Azzolini Riccardo 2019-04-10

Heap e heapsort

1 Heap

Siano U un insieme di elementi e P un insieme totalmente ordinato di **priorità**. Uno **heap** (*orientato al massimo*), o **coda con priorità**, di elementi di tipo U e priorità P è un elemento di $(U \times P)^\star$ che supporta le operazioni di

inserimento: $\text{inserisci} : (U \times P)^\star \times U \times P \to (U \times P)^\star$
inserisci(H, e, p) inserisce nell'heap H l'elemento e, con priorità P;

rimozione: $\text{cancella} : (U \times P)^\star \to (U \times P)^\star$
cancella(H) elimina dall'heap H l'elemento con priorità maggiore;

lettura: $\text{leggi} : (U \times P)^\star \to U$
leggi(H) restituisce l'elemento con priorità maggiore presente in H.

Osservazioni:

- Se H contiene più elementi con la stessa priorità, non è specificato (e può quindi dipendere dall'implementazione) quale di questi sarà letto/rimosso per primo.
- Una semplice implementazione di uno heap può essere realizzata mediante una lista contenente gli elementi e le rispettive priorità, ma così facendo almeno una delle operazioni ha per forza costo lineare:
  - se la lista è mantenuta in ordine di priorità, lettura e rimozione hanno costo costante, $O(1)$, ma l'inserimento richiede tempo $O(n)$;
  - se, invece, la lista non è ordinata, l'inserimento si può effettuare sempre in testa, $O(1)$, ma in compenso la lettura e la rimozione richiedono tempo $O(n)$.

2 Vettore heap-ordinato

Un vettore A di lunghezza n si dice **heap-ordinato** se

$$
\begin{aligned}
\text{priority}(A[i]) &\geq \text{priority}(A[2i]) & \forall i,\ 1 \leq i < \frac{n}{2} \\
\text{priority}(A[i]) &\geq \text{priority}(A[2i+1]) & \forall i,\ 1 \leq i < \frac{n}{2} \\
\text{priority}\left(A\left[\frac{n}{2}\right]\right) &\geq \text{priority}(A[n]) & \text{se } n \text{ pari}
\end{aligned}
$$

Un vettore heap-ordinato corrisponde univocamente a un albero binario completo nel quale ogni nodo interno ha priorità maggiore o uguale a quella dei figli:

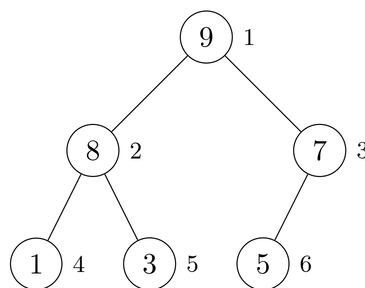
- $A[1]$ è la radice, con priorità massima.
- $\forall i,\ 1 < i \leq n$, $A[i]$ ha padre $A\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$ (quindi $A[j]$ ha figli $A[2j]$ e $A[2j+1]$, se esistono).

In pratica, gli elementi sono disposti nel vettore secondo l'ordine di attraversamento per livelli dell'albero.

I vettori heap-ordinati consentono l'implementazione degli heap in modo efficiente.

2.1 Esempio

1	2	3	4	5	6
9	8	7	1	3	5



3 Costruzione top-down e inserimento

La **costruzione top-down** di uno heap è basata sulla soluzione al seguente problema:

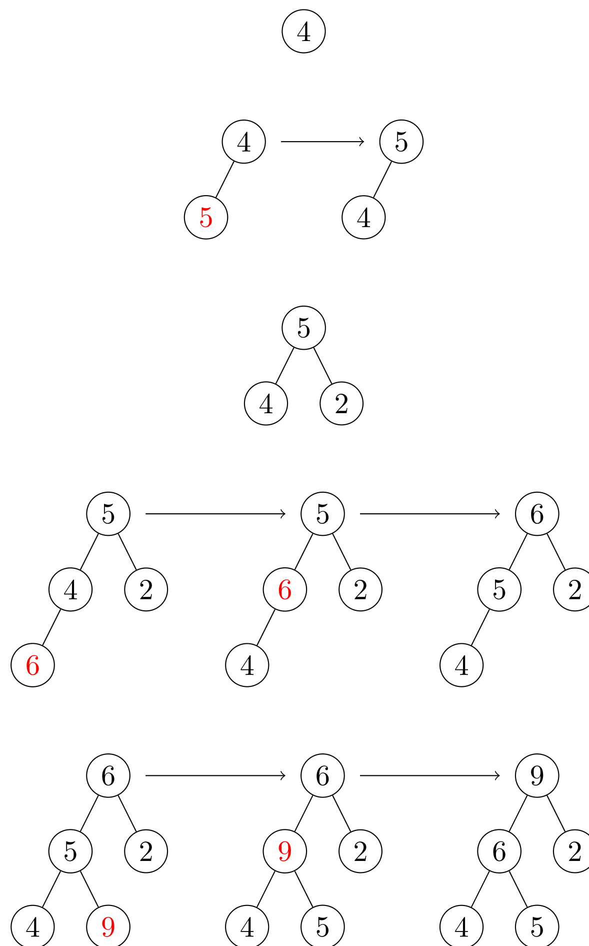
- *Problema*: Supponendo che i primi $i-1$ elementi di un vettore formino uno heap, considerare anche l'i-esimo e ripristinare lo heap di i elementi.

- *Soluzione*: Si risale il percorso dall'i-esimo elemento alla radice, facendo scorrere verso il basso gli elementi minori incontrati, fino a trovare un elemento maggiore.

Lo stesso procedimento di risalita si utilizza per inserire un nuovo elemento in uno heap già esistente.

3.1 Esempio

Si vuole costruire uno heap a partire dal vettore $(4, 5, 2, 6, 9)$.



Il vettore heap-ordinato risultante è $(9, 6, 2, 4, 5)$.

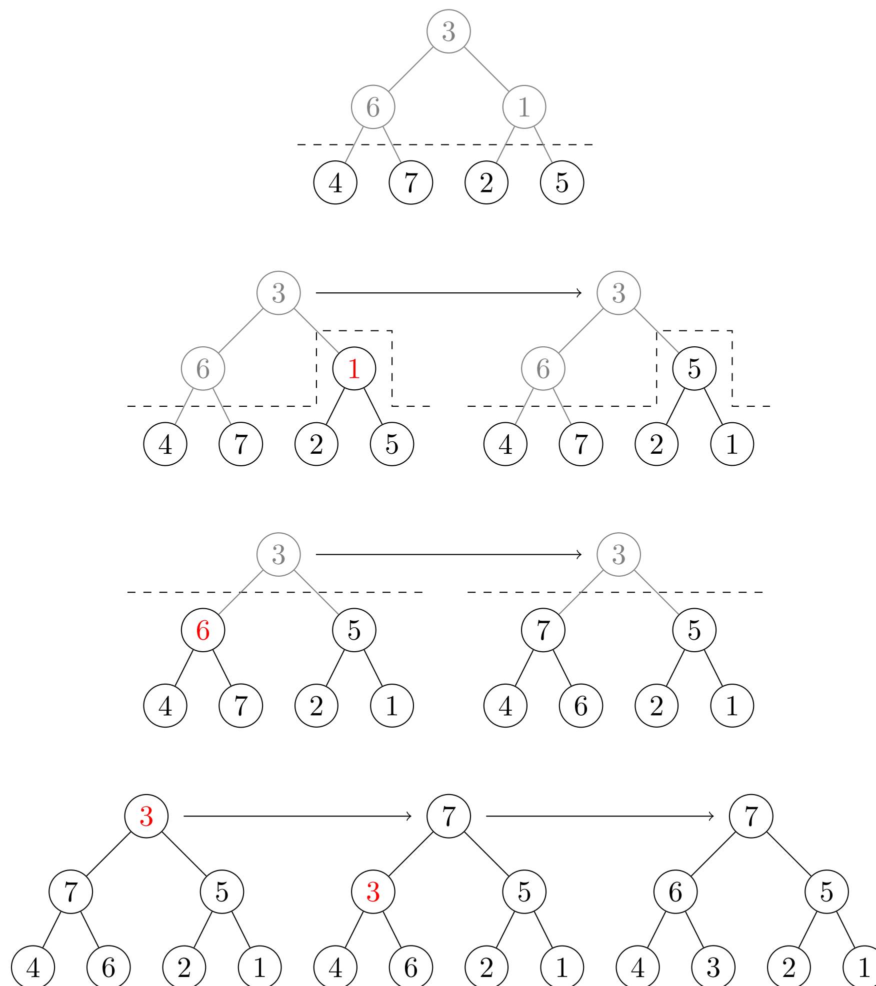
4 Costruzione bottom-up

La **costruzione bottom-up** sfrutta il fatto che all'inizio, la porzione di vettore corrispondente alle foglie è già heap-ordinata.

Per completare lo heap, si aggiunge un nodo interno alla volta, iniziando dall'ultimo (che si trova in posizione $\frac{n}{2}$). Quando ciascun nodo viene aggiunto, i suoi sottoalberi sono già heap, quindi per ripristinare lo heap è sufficiente far "sprofondare" il nuovo elemento, scambiandolo a ogni passo con il maggiore dei figli e fermandosi quando ha solo figli minori o diventa una foglia.

4.1 Esempio

Si vuole costruire uno heap dal vettore $(3, 6, 1, 4, 7, 2, 5)$.



Si ottiene così il vettore heap-ordinato $(7, 6, 5, 4, 3, 2, 1)$.

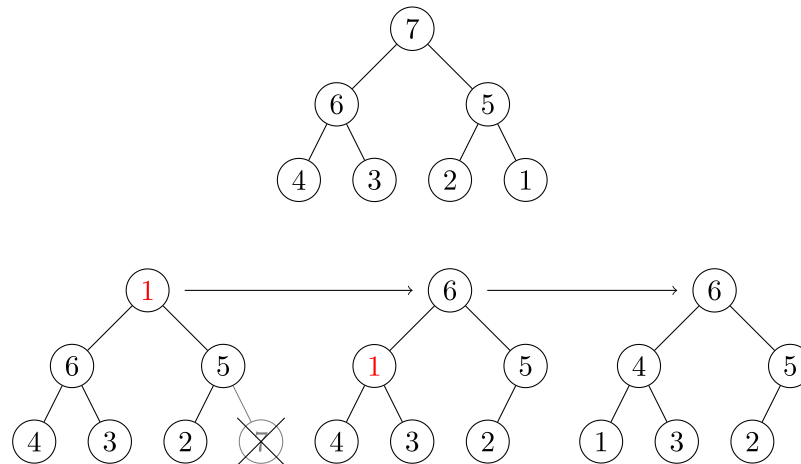
5 Rimozione

L'elemento da cancellare è quello con priorità maggiore, situato alla radice, ma, per mantenere completo l'albero binario, è possibile eliminare solo l'ultima foglia (la più a destra dell'ultimo livello).

Di conseguenza, la rimozione si esegue mediante il seguente procedimento:

1. si scambiano la radice e l'ultima foglia, cioè il primo e l'ultimo elemento del vettore;
2. si elimina l'ultima foglia, che adesso contiene l'elemento da cancellare;
3. si ripristina lo heap, facendo sprofondare la nuova radice fino alla posizione corretta.

5.1 Esempio



6 Implementazione

```
public class MaxPQ {
    private Comparable[] pq;
    private int n = 0;

    public MaxPQ(int dim) {
        pq = new Comparable[dim + 1];
    }
```

```
public boolean isEmpty() { return n == 0; }

public int size() { return n; }

public void insert(Comparable v) {
    pq[++n] = v;
    swim(n);
}

public Comparable read() { return pq[1]; }

public Comparable delete() {
    Comparable max = pq[1];
    exch(1, n--);
    pq[n + 1] = null;
    sink(1);
    return max;
}

public void buildBottomUp(Comparable[] a) {
    if (a.length > pq.length - 1) return;

    n = a.length;
    for (int i = 0; i < a.length; i++)
        pq[i + 1] = a[i];
    for (int i = n / 2; i >= 1; i--)
        sink(i);
}

private boolean less(int i, int j) {
    return pq[i].compareTo(pq[j]) < 0;
}

private void exch(int i, int j) {
    Comparable t = pq[i];
    pq[i] = pq[j];
    pq[j] = t;
}

private void swim(int k) {
    while (k > 1 && less(k / 2, k)) {
        exch(k / 2, k);
        k = k / 2;
    }
```

```
    }

    private void sink(int k) {
        while (2 * k <= n) {
            int j = 2 * k;
            if (j < n && less(j, j + 1)) // Seleziona il figlio maggiore
              j++;
            if (!less(k, j)) break;
            exch(k, j);
            k = j;
        }
    }
}
```

Osservazioni:

- Per semplicità, in quest'implementazione le priorità degli elementi sono date dai valori degli elementi stessi.
- Il vettore `pq` ha dimensione `dim + 1` perché, per semplificare i calcoli, vengono usate solo le posizioni a partire dall'indice 1, saltandone quindi una.
- Il metodo `delete`, oltre a implementare l'operazione di rimozione, restituisce il valore eliminato.
- La costruzione top-down si esegue mediante chiamate ripetute al metodo `insert`:

  ```
  Comparable[] b = new Comparable[B_SIZE];
  // ...
  MaxPQ codap = new MaxPQ(B_SIZE);
  for (int i = 0; i < B_SIZE; i++)
      codap.insert(b[i]);
  ```

- `swim` esegue un confronto per ogni livello.
- `sink` esegue due confronti per ogni livello.

7 Costi

Uno heap con n elementi corrisponde a un albero binario completo con altezza pari a circa $\log_2 n$. Di conseguenza, i costi delle operazioni sono:

- swim: $O(\log n)$
  - $O(1)$ nel caso migliore, quando il padre è già maggiore del valore da far risalire;
  - $\Theta(\log n)$ nel caso peggiore, cioè se si risale fino alla radice;

- sink: $O(\log n)$
  - $O(1)$ nel caso migliore, che si ha quando entrambi i figli del nodo considerato sono già minori;
  - $\Theta(\log n)$ nel caso peggiore, quando si discende fino alle foglie;
- insert e delete: hanno costo pari a un'operazione di swim e di sink, rispettivamente;
- read: sempre $O(1)$, perché l'elemento maggiore si trova alla radice;
- costruzione top-down: $O(n \log n)$
  - $\Theta(n)$ nel caso migliore, che corrisponde a n inserimenti con costo $O(1)$ (ad esempio, se il vettore è in ordine decrescente);
  - $\Theta(n \log n)$ nel caso peggiore, cioè se tutti gli inserimenti risalgono fino alla radice (ad esempio, se il vettore è in ordine crescente), e allora ciascuno degli ultimi $\frac{n}{2}$ inserimenti richiede tempo $\Theta(\log n)$;[1]
- costruzione bottom-up: sempre $\Theta(n)$.

8 Heapsort

Uno heap può essere utilizzato per definire un algoritmo di ordinamento, chiamato **heapsort**:

```
public static void sort(Comparable[] a) {
    MaxPQ cp = new MaxPQ(a.length);
    cp.buildBottomUp(a);
    for (int i = a.length - 1; i >= 0; i--)
        a[i] = cp.delete();
}
```

1. Si costruisce uno heap a partire dal vettore da ordinare (utilizzando il metodo bottom-up, più efficiente).
2. Si rimuovono, uno alla volta, gli elementi dello heap, posizionando ciascuno nel vettore, a partire dalla fine (perché viene rimosso prima l'elemento maggiore).[2]

[1] In questo caso, il tempo richiesto da ogni inserimento è $O(\log n)$ in generale, e dipende dall'altezza dell'albero quando viene effettuato: gli ultimi $\frac{n}{2}$ inserimenti richiedono esattamente $\Theta(\log n)$ ciascuno perché vengono effettuati quando l'albero ha altezza $h - 1$ o h (dove $h \sim \log_2 n$ è l'altezza finale).

[2] Quest'algoritmo si può implementare senza utilizzare spazio aggiuntivo, cioè costruendo lo heap direttamente all'interno del vettore `a`: basta osservare che, nel processo di eliminazione dall'heap di ciascun elemento, questo viene scambiato con l'ultima foglia, e ciò equivale all'inserimento nel vettore a partire dalla fine.

8.1 Complessità

Il tempo richiesto è

$$T(n) = c_1 + \Theta(n) + n(c_2 + O(\log n)) = O(n \log n)$$

dove:

- c_1 è il costo (costante) di allocazione dello heap;
- $\Theta(n)$ corrisponde alla costruzione bottom-up;
- $n(c_2 + O(\log n))$ è il costo complessivo del ciclo, che esegue n iterazioni, ciascuna con costi:
  - c_2 (costante) per la gestione dell'iterazione e l'inserimento nel vettore;
  - $O(\log n)$ per la rimozione di un elemento dall'heap.

Di conseguenza, heapsort è un algoritmo *ottimale*.

8.2 Stabilità

Heapsort *non è stabile*.

Dimostrazione: Si considera come controesempio un vettore nel quale i due valori $A[8] = A[6]$ sono, rispettivamente, il massimo del primo e del secondo sottoalbero di $A[1]$. Quando il vettore viene heap-ordinato, entrambi questi elementi diventano le radici dei sottoalberi di appartenenza, mediante gli scambi:

$$A[8] \longleftrightarrow A[4] \longleftrightarrow A[2]$$
$$A[6] \longleftrightarrow A[3]$$

Non viene quindi rispettato l'ordine relativo.

8.3 Implementazione su liste

A differenza di merge sort e quicksort, l'algoritmo heapsort non si può implementare (ragionevolmente) su liste concatenate: aumenterebbe infatti il costo delle operazioni swim e sink, dato che non sarebbe più possibile accedere al padre o ai figli di un nodo in tempo costante.