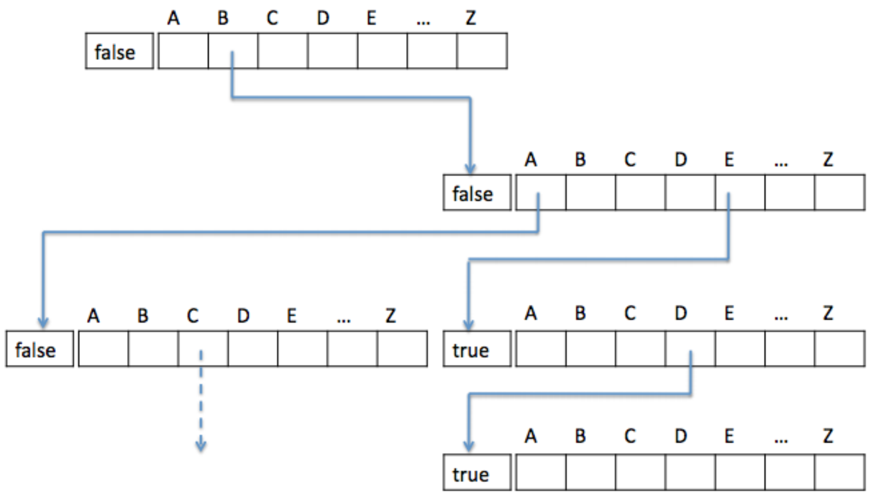
15-122: Principles of Imperative Computation

Recitation 23 **Matt Dee, Josh Zimmerman**

Trie-ing is Cool!

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 realize that we can stop checking some sequence of letters in boggle, and more quickly flag that a word is misspelled, without checking every word)

To do this we keep a tree that contains letters representing prefixes. Each sub-trie is the of all strings in the original trie that started with the letter of the parent. The diagram below shows an example of this.(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, but are prefixes for other 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 (the size of the dictionary does not matter for runtime!).

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. And thats just for lowecase! If we tried to make a trie for all ASCII characters (128 chars), we would be in real trouble, and Unicode is out of the question (>230,000 chars)!

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

So, we'll use a slightly more complicated but far smaller structure, whose size does not depend so much on the size of the dictionary, 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 ASCIIbetically less than c and the right subtree stores words starting with characters ASCIIbetically 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 same words as the image above, with the addition of "or", but is represented as a ternary search trie. Here, we represent the end of a word with an X.

../../lectures/22-tries-diagrams/tries5.pdf

Interface

Note that we now have two possible implementations for tries. Luckily, as the client, we don't have to care about the implementation, as they can both share a interface! This is the interface of tries as we defined them in class.

```
typedef _____________ *trie; // Implementation decides what type this is
trie trie_new();
void trie_free(trie TR);

/* Dictionary interface */
/* These allow us to use tries as a associative array */
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 - the magic of tries! */
typedef ____________ tnode; // Implementation decides what type this is
tnode *trie_lookup_prefix(trie T, char *s);
elem tnode_elem(tnode *T);       /* T != NULL */
```

`trie_insert` and `trie_lookup` allow us to use the trie as an associative array, like a hash table or BST. The real benefit of using a trie over one of these, is to be able to use prefixes to look things up and potentially short-circuit a search, 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.

This `tnode` can use the `tnode_elem` function to get the elem at that point, or `NULL`if there is none (it is just a prefix, not a word).

Implementation

We will represent these using the following structs, using a structure very reminiscent of a BST:

```
typedef struct tst_node tnode;
struct tst_node {
   char c; /* discriminating character */
   elem data; /* possible data element (void pointer) */
   tnode *left; /* left child in bst */
   tnode *middle; /* subtrie following c */
   tnode *right; /* right child in bst */
};
typedef struct trie_header trie
struct trie_header {
   tnode *root;
};
```

Probably the most important functions of the trie are the `trie_lookup` functions and `trie_lookup_prefix`, so we will look at the code to understand how lookups work, and hopefully get a better idea of how data in tries is organized. Both of these functions share some logic, so we will write a helper function to get the `tnode` for a string.

```
/* Returns the tnode for the given string, starting at character i */
tnode *tnode_lookup(tnode *T, char *s, size_t i)
{
   REQUIRES(is_tnode_root(T));
   REQUIRES(s != NULL);
   REQUIRES(i < strlen(s));
   if (T == NULL) return NULL;
   // If the character we are looking for is less than this node, look left
   if (s[i] < T->c) return tnode_lookup(T->left, s, i);
   // if the character is greater, look right
   if (s[i] > T->c) return tnode_lookup(T->right, s, i);
   ASSERT(s[i] == T->c);
   // We have found the character!
   if (s[i+1] == '\0') return T; //We are at the end, return this node
   return tnode_lookup(T->middle, s, i+1); //Continue the search on the next character
}
/* Returns a tnode* if it is a valid prefix, NULL otherwise */
tnode *trie_lookup_prefix(trie *TR, char *s) {
   REQUIRES(is_trie(TR));
   REQUIRES(s != NULL);
   return tnode_lookup(TR->root, s, 0);
}
/* Returns the elem associated with s, if it exists */
elem trie_lookup(trie TR, char *s)
{
   REQUIRES(is_trie(TR));
   REQUIRES(s != NULL);
   tnode *T = trie_lookup_prefix(TR, s);
   if (T == NULL) // This isn't even a valid prefix...
      return NULL;
   else // Return the element there (may be NULL if s is a prefix but not a key)
      return tnode_elem(T);
}
```

Invariants

Tries, like any good data structure, have invariants that we need to ensure hold true. Specifically, lets look at the `is_tnode` function for ternary search tries. Because `NULL` represents the empty trie, this is a valid `tnode`. For any non-null trie, the trie invariants are:

- The `left` and `right` fields act as a BST. In other words, for any `tnode T`, IGNORING `middle` pointers, every `tnode` in the `T->left` subtree represents a character whose ASCII value is less than `T->c`, and every `tnode` in the `T->right` subtree represents a character whose ASCII value is greater than `T->c`.
- For any `tnode T`, `T->middle` is a valid `tnode`.
- If, for some `tnode T`, `T->middle == NULL`, then `T->elem` must not be `NULL` (otherwise this trie would not be a prefix for any element!).

To check this, we use mutually recursive functions. This refers to two functions, which each call each other. Note that in order to do this, we have to forward-declare the second function so that the first function won't cause a compiler error when it calls the second.

```c
// We need this declaration so that is_tnode knows about is_tnode_root
bool is_tnode(tnode *T, int lower, int upper);
bool is_tnode_root(tnode *T);

// Ensure that, each node is a valid tnode, and that every character on this level
// (without taking a middle) is less than lower and greater than upper
bool is_tnode(tnode *T, int lower, int upper) {
   if (T == NULL) return true; // Empty tnode is valid tnode
   // Check bounds (BST invariant)
   if (!(lower < T->c && T->c < upper)) return false;
   // Recusively check children
   if (!(is_tnode(T->left, lower, T->c))) return false;
   if (!(is_tnode(T->right, T->c, upper))) return false;
   // T->middle does not need to be within the same limits since it is a "root"
   // for another tree so we call this to check those invariants.
   if (!(is_tnode_root(T->middle))) return false;
   if (T->middle == NULL && T->data == NULL) return false;
   return true;
}
bool is_tnode_root(tnode *T) {
   return is_tnode(T, 0, ((int)CHAR_MAX)+1);
}
```

Node that `is_tnode_root` is the entry point, checking validity for a root node (whose character value has no limits), and `is_tnode` checks the ordering invariants that come with the BST structure in addition to checking that each `middle` field points to a valid root node.

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