8/15/24, 10:03 AM hash.c

multithreaded/hash.c

```
1
   /*
 2
   A hash set with strings as keys
   - uses hsearch as the underlying hashmap
 3
   */
 4
 5
   #include "hash.h"
 6
7
   /**
8
9
     * @brief initialize hash set with an initial size (capacity)
     * @param p HSET*: a pointer to uninitialized memory
10
     * @param set_size size_t: initial capacity of the hash set to be initialized
11
     * @return 0 on success; 1 otherwise
12
13
14
    int init_hset(HSET *p, size_t set_size)
15
    {
        // create hsearch hash map
16
17
        p->hmap = malloc(sizeof(struct hsearch_data));
        memset(p->hmap, 0, sizeof(struct hsearch data));
18
19
        if (hcreate_r(set_size, p->hmap) == 0)
20
        {
21
            perror("hcreate\n");
22
            return 1;
23
        }
24
25
        // array of keys in hash set
        p->elements = (char **)malloc(sizeof(char *) * set_size);
26
27
        for (int i = 0; i < set size; ++i)</pre>
28
29
            p->elements[i] = NULL;
30
31
32
        p->cur_size = 0;
33
        p->size = set_size;
34
35
        // stack of pointers to strings used as arguments when searching for a string in hsearch;
36
        // kept so we can deallocate them at cleanup
        p->ps = malloc(sizeof(PSTACK));
37
        memset(p->ps, 0, sizeof(PSTACK));
38
39
        init_pstack(p->ps, 1);
40
41
        return 0;
    }
42
43
44
     * @brief check if hash set is at capacity
45
     * @param p HSET*: (pointer to) the hash set to check
46
     * @return true if full; false otherwise
47
48
```

```
bool is_full_hset(HSET *p)
49
50
51
        return (p->size == p->cur size);
52
   }
53
54
   /**
55
     * @brief check if hash set is empty
     * @param p HSET*: (pointer to) the hash set to check
56
     * @return true if empty; false otherwise
57
58
   bool is_empty_hset(HSET *p)
59
60
    {
        return (p->cur size == 0);
61
62
    }
63
    /**
64
     * @brief add key to the hash set; do nothing if key already exists
65
     * @param p HSET*: (pointer to) the hash set to add key to
66
     * @param key char*: key (string) to add
67
     * @return 0 on success (no error); 1 otherwise (error)
68
     */
69
70
    int add_hset(HSET *p, char *key)
71
    {
72
        if (is_full_hset(p))
73
74
            resize_hset(p);
75
76
77
        // add key to hsearch
78
        ENTRY item;
79
        // note we create a copy of the parameter key since hsearch may
        // continue to refer to the passed-in string's memory
80
        item.key = malloc(strlen(key) * (sizeof(char) + 1));
81
        memset(item.key, 0, strlen(key) * (sizeof(char) + 1));
82
83
        strncpy(item.key, key, strlen(key));
84
        item.data = NULL;
85
        ACTION action = ENTER;
86
        ENTRY *retval = NULL;
87
        if (hsearch_r(item, action, &retval, p->hmap) == 0)
88
89
            perror("hsearch_r\n");
            return 1;
90
91
        }
92
93
        // add the new string to our array of keys
94
        // (so we can transfer them on resize and dellocate them at cleanup)
95
        p->elements[p->cur size] = item.key;
        ++p->cur size;
96
97
98
        return 0;
```

```
99
     }
100
     /**
101
102
      * @brief search for the key in the hash set
103
      * @param p HSET*: (pointer to) the hash set to search
104
      * @param key char*: key (string) to search
      * @return 1 if the key is found; 0 if the key isn't found
105
106
107
     int search_hset(HSET *p, char *key)
108
     {
109
         // search in hsearch
         ENTRY item;
110
         // note we create a copy of the parameter key since
111
112
         // hsearch may continue to refer to the passed-in string's memory
113
         item.key = malloc(strlen(key) * (sizeof(char) + 1));
         memset(item.key, 0, strlen(key) * (sizeof(char) + 1));
114
         strncpy(item.key, key, strlen(key));
115
         item.data = NULL;
116
         ACTION action = FIND;
117
118
         ENTRY *retval;
         hsearch r(item, action, &retval, p->hmap);
119
120
         // found key
121
         if (retval != NULL)
122
123
         {
124
             // add to the pointer stack so we can deallocate at destruction
             // we don't want to deallocate now, as hsearch may continue to refer
125
             // to the string's memory
126
             push_pstack(p->ps, item.key);
127
128
             return 1;
129
         }
130
131
         // did not find key: we can deallocate the key string now
132
         free(item.key);
133
         item.key = NULL;
134
         return 0;
135
     }
136
137
138
      * @brief resize hash set to have greater capacity; maintain existing elements
139
      * @param p HSET*: (pointer to) the hash set to resize
      * @return 0 on success; 1 otherwise
140
141
      */
142
     int resize_hset(HSET *p)
143
     {
144
         // destroy hsearch
145
         hdestroy_r(p->hmap);
146
         free(p->hmap);
147
         p->hmap = NULL;
148
```

8/15/24, 10:03 AM

```
149
         // reinitialize hsearch
150
         p->hmap = malloc(sizeof(struct hsearch data));
151
         memset(p->hmap, 0, sizeof(struct hsearch data));
152
         if (hcreate r(HSET RESIZE FACTOR * (p->size), p->hmap) == 0)
153
         {
154
             perror("hcreate\n");
155
             return 1;
156
         }
157
158
         // allocate resized array of keys
159
         size_t old_size = p->size;
160
         char **old elements = p->elements;
161
         p->size = (p->size) * HSET RESIZE FACTOR;
         p->elements = (char **)malloc((p->size) * sizeof(char *));
162
163
164
         // transfer old keys over into new array
         p \rightarrow cur size = 0;
165
         for (size t i = 0; i < old size; ++i)</pre>
166
167
         {
168
             add hset(p, old elements[i]);
             free(old elements[i]);
169
170
             old_elements[i] = NULL;
171
172
         for (size t i = old size; i < p->size; ++i)
173
         {
174
             p->elements[i] = NULL;
175
         }
176
177
         // deallocate old array of keys
178
         free(old_elements);
179
         old elements = NULL;
180
181
         return 0;
182
     }
183
184
185
      * @brief deconstruct hash set: free all allocated memory
      * @param p HSET*: (pointer to) the hash set to deconstruct
186
187
      * @return 0 on success; 1 otherwise
188
189
     int cleanup_hset(HSET *p)
190
     {
191
         for (size_t i = 0; i < p->size; ++i)
192
         {
             if (p->elements[i] != NULL)
193
194
             {
195
                  free(p->elements[i]);
                  p->elements[i] = NULL;
196
197
             }
198
         }
```

```
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 }
```