REVIEW: INTERFACES, METHODS, and IMPLEMENTATIONS

Interface: List
Methods: size(), add(x), add(i, x), remove(i), get(i), set(i, x)
Implementations: ArrayList, LinkedList

Interface: Map
Methods: size(), get(k), put(k, v), keySet()
Implementations: TreeMap, HashMap

Interface: Deque (a.k.a. StackInt)
Methods: empty(), peek(), pop(), push(x)
Implementations: ArrayDeque, LinkedList

Interface: Queue
Methods: size(), offer(x), peek(), poll(), [add(x), element(), remove()]
Implementations: ArrayDeque, LinkedList, PriorityQueue (a.k.a. Heap)

Interface: Comparable
Methods: compareTo(that)

Interface: Comparator
Methods: compare(x, y)

Interface: Iterator
Methods: hasNext(), next()

Interface: Iterable
Methods: iterator()
Implementations: everything

Interface: Set (Like a Map without a value.)
Methods: add(x), contains(x), remove(x)
Implementations: TreeSet, HashSet

Make sure you know that running times of the methods for each implementation.

COMPETING WITH GOOGLE

Let's consider a simple description of the internet. The internet consists of *web pages* which are actually just files. There are many types of files, but the ones which interest us are written in HTML (HyperText Markup Language). Each web page is accessed by its URL (Uniform (or universal) Resource Locator), which looks like this:

    http://www.cs.miami.edu/~vjm/csc220/index.html

An HTML file has words (text) on it and URLs (links), written in a special way:

```
Here is some
<A HREF=http://download.oracle.com/javase/6/docs/api/>
Java Documentation</A>
```

This makes the words Java Documentation appear, but when you click on it, it goes to the web page with URL

    http://download.oracle.com/javase/6/docs/api/

By the way, if no file is mentioned, the default is index.html. If there is no index.html, it just lists the directory, like every time you go to one of my prog directories.

When we give some search words to Google, such as

    Victor Milenkovic Java

Google finds all web pages which contain those words and ranks them by significance. It's a lot more complicated now, but originally the significance was determined by the number of references (links) from other web pages on the internet.

The idea is that if a web page is "good", then people will "vote" for it by putting links to it in their web pages. If this were done honestly, then the original idea would work fine. Of course, people try to subvert this idea for fun or profit by creating lots of "dummy" web pages that link to a page they want people to go to. It's like stuffing the ballot box in the old days of voting.

INDEXING WEB PAGES

Obviously, Google doesn't do the search by going out on the internet when you make a search request. Instead, they gather up information on web pages ahead of time and store this information on their own servers. They have to store it in a way that facilitates rapid search. This is often called *indexing*. We now know enough techniques to have a shot at explaing how Google manages to organize their information in a way that allows searches in a fraction of a second. Since we don't really know how Google does it, we will talk about a new search company called Tinge (This Is Not Google Either) and discuss how they might do it.

To start with, *indexing* is not mysterious at all. Every time Tinge sees a new web page, it creates a *file* on its hard disks to store information about the page, such as its URL, date it was seen, and perhaps even a compressed copy.

Inside the computer, you address individual *bytes*. On a hard disk, the smallest unit that can be addressed is a *block*, usually 512 (2^9) bytes. A file and the web page it represents can be indexed by the address of its first block. However, disks are only terabytes in size. A 2T (2^41 byte) disk has 4 billion (2^32) blocks (32+9=41). But there are trillions of web pages out there. So Tinge is going to need at least 250 hard disks, probably many more. Still, that's pretty reasonable for a company.

The index includes the disk number followed by the block number on that disk. Six bytes is probably plenty. The first two bytes indicate the disk number and the remaining four are the block number on the 2T disk. In any case, an int is not enough since it is only four bytes. An 8-byte long is way more than enough. (Of course, when we switched from 16-bit integers to 32-bit integers, we thought *those* were way more than enough.)

So the index of a web page is a unique long (64-bit) integer. Since it corresponds to a disk and a block on a disk, we can get to the web page information file in one disk seek, about 1ms.

HAVE WE INDEXED A PAGE ALREADY?

Suppose we see a link, http://www.cs.miami.edu/˜vjm/csc220/index.html. Do we need to index it? Or is it indexed already? How can Tinge know?

Even if each info file contains the URL, we would have to scan multiple disks to find the file, if it exists. That will take a ridiculous amount of time.

We need a Map from the URL to the index of the web page. If the URL is not a key, then it has not been indexed yet. With trillions of web pages, this Map cannot fit in RAM. It will barely fit on one hard disk!

B-TREE MAP FROM URL TO INDEX

I have already explained that the b-tree you implemented would make a good external data structure if we switch from 2-3 to 32-63. Lookup is log32(n) disk seeks. For n=2^40 (1 trillion), log32(n)=8. 8ms is not so bad.

However, there are two problems with using URLs as keys. One is easy to fix. The other is harder.

NORMALIZING URLS

What’s wrong with

    http://www.cs.miami.edu/~vjm/csc220/index.html

? It’s like

    November 17, 2015

It goes from month to day then jumps back to year. And those Europeans shouldn’t feel so smug. They would say 17/11/2015, which is completely backwards, although at least it is consistent. But what if they include a time? Class is 11:15 on 17/11/2015? Or 17/11/2015 at 11:15? Inconsistent again. Or are we going to say class starts at 15:11?

So we will use the format

    edu.miami.cs.www/~vjm/csc220/index.html

We drop the http:// because we will only work with web pages.

This fixes the first problem. The following two strings are far apart:

    http://www.miami.edu
    http://www.cs.miami.edu

http://www.fiu.edu is closer to http://www.miami.edu than http://www.cs.miami.edu.

But the normalized versions are close:

    edu.miami.www
    edu.miami.cs.www

AVOIDING LONG DIRECTORY/FILE NAMES

The natural way to store a b-tree node on disk is as a *directory*. A directory maps a name to a subdirectory or a file. Files act as leaves. This is exactly what we need to implement a b-tree. Unfortunately, the URL key can be rather long, but Windows does not allow file names longer than 256 characters. The web page for my course has a 40-character URL, and its pretty simple. The longest URL in the csc220 page is

    edu.miami.edu.www/~vjm/csc220/prog02/doc/prog02/class-use/DirectoryEntry.html

which is almost 80, and I’m not even trying. A big web site will easily go past 256 characters.

COMPRESSED TRIE

In a *TRIE*, each subdirectory in a directory starts with a different letter. The elements of a subdirectory all start with that letter, so it is left off.

In a *compressed TRIE*, if the elements share more than just the first letter, the name of the subdirectory is the common prefix, and they leave that off.

See DirectoryTrie.png. Directory "j" contains "ack...html" and "ill...html".
(Because "." is a reserved directory name, we have to use "..." in place of ".".)

INDEXING PAGES AGAIN

For our implementation, we will use a PageFile class to represent the page file. It contains its index, URL, and reference count (links from other pages).

A HardDisk class will map a Long page index to its PageFile.
A PageTrie class will map a URL to its page index.

INDEXING WORDS

Each word also needs its own file. In order to answer a query like "Victor Milenkovic Java", we need to know the web pages which contain Victor, those which contain Milenkovic, and those which contain Java. (Then we will take the intersection of these three lists.) So for each word, we need to know the list of web pages which have that word. For instance, we need to know the list of web pages which have Milenkovic on them.

What do we mean by a list of web pages? It could be a list of URLs or a list of web page (url) indices. The latter is much more compact. So for each word, we will have a list of the indices of web pages which contain it. Each word has a List<Long>.

So the file contains the list of page indices of pages that contain that word. The index of a word is the (disk number) and block address of the first block of that file.

A HardDisk class will map a Long word index to its word file (list of Long).
A WordTable class will map a word to its index.

There are only millions of words, so the WordTable can be in RAM. A hash table (HashMap) is the best solution for WordTable.

GATHERING ALL WEB PAGES

Every year GoDaddy has a superbowl ad. For years, the commercials had nothing to do with the product, culminating in Bar Refaeli kissing Jesse Heiman in 2013. More recently, the ads finally gave some hint as to what GoDaddy sells.

GoDaddy is a domain name registrar. So when my wife wanted to create sleuthacademy.org, she paid GoDaddy some money and there is was. (She also has to host it somewhere.)

People register about one new domain PER SECOND, most with GoDaddy. So you can understand how they could afford to pay Jesse.

Each domain has multiple web pages, and Google wants to index them all. So every day, GoDaddy tells Google about the new domains and Google indexes all their web pages and everything they link to.

How can we do the same thing? We get a bunch of URLs. For each one, check to make sure we haven’t seen it before. If not, index it and put it in a Queue.

While the Queue is not empty, take out a URL. Using the Browser I provide you, get the List of words and URLs on that page. For each new URL, index it and put it in the queue. Index each new word. Add the index of the current page (the one we just dequeued) to the list for each word. Increment the reference count for every page that is referenced.

AVOID DUPLICATION

If the same URL appears multiple times on a page, does it get multiple votes? Suppose I put sleuthacademy.org a thousand times on my home page. Will my wife’s page increasing its reference count by 1000? We probably don’t want that. Why not?

The Browser class allows you to load a page, given its URL, and get a list of URLs and a list of words on that page. You will need to convert each URL to its index. Put each index into a Set<Long> to keep track of those you have increased the number of references.

We also need to add the index of the current page to the list of indices of each word on that page. But we don’t want to add it twice. However, we don’t need a Set to keep track of multiple words. We can just look at the end of the list for a word to see if the index is already there!

mary.txt shows the result of indexing a small web site.