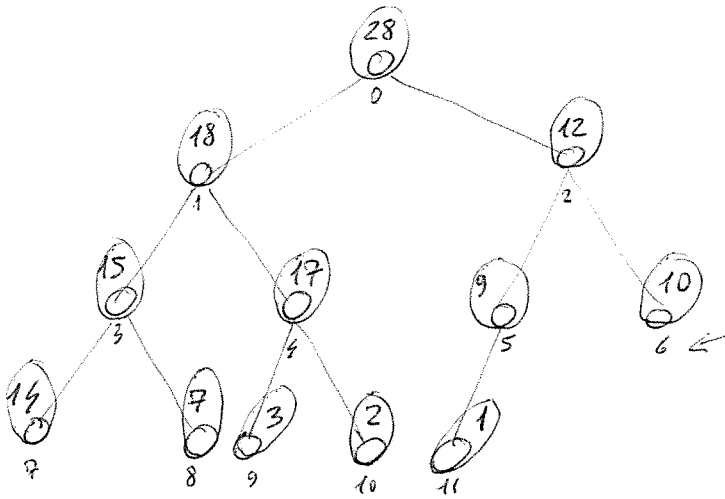
Implementazione di uno HEAP

Possiamo utilizzare un array di n posizioni, dove n indica il numero di nodi dell'albero e applicando la seguente regola:

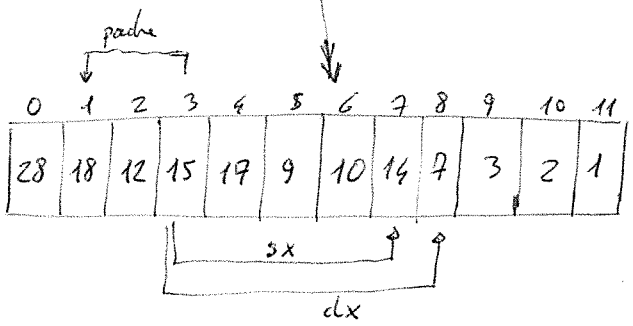
- la radice occupa la posizione $i = 0$.
- se un nodo occupa la posizione i, allora il figlio sinistro (se esiste) occupa la posizione $2i + 1$ e il figlio destro (se esiste) occupa la posizione $2i + 2$.



Gli indici "sotto" ogni nodo corrispondono alle posizioni nell'array

Il padre di un nodo che occupa la posizione i occupa la posizione

$\lfloor (i-1)/2 \rfloor$



Osservazioni:

- se $2i+1 \geq n$, allora $u.sx = NULL$
- se $2i+2 \geq n$, allora $u.dx = NULL$
- se $i = 0$, allora $u.padre = NULL$.

La rappresentazione non richiede memoria aggiuntiva, a parte quella per i dati e quella della variabile heapSize.

Vale per ogni albero completo a sinistra.

L'HEAP è un albero binario completo a sinistra e, pertanto, può essere implementato in questo modo, usando un array dinamico.

Chiamiamo l'array dinamico heapArray.
La variabile heapSize indica il numero di elementi presenti nella heap (prima posizione libera in heapArray).

Il codice che implementa la coda a priorità tramite un HEAP è quindi il seguente:

```
EMPTY():
RETURN (heapSize == 0);
```

```
FIRST():
IF (!EMPTY()) RETURN heapArray[0];
```

```
ENQUEUE (e):
VerificaRaddoppio();
heapArray[heapSize] = e;
heapSize = heapSize + 1;
Riorganizzazione (heapSize - 1);

DEQUEUE ():
IF (!EMPTY()){
  massimo = heapArray[0];
  heapArray[0] = heapArray[heapSize - 1];
  heapSize = heapSize - 1;
  Riorganizzazione (0);
  VerificaDimezzamento();
  RETURN massimo;
}
```

La procedura Riorganizzazione() (e le sottoprocedure da essa usa) sono descritte di seguito...

```
Riorganizzazione (i):

WHILE ( i > 0  &&  HeapArray[i].prio > HeapArray[Padre(i)].prio)
{  Scambia (i, Padre(i));
   i = Padre (i);
}
WHILE (Sinistro (i) < heapSize  &&  i != MIGLIORE PADRE FIGLI (i))
{  migliore = MigliorePadreFigli (i);
   Scambia (i, migliore);
   i = migliore;
}
```

```
MigliorePadreFigli (i):
j = K = Sinistro (i);
IF (K+1 < HeapSize)  K = K+1;
IF (HeapArray[K].prio > HeapArray[j].prio)  j = K;
IF (HeapArray[i].prio >= HeapArray[j].prio)  j = i;
RETURN j;
```

```
Padre (i):
RETURN (i-1)/2;
```

```
Sinistro (i):
RETURN (2i+1);
```

```
Scambia (i, j):
tmp = HeapArray[i];
HeapArray[i] = HeapArray[j];
HeapArray[j] = tmp;
```

Teorema 2.2. L'esecuzione di K operazioni EMPTY, ENQUEUE, FIRST o DEQUEUE su uno heap contenente inizialmente m elementi richiede tempo $O(K \log n)$, dove $n = m + K$.

Dimostrazione. L'analisi è abbastanza semplice.

Infatti:

EMPTY ()

FIRST () $\Rightarrow$ Complessità $O(1)$

Il costo di ENQUEUE e DEQUEUE è dato da:

- numero costante di operazioni → $O(1)$
- costo del RADDOPPIO/DIMEZZAMENTO quando si verificano → costo "ammortizzato" $O(1)$
- costo di RIORGANIZZAZIONE → ?

↓

Viene invocata sull'elemento che viola la proprietà

Tale procedura ripristina la proprietà di heap.

Il primo ciclo WHILE permette ad un elemento di "risalire" lungo l'heap, al più fino alla radice.

Il secondo ciclo WHILE fa "scendere" un elemento lungo l'heap.

Si noti che, per ogni posizione i dello heap e per ogni elemento e = heapArray[i], NON può mai accadere che vengano eseguiti entrambi i cicli WHILE:

- o il primo
- o il secondo
- o nessuno (l'elemento è al posto giusto)

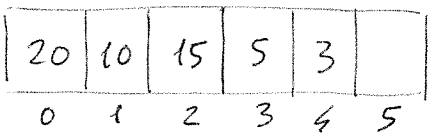
Per esempio, nel primo ciclo WHILE, lo scambio avviene SOLO se heapArray[i].prio > heapArray[padre(i)].prio. In questo caso, heapArray[i].prio è sicuramente maggiore delle priorità dell'altro nodo figlio (e dei suoi discendenti), per la definizione di heap. Non può "scendere".

Parimenti, il secondo ciclo WHILE, scambia il contenuto del nodo con il maggiore dei suoi figli. Essendo la priorità di quest'ultimo nodo minore di quella del "padre precedente alla violazione", a sua volta minore delle priorità dei nodi dei livelli superiori, sicuramente non può "risalire".

Pertanto il costo di RIORGANIZZAZIONE è proporzionale all'altezza dell'~~heap~~, cioè logaritmico nel numero di elementi presenti, al più $n = m + K$.

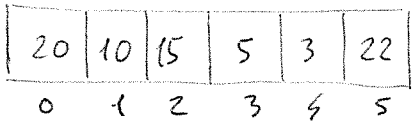
In conclusione, l'esecuzione delle K operazioni richiede un tempo ammortizzato (per via di raddoppi e dimezzamenti) logaritmico nel numero di elementi presenti nello heap, cioè $O(K \cdot \log n)$.

Esempio di esecuzione



HeapSize = 5

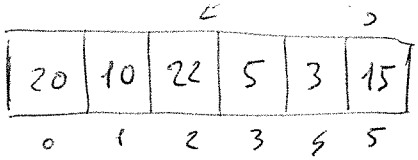
Enqueue (22)



HeapSize = 6

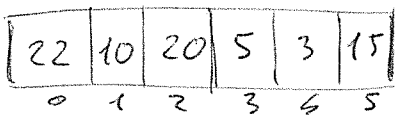
Viene invocata RIORGANIZZAZIONE che scambia l'elemento appena inserito con il padre (22 > 15)

$i = 5$ $\quad\quad$ $\lfloor (i-1)/2 \rfloor = 2$

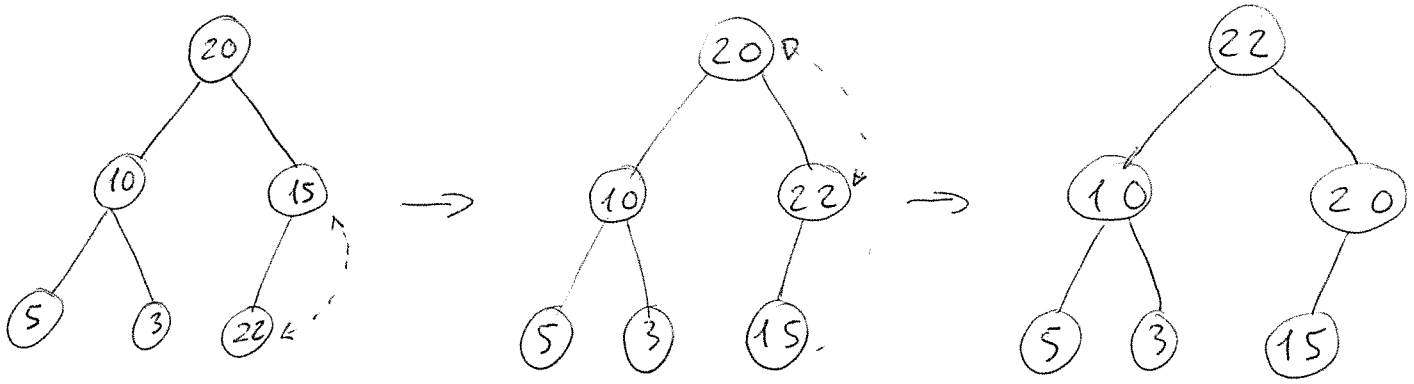


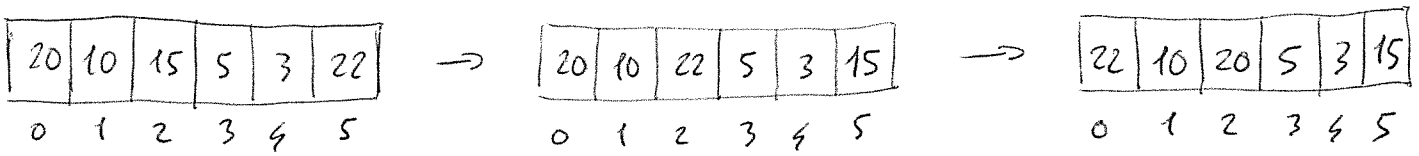
HeapSize = 6

L'elemento in posizione $i = 2$ è ancora minore del padre $\lfloor \frac{i-1}{2} \rfloor = 0$ ($22 > 20$), per cui i due elementi vengono scambiati



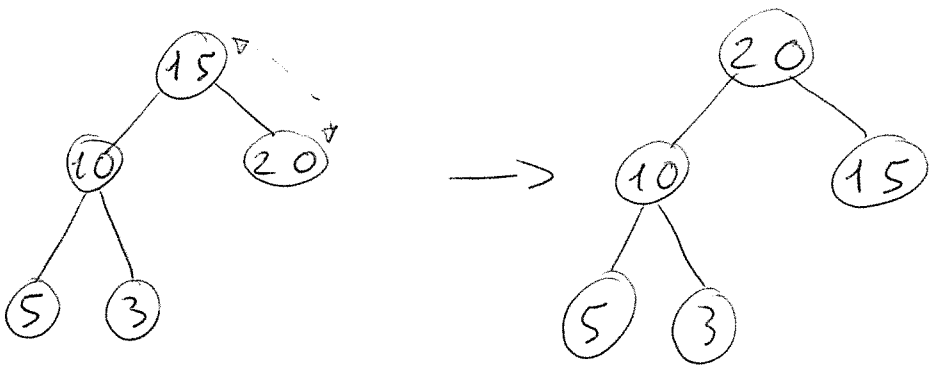
HeapSize = 6.

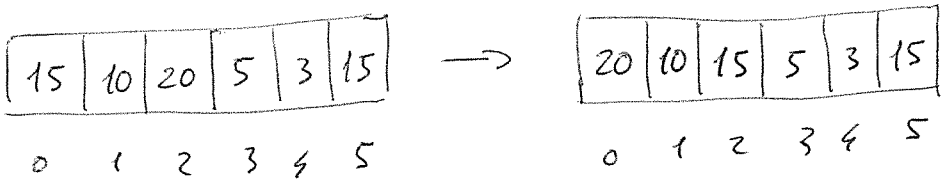




L'operazione Dequeue, dopo aver memorizzato il max in x, procede all'eliminazione: copia l'ultimo elemento nello heap nella radice

$x = 22$





$heapSize = 5$ $heapSize = 5$

Riorganizzazione (0)
scambia l'elemento in $i=0$ con il max dei suoi figli ($2i+1 = 1$ e $2i+2 = 2$).
La variabile i assume il valore 2.
Poiché $2 \cdot 2 + 1 \geq 5$ ($2 \cdot i + 1$) $heapSize$ la procedura si arresta.