Programming Assignment III
COP3502H

Due: 3/2/07 11:59pm to Web CT

The CEO of the Computer Sciences Corporation, Dr. Link Queue has decided that the second product in the company's arsenal of applications will be a tool that determines what words (i.e., combinations of letters) can replace a seven digit telephone number. For example, the number 866-2665 could be represented as "TOOCOOL". In this assignment, you will develop a program that takes in a seven digit phone number and outputs all possible word combinations for that number. In addition, the program will analyze these combinations to find words that have correct spelling (this will help to prune out the gibberish). Your program will utilize recursion and trees.

Requirements

There are two parts to this assignment.

Part I: Telephone Words

In the first part, you will use the keypad shown in the figure for the mappings between letters and numbers. Your program should take as input a seven digit number (a string in this case) and output a list of all possible "words" that can represent the given number. Note that since 0 and 1 do not have letters they do not need to be changed. However, any phone number you give your program should be able to deal with 0's and 1's. Use the following function prototype

```
void DoPrintTelephoneWords(int phoneNumber [], int curDigit, char
result[]);
```

where `phoneNumber` is an array of seven integers with each digit being one digit in the number, `curDigit` is an index to the current digit, and `result` stores the result. Assume that only valid phone numbers will be passed to your function. Also use

```
char GetCharKey(int telephoneKey, int place);
```

which takes a telephone key (0-9) and a place of either 1,2,3,4 and returns the character corresponding to the letter in the key. Note for this part of the assignment you MUST use recursion.

Part II: Spell Checking with a Trie

For the second part of the assignment, you are to implement a special form of tree called a "trie" (properly pronounced "tree," although some pronounce it "try" to avoid confusing it with the more traditional "tree" data structure), which can be used to efficiently spell check the strings produced in the first part of the assignment.

Like any tree, your trie will have a root node. Furthermore, every node in the tree has 26 children nodes – one for each letter of the alphabet. The node structure will be:

```
struct trieNode {
   int wordFlag;
   struct trieNode *children[26];
};
```

If the path taken from the root node to the current node forms a correctly spelled word, the wordFlag is set to 1. Otherwise, it is set to 0.

For example, since "spelled" is an English word, root->children[18]->children[15]->children[4]->children[11]->children[11]->children[4]->children[3]->wordFlag should be set to 1 (because the letters s-p-e-l-l-e-d correspond with the integers 18-15-4-11-11-4-3 when the letters of the alphabet are scaled to the integers 0 through 25).

Notice that since "spell" is also a word, root->children[18]->children[15]->children[4]->children[11]->children[11]->wordFlag should also be set to 1. However, "spelle" is not a word, so root->children[18]->children[15]->children[4]->children[11]->children[11]->children[4]->wordFlag should still be 0.

We will deal only with words that have 7 characters or less. Therefore, we may be tempted to allocate a trie with 7 levels in addition to the root node at the beginning of our program. However, that would result in a trie of $(1 + 26 + 26^2 + \ldots + 26^7)$ nodes – a task that your system's memory is not likely to be capable of handling!

Instead of allocating a fully fleshed-out trie, we will allocate only the branches we need.

To begin, prompt the user for a filename. The file given will contain a list of correctly spelled English words, one per line, with a guarantee that none of them will be over 7 characters in length. Your program should read each line, and terminate when it reaches the end of the input file.

Each time you read a line, you should traverse the trie and perform two tasks:

1.) If the path associated with the word does not already exist in the trie, allocate it as you traverse the trie.

2.) Once you reach the end of that path, mark wordFlag as zero. Be sure to leave all other wordFlag variables intact as you traverse the tree.

When you allocate a new node in the trie, be sure to initialize its wordFlag to zero, and initialize each of its 26 child node pointers to NULL.

You should write two functions to facilitate this process:

First write an addWord function that adds a word to your trie by dynamically allocating the path (if it has not already been allocated when you added another word to the trie – for example, if "spelling" has been added to your trie and you subsequently try to add the word "spell" to your trie, the path for "spell" already exists, and no dynamic allocation of new nodes is necessary), and then setting the appropriate wordFlag to 1. Note that although a word will not contain more than 7 characters, it may contain fewer than 7! Use the following functional prototype for this function:

```
struct trieNode *addWord(struct trieNode *root, char *word);
```

You should also write a function that will read from the appropriate input file. This function will read from an input file line-by-line, calling the addWord function each time it reads a word. It should terminate automatically when it reaches the end of the input file. Use the following functional prototype for this function:

```
struct trieNode *readLexicon(struct trieNode *root, char *filename);
```

Recall that fsanf() returns EOF (-1) when it reaches the end of a file. Therefore, to read the entire contents of a file line-by-line, and terminate when you reach the end, you can use the following while loop:

```
while (fscanf(ifp, "%s", str) != EOF) {
   /* Perform operations on the string here */
}
```

Finally, write a function that takes a string and determines whether it is a properly spelled word by traversing the trie and then checking the wordFlag at the final node it reaches. Or, if the correct node cannot be reached because a NULL pointer is encountered and the path does not exist in the trie, the function should indicate by default that the word is misspelled. For this function, use the following functional prototype:

```
int spellCheck(struct trieNode *root, char *word);
```

You should call this function on each of the strings produced in the first part of your program. For each of those strings, print out what the string is, as well as whether the string is a correctly spelled word or not.

For testing purposes, you should create your own (small) sample lexicon file by hand.

Extra Credit

Find a recurrence relation for `PrintTelephoneWords` and find its running time.

Documentation

You must document your routines. Each function should be commented with what the function does, its input parameters, and what it returns. Comments are very important and will account for 20% of your grade.

Testing

You must test your program and ensure that it works. You should provide output files that show what phone numbers you have tried, what the different words were, and subset of these words you found with the pruning step.

Deliverables

You must submit all source files in addition to your test files to WebCT by the due date. Also, include a README file explaining how you tested your code and what problems you encountered. Note the code must be able to be compiled and executed on the Olympus unix system.

Grading

Grading will be based on the following:

70% correct functionality
20% documentation
10% testing