

Code con priorit 

Luca Tagliavini

April 8-12, 2021

Contents

0.1	Coda con priorit�	2
0.1.1	Operazioni disponibili	2
0.2	<i>d</i> -heap	3
0.2.1	Funzioni ausiliarie: <i>muovi_basso</i>	3
0.2.2	Funzioni ausiliarie: <i>muovi_alto</i>	3
0.2.3	Costi degli altri metodi della <i>d</i> -heap	4

0.1 Coda con priorit 

Struttura che mantiene il minimo (o massimo) in un insieme dinamico di chiavi su cui   definita una relazione d'ordine totale. Una coda con priorit    un insieme di n elementi di tipo *elem* cui sono associate chiavi.

- $k_1 \Rightarrow elem_1$
- $k_2 \Rightarrow elem_2$
- $k_3 \Rightarrow elem_3$
- ...

Un esempio puo' essere la sincronizzazione della banda dove c'  un continuo scambio di pacchetti tra client e server, e alcuni pacchetti hanno giustamente priorit  piu' alta e vengono dunque scambiati prima di altri (i.e. uno stream audio del presentatore della lezione vs un messaggio in chat). I pacchetti in ingresso possono dunque essere mantenuti in una coda con priorit  per gestire prima quelli con priorit  maggiore. Una cosa analoga puo' essere fatta lato server dove i pacchetti vengono distribuiti ai vari client.

0.1.1 Operazioni disponibili

- *find_min()* $\rightarrow elem$: restituisce l'elemento associato alla chiave minima.
- *insert(elem e, chiave k)* $\rightarrow void$: inserisce un nuovo elemento e con associata la chiave k .
- *delete(elem e)* $\rightarrow void$: rimuove un elemento dalla coda al quale assumiamo di avere si ha accesso diretto.
- *delete_min()* $\rightarrow void$: rimuove l'elemento associato alla chiave minima.
- *increase_key(elem e, chiave k)* $\rightarrow void$: aumenta la priorit  dell'elemento e della quantit  k .
- *decrease_key(elem e, chiave k)* $\rightarrow void$: decrementa la chiave dell'elemento e e della quantit  d .

La struttura ammette metodi e funzionalit  molto simili quindi ci ispireremo fortemente alla sua costruzione. Tuttavia applicheremo alcuni cambiamenti:

- Non avremo piu' un albero binario, ma ogni nodo potra' avere un numero arbitrario di figli fino ad un numero massimo d . Chiameremo questa struttura d -heap.
- Implementeremo le operazioni *delete*, *increase_key*, *decrease_key* che non avevamo implementato ai tempi dell'heap sort poiche' non sfruttate.

NOTA: *increase_key* torner  utile quando studieremo l'algoritmo di Dijkstra.

0.2 *d*-heap

Estende il concetto di min/max-heap già visto, con la differenza che un heap classico è binario mentre invece un *d*-heap sarà *d*-ario. Ha le seguenti proprietà:

- Un *d*-heap di *h* altezza è un albero perfetto almeno fino alla profondità $h - 1$. Le foglie (nel livello *h* vengono accatastate a sinistra).
- Ogni nodo *v* contiene una *chiave*(*v*) e un *elem*(*v*). L'insieme delle chiavi è un insieme ordinato (i.e. 1, 2, 3, ..., 10)
- Ogni figlio (diverso dalla radice) ha la chiave *non inferiore* (\geq) a quella del padre.
- Un *d*-heap con *n* nodi ha profondità $O(\log_d n)$.
Lo si prova con le serie geometriche come visto per le normali heap, vdi slide.

Anche i *d*-heap possono essere memorizzati tramite un array (index 1 - *n*), il cui processo è descritto nelle slide e ci offre un accesso diretto agli elementi in posizioni arbitrarie in modo da abbassare il costo di alcuni metodi (altrimenti logaritmici) a costanti.

0.2.1 Funzioni ausiliarie: *muovi_basso*

```
algorithm muovi_basso(Node v)
  while true do
    if has_no_child(v) then
      break
    else
      u := min_child(v)
      if chiave(u) < chiave(v) then
        swap(u, v)
        v = u
      else
        break
    endif
  endwhile
endalgorithm
```

Costo: $O(dh)$

Sposta il nodo *v* in basso nella heap (tra i suoi figli) fino ad una posizione in cui rende l'albero un *d*-heap, ossia posiziona il nodo *v* scambiandolo con un figlio in modo che esso sia il minore tra il sottoalbero preso in considerazione.

0.2.2 Funzioni ausiliarie: *muovi_alto*

```

algorithm muovi_alto(Node v)
  while v != root(T) and chiave(v) < chiave(parent(v)) do
    swap(v, parent(v))
    v = parent(v)
  endwhile
endalgorithm

```

Costo: $O(d)$

Sposta il nodo v in alto fino a quando o esso non e' la radice (ci fermiamo perche' la chiamata *parent* fallirebbe) o **ha una chiave \geq di quella del padre**

0.2.3 Costi degli altri metodi della d -heap

Si noti che $d = \log_d n$ per le proprieta' della d -heap.

- $find_min() \Rightarrow O(1)$ in quanto il minimo e' il nodo radice
- $insert(elem\ e, chiave\ k) \Rightarrow O(\log_d n)$ in quanto aggiunge l'elemento il piu' possibile a destra nell'ultimo livello e necessita di una chiamata di *muovi_alto*
- $delete(elem\ e) \Rightarrow O(d \log_d n)$ in quanto si scambia la foglia piu' a destra dell'ultimo livello con il nodo scambiato e poi si esegue *muovi_alto* o *muovi_basso* in base alla situazione, tra i quali il peggiore ha costo $O(d \log_d n)$
- $delete_min() \Rightarrow O(d \log_d n)$ in quanto chiama *delete*
- $increase_key(elem\ e, chiave\ k) \Rightarrow O(\log_d n)$ in quanto aumenta il valore e chiama *muovi_alto* per riordinare l'albero dopo l'incremento
- $decrease_key(elem\ e, chiave\ k) \Rightarrow O(d \log_d n)$ in quanto chiama *muovi_basso* per ribilanciare la d -heap