ESERCIZI (da temi d'esame)

Esercizio 1.
Si consideri un array S di n chiavi intere.

- Si dia il codice di un algoritmo che con un'unica scansione di S conti il numero r di chiavi distinte in S. Usare un dizionario D inizialmente vuoto (non interessa l'implementazione di D).
- Facendo l'assunzione che D sia implementato come un array ordinato, si analizzi la complessità in funzione di n e del numero r di chiavi distinte.

Esercizio 2.
Sia dato un array S di n interi di valore non limitato, ma che possono assumere solo $\lfloor \log n \rfloor$ valori distinti:

Esempio: $S = \langle 349; 12; 12; 102; 349; 12; 102; 102 \rangle$

Progettare un algoritmo di ordinamento che operi in tempo minore di $O(n \log n)$. Spiegare dettagliatamente l'analisi della complessità.

Esercizio 3.
Sia T un albero binario di ricerca che implementa un dizionario. Sia v un nodo di T, e sia T_v il sottoalbero con radice v.

- Si progetti un algoritmo efficiente `countLE(v, k)` che, ricevuto in input un nodo $v \in T$ e una chiave k restituisca il numero di elementi in T_v con chiave **minore o uguale a** k.
- Analizzare la complessità dell'algoritmo.

Esercizio 4.
Un albero binario *proprio* è un albero in cui ogni nodo interno ha esattamente due figli. Sia T un albero binario proprio. Dato un nodo $v \in T$ si definisca `imbalance(v)` la differenza in valore assoluto tra il numero di foglie nei sottoalberi sinistro e destro di v (se v è una foglia `imbalance(v)=0`). Si definisca anche `imbalance(T)` $= max_{v \in T}$ `imbalance(v)`.

- Dimostrare un limite superiore all'imbalance di un albero binario proprio con n nodi, e descrivere un albero il cui imbalance raggiunge tale limite.
- Disegnare un albero binario proprio T in cui `imbalance(T)` = `imbalance(v)` e v non è la radice dellalbero.
- Si progetti un algoritmo efficiente per determinare `imbalance(T)` e analizzarne la complessità in tempo.

Esercizio 5.
Dato un albero binario T, una catena sinistra di T è una sequenza di r nodi ($r \geq 1$) legati uno all'altro dal puntatore sinistro. Una catena massimale sinistra è una catena che non è contenuta in nessun'altra catena sinistra. Detto L_T il numero di catene massimali sinistre

1. Indicare le catene massimali su un albero binario completamente bilanciato di altezza 4.
2. Dimostrare che se T è completamente bilanciato e ha $n = 2^k - 1$ nodi, $L_T = 2^{k-1}$.
3. Si definisca un algoritmo efficiente che calcoli il numero di catene massimali sinistre L_T per un qualsiasi albero binario T.

Esercizio 1

(1) ctr = contatore del numero di chiavi distinte

ChiaviDistinte(S)

```
D = nuovo Dizionario();
ctr = 0;
for (i=0; i<n; i++) {
    if (!Appartiene(D, S[i])) {
        Inserisci(D, S[i]);
        ctr++;
    }
}
return ctr;
```

(2) D contiene solo gli r elementi distinti. Se D è implementato con un array ordinato, ogni verifica di appartenenza costa $O(\log r)$ [→ ricerca binaria] e ogni inserimento costa $O(r)$ [→ per mantenere l'array ordinato]. Dato che le verifiche di appartenenza sono n e gli inserimenti sono r, il costo complessivo è $O(n \log r + r^2)$.

Esercizio 2

Si utilizza un albero AVL in cui si memorizzano gli interi in S.
Ogni nodo dell'albero ha i campi
u.altezza
u.sx
u.dx
u.dato.chiave ← elemento dell'array S
u.dato.sat ← # di occorrenze dell'elemento nell'array S.

Prima di inserire un elemento dell'array S nell'albero si controlla se tale elemento è già presente. In questo caso NON si effettua l'inserimento, ma si incrementa il campo dato.sat.

L'array S si ordina visitando l'albero AVL in ordine simmetrico.

OSSERVAZIONE

La dimensione dell'albero è $\Theta(\log n)$ in quanto si inseriscono solo le chiavi distinte.
Dunque, l'altezza dell'albero è $O(\log \log n)$.

Ordina(S)

```
T = nuovo albero AVL;
for (i=0; i<n; i++) {
   f = Ricerca (T.radice, S[i]);
   if (f ≠ null) f.sat++;
   else { e = nuovoElemento;
          e.chiave = S[i];
          e.sat = 1;
          Inserisci (T.radice, e);
   }
}
VisitaSimmetrica (T.radice, S, 0);
return S;
```

$O(\log \log n)$ (corpo del ciclo); $O(n \log \log n)$ (ciclo for); $O(n)$ (VisitaSimmetrica)

VisitaSimmetrica (u, S, i)

```
if (u == null) return i;
k = VisitaSimmetrica (u.sx, S, i);
for (j=0; j< u.dato.sat; j++) {
     S[k] = u.dato.chiave;
     k++;
}
return VisitaSimmetrica (u.dx, S, k);
```

Analisi

Il corpo del ciclo for, che è ripetuto n volte, costa $O(\log\log n)$, in quanto l'albero AVL ha altezza $O(\log\log n)$.

↳ Ricerca e Inserimento costano $O(\log\log n)$

La visita dell'albero ha un costo complessivo ~~di~~ $\Theta(n)$.

$$\Rightarrow T(n) = O(n \log\log n + n)$$
$$= O(n \log\log n)$$
$$= o(n \log n)$$

Soluzione alternativa

Si modifica l'algoritmo di inserimento negli AVL:

AggiornaAVL(u, k)

```
if (u == null) {
    e = nuovo Elemento;
    e.chiave = k;
    e.sat = 1;
    f = nuovo Nodo();
    f.dato = e;
    f.altezza = 0;
    f.sx = f.dx = null;
    return f;
}
if ( k < u.dato.chiave ) {
    u.sx = AggiornaAVL(u.sx, k);
    if (u è critico) * esegui la rotazione *;
}
else if ( k > u.dato.chiave ) {
    u.dx = AggiornaAVL(u.dx, k);
    if (u è critico) * esegui la rotazione *;
}
else // k = u.dato.chiave
    u.dato.sat ++;
u.altezza = max {Altezza(u.sx), Altezza(u.dx)} + 1;
return u;
```

(blocco iniziale: $O(1)$)

Sort(S)

```
T = nuovo albero AVL;
for (i = 0; i < n; i++)
    AggiornaAVL(T.radice, S[i]);   } O(loglog n)
VisitaSimmetrica(T.radice, S, 0);   O(n)
return S;
```

$O(n \log\log n)$

$T(n) = O(n \cdot \log\log n)$

Esercizio 3

```
CountLE(v, k)
    if (v == null) return 0;
    q = v.dato.chiave;
    if (k < q) return CountLE(v.sx, k);
    else return 1 + CountLE(v.sx, k) + CountLE(v.dx, k);
```

Analisi

Si noti che se tutte le entry in T_v hanno chiave $\leq k$ allora l'algoritmo non fa altro che eseguire una visita del sottoalbero T_v e quindi la sua complessità è $O(m)$, dove m è il numero di entry in T_v. Nel caso generale, sia h l'altezza di T_v ed s il numero di entry in T_v con chiave $\leq k$. Se la chiave q della entry nel nodo v è $> k$, allora l'algoritmo esegue un numero di operazioni costanti e richiama se stesso solo sul figlio sinistro di v. Se $q \leq k$ l'algoritmo esegue un numero di operazioni costanti, richiama se stesso sentrambi i figli di v. Il figlio sinistro è radice di un sottoalbero che contiene solo entry con chiave $\leq k$, e che verrà quindi visitato completamente. Possiamo quindi concludere che le varie chiamate ricorsive percorrono una *dorsale* da v a una foglia: per ogni nodo u della dorsale vengono eseguite $O(1)$ operazioni e, se u contiene una entry con chiave $\leq k$, allora tutto il suo sottoalbero sinistro, che contiene solo entry con chiave $\leq k$, viene visitato in tempo proporzionale al numero di tali entry e la dorsale prosegue nel figlio destro, mentre se u contiene una entry con chiave $> k$ la dorsale prosegue nel figlio sinistro e il figlio destro non viene toccato. Il lavoro fatto nei nodi della dorsale richiede complessivamente tempo $O(h)$, mentre le visite complete dei sottoalberi a sinistra della dorsale richiedono, complessivamente, tempo $O(s)$. La complessità dell'algoritmo è quindi $O(h+s)$.

Esercizio 4

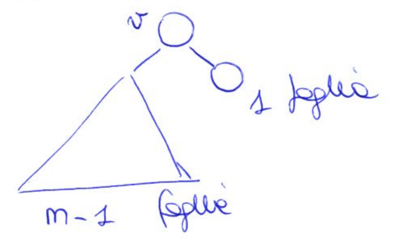
1) Un albero binario proprio di n nodi contiene esattamente

$$m = \frac{n+1}{2}$$

foglie

(si può dimostrare per induzione)

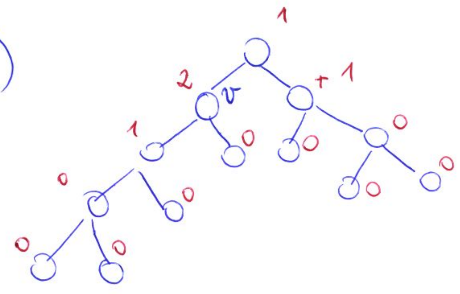
L'albero con ~~max~~ imbalance massimo è:



$$\Rightarrow \text{Imbalance}(v) = (m-1) - 1 = m-2$$

$$= \frac{n+1}{2} - 2 = \frac{n-3}{2}$$

2)



Imbalance $(T) = 2 =$
$=$ Imbalance (v).

3)

Imbalance (u) // restituisce la coppia di valori
<Imbalance T(u), Foglie T(u)>

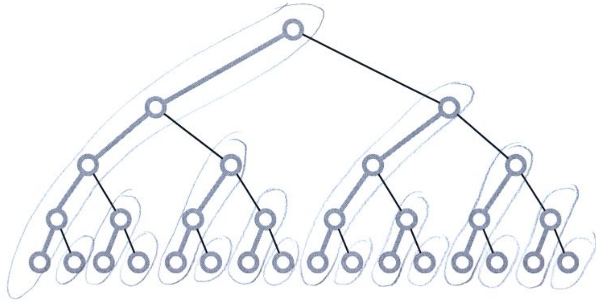
```
if (u==null) return <0, 0>;
if (u.sx==null && u.dx==null) return <0, 1>;

<ImSx, FoglieSx> = Imbalance (u.sx);
<ImDx, FoglieDx> = Imbalance (u.dx);

Imb = max { ImSx, ImDx, |FoglieSx - FoglieDx| }
                        (Imbalance del nodo u)

foglie = foglieSx + FoglieDx;
return <Imb, foglie>.
```

$$T(n) = \Theta(n)$$

(la struttura dell'algoritmo è quella di una visita posticipata).

Esercizio 5

18



- Per induzione.

  Caso base. Per $k = 1$, l'albero contiene un solo nodo e una catena massimale: $n = 2^1 - 1 = 1$, $L_T = 1 = 2^{1-1}$.

  Passo. Sia T un albero completamente bilanciato con $n = 2^k - 1$ nodi. T è composto dal nodo radice, e da due sottoalberi T_{sx} e T_{dx} completamenti bilanciati di $2^{k-1} - 1$ nodi ciascuno. Applicando l'ipotesi induttiva sui sottoalberi, e tenendo presente che il numero di catene massimali di T è dato dalla somma delle catene massimali dei suoi sottoalberi (infatti la radice di T estende una catena massimale di T_{sx} e non dà origine a nuove catene massimali) si ottiene $L_T = L_{T_{sx}} + L_{T_{dx}} = 2^{(k-1)-1} + 2^{(k-1)-1} = 2^{k-1}$.

- **Catene(u)**

```
if (u == NULL) return 0;
if (u.sx == NULL && u.dx == NULL) return 1;
if (u.sx == NULL) return 1 + Catene(u.dx);
else return Catene(u.sx) + Catene(u.dx);
```

  Complessità: $T(n) = O(n)$