Lecture Notes on Interfaces

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 14
October 15, 2013

1 Introduction

The notion of an *interface* to an implementation of an abstract data type or library is an extremely important concept in computer science. The interface defines not only the *types*, but also the available operations on them and the pre- and postconditions for these operations. For general data structures it is also important to note the asymptotic complexity of the operations so that potential clients can decide if the data structure serves their purpose.

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*.

What makes interfaces often complex is that in order for the library to provide its services it may in turn require some operations provided by the client. Hash tables provide an excellent example for this complexity, so we will discuss the interface to hash tables in details before giving the hash table implementation. Binary search trees, discussed in Lecture 15 provide another excellent example.

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.

2 Generic Hash Tables

We call hash tables *generic* because the implementation should work regardless of the type of keys or elements to be stored in the table.

We start with the types. The implementations of which types are provided by the library? Clearly, the type of hash tables.

```
/* library side types */
typedef ___ ht;
```

where we have left it open for now (indicated by `___`) how the type `ht` of hash tables will eventually be defined. That is really the only type provided by the implementation. In addition, it is supposed to provide three functions:

```
/* library side functions */
ht ht_new(int capacity)
//@requires capacity > 0;
  ;
elem ht_lookup(ht H, key k);    /* O(1) avg. */
void ht_insert(ht H, elem e)    /* O(1) avg. */
//@requires e != NULL;
  ;
```

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

The function `ht_lookup(ht H, key k)` searches for an element with key k in the hash table H. If such an element exists, it is returned. If it does not exist, we return `NULL` instead.

From these decisions we can see that the *client* must provide the type of keys and the type of elements. Only the client can know what these might be in any particular use of the library. In addition, we observe that `NULL` must be a value of type `elem`.

The function `ht_insert(ht H, elem e)` inserts an element `e` into the hash table `H`, which is changed as an effect of this operation. But `NULL` cannot be a valid element to insert, because otherwise the client would not know how to interpret a return value `NULL` for `ht_search`. We therefore require e not to be null.

To summarize the types we have discovered will have to come from the client:

```
/* client-side types */
typedef ___* elem;
typedef ___ key;
```

We have noted the fact that `elem` must be a pointer by already filling in the `*` in its definition. Keys, in contrast, can be arbitrary.

Does the client also need to provide any functions? Yes! The hash table implementation needs functions that can operate on values of the types `elem` and `key`, so that it can hash and compare keys and find the key of an element. Since the library is supposed to be generic, the library implementors cannot write these functions; we require the client to provide them.

There are three of these "client-side" functions. First, and most obviously, we need a hash function which maps keys to integers.

```
/* client-side functions */
int hash(key k);
```

The result can be any intenger, so our hash table implementation will have to take this arbitary integer and m, the size of the hash table's table. For the hash table implementation to achieve its advertised (average-case) asymptotic complexity, the hash function should have the property that its results are evenly distributed between 0 and m. Interestingly, it will work correctly (albeit slowly), as long as `hash` satisfies its contract even, for example, if it maps every key to 0.

Now recall how lookup in a hash table works. We map the key to an integer and retrieve the chain of elements stored in this slot in the array. Then we walk down the chain and compare keys of the stored elements with the lookup key. This requires the client to provide two additional operations: one to compare keys, and one to extract a key from an element.

```
/* client-side functions */
bool key_equal(key k1, key k2);

key elem_key(elem e)
//@requires e != NULL;
  ;
```

Key extraction works only on elements that are not null.

This completes the interface which we now summarize.

```
/*************************/
/* client-side interface */
/*************************/
typedef ___* elem;
typedef ___ key;

int hash(key k);
bool key_equal(key k1, key k2);
key elem_key(elem e)
//@requires e != NULL;
  ;

/*************************/
/* library side interface */
/*************************/
typedef struct ht_header* ht;

ht ht_new(int capacity)
//@requires capacity > 0;
  ;
elem ht_search(ht H, key k);    /* O(1) avg. */
void ht_insert(ht H, elem e)    /* O(1) avg. */
//@requires e != NULL;
  ;
int ht_size(ht H);              /* O(1) */
void ht_stats(ht h);
```

The function `ht_size` reports the total number of elements in the array (remember that the load factor is the size n divided by the capacity m). The function `ht_stats` has no effect, but prints out a histogram reporting how many chains in the hash table are empty, how many have length 1, how many have length 2, and so on. For a hashtable to have good performance, chains should be generally short.

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. Data elements are pairs of words and word counts, the latter represented as integers.

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

int hash(string s) {
  return hash_string(s);     /* from hash-string.c0 */
}

bool key_equal(string s1, string s2) {
  return string_equal(s1, s2);
}

string elem_key(struct wcount* wc) {
  return wc->word;
}
```

We can now fill in the types in the client-side of the interface.

```
typedef struct wcount* elem;
typedef string key;
```

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 characers in a string with the `string_charat(s, i)` function, and we can get the integer ASCII value of a `char` with the `char_ord(c)` function; 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 hashtable 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 running entering the $n = 45,600$ news vocabulary words from Homework 2 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 news 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 pseudorandom 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 A Fixed-Size Implementation of Hash Tables

The implementation of hash tables we wrote in lecture did not adjust their size. This requires that we can a priori predict a good size, or we will not be able to get the advertized $O(1)$ 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 elements, and the hash table type itself.

```
/*******************************/
/* library-side implementation */
/*******************************/
struct list_node {
  elem data;                    /* data != NULL */
  struct list_node* next;
};
typedef struct list_node list;

struct ht_header {
  int size;                     /* size >= 0 */
  int capacity;                 /* capacity > 0 */
  list*[] 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 that each data value in each chain in the hash table should be non-null and the hash value of the key of every element in each chain stored in $A[i]$ is indeed i. (This `is_ht` function is incomplete.)

```
bool is_ht(ht H) {
  if (H == NULL) return false;
  if (!(H->size >= 0)) return false;
  if (!(H->capacity > 0)) return false;
  //@assert \length(H->table) == H->capacity;
  /* check that each element of table is a valid chain */
  /* includes checking that all elements are non-null */
  return true;
}
```

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.

Allocating a hash table is straightforward.

```
ht ht_new(int capacity)
//@requires capacity > 0;
//@ensures is_ht(\result);
{
  ht H = alloc(struct ht_header);
  H->size = 0;
  H->capacity = capacity;
  H->table = alloc_array(list*, capacity);
  /* Every cell in H->table is initialized to NULL */
  return H;
}
```

Equally straightforward is searching for an element with a given key. We omit an additional loop invariant and add an assertion that should follow from it instead.

```
elem ht_lookup(ht H, key k)
//@requires is_ht(H);
{
  int i = abs(hash(k) % H->capacity);
  list* p = H->table[i];
  while (p != NULL)
    // loop invariant: p points to a chain (no NULL data)
    {
      //@assert p->data != NULL;
      if (key_equal(elem_key(p->data), k))
        return p->data;
      else
        p = p->next;
    }
  /* not in list */
  return NULL;
}
```

We can extract the key from the element `l->data` because the data can not be null in a valid hash table. (Think: how would we phrase this as a loop invariant?)

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

```
void ht_insert(ht H, elem e)
//@requires is_ht(H);
//@requires e != NULL;
//@ensures is_ht(H);
//@ensures ht_lookup(H, elem_key(e)) != NULL;
{
  key k = elem_key(e);
  int i = abs(hash(k) % H->capacity);

  chain* p = H->table[i];
  while (p != NULL)
    // loop invariant: p points to a chain (no NULL data)
    {
      //@assert p->data != NULL;
      if (key_equal(elem_key(p->data), k))  {
        /* overwrite existing element */
        p->data = e;
        return;
      } else {
        p = p->next;
      }
    }
  //@assert p == NULL;
  /* prepend new element */
  chain* q = alloc(struct chain_node);
  q->data = e;
  q->next = H->table[i];
  H->table[i] = q;
  (H->size)++;
  return;
}
```

Exercises

Exercise 1 *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 element from the old hash table into the new one, because the result of hashing depends on the size of the hash table.*

Exercise 2 *Redo the library implementation for a different client interface that has a function* `hash(key k, int m)` *that returns a result between 0 (inclusive) and* m *(exclusive).*

Exercise 3 *Extend the hash table interface with new functions* `ht_size` *that returns the number of elements in a table and* `ht_tabulate` *that returns an array with the elements in the hash table, in some arbitrary order.*

Exercise 4 *Complete the client-side code to build a hash table containing word frequencies for the words appearing in Shakespeare's collected works. You should build upon the code in Assignment 2.*

Exercise 5 *Extend the hash table interface with a new function to delete an element with a given key from 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.*