Algoritmi e Strutture Dati
Heap e Heapsort

P. Massazza[1]

[1]Dipartimento di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria
Varese Italy

Outline

Outline

Heap (coda con priorità)

Siano U un insieme di elementi e P un insieme totalmente ordinato (priorità)

Definizione [Heap]

Uno Heap di elementi di tipo U e priorità P è un elemento di $(U \times P)^\star$ che supporta le operazioni di

Inserimento inserisci: $(U \times P)^\star \times U \times P \to (U \times P)^\star$
inserisci(H, e, p) inserisce nello heap H l'elemento e con priorità p

Rimozione cancella: $(U \times P)^\star \to (U \times P)^\star$
cancella(H) elimina dallo heap H l'elemento con priorità maggiore

Lettura leggi: $(U \times P)^\star \Rightarrow U$
leggi(H) restituisce l'elemento con priorità maggiore presente in H.

Heap (coda con priorità)

Siano U un insieme di elementi e P un insieme totalmente ordinato (priorità)

Definizione [Heap]

Uno Heap di elementi di tipo U e priorità P è un elemento di $(U \times P)^\star$ che supporta le operazioni di

Inserimento inserisci: $(U \times P)^\star \times U \times P \to (U \times P)^\star$
inserisci(H, e, p) inserisce nello heap H l'elemento e con priorità p

Rimozione cancella: $(U \times P)^\star \to (U \times P)^\star$
cancella(H) elimina dallo heap H l'elemento con priorità maggiore

Lettura leggi: $(U \times P)^\star \Rightarrow U$
leggi(H) restituisce l'elemento con priorità maggiore presente in H.

Heap (coda con priorità)

Siano U un insieme di elementi e P un insieme totalmente ordinato (priorità)

Definizione [Heap]

Uno Heap di elementi di tipo U e priorità P è un elemento di $(U \times P)^\star$ che supporta le operazioni di

Inserimento inserisci: $(U \times P)^\star \times U \times P \to (U \times P)^\star$
inserisci(H, e, p) inserisce nello heap H l'elemento e con priorità p

Rimozione cancella: $(U \times P)^\star \to (U \times P)^\star$
cancella(H) elimina dallo heap H l'elemento con priorità maggiore

Lettura leggi: $(U \times P)^\star \Rightarrow U$
leggi(H) restituisce l'elemento con priorità maggiore presente in H.

Heap (coda con priorità)

Siano U un insieme di elementi e P un insieme totalmente ordinato (priorità)

Definizione [Heap]

Uno Heap di elementi di tipo U e priorità P è un elemento di $(U \times P)^\star$ che supporta le operazioni di

Inserimento inserisci: $(U \times P)^\star \times U \times P \rightarrow (U \times P)^\star$
inserisci(H, e, p) inserisce nello heap H l'elemento e con priorità p

Rimozione cancella: $(U \times P)^\star \rightarrow (U \times P)^\star$
cancella(H) elimina dallo heap H l'elemento con priorità maggiore

Lettura leggi: $(U \times P)^\star \Rightarrow U$
leggi(H) restituisce l'elemento con priorità maggiore presente in H.

Vettori heap-ordinati

Implementazione efficiente di Heap $\Rightarrow$ vettori heap-ordinati

Definizione [vettore heap-ordinato]

Uno vettore A di lunghezza n si dice heap-ordinato se

$$\begin{aligned} \text{priority}(A[i]) &\geq \text{priority}(A[2i]) \quad 1 \leq i < n/2 \\ \text{priority}(A[i]) &\geq \text{priority}(A[2i+1]) \quad 1 \leq i < n/2 \\ \text{priority}(A[n/2]) &\geq \text{priority}(A[n]) \quad n \text{ pari} \end{aligned}$$

Un vettore heap-ordinato può essere pensato come un albero binario completo osservando che

$$A[1] \approx \text{radice},$$

$$\forall i, 1 < i \leq n, \; A[\lfloor i/2 \rfloor] \text{ è il padre di } A[i].$$

(i figli di $A[j]$ sono $A[2j]$ e $A[2j+1]$).

Vettori heap-ordinati

Implementazione efficiente di Heap $\Rightarrow$ vettori heap-ordinati

Definizione [vettore heap-ordinato]

Uno vettore A di lunghezza n si dice heap-ordinato se

$$
\begin{array}{rcll}
\text{priority}(A[i]) & \geq & \text{priority}(A[2i]) & 1 \leq i < n/2 \\
\text{priority}(A[i]) & \geq & \text{priority}(A[2i+1]) & 1 \leq i < n/2 \\
\text{priority}(A[n/2]) & \geq & \text{priority}(A[n]) & n \text{ pari}
\end{array}
$$

Un vettore heap-ordinato può essere pensato come un albero binario completo osservando che

$$A[1] \quad \approx \quad \text{radice},$$

$$\forall i, 1 < i \leq n, \ A[\lfloor i/2 \rfloor] \text{ è il padre di } A[i].$$

(i figli di $A[j]$ sono $A[2j]$ e $A[2j+1]$).

Vettori heap-ordinati

Implementazione efficiente di Heap $\Rightarrow$ vettori heap-ordinati

Definizione [vettore heap-ordinato]

Uno vettore A di lunghezza n si dice heap-ordinato se

$$\begin{array}{rcll} \text{priority}(A[i]) & \geq & \text{priority}(A[2i]) & 1 \leq i < n/2 \\ \text{priority}(A[i]) & \geq & \text{priority}(A[2i+1]) & 1 \leq i < n/2 \\ \text{priority}(A[n/2]) & \geq & \text{priority}(A[n]) & n \text{ pari} \end{array}$$

Un vettore heap-ordinato può essere pensato come un albero binario completo osservando che

$$A[1] \quad \approx \quad \text{radice},$$

$$\forall i, 1 < i \leq n,\ A[\lfloor i/2 \rfloor] \text{ è il padre di } A[i].$$

(i figli di $A[j]$ sono $A[2j]$ e $A[2j+1]$).

Vettori heap-ordinati

Implementazione efficiente di Heap $\Rightarrow$ vettori heap-ordinati

Definizione [vettore heap-ordinato]

Uno vettore A di lunghezza n si dice heap-ordinato se

$$\begin{aligned} \text{priority}(A[i]) &\geq \text{priority}(A[2i]) && 1 \leq i < n/2 \\ \text{priority}(A[i]) &\geq \text{priority}(A[2i+1]) && 1 \leq i < n/2 \\ \text{priority}(A[n/2]) &\geq \text{priority}(A[n]) && n \text{ pari} \end{aligned}$$

Un vettore heap-ordinato può essere pensato come un albero binario completo osservando che

$$A[1] \approx \text{radice},$$

$$\forall i, 1 < i \leq n,\ A[\lfloor i/2 \rfloor] \text{ è il padre di } A[i].$$

(i figli di $A[j]$ sono $A[2j]$ e $A[2j+1]$).

Heap: costruzione top-down

L'idea ruota intorno alla soluzione del seguente

Problema: supponi che i primi $i-1$ elementi formino uno heap, considera l'i-esimo elemento e ripristina lo heap di i elementi.

Soluzione: basta risalire il percorso che collega l'elemento i-esimo alla radice fino a trovare un elemento maggiore, facendo scorrere verso il basso gli elementi minori incontrati.

Heap: costruzione top-down

L'idea ruota intorno alla soluzione del seguente

Problema: supponi che i primi $i - 1$ elementi formino uno heap, considera l'i-esimo elemento e ripristina lo heap di i elementi.

Soluzione: basta risalire il percorso che collega l'elemento i-esimo alla radice fino a trovare un elemento maggiore, facendo scorrere verso il basso gli elementi minori incontrati.

```
public class MaxPQ<Comparable>{
 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;}
 private boolean less(int i,int j)
 private void exch(int i,int j)
 private void swim(int k)
 private void sink(int k)
}
```

```
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))j++;
   if(!less(k,j))break;
   exch(k,j);k=j;}
 }
```

Heap: costruzione top-down

```
Integer[] b=new Integer[SIZEB];
...
MaxPQ<Integer> codap=new MaxPQ<Integer>(m+1);
for(int i=0;i<SIZEB;i++) codap.insert(b[i]);
```

Heap: costruzione bottom-up

```
public void buildBU(Comparable[] a)
{if(a.length<pq.length){
  n=a.length;
  for(int i=0;i<a.length)pq[i+1]=a[i];
  for(int i=n/2;i>=1;i--)sink(i);
 }
}
```

Heap: costi

Si noti che uno heap con "n" elementi rappresenta un albero binario completo avente altezza circa $\log_2 n$. I costi della varie operazioni sono allora:

- **swim:** caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si risale l'albero fino alla radice)
- **sink:** caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si discende l'albero fino alle foglie)
- **insert:** come swim
- **delete:** come sink
- **read:** $O(1)$ (sempre)

Heap: costi

Si noti che uno heap con "n" elementi rappresenta un albero binario completo avente altezza circa $\log_2 n$. I costi della varie operazioni sono allora:

swim: caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si risale l'albero fino alla radice)

sink: caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si discende l'albero fino alle foglie)

insert: come swim

delete: come sink

read: $O(1)$ (sempre)

Heap: costi

Si noti che uno heap con "n" elementi rappresenta un albero binario completo avente altezza circa $\log_2 n$. I costi della varie operazioni sono allora:

- **swim:** caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si risale l'albero fino alla radice)
- **sink:** caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si discende l'albero fino alle foglie)
- **insert:** come swim
- **delete:** come sink
- **read:** $O(1)$ (sempre)

Heap: costi

Si noti che uno heap con "n" elementi rappresenta un albero binario completo avente altezza circa $\log_2 n$. I costi della varie operazioni sono allora:

- **swim:** caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si risale l'albero fino alla radice)
- **sink:** caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si discende l'albero fino alle foglie)
- **insert:** come swim
- **delete:** come sink
- **read:** $O(1)$ (sempre)

Heap: costi

Si noti che uno heap con "n" elementi rappresenta un albero binario completo avente altezza circa $\log_2 n$. I costi della varie operazioni sono allora:

swim: caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si risale l'albero fino alla radice)

sink: caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si discende l'albero fino alle foglie)

insert: come swim

delete: come sink

read: $O(1)$ (sempre)

Heap: costi

Si noti che uno heap con "n" elementi rappresenta un albero binario completo avente altezza circa $\log_2 n$. I costi della varie operazioni sono allora:

- **swim:** caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si risale l'albero fino alla radice)
- **sink:** caso migliore $O(1)$ (al primo confronto padre-figlio si termina), caso peggiore $O(\log n)$ (si discende l'albero fino alle foglie)
- **insert:** come swim
- **delete:** come sink
- **read:** $O(1)$ (sempre)

Heap: costo della costruzione

top down: caso migliore $O(n)$ (n inserimenti con costo $O(1)$, ad es. vettore ordinato al contrario)
caso peggiore $\Theta(n \log n)$ (gli ultimi $n/2$ inserimenti richiedono ciascuno $\Theta(\log n)$)

bottom up: si dimostra che il costo è sempre $\Theta(n)$

Heap: costo della costruzione

top down: caso migliore $O(n)$ (n inserimenti con costo $O(1)$, ad es. vettore ordinato al contrario)
caso peggiore $\Theta(n \log n)$ (gli ultimi $n/2$ inserimenti richiedono ciascuno $\Theta(\log n)$)

bottom up: si dimostra che il costo è sempre $\Theta(n)$

Heap: costo della costruzione

top down: caso migliore $O(n)$ (n inserimenti con costo $O(1)$, ad es. vettore ordinato al contrario)
caso peggiore $\Theta(n \log n)$ (gli ultimi $n/2$ inserimenti richiedono ciascuno $\Theta(\log n)$)

bottom up: si dimostra che il costo è sempre $\Theta(n)$

Heap: costo della costruzione

top down: caso migliore $O(n)$ (n inserimenti con costo $O(1)$, ad es. vettore ordinato al contrario)
caso peggiore $\Theta(n \log n)$ (gli ultimi $n/2$ inserimenti richiedono ciascuno $\Theta(\log n)$)

bottom up: si dimostra che il costo è sempre $\Theta(n)$

Outline

Heapsort

Possiamo usare uno heap per definire un algoritmo di ordinamento ottimale

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

Costo: (Heap)sort(a) richiede tempo

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

Heapsort

Possiamo usare uno heap per definire un algoritmo di ordinamento ottimale

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

Costo: (Heap)sort(a) richiede tempo

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

Heapsort

Fatto

L'algoritmo Heapsort non è stabile

Dimostrazione: supponete di avere un vettore in cui i due valori A[8] e A[6] sono uguali e risultano rispettivamente il massimo del primo e del secondo sottoalbero di A[1]. Una volta che il vettore sarà heap-ordinato, il primo verrà posto in A[2] mentre il secondo finirà in A[3]. L'ordine relativo non è quindi rispettato.

Heapsort

Fatto

L'algoritmo Heapsort non è stabile

Dimostrazione: supponete di avere un vettore in cui i due valori A[8] e A[6] sono uguali e risultano rispettivamente il massimo del primo e del secondo sottoalbero di A[1]. Una volta che il vettore sarà heap-ordinato, il primo verrà posto in A[2] mentre il secondo finirà in A[3]. L'ordine relativo non è quindi rispettato.