# Lecture 13
# Hash Dictionaries

15-122: Principles of Imperative Computation (Spring 2023)
Frank Pfenning, Rob Simmons, Iliano Cervesato

In this lecture, we will discuss the data structure of hash tables further and use hash tables to implement a very basic interface of dictionaries. With this lecture, we will also begin to discuss a new separation of concerns. Previously, we have talked a great deal about the distinction between a library's interface (which the client can rely on) and a library's implementation (which should be able to change without affecting a correctly-designed client).

The interface defines not only the *types*, but also the available *operations* on them and the pre- and post-conditions for these operations. For general data structures it is also useful to note the asymptotic complexity of the operations so that potential clients can decide if the interface serves their purpose.

One wrinkle we have not yet encountered is that, in order for a library to provide its services, it may in turn require some operations supplied by the client. Hash tables provide an excellent example for this complication, so we will discuss the interface to hash tables in details before giving the hash table implementation.

For the purposes of this lecture we call the data structures and the operations on them provided by an implementation the *library* and code that uses the library the *client*.
Relating to our learning goals, we have

**Computational Thinking:** We discuss the separation of client interfaces and client implementations.

**Algorithms and Data Structures:** We discuss algorithms for hashing strings.

**Programming:** We revisit the **`char`** data type and use it to consider string hashing. We use this to implement a data structure based on a hash table.

 

# 1 Generic Data Structures — I

So far, all the data structures that we've considered, have always had particular type information that seemed irrelevant. In the implementation of queues, why is it important that we have a queue of *strings* in particular?

```
// typedef ______* queue_t;
bool queue_empty(queue_t Q)                    /* O(1) */
  /*@requires Q != NULL; @*/;
queue_t queue_new()                            /* O(1) */
  /*@ensures \result != NULL; @*/;
void enq(queue_t Q, string x)                  /* O(1) */
  /*@requires Q != NULL; @*/;
string deq(queue_t S)                          /* O(1) */
  /*@requires Q != NULL && !queue_empty(S); @*/ ;
```

It's both wasteful and a potential source of errors to have to rewrite our code if we want our program to use integers (or chars, or pointers to structs, or arrays of strings, ...) instead of strings. A way we deal with this is by creating a type, `elem`, that is used *by* the library but not defined *in* the library:

```
/*** Client interface ***/
// typedef _______ elem;     // Supplied by client

/*** Library interface ***/
// typedef ______* queue_t;
bool queue_empty(queue_t Q)                    /* O(1) */
/*@requires Q != NULL; @*/;
queue_t queue_new()                            /* O(1) */
/*@ensures \result != NULL; @*/;
void enq(queue_t Q, elem x)                    /* O(1) */
/*@requires Q != NULL; @*/;
elem deq(queue_t Q)                            /* O(1) */
/*@requires Q != NULL && !queue_empty(S); @*/ ;
```

The underscores in the library interface, before `queue_t`, mean that the client doesn't know how the abstract type `queue_t` is implemented beyond knowing that it is a pointer. The library is therefore free to change this implementation without breaking any (interface-respecting) client code. The underscores in the *client* interface mean that the *library* doesn't know how the abstract type `elem` is implemented, which means that the client is free to change this implementation without breaking the library. The library's implementation just refers to the `elem` type, which it expects the client to have

already defined, whenever it needs to refer to client data. Therefore, the client code must be split into (at least) two files: one, call it `queue-client.c0`, which defines `elem`, for example as

```
typedef string elem;
```

if we are interested in queues of strings, and the rest of the program, for example `main.c0`. Now, if the file containing the implementation of the queue library is called `queue.c0`, the overall program shall be compiled as

```
% cc0 queue-client.c0 queue.c0 main.c0
```

in order to respect the dependencies.

This approach is still not perfect, because any given program only supports a single type of queue element. We'll start working on that problem in the next lecture.

## 2 Generic Hash Dictionaries

When we implement the dictionary interface with a hash table, we'll call it a *hash dictionary* or `hdict`. Our hash dictionary implementation will be *generic*; it will work regardless of the type of entries to be stored in the table as well as of the type of their keys.

We need to think carefully about which types and functions are provided by the client of the hash dictionary, and which are provided by the library itself. Clearly, the library should determine the type of hash dictionaries:

```
/* library side types */
// typedef ______* hdict_t;
```

That is really the only type provided by the implementation. In addition, the library interface is supposed to provide a few functions:

```
/* library side functions */
hdict_t hdict_new(int capacity)          /* O(1) */
/*@requires capacity > 0; @*/
/*@ensures \result != NULL; @*/ ;

entry hdict_lookup(hdict_t H, key k)     /* O(1) avg. */
/*@requires H != NULL; @*/ ;

void hdict_insert(hdict_t H, entry x)    /* O(1) avg. */
/*@requires H != NULL && x != NULL; @*/ ;
```

The function `hdict_new(int capacity)` takes the initial capacity of the hash table as an argument (which must be strictly positive) and returns a new hash dictionary without any entry in it.

The function `hdict_lookup(hdict_t H, key k)` answers the question of whether an entry with key `k` has been added to the dictionary already and, if the answer is positive, it returns this entry. We will see momentarily how to express these outcomes. This will allows us to add postconditions that the client can use to reason about his/her code.

The last function, `hdict_insert(hdict_t H, entry x)`, adds entry `x` to the dictionary. It too will be extended with a postcondition.

From these decisions we can see that the *client* must provide the type of entries and the type of their keys. Only the client can know what these might be in any particular use of the library. In this implementation, we don't need to know anything about the type `key` of keys. We will however require entries to have pointer type and be non-`NULL`. Doing so allows `hdict_lookup` to return `NULL` to signal that the dictionary does not contain any entry with the requested key. Thus, the client interface specifies that the following types be provided:

```
/* client-side types */
// typedef ______* entry;              // Supplied by client
// typedef ______  key;                // Supplied by client
```

Does the client also need to provide any functions? Yes! The hash table implementation needs functions that can operate on values of the types `key` and `entry` so that it can hash keys, determine whether keys are equal, and extract keys from entries. Since the library is supposed to be generic, the library implementer cannot write these functions; we require the client to provide them.

There are three of these "client-side" functions.

1. When looking up a key, it needs to match this key with the key of entries of interest in the dictionary. To do so, the library needs a function that recovers a key from an entry:

   ```
   key entry_key(entry x)              // Supplied by client
   /*@requires x != NULL; @*/ ;
   ```

   Since `hdict_lookup` returns `NULL` to signal that an entry with the requested key is not present, we disallow `NULL` entries.

   This function allows us to provide `hdict_insert` with a useful postcondition: after inserting an entry, we expect to be able to get it back when looking up its key.

```
void hdict_insert(hdict_t H, entry x)
/*@requires H != NULL && x != NULL; @*/
/*@ensures hdict_lookup(H, entry_key(x)) == x; @*/ ;
```

2. We also need a hash function which maps keys to integers.

   ```
   int key_hash(key k);                    // Supplied by client
   ```

   The result, the *hash value*, can be any integer, so our hash table implementation will have to take both this arbitrary integer and $m$, the size of the hash table's table, into consideration when figuring out which index of the table the key hashes to. For the hash table implementation to achieve its advertised (average-case) asymptotic complexity, the resulting index should have the property that its results are evenly distributed between $0$ and $m$. The hash set implementation will work correctly (albeit slowly) even if it maps every key to $0$.

3. Hash table operations also need to check for the equality of keys in order to be able to tell whether two objects that collide are actually the same or not.

   ```
   bool key_equiv(key k1, key k2);     // Supplied by client
   ```

   With this function, we can add a postcondition to `hdict_lookup`: either it returns `NULL` or the returned entry has the key we were looking for:

   ```
   entry hdict_lookup(hdict_t H, key k)
   /*@requires H != NULL; @*/
   /*@ensures \result == NULL
           || key_equiv(entry_key(\result), k); @*/ ;
   ```

This completes the interface which we now summarize.

```
/************************/
/*** Client interface ***/
/************************/

// typedef ______* entry;               // Supplied by client
// typedef ______  key;                 // Supplied by client

key  entry_key(entry x)                 // Supplied by client
  /*@requires x != NULL; @*/ ;
int  key_hash(key k);                   // Supplied by client
bool key_equiv(key k1, key k2);         // Supplied by client
```

```
/************************/
/*** Library interface ***/
/************************/

// typedef ______* hdict_t;

hdict_t hdict_new(int capacity)
/*@requires capacity > 0; @*/
/*@ensures \result != NULL; @*/ ;

entry hdict_lookup(hdict_t H, key k)
/*@requires H != NULL; @*/
/*@ensures \result == NULL
        || key_equiv(entry_key(\result), k); @*/ ;

void hdict_insert(hdict_t H, entry x)
/*@requires H != NULL && x != NULL; @*/
/*@ensures hdict_lookup(H, entry_key(x)) == x; @*/ ;
```

## 3 A Tiny Client

One sample application is to count word occurrences — say, in a corpus of Twitter data or in the collected works of Shakespeare. In this application, the keys are the words, represented as strings. Entries are pairs of words and word counts, the latter represented as integers.

```
/*****************************/
/* client-side implementation */
/*****************************/
struct wcount {
  string word;     // key
  int count;       // other data
};

// Fulfilling the client interface
typedef struct wcount* entry;
typedef string         key;

key entry_key(entry x)
//@requires x != NULL;
{
  return x->word;
}

int key_hash(key k) {
  return hash_string(k);            /* defined below */
}

bool key_equiv(key k1, key k2) {
  return string_equal(k1, k2);
}
```

## 4 A Universal Hash Function

One question we have to answer is how to hash strings, that is, how to map strings to integers so that the integers are evenly distributed no matter how the input strings are distributed.

We can get access to the individual characters in a string with the function `string_charat(s, i)`, and we can get the integer ASCII value of a **`char`** with the function `char_ord(c)`; both of these are defined in the C0

**string** library. Therefore, our general picture of hashing strings looks like this:

```
int hash_string(string s) {
  int len = string_length(s);
  int h = 0;
  for (int i = 0; i < len; i++)
  //@loop_invariant 0 <= i;
  {
    int ch = char_ord(string_charat(s, i));
    // Do something to combine h and ch
  }
  return h;
}
```

Now, if we don't add anything to replace the comment, the function above will still allow the hash table to work correctly, it will just be very slow because the hash value of every string will be zero.

A slightly better idea is combining `h` and `ch` with addition or multiplication:

```
  for (int i = 0; i < len; i++)
  //@loop_invariant 0 <= i;
  {
    int ch = char_ord(string_charat(s, i));
    h = h + ch;
  }
```

This is still pretty bad, however. We can see how bad by entering the $n = 45,600$ vocabulary words from the Collected Works of William Shakespeare, say, into a table with $m = 22,800$ chains (load factor is 2) and running `ht_stats`:

```
Hash table distribution: how many chains have size...
...0:   21217
...1:   239
...2:   132
...3:   78
...4:   73
...5:   55
...6:   60
...7:   46
...8:   42
...9:   23
...10+: 835
```

```
Longest chain: 176
```

Most of the chains are empty, and many of the chains are very, very long. One problem is that most strings are likely to have very small hash values when we use this hash function. An even bigger problem is that rearranging the letters in a string will always produce another string with the same hash value — so we know that `"cab"` and `"abc"` will always collide in a hash table. Hash collisions are inevitable, but when we can easily predict that two strings have the same hash value, we should be suspicious that something is wrong.

To address this, we can manipulate the value $h$ in some way before we combine it with the current value. Some versions of Java use this as their default string hashing function.

```
for (int i = 0; i < len; i++)
//@loop_invariant 0 <= i;
{
  int ch = char_ord(string_charat(s, i));
  h = 31*h;
  h = h + ch;
}
```

This does much better when we add all the vocabulary strings into the hash table:

```
Hash table distribution: how many chains have size...
...0:   3057
...1:   6210
...2:   6139
...3:   4084
...4:   2151
...5:   809
...6:   271
...7:   53
...8:   21
...9:   4
...10+: 1
Longest chain: 10
```

We can try adding a bit of randomness into this function in a number of different ways. For instance, instead of multiplying by 31, we could multiply by a number generated by the pseudo-random number generator from C0's library:

```
rand_t r = init_rand(0x1337BEEF);
```

```
  for (int i = 0; i < len; i++)
    //@loop_invariant 0 <= i;
    {
      int ch = char_ord(string_charat(s, i));
      h = rand(r) * h;
      h = h + ch;
    }
```

If we look at the performance of this function, it is comparable to the Java hash function, though it is not actually quite as good — more of the chains are empty, and more are longer.

```
Hash table distribution: how many chains have size...
...0:   3796
...1:   6214
...2:   5424
...3:   3589
...4:   2101
...5:   1006
...6:   455
...7:   145
...8:   48
...9:   15
...10+: 7
Longest chain: 11
```

Many other variants are possible; for instance, we could try directly applying the linear congruential generator to the hash value at every step:

```
  for (int i = 0; i < len; i++)
    //@loop_invariant 0 <= i;
    {
      int ch = char_ord(string_charat(s, i));
      h = 1664525 * h + 1013904223;
      h = h + ch;
    }
```

The key goals are that we want a hash function that is very quick to compute and that nevertheless achieves good distribution across our hash table. Handwritten hash functions often do not work well, which can significantly affect the performance of the hash table. Whenever possible, the use of randomness can help to avoid any systematic bias.

## 5 Coherence

Recall the functions `key_hash` and `key_equiv` of our tiny client:

```
int key_hash(key k) {
  return hash_string(k);            /* from hash-string.c0 */
}

bool key_equiv(key k1, key k2) {
  return string_equal(k1, k2);
}
```

In this example, a key is a string.

Consider replacing `key_equiv` with a function that checks that the input keys have the same length:

```
bool key_equiv(key k1, key k2) {
  return string_length(k1) == string_length(k2);
}
```

This does not feel right. Let's see what would happen when we use this `key_equiv` and our original `key_hash` in our hash dictionary. After populating it with a large corpus, for example the works of William Shakespeare, assume we lookup a misspelled word, for example `"amlet"`. The function `key_hash` will convert it to a hash value which will be used to compute the index of a chain. If this chain contains an entry whose key has length 5 (the length of `"amlet"`), it will return this entry, although its key is not `"amlet"`! This result is incorrect.

In a well-designed hash table, keys that are considered equal should hash to the same value, a condition we call *coherence*. As we just saw, whenever the hashing and the equivalence functions are incoherent, there is the concrete risk of our hash dictionaries working incorrectly.

What if we weaken the hash function instead of the equality check? Consider now our original `key_equiv` and a hash function that returns the ASCII code of the first letter of its input (or 0 when passed the empty string):

```
int key_hash(key k) {
  if (string_equal(k, "")) return 0;
  return char_ord(string_charat(k, 0));
}
```

Then every word starting with the same letter will have the same hash value, and therefore will end up at the same table index. Its chains will be very long for a large dictionary, while other indices will be empty. When

looking up a word, once on the appropriate chain, `key_equiv` will correctly either find it or report that it is not in the dictionary. The issue here is performance, not correctness: this choice leads to long chains and therefore slow searches.

The problem with both of these setups is that `key_hash` and `key_equiv` used the information in the key in very different measures. Instead, our original choices were very much in harmony.

## 6 A Fixed-Size Implementation of Hash Tables

For simplicity, we will now write a non-resizing implementation of hash dictionaries. We leave it as an exercise to modify this code to use unbounded arrays to support on-demand resizing of the table.

A non-resizing implementation requires that we can a priori predict a good size, or we will not be able to get the advertised $O(1)$ average time complexity. Choose the size too large and it wastes space and slows the program down due to a lack of locality. Choose the size too small and the load factor will be high, leading to poor asymptotic (and practical) running time.

We start with the type of lists to represent the chains of entries, and the hash table type itself.

```
/*******************************/
/* library-side implementation */
/*******************************/
typedef struct chain_node chain;
struct chain_node {
  entry data;              // != NULL
  chain* next;
};

typedef struct hdict_header hdict;
struct hdict_header {
  int size;                 // 0 <= size
  int capacity;             // 0 < capacity
  chain*[] table;           // \length(table) == capacity
};
```

The first thing after the definition of a data structure is a function to verify its invariants. Besides the invariants noted above we should check the hash index of the key of every entry in each chain stored in $A[i]$ is indeed $i$. (This `is_hdict` function is incomplete.)

```
bool is_hdict(hdict* H) {
  return H != NULL
      && 0 <= H->size
      && 0 < H->capacity
      && is_array_expected_length(H->table, H->capacity);
   /* && there are no NULL entries */
   /* && each entry satisfies its own representation invariants */
   /* && there aren't entries with equal key */
   /* && the number of entries matches the size */
   /* && every entry in H->table[i] hashes to i */
   /* && ... */
}
```

Recall that the test on the length of the array must be inside an annotation, because the `\length` function is not available when the code is compiled without dynamic checking enabled.

In order to check that the keys in a hash dictionary hash to the correct index, we need a way of mapping the hash value returned by `key_hash` to an index of the table. This is a common enough operation that we'll write a helper function:

```
int index_of_key(hdict* H, key k)
//@requires is_hdict(H);
//@ensures 0 <= \result && \result < H->capacity;
{
  return abs(key_hash(k) % H->capacity);
}
```

Allocating a hash table is straightforward.

```
hdict* hdict_new(int capacity)
//@requires capacity > 0;
//@ensures is_hdict(\result);
{
  hdict* H = alloc(hdict);
  H->size = 0;
  H->capacity = capacity;
  H->table = alloc_array(chain*, capacity);
  return H;
}
```

Equally straightforward is searching for an entry with a given key.

```
entry hdict_lookup(hdict* H, key k)
//@requires is_hdict(H);
```

```
//@ensures \result == NULL || key_equiv(entry_key(\result), k);
{
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if (key_equiv(entry_key(p->data), k))
      return p->data;
  }
  return NULL;
}
```

Inserting an entry follows generally the same structure as search. If we find an entry in the correct chain with the same key we replace it. If we find none, we insert a new node at the beginning of the chain.

```
void hdict_insert(hdict* H, entry x)
//@requires is_hdict(H);
//@requires x != NULL;
//@ensures is_hdict(H);
//@ensures x == hdict_lookup(H, entry_key(x));
{
  key k = entry_key(x);
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    //@assert p->data != NULL; // Not given by loop invariant!
    if (key_equiv(entry_key(p->data), k)) {
      p->data = x;
      return;
    }
  }

  // prepend new entry
  chain* p = alloc(chain);
  p->data = x;
  p->next = H->table[i];
  H->table[i] = p;
  (H->size)++;
}
```

## 7 Hash Sets

A *set* is a collection of elements without duplicates. Elementary operations we will be interested in are inserting an element into a set and asking

whether an element is a member of a set.

The basic technology of hash table can also be leveraged to give an efficient implementation of sets. Specifically, we can think of a set as a dictionary where both keys and entries are the elements. This allows us to simplify and specialize the hash dictionary interface into a hash set interface which provides the abstract type `hset` and the above operations.

- We collapse the types `key` and `entry` into a single type that we call `elem`.
- Extracting a key from an entry becomes vacuous. Consequently we can do without an `entry_key` function.
- The lookup function of dictionaries underlies a membership test: it returns `NULL` if the element is not in the set, and the element itself if it is. We can streamline this behavior by having the hash set version, which we call `hset_contains`, return a boolean. This also lets us do without the requirement that the data contained in the hash table be a pointer type: `elem` can be any type.

The resulting interface for hash sets is as follows:

```
/************************/
/*** Client interface ***/
/************************/

// typedef _______ elem;           // Supplied by client
bool elem_equiv(elem x, elem y);   // Supplied by client
int elem_hash(elem x);             // Supplied by client

/*************************/
/*** Library interface ***/
/*************************/

// typedef ______* hset_t;

hset_t hset_new(int capacity)          /* O(1) */
/*@requires capacity > 0; @*/
/*@ensures \result != NULL; @*/ ;

bool hset_contains(hset_t H, elem x)   /* O(1) avg. */
/*@requires H != NULL; @*/ ;

void hset_add(hset_t H, elem x)        /* O(1) avg. */
```

```
/*@requires H != NULL; @*/
/*@ensures hset_contains(H, x); @*/ ;
```

# 8 Exercises

**Exercise 1** (sample solution on page 19)**.** *We want to use the hash dictionaries developed in this chapter to keep track of grades and other student data, and access them by student id. A student id is an integer and the associated grades are a stored in a character array of some fixed length tracked outside the hash dictionary. A student record also includes notes about the student (a string) and whether this student is auditing the course (a boolean).*

*Implement the client interface of hash dictionaries by defining the types* `entry` *and* `key` *and the functions* `entry_key`, `key_hash` *and* `key_equiv`*.*

**Exercise 2** (sample solution on page 19)**.** *We are extending the hash table interface with the new functions*

```
int hdict_size(hdict_t D)
/*@requires D != NULL; @*/
/*@ensures \result >= 0; @*/ ;

entry[] hdict_tabulate(hdict_t D)
/*@requires D != NULL; @*/
/*@ensures \length(\result) == hdict_size(D); @*/ ;
```

*that returns an array with all the entries in the hash table, in some order of your choice. Implement these functions including contracts and annotations as appropriate.*

**Exercise 3** (sample solution on page 20)**.** *Extend the hash table implementation so it dynamically resizes itself when the load factor exceeds a certain threshold. When doubling the size of the hash table you will need to explicitly insert every entry from the old hash table into the new one, because the result of hashing depends on the size of the hash table.*

**Exercise 4** (sample solution on page 22)**.** *Redo the library implementation for a different client interface that has a function* `key_hash(key k, int m)` *that returns a result between 0 (inclusive) and* `m` *(exclusive).*

**Exercise 5** (sample solution on page 23)**.** *Extend the hash table interface with a new function to delete the entry with a given key if present in the table. To be extra ambitious, shrink the size of the hash table once the load factor drops below some minimum, similarly to the way we could grow and shrink unbounded arrays.*

**Exercise 6** (sample solution on page 24)**.** *The hash dictionaries in this lecture relied on entries embedding their keys (which we retrieved with the function* `entry_key`*). A different design is for the hash table to implement a mapping from keys to* values*, where values do not embed the key. The interface of such hash dictionaries is*

```
/******************** Client interface ********************/
// typedef _______ key;         // Supplied by client
// typedef _______ *value;      // Supplied by client
bool key_equiv(key x, key y);   // Supplied by client
int key_hash(key x);            // Supplied by client

/******************** Library interface *******************/
// typedef ______* hdict_t;

hdict_t hdict_new(int capacity)
/*@requires capacity > 0; @*/
/*@ensures \result != NULL; @*/ ;

value hdict_lookup(hdict_t H, key k)
/*@requires H != NULL; @*/ ;

void hdict_insert(hdict_t H, key k, value v)
/*@requires H != NULL; @*/
/*@requires v != NULL; @*/
/*@ensures hdict_lookup(H, k) == v; @*/ ;
```

*In particular,* `hdict_insert` *takes both a key and a value as parameters.*

*Implement this interface.*

**Exercise 7** (sample solution on page 26)**.** *Write an implementation of hash sets based on the interface provided in this chapter.*

## Sample Solutions

**Solution of exercise 1**
This solution assumes that we have written a function `hash_int` that maps integers to values uniformly distributed over the range of all C0 integers.

```
struct  student_record {
  int student_id;
  char[] grades;
  string notes;
  bool auditor;
};

typedef struct student_record* entry;
typedef int key;

key entry_key(entry x)
//@requires x != NULL;
{
  return x->student_id;
}

int key_hash(key k) {
  return hash_int(k);
}

bool key_equiv(key k1, key k2) {
  return k1 == k2;
}
```

**Solution of exercise 2**
The function `hdict_size` simply returns the value of the `size` field of the dictionary.

The function `hdict_tabulate` goes over each position in the underlying hash table, and for each of them it traverses the chain attached to it. It copies the entry in the `data` field of each node in a chain into the output array.

```
int hdict_size(hdict* H)
//@requires is_hdict(H);
//@ensures \result >= 0;
{
  return H->size;
}

entry[] hdict_tabulate(hdict* H)
//@requires is_hdict(H);
//@ensures is_hdict(H) && \length(\result) == hdict_size(H);
{
  entry[] result = alloc_array(entry, H->size);
  int j = 0;
  for (int i = 0; i < H->capacity; i++)
  //@loop_invariant 0 <= i && i <= H->capacity;
  {
    for (chain* p = H->table[i]; p != NULL; p = p->next)
    //@loop_invariant 0 <= j && j <= H->size;
    {
      result[j] = p->data;
      j++;
    }
  }
  return result;
}
```

**Solution of exercise 3**

The first step in implementing a resizing hash dictionary is to decide on the threshold for the load factor beyond which we shall resize the underlying table. This threshold, for example 2, could be hard-coded inside the library implementation. For applications that need careful fine-tuning to achieve to performance, it is however best to let the client decide on what this threshold should be. This is the route We will pursue. To this end, we add a parameter `maxload` to the interface function `hdict_new` and a corresponding field in the header:

```
// Change to the library interface
hdict_t hdict_new(int capacity, int maxload)     // EXTENDED
/*@requires capacity > 0 && maxload > 0; @*/     // EXTENDED
/*@ensures \result != NULL; @*/ ;

// Change to the type definition
typedef struct hdict_header hdict;
struct hdict_header {
  int size;              // 0 <= size
  chain*[] table;        // \length(table) == capacity
  int capacity;          // 0 < capacity
  int maxload;           // 0 < maxload             // ADDED
};
```

It would be good to extend the data structure invariants to check that the load factor of the hash dictionary is no more than `maxload` (see below for how to do this).

Next, we need to update `hdict_insert` to resize the table if the insertion causes the load factor to exceed `maxload`. The naive way to perform this test, by checking whether `H->size/H->capacity >= H->maxload`, will may yield unexpected outcomes as `/` is *integer division*. We are better off multiplying both sides by `H->capacity` (here we ignore possible overflow issues). We call the helper function `hdict_resize`, which takes our hash dictionary and the new capacity as arguments, to carry out the resizing. Altogether, `hdict_insert` is extended with the following two lines:

```
void hdict_insert(hdict* H, entry x) {
  ...
  // Resize
  if (H->size >= H->capacity * H->maxload)
    hdict_resize(H, 2 * H->capacity);
}
```

The last part is to write the function `hdict_resize`. There are several ways to do so. The simplest is to provide our hash dictionary with a new table of the updated capacity, and then use `hdict_insert` to re-insert all the entries of the old table into the new table. Because the new table is bigger than the old table, we will not trigger cascaded resizings. The resulting code is as follows:

```
void hdict_insert(hdict* H, entry x); // forward declaration
void hdict_resize(hdict* H, int new_capacity)
//@requires is_hdict(H);
//@ensures is_hdict(H);
{
  int old_capacity = H->capacity;
  chain*[] old_table = H->table;

  H->capacity = new_capacity;
  H->table =  alloc_array(chain*, H->capacity);
  H->size = 0;
  for (int i = 0; i < old_capacity; i++) {
    for (chain* p = old_table[i]; p != NULL; p = p->next) {
      hdict_insert(H, p->data);
    }
  }
}
```

Note that `hdict_insert` calls `hdict_resize` and `hdict_resize` calls `hdict_insert`: these two functions are *mutually recursive*. Were we to simply place either one before the other, the code would fail to compile. We notify the compiler that a function will be defined after its first use in a file by means of a *forward declaration*, i.e., by writing the function's prototype before the code that first uses it. Here we forward-declared `hdict_insert` just before `hdict_resize`.

One approach to writing `hdict_resize` that does *not* work is to redefine the hash table `H` through a call to `hdict_new`. This does not work because this changes the value contained in the variable `H` to a new address. But this new address is not shared with the caller of `hdict_resize`, here `hdict_insert`, which would never see the resized table (unless `hdict_resize` were to return it).

**Solution of exercise 4**

The main effect of this change to the function `key_hash` is that we can rely on it to carry out the bulk of the functionalities performed by the helper function `index_of_key`: simply pass `H->capacity` as the added argument to `key_hash`. We need to be careful, however, to give `key_hash` sufficiently strong contracts: the bound `m` should be strictly positive, and its output should be between 0 inclusive and `m` exclusive. Here is the prototype of this function pulled as it would appear in the client interface and the updated `index_of_key`:

```
/****************** Client interface *****************/
int  key_hash(key k, int m)             // Supplied by client
  /*@requires m > 0;     @*/
  /*@ensures 0 <= \result && \result < m; @*/ ;
// ...
/****************** Implementation *****************/
int index_of_key(hdict* H, key k)
//@requires is_hdict(H);
//@ensures 0 <= \result && \result < H->capacity;
{
  return key_hash(k, H->capacity);
}
```

**Solution of exercise 5**
We begin by writing down the prototype of the function `hdict_delete` as it appears in the library interface:

```
void hdict_delete(hdict_t H, key k)
/*@requires H != NULL; @*/
/*@ensures hdict_lookup(H, k) == NULL; @*/ ;
```

We are deleting the entry with key `k` (if any) from the dictionary `H`. Observe the postcondition: it says that after performing the deletion, no entry with such key is present in the dictionary.

Implementing deletion is trickier than it appears at first. It is tempting to loop through the chain where `k` belong (at index `i` of the hash table) using a pointer `p` and, once we find `k`, simply set `p` to `p->next`. This doesn't work however: `p` is an alias of the pointer in the chain, and modifying it does not change the pointer in the chain. To avoid this pitfall, we need to update the pointers in the chain themselves (not aliases to them). Once we realize this, we have another problem to handle: to update the first pointer in the chain, we need to update `H->table[i]` (we do this if `k` is in the first node of the chain). For any other pointer, we need to update the `next` field of a node in the chain. With this insight, we could separate out the code that handles the first node of the chain from the code that examines the nodes after it. Handling the first node as a special case in this way would lead to (some) code repetition and likely bugs. A better way to implement the above idea is to maintain a pointer `prev` to the node before the node we are currently examining (the *previous* node). We set `prev` to `NULL` for the previous node of the first node of the chain. Altogether, this leads to the following implementation for `hdict_delete`:

```
void hdict_delete(hdict* H, key k)
//@requires is_hdict(H);
//@ensures hdict_lookup(H, k) == NULL;
//@ensures is_hdict(H);
{
  int i = index_of_key(H, k);
  chain* prev = NULL;
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if (key_equiv(k, entry_key(p->data))) {
      if (prev == NULL)
        H->table[i] = p->next;
      else
        prev->next = p->next;
    }
    prev = p;
    (H->size)--;
  }

  if (H->size <= (H->capacity * H->maxload) / 4)
    hdict_resize(H, H->capacity/2 > 0 ? H->capacity/2 : 1);
}
```

The last two lines take care of resizing the table. As for unbounded arrays, we resize the table when it is less than a quarter "full". Here, our notion of "full" is when it becomes a quarter of the maximum load factor, `H->maxload`, we configured the hash table with (see earlier exercise). When resizing is in order, we halve the size of the table, just like for unbounded arrays. As we do so, we need to be careful about one last thing: the table capacity should never be zero. We do so by using the *ternary operator* `H->capacity/2 > 0 ? H->capacity/2 : 1`. This expression evaluates to `H->capacity/2` if `H->capacity/2 > 0` and to `1` otherwise.

**Solution of exercise 6**

Because in this variant of hash dictionaries keys and values are separate entities, we need to store both as separate fields of in our chain nodes. We update the type definition of chain nodes as follows:

```
typedef struct chain_node chain;
struct chain_node {
  key    key;
  value  value;  // value != NULL
  chain* next;
};
```

We do not allow valid values to be `NULL` because, just like we did in the rest of this chapter, we reserve `NULL` to signal that a value is not present in the dictionary.

The main changes in the rest of the library involve the functions `hdict_lookup` and `hdict_insert`. In the case of `hdict_lookup`, we compare the target key `k` directly with the `key` field of each chain node, and when found we return the `value` fields. (As usual, we return `NULL` if this key was not found.

```
value hdict_lookup(hdict* H, key k)
//@requires is_hdict(H);
{
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if (key_equiv(p->key, k)) {
      return p->value;
    }
  }
  return NULL;
}
```

Aside from taking a key and a value as separate parameters, `hdict_insert` is updated similarly.

```
void hdict_insert(hdict* H, key k, value v)
//@requires is_hdict(H);
//@requires v != NULL;
//@ensures is_hdict(H);
//@ensures hdict_lookup(H, k) == v;
{
  int i = index_of_key(H, k);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if (key_equiv(p->key, k)) {
      p->value = v;  // Override previous value for given key
      return;
    }
  }

  // prepend new entry
  chain* p = alloc(chain);
  p->key = k;
  p->value = v;
  p->next = H->table[i];
  H->table[i] = p;
  (H->size)++;
}
```

**Solution of exercise 7**
Since a hash set is essentially a hash dictionary where keys and entries are identified with the set elements, we update the contents of chain nodes to contain data of type `elem`, the type of the elements of the set.

```
typedef struct chain_node chain;
struct chain_node {
  elem data;
  chain* next;
};
```

We update lookup into the function `hset_contains` that takes an element (of type `elem`) as input (instead of a key), and returns `true` if it is found in the hash set (instead of the entry where it appears) and `false` if it doesn't (instead of `NULL`). Since we do not need `NULL` to signal a non-existent entry, elements do not need to have pointer type. This means that the data representation function `is_hset` (not shown) does not need to check that the chain node have non-`NULL` `data` fields.

```
bool hset_contains(hset* H, elem x)
//@requires is_hset(H);
{
  int i = index_of_elem(H, x);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if (elem_equiv(p->data, x))
      return true;
  }
  return false;
}
```

The insertion function, `hset_add`, is essentially the same as `hdict_insert` for hash dictionaries (one different, also seen in `hset_contains`, is that we do not need to extract keys from entries since they are the same).

```
void hset_add(hset* H, elem x)
//@requires is_hset(H);
//@ensures is_hset(H);
//@ensures hset_contains(H, x);
{
  int i = index_of_elem(H, x);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    if (elem_equiv(p->data, x)) {
      p->data = x;
      return;
    }
  }

  // prepend new element
  chain* p = alloc(chain);
  p->data = x;
  p->next = H->table[i];
  H->table[i] = p;
  (H->size)++;
}
```