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 type `Dict` satisfies the interface `dictionary<K,V>`, with key type `K` and value type `V`, if it supports:

- `insert(d : Dict, k : K, v : V)`:
  add the key-value pair (k, v) to d
- `find(d : Dict, k : K) -> optional<V>`:
  return the value for the given key k in d (or `NONE` if it doesn't exist)
- `delete(d : Dict, k : K)`:
  delete the item with the given key k from d

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.

Whether a data structure is mutable or persistent should be made clear in its description, and is typically reflected in the return types of its update operations. In a persistent dictionary, `insert` would be typed as:

```
insert(d : Dict, k : K, v : V) -> Dict
```

i.e., it would explicitly return a *new* dictionary, leaving the old one unmodified. In a mutable dictionary, `insert` instead returns nothing (`unit` or `void` depending on your favourite language) since its behavior is to *modify* the given dictionary.

To avoid defining every interface twice (mutable and persistent), we will omit the return types of update operations in this lecture. The same descriptions then apply to both kinds of implementations: persistent updates return a new collection, while mutable updates modify the existing one.

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

The interface `SortedDict<K,V>` *extends* `dictionary<K,V>` by requiring the keys type `K` be *totally ordered* and adding the following operations for a type `Dict`:

- `first(d : Dict) -> option<(K,V)>`:
  return the item with the least key (or `NONE` if empty)
- `last(d : Dict) -> option<(K,V)>`:
  return the item with the greatest key (or `NONE` if empty)
- `prev(d : Dict, k : K) -> option<(K,V)>`:
  returns the item with the greatest key less than k (or `NONE` if no such key exists)
- `next(d : Dict, k : K) -> option<(K,V)>`:
  returns the item with the least key greater than k (or `NONE` if no such key exists)

Since they support more operations, sorted dictionaries typically require a higher cost than unordered 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 type `Set` satisfies the interface `set<K>`, where `K` is the type of its keys, if it supports:

- `insert(S : Set, k : K)`:
  add the given key k to S
- `contains(S : Set, k : K) -> bool`:
  return `True` if S contains k, `False` otherwise
- `delete(S : Set, k : K)`:
  delete k from S
- `union(S1 : Set, S2 : Set) -> Set`:
  return a set containing all keys in either S_1 or S_2
- `intersection(S1 : Set, S2 : Set) -> Set`:
  return a set containing all keys in both S_1 and S_2
- `difference(S1 : Set, S2 : Set) -> Set`:
  return a set containing all keys in S_1 but not in S_2

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

Interface: Sorted Set ADT

The interface `SortedSet<K>` extends `set<K>` by requiring the key type `K` to be totally ordered and adding the following operations for a type `Set`:

- `first(S : Set) -> option<K>`:
  return the least key in S (or `NONE` if empty)
- `last(S : Set) -> option<K>`:
  return the greatest key in S (or `NONE` if empty)
- `prev(S : Set, k : K) -> option<K>`:
  return the greatest key in S less than k (or `NONE` if no such key exists)
- `next(S : Set, k : K) -> option<K>`:
  return the least key in S greater than k (or `NONE` if no such key exists)

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 typically implemented as an imperative/mutable data structure since they are much less naturally suited 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 implement 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

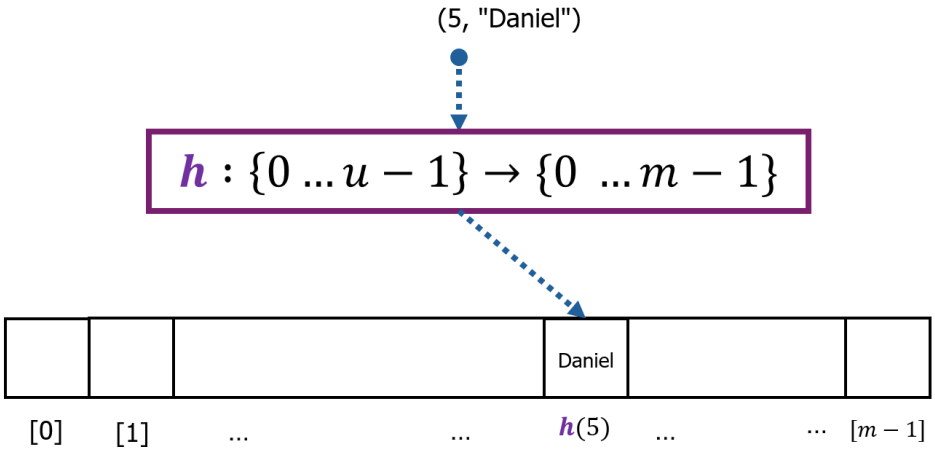
Given a universe of keys U and an integer m (where typically it is assumed that $m \ll |U|$), a hash function is a mapping from U to the integers $\{0, \ldots, m-1\}$, i.e.,

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

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 a collection of items whose keys form a set $S \subseteq U$ (which may change over time with `insert` and `delete` operations). Let $n = |S|$ be the *size* of the hash table. 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 $A[h(x)]$.



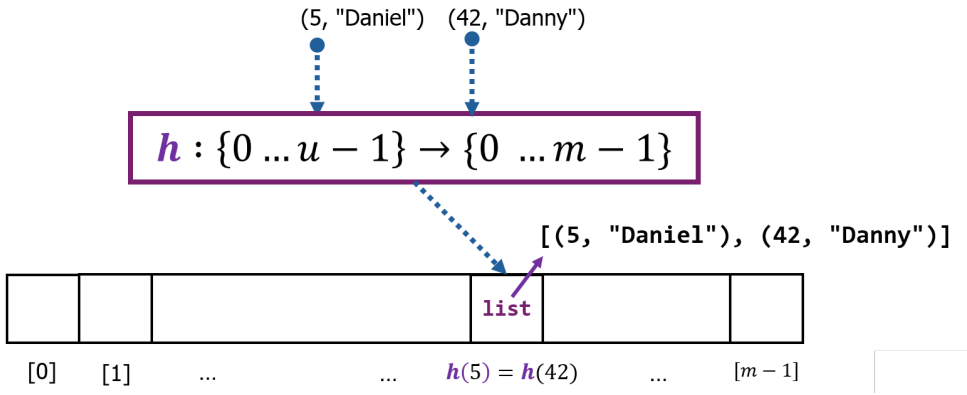
The motivation and analysis of hashing typically assumes that $n \ll u$ (i.e., n is *much smaller* than u), since if the universe was small, you could just store the item with key x in $A[x]$ directly; no need for hashing at all! Such a data structure is often called a *direct-address table* (or a direct-access array). When the universe is assumed to be very very big, this strategy would use an impossible amount of memory[1], hence 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.

[1] For example, if we assume 64-bit integer keys, then $u = 2^{64}$. However, no computer in the world has enough RAM (and probably never will have, although AI data centers are trying their hardest) to store 2^{64} items

- An **open addressing** hash table resolves collisions by finding an alternate slot in the table for one of the colliding 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. Assuming evaluating $h(x)$ costs $O(1)$, the cost to find, insert, or delete an item with key x is

$$\Theta(1 + |A[h(x)]|),$$

i.e., constant work to compute the hash value and access the array, plus work proportional to the length of the list. If h is a "good" hash function, then our hope is that the lists will be small. So, *our main goal is to be able to analyze how big these lists get.*

Hashing 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 them to work even outside the theoretical niceness of the assumption that everything is an integer.

To handle such keys, we can first convert them into representative integers in the universe U, and then use our hash function to map these integers into the range $\{0, \ldots, m-1\}$. This gives us a *two-stage* process for hashing arbitrary objects.



[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.

Remark: Hashing Terminology

Unfortunately, the terminology used to describe these two stages is not entirely consistent across textbooks and programming languages.

- Sometimes, converting an arbitrary object into an integer is called *prehashing*, and mapping that integer into the range $\{0, \ldots, m-1\}$ is called *hashing*.
- Sometimes, the first stage is called *hashing* (e.g., Python's `hash` or C++'s `std::hash`), and the second stage is called *compression*.
- Sometimes, the entire two-stage process is simply called *hashing*.

In this course, we will focus entirely on the latter stage: mapping an integer from a large universe U to an integer in the much smaller range $\{0, \ldots, m-1\}$, since this is where the mathematical analysis of why hash tables are efficient lives.

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 increase the cost of operations,
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 cheap to compute.

In our analysis of hash table algorithms, given a hash function h, we will assume that the cost to compute $h(x)$ is $O(1)$. Later, we will see that there are hash families that satisfy this requirement while also providing good collision guarantees, so this assumption is achievable in practice.

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:

Theorem: Collisions are inevitable

For any hash function h, if $|U| > (n-1)m$, 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, it is still *possible* that the algorithm spends $\Theta(n^2)$ work if the pivots always land on the smallest or largest element, but the 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.

Random hashing is the same solution applied to hash tables. Since for any fixed hash function we could commit to, an adversary could construct a terrible input for it, we are going to **pick the hash function randomly**. We can then prove that this results in good cost bounds *for any input*.

3.1 Analyzing Randomized Data Structures

Before we can analyze randomized hash tables, we need to carefully define what we actually mean when we say things like "`find(x)` costs $O(1)$ in expectation." What exactly does that statement mean, and what are we proving mathematically?

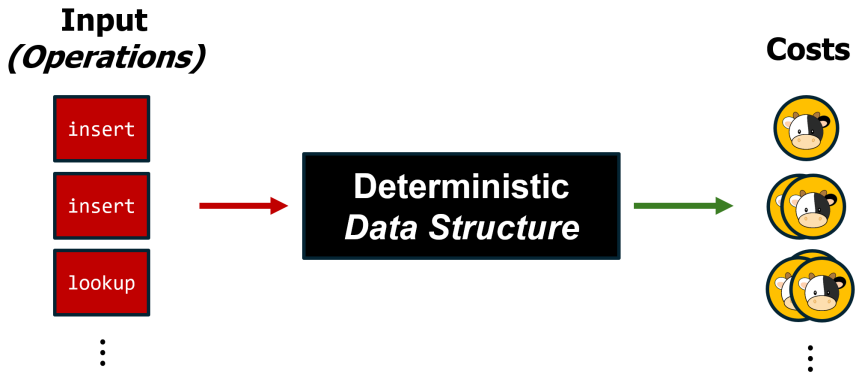
Analyzing deterministic algorithms When analyzing a deterministic algorithm, every possible input to the algorithm has a fixed cost, so we typically to to analyze its *worst-case cost*: the maximum cost over all possible inputs.



Of course, when we talk about the maximum cost over all possible inputs, we must *parameterize* this by the size of the input. Otherwise, the maximum cost would generally be unbounded, since we could simply consider larger and larger inputs! For an ordinary algorithm, we typically

consider inputs of size n. The precise size parameter depends on what we are analyzing, and need not be called n. There may even be multiple parameters.

Analyzing deterministic data structures For a data structure, we typically consider operations on a data structure containing n elements. However, we often care about more than just the cost of an individual operation. The costs of successive operations are related: for example, an expensive operation might only be possible after many cheap operations. This motivates analyzing entire *sequences of operations*, rather than individual operations in isolation.

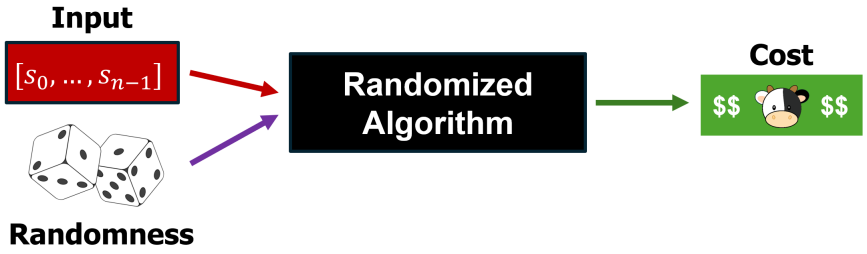


When analyzing data structures, we therefore consider two kinds of cost bounds, over all possible operation sequences on the data structure, parameterized by its size[3]:

- **Worst-case cost:** The **maximum cost of any individual operation**, over all possible sequences.
- **Amortized cost:** The **maximum average cost per operation**, over all possible sequences.

Analyzing randomized algorithms Now, what changes when we introduce *randomization*? Recall that for a randomized algorithm, the cost depends not only on the input, but also on the random choices made by the algorithm. Even if we fix the input, different random choices may result in different costs. Thus, for any fixed input, the cost is a *random variable*.

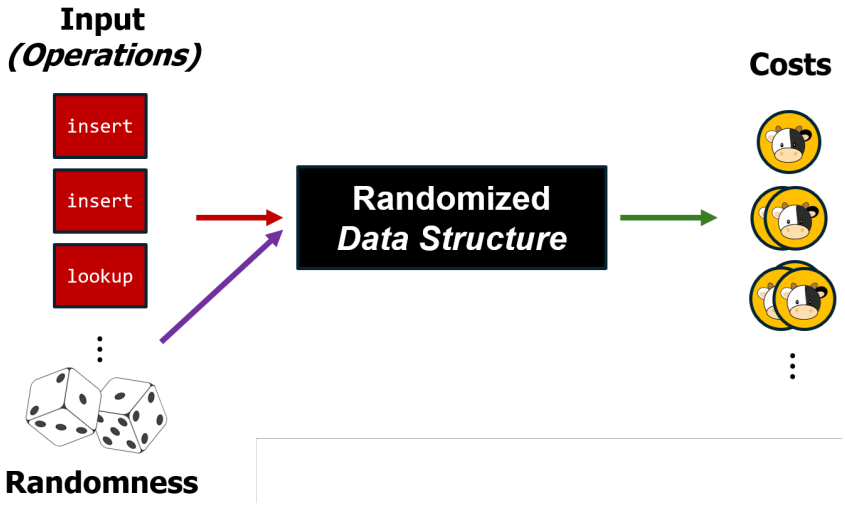
Importantly, we still care about *worst-case* guarantees! We can take the expected cost over the algorithm's random choices for each possible input, and then consider the **maximum expected cost over all inputs of a given size**. This gives us a *worst-case expected cost* bound: a guarantee that holds for every possible input, regardless of how adversarial it might be.



[3]The size of the data structure may *change* throughout a sequence, e.g., as more elements are inserted, so the precise parameterization of an amortized bound depends on the analysis and is out of scope here (see 15-451).

Analyzing randomized data structures The same idea applies to *randomized data structures*. The difference is that our input now consists of a sequence of operations. Even if we fix the entire sequence, the costs of individual operations may vary depending on the random choices made by the data structure.

For example, in a randomized hash table, we randomly choose which hash function to use. Different choices of hash function may lead to different collisions, and hence different costs for the same sequence of operations.



Just as before, we still care about *worst-case* guarantees! For each possible operation sequence, we can take the expected cost of an individual operation over the data structure's random choices, and then consider the **maximum expected cost over all possible operation sequences**. This gives us a *worst-case expected cost* bound for an individual operation.

Alternatively, we can analyze the total cost across a sequence of operations, giving us an *amortized expected cost* bound. The key insight is that randomization introduces a *new dimension* to our cost analysis, without replacing the distinctions we already had.

Definition: Analyzing Randomized Data Structures

There are two independent dimensions of our cost analysis:

- **Worst-case vs. amortized:** How we measure costs across possible operation sequences.
- **Expected vs. with high probability:** How we measure costs over the random choices made by the data structure.

In particular, an operation has *worst-case expected cost* $O(f(n))$ if its expected cost is $O(f(n))$ for **every possible operation sequence**, where the expectation is taken over the data structure's random choices.

These guarantees assume that the operation sequence is chosen independently of the data structure's random choices. Otherwise, someone who knew our randomly chosen hash function could deliberately choose keys that collide!

Remark: Don't Confuse Different Kinds of Averages!

The complexity analysis of hashing can easily become confusing because we talk about several different kinds of *averages*. Three different metrics are all defined as the average of *something*, but they mean very different things:

- **Average-case analysis:** We take the average cost[a] over all possible *inputs* to the algorithm. This is an *alternative* to the worst-case cost.
- **Amortized analysis:** We consider the maximum possible average cost across *operations in a sequence*, even if some individual operations are expensive. This is also an *alternative* to the worst-case cost.
- **Expected-cost analysis:** We take the average over the *random choices made by the algorithm or data structure*, such as random pivots in Quicksort or the random choice of a hash function in a hash table. This is *complementary* to and is combined with worst-case cost or amortized cost.[b]

[a] typically assuming uniformly random inputs; generally, inputs can follow any probability distribution
[b] Technically, it would also be valid to ask for average-case expected cost, but no one ever does that.

3.2 Totally Random Hashing

So, we need to find a way to add randomness to our hashing algorithms. Just like how randomized Quicksort defeated the evil adversary by picking its pivots randomly, random hashing must fool the adversary by **picking the hash function randomly**. But what does that actually mean? The first thing that might come to mind is to simply make our hash function itself random:

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

Surely the adversary can't guess that! 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 *pure*). This makes it useless for hashing, 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!

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 $\mathscr{H}$ 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

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

Proof. By the law of total probability:

$$\begin{aligned}\Pr_{h \in \mathscr{H}}[h(x) = h(y)] &= \sum_{0 \leq i < m} \Pr[h(x) = h(y) | h(x) = i] \cdot \Pr[h(x) = i] \\ &= \sum_{0 \leq i < m} \Pr[h(y) = i | h(x) = i] \cdot \Pr[h(x) = i].\end{aligned}$$

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

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

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 an arbitrary 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 storing u random $(\lg m)$-bit hash codes, 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 $\mathscr{H}$ where each $h \in \mathscr{H}$ 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

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

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 $\mathscr{H}$ that cause x and y to collide. This is

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

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

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

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

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

- 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:

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

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

4.2 Using Universal Hashing

Theorem: Universal hashing

Let $\mathscr{H}$ be a universal family and let $S \subseteq U$ be a set of n keys. For any key $x \in U$, if h is drawn randomly from $\mathscr{H}$, the **expected** number of keys $y \in S \setminus \{x\}$ that collide with x (i.e., $h(x) = h(y)$) is at most n/m.

Proof. Define an indicator 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,

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

By the definition of universal, we know $\mathbb{E}[C_{xy}] = \Pr[h(x) = h(y)] \leq 1/m$. Therefore, by linearity of expectation,

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

□

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

Given a hash table using universal hashing and separate chaining with load factor $\alpha = n/m$, `insert`, `find`, and `remove` have $O(1+\alpha)$ **worst-case expected** cost.

Proof. Using separate chaining, the cost for insert, find, and remove, for any key x is

$$\Theta(1 + |A[h(x)]|)$$

By the universal hashing theorem, the expected length of $A[h(x)]$ is at most $1 + n/m$, since the number of colliding elements is at most n/m plus one for x itself. Hence for any key x, the expected cost is at most $O(2 + n/m) = O(1+\alpha)$. □

4.3 Dynamic resizing

Using universal hashing, we have shown that we can achieve cost bounds of $O(1+\alpha)$ for insert, find, and remove! 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. You learned how to fix this in 15-122. 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 universal hashing and separate chaining *with dynamic resizing*, `find`, and `remove` have $\Theta(1)$ **worst-case expected** cost, and `insert` has $\Theta(1)$ **amortized expected** cost.

The space used by the table will be proportional to the maximum number of keys that were ever present at once. If desired, a stronger space bound can be achieved in which the table always has size $\Theta(n)$, the current number of keys, by *shrinking* the table whenever it goes *below* a load factor of say 0.25. This makes `remove` have amortized cost as well.