Tries

Tries are another data structure that we can store values in, indexed by keys of a particular data type. In the value-storing data structures we’ve discussed previously, we’ve assumed that storage of keys and values keeps everything completely distinct; every key-value pair is stored separately from every other one. In tries, however, we have keys overlapping at common prefixes. Our keys are assumed to be a sequence of elements with a defined ordering. We access a stored value by traversing a key chain and getting the value stored in the node containing the key chain element.

Here we have an example that uses strings as keys. They’re a sequence of characters (char *) subject to ascii-betical ordering. We use booleans as our stored values (represented as checks for true and lack of checks for false).

This is what we’d call a Ternary Search Trie. Each trie node is a ternary tree containing a character c with pointers to a left, middle, and right subtrie. The left subtrie stores all words starting with characters alphabetically less than c. The right subtree stores all words starting with characters alphabetically greater than c and the middle stores a subtrie with all words starting with c, from the second character on.

Can you read the secret message stored here? Do you like tries? No? Well trust me, my tries are truly awesome.

Another representation of tries is the Multiway Trie, where sequence-keys are still spread out across multiple levels. The difference, though, is that at each level, we have an array for each possible alphabet letter to be expected, and the associated value would be at each array. The following trie contains the strings tries, are, and awesome, as keys, each mapped to true. Which implementation do you like better? What are the advantages and disadvantages of using either one?
Experimental Interface Happy Fun Time

In class, we came up with an interface to deal implement and work with tries. Here is what we came up with:

```c
trie trie_new(void (*elem_free)(void));
void trie_insert(trie T, key k, elem e);
elem trie_lookup(trie T, key k);
void trie_remove(trie T, key k);
void trie_free(trie T);
```

For practice, try implementing your own version. `elem` should be polymorphic, so use `void *`. 