CSC 325 Adv Data Structures

Lecture 7

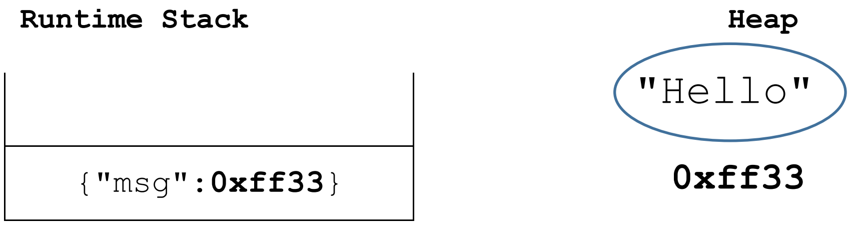
Tries

String processing can be very complex. There are a lot of things we can do with strings, such as substring matching and retrieval, pattern matching, counting occurrences, starts with, ends with, efficient concatenation and more. All of these operations need to worry about space and efficiency. Recall that strings are immutable in some languages (e.g. Java). This means that if I were to write the following code:

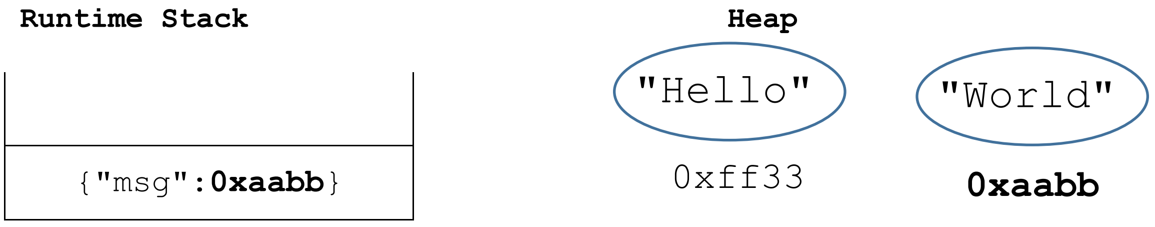
```
String msg = "Hello";
msg = "World";
msg = "Hello " + msg;
msg += "!";
```

We would have the following:

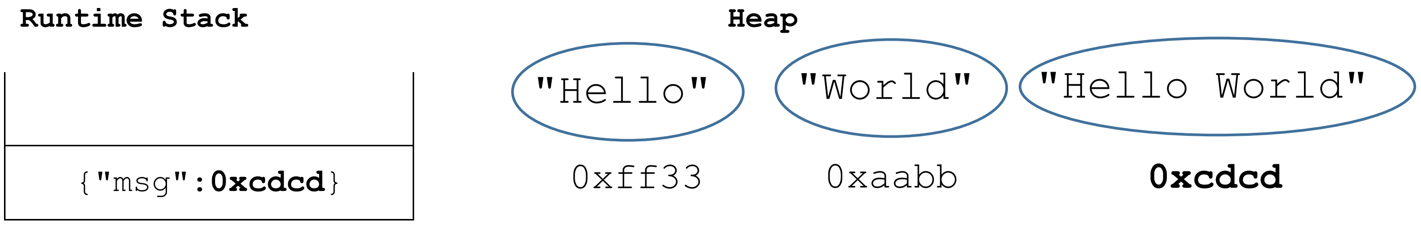
```
String msg = "Hello";
```



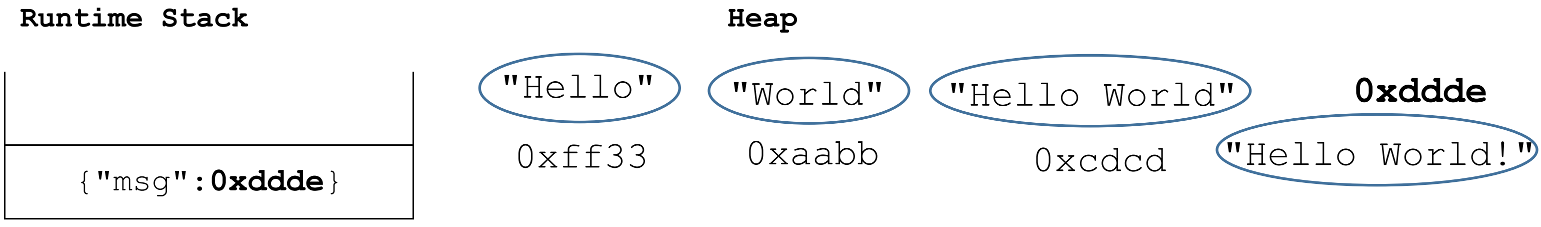
```
msg = "World";
```



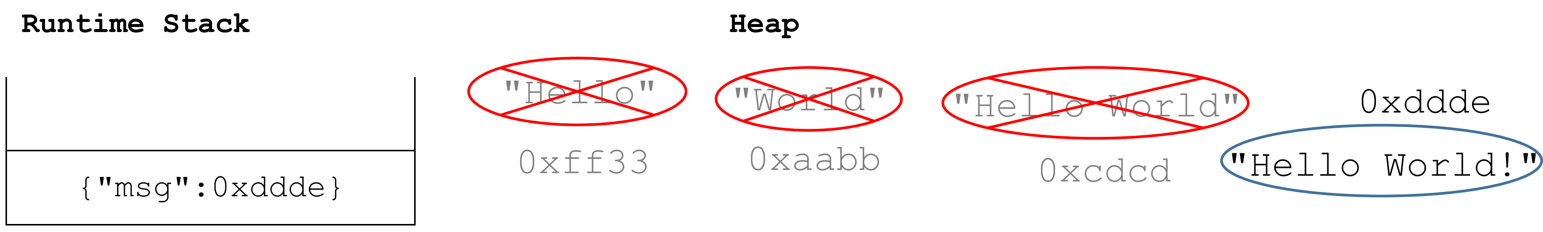
```
msg = "Hello " + msg;
```



```
msg += "!";
```



Every time the string changes, new memory is created. It is then up to the garbage collector to collect that memory:



This is very wasteful with memory. Even though the Garbage Collector reclaims this memory, it is still being used until the GC runs. So memory still takes a temporary hit. It is also inefficient since we are giving the GC more work to do. This is why code such as the following is considered bad practice:

```
String[] array = { ... }; // a large array of strings
String composite = "";
for (String item : array) {
  // creates a new String object every time
  // and has to copy over all old characters
  composite += item;
}
System.out.println(composite);
```

It is far better to use Java’s built in “StringBuilder” class:

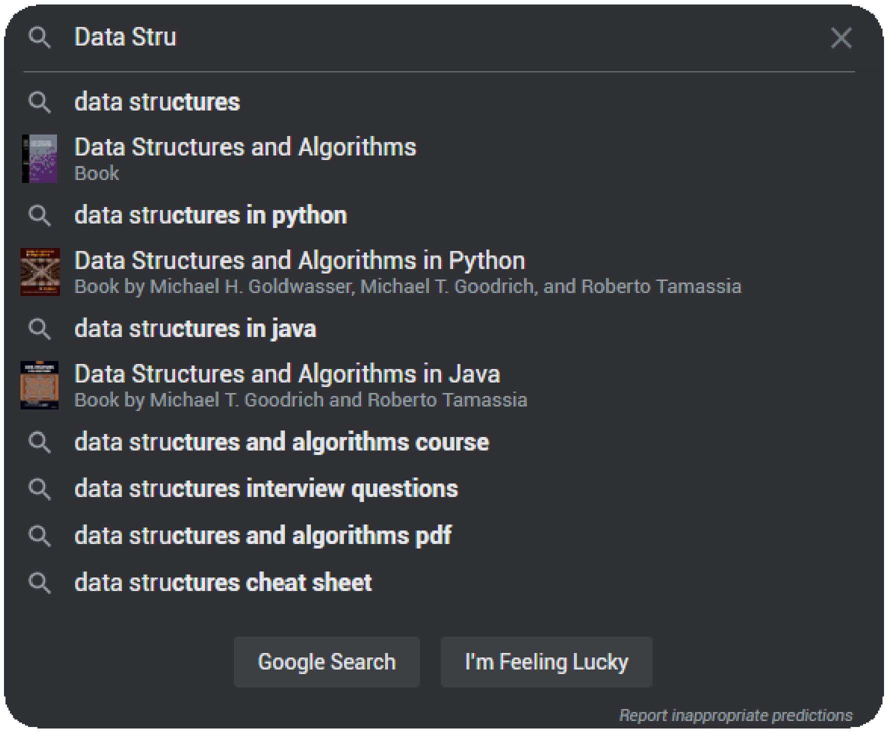
```
StringBuilder sb = new StringBuilder();
String[] array = { ... }; // a large array of strings
for (String item : array) {
  // stores item into a hidden string array (inside of sb)
  // this is no more expensive than simple assignment
  // as it just copies item from one array to another
  sb.append(item);
}
// this part builds the final string so the big cost
// is really only incurred once
System.out.println(sb);
```

StringBuilder has a char array inside of it. When you call “append”, it just adds the characters from the string to the next empty consecutive spots in the array. If the array is full, you will pay the cost of doubling the array (i.e. “table doubling”). With using the String class in the code above, you pay that cost EVERY TIME you concatenate. With StringBuilder, you pay the cost only when the inner array is full. The real cost comes in when you call the toString() method on a StringBuilder object. This is where the final String is allocated (with enough size to fit everything in it), then characters are copied over one at a time. This occurs in the final line, i.e. “System.out.println(sb):”. Recall that printing an object automatically calls its toString() method. So while we could have written “System.out.println(sb.toString());”, this would be redundant, as the println method already calls toString for us.

Note that this does NOT mean you should use StringBuilder EVERY time you want to concatenate strings. If you only have a few strings, or even a few dozen, you are probably just fine using a String object. I doubt the difference would be noticeable and if you only have a couple strings to concatenate, you may actually pay more to use StringBuilder! But when you have a lot of concatenation to do, you should absolutely use StringBuilder (or Java's StringBuffer class). There is also a StringBuilder class in C#, a StringIO class in Python (also you can use the 'join' method), and stringstream in C++.

Hopefully this simple example shows that strings are more complicated than you may have thought. There are a lot of considerations to be made for efficiency and a lot of complicated operations that are tricky to be done unless you have a more sophisticated way of storing the data. Let's look at the motivation behind the data structure we will be learning about.

Ever wonder how Google can complete you search query before you even completely type it?



Data Stru

data structures

Data Structures and Algorithms
Book

data structures in python

Data Structures and Algorithms in Python
Book by Michael H. Goldwasser, Michael T. Goodrich, and Roberto Tamassia

data structures in java

Data Structures and Algorithms in Java
Book by Michael T. Goodrich and Roberto Tamassia

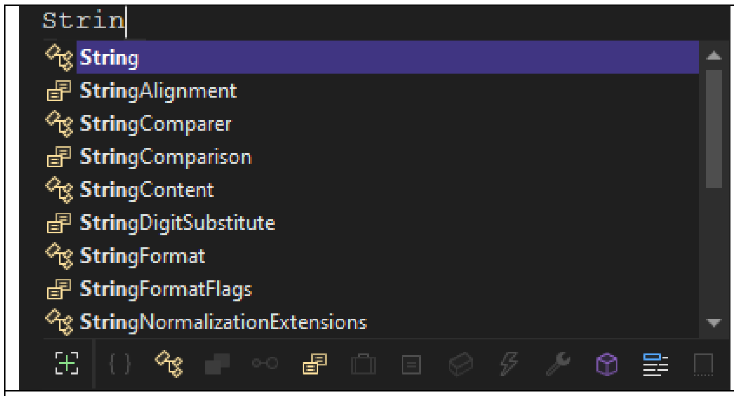
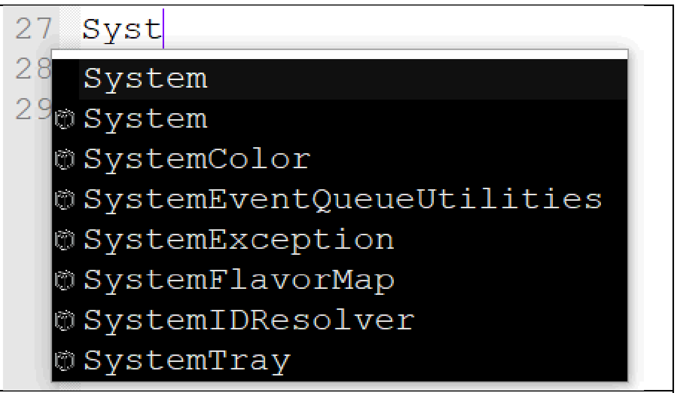
data structures and algorithms course

data structures interview questions

data structures and algorithms pdf

data structures cheat sheet

Or how about in your favorite IDE or text editor:

	
Visual Studio autocompleting C#	Notepad++ autocompleting Java

```
Strin
String
StringAlignment
StringComparer
StringComparison
StringContent
StringDigitSubstitute
StringFormat
StringFormatFlags
StringNormalizationExtensions
```

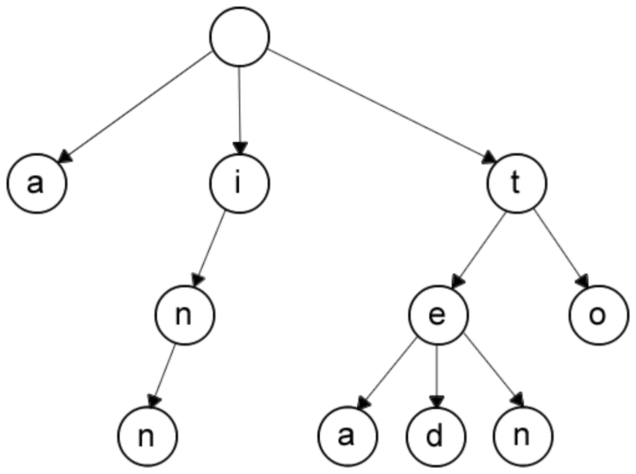
```
Syst
System
System
SystemColor
SystemEventQueueUtilities
SystemException
SystemFlavorMap
SystemIDResolver
SystemTray
```

Considering the high number of possibilities in each case, we need a highly efficient data structure to search those possibilities and return the ones that match the characters you have typed so far. The “trie” data structure can help us. Note that trie is pronounced like “try” and NOT like “tree” to avoid any confusion.

Abstract Data Type: Trie

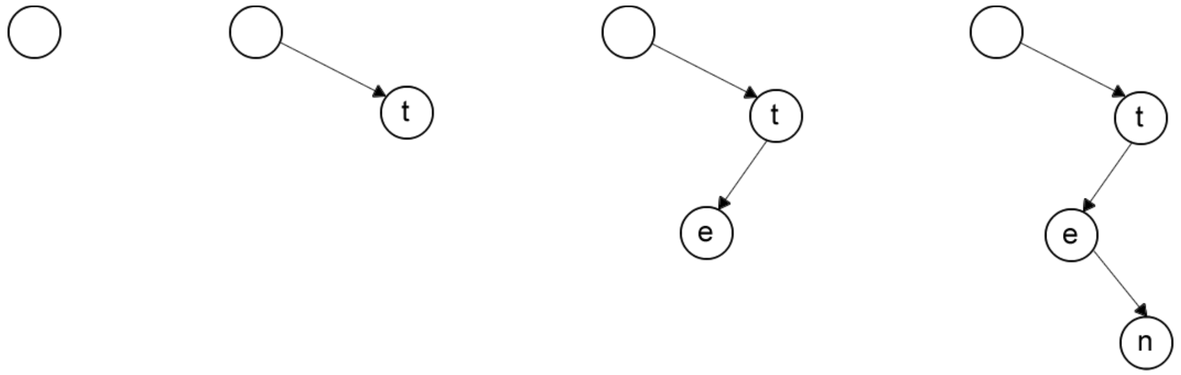
`insert(word)`	Inserts the given string into the trie
`search(word)`	Searches for the given string in the trie. Returns true if found, false otherwise
`printAllWords()`	Prints all words that have been added to the trie
`printAllWords(prefix)`	Prints all words that start with the given string prefix

A trie is a tree data structure that stores characters in its nodes. Here is a visual example:

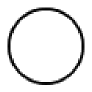


Notice a few things. One is that this is NOT a “binary tree”, i.e. there are clearly nodes with more than 2 children. Secondly, the root doesn’t have any data in it. This is not a mistake, this is by design. In fact, implementation wise the nodes don’t actually have data inside of them either! But we will get to that later. The trie above actually stores 6 words. Those words are: “a”, "inn", “tea”, “ted”, “ten”, and “to”. See if you can look at the data structure and see how these words are stored. Notice that they are in alphabetical order when listed out. This is also not a mistake.

Words are inserted into the trie one letter at a time. Let’s look at the insert method and start with an empty trie. Inserting the word “ten” looks like this:

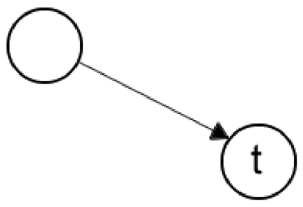


Each node has an array of possible child nodes, all set to null initially. The array represents a mapping from a character to a child node. So, for example, if we restricted our words to just contain lowercase letters, the child node array in the root for an empty tree would look like this:



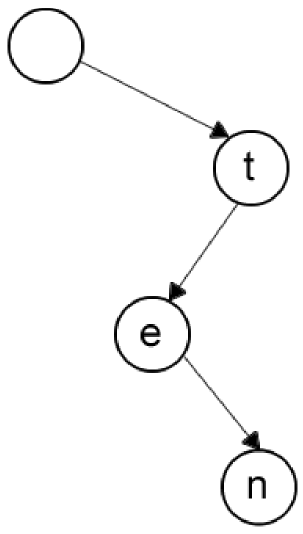
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

After the first letter is inserted, “t”, the root’s children look like this:



a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	node	Ø	Ø	Ø	Ø	Ø	Ø

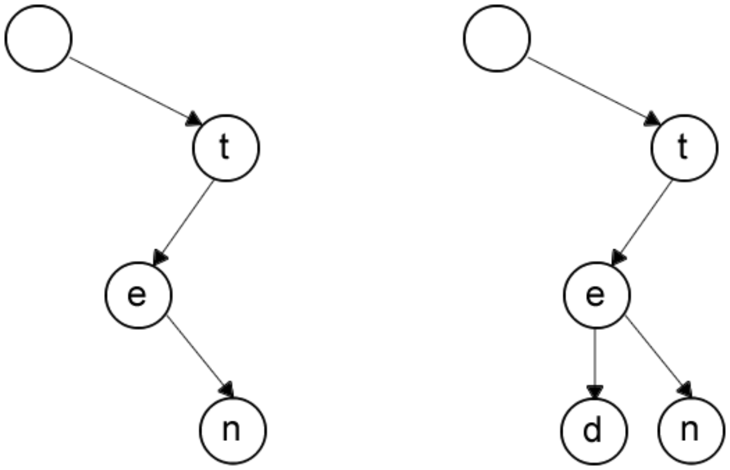
After inserting all letters, this is what the array of children look like for each node:



Root	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	node	Ø	Ø	Ø	Ø	Ø	Ø
t	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
	Ø	Ø	Ø	Ø	node	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
e	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	node	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
n	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

The nodes themselves only need to contain an array of child node pointers. If we imagine that ‘a’ is 0, ‘b’ is 1, ‘c’ is 2, and so on, then we can easily see how characters can map to indices (we could also use ASCII values, but we would need a bigger array). If an index points to null, then there is no child for that character. If an index points to a node, then there is a child for that character and we can jump to that child node easily. We can then examine that child node to see which characters it contains as children. For this reason, a node really does NOT have to contain the character it represents, as the path to get there tells us this already. For this reason, the root doesn’t really represent a character.

Let’s now insert the word “ted”. Since we already have “t” and “te”, we can go down that path and only add the “d”.

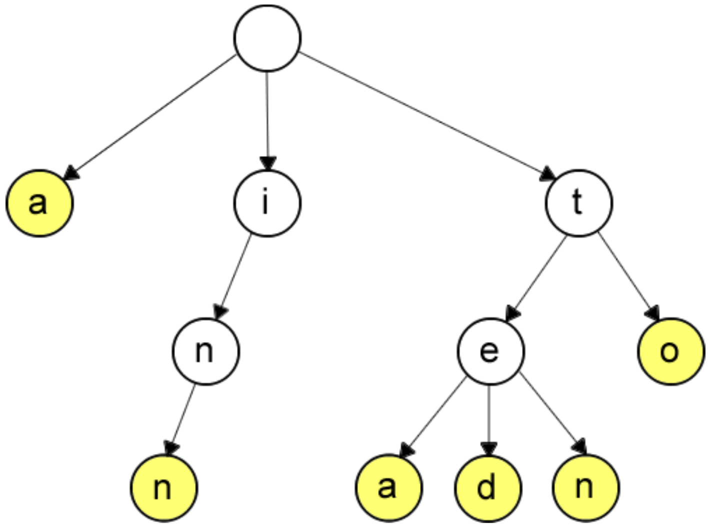


Now the array of children looks like this for each node:

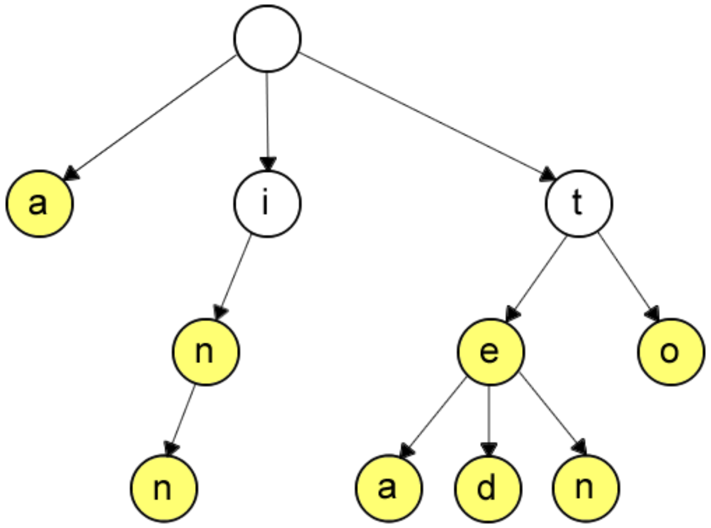
Root	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	node	Ø	Ø	Ø	Ø	Ø	Ø
																										dragonballbonus
t	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
	Ø	Ø	Ø	Ø	node	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
e	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
	Ø	Ø	Ø	node	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	node	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
d	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
n	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

There is one more piece of information we need to store in each node. How do we know that “ted” and “ten” were inserted but not “te”? Sure you may say that “te” is not a real word, but neither are some of the functions built into a language, yet auto-complete in your favorite text editor still needs to be able to show you those possibilities. So, given that “te” could have theoretically been inserted into this tree, without changing the structure of the tree, how do we then know that “te” was or wasn’t a word that was inserted?

The answer is quite simple. We just add a boolean to each node that tell us if this is a “terminal” node or not. If a node is terminal, that means it is the last letter in a completed word. It may also be within a longer word, but then the last character of that word would also be a terminal node. Let me show you the complete tree again, but this time highlight the terminal nodes:



We can see now that, for example, the word “inn” is in this tree but it does not recognize the word “in”. It is the case that all leaf nodes are, by definition, a terminal node, since a word would have had to have been inserted that pushed the tree that far down and ended on the final character. However, inner nodes may be terminal as well. Let’s take this same tree and add the words “in” and “te” to it. This is what it becomes:



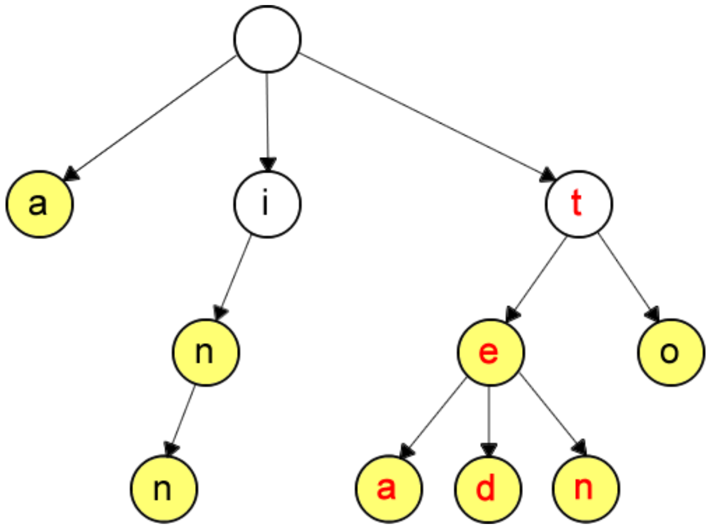
The structure doesn’t change at all since those letters in that order were already added. However the node containing “e” is now terminal since “te” is now a word and the node containing the first “n” is now terminal since “in” is now a word. We still don’t recognize “t” and “i” alone as words since they are not marked as terminal nodes.

One thing to note about tries is that they are horrible with efficient memory usage. If the alphabet contains only the lowercase letters from ‘a’ to ‘z’, this would mean that each node contains an array of size 26 (which are possibly mostly null). While this is in fact linear (i.e. O(n)) in terms of storage, the constant term is quite high, i.e. 26. This leads to a lot of memory overhead. If the alphabet included uppercase letters as well, now each node would contain an array of 52 elements. If we wanted to include the entire standard ASCII set (i.e. letters, numbers, and symbols), then each node would have an array of size 127! Your programming assignment has an alphabet of this size and you will see what I mean when you view the memory usage (refer to the assignment for more information on that).

One thing we could do, is to use a dictionary (that is implemented as a hash table) to store the child pointers instead. This would mean that, for the most part, we only pay the cost of storage for the child nodes that we actually have, and we would have far less empty slots. The problem here is that we would lose one of the benefits of the trie, that being that words are stored in alphabetical order. If we traverse the trie using an inorder traversal and print out all the words (by checking if each node is terminal or not), they will be printed out in order. This is essentially the `printAllWords` operation above. This is because we would visit the children in the order they are stored in the array, and naturally 'a' would come before 'b' in the array. However, if we use a hash table, which uses an entirely different system to determine where a value is stored, we would completely lose that ability. Sure we could sort all the words after collecting them, but considering how that would take linearithmic time (i.e. O(n lg n)) at best, this isn't an efficient solution. So, in the end, is order important to you? Then you will need to store the nodes in a normal array and take the hit on memory. If order is not important to you, then you can use a hash table and conserve memory. Tradeoffs!

How searching for a given word works should be decently obvious. We traverse the trie from the root, looking at the children array and see if the first character in the word points to a valid node. If we find it pointing to null, then the word isn't in our trie. If we find it pointing to a node, then we search that node's children for the next character in the word. Once we arrive at the final node, we examine whether it is a terminal node or not. If it is, then we found the word. If it isn't, we report that the word isn't in our trie.

The true power of our trie comes into play when we use the `printAllWords(prefix)` operation. You may see this operation called something different online (such as `startsWith` or `beginsWith`), but the function is essentially the same. We need to find all words that start with the given prefix. For example, if we call `printAllWords("te")` on our trie, we would get the following result:



To carry out this operation, first we check the root and find that 't' does map to a real node (and not null). We then check that node's children and see that 'e' also maps to a node. We traverse the tree, examining all paths from node 'e' and grabbing each word that is formed by landing on a terminal node. We arrive at "te", "tea", "ted", and "ten" as our words (remember that we added the word "te" on page 7). Note that since "e" itself is marked as a terminal node, we include the prefix “te” in our list. Now you should be able to see how your favorite IDE or text editor can suggest things to you based on what you have already typed.

References:

- https://lemire.me/blog/2017/07/07/are-your-strings-immutable/
- https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html
- https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html
- https://www.w3schools.com/python/ref_string_join.asp
- https://www.youtube.com/watch?v=3CbFFVHQrk4