CS310 – Advanced Data Structures and Algorithms

Class 13+: Expanded Intro to Project 2

Inheritance Example from 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 it’s 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/

Intro to pa2

- We’ll work with the actual Java 8 HashMap code, with the TreeNode support removed to simplify it.
- TreeNode support: To cover the case of using a bad hash function, a bucket that has 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.

AbstractMap looms…

- 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!

Pa2 steps

- First you’ll “seal up” 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 to use a container class for the collision list instead of the open-coded linked list structure it has now, a list of Nodes.
- Specifically, you will replace the list of Nodes with S&W’s SequentialSearchST, one for each bucket. Then the list will be hidden inside this “little map” for the bucket.
Using SSSTs for lists

- Look at SeparateChainingHashST 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, which is a little symbol table (i.e. Map) for the bucket. Just use the key to get the value from this st, and return it from the outer get.
- Wow, is that neat! Can we tame the mess in HashMap?
- Note that hash() in SeparateChainingHashST, and HashMap are different.

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 an additional reference at the start of list involved in using SequentialSearchSTs instead of the open-code 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:

  ```java
  tab[i=(n-1)&hash]
  ```

- n is the size of the hash table
- tab is the hash table itself
- hash is a 32-bit hashed value computed from the key.
- It is important to realize that n here is always an exact power of 2, by the design of this implementation.

   ![Image](https://via.placeholder.com/150)

Understanding tab[i=(n-1)&hash]

- 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 10000000000000000000000000000000, so is 11111111111 (11 binary 1s).
- This forms a "mask for 11 bits", so when the bitwise AND is computed by i=(n-1)&hash, the 11 lowest bits of hash are extracted from hash, along with 0s in all higher bits, for i’s value.
- So i <= 2047, and i is a good bucket number for the hash table of 2048 buckets. Then tab[i] is the reference to the collision list for the bucket.
- See https://stackoverflow.com/questions/10493411/what-is-a-bit-masking for more info.

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 names inside Java. “cs310.util” is just a different package from “cs310.client”.

Building the project

- To compile, we need two `javac` commands, one for each source directory:

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

- To run TestMap: cd .. To return to the base directory, and then

  ```
  cd bin
  java cs310.client.TestMap
  ```

- It doesn’t need an input file.
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, say `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.