Lecture Notes on Interfaces

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 14
March 1, 2011

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 they 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 provides another excellent example.

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 m)
//@requires m > 0;
  ;
elem ht_search(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 m)` takes the initial size of the hash table as an argument (which must be strictly positive) and returns a new hash table without any elements.

The function `ht_search(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 we would know whether the return value `NULL` for `ht_search` means that an element is present or not. 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! Any function the hash table requires which must understand the implementations of the type `elem` and `key` must be provided by the client, since the library is supposed to be generic.

It turns out there are three. First, and most obviously, we need a hash function which maps keys to integers. We also provide the hash function with a modulus, which will be the size of array in the hash table implementation.

```
/* client-side functions */
int hash(key k, int m)
//@requires m > 0;
//@ensures 0 <= \result && \result < m;
  ;
```

The result must be in the range specified by m. For the hash table implementation to achieve its advertised 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 search 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, int m)
//@requires m > 0;
//@ensures 0 <= \result && \result < m;
  ;

bool key_equal(key k1, key k2);

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

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

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

3 A Tiny Client

Our sample application is to count word occurrences 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, int m) {
  return hash_string(s, m);     /* 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 string to integers so that the integers are evenly distributed now matter how the input strings are distributed.

Pseudorandom number generators satisfy a similar criterion. They have to generate numbers that are uniformly distributed over the range of integers, here -2^{31} to $2^{31}-1$. Their interface is pretty simple:

```
/* library file rand.h0 */
typedef struct rand* rand_t;
rand_t init_rand (int seed);
int rand(rand_t gen);
```

One can generate a random number generator (type `rand_t`) by initializing it with an arbitrary seed. Then we can generate a sequence of random numbers by repeatedly calling `rand` on such a generator.

We now show a so-called linear congruential pseudorandom number generator. It stores a last random number (or the seed, at the start) and generates the next number by just one multiplication and one addition. It exploits modular arithmetic, and the trick for this kind of generator is to find a good multiplier and summand. This kind of generator is fine for random testing or (indeed) the basis for a hashing function, but the results are too predictable to use it for cryptographic purposes such as encrypting a message.

```
/* library file rand.c0 */
struct rand {
  int seed;
};

rand_t init_rand (int seed) {
  rand_t gen = alloc(struct rand);
  gen->seed = seed;
  return gen;
}

int rand(rand_t gen) {
  gen->seed = gen->seed * 1664525 + 1013904223;
  return gen->seed;
}
```

In particular, a linear congruential generator will sometimes have repeating patterns in the lower bits. If one wants numbers from a small range it is better to use the higher bits of the generated results rather than just applying the modulus operation.

It is important to realize that these numbers just *look* random, they aren't really random. In particular, we can reproduce the exact same sequence if we give it the exact same seed. This property is important for testing purposes: if we discover a bug during testing with pseudorandom numbers, we can reliably reproduce it.

We can exploit this idea to construct a hash function for strings. We multiply the ASCII value of each character in a string with a pseudorandom number and add up all the results (in modular arithmetic). At the end, we reduce the value to the desired range.

```
#use <string>

int hash_string(string s, int m)
//@requires m > 1;
//@ensures 0 <= \result && \result < m;
{
  int a = 1664525; int b = 1013904223; /* inlined random number generator */
  int r = 0x1337beef;                  /* initial seed */
  int len = string_length(s);
  int h = 0;                    /* empty string maps to 0 */
  for (int i = 0; i < len; i++)
    //@loop_invariant 0 <= i && i <= len;
    {
      h = r*h + char_ord(string_charat(s, i)); /* mod 2^32 */
      r = r*a + b;              /* mod 2^32, linear congruential random no */
    }
  h = h % m;                    /* reduce to range */
  //@assert -m < h && h < m;
  if (h < 0) h += m;            /* make positive, if necessary */
  return h;
}
```

This clearly has the crucial property that on each string the result is unique, so calling it on the same string multiple times will always give the same answer.

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. 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. In one of the homework exercises you will get the opportunity to write a hash table that adjusts its size as necessary, similar to the way unbounded arrays worked.

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

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

struct ht {
  int size;                     /* 0 < size */
  list[] A;                     /* \length(A) == size */
};
```

It will be convenient to have a function to create a new list cell with element e and a given $tail$.

```
list list_new(elem e, list tail) {
  list lnew = alloc(struct list);
  lnew->data = e;
  lnew->next = tail;
  return lnew;
}
```

Next we provide a function to check if a hash table is valid. Besides the invariants noted above we should check that the hash value of the key of every element in the chain stored in $A[i]$ is indeed i.

```
bool is_chain(list l, int i, int m) {
  while (l != NULL) {
    if (l->data == NULL) return false;
    if (hash(elem_key(l->data), m) != i) return false;
    l = l->next;
  }
  return true;
}

bool is_ht(ht H) {
  if (H == NULL) return false;
  if (!(H->size > 0)) return false;
  //@assert H->size == \length(H->A);
  for (int i = 0; i < H->size; i++)
    //@loop_invariant 0 <= i && i <= H->size;
    if (!(is_chain(H->A[i], i, H->size))) return false;
  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 m)
//@requires m > 0;
//@ensures is_ht(\result);
{
  ht H = alloc(struct ht);
  H->size = m;
  H->A = alloc_array(list, m);
  return H;
}
```

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

```
elem ht_search(ht H, key k)
//@requires is_ht(H);
{
  int h = hash(k, H->size);
  list l = H->A[h];
  while (l != NULL)
    //@loop_invariant is_chain(l, h, H->size);
    {
      if (key_equal(elem_key(l->data), k))
        return l->data;
      l = l->next;
    }
 return NULL;
}
```

We can extract the key from the element `l->data` because the data can not be null in a valid hash table. This follows from the loop invariant that l is a valid chain for hash value h and the current size.

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);
//@ensures is_ht(H);
{
  assert(e != NULL);            /* cannot insert NULL element */
  key k = elem_key(e);
  int h = hash(k, H->size);
  list l = H->A[h];
  while (l != NULL)
    //@loop_invariant is_chain(l, h, H->size);
    {
      if (key_equal(elem_key(l->data), k)) {
        l->data = e;            /* modify in place if k already there */
        return;
      }
      l = l->next;
    }
  /* k is not already in the hash table */
  /* insert at the beginning of the chain at A[h] */
  H->A[h] = list_new(e, H->A[h]);
  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 *Extend the hash table interface with new functions* `num_elems` *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 3 *Complete the client-side code to build a hashtable containing word frequencies for the words appearing in Shakespeare's collected works. You should build upon the code in Assignment 2.*

Exercise 4 *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.*