multithreaded/hash.c

```c
/*
A hash set with strings as keys
- uses hsearch as the underlying hashmap
*/

#include "hash.h"

/**
 * @brief initialize hash set with an initial size (capacity)
 * @param p HSET*: a pointer to uninitialized memory
 * @param set_size size_t: initial capacity of the hash set to be initialized
 * @return 0 on success; 1 otherwise
 */
int init_hset(HSET *p, size_t set_size)
{
    // create hsearch hash map
    p->hmap = malloc(sizeof(struct hsearch_data));
    memset(p->hmap, 0, sizeof(struct hsearch_data));
    if (hcreate_r(set_size, p->hmap) == 0)
    {
        perror("hcreate\n");
        return 1;
    }

    // array of keys in hash set
    p->elements = (char **)malloc(sizeof(char *) * set_size);
    for (int i = 0; i < set_size; ++i)
    {
        p->elements[i] = NULL;
    }

    p->cur_size = 0;
    p->size = set_size;

    // stack of pointers to strings used as arguments when searching for a string in hsearch;
    //   kept so we can deallocate them at cleanup
    p->ps = malloc(sizeof(PSTACK));
    memset(p->ps, 0, sizeof(PSTACK));
    init_pstack(p->ps, 1);

    return 0;
}

/**
 * @brief check if hash set is at capacity
 * @param p HSET*: (pointer to) the hash set to check
 * @return true if full; false otherwise
 */
```

```c
bool is_full_hset(HSET *p)
{
    return (p->size == p->cur_size);
}

/**
 * @brief check if hash set is empty
 * @param p HSET*: (pointer to) the hash set to check
 * @return true if empty; false otherwise
 */
bool is_empty_hset(HSET *p)
{
    return (p->cur_size == 0);
}

/**
 * @brief add key to the hash set; do nothing if key already exists
 * @param p HSET*: (pointer to) the hash set to add key to
 * @param key char*: key (string) to add
 * @return 0 on success (no error); 1 otherwise (error)
 */
int add_hset(HSET *p, char *key)
{
    if (is_full_hset(p))
    {
        resize_hset(p);
    }

    // add key to hsearch
    ENTRY item;
    // note we create a copy of the parameter key since hsearch may
    //  continue to refer to the passed-in string's memory
    item.key = malloc(strlen(key) * (sizeof(char) + 1));
    memset(item.key, 0, strlen(key) * (sizeof(char) + 1));
    strncpy(item.key, key, strlen(key));
    item.data = NULL;
    ACTION action = ENTER;
    ENTRY *retval = NULL;
    if (hsearch_r(item, action, &retval, p->hmap) == 0)
    {
        perror("hsearch_r\n");
        return 1;
    }

    // add the new string to our array of keys
    //  (so we can transfer them on resize and dellocate them at cleanup)
    p->elements[p->cur_size] = item.key;
    ++p->cur_size;

    return 0;
```

```
}

/**
 * @brief search for the key in the hash set
 * @param p HSET*: (pointer to) the hash set to search
 * @param key char*: key (string) to search
 * @return 1 if the key is found; 0 if the key isn't found
 */
int search_hset(HSET *p, char *key)
{
    // search in hsearch
    ENTRY item;
    // note we create a copy of the parameter key since
    //  hsearch may continue to refer to the passed-in string's memory
    item.key = malloc(strlen(key) * (sizeof(char) + 1));
    memset(item.key, 0, strlen(key) * (sizeof(char) + 1));
    strncpy(item.key, key, strlen(key));
    item.data = NULL;
    ACTION action = FIND;
    ENTRY *retval;
    hsearch_r(item, action, &retval, p->hmap);

    // found key
    if (retval != NULL)
    {
        // add to the pointer stack so we can deallocate at destruction
        //  we don't want to deallocate now, as hsearch may continue to refer
        //  to the string's memory
        push_pstack(p->ps, item.key);
        return 1;
    }

    // did not find key: we can deallocate the key string now
    free(item.key);
    item.key = NULL;
    return 0;
}

/**
 * @brief resize hash set to have greater capacity; maintain existing elements
 * @param p HSET*: (pointer to) the hash set to resize
 * @return 0 on success; 1 otherwise
 */
int resize_hset(HSET *p)
{
    // destroy hsearch
    hdestroy_r(p->hmap);
    free(p->hmap);
    p->hmap = NULL;

```

```c
    // reinitialize hsearch
    p->hmap = malloc(sizeof(struct hsearch_data));
    memset(p->hmap, 0, sizeof(struct hsearch_data));
    if (hcreate_r(HSET_RESIZE_FACTOR * (p->size), p->hmap) == 0)
    {
        perror("hcreate\n");
        return 1;
    }

    // allocate resized array of keys
    size_t old_size = p->size;
    char **old_elements = p->elements;
    p->size = (p->size) * HSET_RESIZE_FACTOR;
    p->elements = (char **)malloc((p->size) * sizeof(char *));

    // transfer old keys over into new array
    p->cur_size = 0;
    for (size_t i = 0; i < old_size; ++i)
    {
        add_hset(p, old_elements[i]);
        free(old_elements[i]);
        old_elements[i] = NULL;
    }
    for (size_t i = old_size; i < p->size; ++i)
    {
        p->elements[i] = NULL;
    }

    // deallocate old array of keys
    free(old_elements);
    old_elements = NULL;

    return 0;
}

/**
 * @brief deconstruct hash set: free all allocated memory
 * @param p HSET*: (pointer to) the hash set to deconstruct
 * @return 0 on success; 1 otherwise
 */
int cleanup_hset(HSET *p)
{
    for (size_t i = 0; i < p->size; ++i)
    {
        if (p->elements[i] != NULL)
        {
            free(p->elements[i]);
            p->elements[i] = NULL;
        }
    }
```

```
199     free(p->elements);
200     p->elements = NULL;
201 
202     hdestroy_r(p->hmap);
203     free(p->hmap);
204     p->hmap = NULL;
205 
206     // clean up the pstack, which cleans up the extra string pointers created during search
207     cleanup_pstack(p->ps);
208     free(p->ps);
209 
210     return 0;
211 }
```