UNIVERSITÀ DEL PIEMONTE ORIENTALE

Dipartimento di Scienze e Innovazione Tecnologica

Esame di
Algoritmi 1 – Sperimentazioni (VC)

25 giugno 2020

Testo d'Esame

Esercizio 1 (max 15 punti)

Nella definizione classica di un albero binario di ricerca (BST) non possono esserci chiavi duplicate. Si consideri la seguente estensione dei BST che permette di gestire chiavi duplicate. Ogni nodo del BST, anziché contenere una chiave k e il suo valore associato v (come nella versione classica), contiene una chiave k e una lista di valori $[v_1, v_2, \ldots, v_{n_k}]$ associati alla chiave k (un nodo della lista per ogni valore). In questo modo, quando si vuole inserire nel BST una coppia chiave-valore $\langle k, v' \rangle$, se il BST non contiene alcun nodo associato alla chiave k, se ne crea uno nuovo, memorizzandovi la chiave k e la lista $[v']$ composta da un solo nodo; invece, se nel BST c'è già un nodo associato alla chiave k con associata la seguente lista di valori $[v_1, v_2, \ldots, v_{n_k}]$, lo si aggiorna, aggiungendo il nuovo valore nella lista dei valori (per es., aggiungendo un nuovo nodo in testa alla lista, ottenendo la nuova lista $[v', v_1, v_2, \ldots, v_{n_k}]$).

Implementare un algoritmo che, dato un BST con l'estensione suddetta e una coppia chiave-valore $\langle k, v \rangle$, inserisca $\langle k, v \rangle$ nel BST.

Per es., dato un BST vuoto:

- l'inserimento di ⟨10, "foo"⟩, aggiornerà il BST nel seguente modo:

  ⟨10, ["foo"]⟩

- l'inserimento di ⟨3, "xyz"⟩, aggiornerà il BST nel seguente modo:

  ⟨10, ["foo"]⟩
  /
  ⟨3, ["xyz"]⟩

- l'inserimento di ⟨10, "bar"⟩, aggiornerà il BST nel seguente modo:

  ⟨10, ["bar", "foo"]⟩
  /
  ⟨3, ["xyz"]⟩

- l'inserimento di ⟨10, "geek"⟩, aggiornerà il BST nel seguente modo:

  ⟨10, ["geek", "bar", "foo"]⟩
  /
  ⟨3, ["xyz"]⟩

- l'inserimento di ⟨12, "abc"⟩, aggiornerà il BST nel seguente modo:

  ⟨10, ["geek", "bar", "foo"]⟩
  / \
  ⟨3, ["xyz"]⟩ ⟨12, ["abc"]⟩

Nell'esempio precedente, la scelta di effettuare l'inserimento di un nuovo nodo nella lista dei valori in testa alla lista è del tutto arbitraria; è possibile utilizzare una strategia differente.

L'algoritmo implementato deve essere ottimo, nel senso che non deve visitare parti del BST inutili ai fini dell'esercizio.

* * *

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

```
void upo_bst_insert(upo_bst_t bst, void *key, void *value);
```

Parametri:

- `bst`: BST.
- `key`: puntatore alla chiave.
- `value`: puntatore al valore.

Valore di ritorno: nessuno.

Il tipo `upo_bst_t` è dichiarato in `include/upo/bst.h`. All'interno dello stesso file, il tipo `upo_bst_node_t` rappresenta un nodo del BST ed è una struttura contenente i seguenti campi:

- `key`: puntatore alla chiave del nodo;
- `values`: puntatore alla lista concatenata dei valori associati alla chiave del nodo; ogni nodo della lista ha come tipo `upo_bst_value_list_node_t`, il quale è una struttura il cui campo `value` contiene un puntatore a un valore, e il campo `next` punta al prossimo nodo della lista;
- `left`: puntatore al figlio sinistro del nodo;
- `right`: puntatore al figlio sinistro del nodo.

Per confrontare il valore di due chiavi del BST 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 s'implementino nuove funzioni, i prototipi e le definizioni devono essere inserite nel file `exam.c`.

Il file `test/bst_insert.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
```

Esercizio 2 (max 15 punti)

Implementare un algoritmo che, date due tabelle hash *dest_ht* e *src_ht* con gestione delle collisioni basata su concatenazioni separate (HT-SC), effettui il "merge" (cioè, la fusione) di *src_ht* in *dest_ht*.

Come mostrato nell'esempio di Figura 1, il "merge" di *src_ht* in *dest_ht* consiste nell'inserire tutte le chiavi di *src_ht* in *dest_ht*. Si noti che:

- La HT-SC *src_ht* non deve essere modificata.
- La HT-SC *dest_ht* non è necessariamente vuota. In particolare, dopo il "merge" tutte le chiavi (e i valori) precedentemente presenti in *dest_ht*, devono rimanere inalterati (per es., nella figura, si vedano le chiavi "A", "B", e "X" – e i valori associati – di *dest_ht*). Inoltre, se in *dest_ht* c'è già una chiave contenuta anche in *src_ht*, quella chiave (e il valore associato) deve rimanere inalterato in *dest_ht* (per es., nella figura, si vedano le chiavi "A" e "X" – e i valori associati – di *dest_ht*).
- Le due HT-SC possono avere capacità differenti (per es., nella figura, *src_ht* ha 5 slot, mentre *dest_ht* ha 3 slot).
- Le due HT-SC possono usare funzioni hash differenti.
- Si assume che i tipi delle chiavi e dei valori delle due HT-SC siano uguali.

Si noti che, nell'esempio, la scelta di effettuare l'inserimento di un nuovo nodo nella lista delle collisioni in coda alla lista è del tutto arbitraria; è possibile utilizzare una strategia differente.

L'algoritmo implementato deve essere ottimo, nel senso che non deve visitare parti della HT-SC inutili ai fini dell'esercizio.

* * *

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

```
void upo_ht_sepchain_merge(upo_ht_sepchain_t dest_ht, const upo_ht_sepchain_t src_ht);
```

Parametri:

- `dest_ht`: HT-SC di destinazione.

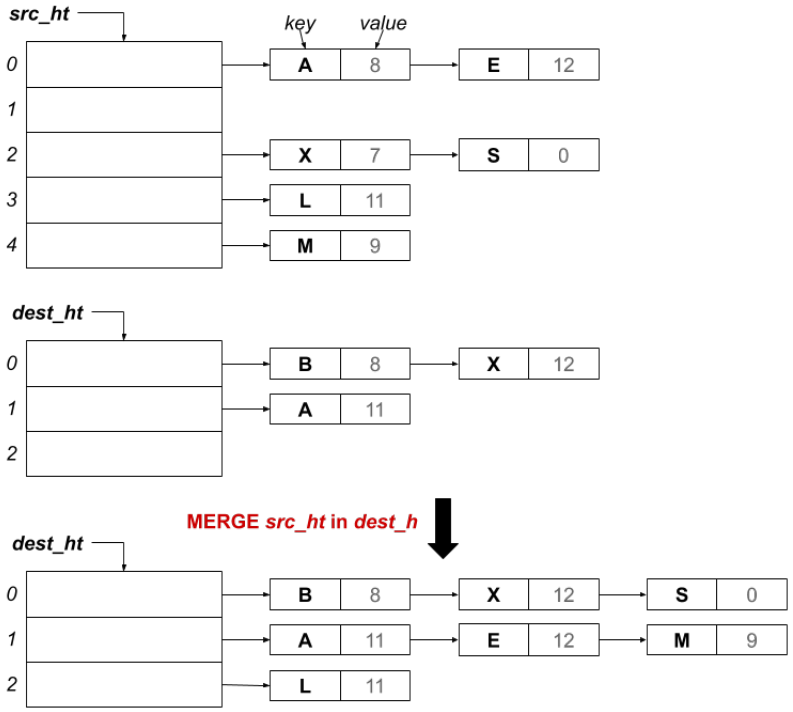

Figura 1: Un esempio di operazione “merge” tra due HT-SC: “merge” di *src_ht* in *dest_ht*.

- `src_ht`: HT-SC di origine.

Valore di ritorno: nessuno.

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_merge.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 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
```