COMP26120
Academic Session: 2020-21

Lab Exercise 3: Spellchecking (Better Trade-offs)

Duration: 2 weeks

You should do all your work on the lab123 branch of the COMP26120_2020 repository - see Blackboard for further details. You will need to make use of the existing code in the branch as a starting point.

Important: This is the third of three labs. **This is an assessed lab**. You have the choice to complete the lab in C, Java or Python. Program stubs for this exercise exist for each language. ***Only one*** language solution will be marked. To submit your work please run the submit script in the relevant directory this means:

- If you are submitting a C solution you should run *submit_lab3_c.sh*
- If you are submitting a Java solution you should run *submit_lab3_java.sh*
- If you are submitting a Python solution you should run *submit_lab3_python.sh*

The most recent submission is the one that will count.

Also Important: After submitting your code log in to COMPjudge to check that you are passing all of the tests (see Blackboard for details on COMPjudge). If you are not then you can resubmit. If you do not see the submission in COMPjudge then first check that you can see the tag in your GitLab page — make sure you are on the correct branch for the submit scripts to work.

For this assignment, you can only get full credit if you do not use code from outside sources.

Note on extension activities: The marking scheme for this lab is organised so that you can get over 80% without attempting any of the extension activities. These activities are directed at those wanting a challenge and may take considerable extra time or effort.

Reminder: It is bad practice to include automatically generated files in source control (e.g. your git repositories). This applies to object files (C), class files (Java), and compiled bytecode files (Python). This is a general poor practice but for this course it can also cause issues with the online tests.

Learning Objectives

By the end of this lab you will be able to:

- Describe the role of the hash function in the implementation of hash tables and describe and compare various hash collision strategies
- Describe the basic structure of a binary search tree as well as the basic operations and their complexities
- Write C, Java, or Python code to implement the above concepts

Introduction

The aim of this exercise is to use the context of a simple spell-checking program to explore the *binary search trees* and *hash tables* data structures.

Data structures

The spell-checking stores the dictionary of words using a *Set* datatype. There are three data structures used to implement this *Set* datatype in this lab. You have already used the *dynamic array* in Lab 1.

In this exercise we use *hash tables* and *binary search trees* to implement the *Set* datatype. These have been introduced in lectures and we describe these briefly below. You may want to look at the recommended textbooks or online to complete your knowledge.

Hash Table. The underlying memory structure for a hash table is an array. A hash function is then used to map from a large 'key' domain to the (much smaller) set of indices into the array, where a value can be stored. When two values in the input domain map to the same index in the array this is called a *collision* and there are multiple ways to resolve these collisions. To use a hash table to represent a set we make the key and value the same - the result is usually called a *hash set*.

Binary Search Tree. You can think of a tree as a linked list where every node has two 'children'. This allows us to differentiate between what happens in each sub-tree. In a binary search tree the general idea is that the 'left' sub-tree holds values smaller than that found in the current node, and the 'right' sub-tree holds larger values.

Lab 3: Better Storage

In the Lab 1 we achieved a faster *find* function by first sorting the input. In this part we explore two data structures that organise the data in a way that should make it faster to perform the find operation.

Part a: Hash Table Lookup

So far our solution to a fast find function has been sorting the input and in Part b we will look at storing it in a sorted order. In this part you will take a different approach that relies on a *hash function* to distribute the values to store into a fixed size array. We have only provided you with very basic program stubs. You need to decide what internal data structure you need etc.

The hash-value(s) needed for inserting a string into your hash-table should be derived from the string. For example, you can consider a simple summation key based on ASCII values of the individual characters, or a more sophisticated polynomial hash code, in which different letter positions are associated

with different weights. **(Warning: if your algorithm is too simple, you may find that your program is very slow when using a realistic dictionary.)** You should experiment with the various hash functions described in the lectures and textbook and implement at least two for comparison. One can be terrible. These can be accessed by the modes already given in the program stubs, please do not change these. Write your code so that you can use the $-m$ parameter, which sets the *mode*.

Initially, you need to use an open addressing hashing strategy so that collisions are dealt with by using a collision resolution function. As a first step you should use linear probing, the most straightforward strategy. To understand what is going on you should add code to count the number of collisions so you can calculate the average per access in print_stats.

An issue with open addressing is what to do when it is not possible to insert a value. To make things simple, to begin with you can simply fail in such cases. Your code should keep the *num_entries* field of the table up to date. Your *insert* function should check for a full table, and exit the program cleanly should this occur. Once this is working you should increase (double?) the hash table size when the table is getting full and then *rehash* into a larger table. In C, beware of memory leaks![1]

Once you have done this, you can submit to COMPjudge to check that it works correctly.

Extension Activities. If you still have time, consider also implementing alternative methods for dealing with collisions. But make sure you have completed Parts b & c without these extensions first as it is likely to be more work than it may initially appear. The alternative methods you may consider (and reasons for considering them) are:

1. *Quadratic Probing.* It is a good idea to get used to the general quadratic probing scheme. The observant among you will notice it's a common question in exam papers!
2. *Double Hashing.* You have two hash functions anyway, put them to work!
3. *Separate Chaining.* This is the most challenging as it requires making use of an additional data structure[2] but it is also how many hash table implementations actually work so worth understanding.

During the marking of this part you will be asked to discuss the asymptotic algorithmic complexity of your function $find$, and the potential problems that linear and quadratic probing may cause with respect to clustering of the elements in a hash table. **You may be asked these questions even if you did not implement these collisions resolution methods.**

Part b: Binary Search Tree Lookup

In this part you need to edit the program stub in your chosen language for binary search trees. You should:

1. Complete the insertion function by comparing the value to insert with the existing value. What should you do if they are equal? Hint: this is representing a set.
2. Complete the find function. Importantly, it should follow the same logic as insertion.
3. Extend the code to record statistics such as the height and the average number of comparisons per insertion or find. This will require some extra fields in the class (Java, Python) or node struct (C).

[1] One can also get memory leaks in memory managed languages but these are more semantic e.g. preserving a reference to something you will never use again.

[2] You don't need to write this yourself. In Java and Python you can use the standard library. In C you can use an implementation you find online.

Once you have done this, submit to COMPjudge and check the tests.

In this lab exercise we will stop there with trees. However, it is worth noting that this implementation is not optimal. What will happen if the input dictionary is already sorted? In the next lab we will be exploring self-balancing trees.

Testing

Once you have implemented hash sets and binary trees you should test them. We have provided a test script and some data. For example, to run simple tests for your hash set implementation in modes 0, 1, and 2 if your language is Java you should run

```
./python3 test.py simple hashset java 2
```

You might want to edit the script to do different things. We have also provided some `large` tests (replace `simple` by `large` above) which will take *a lot* longer to run (see help box below). You will need to unzip the files by running `unzip henry.zip` in the `data/large` directory. These large tests should give you an insight into the relative performance of the different techniques.

Failing Tests:
If the tests are failing there are two possible explanations. Either your implementation is wrong or the test script is being more picky than you thought.

For the first reason, make sure you understand what you expect to happen. Create a small dictionary and input file yourself and test using these. Remember that the program should print out all of the words that are in the input file but not in the dictionary. For binary search make sure that you are searching using the same order that you sort with. Make sure that your code is connected up properly in `find` so that it returns true/false correctly.

For the second reason, make sure that you don't print anything extra at verbosity level 0. If you look at `test.py` you'll see that what it's doing is comparing the output of your program between "Spellchecking:" and "Usage statistics:" against some expected output. It runs the program at verbosity level 0 and expects that the only spelling errors are output in-between those two points. See the note below for C programs for an extra thing to check.

You should use the relevant `submit` command to upload your work to GitLab and tag it appropriately. This provides a way for TAs to see your code and is a good backup — there are a few cases every year where students lose code that could have been avoided if they had kept their work up to date on GitLab. When you do this, it will be picked up by COMPjudge for an online version of these tests — see Blackboard for details on how to access these; they will be used as part of assessment later.

Part c: Making a Comparison

You should now perform an experimental evaluation like the one you did for Lab 2. Ideally, this should compare at least one implementation of dynamic arrays (with or without sorting) either using our own implementation from Lab 1 or the model solutions, your implementation of binary trees and at least one implementation of hash sets.

You should address the question:

> Under what conditions are different implementations of the dictionary data structure preferable?

Note that for hash set the initial size of the data structure will now play a larger role. You may find it useful to correlate your findings with statistics such as the number of collisions in the hash table.

We recommend you read, if you have not already done so, the Lab 2 instructions on how to generate inputs for your experiments. You are also encouraged to reuse the inputs you generated as part of that lab as part of your evaluation here.

Write-Up. It is important to keep notes on your experimental design and results. Ideally, these would then be written up as a detailed and well-argued LaTeX report. However, we would like you to focus your time on doing the evaluation rather than writing about it.

Therefore, we ask you to complete the provided *report.txt* document. **This includes some questions to answer before you start**. If you wish to turn this into a LaTeX document with graphs etc then this is a bonus but not required.

Note that it is *more* important to write clearly about your experimental design (justifying your decisions) than it is to write about your results and your analysis of them. The answers to the wrong question are useless.

Instructions for C Solutions

If you intend to provide a solution in C you should work in the `c` directory of the COMP26120_2020 repository. All program stubs and other support files for C programs can be found in this directory.

The completed programs will consist of several ".c" and ".h" files, combined together using *make* (the makefile covers Labs 1-3). You are given a number of complete and incomplete components to start with:

- *global.h* and *global.c* - define some global variables and functions
- *speller.c* - the driver for each spell-checking program, which:
  1. reads strings from a dictionary file and *insert*s them in your data-structure
  2. reads strings from a second text file and *find*s them in your data-structure

     - if a string is not found then it is assumed to be a spelling mistake and is reported to the user, together with the line number on which it occurred
- *set.h* - defines the generic interface for the data-structure
- *hashset.h* and *hashset.c* - includes a partial implementation for hash sets that you need to complete in Part a.
- *bstree.h* and *bstree.c* - includes a partial implementation for binary search trees that you need to complete in Part b.

Fix: Please note that there is a small error in *speller.c* that will cause the *test.py* script to fail. On line 219 the print statement should be outside of the verbosity check so that it always prints. Please move this before using *test.py*.

Note: The code in *speller.c* that reads words treats any non-alphabetic character as a separator, so e.g. "non-alphabetic" would be read as the two words "non" and "alphabetic". This is intended to extract words from any text for checking (is that "-" a hyphen or a subtraction?) so we must also do it for the dictionary to be consistent. This means that your code has to be able to deal with duplicates i.e. recognise and ignore them. For example, on my PC */usr/share/dict/words* is intended to contain 479,829 words (1 per line) but is read as 526,065 words of which 418,666 are unique and 107,399 are duplicates.

Running your code

To test your implementation, you will need a sample dictionary and a sample text file with some text that contains the words from the sample dictionary (and perhaps also some spelling mistakes). You are given several such files in the `data` directory of the COMP26120_2020 repository, and you will probably need to create some more to help debug your code. These files will need to be copied to the directory where you are working or you will need to set your $PATH$ so that your program can be executed from anywhere.

You should also test your program using a larger dictionary. One example is the Linux dictionary that can be found in */usr/share/dict/words*.

Compile and link your code using *make hashset* (for part a), or *make bstree* (for part b). These will create executables called *speller_hashset* and *speller_bstree* respectively.

When you run your spell-checker program, you can:

- use the $-d$ flag to specify a dictionary.
- (for part a) use the $-s$ flag to specify a hash table size: try a prime number greater than twice the size of the dictionary (e.g. 1,000,003 for */usr/share/dict/words*).
- use the $-m$ flag to specify a particular mode of operation for your code (e.g. to use a particular hashing algorithm). You can access the mode setting by using the corresponding mode variable in the code you write. Note that the modes you should use have already been specified in existing header files.
- use the $-v$ flag to turn on diagnostic printing, or $-vv$ for more printing (or $-vvv$ etc. - the more "v"s, the higher the value of the corresponding variable *verbose*). We suggest using this verbose value to control your own debugging output.

e.g.:

```
speller_hash -d /usr/share/dict/words -s 1000003 -m 2 -v sample-file
```

Instructions for Java Solutions

If you intend to provide a solution in Java you should work in the `java` directory of the COMP26120_2020 repository. The completed programs will form a Java package called `comp26120`. You can find this as a sub-directory of the `java` directory. All program stubs and other support files for Java programs can be found in this directory.

You are given a number of complete and incomplete components to start with:

- *GetOpt.java*, *LongOpt.java* and *MessagesBundle.properties* - control command line options for your final program. You should not edit these.
- *speller_config.java* - defines configuration options for the program.
- *speller.java* - the driver for each spell-checking program, which:
  1. reads strings from a dictionary file and *insert*s them in your data-structure
  2. reads strings from a second text file and *find*s them in your data-structure
     - if a string is not found then it is assumed to be a spelling mistake and is reported to the user, together with the line number on which it occurred

- *set.java* - defines the generic interface for the data-structure
- *set_factory.java* - defines a factory class to return the appropriate data structure to the program. This will be used in later labs.
- *hashset.java* -includes a partial implementation for hash sets that you need to complete in Part a.
- *speller_hashset.java* - sub-classes *speller* for use with *hashset*.
- *bstree.java* - includes a partial implementation for binary search trees that you need to complete in Part b. You will need to edit this.
- *speller_bstree.java* - sub-classes *speller* for use with *bstree*.

Note: The code in *speller.java* that reads words treats any non-alphabetic character as a separator, so e.g. "non-alphabetic" would be read as the two words "non" and "alphabetic". This is intended to extract words from any text for checking (is that "-" a hyphen or a subtraction?) so we must also do it for the dictionary to be consistent. This means that your code has to be able to deal with duplicates i.e. recognise and ignore them. For example, on my PC */usr/share/dict/words* is intended to contain 479,829 words (1 per line) but is read as 526,065 words of which 418,666 are unique and 107,399 are duplicates.

Running your code

To test your implementation, you will need a sample dictionary and a sample text file with some text that contains the words from the sample dictionary (and perhaps also some spelling mistakes). You are given several such files in the `data` directory of the COMP26120_2020 repository, and you will probably need to create some more to help debug your code. These files will need to be copied to the `java` directory or you will need to set your $CLASSPATH$ so that your program can be executed from anywhere.

You should also test your program using a larger dictionary. One example is the Linux dictionary that can be found in */usr/share/dict/words*.

Compile your code using *javac *.java*. This will create an executable called *speller_bstree.class* (for Part a) and *speller_hashset.class* (for Part a). To run your program you should call `java comp26120.speller_bstree sample-file` and `java comp26120.speller_hashset sample-file` respectively. Note that you will either need to set your $CLASSPATH$ to the `java` directory of the COMP26120_2020 repository or call `java comp26120.speller_bstree sample-file` (resp. `java comp26120.speller_hashset sample-file`) in that directory.

When you run your spell-checker program, you can:

- use the $-d$ flag to specify a dictionary.
- (for part a) use the $-s$ flag to specify a hash table size: try a prime number greater than twice the size of the dictionary (e.g. 1,000,003 for */usr/share/dict/words*).
- use the $-m$ flag to specify a particular mode of operation for your code (e.g. to use a particular sorting or hashing algorithm). You can access the mode setting by using the corresponding mode variable in the code you write. Note that the modes you should use have already been specified in existing header files.

- use the $-v$ flag to turn on diagnostic printing, or $-vv$ for more printing (or $-vvv$ etc. - the more "v"s, the higher the value of the corresponding variable *verbose*). We suggest using this verbose value to control your own debugging output.

e.g.:

java comp26120.speller_hashset -d /usr/share/dict/words -s 1000003 -m 2 -v sample-file

Instructions for Python Solutions

If you intend to provide a solution in Python you should work in the `python` directory of the COMP26120_2020 repository. All program stubs and other support files for Python programs can be found in this directory. You will need to use Python 3.

You are given a number of complete and incomplete components to start with:

- *config.py* - defines configuration options for the program.
- *speller.py* - the driver for each spell-checking program, which:
  1. reads strings from a dictionary file and *insert*s them in your data-structure
  2. reads strings from a second text file and *find*s them in your data-structure
     - if a string is not found then it is assumed to be a spelling mistake and is reported to the user, together with the line number on which it occurred
- *set_factory.py* - defines a factory class to return the appropriate data structure to the program. This will be used in later labs.
- *hashset.py* - includes a partial implementation for binary search trees that you need to complete in Part a. You will need to edit this.
- *speller_hashset.py* - front end for use with *hashset* which then calls the functionality in *speller.py*.
- *bstree.py* - includes a partial implementation for binary search trees that you need to complete in Part b. You will need to edit this.
- *speller_bstree.py* - front end for use with *bstree* which then calls the functionality in *speller.py*.

Note: The code in *speller.py* that reads words treats any non-alphabetic character as a separator, so e.g. "non-alphabetic" would be read as the two words "non" and "alphabetic". This is intended to extract words from any text for checking (is that "-" a hyphen or a subtraction?) so we must also do it for the dictionary to be consistent. This means that your code has to be able to deal with duplicates i.e. recognise and ignore them. For example, on my PC */usr/share/dict/words* is intended to contain 479,829 words (1 per line) but is read as 526,065 words of which 418,666 are unique and 107,399 are duplicates.

Running your code

Important: To run the provided program stubs in python2.7 (which is supplied on the CS provided virtual machine) you will need to install the enum34 package. You can do this from the UNIX command line.

To test your implementation, you will need a sample dictionary and a sample text file with some text that contains the words from the sample dictionary (and perhaps also some spelling mistakes). You are given several such files in the `data` directory of the COMP26120_2020 repository, and you will probably need to create some more to help debug your code. These files will need to be copied

to the `python` directory or you will need to set your $PYTHONPATH$ so that your program can be executed from anywhere.

You should also test your program using a larger dictionary. One example is the Linux dictionary that can be found in */usr/share/dict/words*.

To run your program you should call `python3 speller_hashset.py sample-file` (where $sample-file$ is a sample input file for spell checking) for Part a and `python3 speller_bstree.py sample-file` for Part b. Note that you will either need to set your $PYTHONPATH$ to the `python` directory of the COMP26120_2020 repository or call `python3 speller_hashset.py sample-file` (resp. `python3 speller_bstree sample-file`) in that directory.

When you run your spell-checker program, you can:

- use the $-d$ flag to specify a dictionary.
- (for part a) use the $-s$ flag to specify a hash table size: try a prime number greater than twice the size of the dictionary (e.g. 1,000,003 for */usr/share/dict/words*).
- use the $-m$ flag to specify a particular mode of operation for your code (e.g. to use a particular sorting or hashing algorithm). You can access the mode setting by using the corresponding mode variable in the code you write. Note that the modes you should use have already been specified in existing configuration files.
- use the $-v$ flag to turn on diagnostic printing, or $-vv$ for more printing (or $-vvv$ etc. - the more "v"s, the higher the value of the corresponding variable *verbose*). We suggest using this verbose value to control your own debugging output.

e.g.:

```
python3 speller_hashset.py -d /usr/share/dict/words -s 1000003 -m 2 -v sample-file
```

Marking Scheme

This coursework is worth 10% of your final mark for COMP26120. This means each mark in this mark scheme is worth 0.5% of your final mark for the module.

Part a. Has the basic hash table been implemented including at least one suitable hash function and insertion/lookup using linear probing? (Maximum 3 marks)	
The hash table is properly initialised. The hash function is sensible (unusual hash functions should be carefully justified). Insertion and lookup using linear probing has been implemented correctly (ignores duplicates, performs insertions when it should, always finds things that are in the table). Statistics (including number of collisions) are reported.	3
As above but there is a small flaw e.g. failure to detect duplicates or statistics missing	2
As above but a key component (e.g. the hash function) does not work as required or is far from optimal. For example, if the hash function does not produce a value in the required range or lookup using linear probing terminates too early.	1
No attempt has been made	0

Part a. Has a second hash function been given that is different from the first? (Maximum 1 mark)	
There is a second hash function and the student can explain how it works	1
A second hash function exists but it is either flawed or cannot be explained	0.5
No attempt has been made	0

Part a. Has rehashing been implemented? (Maxiumum 1 mark)	
Rehashing has been implemented and works correctly	1
An attempt has been made to implement rehashing but it is flawed.	0.5
No attempt has been made	0
Part a. Does the student understand the complexity of hash table operations and the issues related to probing? (Maximum 2 marks)	
The student can explain the complexity of the insert and find functions. The student can discuss potential problems that linear and quadratic probing may cause with respect to clustering of the elements in a hash table.	2
The student can almost explain all of the concepts	1.5
The student can explain either the complexity or the problems with probing but not both.	1
No attempt has been made	0
Part b. Has the implementation of binary search tree been completed completely and correctly? Note that code quality is addressed later. (Maximum 2 marks)	
Insertion uses a suitable ordering on values to create an ordered binary tree. Duplicates are handled properly. Find uses the same ordering to correctly identify whether a value is in the tree. Appropriate statistics are computed and presented.	2
As above but statistics are missing or inappropriate	1.5
As above but duplicates are not detected. However, the student can explain how to detect duplicates AND what the negative impact of not detecting duplicates is.	1.5
As above but duplicates are not detected	1
Some of the functionality is missing or does not work e.g. sometimes values are not inserted or sometimes find does not detect if a value is in the tree.	0.5
No attempt has been made	0
Part b. The student understands how ordered binary trees work and the complexity of the associated algorithms. (Maximum 2 marks)	
The student can give the complexity of insert and find in an ordered binary tree and explain *why* this is the case. The student can compare this to a similar setting in linked lists. The student can explain what a balanced tree is and what impact the structure of the tree has on find.	2
The student can state the complexities of the different operations but struggles to explain why these hold.	1
The student can attempt an explanation but the topic needs to be revised.	0.5
No attempt has been made	0
Part c. Has the experimental analysis been well-designed? (Maximum 2 marks)	
There is evidence that the student has carefully considered how best to answer the stated question. The choice of conditions to vary have been justified. The student discusses what they expect to find and links this to the theoretical complexities of the different approaches. It is easy to read.	2
Design decisions are justified but there is no reference to the theoretical complexities of the different approaches The design is described without justification of the choices made	1
Part of the design is described but it is incomplete	1
No attempt made	0

Part c. Have the results of the experimental analysis been well-presented with thoughtful discussion and conclusions drawn? (Maximum 3 marks)	
The results are clearly presented and explained.	3
There may be some discussion but no conclusions are drawn or conclusions are not supported by the data given.	2
The results are given without explanation or discussion.	1
The student cannot explain what they did or how they reached their conclusions when questioned.	0
Extension. Has the hash table been improved with (i) quadratic probing and/or (ii) double hashing, has the alternative (iii) separate chain- ing method been implemented? Can the student explain how they work? (Maximum 3 marks)	
All three have been implemented and work correctly. The student can explain how they work.	3
Two have been implemented and work correctly. The student can explain how they work.	2
Two or three have been implemented correctly but the student struggles to explain how they work sufficiently.	1.5
One has been implemented, works correctly, and can be explained.	1
An attempt has been made but they do not work correctly.	0.5
No attempt has been made	0
Is the code of good quality and does it handle errors properly? Note that many major errors are handled in the provided code. (Maximum 1 mark)	
The quality of the code is good throughout with reasonable names for variables/functions, sufficient comments, and a sensible layout. All obvious errors are handled appropriately.	1
As above but there are a few minor issues	0.75
As above but there are a few significant issues e.g. zero comments, completely illogical variable/function names	0.5
The quality of the code is poor throughout	0