Coding Interview: Compressed Tries

Tries are data structures which can be used to implement symbol tables, much like hash tables can. If we define the alphabet for valid keys to be Σ, then a key is just a string on $s \in \Sigma^*$. As all symbol tables, tries must support the usual operations; e.g., in Python:

```
class Trie(Generic[V]):
    def search[V](self, key: str) -> Optional[V]:
        ...
    def insert[V](self, key: str, value: V):
        ...
    def delete[V](self, key: str):
        ...
```

Regular tries are m-ary trees which store prefixes in internal nodes and content on both leaves and internal nodes[1]. Each node can have as many children as there are letters in Σ; i.e., $|\Sigma| = m$, and each edge corresponds to a character in Σ. For instance, calling the sequence:

```
trie = Trie[int]()

trie.insert('carbonated', 1)
trie.insert('carbonic', 3)
trie.insert('carbide', 5)
trie.insert('car', 7)
trie.insert('bible', 9)
trie.insert('biblical', 11)
```

would produce a trie that looks like the one in Fig. 1. Nodes that contain values are highlighted in blue, with the value on the side.

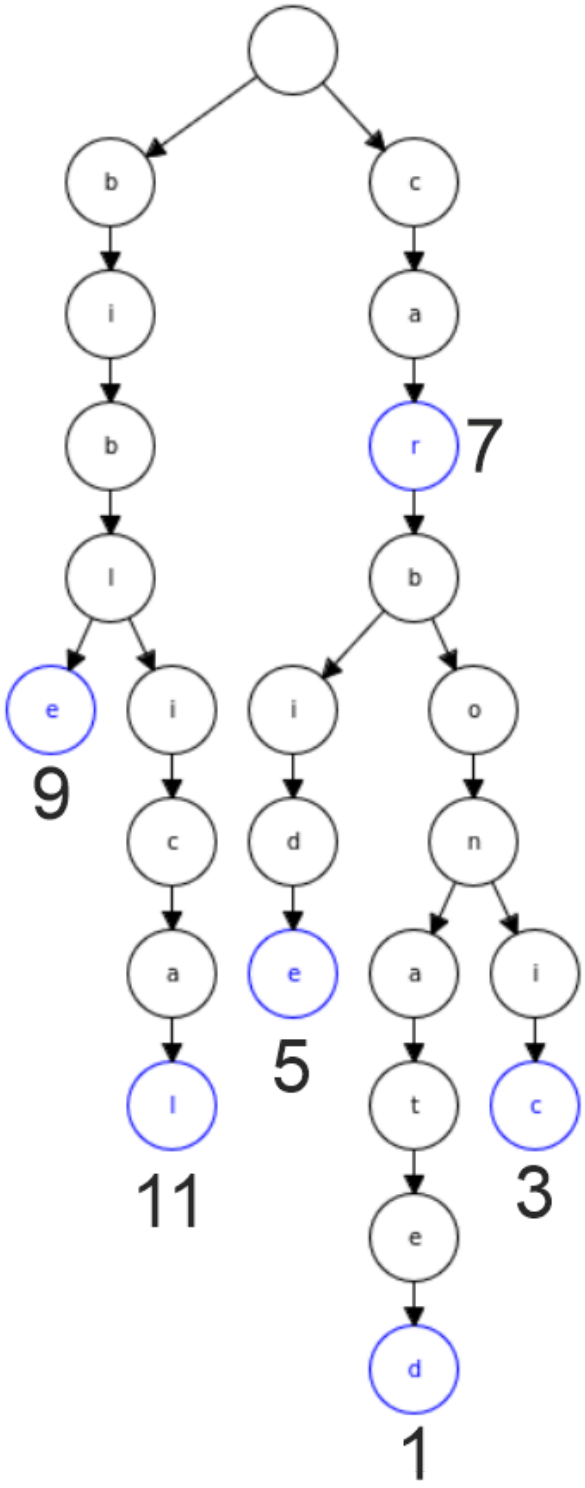

Figure 1. A regular trie.

Performing a search amounts to following the path corresponding to each letter in the key and testing: *i)* if we can get to the end of the key before bumping into a leaf, meaning the key is not in the trie and; *ii)* the node where our search ends contains a value. If it does not, then the key is not in the trie.

When we perform a search for, say, "car", we follow a path from the root through the link labelled `c`, then through the link labelled `a`, then through the link labelled `r`. The node pointed to by this last link contains the value 7, so we return it.

Tries are space-inefficient: unique prefixes are represented by long paths formed by single nodes without values (e.g. "carb" and "bibl" in Fig. 1), so we would like to implement what is known as *compressed* trie, and sometimes a PATRICIA trie[2]. In compressed tries, we compact all single-child nodes with no values into a prefix within the node itself. A compressed trie equivalent to the one in Fig. 1 would look like:

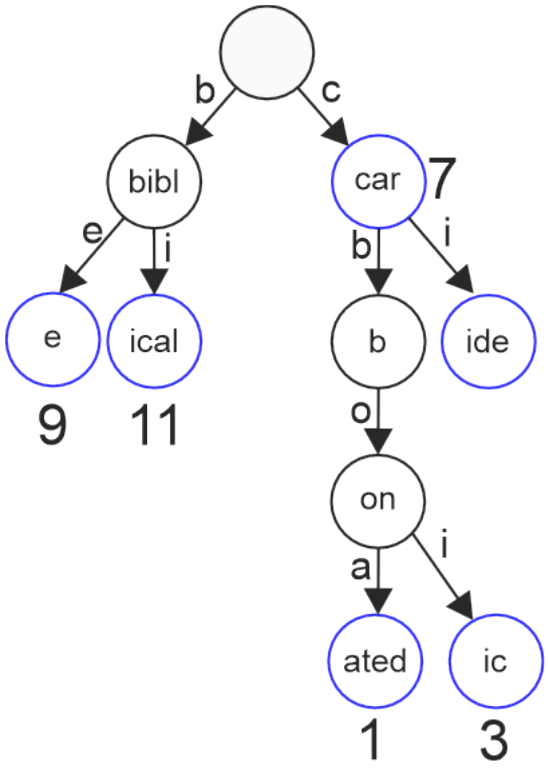

Figure 2. A compressed trie.

Note that edges are labeled by path as before, and nodes can still have as many children as there are letters in Σ, but one-way paths are compressed into string prefixes within the node itself. Your job will be to implement `insert` and `search` on a compressed trie, using the language of your choosing.

Example input:

```
trie = CompressedTrie()

trie.insert('carbonated', 2)
trie.insert('carbonic', 3)
trie.insert('carbide', 4)
trie.insert('car', 1)
trie.insert('bible', 5)
trie.insert('biblical', 6)

print(str(trie))
```

example output:

```
<<: None>>
  <<car: 1>>
    <<b: None>>
      <<ide: 4>>
      <<on: None>>
        <<ic: 3>>
        <<ated: 2>>
  <<bibl: None>>
    <<ical: 6>>
    <<e: 5>>
```

1. Depending on how you implement them - if you choose to a special string termination character like C strings, then you do not need to store values in internal nodes. ↩

2. Though PATRICIA tries are originally binary radix trees, the term is nowadays often used in reference to any type of compressed trie. ↩