Tries and `Lexicons`

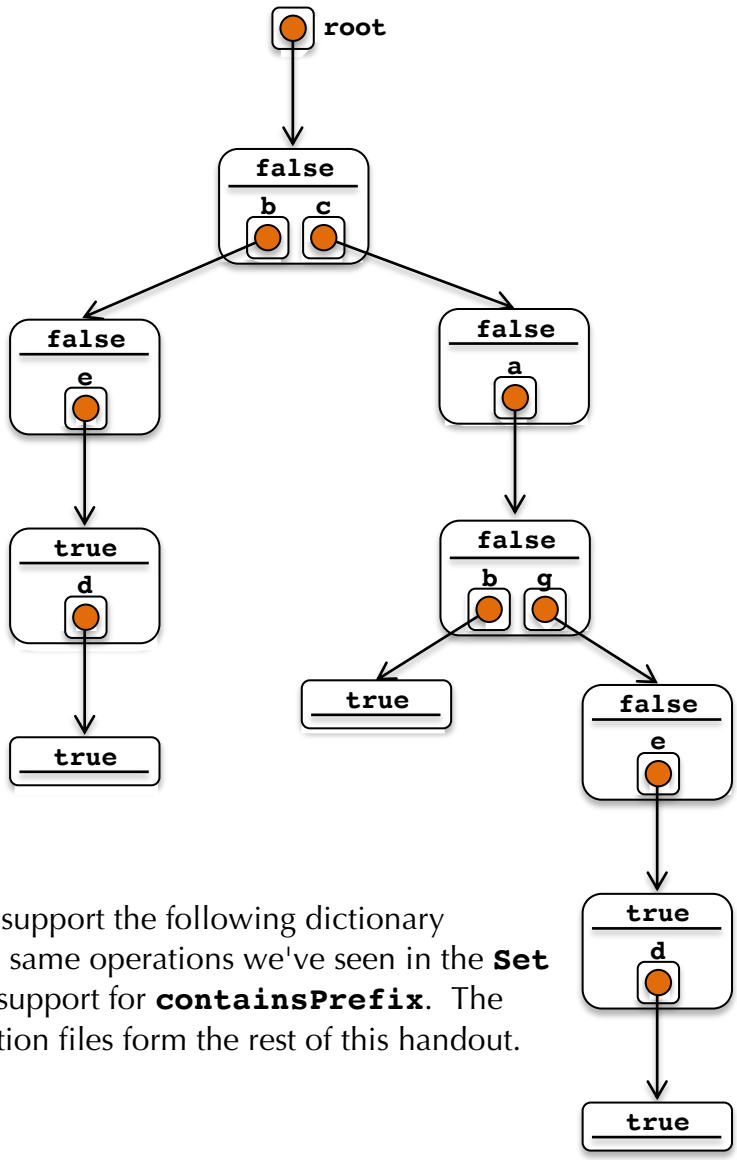
I don't want to leave you with the impression that all trees are binary, much less binary search. You've already seen the **`PQueue`** modeled as an array-backed binary tree, although binary *search* had nothing to do with the way elements were stored. You've also seen the binomial heap, which is a binomial tree that incidentally respects a heap property. Handout 28 here presents the tree structure used to back our **`Lexicon`**.

Implementing the `Lexicon`

Trees can be used as the underlying implementation of a **`Lexicon`** data type—one which stores a large collection of words and provide very efficient enter and lookup times. The resulting structure, first developed by Edward Fredkin in 1960, is called a **trie**. Over time, its pronunciation has evolved to where it is now pronounced *try*, even though the name comes from the central letters of *retrieval*. The trie-based implementation of the **`Lexicon`** makes it possible to determine whether a word is in the dictionary more quickly than you can using a hash table, and it offers natural support for confirming the presence of prefixes in a way that hash tables can't.

On one level, a trie is simply a tree in which each node branches in as many as 256 different directions, one for each position in the ASCII table. When using a trie to back a lexicon, the words are implicitly represented by the tree itself, each word represented as a chain of links moving downward from the root. The root of the trie corresponds to the empty string, and each successive level of the trie corresponds to the prefix of the entire word list formed by appending another letter to the **`string`** represented by its parent. The **`A`** link descending from the root leads to the sub-trie containing all of the words beginning with **`A`**, the **`B`** link from that node leads to the sub-trie containing all of the words beginning with **`AB`**, etc. Each node stores a **`true`** whenever the substring that ends at that particular point is a legitimate word.

If we pretend that the English alphabet only has 7 letters (let's say **`A`** through **`G`**) and we further assume that the English language only has five words—**`be`**, **`bed`**, **`cab`**, **`cage`**, and **`caged`**—then the underlying trie structure of the English **`Lexicon`** would look like that presented on the next page.



In a nutshell, we'd like to support the following dictionary operations—basically, the same operations we've seen in the **Set** template, with the added support for **containsPrefix**. The interface and implementation files form the rest of this handout.

lexicon.h

```
class Lexicon {
public:
   Lexicon() { root = NULL; }
   ~Lexicon() { delete root; }

   void add(const std::string& word);
   bool contains(const std::string& word) const;
   bool containsPrefix(const std::string& prefix) const;

private:
   struct node *root;
   const struct node *findNode(const std::string& str) const;
   struct node *ensureNodeExists(const std::string& str);
};
```

One thing that might surprise you: the declaration of the **root** field, which is declared to be a pointer to a **struct node**. **struct node** isn't actually defined yet, but in spite of that you're allowed to declare pointers to one anyway. The **struct node** tag is an example of an **incomplete type**—something that hasn't been defined yet, but would be compatible with any data type that's eventually fleshed out under the name **struct node**. I'm operating on the premise that the decision to use a trie is so secret that I don't even want to expose the full **struct node** definition in the interface file.

We'll see that **lexicon.cpp** will provide a **struct node** definition, and code appearing after it will be able to dereference **root**, allocate **struct node**s, and so forth.

lexicon.cpp

```
struct node {
   bool isWord;
   Map<char, node *> suffixes;
   node();
   ~node();
};

node::node() {
   isWord = false;
}

node::~node() {
   for (char ch: suffixes) {
      delete suffixes[ch];
   }
}
```

Yes, we've revisited the **struct node** tag here, this time associating it with a full record definition. Recall that **struct**s are really just classes where everything is **public**. I've implanted a small, obvious constructor and a clever, recursive destructor, and left the data members as **public**. OO purists wince at **public** data members, but it's different here, because we're subscribing to an OO mentality not so much to encapsulate and hide information from a client, but to provide initialization and destruction directives to be invoked automatically whenever **new** and **delete** are used to create and kill off **node**s. We **could** make the data members **private**, but the **node** is defined specifically to assist the implementation of the **Lexicon**, and we don't want to put up artificial, academic roadblocks to get in the implementer's way.

```
void Lexicon::add(const string& word) {
   ensureNodeExists(word)->isWord = true;
}

bool Lexicon::containsPrefix(const string& prefix) const {
   return findNode(prefix) != NULL;
}

bool Lexicon::contains(const string& word) const {
   const node *found = findNode(word);
   return found != NULL && found->isWord;
}
```

The above implementations pass the buck to **findNode** and **ensureNodeExists** methods. **containsPrefix** returns **true** if and only if a node exists on behalf of the provided **string**. **contains** returns **true** if and only if a node exists and that node states the accumulation of characters that led to it represent a word. **add** needs to ensure that a node exists on behalf of the provided character sequence and that the **isWord** field within that node is **true**. Of course, **findNode** and **ensureNodeExists** do the spidery crawl down the trie, and those implementations are presented here:

```
const node *Lexicon::findNode(const string& str) const {
   const node *curr = root;
   for (size_t pos = 0; pos < str.size() && curr != NULL; pos++) {
      curr = curr->suffixes.containsKey(str[pos]) ?
             curr->suffixes.get(str[pos]) : NULL;
   }

   return curr;
}

node *Lexicon::ensureNodeExists(const string& str) {
   node **currp = &root;
   size_t pos = 0;
   while (true) {
      if (*currp == NULL) *currp = new node;
      if (pos == str.size()) break;
      currp = &((*currp)->suffixes[str[pos]]);
      pos++;
   }
   return *currp;
}
```