Programming in C

Deadline: *March 20, at midnight.*

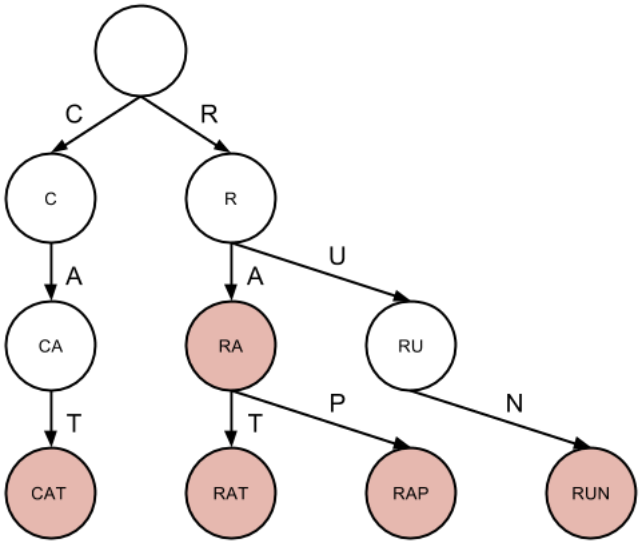
Assignment 4: Autocomplete

You have probably encountered it before: you start typing a word into a website or application and your computer finishes the word for you. This so-called *autocomplete* functionality has become very popular because it saves users a lot of time. In this assignment, you will delve deeper into how such suggestions work and you will build your very own C program with autocomplete functionality.

The trie

To implement your autocomplete, you will use a data structure known as a trie (pronounced either *tree* or *try*, it's one of the more fiercely debated topics in computer science). A trie is a variant on the tree data structure that you have encountered before and is often used to store strings. The root of the trie will represent an empty string and can have many children: one for each character. Each node in the tree can, in turn, have its own children to form longer strings.

In the example below, we have illustrated a trie with the words *cat*, *rat*, *rap*, *run* and the Egyptian solar god *Ra*. It is worth noting that, while the nodes have text on them, nodes do not actually store any strings. A node only stores a value indicating if that node terminates a word. Such nodes have been marked red in this illustration. Each link to a child node has a character associated with it in the trie. The characters stored with the edges make up the actual words stored in the trie when you walk from the root to a red node that terminates a word.



Designing your trie

You should start by thinking about how you will implement your trie. Remember the fields which are contained in a node of a trie: a value indicating if the node terminates a word and, of course, links which are identified by a letter of the alphabet.

Populating your trie

After you have implemented your trie, you need to be able to populate it. Indeed, an autocomplete feature which doesn't actually know any words to complete is not very useful. Your program will need to read its way through a large file containing many words and add each word to the tree. To add a word to a trie, consider the following algorithm:

1. Start out in the root node.
2. Pop the first letter from the string. If the string is empty, mark the current node as a valid word and terminate the algorithm.
3. Does a link exist which is labeled with this letter and which exits the current node? If not, create such a node and the corresponding link.
4. Move along the trie using the link found or created in step 3 and return to step 2.

Your autocomplete program should be able to add words to the trie and print all the words in the trie. If you have implemented this functionality, congratulations! You should be well on your way to finishing your autocomplete program.

Completing strings

After you have populated a trie, you will want to start completing strings. To sketch how this functionality should work: you should be able to supply the program with a prefix and the program should be able to return to you a list of all the words in the trie that start with that prefix.

Using the supplied prefix, you can traverse the tree up to a certain node. The list of all words starting with that prefix is then the list of all nodes which are marked as terminating a word *and* which are descendants (children, grandchildren, great grandchildren, etc.) of that node. Consider using one of the search algorithms which you have learned about, such as **depth first search** to find these nodes.

Example output

You will be using your program on very large datasets, but for now let's consider the following piece of text, contained in the file test1.txt:

```
Hi, how are you doing today? I am doing quite well, thank you!
```

Your program should accept the `-p` flag which accepts a string as the prefix to look for, followed by the name of a text file. For example, the command `$ ./autocomplete -p t test1.txt` should look for all words starting with the letter *t* and should output something like this:

```
$ ./autocomplete -p t test1.txt
Words added to trie: 11
All words starting with t:
thank
today
Number of words in trie: 11
```

Also include a `-P` flag which prints all the words in the trie. This is also very useful for debugging. Example output for this could be as follows:

```
$ ./autocomplete -P test1.txt
Words added to trie: 11
All words stored:
well
quite
I
thank
today
doing
you
am
are
```

```
how
Hi
```

Removing words

It is really helpful if an application autocompletes words for you. It is also really annoying if you misspell a word once and it keeps suggesting that misspelled word to you. So now that you have your trie up and running it is time to add the ability to remove words from the trie. Depending on the location of the word zero or more trie nodes need to be removed when a single word is removed from the trie. If the word *cat* is removed from our example trie you will need to remove three trie nodes. But if the word *ra* is removed from the trie no trie nodes need to be removed.

Extend your autocomplete program so that it accepts a new flag `-r file` that removes all the words listed in `file` from the trie. After the words are removed the modified trie should be printed again.

Template code

The template code that you can use for this assignment consists of the following three source files:

`trie.h`
: The header file that contains the prototypes of the trie functions.

`trie.c`
: Should contain the implementation of these trie functions declared in the header file.

`autocomplete.c`
: The main program that reads words, builds the trie and autocompletes prefixes.

To parse arguments from the commandline `autocomplete.c` uses the `getopt()` function. Please read the template code carefully and make sure you understand what it does before making changes to it. Always refer to the manual pages of an unknown function. Don't assume some behaviour that you think is logical. This will save you time in the end.

In addition to these source files, it also contains a Makefile and some test input. The code compiles, but is ofcourse not yet functional.

Bonus

When an application recommends autocomplete suggestions it can be convienent if they are ordered by the frequency of their occurence in the input text. This is what the soft-keyboard of your phone does when it provides you with suggested words. As a bonus assignment you can add this behaviour to your autocomplete implementation.

PAV Report

The Informatics students that follow PAV will also need write a report on this assignment. For more information on the contents you should go to the blackboard page of PAV.

For the report you will also need to perform timing experiments on your trie implementation. Measure how long it takes to fill the trie with different amounts of input data and plot your measurements. Can you explain the plot? What do you expect if you would use the linked list from the previous assignment to store the words? Note that the list would have to be kept in alphabetical order while it is filled to ensure that it will not contain duplicates.

You can also measure the time it takes to find the prefixes. See if you can find a relation between the size of the trie and the time it takes to find the prefixes.

For timing your code you can use the `clock()` function from `<time.h>`. When you measure your code do not use `printf()` in the code you are measuring. This will effect your timing results.