```
/*
 * rbtree.c
 *
 *  Created on: Feb 21, 2016
 *      Author: Stephen
 */

#include <assert.h>
#include <stdlib.h>

#include "bst.h"
#include "rbtree.h"
#include "disk_loc.h"

/*
 * Allocates a new node in the red black tree tree and returns a pointer to it
 * via 'n'.
 *
 * Return 0 on success and 1 on failure.
 */

/* the root */
struct rbtree_node * root;

static int make_rbtree_node(struct disk_location *loc,
                struct rbtree_node **n, struct rbtree_node **parent)
{
        struct rbtree_node *node;

        node = calloc(1, sizeof(*node));
        if (!node) {
                perror("calloc failed");
                return 1;
        }

        node->loc = *loc;
        if(parent != NULL)
            node -> parent = *parent;
        else
                node -> parent = NULL;
        node->color = RED;
        *n = node;

        return 0;
}


struct rbtree_node * insert_rbtree_location(struct rbtree_node **cur, struct rbt
ree_node **parent,
                struct disk_location *loc)
{
        struct rbtree_node *r;
        assert (cur);

        r = *cur;

        if (!r){
                make_rbtree_node(loc, cur, parent);
                if(*cur && (*cur)->parent == NULL){
                        root = *cur;
                }
                return fixup(*cur);
        }

        if (compare_locations(loc, &r->loc) <= 0)
                return insert_rbtree_location(&r->left, &r, loc);

        return insert_rbtree_location(&r->right, &r, loc);
}
```



```
struct rbtree_node* fixup(struct rbtree_node *cur){
        struct rbtree_node * gparent;
        struct rbtree_node * uncle;

        /*fprintf(stderr, "Fixing up (%d,%d)\n", cur->loc.track, cur->loc.sector
);*/

        while(cur-> parent != NULL && cur->parent->color == RED){
                gparent = cur->parent->parent;
                /* if cur's parent is a left child */
                if(gparent -> left != NULL && gparent -> left == cur->parent){
                        uncle = gparent -> right;
                        /* case 1: cur's uncle is red */
                        if(uncle != NULL && uncle->color == RED){
                                /* fprintf(stderr, "case1\n"); */
                                cur->parent->color = BLACK;
                                uncle->color = BLACK;
                                gparent->color = RED;
                                cur = gparent;
                        }
                        else{
                                /* case 2: if cur is a right child */
                                if(cur->parent != NULL && cur->parent->right ==
cur){
                                        /* fprintf(stderr, "case2\n"); */
                                        cur = cur->parent;
                                        rotate_left(cur);
                                }
                                /* case 3 */
                                /* fprintf(stderr, "case3\n"); */
                                cur->parent->color = BLACK;
                                gparent->color = RED;
                                rotate_right(gparent);
                        }
                }
                else{
                        /* if cur's parent is a right child */
                        if(gparent->right != NULL && gparent->right == cur->pare
nt){
                                uncle = gparent -> left;
                                /* case 4: cur's uncle is red */
                                if(uncle != NULL && uncle->color == RED){
                                        /*fprintf(stderr, "case 4\n"); */
                                        cur->parent->color = BLACK;
                                        uncle->color = BLACK;
                                        gparent->color = RED;
                                        cur = gparent;
                                }
                                else{
                                        /* case 5: if cur is a left child*/
                                        if(cur->parent != NULL && cur->parent->l
eft == cur){
                                                /* fprintf(stderr, "case 5\n");
*/
                                                cur = cur->parent;
                                                rotate_right(cur);
                                        }
                                        /* case 6 */
                                        /*fprintf(stderr, "case 6\n"); */
                                        cur->parent->color = BLACK;
                                        gparent->color = RED;
                                        rotate_left(gparent);
                                }
                        }
                }
        }

        if(root != NULL)
```

```
                root->color = BLACK;
        return root;
}

void rotate_left(struct rbtree_node * node){
        struct rbtree_node * q;
        q = node->right;
        if(q != NULL){
                node->right = q->left;
                if(q->left != NULL)
                        q->left->parent = node;
                q->parent = node->parent;
                if(q->parent == NULL){
                        root = q;
                }
                /* set node's parents child accordingly */
                if(node->parent != NULL && node->parent->right == node){
                        node->parent->right = q;
                }
                else if(node->parent != NULL && node->parent->left == node){
                        node->parent->left = q;
                }
                q->left = node;
                node->parent = q;
        }
}

void rotate_right(struct rbtree_node * node){
        struct rbtree_node * p;
        p = node->left;
        if(p != NULL){
                node->left = p->right;
                if(p->right != NULL)
                        p->right->parent = node;

                p->parent = node->parent;
                if(p->parent == NULL){
                        root = p;
                }
                /* set node's parents child accordingly */
                if(node->parent != NULL && node->parent->right == node){
                        node->parent->right = p;
                }
                else if(node->parent != NULL && node->parent->left == node){
                        node->parent->left = p;
                }
                p->right = node;
                node ->parent = p;
        }
}

void print_node(struct rbtree_node * node){
        fprintf(stderr, "(%d,%d,%d)\n", node->loc.track, node->loc.sector, node->c
olor);
}


void remove_rbtree_node(struct rbtree_node **nodep)
{
        struct rbtree_node *z, *pz, *y, *py, *x;

        z = *nodep;

        pz = NULL;

        if (!z->left || !z->right) {
                y = z;
                py = pz;
        } else {
```



```
                 /*'minimum' is sufficient instead of 'successor'
                  because this branch only happens if both the left
                  and right children are non NULL.*/
                y = rb_minimum_with_parent(z->right, z, &py);
        }

        if (y->left)
                x = y->left;
        else
                x = y->right;

        if (!py) {
                *nodep = x;
        } else {
                if (py->left == y)
                        py->left = x;
                else
                        py->right = x;
        }

        if (y != z)
                z->loc = y->loc;

        free(y);
}

/**
 * Get the minimum with parent
 */
struct rbtree_node * rb_minimum_with_parent(struct rbtree_node *node,
                                            struct rbtree_node *prev,
                                            struct rbtree_node **parent)
{
        if (!node)
                return NULL;

        while (node->left) {
                prev = node;
                node = node->left;
        }

        *parent = prev;

        return node;
}



int remove_rbtree_location(struct rbtree_node **root, struct disk_location *loc)
{
        int c;
        struct rbtree_node *r;

        assert(root);
        r = *root;
        if (!r)
                return 0;

        c = compare_locations(loc, &r->loc);
        if (c == 0) {
                remove_rbtree_node(root);
                return 1;
        } else if (c < 0) {
                return remove_rbtree_location(&r->left, loc);
        }

        return remove_rbtree_location(&r->right, loc);
}
```

```
int rbtree_n_after(struct rbtree_node *node, struct disk_location *loc,
                   struct disk_location *locs[], unsigned int n,
                   unsigned int fill)
{
        int c;

        if (!node)
                return fill;

        c = compare_locations(&node->loc, loc);
        if (c < 0) {
                return rbtree_n_after(node->right, loc, locs, n, fill);
        } else if (c >= 0) {
                int f = rbtree_n_after(node->left, loc, locs, n, fill);
                if (f < n) {
                        locs[f] = &node->loc;
                        f += 1;
                        f = rbtree_n_after(node->right, loc, locs, n, f);
                }

                return f;
        }

        return fill;
}


int rbtree_n_before(struct rbtree_node *node, struct disk_location *loc,
                    struct disk_location *locs[], unsigned int n,
                    unsigned int fill)
{
        int c;

        if (!node)
                return fill;

        c = compare_locations(&node->loc, loc);
        if (c > 0) {
                return rbtree_n_before(node->left, loc, locs, n, fill);
        } else if (c <= 0) {
                int f = rbtree_n_before(node->right, loc, locs, n, fill);
                if (f < n) {
                        locs[f] = &node->loc;
                        f += 1;
                        f = rbtree_n_before(node->left, loc, locs, n, f);
                }

                return f;
        }

        return fill;
}

/*void output_rbtree(FILE *outfile, int depth, struct rbtree_node *root)
{
        unsigned int i;
        for (i = 0; i < depth; i += 1)
                fprintf(outfile, " ");
        if (root) {
                output_location(outfile, &root->loc);
                fprintf(outfile, "\n");
                output_tree(outfile, depth + 1, root->left);
                output_tree(outfile, depth + 1, root->right);
        } else {
                fprintf(outfile, "(nil)\n");
        }
}*/
```

```
/*
 * rbtree.h
 *
 *  Created on: Feb 21, 2016
 *      Author: Stephen
 */
#if !defined(_RBTREE_H_)
#define _RBTREE_H_

#include "disk_loc.h"

/* The color of a node. */
enum color { RED, BLACK };


/* A simple binary tree node of disk locations. */
struct rbtree_node {
        struct disk_location loc;
        struct rbtree_node *left;
        struct rbtree_node *right;
        struct rbtree_node *parent;
        enum color color;
};


/*
 * Inserts a location into the rbtree tree.
 *
 * Returns 0 on success and 1 on failure.
 */
struct rbtree_node * insert_rbtree_location(struct rbtree_node **root, struct rb
tree_node **parent, struct disk_location *loc);

/**
 * The fixup routine for rbtree insertion
 */
struct rbtree_node * fixup(struct rbtree_node *cur);

/*
 * Removes the given node from the tree.
 *
 */
void remove_rbtree_node(struct rbtree_node **nodep);

/*
 * Remove the given location from the tree if it is there.
 *
 * Returns 1 if the location was removed and 0 if not (because it was
 * not found).
 */
int remove_rbtree_location(struct rbtree_node **root, struct disk_location *loc)
;
struct rbtree_node * rb_minimum_with_parent(struct rbtree_node *node, struct rbt
ree_node *prev, struct rbtree_node **parent);

/*
 * Gets up to 'n' elements that come directly after (and including)
 * 'loc'.
 */
int rbtree_n_after(struct rbtree_node *node, struct disk_location *loc,
                struct disk_location *locs[], unsigned int n,
                unsigned int fill);

/*
 * Gets up to 'n' elements that come directly before (and including)
 * 'loc'.
 */
int rbtree_n_before(struct rbtree_node *node, struct disk_location *loc,
                 struct disk_location *locs[], unsigned int n,
```



```
                 unsigned int fill);

/*
 * Output the tree to the given file.
 */
void output_rbtree(FILE *outfile, int depth, struct rbtree_node *root);

/**
 * Print a node
 */
void print_node(struct rbtree_node * node);

/**
 * Rotate the tree left around node
 */
void rotate_left(struct rbtree_node * node);

/**
 * Rotate the tree right around node
 */
void rotate_right(struct rbtree_node * node);

#endif /* !_RBTREE_H_ */
```

```
/**
 * \file disk_sched.c
 *
 * A simple disk scheduler.
 *
 * \author eaburns
 * \date 09-08-2010
 */

#include <assert.h>
#include <ctype.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sys/time.h>

#include "disk_loc.h"
#include "list.h"
#include "bst.h"
#include "rbtree.h"

#if !defined(LINE_MAX)
#if !defined(_POSIX2_LINE_MAX)
#define LINE_MAX 4096           /* should be large enough. */
#else
#define LINE_MAX _POSIX2_LINE_MAX
#endif  /* !_POSIX2_LINE_MAX */
#endif  /* !LINE_MAX */

/*
 * Reads at most 'len - 1' bytes of the next token from the input file
 * into the buffer as a string with a null terminator on the end.
 *
 * Returns 0 on success and 1 on failure.  This function fails if a
 * space character of EOF is not read in the next 'len - 1' bytes.
 */
static int next_token(FILE *infile, char buf[], unsigned int len)
{
        unsigned int i;
        char c;

        for (i = 0; i < len - 1; i += 1) {
                c = getc(infile);
                if (c == EOF || isspace(c))
                        break;
                buf[i] = c;
        }
        buf[i] = '\0';

        if (c == EOF && i == 0)
                return EOF;

        if (!isspace(c)) {
                fprintf(stderr, "Buffer is too small\n");
                return 1;
        }

        return 0;
}

/* Gets the time of day in seconds. */
static double get_current_seconds(void)
{
    double sec, usec;
    struct timeval tv;

    if (gettimeofday(&tv, NULL) < 0) {
```



```
        perror("gettimeofday failed");
        exit(EXIT_FAILURE);
    }

    sec = tv.tv_sec;
    usec = tv.tv_usec;

    return sec + (usec / 1000000);
}


/*************************************************************
 * A simple interface for adding, removing and scanning requests on
 * the disk.
 *************************************************************/

/*
 * Adds a request to the given structure.
 *
 * The first paramater is a pointer to the structure cast into a void*
 * and the second parameter is the location to add.  This function
 * should return 0 on success and 1 on error.
 */
typedef int (*add_request_t)(void *, struct disk_location *);

/*
 * Removes a request from the given structure.
 *
 * The first paramater is a pointer to the structure cast into a void*
 * and the second parameter is the location to delete.  This function
 * should return 0 on success and 1 on error.
 */
typedef int (*del_request_t)(void *, struct disk_location *);

/*
 * Scans the structure in the given direction and fills in the array
 * with a given number of elements.
 *
 * The first parameter is a pointer to the structure cast into a
 * void*, the second is the scan direction, the third is the starting
 * location, the fourth is the array to fill with the scanned
 * locations and the final is the number of locations to fill in in
 * the 4th parameter array.  This function should return the number of
 * elements scanned on success or a negative number on error.
 */
typedef int (*scan_t)(void *, enum direction, struct disk_location *,
                      struct disk_location *[], unsigned int);


/* Handles an incoming request by adding it to the data structure. */
static int handle_request(FILE *infile, FILE *outfile, add_request_t add_req,
                          void *data)
{
        int err, ret;
        struct disk_location loc;

        ret = read_location(infile, &loc);
        if (ret == EOF) {
                fprintf(stderr, "Unexpected end of file\n");
                return 1;
        } else if (ret) {
                return 1;
        }

#if !defined(NDEBUG)
        fprintf(stdout, "Adding request ");
        output_location(stdout, &loc);
        fprintf(stdout, "\n");
        /*printf("\n"); */
```

```
#endif  /* !NDEBUG */

        err = add_req(data, &loc);
        if (err) {
                fprintf(outfile, "Failed to cancel request ");
                output_location(outfile, &loc);
                fprintf(outfile, "\n");
                return 1;
        }

        return 0;
}

/* Handles the cancelation of a request. */
static int handle_cancel(FILE *infile, FILE *outfile, del_request_t del_req,
                         void *data)
{
        int err, ret;
        struct disk_location loc;

        ret = read_location(infile, &loc);
        if (ret == EOF) {
                fprintf(stderr, "Unexpected end of file\n");
                return 1;
        } else if (ret) {
                return 1;
        }

#if !defined(NDEBUG)
        printf("Deleting request ");
        output_location(stdout, &loc);
        printf("\n");
#endif  /* !NDEBUG */

        if (!del_req) {
                fprintf(stderr, "WARNENG: Request removal is not supported\n");
                return 0;
        }

        err = del_req(data, &loc);
        if (err) {
                fprintf(outfile, "Failed to cancel request ");
                output_location(outfile, &loc);
                fprintf(outfile, "\n");
                return 1;
        }

        return 0;
}

/* Allocates a scan array, performs the scan and outputs the scan. */
static int do_scan(FILE *outfile, void *data, scan_t scan, enum direction dir,
                   unsigned int num, struct disk_location *loc)
{
        int nscanned;
        struct disk_location **locs;

        locs = malloc(num * sizeof(*locs));
        if (!locs) {
                perror("malloc failed");
                return 1;
        }

        nscanned = scan(data, dir, loc, locs, num);
        if (nscanned >= 0) {
                int i;
                for (i = 0; i < nscanned; i += 1) {
                        output_location(outfile, locs[i]);
                        fprintf(outfile, "\n");
```



```
                }
        }

        free(locs);

        return nscanned < 0;
}

/* Outputs the information about the scan. */
static void output_scan(FILE *outfile, enum direction dir, unsigned int num,
                        struct disk_location *loc)
{
        fprintf(outfile, "scanning %s by %d from ",
                (dir == DOWN ? "down" : "up"), num);
        output_location(outfile, loc);
        fprintf(outfile, "\n");
}


/* Handles a scan. */
static int handle_scan(FILE *infile, FILE *outfile, scan_t scan, void *data)
{
        int ret, err, num;
        char buf[LINE_MAX + 1];
        enum direction dir = UP;
        struct disk_location loc;

        ret = next_token(infile, buf, LINE_MAX + 1);
        if (ret == EOF) {
                fprintf(stderr, "Unexpected end of file\n");
                return 1;
        } else if (ret != 0) {
                return 1;
        }

        if (strncmp("down", buf, LINE_MAX) == 0) {
                dir = DOWN;
        } else if (strncmp("up", buf, LINE_MAX) != 0) {
                fprintf(stderr, "Unknown direction [%s]\n", buf);
                return 1;
        }

        ret = fscanf(infile, " %d", &num);
        if (ret == EOF) {
                fprintf(stderr, "Unexpected end of file\n");
                return 1;
        } else if (ret != 1) {
                fprintf(stderr, "Failed to read length of scan\n");
                return 1;
        }

        ret = read_location(infile, &loc);
        if (ret == EOF) {
                fprintf(stderr, "Unexpected end of file\n");
                return 1;
        } else if (ret) {
                return 1;
        }

        output_scan(outfile, dir, num, &loc);

        err = do_scan(outfile, data, scan, dir, num, &loc);
        if (err) {
                fprintf(stderr, "Scan failed\n");
                return 1;
        }

        return 0;
}
```

```
/*
 * Schedule the I/O requests using a sorted linked list.
 *
 * Return 0 on success and 1 on failure.
 */
static int schedule(FILE *infile, FILE *outfile, add_request_t add_req,
                    del_request_t del_req, scan_t scan, void *data)
{
        int ret, err;
        char buf[LINE_MAX + 1];

        ret = next_token(infile, buf, LINE_MAX + 1);
        while (ret == 0) {
                if (strncmp("request", buf, LINE_MAX) == 0) {
                        err = handle_request(infile, outfile, add_req, data);
                        if (err)
                                return 1;
                } else if (strncmp("cancel", buf, LINE_MAX) == 0) {
                        err = handle_cancel(infile, outfile, del_req, data);
                        if (err)
                                return 1;
                } else if (strncmp("scan", buf, LINE_MAX) == 0) {
                        err = handle_scan(infile, outfile, scan, data);
                        if (err)
                                return 1;
                } else if (strncmp("time", buf, LINE_MAX) == 0) {
                        fprintf(outfile, "time: %f seconds\n",
                                get_current_seconds());
                } else if (buf[0] != '\0') {
                        fprintf(stderr, "Unknown command [%s]\n", buf);
                        return 1;
                }
                ret = next_token(infile, buf, LINE_MAX + 1);
        }

        return 0;
}


/*************************************************************
 * Interface for the linked list.
 *************************************************************/

/*
 * Adds a request to the list.  This adhers to the add_request_t type.
 */
static int list_add_request(void *data, struct disk_location *loc)
{
        return insert_list_location((struct list_node **)data, loc);
}


/*
 * Deletes a request from the list.  This adhers to the del_request_t
 * type.
 */
static int list_del_request(void *data, struct disk_location *loc)
{
        int removed = remove_list_location((struct list_node **) data, loc);

        return !removed;
}


/*
 * Scans the list for a set of locations.  This adhers to the scan_t
 * type.
 */
```



```
static int list_scan(void *data, enum direction dir, struct disk_location *loc,
                     struct disk_location *locs[], unsigned int n)
{
        int left = -1;
        struct list_node **head = data;

        assert(head);

        switch (dir) {
        case UP:
                left = list_n_after(*head, loc, locs, n, n);
                break;
        case DOWN:
                left = list_n_before(*head, loc, locs, n);
                break;
        }

        if (left < 0)
                return left;

        return n - left;
}


/*
 * The scheduler function that uses linked lists.
 */
static int schedule_list(FILE *infile, FILE *outfile)
{
        int err;
        struct list_node *head = NULL;

        err = schedule(infile, outfile, list_add_request,
                       list_del_request, list_scan, (void*) &head);

        free_list(head);
        return err;
}

/*************************************************************
 * Interface for the binary search tree.
 *************************************************************/


static int bst_add_request(void *data, struct disk_location *loc)
{
        return insert_tree_location((struct tree_node **) data, loc);
}

static int bst_del_request(void *data, struct disk_location *loc)
{
        int removed;

        removed = remove_tree_location((struct tree_node **) data, loc);

        return !removed;
}

static int bst_scan(void *data, enum direction dir, struct disk_location *loc,
                    struct disk_location *locs[], unsigned int n)
{
        int fill = -1;
        struct tree_node **root = data;

        assert(root);

        switch (dir) {
        case UP:
                fill = tree_n_after(*root, loc, locs, n, 0);
```

```
                break;
        case DOWN:
                fill = tree_n_before(*root, loc, locs, n, 0);
                break;
        }

        return fill;
}

/*
 * The scheduler function that uses a binary search tree.
 */
static int schedule_bst(FILE *infile, FILE *outfile)
{
        int err;
        struct tree_node *root = NULL;

        err = schedule(infile, outfile, bst_add_request,
                        bst_del_request, bst_scan, (void*) &root);

        free_tree(root);
        return err;
}

/*********************************************************
 * The following functions conform to the interface for add_request_t,
 * del_request_t and scan_t (see above for comments).  You should
 * implement these functions using a red/black tree.
 *
 * There are two samples above using a linked list and using an
 * unbalanced binary tree.
 *********************************************************/

static int rbtree_add_request(void *data, struct disk_location *loc)
{
        /* Insert the node using a standard BST insertion */
        struct rbtree_node * new_root;
        new_root = insert_rbtree_location((struct rbtree_node **) data, NULL, lo
c);
        *((struct rbtree_node**)data) = new_root;
        return 0;
}

static int rbtree_del_request(void *data, struct disk_location *loc)
{
        /* Delete the node using a standard BST insertion */
        struct rbtree_node * new_root = *((struct rbtree_node**)data);
        remove_rbtree_location((struct rbtree_node **) data, loc);
        while(new_root->parent){
                new_root = new_root -> parent;
        }
        *((struct rbtree_node**)data) = new_root;

        return 0;
}

/**
 * Copying code from bst_scan, as traversal in bst and rbtree
 * are equivalent.
 */
static int rbtree_scan(void *data, enum direction dir,
                        struct disk_location *loc,
                        struct disk_location *locs[], unsigned int n)
{
        int fill = -1;
        struct rbtree_node **root = data;


        assert(*root);
```



```
        switch (dir) {
        case UP:
                fill = rbtree_n_after(*root, loc, locs, n, 0);
                break;
        case DOWN:
                fill = rbtree_n_before(*root, loc, locs, n, 0);
                break;
        }

        return fill;
}

/*
 * The scheduler function that uses a binary search tree.
 */
static int schedule_rbtree(FILE *infile, FILE *outfile)
{
        int err;
        struct rbtree_node *root = NULL;

        del_request_t _rbtree_del_request = rbtree_del_request;

        /*
         * Undergraduates can set this to NULL because you do not need
         * to implement request cancellation, however, graduate
         * students should delete the following line once the
         * rbtree_del_request() function has been implemented.
         */
        /*_rbtree_del_request = NULL;*/

        err = schedule(infile, outfile, rbtree_add_request,
                        _rbtree_del_request,
                        rbtree_scan,

                        /* This should be a pointer to your RB tree's
                         * root, it will be passed as the first
                         * argument to the rbtree_add_request(),
                         * rbtree_del_request() and rbtree_scan()
                         * functions. */
                        (void*) &root
                );

        return err;
}

/*********************************************************
 * The main function and friends.
 *********************************************************/


static void usage(void)
{
        fprintf(stderr, "Usage:\n"
                        "disk-sched <data structure> <infile> <outfile>\n");
        exit(EXIT_FAILURE);
}

int main(int argc, const char * argv[])
{
        int err;
        FILE *infile = stdin;
        FILE *outfile = stdout;

        if (argc != 4)
                usage ();

        if (strcmp(argv[2], "-") != 0) {
                infile = fopen(argv[2], "r");
```

```
                if (!infile) {
                        perror("failed to open input file");
                        err = 1;
                        goto out;
                }
        }
        if (strcmp(argv[3], "-") != 0) {
                outfile = fopen(argv[3], "w");
                if (!outfile) {
                        perror("failed to open output file");
                        err = 1;
                        goto out;
                }
        }

        if (strcmp("list", argv[1]) == 0) {
                err = schedule_list(infile, outfile);
        } else if (strcmp("bst", argv[1]) == 0) {
                err = schedule_bst(infile, outfile);
        } else if (strcmp("rbtree", argv[1]) == 0) {
                err = schedule_rbtree(infile, outfile);
        } else {
                fprintf(stderr, "Unknown data structure: %s\n", argv[1]);
                err = 1;
        }

out:
        if (outfile != stdout)
                fclose(outfile);
        if (infile != stdin)
                fclose(infile);

        if (err)
                return EXIT_FAILURE;

        return EXIT_SUCCESS;
}
```

```
CC=gcc
CFLAGS=-ansi -pedantic -Wall -Werror

all: disk-sched disk-sched_debug


disk-sched: rbtree.c bst.c list.c disk_loc.c disk_sched.c
	$(CC) $(CFLAGS) -DNDEBUG -O2 $^ -o disk-sched


disk-sched_debug: rbtree.c bst.c list.c disk_loc.c disk_sched.c
	$(CC) $(CFLAGS) -g $^ -o disk-sched_debug


clean:
	rm -f disk-sched disk-sched_debug
	rm -f *.o
```