CS 352 (Spring 22): Systems Programming and UNIX

PA 7+8 - Autocomplete

PA 7 due at 7pm, Wed 30 Mar 2022
PA 8 due at **TBD**

This PA has been split into two pieces. Everything will be required in PA 8. However, only some of the functions will be due with PA 7. See the Required Functions below to see which are due, with each PA.

1 Overview

A user types a few letters from a word. We look through a big array of words, and find all of the words that contain the letters that the user typed. We then print out, for the user, all of the matches that we find. It's a fairly simple problem, right? We can just scan through all of the words, call `strncmp()` at every possible position inside each of the words, and report any of the matches that we find.

But what if we wanted to do it **more efficiently?**

The algorithm that we described above is actually moderately costly. We must search **every single** word in our dictionary - worse than that, we have to search at many different locations inside each word! Can we make this faster? Let's consider another algorithm....

SECOND TRY
We start with a tree. Each node in the tree has (up to) 26 children, representing the letters in the English alphabet. As the user types each letter, we traverse one of the links, going down through the tree. When the user stops typing, we are at one of the subtrees; we then iterate through the subtree, and collect a set of words from each node in the subtree. The words that we collect are all of the possible matches!

(I didn't invent this data structure; ask me more about it, after the PA comes due.)

1.1 The Structure of our Tree

You've studied binary trees extensively, in previous classes. And you probably have heard that some trees allow any number of children for each node. But the tree that we're using here is quite odd - it has many children, but a fixed number of children: each node must store 26 pointers.

Storing 26 pointers takes up a lot of space - 208 bytes, in most computers. And so, in the real world, I might consider many strategies to make our tree

more efficient: I might use a flexible-length array of child pointers (and do binary search to find the one I want); I might use a hash table instead of a fixed array; I might consider strategies for combining levels together to save space. However, for this program, we're going to be simple: exactly 26 pointers in every node.

1.2 What the Tree Represents

Each node in the tree represents a string: namely, the string of characters as you follow links from the root to the node. Or viewed another way, each node represents a whole bunch of strings which **begin** with that substring. That is, the node `CAT` represents the words `CAT, CATCH, CATEGORY,` and many more.

But each node also represents more than that. We want to search for any instance of the substring, in a whole bunch of words - and not just the words that *begin* with the word. And so the node `CAT` also represents `BOBCAT, CONCATENATE, SYNDICATE,` and more.

So how, and where, do we list the words that each node represents?

Our solution is to store, with each node, each of the words that **end** with the string. From the example above, the node `CAT` would store the words `CAT, BOBCAT,` and more. But it would **not** store the word `CATCH;` instead, if you followed two more links (C, then H), you would find the node `CATCH,` which would store `CATCH.`

Thus, if the user types `CAT`, then we will go to the node `CAT`; we will print out `CAT`, but also all of the words that we find in all descendant nodes, including `CATCH` and many more.

1.3 Multiple Entries For Each Word

So does this mean that each word is stored only once in the tree? **No!** Instead, each word is stored many times - once for each tail-aligned substring. So the word `CATCH` is stored in 5 different nodes: `CATCH, ATCH, TCH, CH, H.`

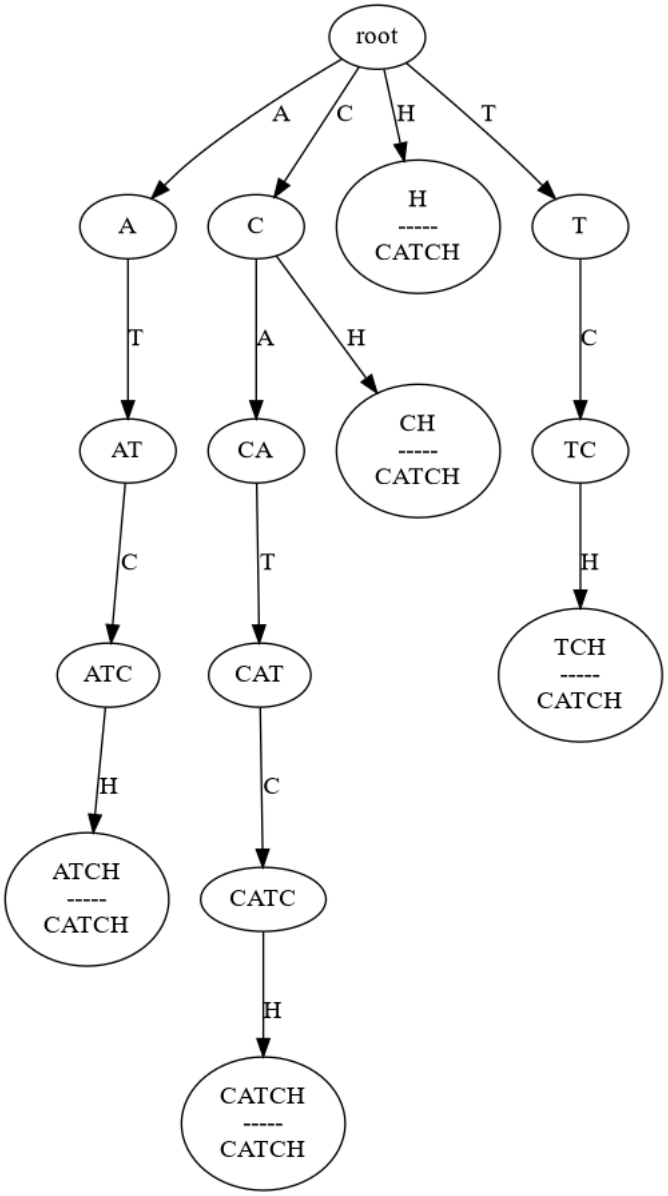
So let's work an example. If the user types `ATC`, how do we find the word `CATCH`? We follow these steps:

- Always start every search at the root
- Follow the links `A, T, C` - which takes us to the node `ATC`
- Print out all of the words (if any) stored there
- Recurse through the entire subtree; one of the descendant nodes is `ATCH`
- `CATCH` is one of the strings stored inside `ATCH`; print that out, too.

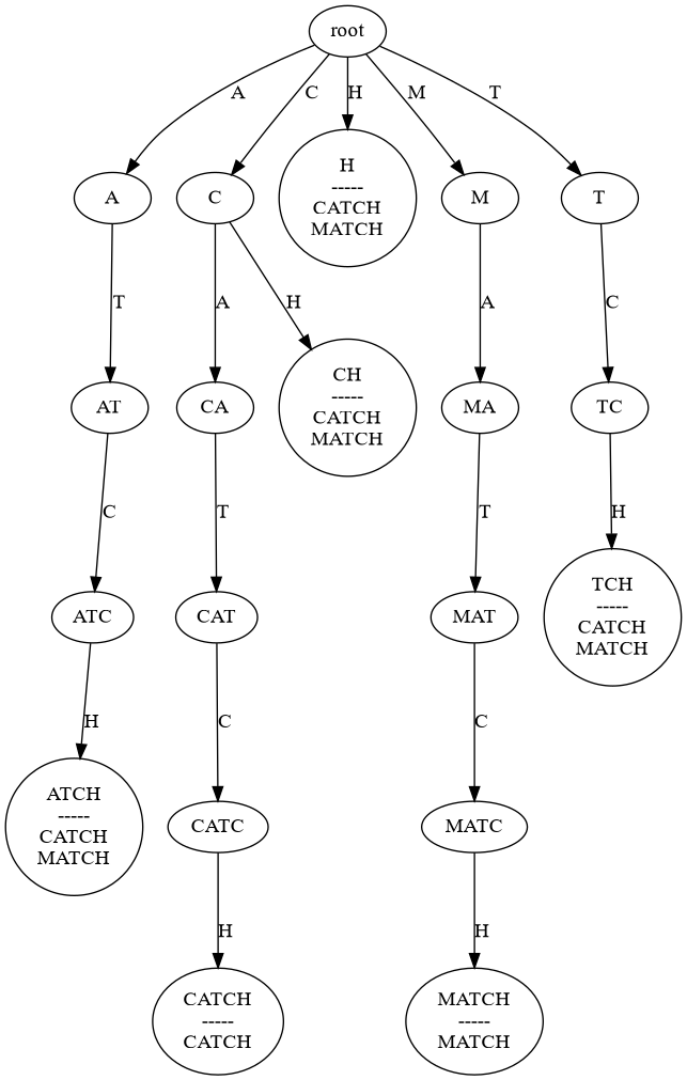
(spec continues on the next page)

1.4 Examples

The following tree is for CATCH. Notice that the result CATCH shows up in five different nodes, representing the search strings CATCH, ATCH, TCH, CH, and H. Also notice that the node C is part of the search path for both CATCH and CH:



This tree adds the word MATCH to the tree. Note that we add one new major chain (starting with the node M), but for the most part, this shares the same nodes that already existed. Also note that many nodes now have multiple result words.



(spec continues on the next page)

2 What You Must Turn In

You will turn in two files, named `autocomplete.c` and `autocomplete.h` . Your C file should include your header; my testcases will also include the header.

Unfortunately, when I set up the autograder, I forgot that Ben likes to put all of the files in sub-directories, inside of a zip file. Instead, for this PA, you will need to upload the files individually.

By mistake, I posted the already-completed version of this header to D2L. So you do not need to edit it, after all. I have provided an complete version of `autocomplete.h` for you. You should make only one change to it: change the `children` field of struct `LookupTreeNode` to be the proper type; see the comment in the header for details. Don't change `autocomplete.h` otherwise.

Your C file must not contain a `main()` function, because each testcase will have its own `main()` function. How, then, will you test your code? Write some testcases of your own!

3 `valgrind`

This is required for both PA 7 and PA 8.

We are requiring `valgrind` for this project! We will run each testcase twice: once without `valgrind` (just checking to see if your code works) and once with it. To pass both testcases, you must get the correct output **and** also not have any memory leaks or errors.

4 Word Lists

There are two parts of our program where "lists of words" will be useful. We don't want to duplicate our code for handling this, so one of your tasks will be to implement a `WordList` struct.

A `WordList` is simply a struct that models a growing array of string pointers; I have provided the following definition in `autocomplete.h`:

```
typedef struct WordList WordList;
struct WordList
{
    int    count;
    int    alloc;
    char **words;
};
```

The `words` field stores a pointer to `malloc()`ed buffer of pointers; you must allocate this buffer, but you should not allocate any buffer to hold the strings themselves (the caller will do that for you). You should treat this field as **public;** my testcase code will read it directly.

The `count` field indicates how many strings there are in the buffer; it must be 0 when you initialize a new `WordList`, and will be incremented every time that we add a new string to the list. You should treat this field as **public;** my testcase code will read it directly.

The `alloc` field is optional, and you can treat it as **private**; you have complete freedom about how to use it, and can even ignore it if you want. See the Appendix of this spec to see how to use it, if you want.

4.1 Required Functions: `WordList` Constructor, Destructor

This is required for both PA 7 and PA 8.

Write a constructor and destructor for the `WordList` type:

```
WordList *wl_create (void)
void      wl_destroy(WordList *list)
```

A `WordList` is made up of **two** `malloc()`ed buffers: one for the struct itself, and one for the array of pointers. These **must** be separate buffers, because you will need to re-allocate the array from time to time, without moving the `WordList` struct itself. The destructor must free **both** of these buffers - but **do not** free the strings themselves; your caller will be responsible for that.

LIMITATION:
In `wl_create()`, you will almost certainly want to pre-allocate an array for the pointers. And many of you will be tempted to pre-allocate a huge array, so that you don't have to write the array-expansion code. To prevent this, your initial array, allocated by `wl_create()`, **must not be any larger than 4 pointers.**

(spec continues on the next page)

4.2 Required Function: `wl_add()`

This is required for both PA 7 and PA 8.

You must write the following function:

```
int wl_add(WordList *list, char *word)
```

This function adds a new word to the list. If the buffer is not large enough to hold another pointer, then you must reallocate the buffer to make more space.

Do not duplicate the string itself; store the pointer that you have been given into the array.

This function must return 0 if it works properly, and nonzero if there is any type of failure.

IMPORTANT PERFORMANCE NOTE:
Most of you will probably, at first, expand the array one step at a time; each time that a new string is added, you reallocate the memory to make it a little larger. **This is permissible, but slow - meaning that some of the testcases will time out.** To pass the largest testcases, you will need to use a more efficient algorithm. It is described in the Appendix at the end of this spec.

(spec continues on the next page)

5 The `LookupTreeNode` Struct

A `LookupTreeNode` represents one of the nodes in the search tree that we described at the beginning of this spec. I have provided part of a declaration for it in `autocomplete.h` . It has only two fields: `children` and `words`; `words` is simply a pointer to a `WordList`.

The `children` field must be an array of exactly 26 `LookupTreeNode` pointers. However, I'm not showing you the syntax for how to declare this; instead, I want you to experiment with it, using the `sizeof()` operator. Confirm that your declaration has the proper size (208 bytes), and that each field inside the array is 8 bytes in size.

Why do I do this? Because even though I've been programming in C for 30 years, I still forget! In fact, I forgot the proper syntax when I was writing my solution for this program! Every time that I need to declare a variable like this, I write a tiny test program that uses `sizeof()` to remind myself how to do this correctly. Get used to it. :)

5.1 Required Functions: `LookupTreeNode` Constructor, Destructor

This is only required for PA 8.

Write a constructor and destructor for the `LookupTreeNode` type:

```
LookupTreeNode *ltn_create (void)
void            ltn_destroy(LookupTreeNode *list)
```

When you build your `LookupTreeNode` object, you **must** create a `WordList` object for it as well. Do not try to build the `WordList` by hand; make sure to call `wl_create()`. Likewise, make sure to use `wl_destroy()` to destroy the `WordList`, when the time comes.

`ltn_destroy()` **must be recursive:** you must also destroy all of your child nodes - and they must destroy their nodes, all the way down through the subtree.

(spec continues on the next page)

5.2 Required Function: `ltn_add_result_word()`

This function is the heart of your tree-building algorithm.

This is only required for PA 8.

Implement the following function:

```
void ltn_add_result_word(LookupTreeNode *ltn, char *search_word, char *result_word)
```

The first parameter is the root of a tree (or subtree); the second is the path that you will follow through the tree, and the third is the word that you should store into the `WordList`, in the destination.

This function may assume that first parameter is certainly not `NULL`, but must build any missing nodes along the path, as necessary. Thus, when this function returns, the tree must contain all of the necessary nodes to get to the destination, and the destination must contain the result word.

(If there is a `malloc()` failure along the way, detect it and return immediately. But if you notice, there is no return value - so there is no way to report this error to your user.)

The search and result words might be the same, but do not have to be; the caller will ensure that the search word is a subset of the result (aligned at the end). Your function does not need to confirm this.

CASE:
You may assume that both the search string, and the result string, are made up **only** of uppercase letters.

HINT:
While it's permissible to write this function using a loop, I found recursion helpful.

(**EXAMPLE** on the next page)

EXAMPLE:
Let's assume that the function is first called on our root node. The `search_word` is `ATCH`, and our `result_word` is `CATCH`. We will follow these steps:

- Starting at the root node, we follow the A link. Let's assume that it exists.
- Looking at node `A`, we follow the T link. Let's assume that it exists as well.
- Looking at node `AT`, we attempt to follow the C link. However, it doesn't exist, so we call `ltn_create()` to build a new node, and store its pointer into the `children` array of node `AT`. We then follow the link.
- Since node `ATC` is brand-new, obviously it has no children; we thus build an H child for it. We then head into that node.
- In node `ATCH`, since we have consumed the entire search string, we add the result word `CATCH` to the `words` field.

5.3 Required Functions: Counting

This is only required for PA 8.

You must implement two functions which traverse a tree, and count either the number of nodes in the tree, or else the total number of result words in the entire tree:

```
int node_count  (LookupTreeNode *root)
int result_count(LookupTreeNode *root)
```

Both of these functions should return 0 if passed an empty tree.

(spec continues on the next page)

6 The Core Algorithms

The functions in this section represent the major steps of the search program. We will have testcases on GradeScope that test most of these individually. Additionally, we will have some testcases that test the entire program at once - which will ony work properly if you have completed **all** of these functions.

6.1 Required Function: `lookup()`

This is only required for PA 8.

The following function traverses a tree from a root node (or subtree) based on a search string, and returns the node that it finds.

```
LookupTreeNode *lookup(LookupTreeNode *root, char *search)
```

For example, if we start at the true root of the tree and our search string is `ATCH`, then this should return the node `ATCH`. However, if we start from the node `CAT` and our search string is `EGOR`, then this should return the node `CATEGOR`.

If the search starts with an empty tree, or if it at any point traverses into a non-existent child, then the function must return `NULL`.

CASE:
The search string will only contain alphabetic characters, but you must handle both upper and lower-case characters. The tree ignores case, so treat everything as upper-case.

(spec continues on the next page)

6.2 Required Functions: Converting Input to a Tree

The following functions are used to convert the input to a tree:

```
WordList       *build_wordlist_from_file(FILE *fp)
LookupTreeNode *build_tree_from_words   (WordList *words)
```

`build_wordlist` is required for both PA 7 and PA 8.
`build_tree` is only required for PA 8.

The `main()` function in the testcase will open up a file (that the user provides as a command-line argument). It will then pass this file to the first function. This function must call `wl_create()` and then populate this list with all of the strings that it finds in the file.

This function should also close the file.

Next, `main()` will pass this `WordList` to the second function; this function must build a tree (as described in the top of this spec). **Remember to use `ltn_add_result_word()` as a helper for this function!**

6.2.1 Handling the Word List

Why do we have two functions? Why a `WordList` in-between them? This is because we need to remember the list of words until the end of the program (so that `main()` can free them).

Remember, we said (at the top of this spec) that each word will show up as a "result" word in many different nodes - and that all of these nodes will share the same pointers. Thus, in order to clean up our memory at the end, we need some way to find (and free) all of these word buffers. This is why the `main()` function needs to know the `WordList` - it will save this until the very end, and free all of the pointers after all searches have completed.

To make this work, you need to ensure that:

- In `build_wordlist_from_file()`, you must `malloc()` a separate buffer for each word that you read, and copy the word into that buffer before you add it to the `WordList`
- `build_tree_from_words()` must not free any of the words - nor may it reallocate or copy them. Simply use the pointers that you find in the `WordList`

6.2.2 Input Format

The input file will be a simple text file, with exactly one word on each line. There may be blank lines (skip over them), but there will be no whitespace in the file, other than newlines.

All of the words will be only made up of alphabetic characters. Since all of your algorithms in this program are **case-insensitive,** convert any lowercase characters to uppercase when you read it from the file.

6.3 Required Function: `print_words()`

This is only required for PA 8.

You must implement the following function:

```
void print_words(LookupTreeNode *result, char *search)
```

This function must print out all of the result words in the entire subtree, rooted at the given node. The `search` parameter does not change what words you print out - instead, it is used only for printing the output.

For each word that you find, print it out using the following format:

```
printf("The string '%s' was found inside the word '%s'\n", search, result_word);
```

You may print these lines out in any order; I will sort the output from your program before checking it against the testcase, so order doesn't matter. (That's why we included the search string in the output - so we can figure out which output lines came from which searches.)

7 Turning in Your Solution

You must turn in your code using GradeScope.

8 Appendix - Efficiently Appending to an Array

Arrays in C are **never** resizable - at least, not automatically. You have to resize them by hand. In the case of a `malloc()` array, we can do this by calling `realloc()` to expand the buffer (or move it to a new location in memor, if necessary).

When we are handling appending into an array, it's tempting reallocate every time that a new value is added. After all, it's easy - we already know the length of the array, and so we just increment that size, and reallocate the memory. But remember, `realloc()` potentially **copies the entire array** from one location in memory to another. So the first time that we resize the array, it moves 1 element; then 2; then 3; then 4; etc.

The total work done, resizing the array, is thus $O(n^2)$:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Instead, let's expand the array rarely. What if, instead of doing +1 each time, we always expanded the array by **doubling** its size? If we add up the total cost of all of these expansions over time, it is far better! One way to look at it is to realize that we pay a cost of 1, then 2, then 4:

$$1 + 2 + 4 + 8 + ... + n \approx 2n = O(n)$$

Another way is to think **backwards**; our last copy was (roughly) a cost of n, and the one before that was half the cost, and so on, back to the beginning:

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... = \ n(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...) = \ 2n = O(n)$$

Thus, we see that **all of the work done to expand the array** totals up to $O(n)$; thus, the average cost (per single append is $O(1)$).

8.1 The Algorithm in Practice

In practice, this means that we need **two** different integers to keep track of the array size: one that tracks the current length, and another that tracks the **allocated** length. Most of the time, when a user wants to append a new value into the array, we discover that there is some free space, and so we can write into the array, with no need for `realloc()`.

But once in a while, a user will try to append into an array, but the array is completely full. In that case, we will **double its size** - and then there will be plenty of space to add the new value.