Dictionaries, Hashing, and Hash Tables

1 Dictionaries and Sets

1.1 The Dictionary Interface

One of the main motivations behind the study of hashing and hash functions is the **dictionary** abstract data type. A dictionary D stores a set of *items*, which we will assume are **key-value pairs**. The operations we want to support with a dictionary are:

Interface: Dictionary ADT

A `dictionary<K,V>`, parameterized on the type of its keys `K` and values `V` supports:

- `insert(k,v)`: add the given key, value pair `k:v`
- `find(k)`: return the value for the given key k (or `NONE` if it doesn't exist)
- `delete(k)`: delete the item with the given key k

Dictionaries go by many other names in different programming languages or textbooks. They are often called *maps* (e.g., in Java, the `map` interface which is implemented by `TreeMap` and `HashMap`; in C++, `map` and `unordered_map`; and in OCaml, the `Map` module), but this name conflicts with the higher-order function `map` on sequences. We will use the term dictionary, which is probably the most common name used in theoretical algorithms books, and is the name used in Python. They are also sometimes called *associative arrays* or *tables*.

We will use Python-like pseudocode for dictionaries, i.e., curly brackets with `{key:value}` pairs. Just like other data structures with update operations, we can consider either *mutable/imperative* data structures or *persistent/functional* data structures.

Remark: Persistence vs. Mutation

Throughout this course, most abstract data types we study are treated as *persistent*: operations conceptually return a new data structure, leaving the original unchanged. This makes reasoning about correctness and parallelism significantly easier.

However, some implementations, most notably hash tables, are most naturally implemented as *mutable* data structures, where operations modify the dictionary in place. When this distinction matters, we will state it explicitly. For now, you should think of the dictionary interface as describing *behavior*, not whether updates are persistent or mutable. This behavior would be encoded explicitly if we were to type the update operations: In a persistent dictionary, `insert` would be typed as:

```
insert(dictionary<K,V>, K, V) -> dictionary<K,V>
```

i.e., it would explicitly return a *new* dictionary, leaving the old one unmodified. On the other hand, a mutable dictionary would have `insert` typed as:

```
insert(dictionary<K,V>, K, V) -> void
```

i.e., with no return value (this is denoted as `unit` in functional languages).

1.2 Variants of the Dictionary Interface

The basic dictionary ADT supports only three simple operations: `insert`, `find`, and `delete`. There are variants of dictionaries that support more operations, at the expense of being harder to implement or more limited on the kind of data structures that can be used to support them.

1.2.1 Sorted Dictionaries

A common variant involves assuming that the keys are from a totally ordered set and can therefore be sorted. This allows us to define a **sorted dictionary**. Sorted dictionaries support order-based operations, which are essential in applications such as range queries, predecessor/successor search, and ordered iteration.

Interface: Sorted Dictionary ADT

A sorted dictionary, `SortedDict<K,V>`, parameterized on the type of its keys `K` (which must be totally orderable) and values `V` supports:

- `insert(k,v)`: add the given key, value pair `k:v`
- `find(k)`: return the item with the given key k (or `NONE` if it doesn't exist)
- `delete(k)`: delete the item with the given key k
- `first()`: return the item with the least key (or `NONE` if empty)
- `last()`: return the item with the greatest key (or `NONE` if empty)
- `prev(k)`: returns the item with the greatest key less than k (`NONE` if k is the least)
- `next(k)`: returns the item with the least key greater than k (`NONE` if k is the greatest)

Since they support more operations, operations on sorted dictionaries typically require a higher cost than on ordinary (unsorted) dictionaries. We will see in a few lectures from now how to implement sorted dictionaries using balanced binary search trees, where operations cost $O(\log n)$.

1.2.2 Sets

Many programming languages also support the notion of a **set** data type (and similarly, a sorted set). A set represents a collection of elements that can be efficiently tested for membership (i.e., one can efficiently answer question: is item x present in the set or not).

The set abstraction is useful for representing the mathematical notion of a set. As such, it is common for programming languages to also support set mathematical operations on sets such as intersection, union, and difference. This leads to the following common API for a set.

Interface: Set ADT

A `set<K>` parameterised on the type of its keys `K`, supports:

- `insert(k)`: add the given key k
- `contains(k)`: return True if the set contains k, false otherwise
- `delete(k)`: delete k from the set
- `union(S)`: return a set containing keys that are either in this set or in S
- `intersection(S)`: return a set containing the keys that are in both this set and S
- `difference(S)`: return a set containing the keys in this set but not in S

Similarly, one can define a **sorted set**:

Interface: Sorted Set ADT

A `SortedSet<K>` parameterised on the type of its keys `K` (which must be totally orderable), supports:

- `insert(k)`: add the given key k
- `contains(k)`: return True if the set contains k, false otherwise
- `delete(k)`: delete k from the set
- `union(S)`: return a set containing keys that are either in this set or in S
- `intersection(S)`: return a set containing the keys that are in both this set and S
- `difference(S)`: return a set containing the keys in this set but not in S
- `first()`: return the least key in the set (or `NONE` if empty)
- `last()`: return the the greatest key in the set (or `NONE` if empty)
- `prev(k)`: returns the greatest key in the set less than k (`NONE` if k is the least)
- `next(k)`: returns the least key in the set greater than k (`NONE` if k is the greatest)

The APIs for sets and dictionaries have many similarities, and indeed, implementation-wise, sets are essentially just the same thing as a dictionary but without the notion of values. One way to think of a set is therefore as a dictionary where the value type is empty/nothing. Alternatively, if you prefer the other way around, one can think of a dictionary as a set where the items are key-value pairs. This means that the same data structures are typically viable for both.

Remark: Set and Dictionary Implementations in the Wild

Indeed, in most programming language standard library implementations, sets and dictionaries are usually implemented as the same data structure under the hood, and simply present a different API for the user to interact with them.

For example, in libstdc++ (the GNU/GCC implementation of the C++ standard library), they use the same Red-Black tree code as the underlying data structure for both `std::set` and `std::map`, just exposing a different interface to the user. Similarly, in CPython, `set` and `dict` were originally based on the same hash table implementation.

2 Hash Tables

Hash tables are an approach for implementing (non-sorted) dictionaries and sets that is often the fastest in both theory and practice. The goal of hash tables is to implement the dictionary interface with *constant-time*, i.e., $O(1)$ time operations for *any set of keys*, i.e., in the worst case! That is, we do not want to make assumptions about the particular keys being stored that are beneficial to us. Our algorithm should work for *any* set of keys given as input.

The secret ingredient to achieving this level of efficiency is *randomization*; efficient hash tables are *randomized data structures* that achieve $O(1)$ cost in expectation or $O(1)$ with high probability for operations with any set of keys.

Hash tables are inherently an imperative/mutable data structure which do not lend themselves well to persistence. The tradeoff is that they are essentially *as fast as possible*; no data structure can be faster than $O(1)$ per operation. We will see later how to implemented persistent dictionaries (specifically, persistent sorted dictionaries) using balanced binary search trees which achieve $O(\log n)$ cost for most operations.

You should already be familiar with the main ideas of hash tables from your previous studies. The purpose of this lecture is to delve into the theory of why they work so well, and the theoretical justification behind when we can and cannot truly say they are constant time.

2.1 Setup and terminology

To design and analyze hashing and hashing-based algorithms, we need to first establish the notion and terminology we will work with.

The key space (the universe): The *keys* are assumed to come from some large *universe* U. When analyzed on the word RAM model (or the fork-join RAM), we will assume that the universe is $U = \{0, \ldots, u-1\}$, where $u = 2^w$ is the universe size, i.e., the keys are word-sized integers.

Hash functions: Hash functions are functions defined on the set of keys (the universe U):

Definition: Hash Function

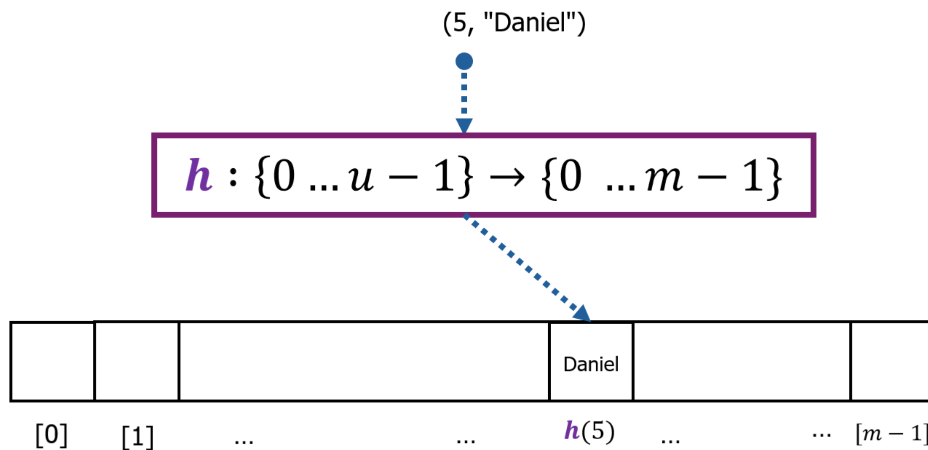
Given a universe of keys U and an integer m (where typically it is assumed that $m \ll |U|$), a hash function is a mapping from U to the integers $\{0, \ldots, m-1\}$, i.e.,

$$h : U \to \{0, \ldots, m-1\}.$$

Intuitively, the purpose of a hash function is to map integers from a very large set into a much smaller set. The output of a hash function is called the *hash value* or the *hash code* of the key.

A hash table: A hash table stores some set of items with keys $S \subseteq U$ (which may change over time with `insert` and `delete` operations). Let $n = |S|$ be the *size* of the hash table. We assume that n is much smaller than u. It stores the items in an array A of some size m (called the *capacity*). The ratio $\alpha = n/m$ is called the **load factor** and represents how full the hash table is.

Hash tables perform inserts, finds, and deletes by using the hash function to determine the position that they key should be stored in the array. That is, given an item with key x, the idea of a hash table is that we want to store it in $A[h(x)]$.



Note that if U was small then you could just store the item in $A[x]$ directly, no need for hashing! Such a data structure is often called a direct-access array or direct-address table. The problem is that U is assumed to be very very big, so this would use an impossible amount of memory[1]. That is why we employ hashing.

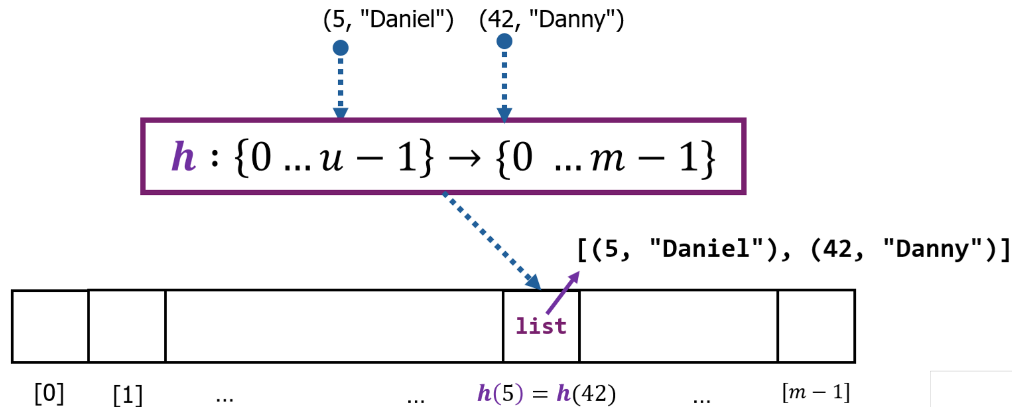
Collisions: The strategy of simply storing an item with key x in $A[h(x)]$ falls apart when we realize that there could be two distinct keys x and y such that $h(x) = h(y)$. This is called a **collision**. There are many different methods for resolving collisions. Most of them roughly fall into one of two categories: open addressing and closed addressing.

- An **open addressing** hash table resolves collisions by finding an alternate slot in the table for one of the colliding items.

[1] For example, on most computers, $u = 2^{64}$ since modern hardware uses 64-bit registers. However, no computer in the world has enough RAM (and probably never will have) to store 2^{64} items

- A **closed addressing** hash table resolves collisions by storing colliding items in a separate data structure.

For this lecture, we will assume that collisions are handled using the strategy of **separate chaining**, which is a closed addressing scheme where each entry in A is a list[2] containing a a set of items with the same hash code.



To insert an item with key x, we check whether the list at $A[h(x)]$ already contains an item with key x. If so, we replace it. Otherwise, the item is inserted into the list. Given this, the time to find, insert, or delete an item with key x is $O(|A[h(x)]|)$, i.e., the length of the list $A[h(x)]$. If h is a "good" hash function, then our hope is that the lists will be small. So, *our main goal will be to be able to analyze how big these lists get.*

Prehashing non-integer keys: One issue that we sweep under the rug in theory but that matters a lot in practice is dealing with non-integer keys. Hash tables in the real world are frequently used with data types such as strings, so we want this to be applicable.

The way that we get around this in theory is to require non-integer key types to come equipped with a *prehash* function, i.e., a function that converts the keys reasonably uniformly into integers in the universe U. Then we can proceed as normal assuming integer keys.

Remark: Hashing versus Prehashing (Theory versus Practice)

In programming languages, the term prehashing is not often used; most languages simply refer to prehashing as "hashing" (e.g., `hash` in Python and `std::hash` in C++ take a value of any (hashable) type and return a (usually) 64-bit integer). They then map this integer to an integer in the range $\{0, 1, \dots, m-1\}$ in the implementation of the hash table.

The definition of hashing used in this class is the one used in theoretical computer science in the analysis of algorithms. That is, a hash function is a mapping from a large domain of integers $\{0, 1, \dots, u-1\}$ into a much smaller range of integers $\{0, 1, \dots, m-1\}$.

[2] Separate chaining is usually described as using *linked lists* to store colliding items. For most theoretical purposes however, the kind of list (e.g., linked list vs. dynamic array) is not particularly important. There are minor tradeoffs in practice but they are not relevant to our analysis.

2.2 Properties of hash functions

The main desired properties for a good hashing scheme are:

1. The keys are nicely spread out so that we do not have too many collisions, since collisions affect the time to perform finds and deletes.
2. $m = O(n)$: in particular, we would like our scheme to achieve property (1) without needing the table size m to be much larger than the number of items n.
3. The function h is fast to compute. In our analysis hash table algorithms, given an arbitrary hash function h, we will assume that the time to compute $h(x)$ is $O(1)$. This is a reasonable assumption because such good quality hash functions exist!

Defeating a hash function Since a hash function maps from a large set $\{0, 1, \ldots, u-1\}$ to a small set $\{0, 1, \ldots, m-1\}$, how do we achieve Property (1)? It sounds like no matter what function we choose, there will be some input that pigeonholes us into getting a lot of collisions. Indeed, this is entirely true and unavoidable:

Claim: Every hash function can have lots of collisions

For any hash function h, if $|U| \geq (n-1)m+1$, there exists a set S of n items that all hash to the same location.

Proof. By the pigeonhole principle. In particular, to consider the contrapositive, if every location had at most $n-1$ items of U hashing to it, then U could have size at most $m(n-1)$. □

So, how can hashing be good in theory? For any hash function I choose, you could always find a set of keys to use as input to a hash table that would make operations cost $O(n)$ by forcing every item to collide.

A commonly cited answer—a **very incorrect answer**—to this question, is that hashing is just good in practice but "terrible in theory" because it has $O(n)$ worst-case cost per operation. But this answer completely ignores one of the most powerful tools in algorithm design, **randomized algorithms**. Randomness is what allows us to elevate hashing from the status of "good in practice, bad in theory" to "amazingly powerful in theory and practice"; we will be able to obtain good *worst-case* guarantees, that is, for any input set of keys, the expected performance (or the performance with high probability) of the data structure will be good.

3 Random Hashing

Over the last few lectures, our primary tool for designing efficient algorithms with good worst-case guarantees against adversarial inputs has been *randomness*. To foil an adversary from constructing worst-case inputs to your algorithm, *introduce randomness into the algorithm!*

This is the tactic we used to design algorithms like randomized Quicksort. For any fixed-position pivot selection strategy, you could always find an input sequence that would cause Quicksort to always partition the sequence into sequences of size 0 and $n-1$, thus leading to $\Theta(n^2)$ work.

The solution was to **pick the pivot randomly**. Now, there exists no input sequence that can force the algorithm to run in $\Theta(n^2)$ work. Of course, you might say that it is still *possible* that the algorithm spends $\Theta(n^2)$ work before terminating, if the pivots always land on the smallest or largest element, but the key distinction here is that no particular input can force this to happen; this can only happen by sheer insanely bad luck. And indeed, we can prove, that *with high probability*, this does not happen.

3.1 Analyzing Randomized Data Structures

When analyzing randomized data structures, to make claims like "`find(x)`" costs $O(1)$ in expectation, we need to be quite careful to define *what this actually means*. First and foremost, this means that the cost of our operations is a *random variable*, not just a number.

When working with random variables, it's important to understand the *sample space*. Or, in simpler words, **which part of the analysis is random**. A common misconception is to assume that the *keys* are chosen randomly. This is not accurate—this would correspond to *average-case analysis*, which you may have seen before, but is not what we are doing.

When we analyze a randomized data structure, our analysis is over **any given set of keys and operations on them**, i.e., it is a *worst-case analysis*. The randomness comes in next, where our algorithm makes some random choices. For example, the random choices/sample space for Quickselect are the selection of pivots. For hash tables, the randomness will correspond to making some **random choices about the hash function**.

Definition: Cost of a randomized data structure

When we make a claim about the cost of an operation on a randomized data structure, e.g., `find(x)` costs $O(1)$ in expectation, what we mean, precisely, is:

- The adversary chooses a sequence of operations to perform on the data structure:

  e.g., build a dictionary then perform `insert(5, 2)` then perform `find(3)`.

- We build the data structure and execute the sequence of operations, using random coin flips to guide decisions as specified by the randomized algorithm

The statement "`find(x)` costs $O(1)$ in expectation" means that for **any choice of keys and operations** made by the adversary, the **expected cost** over the sample space of random coin flips, of any `find` operation in the sequence of operations is $O(1)$.

Note that in this definition, the **order** of quantifiers is very important. When we state that the expected cost of an operation is $O(f(n))$, we are stating that

> *for **any** sequence of operations, **the expected value** of the cost over the random choices made by the algorithm, of that operation, is $O(f(n))$.*

In other words, the adversary commits to the input **before** your algorithms flips its coins. If it were the other way around, and the algorithm were required to flip its coins first, the adversary could just find the input that causes the maximum number of collisions for those coin flips. This would essentially make the algorithm not randomized, with a worst-case cost of $O(n)$.

3.2 Totally Random Hashing

So, we need to find a way to add randomness to our hashing algorithms. The first thing that might come to mind is to simply make our hash function return a random number:

```
fun hash(x : int) -> int:
  return random_integer(0, m)    // Extremely wrong, don't do this!
```

Unfortunately, **this is extremely wrong**. The issue is that this function is not a mathematical function; it does not return the same output for the same input (i.e., it is not a *pure function*). This makes it useless as a hash function, because if we use it to store an item in our hash table, when we later want to find that item, we might look in the wrong location. That's totally useless!

So, if our hash function needs to be deterministic, how can it be randomized? That feels like a contradiction! The idea is not to make the hash function itself random, as that would not be a valid hash function, but rather to **choose the hash function randomly**. That is, we randomly select a hash function (which itself is a pure, deterministic function) from a *pool of possible hash functions*, which is often called a random hash family.

What makes random hashing an interesting field is analyzing the properties and tradeoffs of different random hash families.

Definition: Totally Random Hashing

A hash function h is *totally random* if it is selected uniformly at random from the set of *all possible functions* $h : U \to \{0, 1, \ldots, m-1\}$. This is equivalent to selecting, for each key $x \in U$, an independent, uniformly random hash code in $\{0, 1, \ldots, m-1\}$.

Note that the discussion of random hashing often uses some slightly confusing terminology. When we say "a totally random hash function", again, we do not mean that the hash function *behaves* random in any way—what we mean is "a hash function that was selected uniformly randomly from the family of all possible hash functions". The latter, however, is far too verbose, so we accept this confusing choice of terminology.

Totally random hashing has some excellent properties and some horrendous ones. To its advantage, totally random hashing tends to be the easiest family of random hashing to analyze, because its definition implies independent and totally random hash codes for every key. More restricted hash families will have hash codes that are not independent, which complicates (often substantially so) the mathematical analysis.

3.2.1 Collision Probability Analysis

Theorem: Probability of collisions under totally random hashing

Let $\mathcal{H}$ be the family of all possible hash functions $h : U \to \{0, 1, \ldots, m-1\}$. Then, for all $x, y \in U$ such that $x \neq y$, we have

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{m}.$$

Proof. By the law of total probability:

$$\Pr_{h\in\mathcal{H}}[h(x)=h(y)] = \sum_{0\le i<m} \Pr[h(x)=h(y)|h(x)=i]\cdot\Pr[h(x)=i]$$
$$= \sum_{0\le i<m} \Pr[h(y)=i|h(x)=i]\cdot\Pr[h(x)=i].$$

Since $h(x)$ and $h(y)$ are **independent**, $\Pr[h(y)=i|h(x)=i]=\Pr[h(y)=i]$. Since hash codes are uniformly random,

$$\Pr[h(x)=i]=\Pr[h(y)=i]=\frac{1}{m},$$

and therefore

$$\Pr_{h\in\mathcal{H}}[h(x)=h(y)] = \sum_{0\le i<m}\left(\frac{1}{m}\right)\left(\frac{1}{m}\right) = m\left(\frac{1}{m}\right)^2 = \frac{1}{m},$$

as desired. □

So, as we would hope for intuitively, collisions are unlikely when using totally random hashing. Unfortunately, there is also bad news.

3.2.2 Infeasibility of Totally Random Hashing

Theorem: Space requirement of totally random hashing

To store any total function $h : U \to \{0,1,\ldots,m-1\}$ from the set of all possible such functions requires at least $u\log_2(m)$ bits of space.

We won't prove this theorem formally since doing so technically requires information theory, but an informal argument is simple: there are u possible keys, and each hash code is an integer from 0 to $m-1$, which means it takes $\log_2 m$ bits. So, storing u of these takes $u\log_2 m$ bits.

Now, this only proves that the *naive* representation of such a function requires this much space. Perhaps it can be compressed and represented more compactly? Information theory says no. Since there are u possible keys each with m possible hash codes, the number of possible functions is m^u, and information theory says that storing one of these takes $\log_2(m^u)$ bits, which is $u\log_2 m$. In other words, information theory says **randomness is not compressible**.

Unfortunately, this renders totally random hashing completely useless in practice. Despite its nice probability properties, our goal was to be able to store a hash table using an array of size m. Storing a totally random hash function would require an additional u words of memory, but we assume $u \gg m$, so this is terrible. In fact, if we were willing to spend u space, we wouldn't need hashing at all. We could just set $m=u$ and use the "hash function" $h(x)=x$.

4 Universal Hashing

Our study of totally random hashing revealed that while it has some nice properties (i.e., collisions are unlikely), it is infeasible to actually use. Instead, our goal is to find other families of hash functions that can meet a trade-off:

1. still has nice properties (e.g., collisions are rare),
2. can be represented efficiently.

A class of hash families that meets both of these properties is called **universal hashing**. Interestingly, universal hashing does not need to prove Trade-Off (1) above, instead it is *defined* such that collisions are rare, and it then remains for us to find such families that are efficient.

Definition: Universal Hashing

A set of hash functions $\mathcal{H}$ where each $h \in \mathcal{H}$ maps $U \to \{0, \ldots, m-1\}$ is called **universal** (or is called a *universal family*) if for all $x \neq y$ in U, we have

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}. \quad (1)$$

Observe that this definition is just the property that we proved for totally random hashing, except we now allow the probability to be *at most* $\frac{1}{m}$ instead of exactly. This difference just allows edge cases where u is small to still be considered universal, since in those cases you can have fewer or even no collisions.

Here's an equivalent way of looking at this. First, count the number of hash functions in $\mathcal{H}$ that cause x and y to collide. This is

$$|\{h \in \mathcal{H} | h(x) = h(y)\}|.$$

Divide this number by $|H|$, the number of hash functions. This is the probability on the left hand side of (1). So, to show universality you want

$$\frac{|\{h \in \mathcal{H} \mid h(x) = h(y)\}|}{|H|} \leq \frac{1}{m}$$

for every $x \neq y \in U$.

4.1 Some universal families

There are many universal families of hash functions. In this class we won't go through the method of *proving* that a particular family is universal—that's quite challenging and will be done in 15-451.

- The family of **all hash functions**. We proved that this was universal when we showed that

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{m}.$$

- The original universal hash family by its inventors Carter and Wegman: Pick a prime number $p \geq u$ (does not have to be random, can just pick one and hardcode it), then pick **random integers** $0 < a < p$ and $0 \leq b < p$ and define:

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m.$$

  The family of hash functions $\mathcal{H} = \{h_{a,b}(x)$ for all $0 < a < p$, $0 \leq b < p\}$ is universal.

4.2 Using Universal Hashing

Theorem 1: Universal hashing

If $\mathscr{H}$ is universal, then for any set $S \subseteq U$ of size n, for any key $x \in S$ (e.g., that we might want to find), if h is drawn randomly from $\mathscr{H}$, the **expected** number of collisions between x and other keys in S is less than n/m.

Proof. Each $y \in S$ ($y \neq x$) has at most a $1/m$ chance of colliding with x by the definition of universal. So, let the random variable $C_{xy} = 1$ if x and y collide and 0 otherwise. Let C_x be the random variable denoting the total number of collisions for x. So,

$$C_x = \sum_{\substack{y \in S \\ y \neq x}} C_{xy}.$$

We know $\mathbb{E}[C_{xy}] = \Pr(x \text{ and } y \text{ collide}) \leq 1/m$. Therefore, by linearity of expectation,

$$\mathbb{E}[C_x] = \sum_{\substack{y \in S \\ y \neq x}} \mathbb{E}[C_{xy}] \leq \frac{|S|-1}{m} = \frac{n-1}{m},$$

which is less than n/m. □

When a table is storing n items in m slots, this quantity n/m represents how "full" the table is and shows up frequently in the analysis of hashing, so it gets a name. $\alpha = n/m$ is called the **load factor** of the hash table. We now immediately get the following theorem.

Theorem

Insert, find, and delete, on a hash table using universal hashing with separate chaining cost $\Theta(1+\alpha)$ time in expectation.

Proof. The runtime for insert, find, and delete, for a key x is proportional to the number of items in the list at slot $A[h(x)]$ (plus a constant cost to compute the hash function). Suppose x is not currently in the hash table, then by Theorem 1 the expected number of items in $A[h(x)]$ is the number of items in the hash table that collide with x, which is less than $(n+1)/m = \Theta(1+\alpha)$. Similarly if x is currently in the table then then the number of items in $A[h(x)]$ is the the number of items in the hash table that collide with x plus one (itself, since x is definitely at location $A[h(x)]$), so by Theorem 1 the cost is again at most $\Theta(1+n/m) = \Theta(1+\alpha)$. □

Corollary

For any sequence of L insert, find, and delete operations in which there are at most m keys in the hash table at any one time, using separate chaining with universal hashing, the expected total cost of the L operations is only $O(L)$.

Proof. Since there are at most m keys in the table at any time, $\alpha \leq 1$, so every operation costs $\Theta(1+\alpha) = \Theta(1)$ in expectation. By linearity of expectation, the expected total cost is $O(L)$. □

4.3 Dynamic resizing

Using universal hashing, we have shown that we can achieve cost bounds of $\Theta(1+\alpha)$ for insert, find, and delete! This is constant (expected) time if we pick m large enough and make sure we never insert more than $O(m)$ keys.

But what if we don't know n in advance and might insert a lot of keys? Eventually our hash table will cost more than constant time because α will grow large. We learned how to fix this in 15-122, actually. Just pick a threshold, say $\alpha = 1$, and whenever an insert causes $\alpha > 1$, double the value of m and rebuild the hash table from scratch with a new randomly chosen hash function.

Corollary

Using dynamic resizing, find, and delete on a hash table using universal hashing with separate chaining cost $\Theta(1)$ in expectation. insert costs $\Theta(1)$ **amortized in expectation**.