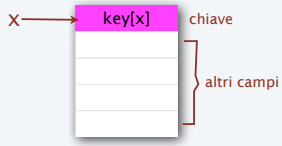
DIZIONARI

- Alberi Binari di Ricerca
- Tabelle Hash

Struttura dati Dizionario



Dizionario -

- struttura dati dinamica
- ogni elemento ha una chiave associata
- ***operazioni di base***
  - Search(S, k) ritorna "un puntatore" all'elemento con chiave k o Nil
  - Insert(S,k) inserisce un nuovo elemento con chiave k nel dizionario S
  - Delete(S,x) elimina dal dizionario l'elemento puntato da x
- altre operazioni
  - Min(S) ritorna un puntatore all'oggetto con la chiave minima in S
  - Max(S) ritorna un puntatore all'oggetto con la chiave massima in S
  - Successore(S,x) ritorna il puntatore all'oggetto la cui chiave segue immediatamente quella dell'oggetto puntato da x
  - Predecessore(S,x) ritorna il puntatore all'oggetto la cui chiave precede immediatamente quella dell'oggetto puntato da x.

Implementazione di un Dizionario

Array ordinato.

- Search(S,k) --> O(log(|S|))
- Insert(S,k) --> O(|S|)
- Delete(S,i) --> O(|S|)
- Min(S), Max(S), Successore(S,x), Predecessore(S,x) --> O(1)

Implementazione di un Dizionario

Array ordinato.

- Search(S,k) --> O(log(|S|))
- Insert(S,k) --> O(|S|)
- Delete(S,i) --> O(|S|)
- Min(S), Max(S), Successore(S,x), Predecessore(S,x) --> O(1)

0	10
1	22
2	25
3	50
4	
5	

Implementazione di un Dizionario

Array ordinato.

- Search(S,k) --> O(log(|S|))
- Insert(S,k) --> O(|S|)
- Delete(S,i) --> O(|S|)
- Min(S), Max(S), Successore(S,x), Predecessore(S,x) --> O(1)

Array o Lista non ordinata.

- Search(S,k) --> O(|S|)
- Insert(S,k) --> O(1)
- Delete(S,i) --> O(1)
- Min(S), Max(S), Successore(S,x), Predecessore(S,x) --> O(|S|)

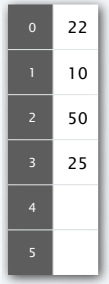
Implementazione di un Dizionario

Array ordinato.

- Search(S,k) --> O(log(|S|))
- Insert(S,k) --> O(|S|)
- Delete(S,i) --> O(|S|)
- Min(S), Max(S), Successore(S,x), Predecessore(S,x) --> O(1)

Array o Lista non ordinata.

- Search(S,k) --> O(|S|)
- Insert(S,k) --> O(1)
- Delete(S,i) --> O(|S|) [O(1) per le liste]
- Min(S), Max(S), Successore(S,x), Predecessore(S,x) --> O(|S|)



Implementazione di un Dizionario

Array ordinato.

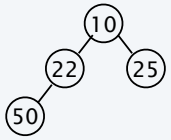
- Search(S,k) --> O(log(|S|))
- Insert(S,k) --> O(|S|)
- Delete(S,i) --> O(|S|)
- Min(S), Max(S), Successore(S,x), Predecessore(S,x) --> O(1)

Array o Lista non ordinata.

- Search(S,k) --> O(|S|)
- Insert(S,k) --> O(1)
- Delete(S,i) --> O(|S|) [O(1) per le liste]
- Min(S), Max(S), Successore(S,x), Predecessore(S,x) --> O(|S|)

Heap binario (min-heap).

- Search(S,k) --> O(|S|)
- Insert(S,k), Delete(S,i) --> O(log|S|)
- Min(S) --> O(1)
- Max(S), Successore(S,x), Predecessore(S,x) --> O(|S|)



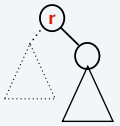
Albero Binario

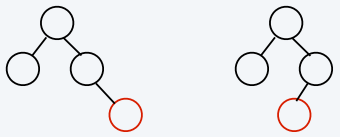
Definizione 1. Albero radicato, ogni nodo ha al più due figli.

- I figli di un nodo sono distinti: figlio destro e figlio sinistro.

Definizione 2. Un albero binario è una struttura su un insieme di nodi:

- vuota
- una tripla (r, T_s, T_d)
  - r è un nodo che prende il nome di radice
  - T_s, T_d sono alberi binari
  - se T_s (T_d) non è vuoto, allora la sua radice prende il nome di figlio sinistro (destro) di r



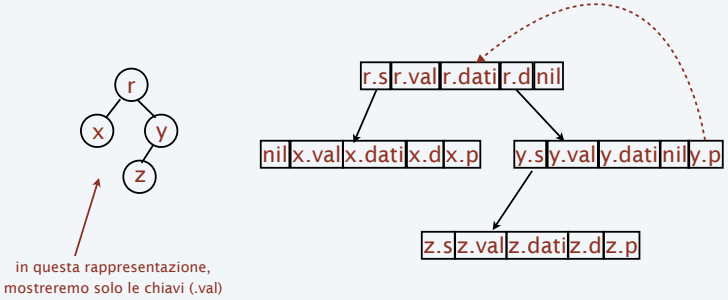


Questi sono due alberi Binari differenti

Albero Binario - rappresentazione

Rappresentazione mediante strutture e puntatori.

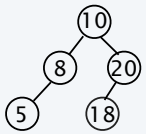
- ogni elemento x ha diversi campi
- x.val : valore o chiave di x, (indicato nel record x come come key[x])
- x.s, x.d : puntatori al figlio sinistro ed al figlio destro (nil se non esiste)
- x.p : puntatore al padre (nil se x è la radice)
- x.dati : può contenere altri dati o un puntatore ad altri dati



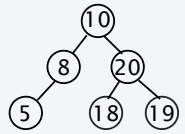
Albero Binario di Ricerca - *assumiamo valori differenti nei nodi*

Un albero binario di ricerca è un albero binario t.c.

- per ogni nodo u
  - ogni nodo x nel sottoalbero destro di u ha x.val > u.val
  - ogni nodo y nel sottoalbero sinistro di u ha y.val < u.val



E' un albero binario di ricerca



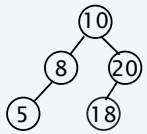
NON e' un albero binario di ricerca

Albero Binario di Ricerca - Search(S, k)

Ricerca di una chiave k

- Parti dalla radice r
  - se r = Nil ritorna NON-Trovata
  - se r.val = k, ritorna r
  - se k < r.val cerca ricorsivamente a partire da r.s
  - se k > r.val cerca ricorsivamente a partire da r.d

T(|S|) = O(altezza(S))




```
SEARCH(r, k) // S è indicato dalla radice r
IF (r = Nil)
    RETURN not found.
IF (r.val = k)
    RETURN r
IF (r.val > k)
    RETURN SEARCH(r.s, k)
IF (r.val < k)
    RETURN SEARCH(r.d, k)
```

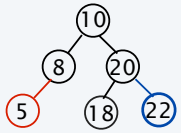
Albero Binario di Ricerca - Insert(r, k)

T(|S|) = O(altezza(S))

```
INSERT(r, k) // S è indicato dalla radice r
IF (r = Nil)
    r = Newnode(); r.val ← k
    RETURN
IF (r.val > k)
    IF (r.s ≠ Nil)
        INSERT(r.s, k)
    ELSE
        u = Newnode();
        r.s ← u; u.val ← k; u.p ← r
```

Inserisce un nuovo nodo con chiave k

- Parti dalla radice r
  - se r = Nil
    crea un nuovo nodo r con r.val = k e r.d = r.s = Nil
  - se k < r.val e r.s ≠ Nil ripete ricorsivamente a partire da r.s
  - se k < r.val e r.s = Nil
    crea un nuovo nodo u con u.val = k, u.p = r, r.s = u, u.s = u.d = Nil
  - se k > r.val e r.d ≠ Nil ripete ricorsivamente a partire da r.d
  - se k > r.val e r.d = Nil
    crea un nuovo nodo u con u.val = k, u.p = r, r.d = u, u.s = u.d = Nil



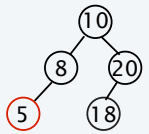
INSERT(*r*, 5) INSERT(*r*, 22)

Albero Binario di Ricerca - Min(S)

Ricerca del minimo in S

- Il minimo è l'ultimo elemento che si trova andando sempre a sinistra
  - se la radice è Nil, il dizionario è vuoto
  - altrimenti vado sempre a sinistra
  - mi fermo quando il figlio di sinistra è Nil

T(|S|) = O(altezza(S))



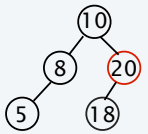

```
MIN (r) // S è indicato dalla radice r
IF (r = Nil)
    RETURN dizionario vuoto.
WHILE (r.s ≠ Nil)
    r ← r.s
RETURN r
```


Albero Binario di Ricerca - Max(S)

Ricerca del massimo in S

- Il massimo è l'ultimo elemento che si trova andando sempre a destra
  - se la radice è Nil, il dizionario è vuoto
  - altrimenti vado sempre a destra
  - mi fermo quando il figlio di destra è Nil

T(|S|) = O(altezza(S))



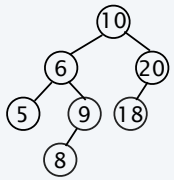

```
MAX (r) // S è indicato dalla radice r
IF (r = Nil)
    RETURN dizionario vuoto.
WHILE (r.d ≠ Nil)
    r ← r.d
RETURN r
```


Albero Binario di Ricerca - Successore(S,x)

Ricerca del successore di un elemento in S (puntato da x)

- Il successore di un elemento *x* è
  - il minimo nel sottoalbero di destra (se tale sottoalbero esiste)
  - il padre del più lontano dei suoi antenati (incluso se stesso) da cui *x* è raggiungibile andando sempre a destra

T(|S|) = O(altezza(S))



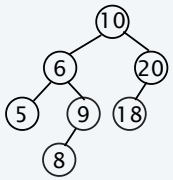

```
SUCCESSORE(S,x) // x punta all'elemento
IF (x.d ≠ Nil)
    RETURN MIN(x.d)
y ← x.p
WHILE (y ≠ Nil AND x = y.d )
    x ← y,  y ← x.p,
RETURN y // ritorna nil se x non ha successore
```


Albero Binario di Ricerca - Delete(S,x)

Cancellazione di un elemento

T(|S|) = O(altezza(S))

- Se l'elemento non ha entrambi i figli
  - lo eliminiamo e colleghiamo il figlio esistente (o nil) al padre
- Se l'elemento da cancellare ha entrambi i figli
  - lo sostituiamo con il suo successore
  - cancelliamo il successore (che non avrà figlio sinistro)




```
DELETE (S,x) // x punta all'elemento
IF (x.d ≠ Nil AND x.s ≠ Nil) // ha entrambi i figli
   y ← SUCCESSORE (S,x)
   x.val ← y.val)
   DELETE (S,y)
ELSE IF (x ha un figlio)
        y ← figlio non-Nil di x
        y.p ← x.p
     ELSE y ← Nil
     IF (x.p ≠ Nil AND x = (x.p).d )
        (x.p).d ← y
     IF (x.p ≠ Nil AND x = (x.p).s )
        (x.p).s ← y
```

Implementazione di un Dizionario

Albero Binario di Ricerca.

- Search(S,k) --> O(altezza(S))
- Insert(S,k) --> O(altezza(S))
- Delete(S,x) --> O(altezza(S))
- Min(S), Max(S) --> O(altezza(S))
- Successore(S,x), Predecessore(S,x) --> O(altezza(S))

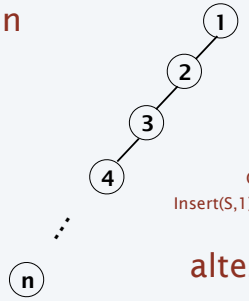
Tutte le operazioni in O(altezza(S))

Implementazione di un Dizionario

Albero Binario di Ricerca.

- Search(S,k) --> O(altezza(S))
- Insert(S,k) --> O(altezza(S))
- Delete(S,x) --> O(altezza(S))
- Min(S), Max(S) --> O(altezza(S))
- Successore(S,x), Predecessore(S,x) --> O(altezza(S))

Tutte le operazioni in O(altezza(S))



Con n inserimenti:
Insert(S,1), Insert(S, 2), ..., Insert(S,n)

altezza(S) = n-1

!?!?!

Allora...perchè studiare gli alberi binari di ricerca

Altezza media degli alberi binari di ricerca

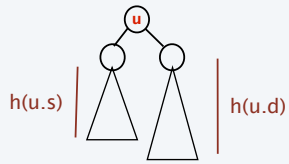
- L'altezza *media* di un albero binario di ricerca, costruito su *n* chiavi è O(log n)
- *altezza media*: calcolata assumendo che ogni ordine di inserimento (permutazione dell'input) abbia la stessa probabilità di essere quello in cui gli elementi sono effettivamenti inseriti.

Allora...perchè studiare gli alberi binari di ricerca

Possiamo anche garantire che l'altezza sia ***O(log n)***

Definizione ***1-bilanciamento***

- un albero binario si dice 1-bilanciato se per ogni nodo u, l'altezza dei sottoalberi radicati nei figli di u differisce di al più 1
- h(x): altezza del sottoalbero radicato in x



Albero 1-bilanciato sse per ogni u, vale | h(u.d) - h(u.s) | ≤ 1

Allora...perchè studiare gli alberi binari di ricerca

Possiamo anche garantire che l'altezza sia ***O(log n)***

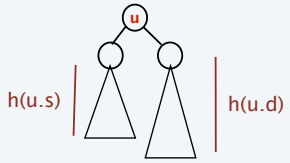
Definizione ***1-bilanciamento***

- un albero binario si dice 1-bilanciato se per ogni nodo u, l'altezza dei sottoalberi radicati nei figli di u differisce di al più 1

Lemma ***1-bilanciamento***

- un albero binario 1-bilanciato di altezza h ha un numero di nodi n che soddisfa la relazione $n \geq c^h$ (c è una costante)

Corollario. Per un albero 1-bilanciato con ***n*** nodi, vale ***h = O(n)***



Albero 1-bilanciato sse per ogni u, vale | h(u.d) - h(u.s) | ≤ 1

Come garantiamo che un albero binario sia 1-bilanciato?

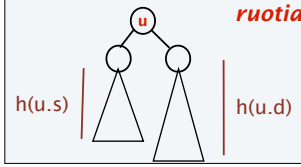
Le operazioni critiche sono: Inserimento e Cancellazione

Inserimento

- manteniamo in ogni nodo *u* l'altezza del sottoalbero radicato in *u*
  - per far questo, ad ogni inserimento di una nuova foglia f, l'altezza di f vale 0, poi risaliamo l'albero e aggiorniamo l'altezza di ogni antenato di f,

    altezza(x) = max{altezza(x.d), altezza(x.s)} + 1
  - in totale questi aggiornamenti fatti in corrispondenza di un inserimento costano in tempo: O(altezza(S))
  - se scopriamo che un qualche nodo u non rispetta più l'1-bilanciamento

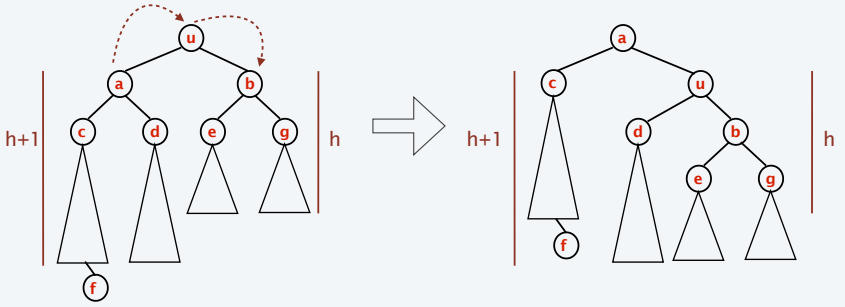
    ruotiamo opportunamente l'albero intorno a u (in tempo costante) e ristabiliamo l'1-bilanciamento



Esempi di rotazione - 1

La foglia aggiunta f è nel sottoalbero del figlio sinistro del figlio sinistro di u (il nodo per il quale l'1-bilanciamento risulta violato

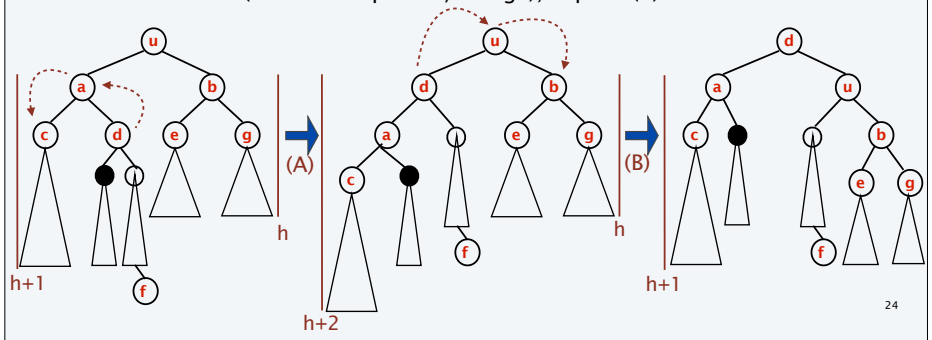
- facciamo una rotazione intorno a u
  - i sottoalberi vengono riorganizzati in modo da rispettare l'ordine tra le chiavi
  - per ottenere la rotazione vanno cambiati solo i puntatori dei nodi u, a, c, d, b (cambiano i padri e/o i figli) quindi al più 15 modifiche: O(1)



Esempi di rotazione - 2

La foglia aggiunta f è nel sottoalbero del figlio destro del figlio sinistro di u (il nodo per il quale l'1-bilanciamento risulta violato

- facciamo prima una rotazione (A) intorno ad a poi (B) intorno ad u
  - per ottenere la rotazione vanno cambiati solo i puntatori dei nodi interessati (cambiano i padri e/o i figli), al più O(1) modifiche



DIZIONARI

- Alberi Binari di Ricerca
- Tabelle Hash

Tabella ad accesso diretto

Memorizziamo l'intero universo delle chiavi.

- U = {0, 1, ..., m-1} è l'insieme delle chiavi possibili
- Costruiamo un array T[0...m-1] tale che

$$T[k] = \begin{cases} x & \text{se } x \in S \text{ e } key[x] = k \\ NIL & \text{altrimenti} \end{cases}$$

Quindi ogni operazione richiede tempo Θ(1).

- data la chiave k dobbiamo semplicemente accedere alla locazione T[k] per trovare il record corrispondente, se è stato inserito.

Problemi

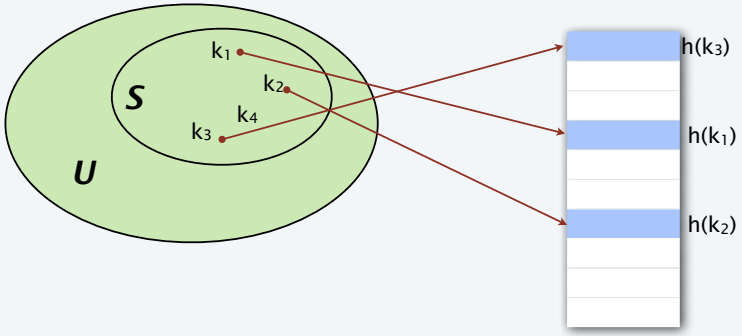
- il numero di possibili chiavi può essere enorme, mentre l'insieme S da rappresentare è piccolo
- vorremmo usare spazio proporzionale ad |S|, non ad |U|.

Funzioni hash e tavole hash

Funzione hash

- una funzione *h* che mappa l'universo U delle possibili chiavi in {0,1,...,m-1}

T : tavola/tabella hash

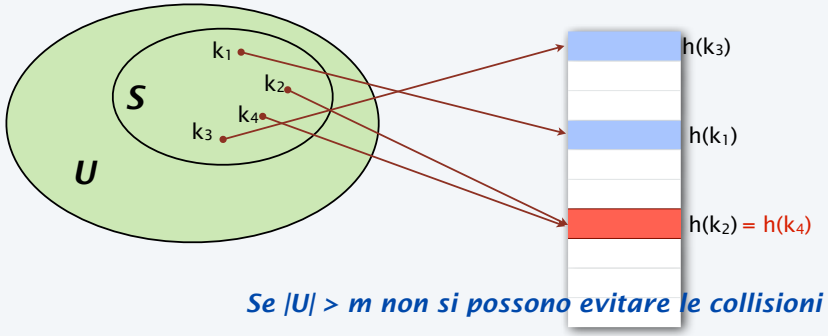


Funzioni hash e tavole hash

Funzione hash

- una funzione *h* che mappa l'universo U delle possibili chiavi in {0,1,...,m-1}

T : tavola/tabella hash



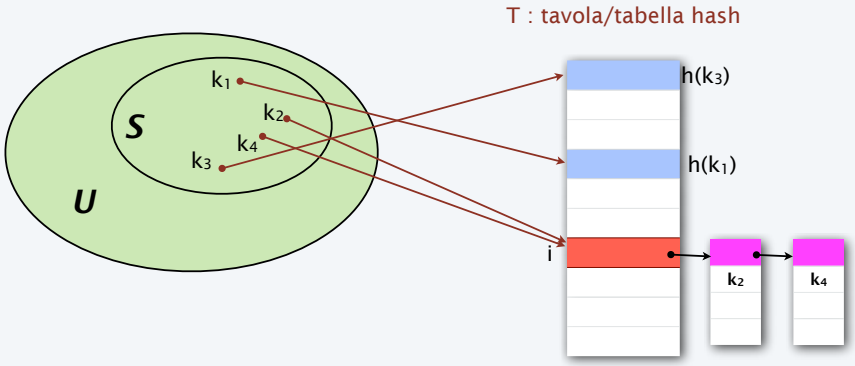
Se |U| > m non si possono evitare le collisioni

Collisione: più di una chiave viene mappata nello stesso slot della tavola hash

Risoluzione delle collisioni mediante liste di trabocco (chaining)

Chaining

- ogni slot della tabella hash rappresenta una lista di tutti gli elementi la cui chiave è mappata da h in quello slot



Tempo di accesso (caso pessimo)

- tutte le chiavi in S nello stesso slot

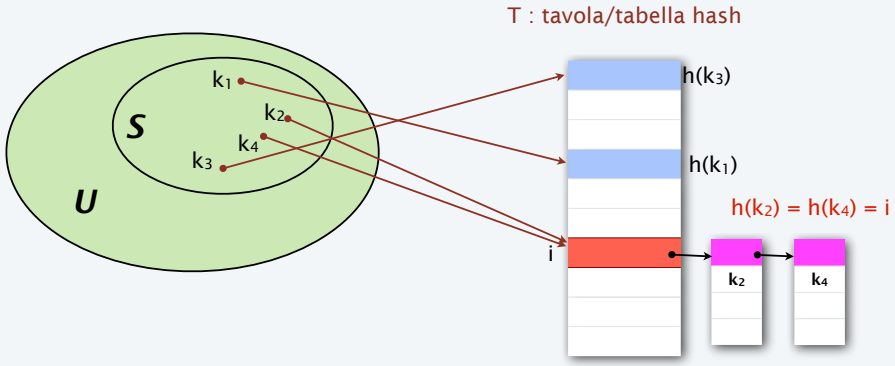
$h(k_2) = h(k_4) = i$

$\Theta(n)$ *dove* $n = |S|$

Risoluzione delle collisioni mediante liste di trabocco (chaining)

Chaining

- ogni slot della tabella hash rappresenta una lista di tutti gli elementi la cui chiave è mappata da h in quello slot



Tempo di accesso (caso pessimo)

- tutte le chiavi in S nello stesso slot

$\Theta(n)$ *dove* $n = |S|$

- Buttiamo via tutto?

NO! caso medio

$\Theta(1)$ *per* $n = |S| = O(m)$

Tabelle Hash con liste di trabocco - Analisi del Caso Medio

Hashing uniforme

- per ogni slot i la probabilità che una chiave k sia mappata in i è 1/m indipendentemente da dove vengono mappate le altre chiavi.
  - Prob[h(k) = i] = 1/m
  - Prob[$h(k_1) = h(k_2)$] = 1/m

Fattore di Carico

- n = numero di chiavi inserite in T (n = |S|)
- m = taglia di T
- α = n/m (fattore di carico)
  - α = lunghezza media di una lista di trabocco

Tabelle Hash con liste di trabocco - Analisi del Caso Medio

Hashing uniforme

- per ogni slot i e ogni chiave k la probabilità che k vada in i è 1/m indipendentemente da dove vengono mappate le altre chiavi.
  - Prob[h(k) = i] = 1/m
  - Prob[$h(k_1) = h(k_2)$] = 1/m

Fattore di Carico

- n = numero di chiavi inserite in T (n = |S|)
- m = taglia di T
- α = n/m (fattore di carico)
  - α = lunghezza media di una lista di trabocco

Ricerca di una chiave **non** presente in tabella

- $\Theta(1+\alpha)$

calcolo di h
accesso allo slot

ricerca all'interno
di una lista di
lunghezza α

Tempo atteso:

$\Theta(1)$ *per* $n = O(m)$

Tabelle Hash con liste di trabocco - Analisi del Caso Medio

Hashing uniforme

- per ogni slot i e ogni chiave k la probabilità che k vada in i è 1/m independentemente da dove vengono mappate le altre chiavi.
  - Prob[h(k) = i] = 1/m
  - Prob[$h(k_1) = h(k_2)$] = 1/m

Fattore di Carico

- n = numero di chiavi inserite in T (n = |S|)
- m = taglia di T
- $\alpha = n/m$ (fattore di carico)
  - α = lunghezza media di una lista di trabocco

Ricerca di una chiave *presente* in tabella

- $\Theta(1+f(\alpha))$

calcolo di h
accesso allo slot

dipende dalla posizione della chiave cercata nella lista. In media (si prova formalmente) $< \alpha/2$

Come Scegliere una buona funzione hash - euristiche

Caratteristiche desiderate per un buon hashing

- distribuire le chiavi uniformemente nei vari slot
- evitare che regolarità nelle chiavi di S inficino l'uniforme distribuzione

Euristiche

- metodo della divisione o del modulo
- metodo della moltiplicazione o della ruota modulare

Funzioni Hash - Divisione

h(k) = k mod m

Da evitare

- m con divisori piccoli
- $m = 2^r$ (m potenza di 2)
  - la funzione hash dipende solo dai primi r bit della chiave
  - se k = 10101110110110 e r = 7

h(k)

h(k) è indipendente dal valore di questi bit

Possibili scelte euristiche (che evitano evidenti problemi)

- m numero dispari senza divisori piccoli
- m numero primo
- m numero primo non vicino ad una potenza di 10 o di 2

Funzioni Hash - Moltiplicazione

$h(k) = (A \bullet k \bmod 2^w)\ rsh(w-r)$

$m = 2^r$ taglia della tabella una potenza di 2 ($r < w$)
w : lunghezza della parola macchina
A : numero dispari, $2^{w-1} < A < 2^w$ (non vicino agli estremi dell'intervallo)
rsh(i) : shift a destra di i posti

Punti di forza

- la moltiplicazione modulo 2^w è veloce (rispetto alla divisione)
- rsh è veloce

Funzioni Hash - Moltiplicazione - esempio

$h(k) = (A \cdot k \bmod 2^w) \text{ rsh}(w-r)$

$m = 8 = 2^3$ r = 3

w = 7

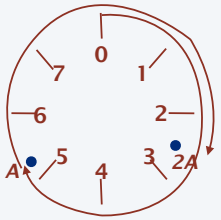
							1	0	1	0	1	0	1
							1	1	1	0	1	0	1
1	0	0	1	1	0	1	1	0	1	1	0	0	1

A

k

h(k)

Visualizziamo i tre bit come otto punti su questa ruota



Interpretiamo A come il numero razionale tra 0 e 8, la cui parte intera è data dai primi r bit e la parte frazionaria dai restanti w-r bit

$A = (101.0101)_2 = (5.3125)_{10}$

Moltiplicare A per k è come spostarsi di A per k volte e prendere la parte intera... un pò come girare una roulette.

Tabelle hash ad indirizzamento aperto

Tutte le chiavi sono nella tabella hash

- non abbiamo elementi a puntatori
- il massimo numero di chiavi è la massima taglia della tabella hash

Risoluzione delle collisioni

- inserimento di k'
- h(k') = i = h(k) [con k già inserita]
- cerchiamo un altro slot per k'
  - usiamo una diversa funzione hash per ogni nuovo tentativo

Tabelle hash ad indirizzamento aperto

- m funzioni hash, $h_1, h_2, \ldots, h_m$
- per ogni chiave k
  - (a) $h_1(k), h_2(k), \ldots, h_m(k)$ è una permutazione di {0, 1, ..., m-1}
  - (b) se $h_1(k)$ è occupato, provo $h_2(k)$ quindi $h_3(k)$ ....

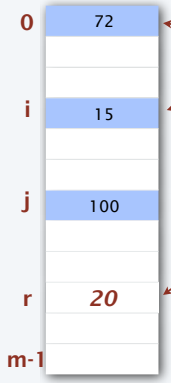
    (a) e (b) implicano che se esiste uno slot libero inserirò k

Risoluzione delle collisioni mediante indirizzamento aperto

Indirizzamento aperto (Open addressing)

- per ogni k, $<h_1(k), h_2(k) \ldots h_m(k)>$ è una permutazione di <0,1,...,m-1>

T : tavola/tabella hash



Insert(S,20)

$h_1(20) = i$ *collisione*

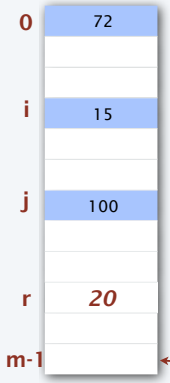
$h_2(20) = 0$ *collisione*

$h_3(20) = r$ *T[r] libero*

Indirizzamento aperto (Open addressing)

- per ogni k, $<h_1(k), h_2(k) \ldots h_m(k)>$ è una permutazione di <0,1,...,m-1>

T : tavola/tabella hash



Insert(S,20)

$h_1(20) = i$ *collisione*

$h_2(20) = 0$ *collisione*

$h_3(20) = r$ *T[r] libero*

Search(S,20)

$h_1(20) = i$ $T[i] \neq 20$

$h_2(20) = 0$ $T[i] \neq 20$

$h_3(20) = r$ $T[i] = 20$

20 è presente

Search(S,99)

$h_1(99) = j$ $T[j] \neq 99$

$h_2(99) = i$ $T[i] \neq 99$

$h_3(99) = m\text{-}1$ *T[m-1] libero*

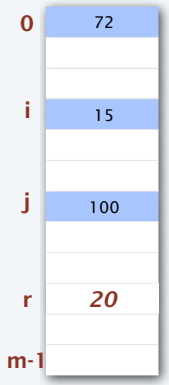
99 NON è presente

Risoluzione delle collisioni mediante indirizzamento aperto

Indirizzamento aperto (Open addressing)

- per ogni k, <$h_1(k)$, $h_2(k)$ ... $h_m(k)$> è una permutazione di <0,1,...,m-1>

T : tavola/tabella hash



Insert(S,20)

$h_1(20) = i$ *collisione*

$h_2(20) = 0$ *collisione*

$h_3(20) = r$ *T[r] libero*

```
INSERT (S,k)
FOR i = 1 to m
    IF ( T[h_i(k)] is empty )
        T[h_i(k)] ← k
        EXIT
RETURN Tabella piena
```

```
SEARCH (S,k)
FOR i = 1 to m
    IF ( T[h_i(k)] = k )
        RETURN i
    IF ( T[h_i(k)] is empty)
        RETURN non presente
RETURN non presente
```

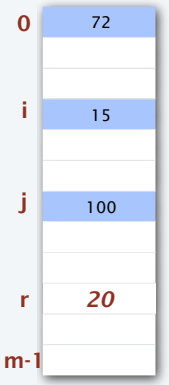
Risoluzione delle collisioni mediante indirizzamento aperto

Indirizzamento aperto (Open addressing)

- per ogni k, <$h_1(k)$, $h_2(k)$ ... $h_m(k)$> è una permutazione di <0,1,...,m-1>

T : tavola/tabella hash

Delete(S,72)



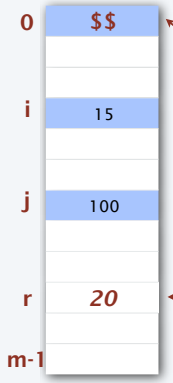
Risoluzione delle collisioni mediante indirizzamento aperto

Indirizzamento aperto (Open addressing)

- per ogni k, <$h_1(k)$, $h_2(k)$ ... $h_m(k)$> è una permutazione di <0,1,...,m-1>

T : tavola/tabella hash

Delete(S,72)



Search(S,20)

$h_1(20) = i$ *T[i] ≠ 20*

$h_2(20) = 0$ *T[0] libero*

20 NON presente

?!?!

Risoluzione delle collisioni mediante indirizzamento aperto

Indirizzamento aperto (Open addressing)

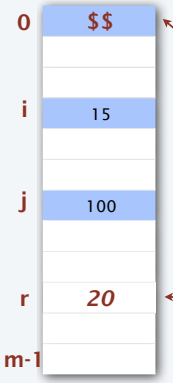
- per ogni k, <$h_1(k)$, $h_2(k)$ ... $h_m(k)$> è una permutazione di <0,1,...,m-1>

T : tavola/tabella hash

Delete(S,72)

Soluzione

Delete(S,k) non libera lo slot ma inserisce un marcatore



Search(S,20)

$h_1(20) = i$ *T[i] ≠ 20*

$h_2(20) = 0$ *T[0] ≠ 20*

$h_3(20) = r$ *T[r] = 20*

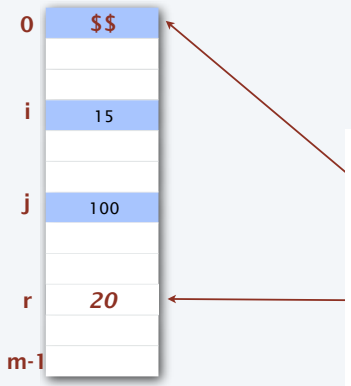
20 è presente

Risoluzione delle collisioni mediante indirizzamento aperto

Indirizzamento aperto (Open addressing)

- per ogni k, <$h_1(k)$, $h_2(k)$ ... $h_m(k)$> è una permutazione di <0,1,...,m-1>

T : tavola/tabella hash



Delete(S,72)

Soluzione

Delete(s,k) non libera lo slot ma inserisce un marcatore

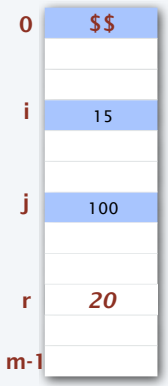
```
INSERT (S,k)
FOR i = 1 to m
    IF ( T[h_i(k)] is empty OR T[h_i(k)] = $$)
        T[h_i(k)] ← k
        EXIT
RETURN Tabella piena
```


Risoluzione delle collisioni mediante indirizzamento aperto

Indirizzamento aperto (Open addressing)

- per ogni k, <$h_1(k)$, $h_2(k)$ ... $h_m(k)$> è una permutazione di <0,1,...,m-1>

T : tavola/tabella hash



Ipotesi di Hashing Uniforme

ogni chiave k ha la stessa probabilità che <$h_1(k)$, $h_2(k)$, ..., $h_m(k)$> sia una qualsiasi delle m! permutazioni indipendentemente dalle permutazioni associate alle altre chiavi

Teorema

Sotto l'ipotesi di Hashing Uniforme, le operazioni di inserimento e ricerca in una tabella hash con indirizzamento aperto richiedono in media $\Theta(1)$

Risoluzione delle collisioni mediante indirizzamento aperto

Indirizzamento aperto (Open addressing)

- per ogni k, <$h_1(k)$, $h_2(k)$ ... $h_m(k)$> è una permutazione di <0,1,...,m-1>

Ipotesi di Hashing Uniforme

ogni chiave k ha la stessa probabilità che <$h_1(k)$, $h_2(k)$, ..., $h_m(k)$> sia una qualsiasi delle m! permutazioni indipendentemente dalle permutazioni associate alle altre chiavi

Teorema (ricerca di chiave assente)

Sotto l'ipotesi di Hashing Uniforme, la ricerca di una chiave assente in una tabella hash con fattore di carico $\alpha = n/m$, richiede in media $\Theta(1/(1-\alpha))$

Dim.: il numero di accessi diventa

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\left(1+\frac{1}{m-n+1}\right)\cdots\right)\right)\right)$$
$$\le 1+\alpha(1+\alpha(1+\alpha(\cdots(1+\alpha)\cdots)))$$
$$\le 1+\alpha+\alpha^2+\alpha^3+\cdots$$
$$=\sum_{i=0}^{\infty}\alpha^i = \frac{1}{1-\alpha}.$$

Tecniche di scansione - Euristiche

Indirizzamento aperto (Open addressing)

- per ogni k, <$h_1(k)$, $h_2(k)$ ... $h_m(k)$> è una permutazione di <0,1,...,m-1>

Scansione lineare

- Scelta una funzione hash $\hat{h}()$

$$h_i(k) = (\hat{h}(k) + i) \bmod m$$

se c'è una collisione proviamo con lo slot immediatamente successivo.

Hashing doppio

- Scelte due funzioni hash h() e g()

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

g(k) non dovrebbe avere divisori comuni con m.

Scelta possibile: $m = 2^r$ e g() una funzione a valori dispari.

Hashing - conclusioni

Q. Possiamo garantire il bound mostrato sul caso medio ?

A. No. In generale, per qualsiasi funzione hash scelta si può costruire una sequenza di chiavi che provoca molte collisioni.

A. Si può usare la randomizzazione.

- costruiamo in maniera random la funzione hash a tempo di esecuzione, estraendola da una famiglia di funzioni hash cosiddette universali.
- H è una famiglia universale se per ogni coppia di chiavi k_1, k_2, il numero di funzion i hash in H per le quali $h(k_1) = h(k_2)$ è $|H|/m$
  - per una funzione hash scelta a caso da H, la probabilità che k_1 e k_2 siano una collisione è $1/m$
  - questa è la probabilità di una collisione se $h(k_1)$ e $h(k_2)$ fossero scelte in maniera random uniformemente da {0,1, ..., m-1}
- Se usiamo una funzione hash estratta da una famiglia universale il numero di collisioni attese su una particolare chiave è $< n/m$ (= $\alpha < 1$).

Universal Hashing

La ricetta - (difficoltà: *facile-facile)*

- scegliamo m primo e non vicino ad una potenza di 2 o 10
- $b = \lfloor \log m \rfloor$ (numero di bit per scrivere m) −1
- $u = \lceil \log |U| \rceil$ (numero di bit per scrivere una chiave)
- scegli $r+1$ numeri a caso da {0,1,...,m} con $r = u/b$
  - $a_0, \ldots, a_r$
- poni $k_i = (k \bmod 2^{b(i+1)}) / 2^{ib}$ per $i = 0, \ldots, r$
- $h(k) = \sum_{i=0}^{r} a_i k_i \bmod m$

Esempio - m = 23, b = 4, |U| = 3 milioni, u = 22, r = 6

																	k_1					k_0	
k	0	0	1	0	0	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1	1	1	0	1
a's	0	0	1	1	1	1	1	0	0	1	0	1	1	0	1	1	1	0	1	0	1	1	1	1
																	a_1					a_0	

Esercizio: implementare universal hashing e provarlo

ponete m = 1087, ed inserite nella tabella 600 chiavi random tra 0 e 10^9 e contate il numero medio di collisioni