Auto-predict using Retrieval Trees

Data Structures Mini-project

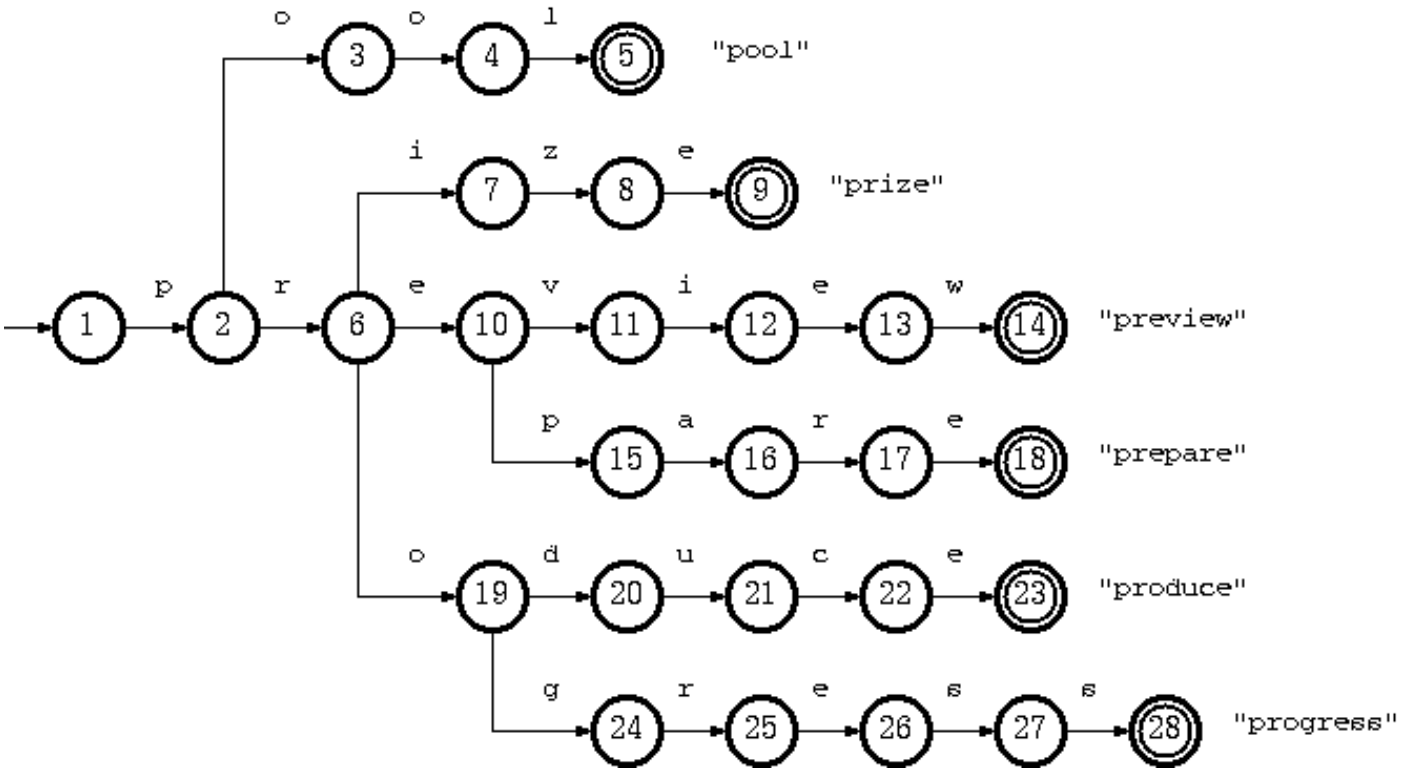
Batch B SAP 60004170119

Computer Engineering Semester 3 Year 2018-19

AIM: To implement a word auto-predict system using retrieval trees (trie).

TECHNOLOGY: C language, compiled using GCC 4.2.1 on a POSIX based operating system.

THEORY: A Retrieval tree, also called digital tree, radix tree or prefix tree is a kind of search tree—an ordered tree data structure used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Keys tend to be associated with leaves, though some inner nodes may correspond to keys of interest.



EXAMPLE OF A TRIE. DOUBLE BORDER REPRESENTS END OF WORD.

For example suppose we want to make a trie for the lower case English letters. Since there are 26 English letters, we have an array of 26 pointers. If the pointer at i^{th} location exists then it represents the $i+1^{th}$ English alphabet. If the pointer at index 9 points to another node then it represents the letter *'j'*. Hence we can define a node in a trie as:

```
struct node {
     struct node* children [26];
}
```

Data is actually stored in the form of routes to the child nodes in a retrieval tree.
However we also need to some sort of data in the trie to represent end of traversal. In the case of an auto-predict system we mark it as the end of a word or not.

```
struct node {
     struct node* children [26];
     bool isEnd; // from <stdbool.h> library in GCC 4.2.1
}
```

ALGORITHMS:

1. Insertion:

Value is stored at the end of the string in the tree.

```
insert(root, string, value):
    1. LET node ← root
    2. LET i ← 0
    3. WHILE i < length (string) - 1 LOOP
        a. LET index ← toIndex (string [i])
        b. IF node→children [index] == NULL THEN
                node→children [index] ← new node
           END IF
        c. node ← node.children [index]
        d. i ← i + 1
       END WHILE
    4. node.value ← value
```

2. Search:

This is a simple iterative function for finding a key in a general trie.

```
find(node, key):
    1. LET i ← 0
    2. WHILE i < length (key) - 1 LOOP
        a. LET index ← toIndex (string [i])
        b. IF node→children [index] == NULL THEN
             return Not Found
        c. node = node→children [index]
        d. i ← i + 1
    3. return node.value
```

3. Suggest:

This is a recursive function for finding all the words with a given prefix.

```
recursiveSuggest(node, currentPrefix):
    1. IF node→isEndOfWord THEN
         PRINT currentPrefix
    2. LET i ← 0
    3. WHILE i < length (node→children) LOOP
        a. IF node→children [i] == NULL THEN
             CONTINUE
        b. LET c ← toChar (i)
        c. LET n ← node→children [i]
        d. LET newPrefix ← currentPrefix + c
        e. recursiveSuggest (n, newPrefix)
       END WHILE

suggest(node, prefix):
    1. LET i ← 0
    2. WHILE i < length (prefix) - 1 LOOP
        a. LET index ← toIndex (string [i])
        b. IF node→children [index] == NULL THEN
             return Not Found
        c. node = node→children [index]
        d. i ← i + 1
    3. recursiveSuggest (node, prefix)
```