Giorno 3

1 Heapsort

1.1 Heap

Lo HEAP è una struttura dati composta da un albero binario quasi completo che rispetta la *proprietà dell'heap*.

1.1.1 Proprietà dell'heap

Vi sono due tipi di heap: il MAX-HEAP ed il MIN-HEAP; a questi sono associate due diverse proprietà:

1. **Proprietà del Max-Heap**: Il valore di ogni nodo non foglia è maggiore di quello dei suoi figli.
2. **Proprietà del Min-Heap**: Il valore di ogni nodo non foglia è minore di quello dei suoi figli.

Di conseguenza, si ha che il massimo (-minimo) di un Max(-Min)-Heap si trova nella sua radice.

1.1.2 Gestione dell'heap

Uno heap viene memorizzato in un array A.
Oltre alla proprietà `length`, A ha anche una proprietà `heapsize`, che rappresenta la lunghezza della porzione di array da considerarsi heap.
Dato un indice i, associato al nodo $A[i]$, si usano le seguenti funzioni per accedere al parent e ad i figli:

$$\texttt{Parent(i) = } \left\lfloor \frac{i}{2} \right\rfloor; \qquad \texttt{Left(i) = 2i;} \qquad \texttt{Right(i) = 2i+1;}$$

Un'importante proprietà di questa rappresentazione è che i nodi in $A\left[\left\lceil \frac{\texttt{A.heapsize}}{2} \right\rceil..\texttt{A.heapsize}\right]$ sono tutti foglie, e che il massimo/minimo di un Max/Min-Heap si trova in $A[1]$;

1.1.3 Funzioni dell'heap

Le funzioni legate agli heap che vedremo sono le seguenti (le vedremo sui max-heap):

Max-Heapify(A, i)	Assume che i figli di $A[i]$ siano Max-Heap, rende i radice di uno heap.
Build-Max-Heap(A)	Prende in input un array non ordinato e lo rende uno heap.
Heapsort(A)	Heapifica A; Ordina A estraendo ripetutamente il massimo dall'heap.
Heap-Maximum(A)	*(Funzioni relative alle code di priorità)* - restituisce $A[1]$
Heap-Extract-Max(A)	Estrae il massimo dall'heap e ripristina la proprietà dell'heap
Heap-Increase-Key(A, i, k)	Aumenta il valore di un nodo e ripristina la proprietà dell'heap

1.1.4 Mantenere la proprietà dell'heap

Siano $A[Left(i)]$, $A[Right(i)]$ radici di due heap.

La procedura Max-Heapify controlla se $A[i]$ è maggiore dei suoi due figli, e se non lo è scambia $A[i]$ con il valore del massimo tra i suoi figli, e si chiama ricorsivamente su di esso.

```
Max-Heapify(A, i):
    if(Left(i) ≤ A.heapsize && A[Left(i)] > i):
        massimo = Left(i);

    if(Right(i) ≤ A.heapsize && A[Right(i)] > i):
        massimo = Right(i);

    if(A[i] ≠ A[massimo])
        scambia A[i], A[massimo];
        Max-Heapify(A, massimo);
```

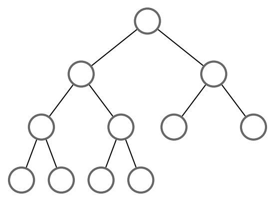
Costo di Max-Heapify: Un sottoalbero di un heap ha al più dimensione $\frac{2n}{3}$ ($\star$); la ricorrenza che definisce il costo della procedura è quindi:

$$T(n) < T\left(\frac{2n}{3}\right) + O(1)$$

Per il Master Theorem, $n^{log_{3/2}1} = 0 = \Theta(1)$ implica che (caso 2) $T(n) = O(\log n)$.

Dimostrazione. ($\star$)

Sia i la radice dell'heap A. Sia B il sottoalbero di dimensione massima che ha per radice un figlio di i. Il rapporto $\frac{\dim(B)}{\dim(A)}$ è massimo quando l'ultimo livello è pieno a metà.



In questo caso tale rapporto, espresso rispetto all'altezza h di B, è uguale a $\frac{2^{h+1}-1}{2^{h+1}+2^h-1}$.

Tale rapporto è sempre $< \frac{2}{3}$: infatti $\lim_{h\to+\infty} \frac{2^{h+1}-1}{2^{h+1}+2^h-1} = \lim_{h\to+\infty} \frac{2^h(2-\frac{1}{2^h})}{2^h(2+1-\frac{1}{2^h})} = \frac{2}{3}^-$ ∎

1.1.5 Costruire uno heap

La procedura Build-Max-Heap costruisce uno heap a partire da un array non ordinato, applicando ripetutamente Max-Heapify.

```
Build-Max-Heap(A):
    for(i = ⌊A.heapsize/2⌋ downto 1):
        Max-Heapify(A, i);
```

Costo di Build-Max-Heap: Una prima analisi potrebbe portare ad individuare un limite superiore di $n \log n$, dato che Max-Heapify costa $\log n$ ed il ciclo è ripetuto $n/2$ volte.

Questo limite superiore è corretto, ma non è stretto; una analisi più accurata viene dall'osservazione di alcune proprietà:

1. Il costo di Max-Heapify è $O(h)$
2. Il numero di nodi di altezza h è sempre al più $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.

Dimostrazione.

1. Uno heap è un albero binario quasi completo; tutti i livelli di profondità i tranne l'ultimo hanno esattamente 2^i nodi. Considerando che l'ultimo ha un numero di nodi compreso tra 1 e 2^h, si ha che $2^h \leq n \leq 2^{h+1}-1$. Quindi $h \leq \log n \leq \log(2^{h+1}-1) \leq h+1$, ossia $h = \lfloor \log n \rfloor$.

   Ciò implica che $O(h) = O(\log(n))$

2. Sia h l'altezza dell'heap. Il numero massimo di nodi al livello di altezza i è 2^{h-i}.

   La dimensione massima di uno heap è $2^{h+1}-1$. Quindi:

$$2^{h-1} \leq \left\lceil \frac{2^{h+1}-1}{2^{i+1}} \right\rceil = \left\lceil 2^{h+i} - \frac{1}{2^{i+1}} \right\rceil = 2^{h-1}$$

■

Quindi, per heapificare ogni nodo di ogni livello si impiega tempo inferiore a

$$\sum_{i=0}^{h} \left\lceil \frac{n}{2^{i+1}} \right\rceil O(i) = nO\left(\sum_{i=0}^{\lfloor \log n \rfloor} \frac{i}{2^{i+1}} \right) = nO\left(\sum_{i=0}^{\lfloor \log n \rfloor} \frac{i}{2^{i}} \right)$$

Sappiamo che $\sum_{i=0}^{+\infty} x^i = \frac{1}{1-x}$, per $|x| < 1$. Deriviamo entrambi i lati rispetto a x e moltiplichiamoli per x:

$$\text{(CLRS A.8)} \sum_{i=0}^{+\infty} ix^i = \frac{x}{(1-x)^2}$$

Sia $x = 1/2$:

$$nO\left(\sum_{i=0}^{\lfloor \log n \rfloor} \frac{i}{2^{i}} \right) \leq \frac{\frac{1}{2}}{(\frac{1}{2})^2} n = O(n)$$

Quindi un limite superiore più stretto alla complessità di Build-Max-Heap è $O(n)$.

1.2 Heapsort

Sappiamo che il massimo di un array max-heapificato sta in $A[1]$; Possiamo usare questa proprietà per ordinare l'array.

```
Heapsort(A):
    A.heapsize = A.length;
    Build-Max-Heap(A);
    for(i = A.length downto 2):
        Scambia A[1], A[i];
        A.heapsize--;
        Max-Heapify(A, 1);
```

Costo: $O(n \log n)$

1.3 Code di priorità

Una coda di priorità è una struttura dati simile ad una coda, ma tale che gli elementi siano inseriti e rimossi in base ad una *priorità*.

Nelle code di max-priorità, ad esempio, si inserisce "in testa" l'elemento massimo; alcune delle operazioni associate alle code di priorità sono:

- Insert(A, k): inserisce nella coda di priorità;
- Maximum(A): restituisce l'elemento massimo;
- Extract-Max(A): estrae il massimo;
- Increase-key: aumenta il valore di una chiave.

Si potrebbe usare un array, ordinato o meno, ma il costo della ricerca del massimo e dell'ordinamento sono considerevoli, perciò una scelta migliore è usare un heap di massimo.

```
Heap-Maximum(A): return A[1]
```

```
Heap-Extract-Maximum(A):
    if(A.heapsize < 1): error <underflow>
    maximum = A[1];
    A[1] = A[A.heapsize];
    A.heapsize--;
    Max-Heapify(A, 1);
    return maximum;
```

```
Heap-Increase-Key(A, i, k):
    if(k < A[i]) error <k minore del valore>;
    A[i] = k;
    j = i;
    while(i > 1 && A[j] > A[parent(j)]):
        scambia A[j], A[parent(j);
        j = parent(j);
```

```
Heap-Insert(A, k):
    A.heapsize++;
    A[A.heapsize] = -∞;
    Heap-Increase-Key(A, A.heapsize, k);
```

2 Ordinamento in tempo lineare

2.1 Counting Sort

Sia A un array con elementi in $[0, k]$. Counting Sort conta gli elementi minori o uguali ad ogni elemento di A. Il numero di elementi minori o uguali ad $A[i]$ è uguale ad i.

```
CountingSort(A, B, k):
    C = new array [0..k];
    for(i = 0 to k): C[i] = 0;
    for(i = 1 to A.length):
        C[A[i]]++;
    for(i = k downto 1):
        C[i] += C[i-1];
    for(i = A.length downto 1):
        B[C[A[i]]] = A[i];
        C[A[i]]--;
```

[radix + correttezza oc]