15-122: Principles of Imperative Computation

Recitation 14 — Josh Zimmerman, Matt Dee

Before we get started - Your TAs would appreciate feedback! Please visit https://www.ugrad.cs.cmu.edu/ta/S14/feedback/ to fill out the short form so that we can make 15-122 better for you!

Hash Tables Interface - Library vs. Client

In lecture we looked at how we can go about implementing a hash table with separate chaining. One observation is that there is some code that the library writes and some code that the client needs to implement.

Note that, in stacks and queues, the client had only specified datatypes (with `typedef ___ elem`). Now, however, the client interface includes functions.

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

- Client: 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 functions, etc). It does not know how the data structure is implemented. The client provides the following types/functions for the hash table to use:
  - The type of the key and the data. This is done with `typedef ___ elem` and `typedef ___ key`. Note that for our implementation, `elem` must be a pointer, and any elements that we put in the array must not be NULL. This is so that we can return NULL if the element is not found.
  - A function to obtain a key from a data item (`key elem_key(elem e)`)
  - A function to compare keys for equality (`bool key_equal(key k1, key k2)`)
  - A function to hash a key to an integer value (`int hash(key k)`)
- Library: 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/ functions provided by the client to complete its function implementations. The library provides the following types/functions to the client:
  - 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.
  - A function to create a new hash table with a desired backbone length (`ht ht_new(int capacity)`)
  - A function to lookup an element from the hash table using a key (`elem ht_lookup(ht H, key k)`)
  - A function to insert an element into a hash table (`void ht_insert(ht H, elem e)`)
  - A function 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, if we wanted to use the hash table with a different type, we would only need to change the client functions! Similarly, if we wanted to change the implementation of the hash table (eg., change from separate chaining to linear probing) we would only need to change the library code.

Excercise: Client Interface

The NSA has contracted you to write a hash table which allows you to lookup any person's name by their social security number! We define the following struct:

```
struct person_info {
  int ssn;
  string name;
};
typedef struct person_info person;
```

Now, implement the client interface to store citizens by their ssns.

```
typedef int key;
typedef person* elem;

key elem_key(elem e) {
  return e->ssn;
}
bool key_equal(key k1, key k2) {
  return k1 == k2;
}
// Could also be ‘’return k+ 7*(k%2) + 13*(k%5);‘, or
// anything that has an even-ish distribution.
int hash(key k) {
  return k;
}
```

Why do we need a `key_equal` function? Why couldn't we simply compare their hashes, or compare them directly with ==?

In many cases, there may be two different keys which hash to the same value. And == would only check pointer equality, which is not always desired.

Hash Table Code

Let's now take a look at some library code from lecture, and the do some exercises.

As a reminder, we define the following datatypes for hashtables:

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

```
struct ht_header {
  int size;         /* size >= 0 */
  int capacity;     /* capacity > 0 */
  chain*[] table;   /* \length(table) == capacity */
};
typedef struct ht_header* ht;
```

Note that size can actually be greater than capacity! What does it mean if this is the case? It means that the load factor is greater than 1.

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);
  /* Each element initialized to NULL */
  return H;
}
```

Note that the hash table is initially an array of NULL pointers, which indicate empty lists.

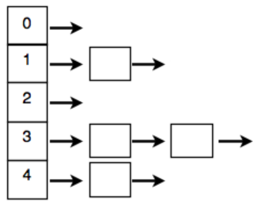
ht_insert

```
void ht_insert(ht H, elem e)
//@requires is_ht(H) && e != NULL;
//@ensures is_ht(H);
//@ensures ht_lookup(H, elem_key(e)) == e;
{
  key k = elem_key(e);
  // Get the index of the table that we insert to
  // Take the absolute value because result of modulus can be negative
  int i = abs(hash(k) % H->capacity);
  chain* p = H->table[i];

  // Replace the element if it's there...
  while (p != NULL)
    {
      if (key_equal(k, elem_key(p->data))) {
        // Overwrite the element
        p->data = e;
        // Do not increase H->size here!
        return;
      }
      p = p->next;
    }

  // If it is not already there, add it to the front of the chain
  chain* newc = alloc(chain);
  newc->data = e;
  newc->next = H->table[i];
  H->table[i] = newc;
  (H->size)++;
}
```

Here, we first find the chain corresponding to the hash value, and then search for our key. If we find it, we overwrite the old element, otherwise, we add this element to the hash table.

Excercise: Pointers and Array Access in Hash Tables

Suppose that we have a hash table H that looks like this. Write expressions to perform the following operations, given the has table representation above.



(a) Access the 1st chain in the hash table.

`H->table[0]`

(b) Access the data of the 1st node in the 2nd chain of the hash table.

`H->table[1]->data`

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

`H->table[2]->next->data`

Excersise: Hash Table Invariants

As with any data structure, we should develop invariants for our hash table implementation using separate chaining. Below is 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` function should include. Here are a few hints to get you started:

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

We need to check that each of the data fields are not NULL.

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

We need to make sure that they are placed in the right bucket, by hashing again and comparing the indices.

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

They need to be acyclic.

Excersise: Hashtable lookup

Next, 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.

```
elem ht_lookup(ht H, key k)
//@requires is_ht(H);
{
  int i = abs(hash(k) %H->capacity);
  chain* p = H->table[i];
  while (p != NULL) {
    if (key_equal(k, elem_key(p->data))) {
      return p->data;
    }
  }
  return NULL;
}
```