Gigatron Data Structure Pseudo-Code

Yue Shu
EECS 293
Prof. Liberatore

> Reference: for this assignment, I have made some high-level discussions with my discussio classmate Yihe Guo. We collaborated conceptually on the use of group IP addresses.

Class-Level Design

> This part is simply for my reference for future implementation.

Class `IPEntry` :

```
global field MAXFIRSTBYTE <- 255
global field MAXSECONDBYTE <- 255
global field MAXTHIRDBYTE <- 255
global field MAXFOURTHBYTE <- 15

field firstByte <- the fist byte of the IPv4 address, ranging from 0 - MAXFIRSTBYTE
field secondByte <- the second byte of the IPv4 address, ranging from 0 - MAXSECONDBYTE
field thirdByte <- the third byte of the IPv4 address, ranging from 0 - MAXTHIRDBYTE
field fourthByte <- the fourth byte of the IPv4 address, ranging from 0 - MAXFOURTHBYTE
field count <- the count of hits corresponding to this IP address entry

constructor to initialize the local fields

builder of to invoke the constructor and perform error checks

getter to retreive the five local fields

increment to increment the count of the IPEntry

a global method isValid to validate whether the input IP address string is in valid IPv4 format
```

Notice that the range of the IP address object would be 0.0.0.0 - 255.255.255.15, i.e. the last part of the IPv4 address would be partitioned into the range of 0 - 15 to save the space. Therefore, there should be in total $256^3 * 16 = 268435456$ IP addresses in total.

Interface `GigatronDataStructure` :

```
method stub increment(x): increment the number of hits originating from IP address x

method stub count(x): returns the count of the hits from IP address x
```

The interface `GigatronDataStructure` is supposed to be made public and accessible to the users. It should be similar to the `TypeEntry` in our previous assignments. The interface is used to abstract the actual implementations as in the future we can provide other implementations to replace the one introduced below.

Class `ReducedIPEntryList` implements `GigatronDataStructure` :

```
field ipEntryList <- a list of IPEntry, should be unchangeable once initialized

new constructor as defined in the next section

override increment(x) as defined in the next section

override count(x) as defined in the next section
```

The `ReducedIPEntryList` is the actual implementation of `GigatronDataStructure` , something similar to the `BasicType` or `CompoundTypeEntry` in our previous assignments. In general the data structure is an abstraction of a array list, but with reduced size since the total size of IP addresses has been reduced by 16, as well as reduced space overhead since the data structure is no longer a hash table. Further details of the routines will be introduced in the section below.

Routine-Level Design

Algorithm `new` of `ReducedIPEntryList` :

```
Input: nothing
Output: a new instance of GigatronDataStructure in which all IP addresses start with no hits

ipEntryList <- an empty list field to store IPEntry

initialize the size of ipEntryList to be IPEntry.MAXFIRSTBYTE * IPEntry.MAXSECONDBYTE
                                                              * IPEntry.MAXTHIRDBYTE
                                                              * IPEntry.MAXFOURTHBYTE

for each integer value firstByte in the range of IPEntry.MAXFIRSTBYTE
    for each integer value secondByte in the range of IPEntry.MAXSECONDBYTE
        for each integer value thirdByte in the range of IPEntry.MAXTHIRDBYTE
            for each integer value fourthByte in the range of IPEntry.MAXFOURTHBYTE
                ipEntryList.add(new IPEntry(firstByte, secondByte, thirdByte, fourthByte, 0))
            end for
        end for
    end for
end for
```

Algorithm `increment(x)` of `ReducedIPEntryList` :

```
Input: an IP address x, should be a string value in the IPv4 format
Output: nothing

if x is a null input
    throw an appropriate exception
end if

if x is not a valid IPv4 format input
    throw an appropriate exception
end if

ipEntry <- null
index <- 0

firstByte <- parsed first byte of x
secondByte <- parsed second byte of x
thirdByte <- parsed third byte of x
fourthByte <- (parsed fourth byte of x) / 16

index <- firstByte * IPEntry.MAXSECONDBYTE * IPEntry.MAXTHIRDBYTE * IPEntry.MAXFOURTHBYTE
         + secondByte * IPEntry.MAXTHIRDBYTE * IPEntry.MAXFOURTHBYTE +
         + thirdByte * IPEntry.MAXFOURTHBYTE
         + fourthByte
ipEntry <- the {index}th element in the ipEntryList

ipEntry.increment()
```

Algorithm `count(x)` of `ReducedIPEntryList` :

```
Input: an IP address x, should be a string value in the IPv4 format
Output: a count of the hits from IP address x

if x is a null input
    throw an appropriate exception
end if

if x is not a valid IPv4 format input
    throw an appropriate exception
end if

ipEntry <- null
index <- 0

firstByte <- parsed first byte of x
secondByte <- parsed second byte of x
thirdByte <- parsed third byte of x
fourthByte <- (parsed fourth byte of x) / 16

index <- firstByte * IPEntry.MAXSECONDBYTE * IPEntry.MAXTHIRDBYTE * IPEntry.MAXFOURTHBYTE
         + secondByte * IPEntry.MAXTHIRDBYTE * IPEntry.MAXFOURTHBYTE
         + thirdByte * IPEntry.MAXFOURTHBYTE
         + fourthByte
ipEntry <- the {index}th element in the ipEntryList

return ipEntry.count() / 16
```

Usage Examples

Suppose in we are to run the program in Java language environment. We can instantiate a new Gigatron data structure and invoke the routines as follows

```java
GigatronDataStructure gigatronDataStructure = new ReducedIPEntryList();

gigatronDataStructure.increment("129.0.0.1");
gigatronDataStructure.count("129.0.0.1");
```

Time and Space Complexity

In general my implementation is an abstraction of an 1-D array with entries corresponding to the IP address. The size of the 1-D array would be the total size of the IP addresses in the field, which in my case would be $256^3 * 16 = 268435456$. Therefore, to calculate the index of each IP address, the general formular to consider is

```
firstByte * IPEntry.MAXSECONDBYTE * IPEntry.MAXTHIRDBYTE * IPEntry.MAXFOURTHBYTE + secondByte * IPEntry.MAXTHIRDBYTE * IPEntry.MAXFOUR
```
.

By using the index calculated as above, we could achieve both `increment(x)` and `count(x)` in constant time $O(1)$, and store the entire data in reduced linear space $O(n/16)$. In the future, when we need to access the most hit IP addresses, we can just use a list to keep track of the IPEntries with hits we are interested in, with constant extra space.

Justification

The algorithm involves nothing complex enough that requires very specific justification. Generally, we trade-off the accuracy of the algorithm by a fraction of 16 to save more space complexity. In other words, we combine every 16 of the IP addresses together as one single IPEntry.

As a result, every time the count corresponding to an IP address is to be incremented, the IPEntry corresponding to this specific IP address along with 15 others is incremented. On the other hand, each time we need to retreive the count of one certain IP address, the retrieved value would actually be the count of the corresponding IPEntry factored by 16, which could slightly affect the accuracy of the result. However, since we are generally interested in indentifying whether such automatic bots or search engines exist, it is less important for us to get an exact number than to detect a number large enough. Even though the actual value is split up by 16 to the neighbouring IP addresses, an entity with count greater than a threshond we set could generally mean something specific to us.