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

#define max(x, y) ((x) > (y) ? (x) : (y))

typedef struct _node {
    int key;
    int height;
    struct _node *left;
    struct _node *right;
} node;

// Inserimento in un albero binario non bilanciato.
// Ai fine delle soluzione 2, contestualmente, calcola e memorizza
// l'altezza dei nodi.
node *insert(node *root, int key) {
    if (root == NULL) {
        node *new = malloc(sizeof(node));
        new->key = key;
        new->left = NULL;
        new->right = NULL;
        new->height = 0;
        return new;
    }

    if (key <= root->key) {
        root->left = insert(root->left, key);
    } else {
        root->right = insert(root->right, key);
    }

    int left_height = root->left == NULL ? -1 : root->left->height;
    int right_height = root->right == NULL ? -1 : root->right->height;
    root->height = max(left_height, right_height) + 1;

    return root;
}

// Calcola ricorsivamente l'altezza di un nodo. Richiede costo proporzionale alla
dimensione dell'albero.
int height(const node *root) {
    if (root == NULL) {
        return -1;
    }

    if (root->left == NULL && root->right == NULL) {
        return 0;
    }

    return 1 + max(height(root->left), height(root->right));
}

// Primo approccio risolutivo, seguiamo la definizione di 1-bilanciato,
// calcolando quando necessario l'altezza.
```

```c
// Questa soluzione ha complessità O(n^2) poiché ad ogni passo utilizziamo la
funzione height che ha costo O(n).
int one_balanced1(const node *root) {
    if (root == NULL) {
        return 1;
    }

    int left_height = height(root->left);
    int right_height = height(root->right);
    // Il nodo radice deve essere bilanciato
    int is_balanced = abs(left_height - right_height) <= 1;
    // I due sottoalberi devono essere bilanciati
    int subtree_balanced = one_balanced1(root->left) && one_balanced1(root-
>right);

    return is_balanced && subtree_balanced;
}

// Questa seconda soluzione segue lo schema della prima soluzione, ma sfrutta
// il campo height che contiene l'altezza del nodo. Height viene precomputato
// in fase di inserimento. Questo ci permette di ridurre il costo computazionale a
O(n).
int one_balanced2(const node *root) {
    if (root == NULL) {
        return 1;
    }

    // È necessario controllare che i sottoalberi non siano vuoti
    int left_height = root->left == NULL ? -1 : root->left->height;
    int right_height = root->right == NULL ? -1 : root->right->height;
    int is_balanced = abs(left_height - right_height) <= 1;
    int subtree_balanced = one_balanced1(root->left) && one_balanced1(root-
>right);

    return is_balanced && subtree_balanced;
}

struct result {
    int is_balanced;
    int height;
};

// Questa soluzione assume che non possiamo precalcolare l'altezza e durante
// il passo ricorsivo calcola contemporaneamente sia il bilanciamento sia
l'altezza,
// per questo motivo ritorniamo non un singolo valore ma una coppia di valori.
// Questa soluzione ad ogni passo, calcola la profondità che abbiamo raggiunto
// che viene passata come parametro della funzione. La prima chiamata richiede di
passare depth come -1
// Esempio: one_balanced3(root, -1)
struct result one_balanced3(const node *root, int depth) {
    if (root == NULL) {
        struct result res = { 1, -1 };
        return res;
    }
```

```c
    struct result left = one_balanced3(root->left, depth + 1);
    struct result right = one_balanced3(root->right, depth + 1);

    int is_balanced = abs(left.height - right.height) <= 1;
    int subtree_balanced = left.is_balanced && right.is_balanced;

    struct result res = { is_balanced && subtree_balanced, max(left.height,
right.height) + 1 };
    return res;
}

// Funzione di supporto nel caso volete solo ritornare il bilanciamento
// e volete evitare di passare manualmente il parametro depth iniziale.
int one_balanced(const node *root) {
    return one_balanced3(root, -1).is_balanced;
}

int main(int argc, char *argv[]) {
    int k, n, d;
    node *root = NULL;

    scanf("%d", &n);
    for (size_t i = 0; i < n; i++) {
        scanf("%d", &k);
        root = insert(root, k);
    }

    printf("%d\n", one_balanced(root));

    return 0;
}
```