C Final Test

Imperial College London

Department of Computing

Doublets

Monday 10 June 2019
from 10am to 1pm
THREE HOURS
(including 10 minutes planning time)

- Please make your swipe card visible on your desk.
- After the planning time log in using your username as **both** your username and password.
- The maximum total is **30 marks**.
- Credit will be awarded throughout for clarity, conciseness, useful commenting, appropriate use of assertions and error checking.
- **Important:** THREE MARKS will be deducted from solutions that do not compile. Comment out any code that does not compile before you leave.
- The files can be found under the subfolder `doublets` inside the Home folder of Lexis.

Doublets

Doublets (also known as *Word Ladders*, *Word-Links*, *Laddergrams* and *Word Golf*) is a word game invented in the 1870s by Charles L. Dodgson, under his (more famous) pen-name Lewis Carroll[1].

The aim of the game is to form a *Chain* (or ladder) of words representing the transformation, in a finite number of steps, of a given *start word* into a given *target word* (whose length matches that of the start word). At each step, one (and only one) letter may be changed in the word from the previous step, provided that the new word so formed appears in an approved list of English words (which Dodgson published as a *glossary*). Words that have already appeared in previous steps are not permitted. Figure 1 shows pertinent extracts from the original rules.

---

1. The words given to be linked together constitute a "Doublet;" the interposed words are the "Links;" and the entire series a "Chain."

2. Each word in the Chain must be formed from the preceding word by changing one letter in it, and one only. The substituted letter must occupy the same place, in the word so formed, which the discarded letter occupied in the preceding word, and all the other letters must retain their places.

3. When three or more words are given to be made into a Chain, the first and last constitute the "Doublet." The others are called "Set Links," and must be introduced into the Chain in the order in which they are given. A Chain of this kind must not contain any word twice over.

4. No word is admissible as a Link unless it (or, if it be an inflection, a word from which it comes) is to be found in the following Glossary.

Figure 1: Extracts of the rules of Doublets as published in a March 1879 edition of *Vanity Fair*.

An example of a Chain in which the start word "WHEAT" is transformed into the target word "BREAD" in seven steps is shown in Figure 2. **It is conventional when presenting a Chain to display the start words and target words in uppercase while words in intermediate steps, i.e. the Links, are displayed in lowercase.**

W H E A T
c h e a t
c h e a p
c h e e p
c r e e p
c r e e d
b r e e d
B R E A D

Figure 2: Sample Chain from "WHEAT" to "BREAD".

[1] Works published by Dodgson using the same pen-name include *Alice in Wonderland* and the poem *Jabberwocky*.

Digital Trees

Digital Trees (also known as *Tries*, from the middle syllable of *re**trie**val*) are tree-like data structures optimised for search time, and are commonly used to store *dictionaries* of strings. Every node at each level of the Trie is associated with one alphabetic value and has up to as many references to other nodes as the alphabetic values for the chosen alphabet (e.g. 26, if the relevant alphabet is the English one).

The "children" of a node n share a common prefix, namely the string formed by the characters associated with the nodes on the path from the root node to n. In this sense, the root node represents the empty string.

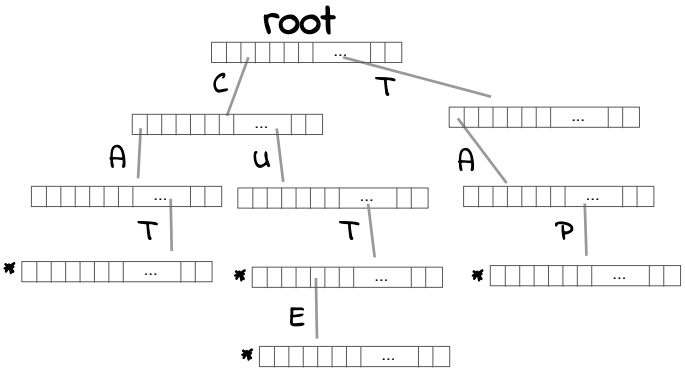

Figure 3: Example of a Trie containing the strings "CAT", "CUT", "CUTE" and "TAP". A star next to a node represents the end of a string at that node.

To signal the *end of a string* in the Trie, each node also holds a boolean value. When searching for a string in the Trie, the characters of the string define a path from a node to the next, starting from the root. The search is successful if and only if the last character of the string leads to a node whose boolean value is True. A simple example of Trie is given in Figure 3.

Your Task

You will implement a Digital Tree to hold the glossary for the game of Doublets. You will then use the glossary to implement an algorithm for generating a valid Chain whose length is less than or equal to some given maximum. You are given the data file `words.txt` containing a set of valid uppercase English words[2], one per line. An extract of `words.txt` is presented below:

```
ABACK
ABAFT
ABAND
...
JUGS
JUMBLE
JUMBLES
...
ZONED
ZONELESS
ZONES
```

You have been provided with some skeleton code and various test cases. The source code is structured as follows:

- **include/trie.h** - Defines the struct for a Trie node and the API of the data structure. **You should not modify this file**.

[2]Painstakingly adapted from the original glossary published by Dodgson.

- **include/doublets.h** - Defines the function signatures for the implementation of the algorithm for Doublets. **You should not modify this file**.
- **include/tests.h** - Test logic and imports. **You should not modify this file**.
- **trie.c** - Write your answer to Part I here.
- **doublets.c** - Write your answer to Part II here.
- **trie_tests.c, doublets_tests.c** - Unit test suites for `trie.c` and `doublets.c`.

You should run the tests frequently to check your implementation. If you get stuck on Part I, you can move on to Part II, using the precompiled alternative dictionary (`lib/dict_alt.o`).

To generate the `Makefile` for your code, from `doublets/` run the following:

```
$ cd cmake-build-debug
$ cmake ..
```

The generated `Makefile` will then expose the following targets, which can be run from the `cmake-build-debug` directory:

```
make tests     # tests trie.c and doublets.c (using your Trie as dictionary)
make tests_alt # tests doublets.c (using the presupplied dictionary)
make run       # runs main.c (using your Trie as dictionary)
make run_alt   # runs main.c (using the presupplied dictionary)
```

You are free to write your solution using any available text editor or, alternatively, the CLion IDE. Should you opt for using CLion, you might find the comments at the top of `CMakeLists.txt` useful.

Part I: The Trie

You are asked to complete the implementation of a Trie to store uppercase English words. More specifically, the core functionalities that you will implement are those of *inserting* a word into a Trie and *searching* for a given word in a Trie. You will also need to define a function for *loading* words into a Trie from a given file. Listing 1 shows the structure of a `TrieNode`, aliased to type `dictionary_t`. An iterative and a recursive solution exists to each of the following questions: you are free to adopt whichever you deem most suitable and elegant.

Listing 1: Definition of a Trie node (excerpt from `trie.h`).

```
12 typedef struct TrieNode {
13
14   /* The array of pointers to the children of this node */
15   struct TrieNode **children;
16
17   /* END-OF-WORD flag, true if node represents
18    * the end of a word; false by default */
19   bool end_of_word;
20
21 } dictionary_t;
```

Complete the following functions

1. `dictionary_t *create_node(void) { ... }`

   This function creates and initialises *on the heap* a new node for a Trie with an array of `ALPHABET_SIZE` `NULL` pointers as children, where `ALPHABET_SIZE` is the size of the chosen

alphabet, e.g. `ALPHABET_SIZE` = 26 for the English language. A new node, by default, does **not** represent the end of any word.

[1 Mark]

2. `void free_node(dictionary_t *root) { ... }`

   This function frees *all* the resources associated with the Trie starting at `root`.

   [2 Marks]

3. `bool insert(dictionary_t *root, const char *word) { ... }`

   This function inserts the given `word` into the Trie starting at `root`. The insertion proceeds as follows: for each character `c` in `word`, move down the Trie by one level (i.e. to the next node) according to `c`. If, from the current node, there is no next node associated with `c`, create a new node and link it to the current node by updating the node's array at index `c`. When the node associated with the last character of the word is reached, it is marked as `end_of_word`. The function should return `false` if the `word` contained any non-alphabetic character, `true` otherwise.

   [4 Marks]

4. `bool find(dictionary_t *root, const char *word) { ... }`

   This function attempts to find the given `word` in the Trie starting at the `root`. It should return `true` if, after a successful traversal of the Trie using the characters in the word, the `end_of_word` flag of the last reached node is set.
   Hint: the overall structure of this function will be very similar to the structure of ***`insert()`****.*

   [3 Marks]

5. `bool load_from_file(dictionary_t *root, const char *filename) { ... }`

   This function reads words from `filename` and inserts each word in the Trie starting at `root`. You can assume a maximum word size of `MAX_WORD_SIZE` characters, where `MAX_WORD_SIZE` is 20. The function should return `false` in case of any file-related error, `true` otherwise. For the purpose of this exercise, you may also assume that each word in the given file is on a new line.
   Hint: the last sentence above also entails that each string you extract from a line end with a new-line character. Be sure you ***don't*** *insert that in the Trie.*

   [3 Marks]

Part II: Doublets

You will now implement the algorithm at the heart of the game of Doublets. For all the functions below, you may assume that the given words are provided in uppercase format. Chains are represented as arrays of `NULL`-terminated strings i.e. their type is `char **`. You are free to use as many helper functions as you see fit.

Complete the following functions

1. `bool valid_step(dictionary_t *dict, const char *curr_word, const char *next_word) { ... }`

   This function should return `true` if the step from `curr_word` to `next_word` is valid according to Rules 2 and 4 in Figure 1. For example, given the provided glossary as `dict`, the call to `valid_step(dict, ''WHEAT'', ''CHEAT'')` should yield `true`, while `valid_step(dict, ''WHEAT'', ''WHEAD'')` should yield `false`, because "WHEAD" is not an element of `dict`.

   [4 Marks]

2. `void print_chain(const char **chain) { ... }`

   This function prints a full Chain to standard output according to the convention presented in Figure 2. The parameter `chain` is a `NULL`-terminated array of strings. For example, given the `wheat_chain` from Listing 2, the call to `print_chain(wheat_chain)` should produce the following output:

   ```
   WHEAT
   cheat
   cheap
   CHEEP
   ```

   Hint: you might want to use `toupper(char)` *and* `tolower(char)`.

   [4 Marks]

3. `bool valid_chain(dictionary_t *dict, const char **chain) { ... }`

   Given a dictionary of valid words and a Chain, this function should return `true` if the Chain is valid according to all the rules in Figure 1. For example, given the provided glossary as `dict` and the `wheat_chain` and `repeat_chain` from Listing 2, the call to `valid_chain(dict, wheat_chain)` should yield `true`, while calling `valid_chain(dict, repeat_chain)` should yield `false`.

   [2 Marks]

4. `bool find_chain(dictionary_t *dict, const char *start_word, const char *target_word, const char **chain, int max_words) { ... }`

   The function attempts to find a valid Chain of at most `max_words` beginning with `start_word` and ending with `target_word`.

   If a valid Chain c can be found, the output parameter `chain` should contain c in the form of an array of *heap-allocated*, `NULL`-terminated strings, and the function should return `true`. If a valid Chain *cannot* be found, the function should instead return `false`, leaving `chain` untouched.

   Hint 1: Note that your solution should not necessarily produce the ***shortest*** *Chain. Any valid Chain composed by at most* `max_words` *words is an acceptable result.*

   Hint 2: One valid way to solve this problem is via a brute force, recursive search that, starting from the current word, tries out every possible alternative letter in turn. If you choose to implement a different strategy for solving the problem, please add some comments explaining your approach.

   [6 Marks]

Listing 2: Example chains.

```
const char *wheat_chain[] = { "WHEAT", "CHEAT", "CHEAP", "CHEEP", NULL };
const char *repeat_chain[] = { "WHEAT", "CHEAT", "CHEAP", "CHEAT", NULL };
```

Part III: Bonus Doublet

Invent a Doublet puzzle of your own. You can test it (and ultimately leave proof of it) in `main.c`. Use your imagination to come up with a clever connection between the start and target words to form a Chain of **maximum 7 words**. Finally, copy the printed chain as a comment just below your code. You can run the code in `main.c` with the commands `make run` or `make run_alt` (in case you wish to use the precompiled alternative dictionary).

[1 Mark]

You can use this paper for planning

You can use this paper for planning