Algoritmica
Soluzione - Appello 13/1/2009

Esercizio 1 [punti 15]

È dato un albero binario di ricerca; per ciascun nodo vi è un campo aggiuntivo che contiene il numero complessivo di nodi del sottoalbero di cui il nodo è radice (nodo compreso).

1. Descrivere a parole come si possono modificare le procedure di inserzione e cancellazione per mantenere anche questa informazione e realizzarle con la stessa complessità.
2. Dare il codice di un algoritmo SELECT che selezioni il k-esimo valore tra quelli contenuti nei nodi dell'albero e darne la complessità.
3. Dare il codice di SELECT nel caso che l'albero di ricerca non abbia informazioni aggiuntive sul numero di nodi di ogni sottoalbero e darne la complessità.

Chiamiamo `nodi` il campo aggiuntivo.

1.1 La nuova procedura di inserzione dovrà incrementare di 1 il campo `nodi` di tutti gli elementi sul percorso radice-punto d'inserzione, che sono gli unici influenzati. Questa operazione potrà essere fatta durante la discesa, nel caso che l'inserzione venga fatta sicuramente, oppure a inserzione avvenuta percorrendo a ritroso col campo `padre.` In tutti e due i casi l'operazione costa O(h). Simmetricamente, nel caso della cancellazione dovrà essere decrementato il campo `nodi` di tutti gli elementi sul percorso radice-nodo da cancellare oppure radice-successore nodo da cancellare.

1.2
```
Select1(r, k):
  if (r!= Null) {
     if (r.nodi < k ) print "Non esiste";
         else {
         if (r.sx != null) dim = r.sx.nodi+1;
             else dim= 1;
         if (k < dim) Select1(r.sx, k);
         if (k == dim) Print (r.dato);
         if (k > dim) Select1 (r.dx, k-dim);}
```

Complessità O(h)

1.3 Inizialmente dim=0;

```
Select(r, k, dim) {
if (r == null) return dim;
dim = 1 + select(r.sx, k, dim);
if (dim == k)  print r.dato;
if (dim >= k) return dim;
else return select(r.dx, k, dim);
}
```

Complessità O(h + k).

Esercizio 2 **[punti 10]**

Considerare il seguente algoritmo di tipo Divide et Impera che viene richiamato la prima volta come BOH! (A, 0, n) e che opera su un array A di n interi.

```
BOH!(A,sx,dx):
IF (sx > dx) RETURN TRUE;
IF (sx = = dx) {
```

```
IF ((A[sx] % 2)== 0) RETURN TRUE;
     ELSE RETURN FALSE;}
cx = (sx + dx)/2;
ps = BOH!(A, sx, cx);
pd = BOH!(A, cx+1, dx);
IF   (ps == pd) RETURN TRUE;
     ELSE RETURN FALSE;
```

Dire cosa calcola BOH!

L'algoritmo restituisce TRUE se la somma di tutti gli elementi dell'array A è pari o dispari, poiché la somma di due elementi pari è pari e la somma di due elementi dispari è sempre pari. Ovvero se il numero di elementi dispari di A è pari oppure dispari.

Esercizio 3 **[punti 7]**

Spiegare brevemente perché il seguente algoritmo ricorsivo che calcola l'ennesimo numero di Fibonacci è tremendamente inefficiente.

```
FIB(n):
IF (n = = 0) RETURN 0;
IF (n = = 1) RETURN 1;
     ELSE
     RETURN (FIB(n-1) + FIB(n-2));
```

L'algoritmo FIB è inefficiente perché si richiama ricorsivamente 2 volte, su n-1 e n-2, senza tener conto che per trovare l'(n-1)-esimo numero di Fibonacci bisogna aver già trovato l'(n-2)-esimo. Dunque tutti i calcoli relativi alla seconda chiamata ricorsiva sono inutili perché già fatti. La complessità è $O(2^n)$, inutilmente esponenziale, esiste infatti un algoritmo che risolve lo stesso problema in O(n).