**Homework 5 - Trie for T9**

**Due: Wednesday, Dec 1st, 2021, at 11:59pm**

**Assignment goals**

In this assignment, you will develop a more complex program using dynamic data structures. In doing so you will:

● Gain experience developing a larger system one part at a time, testing as you go. ● Learn about the trie data structure, a version of a search tree.

● Gain experience working with trees, structs, and dynamically allocated data. ● Gain more experience reading and processing text files in C.

● Practice writing simple Makefiles.

You may choose to do this project individually, or with your partner (as usual). If you choose to work with your partner we suggest using a github or gitlab to coordinate your coding efforts, but please make sure they are private so no one steals your work.

**This is a significantly larger task than the previous programming assignments!**

**Note**: Kasey went over Tries and gave some sample code that should hopefully help you on this assignment in Lecture 15 (slides) and Lecture 16 (slides)

Lecture 16

**Synopsis**

In this assignment, you will build programs to implement T9 predictive text, a text input mode developed originally for cell phones and still used for numeric keypads. Each number from 2-9 on the keypad represents three or four letters, the number 0 represents a space, and 1 represents a set of symbols such as { , . ! ? } etc. The numbers from 2-9 represent letters as follows:

2 => ABC

3 => DEF

4 => GHI

5 => JKL

6 => MNO

7 => PQRS

8 => TUV

9 => WXYZ

Since multiple letters map to a single number, many key sequences represent multiple words. For example, the input 2665 represents "book" and "cool", among other possibilities.

2 6 6 5 2 6 6 5 a B c m n O m n O j K l OR a b C m n O m n O j k L | | | | | | | | ----------------------- ----------------------- | |

book cool

To translate from number sequences to words, we will use a data structure known as a trie. A trie is a multiway branching structure (tree) that stores the prefixes of sequences. As we travel down a path in the trie, we reach word sequences spelled out by the numbers along that path. Classic trie data structures have edges labeled with letters to store prefixes of strings. But for this application, we use a compressed trie that has only 10 possible branches at each node instead of 26, since the digits 0-9 represent the 26 letters, space and symbols. Because of this, an extra layer of complexity is needed to figure out the string represented by a path.

(Actually, for our application, each node only needs 8 possible children numbered 2-9, since digits 0 and 1 don't encode letters. But writing the code might be easier if nodes have 10 children numbered 0-9, since then subtree number *n* corresponds to digit *n*. Feel free to use either representation for the trie depending on which seems simpler to implement.)

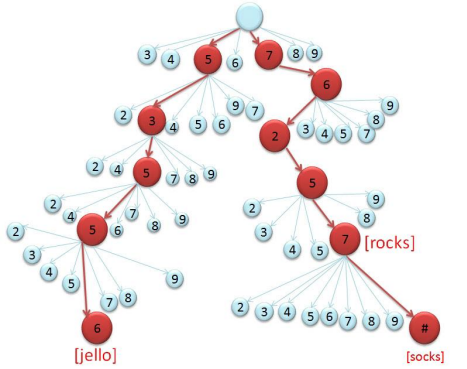
For more information on the trie data structure, here is a link to the Wikipedia article. **Technical Requirements**

Implement a program t9. You must use C for this assignment. The command $ ./t9 FILE

should read in a dictionary file (FILE) that contains a list of words. Translate each word in the dictionary into its numeric key sequence, then add the key sequence to your trie, with the word at the end of the path corresponding to the digits. If a word with the same

numeric sequence already exists in the trie, add the new word to the trie as a link to a new node with an edge labeled '#' instead of one of the digits 2-9. (The words linked from a node by the '#' edges essentially form a "linked list" of words that have the same numeric code, but we use additional tree nodes to link the words together instead of defining a separate kind of linked-list node just for this situation.)

For example, if the program reads the set of words "jello","rocks", and "socks" from the dictionary and adds it to an empty trie, the resulting trie should look like this:



Once your program has read the dictionary and built the trie containing the words in it, it should start an interactive session. The user should be able to type numbers and the program should print out the word corresponding to the sequence of numbers entered. Your program should use the numbers typed by the user to traverse the trie that has already been created, retrieve the word, and print it to the screen. If the user then enters '#', the program should print the next word in the trie that has the same numeric value,

and so forth. The user can also type a number followed by one or more '#' characters - this should print the same word that would be found by typing the number and individual '#' characters on separate input lines.

As an example, if we run the program using the above trie, an interactive session might look like this:

Enter "exit" to quit.

Enter Key Sequence (or "#" for next word):

> 76257

'rocks'

Enter Key Sequence (or "#" for next word):

> #

'socks'

Enter Key Sequence (or "#" for next word):

> 53556

'jello'

Enter Key Sequence (or "#" for next word):

> #

There are no more T9onyms

Enter Key Sequence (or "#" for next word):

> 76257#

'socks'

Enter Key Sequence (or "#" for next word):

> 76257##

There are no more T9onyms

Enter Key Sequence (or "#" for next word):

> 4423

Not found in current dictionary.

Enter Key Sequence (or "#" for next word):

> exit

The interactive session should terminate either when the user enters the word "exit" or when the end-of-file is reached on the interactive input (indicated by typing control-D on the keyboard on a separate line by itself).

**Note**: You can use printf to print to the console and scanf to receive user input from the terminal.

**Note:** Make sure your program properly handles the case if the user types more "#"s than there are T9onyms for a particular number.

We provide you with two text files, smallDictionary.txt and dictionary.txt (right-click on the links to download the files). Each of these text files contains a list of words to be used in constructing a trie - the small one primarily for testing, and the large one for the final program. Translate each word in the file into its associated T9 key sequence, and add the word to the trie. In the case of multiple words having the same key sequence *k*, let the first word encountered in the text file be represented by the key sequence *k*, the next encountered represented by *k*#, the next *k*##, etc. For example, 2273 can represent acre, bard, bare, base, cape, card, care, or case. To disambiguate, acre would be represented by 2273, bard by 2273#, bare by 2273##, and so forth. When a user inputs a key sequence, print the appropriate word.

Your trie data structure should contain nodes to represent the tree, and strings (char arrays) containing copies of the words read from the dictionary file, linked to appropriate nodes in the trie.

Besides the general specification given above, your program should meet the following requirements to receive full credit.

1. You should create a Makefile and use make to compile your program. Your Makefile should only recompile the necessary part(s) of the program after changes are made.

2. Use malloc to dynamically allocate the nodes, strings, and any other data that make up your trie.

3. If you need to create a copy of a string or other variable-size data, you should dynamically allocate an appropriate amount of storage using malloc and release the storage with free when you are done with it. The amount allocated should be based on the actual size needed, not some arbitrary size that is assumed to be "large enough".

4. Use standard C library functions where possible; do not reimplement operations available in the standard libraries.

5. You must check the return status (result code) of every library function you call to be sure that no errors occurred. In particular, malloc will return NULL if it is unable to allocate storage. Although this is extremely unlikely to happen, a robust program must check for the possibility and react appropriately if it does.

6. If an error occurs when opening or reading a file, the program should write an appropriate error message to stderr and terminate if there is no further work to be done.

7. Before the program terminates, all dynamically allocated data must be properly freed (i.e., free everything acquired with malloc). This should be done explicitly without relying on the operating system to clean up after the program finishes.

8. Your code must compile and run without errors or warnings when compiled with gcc -Wall on klaatu or the CSE Linux VM. Your program should build without errors when make is used to run your Makefile. You are, of course, free to use other systems for development, and you should be fine as long as you have a relatively recent version of gcc. But we will test the code on the CSE Linux machines.

9. Your program should terminate cleanly with no memory leaks or other memory errors reported when it is run using valgrind. (Warning: valgrind slows down execution considerably. It will take several minutes to load the full dictionary.txt file and then free the resulting tree under valgrind. We suggest you use smaller input files during development to test for memory problems with valgrind.) If memory leaks are detected, valgrind's --leak-check=full option will be useful to generate more extensive messages with information about the memory leaks.

**Code Quality Requirements**

As with any program you write, your code should be readable and understandable to anyone who knows C. In particular, for full credit your code must observe the following requirements.

1. Divide your program into suitable source files (at least three) and functions, each of which does a single well-defined aspect of the assignment. For example, there should almost certainly be a header and source file for the trie data structure and the operations needed on it (create a new empty trie, insert a word, search, delete the trie, etc.), and an additional file that contains the the main part of the program that uses the trie data structure. Your program most definitely may not consist of one huge main function that does everything.

2. The header (.h) file for the trie (and any other header files) should only declare items that are shared between client programs that use the header and the file(s) that implement it. Don't include in the header file implementation details that should be hidden. Be sure to use the standard #ifndef preprocessor trick so your header files work properly if included more than once in a source file, either directly or indirectly.

3. Be sure to include appropriate function prototypes near the beginning of each source file for functions defined in that file whose declarations are not included in a header file.

4. Comment sensibly, but not excessively. You should not use comments to repeat the obvious or explain how the C language works - assume that the reader knows C at least as well as you do. Your code should, however, include the following minimum comments:

○ Every function must include a heading comment that explains what the function does (not how it does it), including the significance of all parameters. It must not be necessary to read the function code to determine how to call it or what happens when it is called. (But these comments do not need to be nearly as verbose as, for example JavaDoc comments.)

○ Every significant variable must include a comment that is sufficient to understand the information in the variable and how it is stored. It must not be necessary to read code that initializes or uses a variable to understand this.

○ Every source file should begin with a comment identifying the file, author, and purpose (i.e., the assignment or project).

5. Use appropriate names for variables and functions: nouns or noun phrases suggesting the contents of variables and the results of value-returning functions; verbs or verb phrases for void functions that perform an action without returning a value. Variables of local significance like loop counters or indices should be given simple names like i, n, or p and do not require further comments. Avoid names like count or flag that provide no useful information - use names that suggest the values being counted or the condition that is represented.

6. No global variables. Use parameters (particularly pointers) appropriately. Exception: It may be appropriate to use global variables for constant data like translation tables if the program is better structured this way.

7. No unnecessary computation or excessive use of malloc or free - these are expensive. Don't make unnecessary copies of large data structures; use pointers. (Copies of ints, pointers, and similar things are cheap; copies of large arrays and structs are expensive.)

As with the previous assignment, you should use the clint.py style checker to review your code. While this checker may flag a few things that you wish to leave as-is, most of the things it catches, including whitespace errors in the code, should be fixed. We will run this style checker on your code to check for any issues that should have been fixed. Please use the class discussion board if you have questions about any of clint's warnings and whether they can be ignored.

**Implementation Hints**

1. There are a lot of things to get right here; the job may seem overwhelming if you try to do it all at once. But if you break it into small tasks, each one of which can be done individually by itself, it should be quite manageable. For instance, design the structure of the trie and the structs (and struct fields) you need. Figure out how to add a single word to the trie before you implement the logic to process all of the words in the dictionary. Figure out how to add a few words that have different

numeric codes before you handle words that have the same codes. Implement the code to traverse the trie to translate an input key sequence into the corresponding word once you've built the trie, not before.

2. Before you start typing code into the computer, spend some time sketching out data structures and code (particularly trie node structs) on paper or on a whiteboard. Be sure you understand what you are trying to do before you start typing.

3. Every time you add something new to your code (see hint #1), test it. **Right Now!** It is much easier to find and fix problems if you can isolate the potential bug to a small section of code you just added or changed. gdb and printf are your friends here to examine values while debugging.

4. You will probably find it very useful to include code that can print the contents of the trie in some understandable format. This is not required, but how can you be sure your code is correct if you can't look at the trie that is built for a small set of input data?

5. Start with a small data file and figure out in advance what the resulting trie should look like. Then verify that the program does, in fact, create that trie. 6. gdb is your friend.

7. To build the trie, you need some way to translate characters (primarily letters) from words in the dictionary file to the corresponding keypad digits. It is probably a great idea to include in your code a function that takes a character as an argument and returns the corresponding digit. This can be implemented with a series of if-elseif-else statements, but another way to do it is to have an array with one entry for each possible character. In the entry for each character, store the corresponding digit code. Then you can look up the code for a character without a complicated nest of if statements. (Recall that in C, as in Java and other languages, a character can be used as a small integer. That is particularly helpful for exactly this kind of application, where we might want to use a character value as an index into a table of data.)

8. Be sure to check for errors like trying to open a nonexistent file to see if your error handling is working properly.

9. Once you're done, read the instructions again to see if you overlooked anything. 10.Reread the previous hint and obey.

**Test Sequences**

The sequences below can be used to validate your trie against the supplied dictionary.txt file.

1. 22737: acres, bards, barer, bares, barfs, baser, bases, caper, capes, cards, carer, cares, cases

2. 46637: goner, goods, goofs, homer, homes, honer, hones, hoods, hoofs, inner 3. 2273: acre, bard, bare, barf, base, cape, card, care, case

4. 729: paw, pax, pay, raw, rax, ray, saw, sax, say

5. 76737: popes, pores, poser, poses, roper, ropes, roses, sords, sorer, sores

**What to turn in**

You will need to submit your Makefile and all source code required to make this project. Each file should have your name(s) and date in a comment at the top.

Submit your files to the Gradescope assignment.