Using Ternary Search Trees

Step 1

Thus far, we have discussed implementing a lexicon using a few different data structures and we have discussed their respective pros and cons. Recall that the "best" data structures we discussed for this task were the following, where *n* is the number of words in our lexicon and *k* is the length of the longest word:

- A **balanced Binary Search Tree** (such as an **AVL Tree** or a **Red-Black Tree**), which is the **most space-efficient** we can get, but has a **worst-case time complexity** of **O(log *n*)**. A **Binary Search Tree** has the added benefit of being able to iterate over the elements of the lexicon in **alphabetical order**
- A **Hash Table**, which is not *quite* as space-efficient as a **Binary Search Tree** (but not *too* bad), and which has an **average-case time complexity** of **O(*k*)** (when we take into account the time it takes to compute a hash value of a string of length *k*) and a **worst-case time complexity** of **O(*n*)**. Unfortunately, the elements of a **Hash Table** are **unordered**, so there is no clean way of iterating through the elements of our lexicon in a meaningful order
- A **Multiway Trie**, which is the most **time-efficient** we can get in the **worst case**, **O(*k*)**, but which is ***extremely* inefficient memory-wise**. A **Multiway Trie** has the added benefit of being able to iterate over the elements of the lexicon in **alphabetical order** as well as the ability to perform **auto-complete** by performing a simple **pre-order traversal**

In this section, we will discuss the **Ternary Search Tree**, which is a data structure that serves as a middle-ground between the **Binary Search Tree** and the **Multiway Trie**. The **Ternary Search Tree** is a type of trie, structured in a fashion similar to **Binary Search Trees**, that was first described in 1979 by Jon Bentley and James Saxe.

Figure: Jon Bentley

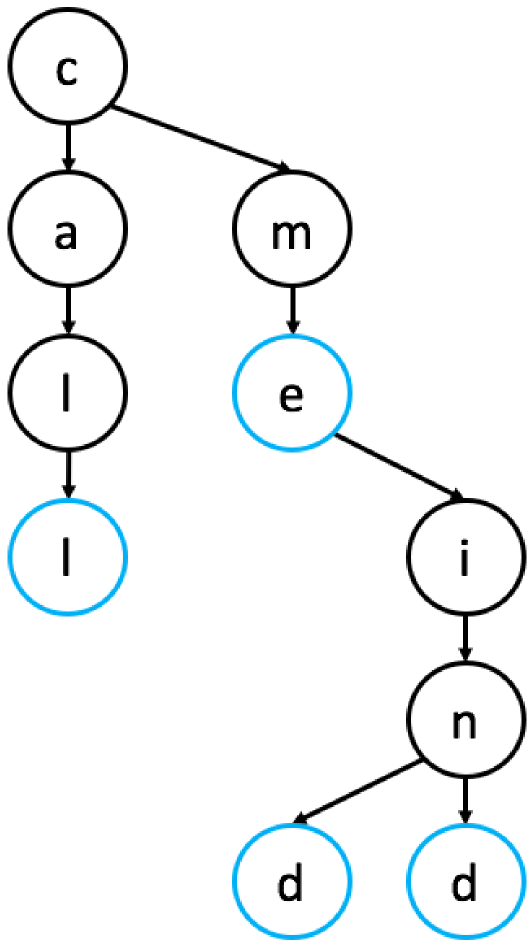
Step 2

The **Trie** is a tree structure in which the elements that are being stored are *not* represented by the value of a single node. Instead, elements stored in a **Trie** are denoted by the concatenation of the labels on the path from the root to the node representing the corresponding element. The **Ternary Search Tree (TST)** is a type of trie in which nodes are arranged in a manner similar to a **Binary**

Search Tree, but with up to three children rather than the binary tree's limit of two.

Each node of a **Ternary Search Tree** stores a single character from our alphabet Σ and can have three children: a *middle child*, *left child*, and *right child*. Further, just like in a **Multiway Trie**, nodes that represent keys are labeled as "word nodes" in some way (for our purposes, we will color them **blue**). Just like in a **Binary Search Tree**, for every node *u*, the *left child* of *u* must have a value *less than u*, and the *right child* of *u* must have a value *greater than u*. The *middle child* of *u* represents the next character in the current word.

Below is an example of a **Ternary Search Tree** that contains the words "call," "me," "mind," and "mid":



If it is unclear to you *how* this example stores the words we listed above, as well as how to go about finding an arbitrary query word, that is perfectly fine. It will hopefully become more clear as we work through more examples together.

Step 3

In a **Multiway Trie**, a word was defined as the concatenation of edge labels along the path from the root to a "word node." In a **Ternary Search Tree**, the definition of a word is a bit more complicated:

For a given "word node," define the path from the root to the "word node" as *path*, and define *S* as the set of all nodes in *path* that have a middle child also in *path*. The word represented by the "word node" is defined as the concatenation of the labels of each node in *S*, along with the label of the "word node" itself.

To **find** a word *key*, we start our tree traversal at the root of the the **Ternary Search Tree**. Let's denote the current node as *node* and the current letter of *key* as *letter*:

- If *letter* is less than *node*'s label: If *node* has a left child, traverse down to *node*'s left child. Otherwise, we have failed (*key* does not exist in this **Ternary Search Tree**)

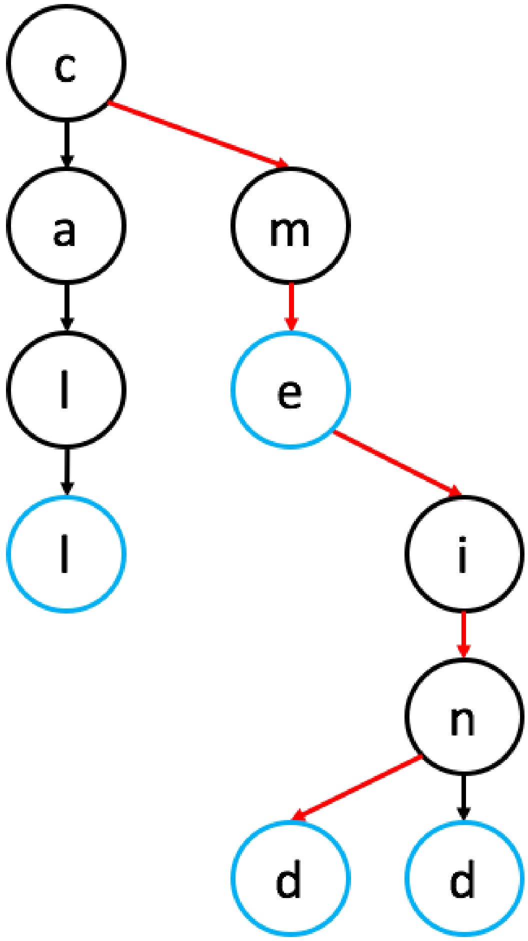
- If *letter* is greater than *node*'s label: If *node* has a right child, traverse down to *node*'s right child. Otherwise, we have failed (*key* does not exist in this **Ternary Search Tree**)
- If *letter* is equal to *node*'s label: If *letter* is the last letter of *key* and if *node* is labeled as a "word node," we have successfully found *key* in our **Ternary Search Tree**; if not, we have failed. Otherwise, if *node* has a middle child, traverse down to *node*'s middle child and set *letter* to the next character of *key*; if not, we have failed (*key* does not exist in this **Ternary Search Tree**)

Below is formal pseudocode for the find algorithm of the **Ternary Search Tree**:

```
find(key): // return True if key exists in this TST, otherwise return False
    node = root node of the TST
    letter = first letter of key
    loop infinitely:
        // left child
        if letter < node.label:
            if node has a left child:
                node = node.leftChild
            else:
                return False     // key cannot exist in this TST

        // right child
        else if letter > node.label:
            if node has a right child:
                node = node.rightChild
            else:
                return False     // key cannot exist in this TST

        // middle child
        else:
            if letter is the last letter of key and node is a word-node:
                return True      // we found key in this TST!
            else:
                if node has a middle child:
                    node = node.middleChild
                    letter = next letter of key
                else:
                    return False // key cannot exist in this TST
```

Step 4

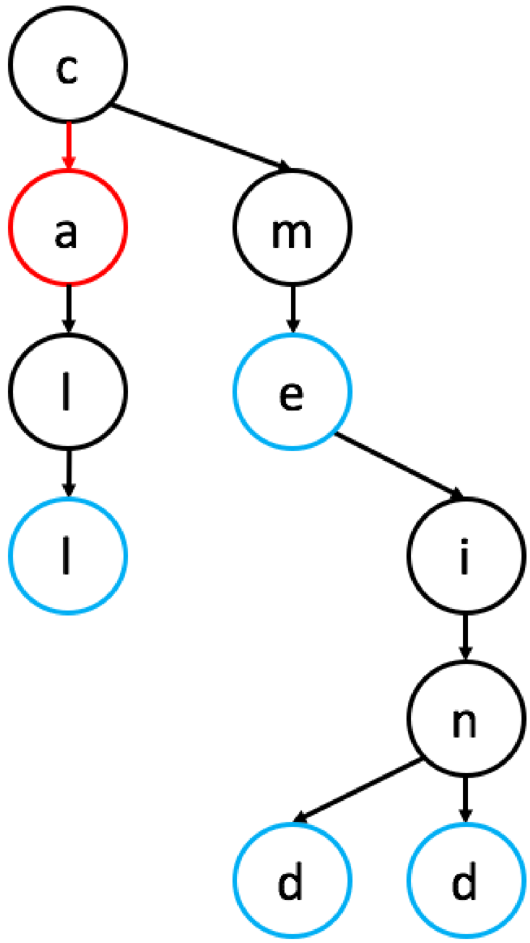
Below is the same example from the previous step, and we will step through the process of finding the word "mid":



1. We start with *node* as the root node ('c') and *letter* as the first letter of "mid" ('m')
2. *letter* ('m') is greater than the label of *node* ('c'), so set *node* to the right child of *node* ('m')
3. *letter* ('m') is equal to the label of *node* ('m'), so set *node* to the middle child of *node* ('e') and set *letter* to the next letter of "mid" ('i')
4. *letter* ('i') is greater than the label of *node* ('e'), so set *node* to the right child of *node* ('i')
5. *letter* ('i') is equal to the label of *node* ('i'), so set *node* to the middle child of *node* ('n') and set *letter* to the next letter of "mid" ('d')
6. *letter* ('d') is less than the label of *node* ('n'), so set *node* to the left child of *node* ('d')
7. *letter* ('d') is equal to the label of *node* ('d'), *letter* is already on the last letter of "mid" ('d'), and *node* is a "word node", so **success**!

Step 5

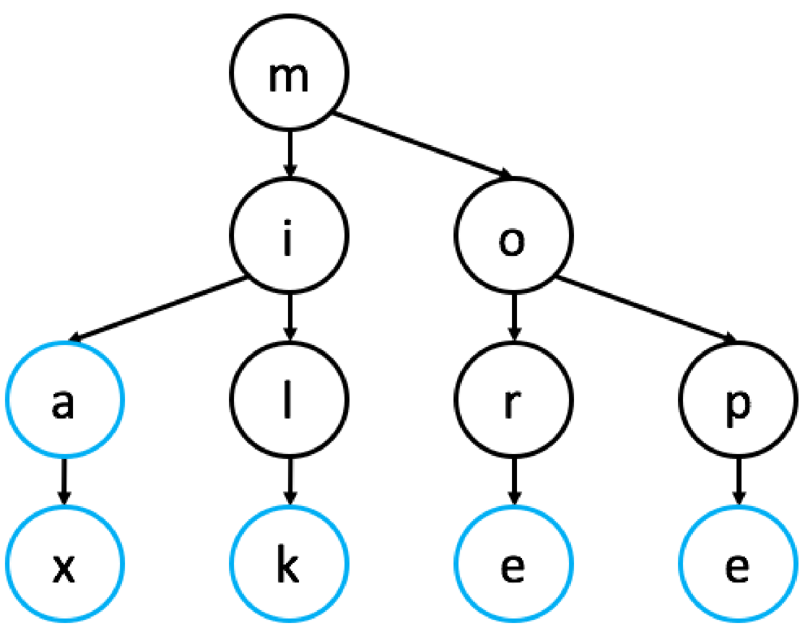
Using the same example as before, let's try finding the word "cme," which might *seem* like it exists, but it actually doesn't:



1. We start with *node* as the root node ('c') and *letter* as the first letter of "cme" ('c')
2. *letter* ('c') is equal to the label of *node* ('c'), so set *node* to the middle child of *node* ('a') and set *letter* to the next letter of "cme" ('m')
3. *letter* ('m') is greater than the label of *node* ('a'), but *node* does not have a right child, so we **failed**

Step 6

EXERCISE BREAK: Which of the following words appear in the **Ternary Search Tree** below? (Select all that apply)



To solve this problem please visit https://stepik.org/lesson/30820/step/6

Step 7

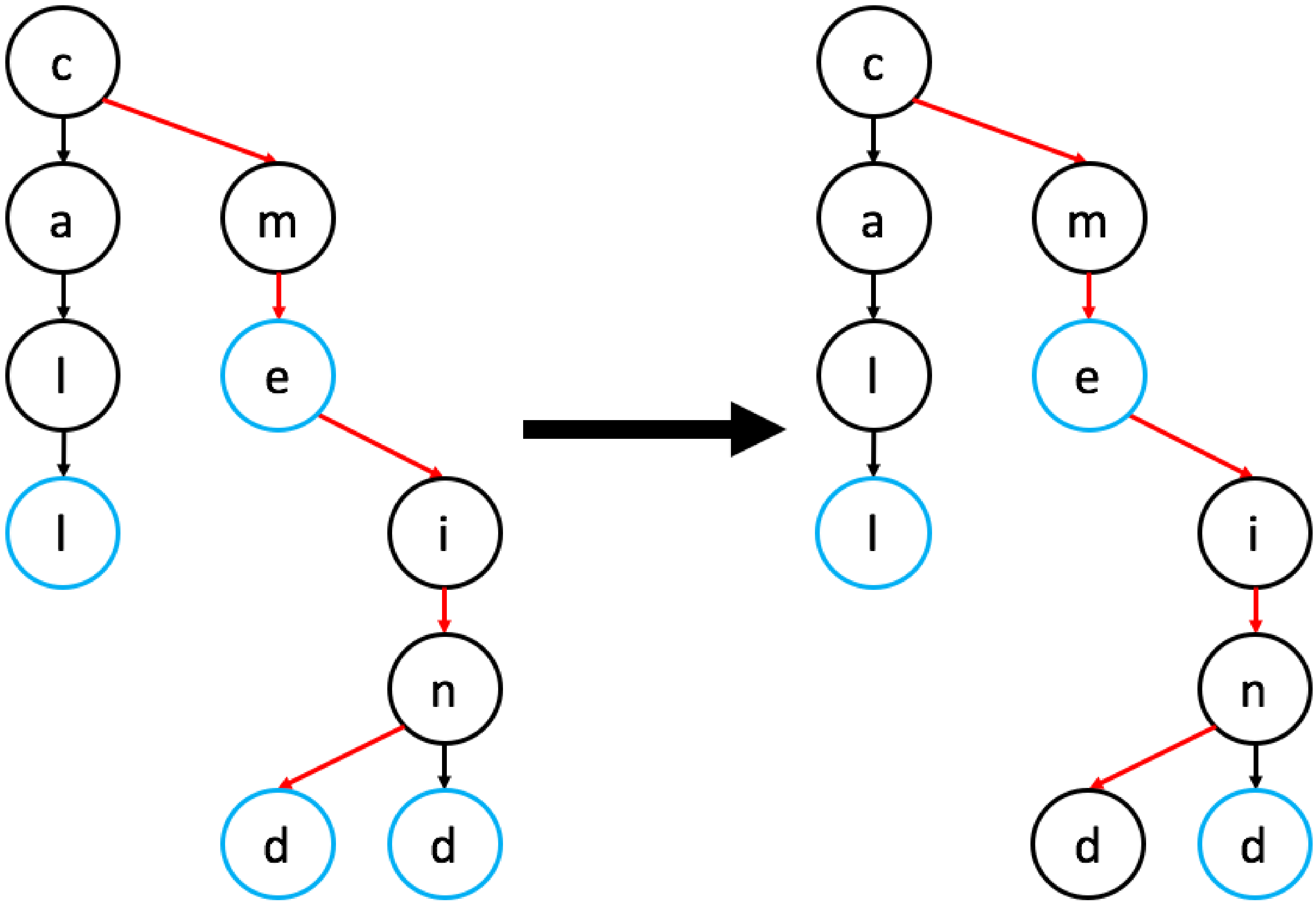
The **remove** algorithm is extremely trivial once you understand the find algorithm. To remove a word *key* from a **Ternary Search Tree**, simply perform the find algorithm. If you successfully find *key*, simply remove the "word node" label from the node at which you end up.

Below is formal pseudocode for the remove algorithm of the **Ternary Search Tree**:

```
remove(key): // remove key if it exists in this TST
    node = root node of the TST
    letter = first letter of key
    loop infinitely:
        // left child
        if letter < node.label:
            if node has a left child:
                node = node.leftChild
            else:
                return                            // key cannot exist in this TST

        // right child
        else if letter > node.label:
            if node has a right child:
                node = node.rightChild
            else:
                return                            // key cannot exist in this TST

        // middle child
        else:
            if letter is the last letter of key and node is a word-node:
                remove the word-node label from node // found key, so remove it from the TST
                return
            else:
                if node has a middle child:
                    node = node.middleChild
                    letter = next letter of key
                else:
                    return                        // key cannot exist in this TST
```

Step 8

Below is the initial example of a **Ternary Search Tree**, and we will demonstrate the process of removing the word "mid":



1. We start with *node* as the root node ('c') and *letter* as the first letter of "mid" ('m')
2. *letter* ('m') is greater than the label of *node* ('c'), so set *node* to the right child of *node* ('m')
3. *letter* ('m') is equal to the label of *node* ('m'), so set *node* to the middle child of *node* ('e') and set *letter* to the next letter of "mid" ('i')
4. *letter* ('i') is greater than the label of *node* ('e'), so set *node* to the right child of *node* ('i')
5. *letter* ('i') is equal to the label of *node* ('i'), so set *node* to the middle child of *node* ('n') and set *c* to the next letter of "mid" ('d')
6. *letter* ('d') is less than the label of *node* ('n'), so set *node* to the left child of *node* ('d')
7. *letter* ('d') is equal to the label of *node* ('d'), *letter* is already on the last letter of "mid", and *node* is a "word node", so "mid" exists in the tree!
8. Remove the "word node" label from *node*

Step 9

The **insert** algorithm also isn't too bad once you understand the find algorithm. To insert a word *key* into a **Ternary Search Tree**, perform the find algorithm:

- If you're able to legally traverse through the tree for every letter of *key* (which implies *key* is a prefix of another word in the tree), simply label the node at which you end up as a "word node"
- If you are performing the tree traversal and run into a case where you want to traverse left or right, but no such child exists, create a new left/right child labeled by the current letter of *key*, and then create middle children labeled by each of the remaining letters of *key*
- If you run into a case where you want to traverse down to a middle child, but no such child exists, simply create middle children labeled by each of the remaining letters of *key*

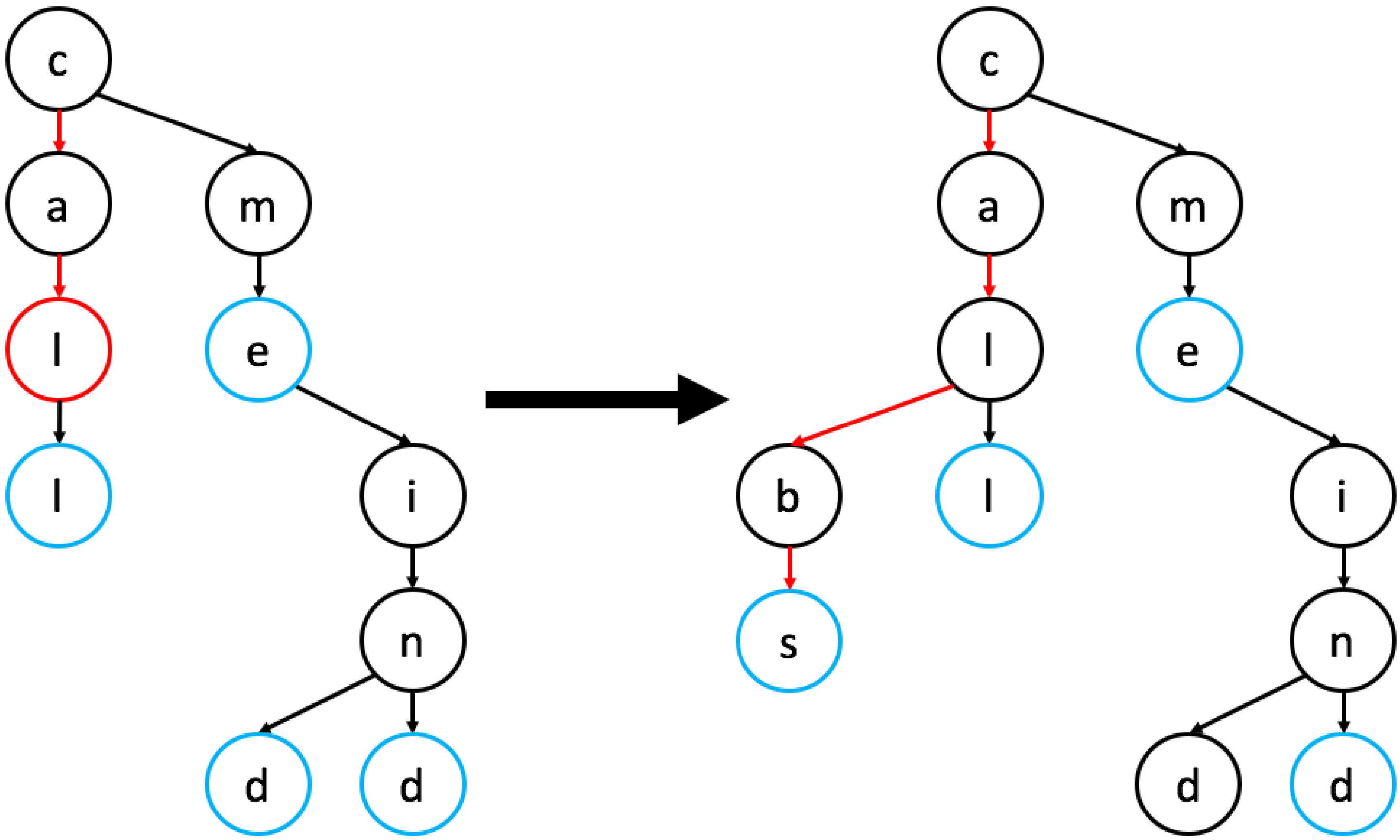
Note that, for the same reasons insertion order affected the shape of a **Binary Search Tree**, the order in which we insert keys into a **Ternary Search Tree** affects the shape of the tree. For example, just like in a **Binary Search Tree**, the root node is determined by the first element inserted into a **Ternary Search Tree**.

Below is formal pseudocode for the insert algorithm of the **Ternary Search Tree**:

```
insert(key): // insert key into this TST
    node = root node of the TST
    letter = first letter of key

    loop infinitely:
        // left child
        if letter < node.label:
            if node has a left child:
                node = node.leftChild

            else:
                node.leftChild = new node labeled by letter
                node = node.leftChild

                iterate letter over the remaining letters of key:
                    node.middleChild = new node labeled by letter
                    node = node.middleChild

                label node as a word-node     // inserted key into the TST

        // right child
        else if letter > node.label:
            if node has a right child:
                node = node.rightChild

            else:
                node.rightChild = new node labeled by letter
                node = node.rightChild

                iterate letter over the remaining letters of key:
                    node.middleChild = new node labeled by letter
                    node = node.middleChild

                label node as a word-node     // inserted key into the TST

        // middle child
        else:
            if letter is the last letter of key:
                label node as a word-node     // inserted key into the TST

            else:
                if node has a middle child:
                    node = node.middleChild

                else:
                    iterate letter over the remaining letters of key:
                        node.middleChild = new node labeled by letter
                        node = node.middleChild

                    label node as a word-node // insert key into the TST
```

Step 10

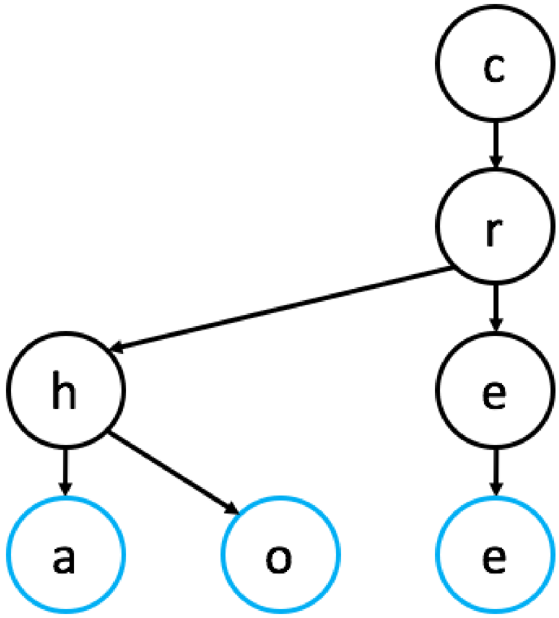
Below is the initial example of a **Ternary Search Tree**, and we will demonstrate the process of inserting the word "cabs":



1. We start with *node* as the root node ('c') and *letter* as the first letter of "cabs" ('c')
2. *letter* ('c') is equal to the label of *node* ('c'), so set *node* to the middle child of *node* ('a') and set *letter* to the next letter of "cabs" ('a')
3. *letter* ('a') is equal to the label of *node* ('a'), so set *node* to the middle child of *node* ('l') and set *letter* to the next letter of "cabs" ('b')
4. *letter* ('b') is less than the label of *node* ('l'), but *node* does not have a left child:
5. Create a new node as the left child of *node*, and label the new node with *letter* ('b')
6. Set *node* to the left child of *node* ('b') and set *letter* to the next letter of "cabs" ('s')
7. Create a new node as the middle child of *node* ('s'), and label the new node with *letter* ('s')
8. Set *node* to the middle child of *node* ('s')
9. *letter* is already on the last letter of "cabs", so label *node* as a "word node" and we're done!

Step 11

EXERCISE BREAK: Sort the keys stored in the **Ternary Search Tree** below in the order in which they must have been inserted in order to obtain the observed tree structure.



To solve this problem please visit https://stepik.org/lesson/30820/step/11

Step 12

Saying that the structure of a **Ternary Search Tree** is "clean" is a bit of a stretch, especially in comparison to the structure of a **Multiway Trie**, but the structure is ordered nonetheless because of the **Binary Search Tree** property that a **Ternary Search Tree** maintains. As a result, just like in a **Multiway Trie**, we can iterate through the elements of a **Ternary Search Tree** in sorted order by performing a **pre-order traversal** on the trie. Below is the pseudocode to recursively output all words in a **Ternary Search Tree** in ascending or descending alphabetical order (we would call either function on the root):

```
ascendingPreOrder(node): // Recursively iterate over the words in ascending order
    if node is a word-node:
        output the word labeled by path from root to node
    for each child of node (in ascending order):
        ascendingPreOrder(child)
```

```
descendingPreOrder(node): // Recursively iterate over the words in descending order
    if node is a word-node:
        output the word labeled by path from root to node
    for each child of node (in descending order):
        descendingPreOrder(child)
```

We can use this recursive **pre-order traversal** technique to provide another useful function to our **Ternary Search Tree**: auto-complete. If we were given a prefix and we wanted to output *all* words in our **Ternary Search Tree** that start with this prefix, we can traverse down the trie along the path labeled by the prefix, and we can then call the recursive **pre-order traversal** function on the node we reached.

Step 13

We have now discussed yet another data structure that can be used to implement a lexicon: the **Ternary Search Tree**. Because of the **Binary Search Tree** properties of the **Ternary Search Tree**, the ***average*-case time complexity** to find, insert, and remove elements is **O(log *n*)** as well as a ***worst*-case time complexity** of **O(*n*)**. Also, because inserting elements in a **Ternary Search Tree** is very similar to inserting elements in a **Binary Search Tree**, the structure of a **Ternary Search Tree** (i.e., the balance), and as a result, the performance of a **Ternary Search Tree**, largely depends on the order in which we insert elements. As a result, if the words we will be inserting are known in advance, it is common practice to randomly shuffle the words before inserting them to help improve the balance of the tree.

Because of the structure of a **Ternary Search Tree**, we can easily and efficiently iterate through the words in the lexicon in alphabetical order (either ascending or descending order) by performing a **pre-order traversal**. We can use this exact **pre-order traversal** technique to create **auto-complete** functionality for our lexicon.

Ternary Search Trees are a bit slower than **Multiway Tries**, but they are *significantly* more **space-efficient**, and as a result, they are often chosen as the data structure of choice when people implement lexicons. In short, **Ternary Search Trees** give us a nice middle-ground between the **time-efficiency** of a **Multiway Trie** and the **space-efficiency** of a **Binary Search Tree**.