McGill University:
School of Computer Science
Winter 1997 Class Notes for 308-251

DATA STRUCTURES AND ALGORITHMS

Topic #7: TRIES AND SUFFIX TREES

This lecture is about tries and suffix trees. Here is some general information on tries:

- The term trie comes from the word "re**trie**val".
- Tries were introduced in the 1960's by Fredkin.
- A suffix tree is a member of the trie family.

Why do we use tries?

- The trie is a data structure that can be used to do a fast search in a large text. For example, we can think of the Oxford English dictionary which contains several gigabytes of text. The Oxford dictionary is a static structure because we do not want to add or delete any items. However, searching for an item in the dictionary is very important. Also, searching for a string should be efficient because overlapping of strings can occur.

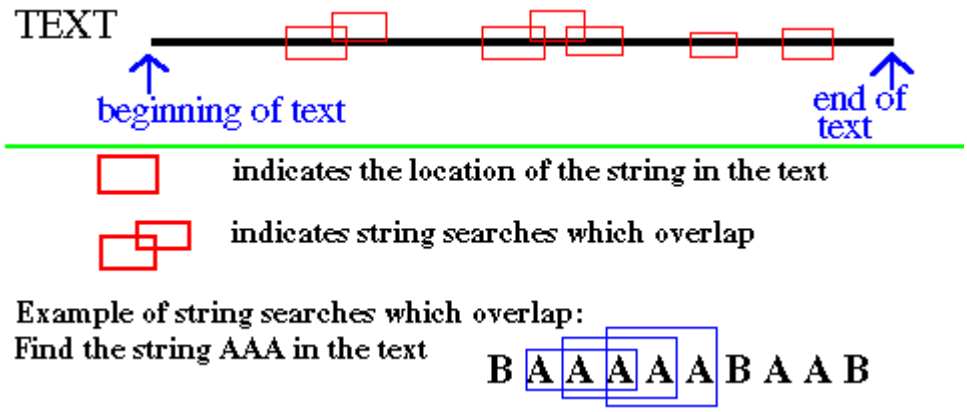


FIGURE 1

- Tries are used to implement the dictionary abstract data type (ADT) where basic operations like makenull, search, insert, and delete can be performed.
- They can be used for encoding and compression. (To be done in a later lecture.)
- They can be used in regular expression search and approximate string matching.

Regular expression

A regular expression is a way to define a pattern of characters. One example of a regular expression is **[aB]*[cd][efg]** where * is an indicator of repetition. The above expression represent any string with the following properties:

The string starts with any (possibly zero) number of a **or** B
followed by exactly one of c **or** d
followed by exactly one of e **or** f **or** g

Example: **aaace** and **Bdg** are two strings which can be represented by the above regular expression.

Another example of a regular expression is **^.*$** where ^ denotes the beginning of a line, the dot is a wildcard that represents any symbol and $ denotes the end of a line. Therefore, the expression **^.*$** represents a line containing any string repeated any number of time. That is, a whole line.
In UNIX, there is a command called grep which is used to report all matching substrings in a file.
We can type the command: **grep '[aB]*[cd][efg]' <filename>** and all substrings defined by this regular expression contained in <filename> will be listed.

Structure of a trie

-A trie is a k-ary position tree.
-It is constructed from input strings, i.e. the input is a set of n strings called $S_1,S_2,...,S_n$, where each S_i consists of symbols from a finite alphabet and has a unique terminal symbol which we call $.

Some examples of alphabets are:

- {0,l} for binary files.
- {all 256 ASCII characters}
- {a,b,c,d,...,x,y,z}

Tries are not only used to search for strings in normal text, but also to search a pattern in a picture. Here, the picture can be a gif file.

Kinds of tries

1. Non compact tries.
2. Compact tries.
3. Tries called &quotPATRICIA" which are even more compact.
4. Suffix tries.
5. Suffix trees.

Non compact tries

A non compact trie is one in which every edge of the underlying tree represents a symbol of the alphabet. (In the following example and for the rest of the this page we will always assume that the symbols in our alphabet are the CAPITAL letters A..Z with terminal symbol $)
Let's construct the trie from the following 5 strings: BIG, BIGGER, BILL, GOOD, GOSH.

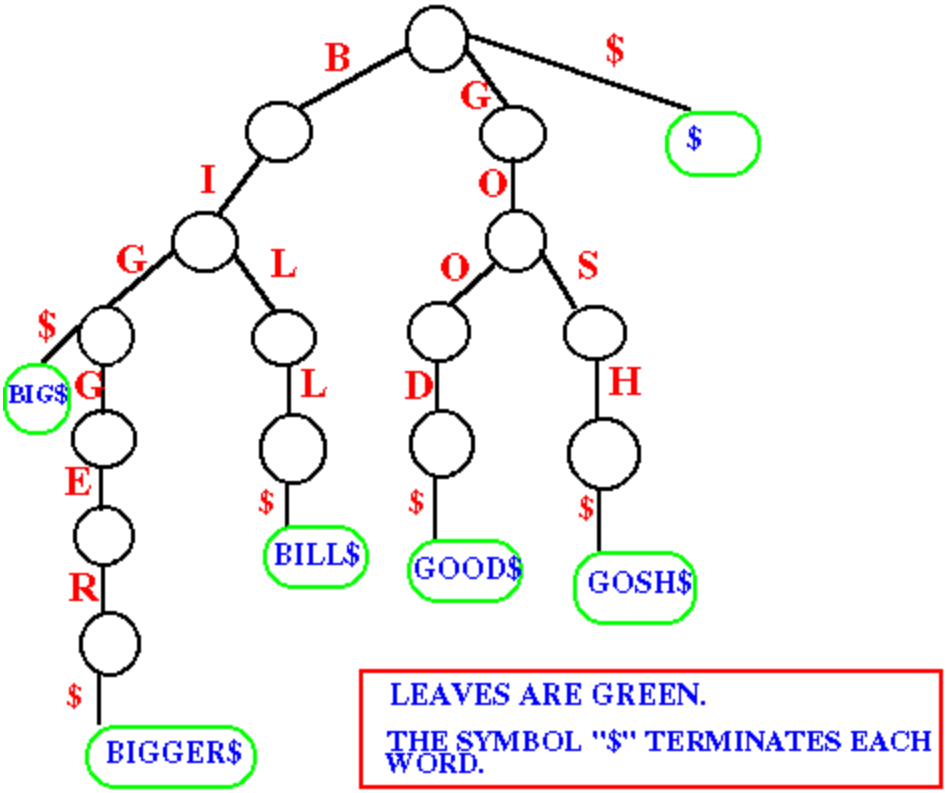


FIGURE 2

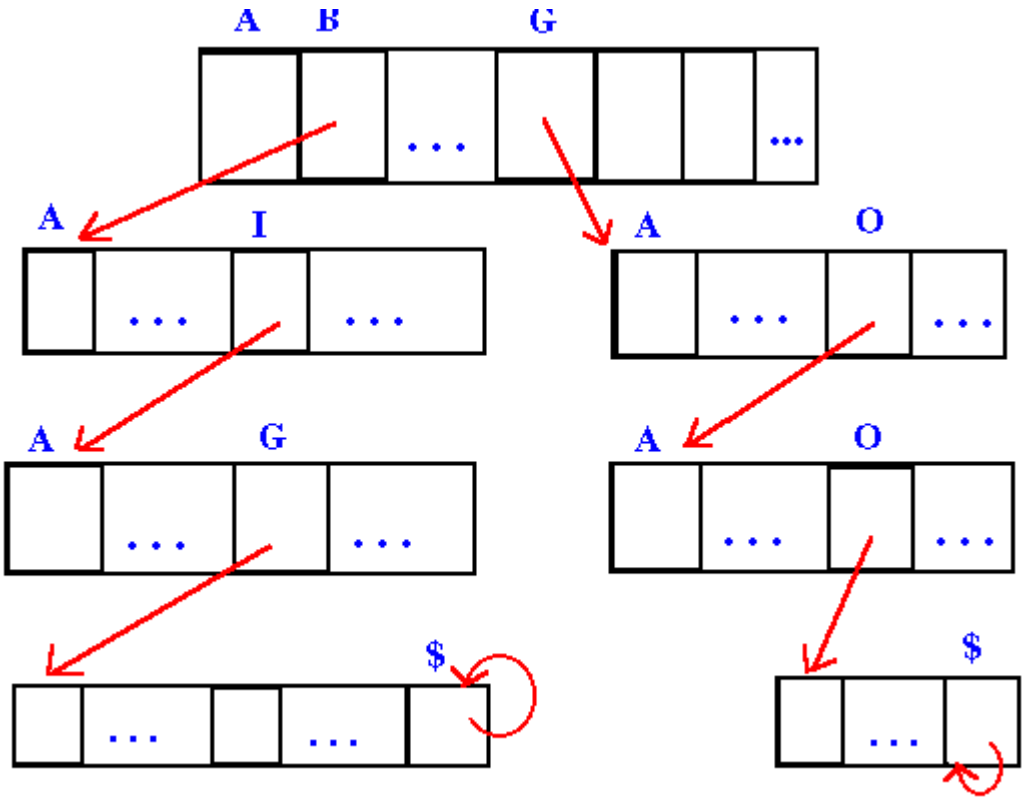
When we look for the string GOOD, we start at the root and we follow the G edge, followed by the O edge, another O edge and finally the D edge.
If we want to look for the string BAD, we start from the root, follow the B edge and find out that there is no A edge after. Thus BAD is not in the text.
The above structure is rather wasteful because each edge represents a single symbol. For huge texts this is an enormous waste of space. Instead, we must find a way to represent the trie in a more compact form.

Before building a compact trie, let's look at 2 types of implementation for tries.

1. **The first implementation uses an array of pointers.**

   Each array cell has an index. If the alphabet is from A to Z, the array indices will range from A to Z plus the terminal symbol $, so the size of the array is equal to the alphabet size plus 1.

   Here is an array representation for the strings BIG and GOO (in figure 3). A string start with either a B or a G. This is why in the first array, cell B and G have children to point to. The other cells have NIL pointers. For example, no word starts with the letter A, so cell A has a NIL pointer.



Trie representation for words BIG and GOO using array of pointers

FIGURE 3

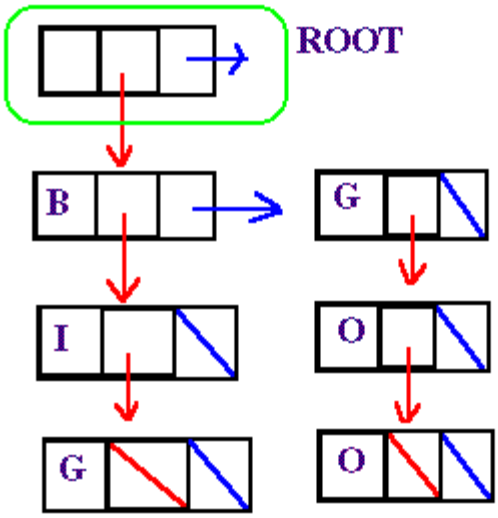
This implementation is wasteful when we have few words because most of the pointers are NIL.

2. **The second implementation uses linked lists and is due to &quotde la Briandais&quot.**

   Each node of the linked list represents a single symbol and has pointers to its next sibling and to its first child. Consider the following text:

   $$[s_1 \ldots s_{i-1} s_i \ldots s_n \$ \; s_1 \ldots s_{i-1} \; t_1 \ldots t_m \$]$$

   The first child of s_k is s_{k+1} for all k such that $1 < k < n$ (where $s_{n+1} = \$$)
   The next sibling of s_i is t_1. No other node have a sibling.



Trie representation for words BIG and GOO using linked lists

pointer to child **pointer to sibling**

FIGURE 4

Compact tries

This type of trie resembles the one in figure 2 except that chains which lead to leaves are trimmed. This is illustrated in figure 5.

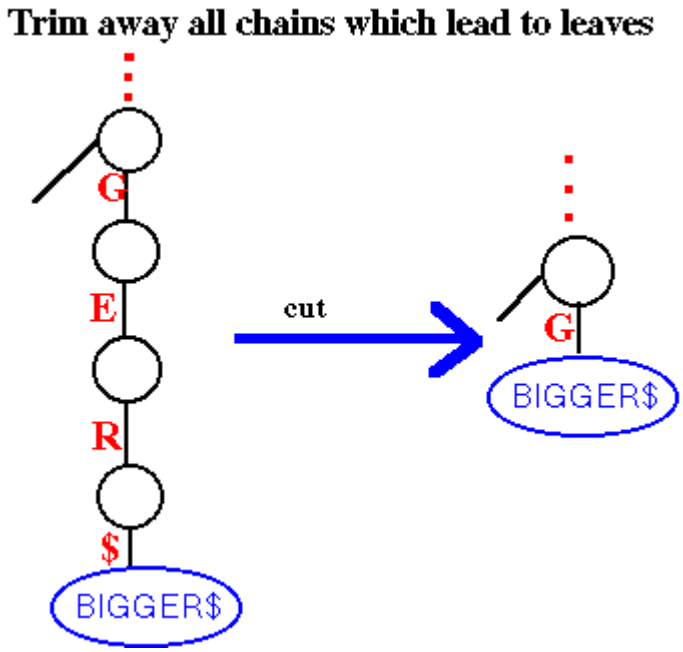
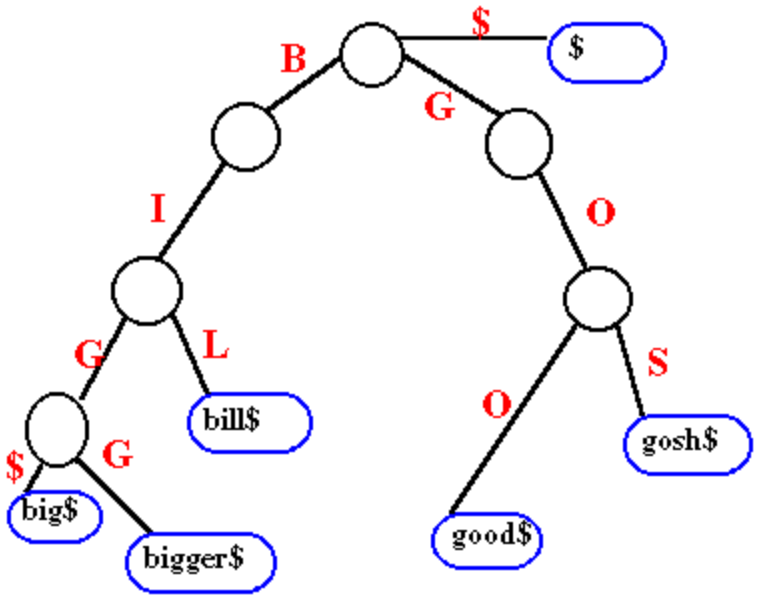


FIGURE 5

The compact form of the trie is in figure 6:



Trie for strings big, bigger, bill, good, gosh
This trie is more compact than the trie in figure 2.

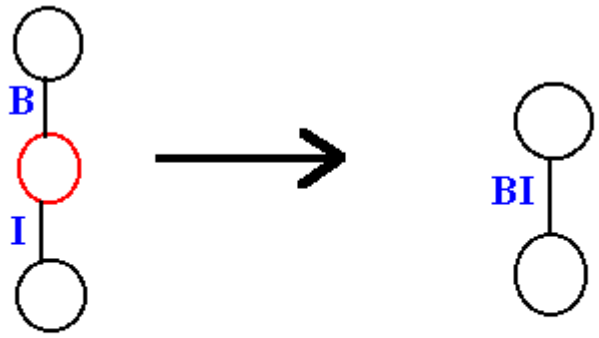
FIGURE 6

The number of leaves is n+1, where n is the number of input strings. Furthermore, in the leaves, we may store either the strings themselves or pointers to the strings (that is, integers).

The compact trie can be even more compacted. This will be discussed under the topic Tries called &quotPATRICIA&quot.

Tries called &quotPATRICIA"

&quotPATRICIA" stands for &quotpractical algorithm to retrieve information coded in alphanumeric&quot. This will be different from the previous trie in that an edge can be labeled with more than one character. Hence, all the unary nodes will be collapsed. The following figure illustrates the collapsing process:



Before, one edge is used to represent a character.
Now, the red node (unary node) is collapsed and one edge can hold more than one character.

FIGURE 7

The very compact trie will look as follows:

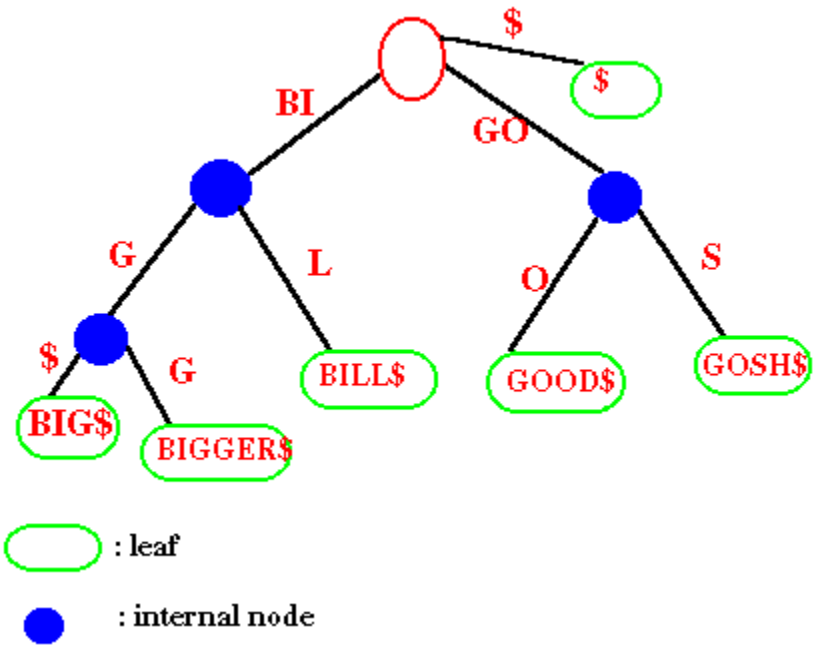


FIGURE 8

Note:
For binary PATRICIA tries (where there are only 2 symbols), the number of internal nodes is equal to the number of leaves minus 1. The height of the PATRICIA trie is bounded by n (the number of leaves). The heights of the ordinary trie or the compact trie are not necessarily bounded by n.

Suffix trie

Principles:
The idea behind suffix TRIE is to assign to each symbol in a text an index corresponding to its position in the text.(ie: First symbol has index 1, last symbol has indice n= #of symbols in text). To build the suffix TRIE we use these indices instead of the actual object.

The structure becomes exogenous and has several advantages:

1. It requires less storage space.
2. We do not have to worry how the text is represented. (bin, ASCII, etc)
3. We do not have to store the same object twice. (no duplicate)

A suffix trie is an ordinary trie in which the input strings are all possible suffixes.
A suffix of a text $[t_1 \ldots t_n]$ is a substring $[t_i \ldots t_n]$ where i is an integer between 1 and n.
To demonstrate the structure of the resulting tree we will build the suffix trie corresponding to the following text:

```
TEXT:      G O O G O L $

POSITION:  1 2 3 4 5 6 7
```

We begin by giving a position to every suffix in the text. We can now build a SUFFIX TRIE for all n suffixes of the text.

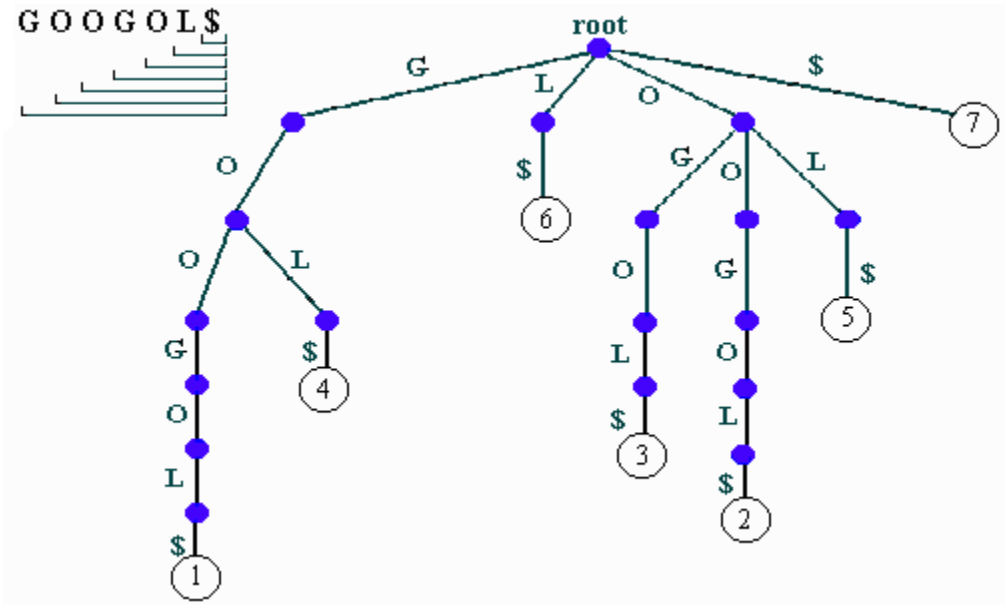


FIGURE 9

Note: The resulting tree has n leaves and height n.

Suffix tree

The suffix tree is created by TRIMMING (compacting + collapsing every unary node) of the suffix TRIE.

The following is a picture of a suffix trie. The corresponding suffix tree is drawn below.

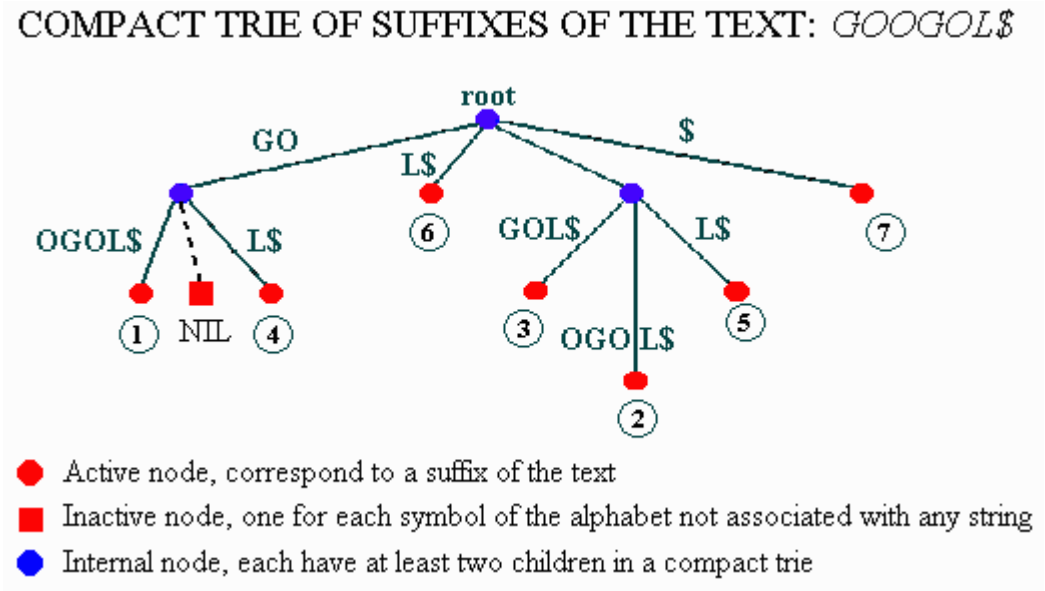


FIGURE 10

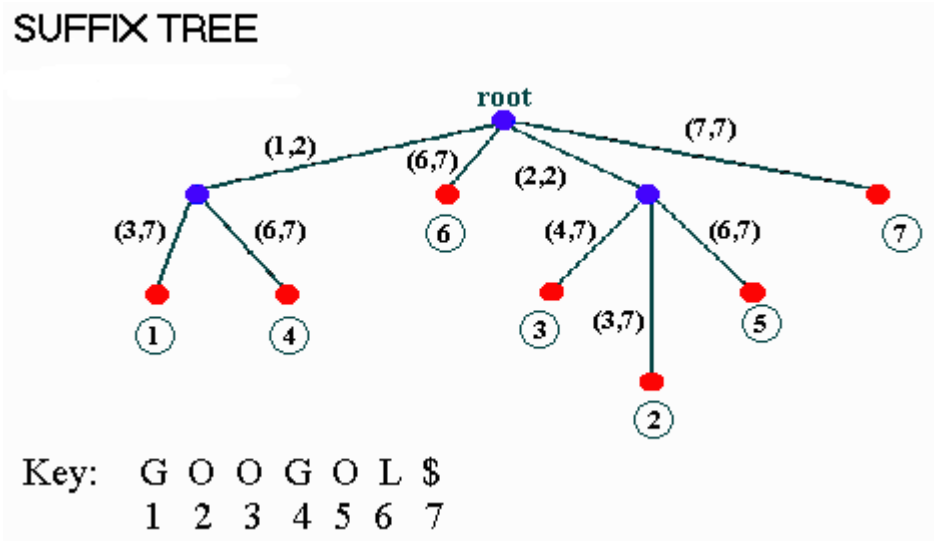


FIGURE 11

We store pointers rather than words in the leaves. And we replaced every string by a pair of indices, (a,b), where a is the index of the beginning of the string and b the index of the end of the string. i.e: We write

- (3,7) for OGOL$
- (1,2) for GO
- (7,7) for $

Remark: This make the storage in a suffix tree strickly O(n).

Search in suffix tree

Searching for all instances of a substring S in a suffix tree is easy since the symbols in S define a path down the suffix tree. Following this path, if we encountered a NIL pointer before reaching the end, then S is not in the tree. If we end up at a node x then S occurs at least once. Moreover the places where S can be found are given by the pointers in all the leaves in the subtree rooted at x.

Pseudo-code for searching in suffix tree:

SEARCH

> Start at root
> Go down the tree by taking each time the corresponding bifurcation
> If S correspond to a node then return all leaves in subtree
> If S encountered a NIL pointer then S is not in the tree

Ex:
If S = "GO" we take the GO bifurcation and return: GOOGOL$,GOL$.
If S = "OR" we take the O bifurcation and then we hit a NIL pointer so "OR" is not in the tree.

Analysis

If we have

- TRIE
- Alphabet = {0,1}
- Each string consists of infinitely many independent coin-flips

s_1 = 01101111010001....

s_2 = 11010110110101101...
...
s_n = 010111011101...

Putting a period in front, each string can be thought of as the binary expansion of a real number between 0 and 1.

- Compact the resulting tree by eliminating all chains leading to leaves

If a new string S of length k is to be added in the TRIE, we can expect that the distance between the leaf representing S and the root is approximately $\log_2 n$. Intuitive proof:

1. Each coin-flip has a 50% chance of being 0 or 1. So starting at any node, we have 50% chance of going left, 50% chance of going right.

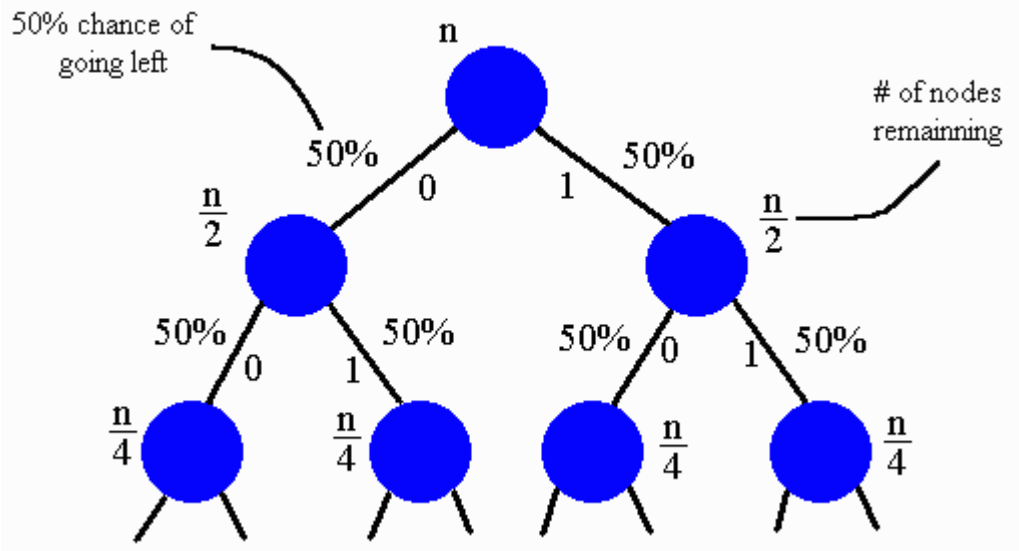


FIGURE 12

2. The process ends with the last digit of S. At that point we have:

$$n / 2^k \sim 1 \Longrightarrow k \sim \log_2 n$$

Insertion/Growing (Trie & Suffix tree)

Java Applet