PROGRAMMAZIONE AVANZATA
ed elementi di
Ingegneria del Software

HEAP

Alberi (richiamo)

Un *albero* è un grafo connesso, aciclico, non orientato

– un albero è *radicato* se un nodo viene indicato come la *radice*

Ogni nodo dell'albero è raggiungibile dalla radice tramite un cammino (che è unico, in quanto il grafo è aciclico).

Chiamiamo *foglie* gli ultimi nodi dei cammini dalla radice.

Alberi

- Ogni nodo ha un *padre* (a parte la radice) e uno o più *figli* (a parte le foglie).

- Chiamiamo:
  - *nodi interni*: tutti i nodi dei cammini tra la radice e le foglie
  - *profondità* (di un nodo *N*): la distanza di *N* dalla radice
  - *altezza* (dell'albero): la distanza massima tra la radice e una foglia
  - *antenato* (di un nodo *N*): ogni nodo che precede *N* sul cammino dalla radice a *N*
  - *padre* (di un nodo *N*): il nodo che immediatamente precede *N* lungo il cammino dalla radice a *N*
  - *figlio* (di un nodo *N*): ogni nodo di cui *N* è padre
  - *fratelli* (di un nodo *N*): i nodi che hanno lo stesso padre di *N*

- Un albero è *binario* se ogni nodo ha *al più* 2 figli

Alberi

Definizioni

Un albero binario è un albero i cui nodi hanno **grado** compreso tra 0 e 2.

Grado: numero di archi incidenti sul nodo.

Un albero binario completo è un albero binario in cui ogni livello, tranne eventualmente l'ultimo, è completamente pieno, e tutti i nodi sono il più a sinistra possibile.

Un albero viene chiamato albero quasi completo se l'ultimo livello non è completamente pieno.

Alberi

Heap

Un heap (binario) è una struttura dati composta da un array che possiamo considerare come un **albero binario quasi completo.**

Ogni nodo dell'albero corrisponde a un elemento dell'array che memorizza il valore del nodo.
Tutti i livelli dell'albero sono completamente riempiti, tranne eventualmente l'ultimo che può essere riempito da sinistra fino a un certo punto.

Heap

Array One-based

$$Parent\ (i) = \lfloor i/2 \rfloor$$

$$Left(i) = 2i$$

$$Right(i) = 2i + 1$$

$\lfloor\ \rfloor$ intero inferiore

Heap

Array Zero-based

$$Parent\ (i) = \left\lfloor \frac{i-1}{2} \right\rfloor$$

$$Left(i) = 2i + 1$$

$$Right(i) = 2(i + 1)$$

$\lfloor\ \rfloor$ intero inferiore

Heap

0
16
1
14
2
10
3
8
4
7
5
9
6
3
7
2
8
4
9
1

0 1 2 3 4 5 6 7 8 9
16 14 10 8 7 9 3 2 4 1 ... ...
heap-size
len(A)

Heap

Esercizio

1. Implementate le funzioni parent, left, right e testatele esaustivamente utilizzando il vettore nella figura precedente (file heap_baseAccess.c).

2. separate il main dall'heap utilizzando un file per il main e uno o più file per l'heap.

heap_baseAccess.c

Max Heap

Max-heap: i valori dei nodi soddisfano la seguente proprietà: per ogni nodo *i* diverso dalla radice, si ha

$$A[\textsc{Parent}(i)] \geq A[i]$$

il valore di un nodo e` al massimo il valore di suo padre

- L'elemento piu` grande di un max-heap e` memorizzato nella radice
- Il sottoalbero di un nodo contiene valori minori o uguali di quello contenuto nel nodo stesso

(esiste anche min-heap, con proprietà opposte)

Heap

Altezza di un nodo:
numero di archi
fino alla foglia

Altezza di un heap:
altezza della radice

L'altezza di un heap è $\lfloor \log_2 n \rfloor$

Max Heap

Se un albero binario gode della proprietà di Max-heap, il valore massimo si trova sempre nella radice.

Sfruttando questa proprietà possiamo ordinare un vettore seguendo questi passi, illustrati nel dettaglio più avanti:

- si crea un Max-heap
- si prende l'elemento in radice, eliminandolo dall'albero. Otteniamo l'elemento più grande.
- si ripristina la proprietà di max-heap
- si prende il nuovo elemento in radice, eliminandolo dall'albero. Otteniamo il secondo elemento più grande.

e così via.

MAX-HEAPIFY

MAX-HEAPIFY

Funzione che permette di ripristinare le proprietà dell'heap **nel caso in cui un nodo sia più piccolo dei suoi figli.**

MAX-HEAPIFY

MAX-HEAPIFY

Gli alberi binari con radici in LEFT(i) e RIGHT(i) sono max-heap, **ma A[i] può essere più piccolo dei suoi figli**, violando così la proprietà del max-heap.

MAX-HEAPIFY

MAX-HEAPIFY fa scendere A[i] nel max-heap in modo che il sottoalbero con radice di indice *i* diventi un max-heap.

Attenzione: devo scambiare il 4 con il 14 e non con il 7
Altrimenti non ottengo un max-heap!

MAX-HEAPIFY

MAX-HEAPIFY

- max=max(A[left(i)], A[right(i)])
- se A[i]> max, termina
- altrimenti scambia A[i] col massimo

MAX-HEAPIFY

MAX-HEAPIFY

- max=max(A[left(i)], A[right(i)])
- se A[i]> max, termina
- altrimenti scambia A[i] col massimo
- applica ricorsivamente

MAX-HEAPIFY

Esercizio
Implementate MAX-HEAPIFY in modo ricorsivo

Possibile implementazione:
max_heapify(**int*** A, **int** i, **int** heap_size)

MAX-HEAPIFY

```
max_heapify(A,i,heap_size)
Calcolo l, r
se l è valido,
    se A[l]>A[i], MASSIMO=A[l] .
    Altrimenti MASSIMO=A[i]
se r è valido,
    se A[r]>MASSIMO, MASSIMO=A[r].
    Altrimenti PASS;
se MASSIMO!=A[i],
    scambia
    max_heapify(A,idx_max,heap_size)
altrimenti STOP
```

MAX-HEAPIFY

```
void max_heapify(int* A, int i, int heap_size){
    int l, r, idx_max, max;
    l=left(i);
    r=right(i);


    // se l è valido,
    //  se A[l]>A[i],  MASSIMO=A[l] . Altrimenti MASSIMO=A[i]
    if ( l<heap_size && A[l]>A[i] ){
        idx_max=l;
        max=A[idx_max];
    }
    else{
        idx_max=i;
        max=A[idx_max];
    }

    // nota: posso fare a meno di assegnare  max=A[idx_max]
    // lo uso solo per semplicità di notazione
```

MAX-HEAPIFY

```
    // se r è valido,
    //  se A[r]>MASSIMO,  MASSIMO=A[r]. Altrimenti PASS;
    if ( r<heap_size && A[r]>max ){
        idx_max=r;
        max=A[idx_max];
    }

   // se il massimo NON è A[i],
    //  scambia, e applica max_heapify al nodo scambiato
    //  altrimenti la procedura si conclude
   //     (è già un max_heap)
    if (idx_max!=i){
        A[idx_max]= A[i];
        A[i]=max;
        max_heapify(A, idx_max,  heap_size);
    }
}
```

MAX-HEAPIFY

Esercizio
Implementate MAX-HEAPIFY in modo NON ricorsivo

Possibile implementazione:
max_heapify(**int*** A, **int** i, **int** heap_size)

MAX-HEAPIFY

max_heapify: calcolo complessità computazionale

Dato che **max_heapify** agisce sul nodo *i* e poi, ricorsivamente, su uno dei sottoalbero sinistro o destro, per calcolare la complessità computazionale $T(n)$ su un albero di n nodi possiamo scrivere $T(n)$ in funzione del costo di applicare **max_heapify** ad un sottoalbero.

Se n' è la dimensione del sottoalbero, allora

$$T(n) = \mathrm{T}(n') + costante$$

MAX-HEAPIFY

$$T(n) = \mathrm{T}(n') + costante$$

Soluzione approssimata. Ipotizziamo che $n' \cong n/2$ (che è falso, ma è una ipotesi di lavoro).

$$T(n) = \mathrm{T}(n/2) + costante$$

È nella forma

$$C_n = C_{n/2} + 1; n \geq 2; C_1 = 1$$

$$C_n = \log_2 n + 1 \quad \Longrightarrow \quad \boldsymbol{O(log_2\, n)}$$

MAX-HEAPIFY

$$T(n) = \mathrm{T}(n') + costante$$

Soluzione corretta, per chi si vuole cimentare… (porta alla stessa soluzione asintotica, ma i valori delle costanti sono diverse)

Si dimostra facilmente, ragionando sull'albero binario, che

$$\frac{n'}{n} < \frac{2}{3}$$

MAX-HEAPIFY

max_heapify: calcolo complessità computazionale

Dato che **max_heapify** agisce sul nodo *i* e poi, ricorsivamente, su uno dei sottoalbero sinistro o destro, per calcolare la complessità computazionale $T(n)$ su un albero di n nodi possiamo scrivere $T(n)$ in funzione del costo di applicare **max_heapify** ad un sottoalbero.

Se n' è la dimensione del sottoalbero, allora

$$T(n) = \mathrm{T}(n') + costante$$

Mostreremo ora che

$$\frac{n'}{n} < \frac{2}{3}$$

Parte non richiesta

Heap Sort

max_heapify: calcolo complessità computazionale

Premessa:

Un albero binario (completo o quasi completo) di altezza k ha numero di nodi n compreso tra 2^ke $2^{k+1} - 1$

Esempio: $k = 3 \rightarrow \ n$ compreso tra $2^3 = 8$ e $2^4 - 1 = 15$

Heap Sort

Infatti il **massimo numero di nodi** (albero completo) è:

$n_{max}\ (k = 0) = 2^0$

$n_{max}(k = 1) = 2^0 + 2^1$

…

$$n_{max}(k) = 2^0 + 2^1 + \cdots + 2^k = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

mentre il **minimo numero di nodi** è pari al numero di nodi di un albero di altezza $k - 1$ aumentato di 1

$$n_{min}(k) = n_{max}(k - 1) + 1 = \ 2^k$$

Parte non richiesta

Heap Sort

Dato un albero di altezza k, quanti nodi può avere il sottoalbero SX?

Il sottoalbero SX è di altezza k-1 quindi avrà

$$2^{k-1} \leq n' \leq 2^k - 1$$

Heap Sort

Dato un albero di altezza k, quanti nodi può avere il sottoalbero DX?

Il sottoalbero DX è di altezza k-1 o k-2 (ma completo), come in figura, quindi avrà

$$2^{k-1} - 1 \leq n' \leq 2^k - 1$$

Heap Sort

È facile mostrare che il rapporto n'/n è massimo se SX è pieno e DX è vuoto, e facendo riferimento a n' di SX.

Con questa configurazione si ha:

$n' = (2^k - 1)$;

$$n = \underbrace{(2^k - 1)}_{\text{SX}} + \underbrace{(2^{k-1} - 1)}_{\text{DX}} + \underbrace{1}_{\text{root}} = 3 \cdot 2^{k-1} - 1$$

Heap Sort

$$n' = 2^k - 1 = 2 \cdot 2^{k-1} - 1;$$
$$n = 3 \cdot 2^{k-1} - 1$$

$$\frac{n'}{n} = \frac{2 \cdot 2^{k-1} - 1}{3 \cdot 2^{k-1} - 1}$$

che è nella forma

$$f(x) = \frac{2 \cdot x - 1}{3 \cdot x - 1}; \lim_{x \to \infty} f(x) = \frac{2}{3}; \frac{df}{dx} = \frac{1}{(1 - 3\,x)^2} > 0$$

Dunque

$$\frac{n'}{n} < \frac{2}{3}$$

Parte non richiesta

Heap Sort

max_heapify: calcolo complessità computazionale

Consideriamo un sottoalbero di radice i e dimensione n.

I due sottoalberi $left(i)$ e $right(i)$ hanno $n' < \frac{2}{3}n$

Heap Sort

max_heapify: calcolo complessità computazionale

Consideriamo un sottoalbero di radice i e dimensione n.

I due sottoalberi $left(i)$ e $right(i)$ hanno $n' < \frac{2}{3}n$

Sia $T(n)$ il tempo di esecuzione di max_heapify sul sottoalbero di radice i.

$$T(n) < \mathrm{T}\left(\frac{2}{3}n\right) + c$$

$\mathrm{T}\left(\frac{2}{3}n\right)$ è legato al tempo massimo per applicare max_heapify ad uno dei due sottoalberi

c è il tempo costante necessario per sistemare le relazioni fra i nodi i , $left(i)$ e $right(i)$

Parte non richiesta

Heap Sort

max_heapify: calcolo complessità computazionale

Relazione di ricorrenza

$$T(n) \cong \begin{cases} c & n \leq 3 \\ T\left(\frac{2}{3}n\right) + c & n > 3 \end{cases}$$

Soluzione

$$T(n) = O(\lg_2 n)$$

Parte non richiesta

BUILD-MAX-HEAP

BUILD-MAX-HEAP

Funzione che permette di costruire un max-heap

Convertiamo un array di n elementi in un max-heap di *n* elementi

A	4	1	3	2	16	9	10	14	8	7

BUILD-MAX-HEAP

BUILD-MAX-HEAP

Funzione che permette di costruire un max-heap

Convertiamo un array di n elementi in un max-heap di n elementi

A	4	1	3	2	16	9	10	14	8	7

È banale applicare insertion-sort o simili. Ma avrei O(n^2).
Invece so che max-heapify è O(lg n). Provo a sfruttare questa proprietà

BUILD-MAX-HEAP

BUILD-MAX-HEAP

parto da i=n/2 e applico **max_heapify(A,i)**

decremento i e applico **max_heapify(A,i)**

...

A	4	1	3	2	16	9	10	14	8	7

Nota: con i=n/2
in quale posizione sono nell'albero?

n=10;
per comodità gli indici partono da 1

BUILD-MAX-HEAP

BUILD-MAX-HEAP

parto da i=n/2 e applico **max_heapify(A,i)**

decremento i e applico **max_heapify(A,i)**

...

A	4	1	3	2	16	9	10	14	8	7

Nota: con i=n/2
in quale posizione sono nell'albero?

BUILD-MAX-HEAP

parto da i=n/2 e applico **max_heapify(A,i)**
decremento i e applico **max_heapify(A,i)**
...

BUILD-MAX-HEAP

BUILD-MAX-HEAP

Ogni volta che viene chiamata la procedura MAX-HEAPIFY per un nodo, **i due sottoalberi del nodo sono <u>già</u> entrambi max-heap**.

BUILD-MAX-HEAP

BUILD-MAX-HEAP

Ora ho un MAX-HEAP

BUILD-MAX-HEAP

IMPLEMENTARE IN C…

BUILD-MAX-HEAP

```
void build_max_heap(int* A, int heap_size){
    for (int i=(heap_size/2)-1; i>=0; i--){
        max_heapify(A, i,  heap_size);

    }
}
```

Heap Sort

Complessità computazionale di BUILD_MAX_HEAP

Ogni chiamata di MAX-HEAPIFY costa un tempo $O(\lg n')$ con $n' \leq n$, e ci sono $O(n)$ di queste chiamate. Quindi, il tempo di esecuzione **massimo** è limitato da $O(n \lg n)$.
Questo è un limite superiore molto ampio.
Il tempo di esecuzione varia con l'altezza del nodo, e la maggior parte dei nodi hanno altezza piccola.

Si dimostra che la complessità computazionale è limitata da $O(n)$.
Possiamo costruire un max-heap da un array non ordinato in tempo lineare.

Heap Sort

Algoritmo HEAPSORT

Heapsort ordina un array
- costruendo prima un max-heap mediante l'algoritmo BUILD-MAX-HEAP,
- estraendo il primo elemento (che per la proprietà degli max_heap è nella radice) e scambiandolo con l'ultimo
- ripristinando la proprietà di max_heap
- e iterando il procedimento.

Heap Sort

Algoritmo HEAPSORT

- L'elemento più grande è memorizzato nella radice (elemento A[0] del vettore). Può essere inserito nella sua posizione finale corretta scambiandolo con A[n-1], ed eliminandolo dall'heap.
- I figli della radice sono max-heap, ma la radice è errata. Mediante **max_heapify** (A, 1) ripristiniamo la proprietà max_heap.
- Riapplico al max-heap di dimensione n-1
- Alla fine il vettore sarà ordinato in ordine inverso e memorizzato nello stesso array che prima memorizzava l'heap-

(per comodità gli indici partono da 0)

Heap Sort

A	4	1	3	2	16	9	10	14	8	7

Vettore DISORDINATO.
Applico BUILD_MAX_HEAP e ottengo

16	**14**	**10**	**8**	**7**	**9**	**3**	**2**	**4**	**1**

Heap Sort

max_heapify (A, 1)

Heap Sort

max_heapify (A, 1)

Heap Sort

max_heapify (A, 1)

Heap Sort

max_heapify (A, 1)

Heap Sort

max_heapify (A, 1)

Heap Sort

max_heapify (A, 1)

Heap Sort

max_heapify (A, 1)

Heap Sort

max_heapify (A, 1)

Heap Sort

max_heapify (A, 1)

Heap Sort

Vettore ordinato in ordine inverso e memorizzato nello stesso array che prima memorizzava l'albero.

A	1	2	3	4	7	8	9	10	14	16

Heap Sort

Implementare in c…

Heap Sort

```
void heapsort(int* A, int heap_size){
    int tmp;

    build_max_heap(A, heap_size);
    printArray(A, heap_size);
    for (int i=heap_size; i>1; i--){
     // swap radice con ultimo elemento
        tmp=A[i-1];
        A[i-1]= A[0];
        A[0]=tmp;

        max_heapify(A, 0,  i-1);
    }
}
```

Heap Sort

Complessità computazionale

- La chiamata di BUILD-MAX-HEAP impiega $O(n)$
- Ciascuna delle $n - 1$ chiamate di MAX- HEAPIFY impiega $O(\lg n)$.

HEAPSORT impiega quindi $O(n \lg n)$.

Complessità computazionale

Caso Peggiore:

insertion sort: $O(n^2)$
merge sort: $O(n \lg n)$
heapsort: $O(n \lg n)$.

NB: i fattori costanti di insertion sort lo rendono piu` veloce per piccoli valori di n.

Heap Sort

La sequenza ⟨23, 17, 14, 6, 13, 10, 1, 5, 7, 12⟩ e` un max-heap?

Illustrate l'azione di BUILD-MAX-HEAP sull'array A = ⟨5, 3, 17, 10, 84, 19, 6, 22, 9⟩. Utilizzate le figure dell'albero analogamente a quanto fatto sulle slides.

Perché l'indice i del ciclo di BUILD-MAX-HEAP deve diminuire da ⌊lunghezza[A]/2⌋ a 1, anziché aumentare da 1 a ⌊lunghezza[A]/2⌋?

Heap Sort

Illustrate l'azione di HEAPSORT sull'array A = ⟨5, 13, 2, 25, 7, 17, 20, 8, 4⟩. Utilizzate le figure dell'albero analogamente a quanto fatto sulle slides.

Qual e` il tempo di esecuzione di heapsort con un array A di lunghezza n che e` gia` ordinato in senso crescente? E se l'array e` ordinato in senso decrescente?

Heap Sort

MAX-HEAPIFY

Funzione che permette di ripristinare le proprietà dell'heap **nel caso in cui un nodo sia più piccolo dei suoi figli.**

INCREMENTATE il valore del nodo. Ha senso applicare MAX-HEAPIFY?

CODE DI PRIORITÀ

Applicazioni piu` diffuse dell'heap: coda di priorità.
(code di max-priorità e code di min-priorità).

Una coda di priorità è una struttura dati che serve a mantenere un insieme S di elementi, ciascuno con un valore associato detto chiave.
Una coda di max-priorità supporta le seguenti operazioni.

CODE DI PRIORITÀ

Un'applicazione delle code di max-priorità è quella di schedulare i task.

La coda di max-priorità tiene traccia dei compiti da svolgere e delle loro relative priorità. Quando un compito è ultimato o interrotto, viene selezionato il compito con priorità piu` alta fra quelli in attesa mediante EXTRACT-MAX. Un nuovo compito puo` essere aggiunto alla coda in qualsiasi istante mediante INSERT.

CODE DI PRIORITÀ

MAXIMUM(S) restituisce l'elemento di S con il valore piu` grande.

INSERT(S,x) inserisce l'elemento x nell'insieme S. Questa operazione potrebbe essere scritta cos`ı: $S \leftarrow S \cup \{x\}$.

EXTRACT-MAX(S) elimina e restituisce l'elemento di S con il valore piu` grande.

INCREASE-KEY (S, i, x) aumenta il valore dell'elemento i al nuovo valore x, che si suppone sia almeno pari al valore corrente dell'elemento i.
DECREASE-KEY idem.

Le funzioni devono accettare un max-heap e rendere un max-heap

Implementare le funzioni per la struttura duale min-heap

CODE DI PRIORITÀ

MAXIMUM(S) restituisce l'elemento di S con il valore piu` grande.

INSERT(S,x) inserisce l'elemento x nell'insieme S. Questa operazione potrebbe essere scritta cos`ı: $S \leftarrow S \cup \{x\}$.

EXTRACT-MAX(S) elimina e restituisce l'elemento di S con il valore piu` grande.

INCREASE-KEY (S, i, x) aumenta il valore dell'elemento i al nuovo valore x, che si suppone sia almeno pari al valore corrente dell'elemento i.
DECREASE-KEY idem.

Le funzioni devono accettare un max-heap e rendere un max-heap

Implementare le funzioni per la struttura duale min-heap

L'implementazione è lasciata come esercizio.
Nell'implementazione fate riferimento alle funzioni per la manipolazione dell'heap

Code priorità

Note su **increase-key** e **decrease-key**.

L'ipotesi di partenza è avere un max-heap

Dopo l'applicazione di **decrease-key** al nodo i, l'albero è nella condizione di applicabilita della max-heapify (è un max-heap tranne al più un nodo più piccolo)

Dopo l'applicazione di **increase-key** al nodo I, l'albero **NON** è nella condizione di applicabilita della max-heapify (non solo potrebbe essere parent(i)<A[i], ma anche parent(parent(i))< A[i].)

Applicare max-heapify non risolve: occorre spostare il nodo VERSO L'ALTO.

FONTI

Introduzione agli algoritmi e strutture dati
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein