Lecture 13 Notes
Sets

15-122: Principles of Imperative Computation (Summer 1 2015)
Frank Pfenning, Rob Simmons

1 Introduction

In this lecture, we will discuss the data structure of hash tables further and use hash tables to implement a very basic interface of sets. 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 postconditions 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 discussed is that, in order for a 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.

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.

2 Generic Data Structures

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;

/*** 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`, mean that the client doesn't know how the abstract type `queue` is implemented, and that the library is 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 implement, 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.

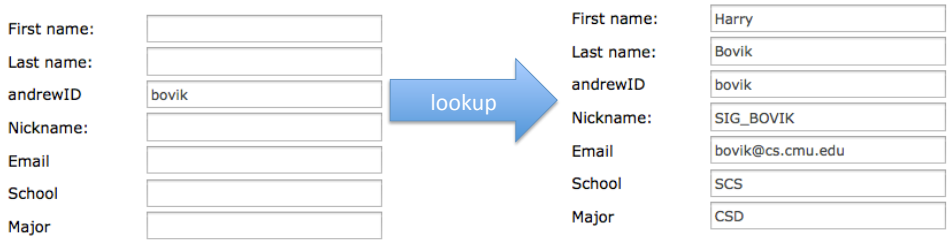
This approach is still not perfect, because any given program only supports a single type of queue element. We'll start working on that problem in the next lecture.

3 Generic Sets and Dictionaries

Hash tables are a way of implementing both *sets* and *dictionaries*. We've seen dictionaries in a couple of settings: dictionaries that map from dictionary words (the keys) and frequency counts (the values), as well as from operator names (the keys) to definitions stored in queues (the values).

In a **dictionary**, we want to have some notion of what a *key* is and some notion of what a *value*. We want to be able to insert key-value pairs into the dictionary, and we want to be able to lookup which value (if any) is associated with a particular key. In a **set**, we don't have a separate notion of keys and values. Instead, we have a single idea of an *element*.

Sometimes the elements of a set are like examples we have already seen: strings, zip codes, and so on. Sometimes, however, we think of the elements of a hash set as *containing* a key, and two elements are *equivalent* if they share the same key part – a notion that will be entirely decided by the client when they decide what an element is. Our interface for sets will then give us a lookup function that allows us to take an element and determine whether an equivalent element already exists in the set. As an example, the elements in a set might be structs with fields for a student's name, student, school, and major. If we decide that the student ID is the key part, then we can look up a student's full record by creating a struct that just has the correct student id, and then finding the equivalent record in the set of student records:



4 Generic Hash Sets

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

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

```
/* library side types */
typedef ______* hset_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 */
hset_t hset_new(int capacity)              /* O(1) */
  /*@requires capacity > 0; @*/
  /*@ensures \result != NULL; @*/ ;

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

void hset_insert(hset_t H, elem x)         /* O(1) avg. */
  /*@requires H != NULL && x != NULL; @*/
  /*@ensures hset_lookup(H, x) == x; @*/ ;
```

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

The function `hset_lookup(ht H, elem x)`, works as described in the previous section: we pass the lookup function an element `x`, and it returns to us an equivalent element, if an equivalent element already exists in the set.

From these decisions we can see that the *client* must provide 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`, so that `elem` must be a pointer.

```
/* client-side types */
typedef ______* elem;
```

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` so that it can hash elements and so that it can determine whether they are equal. Since the library is supposed to be generic, the library implementer cannot write these functions; we require the client to provide them.

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

```
/* client-side functions */
int elem_hash(elem x)
  /*@requires x != NULL; @*/ ;
```

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 element hashes to. 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, the implementation will work correctly (albeit slowly) even if it maps every key to 0.

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

```
/* client-side functions */
bool elem_equiv(elem x, elem y)
  /*@requires x != NULL && y != NULL; @*/ ;
```

This completes the interface which we now summarize.

```
/************************/
/*** Client interface ***/
/************************/
typedef ______* elem;

bool elem_equiv(elem x, elem y)
  /*@requires x != NULL && y != NULL; @*/ ;

int elem_hash(elem x)
  /*@requires x != NULL; @*/ ;

/************************/
```

```
/*** Library interface ***/
/*************************/
typedef ______* hset_t;

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

elem hset_lookup(hset_t H, elem x)
  /*@requires H != NULL && x != NULL; @*/ ;

void hset_insert(hset_t H, elem x)
  /*@requires H != NULL && x != NULL; @*/
  /*@ensures hset_lookup(H, x) == x; @*/ ;
```

5 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;
};

typedef struct wcount* elem;

int elem_hash(elem x)
//@requires x != NULL;
{
  return hash_string(x->word);     /* from hash-string.c0 */
}

bool elem_equal(elem x1, elem x2)
```

```
//@requires x1 != NULL && x2 != NULL;
{
  return string_equal(x1->word, x2->word);
}
```

6 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 `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 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 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 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.

7 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 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 elements, and the hash table type itself.

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

```
struct hset_header {
  int size;                     /* size >= 0 */
  int capacity;                 /* capacity > 0 */
  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 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_hset` function is incomplete.)

```
bool is_hset(hset H) {
  return  H != NULL
    && H->capacity > 0
    && H->size >= 0
    && is_table_expected_length(H->table, H->capacity);
 /* && each element is non-null */
 /* && there aren't equivalent elements */
 /* && the number of elements matches the size */
 /* && every element 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 elements of a hash set hash to the correct index, we need a way of mapping the hash value returned by `elem_hash` to an index of the table. This is a common enough operation that we'll write a helper function:

```
int elemhash(hset H, elem x)
//@requires H != NULL && H->capacity > 0;
//@requires x != NULL;
//@ensures 0 <= \result && \result < H->capacity;
{
  return abs(elem_hash(x) % H->capacity);
}
```

Allocating a hash table is straightforward.

```
hset hset_new(int capacity)
//@requires capacity > 0;
//@ensures is_hset(\result);
{
  hset H = alloc(struct hset_header);
  H->size = 0;
  H->capacity = capacity;
  H->table = alloc_array(chain*, capacity);
  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 hset_lookup(hset H, elem x)
//@requires is_hset(H);
//@requires x != NULL;
{
  int i = elemhash(H, x);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    //@assert p->data != NULL;
    if (elem_equal(p->data, x)) return p->data;
  }
  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 hset_insert(hset H, elem x)
//@requires is_hset(H);
//@requires x != NULL;
//@ensures is_hset(H);
//@ensures x == hset_lookup(H, x);
{
  int i = elemhash(H, x);
  for (chain* p = H->table[i]; p != NULL; p = p->next)
  // loop_invariant: p points to a chain (no NULL data)
  {
    //@assert p->data != NULL;
    if (elem_equal(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)++;
}
```

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* `elem_hash(key k, int m)` *that returns a result between 0 (inclusive) and* m *(exclusive).*

Exercise 3 *Extend the hash table interface with new function* `ht_tabulate` *that returns an array with the elements in the hash table, in some arbitrary order.*

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.*