COMP26120
Academic Session: 2022-23

Lab Exercise 3: Spellchecking (Better Trade-offs)

1. 问你一些关于你的实现的问题，包括你希望我们标记的提交的哈希值和标签。包括你希望我们标记的提交的哈希值和标签（详情见下文）。
2. 要求你上传一个你工作过的python、c或java文件夹的压缩文件。
3. 要求你上传一份关于你将在本实验室C部分进行的实验的PDF报告。

Duration: (almost) 4 weeks

You should do all your work in the lab3 directory of the COMP26120_2022 repository - see Blackboard for further details. You will need to make use of the existing code in the branch as a starting point.

Important: You submit this lab via a quiz on Blackboard. This will:

1. Ask you some questions about your implementation, including the **hash and tag** of the commit you want us to mark (see below for details of this).
2. Ask you to upload a zip file of the python, c or java folder you have been working in.
3. Ask you to upload a PDF report of the experiments you will run in Part C of this lab.

You can save your answers and submit later so we recommend filling in the questions for each part as you complete it, rather than entering everything at once at the end.

Code Submission Details

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.

Because people had a number of issues with GitLab last year we are going to take a multiple redundancy approach to submission of code. This involves both pushing and tagging a commit to GitLab and uploading a zip of your code to Blackboard. By preference we will mark the code you submitted to GitLab but if we can't find it or it doesn't check out properly then we will look in the zip file on Blackboard. Please **do both** to maximise the chance that one of them will work.

When you submit the assignment through Blackboard you will be asked for the *hash* and *tag* of the commit you want marked. This is to make sure the TAs can identify exactly which GitLab commit you want marked. You tag a commit **lab3_solution** (we recommend you use this tag, but you do not have to) by typing the following at the command line:

```
git tag lab3_solution
git push
git push origin lab3_solution
```

You can find the hash of your most recent commit by looking in your repository on GitLab as shown in figure 1.

You can also find the hash for a previous commit by clicking on the "commits" link and then identifying the commit you are interested in. This is shown in figure 2.

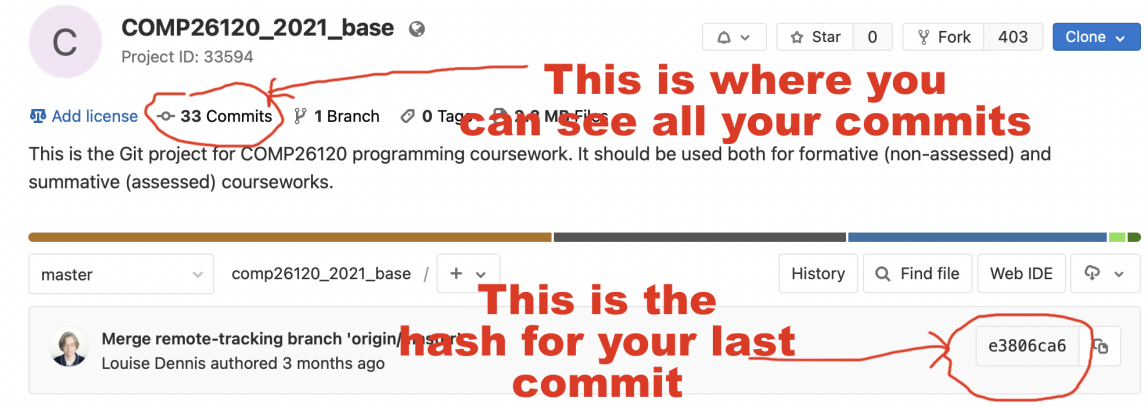

Figure 1: Identifying the hash of your most recent commit in GitLab

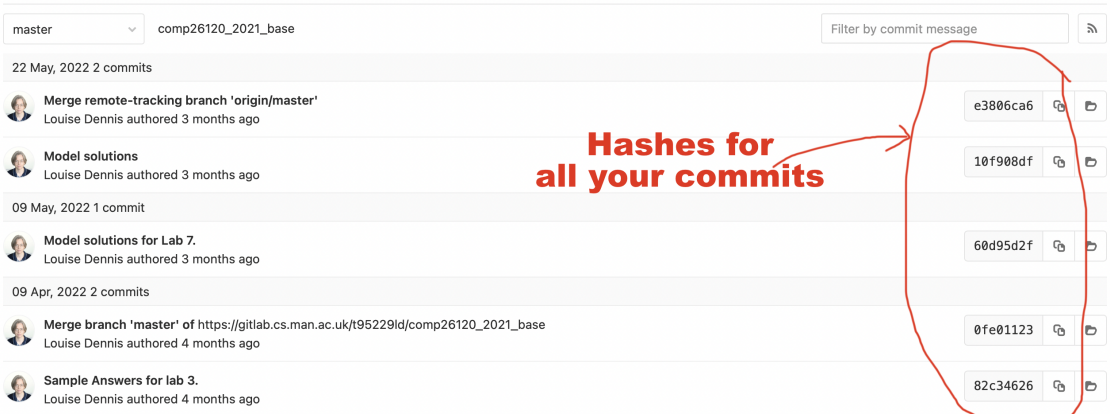

Figure 2: Identifying the hashes of previous commits in GitLab

For this assignment, you can only get full credit if you do not use code from outside sources. Built-in libraries in your chosen language are allowed but bear in mind that we ask you to implement hash sets - simply using the Java HashMap class (for instance) will not get you the marks for implementation.

Note on extension activities: The marking scheme for this lab is organised so that you can get over 75% without attempting any of the extension activities. These activities are directed at those wanting a challenge and may take considerable extra time or effort. This lab is designed to take up 8 hours of student time spread over four weeks – if you have not achieved the extension activities after you have put 8 hours of work into this lab then we recommend that you do not attempt them.

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. Please also **do not include** files containing dictionaries and queries that you have generated – the size of these tends to cause issues with marking.

While it is fine to discuss this coursework with your friends and compare notes, the work submitted **should be your own**. In particular this means you should not have copied any of the source code, or the report. We will be using the turnitin tool to compare reports for similarities.

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
- Evaluate experimentally the behaviour of hash sets and binary search trees.

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.

This exercise builds on work in Labs 1 and 2. Even if you have not done these labs we recommend you take a look at them and their model solutions in order to familiarise yourself with the spell-checking program and our expectations for creating and writing up experiments.

Getting the Support Code

You should then be able to see a `lab3` folder in your Gitlab that contains all the support files for this lab.

Data Structures

The spell-checking program stores a dictionary of words using a *Set* datatype. There are three data structures used to implement this *Set* datatype in this lab. In Lab 1 we introduced this datatype but we repeat that information here but you may also need to look online (e.g. the Wikipedia page) to complete your knowledge.

Set. This is an abstract datatype that is used to store the dictionary of words. The operations required for this application are:

- `find` whether a given value (i.e. string, alphabetic word) appears in the set.
- `insert` a given value in the set. Note that there is no notion of multiplicity in sets - a value either appears or it does not. Therefore, if `insert` is called with a duplicate value (i.e. it is already in the set) it can be ignored.

There would usually also be a `remove` function but this is not required for this application. We also include two further utility functions, which we ask you to implement, that are useful for printing what is happening :

- `print_set` to list the contents of a Set.
- `print_stats` to output statistical information to show how well your code is working.

You should have already used a *dynamic array* to implement this datatype 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 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

This part of the lab should take you between one and two weeks to complete (i.e., 3-4 hours).

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.

In this part you need to edit the program stub in your chosen language for hash sets. You should:

1. Complete the *insert* function for hash sets. What should you do if the value to be inserted already exists? Hint: this is representing a set.
2. Complete the *find* function. Importantly, it should follow the same logic as insertion.
3. Implement the *print_set* function – This should print out a sensible representation of the hash set when called (for instance when running the program with the $-vv$ flag)
4. Implement the *print_stats* function – This should print out a sensible set of statistics, such as average collisions per insert, that can be used to understand how your program is performing. This will require some extra fields in the class (Java, Python) or node struct (C).

You can find a discussion of how to test your code in the Section of these instructions on Testing (after Part b).

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, but one should be reasonable otherwise you are likely to have issues when performing an experimental comparison in Part c. These can be accessed by *modes* already given in the program stubs and more details of these are contained in the language specific instructions, do not change these since they are used in our testing processes. Write your code so that you can use the $-m$ parameter, which sets the *mode*.

Initially, you should **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. **Implement this resize and rehash functionality. When the program is called using the** $-vvv$ **flag it should print out a message when it rehashes and then call the** *print_stats* **function to show the new state of the hash set once rehashing completes**. In C, beware of memory leaks![1]

Mod of negative numbers:
Most of you are likely to use the mod operator in your hash functions by doing something like

```
h = x % size
```

to try to get a value of `h` that is between 0 and `size - 1`.

If `x` is negative then the behaviour of this operator varies by language: in particular in C and Java it will return a negative number. See the discussion at https://torstencurdt.com/tech/posts/modulo-of-negative-numbers/

You might think that `x` cannot be negative. However, if you are adding or multiplying large numbers together (e.g., when hashing a string) then you are likely to get integer overflow. In hashing, we don't care about such overflows as it is deterministic and just adds to the 'chaos' a bit but Java, for instance, may throw `ArrayIndexOutOfBoundsException` if a negative number is passed to the hash data structure, if you have not accounted for this possibility in your implementation.

In C you could use an unsigned integer to make sure that you only have positive numbers.

Once you have completed this implementation you should look at the Blackboard submission form where you will be asked the following questions about Part a.

1. What did you implement as your first hash function? How does it work? (No more than 100 words)
2. How do find and insert with linear probing work? Illustrate your explanation with a sample of your code for insert - this may be tidied up a bit for presentation but should clearly be the same as the code you have submitted. (no more than 200 words, not counting the code snippet).
3. What did you implement as your second hash function? How does it work? (No more than 100 words)
4. Did you implement rehashing? If so briefly explain your implementation including explaining *when* you choose to rehash. (No more than 200 words, you may include code snippets which do not count towards the word count)

Part b: Binary Search Tree Lookup

This part of the lab should take you a week to complete (i.e., 2 hours).

[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.

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

1. Complete the *insert* function by comparing the value to insert with the existing value.
2. Complete the *find* function. Importantly, it should follow the same logic as insertion.
3. Implement the *print_set* function – This should print out a sensible representation of the hash set when called (for instance when running the program with the $-vv$ flag)
4. Implement the *print_stats* function – This should print out a sensible set of statistics, such as the height and average number of comparisons per insert and find.

Once you have completed this implementation you should look at the Blackboard submission form where you will be asked the following question about Part b.

1. How do find and insert for binary trees work? Include your code for insert and relate your explanation to this code. (No more than 200 words, not including the code snippet)

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

There are instructions in the individual language sections explaining how to test your algorithm on a single example and you should try that first. Once that is done you can test them on some test suites.

We have provided a test script, `test.py` in the top level directory of the COMP26120_2022 repository and some data in the `data` directory. `test.py` takes four arguments: the first is the language you are using; the second is the test suite (`simple` is provided but you can create more); the third is the data structure (`bstree` or `hashset`) you are testing; and the fourth is the mode you want to test.

For example, to run the simple tests for mode 2, your language is Java and you want to test your binary tree implementation, you should run

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

You might want to edit the script to do different things but we will use the one provided when marking your code so make sure it runs with this. 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.

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 or use one of the individual simple tests. There are nine of these and you will find them organised in the `data/simple` directory where each sub-directory contains an example dictionary, input file and the answer expected by the test. Most of these are small and easy to understand. Remember that the program should print out all of the words that are in the input file but not in the dictionary. 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.

As part of marking, we will run your implementations on the `simple` test suites and may also run them on the `large` test suite if we feel additional testing is required.

Part c: Making a Comparison

This part of the lab should take you one to two weeks to complete (i.e., 3-4 hours).

You should now perform an experimental evaluation like the one you did for Lab 2. We want to 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 may also (if possible) reuse the inputs you generated as part of that lab as part of your evaluation here in order to save time. Note that generation of inputs is likely to take some time (hours for large dictionaries and queries), so you should plan your work so you can set a generation script running and then leave it while you do something else.

You should perform two experiments for your evaluation and draw a conclusion. You can then devise and run a third optional (set of) experiments as an extension exercise.

The two experiments you should perform should answer the questions:

1. Does your implementation of insert for hash sets work as expected in terms of complexity. You should consider only one of your hash functions, with linear probing.

2. Does your implementation of insert for binary trees work as expected in terms of complexity. This should consider both average and worst case.

Your presentation of each experiment should have four sections – these are likely to be very similar for each experiment:

Hypothesis You should state, as a hypothesis, what you expect the performance to be and then write a short paragraph explaining why you believe this to be the case, based on the theoretical complexities of insert for the data structure. (This should be about 200 words and take up **no more than** half a page in your report).

Design You should describe how you designed the experiment. This should include:

- The input files you used and why;
- How you reduced the chance that some randomly generated input had properties that were not typical of the average case;
- The command line call(s) you used or, if you created your own script for this, include the script as an appendix or give its name and include it in the code you submit to GitLab and the zip file you upload to Blackboard;
- Anything else that would help your marker judge how well you designed your experiment to test your hypothesis.

(This should be about 500 words and take up **no more than** a page in your report – not counting any scripts included as appendices).

Results You should present your results as a graph or a table. If you do any processing on results, such as generating best fit lines, computing averages, etc., then these should be described. Raw data should be presented in an appendix or included with your code submission, if any processing has taken place here. (This should be about 200 words and take up **no more than** half a page in your report – not counting graphs and tables).

Discussion You should briefly discuss whether your results confirm your hypothesis. If they do not confirm your hypothesis then you should briefly discuss any ideas you have for why they did not. (This should be about 200 words and take up **no more than** half a page in your report).

Note that it is *more* important to be able to clearly explain your experimental design (justifying your decisions) than it is to have excellent results. The answers to the wrong question are useless. Top marks can be obtained for Part c for a well-designed experiment that has disappointing results (either disproving a hypothesis or just being unclear in what they show), so long as the conclusions you draw make it clear you are aware of the issues with the results.

The final section of your report should be a **Conclusion** where you use your results to address the initial question: when should you use your hash set implementation and when should you use your binary tree implementation. You do not need to validate this experimentally since this is likely to take a lot of time, even with a good implementation on a fast machine.

Hints

- You should be able to do all of the analysis you need to do for this whole lab using a simple spreadsheet application. However the unix utility `gnuplot` and the python `matplotlib` library also both allow you to create plots from data and fit lines to those plots.
- As the numbers are all small you might want to count n in lots of 10k e.g. for an input of 10,000 say it's $n = 1$, for 20,000 say it's $n = 2$ etceteras. Alternatively, you may wish to measure time in milliseconds rather than seconds. This is just to make numbers look more sensible.
- You can use the UNIX `time` utility to measure running time. This gives you times for `real`, `user` and `sys`. The most accurate way to judge the running time of your algorithm is to take the sum or `user` and `sys` (`real` includes the time when other resources might be using your CPU and can't account for computation time across multiple cores).

- You may want to plot/calculate average time per insert, rather than plotting the total time taken to build the dictionary and query it for words. When doing this (particularly for Java and Python) be aware that there may be a fixed "start up" time that is independent of the size of dictionary or query – if this is the case and you are calculating averages you may see that the average time appears to get better as the dictionary size increases not worse as you would expect, particularly if you are using relatively small dictionaries and queries, where this start up time will have a bigger influence on the overall time taken. If this is happening, you may want to start by plotting a graph of total time and determining where this crosses the origin, in order to give you a guess at start up time, and then deduct this start up value from the total time before calculating the average. If you do this, make sure to document it in your report.

Extension Activities

Implementation Extensions. If you still have time, consider implementing alternative methods for dealing with collisions for your hash set. 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.
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.

As you implement these you should look at the Blackboard submission form where you will find the following questions:

1. What potential problems might linear probing cause with respect to the clustering of elements in a hash table?
2. How does quadratic probing work and what are its advantages and disadvantages? Include some of your code and relate your explanation to specific lines in the code. (no more than 200 words not counting code snippets)
3. How does double hashing work and what are its advantages and disadvantages? Include some of your code and relate your explanation to specific lines in the code. (no more than 200 words not counting code snippets)
4. How does separate chaining work and what are its advantages and disadvantages? Include some of your code and relate your explanation to specific lines in the code. (no more than 200 words not counting code snippets)

Report Extension. Include a third (set of) experiment(s) in your report. This experiment can be to explore any aspect of the practical complexity of your implementation of hash sets or binary trees that you wish – good things to look at include: query time; the other collision resolution strategies for hash sets (if you have implemented them); your other hash function; and the effect of rehashing on the performance of hash sets, but there are many opportunities here.

[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.

Instructions for C Solutions

If you intend to provide a solution in C you should work in the `lab3/c` directory of the COMP26120_2022 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*. You are given a number of complete and incomplete components to start with:

- *global.h* and *global.c* - define some global variables and functions, including the verbosity level.
- *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
  3. processes input flags to set the dictionary, query file, size, mode and verbosity level to be used.

  You should not need to change *speller.c* – if you do, make sure that the input flags for running the program continue to work as specified below.
- *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.

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_2022 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*). **Do not hard code the dictionary size into your code. You should let it be set by the *size* argument to the *initialize_set* function.**
- 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 *hashset.h*.
- 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. You can access the verbosity level as the global *verbose*. This is 0 by default and will be set to 1, 2 or 3 by use of the $-v$, $-vv$ and $-vvv$ flags.

e.g.:

```
speller_hashset -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 `lab3/java` directory of the COMP26120_2022 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 but you will need to copy in either your solution or the model solutions for Lab 1 into this directory – this should be the file `darray.java`.

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

- *speller_config.java* - defines configuration options for the program, including the verbosity level.
- *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
  3. processes input flags to set the dictionary, query file, size, mode and verbosity level to be used.

  You should not need to change *speller.java* – if you do, make sure that the input flags for running the program continue to work as specified below.
- *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.
- *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_2022 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_2022 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*). **Do not hard code the dictionary size into your code. You should let it be set by *config.size* field of the *config* object supplied as an argument to the *hashset* constructor.**
- 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 *hashset.java*.
- 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. We suggest using this verbose value to control your own debugging output. You can access the verbosity level as the *verbose* field of the hashset object. This is 0 by default and will be set to 1, 2 or 3 by use of the $-v$, $-vv$ and $-vvv$ flags.

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 `lab3/python` directory of the COMP26120_2022 repository. All program stubs and other support files for Python programs can be found in this directory but you will need to copy in either your solution or the model solutions for Lab 1 into this directory – this should be the file `darray.py`. You will need to use Python 3.

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

speller.py - 每个拼写检查程序的驱动，它的作用是：
1. 从一个字典文件中读取字符串，并将其插入你的数据结构中
2. 从第二个文本文件中读取字符串并在你的数据结构中找到它们
- 如果一个字符串没有被找到，那么它就被认为是一个拼写错误，并被报告给用户，同时报告它所发生的行号

- *config.py* - defines configuration options for the program, including the verbosity level.
- *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

  You should not need to change *speller.py* – if you do, make sure that the input flags for running the program continue to work as specified below.
- *set_factory.py* - defines a factory class to return the appropriate data structure to the program.
- *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 you will need to install the enum34 package. You can do this from the UNIX command line. We advise that you use Python 3 instead.

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_2022 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_2022 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 $/usr/share/dict/words$). **Do not hard code the dictionary size into your code. You should use *self.hash_table_size* which is created from the configuration options when the hash set object is created by *__init__*.**
- 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 *hashset.py*.
- 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. You can access the verbosity level as the *self.verbose* field of the hashset object. This is 0 by default and will be set to 1, 2 or 3 by use of the $-v$, $-vv$ and $-vvv$ flags.

e.g.:

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

Marking Rubric

This coursework is worth 20% of your final mark for COMP26120. This means each mark in this mark scheme is worth just 0.4% of your final mark for the module.

In the rubric below the marks an item is worth are slightly approximate due to the way Blackboard allocates percentages across sections. In general unsatisfactory performance will get you 10% of the available marks for a section, satisfactory will get you 50% and excellent will get you 100% but this can vary for items worth more than 1 mark because several criteria may be involved each of which can be marked unsatisfactory, satisfactory or excellent.

Note all that on Blackboard, most of the execution marks are given for the Code Upload question since the TA will run a bunch of tests on your code in order to determine how well it executes – we separate out these marks here into the sections related to specific parts of the implementation.

Code Submission (2 marks)

Description	Satisfactory	Excellent
Commit Hash and Tag (1 mark)	A markable submission can be identified (with a bit of work) using either the commit hash or tag provided, but either they point to different commits, only one is supplied, or neither directly identify a markable submission.	The commit hash and tag accurately identify to a markable submission

Part A

Basic implementation and first Hash Function (10 marks)

Description	Unsatisfactory	Satisfactory	Excellent
Execution (1 mark)	Code passes at least one of the simple tests when run with mode 0 (which is hash function 1 and linear probing)	Code passes more than half the tests when run on the simple tests with mode 0	Code passes all tests when called using `test.py` on the simple tests with mode 0.
Hash Function (2 marks)	There an attempt to describe a hash function but it doesn't appear to be related to the one in the implementation	There is a description and discussion of the implemented hash function, but it is difficult to understand or there are errors in the implementation	The implemented hash function is correct and is described and discussed clearly and concisely
Find and Insert (5 marks)	An attempt has been made to implement find and/or insert with linear probing, but there are many errors causing most test cases to fail or the description is non-existent or does not seem to relate to this implementation.	Find and Insert have been implemented but there are minor errors that mean it won't work on all cases or the discussion is flawed or difficult to follow	Find and Insert have been implemented correctly, using the same logic, and linear probing is clearly and concisely described.
Statistics (1 mark)	Some statistics are printed but they are incorrectly implemented	Some correctly calculated statistics are printed but there could be more to help in evaluating performance	Statistics are printed including number of collisions, rehashes and average collisions per access
Print Set (1 mark)	Something is printed out but it isn't clear how it relates to the current state of the hash set.	The current states of the hash set is printed but is difficult to interpret.	A nice, easy to understand, representation of the hash set is printed.

Second Hash Function (3 marks)

Description	Unsatisfactory	Satisfactory	Excellent
Execution (1 mark)	Code passes at least one of the simple tests when run with mode 4	Code passes more than half the tests when run on the simple tests with mode 4.	Code passes all tests when called using `test.py` on the simple tests with mode 4.
Hash Function (2 marks)	There an attempt to describe a hash function but it doesn't appear to be related to the one in the implementation	There is a description and discussion of the implemented hash function, but it is difficult to understand or there are errors in the implementation	The implemented hash function is correct and is described and discussed clearly and concisely

Rehashing (2 marks)

Description	Unsatisfactory	Satisfactory	Excellent
Execution (1 mark)	-	Code passes simple tests when run using a small initial dictionary size (e.g., 3 or 5) but no information is printed out to judge whether rehashing is actually working as intended.	Code passes simple tests when run using a small initial dictionary size (e.g., 3 or 5). Enough information is printed to confirm that resizing is taking place sensibly.
Implementation & Discussion (1 mark)	There seems to be an attempt at implementing rehashing which is described but either the implementation is very flawed or the description very difficult to understand.	There is an implementation of rehashing that is described but either the implemenation has minor errors or isn't well thought through (e.g., it is called too eagerly or not eagerly enough) or the description is hard to follow, goes over length or omits details such as when rehashing is called.	The implementation of rehashing is correct and sensible and is clearly, concisely and correctly described.

Part B

Basic implementation (6 marks)

Description	Unsatisfactory	Satisfactory	Excellent
Execution (1 mark)	Code passes at least one of the simple tests when run with bstree	Code passes more than half the tests when run on the simple tests with bstree	Code passes all tests when called using `test.py` on the simple tests with bstree.
Find and Insert (3 marks)	An attempt has been made to implement find and/or insert, but there are many errors causing most test cases to fail or the description is non-existent or does not seem to relate to this implementation.	Find and Insert have been implemented but there are minor errors that mean it won't work on all cases or the discussion is flawed or difficult to follow	Find and Insert have been implemented correctly, using the same logic, and are clearly and concisely described.
Statistics (1 mark)	Some statistics are printed but they are incorrectly implemented	Some correctly calculated statistics are printed but there could be more to help in evaluating performance	Statistics are printed including the height of the tree and average comparisons per insert and find.
Print Set (1 mark)	Something is printed out but it isn't clear how it relates to the current state of the binary tree.	The current states of the binary tree is printed but is difficult to interpret.	An easy to understand (provided it is not too large) representation of the binary tree is printed.

Part C

Experiments 1 & 2 (7 marks each)

Description	Unsatisfactory	Satisfactory	Excellent
Hypothesis (2 marks)	The stated hypothesis and explanation appear unrelated to the question posed.	A sensible hypothesis is given but not all details are supplied (e.g., what n represents in $O(n)$ and similar formulae). The explanation for the hypothesis is unclear.	A sensible hypothesis is clearly stated and explained.
Design (2 marks)	A deeply flawed design, unlikely to yield results that can confirm or disprove the hypothesis	A reasonable design though with some flaws which might make results harder to interpret. Mention is made of what dictionaries and queries were generated, and what was measured.	A good design likely to confirm or disprove the hypothesis. It is clear what dictionaries and queries were generated, what was run and what was measured.
Results (1 mark)	Results are hard to understand or don't seem to relate to hypothesis and design.	Results are presented in a graph or table though there may be issues with missing labels, or there seems to have been some processing that isn't described. There is a description of what this graph or table show.	Results are presented in a clearly labelled table or graph. Any processing of raw data is clearly described. The description highlights key features of the results.
Discussion (1 mark)	There is a discussion of the results but either it is unclear whether the hypothesis is confirmed, disproved or it is equivocal or this is stated incorrectly.	There is a discussion of the results that correctly states whether the hypothesis has been confirmed, disproved or is equivocal but if the result was unexpected any attempt to hypothesis why is unclear or unlikely	There is a discussion of the results that correctly states whether the hypothesis has been confirmed, disproved or is equivocal. If there result was unexpected there is a good discussion of why that might be.

Conclusion (2 marks)

Description	Unsatisfactory	Satisfactory	Excellent
Conclusion (2 marks)	The conclusion drawn is difficult to understand or makes no sense in relation to the experimental results	A conclusion is drawn about when which technique is preferable, but it contains no or incorrect calculations or is otherwise vague or only loosely supported by the experimental results	A conclusion is drawn, giving convincing information or estimates about when which technique is preferable based on the experimental results.

Data and Scripts (1 mark)

Data and Scripts (1 mark)	Some attempt has been made to provide the information necessary to reproduce the experiments or validate the results	Enough information is given to either reproduce the experiments, or the raw data created for the results sections is provided (either in the results sections themselves, in an appendix, or as a file included in the commit/zip file) but some detail or some files seem to be missing.	All program calls, scripts used and raw data is clearly documented (in the report, appendices or as separate files as appropriate).

Extensions

Linear Probing Discussion(1 mark)

Description	Unsatisfactory	Satisfactory	Excellent
Discussion of Linear Probing (1 mark)	While the student knows how linear probing works, it isn't obvious that they appreciate the issues it can cause.	Issues with linear probing are discussed but the discussion is flawed, unclear or over-long.	Potential issues with linear probing are discussed clearly and concisely.

Alternative Implementations (3 marks each)

Description	Unsatisfactory	Satisfactory	Excellent
Execution (1 mark)	Code passes at least one of the simple tests when run with mode 1, 2 or 3 (as appropriate)	Code passes more than half the tests when run on the simple tests with mode 1, 2 or 3 (as appropriate).	Code passes all tests when called using `test.py` on the simple tests with mode 1, 2 or 3 (as appropriate).
Implementation (1 mark)	An attempt has been made to implement the collision resolution technique, but there are many errors causing most test cases to fail.	The technique has been implemented but there are minor errors that mean it won't work on all cases	The technique has been implemented correctly.
Discussion (1 mark)	The description is nonexistent or does not seem to relate to the implementation.	The discussion is flawed or difficult to follow	The technique is clearly and concisely described.

Third Experiment (4 marks)

Description	Unsatisfactory	Satisfactory	Excellent
Third Experiment	Some attempt is made at an experiment but it is poorly presented, and makes little sense	A well-structured presentation of an experiment, containing minor flaws or presentational issues, or lacking in novelty or ambition (for instance it is a simple variation on one of the previous experiments that requires no additional design work) or superficial discussions	An interesting and novel question is proposed and explored with a clear hypothesis, rigorous design, clear presentation of results, and a thorough interpretation of their meaning and ways forward drawn.