C Final Test

Imperial College London

Department of Computing

Sonnets

Monday 8 June 2020
09:00 to 12:00
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 either Part A or Part B.
- The files can be found directly under the `sonnets` directory inside your gitlab repository.
- Push the final version of your code to `gitlab` before the deadline, and then go to `LabTS`, find the final/correct commit and submit it to `CATe`.
- **Important:** You should only modify files `trie.c` and `sonnets.c`. All other files are read-only, and are going to be overwritten by our autotester.

Overview

A sonnet is a short poem of (usually) fourteen lines that rhyme according to some pattern called a *rhyme scheme*. This pattern is described by a sequence of alphabetic letters where all the lines that rhyme with each other are designated the same letter. For instance, the following *Shakespearean* sonnet has rhyme scheme: *ABABCDCDEFEFGG*.

Shall I compare thee to a Summer's **day**?	*A*
Thou art more lovely and more **temperate**:	*B*
Rough winds do shake the darling buds of **May**,	*A*
And Summer's lease hath all too short a **date**:	*B*
Sometime too hot the eye of heaven **shines**,	*C*
And oft' is his gold complexion **dimmed**;	*D*
And every fair from fair sometime **declines**,	*C*
By chance or nature's changing course **untrimmed**;	*D*
But thy eternal Summer shall not **fade**	*E*
Nor lose possession of that fair thou **owest**;	*F*
Nor shall Death brag thou wanderest in his **shade**,	*E*
When in eternal lines to time thou **growest**:	*F*
So long as men can breathe, or eyes can **see**,	*G*
So long lives this, and this gives life to **thee**.	*G*

You are challenged to identify the preferred rhyme schemes of sonnet writers shown below.

Figure 1: Famous sonnet writers: Shakespeare, Petrarch and Spenser (from left to right).

Two lines rhyme with each other if they have the same phonetic ending. To this end, you are given a phonetic dictionary, **dictionary.txt**[1], that maps words to their phonetic spelling using sound units called *phonemes*. Here are some examples:

```
DAY D EY
MAY M EY
CONVICT K AA N V IH K T
PICKED P IH K T
```

Notice how the words `CONVICT` and `PICKED` rhyme because their phonetic spellings end in the same phonemes: `IH K T`.

In Part A, you will implement a data structure that can be used to load this dictionary in memory and perform fast lookups on it. In Part B you will develop an algorithm to identify rhymes and compute rhyme schemes. You can complete Part B without Part A. Remember to check the section on Building and Testing.

[1]A simplified version of the CMU Pronouncing Dictionary (Credit: Carnegie Mellon University).

Part A: Data Structures – Sparse Trie

You are asked to complete the implementation of a sparse trie that maps uppercase strings to integer values. Tries are tree-like data structures optimised for search time, and are commonly used to store *dictionaries* of strings. The key associated to a node is not stored explicitly, but is implied by the path from the root to the node. Usually, a trie node stores a child pointer for every letter in the alphabet, but this is wasteful as most of the pointers are NULL.

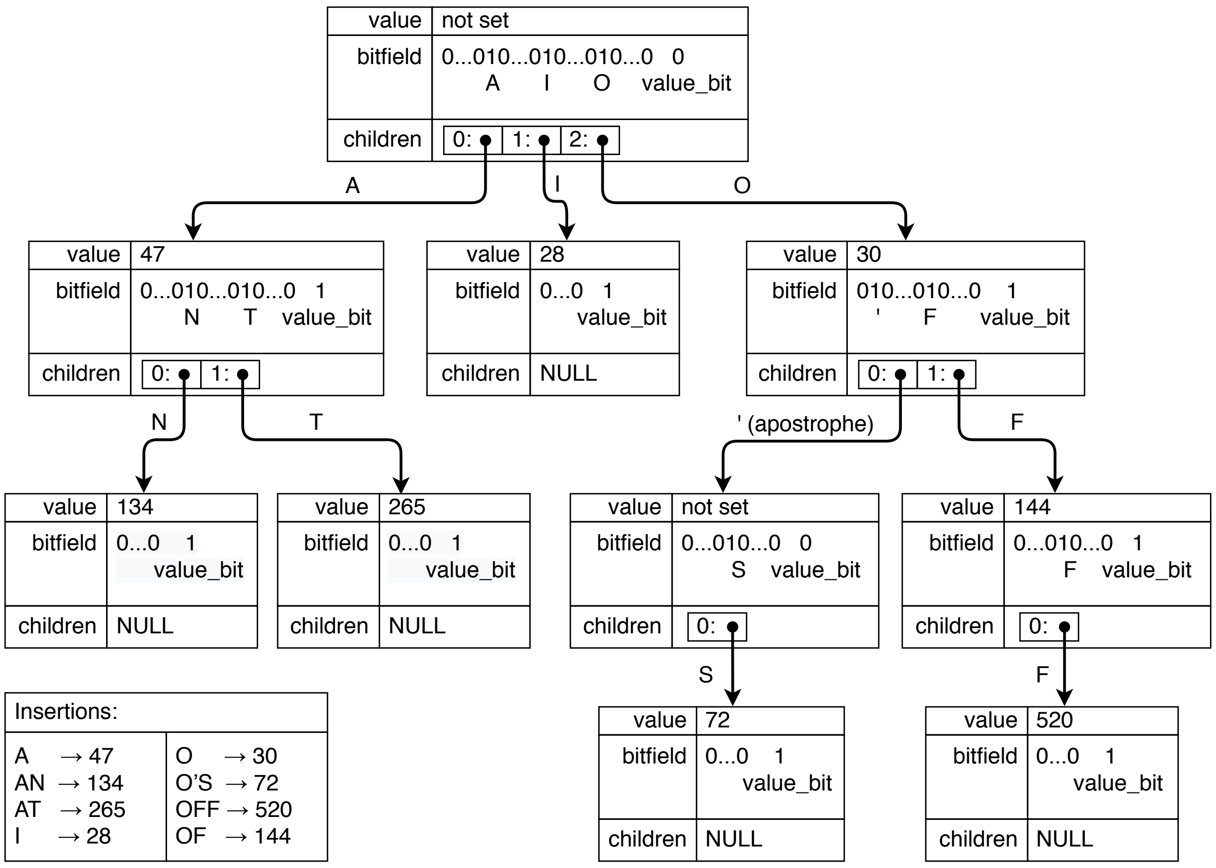
value | not set
bitfield | 0...010...010...010...0 0 | A I O value_bit
children | 0: 1: 2:
A
I
O
value | 47
bitfield | 0...010...010...0 1 | N T value_bit
children | 0: 1:
value | 28
bitfield | 0...0 1 | value_bit
children | NULL
value | 30
bitfield | 010...010...0 1 | ' F value_bit
children | 0: 1:
N
T
' (apostrophe)
F
value | 134
bitfield | 0...0 1 | value_bit
children | NULL
value | 265
bitfield | 0...0 1 | value_bit
children | NULL
value | not set
bitfield | 0...010...0 0 | S value_bit
children | 0:
value | 144
bitfield | 0...010...0 1 | F value_bit
children | 0:
S
F
value | 72
bitfield | 0...0 1 | value_bit
children | NULL
value | 520
bitfield | 0...0 1 | value_bit
children | NULL
Insertions:
A → 47
AN → 134
AT → 265
I → 28
O → 30
O'S → 72
OFF → 520
OF → 144

Figure 2: The figure shows a sparse trie that holds the string to int mappings from the *Insertions* table (lower left). Child pointers are labelled with the symbol they represent.

To solve this issue, a **sparse trie** (as shown in Figure 2) only holds non-NULL pointers; i.e. for the "children" that actually exist, sorted according to the symbols they represent. Each node uses a bitfield of 32 bits (encoded as a `uint32_t`) to mark what children exist and whether the node's value was set or not. Bit 0 is the `value_bit` and is set (i.e. has value 1) only if the node is associated with an inserted key–value pair. Bits 1 through 31 are associated with symbols from the supported alphabet; for each set bit, there is a child pointer in the children array. Figure 3 summarises the meanings of each bit position in the bitfield.

bit_position:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
meaning:	␣	‘	-	.	_	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	value_bit

Figure 3: Meaning of each bit from the bitfield. Note that ␣ is the *space* character.

Note how the children array of the root node from Figure 2 contains exactly 3 pointers because there are 3 bits set between bits 1 and 31 (i.e. for symbols: `A, I, O`). Furthermore, note how the children arrays are always sorted with respect to the symbols represented by the elements.

Part A Tasks

For Part A the following 2 files are of interest:

1. `trie.h` contains the definition for the `struct` that represents a trie node.
2. `trie.c` contains a function `int get_bit_pos(char symbol)` that takes as input a character and returns the position of its corresponding bit in the bitfield (as shown in Figure 3), or -1 if the character is not in the supported alphabet.

Important: Be aware that when working with 32 bit numbers (like `uint32_t`) shifts of more than 31 places have undefined behaviour.

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

1. `trie_t *trie_new();`

   This function returns a pointer to a new *heap-allocated* trie node with no children.

   [7 Marks]

2. `int count_set_bits(uint32_t n);`

   This function returns the number of set bits in the binary representation of `n`.

   [8 Marks]

3. `void trie_free(trie_t *root);`

   This function recursively frees *all* the resources associated with the trie starting at `root`. The function should be able to handle a NULL input argument.

   Hint: You can use `count_set_bits` to determine the number of children.

   [8 Marks]

4. `bool trie_get(const trie_t *root, const char *key, int *value_out);`

   This function attempts to retrieve the value corresponding to the given `key` in the trie starting at `root`. If a value exists for `key`, it is written into the memory pointed to by `value_out` (the output parameter) and the function returns `true`. Otherwise, the function returns `false`. For a node at depth *d*, you should follow the child pointer that corresponds to the character on position *d* in `key`.

   Hint: To check if a child for character `c` exists, use `get_bit_pos(c)`, build a bitmask, and query the node's bitfield. To find its position in the children array rely on the fact that the array is sorted, a property maintained by insert.

   [12 Marks]

5. `bool trie_insert(trie_t *root, const char *key, int value);`

   This function inserts the given key–value pair into the trie starting at `root`. The function returns `false` only if `key` contains invalid characters (i.e. not in the supported alphabet). The trie traversal is similar to that implemented for `trie_get`. Additionally, whenever a child node for a character `c` does not exist, you need to create it and add it to the (resized) children array at the correct position. That is, children array entries must remain in lexicographical order with respect to the characters they represent. When the child node associated with the last character of `key` is reached, `value` is saved (possibly overwriting a previous value for the same key) and the `value_bit` is set to mark the insertion.

   Hint: A full marks solution should resize the children array using `realloc` and reorganise its elements with `memmove` when needed.

   [15 Marks]

Part B: Algorithms – Rhyme Schemes

You are asked to compute the rhyme schemes of sonnets from three files and to find the most common rhyme scheme in each file. To compute a sonnet's rhyme scheme you need to identify the lines whose last words have the same rhyme. The rhyme of a word is the part of the phonetic spelling that starts with the last phoneme vowel, i.e. a phoneme that starts with a vowel: A, E, I, O, U. In case there are no phoneme vowels (e.g. THS), then the rhyme is the whole phonetic spelling. Below are examples of words, their phonetic spellings and rhymes.

word	A'S	SCIENCE	TO	THS	RATHER
phonetic spelling	EY Z	S AY AH N S	T UW	TH S	R AE DH ER
rhyme	EY Z	AH N S	UW	TH S	ER

Table 1: Each column shows an example of a word with its phonetic spelling and rhyme.

Each line of **dictionary.txt** contains a word, followed by a space, followed by a sequence of space-separated phonemes that represent the word's phonetic spelling. To avoid repeatedly searching for a word's phonetic spelling in **dictionary.txt** and then finding its rhyme, you should pre-compute an in-memory dictionary of words (`strings`) to unique rhyme ids (`ints`). Words that have the same rhyme must be mapped to the same rhyme id.

To summarise your tasks: you have to parse **dictionary.txt** for words and their rhymes, allocate a unique integer id to each rhyme, populate an in-memory dictionary with mappings from words to rhyme ids, parse a file into sonnets and lines, extract the last word from each line, lookup its rhyme id in the in-memory dictionary, generate a sonnet's rhyme scheme, and finally, find the most common rhyme scheme from sonnets in a file.

Part B Tasks

For Part B the following files, functions and macros are of interest:

1. `shakespeare.txt, spenser.txt, petrarch.txt` each contain several sonnets written by Shakespeare, Spenser, and Petrarch, respectively.
2. `dictionary.txt` contains a dictionary of phonetic translations (see Overview).
3. `maps.h` and `maps.c` provide two map implementations: `strintmap_t` maps strings to ints and has the same interface as the trie from Part A, and `intcharmap_t` maps ints to chars.
4. `sonnets.h` contains macros `dict_t`, `dict_new`, `dict_free`, `dict_get`, `dict_insert` that alias either your `trie_t` implementation or the pre-supplied `strintmap_t`, depending on a compiler flag. You **must** use these macros to represent the in-memory dictionary from words to rhyme ids. See the section on **Building and Testing** for more details about building and compiling using your trie or the pre-supplied map.
5. `sonnets.c` contains a completed `main` function and some macros that you should use in your implementation.
6. `char *lastwordtok(char *line)` from `sonnets.c` returns a pointer to the last word in `line` or `NULL` if `line` does not contain any words. This function modifies `line` by adding a sentinel character '\0' immediately after the last word. A word is a sequence of non-*space* characters that starts and ends with alphabetic letters.

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

1. `char *uppercase(char *str);`

   This function modifies `str` in-place by setting each character to uppercase. For ease of use, it also returns a pointer to the beginning of `str`.

   [4 Marks]

2. `const char *strrhyme(const char *phonemes);`

   This function takes as input a string that represents a sequence of phonemes and returns a pointer to the last phoneme vowel, if one exists, or to the first phoneme otherwise. E.g. `strrhyme("S AY AH N S")` returns a pointer to `"AH N S"` (also see Table 1).

   [6 Marks]

3. `dict_t *load_rhyme_mappings_from_file(const char *phonetic_dict_filename);`

   This function returns a pointer to an initialised `dict_t` that contains mappings from every word in `phonetic_dict_filename` to an (integer) id that represents its rhyme.

   Assume that the format of the file is identical to that of **dictionary.txt** and the maximum length of a line from the file is macro `MAX_DICT_LINE_LENGTH`. All the characters in the file are from the supported alphabet from Part A.

   Hint: Use `strrhyme` to extract the rhyme from a phonetic spelling and assign it a unique id using a `dict_t` that maps rhymes to their assigned rhyme ids.

   [15 Marks]

4.
   ```
   bool next_rhyme_scheme(FILE *sonnets_file,
                          const dict_t *rhyme_mappings, char *out);
   ```

   This function sets the `out` parameter to be the rhyme scheme of the next sonnet in the given `sonnets_file`. Sonnets are separated by lines (could be multiple) without any words on them. The return value indicates whether a sonnet has been parsed. When there are no more sonnets in `sonnets_file`, the function returns `false`. If a word does not have a rhyme id stored in `rhyme_mappings`, an error message including the word should be printed.

   Assume that macro `MAX_SONNET_LINE_LENGTH` is the maximum length of a line from `sonnets_file` and that the input parameter `rhyme_mappings` contains a dictionary from words to unique rhyme ids as returned from `load_rhyme_mappings_from_file`.

   Hint: Parse the last word on a line using `lastwordtok`. Use an `intcharmap_t` to map rhyme ids to the character they represent in the rhyme scheme.

   [15 Marks]

5.
   ```
   void most_common_rhyme_scheme(FILE *sonnets_file,
                                 const dict_t *rhyme_mappings, char *out);
   ```

   This function sets the `out` parameter to be the rhyme scheme that appears most often in the given `sonnets_file`, or “N/A” if there are no sonnets in the file.

   Assume that macro `MAX_NUM_SONNET_LINES` is the maximum number of lines in a sonnet and that the input parameter `rhyme_mappings` contains a dictionary from words to unique rhyme ids as returned from `load_rhyme_mappings_from_file`.

   Hint: Use `next_rhyme_scheme` to get the rhyme schemes of sonnets in the file and track their frequency counts using (yet another) `dict_t`.

   [10 Marks]

Building and Testing

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

Targets: Both the `Makefile` and the `CMakeLists.txt` specify 5 targets:

1. `trie` 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. `sonnets_trie` runs the `main` function provided in `sonnets.c` using the trie implemented in Part A. If the trie was implemented correctly, you can run this target using `valgrind` to check for memory leaks in your implementation of Part B. The runtime should be under a minute.
3. `sonnets_map` runs the `main` function provided in `sonnets.c` using the pre-supplied map implementation. Run this target using valgrind to check for memory leaks in Part B, without being affected by possible memory leaks or errors in your Part A. Note that running valgrind with this target might take a couple minutes on slower systems.
4. `testA` runs all the tests for Part A.
5. `testB` runs all the tests for Part B using the pre-supplied map implementation to avoid any errors from Part A propagating to Part B.

If using `make`, enter the `sonnets` folder and build a specific target (e.g. `make trie`, `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 ./trie`
3. `./testB`
4. `valgrind --leak-check=full --show-leak-kinds=all ./sonnets_map`

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!