Prova_parziale 2016_01_20 soluzione

Dato un albero generico i cui nodi hanno attributi **key, left-child, right-sib**, scrivere una funzione C che restituisce il numero di nodi interni i cui figli hanno tutti la stessa chiave.
Qual è la complessità della funzione?

```
/*
* La complessità di questo codice risulta essere:
* - T(n) = 2T(n/2) + O(1)
* - Si ricade nel primo caso del teorema master,
*   dove n^1 > O(1), facendo si che T(n) = \Theta(n)
*/


#include <iostream>
using namespace std;

// struttura del nodo
struct Node {
        int key;
        Node* left_child;
        Node* right_sib;
        Node(int val): key(val), left_child(nullptr), right_sib(nullptr) {}
};

// potrebbe essere ottimizzata riducendo gli accessi al left_child
int checkSameKey(Node* u, int& value){
        if(u == nullptr) return 0; //O(1)
        if(u->key == value) // sempre T(n/2) sia if che else
                return 1 + checkSameKey(u->left_child, value) + checkSameKey(u->right_sib,
value);
        else
                return 0 + checkSameKey(u->left_child, value) + checkSameKey(u->right_sib,
value);
}

int main(){
        Node* root = new Node(10);
        root->left_child = new Node(5);
        root->left_child->left_child = new Node(4);
        root->left_child->left_child->left_child = new Node(10);
        root->left_child->left_child->right_sib = new Node(10);
        root->left_child->left_child->right_sib->right_sib = new Node(2);

        int val = 10;
        cout << checkSameKey(root, 10) << endl;
        return 0;
}
```

Sia T un albero binario di ricerca di altezza h e avente n nodi con chiavi intere eventualmente ripetute. Si progetti un algoritmo **efficiente** che, ricevuto in ingresso T e un intero k, conta il numero di occorrenze di k in T.
Analizzare la complessità dell'algoritmo.

```
#include <iostream>
using namespace std;
```

```
struct Node {
        int key;
        Node* left;
        Node* right;
        Node(int val): key(val), left(nullptr), right(nullptr), {}
}

int checkValue(Node* u, int value){
        if(u == nullptr) return 0; // O(1)
        if(value == u->key) // entrambi T(n/2)
                return 1 + checkValue(u->left, value) + checkValue(u->right, value);
        else
                return 0 + checkValue(u->left, value) + checkValue(u->right, value);
}
// complexity checkValue = T(n) = 2T(n/2) + O(1) = \Theta(n)


int main(){
        Node* root = new Node(10);
        root->left = new Node(5);
        root->right = new Node(7);
        root->left->left = new Node(10);
        root->left->right = new Node(8);
        root->right->right = new Node(5);
        cout << checkValue(root, 10) << endl;
        return 0;
}
```

Si definiscano formalmente le relazioni O, Ω, Θ, o, ω e si dimostri la verità o la falsità di ciascuna delle seguenti affermazioni, giustificando formalmente le risposte:

a) Se $P(n)$ è un polinomio di grado k, allora $P(n) = \Theta(n^k)$

b) $n = O(n \log \log n)$

c) $n \log \log n = O(n^{1+\varepsilon})$, per ogni $\varepsilon > 0$

d) $f(n) = O(g(n))$ se e solo se $g(n) = \Omega(f(n))$

e) $\omega(f(n)) \cap O(g(n)) = \varnothing$

- Definizione formale delle relazioni:
  - O
    - Esiste una funzione $f(n)$ tale che, per n abbastanza grandi, se abbiamo $O(g(n))$ allora $0 \leq f(n) \leq cg(n)$ dove c è una costante intera positiva
  - Ω
    - Esiste una funzione $f(n)$ tale che, per n abbastanza grandi, se abbiamo $\Omega(g(n))$ allora $0 \leq cg(n) \leq f(n)$ dove c è una costante intera positiva
  - Θ
    - Esiste una funzione $f(n)$ tale che, per n abbastanza grandi, se abbiamo $O(g(n))$ allora $0 \leq f(n) \leq cg(n)$ dove c è una costante intera positiva
  - o
    - Esiste una funzione $f(n)$ tale che, per n abbastanza grandi, se abbiamo $\Theta(g(n))$ allora $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ dove c_1 e c_2 sono due costanti intere positive
  - ω
    - Esiste una funzione $f(n)$ tale che, per n abbastanza grandi, se abbiamo $\omega(g(n))$ allora $0 \leq f(n) \leq cg(n)$ per ogni $c > 0$

- a) Essendo $P(n)$ un polinomio possiamo pensarlo come $a_k n^k$ come elemeno rilevante.
  - $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, uguale ad $0 \leq c_1 n^k \leq a_k n^k \leq c_2 n^k$ quindi possiamo dare come rispettivo valori a c_1 e c_2 un valore che sia maggiore/minore di a_k e la funzione è verificata
- b) In questo caso abbiamo una O, quindi $0 \leq f(n) \leq cg(n)$
  - Semplificando viene $1 \leq log(log(n))c$ ipotizzando poi un $n \geq 2$
  - Sappiamo che un logaritmo per essere positivo deve far si che $n \geq 2$
  - Da qui rimane solo trovare un $c \geq loglog(n)$ per $n \geq 2$
- c) In questa equazione abbiamo che:
  - Semplificando per n si ha $loglog(n) \leq cn^\epsilon$
  - Ipotizziamo un $c = 1$
  - n^ϵ sarà sempre più grande essendo un esponenziale
- d) L'affermazione risulta essere vera, essendoci entrambe le condizioni per Big o e Omega rispettate
- e) L'affermazione risulta essere falsa nel momento in cui, per esempio, la funzione $f(n)$ cresce uguale a quella di $g(n)$.