Nathan Miniovich
RUID: 151004690
CS211 Project 1: Dictstat

Readme

Trie Data Structure:

The Trie data structure is made up of struct trieNode*'s that have the parameters

char data;
The relevant character stored in the array from the dictionary file.

int count;
The number of times a character occurs in the dictionary file.

int isWord;
This is used as a boolean as well as a way to index into the proper spot in the word array (not the same as the next array of the struct). When build node is called this value is initialized to -1 to show false, and anything greater than -1 is true. When a character needs to be marked as the end of a word its value is changed to 0. In the trieDFS method where the word array is being built (the word array is used to print the word part of the result in alphabetical order) the isWord is changed to the index of that word in the word array.

int level;
This shows the level of the data structure where the current array sits. The root starts at -1 and the level goes up by one with each next array, so the first actual character in a chain would be stored at level zero. This is also used to update the correct index of the file buffer with a new character when dumping the file buffer into the word array. The maximum the level can be is 99 because the maximum length a word can be in this program is 100.

struct trieNode_* next[26]
Initialized to size 26 in order to be able to store all alphabetical characters a-z. A character subtraction is used to calculate the index of insertion in the array in order to give an insertion time of O(1). Each node has its own sub array of 26.

struct trieNode_* parent;
Makes reference to the parent of a node in the array. Not used in this program

The data value of the root is set to '$' so that it can be conditioned on. This is acceptable because a non-alphabetical character will never be added to the array. The isWord value of the root is set to -1 as well as the count since the count will never be incremented. Anything in the subarray of a node containing a valid alphabetical character is the next letter in a chain which will end in a word, with the last character in a word (but not necessarily the whole chain) having its isWord value sent to 0. For example you can have the chain c-a-t-e-r-s with "cat" being marked as a word (t->isWord = 0) and "caters" being marked as a word (s->isWord = 0). All characters are set to lowercase before insertion into the Trie. The tree can be searched and printed out using a modified DFS.

Efficiency:

The run and space time complexities of the program are O(m) and O(n) respectively.

Run time O(m):

When reading the files every character is read once and dumped into a file buffer. O(m)

The dictionary buffer is processed character by character in order to build nodes for the trie, and insertion into the Trie next array is O(1) as illustrated in the description of the Trie data structure. O(m)

To traverse the Trie the program goes along the data file character by character and advanced a position in the Trie if it reads an alphabetical character. If the character is non-alphabetical it goes back to/stays at the root. O(m)

Occurrence and superword are simple if checks which increment the count if true. Prefix needs to check everything under the point where the temp is to see if any words reside there, so the temp is passed to a helper method as the root of the check. Everything under the "root" in the prefix method is only checked once to see if it has an isWord value greater than -1. O(m)

The above simplifies to **O(m)**

Space time O(n):

The length of a word can at most be 100 characters, so the limit of the size the Trie can take up is finite. All data structures like count arrays and file buffers are initialized based on the size of the input, and will never expand beyond that. As such, the space time complexity is directly proportional to the size of the input. Therefore the space time complexity is linear or **O(n)**

Challenges:

The biggest challenge was learning the small things C does differently/complains about. This was made harder by a debugger that doesn't really explain what is going on.

There was a specific case where I was trying to read the isWord value of a trieNode and the program was seg faulting because the value was -1

Figuring out how to allocate memory properly was also a challenge. It was much nicer when I started using calloc instead of malloc

Context Registers:

readDict register:

rax	0x0	0
rbx	0x0	0
rcx	0x66	0
rdx	0x0	0
rsi	0x0	0
rdi	0x602508	6300936
rbp	0x7fffffffe380	0x7fffffffe380
rsp	0x7fffffffe350	0x7fffffffe350

```
r8             0x78     120
r9             0x101010101010101     72340172838076673
r10            0x76     118
r11            0x7ffff7acca92  140737348684434
r12            0x400710        4196112
r13            0x7fffffffe490  140737488348304
r14            0x0      0
r15            0x0      0
rip            0x400973        0x400973        <readDict+121>
eflags         0x10206         [ PF IF RF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

scanData register:

```
rax            0x4      4
rbx            0x603140        6304064
rcx            0x604fb0        6311856
rdx            0x603230        6304304
rsi            0x604fb0        6311856
rdi            0x6035f0        6305264
rbp            0x7fffffffd890  0x7fffffffd890
rsp            0x7fffffffd860  0x7fffffffd860
r8             0x7ffff7fbd700  140737353864960
r9             0x0      0
r10            0x7fffffffdf60  140737488346976
r11            0x7ffff7a74e50  140737348324944
r12            0x400710        4196112
r13            0x7fffffffe490  140737488348304
r14            0x0      0
r15            0x0      0
rip            0x401242        0x401242        <prefixBot+79>
eflags         0x10202         [ IF RF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```