Using Multiway Tries

Step 1

With regards to implementing our three lexicon functions, we've figured out how to obtain a O(log *n*) worst-case time complexity—where *n* is the number of words in the lexicon—by using a balanced **Binary Search Tree**. We've also figured out how to obtain a O(*k*) average-case time complexity—where *k* is the length of the longest word in the lexicon—by using a **Hash Table** (O(*k*) to compute the hash value of a word, and then O(1) to actually perform the operation). Can we do even better?

In this section, we will discuss the **Multiway Trie**, a data structure designed for the exact purpose of storing a set of words. It was first described by R. de la Briandais in 1959, but the term *trie* was coined two years later by Edward Fredkin, originating from the word *retrieval* (as they were used to *retrieve* words). As such, *trie* was originally pronounced "tree" (as in the middle syllable of *retrieval*), but it is now typically pronounced "try" in order to avoid confusion with the word *tree*.

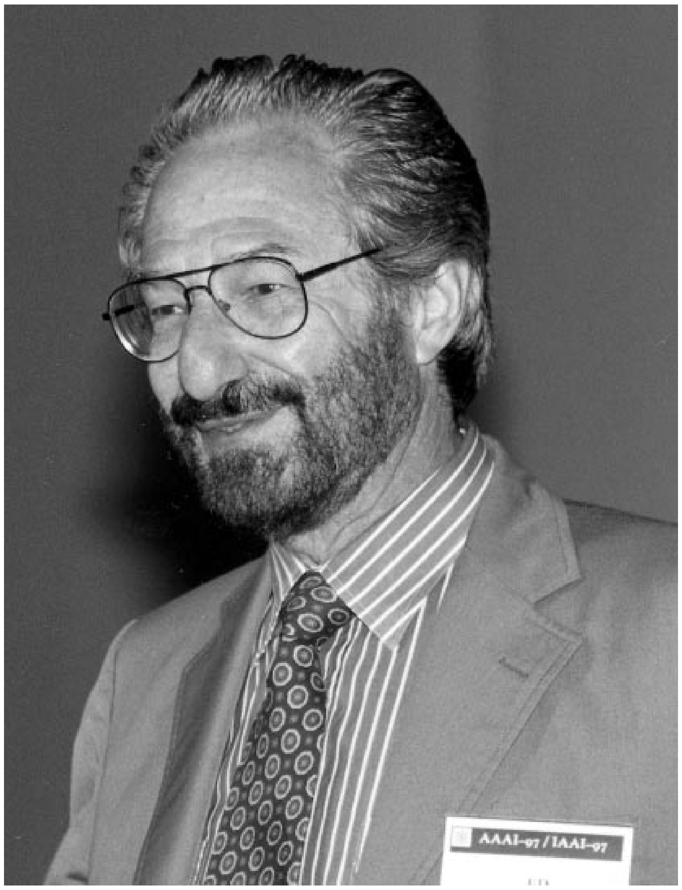


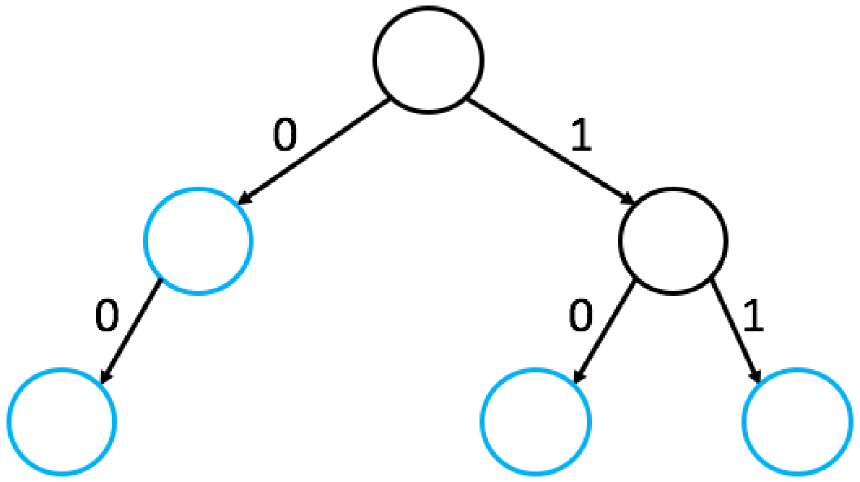
Figure: Edward Fredkin

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. Nodes that represent keys are labeled as "word nodes" in some way (for our purposes, we will color them **blue**). In this section, the **Tries** that we will discuss will have **edge labels**, and it is the concatenation of these edge labels that spell out the words we will store.

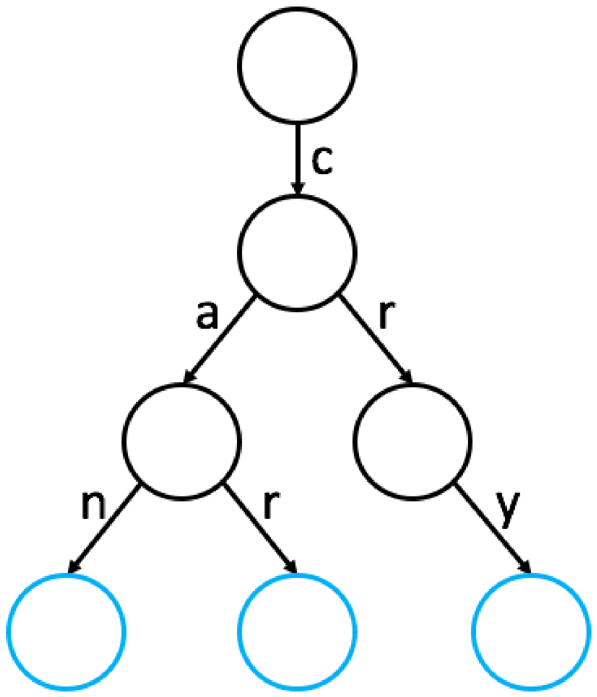
Below is an example of a **Trie**. Specifically, it is an example of a **Binary Trie**: a **Trie** that is a binary tree. The words stored in the **Trie** below are denoted by the prefixes leading up to all **blue** nodes: 0, 00, 10, and 11. Note that 01 is not in this trie because there is no valid path from the root labeled by 01. Also note that 1 is not in this trie because, although there is a valid path from the root labeled by

1, the node that we end up at is not a "word node."



Of course, as humans, we don't speak in binary, so it would be more helpful for our data structure to be able to represent words in our own alphabet. Instead of having only two edges coming out of each node (labeled with either a 1 or a 0), we can expand our alphabet to any alphabet we choose! A **Trie** in which nodes can have more than two edges are known as **Multiway Tries (MWT)**. For our purposes, we will use Σ to denote our alphabet. For example, for binary, Σ = {0,1}. For the DNA alphabet, Σ = {A, C, G, T}. For the English alphabet, Σ = {a, ..., z}.

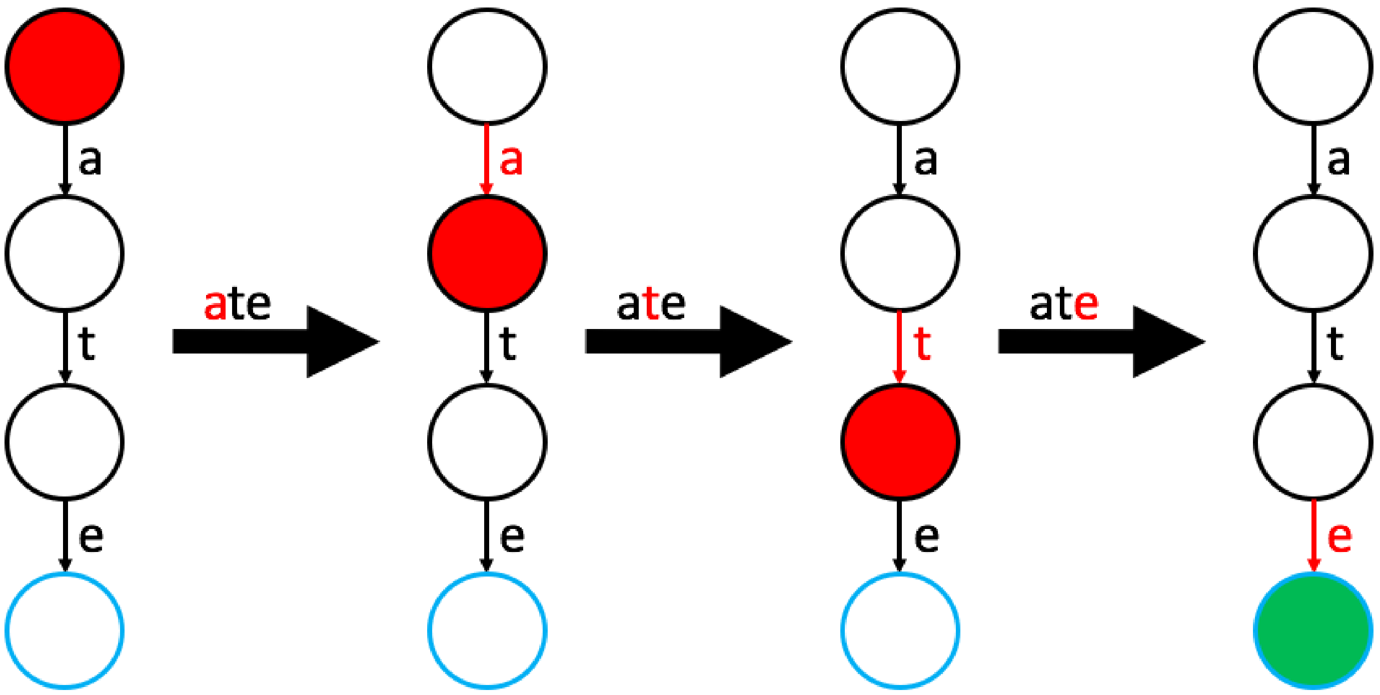
Below is a **Multiway Trie** with Σ = {a, ..., z} containing the following words: *can*, *car*, and *cry*.



Step 3

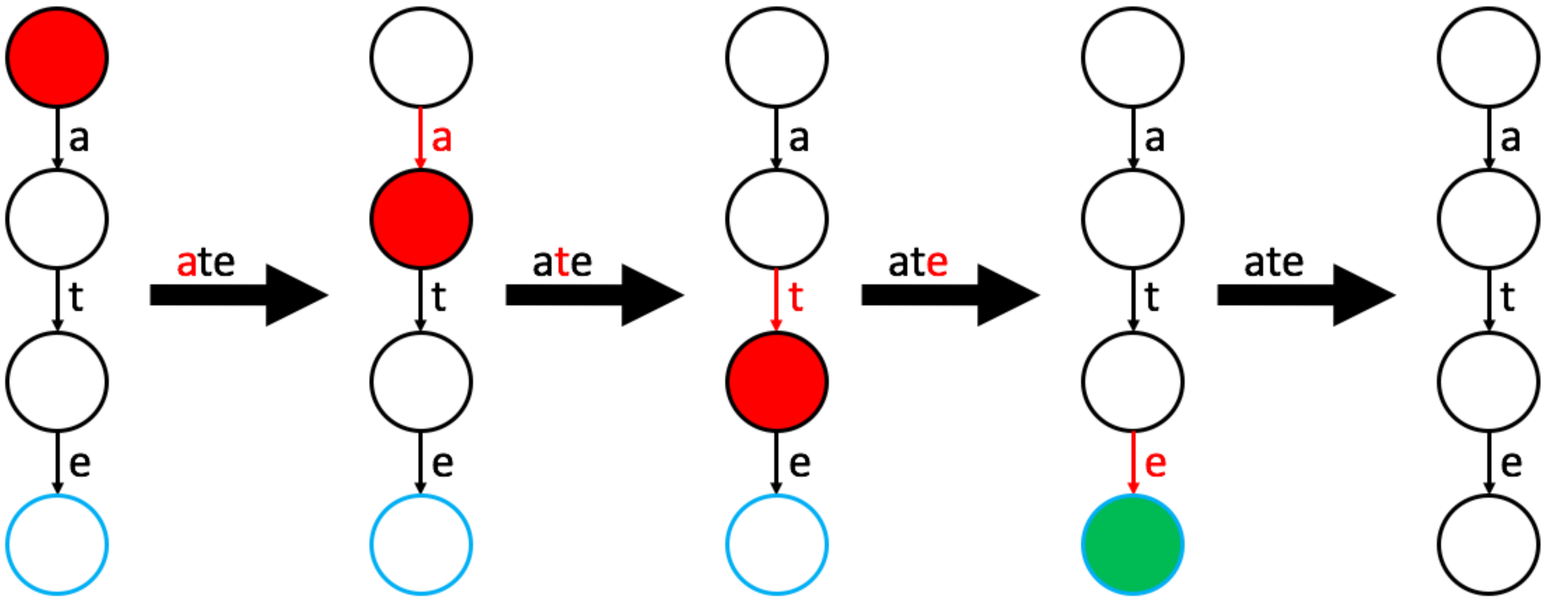
The algorithms behind the insert, remove, and find operations of a **Multiway Trie** are actually very simple. To **find** a word, simply start at the root and, for each letter of the word, follow the corresponding edge of the current node. If the edge you need to traverse does not exist, the word does not exist in the trie. Also, even if you are able to traverse all of the required edges, if the node you land on is not labeled as a "word node," the word does not exist in the trie. A word *only* exists in the trie if you are able to traverse all of the required edges and the node you reach at the end of the traversal is labeled as a "word node".

Below is a simple example, where Σ = {a, ..., z}, on which we perform the find operation on the word "ate". Note that, even though there exists a valid path from the root to the word "at" (which is a prefix of "ate"), the word "at" does not appear in our **Multiway Trie** because the node we reach after following the path labeled by "at" is not labeled as a "word node".



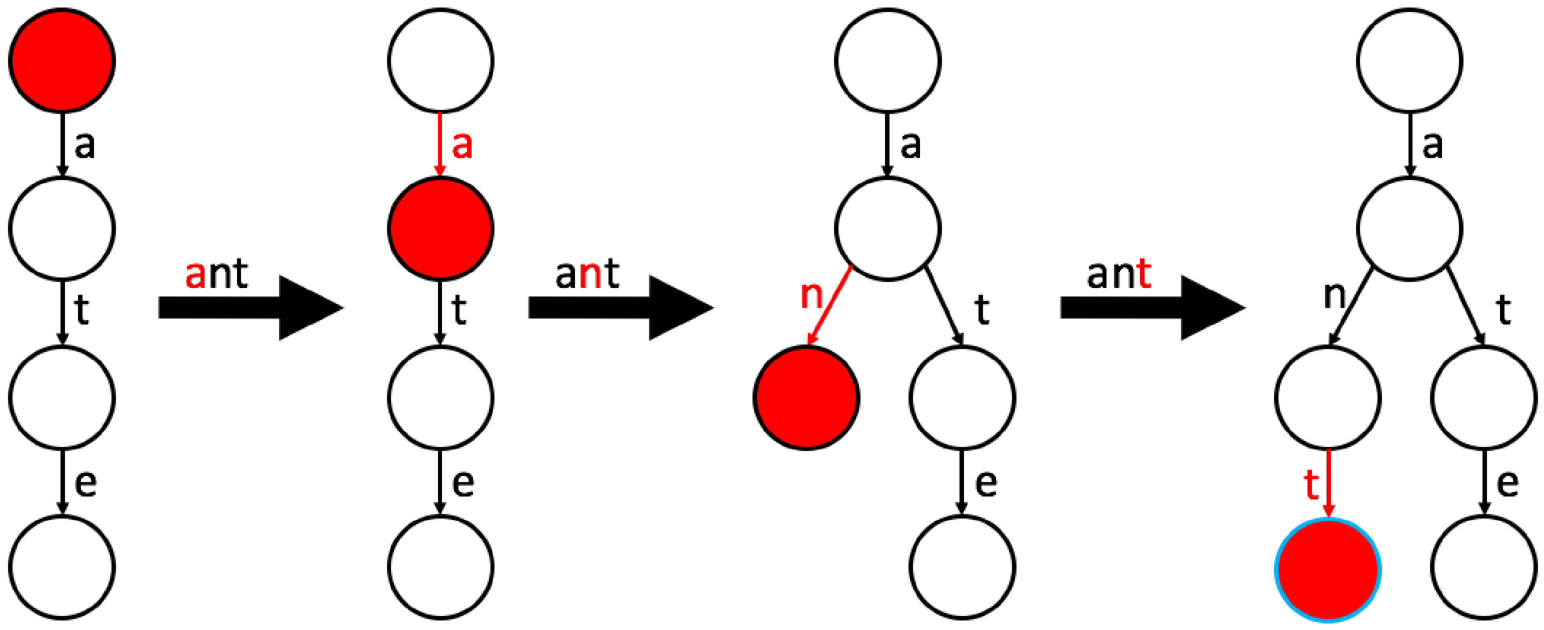
Step 4

To **remove** a word, simply follow the find algorithm. If the find algorithm fails, the word does not exist in the trie, meaning nothing has to be done. If the find algorithm succeeds, simply remove the "word node" label from the node at which the find algorithm terminates. In the example below, we remove the word "ate" from the trie above by simply traversing the path spelled by "ate" and removing the "word node" label from the final node on the path:

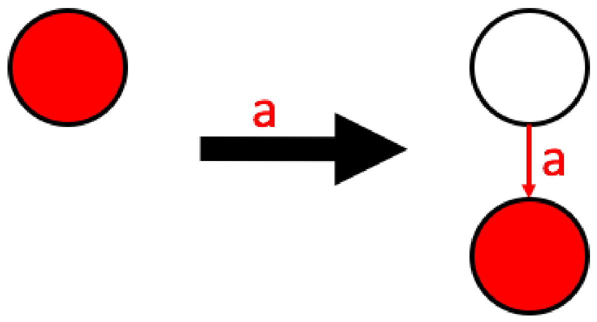


Step 5

To **insert** a word, simply attempt to follow the find algorithm. If, at any point, the edge we need to traverse does not exist, simply create the edge (and node to which it should point), and continue. In the example below, we add the word "ant" to the trie above:

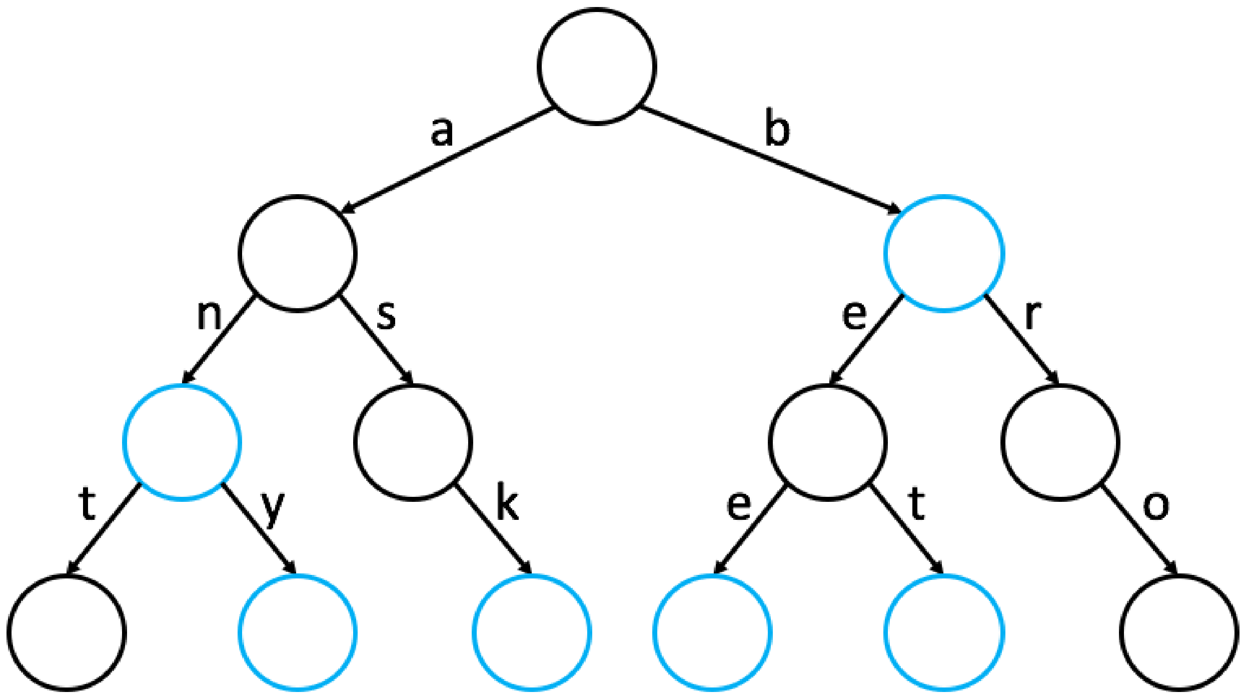


We want to emphasize the fact that, in a **Multiway Trie**, letters label *edges* of the trie, *not* nodes! Although this fact is quite clear given the diagrams, students often make mistakes when implementing **Multiway Tries** in practice. Specifically, note that an "empty" **Multiway Trie** (i.e., a **Multiway Trie** with no keys) is a **single-node tree with no edges**. Then, if I were to insert even a single-letter key ('a' in the example below), I would create a *second* node, and 'a' would label the edge connecting my root node and this new node:



Step 6

EXERCISE BREAK: List all of the words are contained in the **Multiway Trie** below. Enter a single word on each line in any order. Your answer should only contain characters in the alphabet Σ = {a, ..., z} and newline characters separating each word (so no commas, semicolons, etc.).



To solve this problem please visit https://stepik.org/lesson/30819/step/6

Step 7

Below is formal pseudocode for the three operations of a **Multiway Trie** that we mentioned previously. In the pseudocode, we denote the **Multiway Trie's** root node as the variable `root`.

```
find(word): // find word in this Multiway Trie
    curr = root
    for each character c in word:
        if curr does not have an outgoing edge labeled by c:
            return False
        else:
            curr = child of curr along edge labeled by c
    if curr is a word-node:
        return True
    else:
        return False
```

```
remove(word): // remove word from this Multiway Trie
    curr = root
    for each character c in word:
        if curr does not have an outgoing edge labeled by c:
            return
        else:
            curr = child of curr along edge labeled by c
    if curr is a word-node:
        remove the word-node label of curr
```

```
insert(word): // insert word into this Multiway Trie
    curr = root
    for each character c in word:
        if curr does not have an outgoing edge labeled by c:
            create a new child of curr with the edge labeled by c
        curr = child of curr along edge labeled by c
    if curr is not a word-node:
        label curr as a word-node
```

Step 8

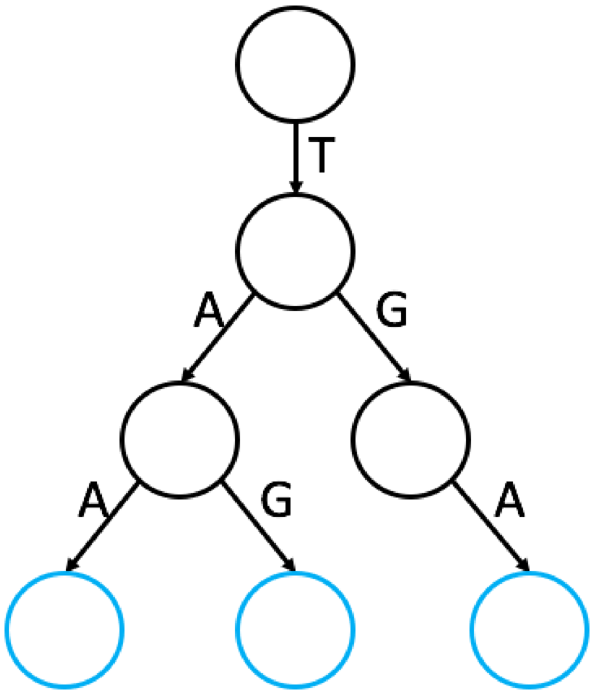
EXERCISE BREAK: Assume that, if you are given a node *u* and a character *c*, you can access the outgoing edge of *u* labeled by *c* (or determine that no such edge exists) in O(1) time. Given this assumption, what is the worst-case **time** complexity of the find, insert, and remove algorithms described on the previous step?

In the answer choices, *n* denotes the number of words stored in the **Multiway Trie**, and *k* denotes the length of the longest work in the **Multiway Trie**.

To solve this problem please visit https://stepik.org/lesson/30819/step/8

Step 9

Abstractly, when we draw **Multiway Tries**, for any given node, we only depict the outgoing edges that we actually see, and we completely omit any edges that we don't observe. For example, below is an example of a **Multiway Trie** with Σ = {A, C, G, T} which stores the words TGA, TAA, and TAG (the three STOP codons of protein translation as they appear in DNA, for those who are interested):

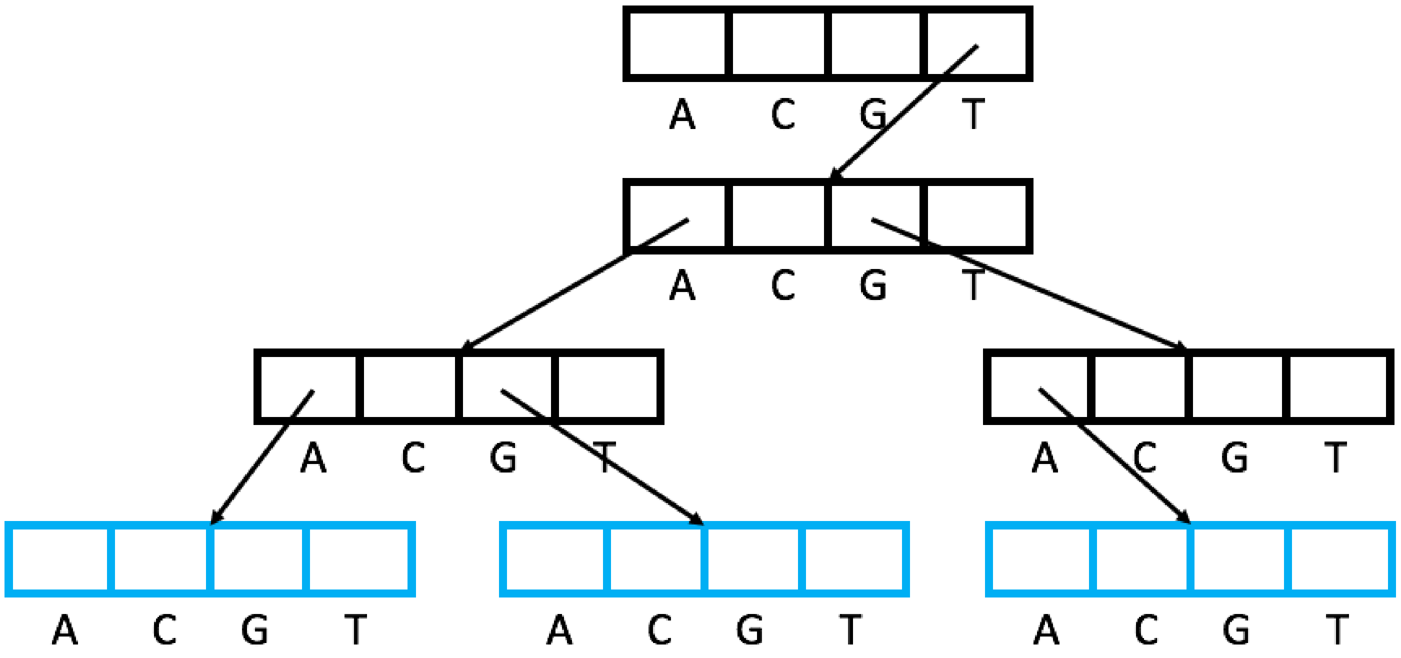


Based on this representation, intuitively, we might think that each Node object should have a list of edges. Recall that our motivation for discussing a **Multiway Trie** was to achieve a worst-case time complexity of O(*k*), where *k* is the length of the longest word, to find, insert, and remove words. This means that each individual edge traversal should be O(1) (and we do *k* O(1) edge traversals, one for each letter of our word, resulting in a O(*k*) time complexity overall). However, if we were to store the edges as a list, we would unfortunately have to perform a O(|Σ|) search operation to find a given edge, where |Σ| is the size of our alphabet.

To combat this and maintain the O(1) edge traversals, we instead allocate space for *all* of the edges that can come out of a given node right off the bat in the form of an array. We fill in the slots for edges that we use, and we leave the slots for unused edges empty. This way, if we're at some node *u* and if we're given some character *c*, we can find the edge coming out of *u* that is labeled by *c* in O(1) time by simply going to the corresponding slot of the array of edges. This O(1) time complexity to find an edge given a character *c* assumes that we have a way of mapping *c* to an index in our array of edges in O(1) time, which is a safe assumption. For example, if Σ = {a, ..., z}, in C++, we could do something like this:

```
// this simple C++ function maps 'a' to 0, 'b' to 1, ..., 'z' to 25 in O(1) time
int index(char c) {
    return (int)c - (int)'a'; // subtract the ASCII value of the char 'a' from that of c
}
```

Below is a diagram representing the same **Multiway Trie** as above, but in a fashion more similar to the actual implementation. Note that, in the example, just like before, we are using the DNA alphabet for the sake of simplicity (i.e., Σ = {A, C, G, T}), *not* the English alphabet.



Of course, the more implementation-like representation of **Multiway Tries** is a bit harder to read, which is why we draw them the more abstract way (as opposed to a bunch of arrays of edges). We will continue to draw them using the abstract way, but be sure to never forget what they actually look like behind-the-scenes.

Step 10

Because of the clean structure of a **Multiway Trie**, we can iterate through the elements in the trie in sorted order by performing a **pre-order traversal** on the trie. Below is the pseudocode to recursively output all words in a **Multiway Trie** 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 **Multiway Trie**: auto-complete. So, if we were given a prefix and we wanted to output *all* the words in our **Multiway Trie** 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 11

CODE CHALLENGE: Implementing a Multiway Trie

In this challenge, you will be implementing a **Multiway Trie** with Σ = {a, ..., z}. We will define a `Node` class for you, and you will use it in your implementation of a **Multiway Trie**.

Below is the C++ `Node` class we have defined for you:

```
class Node {
    public:
        bool word = false;  // Node's "word" label
        Node* children[26]; // children[0] corresponds to 'a', children[1] to 'b', etc.
        Node();             // Node constructor
}

Node::Node(void) {
    for(int i = 0; i < 26; ++i) {
        children[i] = NULL;
    }
}
```

Below is the C++ `Lexicon` class we have declared for you:

```
class MultiwayTrie {
    public:
        Node* root = new Node();  // root node of Multiway Trie
        bool find(string word);   // "find" function of MultiwayTrie class
        void insert(string word); // "insert" function of MultiwayTrie class
        void remove(string word); // "remove" function of MultiwayTrie class
};
```

Sample Input:

```
N/A
```

Sample Output:

```
N/A
```

To solve this problem please visit https://stepik.org/lesson/30819/step/11

Step 12

CODE CHALLENGE: Implementing a Lexicon Using a Multiway Trie

Unfortunately, C++ does not natively have a **Multiway Trie** implementation for us to use in our Lexicon class implementation. Fortunately for you, we have defined the following MultiwayTrie class for you to use (the same class you implemented on the previous step):

```
class MultiwayTrie {
    public:
        bool find(string word);   // "find" function of MultiwayTrie class
        void insert(string word); // "insert" function of MultiwayTrie class
        void remove(string word); // "remove" function of MultiwayTrie class
};
```

In this challenge, you will use the `MultiwayTrie` class we defined for you to implement the `Lexicon` class. Below is the C++ `Lexicon` class we have declared for you:

```
class Lexicon {
    public:
        MultiwayTrie mwt;         // instance variable MultiwayTrie object
        bool find(string word);   // "find" function of Lexicon class
        void insert(string word); // "insert" function of Lexicon class
        void remove(string word); // "remove" function of Lexicon class
};
```

Sample Input:

```
N/A
```

Sample Output:

```
N/A
```

To solve this problem please visit https://stepik.org/lesson/30819/step/12

Step 13

We previously discussed using a **Hash Table** to implement a lexicon, which would allow us to perform find, insert, and remove operations with an *average*-case time complexity of O(k), assuming we take into account the time it takes to compute the hash value of a string of length k (where k is the length of the longest word in the dictionary). However, with this approach, it would be impossible for us to iterate through the words in the lexicon in any meaningful order.

In this chapter, we have now discussed using a **Multiway Trie** to implement a lexicon, which allows us to perform find, insert, and remove operations with a **worst-case time complexity** of **O(k)**. Also, because of the structure of a **Multiway Trie**, 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.

However, **Multiway Tries** are *extremely* **inefficient** in terms of **space usage**: in order to have fast access to edges coming out of a given node u, we need to allocate space for each of the $|\Sigma|$ possible edges that could theoretically come out of u. If our trie is relatively dense, this space usage is tolerable, but if our trie is relatively sparse, there will be a *lot* of space wasted in terms of empty space for possible edges.

In the next section, we will discuss a data structure that lets us obtain a time efficiency *close* to that of a **Multiway Trie**, but without the terrible space requirement: the **Ternary Search Tree**.