15-122: Principles of Imperative Computation

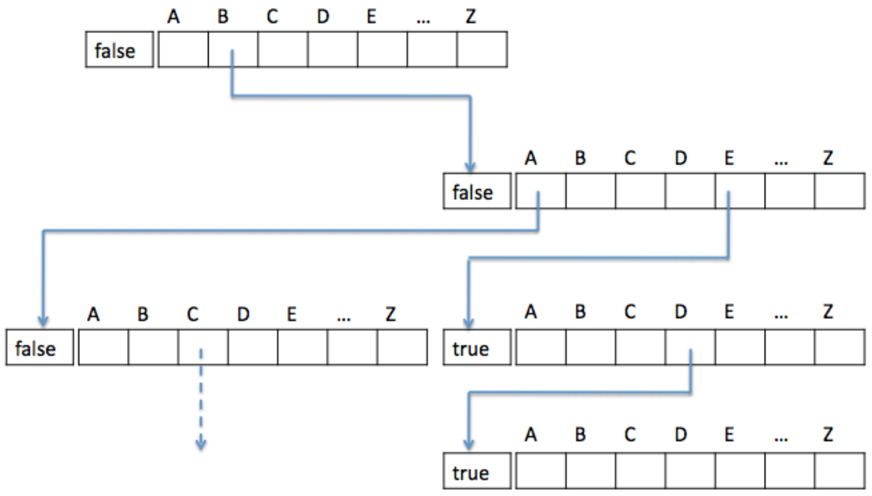
Recitation 21b — Josh Zimmerman

Trie, trie again

Let's revisit the idea of tries, since I'm sure a lot of you were preoccupied by the midterm when we went over them on Thursday.

The basic idea of a trie is that for some purposes, like spell checking and boggle, it often makes sense to see if a string is the *prefix* of some word in a dictionary. (That way, we can more quickly realize that we can stop checking some sequence of letters in boggle, and more quickly flag that a word is misspelled)

To do this we keep a tree that contains letters representing prefixes. This is a bit hard to describe, so we'll use this diagram. (The diagram is an example of a *multi-way* trie, which is one implementation of the basic idea of looking up prefixes of strings.)



This multi-way trie contains the words "be", "bed", and the start of "baccalaureate". Image credit: Frank Pfenning.

The boolean flags in the trie are true if and only if that node in the trie is the end of a word. (It's useful to know when we've reached the end of a word so we can know that, for instance, "b" and "bacca" are not words.)

This approach is great, since it lets us look up whether a word of length k is in the trie in $O(k)$ time.

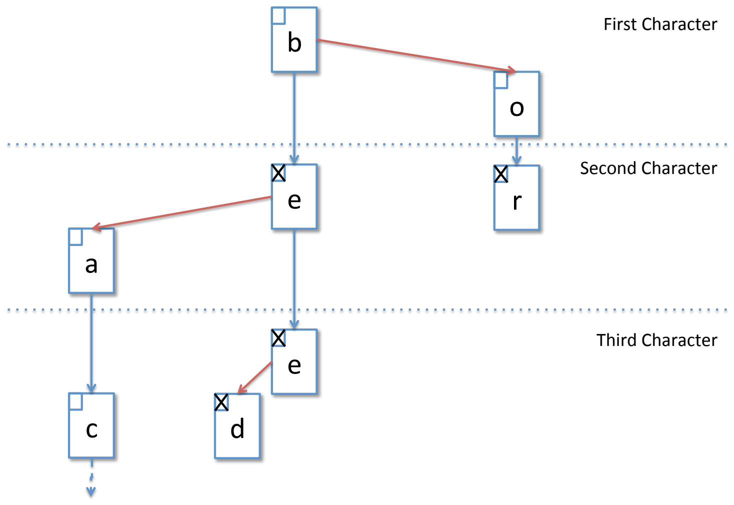
However, it has some major problems. In particular, we'll be wasting a *lot* of space, since we have an array of 26 elements at each level, of which we'll only need a few the majority of the time.

We don't have unlimited memory, so this is bad.

So, we'll use a slightly more complicated but far smaller structure, a *ternary search trie*.

In a ternary search trie node, we'll store a character c and three pointers, `left`, `middle`, and `right`. Similarly to a binary search tree, the left subtree stores words starting with characters alphabetically less than c and the right subtree stores words starting with characters alphabetically greater than c. `middle` stores a subtrie with all words that start with c.

Here's another image, again from lecture notes. It contains the word "be", "bee", "bed", "baccalaureate" (we only show the first few letters in the diagram to save space, but if we were implementing a trie we'd store the whole word), and "or". Here, we represent the end of a word with an X.



Trie interface?

This is the interface of tries as we defined them in class.

```
typedef struct trie_header *trie;
trie trie_new();
void trie_free(trie TR);

/* Dictionary interface */
void trie_insert(trie TR, char *s, elem e); /* strlen(s) > 0 */
elem trie_lookup(trie TR, char *s); /* strlen(s) > 0 */

/* Prefix search interface */
typedef struct trie_node tnode;
tnode *trie_lookup_prefix(trie T, char *s);
elem tnode_elem(tnode *T);       /* T != NULL */
```

This interface is a little confusing, so let's go over exactly what it does for us.

The basic functions we have are to create a trie and free it. Importantly, by looking at the source code for `trie_free`, we can see that it doesn't free any of the `elem`s. (The reason for this is that it doesn't know what the `elem`s are, or how to free them. We'll talk about a solution to that problem soon, when we introduce the idea of a *function pointer*.)

```
void tnode_free(tnode *T) {
  REQUIRES(is_tnode_root(T));
  if (T == NULL) return;
  tnode_free(T->left);
  tnode_free(T->middle);
  tnode_free(T->right);
  free(T);
}
void trie_free(trie TR) {
  REQUIRES(is_trie(TR));
  tnode_free(TR->root);
  free(TR);
}
```

Some other important interface functions we have are `trie_insert` and `trie_lookup`. These allow us to simply use the trie as an implementation of an associative array.

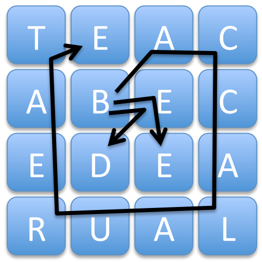
The real benefit of using a trie over a hash table or a BST, is to be able to use prefixes to look things up, so we need functions to do that too.

Accordingly, we have `trie_lookup_prefix` and `tnode_elem`. `trie_lookup_prefix` returns a `tnode` (which is a pointer to some part of the trie) if the prefix was found in the trie, or NULL otherwise.

We can use `tnode_elem` to convert from a `tnode` to an `elem`. This function returns `NULL` if there's no `elem` at this spot in the trie and otherwise returns the `elem` that was stored at this location in the trie.

Boggle

Boggle is a word game that we can use a trie to solve. The basic idea is that we have a four-by-four grid of random letters. We try to make words out of them by drawing a line through letters.



An example boggle board. The lines show words.

We can use the trie interface to help us with this. Let's talk about how, and how we might improve the interface to allow us to find all words in a boggle board as quickly as possible.