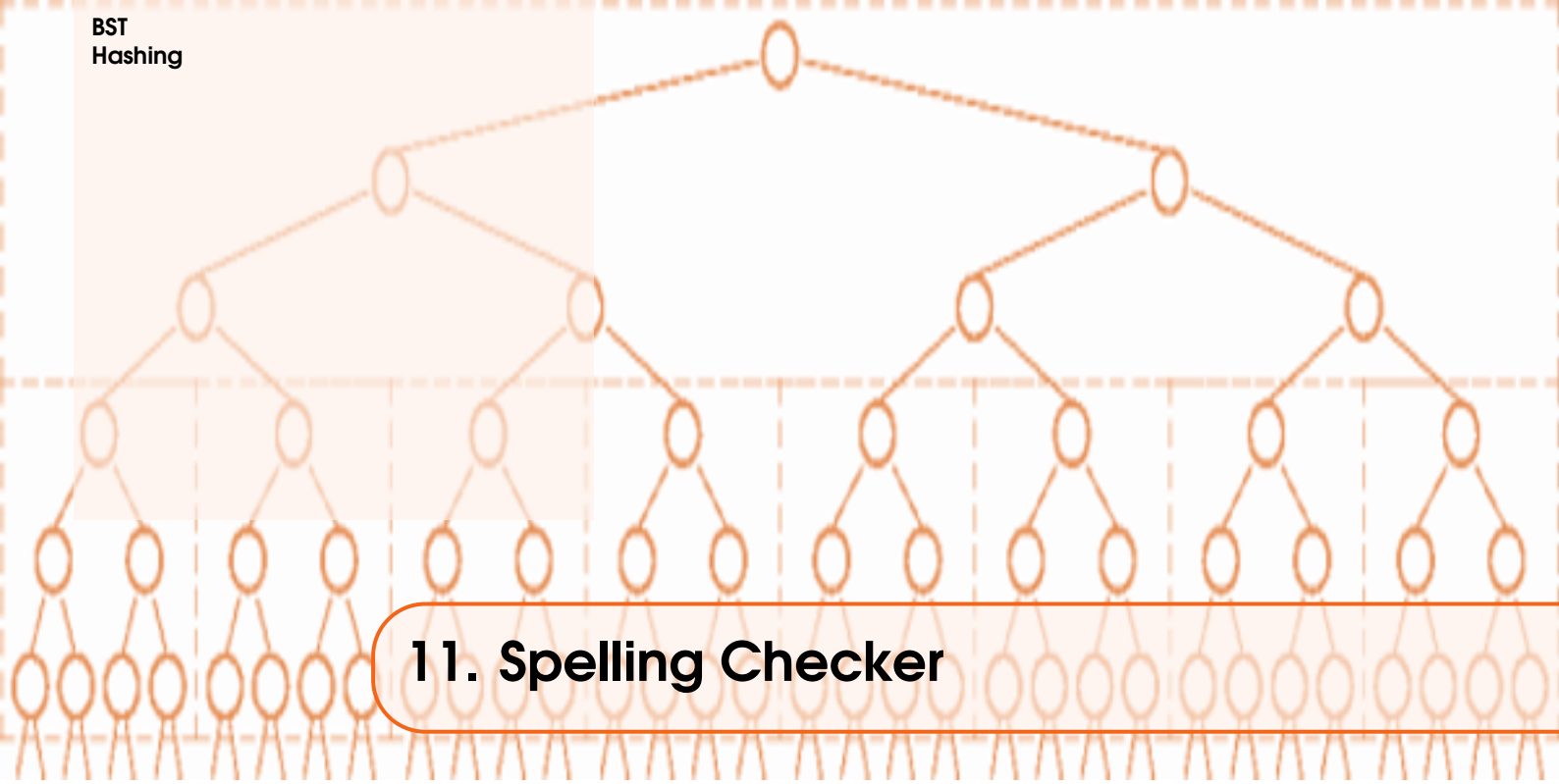

11. Spelling Checker

This assignment is about implementing an Abstract Data Type (ADT) for a Dictionary - which is simply a collection of unique words.

The interface for the ADT exposes four functions:

```
/* Create empty dic */
dic* dic_init(int size);

/* Add one element into the dic */
void dic_insert(dic* s, char* v);

/* Returns true if v is in the array, false elsewise */
bool dic_isin(dic* s, char* v);

/* Finish up */
/* Clears all space used, and sets pointer to NULL */
void dic_free(dic** s);
```

We will implement the Dictionary ADT using two 'rival' techniques - the Binary Search Tree (BST) and using Hashing.

One file is provided to test these two approaches. This is called `spl.c` and reads in a file of correctly spelt words (`argv[1]`) and then a list of words to check (`argv[2]`). The program then prints out all words in the second file that are incorrectly spelt.

Whilst I provide `spl.c`, you need to provide the dictionary implementation that this program uses. These are the BST approach `bst.c/bst.h` and the Hashing approach `hsh.c/hsh.h`. You will compile `spl.c` along with each of these two files in turn, to create two executables that perform the same task, but using different underlying methodologies. Much of the compilation is dealt with by the provided `Makefile`; this also specifies which header file (`bst.h` or `hsh.h`) to put at the very top line of `spl.c` using the `-include` compile option.

11.1 BST

The BST stores keys in an ordered manner ('smaller' to the left, and 'larger' to the right of the root node). How 'smaller' and 'larger' are defined is up to the implementation. No key may be stored twice (i.e. they are unique).

Exercise 11.1 Using the Dictionary you have implemented using BSTs, compile the spell checker `spl.c` and check that it works using the provided `Makefile`. The program is passed a dictionary file (`argv[1]`), and stores each word in a (dictionary) BST. The list of words (`argv[2]`) to be checked is read from file, and any misspelt word printed out and also added to a second (misspelt) BST. No misspelt word is printed twice, via use of this second BST.

```
./splbst eng_370k_shuffle.txt heart_darkness.txt
gravesend
marlow
deptford
erith
ravenna
fresleven
morituri
salutant
.
. ETC
.
```

■

11.2 Hashing

Exercise 11.2 Now provide another, alternative, implementation of the Dictionary ADT, using Hashing. Many of the details of how you do this are up to you; choices include whether you use open addressing, probing, chaining, and exactly which hash function(s) to use. However, you should make no assumptions about the maximum number of words to be stored.

Using the Dictionary you have implemented via Hashing, compile the spell checker `spl.c` and check that it works using the provided `Makefile`. The output should be identical to that shown above.

■

Hints

- Make sure your final submission consists of four files; `bst.c`, `bst.h`, `hsh.c` and `hsh.h`. Everything else, including `spl.c` and the `Makefile` is provided. Do not resubmit these - my versions will be used.
- The quality of your code is being assessed. I'll read it in detail, and also run more tests, other than those provided. Make sure your code clears up all memory allocated.
- If you know of any issues with your code (incomplete parts, bugs, leaks), put a comment about it at the top of `bst.c/hsh.c`.
- For reference, my `spl.c/hsh.c` file executes in around 0.3s on a virtual machine, whilst `spl.c/hsh.c` executes in around 0.1s. Speed is not the main factor here, though; readability of your code is.