DIGITAL SEARCH AND TRIES

by

David Toman

University of Waterloo

List of Slides

1
2 Searching Letter-by-Letter
3 Example
4 Trie Search
5 Trie Insertion
6 Binary Tries
7 PATRICIA Trees
8 Suffix Array
9 Space Requirements
10 Summary

Searching Letter-by-Letter

IDEA1: instead of basing search on comparisons of *complete* keys, we can use their internal representation as a sequence of letters (digits) instead

$\Rightarrow$ especially good for very long keys (strings)

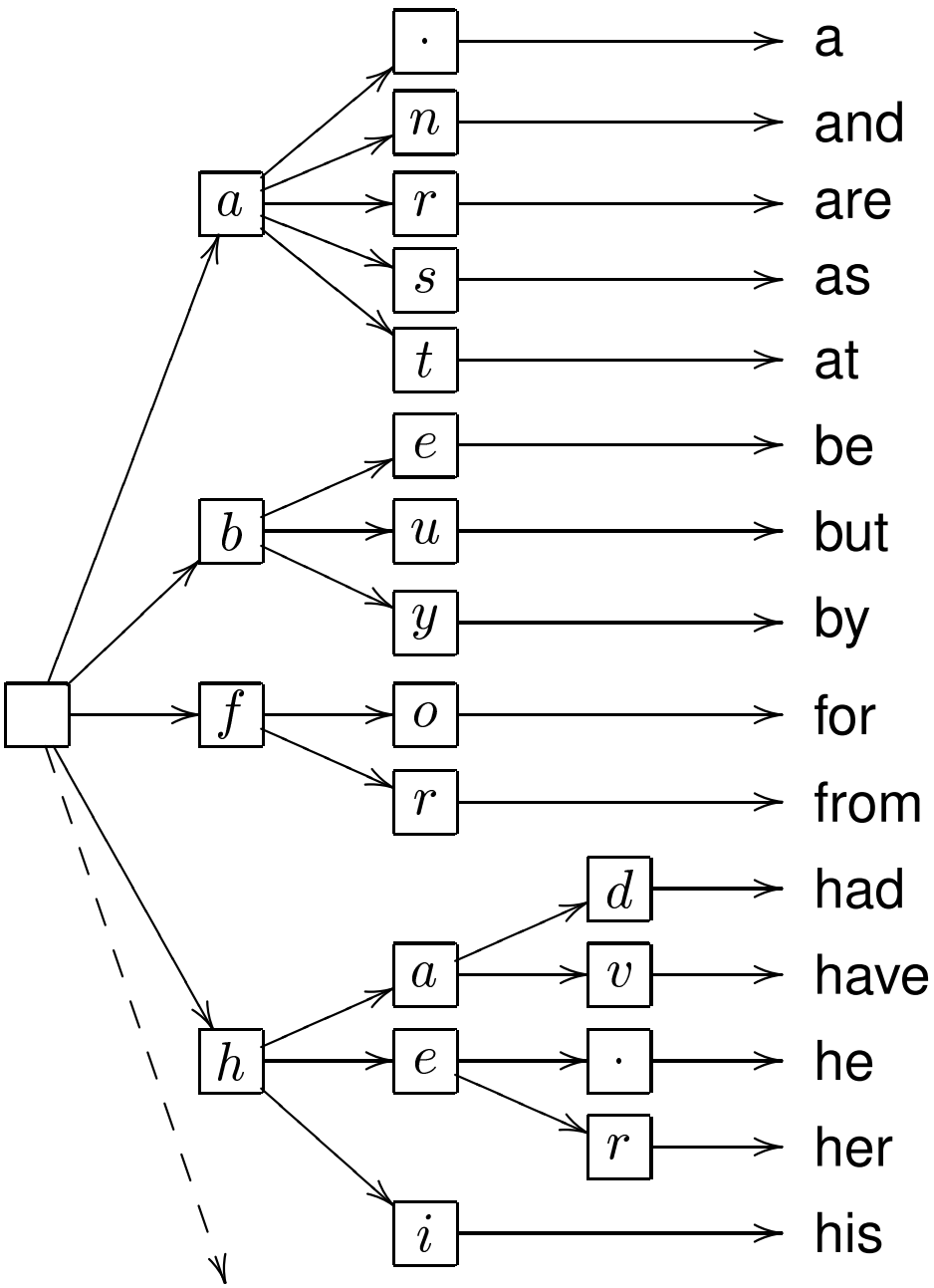
IDEA2: group words by the first letter, then within the groups by the second letter, etc.

$\Rightarrow$ a **Trie**: a $|\Sigma|$-ary tree

i-th level of the tree represents keys (words) with the same prefix up to i-th character

leaf nodes store "complete" keys

Example


·
n
a
r
s
t
e
b
u
y
f
o
r
d
a
v
h
e
·
r
i
a
and
are
as
at
be
but
by
for
from
had
have
he
her
his

Trie Search

Searching for key k of the form "$a_1 a_2 a_3 \ldots a_k$.":

Start with first letter ($i = 1$) and the root of the trie

1. is the current node a leaf?

   $\Rightarrow$ compare k with the word stored there and finish

   report success on match

   report failure otherwise

2. if not: is there a child of the current node labeled a_i?

   $\Rightarrow$ if not report failure.

3. move to the child labeled a_i and increase i

4. goto 1

NOTE: no key can be prefix of another key

$\Rightarrow$ usual solution: add end-of-key marker (".")

Trie Insertion

Similar to search:

$\Rightarrow$ if found nothing to do
$\Rightarrow$ otherwise we need to distinguish 2 cases:

1. search ends in an “internal” node

   $\Rightarrow$ add an appropriate child with the appropriate label
   $\Rightarrow$ add leaf with the new key

2. search ends in a leaf node but the key doesn’t match

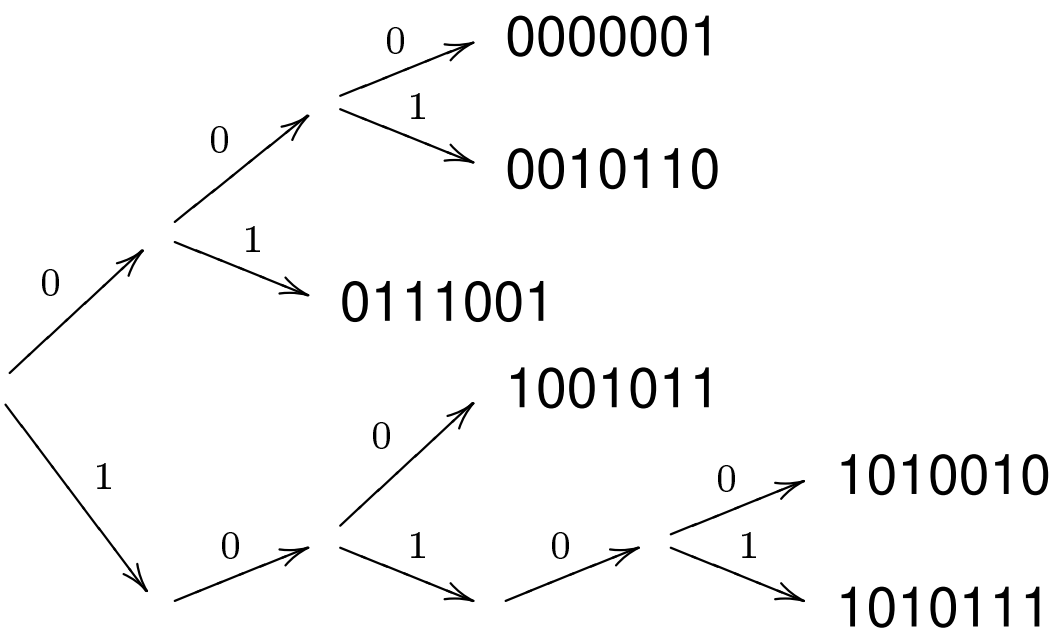
   $\Rightarrow$ create internal nodes for the common parts
   of the key and the original entry
   $\Rightarrow$ after the first mismatch add two leafs
   holding the key and the original entry

Binary Tries

IDEA: all keys can be represented as a sequence of **bits**

$\Rightarrow$ binary tree trie

Example:

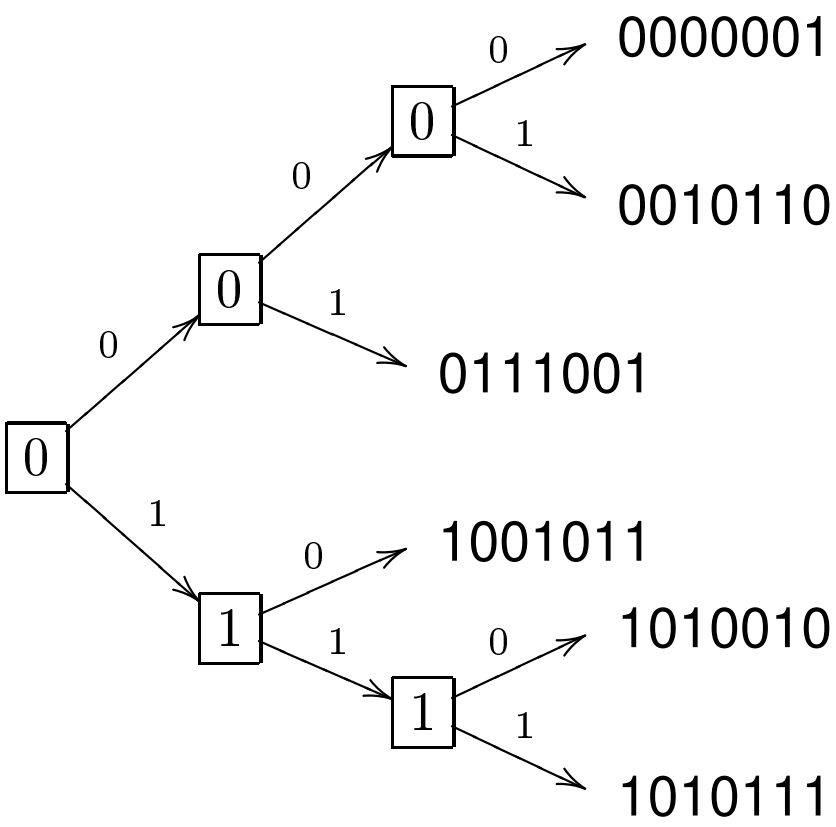


PATRICIA Trees

IDEA: eliminate "unary nodes in the trie

$\Rightarrow$ add *skip* values in the nodes

Example:



$\Rightarrow$ needs to "compensate" for skips by remembering one (leftmost) descendent of every node

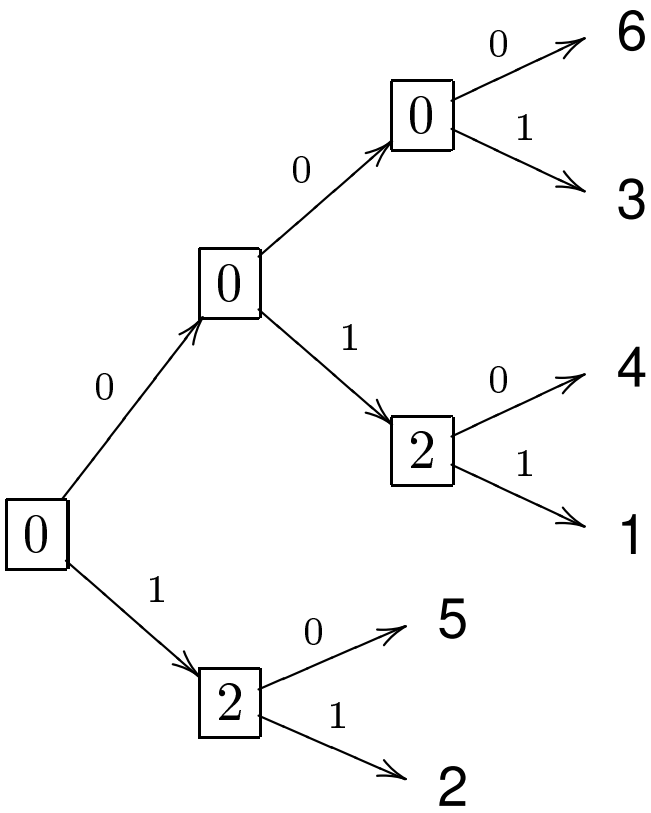
Suffix Array

Application: full text search

$\Rightarrow$ create a trie that stores **every** suffix of your text

$\Rightarrow$ instead of storing the actual keys in the leafs
store just a pointer to the original text.

Example: text = "01001000."



Note: search linear in size of the **key**!

Space Requirements

Problem: a naive implementation takes a lot of space

Example: text size $n = 4.10^9$ bytes

$\Rightarrow$ references $\sim$ 4 bytes
$\Rightarrow$ number of "index points" $\sim n/4$

Then

- index references: $n/5(4+1) = n$ bytes
- internal nodes: l/r child+leftmost descendent+skip $(4+4+4+1)n = 13n$.

. . . total more than 4 times the size of the original text.

Summary

- digital search takes advantage of key structure

  $\Rightarrow$ very efficient to detect failed searches
  $\Rightarrow$ binary version applicable for any key type

- very efficient for text searching

- need to be careful about space

  $\Rightarrow$ PATRICIA trees is a must
  $\Rightarrow$ advanced techniques $2n + o(n)$ bits exist (optimal)