# The Trie of Knowledge

James W Cooper

If you've ever watched a skilled chef work, you see that the chef not only has a broad knowledge of the properties and techniques for preparing food, but a whole bag of tricks for doing common tasks rapidly and effectively. For example Jacque Pepin's technique for cutting out a circle to line a cakepan comes to mind. You just fold a square of wax paper diagonally and then in half over and over until you have a small thick triangle. Then you put the point at the center of the cakepan and snip off the outside just at the outside of the pan. When you unfold it you have a perfectly cut pie liner.

Computer scientists have a lot of tricks up their sleeves as well, and we have talked a lot about a whole class of tricks called Design Patterns in any of a number of columns. However, there are older techniques that are just as valuable, and one of these is the Trie structure. I found a reference on the web to some class notes where the professor referred to the Trie as the "worst named concept in computer science." The weird spelling comes from the middle of the word re*trie*val, but the structure is pronounced "try." Nonetheless, it really represents a tree structure for storing character data.

Suppose you have a fairly large set of names, or of technical terms, or even of numbers as strings, that you want to organize and access rapidly. You could alphabetize them, of course, and you could keep a table of where each new letter started in the list so you could go to terms starting with that letter directly. But, if you have a fairly large list of names or words, you might have to alphabetize further, by interior letters. This is what you can do effectively using a Trie.

For an alphabetic system, a Trie is organized as a set of 26 bins, one each for each letter of the alphabet. (In this simple example, we will ignore case sensitivity and numbers and punctuation.) Then, each bin contains an additional 26 bins for the second letter of the word, and each of those bins an additional 26 bins for the third letter, and so forth. This is illustrated in Figure 1.

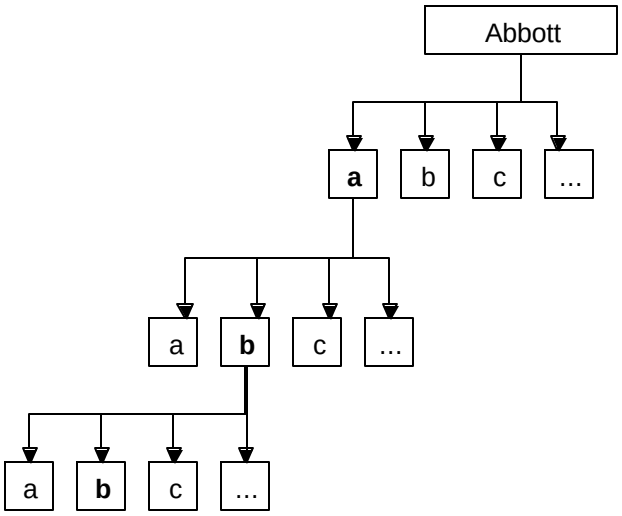

**Figure 1- Part of an alphabetic Trie structure, showing how we would begin to find the name Abbott.**

Now, how do we implement a Trie in Java? The approach I came up with is an array of vectors at each level. Thus, while we need to reserve space for the vectors, we don't have to reserve space for their contents unless they are used. This is significant because all alphabetic combinations are not equally likely, and we need not reserve spaces for names like "Mxyztplk."

Computing the position in an array for each letter amounts to finding that letter's distance from the letter 'a.' If we assume that each character in a name can only be a letter, then we need to translate any non-alphabetic characters into letter positions using some simple algorithm. In the example code below, we simply make them all equivalent to the letter 'a.'

```
//gets index of character into array of nodes
//non alphs are all returned as 0-offset
protected int getIndex(String t, int posn) {
  int index = 0;
  if (posn < t.length()) {
      char ch = t.charAt(posn);
      //return offset if letter, else return 0
      if (Character.isLetter(ch)) {
            index =
            Character.getNumericValue(t.charAt(posn)) - base;
      }
  }
  return index;
}
```

where we define the base variable in the constructor as

```
base = Character.getNumericValue('a');
```

## *A Ticketing Example*

In my previous column, I showed how you might write a program to print cards or theater tickets, and how you would display a seating layout using the JTable object. We'll now continue that example, and write a program that allows you to enter a few characters of a customer's last name, and then bring up that customer's name and address. We will be creating a Trie of Customer objects.

We will parse a file which was exported from a mailing list database, where each line represents one customer:

```
"Mr.","William","Abbott","225 Lousy Ave","Bilgeport","CT", ,"06669",
```

Each customer object stores these values and has getter methods to obtain their values:

```
public class Customer {
      private String salutation, frname,lname,
            address, town, state, zip,
            phone, email;
      private StringTokenizer tok;
      //parse a line read from a data file
      public Customer(String s) {
```

```
            if(s.substring(0,1).equals(",")) {
                  s= " "+s;
            }
            tok = new StringTokenizer(s, ",");
            salutation = getToken();
            frname = getToken();
            lname = getToken();
            address = getToken();
            town = getToken();
            state = getToken();
            phone = getToken();
            zip = getToken();
            email = getToken();
      }
      //read the next token and strip its quotes
      private String getToken() {
            String s = "";
            if (tok.hasMoreTokens()) {
                  s = tok.nextToken();
                  if (s.length() > 1) {
                        s = s.substring(1);
                        s = s.substring(0, s.length() - 1);
                  }
            }
            return s;
      }
      //return the last name as the string
      //representation of the customer
      public String toString() {
            return lname;
      }
```

Note that we created a toString() method for this class that returns the last name for each object. This will make it possible to add a list of Customer objects to a list box and have them display their names without writing further code.

## *The TrieNode Class*

Each node that we add to the top level must contain a set of 26 vectors to the next lower level. Of course, we only initialize this array of vectors if that particular pathway is needed. This is illustrated in Listing 1, the TrieNode class:

```
public class TrieNode {
      //list of objects whose names fit here alphabetically
      private Vector objects;
      //Vector of 26 alphabetic links at next level
      private Vector links;
      //----------------------------
      public TrieNode() {
            objects = new Vector();
            links = new Vector();
      }
      public int size() {
            return links.size();
      }
      public Vector getLinks() {
            return links;
```

```
	}
	//add a customer object
	public void addObject(Object cust) {
		objects.addElement(cust);
	}
	public Object getObject() {
		if (objects.size() > 0) {
			return objects.elementAt(0);
		} else
			return null;
	}
	public boolean hasObjects() {
		return objects.size() > 0;
	}
	//get the i-th subnode
	public TrieNode elementAt(int i) {
		if (i < links.size()) {
			return (TrieNode) links.elementAt(i);
		} else
			return null;
	}
	public Vector getObjects() {
		return objects;
	}
	//return true if this node has subnodes
	public boolean hasNodes() {
		return links.size() > 0;
	}
	//converts a node to a string using the name of the customer
	public String toString() {
		Object cust = getObject();
		if (cust != null) {
			return cust.toString();
		} else
			return "";
	}
	//-----
	//if this node does not yet have any subnodes, create them
	//the depth number is really just for inspection and debugging
	public void addVectors(int depth) {
		if (links.size() <= 0) {
			for (int i = 0; i < JTrie.ALPH; i++) {
				TrieNode n = new TrieNode();
				links.add(n);
			}
		}
	}
	//-----
	//get an iterator to the links and their sublinks
	public Iterator getIterator() {
		return links.iterator();
	}
}
```

There is one significant design decision we have made in create this TrieNode class. What happens if there is a name collision, and there is more than one person with the

same last name? One solution is just to continue on to the first name, and the address if necessary, but this can lead to a needlessly deep Trie with most of the nodes completely empty.

Instead, we simply create a Vector of Customers whose last names are identical, and return the entire list of them for further selection by the user. We can see this in Figure 2, where we see that there are 3 people named "Abbott."

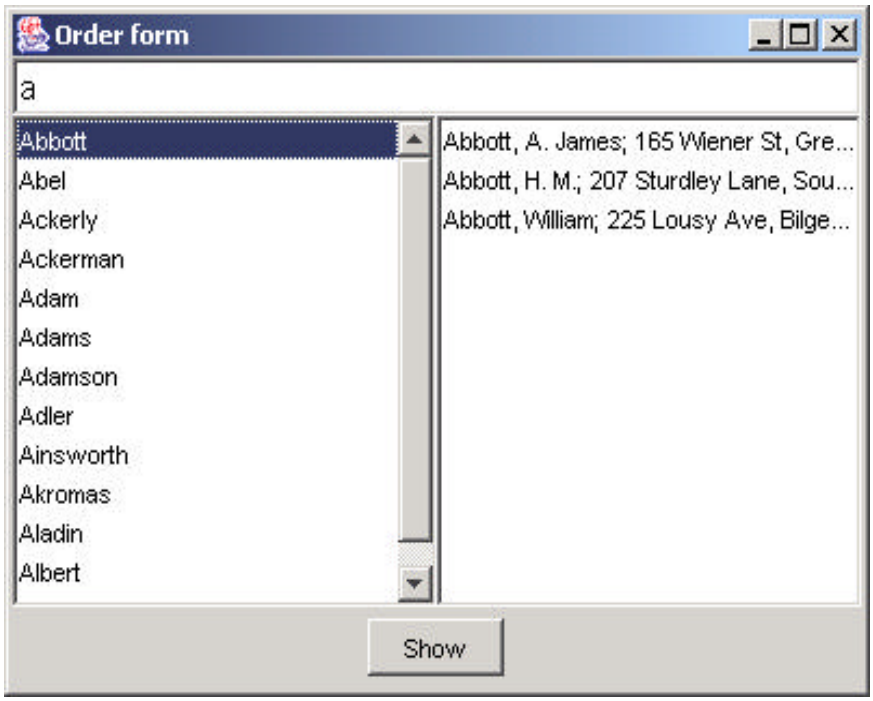


**Figure 2 – Selecting one of several people named "Abbott."**

Then you can select one of them and bring up the details of that customer in a window for alteration, as shown in Figure 3:

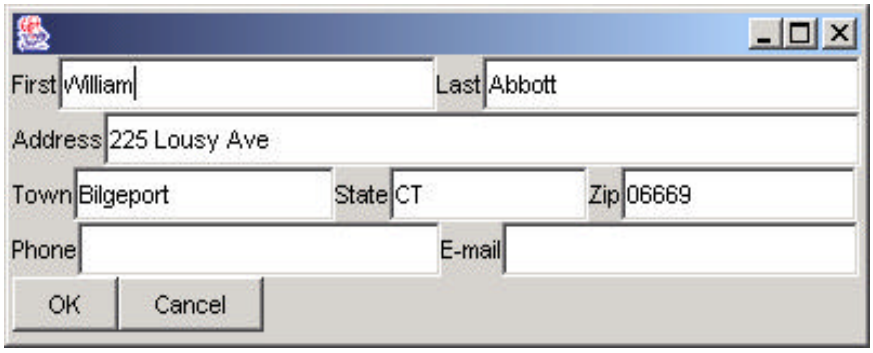


**Figure 3 – Details of one customer.**

## The JTrie Class

The JTrie class simply manages the nodes, and allows you to add objects to the right node based on the spelling. We immediately derive a CustTrie class from that uses the Customer last name to decide on where the object will be added:

```
/a specific implementation of a JTrie
//for arrays of Customer objects
public class CustTree extends JTrie{

	public void addCustomer(Customer cust) {
		//get a lower case version of string
		String t = cust.getLname().toLowerCase();
		int len = t.length();
		TrieNode v = node;	//start at current node
		//
		for (int i = 0; i < len; i++) {
			int index = getIndex(t, i);
			v.addVectors(i);
			v = (TrieNode) v.elementAt(index);
		}
		v.addObject(cust);
	}
}
```

## Iterating through the Nodes

Now, if you want to look for a given entry by name, you need only spell out the name and look down that path to find whether there is an entry in that position. However, if you want to see all of the entries below a certain partially entered name, we need to be able to iterate through the nodes depth first. This is the purpose of the NodeIterator class, which implements the Iterator interface, and keeps a Stack of nodes by width so that after you get to the bottom of a given node, you can pop back up and look under other nodes. Here are the relevant next methods.

```
/**
iterator is true if current one has more nodes
or if there are more at this level.
This is depth first iteratiion.
 */
public boolean hasNext() {
	if (node == null)
		return false;
	if (node.hasNodes())
		return true;
	else
		return iter.hasNext();
}
/**
	Get next node
 */
public Object next() {
	if (node != null) {
		node = nextNode();
	}
```

```
        while (node != null && !node.hasObjects()) {
                node = nextNode();
        }
        return node;
}
//compute next node
private TrieNode nextNode() {
        if (node.hasNodes()) {
                stack.push(iter); //save current iterator
                //and go for depth at this node
                iter = node.getIterator();
                node = (TrieNode) iter.next();
        } else {
                if (iter.hasNext()) {
                        node = (TrieNode) iter.next();
                } else {
                        //return to previous iterator when depth exhausted
                        if (!stack.isEmpty()) {
                                iter = (Iterator) stack.pop();
                        }
                        if (iter.hasNext()) {
                                node = (TrieNode) iter.next();
                        } else
                                return null;
                }
        }
        return node;
}
```

## *Summary*

The Trie structure is an efficient system for storing and accessing alphabetic or alphanumeric data, where each node in the Trie conatins a reference to the next lower set of 26 nodes. Of course, if there is no such lower node, the reference is null. You can insert data into the Trie by finding the correct spot character by character, and can fetch all the data below a partially spelled-out name by using a depth-first iterator through the nodes below the node representing that partial spelling.