Intro to Project 2: Hashing
Spring, 2021
Version 2
Inheritance at work in JDK

- HashMap extends AbstractMap, and so do six other JDK classes, so AbstractMap is the base class, the superclass.
- AbstractMap has lots of code that all these classes use.
- It offloads code from seven JDK classes, making them smaller and easier to support, and reduces code duplication.
- Note that the code in AbstractMap is not efficient for get and put: to do get(x), it looks through the entrySet for x.
- So many (all?) of these seven classes override get and put.
UML for the JDK Map family

From https://thenafi36.wordpress.com/2014/09/29/the-efficient-data-structure-for-unique-key-value-entries/
We’ll work with the actual Java 8 HashMap code, with the TreeNode support removed to simplify it.

That TreeNode support: It’s there to cover the case of using a bad hash function. A bucket that has way too many entries has its collision list converted to a tree structure. This code has been deleted.

Also, many methods that AbstractMap can handle have been removed to further simplify the code, to HashMap1, provided in the setup project.

This greatly simplified HashMap1 should be readable. It uses a separate-chaining hash table, like SeparateChainingHashST of S&W Sec. 3.4 (code pg. 465).
AbstractMap looms...

- JDK HashMap is a subclass of its AbstractMap.
- When you are working on HashMap, you need to be aware of how AbstractMap is there, hovering over your code.
- If you remove put(), perhaps by accident, the code still compiles and runs, because the runtime system simply looks for put in AbstractMap, the superclass, and there it is.
- If you accidentally break entrySet or something it depends on (those iterators near the end), lots of things stop working, because AbstractMap depends on entrySet.
- So keep good backups of your code!

3/30/2021
Pa2 steps

• First you’ll “seal up” the provided HashMap1. Lots of its data structures are not private. Of course you can’t make everything private: you need to be able to call its public API, covered in the JDK Javadoc. You’ll end up with HashMap2.

• Then you’ll refactor HashMap<K,V> to use a container class for the collision list instead of the unencapsulated linked list structure it has now, a list of Nodes.
  • Specifically, you will replace the list of Nodes with an ArrayList<Node>, one for each bucket. Then the details of holding the Nodes will be hidden inside the ArrayList. We’ll have little ArrayLists in buckets, handling collisions.
  • This has changed from version 1 of these slides: it turns out HashSet uses HashMap, so we would be in a circular dependency with that choice.

3/30/2021
Vs. using SSSTs for collision lists

- Remember SeparateChainingHashST covered in class 7 and coded on pg. 465, a class like HashMap. It has very neat code because it is using a SequentialSearchST (SSST for short) for each bucket’s collision list. Here is `get` from there:

  ```java
  get(Key key) {
      return (Value) st[hash(key)].get(key);
  }
  ```

- This says: hash the key, find the bucket, and it will have a collision list `st[bucket]`, which is a little SSST (i.e. Map) for the bucket. Just use the key to get the value from this SSST, and return it from the `get`.

- Wow, is that neat! Can we tame the mess in HashMap?

- Yes, to a great extent, though we will need to iterate down the encapsulated List. And we need to support resizing, not included in the code of pg. 465.
Separate Chaining Data Structure diagram from pg. 464

For pa2 picture: replace the SSSTs with little ArrayLists, one for each bucket.
A worry about performance

- But there’s a worry when you make code beautiful, esp. if it was written by real experts.
- Is the code ugly because it will run faster that way?
- We can draw the collision list structures in both cases and see there’s at least one additional reference at the start of the ArrayList instead of the unencapsulated list of Nodes.
- Let’s find out if and how much the performance changes. Be suspicious if your code runs faster than the original HashMap!
Understanding code in HashMap

- When you read HashMap's get and put, you will see the following mysterious code:
  \[ \text{tab}[i=(n-1) \& \text{hash}] \]
  - n is the size of the hash table
  - tab is the hash table itself
  - hash is a 32-bit hashed value based on the key’s hashCode.
- It is important to realize that n here is always an exact power of 2, by the design of this implementation
Understanding
\[ \text{tab}[i=(n-1) \& \text{hash}] \]

- \(n\) here is always an exact power of 2, so for example, \(n = 2^4 = 16 = \text{binary 10000} = \text{size (or capacity) of the hash table (initial value)}\).
- Then \(n-1\) is the binary number just below this binary 10000, so is 1111 (4 binary 1s).
- This forms a "mask for 4 bits\(^*\), so when the bitwise AND is computed by \(i=(n-1) \& \text{hash}\), the 4 lowest bits of \text{hash} are extracted from \text{hash} and put in \(i\), along with 0s in all higher bits, for \(i\)’s value.
- Ex: if \text{hash} is ...1011001001101 and \((n-1)\) is ...0001111, \(\text{(n-1)} \& \text{hash} = \ldots00000001101 = 8+4+1 = 13\), a good bucket number.
- In all cases, \(i \leq 15\), and \(i\) is a good bucket number for the hash table of 16 buckets. Then \text{tab}[i] is the reference to the collision list for the bucket.

\(^*\) See https://stackoverflow.com/questions/10493411/what-is-bit-masking
Understanding $\text{tab}[i=(n-1) \& \text{hash}]$ in larger hash table case (2048 buckets)

- n here is always an exact power of 2, so for example, $n = 2^{11} = 2048$ = size (or capacity) of the hash table.
- Then $n-1$ is the binary number just below this binary 1000000000000, so is 11111111111 (11 binary 1s).
- This forms a "mask for 11 bits"*, so when the bitwise AND is computed by $i=(n-1) \& \text{hash}$, the 11 lowest bits of $\text{hash}$ are extracted from $\text{hash}$ and put in $i$, along with 0s in all higher bits, for $i$’s value.
- So $i \leq 2047$, and $i$ is a good bucket number for the hash table of 2048 buckets. Then $\text{tab}[i]$ is the reference to the collision list for the bucket.

* See https://stackoverflow.com/questions/10493411/what-is-bit-masking
The project directories and packages

- Unzip the provided zip file and you should see the following directory structure:

```
├── bin
│   └── cs310
│       ├── client
│       └── util
└── src
    └── cs310
        ├── client
        └── util
```

Packages cs310.util and cs310.client are accommodated in subdirectories of subdirectories of src, and separately bin.

Note that although Java requires this directory structure based on the dots in the package names, there is no special meaning to the structure of the package name *inside* Java. “cs310.util” is just a different package from “cs310.client”.

3/30/2021
Building the project

• To compile, we need two javac commands, one for each source directory: Here we are allowing the .class files to live in the same directory as their .java file.

  cd src
  javac cs310/util/*.java
  javac cs310/client/*.java

• To run TestMap:

  java cs310.client.TestMap

• It doesn't need an input file. As provided, it tests HashMap1, so edit it to use HashMap3 as needed.
Working on HashMap: Using two HTs

• Since AbstractMap is so dependent on entrySet, we can’t yank out the lists of Nodes hanging off table right away without breaking everything.

• So we proceed by adding a new hash table, table1, while preserving the old hash table, table.

• We let get and put access table1, while entrySet, keyset, and values access the old table.

• Thus the set-related methods will return the empty set during our testing of get and put. No problem if we understand what’s going on.
Testing

- Two clients are provided, TestMap and TestMapPerf
- TestMap just executes a few gets and puts to help you debug get and put in your HashMap3 code
  - Edit it to use HashMap3 instead of HashMap1
- TestMapPerf loads up a lot of data, using the words file from UNIX spell, so it’s good for testing resize, and then finally, doing the performance testing. Also needs editing to use the right class.