Algoritmi 1 – Sperimentazioni
Esercizi Aggiuntivi

Marco Guazzone
DiSIT, Università del Piemonte Orientale
marco.guazzone@uniupo.it

Indice

Parte I
Puntatori C – Concetti Avanzati

Esercizio 1:

Implementare in C la funzione:

```
void upo_hex_fprint(FILE *stream, const void *p, size_t n);
```

per stampare in esadecimale sullo stream di output `stream` i primi `n` byte (ciascuno interpretato come **`unsigned char`**) dell'area di memoria puntata da `p`. Ogni valore esadecimale è separato da uno spazio da quello precedente.

Parametri

La funzione accetta i seguenti parametri:

- `stream`: stream di output;
- `p`: puntatore all'area di memoria;
- `n`: numero di byte dell'area di memoria che devono essere stampati.

Valore di Ritorno

La funzione non ritorna alcun valore.

Esempi

- La chiamata:

```
char *s = "Hello, World!";
upo_hex_fprintf(stdout, s, strlen(s));
```

  deve stampare sullo standard output la seguente stringa:

```
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21
```

- La chiamata:

```
char cary[] = "GNU is Not Unix";
upo_hex_fprintf(stdout, cary + (sizeof cary)/2, sizeof cary - (sizeof cary)/2);
```

  deve stampare sullo standard output la seguente stringa:

```
6F 74 20 55 6E 69 78 0
```

- La chiamata:

```
unsigned char ucary[] = {255,128,64,32,16};
upo_hex_fprintf(stderr, ucary, sizeof ucary);
```

  deve stampare sullo standard error la seguente stringa:

```
FF 80 40 20 10
```

Esercizio 2:

Implementare in C la funzione:

```
void upo_mem_set(void *p, unsigned char c, size_t n);
```

per impostare i primi n byte (ciascuno interpretato come **unsigned char**) dell'area di memoria puntata da p al valore c.

Parametri

La funzione accetta i seguenti parametri:

- p: puntatore all'area di memoria;
- c: valore (byte) usato per impostare i byte di p;
- n: numero di byte dell'area di memoria che devono essere impostati al valore c.

Valore di Ritorno

La funzione non ritorna alcun valore.

Esempi

- La chiamata:

  ```
  char cary[] = "Hello, World!";
  upo_mem_set(cary, '?', strlen(cary));
  ```

  deve modificare l'input in modo da produrre il seguente risultato:

  ```
  cary: "?????????????"
  ```

- La chiamata:

  ```
  int i = 10;
  upo_mem_set(&i, 0, sizeof i);
  ```

  deve modificare l'input in modo da produrre il seguente risultato:

  ```
  i: 0
  ```

- La chiamata:

  ```
  unsigned char ucary[] = {255,128,64,32,16,8};
  upo_mem_set(ucary, 127, (sizeof ucary)/2);
  ```

  deve modificare l'input in modo da produrre il seguente risultato:

  ```
  ucary: {127,127,127,32,16,8}
  ```

Note

La funzione esegue la stessa operazione della funzione `memset()` della libreria standard del C. Pertanto, nell'implementazione *non* è consentito usare tale funzione. È invece possibile usarla per verificare la corretta della propria implementazione.

Esercizio 3:

Implementare in C la funzione:

```
int upo_mem_cmp(const void *p1, const void *p2, size_t n);
```

per confrontare i primi n byte (ciascuno interpretato come **unsigned char**) dell'area di memoria puntata da p1 con quelli dell'area di memoria puntata da p2.

Parametri

La funzione accetta i seguenti parametri:

- p1: puntatore alla prima area di memoria da confrontare;
- p2: puntatore alla seconda area di memoria da confrontare;
- n: numero di byte delle due aree di memoria da confrontare.

Valore di Ritorno

La funzione ritorna un valore intero minore di, uguale a, o maggiore di zero se i primi n byte dell'area di memoria puntata da p1 sono minori di, uguali a, o maggiori dei primi n byte dell'area di memoria puntata da p2. In particolare, quando il valore di ritorno è diverso da zero (cioè i primi n byte delle due aree di memoria sono differenti), il segno è determinato dal segno della differenza tra la prima coppia di byte delle due aree di memoria che differisce.

Note

La funzione esegue la stessa operazione della funzione memcmp() della libreria standard del C. Pertanto, nell'implementazione *non* è consentito usare tale funzione. È invece possibile usarla per verificare la corretta della propria implementazione.

Esercizio 4:

Implementare in C la funzione:

```
int upo_all_of(const void *base, size_t n, size_t sz, int (*pred)(const void *));
```

per controllare se il predicato unario `pred` ritorna *true* (cioè, un valore diverso da zero) per tutti gli `n` elementi dell'array `base`.

Parametri

La funzione accetta i seguenti parametri:

- `base`: puntatore alla prima cella dell'array;
- `n`: numero di elementi dell'array;
- `sz`: numero di byte occupati in memoria da ciascun elemento dell'array;
- `pred`: puntatore a una funzione rappresentante un predicato unario che ritorna un valore diverso da zero (cioè, *true*) se il predicato è soddisfatto, e un valore uguale a zero (cioè, *false*) in caso contrario.

Valore di Ritorno

La funzione ritorna un valore intero diverso da zero se il predicato `pred` è soddisfatto per tutti gli `n` elementi dell'array `base`; altrimenti, ritorna zero.

Note

La funzione valuta il predicato `pred` su un ciascuno elemento dell'array finché il predicato è soddisfatto. In particolare, essa esegue la stessa operazione del seguente frammento di pseudo-codice (rappresentante codice C non valido perché si dereferenzia un `void*`):

```
if (pred(base[0]) && pred(base[1]) && ... && pred(base[n-1]))
    return 1;
else
    return 0;
```

Esempi

Per esempio, un possibile predicato unario è quello che verifica se un numero è pari; quindi, il seguente frammento di codice verifica se l'array di interi `iary` contiene esclusivamente numeri pari:

```
int is_even(const void *v)
{
    assert( v );
    return *((const int*) v) % 2;
}

int iary[] = {...};
...
ret = upo_all_of(iary, n_iary, sz_iary, is_even);
```

Un altro esempio di predicato unario è quello che verifica se una stringa è un palindromo (cioè, una parola che, letta in senso inverso, sia da sinistra sia da destra, rimane identica); quindi, il seguente frammento di codice verifica se tutte le stringhe nell'array `sary` sono palindromi.

```
int is_palindrome(const void *v)
{
    assert( v );

    const char **ps = (const char**) v;
    size_t len = strlen(*ps);
```

```
    for (size_t i = 0; i < len/2; ++i)
    {
        if ((*ps)[i] != (*ps)[len-i-1])
            return 0;
    }
    return 1;
}

char *sary[] = {...};
...
ret = upo_all_of(sary, n_sary, sz_sary, is_palindrome);
```

* * *

Parte II
Strutture Dati Elementari

Esercizio 5:

La struttura dati **Coda** (**Queue** in inglese) è un contenitore di elementi in cui vige la politica *First-In First Out* (FIFO), cioè la rimozione di un elemento avviene secondo l'ordine di arrivo.

Per esempio, se tre elementi `X`, `Y` e `Z` arrivano nell'ordine definito da sinistra verso destra, la coda associata è:

```
[X, Y, Z]
```

dove `X` è il primo elemento della coda (cioè, il primo arrivato) e `Z` è l'ultimo elemento (cioè, l'ultimo arrivato).

Un ADT per la struttura dati Coda deve fornire almeno le seguenti operazioni fondamentali:

- `enqueue(q,x)`: inserisce l'elemento `x` alla fine della coda `q`; ad esempio, data la coda `q=[X,Y,Z]`, la chiamata a `enqueue(q,W)` modifica la coda in `q=[X,Y,Z,W]`;
- `dequeue(q)`: rimuove il primo elemento della coda `q`; ad esempio, data la coda `q=[X,Y,Z]`, la chiamata a `dequeue(q)` modifica la coda in `q=[Y,Z]`;
- `peek(q)`: ritorna il primo elemento della coda `q`, senza rimuoverlo; ad esempio, data la coda `q=[X,Y,Z]`, la chiamata a `peek(q)` ritorna l'elemento `X`;
- `size(q)`: ritorna la lunghezza della coda `q`; ad esempio, data la coda `q=[X,Y,Z]`, la chiamata a `size(q)` ritorna 3;
- `is_empty(q)`: ritorna `true` o `false` a seconda che la coda `q` sia vuota oppure no; ad esempio, data la coda `q=[X,Y,Z]`, la chiamata a `is_empty(q)` ritorna `false`.

Implementare in `C` l'ADT `upo_queue_t` per la struttura dati Coda il cui header pubblico `queue.h` deve avere la seguente struttura:

```c
#ifndef UPO_QUEUE_T
#define UPO_QUEUE_T

#include <stddef.h>

/** \brief The queue abstract data type. */
typedef struct upo_queue_s* upo_queue_t;

/** \brief Creates a new queue. */
upo_queue_t upo_queue_create();

/** \brief Destroys the given queue. */
void upo_queue_destroy(upo_queue_t queue, int destroy_data);

/** \brief Removes all elements from the given queue. */
void upo_queue_clear(upo_queue_t queue, int destroy_data);

/** \brief Inserts the given element into the given queue. */
void upo_queue_enqueue(upo_queue_t queue, void *data);

/** \brief Removes the next element from the given queue. */
void upo_queue_dequeue(upo_queue_t queue, int destroy_data);

/** \brief Returns the next element from the given queue. */
void* upo_queue_peek(const upo_queue_t queue);

/** \brief Returns the length of the given queue. */
size_t upo_queue_size(const upo_queue_t queue);

```

```
30 /** \brief Tells whether the given queue is empty. */
31 int upo_queue_is_empty(const upo_queue_t queue);
32
33 #endif /* UPO_QUEUE_T */
```

dove:

- Il tipo incompleto `struct upo_queue_s` è una struttura incompleta che deve essere completata nell'header privato `src/queue_private.h`.
- Il parametro `destroy_data`, quando contiene un valore diverso da zero, indica che la memoria associata ai dati utente deve essere liberata tramite una chiamata alla funzione `free()`. Per esempio, la chiamata a `upo_queue_deque(q,1)` rimuove il primo elemento della coda `q`, invocando anche la funzione `free()` sull'elemento rimosso.

L'implementazione del tipo di dati Coda deve utilizzare una lista concatenata e deve essere strutturata nei seguenti file:

- `include/upo/queue.h`: header pubblico;
- `src/queue_private.h`: header privato;
- `src/queue.c`: file di implementazione.

Bonus La complessità computazionale delle operazioni `enqueue()`, `deque()` e `peek()` deve essere $O(1)$.

Bonus+ La complessità computazionale dell'operazione `size()` deve essere $O(1)$.

* * *

Parte III
Algoritmi di Ordinamento

Esercizio 6:

L'algoritmo **bubble-sort** è un algoritmo di ordinamento di complessità quadratica in cui ogni coppia di elementi adiacenti viene scambiata di posizione se i due elementi si trovano nell'ordine sbagliato. L'algoritmo scansiona ripetutamente l'intera sequenza di elementi fino a quando non vengono più effettuati scambi (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:
   (a) Controlla 5 e 1 → Scambia: $[\mathbf{1}, \mathbf{5}, 4, 2, 8]$.
   (b) Controlla 5 e 4 → Scambia: $[1, \mathbf{4}, \mathbf{5}, 2, 8]$.
   (c) Controlla 5 e 2 → Scambia: $[1, 4, \mathbf{2}, \mathbf{5}, 8]$.
   (d) Controlla 5 e 8 → Nulla da scambiare.

   C'è stato almeno uno scambio → Ripeti scansione (si veda il passo sottostante).
2. Scansione:
   (a) Controlla 1 e 4 → Nulla da scambiare.
   (b) Controlla 4 e 2 → Scambia: $[1, \mathbf{2}, \mathbf{4}, 5, 8]$.
   (c) Controlla 4 e 5 → Nulla da scambiare.
   (d) Controlla 5 e 8 → Nulla da scambiare.

   C'è stato almeno uno scambio → Ripeti scansione (si veda il passo sottostante).
3. Scansione:
   (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.

Implementare in `C` l'algoritmo bubble-sort la cui funzione deve avere il seguente prototipo:

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

dove:

- `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 da ordinare avente come signature:

  ```
  int cmp(const void *a, const void *b);
  ```

  dove:
  - `a` è un puntatore al primo elemento da confrontare;
  - `b` è un puntatore al secondo elemento da confrontare;

- il valore di ritorno è un numero intero maggiore di, uguale a, o minore di zero se, rispettivamente, il primo elemento è maggiore del, uguale al, o minore del secondo elemento.

Il tipo `upo_comparator_t` è un alias per il tipo puntatore a funzioni di comparazione, cioè:

```
typedef int (*upo_sort_comparator_t)(const void*,const void*);
```

L'algoritmo implementato deve essere ottimo, nel senso che dato una sequenza a di n elementi da ordinare:

- la complessità computazionale nel caso peggiore (a è ordinato al contrario) deve essere $O(n^2)$,
- la complessità spaziale deve essere $O(n)$ (cioè, non deve richiedere spazio superiore a quello occupato da a ed eventualmente da altre variabili ausiliarie, come quelle di ciclo, la cui occupazione di memoria è costante, cioè $O(1)$).

Esercizio 7:

La complessità computazionale nel caso peggiore dell'algoritmo **quick-sort** classico (come quello presentato a lezione) è pari a $\Theta(n^2)$, dove n è la dimensione dell'array da ordinare. Il caso peggiore si verifica quando ad ogni passo l'array viene partizionato in due regioni di cui una contiene un solo elemento. Nella versione classica del quick-sort, tale situazione capita quando l'array da ordinare è già ordinato; inoltre, l'algoritmo esibisce prestazioni mediocri anche a fronte di array parzialmente ordinati.

In letteratura, sono state proposte diverse varianti per cercare di abbassare tale complessità. Per esempio, è possibile ottenere una complessità nel caso peggiore pari a $O(n \log n)$ combinando le seguenti modifiche:

- Partizionamento tramite la tecnica *mediana-di-3*;
- Ottimizzazione per l'ordinamento di array piccoli.

In Alg. 1 è mostrata una traccia dello pseudo-codice dell'algoritmo quick-sort con la combinazione delle suddette tecniche, le quali sono descritte in dettaglio nel testo che segue.

Algorithm 1 Quick-sort con tecnica della mediana-di-3 e ottimizzazione per l'ordinamento di array piccoli.

```
1:  procedure QuickSortMedian3Cutoff(a, n)
2:      QuickSortMedian3CutoffRec(a, 0, n − 1)
3:  end procedure
4:  procedure QuickSortMedian3CutoffRec(a, lo, hi)
5:      if lo ≥ hi then
6:          return
7:      end if
8:      /* Applicazione dell'ottimizzazione per l'ordinamento di array piccoli */
9:      ...
10:     j ← PartitionMedian3(a, lo, hi)
11:     /* Chiamate ricorsive a QuickSortMedian3CutoffRec per ordinare le due regioni di a appena partizionate */
12:     ...
13: end procedure
14: function PartitionMedian3(a, lo, hi)
15:     mid ← (lo + hi)/2
16:     /* Selezione pivot come mediana-di-3 tra a[lo], a[mid] e a[hi], con scambio */
17:     ...
18:     /* Partizionamento del sotto-array a[lo + 1], ..., a[hi − 1] */
19: end function
```

Partizionamento tramite la tecnica *mediana-di-3*

Questa tecnica consiste nel selezionare come elemento di partizionamento (*pivot*) la mediana fra i 3 elementi del sotto-array da partizionare che si trovano all'inizio, al centro e alla fine del sotto-array.

Si definisce mediana fra 3 elementi di e_1, e_2, e_3 quell'elemento e_k tale per cui se la sequenza venisse ordinata in senso crescente e_k si troverebbe in posizione centrale (nel caso ci siano due o più elementi uguali, se ne sceglie uno in modo arbitrario). Per esempio,:

- la mediana di $2, 4, 10$ è 4;
- la mediana di $4, 10, 2$ è 4 (infatti, se ordinassimo la sequenza si otterrebbe $2, 4, 10$ il cui elemento centrale è 4);
- la mediana di $10, 4, 2$ è 4 (infatti, se ordinassimo la sequenza si otterrebbe $2, 4, 10$ il cui elemento centrale è 4);
- la mediana di $2, 10, 2$ è 2;
- la mediana di $10, 2, 10$ è 10.

Questa tecnica risulta particolarmente efficace se, mentre si cerca quale dei tre suddetti elementi rappresenta quello mediano, si scambiano di posizione i tre elementi nel sotto-array in modo che siano tra loro ordinati, cioè in modo che l'elemento che precede quello mediano, l'elemento mediano e l'elemento che segue quello mediano finiscano, rispettivamente, all'inizio, al centro e alla fine del sotto-array da partizionare. Grazie a questi scambi è possibile effettuare il

partizionamento su una porzione più piccola dell'array che parte dal secondo elemento e termina con il penultimo elemento del sotto-array da partizionare (perchè il primo e l'ultimo elemento si trovano già nella posizione corretta, cioè a sinistra e a destra del pivot, rispettivamente).

Quindi, la procedura di partizionamento PartitionMedian3(a, lo, hi) esegue le seguenti operazioni:

1. Seleziona come pivot l'elemento mediano tra $a[lo]$, $a[mid]$ (con $mid = (lo + hi)/2$) e $a[hi]$ (anzichè usare l'elemento $a[lo]$, come nella versione classica del quick-sort).
2. Scambia di posizione gli elementi $a[lo]$, $a[mid]$ e $a[hi]$ in modo che risulti:

   $$a[lo] \leq a[mid] \leq a[hi]$$

   Gli scambi sono effettuati durante la scelta dell'elemento mediano.
3. Partiziona il sotto-array a a partire da $lo + 1$ fino a $hi - 1$ usando come pivot l'elemento $a[mid]$

Suggerimento. Si consiglia di implementare la nuova funzione PartitionMedian3 in modo tale che al suo interno effettui i passi suddetti e che in particolare per l'ultimo passo utilizzi la funzione Partition della versione classica del quick-sort. Per poter fare ciò però occorre prima scambiare il pivot (che si trova in $a[mid]$) con un altro elemento dell'array a in modo da tale che Partition scelga come pivot l'elemento corretto. Si riveda il funzionamento di Partition per capire quale elemento usare per lo scambio.

Ottimizzazione per l'ordinamento di array piccoli

Questa tecnica consiste nell'interrompere le chiamate ricorsive all'algoritmo quick-sort quando la dimensione del sotto-array da ordinare è minore o uguale a un certo valore soglia (*cutoff*), e al loro posto si chiama l'algoritmo *insertion-sort*, il quale risulta essere più efficiente per array di piccole dimensioni. Tipicamente il cutoff è minore di 20 e spesso uguale a 10.

Implementazione in `C`

Implementare in `C` l'algoritmo quick-sort con le modifiche descritte nei punti precedenti, la cui funzione deve avere il seguente prototipo:

```
void upo_quick_sort_median3_cutoff(void *base, size_t n, size_t size,
    upo_sort_comparator_t cmp);
```

dove:

- `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 da ordinare avente come signature:

  ```
  int cmp(const void *a, const void *b);
  ```

  dove:

  - `a` è un puntatore al primo elemento da confrontare;
  - `b` è un puntatore al secondo elemento da confrontare;
  - il valore di ritorno è un numero intero maggiore di, uguale a, o minore di zero se, rispettivamente, il primo elemento è maggiore del, uguale al, o minore del secondo elemento.

  Il tipo `upo_comparator_t` è un alias per il tipo puntatore a funzioni di comparazione, cioè:

  ```
  typedef int (*upo_sort_comparator_t)(const void*,const void*);
  ```

Come *cutoff* per l'applicazione dell'ottimizzazione per array piccoli si utilizzi il valore 10.

* * *

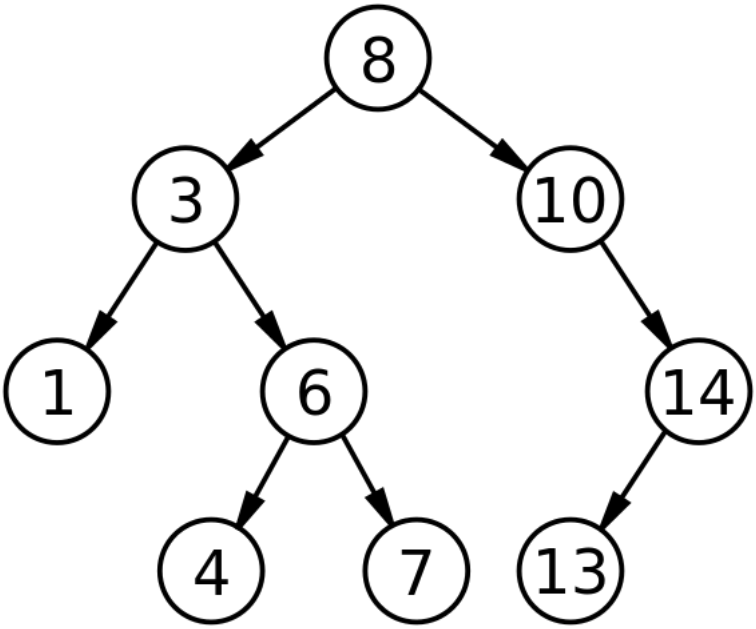

Figura 1: Un esempio di BST.

Parte IV
Alberti Binari di Ricerca

Esercizio 8:

Implementare un algoritmo che ritorni il **rango di una data chiave** k **in un albero binario di ricerca (BST)**. Dato un BST e una chiave k (non necessariamente contenuta nel BST), il rango di k nel BST equivale al numero di chiavi presenti nel BST che sono minori di k. Per esempio, dato l'albero di Fig. 1, si ha:

- Rango di 8: 5
  (nel BST le chiavi $1, 3, 4, 6, 7$ sono minori di 8)
- Rango di 1: 0
  (nel BST non esistono chiavi minori di 0)
- Rango di 13: 7
- Rango di 14: 8
- Rango di 0: 0
- Rango di 5: 3
- Rango di 12: 7
- Rango di 100: 9

Implementare in `C` l'algoritmo richiesto la cui funzione deve avere il seguente prototipo:

```
size_t upo_bst_rank(const upo_bst_t tree, const void *key)
```

Parametri:

- `bst`: BST in cui calcolare il rango della chiave,
- `key`: puntatore alla chiave per cui si vuole calcolare il rango.

Valore di ritorno:

- Numero intero rappresentante il rango di `key` nel BST `bst`, se il BST non è vuoto.
- `0`, se l'albero è vuoto.

L'algoritmo implementato deve essere ottimo, nel senso che deve visitare l'albero una sola volta e non deve visitare sotto-alberi inutili ai fini dell'esercizio. In particolare, l'algoritmo non deve visitare sotto-alberi che contengono esclusivamente chiavi maggiori di k. Per esempio, dato l'albero di Fig. 1, per il calcolo del rango di 5 l'algoritmo non deve visitare il sotto-albero radicato in 10.

Esercizio 9:

Implementare un algoritmo per trovare il **predecessore di una chiave in un albero binario di ricerca** (BST). Dato un BST e una chiave k (non necessariamente contenuta nel BST), il predecessore di k è la più grande chiave k' contenuta nel BST tale che $k' < k$. Per esempio, dato l'albero di Fig. 1, si ha:

- Predecessore di 8: 7
- Predecessore di 1: non esiste
- Predecessore di 13: 10
- Predecessore di 12: 10
- Predecessore di 0: non esiste
- Predecessore di 100: 14

Implementare in `C` l'algoritmo richiesto la cui funzione deve avere il seguente prototipo:

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

Parametri:

- `bst`: BST in cui cercare il predecessore della chiave,
- `key`: puntatore alla chiave per cui si vuole ottenere il suo predecessore.

Valore di ritorno:

- Puntatore alla chiave rappresentante il predecessore di `key`, se esiste,
- `NULL`, se l'albero è vuoto o il predecessore di `key` non esiste.

L'algoritmo implementato deve essere ottimo, nel senso che deve visitare l'albero una sola volta e non deve visitare sotto-alberi inutili ai fini dell'esercizio.

Esercizio 10:

Implementare un algoritmo che ritorni il **valore e la profondità di una chiave in un albero binario di ricerca** (BST). Se la chiave non è contenuta nel BST, l'algoritmo deve ritornare **nil** come valore della chiave e -1 come profondità. Per profondità di una chiave s'intende la profondità del nodo del BST in cui la chiave è memorizzata. Si ricordi che la radice di un BST si trova a profondità zero. Per esempio, dato l'albero di Fig. 1 e denotando il valore associato a una certa chiave k con "valore di k", si ha:

- Valore e profondità di 8: "valore di 8" e 0
- Valore e profondità di 1: "valore di 1" e 2
- Valore e profondità di 13: "valore di 13" e 3
- Valore e profondità di 12: **nil** e -1
- Valore e profondità di 0: **nil** e -1
- Valore e profondità di 100: **nil** e -1

Implementare in `C` l'algoritmo richiesto la cui funzione deve avere il seguente prototipo:

```
void* upo_bst_get_value_depth(const upo_bst_t tree, const void *key, long *depth);
```

Parametri:

- `bst`: BST in cui cercare la chiave,
- `key`: puntatore alla chiave per cui si vuole ottenere il valore e la profondità associati,

- `depth`: variabile in cui verrà memorizzata la profondità della chiave.

Valore di ritorno:

- Puntatore al valore associato alla chiave `key`, se `key` è contenuta nel BST,
- `NULL`, se l'albero è vuoto o la chiave `key` non è contenuta nel BST.

L'algoritmo implementato deve essere ottimo, nel senso che deve visitare l'albero una sola volta e non deve visitare sotto-alberi inutili ai fini dell'esercizio. In particolare, l'algoritmo non deve visitare interi sotto-alberi che contengono chiavi esclusivamente minori di k o esclusivamente maggiori di k. Per esempio, dato l'albero di Fig. 1, per la ricerca della chiave 13 l'algoritmo non deve visitare il sotto-albero radicato in 3.

Esercizio 11:

Implementare un algoritmo che ritorni la **lista delle chiavi in un albero binario di ricerca (BST) che sono minori di o uguali a una data chiave** k. Dato un BST e una chiave k (non necessariamente contenuta nel BST), il numero di chiavi nel BST minori o uguali a k si ottiene contando tutte le chiavi contenute nel BST che sono minori della o uguali alla chiave k. Per esempio, dato l'albero di Fig. 1, si ha:

- Chiavi ≤ 8: $[1, 3, 4, 6, 7, 8]$
- Chiavi ≤ 1: $[1]$
- Chiavi ≤ 14: $[1, 3, 4, 6, 7, 8, 10, 13, 14]$
- Chiavi ≤ 0: [] (lista vuota)
- Chiavi ≤ 5: $[1, 3, 4]$
- Chiavi ≤ 100: $[1, 3, 4, 6, 7, 8, 10, 13, 14]$

Implementare in `C` l'algoritmo richiesto la cui funzione deve avere il seguente prototipo:

```
upo_bst_key_list_t upo_bst_keys_le(const upo_bst_t tree, const void *key);
```

Parametri:

- `bst`: BST,
- `key`: puntatore alla chiave per cui si vuole ritornare la lista delle chiavi minori del o uguali al valore puntato.

Valore di ritorno:

- La lista delle chiavi nel BST `bst` che sono minori del o uguali al valore puntato da `key`, se il BST non è vuoto.
- Una lista vuota, se l'albero è vuoto o se nel BST non esistono chiavi minori della o uguali alla chiave data.

Il tipo `upo_bst_key_list_t` è definito nel file `include/upo/bst.h`.

L'algoritmo implementato deve essere ottimo, nel senso che deve visitare l'albero una sola volta e non deve visitare sotto-alberi inutili ai fini dell'esercizio. In particolare, l'algoritmo non deve visitare sotto-alberi che contengono esclusivamente chiavi maggiori di k. Per esempio, dato l'albero di Fig. 1, per la ricerca delle chiavi minori di 5 l'algoritmo non deve visitare il sotto-albero radicato in 10.

* * *

Parte V
Tabelle Hash

Esercizio 12:

Implementare in C la funzione `upo_find_dups()`:

```
upo_strings_list_t upo_find_dups(const char **strs, size_t n);
```

che restituisce una lista delle stringhe che si ripetono più di una volta nell'array di `n` stringhe `strs`.

Per esempio, se `strs` contiene le seguenti stringhe:

```
{"Tre", "tigri", "contro", "tre", "tigri"}
```

la funzione deve ritornare una lista con un nodo per `"tigri"`.

Successivamente, implementare in C la funzione `upo_find_idups()`:

```
upo_strings_list_t upo_find_idups(const char **strs, size_t n, int ignore_case);
```

che, come la funzione precedente, restituisce una lista delle stringhe che si ripetono più di una volta nell'array di `n` stringhe `strs` e tale per cui se `ignore_case` è diverso da zero il confronto tra stringhe viene effettuato ignorando il fatto che un carattere sia maiuscolo o minuscolo. Nel valore di ritorno le stringhe devono avere tutti i caratteri minuscoli.

Per esempio, se `strs` contiene le seguenti stringhe:

```
{"Tre", "tigri", "contro", "tre", "tigri"}
```

la chiamata alla funzione `upo_find_idups(strs,0)` deve ritornare una lista con un nodo per `"tigri"`, mentre la chiamata a `upo_find_idups(strs,1)` deve ritornare una lista con due nodi, uno per `"tre"` e un altro per `"tigri"`.

Entrambe le funzioni devono utilizzare una *Tabella Hash* per tenere traccia dei duplicati (si scelga il metodo di gestione delle collisioni che si preferisce). Come capacità della tabella hash si può specificare una dimensione a piacere oppure utilizzare le dimensioni di default (`UPO_HT_SEPCHAIN_DEFAULT_CAPACITY` nel caso di tabella hash con concatenazioni separate e `UPO_HT_LINPROB_DEFAULT_CAPACITY`, nel caso di tabella hash con indirizzamento aperto) dichiarate nel file `include/upo/hashtable.h` fornito a lezione. Come funzione hash si può definire una propria funzione (il cui prototipo sia conforme al tipo `upo_ht_hasher_t`) oppure utilizzare la funzione `upo_ht_hash_str_kr2e()` dichiarata nel file `include/upo/hashtable.h` fornito a lezione.

Esercizio 13:

Si consideri la struttura dati *Tabella Hash con concatenazione separata* in cui le collisioni vengono gestite tramite *liste concatenate ordinate* (in cui l'ordinamento dei nodi di una lista avviene secondo il valore della chiave).

Per esempio, si consideri la lista delle collisioni associata a un certo slot della tabella hash (la testa della lista è si trova a sinistra):

`[slot j]` → `<X,20>`→`<Z,5>`

Se in questo slot deve essere inserita la coppia chiave-valore `<Y,10>` (dove `Y` è la chiave e `10` è il valore), dopo l'inserimento la lista delle collisioni dello slot diventa:

`[slot j]` → `<X,20>`→`<Y,10>`→`<Z,5>`

Implementare in `C` l'ADT `upo_ht_sepchain_olist_t` per la suddetta struttura dati aggiungendo all'header pubblico `hashtable.h` (già utilizzato per definire gli ADT per le Tabelle Hash visti a lezione) le seguenti dichiarazioni:

```
1  /** \brief The hash table with separate chaining (based on ordered linked lists)
       abstract data type. */
2  typedef struct upo_ht_sepchain_olist_s* upo_ht_sepchain_olist_t;
3
4  /** \brief Creates a new hash table with separate chaining (based on ordered linked
       lists). */
5  upo_ht_sepchain_olist_t upo_ht_sepchain_olist_create(size_t m, upo_ht_hasher_t
       key_hash, upo_ht_comparator_t key_cmp);
6
7  /** \brief Destroys the given hash table with separate chaining (based on ordered
       linked lists). */
8  void upo_ht_sepchain_olist_destroy(upo_ht_sepchain_olist_t ht, int destroy_data);
9
10 /** \brief Removes all elements from the given hash table with separate chaining (based
       on ordered linked lists). */
11 void upo_ht_sepchain_olist_clear(upo_ht_sepchain_olist_t ht, int destroy_data);
12
13 /** \brief Returns value associated to the given key in the given hash table with
       separate chaining (based on ordered linked lists). */
14 void* upo_ht_sepchain_olist_get(const upo_ht_sepchain_olist_t ht, const void *key);
15
16 /** \brief Tells whether the given key is present in the given hash table with separate
       chaining (based on ordered linked lists). */
17 int upo_ht_sepchain_olist_contains(const upo_ht_sepchain_olist_t ht, const void *key);
18
19 /** \brief Inserts/updates the given key-value pair into the given hash table with
       separate chaining (based on ordered linked lists). */
20 void* upo_ht_sepchain_olist_put(upo_ht_sepchain_olist_t ht, void *key, void *value);
21
22 /** \brief Inserts the given key-value pair into the given hash table with separate
       chaining (based on ordered linked lists); updates are ignored. */
23 void upo_ht_sepchain_olist_insert(upo_ht_sepchain_olist_t ht, void *key, void *value);
24
25 /** \brief Removes the key-value pair associated to the given key from the given hash
       table with separate chaining (based on ordered linked lists). */
26 void upo_ht_sepchain_olist_delete(upo_ht_sepchain_olist_t ht, const void *key, int
       destroy_data);
27
28 /** \brief Returns the capacity of the given hash table with separate chaining (based
       on ordered linked lists). */
29 size_t upo_ht_sepchain_olist_capacity(const upo_ht_sepchain_olist_t ht);
30
31 /** \brief Returns the number of stored keys in the given hash table with separate
       chaining (based on ordered linked lists). */
```

```
32 size_t upo_ht_sepchain_olist_size(const upo_ht_sepchain_olist_t ht);
33
34 /** \brief Returns the load factor of the given hash table with separate chaining
       (based on ordered linked lists). */
35 double upo_ht_sepchain_olist_load_factor(const upo_ht_sepchain_olist_t ht);
36
37 /** \brief Tells whether the given hash table with separate chaining (based on ordered
       linked lists) is empty. */
38 int upo_ht_sepchain_olist_is_empty(const upo_ht_sepchain_olist_t ht);
```

dove:

- Il tipo incompleto **struct** `upo_ht_sepchain_olist_s` è una struttura incompleta che deve essere completata nell'header privato `hashtable_private.h`.
- Il parametro `destroy_data`, quando contiene un valore diverso da zero, indica che la memoria associata ai dati utente deve essere liberata tramite una chiamata alla funzione `free()`. Per esempio, l'invocazione di `upo_ht_sepchain_olist_delete(ht,key,1)` rimuove dalla Tabella Hash `ht` la coppia chiave-valore identificata da `key`, invocando anche la funzione `free()` sia sulla chiave sia sul valore.
- Il significato delle suddette funzioni è identico a quello delle operazioni omonime definite per gli ADT delle Tabelle Hash visti a lezione. Per esempio, la chiamata a `upo_ht_sepchain_olist_put(ht,key,value)` (in maniera simile a `upo_ht_sepchain_put()` e `upo_ht_linprob_put()`) inserisce nella Tabella Hash `ht` la coppia chiave-valore identificata da `key` e `value`, rispettivamente, o ne aggiorna il valore rimpiazzando quello esistente con `value` nel caso la Tabella Hash contenga già una coppia chiave-valore identificata da `key`; la funzione ritorna il valore rimpiazzato, in caso di aggiornamento, o `NULL` in caso di inserimento di una nuova coppia chiave-valore.

L'implementazione del tipo di dati "Tabella Hash con concatenazione separata basata su liste concatenate ordinate" deve essere strutturata nei seguenti file:

- `include/upo/hashtable.h`: header pubblico;
- `src/hashtable_private.h`: header privato;
- `src/hashtable.c`: file di implementazione.

Esercizio 14:

Implementare in C la funzione `upo_ht_linprob_merge()`:

```
upo_ht_linprob_merge(upo_ht_linprob_t dest_ht, const upo_ht_linprob_t src_ht);
```

che, date due tabelle hash `dest_ht` e `src_ht` con gestione delle collisioni basata su indirizzamento aperto e scansione lineare (HT-LP), effettui il "merge" (cioè, la fusione) di `src_ht` in `dest_ht`.

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-LP `src_ht` non deve essere modificata.
- La HT-LP `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. 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`.
- Se, durante il "merge", il fattore di carico di `dest_ht` diventa maggiore o uguale del 50%, la capacità di `dest_ht` dev'essere raddoppiata.
- Le due HT-LP possono avere capacità differenti.
- Le due HT-LP possono usare funzioni hash differenti.
- I tipi degli elementi contenuti nelle due HT-LP (cioè, i tipi delle chiavi e dei valori delle due HT-LP) sono uguali.

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