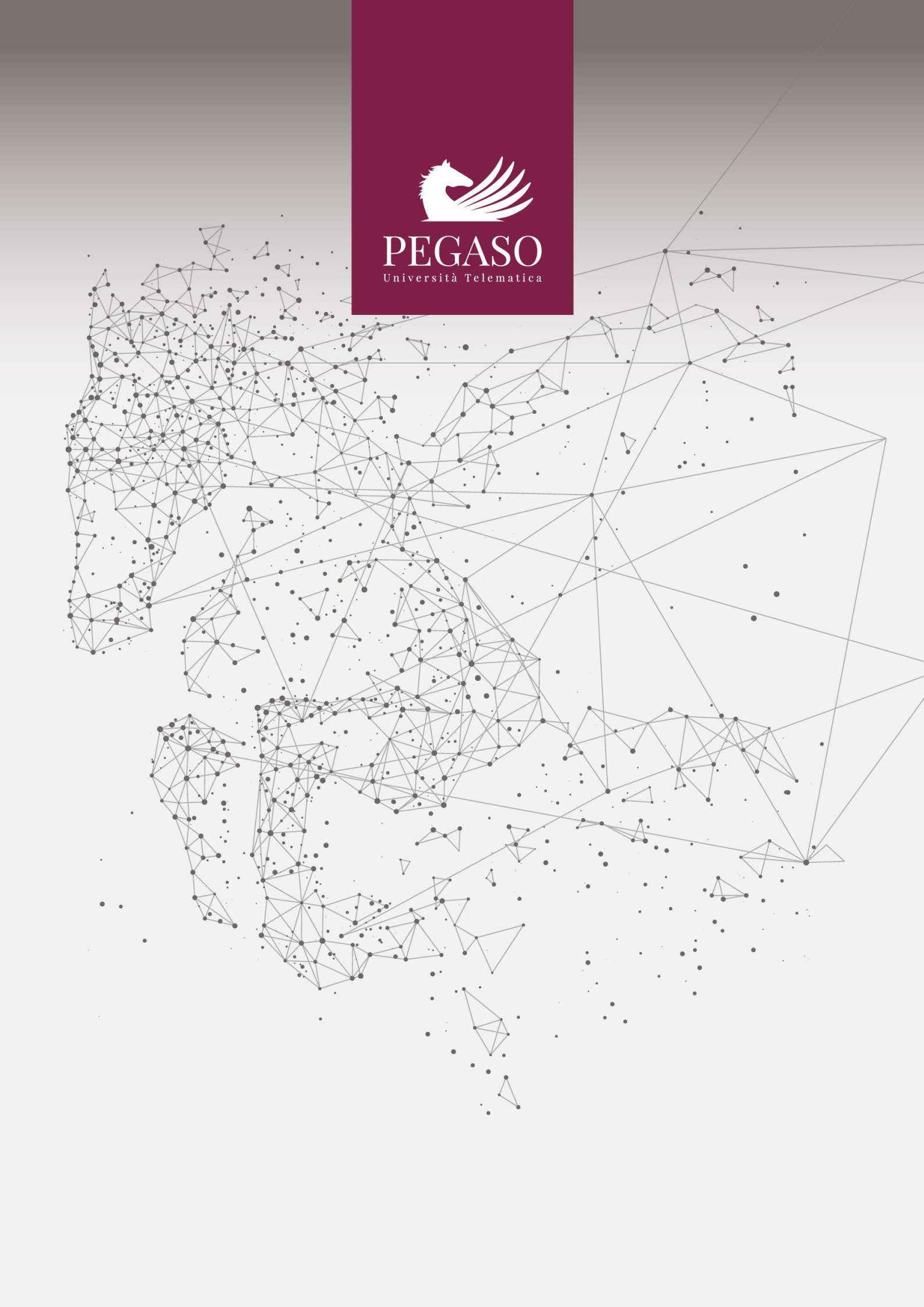

PEGASO
Università Telematica

Indice

1. Il problema dell'ordinamento

Il problema dell'ordinamento può essere definito "formalmente" nella seguente maniera: data una sequenza di n numeri $\langle a_1, a_2, \ldots, a_n \rangle$ un ordinamento è una permutazione (un ri-arrangiamento) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ della sequenza di input tale che $1 \leq a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Più semplicemente possiamo dire che l'ordinamento di una sequenza di informazioni consiste nel disporre le stesse informazioni in modo da rispettare una qualche relazione d'ordine di tipo lineare (ad esempio una relazione d'ordine "minore o uguale" dispone le informazioni in modo "non decrescente").

Oltre che per il loro principio di funzionamento e per la loro efficienza, gli algoritmi di ordinamento possono essere confrontati in base ai seguenti criteri:

- **Stabilità**: un algoritmo di ordinamento è stabile se non altera l'ordine relativo di elementi dell'array aventi la stessa chiave. Algoritmi di questo tipo evitano interferenze con ordinamenti pregressi dello stesso array basati su chiavi secondarie. Se ad esempio si ordina per anno di corso una lista di studenti già ordinata alfabeticamente, un metodo stabile produce una lista in cui gli alunni dello stesso anno sono ancora in ordine alfabetico mentre un ordinamento instabile probabilmente produrrà una lista senza più alcuna traccia del precedente ordinamento.
- **Sul posto (in place)**: un algoritmo di ordinamento opera in place se la dimensione delle strutture ausiliarie di cui necessita è indipendente dal numero di elementi dell'array da ordinare. In altre parole: un algoritmo in place non crea una copia dell'input per raggiungere l'obiettivo (l'ordinamento), pertanto un algoritmo in place risparmia memoria rispetto ad un algoritmo non in place. Si intuisce quanto sui grandi numeri la proprietà "in place" sia rilevante.

È inoltre possibile classificare in base alla complessità del tempo di calcolo. La complessità di calcolo si riferisce soprattutto al numero di operazioni necessarie all'ordinamento (principalmente operazioni di confronto e scambio), in funzione del numero di elementi da ordinare:

- **Algoritmi Semplici di Ordinamento**: algoritmi che presentano una complessità proporzionale a n^2, essendo n è il numero di informazioni da ordinare; essi sono generalmente caratterizzati da poche e semplici istruzioni.
- **Algoritmi Evoluti di Ordinamento**: algoritmi che offrono una complessità computazionale proporzionale ad $n \log_2 n$ (che è sempre inferiore a n^2). Tali algoritmi sono molto più complessi e

fanno molto spesso uso di ricorsione. La convenienza del loro utilizzo si ha unicamente quando il numero n di informazioni da ordinare è molto elevato.

Da precisare che è possibile dimostrare che il problema dell'ordinamento non può essere risolto con un algoritmo di complessità asintotica inferiore a quella pseudo-lineare: per ogni algoritmo che ordina un array di n elementi, il tempo d'esecuzione soddisfa $T(n) = \Omega(n \cdot \log_2 n)$.

Riportiamo schematicamente le caratteristiche dei principali algoritmi in termini di complessità e criteri di confronto:

Nome	***Migliore***	***Medio***	***Peggiore***	***Memoria***	***Stabile***	***In place***
Bubble sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	Sì	Sì
Heap sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$\Theta(1)$	No	Sì
Insertion sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	Sì	Sì
Merge sort	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$O(n)$	Sì	No
Quick sort	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$O(n^2)$	$O(n)$	No	Sì
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	No	Sì

Ricordiamo brevemente il significato delle notazioni asintotiche:

- **Notazione asintotica O** (notazione O grande): limite superiore asintotico
- **Notazione asintotica Ω** (notazione Omega): limite inferiore asintotico
- **Notazione asintotica θ** (notazione Theta): limite asintotico stretto (se una funzione è $\theta(g(n))$ allora è anche $O(g(n))$ e $\Omega(g(n))$)

2. Heap Sort

Heap sort è un algoritmo di ordinamento iterativo proposto da Williams nel 1964, il quale si basa su una struttura dati chiamata heap.

In origine, il termine "heap" fu coniato nel contesto dell'Heap sort, ma da allora è stato utilizzato per fare riferimento ai meccanismi automatici di recupero della memoria (garbage collector), come quelli forniti dai più comuni linguaggi di programmazione. Da precisare che in questo contesto però, l'heap non è un meccanismo garbage collector. In particolare, nel contesto della gestione della memoria, per heap si intende un grande pool di memoria che può essere usato dinamicamente ed è gestito "manualmente": occorre allocare e de-allocare esplicitamente la memoria e se non lo si fa si incorre in un **memory leak** (memoria ancora usata e non disponibile per altri processi). Non vi sono restrizioni sulla dimensione dell'heap (al netto della memoria fisica della macchina).

Riassumendo:

- l'heap è gestito dal programmatore;
- l'heap è grande e limitato solo dalla dimensione della memoria fisica;
- l'heap richiede un accesso tramite "puntatori".

Nel contesto dell'Heap sort, un heap è una struttura dati composta da un array che possiamo considerare come un **albero binario quasi completo**.

In un albero intendiamo per:

- **profondità**: la lunghezza del cammino che porta dalla radice a quel nodo (cioè il numero di archi tra la radice ed il nodo n) (Figura 1);
- **altezza** (Figura 1);
    - **del nodo**: la profondità massima di un nodo all'interno di un albero (il numero di archi dal nodo stesso fino a scendere ad una foglia);
    - **massima**: la profondità massima a partire dalla radice fino a scendere ad una foglia;
- **grado**: il numero di figli di uno specifico nodo;
- **completezza**: tutte le foglie hanno la stessa profondità e tutti i nodi interni hanno grado 2 (hanno esattamente due figli) (Figura 3);
- **quasi completezza**: tutti i livelli, tranne al più l'ultimo, sono completi (Figura 2, 4).

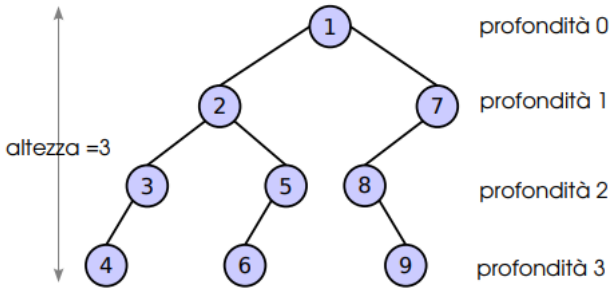


Figura 1 *- profondità di un albero*

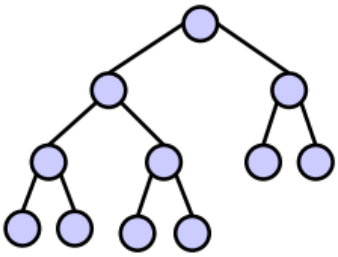


Figura 2 *- albero quasi completo*

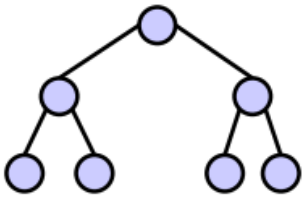


Figura 3 *- albero completo*

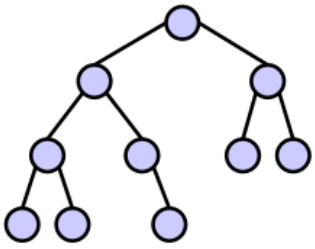


Figura 4 *- albero quasi completo*

Nell'heap, ogni nodo dell'albero corrisponde a un elemento dell'array che memorizza il valore del nodo; tutti i livelli dell'albero sono completamente riempiti, tranne eventualmente l'ultimo che può essere riempito da sinistra fino a un certo punto.

Un array A che rappresenta un heap è un oggetto con due attributi:

- $lunghezza[A]$ indica il numero di elementi nell'array;
- $heap-size[A]$ indica il numero degli elementi dell'heap che sono registrati nell'array A (anche se $A[1\,..lunghezza[A]]$ contiene numeri validi, nessun elemento dopo $A[heap-size[A]]$, dove $heap-size[A] \leq lunghezza[A]$, è un elemento dell'heap).

La radice dell'albero è $A[1]$. Se i è l'indice di un nodo, gli indici di suo padre $PARENT(i)$, del figlio sinistro $LEFT(i)$ e del figlio destro $RIGHT(i)$ possono essere così calcolati:

- $PARENT(i) \rightarrow \frac{i}{2}$
- $LEFT(i) \rightarrow 2i$
- $RIGHT(i) \rightarrow 2i+1$

Da notare che tali operazioni (parent / left / right) sono operazioni che possono essere calcolate mediante una sola istruzione.

Di seguito una rappresentazione dell'albero e del corrispondente array:

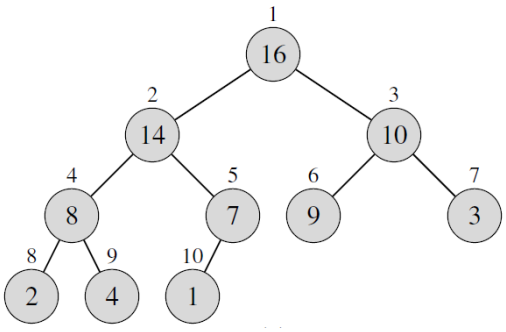


Figura 5 *- albero*

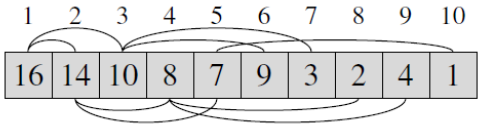


Figura 6 *- array*

In Figura 5 e 6 evidenziamo:

- il numero all'interno del cerchio di un nodo dell'albero è il valore registrato in quel nodo.
- Il numero sopra un nodo è il corrispondente indice dell'array.
- Sopra e sotto l'array ci sono delle linee che rappresentano le relazioni padre-figlio; i padri sono sempre a sinistra dei loro figli.
- L'albero ha altezza 3.
- Il nodo con indice 4 e valore 8 ha altezza 1.

Ci sono due tipi di heap binari:

- **max-heap**: per ogni i diverso dalla radice, si ha $A[PARENT(i)] \geq A[i]$; quindi, l'elemento più grande di un max-heap è memorizzato nella radice e il sottoalbero di un nodo contiene valori non maggiori di quello contenuto nel nodo stesso (in altre parole: ogni elemento è minore o uguale al nodo padre)
- **min-heap**: che per ogni nodo i diverso dalla radice, si ha $A[PARENT(i)] \leq A[i]$; il più piccolo elemento in un min-heap è nella radice.

Per l'algoritmo Heap sort è possibile usare uno dei 2 heap.

In un Heap di n elementi si ha un albero binario completo con altezza $\log_2 n$ → le operazioni che effettuano sull'albero saranno proporzionali all'altezza stessa dell'albero → $O(\log_2 n)$

Di seguito riportiamo una animazione[1] che ci dà un'idea di come opera Heap Sort:

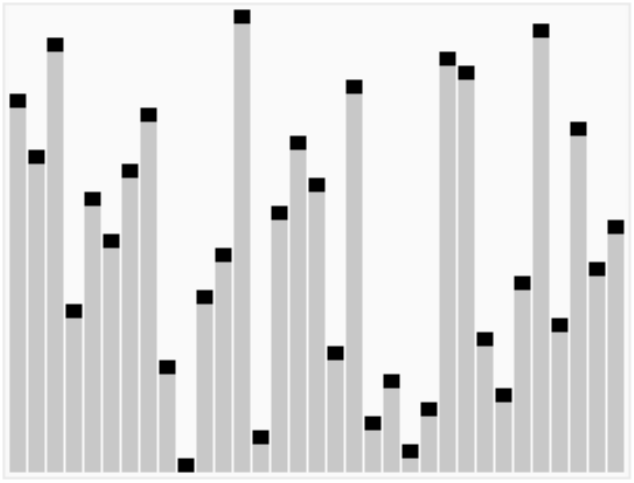


Figura 7 *- Heap Sort Animation*

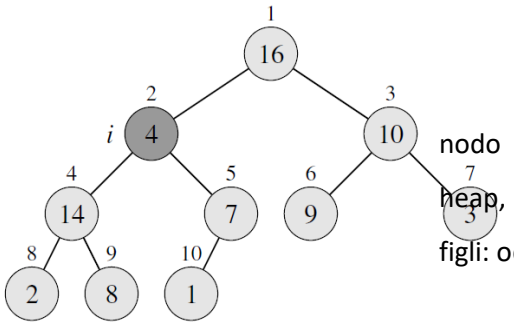
Concettualmente, l'algoritmo opera in due fasi:

1. Nella prima fase, l'algoritmo permuta i valori contenuti negli elementi dell'array in modo tale che la nuova disposizione delle chiavi costituisca uno heap; ciò viene ottenuto facendo scivolare al posto giusto i valori contenuti negli elementi $a[n/2];\ \dots\ ;\ a[1]$
2. Nel generico passo i della seconda fase, l'algoritmo vede l'array diviso in una sequenza di origine $a[1];\ \dots\ ; a[i-1]$ organizzata ad heap e una sequenza di destinazione $a[i];\ \dots\ ;\ a[n]$ già ordinata. Poiché $a[1]$ contiene il valore massimo presente nella sequenza di origine, il valore di $a[1]$ viene scambiato con il valore di $a[i-1]$ in modo da metterlo subito prima dell'inizio della sequenza di destinazione. Facendo poi scivolare al posto giusto il nuovo valore di $a[1]$, si garantisce che le chiavi dei valori contenuti negli elementi della sequenza di origine ridotta di un elemento costituiscano ancora uno heap.

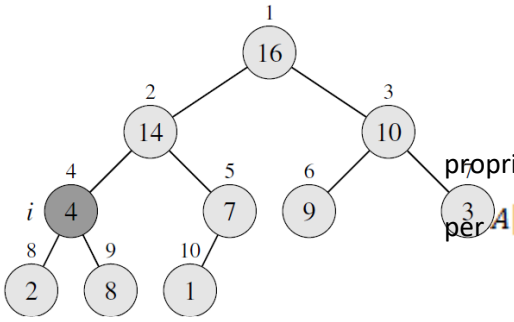
[1] Sorting heapsort anim - Heapsort - Wikipedia

3. Procedura di calcolo

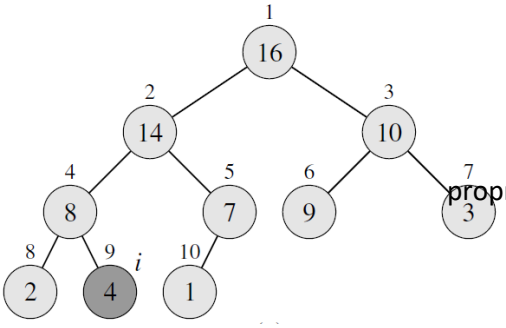
Vediamo da un punto di vista operativa come viene fatto "scivolare" l'elemento che non si trova nella posizione giusta:



La configurazione di $A[2]$ (cioè del nodo con $i = 2$) viola la proprietà del max-heap, in quanto non è più grande di entrambi i figli: occorre farlo "scivolare"



Scambiando $A[2]$ con $A[4]$, la proprietà è ripristinata per $A[2]$ ma non lo è per $A[4]$: occorre farlo "scivolare" ancora



Scambiando $A[4]$ con $A[9]$, la proprietà è ripristinata per tutti i nodi

Consideriamo il seguente array: [42, 38, 11, 75, 99, 23, 84, 67]

L'albero corrispondente è:

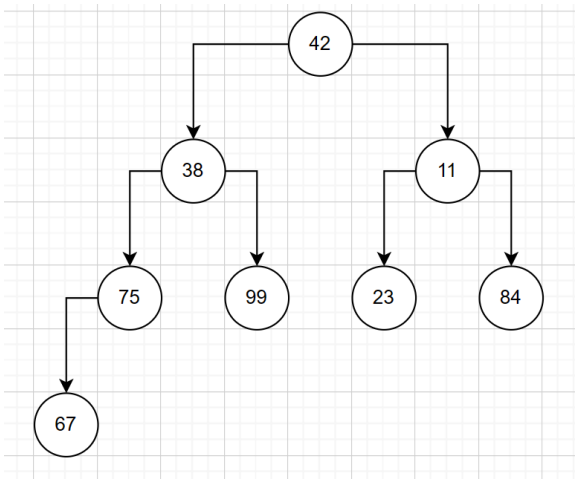
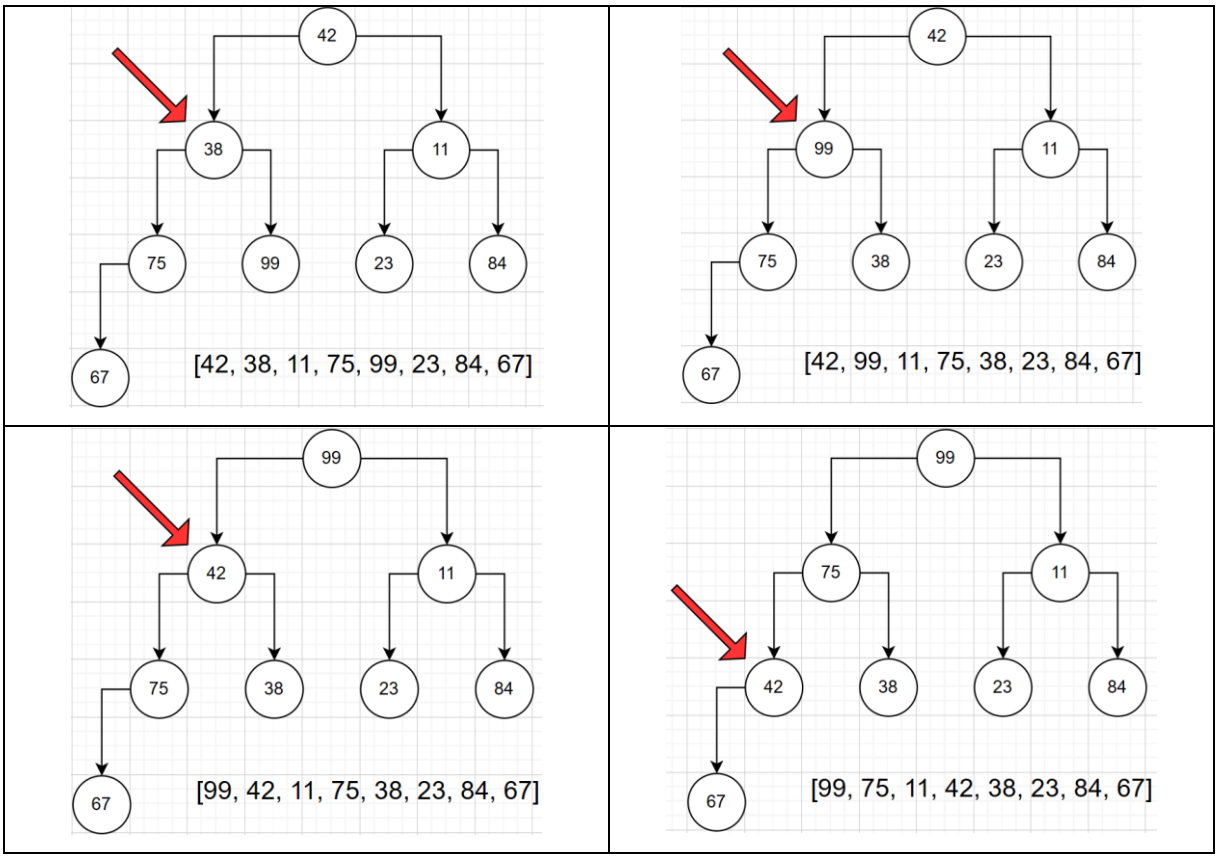
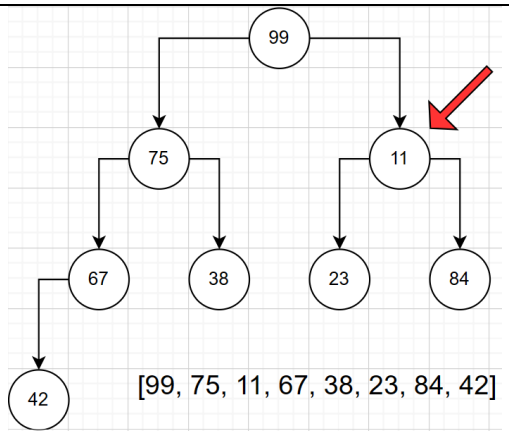
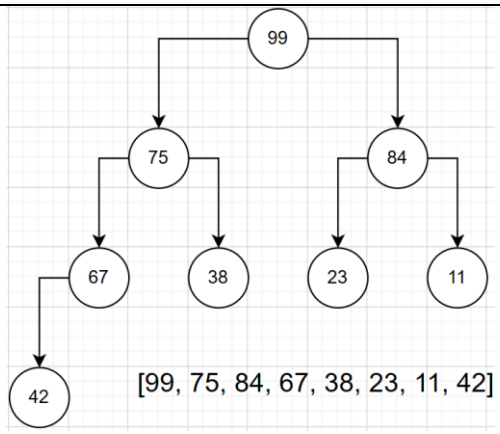


*Figura 8 - **Albero Binario***

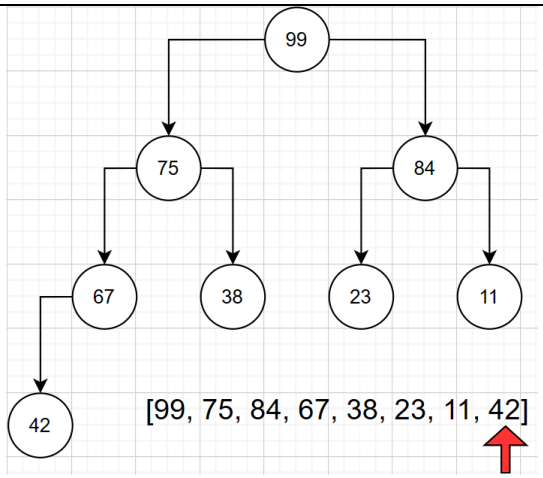
Consideriamo ora la prima fase di "trasformazione" in albero heap (la freccia indica l'elemento fuori posto):



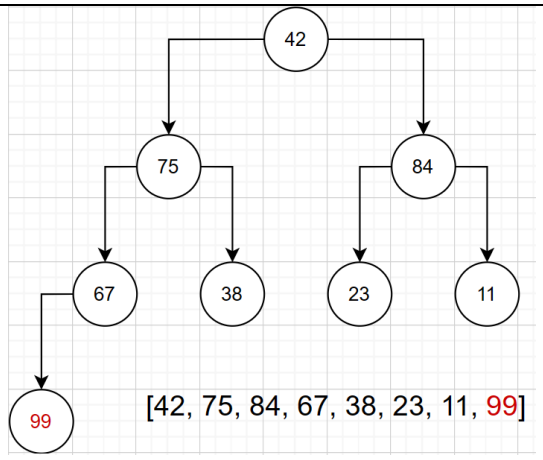




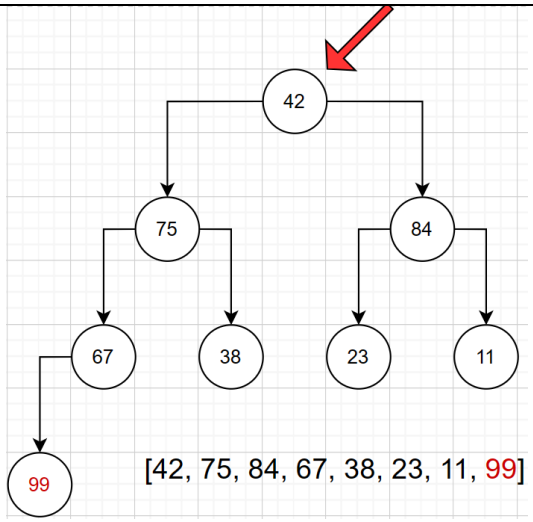
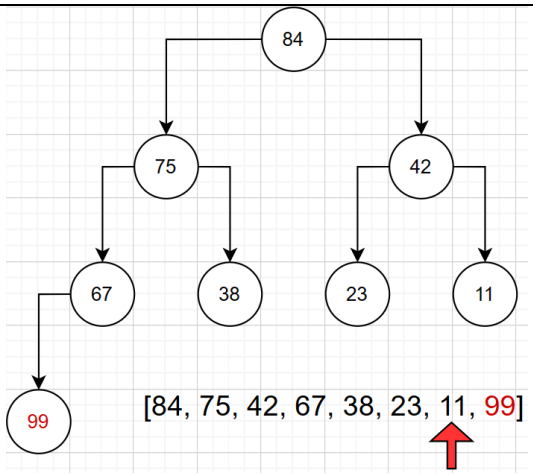
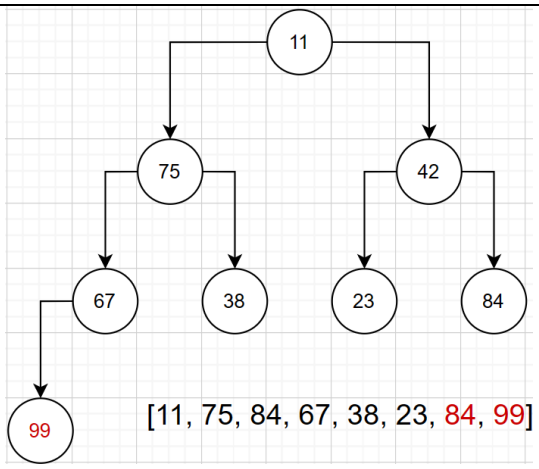
Finita questa prima fase inizia la seconda; analizziamo l'inizio di questa seconda fase:



Sostituisco l'ultimo elemento del vettore con il primo: l'ultimo elemento del vettore sarà dunque ordinato



Devo analizzare in questa nuova configurazione se è rispettata la condizione di albero heap

	Tale condizione non è rispettata per il A[1] con valore 42 e quindi procedo per il ripristino dell'albero
	Ripristinato l'albero procedo con la sostituzione del penultimo elemento con il primo
	Il nuovo array ha quindi le ultime 2 posizioni ordinate. Devo dunque procedere fino a ri-ordinare il tutto

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley
- Sedgewick: Algoritmi in C, Pearson, 2015
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione