Code con priorità

ADT Coda con priorità

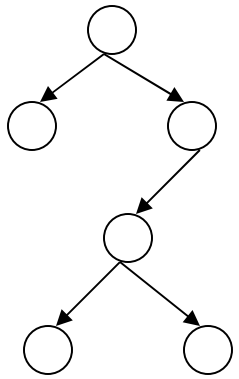
- Una coda con priorità è una struttura dati dinamica che permette di gestire una collezione di dati con chiave numerica.
- Una coda con priorità offre le operazioni di
  - inserimento: di un elemento nell'insieme
  - massimo: restituisce l'elemento con chiave più grande
  - cancellazione-massimo: restituisce l'elemento con chiave più grande e lo rimuove dalla collezione

Applicazioni della Coda con Priorità

- Le Code con priorità sono strutture dati molto comuni in informatica.
- Es:
  - *Gestione di processi*: ad ogni processo viene associata una priorità. Una coda con priorità permette di conoscere in ogni istante il processo con priorità maggiore. In qualsiasi momento i processi possono essere eliminati dalla coda o nuovi processi con priorità arbitraria possono essere inseriti nella coda.
- Per implementare efficientemente una coda con priorità utilizzeremo una struttura dati chiamata *heap*

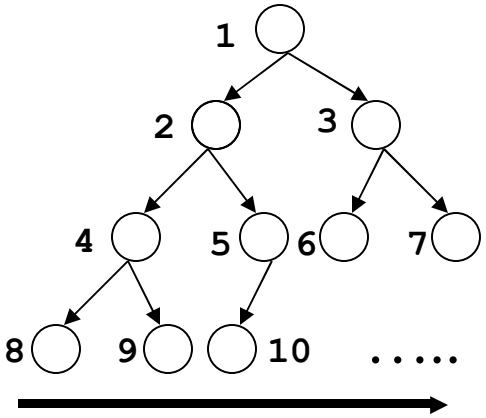
Nota su gli alberi binari

- Intuitivamente (*vedremo meglio successivamente*) un albero binario è una struttura dati formata da nodi collegati fra di loro (come per la struttura dati lista)
- Come per una lista, per ogni nodo esiste un unico nodo predecessore
- A differenza di una lista, ogni nodo è collegato con uno *o due* nodi successori



Heap Binario

- La struttura dati *heap binario* è un albero binario *quasi completo*
- Un albero binario *quasi completo* è un albero binario riempito completamente su tutti i livelli tranne eventualmente l'ultimo che è riempito da sinistra a destra

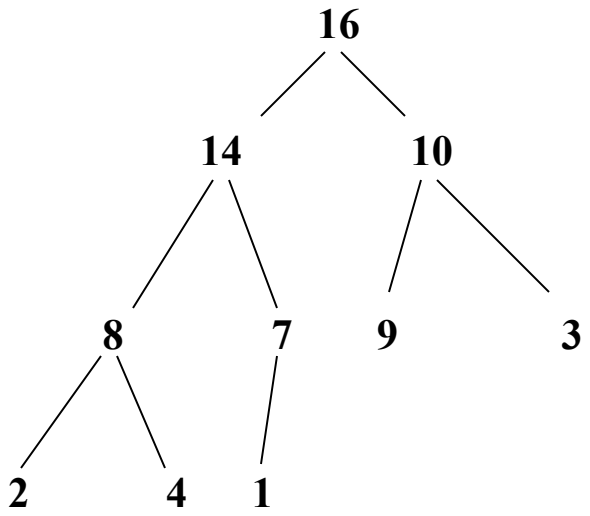


Proprieta' Heap

- Perché un albero binario quasi completo sia uno heap deve valere la seguente:

  Proprietà dell'ordinamento parziale dello heap

  il valore di un nodo figlio (successore) è minore o uguale a quello del nodo padre (predecessore)



Implementazione Heap tramite Vettore

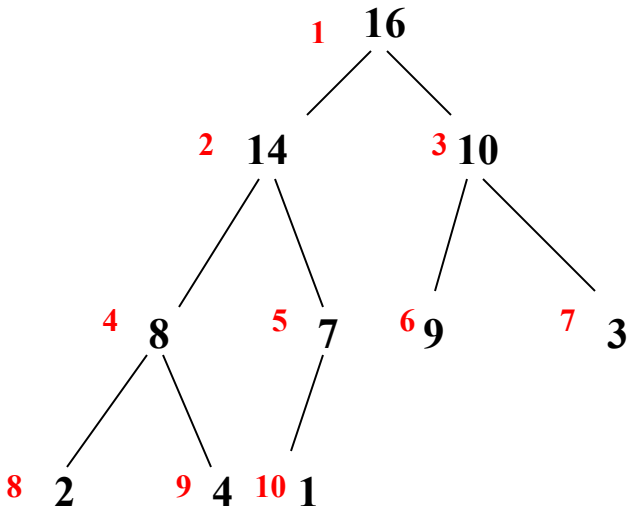
- Si implementa l'albero tramite un vettore di lunghezza length[A]
- Uno heap A ha un attributo heap-size[A] che specifica il numero di elementi contenuto nello heap
- Nessun elemento in A[1,length[A]] dopo heap-size[A] è un elemento valido dello heap

Codifica di un albero binario quasi completo con una struttura dati vettore

- La radice dell'albero è A[1]
- L'indice del padre di un nodo di posizione i è $\lfloor i/2 \rfloor$
- L'indice del figlio sinistro di un nodo i è 2 i
- L'indice del figlio destro di un nodo i è 2 i +1

Visualizzazione di uno heap

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



Pseudocodice Operazioni Heap

```
Parent(i)
1 return  i/2

Left(i)
1 return 2 i

Right(i)
1 return 2 i + 1
```

Mantenimento proprietà heap

- A seguito di varie operazioni sullo heap può accadere che un nodo violi la proprietà dello heap
- La procedura Heapify prende in ingresso uno heap A e l'indice *i* di un nodo che potenzialmente viola la proprietà e ristabilisce la proprietà di ordinamento parziale sull'intero heap
- Si assume che i sottoalberi figli del nodo *i* siano radici di heap che rispettano la proprietà di ordinamento parziale

Spiegazione

- L'idea è di far "affondare" il nodo che viola la proprietà di ordinamento parziale fino a che la proprietà non viene ripristinata
- Per fare questo si determina il nodo figlio più grande e si scambia il valore della chiave fra padre e figlio
- Poi si procede ricorsivamente sul nodo figlio per cui e' avvenuto lo scambio

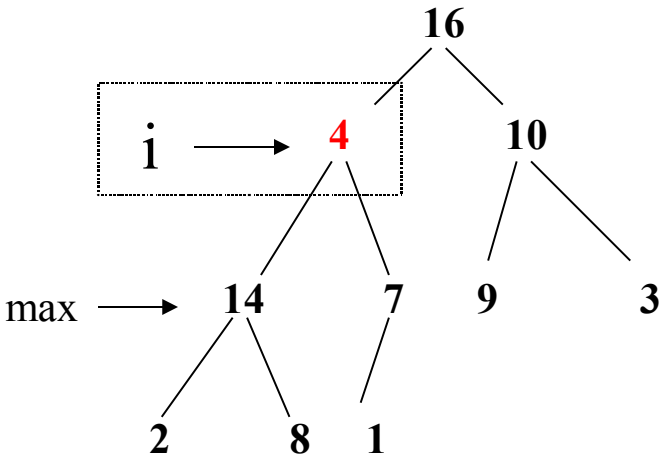
Pseudocodice Heapify (semplificato)

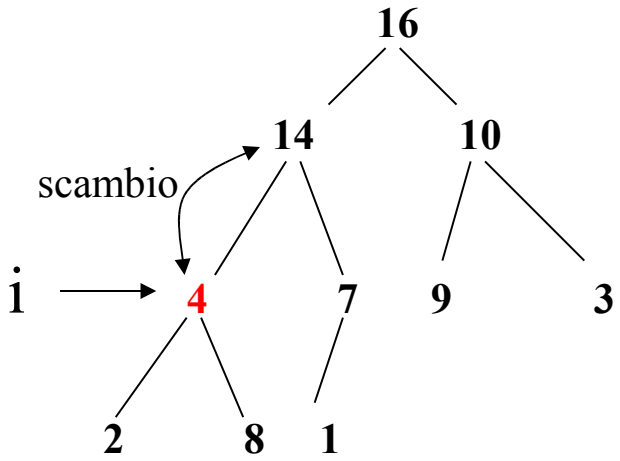
```
Heapify(A,i)
1 l ← Left(i)
2 r ← Right(i)
3 largest ← Max between A[l],A[r],A[i]
4 if largest ≠ i
5      then   scambia A[i] ↔ A[largest]
6             Heapify(A,largest)
```

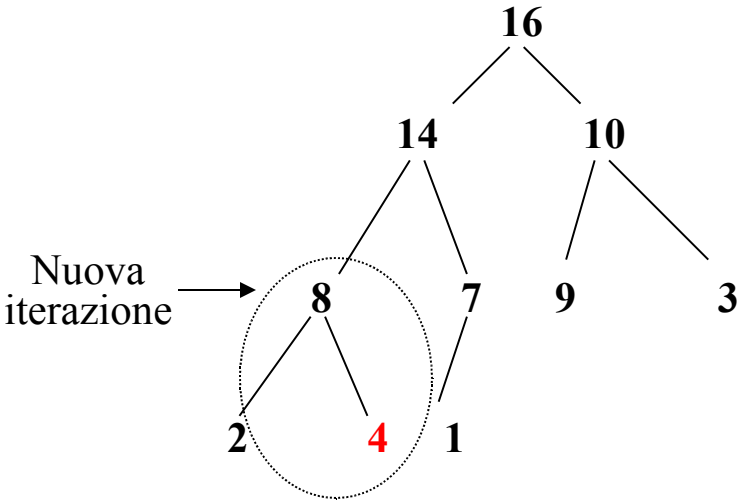
Pseudocodice Heapify

```
Heapify(A,i)
1 l ← Left(i)
2 r ← Right(i)
3 if l ≤ heap-size[A] e A[l]>A[i]
4      then   largest ← l
5      else   largest ← i
6 if r ≤ heap-size[A] e A[r]>A[largest]
7      then   largest ← r
8 if largest ≠ i
9      then   scambia A[i] ↔ A[largest]
10            Heapify(A,largest)
```

Visualizzazione Procedura Heapify







Nota sugli alberi binari

- Un albero binario completo di altezza h ha 2^{h+1}-1 nodi
- Infatti intuitivamente:
  - un albero binario completo di altezza 0 ha un unico nodo: la radice
  - un albero binario completo di altezza 1 è composto dalla radice e dai suoi due figli
  - all'aumentare di un livello ogni figlio genera altri due figli e quindi si raddoppia il numero di nodi del livello precedente
- Ogni livello di un albero binario completo contiene tanti nodi quanti sono contenuti in tutti i livelli precedenti +1
  - infatti passando da h a h+1 si passa da 2^{h+1}-1 a 2^{h+2}-1 nodi ovvero a $2(2^{h+1})$ -1 nodi quindi il nuovo livello ha aggiunto 2^{h+1} nodi ad un albero che prima ne conteneva 2^{h+1}-1

Tempo di calcolo di Heapify

- Le istruzioni per determinare il maggiore fra i, l e r impiegano un tempo $\Theta(1)$
- Ricorsivamente si chiama Heapify su uno dei sottoalberi radicati in l o r
- Il sottoalbero di un figlio ha al più dimensione 2n/3
  - il caso peggiore è quando l'ultimo livello è pieno per metà ovvero quando uno dei sottoalberi (quello in cui proseguirà la ricorsione) è completo
- Il tempo di esecuzione è pertanto:
- $T(n)=T(2n/3)+ \Theta(1)$

Tempo di calcolo: Teorema principale

- Per il Teorema Principale si ha $T(n)= \Theta(\lg n)$ infatti
- $T(n)=T(2n/3)+1$
  - $f(n)=1$, $a=1$, $b=3/2$
  - $n^{\log_b a} = n^{\log_b 1} = n^0 = \Theta(1)$
  - pertanto, dato che $f(n)=\Theta(n^{\log_b a})=\Theta(1)$ allora (caso 2)
  - $T(n)= \Theta(\lg n)$

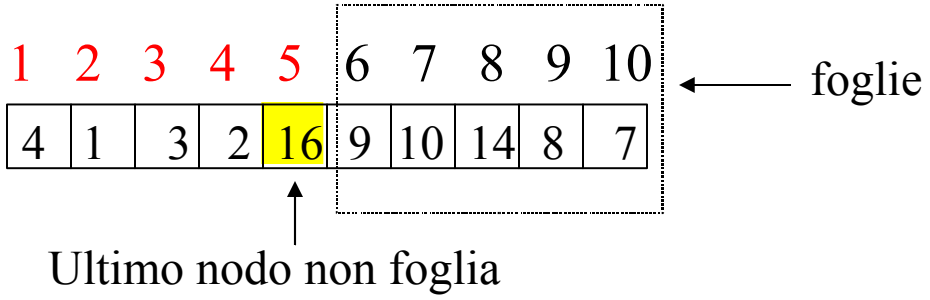
Costruzione di uno heap

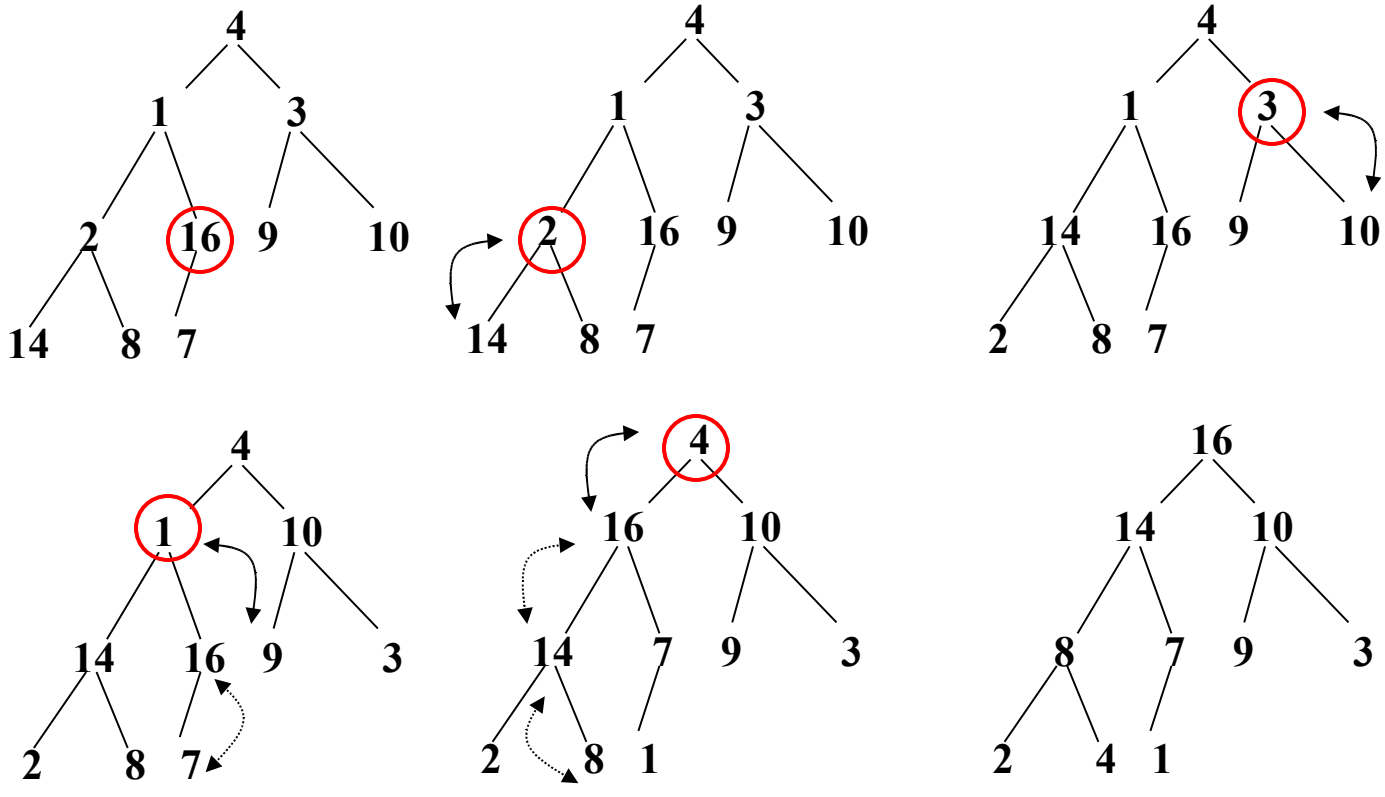
- Si può usare la procedura Heapify in modo bottom-up, cioè a partire dai livelli più bassi dell'albero, per convertire un Array in uno Heap
- Gli elementi A[$\lfloor n/2 \rfloor + 1$ .. n] sono tutte foglie dell'albero e pertanto ognuno di essi è già uno heap di 1 elemento
- Si inizia dagli elementi padri dei nodi A[$\lfloor n/2 \rfloor + 1$ .. n]
- Dato che procediamo in modo bottom-up allora sicuramente i sottoalberi di un nodo sottoposto a Heapify sono heap

Pseudocodice Costruzione Heap

```
Build-Heap(A)
1 heap-size[A]← Length[A]
2 for i ←⌊length[A]/2⌋ downto 1
3      do Heapify(A,i)
```

Visualizzazione Costruzione Heap





Coda con priorità con heap

- Risulta semplice implementare le varie operazioni di una coda con priorità utilizzando uno heap
  - Extract Max: basta restituire la radice dello heap
  - Heap Extract Max: dopo la restituzione dell'elemento massimo, posiziona l'ultimo elemento dello heap (non il piu' piccolo!) nella radice ed esegue Heapify per ripristinare la proprietà di ordinamento parziale
  - Heap Insert: la procedura inserisce il nuovo elemento come elemento successivo all'ultimo e lo fa salire fino alla posizione giusta facendo "scendere" tutti padri

Pseudo codice operazioni

```
Heap-Extract-Max(A)
1 max ← A[1]
2 A[1] ← A[heap-size[A]]
3 heap-size[A] ← heap-size[A]-1
4 Heapify(A,1)
5 return max

Heap-Insert(A,key)
1 heap-size[A] ← heap-size[A]+1
2 i ← heap-size[A]
3 while i>1 e A[Parent(i)]<key
4       do      A[i] ← A[Parent(i)]
5               i ← Parent(i)
6 A[i] ← key
```

Visualizzazione Heap Insert