▼ **8.0 - Heap**

▼ **8.1 - Heap binari**

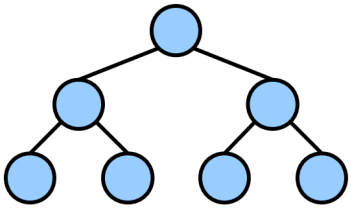
Alberi binari heap e array heap

Albero binario completo

Un **albero binario completo** è un albero binario che rispetta le seguenti proprietà:

- Tutte le foglie si trovano allo stesso livello h
- Tutti i nodi interni hanno grado (numero di figli diretti) 2

Osservazione: un albero binario completo con N nodi ha altezza $h \approx \log N$ e $N = 2^{h+1} - 1$.



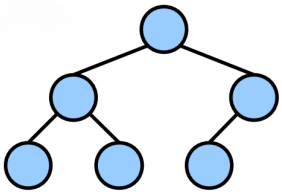
Albero binario completo.

Albero binario quasi completo

Un albero binario quasi completo è un albero binario che rispetta le seguenti proprietà:

- Albero completo almeno fino al livello $h - 1$.
- Tutti i nodi del livello h sono disposti a sinistra il più possibile.

Osservazione: tutti nodi interni di un albero quasi completo hanno grado 2, meno al più uno.



Albero binario quasi completo.

Albero binario heap

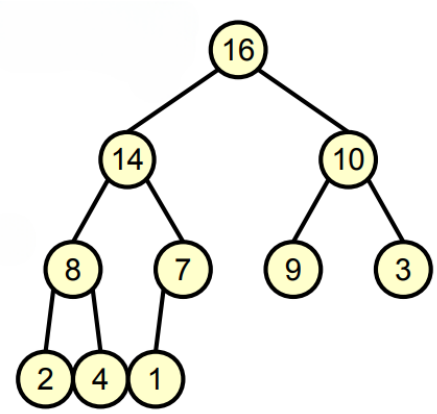
Un albero binario quasi completo è un **albero max-heap** se e solo se:

- Ad ogni nodo i viene associato un valore $A[i]$.
- $A[parent(i)] \geq A[i]$.

Un albero binario quasi completo è un **albero min-heap** se e solo se:

- Ad ogni nodo i viene associato un valore $A[i]$.
- $A[parent(i)] \leq A[i]$.

Nota: le operazione che vedremo su un albero max-heap sono simmetriche a quelle di un albero min-heap.

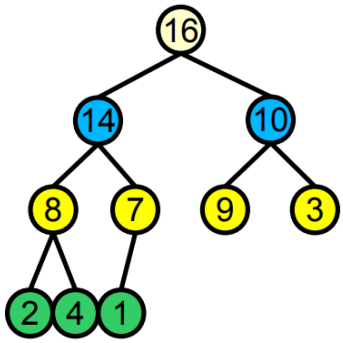


Albero max-heap.

Array heap

È possibile rappresentare un qualunque albero binario heap utilizzando un array A strutturato in questo modo:

- $A[1] =$ radice dell'albero.
- $left(i) = 2 \cdot i$.
- $right(i) = 2 \cdot i + 1$.
- $parent(i) = Math.floor(i/2)$.



Array heap.

Operazioni su array heap

Le **operazioni basilari** per array heap che vedremo sono le seguenti:

- **findMax**: ritorna il valore massimo contenuto in un max-heap.
- **heapify**: costruisce un max-heap a partire da un array privo di alcun ordine.
- **fixHeap**: rispristina la proprietà di max-heap in un array con solo una radice di indice i fuori posto.
- **deleteMax**: rimuove l'elemento con il valore massimo da un max-heap, mantenendo le proprietà di max-heap.

FindMax

Il valore massimo di un max-heap è sempre la **radice**, dunque il valore da ritornare è quello dell'elemento $A[1]$.

Heapify

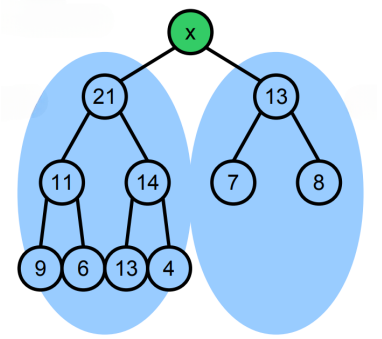
Per costruisce un max-heap a partire da un array privo di alcun ordine è possibile utilizzare una funzione di tipo ricorsivo, la quale prima effettua l'heapify sul figlio sinistro della radice e poi su quello destro, per poi richiamare la funzione fixHeap una volta che l'unico elemento che ha la possibilità di essere fuori posto è la radice. Per creare una funzione di questo tipo occorre utilizzare come parametri l'array heap A, l'indice dell'ultimo elemento dell'array n e l'indice della radice i.

Lo **pseudocodice** di tale funzione è il seguente:

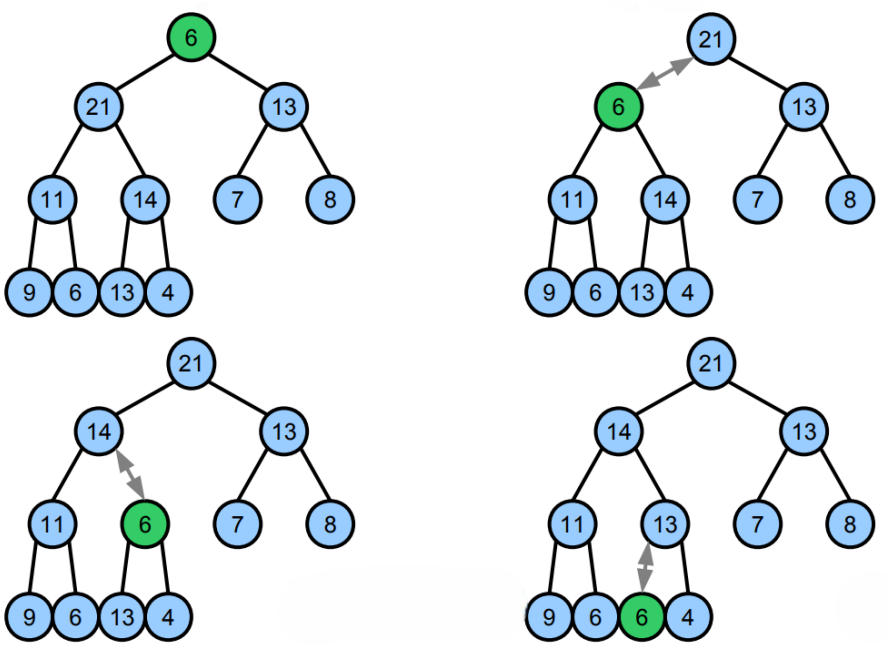
```
void heapify(Comparable A[], int n, int i) {
  if (i > n) return
  heapify(A, n, 2 * i) // heapify left child
  heapify(A, n, 2 * i + 1) // heapify right child
  fixHeap(A, n, i)
}
```

FixHeap

L'idea è quella di partire dalla radice di indice i e scambiarla con il figlio di valore massimo in maniera ricorsiva fino a quando le proprietà di max-heap non vengono ristabilite.



Albero con figlio destro e sinistro che rispettano il max-heap.



Esempio di fixHeap su albero con radice fuori posto.

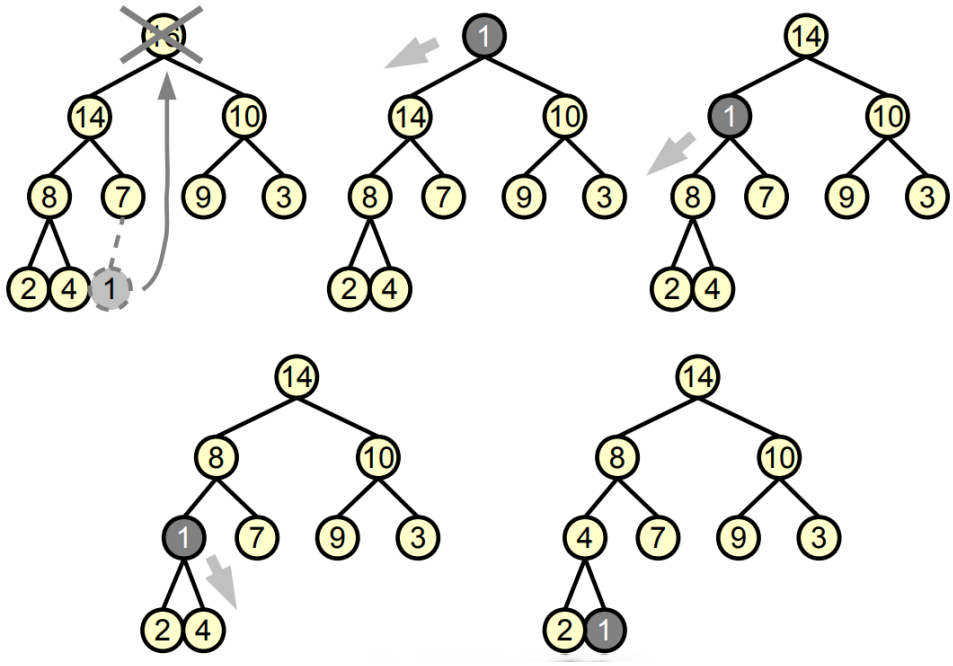
Lo **pseudocodice** di tale funzione è il seguente:

```
void fixHeap(Comparable S[], int c, int i) {
  int max = 2 * i
  if (2 * i > c) return
  if (2 * i + 1 <= c && S[2 * i].compareTo(S[2 * i + 1]) < 0) {
    // left child < right child
    max = 2 * i + 1
  }
  if (S[i].compareTo(S[max]) < 0) {
    // switch S[i] and S[max]
    Comparable temp = S[max]
    S[max] = S[i]
    S[i] = temp

    fixHeap(S, c, max)
  }
}
```

DeleteMax

L'idea è quella di sostituire il valore presente nella radice dell'albero con quello dell'elemento in ultima posizione nell'array heap, per poi richiamare la funzione fixHeap per ripristinare la proprietà di heap.



Esempio di deleteMax.

Costi computazionali

- **FindMax**: $O(1)$.
- **Heapify** (Master Theorem): $O(n)$.
- **FixHeap** (nel caso pessimo, il numero di scambi è uguale alla profondità dell'heap): $O(\log n)$.
- **DeleteMax** (nel caso pessimo, il numero di scambi è uguale alla profondità dell'heap): $O(\log n)$.

HeapSort

È possibile utilizzare la struttura dati heap e le sue operazioni basilari al fine di **ordinare un array**. Per fare ciò occorre seguire i seguenti passaggi:

1. Richiamare la funzione heapify sull'array da ordinare al fine di farlo diventare un array heap.
2. Estrarre il massimo dall'array heap tramite la funzione deleteMax ed inserirlo in ultima posizione dell'array.
3. Ripetere il punto 2 finchè l'heap non diventa vuoto.

Lo **pseudocodice** della funzione heapSort è il seguente:

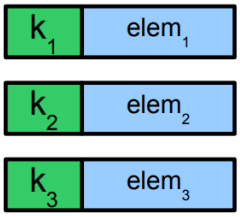
```
void heapSort(Comparable S[]) {
  heapify(S, S.length - 1, 1)
  for (int c = (S.length - 1); c > 0; c--) {
    Comparable k = findMax(S)
    deleteMax(S, c)
    S[c] = k
  }
}
```

Il costo computazionale di heapSort è dettato dalla funzione heapify, la quale ha un costo computazionale $O(n)$, e dal for, ciascuna delle cui iterazioni ha costo $O(\log c)$. Troviamo dunque che il costo computazionale è:

$$O(n) + O(\sum_{c=n}^{1} \log c) = O(n \log n)$$

▼ 8.2 - Code con priorità

> Una **coda con priorità** è una struttura dati che contiene un insieme di elementi ai quali sono associate delle chiavi che presentano una relazione d'ordine, ovvero sono ordinabili.



Coda con priorità.

Una tale struttura dati può essere utile ad esempio nella **gestione della banda di trasmissione**, in quanto nel routing dei pacchetti in rete è importante processare per primi i pacchetti con priorità più alta, dunque tali pacchetti possono essere inseriti in una coda con priorità.

Una coda con priorità inoltre può essere implementata in due modi: la prima implementazione corrisponde ad una semplice modifica della struttura dati heap, mentre la seconda consiste nell'utilizzare gli heap binomiali e gli heap di Fibonacci, i quali però non verranno trattati.

Operazioni basilari

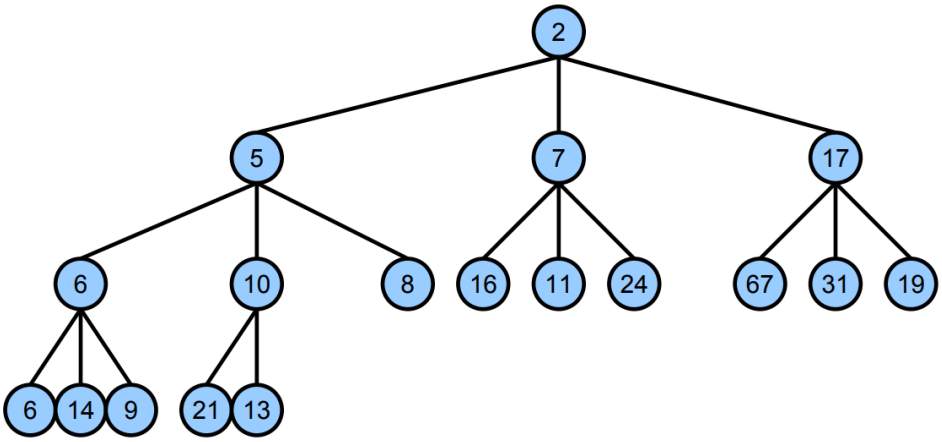
Le **operazioni basilari** per una coda con priorità sono le seguenti:

- **findMin()**: restituisce l'elemento associato alla chiave minima.
- **insert(e, k)**: inserisce un nuovo elemento e con associata la chiave k.
- **delete(e)**: rimuove l'elemento e.
- **deleteMin()**: rimuove l'elemento associato alla chiave minima.
- **increaseKey(e, c)**: incrementa la chiave dell'elemento e di un valore c.
- **decreaseKey(e, c)**: decrementa la chiave dell'elemento e di un valore c.

D-heap

> Un **d-heap** corrisponde ad un heap basato su un albero d-ario, dunque non solo binario, in cui ogni nodo diverso dalla radice ha chiave maggiore o uguale a quella del padre.

Osservazione: un d-heap con N nodi ha altezza $O(\log_d N)$ e $N > \sum_{i=0}^{h-1} d^i = \frac{d^h - 1}{d - 1} =$ numero nodi di un albero completo con altezza $h - 1$.



Esempio di d-heap con d = 3.

D-heap array

È possibile rappresentare un qualunque albero d-heap utilizzando un array A strutturato in questo modo:

- $A[1] =$ radice dell'albero.
- Il livello h inizia in $1 + \sum_{i=0}^{h-1} d^i = 1 + \frac{d^h - 1}{d - 1}$.
- Il livello h termina in $d^h + \sum_{i=0}^{h-1} d^i = d^h + \frac{d^h - 1}{d - 1}$.
- $firstChild(i) = ((i - 1) \cdot d) + 2$.
- $lastChild(i) = (i \cdot d) + 1$.
- $parent(i) = Math.floor(\frac{i-1}{d})$.

Proprietà fondamentale dei d-heap

> La radice di un d-heap contiene l'elemento con chiave minima.

Dimostrazione

Effettuiamo una dimostrazione per induzione sul numero di nodi n di un d-heap:

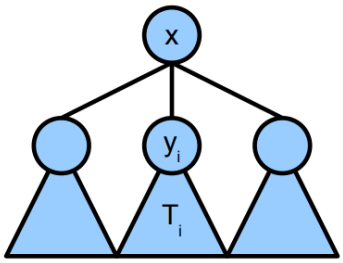
- Caso $n = 0$ o $n = 1$

  La proprietà vale.

- Caso $n > 1$

  Supponiamo per ipotesi induttiva che la proprietà vale per qualunque d-heap con al massimo $n - 1$ nodi.

  Consideriamo la seguente figura:



Sappiamo per ipotesi induttiva che y_i ha la chiave minima nell'albero T_i, il quale ha $n-1$ nodi.

Per la definizione di d-heap sappiamo inoltre che la chiave in $x \leq$ della chiave di ciascun figlio, dunque la chiave in x ha il valore minimo dell'intero heap.

Operazioni

Operazioni ausiliarie

- **MuoviAlto**: muove un elemento in alto nella coda con priorità fino a ristabilire l'heap.

  Lo **pseudocodice** di tale funzione è il seguente:

  ```
  void muoviAlto(v) {
    while(v != root(T) && chiave(v) < chiave(padre(v))) {
      scambia di posto v e padre(v) in T
      v = padre(v)
    }
  }
  ```

  Costo computazionale nel caso **peggiore** (v viene spostato da una foglia alla radice): $\log_d n$.

- **MuoviBasso**: muove un elemento in basso nella coda con priorità fino a ristabilire l'heap.

  Lo pseudocodice di tale funzione è il seguente:

  ```
  void muoviBasso(v) {
    while (true) {
      if (v non ha figli) return
      else {
        sia u il figlio di v con la minima chiave
        if (chiave(u) < chiave(v)) {
          scambia di posto u e v
          v := u;
        } else return
      }
    }
  }
  ```

  Costo computazionale nel caso **peggiore** (v viene spostato dalla radice a una foglia e per ogni spostamento vengono controllati tutti i d figli per trovare quella con la chiave minima): $d \cdot \log_d n$.

FindMin

In base alla proprietà fondamentale dei d-heap, la radice di una coda con priorità è l'elemento con chiave minima, dunque findMin deve ritornare la radice.

Il **costo computazionale**: $O(1)$.

Insert

Si inserisce il nuovo nodo come ultima foglia a destra della coda. In questo modo viene rispettata la proprietà di struttura della coda. Per rispettare anche la proprietà di ordine, occorre eseguire muoviAlto sul nuovo nodo.

Il **costo computazionale** nel caso **peggiore** (dettato dalla funzione muoviAlto): $O(\log_d n)$.

Delete

Viene sostituito il nodo da eliminare con il nodo presente nell'ultima foglia a destra della coda con priorità. A questo punto di esegue su tale nodo le operazioni muoviAlto e muoviBasso, una delle quali terminerà immediatamente.

Il **costo computazionale** nel caso **peggiore** (dettato dalla funzione muoviAlto o muoviBasso, a seconda di quale viene eseguita): $O(d \cdot \log_d n)$.

DeleteMin

Viene richiamata la funzione delete sulla radice dell'albero, la quale ha chiave minima per la proprietà fondamentale dei d-heap.

Il **costo computazione** nel caso **peggiore**: $O(d \cdot \log_d n)$.

IncreaseKey

Viene incrementata la chiave del nodo passato in input e successivamente viene richiamata la funzione muoviBasso su di esso.

Il **costo computazionale** nel caso **peggiore** (dettato dalla funzione muoviBasso): $O(d \cdot \log_d n)$.

DecreaseKey

Viene decrementata la chiave del nodo passato in input e successivamente viene richiamata la funzione muoviAlto su di esso.

Il **costo computazionale** nel caso **peggiore** (dettato dalla funzione muoviAlto): $O(\log_d n)$.