

# Algoritmi e strutture dati: rb-sorter

Ferruccio Vitale

## Indice

Indice.....	2
Specifica del problema.....	3
Analisi del problema.....	4
Progettazione dell'algoritmo.....	5
Implementazione dell'algoritmo.....	8
Makefile.....	8
datatypes.h.....	8
arguments.h.....	9
arguments.c.....	9
log.h.....	11
log.c.....	11
rb-tree.h.....	12
rb-tree.c.....	12
rb-sorter.h.....	17
rb-sorter.c.....	17
Testing del programma.....	24
Valutazione della complessità del programma.....	25

## Specifica del problema

Sia dato un semplice database che rappresenta un elenco di studenti che hanno sostenuto un esame. Il database è organizzato sotto forma di file di testo su 3 colonne contenenti informazioni relative a (Cognome, Matricola, Voto) come ad esempio:

Cognome	Matricola	Voto
Bianchi	212	21
Rossi	128	30
Verdi	54	22

Scrivere un programma ANSI C che acquisisce il database da file, ne effettua un ordinamento in base alla chiave primaria (Cognome) o alle chiavi secondarie (Matricola, Voto) sulla base della scelta dell'utente e produce in uscita il database ordinato. L'ordinamento sulle chiavi secondarie deve conservare l'ordine relativo prodotto dalla chiave primaria.

## Analisi del problema

Il problema richiede di ordinare una tabella mediante l'uso di un algoritmo **stabile**, ovvero un procedimento che, in presenza di più chiavi di ordinamento, preservi l'ordine relativo prodotto da altre chiavi; è facilmente intuibile che il problema posto sia **decidibile**, ovvero che esiste un algoritmo che lo risolve in tempo finito e per una qualunque istanza di dati.

Come da specifica, il dato di ingresso per il programma è costituito da un file, quale rappresentazione testuale della tabella del database; il programma impone che il formato del file sia CSV, ovvero delimitato da un determinato separatore di campo.

*Per semplicità, si assume che il file sia sintatticamente corretto, ovvero che non esistano righe (record) che non abbiano i tre campi valorizzati correttamente.*

L'output del programma sarà la medesima tabella, opportunamente formattata, ordinata secondo i criteri che l'utente potrà specificare da riga di comando, ed eventuali messaggi d'errore.

## Progettazione dell'algoritmo

La scelta della struttura dati e del relativo algoritmo tiene conto dei parametri dettati dalla specifica del problema: l'efficacia dell'algoritmo di ordinamento sarà il parametro fondamentale per la scelta del *modus operandi*; l'analisi non terrà conto delle comuni problematiche relative alla gestione di una base dati, quale la gestione degli indici, un'efficiente gestione dello storage o la disponibilità del dato a fronte di *crash* applicativi.

Si è scelto di articolare il programma in tre fasi distinte:

1. analisi dei parametri forniti su riga di comando: di particolare rilievo, la possibilità di definire un criterio di ordinamento;
2. lettura del file contenente i dati e relativa creazione della struttura dati in memoria;
3. stampa dei dati ordinati e successiva deallocazione della struttura dati.

## Struttura dati

Al fine di risolvere il problema fornito in modo ottimale, si è scelto di adottare una struttura dati del tipo **albero rosso nero**, particolare evoluzione dell'albero binario.

La modalità di rappresentazione del dato in memoria non può essere affidata ad una struttura statica quale il vettore<sup>1</sup>: tale soluzione, infatti, risulterebbe inefficiente in termini computazionali, in quanto gli algoritmi di ordinamento di vettori sono caratterizzati, nel caso migliore, da una complessità pari a  $n \cdot \log(n)$ ; non è inoltre noto a priori il numero di record che il vettore dovrebbe contenere, il che renderebbe la soluzione poco flessibile e scalabile.

*La struttura dati originale prevede l'uso di un particolare nodo chiamato "sentinella", in modo tale che non esistano foglie e tutti i nodi puntino alla sentinella: l'implementazione proposta farà uso di un puntatore nullo piuttosto che allocare un nodo.*

## Algoritmo

Il metodo di ordinamento scelto è di tipo **interno**, in quanto il file da ordinare può essere contenuto in memoria: questo approccio permette di accedere direttamente ad un record generico della tabella, riducendo sensibilmente i tempi di ordinamento; al contrario, un ordinamento *esterno*

---

<sup>1</sup> Sebbene un vettore possa essere "reallocato", questo viene classificato quale struttura 'statica'.

prevede che i dati risiedano su disco e si possa accedere ad essi solo in modo sequenziale o al più per grandi blocchi<sup>2</sup>.

Sebbene l'implementazione di un albero rosso-nero risulti complessa, questo offre un eccellente tempo di esecuzione anche nel caso peggiore.

L'implementazione proposta segue il paradigma di massimizzare l'efficienza in termini temporali di esecuzione, a discapito di un maggiore utilizzo di memoria: questa scelta progettuale si basa sulla considerazione che il sistema operativo, sfruttando diverse tecniche di virtualizzazione e gestione della memoria di sistema, metta a disposizione di un programma in esecuzione un quantitativo di memoria praticamente illimitato, al contrario del fattore "tempo" che non può essere riutilizzato e condiviso.

L'algoritmo di ordinamento proposto prevede di memorizzare il record della tabella all'interno di un nodo dell'albero: ne consegue che l'operazione di lettura del file, la relativa creazione dei nodi e l'inserimento degli stessi all'interno dell'albero, generi un albero perfettamente bilanciato ed ordinato secondo i criteri richiesti.

Nella tabella seguente vengono riassunte le principali proprietà, in termini di efficienza di algoritmo:

Proprietà	Valore
Altezza albero	$O(\log n)$
Inserimento	$O(\log n)$
Cancellazione	$O(\log n)$
Ricerca	$O(\log n)$

**Tabella 1 Proprietà di un albero rosso-nero**

*Si fa riferimento ad un albero contenente "n" nodi. I suddetti valori rappresentano il caso peggiore che si può verificare.*

Le operazioni di lettura su un albero rosso-nero non richiedono particolari considerazioni rispetto a quelle utilizzate per gli alberi binari di ricerca, poiché gli alberi rosso-neri sono una loro specializzazione.

## **Prima fase: analisi dei parametri**

Il programma inizierà la sua esecuzione inizializzando le strutture dati e le variabili necessarie al suo funzionamento.

Successivamente verranno analizzati i parametri specificati su riga di comando, necessari per definire il nome del file contenente la tabella, nonché per modificare alcuni comportamenti predefiniti dell'applicativo.

---

<sup>2</sup> Il comando *sort* dei sistemi \*nix è un esempio di ordinamento esterno.

## Seconda fase: lettura del file di input

La lettura del file di input corrisponde alla fase più rilevante dell'applicazione: durante questa fase, in corrispondenza di ogni record letto dal file, verrà creato un nuovo nodo ed effettuato un inserimento nell'albero, qualora questa operazione risulti possibile.

L'implementazione della struttura dati e delle funzioni necessarie al suo mantenimento costituiscono una parte statica ed immutabile del programma; al contrario, la funzione preposta al confronto di due nodi generici tiene conto dei criteri di ordinamento specificati dall'utente.

Si è scelto di creare tre funzioni differenti (*compare\_name*, *compare\_mark* e *compare\_registration*) preposte al confronto di due generici nodi dell'albero, ognuna delle quali tiene conto di uno solo dei tre campi del record; un vettore di puntatori a funzione verrà opportunamente inizializzato durante la prima fase con queste tre funzioni, seguendo l'ordine specificato dai criteri di ordinamento indicati dall'utente; in ultimo, la funzione che dovrà effettuare il confronto tra due nodi dell'albero, scorrerà il vettore ed userà la sotto-funzione specifica.

Le funzioni di ordinamento sono necessarie per individuare il punto esatto dell'albero nel quale inserire il nuovo nodo; il risultato immediato di un inserimento o di una cancellazione può corrispondere ad una violazione di una delle quattro proprietà fondamentali dell'albero rosso-nero: ristabilire tali proprietà richiede un numero limitato di operazioni, che possono avere una complessità costante nel caso medio, logaritmica nel caso peggiore.

## Terza fase: stampa dei risultati

La terza fase si limita ad una ricorsione sull'albero per stampare i nodi presenti ed a una successiva fase di deallocazione della struttura dati utilizzata.

Va sottolineato che la visita ricorsiva dell'albero per la cancellazione di un nodo, è di tipo **non lineare**: non è pertanto possibile ottimizzare ulteriormente l'implementazione a meno di non perdere di leggibilità e semplicità di gestione del codice (al contrario, una ricorsione lineare potrebbe essere sostituita con un'iterazione, portando ad una riduzione della complessità).

Si è scelto di non implementare in forma iterativa la funzione "node\_traverse": sebbene questa forma risulti più efficiente, si vuole dare esempio di visita ricorsiva in ordine simmetrico e posticipato (node\_dealloc).

## Implementazione dell'algoritmo

Di seguito i file che compongono il progetto.

### Makefile

```
CC = gcc
CFLAGS = -ansi -pedantic -Wall -Werror -g
LIBS =
INCLUDES =
SRCS = arguments.c rb-sorter.c rb-tree.c log.c
OBJS = arguments.o rb-sorter.o rb-tree.o log.o

all: rb-sorter

rb-sorter: $(OBJS)
    ${CC} ${CFLAGS} ${INCLUDES} ${LIBS} ${OBJS} -o rb-sorter

.c.o:
    ${CC} ${CFLAGS} ${INCLUDES} -c $<

clean:
    rm -f *.o core *~ rb-sorter output.txt

package:
    tar cvfz rb-sorter.tgz ${SRCS} *.h Makefile

indent:
    ls -l *.ch | xargs indent --no-tabs --original
    make clean
```

### datatypes.h

```
/*
 * datatypes.h
 * rb-sorter
 *
 * Universita' degli studi di Urbino "Carlo Bo"
 * Algoritmi e Strutture dati
 * Professore Valerio Freschi
 * Anno Accademico 2007 - 2008
 *
 * Ferruccio Vitale (22/04/2008)
 * unixo@devzero.it
 */

#ifndef _DATATYPES_H_
#define _DATATYPES_H_

#define XMALLOC(type) (type *) malloc(sizeof(type))
#define XFREE(x) if (x) free(x)

typedef unsigned long ulong_t;
typedef unsigned int uint_t;
typedef enum _color_t { RED, BLACK } color_t;

/*
 * Struttura per rappresentare un record della tabella.
 */
typedef struct _record_t {
    char *name;
    ulong_t registration;
    uint_t mark;
} record_t;
```



```

/*
 * Struttura per rappresentare un nodo dell'albero rosso-nero.
 */
typedef struct _node_t {
    record_t      *data;
    color_t       color;
    struct _node_t *parent;
    struct _node_t *right;
    struct _node_t *left;
} node_t;

/*
 * Puntatore a funzione per il confronto tra due record/nodi.
 */
typedef int      (*sort_func_t) (const record_t *, const record_t *);

/*
 * Struttura per rappresentare l'albero rosso-nero.
 */
typedef struct _tree {
    node_t      *root;
    ulong_t     nodes_count;
    sort_func_t  sort_func;
} tree_t;

/*
 * Puntatore a funzione per la stampa di un nodo dell'albero.
 */
typedef void    (*node_print_func_t) (record_t *);

#endif                                     /* _DATATYPES_H_ */

```

## arguments.h

```

/*
 * arguments.h
 * rb-sorter
 *
 * Universita' degli studi di Urbino "Carlo Bo"
 * Algoritmi e Strutture dati
 * Professore Valerio Freschi
 * Anno Accademico 2007 - 2008
 *
 * Ferruccio Vitale (22/04/2008)
 * unixo@devzero.it
 */
#ifndef _ARGUMENTS_H_
#define _ARGUMENTS_H_

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int      analyze_args(int, char *const *, const char *, char **);

#endif                                     /* _ARGUMENTS_H_ */

```

## arguments.c

```

/*
 * arguments.c
 * rb-sorter
 *
 * Universita' degli studi di Urbino "Carlo Bo"
 * Algoritmi e Strutture dati

```

```

* Professore Valerio Freschi
* Anno Accademico 2007 - 2008
*
* Ferruccio Vitale (22/04/2008)
* unixo@devzero.it
*
*/
#include "arguments.h"

static int      optind = 1;

/*
* La funzione analizza i parametri passati al programma da riga di comando.
* Il set di parametri ammessi e' descritto nella stringa "opts": la
* presenza del carattere ':' implica che il parametro necessita di un
* valore aggiuntivo (in questo caso "optarg" puntera' a questo valore).
*
* @param argc      il numero di parametri passati al programma
* @param argv      Vettore di argomenti
* @param opts      Elenco dei paramtri consentiti
* @param optarg    Valore dell'eventuale parametro opzionale
* @return          Restituisce il carattere che ha riconosciuto
*                  '?' qualora il parametro non sia tra quelli
*                  consentiti
*                  -1  quando sono stati esaminati tutti i parametri
*/
int
analyze_args(int argc, char *const *argv, const char *opts, char **optarg)
{
    static int      sp = 1;
    int             ch,
                   cont = 1;
    char            *pTmp;

    if (sp == 1) {
        if (optind >= argc || argv[optind][0] != '-'
            || argv[optind][1] == '\0') {
            ch = -1;
            cont = 0;
        } else if (strcmp(argv[optind], "--") == 0) {
            optind++;
            ch = -1;
            cont = 0;
        }
    }

    if (cont) {
        ch = argv[optind][sp];
        if (ch == ':' || (pTmp = strchr(opts, ch)) == NULL) {
            if (argv[optind][++sp] == '\0') {
                optind++;
                sp = 1;
            }
            ch = '?';
            cont = 0;
        }
    }

    if (cont) {
        if (*++pTmp == ':') {
            if (argv[optind][sp + 1] != '\0')
                *optarg = &argv[optind++][sp + 1];
            else if (++optind >= argc) {
                sp = 1;
                ch = '?';
            } else
                *optarg = argv[optind++];
            sp = 1;
        } else {
            if (argv[optind][++sp] == '\0') {
                sp = 1;
                optind++;
            }
            optarg = NULL;
        }
    }

    return (ch);
}

```

## log.h

```
/*
 * log.h
 * rb-sorter
 *
 * Universita' degli studi di Urbino "Carlo Bo"
 * Algoritmi e Strutture dati
 * Professore Valerio Freschi
 * Anno Accademico 2007 - 2008
 *
 * Ferruccio Vitale (22/04/2008)
 * unixo@devzero.it
 */

#ifndef _LOG_H_
#define _LOG_H_

#include <stdarg.h>

/*
 * Livello di criticita' di un messaggio prodotto dal programma.
 */
typedef enum { LOG_INFO = 0, LOG_WARNING, LOG_ERROR } severity_t;

int          flag_silent;

void         log_message(severity_t, const char *, ...);

#endif                          /* _LOG_H_ */
```

## log.c

```
/*
 * log.c
 * rb-sorter
 *
 * Universita' degli studi di Urbino "Carlo Bo"
 * Algoritmi e Strutture dati
 * Professore Valerio Freschi
 * Anno Accademico 2007 - 2008
 *
 * Ferruccio Vitale (22/04/2008)
 * unixo@devzero.it
 */

#include "log.h"
#include <stdio.h>

/*
 * Stampa il messaggio passato come input, scegliendo tra stdout e stderr
 * in base alla severita' del messaggio (warning, error, info). La funzione
 * terra' anche in considerazione l'eventuale paramentro "-q" (quiet).
 *
 * @param severity    Un intero che determina la severita' del messaggio
 * @param fmt         La stringa che rappresenta il formato (vedi printf)
 * @param ...         Eventuali parametri aggiuntivi
 */
void
log_message(severity_t severity, const char *fmt, ...)
{
    va_list          ap;
    FILE             *stream;
    char             *str_severity;

    if (!((severity != LOG_ERROR) && (flag_silent == 1))) {
```

```

        switch (severity) {
        case LOG_INFO:
            stream = stdout;
            str_severity = "[INFO] ";
            break;
        case LOG_WARNING:
            stream = stdout;
            str_severity = "[WARN] ";
            break;
        case LOG_ERROR:
            stream = stderr;
            str_severity = "[ERR ] ";
            break;
        }
        fprintf(stream, str_severity);
        va_start(ap, fmt);
        vfprintf(stream, fmt, ap);
        va_end(ap);
    }
}

```

## rb-tree.h

```

/*
 * rb-tree.h
 * rb-sorter
 *
 * Universita' degli studi di Urbino "Carlo Bo"
 * Algoritmi e Strutture dati
 * Professore Valerio Freschi
 * Anno Accademico 2007 - 2008
 *
 * Ferruccio Vitale (22/04/2008)
 * unixo@devzero.it
 */
#ifndef _RB_TREE_H_
#define _RB_TREE_H_

#include "datatypes.h"
#include "log.h"

void      tree_traverse(tree_t *, node_print_func_t);
tree_t    tree_create(sort_func_t);
void      tree_destroy(tree_t *);
node_t    tree_add_node(tree_t * tree, record_t * object);

#endif                                     /* _RB_TREE_H_ */

```

## rb-tree.c

```

/*
 * rb-tree.c
 * rb-sorter
 *
 * Universita' degli studi di Urbino "Carlo Bo"
 * Algoritmi e Strutture dati
 * Professore Valerio Freschi
 * Anno Accademico 2007 - 2008
 *
 * Ferruccio Vitale (22/04/2008)
 * unixo@devzero.it
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include "rb-tree.h"

```

```

/*
 * Dichiarazione delle funzioni
 */
node_t      *node_alloc(record_t *, color_t);
void         node_dealloc(node_t *);
void         node_traverse(node_t *, node_print_func_t);
void         tree_restore_props(tree_t *, node_t *);
void         tree_rotate_left(tree_t *, node_t *);
void         tree_rotate_right(tree_t *, node_t *);

/*
 * Alloca ed inizializza un nodo dell'albero con il record appena letto.
 *
 * @param data      Puntatore al record
 * @param color     Colore del nodo
 * @return          Puntatore al nodo dell'albero
 */
node_t      *
node_alloc(record_t * data, color_t color)
{
    node_t      *new_node = (node_t *) malloc(sizeof(node_t));

    if (new_node) {
        new_node->data = data;
        new_node->color = color;
        new_node->parent = new_node->right = new_node->left = NULL;
    }

    return new_node;
}

/*
 * Dealloca un nodo dell'albero: la funzione e' ricorsiva ed effettua una
 * visita in ordine posticipato dell'albero.
 *
 * @param node      Puntatore al nodo da distruggere
 */
void
node_dealloc(node_t * node)
{
    if (node) {
        node_dealloc(node->right);
        node_dealloc(node->left);
        XFREE(node);
    }
}

/*
 * Visita in ordine simmetrico dell'intero albero, invocando per ogni nodo
 * la funzione 'node_handler'; la funzione viene invocata da
 * 'tree_traverse'.
 *
 * @param node      Puntatore al nodo corrent
 * @param node_handler Puntatore a funzione per la manipolazione del nodo
 */
void
node_traverse(node_t * node, node_print_func_t node_handler)
{
    if (node) {
        node_traverse(node->left, node_handler);
        node_handler(node->data);
        node_traverse(node->right, node_handler);
    }
}

/*
 * Crea un albero rosso-nero: la funzione si occupa di allocare ed
 * inizializzare la struttura dati preposta.
 *
 * @param comp      Puntatore a funzione per il confronto tra due nodi
 * @return          Puntatore al nodo dell'albero
 */
tree_t      *

```

```

tree_create(sort_func_t comp)
{
    tree_t      *tree = (tree_t *) malloc(sizeof(tree_t));
    if (tree) {
        tree->sort_func = comp;
        tree->nodes_count = 0;
        tree->root = NULL;
    }
    return tree;
}

/*
 * Distrugge un albero rosso-nero: la funzione si occupa di rimuovere i nodi
 * dall'albero e deallocare la memoria utilizzata.
 *
 * @param tree      Puntatore alla radice dell'albero
 */
void
tree_destroy(tree_t * tree)
{
    if (tree->root)
        node_dealloc(tree->root);
    tree->root = NULL;
    tree->nodes_count = 0;

    free(tree);
}

/*
 * Inserisce un valore nell'albero: alloca ed inizializza un nodo e
 * ricerca il punto esatto cui aggiungere il nuovo valore.
 *
 * @param tree      Puntatore alla radice dell'albero
 * @param data      Puntatore al record appena letto dal file
 * @return          Puntatore al nodo appena creato
 */
node_t
tree_add_node(tree_t * tree, record_t * data)
{
    node_t      *cur_node,
                *new_node,
                *parent_node;
    int          sort_result;

    /*
     * Se l'albero e' vuoto, creo un nuovo elemento (NERO) ed inizializzo
     * la radice.
     */
    if (!tree->root) {
        new_node = node_alloc(data, BLACK);
        if (new_node) {
            tree->root = new_node;
            tree->nodes_count = 1;
        }
    } else {
        /*
         * L'albero non risulta vuoto: ha inizio l'iterazione per cercare la
         * posizione del nuovo nodo che, in questo caso, sara' ROSSO.
         */
        for (parent_node = cur_node = tree->root; cur_node;) {
            sort_result = (*(tree->sort_func)) (data, cur_node->data);
            parent_node = cur_node;
            cur_node = sort_result >= 0 ? cur_node->right : cur_node->left;
        }

        new_node = node_alloc(data, RED);
        new_node->parent = parent_node;
        if (sort_result >= 0)
            parent_node->right = new_node;
        else
            parent_node->left = new_node;
    }

    /*
     * Incremento il numero dei nodi presenti nell'albero e verifico
     * se e' necessario effettuare una rotazione dell'albero.
     */
}

```

```

        */
        tree->nodes_count++;
        tree_restore_props(tree, new_node);
    }
    return new_node;
}

/*
 * Effettua una rotazione del sottoramo sinistro del nodo.
 *
 * @param tree          Puntatore all'albero
 * @param node          Puntatore al nodo
 */
void
tree_rotate_left(tree_t * tree, node_t * x_node)
{
    node_t      *y_node = x_node->right;

    x_node->right = y_node->left;
    if (y_node->left != NULL)
        y_node->left->parent = x_node;
    y_node->parent = x_node->parent;

    if (!x_node->parent)
        tree->root = y_node;
    else {
        if (x_node == x_node->parent->left)
            x_node->parent->left = y_node;
        else
            x_node->parent->right = y_node;
    }
    y_node->left = x_node;
    x_node->parent = y_node;
}

/*
 * Effettua una rotazione del sottoramo destro del nodo.
 *
 * @param tree          Puntatore all'albero
 * @param node          Puntatore al nodo
 */
void
tree_rotate_right(tree_t * tree, node_t * y_node)
{
    node_t      *x_node = y_node->left;

    y_node->left = x_node->right;
    if (x_node->right != NULL)
        x_node->right->parent = y_node;

    x_node->parent = y_node->parent;
    if (!(y_node->parent))
        tree->root = x_node;
    else {
        if (y_node == y_node->parent->left)
            y_node->parent->left = x_node;
        else
            y_node->parent->right = x_node;
    }
    x_node->right = y_node;
    y_node->parent = x_node;
}

/*
 * A seguito dell'inserimento di un nuovo nodo, una o piu' proprieta'
 * dell'albero potrebbero essere state violate: la funzione verifica se e'
 * necessario effettuare una rotazione per ripristinare tale condizione.
 * Questa funzione viene invocata solo quando nell'albero e' gia' presente
 * almeno un nodo, la radice che, per definizione, e' sempre nera.
 *
 * @param tree          Puntatore all'albero
 * @param node          Puntatore al nodo appena inserito
 */
void

```

```

tree_restore_props(tree_t * tree, node_t * node)
{
    node_t      *curr_node = node;
    node_t      *grandparent;
    node_t      *uncle;

    /*
     * Inizio a scorrere l'albero fintanto che non raggiungo la radice e
     * fintanto che il colore del padre del nodo (appena inserito) e' rosso.
     */
    while (curr_node != tree->root && curr_node->parent->color == RED) {
        grandparent = curr_node->parent->parent;

        if (curr_node->parent == grandparent->left) {
            uncle = grandparent->right;

            if (uncle && uncle->color == RED) {
                curr_node->parent->color = BLACK;
                uncle->color = BLACK;
                grandparent->color = RED;

                curr_node = grandparent;
            } else {
                if (curr_node == curr_node->parent->right) {
                    curr_node = curr_node->parent;
                    tree_rotate_left(tree, curr_node);
                }

                curr_node->parent->color = BLACK;
                grandparent->color = RED;

                /*
                 * Effettua una rotazione a DX del sottoramo del 'nonno'.
                 */
                tree_rotate_right(tree, grandparent);
            }
        } else {
            uncle = grandparent->left;

            if (uncle && uncle->color == RED) {
                curr_node->parent->color = BLACK;
                uncle->color = BLACK;
                grandparent->color = RED;

                curr_node = grandparent;
            } else {
                if (curr_node == curr_node->parent->left) {
                    curr_node = curr_node->parent;
                    tree_rotate_right(tree, curr_node);
                }

                curr_node->parent->color = BLACK;
                grandparent->color = RED;

                /*
                 * Effettua una rotazione a SX del sottoramo del 'nonno'.
                 */
                tree_rotate_left(tree, grandparent);
            }
        }
    }

    /*
     * Mi assicuro che la radice dell'albero sia SEMPRE nera.
     */
    tree->root->color = BLACK;
}

/*
 * Visita dell'intero albero ed esecuzione di 'node_handler' su ogni nodo:
 * vedi la funzione 'node_traverse'.
 */
/*
 * @param tree      Puntatore alla radice dell'albero
 * @param node_hanle Puntatore a funzione
 */
void

```



```

tree_traverse(tree_t * tree, node_print_func_t node_handler)
{
    node_traverse(tree->root, node_handler);
}

```

## rb-sorter.h

```

/*
 * rb-sorter.h
 * rb-sorter
 *
 * Universita' degli studi di Urbino "Carlo Bo"
 * Algoritmi e Strutture dati
 * Professore Valerio Freschi
 * Anno Accademico 2007 - 2008
 *
 * Ferruccio Vitale (22/04/2008)
 * unixo@devzero.it
 *
 * 0.1.0    Prima bozza.
 * 0.1.1    Aggiunta la gestione dei parametri da riga di comando.
 * 0.2.0    Aggiunta la gestione dei criteri di ordinamento, la funzione
 *           'strdup'.
 */

#ifndef _RB_SORTER_H_
#define _RB_SORTER_H_

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "rb-tree.h"
#include "arguments.h"
#include "log.h"

#define VER_MAJOR      0
#define VER_MINOR      2
#define VER_REVISION   0
#define MAX_BUFFER_LENGTH 1024
#define MAX_FIELD_LENGTH 340

/*
 * Variabili globali.
 */
sort_func_t      sort_array[3];
FILE             *output_stream;

#endif /* _RB_SORTER_H_ */

```

## rb-sorter.c

```

/*
 * rb-sorter.c
 * rb-sorter
 *
 * Universita' degli studi di Urbino "Carlo Bo"
 * Algoritmi e Strutture dati
 * Professore Valerio Freschi
 * Anno Accademico 2007 - 2008
 *
 * Ferruccio Vitale (22/04/2008)
 * unixo@devzero.it
 *
 */

#include "rb-sorter.h"

```

```

/*
 * Dichiarazione delle funzioni
 */
char      *strdup(const char *);
int        loadfile(tree_t *, char *, ulong_t *);
void       usage();
void       node_print(record_t *);
int        compare_name(const record_t *, const record_t *);
int        compare_registration(const record_t *, const record_t *);
int        compare_mark(const record_t *, const record_t *);
int        node_diff(const record_t *, const record_t *);
int        set_sort_criteria(const char *);
void       open_output_stream(const char *);
void       close_output_stream(const char *);

/*
 * Inizializza lo stream di output, sul quale scrivere i risultati
 *
 * @param filename    Nome del file
 */
void
open_output_stream(const char *filename)
{
    if (filename)
        output_stream = fopen(filename, "w+");
    else
        output_stream = stdout;
}

/*
 * Chiude lo stream di output
 *
 * @param filename    Nome del file
 */
void
close_output_stream(const char *filename)
{
    if (filename)
        fclose(output_stream);
}

/*
 * Duplica una stringa
 *
 * @param str          Stringa da duplicare
 * @return             Puntatore alla nuova stringa
 */
char      *
strdup(const char *str)
{
    size_t    len;
    char      *copy;

    copy = NULL;
    len = strlen(str) + 1;
    if ((copy = malloc(len))
        memcpy(copy, str, len);
    return (copy);
}

/*
 * Legge il contenuto del file: ogni riga dovra' contenere un
 * record da inserire.
 *
 * @param tree          Albero rosso-nero cui inserire i dati
 * @param filename      Il nome del file da aprire e leggere
 * @return int          Esito dell'operazione
 */
int
loadfile(tree_t * tree, char *filename, ulong_t * rec_count)
{
    FILE      *file;
    char      buffer[MAX_BUFFER_LENGTH];

```

```

char          fields[3][MAX_FIELD_LENGTH];
record_t      *record;
int           result = 0,
            i,
            j,
            w;

if (filename) {
    file = fopen(filename, "r");
    if (file) {
        memset(buffer, 0, MAX_BUFFER_LENGTH);
        while (fscanf(file, "%s", buffer) > 0) {
            memset(fields, 0, 3 * MAX_FIELD_LENGTH * sizeof(char));
            for (j = w = i = 0; i < strlen(buffer); i++) {
                if (buffer[i] != ';')
                    fields[j][w++] = buffer[i];
                else
                    fields[j++][w] = '\\0', w = 0;
            }

            record = (record_t *) malloc(sizeof(record_t));
            record->name = strdup(fields[0]);
            record->registration = atol(fields[1]);
            record->mark = atoi(fields[2]);

            tree_add_node(tree, record);
            memset(buffer, 0, MAX_BUFFER_LENGTH);
            (*rec_count)++;
        }
        fclose(file);
        result = 1;
    } else
        log_message(LOG_ERROR, "impossibile aprire il file '%s'\n",
                    filename);
}
return result;
}

/*
 * Stampa la sinossi del programma
 */
void
usage()
{
    fprintf(stderr,
        "rb-sorter versione %d.%d.%d\n"
        "Uso: rb-sorter [-h] [-s criteri] [-q] [-v] "
        "-i nomefile [-o nomefile]\n"
        "\t-h\t\tStampa questa spiegazione ed esce\n"
        "\t-i nomefile\tLegge il contenuto del database "
        "dal file specificato\n"
        "\t-o nomefile\tScrive nel file specificato "
        "la tabella ordinata\n"
        "\t-q\t\tVisualizza solo messaggi d'errore di elevata criticita'\n"
        "\t-s [cmv]\tSpecifica i criteri di ordinamento\n"
        "\t\t\t(c=cognome, m=matricola, v=voto)\n"
        "\t-v\t\tStampa la version del programma ed esce\n",
        VER_MAJOR, VER_MINOR, VER_REVISION);
    fprintf(stderr,
        "\nEsempio d'uso:\n"
        " - Legge i dati dal file 'database.txt'\n"
        "\t rb-sorter -i database.txt\n"
        " - Legge i dati dal file 'database.txt' "
        "ed ordina per 'cognome' e 'matricola'\n"
        "\t rb-sorter -i database.txt -s c+m\n"
        " - Legge i dati dal file 'database.txt' "
        "e li scrive su 'output.txt'\n"
        "\t rb-sorter -i database.txt -s c+m+v -o output.txt\n");
}

/*
 * Stampa su STDOUT il contenuto di un nodo dell'albero
 *
 * @param severity    Un intero che determina la severita' del messaggio
 * @param fmt         La stringa che rappresenta il formato (vedi printf)

```

```

    * @param ...          Eventuali parametri aggiuntivi
    */
void
node_print(record_t * rec)
{
    fprintf(output_stream, " | %-30s | %09lu | %02d |\n", rec->name,
               rec->registration, rec->mark);
}

/*
 * Confronta due record considerando il campo "cognome"
 */
/*
 * @param a          Puntatore al primo record
 * @param b          Puntatore al secondo record
 * @return           Restituisce l'esito del confronto
 *                  (0=uguale, -1=minore, +1=maggiore)
 */
int
compare_name(const record_t * a, const record_t * b)
{
    int          ret = strcmp(a->name, b->name);

    if (ret < 0)
        ret = -1;
    else if (ret > 0)
        ret = 1;
    return (ret);
}

/*
 * Confronta due record considerando il campo "matricola"
 */
/*
 * @param a          Puntatore al primo record
 * @param b          Puntatore al secondo record
 * @return           Restituisce l'esito del confronto
 *                  (0=uguale, -1=minore, +1=maggiore)
 */
int
compare_registration(const record_t * a, const record_t * b)
{
    int          ret;

    if (a->registration > b->registration)
        ret = 1;
    else if (a->registration < b->registration)
        ret = -1;
    else
        ret = 0;
    return (ret);
}

/*
 * Confronta due record considerando il campo "voto"
 */
/*
 * @param a          Puntatore al primo record
 * @param b          Puntatore al secondo record
 * @return           Restituisce l'esito del confronto
 *                  (0=uguale, -1=minore, +1=maggiore)
 */
int
compare_mark(const record_t * a, const record_t * b)
{
    int          ret;

    if (a->mark < b->mark)
        ret = -1;
    else if (a->mark > b->mark)
        ret = 1;
    else
        ret = 0;
    return (ret);
}

```

```

/*
 * Confronta due record applicando il criterio di ordinamento.
 *
 * @param a      Puntatore al primo record
 * @param b      Puntatore al secondo record
 * @return       Restituisce l'esito del confronto
 *              (0=uguale, -1=minore, +1=maggiore)
 */
int
node_diff(const record_t * a, const record_t * b)
{
    int          ret,
                i;
    sort_func_t  f;

    f = sort_array[0];

    for (ret = i = 0, f = sort_array[0];
        f && (ret == 0) && (i < 3); f = sort_array[++i]) {
        ret = (*f) (a, b);
    }

    return ret;
}

/*
 * Analizza il parametro specificato su riga di comando per il criterio di
 * ordinamento e costruisce un vettore di puntatori a funzione che verranno
 * richiamati dalla funzione "node_diff".
 *
 * @param str      Parametro da riga di comando
 * @return         Esito dell'operazione
 */
int
set_sort_criteria(const char *str)
{
    int          idx,
                i,
                result = 0,
                len;

    if (!str) {
        sort_array[0] = compare_name;
        sort_array[1] = compare_registration;
        sort_array[2] = compare_mark;
        result = 1;
    } else {
        len = strlen(str);
        if (len <= 5) {
            result = 1;
            for (idx = i = 0; (i < strlen(str)) && result; i++) {
                switch (str[i]) {
                    case 'C':
                    case 'c':
                        sort_array[idx++] = compare_name;
                        break;
                    case 'M':
                    case 'm':
                        sort_array[idx++] = compare_registration;
                        break;
                    case 'V':
                    case 'v':
                        sort_array[idx++] = compare_mark;
                        break;
                    case '+':
                        break;
                    default:
                        log_message(LOG_ERROR, "carattere non ammesso '%c'\n",
                                    str[i]);
                        result = 0;
                        break;
                }
            }
        }
    }

    return result;
}

```

```

}

int
main(int argc, char **argv)
{
    char          *input_filename,
                  *output_filename,
                  *optarg,
                  *sort_string;
    tree_t        *tree;
    ulong_t       record_count;
    int           error,
                 ch;

    /*
     * Inizializzazione variabili globali e locali
     */
    input_filename = output_filename = sort_string = NULL;
    flag_silent = error = 0;
    record_count = 0L;
    memset(sort_array, 0, 3 * sizeof(sort_func_t));

    /*
     * Analisi dei parametri specificati da riga di comando.
     */
    while ((ch = analyze_args(argc, argv, "hi:os:qs:v", &optarg)) != -1) {
        switch (ch) {
            case 'i':
                if (*optarg)
                    input_filename = strdup(optarg);
                else
                    error = 1;
                break;
            case 'o':
                if (*optarg)
                    output_filename = strdup(optarg);
                else
                    error = 1;
                break;
            case 'q':
                flag_silent = 1;
                break;
            case 's':
                if (*optarg)
                    sort_string = strdup(optarg);
                else
                    error = 1;
                break;
            case 'v':
                fprintf(stderr, "rb-sorter versione %d.%d.%d\n",
                        VER_MAJOR, VER_MINOR, VER_REVISION);
                error = 2;
                break;
            case 'h':
            default:
                error = 1;
                break;
        }
    }

    if (input_filename && !error && set_sort_criteria(sort_string)) {
        log_message(LOG_INFO, "Creazione struttura dati\n");
        tree = tree_create(node_diff);
        if (tree) {
            open_output_stream(output_filename);
            log_message(LOG_INFO, "Lettura dati dal file '%s'\n",
                        input_filename);
            if (loadfile(tree, input_filename, &record_count)) {
                log_message(LOG_INFO, "Record letti : %lu\n",
                            record_count);
                log_message(LOG_INFO, "Nodi inseriti: %lu\n",
                            tree->nodes_count);
                log_message(LOG_INFO, "Scrittura risultati\n");
                tree_traverse(tree, node_print);
                log_message(LOG_INFO, "Distruzione struttura dati\n");
                tree_destroy(tree);
            }
        }
    }
}

```

```
    }  
    close_output_stream(output_filename);  
    XFREE(input_filename);  
    XFREE(output_filename);  
    XFREE(sort_string);  
} else  
    log_message(LOG_ERROR, "Impossibile allocare memoria\n");  
} else  
    usage();  
  
return 0;  
}
```

## Testing del programma

Unitamente ai sorgenti che compongono il progetto, sono stati inclusi anche alcuni file contenenti dati di esempio; questi rappresentano istanze di input con dimensioni differenti, di modo che sia possibile un test più esaustivo dello strumento proposto.

Di seguito verrà incluso qualche test significativo ed il relativo output del programma; i seguenti test sono stati effettuati usando questi parametri da riga di comando:

<code>-i file</code>	Legge i dati dal file specificato
<code>-o output.txt</code>	Scrive la tabella ordinata nel file specificato
<code>-s criterio</code>	Criterio di ordinamento

### Test 1. lettura da un file inesistente

```
./rb-sorter -i not-exist.txt
[INFO] Creazione struttura dati
[INFO] Lettura dati dal file 'not-exist.txt'
[ERR ] impossibile aprire il file 'not-exist.txt'
```

### Test 2. database di 15 linee, duplicati presenti

```
$ time ./rb-sorter -i database_15_linee.txt -o output.txt -s c+m+v
[INFO] Creazione struttura dati
[INFO] Lettura dati dal file 'database_15_linee.txt'
[INFO] Record letti : 15
[INFO] Nodi inseriti: 15
[INFO] Scrittura risultati
[INFO] Distruzione struttura dati

real    0m0.005s
user    0m0.001s
sys     0m0.003s
```

### Test 3. database con 1,7 milioni di linee, duplicati non presenti

```
$ time ./rb-sorter -i database_1.7M_linee.txt -o output.txt -s c+m+v
[INFO] Creazione struttura dati
[INFO] Lettura dati dal file 'database_1.7M_linee.txt'
[INFO] Record letti : 1699062
[INFO] Nodi inseriti: 1699062
[INFO] Scrittura risultati
[INFO] Distruzione struttura dati

real    0m10.839s
user    0m8.470s
sys     0m0.665s
```



## Valutazione della complessità del programma

Per calcolare il tempo di esecuzione del programma nonché l'ordine di grandezza della complessità, si è scelto di adoperare due approcci distinti: in prima istanza, utilizzando un metodo analitico e, successivamente, uno sperimentale.

### Approccio analitico

Sono stati identificati, all'interno del codice, tre macro blocchi di istruzioni: ad ognuno di questi è stato associato un costo, calcolando dapprima il numero effettivo di passi base eseguiti e derivando in seguito la classe dell'ordine di grandezza.

Il primo blocco, la funzione *loadfile*, legge i dati dal file contenente il database; facendo coincidere il numero di righe del file con il numero di nodi che possibilmente verranno inseriti nell'albero, possiamo esprimere l'intero costo del programma in funzione del numero di record letti.

Il costo del suddetto blocco sarà equivalente al prodotto di  $n$  record per il costo della funzione di inserimento di un nodo nell'albero, ovvero  $O(\log n)$ .

Il secondo ed il terzo blocco del programma equivalgono alla visita dell'intero albero per stamparne i nodi ed alla deallocazione della struttura dati: entrambi hanno una complessità proporzionale al numero di nodi (record) presenti nell'albero, quindi pari ad  $O(n)$ .

Di seguito, il riepilogo di quanto esposto:

FUNZIONE	COSTO
LOAD_FILE	$O(n \log n)$
TREE_TRAVERSE	$O(n)$
TREE_DESTROY	$O(n)$

La complessità totale quindi è pari a  $2 \cdot n + n \cdot \log n$ , assimilabile alla classe di complessità pseudo lineare  $T(n) = O(n \log n)$ .

### Approccio sperimentale

L'approccio empirico proposto si basa sulla misurazione dei tempi di esecuzione del programma, fornendo istanze di input progressivamente maggiori.

*Per ogni istanza di input, il programma è stato eseguito più volte: si è quindi valutato e riportato il valore medio.*

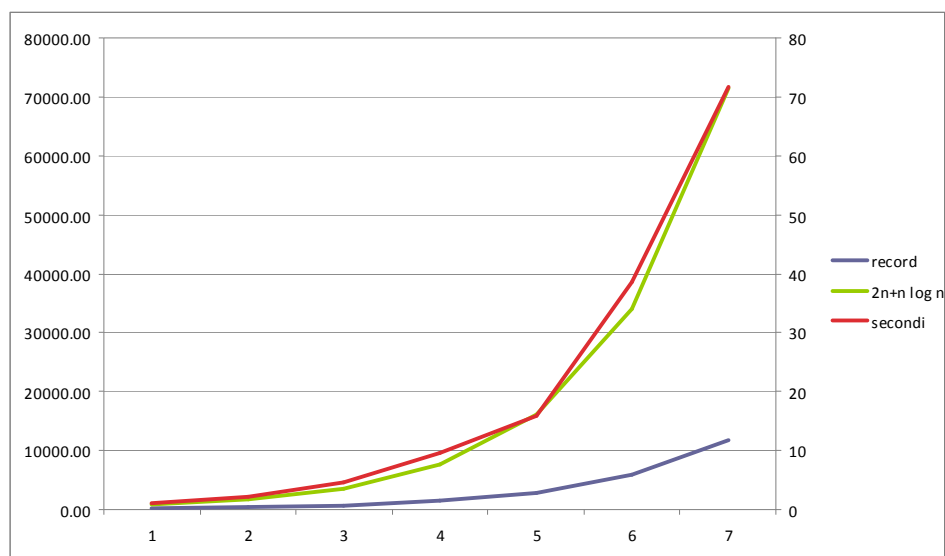
Mettendo in relazione il numero di record processati ed il numero di secondi impiegati, si è notato come l'andamento della complessità

temporale (espressa in secondi) seguisse la curva espressa dalla complessità calcolata con l'approccio analitico.

Di seguito i valori rilevati e loro rappresentazione grafica.

N. RECORD	SECONDI	$2n+n \cdot \text{LOG } n$
183.830	1,022	783.904
367.650	2,164	1678.482
735.300	4,546	3578.311
1.470.600	9,562	7599.317
2.941.200	15,838	16084.024
5.882.400	36,617	33938.827
11.764.800	71,812	71419.213

**Tabella 2 Misurazione tempi d'esecuzione**



**Figura 1 Grafico tempi di esecuzione ed andamento pseudo lineare**