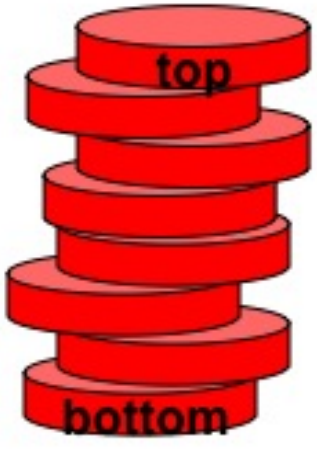
Estratto delle lezioni su pila, coda e coda a priorita`

Annalisa De Bonis

IASD 2014-15

Stack (pila)

- Collezione di elementi in cui inserimenti e cancellazioni avvengono secondo lo schema LIFO (last-in-first-out)
  - Un nuovo oggetto viene inserito in cima (top) allo stack
  - L'elemento cancellato è sempre quello che si trova in cima allo stack



Stack

- Operazioni supportate:
  - Push(x): inserisce l'elemento x allo stack
  - Pop(): cancella e restituisce in output l'ultimo elemento inserito
  - Top(): restituisce l'ultimo elemento senza cancellarlo
  - Empty(): restituisce **true** o **false** a seconda che lo stack sia vuoto o meno

Stack

- Esempi di applicazioni:
  - Storia delle pagine web visitate da un browser
  - Sequenza dei record di attivazione durante l'esecuzione di un programma
  - Struttura dati ausiliaria di un algoritmo

Lo stack dei metodi nella JVM

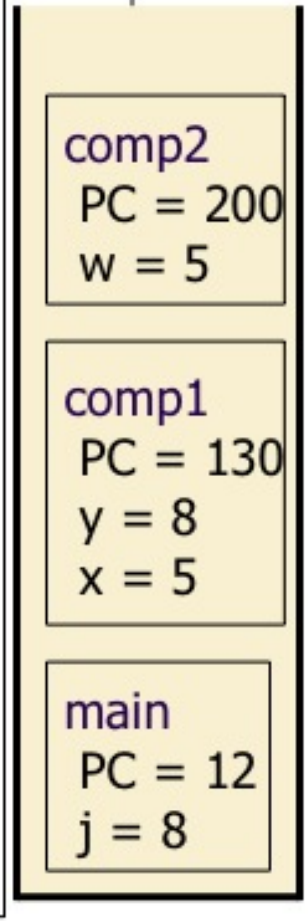
- La Java Virtual Machine (JVM) tiene traccia della catena di metodi attivi con uno stack
- Quando viene invocato un metodo JVM fa il push nello stack del frame contenente
  - **Parametri, variabili locali e valori di ritorno**
  - **Program counter**
- Quando l'esecuzione di un metodo termina, il suo frame viene estratto (pop) dallo stack e il controllo passa al metodo al top dello stack.
- Permette la ricorsione

```
main() {
   int j = 8;
   …
12  comp1(j);
  … }

comp1(int y) {
     int x;
    x = y-3;
130 comp2(x);
  … }

comp2(int w) {

200 … }
```



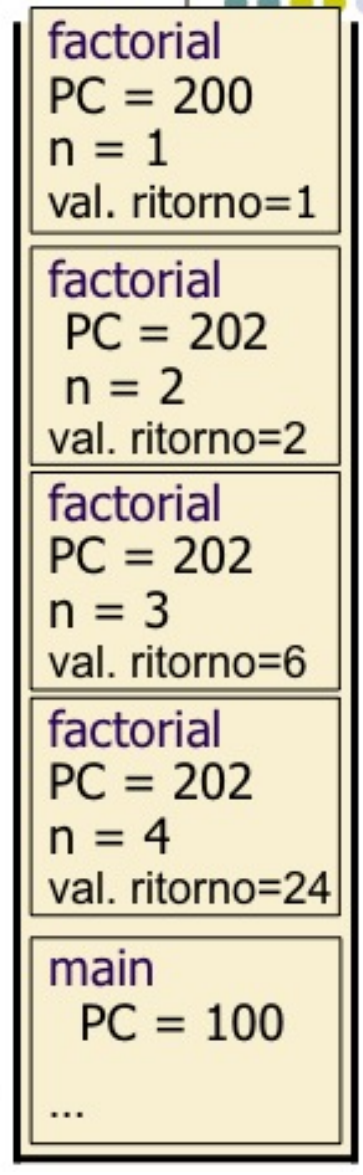
Lo stack dei metodi nella JVM

- Esempio dell'uso dello stack dei metodi nell'esecuzione di un metodo ricorsivo

```
main() {
…

100 factorial(4);
….
}


factorial(int n) {
      if(n<2)
200       return 1;
      else
202       return n*factorial(n-1);
}
```

Stack implementati con array

- Gli elementi vengono aggiunti da sinistra verso destra
- Una variabile tiene traccia dell'indice dell'elemento al top

Uno stack di n elementi richiede spazio O(n)

Stack basati su array dinamici

- Quando l'array è pieno, push rimpiazza l'array con uno più grande
- Quanto più grande deve essere l'array?
  - Strategia del raddoppio: raddoppia la dimensione
- Quando il numero di elementi divena pari a ¼ della dimensione dell'array, pop rimpiazza l'array con uno più piccolo
- Quanto più grande deve essere l'array?
  - Strategia del dimezzamento: dimezza la dimensione

Abbiamo gia` dimostrato che queste strategie comportano tempo ammortizzato O(n) per una sequenza di n operazioni

→ ciascuna operazione ha tempo ammortizzato costante

Implementazione di una pila con array

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

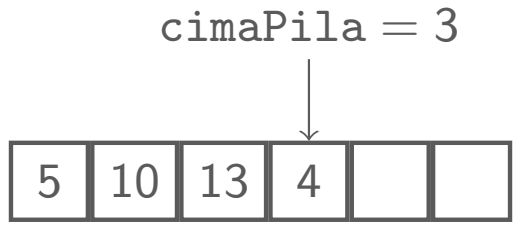
```
1  Pop( ):
2    IF (!Empty( )) {
3      x = pilaArray[ cimaPila ];
4      cimaPila = cimaPila - 1;
5      VerificaDimezzamento( );
6      RETURN x;
7    }
```

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

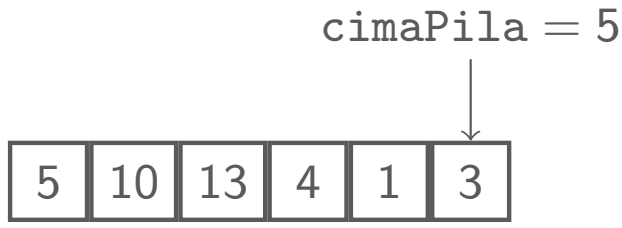
```
1  Empty( ):
2    RETURN (cimaPila == -1);
```

Implementazione di una pila con array

Dopo una **Pop**

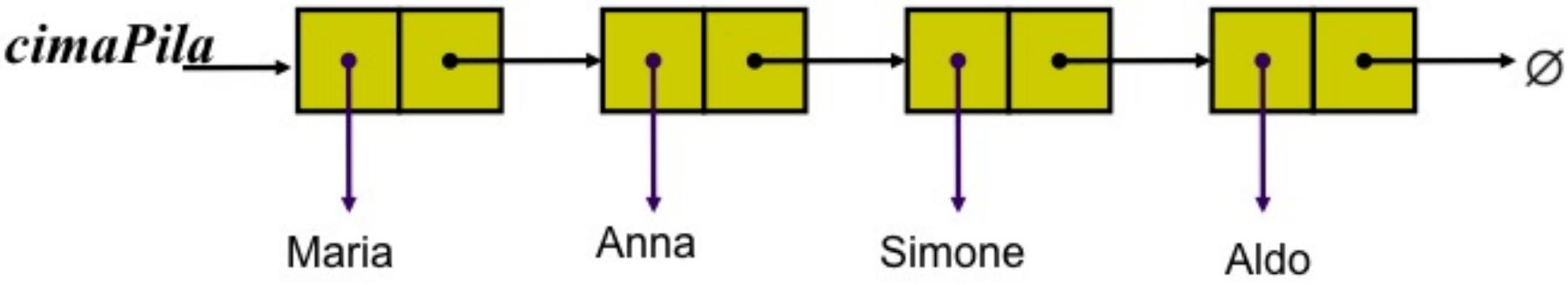


Dopo **Push**(1) e **Push**(3)



Implementazione di Stack con lista a puntatori singoli

- L'elemento in cima è immagazzinato nel primo nodo della lista
- Per uno stack di n elementi si usa uno spazio di dimensione $O(n)$
- Ciascuna operazione richiede tempo $O(1)$



Implementazione di una pila con una lista

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

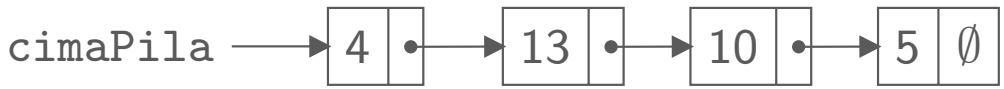
```
1  Pop( ):
2    IF (!Empty( )) {
3      x = cimaPila.dato;
4      cimaPila = cimaPila.succ;
5      RETURN x;
6    }
```

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

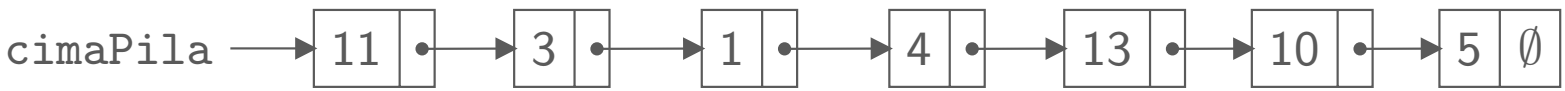
```
1  Empty( ):
2    RETURN (cimaPila == null);
```

Implementazione di una pila con una lista

Dopo una **Pop**



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



Uso dello stack come struttura dati ausiliaria di un algoritmo

- Ho una stringa che contiene un'espressione parentesizzata
- Esempio (a(b)c)()

- Voglio sapere se le parentesi sono ben accoppiate
- Esempio (a(b) Non bene accoppiate!
- Esempio (a()) Ben accoppiate

Algoritmo che controlla se in una stringa le parentesi tonde sono ben accoppiate

Input: stringa $s=s_0,..,s_{n-1}$ contente parentesi tonde

Output: true se s è ben parentesizzata; false altrimenti

```
Sia S uno stack
FOR (i=0 ;i< n;i++) {
  IF ( s_i ='(' )
        Push(s_i)
  ELSE IF(s_i =')' )
        IF (Empty() )
              RETURN false //non c'è la corrispondente
                           //parentesi aperta
        else Pop()
  IF  Empty() RETURN true
  ELSE RETURN false //una o più parentesi aperte per cui
                    //non c'è la corrispondente parentesi chiusa
```

Come modificare l'algoritmo in modo che funzioni in presenza di diversi tipi di parentesi

- Se si incontra una parentesi aperta (di qualsiasi tipo essa sia) si fa il push della parentesi incontrata
- Se si incontra una parentesi chiusa si fa il pop e si controlla che la parentesi estratta dallo stack sia dello stesso tipo di quella incontrata

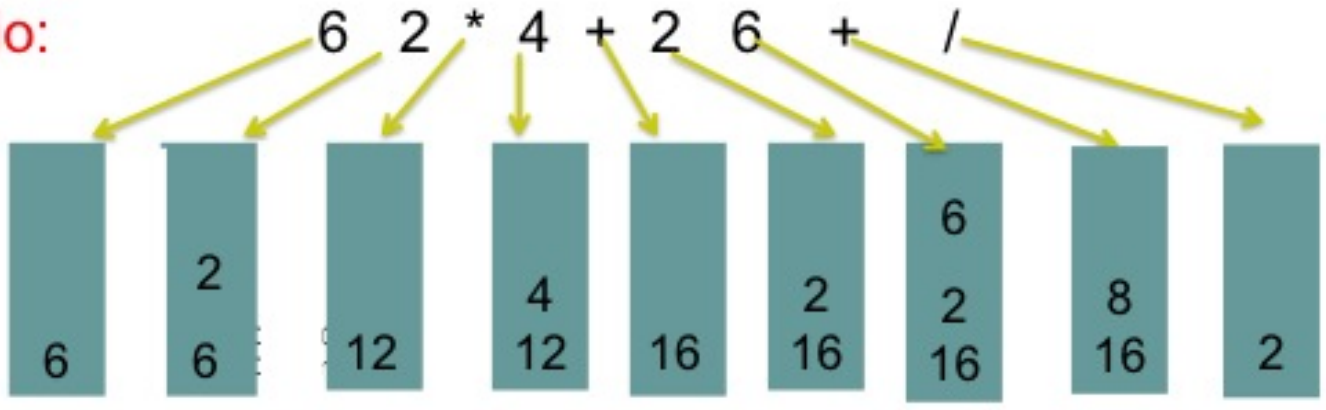
Valutazione di un'espressione in notazione postfissa

- Nella notazione postfissa l'operatore appare dopo gli operandi.
  - Esempi di espressioni aritmetiche in notazione postfissa
    - 1 2 + 5 * corrisponde a (1 + 2) * 5
    - 1 2 5 * + corrisponde a 1 + 2 * 5

Valutazione di un'espressione in notazione postfissa

- Usiamo uno stack per gli operandi
  - Ogni volta che incontriamo un operando facciamo il push dell'operando
  - Ogni volta che incontriamo un operatore
    1. preleviamo (con 2 operazioni di pop) 2 operandi dallo stack
    2. Applichiamo l'operatore ai due operandi e facciamo il push del risultato nello stack

Esempio:



Trasformazione di un'espressione in notazione infissa in postfissa

- Gli operandi compaiono nello stesso ordine in cui comparivano nella postfissa
  - Di conseguenza non appena si incontra un operando lo aggiungiamo alla stringa output
- Gli operatori invece cambiano posizione in quanto
  1. devono seguire gli operandi
  2. possono cambiare d'ordine a causa delle regole di priorità e delle parentesi

Trasformazione di un'espressione in notazione infissa in postfissa

- Idea: Usiamo uno stack per gestire i raggrupamenti nelle parentesi e le precedenze degli operatori.
  - Quando incontriamo '(' facciamo il push della parentesi nello stack.
  - Quando incontriamo ')' facciamo il pop di tutti gli operatori fino a che al top dello stack non compare '('. (Facciamo il pop anche di '(').
    - Man mano che estraiamo gli operatori dallo stack, li inseriamo nella stringa output.
  - Quando incontriamo uno degli operatori +, -, /, *, lo inseriamo nello stack **dopo** aver fatto il pop di tutti gli operatori con priorità maggiore o uguale.
  - Man mano che estraiamo gli operatori dallo stack, li inseriamo nella stringa output.

Trasformazione di un'espressione in notazione infissa in postfissa

Input: stringa che rappresenta un'espressione aritmetica in notazione infissa

Output: stringa che rappresenta l'espressione input in notazione postfissa

```
/* Lo pseudocodice assume che l'espressione sia ben formata*/
Crea una stringa vuota Postfix;
Crea uno stack opStack;
WHILE (c'è un altro carattere c da scandire){
IF( c è un operando)  aggiungi c alla fine della stringa Postfix;
ELSE IF (c== '(' )  opStack.push(c);
ELSE IF( c == ')' ) {
  WHILE ( opStack non è vuoto && opStack.top() != '(' )
    { aggiungi opStack.top() alla fine della stringa Postifix ; opStack.pop(); }
   opStack.pop(); //faccio il pop di '('
} //fine caso c == ')'
```

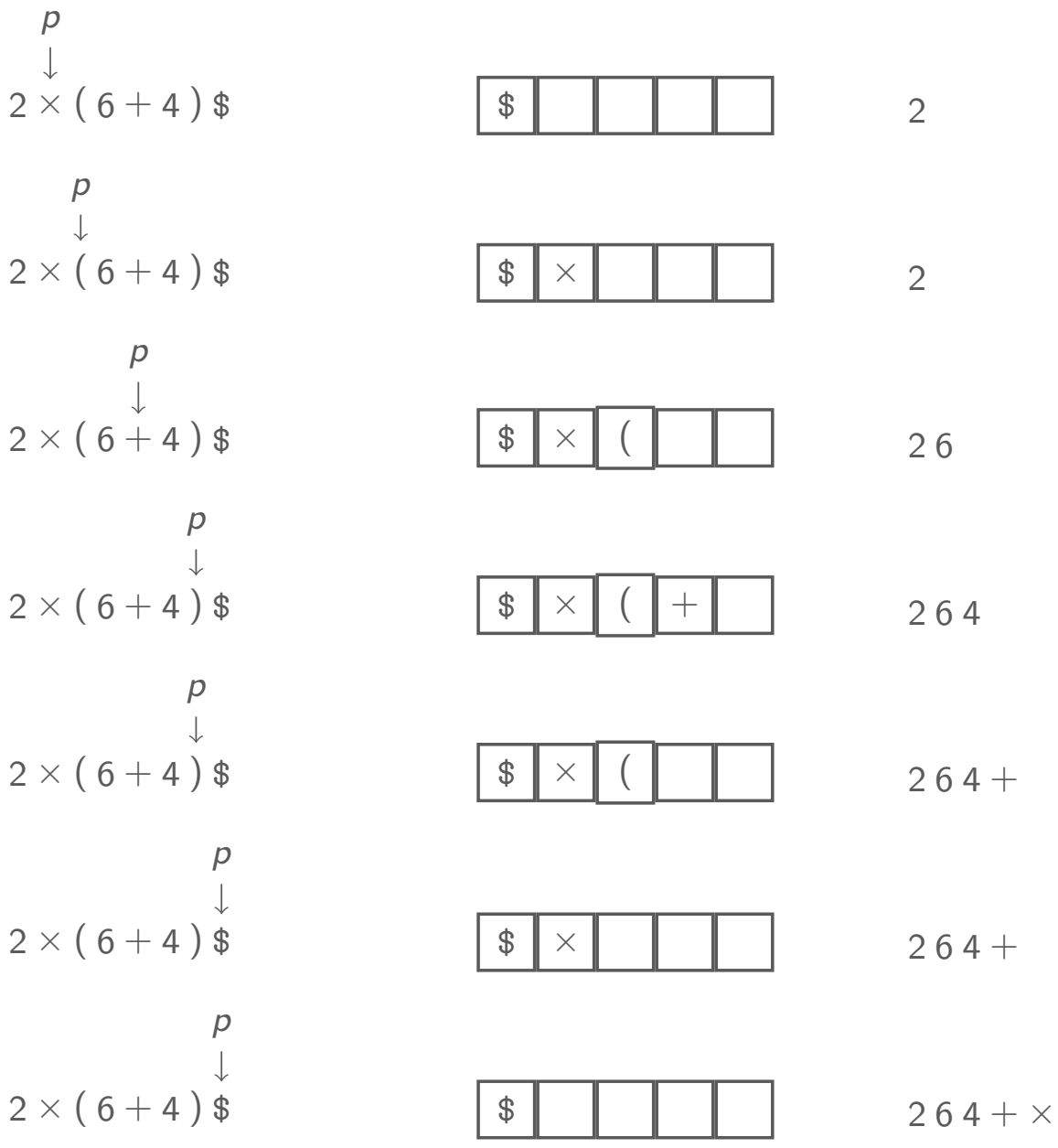

Continua →

Trasformazione di un'espressione in notazione infissa in postfissa

```
ELSE { // c è uno degli operatori +, -, /, *
   /*estraiamo dallo stack e aggiungiamo alla stringa
     tutti gli operatori con precedenza maggiore o uguale di c*/
  WHILE (opStack non è vuoto && opStack.top() != '(' &&
            prec(opStack.top()) >= prec(c) ) {
     aggiungi opStack.top() alla fine di Postfix; opStack.pop();
  }
   opStack.push(c );
  } // fine ELSE
} //fine del WHILE esterno che serve a scandire la stringa input
WHILE (opStack non è vuoto) //al termine svuoto lo stack
{
  aggiungi opStack.top() alla fine di Postfix ; opStack.pop();
}
```

Conversione di un'espressione infissa in postfissa

l'espressione è racchiusa tra $.



La Coda

- La coda (queue) e` un TDA che immagazzina oggetti arbitrari in cui inserimenti e cancellazioni avvengono secondo lo schema FIFO (first-in-first-out)
  - Un nuovo oggetto viene inserito in coda (rear)
  - L'elemento cancellato e` sempre quello che si trova davanti a tutti gli altri (front)

    • Viene cancellato l'elemento che è stato più a lungo nella coda

Operazioni sulla coda

Enqueue(x) aggiunge x alla fine (rear) della coda

Dequeue() restituisce l'elemento in testa (front) alla coda eliminandolo

First() restituisce il valore in testa alla coda

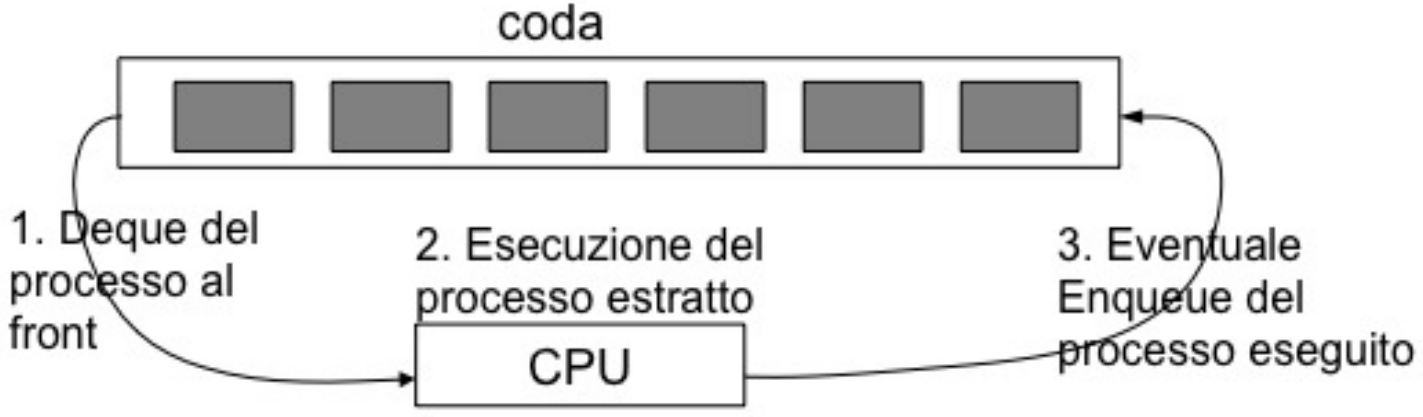
Empty() restituisce true se la coda non ha elementi

Applicazioni della coda

- Lista di attesa per la fruizione di un servizio
- Ordine di accesso ad una risorsa condivisa (per esempio le stampanti)
- Struttura dati ausiliaria di algoritmi

Esempio di applicazione: sistema multitasking con scheduler Round Robin

- L'algoritmo di scheduling **Round Robin** esegue i processi nell'ordine d'arrivo
  - I processi vengono messi in una coda in base all' ordine di arrivo
  - A ciascun processo viene allocato lo stesso tempo t
  - Il controllo viene trasferito al processo che si trova al front della coda
    - Il processo viene cancellato dalla coda
    - Se allo scadere del tempo t l'esecuzione del processo non è terminata allora il processo viene reinserito nella coda



Code implementate con array

due soluzioni inefficienti

- Soluzione 1: Il primo elemento e` nella locazione di indice 0 e gli elementi vengono aggiunti da sinistra a destra
  - Svantaggio: ogni volta che estraiamo un elemento dobbiamo shiftare a sinistra di una posizione i rimanenti elementi

Cancelliamo primo elemento e shiftiamo gli elementi a sinistra

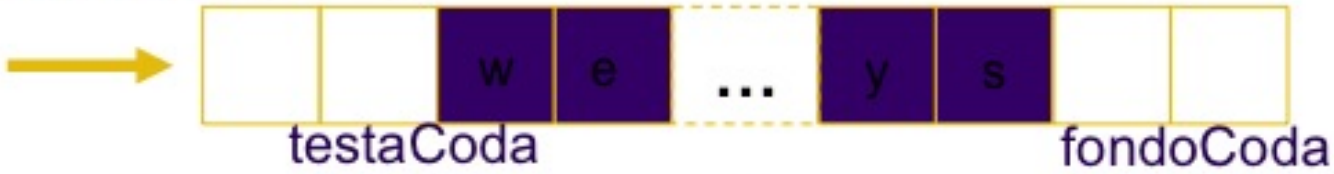
La coda dopo un'operazione di Dequeue

Code implementate con array

due soluzioni inefficienti

Soluzione 2: Due variabili tengono traccia del primo e dell'ultimo elemento

- testaCoda : indice dell' elemento in testa (incrementata da Deque)
- fondoCoda : indice dell'elemento successivo a quello in coda (incrementata da Enqueue)
- testaCoda≤ fondoCoda ; inizialmente testaCoda= fondoCoda =0



La coda dopo un'operazione di Dequeue

Code implementate con array

due soluzioni inefficienti

- Svantaggio della soluzione 2:
  - Potrei non riuscire ad inserire elementi nella coda pur avendo spazio a disposizione nell'array.
  - Si consideri, per esempio, una coda inizialmente vuota implementata mediante un array di dimensione N.
    - Se si inserisce e cancella N volte un elemento si ottiene testaCoda=fondoCoda=N per cui non sono possibili ulteriori inserimenti

- Svantaggio della soluzione 2:

testaCoda=0
fondoCoda=1

1) Enqueue(a)

testaCoda=1
fondoCoda=1

2) Dequeue()

testaCoda=1
fondoCoda=2

3) Enqueue(a)

testaCoda=2
fondoCoda=2

4) Dequeue()

testaCoda=2
fondoCoda=3

5) Enqueue(a)

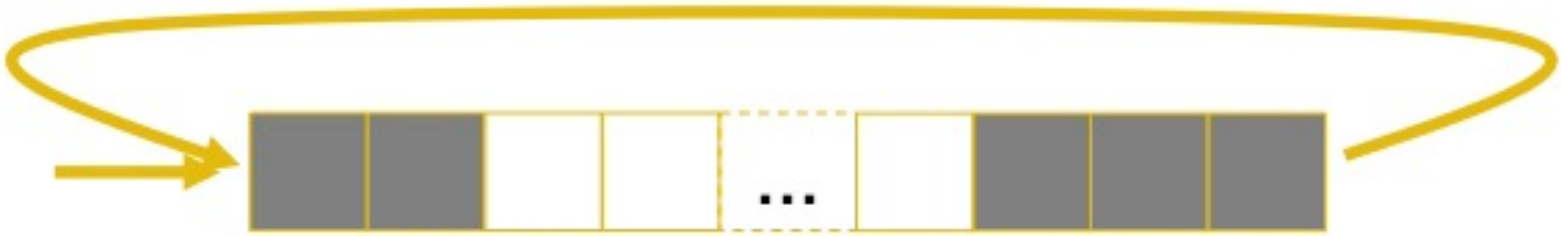
testaCoda=3
fondoCoda=3

Dequeue()

Code implementate con array

soluzione efficiente

- L'array viene usato "in maniera circolare"
- Due variabili tengono traccia del front e del rear
  - testaCoda : indice dell'elemento in testa
  - fondoCoda : indice successivo a quello in coda



Implementazione di una coda con array

- Quando nell'array non ci sono più locazioni disponibili si invoca VerificaRaddoppio che però è modificata in modo tale che gli elementi vengano copiati a partire dalla locazione di indice 0 nel nuovo array e *cimaCoda* viene inizializzato a 0 e *fondoCoda* a $cardCoda - 1$
- Quando nell'array ci sono al più 1/4 di locazioni occupate si invoca VerificaDimezzamento che però è modificata in modo tale che gli elementi vengano copiati a partire dalla locazione di indice 0 nel nuovo array e *cimaCoda* viene inizializzato a 0 e *fondoCoda* a $cardCoda - 1$

Implementazione di una coda con array

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

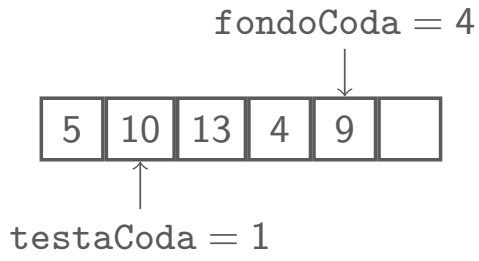
```
1  Dequeue( ):
2    IF (!Empty( )) {
3      cardCoda = cardCoda - 1;
4      x = codaArray[ testaCoda ];
5      testaCoda = (testaCoda + 1) % lunghezzaArray;
6      VerificaDimezzamento( );
7      RETURN x;
8    }
```

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

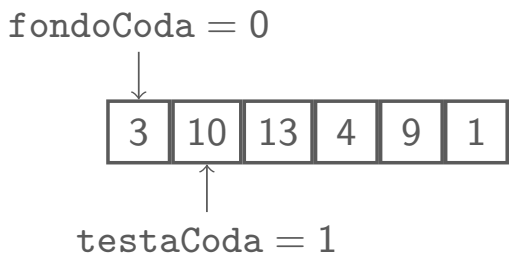
```
1  Empty( ):
2    RETURN (cardCoda == 0);
```

Implementazione di una coda con array

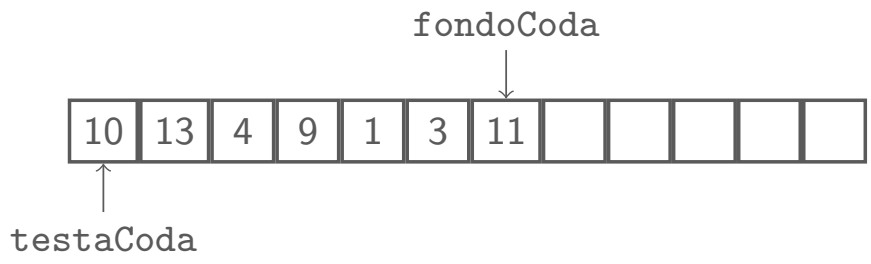
Dopo una **Dequeue**



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

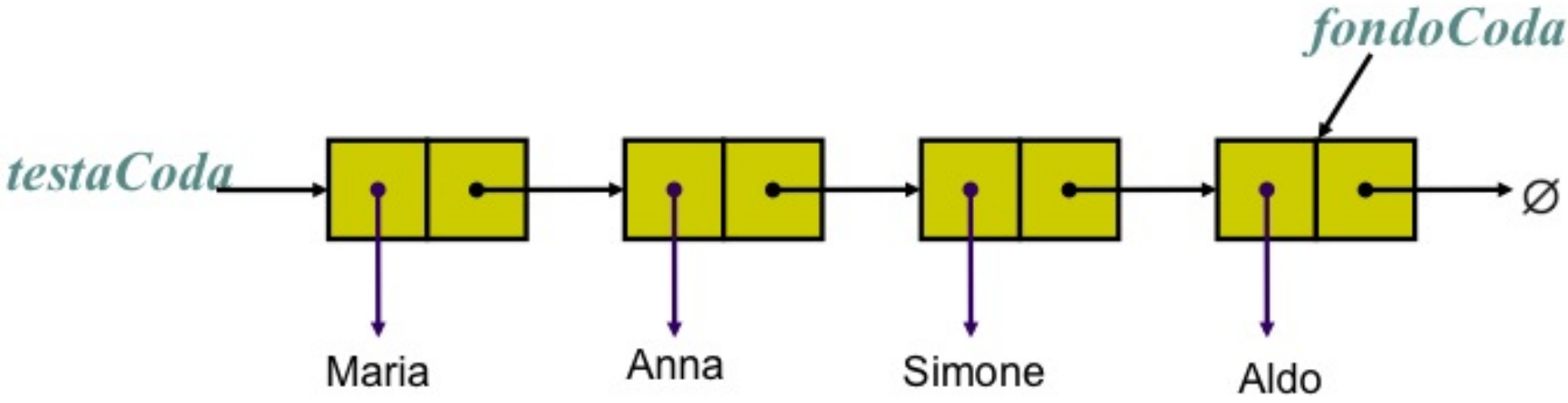


Dopo **Enqueue**(11)



Code implementate con liste a puntatori singoli

- L'elemento in testa è memorizzato nel primo nodo
- L'elemento in coda è memorizzato nell'ultimo nodo
- Per una coda di n elementi lo spazio usato e` $O(n)$
- Ciascuna operazione del TDA Coda richiede tempo $O(1)$



Implementazione di una coda con una lista

```
1  Enqueue( x ):
2    u = NuovoNodo( );
3    u.dato = x;
4    u.succ = null;
5    fondoCoda.succ = u;
6    fondoCoda = u;
```

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

```
1  First( ):
2    IF (!Empty( )) RETURN testaCoda.dato;
```

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

Coda a priorita`

- Una coda a priorità è una collezione di elementi a ciascuno dei quali è assegnata una priorita`
  - La priorita` viene assegnata nel momento in cui l'elemento è inserito nella coda
  - Le priorità degli elementi determinano l'ordine in cui vengono rimossi dalla coda
- Applicazioni
  - Viaggiatori in standby
  - Processi in attesa di usare una risorsa condivisa
  - Struttura ausiliaria di algoritmi

Coda a priorita`

- Ciascun elemento e ha due campi
  - e.prio : indica la priorita` dell'elemento
  - e.dato: dato a cui e` associata la priorita`

Priority scheduling

- Ad ogni processo è assegnata una priorità
- I processi in attesa di essere eseguiti sono inseriti in una coda priorità
- Viene estratto dalla coda ed eseguito il processo con priorità più grande
- Problema
  - Starvation: i processi con priorità più bassa non vengono mai eseguiti
- Soluzione
  - Aging: le priorità dei processi in coda vengono gradualmente aumentate

Relazione di ordine totale (≤)

- Proprietà
  - Riflessiva:

$$x \leq x$$

  - Antisimmetrica:

$$x \leq y \wedge y \leq x \Rightarrow x = y$$

  - Transitiva:

$$x \leq y \wedge y \leq z \Rightarrow x \leq z$$

- Su qualsiasi insieme finito sono sempre definiti sia il max che il min

Coda a priorita`

- L'operazione di Deque estrae e restituisce in output l'elemento con priorita` piu` alta
- L'operazione di First restituisce in output l'elemento con priorita` piu` alta

Implementazione con una lista non ordinata

- Memorizza gli elementi della coda in una lista in un ordine qualsiasi
- Complessità:
  - Enqueue richiede tempo $O(1)$ in quanto possiamo semplicemente inserire l'elemento alla fine o all'inizio della lista
  - Deque, First richiedono tempo $O(n)$ in quanto bisogna scorrere tutta la lista per determinare l'elemento con priorita` massima

Implementazione con una lista ordinata

- Memorizza gli elementi della coda in una lista ordinata in modo non crescente in base alle priorita` degli elementi
- Complessità:
  - Enqueue richiede tempo $O(n)$ in quanto occorre trovare il posto dove inserire l'oggetto in modo da mantenere l'ordine non crescente delle priorita`
  - Deque, First richiedono tempo $O(1)$ in quanto la priorita` massima è all'inizio della lista

Implementazione mediante Heap

- Bilanciamento del costo di Deque ed Enqueue
  - Ottenuto ordinando solo in parte gli elementi
- Organizzazione degli elementi ad albero

Implementazione mediante Heap

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

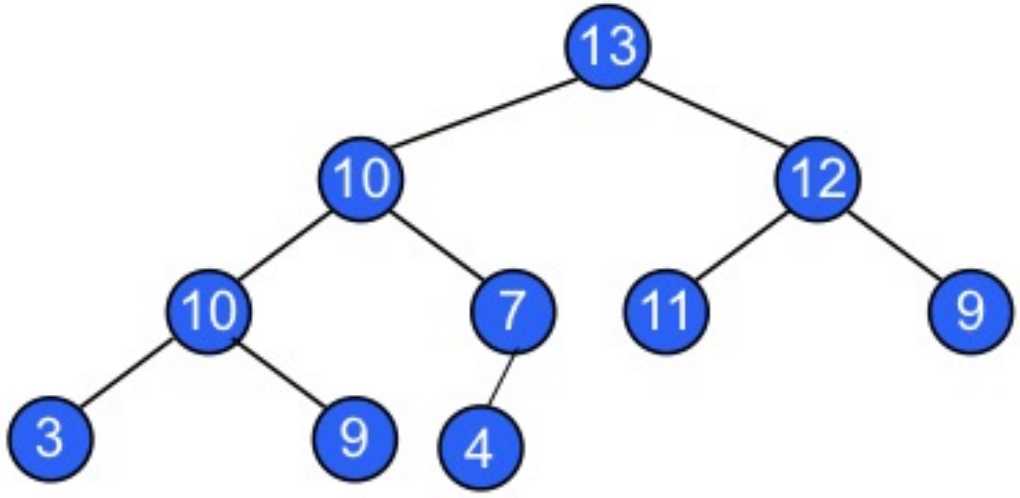
Implementazione mediante Heap

- La definizione di heaptree implica che la radice di un heaptree contiene l'elemento di priorità massima.

Definizione

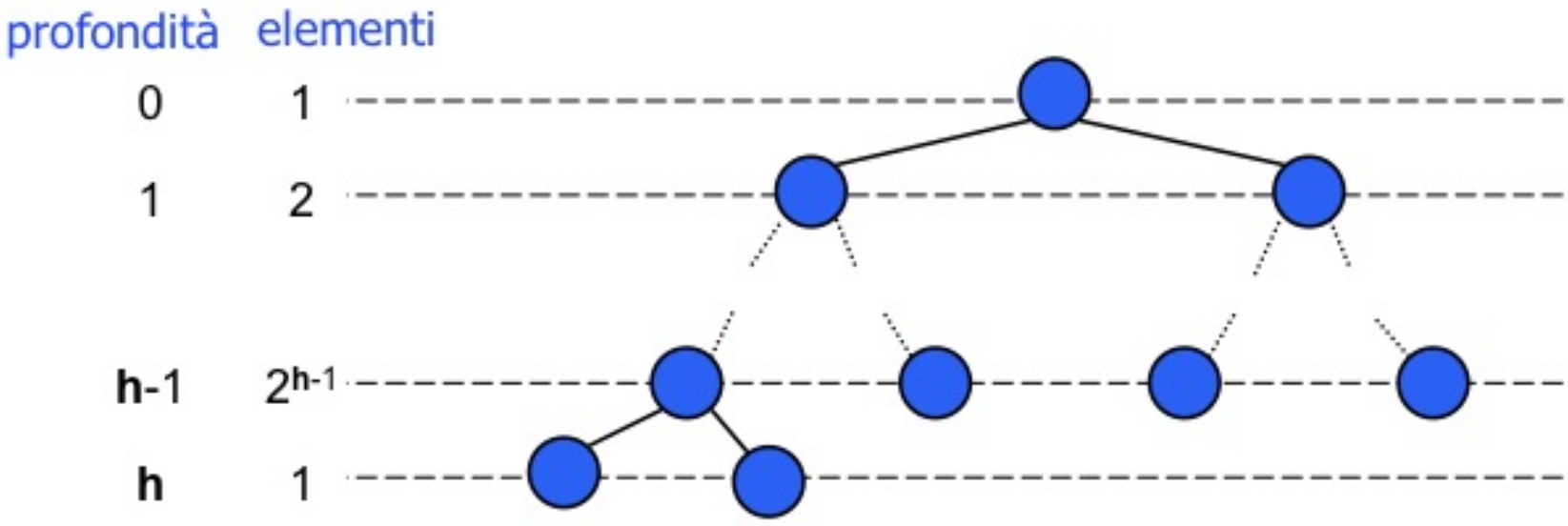
- Un heap è un albero binario soddisfa le seguenti proprietà:
  - Heaptree: per ogni nodo v ≠ radice
    - Sia x l'elemento di v e y l'elemento del padre di v. Si ha che y.prio ≥ x.prio
  - Albero binario completo a sinistra: dato un heap di altezza **h**
    - per **i** = 0, … , **h**-1, ci sono 2^i nodi di profondità i (tutti i livelli, salvo al più l'ultimo, sono pieni)
    - L'ultimo livello è riempito da sinistra verso destra

Esempio



Altezza di un heap

- Un heap che memorizza n elementi ha altezza $\lfloor \log n \rfloor$
  Dimostrazione: Sia h l'altezza dell'albero
  - Ci sono 2^i elementi a profondità $i = 0, \ldots, h - 1$ ed almeno un elemento a profondità **h** $\rightarrow n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 = 2^h$
  - quindi $h \leq \log n$



Altezza di un heap

- Il numero massimo di nodi di un albero binario di altezza h è
  - $n \leq 1 + 2 + 4 + \ldots + 2^h = 2^{h+1}-1$
  - NB: Il massimo si ottiene quando l'albero è completamente bilanciato e cioè quando tutte le foglie hanno la stessa profondità

Altezza di un heap

- Le disuguaglianze nelle due slide precedenti implicano

$2^h \leq n \leq 2^{h+1}-1$ ➔ $\log(n+1)-1 \leq h \leq \log n$

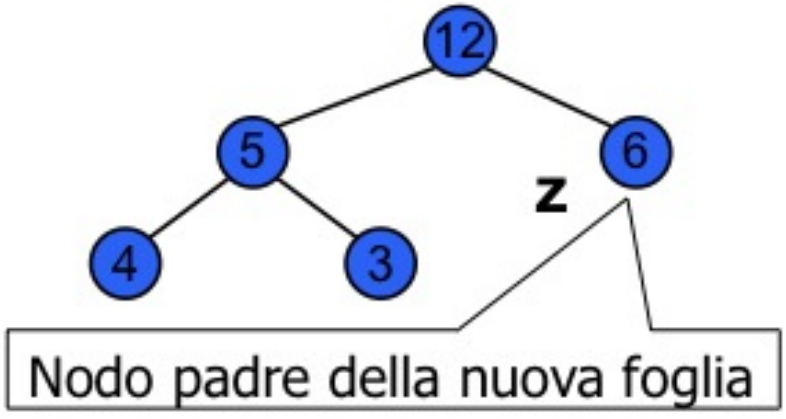
➔ $\log(n)-1 < h \leq \log n$

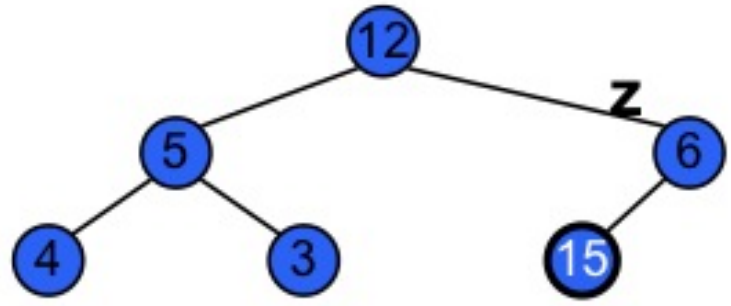
$h = \lfloor \log n \rfloor$

Enqueue di e in heap H

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.





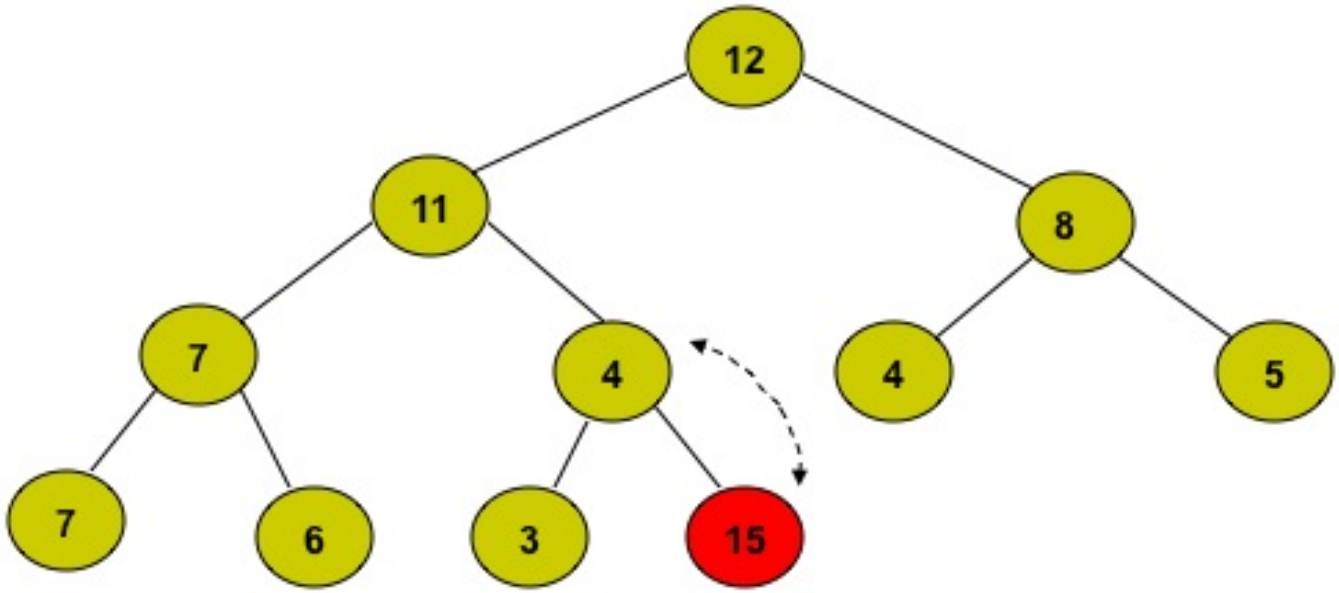
3. Iterativamente, la priorità di e viene confrontata con quella dell'elemento f contenuto nel padre di v: se e.prio > f.prio, i due nodi vengono scambiati.

4. L'iterazione termina quando v diventa la radice oppure quando e.prio ≤ f.prio.

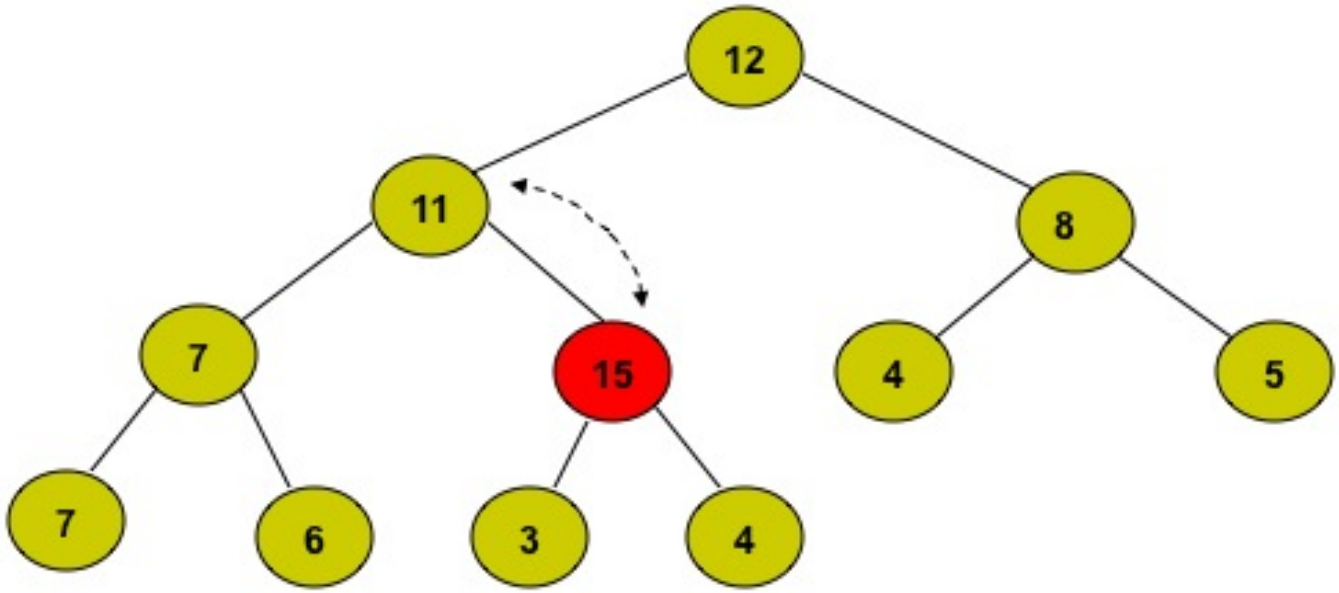
Enqueue di e in heap H

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

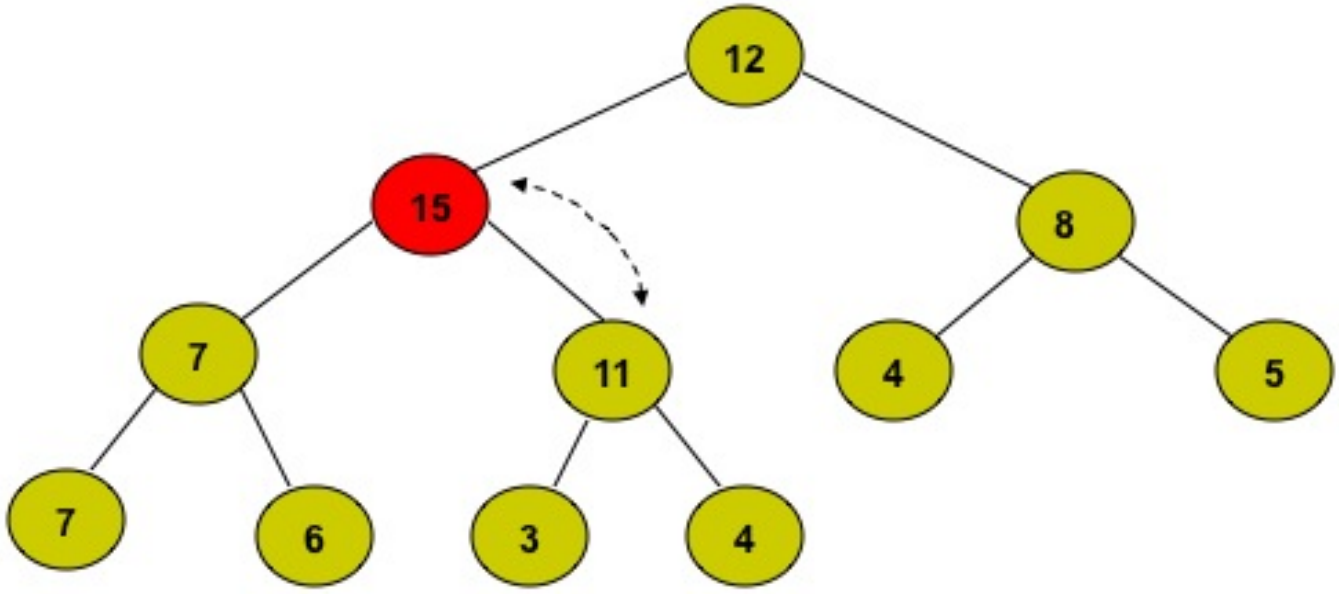
Inserimento della chiave 15



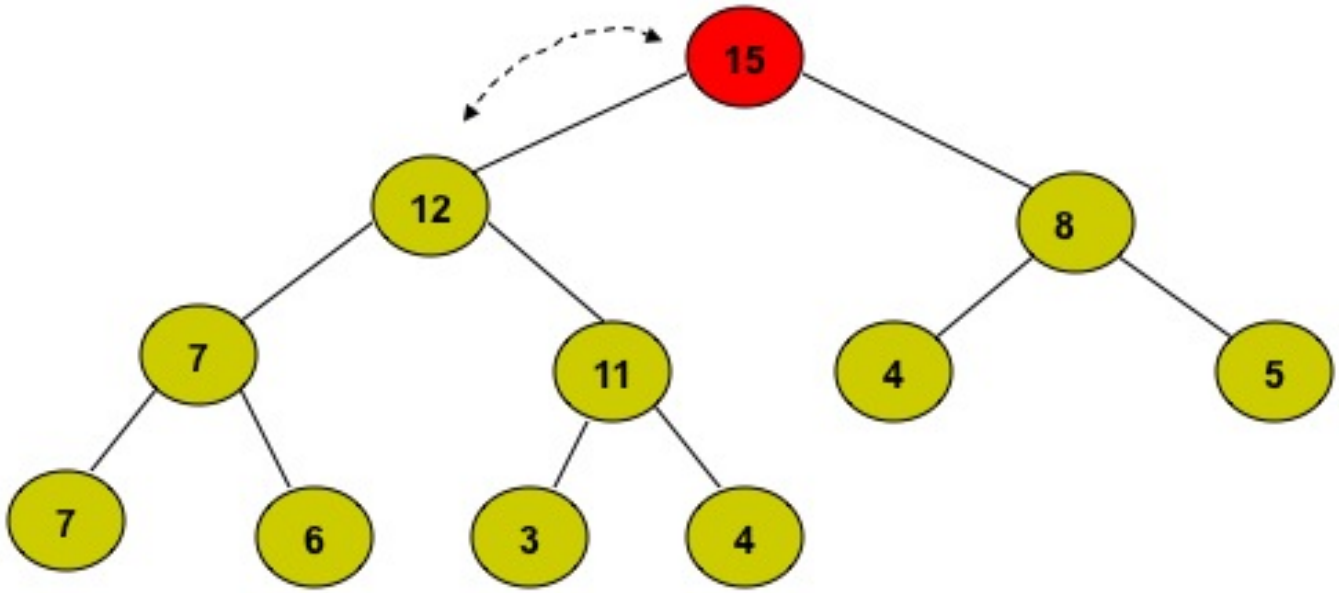
Inserimento della chiave 15



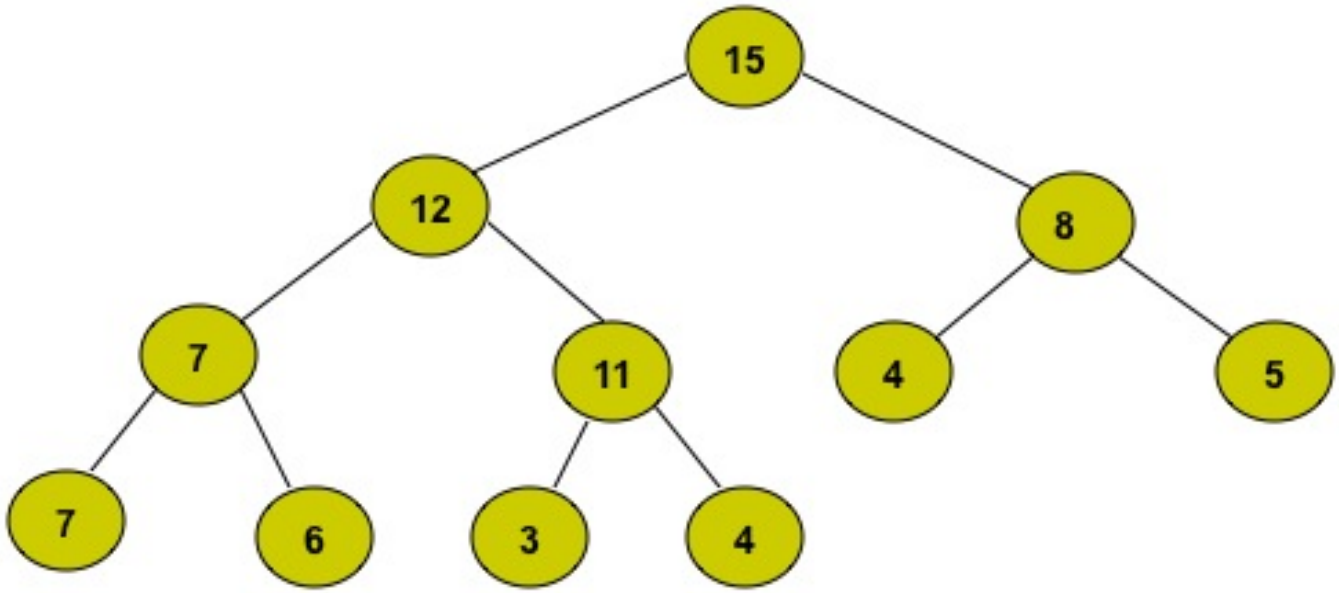
Inserimento della chiave 15



Inserimento della chiave 15

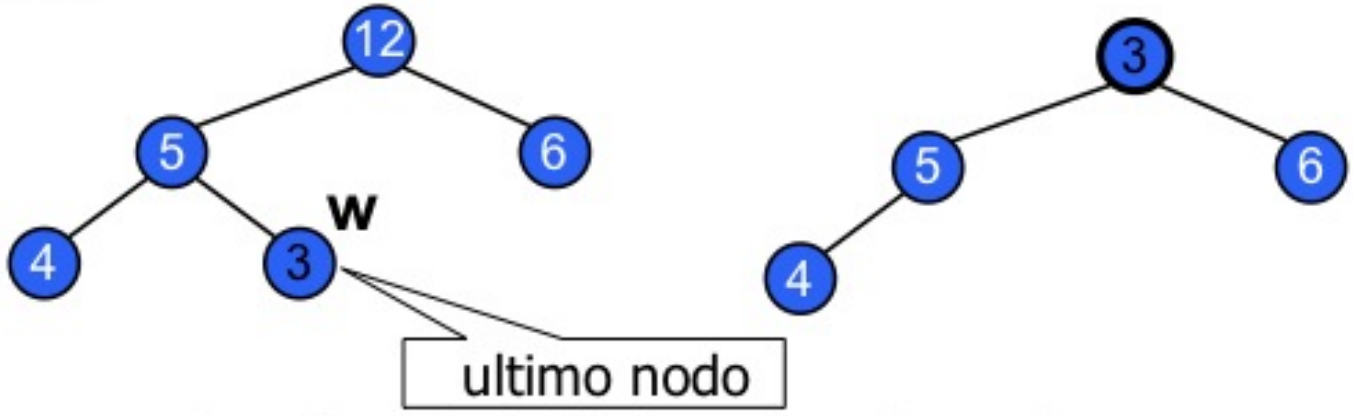


Inserimento della chiave 15



Dequeue da H

1. La radice di H viene rimossa e l'elemento in essa contenuto viene poi restituito.
2. La foglia più a destra sull'ultimo livello di H, viene inserita come radice v.

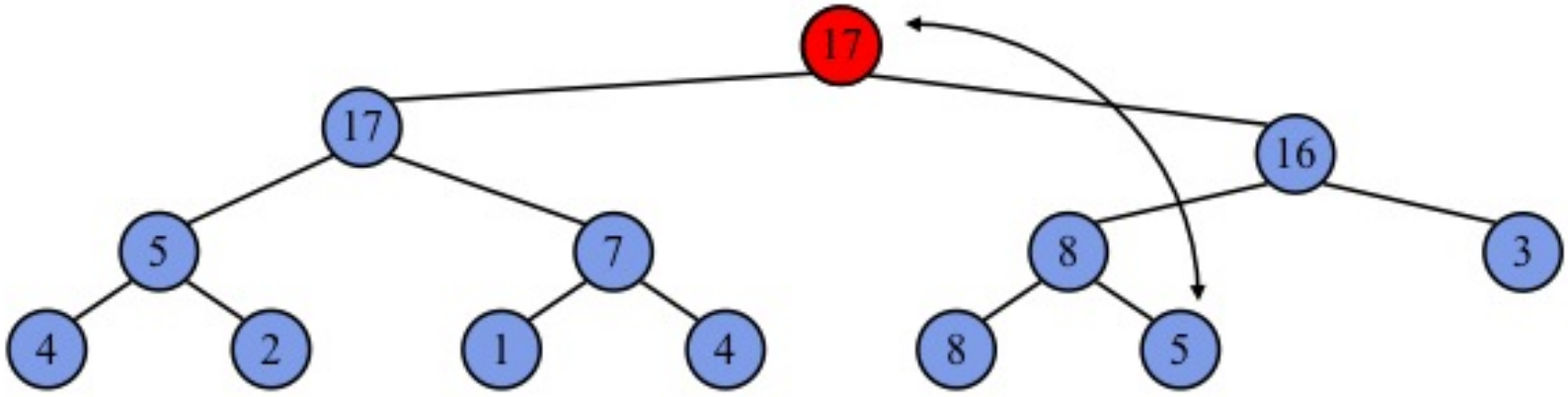


3. Iterativamente, la priorità dell'elemento in v viene confrontata con quelle degli elementi nei suoi figli: se fra i tre nodi, v non ha priorità max, il nodo v viene scambiato con il figlio contenente l'elemento di priorità max.
4. L'iterazione termina se v diventa una foglia o se contiene un elemento di priorità maggiore di quelle degli elementi contenuti nei suoi figli.

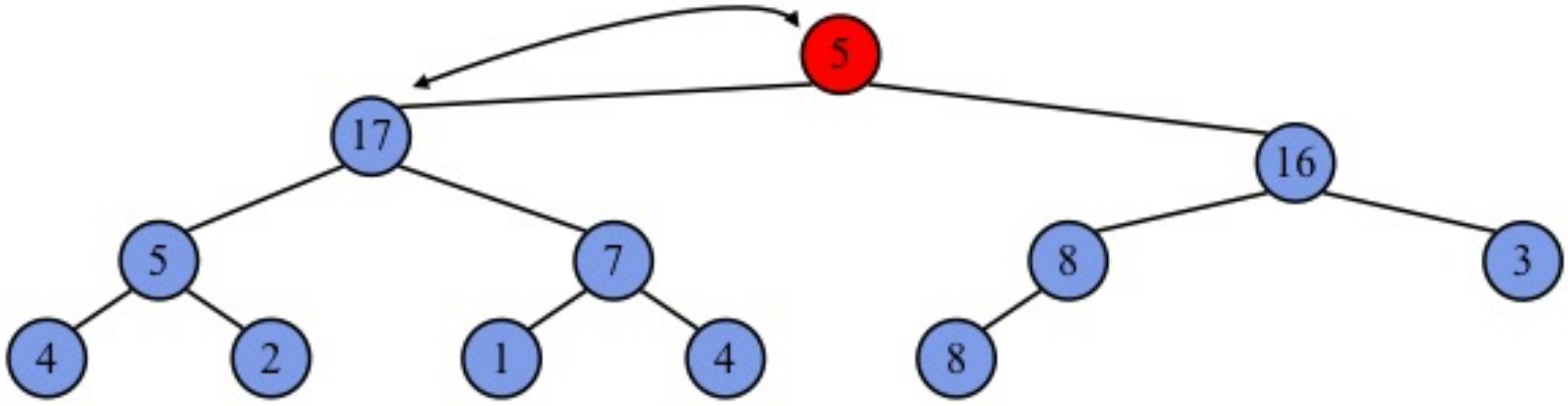
Dequeue da H

- 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

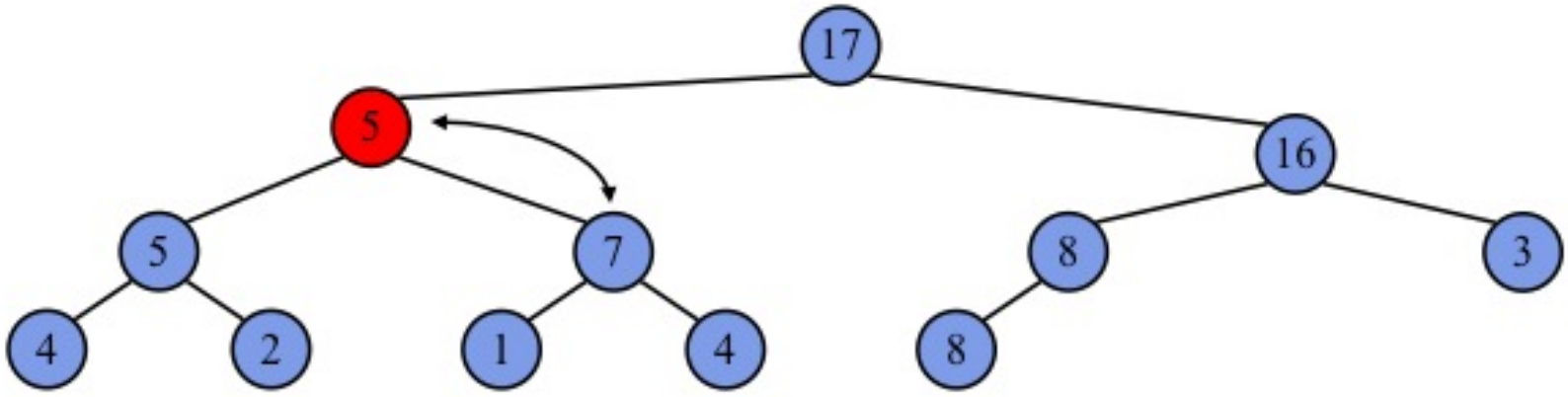
Cancellazione del massimo



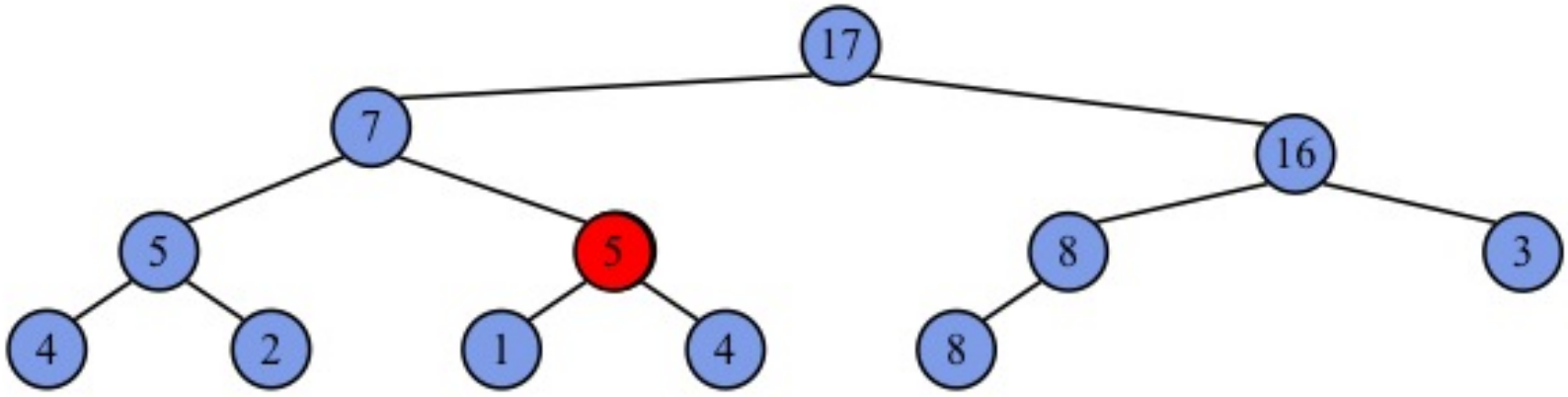
Cancellazione del massimo



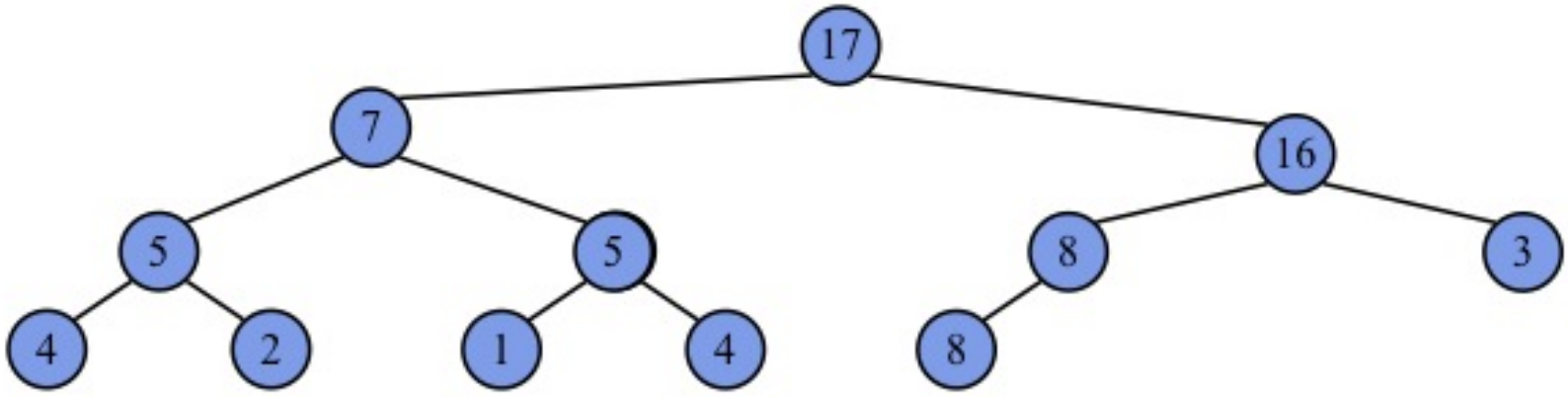
Cancellazione del massimo



Cancellazione del massimo



Cancellazione del massimo

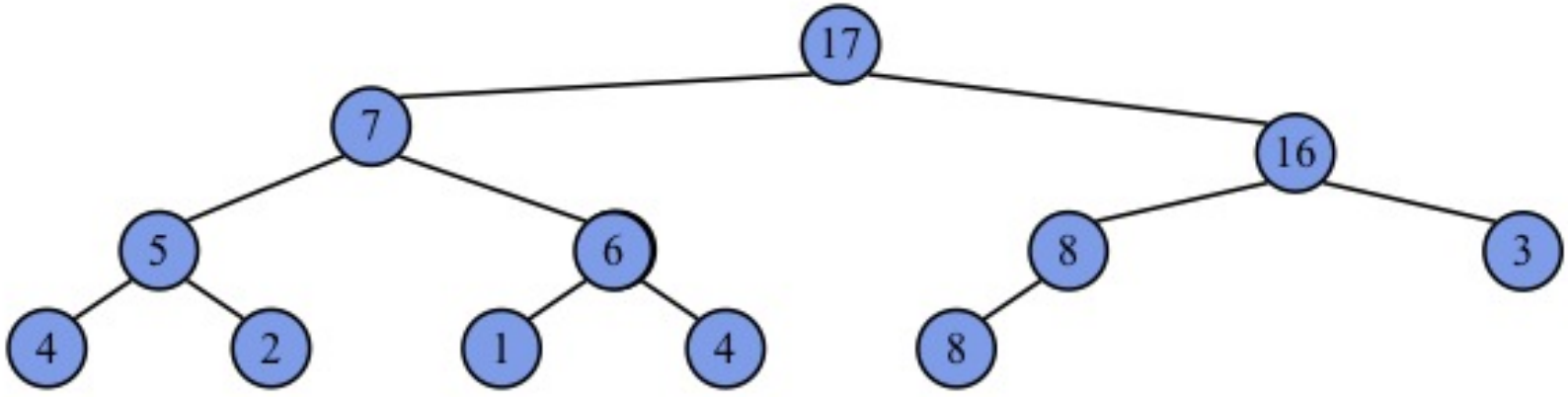


Implementazione di un heap mediante array (heap implicito)

- La relazione tra i nodi di un albero completo a sinistra di n nodi può essere rappresentata in modo implicito utilizzando un array di lunghezza n.
- Regola di posizionamento
  - La radice occupa la locazione di indice i = 0.
  - Se un nodo occupa la locazione di indice i, il suo figlio sinistro (se esiste) occupa la posizione 2i + 1 e il suo figlio destro (se esiste) occupa la posizione 2i + 2.

Implementazione di un heap mediante array (heap implicito)

- Esempio di heap implicito



17	7	16	5	6	8	3	4	2	1	4	8

Implementazione di un heap mediante array (heap implicito)

- i=0 ➔u.padre=null
- 2i+1≥n➔u.sx=null
- 2i+2≥n➔u.dx=null

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

- Se un nodo occupa la locazione di indice i, suo padre occupa la posizione $\lfloor (i-1)/2 \rfloor$

Empty e First in un heap implicito

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

```
1  First( ):
2  if (!Empty( )) return heapArray[0];
```

Enqueue in un heap implicito

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

Dequeue in un heap implicito

```
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 }
```

Riorganizzazione di un heap

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. {
4. Scambia(i, Padre(i));
5. i = Padre(i);
6. }
7. while (Sinistro(i) < heapSize && i != MigliorePadreFigli(i)) {
8. migliore = MigliorePadreFigli(i);
9. Scambia(i, migliore);
10. i = migliore;
11. }

Riorganizzazione di un heap

```
1. MigliorePadreFigli( i ):
2. j = k = Sinistro(i);
3. if (k+1 < heapSize) k = k+1;    <figlio destro>
4. if (heapArray[k].prio > heapArray[j].prio) <se destro>sinistro
5.  j = k;
6. if (heapArray[i].prio >= heapArray[j].prio) <se padre >=figli>
7.  j = i;
8.  return j;
```

Riorganizzazione di un heap

1. Padre(i):
2. return (i-1)/2;

Riorganizzazione di un heap

1. Sinistro(i):
2. return 2 × i + 1;

Costo operazioni su heap

- Empty e First
  - richiedono tempo O(1)
- Enqueue e Dequeue
  - Se escludiamo tempo impiegato da VerificaDimezzamento e VerificaRaddoppio, richiedono O(log n)
  - Tempo VerificaDimezzamento e VerificaRaddoppio richiedono tempo ammortizzato O(1)

➔sequenza di k operazioni di First, Empty, Dequeue e Enqueue su una coda che inizialmente contiene m elementi e` O(klog n), dove n=m+k

Heapsort

Ordinamento mediante coda con priorità:

- Vogliamo ordinare gli elementi di una sequenza (ad esempio, una lista o un array).
- Gli n elementi sono cancellati dalla sequenza input e inseriti uno dopo l'altro nella coda con priorità : n operazioni **Enqueue**.
- Gli n elementi sono estratti uno dopo l'altro nella coda con priorità : n operazioni **Dequeue** e inseriti man mano all'inizio della sequenza.

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

Heapsort

Ordinamento mediante coda con priorità:

Possiamo implementare heapsort in modo che operi in loco: non fa uso di memoria aggiuntiva a parte un numero costante di variabili ausiliarie

```
HeapSort(heapArray ):
  heapSize = 0;
  FOR (i = 0; i < n; i = i+1) {
    Enqueue(heapArray[i]); //Enqueue incrementa heapSize
  }
  WHILE (heapSize>0) {
    Scambia(heapSize-1,0);
    heapSize = heapSize-1;
    RiorganizzaHeap(0);
  }
```

Heapsort

Ordinamento mediante coda con priorità:

- Gli n elementi sono inizialmente nell'array heapArray che però non rappresenta ancora un heap (NB: heapSize=0)
- nel for gli elementi vengono riposizionati in modo tale che l'array heapArray formi un heap. Dopo l'i-esima iterazione le prime $i+1$ locazioni formano un heap.
- nel while gli elementi vengono ordinati:
  - ad ogni iterazione del while la porzione dell'array già ordinata è quella formata dalle locazioni con indice compreso tra gli indici *heapSize* ed $n-1$ mentre lo heap è formato dalle locazioni con indice compreso tra 0 e *heapSize* -1.
  - Ad ogni iterazione, l'elemento nella radice dell'heap (*heapArray*[0]) viene scambiato con quello in *heapArray*[*heapSize* -1], *heapSize* viene decrementato e lo heap viene "riorganizzato" a partire dalla radice.

Esempio HeapSort

- Esecuzione del for

4	9	2	5	7	3	
4	9	2	5	7	3	i=0
9	4	2	5	7	3	i=1
9	4	2	5	7	3	i=2
9	5	2	4	7	3	i=3
9	7	2	4	5	3	i=4
9	7	3	4	5	2	i=5

Continua nella prossima slide

- Esecuzione del while

9	7	3	4	5	2	heapSize=6
2	7	3	4	5	9	Scambio 2 e 9
7	5	3	4	2	9	heapSize=5
2	5	3	4	7	9	Scambio 2 e 7
5	4	3	2	7	9	heapSize=4
2	4	3	5	7	9	Scambio 2 e 5
4	2	3	5	7	9	heapSize=3
3	2	4	5	7	9	Scambio 3 e 4
3	2	4	5	7	9	heapSize=2
2	3	4	5	7	9	Scambio 2 e 3

Lower bound sull'ordinamento

Ogni algoritmo di ordinamento basato su confronti di elementi richiede $\Omega(n \log n)$ confronti nel caso pessimo per ordinare un insieme di n elementi.

- Consideriamo un qualsiasi algoritmo di ordinamento che usa confronti tra coppie di elementi.
- Dal momento che siamo interessati a stabilire un lower bound nel caso pessimo, possiamo fare qualsiasi assunzione sull'input (il caso pessimo richiederà almento lo stesso numero di confronti)
  - Assumiamo quindi senza perdere di generalità che gli elementi da ordinare siano tutti distinti
  - In questo caso i confronti ci danno sempre solo due tipi di risposte $<$ o $>$
- se faccio t confronti, ho 2^t sequenze associate alle possibili sequenze di risposte ai confronti effettuati.
  - Esempio: per $t = 2$ le possibili sequenze di risposte sono "$<, <$" " $<, >$", "$>, <$" e "$>, >$".

Lower bound sull'ordinamento

- Ad ogni possibile sequenza di risposte corrisponde un ordinamento output diverso
  - NB: alcune sequenze di risposte potrebbero non essere possibili perchè non compatibili con alcun ordinamento.
  - Ad esempio, nel caso di $n = 3$ e $t = 3$, tre confronti ci permettono di confrontare tutte le coppie.
  - Supponiamo che l'insieme input sia $\{a, b, c\}$ e che la sequenza di confronti sia "$a : b$, $a : c$, $b : c$".
  - La sequenza di risposte "<,>,<" non può verificarsi perchè il primo e il terzo confronto inplicano $a < b < c$ mentre il secondo implica $a > c$
- Con t confronti, possiamo quindi distinguere tra al più 2^t diversi ordinamenti.
- Il numero di possibili ordinamenti di n elementi è pari a $n!$.
- L'algoritmo deve discernere tra $n!$ possibili situazioni: quindi, deve valere $2^t \geq n!$
- $$n! = n(n-1)\cdots 1 > n(n-1)\cdots\left(\frac{n}{2}+1\right) > \underbrace{\frac{n}{2}\cdots\frac{n}{2}}_{n/2 \text{ volte}} = (n/2)^{n/2}$$
- Quindi, deve essere $2^t \geq (n/2)^{n/2}$ e occorrono $t \geq (n/2)\log_2(n/2) = \Omega(n \log n)$ confronti.

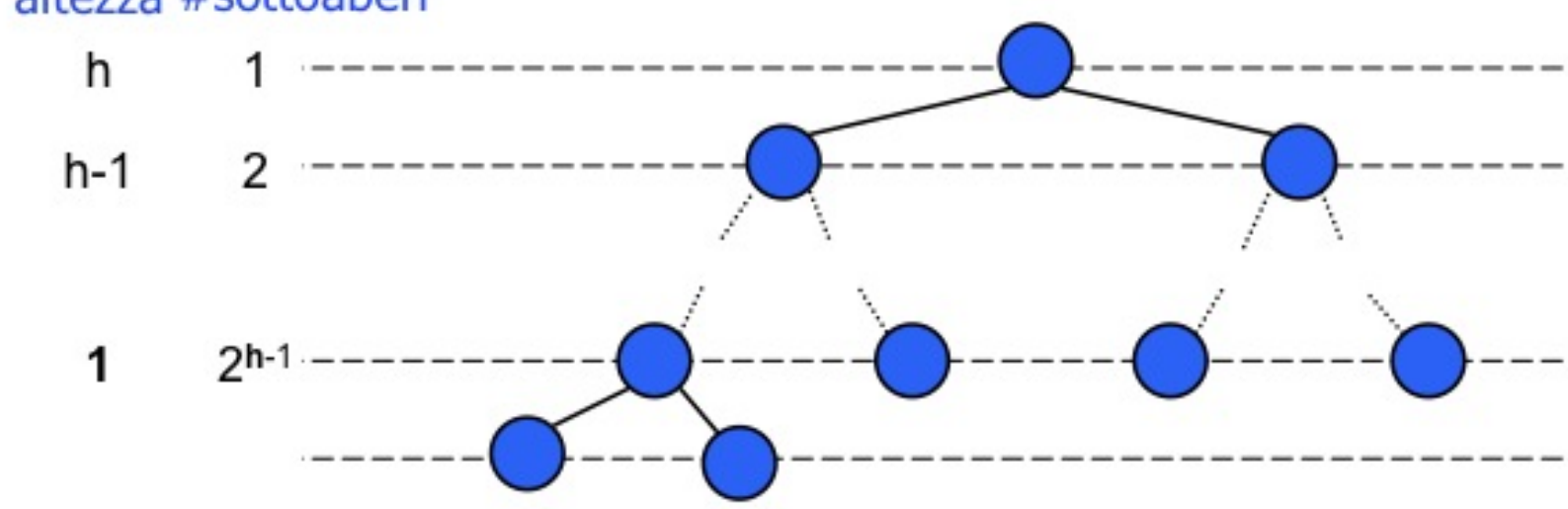
Costruzione di un heap con n entrate in tempo lineare (Esercizio 2.12)

- Se conosciamo in anticipo gli elementi che costituiscono la coda a priorità allora possiamo costruire l'heap in tempo lineare
  - NB: Se invochiamo n volte Enqueue, la costruzione dell' heap richiede tempo O(nlog n)

Costruzione bottom-up di un heap

- Inseriamo tutti gli elementi nell'array
- Ripristiniamo la proprietà di heapTree dal basso verso l'alto

```
CreaHeapMigliorato():
heapSize=n;
FOR(i=n/2-1;i>0;i=i-1){
  RiorganizzaHeapFigli(i); //esegue il while di RiorganizzaHeap
}
```

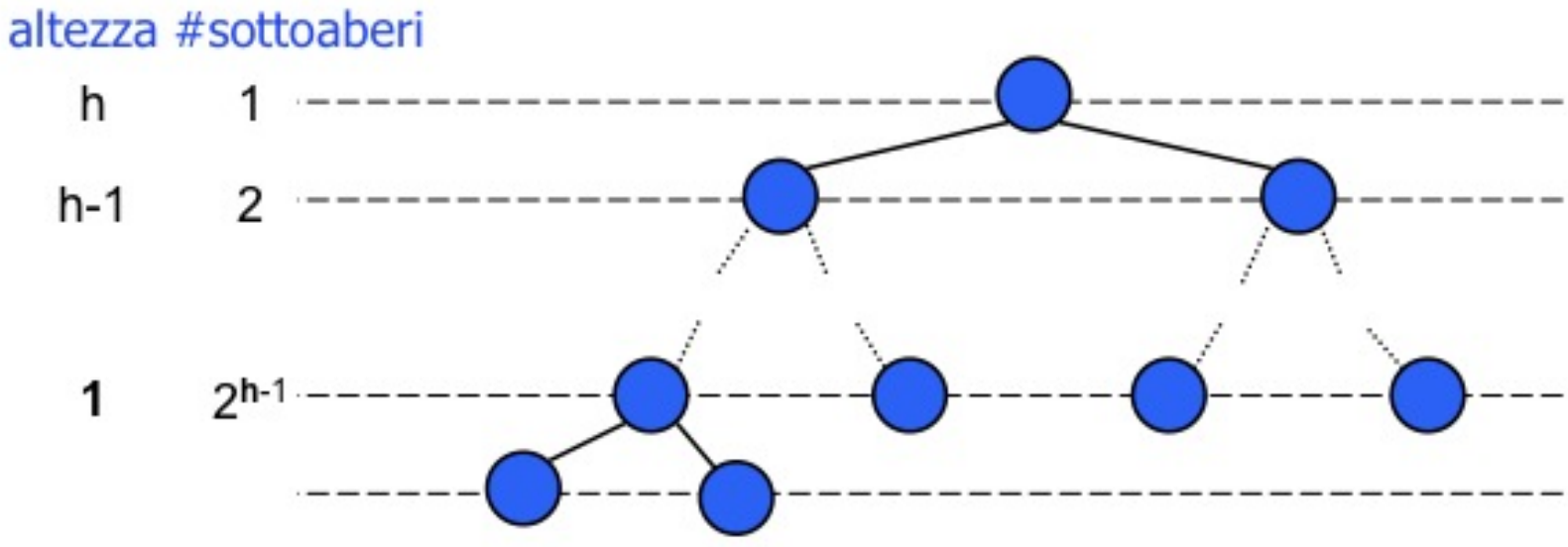


La funzione RiorganizzaHeapFigli

```
RiorganizzaHeapFigli( i ):
while (Sinistro(i) < heapSize && i != MigliorePadreFigli(i)) {
migliore = MigliorePadreFigli( i );
Scambia( i, migliore );
i = migliore;
 }
```

Costruzione bottom-up di un heap

- Il for procede per livelli: prima considera tutti i nodi interni di altezza 1, poi quelli di altezza 2 e così via fino ad arrivare alla radice che ha altezza h
- Per ogni $a = 1,\ldots,h$, invochiamo RiorganizzaHeapFigli su tutti i nodi di altezza a
- Dopo la fase a, tutti i sottoalberi di altezza $\leq a$ soddisfano la proprietà dell'heaptree.



Costruzione bottom-up di un heap

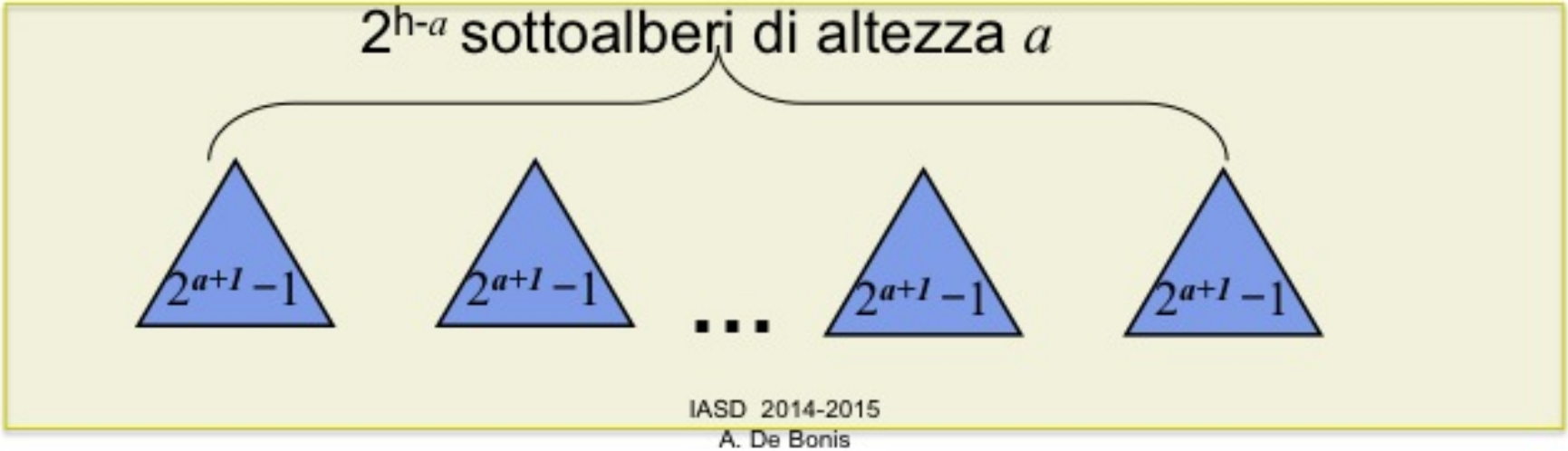
- Possiamo costruire un heap contenente n elementi in h = O(log n) fasi.
- Analisi: Per semplicità consideriamo il caso in cui anche l'ultimo livello è pieno ($n=2^{h+1}-1$)

  I nodi di altezza a hanno profondità h-a

  ➔ci sono 2^{h-a} nodi di altezza a

  ➔ogni fase richiede tempo $O(a2^{h-a})$

  ➔in totale il tempo di esecuzione è $O(\sum_{i=1}^{h} a\,2^{h-a}) = O(2^h) = O(n)$



Per dimostrare che il tempo di esecuzione di CreaHeapMigliorato è $O(2^h)$ dobbiamo stimare $\sum_{a=1}^{h} a2^{h-a}$.

Dimostrazione.

$$\sum_{a=1}^{h} ax^{h-a} = x^{h+1} \sum_{a=1}^{h} ax^{-(a+1)} \tag{1}$$

$$\begin{aligned}
\sum_{a=1}^{h} ax^{-(a+1)} &= \sum_{a=1}^{h} -\frac{d}{dx}\left(\frac{1}{x^a}\right) = -\frac{d}{dx}\left(\sum_{a=1}^{h}\left(\frac{1}{x^a}\right)\right) = -\frac{d}{dx}\left(\frac{1-1/x^{h+1}}{1-1/x} - 1\right) \\
&= -\frac{d}{dx}\left(\left(1-1/x^{h+1}\right)\cdot\frac{x}{x-1} - 1\right) \\
&= -\left[\frac{(h+1)}{x^{h+2}}\cdot\frac{x}{x-1} - \left(1-1/x^{h+1}\right)\cdot\frac{1}{(x-1)^2}\right] \\
&= \left[\frac{x^{h+1}-1}{x^{h+1}}\cdot\frac{1}{(x-1)^2} - \frac{(h+1)}{x^{h+2}}\cdot\frac{x}{x-1}\right].
\end{aligned}$$

$$
\begin{aligned}
\rightarrow \sum_{a=1}^{h} a x^{h-a} &= x^{h+1} \sum_{a=1}^{h} a x^{-(a+1)} \\
&= x^{h+1} \left[\frac{x^{h+1}-1}{x^{h+1}} \frac{1}{(x-1)^2} - \frac{(h+1)}{x^{h+2}} \frac{x}{x-1} \right] \\
&= \left[\frac{x^{h+1}-1}{(x-1)^2} - \frac{(h+1)}{x-1} \right]
\end{aligned}
$$

$$
\rightarrow \sum_{a=1}^{h} a 2^{h-a} = (2^{h+1} - 1 - h - 1) = O(2^h)
$$