CS2850 Lab Assignment 2 (instructions)

Nicolo Colombo

December 1, 2020

Preliminary note The lab part of Assignment 2, A2 Lab Quiz , is a Moodle quiz where all questions require you to upload and run parts of the C program described in this document. Completing the Moodle quiz is the only way to submit your work. The quiz has no time-limits but you should complete it by

Friday 4th December 2020, 10 am

and no attempts will be allowed after this deadline. If you have not submitted your answers when the Quiz closes, they will be submitted automatically and graded as they are. For each question in A2 Lab Quiz , you can modify and re-check your code up to 5 times before submitting your answer. For every extra non-submitted check you will get a 20% penalty on that question.
Before starting A2 Lab Quiz , be sure that

- your program does not produce any error or warning messages if you *compile* it using

  `gcc -Wall and -Werror your_code.c`

  because this is the compiler and the options used by the Moodle code checker[1]
- your program reproduces the expected *output* on the test files in test.zip [2]
- Valgrind does not show any error or *memory leaking* problems[3]

> **NOTE:** All the work you use to answer the questions in A2 Lab Quiz should be solely your own work. Coursework submissions are routinely checked for this.

1 A binary search tree for text analysis

For this assignment, we ask you to write a program, `wordTree.c`, that creates a *binary search tree*, i.e. a special kind of linked-list, to store and compute the frequency of the words in a given input text. Before you start coding, we suggest you have a quick look at the presentation of binary search trees as data structures in Section 12 of Chapter IV in *Introduction to Algorithms* by Cormen et al.. The program can be seen as a string-based version of the binary search tree for classifying integers described in Section 2 of Exercise lab-sheet 8 .[4]
More concretely, your program will

1. parse the *input* word-by-word
2. *check* if the incoming words have already been stored in the tree

[1] In general, we encourage you to use `clang` because it produces more user-friendly messages. The `gcc` compiler with the `-Wall -Werror` flags is stricter than `clang` as it considers all warnings as errors.

[2] The content of the files in test.zip and the expected output is also shown in Section 2.

[3] To see this, run your program on `linux.cim.rhul.ac.uk` with the command

`valgrind ./executable < input_file`

[4] It is a good idea to try to complete the exercise suggested in Section 2 of Exercise lab-sheet 8 before writing `wordTree.c`.

3. increase a word-specific *count variable* if they have already been stored in the tree

4. create a *new node* to store them if they are not already in the tree

The binary structure of the tree will be based on the *alphabetic order*, i.e.

- a new word will be assigned to the *left child* of a node if the new word comes *before* the word stored in that node

- otherwise, the new word will be assigned to the *right child*

For example, the word `quiz` comes after the word `quit` and the word `tree` comes before both. To avoid ambiguities, your program will also *ignore* all numerical or special characters, e.g. `a3.b*72` will be converted to `ab`, and *convert* all capital letters to lower-case letters, e.g. `AbCd` will be converted to `abcd`, before performing any word-comparison. In the end, `wordTree.c` will print all words contained in the input document (with all numerical and special characters removed and all letters converted to lower-case letters) The words will be printed on `stdout` in alphabetic order, each one followed by the corresponding frequency as shown in the examples of Section 2. To write your program, please follow the instructions given in the next sections. In particular, the return values and arguments of all your *auxiliary functions* must be as requested. The reason is that A2 Lab Quiz will check the output of your *entire* program (Question 5) but also the correct behaviour of the auxiliary functions `insertNode` (Question 1), `getWordFromFile` (Question 2), `printNode` (Question 3), and `freeNode` (Question 4). Finally, note that the code of all *string-handling* functions you need in `wordTree.c` is provided in Section 3.

1.1 Define a self-referential structure called `node`

Start by declaring the tree data structure as

```
struct node{
        char word[MAXCHARS];
        int length;
        int count;
        struct node *left;
        struct node *right;
};
```

where:
the `word` member will store the *word* associated with the node
the `length` member will store the *length* of `word` (in number of characters)
the `count` member will store the *number of times* `word` appears in the input document
the `left` and `right` members will be pointers to the left and right *children* of the node
Note that the declaration of `struct node` implicitly assumes that the maximum length of the words is `MAXCHARS`, which should be a fixed constant defined just after the `#include` lines by

```
#define MAXCHARS 100
```

1.2 Initialize the tree with a `NULL` pointer

In `main`, declare the following variables:

- `char buf[MAXCHARS]`: a character array of size `MAXCHARS`, to *buffer* the word that you are reading from the input

- `int end`: an integer variable initialized to 0, to be used in the *stopping* condition of the `while`-loop that goes through the input file word-by-word

- `struct node* root`: a pointer to a `struct node` object initialized to `NULL`, to be used as *root* of the tree

1.3 Define an *recursive* function to insert new nodes in the tree

The advantage of linked-list is that you do not need to know the number of nodes in advance. Similarly to the case of simply-linked lists[5] you can let the tree grow as new words arrive, by attaching new nodes at the right position. Given a new word, a new node can be inserted by calling a recursive function that searches the tree for the correct *terminal node* and attaches the new node to that node. Starting from the root, you can find the correct terminal node by *comparing*[6] the new word with the words stored in the tree nodes to choose the branch to follow, i.e. whether to go to the left or to the right.

More concretely, define a *recursive* function declared as

`void insertNode(struct node **t, char *s)`

where:

- the `t` argument is a pointer to an object of type `struct node*`, i.e. a *pointer to a pointer* to an object of type `struct node`
- the `s` parameter is a *string* containing the new word that will be inserted in the tree

Note that the `t` argument is a pointer to a pointer to a tree node, which allows you to call `insertNode` with the address of a pointer, e.g. the pointer to the root node `root`, and *change* its value. To write `insertNode`, implement the following pseudo-code:

```
insertNode(t, s):
    if *t is NULL:
        create a new node and let *t point to it
        copy s to **t.word
        set **t.count to 1
        set **t.length to the length of s
        set **t.left and **t.right to NULL
    else:
        compare s with **t.word according to the alphabetic order
        if **t.word comes after s:
            call insertNode with parameters **t.left and s
        if **t.word comes before s:
            call insertNode with parameters **t.right and s
        if **t.word and s are the same:
            increase **t.count by one
```

To clean, compare, and copy strings you can use the *string-handling* subroutines given in Section 3.

1.4 Try to insert the words "hello" and "world" in the tree

Once you have defined `insertNode`, you can insert a new word, e.g. `buf`, in the tree by calling it from `main`, with parameters `&root` and `buf`. E.g. check if your version of `insertNode` works by writing[7]

```
insertNode(&root, "hello");
insertNode(&root, "world");
printf("%s\n", root->word);
printf("%s\n", root->right->word);
```

Note Be sure that your implementation of `insertNode` accepts the same parameters as in the declaration given above and behaves as suggested. Question 1 in A2 Lab Quiz will ask you to copy your version of `insertNode` to a sandbox and test it automatically.

[5]See for example `simpleList.c` .

[6]As mentioned before, the comparison between words should be based on the alphabetic order, after removing or special characters and transform all upper-case letters into the corresponding lower-case letters.

[7]Remove these lines before attempting A2 Lab Quiz .

1.5 Parse the input word-by-word

To parse the input file string word-by-word and load the *buffer* with the current word, write a function

`int getWordFromFile(char *buf, int *end, int maxLength)`

where:

- the `buf` argument is an array of `char`, to temporarily store the *current* word
- the `end` argument is a pointer to `int`, to be used in `main` for *exiting* the reading-loop when you reach the end of the input file (`EOF`)
- the `maxLength` argument is the*maximum* number of characters that you can store in a node
- the return value is the number of `char` *copied* to `buf` (excluding the null-character at the end)

To write `getWordFromFile`, you can modify the function `getWord` you used in `simpleList.c` to stop reading from the input. Note that, in this case, you should stop reading when you reach a `EOF` instead of a new-line character. This will allow `wordTree.c` to process entire text files instead of a simple `stdin` line. Use

`int lowerCase(int *c)`

given in Section 3 to ensure that only lower-case letters get stored in `buf`. Note that `lowerCase` takes a pointer to a `int` and *not* a pointer to `char` as a parameter because you need treat the output of `getchar()` as an `int` instead of a `char`.[8]

Note Be sure that your implementation of `getWordFromFile` accepts the suggested parameters and has the same return value as in the declaration given above. Question 2 in A2 Lab Quiz will ask you to copy your version of it to a sandbox and test it automatically.

1.6 Define a `while`-loop to call the node-insertion function

In `main`, create an *infinite* `while`-loop to read from the input until `getWordFromFile` changes the value of `end` to 1. For example, you can have

```
while (end == 0){
    int j = getWordFromFile(buf, &end, MAXCHARS);
    if (j > 0) insertNode(&root, buf);
}
```

as `getWordFromFile` returns the number of `char` copied to `buf`, excluding the null-termination character.

1.7 Print all tree nodes *from the left to the right*

To print the words stored in the tree in alphabetic order, you need to print *all* tree nodes (terminal and non-terminal nodes) from the left to right.[9] To do this, define a recursive function

`void printNode(struct node *t)`

where:

- the `t` argument is a *pointer* to an object of type `struct node`
- the function *does not* return any value

To write `printNode`, implement the following pseudo-code:

```
printNode(t){
        if t is NULL:
            return
        if t->left is not NULL:
            call printNode with parameter t->left
        print t->word and the corresponding count in the suggested format
        if t->right is not NULL:
            call printNode with parameter t->right
}
```

[8] Why?

[9] Why? Draw a simple example to understand how words get placed in your binary search tree (or look at some figures in Section 12 of Chapter IV in *Introduction to Algorithms* by Cormen et al.)

where, in the print instruction, you should use the following call to `printf`

`printf(" %s(%d) ", t->word, t->count)`

When you are done, add a call `printNode` in `main`[10], recompile and run your program to see if it prints on `stdout` the list of words and their count as expected. For example, if you write

`hello hello world!`

into a file called `in.txt` and then run

`./a.out < in.txt`

you should obtain the following output[11]

```
 hello(2)  world(1)
```

Note Be sure that your version of `printNode` behaves as explained in this section. Question 3 in A2 Lab Quiz will ask you to copy it to a sandbox and test it automatically.

1.8 Free all tree nodes

To free all tree nodes, define a similar recursive function

`void freeNode(struct node *t)`

where:

- the `t` argument is the a *pointer* to a an object of type `struct node`
- the function *does not* return any value

To write `freeNode`, implement the following pseudo-code:

```
freeNode(t){
    if t is not NULL:
    call freeNode with parameter t->left
    call freeNode with parameter t->right
    free node t
}
```

Add a call of `freeNode` to `main` and run your code with Valgrind to test that all heap memory is correctly freed.

Note Be sure that your version of `freeNode` frees all nodes and does not produce any error when you compile your code with `gcc -Wall -Werror` or run your code with Valgrind. Question 4 in A2 Lab Quiz will ask you to upload your version of `freeNode` and test it automatically.

2 Expected output and examples

Compile your program with the `-Wall -Werror` flagged version of the `gcc` compiler[12], i.e. enter

`gcc -Wall -Werror wordTree.c`

and check if you get any error or warning messages. Run the executable with Valgrind by typing

`valgrind ./a.out`

Wait for Valgrind to start and press `ctrl-D` to send an `EOF` signal. Check the *memory leak* report and if the are any error. If everything looks fine, extract the testing examples `ex1.txt` and `ex2.txt` from test.zip and save them in the same directory as `wordTree.c` and `a.out`. Check that your program reproduces the content of the output files, `out1.txt` and `out2.txt`[13], when you run

`./a.out < ex#.txt`

on `linux.cim.rhul.ac.uk`. For illustrative purposes only, the content of the files in test.zip is reported in the next subsections.[14]

[10] What is the parameter you need to pass to `printNode` when you call it from `main`?

[11] Note the white space at the beginning of the printed line.

[12] The the Moodle code-running system uses this version.

[13] `out1.txt` and `out2.txt` are also in test.zip

[14] Note that cut and copy from a pdf file may change some characters.

Note Question 5 in A2 Lab Quiz will ask you to run your full program `wordTree.c`. As the check is based on its output on the terminal, be sure that your code prints the words list in the specific format shown in the example above. In particular, note that:
- there is a white-space at the *beginning*
- the is *no white-space* between a word and the corresponding count indicated between brackets

2.1 Test example 1

ex1.txt

```
one
two Two,
three Three THree\\
four Four, FOur, ... , FOUR
```

out1.txt

```
 four(4)  one(1)  three(3)  two(2)
```

2.2 Test example 2

ex2.txt

```
A structure is a collection of one or more variables, possibly of different types, grouped
together under a single name for convenient handling. (Structures are called \records" in some
languages, notably Pascal.) Structures help to organize complicated data, particularly in large
programs, because they permit a group of related variables to be treated as a unit instead
of as separate entities.

One traditional example of a structure is the payroll record: an employee is described by a
set of attributes such as name, address, social security number, salary, etc. Some of these
in turn could be structures: a name has several components, as does an address and even a salary.
Another example, more typical for C, comes from graphics: ...


from Chapter 6 of "C programming Language", by Brian W. Kernighan and Dennis M. Ritchie
```

out2.txt

```
 a(9)  address(2)  an(2)  and(2)  another(1)  are(1)  as(4)  attributes(1)  be(2)  because(1)  brian(1)
 by(2)  c(2)  called(1)  chapter(1)  collection(1)  comes(1)  complicated(1)  components(1)
 convenient(1)  could(1)  data(1)  dennis(1)  described(1)  different(1)  does(1)  employee(1)
 entities(1)  etc(1)  even(1)  example(2)  for(2)  from(2)  graphics(1)  group(1)  grouped(1)
 handling(1)  has(1)  help(1)  in(3)  instead(1)  is(3)  kernighan(1)  language(1)  languages(1)
 large(1)  m(1)  more(2)  name(3)  notably(1)  number(1)  of(8)  one(2)  or(1)  organize(1)
 particularly(1)  pascal(1)  payroll(1)  permit(1)  possibly(1)  programming(1)  programs(1)  record(1)
 records(1)  related(1)  ritchie(1)  salary(2)  security(1)  separate(1)  set(1)  several(1)  single(1)
 social(1)  some(2)  structure(2)  structures(3)  such(1)  the(1)  these(1)  they(1)  to(2)  together(1)
 traditional(1)  treated(1)  turn(1)  types(1)  typical(1)  under(1)  unit(1)  variables(2)  w(1)
```

3 String functions

To parse the input, compare words and measure their length, you can use the following *string-handling* functions[15]

[15] Note that some special characters, such as the apex " ’ " used for `char`, may have changed in this pdf.

```
int lowerCase(int *c){
        if (*c>='a' && *c<='z') return 0;
        if (*c>='A' && *c<='Z'){
                *c = *c + 'a' - 'A';
                return 0;
        }
        *c = '\0';
        return -1;
}

int stringLength(char *s){
        int i = 0;
        while (s[i] != '\0')
                i++;
        return i;
}

int copyString(char *in, char *out){
        int i = 0;
        while(in[i] != '\0'){
                out[i] = in[i];
                i++;
        }
        out[i] = '\0';
        return i;
}

int compareString(char *s1, char *s2){
        int i;
        for (i = 0; s1[i] == s2[i]; i++)
                if (s1[i] == '\0') return 0;
        return s1[i] - s2[i];
}
```