CS 14 • Home Prog. Assignment 4

Using Ternary Search Trees to Implement Automatic Word Completion

W. L. Truppel
wagner@cs.ucr.edu
Department of Computer Science and Engineering
University of California, Riverside

May 22, 2003

Introduction

Consider the following simple exercise under Unix: open a terminal window and create two new directories, named `directory` and `directories`, then type `cd dir` and follow that by hitting the TAB key.

In certain systems you will hear a beep but even if you don't you should see that the command line will have changed to `cd director`, even though you only typed `cd dir`. What happened is that the Unix shell you're running attempted to complete the command for you, but it stopped at *director* because more than one directory name starts with the string *director*. Now type an `i` and hit TAB again. Since there's now only one directory whose name starts with the string *directori*, the shell will entirely complete the command for you, resulting in `cd directories/`.

This neat feature is called word auto-completion. Microsoft's *Internet Explorer* also has this feature for URL's that you've previously visited: as you type the URL, you see all possible completions that *Explorer* knows about.

How is this feature implemented? I'm glad you asked, because that's exactly what your next programming assignment is about: you will write a program which, given a string, prints out all completions of that string, according to a dictionary which I will provide you with. Thus, for example, if you run your program on the input string for, your program should print the strings *for*ce, *for*cible, *for*m, *for*midable, and *for*esee, among others.

So, how *is* auto-completion implemented?

That's actually not quite the right question to ask because there are many ways one could implement this feature. The right question is: how is auto-completion implemented *efficiently*? The answer, of course, is that it uses a data structure that allows one to efficiently store, and search for, a large number of strings.

As it turns out, I don't know for sure which data structures Unix and *Internet Explorer* use to implement their versions of auto-completion. I suspect that they both use what's known as a Suffix Tree. In your assignment, however, you will be using another data structure, called a Ternary Search Tree.

A tree for every taste

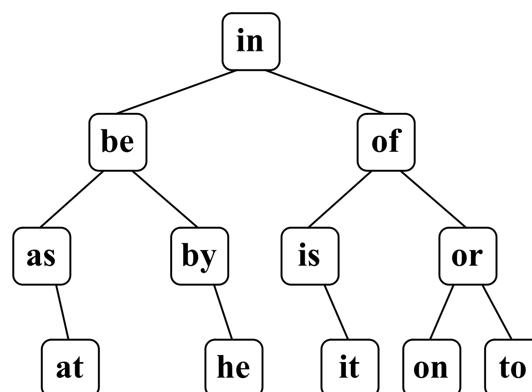
Our first problem in developing an algorithm for automatic word completion is, not surprisingly, that of storing strings. In order to motivate the ternary search tree approach, let's see how we could use other kinds of trees to solve this problem.

Binary Search Trees

Everyone is used to sorting strings *alphabetically*. This is possible because strings have a natural ordering: A comes before B, which comes before C, and so on, which allows us to say that the word ordering comes before the word ordinary, but after the word ordeal. Since words can be ordered, it's natural to ask whether we can use a Binary Search Tree (BST) to store and retrieve them. Well, we can!

Consider the following 2-letter words: in, be, of, as, by, is, or, at, he, it, on, and to.[1] Recalling that, in a BST, every node is *larger* than *every* descendant on its *left* subtree and *smaller* than *every* descendant on its *right* subtree, we can store these words in a BST as shown in the figure below.

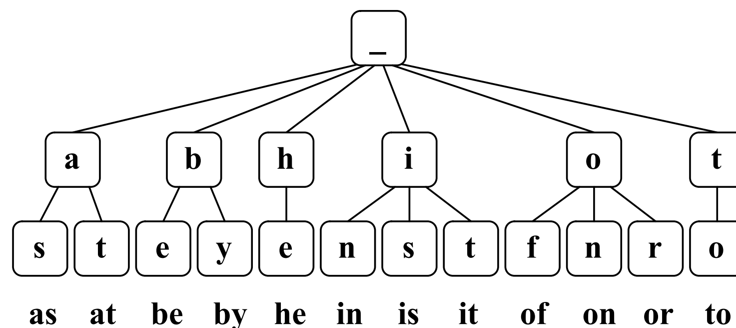
This approach is fine both in terms of storage and runtime efficiency, since insertions, deletions, and searches are logarithmic in the number of strings stored (assuming a balanced tree). The problem, however, is that we have no measure of how close one string is from another; all we can ascertain from any two nodes is whether the strings they store are ordered in ascending or descending order. Therefore, it's not easy to print all completions of a string.



[1] I'll be using 2-letter words as examples only because they're smaller and don't occupy as much space on the page. Everything I'll say here applies to words of any length.

Digital Search Tries

Another data structure we could think of using is a Digital Search Trie[2] (DST). A DST stores a single character in every node and each node has a branching ratio of at most 26. In other words, each node can have at most 26 children, each one storing one of the 26 letters of the latin alphabet.[3] A DST storing the same 2-letter words as before (in, be, of, as, by, is, or, at, he, it, on, and to) would look like this (the actual words are not stored in the tree, but are written below the leaves to make the picture more clear):



Without getting into too much detail, a DST lets us define some measure of closeness between strings, but at a huge cost. For example, in order to store all 3-letter words, from aaa to zzz, we would need a total of $1 + 26 + 26^2 + 26^3 = 18,279$ nodes. The storage cost is exponential on the length of the strings being stored. The advantage is, however, that we can in this case easily print all completions of a string. For example, all the completions of the string o, according to the DST above, are of, on, and or. These are obtained simply by printing the entire subtree rooted at o.

Ternary Search Trees

As we've seen, BST's are efficient in both storage and runtime, but aren't useful if one wants to implement word auto-completion. On the other hand, DST's are capable of performing auto-completion but entail a huge storage cost. Although the most useful data structure for string manipulations is a Suffix Tree, the solution we'll adopt here is to use a Ternary Search Tree (TST).

The easiest way to understand how TST's are used to implement auto-completion is to see how words are inserted into the tree. After a few examples, it should be clear how the whole procedure works and we'll be able to write some code.

Thus, suppose we start with an empty TST and we insert the word is. The result is that shown in part (a) of the figure on the next page. Next, suppose we want to insert the word be. In order to do that, we must compare the root of the tree (i) with the first character of the word to be inserted (b). They're not the same, so we need to create a node for the character b. Since b comes *before* i, we store the node for b as the *left* child of i. Also, since we just created a node for b, the rest of

[2] Yes, there is no spelling mistake; the tree *is* called a trie, pronounced the same as the verb to try.

[3] If we were using a different alphabet, we'd obviously have a different branching ratio.

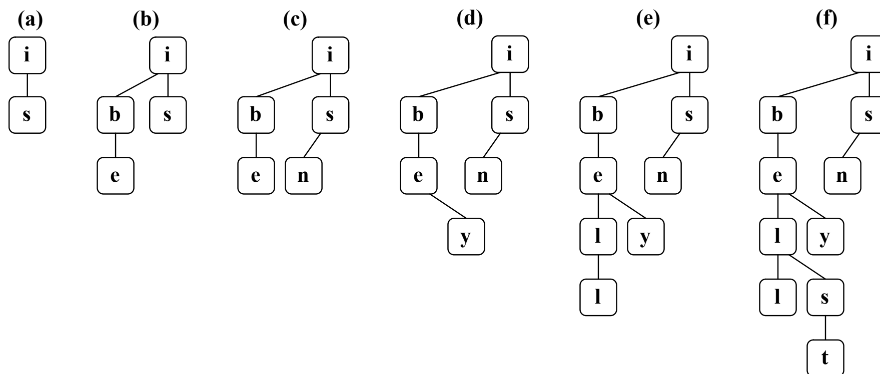
the word is stored vertically down from b, one character per node. This rule will become clear as we insert more items into the tree. The result so far is shown in part (b) of the same figure.

Now, suppose we want to insert the word in. Again, we compare the first character of the new word (i) with the root (i). In this case, they are the same so we move *straight down* from the node and compare the next letter in the new word (n) with the node we're now on (s). Since they are *not* the same, we need to create a new node for that second letter (n) and store it. We do so as the *left* child of the current tree node (s), since (n) comes before (s) in the alphabetical order. The result so far is then part (c) of the figure.

Next, consider what happens when we try to insert the word by. Once again we start by comparing the first letter of the word to be inserted (b) with the root (i). They're *not* the same, so we'd be tempted to insert a new node for b as the left child of i. But such a node already exists, so we don't create anything and, instead, we just move to that node's *straight-down* child (e) and, simultaneously, to the next character of the word to be inserted (y). Since y comes *after* e, we must create a new node for y and insert it as the *right* child of the current tree node (e). The result is part (d) of the figure.

Suppose then that we want to insert the word bell. Yet again, we start at the root (i) and compare it with the first character of the word to be inserted (b). They're not the same, so we move to the *left* child (b) of the current node (i) without changing the current letter (b) from the word to be inserted. These two are now the same, so we move *straight down* (to e) and, simultaneously, move to the next character (e) of the word to be inserted. Again, we found a match, so we again move *straight down* from the current tree node and also move to the next character (l) of the word to be inserted. But, wait... There is no node *straight down* from the current tree node, so we must create one. And again, since we just created a node, the rest of the word is stored in *straight-down* fashion. The result is part (e) of the figure.

Our final example is the insertion of the word best. Its first character (b) doesn't match the root's character (i), so we move to the *left* child of the root (b comes before i). Now the characters match, so we advance both the tree node and the string character, to e in both cases. Again, they match, and again we advance to the next tree node (l) and string character (s). Now they do *not* match, so we need to create a new node for s and, since s comes *after* l, we should do so as the *right* child of the current tree node (l). Finally, since we're creating a node, we insert the remainder of the word in *straight-down* fashion. The result is part (f) of the figure below.



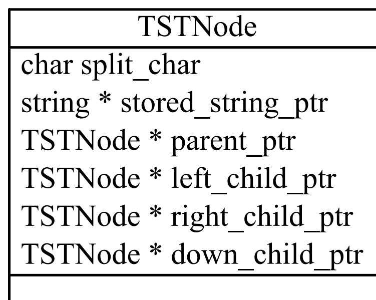
Some questions remain, however. How do we know which words are stored in the tree? For example, how do we know that be and by are stored in the tree but bey is not? How do we know that bel, with only one l, is not stored on the tree? These questions are important because, for example, if we were to ask for the completions of the string be, according to the tree as it is so far, we would *not* want to find bey, bel, and belst as possible completions. We'd only want bell and best.

The answer is that we store in each node a pointer to the string that it corresponds to. Thus, the node s under the node i stores a pointer to the string is, the node n under the node s stores a pointer to the string in, the node e under the node b stores a pointer to be, the node y stores a pointer to by, the leaf node l stores a pointer to the string bell, and the node t stores a pointer to the string best. All other nodes would have their string pointers set to NULL.

With this scheme in place, it's easy to find all completions of, say, the string be. First, we search the tree for the string be, landing on the node storing the character e. Now, we simply traverse the tree from that node down, printing all strings we find along the way. Moreover, if we use an in-order traversal, the completion strings will be printed in alphabetical order.

The TSTNode structure

We're now in position to formally define the characteristics of a TST node. Each node must store a pointer to its parent, a pointer to its left child, a pointer to its right child, a pointer to its straight-down child, a pointer to a string, and a character. Since only the tree itself will need to have access to the nodes, we can make TSTNode a struct, rather than a class. Its UML diagram follows.



The TST class

Our word auto-completion implementation will require an ADT TST with the following fundamental operations: **node insertion**, **node search**, and **in-order traversal**. We will not require **deletions**, although we might have. In order to obtain good performance, we'd want our TST to be balanced, which would require us to include some kind of balancing code in the code for insertion. However, to simplify the assignment, we'll have a different approach: the TST will not re-balance itself but, instead, we'll attempt to build it with a balanced input. The UML for the TST class appears below.

TST
- TSTNode * root
+ TST() + ~TST() + void readDictionary(const string & file_name) + void insertString(const string & string_ref) + bool hasString(const string & string_ref) const + void printCompletions(const string & string_ref) const - void inOrderPrintCompletions(const TSTNode * nptr) const - bool hasPrefix(const string & string_ref, TSTNode * & node_ptr, & int char_idx) const

Here's an explanation of the class' member functions:

- + TST(): this is the public constructor for the TST class.
- + ~TST(): the public destructor for the class.
- - bool hasPrefix(const string & string_ref, TSTNode * & node_ptr, int & char_idx) const: this private member function is used by the other member functions to search the TST for the node corresponding to a given prefix string. If it finds that node, it sets node_ptr to a pointer to the node found, sets char_idx to the index of the corresponding character from the prefix string (which should be the last one), and then returns true. If it does not find such a node, it sets node_ptr to a pointer to the last node found, char_idx to the index of the character in the prefix string associated with that node, and then returns false. Why this function returns some extra information will become clear later.

  A basic description of how it works is as follows: we must keep track of the current tree node, its parent node, the current character from the prefix string passed as the argument, and its index in that string. The current node starts as the root and the character starts as the first character of the prefix string (thus, the parent node starts with the value NULL and the character index with the value 0).

  Of course, if the tree is empty (the root is NULL), we should return false, setting node_ptr to NULL and char_idx to the only value that makes sense in this situation, namely, -1. Also, we should throw an exception if the prefix string is the empty string, since that input would be invalid.

  With those degenerate cases taken care of, whenever the current character from the prefix matches the character stored in the current node we must advance both, that is, we advance the character from the prefix to its successor and the node to its down child (the parent node then is set to what the current node was). If, however, the characters do not match, then we advance only the node, to either its left or its right child, depending on a comparison between the characters. If the current character from the prefix comes before the node character, we

advance the node to its left child, otherwise to its right child. Either way, we must also update the parent node.

If we reach a NULL node pointer for the current node before we reach the end of the prefix string, then we know that the prefix string isn't in the tree. In this case, we set node_ptr to the current parent node pointer (since that was the last valid node), set char_idx to the index in the prefix string of that valid node's character, and return false. If, on the other hand, we reach the end of the prefix string, the current node storing the last character of the prefix string is the node for which we should set node_ptr, char_idx is set to one less than the length of the prefix string (the index of the last character in the prefix string), and we return true.

- + bool hasString(const string & string_ref) const: this public accessor function is very similar to hasPrefix(). Users of the class typically will only care whether or not a string already exists in the tree and that's all the information provided by this function. It calls on hasPrefix() to do its job but must also check to make sure that the returned node contains a non-NULL value for its string pointer.

- + void insertString(const string & string_ref): this public mutator function is responsible for adding a string to the TST data structure. This function must call hasPrefix(), and also perform the same test as hasString(), to determine first if the string is already stored in the tree, in which case there's nothing to be done. If the string is not in the tree, it may be that the tree is empty (the root value is NULL) so we must deal with that contingency first. When the tree is empty, we must create a node to be the root and it will contain the first character of the string to be inserted. The remaining characters of the string are then inserted as *straight-down* child nodes of one another, in sequence.

  If the tree isn't empty and the string isn't already stored in the tree, we know that hasPrefix() will have returned the last valid node with a character from the string to be inserted, along with the index of that character in the string. It's now a matter of creating the necessary nodes for the remaining characters of the string to be inserted as we continue to descend along the tree. The algorithm is similar to that of hasPrefix(). We test the current string character against the character stored in the current node. If they match and the node has no *straight-down* child, we create such a node and store in it the *next* character of the string (recall that when the string character and the node character match, we must advance both of them).

  If the current string character and the character stored in the current node don't match, we move to the left or right child, depending on the result of the comparison between the characters, left if the current string character comes before the character stored in the current node, right otherwise. Once again, if such a node does not exist, we create one and store in it the current string character (not the next one). Of course, in the end, we must also not forget to set the string pointer of the last node to point to *a copy* of the actual string being stored.

- + void printCompletions(const string & string_ref) const: this public member function executes an in-order traversal of the tree to print out, in alphabetical order, all words stored in the TST which have for a prefix the string passed as the argument. We first call hasPrefix() to locate the last node associated with the prefix string passed as the argument. That node is the root of the subtree which we must traverse in in-order fashion, testing every node for a non-NULL string pointer.

- - void inOrderPrintCompletions(const TSTNode * nptr) const: this private member function is the heart and soul of the in-order traversal called for by printCompletions(). It's a recursive function, although it could also be written iteratively.
- + void readDictionary(const string & file_name): this public member function reads words from a dictionary, inserting them one by one into the TST. The dictionary will have its words sorted in a way to keep the tree as balanced as possible.

Good morning, Mr. Phelps[4]

Your assignment is to implement the TST class and to write a program that takes in a prefix string from the user and prints out all completions of that string, according to the dictionary which we'll be providing you with. More details of the assignment will be provided, if necessary, on a separate web page.

This document will self-destruct in 10 seconds[5]...

References

This document was adapted from the following two articles:

- *Plant your data in a ternary search tree*, by Wally Flint, in http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-ternary.html
- *Ternary Search Trees*, by Jon Bentley and Bob Sedgewick, in http://www.ddj.com/documents/s=921/ddj9804a/9804a.htm

[4]In case you didn't catch it, this is a reference to the TV series and movies Mission: Impossible. Every assignment starts with *Good morning, Mr. Phelps. Your mission, should you choose to accept it, is...*

[5]Another reference to MI...