for k
 int pos = home = h(k); // Initial position is
     home slot
 for (int i = 1; (k != (HT[pos]).key()) &&
                 (EMPTYKEY != (HT[pos]).key());
                    i++)
   pos = (home + p(k, i)) % M; // Next on probe
       sequence
 if (k == (HT[pos]).key()) // Found it
   return (HT[pos]).value();
 else return NULL; // k not in hash table
}
```

Se la tabella è piena?

- Sia ricerca che inserimento funzionano solo se c'è almeno una casella vuota.
- Altrimenti ciclo infinito!
- Quindi occorre mantenere aggiornato il numero di record inseriti, in modo da fermarsi quando la tabella è piena.

Probing lineare

- È il più semplice a cui possiamo pensare:

$$p(k, i) = i$$

- Si scorre la tabella verso il basso fino a che non trovo una posizione libera o fino alla fine della tabella.
- Se sono in fondo, l'operazione di modulo mi riporta all'inzio della tabella.
- **Pro** Vengono considerate **tutti** gli slot prima di tornare alla posizione home.
- **Contra** tra tutte le funzioni di probe possibili, è una delle peggiori in quanto a probabilità di creare collisioni.

Linear probing

Problema

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

- Probing lineare: $h(k) = k\%10$.
- **Condizione ideale**: stessa probabilità di ricevere il prossimo record per tutti gli slot **vuoti**.
- Se la funzione di hash è stata ben progettata, darà ad ogni slot (**pieno o vuoto**) la stessa probabilità di venir riempito (0.1).

Linear probing

Problema – continua

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

- Supponiamo di dover inserire un record con chiave tale che la sua home è lo slot 0: esso finirà nello slot 2.
- Lo stesso se la home è in 1: anche questo finirà nello slot 2.
- Quindi lo slot 2 avrà probabilità 0.3 di venir riempito.

Linear probing

Problema – continua

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

- Analogamente, i record destinati agli slot 7 e 8 finiranno tutti in 9.
- Anche lo slot 9 ha probabilità 0.3 di venir riempito.
- Per gli slot 3,4,5 e 6, invece, la probabilità resta a 0.1.

Linear probing

Problema – continua

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

- Riassumendo, la probabilità di venir riempiti al prossimo passo vale 0.1 per gli slot 3,4,5 e 6, mentre vale 0.1 per gli slots 2 e 9.
- Noi invece vorremmo che tali probabilità fossero uguali!

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

0	9050
1	9050
2	
3	
4	
5	
6	
7	9877
8	2037
9	1059

- Peggio ancora, se il prossimo record va a riempire nello slot 9 (e questo ha una probabilità relativamente alta di succedere), nell'inserimento successivo la probabilità di riempire lo slot 2 sarà 0.6, mentre per gli slot 3,4,5 e 6 la probabilità resta a 0.1.

Clustering primario

- Questo tipo di comportamento viene chiamato **primary clustering**.
- Tendono a crearsi dei clusters di slot pieni, che a loro volta andranno a fondersi in clusters più grossi.
- Questo ha l'effetto di far allontanare la distribuzione di probabilità dalla distribuzione uniforme.

Come evitare il clustering primario

Prima idea

- Gli slot da considerare non devono essere adiacenti:

$$p(k, i) = ci$$

  con $c > 1$.
- In questo modo, i-esimo slot nella sequenza di probe sarà

$$(h(k) + ic)\%M$$

- E quindi anche se le home sono adiacenti, il resto della sequenza non è lo stesso.
- Questo è un punto a favore, ma noi vorremmo che la sequenza di probe visitasse **tutti** gli slot della tabella di hash.

Come evitare il clustering primario

sequenza che visiti tutti gli slot

- Quindi, la sequenza di probe dovrebbe visitare **tutti** gli slot della tabella di hash.
- Dipende da c e da M:
  - $c = 1$: lo fa per M qualsiasi;
  - $c = 2$ e M pari, no: se la home è pari, visita tutti e soli i pari, se è dispari, tutti e soli i dispari: è come se avessi due sezioni distinte e incomunicanti.
  - Questo diventa un problema se una delle due sezioni si riempie troppo più dell'altra: il costo per la gestione di questa sezione supera i vantaggi di avere l'altra vuota.
- Per avere una sequenza di probe che visiti tutti gli slot occorre che M e c siano co-primi (cioè MDC = 1).
- Esempi: $M = 10$, $c = 1, 3, 7, 9$ oppure $M = 11$ e $c \in [10]$.

Come evitare il clustering primario

Il problema persiste

- Finché uso probing lineare con *c* costante il problema persiste: due chiavi diverse possono condividere la sequenza di probing da un certo punto in poi.
- Record con chiavi con valori nella probing list non escono dalla sezione di HT determinata dalla sequenza.
- Questo è un caso che vorremmo evitare: le sequenze di probe dovrebbero divergere.
- **Caso Ideale**: sequenza di probe data da una permutazione aleatoria (random) delle posizioni in HT.
- Se è aleatoria non è ripetibile (per la ricerca): non va bene.

Probing pseudo-aleatorio

1. Genero una permutazione aleatoria degli slot di HT e la salvo in Perm.
2. $p(k, i) = \mathrm{P}erm[i - 1]$ per tutte le chiavi.

Esempio:

- $M = 101$,
- Perm$= (5, 2, 32, \ldots)$,
- due chiavi k_1 e k_2, con $h(k_1) = 30$ e $h(k_2) = 35$
- $k_1 : 30, \mathbf{35}, 37, 62, \ldots$
- $k_2 : \mathbf{35}, 40, 37, 67, \ldots$

Probing quadratico

- $p(k, i) = c_1 i^2 + c_2 i + c_3$ dove bisogna assegnare valori adeguati alle tre costanti c_1, c_2, c_3.
- Con $c_1 = 1, c_2 = c_3 = 0$, ottengo $p(k, i) = i^2$: se la home è diversa, la sequenza di probe diverge: $h(k) + i^2 \% M$
- **Esempio:** $M = 101$**;** $h(k_1) = 30 : \mathbf{30}, 31, 34, 39, \ldots$
- $h(k_2) = 29 : 29, \mathbf{30}, 33, 38, \ldots$
- Problema: la sequenza di probe non include molti degli slot di HT, e vale in generale per il probing quadratico.

Probing quadratico

Esempio

- Con $p(k, i) = i^2$ i risultati sono particolarmente sfortunati, nel senso che per molti valori di M vengono considerati solo un piccolo numero di slots
- e basta riempire quelli perché dei records rischino di venir rifiutati.

```
> a=unique(array([i*i for i in range(10000)])%105)
[ 0,  1,  4,  9, 15, 16, 21, 25, 30, 36, 39, 46,
   49, 51, 60, 64, 70, 79, 81, 84, 85, 91, 99,
   100]
> a.shape
(24,)
```

Probing quadratico

Soluzione

- Bisogna trovare la giusta combinazione tra funzione di probe e dimensione della HT.
- Se M è primo e $p(k, i) = i^2$, almeno metà degli slot vengono presi in considerazione $\Rightarrow$ per garantire che ci sia almeno uno slot disponibile occorre garantire che almeno metà della HT sia vuota.
- Se esiste $s : M = 2^s$ e $p(k, i) = \frac{i^2+1}{2}$, allora tutti gli slot verranno visitati.

Clustering primario: soluzione

- Il problema del clustering primario è legato al fatto che sequenze di probe diverse condividono larghi tratti.
- Viene **risolto** sia dal probing pseudo-aleatorio che da quello quadratico.
- Tuttavia, se due chiavi hanno la stessa home (e ovviamente il caso si presenta), allora avranno esattamente la stessa sequenza di probe.
- Questo perché la sequenza di probe è costruita solo a partire da i e le relative posizioni dipendono poi sempre nello stesso modo dalla home.
- In altre parole, $p(\cdot, \cdot)$ ignora la chiave k.
- **Clustering secondario**: se la funzione di hash genera clusters in particolari posizioni assembrando le home, questo problema non viene risolto né dal probing aleatorio né da quello quadratico.

Clustering secondario

- Per risolvere il problema del clustering secondario bisogna costruire sequenze di probing che tengano conto di k.
- bf Doppio hashing: proobing lineare con la “costante” che dipende dalla chiave: $p(k, i) = i \cdot h_2(k)$
- Esempio: $M = 101$, tre chiavi k_1, k_2, k_3:
  - $h(k_1) = 30, h(k_2) = 28, h(k_3) = 30$
  - $h_2(k_1) = 2, h_2(k_2) = 5, h_2(k_3) = 5$
  - Sequenza di probe per k_1 : $30, 32, 34, 36, \ldots$
  - Sequenza di probe per k_2 : $28, 33, 38, 41, \ldots$
  - Sequenza di probe per k_3 : $30, 35, 40, 45, \ldots$
  - Naturalmente se k_4 : $h(k_4) = 28, h_2(k_4) = 2$ il problema resta (con k_2.
  - Questi casi si possono evitare combinando il doppio hashing con il probing pseudo-aleatorio o quadratico.

Doppio hashing

- Una buona implementazione dovrebbe garantire che tutte le costanti nella sequenza di probe siano co-prime rispetto a M.
- Esempio 1: M **primo** e $h_2(\cdot)$ con valori in $[1, M-1]$.
- Esempio 2: $M = 2^s$ e $h_2(\cdot)$ con valori **dispari** in $[1, 2^s]$.

Cancellazione

- Due requisiti:
  1. Cancellare un elemento dalla HT non deve ostacolare successive ricerche inserendo vuoti.
  2. Le posizioni svuotate devono poter essere riutilizzate.
- Soluzione: pietra tombale (tombstone)
- Nella ricerca dice che si deve proseguire.
- Nell'inserimento si può usare la posizione per l'inserimento, ma solo dopo aver controllato che la chiave non c'è nel seguito della sequenza (vanno evitate le chiavi duplicate).

Cancellazione via pietre tombali

- Svantaggio: allungano la distanza media di ogni record dalla sua home.
- Nel tempo si arriva ad una situazione di equilibrio dovuta all'alternarsi di cancellazioni e inserimenti.
- Due possibili soluzioni:
  - Riorganizzazione locale (scambiando record con pietre tombali, ma solo se la politica di gestione delle collisioni lo permette).
  - Rehashing periodica della tabella (permette anche di mettere i record con accessi più frequenti nella loro home).