Data Structures and Algorithms – Coursework 1.

Daniyar Nazarbayev.

H00204990.

1. About the program.

The program that was asked of us, is a spellchecker. The spellchecker, should take in a dictionary file, which is a .txt file with 180,000 words. Then it should take in the second file, one that needs checking. University provided us with 6 files, named: d1.txt, d2.txt etc. The first file has 1000 words, the sixth 45,000.

We should use a hashtable to store the dictionary, and as for the spellchecker itself – there should be 4 methods of spell correction.

In my case, I use a filewriter. Any file I take in, I delete the last 4 charaters off the name (.txt) and add -corrected.txt at the end.

My program includes it all.

2. What was used.

I personally used eclipse. I also used the provided interfaces, dictionary, and the 6 txt files. The only class that I did not use was the FileWordRead.

At first I had a separate class called Test, that would call the Spellchecker constructor, but in the end, after doing some research, I ended up implementing a main method with arguments in the spellchecker class itself. It still calls the constructor, but this time its own. I decided that it would be much more convenient like that, since I would have to change a lot of variables and methods to static.

To run the main method in eclipse, you can just right click on the SpellCheck class and click

Run As -> Run Configurations

Then, there should be a tab called arguments – input 2 of them, including the file extensions (which should be .txt). The first one should be the dictionary, and the second one should be the file you want to check. You can create your own, or use ones provided.

3. Coursework.

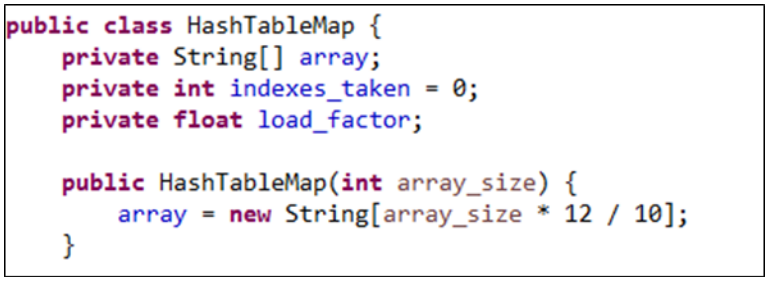
When I first started on my coursework, it was the 20th of October. I had a workable copy in 3 days, with about 2 correction methods, but I took a few liberties. All of it was built from ground up, but I have used none of the interfaces provided by the university.

In the coursework paper, it says that they want you to start with an array size of 7, and increase it to the next prime number that is at least twice as large. Basically, 7, 17, 37 … 175447, 350899. Altogether it takes 16 iterations to get to the final number, and odds are that it can take 17 if your max load factor is too low, like 50%.

With each iteration, you would have to start over, and rehash all the words already in the array. Up until you get to iteration 16-17, you will keep doing that.

In my constructor, I would just read all the lines in the file (in this case, they were separated by line feed and carriage return, so one line would be one word), and that would be the input of my constructor. That input is multiplied by 20% and then that number is assigned as array size. In the very beginning, it took me 15 seconds to add a dictionary to the hash table, and about 30 seconds for the other – both using the same hash function.

That said, in the subsequent weeks, I have been working on implementing the coursework in exactly your way. The subsequent sections are basically comparisons between my old and new code.



```
public class HashTableMap {
    private String[] array;
    private int indexes_taken = 0;
    private float load_factor;

    public HashTableMap(int array_size) {
        array = new String[array_size * 12 / 10];
    }
```

3a. Hash Function.

Before going into details on my coursework, I want to focus on the hash function, or to be precise, the interface provided. Interfaces in java, force the class to add the methods in the interface. In this interface, the only method that is provided, giveCode() returns a value of int.

When I was making my original hash table, my hash function was divided a string into a char array and using a loop to add each character's ascii code, each multiplied by 33. This is what I thought to be the polynomial accumulation, but there was a slight error in that – with each character, I had to increase the power of 33 that I multiply it on, which I did not.

But even if my function was correct from the start, it still won't work, since it returns int. Polynomial accumulation definitely needs "long" type (64 bit). In fact, I did some tests. In java, there are no unsigned types, which is unfortunate considering that there can be no negative values when you use modulo. Signed 32 bit has the max value at about 2,147,483,647. If I were to find the polynomial of the word "hello", it would give 135,639,476. Something like "goodbye" gives 135,293,586,025.

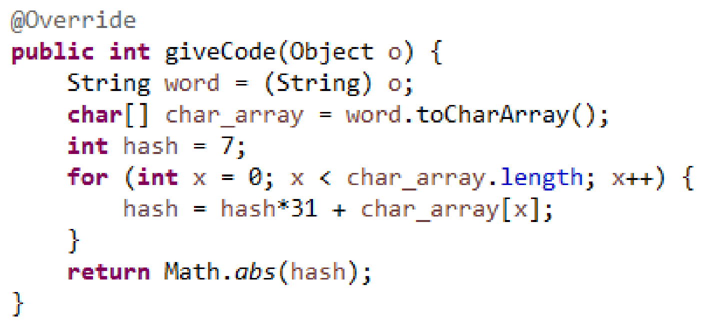
This is the max value of signed long: 9,223,372,036,854,775,807.

Overflow is inevitable in case of int. I did read the paper though, and from what I understood, you want us to have a hash function that will give collisions. It is kind of oxymoron though, since hash functions were designed to give unique values.

My original function took about 15 seconds to add. The test class provided says I get 900 probes on average on every load factor. When I improved the function, and changed return type to long, it would take only 1 second – amount of probes is at about 1 too. There was no rehashing in there though, but fact is that the complexity of inserting and finding became O(1), since pretty much every key has a unique value.

Since we are not allowed to use long, I decided to look up a few functions, and here is one that I saw on stack overflow. He even added a statement, that one should not base his hash function on addition of all characters, since both house and esouh will have the same hash code.

At the end, the hash function even started giving negative values, meaning it started overflowing. Nevertheless, I just added the absolute function to it, and the test showed average probes being at about 1.



```
@Override
public int giveCode(Object o) {
    String word = (String) o;
    char[] char_array = word.toCharArray();
    int hash = 7;
    for (int x = 0; x < char_array.length; x++) {
        hash = hash*31 + char_array[x];
    }
    return Math.abs(hash);
}
```

3b. Inserting function.

In one part of the paper, it says that we have to use a second hash table to store the corrected words. There are times when duplicates will appear, and the paper says that we should deal with it, but we have to use a hash table.

The dictionary has no repetition, but some incorrect words do.

At first I used a find method inside my insert method. The problem with that is that it puts a lot of stress on the time it takes. Variable j has to go through N-1 iterations, where N is the array size. When I was first making my find function, one misconception I had was that if say j=3 is not null, then every j after that would be null. That is not the case, and if the array does not have that value, it would increase the complexity a lot, since the method has to check every possible value until j is N-1.

Later on, I did it more elegantly and instead put find in an if statement, and if that will show false, it will insert the value.

Because of that, the initial insertion into the dictionary is much faster.

3c. Find function.

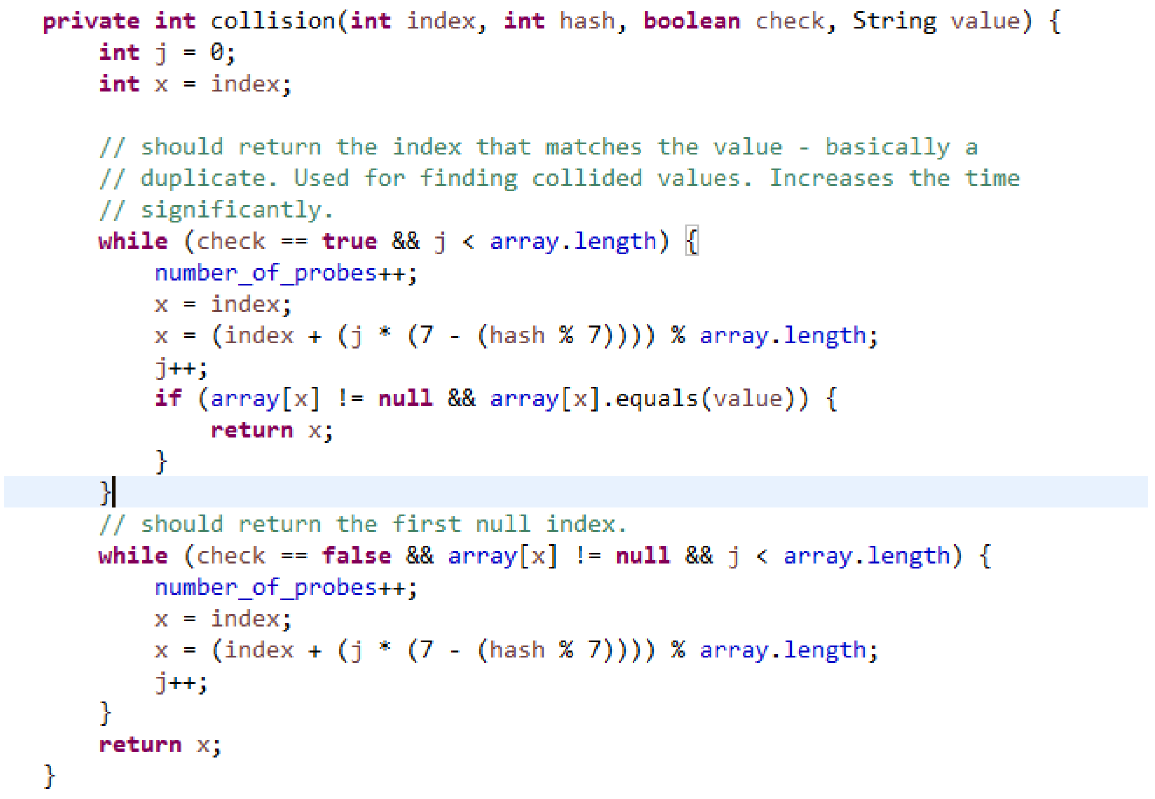
Like I said in my previous section, if the index with a certain number j has a value, does not mean that all the subsequent j's will have indexes equal null (empty). That means, that if the word does not exist in the hash table, the find function will have to go through all the indexes of j, until it confirms that it is not there.

My original implementation needs about 60 seconds to check the d1.txt, and the second hash table implementation takes 250 seconds. What I concluded was that it is all due to array size. In the first implementation, the constructor is given the amount of words there will be, and that number is increased by 20%. In the second implementation, the insert function checks whether the load factor is more than the maximum or not. If it is more than the maximum, it will increase the array size by at least 50% though.

The strength of hash table is in the hash function itself. It should be able to give each value a unique code, and hopefully that code is translated into a unique index.

3d. Collisions.

Here is what I use for my collisions.



```java
private int collision(int index, int hash, boolean check, String value) {
    int j = 0;
    int x = index;

    // should return the index that matches the value - basically a
    // duplicate. Used for finding collided values. Increases the time
    // significantly.
    while (check == true && j < array.length) {
        number_of_probes++;
        x = index;
        x = (index + (j * (7 - (hash % 7)))) % array.length;
        j++;
        if (array[x] != null && array[x].equals(value)) {
            return x;
        }
    }
    // should return the first null index.
    while (check == false && array[x] != null && j < array.length) {
        number_of_probes++;
        x = index;
        x = (index + (j * (7 - (hash % 7)))) % array.length;
        j++;
    }
    return x;
}
```

I have 2 loops in there, and only one can run – which one depends on the input. I use the second loop for inserting, while the first loop is used for finding values (both for remove and find).

I am not quite sure about my double hashing function though. It seems to work, but I still have my doubts. It should be:

(h1 + j * h2) % N

h1 = hash % N

h2 = 7 – hash % 7

On one of the lectures I heard that

3e. Spellchecker.

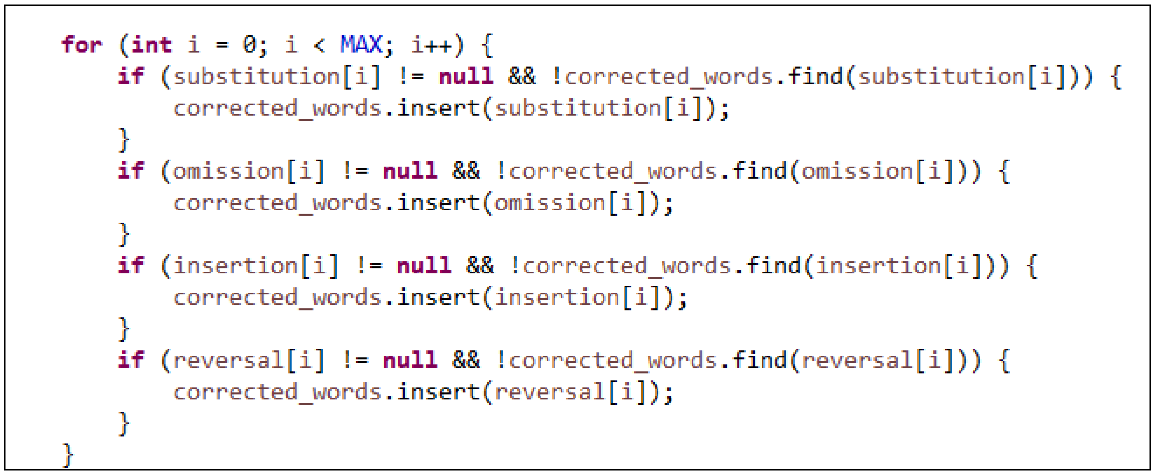
If a find method gives back false, I have to correct the spelling. For my spellchecking, I implemented 4 methods. Not going to go into details, all I will say is that I used toCharArray() to convert the string to chars, and String.valueOf() to convert it back.

Each of the spellchecking methods would give back a string array of 5, so altogether all of them can give 20 possible word collections (that said none ever got that high). If you want to change the maximum array value for each method, change the final variable called MAX.

Then a new instance of a hash table is created, and those words are added there. Like in said in the previous section, I use an if statement before inserting.

Then I use an iterator to print out all the words, and then return that string.

I find all this very unnecessary, but the coursework paper asked to use a hash table to deal with duplication.



```
for (int i = 0; i < MAX; i++) {
    if (substitution[i] != null && !corrected_words.find(substitution[i])) {
        corrected_words.insert(substitution[i]);
    }
    if (omission[i] != null && !corrected_words.find(omission[i])) {
        corrected_words.insert(omission[i]);
    }
    if (insertion[i] != null && !corrected_words.find(insertion[i])) {
        corrected_words.insert(insertion[i]);
    }
    if (reversal[i] != null && !corrected_words.find(reversal[i])) {
        corrected_words.insert(reversal[i]);
    }
}
```

3f. Iterator.

Iterator is another thing that I find very taxing on the program. Hash table is basically an array. I can not just make an iterator. First of all, I have to add each index to the ArrayList, and second of all, for the iterator test to work, I have to remove all the nulls. Basically, to get an iterator, I need to go through the whole array size, picking out any values that are not null, and usually, the load factor is always 50%, so half the values will always be null.

In my original implementation, I used a getter for my array.

4. Problems I faced.

There were 3 problems I faced during the making of the program.

First was that I used == to check whether strings are equal instead of equals().

Second was that when I was implementing the second hash table, I had to make a rehash. When I would make one, I would create a new array and would point it to the old one. While the old array I would point to the new object in memory, but the error in there is that the old array would point to that same "new" object in memory as well. Using Arrays.copyOf() fixed that.

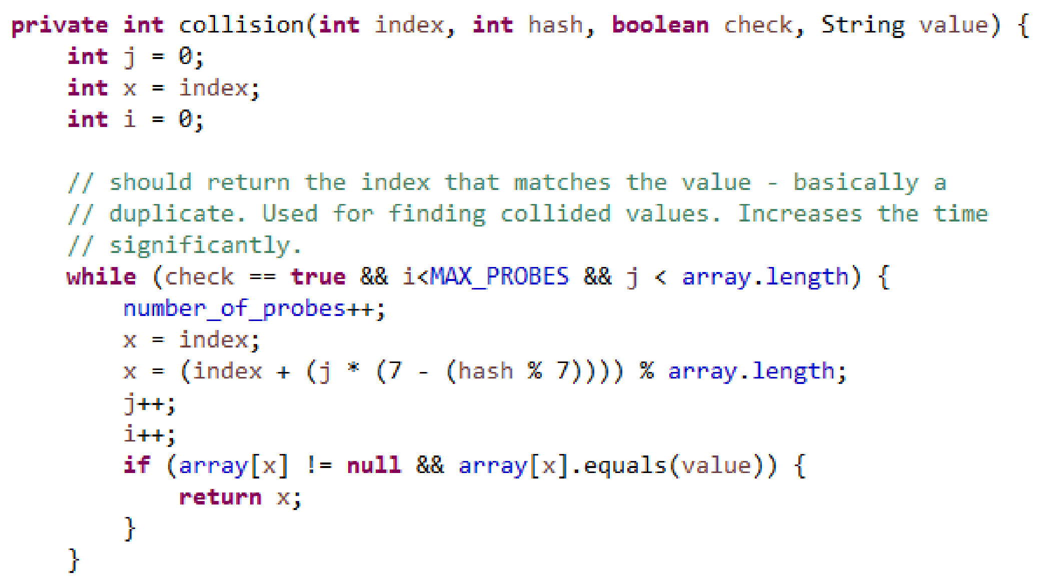
Final problem was the find function, which I talked about 2 times in this report. The couldn't pass the tests provided, to be specific 6-8. It made me look over through all of my code – a day worth of work, until I found the problem in my find method. Here is me repeating it for the 3rd time – if the hash code with a certain j is not null, does not mean that all the indexes with subsequent j's are null.

I wanted to further optimize my code. One thing that I wanted to use was StringBuilder/Buffer, because I was told that having concatenation in a loop can slow down the program. Unfortunately, it did more harm than good, because I would just run out of heap space, even with d1.txt. Turns out, string builder reuses words, if there is duplication – instead of creating a new space in the memory, it instead points to an already existing one. It would not really work for me, because mine should not have, and most importantly, is that I was creating a new string builder for every word, which is what caused the heap space error.

This had me thinking, and I realized that I am doing the same with my hash table for the incorrect words. For every incorrect word, I had created a hash table, and that is only to remove the duplicates. Same thing for the iterators – all this is an incredible waste of memory in my opinion.

5. Complexity.

What I figured out of all this, is that the high complexity is pretty much due to find method – which makes it O(n). It is going to do N number of iterations, and if no such value exists and then it is going to return false. It takes me about an hour to check d6.txt, which has 45,000 words. What I did to fix that was adding a limit to amount of probes a program can make.



```
private int collision(int index, int hash, boolean check, String value) {
    int j = 0;
    int x = index;
    int i = 0;

    // should return the index that matches the value - basically a
    // duplicate. Used for finding collided values. Increases the time
    // significantly.
    while (check == true && i<MAX_PROBES && j < array.length) {
        number_of_probes++;
        x = index;
        x = (index + (j * (7 - (hash % 7)))) % array.length;
        j++;
        i++;
        if (array[x] != null && array[x].equals(value)) {
            return x;
        }
    }
```

I have set max probes to 20. The test provided shows that load factor of 90% still works fine. 95% gives errors. If you are going to check tests, I suggest you comment out the (i<MAX_PROBES) statement.

That said, the performance increase is significant. It takes 1.7 seconds to check the d6.txt file.

I have implemented linked list version, but in all honesty, I do not want to do any tests. It takes 60 ms to add to the dictionary, but about 260 seconds to check d1.txt. It is pretty obvious, since the complexity increases the farther the element is, unlike the hash table where if every value has a unique value, the complexity should be O(1). In the tests that were provided, my average probe counter shows a pretty consistent 1 on all load factors, with 3 later on with load factor 95%, so I guess having a unique number is not a problem.