ESERCIZIO 1

(1) ctr = contatore del numero di chiavi distinte

ChiaviDistinte(S)

```
D = nuovo dizionario
ctr = 0
for i = 1 to n {
    if ( Search(D, S[i]) == NIL ) {
        Insert(D, S[i])
        ctr++
    }
}
return ctr
```

(2) D contiene solo gli r elementi distinti.
Se D è implementato con un array ordinato ogni esecuzione di Search... costa $O(\log r)$ [→ ricerca binaria] e ogni inserimento costa $O(r)$ [→ per mantenere ordinato l'array].
Dato che le esecuzioni della ricerca 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.left
- u.right
- u.key
- u.occ ← # di occorrenze della chiave 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 occ.

L'array S' si ordina visitando l'albero 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 = 1; i ≤ n; i++) {
    u = Search(T.root, S[i])
    if (u ≠ NIL)  u.occ ++;
    else {
        v = nuovo Nodo
        v.key = S[i]
        v.occ = 1
        v.left = v.right = NIL
        Insert(T, v)
    }
}
VisitaSimmetrica(T.root, S, 1);   Θ(n)
return S
```

O(log log n)

O(n log log n)

VisitaSimmetrica (u, S, i)

```
if (u == NIL) return i
k = VisitaSimmetrica(u.left, S, i)
for (j = 1; j ≤ u.occ; j++) {
        S[k] = u.key
        k++
}
return VisitaSimmetrica(u.right, 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 $\Theta(n)$

$\Rightarrow$

$$T(n) = O(n \log\log n + n)$$
$$= O(n \log\log n)$$
$$= \Theta(n \log n)$$

Esercizio 3

```
CountLE (v, k)
if (v == NIL) return 0;
q = v.key
if (k < q) return CountLE(v.left, k)
else return 1 + CountLE(v.left, k) + CountLE(v.right, 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

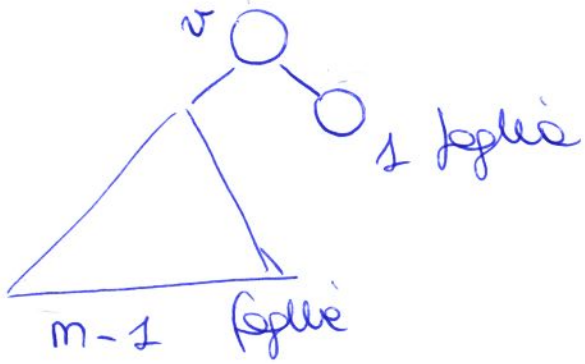
1) Un albero binario proprio di n nodi contiene esattamente

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

foglie

(si può dimostrare per induzione)

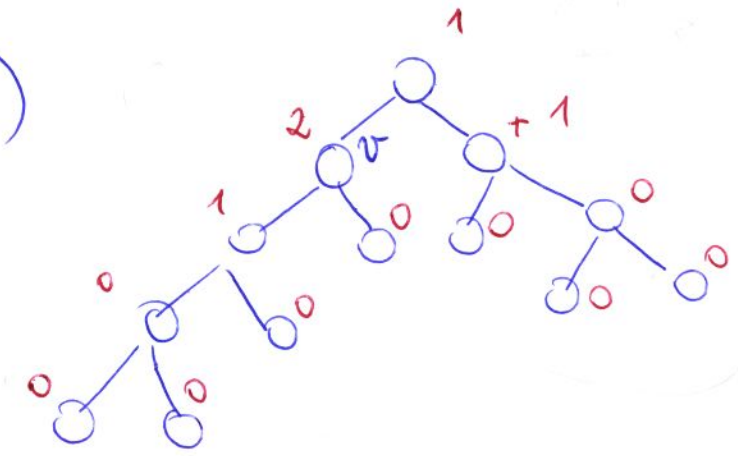
L'albero con ~~max~~ imbalance massimo è:



$\Rightarrow$ Imbalance $(v) = (n-1) - 1 = n-2$

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

2)



Imbalance $(T) = 2 =$
$= \text{Imbalance}(v)$.

3) L'algoritmo Imbalance(u) restituisce una coppia di interi

$< imb ; foglie >$

imb: imbalance dell'albero radicato in u

$foglie$: # foglie dell'albero radicato in u.

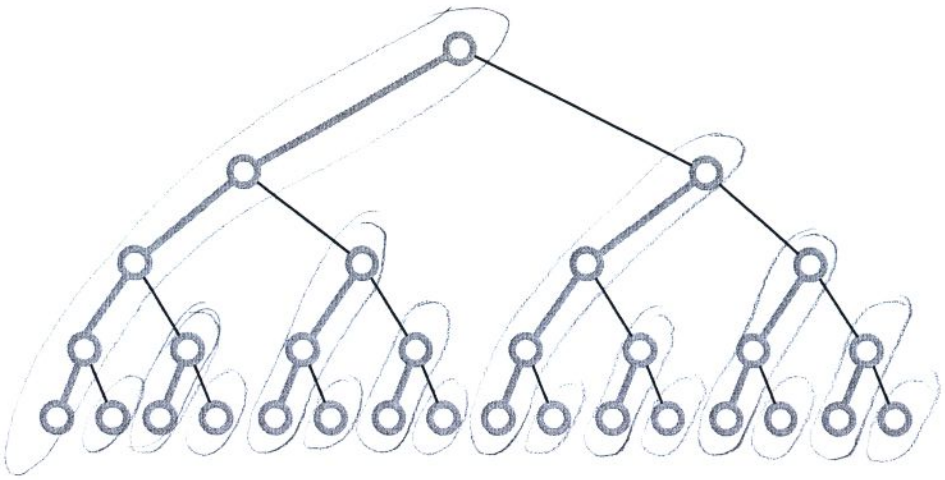
```
Imbalance(u)
if (u == NIL) return <0, 0>
if (u.left == NIL && u.right == NIL) return <0, 1>
<imbSx, fSx> = Imbalance(u.left)
<imbDx, fDx> = Imbalance(u.right)
imb = max{imbSx, imbDx, |fSx - fDx|}
foglie = fSx + fDx
return <imb, foglie>
```

$|fSx - fDx|$: imbalance del nodo u

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

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

Esercizio 3



- 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==NIL) return 0;
if(u.left==NIL && u.right==NIL) return 1;
if(u.left==NIL) return 1 + Catene(u.right);
else return Catene(u.left) + Catene(u.right);
```

Complessità: $T(n) = \Theta(n)$