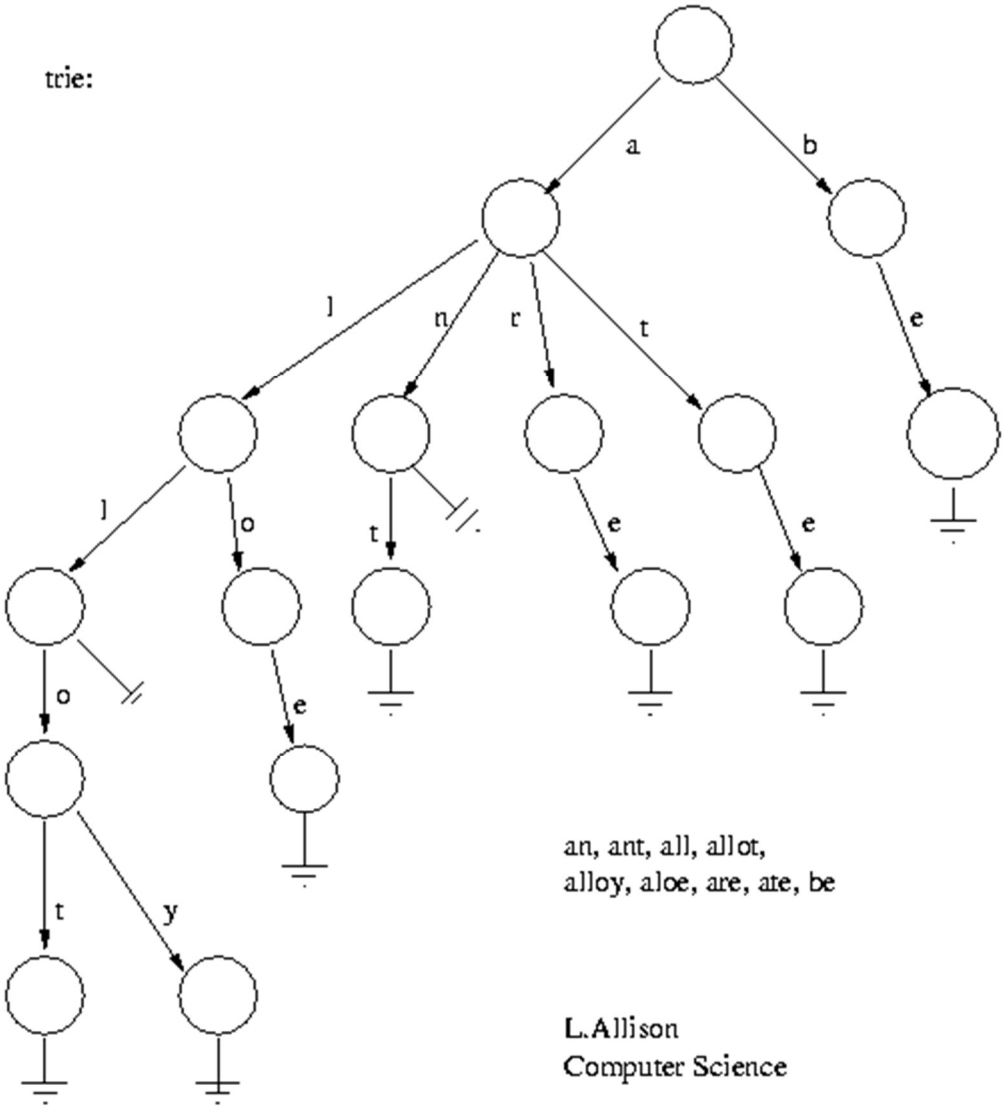
Kyle Lund

Tries

At a high-level, a trie is a tree-like data structure that allows the user to store and retrieve sequences of the contained data. More specifically, tries are N-ary trees where the children of each node represent valid additions to the previously constructed sequence. The most common use case of a trie stores the letters of the alphabet. In this case, the various paths through the trie would correspond to valid words that this trie has stored. One important characteristic is that the tree must store some sort of terminating node, a specific value or flag that is stored in a child node which tells the data structure that the path taken to get to this node is a valid output.



This example is taken from this article. In this particular trie, all words in the stored dictionary start with either an ‘a’ or a ‘b’, as the root node only has these two letters as children. To examine how a search operation works, we’ll start with the word ‘be’. Starting at the root, we search for the first letter in our target word, ‘b’ in this case. This letter is a child of the root node, so we proceed to that node and the next letter in the target word. At the ‘b’ node, we then search for the letter ‘e’, once again finding it.

If there were more letters in the target word, we would proceed in this manner until we reach the end of the word. Once the end of the word is reached, there is one final step. We must check that the terminator character or flag is a child of the last node we visited. In this example, the flag is represented by a ground symbol from electrical engineering. This flag is a node of the ‘e’ node, so in this example we would return true.

Now let’s search for the word ‘bear’. Just as before, we successfully find the letters ‘b’ at the root node, and that an ‘e’ node is a child of the ‘b’ node. At this point we would search for an ‘a’ node that is a child of the ‘e’ node. In this tree, that node is not found, and we would return false. As an aside, this operation seems to call for a recursive implementation. At the root node, we are searching for ‘bear’. This transitions us to the ‘b’ node, where we are effectively searching for ‘ear’. With each call of the search function, we simply drop the first character in the target word and continue searching from the last found node.

Pivoting to the insertion operation, this will function very similarly to the search function. For this example, as we failed to find the word ‘bear’ last time (and because bears are incredible creatures), we will insert ‘bear’ into this trie. Once again, starting at the root node, we search for the first character in the target word, and in this case finding it. As the ‘b’ node already exists in the trie, we will transition to that node and continue with inserting the rest of the target word. In this case, we once again find the ‘e’ node and proceed to that node. Now, we check if the ‘e’ node has a ‘a’ node as a child. In the current trie, it does not. In this situation, we insert an ‘a’ node as a child of the ‘e’ node and move to that node to continue the operation. At this new ‘a’ node, we continue the process. Searching for an ‘r’ node, not finding it, creating it, and moving to it. And finally at the end of the word we have one final step. We insert the termination flag node as a child of the last node in the target node. And with this, our word is inserted, and subsequent searches for this word will return true.

Wrapping things up, under the hood a trie is often constructed as an array of pointers. These pointers then point to other arrays of pointers. Some advantages of a trie are that input, search, and deletion all run in O(n) time, when n is the length of the word. A trie may require less space than a corresponding BST, as it does not store the keys explicitly. A hash function is also not required for a trie, and unlike a hash table, the stored contents of a trie can be retrieved in alphabetical order and can be iterated through. The primary disadvantage is that the underlying arrays take up a lot of space, and most of that space will consist of null pointers. Some of the common use cases for tries include storing browser history, autocomplete, and spell checkers. Tries are also well suited for longest prefix matching applications.