Problem Set 6

Part 1: Pointers to pointers. Multidimensional arrays. Stacks and queues.

Problem 6.1

In this problem, we will implement a simple "four-function" calculator using stacks and queues. This calculator takes as input a space-delimited infix expression (e.g. **3 + 4 * 7**), which you will convert to postfix notation and evaluate. There are four (binary) operators your calculator must handle: addition (+), substraction (-), multiplication (*), and division (/). In addition, your calculator must handle the unary negation operator (also -). The usual order of operations is in effect:

- the unary negation operator **-** has higher precedence than the binary operators, and evaluated right-to-left (*right-associative*)
- ***** and **/** have higher precedence than **+** and **-**
- all binary operators are evaluated left-to-right(*left-associative*)

To start, we will not consider parentheses in our expressions. The code is started for you in **prob1.c**, which is available for download from Stellar. Read over the file, paying special attention to the data structures used for tokens, the stack, and queue, as well as the functions you will complete.

(a) We have provided code to translate the string to a queue of tokens, arranged in infix (natural) order. You must:

- - fill in the `infix_to_postfix()` function to construct a queue of tokens arranged in postfix order(the infix queue should be empty when you're done)
  - complete the `evaluate_postfix()` function to evaluate the expression stored in the postfix queue and return the answer

  You may assume the input is a valid infix expression, but it is good practice to make your code robust by handling possible errors(e.g. not enough tokens). Turn in a printout of your code, along with a printout showing the output from your program for a few test cases (infix expressions of your choosing) to demonstrate it works properly.

(b) Now, an infix calculator really is not complete if parentheses are not allowed. So, in this part, update the function `infix_to_postfix()` to handle parentheses as we discussed in class. Note: your postfix queue should contain no parentheses tokens. Turn in a printout of your code, along with a printout showing the output from your program for a few test cases utilizing parentheses.

Answer:

The program can be found in **prob1.c**, and the test results are shown in two PNG files. The problem is known as **shunting yard algorithm**, which is very interesting. My program has the memory leakage program although it implements the basic function as required. I have used **valgrind** to check the memeory leakage, and the program still needs improving.

There are some attentions when you input the equation. When you use `-` as SUBSTRACT operator, do not omit the space between `-` and number, like `-3`, because the program will treat `-3` as NEGATE 3, not minus 3, which will cause unexpected results.

Problem 6.2

A useful data structure for storing lots of strings is the "trie". This tree structure has the special property that a node's key is a prefix of the keys of its children. For instance, if we associate a node with the string "a", that node may have a child node with the key "an", which in turn may have a child node "any". When many strings share a common prefix, this structure is a very inexpensive way to store them. Another consequence of this storage method is that the trie supports very fast searching - the complexity of finding a string with *m* characters is O(*m*).

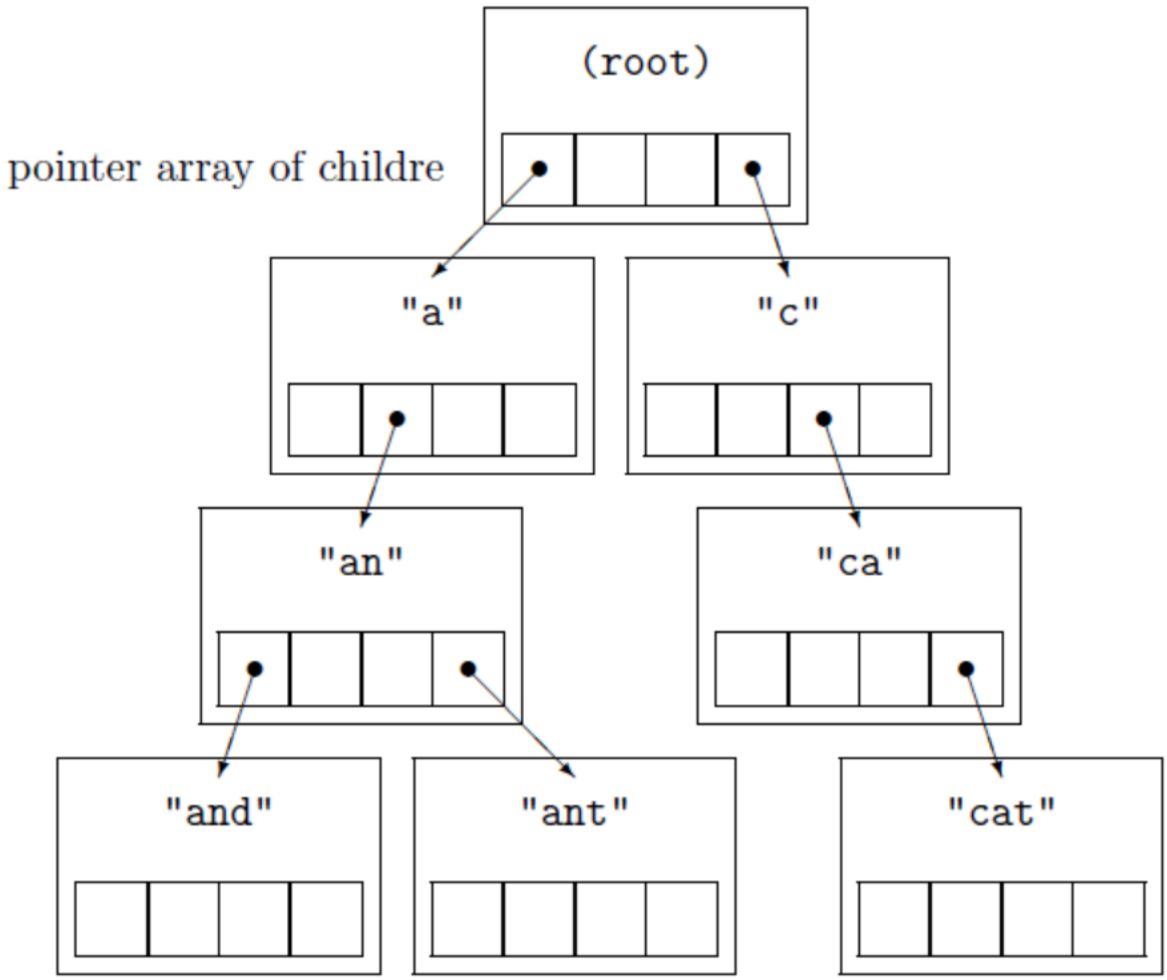

Figure 6.2-1: Trie structure (translations not shown). For each node, its key (e.g. "and") is not explicitly stored in the node; indeed, the key is defined by the node's position in the tree; the key of its parent node + its index in that parent's pointer array of children.

In this problem, you will utilize a trie structure and to implement a simple one-way English-to-French dictionary. The trie structure, shown in Figure 6.2-1, consists of nodes, each of which contains a string for storing translations for the word specified at that node and an array of pointers to child nodes. Each node, by virtue of its position in the trie, is associated with a string; in the dictionary context, this string is the word (or part of a word) in the dictionary. The dictionary may contain multiple translations for the same word; in this case, words should be separated by commas. For example: the word like, which has two meanings, could translate as *comme*, a preposition, or as *aimer*, a verb. Thus, the translation string should be "comme,

aimer". To get you started, we've provided code in **prob2.c**, which can be downloaded from Stellar. You will need to:

- fill in the helper functions `new_node()`, `delete_node()`
- complete the function `add_word()`, which adds a word to the trie
- complete the function `lookup_word()`. which searches the trie for a word and returns its translation(s)

Once your code is working, run a few test cases. Hand in a copy of your code, and a printout of your program running in gdb, with a few example translations to demonstrate functionality.

Answer:

The code can be found in **prob2.c** and the funcationality has been tested.

I have to say I took a look at the answer for completing the function `add_word()`. I was like, totally confused about how to add words to the trie structure in this function without taking the trie structure as parameter until the answer inspired me to find out that the root of trie tree had been defined as **static** variable, which meant I could use the trie tree in this function directly.

Another thing I want to mention when I dealt with this problem is one statement in the function `load_dictionary()`:

```
*translation++ = '\0';
```

So when I used gdb to debug the code and try to figure out what happened inside the function `load_dictionary()`, I got the sense of what the purpose of the function. It read every sentence in the `dict.txt` file and split every sentence into two words apart from the sentences starting with `#`. But I couldn't make sense of what's the functionality of the statement and gdb just showed that the `word` and `translation` variables turned out to be what we wanted.

In fact, I had to keep an eye on the previous code to understand why.

```
word = line + strspn(line, DELIMS);
translation = word + strcspn(word, DELIMS);
```

The `strspn` function used in the first line is to get the length of the starting delimiter in the `line` variable, so `word` points to the position of the first character not belonging to `DELIMS` in `line`.

The `strspn` function used in the second line is to get the index of the first character in `word` matching with one character in `DELIMS`. `DELIMS` is defined as `\t` in this code, so `word` now points to the position when `\t` first occurs, which than assigned to `translation`.

Now we finally come to the critical statement at the beginning and have a glimpse of what the code tries to do. In order to make the code easier to understand, we can break the code into two lines:

```
*translation = '\0';
translation = translation + 1;
```

So the code replaces `\t` with `\0` and increments the `translation` by one. We all know string is made up of characters ended with `'\0'`, now `word` points to the first part of the original sentence and `translation` points to second part of the original sentence, which completes the process of reading the file to generate the trie tree dictionary.