CS1102 Semester 2 AY 2003/2003

Annotated Solutions for Lab 4A

Problem 41: Predictive Text-Based Input

Introduction

This problem, like problem 40, requires some amount of thinking and design before implementation. The goal is to ask students to implement their own tree-based data structure based on a description, and to write elegant recursive methods that exploit the recursive nature of trees. We pick an application that is familiar to everyone with a handphone.

Data Structures

First, we must decide on how to implement a trie. The structure of a trie is just like the structure of the binary tree, except that it can have 26 or 27 children, instead of just two. We can model the class structure of trie based on the structure of binary trees.

```
class Trie {
   TrieNode root;

   :
}

class TrieNode {
   TrieNode child[];
   boolean end;
   :

}
```

As hinted by the problem, we use an array to store the references to the children -- child[0] points to a subtree which stores all words prefixed by 'A', child[1] points to a subtree which stores all words prefixed by 'B' and so on. The other hint given in the problem is that we do not really have to store a node with '$'. Hence, we store a data field "end" that indicates if the current node is the end of a word.

Inserting Words

After we decided on the implementation of the data structure, we can now implement insertion of words into the Trie. We add a method call insert() to TrieNode that takes in a word as argument.

```
class TrieNode {

    :

1     void insert(String word) {
2         int index = charToIndex(word.charAt(0));
3
4         if (child == null) {
5             // allocate child array if this is the first time we visit this
6             // node.
7             child = new TrieNode[26];
8         }
9
10        if (child[index] == null) {
11            // create new child.
12            child[index] = new TrieNode();
13        }
14        if (word.length() == 1) {
15            // terminate
16            child[index].end = true;
17        } else {
18            // recursively insert
19            child[index].insert(word.substring(1, word.length()));
20        }
21    }
```

```
}
```

Insertion into the tree is pretty straight forward. We insert the first character of the word (lines 10-13), then recursively inserts the rest of the characters (line 19). The base case is when the word consists of only one character (lines 14-16). In this case, we simply set the "end" flag to be true to signify the end of word. Note that in the above methods, we optimize for space by allocating the array only when necessary, so that the leave nodes do not unnecessarily holds a large array that stores only nulls.

Exact Matching

After insertion, we can now perform matching of digit inputs to words. It is useful to store the mapping from digits to characters somewhere. In our implementation, we store it inside the class Trie, and make it static.

```
class Trie {
    :
    public static char map[][] = {
        {' ', ' ', ' ', ' '},  // 0
        {' ', ' ', ' ', ' '},  // 1
        {'A', 'B', 'C', ' '},  // 2
        {'D', 'E', 'F', ' '},  // 3
        {'G', 'H', 'I', ' '},  // 4
        {'J', 'K', 'L', ' '},  // 5
        {'M', 'N', 'O', ' '},  // 6
        {'P', 'Q', 'R', 'S'},  // 7
        {'T', 'U', 'V', ' '},  // 8
        {'W', 'X', 'Y', 'Z'}}; // 9
}
```

To perform exact matching, we can do it recursively. Given a sequence of digits $d_0d_1..d_n$, we can recursively match the sequence $d_1..d_n$ (lines 24-28) for each character represented by d_0 (line 7). If exact matches are found, we then prepend the character represented by d_0 to all the exact matches (lines 28-34). The base case is when input sequence has only one digit (lines 19-22). In this case, we found an exact match if and only if the corresponding node of that digit has its "end" field set to true.

An issue is where to store the exact matches and how to return them to the caller? In our implementation, we use a string called "result" to store the matches. A space is inserted between matches, and a StringTokenizer is used to iterate through each match. This is not very efficient. You can also use a Vector to store the matches.

The matching function is implemented in a method called match in class TrieNode.

```
1     String match (String digits)
2     {
3         String result = "";
4         int digit = Character.digit(digits.charAt(0), 10);
5
6         // for each character represented by digit
7         for (int i = 0; i <= 3; i++)
8         {
9             char prefix = Trie.map[digit][i];
10            int index = charToIndex(prefix);
11
12           // take care of the case where some digits maps to fewer letters
13           // than others.
14            if (index < 0 || index >= 27) {
15                continue;
16            }
17
18            if (child != null && child[index] != null) {
19                if (digits.length() == 1) {
20                    if (child[index].end) {
21                        result += " " + prefix;
22                    }
23                } else if (digits.length() != 1) {
24                    String digitsSuffix;
```

```
25                      digitsSuffix = digits.substring(1, digits.length());
26
27                      // recursive calls to match words down the tree.
28                      String suffix = child[index].match(digitsSuffix);
29
30                      // update exact match
31                      StringTokenizer st = new StringTokenizer(suffix);
32                      while (st.hasMoreTokens()) {
33                          result += " " + prefix + st.nextToken();
34                      }
35                  }
36              }
37          }
38          return result;
39      }
40  }
```

Predictive Matching

This is the hard part of the problem, and requires us to modify the methods above. To enable predictive matching, when we reach the end of the digit sequence (line 19) in match(), and we did not successfully find an exact match (line 20), we must check if the rest of the subtree lead to *exactly* one word. We could search the subtree for nodes with "end" flag set to 1 and see if there is exactly one such node, but we could also borrow a trick from the previous problem (Problem 40: Augmented Binary Search Tree) and augment the Trie with additional data fields to help us.

The trick is to augment each TrieNode with a data field "onlyWord" that stores the suffix of the only word stored in the subtree, or stores null if more than one word are stored in the subtree.

The initialization of "onlyWord" can be done in insert(). In insert(), the first word inserted is always the only word. Hence, when we create a new child node, we set onlyWord to the current suffix. During subsequence insertion into the child node, we set onlyWord back to null.

```
1     void insert(String word) {
2        int index = charToIndex(word.charAt(0));

         :

10       if (child[index] == null) {
11           // create new child.
12           child[index] = new TrieNode();
             child[index].onlyWord = word;
13       } else {
             child[index].onlyWord = null;
         }

       :
   }
```

Performing predictive matching now becomes trivial. If we ran out of digits, and the child node is not an end of word, we check if the child contains exactly one word somewhere down the subtree. If so, we just add the word to the result.

```
1     String match (String digits)
2     {
            :
18             if (child != null && child[index] != null) {
19                if (digits.length() == 1) {
20                    if (child[index].end) {
21                        result += " " + prefix;
                      } else if (child[index].onlyWord != null) {
                          result += " " + child[index].onlyWord;
22                    }
23                } else if (digits.length() != 1) {
                      :
                  }
              :
```

```
    }
```

One cute thing to note is that since we are visiting the children in increasing order of their alphabets, the matching results are automatically sorted.

Partial Solution

```
import java.util.*;
import java.io.*;

class TrieNode {
    TrieNode child[];
    String onlyWord;
    boolean end;

    TrieNode() {
        onlyWord = null;
        end = false;
    }

    void insert(String word) {
        int index = charToIndex(word.charAt(0));

        if (child == null) {
            // allocate child array if this is the first time we visit this
            // node.
            child = new TrieNode[26];
        }

        if (child[index] == null) {
            // create new child.
            child[index] = new TrieNode();
            child[index].onlyWord = word;
        } else {
            child[index].onlyWord = null;
        }

        if (word.length() == 1) {
            // terminate
            child[index].end = true;
        } else {
            // recursively insert
            child[index].insert(word.substring(1, word.length()));
        }
    }

    int charToIndex(char c)
    {
        return Character.getNumericValue(c) -
            Character.getNumericValue('A');
    }


    String match (String digits)
    {
        String result = "";

        int digit = Character.digit(digits.charAt(0), 10);

        for (int i = 0; i <= 3; i++)
        {
            char prefix = Trie.map[digit][i];
            int index = charToIndex(prefix);

            // take care of the case where some digits maps to fewer letters
            // than others.
            if (index < 0 || index >= 27) {
                continue;
```

```
            }

            if (child != null && child[index] != null) {
                if (digits.length() == 1) {
                    if (child[index].end) {
                        result += " " + prefix;
                    } else if (child[index].onlyWord != null) {
                        result += " " + child[index].onlyWord;
                    }
                } else if (digits.length() != 1) {
                    String digitsSuffix;

                    digitsSuffix = digits.substring(1, digits.length());

                    // recursive calls to match words down the tree.
                    String suffix = child[index].match(digitsSuffix);

                    // update exact match
                    StringTokenizer st = new StringTokenizer(suffix);
                    while (st.hasMoreTokens()) {
                        result += " " + prefix + st.nextToken();
                    }
                }
            }
        }
        return result;
    }
}

class Trie {
    TrieNode root;

    public static char map[][] = {
        {' ', ' ', ' ', ' '},  // 0
        {' ', ' ', ' ', ' '},  // 1
        {'A', 'B', 'C', ' '},  // 2
        {'D', 'E', 'F', ' '},  // 3
        {'G', 'H', 'I', ' '},  // 4
        {'J', 'K', 'L', ' '},  // 5
        {'M', 'N', 'O', ' '},  // 6
        {'P', 'Q', 'R', 'S'},  // 7
        {'T', 'U', 'V', ' '},  // 8
        {'W', 'X', 'Y', 'Z'}}; // 9

    void insert(String word)
    {
        if (root == null) {
            root = new TrieNode();
        }
        root.insert(word);
    }

    String match(String digits)
    {
        String exact, predict;

        if (root != null) {
            return root.match(digits);
        } else {
            return "";
        }
    }
}
```