15-122: Principles of Imperative Computation

Recitation 14 — Josh Zimmerman, Arjun Hans

Hash tables

There are a few ideas that are key to hash tables:

Key-value mapping A hash table maps *keys* to *values*.

Hash function A hash table maps keys to values by applying a *hash function* to the key. The hash table implementation uses the hash function to index into the hash table backbone.

More precisely, we say that if the hash value of a key k is $hash(k)$, then the index that we use is $hash(k)\%m$, where m is the length of the hash table backbone.

Load factor The *load factor* of a hash table is $\frac{n}{m}$ where n is the number of items we're storing in the hash table and m is the length of the array we're using for the hash table. If the load factor is too high, we'll have lots of collisions and should consider resizing the hash table to improve speed.

Collisions If there are more than m elements that we want to store in our hash table, we'll try to put two values at the same index, since there aren't enough places in the array to store values.

This creates a collision, so we need some kind of *collision resolution policy* to handle this issue.

Mathematically, we say that two elements collide if their keys k_1 and k_2 are such that $k_1 \neq k_2$ and $hash(k_1)\%m = hash(k_2)\%m$.

Clearly, we need a way to handle collisions. Some of the common policies that we use include:

(1) Separate Chaining: We store elements that hash to the same value modulo the backbone length in a chain, which can be readily implemented using a linked list.

(2) Linear Probing: We first access index i obtained by hashing the key modulo the backbone length. Future elements hashing to the same index i are stored at positions $i + k$, where k is the attempt counter (so we'll first access i, then $i + 1$, then $i + 2$, and so on).s

(3) Quadratic Probing: We first access the index i obtained by the hashing the key modulo the backbone length. Future elements bashing to the same index i are stored at positions $i + k^2$, where k is the attempt counter (so we'll first access i, then $i + 1$, then $i + 4$, $i + 9$, and so on).

We'll mainly consider separate chaining in our hash table implementations, although you will answer some theory questions regarding linear probing and quadratic probing too.

What goes in the library? The client?

In lecture, we talked about some code that the client needs to implement and some code that the library needs to implement. This is the notion of client vs library, which is one of the most important ideas in computer science. We've seen this before when working with stacks and queues, but hash tables add an extra layer of complexity, especially because the client needs to provide functions

(1) The client knows the types of the elements that are to be used with the data structure. It only knows about the signature of the data structure it wishes to use (the interface functions, the arguments

required, the runtime complexity of the methods, etc). It does not know how the data structure is implemented.

(2) The library, on the other hand, knows precisely how the data structure is implemented. However, it should not make any assumptions about the types that the client wants to use. It will use the element types/ methods provided by the client to complete its method implementations.

Failing to follow this distinction can lead to really bad things!

In the context of our hash table code, we observe the following distinctions between the client and the library.

(1) Client: The client provides the following types/methods:

  (i) The type of the key and the data. In our hash table implementation, `elem` must be a pointer.
  (ii) A method to obtain a key from a data item (`key elem_key(elem e)`)
  (iii) A method to compare keys for equality (`bool key_equal(key k1, key k2)`)
  (iv) A method to hash a key to an integer value (`int hash(key k)`)

(2) Library: The library provides the following types/methods:

  (i) The type of a hash table. Our hash table has fields storing the number of elements in the hash table, the length of the backbone and a backbone of pointers to linked lists.
  (ii) A method to create a new hash table with a desired backbone length (`ht ht_new(int capacity)`)
  (iii) A method to lookup an element from the hash table using a key (`elem ht_lookup(ht H, key k)`)
  (iv) A method to insert an element into a hash table (`void ht_insert(ht H, elem e)`)
  (v) A method to obtain the number of elements in the hash table (`int ht_size(ht H)`)

Observe how the client only deals with the key and data types, while the library only deals with the hash table data structures. This means that, in theory, the same hash table implementation can be used with numerous clients (unfortunately, in c0, we cannot use multiple clients with a single hash table implementation simultaneously, as types are not polymorphic). However, the idea is still valid; when we begin our discussion of C, we'll see how to bring this concept into practice.

Of course, we can have more methods in the library implementation. As a fun exercise, think of some additional methods that the hash table library can provide.

Hash Table Code

Let's now take a look at some library code from lecture. It's more complicated than anything we've worked with up to now, so we'll also do some practice exercises at the end:

`ht_new`

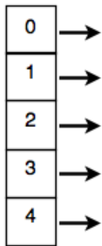
```
ht ht_new(int capacity)
//@requires capacity > 0;
//@ensures is_ht(\result);
```

```
{
  ht H = alloc(struct ht_header);
  H->size = 0;
  H->capacity = capacity;
  H->table = alloc_array(chain*, capacity);
  /* initialized to NULL */
  return H;
}
```

(a) Note the distinction between the size and the capacity; the *size* is the number of elements in the hash table, while the *capacity* is the length of the hash table backbone.

(b) We create a backbone of `capacity` `chain*`'s. These chains are initially empty, so each `chain*` points to `NULL`.

So, hash table initially looks like this (if we used `capacity = 5`).



ht_insert

```
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];

  // Either replace the element if it's there
  while (p != NULL)
    // loop_invariant is_good_chain (no NULL elems)
    {
      if (key_equal(k, elem_key(p->data))) {
        p->data = e;
        return;
      }
      p = p->next;
    }
```

```
  // Or add it to the chain
  chain* newc = alloc(chain);
  newc->data = e;
  newc->next = H->table[i];
  H->table[i] = newc;
  (H->size)++;
}
```

(a) First, we need to extract the key from the element to be inserted. We do this using the `elem_key` function provided by the client.

(b) We then hash the key and index into the appropriate chain modulo the length of the hash table backbone. This gives us the correct chain to insert the element at.

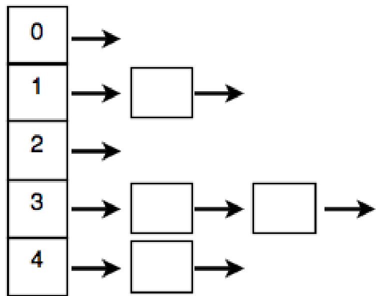
(c) We then traverse the appropriate chain; for each node in the chain, we check if the key has already been inserted. If we find a matching element, we just overwrite the old element with the new element.

(d) If we reach the end of the chain without finding a matching key, we then create a new node. We initialize the corresponding fields and place the node at the front of the chain.

Let's now work through some practice problems.

Pointers and Array Access in Hash Tables

Suppose that our hash table looks like this. Write expressions to perform the following operations, given the hash table code provided in lecture:



(a) Access the 0th chain in the hash table.

(b) Access the 1st node in the 1st chain of the hash table.

(c) Access the data of the 2nd node in the 3rd chain of the hash table.

Using `ht_insert`

Suppose that our hash function is $h(k) = 2k + 1$ and our hash table capacity is 5. Draw the hash table after each of the following key-value pair insertions: $(2, a), (4, b), (5, c), (9, d), (4, e), (7, f)$.

Hash Table Invariants

As with any data structure, we should develop invariants for our hash table implementation using separate chaining. In lecture, we started sketching up a very incomplete (`is_ht`) contract:

```
bool is_ht(ht H) {
  if (H == NULL) return false;
  if (!(H->size >= 0)) return false;
  if (!(H->capacity > 0)) return false;
  /* Finish me */
  return true;
}
```

Obviously, this is not sufficient to ensure that a hash table instance `H` is valid; it just checks that it is not NULL and both the capacity and size are positive. As an exercise, come up with some other invariants (in prose) that our `is_ht` method should include. Here are a few hints to get you started:

(a) What sort of data can be inserted into the hash table?

(b) Where should the data elements be located in the hash table?

(c) What should hold true about the hash table chains?

Hashtable lookup

Next, work together to write hashtable lookup code, using the following interface and struct definition of hashtables. It should return the element if it is in the hash table or NULL if it is not in the hash table.

```
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 */
};
typedef struct ht_header* ht

elem ht_lookup(ht H, key k)
//@requires is_ht(H);
```