```c
#include <stdio.h>
#include <stdlib.h>

/* Nodo di albero */
typedef struct _node {
    int key;
    struct _node* left;
    struct _node* right;
} node;

/* Inserimento ricorsivo nell'ABR T della chiave val.
*
* Tempo: O(h) = O(n) al caso pessimo per gli ABR non bilanciati
*             = O(log n) al caso medio per gli ABR non bilanciati
*             = O(log n) al caso pessimo per gli ABR bilanciati (AVL)
*/
node* ABR_insert(node* T, int val) {
    // Caso base: se l'albero è vuoto creiamo un nuovo nodo e lo restituiamo
    if(T == NULL) {
        node* newLeaf = (node*) malloc(sizeof(node));
        newLeaf->key = val;
        newLeaf->left = NULL;
        newLeaf->right = NULL;
        return newLeaf;
    }

    /* Inserimento ricorsivo:
     * se la chiave da inserire è minore o uguale a quella del nodo corrente
     * proseguiamo l'inserimento nel sottoalbero sinistro
     * altrimenti nel sottoalbero destro.
     */
    if(val <= T->key)
        T->left = ABR_insert(T->left, val);
    else
        T->right = ABR_insert(T->right, val);

    return T; // Restituiamo l'albero aggiornato
}

/* Ricerca di val all'interno dell'ABR puntato da root.
* Restituisce 1 se la chiave è presente, altrimenti 0.
* Nella variabile puntata da depth viene salvata la profondità del nodo trovato.
*
* Tempo: O(h) = O(n) al caso pessimo per gli ABR non bilanciati
*             = O(log n) al caso medio per gli ABR non bilanciati
*             = O(log n) al caso pessimo per gli ABR bilanciati (AVL)
*/
int ABR_find(node* root, int val, int* depth) {
    if(root == NULL) // Caso base ricorsione: albero vuoto, chiave non presente
        return 0;

    if(root->key == val) // Altrimenti, se questo nodo contiene la chiave
cercata...
        return 1;
```

```c
    /* Se questo nodo non contiene la chiave cercata
     * dobbiamo proseguire la ricerca a sinistra o a destra.
     * Così facendo scendiamo di un livello, e quindi aumentiamo la profondità.
     */
    (*depth)++;

    if(val < root->key)
        return ABR_find(root->left, val, depth);
    else
        return ABR_find(root->right, val, depth);
}

/* Funzione di utilità per deallocare un albero.
* Segue la struttura di una postvisita, deallocando prima i sottoalberi
* e poi il nodo corrente.
*/
void ABR_free(node* root) {
    if(root != NULL) {
        ABR_free(root->left);
        ABR_free(root->right);

        free(root);
    }
}

/* Funzione di utilità per il valore assoluto di un intero.
*
* In alternativa si può usare una macro come questa:
#define abs(n) ((n >= 0) ? (n) : (-n))
*/
int abs(int n) {
    if(n < 0) return -n;
    else return n;
}

/* Funzione di utilità che calcola l'altezza di un albero.
* Per convenzione height(NULL) = -1.
*
* Tempo = Theta(n)
*/
int height(node* root) {
    if(root == NULL)
        return -1;

    int hsx = height(root->left);
    int hdx = height(root->right);

    // height(root) = 1 + max(height(root.left), height(root.right))
    if(hsx > hdx)
        return 1 + hsx;
    else
        return 1 + hdx;
}
```

```c
/* Controlla se l'albero radicato in root è 1-bilanciato. (versione inefficiente)
*
* Dalla teoria sappiamo che un albero è 1-bilanciato se e solo se ogni nodo
* ha coefficiente di bilanciamento minore o uguale a 1, in valore assoluto.
* Ricordiamo che il coefficiente di bilanciamento è definito come
* u(v) = altezza(v.left) - altezza(v.right)
*
*
* Restituisce 0 se l'albero non è 1-bilanciato, altrimenti 1.
*
* Tempo = Theta(n^2) (inefficiente)
*/

int isOneBalanced_ineff(node* root) {
    if(root == NULL)
        return 1;

    int hsx = height(root->left);
    int hdx = height(root->right);

    int balance = abs(hsx - hdx);
    if(balance > 1)
        return 0;
    else
        return isOneBalanced_ineff(root->left) && isOneBalanced_ineff(root-
>right);
}


/* Controlla se l'albero radicato in root è 1-bilanciato.
*
* Dalla teoria sappiamo che un albero è 1-bilanciato se e solo se ogni nodo
* ha coefficiente di bilanciamento minore o uguale a 1, in valore assoluto.
* Ricordiamo che il coefficiente di bilanciamento è definito come
* u(v) = altezza(v.left) - altezza(v.right)
*
* Restituisce 0 se l'albero non è 1-bilanciato, altrimenti 1.
* Nella variabile puntata da height viene salvata l'altezza di root.
*
* Tempo = Theta(n) (ottimo)
*
* Questa funzione usa la convenzione per cui altezza(NULL) = -1.
*/
int isOneBalanced(node* root, int *height) {
    /* Se l'albero è vuoto la sua altezza è -1 ed è banalmente 1-bilanciato. */
    if(root == NULL) {
        *height = -1;
        return 1;
    }

    int hsx = -1, hdx = -1;
    /* Otteniamo ricorsivamente la risposta alla domanda per il sottoalbero
     * sinistro e per quello destro, insieme alle rispettive altezze nelle
variabili
     * hsx e hdx.
```

```
     */
    int isBalanced_left = isOneBalanced(root->left, &hsx);
    int isBalanced_right = isOneBalanced(root->right, &hdx);

    int balanced;
    /* Il nodo attuale è 1-bilanciato se:
     * 1. il sottoalbero sinistro è bilanciato
     * 2. il sottoalbero destro è bilanciato
     * 3. il suo coefficiente di bilanciamento è |u| <= 1
     */
    if(isBalanced_left && isBalanced_right && (abs(hsx - hdx) <= 1))
        balanced = 1;
    else
        balanced = 0;

    // Calcoliamo l'altezza di questo nodo e la salviamo nel puntatore height
    // per restituirla al chiamante
    if(hsx > hdx)
        *height = 1 + hsx;
    else
        *height = 1 + hdx;

    return balanced;
}

int main(void) {
    int i;
    int n;
    int num;

    scanf("%d", &n); // Numero di nodi da leggere

    node* T = NULL; // Inizialmente l'albero è vuoto

    // Inserimento dei valori letti da input nell'albero
    for(i=0;i<n;i++) {
        scanf("%d", &num);
        T = ABR_insert(T, num);
    }

    /* ********************************************************************
     * Inizio parte in più rispetto all'esercizio 1 sulla piattaforma:
     * stampa 1 se l'albero fornito in input è un AVL, altrimenti 0.
     * Inoltre, ne stampa l'altezza.
     */
    int altezza;

    int avl = isOneBalanced(T, &altezza);

    printf("AVL: %d, altezza: %d\n", avl, altezza);
    /* Fine parte in più rispetto all'esercizio 1 della piattaforma
     ********************************************************************
     */

    /* Ricerca dei valori forniti in input */
```

```c
    int search = 0;

    while(search >= 0) {
        scanf("%d", &search);

        if(search < 0)
            break;

        int depth = 0; // La profondità minima è zero

        if(ABR_find(T, search, &depth))
            printf("%d\n", depth); // Stampa la profondità, se il valore è
presente
        else
            printf("NO\n");
    }

    /* Deallocazione dell'albero */
    ABR_free(T);

    return 0;
}
```