Algoritmi (9 CFU)
(A.A. 2009-10)

Heap e Algoritmo HeapSort.

Overview

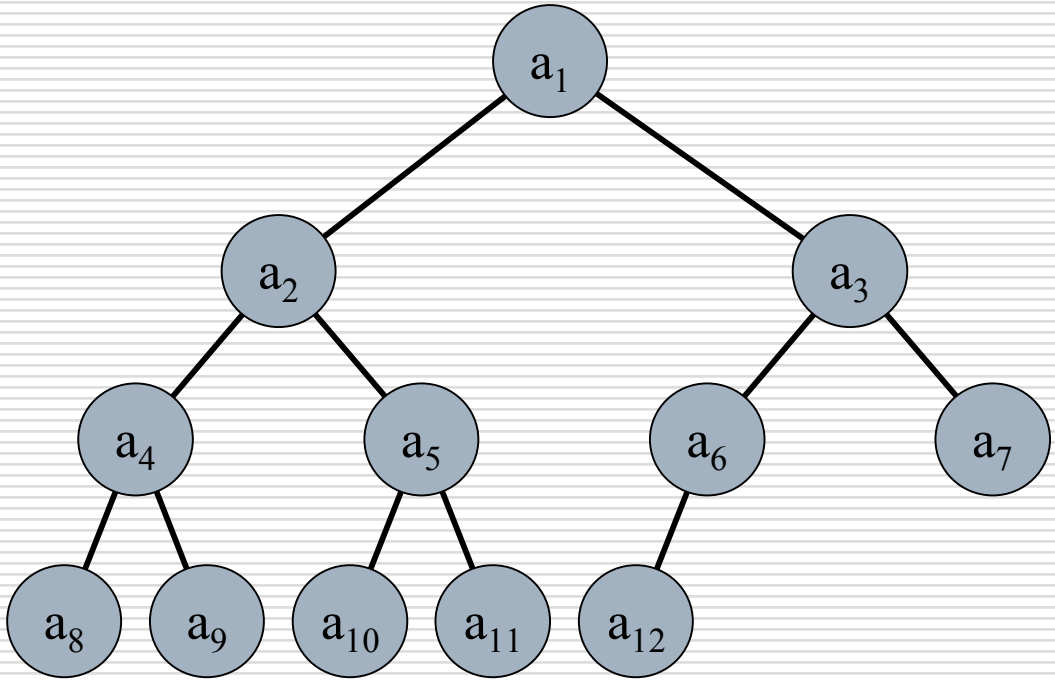
- Definiamo la struttura dati heap
- Operazioni di costruzione e gestione di un heap
- Algoritmo Heapsort
- Code di Priorità

Array come Albero Binario

- Un array A[1…n] può essere visto come un albero binario, ossia
  - A[1] è la radice
  - A[2i] e A[2i+1] sono i figli di A[i]
  - A[⌊i / 2⌋] è il padre di A[i]
- Nota che l'albero binario che si definisce in questo modo è
  - Completo sino al penultimo livello
  - L'ultimo livello potrebbe essere parzialmente completo ma è riempito da sinistra a destra.

Esempio

- A=[a_1, …, a_{12}]

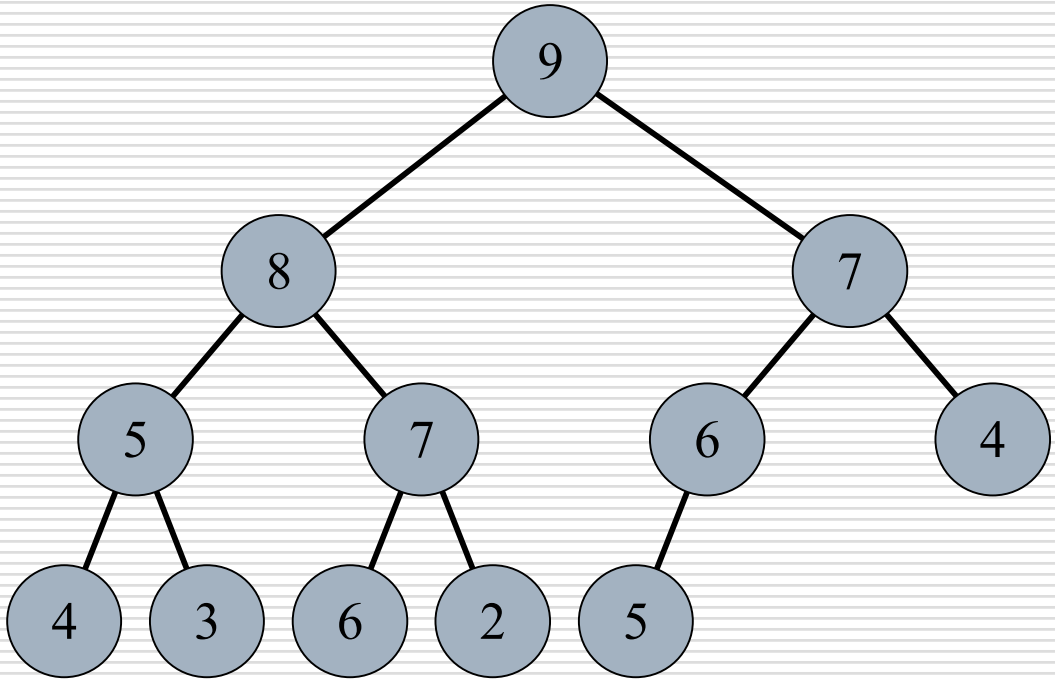


Vincolo numerico

- Se per ogni i, A[i] ≥ max(A[2i],A[2i+1], allora l'array viene detto Max-Heap
- Se per ogni i, A[i] ≤ min(A[2i],A[2i+1], allora l'array viene detto Min-Heap

Esempio: Max-Heap

- A=[9, 8, 7, 5, 7, 6, 4, 4, 3, 6, 2, 5]



Proprietà

- In una Max-Heap
  - Il massimo è la radice
  - Il minimo una delle foglie
- In una Min-Heap
  - Il Minimo è la radice
  - Il massimo una delle foglie

Conservare le proprietà di Heap

- Max-Heapify e Min-Heapify si applicano a partire dal nodo j supponendo che i sottoalberi di sinistra e di destra siano rispettivamente Max-Heap e Min-Heap

```
Max_Heapify(A,j,n)
  k=j
  if 2j+1 ≤ n and A[2j+1] > A[k]
  then k=2j+1
  if 2j ≤ n and A[2j] > A[k]
  then k=2j
  if k ≠ j
  then t=A[j], A[j]=A[k], A[k]=t
       Max_Heapify(A,k,n)
```

```
Min_Heapify(A,j,n)
  k=j
  if 2j+1 ≤ n and A[2j+1] < A[k]
  then k=2j+1
  if 2j ≤ n and A[2j] < A[k]
  then k=2j
  if k ≠ j
  then t=A[j], A[j]=A[k], A[k]=t
       Min_Heapify(A,k,n)
```

Complessità

- Max-Heapify e Min-Heapify sono algoritmi ricorsivi.
- Equazione di ricorrenza per entrambi

$$T(k)=T(2k/3)+\Theta(1)$$

(*perché? Soluzione?*)

- Intuitivamente il numero di passi è l'altezza h del nodo su cui la procedura è chiamata. Quindi

$$O(h)$$

Costruzione di una Heap

- Utilizziamo Max-Heapify o Min-Heapify come subroutine

```
Build_Max-Heap(A,n)
   for i=⌊n/2⌋ downto 1
         Max_Heapify(A,i,n)
```

```
Build_Min-Heap(A,n)
   for i=⌊n/2⌋ downto 1
         Min_Heapify(A,i,n)
```

Complessità

- Banalmente, n/2 chiamate a una procedura O(log n) quindi O(n log n). Ma
- Quanti nodi ci sono ad altezza h?
- Risposta: al massimo $\lceil n/2^{h+1} \rceil$ (perché?)
- Quindi complessità Build-Heap

$$\sum_h \lceil n/2^{h+1} \rceil O(h) = O(n \sum h/2^h)$$

- Problema:

$$\sum h/2^h = ???$$

Complessità

- Serie geometrica infinita con $0 < x < 1$

$$\sum x^h = 1/(1-x)$$

- Differenziando entrambi i termini

$$\sum h x^{h-1} = 1/(1-x)^2$$

- Moltiplicando per x entrambi i termini

$$\sum h x^h = x/(1-x)^2$$

- Ponendo $x=1/2$

$$\sum h(1/2)^h = (1/2)/(1-(1/2))^2 = 2$$

- Quindi complessità Build-Heap

$$\sum_h \lceil n/2^{h+1} \rceil O(h) = O(n \sum h/2^h) = O(n)$$

- E in conclusione

$$\Theta(n)$$

Algoritmo Heapsort

- Idea:
  - Costruiamo una Max-Heap
  - Scambiamo il primo elemento dell'array (massimo) con l'ultimo e chiamiamo Max-Heapify
- Complessità: O(n log n)

```
Heapsort(A)
    n=length(A)
    Build_Max-Heap(A,n)
    for i=n downto 2
        scambia(A[1],A[i])
        n=n-1;
        Max_Heapify(A,1,n)
```

Implementazione di code con priorità.

Le heap si possono usare, oltre che per ordinare un array, anche per implementare delle *code con priorità*.

Le code con priorità sono delle strutture dati in cui è possibile immagazzinare degli oggetti x a cui è attribuita una priorità $pr[x]$ ed estrarli uno alla volta in ordine di priorità decrescente.

Operazioni su code di priorità

Sulle code con priorità sono definite le operazioni:
***Insert*(*S*, *x*)**: aggiunge x alla coda S;
***Maximum*(*S*)**: ritorna $x \in S$ con $pr[x]$ massima
***ExtractMax*(*S*)**: toglie e ritorna $x \in S$ con $pr[x]$ massima

Possono inoltre essere definite anche le operazioni:
***IncreasePr*(*S*,*x*,*p*)**: aumenta la priorità di x
***ChangePr*(*S*,*x*,*p*)**: cambia la priorità di x

Operazioni su code di priorità

```
Maximum (S)                          ▷ S è una heap
    n ← size[S]
    if n = 0 then return nil
    else  return S[1]
```

$$T_{\max}^{Maximum}(n) = \Theta(1)$$

```
ExtractMax (S)                       ▷ S è una heap
    n ← size[S]
    if n = 0 then return nil
    else
        x ← S[1], S[1] ← S[n], size[S] ← n-1
        Heapfy(S,1,n-1)
        return x
```

$$T_{\max}^{ExtractMax}(n) = O(\log n)$$

Per realizzare *Insert* e *IncreasePr* ci serve una versione diversa della *Heapify* che invece di usare la proprietà:

"$H_n(i)$: $A[i]$ è maggiore o uguale di ogni suo discendente in $A[1..n]$"

che se verificata per ogni elemento dell'array caratterizza la struttura di mucchio, usa la proprietà simmetrica:

"HR(i) : $A[i]$ è minore o uguale di ogni suo ascendente"

(non serve specificare n!!!)

Anche questa, se verificata per ogni elemento dell'array, caratterizza la struttura di mucchio.

La nuova versione *HeapifyR* della *Heapify* ricostruisce il mucchio quando tutti gli elementi dell'array soddisfano la proprietà HR(*i*) tranne quello in posizione j.

```
HeapifyR(S,j)
   ▷ solo A[j] non soddisfa HR(j)
   while j > 1 and A[⌊j/2⌋] < A[j] do
      t ← A[⌊j/2⌋], A[⌊j/2⌋] ← A[j], A[j] ← t
      ▷ solo A[⌊j/2⌋] non soddisfa HR(⌊j/2⌋)
      j ← ⌊j/2⌋
```

$$T_{\max}^{HeapfyR}(j) = O(\log j)$$

```
IncreasePr (A,i,p)        ▷ A è una heap
    if p < A[i] then
        "errore: la nuova priorità è minore della vecchia"
    else
        A[i] ← p
        HeapifyR(S,i)
```

$$T_{\max}^{IncreaseKey}(i) = O(\log i)$$

```
Insert (A,x)                    ▷ S è una heap
    n ← size[A]+1, A[n] ← x
    HeapifyR(A,n)
    size[A] ← n
```

$$T_{\max}^{Insert}(n) = O(\log n)$$

Possiamo facilmente realizzare anche una *ChangePr* nel modo seguente:

```
ChangePr (A,i,p)          ▷ S è una heap
    if p < A[i] then
        n ← size[A]
        A[i] ← p
        Heapify(A,i,n)
    else
        A[i] ← p
        HeapifyR(A,i)
```

$$T_{\max}^{ChangeKey}(n) = O(\log n)$$