

Esame di Algoritmi 1 – Sperimentazioni (VC)

Prova di esempio

Testo d'Esame

Esercizio 1

Implementare un algoritmo che ritorni **il numero di nodi di un sotto-albero in un albero binario di ricerca (BST) che si trovano a una profondità pari**.

Dato un BST e una chiave k , il numero di nodi del sotto-albero (del BST) radicato in k e situati a una profondità pari si ottiene contando tutte i nodi che si trovano a profondità pari e che sono contenuti nel sotto-albero il cui nodo radice ha come chiave il valore k . Si noti che la radice dell'intero BST ha profondità 0, che è un numero pari. Il conteggio dei nodi include anche la radice del sotto-albero se si trova a una profondità pari. Se la chiave k non è presente nel BST o se il BST è vuoto o se il sotto-albero non contiene nodi a profondità pari, l'algoritmo deve ritornare il valore 0.

Per esempio, dato l'albero di Figura 1, si ha:

- Conteggio nodi a profondità pari nel sotto-albero radicato in 8: 4 (nodi conteggiati: 8, 1, 6, 14).
- Conteggio nodi a profondità pari nel sotto-albero radicato in 3: 2 (nodi conteggiati: 1, 6).
- Conteggio nodi a profondità pari nel sotto-albero radicato in 1: 1 (nodi conteggiati: 1).
- Conteggio nodi a profondità pari nel sotto-albero radicato in 4: 0 (non ci sono nodi a profondità pari nel sotto-albero radicato in 4).
- Conteggio nodi a profondità pari nel sotto-albero radicato in 5: 0 (la chiave non esiste).

L'algoritmo implementato dev'essere ottimo, nel senso che deve visitare l'albero una sola volta e non deve visitare sotto-alberi inutili ai fini dell'esercizio, e la complessità temporale nel caso peggiore dev'essere $O(n)$, dove n è il numero di chiavi nel BST.

* * *

La funzione da implementare si trova nel file `exam.c`:

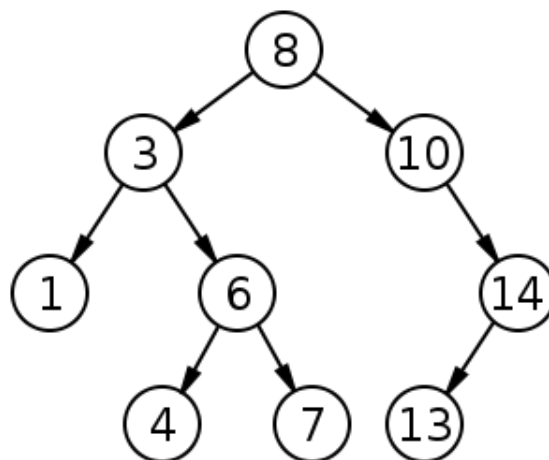


Figura 1: Un esempio di BST.

```
size_t upo_bst_subtree_count_even(const upo_bst_t bst, const void *key)
```

Parametri:

- `bst`: BST.
- `key`: puntatore alla chiave rappresentante la radice del sotto-albero per cui si vuole contare il numero di nodi a profondità pari.

Valore di ritorno:

- Se il BST non è vuoto e la chiave `key` è contenuta nel BST: numero intero rappresentante il numero di nodi a profondità pari nel sotto-albero radicato nel nodo avente come chiave il valore puntato da `key`.
- Se il BST è vuoto o la chiave `key` non è contenuta nel BST o il sotto-albero radicato in `key` non ha nodi a profondità pari: il valore intero 0.

Il tipo `upo_bst_t` è dichiarato in `include/upo/bst.h`. Per confrontare il valore di due chiavi si utilizzi la funzione di comparazione memorizzata nel campo `key_cmp` del tipo `upo_bst_t`, la quale ritorna un valore `<`, `=`, o `>` di zero se il valore puntato dal primo argomento è minore, uguale o maggiore del valore puntato dal secondo argomento, rispettivamente.

Nella propria implementazione è possibile utilizzare tutte le funzioni dichiarate in `include/upo/bst.h`. Nel caso si implementino nuove funzioni, il loro prototipo deve essere dichiarato nel file `exam.c`.

Il file `test/bst_subtree_count_even.c` contiene alcuni casi di test tramite cui è possibile verificare la correttezza della propria implementazione. Per compilarlo con la propria implementazione, è sufficiente eseguire il comando:

```
make clean all
```

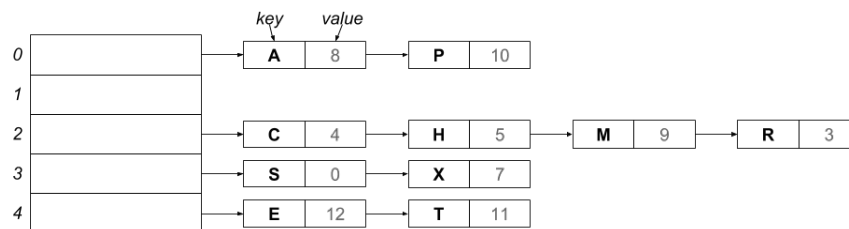


Figura 2: Un esempio di HT-SC.

Esercizio 2

Si consideri una tabella hash H con gestione delle collisioni basata su concatenazioni separate (HT-SC) in cui **le liste delle collisioni sono ordinate secondo il valore della chiave**. Implementare un algoritmo che realizzi l'operazione “odelete” la quale, data una HT-SC H , una chiave k e un valore booleano d (abbreviazione di *destroy-data*), cancelli da H la coppia chiave-valore identificata da k , preservando l'ordinamento delle liste delle collisioni, e, nel caso in cui d valga *true*, deallochi la memoria occupata dalla coppia chiave-valore rimossa. In particolare, l'operazione “odelete” funziona nel seguente modo:

- se la chiave k è contenuta in H , l'algoritmo deve rimuovere la coppia chiave-valore identificata da k dalla relativa lista delle collisioni **in maniera ordinata secondo il valore della chiave**, aggiornare la dimensione di H e, se d è *true*, deallocare la memoria allocata per la coppia chiave-valore rimossa;
- invece, se la chiave k non è contenuta in H , l'algoritmo non effettua alcuna rimozione.

In pratica, l'operazione “odelete” è simile alla classica operazione “delete” ma con la differenza che “odelete”, in caso di rimozione di una coppia chiave-valore, deve preservare l'ordinamento della lista delle collisioni da cui la coppia viene rimossa.

Per esempio, data la tabella hash di Figura 2:

- Cancellazione di $A \rightarrow$ la coppia chiave-valore identificata da questa chiave viene rimossa, preservando l'ordine dei nodi nella lista delle collisioni associata; in particolare, dopo la cancellazione, la lista conterrà le seguenti chiavi: $[P]$.
- Cancellazione di $M \rightarrow$ la coppia chiave-valore identificata da questa chiave viene rimossa, preservando l'ordine dei nodi nella lista delle collisioni associata; in particolare, dopo la cancellazione, la lista conterrà le seguenti chiavi: $[C, H, R]$.
- Cancellazione di $B \rightarrow$ non si effettua alcuna cancellazione in quanto non esiste una coppia-chiave valore identificata da questa chiave.
- Cancellazione di $Y \rightarrow$ non si effettua alcuna cancellazione in quanto non esiste una coppia-chiave valore identificata da questa chiave.

L'algoritmo implementato dev'essere ottimo, nel senso che deve visitare la HT-SC una sola volta e non deve visitare parti della HT-SC inutili ai fini dell'esercizio, e la complessità temporale nel caso medio dev'essere $\Theta(1)$.

* * *

La funzione da implementare si trova nel file `exam.c` e ha il seguente prototipo:

```
void upo_ht_sepchain_odelete(upo_ht_sepchain_t ht, const void *key, int destroy_data);
```

Parametri:

- `ht`: HT-SC.
- `key`: puntatore alla chiave.
- `destroy_data`: se diverso da 0, la funzione, oltre a deallocare la memoria allocata per il nodo da cancellare, deve deallocare anche la memoria allocata per la chiave e il valore memorizzati in quel nodo; altrimenti (se uguale a 0), la funzione deve deallocare solo la memoria allocata per il nodo da cancellare.

Valore di ritorno: nulla.

La deallocazione della memoria deve essere effettuata tramite la funzione `free()` della libreria standard del c.

Il tipo `upo_ht_sepchain_t` è dichiarato in `include/upo/hashtable.h`. Per confrontare il valore di due chiavi si utilizzi la funzione di comparazione memorizzata nel campo `key_cmp` del tipo `upo_ht_sepchain_t`, la quale ritorna un valore $<$, $=$, o $>$ di zero se il valore puntato dal primo argomento è minore, uguale o maggiore del valore puntato dal secondo argomento, rispettivamente. Per calcolare il valore hash di una chiave si utilizzi la funzione di hash memorizzata nel campo `key_hash` del tipo `upo_ht_sepchain_t`, la quale richiede come parametri il puntatore alla chiave di cui si vuole calcolare il valore hash e la capacità totale della HT-SC

(memorizzata nel campo `capacity` del tipo `upo_ht_sepchain_t`). Per tenere traccia della dimensione della HT-SC, si utilizzi il campo `size` del tipo `upo_ht_sepchain_t`. Infine, gli slot della HT-SC sono memorizzati nel campo `slots` del tipo `upo_ht_sepchain_t`, che è una sequenza di slot, ciascuno dei quali di tipo `upo_ht_sepchain_slot_t` e contenente il puntatore alla propria lista delle collisioni.

Nella propria implementazione è possibile utilizzare tutte le funzioni dichiarate in `include/upo/hashtable.h`. Nel caso si implementino nuove funzioni, i prototipi e le definizioni devono essere inserite nel file `exam.c`.

Il file `test/ht_sepchain_odelete.c` contiene alcuni casi di test tramite cui è possibile verificare la correttezza della propria implementazione. Per compilarlo con la propria implementazione, è sufficiente eseguire il comando:

```
make clean all
```

Informazioni Importanti

Superamento dell'Esame

Un esercizio della prova d'esame viene considerato completamente corretto se tutti i seguenti punti sono soddisfatti:

- è stato svolto,
- è conforme allo standard ISO C11 del linguaggio C,
- compila senza errori
- realizza correttamente la funzione richiesta,
- esegue senza generare errori,
- non contiene *memory-leak*,
- è ottimo dal punto di vista della complessità computazionale e spaziale.

Per verificare la propria implementazione è possibile utilizzare i file di test nella directory `test`, oppure, se si preferisce, è possibile scriverne uno di proprio pugno. Per verificare la presenza di errori è possibile utilizzare i programmi di debug *GNU GDB* e *Valgrind*.

In ogni caso, l'implementazione deve funzionare in generale, indipendentemente dai casi di test utilizzati durante l'esame. Quindi, il superamento dei casi di test nella directory `test` è una *condizione necessaria ma non sufficiente al superamento dell'esame*.

Istruzioni per la Consegna

- L'unico elaborato da consegnare è il file `exam.c`.
- La consegna avviene tramite il caricamento del file `exam.c` nell'apposito form sul sito D.I.R. indicato dal docente.

Gli elaborati consegnati che non rispettano tutte le suddette istruzioni o che vengono consegnati in ritardo, non saranno soggetti a valutazione.

Comandi utili

- Comando di compilazione tramite GNU GCC:

```
gcc -Wall -Wextra -std=c11 -pedantic -g -I./include -o eseguibile sorgente1.c sorgente2.c ... -L./lib -lupoalglib
```

- Comando di compilazione tramite GNU Make:

```
make clean all
```

- Comando di debug tramite GNU GDB:

```
gdb ./eseguibile
```

- Verifica di memory leak e accessi non validi alla memoria tramite Valgrind:

```
valgrind --tool=memcheck --leak-check=full ./eseguibile
```

- Manuale in linea di una funzione standard del C:

```
man funzione
```