CSC2001F 2019 - Assignment 3

Instructions

The goal of this assignment is to compare the insert/search cost (in terms of probes required) for a hashing scheme that uses one of **linear probing, quadratic probing** or **chaining**. You will be using the same data set and key field that you used for Assignments 1 and 2. Note that we do NOT count the initial hash index computation as a probe.

Dataset

The attached file represents a time series of power usage for a suburban dwelling. The base data set and additional information can found at

https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption

Load the file into LibreOffice/OpenOffice/MS-Excel to see what the data looks like. Note that the first line consists of headings and needs to be ignored by your code. Every subsequent line represents a single data item/record.

In your application, you MUST write your own code to read in the Comma Separated Value (CSV) file. You may not use a CSV library. Your data structure items must each store the following **3** values extracted from the CSV file:

- "Date/time"
- "Global_active_power" (which we will call "Power")
- "Voltage"

Programming Brief

Use the attached data file of 500 power usage records to build a hash table. You Java program needs to take parameters that allow you to set

- the table size
- the collision resolution scheme (quadratic probing or linear probing or chaining)
- the input data file (to be hashed)
- the **number** of keys, *K*, to be searched for in the hash table. These keys must be chosen as a random subset (no duplicates!) of the full set of keys/records used to build the table (see later). If this parameter is set to 0, then the table will be built but not searched (you should still report appropriate output in this case – see below). Hint: you can use the shuffle() function in Java Collections to randomly shuffle an array/list of elements.

The hashing insert/search key will be the “Date/time” string as before.

Insertion/construction

The program should output

1. the **load factor** after initial table construction,
2. the **total number of probes** required for all insertions.

Note that the load factor for linear/quadratic probing will be in [0,1] while chaining will typically have a load factor greater than 1 (see notes). A "probe" for chaining will be a link traversal in the linked list for that hash table cell. To be consistent with the other methods, assume that your initial hash index also includes moving to the first entry in your linked list. So the number of probes would be any *additional* link traversals required as you search for the target key in the list for that hash table cell. The list is NOT sorted. Always insert at the start/end of the list and assume no duplicate keys.

Key search

If a search is specified over *K* random keys, then the program should also output

1. the **total** number of probes generated over the *K* searches,
2. the **average number of probes** per search, i.e. (sum each key's probe count)/*K*
3. the **maximum number of probes** over the set of *K* searches, i.e. the number of probes in the longest probe sequence over all K keys.

You will also need to choose your own hash function – remember, "Date/time" is a string, unless you choose to convert it into some other form before hashing.

You may wish to use shell scripting to automate some of the testing – but this is not required.

Constraints

1) The hash table size must be a **prime number**; you can run a simple primality tester (if not your code, it needs to be appropriately referenced) to confirm and exit – with an appropriate message - if this is not the case. The tutors will test this.
2) When using quadratic probing for *insertion*, declare the probe a **failure** if the number of probes exceeds the table size. You can throw an exception to signal this, and must handle the exception in a sensible manner. This may happen in tables with very high load factor. For linear probing *insertion*, failure will only occur if the table is full – this should also cause an exception to be thrown. Insertion failure cannot occur in chaining since the lists will grow as required. Note that if we can find a probe sequence that inserts an element, we will always be able to successfully *search* for that element (the calculation mirrors the insertion probe sequence).
3) For quadratic probing, you must implement the more efficient incremental probe location update mentioned in the slides and text book. This avoids squaring the index value (an integer). You can do this with a left bit shift e.g. x<<1 for an integer x. Be sure to note this, and any other optimization you included, in your report.
4) You must write all the code for this assignment, with the exception of primality testing code. You may also reuse any I/O code you wrote for Assignments 1 and 2. You may use Java Lists or ArrayLists, but note that the lists must remain unsorted – this is an underlying assumption of the basic chaining method and will break the comparisons otherwise. You need to move through the list from start to end, searching for the target key.

Testing and Analysis

Test a sequence of **5** table sizes (all prime), starting from the prime number 653. You can choose the sequence, but the numbers need to be steadily growing in size and sample the range [653, 1009]. If the quadratic probe sequence fails on the starting prime (it should not), then chose another larger prime and start your experiments from that prime number instead (note: we want the same set of 5 primes for the linear probing, quadratic probing and chaining experiments).

For a list of primes, have a look at http://math.info/Arithmetic/Prime_Numbers_List/

Insertion performance: for each table size, insert all 500 records and record the *load factor* and *total probe count*, for *linear probing, quadratic probing and chaining.* Produce **1** graph showing total probe count vs load factor. Show all three sets of results (linear, quadratic, chaining) on the same graph. Use sensible colours/line patterns (with associated graph legend) to make it clear which data points/line etc belong to which method.

Search performance: for each table size (equivalent to load factor) and scheme (linear, quadratic, chaining), search for 400 (randomized) keys chosen from the set of all keys that you used to build the table. Recall that for *K* searches, your program will return the total number of probes over *K* searches, average number of probes across the *K* searches and the longest probe sequence across the *K* searches.

Produce **3** graphs: i) total probes vs load factor, ii) avg probes vs load factor and iii) longest probe vs load factor. Each graph should contain results for all three methods (linear, quadratic, chaining), with appropriate colour/line pattern to make it clear what each refers to. The graph should have a legend that makes this clear. This is standard in decent graphing tool.

In your report, you will need to interpret and comment on **all** these graphs.

Note: You can also manage everything **inside** your Java program (avoiding the need for multiples runs/scripting); *your report should note how these modifications change required command line parameters. Be sure to explain how you implemented testing to make it as efficient as possible*

Report

Write a report (of up to 6 pages of text; additional figures can add an extra 2 page at most – you only need 4 separate graphs) that includes the following (with appropriate section headings):

- What your OO design is: what classes you created and how they interact. You do not need to draw class diagrams.
- What the goal of the experiment is and how you executed the experiment.
- Present your performance graphs and discuss the results. Comment on the relative probe performance of the different approaches, and also consider the impact of load factor on the results. Comment specifically on all the metrics/measures you made for each approach – if there is a graph you need to discuss what it shows. Also make sure to provide captions for each graph and use consistent scales on the axes. Draw conclusions about the performance of the methods and anything else relevant that you observe.
- Include a statement of what you added in your application that constitutes creativity - how you went beyond the basic requirements of the assignment.

- Summary statistics from your use of git to demonstrate usage. Print out the first 10 lines and last 10 lines from "git log" , with line numbers added. You can use a Unix command such as:

```
git log | (ln=1; while read l; do echo $ln\: $l; ln=$((ln+1)); done) |
(head -10; echo ...; tail -10)
```

Dev requirements

As a software developer, you are required to make appropriate use of the following tools:

- **git**, for source code management
- **javadoc**, for documentation generation
- **make**, for automation of compilation and documentation generation

Submission requirements

Submit a .tar.gz compressed archive containing:

- Makefile
- src/
  - all source code
- bin/
  - all class files
- doc/
  - javadoc output
- report.pdf

Your report must be in PDF format. **Do not submit the git repository.**

Marking Guidelines

Your assignment will be marked by tutors, using the following marking guide.

Artefact	*Aspect*	*Mark*
Report	Appropriate design of OOP and data structures	5
Report	Experiment description	5
	Results graphs/data– required graphs produced	5
	Discussion of results - insertion	5
	Discussion of results - searching	5
	Creativity	5
	Git usage log	2
Code	Obeys specification & no obvious inefficiencies	5
	Program runs with test args and produces sensible output	5
Dev	Documentation - javadoc	5
	Makefile - compile, docs, clean targets	3