Tabelle di Hash

Fulvio Corno, Matteo Sonza Reorda
Dip. Automatica e Informatica
Politecnico di Torino

ADT Dizionario

In molte applicazioni è necessario un ADT "Dizionario" che supporti le seguenti operazioni:

- INSERT: Inserisce un elemento nuovo, con un certo valore (unico) di un campo chiave
- SEARCH: Determina se un elemento con un certo valore della chiave esiste; se esiste, lo restituisce
- DELETE: Elimina l'elemento identificato dal campo chiave, se esiste.

Esempi

- Tabella dei simboli di un compilatore
  - Chiave = nome di un identificatore
  - Dati aggiuntivi = tipo, contesto, dichiarazione
- Cache di file o URL
  - Chiave = path
  - Dati aggiuntivi = attributi e contenuto

Array associativi

Una struttura a dizionario si potrebbe implementare facilmente disponendo di *array associativi*, ossia vettori indicizzabili per contenuto anziché per posizione.

Esempio (di fantasia):

- Simboli[“main”] = { prog.c, 100, void, {int, char **}}
- Line n = Simboli[“counter”].linenum

Obiettivi

Le tabelle di hash sono una tecnica implementativa per realizzare array associativi.

Si vuole ottenere una complessità nel caso più frequente O(1) per le 3 operazioni fondamentali, anche se nel caso peggiore è $\Theta(n)$.

Idea base

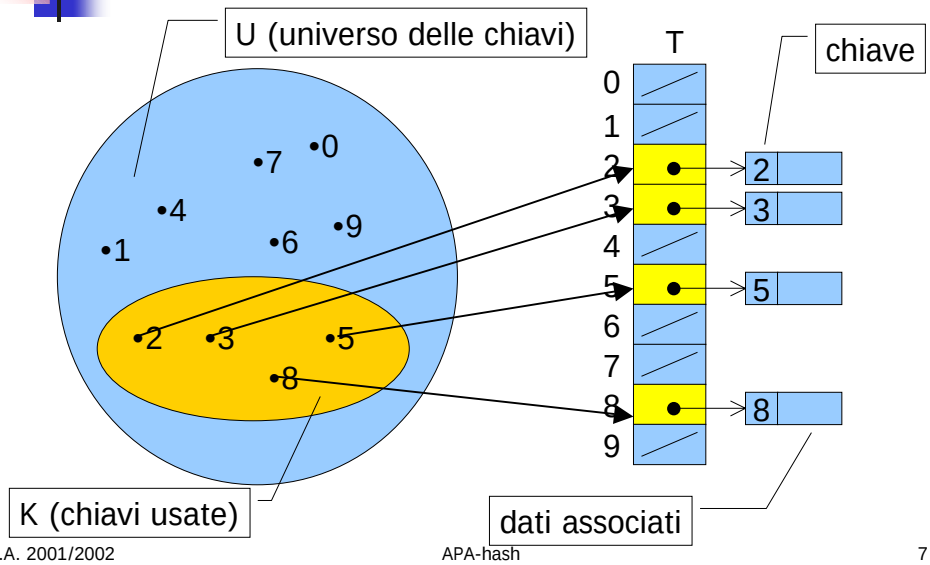
Ogni elemento è memorizzato ad un certo *indirizzo* di un array.

L'indirizzo, anziché venire calcolato da una funzione di ricerca, viene *calcolato* da un'opportuna funzione, detta funzione di hash, in tempo O(1).

Esempio:

- Hash("main") = 117: il simbolo "main" è memorizzato alla posizione 117 dell'array.

Tabelle associative



Dizionario mediante tabella associativa

- T: tabella associativa, k: chiave, x: elemento
- Search(T, k)
  - Return T[k]
- Insert(T, x)
  - T[key[x]] ← x
- Delete(T, x)
  - T[key[x]] ← NIL
- Complessità O(1), occupazione O(|U|)

Ipotesi

Lo schema precedente funziona solamente se sono verificate delle assunzioni fondamentali:

- Non esistono elementi con chiave uguale
- L'array T contiene tanti elementi quanti sono i possibili valori diversi delle chiavi.

Tabelle di Hash

Nella maggior parte dei casi, il numero di elementi |K| è molto minore del numero di valori possibili delle chiavi |U|.

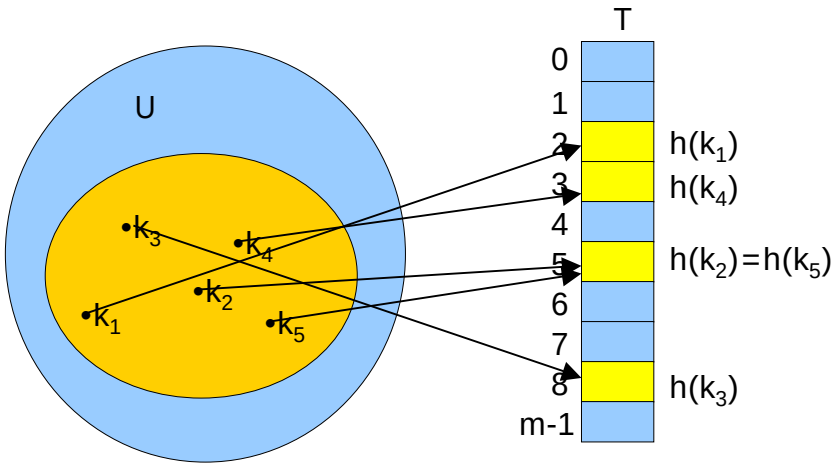
Quando l'universo delle chiavi è vasto (|U| cresce) non è possibile allocare il vettore T.

Una tabella di hash è una struttura dati con un'occupazione di spazio O(|K|) e tempi di accesso O(1), nel caso medio.

Funzione di hash

- La tabella di hash contiene m elementi (m<<|U|)
- Funzione che mappa una chiave k in una posizione del vettore h(k)
- h: U $\rightarrow$ { 0, 1, ..., m-1 }
- L'elemento x viene memorizzato nella locazione T[h(key[x])]

Funzione di hash



Collisione

- Ogniqualvolta $h(k_i)=h(k_j)$ quando $k_i \neq k_j$, si verifica una collisione
- Occorre:
  - Minimizzare il numero di collisioni (ottimizzando la funzione di hash)
  - Gestire le collisioni residue, quando avvengono (permettendo a più elementi di risiedere nella stessa locazione)

Ridurre le collisioni

Le funzioni di hash migliori sono quelle che distribuiscono il più uniformemente possibile i |K| elementi negli m indirizzi a disposizione.

La funzione h(k) deve sembrare il più “casuale” possibile. Solitamente di effettuano manipolazioni sui bit della chiave k, unitamente ad una scelta di un numero primo per il valore di m.

Gestire le collisioni residue

Solitamente si utilizzano due tecniche:

- Chaining
- Open Addressing

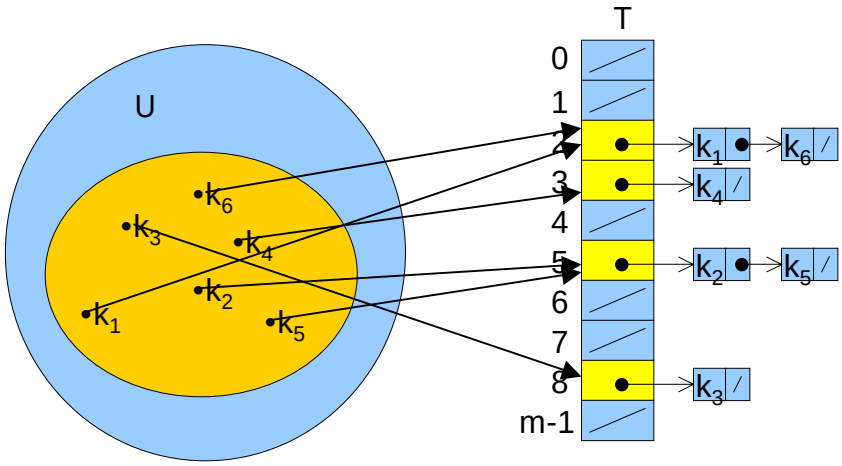
Chaining (I)

La soluzione più semplice per gestire le collisioni è permettere a più elementi di risiedere nella stessa locazione della tabella T.

Ogni locazione di T è quindi un *insieme* di elementi, e può essere implementata sotto forma di lista concatenata.

Tale tecnica viene detta *chaining*.

Chaining (II)



Pseudo-codice

- T[i] sono puntatori a liste, inizializzati a NIL.
- CHAINED-HASH-INSERT(T,x)
  - inserisci x alla testa della lista T[h(key[x])]
- CHAINED-HASH-SEARCH(T,k)
  - cerca l'elemento con chiave k nella lista T[h(k)]
- CHAINED-HASH-DELETE(T,x)
  - cancella x dalla lista T[h(key[x])]

Esercizio proposto

Si definiscano in C le strutture dati ed i prototipi delle funzioni necessarie per memorizzare in una hash table con chaning degli elementi contenenti una stringa (campo chiave) e due interi (dati aggiuntivi).

Esercizio proposto

Si completi l'esercizio precedente implementando le funzioni di inserimento, ricerca e cancellazione.

Si assuma di disporre di un'opportuna funzione h(k), di cui occorre fornire il prototipo.

Complessità

- Ipotesi: liste non ordinate
- Inserimento: O(1)
- Ricerca: O(lunghezza delle liste)
- Cancellazione:
  - O(1) se ho il puntatore ad x e la lista è doppiamente linkata
  - Uguale alla ricerca se ho il valore di x, oppure il valore della chiave k, oppure la lista è semplicemente linkata

Complessità delle ricerche (I)

- Detti:
  - n il numero di elementi memorizzati
  - m la dimensione della tabella di hash
- Si definisce:
  - α=n/m: fattore di carico della tabella di hash T
  - Può essere α>1
- Che cosa succede quando m,n$\rightarrow\infty$ (a parità di α) ?

Complessità delle ricerche (II)

- Nel caso peggiore la ricerca richiede $\Theta(n)$, più il tempo per calcolare h(k): la tabella di hash degenera in una lista semplice non ordinata
- Il caso migliore dipende da quanto uniformemente h(k) distribuisce gli elementi. Assumiamo per ora che h(k) abbia egual probabilità di generare gli m valori di uscita: *hashing semplice uniforme*

Hashing semplice uniforme

Assumiamo di saper calcolare h(k) in O(1). La complessità per la ricerca dipende linearmente dalla lunghezza della lista T[h(k)].

Occorre valutare separatamente il caso di elemento trovato ed elemento non trovato.

Si può dimostrare che in entrambi i casi la complessità è $\Theta(1+\alpha)$.

Conclusione

Se:

- Il numero m di “slot” cresce proporzionalmente ad n (α costante)
- h(k) distribuisce uniformemente gli elementi

Allora:

- La funzione di ricerca in una tabella di hash con chaining è $\Theta(1+\alpha)=O(1)$.

Progettare le funzioni di hash

La scelta della funzione di hashing è cruciale per l’efficienza dell’intera struttura dati.

Si assume che le funzioni migliori siano quelle che realizzano un hashing uniforme: se i valori delle chiavi k sono equiprobabili, allora tutti i valori della funzione h(k) devono essere anch’essi equiprobabili.

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m}, \quad j = 0,1,\ldots,m-1$$

Criteri generali

- Poiché le chiavi k solitamente non sono equiprobabili, anzi spesso sono molto correlate (si pensi ai nomi di variabili), occorre:
  - Usare tutti i bit della chiave
  - "Amplificare" le differenze
- Si può sempre pensare che le chiavi siano rappresentate come numeri interi (illimitati)
  - Es: "abc" può essere interpretata come 'a'*256^2 + 'b'*256 + 'c'

Chiavi come numeri

Nel seguito si assume che k siano numeri interi, o siano ricondotti a numeri interi.

Nella pratica, lavorando con stringhe di una certa lunghezza non è pratico convertire in numeri interi, per cui si adotteranno delle varianti dei metodi esposti.

Hashing per divisione

- Interpretando k come un numero intero, si definisce:
  - h(k) = k mod m
- Dato un numero previsto di elementi n, per garantire una certa complessità occorre scegliere $m \geq \alpha n$.

Scelta di m

- Occorre evitare che m sia
  - una potenza di 2 (usa solo gli ultimi m bit di k)
  - una potenza di 10 (se k sono numeri decimali)
  - 2^p-1 (se si trattano stringhe, in quanto trasposizioni di caratteri generano collisioni)
  - ...
- Solitamente si sceglie per m un valore:
  - corrispondente ad un numero primo
  - non troppo vicino ad una potenza di 2

Esempio

- n = 2000 elementi previsti
- Vogliamo un numero di confronti medio pari a 3 nelle ricerche
- m = 701 è un numero primo vicino a 2000/3 ma distante dalle potenze di 2
- h(k) = k mod 701

Hashing per moltiplicazione

- Interpretando k come un numero intero, si definisce:
  - Una costante 0<A<1
  - Frac(x) = x – ⌊x⌋
  - h(k) = ⌊ m · frac(k · A) ⌋
- La moltiplicazione k·A “rimescola” i bit i k, la moltiplicazione per m espande l’intervallo [0,1] nell’intervallo [0,m]

Scelta di m e A

- Il valore di m non è affatto critico. Solitamente si sceglie una potenza di 2, in modo che moltiplicazione e parte intera si riducano ad estrarre una sotto-sequenza di bit
- La scelta ottima di A dipende dalle caratteristiche statistiche delle chiavi
- $A = (\sqrt{5} - 1) / 2 = 0.6180339887...$ è una "buona" scelta

Hashing universale

Tutte le funzioni di hashing sono suscettibili di qualche caso peggiore nella scelta "cattiva" delle chiavi.

Si può pensare di "randomizzare" la scelta della funzione h(k), per "proteggerla" contro i casi peggiori.

Ad ogni esecuzione del programma, si sceglie a caso una funzione di hash tra un insieme di funzioni predefinite. La probabilità del caso peggiore viene così notevolmente ridotta.

Considerazioni pratiche

- Quasi sempre le chiavi sono stringhe (trattarle come numeri interi è complesso)
- Gli operatori bit-a-bit del C sono molto efficienti
  - Gli shift << e >> possono spostare parti della chiave per rompere schemi ripetuti
  - L'or esclusivo ^ permette di combinare sottosequenze di bit senza il *mascheramento* di and (&) e or (|)
  - Si può sfruttare il parallelismo delle parole della CPU (16, 32 bit)

HashPJW

```
#define PRIME 211
int hashpjw(char *s)
{     char *p ;
      unsigned int h=0, g;
      for ( p=s; *p != '\0'; p++ ) {
            h = ( h << 4 ) + (*p) ;
            if ( g = h & 0xf0000000) {
                  h = h ^ ( g >> 24 ) ;
                  h = h ^ g ;
            }
      }
      return h % PRIME ;
}
```

Esercizio proposto

Si completi il programma implementando la funzione hashpjw e confrontandone sperimentalmente le prestazioni con altre funzioni di hash più semplici.

Si può calcolare la lunghezza minima/massima (non media!) delle liste, o la varianza della lunghezza, o il numero di liste vuote, il numero di confronti totale, ...

Analisi sperimentale

È stata condotta un'analisi sulle prestazioni di diverse funzioni di hash su diverse tipologie di dati di ingresso.

Per ciascuna, è stato misurato il rapporto tra il numero di confronti misurato ed il caso atteso per una funzione di hash totalmente uniforme.

La tabella di hash conteneva 211 elementi (numero primo).

Input utilizzati

1: i 50 identificatori e parole chiave più frequenti in un campione di programmi C
2: i 100 identificatori e parole chiave più frequenti in un campione di programmi C
3: i 500 identificatori e parole chiave più frequenti in un campione di programmi C
4: 952 nomi 'extern' nel kernel di Unix
5: 627 identificatori in un programma C generato dal compilatore C++
6: 915 stringhe generate casualmente
7: 614 parole tratte da un testo di informatica
8: 1201 parole inglesi, con "xxx" aggiunto come prefisso e suffisso
9: i 300 nomi: "v100", "v101", ..., "v399"

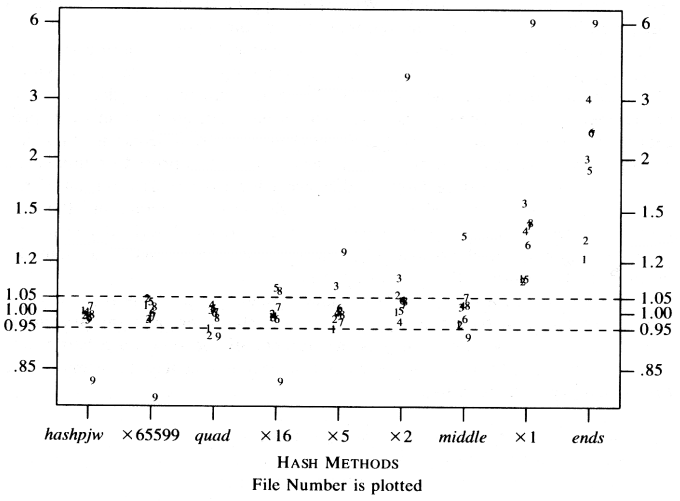
Funzioni di hash

- hashpjw
- $\times\beta$, con β=65599, 16, 5, 2, 1
  - $h(k) = \Sigma\, k[i] \times \beta^i$
- middle: considera i 4 caratteri centrali
- ends: considera i primi 3 e gli ultimi 3 caratteri
- quad: raggruppa i caratteri 4 a 4 e somma gli interi corrispondenti

Quantità misurate

- Il numero di confronti attesi per una lista di lunghezza b_j è $b_j(b_j+1)/2$.
- Il numero totale è ottenuto sommando il contributo delle m liste: $\Sigma_{j=0..m-1}\ b_j(b_j+1)/2$
- Il caso migliore è dato da $(n/2m)(n+2m-1)$
- Viene calcolato il rapporto
  - $\Sigma_{j=0..m-1}\ b_j(b_j+1)/2 \div (n/2m)(n+2m-1)$

Risultati



Open Addressing

La tecnica nota come Open Addressing è un'alternativa al Chaining per gestire le collisioni.

Ogni cella di T può contenere un solo elemento, e non è necessario gestire le liste di collisione.

In caso di collisione si ricerca un'altra cella non ancora occupata.

Funziona solo con $\alpha<1$.

Definizione formale

La funzione di hash deve generare una *permutazione* delle celle, che verrà interpretata come un *ordine di ricerca* della cella libera.

- $h : U \times \{ 0,1,...,m-1 \} \rightarrow \{ 0,1,...,m-1 \}$
- h(k,i) al variare di i deve essere una permutazione degli elementi { 0,1,...,m-1 }

Si tenta prima h(k,0), poi h(k,1), ... infine h(m-1).

Hash-Insert

```
HASH-INSERT(T, k)
1   i ← 0
2   repeat j ← h(k, i)
3       if T[j] = NIL
4           then T[j] ← k
5                return
6           else i ← i + 1
7   until i = m
8   error "hash table overflow"
```


Hash-Search

```
HASH-SEARCH(T, k)
1   i ← 0
2   repeat j ← h(k, i)
3       if T[j] = k
4           then return j
5       i ← i + 1
6   until T[j] = NIL or i = m
7   return NIL
```

Cancellazione

La cancellazione è un'operazione complessa, in quanto "rompe" le catene di collisione.

L'open addressing è in pratica utilizzato solo quando non si deve mai cancellare.

Funzioni di hash

Linear probing

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

Quadratic probing

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

Double hashing

- $h(k, i) = (h_1(k)+ i\, h_2(k)) \bmod m$

Complessità

Nel caso di hashing uniforme e di probing uniforme, si può dimostrare che:

- Il numero atteso di tentativi di "probing" è $1/(1-\alpha)$, ed è uguale alla complessità per l'inserimento
- La complessità della ricerca è invece

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}+\frac{1}{\alpha}$$

Esercizio proposto

Si implementi in C una tabella di hash con probing lineare, realizzando le funzioni di inserimento e di ricerca.