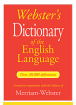
ANNOUNCEMENTS

today is **No Laptop For the First Half of Lecture Monday**!
today is **Prof. Bill Jannen Is Visiting LEC-02 Monday**!

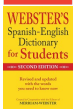
WARMUP

long, long, ago, many people owned a large, physical book called a ✨Dictionary✨
- what was it used for?
- did you know there were different kinds of dictionaries?

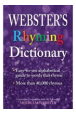
TODAY maps (dictionaries, tables)



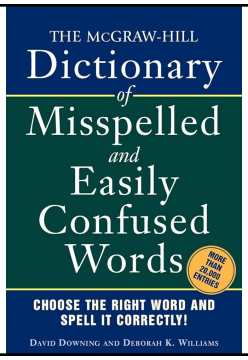
```
HashMap<String, String> dictionary;
// {Dog=A cute, four-legged mammal, ...}
```



```
HashMap<String, String> englishToSpanishDictionary;
// {Gato=Cat, Perro=Dog, ...}
```



```
HashMap<String, ArrayList<String>> rhymingDictionary;
// {Dog=[Bog, Cog, Frog], ...}
```



Google

maps

map interface

map

- a **pair** is two things
- a **map** (**dictionary**, **table**) stores **key-value pairs**
  - a map is used to **lookup** a key's value
  - a map in Java's standard library is `HashMap<KeyType, ValueType>`
    - `HashMap<String, ArrayList<String>> rhymingDictionary;`

let us imagine...

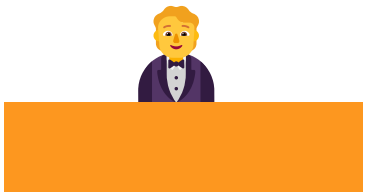
you have a three week long road trip planned to Bennington, but your prized parrot Hans is deathly afraid of driving

it's time to **put** Hans in the bird kennel

you return invigorated from a lovely trek to Bennington

it's time to **get** Hans from the bird kennel

HashMap<String, Bird>



map interface

```
// Put (add, insert) a new key-value pair into the map.
// NOTE: Can also be used to update a key's value.
void put(KeyType key, ValueType value);

// Get (lookup) a key's value in the map.
ValueType get(KeyType key);

// Get the (unordered) set of all keys.
// NOTE: Use this to iterate through all keys.
//       for (KeyType key : map.keySet()) { ... }
Set<KeyType> keySet();
```

motivating example

Pokémon

- there are 151 **Pokémon**
- each has its own **number**
  1. Bulbasaur
  2. Ivysaur
  3. Venusaur
  4. Charmander
  5. ...

number → name

```
1 → "Bulbasaur"
2 → "Ivysaur"
3 → "Venusaur"
4 → "Charmander"
...
```

number → name (Option I)

```
String[] names = { "Bulbasaur", "Ivysaur", ... };
```

```
int number = ...;
String name = names[number - 1]; // NOTE: Bulbasaur is #1
```

number → name (Option II)

```
String[] _names = { "Bulbasaur", "Ivysaur", ... };
String getName(int number) {
    return _names[number - 1];
}
```

```
int number = ...;
String name = getName(number);
```

number → name (Option I)

```
String[] names = { "Bulbasaur", "Ivysaur", ... };
```

```
int number = ...;
String name = names[number - 1]; // NOTE: Bulbasaur is #1
```

number → name (Option III)

```
String[] names = { "", "Bulbasaur", "Ivysaur", ... };
```

```
int number = ...;
String name = names[number];
```

lesson: if you're mapping from A → B, and A is something like 0, 1, 2, 3, ...

...then you can just use an array 😳 👍

buuut, if you're mapping from A → B, and A is *nothing* like 0, 1, 2, 3, ...

...then you're going to want a **map** 🗺️

name → number

🗺️ **solve this problem using a HashMap<KeyType, ValueType>**

```
"Bulbasaur"  → 1
"Ivysaur"    → 2
"Venusaur"   → 3
"Charmander" → 4
...
```

```
// Put (add, insert) a new key-value pair into the map.
// NOTE: Can also be used to update a given key's value.
void put(KeyType key, ValueType value);

// Get (look up) a key's value in the map.
ValueType get(KeyType key);
```

1. what **data structures** will you need?
2. how will you **set them up**?
3. how will you **use them**?

name → number

```
// data structures
HashMap<String, Integer> numberFromName;
```

```
// setup
String[] names = { "Bulbasaur", "Ivysaur", ... };
numberFromName = new HashMap<>();
for (int i = 0; i < names.length; ++i) {
    // Bulbasaur = 1
    numberFromName.put(names[i], i + 1);
}
```

```
// usage
String name = ...;
int number = numberFromName.get(name);
```

let's implement `numberFromName`

TODO (Jim): use `keySet()` method in test code

`PokemonExample`

```
import java.util.*;

class PokemonExample {
    public static void main(String[] arguments) {
        // setup
        HashMap<String, Integer> numberFromName = new HashMap<>();
        String[] names = { "Bulbasaur", "Ivysaur", "Venusaur", ... };
        for (int number = 0; number < names.length; number++) {
            numberFromName.put(names[number], number + 1);
        }

        // usage
        for (String name : numberFromName.keySet()) {
            System.out.println(name + " -> " + numberFromName.get(name));
        }
    }
}
```

```
Mewtwo -> 150
Raichu -> 26
Arbok -> 24
Kabuto -> 140
Charmander -> 4
Charmeleon -> 5
Shellder -> 90
Fearow -> 22
Kangaskhan -> 115
Mankey -> 56
Dodrio -> 85
Vaporeon -> 134
Golem -> 76
Marowak -> 105
Pikachu -> 25
Raticate -> 20
Kadabra -> 64
Primeape -> 57
Venomoth -> 49
Magikarp -> 129
Pidgeotto -> 17
```

maps in other languages

Python has a built-in map (dictionary)

```
numbers = {}
numbers['Bulbasaur'] = 1
numbers['Ivysaur'] = 2
print(numbers['Bulbasaur']) # 1
print(numbers) # {'Bulbasaur': 1, 'Ivysaur': 2}
```

```
# Map + Dynamic type = hmm...
map = {}
map[True] = False
map['Jim'] = 3
map[-1] = map
print(map) # {True: False, 'Jim': 3, -1: {...}}
```

Lua's *only* data structure is a map (table)

- why?
- how?
  - what about like...arrays?

```
names = {}
names[1] = "Bulbasaur" -- okay fine, Lua has strings too
names[2] = "Ivysaur"
print(names[1]) -- Bulbasaur
```

11 – Data Structures

Tables in Lua are not a data structure; they are *the* data structure. All structures that other languages offer---arrays, records, lists, queues, sets---are represented with tables in Lua. More to the point, tables implement all these structures efficiently.

In traditional languages, such as C and Pascal, we implement most data structures with arrays and lists (where lists = records + pointers). Although we can implement arrays and lists using Lua tables (and sometimes we do that), tables are more powerful than arrays and lists; many algorithms are simplified to the point of triviality with the use of tables. For instance, you seldom write a search in Lua, because tables offer direct access to any type.

It takes a while to learn how to use tables efficiently. Here, we will show how you can implement typical data structures with tables and will provide some examples of their use. We will start with arrays and lists, not because we need them for the other structures, but because most programmers are already familiar with them. We have already seen the basics of this material in our chapters about the language, but I will repeat it here for completeness.

ANNOUNCEMENTS

today is **Prof. Bill Is Visiting LEC-02 Laptop Wednesday!**

WARMUP

what is `7 % 3` ?
- 🖍️ do it on paper
- ☕ do it in Java
- 🐍 do it in Python

(java means coffee)
(a python is a kind of snake)

what is `(-7) % 3` ?
- 🖍️ do it on paper
- ☕ do it in Java
- 🐍 do it in Python

TODAY hash maps

let's learn someting upsetting

7 % 3

```
  2 R 1
3 | 7
  - 6
  ---
    1
```

☕ -7 % 3

```
  -2 R -1
3 | -7
  -  -6
  ------
    -1
```

-7 - (-6)
= -7 + 6
= -1

🐍 -7 % 3

☕ Math.floorMod(-7, 3)

-1 + 3 = 2

%

- x % y returns the **remainder** of (x / y) and is read "x **modulo** y"
  - `int foo = 17 % 5; // 2 ("17 divided by 5 is 3 remainder 2")`
- 🤖 **% probably doesn't do what you expect for negative numbers; if x can be negative, use `Math.floorMod(x, y)` instead**
  - `5 % 3 // 2`
  - `4 % 3 // 1`
  - `3 % 3 // 0`
  - `2 % 3 // 2`
  - `1 % 3 // 1`
  - `0 % 3 // 0`
  - `-1 % 3 // -1 WAIT WHAT` 🤨
  - `Math.floorMod(-1, 3) // 2` 🙂👍

`Math.floorMod(x, y)`

- `Math.floorMod( 5, 3) // 2`
- `Math.floorMod( 4, 3) // 1`
- `Math.floorMod( 3, 3) // 0`
- `Math.floorMod( 2, 3) // 2`
- `Math.floorMod( 1, 3) // 1`
- `Math.floorMod( 0, 3) // 0`
- `Math.floorMod(-1, 3) // 2`
- `Math.floorMod(-2, 3) // 1`
- `Math.floorMod(-3, 3) // 0`
- `Math.floorMod(-4, 3) // 2`
- `Math.floorMod(-5, 3) // 1`
- `Math.floorMod(-6, 3) // 0`
- `Math.floorMod(-7, 3) // 2`

hash maps

hash map

- a **hash map** (**hash table**) is a great way to implement a map
  - hash map are implemented using an **array** and a **hash function**
  - hash maps are FAST
    - `put(key, value)` is $\mathcal{O}(1)$ 😍 FAST
    - `get(key)` is $\mathcal{O}(1)$ 😍 FAST
  - there are many different "flavors" of hash maps
    - separate chaining (today)
    - linear probing (friday)
    - and so much more!

hash function

hash function

- a **hash function** takes some data and returns an int called the data's **hash code** (**hash**, hash value)
  - this is called "**hashing**" the data
  - 🚨 a hash function MUST be **deterministic**
    - given the same data, a hash function MUST return the same value
- a **hash collision** (hash clash) happens when two different pieces of data have the hash code
  - a *good* hash function has very few collisions

```
/* String.java -- immutable character sequences; the object of string literals
   Copyright (C) 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005
   Free Software Foundation, Inc.


/**
 * Computes the hashcode for this String. This is done with int arithmetic,
 * where ** represents exponentiation, by this formula:<br>
 * <code>s[0]*31**(n-1) + s[1]*31**(n-2) + ... + s[n-1]</code>.
 *
 * @return hashcode value of this String
 */
public int hashCode()
{
  if (cachedHashCode != 0)
    return cachedHashCode;

  // Compute the hash code using a local variable to be reentrant.
  int hashCode = 0;
  int limit = count + offset;
  for (int i = offset; i < limit; i++)
    hashCode = hashCode * 31 + value[i];
  return cachedHashCode = hashCode;
}
```

hash function

- all Java objects have a (fine) built-in hash function called `hashCode()`
  - `hashCode()` can be negative!

```
System.out.println(  "Hans".hashCode());          //     2241694
System.out.println(  "Gary".hashCode());          //     2212033
System.out.println("Kahoot".hashCode());          // -2054990942
System.out.println("Kahoot".hashCode());          // -2054990942
```

hash function

- often, you want to turn a hash code into an index
  - ✨ `Math.floorMod(object.hashCode(), array.length)`

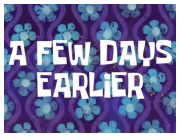
```
System.out.println(Math.floorMod(  "Hans".hashCode(), 10)); // 4
System.out.println(Math.floorMod(  "Gary".hashCode(), 10)); // 3
System.out.println(Math.floorMod("Kahoot".hashCode(), 10)); // 8
System.out.println(Math.floorMod("Kahoot".hashCode(), 10)); // 8
```

separate chaining

separate chaining

- **separate chaining** is one way of implementing a hash map
  - key-value pairs are stored in an array of **buckets**
    - one good choice of bucket is an array list
      - `ArrayList<KeyValuePair> bucket;`
    - in practice, you don't have to use an array list; you can use linked lists, binary search trees, or some mixture of the two (this is actually what Java does)

previously, on *Hans the Parrot*

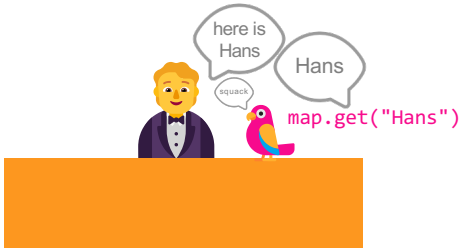


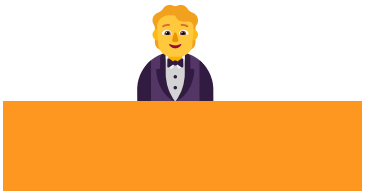
you have a three week long road trip planned to Bennington, but your prized parrot Hans is deathly afraid of driving

it's time to **put** Hans in the bird kennel

you return invigorated from a lovely trek to Bennington

it's time to **get** Hans from the bird kennel



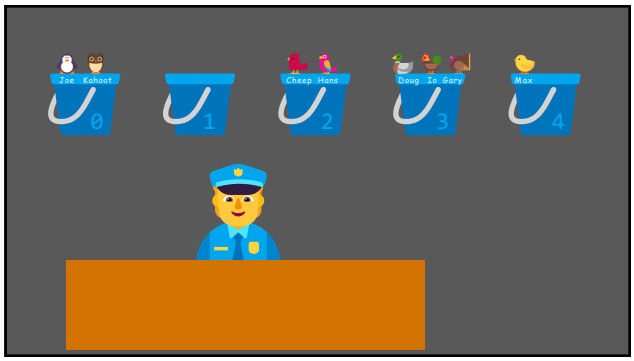


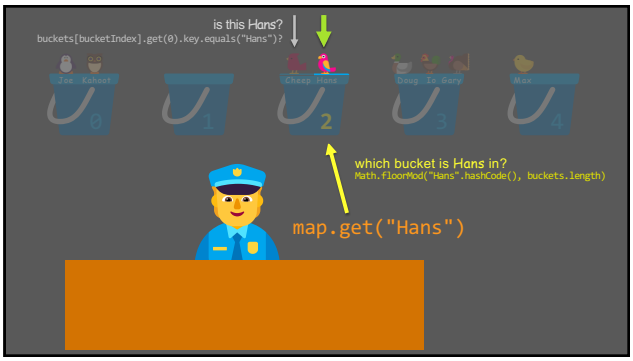
now, let us imagine again...

in a thrilling twist, it turns out that the Bennington trip was secretly cover for Agent Hans's corporate espionage of the bird kennel

(rumor had it they were keeping the birds in *buckets* 😳)

it's time to **get** Hans back to the hideout






```
//                Bucket[] buckets
ArrayList<KeyValuePair>[] buckets;
```

`ArrayList<KeyValuePair>[] buckets;`

- to **construct** the hash map
  - create a new buckets array
  - iterate through the array and create a new (empty) bucket in each slot

`ArrayList<KeyValuePair>[] buckets;`

- to **put** a key-value pair into the hash map
  - `// NOTE: hashCode() can be negative`
    `bucketIndex = Math.floorMod(key.hashCode(), buckets.length)`
  - `// update value if key in the map`
    iterate through that bucket
    - **if** you find a key-value pair with matching key...
      - update its value
      - `return;`
  - **if** you did NOT find any key-value pair with matching key
    - add the key-value pair to the bucket

`ArrayList<KeyValuePair>[] buckets;`

- to **get** a key's value from the hash map
  - `// NOTE: hashCode() can be negative`
    `bucketIndex = Math.floorMod(key.hashCode(), buckets.length)`
  - `// return value if key in the map`
    iterate through that bucket
    - **if** you find a key-value pair with matching key...
      - `return` its value
  - **if** you did NOT find any key-value pair with matching key
    - `return null;`

ANNOUNCEMENTS

today is an important **Colloquium!** 2:35 in Wege (TCL 123)
also juggling club after colloquium :)
today is **Kahoot Friday!** today is also No Blazer Friday :(

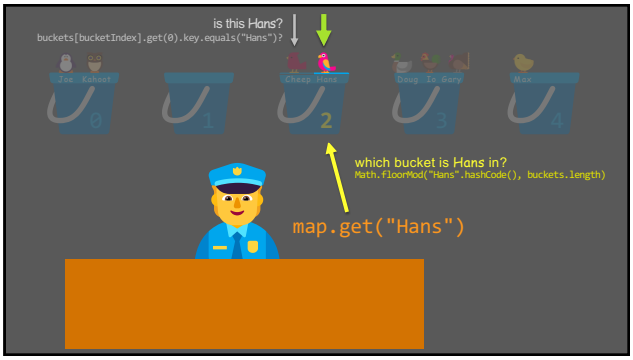
WARMUP

last class, i said that hash maps have O(1) get
why is this sus? 😳 list many reasons.

TODAY more hash maps

record LEC-01

what happens when we call get(key)?



step 1: hashing the key

(in our specific case, where key is a `String` and we use `key.hashCode()`)

```
/**
 * Computes the hashcode for this String. This is done with int arithmetic,
 * where ** represents exponentiation, by this formula:<br>
 * <code>s[0]*31**(n-1) + s[1]*31**(n-2) + ... + s[n-1]</code>.
 *
 * @return hashcode value of this String
 */
public int hashCode()
{
  if (cachedHashCode != 0)
    return cachedHashCode;

  // Compute the hash code using a local variable to be reentrant.
  int hashCode = 0;
  int limit = count + offset;
  for (int i = offset; i < limit; i++)
    hashCode = hashCode * 31 + value[i];
  return cachedHashCode = hashCode;
}
```

hashing the key

(our specific case, where key is a `String` and we use `key.hashCode()`)

```
// hash the key (not including the Math.floorMod)
hash = 0
foreach character in key
    hash *= 31
    hash += character
return hash
```

- 😳 **this is O(numberOfCharactersInString)**
  - but, if we assume all strings are less than, say, 64 characters... (assume a constant time hash function)
    - O(1) 😍

getting (a reference to) the correct bucket

(our specific case, where `bucket` is an unsorted `ArrayList<KeyValuePair>`)

```
// get the bucket
int bucketIndex = Math.floorMod(hash, buckets.length);
ArrayList<KeyValuePair> bucket = buckets[bucketIndex];
```

- O(1) 😍

searching the bucket for the key

(our specific case, where `bucket` is an unsorted `ArrayList<KeyValuePair>`)

```
// find a key-value pair in the bucket with matching key
// (or conclude no such pair exists)
foreach pair in bucket
    if (key matches pair.key):
        return pair.value
return null
```

- 😳 **this is actually O(numberOfPairsInBucket)**
  - assume a lot of buckets...
  - assume a good hash function...
  - do a lot of math...
    - O(1) 😍

😳 also what about *resizing* the hash map?!

that has to be O(n) !

resizing is O(1) *amortized* runtime

😳 😳

REVIEW: amortized

- the **amortized runtime** is how long a function takes (on average) "in the long run"
  - for adding an element to the back of an array list...
    - $\frac{1+2+4+1+8+1+1+1+1+16+1+1+1+1+1+1+1+32+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+\cdots}{n}$
    - $\frac{O(1+\cdots+1)}{n} + \frac{O(1+2+4+8+16+\cdots n)}{n}$
    - ...
    - $\frac{O(n)}{n} + \frac{O(n)}{n}$
    - $O(1)$ 😍
- **we can make a similar argument for a hash map**

lesson: there is a lot of subltety hiding in the seemingly simple statement "hash maps are O(1)"

buuut, in an interview, still probably just say O(1) 😳 👍

Tutorial: hash map with separate chaining

(using array list buckets)

```
// Put (add, insert) a new key-value pair into the map.
// NOTE: Can also be used to update a key's value.
// NOTE: be sure to compare strings using stringA.equals(stringB),
//       NOT the == operator
void put(String key, Integer value) {
    int bucketIndex = Math.floorMod(key.hashCode(), buckets.length);
    ArrayList<KeyValuePair> bucket = buckets[bucketIndex];
    for (int pairIndex = 0; pairIndex < bucket.size(); ++pairIndex) {
        KeyValuePair pair = bucket.get(pairIndex);
        if (key.equals(pair.key)) {
            pair.value = value;
            return;
        }
    }
    bucket.add(new KeyValuePair(key, value));
}

// Get (lookup) a key's value in the map.
// NOTE: Return null if map doesn't contain key.
Integer get(String key) {
    int bucketIndex = Math.floorMod(key.hashCode(), buckets.length);
    ArrayList<KeyValuePair> bucket = buckets[bucketIndex];
    for (int pairIndex = 0; pairIndex < bucket.size(); ++pairIndex) {
        KeyValuePair pair = bucket.get(pairIndex);
        if (key.equals(pair.key)) {
            return pair.value;
        }
    }
    return null;
}
```

open addressing

a hash map with
separate chaining used an
array of buckets of key-value pairs

`put(key,value)`
hash `key` to find `bucketIndex`
add `(key, value)` to `buckets[bucketIndex]`

a hash map with
open addressing just uses an
array of key-value pairs (slots)

`put(key, value)`
hash `key` to find `slotIndex`
if `slots[slotIndex]` is full, try `slots[slotIndex + 1]`
if `slots[slotIndex + 1]` is full, try `slots[slotIndex + 2]`
if `slots[slotIndex + 2]` is full, try `slots[slotIndex + 3]`
...
put `(key,value)` in first empty slot

`get(key)` is interesting
hash `key` to find `slotIndex`

is `key` in `slots[slotIndex]`?
  if yes, `return slot.value;`
is `key` in `slots[slotIndex + 1]`?
  if yes, `return slot.value;`
is `key` in `slots[slotIndex + 2]`?
  ...

give up if you hit a `null` (empty) slot
map does NOT contain `key` → `return null;`

open addressing -- deleting an entry

- **deletion** must be handled (very) carefully!
  - **BAD broken approach:** delete an entry by making it **null**

array: [..., ???, notNull, notNull, **deleteMe**, notNull, notNull, ???, ...]
array: [..., ???, notNull, notNull, **null**, notNull, notNull, ???, ...]

  - but later...what if we call **`get`** and **null** is in the middle?

array: [..., ???, **initialCollision**, notNull, **null**, notNull, **notNullKeyValuePairWeAreActuallyLookingFor**, ???, ...]

  - then we return null instead of **notNullKeyValuePairWeAreActuallyLookingFor**.value!

- **lazy solution:** mark newly empty slot; **`slot.usedToBeFull = true;`**
- **good solution: swap the rightmost contiguous notNull entry** into newly empty slot

array: [..., ???, notNull, notNull, **deleteMe**, notNull, **notNull**, ???, ...]
array: [..., ???, notNull, notNull, **notNull**, notNull, **null**, ???, ...]

note: these slides described a specific kind of open addressing called **linear probing**

open addressing is a strategy for **resolving collisions**
linear probing is just one way of doing open addressing

would you fancy a Kahoot?