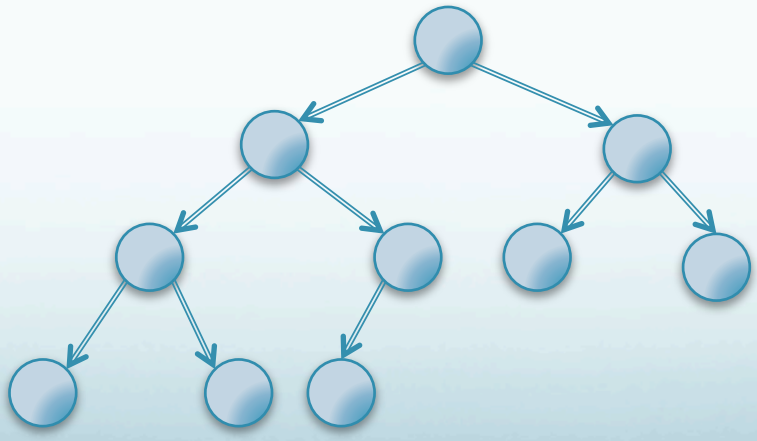
Heap

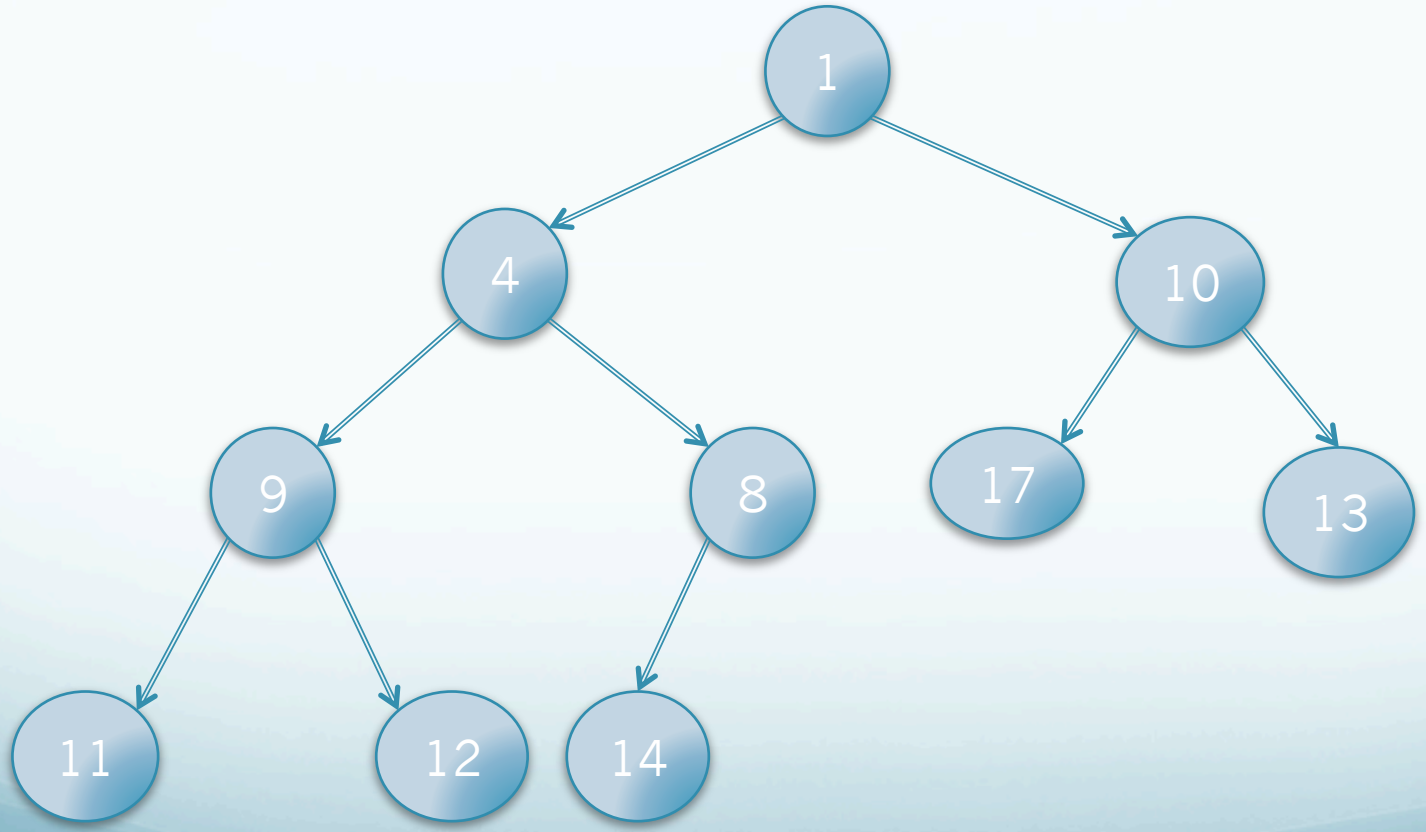
- Un heap binario è un albero binario con le seguenti caratteristiche:
  - È quasi completo: tutti i livelli, tranna al più l'ultimo sono completi e le foglie dell'ultimo livello sono tutte adossate a sinistra.



Heap

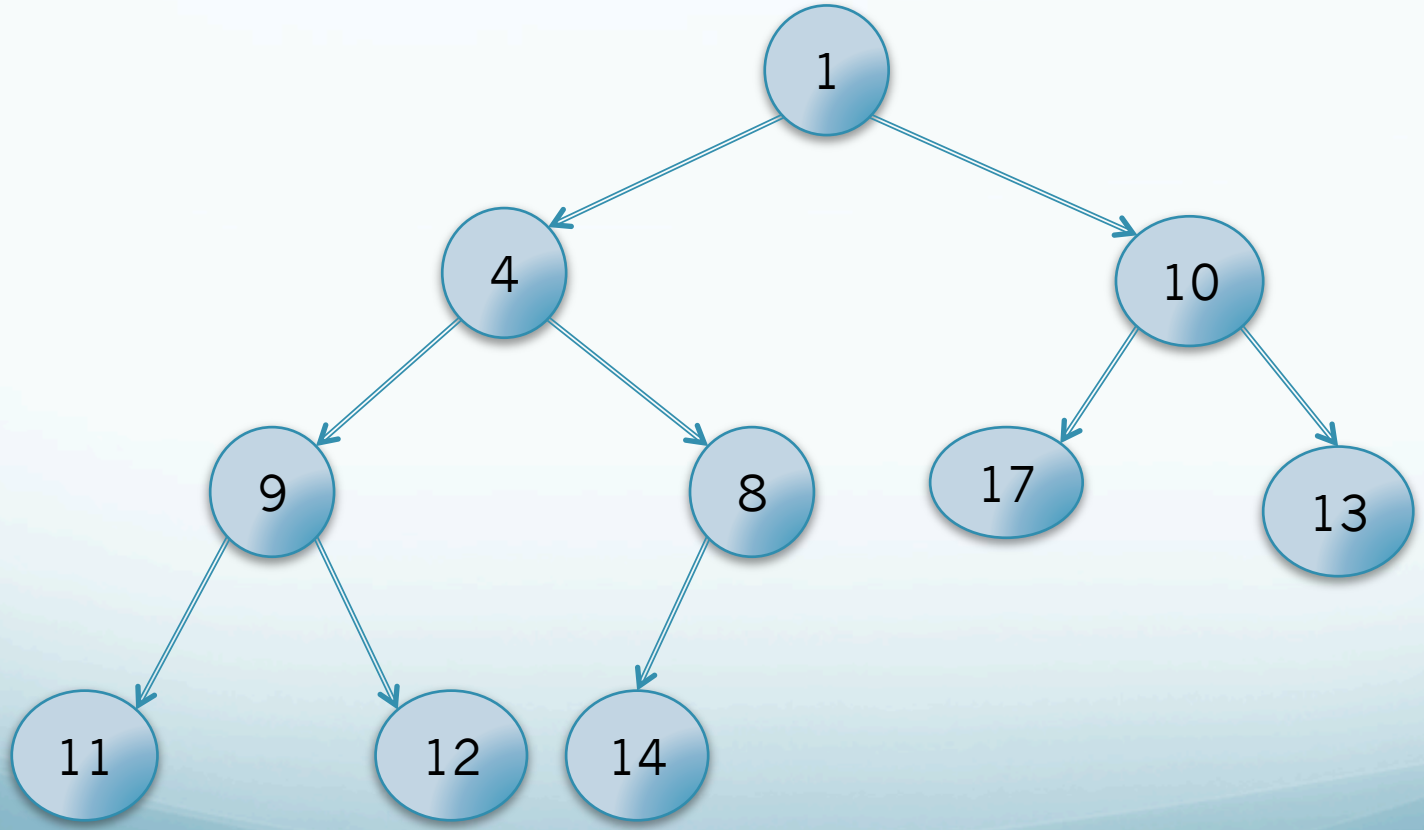
- Un heap è un albero binario con le seguenti caratteristiche:
  - È quasi completo: tutti i livelli, tranne al più l'ultimo sono completi e le foglie dell'ultimo livello sono tutte addossate a sinistra.
  - Ad ogni nodo è associato un valore
  - Tutti i valori presenti nei nodi nell'albero costituiscono un insieme totalmente ordinato
  - Per ogni nodo, il valore contenuto nel padre del nodo è minore o uguale di quello contenuto nel nodo stesso (min-heap)
  - Per ogni nodo, il valore contenuto nel padre del nodo è maggiore o uguale di quello contenuto nel nodo stesso (max-heap)

Min-heap: esempio



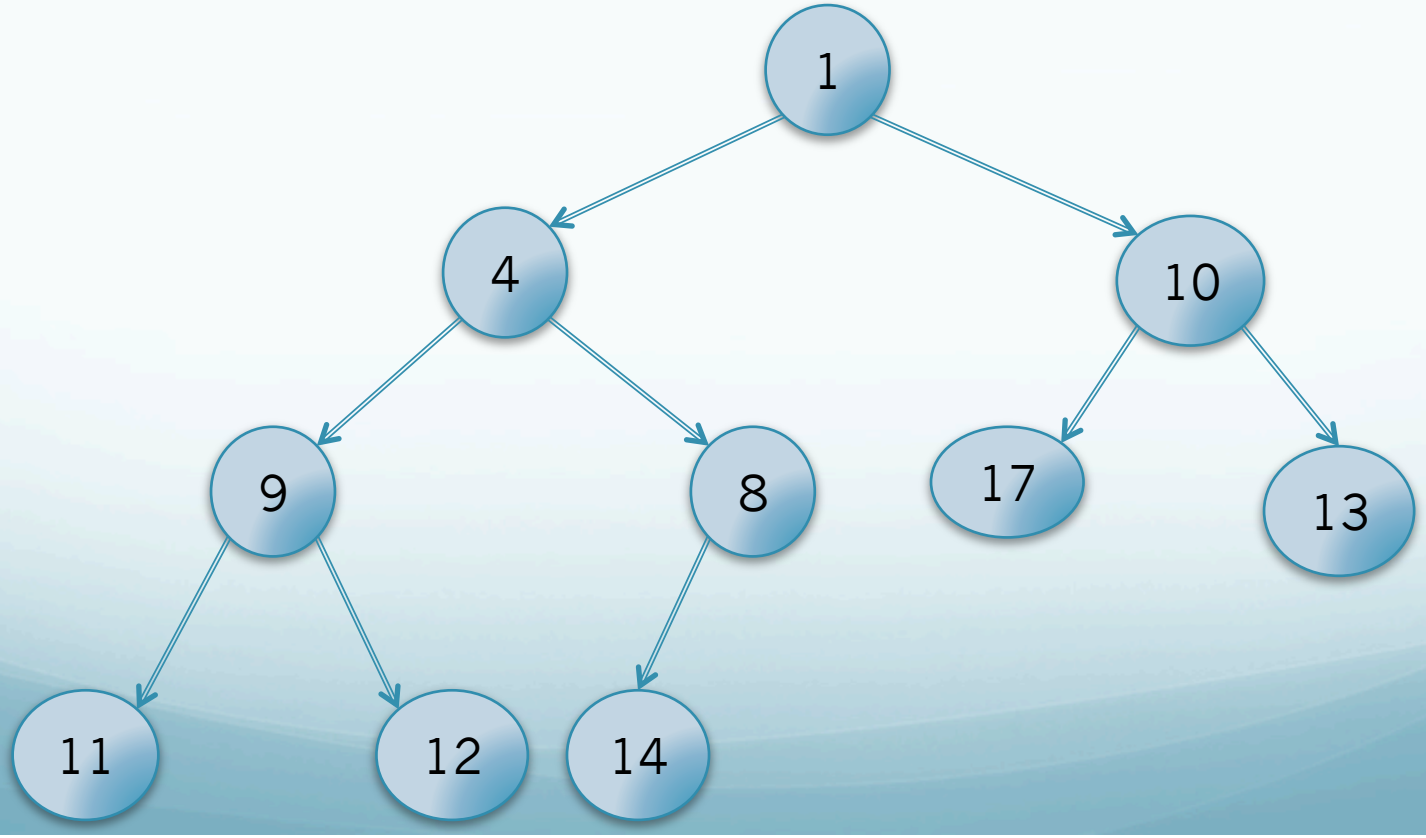
Min-heap: proprietà

- L'elemento più piccolo viene memorizzato nella radice



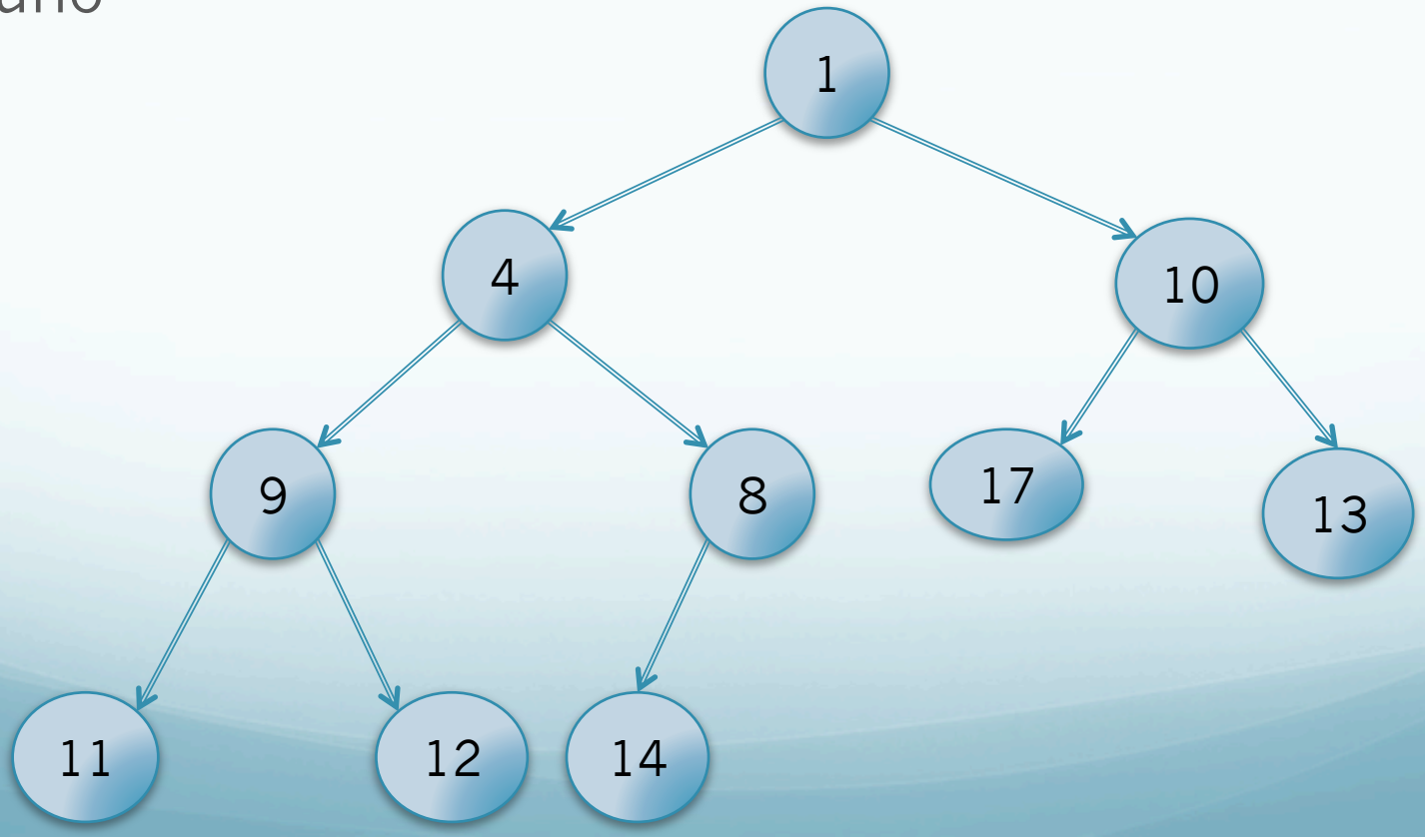
Min-heap: proprietà

- Tutti i valori dei nodi presenti nel sottoalbero di un nodo (u) sono maggiori o uguali del valore del nodo u



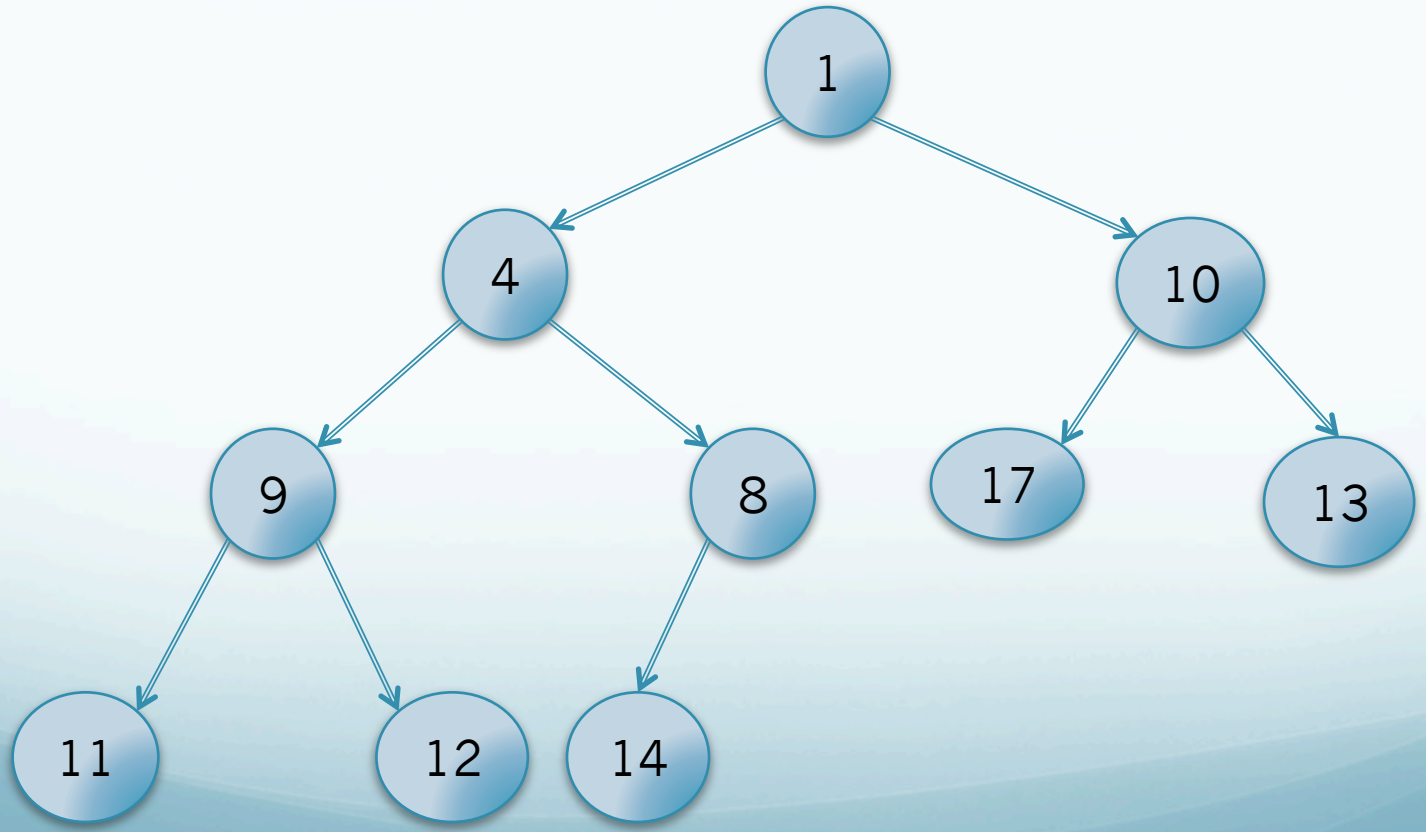
Min-heap: proprietà

- Tutte le foglie hanno profondità *h* o *h-1*
- Tutti i nodi interni hanno grado 2, eccetto al più uno



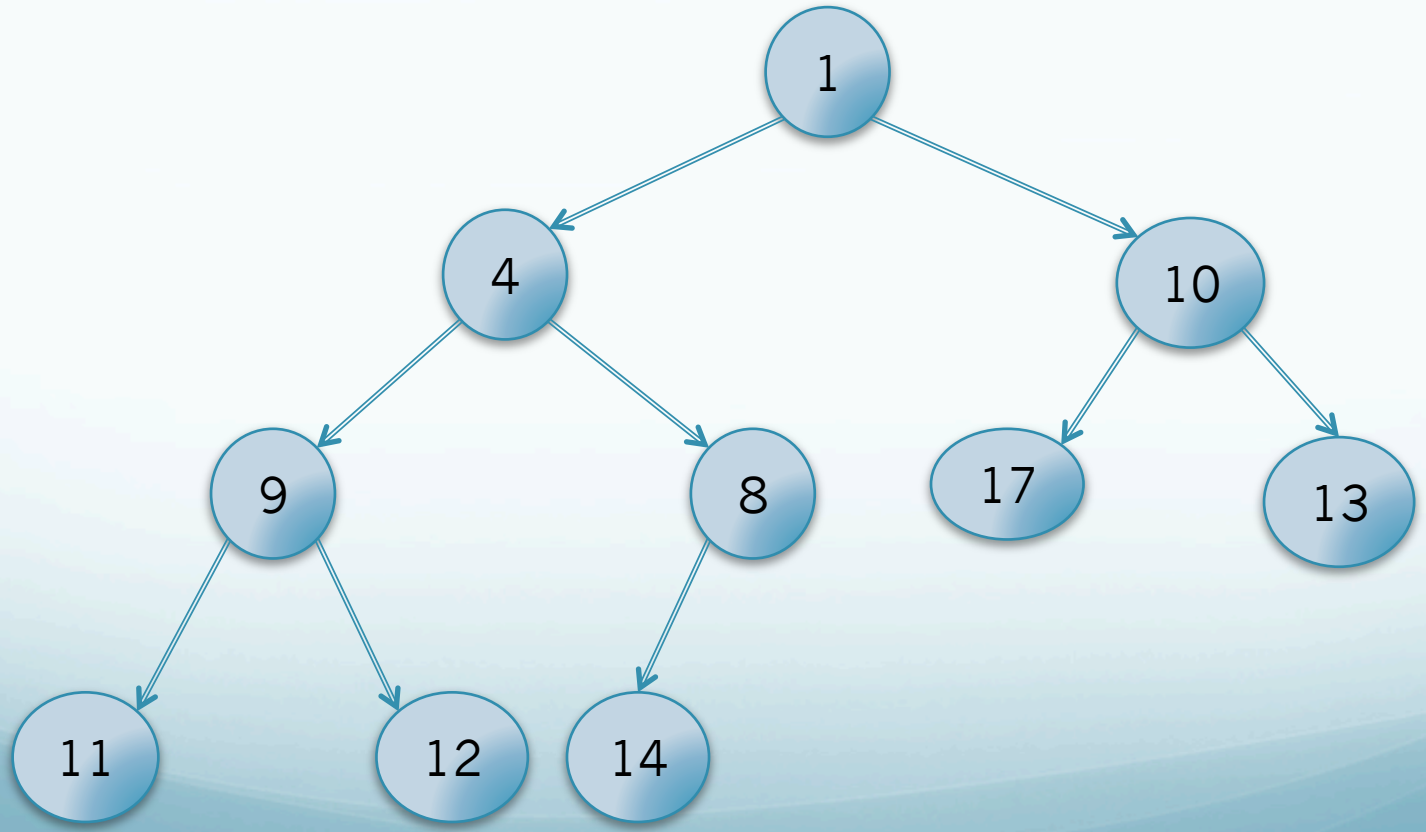
Min-heap: proprietà

- Se h è il livello massimo delle foglie allora ci sono esattamente 2^h-1 nodi di livello minore di h



Min-heap: proprietà

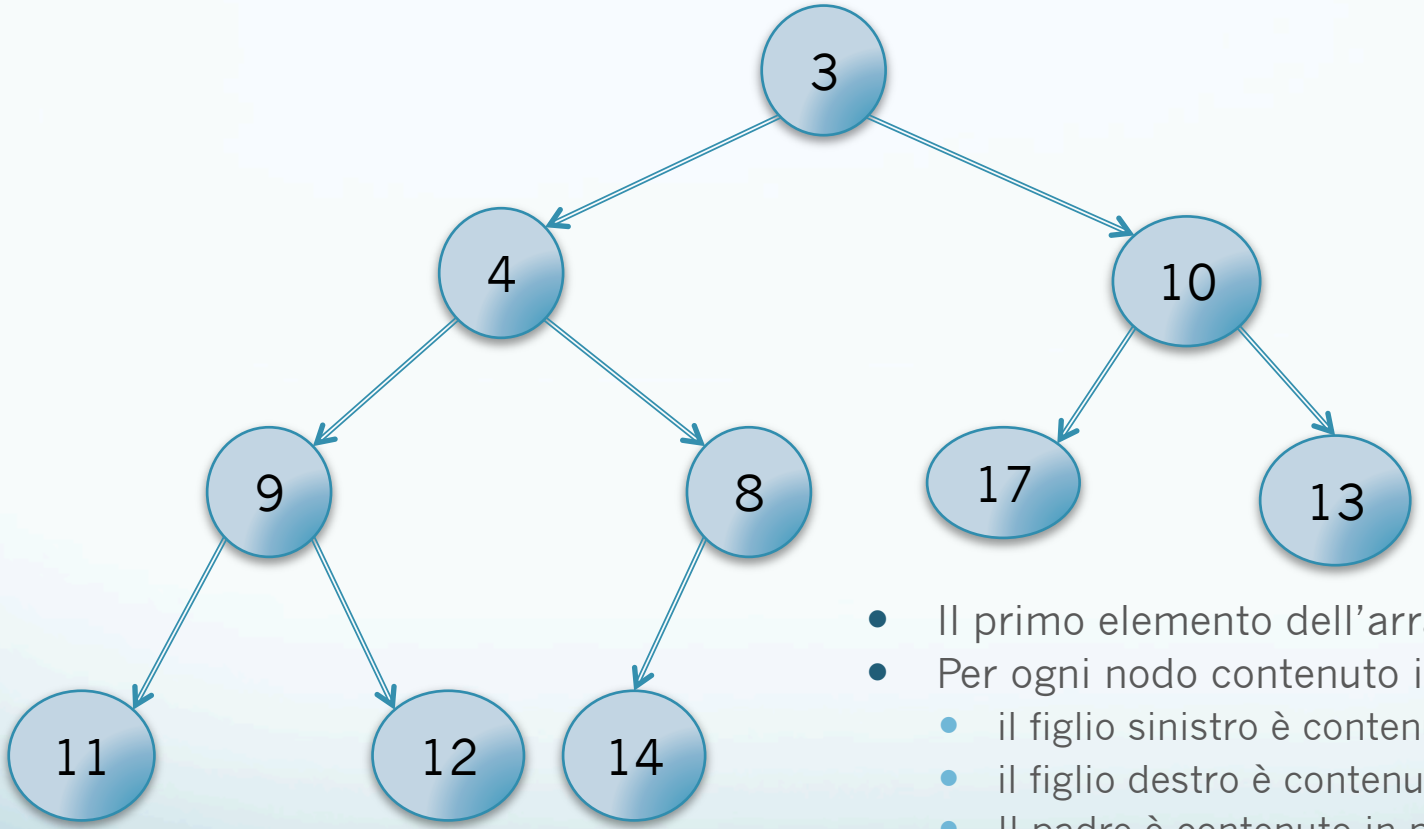
- La “quasi” completezza garantisce che l’altezza di un heap con n nodi `e Θ(log2 n)



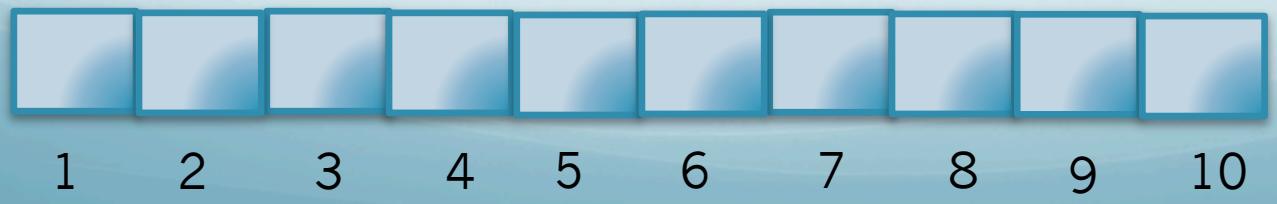
Min-heap: implementazione

- Uno heap può essere rappresentato con un array A nel quale:
  - Il primo elemento dell'array contiene la radice
  - Per ogni nodo contenuto in posizione i:
    - il figlio sinistro è contenuto in posizione 2*i
    - il figlio destro è contenuto in posizione 2*i + 1
    - Il padre è contenuto in posizione i/2

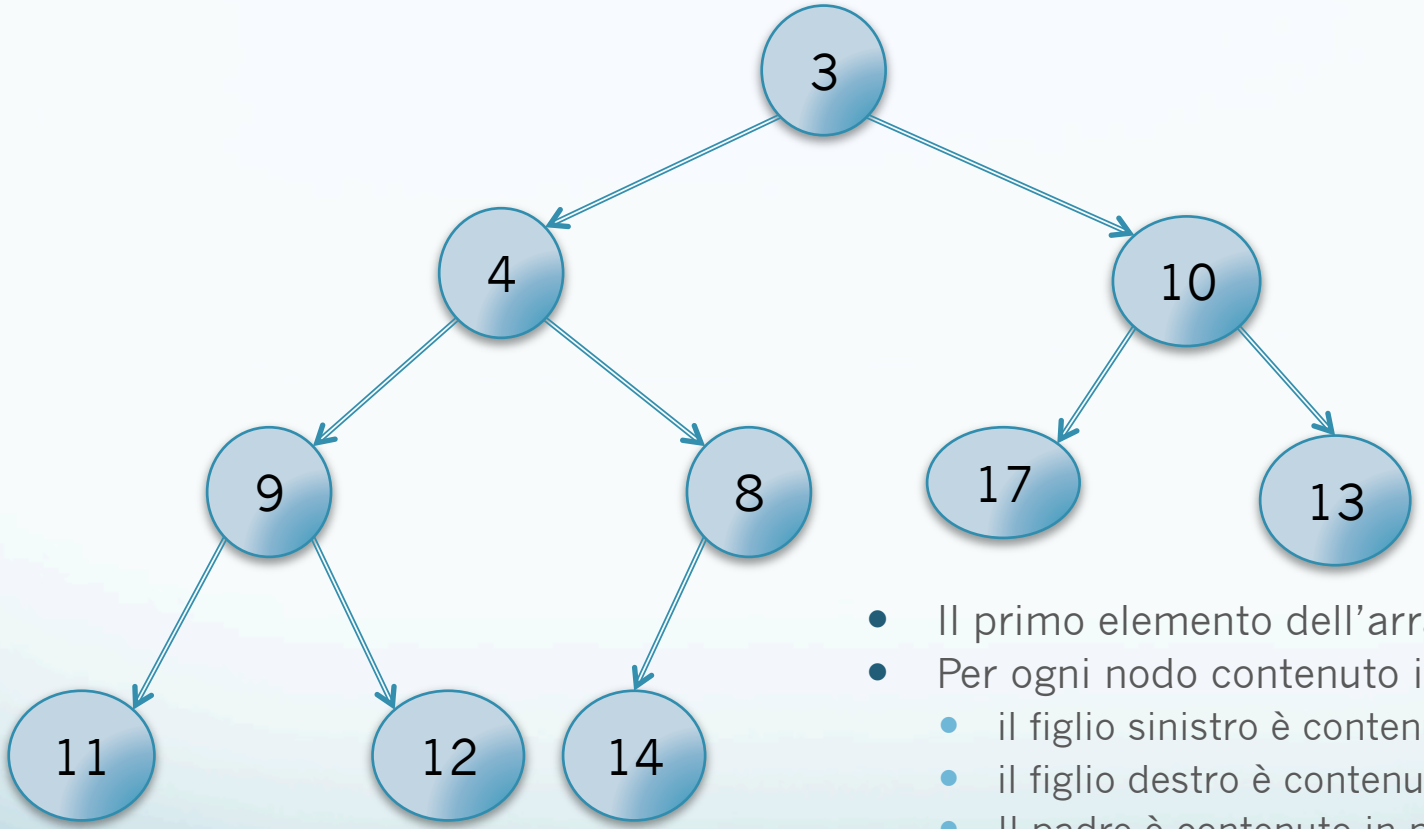
Min-heap: implementazione



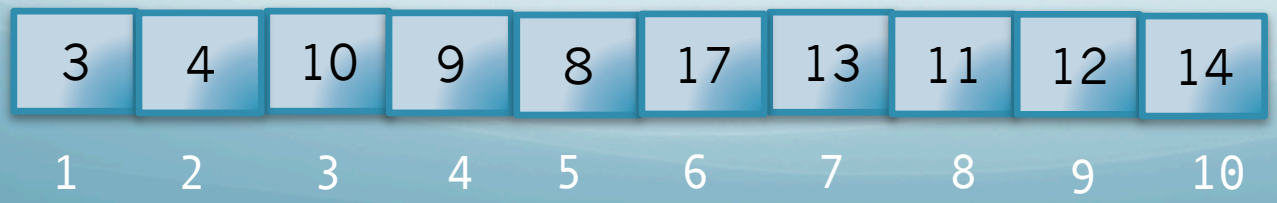
- Il primo elemento dell'array contiene la radice
- Per ogni nodo contenuto in posizione i:
  - il figlio sinistro è contenuto in posizione 2*i
  - il figlio destro è contenuto in posizione 2*i + 1
  - Il padre è contenuto in posizione i/2



Min-heap: implementazione

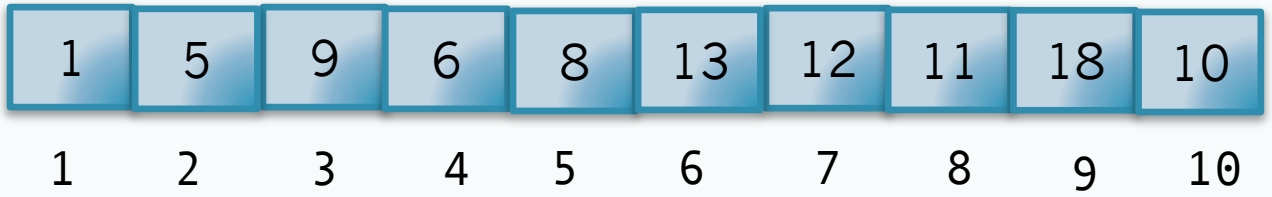


- Il primo elemento dell'array contiene la radice
- Per ogni nodo contenuto in posizione i:
  - il figlio sinistro è contenuto in posizione 2*i
  - il figlio destro è contenuto in posizione 2*i + 1
  - Il padre è contenuto in posizione i/2



Esercizio

- Costruire l'albero binario a partire dalla rappresentazione proposta in questo heap:



Code con priorità

- Struttura di dati che serve a mantenere un insieme S di elementi, ciascuno con un valore associato di priorità

Code con priorità

- Struttura di dati che serve a mantenere un insieme S di elementi, ciascuno con un valore associato di priorità

PRIORITYITEM	
integer *priorità*	% Priorità
ITEM *valore*	% Elemento
integer *pos*	% Posizione nel vettore heap

```
public class PriorityItem<E,P extends Comparable> {

    private P priority;
    private E value;
    private int position;
…
}
```

Specifica

PRIORITYQUEUE

% Crea una coda con priorità vuota
PriorityQueue()

% Restituisce **true** se la coda con priorità è vuota
boolean isEmpty()

% Restituisce l'elemento minimo di una coda con priorità non vuota
ITEM min()

% Rimuove e restituisce il minimo da una coda con priorità non vuota
ITEM deleteMin()

% Inserisce l'elemento x con priorità p in una coda con priorità non piena
% Restituisce un oggetto PRIORITYITEM che identifica x all'interno della coda
PRIORITYITEM insert(ITEM x, **integer** p)

% Diminuisce la priorità dell'elemento identificato da x portandola al valore p
decrease(PRIORITYITEM x, **integer** p)

PRIORITYQUEUE

```
integer capacità                          % Numero massimo di elementi nella coda
integer dim                               % Numero attuale di elementi nella coda
PRIORITYITEM[] H                          % Vettore heap

PRIORITYQUEUE PriorityQueue(integer n)
    PRIORITYQUEUE t ← new PRIORITYQUEUE
    t.capacità ← n
    t.dim ← 0
    t.H ← new PRIORITYITEM[1 . . . n]
    return t
```

PRIORITYQUEUE

```
integer capacità                          % Numero massimo di elementi nella coda
integer dim                               % Numero attuale di elementi nella coda
PRIORITYITEM[] H                          % Vettore heap

PRIORITYQUEUE PriorityQueue(integer n)
    PRIORITYQUEUE t ← new PRIORITYQUEUE
    t.capacità ← n
    t.dim ← 0
    t.H ← new PRIORITYITEM[1...n]
    return t
```

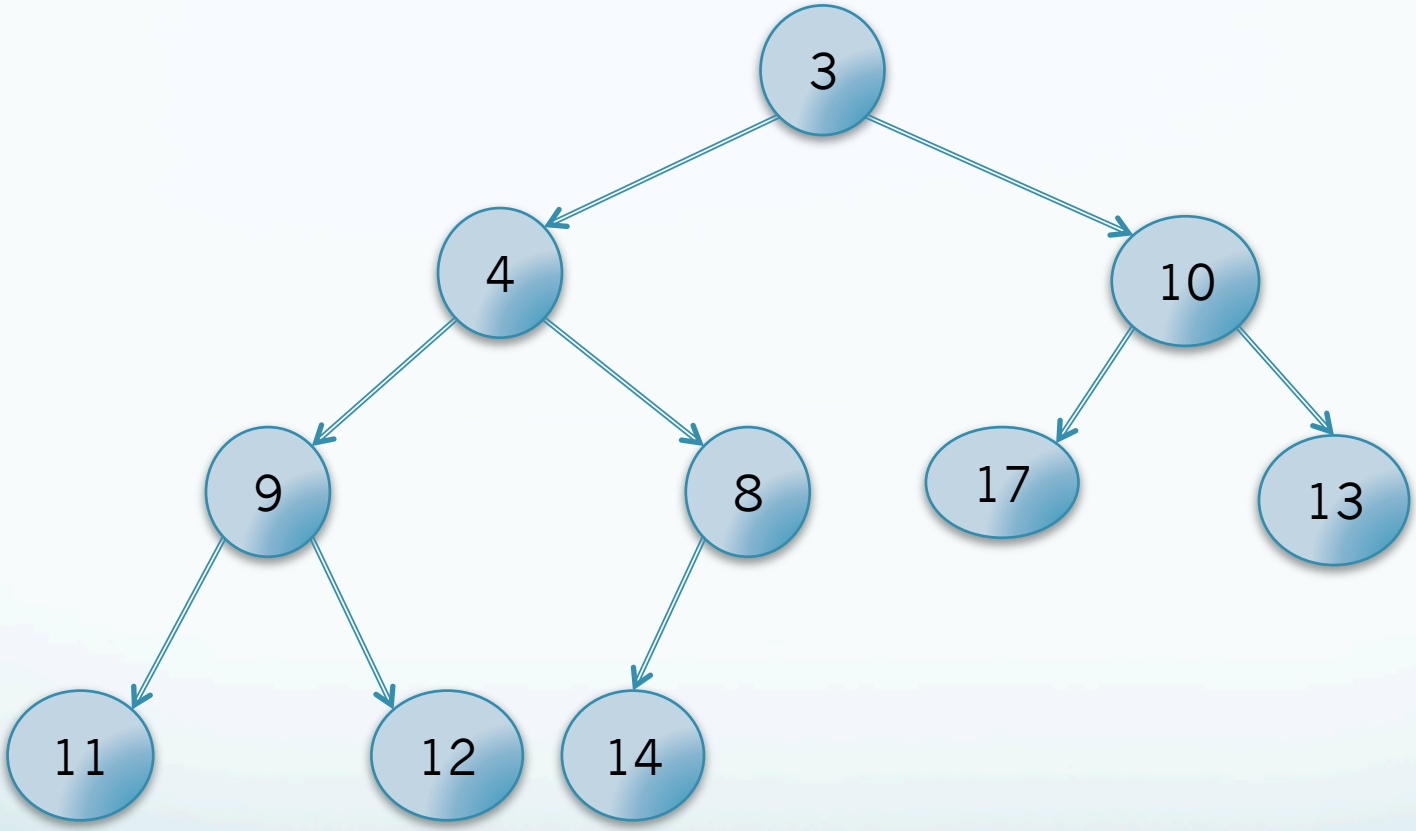
```
public class PriorityQueue<E, P extends Comparable> {

    private int capacity;
    private int heapSize;
    private PriorityItem<E,P>[]  heap;

    public PriorityQueue(int capacity) {
        this.capacity = capacity;
        heapSize = 0;
        heap = new PriorityItem[this.capacity +1];

    }
```

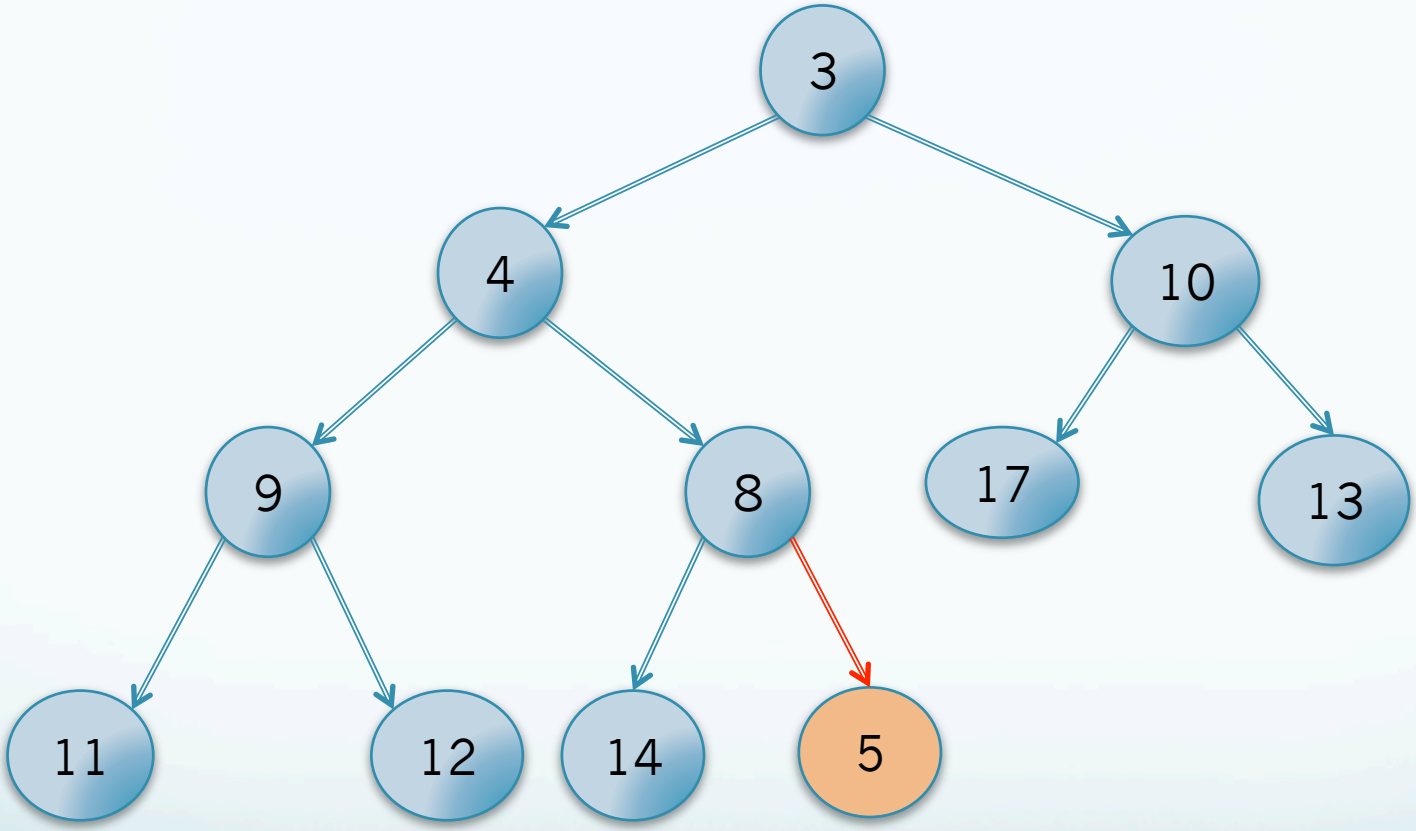
ITEM min()
precondition: $dim > 0$
return $H[1].valore$

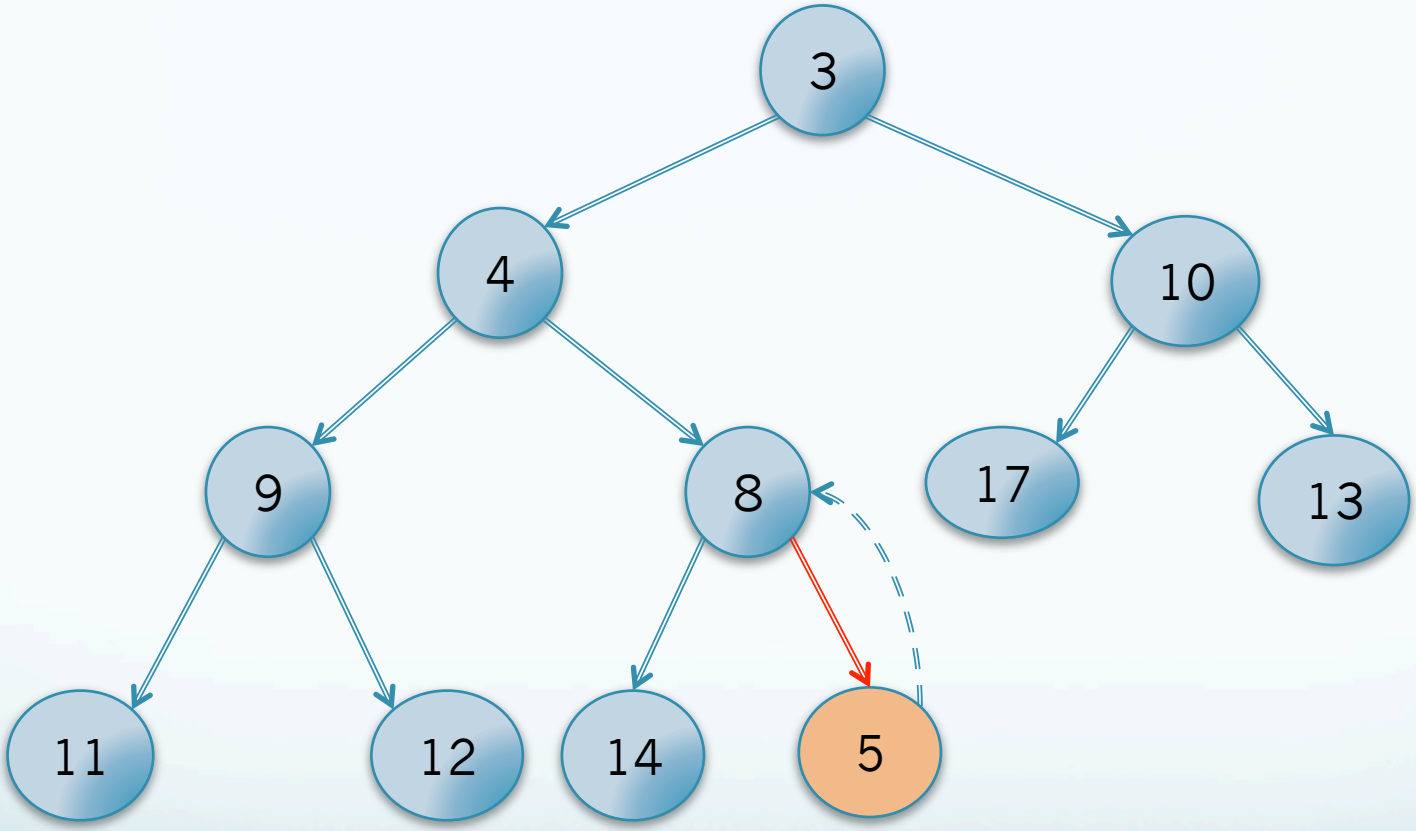
```
public E getMin() {

        if (heapSize == 0){
            throw new EmptyPriorityQueueException();
        }
        return heap[1].getValue();
    }
```

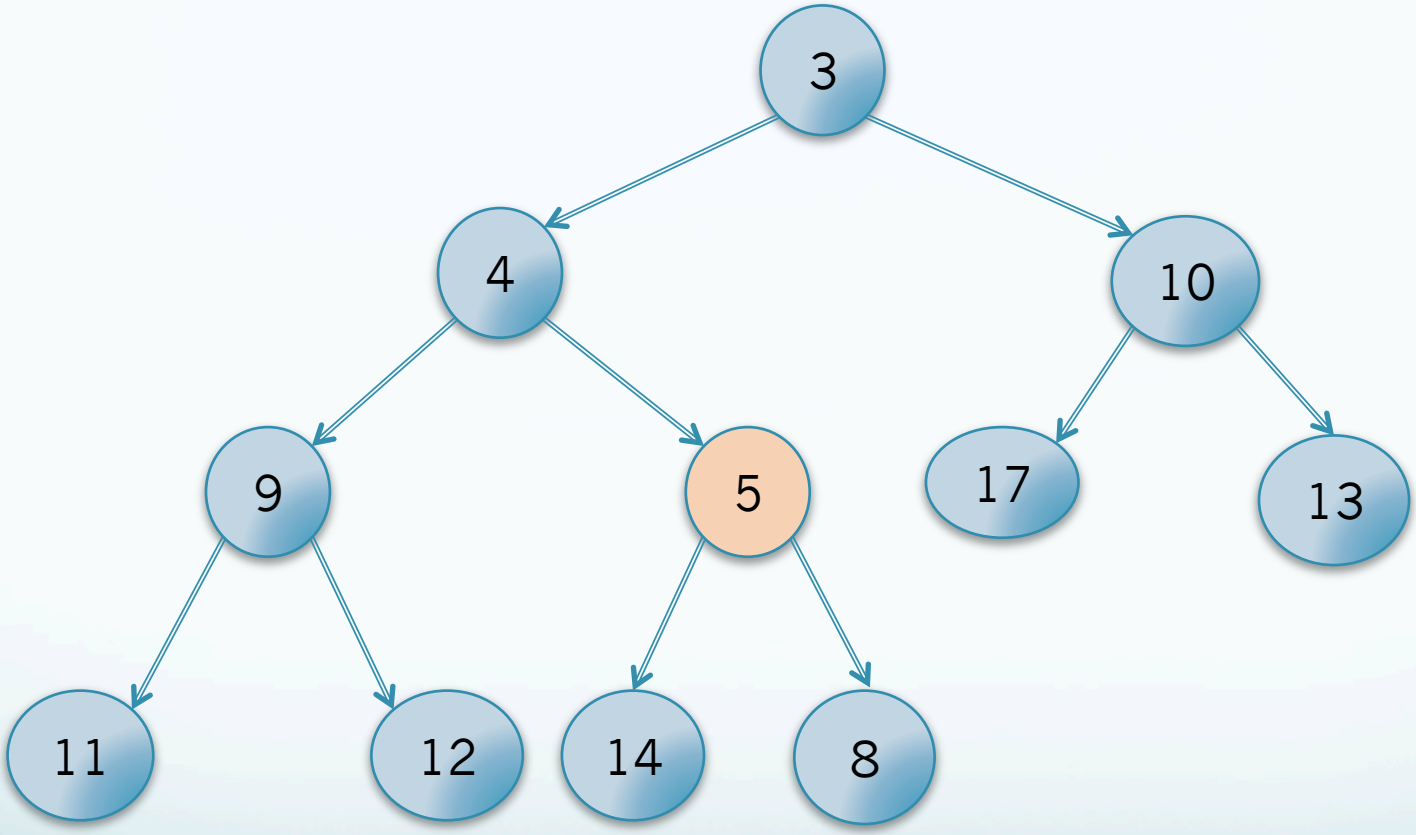
Min-heap: Inserimento

- Si inserisce un nuovo elemento come foglia posta “a destra dell’ultima foglia”, cioè l’albero viene completato per livelli.
- La nuova foglia viene poi sistemata nella posizione corretta del cammino che porta dalla sua posizione alla root.


3
4
10
9
8
17
13
11
12
14


3
4
10
9
8
17
13
11
12
14
5


3
4
10
9
8
17
13
11
12
14
5


3
4
10
9
5
17
13
11
12
14
8

PRIORITYITEM insert(ITEM x, **integer** p)

precondition: $dim < capacità$

$dim \leftarrow dim + 1$
$H[dim] \leftarrow$ **new** PRIORITYITEM()
$H[dim].valore \leftarrow x$
$H[dim].priorità \leftarrow p$
$H[dim].pos \leftarrow dim$
integer $i \leftarrow dim$
while $i > 1$ **and** $H[i].priorità < H[p(i)].priorità$ **do**
    swap$(H, i, p(i))$
    $i \leftarrow p(i)$
return $H[i]$

swap(PRIORITYITEM[ ] H, **integer** i, **integer** j)

$H[i] \leftrightarrow H[j]$
$H[i].pos \leftarrow i$
$H[j].pos \leftarrow j$

swap(PRIORITYITEM[ ] H, **integer** i, **integer** j)

$H[i] \leftrightarrow H[j]$

$H[i].pos \leftarrow i$

$H[j].pos \leftarrow j$

swap(PRIORITYITEM[ ] H, **integer** i, **integer** j)

$H[i] \leftrightarrow H[j]$
$H[i].pos \leftarrow i$
$H[j].pos \leftarrow j$

```
private void swap(int i, int j){

        PriorityItem tmp = heap[i];
        heap[i] = heap[j];
        heap[j] = tmp;

        heap[i].setPosition(i);
        heap[j].setPosition(j);

    }
```

PRIORITYITEM insert(ITEM x, **integer** p)

precondition: $dim < capacità$

$dim \leftarrow dim + 1$
$H[dim] \leftarrow$ **new** PRIORITYITEM()
$H[dim].valore \leftarrow x$
$H[dim].priorità \leftarrow p$
$H[dim].pos \leftarrow dim$
integer $i \leftarrow dim$
while $i > 1$ **and** $H[i].priorità < H[p(i)].priorità$ **do**
    swap$(H, i, p(i))$
    $i \leftarrow p(i)$
return $H[i]$

swap(PRIORITYITEM[ ] H, **integer** i, **integer** j)

$H[i] \leftrightarrow H[j]$
$H[i].pos \leftarrow i$
$H[j].pos \leftarrow j$

```
public  PriorityItem<E,P> insert(E item, P priority){

        if (heapSize == capacity  ){
            throw new FullPriorityQueueException();
        }

       heapSize = heapSize +1;
        heap[heapSize] = new PriorityItem<E,P>(priority, item, heapSize);
        int i = heapSize;

        while ( (i>1) &&
       (heap[i].getPriority().compareTo(heap[i/2].getPriority()) < 0) ){
            swap(i,i/2);
            i = i/2;
        }

       return heap[i];
    }
```

Min-heap: decrease priority

- Diminuisce la priorità di un dato elemento portandola ad un valore dato.

decrease(PRIORITYITEM x, **integer** p)

precondition: $p < x.priorità$

$x.priorità \leftarrow p$

integer $i \leftarrow x.pos$

while $i > 1$ **and** $H[i].priorità < H[p(i)].priorità$ **do**

    swap$(H, i, p(i))$

    $i \leftarrow p(i)$

decrease(PRIORITYITEM x, **integer** p)

precondition: $p < x.priorità$

$x.priorità \leftarrow p$

integer $i \leftarrow x.pos$

while $i > 1$ **and** $H[i].priorità < H[p(i)].priorità$ **do**

    swap$(H, i, p(i))$

    $i \leftarrow p(i)$

decrease(PRIORITYITEM x, **integer** p)

precondition: $p < x.priorità$

$x.priorità \leftarrow p$

integer $i \leftarrow x.pos$

while $i > 1$ **and** $H[i].priorità < H[p(i)].priorità$ **do**

    swap$(H, i, p(i))$

    $i \leftarrow p(i)$

```
public void decrease(PriorityItem<E,P> x, P priority){

        if (priority.compareTo(x.getPriority()) > 0){
            throw new WrongPriorityException();
        }
        x.setPriority(priority);
        int i = x.getPosition();
        while ( (i>1) &&
            (heap[i].getPriority().compareTo(heap[i/2].getPriority())<0))
        {
            swap(i,i/2);
            i = i/2;
        }
    }
```

```
public void decrease(PriorityItem<E,P> x, P priority){

        if (priority.compareTo(x.getPriority()) > 0){
            throw new WrongPriorityException();
        }
        x.setPriority(priority);
        int i = x.getPosition();
        while ( (i>1) &&
            (heap[i].getPriority().compareTo(
             heap[i/2].getPriority())<0))     {
            swap(i,i/2);
            i = i/2;
        }
    }
```

Min-heap: rimuovi minimo

- Si elimina l'ultima foglia inserita e si salva il suo valore e la sua chiave nella radice
- Si ripristina la proprietà del min-heap (usando la procedura minHeapRestore)

```
ITEM deleteMin(ITEM x)
    precondition: dim > 0
    swap(H, 1, dim)
    dim ← dim − 1
    minHeapRestore(H, 1, dim)
    return H[dim + 1]
```

```
minHeapRestore(PRIORITYITEM[] A, integer i, integer dim)
    integer min ← i
    if l(i) ≤ dim and A[l(i)].priorità < A[min].priorità then min ← l(i)
    if r(i) ≤ dim and A[r(i)].priorità < A[min].priorità then min ← r(i)
    if i ≠ min then
        swap(A, i, min)
        minHeapRestore(A, min, dim)
```

ITEM deleteMin(ITEM x)

precondition: $dim > 0$

swap$(H, 1, dim)$

$dim \leftarrow dim - 1$

minHeapRestore$(H, 1, dim)$

return $H[dim + 1]$

```
public E deleteMin(){

        if (heapSize == 0 ){
            throw new EmptyPriorityQueueException();
        }

        E minItem = heap[1].getValue();

        swap(1,heapSize);
        heapSize = heapSize-1;
        minHeapRestore(1);
        return minItem;
    }
```

```
minHeapRestore(PRIORITYITEM[] A, integer i, integer dim)
    integer min ← i
    if l(i) ≤ dim and A[l(i)].priorità < A[min].priorità then min ← l(i)
    if r(i) ≤ dim and A[r(i)].priorità < A[min].priorità then min ← r(i)
    if i ≠ min then
        swap(A, i, min)
        minHeapRestore(A, min, dim)
```

```
private void minHeapRestore(int i){

        int positionMinPriority = i;
        int positionLeftSon = i*2;
        int positionRightSon = i*2+1;

        if ((positionLeftSon <= heapSize) &&
                (heap[positionLeftSon].getPriority().compareTo
             (heap[positionMinPriority].getPriority())<0)){
            positionMinPriority = positionLeftSon;
        }


        if ((positionRightSon <= heapSize) &&
                (heap[positionRightSon].getPriority().compareTo
             (heap[positionMinPriority].getPriority())<0)){
            positionMinPriority = positionRightSon;
        }

        if (i != positionMinPriority){
            swap(i,positionMinPriority);
            minHeapRestore(positionMinPriority);
        }
    }
```

Esercizi

- Modificare il codice del min-heap in modo tale da implementare un max heap
- Modificare il codice del min-heap in modo tale che la radice dell'albero sia memorizzata nella posizione di indice 0 dell'array.

Bibliografia

- http://disi.unitn.it/~montreso/asd/lucidi/10-heapmfset-up.pdf
- http://www.cs.unicam.it/merelli/algoritmi06/[05]heap.pdf
- http://www.dsi.unive.it/~labasd/lezioni/lezione09/lezione9.pdf