Heap

Luca Becchetti

Presentazione tratta dalle slide che accompagnano il testo Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

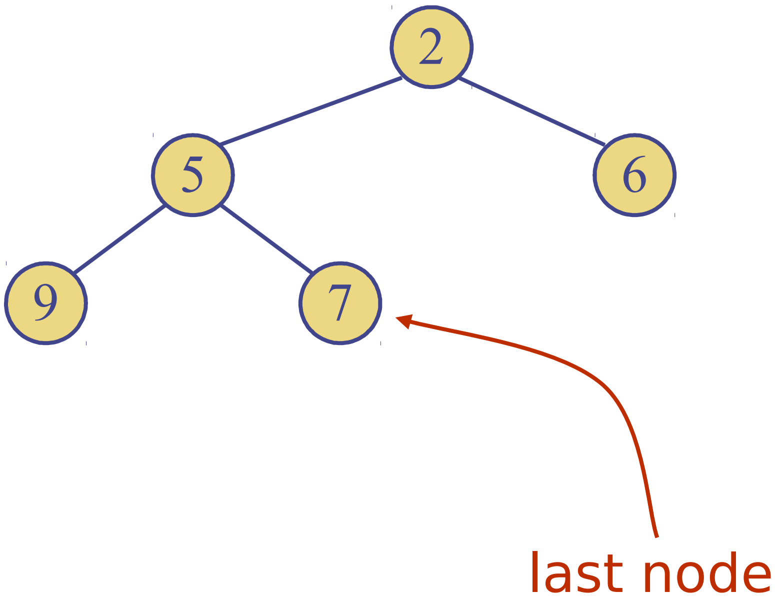
Coda di priorità → ADT

- **Collezione di elementi**
- **Ogni elemento → (chiave, valore)**
- **Operazioni fondamentali**
  - insert(k, v)
  - RemoveMin()
    - Rimuove e restituisce elemento con chiave minima
    - restituisce **null** se coda vuota
- **Operazioni ulteriori**
  - min(): restituisce (senza rimuovere) l'elemento con chiave minima o **null** se coda vuota
  - size(), isEmpty()
- Applicazioni
  - Liste di attesa
  - Aste on-line
  - Mercati azionari

Heap

- **Albero binario con chiavi associate ai vertici**
- **Proprietà di heap: Per ogni v ≠ radice**
  - *key(v) ≥ key(parent(v))*
  - Min Heap
  - Max Heap definito in modo analogo
- **Albero binario completo**
  - *Per motivi di efficienza*
  - I livelli da 0 a h-1 sono completi
  - A livello h: I nodi si trovano nella posizioni più a sinistra

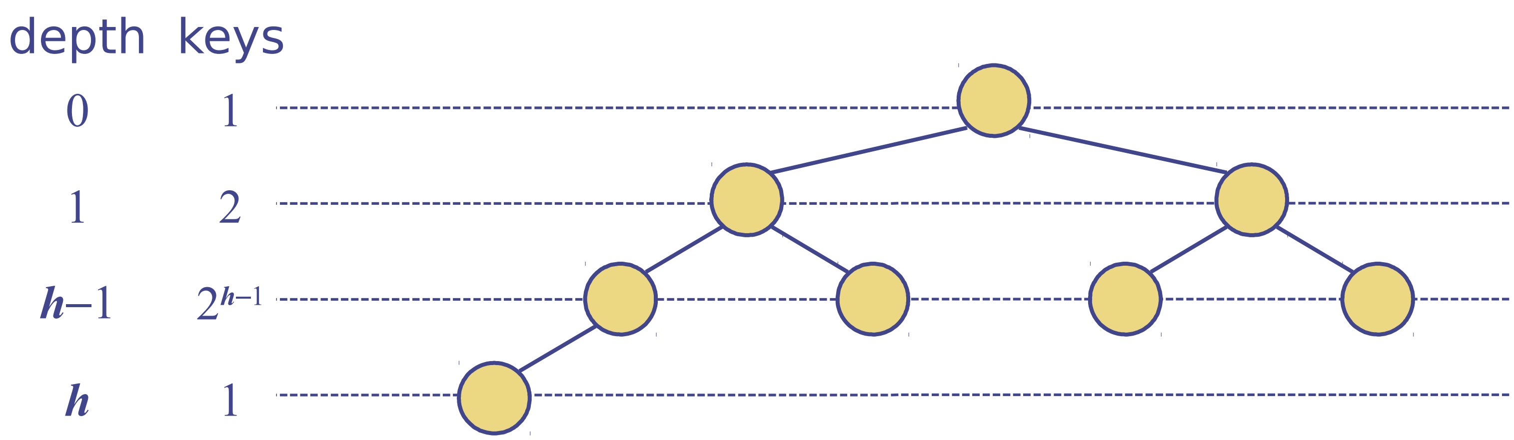
- ***Ultimo nodo:* nodo più a destra avente profondità max**



Altezza di un heap

- **Un heap con n chiavi ha altezza O(log n)**

  Prova (v. libro di testo)



Resto della lezione

1) **Heap e code di priorità**

2) **Realizzazione di *insert(k, v) e removeMin()* usando un heap**

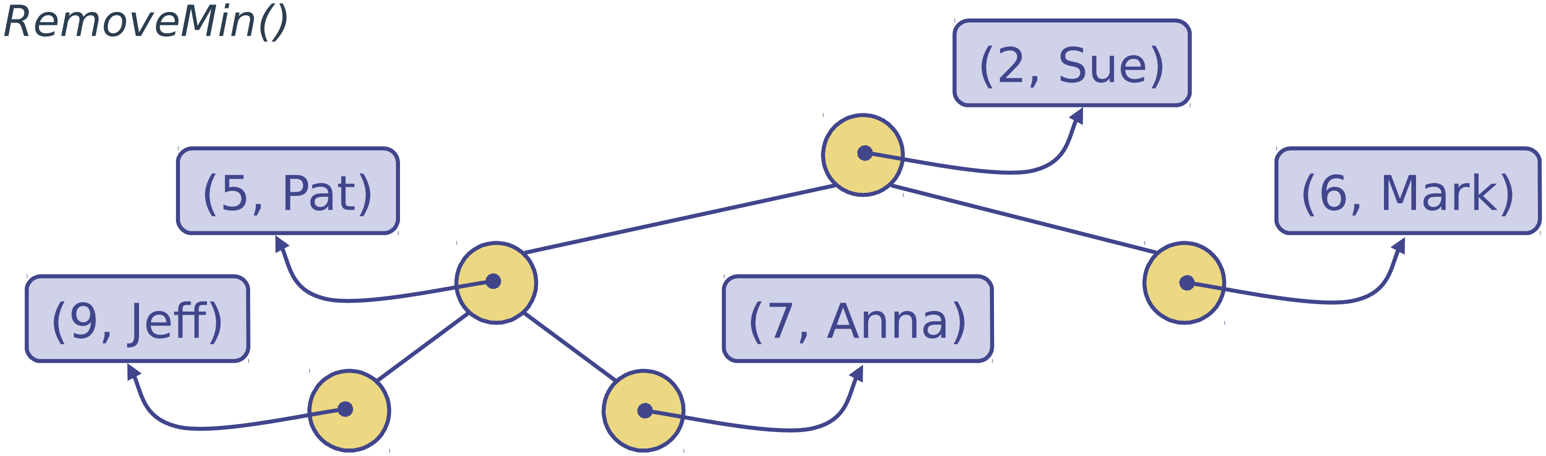
3) **Ordinamento e heap → Heap-sort**

4) **Rappresentazione di heap mediante array**

5) **Creazione bottom-up di un heap**

Heap e code di priorità

- **Possiamo usare heap per implementare code di priorità in modo efficiente**
- **Associamo coppie (chiave, elemento) ai nodi**
- **Teniamo traccia della posizione dell' *ultimo nodo***
- **Operazioni principali da implementare:**
  - *insert(k, v)*
  - *RemoveMin()*

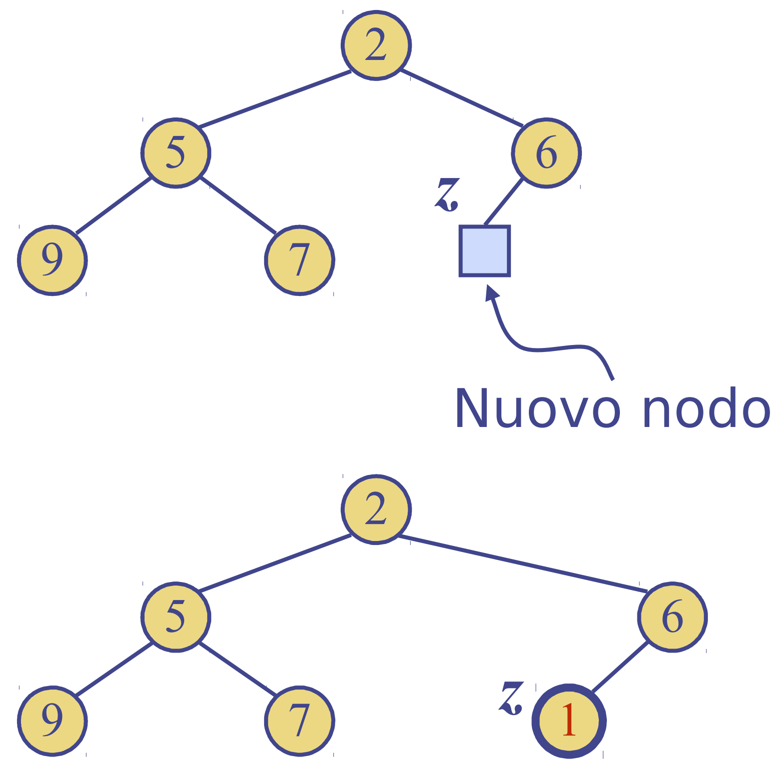


Inserimento

insert(k, v)

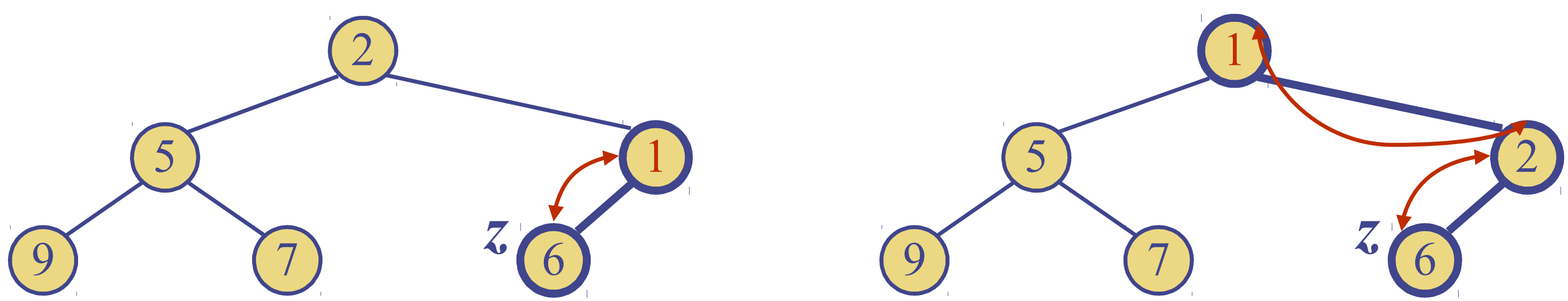
Algoritmo di inserimento

- **3 passi:**
  - Si trova la posizione in cui inserire il nuovo nodo
    - Si ricordi che teniamo traccia del nodo più a destra sull'ultimo livello
  - Creare il nuovo nodo
  - Ricreare ordinamento dell'heap (prossima slide)



Up-heap bubbling

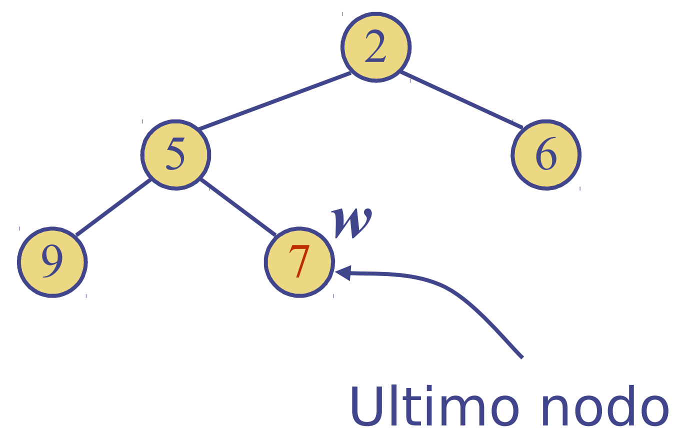
- **Proprietà di heap potrebbe essere violata dopo l'inserimento**
- **l'algoritmo *upheap* fa risalire la nuova chiave *k* verso la posizione giusta**
  - Scambia con genitore ogni volta che ordinamento di heap è violato
  - Terminazione: si raggiunge la radice o un nodo il cui genitore ha chiave ≤ *k*
- ***Complessità di upheap?***

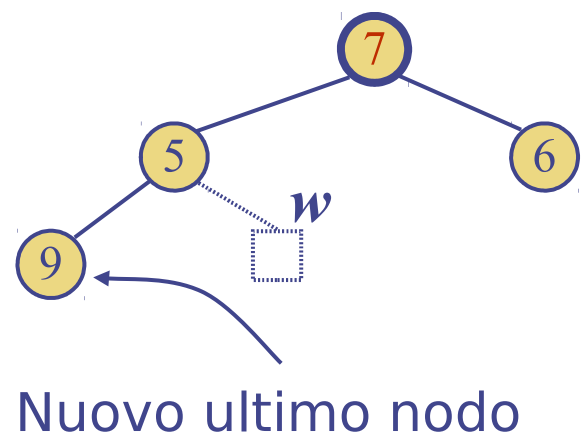


Rimozione chiave minima
removeMin()

Algoritmo di rimozione

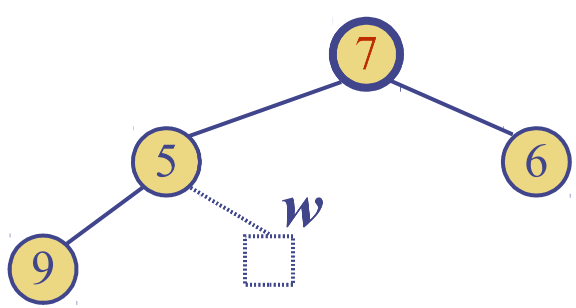
- **Si rimuove la radice**
- **Corrisponde all'elemento avente chiave minima**
- **3 passi**
  - Si copia la radice e se ne sostituisce la chiave con quella dell'*ultimo nodo*
  - Si rimuove l'ultimo nodo
  - Si ripristina la proprietà di heap (v. prossima slide)

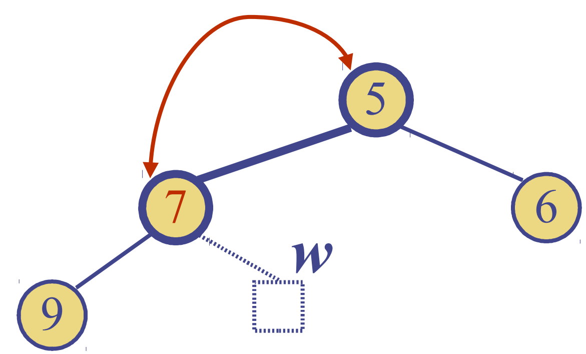




Downheap

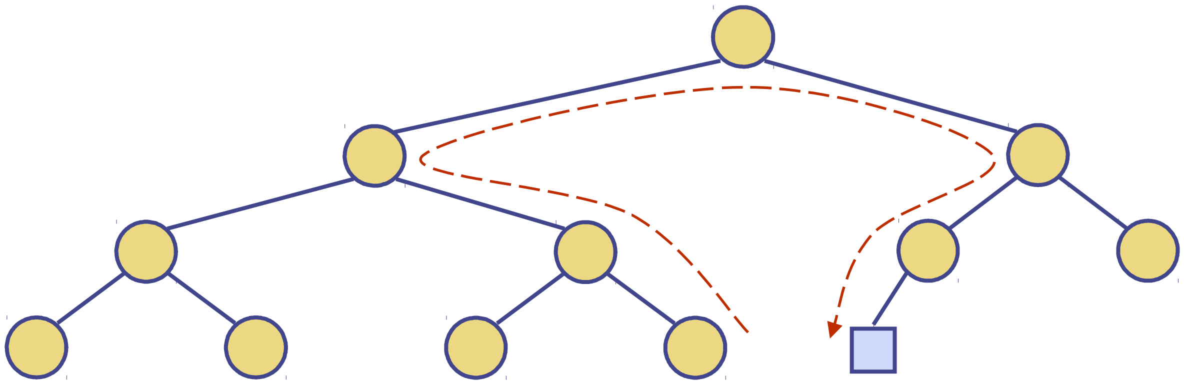
- **La proprietà di heap potrebbe essere violata dopo i primi due passi**
- **Si ripristina facendo "scendere" la chiave della radice verso il basso**
  - Si scambia con la chiave minore tra quelle dei figli
  - Terminazione: il nodo viene collocato come foglia o proprietà di heap rispettata
- **Complessità di Downheap?**





Ultimo nodo

- Trovare/aggiornare l'ultimo nodo è semplice nell'implementazione che usa array (v. più avanti)
- Leggermente più complessa se si usa una struttura collegata



Heap sort

- **Eseguiamo PQSort ma la coda di priorità è implementata mediante heap**

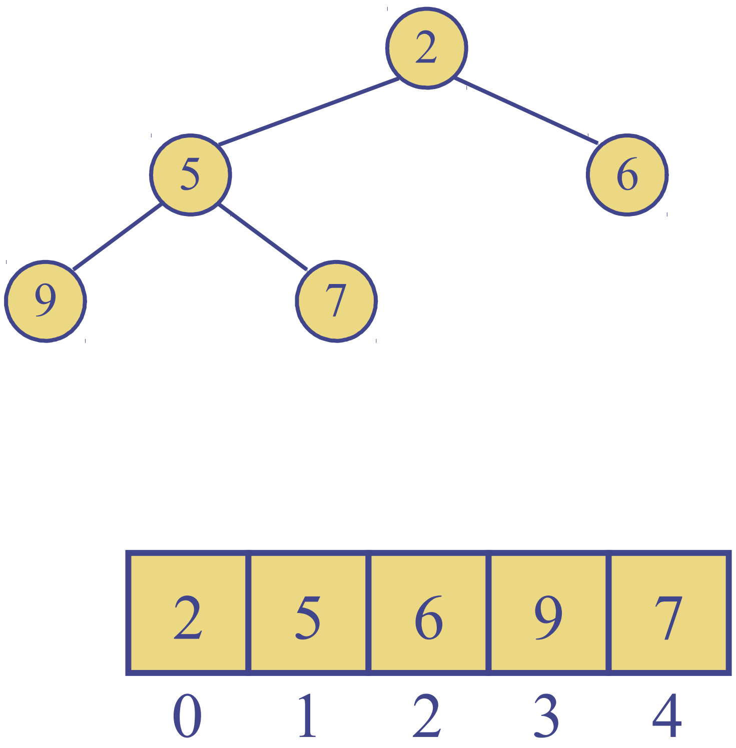
```
HeapSort(S) //S: lista/array da ordinare
// Usiamo un Heap che chiameremo heap
Output: <array a ordinato rispetto alle chiavi degli elementi in S>
while (!S.isEmpty):
    heap.insert(S.remove())
    // Assumiamo che gli elementi di S siano coppie (k, v)
    // S.remove() rimuove un elemento da S secondo un criterio qualsiasi
while (!heap.isEmpty):
    a.add(heap.removeMin()) //add aggiunge alla fine dell'array
return a
```

Dimostrare che la complessità dell'algoritmo è O(nlog n)

Implementazione di heap con array

Rappresentazione

- **Rappresentiamo albero binario completo con array**
- **Per un nodo in posizione i nell’array**
  - Figlio sx in posizione 2i + 1
  - Figlio dx in posizione 2i + 2



Operazioni elementari

- **Abbiamo bisogno delle seguenti operazioni elementari:**
  - parent(i) → return j = (i-1)/2 (divisione con troncamento)
    - La radice è in posizione 0
    - I figli sinistri sono in posizioni dispari
      - Se i dispari → i = 2j + 1
    - I figli destri sono in posizioni pari
      - Se i pari → i = 2j + 2
  - Ultimo nodo → occupa semplicemente l’ultima posizione nell’array

Esempio di implementazione (Java)

Implementazione/1

```
/** An implementation of a priority queue using an array-based heap. */
public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  protected ArrayList<Entry<K,V>> heap = new ArrayList<>();
  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public HeapPriorityQueue() { super(); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public HeapPriorityQueue(Comparator<K> comp) { super(comp); }
  // protected utilities
  protected int parent(int j) { return (j−1) / 2; }          // truncating division
  protected int left(int j) { return 2*j + 1; }
  protected int right(int j) { return 2*j + 2; }
  protected boolean hasLeft(int j) { return left(j) < heap.size(); }
  protected boolean hasRight(int j) { return right(j) < heap.size(); }
  /** Exchanges the entries at indices i and j of the array list. */
  protected void swap(int i, int j) {
    Entry<K,V> temp = heap.get(i);
    heap.set(i, heap.get(j));
    heap.set(j, temp);
  }
  /** Moves the entry at index j higher, if necessary, to restore the heap property. */
  protected void upheap(int j) {
    while (j > 0) {               // continue until reaching root (or break statement)
      int p = parent(j);
      if (compare(heap.get(j), heap.get(p)) >= 0) break;   // heap property verified
      swap(j, p);
      j = p;                                  // continue from the parent's location
    }
  }
```

Implementazione/2

```
/** Moves the entry at index j lower, if necessary, to restore the heap property. */
protected void downheap(int j) {
  while (hasLeft(j)) {                    // continue to bottom (or break statement)
    int leftIndex = left(j);
    int smallChildIndex = leftIndex;          // although right may be smaller
    if (hasRight(j)) {
        int rightIndex = right(j);
        if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
          smallChildIndex = rightIndex;       // right child is smaller
    }
    if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
      break;                                  // heap property has been restored
    swap(j, smallChildIndex);
    j = smallChildIndex;                      // continue at position of the child
  }
}

// public methods
/** Returns the number of items in the priority queue. */
public int size() { return heap.size(); }
/** Returns (but does not remove) an entry with minimal key (if any). */
public Entry<K,V> min() {
  if (heap.isEmpty()) return null;
  return heap.get(0);
}
```

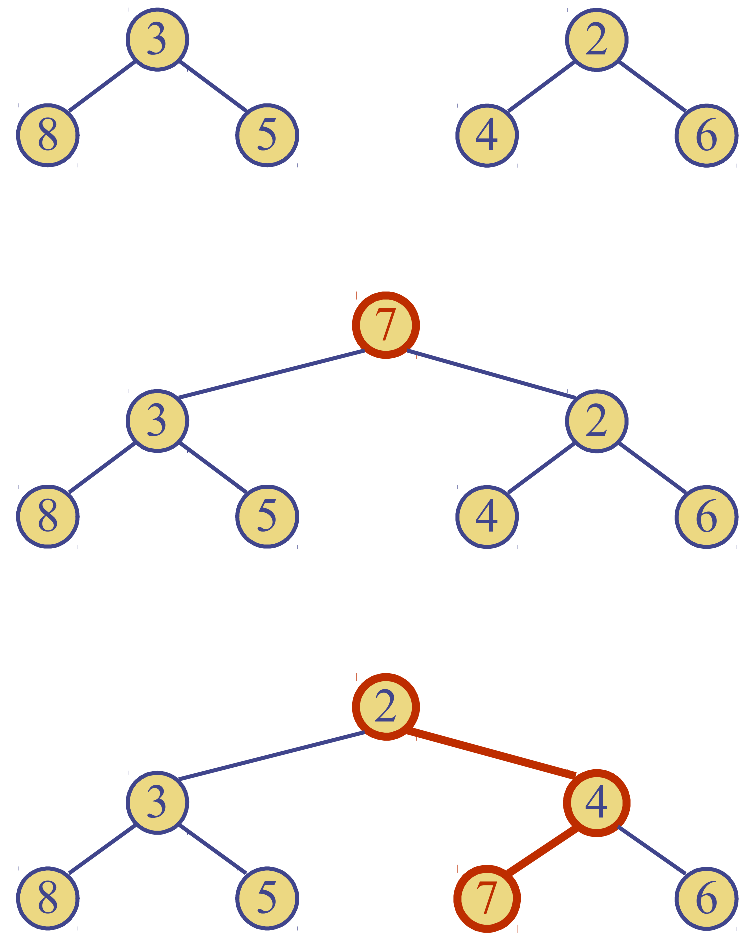
Implementazione/3

```
  /** Inserts a key-value pair and returns the entry created. */
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key);        // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    heap.add(newest);                              // add to the end of the list
    upheap(heap.size() − 1);                       // upheap newly added entry
    return newest;
  }
  /** Removes and returns an entry with minimal key (if any). */
  public Entry<K,V> removeMin() {
    if (heap.isEmpty()) return null;
    Entry<K,V> answer = heap.get(0);
    swap(0, heap.size() − 1);                      // put minimum item at the end
    heap.remove(heap.size() − 1);                  // and remove it from the list;
    downheap(0);                                   // then fix new root
    return answer;
  }
}
```

Creazione bottom-up di un heap

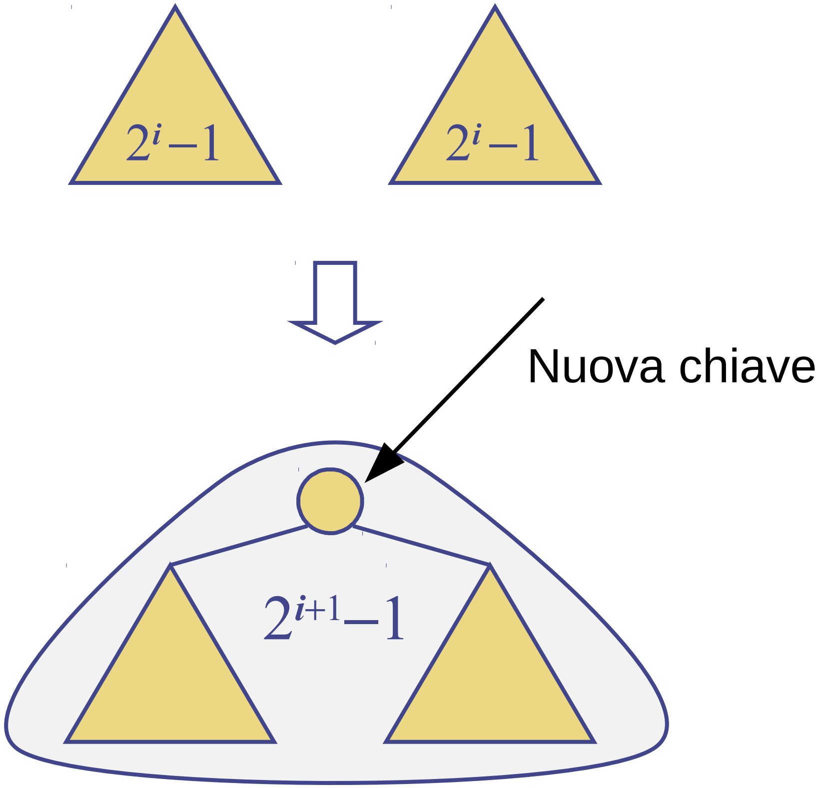
Fusione di due heap

- **Creare un heap aggiungendo chiavi (coppie) in sequenza costa O(n log n)**
- **Dati due heap e una chiave *k***
  - Nuovo heap avente la chiave come radice
  - Downheap



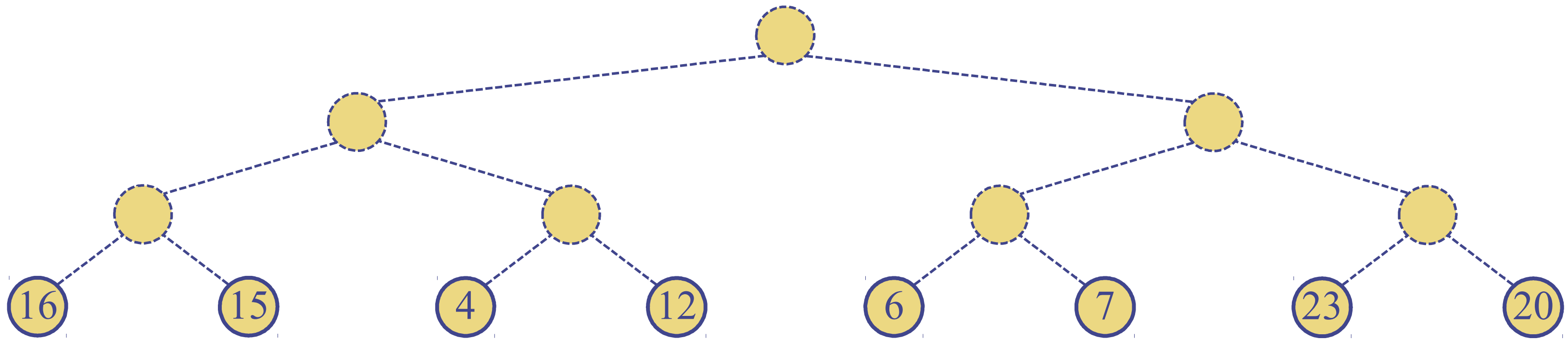
Costruzione bottom-up di un heap (heapify)

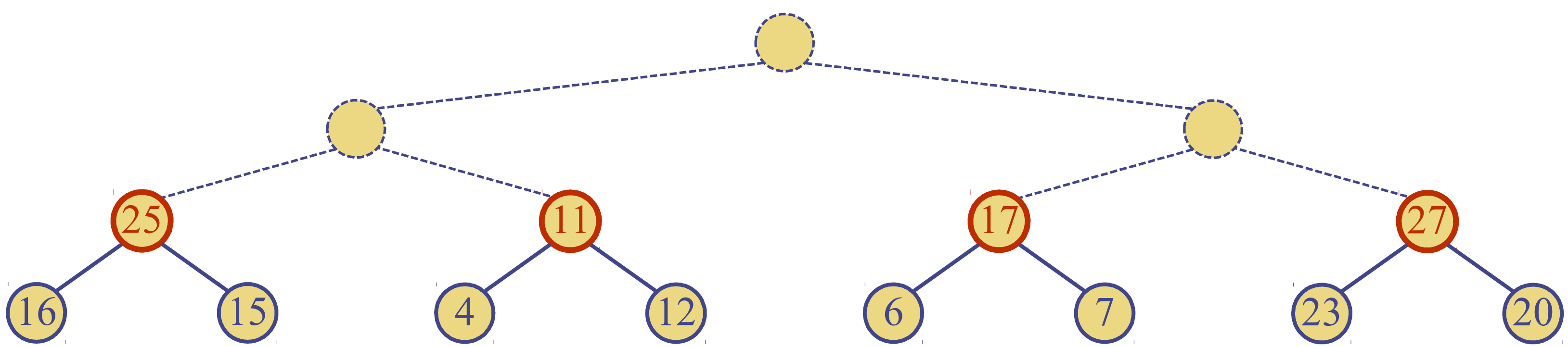
- **Heap con n chiavi può essere costruito in tempo O(n)**
- **Durante l'i-esimo passo due heap con $2^i - 1$ chiavi sono fusi in uno con $2^{i+1} - 1$ chiavi**



Codice Java disponibile tra le risorse del libro di testo

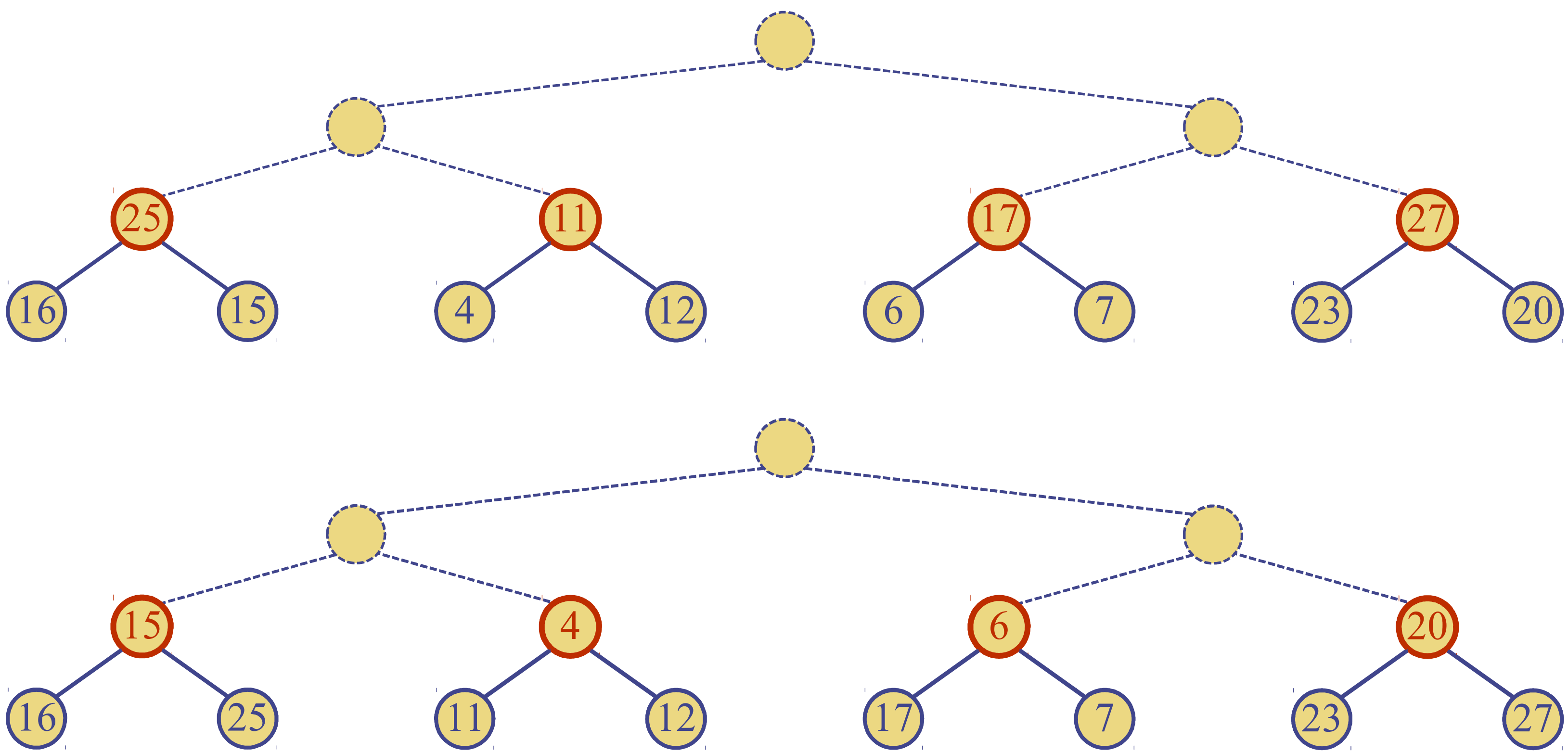
Esempio/1





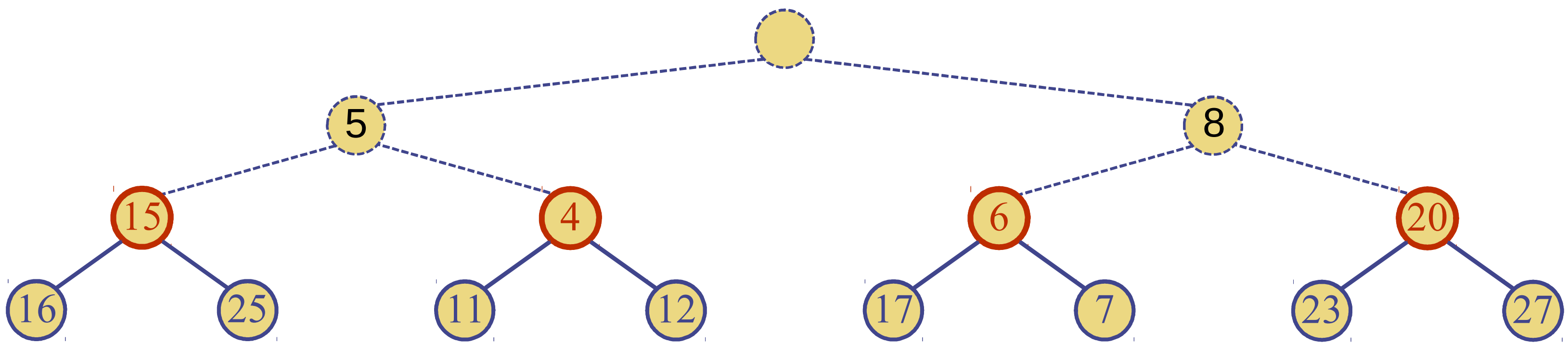
14	5	8	25	11	17	27	16	15	4	12	6	7	23	20

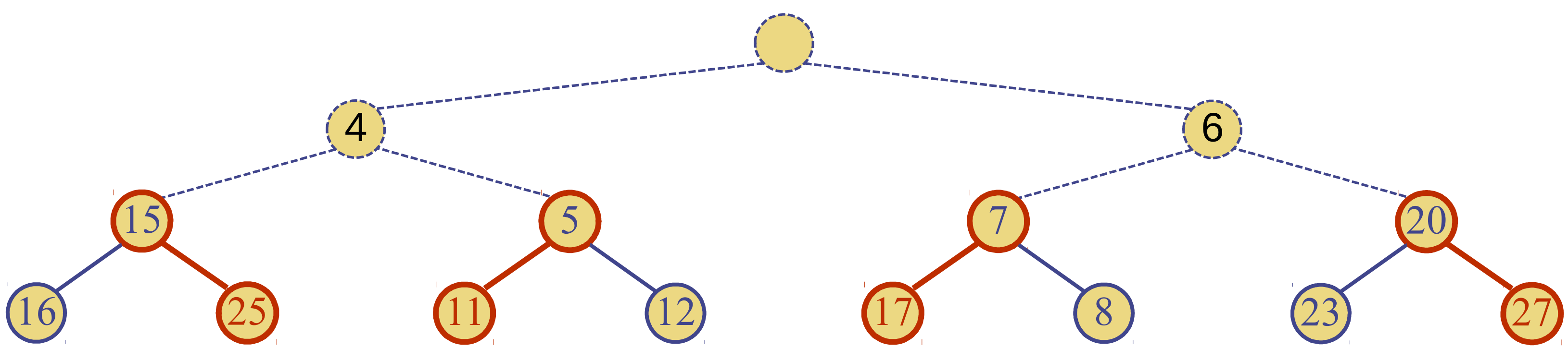
Esempio/1



14	5	8	15	4	6	20	16	25	11	12	17	7	23	27

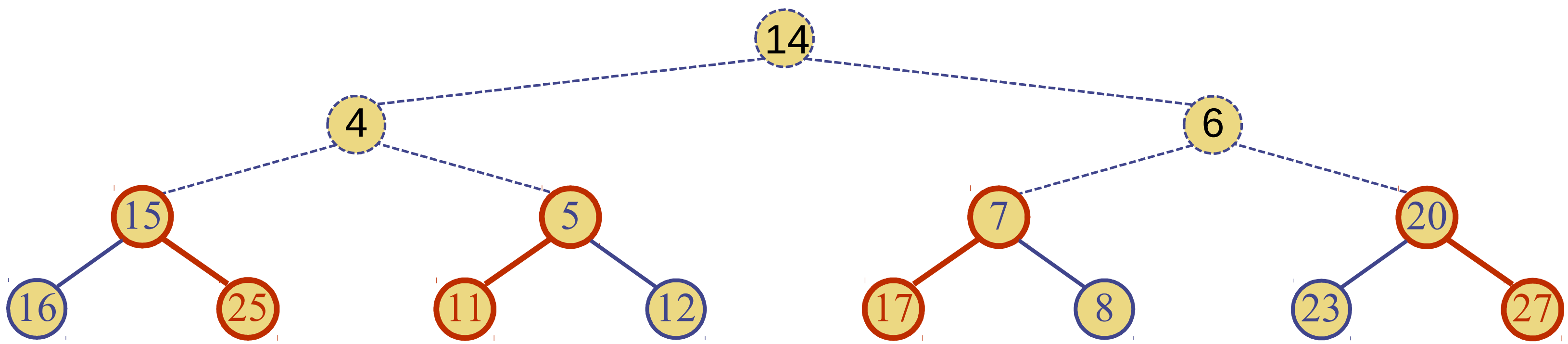
Esempio/2

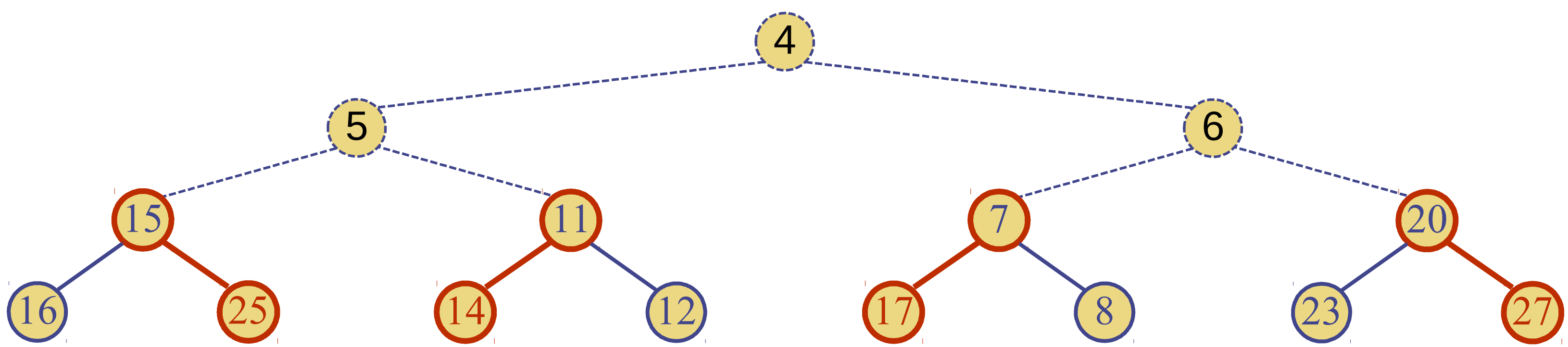




14	4	6	15	5	7	20	16	25	11	12	17	8	23	27

Esempio/3

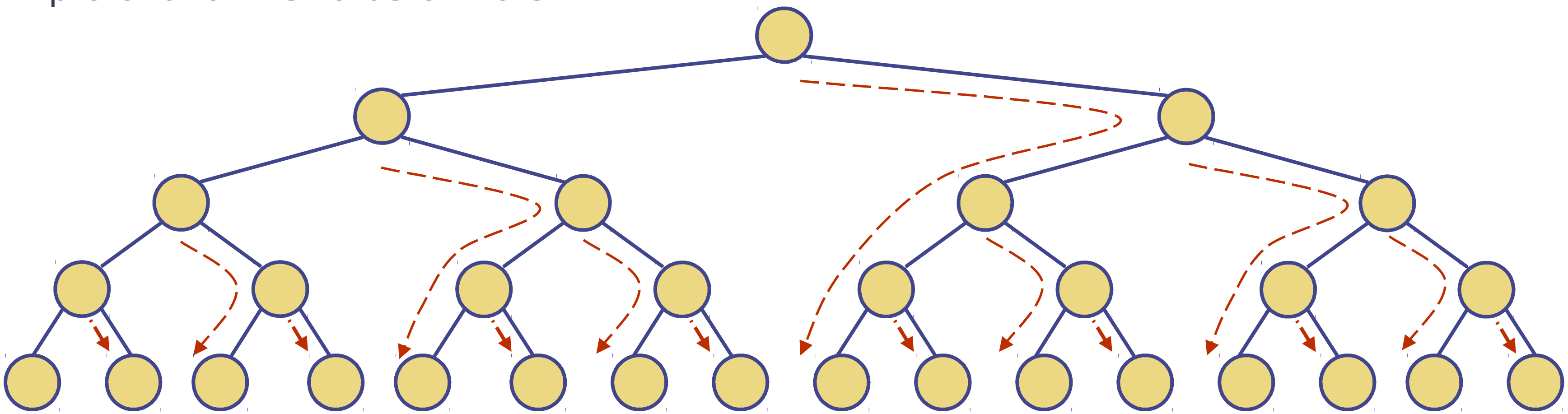




4	5	6	15	11	7	20	16	25	14	12	17	8	23	27

Analisi di heapify

- **Dato l'heap finale, consideriamo dei cammini disgiunti che descrivono, per ogni nodo, il numero di attraversamenti nel caso peggiore**
  - **Osservazione:** il costo è proporzionale alla somma del numero di attraversamenti complessivamente subiti dai nodi nelle diverse procedure downheap seguite a ogni fusione
- **I cammini coprono l'albero (O(n) archi)**
- **Quindi il costo complessivo è O(n)**
- **E' possibile un'analisi alternativa**
  - **Intuizione**: qual è il numero complessivo di attraversamenti subiti dai nodi che si trovano a profondità i nell'albero finale?

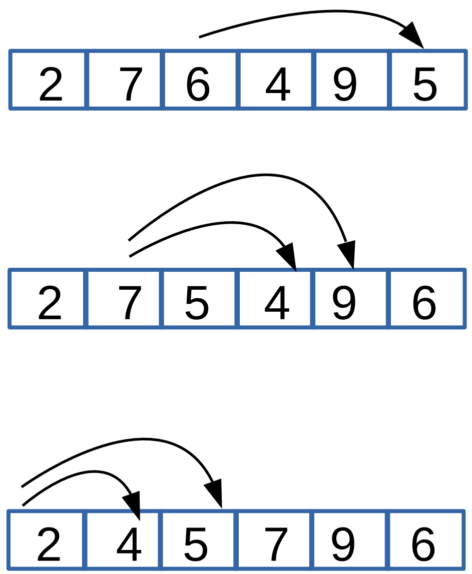


Algoritmo (in Java)

```
protected void heapify() {
  int startIndex = parent(size()-1);    // start at PARENT of last entry
  for (int j=startIndex; j >= 0; j--)   // loop until processing the root
    downheap(j);
}
```

Perché funziona

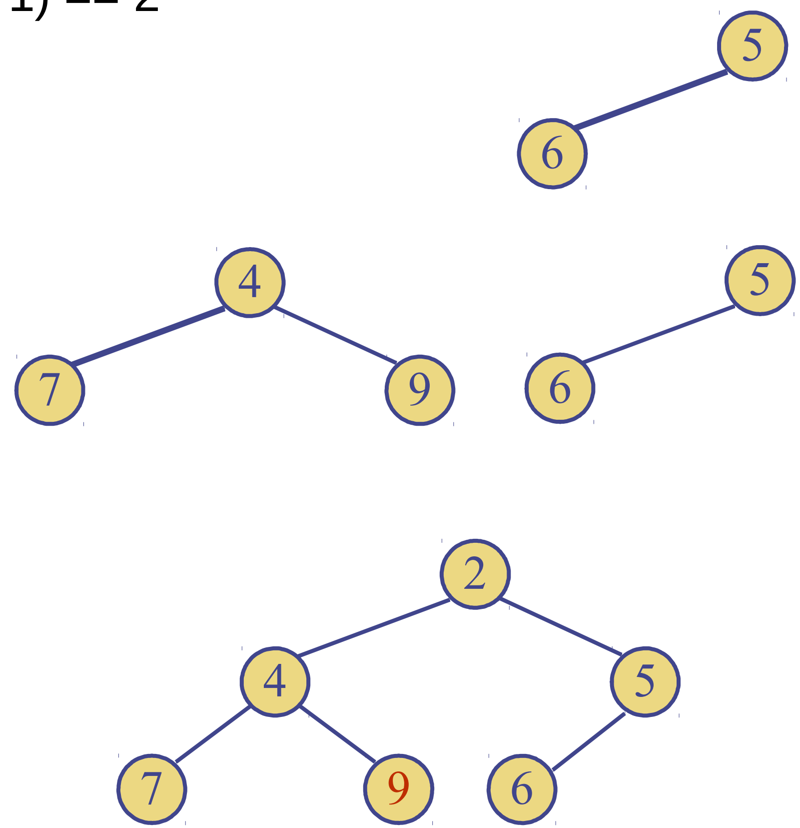
parent(size()-1) == 2



j == 2

j == 1

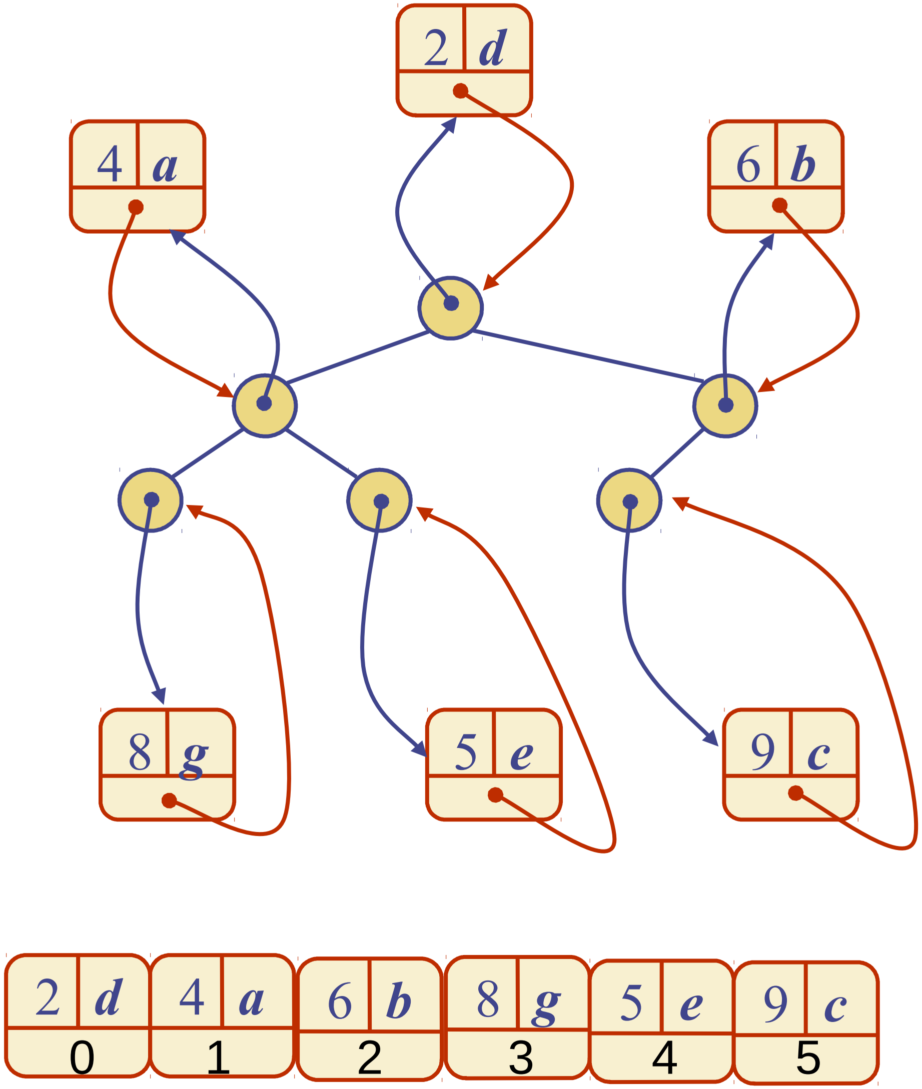
j == 0




```
protected void heapify() {
  int startIndex = parent(size()-1);
  for (int j=startIndex; j >= 0; j--)
    downheap(j);
}
```

Coda di priorità (heap) flessibile (Sez. 9.5 del libro)

- **Ogni entry contiene tre campi (invece di chiave e valore)**
  - Chiave
  - Valore
  - Posizione
- **Rappresentazione**
  - Albero binario perfettamente bilanciato
  - In pratica: array



Vantaggio: modifica chiave/valore

- **replace(e, k):**
  - e è un oggetto (entry) dell'heap
  - Sostituisce k al valore corrente della chiave della entry e
  - Ripristina l'heap
- **Complessità O(log n)**

```
replace(e, k) {
	i = e.position;
	e.key = k;
	if (heap.get(i).key < heap.parent(i).key)
		upheap(i);
	else
		downheap(i); // Potrebbe non essere
		             // necessario
}
```