Laboratorio di Algoritmi e Strutture Dati

Esercitazioni dell'11 Dicembre 2012

Esercizio 1: heap, Heapsort, code di priorità

Un *max-heap (binario)* è una struttura dati composta da un albero binario A quasi completo tale che:

1. A ogni nodo p di A è associato un valore $v(p)$.
2. Per ogni nodo p di A diverso dalla radice, vale che, detto q il predecessore (padre) di p in A, allora $v(q) \geq v(p)$.

Si noti che il nodo di valore massimo in un max-heap è la radice.

Un *min-heap (binario)* è definito analogamente, sostituendo in (2) la condizione $v(q) \geq v(p)$ con $v(q) \leq v(p)$.

Ovviamente l'altezza di uno heap contenente n nodi è $O(\log n)$.

Gli heap possono essere facilmente implementati tramite array in questo modo. Sia A uno heap. Allora:

1. la radice di A sarà memorizzata nella posizione 0 dell'array.
2. Se p è memorizzato in posizione i allora:
   - Il padre $P(p)$ di p (se esiste) è memorizzato nella posizione $\lfloor (i-1)/2 \rfloor$.
   - Il figlio sinistro $L(p)$ di p (se esiste) è memorizzato nella posizione $2i+1$.
   - Il figlio destro $R(p)$ di p (se esiste) è memorizzato nella posizione $2i+2$.

Descriviamo brevemente le operazioni principali per manipolare gli heap. (Per brevità ci occuperemo solo di max-heap. Le corrispondenti operazioni per i min-heap sono del tutto analoghe.) (Si rimanda al libro di testo per maggiori dettagli).

- **heapify**(A, i):
  Converte l'array A in modo che il sottoalbero di radice $A(i)$ sia uno heap.
  In particolare, si supponga che A sia un array tale che i sottoalberi di radice $L(i)$ e $R(i)$ siano heap, ma $A(i)$ possa essere minore di uno o di entrambi i suoi figli, violando così la proprietà definitoria degli heap.
  Allora **heapify**(A, i) ripristina (localmente) la proprietà facendo scendere tramite scambi $A(i)$ nel sottoalbero di cui è radice, fino a che il nuovo sottoalbero di radice $A(i)$ è uno heap. Eccone lo pseudocodice.

  1. Determinare l'indice j del valore massimo tra $A(i)$, $A(L(i))$ e $A(R(i))$.
  2. Se $i = j$ abbiamo terminato. Altrimenti, scambiare $A(i)$ con $A(j)$.
  3. Richiamare ricorsivamente **heapify**(A, j).

  (Ovviamente, per certi valori di i uno o entrambi i figli potrebbero mancare, e questa situazione deve essere gestita.)

  Il costo di **heapify**(A, i) è $O(h)$, dove h è l'altezza del nodo i.

- **buildheap**(A):
  Converte l'array A in uno heap.

1. Cicla su i da $\lfloor n/2 \rfloor - 1$ a 0
2. Ad ogni iterazione richiama **heapify**(A, i).

Si può mostrare che il costo di **buildheap**(A) è $O(n)$.

Heapsort

L'algoritmo di ordinamento di array *Heapsort* si basa per l'appunto sugli heap.
Si supponga di avere un array A di n elementi da ordinare.

1. In primo luogo si converte A in un max-heap.
2. Si procede iterando sull'indice i dall'ultimo elemento (di indice $n-1$) al secondo (di indice 1) in questo modo:
   (a) Si scambia l'elemento $A(i)$ con $A(0)$;
   (b) Si converte il sottoarray $[A(0), \ldots, A(i-1)]$ in un max-heap.

Alla fine di questa procedura l'array risultante è ordinato. Perchè?

Il costo di *Heapsort*, per un array di n elementi è $O(n \log n)$.

Si richiede di implementare l'algoritmo di `heapsort` su array di interi (`int`).

Code di priorità

Una struttura dati basata sugli heap è la *coda di priorità*.

Una coda di max/min-priorità memorizza una collezione di elementi, ognuno dei quali associato a un valore detto *chiave*.

Una coda di max-priorità deve supportare efficientemente le seguenti operazioni:

- **insert**(S, x): inserisce l'elemento x nella coda S.
- **maximum**(S): restituisce l'elemento di S di chiave massima, senza estrarlo dalla coda.
- **extractmax**(S): restituisce l'elemento di S di chiave massima, estraendolo dalla coda.
- **increasekey**(S, i, k): pone k come nuovo valore della chiave dell'elemento i. Si suppone che k non sia minore della chiave attuale di i.

Queste operazioni sono implementabili efficientemente se si implementa la coda con un max-heap.

In particolare, se S è un max-heap (rispetto alle chiavi) di n elementi:

- **maximum**(S) consiste semplicemente nel leggere il valore della radice di S.
  Costo: $O(1)$.
- **extractmax**(S): si memorizza il valore della radice, poi si scambia la radice con l'ultimo elemento e si converte in heap il sottoarray $[S(0), \ldots, S(n-2)]$. Si restituisce infine il valore memorizzato. Si noti che bisogna tener traccia della nuova dimensione (decrementata di 1) della coda (e dello heap sottostante).
  Costo: $O(\log n)$.

- **increasekey**(S, i, k) (Nell'ipotesi che la chiave $k(i)$ di $S(i)$ sia tale che $k(i) \leq k$): si pone $k(S(i)) = k$; si procede scambiando l'elemento $S(i)$ con il padre fino a quando non succede che la chiave di $S(i)$ è $\geq$ della chiave del padre. In questo modo l'elemento $S(i)$ trova la sua nuova collocazione risalendo nell'albero.
  Costo: $O(\log n)$.

- **insert**(S, x): si rialloca l'array S in un array di dimensione $|S| + 1$; si pone l'elemento x in ultima posizione (vale a dire n) nel nuovo array; si richiama **increasekey**$(S, n, k(x))$.
  Costo: $O(\log n)$.

Si richiede di implementare una coda di max-priorità su interi (`int`), dove le chiavi sono gli interi stessi.