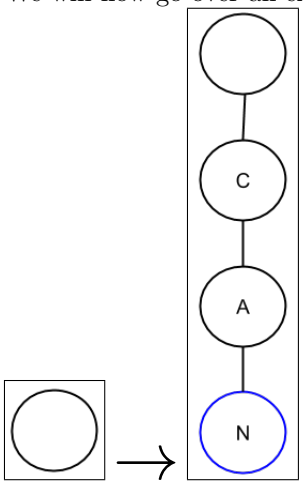
Tries

1 Tries

A **Trie** is a type of data structure that is used for storing strings, numbers, or just about anything that has some sort of "prefix". These are very niche data structures; however, they are very useful at what they do. Tries, like most data structures, are composed up of nodes. Each Trie node has an array of children, the array's size is dependent on how many letters are in your alphabet. Additionally, each node will have a boolean value that will tell us if the letter we are at marks the end of a word, we will call nodes at the end of words, blue nodes. The API for a trie node , assuming we are dealing with words, is as follows

```
1 public class Trie(){
2     private class TrieNode{
3        char value;
4        char[] children;
5        boolean isbluenode;
6     public void insert(String s);
7     public boolean search(String s)
8     public TrieNode root;
9 }
```

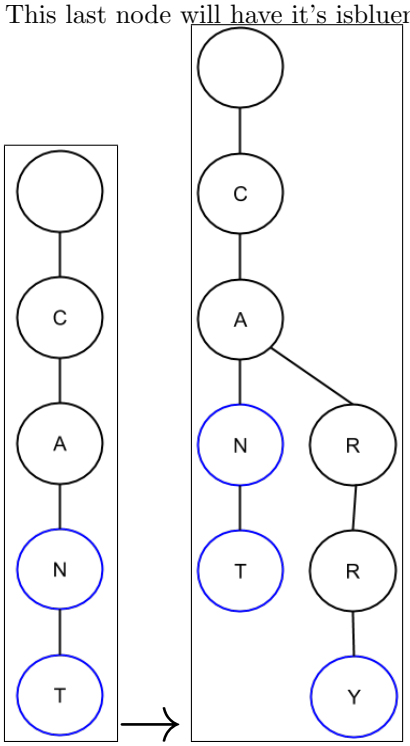
We will now go over an example of how to construct a trie.



To start off with a trie, we will have one empty "parent" node. We will see why this is necessary a bit later.

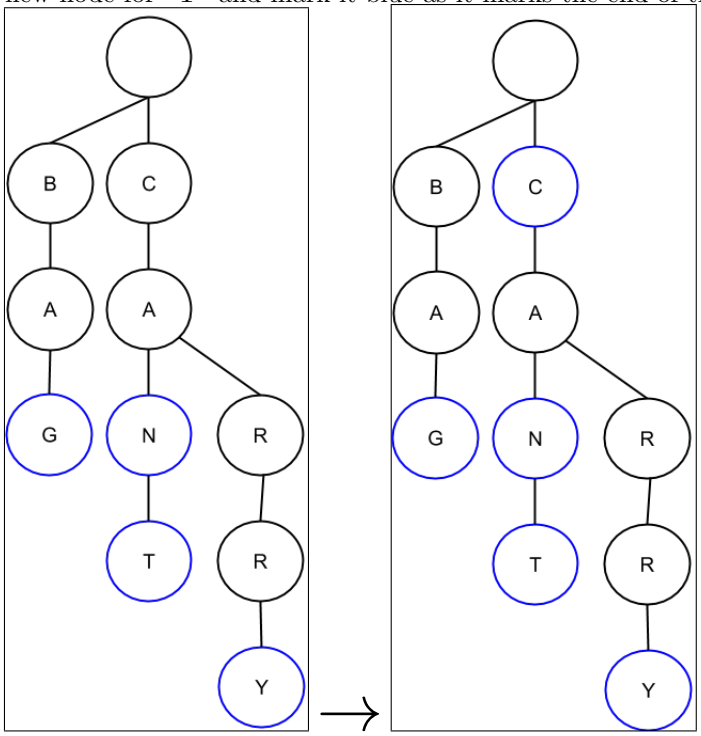
We will start off by inserting the word "CAN". The first node that we insert is one which has a value of "C". Under it we put a node that has a value of "A", and finally, we have a node that has a value of "N".

This last node will have it's isbluenode boolean made to true so that we know that "CAN" is a word.



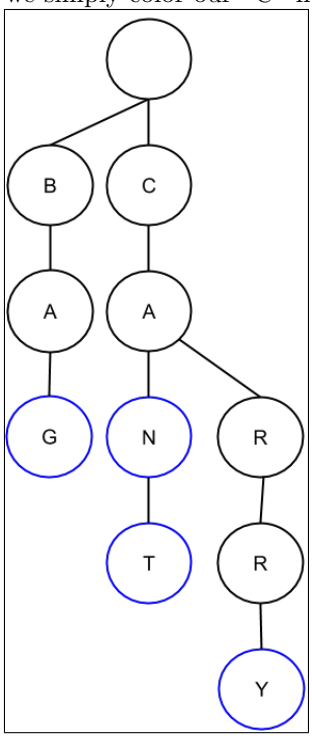
Now we will insert the word "CANT". To do this, we trace down our trie to see how long the longest prefix.

We move through "C", "A", "N" and realize that there is no more nodes to trace through so we create a new node for "T" and mark it blue as it marks the end of the word "CANT".



Next, we insert "CARRY". We trace through "C" and "A" then realize that our "longest prefix is done", now we insert the rest of the nodes "R", "R", and "Y", marking the "Y" node blue. Finally, we will insert

"BAG". We note that there is no prefix that starts with "B" yet. This means that we need to create our word from scratch. We create "BAG" with the "G" node being blue. Now, we will insert the string "C" into our trie. What we notice is that "C" exists as a part of "CAN"/"CANT". So instead of inserting a new C, we simply color our "C" node blue.



The final string that we will add to our trie is "FUN". We simply insert "F" as a child of the root node and then after that, we insert "U" and then "N".

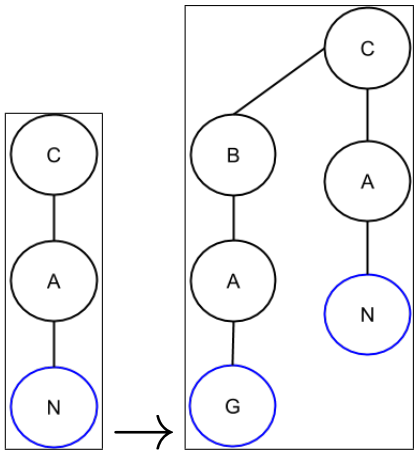
The power of the trie can be seen in the above example for "C". We do not have to create a new node for C since it is a part of "CAN"/"CANT". Other searching data structures that we have dealt with have some form of comparison. With the trie data structure, we no longer need to compare. We don't see if "CAN" is less than "CANT", we just base our searching/insertion off of prefixes. This leads to an enhanced search time.

Tries do have a pretty good runtime, specifically, their worst case runtime for insert is $\Theta(M)$ where M is the length of the string that you are attempting to insert. This runtime comes at a trade off of memory. Each node will have to have an array that is the size of the alphabet, which we will represent by R. For N keys of length L, this means that you will have $N * L * R$ space total, quite a large amount.

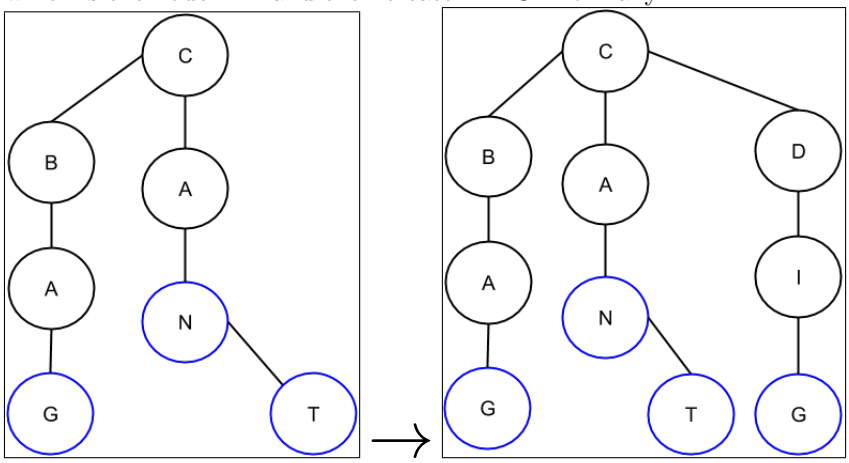
2 Ternary Search Tries

Ternary Search Tries are a data structure that serve the same purpose as Tries; however, they save a lot of memory, they do this by having a worse overall runtime. Instead of having an array that is the size of the full alphabet, each node has 3 sets of links- items that are less than, equal, or greater than the current node. In this case, less than means that the edge points towards a character that is less than the current one, greater than means the edge points towards a character that is greater than the node and an equal child refers to the next character in the sequence.

We will now go over how to construct a Ternary Search Trie:

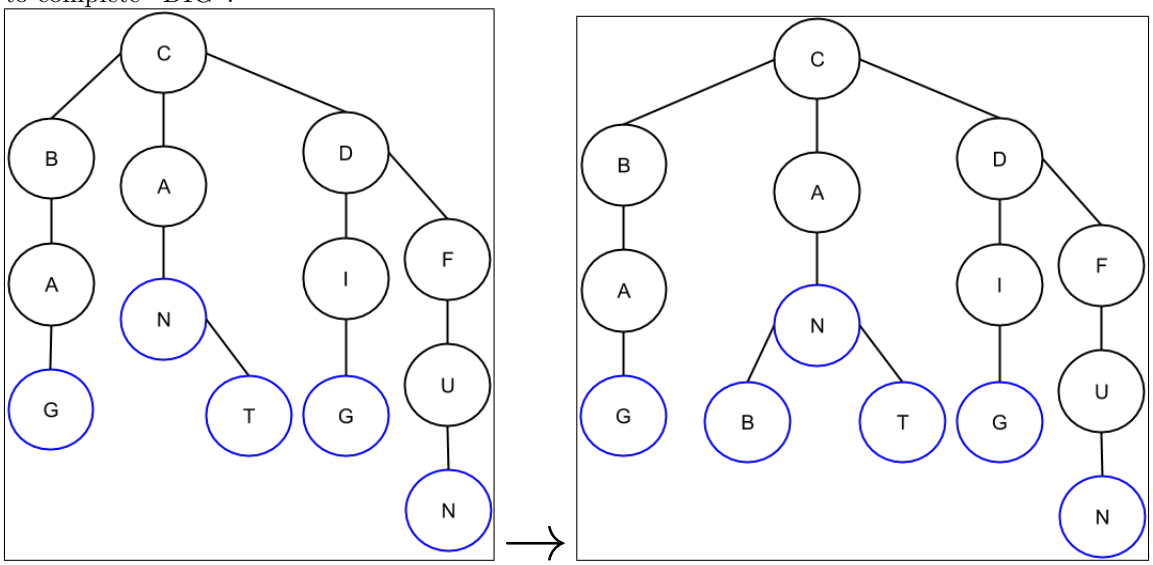


We initially start by inserting the string "CAN". The marking of a word via a blue node is true for ternary search tries just as it is for normal tries. We then insert the word "BAG". We insert a left child for "C" which is the node "B" and then create "BAG" normally.



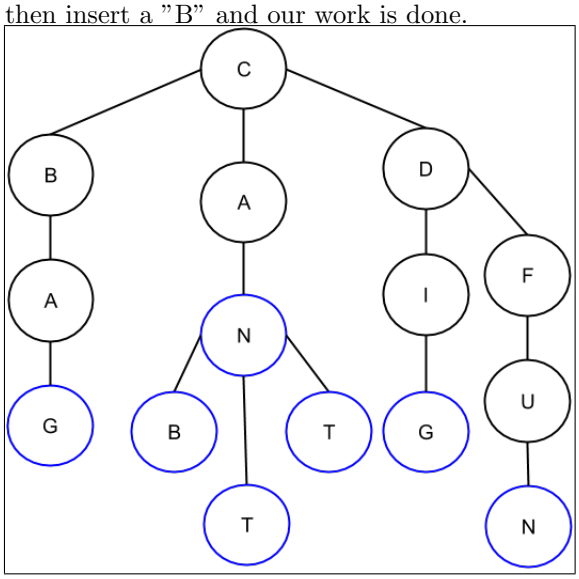
We now insert the word "CAT". To do so, we go through our current "C" and "A" nodes. We realize "T" is not the next item in the sequence, so we move to "N" and look to the right. There is no "T" so we mae one. It is very important to see that the "T" is the right child of the "N" from the word "CAN". Following this, we insert the word "DIG". We realize that our first letter is not "C" so we look to it's right child and see that one does not exist. To fix this we create some "D" node and then follow through to normal process

to complete "DIG".



The next insertion that is performed is "FUN". We start at C and see it has a right child "D". At the "D" node, we realize we need to go one more node right but there isn't one that exists yet. To fix this, we insert an "F" node and then we do the normal process to finish the rest of "FUN"

We'll now insert "CAB". We trace down the ternary search trie and go down the "C" and the "A. At this point, we need to insert the character "B". We move down to the node "N" and see it has no left child. We then insert a "B" and our work is done.



The final insertion we do is "CANT". We follow the same steps as before and at the "N" node, we see we are continuing and make a new bottom link "T". After this, our process is done.

In the above example, there are two "maxed out" nodes, the "C" at the start of "CAN" and the "N" in "CAN". Both of these nodes have a left child, an element less than them, a right child, an element greater than them, and a middle child, which continues the same word. The left child and right child of a node are continuations of the prefix above the node, but does not include the word itself.

The structure of ternary search tries are quite different from normal Tries, and while this method does save memory since we are not allocating memory for the entire alphabet, it comes at a cost of time. We can

compare the example from the Trie secton to the one in this section. The word "FUN" can be completed in 4 levels in the normal trie; however, in a ternary search trie, we must use 5 levels. This difference be even more extreme if we have words that all start on different alphabets.
In the worst case, Ternary Search Tries can take $O(N)$ time for all the insertion and deletion operations

where N is the amount of keys. This can happen in the case where your ternary search trie takes on the shape of many left/right leaning linked lists. On average; however, the insertion and deletion operations take $\Theta(nlogn)$ time because that happens to be the average height of a tree/trie. The memory saved compared to a normal trie is substantial. Ternary Search Tries only take up $O(N * L)$ space where N is the amount of keys and L the maximum length of a key.