COM S 228, Fall 2020
Programming Project 5
Generating a Perfect Hash Table Implementation

Due 11:59 pm Friday November 13

No-penalty grace period until 11:59 pm Friday November 20

Problem Overview

Graphs are the most general of the common data structures; consider that a scalar object is a degenerate list, which is a degenerate tree, which is itself a degenerate graph, or in other words, every data structure we've discussed this semester can be represented as a graph. Due to their extreme generality, it's rare that you'll use a generic "graph" data structure. The limited set of things you might do with a list make two generic list implementations—an array based- and a linked-structure—good enough for most implementations, and it's only if you are doing, e.g., low-level system programming that you might have a good reason to write a custom, one-off list. But there is so much variation in graphs that it is perhaps impossible to write a good and useful generic graph implementation.

Hash tables provide a means to get constant-time look-up performance on a collection. Usually, the data comes to us at runtime, often in an on-line fashion. In this case, we take what foreknowledge we can to inform best practices in hash table construction, choosing a hash function, etc., to build a data structure with as close to optimal performance as possible; however, if the keys in the table are known beforehand, it's always possible to construct a *perfect hash table*, a table in which every key maps to a unique slot. There are many algorithms to construct such a table and the associated hash function; almost all of them are implemented using graphs.

For this project, you will be implementing a perfect hash table construction algorithm[1]. This algorithm generates a pair of tables of random numbers. Keys are hashed with the values in each these tables, giving two integers (one for each table). These integers are then taken as node indices in a graph, with an edge between them. So long as the resultant graph—after hashing all of the keys—contains no cycles, the algorithm is ready to generate the hash table as a function of the tables (the T tables) used in the graph generation and a function $g()$ on those tables.

Algorithm Example

NOTE: This algorithm looks pretty intimidating at first glance. With some careful reading and working-through of examples, you should find it's actually fairly straightforward. That said, you don't necessarily have to understand it at all! If you find yourself getting anxious about all this "scary-looking stuff", jump down to the *Requirements* sections, read that, then come back here with a more relaxed mindset. It is very common for professional programmers to be required to implement solutions to problems that they don't actually understand, so if you find yourself doing that, here or elsewhere, please know that it is normal. This is not to advocate for ignorance, however–it's always better to understand something than not to understand it–only to recognize that sometimes the cost-benefit of understanding is too high!

The example below was generated by hand, and uses an optimization that would be very difficult to implement in program control. We note that our original keys (the number names from "one" to "ten")

[1]The algorithm is presented in: Zbigniew J. Czech, George Havas, and Bohdan S. Majewski, "An optimal algorithm for generating minimal perfect has functions", *Information Processing Letters*, 43(5):257-264, October 1992.

are unambiguously differentiable by the first two letters. Thus, we can simplify the presentation by using only the first two letters of the keys. In your Java program, you will always use the entirety of every word; however, the method that hashes the words is already implemented for you, so you don't need to concern yourself with it unless you are endeavoring to fully understand the implementation.

Our keys and the "sub-keys" that we're using in this example follow:

Key	Minimum Unique Sub-key
one	on
two	tw
three	th
four	fo
five	fi
six	si
seven	se
eight	ei
nine	ni
ten	te

A modulus must be chosen, giving a maximum number in our tables, and a maximum number of nodes in the resultant graph. The paper authors choose a value that is one more than twice the number of keys, so we do too. This choice has implications on the runtime of the algorithm (smaller moduli increase the probability of cycles in the resultant graph, forcing a restart of the algorithm). If the modulus is too small, the algorithm will enter an infinite loop.

Next, we populate our tables, `T1` and `T2`, with random numbers less than the modulus. Each of the two tables will have the same number of rows as the length l of the sub-key. The i^{th} row, $0 <= i <= l - 1$, defines a separate mapping from letters at the i^{th} position (increasing from left to right) within the sub-key to random numbers.

	e	f	h	i	n	o	s	t	w
T1	8	15	10	2	10	19	13	0	13
	3	5	0	5	5	6	19	5	6

	e	f	h	i	n	o	s	t	w
T2	20	2	4	19	11	15	3	8	11
	16	4	2	6	19	10	18	4	18

These tables define a function that we will use to transform our keys into nodes in a graph. For each key, we apply the function twice, once each for `T1` and `T2`. Index the tables by the letter position and the letter value, add the numbers found in the tables, and return a result modulo the modulus[2]. We get two values per key, one for `T1`, and one for `T2`. These values are taken as node IDs in our graph, and an edge runs between them corresponding with the index of the key in the input vector, so "one" corresponds with index zero, "two" with index one, etc.

[2]If you look carefully at the hash function defined in the project source template, you will notice that we always use tables of size 4×64. We're doing things slightly differently there, to reduce code complexity. Instead of indexing with letter position, we index with letter position mod 4, and instead of letter value, we use letter value mod 64. The example in this document uses a simpler (and better) approach that is easy to do when hand tuning, but difficult to automate. The fundamental structure of the algorithm is unchanged in either case.

Applying our table to the first key, "one", and recall that we are using only the first two letters in this example (and that we always use the entirety of every word in our program code), we have `T1(0, o) == 19` and `T1(1, n) == 5`. $19 + 5 \equiv 3 \mod 21$, so one end of our "one" edge is node 3. The other end is given by `T2(0, o) == 15` and `T2(1, n) == 19`, and $15 + 19 \equiv 13 \mod 21$. So the "one" edge runs from node 3 to node 13.

The following table gives the calculation for the entire input set:

key	`T1(0,`*letter*`)`	`T1(1,`*letter*`)`	sum $\mod 21$	`T2(0,`*letter*`)`	`T2(1,`*letter*`)`	sum $\mod 21$
one	19	5	3	15	19	13
two	0	6	6	8	18	5
three	0	0	0	8	2	10
four	15	6	0	2	10	12
five	15	5	20	2	6	8
six	13	5	18	3	6	9
seven	13	3	16	3	16	19
eight	8	5	13	20	6	5
nine	10	5	15	11	6	17
ten	0	3	3	8	16	3

The T1 sum column is also called $u(\text{key})$ and the T2 sum is $v(\text{key})$. We'll be referring to them by these names later in the algorithm.

Now our graph is built. The next step is to check if the graph is suitable to build our hash table. A graph is suitable if it contains no cycles. Cycle detection is usually done with a depth first search, but in this example, we'll inspect by eye; in program control we would call a method for this check, one that you will be implementing!

Look at the final row in the above table and note that the "ten" edge is a self loop; it both begins and ends on node 3. Therefore, we need to discard this graph and the associated tables. We generate a new set of random tables:

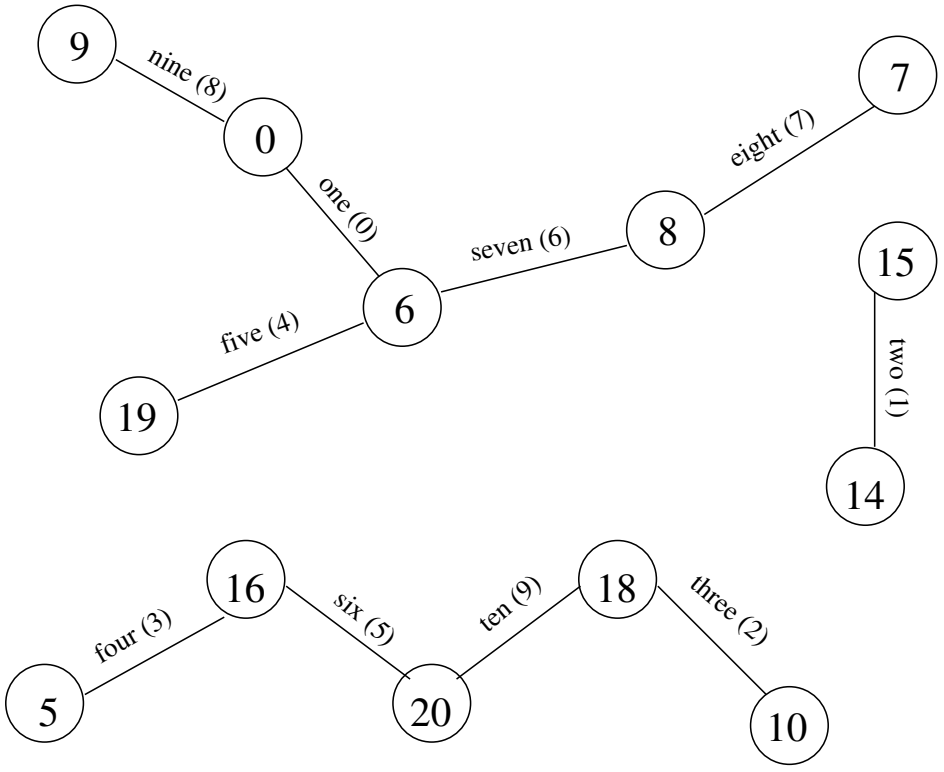
	e	f	h	i	n	o	s	t	w
T1	1	13	16	3	3	7	10	1	17
	19	11	17	6	20	3	0	17	14

	e	f	h	i	n	o	s	t	w
T2	7	5	6	13	20	8	19	10	16
	8	3	0	1	13	0	17	14	4

and calculate a new graph:

key	T1(0,*letter*)	T1(1,*letter*)	sum mod 21	T2(0,*letter*)	T2(1,*letter*)	sum mod 21
one	7	20	6	8	13	0
two	1	14	15	10	4	14
three	1	17	18	10	0	10
four	13	3	16	5	0	5
five	13	6	19	5	1	6
six	10	6	16	19	1	20
seven	10	19	8	19	8	6
eight	1	6	7	7	1	8
nine	3	6	9	20	1	0
ten	1	19	20	10	8	18

Applying our cycle-detecting eyeballs, we see that this graph has no loops:



The final step is to fill the array that defines the $g()$ function (as named in the paper; "g" does not stand for "graph"). $g()$ maps edges in our graph to indices in our original input array. We do this by traversing the connected components of the graph. We always start with the lowest-numbered, unvisited vertex. When traversing through a node where you have a choice of which direction to travel next, the choice does not matter.

We start with node zero and assign $g(0) = 0$. Then we traverse the neighbors of node 0; since neighbor order doesn't matter, let's do node 9 next. $g(9) = 8 - g(0) \mod 10 \equiv 8$, where the 8 in the subtraction comes from the edge value and 10 is the number of keys in the input vector. Going back to node 1 and heading the other direction, we have $g(6) = 0 - g(0) \mod 10 \equiv 0$; again, the 0 in the subtraction is the

edge value. Continuing to node 8, $g(8) = 6 - g(6) \mod 10 \equiv 6$. Node 7, $g(7) = 7 - g(8) \mod 10 \equiv 1$. And node 19, $g(19) = 4 - g(6) \mod 10 \equiv 4$.

This completes the first connected subgraph. Let's move to the next subgraph. The lowest-numbered, unvisited node is node 5. We assign $g(5) = 0$, and proceed as above. Node 16, $g(16) = 3 - g(5) \mod 10 \equiv 3$. Node 20, $g(20) = 5 - g(16) \mod 10 \equiv 2$. Node 18, $g(18) = 9 - g(20) \mod 10 \equiv 7$. And node 10 completes this subgraph, $g(10) = 2 - g(18) \mod 10 \equiv 5$.

There is one more subgraph, containing nodes 14 and 15. Start with the smaller, $g(14) = 0$, and $g(15) = 1 - g(14) \mod 10 \equiv 1$.

And here is g:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
g	0	0	0	0	0	0	0	1	6	8	5	0	0	0	0	1	3	0	7	4	2

Positions in this array that were not assigned in the last step do not matter. We choose to fill them in with zeros.

As stated above, $u(\text{key})$ is the first "sum $\mod 21$" in the final graph calculation table above, and $v(\text{key})$ is the second such column. Our hash function is given by:

$$\text{HASH(key)} = \left(g\left(u\left(\text{key}\right)\right) + g\left(v\left(\text{key}\right)\right)\right) \mod 10$$

thus:

$$\begin{aligned}\text{HASH(“seven”)} &= g(8) + g(6) \mod 10 \\ &= 6 + 0 \mod 10 \\ &= 6\end{aligned}$$

which is the seventh element, since arrays are zero indexed; and

$$\begin{aligned}\text{HASH(“three”)} &= g(18) + g(10) \mod 10 \\ &= 7 + 5 \mod 10 \\ &= 12 \mod 10 \\ &= 2\end{aligned}$$

A Second Example

Here we present one more example, this one from the paper. In this example, the keys are the English month names, with *sub-keys* being the second and third letters, as these are sufficient to unambiguously differentiate between the names. Again, remember that the concept of a "sub-key" is for illustration only! The Java implementation has no such notion.

Key	Minimum Unique Sub-key
January	an
February	eb
March	ar
April	pr
May	ay
June	un
July	ul
August	ug
September	ep
October	ct
November	ov
December	ec

The modulus is chosen to be 25, again, one more than twice the number of keys.

We populate our tables, `T1` and `T2`, with random numbers less than the modulus. This example omits values that will never be used to emphasize that those values don't matter. For instance, note that none of our sub-keys starts with a *b* or ends with an *o*:

	a	b	c	e	g	l	n	o	p	r	t	u	v	y
T1	11		1	13				12	17			1		
		9	21		13	5	19		20	1	0		3	12

	a	b	c	e	g	l	n	o	p	r	t	u	v	y
T2	5		2	21				24	8			12		
		9	23		5	2	7		12	17	2		11	8

Applying the tables to our keys:

key	`T1(0,`*letter*`)`	`T1(1,`*letter*`)`	sum mod 25	`T2(0,`*letter*`)`	`T2(1,`*letter*`)`	sum mod 25
January	11	19	5	5	7	12
February	13	9	22	12	9	5
March	11	1	12	5	17	22
April	17	1	18	8	17	0
May	11	12	23	5	8	13
June	1	19	20	12	7	19
July	1	5	6	12	2	14
August	1	13	14	12	5	17
September	13	20	8	21	12	8
October	1	0	1	21	2	23
November	12	3	15	24	11	10
December	13	21	9	21	23	19

September has a self loop, but there is a also a cycle between January, February, and March on nodes 5, 12, and 22, so this graph is unsuitable and the tables must be regenerated:

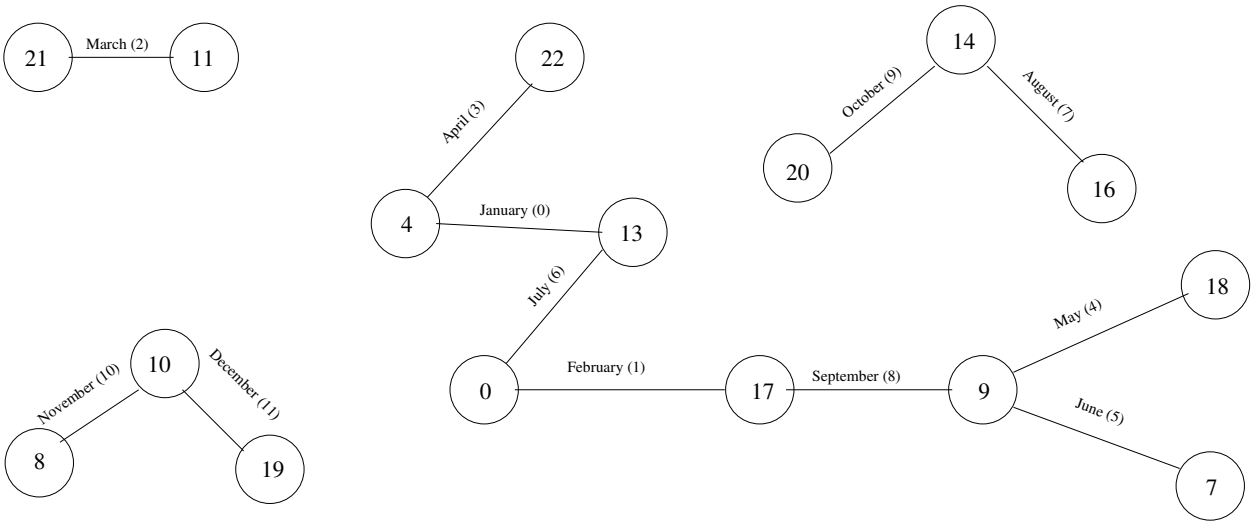
	a	b	c	e	g	l	n	o	p	r	t	u	v	y
T1	19		3	14				7	20			24		
		11	21		15	14	10		3	2	17		1	15

	a	b	c	e	g	l	n	o	p	r	t	u	v	y
T2	3		13	7				11	21			22		
		10	12		19	3	10		2	8	1		24	15

and a new graph must be calculated:

key	T1(0,*letter*)	T1(1,*letter*)	sum mod 25	T2(0,*letter*)	T2(1,*letter*)	sum mod 25
January	19	10	4	3	10	13
February	14	11	0	7	10	17
March	19	2	21	3	8	11
April	20	2	22	21	8	4
May	19	15	9	3	15	18
June	24	10	9	22	10	7
July	24	14	13	22	3	0
August	24	15	14	22	19	16
September	14	3	17	7	2	9
October	3	17	20	13	1	14
November	7	1	8	11	24	10
December	14	21	10	7	12	19

Here is its graphical representation, cycle free:



And the last step is to generate $g()$. As above, we start with node zero and assign $g(0) = 0$. Then we traverse the neighbors of node 0. We have twelve keys in this set, so we use mod 12, not mod 10 like we did in the first example. We'll start on the right, $g(17) = 1 - g(0) \mod 12 \equiv 1$. $g(9) = 8 - g(17) \mod 12 \equiv 7$. Going up, $g(18) = 4 - g(9) \mod 12 \equiv 9$. And down, $g(7) = 5 - g(9) \mod 12 \equiv 10$.

Now going left from node 0, $g(13) = 6 - g(0) \mod 12 \equiv 6$. $g(4) = 0 - g(13) \mod 12 \equiv 6$. And $g(22) = 3 - g(4) \mod 12 \equiv 9$.

There remain three more connected subgraphs. We leave these as an exercise. Remember to start at the smallest node value in each subgraph and assign it zero.

Here is the final g, with the unused values omitted:

index	0	4	7	8	9	10	11	13	14	16	17	18	19	20	21	22
g	0	6	10	0	7	10	0	6	0	7	1	9	1	9	2	9

Once again, here's our hash function (the only thing different from above is the modulus, which is the number of keys):

$$\text{HASH(key)} = (g(u(\text{key})) + g(v(\text{key}))) \mod 12$$

Let's hash some keys (Remember that we are using the second and third characters in this example!):

$$\begin{aligned}\text{HASH(“February”)} &= g(0) + g(17) \mod 12 \\ &= 0 + 1 \mod 12 \\ &= 1\end{aligned}$$

and

$$\begin{aligned}\text{HASH(“November”)} &= g(8) + g(10) \mod 12 \\ &= 0 + 10 \mod 12 \\ &= 10\end{aligned}$$

Requirements

The code template implements most of the "hairier" parts of this algorithm; indeed, you could complete this assignment without ever putting in the time or effort to understand the algorithm, and that's probably the optimal approach to take with respect to your total time, but it's sub-optimal with respect to your learning. You will be implementing most of the graph functionality, including cycle detection, and some code generation (much of code generation is done for you, partly so that you have example code to look at, and partly so that you have working functions in your generated output that you may use to test it.

Required classes, interfaces, and methods

All classes, interfaces, and methods appearing in the template are required and described therein.

`Edge.java`, `Vertex.java`, and `Visitable.java` all describe interfaces, and you will not be modifying these files (though you will be implementing methods of these interfaces).

Input

`main()` takes one required and two optional arguments. The required argument contains the keys, one per line. Note that if the file contains any repeated keys, the resultant graph will always have a cycle, so your program will go into an infinite loop.

The optional arguments to `main()` are a *prefix* and a *seed*. By default, the program will generate a class named `CHM92Hash` in a file named `CHM92Hash.java`. This makes in impossible to use multiple of these hash tables in the same program (without manually editing the files). The *prefix* makes the manual edits unnecessary; a prefix of `foo` results in the class `fooCHM92Hash` and file `fooCHM92Hash.java`. The *seed* parameter allows you to get reproducible behavior from the random number generator by forcing a particular seed. This is useful for debugging. You cannot specify a seed without using a prefix.

Classes

Templates for the following classes are provided and will require modification where noted in the source.

PerfectHashGenerator

This class contains the `main()` method. The class is fully implemented and should not be modified. The two `generate()` methods build graphs and run the code generator. `readWordFile()` reads the input. `mapping()` handles all of the algorithm steps through cycle testing.

Graph

This class implements the graph data structure. A `toString()` method is implemented for you so that you can easily print and inspect your graphs while you debug. You will be implementing methods to add edges to the graph, to mark nodes as visited or unvisited, and to test for cycles.

Specific to the algorithm in this class is the `fillGArray()` method, which traverses the final graph and calculates the $g()$ function, as described above. You will not have to modify this method.

CodeGenerator

Given `T1`, `T2`, $g()$, and the modulus, we can write a Java file that implements our hash table. The code generator also takes the key list in order to write it to the output source file, but this isn't strictly necessary for the hash table to function (see the comments in the output). You will complete the implementation of the `begin()`, `array()`, `table()`, and `end()` methods, all of which write code to the output file.

Testing your code

Once you are successfully writing code that compiles and that you believe to be correct, uncomment the generated `main()` method, compile, and run. It should produce no output. If it produces output, there is an error in your code.

Once you believe your solution is working, modify one or more of the keys in a generated `KEY_LIST` array, recompile, and run again. Your code should (probably[3]) print output about hashing errors, one for each key changed.

I can run the solution on `/usr/dict/words` (a word list that most `UNIX` and `UNIX`-like systems have) on my workstation, which contains 38619 keys. This takes less than 2 seconds to complete. When I then attempt to compile the output, I get:

[3]It's possible (roughly 1 in *number of keys* probability) that your change will result in a new key that maps to the same index, in which case you won't see an error message.

```
sheaffer@sheaffer-pc:~/228-f2020-assignment5/src$ javac CHM92Hash.java
CHM92Hash.java:8: error: code too large
  public static final String[] KEY_LIST = {
                               ^
CHM92Hash.java:1: error: too many constants
public class CHM92Hash {
       ^
2 errors
sheaffer@sheaffer-pc:~/228-f2020-assignment5/src$
```

This is the result of a hard limit that is coded into the JVM. There is no workaround; Google the error message for details. Some testing shows that the maximum input size in Java to allow compilation of the output is about 2500 words.

I wrote a C version of this algorithm which I've used to create, compile, and run perfect hash tables that contain over 4 million keys.

Submission

You are required to include, in your submission, the source code for each of the classes and interfaces in the code template, as well as any additional classes or methods you may have written to complete the assignment (you shouldn't need any). You need to write proper documentation with JavaDoc for each method in each class. Write your class so that its package name is `edu.iastate.cs228.hw5`. Your source files (`.java` files) will be placed in the directory `edu/iastate/cs228/hw5` (Linux) or `edu\iastate\cs228\hw5` (Windows), as defined in the template code. Be sure to put down your name after the `@author` tag in each class source file. Your zip file should be named `Firstname_Lastname_HW5.zip`.