Algoritmi e Strutture Dati

Capitolo 8

Il problema della

Coda con priorità

Tipo di dato CodaPriorità (1/2)

tipo `CodaPriorita`:
dati:

un insieme S di n elementi di tipo $elem$ a cui sono associate chiavi di tipo $chiave$ prese da un universo totalmente ordinato.

operazioni:

`findMin`$() \rightarrow elem$
restituisce l'elemento in S con la chiave minima.

`insert`$(elem\ e, chiave\ k)$
aggiunge a S un nuovo elemento e con chiave k.

`delete`$(elem\ e)$ ← **Suppongo sempre che mi venga dato un riferimento diretto all'elemento da cancellare**
cancella da S l'elemento e.

`deleteMin`$()$
cancella da S l'elemento con chiave minima.

Tipo di dato CodaPriorità (2/2)

Operazioni aggiuntive

`increaseKey`$(elem\ e, chiave\ d)$
incrementa della quantità d la chiave dell'elemento e in S.

`decreaseKey`$(elem\ e, chiave\ d)$
decrementa della quantità d la chiave dell'elemento e in S.

`merge`$(\texttt{CodaPriorita}\ c_1, \texttt{CodaPriorita}\ c_2) \rightarrow \texttt{CodaPriorita}$
restituisce una nuova coda con priorità $c_3 = c_1 \cup c_2$.

Applicazioni: gestione code in risorse condivise, gestione priorità in processi concorrenti, progettazione di algoritmi efficienti per diversi problemi fondamentali (es: calcolo cammini minimi in un grafo, minimo albero ricoprente, ordinamento, ecc.)

Quattro implementazioni elementari

1. Array non ordinato
2. Array ordinato
3. Lista non ordinata
4. Lista ordinata

Ci focalizzeremo soltanto sulle operazioni di base.

Array non ordinato

Lo dimensiono sufficientemente grande e tengo traccia del numero n di elementi nella coda in una variabile di appoggio

- FindMin: $\Theta(n)$ (devo guardare tutti gli elementi)
- Insert: $O(1)$ (inserisco in coda)
- Delete: $O(1)$ (poiché mi viene fornito il riferimento diretto all'elemento da cancellare, lo posso cancellare in $O(1)$ sovracopiando l'ultimo elemento)
- DeleteMin: $\Theta(n)$ (devo prima cercare il minimo in $\Theta(n)$, poi lo posso cancellare in $O(1)$)

Array ordinato

Lo dimensiono sufficientemente grande, lo tengo ordinato in ordine decrescente e tengo traccia del numero n di elementi nella coda in una variabile di appoggio

- FindMin: O(1)
- Insert: O(n) (trovo in Θ(log n) la giusta posizione, ma poi devo fare O(n) spostamenti)
- Delete: O(n) (devo fare O(n) spostamenti)
- DeleteMin: O(1) (l'elemento minimo è in fondo all'array, non devo fare spostamenti)

Lista non ordinata

La considero bidirezionale

- FindMin: $\Theta(n)$ (devo guardare tutti gli elementi)
- Insert: $O(1)$ (inserisco in coda o in testa)
- Delete: $O(1)$ (poiché mi viene fornito il riferimento diretto all'elemento da cancellare, lo posso cancellare in $O(1)$ agendo sui puntatori)
- DeleteMin: $\Theta(n)$ (devo prima cercare il minimo in $\Theta(n)$, poi lo posso cancellare in $O(1)$)

Lista ordinata

Tengo la lista bidirezionale ordinata in ordine crescente

- FindMin: O(1) (il minimo è in testa alla lista)
- Insert: O(n) (trovo in O(n) la giusta posizione, e poi faccio in O(1) l'inserimento)
- Delete: O(1) (agisco sui puntatori)
- DeleteMin: O(1) (basta far puntare la testa della lista al secondo elemento della lista stessa)

Riepilogo implementazioni elementari

	FindMin	Insert	Delete	DeleteMin
Array non ord.	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$
Array ordinato	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Lista non ordinata	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$
Lista ordinata	$O(1)$	$O(n)$	$O(1)$	$O(1)$

Tre implementazioni evolute

- d-heap: generalizzazione degli heap binari visti per l'ordinamento
- Heap binomiali
- Heap di Fibonacci (cenni)


d-heap

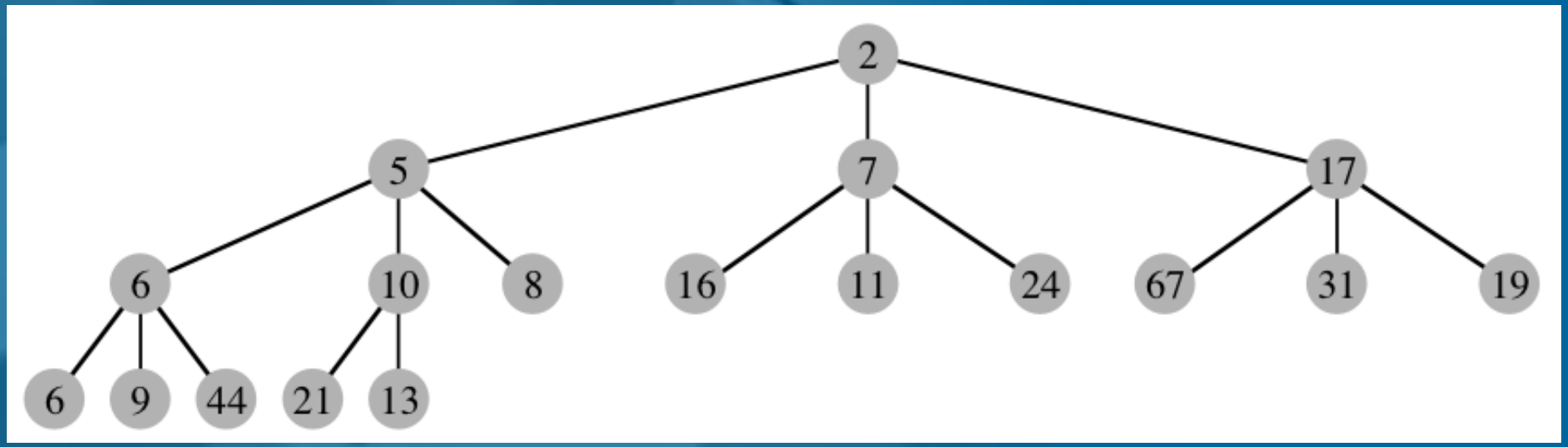
Definizione

Un d-heap è un albero radicato d-ario con le seguenti proprietà:

1. Struttura: è completo almeno fino al penultimo livello, e tutte le foglie sull'ultimo livello sono compattate verso sinistra
2. Contenuto informativo: ogni nodo v contiene un elemento elem(v) ed una chiave chiave(v) presa da un dominio totalmente ordinato
3. Ordinamento parziale (inverso) dell'heap (min-heap): chiave(v) ≥ chiave(parent(v)) per ogni nodo v diverso dalla radice

Esempio

Heap d-ario con 18 nodi e d=3



Proprietà

1. Un d-heap con n nodi ha altezza $\Theta(\log_d n)$
2. La radice contiene l'elemento con chiave minima (per via della proprietà di ordinamento a heap)
3. Può essere rappresentato implicitamente tramite vettore posizionale grazie alla proprietà di struttura

Procedure ausiliarie

Utili per ripristinare la proprietà di ordinamento a heap su un nodo v che non la soddisfi

$T(n)=O(\log_d n)$

procedura `muoviAlto`(v)
  while ($v \neq radice(T)$ **and** $chiave(v) < chiave(padre(v))$) **do**
    scambia di posto v e $padre(v)$ in T

il vecchio amico FixHeap!

$T(n)=O(d \log_d n)$

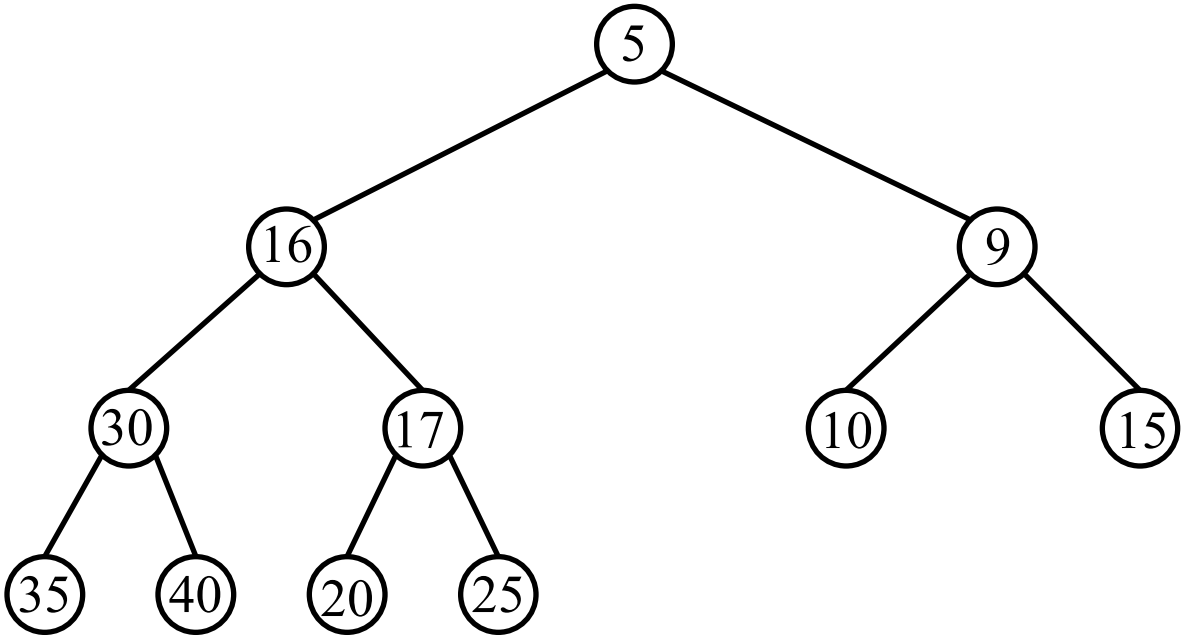
procedura `muoviBasso`(v)
  repeat
    sia u il figlio di v con la minima $chiave(u)$, se esiste
    if (v non ha figli o $chiave(v) \leq chiave(u)$) **break**
    scambia di posto v e u in T

findMin

`findMin`$() \rightarrow elem$
restituisce l'elemento nella radice di T.

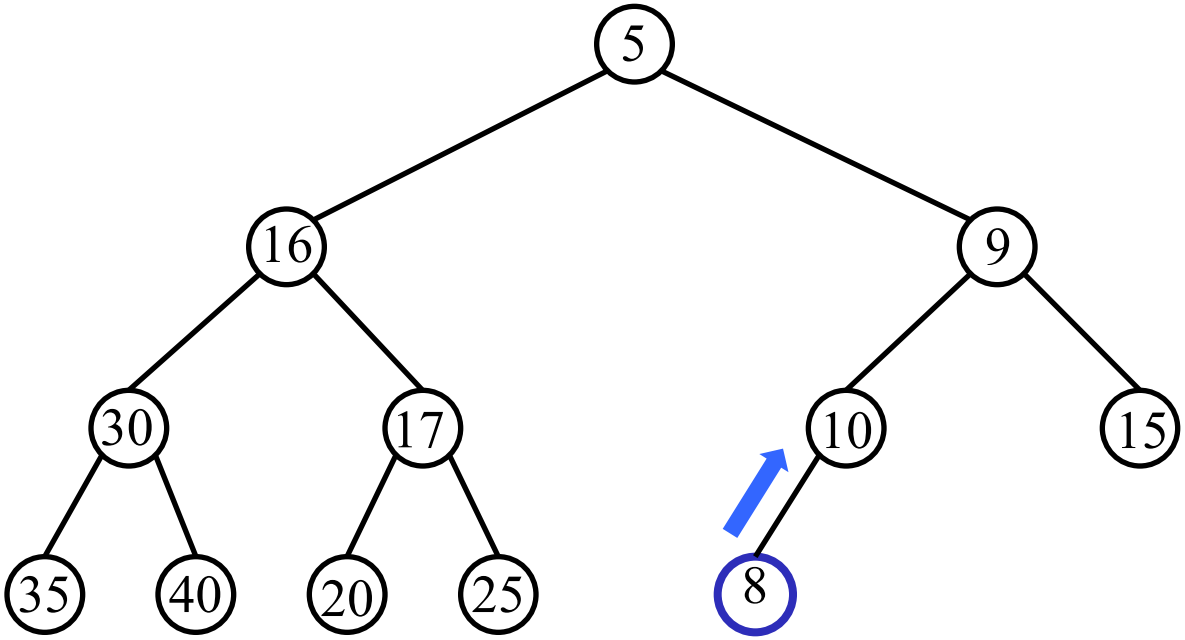
T(n)=O(1)

insert(elem e, chiave k)



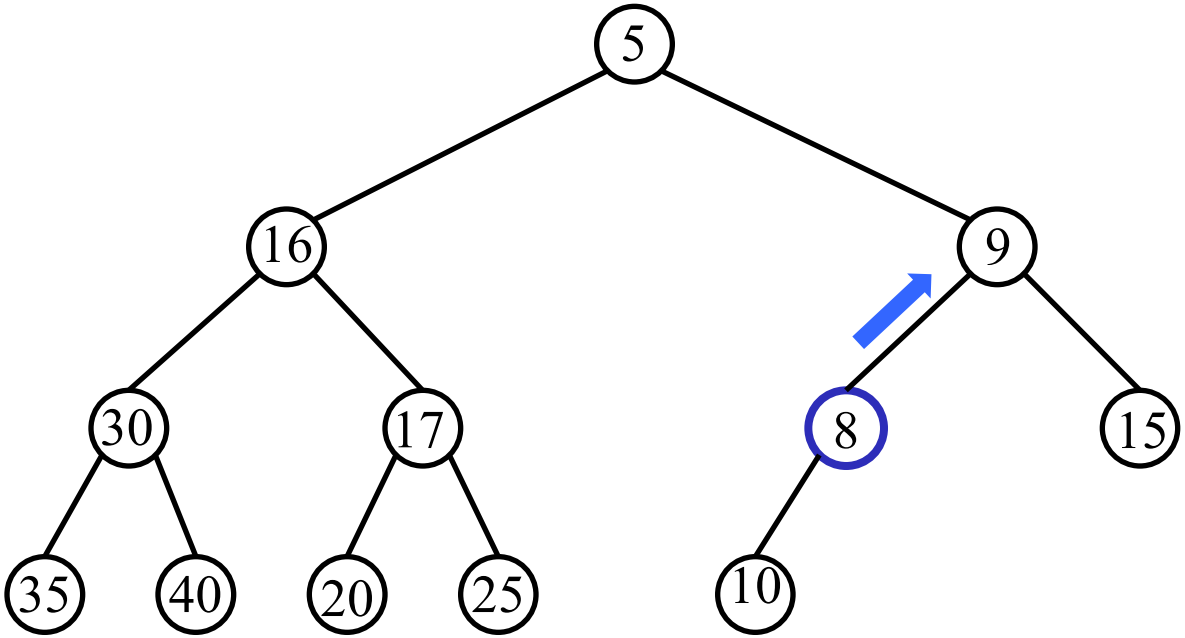
Insert(e,8)

insert(elem e, chiave k)



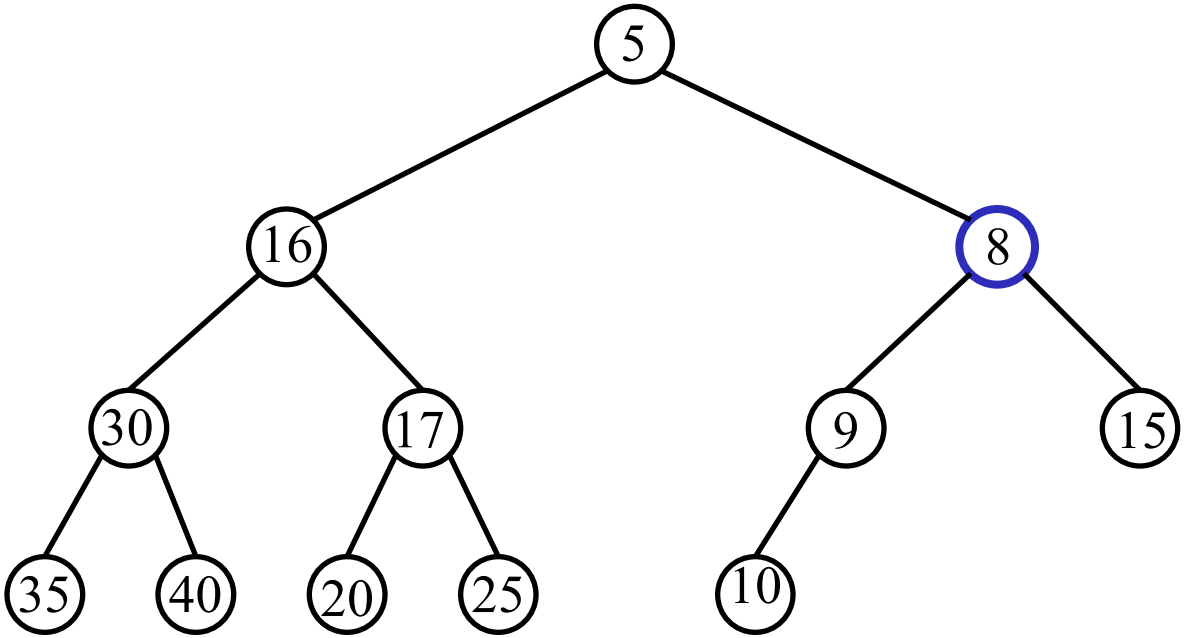
Insert(e,8)

insert(elem e, chiave k)



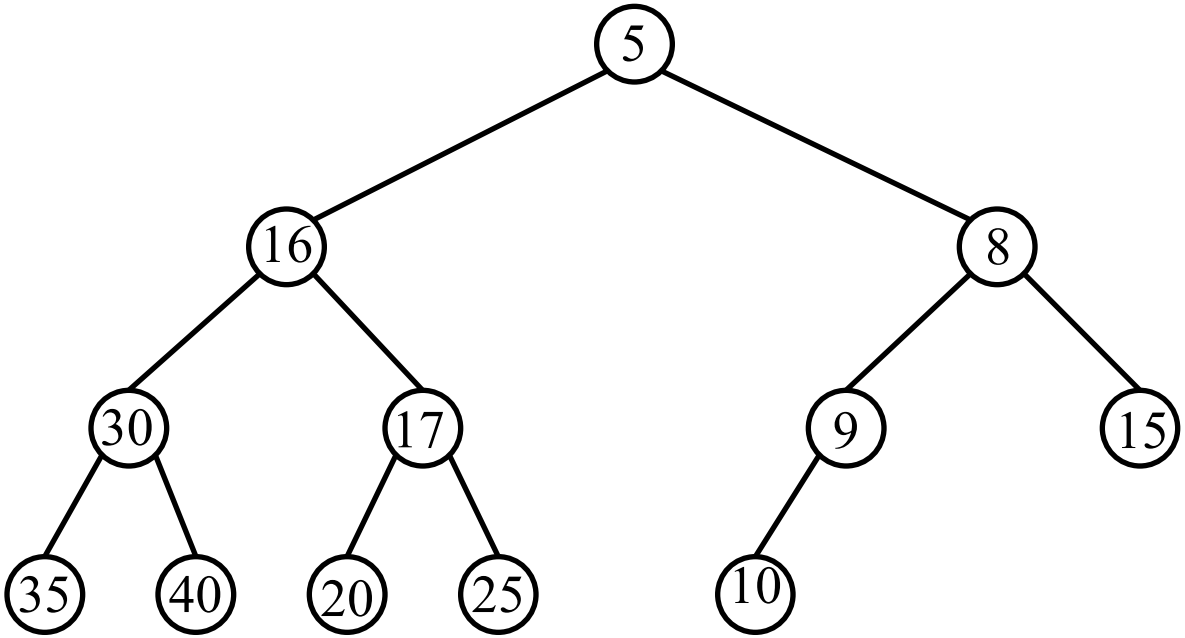
Insert(e,8)

insert(elem e, chiave k)



Insert(e,8)

insert(elem e, chiave k)



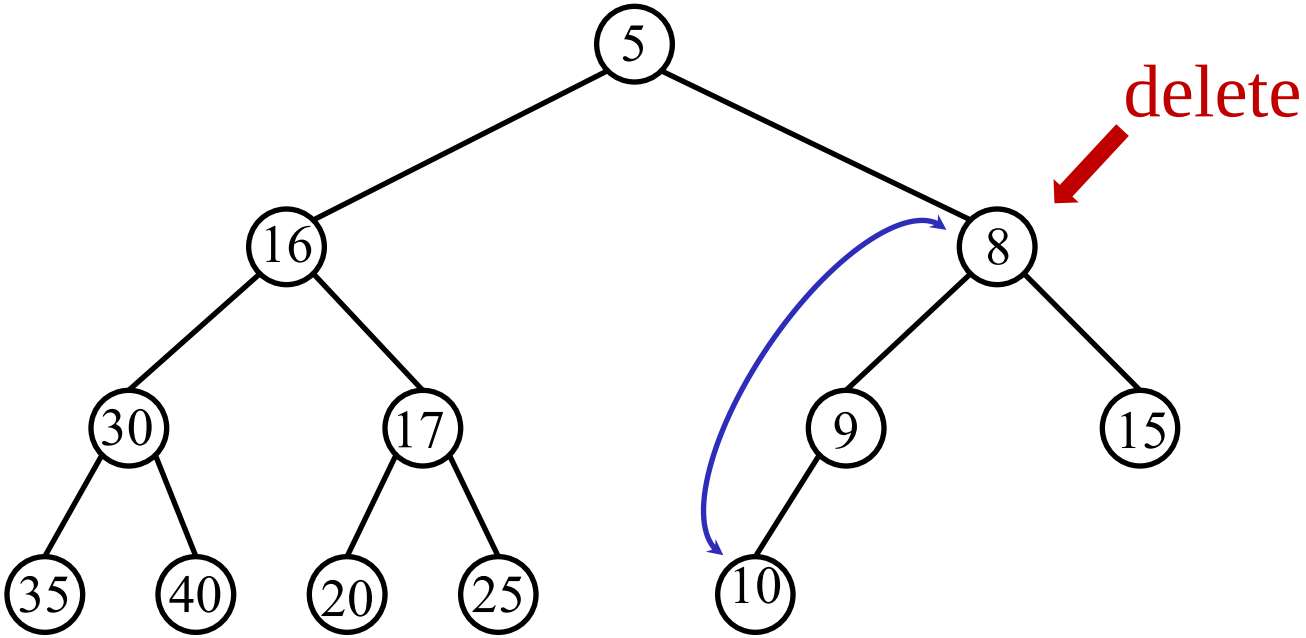
Insert(e,8)

insert(elem e, chiave k)

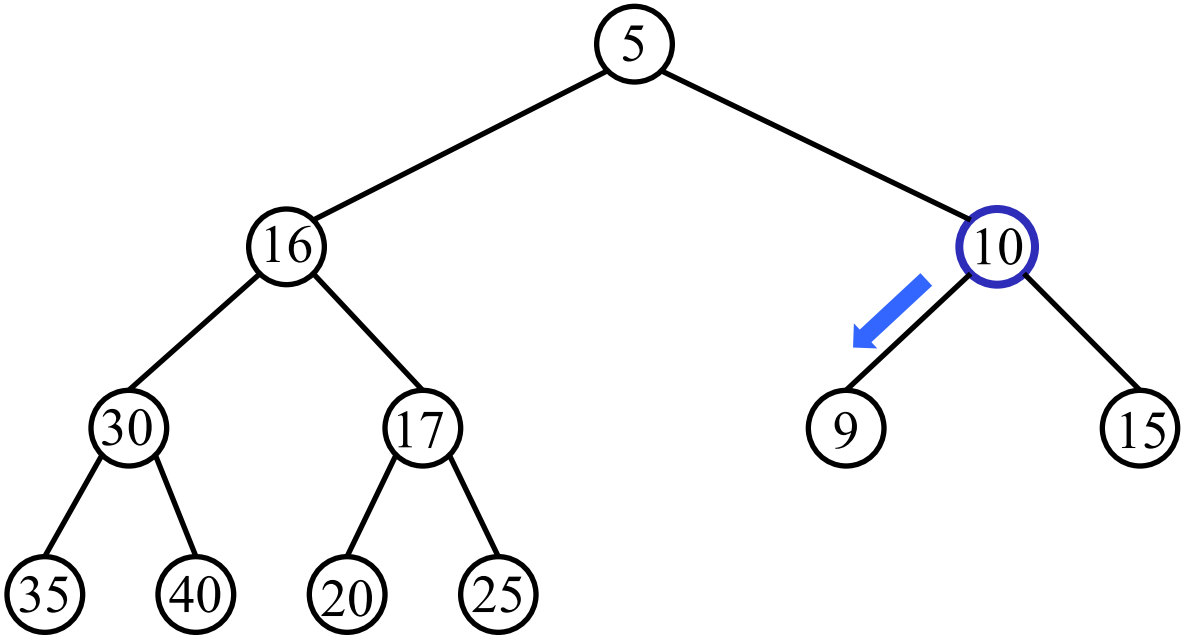
crea un nuovo nodo v con elemento e e chiave k, in modo che diventi una foglia sull'ultimo livello di T. La proprietà dell'ordinamento a heap viene poi ripristinata spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi.

$T(n)=O(\log_d n)$ per l'esecuzione di muoviAlto

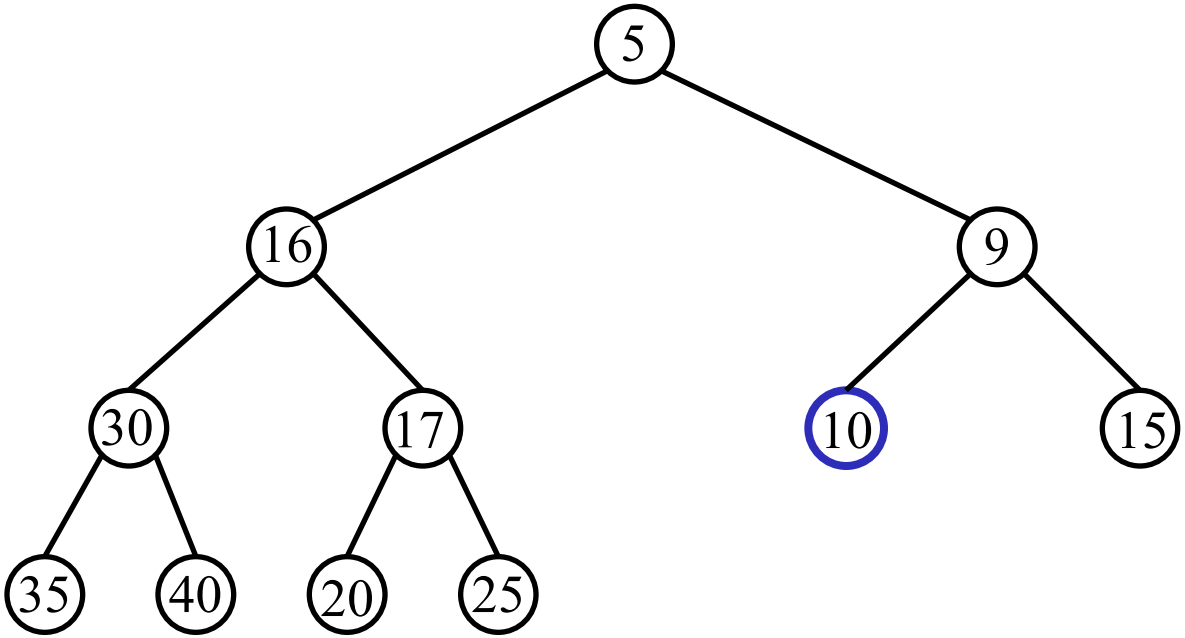
delete(elem e) e deleteMin



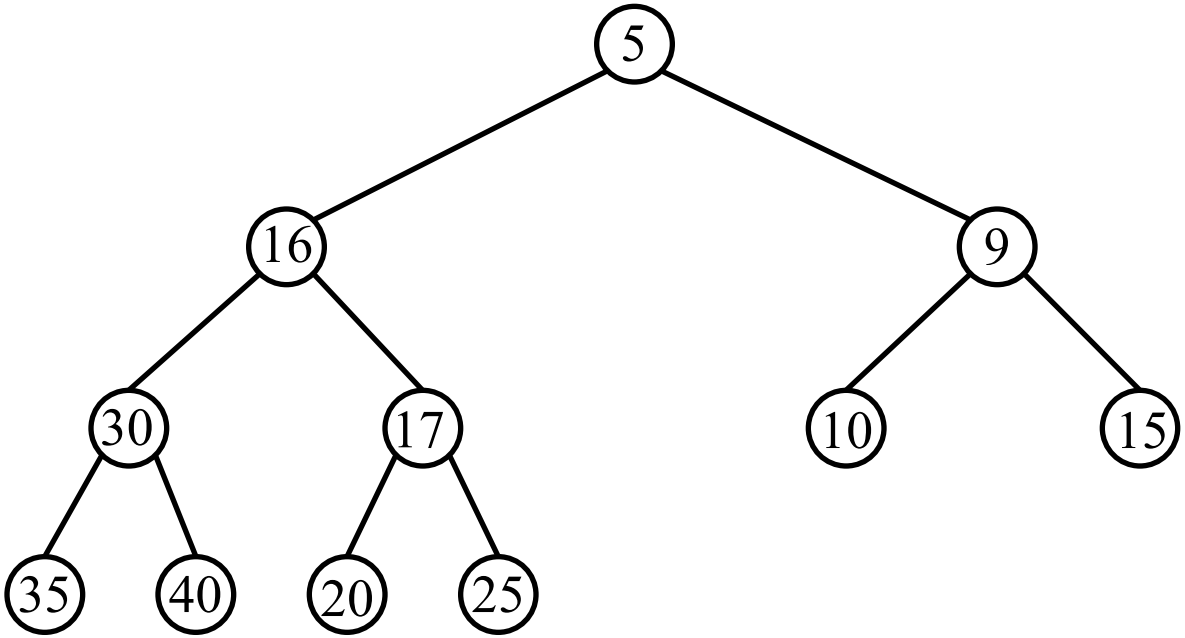
delete(elem e) e deleteMin



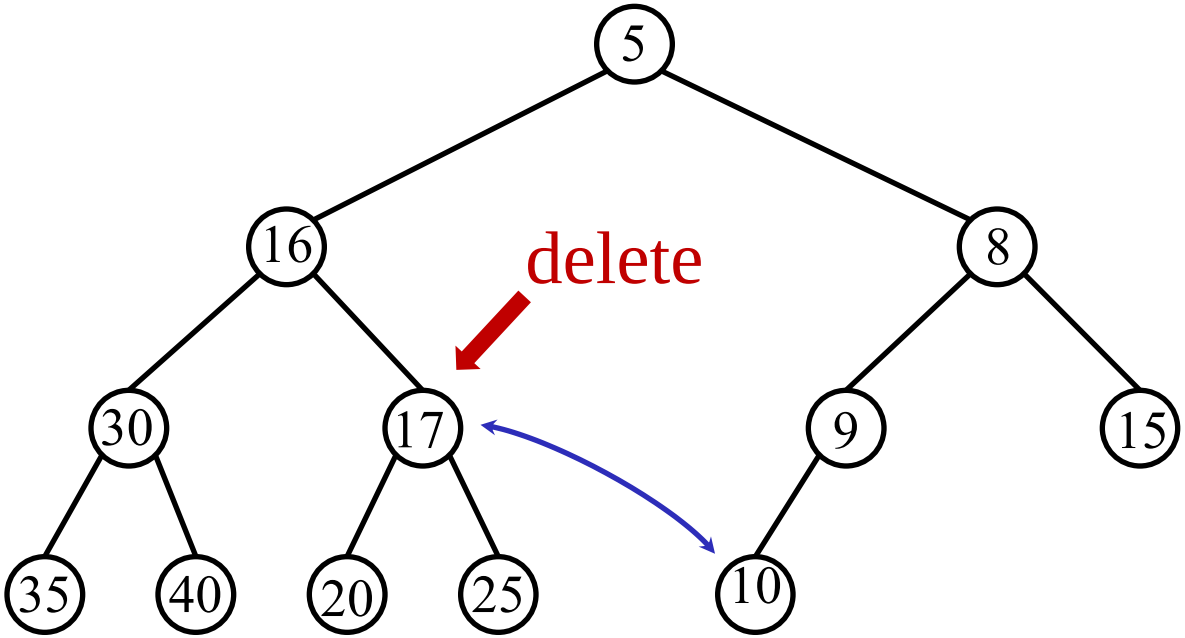
delete(elem e) e deleteMin



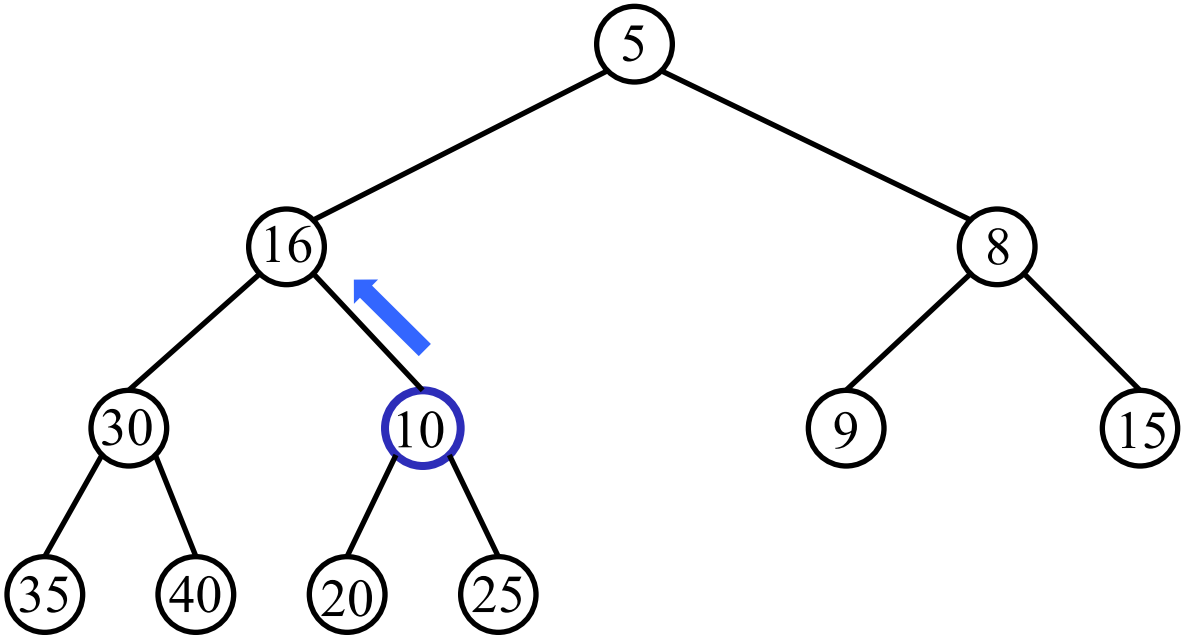
delete(elem e) e deleteMin



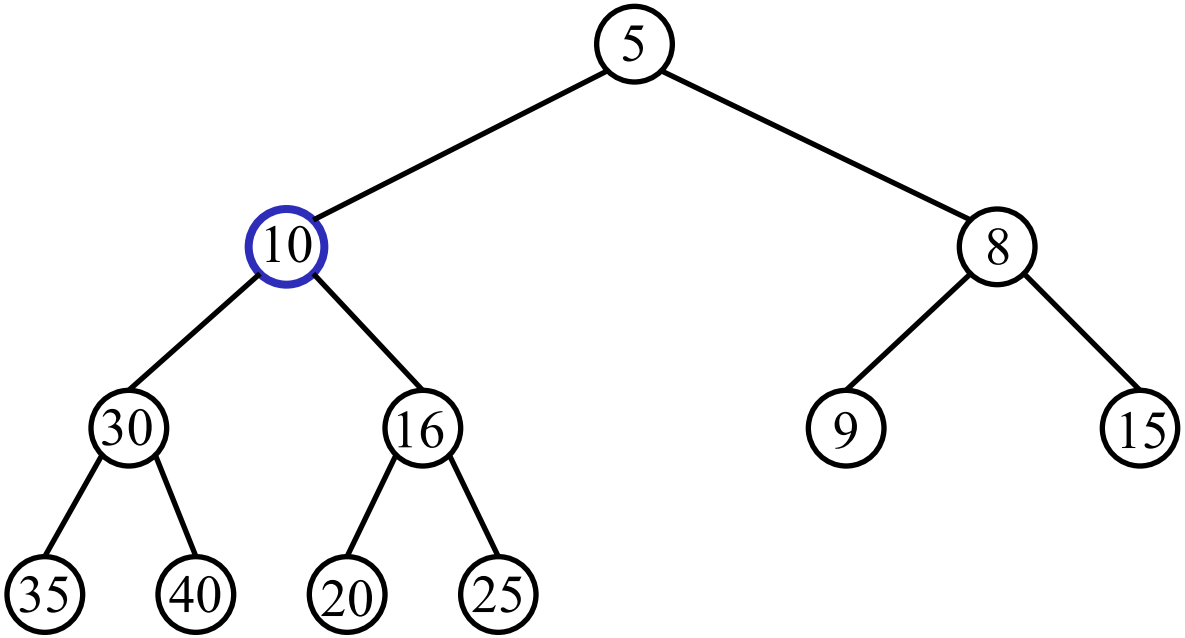
delete(elem e) e deleteMin



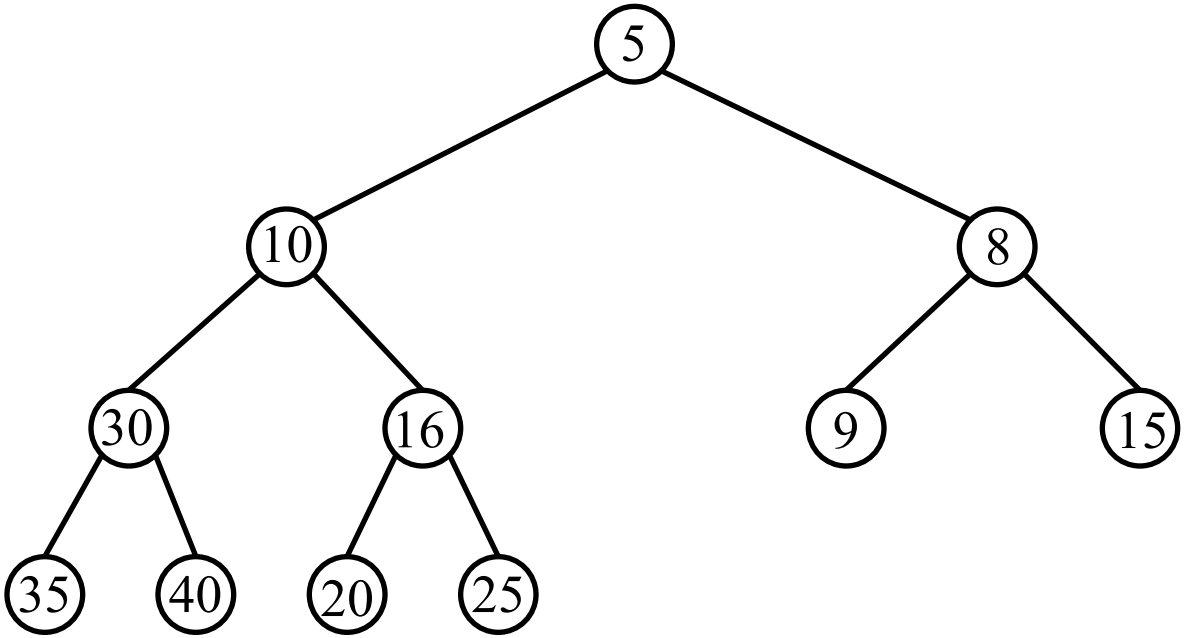
delete(elem e) e deleteMin



delete(elem e) e deleteMin



delete(elem e) e deleteMin



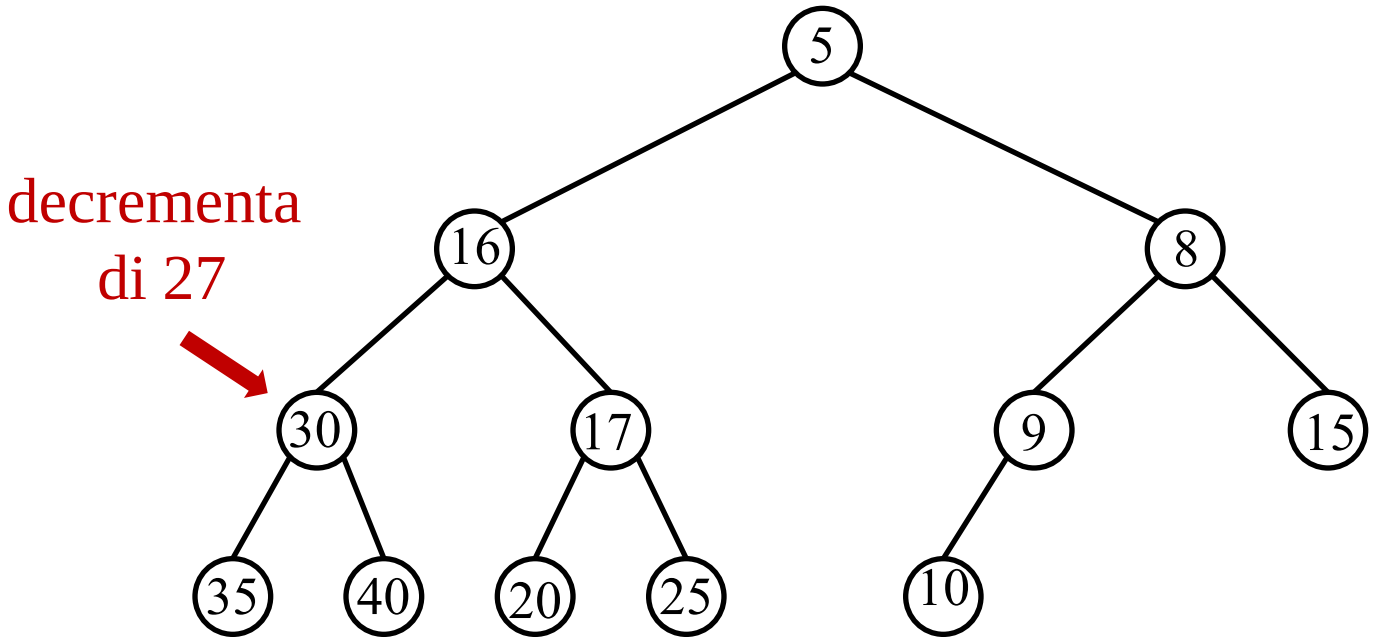
delete(elem e) e deleteMin

scambia il nodo v contenente l'elemento e con una qualunque foglia u sull'ultimo livello di T, e poi elimina v. Ripristina infine la proprietà dell'ordinamento a heap spingendo il nodo u verso la sua posizione corretta scambiandolo ripetutamente con il proprio padre o con il proprio figlio contenente la chiave più piccola

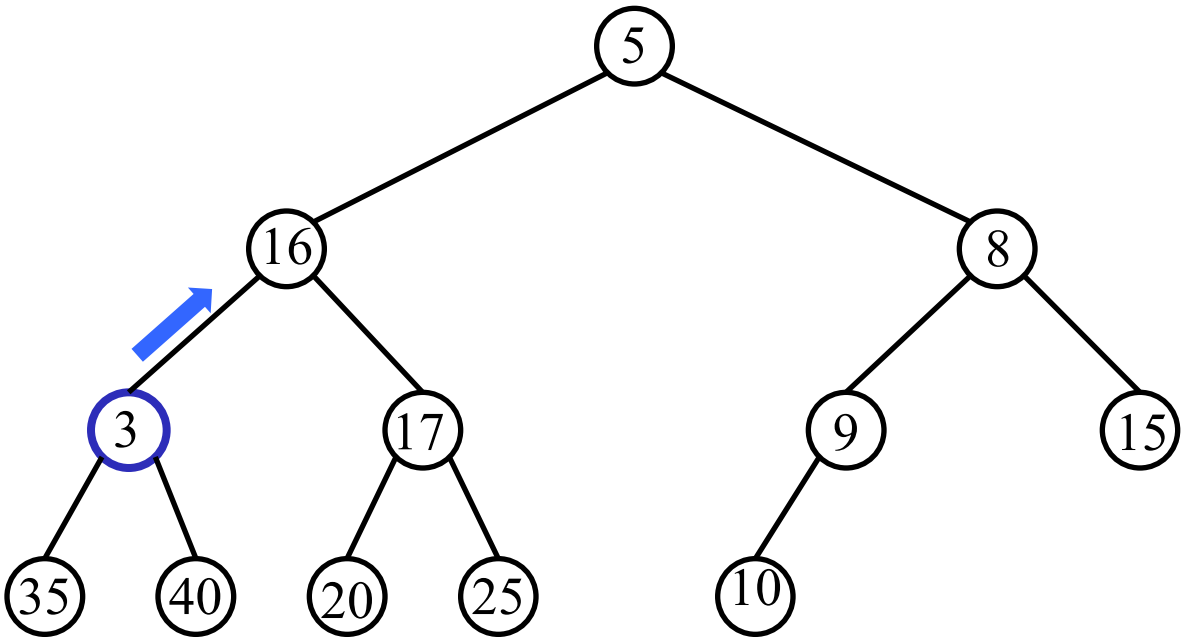
$T(n)= O(\log_d n)$ o $O(d \log_d n)$ per l'esecuzione di muoviAlto o muoviBasso

Può essere usata anche per implementare la cancellazione del minimo, con costo $O(d \log_d n)$

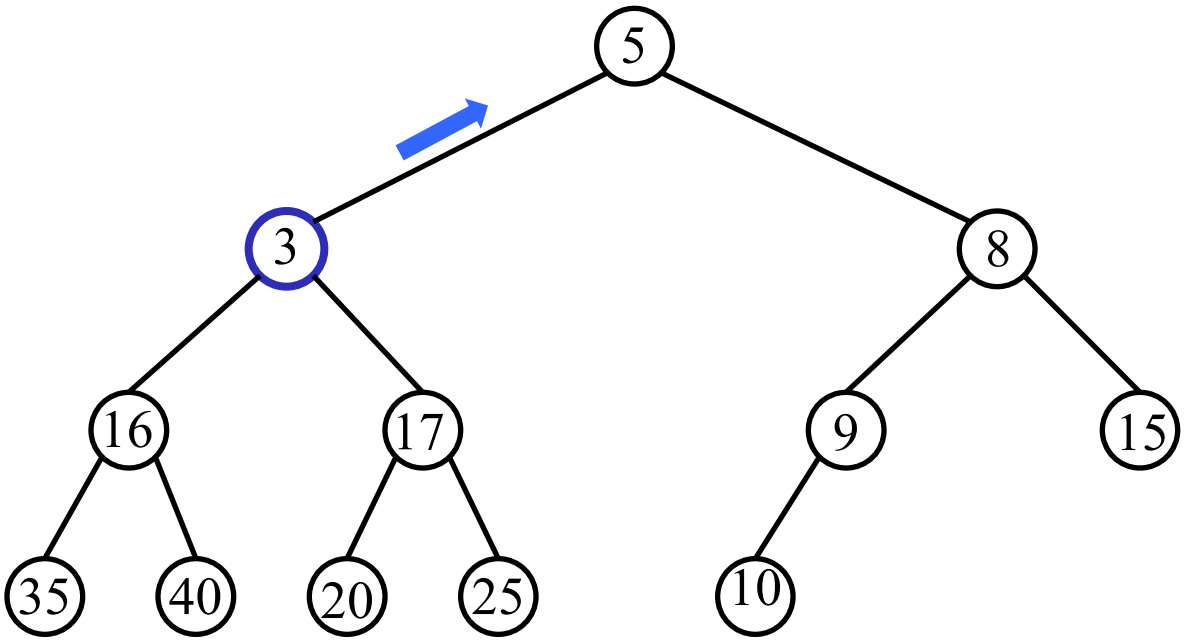
decreaseKey(elem e, chiave d)



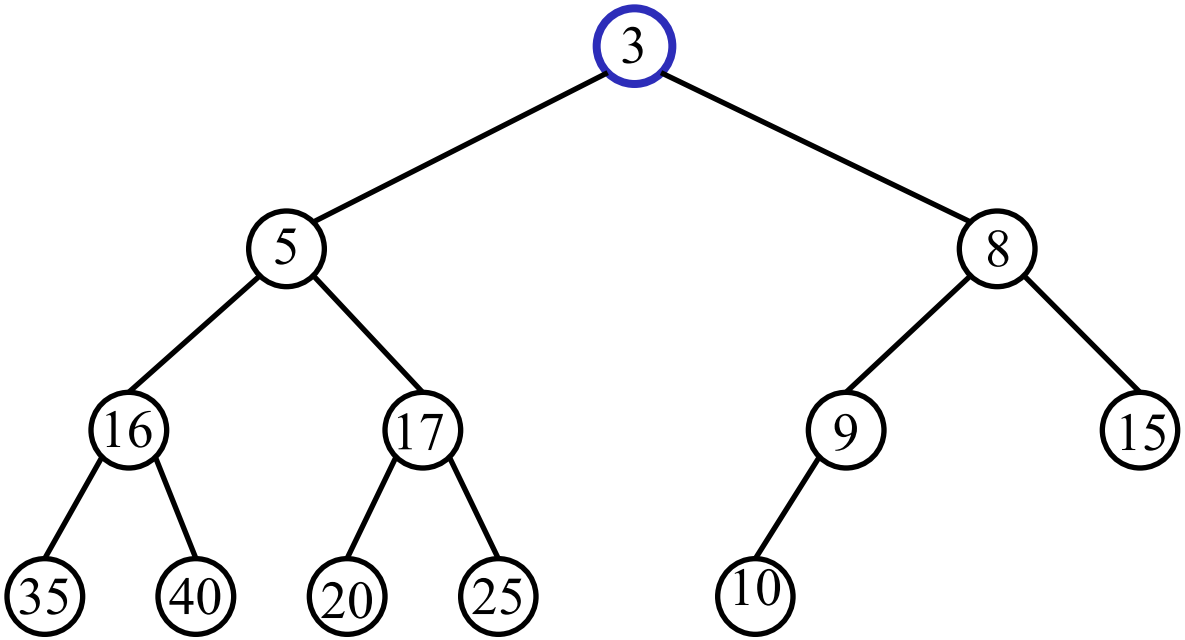
decreaseKey(elem e, chiave d)



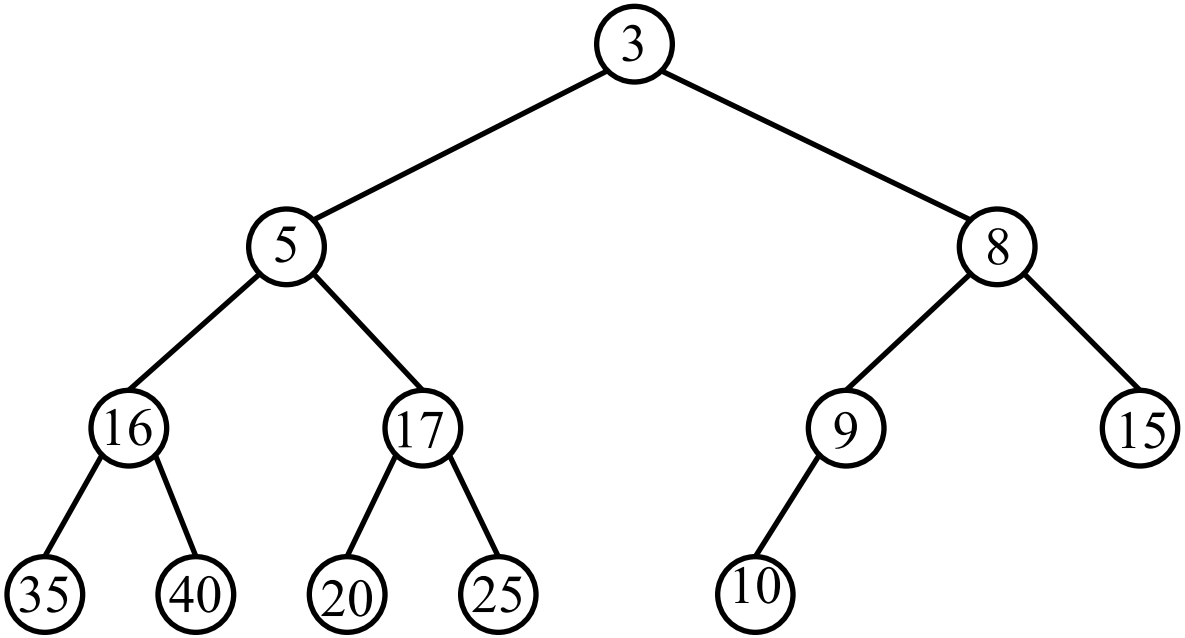
decreaseKey(elem e, chiave d)



decreaseKey(elem e, chiave d)



decreaseKey(elem e, chiave d)

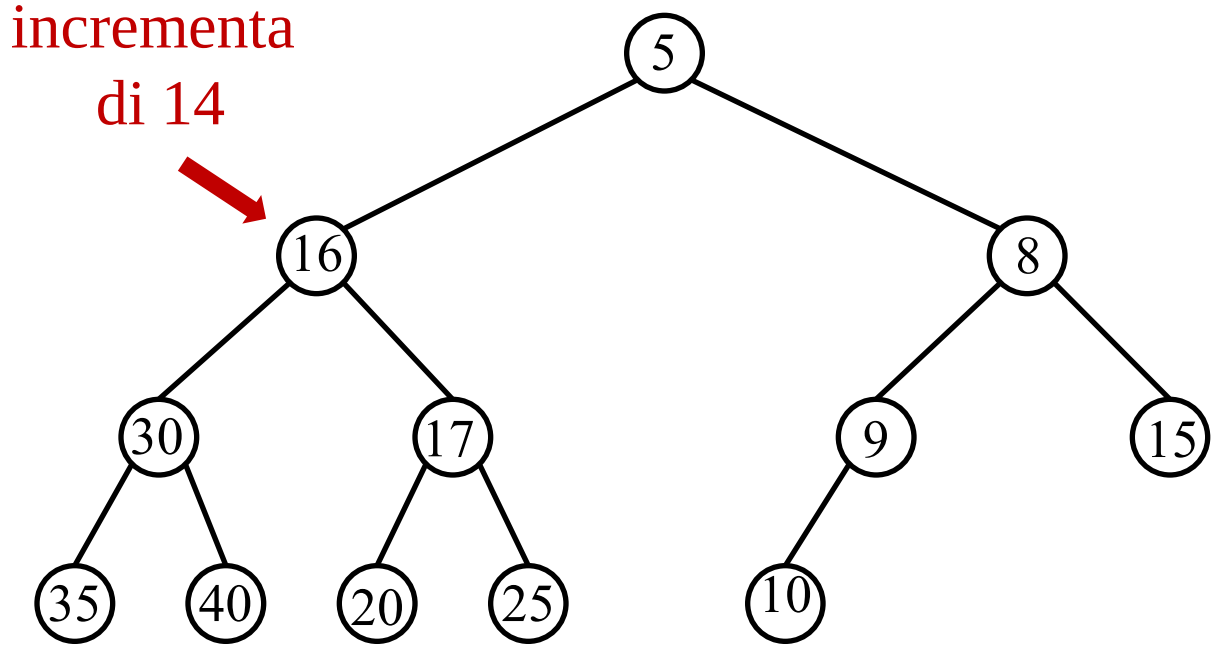


decreaseKey(elem e, chiave d)

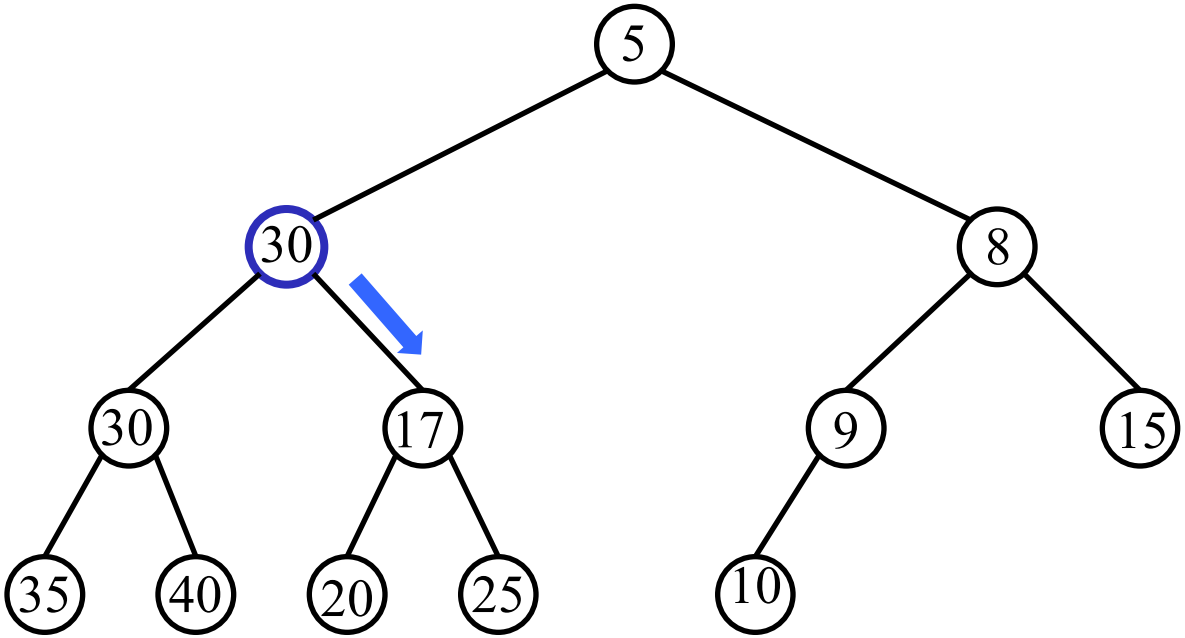
decrementa il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta d. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi.

$T(n)=O(\log_d n)$ per l'esecuzione di muoviAlto

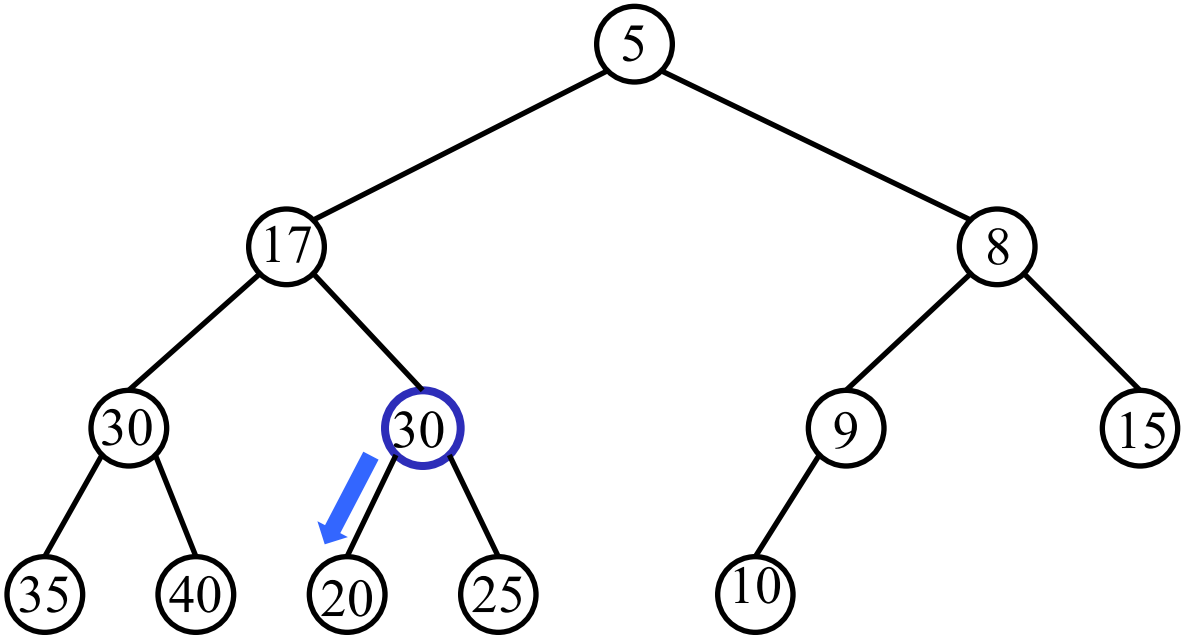
increaseKey(elem e, chiave d)



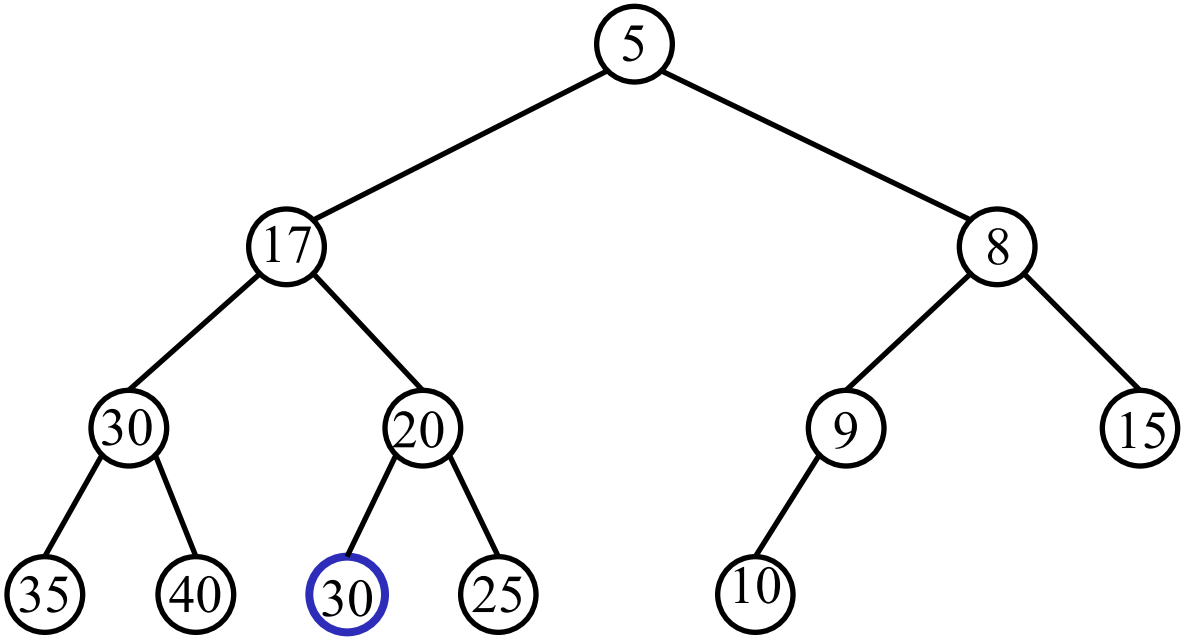
increaseKey(elem e, chiave d)



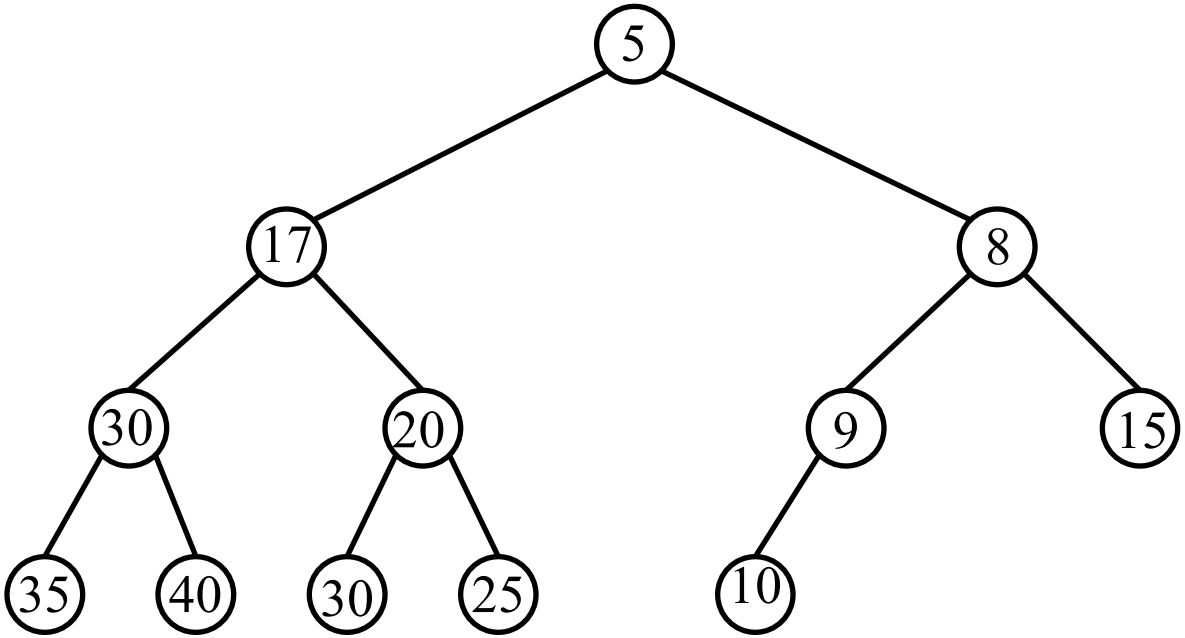
increaseKey(elem e, chiave d)



increaseKey(elem e, chiave d)



increaseKey(elem e, chiave d)



increaseKey(elem e, chiave d)

aumenta il valore della chiave nel nodo contenente l'elemento e della quantità richiesta d. Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso il basso tramite ripetuti scambi di nodi.

$T(n)=O(d \log_d n)$ per l'esecuzione di muoviBasso

merge(CodaPriorità c_1, CodaPriorità c_2)

Due modi:

1. Costruire da zero: si distruggono le due code e se ne crea una nuova contenente l'unione degli elementi.
2. Inserimenti ripetuti: si inseriscono ripetutamente gli elementi della coda più piccola in quella più grande.

Costruire da zero

Analogamente a quanto mostrato per l'heap binario, la creazione di un heap d-ario (con d costante) di n elementi può essere eseguita in $\Theta(n)$. Infatti, il tempo di esecuzione di heapify diventa ora:

$$T(n)= d\,T(n/d)+O(d \log_d n)$$

ove il fattore $O(d \log_d n)$ è relativo all'esecuzione della procedura muoviBasso (`fixheap` nell'heap binario).

Siamo quindi di nuovo nel Caso 1 del Teorema Master:

$d \log_d n = f(n) = O(n^{\log_d d - \varepsilon})$ per $\varepsilon>0$, e quindi $T(n) = \Theta(n^{\log_d d}) = \Theta(n)$

$\Rightarrow$ Il merge viene quindi eseguito in $\Theta(n)$, ove $n=|c_1|+|c_2|$, generando un nuovo heap d-ario che contiene tutti gli elementi in c_1 e c_2

Inserimenti ripetuti

Inseriamo ad uno ad uno tutti gli elementi della coda più piccola nella coda più grande.

Sia $k=\min\{|c_1|,|c_2|\}$ e $n=|c_1|+|c_2|$.

Eseguiamo quindi k inserimenti nella coda più grande.

Costo: $O(k \log n)$, dove $n=|c_1|+|c_2|$.

L'approccio conviene quindi per $k \log n=o(n)$, cioè per

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

merge(CodaPriorità c_1, CodaPriorità c_2)

Due modi:

1. Costruire da zero: si distruggono le due code e se ne crea una nuova contenente l'unione degli elementi.
2. Inserimenti ripetuti: si inseriscono ripetutamente gli elementi della coda più piccola in quella più grande.

Osservazione: nel caso peggiore entrambe le operazioni hanno un costo di $\Omega(n)$.

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.	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$	$O(1)$	$O(1)$	$O(n)$
Array ordinato	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
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	$O(1)$	$O(\log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(n)$

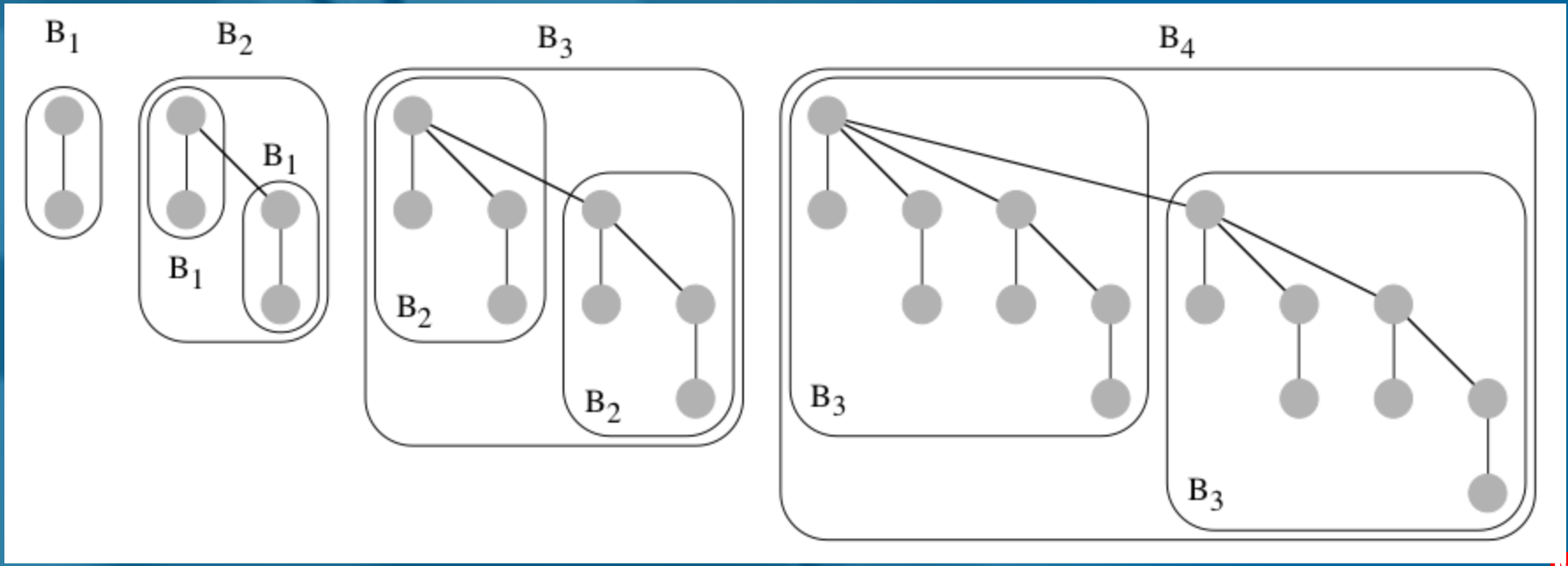
$\Rightarrow$ Il nostro obiettivo è implementare una coda di priorità con una struttura dati che non comporti costi lineari!

heap binomiali

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

Dimostrazione per induzione su *h*

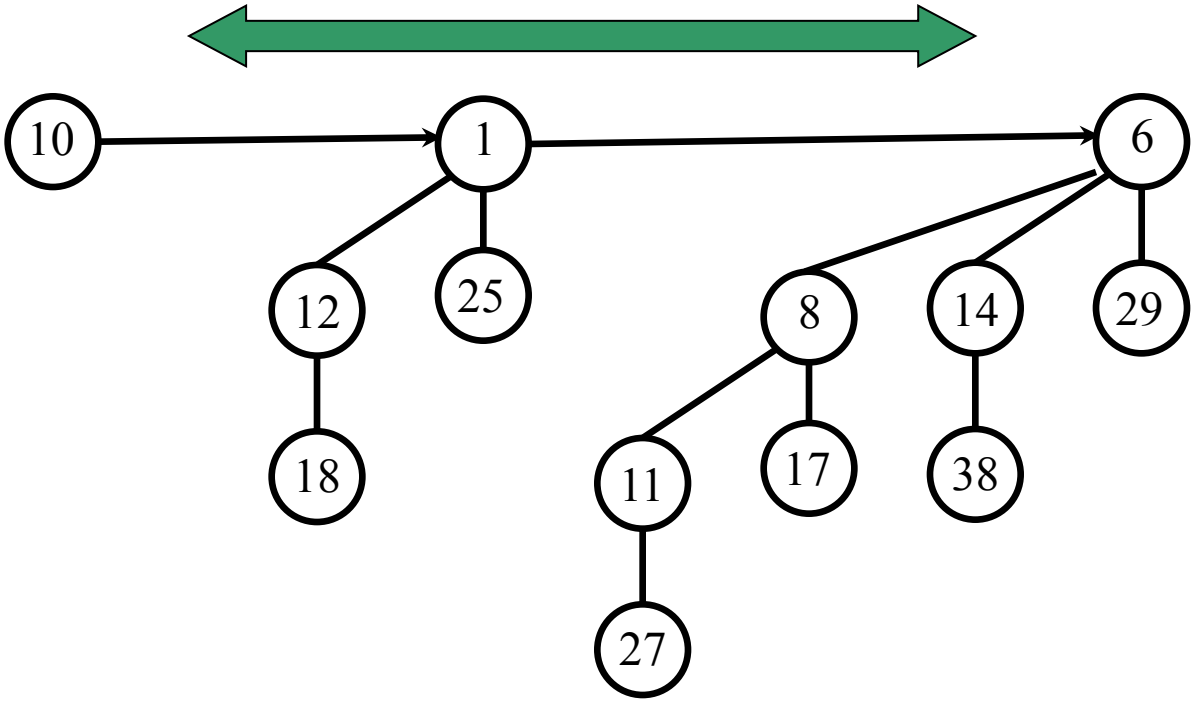
Definizione di heap binomiale

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: ogni nodo v contiene un elemento elem(v) ed una chiave chiave(v) presa da un dominio totalmente ordinato
3. Ordinamento a heap: chiave(v) ≥ chiave(parent(v)) per ogni nodo v diverso da una delle radici

Un esempio di Heap Binomiale con n=13 nodi

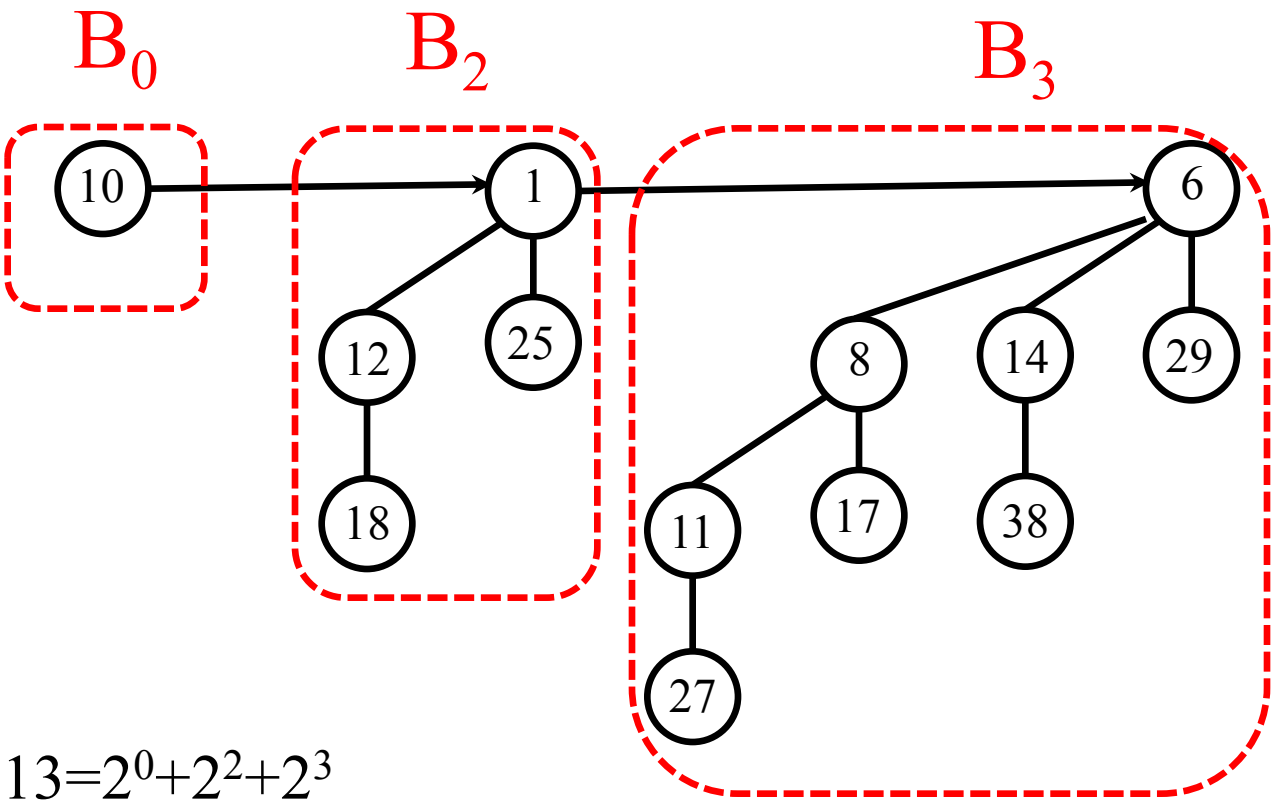
in questa direzione non è presente un ordinamento



in questa direzione è presente un ordinamento

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

Un esempio di Heap Binomiale con n=13 nodi



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

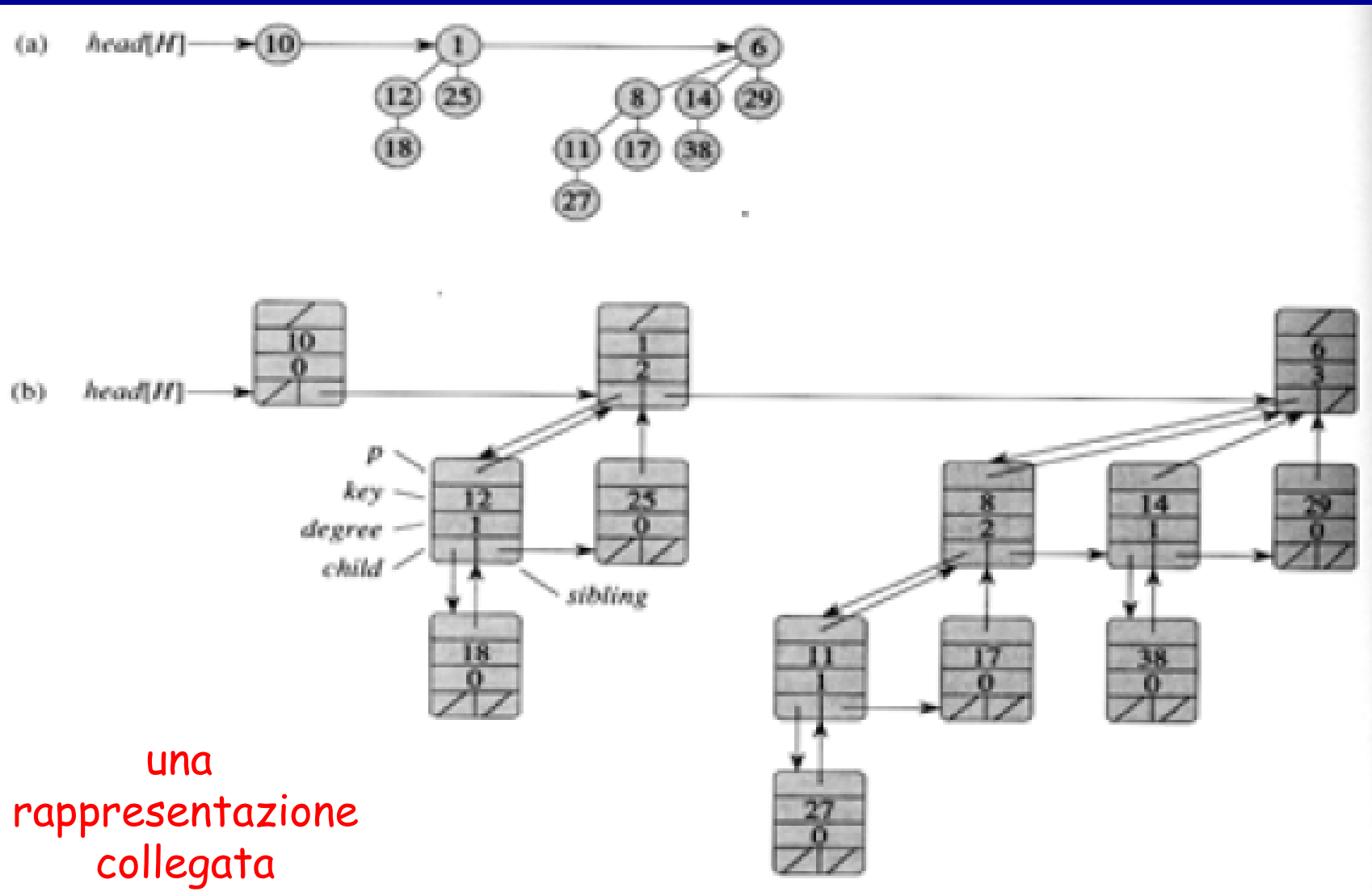
13 in binario: 1101

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 $O(\log n)$

A Binomial Heap



una rappresentazione collegata

Procedura ausiliaria

Utile per ripristinare la proprietà di unicità in un heap binomiale (ipotizziamo di scorrere la lista delle radici da sinistra verso destra, in ordine crescente rispetto all'indice degli alberi binomiali)

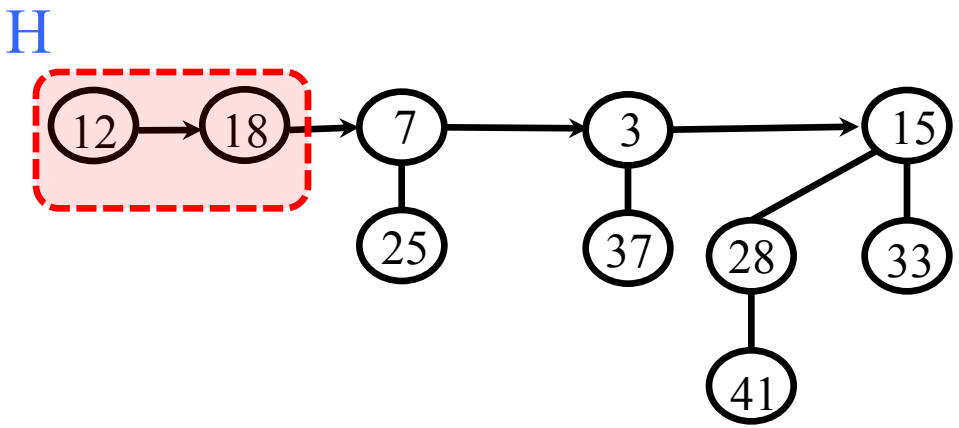
procedura `ristruttura`()
  $i = 0$
  while (esistono ancora due B_i) **do**
    si fondono i due B_i per formare un albero B_{i+1}, ponendo la radice con chiave più piccola come genitore della radice con chiave più grande
    $i = i + 1$

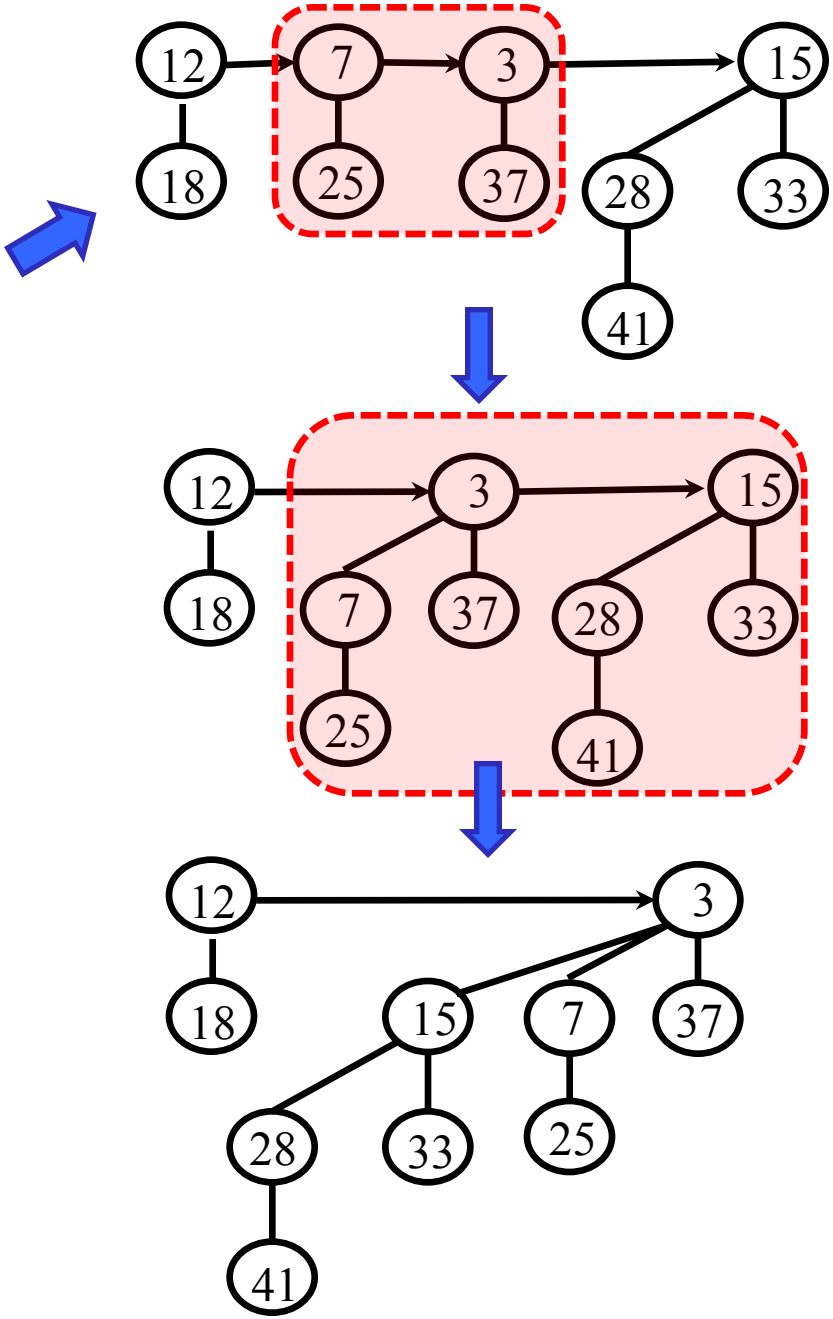
T(n): lineare nel numero di alberi binomiali in input

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

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.

`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`$(elem\ e, chiave\ d)$

decrementa di d 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`$(e, -\infty)$ e poi `deleteMin`$()$.

Realizzazione (3/3)

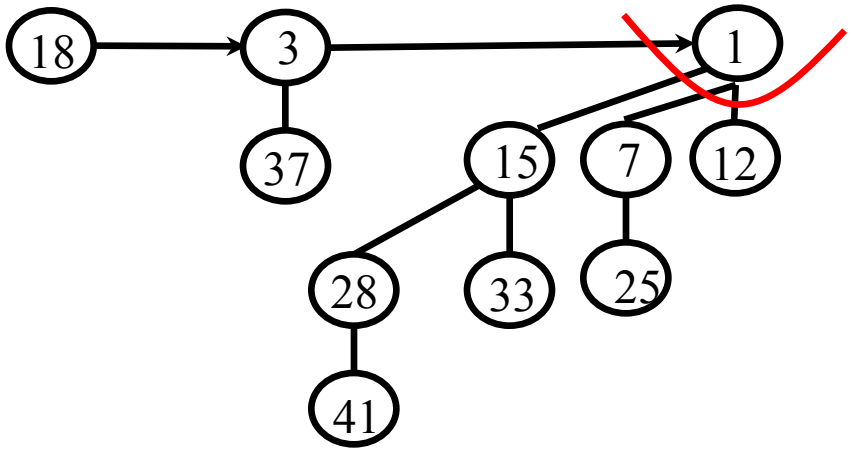
`increaseKey`$(elem\ e, chiave\ d)$
richiama `delete`(e) e poi `insert`$(elem, k + d)$, dove k è la chiave associata all'elemento e.

`merge`$(\texttt{CodaPri}.\ c_1, \texttt{CodaPri}.\ c_2) \rightarrow \texttt{CodaPri}.$
unisce gli alberi in c_1 e c_2 in un nuovo heap binomiale c_3. Ripristina poi la proprietà di unicità nell'heap binomiale c_3 mediante fusioni successive dei doppioni B_i.

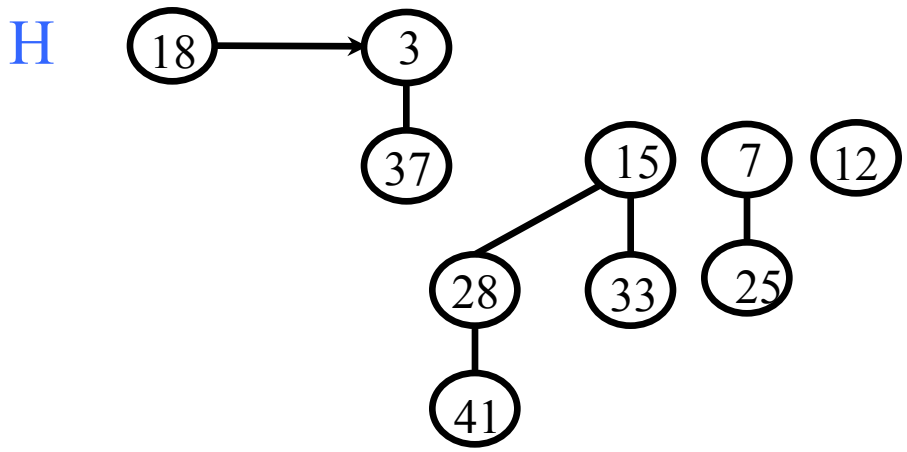
Tutte le operazioni richiedono tempo T(n) = O(log n)
Durante l'esecuzione della procedura ristruttura esistono infatti al più tre B_i, per ogni $i \geq 0$

DeleteMin(H)

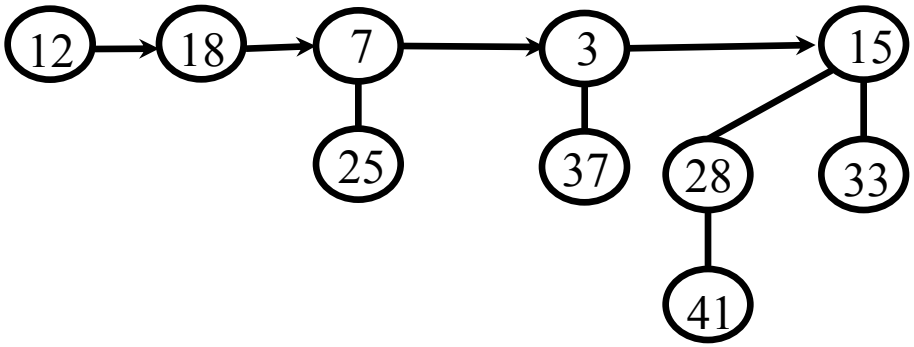
H



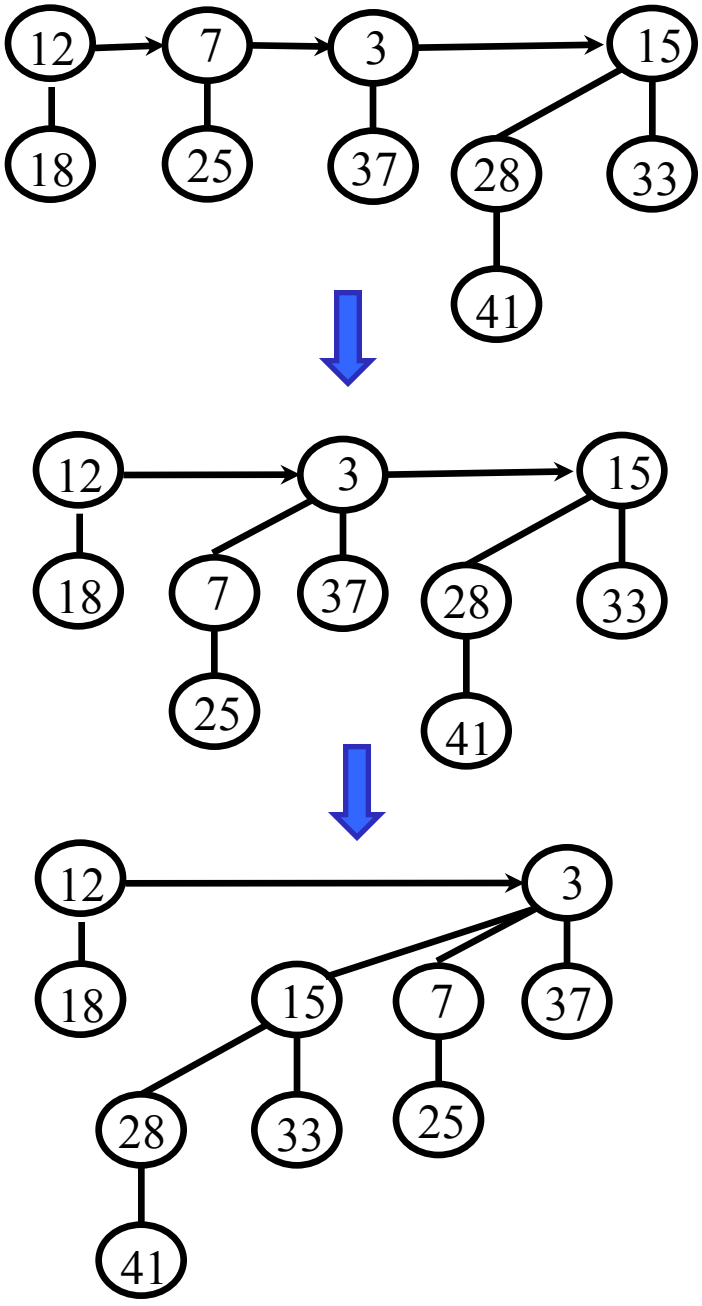
DeleteMin(H)


H
18
3
37
15
7
12
28
33
25
41

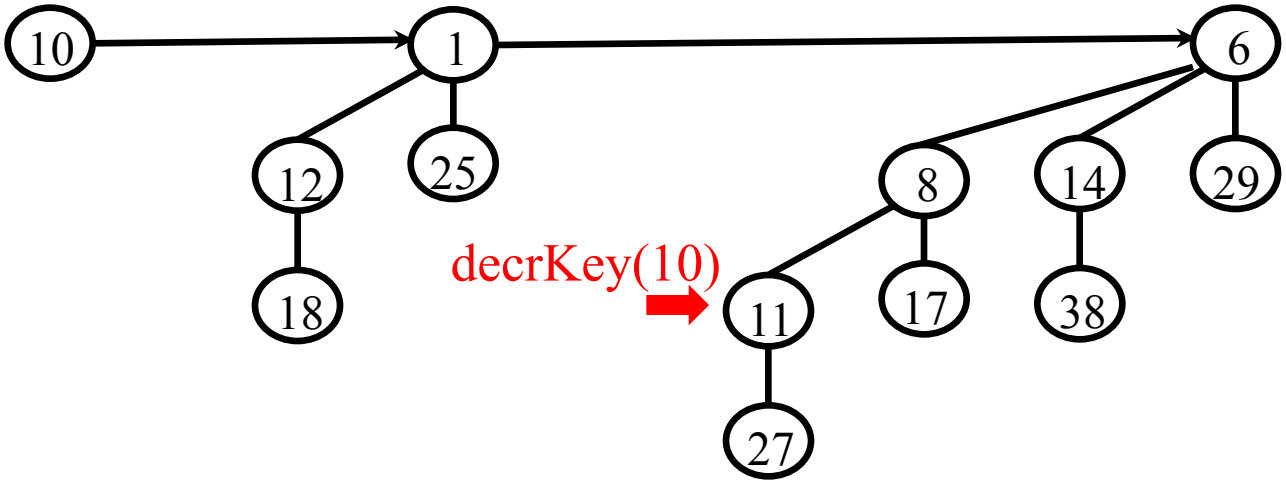

fondi

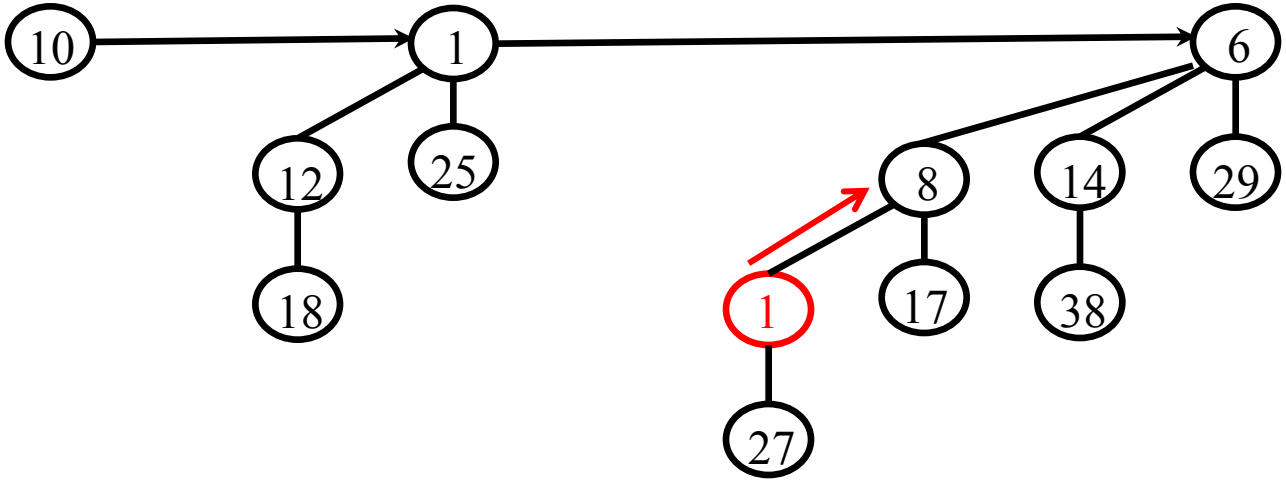

12
18
7
3
15
25
37
28
33
41


ristruttura

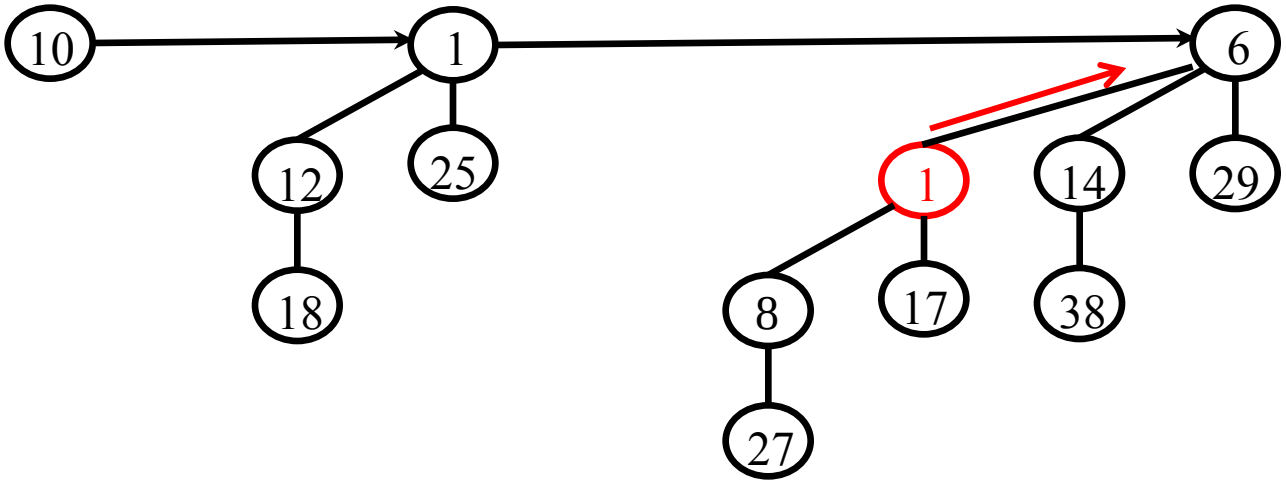

12
7
3
15
18
25
37
28
33
41
12
3
15
18
7
37
28
33
25
41
12
3
18
15
7
37
28
33
25
41

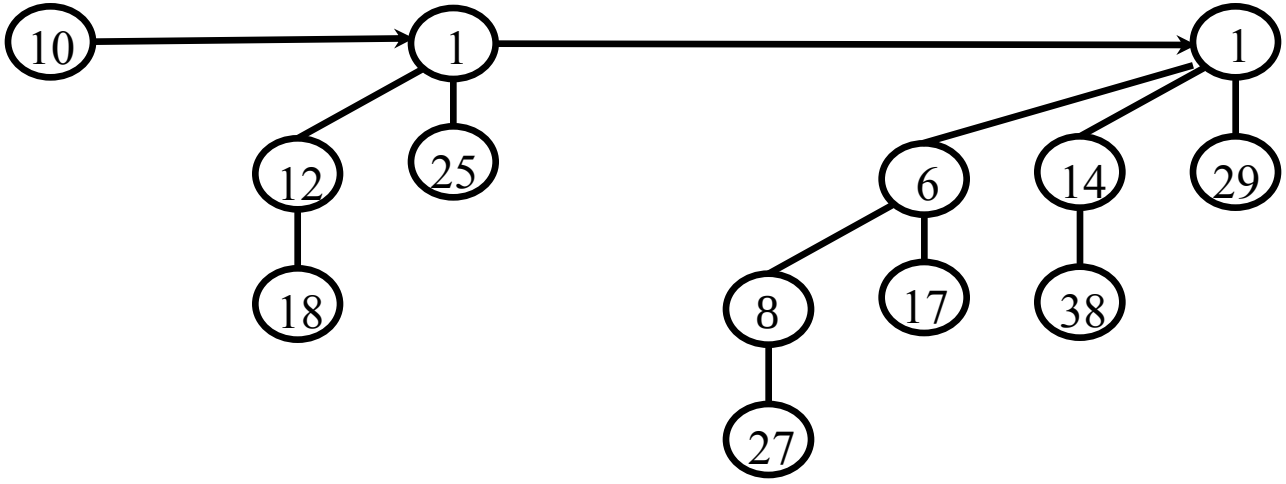
…decremento di una chiave

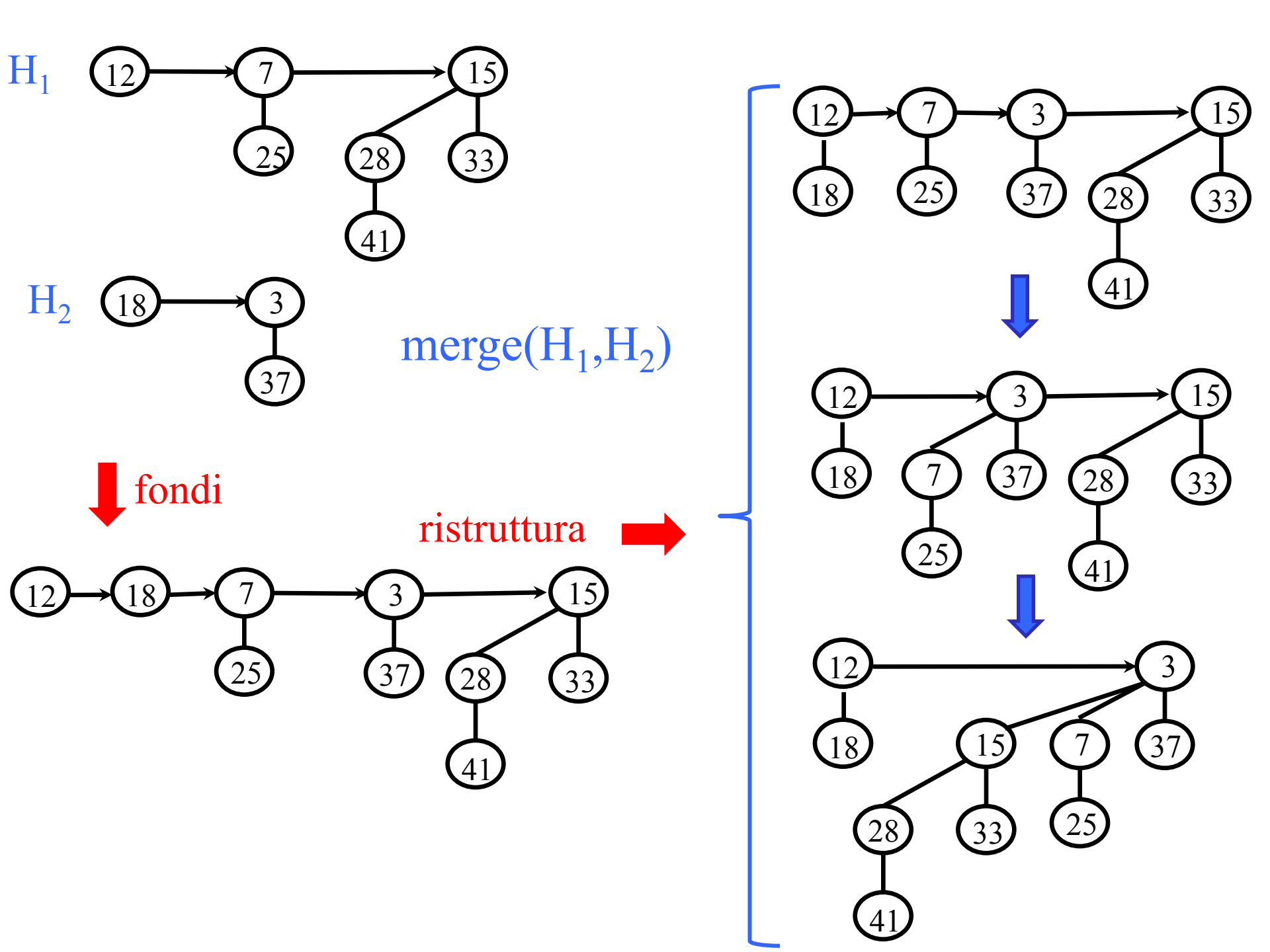




…decremento di una chiave






H1
12
7
15
25
28
33
41
H2
18
3
37
merge(H1,H2)
fondi
ristruttura
12
18
7
3
15
25
37
28
33
41

Heap di Fibonacci

(Fredman,Tarjan, 1987)

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	Delete	DelMin	IncKey	DecKey	merge
d-Heap (d cost.)	O(1)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(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(log n)*	O(log n)*	O(1)*	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)

Analisi ammortizzata

- Il costo ammortizzato di un'operazione è il costo "medio" rispetto a una sequenza *qualsiasi* di operazioni.
- Esempio: se un'operazione ha costo ammortizzato costante e eseguo una sequenza(qualsiasi) di k opearazioni è possibile che il costo di una singola operazione può non essere costante, ma l'intera sequenza costerà O(k)
- Diverso dal costo medio: non c'è nessuna distribuzione di probabilità (sulla sequenza da eseguire) e l'algoritmo è un algoritmo deterministico
- Molto utile quando si vogliono buone prestazioni sull'intera sequenza e non garanzie sulla singola operazione
  - esempio: progettare algoritmi veloci attraverso strutture dati efficienti

…per esempio nel nostro caso:

Teorema

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

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

Esercizio (di manipolazione)

Creare ed unire 2 Heap Binomiali sui seguenti insiemi:

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

$A_2=\{1,11,6,22,13,12,23,31\}$

Esercizio
Si consideri il problema di trovare, dato un vettore ordinato $A[1;n]$ di n bit, ovvero dove $A[i] \in \{0,1\}$ per ogni i, il numero k di zeri presenti in A. Si progetti un algoritmo con complessità $O(\log n)$ e poi un miglior algoritmo con tempo $O(\log k)$.

Esercizio
Sia $A[1;n]$ un vettore di n bit, ovvero di zeri e di uni. Si progetti un algoritmo che restituisca un indice k tale che il numero di zeri prima di k, ovvero in $A[1;k]$, è uguale al numero di uni dopo k, ovvero in $A[k+1;n]$. L'algoritmo deve impiegare tempo $O(n)$. E' possibile risolvere il problema con la stessa complessità computazionale e usando memoria ausiliaria costante?