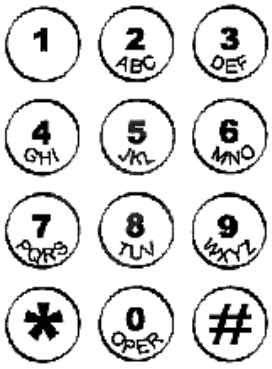
CSE 303, Spring 2009
Homework 5: T9, 60 points
Due Monday, May 18, 2009, 9:00 AM

This assignment focuses on implementing a complex linked data structure in C, pointers and memory management, strings, input files, multi-file programs, Makefiles, and working in a small group. It is inspired by a similar assignment given by Hal Perkins here at UW CSE. Turn in the following files from the Homework section of the course web site:

- `t9.c` - the main program and user interface
- `trie.c` - the code for your trie (prefix tree) data structure
- `trie.h` - the header for your trie (prefix tree) data structure
- `Makefile` - basic compilation manager script to build your program



T9 Predictive Text System:

As you know, a cell phone is not an ideal text input device because it has only the 0-9 number keys and a few other buttons (`*` and `#`). Each number is mapped to 3 or 4 letters as shown in the diagram at right. For example, the number 2 maps to the letters A, B, and C. On traditional cell phones you would enter text by pressing appropriate numbers multiple times. For example, to enter the word `HOMEWORK`, you would press the following keys:

Key(s) pressed	`44`	`666`	`6`	`33`	`9`	`666`	`777`	`55`
Resulting letter	`H`	`O`	`M`	`E`	`W`	`O`	`R`	`K`

Table 0.1 Keys pressed to type "HOMEWORK", without T9

More modern cell phones instead use a "T9" input system for predictive text, developed to allow input for text messaging on cell phones. The T9 algorithm allows the user to type each number only once; for example, `HOMEWORK` would be `46639675`. What's more, as the user begins to type the numbers of the word, the system suggests what word the user might be typing. So if the user has typed `4663`, the T9 system will suggest any word that begins with [`ghi`], [`mno`], [`mno`], [`def`]. So words that begin with prefixes like "`home`" or "`good`" would be suggested. If the user presses 0 (which represents a space), the T9 system auto-completes the rest of the word. If more than one word matches, the user can press `#` to move to the next suggestion until finding the right one. This system saves the user a lot of typing. Some T9 devices get smarter as they are used, suggesting more frequently typed words first.

Prefix Tree ("Trie") Data Structure:

At the core of the T9 algorithm is a special advanced data structure called a prefix tree or *trie* (pronounced like "try"). A trie is a multi-way branching linked tree structure that stores prefixes (beginnings) of text sequences. It has a root and nodes, much like a binary tree, but instead of each node having a left and right child, a typical trie has a child for each letter of the alphabet. So a standard trie node has pointers to 26 children, one for each letter A-Z. In our case, a T9 trie is slightly different; each node has 8 children, corresponding to the possible numbers the user might press from 2 through 9. The data stored in each node is a linked list of all words that start with the numbers used to reach that node.

A trie is built from input data by examining each letter of each word submitted to it and following the appropriate child links, creating nodes at each step. For example, if you want to represent the word "cse", you create the root node, then its 2 child ('c'), then that node's 7 child ('s'), then that node's 3 child ('e'), and storing "cse" in each node's list of words along the way. Figure 0.1 on the next page shows a trie that stores the words "bat", "book", "con", "cool", and "cse".

Now suppose we have built a large trie representing all words in a dictionary. How do we use it? Performing a lookup on a trie involves a similar process to building it, following the correct child links at each step. Suppose the user has pressed the numbers `266`. We begin at the root of our trie and follow the 2, 6, and 6 links. If any link along this chain is null, that means there is no word for these numbers, so the input is not a valid word. Assuming for the moment that all links on the path are non-null, we eventually reach a node that stores the word "con". If the user presses 0 (Space) to indicate that the word is done, we know that the word the user intended was "con," so this is what is output by the tree.

One tricky issue in a T9 trie is dealing with conflicts. Since each number on the phone corresponds to 3-4 letters, more than one word can map to the same sequence of numbers. For example, if the user had pressed `2665`, we would have

gone one node deeper in the trie and discovered a node that contains both "book" and "cool". How do we know which one the user intended? We don't, so we must simply suggest one of the possible words and let the user decide. A truly smart trie would have some notion of what words are the most popular and would present the most likely/popular word. Our trie will not attempt to be smart in this way; we will just present the first possible word, "book". If the user doesn't like this word, he/she can type # to cycle to the next possible word, "cool". Another # key press goes back to "book".

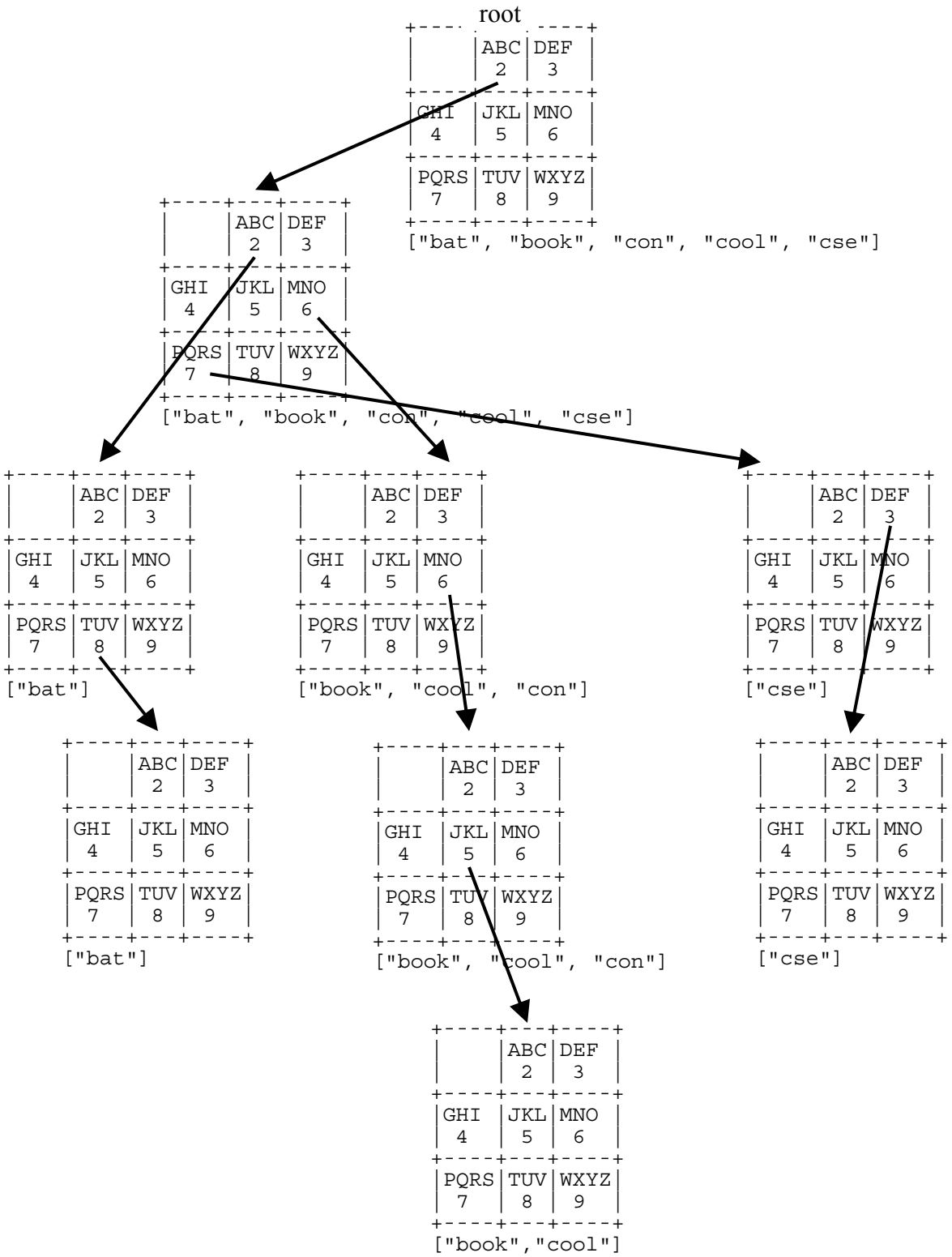


Figure 0.1 Trie prefix tree containing "bat", "book", "con", "cool", and "cse"

Program Description:

Your program should compile into an executable named `t9`. Your `t9` program should accept a mandatory command-line argument specifying the dictionary file from which to build the trie. If no such argument is specified, your program should build the trie from the file `dictionary.txt`. If any error occurs during file I/O, such as an unreadable or missing file, your program should print an error message of your choice and exit.

Your program will present a console user interface where the user can type characters for T9 input. The idea is that the user builds a word, number by number, and then types 0 (space) once finished with the word; then the input for the next word begins. The input could be one character per line or multiple characters in a row followed by a line break. Read your input from the console a single character at a time (we suggest using `getchar`). The user may type the following characters; any other character should be ignored.

- `2` - `9` traverses the tree using that number as the next number in the current word
  (if the trie has no words that have this as the next number, the number is ignored)
- `0` ends the current word and starts a new word
- `#` advances to the next word in the possible completions list (wraps back to beginning if at end)
- `*` lists all possible completions for the current word prefix
- `?` prints debugging information of your own choosing (such as info about your trie state)
- `1` exits the program
- *any other key* re-prints the initial "phone keypad" text from the start of the program

Your t9 program should accept an optional second command-line argument `-q` specifying "quiet mode." If this argument is given, then your program should print no output other than the display of the completed word when the user types `0`. This argument is most useful when redirecting input to your program from an external file.

The following are two example logs of execution of the program, one in standard mode and one in "quiet" mode. Assume that the file `input.txt` referred to in the right log contains the same set of characters as were typed in the left log.

```
$ ./t9 dictionary.txt
Welcome to the CSE 303 T9 simulator!
1=quit 2=ABC 3=DEF
4=GHI  5=JKL 6=MNO
7=PQRS 8=TUV 9=WXYZ
       0=spc

next number? 4
in
next number? 3
he
next number? 5
help
next number? 5
hell
next number? 6
hello
next number? 0
you typed the word: hello
next number? 435560
in
he
help
hell
hello
you typed the word: hello
next number? !
1=quit 2=ABC 3=DEF
4=GHI  5=JKL 6=MNO
7=PQRS 8=TUV 9=WXYZ
       0=spc

next number? 9
you
next number? 6
you
next number? 7
work
next number? #
world
next number? 0
you typed the word: world
next number? 1
```

```
$ ./t9 dictionary.txt -q < input1.txt
hello
hello
world
```

In this program you will begin by reading a dictionary input file of words and building a trie from those words. You may assume that the dictionary file consists of valid input containing exactly one word per line, as in Figure 0.2. Each word consists entirely of lowercase letters from a-z. The file could contain just a few words or a very large number of words. The words might be alphabetically sorted, or they might be arranged in some other order, such as in order of most to least frequently used words. You may assume that no word is more than 99 letters in length.

```
apple
bar
basic
book
books
car
...
```

Figure 0.2 Example partial input file, `tiny.txt`

Trie Implementation Details:

We are not going to rigidly specify exactly what code and functions your trie file must contain. This is up to you as part of designing your program. However, we strongly suggest that you have functions to perform the following actions (you can have others as well):

- Adding a word to the trie.
- Printing the entire trie (probably recursively).
- Walking a step down the trie based on a given number from 2-9.
- Cycling through the possible words to match a given prefix.
- Displaying the current word/words that match the user's current word prefix.

The rough idea is that the main `t9.c` code reads the user's next character and then uses this to decide what function to call from your `trie.c` code. The `t9.c` file should not be doing the bulk of the work to implement these operations. Nor should `trie.c` be directly reading user input.

Since there will only ever be a single trie structure in your program at a time, it is fine if you want to declare the trie's overall root as a global static variable in `trie.c` and manipulate that root structure in all of your trie functions.

Since the structure of a tree is self-similar, some of your trie methods will be recursive. C does not allow overloading of functions; that is, you cannot have two functions with the same name but different parameters. If you have two versions of the same function (as with public/private pairs seen in recursive CSE 143 assignments), you need to give them distinct names. Note that if you have a bug that leads to infinite recursion, the program will crash with a segfault.

Linked List:

Each node of your trie needs to store a linked list of strings representing the words that begin with that node's corresponding sequence of numbers. To help you solve this part of the problem, you are provided with files `list.c` and `list.h` containing a C implementation of linked lists of strings. (You don't HAVE to use this library, but you probably should. If you re-implement your own linked list, you are responsible for it working properly.) You can store a field of type `List*` in each trie node to represent its list of words. The linked list code provided contains the following methods:

```
List* list_new(void);                                  // construct new list
void  list_add(List* list, int index, char* word);     // add at a given index
void  list_append(List* list, char* word);             // add at end of list
bool  list_contains(List* list, char* word);           // true if word is in list
void  list_free(List* list);                           // free list memory
void  list_free_deep(List* list);                      // free list and its strings
char* list_get(List* list, int index);                 // return element at an index
int   list_index_of(List* list, char* word);           // 1st index of word (or -1)
void  list_insert(List* list, char* word);             // add at front of list
bool  list_is_empty(List* list);                       // true if list size is 0
int   list_last_index_of(List* list, char* word);      // last index of word (or -1)
void  list_print(List* list);                          // prints, e.g.  [a, b, c]
void  list_print_plain(List* list);                    // prints, e.g.  a b c
char* list_remove(List* list, int index);              // remove/return from index
void  list_remove_element(List* list, char* word);     // remove 1st occurrence
```

```
void  list_set(List* list, int index, char* word);    // set element at index
int   list_size(List* list);                          // returns list size
```

Detailed descriptions of each method are available in the list source code files. For the most part, the functions behave similarly to methods of the `java.util.List` interface from Java. Note that a list must be created with `list_new`, and the list must be passed as the first parameter to all of the other functions. For example:

```
List* mylist = list_new();
list_append(mylist, "hello");
list_append(mylist, "how r u");
list_append(mylist, "goodbye");
list_print(mylist);             // [hello, how r u, goodbye]
list_remove(mylist, 1);
list_print(mylist);             // [hello, goodbye]
```

If a list method is passed a null list or word, or an index out of bounds, it will print an error message and halt the program.

Makefile:

You should submit a basic make file named `Makefile` to build your application. Not many points will be assigned to this part of the program. Your Makefile must provide the following functionality (it can optionally have more if you like):

- The command `make` must compile your entire `t9` executable.
- The command `make clean` must clean up your directory by removing any .o files and your `t9` executable.

You won't be graded on the "internal correctness" of your Makefile, such as elegant rules or dependencies. As long the two targets specified above work properly, you will get full credit for this part of the assignment. All files in your program should produce no errors or warnings from `gcc -Wall`.

Extra Credit: Freeing Memory (+2 points):

To get full credit, your program does not need to free any allocated memory. However, you can choose to optionally free all of your program's heap-allocated memory for +2 points extra credit. To get the +2 points, `valgrind` should report "0 errors in 0 contexts" and "0 bytes in 0 blocks" in use at exit. If you are close but don't quite achieve the coveted "0 bytes in 0 blocks", you may still earn +1 point partial credit.

Freeing the memory can be tricky because of all the dynamically allocated structures you are likely to have, such as trie nodes, strings, and linked lists. We suggest that you save the freeing for last, after the other functionality has been written and tested. You should use the `list_free` or `list_free_deep` methods to free the memory of a linked list; the latter function also frees the memory used by the strings in the list. We suggest that you write a recursive function that frees all of a trie node's children, followed by the node itself. One tricky aspect of the freeing is that multiple nodes will store pointers to the same strings, and you do not want to free the same string multiple times. A suggested way of handling this is to store in the root trie node a list of every single word in the dictionary, and then when freeing, free the strings from this list. Then when freeing any child trie nodes of the root, you do not need to free their strings again.

Extra Credit: Better Command-Line Arguments (+1 point):

You can also choose to optionally add some additional flexibility to your program's handling of command-line arguments for +1 point extra credit. To earn this extra point, implement the following functionality:

- Add a third possible optional command-line argument `--help` that, when passed, will cause the program to print brief information to the user about usage of the program, such as its command-line arguments, authors' names, etc., and then exit. The `--help` parameter should override any other arguments passed.
- Make all of the command-line arguments be optional. If the argument for the dictionary file name is omitted, your program will default to `dictionary.txt`.
- The `t9` program should be able to accept its command-line argument(s) in any order. Any argument is assumed to be the dictionary input file name unless it is `-q` or `--help`. (You may assume that the user will pass no more than 1 command-line argument whose value is neither `-q` nor `--help`.) All of the following would be legal:

```
$ ./t9
$ ./t9 -q
```

```
$ ./t9 filename -q
$ ./t9 --help
$ ./t9 -q filename
$ ./t9 -q --help
$ ./t9 --help filename -q
$ ./t9 -q filename --help
```

Extra Credit: DEBUG Preprocessor Flag (+1 point)

You can also choose to optionally add good debugging code to your program for +1 point of extra credit. To earn this extra point, implement a preprocessor flag named `DEBUG` that, when set, causes your program to print a significant amount of relevant debugging information, such as information about the construction or traversal of your trie, various tests for state of lists or nodes or other relevant values, and so on. By default this flag should be turned off when you submit your program, but either by manually enabling it or by compiling your program with `gcc -DDEBUG`, the flag should enable and the extra output should appear.

You may do any number of these extra credit features: some, all, or none. It is possible to earn a score over the maximum for the assignment; if so, you will be allowed to keep the extra points above 100% credit. Your score will not be capped.

Groups:

You must work with one partner on this assignment. (If the course has an odd number of students, a single group of the instructor's choice will be given a third member.) You may work with anyone you like. Please register your group using the Grouper tool found on the course web site. Groups will be given some shared `attu` resources to be specified later.

Each group should submit only one set of files for the assignment. In most cases, both group members will be given the same grade for the program. If a group member did not do his/her share of the work, the other group member should contact the instructor immediately to discuss the situation. If there is a significant group problem, the instructor may ask to meet with one or both members to get more information and/or to possibly assess different grades to the members.

Please place a comment at the top of each file indicating which group member(s) worked on which parts of that file.

Lateness penalties will be applied individually to each student in the group based on how many late days that student has remaining. For example, if a group submits the project a day late and one student has a late day and the other does not, then the student with the late day will consume that late day, while the other student will receive a lateness deduction.

Development Strategy and Hints:

This program is hard. We suggest that you solve this assignment in stages, verifying that your code works properly after each stage. A good initial stage to reach would be to have the program files set up and compiling, and then to add functionality piece by piece. Here is a possible structure of things to work on:

- Read the dictionary file and print each word in it. Verify that you are reading the words correctly.
- Read each word of the dictionary file and print its T9 encoding (e.g. "hello" → 43556). Verify the encodings.
- Write initial code to build your trie from the dictionary words.
- Write code to print your tree to verify its contents.
- Try a larger dictionary and make sure that your tree is still built properly.
- Write the code to traverse the tree based on

Cycling through the possible completions for a word (when the user presses #) can be tricky, so you may want to delay that until later in your development.

Start out by testing with very small input files (two or three words) and building up from there to larger ones. Use a debugger such as `gdb` or `ddd` to verify that your trie is being built properly and that your pointers point where you expect.

You can verify the correctness of your output using `diff`.

You will likely need to use a debugger such as `gdb` to find and fix bugs. Recall that you can set a breakpoint at a particular line of a particular file by issuing the gdb command, `b` ***`file`*****`:`*****`line`**. For example, to break at line 27 of `trie.c`, you could write `b trie.c:27`.

You might want to type in some temporary aliases into your shell (not in your bash login script, though) while you're working on the program, such as:

```
alias m=make
alias c=make clean
alias r=./t9
alias d=gdb ./t9
```

A runnable sample solution to this program will be placed on `attu` in the executable `~stepp/hw5/t9` so that you can test its behavior. Attempting to reverse-engineer its source code is a violation of the course academic integrity policy.

Grading:

Over half of the points will come from the behavioral ("external") correctness and output of your program when run with various test cases. Getting correct output for the preceding test cases does not necessarily guarantee full credit. You should test your program using the provided input files as well as performing your own testing.

A significant amount of points will also come from the style, design, and "internal correctness" of your code. Avoid redundancy and overly complex logic as much as possible. Minimize the use of global variables. The only acceptable global variables are constants as `#define` macros or as variables declared with the `const` keyword. Use functions appropriately to split up your program to indicate its structure and also to capture repeated code that would be redundant.

Part of your grade comes from showing a good separation of functionality into the different files for the assignment. The `t9.c` file should contain all user interface and I/O code such as reading the dictionary file and prompting the user for input. `t9.c` should not contain any code to directly manipulate trie nodes. The `trie.c` file should contain all code related to the trie data structure, such as adding words to the tree, traversing the tree, freeing the memory for the tree, etc. `trie.c` should not contain any user interface code other than possibly printing the data of an individual node as needed.

Part of your grade comes from properly using header files in your multi-file program. The `trie.h` file should contain declarations for all functions and relevant data types related to your trie structure. No `.c` file should include any other .c file. Each `.c` file should be able to be successfully compiled into a `.o` file without warnings from `gcc -Wall -c`. Your `trie.c` file should not expose any global variables or functions that are not declared in its `.h` file; any such other variables/functions should be declared `static` to protect them from outside modification.

You may use libraries `stdio`, `stdlib`, `stdbool`, `string`, `ctype`, and `unistd` on this assignment, along with the provided `list.h`. You may not use any other pre-written data structures or other libraries without instructor permission.

Part of your grade will be based on appropriate allocation of memory. You should show an understanding of when to allocate data on the stack vs. on the heap, and proper use of pointers and `malloc` or `calloc`. In general you may assume that the computer has enough heap memory to accommodate all data structures you will need to create; for example, you don't need to test the result of `malloc`/`calloc` to see whether it is null. For reference, our solution to this program uses around 4MB of memory using the standard frequency-sorted dictionary provided.

Format your code nicely with proper indentation, spacing, and clear variable names. Place a descriptive comment heading at the top of each file describing the assignment as well as describing the purpose of that file. Place a descriptive comment header atop each function, type declaration, global variable declaration, and brief comments throughout your code inside methods explaining what each major section of code is doing. The .c files should generally have more detailed commenting than the `.h` files; see our `list.c` and `list.h` as examples.

For reference, our `trie.c` is around 165 lines long (85 lines long if you exclude blank lines and comments) and our `t9.c` is around 90 lines long (65 without blanks/comments), though you do not need to match this exactly to get full credit; it is just a rough guideline.

Different computers can behave differently with the same C program; for full credit, your program must compile and run successfully on `attu` or on the basement lab computers.