B.Sc. Thesis
Bachelor of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Search Engine Project

Searching Through Wikipedia

Thomas Løye Skafte

Kongens Lyngby 2018

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

This thesis concerns itself with the use and theory of basic algorithms and data structures in order to improve upon an initially poor performance search engine. The objective is to modify the search engine to compactly represent large data-sets in a scalable and efficient manner, to perform fast searches. Improvements are done in a step by step fashion, modifying one aspect at a time. For each modification an analysis and empirical test is performed to quantify the performance. The final version utilises a chained hash table with a c-universal hash function and a look-up table to reduce redundancy of memory usage. This version is capable of searching through the entirety of Wikipedia from 2010.

Preface

This thesis was prepared at DTU Compute at the Technical University of Denmark in fulfilment of the requirements for acquiring a B.Sc. degree in Software Engineering. The thesis was completed in the period 19-01-2018 to 14-06-2018 with guidance from Inge Li Gørtz and Eva Rotenberg. The project is valued at 15 ECTS points.

The thesis deals with analysis and implementation of algorithms and data structures used in a basic search engine and analysis of empirical data from tests of these algorithms and data structures.

Kongens Lyngby, June 14, 2018

Thomas Løye Skafte

Acknowledgements

My supervisors Inge Li Gørtz and Eva Rotenberg have my sincere gratitude for helping me structure the project along with general guidance and for answering my countless questions.

Contents

CHAPTER 1

Introduction

Search engines are used extensively in many different fields, and are a very useful tool. The implementations may vary, but the cores concepts that are at the heart of simple search engines are often very efficient and enough for most applications. To learn about these concepts I will be creating my own implementation of such a search engine, which objective is to search through words in Wikipedia returning the documents that the given search term occurs in.

1.1 Statement of Intent

The overarching goal of this thesis is to create a high performance Search Engine that can efficiently read and store a large amount of strings. This has to be done in a scale-able fashion that allows for quick searches.

The question for which the program should comply is *Which documents contain word x?*

The project will be improved in iterations, starting from an initial version, Index 1, provided by my supervisors Inge Li Gørtz and Eva Rotenberg. Additional information can be found at this link.[1]

Requirements It is required that any data structure used is created by me and the only Java packages that are approved for use are java.io, java.util.Scanner and java.lang.

The four mandatory versions are Index 1,2,3 and 4.

- **Index 1** – Returns true if searched term exists in the input data.
- **Index 2** – Returns a list of documents that the search term occurs in.
- **Index 3** – Modify the data structure to consist of a linked list of unique words, each with a linked list of occurrences.
- **Index 4** – Modify the data structure to use chained hashing, using Java's String.hashCode function.

[1]Search Engine Project - https://searchengineproject.wordpress.com/

With elected features producing Index 5 and 6. Here usage of any additional packages is acceptable.

- **Index 5** – Change the hash function to a universal random hash function.
- **Index 6** – Optimise the space used by converting linked lists to doubling arrays and by use of look-up tables.

The end goal is to have a search engine efficient enough to be able to search through the entirety of the Wikipedia.

1.2 Changes Made to Index 1

Before I started working on improving the search engine, I made some changes to Index 1. When you search in a search engine you look for specific words, and not specific strings. For example if you want to search for the word *binary* then you want all strings of this word, even if there is a string such as *binary!* or similar grammatical punctuation. Therefore whenever a string is read the last character is removed if it is a *comma*, *full stop*, *exclamation mark* or *question mark*. To what extend this kind of procedure should be implemented is difficult to quantify. I selected to do as described here.

In a similar vein, search engines should ignore capitalisation. I chose to force every input into lower case, since the majority of strings will be in lower case. This feature allows us to handle words at the start of a sentence equally with those that do not.

There are side effects of these changes which are not optimal. For example, searching for the abbreviation '*U.S.*' in a data set of which the string actually does occur, will return a incorrect result, since '*U.S.*' will be stored as '*u.s*'

1.3 Test Files

For testing I was given a snapshot of Wikipedia from 2010. It consists of documents for which the first line is the title, followed by the main text. The end of a document is signaled by a new lined that consists only of the string `---END.OF.DOCUMENT---`. Documents contain only text and is read using UTF-8.

As an example here is the document for the Wikipedia page *Actrius*.

```
Actrius.
"Actrius" (Catalan: "Actresses") is a 1996 film directed by Ventura
Pons. In the film, there are no male actors and the four leading
actresses dubbed themselves in the Castilian version.
Synopsis.
In order to prepare the role of an important old actress, a theatre
student interviews three actresses who were her pupils: an
```

```
international diva (Glòria Marc, played by Núria Espert),
a television star (Assumpta Roca, played by Rosa Maria Sardà) and
a dubbing director (Maria Caminal, played by Anna Lizaran).
---END.OF.DOCUMENT---
```

For testing an initial file of 329,292 words is used, where after each successive file is attempted to be doubled in size, or at least close to. The files are:

File	#words	Size
1	329,292	2 KB
2	828,068	5 MB
3	1,672,002	10 MB
4	3,348,766	20 MB
5	8,356,373	50 MB
6	16,717,033	100 MB
7	33,442,256	200 MB
8	67,263,274	400 MB
9	135,750,192	800 MB
10	990,248,676	6,000 MB

Every file is a subset of the following such that file1 $\subset$ file2 and so on. File 10 is the complete list of documents that combine to be the entire snapshot from 2010.

Computer Specifications
The specifications used to perform the tests are

- Intel Core i5-6400 CPU @ 2.70GHz (4 CPUs)
- 16384 MB RAM
- Windows 10 Home 64-bit (10.0, build 17143)
- Java Version 8 Update 172
  - With 8192 MB allocated memory (Using the -Xmx8192M flag)

1.4 Analysing Algorithms

Analysis is a vital part of developing algorithms, both in order to compare different solutions and for the sake of controlling that the implementation behaves in the desired fashion.

The run time, or other features, of an algorithm can be depicted as a function on the size of the input. A traversal through a list of size n will have n iterations of approximately the same time cost C. This can be interpreted as a the function

$$T(n) = n \cdot C$$

When analysing algorithms we prefer to talk about them in a generalised sense, since the size of C is going to depend on both implementation and computation power along with many other factors. Instead we prefer to focus on how well the algorithm handles up scaling of the input size. We can accomplish these goals by using the notion of *asymptotic notation* or *asymptotic complexity*, which can be read about in the book *Introduction to Algorithms* by Thomas H. Cormen et. al., [Cor+09, Chapter 3.1, p.44-48]. We will be using three different asymptotic notations, Θ, O and Ω. If $f(n)$ is the function of the run time of some algorithm, then the three notations can be explained as

- $f(n) = O(g(n))$ depicts the situation where there exists a constant c such that $c \cdot g(n) \geq f(n)$ for any n.
- $f(n) = \Omega(g(n))$ depicts the situation where there exists a constant c such that $c \cdot g(n) \leq f(n)$ for any n.
- $f(n) = \Theta(g(n))$ depicts the situation where there exists two constants c_1 and c_2 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

This means that when we find the asymptotic complexity of a function we can always remove any constant, even coefficients, as c can simply be picked such that it satisfies the notation. Similarly lower-order terms can also be ignored since n can be picked such that they are irrelevant.

In the previous example of list traversal we can pick c_1 and c_2 such that $c_1 \leq C \leq c_2$ and therefore

$$T(n) = \Theta(n)$$

CHAPTER 2

Index 1

Initially I was given an algorithm, *Index1*, that can search through a data file, containing different Wikipedia documents, with a specific setup. In order to improve upon this algorithm it is vitally important to properly understand how and why it works, along with an appropriate analysis of the run time, space and correctness of the different parts that combine into the algorithm.

When faced with a search problem it is often beneficial to look at it in two parts. The initial issue is to sort the data that you will be searching through in such a way that the actual search will be easier or at least take less time. This structuring of data is called the *data structure*.

The second problem is how do you traverse the data structure in an efficient fashion, in order to find what you are searching for. This largely depends on how the data structure is composed, and having a well thought out system often makes the actual search seem trivial and generously improves the run time.

I will be referring to the first issue of structuring the data as the *preprocess phase*, and traversing the data I'll refer to as the *searching phase*. The combination of both steps will be recognised as the complete solution.

2.1 Description

The problem that Index 1 solves is to find out if a given search term, exists within a text file. It does not care where nor how many times it occurs, but simply returns true if it finds a match and false if it reaches the end of the file.

Index 1 is essentially the simplest possible solution to the Search Engine problem. The method used can be compared to a human having a stack of paper and asked to find an occurrence of a given string, by checking each individual word from start to finish. Additionally there is no memory so if asked to find a different string afterwards the same procedure would have to be done again.

It does however quite clearly divide the problem into a preprocess phase and a searching phase. As such there is actually a data structure, but all it does is keep each word in the order it read it from the file.

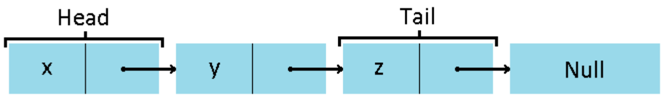
Whenever I will be describing or analysing a solution to the problem, I will talk about the preprocess phase and searching phase separately same as below.

2.1.1 Preprocess

The simplest possible way of structuring the data is to create an array big enough to contain all of the relevant words in the file. This would require that the amount of words in the file known in advance, which could be handled by counting all of them, but that would mean going through the file twice which is a waste. This issue can be circumvented by several solutions. In Index 1 a *singly linked list* solution is chosen.

A *singly linked list* is a data structure that consists of nodes that each contain a data point along with a pointer towards the next node, see figure 2.1. This creates a situation where there is a *head* node that is the initial starting point, which points at the next node in the data structure. That system then continuous until the *tail* node is reached, which simply has a null pointer instead of pointing at another node.

Figure 2.1: A singly linked list.



For this algorithm the nodes consists of a string containing a single word from the data file, which in the figure above would be x, y and z, and then of course the pointer to the next element.

This is implemented by cycling through the words in the data file and creating a node for each word. In this way duplicates are allowed and the ordering of the words in the data file will remain within the linked list structure.

The linked list solution is a very flexible and easy to work with solution, and the main downsides are:

1. Not being able to directly access locations within the list
2. Extra pointers taking up additional space
3. Poor cache locality due to nodes being spread throughout the memory

Since the data points are sorted by whatever came first, there is no structure that would allows us to utilise direct access if a linked list structure was not used. The two others are both reasonable criticisms of the linked list solution, and will be looked at in further detail in a later chapter.

Given that all that is desired is to cycle through the data from start to end, makes a singly linked a reasonable choice.

2.1.2 Search

This simple version of the algorithm returns whether the search term exists in the file or not, and as such the search only has to find the word and not where it is located.

Therefore a simple traversal through the linked list using the pointers, comparing the search term with the current word, is the only solution. Because it is only necessary to find a single example of the search term the algorithm can terminate after the first instance.

2.2 Analysis

In order to analyse the scale-ability of the solution I will be using the asymptotic notation explained earlier on both the preprocess phase, the searching phase and the space usage of the algorithm.

Afterwards we will compare the analysis to some empirical data from testing the different phases on the Wikipedia test files.

2.2.1 Preprocess

Since the data structure that is setup during the preprocess is simply a linked list containing an object for each word in the data file, the increase of run time is going to be linearly proportionate with the amount of words in the file.

The amount of time it takes to read in a single word and create a linked list object that represents the given word, will always take roughly the same amount of time, C. Say that n is equal to the amount of words in the file, then the total run time of the preprocess will be:

$$T(n) = n \cdot C \tag{2.1}$$

Which's asymptotic complexity is:

$$T(n) = \Theta(n) \tag{2.2}$$

A quick sanity check notes that this is the expected linear behaviour that was initially expected.

2.2.2 Search

A run of the search phase is as simple as traversing through the linked list continuously comparing the current selected word in the linked list with the actual search term provided. This means that the amount of iterations is equal to the amount of linked list objects, unless a match is found in which case the algorithm terminates.

Because we are interested in the run time of the worst case situation, we expect the algorithm to go through the highest amount of iterations possible, which is exactly the amount of objects in the linked list, and also corresponds to the amount of words in the data file.

Thus we again have the algorithm go through n iterations, each iteration having about the same run time, which we will call C, meaning the total run time of the search is:

$$\begin{aligned} T(n) &= n \cdot C \\ &= \Theta(n) \end{aligned}$$

This is not to say the preprocess and search part of the algorithm will have the same run time, because they have two different C values. The fact that both run times are linearly proportionate to the size of the input, means that the ratio between the two run times will stay the same, no matter the size of the input.

2.2.3 Space Usage

The program creates a linked list element for each word, so the amount of space that the data structure takes up will be however large a single element in the linked list is, C, times the amount of words, n.

$$\begin{aligned} \text{space} &= C \cdot n \\ &= \Theta(n) \end{aligned}$$

2.3 Test

In order to figure out if the analysis correlates with actual reality, it is possible to compare our theory with actual evidence from tests. The asymptotic complexity that was derived tells us how the run time of the algorithm changes depending on how big the input data is. It is not sufficient to simply figure out how big C is, but to satisfyingly proclaim that the proposed asymptotic complexity correlates with the implementation, would require that the run times actually does increase linearly compared with the size of the data file.

In order to show this both the preprocess and searching has been run on the 6 smallest test files from Wikipedia. The last 4 were too big to fit in memory.

file	n	$\#documents$
1	$329,292$	126
2	$828,068$	359
3	$1,672,002$	842
4	$3,348,766$	1745
5	$8,356,373$	4182
6	$16,717,033$	8246

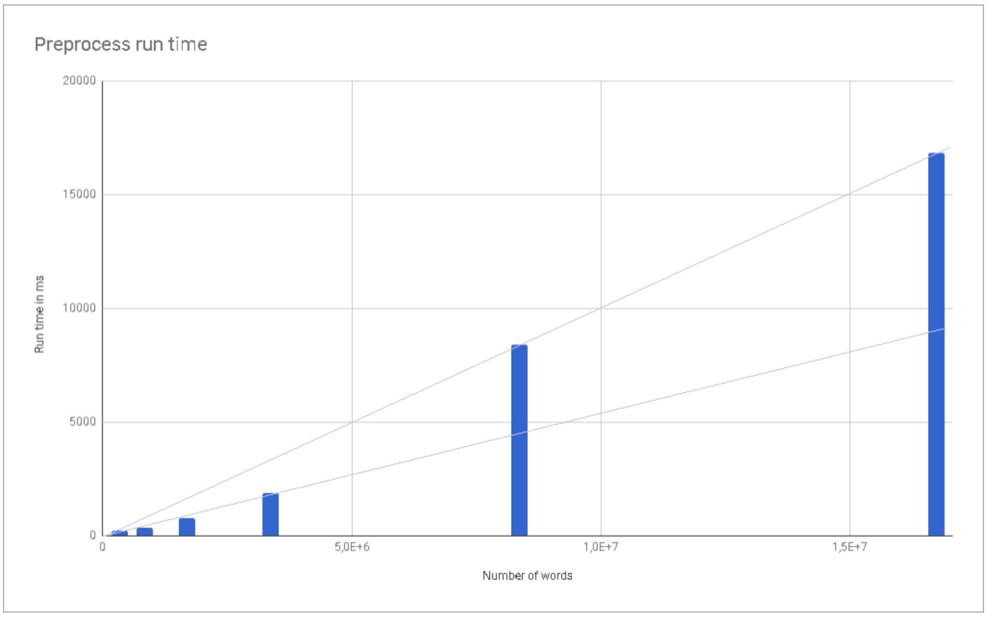
Generally the preprocess phase takes a lot longer than a single search, so the amount of runs for each phase is also different. Throughout the thesis the preprocess phase will be run 10 times on each file tested on, and the search phase will be run 100 times.

The median of all of these runs is chosen as the result of the test. This is done to have unrelated stress on the CPU have minimal affect on the results.

2.3.1 Preprocess

The preprocess has the asymptotic complexity of $t = \Theta(n)$, which agrees perfectly with the expect linearity relation between run time and size expected from our initial look.

Figure 2.2: Index 1 preprocess run time.



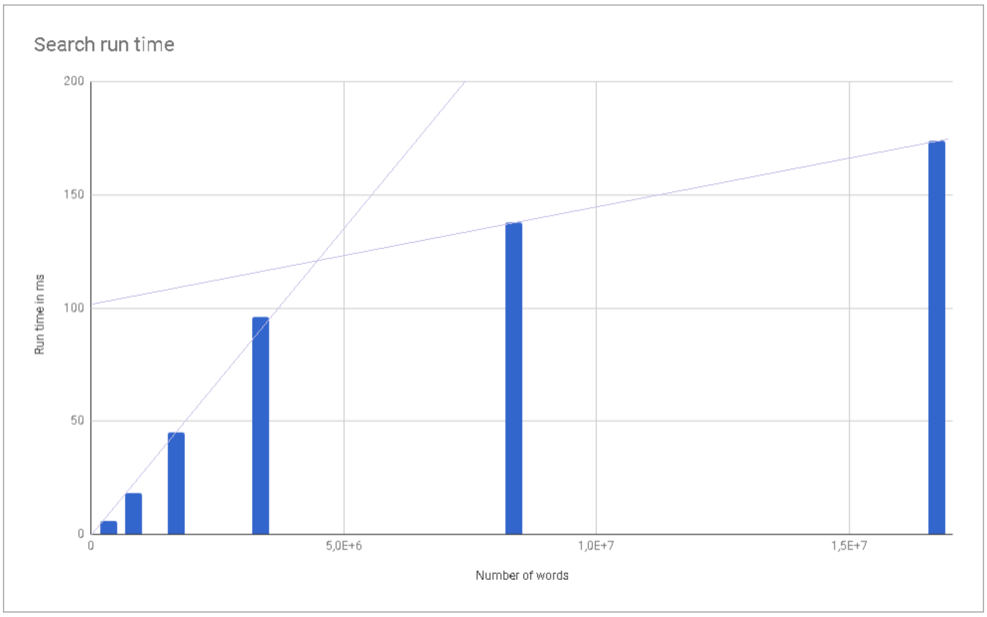
It might seem odd at first that there are two trend lines, however this is due to switching to a bigger cache hierarchy, which is farther from the CPU, increasing the size of C.

Besides that it shows very nicely that the run time of the preprocess indeed does follow linearly with the size of the data file.

2.3.2 Search

The search was expected to have the same asymptotic complexity of $t = \Theta(n)$, which means its run time also should follow linearly. Because we want the worst case situation the random non-sense string " %&/#%&(% " was used as search term, in order to guarantee that the searching continues all the way to the end.

Figure 2.3: Index 1 search run time.



Again we see the shift in cache hierarchy at the same file as before. Besides that the test does show the run time to be linear proportionate to the input.

2.3.3 Space Usage

Just like with run time we are interested in figuring out how the space usage scales up when the input size increases. Therefore we are interested in the asymptotic complexity of the space usage. To check that the analysis is correct we can count the amount of objects being made and see if it is equal to the input size as the analysis says

Table 2.1: Count of linked list objects.

File	n	#linked list objects
1	329,292	329,292
2	828,068	828,068
3	1,672,002	1,672,002
4	3,348,766	3,348,766
5	8,356,373	8,356,373
6	16,717,033	16,717,033

Which it turns out it does.

2.3.4 Correctness

To ensure that the program behaves correctly I run the preprocess on the smallest file used for the other tests and then search for each word in the file. If any of these searches return false, then there is an issue, but luckily that was not the case.

Additionally a search on a word that I know with garentee is not in the file is performed. The search string used earlier, " %&/#%&(% ", is used again. The search returns false.

These two tests are performed on every version of the search engine. If not otherwise stated it is to be assumed that the tests provided the same results as here, and therefore function as desired.

CHAPTER 3

Index 2

The version of the algorithm from part one simply returns whether or not an occurrence of the search term exists within the data file. For this version, suppose we desire to return every single document where the requested search term occurs. This can be implemented by changing solely the searching part of the algorithm, and as such the preprocess is exactly the same as the previous version.

3.1 Description

In order to create the list of occurrences, some way of distinguishing between documents is required. As mentioned in the introduction, the documents are separated by the string '---*END.OF.DOCUMENT*---' and the following line will always be the subsequent document's title. This can be used by keeping a string dedicated to having the current document title and then changing it whenever a '---*END.OF.DOCUMENT*---' occurs. During the search a list of documents wherein the search string appeared can be kept.

A small improvement is to skip the comparisons whenever an occurrence of the search term appears and just skip ahead to the next document. Unfortunately the linked list still needs to be traversed through to find the next document because it is kept as a linked list. This can be fixed by keeping pointers to the start of documents in a side table, but as you will see in the next chapter there are more important improvements to do.

In this way each document is searched through one by one, and every document with an occurrence of the search term is added to the list of document titles, with no occurrences no matter the amount of instances of the search term in a given document.

3.2 Analysis

As mentioned earlier the traversal of the data structure is the only part of the algorithm that has been changed, so that is the only part to be analysed.

Essentially not much has been altered, except the actual output. In order to talk about the run time of the worst case situation, it is required to firstly figure out what the worst case situation is. In the previous version of the algorithm the worst case situation of the traversal of the data structure was when the search term was never found, because if it ever was the algorithm was done and could be terminated. So in the worst case situation the whole linked list would be traversed.

In this version the whole linked list is going to be traversed no matter if it is found or not. This is the case because, say an occurrence is found, then it is still needed to find the next document and the only way to do that is to keep looking at the next element in the linked list to see if it is the '---*END.OF.DOCUMENT*---' indicator.

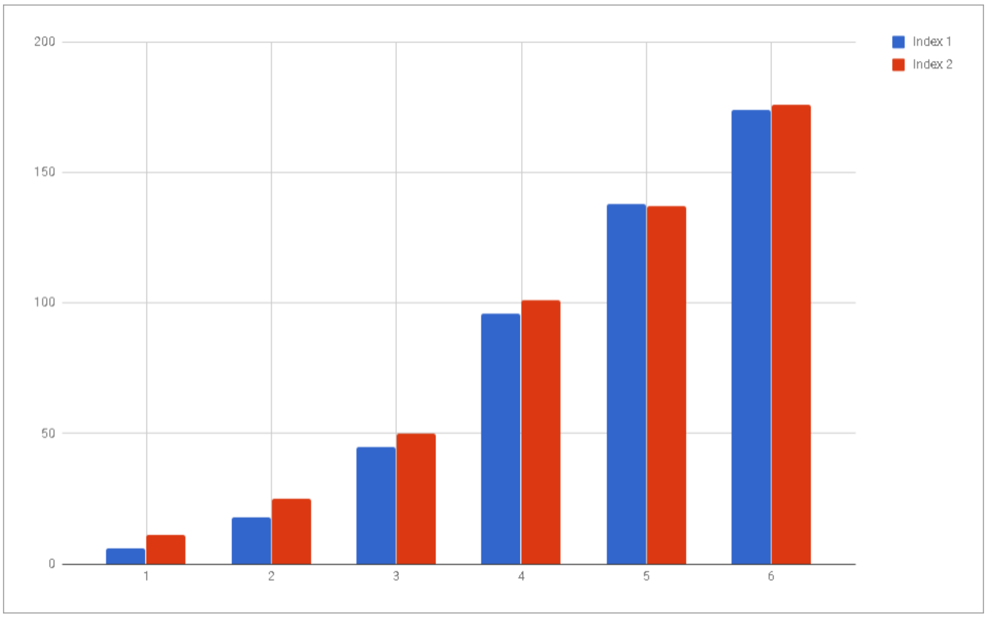
This does not change the worst case asymptotic complexity, since both versions traverse the whole linked list in that situation, and therefore the asymptotic complexity is going to remain the same.

$$t = \Theta(n)$$

3.3 Test

It is to be expected that the run time is by in large the exact same. The following histogram shows the exact results for you to compare. n is the amount of words in the file.

Figure 3.1: Search run time comparison.



These results are about as close as it gets, showing that the result is as we expected.

CHAPTER 4
Index 3

In the English language there are quite a lot of words which are used extensively, obvious examples are *and*, *the*, *is* and so on.

In the previous version of the algorithm these words would create a linked list element each time they are found, meaning that when traversing through the data structure and comparing the element's strings to the search string, it might happen that the search string is compared to words such as *the* several times, which is obviously inefficient, as a single comparison would be enough.

This could easily be fixed by not adding words that already exists within the linked list. In that way comparisons only happen once and the size of the linked list is smaller so there are fewer nodes to traverse through. However now we can no longer use the same method for finding each document the search term occurs in. Instead we can utilise that each node in the linked list is unique, and simply store a list of documents that the specific word occurs in.

4.1 Description

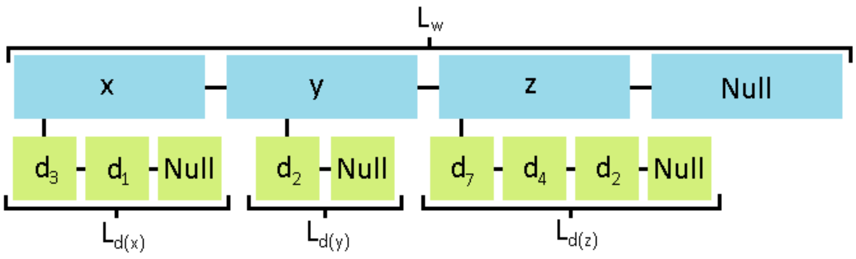
Using this scheme the linked list is more so a list of unique words that occur in the whole set of words, where each unique word has a list of documents where it occurs. This is a slight change of what kind of information the data structure holds. It used to hold every word in the order it was found in the text file, and now it keeps track of unique words only, along with a list of where they appear.

Since it is not certain how many documents a word may appear in, a linked list will be used for the list of documents. The functionalities that are lost by using a linked list are not useful to us.

Now that we have two different kinds of linked list they will be mentioned as L_w for the linked list of unique words and $L_d(u)$ as the linked list of documents for the unique word u. The Java objects that make up L_w will be called *WikiItem* objects and for L_d they will be called *DocItem* objects.

A depiction of an instance of the data structure could be:

Figure 4.1: Example of the data structure.



4.1.1 Preprocess

Since the architecture of the data structure has changed, then the naturally the construction of it has as well. At any point during reading from the text file it is important to keep track of what document is currently being read from.

When a word, x, is read, it is needed to check if that is the first occurrence of x. This is done by going through L_w and comparing each word in there with x. If no match is found, then this is indeed the first instance of x and x is added to L_w, along with a $L_d(x)$, in which the current document is stored.

However if there is an instance of x in L_w, then we need to ensure that there is an instance of the current document in $L_d(x)$, adding one if there is not. Here we can utilise the way the text file is constructed. Since every word in a given document comes in a row, we only need to check the most recently added document title in $L_d(x)$ to decide whether or not the current document is in $L_d(x)$.

To show that this is the case consider this contradiction. x is read, and found in L_w. The current document is in $L_d(x)$, but it is not the most recently added element. This means that x was read in a document, say d_1, and there added to the data structure. Then a document switch was done to d_2, and x was read again, so that d_2 is the most recent document added to $L_d(x)$. But now x is read again in a document d_1. This means that d_1 is split up in two, and that is against the structure of the text file.

We can use this by always inserting in L_d at the head of the linked list. This way whenever we need to ensure that the current document is in L_d we simply need to check the head of $L_d(x)$.

4.1.2 Search

The goal of searching the data structure is to return the correct L_d that corresponds to the word searched for. Any word that can be found in the set S of words from the input, is in L_w and that instance has a pointer to the correct L_d.

All that is needed is to find the correct word in L_w and then return the corresponding L_d. If there is no instance of the search term in L_w then that means the word is not in S. Given that the word was found in L_w then there will always be a L_d with at least one entry, from when the word was initially inserted in L_w.

4.2 Analysis

4.2.1 Preprocess

Since the preprocess phase consists of inserting all the data from the text file into our data structure, we need to figure out how long a single insertion takes to figure out how long the entire preprocess takes.

When reading a word, x, form the text file we will always need to check through L_w in order to see if it is the first occurrence or not. Depending on the result we will either have to check L_d or insert a new element in L_w. Additionally we might have to insert an element int L_d even after the check. Luckily these actions take constant time, and as such the outcome does not have much impact on the run time, so for the complexity it can be ignored.

$$T(\text{insertion}) = O(T(\text{check}(L_w)))$$

To check if x is in L_w it is needed to check against every element in L_w. We know that for each unique word that has been read so far there is one entry in L_w, $|L_w| = u$, where u is the amount of unique words. A traversal through L_w will therefore take u units of time.

This potentially has to be done for each word in S, meaning n times. Which leaves us with a initialisation run time complexity of

$$T(n) = O(u \cdot n)$$

4.2.2 Search

The searching consist simply of searching through L_w and then returning L_d for the search term. We already know that searching through L_w will take u amounts of time and since L_d is a linked list, returning the entire content means going through every entry, which will take $|L_d(x)|$ which we will denote d_x. A single look-up thus takes

$$T(n) = O(u + d_x)$$

If every word in S is unique, then every entry from S is in L_w and we have the same situation as previous. However, in an average text duplicates occur all the time, meaning this complexity is a lot better.

4.2.3 Space Usage

The difference in memory usage of WikiItem and DocItem objects are of a constant degree, so for asymptotic complexity we simply desire to know how many of these objects are created.

By implementation L_w has one object for each unique word, $|L_w| = u$. If each word $x \in S$ only occurs at most once in each document, then the amount of L_d objects is n. The only way this number changes is if the same word occurs more than once in a document, in which case there are at most n objects, therefore $|L_d| \leq n$.

The total space used to store set S of size n is

$$\text{Space}(n) = O(n)$$

4.3 Test

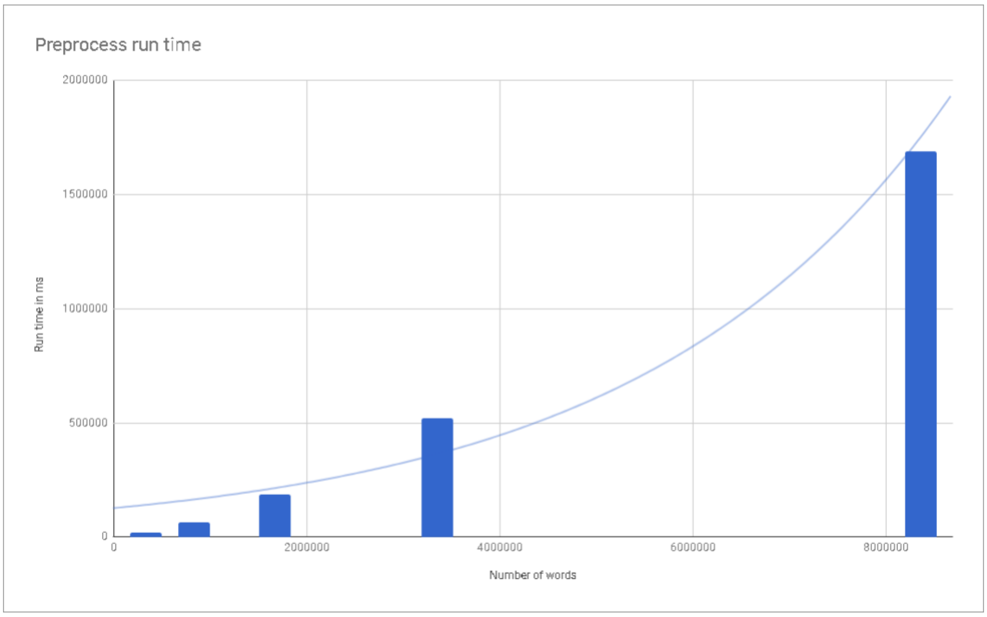
When testing to see if the results from your analysis holds up to reality it is important to ensure that the variables that are plotted are relevant. In the previous version the run time was plotted against n, the amount of words in the file. This was to show that as n increased, the run time increased linearly along with it, showing the asymptotic complexity from the analysis was correct.

For this testing we also have the option to compare the results to the previous version, seeing how well the improvements that were implemented worked. In order to do this it is important that the tests were carried out in similar fashion, such that it is a fair comparison.

4.3.1 Preprocess

In the analysis the complexity was found to be $O(n \cdot u)$. Since $n \geq u$, we can also say that $n \cdot u \leq n^2$, which means we will be plotting against n and expect a quadratic increase as n increases.

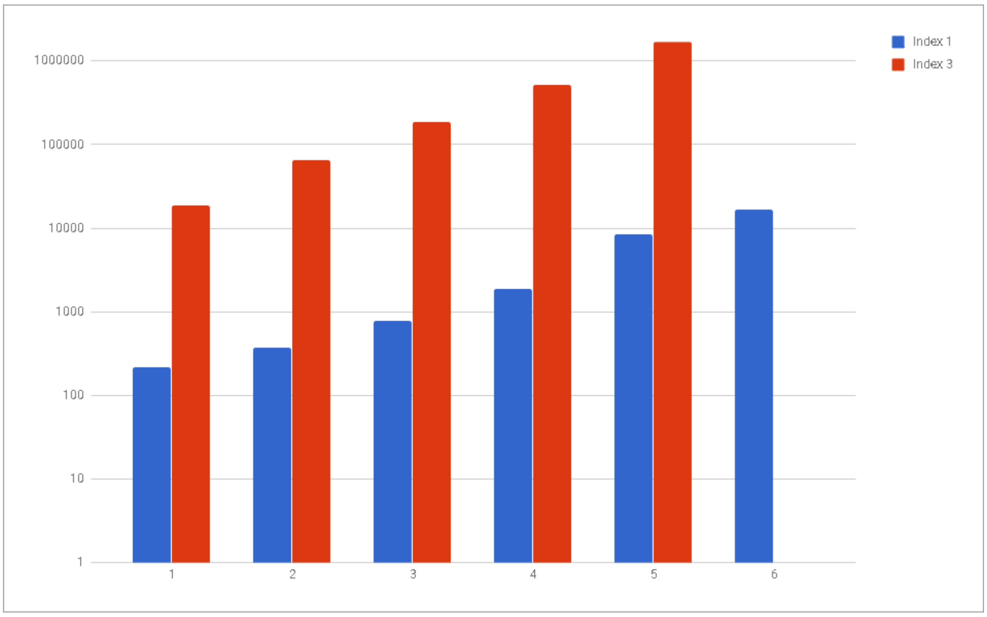
Figure 4.2: Index 3 preprocess run time.



Unfortunately the last file is so big that the preprocessing took over half an hour, which was deemed too much. This graph quite clearly demonstrates a quadratic increase in run time as u increases, which correlates exactly with what we found in the analysis.

To compare the run time of the preprocess against Index 1 the following histogram is constructed. Logarithmic scale is used due to the discrepancy making it imperceptible otherwise.

Figure 4.3: Preprocess run time comparison.



Index 3 is a lot slower, but this is expected from the analysis as $u \cdot n$ is a lot bigger than n. The fact that the second biggest file took a total of about 28 minutes along with the biggest taking over 30 minutes, implies that this solution has too big of a setup and something has to be changed.

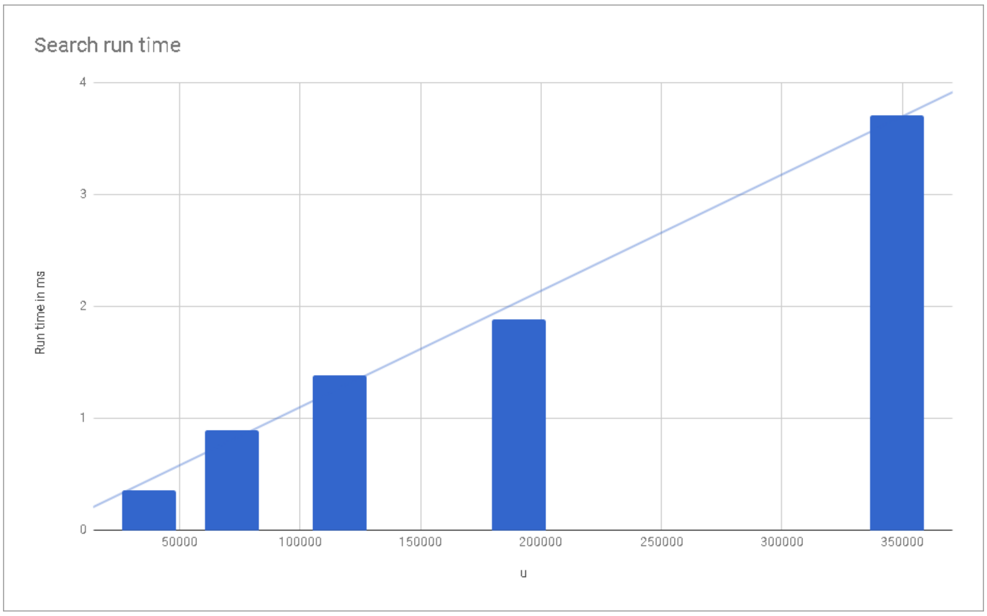
4.3.2 Search

In the analysis for the search the complexity $O(u + d_x)$ was found. In the test data u is prominently the dominating variable in $u + d_x$, so we will plot the run time against u

So that the result can be compared to the testing of Index 1, the same non-sense string " %&/#%&(% " was used to ensure a search that traverses through the entirety of L_w.

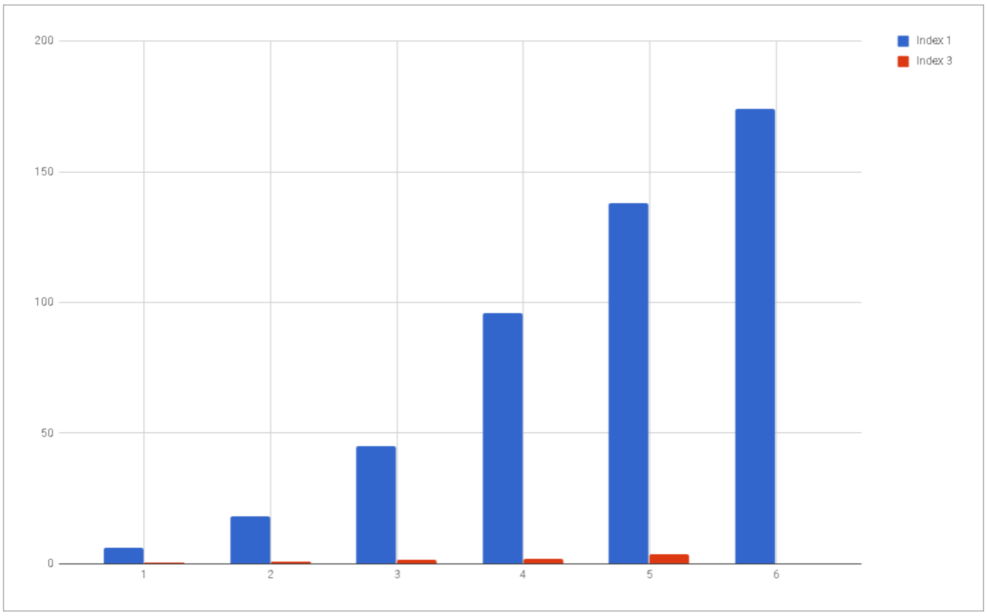
Here a slight problem arises, in that the search is so quick that the test of some of the files take less than one millisecond, rendering it impossible to time them. In order to combat this issue, the test was changed slightly so that for each of the original 100 runs of the test, the search is done 100 times, and then divided by 100. By doing this the test is capable getting an actual number representing the run time of the search.

Figure 4.4: Index 3 search run time.



When comparing to Index 1 the improvement is massive, by around a factor of 15-50. Logarithmic scale is not used due to some of the results being below 1 ms.

Figure 4.5: Search run time comparison.



4.3.3 Space Usage

A complexity of $O(n)$ simply means that the amount of objects is less or equal to the input size n.

To find the exact amount of linked list objects counters can be used, and increment them whenever new objects are made. This yields the following data. The amount of WikiItem objects in L_w will be named V_w and V_d for DocItem objects.

File	n	V_w	V_d	Total	Ratio
1	$329,292$	$37,232$	$113,307$	$150,539$	0.46
2	$828,068$	$71,494$	$289,279$	$360,773$	0.44
3	$1,672,002$	$116,600$	$600,650$	$717,250$	0.43
4	$3,348,766$	$190,748$	$1,202,962$	$1,393,710$	0.42
5	$8,356,373$	$347,798$	$2,971,333$	$3,319,131$	0.40

The $O(n)$ complexity seems to hold up. A side note is that the ratio between n and the total amount of objects seems to be dropping as the file size increases. This is due to the ratio of u to n also dropping as n increases.

CHAPTER 5
Index 4

There are still some very big issues with the nested linked list solution, which we saw in the testing of the previous chapter. The most glaring one is that preprocessing took $O(u \cdot n)$ time which turned out to be way too much, making it impossible to load big files.

The second issue, which is not as visible in the tests, is that having a look up of $O(u + d_x)$ is actually not that good, and can be done a lot quicker with a better data structure.

The fact that it is only the operations *insertion* and *look-up* that this data structure has to support, means that we can simply choose a data structure which performs well on those two operations.

I will now show how and why chained hashing can be used for such an objective, which can be verified in the book *Introduction to Algorithms*, [Cor+09, chapter 11]. To do so we will first look at direct addressing.

5.1 Direct Addressing

A data structure that has constant insertion and look-up is *direct addressing*.

Give each data point, x, a correlated key, k, where k is always a positive integer, such that they make the pair (x, k). Say if x is a word, then its key could be the integer equal to the product of each letters ASCII code. Because a word can potentially be infinitely long or the empty string, the range of possibilities for k is the set $\{0, 1, ..., \infty\}$, which is the universe, U, of all possible keys for this example. If the keys are stored as integers the upper bound on U is $2^{32} - 1$.

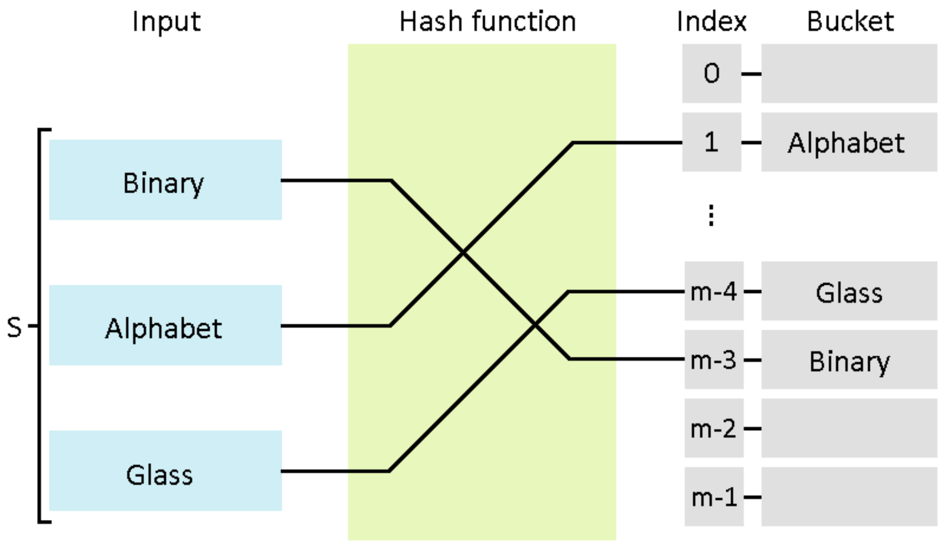
Direct addressing stores a set of data points, S, in a simple table, T, using k as indices. If S is a set of n words, then T would have to be the same size as U of the keys, which potentially is a lot bigger than n, leaving a lot of wasted space.

When this is the case, that $|U| \gg n$, a deviation of direct addressing called *hash tables* are very efficient.

5.2 Hash Tables

For hash tables it is assumed that $|U| \gg n$, or else direct addressing would be sufficient. Instead of simply using k as indices in T, a hash function, h, is used to transform k into a different number, called the *hash value*. The hash function is chosen such that any k in U is transformed into a different set B. If $h(k)$ is used as indices in T instead of simply k, then the size of T only has to be equal to B instead of U. We will denote the size of T as $|T| = m$. Now it is only a matter of choosing h and m such that $|B| \geq n$.

Figure 5.1: Graphical explanation of a Hash Table.



An easy way of restricting the size of B is by using a modulo operation as the last operation in the hash function. If the amount of input is known then n can be used for the modulo operation guaranteeing that every element in S has its own spot in T, by $|B| = n$.

If n is not known then the idea of *table doubling* can be used. The size of the table, m, is chosen to be some constant c at initialisation. When the table hits its limit a new table is constructed with the size $m \cdot 2$, and all of the previous data points will be moved to the new table. Now there is a table half full, with the size $c \cdot 2$. The next time the doubling happens the table will be of size $c \cdot 4$.

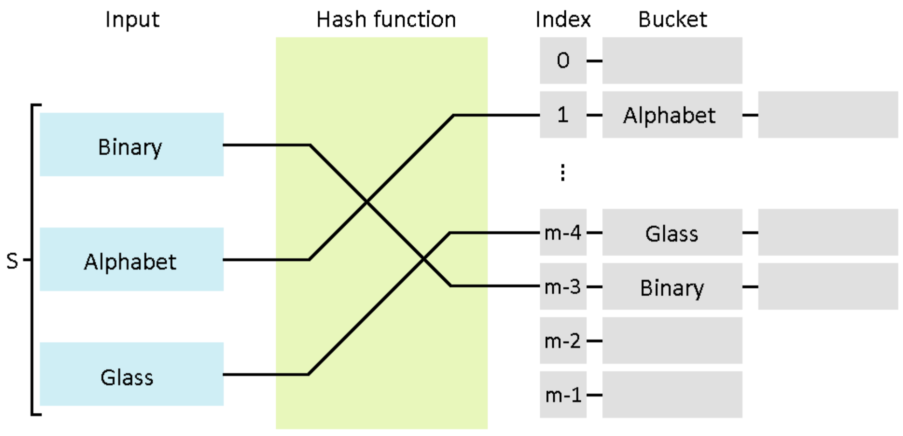
Whenever the doubling happens the hash function has to be changed such that $|B|$ is equal to the size of the table, or the new indices of the table will never be used. Since the hash function has changed all of the keys have to be rehashed into the new table. This leads to that $m = |B| \geq n$ at all times.

5.3 Chained Hashing

As stated earlier the idea of a hash function is to transform the set of possible keys U into a smaller set B such that we do not need as big of a table to store them, $|U| > |B|$. By the pigeon hole principle, there has to be at least one pair of possible keys (k_1, k_2) such that $h(k_1) = h(k_2)$ and therefore hash into the same index in T. This is called a collision, and there are several ways to handle them. I have chosen to use the solution *chained hashing*.

The space allocated for a given index is called that index's bucket. In hash tables without chained hashing, such a bucket simply contains either **null** or a single element of the value that the table is suppose to store. For chained hashing such a bucket contains **null** or the head of a linked list of elements. Whenever a key is hashed to a bucket which already contains an element, the new element is placed in front as the new head of the linked list. This allows collisions to happen naturally.

Figure 5.2: Graphical explanation of chained hashing.

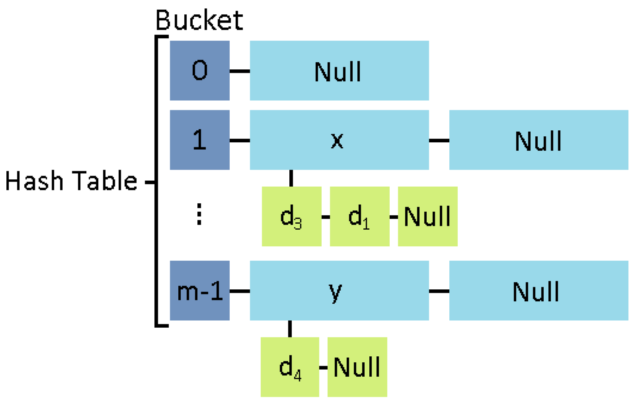


It is still possible to use table doubling along with chained hashing, but there is no longer a maximum amount of entries that the table can holster. Instead we use the notion of a *load factor*, $\alpha = \frac{n}{m}$. A load factor of 1 then means you have inserted the same amount of elements as there are indexes in the table, so to imitate table doubling on simple tables, a chained hash table should be doubled when $\alpha \geq 1$. Since the probability of collisions happening increase the closer α is to 1, it may be beneficial to do the table doubling earlier. We will look further into this in the testing section.

5.4 Description

In general we want to use the same system of linked lists as in the previous version, but splitting the L_w linked list into several lists, one for each bucket in a chained hash table. Each L_w then contain the words that hash to the given bucket. Every element in any L_w still has its own L_d list, which contains the list of documents the word appears in.

Figure 5.3: Data structure of Index 4.



This means that, like the L_w list from Index 3, the hash table only contains the subset $S' \subseteq S$, that consist of every unique word in S. The load factor is the size of S' divided by the size of the table, $\alpha = \frac{|S'|}{|T|}$. The size of S' is equal to the amount of unique words, u.

$$\alpha = \frac{u}{m}$$

Since there is no knowledge of how many words we will insert into the hash table, the notion of table doubling previously mentioned will be used. If the set S inserted into the table is distributed evenly into all buckets by the hash function, then a table doubling when the load factor α is 1, would mean the L_w lists never exceed a size of 1, giving a constant look-up and insertion run time.

That said, hash functions are never that precise and collisions will happen, but a load factor of 1 means that the pigeon hole principle will never force two elements to be in the same bucket and is the smallest α to do so. As such it lets us waste the least amount of space while still giving the desired property, and therefore increasing the probability of having a good distribution in the table.

The hash code that will be used for the implementation is the inbuilt Java hash code for strings. It works by taking the bit representation for the every character in the string as a integer, to the power of the index it is located at and sums all of the results together. [1]

$$h(x) = \sum_{i=0}^{d-1} x[i] \cdot 31^{d-1-i}$$

Where x is the string and d is the length of the string.

In order to have it fit into the hash table a modulo operation with the size of the table is done.

$$\text{bucket index for } x = h(x) \ \% \ m$$

5.4.1 Preprocess

The initial table starts with a hash table of the size 128, with every bucket empty. When inserting a word, x, first the correct bucket is found using Java's hash code for strings, then the L_w is handled the same way it was done in Index 3.

If x already is in L_w then ensure the current document is in $L_d(x)$; if it is not, then add x to L_w, and create a new L_d for it.

Before inserting x a check on α is done. If $\alpha \geq 1$ then table doubling is performed on the hash table. The initial size of the table is 128 buckets, so the table size can be 2^z where $z \in \{7, \dots, 30\}$ due to limits for tables in Java. The doubling is done by creating a new table of the size 2^{z+1} and changing the modulo operation of the hash function to fit the new size. Then all of the entries in the previous table are re-hashed into the new table. No new objects need be created as the previous ones can be used. The previous hash table is made eligible for Java's garbage collection by overriding the pointer to the hash table.

5.4.2 Search

One of the features that lets hash functions be useful is that they are consistent. This means that when we have inserted word x into a bucket, using the hash function, we can find that bucket again by running the hash function on x.

This functionality we use for searching, whereafter we only need to go through L_w and return the entirety of $L_d(x)$.

If we go through L_w and do not find x, then that means x is not in the input set.

5.5 Analysis

Chained hash tables are very efficient tools when used properly, but comparatively their worst case situations are appalling. If every key hash to the same bucket then

[1] Oracle's Java 8 Soruce code - https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

we have the exact same situation as in Index 3, which is one long linked list. If this happens none of the useful attributes that using a hash table provides are sustained.

Hash tables are still a very elegant solution and are used extensively in many fields of computer science. This is due to having very good average run times on the three dictionary actions, *insert*, *delete* and *search*. We are only interested in *insert* and *search*, therefore *delete* will be ignored, but if desired it could easily be implemented, with good results.

For this analysis we assume that the hash function distributes the universe of keys, U, uniformly over the value-set B. This means that when inserting x all buckets have an even chance of being chosen.

The hash function's run time is linearly dependent on how long the string is, since it does one operation per character. Since this is a analysis based upon the average we can say that there is some average amount of characters that words consist of, which we could use, making it a constant. Therefore the run time of the hash function can be said to be $O(1)$ and this we will use throughout the analysis.

5.5.1 Preprocess

The total time of a single insertion of x is equal to however long it takes to find the correct bucket plus the time spent figuring out how the new information has to be stored in the linked list system within the bucket.

Finding the correct bucket is done by taking $h(x)$ which is done in constant time, and the bucket is found by direct addressing in the table, which also is constant time.

Using the assumption that the hash function distributes the keys uniformly, the expected amount of unique words in a bucket is equal to the load factor. If this is the first time x is read, then x is not in L_w and so it takes $\alpha + 1$ to search L_w and insert it. If x is found in L_w, then we can again use that we only need to check the head of L_d to see if the current document needs to be added. Therefore the difference is of constant time, and they have the same run time complexity.

$$T(1) = O(\alpha + 1)$$

Due to the table doubling, α is always less than one, so the complexity of insertion is actually constant.

$$T(1) = O(1)$$

So inserting any word takes on average constant time, and thus inserting the whole set S, with n words will take

$$T(n) = O(n)$$

5.5.2 Table Doubling

The only thing that makes a hash table different from a normal table is that a hash function is used to find the index that entries are inserted into. That said, chained hashing does make it different, in that there are multiple entries in a single index. Luckily this analysis only cares about the amount of objects in the table and not that they are in different buckets. Consequently the run time analysis of table doubling on normal tables can be applied to our hashing tables.

If we imagine a normal doubling table that starts at the size 1, and say that inserting a number into an index has a cost of 1, then the following table can be made.

n'th element	1	2	3	4	5	6	7	8	9	...
Table size	1	2	4	4	8	8	8	8	16	...
Cost	1	2	3	1	5	1	1	1	9	...

Here we see that inserting when there is no need of doubling costs 1. When inserting in an already full table, with size z, table doubling needs to be performed. Creating a new table takes constant time, but moving the elements from the previous table takes one insertion for each element. Thus the insertion on a full table costs $z + 1$.

The amortised cost of inserting the set S, with n elements, in the doubling table is equal to the sum of all the costs up to n, divided by n.

$$\text{Amortised cost} = \sum_{i=0}^{n} \frac{cost(i)}{n} = \frac{1+2+3+1+5+1+1+1+9+...}{n}$$

Here we can split the costs of doubling into the $z + 1$ we found earlier, and split the z form the rest of the summation,.

$$\begin{aligned}\text{Amortised cost} &= \frac{\sum_0^n 1 + \sum_{i=0}^{\lfloor Log_2(n-1)\rfloor+1} 2^i}{n} \\ &= \sum_{i=0}^{\lfloor Log_2(n-1)\rfloor+1} 2^i \\ &\leq O(n)\end{aligned}$$

So the total running time of n insertions is $O(n)$, and thereby each insertion is constant. From this we can conclude that using the doubling table technique we do not change the complexity of the run time.

5.5.3 Search

There are three parts to searching for word x in the data structure. First the correct bucket has to be found using the hash function, which takes constant time. Then the L_w list has to be traversed to find the search term, which we also argued to be constant. If x is in the L_w, then $L_d(x)$ has to be returned as an answer, which means traversing $L_d(x)$ of length d_x.

Thus a complete search takes

$$T(n) = 1 + 1 + d_x = O(1 + d_x)$$

We keep the constant 1 due to if x is not in the data structure then d_x is 0 , but the constant operations of the hash function and checking L_w still needs to be performed.

5.5.4 Space Usage

All of the same linked list objects from Index 3 are still used in Index 4, even though they are split up into different buckets. So the only memory usage difference is in the hash table itself which is of size m. Right after a doubling of the table occurs the size of the table is two times the amount of unique words, and this is when the table is at it biggest compared to the amount of unique words. For the complexity we do not care about the coefficient so

$$m = O(u)$$

Adding this to the complexity of the amount of linked list objects does not change the complexity because $u \leq n$, so

$$n + u \leq n + n = 2n$$

And we simply remove the coefficient.

Thus the complexity of the memory usage is still

$$Space(n) = O(n)$$

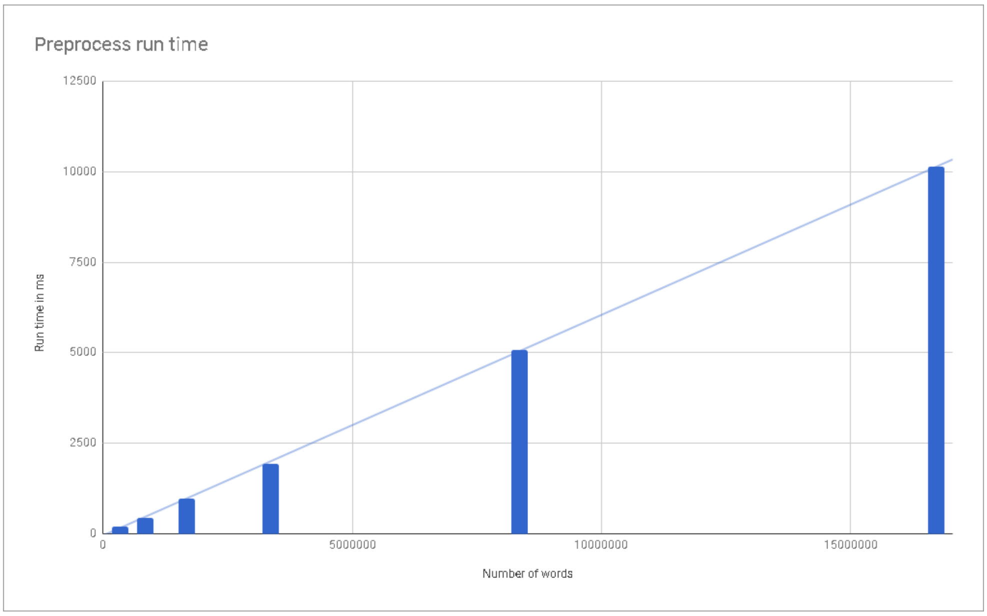
5.6 Test

Getting constant look-up and insertion run time is a huge improvement compared to the previous version, and we expect that to be reflected in the testing. In the analysis we used the assumption that the hash function distributes the keys with an uniform probability, but we do not have any proof of that. If it turns out that the tests follow the predictions of the analysis then, that does not necessarily mean that the assumption is correct. However it would mean that Java's hash function has the same scalability as a hash function with uniform distribution, but only on this specific data set.

5.6.1 Preprocess

Remember that if every key hashed to the same bucket then it would be the same solution as Index 3, so the worse the hashing is the closer the run time would be to $O(u \cdot n)$.

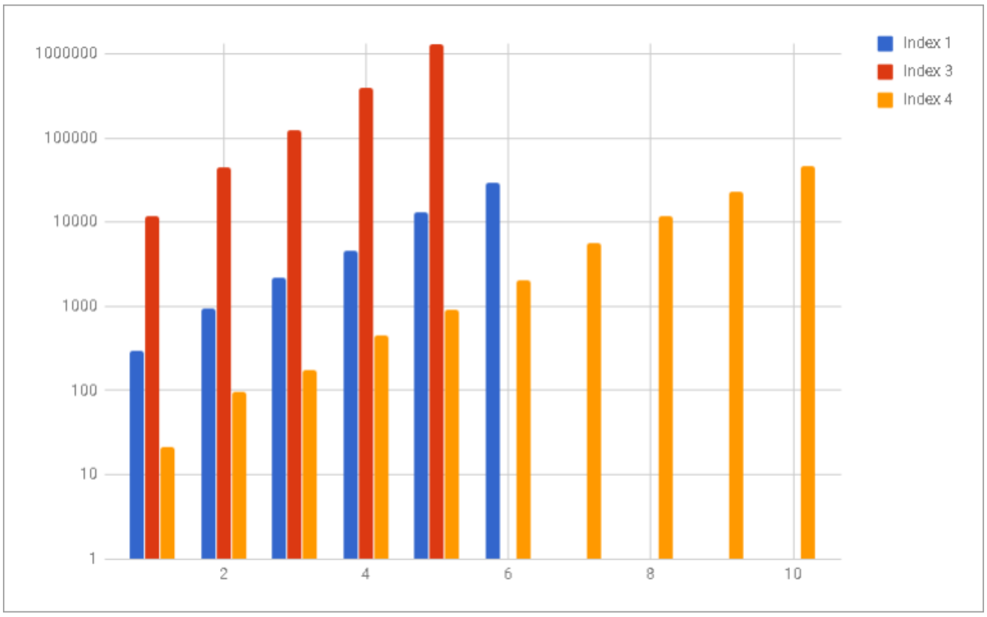
Figure 5.4: Index 4 preprocess run time.



As we can see the insertion does seem to be constant. This does not necessarily mean that the assumption of the hash function distributing the keys uniformly is correct, but it does indicate that the hash function spreads them out fairly well.

Compared to the previous versions Index 4 is sizeably better, and there are even bigger files that the previous versions could not handle, that Index 4 can.

Figure 5.5: Preprocess run time comparison.



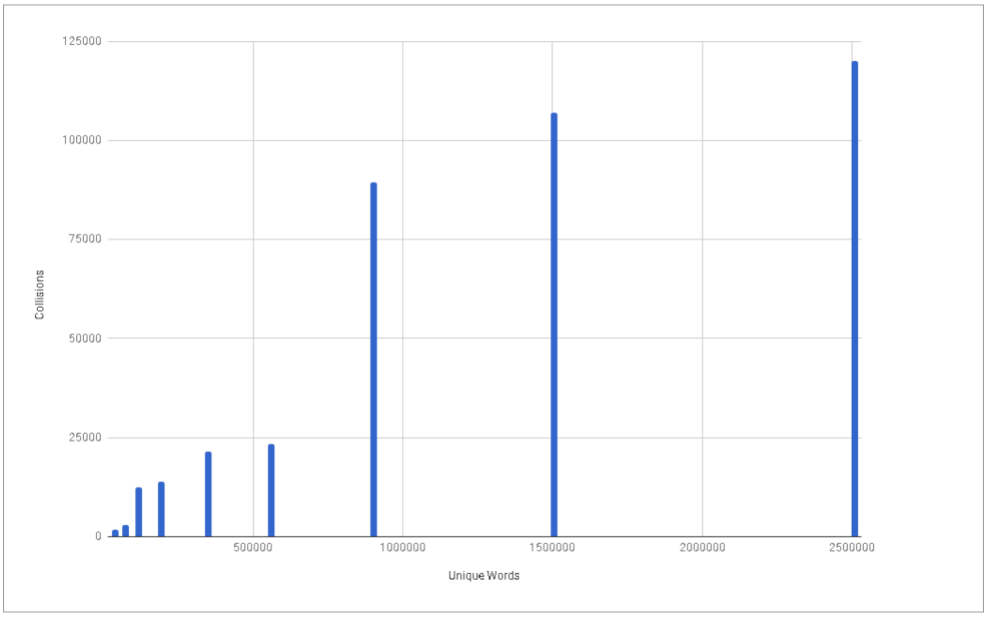
This graph uses logarithmic scale, so the differences are larger than they appear in the graph.

5.6.2 Collisions

As with everything else we want to be aware of how well each part of the algorithm is doing, and this also includes the hash function. In order to do that we can count the amount of collisions that happen in the final table of a preprocess phase.

A collision occurs whenever two separate keys x and y hash to the same bucket in the table. So a collision is when $h(x) = h(y), y \neq x$, and thereby the amount of collisions can be calculated by adding the binomial coefficient of every bucket with two or more objects, $\sum_{i=0}^{m-1} \binom{V_w}{2}, V_w \geq 2$ where W_v is the length of the L_w linked list in bucket i.

Figure 5.6: Number of Collisions compared to input of unique words.



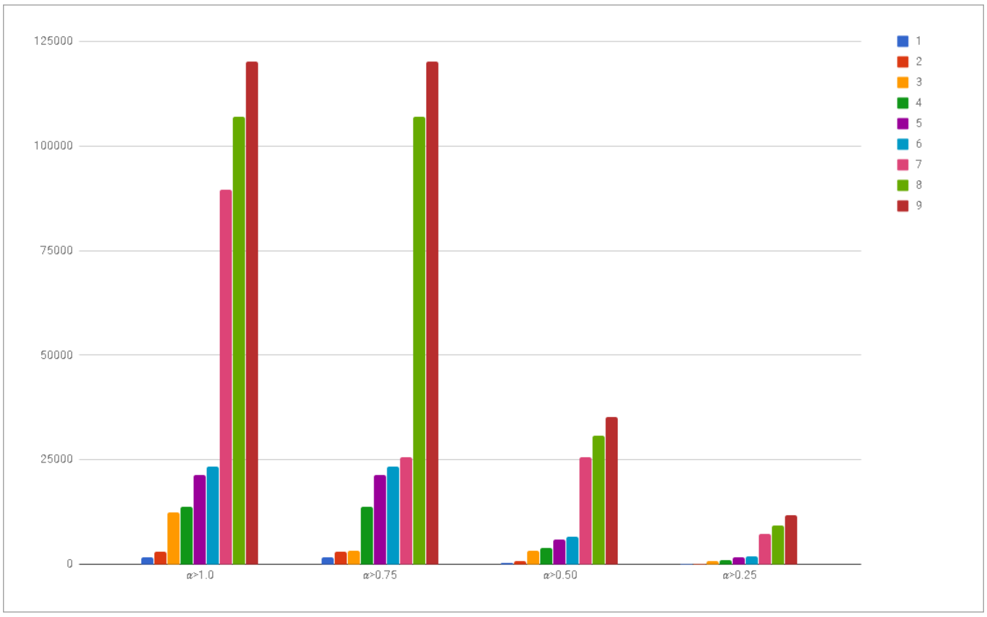
file	#objects in T	m	α	Collisions
1	37,232	65,536	0.57	1,704
2	71,494	131,072	0.54	2,994
3	116,600	131,072	**0.89**	12,403
4	190,748	262,144	0.72	13,876
5	347,798	524,288	0.66	21,362
6	560,184	1,048,576	0.53	23,375
7	900,298	1,048,576	**0,85**	89,478
8	1,503,743	2,097,152	0.70	106,958
9	2,507,802	4,194,304	0.57	120,138

Here we can see that the amount of collisions hugely depends on if the table has doubled recently. While the size of the input set S does double for each file, the amount of unique words does not necessarily. Thus it is possible that the hash table handling those two files have the same size, and the one with a bigger input is going to have a lot more collisions.

To reduce the amount of collisions one could change at what ratio the table doubles, to say $\alpha < 0.75$. The downside of decreasing the table doubling threshold is that table doubling does consume a substantial amount of time, but having to do less traversal through buckets might be worth it.

To check this I have tested the collisions and preprocess run time of this implementation with $\alpha = 0.25, 0.50, 0.75$ and compare them together with the $\alpha = 1.0$ from previous.

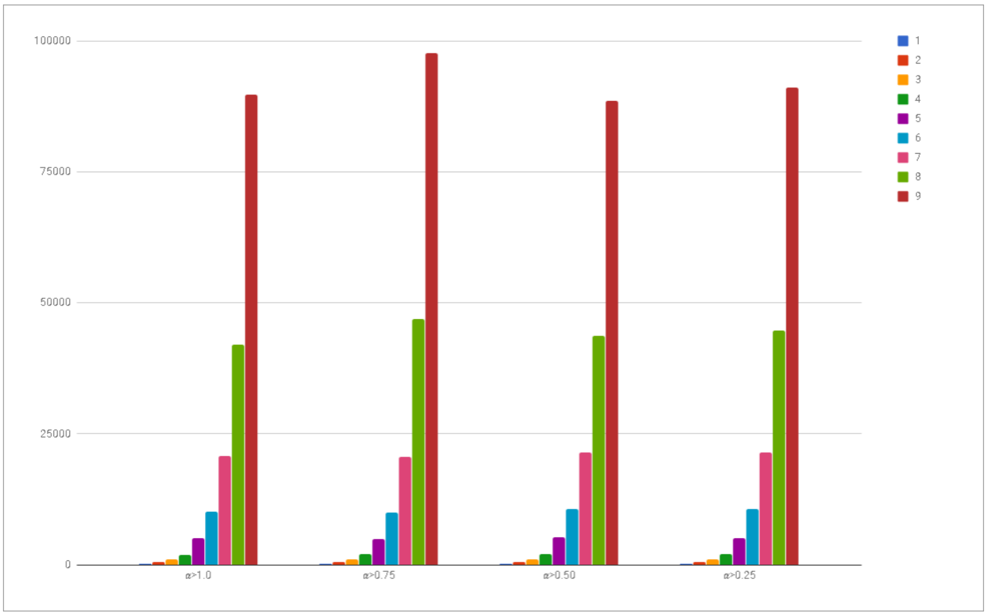
Figure 5.7: Collision comparison between different load factor thresholds.



The collision rate goes down hugely when doubling at $\alpha > 0.50$, but at the same time the amount of space used by the hash table is about doubled as well.

A more interesting question is if the preprocess run time changes in a positive or negative fashion.

Figure 5.8: .



Both $\alpha > 1.0$ and $\alpha > 0.50$ have about the same run time. The $\alpha > 1.0$ solution takes up less space, but the other solution gives us less collisions which will help the run time of searching, given that the size of buckets will be smaller.

As will be shown in the next section the searching is already very quick, so the reduction in memory usage is more valuable, and I have chosen to use the $\alpha > 1.0$ solution

5.6.3 Search

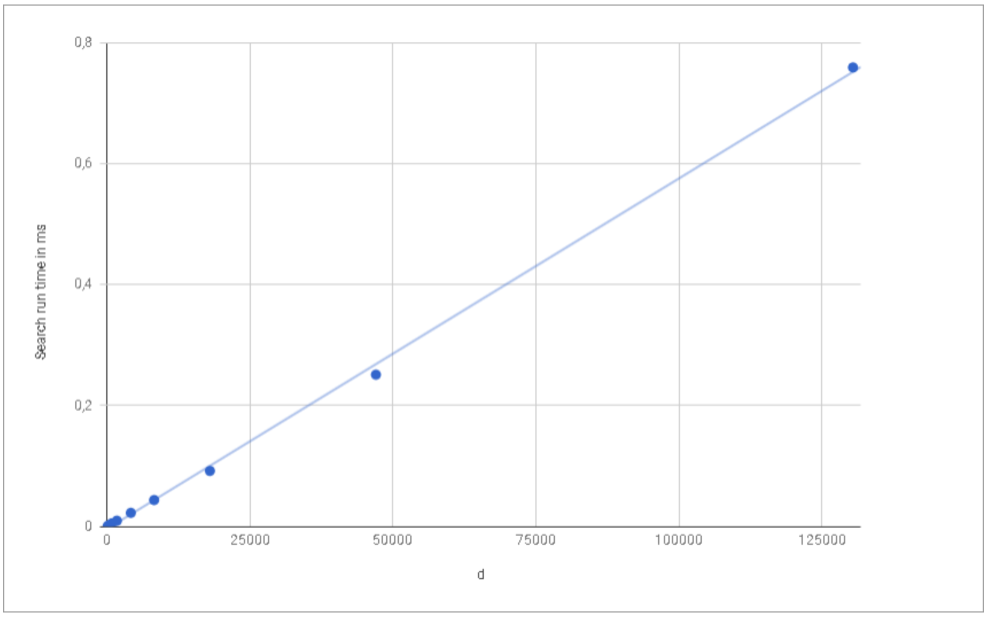
The look-up operation is the operation that has had the most change. If the same test for worst case situation was used here, by searching for " %&/#%&(% " then the hash function will pick out a bucket that the string hashes to, which potentially is empty. Additionally the dominating factor in the look-up was analysed to be d, which for strings that are not in the data structure is 0.

Instead the worst case situation is now to search for a string that appears in every document, for example the most used English word "the". With very high probability "the" will be used in every document and thus will have a run time of $\Theta(d)$.

To show this the search was tested with searching the string "the", and plotted against d, and it is shown that it is indeed linear.

Index 4 has an even quicker look-up than Index 3, so the same 'boosting' trick is needed in order to capture how long the search takes. This time each look-up is performed 10000 times.

Figure 5.9: Preprocess run time comparison.



Comparing the worst case situations of the different versions in the following table.

file	Index 1	Index 3	Index 4
1	5	0.21	0.0007
2	25	0.46	0.0021
3	17	1.02	0.0051
4	23	1.47	0.0095
5	56	2.86	0.0225
6	111	–	0.0437
7	–	–	0.0919
8	–	–	0.2511
9	–	–	0.7593

5.6.4 Space Usage

Using the same method as for Index 3 we can count the linked list objects and size of the hash table and see that it is less than the size of the input. The ratio column shows the ratio between the amount of objects and the input size n.

File	n	V_w	V_d	m	Total	Ratio
1	$329,292$	$37,232$	$113,307$	$65,536$	$216,075$	0.66
2	$828,068$	$71,494$	$289,279$	$131,072$	$491,845$	0.59
3	$1,672,002$	$116,600$	$600,650$	$131,072$	$848,322$	0.51
4	$3,348,766$	$190,748$	$1,202,962$	$262,144$	$1,655,854$	0.49
5	$8,356,373$	$347,798$	$2,971,333$	$524,288$	$3,843,419$	0.46
6	$16,717,033$	$560,184$	$5,932,491$	$1,048,576$	$7,541,251$	0.45
7	$33,442,256$	$900,298$	$11,883,285$	$1,048,576$	$13,832,159$	0.41
8	$67,263,274$	$1,503,743$	$25,077,735$	$2,097,152$	$28,678,630$	0.43
9	$135,750,192$	$2,507,802$	$53,883,413$	$4,194,304$	$60,585,519$	0.45

Here we see that the majority of the memory usage is being allocated to DocItem objects in the L_d linked list.

CHAPTER 6

Index 5

Any deterministic hash function, such as Java's *hashCode* for strings, has a set S' of keys where every key will hash to the same bucket, meaning that only that single bucket is used, and we are back to the same solution as Index 3. An adversarial could halt the operation by giving a big set of such keys. To combat this some sort of randomisation is needed, however we still need the hash function to be consistent in the sense that the look-up that will be performed later needs to locate the correct bucket.

This is done by having a set of different hash functions, $\mathcal{H}$, where every hash function hashes from the same universe to the same set of values, $h \in \mathcal{H} : U \to B$. Whenever a hash table is created a h can be pick at random from $\mathcal{H}$, and now the adversary would not know what set of keys would hash to the same bucket.

In order to ensure that these families provide useful hash functions, the conditions of *Universal Hashing* and *Strong Universal Hashing* were introduced by J. Carter and M. Wegman, [CW79] [WC81].

6.1 Universal Hashing

A family of hash functions $\mathcal{H}$ is universal if when h is picked at random, there is a low probability of collision between any two distinct keys $x, y \in U$.

$$P[h(x) = h(y)] \leq \frac{1}{m} \quad , h \in \mathcal{H}$$

A relaxed version is called c-universal, which means that for a constant c the probability is

$$P[h(x) = h(y)] \leq \frac{c}{m} \quad , h \in \mathcal{H}$$

A simple example of a universal hashing scheme is the *multiply-mod-prime* family, $\mathcal{H}_1$, found by Carter and Wegman in [CW79], where $h_1 : U \to [m]$. Pick a prime number $p \geq |U|$ along with uniform random $a \in [p_+] = \{1, \ldots, p-1\}$ and $b \in [p] = \{0, \ldots, p-1\}$ then the hash function is defined by

$$h_1(x) = ((a \cdot x) + b) \mod p) \mod m,$$

The numbers a, b are known as the *seeds* of the hash function, since they provide the necessary randomness that is required.

In lecture notes about universal hashing by M. Thorup,[Tho15], he explains how Carter and Wegman showed $\mathcal{H}_1$ to be universal by proving that the pair $(a, b) \in [p]^2$ has a $1-1$ correspondence with the pair $(q, r) \in [p]^2$ from

$$(a \cdot x) + b = q \tag{6.1}$$

$$(a \cdot y) + b = r \tag{6.2}$$

The $1-1$ correspondence also means that there will only be one a which satisfies the equation of subtracting (6.1) from (6.2) modulo p.

$$(a \cdot y + b) - (a \cdot x + b) \equiv a(y - x) \equiv r - q \mod p \tag{6.3}$$

If there was a different a' which fulfil (6.3), then

$$((a - a') \cdot x + b) - ((a - a') \cdot x + b) \equiv (a - a')(y - x) \equiv 0 \mod p \tag{6.4}$$

Here we use a fact about primes, which is

$$j \cdot k \not\equiv 0 \mod p \quad , \quad j, k \in [p_+] \tag{6.5}$$

Both $(a - a')$ and $(y - z)$ are both not zero modulo p, this means that (6.4) contradicts the rule just stated, and a' cannot exist. Having b satisfy (6.1) sets b as

$$b = (q - a \cdot x) \mod p$$

Therefore there is only one pair of (a, b) that satisfy (6.1) and (6.2) for each (q, r) pair and likewise there is only one pair of (q, r) for each (a, b) pair. Both pairs are within the set of $[p]^2$, so a $1-1$ correspondence is unavoidable.

Lets look at the situation of $r = q$, which would give

$$\begin{aligned} (a \cdot y + b) - (a \cdot x + b) \mod p = 0 \Rightarrow \\ a(y - x) \mod p = 0 \end{aligned}$$

Since $x \neq y$, we know from (6.5) that a would have to be 0, but in the definition of a we set it to be from $[p_+]$ which does not contain 0, so $r \neq q$.

A collision only happens when $q \equiv r \mod m$. When given $r \in [p]$ there are $\lceil p/m \rceil$ ways of satisfying $q \equiv r \mod m$, but since $q \neq r$ there are actually $\lceil p/m \rceil - 1$.

$$\lceil p/m \rceil - 1 \leq \frac{p + m - 1}{m - 1} = \frac{(p-1)}{m}$$

When given r there are $\frac{(p-1)}{m}$ ways of picking q such that they collide. Since $r \in [p]$ there are at most $\frac{p(p-1)}{m}$ collision pairs of (q, r). Due to the $1-1$ correspondence there are an equal amount of (a, b) collision pairs and since each (a, b) pair is equally likely, the collision probability is bounded by $1/m$.

6.2 Strongly Universal Hashing

For strong universality we regard the pair-wise events of two distinct keys $x, y \in U$ to be hashed into the possibly equivalent values $q, r \in [m]$, written as $h(x) = q \wedge h(y) = r$.

A random hash function h is strongly universal if

$$P[h(x) = q \wedge h(y) = r] = \frac{1}{m^2}$$

A strongly universal hash function is also universal, as seen from the following

$$P[h(x) = h(y)] = \sum_{q \in [m]} P[h(x) = q \wedge h(y) = q] = \frac{m}{m^2} = \frac{1}{m}$$

We will later show that for chained hash tables c-universality is enough to get constant look-up and insertion operations, but seeing that strongly universal hash functions also are universal means they also give this property.

An interesting feature of strongly universal hash functions is that they hash uniformly into $[m]$.

An example of such a hash function is the *multiply-shift* family, $\mathcal{H}_2$, originally proposed by Dietzfelbinger, [Die96], and explained in M. Thorup's paper, [Tho15]. For hash function $h_2 : [2^w] \to [2^l]$ a $\overline{w} \geq w + l$ is picked and random seeds $a, b \in [\overline{w}]$ are used.

$$h_2(x) = \left\lfloor (a \cdot x + b \mod 2^{\overline{w}}) / 2^{\overline{w} - l} \right\rfloor$$

The $\mathcal{H}_2$ family is very easy for computers to compute due to innate features of computers. The first is that z bit multiplication automatically discards overflow, which can be used as a free modulo 2^z multiplication. Additionally, due to the nature of bits a division of 2 can be performed as a right shift.

These two aspects can be utilised for h_2 to get a very quick hash function. If we have a U consisting of 32 bit integers and want it to hash into 20 bit integers, then $\overline{w}$ can be set to $\overline{w} = 64 \geq 32 + 20$, thus having the multiplications in a 64 bit scope, where the overflow will act as modulo 2^{64}. Within this scope the hash function is thus equivalent to

$$h_2(x) = \left\lfloor (a \cdot x + b \mod 2^{\overline{w}}) / 2^{\overline{w} - l} \right\rfloor \equiv (a \cdot x + b) >> \overline{w} - l$$

6.3 Usage of Mersenne Primes

Mersenne primes are primes that follows the formula of $p = 2^q - 1$. These primes are useful choices for hash functions because they provide easy modulo calculations.

$$x \mod p \equiv x \bmod 2^q + \lfloor x / 2^q \rfloor$$

The point is that this can be done by logical bit operations as

$$x \mod p \equiv (x \;\&\; 2^q - 1) + (x >> q)$$

With the exception of when $x \mod p = 0$, then the above bit operations will return p as result. A simple check is sufficient to fix it.

6.4 Hashing on Strings

In M. Thorup's notes, [Tho15], he explains two ways of having universal hashing of strings, one for variable length strings, $\mathcal{H}_3$ and the other for bounded length, $\mathcal{H}_4$.

6.4.1 Hashing of variable length strings

The idea is to use the hashing scheme from $\mathcal{H}_1$, but change x such that it handles strings instead of integers.

This is done by using the bit string of the characters from the key as coefficients for a polynomial Q along with a random seed $c \in [p]$. For key $x = [x_0, x_1, \ldots, x_d]$ the polynomial will be of degree d, and will look like

$$Q_{x_0,\ldots,x_d} = \sum_{i=0}^{d} x_i \cdot c^i \mod p$$

The polynomial Q is then used as a sub hash function $h_s : String \to R$, for transforming the string into a number. For h_s the probability of collision between string x and $y = [y_0, y_1, \ldots, y_d']$ where $d \geq d'$ is at most d/p.

$$P[h_s(x) = h_s(y)] \leq d/p$$

We can be prove this by looking at the *root* of the polynomial $Q_{xy} = Q_y - Q_x$. The roots of a polynomial is the set of numbers such that the polynomial is equal to zero and in any polynomial the amount of roots is at most equal to the degree of the polynomial.

A collision only happens if the seed c is a root of the polynomial Q_{xy}. This polynomial is not the constant zero polynomial since the strings are distinct. Q_{xy} is of degree d, so there are at most d roots, and the probability of c being one of those roots is d/p.

If we want h_s to be universal then m and p can be picked such that $d/p \leq 1/m$. If we for example use $p = 2^{61} - 1$ and $m = 2^{31} - 1$ then d can be up to 2^{30}, which should be plenty.

Thus the full hash family $\mathcal{H}_3$ consists of hash functions $h_3(x_0, \ldots, x_d) : U \to [m]$ with prime $p \geq |U|$ and random seeds $a, b, c \in [p]$.

$$h_3(x_0,\dots,x_d) = \left(\left(a\left(\sum_{i=0}^{d} x_{d-i}\cdot c^i\right)+b\right) \mod p\right) \mod m$$

In h_3 we use the result of h_s as input for h_1, so for there to be a collision in h_3 there simply has to be a collision in either h_s or h_1.

$$P[h_3(x) = h_3(y)] = P[h_s(x) = h_s(y)] + P[h_1(h_s(x)) = h_1(h_s(y))]$$

We know the collision probability of h_s and h_1, and therefore also h_3

$$P[h_3(x) = h_3(y)] = \frac{1}{m} + \frac{1}{m} = \frac{2}{m}$$

Which implies that h_3 is 2-universal if the length of the input key follows the criteria.

6.4.2 Hashing of bounded length strings

A different solution for converting a string to a hash-able integer is to have a random seed for each possible index in the string. If you know the maximum length of the strings, D, in U then a table of random seeds, $a = \{a_0, a_1, \dots, a_D - 1\}$, with that size can be made at initialisation.

The conversion of string $x = \{x_0, \dots, x_d - 1\}$ to integer is then done by multiplying each character in the string with the correlated a value, and summed together.

$$\sum_{i\in[d]} a_i \cdot x_i$$

To increase the speed of this summation the amount of multiplications can be halved by doing it in pairs.

$$\sum_{i\in[d/2]} (a_{2i} + x_{2i+1})(a_{2i+1} + x_{2i})$$

If the string x is of an odd length, then the above method can be used up until $d-1$ and the multiplication $a_d \cdot x_d$ can be added.

This method is then combined with the multiply-shift scheme, $\mathcal{H}_2$, to combine into a strongly universal hash function $h_4(x_0, \cdots, x_{d-1}) : [2^w]^d \to [2^l]$ with $\overline{w} \geq w + l - 1$ and random seeds $a = \{a_0, \dots, a_d - 1\} \in [2^{\overline{w}}]$

$$h_4(x_0,\dots,x_{d-1}) = \left\lfloor \left(\left(\left(\sum_{i\in[d/2]} (a_{2i} + x_{2i+1})(a_{2i+1} + x_{2i})\right) + a_d\right) \mod 2^{\overline{w}}\right) / 2^{\overline{w}-l} \right\rfloor$$

Concatenating Characters Depending on the bit size of the characters in x, it is possible to decrease the amount of multiplications even further.

The intention is to have an array, z, of 64 bit integers that can house the same amount of bits that the input key x can. A memory copy is then performed to insert the bits representing x into the array.

Since x is not necessarily of the maximum length, not every index in z will be used, and it is important to ensure that the last used 64-bit integer in z does not contain any bits from previous use.

h_4 can then be used on pairs from z, which with 8-bit characters results in changing from a maximum of $\frac{D}{2}$ multiplications to $\frac{D/8}{2}$ multiplications.

6.5 Description

We will be looking at three different string hashing implementations one of $\mathcal{H}_3$ and two of $\mathcal{H}_4$, one with the concatenation trick explained above and one without.

6.5.1 $\mathcal{H}_3$

First we need to ensure that h_s is universal, which means we need to ensure $d/p \leq 1/m$. Earlier it was shown that picking a Mersenne prime lets us do the modulo calculations in simple bit operations, and the biggest Mersenne prime that fits into a single Java variable is $p = 2^{61} - 1$ which can be stored in a *long*.

Since we use the doubling method for controlling the size of m, we need to ensure that the biggest possible size of m still lets us handle strings of a sufficient size. For the initial table m is of the size 2^l where $l = 7$, and whenever it is doubled l is incremented. In Java the biggest possible size of a table is $2^{31} - 1$, so m will never surpass that. Thus any hash table made this way can handle strings of 2^{30}.

This means that as long as the keys $x = \{x_0, \cdots, x_d - 1\}$ are shorter than 2^{30} we get $h_3 : [p] \rightarrow [2^l]$ to be 2-universal. With random seeds $a, b, c \in [p]$ the hash function is thus

$$h_3(x_0, \ldots, x_d) = \left(\left(a \left(\sum_{i=0}^{d} x_{d-i} \cdot c^i \right) + b \right) \mod 2^{61} - 1 \right) \mod 2^l$$

Whenever a table doubling is performed, a new hash table is created, and every previous entry has to be rehashed in the new table. This also means that a new h_3 hash function has to be created, along with new triple a, b, c of random seeds.

For computing polynomials Horner's rule can be used to calculating them quicker. This is done by reversing the order of coefficients, which reduces the amount of multiplications that are needed.

$$Q_{x_0,\dots,x_d} = \sum_{i=0}^{d} x_i \cdot c^i \equiv ((x_0 \cdot c + x_1)c + \dots)c + x_d \mod p$$
$$\equiv \sum_{i=0}^{d} x_{d-i} \cdot c^i \mod p$$

6.5.2 $\mathcal{H}_4$

Initially we have a big problem in that we do **not** know how the length of the keys that will be hashed. As a insurance policy $\mathcal{H}_3$ will be used as a substitute hash function for any key that is bigger than this implementation allows.

For h_4 to work a maximum string size has to be picked, of which I chose 256, a size that is big enough for any word. I later found out that the only strings in the text files that have a length of more than 256 are links.

Thereby we have an array $a = \{a_0, \cdots, a_{255}\}$ of random seeds $a_i \in [2^{\overline{w}}]$. Since $\overline{w} \geq w + l$ we need to know the size of the universe we are hashing,$U = [2^w]$, and the biggest table that can be used, $m \leq 2^l$, before deciding on a $\overline{w}$.

In Java Strings are stored using UTF-16, which uses anywhere between 2 and 4 bytes to represent symbols.[1] This means that the universe of keys is $w = [2^{32}]$. As for l, we know that the table can at most be of size $2^{31} - 1$, so $l \leq 31$. Therefore $\overline{w} = 64 \geq 31 + 32$ is picked in order to let the 64-bit multiplication overflow be used as modulo $\overline{w}$, as needed for $\mathcal{H}_2$.

The hash function $h_4(x_0, \cdots, x_{d-1} : [2^w]^2 \to [2^l]$ that we end up with have the following values, $\overline{w} = 64, w = 32, l \leq 2^{31} - 1$ and random seeds $a = \{a_0, \cdots, a_{256}\} \in [2^{\overline{w}}]$.

$$h_4(x_0, \dots, x_{d-1}) = \left\lfloor \left(\left(\left(\sum_{i \in [d/2]} (a_{2i} + x_{2i+1})(a_{2i+1} + x_{2i}) \right) + a_d \right) \mod 2^{64} \right) / 2^{64-l} \right\rfloor$$

For this to work in Java we need some way of having 64 bit unsigned integers, which is not standard in Java, but can be obtained through libraries such as Google's Guava, which is what I used.

6.5.3 Utilising Concatenated Characters, $\mathcal{H}_5$

As briefly mentioned Strings use UTF-16, which represents characters with 2-4 bytes depending on what symbol it is. The most general characters, such as the Latin

[1] Oracle source code - https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

alphabet, 0-9, and some other often used symbols, are represented with 8 bits. This means that we can use the concatenation scheme explained earlier, given that the string length, d, is equal to the amount of bytes used for that string. We can now have a fork in the hash function, having keys that do not follow the criteria for concatenating characters use the normal approach.

For those that do fulfil the criteria, we know that every character is 8 bits. With at most 256 characters the string can at most be a total of $256 \cdot 8 = 2048$ bits, which is equal to 32 64-bit integers, so the array needed to store the string is of size 32. We will call this array $b = \{b_0, \cdots, b_{31}\}$.

Unfortunately there is not any way of performing the memory copy command in Java, but instead it can be done manually. In a single 64-bit integer up to 8 8-bit characters can be stored. The manual memory copy is done by inserting the initial character in a long variable b_i, and then for each following character we bit-shift 8 times to the left and then perform a logical or-operation on b_i and the next character's binary representation, x_{bit}.

$$b_i = (b_i << 8) \mid x_{bit}$$

There is then $\lceil d/8 \rceil$ 64-bit integers with the last one potentially not full, that we can run h_4 on. This scheme decreases the amount of multiplications by up to a factor of 8, depending on how full the last b_i is and if the amount of b_i used is even or odd.

6.6 Analysis

We now have two hashing schemes that are strongly universal and one that is 2-universal, but we have yet to show that these qualities are important to chained hashing.

In the analysis for Index 4 we assumed that the hash function distributed the keys uniformly over the set B that it hashed into. In reality it is not the actual uniform distribution that we desire from a hash function, but rather insurance that it distributes the keys with as few collisions as possible, and that is exactly what a universal hash function does.

As argued in Index 4 the run time of performing a look-up operation in the data structure is equal to the expected size of the bucket you end up doing the look-up in. To show that the look-up and insertion operations run in constant time, it is needed to show that the expected length of any given bucket is constant.

Assume that the set S has been inserted into the data structure, and that key $x \notin S$. By definition the length of the bucket that x is hashed to is equal to the amount of collisions for x.

$$|L_w(h(x))| = \sum_{y \in S} h(x) = h(y)$$

As such the expect length of $L_w(h(x))$ is also equal to the expected amount of collisions.

$$E[|L_w(h(x))|] = E\left[\sum_{y\in S} h(x) = h(y)\right] = \sum_{y\in S} P[h(x) = h(y)]$$

From the hash function being universal we know that $P[h(x) = h(y)] \leq 1/m$.

$$E[|L_w(h(x))|] = \frac{1}{m} \cdot n = \frac{n}{m} = \alpha \leq 1$$

If $x \in S$, then we know that there is already at least 1 in $h(x)$, namely x, and the amount of different keys $y \in S$ is $n - 1$.

$$E[|L_w(h(x))|] = 1 + \frac{1}{m} \cdot (n-1) = 1 + \frac{n-1}{m} \leq 1 + \alpha \leq 2$$

Whether or not x is in S the expected length of $L_w(h(x))$ is constant, and therefore the expected run of both insertion and look-up is constant.

If the hash function is c-universal, then $P[h(x) = h(y)] \leq c/m$, which means that for $x \notin S$

$$E[|L_w(h(x))|] = \frac{c}{m} \cdot n = \frac{c \cdot n}{m} \leq c$$

and for $x \in S$

$$E[|L_w(h(x))|] = 1 + \frac{c}{m} \cdot (n-1) = 1 + \frac{c \cdot (n-1)}{m} \leq 1 + c$$

So to get constant look-up and insertion, we only need the hash function to be at least c-universal. The hash family $\mathcal{H}_3$ is 2-universal and $\mathcal{H}_4$ and $\mathcal{H}_5$ are both strongly universal, and therefore also provide the required property of universality to get constant operations.

We can also show that a hash function that perfectly uniformly distributes the keys over $[m]$, has the same expected amount of collisions as a universal hash function. Any key x has a $1/m$ chance of being hashed to a given bucket. Given that h hashes independently on each key, then the probability that two keys hash to the same bucket is

$$P[h(x) = j \wedge h(y) = j] = \frac{1}{m} \cdot \frac{1}{m} = \frac{1}{m^2}$$

And thus the probability that two keys collide is equal to those two keys hashing to the same bucket over all buckets.

$$P[h(x) = h(y)] = \sum_{j=0}^{m-1} P[h(x) = j \wedge h(y) = j] = \frac{1}{m^2} \cdot m = \frac{1}{m}$$

6.7 Testing

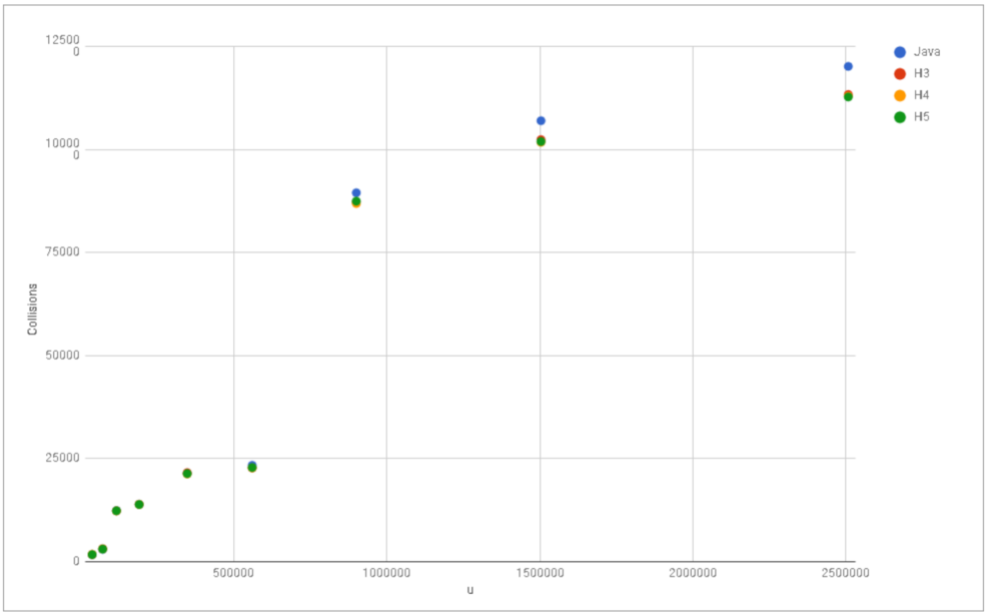
Due to the hash functions having random seeds sometimes, although very rare, the seeds are picked in such a fashion that the resulting hash function works very inefficiently on the specific test data. Due to the testing technique of choosing the median result from a series of runs this innate feature of random seeds does not impact the test data.

The only part that was changed for Index 5 is what hash function the hash table is using. As such the memory space used has not changed and we will not be testing for it.

6.7.1 Collisions

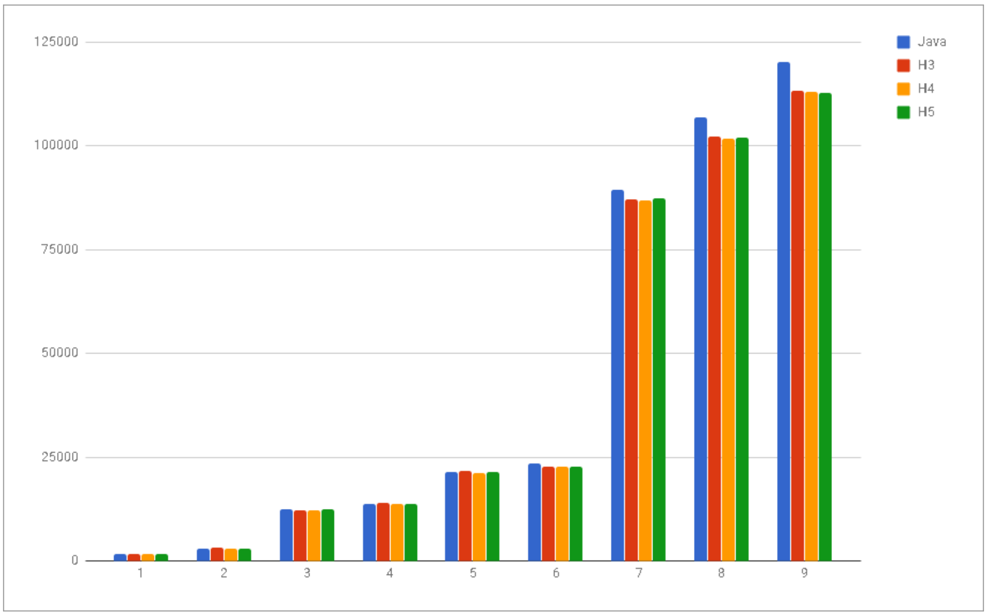
When testing for collisions on a deterministic hash function such as Java's the choice of input set S is going to have a very big impact on the results. Due to the nature of randomised hash families this is not a problem for our three other hash functions. However we are still interested in seeing how well the hash functions compare to each other.

Figure 6.1: Amount of collisions plottet against u.



And plottet against the file name instead of u to make it easier to see.

Figure 6.2: Amount of collisions for each file.



Interestingly it occurs that there are huge jumps in collisions from file 2 to 3 and 6 to 7. If we look at the size of the hash tables, these jumps align perfectly with whenever the size of that hash table does not change, while the size of the input obviously does change and therefore the load factor α is greatly increased.

File	α
1	0,57
2	0,54
3	**0,89**
4	0,72
5	0,66
6	0,53
7	**0,85**
8	0,70
9	0,57

The four hash functions are following each other rather nicely, except for Java's hashCode function being worse at big input sizes. While it seems to be handling the input set well, it is hard to argue from this data that it would handle a random input set S' well. This gives the randomised hash functions a big theoretical advantage.

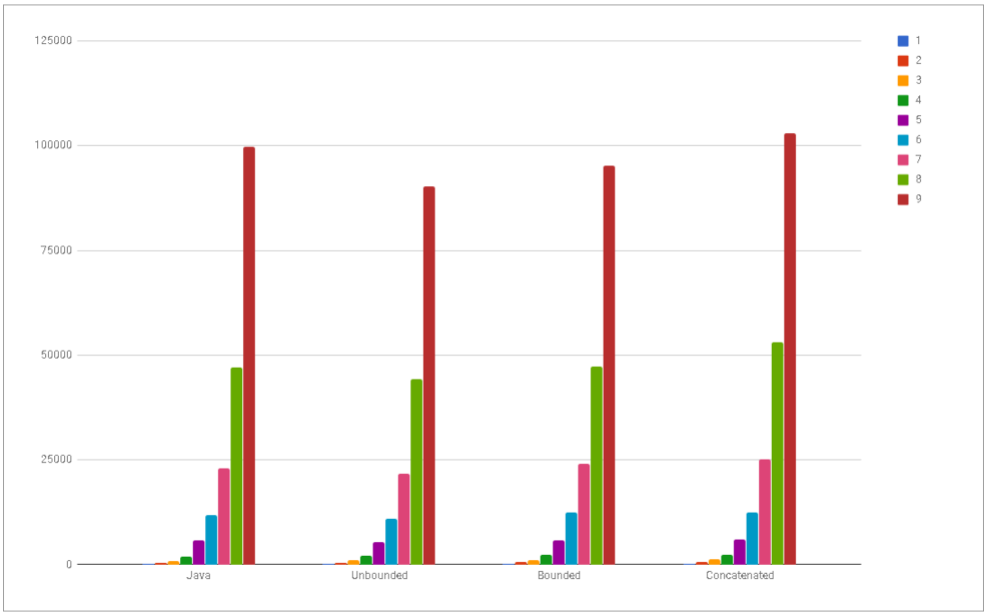
The three new hash functions all hash with almost exact equal amount of collisions. This is not the expect result, as $\mathcal{H}_3$ is 2-universal and the other two are 1-universal.

However the universality analysis only gives an upper bound, and thus this result is still within the bounds of the analysis. This result might be a consequence of the specific data input used, and using a different input might yield different results. Figuring out why $\mathcal{H}_3$ hashed with the same amount of collisions could be an interesting addition to the thesis, but I chose not to go deeper into this avenue.

6.7.2 Preprocess

Since the insert operation runs in constant time, the run time of the entire preprocess is linearly dependent on the size of the input set. Therefore the difference between the hash functions is going to be dependent on the computation time of the hash value. For example, $\mathcal{H}_5$ has fewer multiplications than $\mathcal{H}_4$ so we would expect it spends less time computing the hash value. However it turns out that it does not.

Figure 6.3: Preprocess run time on file i, with the four hash functions.



This is due to the manual memory copy having to insert 8 bits at a time, resulting in a lot of overhead, making the operation very inefficient, so much so that simply doing the multiplications is better. If the program was implemented in a language with the *memcpy* operation this scheme would have worked a lot better.

Interestingly Java's hash function does compare rather well to the randomised hash functions on these files, even being faster than the $\mathcal{H}_5$ implementation. It only loses out to $\mathcal{H}_3$ on the three largest files, which are also the ones where it had more collisions. Both Java's hash function and $\mathcal{H}_3$ uses Horner's rule to calculate the input key as a polynomial, so their implementation is rather close, with the exception that $\mathcal{H}_3$ uses modulo prime and random seeds to make it 2-universal. As such it makes sense that on the files they have equal number of collisions they also have equal run time.

Even though Java's hash function has no promises of having low collision probability, like a universal hash function has, it still performs rather well on the input set, both in the collision and run time area.

6.7.3 Search

The search is performed in the same way as in index 4, with searching for the string '*the*' and plottet against d.

Not too surpricing $\mathcal{H}_3$ is the quickest, but surprisingly $\mathcal{H}_5$ is second, although not by much. All four implementations follow the expected linearity with the amount of documents d.

CHAPTER 7
Index 6

The implementation of the search engine in Index 5 still cannot handle the biggest test file, which contains the entirety of Wikipedia. When the Java Virtual Machine reaches a memory usage close to the maximum allowed capacity it tries to execute the garbage collection more often in order to escape the *out of memory* error. To prevent this we need some way of decreasing the memory usage. We will now look at some concepts that allow us to reduce the amount objects with big memory usage, by replacing them with systems that use less space.

7.1 Doubling Arrays Instead of Linked Lists

In Index 1 through 5 we extensively use linked lists for to store our data, but this is very inefficient. For each linked list object there is a finite amount of bytes, lets call it C, which is used as a *header* for the object. The header contains information about what kind of class the object is an instance of, along with other meta-data. However each of the linked list objects contain such a header, so if we have k of such objects, then we will be using $C \cdot k$ amounts of bytes on headers alone.

For any linked list object that only contains one field, we can simply use a doubling array instead. All the memory used for having a header for each object is thus replaced by the header of a single array, which is of constant size no matter the size.

Additionally linked lists provide terrible caching. Standard caching schemes utilise that data that is related is often placed close to each other, but this is not necessarily the case for linked lists. This concept is called the Principle of Locality, and you can find more about it in the book Modern Operating Systems by Andrew Tanenbaum and Herbert Bos [TB14, Chapter 1.3.2]. Changing to arrays also removes this possibility of having bad caching.

7.2 Look-Up Tables

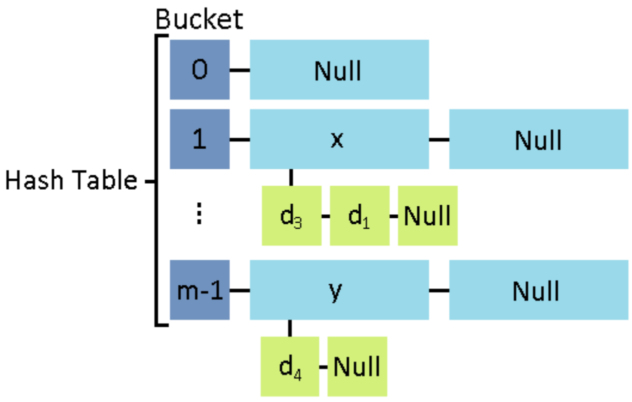
The concept of look-up tables is that whenever we have several objects containing the same big field, we can instead have a table which contains those fields and have the objects store a integer primitive of 4 bytes that is used as index in the look-up table in order to get the desired instance of that field.

In many areas Java already does this by the use of a system called the *constant pool*[1]. For example, strings in Java are objects containing a pointer to a character-array object which contains the actual string. This character-array object is stored in the relevant constant pool. Whenever a new string object with the same string is made, Java simply gives this new string object the pointer to the character-array object that was made previously. Now there are two string objects pointing to the same character-array object.

7.3 Description

In Index 5 the data structure consists of a hash table using chained hashing. Each bucket has a L_w linked list of WikiItem objects, that each represent a unique word from the input set S. Each WikiItem object contains another L_d linked list system of DocItem objects, each representing the documents wherein the given unique word appears.

Figure 7.1: Data structure of Index 5.



Light blue being WikiItem objects and green being DocItem objects. From the testing of memory usage in Index 4 we recollect that the DocItem objects represent the majority of the memory usage. Therefore we want to use the just described methods to decrease the amount of space they use.

[1]The Java Constant Pool – https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.htmljvms-5.1

7.3.1 Using a Look-Up Table for Document Title References

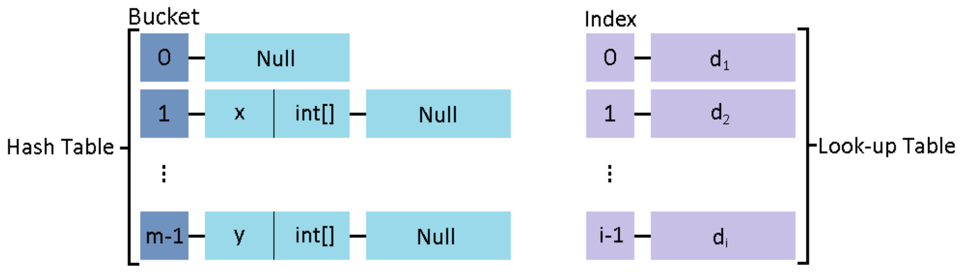
For each DocItem object, that is created from the same document, there is a string object pointing at the given character array that actually holds the document title. Since there is one DocItem for each unique word in a document, lets call the number u_d, there are also u_d string objects which point to the same character array. To reduce this redundancy we can use a look-up table. This would mean we have only one string object per document, and to replace the reference in the DocItem objects we have a integer that points to the index in the look-up table where the string object for the document title is stored. Thus we have reduced the amount of string objects from u_d per document to 1, but instead we have a integer primitive in each DocItem.

7.3.2 Changing L_d to Doubling Arrays

Given that the DocItem objects in the L_d linked list only contain the reference to the look-up table, we can replace the L_d system by having a doubling array as previously explained. This means that the WikiItem objects will store an integer array which contains all of the look-up table indexes of the documents that the specific word occurs in.

The new data structure looks as followed.

Figure 7.2: Data structure with look-up table.



Both the look-up table and the integer array is implemented using doubling arrays, which means they can be anywhere between half full and filled, except on the initial table. The int[] array is initialised at a length of 1, and the look-up table starts at 4.

We still have the L_w linked list of WikiItem objects, and they still have their own string objects related to them. We keep it this way because each of these strings point to a unique word, (except if they appear in a document title) and the L_w linked lists we have shown to be of constant length in Index 4. Therefore there is little to gain from either changing to a look-up table nor changing to a doubling array.

7.4 Analysis

Our goal with this implementation was to decrease the memory usage, so we will start with trying to approximation of how well that was accomplished. Unfortunately there is no way of actually testing it, due to how the Java garbage collection can be instantiated at seemingly random times means that looking at the differences in heap memory is not an option. Instead we must be satisfied with the following approximation.

7.4.1 Approximation of Memory Usage

The modifications that have been made in this chapter does not change the amount of objects per input, so the complexity remains the same. However there have been drastic changes to the amount of space each input results in. To get an approximation I will find the amount of allocated bytes for each object and multiply it with the amount of instance of that object.

In order to find the size of objects I will be using the Java Object Layout (JOL) library, which is a tool that comes with the open source package OpenJDK. [2] JOL uses Unsafe, JVMTI, and Serviceability Agent (SA) to decode object layout, footprint and references. I will be using the function " `ClassLayout.parseClass(<class>)` " which is run on the layout of objects only, returning the fields within the object along with how much memory is allocated to said field. Using it on an integer array yields the following output.

```
[I object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0    16        (object header)                           N/A
     16     0    int [I.<elements>                             N/A
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

Thus it shows the memory used for an empty integer array. To figure out how much space any given integer array uses simply multiply the size of an integer with the amount of integers and then add the 16 bytes from the header.

When used on a stirng object the following is returned.

```
java.lang.String object internals:
 OFFSET  SIZE     TYPE DESCRIPTION                             VALUE
      0    12          (object header)                         N/A
     12     4   char[] String.value                            N/A
     16     4      int String.hash                             N/A
```

[2]Java Object Layout – http://openjdk.java.net/projects/code-tools/jol/

```
     20     4           (loss due to the next object alignment)
Instance size: 24 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

Here we see that any string has a pointer to a character-array and has an integer field for a hash value. Additionally there have been a 4 byte " *loss due to the next object alignment* ". This is because Java uses 8 bytes addresses, so 4 bytes of padding is added to adhere to the structure. All in all a string object (without the character array) uses 24 bytes.

When trying to approximate the size of the data structure, we will be ignoring the character-array objects that contain the actual words, since the size of these will be equal for both of the implementations. This is true because of how the constant pool works. Thus all we need to worry about is how much space the referencing uses. For the following approximations we do not take into account that doubling tables allocate more space than is actually occupied.

7.4.1.1 Index 5

There are three major objects in this implementation, the hash table, the WikiItem objects and the DocItem objects.

Hash Table I use a wrapper class to keep track of relevant information about the hash function. The memory usage of this wrapper class is constant compared to the input size, and equal in both implementations. Therefore it will be ignored.

The hash table itself is an WikiItem-array object, since it only contains the head of each L_w list. When running the parsing function from JOL on it the following output is returned.

```
[LIndex5$WikiItem; object internals:
 OFFSET  SIZE              TYPE DESCRIPTION                      VALUE
      0    16                   (object header)                  N/A
     16     0   Index1$WikiItem [LIndex1$WikiItem;.<elements>    N/A
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

Which is almost the same result as with the integer array. Therefore the memory usage of the hash table is $m \cdot 4 + 16$ bytes, since a reference uses 4 bytes in 32-bit Java. We will ignore the initial 16 bytes since they are in both approximations.

$$\text{Space(Hash Table)} = m \cdot 4$$

WikiItem

```
Index5$WikiItem object internals:
 OFFSET  SIZE               TYPE DESCRIPTION                    VALUE
      0    12                    (object header)                N/A
     12     4   java.lang.String WikiItem.str                   N/A
     16     4     Index1.DocItem WikiItem.docs                  N/A
     20     4    Index1.WikiItem WikiItem.next                  N/A
     24     4             Index1 WikiItem.this$0                N/A
     28     4           (loss due to the next object alignment)
Instance size: 32 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

Since there is a string reference for each WikiItem we need to add the memory used by said string object. The rest of the fields are simple references. All in all a WikiItem object uses $32 + 24 = 56$ bytes. Meaning all of the WikiItem objects together will be

$$\text{Space(WikiItems)} = L_w \cdot 56$$

DocItem

```
Index5$DocItem object internals:
 OFFSET  SIZE               TYPE DESCRIPTION                    VALUE
      0    12                    (object header)                N/A
     12     4   java.lang.String DocItem.str                    N/A
     16     4     Index1.DocItem DocItem.next                   N/A
     20     4             Index1 DocItem.this$0                 N/A
Instance size: 24 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

Each DocItem also has a string object related to it, so the size of a DocItem is $24 + 24 = 48$ bytes.

$$\text{Space(DocItems)} = L_d \cdot 48$$

We can now setup a formula to approximate the total memory usage for referencing in Index 5.

$$\begin{aligned}\text{Space(references)} &= \text{Space(hash table)} \quad &&+\text{Space(WikiItems)} \quad &&+\text{Space(DocItems)}\\ &= (m \cdot 4) &&+(L_w \cdot 56) &&+(L_d \cdot 48)\end{aligned}$$

7.4.1.2 Index 6

The major change lies in switching to a look-up table instead of using the DocItem. The three objects for this implementation are the hash table, the WikiItems and the look-up table, but the hash table is exactly the same.

WikiItem

```
Index6$WikiItem object internals:
 OFFSET  SIZE              TYPE DESCRIPTION                     VALUE
      0    12                   (object header)                 N/A
     12     4               int WikiItem.docsIndex              N/A
     16     4             int[] WikiItem.docs                   N/A
     20     4  java.lang.String WikiItem.str                    N/A
     24     4    Index1.WikiItem WikiItem.next                  N/A
     28     4            Index1 WikiItem.this$0                 N/A
Instance size: 32 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

With the new setup for the WikiItem class the size is still 32 bytes, but now there is both a string reference and a integer array that needs to be counted as well.

$$\begin{aligned}\text{Space(WikiItem)} &= (L_w \cdot 32) + (L_w \cdot 24) + (L_w \cdot 16 + L_d \cdot 4)\\ &= L_w \cdot 76 + L_d \cdot 4\end{aligned}$$

Look-up Table The look-up table contains a reference to a string for each document title, d, which together is $28 \cdot d$ bytes.

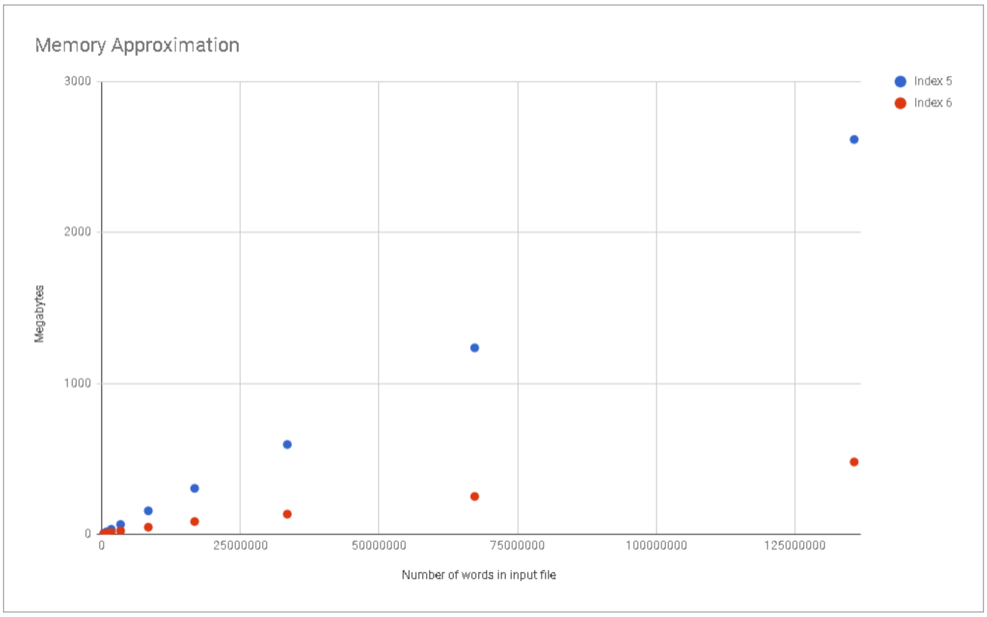
$$\text{Space(Look-up Table)} = d \cdot 28$$

The formula for the combined approximation is thus

$$\begin{aligned}\text{Space(references)} &= \text{Space(hash table)} && +\text{Space(WikiItems)} && +\text{Space(Look-up Table)}\\ &= (m \cdot 4) && +(L_w \cdot 76 + L_d \cdot 4) && +(d \cdot 28)\end{aligned}$$

All of these variables are specific values for each file, and can be found through counters. Using this we can find an approximate memory usage for Index 5 and Index 6 for each file and compare them in a plot. The unit has been changed from bytes to megabytes for the graph.

Figure 7.3: .



Here we see that the difference in approximate memory usage is massive. A closer look at the numbers yield that up towards 2200 MB memory is retained on the biggest file. The follwing numbers are all in MB.

File	Index 5	Index 6	Difference
1	7.43	3.38	4.05
2	17.56	6.79	10.77
3	34.22	11.26	22.96
4	66.25	19.46	46.79
5	156.59	38.65	117.94
6	305.48	67.45	238.03
7	596.05	115.06	480.99
8	1236.28	213.91	1022.37
9	2616.52	406.80	2209.72

With this new-found memory the search engine has enough space to run the preprocessing on file 10 containing the entirety of the Wikipedia snapshot from 2010.

7.4.2 Preprocess

The modifications made should not change the preprocess run time. If anything the additional book-keeping from having doubling arrays might make it a little slower.

Thus it should have the same complexity we found in Index 4 of

$$T(n) = O(n)$$

7.4.3 Search

In Index 4 we found the searching phase to be of run time complexity

$$T(n) = O(1 + d_x)$$

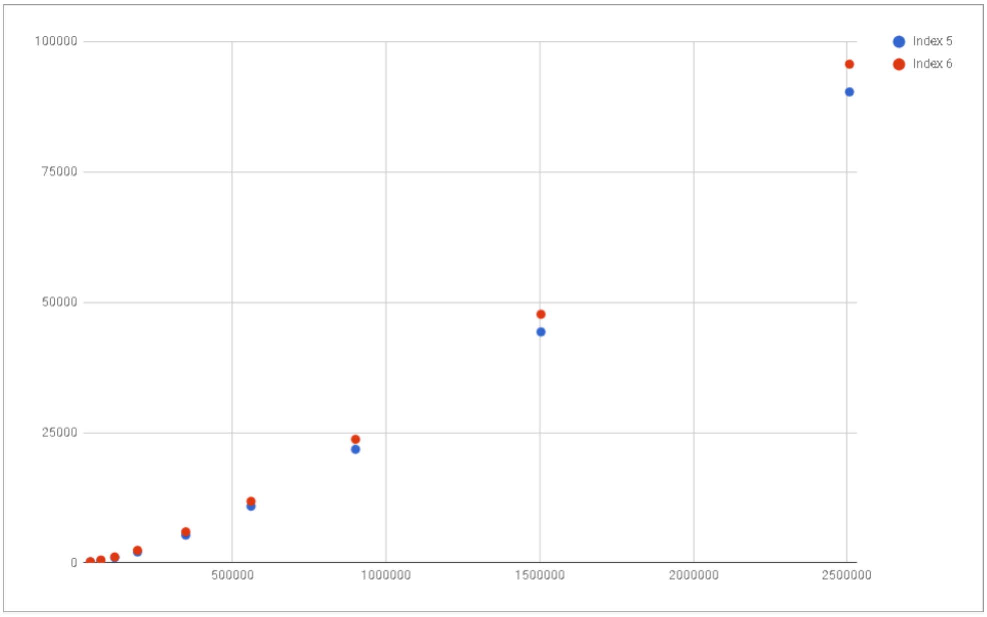
Which should still be the case in this version. However, since we have switched the structure of the L_d linked list to be a doubling array referencing a look-up table, the objects that we are checking should be located closer to each other in memory allowing for better cache performance.

7.5 Test

7.5.1 Preprocess

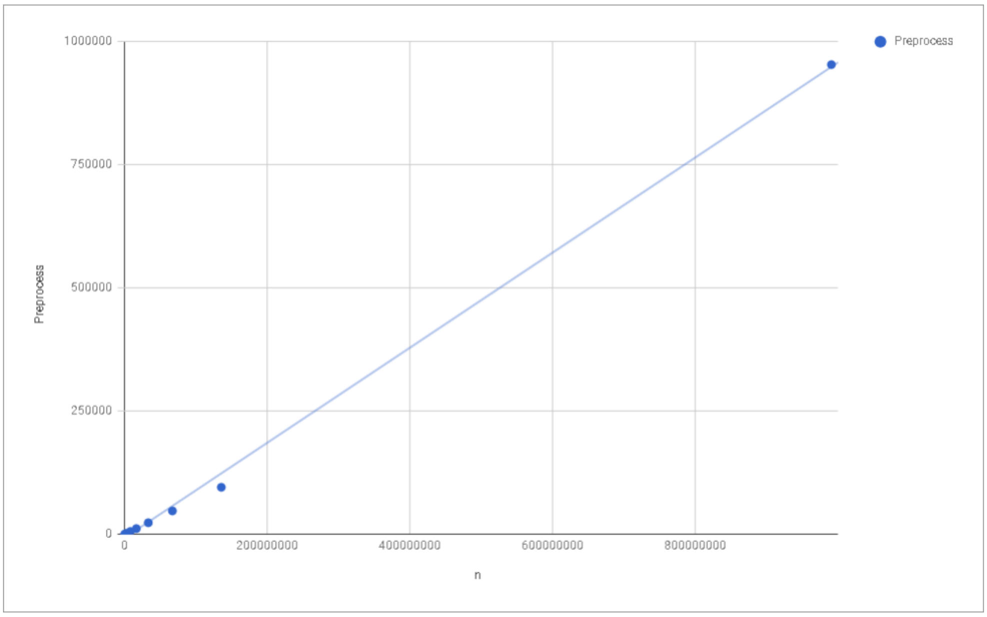
To show that the preprocess run time is equal, with a little bit bigger constant, due to a little extra book-keeping we have the following graph.

Figure 7.4: Preprocess runtime comparison.



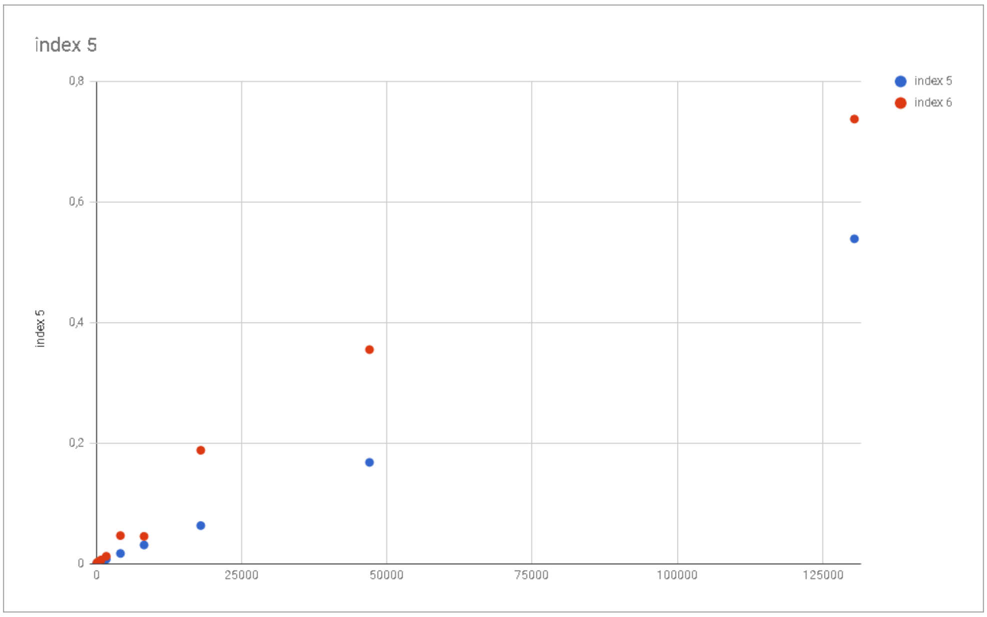
Since this new solution can handle the 10th file, a graph showing the linear dependency on the input size can show made.

Figure 7.5: Preprocess runtime comparison.



7.5.2 Search

Figure 7.6: Preprocess runtime comparison.

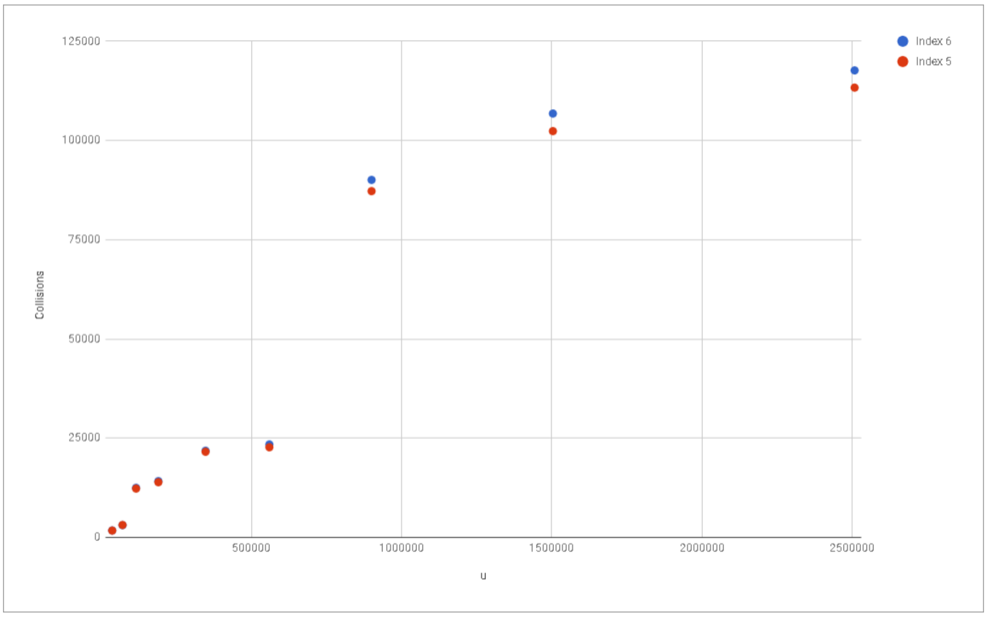


Index 6 seems to be a little slower. This is surprising considering the caching should be better for Index 6.

7.5.3 Collisions

We should also show that the collisions have not suffered due to this modification we will compare the number of collision to the Index 5 implementation where they both use the hash family $\mathcal{H}_3$.

Figure 7.7: Collision comparison.



CHAPTER 8

Further Possibilities and Interesting Inquiries

8.1 Why did $\mathcal{H}_3$ Work so Well?

In Index 5 I tested the 2-universal hash family $\mathcal{H}_3$ to have the same amount of collisions as the strongly universal $\mathcal{H}_4$ and $\mathcal{H}_5$. The fact that $\mathcal{H}_3$ is 2-universal means that it is suppose to have an average bucket length that is longer than a strongly universal hash function, which should result in more collisions.

8.2 Java's Hash Function

Interestingly Java's inbuilt hash code for strings was tested to perform almost as well as the 2-universal hash family $\mathcal{H}_3$ and the strongly universal hash families $\mathcal{H}_4$ and $\mathcal{H}_5$. It would be interesting to test if this is a product of the specific test files used in this thesis or if it behaves equally on all inputs.

8.3 Additional Features

There are a lot of basic features that my implementation of this search engine does not provide, such as sub-string searching, searching for strings with multiple words, prefix searching and many others.

CHAPTER 9

Conclusion

The objective of this thesis was to make an efficient search engine with a data structure that can handle large input files, while still being able to perform fast searches on the entirety of the data structure. I have accomplished this with a step by step procedure, improving the search engine with one modification at a time. After each modification analysis and testing has been performed the evaluate the performance.

The end result is a data structure of a chained hash table, with an element for each unique word in the input file, along with list of indexes for a look-up table that tells us in which documents each unique word appears.

The hash table uses a 2-universal hash function, $\mathcal{H}_3$, which utilises random seeds while still providing a collision probability of $\frac{2}{m}$, where m is the size of the table.

The specific complexity of each version are as followed:

File	Preprocess	Search	Space
Index 1	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Index 2	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Index 3	$\mathrm{O}(\mathrm{u}\cdot n)$	$\mathrm{O}(\mathrm{u}+\mathrm{d}_x)$	O(n)
Index 4	O(n)	$1+\mathrm{d}_x$	O(n)
Index 5	O(n)	$1+\mathrm{d}_x$	O(n)
Index 6	O(n)	$1+\mathrm{d}_x$	O(n)

While the complexity of the last two version have not changed, there have still been improvements. Index 5 gave the search engine a 2-universal hash function which provides an proven average expected length of each bucket in the hash table, and Index 6 improved the memory usage drastically in order to run much bigger files.

The end goal of being able to search through the entirety of the Wikipedia snapshot from 2010, was reached by the end of Index 6 with reasonable speed.

APPENDIX A
An Appendix

I will now provide the test data in tables following the order they were showed in the thesis. My tests were designed to only output the median, so these tables only provide the median as well. All run times are showed in miliseconds.

Index 1

File	n	Preprocess	Search	#Objects
1	329292	216	6	329292
2	828068	378	18	828068
3	1672002	778	45	1672002
4	3348766	1874	96	3348766
5	8356373	8421	138	8356373
6	16717033	16857	174	16717033

Index 2

File	n	Preprocess	Search	#Objects
1	329292	291	11	329292
2	828068	998	25	828068
3	1672002	2159	50	1672002
4	3348766	4597	101	3348766
5	8356373	13309	137	8356373
6	16717033	29167	176	16717033

Index 3

File	n	u	#documents	Preprocess	Search	V_w	W_d
1	329292	37232	126	18651	0,36	37232	113307
2	828068	71494	359	66082	0,9	71494	289279
3	1672002	116600	842	185036	1,39	116600	600650
4	3348766	190748	1745	521444	1,89	190748	1202962
5	8356373	347798	4182	1688393	3,71	347798	2971333

Index 4

File	n	u	Preprocess	Search	V_w	V_d	m	Total space
1	329292	37232	216	0,0007	37232	113307	65536	216075
2	828068	71494	455	0,0021	71494	289279	131072	491845
3	1672002	116600	906	0,0051	116600	600650	131072	848322
4	3348766	190748	2011	0,0095	190748	1202962	262144	1655854
5	8356373	347798	5721	0,0225	347798	2971333	524288	3843419
6	16717033	560184	11821	0,0437	560184	5932491	1048576	7541251
7	33442256	900298	23103	0,0919	900298	11883285	1048576	13832159
8	67263274	1503743	47029	0,2511	1503743	25077735	2097152	28678630
9	135750192	2507802	99754	0,7593	2507802	53883413	4194304	60585519

$\alpha > 1.0$

File	n	u	Preprocess	Collisions	V_w	m	Load factor
1	13878	37232	216	1704	37232	65536	0,5681152344
2	164874	71494	455	2994	71494	131072	0,5454559326
3	329292	116600	906	12403	116600	131072	0,8895874023
4	828068	190748	2011	13876	190748	262144	0,727645874
5	1672002	347798	5721	21362	347798	524288	0,6633720398
6	3348766	560184	11821	23375	560184	1048576	0,5342330933
7	8356373	900298	23103	89478	900298	1048576	0,8585910797
8	16717033	1503743	47029	106958	1503743	2097152	0,7170405388
9	33442256	2507802	99754	120138	2507802	4194304	0,5979065895

$\alpha > 0.75$

File	n	u	Preprocess	Collisions	V_w	m	Load factor
1	329292	37232	210	1704	37232	65536	0,5681152344
2	828068	71494	460	2994	71494	131072	0,5454559326
3	1672002	116600	994	3367	116600	262144	0,4447937012
4	3348766	190748	2048	13876	190748	262144	0,727645874
5	8356373	347798	4999	21362	347798	524288	0,6633720398
6	16717033	560184	9964	23375	560184	1048576	0,5342330933
7	33442256	900298	20566	25695	900298	2097152	0,4292955399
8	67263274	1503743	46923	106958	1503743	2097152	0,7170405388
9	135750192	2507802	97755	120138	2507802	4194304	0,5979065895

$\alpha > 0.50$

File	n	u	Preprocess	Collisions	V_w	m	Load factor
1	13878	37232	225	446	37232	131072	0,2840576172
2	164874	71494	466	788	71494	262144	0,2727279663
3	329292	116600	986	3367	116600	262144	0,4447937012
4	828068	190748	2000	3870	190748	524288	0,363822937
5	1672002	347798	5297	5895	347798	1048576	0,3316860199
6	3348766	560184	10694	6607	560184	2097152	0,2671165466
7	8356373	900298	21485	25695	900298	2097152	0,4292955399
8	16717033	1503743	43676	30705	1503743	4194304	0,3585202694
9	33442256	2507802	88520	35169	2507802	8388608	0,2989532948

$\alpha > 0.25$

File	n	u	Preprocess	Collisions	V_w	m	Load factor
1	13878	37232	215	112	37232	262144	0,1420288086
2	164874	71494	456	217	71494	524288	0,1363639832
3	329292	116600	1037	923	116600	524288	0,2223968506
4	828068	190748	1995	1017	190748	1048576	0,1819114685
5	1672002	347798	5167	1605	347798	2097152	0,1658430099
6	3348766	560184	10671	1905	560184	4194304	0,1335582733
7	8356373	900298	21476	7386	900298	4194304	0,2146477699
8	16717033	1503743	44700	9408	1503743	8388608	0,1792601347
9	33442256	2507802	91186	11656	2507802	16777216	0,1494766474

Index 5
Collisions:

File	n	u	m	Load factor	Java	H3	H4	H5
1	329292	37232	65536	0,5681152344	1704	1746	1762	1717
2	828068	71494	131072	0,5454559326	2994	3155	3073	3054
3	1672002	116600	131072	0,8895874023	12403	12321	12325	12352
4	3348766	190748	262144	0,727645874	13876	13924	13865	13868
5	8356373	347798	524288	0,6633720398	21362	21595	21267	21375
6	16717033	560184	1048576	0,5342330933	23375	22732	22755	22840
7	33442256	900298	1048576	0,8585910797	89478	87247	86878	87454
8	67263274	1503743	2097152	0,7170405388	106958	102364	101731	101912
9	135750192	2507802	4194304	0,5979065895	120138	113319	112919	112723

Preprocess run time:

File	n	u	Java	H3	H4	H5
1	329292	37232	216	191	246	249
2	828068	71494	455	477	561	575
3	1672002	116600	906	1028	1129	1195
4	3348766	190748	2011	2099	2279	2387
5	8356373	347798	5721	5336	5889	6041
6	16717033	560184	11821	10874	12368	12467
7	33442256	900298	23103	21798	24121	25160
8	67263274	1503743	47029	44327	47226	53180
9	135750192	2507802	99754	90376	95264	102895

Searching run time:

File	n	u	Java	H3	H4	H5
1	329292	37232	0,0007	0,0008	0,0006	0,0006
2	828068	71494	0,0021	0,0016	0,0032	0,0022
3	1672002	116600	0,0051	0,0045	0,0046	0,0057
4	3348766	190748	0,0095	0,0071	0,0092	0,0098
5	8356373	347798	0,0225	0,0168	0,0196	0,0218
6	16717033	560184	0,0437	0,0308	0,0406	0,0427
7	33442256	900298	0,0919	0,063	0,0829	0,0858
8	67263274	1503743	0,2511	0,1679	0,2109	0,2174
9	135750192	2507802	0,7593	0,5389	0,7215	0,666

Index 6

File	n	Preprocess	Search	V_w	m	col	Load Factor
1	329292	240	0,0009	37232	65536	1773	0,5681152344
2	828068	562	0,0031	71494	131072	3114	0,5454559326
3	1672002	1151	0,006	116600	131072	12521	0,8895874023
4	3348766	2420	0,0121	190748	262144	14204	0,727645874
5	8356373	5981	0,0464	347798	524288	21857	0,6633720398
6	16717033	11833	0,0451	560184	1048576	23446	0,5342330933
7	33442256	23717	0,188	900298	1048576	90072	0,8585910797
8	67263274	47708	0,355	1503743	2097152	108791	0,7170405388
9	135750192	95699	0,7376	2507802	4194304	127663	0,5979065895
10	990248676	953490	28,8514	13071636	16777216	1118980	0,7791302204

Bibliography

[Cor+09] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition.* 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[CW79] J.Lawrence Carter and Mark N. Wegman. "Universal classes of hash functions". In: *Journal of Computer and System Sciences* 18.2 (1979), pages 143–154. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/0022-0000(79)90044-8`. URL: `http://www.sciencedirect.com/science/article/pii/0022000079900448`.

[Die96] Martin Dietzfelbinger. "Universal hashing and k-wise independent random variables via integer arithmetic without primes". In: *STACS 96.* Edited by Claude Puech and Rüdiger Reischuk. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pages 567–580.

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems.* 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN: 013359162X, 9780133591620.

[Tho15] Mikkel Thorup. "High Speed Hashing for Integers and Strings". In: *CoRR* abs/1504.06804 (2015). arXiv: `1504.06804`. URL: `http://arxiv.org/abs/1504.06804`.

[WC81] Mark N. Wegman and J.Lawrence Carter. "New hash functions and their use in authentication and set equality". In: *Journal of Computer and System Sciences* 22.3 (1981), pages 265–279. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/0022-0000(81)90033-7`. URL: `http://www.sciencedirect.com/science/article/pii/0022000081900337`.