CS 228: Introduction to Data Structures
Lecture 33

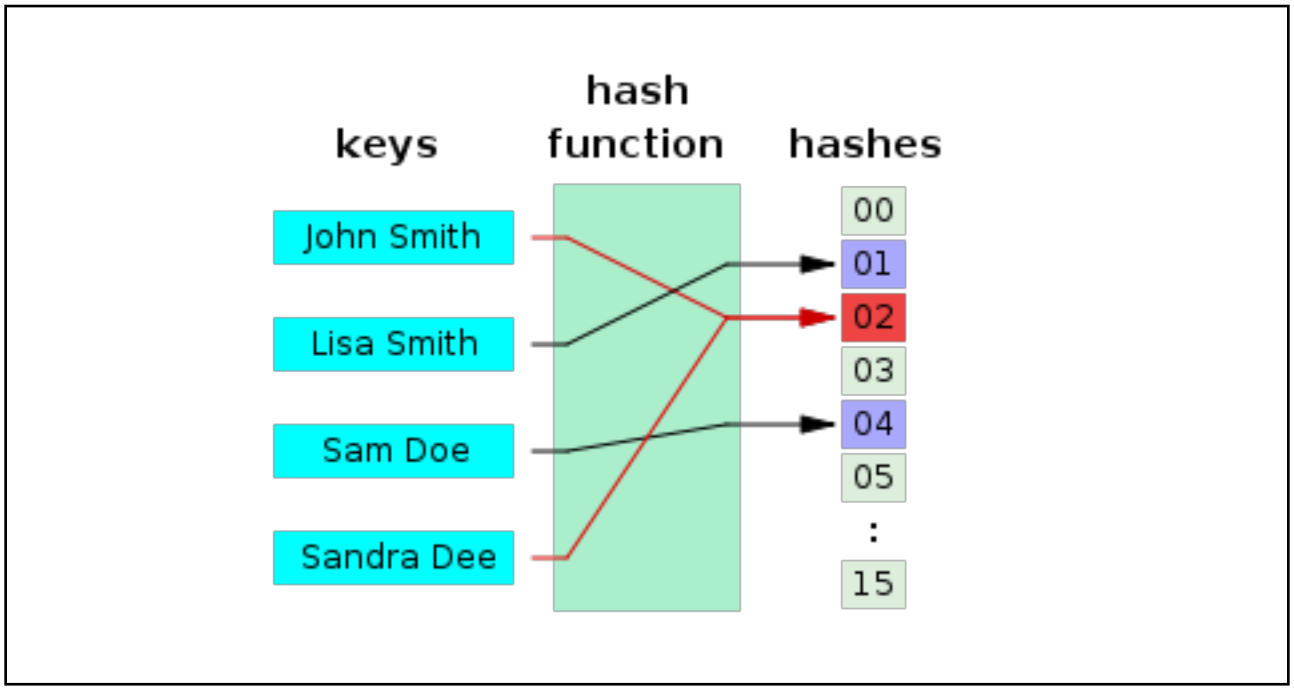
Hashing

A widely used technique to implement maps is via "hashing". One reason for the popularity of hashing is that, under certain reasonable assumptions, it enables us to implement maps so that, under certain reasonable assumptions, `get()`, `put()`, `containsKey()`, and `remove()` take constant expected time. Hashing is the technique behind Java's `HashMap` class.

To get the intuition behind hashing, consider first the case where the keys are integers in a small range. Then, we could implement a `Map` by using a lookup table, represented as an array of values, indexed by key: The value associated with key k would be stored in the array cell k. Adding and looking up entries would be O(1), since we now have random access based on the array index. (Of course, there might be a lot of unused space.)

A ***hash table*** mimics the behavior of a lookup table for key/value pairs where the keys are arbitrary objects. Done properly, a hash table will act as if we had random access into an array. The core of hashing is a ***hash function***, that associates an integer value, called a ***hash code***, with any key. This integer value is used as an index into a table. To put a new (key,value) pair in the table, we do as follows[1].

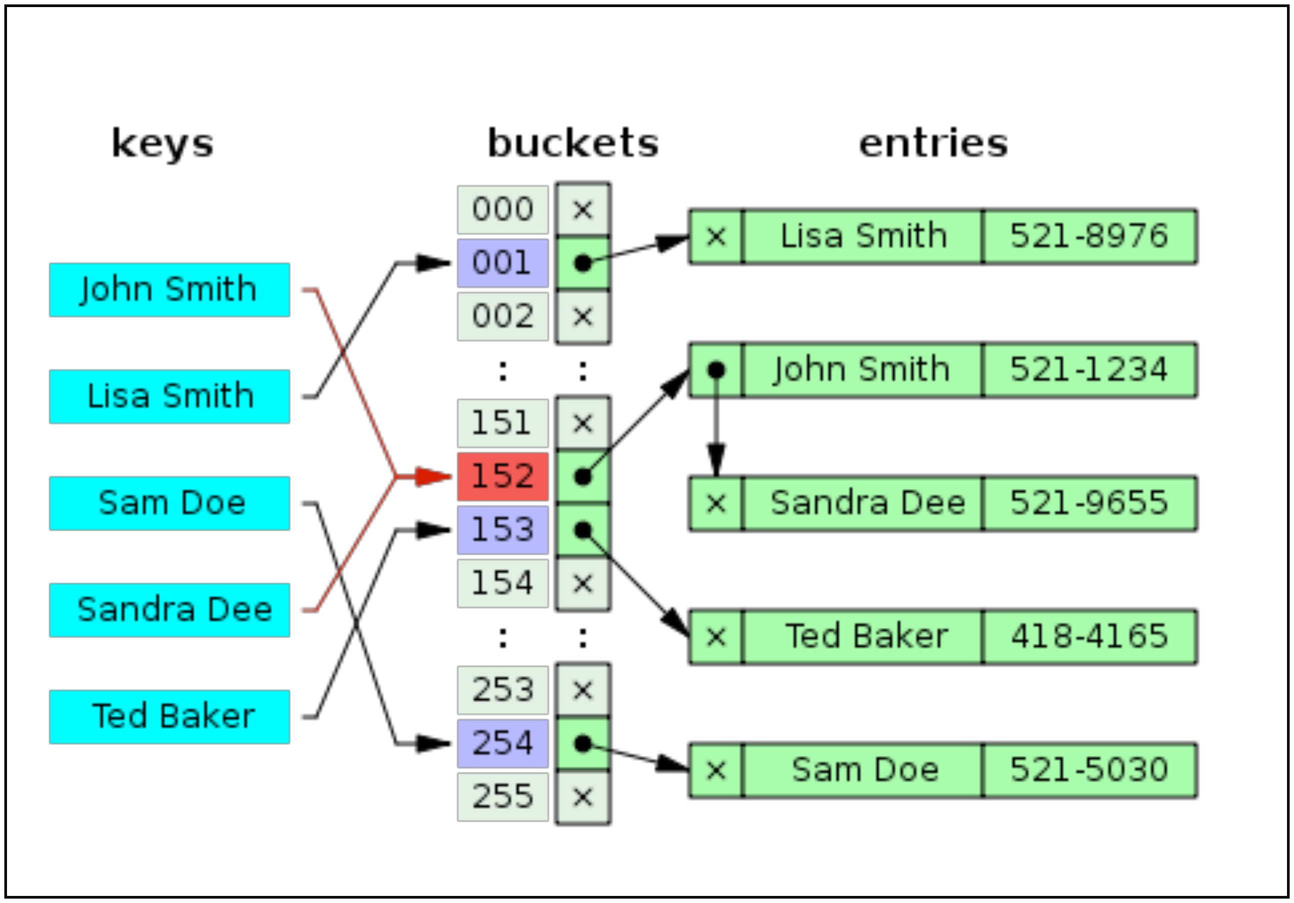
1. Use the hash function, to compute the key's hash code.



2. Take the hash code modulo the array size to get an index into an array.

[1] The figures are from Wikipedia.

3. Store the key and the value in a linked list of entries at the array index.



The linked lists in step (3) are called ***buckets***. They allow us to store multiple values at the same array index, which is needed in case of ***collision***. This is what happens when two different keys end up with the same hash code, or two different hash codes result in the same array index.

To get the value associated with a given key k, we

1. use the hash function to compute k’s hash code,

2. take a modulus of the hash code to find the appropriate bucket, and

3. search the list for an entry whose key equals k.

Hashing works well if

(A) the bucket array is not much bigger than the total number of entries and

(B) the hash function disperses the keys evenly among the buckets.

Suppose n is the number of key-value pairs and M is the size of the bucket array. (A) and (B) roughly say that the table does not waste a lot of space, and that the average number of entries per bucket is n/M.

The `HashMap` Class

`HashMap` is Java’s implementation of hash tables[2]. `HashMap` computes the hash code of a key using the `hashCode()` method for the key’s class. Unless you override it, Java will use the default `hashCode()` method, which every `Object` possesses, and which returns a 32-bit integer. For instance, the `hashCode()` method for the `String` class (as specified in the Java Standard), looks like this:

```
public int hashCode() {
   int hash = 0;
   for (int i = 0; i < length(); i++)
      hash = (hash * 31) + charAt(i);
   return hash;
}
```

(Note that the computation is allowed to overflow.) The two special things about the number 31 are that (i) it is prime, so that when the user mods out by another number, they have no common factors (unless it's a multiple of 31) and (ii) it is a ***Mersenne prime*** (like 127 or 8191), a prime number of the form $2^n - 1$. Thus, the product can be done quickly, with one shift and one subtraction.

[2] You may notice that there is an implementation of the `Map` interface called `HashTable`. This is a leftover from Java 1, and generally should not be used.

To convert a `hashCode()` to an array index, we use the mod ("%") operation to produce integers between `0` and `M–1`:

```
private int hash(Key key) {
   return (key.hashCode() & 0x7fffffff)
     % M;
}
```

The code masks off the sign bit, to turn the 32-bit integer into a 31-bit nonnegative integer. Note that for the `hashCode()` function defined for the `String` class, we want the hash table size `M` to be relatively prime to 31. Programmers commonly use a *prime* number for `M` in code like this, to attempt to make use of all the bits of the hash code. For instance, if we chose `M` = 31, the code for every string would just be the last character, which would be a very bad idea.

To get the value associated with a key k, we go to the bucket determined by the key's hash code and search sequentially through the bucket for an entry whose key is equal to k in the sense of the `equals()` method. (Recall that the search is needed, because even if the `hashCode()` values of two objects are the same, the objects may not be equal.) The fact that HashMap uses `equals()` to identify the right entry leads to a crucial consideration.

The implementation of `hashCode()` for a class must be consistent with the implementation of `equals()` for that class. That is, if `a.equals(b)` is true, then `a.hashCode()` must have the same numerical value as `b.hashCode()`. Thus, if you override `equals()`, you must also override `hashCode()`.

The other Map methods depend on `equals()` and are implemented similarly. For example, `put(key,value)` would look for `key` in the bucket whose index is `key.hashCode()`. If `key` is not in that bucket, we put the pair `(key,value)` in the bucket. Otherwise, return `false`.

Time Complexity of Hashing

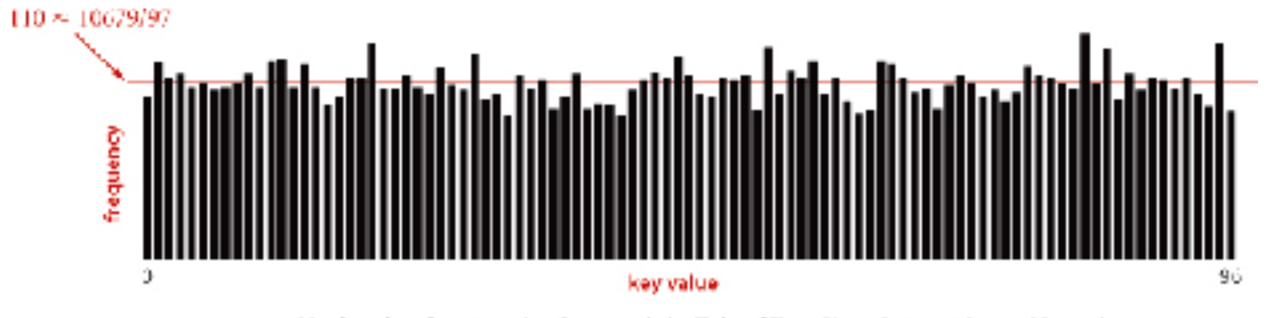
Let n be the number of elements in the hash table and M be the number of buckets. To describe the performance of hashing, we make an assumption:

Uniform hashing assumption. Our hash function uniformly distributes keys among the integer values between 0 and M–1.

Under this assumption, we have the following property.

Fact. In a hash table with M buckets and n keys, the probability that the number of keys in a list is within a small constant factor of n/M is extremely close to 1.

Of course, this fact completely depends on the uniform hashing assumption. Nevertheless, if we use a good function, the same behavior occurs in practice. For example, the next histogram[3] shows that the standard `hashCode()` method for strings gives a nice dispersion of the words from Charles Dickens' *Tale of Two Cities* on a table with M = 97 buckets.



Hash value frequencies for words in *Tale of Two Cities* (10,679 keys, *M* = 97)

The next table summarizes the time complexities of the operations on the three implementations of Maps we have seen in class.

[3] From Sedgewick and Wayne (2011).

	`put()` `get()` `remove()` `containsKey()`	**Iterating over the collection**	**Notes**
List	O(n)	O(n)	
Balanced BST: `TreeMap`	O(log n)	O(n)	

Hash Table: `HashMap`	O(n/M)
	
	O(1)
		O(n+M)
		
		O(n)
			
			If M ≥ n and M is O(n)

Ordering and Iteration

Most hash table implementations, including `HashMap`, offer no guarantees about the iteration order of elements. In fact, `HashMap` does not even guarantee that the order will remain constant over time. In contrast, `TreeMap` always iterates through the key set in the same order (either the natural ordering or one based on a specified comparator). So, if you frequently need to access the elements in a particular order, consider using a `TreeMap` instead of a `HashMap`.

How to Write a Pretty Good Hash Function

A good hash function should

- be ***deterministic***—equal keys must produce the same hash value—,
- be ***efficient to compute***, and
- ***distribute the keys uniformly*** —after all, the point of hashing is to get close to O(1) lookup time (ideally the lists at each bucket would be of length 1, or at least be very short).

The best way to understand good hash functions may be to understand why bad hash functions are bad. Here are two ***bad*** hash functions on strings.

- *Sum up the ASCII values of the characters.* Unfortunately, the sum will rarely exceed 500 or so, and most of the entries will be bunched up in a few hundred buckets. Moreover, anagrams like “pat,” “tap,” and “apt” will collide.
- *Use the first three letters of a word, in a table with 26^3 buckets.* Unfortunately, words beginning with “pre” are much more common than words beginning with “xzq”, so the former will be bunched up in one long list. This is far from our uniformly distributed ideal.

Java's `hashCode()` function for strings does not have such obvious flaws. Aside from that, it seems to work well in practice. Note, though, that it is possible to come up with (admittedly very artificial) examples where the function performs badly.

So, how can you design a good hash function? There is a lot of discussion on this subject on the web, and several hash functions have been proposed. Although designing good hash functions seems to involve some amount of magic, there is one good general principle:

> The hash code produced by the hash function should incorporate ***all*** the data in the key.

For instance, Java's hash function for strings uses all the characters, as well as their order. We can extend this idea to get hash functions for arbitrary objects like this:

```
hashCode():
    hash = some initial value
    foreach instance variable v used by equals()
    |   c = v.hashCode()                              (*)
    |   hash = hash * 31 + c
    return hash
```

To compute the hash code of an array, for example, the instance variables considered in the **foreach** loop would be the individual array entries. Note that we allow `hash` to overflow during the calculation.

Line (*) assumes that each instance variable `v` has a hash code, which can be computed recursively. In particular, if `v` is a primitive value, it is converted to an `int`, if necessary. Here's how you can compute a hash code `c` for different primitive types:

- `v` is a `short`, `char`, or `byte`: Set `c = (int) v`
- `v` is a `boolean`: Set `c = (v? 0:1)`. I.e.; if `v` is true, `c` is `0`, else it is `1`.
- `v` is a `float`: Set `c = Float.floatToIntBits(v)`.[4]
- `v` is a `long`: Set `c = (int)(v ^ (v >>> 32))`. This performs the XOR of the lower 32 bits with the upper 32 bits.

[4] `floatToIntBits` returns an `int` representing the specified floating-point value according to the IEEE 754 floating-point "single format" bit layout.

- `v` is a `double`: Set `long x = Double.doubleToLongBits(v); c = (int)(x ^ (x >>> 32))`.[5]

Rehashing and the Load Factor

If a hash table is implemented using linked lists for the buckets, we can add new entries indefinitely without expanding the array. However, as the table becomes fuller, collisions increase, the bucket lists get longer and longer, and performance deteriorates.

When a hash table becomes too full, collisions increase, and performance deteriorates. To avoid this, it helps to, from time to time, expand the array and ***rehash*** the elements. More precisely, we expand and rehash when

n / `M` = (number of entries) / (number of buckets) > L,

where L is a predetermined value called the ***load factor***. If the load factor is exceeded, the table size is rehashed so that the hash table has approximately twice the number of buckets. (Rehashing, by the way, is the reason why the

[5] `doubleToLongBits` returns a `long` representing the specified floating-point value according to the IEEE 754 floating-point “double format” bit layout.

iteration order may change.) The default load factor for `HashMap` is 0.75, a number that offers a good tradeoff between time and space costs. A higher load factor decreases the space overhead, but increases the time to look up an entry.

Besides the load factor, another parameter that affects the performance of hashing is the initial table capacity; i.e., the initial number of buckets allocated to the table. The initial capacity should be chosen so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, rehashing will never be needed.   `HashMap`'s default initial capacity is 16.