Capitolo 3
Pile e code

Lucidi tratti da
P. Crescenzi · G. Gambosi · R. Grossi · G. Rossi
Strutture di dati e algoritmi
Progettazione, analisi e visualizzazione
Addison-Wesley, 2012
`http://algoritmica.org`

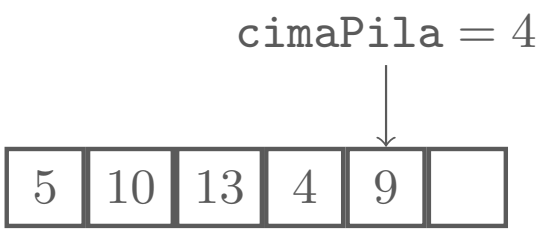
1. I lucidi sono utilizzabili dai soli docenti e se ne sconsiglia la distribuzione agli studenti: oltre al rischio di violare una qualche forma di copyright, il problema principale è che gli studenti studino in modo superficiale la materia senza il necessario approfondimento e la dovuta riflessione che la lettura del libro fornisce
2. Il simbolo [alvie] nei lucidi indica l'uso di ALVIE per visualizzare il corrispettivo algoritmo: per un proficuo rendimento dello strumento, conviene esaminare in anticipo la visualizzazione per determinare i punti salienti da mostrare a lezione (l'intera visualizzazione potrebbe risultare altrimenti noiosa)

Pile

- Collezioni di elementi in cui le operazioni disponibili, come l'estrazione di un elemento, sono ristrette unicamente a quello più recentemente inserito
- Politica di accesso **Last In First Out** (LIFO): l'ultimo elemento inserito è il primo ad essere estratto
- Operazioni:
  1. **Push**: inserisce un nuovo elemento in cima alla pila
  2. **Pop**: estrae l'elemento in cima alla pila restituendo l'informazione in esso contenuta
  3. **Top**: restituisce l'informazione contenuta nell'elemento in cima alla pila senza estrarlo
  4. **Empty**: verifica se la pila è vuota o meno

Pile e array

- Elementi della pila memorizzati in un array di dimensione iniziale predefinita
- Array ridimensionato per garantire che la dimensione sia proporzionale al numero di elementi effettivamente nella pila
- Elementi memorizzati in sequenza nell'array a partire dalla locazione iniziale, inserendoli man mano nella prima locazione disponibile
- La cima della pila corrisponde all'ultimo elemento della sequenza



Pile e array

```
1  Push( x ):
2    VerificaRaddoppio( );
3    cimaPila = cimaPila + 1;
4    pilaArray[ cimaPila ] = x;
```

```
1  Pop( ):
2    IF (!Empty( )) {
3      x = pilaArray[ cimaPila ];
4      cimaPila = cimaPila - 1;
5      VerificaDimezzamento( );
6      RETURN x;
7    }
```

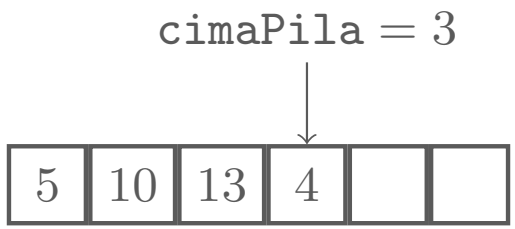
```
1  Top( ):
2    IF (!Empty( )) RETURN pilaArray[ cimaPila ];
```

```
1  Empty( ):
2    RETURN (cimaPila == -1);
```

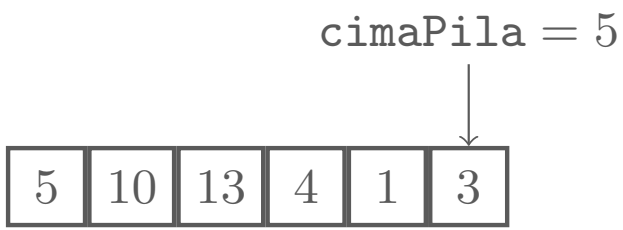
[alvie]

Pile e array

Dopo una **Pop**

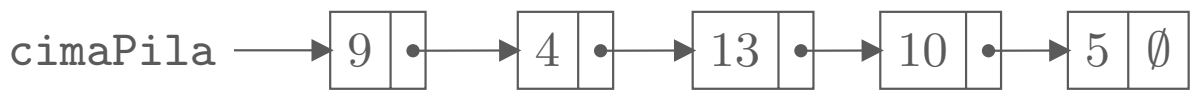


Dopo **Push**(1) e **Push**(3)



Pile e liste

- Elementi della pila memorizzati in una lista ordinata per tempo di inserimento decrescente
- La cima della pila corrisponde all'inizio della lista
- Le operazioni agiscono tutte sull'elemento iniziale della lista



Pile e liste

```
1  Push( x ):
2    u = NuovoNodo( );
3    u.dato = x;
4    u.succ = cimaPila;
5    cimaPila = u;
```

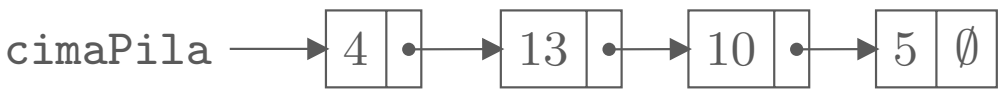
```
1  Pop( ):
2    IF (!Empty( )) {
3      x = cimaPila.dato;
4      cimaPila = cimaPila.succ;
5      RETURN x;
6    }
```

```
1  Top( ):
2    IF (!Empty( )) RETURN cimaPila.dato;
```

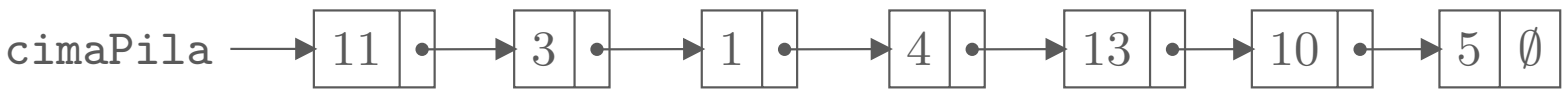
```
1  Empty( ):
2    RETURN (cimaPila == null);
```

[alvie]

Dopo una **Pop**



Dopo **Push**(1), **Push**(3) e **Push**(11)



Postscript e notazione postfissa

- **Postscript**: linguaggio di programmazione per la grafica, eseguito da un interprete
- Usa la notazione postfissa o polacca inversa
- L'interprete usa una pila: se un'operazione in Postscript ha k argomenti, questi ultimi si trovano nelle k posizioni in cima alla pila
- L'esecuzione di k operazioni **Pop** fornisce gli argomenti all'operazione in Postscript, il cui risultato viene posto sulla pila tramite un'operazione **Push**

Notazione postfissa

Utilizzata per scrivere espressioni aritmetiche e algebriche

- Notazione infissa: operatore tra gli operandi: $A + B$
- Notazione postfissa: operatore dopo gli operandi: $AB+$

Le espressioni postfisse possono essere valutate semplicemente, da sinistra a destra, mediante l'uso di una pila.

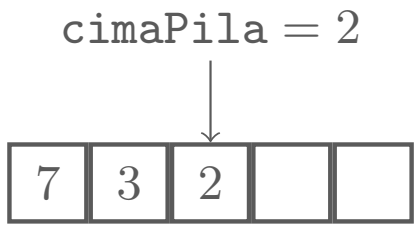
Per operazioni binarie:

- operando: viene eseguita la **Push** di tale operando;
- operatore: vengono eseguite due **Pop**, l'operatore viene applicato ai due operandi prelevati dalla pila (nel giusto ordine) e viene eseguita la **Push** del risultato.

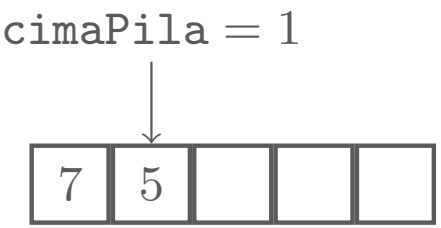
Notazione postfissa

Espressione $7\ 3\ 2\ +\ \times$

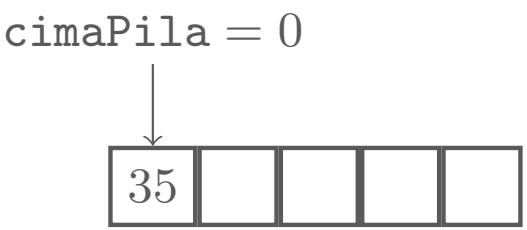
Dopo lettura operandi:



Dopo valutazione $+$



Dopo valutazione $\times$



Conversione di espressioni infisse in postfisse

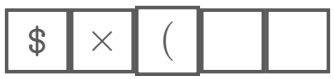
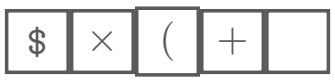
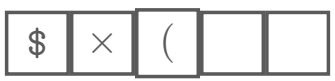
L'espressione infissa viene letta da sinistra a destra. La postfissa viene costruita man mano. Uso di una pila.

- Simbolo letto: operando. Viene appeso in fondo alla postfissa
- Simbolo letto: operatore. Si applica una delle regole seguenti.
  1. **Push** del simbolo corrente, si passa al simbolo successivo;
  2. **Pop** e l'operatore ottenuto viene appeso in fondo all'espressione postfissa;
  3. simbolo corrente ignorato nell'espressione infissa, viene eseguita una **Pop** scartando il simbolo (è una "("), si passa al simbolo successivo;
  4. la conversione ha avuto termine, simbolo corrente viene cancellato e viene eseguita una **Pop** scartando il simbolo (è un "$").

Selezione della regola, dipende dal simbolo letto e dalla cima della pila:

	simbolo corrente dell'espressione infissa				
cima della pila	$	$+$ oppure $-$	$\times$ oppure $/$	(	)
$	4	1	1	1	
$+$ oppure $-$	2	2	1	1	2
$\times$ oppure $/$	2	2	2	1	2
(		1	1	1	3

Conversione di espressioni infisse in postfisse

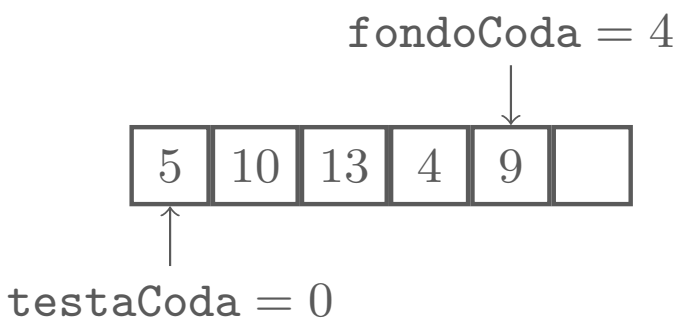
p ↓ $2 \times (6 + 4) \$$		2
p ↓ $2 \times (6 + 4) \$$		2
p ↓ $2 \times (6 + 4) \$$		$2\ 6$
p ↓ $2 \times (6 + 4) \$$		$2\ 6\ 4$
p ↓ $2 \times (6 + 4) \$$		$2\ 6\ 4\ +$
p ↓ $2 \times (6 + 4) \$$		$2\ 6\ 4\ +$
p ↓ $2 \times (6 + 4) \$$		$2\ 6\ 4\ +\ \times$

Code

- Collezioni di elementi in cui l'estrazione di un elemento, viene effettuata in "testa" alla coda, relativamente all'elemento presente da più tempo, mentre l'inserimento di un nuovo elemento viene effettuata in fondo alla coda stessa
- Politica di accesso **First In First Out** (FIFO): il primo elemento inserito è il primo ad essere estratto
- Operazioni:
  1. **Enqueue**: inserisce un nuovo elemento in fondo alla coda
  2. **Dequeue**: estrae l'elemento in testa alla coda restituendo l'informazione in esso contenuta
  3. **First**: restituisce l'informazione contenuta nell'elemento in testa alla coda senza estrarlo
  4. **Empty**: verifica se la coda è vuota o meno

Code e array

- Elementi della coda memorizzati in un array di dimensione iniziale predefinita
- Array ridimensionato per garantire che la dimensione sia proporzionale al numero di elementi effettivamente nella coda
- Elementi memorizzati in sequenza nell'array a partire dalla locazione iniziale, inserendoli man mano nella prima locazione disponibile
- La testa della coda corrisponde al primo elemento della sequenza
- Il fondo della coda corrisponde all'ultimo elemento della sequenza
- Gestione "circolare" della coda



Code e array

```
1  Enqueue( x ):
2    VerificaRaddoppio( );
3    cardCoda = cardCoda + 1;
4    fondoCoda = (fondoCoda + 1) % lunghezzaArray;
5    codaArray[ fondoCoda ] = x;
```

```
1  Dequeue( ):
2    IF (!Empty( )) {
3      cardCoda = cardCoda - 1;
4      x = codaArray[ testaCoda ];
5      testaCoda = (testaCoda + 1) % lunghezzaArray;
6      VerificaDimezzamento( );
7      RETURN x;
8    }
```

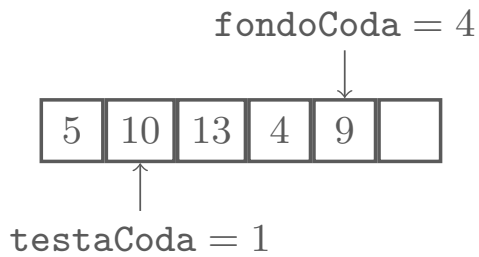
```
1  First( ):
2    IF (!Empty( )) RETURN codaArray[ testaCoda ];
```

```
1  Empty( ):
2    RETURN (cardCoda == 0);
```

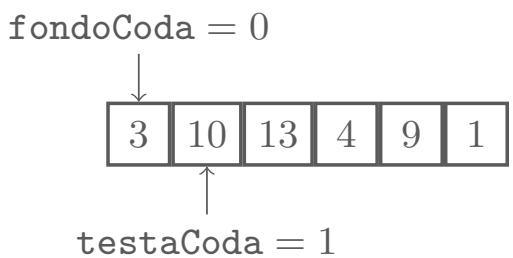
[alvie]

Code e array

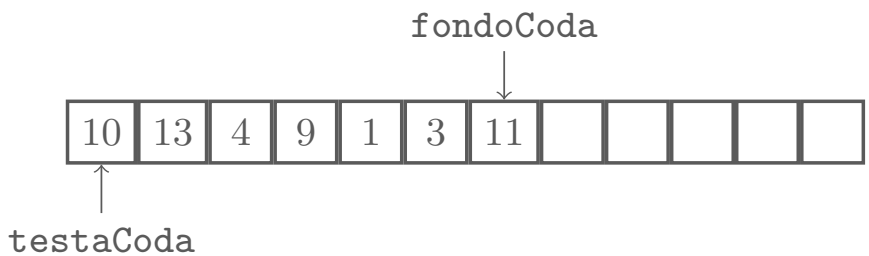
Dopo una **Dequeue**



Dopo **Enqueue**(1) e **Enqueue**(3)



Dopo **Enqueue**(11)



Code e liste

- Nodi concatenati e ordinati in modo crescente secondo l'istante di inserimento
- Il primo nodo della sequenza corrisponde alla "testa" della coda ed è il nodo da estrarre nel caso di una **Dequeue**
- L'ultimo nodo corrisponde al "fondo" della coda ed è il nodo a cui concatenare un nuovo nodo, inserito mediante **Enqueue**

Code di priorità

- Collezioni di elementi in cui a ogni elemento è associato un valore (**priorità**) appartenente a un insieme totalmente ordinato (solitamente l'insieme degli interi positivi).
- Estensione della coda: le operazioni sono le stesse della coda: **Empty**, **Enqueue**, **First** e **Dequeue**.
- **First** e **Dequeue** restituiscono l'elemento di priorità massima (o minima).

Code di priorità e liste

- Prima soluzione: **lista non ordinata**.
  - La **Enqueue** richiede tempo costante, con i nuovi elementi inseriti a un estremo della lista
  - La **Dequeue** e la `First` richiedono tempo $O(n)$: è necessario individuare l'elemento di priorità massima all'interno della lista.
- Seconda soluzione: **lista ordinata**.
  - La **Dequeue** e la `First` richiedono tempo costante: l'elemento di massima priorità è in cima alla lista.
  - La **Enqueue** richiede tempo $O(n)$: i nuovi elementi vanno inseriti alla posizione corretta rispetto all'ordinamento.

Il costo delle operazioni è sbilanciato: necessità di un implementazione che lo renda più simile.

Code di priorità e heap

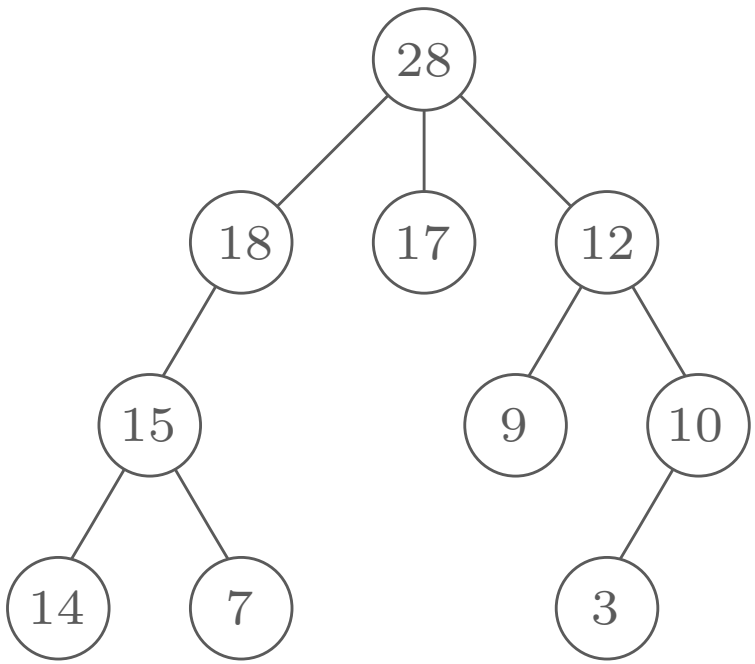
Definizione di **heaptree**

1. albero vuoto
2. albero H che soddisfa la proprietà di heap, dove r indica la radice di H e $v_0, v_1, \ldots, v_{k-1}$ i suoi figli:
   1. l'elemento contenuto nella radice di H ha priorità maggiore o uguale di quella degli elementi nei figli della radice
   2. per ogni figlio v della radice di H, l'albero di radice v è uno heaptree.

Osservazione: la radice contiene l'elemento di priorità massima dell'insieme.

La ricerca dell'elemento di massima priorità richiede tempo costante.

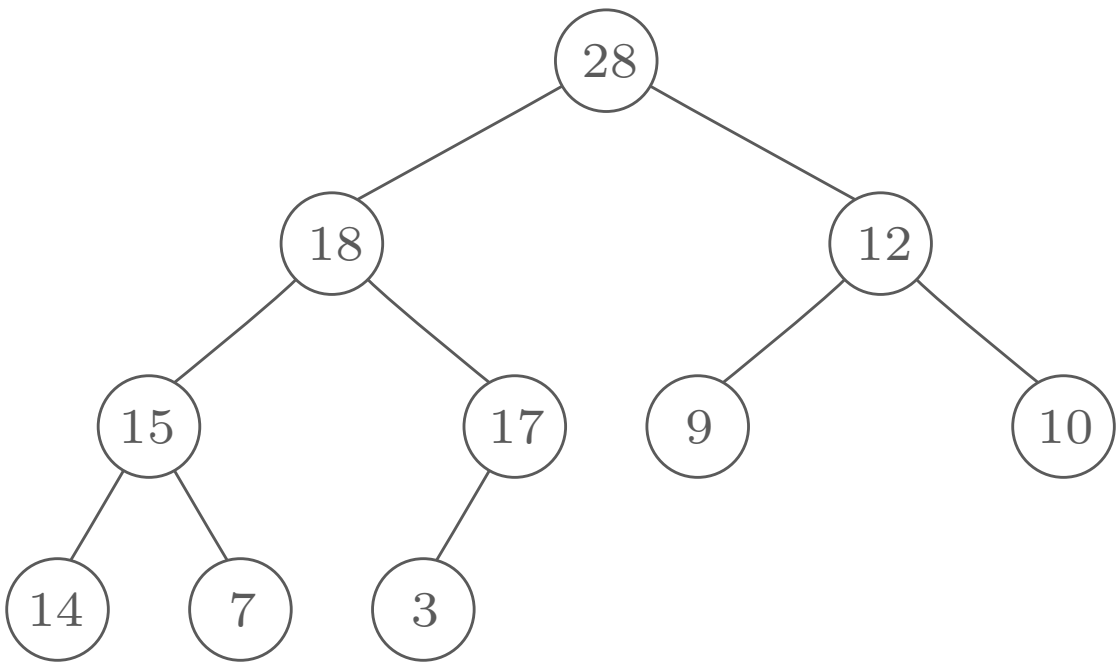
Esempio di **heaptree**



Code di priorità e heap

Uno **heap** è uno heaptree con i vincoli aggiuntivi di essere binario e completo a sinistra

Conseguenza: uno heap con n nodi ha altezza pari a $h = O(\log n)$.



Enqueue su heap

1. Viene creato un nodo v contenente il nuovo elemento e
2. Il nodo v viene inserito come foglia di H in modo da mantenere l'heap completo a sinistra.
3. Iterativamente, la priorità di e viene confrontata con quella dell'elemento f contenuto nel padre di v: se $e.\texttt{prio} > f.\texttt{prio}$, i due nodi vengono scambiati.
4. L'iterazione termina quando v diventa la radice oppure quando $e.\texttt{prio} \leq f.\texttt{prio}$.

- Numero di passi effettuati proporzionale al numero di elementi con i quali e viene confrontato
- Al più un confronto per ogni livello dell'heap
- Heap ha altezza $O(\log n)$

Quindi, **Enqueue** effettua $O(\log n)$ passi

Dequeue su heap

1. La radice di H viene rimossa e l'elemento in essa contenuto restituito.
2. L'ultima foglia di H (quella più a destra), viene inserita come radice v.
3. Iterativamente, la priorità dell'elemento in v viene confrontata con quelle degli elementi nei suoi figli: se la priorità massima fra le tre non è quella di v, il nodo v viene scambiato con il figlio contenente l'elemento di priorità massima.
4. L'iterazione termina se v diventa una foglia o se contiene un elemento di priorità è maggiore di quelle degli elementi contenuti nei suoi figli.

- Numero di passi effettuati proporzionale al numero di elementi con i quali e viene confrontato
- Al più due confronti per ogni livello dell'heap
- Heap ha altezza $O(\log n)$

Quindi, **Dequeue** effettua $O(\log n)$ passi

Heap implicito

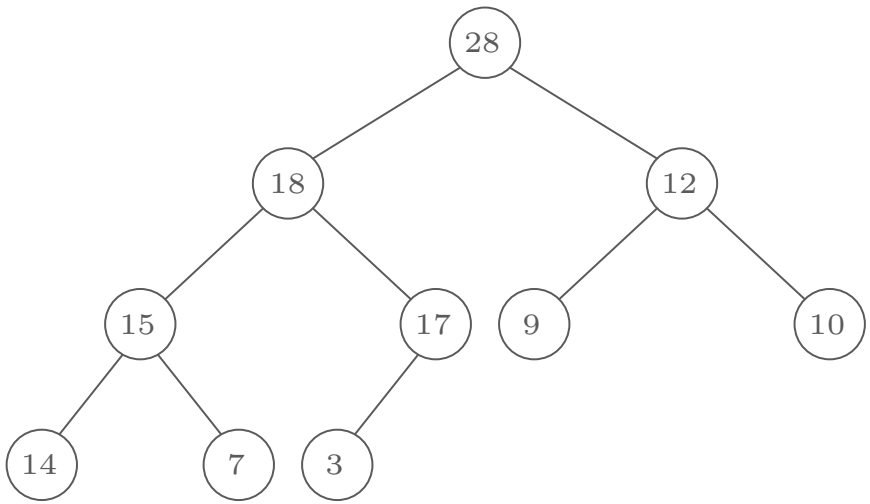
La relazione tra i nodi di un albero completo a sinistradi n nodi può essere rappresentata in modo **implicito** utilizzando un array di n posizioni.

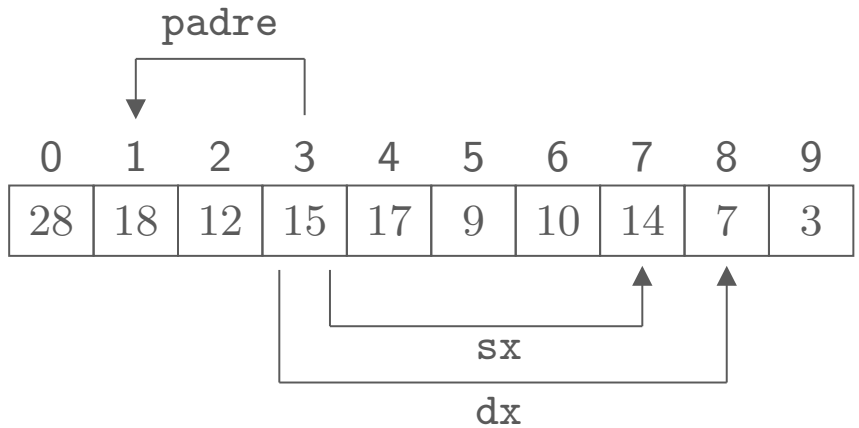
Regola di posizionamento

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

Heap implicito

Esempio di heap implicito





- Il padre di un nodo che occupa la posizione i occupa la posizione $\lfloor (i-1)/2 \rfloor$.
- Se $i = 0$, allora `u.padre` $=$ `null`.
- Se $2i + 1 \geq n$, allora `u.sx` $=$ `null`.
- Se $2i + 2 \geq n$, allora `u.dx` $=$ `null`.

Operazioni su un heap implicito

```
1  Empty( ):
2    RETURN heapSize == 0;
```

```
1  First( ):
2    IF (!Empty( )) RETURN heapArray[0];
```

```
1  Enqueue( e ):
2    VerificaRaddoppio( );
3    heapArray[heapSize] = e;
4    heapSize = heapSize + 1;
5    RiorganizzaHeap( heapSize - 1 );
```

```
1  Dequeue( ):
2    IF (!Empty( )) {
3      massimo = heapArray[0];
4      heapArray[0] = heapArray[heapSize - 1];
5      heapSize = heapSize - 1;
6      RiorganizzaHeap( 0 );
7      VerificaDimezzamento( );
8      RETURN massimo;
9    }
```

[alvie]

Operazioni su uno heap implicito

```
1  RiorganizzaHeap( i ):          ⟨pre: heapArray è uno heap tranne che in posizione i⟩
2    WHILE (i>0 && heapArray[i].prio > heapArray[Padre(i)].prio) {
3      Scambia( i, Padre( i ) );
4      i = Padre( i );
5    }
6    WHILE (Sinistro(i) < heapSize && i != MigliorePadreFigli(i)) {
7      migliore = MigliorePadreFigli( i );
8      Scambia( i, figlio );
9      i = figlio;
10   }
```

```
1  MigliorePadreFigli( i ):          ⟨pre: il nodo in posizione i ha almeno un figlio⟩
2    j = k = Sinistro(i);
3    IF (k+1 < heapSize) k = k+1;
4    IF (heapArray[k].prio > heapArray[j].prio) j = k;
5    IF (heapArray[i].prio >= heapArray[j].prio) j = i;
6    RETURN j;
```

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

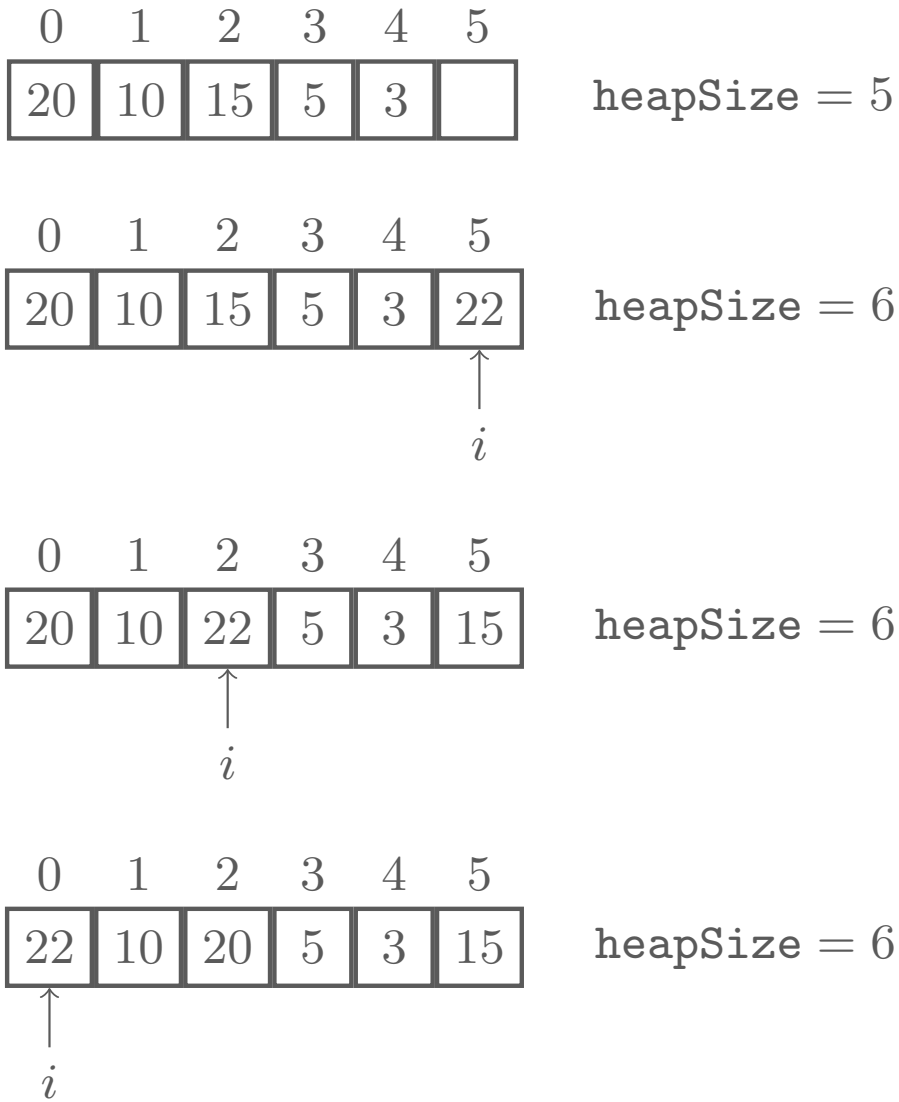
```
1  Sinistro( i ):
2    RETURN 2 × i + 1;
```

```
1  Scambia( i, j ):
2    tmp=heapArray[i];
3    heapArray[i]=heapArray[j];
4    heapArray[j]=tmp;
```

[alvie]

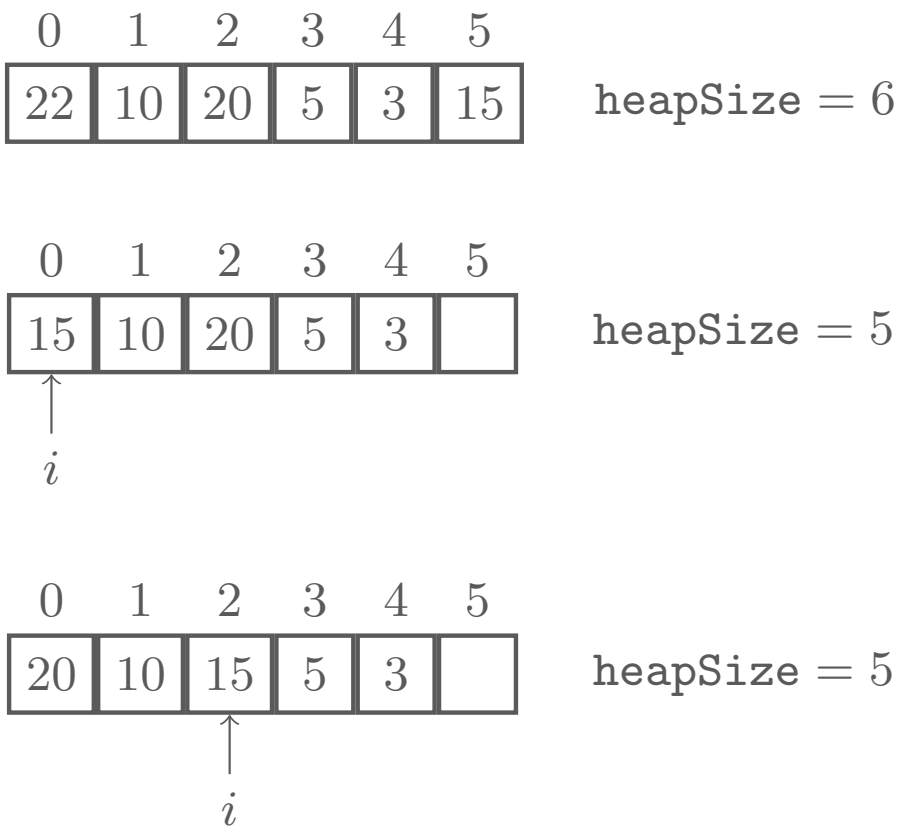
Enqueue su heap implicito

Enqueue(22) nello heap seguente.



Dequeue su heap implicito

Dequeue nello heap seguente.



Heapsort

Ordinamento mediante coda con priorità:

- Gli n elementi sono inseriti uno dopo l'altro nella coda con priorità PQ: n operazioni **Enqueue**.
- Gli n elementi sono estratti uno dopo l'altro nella coda con priorità PQ: n operazioni **Dequeue**.
- Semplificazione: estrazioni effettuate scambiando la radice (il massimo corrente) con l'ultima foglia e riducendo lo heap di un elemento

Lo heapsort opera in loco: non fa uso di memoria aggiuntiva a parte un numero costante di variabili ausiliarie

Per ordinare un array di n elementi richiede tempo $O(n \log n)$

Lower bound sull'ordinamento

Ogni algoritmo di ordinamento basato su confronti di elementi richiede $\Omega(n \log n)$ confronti al caso pessimo.

- Sia A un qualunque algoritmo di ordinamento che usa confronti tra coppie di elementi.
- In t confronti, A può discernere al più 3^t situazioni distinte.
- Il numero di possibili ordinamenti di n elementi è pari a $n!$.
- L'algoritmo deve discernere tra $n!$ possibili situazioni: quindi, deve valere $3^t \geq n!$
- In generale,

$$n! = n(n-1)\cdots 1 > n(n-1)\cdots\left(\frac{n}{2}+1\right) > \underbrace{\frac{n}{2}\cdots\frac{n}{2}}_{n/2 \text{ volte}} = (n/2)^{n/2}$$

- Quindi, deve essere $3^t \geq (n/2)^{n/2}$ e occorrono $t \geq (n/2)\log_3(n/2) = \Omega(n \log n)$ confronti.