Algoritmi e Strutture Dati

Capitolo 8
Code con priorità:
Heap binomiali

Copyright © 2004 - The McGraw - Hill Companies, srl
Mc Graw Hill

Domanda: Durante l'operazione di Insert, come faccio a trovare la giusta posizione della foglia che devo creare?

Risposta: Mantengo un puntatore alla prima foglia disponibile. Per tenere aggiornato questo puntatore, ogni volta che faccio una Insert lo aggiorno al puntatore successivo, che può essere trovato direttamente nel padre della foglia appena inserita, se questi non è saturo (ovvero la foglia appena inserita non è un d-esimo figlio), oppure nel nodo subito a destra del padre, che può essere trovato risalendo dal padre verso l'alto, e fermandosi non appena si risale da un nodo che è un k-esimo figlio del padre con $k<d$; a quel punto si ridiscende sul $(k+1)$-esimo figlio, e si segue sempre il puntatore al primo figlio. In questo modo, si spende tempo $O(\log_d n)$. Ovviamente va gestito il caso in cui un inserimento satura l'utlimo livello del d-heap. In tal caso, il puntatore alla prima foglia disponibile va aggiornato al primo figlio della foglia più a sinistra del d-heap.

Ragionamento analogo può essere fatto per mantenere il puntatore all'ultima foglia più in basso a destra del d-heap, che entra in gioco per eseguire la Delete.

Copyright © 2004 - The McGraw - Hill Companies, srl

Soluzione esercizio di approfondimento #1

Fornire un'implementazione alternativa dell'operazione di **merge(heap d-ario c1, heap d-ario c2)** in cui gli elementi di uno dei due heap vengono aggiunti sequenzialmente all'altro heap. Analizzarne quindi la convenienza asintotica rispetto all'implementazione classica di costo $\Theta(n)$.

Soluzione: Sia $k=\min\{|c_1|,|c_2|\}$. Inseriamo ad uno ad uno tutti gli elementi della coda più piccola nella coda più grande; questo costa $O(k \log_d n)$, dove $n=|c_1|+|c_2|$. L'approccio conviene quindi per $k \log_d n=o(n)$, cioè per

$$k=o(n/\log_d n).$$

Copyright © 2004 - The McGraw - Hill Companies, srl

Soluzione esercizio di approfondimento #2

	Increase Key	Decrease Key	Merge
Array non ord.	O(1)	O(1)	$\Theta(k)$ $k=\min\{\lvert c_1\rvert,\lvert c_2\rvert\}$
Array Ordinato	O(n)	O(n)	$\Theta(n)$ $n=\lvert c_1\rvert+\lvert c_2\rvert$
Lista non Ordinata	O(1)	O(1)	O(1)
Lista Ordinata	O(n)	O(n)	O(n)

Fusione ordinata con array di appoggio

Copyright © 2004 - The McGraw - Hill Companies, srl

Riepilogo

	Find Min	Insert	Delete	DelMin	Incr. Key	Decr. Key	merge
Array non ord.	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(k)$ $k=\min\{\lvert c_1\rvert,\lvert c_2\rvert\}$
Array ordinato	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$\Theta(n)$
Lista non ordinata	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$	$O(1)$	$O(1)$	$O(1)$
Lista ordinata	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
d-Heap	$O(1)$	$O(\log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$\Theta(n)$

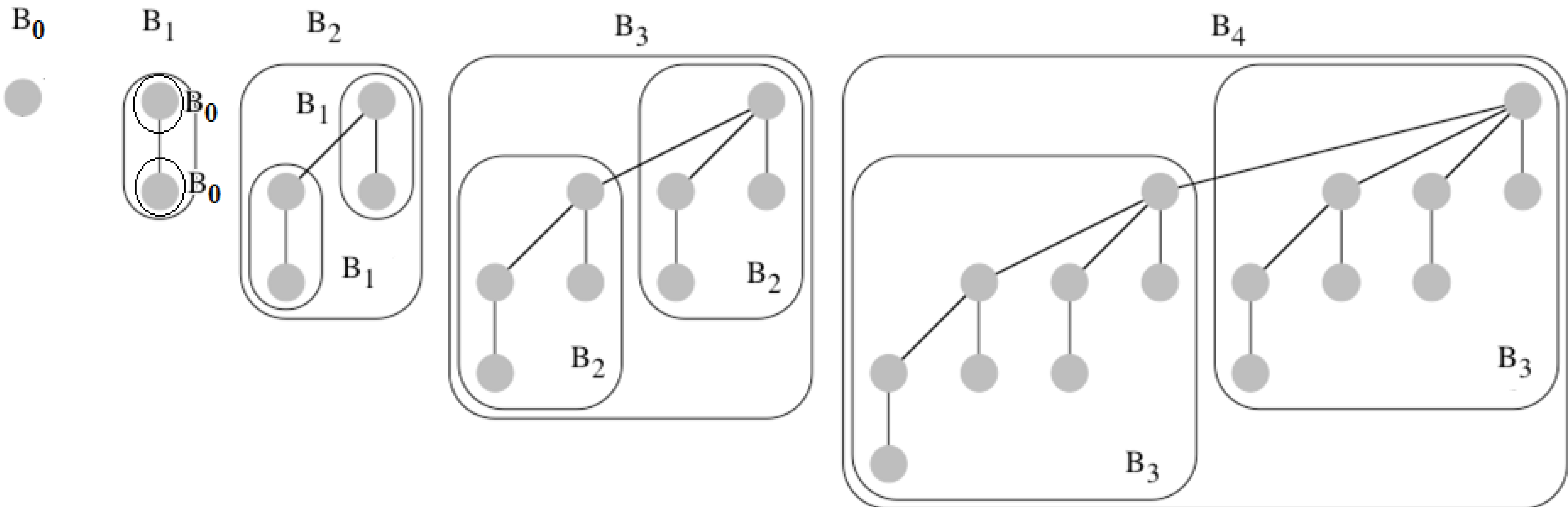
⇒ Il nostro obiettivo di implementare una coda di priorità con una struttura dati che non comporti costi lineari non è ancora raggiunto…

Copyright © 2004 - The McGraw - Hill Companies, srl

Alberi binomiali

Un albero binomiale B_h è definito ricorsivamente come segue:

1. B_0 consiste di un unico nodo
2. Per $i>0$, B_i è ottenuto fondendo due alberi binomiali B_{i-1}, ponendo la radice dell'uno come figlia della radice dell'altro



Copyright © 2004 - The McGraw - Hill Companies, srl

Proprietà strutturali

Un albero binomiale B_h *gode delle seguenti proprietà:*

1. *Numero di nodi* ($|B_h|$): $n = 2^h \Rightarrow h = \log_2 n$.
2. *Grado della radice:* $D(B_h) = h = \log_2 n$.
3. *Altezza:* $H(B_h) = h = \log_2 n$.
4. *Figli della radice: i sottoalberi radicati nei figli della radice di* B_h *sono* $B_0, B_1, \ldots, B_{h-1}$.

Si dimostrano tutte facilmente per induzione (da fare per esercizio)

Copyright © 2004 - The McGraw - Hill Companies, srl

Heap binomiali

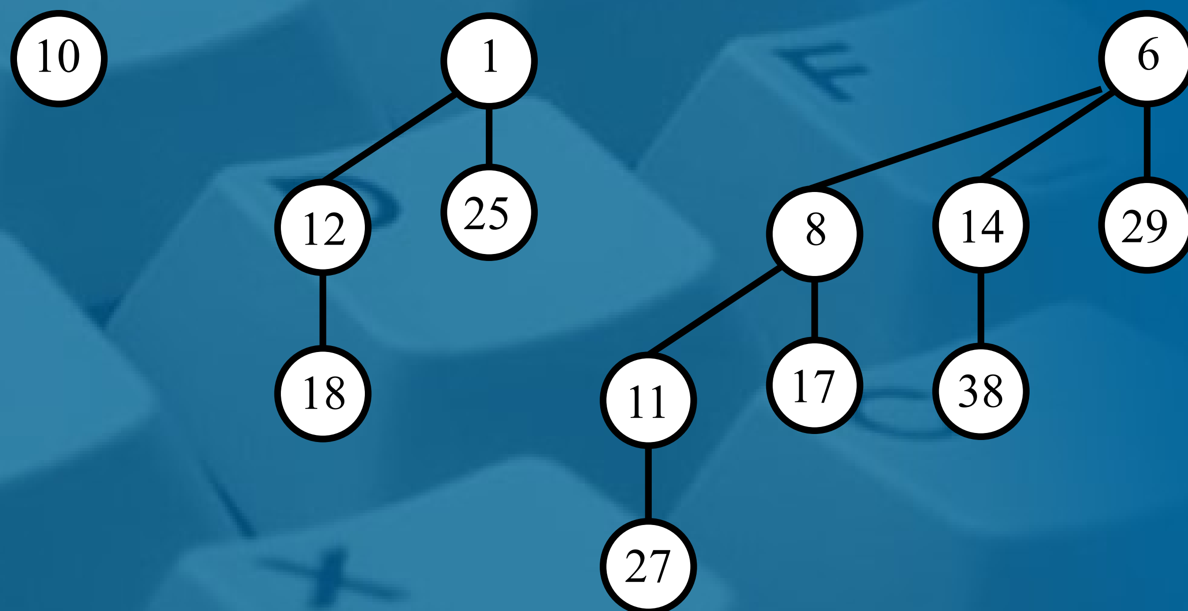
Un heap binomiale è una foresta (ovvero, un insieme) di alberi binomiali che gode delle seguenti proprietà:

1. Unicità: per ogni intero $i \geq 0$, esiste al più un B_i nella foresta
2. Contenuto informativo: in ciascuno degli alberi binomiali, ogni nodo v contiene un elemento elem(v) ed una chiave chiave(v) presa da un dominio totalmente ordinato
3. Ordinamento a min-heap: in ciascuno degli alberi binomiali, chiave(parent(v)) ≤ chiave(v) per ogni nodo v diverso da una delle radici

Copyright © 2004 - The McGraw - Hill Companies, srl

Un esempio di Heap Binomiale con n=13 nodi

in questa direzione non è presente un ordinamento



in questa direzione è presente un ordinamento

Domanda: quanti alberi binomiali può avere al massimo un heap binomiale con n nodi?

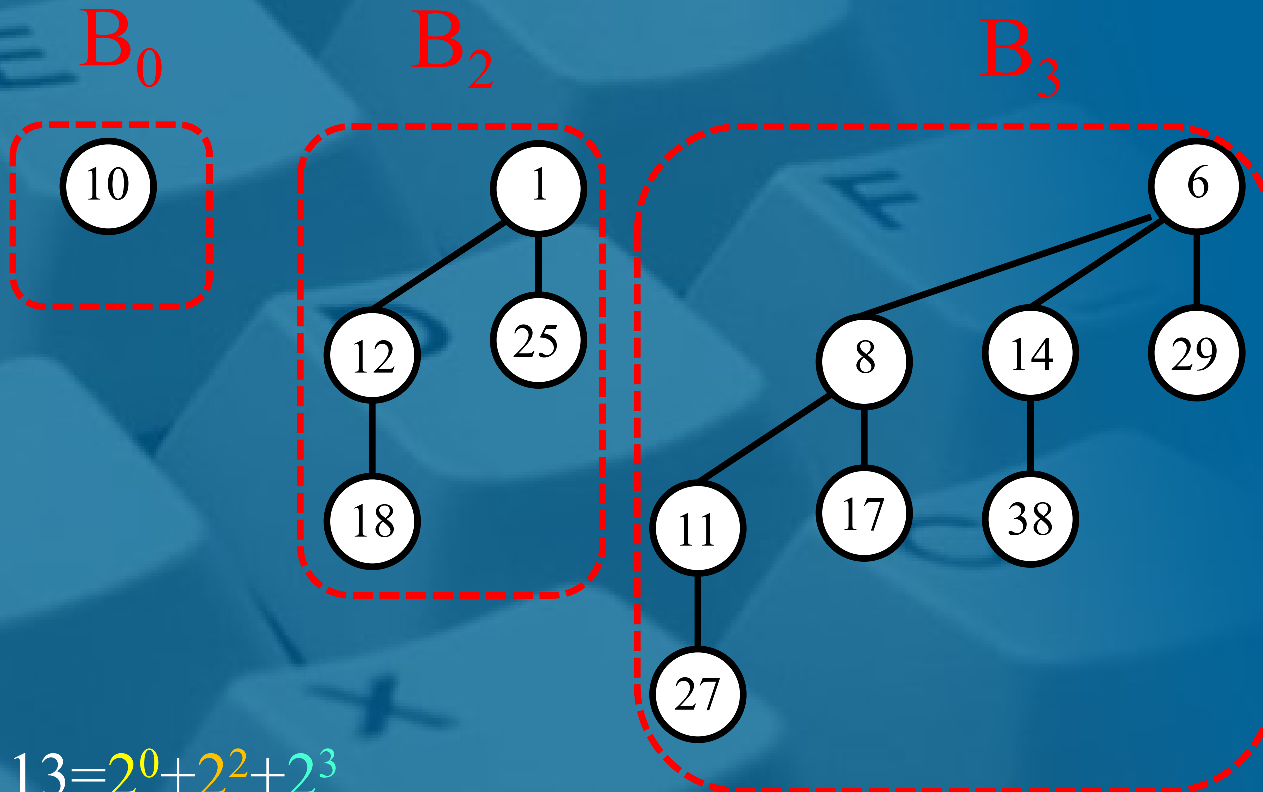
Proprietà topologiche

- Dalla proprietà di unicità degli alberi binomiali che lo costituiscono, ne deriva che un heap binomiale di n elementi è formato dagli alberi binomiali $B_{i_0}, B_{i_1}, \ldots, B_{i_h}$, dove $i_0 < i_1 < \ldots < i_h$ e $n = 2^{i_0} + 2^{i_1} + \ldots + 2^{i_h}$ (cioè, corrispondono alle posizioni degli 1 nella rappresentazione in base 2 di n)

$\Rightarrow$ Ne consegue che in un heap binomiale con n nodi, vi sono al più $\lceil \log n \rceil$ alberi binomiali, ciascuno con grado ed altezza $O(\log n)$

Copyright © 2004 - The McGraw - Hill Companies, srl

Un esempio di Heap Binomiale con n=13 nodi

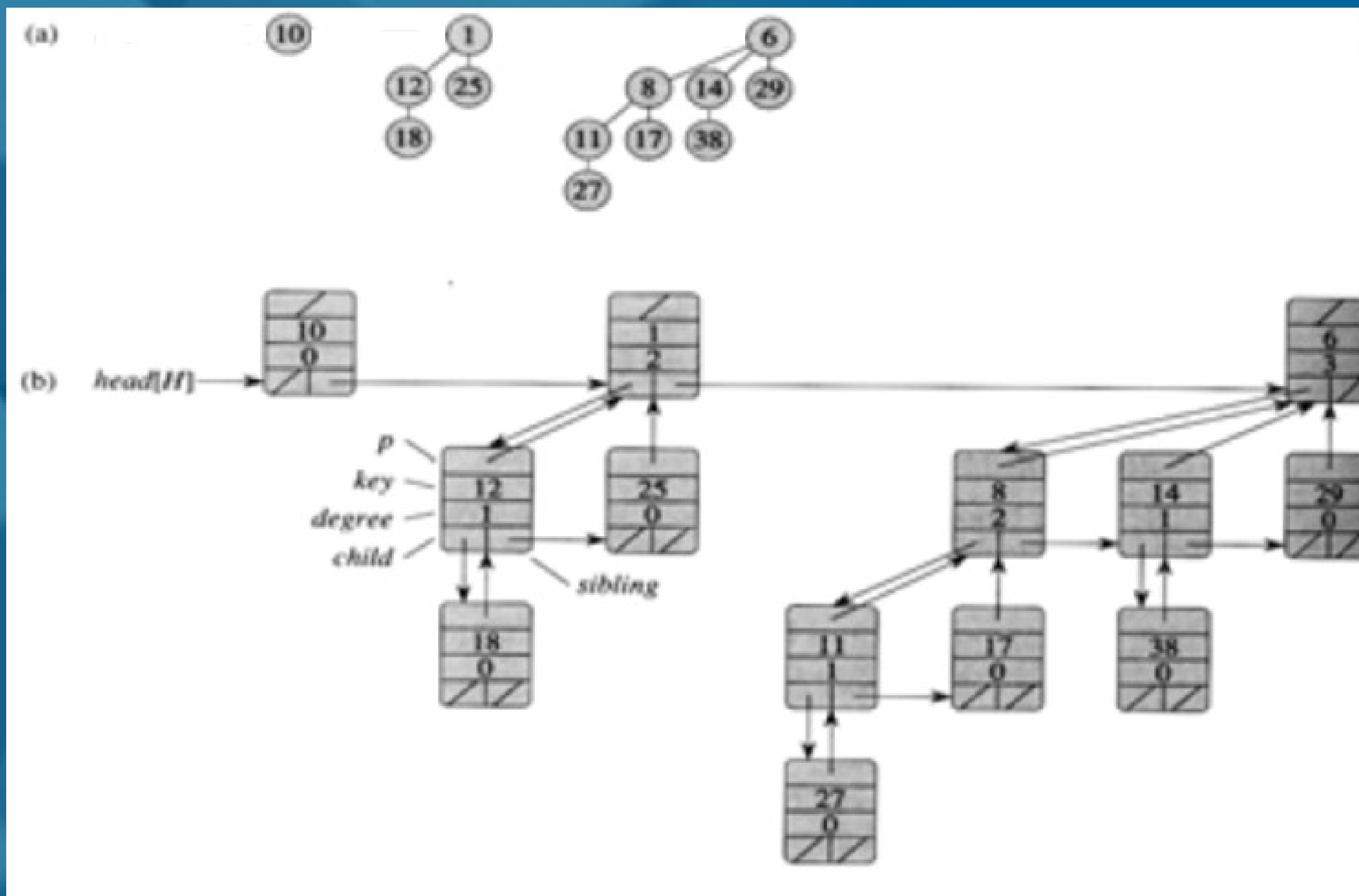


$13=2^0+2^2+2^3$

13 in binario: 1101

Implementazione di un heap binomiale

Vediamo come è implementato fisicamente, l'HB di 13 elementi $H=\{10,1,12,25,18,6,8,14,29,11,17,38,27\}$ del nostro esempio. Come detto, $13_{10}=1101_2$, e quindi H conterrà gli alberi binomiali B_0, B_2 e B_3, ordinati a min-heap:



Copyright © 2004 - The McGraw - Hill Companies, srl

Procedura ausiliaria

Alcune operazioni eseguite sull'HB possono violare la proprietà di unicità; la seguente procedura serve proprio a ripristinare tale proprietà (ipotizziamo di scorrere la lista delle radici da sinistra verso destra, in ordine crescente rispetto all'indice degli alberi binomiali)

procedura `ristruttura`()
  $i = 0$
  while (esistono ancora due B_i) **do**
    si fondono i due B_i per formare un albero B_{i+1}, ponendo la radice con chiave più piccola come genitore della radice con chiave più grande
    $i = i + 1$

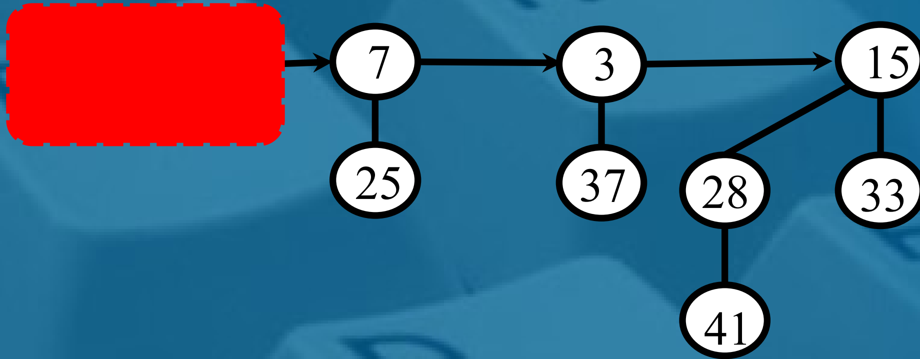
T(n) è proporzionale al numero di alberi binomiali in input

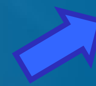
(ogni fusione diminuisce di uno il numero di alberi binomiali residui)

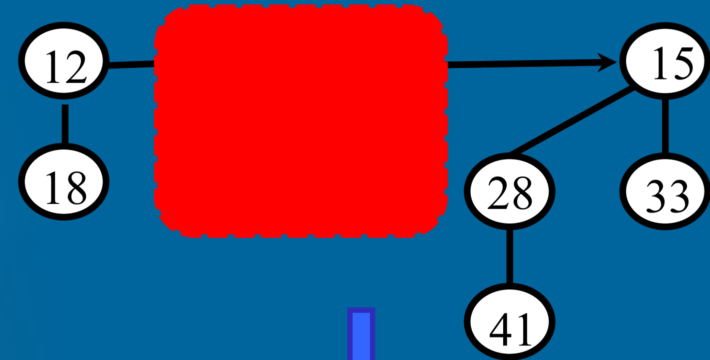
Copyright © 2004 - The McGraw - Hill Companies, srl

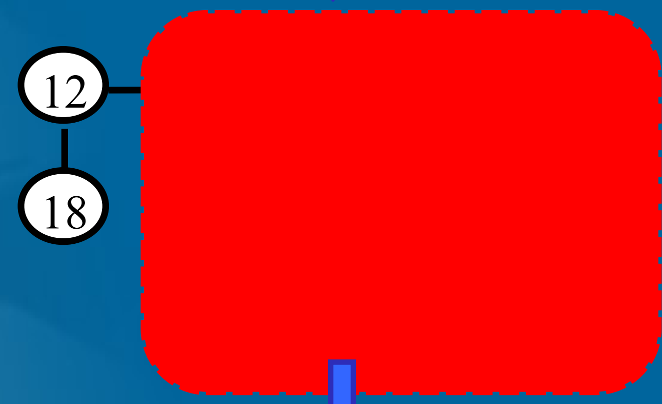
Ristruttura(H)

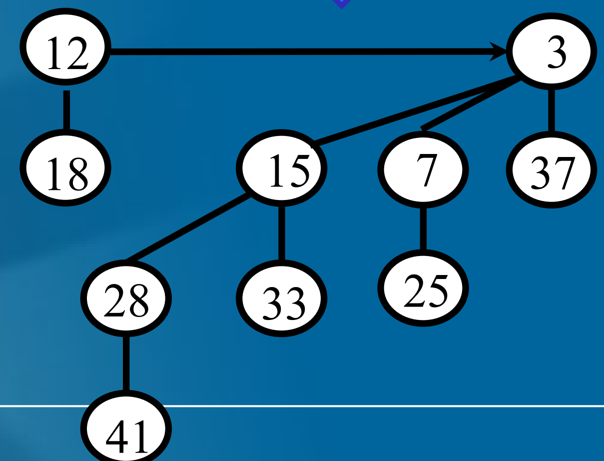
H











Realizzazione (1/3)

classe `HeapBinomiale` **implementa** `CodaPriorita`:

dati:

una foresta H con n nodi, ciascuno contenente un elemento di tipo *elem* e una chiave di tipo *chiave* presa da un universo totalmente ordinato.

operazioni:

`findMin()` $\rightarrow$ *elem*

scorre le radici in H e restituisce l'elemento a chiave minima. ⇒ Costo O(log n)

`insert`(*elem* e, *chiave* k)

aggiunge ad H un nuovo B_0 con dati e e k. Ripristina poi la proprietà di unicità in H mediante fusioni successive dei doppioni B_i. ⇒ Costo O(log n), in quanto durante l'esecuzione della procedura ristruttura esistono al più tre B_i, per ogni $i \geq 0$

Copyright © 2004 - The McGraw - Hill Companies, srl

Realizzazione (2/3)

`deleteMin()`
trova l'albero B_h con radice a chiave minima. Togliendo la radice a B_h, esso si spezza in h alberi binomiali $B_0, \ldots, B_{h-1}$, che vengono aggiunti ad H. Ripristina poi la proprietà di unicità in H mediante fusioni successive dei doppioni B_i. ⇒ Costo O(log n)

`decreaseKey`$(elem\ e, chiave\ \Delta > 0)$
decrementa di Δ la chiave nel nodo v contenente l'elemento e. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi. ⇒ Costo O(log n)

`delete`$(elem\ e)$
richiama `decreaseKey`(e, ∞) e poi `deleteMin()`.
⇒ Costo O(log n)

Copyright © 2004 - The McGraw - Hill Companies, srl

Un esempio di deleteMin

(a) head[H]

(b) head[H]

x

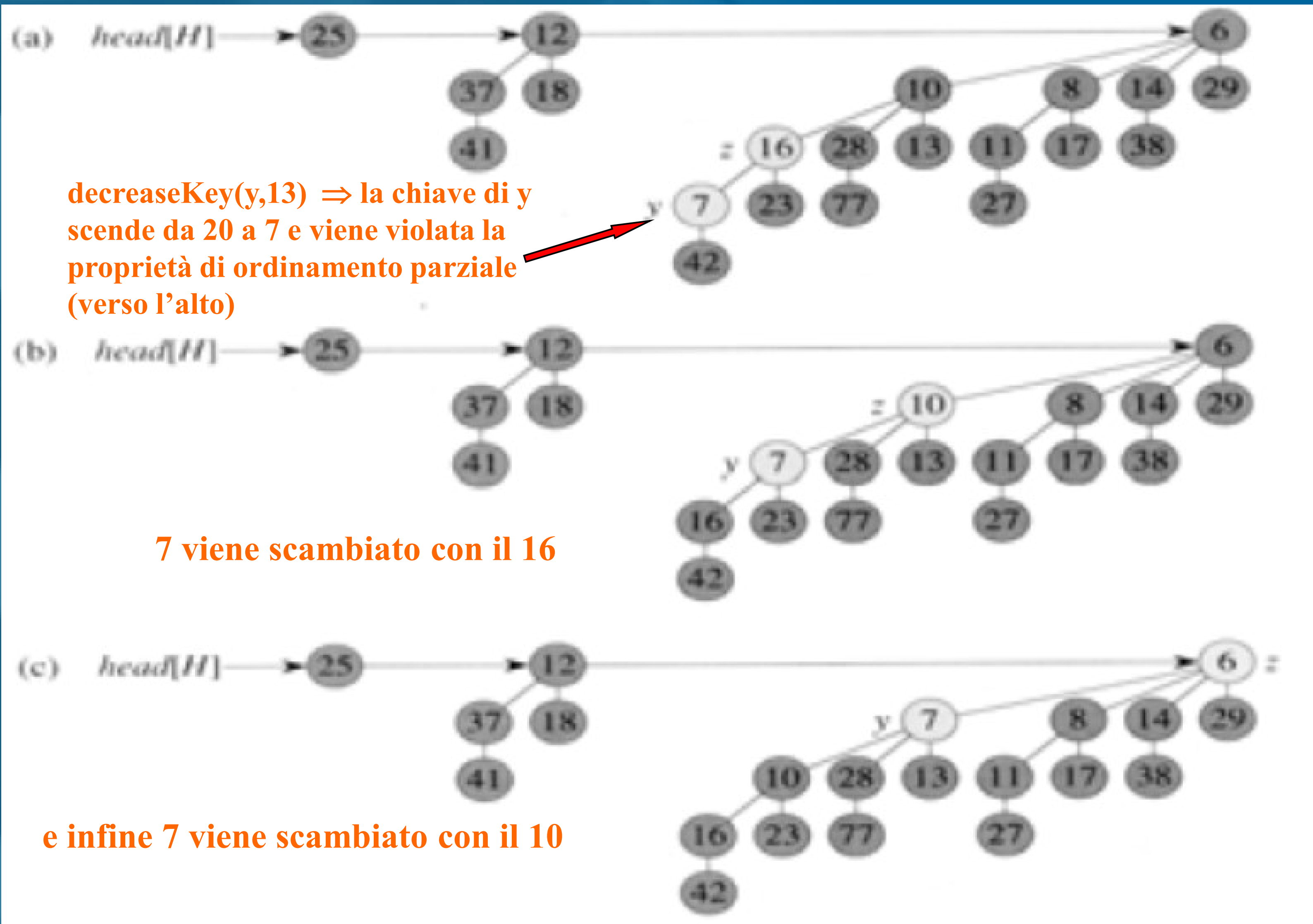
(c) head[H]

Ristruttura

(d) head[H]

Copyright © 2004 - The McGraw - Hill Companies, srl
Mc Graw Hill

Un esempio di decreaseKey



Copyright © 2004 - The McGraw - Hill Companies, srl

Realizzazione (3/3)

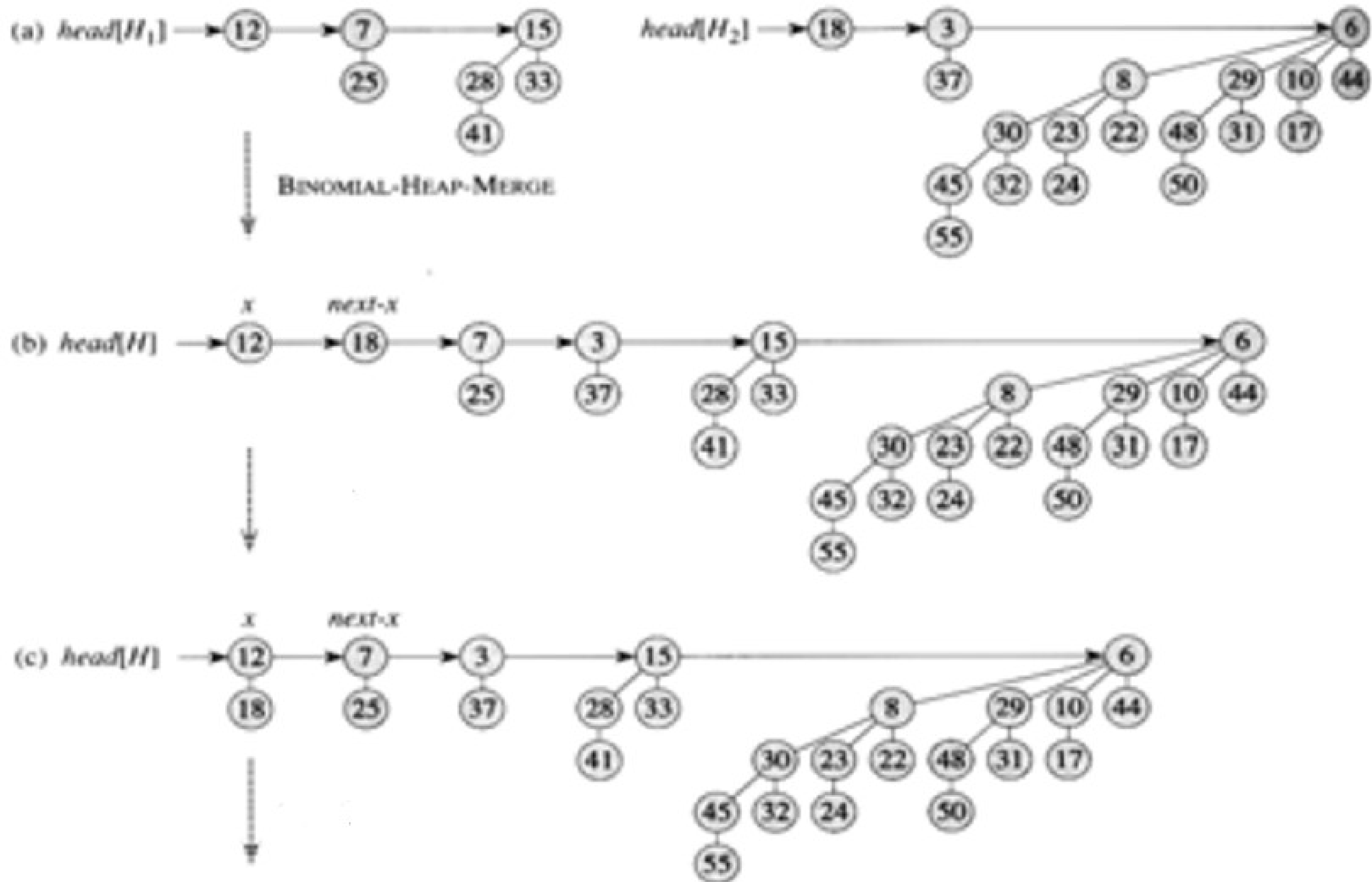
`increaseKey`$(elem\ e, chiave\ \Delta > 0)$
richiama `delete`(e) e poi `insert`$(e, k + \Delta)$, dove k è la chiave associata all'elemento e. $\Rightarrow$ Costo O(log n)

`merge`$($`HeapBin`$.H_1,$`HeapBin`$.H_2) \rightarrow$ `HeapBin`.
unisce gli alberi in H_1 e H_2 in una nuova foresta di alberi binomiali H.
Ripristina poi la proprietà di unicità dell'heap binomiale in H mediante fusioni successive dei doppioni B_i.
$\Rightarrow$ Costo O(log n)

Tutte le operazioni richiedono tempo T(n) = O(log n) (ricorda che durante l'esecuzione di ogni procedura ristruttura esistono infatti al più tre B_i, per ogni $i \geq 0$)

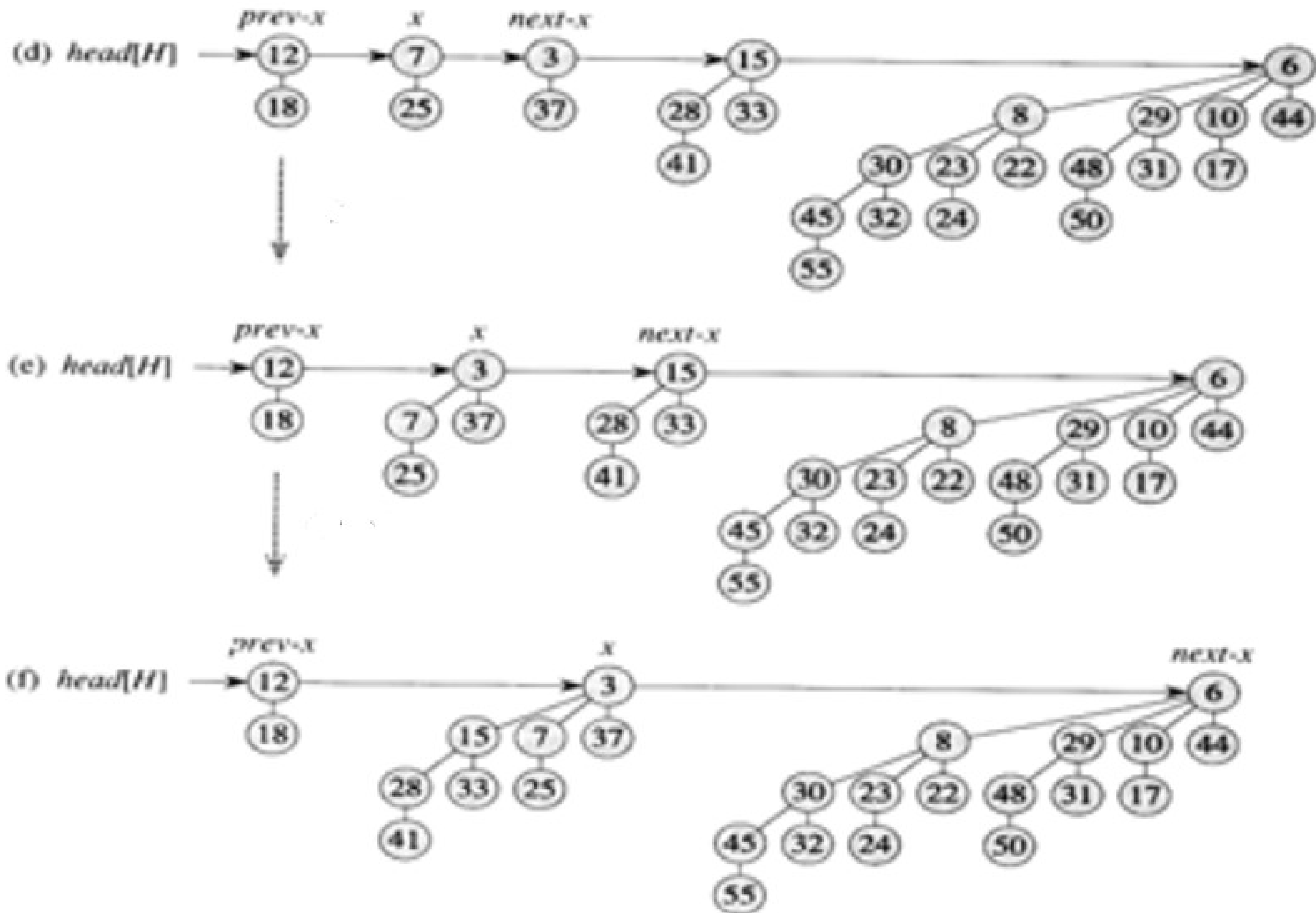
Copyright © 2004 - The McGraw - Hill Companies, srl

Un esempio di Merge



Copyright © 2004 - The McGraw - Hill Companies, srl

Un esempio di **Merge** (2)



Copyright © 2004 - The McGraw - Hill Companies, srl

Heap di Fibonacci

Copyright © 2004 - The McGraw - Hill Companies, srl

Heap di Fibonacci

Heap binomiale rilassato: si ottiene da un heap binomiale rilassando la proprietà di unicità dei B_i ed utilizzando un atteggiamento più "pigro" durante l'operazione insert (perché ristrutturare subito la foresta quando potremmo farlo dopo?)

Heap di Fibonacci: si ottiene da un heap binomiale rilassato indebolendo la proprietà di struttura dei B_i che non sono più necessariamente alberi binomiali

Analisi sofisticata: i tempi di esecuzione sono ammortizzati su sequenze di operazioni, cioè dividendo il costo complessivo della sequenza di operazioni per il numero di operazioni della sequenza

Copyright © 2004 - The McGraw - Hill Companies, srl

Teorema (Fredman e Tarjan, 1987)

Usando un Heap di Fibonacci, una qualsiasi sequenza di k insert, d delete, f findMin, m deleteMin, Δ increaseKey, δ decreaseKey, and μ merge costa tempo (nel caso peggiore)

$$O(k+f+\delta+\mu+(d+m+\Delta)\log n)$$

Si noti che le singole operazioni di delete, deleteMin e increaseKey possono costare $\omega(\log n)$, ma in ammortizzato costeranno ciascuna $O(\log n)$, mentre la singola operazione di decreaseKey può costare $\omega(1)$ ma in ammortizzato costerà $O(1)$

Conclusioni: tabella riassuntiva

	FindMin	Insert	DelMin	DecKey	Delete	IncKey	merge
d-Heap (d cost.)	O(1)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Θ(n)
Heap Binom.	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Heap Fibon.	O(1)	O(1)	O(log n)*	**O(1)***	O(log n)*	O(log n)*	O(1)

L'analisi per d-Heap e Heap Binomiali è nel caso peggiore, mentre quella per gli Heap di Fibonacci è ammortizzata (per le operazioni asteriscate)

Copyright © 2004 - The McGraw - Hill Companies, srl

Esercizi di approfondimento

- Creare ed unire 2 Heap Binomiali definiti sui seguenti insiemi:

  $A_1=\{3,5,7,21,2,4\}$

  $A_2=\{1,11,6,22,13,12,23\}$

- Implementare l'operazione `increaseKey` ripristinando la proprietà di ordinamento parziale verso il basso. Fornire lo pseudocodice e analizzare la complessità temporale. Sotto quale condizioni è conveniente questo approccio rispetto a quello fornito a lezione?

Copyright © 2004 - The McGraw - Hill Companies, srl