A little context

So far, we’ve discussed searching and sorting in a comparison-based model, and were given a convincing argument for comparison-based data structures allowing no better than $O(n \times \log(n))$, and lookups no faster than $O(\log(n))$. We have also seen a few data structures that sidestep these lower bounds by working outside the comparison model:

- Arrays allow $O(1)$ lookup by employing integer keys $k \in \{0...n-1\}$. - We have also seen an example of sorting faster than $O(n \times \log(n))$ by, again, using integer keys in a limited range (counting sort).

Dictionaries

A dictionary is an abstract data structure (ADT) supporting the following interactions:

Given a dictionary $D$,

- **Insert**($k$, $v$) creates a mapping $k \rightarrow v$ within $D$. If $k$ is already mapped in $D$, meaning $\exists v'|(k \rightarrow v') \in D$, the mapping is updated to $k \rightarrow v$. Insertion should take constant ($O(1)$) time, meaning it should be roughly independent of the size of the data set stored in the dictionary. Assume unique keys, meaning no two mappings $k \rightarrow v$ and $k \rightarrow v'$ with the same key may exist. Multiple keys may map to the same value.

- **Delete**($k$) removes the mapping $k \rightarrow v$ for some $v$ within $D$, if $k$ is mapped (remember, keys are unique, so there is at most 1 such mapping). If no such mapping exists, $D$ is unchanged. Deletion should take $O(1)$ time also.

- **Search**($k$) returns $v$ if the mapping $k \rightarrow v$ for some $v$ exists in $D$. If $k$ is not mapped in $D$, **Search** fails, returning **None**. Search should take $O(1)$ time, meaning it should be roughly independent of the size of the data set stored in the dictionary.

In a well formed hash table (dictionary), each of these operations take on average $O(1)$ time, making hashes a very useful and versatile data structure.

Hashing

If the key space were a small space of integers, we could use an array to implement a dictionary! Assume $m$ values with unique keys in $\mathbb{N}_m = \{0, 1, ..., (m-1)\}$. We can now use an array $A$ of $n$ elements to store all $n$ values: any key-value pair $(k, v)$ is stored by $A[k] = v$. Observe that insertions, and searches are correctly $O(1)$ time. Deletions are $O(1)$ also, though are a bit awkward – we need to store **None** at each unmapped location in $A$. 

ANDREEA’S DIAGRAM

What about a larger key space, or even a space with non-integer keys (string, or even composite arbitrary objects). Suppose we now wanted to store integer keys in all $\mathbb{N} =$
Clearly, we cannot allocate an array with $|\mathbb{N}|$ entries, as the space of natural numbers is not finite, but we can transform the keys to a smaller, finite space. Doing so is called “hashing” the key. As a simple example, let $hash(k) = (k) \mod (m)$. The space of hashes of keys is now exactly $\mathbb{N}_m = \{0, 1, \ldots, (m - 1)\}$, which we know how to map to a simple array, but hashes of keys are not necessarily unique.

**Hash collisions: chaining**

Hash collisions are somewhat unavoidable: by mapping a large key space to a small space of key hashes, some repeated keys are expected to occur. In 6.006, we will see numerous ways of dealing with hash collisions. The method described below is one of the simplest, and is called “chaining”.

**DIAGRAM FOR CHAINING.**

Instead of storing values directly, each table entry maps to a linked list of key-value pairs with the same hash. When inserting a new key-value pair $(k,v)$, append $(k,v)$ to the end of the list $A[hash(k)]$. When searching for the key $k$, we must search all key-value pairs in $A[hash(k)]$ - remember this takes time proportional to the length of this list! Likewise, deletion of a key $k$ is simply a deletion of the key-value pair corresponding to $k$ from $A[hash(k)]$, again a linear-time operation.

Assuming a perfect hash function ($hash(k)$ is uniformly distributed across $\mathbb{N}_m$), it is not hard to show that the expected length of $\alpha = \frac{n}{m}$. We denote $\alpha$, the “loading factor” of the hash table. By ensuring $m$ is large enough to keep $\alpha$ small, searches and deletions in a hash table using chaining take $O(1 + \alpha)$, approximately $O(1)$ time.

**Bad hash functions**

**Credit card numbers**

Credit card numbers, although large, have a lot of structure. The first 4 of the 16 digits specify the bank, and are therefore bad candidates for a hash function. MIT students likely have a very small set of banks issuing cards (MITFCU, Bank of America, Citi, etc.), meaning the space of hashed credit card numbers would be poorly distributed across the space of hash values. The last digit of a credit card is a checksum entirely specified by the previous digits, and therefore does not contribute any variation to the hash function.

**Hash via XOR**

Let a key $k$ be a list of integers.

```python
1 def xorhash(key):
2     h = 0
3     for k in key:
4         h ^= k
```
Good hash functions

Suppose our hash function was given by \( h(k) = 0 \). This looks bad: all keys map to the 0th entry in the hash table, resulting in an \( n \)-element chain. As a result, searches and deletions in this unfortunately hashed table take \( O(n) \) time. While this is clearly the worst case, what exactly is the best case?

More generally, here are four main characteristics of a good hash function:

- satisfies (more or less) the assumption of simple uniform hashing: each key is equally likely to hash to any of the \( m \) slots. The hash function should not bias towards any subset of the hash space.

- does not hash similar keys to the same slot (for example in a compiler, the variables \( i \) and \( j \) in a symbol table should not hash to the same slot, as they are often used together).

- is very quick to calculate. A hash function should certainly have \( O(1) \) time complexity.

- is deterministic. \( \text{hash}(k) \) should be well-defined for a given key \( k \).

Examples of hash functions

Division method

The division method is one way to create hash functions. The functions take the form

\[
\text{Division method: } h(k) = k \mod m
\] (1)

Since we’re taking a value \( \mod m \), \( h(k) \) does indeed map the universe of keys to a slot in the hash table. It’s important to note that if we’re using this method to create hash functions, \( m \) should not be a power of 2. If \( m = 2^p \), then the \( h(k) \) only looks at the \( p \) lower bits of \( k \), completely ignoring the rest of the bits in \( k \). A good choice for \( m \) with the division method is a prime number (why are composite numbers bad?).

0.1 Multiplication method

The multiplication method is another way to create hash functions. The functions take the form
\[ h(k) = \lfloor m(kA \mod 1) \rfloor \]  

where \( 0 < A < 1 \) and \((kA \mod 1)\) refers to the fractional part of \(kA\). Since \( 0 < (kA \mod 1) < 1 \), the range of \(h(k)\) is from 0 to \(m\). The advantage of the multiplication method is it works equally well with any size \(m\). \(A\) should be chosen carefully. Rational numbers should not be chosen for \(A\) (why?). An example of a good choice for \(A\) is \(\frac{\sqrt{5} - 1}{2}\).

### Practice problems

#### 0.2 Zero-Sum Game

Suppose that we have a list of \(n\) numbers. How can we detect if two of them add to a target value \(t\)?

*Hint:* Consider hashing the numbers.

Now suppose that the numbers are randomly generated 32-bit integers. Can you solve this problem using sorting? Which approach might be best in practice, and why?

#### 0.3 Mod 9

Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be \(h(k) = k \mod 9\).

#### 0.4 Other practice problems

1. Recall docdist
2. Distinct elements
3. Anagram finder
4. Bi-directional dictionary: translate integers to english words.