```
/*        Quimera, luglio 2003          *
 *           Versione 1.0.0             *
 *                                      *
 * Libreria per la gestione di oggetti *
 * all'interno di un albero binario.    *
 *                                      *
 * Dichiarazione dei tipi di variabili *
 * e di procedure utili alla libreria. */

#ifndef _TREE_H
#define _TREE_H

/* ---------------------- Dichiarazione delle variabili ----------------------- */

struct NODO_T {                 /* Struttura del nodo dell'albero */
  void *elem;
  struct NODO_T *ns, *nd;
};
typedef struct NODO_T nodo_t;

typedef nodo_t *tree_t;         /* Albero                         */

/* ------------------ Dichiarazione delle funzioni e procedure ---------------- */

tree_t new_tree (void);
/* Requisiti : nessuno                                  *
 * Ruolo: crea un albero.                               */

tree_t right_son (tree_t);
/* Requisiti : albero inizializzato e non vuoto         *
 * Ruolo: Restituisce il figlio destro dell'albero,     *
 *   NULL altrimenti.                                   */

tree_t left_son (tree_t);
/* Requisiti : albero inizializzato e non vuoto         *
 * Ruolo: Restituisce il figlio sinistro dell'albero,   *
 *   NULL altrimenti.                                   */

void * min_of_tree (tree_t);
/* Requisiti : albero inizializzato e non vuoto         *
 * Ruolo: Restituisce l'elemento più piccolo dell'albero  */

void * max_of_tree (tree_t);
/* Requisiti : albero inizializzato e non vuoto         *
 * Ruolo: Restituisce l'elemento più grande dell'albero   */

int tree_is_empty (tree_t);
/* Requisiti : albero inizializzato                     *
 * Ruolo: Restituisce 1 se l'albero è vuoto,            *
 *   0 altrimenti.                                      */

void ** get_elem (tree_t);
/* Requisiti : albero inizializzato e non vuoto         *
 * Ruolo: Restituisce il puntatore sulll'elemento della *
 *   radice dell'albero                                 */

int add_elem (tree_t *, void *, int (*) (void *, void *));
/* Requisiti : albero inizializzato                     *
 * Ruolo: aggiunge all'albero l'elemento se non è       *
 *   presente. Se l'operazione ha avuto successo        *
 *   restituisce 1 altrimenti 0. Per fare cio, ha bisogno *
 *   di una funzione di comparazione per gli elementi;  *
 *   questa rende -1, 0, 1 a seconda se il primo elemento *
 *   è rispetivamente minore, uguale o maggiore del     *
```



```
 *   secondo                                            */

int rem_elem (tree_t *, void *, int (*) (void *, void *));
/* Requisiti : albero inizializzato                     *
 * Ruolo: rimuove dall'albero l'elemento, se            *
 *   l'operazione ha successo restituisce 1 altrimenti 0. *
 *   Per fare cio, ha bisogno di una funzione di        *
 *   comparazione per gli elementi; questa rende -1, 0, 1 *
 *   a seconda se il 1 elemento è rispetivamente minore,  *
 *   uguale o maggiore del secondo                      */

void ** find_elem (tree_t, void *, int (*) (void *, void *));
/* Requisiti : albero inizializzato e non vuoto         *
 * Ruolo: cerca all'interno dell'albero l'elemento, se lo *
 *   trova restituisce un puntatore sull'elemento,      *
 *   altrimenti restituisce NULL. Per fare cio, ha      *
 *   bisogno di una funzione di comparazione per gli    *
 *   elementi; questa rende -1, 0, 1 a seconda se il    *
 *   primo elemento è rispetivamente minore, uguale o   *
 *   maggiore del secon                                 */

#endif
```

```
/*        Quimera, luglio 2003          *
 *           Versione 1.0.0             *
 *                                      *
 * Libreria per la gestione di oggetti *
 * all'interno di un albero binario.    *
 *                                      *
 * Definizione delle variabili e delle *
 * procedure utili alla libreria.       */

#include "tree.h"
#include <stdlib.h>
#include <stdio.h>

/* ---------------------------------- new_tree ---------------------------------- */

tree_t new_tree (void) {

  return NULL;
}

/* ---------------------------------- right_son ---------------------------------- */

tree_t right_son (tree_t tree) {

  return (*tree).nd;
}

/* ---------------------------------- left_son ---------------------------------- */

tree_t left_son (tree_t tree) {

  return (*tree).ns;
}

/* ---------------------------------- tree_is_empty ---------------------------------- */

int tree_is_empty (tree_t tree) {

  if ( tree == NULL )
    return 1;
  else
    return 0;
}

/* ---------------------------------- get_elem ---------------------------------- */

void ** get_elem (tree_t tree) {

  return &(*tree).elem;
}

/* ---------------------------------- min_of_tree ---------------------------------- */

void * min_of_tree (tree_t tree) {

  if ( tree_is_empty(left_son(tree)) )
    return *(get_elem(tree));
  else
    return min_of_tree(left_son(tree));
}

/* ---------------------------------- min_of_tree ---------------------------------- */

void * max_of_tree (tree_t tree) {
```



```
  if ( tree_is_empty(right_son(tree)) )
    return *(get_elem(tree));
  else
    return max_of_tree(right_son(tree));
}


/* ---------------------------------- add_elem ---------------------------------- */

int add_elem (tree_t *tree, void *elem, int (* f) (void *, void *)) {

  nodo_t *nodo;

  if ( tree_is_empty(*tree) )
    if ( (nodo = malloc(sizeof(nodo_t))) == NULL )
      return 0; /* problema nell'allocazoine della memoria */
    else { /* inizializzazione del nodo */
      (*nodo).elem = elem;
      (*nodo).ns = NULL;
      (*nodo).nd = NULL;
      *tree = nodo;
      return 1;
    }
  else { /* l'albero non è vuoto */
    /* inserire nella giusta posizione l'elemento */
    switch ( f(elem, (**tree).elem) ) {
    case -1 :
      return add_elem(&(**tree).ns, elem, f);
      break;
    case 0 :
      return 0;
      break;
    case 1 :
      return add_elem(&(**tree).nd, elem, f);
      break;
    default :
      return 0;
    }
  }
}

/* ---------------------------------- rem_elem ---------------------------------- */

int rem_elem (tree_t *tree, void *key, int (* f) (void *, void *)) {

  tree_t tmp;

  if ( tree_is_empty(*tree) )
    /* l'albero è vuoto */
    return 0;

  if ( tree_is_empty(left_son(*tree)) ) {
    /* l'albero non ha figli sinistri */
    if ( !f((**tree).elem, key) ) {
      tmp = *tree;
      *tree = right_son(*tree);;
      free(tmp);
      return 1;
    }
    else
      /* ricorsività sul lato destro dell'albero */
      return rem_elem(&(**tree).nd, key, f);
  }
```

```
  if ( tree_is_empty(right_son(*tree)) ) {
    /* l'albero non ha figli destri */
    if ( !f((**tree).elem, key) ) {
      tmp = *tree;
      *tree = left_son(*tree);
      free(tmp);
      return 1;
    }
    else
      /* ricorsività sul lato sinistro dell'albero */
      return rem_elem(&(**tree).ns, key, f);
  }

  /* albero ha entrambi i figli */
  if ( !f((**tree).elem, key) ) {
    (**tree).elem = max_of_tree(left_son(*tree));
    return rem_elem(&(**tree).ns, (**tree).elem, f);
  }
  else
    /* ricorsività sul lato giusto dell'albero */
    switch ( f(key, (**tree).elem) ) {
    case -1 :
      return rem_elem(&(**tree).ns, key, f);
      break;
    case 0 :
      return 0;
      break;
    case 1 :
      return rem_elem(&(**tree).nd, key, f);
      break;
    default :
      return 0;
    }
}

/* ------------------------------ find_elem ------------------------------ */

void ** find_elem (tree_t tree, void *key, int (* f) (void *, void *)) {

  if ( tree_is_empty(tree) )
     return NULL;

  switch ( f(key, *(get_elem(tree))) ) {
  case -1 :
    return find_elem (left_son(tree), key, f);
    break;
  case 0 :
    return get_elem(tree);
    break;
  case 1 :
    return find_elem (right_son(tree), key, f);
    break;
  default :
    return NULL;
  }
}
```

```
/*            Quimera, luglio 2003            *
 *               Versione 1.0.0               *
 *                                            *
 * Dichiarazione dei tipi di variabili e di   *
 * procedure utili alla gestione dei dati in  *
 * memoria tramite l'utilizzo di un albero    *
 * binario ordinato.                          */

#ifndef _DATI_MEMORIA_H
#define _DATI_MEMORIA_H

#include "tree.h" /* libreria per la gestione dell'albero ordinato */

/* ---------------------- Dichiarazione delle procedure ---------------------- */

void inizializza_memoria (void);
/* Requisiti : nessuno                                  *
 * Ruolo: inizializzazione della memoria                */

int memoria_vuota (void);
/* Requisiti : memoria inizializzata                    *
 * Ruolo: Restituisce 1 se la memoria è vuota, 0 altrimenti. */

int aggiungi_oggetto (void *, int (*) (void *, void *));
/* Requisiti : memoria inizializzata                    *
 * Ruolo: aggiunge in memoria l'oggetto se non è        *
 *   presente. Se l'operazione ha avuto successo        *
 *   restituisce 1 altrimenti 0. Per fare cio, ha bisogno    *
 *   di una funzione di comparazione per gli oggetti;   *
 *   questa rende -1, 0, 1 a seconda se il primo oggetto     *
 *   è rispetivamente minore, uguale o maggiore del     *
 *   secondo                                            */

int rimuovi_oggetto (void *, int (*) (void *, void *));
/* Requisiti : memoria inizializzata                    *
 * Ruolo: rimuove in memoria l'oggetto, se l'operazione      *
 *   ha successo restituisce 1 altrimenti 0.            *
 *   Per fare cio, ha bisogno di una funzione di        *
 *   comparazione per gli oggetti; questa rende -1, 0, 1     *
 *   a seconda se il primo oggetto è rispetivamente     *
 *   minore, uguale o maggiore del secondo              */

void ** cerca_oggetto (void *, int (*) (void *, void *));
/* Requisiti : memoria inizializzata e non vuota        *
 * Ruolo: cerca all'interno della memoria l'oggetto, se lo   *
 *   trova restituisce un puntatore sull'oggetto, altrimenti *
 *   restituisce NULL. Per fare cio, ha bisogno di una  *
 *   funzione di comparazione per gli oggetti; questa rende  *
 *   -1, 0, 1 a seconda se il primo oggetto è rispetivamente *
 *   minore, uguale o maggiore del secondo              */


#endif
```

```
/*            Quimera, luglio 2003            *
 *               Versione 1.0.0               *
 *                                            *
 * Definizione dei tipi di variabili e di     *
 * procedure utili alla gestione dei dati in  *
 * memoria tramite l'utilizzo di un albero    *
 * binario ordinato.                          */

#include "dati_memoria.h"

/* variabili globali */

tree_t memoria;

/* ------------------------------ inizializza ------------------------------ */

void inizializza_memoria (void) {

  memoria = new_tree();
}

/* ------------------------------ memoria_vuota ------------------------------ */

int memoria_vuota (void) {

  return tree_is_empty(memoria);
}

/* ------------------------------ aggiungi_oggetto ------------------------------ */

int aggiungi_oggetto (void *oggetto, int (* f) (void *, void *)) {

  return add_elem(&memoria, oggetto, f);
}

/* ------------------------------ rimuovi_oggetto ------------------------------ */

int rimuovi_oggetto (void *oggetto, int (* f) (void *, void *)) {

  return rem_elem(&memoria, oggetto, f);
}

/* ------------------------------ cerca_oggetto ------------------------------ */

void ** cerca_oggetto (void *oggetto, int (* f) (void *, void *)) {

  return find_elem(memoria, oggetto, f);
}
```

```
/*        Quimera, luglio 2003          *
 *           Versione 1.0.0             *
 *                                      *
 * Libreria per la gestione della mutua *
 * esclusione usando il "file locking". *
 *                                      *
 * Dichiarazione dei tipi di variabili  *
 * e di procedure utili alla libreria.  */

#ifndef _LOCK_H
#define _LOCK_H

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

/* ----------------- Dichiarazione delle funzioni e procedure --------------- */

int create_mutex (const char *path_name);
/* Requisiti : percorso del file valido                    *
 * Ruolo: crea la mutua esclusione utilizzando il file     *
 *   locking. Restituisce il descrittore del file o -1 se  *
 *   l'operazione non e' riuscita e errno contiene l'errore  */

int lock_mutex (int fd, short whence, off_t start, off_t len);
/* Requisiti : fd valido                                   *
 * Ruolo: realizza il lock della mutua esclusione usando il  *
 *   file locking. Restituisce 0 se l'operazione e' riuscita *
 *   o -1 nel caso contrario e errno contiene l'errore       *
 *   Se il file e' libero il lock viene acquisito e la       *
 *   funzione ritorna immediatamente; altrimenti la funzione *
 *   si blocchera' fino al rilascio del lock.                *
 *   Valori possibili dei parametri:                         *
 *    whence : da dove calcolare l'offset                    *
 *      SEEK_SET  dall'inizio del file                       *
 *      SEEK_CUR  dalla posizione corrente                   *
 *      SEEK_END  dalla fine del file                        *
 *    start  : fornisce la posizione di partenzadel segmento *
 *             relativa a whence                             *
 *    len    : e' la lunghezza del segmento in byte, 0 per   *
 *             per tutta la lunghezza del file               */

int unlock_mutex (int fd, short whence, off_t start, off_t len);
/* Requisiti : fd valido                                   *
 * Ruolo: realizza l'unlock della mutua esclusione usando il *
 *   file locking. Restituisce 0 se l'operazione e' riuscita *
 *   o -1 nel caso contrario e errno contiene l'errore       *
 *   Valori possibili dei parametri:                         *
 *    whence : da dove calcolare l'offset                    *
 *      SEEK_SET  dall'inizio del file                       *
 *      SEEK_CUR  dalla posizione corrente                   *
 *      SEEK_END  dalla fine del file                        *
 *    start  : fornisce la posizione di partenzadel segmento *
 *             relativa a whence                             *
 *    len    : e' la lunghezza del segmento in byte, 0 per   *
 *             per tutta la lunghezza del file               */

int remove_mutex (const char *path_name);
/* Requisiti : percorso del file valido                    *
 * Ruolo: elimina la mutua esclusione utilizzando il file  *
 *   locking. Restituisce 0 se l'operazione e' riuscita o -1 *
 *   nel caso contrario e errno contiene l'errore            */
```



```
#endif
```

```
/*        Quimera, luglio 2003          *
 *           Versione 1.0.0             *
 *                                      *
 * Libreria per la gestione della mutua *
 * esclusione usando il "file locking". *
 *                                      *
 * Definizione dei tipi di variabili    *
 * e di procedure utili alla libreria.  */

#include "lock.h"

/* ------------------------------ create_mutex -------------------------------- */

int create_mutex (const char *path_name) {

  return open(path_name, O_EXCL | O_CREAT);
}

/* ------------------------------ lock_mutex ---------------------------------- */

int lock_mutex (int fd, short whence, off_t start, off_t len) {

  struct flock lock;

  /* inizializzazione della struttura */
  lock.l_type = F_WRLCK;  /* tipo: read or write */
  lock.l_whence = whence; /* posizione d'inizio */
  lock.l_start = start;   /* inizio della regione dalla posizione specificata */
  lock.l_len = len;       /* lunghezza in bytes della regione */

  return fcntl(fd, F_SETLKW, &lock);
}

/* ------------------------------ unlock_mutex -------------------------------- */

int unlock_mutex (int fd, short whence, off_t start, off_t len) {

  struct flock lock;

  /* inizializzazione della struttura */
  lock.l_type = F_UNLCK;  /* tipo: read or write */
  lock.l_whence = whence; /* posizione d'inizio */
  lock.l_start = start;   /* inizio della regione dalla posizione specificata */
  lock.l_len = len;       /* lunghezza in bytes della regione */

  return fcntl(fd, F_SETLK, &lock);
}

/* ------------------------------ remove_mutex -------------------------------- */

int remove_mutex (const char *path_name) {

  return unlink(path_name);
}
```