CO322 LAB 04

TREE ADT

PREMATHILAKA M.P.U
E/15/280

INTRODUCTION

There two implementations are included as part 1 and part 2. For the part 1 (trie.h,run.c) tree implemented as a trie data structure.

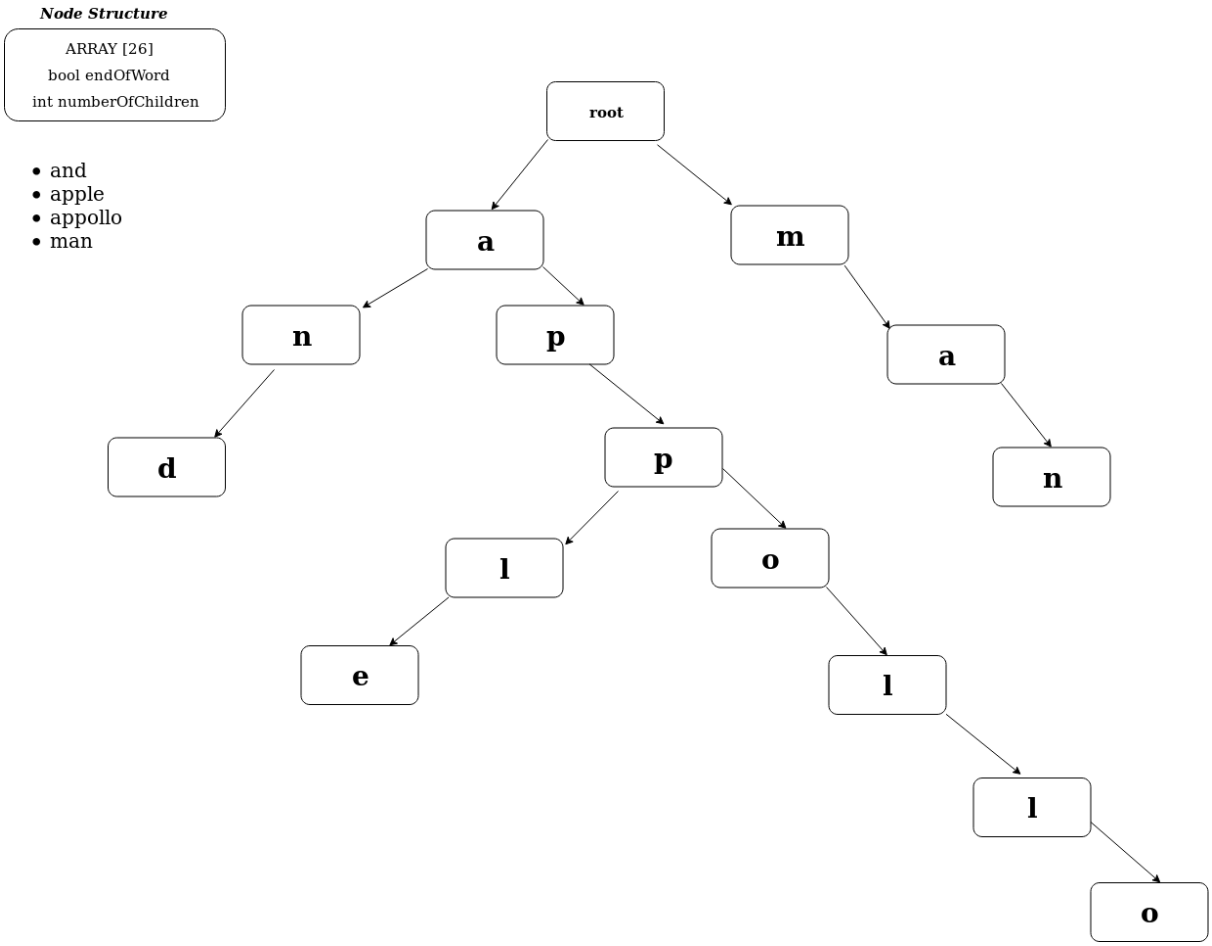

FIGURE 1: TRIE DATA STRUCTURE

As shown above each node consist of ARRAY[26] and each block of the array represents a letter in the alphabet. In a given node, if an array slot is not NULL, that letter contributes to create a word in the dictionary.

The next approach (part 2) is proposed to perform better than the part 1. In this ternary tree data structure is used to reduce the memory consumption as in the trie data structure.

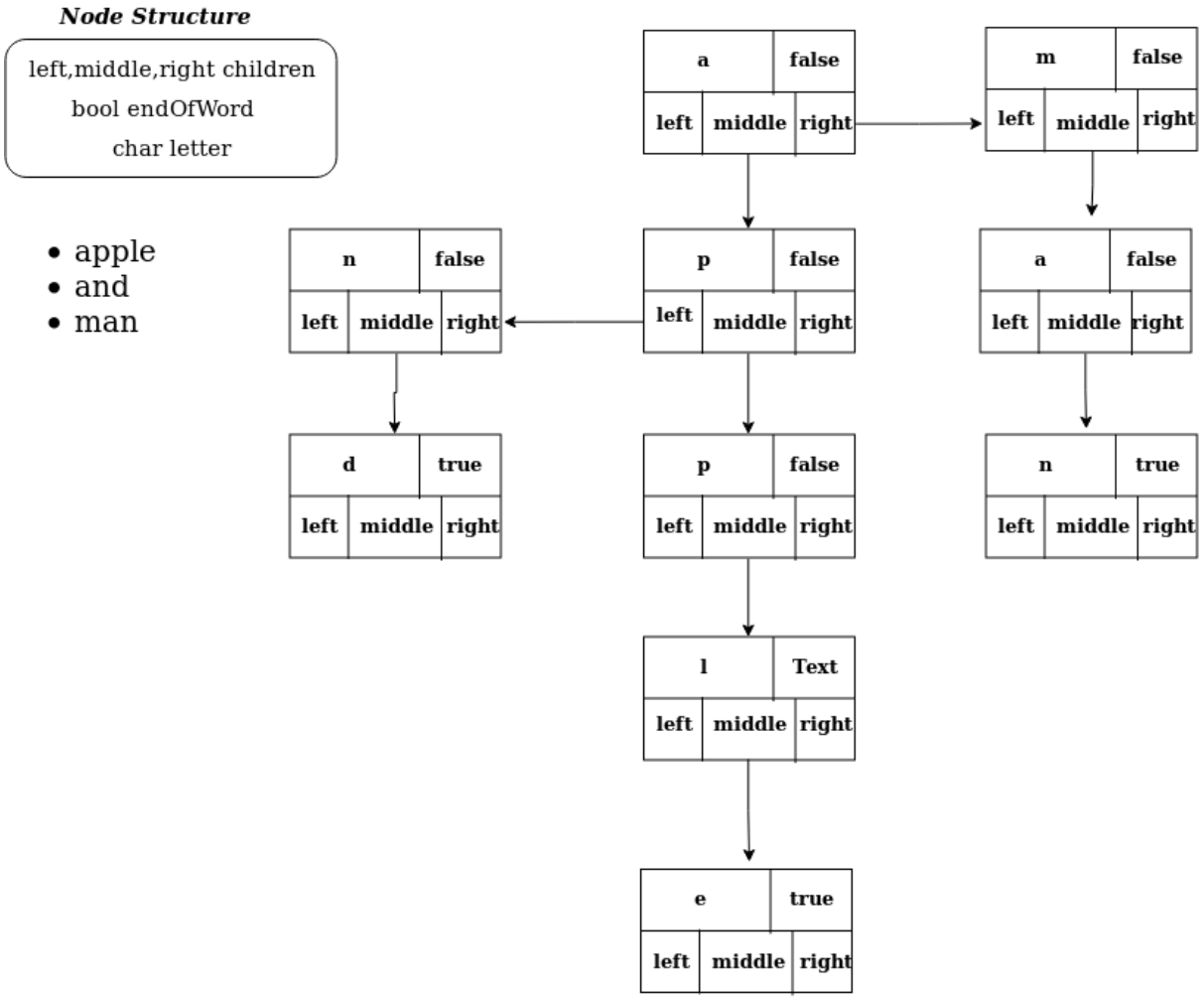

FIGURE 2: TERNARY TREE DATA STRUCTURE

Depending on the data structures and algorithms used, the performance of the task might vary. In order to compare the performance between these two approaches following criteria are proposed to measure.

1. Time taken to store a given dictionary.
2. Memory space usage.
   Here, memory consumed by the data structure to store words in the dictionary is measured
   In order to calculate memory consumption by programs, Linux: ***free*** command is used.
3. Time taken to print a list of suggestions for chosen word prefixes.

RESULT

The program which uses trie data structure in the part 1 executes without any crashing up to 550 words in average
Each word count is selected as the first most words in given files (*wordlist1000.txt, wordlist10000.txt, wordlist70000.txt*) and average time to insert words, average memory usage, average time to predict words are tabulated.

1. Time taken to store a given dictionary

Time

Word Count	Trie d.s (Part 1) average memory usage in 3 test runs	Ternary tree d.s (Part 2) memory usage in 3 test runs
50	0.075ms	0.064ms
100	0.110ms	0.084ms
500	0.403ms	0.243ms
1000	Program crashed	0.607ms
10000	Program crashed	6.795ms
69903	Program crashed	61.744ms

TABLE 1

2. Memory space usage

Memory

Word Count	Trie d.s (Part 1) average memory usage in 3 test runs	Ternary tree d.s (Part 2) memory usage in 3 test runs
50	65KB	86KB
100	89KB	104KB
500	452KB	0KB

1000	Program crashed	110KB
10000	Program crashed	632KB
69903	Program crashed	10000KB

TABLE 2

3. Time taken to print a list of suggestions for chosen word prefixes.

- In order to make the program complex, single words used as prefixes.
- Since Trie d.s (part 1) runs only up to 500 word count, first 500 words are used from given text files.

Time

Prefix	Trie d.s (Part 1) average memory usage in 3 test runs	Ternary tree d.s (Part 2) memory usage in 3 test runs
'a'	0.203ms	0.183ms
'b'	0.150ms	0.122ms
'd'	0.168ms	0.101ms
'f'	0.133ms	0.134ms
'h'	0.128ms	0.087ms
'k'	0.036ms	0.029ms

TABLE 3

DISCUSSION

- As shown in the results, memory used by the trie data structure to store a dictionary of words is greater than the memory used by the ternary data structure.
The reason behind this difference is, in trie data structure a node consists of ARRAY which has 26 slots. In this array one slot is used to a letter of a word and all the other 25 slots are NULL by definition. If the number of words in the dictionary are low, number of unused null slots are huge. Though the inserted number of words are high, if most of the word prefixes are the same, there will be no distribution of words in the slots. Therefore, using this type of an array in a node leads to a huge memory waste.

But in Ternary tree, instead of 26 pointers, only 3 pointers are stored as left, middle and right as shown in figure 2. Therefore, per node memory consumption low compared to the trie structure. Thus, after inserting all the words, total memory consumption in the ternary tree is low.

Most importantly, it can be seen that the trie data structure does not support for huge number of words (approximately 500 words maximum). In trie structure a node consist an array length of 26. When the recursive word inserting algorithm runs, in order to insert a letter to a node, it has to access the relevant slot of the array. This process consumes lot of processing. Since there are excess recursive calls which are not executed stack overflow leads to crash the program. Therefore, trie structure can store approximately 500 words without any crashing.

- When considering time taken to store a dictionary of words, as shown in the table 1 obviously time to store words increases with the size of the dictionary.
But when comparing the two implementations, we can see on average trie structure consumes a higher amount of time than the ternary structure. The reason for this is, in ternary node it has a character variable and 3 pointers pointing to the next node. When storing a word, letter by letter is stored in the character variable and the next letter is pointed by the relevant pointer. But in a trie node, when adding a letter to the node, it has to access through the array to find the relevant place of the letter. This process is time consuming. When storing a word, for each letter the trie structure has to do this process. Therefore, the total time to insert a word to the data structure is higher in the trie data structure.
- When comparing time taken to suggest words, ternary tree consumes less time than the trie data structure (tested for 500 words). As explained on the previous point, when the program accessing the relevant index of the array in trie structure, it consumes a time. In

the ternary tree the data (letter) can be accessed directly. Therefore, in ternary tree perform better than the trie data structure. Also, traverse function in the trie structure uses a function called “join” to combine a letter with a word. This function uses heap memory to combine in each traverse function call and each of the time a for loop runs to generate the word. This leads to a delay in the program. But in ternary structure algorithm, prefix and postfix when they are about to print. Number of “returnWord” function calls are equal to the number of suggested words. Therefore, there will be no considerable delay.

Even the ternary structure terminates if the number of suggestions for a given prefix is high. When there are excess recursive calls in the stack for “traverse function” stack overflow occurs.