CS2630: Computer Organization
Homework 2
T9 predictive text
(Data structures in C)

1 Table of Contents

2 Goals for this assignment

- Write a useful interactive application in C
- Implement a dynamic data structure, managing memory properly with malloc and free

3 Preparation

You should have completed the lab assignments, readings, and knowledge checks through the end of Lab 3. Most relevant are the two lectures on data structures in C (with companion code: liststart.c/listfinish.c) and lab 3's binary search tree.

4 Submission

- t9.c
- report.pdf

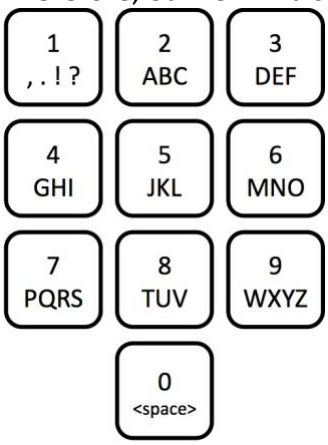
5 What is T9?

In this assignment, you will build programs to implement T9 predictive text, a text input mode available on many old cell phones, automated customer service phone systems, and keypads. Each number from 2-9 on the keypad represent three or four letters, the number 0 represents a space, and 1 represents a set of symbols such as { , . ! ? } etc. In this assignment, we will only use the numbers 2-9, which represent the following letters:

- 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. Therefore, our T9 will also accept “#” as an input to scroll through words with the same code.

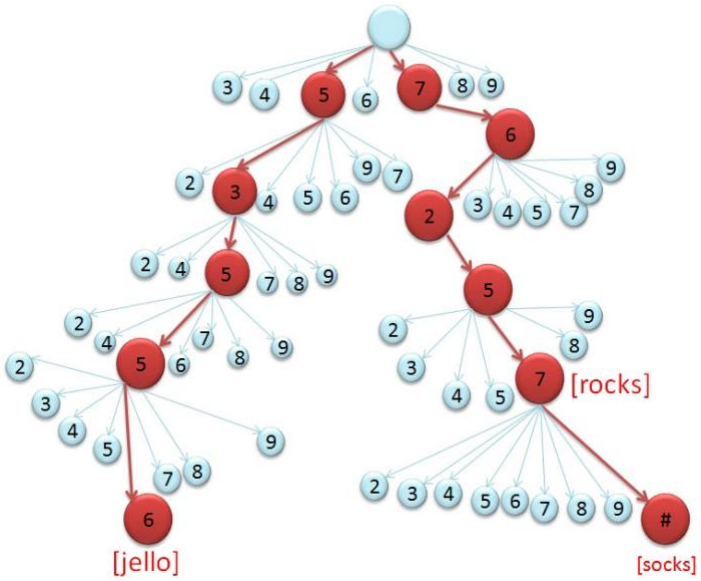


To translate from number sequences to words, we will use a data structure known as a *trie*.

6 What is a trie?

A trie is a multiway branching structure (tree) that is searched using 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 8 possible branches at each node instead of 26. The digits 2-9 represent the 26 letters. A node can represent multiple words so an extra layer of complexity (#) is needed to figure out the string represented by a path.

For more information on trie data structures, read the Wikipedia article. Note that the general concept of a trie as “a multiway branching structure (tree) that is searched using the prefixes of sequences” has many different realizations. Example tries you see online might not be exactly the structure that this T9 project needs. For example, many example tries you might find are searched by the word prefixes, not by a numeric T9 code.



In the above **abstract** representation of a T9 trie that contains

- jello at code 53556
- rocks at 76257
- socks at 76257#

7 Step 1: Insert

The goal of this section is to define a trie data structure and be able to insert words into it.

Specifically, you'll want to pass the test suite called "Insert".

```
> make trieTest
> ./trieTest

Running suite(s): Insert
100%: Checks: 5, Failures: 0, Errors: 0
Insert: # of failed: 0
…
```

In trie.h, you'll need to provide a definition for your struct TrieNode. We recommend one of the four options in 13.2.

Create a file called trie.c. At the top make sure to include the trie declarations with this code: `#include "trie.h"`

In trie.c, you should write definitions of (almost) all the functions declared in trie.h. We say "almost all" because right now you should skip trieNode_search and trieNode_free. For now, just give those two default definitions like returning NULL.

For trieNode_insert, to find where the word will go you'll need to convert the word to a T9 key sequence, represented as an array of integers. 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 child at index 10 (representing #). See the rocks/socks example above.

Note that trieNode_getWord and trieNode_getChild are necessary because they allow the provided tests (in trieTest.c) to abstract away your particular definition of a struct TrieNode.

You are probably ready to move on when you pass the Insert tests. Although, you might want to add some of your own test cases.

8 Step 2: Search

The goal of this section is to do T9 searches on your trie data structure.

Specifically, you'll want to pass both test suites: "Insert" and "Search".

```
> make trieTest
> ./trieTest

Running suite(s): Insert
100%: Checks: 5, Failures: 0, Errors: 0
Insert: # of failed: 0
Running suite(s): Search
100%: Checks: 3, Failures: 0, Errors: 0
Search: # of failed: 0
```

You should now provide a definition for trieNode_search in trie.c.

You are probably ready to move on when you pass the Insert and Search tests. Although, you might want to add some of your own test cases.

9 Step 3: Freeing the trie

The goal of this section is to properly free your trie data structure.

Start by running valgrind on the test program to check for leaks. Note that you need to include the CK_FORK=no because we are using the Check unit testing library.

CK_FORK=no valgrind --leak-check=full ./trieTest

You'll probably see lots of memory leaks because you haven't implemented trieNode_free yet. Go ahead and do that now.

You are ready to move on when valgrind reports no memory leaks. You should also make sure valgrind is reporting no errors of any kind.

10 Step 4: T9 application

Now that you have a tested and debugged implementation of your trie, it is time to use it in a full application!

Implement in C a program called `app`. This is how you build and run it:

```
make app

./app DICTIONARY TESTFILE
```

For example
```
./app dictionaries/smallDictionary.txt tests/small2.txt
```

This program should read in a dictionary file (`DICTIONARY`) that contains a list of words. It should insert each of these words into the trie.

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 the diagram above.

Once your program has read the dictionary and built the trie containing the words in it, it should start reading the test file (`TESTFILE`). Each line is a T9 code, and for each one, your program should print out the word corresponding to the sequence of numbers entered. Your program should use the numbers to traverse the trie that has already been created, retrieve the word, and print it to the screen. If the test file line is '#', the program should print the next word in the trie that has the same numeric value, and so forth. The test file might also include 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.

When your program encounters a line in the `TESTFILE` that says "exit", you can free the trie and end the program.

As an example, if we run the program using the above trie, and the following test file…

76257
#
53556
#
76257#
76257##
4423
exit

…then the program will print…

```
rocks
socks
jello
There are no more T9onyms
socks
There are no more T9onyms
Not found in current dictionary.
```

The program should terminate either when it encounters "exit" or the end of the file.

Make sure your program properly handles the case where there are more ‘#’s than there are T9onyms for a particular number.

We provide you with two dictionary files, smallDictionary.txt and dictionary.txt. 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 making sure your program works on large inputs.

We also provide you with several test files. Each contains a sequence of T9 codes.

Feel free to make your own dictionary or test files, too!

11 Step 5: Check for errors

Make sure app and testTrie both compile without any compiler warnings.

Make sure you run valgrind on `app` and `testTrie` and fix any memory errors or leaks.

12 What goes in report.pdf

12.1 Question 1: Summary

A brief description (3 sentences at most) of your trie data structure.

12.2 Question 2: Box-and-arrow diagram

A **box-and-arrow diagram** of your trie data structure when it has the contents:

- jello (at code 53556)
- rocks (at code 76257)
- socks (at code 76257#)

This is a box-and-arrow diagram for your particular trie data structure as you've coded it in C, NOT an abstract diagram as shown in Section 6.

Where relevant, make sure your box-and-arrow diagram includes slash (/) in boxes that were set to NULL and question mark (?) in boxes with an uninitialized value. Therefore, there should be no empty boxes.

12.3 Question 3: Lookup example

Give a step-by-step description of how your search algorithm looks for 76257## in your trie, when it has the contents given above.

12.4 Question 4: Debugging story

a) Describe one undesired behavior you encountered your program having as you were working on this project. Specifically, what was the ***expected*** behavior and what was the ***observed*** behavior?
b) For the undesired behavior above, describe what you did to investigate the potential cause of it.
c) Based on that potential cause, explain what you modified in your code, why you thought this change would fix the behavior, and what the new program behavior was.

13 Implementation tips

Here are some tips to help you write your code.

13.1 Reading files line by line

You are using two data inputs for the t9 program:

- a file for the dictionary
- a file for the T9 codes

The filenames for the both text files are provided to your program as command line arguments.

```
./app DICTIONARY TESTFILE
```

We have provided an example, readfile.c, of getting the name of a file from the command line and then reading and printing lines from that file. To run it, do

./readfile FILENAME

Note that readfile only takes one argument. In app.c you'll need to take two arguments specifying two files. A good first step to make sure you understand how to do this is to modify readfile.c so it takes in two filenames and reads and prints their contents line by line, one after the other.

13.2 Trie

Your trie implementation needs to involve struct(s), malloc, and free. When coding the trie, use an object-oriented approach. See the linked list in Week 3's liststart.c/listfinish.c and the binary search tree in Lab 3's tree.c for examples of an object-oriented approach to data structures. We are intentionally giving you the freedom to choose the particular way you represent your trie.

To give you some sense of the design space, below are four different ways you could define a trie node, along with a box-and-arrow diagram showing what it looks like.

Each one has different implications on memory usage and on how to manage dynamic memory with malloc and free.

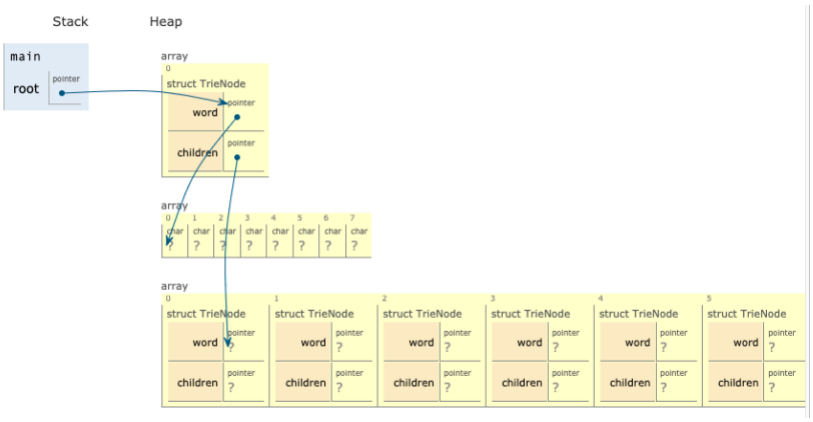
You are not constrained to using one of the approaches below but they likely represent straightforward ways to implement your trie.

The following diagrams are incomplete examples, so boxes having a question mark should not necessarily be assumed to be uninitialized; they could for example contain NULL or a legitimate arrow.

13.2.1 Approach 1

```
struct TrieNode {
      char * word;
      struct TrieNode * children;
}
```

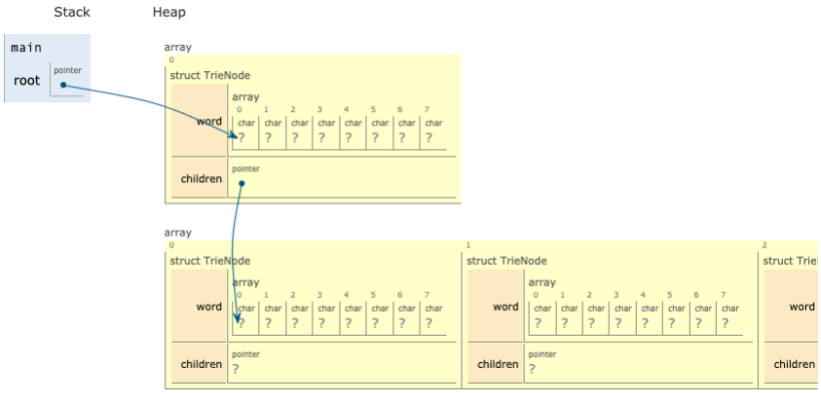
- Array pointed to by word needs to be dynamically allocated
- Array pointed to by children needs to be dynamically allocated



13.2.2 Approach 2

```
struct TrieNode {
	char word[MAX_CHARS_PER_WORD];
	struct TrieNode * children;
}
```

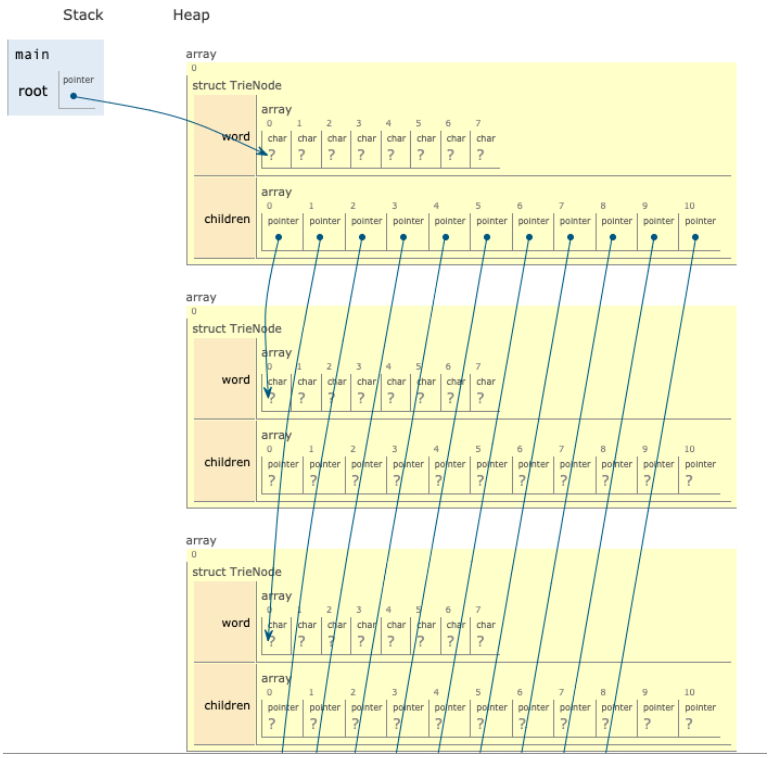
- Array of chars for word is part of the TrieNode
- Array pointed to by children needs to be dynamically allocated



13.2.3 Approach 3

```
struct TrieNode {
	char word[MAX_CHARS_PER_WORD];
	struct TrieNode* children[NUM_CHILDREN];
}
```

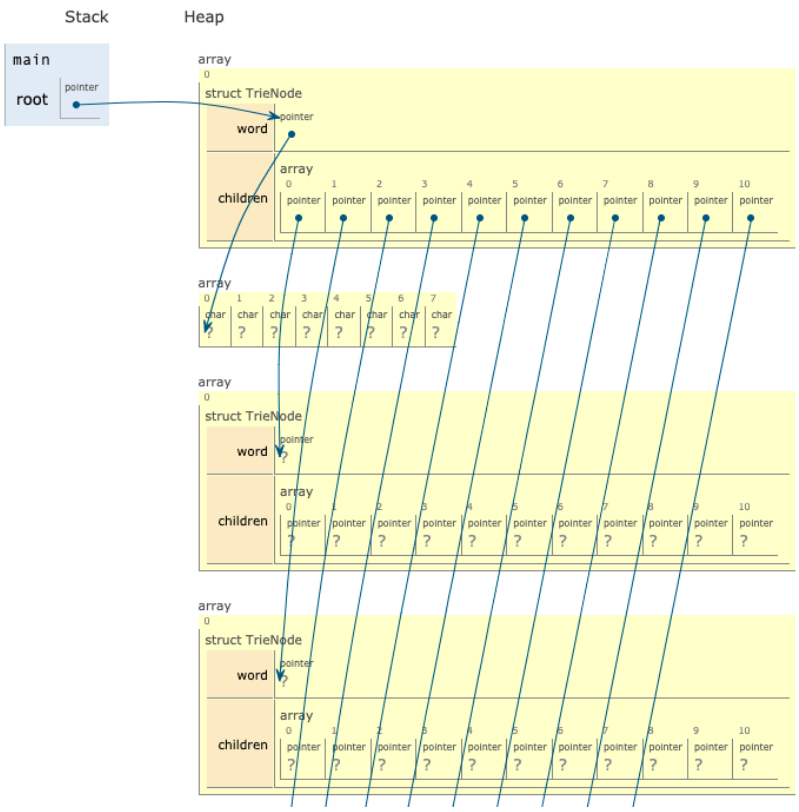
- Array of chars for word is part of the TrieNode
- Array of pointers to the children TrieNodes is part of the TrieNode



13.2.4 Approach 4

```
struct TrieNode {
	char * word;
	struct TrieNode * children[NUM_CHILDREN];
}
```

- Array pointed to by word needs to be dynamically allocated
- Array of pointers to the children TrieNodes is part of the TrieNode



14 Compiling the code

Here are the commands you can type to compile the code.

- make testTrie compiles testTrie.c into the executable testTrie
- make app compiles app.c into the executable app
- make readfile compiles readfile.c into the executable readfile
- make clean delete the executable files

15 Debugging tips

GDB is very, very useful when debugging your unit tests (trieTest). We recommend seeing what the line number is for the failed assertion. Then open gdb, set a breakpoint on that line number, and run. From the break you can then print relevant variables. Because of the way the Check library works, you need to run gdb like this

CK_FORK=no gdb ./trieTest

When using valgrind to check for memory leaks/errors during the unit tests, you also need that prefix.

CK_FORK=no valgrind --leak-check=full ./trieTest

When using gdb or valgrind on app, you do NOT need the CK_FORK=no. This is because app doesn't use the Check library.

16 Coding requirements

Just like with writing, it is unreasonable to expect to code "perfectly" on the first draft. Give yourself time to get the functionality working, and then refactor it until it meets the following requirements.

- You must **NOT** use any casts. This is generally bad practice unless you truly need them (e.g., implementing funky low-level stuff like memory allocators that involve casting between integers and different kinds of pointers). Why bad practice? Because casts force the compiler to stop complaining about code that often leads to bugs. If you are tempted to cast a const char* to a char* or a const struct TrieNode* to a struct TrieNode*, that's the wrong approach! Stop! These are pointers-to-const on purpose.
    - What is a const int*? You can read that using our right-to-left trick for reading C types: "pointer to constant int". It means that when you dereference the pointer, you can only read the data it points to; you cannot write it. For example…

      ```
      int y = 6;
      const int* x = &y;
      int g = *x; // this is okay! Just reading the constant data
      *x = 8; // this is a compiler error! Trying to modify constant data
      x = &g; // this is okay! The pointer itself is not constant, so it
      can be re-assigned
      ```
    - Note that assigning a pointer to a pointer-to-const is fine, because it is going from less strict to more strict. But you cannot go the other way. For example…

      ```
      const int* x;
      int* y;
      x = y; // this is okay!
      y = x; // this is a compiler error! Trying to circumvent the const-ness
      y = (int*) x; // a cast will quiet the compiler. BUT DON'T DO CASTS!
      ```
- Include at least the following ***comments***
    - Comment above every function saying *what* it does (not *how* just *what*), particularly what the arguments and return value mean.
        - You do NOT need to do this for functions declared in trie.h, since they already have WHAT comments.
    - Comment at top of the file saying 1. Your name, 2. Semester, 3. what the application does, including the flags.
- Use appropriate ***function names***: the function name should be a terse description of what the function does.
- Use appropriate ***variable names***
    - Longer names for important variables that are used for a long time
    - Shorter names for less interesting variables, such as a loop index

- Your program must not have any ***compiler warnings*** when compiled with the gcc flag -Wall. This flag is included in the Makefile, so we recommend you compile with make (as described in the earlier section).
- Your program ***must be portable***. You can typically achieve portability by: 1) not having any compiler warnings AND 2) not using any libraries other than the C standard library. We will grade the homework using the ***CLAS Linux machines*** (i.e., fastx), so we recommend you compile and run your program at least once on them. Alternatively to fastx, making sure it works on repl.it is fine.
- Use C ***standard library*** functions where possible; *do not* reimplement operations available in the standard libraries.
- Your program must have ***no memory leaks*** and ***no memory errors*** (e.g., invalid reads). Run it through valgrind to check (see lab 3 for examples). **TIP:** using the large dictionary.txt file will be very slow with valgrind. Use smaller dictionary files when testing for memory leaks. The staff will run valgrind on your program.
- The program needs to process dictionary.txt in a **reasonable amount of time** (less than 1 minute). If it is taking several minutes, you might be doing something algorithmically inefficient.
- Often **coding style** is enforced by your collaborators when they review your code. However, since this is a solo project, we suggest running the following beautifier on your code. It will fix spacing, indentation, and other basic style issues.

```
clang-format -i trie.c
clang-format -i app.c
```

This assignment derived from CSE 374 Spring 2015.