Data Structures for Strings

Chapter 8

Introduction

- Numbers as key values: are data items of constant size, and can be compared in constant time.
- In real applications, text processing is more important than the processing of numbers
- We need different structures for strings than for numeric keys.

Motivating Example

- Example: 112 < 467 , Numerical comparison in O(1).

- Compare Strings lexicographically does not reflect the similarity of strings.
  - Western > Eastern , Strings comparison in O(min(|s1|,|s2|)). where |s| denotes the length of the string s

- Text fragments have a length; they are not elementary objects that the computer can process in a single step.
  - Pneumonoultramicroscopicsilicovolcanoconiosis !!!

Applications

- Bioinformatics
  - (DNA/RNA or protein sequence data).
- Search Engines.
- Spill checker.

8.1 Tries & Compressed Tries

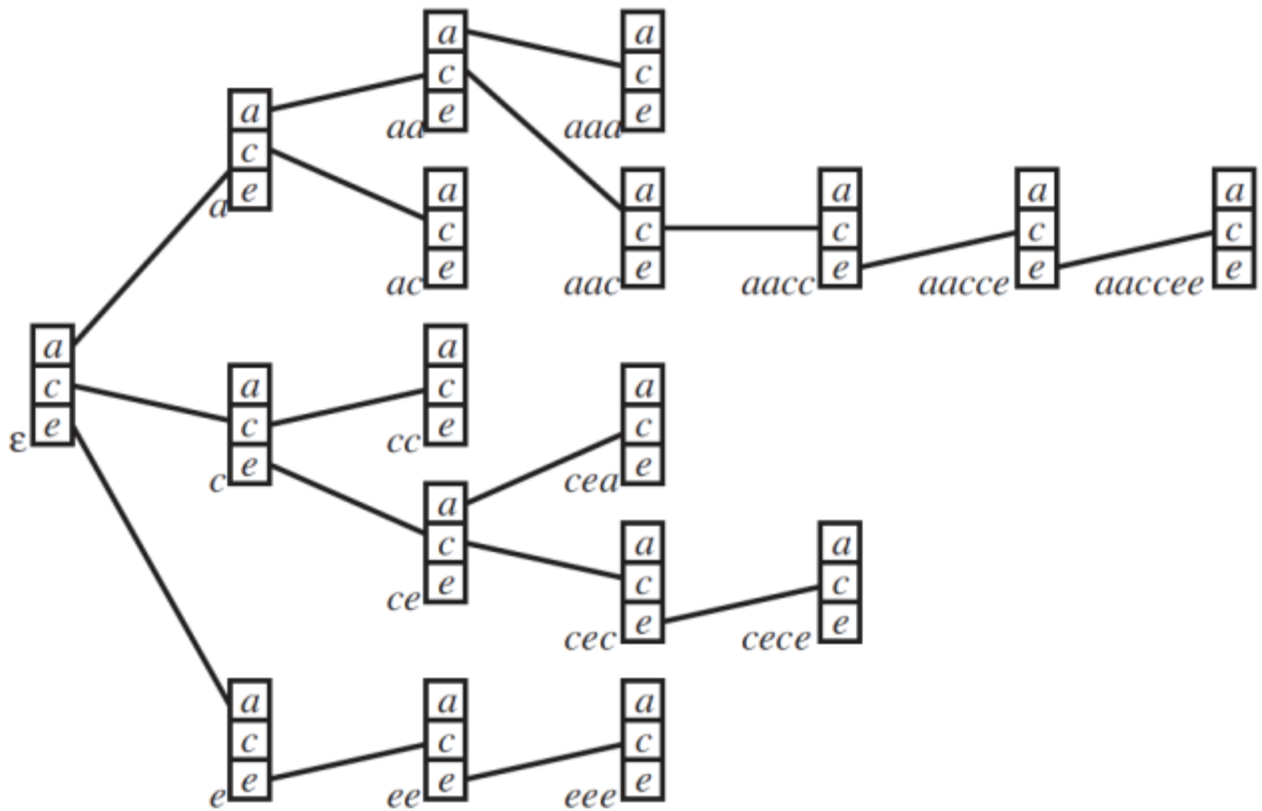
Tries

- The basic tool for string data structures, similar in role to the balanced binary search tree, is called “trie”
- Derive from “retrieval.” (Pronounced either try or tree)
- In this tree, the nodes are not binary. They contain potentially one outgoing edge for each possible character, so the degree is at most the alphabet size |A| .

Tries cont.

- Prefix Vs. Suffix.
- Ex. “computer”.
    - Prefix:(c, co, com).
    - Suffix: (r, er, ter)
- Each node in this tree structure corresponds to a prefix of some strings of the set.
- If the same prefix occurs several times, there is only one node to represent it.
- The root of the tree structure is the node corresponding to the empty prefix.

Tries Example

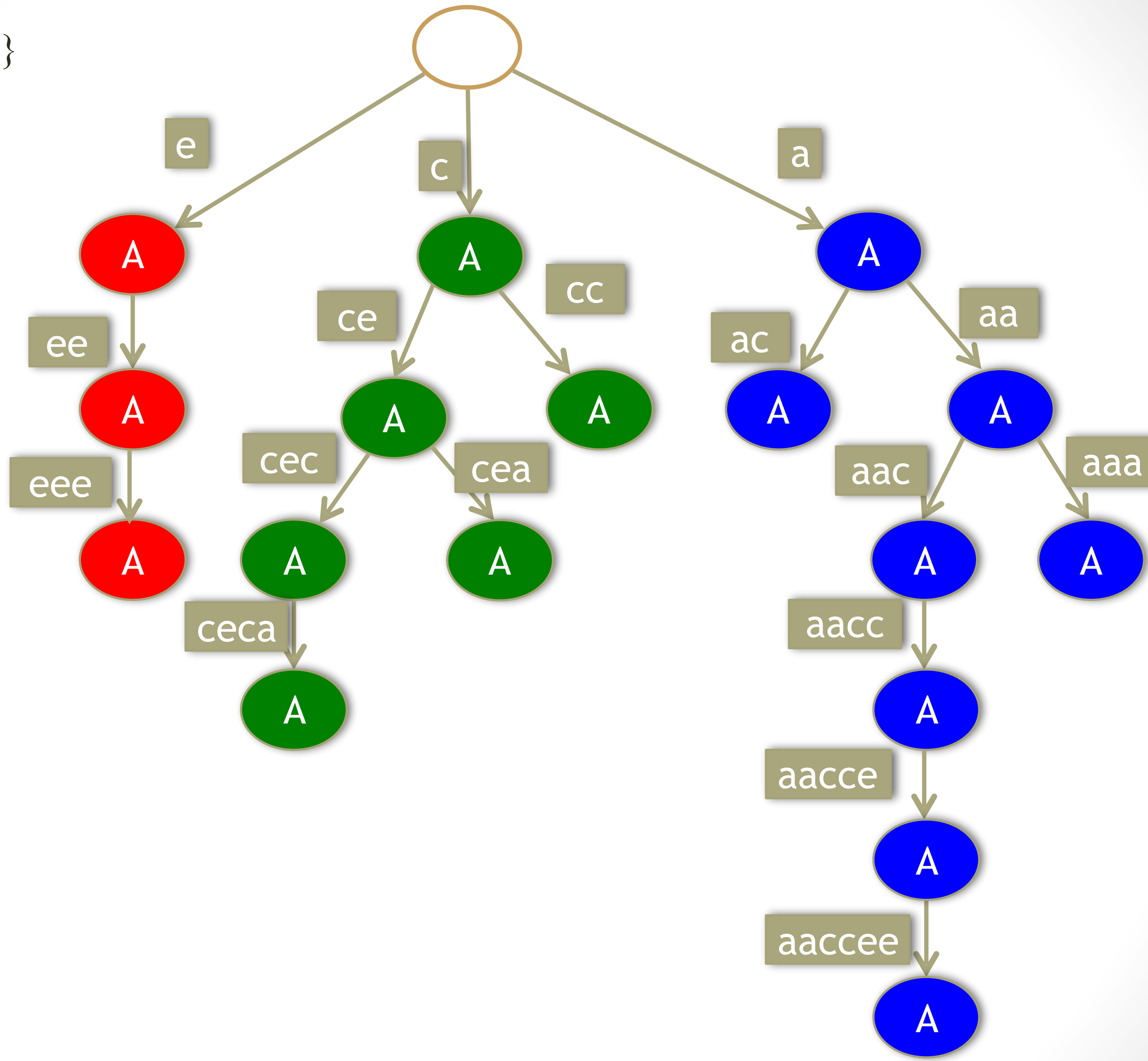


TRIE OVER ALPHABET $\{a, v, e\}$ WITH NODES FOR THE WORDS *aaa*, *aaccee*, *ac*, *cc*, *cea*, *cece*, *eee*, AND THEIR PREFIXES

A = {a, c, e}

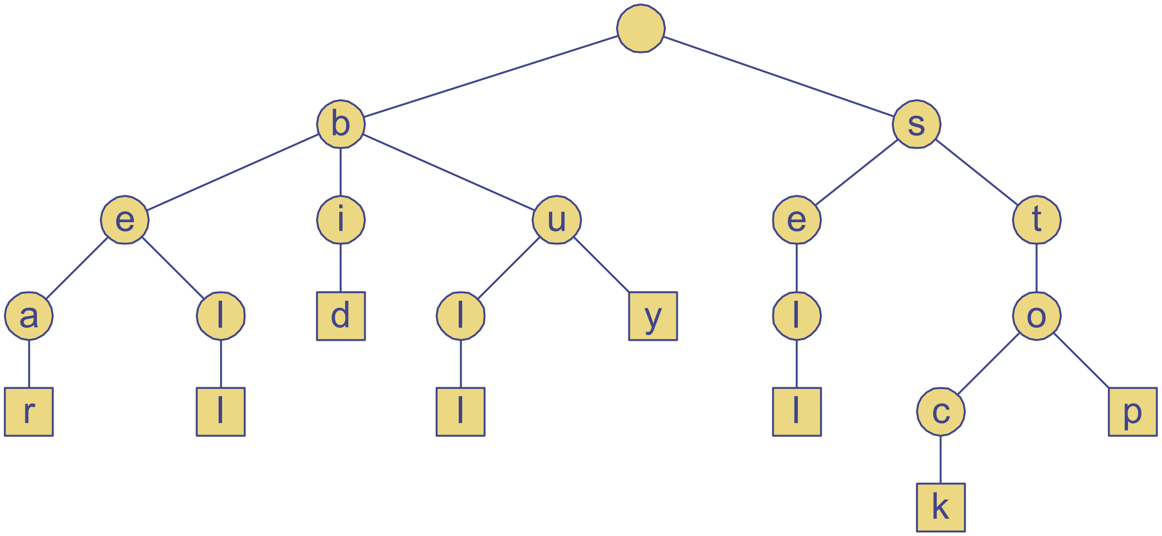
Strings:

- aaa
- aaccee
- ac
- cc
- cea
- cece
- eee



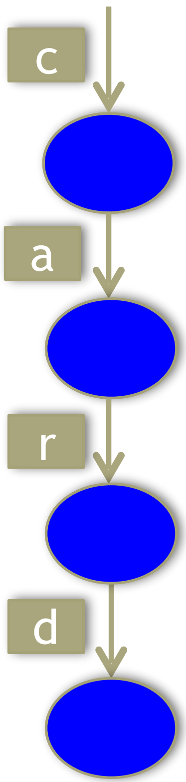
Tries Example

Example: standard trie for the set of strings
S = { bear, bell, bid, bull, buy, sell, stock, stop }



Prefix-free

- What if a "Whole" string is a prefix of other string?
- car
- card
- Are not a prefix-free.

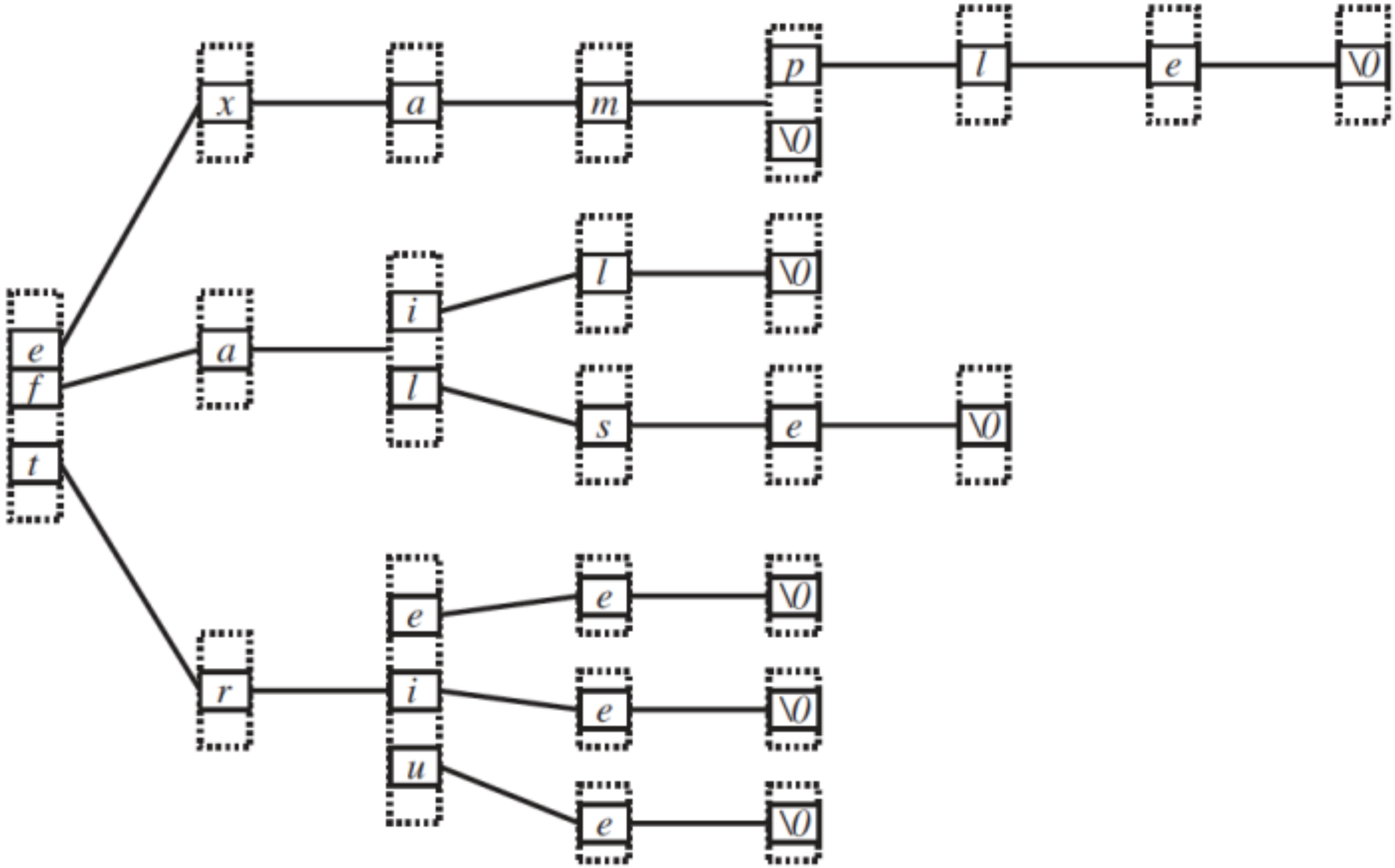


String Termination

- Strings are sequences of characters from some alphabet. But for use in the computer, we need an important further information: how to recognize where the string ends.

- There are two solutions for this:
  - We can have an explicit termination character, which is added at the end of each string, but may not occur within the string “\0” (ASCII code 0) , or
  - We can store together with each string its length.

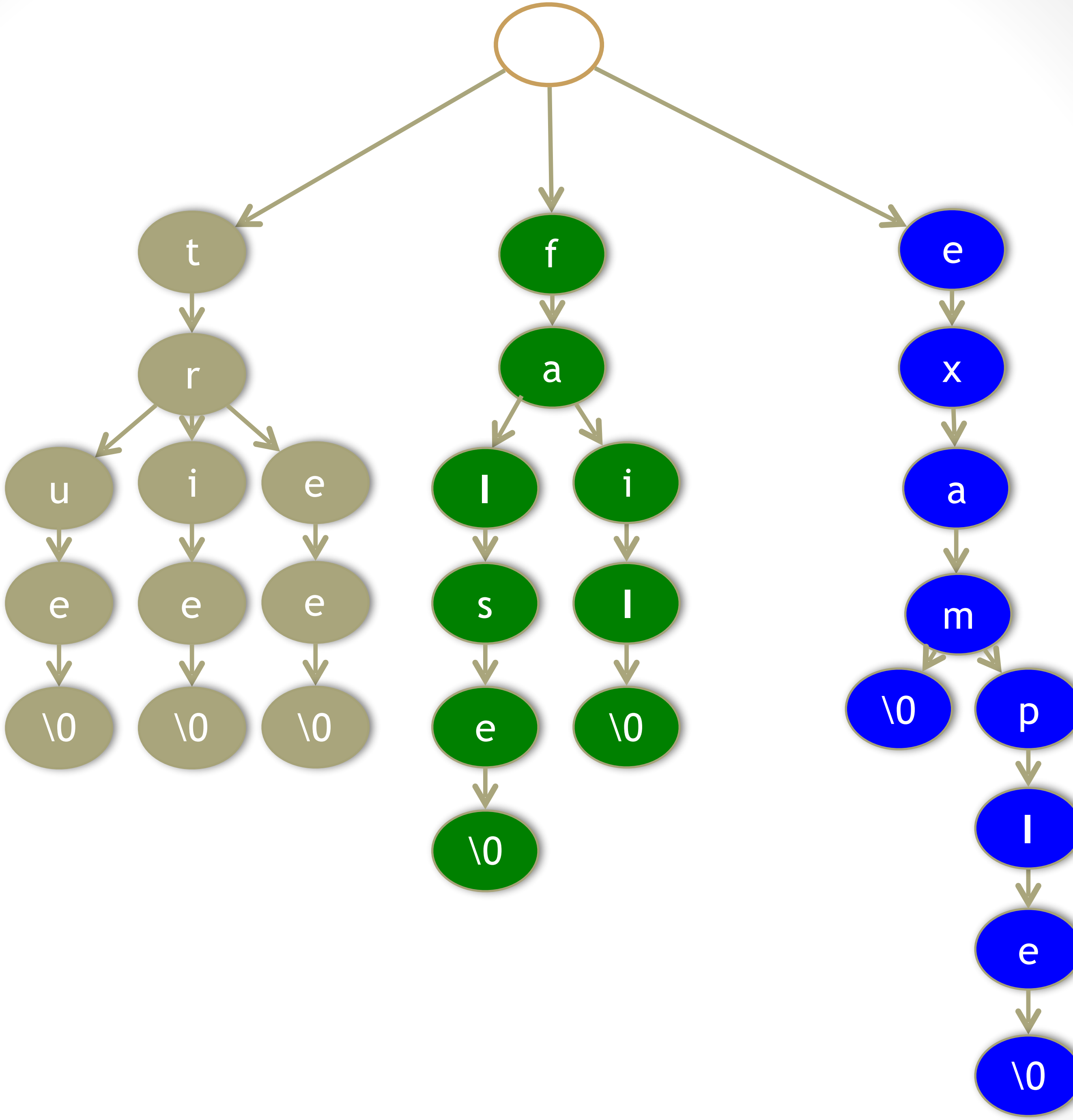
Termination Example



Trie for the Strings *exam, example, fail, false, tree, trie, true:* In Each Array Node, Only the Used Fields Shown

Strings:

- exam
- example
- fail
- false
- tree
- trie
- true



String Termination

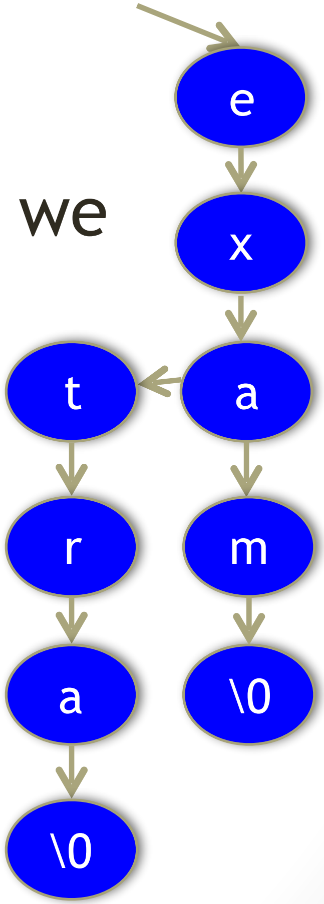
- The use of the special termination character ’\0’ has a number of advantages in simplifying code.
- It has the disadvantage of having one reserved character in the alphabet that may not occur in strings.
- There are many nonprintable ASCII codes that should never occur in a text and ’\0’ is just one of them.
- There are also many applications in which the strings do not represent text, but, for example, machine instructions.

Find, Insert and Delete

- To perform a find operation in this structure:
  1. Start in the node corresponding to the empty prefix.
  2. Read the query string, following for each read character the outgoing pointer corresponding to that character to the next node.
  3. After we read the query string, we arrived at a node corresponding to that string as prefix.
  4. If the query string is contained in the set of strings stored in the trie, and that set is prefix-free, then this node belongs to that unique string.

Find, Insert and Delete

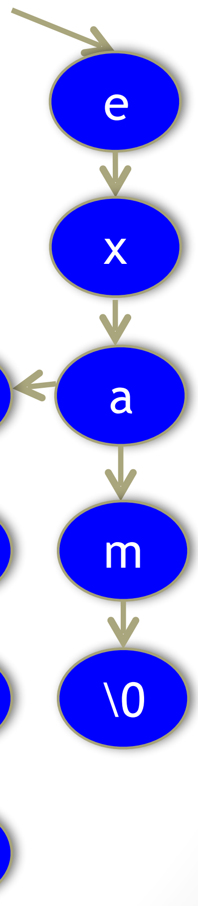
- To perform an insert operation in this structure:
  - Perform find
  - Any time we encounter a nil pointer we create a new node.



- Example:
  - Insert "extra"

Find, Insert and Delete

- To perform a delete operation in this structure:
  - Perform find
  - Delete all nodes on the path from ‘\0’ to the root of the
    - tree unless we reach a node with more than 1 child
- Example:
  - Delete “extra”



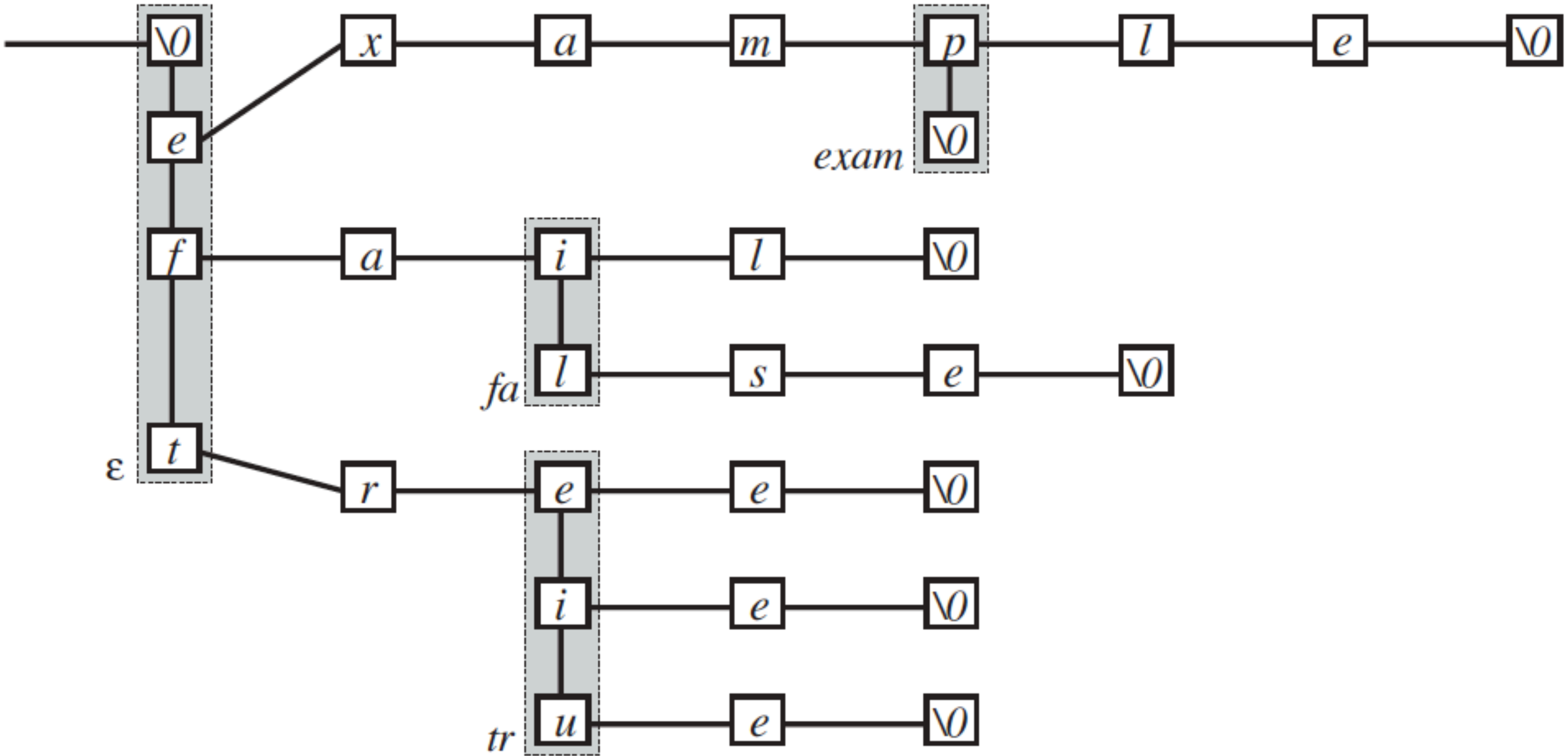
Performance

- q*: query string*
- *Find*: All the characters in the word = $O(|q|)$
- *Insert*: first *find* then *insert* an array of length $|A|$ as a node
    - $= O(|q|.|A|)$
- *Delete*: first *find* then *delete* an array of length $|A|$ as a node
    - $= O(|q|.|A|)$

Alphabet Size

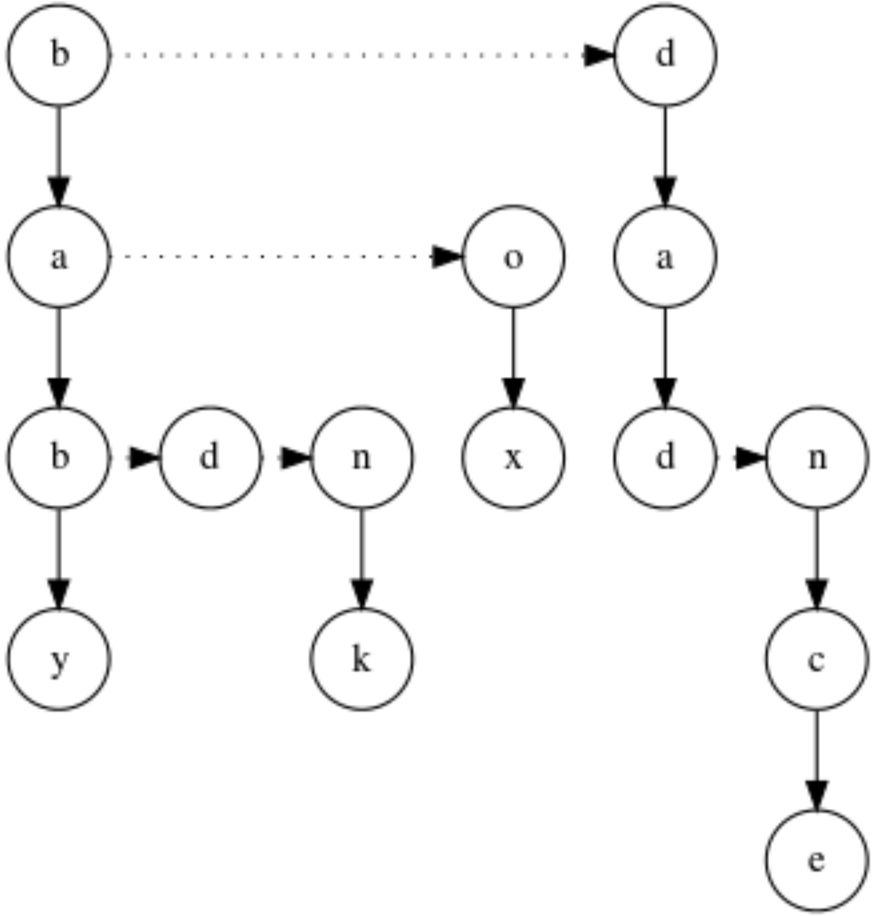
- The problem here is the dependence on the size of the alphabet which determines the size of the nodes.
- There are several ways to reduce or avoid the problem of the alphabet size.
- A simple method, is to replace the big nodes by linked lists of all the entries that are really used.

List Example



TRIE FOR THE STRINGS *exam, example, fail, false, tree, trie, true*
IMPLEMENTED WITH LIST NODES: ALL POINTERS GO RIGHT OR DOWN

A trie implemented as a doubly chained tree



Lists Performance

Theorem. The trie structure with nodes realized as lists stores a set of words over an alphabet A. It supports a `find` operation on a query string q in time $O(|A|\,\text{length}(q))$ and `insert` and `delete` operations in time $O(|A|\,\text{length}(q))$. The space requirement to store n strings $w_1, \ldots, w_n$ is $O\left(\sum_i \text{length}(w_i)\right)$.

Find, Insert and *delete:* O(|q|.|A|)

Alphabet Size

- Another way to avoid the problem with the alphabet size |A| is alphabet reduction.

$$A_1 \times \cdots \times A_k$$

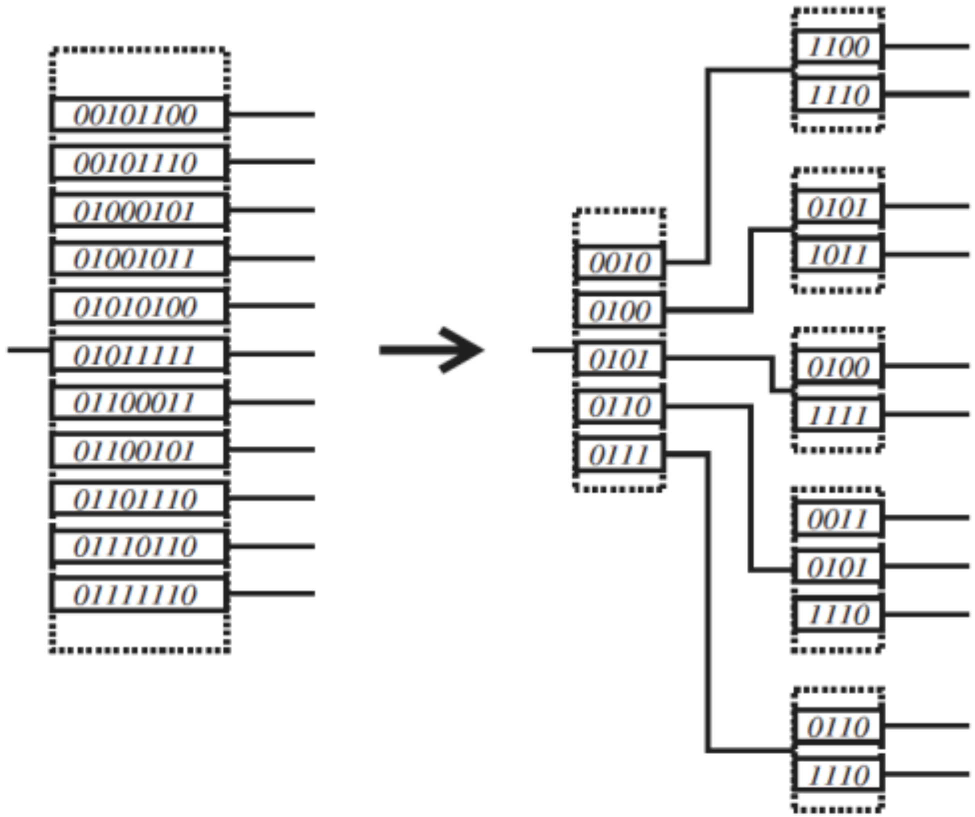
- We can represent the alphabet A as set of k - tuples from some direct product

$$\left\lceil |A|^{\frac{1}{k}} \right\rceil$$

- By this each string gets longer by a factor of k , but the alphabet size can be reduced to

Alphabet Reduction Example

- For the standard ASCII codes, we can break each 8-bit character by two 4-bit characters, which reduces the node size from 256 pointers to 16 pointers



ALPHABET REDUCTION: INSTEAD OF ONE NODE WITH 256 ENTRIES,

Reduction Performance

Theorem. The trie structure with k-fold alphabet reduction stores a set of words over an alphabet A. It supports `find` and `delete` operations on a query string q in time $O(k \text{ length}(q))$ and `insert` operations in time $O(k|A|^{\frac{1}{k}} \text{ length}(q))$. The space requirement to store n strings $w_1, \ldots, w_n$ is $O(k|A|^{\frac{1}{k}} \sum_i \text{length}(w_i))$.

Other Reduction Techniques

- The trie structure with balanced search trees as nodes:
  - *Find, insert* and *delete* Time: $O(\log|A|\,\mathrm{length}(q))$
  - Space: $O(\sum_i \mathrm{length}(w_i))$
- The ternary trie structure: nodes are arranged in a manner similar to a binary search tree, but with up to three children. each node contains one character as key and one pointer each for query characters that are smaller, larger, or equal
  - *Find* time: $O(\log n + \mathrm{length}(q))$
  - Space: $O(\sum_i \mathrm{length}(w_i))$

Ternary search tree

- Each node as three childern:
  - lo, equal, high

Words:
- ”cute”,
- “cup",
- “at",
- “as",
- “he",
- “us",
- “i"

```
        c
      / | \
    a   u   h
    |   |   | \
    t   t   e   u
  /   / |     / |
 s   p  e    i  s
```

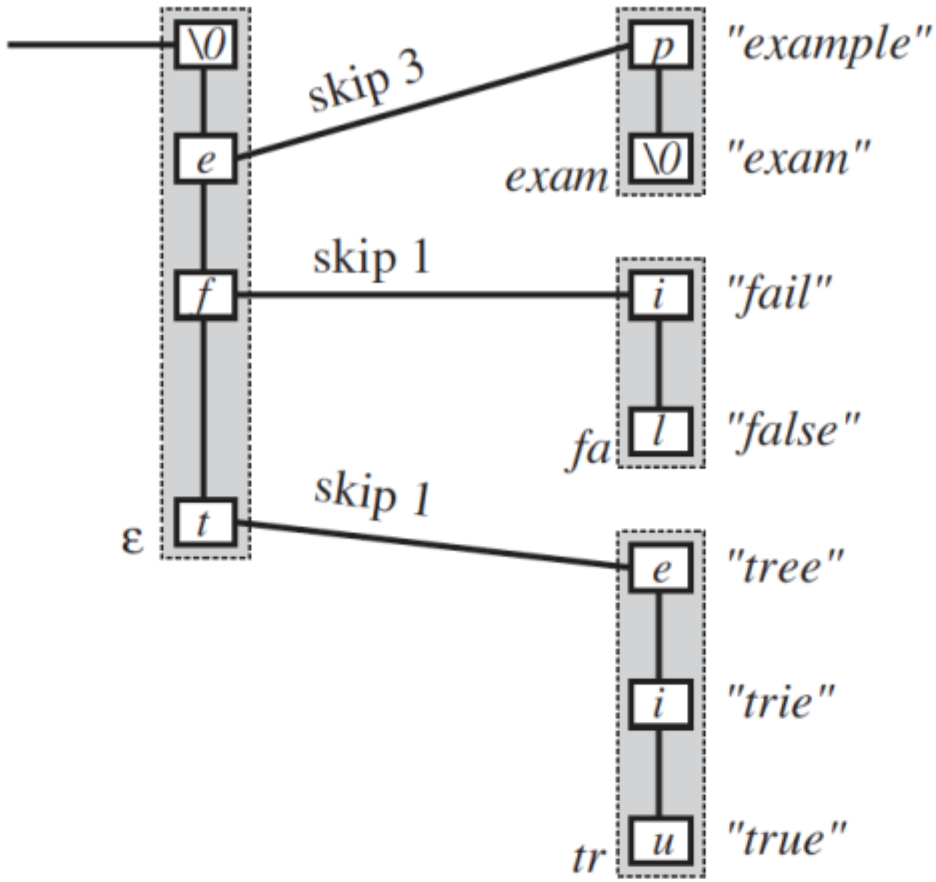
Patricia Tree

- “Practical Algorithm To Retrieve information Coded in Alphanumeric.”
- A path compression trie.
- Instead of explicitly storing nodes with just one outgoing edge, we skip these nodes and keep track of the number of skipped characters.
- The path compressed trie contains only nodes with at least two outgoing edges.

Patricia Tree

- It contains a number, which is the number of characters that should be skipped before the next relevant character is looked at.
- This reduces the required number of nodes from the total length of all strings to the number of words in our structure.
- We need in each access a second pass over the string to check all those skipped characters of the found string against the query string.
- this technique to reduce the number of nodes is justified only if the alphabet is large.

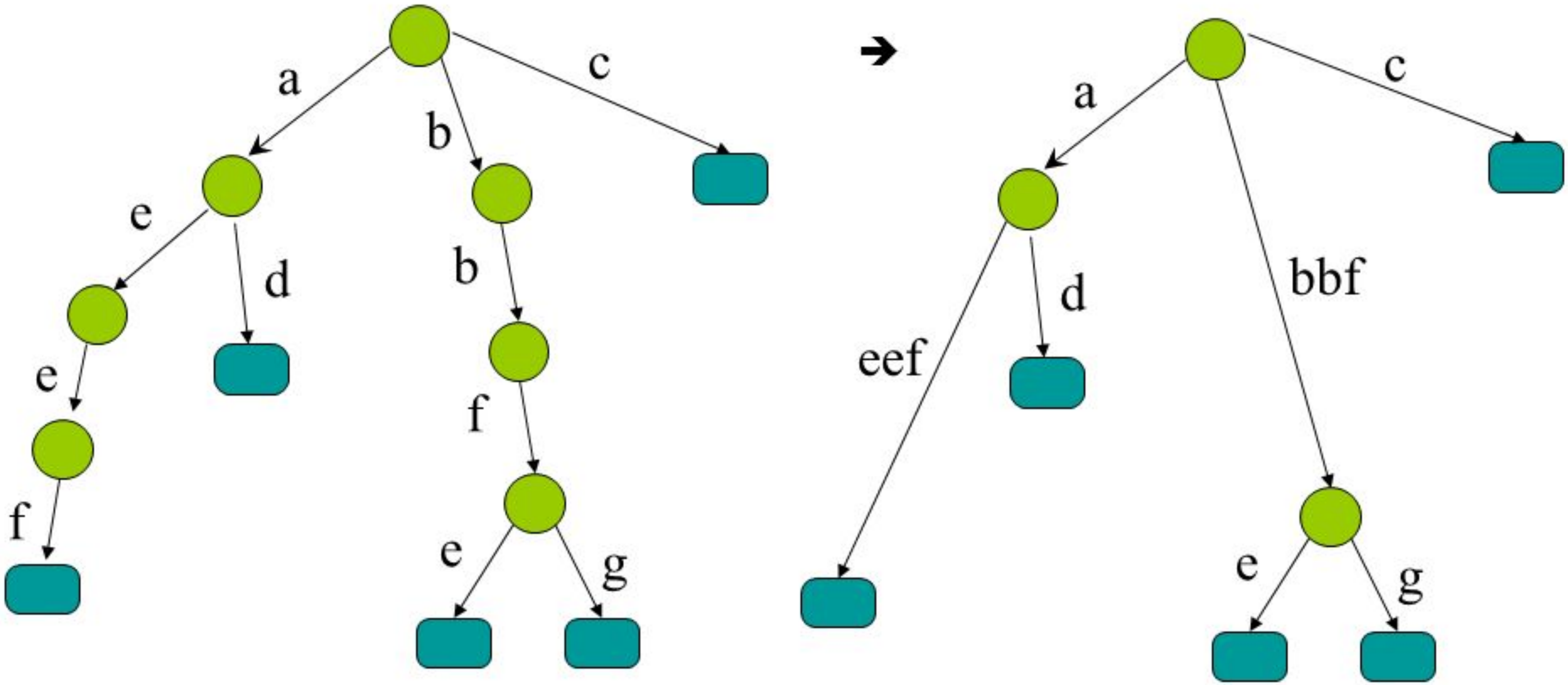
Patricia Tree: Example



PATRICIA TREE FOR THE STRINGS *exam, example, fail, false, tree, trie, true:*
NODES IMPLEMENTED AS LISTS; EACH LEAF CONTAINS ENTIRE STRING

Compressed Trie

- Compress unary nodes, label edges by strings



Patricia Tree: *Insert & Delete*

- The insertion and deletion operations create significant difficulties.
- We need to find where to insert a new branching node, but this requires that we know the skipped characters.

Theorem. The Patricia tree structure stores a set of words over an alphabet A. It supports `find` operations on a query string q in time $O(\text{length}(q))$ and `insert` and `delete` operations in time $O(|A|\,\text{length}(q))$. The space requirement to store n strings $w_1, \ldots, w_n$ is $O(n|A| + \sum_i \text{length}(w_i))$.

- One (clumsy) solution would be a pointer to one of the strings in the subtrie reached through that node, for there we have that skipped substring already available.

8.2 Dictionaries Allowing Errors in Queries

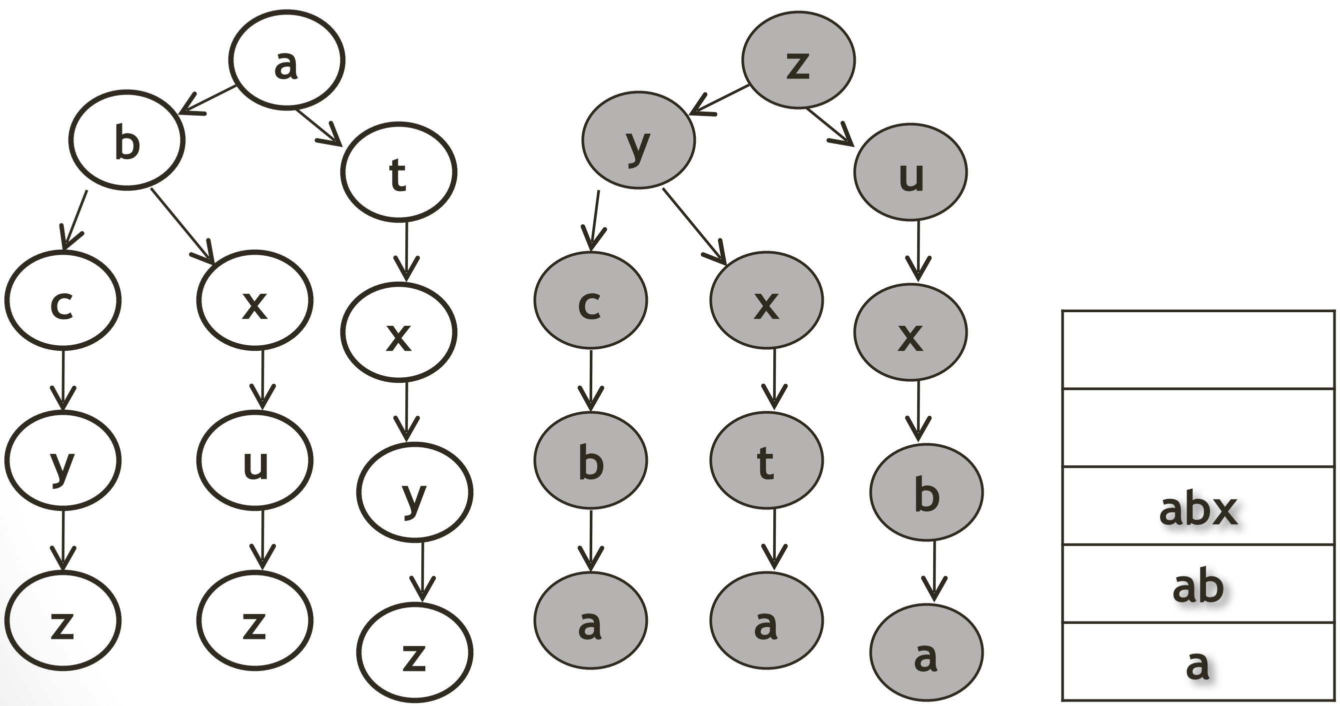
Example

Suppose that we have the following set of words:
{abcyz, abxuz, atxyz}, and a query **abxyz**

Do it yourself:
Build two tries

Example

Suppose that we have the following set of words:
{abcyz, abxuz, atxyz}, and a query **abxyz**



Prefix trie *Suffix trie* *Prefix Stack*

Candidates

…..
…..
abcyz
abxuz
abxyz

Example Cont.

The used technique:

Unless the exact match was not take place, then,

1) Go through prefix trie up to the max matched prefix. Each visited node will be pushed into a stack.
2) Go to suffix trie, and travers up to one character before the maximum prefix.
3) Then, concatenate each visited node in suffix trie with the stack entries.
4) Now, all candidate words are generated, and they will be used in find operation
   over the already built search-tree.

Times Complexities

Building the structure :

a) For a given word w_i, we can generate all the node pairs $O(\text{length}(w_i))$

So, for all words (w_n), we can generate all node pairs in time $\sum w$

b) finding in the search-tree costs only O(Log $\sum w$)

c) Total time = O($\sum w$ Log $\sum w$)

For each query, the worst case will $O(\text{length}(q) \log \Sigma_w)$

What is the best case?

Supporting Insert/Delete

Do it yourself:
Find “abcxyz” with one delete

Do it yourself:
Find “abyz” with one insert

8.3 Suffix Trees

Problem: Find a text within another

- Previous problem:
  - Find text in a set of texts

- New problem:

S : GTTATAGCTGATCGCGGCGTAGCGG

T: AGCT

Solution:
Reduce to previous problem

Why $O(\text{length}(s)^2)$?

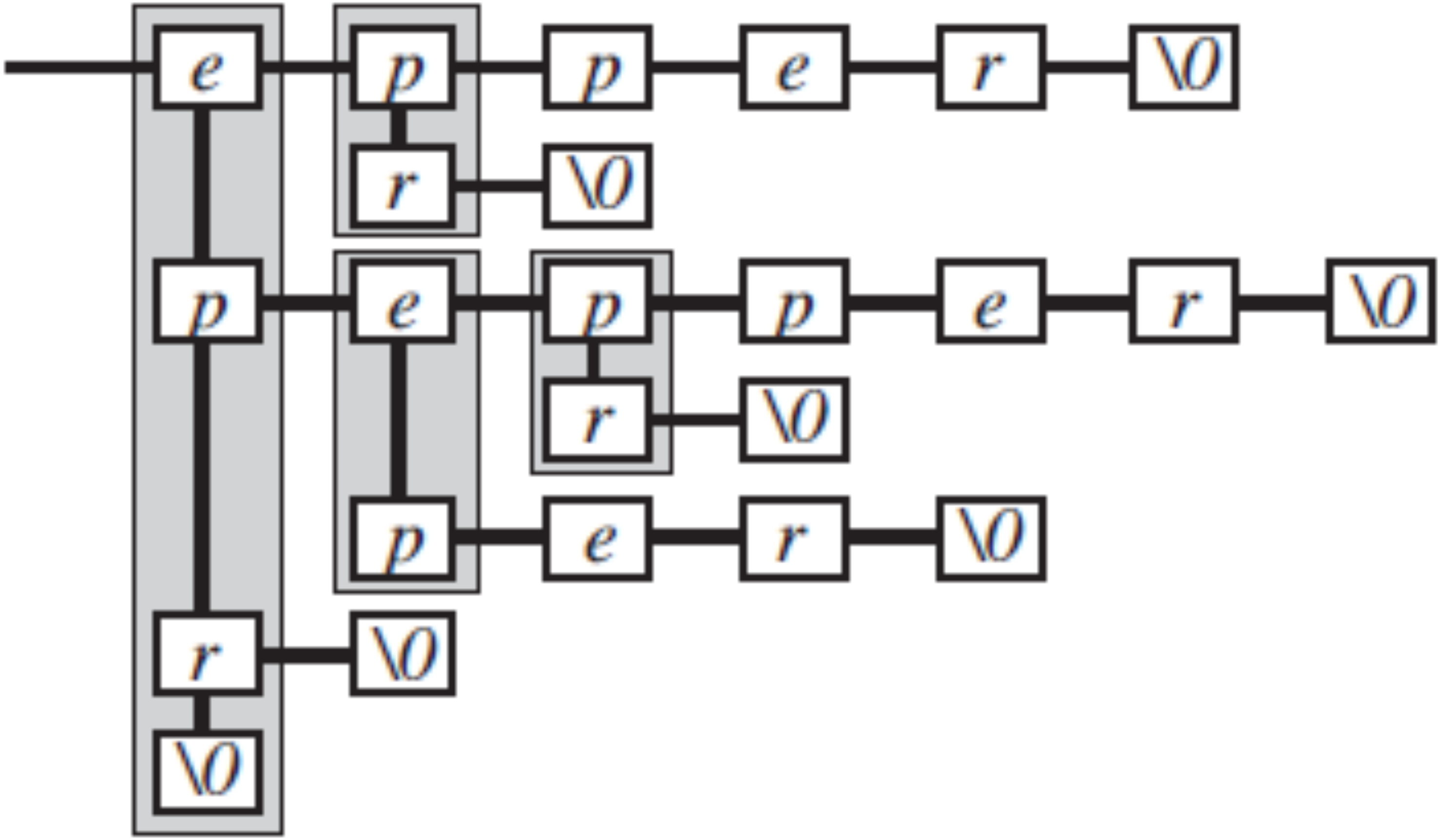
Build a **trie** containing all **suffixes** of a text **S**

```
S : GTTATAGCTGATCGCGGCGTAGCGG
    GTTATAGCTGATCGCGGCGTAGCGG
     TTATAGCTGATCGCGGCGTAGCGG
      TATAGCTGATCGCGGCGTAGCGG
       ATAGCTGATCGCGGCGTAGCGG
        TAGCTGATCGCGGCGTAGCGG
         AGCTGATCGCGGCGTAGCGG
          GCTGATCGCGGCGTAGCGG
           CTGATCGCGGCGTAGCGG
            TGATCGCGGCGTAGCGG
             GATCGCGGCGTAGCGG
              ATCGCGGCGTAGCGG
               TCGCGGCGTAGCGG
                CGCGGCGTAGCGG
                 GCGGCGTAGCGG
                  CGGCGTAGCGG
                   GGCGTAGCGG
                    GCGTAGCGG
                     CGTAGCGG
                      GTAGCGG
                       TAGCGG
                        AGCGG
                         GCGG
                          CGG
                           GG
                            G
```

$m(m+1)/2$ chars

nodes

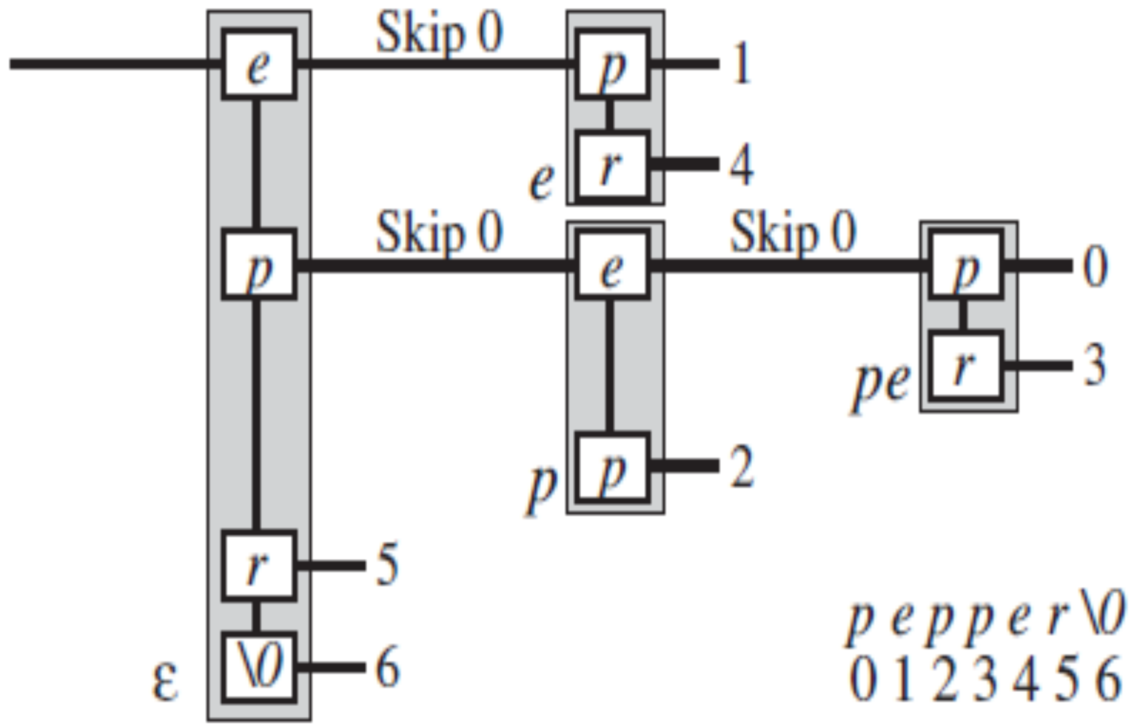
Suffix tree



TRIE OF THE SUFFIXES OF *pepper*

A more Compact Representation

No need to store all suffixes explicitly, but can encode each by a beginning and end address in the long string S. ➔ O(length(s)) nodes representation.

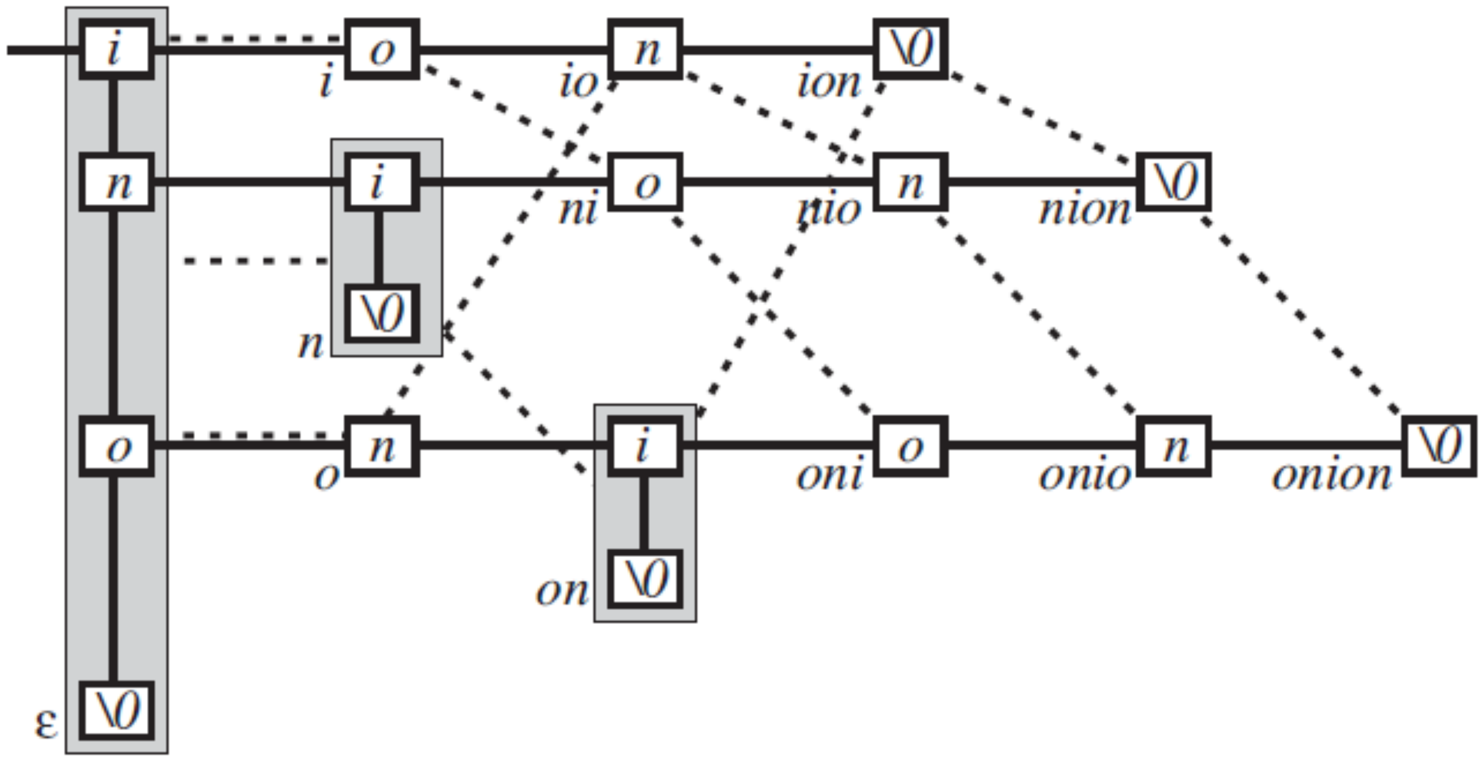


PATRICIA TREE OF THE SUFFIXES OF *pepper*:

THE LEAF NUMBERS GIVE THE STARTING POSITIONS OF THE SUFFIXES

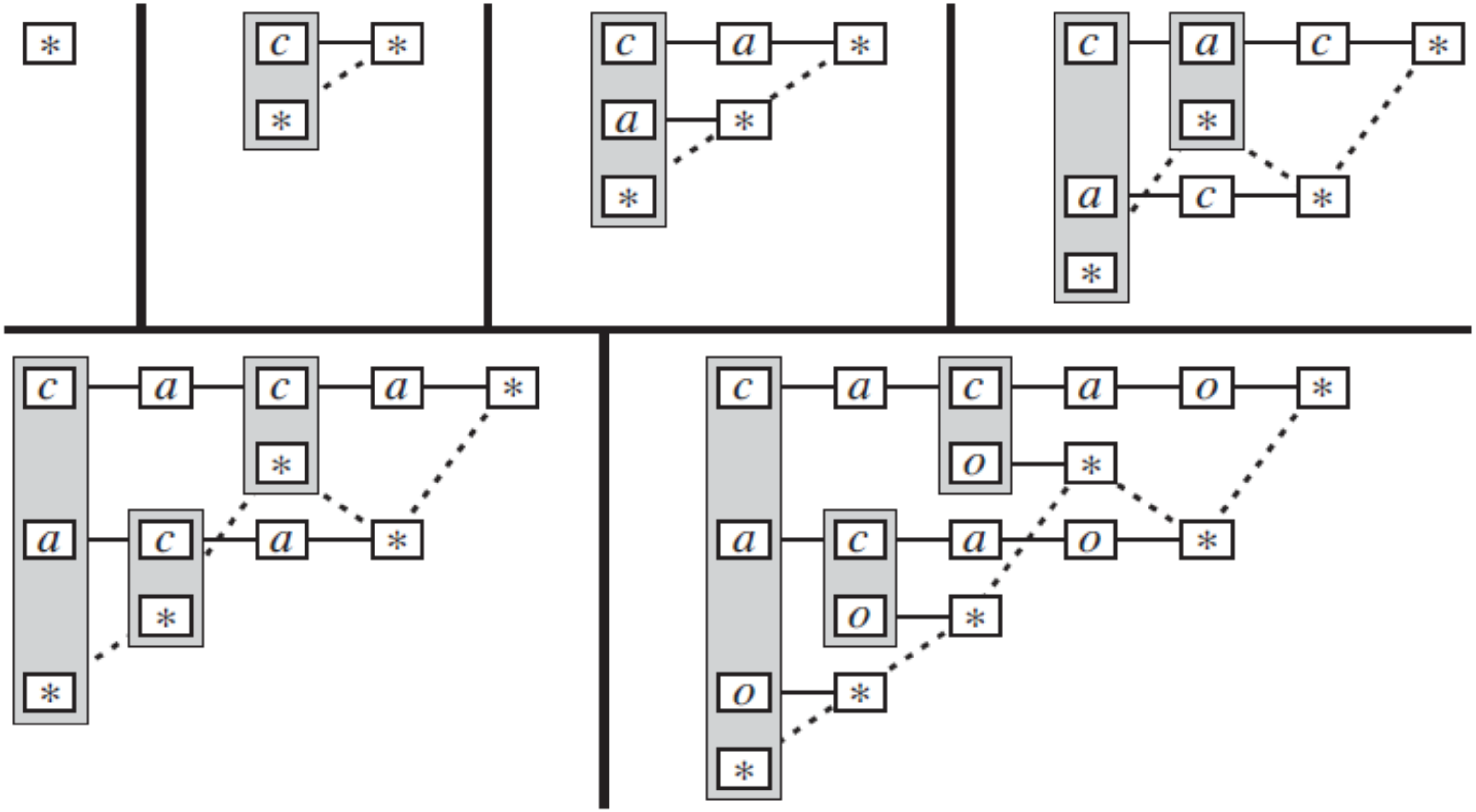
Suffix Links

The suffix link is an extra pointer which points from a node representing a string a0 . . .ak to the node representing the string a1 . . . ak, that is, its suffix after deleting the first character.



TRIE OF THE SUFFIXES OF *onion* WITH SUFFIX LINKS

Building the Structure



INCREMENTAL CONSTRUCTION OF TRIE OF THE SUFFIXES OF *cacao*:
THE $*$-NODES MARK THE CURRENT END; THEY FORM THE BOUNDARY PATH

Building the Structure Algorithm

Algorithm 1. **It takes O(n^2), but It could be O(n) if compressed path used.**

$r \leftarrow top;$

while $g(r, t_i)$ is undefined **do**

create new state r' and new transition $g(r, t_i) = r'$;

if $r \neq top$ **then** create new suffix link $f(oldr') = r'$;

$oldr' \leftarrow r'$;

$r \leftarrow f(r)$;

create new suffix link $f(oldr') = g(r, t_i)$;

$top \leftarrow g(top, t_i)$.

For more details, please see the original paper "On-line construction of suffix trees.".(Esko Ukkonen)

8.4 Suffix Arrays

Definition

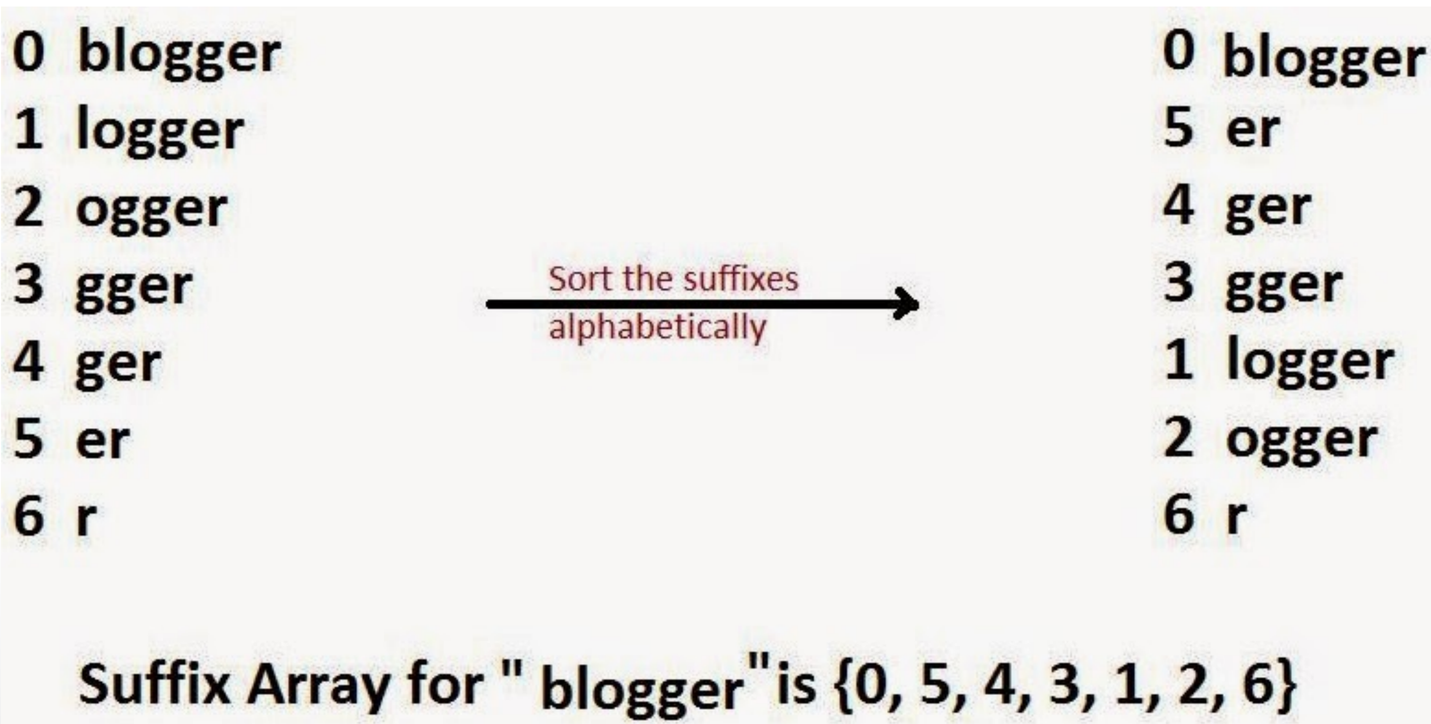
The suffix array is an alternative structure to the suffix tree that was developed by Manber and Myers (1993). It preprocesses a long string and then answers for a query string whether it occurs as substring in the preprocessed string.

Possible Advantages:

- Its size does not depend on the size of the alphabet.
- It offers a quite different tool to attack the same type of string problems.
- straightforward implementation and it is said to be smaller than suffix trees

The Underlying Idea

Sort suffixes
Find in suffixes by binary search



Find by binary search

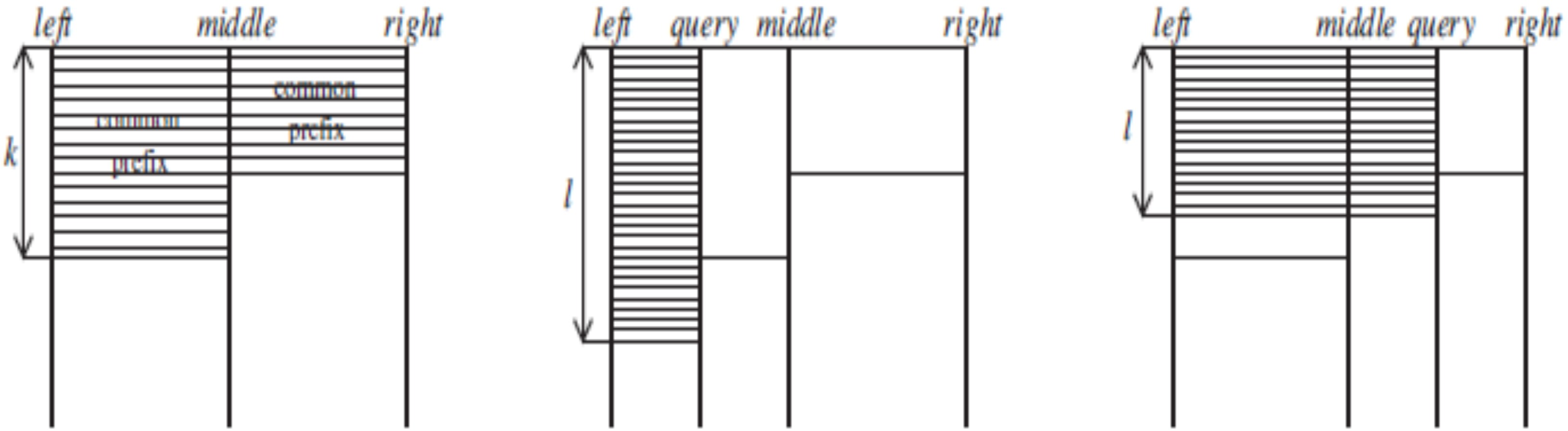
We need $O(\log \text{length}(s))$ lexicographic comparisons between q and some suffix of s. Without additional information, each comparison takes $O(\text{length}(q))$ time for a total of **$O(\text{length}(q) \log \text{length}(s))$.**

Can we do better??

If we have LCP

- LCP: Longest common prefix
- Binary search:
  - We have q (query)
  - low: lower bound
  - high: higher bound
  - mid: middle

Using Longest Common Prefix (LCP)



COMMON PREFIX LENGTHS IN THE BINARY SEARCH AND THE POSITION OF *query* RELATIVE TO *middle*

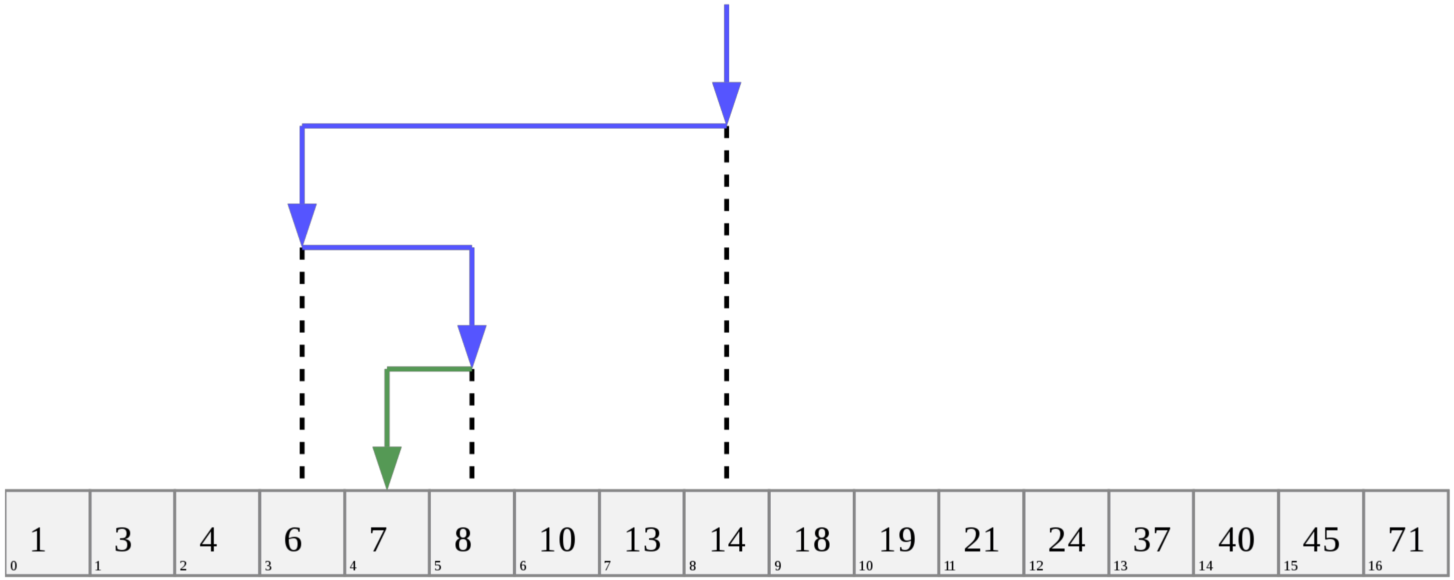
Using Longest Common Prefix (LCP)

if *left* and *q* share the first *l* characters, with *l* < *k*, then the query string cannot be between *left* and *middle*. And by the same argument, if *l* > *k*, then *middle* cannot be between *left* and *q*.

So if we have the numbers *k* and *l*, we can decide the outcome of the comparison in that step of binary search without looking at the string *q* unless *l* = *k*.

If *l* = *k*, we compare and update the value of (l) up to length (q) of times.

How to keep LCP in memory?



- n^2 -> O(n) cells

Time Complexity (2)

Theorem. An array of pointers to n lexicographically sorted strings, together with two arrays of n integers each, containing the common prefix length information, allows to find for a query string q whether it is prefix of any of these strings in time $O(\text{length}(q) + \log n)$.

BUT, How can we build the structure?

Building the structure

Kärkkäinen and Sanders (2003)

1. Construct the suffix array A 12 of the suffixes starting at positions i (mod 3) = 0.
   This is done by a recursive call of the skew algorithm for a string of two thirds the length.

2. Construct the suffix array A0 of the remaining suffixes.

3. Merge the two suffix arrays into one.

Full example at :
http://www.mi.fu-berlin.de/wiki/pub/ABI/SS13Lecture3Materials/script.pdf

Building the structure

Kärkkäinen and Sanders (2003)

This example shows the general idea:

Let us say S = CSWESTERN , 9 Characters.

0	1	2	3	4	5	6	7	8
C	S	W	E	S	T	E	R	N

Group 0 : if (i mod 3= = 0), Group 1: if (i mod 3==1), and Group 2: if (i mod 3= =2)

GROUP 0	CSW (i=0)	EST (i=3)	ERN (i=6)
GROUP 1	SWE (i=1)	STE (i=4)	RN$ (i=7)
GROUP 2	WES (i=2)	TER (i=5)	N$$ (i=8)

Building the structure

Kärkkäinen and Sanders (2003)

Take two part; Group 1 and Group 2, handle them **recursively**.
BY using **Radix Sort**, we have ordered list of group 12 suffixes.

GROUP 0	CSW	EST	ERN
GROUP 12	N$$	RN$	STE
	SWE	TER	WES

Construct the suffix array of Group 12.

Construct the suffix array of Group 0.
Then, merge.

Time Complexity of Building

Kärkkäinen and Sanders (2003)

Running Time

$$T(n) = O(n) + T(2n/3)$$

time to sort and merge

array in recursive calls is 2/3rds the size of starting array

Solves to $T(n) = O(n)$:

- Expand big-O notation: $T(n) \leq cn + T(2n/3)$ for some c.
- Guess: $T(n) \leq 3cn$
- Induction step: assume that is true for all $i < n$.
- $T(n) \leq cn + 3c(2n/3) = cn + 2cn = 3cn$ □

Suffix Arrays in Few Words

Theorem. The suffix array structure is a static structure that preprocesses a string s and supports substring queries. This structure can be built in time $O(\text{length}(s))$, requires space $O(\text{length}(s))$, and supports `find_string` queries for a string q in time $O(\text{length}(q) + \log(\text{length}(s)))$.

THANK YOU