# Lecture 25 — Hashing (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

*Lectured by Guy Blelloch — Nov 22, 2011*

**Today:**
- Hashing

## 1 Hashing

**hash:** transitive verb[1]

1. (a) to chop (as meat and potatoes) into small pieces
   (b) confuse, muddle
2. ...

This is the definition of hash from which the computer term was derived. The idea of hashing as originally conceived was to take values and to chop and mix them to the point that the original values are muddled. The term as used in computer science dates back to the early 1950s.

More formally the idea of hashing is to approximate a random function $h : \alpha \to \beta$ from a source set $\alpha$ to a destination set $\beta$. Most often the source set is significantly larger than the destination set, so the function not only chops and mixes but also reduces. In fact the source set might have infinite size, such as all character strings, while the destination set always has finite size. Also the source set might consist of complicated elements, such as the set of all directed graphs, while the destination are typically the integers in some fixed range. Hash functions are therefore many to one functions.

Using an actual randomly selected function from the set of all functions from $\alpha$ to $\beta$ is typically not practical due to the number of such functions and hence the size (number of bits) needed to represent such a function. Therefore in practice one uses some pseudorandom function.

**Exercise 1.** *How many hash functions are there that map from a source set of size $n$ to the integers from 1 to $m$? How many bits does it take to represent them? What if the source set consists of character strings up length up to 20. Assume there are 100 possible characters.*

Why is it useful to have random or pseudo random functions that map from some large set to a smaller set. Generally such functions might be used to hide data, to reorder data in a random order, or to spread data smoothly over a range. Here we consider some applications of each.

1. We saw how hashing can be used in treaps. In particular we suggested using a hash function to hash the keys to generate the "random" priorities. Here what was important is that the ordering of the priorities is somehow random with respect to the keys. Our analysis assumed the priorities were truly random, but it can be shown that a limited form of randomness that arise out of relatively simple hash functions is sufficient.

[1] Merriam Websters

2. In cryptography hash functions can be used to hide information. One such applications is in digital signatures where a so-called secure hash is used to describe a large document with a small number of bits.

3. Hashing can be used to approximately match documents, or even parts of documents.

4. Hashing is used to implement hash tables. In hash tables one is given a set of keys $K \subset \alpha$ and needs to map them to a range of integers so they can stored at those locations in an array. The goal is to spread the keys out across the integers as well as possible to minimize the number of keys that collide in the array. You should not confuse the terms hash function and hash table. They are two separate ideas, the later that uses the prior.

There is a deep and interesting theory of hash functions. Depending on the application, the needs of the hash function are very different. We will not cover the theory here but you will likely see it in more advanced algorithms classes.

So what are some reasonable hash functions. Here we consider some simple ones. For hashing integers we can use

$$h(x) = (ax + b) \bmod p$$

where $a \in [1, \ldots, p-1]$, $b \in [0, \ldots, p-1]$, and $p$ is a prime. This is called a linear congruential hash function has some nice properties that you will likely learn about in 15-451.

For strings we can simply use a polynomial

$$h(S) = \left( \sum_{i=1}^{|S|} = s_i a^i \right) \bmod p$$

## 1.1 Open Address Hash Tables

In 15-122 you covered hash tables using separate chaining.

We are going to cover a technique that does not need any linked lists but instead stores every key directly in array. Open address hashing using so called linear-probing has an important practical advantage over separate chaining: it causes fewer cache misses since typically all locations that are checked are on the same cache line.

In our discussion I will assume we have a set of keys we want to store and a hash function $h : \texttt{key} \rightarrow [1, \ldots, n]$ for some $n$.

Basic idea of open address is to maintain an array that is some constant factor larger than the number of keys and store all keys directly in this array. Every cell in the array is either empty or contains a key.

To decide what keys go in what cells we assign every key an ordered sequence of locations in which the key can be stored. In particular lets assume we have a function $h(k, i)$ that returns the $i^{th}$ location for key $k$. We refer to the sequence $\langle\, h(k,1), h(k,2), h(k,3), \ldots \,\rangle$ is as the *probe* sequence. We will get back

to how the probe sequence might be assigned, but lets first go through how these sequences are used. The basic idea is when inserting for a key to try each of the locations in order until it finds a slot that is empty, and then insert the key at that location. Sequentially this would look like

```
1  fun insert(T, k) =
2  let
3    fun insert'(T, k, i) =
4      case T[h(k, i)] of
5        NONE ⇒ update(i, k, T)
6      | _ ⇒ insert'(T, k, i + 1)
7  in
8    insert'(T, k, 1)
9  end
```

For example:

```
T = [ _  B  _   _   E  A _ F ]
```

Now if for a key $D$ we had the probe sequence $\langle 1, 5, 3, \cdots \rangle$ (0 based) then we would find location 1 and 5 full (with $B$ and $E$) and place $D$ in location 3 giving:

```
T = [ _  B  _   D  E  A _ F ]
```

Note that the code loop forever if all locations are full. Such an infinite loop can be prevented by ensuring that $h(k, i)$ tries every location as $i$ is incremented, and also checking when the table is full. Also note that the code given will insert the same key multiple times over.

To search we then have the following code:

```
1  fun find(T, k) =
2  let
3    fun find'(T, k, i) =
4      case T[h(k, i)] of
5        NONE ⇒ false
6      | SOME(k') ⇒ if (eq(k, k')) then true
7                   else find'(T, k, i + 1)
8  in
9    find'(T, k, 1)
10 end
```

Now if for a key $E$ we have the probe sequence $\langle 7, 4, 2, \cdots \rangle$ we would first search location 7 and, which is full, then location 4 and find $E$.

```
Do example of find
```

```
What about delete
  -- ask if simple delete works
  -- lazy delete -- replace with a special HOLD VALUE
  -- datatype 'a entry = EMPTY | HOLD | FULL of 'a
  -- find will skip over a HOLD and move to next probe
  -- insert v can replace HOLD in probe sequence with FULL(v)
  -- if insert needs to overwrite old value, then must first search to end

What probe sequence can we use.
 - linear probing
 - quadratic probing
 - multiple hash functions
```

## 1.2 Parallel Hashing

We assume a function $\texttt{injectCond}(IV, S) : (int \times \alpha)seq \times (\alpha\texttt{option})seq \to (\alpha\texttt{option})seq$. It takes a sequence of index-value pairs $\langle (i_1, v_1), \ldots, (i_n, v_n) \rangle$ and a target sequence $S$ and conditionally writes each value $v_j$ into location $i_j$ of $S$. In particular it only writes the value if the location is set to NONE and there is no previous equal index in $IV$.

Using this we have:

```
1  fun insert(T, K) =
2  let
3    fun insert'(T, K, i) =
4      if |K| = 0 then T
5      else let
6        val T' = injectCond({(h(k,i), k) : k ∈ K}, T)
7        val K' = {k ∈ K | T[h(k,i)] ≠ k}
8      in
9        insert'(T', K', i + 1)      end
10 in
11   insert'(T, k, 1)
12 end
```

Note that if $T$ is implemented using single threaded arrays, then this basically does the same work as the sequential version adding the elements one by one.

```
What does it mean for set and table implementations
  - search faster (in expectation)
  - insert faster (in expectation)
  - map, reduce, remain linear
  - union/merge slower or faster depending on particular settings
```