Note su tabelle hash, alberi binari di ricerca e altezza alberi AVL

Annalisa De Bonis

IASD 2014-15

Tabella Hash

- Modo di implementare il dizionario mediante un array

- Problemi dell'implementazione semplice con un array A:
  - Un'entrata con chiave k viene memorizzata nella cella A[k]
    - **Problema: L'array A deve avere dimensione pari alla chiave più grande del dizionario**

Tabella Hash

- Una tabella hash per un dato tipo di chiavi consiste di
  - Una funzione hash h
  - Un array (tabella) chiamato bucket array

Funzioni hash

- Una funzione hash h mappa un insieme chiavi in un intervallo prefissato di interi [0, m–1]
- h(x) è chiamato valore hash di x
- Lo scopo di una funzione hash è di distribuire le chiavi uniformemente nell'intervallo [0, m-1]

- Si verifica una collisione quando due chiavi del dizionario hanno lo stesso valore hash

Funzione Hash

- m = capacità bucket array

- Una buona funzione hash dovrebbe garantire che la probabilità che due chiavi vengano “mappate” nello stesso bucket è 1/m.
  - Questo e` lo stesso comportamento che si avrebbe nel caso in cui la funzione distribuisse con probabilita` uniforme gli elementi nelle celle dell’array.
  - NB: la funzione hash e` deterministica (associa ad una stessa chiave sempre lo stesso valore) ma si comporta come se avesse un comportamento casuale.

Funzione hash

- Per semplicita` assumiamo che le chiavi siano numeri naturali (interi non negativi)
- Nel caso in cui le chiavi non fossero numeri neturali, occorrerebbe applicare un metodo per trasformarli in numeri naturali
- Esempio: se la chiave e` una stringa, si possono sommare i valori ASCII dei caratteri della stringa

Funzione Hash

- Esempi di funzioni hash
  - Funzione modulo
    - hash(k)= k % m , le collisioni sono meno probabili se m è un primo
      - Genera molte collisioni se molte chiavi hanno un hash code della forma

        pm+q per diversi valori di p

    Es. Chiavi {200, 205, 210, 215, …600} e bucket size 100
  - Funzione iterativa
    - hash(k)= $k_0 \oplus k_1 \oplus \ldots \oplus k_{s-1}$, dove $k_0, k_1, \ldots, k_{s-1}$ sono ottenuti dividendo la rappresentazione binaria di k in s segmenti di uguale lunghezza con $0 \leq k_i \leq m-1$
      - Genera molte collisioni quando molte chiavi sono partizionate in sequenze di segmenti uguali a meno di una permutazione

Dizionari implementati con tabelle hash

- Schemi di risoluzione di una collisione:
  - Liste di trabocco (chaining): le entrate che hanno generato la collisione vengono memorizzate in una stessa lista
  - Indirizzamento aperto (open addressing): se un elemento genera una collisione con un elemento gia` presente nella tabella allora viene sistemato in un'altra cella della tabella

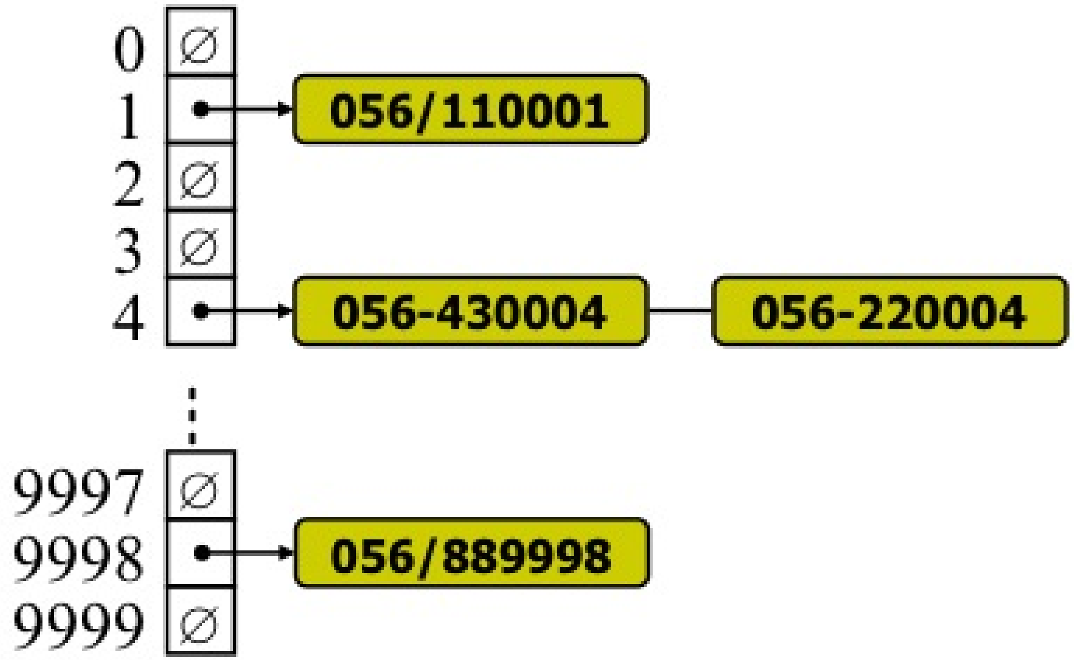
Liste di trabocco (chaining)

- Ciascuna entrata dell'array è l'indirizzo di una lista
- tabella[**h**] → S_h
  - S_h memorizza tutte le chiavi che hanno valore hash uguale ad **h**
  - S_h può essere visto come un piccolo dizionario implementato con una lista

Esempio

- tabella hash per un dizionario che contiene elementi della forma (matricola, nome studente), dove la matricola e` la chiave e il nome dello studente e` il dato satellite (per semplicita` nel disegno compaiono solo le chiavi)
  - Array di dimensione m=10000
  - Valore hash di h(x)= ultime 4 cifre di x
  - Risoluzione delle collisioni con chaining

La matricola 056-430004 e la matricola 056-220004 collidono



Codice per tabelle hash con liste di trabocco

```
Ricerca( k ):
  h = Hash(k);
  p = tabella[h].Ricerca( k );
  IF (p != null) RETURN p.dato ELSE RETURN null;

Inserisci( e ):
  IF (Ricerca( e.chiave ) == null) {
    h = Hash( e.chiave );
    tabella[h].Insfondo( e );
  }

Cancella( k ):
  IF (Ricerca( k ) != null) {
    h = Hash(k);
    tabella[h].Cancella( k );
  }
```

Fattore di caricamento

- m = dimensione bucket array
- n = numero entrate nella tabella
- Load factor: $\alpha = n/m$
- Se la funzione hash distribuisce con probabilita` uniforme gli elementi nelle m liste di trabocco, le liste di trabocco contengono un numero **medio** di elementi pari a $n/m = \alpha$
  - In questo caso il costo **medio** delle operazioni sulla tabella hash con liste di trabocco e` $O(1+\alpha)$
  - se $\alpha = O(1)$ allora il costo **medio** delle operazioni sulla tabella hash con liste di trabocco e` $O(1)$
- NB: Nel caso pessimo il costo delle operazioni e` $O(n)$

Indirizzamento aperto (open addressing)

- Il metodo dell'*open addressing* risolve le collisioni sistemando l'elemento che provoca una collisione in un'altra locazione dell'array
  - Di conseguenza, il fattore di caricamento α e` ≤ 1

- Questo metodo è utile quando non si ha molto spazio a disposizione
  - Non si utilizzano strutture dati ausiliare a differenza di quanto avviene nel chaining

Indirizzamento aperto (open addressing)

- Si usano m funzioni hash:
  - Hash[0], Hash[1],…,Hash[m-1]
  - Queste funzioni producono la cosiddetta sequenza di scansione (probing)
  - Ogni volta che si vuole inserire un elemento con una nuova chiave k nella tabella, vengono scandite le celle della tabella con indice Hash[0](k), Hash[1](k),…,Hash[m-1](k), in quest'ordine, fino a che non si arriva ad una cella di indice Hash[i](k) libera
  - Gli indici Hash[0](k), Hash[1](k),…,Hash[m-1](k) formano una permutazione degli indici dell'array 0,1,…,m-1

Indirizzamento aperto (open addressing)

- Ogni volta che si cerca una chiave k nella tabella, vengono scandite le celle della tabella con indice Hash[0](k), Hash[1](k),…,Hash[m-1](k), in quest'ordine, fino a che non si arriva ad una cella di indice Hash[i](k) che
  - contiene k oppure
  - e` libera (in questo caso l'algoritmo conclude che la chiave k non e` nella tabella)

Indirizzamento aperto (open addressing)

- Per cancellare una chiave k dalla tabella:
  - si effettua la ricerca della chiave k come descritto nella slide precedente
  - si sostituisce l'elemento con chiave k con un elemento marcatore che indica
    - all'algoritmo di inserimento che la cella puo` essere occupata dal nuovo elemento
    - all'algoritmo di ricerca che la cella in precedenza era occupata e che quindi non deve arrestare la ricerca.

Codice per tabelle hash con indirizzamento aperto

```
Ricerca( k ):
  FOR (i = 0; i < m; i = i+1) {
    h = Hash[i](k);
    IF (tabella[h] == null) RETURN -1;
    IF (tabella[h].chiave == k) RETURN tabella[h];
  }
```

```
Inserisci( e ):
  IF (Ricerca( e.chiave ) == null) {
    i = -1;
    DO {
      i = i+1;
      h = Hash[i]( e.chiave );
      IF (tabella[h] == null || tabella[h]==avail) tabella[h] = e;
    } WHILE (tabella[h] != e);
  }
```

```
Cancella( e ):
  k=e.chiave;
  FOR (i = 0; i < m; i = i+1) {
    h = Hash[i](k);
    IF (tabella[h] == null) RETURN;
    IF (tabella[h].chiave == k) {
      tabella[h]=avail;RETURN;
    }
  }
```

Indirizzamento aperto (open addressing)

- Se per ogni chiave k, la sequenza di scansione Hash[0](k), Hash[1](k),…,Hash[m-1](k) forma una delle m! permutazioni degli indici dell'array con probabilita` uniforme allora il costo medio delle operazioni sulla tabella e` $O(1/(1-\alpha))=O(1)$

Open Addressing con scansione lineare (linear probing)

- Il metodo del *linear probing* risolve le collisioni sistemando l'elemento che provoca una collisione nella prossima cella disponibile della tabella (vista come struttura circolare)

$$Hash[i](k)=(Hash(k)+i)\%\ m,\ i=0,\ldots,m-1$$

- Problema:
  - Primary clustering: le entrate tendono ad essere disposte in lunghi blocchi di celle consecutive aumentando così il numero di probe necessari per cercare un'entrata o inserire una nuova entrata

Metodi alternativi al linear probing

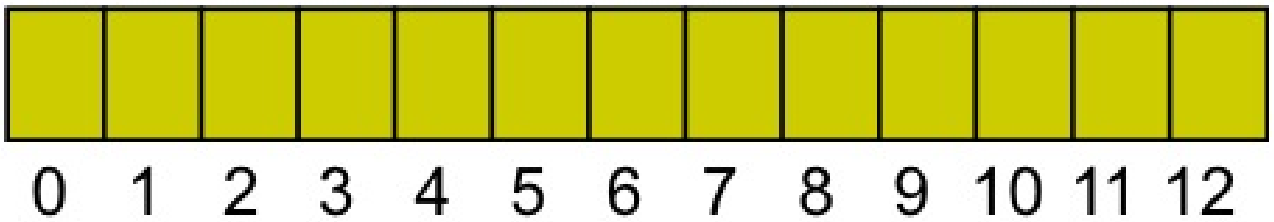
- *Scansione quadratica (quadratic probing):* $Hash[i](k)=(Hash(k)+ai^2+bi+c)\ \%\ m$, a, b e c costanti con $a \neq 0$
  - m, a, b e c devono essere scelti in modo appropriato altrimenti potrebbero esserci locazioni inutilizzate
  - Problema: come nel linear probing se $Hash(k_1) = Hash(k_2)$ allora $Hash[i](k_1)= Hash[i](k_2)$ per ogni i ➔ le sequenze dei probe effettuati per k_1 e per k_2 sono le stesse ➔ secondary clustering

Metodi alternativi al linear probing

- *double hashing (hash doppio):* Hash[i](k)=(Hash(k)+i(1+Hash'(k)))% m, dove Hash e Hash' devono essere scelte in modo che Hash ≠Hash' e che per ogni k vengano ottenuti tutti gli indici 0,1,…,m-1.

Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine



Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine

					18							
0	1	2	3	4	5	6	7	8	9	10	11	12

Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18							

Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine

		41			18				22			
0	1	2	3	4	5	6	7	8	9	10	11	12

Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine

		41			18	44			22			
0	1	2	3	4	5	6	7	8	9	10	11	12

Esempio

- $h(x) = x \% 13$
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine

		41			18	44	59		22			
0	1	2	3	4	5	6	7	8	9	10	11	12

Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine

		41			18	44	59	32	22			
0	1	2	3	4	5	6	7	8	9	10	11	12

Esempio

- $h(x) = x \% 13$
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine

		41			18	44	59	32	22	31		
0	1	2	3	4	5	6	7	8	9	10	11	12

Esempio

- h(x) = x %13
- Inseriamo le chiavi 18, 41, 22, 44, 59, 32, 31, 73 in questo ordine

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

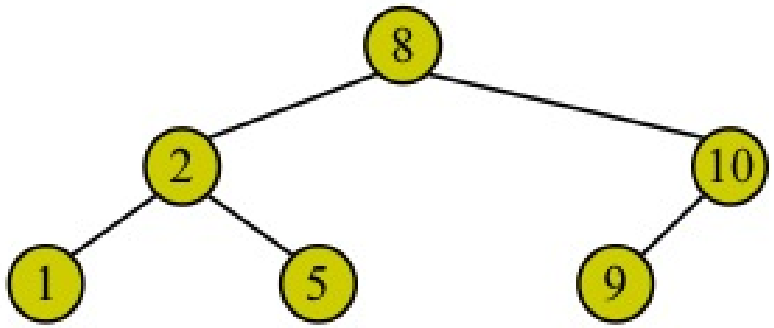
Alberi di ricerca binari

- Un albero di ricerca binario è un albero binario che memorizza in ciascun nodo una chiave in modo tale che
  - Se u, v e w sono tre nodi tali che u si trova nel sottoalbero sinistro di v e w si trova nel sottoalbero destro di v, allora

u.dato.chiave < v.dato.chiave < w.dato.chiave

(sulle chiavi è definita una relazione di ordine totale)

Esempio



- Per semplicita` vengono mostrate solo le chiavi degli elementi

Visita inorder di un albero binario di ricerca

- Una visita inorder di un albero di ricerca binario visita le chiavi in ordine crescente
  - Possiamo ottenere la sequenza ordinata della chiavi

Algoritmo di ricerca

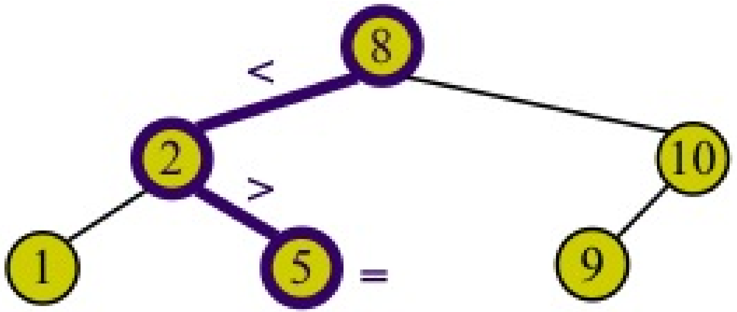
- Input: la chiave da cercare k e un nodo u dell'albero
- Output: l'elemento dell'albero con chiave k se k e` presente nel dizionario; null altrimenti.

Comportamento dell'algoritmo di ricerca

- Se la chiave k e` uguale a quella dell'elemento di u l'algoritmo restiuisce l'elemento presente in u
- Se la chiave k e` minore di quella dell'elemento di u la ricerca prosegue nel sottoalbero sinistroa di u
- Se la chiave k e` maggiore di quella dell'elemento di u la ricerca prosegue nel sottoalbero destro di u
- Se u e` null l'algoritmo resituisce null

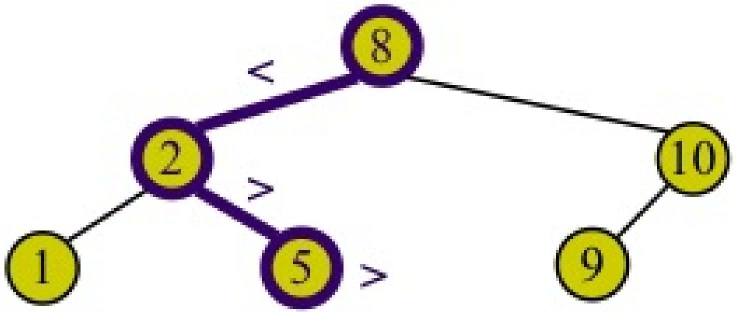
Esempio

- Ricerca(radice,5)



Esempio

- Ricerca(root,6)



Ricerca con una chiave k

Ricorsione con tre casi (come la ricerca binaria):

```
Ricerca( u, k ):
  IF (u == null) RETURN null;
  IF (k == u.dato.chiave) {
    RETURN u.dato;
  } ELSE IF (k < u.dato.chiave) {
    RETURN Ricerca( u.sx, k );
  } ELSE {
    RETURN Ricerca( u.dx, k );
  }
```

$O(h)$ tempo dove h = altezza dell'albero

Algoritmo di Inserimento

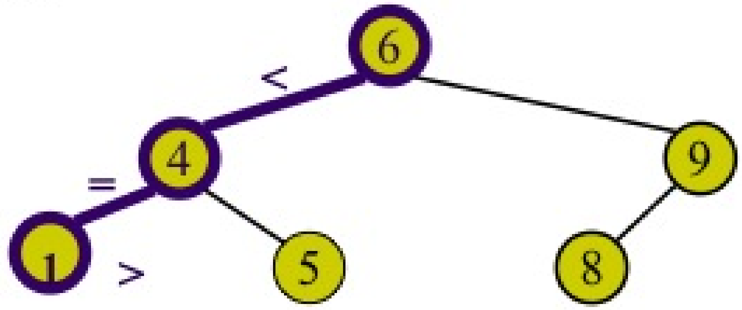
- Input: un elemento e e un nodo u
- Se la chiave di e non e` presente nell'albero inserisce e nell'albero

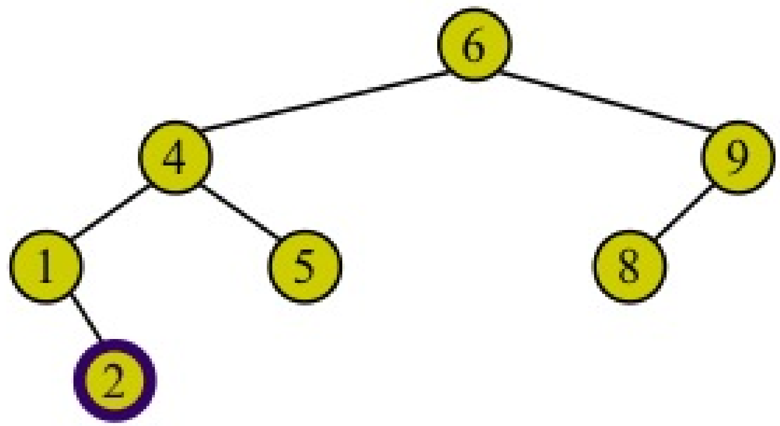
Comportamento dell'algoritmo di inserimento

- Se la chiave k e` uguale a quella dell'elemento di u, la chiamata termina senza fare niente
- Se la chiave k e` minore di quella dell'elemento di u l'inserimento viene effettuato ricorsivamente nel sottoalbero sinistro di u
- Se la chiave k e` maggiore di quella dell'elemento di u l'inserimento viene effettuato ricorsivamente nel sottoalbero destro di u
- Se u e` null  l'algoritmo crea un ndo foglia contenente e che diventa figlio del nodo su cui e` stato invocato precedentemente l'algoritmo .

Esempio

- Inseriamo 2





Inserimento di un elemento e

Simile alla ricerca di $k =$ e.chiave
Arriva a un riferimento `null` che va sostituito con la foglia contenente e

Se nell'albero non e' presente un'elemento con la stessa chiave di e, Inserisce l'elemento e nell'albero.

```
Inserisci( u, e ):
  IF (u == null) {
    u = NuovoNodo();
    u.dato = e;
    u.sx = u.dx = null;
  } ELSE IF (e.chiave < u.dato.chiave) {
    u.sx = Inserisci( u.sx, e );
  } ELSE IF (e.chiave > u.dato.chiave) {
    u.dx = Inserisci( u.dx, e );
  }
  RETURN u;
```

$O(h)$ tempo dove $h =$ altezza dell'albero

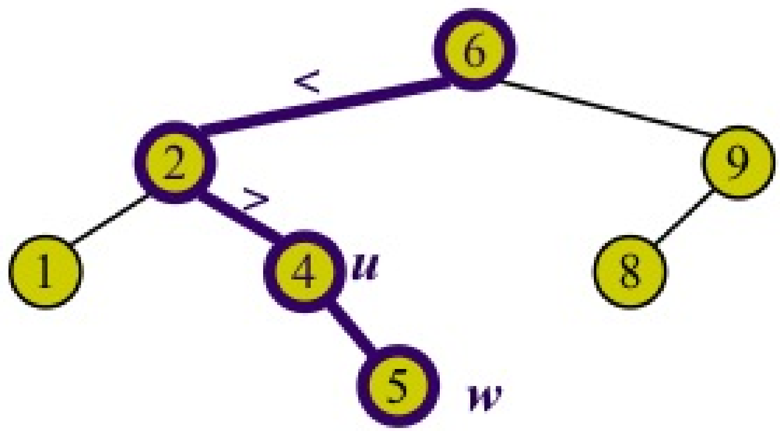
Algoritmo di cancellazione

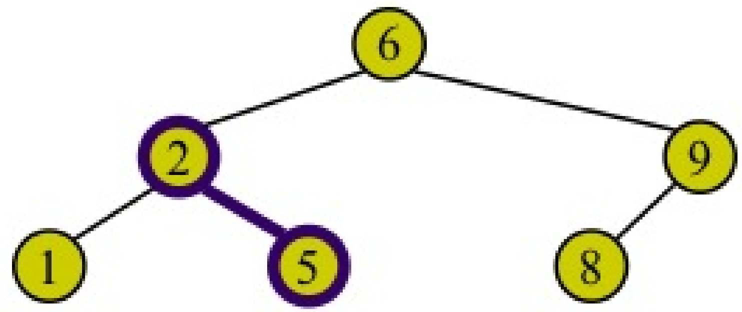
- Input: la chiave k da rimuovere e il nodo u radice del sottoalbero in cui si vuole rimuovere k

- L'algoritmo Cancella(u,k) funziona come segue::
  - Il nodo che contiene k ha al piu`un figlio
  - Il nodo che contiene k ha due figli

Algoritmo di cancellazione

- Caso 1: il nodo u da cancellare ha al piu` un figlio w
  - L'algoritmo di cancellazione rimuove il nodo u e lo rimpiazza con w
  - Esempio: rimoviamo la chiave 4

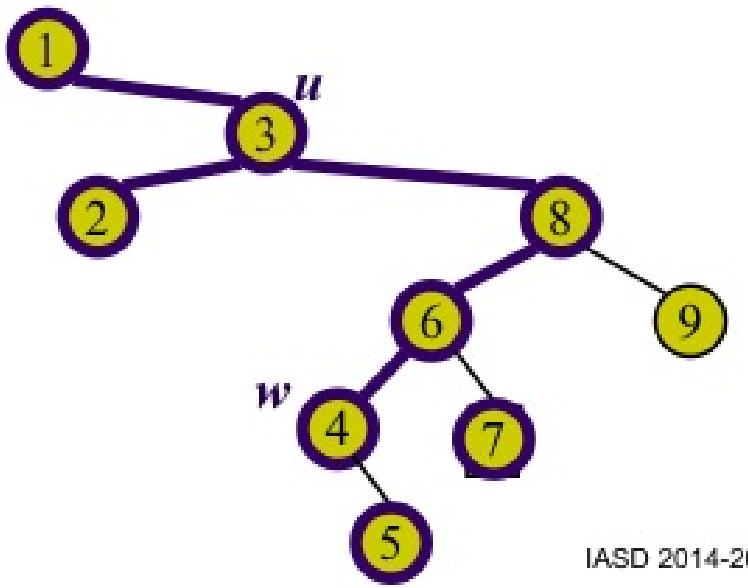


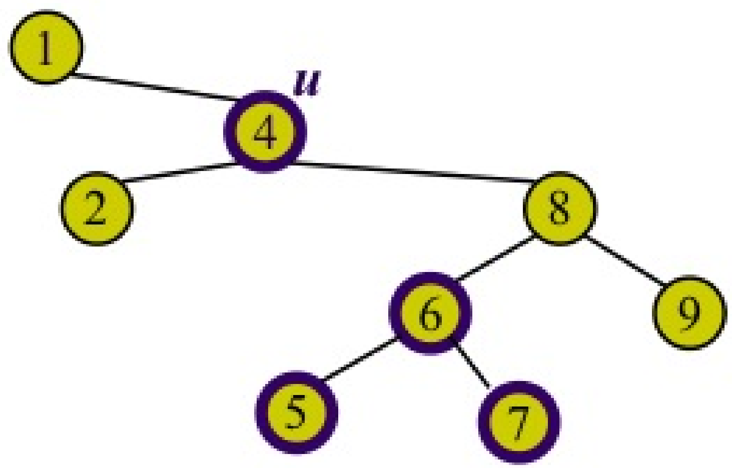


Algoritmo di cancellazione

- Caso 2: la chiave da cancellare è contenuta in un nodo u che ha due figli
  - Troviamo il primo nodo interno w visitato dopo u nella visita inorder (è il nodo interno più a sinistra nel sottoalbero destro di w e quindi non ha figlio sinistro)
  - Copiamo l'elemento contenuto in w nel nodo u
  - Rimuoviamo w come nel caso 1
- Esempio: removiamo la chiave 3

Il tempo di esecuzione è O(altezza)





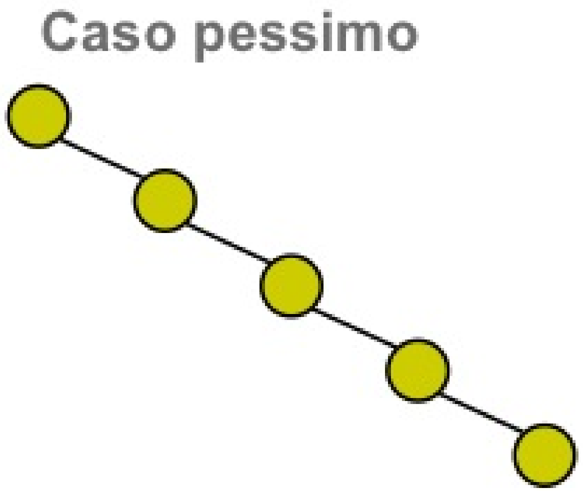
Cancellazione dell'elemento con chiave k in $O(h)$ tempo

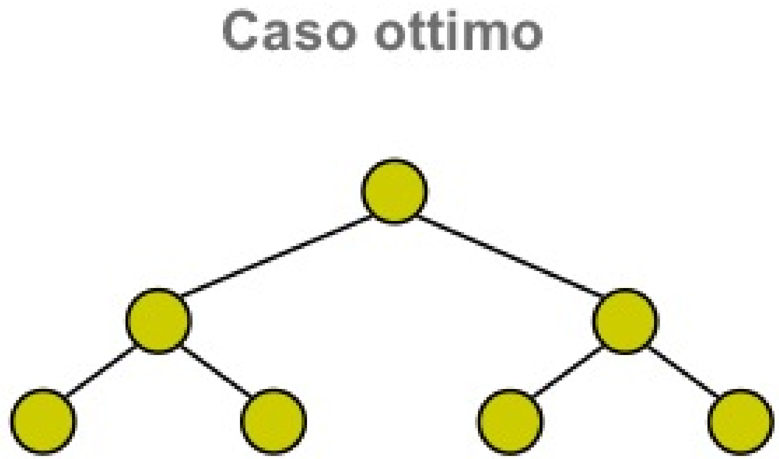
Caso 1 (linee 4-7): il nodo u è una foglia oppure ha un solo figlio
Caso 2 (linee 8-12): il nodo u ha due figli

```
Cancella( u, k ):
  IF (u != null) {
    IF (u.dato.chiave == k) {
      IF (u.sx == null) {
        u = u.dx;
      } ELSE IF (u.dx == null) {
        u = u.sx;
      } ELSE {
        w=MinimoSottoAlbero(u.dx);
        u.dato=w.dato;
        u.dx=Cancella(u.dx, w.dato.chiave);
      }
    } ELSE IF (k < u.dato.chiave) {
      u.sx = Cancella( u.sx, k );
    } ELSE IF (k > u.dato.chiave) {
      u.dx = Cancella( u.dx, k );
    }
  }
  RETURN u;
```

Complessità delle operazioni del dizionario implementato con albero binario di ricerca

- Consideriamo un dizionario con n entrate implementato con un albero binario di ricerca di altezza h
  - Lo spazio usato è $O(n)$
  - I metodi find, insert e remove impiegano tempo $O(h)$
- Nel caso pessimo $h = O(n)$; nel caso ottimo h= $O(\log n)$





Alberi AVL

- Alberi AVL :
- Creati negli anni '60
- AVL: acronimo derivato dalle iniziali degli inventori: Adel'son-Velsky e Landis

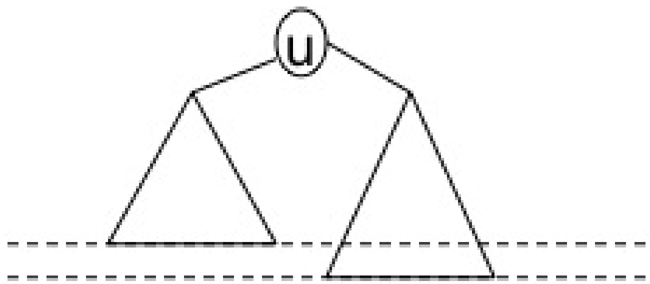
- Proprieta` albero AVL:

1. Albero binario di ricerca
2. Albero 1-bilanciato

Albero 1-bilanciato

- h(u) = altezza del sottoalbero T(u) radicato in u
- h(null) = -1

- Un nodo u è 1-bilanciato se | h(u.sx) – h(u.dx) |≤ 1



- Un albero e` 1-bilanciato se tutti nodi dell’albero sono 1-bilanciati

Altezza di un albero AVL

- Si puo` dimostrare che l'altezza di un albero AVL e` O(log n)

- Di conseguenza la ricerca di una richiede tempo O(log n)

Altezza h di un albero AVL: dimostrazione che h=O(log n)

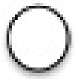
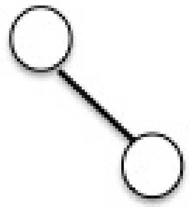
La dimostrazione consiste nel

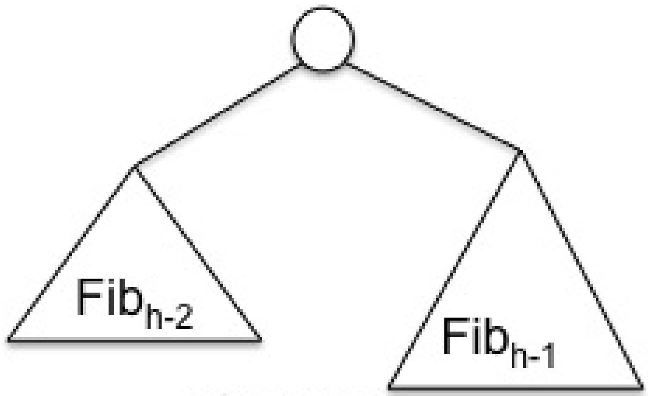
1. dimostrare che gli alberi AVL di altezza h hanno almeno c^h nodi per una certa costante $c>1$
   - Cio` implica immediatamente $h=O(\log n)$

- Per dimostrare il punto 1, consideriamo gli alberi AVL di altezza h con il piu` piccolo numero di nodi e dimostriamo che per essi vale la 1.

Altezza h di un albero AVL: dimostrazione che h=O(log n)

- Definizione di albero di Fibonacci di altezza h:

1. Per h=0, Fib_0 e` 
2. Per h=1 Fib_1 e`  oppure 
3. Per h>1, Fib_h e` ottenuto facendo diventare le radici di Fib_{h-1} e Fib_{h-2} figli di una nuova radice



Altezza h di un albero AVL: dimostrazione che h=O(log n)

Dimostreremo che

a. L'**albero di Fibonacci** di altezza h Fib_h e` l'albero 1-bilanciato di altezza h con il piu` piccolo numero di nodi
b. L'**albero di Fibonacci** di altezza h Fib_h ha almeno c^h nodi per una certa costante c>1.
   - n_h = numero di nodi di Fib_h

Altezza h di un albero AVL: dimostrazione che h=O(log n)

- Per dimostrare la a e cioe` che l'**albero di Fibonacci** di altezza h $Fib_h$ e` l'albero 1-bilanciato di altezza h con il piu` piccolo numero di nodi, dobbiamo mostrare che se eliminiamo un nodo da Fib_h
  - o diminuisce l'altezza dell'albero o l'albero non e` piu` 1-bilanciato (in entrambi i casi, l'albero non e` piu` un albero 1-bilanciato di altezza h).

Altezza h di un albero AVL: dimostrazione che h=O(log n)

- La dimostrazione della a e` per induzione:
  - Base induzione: la a e` banalmente verificata per h=0 e h=1
  - Passo induttivo: Supponiamo la a vera per ogni altezza t<h, con h>1.

    Osserviamo che
    - se il nodo viene eliminato dal sottoalbero Fib_{h-1} allora per ipotesi induttiva il sottoalbero che ne risulta
      - o non e` piu` 1-bilanciato e in questo caso l'intero albero non puo` essere 1-bilanciato
      - o non ha piu` altezza h-1 e in questo caso l'altezza dell'intero albero diminuisce
    - se il nodo viene eliminato dal sottoalbero Fib_{h-2} allora per ipotesi induttiva il sottoalbero che ne risulta
      - o non e` piu` 1-bilanciato e in questo caso l'intero albero non puo` essere 1-bilanciato
      - o non ha piu` altezza h-1 e in questo caso le altezze delle radici dei sottoalberi della radice differiscono di almeno 2

Altezza h di un albero AVL: dimostrazione che h=O(log n)

- Per dimostrare la b, dimostriamo prima che il numero di nodi n_h di Fib_h e` uguale a F_{h+3} -1
- Dimostriamolo per induzione:
  - Base induzione: la b e` banalmente verificata per h=0 e h=1 in quanto $n_0=1$ e $F_3=2$, e $n_1=2$ e $F_4=3$
  - Passo induttivo: Supponiamo la h vera per ogni altezza $t<h$, con $h>1$.

    Osserviamo che
    - $n_h=n_{h-2}+n_{h-1}+1$ e applicando l'ipotesi induttiva a n_{h-2} e n_{h-1} si ha

    $$n_h=n_{h-2}+n_{h-1}+1=F_{h+1}-1+F_{h+2}-1+1=F_{h+1}+F_{h+2}-1=F_{h+3}-1$$

Altezza h di un albero AVL: dimostrazione che h=O(log n)

- Abbiamo dimostrato che $n_h = F_{h+3} - 1$
- Utilizzando la ben nota uguaglianza

$$F_h = \frac{\phi^h - (1-\phi)^h}{\sqrt{5}}, \quad \text{dove } \phi = \frac{1+\sqrt{5}}{2} \approx 1,6180339\cdots$$

si ha che $F_h > (\phi^h - 1)/\sqrt{5}$ (si puo` dim. Per induzione) per cui

$$n_h = F_{h+3} - 1 = F_{h+2} + F_{h+1} - 1 \geq F_{h+2} + F_1 - 1 = F_{h+2}$$
$$= ((1+\sqrt{5})/2)^{h+2})/\sqrt{5} - 1/\sqrt{5} = ((1+\sqrt{5})/2)^{h})\,((1+\sqrt{5})/2)^{2})/\sqrt{5} - 1\sqrt{5}$$
$$> (1+\sqrt{5})/2)^{h} - 1/\sqrt{5} > (\sqrt{5}/2)^{h}$$

- Abbiamo quindi dimostrato che esiste c>1 tale che $n_h > c^h$, per ogni h>1 .