15-122: Principles of Imperative Computation

Recitation 9

Nivedita Chopra, 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*.

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.

It's integral that a hash function be *deterministic* and that the values be *distributed uniformly*

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.

Checkpoint 0

Determine whether the following functions are good hash functions

(a)

```
1 int hash(int x){
2    return 122;
3 }
```

(b)

```
1 int hash(int x){
2    return x;
3 }
```

(c)

```
1 int hash(int x){
2    return x + 122;
3 }
```

(d)

```
1 int hash(int x){
2    return x/122;
3 }
```

(e)

```
1 int hash(int x){
2 rand_t gen = init_rand(122);
3    return rand(gen);
4 }
```

(f)

```
1 int hash(int x){
2 return x + (x % 5) * 122 + (x % 2) * 15122;
3 }
```

Checkpoint 1

Insert the following elements into a hash table of size 5, in order, using the identity hash function (as in part (b) of checkpoint 0) and *separate chaining* to resolve collisions :

$$110, 112, 122, 150, 251, 210, 213, 451$$

Checkpoint 2

Insert the following elements into a hash table of size 7, in order, using the identity hash function (as in part (b) of checkpoint 0) and *linear probing* to resolve collisions:

$$112, 122, 150, 251, 210, 213$$

Checkpoint 3

Insert the following elements into a hash table of size 7, in order, using the identity hash function (as in part (b) of checkpoint 0) and *quadratic probing* to resolve collisions:

$$112, 122, 150, 251, 210, 213$$