






# Create a Trie Search Tree

Here we will move on from binary search trees and take a look at another type of tree structure called a trie. A trie is an ordered search tree commonly used to hold strings, or more generically associative arrays or dynamic datasets in which the keys are strings. They are very good at storing sets of data when many keys will have overlapping prefixes, for example, all the words in a dictionary. Unlike a binary tree, nodes are not associated with actual values. Instead, the path to a node represents a specific key. For instance, if we wanted to store the string code in a trie, we would have four nodes, one for each letter: c — o — d — e. Following that path through all these nodes will then create code as a string — that path is the key we stored. Then, if we wanted to add the string coding, it would share the first three nodes of code before branching away after the d. In this way, large datasets can be stored very compactly. In addition, search can be very quick because it is effectively limited to the length of the string you are storing. Furthermore, unlike binary trees a node can store any number of child nodes. As you might have guessed from the above example, some metadata is commonly stored at nodes that hold the end of a key so that on later traversals that key can still be retrieved. For instance, if we added codes in our example above we would need some way to know that the e in code represents the end of a key that was previously entered. Otherwise, this information would effectively be lost when we add codes.

---

Let's create a trie to store words. It will accept words through an `add` method and store these in a trie data structure. It will also allow us to query if a given string is a word with an `isWord` method, and retrieve all the words entered into the trie with a `print` method. `isWord` should return a boolean value and `print` should return an array of all these words as string values. In order for us to verify that this data structure is implemented correctly, we've provided a `Node` structure for each node in the tree. Each node will be an object with a `keys` property which is a JavaScript Map object. This will hold the individual letters that are valid keys of each node. We've also created an `end` property on the nodes that can be set to `true` if the node represents the termination of a word.

---

	The <code>Trie</code> should have an <code>add</code> method.
	The <code>Trie</code> should have an <code>print</code> method.
	The <code>Trie</code> should have an <code>isWord</code> method.
	The <code>print</code> method should return all items added to the trie as strings in array.
	The <code>isWord</code> method should return <code>true</code> only for words added to the trie and <code>false</code> for all other words.