Programming Assignment 1: Dictstat

Edmond Wu, CS211, Section 02

October 11, 2014

Data Structures:
The main data structure I used to complete this assignment was a variant of the trie structure, implemented using an array of 26 references with each reference corresponding to each (lower-case) letter in the English alphabet (a - z). Starting with a root node with no data, it takes each word from the dictionary file, and inserts the word into the trie, one letter at a time.

Summary of Functions and Structs:

1. **struct node:** A struct representing nodes in the trie.
2. **makeRoot():** Creates the root node of the trie. This root node should not contain anything.
3. **makeNode():** Creates a node; used when inserting words into the trie.
4. **stringLower():** Takes a word (a string) and converts it into lower-case, returning the converted string.
5. **insertWord():** Takes a word and inserts it into the trie.
6. **incrementPrefixes():** Increases prefix count.
7. **addWord():** Takes a word and looks for it in the trie, increasing its prefix count (using incrementPrefixes()), occurrence count, and superword count appropriately.
8. **printTrie():** Prints out all words in the trie along with their prefix, occurrence, and superword counts in alphabetic order.
9. **destroyTrie():** Frees up the memory used creating the trie.
10. **readDict():** Reads a dictionary file and builds a dictionary (trie) from it.
11. **scanData():** Reads a data file and attempts to match it with words in the dictionary.

Challenges Faced:
The biggest challenges I faced during this assignment mainly had to do with my lack of experience with a programming language like C. Prior to C, I was familiar with high-level languages like Java and Python which automatically took care of issues like memory allocation/de-allocation. I was also unfamiliar with the concept of pointers and it took me a long time to grasp the idea of using pointers and de-referencing them. Memory allocation was not that difficult, but figuring out how to free all the used memory at the conclusion of my program also posed its challenges.

Complexity Analysis:
The primary functions to be concerned about are insertWord(), incrementPrefixes(), addWord(), and

destroy/printTrie(). With m total characters and n unique words, the run time of insertWord() just depends on the length of the word inserted, and since each word has at most 100 characters, insertWord() has a run time of $O(100)$ or $O(1)$. The function addWord() has the same run time initially, although at the end of it it calls incrementPrefixes(), a recursive method with run time of $O(n)$ as there are only n words in the trie. The destroy/printTrie() methods traverse the tree in essentially the same fashion (an in-order traversal), so the run times of those two functions are also $O(n)$. The main method calls readDict() and scanData(), with each of those functions taking $O(m)$ and $O(m \times n)$, respectively. After those two methods it calls printTrie() and destroyTrie(), so the total run time will be $O(m + mn + n + n) = O(2n + m + mn) - > O(m + n + mn)$. Each node in the trie created has a size of a node along with the size of the character array (26 * the size of a node). The worst-case would be if every node in the array had its character array filled up, so the space required will be that amount times the size of a node.

Context Registers

readDict()

```
rax 0x603010 6303760
rbx 0x0 0
rcx 0x7ffff7b00160 140737348895072
rdx 0x0 0
rsi 0x7ffff7b926ba 140737349494458
rdi 0x603010 6303760
rbp 0x7fffffffde30 0x7fffffffde30
rsp 0x7fffffffdd80 0x7fffffffdd80
r8 0x4015b9 4199865
r9 0x0 0
r10 0x1 1
r11 0x246 582
r12 0x400950 4196688
r13 0x7fffffffdf40 140737488346944
r14 0x0 0
r15 0x0 0
rip 0x400f7a 0x400f7a ¡readDict+19¿
eflags 0x202 [ IF ]
cs 0x33 51
ss 0x2b 43
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0x0 0
```

scanData()

```
rax 0x606080 6316160
rbx 0x0 0
rcx 0x7ffff7b00160 140737348895072
rdx 0x0 0
rsi 0x7ffff7b926ba 140737349494458
rdi 0x606080 6316160
rbp 0x7fffffffde30 0x7fffffffde30
rsp 0x7fffffffdd80 0x7fffffffdd80
```

```
r8 0x4015b9 4199865
r9 0x0 0
r10 0x1 1
r11 0x246 582
r12 0x400950 4196688
r13 0x7fffffffdf40 140737488346944
r14 0x0 0
r15 0x0 0
rip 0x401193 0x401193 ¡scanData+19¿
eflags 0x202 [ IF ]
cs 0x33 51
ss 0x2b 43
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0x0 0
```