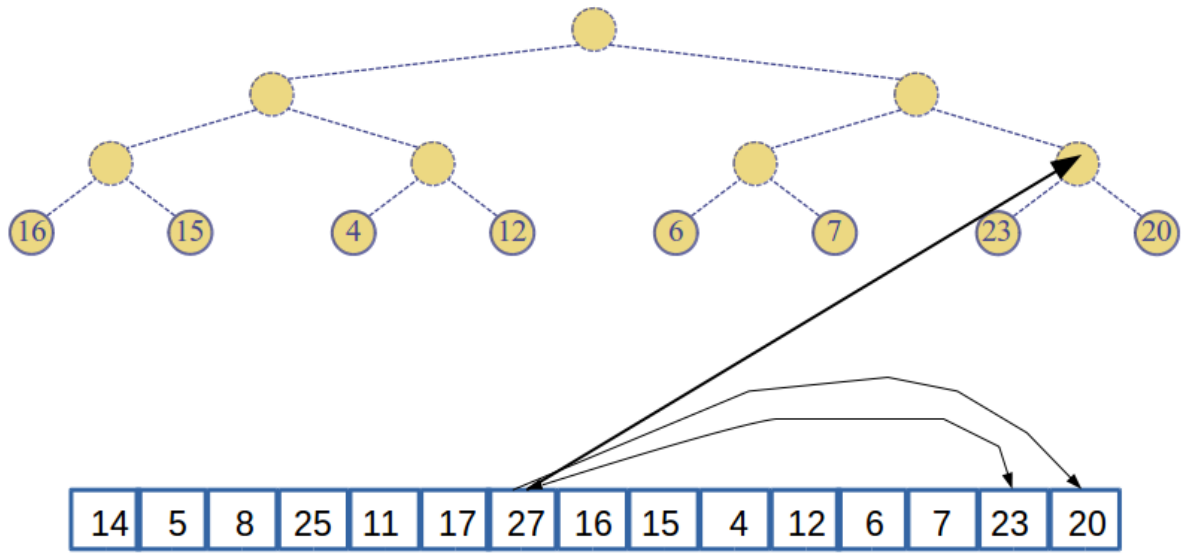
# Appunti su Heap - Parte 2

Consideriamo due problemi: l'analisi di heapify e il problema di raddoppiare l'array quando diventa pieno

Il primo aspetto che affrontiamo è il costo di `heapify`. Per semplicità, consideriamo la versione di tale algoritmo presentata sulle slide e nel libro:

```
protected void heapify() {
  int startIndex = parent(size()-1);    // start at PARENT of last entry
  for (int j=startIndex; j >= 0; j--)   // loop until processing the root
    downheap(j);
}
```

Il problema di calcolare il costo (di caso peggiore) di questo algoritmo è essenzialmente un problema di attribuzione dei costi. Per comprenderlo meglio, partiamo dall'esempio considerato nella slide e nel libro.

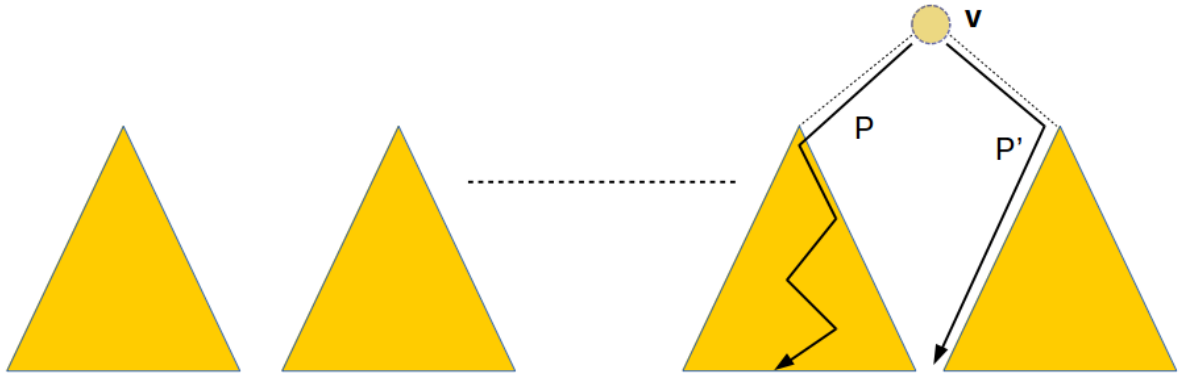



La costruzione bottom-up dell'heap può essere rappresentata dall'albero riportato nella parte superiore della figura, nel quale abbiamo due tipi di nodi: i) i nodi con contorno continuo e ii) quelli con contorno tratteggiato. All'inizio di ogni iterazione del ciclo for di heapify (v. sopra), i primi sono i nodi corrispondenti a componenti dell'array maggiori di `j`, mentre gli altri corrispondono a componenti minori o uguali a `j`. Il  valore iniziale di `j` corrisponde al nodo avente chiave 27 e in posizione 6 nell'array. Tale posizione è quella del genitore dei nodi che in figura hanno chiavi 23 e 20. Ciò significa che tutti i nodi aventi indice maggiore di 6 iniziano con il proprio contorno continuo.

Se osserviamo l'algoritmo ci osserviamo che il passaggio di un nodo da contorno tratteggiato a contorno continuo corrisponde a una iterazione del ciclo for. Ogni iterazione del ciclo for corrisponde a un'operazione `downheap(j)`, dove j vale 6 nella figura soprastante. E' facile convincersi che il costo di `heapify()` è (a meno di qualche costante) pari alla somma dei costi di queste operazioni di downheap. Si noti anche che *tale costo, per come è definito l'algoritmo, va calcolato una volta sola, per la precisione la prima volta che un nodo da tratteggiato diviene continuo.* E' importante tenere presente che, mentre la chiave associata a un nodo v dell'heap (visto come

albero binario completo) può cambiare nel tempo, il nodo diventerà continuo una volta sola e rimarrà tale da quel momento in poi.

Con queste premesse e osservazioni, calcolare il costo di `heapify()` diventa più semplice. Chiediamoci qual è il costo di caso peggiore della generica iterazione del ciclo for di `heapify()`. La situazione è esemplificata dalla figura seguente:




Il nodo v in figura corrisponde a un certo valore di j (ad esempio, nella prima iterazione dell'esempio in figura **v** sarebbe il nodo con chiave *iniziale* 27). Quando il nodo **v** diventa a contorno continuo, ciò corrisponde all'applicazione dell'algoritmo di downheap a partire dalla posizione corrispondente al nodo nell'algoritmo `heapify()`. Nel caso peggiore, la chiave inizialmente presente in **v** andrà scambiata con le chiavi di nodi di livello via via più alto, fino a raggiungere le foglie. Ciò significa che, nel caso peggiore, il costo di `downheap(v)` sarà proporzionale alla lunghezza di un certo cammino P che da **v** raggiunge una delle foglie. Si noti che, poiché tutti gli heap corrispondono ad alberi binari completi, tutti i cammini da **v** a una delle foglie hanno la stessa lunghezza (a meno di 1, in quanto l'ultimo livello potrebbe essere incompleto). Questo significa che, per rappresentare il costo di caso peggiore di `downheap(v)` possiamo scegliere uno qualsiasi dei cammini che da **v** portano a una delle foglie, ad esempio P'.

Nel seguito, indichiamo con $P'(v)$ il cammino che dal nodo $v$ segue prima il figlio destro e poi iterativamente tutti i figli sinistri fino a raggiungere un foglia. In base alle considerazioni fatte sopra possiamo affermare che:

$$Costo(heapify) \leq c \cdot \sum_{i=1}^{n} |P'(i)|,$$

dove $|P'(i)|$ denota il numero di archi del cammino $P'(i)$ e $c$ è la solita "costante abbastanza grande". L'ultima osservazione da fare è che i cammini $P'(i)$ sono *disgiunti* negli archi. Ciò significa che

$$c \cdot \sum_{i=1}^{n} |P'(i)| \leq c \cdot < Numero\, di\, archi\, nell'\, heap >= c(n-1),$$

in quanto ogni albero ha esattamenti $n-1$ se $n$ è il numero di vertici.

## Rappresentare heap con array - Il problema del raddoppio

Come avviene con altre strutture dati (ad esempio liste rappresentate mediante array, come la classe `ArrayList` in Java), rappresentare heap mediante array ha indubbi vantaggi ma qualche svantaggio. Il principale di questi è che l'array è essenzialmente un oggetto statico (si pensi alla gestione di array nel linguaggio C). Ciò significa che per aumentare la dimensione di un array

occorre:

i) creare un nuovo array

ii) copiare tutti gli elementi presenti nel vecchio array all'interno del nuovo.

Tali operazioni hanno ovviamente un costo, che è *proporzionale alla dimensione del nuovo array che viene creato*, come è facile convincersi. Il problema è ovviamente presente nel caso degli array.

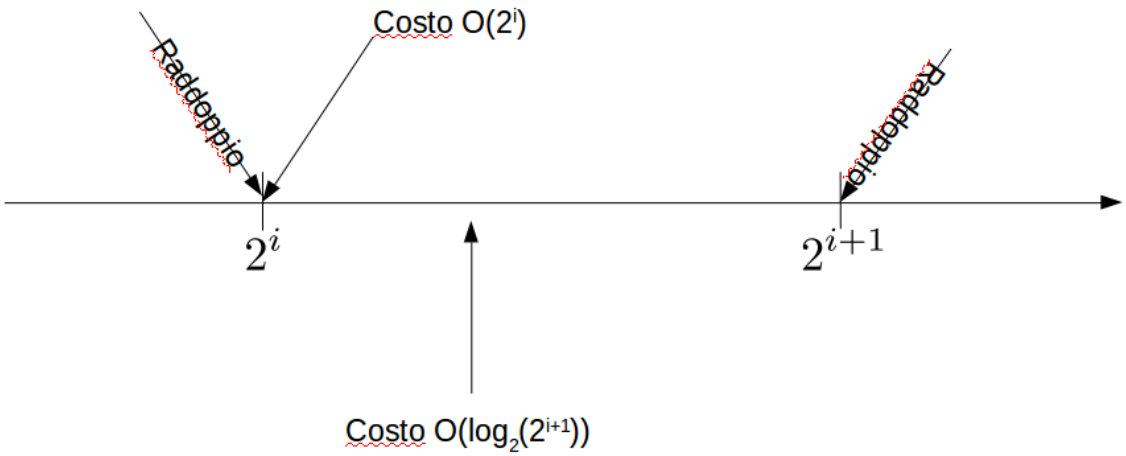
Una tecnica largamente usata è quella del *raddoppio*. Se un array diventa pieno a seguito dell'inserimento dell'$m$-esimo elemento, il nuovo array avrà dimensione $2m$. Cerchiamo di capire, usando l'esempio degli heap, perché tale scelta sia sensata.

Consideriamo un heap rappresentato mediante array e supponiamo che, ogni volta che l'array sottostante diventa pieno, il nuovo inserimento sia preceduto dal raddoppio dell'array usato. In linea di principio, tale operazione rende pessimo il costo di caso peggiore. Ad esempio, se l'array è pieno dopo l'inserimento di $n$ elementi, il successivo inserimento avrà costo proporzionale a $2n$. Ciò sembra contraddire quanto abbiamo detto circa l'efficienza degli array. Tuttavia, a pensarci bene, i casi in cui il costo pagato è addirittura lineare sono pochi, si verificano soltanto quando l'array è pieno. In tutti gli altri casi il costo pagato è quello tipico degli heap.

Un modo per dare conto di tale osservazione è effettuare un' *analisi ammortizzata*. Ciò significa che non analizziamo il costo di caso peggiore di una *singola operazione*, ma il costo di caso peggiore di una *sequenza di operazioni*. Nel nostro caso supporremmo di effettuare $n$ inserimenti in sequenza nell'array, con array di dimensione iniziale pari a $1$. Ci chiediamo quale sia il costo complessivo di tali inserimenti. Tale costo può essere determinato in base alle seguenti osservazioni:

1. Il numero di raddoppi che avremo sarà logaritmico nel numero di inserimenti. In particolare, avremo raddoppi dopo $1, 2, 4, \ldots, 2^{\lfloor \log_2 n \rfloor}$ inserimenti. Regola (più o meno): $i$-esimo raddoppio dopo $2^{i-1}$ inserimenti.
2. Per $i = 1, 2, \ldots, \lfloor \log_2 n \rfloor$, consideriamo l'intervallo (di interi) tra gli inserimenti $2^i$-esimo e $2^{i+1}$-esimo (quest'ultimo escluso). In totale, abbiamo $2^i$ inserimenti in tale intervallo. Il primo di tali inserimenti il $2^i$-esimo, determina il raddoppio dell'array e ha quindi costo proporzionale a $2^{i+1}$. Tutti gli altri $2^i - 1$ inserimenti di questo intervallo hanno costo ottimale, quindi al più logaritmico nella dimensione dell'array. Poiché nell'i-esimo intervallo la dimensione dell'array rimane compresa tra $2^i$ e $2^{i+1}$, il costo di ciascuno di tali inserimenti sarà al più $c \log_2 2^{i+1} = c(i+1) \log_2 2 = c(i+1)$. Il costo complessivo nell'$i$-esimo intervallo è pertanto al più $c \cdot 2^{i+1}$ (costo raddoppio) + $2^i \cdot c(i+1)$.

Queste considerazioni sono riassunte nella figura seguente:



L'ultimo intervallo (quello nel quale avviene l'$n$-esimo inserimento) ha lunghezza $n - 2^{\lfloor \log_2 n \rfloor}$ (potrebbe essere $0$ ovviamente). Con queste considerazioni, il costo sarà al più (scegliendo una costante $c$ sufficientemente grande):

$$Costo \leq c \sum_{i=1}^{\lfloor \log_2 n \rfloor} (2^{i+1} + (i+1)2^{i+1}) < 2c \sum_{i=0}^{\lfloor \log_2 n \rfloor} (2^i + (i+1)2^i)$$

$$= 2c \sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^i + 2c \sum_{i=0}^{\lfloor \log_2 n \rfloor} (i+1)2^i \leq 2c(2^{\lfloor \log_2 n \rfloor + 1} - 1) + 2c(\lfloor \log_2 n \rfloor + 1) \sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^i$$

$$< 4cn + 4c(\lfloor \log_2 n \rfloor + 1)n = O(n \log n)$$

Ciò implica che il costo *medio* di ciascuno degli inserimenti è effettivamente $O(\log n)$.

**Esercizi per gli studenti:**

1. Come cambia l'analisi se la dimensione iniziale dell'array è pari a un valore $k$?
2. Come si argomenta l'analisi considerando sia inserimenti che cancellazioni?