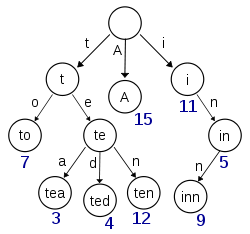
**Trie Report**

 Computer science is an evolving industry that grows rapidly. As computers grow more powerful from year to year there are bigger and bigger problems that need to be solved. In early computer science studies one learns about the different structures that can be used in order to store data. Balanced tree structures and Hash Tables allow for storage and quick searching. The hash table has the advantage of allowing for a method of constant searching within the data structure. The large disadvantage is that hash tables need to take time to resize the table to either grow or shrink the array that is used. This resize method takes O(n) time when the time comes to resize. With large enough data, the rehash could take a fair amount of time, so we look to solve this problem.

In having a discussion with an old friend of mine, we designed a data structure that is called a Trie data structure. The trie works as follows and is best understood when storing words (computer ‘strings’) in the structure. Each node represents one letter of a word. The structure starts at the root and moves from one node to the next based on the corresponding node. In the example to the right, one can see all leaf nodes represent a word (although not all words are leaf nodes) and they each have a particular path in order to reach that destination. The algorithm for such a tree structure is rather simple and it is:

Trie Find:

Start Node n at Root

For each letter c in Word:

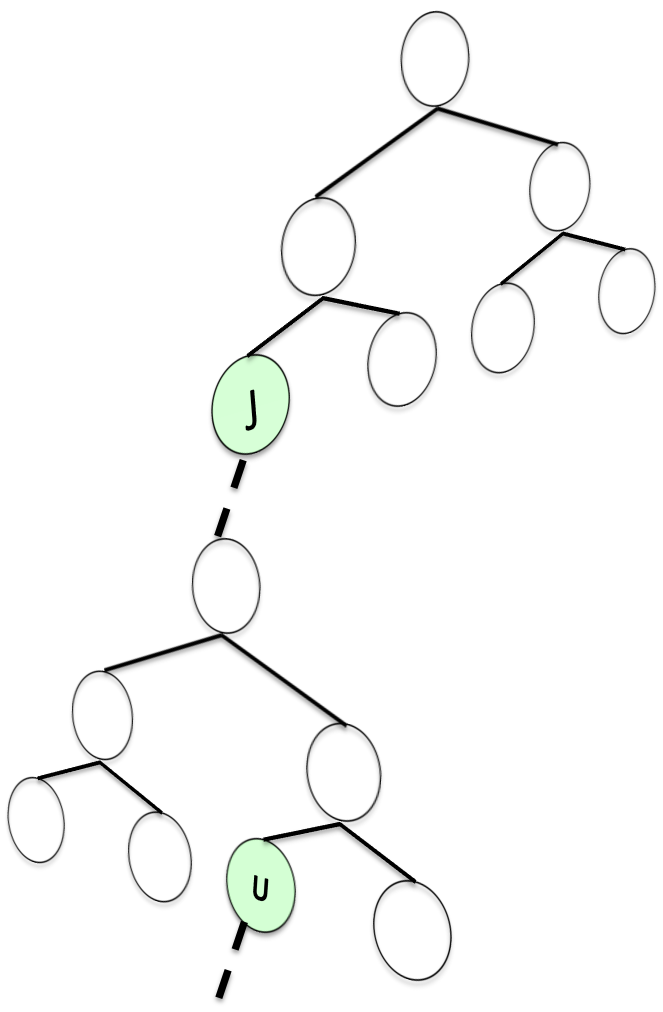
n = n.findNextNode(c)

return true if n is a valid word Node

return false otherwise

\*At any moment, if findNextNode() can’t find the next node to go to, it is immediately implied that Word is not a valid word.

I have chosen to call this particular data structure a Dictionary as it seemed to be fitting for a data collection of words. From the beginning I could see that the above algorithm comes down to O(word size + time to find next node). In implementing this data structure, I did some experimentation on how can we find the next node when keeping two things in mind: memory usage and minimal runtime. I originally used two designs to find the next node; a TreeMap and a HashMap. In the next sections I will address the efficiencies of both designs, the weaknesses of both designs, and my solutions to the flaws that I found with them.

**TreeMap Design**

The dictionary is a set of levels. Every level represents one character of the word we are looking for. In implementing the Map design, I had to map a character to a Node. The picture on the right diagrams how the design looks like. I think of the structure kind of like spreading out into a multidimensional Tree. Every node can simply split of to another map. To make drawing easier, I only split off from the leaf nodes, but any node can point to another tree. Each one of these trees represents just a single node in the picture on page one, transitioning from one letter to another. The case on the right, I am starting to spell the word “jump”. The first tree represents nodes for the first letter. As I branch off from the J node, I enter a tree that reprents “J\*,” all words that start with J and I’m filling in the second letter. Once I branch off from the U node, I’m entering a tree that represents all “JU\*” words, etc. The strengths of this design is flexibility with all characters and memory usage. Words are only created with just enough space to store the word. The downfall of the algorithm, is that searching is a lot slower than I expected.

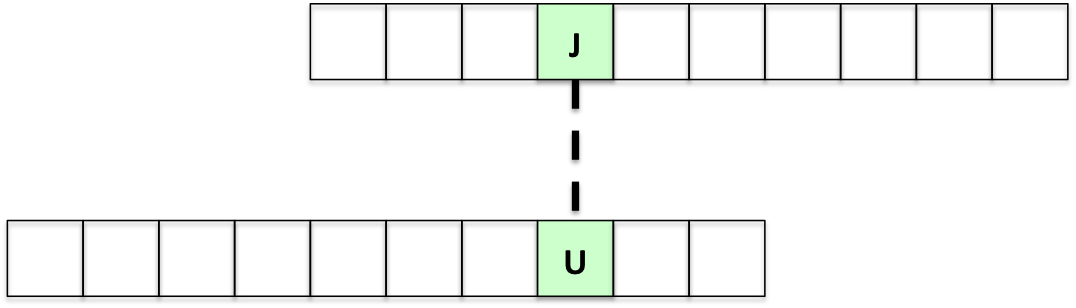
JU Tree

J Tree

Although every tree is flexible with all ascii characters, I only ran tests using our alphabet (26 characters). This means that a tree can be at most log2(26) ≈5. So the search from one character to the next, takes at most 5 comparisons or clicks in order to find the next destination. In the worst case scenario, it would take 5 \* (Word length) comparison just to find one word. Finding a common 5 length word could take 25 checks in order to figure out if the destination is valid or not. On a large scale, 25 checks may not seem like a lot. Especially considering that this search is consistently 25 checks at most no matter how big our data structure grows to be.

Although the tree’s weakness is costly, there is one feature that we consider to make its weakness a strength. Searching per level is only a cost of 5 where the dictionary is very dense. If there are a lot of letters that begin with J, then it is costly to find where the JU words are. But, as we branch further and further out, the trees become sparser. For example, there are fewer words that begin with “JUM.” One we reach the JUM tree, that tree could possibly have just the one “P” node we are looking for to complete the word “JUMP.” So as we get further and further out, our searching speed approaches instantaneous findings while traveling from one level to the next.

**Hashing**

 The previous page was a design completely to solve one part of the algorithm, just finding out how to move from one level to the next. Seeing as how the previous design could be slow under certain circumstances, we needed to design a faster method of moving from one level to the next.

Consider an array of size 26, an array that can store one letter in each entry. Arrays allow for instantaneous access, modification, and use. The first entry in the array is reserved for the letter A, the second for B, and so forth. The advantage with this design is that indexing is done through the calculation of:

int index = nextCharater – ‘a’

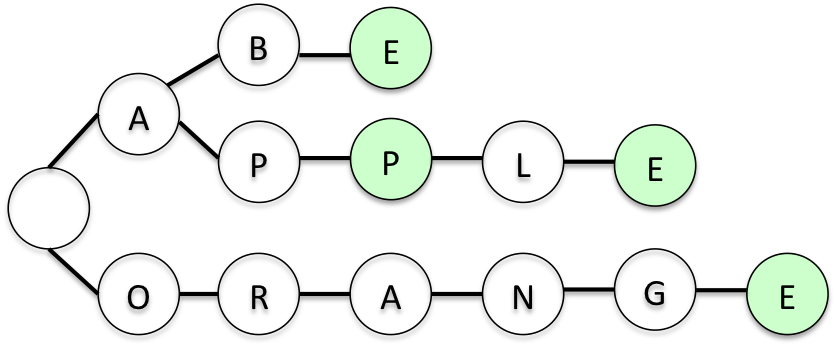
This takes advantage of ascii values in order to do quick calculations for indexing within the array. The advantage of this design is that searching for the next node couldn’t be made any faster as the next destination is calculated by a simple subtraction problem. The great disadvantage with this design is that it consumes a lot of memory. In the example above, if we only store the single word “JU,” it would require 52 places in memory in order to store just a single word. The other disadvantage is that this design assumes that only alphabetical characters can be stored. From how ascii codes are set up, there’s no easy way for this design to implement punctuation to be added as a word such as contractions; don’t, can’t, etc. A simple solution can be added to accommodate for the apostrophe by having a special case check for the apostrophe. This just leads to the problem; what if I want to store a string that contains a comma or a period? There would be too many special cases that we start losing the effectiveness of a single calculation to find the next node.

In the end, this design is effective in dense locations, but uses a lot of memory where values are sparse.

Solutions:

The hashing solution can be improved by using a standard library Hash Table that uses the proper amount of memory and allows for any character to be stored in the has table (punctuation and all). As this solution seemed to simple, I implemented another design to make matters more interesting. I created my “Mutant” tree that uses both designs to its advantage.

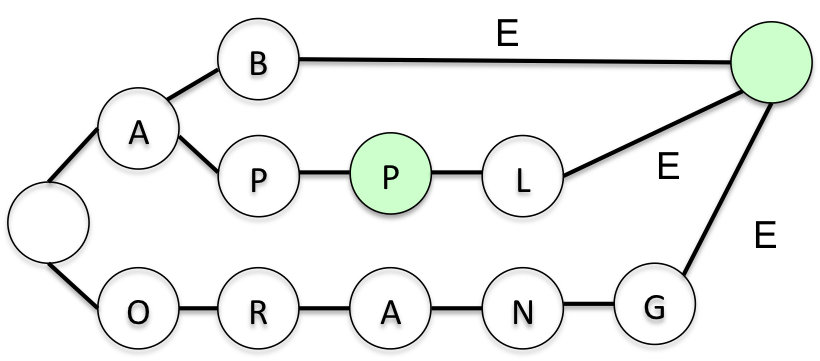
**Mutant**

The two previous reports came to the conclusion that a Tree is effective in sparse sections and an Array is effective in dense portions of the trie. This design gives the idea that a node can grow and involve from one type of node to another. The three types of Nodes involved are: Leaf Nodes (nodes with no neighbors), Tree Nodes (nodes with a few neighbors), and Hash Nodes (nodes with many neighbors). Consider the trie data structure shown here. Although this example is small, it represents just a portion of what could be a completely filled dictionary. The initial root node would be a Hash Node since it has “many” neighbors (A and O), the A node would be a Hash Node, but all other Nodes would be a Tree Node or a Leaf Node. Leaf nodes represent those nodes that have zero neighbors, Tree Nodes contain anywhere from 1 – 8 neighbors, and Hash Nodes handle nodes with 9 – 26 neighbors. Any time a node crosses from one range to the other, it “evolves” to the higher node.

This design gave incredible results that were near maximal results, using minimal memory and obtaining optimal speeds.

**Chaining/Labyrinth Design**

At this point I achieved the results I was looking for, but decided to make further designs that I have not yet implemented. The trie can reduce memory use if turned into a graph with a start and end node. Consider the diagram at the top of this page. Every word has a final E in common in order to be considered a word. The idea of what I call “chaining” is to take common suffixes and link them together to get a resulting graph of:



This creates a form of compression in order to reuse already known data and it particularly creates just a single leaf node instead of a leaf node for every single word in the dictionary. Words that have the same suffix, such as \*tion, will use the same ending path in order to read the end/accept node.

The flaw with this design is that if we have the following data:

Option, carnation, summation, nation

|  |  |  |  |
| --- | --- | --- | --- |
| op- | | | -tion |
| car- | n | -a |
|  |
| summ- | |

If we want to add the word “national” to the trie, we would accidentally add words that were not intended to exist. It would add the words “optional,” “carnational,” and “summational” since all of these words share the same suffix. So, we have to either figure out a clever algorithm, or we introduce the “solid state” idea. Once compressed, we can no longer add any more words to our dictionary.