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:

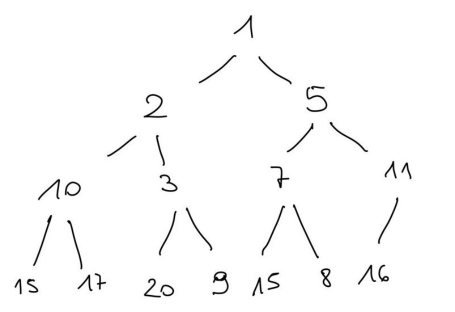
- la ricerca del minimo ha costo $O(1)$;
- l'estrazione del minimo ha costo $O(1)$;
- l'inserimento ha costo $O(n)$:
  - se uso un array: con una ricerca dicotomica trovo la posizione in cui inserire con costo $O(\log n)$ ma poi devo spostare tutti gli elementi più grandi e questo nel caso peggiore ha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

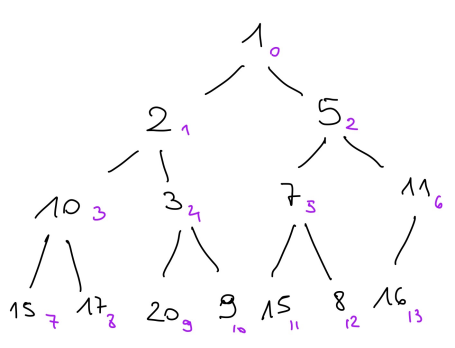
Uno heap è un albero binario completo (bilanciato) dove le chiavi rispettano questa proprietà: la chiave di un nodo è sempre minore della chiave dei sui figli.

$$\text{per ogni nodo } i: \qquad \textit{priority}(\textit{parent}(i)) \leq \textit{priority}(i)$$



Rappresentazione di uno heap

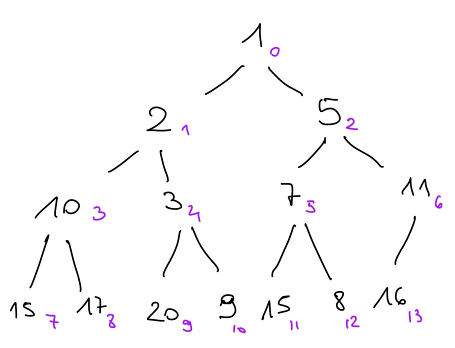
Uno heap può essere rappresentato in memoria come un albero binario (nodi con puntatori ai figli destro e sinistro). Essendo però un albero completo (tutti i livelli sono riempiti tranne al più l'ultimo), è comodo rappresentare uno heap semplicemente con un array.



$$h = \{1, 2, 5, 10, 3, 7, 11, 15, 17, 20, 9, 15, 8, 16\} \qquad n = 14$$

Rappresentazione di uno heap - continua

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



Formalmente: detto n il numero di elementi contenuti nell'array, abbiamo:

$$\begin{array}{ll} \mathit{left}(i) = 2i + 1 & \forall i \geq 0 \text{ con } 2i + 1 < n \\ \mathit{right}(i) = 2i + 2 & \forall i \geq 0 \text{ con } 2i + 2 < n \\ \mathit{parent}(j) = (j - 1)/2 & \forall\ 1 \leq j < n \end{array}$$

Ricerca del minimo

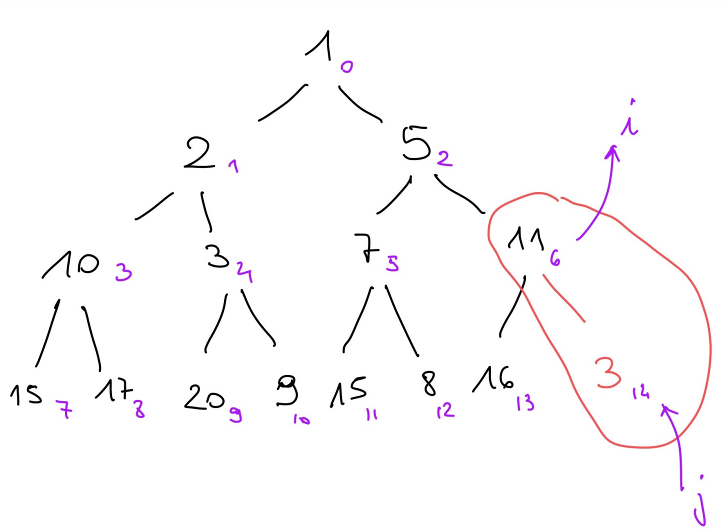
La ricerca del minimo è immediata: si trova nella radice!
Costo $O(1)$

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:

```
func heapify_up(h heap, int j) {
  for {
    i := (j - 1) / 2 // parent
    if i == j || !less(j, i) {
      break
    }
    swap(h, i, j)
    j = i
  }
}
```

Inserimento - esempio



Inserimento - correttezza e complessità

Durante l'esecuzione di `heapify_up(i)`, si ripara innanzitutto il sottoalbero di radice `j` e poi si risale, promuovendo gli elementi con chiave più bassa.

Correttezza

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

Estrazione del minimo

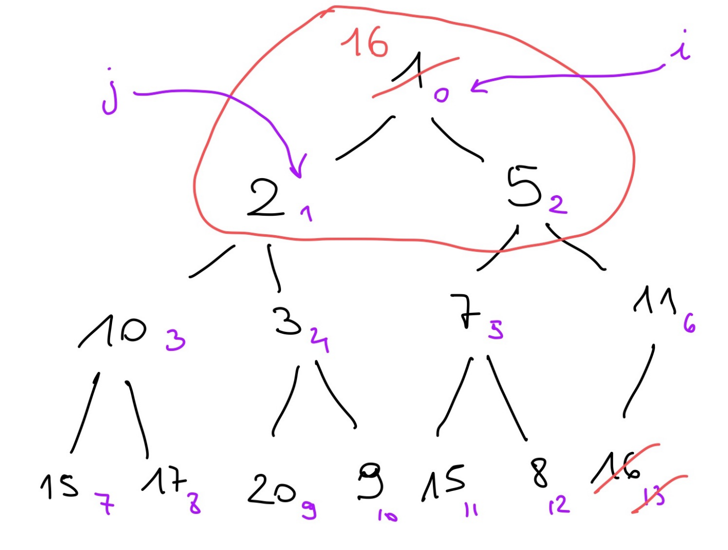
Sia `n` la lunghezza dello heap `h`; per cancellare l'elemento minimo

- spostiamo `h(n)` nella radice (indice 0) e decrementiamo la lunghezza dello heap `n`;
- a questo punto, la proprietà dello heap potrebbe non valere più nella radice:
  se `priority(0) > priority(1)` oppure `priority(0) > priority(2)`, allora aggiusto lo heap verso il basso invocando `heapify_down(h,0,n)`.

Heapify_down

```
func down(h heap, i int) {
  for {
    j := 2*i + 1  // left child
    if j >= len(h)  {
      break
    }
    j2 := j + 1
    if  j2 < len(h) && less(j2, j) {
      j = j2 // right child
    }
    if !less(j, i) {
      break
    }
    swap(h, i, j)
    i = j
  }
}
```

Estrazione del minimo - esempio



Cancellazione

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

- spostiamo `h(n-1)` in posizione `i` e decrementiamo la lunghezza `n`;
- a questo punto, attorno alla posizione `i`, la proprietà dello heap potrebbe non valere più:
  1. se `priority(i) < priority(parent(i))`, allora aggiusto lo heap verso l'alto chiamando `heapify_up(h,i)`
  2. se `priority(i) > priority(left(i))` oppure `priority(i) > priority(right(i))`, allora aggiusto lo heap verso il basso con la funzione `heapify_down(h,i)`.

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)` 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.

Container heap in Go

Documentazione:

`https://pkg.go.dev/container/heap`

Codice:

`https://cs.opensource.google/go/go/+/go1.19.3:src/container/heap/heap.go`