15-122: Principles of Imperative Computation

Recitation 14

Josh Zimmerman

Hash tables

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

Key-value mapping A hash table maps *keys* to *values*. The keys and values can be of any type. This idea is useful in a wide variety of applications including dictionaries (keys are words and values are their definitions) and databases (keys might be user IDs and values might be the user's data).

Hash function A hash table maps keys to values by applying a *hash function* to the key. This function transforms the key into a non-negative integer that can be used to index into an array. However, it's impractical to just use any non-negative integer for an index (we can't have an array that has space for 2,147,483,647 items), so we use the mod operator (%) to restrict the value to be within the size of the array we're using. More concretely, if our array has size m, we use index $h(k)\%m$, where h is our hash function and k is our key.

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.

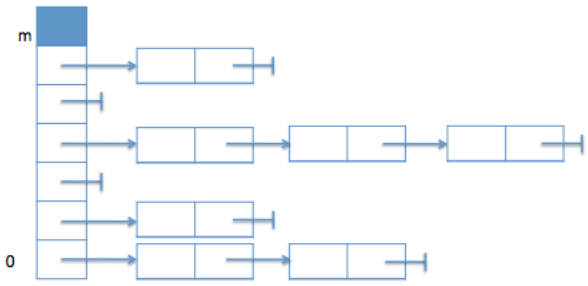
Collisions If there are more than m elements that we want to store in our hash table, we'll try to put two values at the same index, since there aren't enough places in the array to store values.

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

Even if we had a table that *was* large enough, we'd still need to handle collisions. There's a well known problem called the *birthday problem* which states that if people's birthdays are uniformly distributed (not counting leap days) and we have a group of 23 people, there is a 50% chance that two of them will have the same birthday.

I won't delve into detail about the math behind it here, but the problem is very relevant to hashing: if 2,500 keys are hashed with a uniform hash function (one which uses every index in the array equally) into a hash table with capacity 1,000,000, there is roughly a 95% chance that at least two keys will be put into the same place in the array.

One collision resolution policy we'll talk about a lot in this class is called *separate chaining*, where for each entry in the hash table we store a linked list of things that hashed to that index in the array. To be a bit more clear, here's a sketch of a hash table using separate chaining to resolve conflicts, courtesy of Professor Pfenning.



Note that we have to keep the key along with the value for each entry in the hash table so we can check if we have come to the correct entry in the chain.

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. How can we tell the difference?

A good rule of thumb is that we want the library to be as general as possible, but still do everything necessary for a hash table to work. For instance, a library should not implement a specific hash function since we want to be able to put data of any type into our hash function, and a user of our hash table library may define some struct that requires a different hash function from anything we've seen before.

However, the `ht_insert` and `ht_lookup` functions are functions that are the same regardless of what type of data we're storing in the hash table, so the hash table library should implement them.

This distinction between interface and implementation, and library and client, is *incredibly* important, so *please* ask questions if you have any.

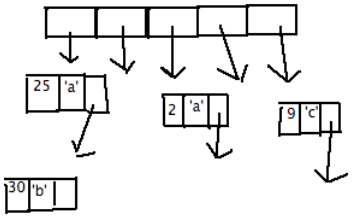
`ht_insert`

Let's go over the `ht_insert` function we wrote in lecture.

```
1  void ht_insert(ht H, elem e)
2  //@requires is_ht(H);
3  //@requires e != NULL; // We require that elem is a pointer type
4  //@ensures is_ht(H);
5  //@ensures ht_lookup(H, elem_key(e)) != NULL;
6  {
7     key k = elem_key(e);
8     int i = hash(k, H->capacity);
9
10    chain* p = H->table[i];
11    while (p != NULL)
12    //@loop_invariant is_chain(p, i, H->capacity);
13    {
14       //@assert p->data != NULL;
15       if (key_equal(elem_key(p->data), k)) {
16          /* overwrite existing element */
17          p->data = e;
18          return;
19       } else {
20          p = p->next;
21       }
22    }
23    //@assert p == NULL;
24    /* prepend new element */
25    chain* q = alloc(struct chain_node);
26    q->data = e;
27    q->next = H->table[i];
28    H->table[i] = q;
29    (H->size)++;
30    return;
31 }
```

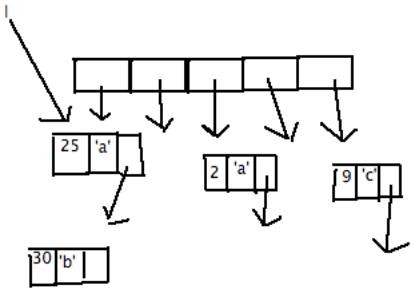
Let's walk through two insertions: one where we update an existing key and one where we add a new key.

Note that the hash function here is $h(x) = |x|\%5$ (we're working with a size-5 array).

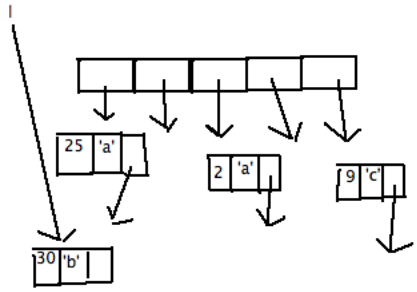


We'll step through updating the key 30 to have the value 'z'.

I'll start diagrams on line 8, after we have extracted the key and the hash.



The key 25 is not equal to 30, so we keep going:



Now, the key 30 is equal to 30, so we update the value to 'z' and return.

Now, we'll work through a problem: draw diagrams for insertion of a new element into the above hash table. We'll insert the key 14 with the value 'j' into the table. As usual, this is at the end of the recitation handout.

Alternate collision resolution policies

Separate chaining is just one possible collision resolution policy. There are several other possibilities.

One is *linear probing*, in which we just iterate over the indices in the hash table if we have a collision (if we first try index x, we'd next try index $x + 1$, then $x + 2$, etc). This has the disadvantage that if there are a lot of collisions, lookup becomes very slow because of the large block of elements right next to each other.

Quadratic probing attempts to resolve the problem of elements being grouped into large blocks by using index x, then index $x+1$, then $x+4$, then $x+9$, etc. This has a very large problem: there's no guarantee

of finding a place to insert something if the array is sufficiently full.

Separate chaining, then, is a solution that starts to look pretty decent since we can fit as many elements as we want into the hash table. However, in the worst case it can perform poorly: If someone knows our hash function and table size they could give input to our hash table such that we'd try to put it all in the same slot of the hash table.

We can potentially improve this if we require that clients give us a comparison function and then internally use a binary search tree instead of a linked list in each spot of the array. (It's possible that we could get a worst case here and that it wouldn't be any better than a linked list, so for an improvement in all cases, we'd need a balanced binary tree. We'll talk about how to implement those later in the semester.)

Insertion into a hash table

Insert the key 14 with the value 'j' into the hash table pictured earlier in the handout, after updating the key '30' to 'z'. Start your diagrams after line 8 of the insert code.

Hashtable lookup

Next, work together to write hashtable lookup code, using the following interface and struct definition of hashtables:

```
1  struct chain_node {
2    elem data;       /* data != NULL */
3    struct chain_node* next;
4  };
5  typedef struct chain_node chain;
6
7  struct ht_header {
8    int size;     /* size >= 0 */
9    int capacity;    /* capacity > 0 */
10   chain*[] table;  /* \length(table) == capacity */
11 };
12 typedef struct ht_header* ht;
13
14 /* ht_lookup(H, k) returns NULL if key k not present in H */
15 elem ht_lookup(ht H, key k)
16 //@requires is_ht(H);
17 ;
```