Algoritmi e Strutture Dati

Capitolo 8
Code con priorità:
Heap binomiali

Riepilogo

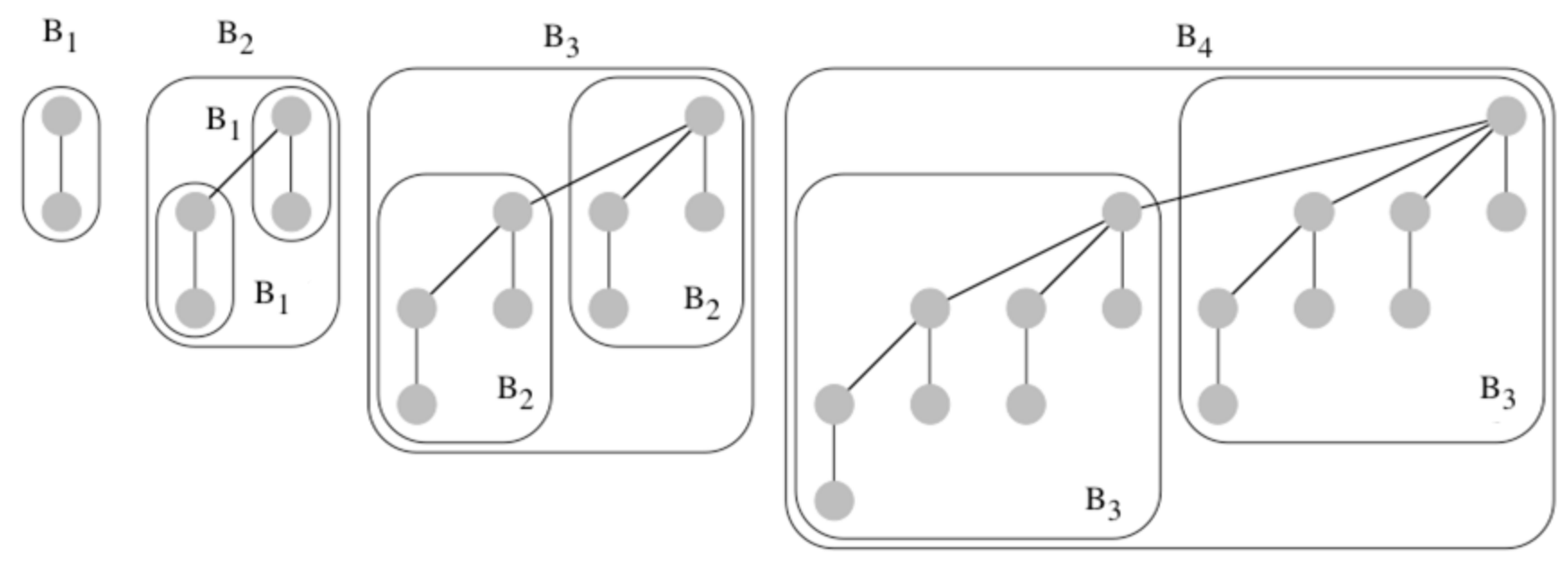
	Find Min	Insert	Delete	DelMin	Incr. Key	Decr. Key	merge
Array non ord.	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(k)$
Array ordinato	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$\Theta(n)$
Lista non ordinata	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$	$O(1)$	$O(1)$	$O(1)$
Lista ordinata	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
d-Heap	$O(1)$	$O(\log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(n)$

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

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+1} è ottenuto fondendo due alberi binomiali B_i, ponendo la radice dell'uno come figlia della radice dell'altro



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 (fatelo per esercizio)

Heap binomiali

Un heap binomiale è una foresta 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(v) ≥ chiave(parent(v)) per ogni nodo v diverso da una delle radici

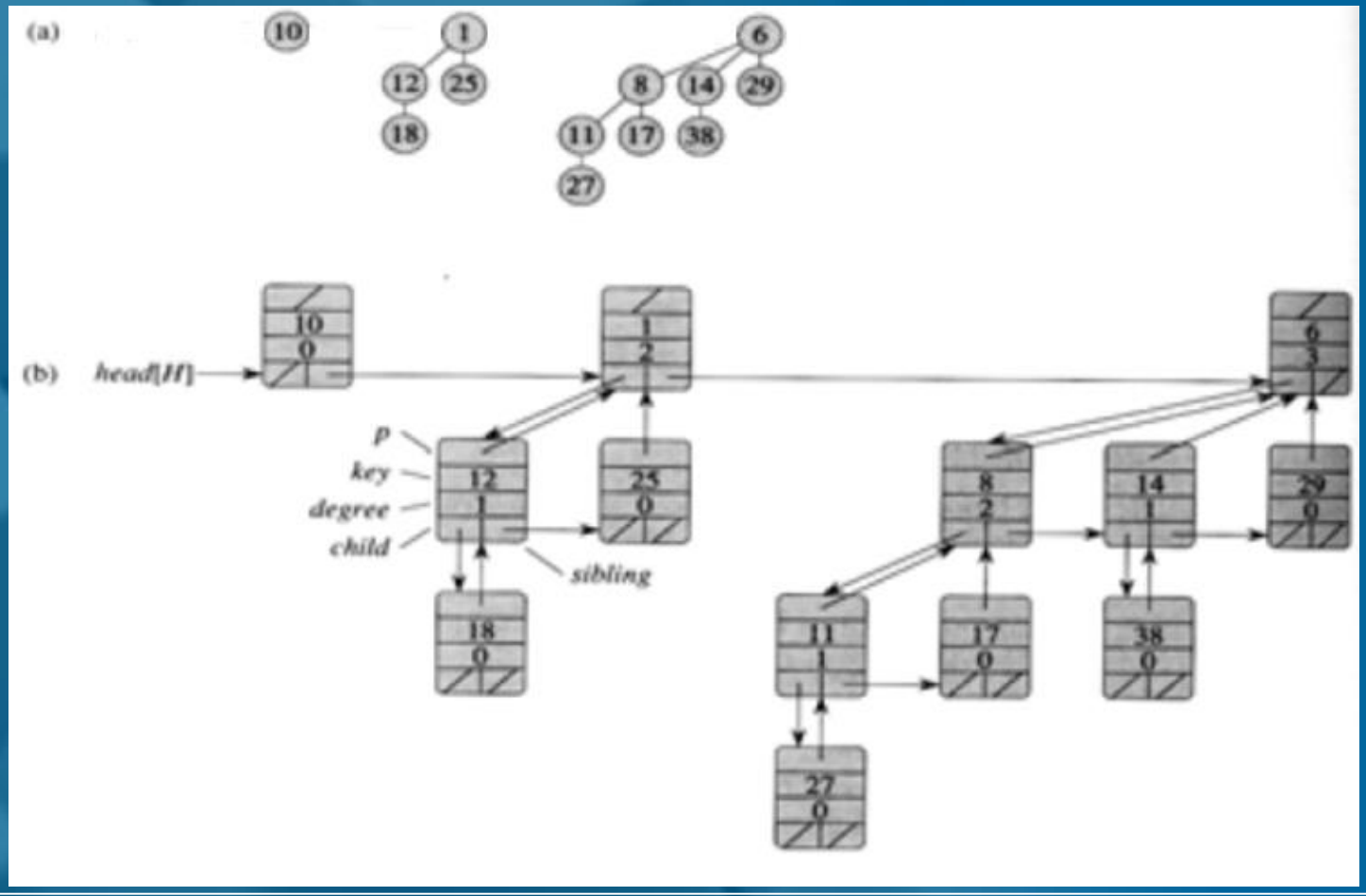
Proprietà topologiche

- 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 $i_0 < i_1 < \ldots < i_h$ 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 $O(\log n)$

Un esempio di heap binomiale

Vediamo come è fatto, logicamente (a) e fisicamente (b), un HB di 13 elementi $H=\{10,1,12,25,18,6,8,14,29,11,17,38,27\}$. Poiché $13_{10}=1101_2$, ne consegue che H conterrà gli alberi binomiali B_0, B_2 e B_3 , ordinati a min-heap:



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

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.

`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.

Realizzazione (2/3)

`deleteMin()`

trova l'albero T_h con radice a chiave minima. Togliendo la radice a T_h, esso si spezza in h alberi binomiali $T_0, \ldots, T_{h-1}$, che vengono aggiunti ad H. Ripristina poi la proprietà di unicità in H mediante fusioni successive dei doppioni B_i.

`decreaseKey`$(elem\ e, chiave\ \Delta)$

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.

`delete`$(elem\ e)$

richiama `decreaseKey`(e, ∞) e poi `deleteMin`$()$.

Realizzazione (3/3)

`increaseKey`$(elem\ e, chiave\ \Delta)$

richiama `delete`(e) e poi `insert`$(e, k + \Delta)$, dove k è la chiave associata all'elemento e.

`merge`$(\texttt{HeapBin}.H_1, \texttt{HeapBin}.H_2) \rightarrow$ `HeapBin`.

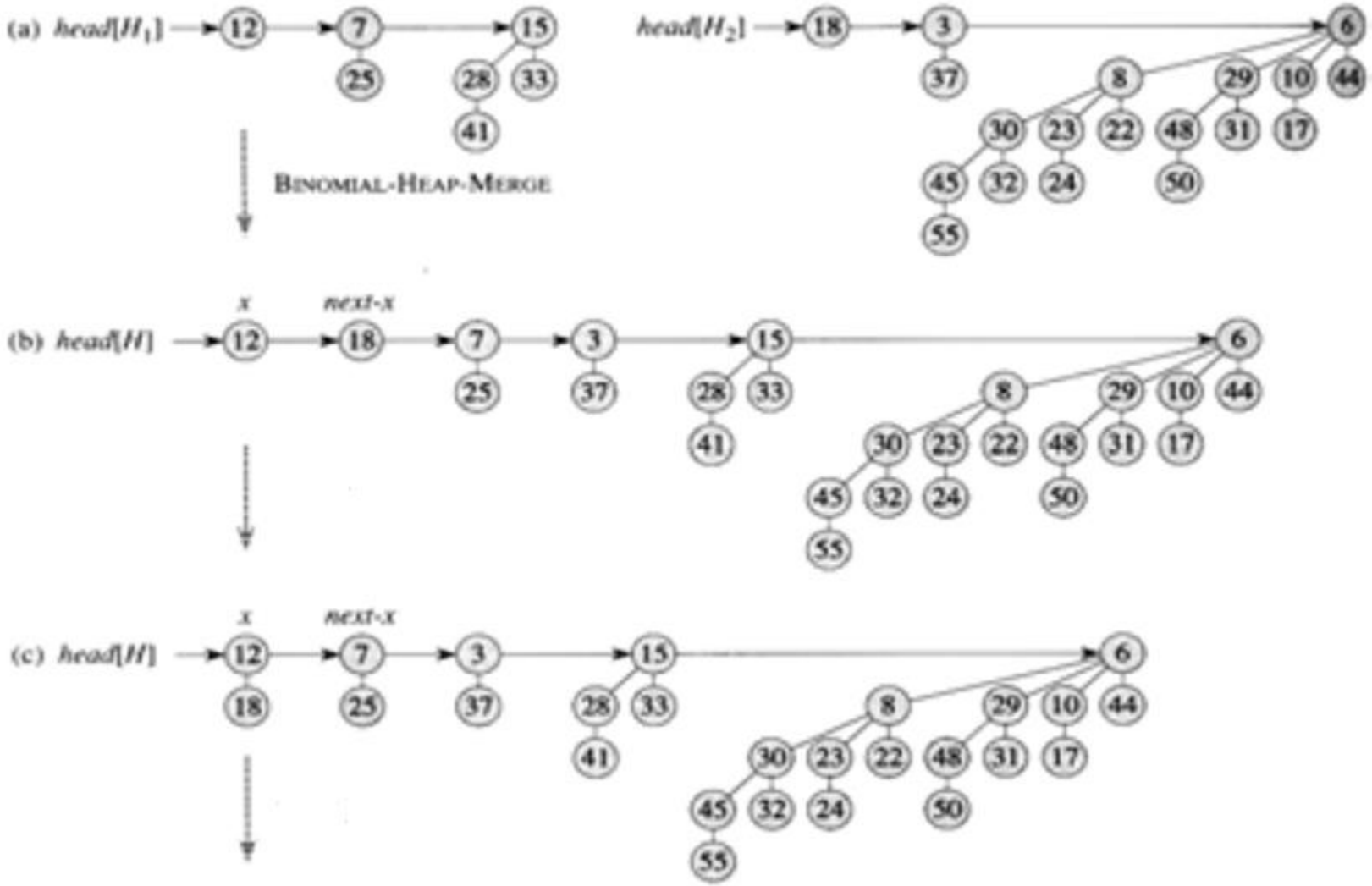
unisce gli alberi in H_1 e H_2 in una nuova foresta di alberi binomiali H.

Ripristina poi la proprietà di unicità dell'heap binomiale in H mediante fusioni successive dei doppioni B_i.

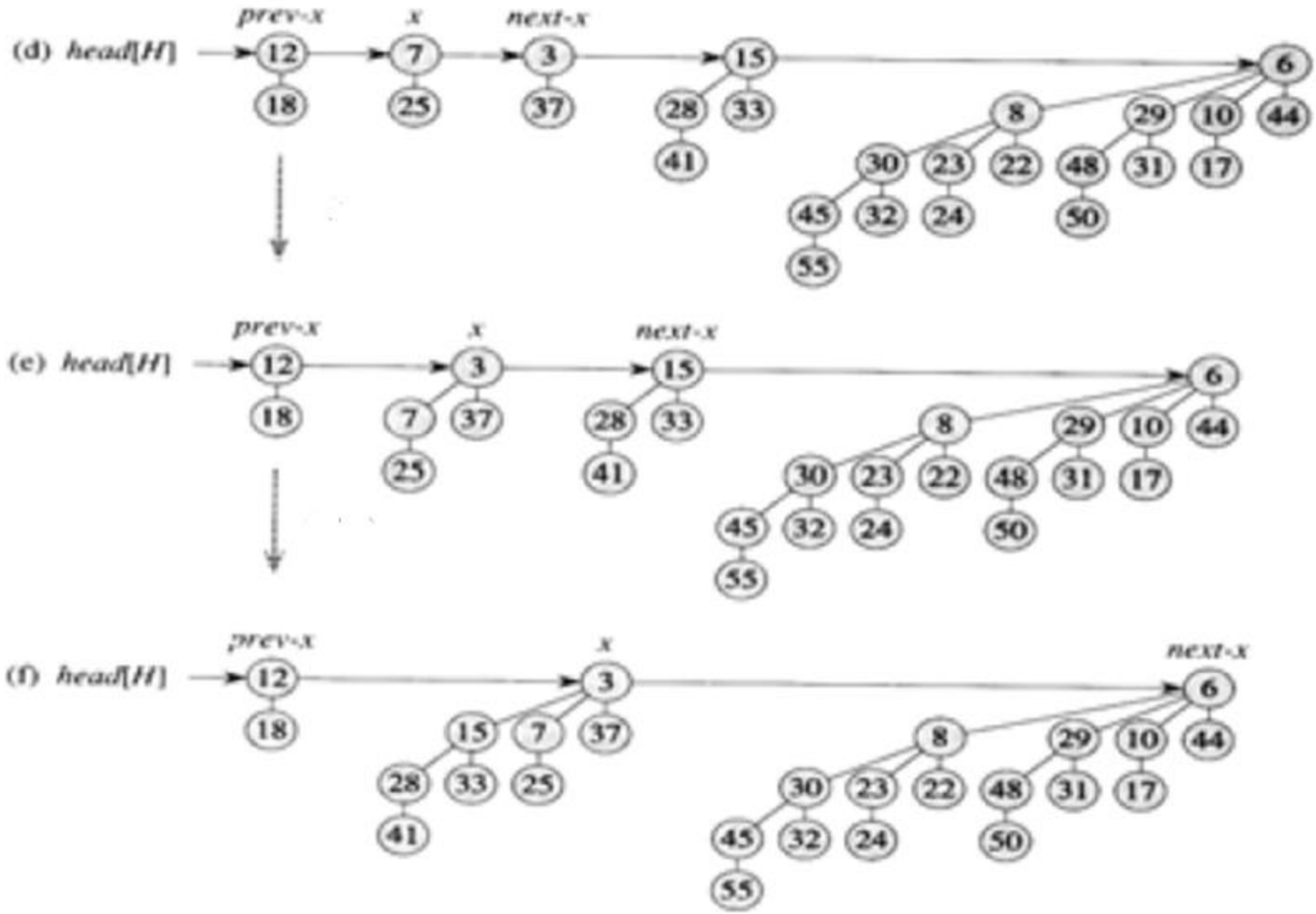
Tutte le operazioni richiedono tempo T(n) = O(log n)

Durante l'esecuzione della procedura ristruttura esistono infatti al più tre B_i, per ogni $i \geq 0$

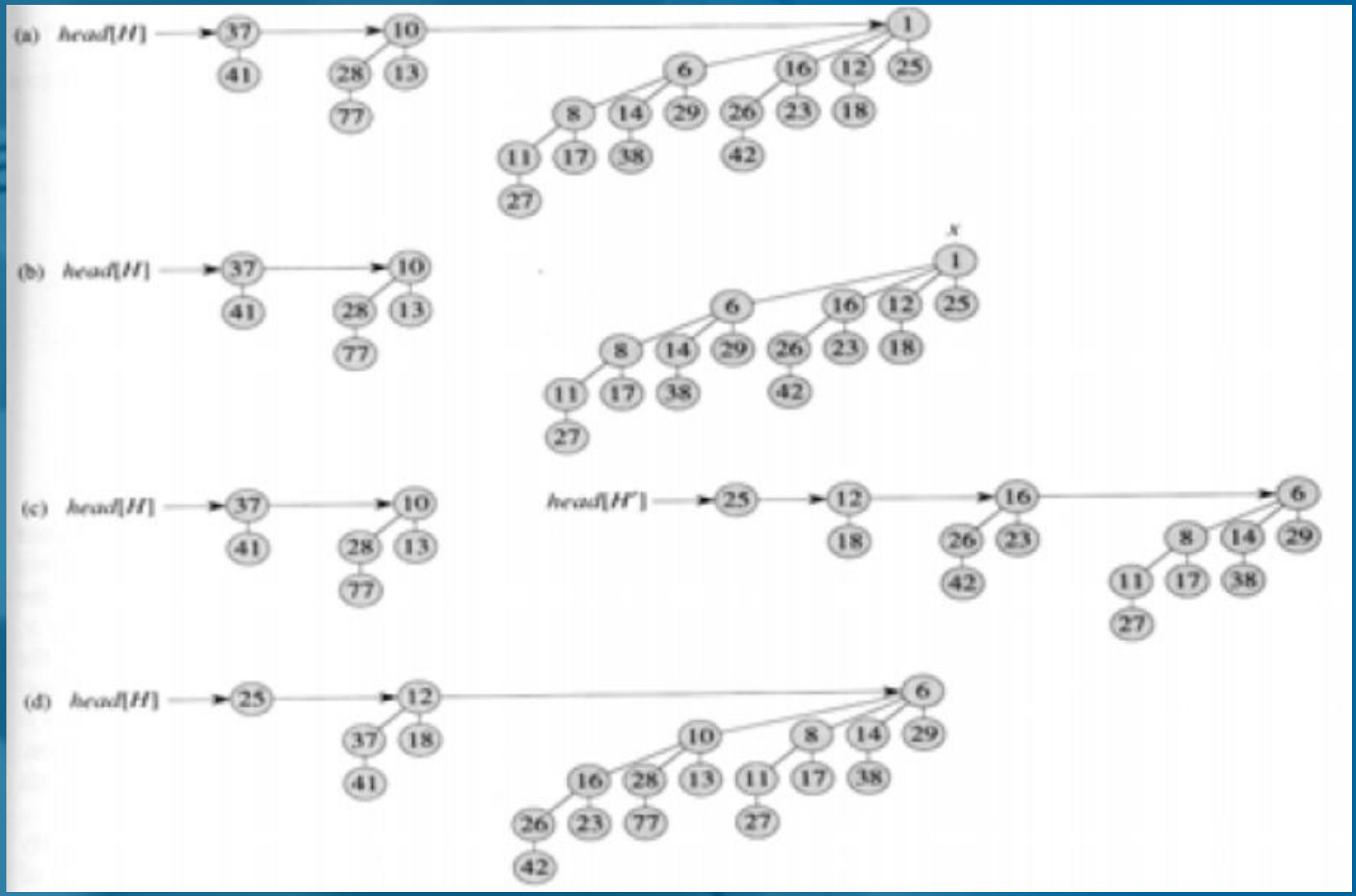
Un esempio di Merge



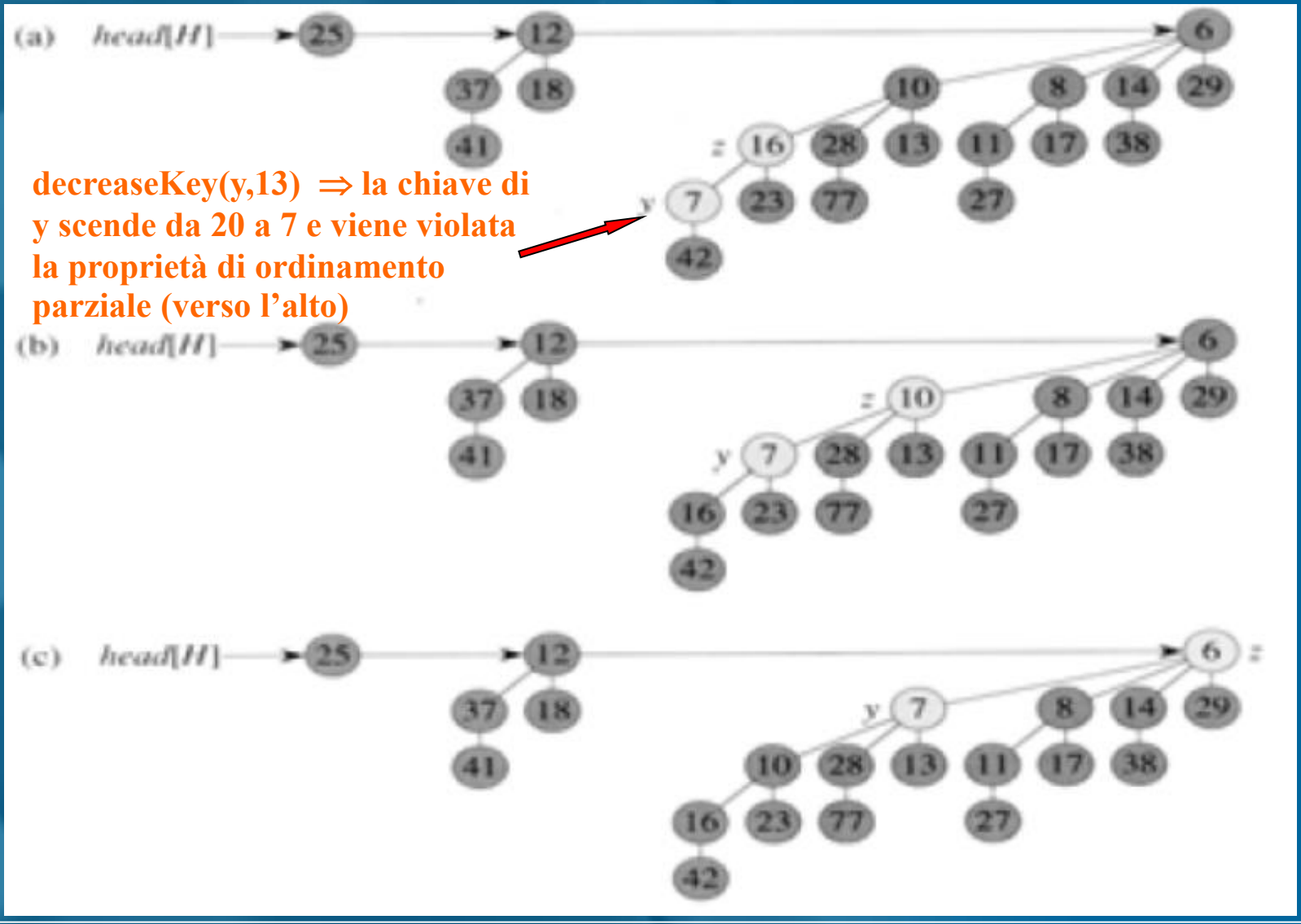
Un esempio di Merge (2)



Un esempio di deleteMin



Un esempio di decreaseKey



Heap di Fibonacci

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

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)

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?