C Practice Test

Imperial College London

Department of Computing

Doublets

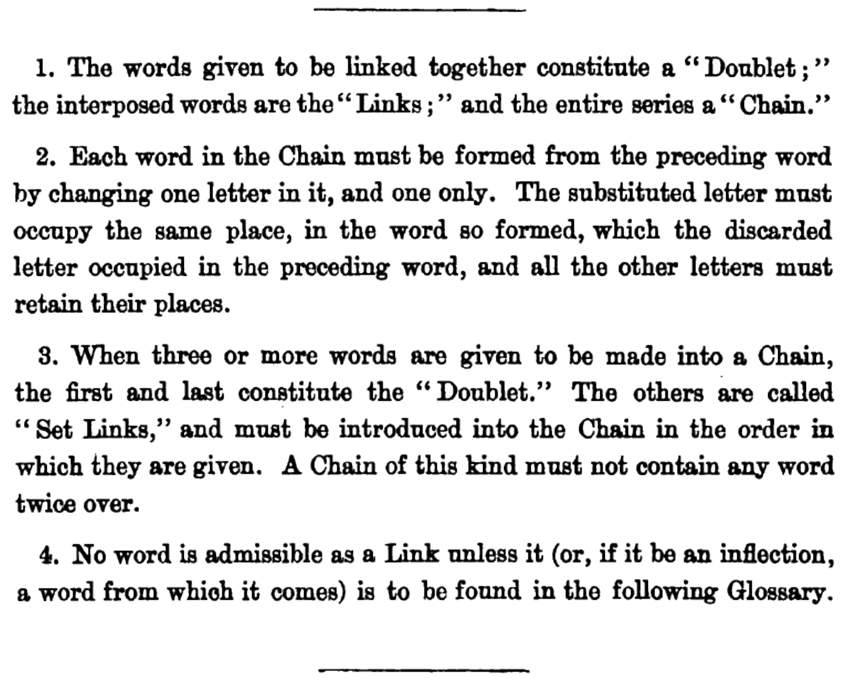
~~THREE HOURS~~
~~(including 10 minutes planning time)~~

- The maximum total is **100 marks**: 50 for each of the two parts.
- **Important:** TEN MARKS will be deducted from solutions that do not compile. Comment out any code that does not compile before you submit.
- Your code needs to compile and work correctly in the test environment which will be the same as the lab machines.
- You can start with any of Part A or Part B.
- The files can be found directly under the `doublets` directory inside your gitlab repository.
- Push the final version of your code to `gitlab` and then go to **LabTS** to select it and submit it to `CATe`.
- **Important:** You should only modify files `trie.c` and `doublets.c`. All other files are read-only, and are going to be overwritten by our autotester.

Overview

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.

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 **??** 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.

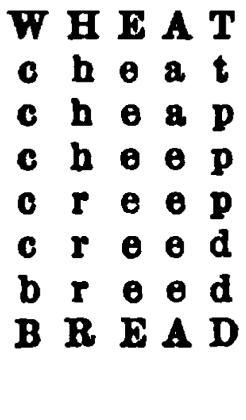
4. No word is admissible as a Link unless it (or, if it be an inflection, a word from which it comes) is to be found in the following Glossary.

Figure 1: Extracts of the rules of Doublets, 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 on the right.

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.

Remember to check the section on Building and Testing for details on how to test your code.



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

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.

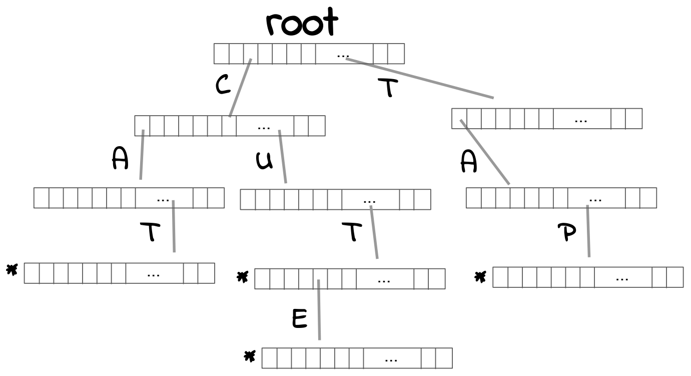


Figure 2: Example of a Trie containing the strings "CAT", "CUT", "CUTE" and "TAP". A star next to a node represents the end of a string at that node.

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 **??**.

Your Task is to 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, one per line. An extract of `words.txt` is presented below:

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

Part A: Data Structures – 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.

For this part the only files that should concern you are:

1. **trie.h** - Defines the struct for a Trie node and the API of the data structure.
2. **trie.c** - Write your answer to Part I here.

Testing frequently, complete the following functions in `trie.c`:

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.

   [5 Marks]
2. `void free_node(dictionary_t *root) { ... }`

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

   [7 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.

   [15 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()`.

   [13 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.*

   [10 Marks]

Part B: Algorithms – 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.

The only files that should concern you for this part are:

1. **doublets.h** - Defines function signatures.
2. **doublets.c** - Write your answer to Part II here.

Testing frequently, complete the following functions in `doublets.c`:

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 **??**. 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`.

   [10 Marks]

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

   This function prints a full Chain to standard output according to the convention presented in the Overview section. The parameter `chain` is a `NULL`-terminated array of strings.

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

   [10 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 **??**.

   [10 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.

   [20 Marks]

Building and Testing

The folder `doublets` contains both a `Makefile` (for use with the command line and a text editor of your choice) and a `CMakeLists.txt` (for use with CLion, if you wish to use an IDE).

Targets: Both the `Makefile` and the `CMakeLists.txt` specify 4 targets:

1. `dictionary` runs the `main` function provided to you in `trie.c`. You are supposed to run this target using `valgrind` to check for memory leaks in your implementation of Part A.
2. `doublets` runs the `main` function provided in `doublets.c`. Run this target using valgrind to check for memory leaks in Part B.
3. `testA` runs all the tests for Part A.
4. `testB` runs all the tests for Part B.

If using `make`, enter the `doublets` folder and build a specific target (e.g. `make dictionary`, `make testA`) or `make all` to build all the aforementioned targets.

You can now run each target from the `sonnets` folder. You should at least run the following commands, as this is what **the auto-tester will run**:

1. `./testA`
2. `valgrind --leak-check=full --show-leak-kinds=all ./dictionary`
3. `./testB`
4. `valgrind --leak-check=full --show-leak-kinds=all ./doublets`

If using CLion, import your project using: `File` → `New CMake Project from Sources` and open the `sonnets` folder as: `Open Existing Project`.

You can now build and run each target by selecting its configuration from: `Run` → `Edit Configurations` and then running it with `Run` → `Run`. We suggest running valgrind from the command line because when ran from within CLion it seems to detect some memory leaks which should not concern you.

Important note to Windows users: As your code needs to compile in the test environment, we suggest you ssh on a lab machine and test your code there as well. A common issue is detecting the end of line using only '\r' as delimiter. You should use '\n' as well.

Note on Valgrind: On some operating systems valgrind is buggy and detects memory leaks which are actually expected such as printed strings. **Therefore, you should only concern yourself with memory leaks reported as:** `definitely lost` or `indirectly lost`

Important: In case you accidentally overwrite any file, you can revert it to a previous version using: `git checkout <git-commit-hash> -- <file-name>`

Good luck!