

Generalization Of Trees

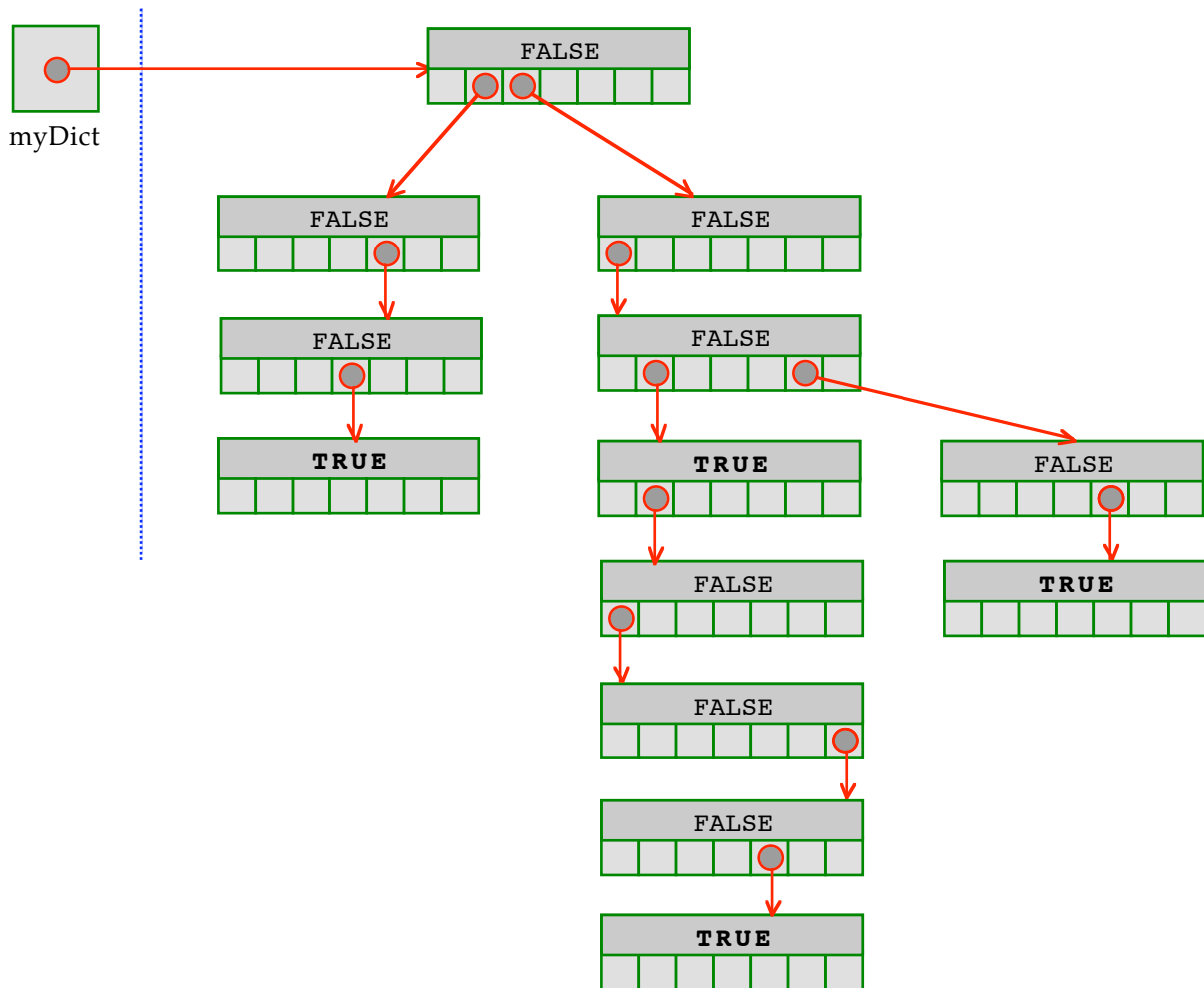
I don't want to leave you with the impression that all trees need to be binary, much less binary search. You've already seen the **PQueue** modeled as an array-backed binary tree, although binary search had nothing to do with the way elements got stored. I want you to see another example that generalizes trees in a novel way to implement the **Lexicon** class you've come to adore. You're all reading Chapter 13 on trees, but this particular example isn't in there, so you'll want to give this handout a good read. It's also another example that's complicated enough that it requires some advanced pointer work, so it's all good fun for memory management enthusiasts as well.

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, the pronunciation of this word has evolved to the point that it is now pronounced *try*, even though the name comes from the extraction of the central letters of *retrieval*. The trie-based implementation of the **Lexicon** makes it possible to determine whether a word is in the dictionary or not much more quickly than you can using a hash table.

At one level, a trie is simply a tree in which each node branches in as many as 26 different directions, one for each letter of the alphabet. When using a trie to represent a lexicon, the words of the lexicon are stored implicitly in the structure of the tree and are 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 adding one more letter to the **string** represented by its parent. For example, the **A** link descending from the root leads to the subtrie containing all of the words beginning with **A**, the **B** link from that node leads to the subtrie containing all of the words beginning with **AB**, etc. Each node stores a Boolean flag that is **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 (**A** through **G**—the names granted to the seven natural tones in music) and then we further assume that the English language only has four words—**bed**, **cab**, **cabbage**, and **cafe**—then the underlying trie structure of the English **Lexicon** would look like that presented on the next page. (Boxes storing the **NULL** pointer are left empty.)



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

```

/**
 * File: lexicon.h
 * -----
 * lexicon.h exports a package almost identical to that used in the Boggle
 * assignment.
 */

class Lexicon {

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

    void add(string word);
    bool containsPrefix(string prefix);
    bool containsWord(string word);

    template <typename UnaryFunc>
        void mapAll(UnaryFunc fn) { mapTrie(fn, "", root); }
    template <typename BinaryFunc, typename ClientDataType>
        void mapAll(BinaryFunc fn, ClientDataType& auxData)
            { mapTrie(fn, auxData, "", root); }

private:
    struct Node *root;
    void add(string suffix, struct Node **nodep);
    struct Node *findNode(string prefix);
    template <typename UnaryFunc>
        void mapTrie(UnaryFunc fn, string stringSoFar, struct Node *curr);
    template <typename BinaryFunc, typename ClientDataType>
        void mapTrie(BinaryFunc fn, ClientDataType& auxData,
            string stringSoFar, struct Node *curr);
};

```

Trie-based implementation of the Lexicon

```

/**
 * File: lexicon.cpp
 * -----
 * lexicon.cpp provides the underlying implementation of the lexicon
 * abstraction. It uses a trie for the underlying implementation.
 *
 * At one level, a trie is simply a tree in which
 * each node branches in as many as 26 different directions,
 * one for each letter of the alphabet. When using a trie to
 * represent a lexicon, the words are implicitly represented by
 * the structure of the tree and 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 subset of the entire word list formed
 * by adding one more letter to the string represented by
 * its parent. For example, the 'a' link descending from
 * the root leads to the subtrie containing all of the words
 * beginning with "a", the 'b' link from that node leads to
 * the subtrie containing all of the words beginning with "ab", etc.
 * Each node stores a Boolean flag which is true whenever the

```

```

* substring that ends at that particular node is a legitimate word.
*/

struct Node {
    bool isWord;
    Node *children[26];

    Node();
    ~Node();
};

/**
* Constructor: Node
* -----
* structs and classes are precisely the same thing, save for
* the one difference that the default access modifier for a struct
* is public, not private. That means that we can embed methods,
* constructors, and destructors inside the declarations of structs.
*
* Because this struct declaration comes in the .cpp, it's effectively
* private to the Lexicon implementation.
*
* The Node constructor is quite simple: it just zeroes everything out.
*/

Node::Node()
{
    isWord = false;
    for (int i = 0; i < 26; i++)
        children[i] = NULL;
}

/**
* Destructor: ~Node
* -----
* Levies the delete operator against all 26 children. Note
* that delete NULL is defined to be a no-op.
*/

Node::~~Node()
{
    for (int i = 0; i < 26; i++) delete children[i];
}

/**
* Private Method: add
* -----
* Recursively descends down through the trie along
* the path that corresponds to the specified suffix, relative
* to the specified node. If along the way it encounters NULLs,
* then it replaces those NULLs with legitimate nodes. Ultimately,
* once the recursion marches to the end of the path, it bottoms
* out by marking the path as one that corresponds to a legitimate
* word.
*
* This is a classic example of tail recursion (recursion comes at
* the end) that could easily be implemented using iteration instead.
*/

```

```

void Lexicon::add(string suffix, Node **currp)
{
    if (*currp == NULL) *currp = new Node();
    Node *curr = *currp;
    if (suffix == "") {
        curr->isWord = true;
        return;
    }

    assert(isalpha(suffix[0]));
    char firstChar = tolower(suffix[0]);
    int childIndex = firstChar - 'a';
    add(suffix.substr(1), &curr->children[childIndex]);
}

/**
 * Method: add
 * -----
 * Defined in terms of the two-argument version of Lexicon::add.
 */

void Lexicon::add(string word)
{
    add(word, &root);
}

/**
 * Private Method: findNode
 * -----
 * Descends into the trie, finds the node whose presence corresponds
 * to specified prefix, and returns its address (or NULL if the
 * specified prefix has no place in our Lexicon.)
 */

Node *Lexicon::findNode(string prefix)
{
    Node *curr = root;
    while (curr != NULL && prefix != "") {
        char firstChar = tolower(prefix[0]);
        if (!isalpha(firstChar)) return NULL;
        int childIndex = firstChar - 'a';
        curr = curr->children[childIndex];
        prefix = prefix.substr(1);
    }
    return curr;
}

/**
 * Method: containsPrefix
 * -----
 * Returns true if and only if the specified string
 * is a prefix of some word in the Lexicon.
 */

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

```

```

/**
 * Method: containsWord
 * -----
 * Returns true if and only if the specified string is
 * a word in the dictionary. It relies on findNode
 * doing its job properly. If findNode returns a non-NULL
 * node whose isWord field is set high, then the word is
 * in there.
 */

bool Lexicon::containsWord(string word)
{
    Node *representativeNode = findNode(word);
    return ((representativeNode != NULL) &&
            (representativeNode->isWord));
}

/**
 * Private Methods: mapTrie (two overloaded versions)
 * -----
 * Recursively dives into the trie, keeping track of the
 * string that corresponds to the specified node. Note that
 * the function pointer is levied against the string (with auxData
 * in the second overloaded version) if and only if the isWord field
 * is true.
 */

template <typename UnaryFunc>
void Lexicon::mapTrie(UnaryFunc fn, string stringSoFar, Node *curr)
{
    if (curr == NULL) return;
    if (curr->isWord) fn(stringSoFar);
    for (int i = 0; i < 26; i++) {
        mapTrie(fn, stringSoFar + char('a' + i), curr->children[i]);
    }
}

template <typename BinaryFunc, typename ClientDataType>
void Lexicon::mapTrie(BinaryFunc fn, ClientDataType& auxData,
                    string stringSoFar, Node *curr)
{
    if (curr == NULL) return;
    if (curr->isWord) fn(stringSoFar, auxData);
    for (int i = 0; i < 26; i++) {
        mapTrie(fn, auxData, stringSoFar + char('a' + i), curr->children[i]);
    }
}

```