§1 A hash table for integers with a universal hash function

Listing 1: `un_i_hash.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define BLOCKSIZE 256
5
6  typedef int object_t;
7  typedef int key_t;
8
9  #define MAXP 46337  /* prime, and 46337*46337 < 2147483647 */
10
11 typedef struct l_node {  key_t      key;
12                          object_t  *obj;
13                    struct l_node  *next; } list_node_t;
14
15 typedef struct { int a; int b; int size; } hf_param_t;
16
17 typedef struct { int          size;
18                  list_node_t **table;
19                  int (*hash_function)(key_t, hf_param_t);
20                  hf_param_t hf_param; }    hashtable_t;
21
22
23 list_node_t *currentblock = NULL;
24 int     size_left;
25 list_node_t *free_list = NULL;
26
27 list_node_t *get_node()
28 { list_node_t *tmp;
29   if( free_list != NULL )
30   {  tmp = free_list;
31      free_list = free_list -> next;
32   }
33   else
34   {  if( currentblock == NULL || size_left == 0)
35      {  currentblock =
36                (list_node_t *) malloc( BLOCKSIZE * sizeof(list_node_t) );
37         size_left = BLOCKSIZE;
38      }
39      tmp = currentblock++;
40      size_left -= 1;
41   }
42   return( tmp );
43 }
44
45
46 void return_node(list_node_t *node)
```

```
47 {  node->next = free_list;
48    free_list = node;
49 }
50
51
52 hashtable_t *create_hashtable(int size)
53 {   hashtable_t *tmp; int i;
54     int a, b;
55     int universalhashfunction(key_t, hf_param_t);
56     if( size >= MAXP )
57        exit(-1); /* should not be called with that large size */
58     /* possibly initialize random number generator here */
59     tmp = (hashtable_t *) malloc( sizeof(hashtable_t) );
60     tmp->size = size;
61     tmp->table = (list_node_t **)malloc( size*sizeof(list_node_t *));
62     for(i=0; i<size; i++)
63        (tmp->table)[i] = NULL;
64     tmp->hf_param.a = rand()%MAXP;
65     tmp->hf_param.b = rand()%MAXP;
66     tmp->hf_param.size = size;
67     tmp->hash_function = universalhashfunction;
68     return( tmp );
69 }
70
71 int universalhashfunction(key_t key, hf_param_t hfp)
72 {  return( ((hfp.a*key + hfp.b)%MAXP)%hfp.size );
73 }
74
75
76 object_t *find(hashtable_t *ht, key_t query_key)
77 {  int i; list_node_t *tmp_node;
78    i = ht->hash_function(query_key, ht->hf_param );
79    tmp_node = (ht->table)[i];
80    while( tmp_node != NULL && tmp_node->key != query_key )
81      tmp_node = tmp_node->next;
82    if( tmp_node == NULL )
83       return( NULL ); /* not found */
84    else
85       return( tmp_node->obj ); /* key found */
86 }
87
88 void insert(hashtable_t *ht, key_t new_key, object_t *new_obj)
89 {  int i; list_node_t *tmp_node;
90    i = ht->hash_function(new_key, ht->hf_param );
91    tmp_node = (ht->table)[i];
92    /* insert in front */
93    (ht->table)[i] = get_node();
94    ((ht->table)[i])->next = tmp_node;
95    ((ht->table)[i])->key  = new_key;
96    ((ht->table)[i])->obj  = new_obj;
97 }
```

```
 98
 99 object_t *delete(hashtable_t *ht, key_t del_key)
100 {  int i; list_node_t *tmp_node; object_t *tmp_obj;
101    i = ht->hash_function(del_key, ht->hf_param );
102    tmp_node = (ht->table)[i];
103    if( tmp_node == NULL )
104       return( NULL ); /* list empty, delete failed */
105    if( tmp_node->key == del_key ) /* if first in list */
106    {  tmp_obj = tmp_node->obj;
107       (ht->table)[i] = tmp_node->next;
108       return_node( tmp_node );
109       return( tmp_obj );
110    }
111    /* list not empty, delete not first in list */
112    while( tmp_node->next != NULL
113           &&  tmp_node->next->key != del_key )
114       tmp_node = tmp_node->next;
115    if( tmp_node->next == NULL )
116       return( NULL ); /* not found, delete failed */
117    else
118    {  list_node_t *tmp_node2; /* unlink node */
119       tmp_node2 = tmp_node->next;
120       tmp_node->next = tmp_node2->next;
121       tmp_obj = tmp_node2->obj;
122       return_node( tmp_node2 );
123       return( tmp_obj );
124    }
125 }
126
127
128
129
130 object_t *find_mtf(hashtable_t *ht, key_t query_key)
131 {  int i; list_node_t *front_node, *tmp_node1, *tmp_node2;
132    i = ht->hash_function(query_key, ht->hf_param );
133    front_node = tmp_node1 = (ht->table)[i]; tmp_node2 = NULL;
134    while( tmp_node1 != NULL && tmp_node1->key != query_key )
135    { tmp_node2 = tmp_node1;
136      tmp_node1 = tmp_node1->next;
137    }
138    if( tmp_node1 == NULL )
139       return( NULL ); /* not found */
140    else /* key found */
141    {  if( tmp_node1 != front_node ) /* move to front */
142       {  tmp_node2->next = tmp_node1->next; /* unlink */
143          tmp_node1->next = front_node;
144          (ht->table)[i] = tmp_node1;
145       }
146       return( tmp_node1->obj );
147    }
148 }
```

```
149
150 void list_table(hashtable_t *ht)
151 {  int i; list_node_t *tmp_node;
152    for(i = 0; i< ht->size; i++ )
153    {  printf("|");
154       tmp_node = (ht->table)[i];
155       while( tmp_node != NULL )
156       {  printf("%d ", (tmp_node->key) );
157          tmp_node = tmp_node->next;
158       }
159    }
160 }
161
162
163 int main()
164 {  hashtable_t *ha;
165    char nextop;
166    ha = create_hashtable(20);
167    printf("Made Hashtable\n");
168    while( (nextop = getchar())!= 'q' )
169    { if( nextop == 'i' )
170      { int inskey,  *insobj;
171        insobj = (int *) malloc(sizeof(int));
172        scanf(" %d", &inskey);
173        *insobj = 10*inskey+2;
174        insert( ha, inskey, insobj );
175        printf("  inserted key = %d, object value = %d\n", inskey, *insobj);
176      }
177      if( nextop == 'f' )
178      { int findkey, *findobj;
179        scanf(" %d", &findkey);
180        findobj = find( ha, findkey);
181        if( findobj == NULL )
182          printf("  basic find failed, for key %d\n", findkey);
183        else
184          printf("  basic find successful, found object %d\n", *findobj);
185        findobj = find_mtf( ha, findkey);
186        if( findobj == NULL )
187          printf("  find (mtf) failed, for key %d\n", findkey);
188        else
189          printf("  find (mtf) successful, found object %d\n", *findobj);
190      }
191      if( nextop == 'd' )
192      { int delkey, *delobj;
193        scanf(" %d", &delkey);
194        delobj = delete( ha, delkey);
195        if( delobj == NULL )
196          printf("  delete failed for key %d\n", delkey);
197        else
198          printf("  delete successful, deleted object %d\n", *delobj);
199      }
```

```
200         if( nextop == '?' )
201         {   printf("  Checking table\n");
202             list_table(ha);
203             printf("  Finished Checking table\n");
204         }
205     }
206     return(0);
207 }
```