Introduction to Software Systems (308-206B)
Assignment 3: a word tree

Winter Term 2001

Practical information

- Due date (official): Tuesday 10 April. Submissions after the due date will be accepted without penalty until Monday 23 April 23:59 (VisualCM will stay open).
- Submission medium: VisualCM. Submit all your relevant files. In particular: `Makefile` and `wordcheck.c`, not your input/output.
  Beware: your submission is for 206B, *not* for 206A !
- You may work in groups of upto 3 people. Each file handed in must contain names and IDs of all contributers. Each contributer must submit all files individually. Every partner should understand the full solution. The final exam *may* contain some assignment-related questions.
- You may develop your code on any platform. Your code will however be evaluated on a SOCS UNIX machine. Make certain it runs correctly on a SOCS UNIX machine. In your Makefile, use

  ```
  CC=gcc
  CFLAGS=-g -Wall -ansi -pedantic
  ```

- The solution to this assignment is quite short (thanks to recursion) and should not take you very long. If however you get stuck with recursion, come to the TA's or my office hours for help !

Objectives

- Thorough understanding of pointers and dynamic memory allocation/de-allocation (`malloc()`/`free()`).
- Reading from file and `stdin`, writing to file and `stdin`.
- Getting experience with processing command-line options.
- The use of recursion to build and traverse complex, dynamically built data structures.
- An exercise in *design*. A good design will be elegant and include all special cases.
- An exercise in *coding style*. Make the code re-usable by including requirements and design information in the comments.

Assignment

A single program `wordcheck` needs to be built. Obviously, the source code may be spread over multiple files. In this stripped-down version of the assignment, wordcheck does not do any word checking at all. It only reads a word file, builds an internal data structure (a tree, described below), and then outputs the information in that data structure.

Command-line options

The program must process command line options (by preference implemented using `getopt()` as that is most elegant and least effort).:

- `-h` :*help*. When this *optional* option is set, the program prints the usage message.
- `-v` :*verbose*. When this *optional* option is set, the program prints its progress to `stderr`. Use your own judgement in deciding how verbose the program should be.
- `-V` :*version*. When this *optional* option is set, the version of the program is printed to `stderr`.
- `-w wordfile`. A mandatory option (not giving it on the command line is an error). `wordfile` is the file containing words (one per line) to be read in and stored in dynamic memory.
- `-m mode` :*mode*. The *optional* mode is either `print` or `check`. The latter should lead to a `not yet implemented` message on `stderr`. If the mode option is not given, the default is `print`. What this mode does is described below.
- `-o outputfile` :*output*. This *optional* option gives the name of an ASCII file to "print" output to. If the file already exists, output is *appended*. If the option is not given, output is sent to `stdout`.

Some requirements:

- Default values for options must be cleanly grouped (easy to modify). Example: by default, verbose is false.
- Wrong use of options must be handled by printing the usage message and `exit(1)`.

Processing the word file

- Blank space in the input should be ignored.
- All letters are converted to lower case (using the system routine `to_lower`.
- All letters (after conversion to lower case) not in the range `a - z` should be ignored and a *warning* should be output to `stderr`.
- Words are stored in a *tree*. The particular tree used stores common starting parts of different words only *once*. A Directed Acyclic Word Graph (DAWG) is a more efficient extension of this used in computer Scrabble games. In this case, also common endings are stored only once.
- The nodes in the tree are structures of type `NodeType`

```
/* 26 elements
 * each corresponds to a lower-case letter
 * Note how the C compiler is able to handle
 * the use of struct element which is only defined later
 */
typedef struct element NodeType[26];

/* the elements */
typedef struct element
{
  unsigned char is_word_end; /* this letter ends a word       */
  NodeType *word_endings;    /* words exist with more letters */
} ElementType;
```

  The *n*-th row of a `NodeType` structure (a node in the tree), corresponds to the *n*-th lower-case letter in the alphabet (taking into account C's numbering from 0, the *n*-th row will have index $n-1$). The properly initialized `NodeType` structure is shown in Figure 1. `is_word_end` denotes whether there exists a word which ends here. `word_endings` points to another `NodeType` structure in case words exist with more letters. In case this is the last letter of a word, `word_endings` will be `NULL`.
- Figure 2 shows the dynamically created tree structure after the words

```
lemon
lemons
```

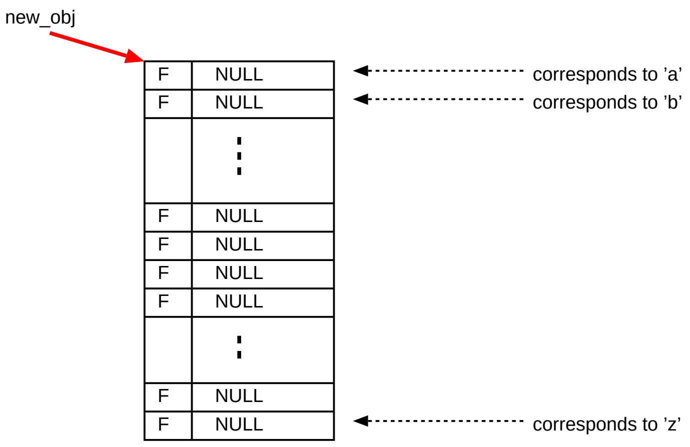

Figure 1: An initialized new node

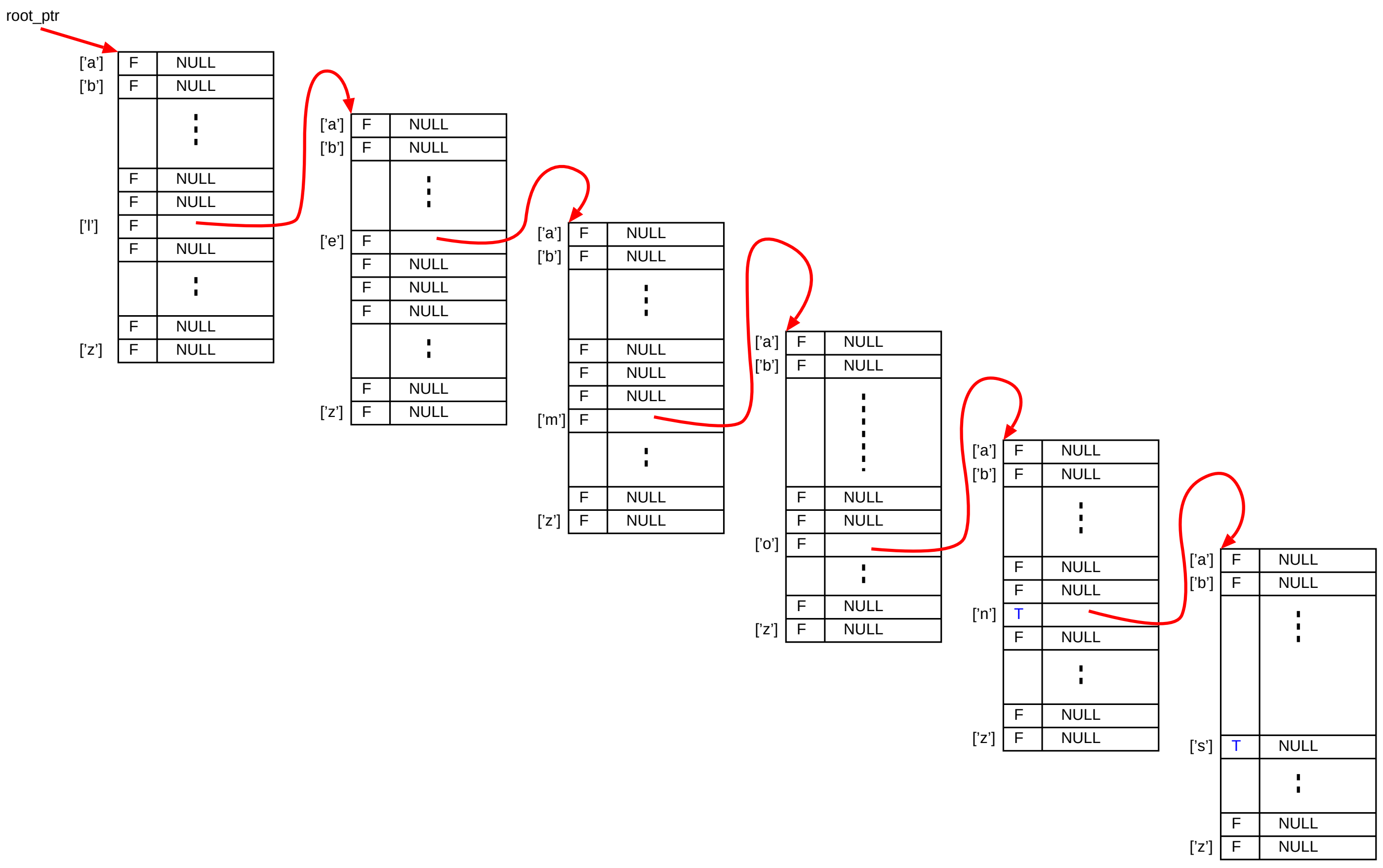

Figure 2: Storing 'lemon' and 'lemons'

have been processed. In the first node, all `is_word_end` fields are False (F) as there are no words with only one letter. The only words processed upto now start with the letter ‘l’, so the `word_endings` field points to a structure with information about the second letter of words starting with ‘l’. In the one but last node, the `is_word_end` field for ‘n’ is True (‘n’ is the end of the word “lemon”). As there is a word starting with “lemon” (“lemons”), the `word_endings` field is not `NULL`, but points to yet another `NodeType` structure.

- Creating the tree is *very* simple when using the right *recursive* function `build()`. Build should take a `NodeType` pointer (possibly `NULL`) and a string (with the remainder of a word) as arguments ... `build()` returns a pointer to a node (new in case the pointer argument was `NULL`, the pointer argument in case it wasn’t). With `root_ptr` (pointer to `NodeType`) initially set to `NULL`, adding words would consist of calls `build(root_ptr, word);` with `word`, the word to be entered.
- Figure 3 shows the tree after processing the words

```
add
ado
adds
added
adder
adders
ape
```

- Once the complete tree has been built, it needs to be printed. This is again done recursively. Given a pointer to a node as well as a string containing the part of the word which was already in the path from the root of the tree till this point, `print()` will print all words reachable from this point.

Resources

- The template to use for your C code: template.c
- The template/example to use for your Makefile: Makefile
- Be consistent in your indentation. Using the GNU tool “indent” can help. Below are some reasonable command-line options (which you may wish to put in your .cshrc file if you use csh).

```
#        indent (GNU C beautifier)
#
#         -ts1: tab stop width = 1
#         -bli0: block indent ({}) 0
#         -c28: try to start code-line comments in column 28
#         -cd28: try to start declaration-line comments in column 28
#         -npcs: no space after function call names
#         -l72: maximum non-comment-line length
#         -lc72: maximum comment-line length
#
alias indent  'indent -ts1 -bli0 -c28 -cd28 -npcs -l72 -lc72'
```

Beware of TABs. TAB expansion (as a number of blank spaces) on one machine may be different from TAB expansion on another. This may result in code which looks properly indented on one platform and chaotic on another. To avoid this:

  - do never insert TABs willingly for indentation purposes,
  - make sure your editor does not substitute a number (8, for example) of consecutive blanks by a TAB. In vi for example, you may wish to `set tabstop=100` to avoid substitution.

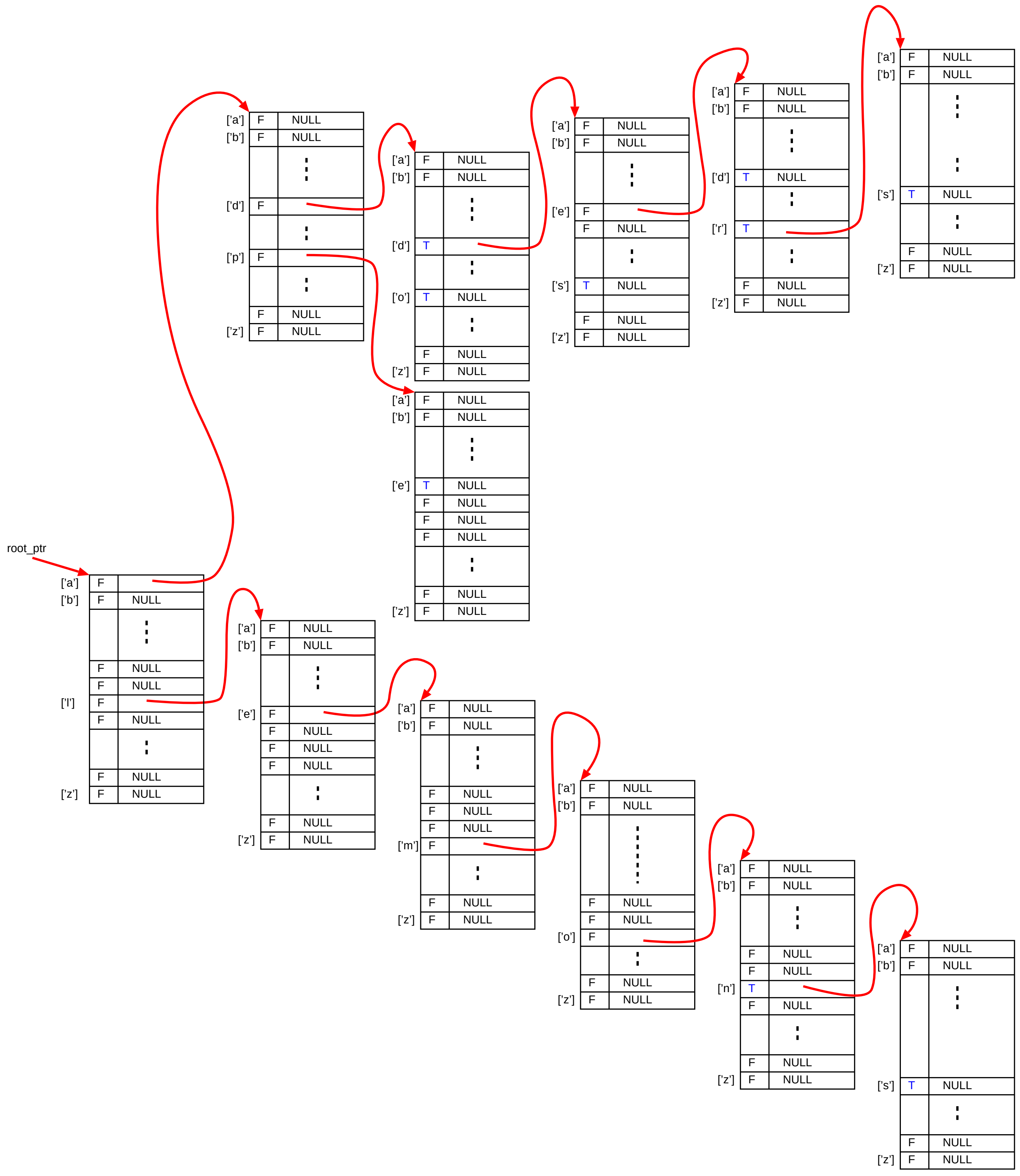

Figure 3: More words are added