

Esame di Algoritmi 1 – Sperimentazioni (VC)

Prova di esempio

Testo d'Esame

Esercizio 1

Implementare un algoritmo che ritorni **il predecessore di una chiave in un albero binario di ricerca (BST)**.

Dati in input un BST e una chiave k , il predecessore di k nel BST è la più grande chiave k' contenuta nel BST tale che $k' < k$. Se il predecessore di k non esiste o se il BST è vuoto, l'algoritmo deve ritornare il valore **NULL**.

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

- Predecessore di 8: 7
- Predecessore di 1: **NULL** (non esiste)
- Predecessore di 5: 4
- Predecessore di 14: 13
- Predecessore di 0: **NULL** (non esiste)
- Predecessore di 11: 10
- Predecessore di 100: 14

L'algoritmo implementato dev'essere ottimo, nel senso che deve visitare l'albero una sola volta e non deve visitare parti del BST 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`:

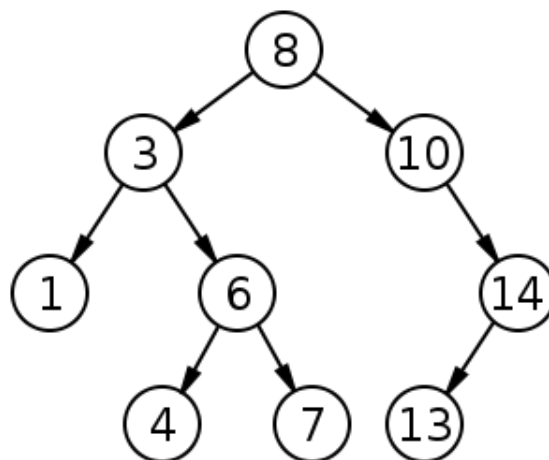


Figura 1: Un esempio di BST.

```
void* upo_bst_predecessor(const upo_bst_t bst, const void *key);
```

Parametri:

- `bst`: BST.
- `key`: puntatore alla chiave di cui si vuole ritornare il predecessore.

Valore di ritorno:

- Se il BST non è vuoto e il predecessore esiste: puntatore alla chiave rappresentante il predecessore di `key`.
- Se il BST è vuoto o il predecessore non esiste: `NULL`.

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_predecessor.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

L'algoritmo **bubble-sort bidirezionale** è un algoritmo di ordinamento di complessità quadratica (variante del bubble sort) in cui si scansione ripetutamente in avanti e all'indietro l'intera sequenza di elementi, scambiando di posizione quegli elementi adiacenti che si trovano nell'ordine sbagliato. In particolare, l'algoritmo effettua ripetutamente i seguenti due passi:

1. *Scansione in avanti*: a partire dall'inizio della sequenza, se l'elemento i -esimo della sequenza e quello successivo sono fuori ordine, vengono scambiati.
2. *Scansione all'indietro*: a partire dalla fine della sequenza, se l'elemento i -esimo della sequenza e quello precedente sono fuori ordine, vengono scambiati.

I passi suddetti sono ripetuti fino a quando non vengono più effettuati scambi in *nessuno dei passi* (cioè, tutti gli elementi si trovano nella posizione corretta). A questo punto, la sequenza è ordinata.

Per esempio, si consideri la sequenza da ordinare: [5, 1, 4, 2, 8]. L'algoritmo esegue i seguenti passi:

1. Scansione in avanti:
 - (a) Controlla 5 e 1 → Scambia: [1, 5, 4, 2, 8].
 - (b) Controlla 5 e 4 → Scambia: [1, 4, 5, 2, 8].
 - (c) Controlla 5 e 2 → Scambia: [1, 4, 2, 5, 8].
 - (d) Controlla 5 e 8 → Nulla da scambiare.
2. Scansione all'indietro:
 - (a) Controlla 8 e 5 → Nulla da scambiare.
 - (b) Controlla 5 e 2 → Nulla da scambiare.
 - (c) Controlla 2 e 4 → Scambia: [1, 2, 4, 5, 8].
 - (d) Controlla 2 e 1 → Nulla da scambiare.
3. Scansione in avanti:
 - (a) Controlla 1 e 2 → Nulla da scambiare.
 - (b) Controlla 2 e 4 → Nulla da scambiare.
 - (c) Controlla 4 e 5 → Nulla da scambiare.
 - (d) Controlla 5 e 8 → Nulla da scambiare.

Non ci sono stati scambi → Termina esecuzione.

* * *

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

```
void upo_bidi_bubble_sort(void *base, size_t n, size_t size, upo_sort_comparator_t cmp);
```

Parametri:

- `base`: puntatore alla prima cella dell'array da ordinare.
- `n`: numero di elementi dell'array da ordinare.
- `size`: numero di byte che ciascun elemento dell'array da ordinare occupa.
- `cmp`: puntatore alla funzione di comparazione utilizzata per confrontare due elementi dell'array, 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.

Valore di ritorno: nulla.

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

Il file `test/bidi_bubble_sort.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
```