Algoritmi e Strutture Dati

Esercitazione 6

Domenico Fabio Savo

Esercitazione: alberi binari

Alberi binari:

- Gli alberi binari sono alberi in cui ogni nodo ha al più **2 figli** (che solitamente vengono chiamati rispettivamente *figlio destro* e *figlio sinistro*).
- La profondità di una nodo **X** è pari al numero di **archi** che bisogna attraversare per raggiungere **X** partendo dalla **radice dell'albero**. (→ la radice ha profondità = 0).
- Una **foglia** è un nodo **senza** figli.
- L'altezza di un albero è la massima profondità raggiunta dalle sue foglie (altezza albero = max profondità foglie).
- Un albero binario di altezza **h** è *completo* se ha $\mathbf{2^{(h+1)}-1}$ nodi.

Esercitazione: alberi binari

Problema:

Sia **A** un albero binario radicato nel nodo **r** e sia **K** un intero maggiore o uguale a zero.

1. Si chiede di scrivere la funzione

   ```
   nodiProfK(nodo r, intero K)
   ```

   che calcola il numero di nodi di **A** che si trovano a profondità **K** .

2. Analizzare il costo computazionale nel caso peggiore dell'algoritmo proposto.

Esercitazione: alberi binari

Soluzione:

```
algoritmo nodiProfK(nodo r, intero K) → intero
if ( r = null ) then return 0;
else    if (K = 0) then return 1;
        else return    nodiProfK(r.sinistro, K-1) +
                       nodiProfK(r.destro, K-1);
```

- Il **caso peggiore** si verifica quando l'albero è completo fino al livello **K**, ovvero quando ogni nodo con profondità **p < K** ha esattamente due figli. Dato che un albero binario completo di altezza **h** ha $\mathbf{2^h-1}$ nodi e dato che fino al livello **K** l'algoritmo visita tutti i nodi, l'algoritmo proposto ha complessità $\mathbf{O(2^K)}$.

Esercitazione: heap

HEAP: è un **albero binario** con le seguenti proprietà:

1) È completo fino al penultimo livello.

2) Vale la seguente regola: il valore dell'elemento in un nodo è sempre maggiore o uguale al valore degli elementi dei suoi figli.

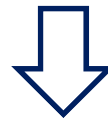
Ovvero:

$$\textbf{chiave}(v) \geq \textbf{chiave}(\textbf{sin}(v))$$
$$\textbf{chiave}(v) \geq \textbf{chiave}(\textbf{des}(v))$$

Esercitazione: heap

HEAP: è un **albero binario** con le seguenti proprietà:

1) È completo fino al penultimo livello



- Un albero è ***bilanciato in altezza*** se ad ogni nodo la differenza di altezza tra i suoi sotto alberi è in modulo al più 1.
- L'altezza di un albero bilanciato con **n** nodi è sempre **O(log n)**
- Dato che un **heap** è per definizione (condizione 1) bilanciato, allora la sua altezza sarà sempre **O(log n)**

Esercitazione: heap

Alcune volte potrebbe essere utile definire un heap "al contrario", ovvero un heap in cui vale la seguente regola: il valore dell'elemento in un nodo è sempre **minore** o uguale al valore degli elementi dei suoi figli.

Quindi:

$$\mathbf{chiave(v) \leq chiave(sin(v))}$$
$$\mathbf{chiave(v) \leq chiave(des(v))}$$

Esercitazione: heap

Chiameremo:

- **MAX-HEAP** gli heap definiti secondo la regola:

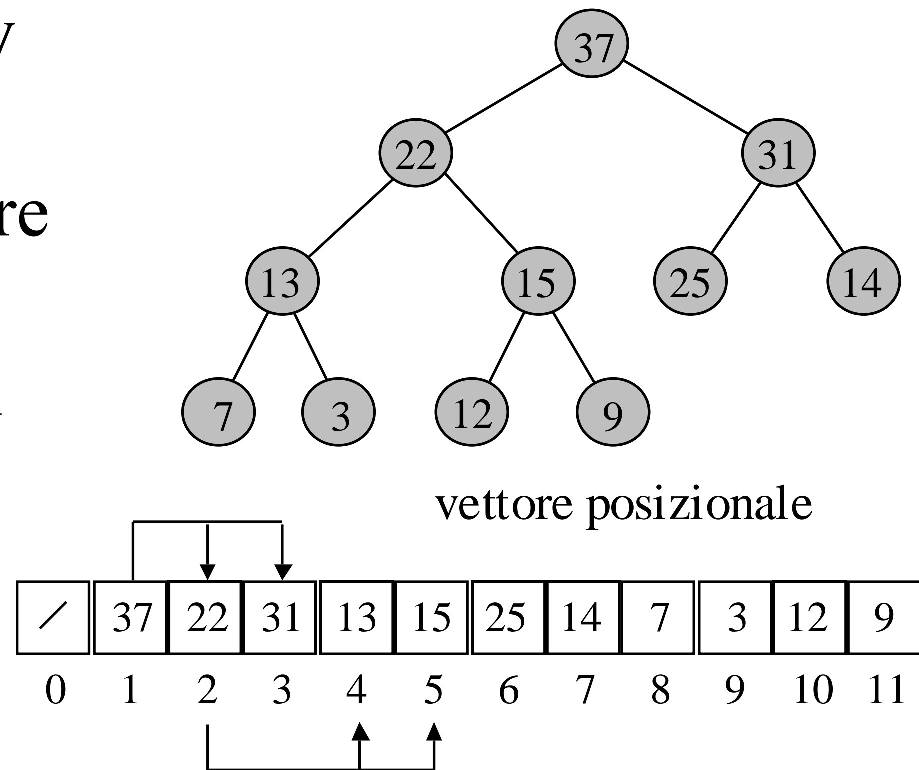
$$\mathbf{chiave(v) \geq chiave(sin(v))}$$
$$\mathbf{chiave(v) \geq chiave(des(v))}$$

- **MIN-HEAP** gli heap definiti secondo la regola:

$$\mathbf{chiave(v) \leq chiave(sin(v))}$$
$$\mathbf{chiave(v) \leq chiave(des(v))}$$

Struttura dati heap

Rappresentiamo un heap con **n** elementi mediante un array **A[1..n]**, memorizzando i valori "per livelli": se il valore del nodo **u** è memorizzato in posizione **i**, il valore dei suoi figli **sin(u)** e **des(u)** saranno memorizzati rispettivamente in posizione **2i** e **2i+1**.



Si utilizza un heap a *struttura rafforzata*: le foglie nell'ultimo livello sono compattate a sinistra

Esercitazione: heap

Esercizio:

Scrivere un algoritmo

```
isMinHeap(array A)
```

il quale, dato un array di interi **A** di dimensione **n**, termina restituendo *true* se **A** rappresenta un MIN-HEAP e ***false*** altrimenti.

Calcolare poi complessità nel caso peggiore e nel caso migliore dell'algoritmo proposto, motivando le risposte.

Esercitazione: heap

Soluzione:

```
algoritmo isMinHeap(array A) → Boolean
n ← A.size();
for(int i ← 1; i < n; i++ ) {
        if ( (i * 2) ≤ n && A[i] > A[2*i] ) then return false;
        if ( (i * 2) +1 ≤ n && A[i] > A[2*i+1] ) then return false;
}
return true;
```

Notare i controlli $(i * 2) \leq n$ e $(i * 2) + 1 \leq n$ necessari per evitare di andare fuori dall'array. Se per un nodo in posizione **i** si ha che $(i * 2) > n$ allora quel nodo è una foglia e quindi non è necessario effettuare il controllo A[i] > A[2*i]

Esercitazione: heap

Soluzione:

```
algoritmo isMinHeap(array A) → Boolean
n ← A.size();
for(int i ← 1; i < n; i++ ) {
        if ( (i * 2) ≤ n && A[i] > A[2*i] ) then return false;
        if ( (i * 2) +1 ≤ n && A[i] > A[2*i+1] ) then return false;
}
return true;
```

- Il **caso peggiore** si ha quando **A** rappresenta un MIN-HEAP. I questo caso infatti, il ciclo **for** viene eseguito interamente effettuando **n** cicli. In questo caso il costo è **O(n)**.
- Il **caso migliore** si verifica quando la radice A[1] risulta maggiore di uno dei due figli (A[2] o A[3]). In questo caso l'algoritmo termina alla prima alla prima iterazione restituendo *false*, e il costo risulta **O(1)**.