CSC 225 SUMMER 2014
ALGORITHMS AND DATA STRUCTURES I
ASSIGNMENT 4 - PROGRAMMING
UNIVERSITY OF VICTORIA

1 Programming Assignment

Hash tables can be very useful in practice for efficiently searching large datasets. A well designed hash table implementation can achieve $O(1)$ expected case search time. However, a poorly designed hash table may have $O(n)$ search time, which is no better than searching an unsorted list (and may have higher overhead). Designing an effective hash table usually requires knowledge of both the type of data being stored and the expected distribution of that data.

This assignment will be an investigation of different hashing schemes for strings of characters, with the goal of finding a highly optimized hashing scheme for a specific type of string data. The dataset used for this assignment will be a collection of place names (cities, towns, etc.).

A string s can be considered as a sequence of characters $s = c_0c_1 \dots c_{k-1}$. For example, when $s =$ `"Victoria"`, $c_0 =$ `'V'`, $c_1 =$ `'i'` and so on. In Java, each character c_i is stored as type `char`, which is a numerical type. Therefore, it is easy to design a hash function for strings based on the numerical value of each character.

A simple hash function for a string $s = c_0c_1 \dots c_{k-1}$ with a table size t adds all of the characters together and takes the result modulo t:

$$h(s) = (c_0 + c_1 + \ldots + c_{k-1}) \bmod t.$$

While this simple scheme is easy to implement, the set of possible hash values is very small, since each character c_i is an 8-bit value, so the sum of all characters is at most 2^8k.

A more advanced hash function for $s = c_0c_1 \dots c_{k-1}$ uses the characters c_i as coefficients for a polynomial.

$$h(s) = (c_0 + c_1x + \ldots + c_{k-1}x^{k-1}) \bmod t.$$

The value x is a fixed positive integer. The simple hash function above corresponds to the case $x = 1$.

1.1 Templates and Testing Procedure

The assignment has six parts, which will be implemented as six separate programs. Six template files have been provided, all containing identical starting code. Each of the six tasks below should be completed in the respective template file (for example, the code for part 1 should be submitted in `HashTable1.java`). Each template contains an implementation of a hash table for strings, using a table size $t = 225225$. You are not permitted to change the table size. There are three hash table methods:

- `hash(s)`: Given a string argument s, return a hash value for s in the range $[0, t-1]$.
- `insert(s)`: Insert the value s into the hash table.
- `find(s)`: Search the hash table for the string s and return the table index at which it was found (or -1 if s was not found).

The exact specification of each method and specific implementation details can be found in the template files. You should read through the comments in the template files carefully before starting the assignment. As provided, the template uses the hash function

$$h(s) = 0$$

and a linear probing scheme to resolve collisions.

The `main()` function in the template contains testing code to perform a series of `insert` and `find` operations based on test data entered by hand or read from a file. Input data consists of a set of strings to insert, one per line, followed by the token `"###"`, followed by a set of strings to search for in the table. The following input file constructs a hash table containing the strings `"Victoria"`, `"Vancouver"` and `"Sidney"`, then searches for the strings `"Duncan"`, `"Nanaimo"` and `"Vancouver"`:

```
Victoria
Vancouver
Sidney
###
Duncan
Nanaimo
Vancouver
```

The hash table implementation in the template has been instrumented to measure the total number of probes during the insertion and search phases, as well as the maximum number of probes required to insert or search for a single value. You can use this information to assess the performance of the different hashing schemes used in this assignment. The output of the unmodified template `HashTable1.java` on the input data above is shown below (console input is shown in blue):

```
Enter a list of strings to store in the hash table, one per line.
To end the list, enter '###'.
Victoria
Vancouver
Sidney
###
Read 3 strings.
Enter a list of strings to search for in the hash table, one per line.
To end the list, enter '###'.
Duncan
Nanaimo
Vancouver
###
Read 3 strings.
Inserted 3 elements.
 Total Time (seconds): 0.00
 Total Probes: 6
 Max. Probes: 3
Searched for 3 items (1 found, 2 not found).
 Total Time (seconds): 0.00
 Total Probes: 10
 Max. Probes: 4
```

The probe statistics at the end of the input indicate that a total of 6 table accesses were needed to insert all 3 items into the table, with at most 3 probes being needed for any individual item. To search for all 3 of the provided values (of which two were not found), 10 probes were needed, with at most 4 needed for any element.

Part 1: Linear Hashing With Linear Probing (10 marks)

In the file `HashTable1.java`, modify the provided implementation to use the hash function

$$h(s) = (c_0 + c_1 + \ldots + c_{k-1}) \bmod t$$

on an input string $s = c_0 c_1 \ldots c_{k-1}$, with the table size $t = 225225$ and linear probing for collision resolution.

In Java, to retrieve the i^{th} character of a `String` s, use `s.charAt(i)`. The length k of s can be obtained with the call `s.length()`.

Part 2: Polynomial Hashing With Linear Probing (10 marks)

In the file `HashTable2.java`, modify the provided implementation to use the hash function

$$h(s) = (c_0 + 2^1 c_1 + 2^2 c_2 \ldots + 2^{k-1} c_{k-1}) \bmod t$$

on an input string $s = c_0 c_1 \ldots c_{k-1}$, with the table size $t = 225225$ and linear probing for collision resolution.

Part 3: Polynomial Hashing With Quadratic Probing (1) (10 marks)

In the file `HashTable3.java`, modify the provided implementation to use the hash function

$$h(s) = (c_0 + 2^1 c_1 + 2^2 c_2 \ldots + 2^{k-1} c_{k-1}) \bmod t$$

on an input string $s = c_0 c_1 \ldots c_{k-1}$, with the table size $t = 225225$ and quadratic probing for collision resolution.

To implement quadratic probing in Java, it is recommended that the `long` type be used instead of `int` during index calculations to avoid overflow.

Part 4: Polynomial Hashing With Quadratic Probing (2) (10 marks)

In the file `HashTable4.java`, modify the provided implementation to use the hash function

$$h(s) = (c_0 + 16^1 c_1 + 16^2 c_2 \ldots + 16^{k-1} c_{k-1}) \bmod t$$

on an input string $s = c_0 c_1 \ldots c_{k-1}$, with the table size $t = 225225$ and quadratic probing for collision resolution.

Part 5: Polynomial Double Hashing (10 marks)

In the file `HashTable5.java`, modify the provided implementation to use the hash function

$$h_1(s) = (c_0 + 2^1 c_1 + 2^2 c_2 \ldots + 2^{k-1} c_{k-1}) \bmod t$$

on an input string $s = c_0 c_1 \ldots c_{k-1}$, with the table size $t = 225225$ and a double hashing scheme with secondary hash function

$$h_2(s) = (1 + c_0 + 3^1 c_1 + 3^2 c_2 \ldots + 3^{k-1} c_{k-1}) \bmod t$$

for collision resolution. Recall that the sequence of indices probed by a double hashing scheme is

$$h_1(s), \quad h_1(s) + h_2(s), \quad h_1(s) + 2h_2(s), \quad h_1(s) + 3h_2(s), \quad \ldots$$

Part 6: Improved Hashing (10 marks + up to 5 bonus marks)

In the file `HashTable6.java`, you will implement a hashing scheme with better performance than any of the previous five schemes on the file `place_names_large.txt` (available on conneX; see the 'Test Datasets' section below). You may use any hashing and collision resolution methods you choose, including schemes you find online or in books. Any algorithms adapted from an external source must include a complete citation (including the address of the web page or name of the book).

You may not change the table size (you must use $t = 225225$) and you may not use any of Java's built-in hashing functionality. You are encouraged to try the same hashing algorithms that Java uses internally, but you must implement them yourself.

The number of marks given for this part will be based on the maximum number of probes required for a single element over the insertion and search phases on the file `place_names_large.txt`. Up to 5 bonus marks may be given for hashing schemes with exceptional performance.

Marks (/10)	**Max. Probes**	
	Search	Insertion
$0 - 3$	22	15
	- or -	
	23	14
$4 - 7$	22	14
$8 - 10$	14	15
	- or -	
	16	14
$11 - 12$	13	15
	- or -	
	15	13
$14 - 15$	12	15
	- or -	
	15	12

2 Test Datasets

Three datasets have been uploaded to the Data tab on conneX. All three insert a list of place names (cities, towns, geographical features, etc.) into the hash table. The place name data is

based on the freely available geographical data repository at `http://www.geonames.org`. The 'search' section of each dataset contains 1865 strings, based on the names of streets around Victoria. Some of these values will be present in the hash table (for example, 'Waterloo' is the name of a road in Victoria and a city in Ontario), but some searches will be unsuccessful. The table below gives the statistics for each dataset.

File	**Insertions**	**Searches**		
		Total	Present	Not Present
`place_names_small.txt`	22107	1865	362	1503
`place_names_med.txt`	43229	1865	527	1338
`place_names_large.txt`	122359	1865	757	1108

3 Evaluation Criteria

Each of the six parts will be marked out of 10 (with up to 5 bonus marks available for part 6), based on a combination of automated testing and human inspection. The only datasets that will be tested are the files `place_names_small.txt`, `place_names_med.txt` and `place_names_large.txt`. However, the data in the files may be reordered for marking.

To be properly tested, every submission must compile correctly as submitted, and must be based on the provided template. **If the submission for any part does not compile for any reason (even trivial mistakes like typos), or was not based on the template, that part will receive 0 out of 10**. The best way to make sure your submission is correct is to download it from conneX after submitting and test it. You are not permitted to revise your submission after the due date, and late submissions will not be accepted, so you should ensure that you have submitted the correct version of your code before the due date. conneX will allow you to change your submission before the due date if you notice a mistake. After submitting your assignment, conneX will automatically send you a confirmation email. If you do not receive such an email, your submission was not received. If you have problems with the submission process, send an email to the instructor **before** the due date.