Algoritmica – Esame di Laboratorio

26/05/2020

Istruzioni

Risolvete il seguente esercizio prestando particolare attenzione alla formattazione dell'input e dell'output. La correzione avverrà in maniera automatica eseguendo dei tests e confrontando l'output prodotto dalla vostra soluzione con l'output atteso. Si ricorda che è possibile verificare la correttezza del vostro programma su un sottoinsieme dei input/output utilizzati. I file di input e output per i test sono nominati secondo lo schema:
`input0.txt output0.txt`
`input1.txt output1.txt`
`...`
Per effettuare le vostre prove potete utilizzare il comando del terminale per la redirezione dell'input. Ad esempio

```
./compilato < input0.txt
```

effettua il test del vostro codice sui dati contenuti nel primo file di input, assumendo che `compilato` contenga la compilazione della vostra soluzione e che si trovi nella vostra home directory. Dovete aspettarvi che l'output coincida con quello contenuto nel file `output0.txt`. Per effettuare un controllo automatico sul primo file input `input0.txt` potete eseguire la sequenza di comandi

```
./compilato < input0.txt | diff - output0.txt
```

Questa esegue la vostra soluzione e controlla le differenze fra l'output prodotto e quello corretto.

Una volta consegnata, la vostra soluzione verrà valutata nel server di consegna utilizzando altri file di test non accessibili. Si ricorda di avvisare i docenti una volta che il server ha accettato una soluzione come corretta.

Suggerimenti

Progettare una soluzione efficiente. Prestare attenzione ad eventuali requisiti in tempo e spazio richiesti dall'esercizio. In ogni caso, valutare la complessità della soluzione proposta e accertarsi che sia *ragionevole*: difficilmente una soluzione con complessità $\Theta(n^3)$ sarà accettata se esiste una soluzione semplice ed efficiente in tempo $\mathcal{O}(n)$.

Abilitare i messaggi di diagnostica del compilatore. Compilare il codice usando le opzioni `-g -Wall` di gcc:

```
gcc -Wall -g soluzione.c -o soluzione
```

risolvere *tutti* gli eventuali *warnings* restituiti dal compilatore, in particolar modo quelli relativi alle funzioni che non restituiscono un valore e ad assegnamenti tra puntatori di tipo diverso.

Provare la propria soluzione in locale. Valutare la correttezza della soluzione sulla propria macchina accertandosi che rispetti gli input/output contenuti nel TestSet. In particolare, si consiglia di provare **tutti** gli input/output contenuti nel TestSet usando le istruzioni nella pagina precedente.

Usare `valgrind`. Nel caso in cui il programma termini in modo anomalo o non calcoli la soluzione corretta, è utile accertarsi che non acceda in modo scorretto alla memoria utilizzando `valgrind` (accertarsi di aver compilato il codice con il flag `-g`):

```
valgrind ./soluzione < input0.txt
```

`valgrind` eseguirà il vostro codice sull'input specificato (in questo caso, il file `input0.txt`), mostrando in output dei messaggi di diagnostica nei casi seguenti:

1. accesso (in lettura o scrittura) ad una zona di memoria non precedente allocata;
2. utilizzo di una variabile non inizializzata precedentemente;
3. presenza al termine dell'esecuzione del programma di zone di memoria allocate con `malloc` ma non liberate con `free` (*memory leak*).

Risolvere *tutti* i problemi ai punti 1. e 2. prima di sottoporre la soluzione al server.

Esercizio

Consideriamo un albero binario in cui ogni nodo è colorato di rosso oppure di bianco. Definiamo "*altezza rossa*" di un nodo v il numero massimo di nodi rossi in un cammino da v ad una foglia nel suo sottoalbero.

Scrivere un programma che legga da tastiera una sequenza di N interi distinti e li inserisca in un albero binario di ricerca (senza ribilanciamento) nello stesso ordine con il quale vengono forniti in input. Ad ogni intero è associato un colore, anch'esso rappresentato da un intero (0 per il rosso, 1 per il bianco). Il programma deve poi verificare se, per **ogni** nodo v dell'albero, l'altezza rossa del figlio sinistro e quella del figlio destro di v differiscono di al più 1. L'altezza rossa di un albero vuoto si considera uguale a zero; ad esempio, se un nodo non ha il figlio sinistro, l'altezza rossa del figlio sinistro è pari a zero in quanto il sottoalbero relativo è vuoto. Il programma deve stampare `TRUE` se la condizione è verificata, `FALSE` altrimenti.

NOTA: Affinché la condizione sia verificata, la proprietà deve valere per **tutti** i nodi dell'albero.

NOTA: Affinché l'esame sia superato, la complessità in tempo dell'algoritmo **deve** essere lineare nel numero N dei nodi.

NOTA: Affinché l'esame sia superato, la struct nodo **deve** contenere solo i campi: chiave, colore, puntatore al figlio sinistro e puntatore al figlio destro.

L'input è così formato:

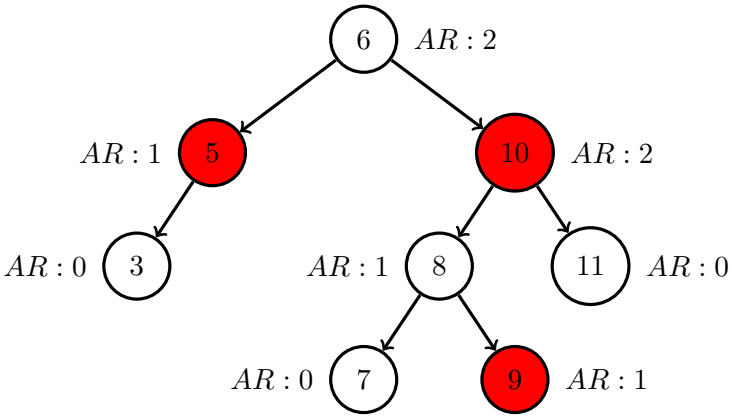
- la prima riga contiene il numero N di interi da inserire nell'albero binario di ricerca;
- le successive $2N$ righe contengono la descrizione degli N elementi da inserire nell'albero: la descrizione di un elemento occupa 2 righe consecutive:
  - la prima riga contiene l'intero;
  - la seconda riga contiene il colore: un intero uguale a 0 per il rosso e 1 per il bianco.

L'output è costituito dalla stringa `TRUE` se la condizione richiesta è verificata, dalla stringa `FALSE` altrimenti. Si ricorda che la condizione è la seguente: per ogni nodo, l'altezza rossa del figlio sinistro e quella del figlio destro differiscono di al più 1.

Esempio

Input

```
8
6
1
5
0
3
1
10
0
8
1
7
1
9
0
11
1
```



Output

```
TRUE
```

La figura dell'esempio mostra, per ogni nodo v, il valore di $AR(v)$ (l'altezza rossa di v). In tal caso, la condizione richiesta è verificata.