Computer Science 2211b
Software Tools and Systems Programming
Winter 2024

Assignment 4 (7%)

Due: Friday March 22 at 11:55 PM

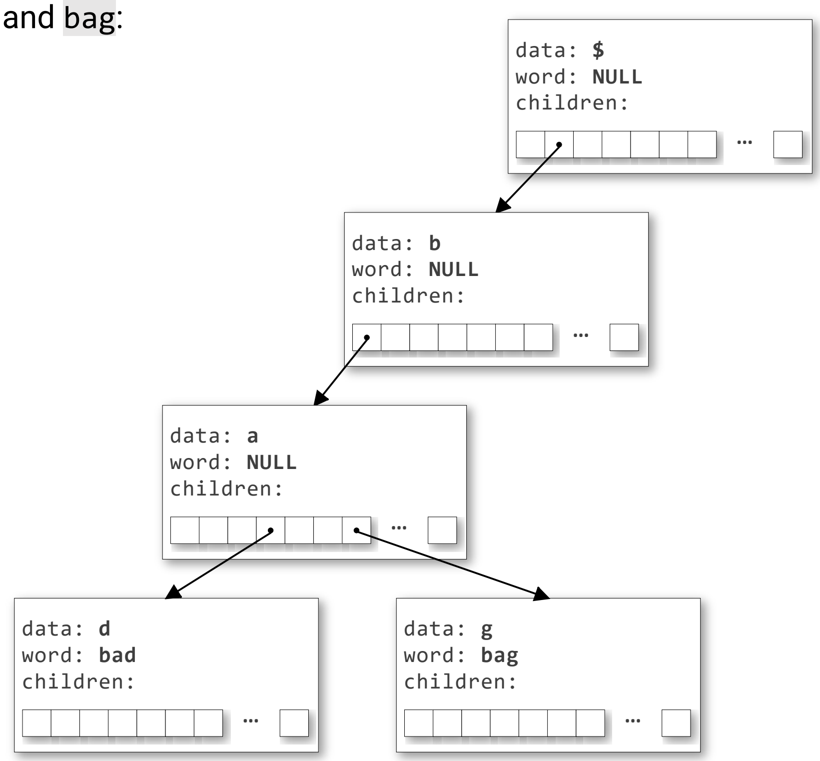
Objectives

In this assignment, you will gain experience with:

- Working with pointers and structs
- Building and using a data structure in C
- Dynamically allocating memory
- Following style conventions and good programming practices

Introduction to Tries

In this assignment, you'll build a data structure called a *trie* (usually pronounced "try", though it comes from the word *retrieval*). This is a tree structure that can be used for efficient storage and retrieval of strings. Each node in a trie represents a character, and paths from the root to leaf nodes represent words. Tries are commonly used for tasks such as autocomplete and spell checking. The following is a sample trie that contains the words `bad` and `bag`:



Each node in the trie has the following members:

`data`	The character stored by the node. The root uses the special character `$` to denote that it's the root.
`word`	Contains the word the node represents, if it's the end of a word. Otherwise, this is set to `NULL`. In a traditional trie, this is not strictly needed, but it simplifies certain operations.
`children`	An array of 26 pointers to trie nodes -- one for each letter of the alphabet: • Element `0` points to a child node representing the character `a` • Element `1` points to a child node representing the character `b` • … • Element `25` points to a child node representing the character `z` We will only deal with lowercase letters in this assignment. Initially, all elements of this array are `NULL`. As words are inserted into the trie, the appropriate child nodes are initialized for each character in the word.

Note that the leaves of the trie will all represent the end of a word. However, not all nodes that represent the end of a word are necessarily leaves, as they may have children that extend to form longer words. For example, in a trie containing the words `cat` and `catch`, the node representing `t` in `cat` is marked as the end of a word but is not a leaf, as it has a child node representing `c` for the word `catch`.

Assignment Skeleton

[redacted]

NOTE: DO NOT CLONE THIS REPOSITORY INTO YOUR REPOSITORY

Either clone it elsewhere outside of your repository and then move the files into your repository, or just download the files individually from the web site and put them in your repository.

The skeleton contains a header file `trie.h`. The structs defined in this header file are as follows:

```c
typedef struct trie_node
{
    char data;
    char* word;
    struct trie_node* children[ALPHABET_SIZE];
} trie_node;

typedef struct
{
    trie_node* root;
    int size;
} trie;
```

The `trie_node` struct represents a single node within the trie, containing the members described above.

The `trie` struct represents an entire trie, containing a pointer to its root node, along with the current size of the trie (the number of words it currently stores).

Do not modify this file. Any changes you make will be overwritten when we mark your assignment.

Part 1: trie.c

In this part, your job is to fill in the functions in `trie.c`, as described below.

```c
trie_node* trienode_create(char data)
```

Purpose	Creates a new trie node
Parameters	`data`: The character to be stored in the data field of the node
What to Do	• Allocate memory for a new `trie_node` • Initialize all `children` pointers to `NULL` • Set its `data` field to the character passed as a parameter • Set the `word` field to `NULL`
Returns	A pointer to the newly-created `trie_node`

```c
trie* trie_create()
```

Purpose	Creates a new trie
Parameters	None
What to Do	• Allocate memory for a new trie

<table>
<tr><td></td><td>Initialize the <code>root</code> field to a new trie node that stores the special character <code>$</code> as the data (used to denote the root node)
Set the <code>size</code> of the trie to zero</td></tr>
<tr><td>Returns</td><td>A pointer to the newly-created <code>trie</code></td></tr>
</table>

```
void trie_insert(trie* t, char* word)
```

<table>
<tr><td>Purpose</td><td>Inserts a word into the trie, if it doesn't already exist</td></tr>
<tr><td>Parameters</td><td><code>t</code>: A pointer to the trie into which the word is to be inserted
<code>word</code>: The word to be inserted into the trie</td></tr>
<tr><td>What to Do</td><td>
This is likely the "hardest" function, but we'll go through the logic step by step
Create a pointer variable to a <code>trie_node</code>
This will represent the current node in the trie
Initialize it to point at the root node of the trie
Create a pointer variable to a <code>char</code>
Initialize it to point at the first character of <code>word</code>
For each character in <code>word</code>
Get the next character (dereference your pointer)
Compute its index in the <code>children</code> array of the current node
If the character is <code>a</code> = the array index is <code>0</code>
If the character is <code>b</code> = the array index is <code>1</code>, etc.
Check if the current node has a child at this index
If not, create a new trie node at this index for the given character using <code>trienode_create</code>
Descend in the trie to this child node
Hint: update your current node pointer
Move to the next character in the word
Hint: update your character pointer

After all characters are processed
We are now at the child node that represents the end of the word. We wish to store the word in this node
Duplicate the word using <code>strdup</code> and store it in the node
This function handles dynamic allocation and copying of a string in one operation, but don't forget to free its memory later in your <code>trie_free</code> function
Increment the <code>size</code> field of the trie

</td></tr>
</table>

	• If the word already exists in the trie, this function should make no changes
Returns	Nothing
Note	This operation can be performed iteratively. If you prefer, you may also choose to create a helper function and perform it recursively

```
int trie_contains(trie* t, char* word)
```

Purpose	Checks if a word is present in the trie
Parameters	• `t`: A pointer to the trie in which to search for the word • `word`: The word to search for in the trie

What to Do	• The logic of this function is like that of `trie_insert`, except we're not inserting a new node
	• Instead, we iterate over each character in the input word, traversing the trie from the root node
	• If a child node for the current character is not found, it means the word is not present in the trie
	• If we reach the end of the word and the current node's `word` member is
	    ○ Not `NULL`: the word is present in the trie
	    ○ `NULL`: the word is not present in the trie (the path represents a prefix of another word but not a complete word itself)

Returns	• `1` (true) if the word is found in the trie `0` (false) otherwise

Example	• If the word `cat` exists in our trie, then `trie_contains` would return:
	    ○ `1` for the word `cat`
	    ○ `0` for the prefix `ca`

```
int trie_contains_prefix(trie* t, char* prefix)
```

Purpose	Checks if a given prefix is present in the trie
Parameters	• `t`: A pointer to the trie in which to search for the prefix • `prefix`: The prefix to search for in the trie
What to Do	• The logic of this function is similar to `trie_contains`, with the main difference being that we do not check if the current node represents the end of a word when we reach the end of the prefix.

	• If we can successfully navigate through the trie, following the child nodes corresponding to each letter of the prefix, then the prefix exists in the trie, and we return `1`
Returns	• `1` (true) if the prefix is found in the trie `0` (false) otherwise

Example	• If the word `cat` exists in our trie, then `trie_contains_prefix` would return
	  ○ `1` for the *prefix* `ca`
	  ○ `1` for the *prefix* `cat` (since a word is a prefix of itself)

Hint	• If you implement a helper function to find and return a pointer to the `trie_node` corresponding to a given string, **then `trie_contains` and `trie_contains_prefix` literally become functions that are each 1-2 lines of code**

```
void trienode_print(trie_node* node)
```

Purpose	Recursively prints (in ascending alphabetical order) all words in a trie starting from the given node
Parameters	`node`: The current node being visited
What to Do	• If the given node represents the end of the word, print the word • **DO NOT PRINT ANYTHING ELSE -- just the word and a newline** • Recursively call the function on each of the node's non-`NULL` children
Returns	Nothing

```
void trie_print(trie* t)
```

Purpose	Recursively prints (in ascending alphabetical order) all words in a trie starting from the root node
Parameters	`t`: The trie to print
What to Do	Recursively prints the trie starting from its root node
Returns	Nothing
Hint	You have a helper function for this

```
void trie_print_prefix(trie* t, char* prefix)
```

Purpose	Recursively prints (in ascending alphabetical order) all words in the given trie that begin with the given prefix
Parameters	• `t`: The trie from which to print words

	• `prefix`: Prefix of words to print
What to Do	• Recursively prints all words in the trie with the given prefix • If you implemented a helper function earlier to find the node corresponding to the end of a given word/prefix earlier, this function will be very short and easy • Simply find the node corresponding to the given prefix, then print the trie starting from that node
Returns	Nothing
Hint	You have a helper function for this

```
void trie_free(trie* t)
```

Purpose	Frees all memory allocated for the given trie
Parameters	`t`: A pointer to the trie to be freed
What to Do	• Frees the memory allocated for all nodes in the trie (don't forget the memory allocated for the words as well) • Frees the memory allocated to the trie itself • *Hint: you'll likely want a recursive helper function for this*
Returns	Nothing

Part 2: lookup.c

In this part, you'll fill in the main function in `lookup.c` to complete a program that uses your trie data structure. The program usage is as follows:

```
Usage: lookup [OPTION] [ARGUMENT]
Options:
  p <prefix>    Print all words with the given prefix
  c <prefix>    Check if the prefix is in the trie
  w <word>      Check if the word is in the trie
  (no option)   Print all words in the trie
```

Your program should:

- Create a trie
- Read words (one per line) from standard input, inserting each into the trie
- Parse the command-line parameters passed to the program (see the comment in `lookup.c` for how to do this)
- Take the appropriate action as noted above
- Free your trie

For example, suppose `words.txt` contains the following lines:

```
cherry
banana
apricot
apple
```

Running the program with no arguments prints the entire trie in alphabetical order:

```
$ ./lookup < words.txt
apple
apricot
banana
cherry
```

Running the program with the argument `p` and a prefix prints all words beginning with that prefix in alphabetical order:

```
$ ./lookup p ap < words.txt
apple
apricot
```

Running the program with the argument `c` and a prefix prints a `1` or `0`, depending on whether the prefix exists in the trie:

```
$ ./lookup c ap < words.txt
Prefix ap: 1
$ ./lookup c ki < words.txt
Prefix ki: 0
```

Running the program with the argument `w` and a word prints a `1` or `0`, depending on whether the word exists in the trie:

```
$ ./lookup w ap < words.txt
Word ap: 0
$ ./lookup w apple < words.txt
Word apple: 1
```

Otherwise, if the user enters any other command-line arguments, your program should call the provided `usage` function and exit (don't forget to free your memory before exiting!).

IMPORTANT: We will be using automated tests to grade your assignments and get them returned to you as soon as possible. If you do not wish to have marks deducted, it is important that your output is **exactly** as shown. For example, when the output above shows "`Word ap: 0`", that means your program should print *exactly* that string.

A Word on Memory Leaks

Don't forget to free the memory allocated for your trie before the program exits. If your program has a memory leak, marks will be deducted. Fortunately, there's a helpful program called `valgrind` that you can use to check for memory leaks. Here's an example of what `valgrind` reports when you have a memory leak:

```
$ valgrind ./lookup < words.txt
==9006== Memcheck, a memory error detector
.
.
.
==9006== LEAK SUMMARY:
==9006==    definitely lost: 16 bytes in 1 blocks       <---
==9006==    indirectly lost: 5,180 bytes in 27 blocks
==9006==      possibly lost: 0 bytes in 0 blocks
==9006==    still reachable: 0 bytes in 0 blocks
==9006==         suppressed: 0 bytes in 0 blocks
```

And here's what it reports when you *don't* have a memory leak:

```
$ valgrind ./lookup < words.txt
==9090== Memcheck, a memory error detector
.
.
.
==9090== All heap blocks were freed -- no leaks are possible
```

Test Files

The `testfiles` directory in the assignment skeleton repository contains some files you can use for testing your trie.

Submitting Your Assignment

Please note that no exceptions will be made for this assignment if it is not submitted correctly. You only need to enter 5 Git commands, which are provided below. As university students, I trust you can manage this. Make sure to follow the instructions carefully.

Your programs must follow the **CS 2211 Style Guide** introduced in Topic 5.

After completing this assignment, your `asn4` directory should look as follows:

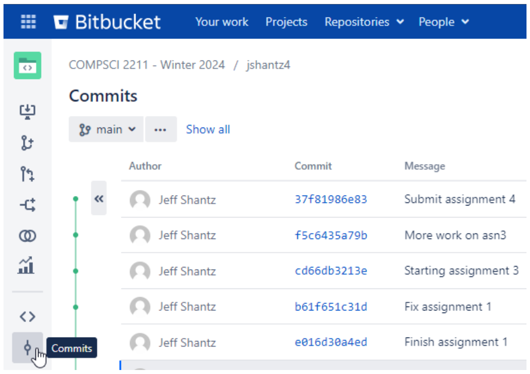
```
[redacted]
├── accommodation.txt        (if you requested academic accommodation)
├── lookup.c
├── testfiles
│   ├── stress-test
│   └── words
├── trie.c
└── trie.h
1 directory, 6 files
```

If not, go back and ensure that your directory matches this structure.

For full details on submission, see the **Assignment Submission Instructions** from the course web site. As a quick overview, submitting will involve the following commands:

```
cd ~/courses/cs2211/asn4
git add .
git commit -m "Submitting assignment 4"
git push
git tag asn4-submission
git push --tags
```

[redacted]



Make sure you see the `asn4-submission` tag associated with your commit. If you do not see this tag on your latest commit, your assignment is pushed to your repo **BUT IT IS NOT SUBMITTED**.