Heap e code di priorità

Violetta Lonati

Università degli studi di Milano
Dipartimento di Informatica

Laboratorio di algoritmi e strutture dati
Corso di laurea in Informatica

Problema

Contesto

- S è un insieme dinamico di n elementi, ciascuno dei quali è dotato di una chiave o valore di priorità; in genere: minore è la chiave, massimo è il suo valore di priorità.
- Le chiavi sono ordinate (totalmente), ovvero per ogni coppia di chiavi k_1 e k_2 si ha $k_1 \leq k_2$ oppure $k_2 \leq k_1$.
- Vogliamo poter eseguire efficientemente le seguenti operazioni:
  - inserire elementi;
  - scegliere l'elemento di S con massima priorità (valore minimo);
  - cancellare l'elemento di S con massima priorità.

Esempio di applicazione

Scheduling online di processi (ad opera del sistema operativo): i processi vanno eseguiti in base ad un certo valore di priorità, ma le richieste non arrivano necessariamente in questo ordine.

Ordinamento tramite code di priorità

Avendo a disposizione una coda di priorità, è possibile effettuare questo algoritmo di ordinamento:

```
crea una nuova coda di priorità Q
inserisci in Q un elemento di S alla volta
finchè Q non è vuota
  estrai il minimo m da Q
  stampa m
```

Se le operazioni di inserimento e estrazione del minimo si possono fare in tempo $O(\log n)$, allora otterremmo un algoritmo di ordinamento ottimale, ovvero di costo $O(n \log n)$, infatti:

- per ogni elemento di S, l'inserimento in coda costa $O(\log n)$, quindi l'inserimento degli n elementi costa $O(n \log n)$;
- l'estrazione del minimo costa $O(\log n)$ quindi il ciclo finale costa $O(n \log n)$.

Obiettivo: implementare queste operazioni con costo $O(\log n)$!!

Implementazioni naïf (1)

Usando una lista con un puntatore all'elemento minimo:

- l'inserimento in testa ha costo $O(1)$;
- la ricerca del minimo ha costo $O(1)$;
- per estrarre il minimo devo aggiornare il puntatore, quindi devo scorrere la lista e il costo diventa $O(n)$.

$\Rightarrow$ Soluzione non ottimale

Implementazioni naïf (2)

Usando una struttura ordinata:

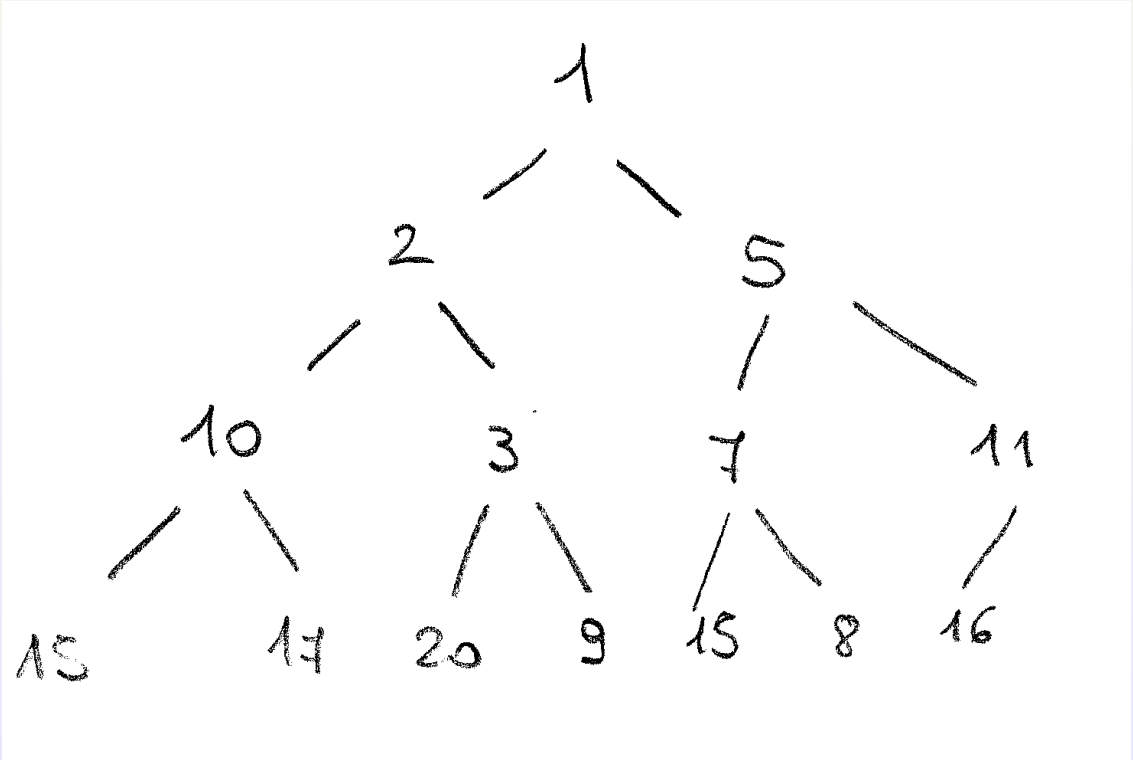
- la ricerca del minimo ha costo $O(1)$;
- l'estrazione del minimo ha costo $O(1)$;
- l'inserimento ha costo $O(n)$:
  - se uso un array: con una ricerca dicotomica trovo la posizione in cui inserire con costo $O(\log n)$ ma poi devo spostare tutti gli elementi più grandi e questo nel caso peggiore ha costo $O(n)$;
  - se uso una lista: l'inserimento ha costo $O(1)$, ma la ricerca della posizione in cui effettuarlo ha costo $O(n)$ (devo scorrerre nel caso peggiore tutta la lista).

$\Rightarrow$ Soluzione non ottimale

Struttura dati Heap

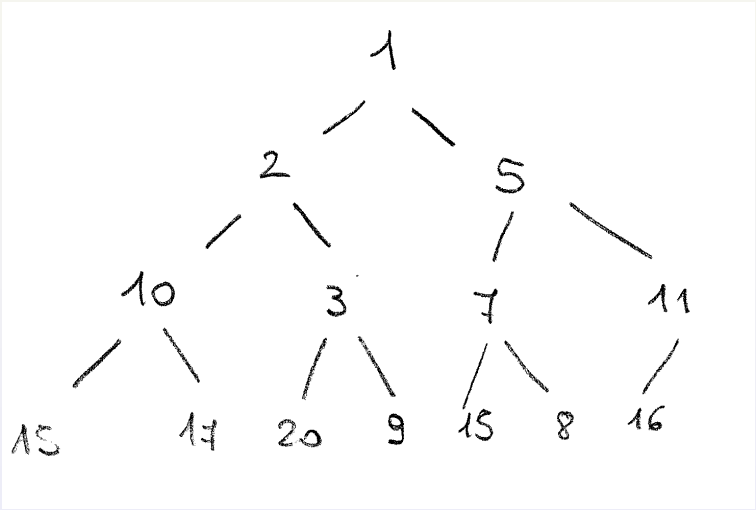
Uno heap è un albero binario completo (bilanciato) dove le chiavi rispettano questa proprietà: la chiave di un nodo è sempre minore della chiave dei sui figli.

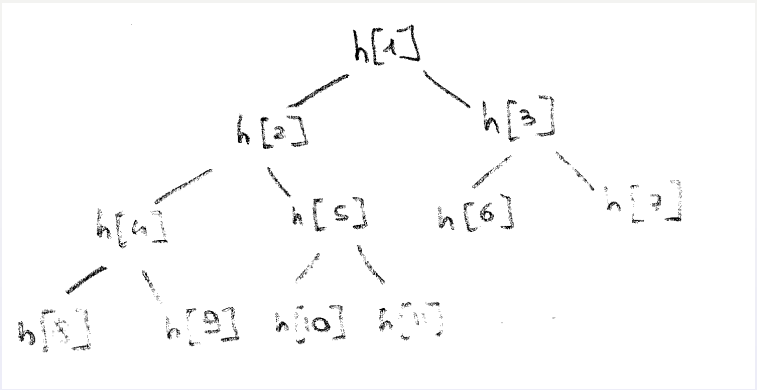
per ogni nodo i : $key(father(i)) \leq key(i)$



Rappresentazione di uno heap

Uno heap può essere rappresentato in memoria come un albero binario (nodi con puntatori ai figli destro e sinistro). Essendo però un albero completo (tutti i livelli sono riempiti tranne al più l'ultimo), è comodo rappresentare uno heap semplicemente con un array.





$$h = \{IGNORE, 1, 2, 5, 10, 3, 7, 11, 15, 17, 20, 9, 15, 8, 16\}$$

Rappresentazione di uno heap - continua

NB: per coerenza con le dispense e il libro riempiamo l'array a partire dalla posizione 1 (lasciando inutilizzata la posizione 0).
Formalmente: detto n il numero di elementi contenuti nell'array, abbiamo:

$$\begin{array}{lll} \forall i \geq 1 & \mathit{left}(i) = 2i & \text{se } 2i \leq n \\ \forall i \geq 1 & \mathit{right}(i) = 2i + 1 & \text{se } 2i + 1 \leq n \\ \forall i \geq 2 & \mathit{father}(i) = \lfloor i/2 \rfloor & \text{se } 1 < i \leq n \end{array}$$

$$h = \{\mathit{IGNORE}, 1, 2, 5, 10, 3, 7, 11, 15, 17, 20, 9, 15, 8, 16\}$$

Ricerca del minimo

La ricerca del minimo è immediata: si trova nella radice!
Costo $O(1)$

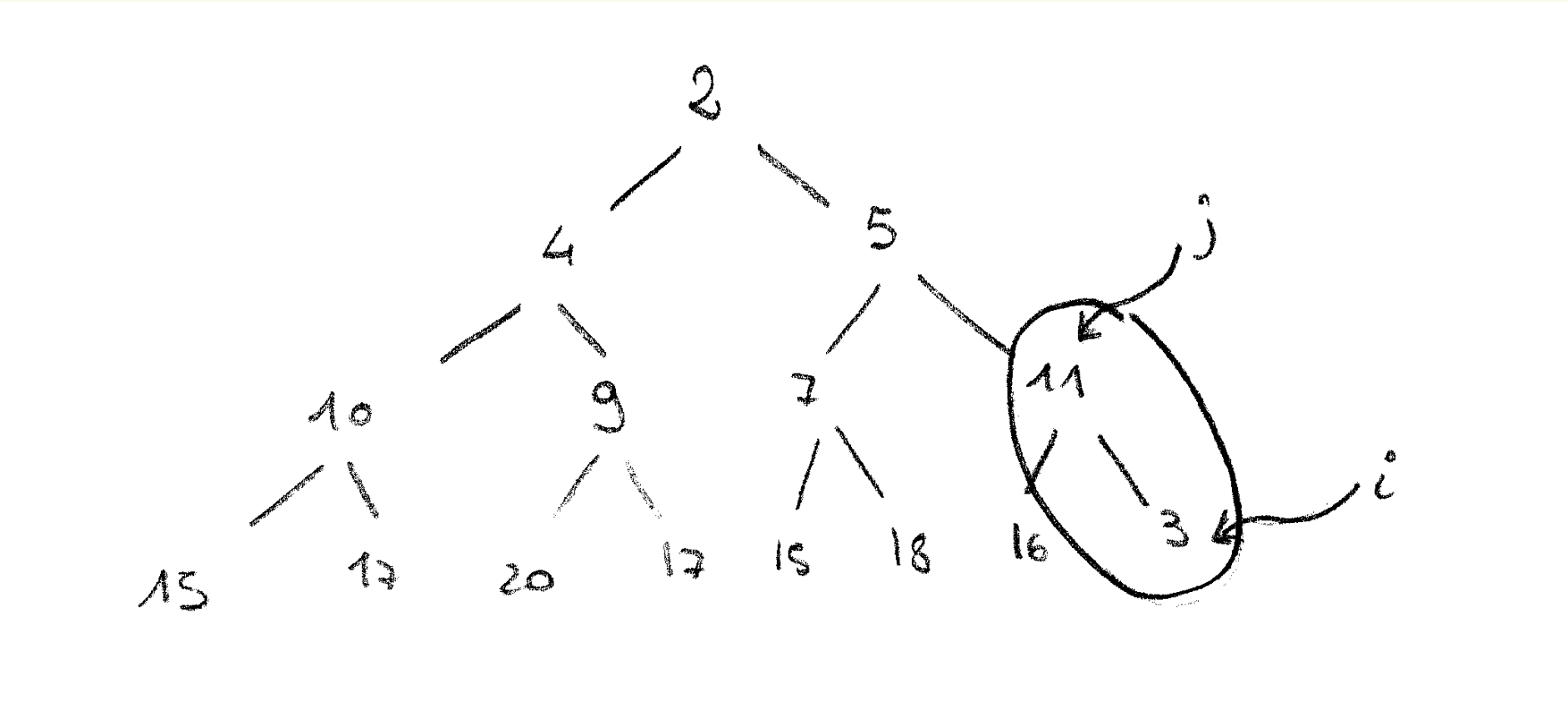
Inserimento

- Chiamiamo `h` il vettore che rappresenta lo heap e sia `n-1` la sua lunghezza (ovvero il numero di elementi che contiene attualmente).
- Se inserisco il nuovo elemento nella posizione `n` di `h`, la proprietà dello heap potrebbe non essere più valida, perchè in posizione `n` potrei avere una chiave troppo piccola.
- Aggiustiamo lo heap a partire dalla posizione `n` risalendo verso l'alto, usando la seguente funzione ricorsiva:

```
void heapify_up( Heap h, int i ) {
  if ( i > 1 ) {
    int j = father(i);
    if ( cmp( key( h[i] ), key( h[j] ) ) < 0 ) {
      swap( h, i, j );
      heapify_up( h, j );
    }
  }
}
```

Chiaramente la funzione `father(i)` deve restituire l'elemento di `h` in posizione `i/2`.

Inserimento - esempio



Inserimento - correttezza e complessità

Ad ogni esecuzione di `heapify_up(i)`, riparo il sottoalbero di radice `i` e risalgo, promuovendo gli elementi di chiave più bassa.

Correttezza

Se parto da un albero che è quasi uno heap tranne che per il fatto che la chiave di `i` è troppo piccola, allora la chiamata di `heapify_up(h,i)` consente di ottenere uno heap corretto.

Complessità

$O(\log n)$: al più effettuo tanti confronti/scambi quanta è l'altezza del nodo `i` nell'albero.

Cancellazione

In genere, una coda di priorità richiede di cancellare solo l'elemento di chiave minima. Qui vediamo la cancellazione in generale.
Sia `n` la lunghezza dello heap `h`; per cancellare l'elemento di posizione `i`:

- spostiamo `h(n)` in `h(i)` e decrementiamo la lunghezza `n`;
- la proprietà dello heap non vale più, poichè in posizione `i` potrei avere una chiave troppo grande;
  1. se `key(i) < key(father(i))`, allora aggiusto lo heap verso l'alto chiamando `heapify_up(h,i)`
  2. se `key(i) > key(left(i))` oppure `key(i) > key(right(i))`, allora aggiusto lo heap verso il basso con la funzione ricorsiva `heapify_down(h,i,n)`.

Cancellazione

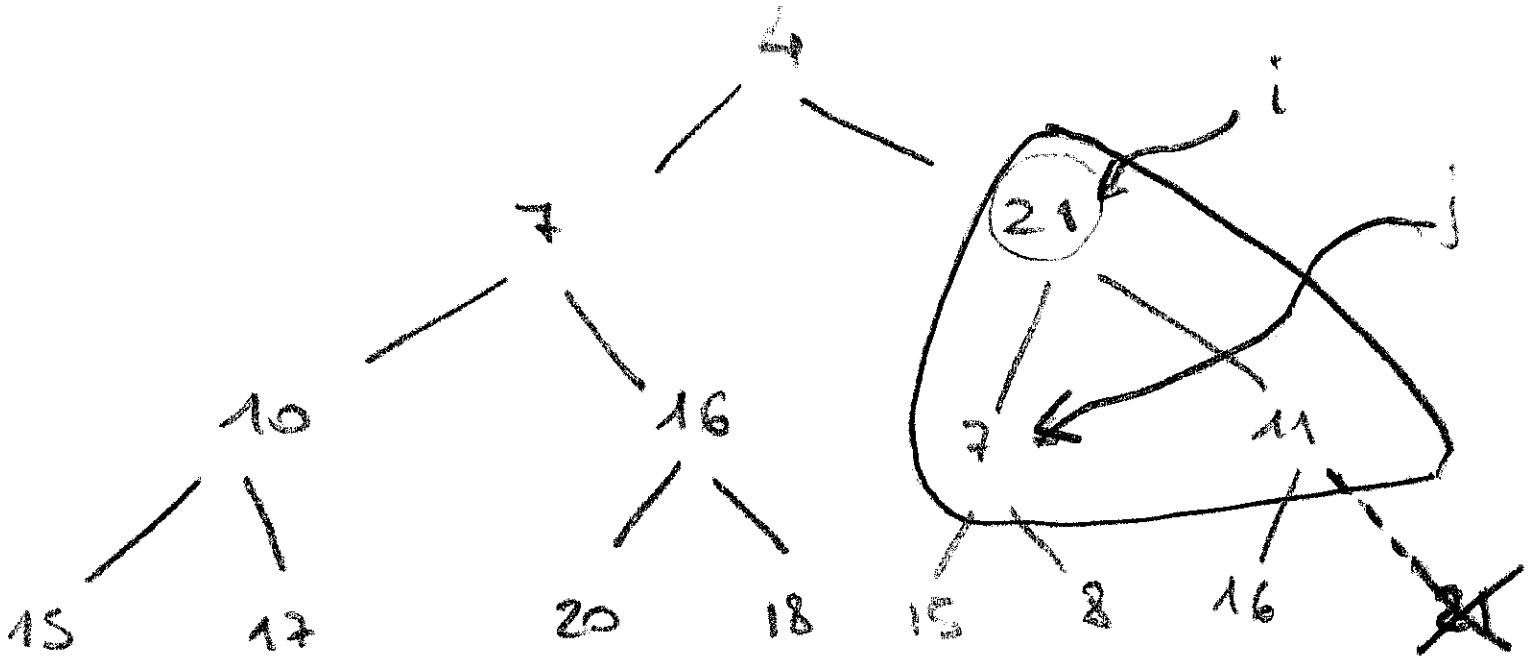
```
void heapify_down( Heap h, int i, int n ) {
  if ( 2*i <= n ) { /* i ha almeno un figlio */

    int j;  /* indice del figlio di i con chiave minore  */
    if ( 2*i == n ) /* i ha solo il figlio sinistro */
      j = 2*i;
    else /* i ha due figli */
      j = cmp( key( h[2*i] ), key( h[2*i+1] ) ) < 0
                        ? 2*i : 2*i + 1;

    if ( cmp( key( h[j] ), key( h[i] ) ) < 0 ) {
      swap( h, i, j );
      heapify_down( h, j, n );
    }
  }
}
```

NB: in questo caso serve sapere quanti sono gli elementi contenuti nello heap, quindi serve l'argomento `n`.

Cancellazione - esempio



Cancellazione - correttezza e complessità

Correttezza

Se parto da un albero che è quasi uno heap tranne che per il fatto che la chiave di `i` è troppo grande, allora la chiamata di `heapify_down(h,i,n)` consente di ottenere uno heap corretto.

Complessità

$O(\log n)$: al più effettuo tanti confronti/scambi quanto è lungo il cammino dal nodo `i` fino ad una foglia.