Course ID1020 - Lab 3

Simon Johannesson

2020-09-28

Assignment 1

File used:

- `blank-non-alpha.c`

There is no actual output. The program writes over the file it receives.
Example file before:

```
Author: Charles Dickens

Release Date: January, 1994 [EBook #98]
Posting Date: November 28, 2009
Last Updated: March 4, 2018

Language: English

Character set encoding: UTF-8
```

Example file after:

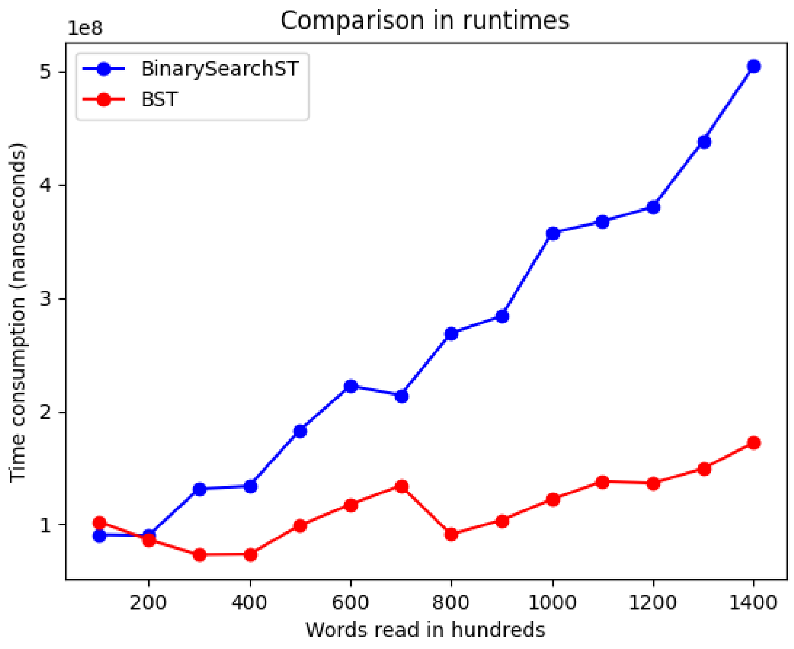
```
Author  Charles Dickens

Release Date  January        EBook
Posting Date  November
Last Updated  March

Language  English

Character set encoding  UTF
```

Assignment 2

In order to compare the runtimes of BinarySearchST and BST the below files were used:

- `BinarySearchST.java`
- `BST.java`
- `FrequencyCounterBinarySearchST.java` a variation of `FrequencyCounter.java`
- `FrequencyCounterBST.java` a variation of `FrequencyCounter.java`

- `parse_and_plot.py`



Assignment 3

The distribution of the `String.hashCode()` function was tested in two different ways.

Storing the hascode values in a BST

To check the distribution all hashcode values were stored in a BST datastructure, any words that have the same hashcode will be recognized as a collision. That way it is possible to realize whether the hashcodes are well distributed based on the ration between unique words in a text and the number of hashcode collisions.

Files used were:

- `BST.java`
- `HashCodeEval.java`
- `plot_distribution.py`

'The text' used as input for the file `HashCodeEval.java` generated the following output:

```
Path is: ./the-text.txt
collisions   = 3
word total   = 141491
unique words = 10982
```

```
Found collisions:
{[He], [IF]}
{[Of], [PG]}
{[VII], [Ugh]}
```

It can be noticed that there are only 3 collisions of the hashcode values over 10982 unique words.

Storing the hascode values in a HashTable using separate chaining

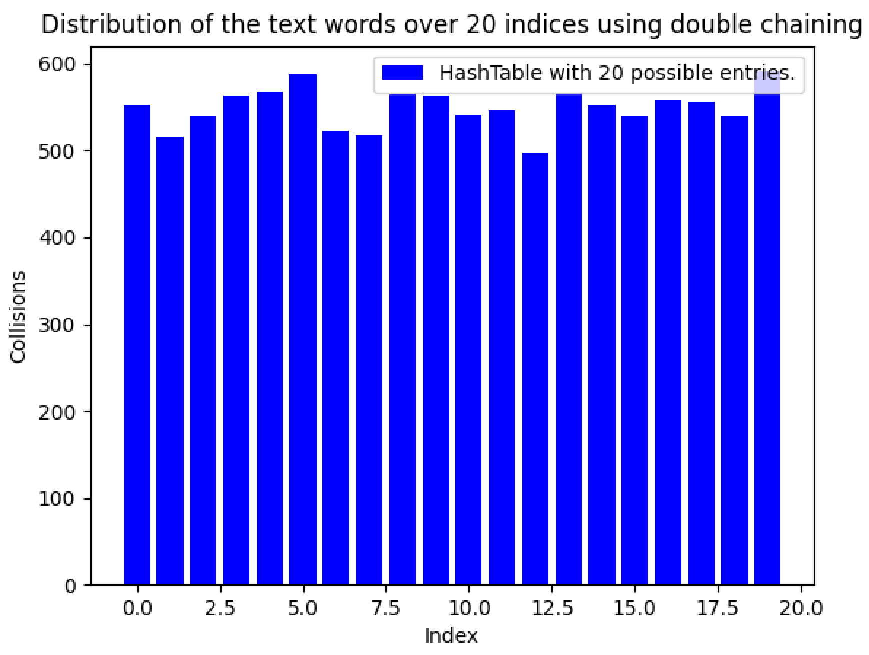
The second test to check the distribution of the hashcode values was to implement a HashTable that will take the hashcode of the key and then hash that again using it's own hash function.

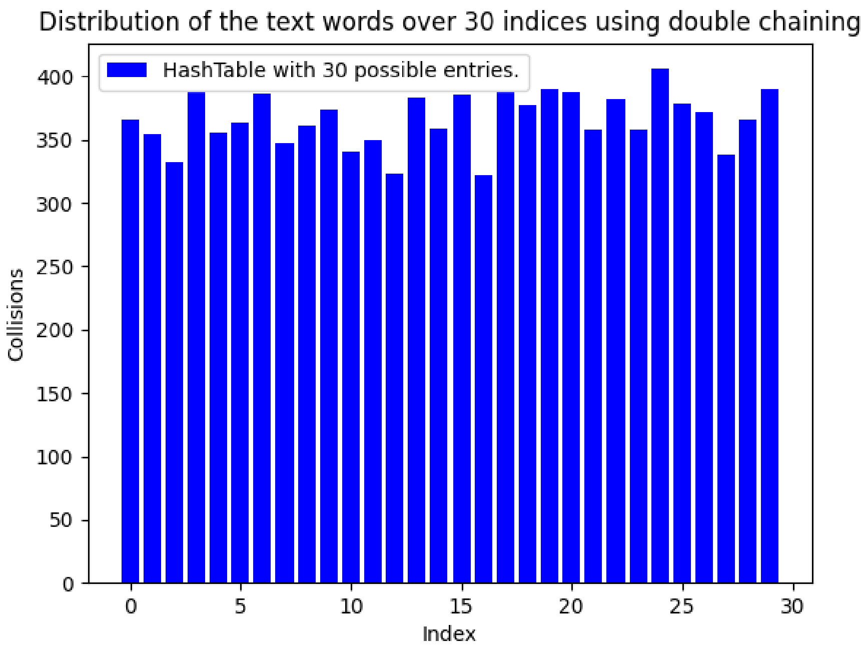
The HashTable's hash function had the following implementation:

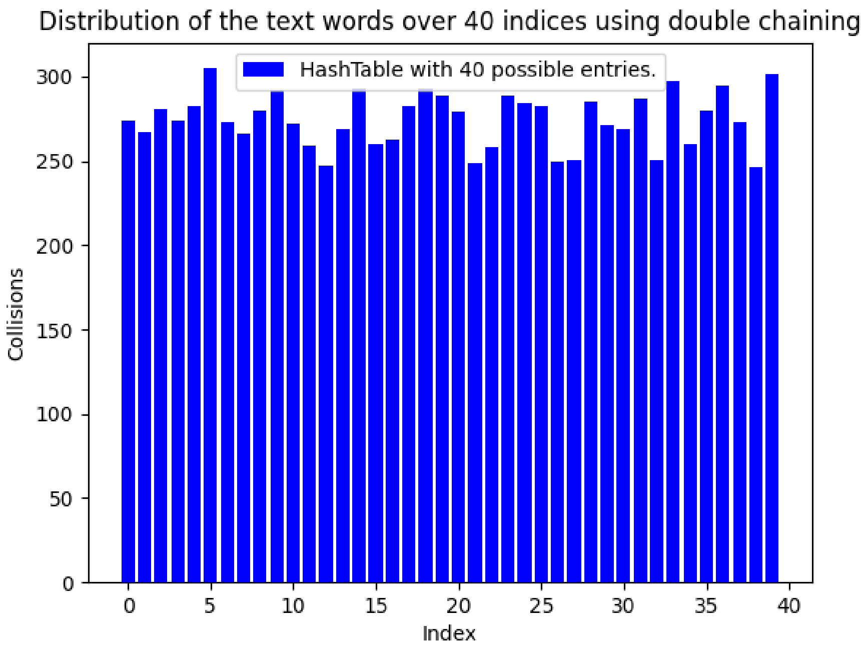
```
/* M is the size of the array the hash must fit into */
private int hash(Key key){
    return (key.hashCode() & 0x7fffffff) % M;
}
```

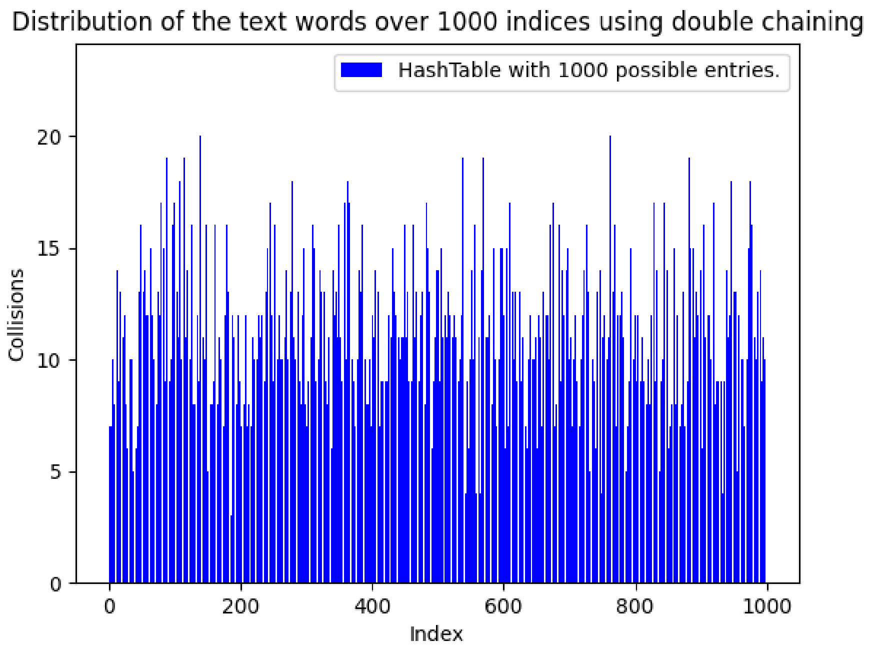
As understood from the course literature the hash functions implementation should portray the characteristic of uniform distribution if the data that it is given is uniformly distributed. So the reasoning is that if the hashcodes are uniformly distributed then the hash function should distribute the keys uniformly in the HashTable.

The results can be seen in the below diagrams with different HashTable-sizes:




Distribution of the text words over 30 indices using double chaining
HashTable with 30 possible entries.
Collisions
Index
0
50
100
150
200
250
300
350
400
0
5
10
15
20
25
30


Distribution of the text words over 40 indices using double chaining
HashTable with 40 possible entries.
Collisions
Index
0
50
100
150
200
250
300
0
5
10
15
20
25
30
35
40



The results on the lower HashTable-sizes seem to be fairly uniform, and even the large HashMap might be considered to be uniform depending on the perspective. Since the words from the beginning are not really uniformly distributed Strings it is unreasonable to expect the result here to be perfectly uniform.

Discussion

The results might be considered inconclusive since the hashcodes are generated from words in a text, and the words in a text are not necissarily a uniform representation of Strings over the same range. Rather it might've been better to use all possible subsets of Strings over the same range instead and compare their hashcodes. Then there would be uniformity through all of the testing and a conclusion could be drawn of the uniformity of the generated hashcode values.

Assignment 4

Files used:

- `IndexFinder.java`
- `BST.java`

Sample execution:

```
 Query index for word:
John
The word 'John' is present at the index/indices:
131478
```

```
356771
356784
365283
603126
603151
603371
603396
603483
607746
Query index for word:
sane
The word 'sane' is present at the index/indices:
482593
```

Theory questions

Show how a binary search tree is built when the following sequence of keys are inserted: W O E C A L H

```
                W
               / \
              /
             /
            /
           O
          / \
         /
        /
       E
      / \
     /    \
    C     L
   / \   / \
  A     H
 / \   / \
```

Show the output if the content of the trees is printed in pre-, in- and postfix order. As per the course pdf by Robert Rönngren.

- Prefix order:

  ```
  W O E C A L H
  ```

- Infix order:

  ```
  A C E H L O W
  ```

- Postfix order:

Explain/show how you measured the execution times in programming assignment 2 above and how the results compare to what the theoretical calculations would suggest

In order to perform the time measurements n-hundred words from 'the text' were read into each datastructure for 0 ¡ n ¡ 1401 with 100 increments on n, and the time was taken for each complete entry of the words. The timing data was then plotted in a graph as seen under section "Assignment 2".

Higher grade

Files:

- `KthWordFinder.java`

Explain the choices of data structures and algorithms, and their complexities.

The problem was split up into three issues.

1. Count frequency of each word
2. Determine k of each word
3. Store data in a datastructure where it can be accessed in constant time

To determine the count frequency the words were put in a BST with the word as the key and the count as value. Once all words had been read in the frequency would be known as the value.

To determine k of each word all the words and values were transferred to a new BST where the count was the key and a linked list of the words with the count was the value. Since it is possible to iterate from smallest to biggest key in the implementation of BST used, each k can be determined when iterating over the keys.

To be able to access the data in constant time a HashTable was used. The HashTable's 'hash array' was initialized with an index that allowed there to be on average 4 elements (words) in each LinkedList connected to each hash. Due to the implementation of hash and the fact that each entry to the HashTable are non-repeating numbers in a range from 0 - k. In order to transfer the data into the HashTable k was determined in conjunction with iterating over the keys in the second BST, and entered into the HashMap. The HashMap had the k-value as key, and the word as value.

On the complexities. To enter a key value pair into a BST is on average the time complexity Log(N), which helps when trying to keep the time consumption down as in this assignment. To iterate over the N first keys in the BST and obtaining their values is on average the time complexity NLog(N), at least in this implementation. To put and get key value pairs in a HashTable the time complexity is constant, or at least near constant. Though it's worst case can be linear, for the reasons mentioned in the previous paragraph it's certain in this case that the worst case will never occur unless an utterly inappropriate Initialization size is used to initialize the HashTable.

Output

Output from running `KthWordFinder.java`:

```
OUT: Path is: ./leipzig1m.txt
     unique: 193903
     Initialization time: about 32.268 seconds

OUT: Shows k:th most common words or k-k+n most common words.
     Use format <k> or <k1>-<k2>
IN:  1
OUT: the
     Shows k:th most common words or k-k+n most common words.
     Use format <integer> or <integer>-<integer>
IN:  2
OUT: of
     Shows k:th most common words or k-k+n most common words.
     Use format <integer> or <integer>-<integer>
IN:  1-5
OUT: the
     of
     to
     a
     and
     Shows k:th most common words or k-k+n most common words.
     Use format <integer> or <integer>-<integer>
IN:  200-205
OUT: industry
     very
     oil
     economic
     re
     another
     Shows k:th most common words or k-k+n most common words.
     Use format <integer> or <integer>-<integer>
IN:  ^D
```