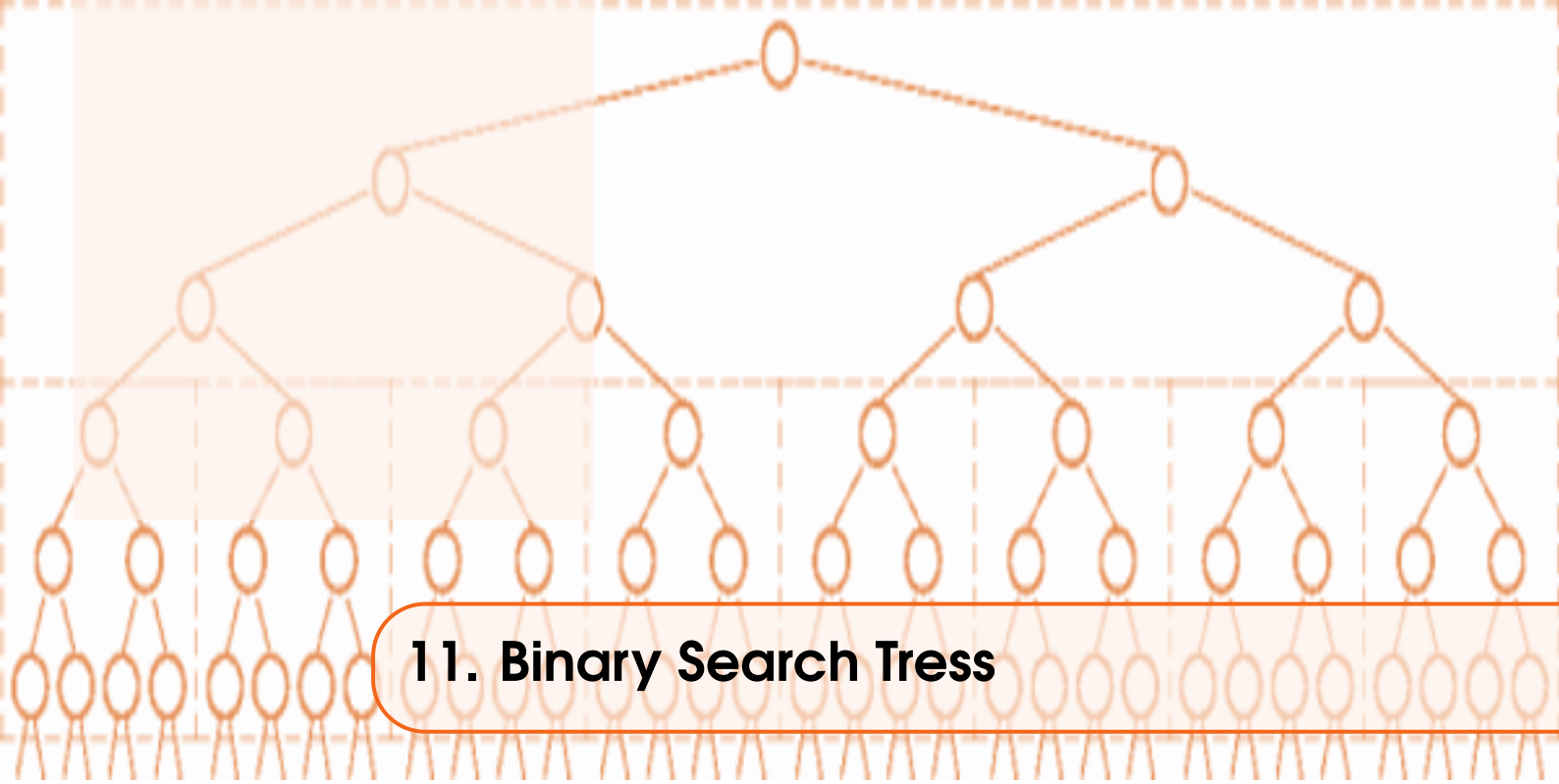

11. Binary Search Tress

This assignment is about implementing an Abstract Data Type (ADT) for Binary Search Trees (BSTs). 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 left to the user of the ADT to define. No key may be stored twice (i.e. they are unique).

The BST used here stores no particular type, but rather a block of memory via a `void` pointer. The way to compare two different blocks of memory, for comparison or sorting, is via a user-provided function, in a similar way to `qsort`.

The interface for this type is provided in the file `bst.h`. You will be writing `bst.c`. There is also a `testbst.c` to aid your testing process, along with a `Makefile` which can also check for memory leaks.

Exercise 11.1 **Basics**. Write `bst.c`, so that the functions :

```
bst* bst_init()
bool bst_insert()
int bst_size()
int bst_maxdepth()
bool bst_isin()
void bst_insertarray()
void bst_free()
```

work correctly. ■

50%

Exercise 11.2 **Spell Checker**. Using your BST implemented above, write a short program that reads in a dictionary file (`argv[1]`), and stores each word in a (dictionary) BST. The list of words (`argv[2]`) to be checked is read fom 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.

```
./spl eng_370k_shuffle.txt heart_darkness.txt
gravesend
marlow
```

```
deptford
erith
ravenna
fresleven
morituri
salutant
.
. ETC
.
```

■

10%

Exercise 11.3 **Simple BST Display**. Write the function `bst_print()` which returns a string representation of the BST in the form: `(root(left subtree)(right subtree))`. This can be done recursively, in a similar manner to that described in the course notes. However, those notes use a very slightly different format for the string, and do not take care of the memory leaks caused by repeated `calloc` calls without the use of `free`. A small example tree is shown below :

```
(M(K(I(D(F(G(H))))))(N))
```

■

10%

Exercise 11.4 **Ordered List**.

Write the function `bst_getordered()` which copies a *sorted* list of the elements in the tree, into an array provided by the user. The middle of this array will hold the *median* element of the tree. You could simply copy the elements into the array and `qsort()` them, or if you use an *In-order Traversal*:

www `https://en.wikipedia.org/wiki/Tree_traversal#In-order`

it's possible to copy the data without requiring the sort.

■

10%

Exercise 11.5 **Simple Rebalancing**.

It is well known that a BST, as implemented here, can become unbalanced if a pathological set of keys is given during the building process. One such example is if a sorted list of keys is presented during the building process. In this case, the tree becomes a simple linked list with search complexity $O(n)$, rather than the desired $O(\log_2 n)$. One (very simple) cure for this is to build a brand new tree, using a better ordering of the input keys. The optimal ordering is achieved by taking a sorted list of the keys, and using the middle key (the median of them) as the first element to insert into the new tree. You then recursively divide the list in two, and recursively add the medians of each of the two sub-lists. This is similar in many ways to the binary search.

Write the function `bst_rebuild()` which returns an optimally rebalanced tree. Generally, the maximum depth of a tree with n keys will decrease upon rebalancing to $\log_2(n)$.

■

10%

Exercise 11.6 **Extension**. If you have time, investigate some other aspect of BSTs that we haven't had time to look at. Some ideas include :

Self balancing trees.

- Threaded trees.
- 'Prettier' textual printing of trees.
- Comparing your spell checker performance with hashing.
- Using the DOT language for describing trees.
- Displaying the tree structure using SDL.
- A more efficient approach to memory allocation (the approach here calls calloc a huge number of times for very small chunks; this is bad for runtime and memory caching).

I'm looking for your understanding of your underlying concepts, as well as your coding ability. Write about it in an `extension.txt` file, provide a `Makefile` and as many other files as you like. Put them all into an `Extension` directory and zip them into a single file.

■

10%

Hints

- Make sure your final submission consists of a **maximum** of three files; `bst.c` and `spl.c` and, if you do an extension, a directory called `Extension` in a `.zip` or `.gz` file.
- If you do an extension, make sure there is a `extension.txt` file explaining what you have done, and a `Makefile` that will build and execute it by typing `make`.
- 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`.
- For reference, my `spl.c` file executes in around 0.5s on a virtual machine.