Heap binomiali

Gli heap binomiali sono strutture dati su cui si possono eseguire efficientemente le operazioni:

***Make*(*H*)** : crea un heap vuoto.
Insert*(*H*, *x*)** : aggiunge il nodo ***x allo heap.
***Minimum*(*H*)** : ritorna il puntatore al nodo con chiave minima.
***ExtractMin*(*H*)** : ritorna il puntatore al nodo con chiave minima dopo averlo tolto dallo heap.
***Union*(H_1, H_2)** : riunisce due heap in un unico heap.

1

Oltre alle precedenti operazioni fondamentali degli *heap riunibili*, sugli heap binomiali definiremo anche le due ulteriori operazioni:

DecreaseKey*(*H*, *x*, *k*)** : cambia la chiave di ***x con una minore.

Delete*(*H*, *x*)** : toglie il nodo ***x.

2

Alberi binomiali

Gli heap binomiali sono insiemi di alberi binomiali.

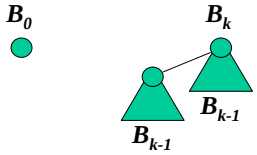
Un albero binomiale B_k di grado k è un albero ordinato (vi è un ordine tra i figli di ogni nodo) definito ricorsivamente nel seguente modo:

L'albero binomiale B_0 di grado 0 è costituito da un solo nodo, la radice.

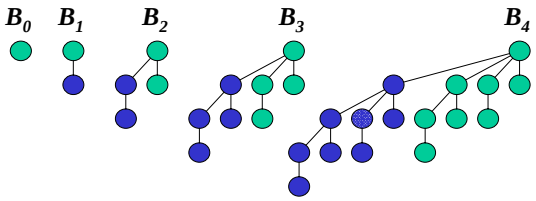
3

L'albero binomiale B_k di grado $k > 0$ è costituito da due alberi binomiali di grado $k - 1$ collegati tra loro ponendo la radice del primo come primo figlio della radice del secondo.

Graficamente:



4

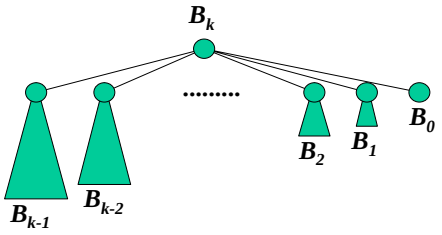


5

Proprietà degli alberi binomiali.
L'albero binomiale B_k:

1) ha 2^k nodi;
2) ha altezza k;
3) ha esattamente $\binom{k}{i}$ nodi a livello i;
4) la radice ha grado k e tutti gli altri nodi hanno grado minore;
5) se $x_{k-1}, x_{k-2}, \ldots, x_0$ sono i figli della radice elencati per indice decrescente da sinistra a destra allora x_i è radice di un albero binomiale B_i di grado i.

6



Limiti conseguenti.
Un albero binomiale di $n = 2^k$ nodi ha sia altezza che grado uguali a $k = \log_2 n$.

7

Dimostrazione.
L'albero binomiale B_0:
1) ha $2^0 = 1$ nodi;
2) ha altezza 0;
3) ha esattamente $\binom{0}{0}=1$ nodi a livello 0;
4) la radice ha grado 0 e non ci sono altri nodi;
5) la radice non ha figli;

quindi le cinque proprietà sono vere per $k = 0$.
Assumiamole vere per $k-1$ e dimostriamole per k.

8

1) B_k è costituito da due copie di B_{k-1} e quindi ha $2^{k-1} + 2^{k-1} = 2^k$ nodi;
2) l'altezza di B_k è uno in più dell'altezza di B_{k-1}. Quindi B_k ha altezza $k-1+1 = k$;
3) Sia $D(k,i)$ il numero di nodi a livello i in B_k. Allora

$$D(k,0) = 1 = \binom{k}{0}$$

$$D(k,k) = D(k-1,k-1) = \binom{k-1}{k-1} = \binom{k}{k}$$

9

Mentre per $0 < i < k$ i nodi a livello i sono i nodi a livello i di una delle due copie di B_{k-1} che formano B_k più i nodi a livello $i+1$ dell'altra e pertanto

$$D(k,i) = D(k-1,i) + D(k-1,i+1)$$
$$= \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$$

10

4) la radice di B_k ha un figlio più della radice di B_{k-1}. Essa ha quindi grado $k-1+1 = k$;

5) il primo figlio x_{k-1} della radice di B_k è radice di uno dei due B_{k-1} che lo formano mentre i figli successivi x_{k-2}, ..., x_0 sono i figli dell'altro B_{k-1} e, per ipotesi induttiva, sono quindi radici di alberi binomiali B_{k-2}, ..., B_0 .

11

Definizione di heap binomiale

Un heap binomiale H è un insieme di alberi binomiali tale che:

1) Ogni albero binomiale di H è ordinato a min-heap: ad ogni nodo è associata una chiave e la chiave di ciascun nodo è maggiore della chiave del padre.
2) Gli alberi binomiali in H hanno gradi distinti.

12

Proprietà degli heap binomiali.
Sia ***H*** un heap binomiale con ***n*** nodi in totale.
Allora:

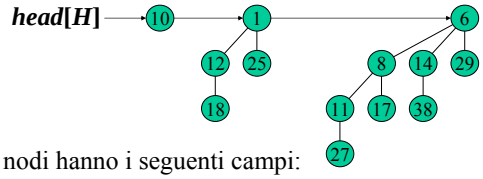
H contiene al più $\lfloor \log_2 n \rfloor + 1$ alberi.

13

Dimostrazione.
Sia ***H*** un heap binomiale con ***n*** nodi in totale e sia $b_k b_{k-1} \ldots b_0$ la rappresentazione binaria di ***n***. Allora:

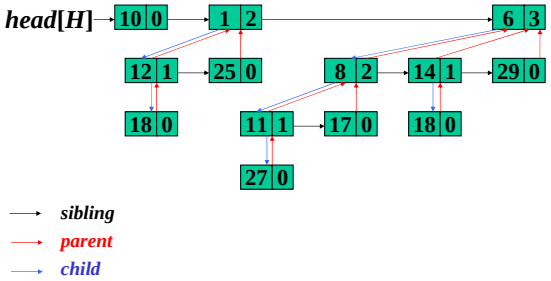
* un albero binomiale B_i in ***H*** contiene 2^i nodi e quindi ***n*** è somma di potenze di ***2*** distinte. Ma l'unico modo in cui si può esprimere ***n*** come somma di potenze di ***2*** distinte è $n = \sum_{i=0}^{k} b_i 2^i$
* Il numero di alberi è ≤ al numero di cifre nella rappresentazione binaria di ***n***, ovvero $\lfloor \log_2 n \rfloor + 1$

14



I nodi hanno i seguenti campi:
key : la chiave;
parent : puntatore al padre
child : puntatore al primo figlio
sibling : puntatore al fratello destro
degree : numero di figli.
oltre ad eventuali altri campi per informazioni associate alla chiave

15



16

Molte operazioni usano la funzione ausiliaria:

```
Link(y, z)    ▷ PRE: y e z sono radici di alberi binomiali
              ▷ dello stesso grado
  parent[y] ← z
  sibling[y] ← child[z]
  child[z] ← y
  degree[z] ← degree[z] + 1
```

Aggiunge ***y*** come primo figlio di ***z***. Richiede tempo costante $O(1)$.

17

La funzione ***Minimum*** è:

```
Minimum(H)    ▷ PRE: H non è vuoto
  x ← head[H], kmin ← key[x]
  while sibling[x] ≠ nil do
    x ← sibling[x]
    if kmin > key[x] then kmin ← key[x]
  return kmin
```

Siccome ci sono al più $\lfloor \log_2 n \rfloor + 1$ alberi essa richiede tempo $O(\log n)$.

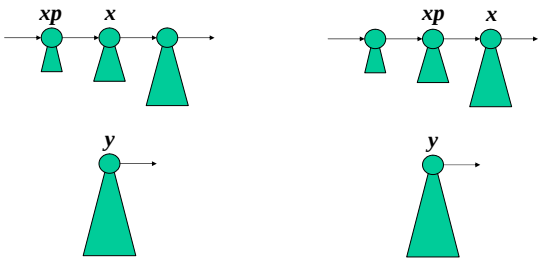
18

La funzione ***Union*** percorre le due liste delle radici riunendole nella prima delle due.

```
Union(H1,H2)
  x ← head[H1], xp ← nil
  y ← head[H2], head[H2] = nil
  while x ≠ nil and y ≠ nil do        ▷ I° while
```

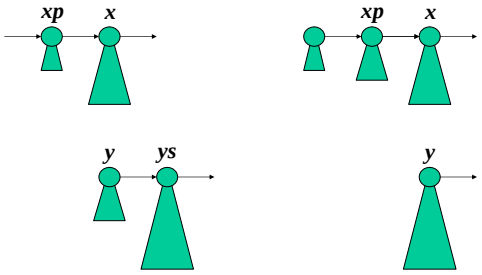
In dipendenza dei gradi di ***x*** e di ***y*** si possono presentare i seguenti casi:

Caso 1. ***deg***[***x***] < ***deg***[***y***]




```
    if degree[y] > degree[x] then        ▷ caso 1
      xp ← x, x ← sibling[x]
```

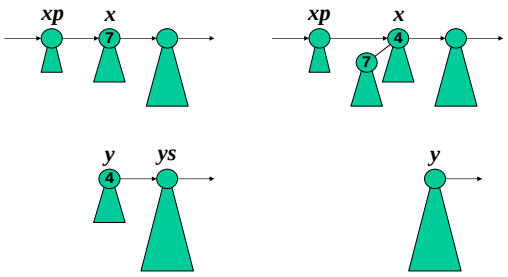

Caso 2. ***deg***[***y***] < ***deg***[***x***]




```
    else if degree[y] < degree[x] then        ▷ caso 2
      ys ← sibling[y]
      if xp = nil then head[H1] ← y
      else sibling[xp] ← y
      sibling[y] ← x, xp ← y
      y ← ys
```

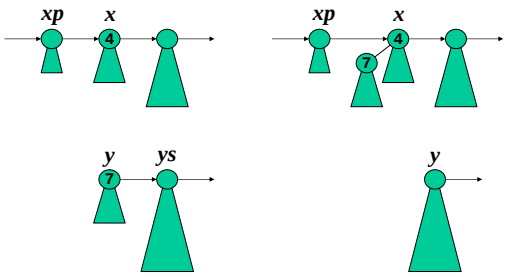
```
    else                                ▷ caso 3: degree[y] = degree[x]
      ys ← sibling[y]
```


Caso 3.1. ***key***[***x***] > ***key***[***y***]



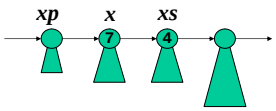
```
else                                  ▷ caso 3: degree[y] = degree[x]
   ys ← sibling[y]

   if key[x] > key[y] then            ▷ caso 3.1
      xs ← sibling[x]
      Link(x,y)
      if xp = nil then head[H1] ← y
      else sibling[xp] ← y
      sibling[y] ← xs
      x ← y, y ← ys
```


Caso 3.2. **key[y] > key[x]**

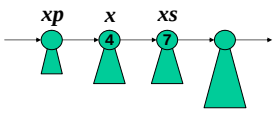


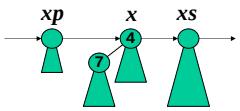

```
else                                  ▷ caso 3.2
   Link(y,x)
   y ← ys

xs ← sibling[x]
while xs ≠ nil and degree[x] = degree[xs] do ▷ II° while
```


Caso 3.3.



Caso 3.4.






```
   if key[x] > key[xs] then           ▷ caso 3.3
      Link(x,xs)
      if xp = nil then head[H1] ← xs
      else sibling[xp] ← xs
      x ← xs
   else                               ▷ caso 3.4
      sibling[x] ← sibling[xs]
      Link(xs,x)
   xs ← sibling[x]
```



```
if y ≠ nil then
   if xp = nil then head[H1] ← y
   else sibling[xp] ← y
```

Siano m_1 ed m_2 il numero di alberi contenuti nei due heap da unire ed m quello dello heap ottenuto.
Il ciclo while più esterno viene eseguito al più m_1+m_2 volte.

Ad ogni esecuzione del ciclo while interno il numero totale di alberi diminuisce di uno. Quindi esso viene eseguito al più $m_1 + m_2 - m$ volte.

Siccome m_1, m_2 ed m sono tutti $O(\log n)$ anche ***Union*** richiede tempo $O(\log n)$.

31

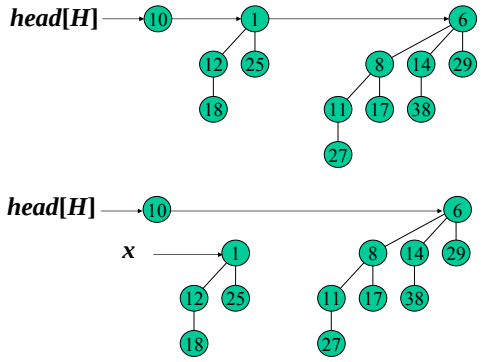
La funzione ***Insert*** è:

```
Insert(H, x)
   parent[x] ← nil,
   sibling[x] ← nil
   child[x] ← nil ,
   degree[x] ← 0
   head[H1] ← x
   Union(H,H1)
```

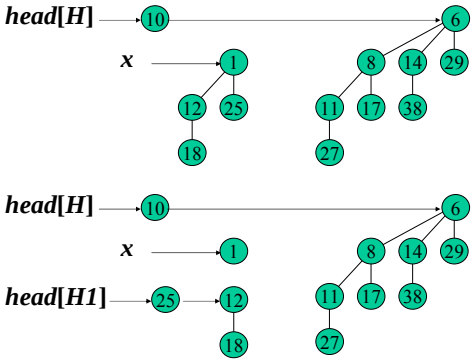
Siccome ***Union*** richiede tempo $O(\log n)$ anche ***Insert*** richiede tempo $O(\log n)$.

32

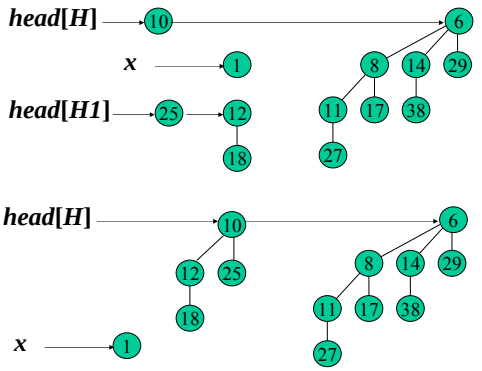
ExtractMin



33



34



35

La funzione ***ExtractMin***: idea

```
ExtractMin(H)
   x ← radice con chiave minima
   - elimina x dalla lista delle radici di H
   - crea uno heap H' tale che head[H'] punta alla
     lista concatenata dei figli di x, invertita
   Union(H,H1)          ▷ unisce i due heap
```

36

La funzione ***ExtractMin*** è:

```
ExtractMin(H)
  if head[H] = nil then return nil
  xp ← nil
  y ← head[H], kmin ← key[head[H]]
  while sibling[y] ≠ nil do          ▷ xp radice che precede
    if kmin > key[sibling[y]] then   ▷ quella minima
      kmin ← key[sibling[y]], xp ← y
    y ← sibling[y]
  if xp = nil then                   ▷ la radice minima è la prima
    x ← head[H], head[H] ← sibling[x]
  else                               ▷ la radice minima è quella che segue xp
    x ← sibling[xp], sibling[xp] ← sibling[x]
```



```
buildHeap(H1, x)   ▷ costruisce un heap H1 con i figli di x
Union(H,H1)        ▷ unisce i due heap
return x
```


dove la procedura ***buildHeap*** è:

```
buildHeap(H1, x)   ▷ costruisce un heap H1 con i figli di x
  head[H1] ← nil
  while child[x] ≠ nil do
    y ← child[x]
    child[x] ← sibling[y]
    parent[y] ← nil
    sibling[y] ← head[H1]
    head[H1] ← y
```


Il primo ciclo while percorre la lista delle radici ed ha quindi complessità $O(\log n)$.

buildHeap percorre la lista dei figli di una radice. Siccome il grado è $O(\log n)$ anche tale ciclo ha complessità $O(\log n)$.

Infine ***Union*** richiede tempo $O(\log n)$ e quindi anche ***ExtractMin*** richiede tempo $O(\log n)$.

La funzione ***DecreaseKey*** è:

```
DecreaseKey(H, x, k)
  if k > key[x] then
    errore "la nuova chiave non è minore della vecchia"
  key[x] ← k
  y ← parent[x]
  while y ≠ nil and key[y] > key[x] do
    k ← key[x], key[x] ← key[y], key[y] ← k
    "scambia anche eventuali campi associati"
    x ← y, y ← parent[x]
```

Siccome l'altezza è $O(\log n)$ anche ***DecreaseKey*** richiede tempo $O(\log n)$.

La funzione ***Delete*** è:

```
Delete(H, x)
  DecreaseKey(H, x, -∞)
  ExtractMin(H)
```

Siccome sia ***DecreaseKey*** che ***ExtractMin*** hanno complessità $O(\log n)$ anche ***Delete*** richiede tempo $O(\log n)$.

Operazione	*Complessità*
Make	**$\Theta(1)$**
Insert	**$O(\log n)$**
Minimum	**$O(\log n)$**
ExtracMin	**$O(\log n)$**
Union	**$O(\log n)$**
DecreaseKey	**$O(\log n)$**
Delete	**$O(\log n)$**

43

Esercizio 26.
Supponiamo che non esista una rappresentazione della chiave $-\infty$.
Riscrivere al funzione ***Delete*** per gli heap binomiali in modo che essa non usi la chiave $-\infty$.
Assicurarsi che la complessità rimanga ***O*(log *n*)**.

44

Esercizio 27.
Nella funzione ***ExtractMin*** abbiamo dovuto percorrere tutta la lista dei figli del nodo estratto per invertirne l'ordine.
Questo perché la lista delle radici è ordinata per grado crescente mentre le liste dei figli sono ordinate per grado decrescente.
Cosa succede se ordiniamo le due liste in modo concorde?

45

Esiste una struttura dati, gli *heap di Fibonacci*, in cui le stesse operazioni si eseguono con le seguenti complessità *ammortizzate*.

Operazione	*Complessità ammortizzata*
Make	**$\Theta(1)$**
Insert	**$\Theta(1)$**
Minimum	**$\Theta(1)$**
ExtractMin	**$O(\log n)$**
Union	**$\Theta(1)$**
DecreaseKey	**$\Theta(1)$**
Delete	**$O(\log n)$**

46

Esercizio 28.
Dimostrare che non esiste nessuna struttura dati che permetta di eseguire le tre operazioni ***Make***, ***Insert*** ed ***ExtractMin*** in tempo costante (sia caso pessimo che ammortizzato).

47