Project 2A: Hashtable Analysis

Instructions

Before you begin, please take a look at the resources linked from the assignment page in Canvas. First, there is an example analysis project on binary search for which we have supplied example responses. Looking at these should give you some idea of what we are looking for in your response. Second, as technical writing is one component of this assignment, we have links to some resources from the Mines Writing Center. If you find the writing aspects of this project challenging, please consider making an appointment for a consultation with the Writing Center. Finally, we have supplied sample code showing how you can collect and save experimental data into files for easier plotting.

For this assignment, please answer the bolded questions below. Your answers should primarily consist of text in complete sentences, along with tables, charts, diagrams, and equations as needed. Your answers should be concise but complete. Diagrams or charts must be clearly labeled and readable; use of a software plotting tool (Excel, matplotlib, etc.) is required. Equations must be properly typeset (e.g., using the equation editor in your word processing software or an equation environment in LaTeX).

The rubric for this project can be found at the end of this document.

Tasks

Overview

Now that you've implemented a hashtable, it's time to put it under a stress test. As we discovered in lecture, hashtables *can* have $O(1)$ time complexity for insert, contains, and remove *if* we design them correctly. The first part of this analysis project will collect evidence supporting this claim.

A. Data Collection and Plotting

In previous projects, we've measured performance by collecting process run times. You may have noticed that timings can be tricky: dependent on hardware, system load and input size. In this analysis we will measure performance by counting the actual number of object comparisons used in hashtable `.contains()` operations. We'll perform a series of experiments and collect random samples at various hashtable load factors, plot the data and explain the patterns we see with our knowledge of how hashtables are implemented.

If you have not done so already, git clone the 2A Hashtable repository for the course (instructions in the Canvas assignment page), and follow the instructions in its README.md to prepare for this project.

Add a counting method

Create a new method for your hashtable class template named `count_comparisons()` or similar. This method should start as a copy of your `contains()` method. Instead of returning a Boolean indicating whether the key argument is stored in the hashtable, your new method should return an integer giving the number of comparison operations performed in testing for containment. Note this is not simply the size of the chain—you must still loop over the

chain, counting comparisons, until you find the key or reach the end of the chain without finding the key.

Write code to run one experiment

The big coding component for this project will be writing an `experiment()` function. Eventually you will write code in `main.cpp` using this new function to run many experiments, but for now just have `main()` call `experiment()` once for testing purposes. The invocation will look like this:

```
experiment( 23432 );
```

The argument 23432 is just a placeholder for now; your later experiments will use many different values. Now let's code up `experiment()`! You can either put it in the same source file as your `main()` or create an independent source file for it. Your `experiment()` function will need to use the `bernoulli()` routine provided in `random220.h` and `random220.cpp`, so provide the appropriate `#include`.

We're going to work with several variables here. To help you keep these straight, here's a quick reference:

- M – a positive integer that is an argument to `experiment()`. M is the *approximate* number of keys you will store in your hashtable.
- W – the number of words in the dictionary.txt file ($W = 172,823$). You will use a randomly sampled subset of strings from this file. You should probably just hard code W as a constant in your code.
- p – for each word in our dictionary, p is the probability with which you include the word in your experiment. You will compute p so that the expected value of the number of words included in our experiment is $2M$.

Here are the steps your `experiment()` function should perform:

1. Open the `dictionary.txt` file for reading. Make sure this file is in the same directory or folder as your `run-main` program (most likely the `build` directory). There are $W = 172,823$ words in this file.
2. Sample the words from `dictionary.txt` to include in the experiment. You want to do a random sampling of words so that repeated calls to `experiment()` with the same M argument yield a statistical sampling of hashtable performance. Also, we want you to test your hashtable with roughly equal numbers of searches for words in *and* not in the hash table. You therefore need approximately $2M$ randomly selected words to work with - about half of these will go into the hashtable and half won't. But, you need to store all of the $2M$ words sampled from the dictionary file for later use.

   This second step might sound complicated but we can accomplish it with a simple algorithm and basic probability. If we want a random sample of (about) $2M$ words from the dictionary file, then each word in `dictionary.txt` should have a $p = \frac{2M}{W}$ probability of being used in the experiment. Of these $2M$ words, we want about half of them to get inserted into the hashtable. The *sampling loop* of `experiment()` should therefore look something like this:

   ```
   // read the entire dictionary.txt
   ```

```
while( inputfile >> str ) {
  // should we use it for the experiment?
  if( bernoulli(p) ) {
    /** save str for testing, maybe in a container named words? **/
    // should it go into the table?
    if( bernoulli(0.5) ) {
      table.insert(str)
    }
  }
}
```

Here we assume `inputfile` is your input file stream (`ifstream`) object, `str` is a string variable, and `table` is your hashtable object. The `bernoulli()` function comes from the `random220` module provided in the project github repository. The `bernoulli()` function is a *random variate*; it is a software wrapper around a pseudo-random number generator (pRNG). A call to `bernoulli(`p`)` returns **true** with probability p, so if you made 1000 calls to `bernoulli(`p`)`, about $1000p$ of them would have true return values and about $1000(1-p)$ would return false values.

3. Measure the average number of comparison operations required in a call to `.contains()` on your hashtable. Remember, about half of the time you want to measure words in the hashtable, and half the time words not in the hashtable. From previous steps, you should already have a collection of approximately $2M$ words which contain roughly equal amounts of words in and not in the hashtable. So you can just test random words from this collection enough times to get a good estimate of the average.

   You do not need to test every single word you saved in the sampling loop. With large values of M, you will not gain any great amount of accuracy for the additional amount of work. Your *testing loop* can play the same game as the sampling loop, shooting for a sample size of about 1000:

```
double tests = 0;
double comparisons = 0;

// for every word in the experiment (not every word in the dictionary!)
for( string str : words ) {
  // should we use it for testing?
  if( bernoulli( 1000.0/words.size() ) ) {
    tests++;
    comparisons += table.count_comparisons(str);
  }
}
```

4. In this single experiment with a table of size `table.size()`, you performed `tests` searches which resulted in `comparisons` total comparison operations; your result (return value) from the experiment is the table size and average number of comparisons per search. That is, your data point is

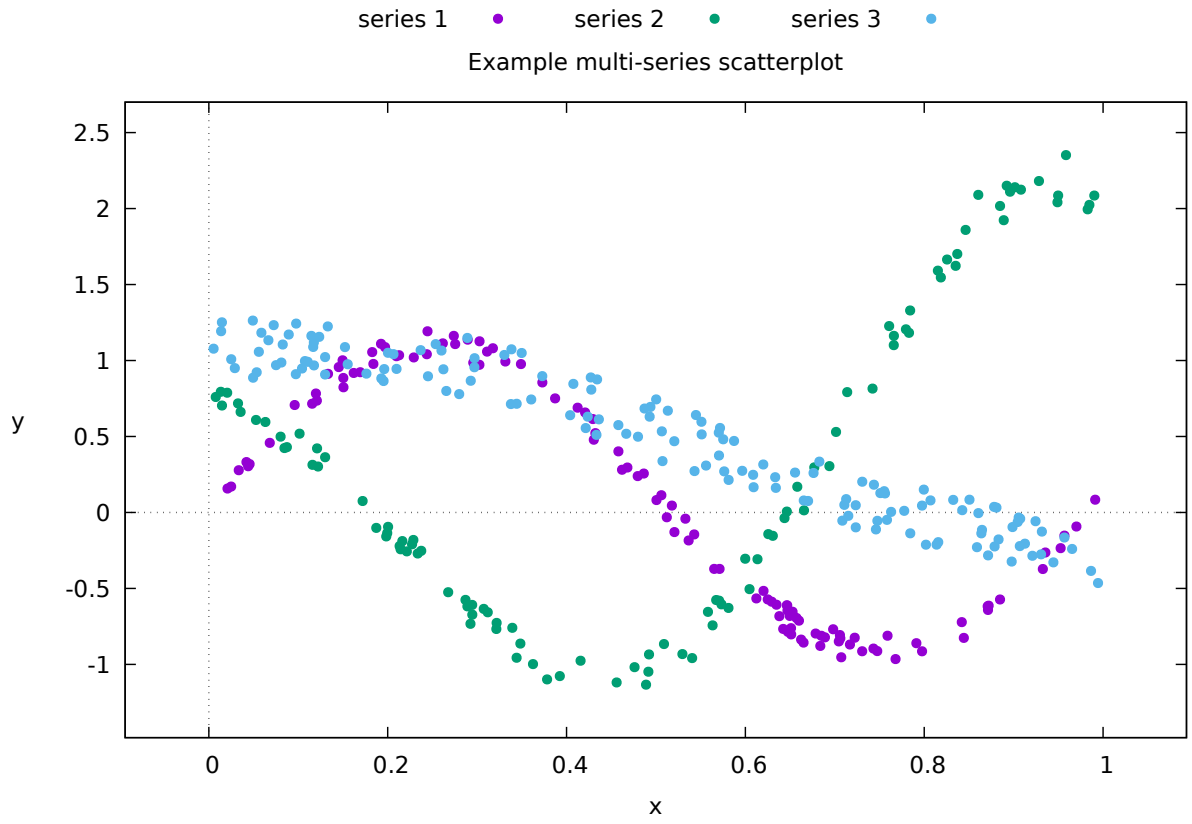
   `(table.size(), comparisons/tests)`

   While you are developing and testing your `experiment()` function, you can send these values to `cout` to verify that you are getting reasonable results for various values of ***M***. The next task will be to add code to collect data for a wide range of ***M*** values.

   Note: Your hashtable will have *about* ***M*** words in it, not necessarily ***M*** exactly. When outputting results from the experiment be sure to use the true size of the hashtable, not ***M***.

Run many experiments and collect data

The object of the first part of this analysis project is to produce a *scatterplot* of data from the `experiment()` function you've just written. A scatterplot is a two-dimensional presentation of *bivariate data*, (x, y) data pairs, without connecting lines. Several different datasets, or "series", are often shown in the same plot to ease comparisons. Here is an example scatterplot for some made up data:



You will make one or more scatterplots showing the data from your experiments for each of the hash functions provided in the project repository (in `hash_functions.h` and `hash_functions.cpp`). You will plot the hashtable size on the independent axis and average

comparisons on the dependent vertical axis. Your scatterplot will *not* look like the example above. In your scatterplot,

- The independent axis should range from zero to $W/2$. (However, to avoid issues with the probabilistic sampling we outlined above, you should collect data starting at M = 1000 or higher.)
- The range of the dependent axis should be guided by the average comparisons in your collected data.
- Of course, you should appropriately label your axis, title your graph and provide a legend (key) for the series plotted.

Note: The starter code in `main.cpp` of the `2P-Hashtable` repository shows how to create a hashtable that uses a particular hash function.

How many data points you plot is up to you, but with too few you will be unable to extract useful insights from your data. Run enough experiments so that you can see the patterns associated with each hash function. Then run some more experiments to confirm your results. **Create one or more scatterplots with convincing evidence that supports your conclusions and explanation of the results in part B. Include your scatterplot(s) in your report.** (Rubric item **A1**)

The `experiment()` function, as described, is not sufficient by itself for generating the amount of data you will likely need – you don't want to recompile and run for each different value of M and hash function! Fortunately, computers are very good at performing tedious tasks if given correct instructions. So, write some code to run all of your experiments for you.

Here are some things you might consider as you automate your experiments:

1. How does your preferred plotting software want bivariate data? A simple text file? CSV? Can all the series data be in one file, or do you need five separate files?
2. Will you read the `dictionary.txt` data into a vector first? Or read from the disk file for each experiment?
3. Will your `main()` routine generate all the data for every hash function? Or will you recompile for each? Or will you run it from the console with command line arguments?
4. Finally, how will you pick table sizes for testing? Going methodically from 1000 to $W/2$ in small fixed increments is one possibility; make sure you have enough data points to catch all the nuances of the data relationships. You may need to use trial and error to determine the correct increments. (The lower bound of 1000 ensures we have enough data to actually run our approximate 1000 searches.)

   Another approach is to randomly sample table sizes from the range. The `random220` module provided with the repository has a function, `equalikely()`, that is ideal for this task. Invoking the function with

   ```
   equalikely(1000, W/2)
   ```

   results in an integer randomly selected from the interval $[1000, W/2]$. Running many experiments each with a different randomly selected M value should give you good coverage over the desired range. See `random220.h` for details.

How you orchestrate the data generation and interaction between `main()` and `experiment()` is your choice. One important step that should be done is to *seed* the random number generator used in random220. Do this in `main()` *one time only*, before any calls to `experiment()`, with the invocation:

```
seed_now();
```

This seeds the pRNG with the current wall clock time, and prints the value used to `cerr`. This will assure that regardless of the particular implementation of the pRNG, you will get different sampling patterns with each run of your application (provided they are at least one second apart).

B. Discussion and Analysis of Results

When you are happy with your plot(s) from part A, answer the following questions completely and concisely in your write-up:

1. **Do the scatterplot(s) inform you of the individual or relative performance of the five hash functions used? List the hash functions in order from best to worst (you may have some ties). Justify your rankings using features of your scatterplot to support your argument.** (**B1**)
2. **Consider the hash function which you found to be the best. What pattern(s) do you find in the plotted data for this hash function? Provide an explanation for any patterns, using your knowledge of your hashtable implementation. Be specific in your discussion. Can you provide a method by which we can generally predict performance of the hashtable (using this hash function) for arbitrary table sizes?** (**B2**)
3. **Next, consider your plot for the `hash0` hash function. How does its behavior differ from that of the hash function you found to be the best? Look at the definition for `hash0` in `hash_functions.cpp`. Explain the behavior you see in your data based on the function definition.** (**B3**) *Hint*: how are the hash codes generated by this function distributed across table indexes?

C. String Searching

In this project you have been exploring hash table performance in the context of string data using five different hash functions. String hashing has applications in other areas of computer science, including string searching. For this part, you will explore a string searching algorithm which uses a *rolling* hash function to improve efficiency.

Problem Definition

In the string-searching problem, we are trying to locate a substring (pattern) in a larger string. For example, in the string “algorithm”, the pattern “or” is at index 3, so we would return 3 when searching for that pattern. The pattern “and” is not in the string, so we might return -1 to indicate that the pattern was not found.

Naïve Approach

First, let us consider a brute-force algorithm to perform string searching. Let’s go back to the example above. If we are looking for the pattern “or” in “algorithm”, we start at the

first 2-character substring “al” and compare it with “or”. This isn’t a match, so we *slide* one character to the right to the substring “lg”; this isn’t a match either. We continue sliding over one character at a time until we find the substring, returning the corresponding index. If we get to the last substring and didn’t find a match, we return -1.

What is the worst-case time complexity of the naïve string-search algorithm where the length of the string is n and the length of the pattern we’re looking for is m? Briefly justify your answer. (**C1**) Remember that comparing two strings is not a constant-time operation!

A Better Approach

Consider the following pseudo-code for a different algorithm for string searching. The function STRINGSEARCHHASHING looks for the substring *pattern* within a larger string *text*.

```
function STRINGSEARCHHASHING (text[1 .. n], pattern[1 .. m])
    h_pattern ← HASH (pattern)
    for j from 1 to (n − m + 1) do
        h_substring ← HASH (text[j .. (j + m − 1)])
        if h_substring = h_pattern then
            if text[j .. (j + m − 1)] = pattern then
                return j
    return −1
```

Note that this pseudo-code indexes by 1 for readability. For string s, the expression $s[i \,..\, j]$ represents the substring of s starting at index i and ending on index j. Without [ ], s means the entire string.

Do you see the speed-up? By comparing the hashes of the pattern and substring, we can avoid performing the full string comparison unless the hash values are the same. If the current substring’s hash matches the hash of the pattern, then we confirm that they match and return the index.

There is one snag with this algorithm. Because a good string hash function uses every character of the given string, the complexity of using such a hash function would be $O(m)$ for every substring. So, it would seem our attempt to improve the naïve algorithm has fallen short.

How can we fix our hashing algorithm so that we improve its time complexity? We need to make our hash function calculation in constant time, so that its cost no longer depends on the length of the pattern. This is where a *rolling hash function* becomes useful. A rolling hash function provides hashes for a sliding window through an input, *using the previous hash in its calculation of the current hash*.

For example, let’s go back to the string “algorithm” and consider searching for the pattern “and” of length 3. We must initially compute the hash of “alg” using an $O(m)$ hash function - this initial cost cannot be avoided. The next hash we’d need to compute is for the substring “lgo”. Note that this substring shares two characters (‘l’ and ‘g’) with the substring whose hash we previously computed. If we are clever, we can reuse the calculation that has already gone in to the previous hash function for these characters and avoid recalculating their contribution to the new hash function.

For this part of the project we will use the hash function

$$hash(s) = \sum_{i=0}^{m-1} 31^{m-i-1} s[i]$$

where $s[i]$ represents the character code of the i^{th} character in a string s of length m (using 0-based indexing).

As an example, let h_j be the hash code for the substring of length $m = 4$ starting at index j within string s. From the above equation,

$$\begin{aligned} h_j &= s[j] \times 31^3 + s[j+1] \times 31^2 + s[j+2] \times 31^1 + s[j+3] \times 31^0 \\ &= ((s[j] \times 31 + s[j+1]) \times 31 + s[j+2]) \times 31 + s[j+3] \end{aligned}$$

The latter formula results from applying Horner's method to our summation (effectively a degree m — 1 polynomial calculated with $x = 31$). Using a loop, we can compute this hash code in $O(m)$ time.

As discussed above, we don't want to do this entire calculation at every index. Instead, write an *update equation* – an equation that calculates h_{j+1} from h_j, s, and m. (**C2**) Write your equation generally in terms of the pattern length, m, not for the specific case of $m = 4$ used above. It must be possible to implement your update equation to run on a computer in $O(1)$ time. You may assume that your output is an unsigned integer type with no bound (no need to account for overflow).

Finally, analyze the hashing string search algorithm using a rolling hash function. What is the worst-case time complexity, assuming our hash function is of good quality (i.e., we never obtain the same hash code for two different strings)? (**C3**)

Rubric

Grade Calculation

For this project, parts A, B, and C are graded as one score. This composite score and the Presentation score are used to calculate the overall grade. For an S on parts A - C, you must provide Satisfactory responses to 6 of the 7 elements defined in the rubric. Similarly, an S on Presentation requires Satisfactory performance on 4 of 5 Presentation categories.

An S on this project requires an S on both scores; an E requires at least one E score.

Part A

A1: Experiment Scatterplot Plot(s)	
S	Each scatterplot has a title, correctly labeled axes, and a legend. Plots show results for different hashes using distinct symbols or colors for each and provides a key or legend.
N	Plot has missing data, is poorly formatted, has poorly chosen axis ranges or scales, or does not permit clear examination of all the data.
U	No plot

Part B

B1: Ranking the hash functions	
S	Correct rankings provided. Data or scatterplot(s) (from part A) are used to justify rankings.
N	Same as S, but there are errors/misunderstandings in the ranking or discussion.
U	No discussion

B2: Patterns in the data for a good hash function	
S	Patterns are clearly and thoroughly discussed. Patterns are explained in terms of the hashtable implementation. A method for predicting patterns for larger table sizes is provided and justified.
N	Same as S, but there are errors/misunderstandings in the discussion.
U	No discussion

B3: Examination of a specific hash function	
S	Behavior of the hash function is compared with that of a good hash function. Behavior is explained in terms of the implementation of the hash function and hashtable implementations.
N	Same as S, but there are errors/misunderstandings in the discussion.
U	No discussion

Part C

C1: Worst Case Complexity for Naïve Algorithm	
S	Correct with some explanation/derivation
N	Incorrect or no explanation/derivation
U	No equation

C2: Rolling Hash Update Equation	
S	Correct with some explanation/derivation
N	Incorrect or no explanation/derivation
U	No equation

C3: Time Complexities of Rolling Hash Function Algorithm	
S	Correct Time Complexity with some explanation/derivation
N	Incorrect Time Complexity or no explanation/derivation
U	No equation

Presentation

Each element of Presentation is graded S or N.

Element	Requirement for S
Clarity	The text contains minimal vague language. The text uses complete sentences (outside of bullet points, labels, or mathematical derivations). The reader can generally follow the meaning of sentences without a second read through.
Conciseness	The text avoids redundant, overly complex, or run-on sentences and paragraphs.
Math	Equations are properly typeset. Derivations leave space between steps for readability, and enough steps are included that the reader can easily follow the reasoning.
Plots	Plot type selection is appropriate to the data being plotted; e.g., measurements collected as a function of some continuous input variable should be plotted as line plots. Scatterplots should be used when multiple different measurements are taken for a given input; e.g., when sampling data points from a random process. Measurements for which the input variable is categorical rather than continuous should use bar charts. Axes should be clearly labeled. • For line plots: each data point is indicated with a marker, and markers are joined by lines to more easily determine the shape of the function. Measurements are indicated along the y-axis, and input values along the x-axis. If the plot compares multiple data sequences, they are plotted with different colors and markers, and a legend is included to label the sequences. Axis tick marks and labels are readable and frequent enough for easy comparison of data points. • For scatterplots: same requirements as for line plots except that markers are not joined by lines. • For bar charts: input values are placed along the x-axis and are clearly labeled; measurements are indicated by bar heights. The y-axis has clearly labeled tick marks. If the plot compares multiple measurements, they are plotted with different colors. The plot is not crowded or unreadable.
General style	Sections are clearly labeled or titled. Formatting (e.g., indentation, font size) is consistent. Separate paragraphs are used for each discussion or point. Spelling and grammar errors are minimized (errors of this type are permitted, as long as we can follow the text without undue effort).