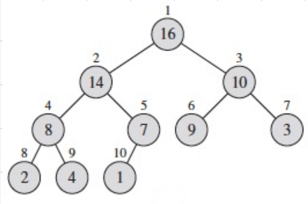
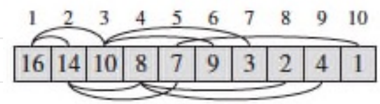
STRUTTURE DATI
-HEAP-

Heap

Struttura dati composta da un array, può essere considerato come un albero binario quasi completo





Ogni nodo dell'albero corrisponde ad un elemento dell'array

A.length = numero di elementi nell'array A

A.heap-size = numero degli elementi dell'heap memorizzati nell'array A

Come trovare gli indici del padre, figlio sinistro, figlio destro:

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

Proprietà heap

- Heap di n elementi ha altezza $\Theta(\log n)$
- Heap di n elementi contiene $n/2$ foglie
- Heap di n elementi ha al più $n/2^{h+1}$ nodi di altezza h, esattamente $n/2^{h+1}$ se heap è un albero binario completo bilanciato
- il numero di nodi di un sottoalbero di un albero binario quasi completo è <= $2n/3$

Max heap

L'elemento più grande è memorizzato nella radice

$A[\text{PARENT}(i)] \geq A[i]$

Min heap

L'elemento più piccolo è memorizzato nella radice

$A[\text{PARENT}(i)] \leq A[i]$

Costruzione heap

È possibile utilizzare la procedura max-heapify dal basso verso l'alto per convertire un array $A[1,\dots,n]$ con $n = A.length$ in un max-heap

```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.heap-size and A[l] > A[i]
       massimo = l
  else massimo = i
  if r ≤ A.heap-size and A[r] > A[massimo]
       massimo = r
  if massimo ≠ i
       scambia A[i] con A[massimo]
       MAX-HEAPIFY(A, massimo)
```

Il tempo di esecuzione può essere descritto dalla ricorrenza:

$$T(n) \leq T(2n/3) + \Theta(1) = O(\lg n)$$

Heapsort

Suddividere il vettore da ordinare in due parti, una che rappresenta l'heap e una che contiene gli elementi ordinati

Il processo di heapsort comprende 3 parti

① Costruzione heap ② Ordinamento ③ Ricostruzione vettore ordinato

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for i = A.length downto 2
       scambia A[1] con A[i]
       A.heap-size = A.heap-size − 1
       MAX-HEAPIFY(A, 1)
```

```
BUILD-MAX-HEAP(A)
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
       MAX-HEAPIFY(A, i)
```

Il seguente algoritmo impiega un tempo $O(n \lg n)$ in quanto la chiamata a Build-max-heap impiega $O(n)$ e le $n-1$ chiamate di Max-heapify impiega $O(\lg n)$

Code di priorità

Struttura dati che serve a mantenere un insieme S di elementi, ciascuno con un valore associato detto chiave

Operazioni - code di max-priorità:

```
HEAP-MAXIMUM(A)
  return A[1]
```

```
HEAP-INCREASE-KEY(A, i, key)
  if key < A[i]
       error "la nuova chiave è più piccola di quella corrente"
  A[i] = key
  while i > 1 and A[PARENT(i)] < A[i]
       scambia A[i] con A[PARENT(i)]
       i = PARENT(i)
```

```
HEAP-EXTRACT-MAX(A)
  if A.heap-size < 1
       error "underflow dell'heap"
  max = A[1]
  A[1] = A[A.heap-size]
  A.heap-size = A.heap-size − 1
  MAX-HEAPIFY(A, 1)
  return max
```

```
MAX-HEAP-INSERT(A, key)
  A.heap-size = A.heap-size + 1
  A[A.heap-size] = −∞
  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Heap di Fibonacci

Struttura dati con duplice compito, supporta un insieme di operazioni che formano l'heap riunibile e inoltre le sue operazioni vengono effettuate in un tempo ammortizzato

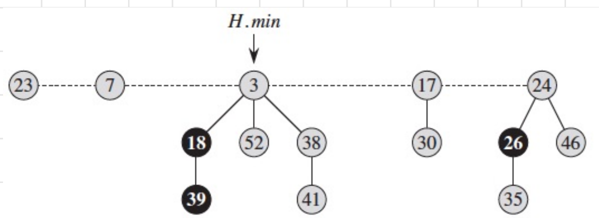
Heap riunibili

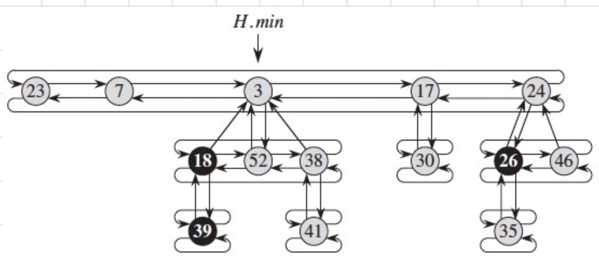
Struttura dati in cui ogni elemento ha una chiave

- Make-heap: crea un nuovo heap senza elementi
- Insert: Inserisce un elemento in un heap
- Minimum: Restituisce un puntatore all'elemento con chiave minima
- Extract-Min: toglie dall'heap l'elemento con chiave minima e ne restituisce il puntatore
- Union: Crea un unico heap a partire da due heap, distruggendoli successivamente
- Decrease-key: Aggiorna il valore di una chiave di un dato elemento
- Delete: Cancella un elemento da un array

Procedura	Heap binario (caso peggiore)	Heap di Fibonacci (ammortizzato)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Un heap di Fibonacci è un insieme di alberi radicati che sono min-heap ordinati



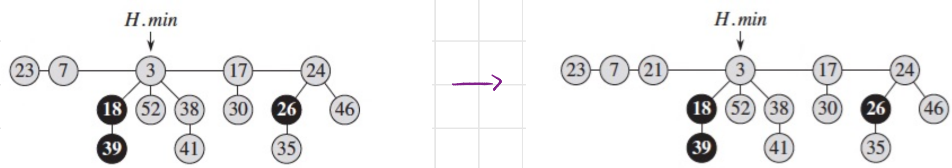


Operazioni su heap riunibili

L'idea chiave è ritardare il lavoro più è possibile

Inserire un nodo

```
FIB-HEAP-INSERT(H, x)
x.degree = 0
x.p = NIL
x.child = NIL
x.mark = FALSE
if H.min == NIL
    crea una lista delle radici per H che contiene solo x
    H.min = x
else inserisce x nella lista delle radici di H
    if x.key < H.min.key
        H.min = x
H.n = H.n + 1
```



Unire due heap di Fibonacci

```
FIB-HEAP-UNION(H1, H2)
H = MAKE-FIB-HEAP()
H.min = H1.min
concatena la lista delle radici di H2 con la lista delle radici di H
if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
    H.min = H2.min
H.n = H1.n + H2.n
return H
```

Estrarre il nodo minimo

```
FIB-HEAP-EXTRACT-MIN(H)
z = H.min
if z ≠ NIL
    for ciascun figlio x di z
        aggiungi x alla lista delle radici di H
        x.p = NIL
    rimuovi z dalla lista delle radici di H
    if z == z.right
        H.min = NIL
    else H.min = z.right
        CONSOLIDATE(H)
    H.n = H.n − 1
return z
```

```
FIB-HEAP-LINK(H, y, x)
Rimuove y dalla lista delle radici di H
Trasforma y in un figlio di x, incrementando x.degree
y.mark = FALSE
```

```
CONSOLIDATE(H)
Sia A[0 . . D(H.n)] un nuovo array
for i = 0 to D(H.n)
    A[i] = NIL
for ciascun nodo w nella lista delle radici di H
    x = w
    d = x.degree
    while A[d] ≠ NIL
        y = A[d]          // Un altro nodo con lo stesso grado di x
        if x.key > y.key
            scambia x con y
        FIB-HEAP-LINK(H, y, x)
        A[d] = NIL
        d = d + 1
    A[d] = x
H.min = NIL
for i = 0 to D(H.n)
    if A[i] ≠ NIL
        if H.min == NIL
            crea una lista di radici per H che contiene solo A[i]
            H.min = A[i]
        else inserisce A[i] nella lista delle radici di H
            if A[i].key < H.min.key
                H.min = A[i]
```

Diminuire il valore di una chiave

```
FIB-HEAP-DECREASE-KEY(H, x, k)
if k > x.key
    error "la nuova chiave è maggiore di quella corrente"
x.key = k
y = x.p
if y ≠ NIL and x.key < y.key
    CUT(H, x, y)
    CASCADING-CUT(H, y)
if x.key < H.min.key
    H.min = x
```

```
CUT(H, x, y)
Rimuove x dalla lista dei figli di y, decrementando y.degree
Aggiunge x alla lista delle radici di H
x.p = NIL
x.mark = FALSE
```

```
CASCADING-CUT(H, y)
z = y.p
if z ≠ NIL
    if y.mark == FALSE
        y.mark = TRUE
    else CUT(H, y, z)
        CASCADING-CUT(H, z)
```

Cancellare un nodo

```
FIB-HEAP-DELETE(H, x)
FIB-HEAP-DECREASE-KEY(H, x, −∞)
FIB-HEAP-EXTRACT-MIN(H)
```