Algorithms in bioinformatics (CSI 5126)[1]

Marcel Turcotte
(turcotte@site.uottawa.ca)

School of Information Technology and Engineering
University of Ottawa
Canada

October 2, 2009

[1] Please don't print these lecture notes unless you really need to!

Plan

- String algorithms
  - Motivation
  - Notation
  - Glimpse of Boyer-Moore string matching algorithm
  - Suffix trees
    - Definition
    - Naïve construction algorithm
    - Daniela Cernea's Java library

Motivation

- The simplest model of a macromolecule is a string. Yet, this level of abstraction is sufficient for a considerably large number of applications
- The size of the biological databases has been doubling every 12 to 18 months for the last few years
- Millions of queries are made to (static) databases
- **Instances of exact and approximate string matching problems are solved as sub-tasks of several bioinformatics applications**, such as the DNA assembly process
- Natural transition between approximate string matching and molecular sequence alignments

Examples of Problems on Strings

1. Exact string matching: finding all occurrences of a string in a text
2. Approximate string matching: find all positions in a text where a pattern occurs, allowing for a certain number of mismatches
3. Longest common substring
4. *Sequence* comparison: highlight similarities and differences between two *sequences*
5. Regular expression matching
6. Structural pattern matching: motif discovery, like repeats, tandems and palindromes

$\Rightarrow$ Efficient data structures, such as suffix trees, are often at the heart of efficient algorithms.

Notation

A string, S, is an ordered list of characters written contiguously from left to right.
$|S|$ denotes the length of the string S.
ϵ represents the empty string.
$S(i)$ denotes the ith character of S.
The index 1 denotes the first character of S.

Notation (cont.)

The set of all characters is called the *alphabet*, often denoted Σ or $\mathcal{A}$. In all our applications the alphabet is a finite set of symbols, although it varies in size from one application to the other:

- DNA alphabet:
  $\{A, C, G, T\}$
- Protein alphabet:
  $\{A, .., Z\} \setminus \{B, J, O, U, X, Z\}$
- Solvent accessibility, Inside (hydrophobic), Surface (hydrophilic):
  $\{I, S\}$
- Secondary structure states of proteins, Helix, Strand and Coil:
  $\{H, E, C\}$

Notation

$S[i..j]$ denotes the (contiguous) **substring** of S that starts at position i and stops at position j, $S(i)S(i+1)\ldots S(j)$; also called a **factor**.
$S[1..i]$ is the **prefix** of S.
$S[i..|S|]$ is the **suffix** of S.
A substring, prefix or suffix is **proper** if it's not the entire string (and it is not empty).
We say that S is a **subsequence** of T, if there exists an increasing set of indices of T, $i_1 < i_2 < \ldots < i_m$, such that $S = T[i_1]T[i_2]\ldots T[i_m]$. In other words, the string S can be obtained by deleting zero or more characters of T.
E.g. *tie* is a subsequence of *otherwise*.
We say that two characters **match** if they are the same; otherwise we say it's a **mismatch**.

Notation (cont.)

Let P denote a pattern (query, for now a string) and T be a text (think of it as a database), in general $|P| << |T|$.

Initial motivation

Problem: Given a pattern P and a text T, determine if P occurs in T.
(`boolean find(P, T)`)
Problem: Given a pattern P and a text T, find all occurrences of P in T. (`int[] findall(P, T)`)
P = string
T = Algorithms on text (strings) have long been studied in computer science, and computation on molecular sequence data (strings) is at the heart of computational molecular biology. Present and potential algorithms for string computation provide a significant intersection between computer science and molecular biology.
How do you approach such problem?

Naïve algorithm

Move a window of size $|P|$ is moved along the text (T).
In the worst case, for every starting location, $1 \ldots |T|$, all the symbols of the pattern ($|P|$) must be considered. Therefore requiring $|T| \times |P|$ comparisons.

Naïve algorithm (cont.)

```
public static int findall( String p, String t ) {
    int lp = p.length(), lt = t.length(), count = 0;
    for ( int pos=0; pos <= lt-lp; pos++ ) {

        int offset = 0;
        boolean done = p.charAt( offset ) != t.charAt( pos+offset );

        while ( ! done ) {
            offset++;
            if ( offset == lp ) {
                done = true;
                count++;
            } else {
                done = p.charAt( offset ) != t.charAt( pos+offset );
            }
        }
    }
    return count;
}
```

Discussion

In practice, what behaviour do you expect?

Discussion

In practice, what behaviour do you expect?

- First, $|P| << |T|$.
- But also, with probability $(1 - \frac{1}{|\Sigma|})$ the algorithm will skip the **while** loop for each iteration of the exterior **for** loop (simplified reasoning).
- Assuming random pattern and text, one would expect to find 1 complete exact match every $|\Sigma|^{|P|}$ positions.
- What is the maximum length of a pattern that you would expect to find at least once in the human genome? $\log_4 3,000,000,000 \sim 16$.
- Conclusion: you'd expect the inner loop stops rapidly.
- How do you speed it up?

Speeding up

There are two fundamentally different approaches:

- Pre-processing P (e.g. Boyer-Moore)
- Pre-processing T (e.g. Suffix Trees)

$\Rightarrow$ But what do you mean by pre-processing? Let's consider the Boyer-Moore algorithm first, before comparing P and T, we are willing to spend time and space, analyzing P, pre-calculating indices that we know will be useful later and will reduce the total number of comparison and shift operations needed. In the case of suffix trees, we are willing to spend time and space on the analysis of T.

Boyer-Moore: ideas

When comparing two strings, P and T, the Boyer-Moore algorithm proceed from right to left:

```
T: xpbctbxabpqxctbpq
P:   tpabxab
       *^^^^
       tpabxab
```

Once a mismatch has been found, it applies one of 2 rules to shift the position of the pattern with respect to the text (instead of systematically shifting the pattern one position to the right, as the naïve algorithm does).

- Bad character rule;
- (Strong) good suffix rule.

Bad character rule

Definition $R(x)$ is the rightmost occurrence of the character x in P. $R(x) = 0$ if x does not occur in P.
Preprocessing. Calculate $R(x)$ $\forall x \in \Sigma$ (alphabet); this necessitates $O(n)$ operations.
P = tpabxab

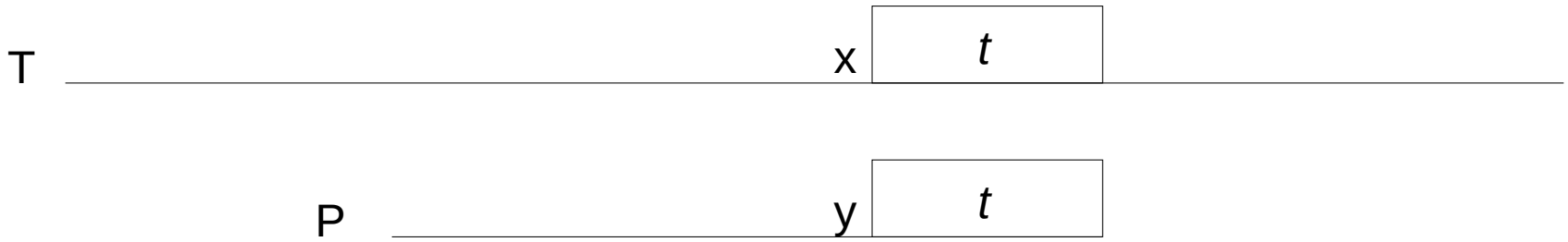
$$\begin{aligned} \Sigma &= \{a, b, p, t, x\} \\ R &= \{6, 7, 2, 1, 5\} \end{aligned}$$

Bad character rule (cont.)

```
T: xpbctbxabpqxctbpq
P:   tpabxab
       *^^^^
       tpabxab
```

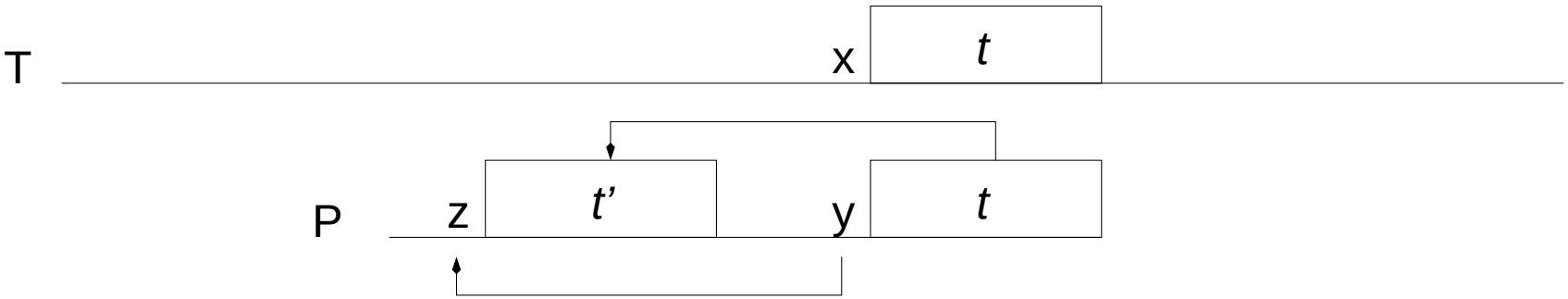
The naïve algorithm would shift the pattern one position to the right, comparing the two strings again.
However, we could have known in advance that a mismatch would occur because the location of the right most occurrence of `t` in P is on the left hand side of the symbol `p` (the symbol that will be aligned with `t` of T when P is shifted one position to the right).

Idea behind the strong suffix rules



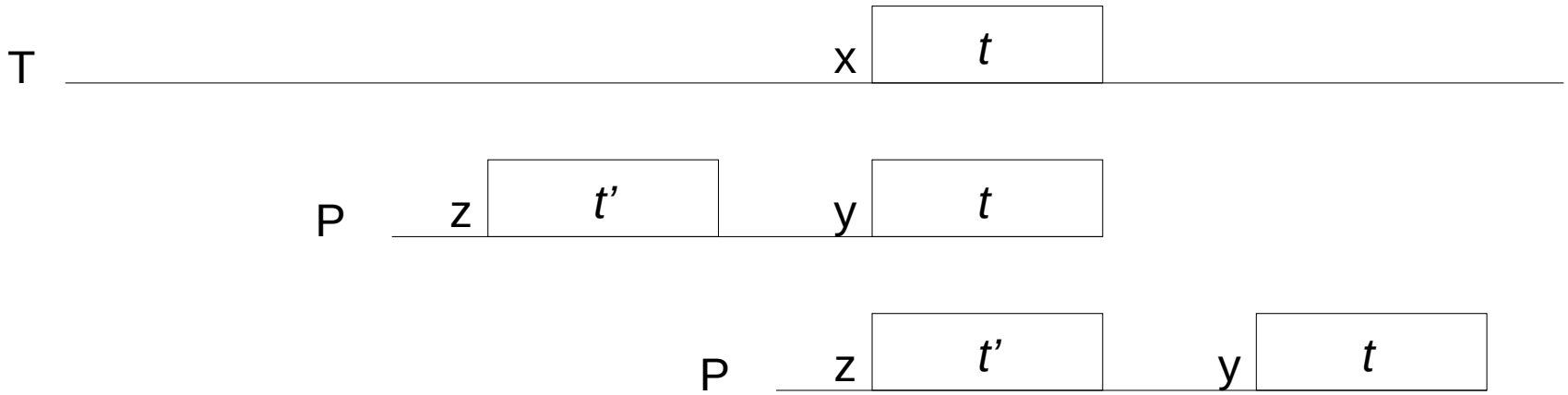
$\Rightarrow$ The boxes labeled $\mathtt{t}$ are identical substrings, the characters $\mathtt{x}$ of T and $\mathtt{y}$ of P are distinct (i.e. the first mismatch).

Idea behind the strong suffix rules



$\Rightarrow$ t' is a substing of P that matches the suffix t, furthermore, characters y and z are distinct.

Idea behind the strong suffix rules



$\Rightarrow$ shift P so that t' now aligns with t in T, since characters y and z are distinct, z has actually a chance of matching x.

Remarks

It can be shown that the pre-processing time to calculate the index for the bad character rule and the strong suffix rule can be done in linear time w.r.t. the size of the pattern.
The resulting algorithm runs in expected linear time w.r.t. the size of the database.
Boyer-Moore method can be extended so that in the worst-case it also runs in linear time.
Other well known algorithms are Knuth-Morris-Pratt, Apostolico-Giancarlo and Aho-Corasick to name a few.
There is an extension of the Knuth-Morris-Pratt algorithm which is advantageous in real-time processing.

Suffix Trees (ST)

With suitable extensions, Boyer-Moore and other exact string matching algorithms run in linear time with respect to the size of the database (T, i.e. text), and its preprocessing necessitates order of $|P|$ operations.
Suffix trees algorithms run in linear time with respect to the size of the query (P, i.e. pattern) but necessitates $\mathcal{O}(|T|)$ preprocessing time/space.
In most applications, $|P| << |T|$.
Search is done in $\mathcal{O}(|P|)$, i.e. independent of the size of the database (!); once the preprocessing has been done.
[ST have many more applications such as finding the longest common substring of two strings or finding the longest repeat. More later.]

Substring problem

Original problem. "One is first given a text T of length m. After $\mathcal{O}(m)$, or linear, preprocessing time, one must be prepared to take in any unknown string S of length n and $\mathcal{O}(n)$ time either find an occurrence of S in T or determine that S is not contained in T".
In practice, the preprocessing takes time and necessitates a lot of disk space, it is therefore used in situations where the database is static and the queries are frequent.
The preprocessing requires $\mathcal{O}(m)$ memory, however, the constant can be as large as a hundred, with the best known implementation (and most complex one) requiring 28 bytes per input byte, i.e. the suffix tree of a 3 Gbytes string would require 84 Gbytes.

Chronology

Weiner (1973); first linear time algorithm for constructing suffix trees. Declared **"algorithm of the year"** by Knuth.
McCreight (1976); presents a simpler algorithm which is also more space efficient.
Ukkonen (1995); this linear algorithm also allows for *online* left-to-right processing and is **conceptually easier to understand** than the previous two methods.
(method of choice)
[Recent developments (last 3–5 years) with **suffix arrays** imply that suffix trees are mainly used as conceptual and/or didactic tools.]

$\Rightarrow$ Our discussion follows Gusfield (1997).

Observation/Motivation

A string S occurs at position i of T iff S is the prefix of the ith suffix of T.

```
 1 mississippi
 2  ississippi
 3   ssissippi
 4    sissippi
 5     issippi
 6      ssippi
 7       sippi
 8        ippi
 9         ppi
10          pi
11           i
```

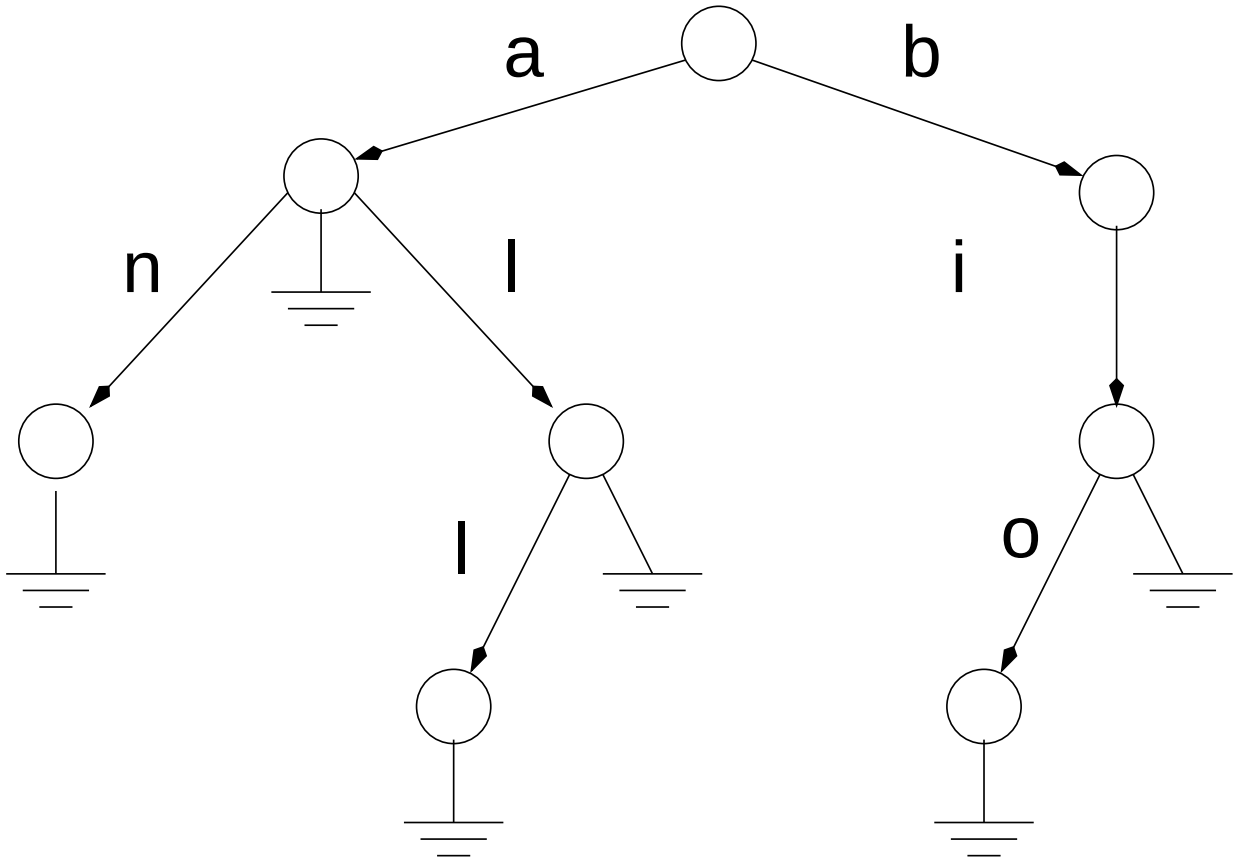
E.g. there are two occurrences of `issi`, positions 2 and 5.

A related topic: Trie, keyword-tree, $\mathcal{A}^+$-tree

The name *trie* comes the word re**trie**val. A trie is a multi-way tree used to store strings (or key values of varying sizes).
A trie is built in such a way that **all the strings sharing a common prefix are represented with a single path** from the root to an internal node representing the prefix, and all the descendants of this node represent all the possible suffixes.

A related topic: Trie, keyword-tree, $\mathcal{A}^+$-tree (cont.)

Here is the trie for the words: a, an, al, all, bi and bio.



$\mathcal{A}^+$-tree

Given an alphabet, $\mathcal{A}$, an $\mathcal{A}^+$-tree is a finite rooted tree such that:

1. the edges of the tree are labelled with non-empty strings over $\mathcal{A}$;
2. the labels of the outgoing edges of a node all start with a different letter.

Corollary: all internal nodes have up to $|\mathcal{A}| + 1$ children; one child for each letter of the alphabet plus one to represent the end of a string.
In an $\mathcal{A}^+$-tree, the nodes are allowed to have a single child.
Given a trie, to determine if a string occurs in the tree, it suffice to find a path from the root to a leaf such that the concatenation of the labels spells out the string.

Suffix Tree

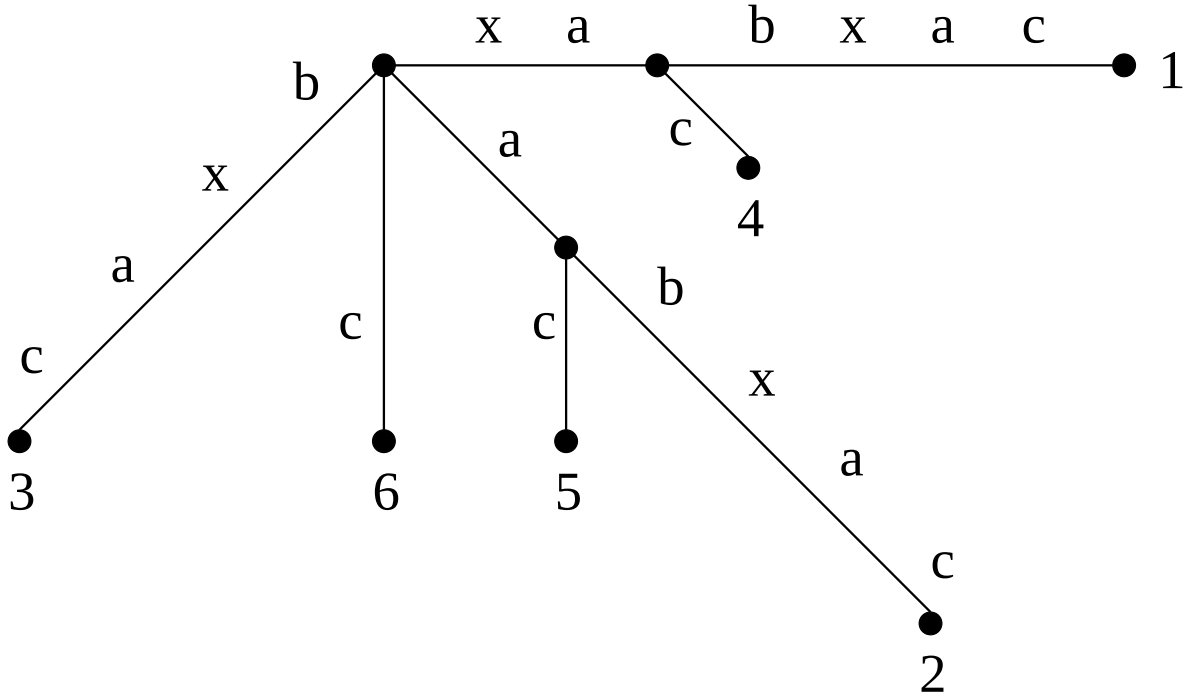
A *suffix tree* is a (PATRICIA[2]) trie in which 1) **all the suffixes of a given string S occur** and 2) **is compact**.
An $\mathcal{A}+$-tree is **compact** if all the nodes are branching nodes (2 or more successors) or a leaf; except for the root, which is allowed to have a single successor.
The concatenation of all the arc labels from the root to a leaf constitutes a suffix of the string S.
By traversing the tree it is possible to enumerate all the suffixes of the string S.
Nodes with a single descendant can be removed, the incoming and outgoing arcs are also removed and replaced by a new edge who's label is the concatenation of the two labels.

[2]Practical Algorithm to Retrieve Information Coded in Alphanumeric

Suffix Tree for xabxac

A suffix tree is a data structure to hold all the suffixes of T.

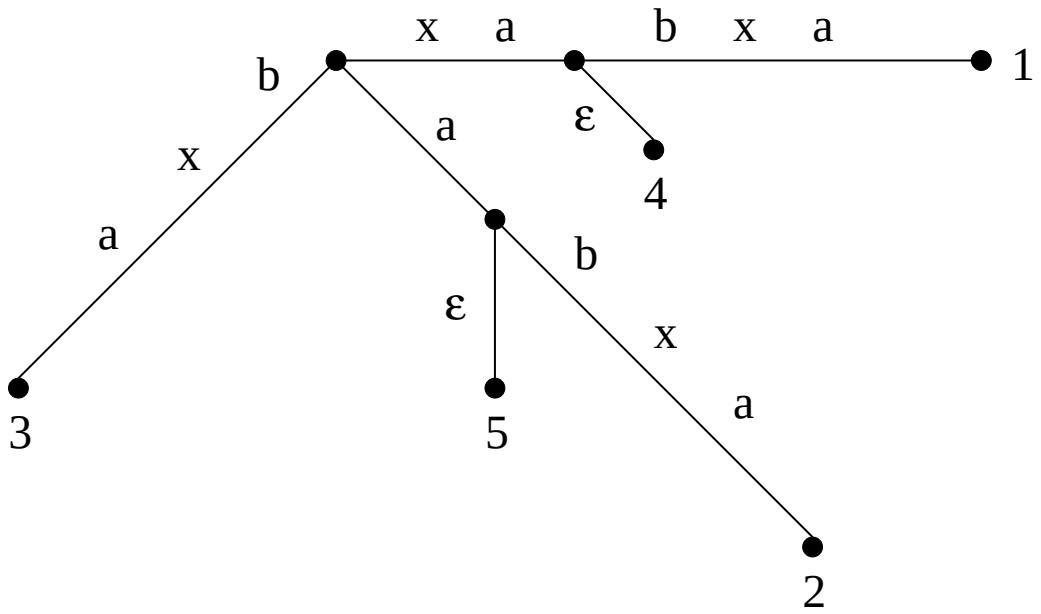
```
  123456
1 xabxac
2  abxac
3   bxac
4    xac
5     ac
6      c
```



Definitions

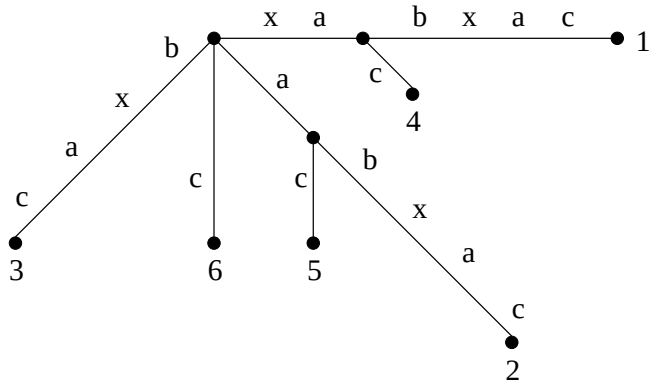
A suffix tree $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m.
Edges are labeled with (non-empty) sub-strings of S.
No two edges out of a node can have edge-labels beginning with the same character.
For any leaf i, the concatenation of the edge-labels on the path from the root to leaf i, exactly spells out the suffix of S that starts at position i, i.e. $S[i..m]$

The Need for a Terminator (xabxa)



In the above tree, *xa* and *a* are two suffixes that are a prefix of another suffix, which means that to insert them in the tree we would have to have empty labels, denoted by ϵ, and this would violate our definition of a suffix tree.

Suffix tree for xabxac



- $\forall$ i the concatenation of all edges from the root to the leaf spells out the suffix that starts at position i, $S[i..m]$, where $m = |S|$.
- if one suffix of S matches also matches a prefix of another suffix of S then no suffix tree can be built, to circumvent the problem a termination character (a symbol which is not part of Σ) is added to the end of S, i.e. $S\$$.

Using ST: Find all occurrences of P in T

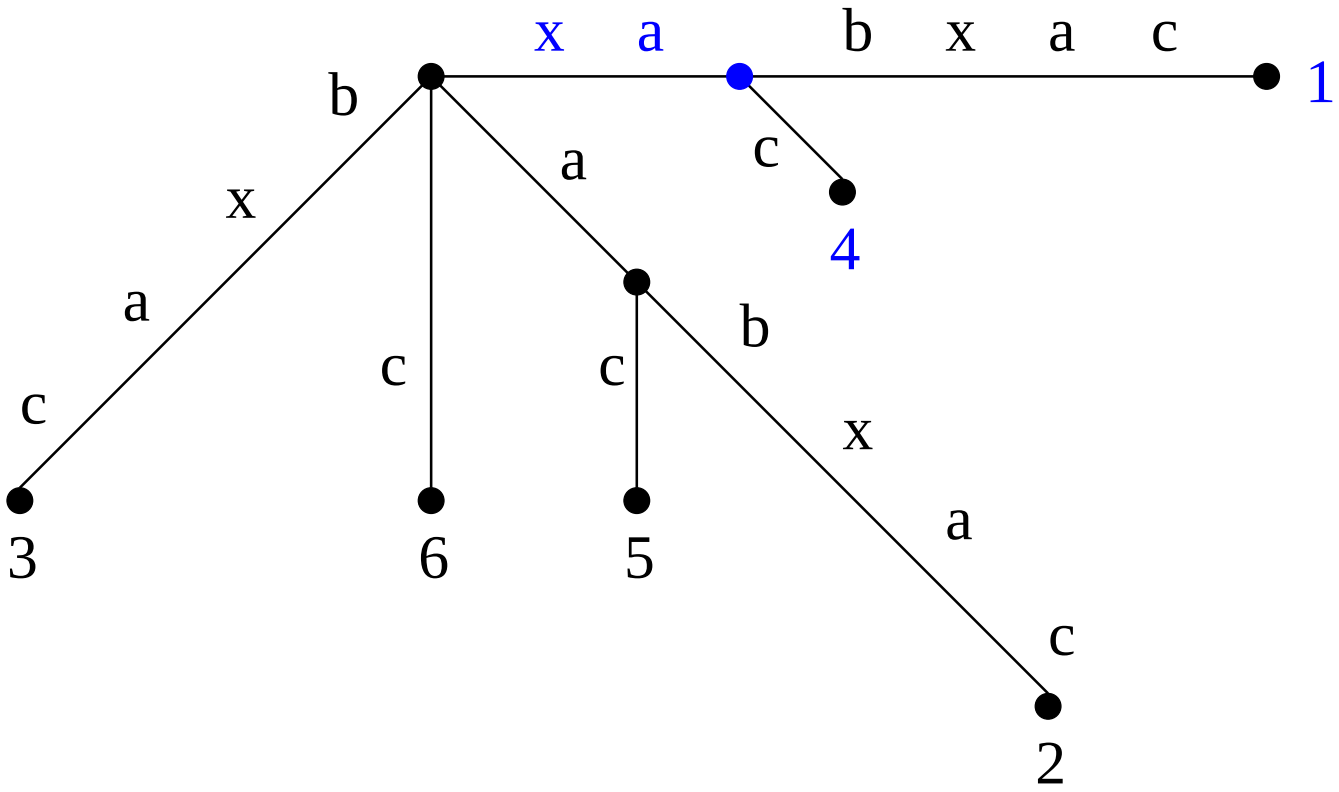
Propose an algorithm for finding all the occurrences of P in T, once a suffix tree of T has been built.
What is the time complexity of your algorithm?

Using ST: Find all occurrences of P in T

```
{ Initialization }
    Let n = |P| and m = |T|
    Build a suffix tree 𝒯 for T in O(m)

{ Search stage }
    i := 1;
    while i ≤ n and match P(i) in 𝒯 do
        i := i + 1;
    od;
    if i ≤ n then
        report failure, P does not appear anywhere in T
    else
        report success, every leaf below the point of the
        last match  is numbered with a starting location
        of P in T.
    fi;
```

Example: finding all xa's in xabxac



1	2	3	4	5	6
x	a	b	x	a	c

Features

- The **path is unique** because there are no two edges out of a node starting with the same letter, thus each branching decision is unique.
- If P occurs in T then it ought to be a prefix of a suffix of T.
- To further report all occurrences requires **traversing the subtree** and will necessitates time proportional to the number of occurrences, k, and is independent of the size of the labels leading to those k leaves.
- The topology of a suffix tree is **unique**, in other words, the suffix trees produced by any two algorithms should identical, except for the order of the children.

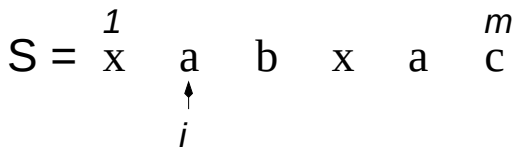
Example: building a suffix tree

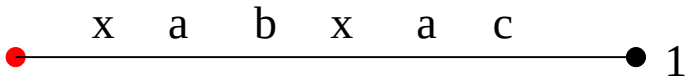
$$S = \overset{1}{x}\ a\ b\ x\ a\ \overset{m}{c}$$

Example: building a suffix tree

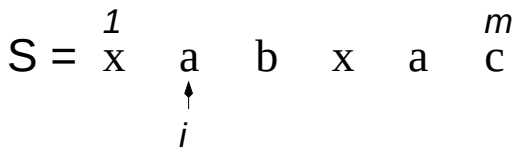
$$S = \overset{1}{x}\ a\ b\ x\ a\ \overset{m}{c}$$

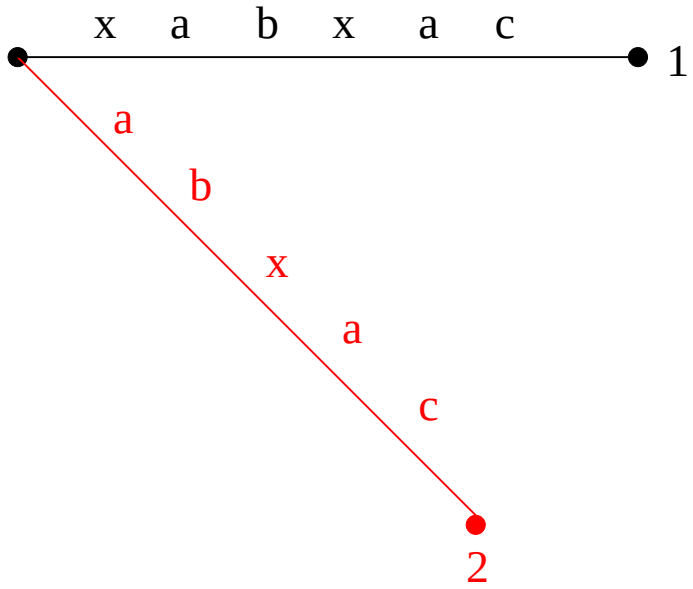
Example: building a suffix tree



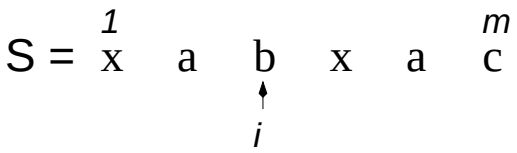


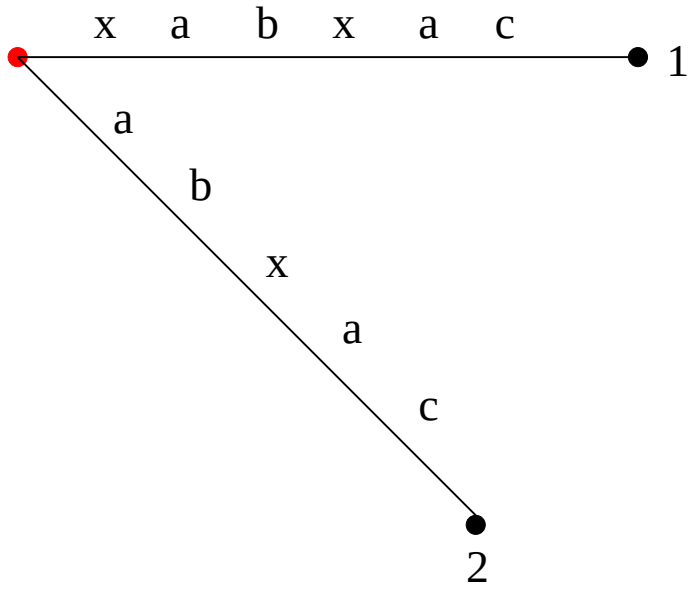
Example: building a suffix tree



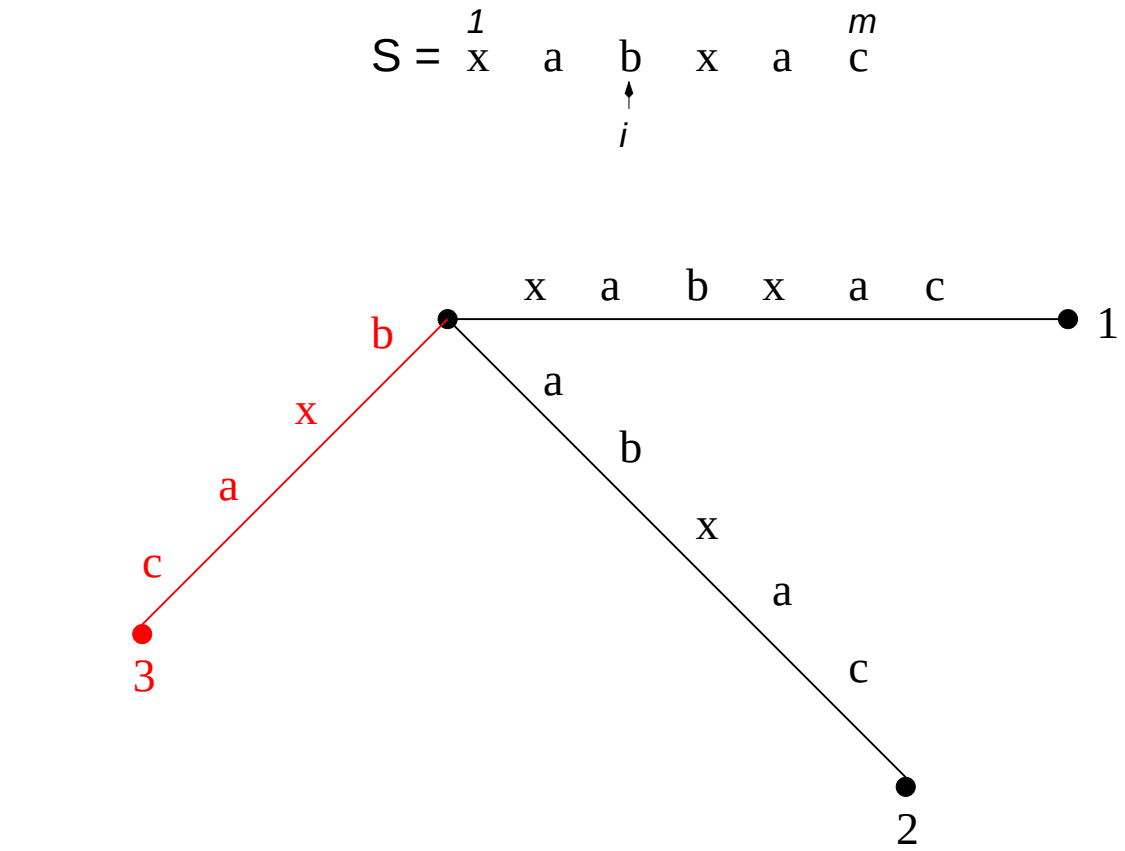


Example: building a suffix tree

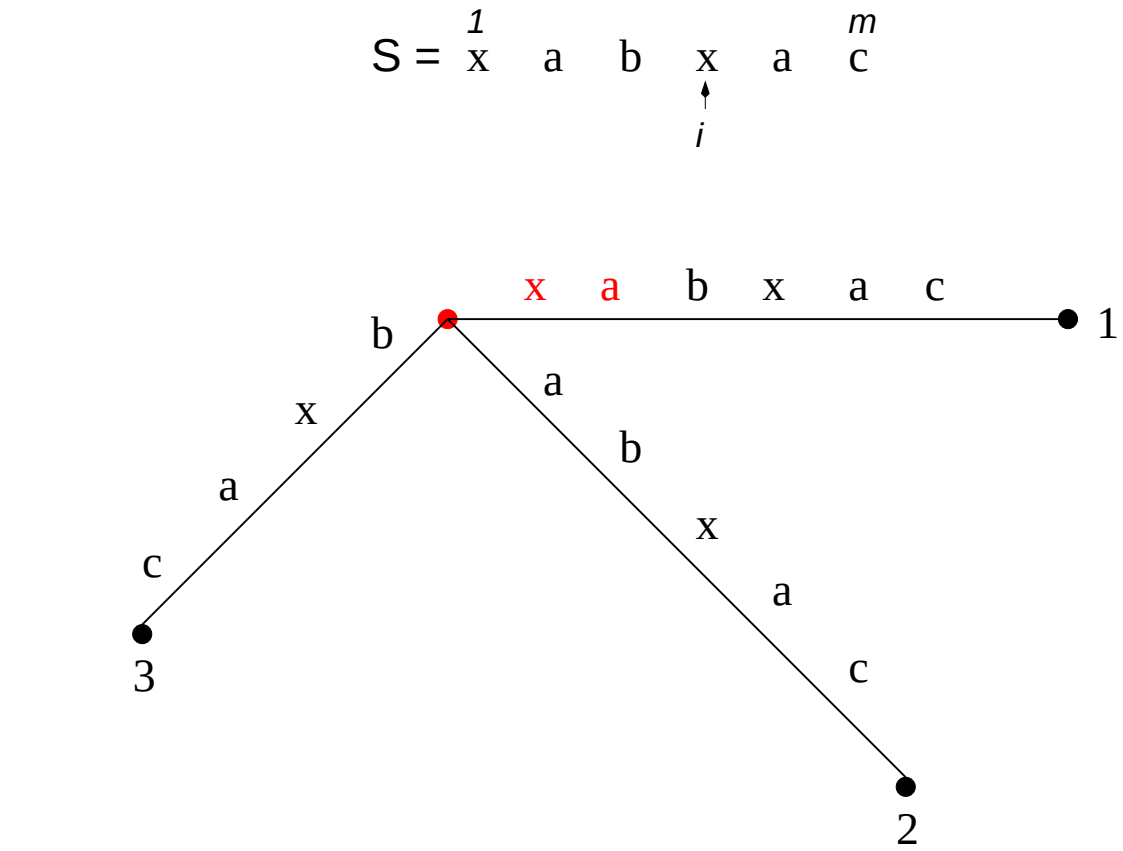




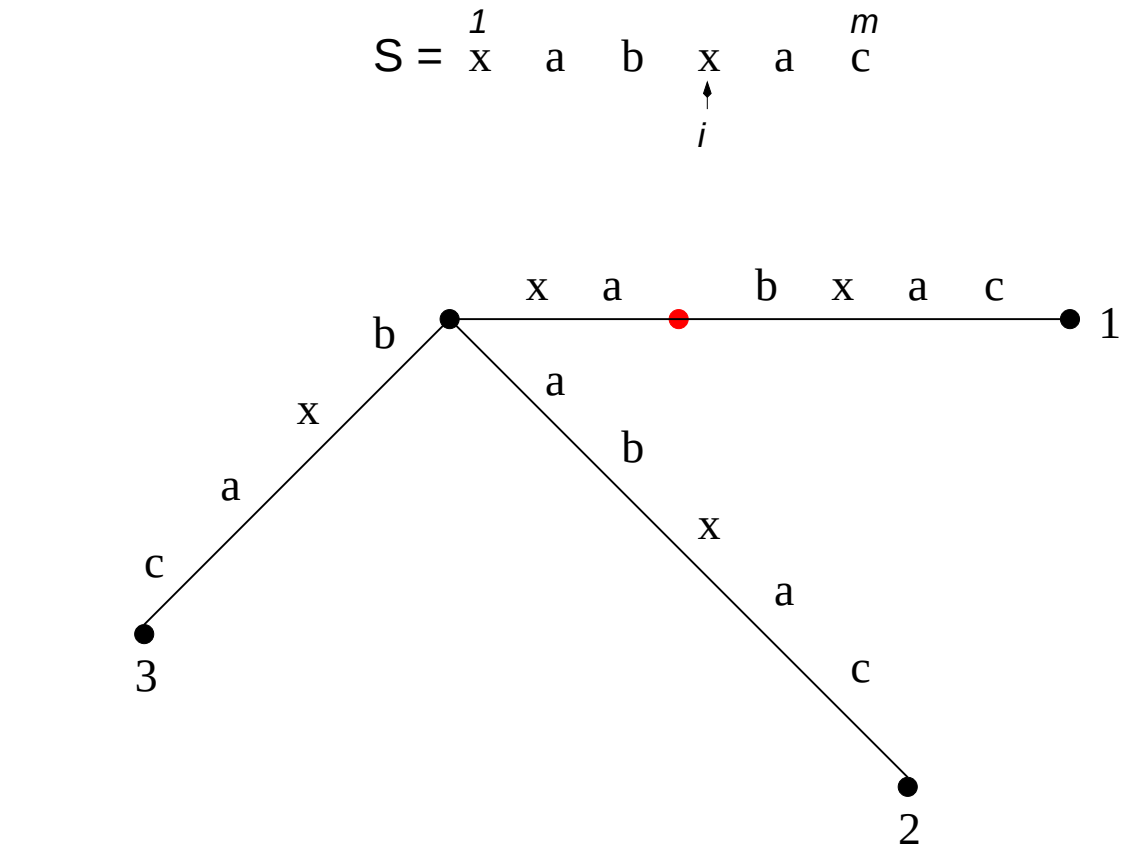
Example: building a suffix tree



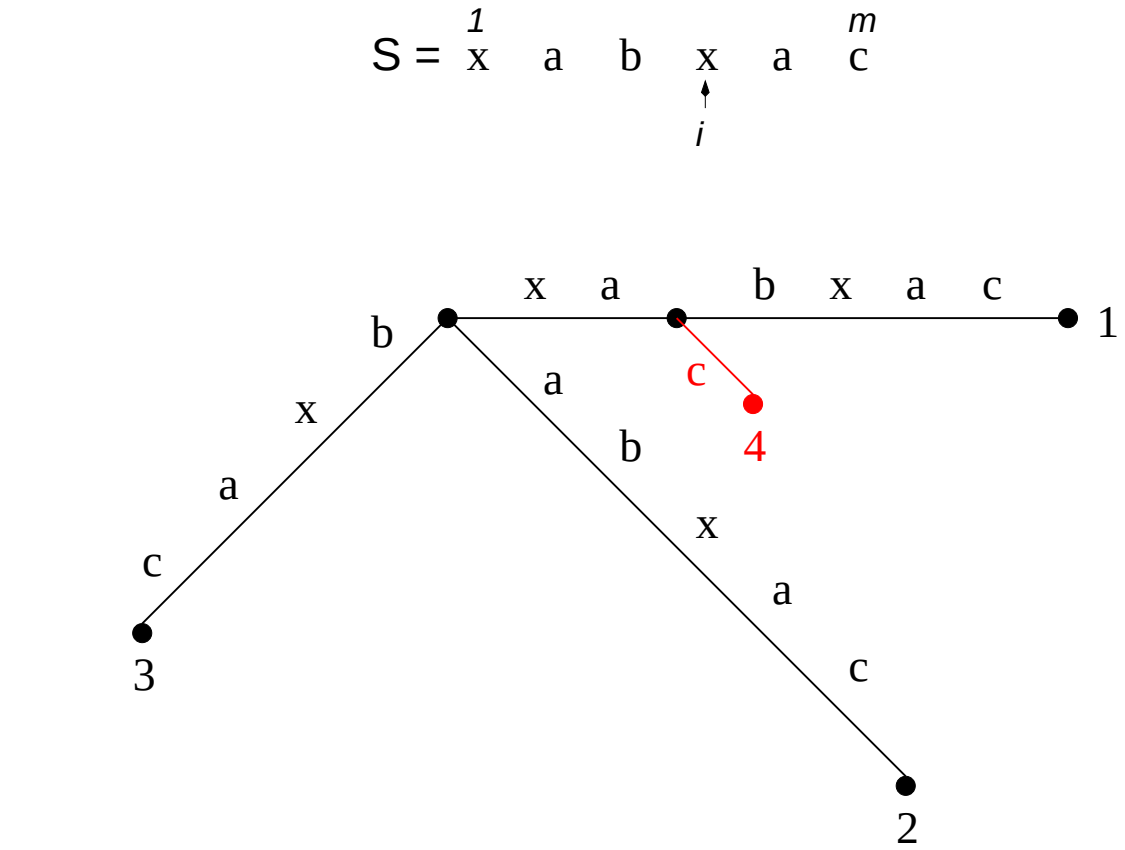
Example: building a suffix tree



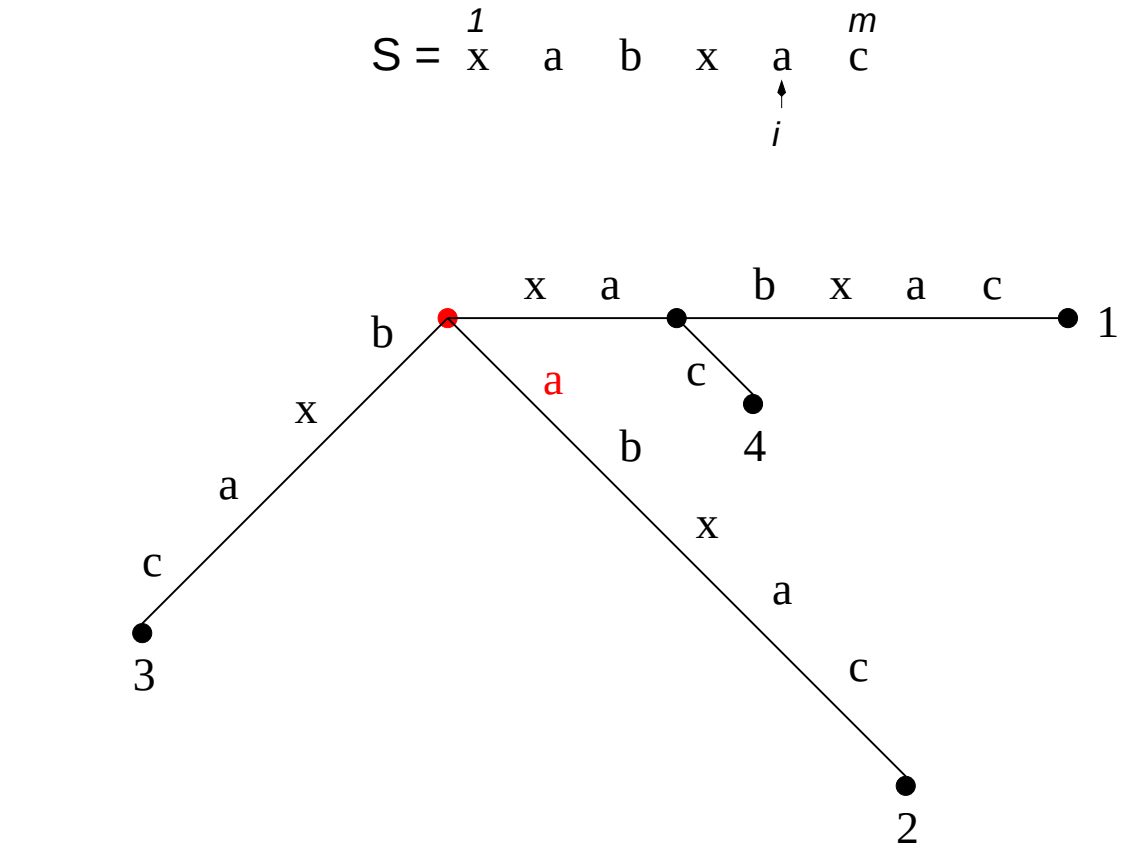
Example: building a suffix tree



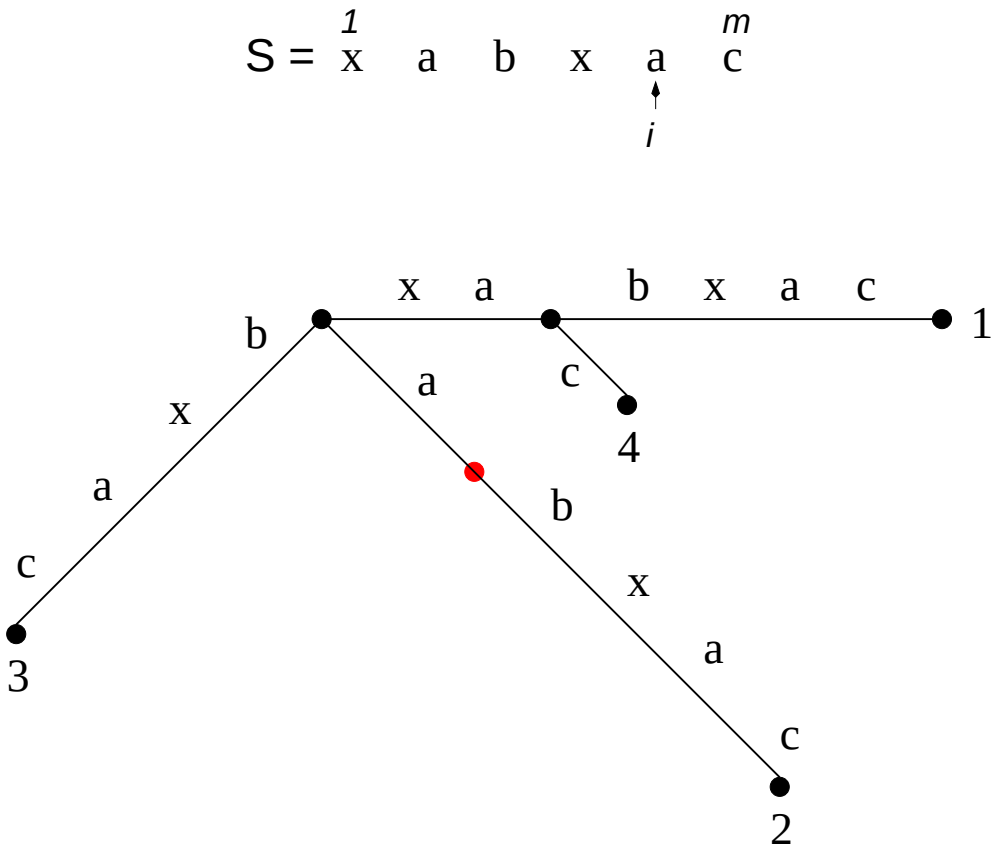
Example: building a suffix tree



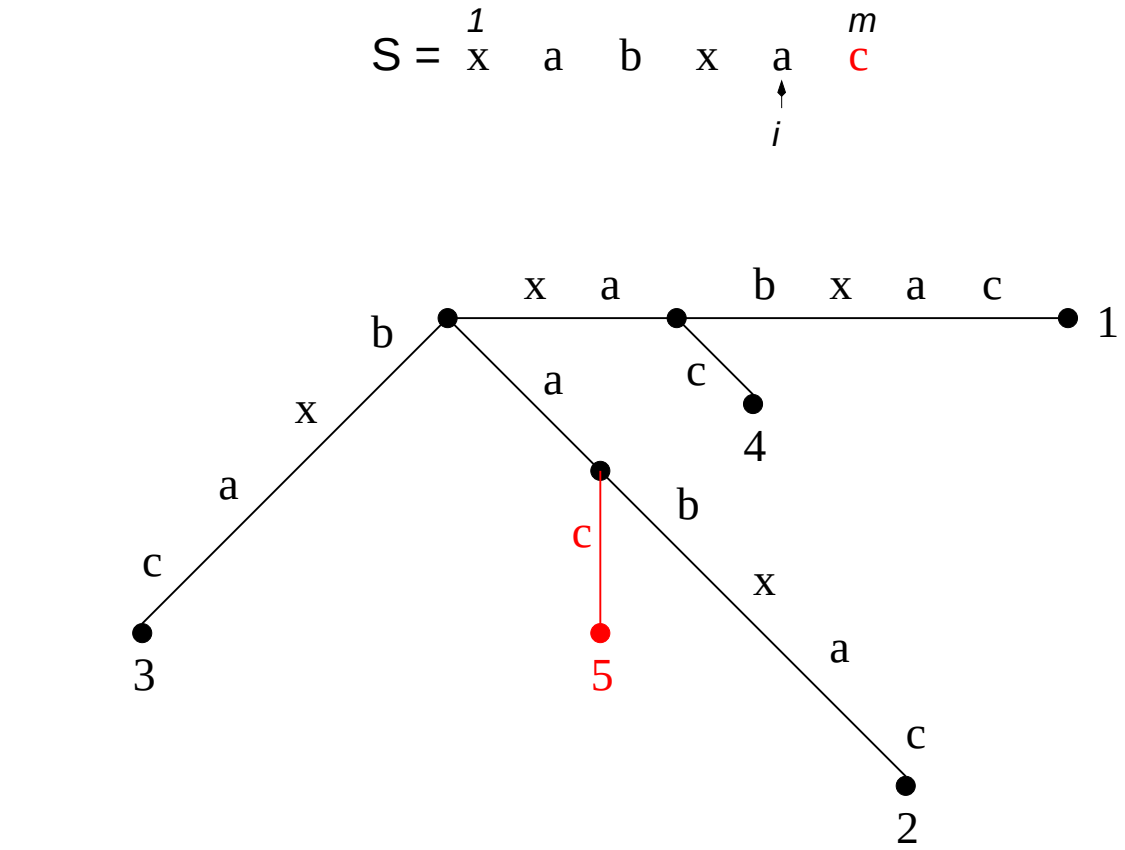
Example: building a suffix tree



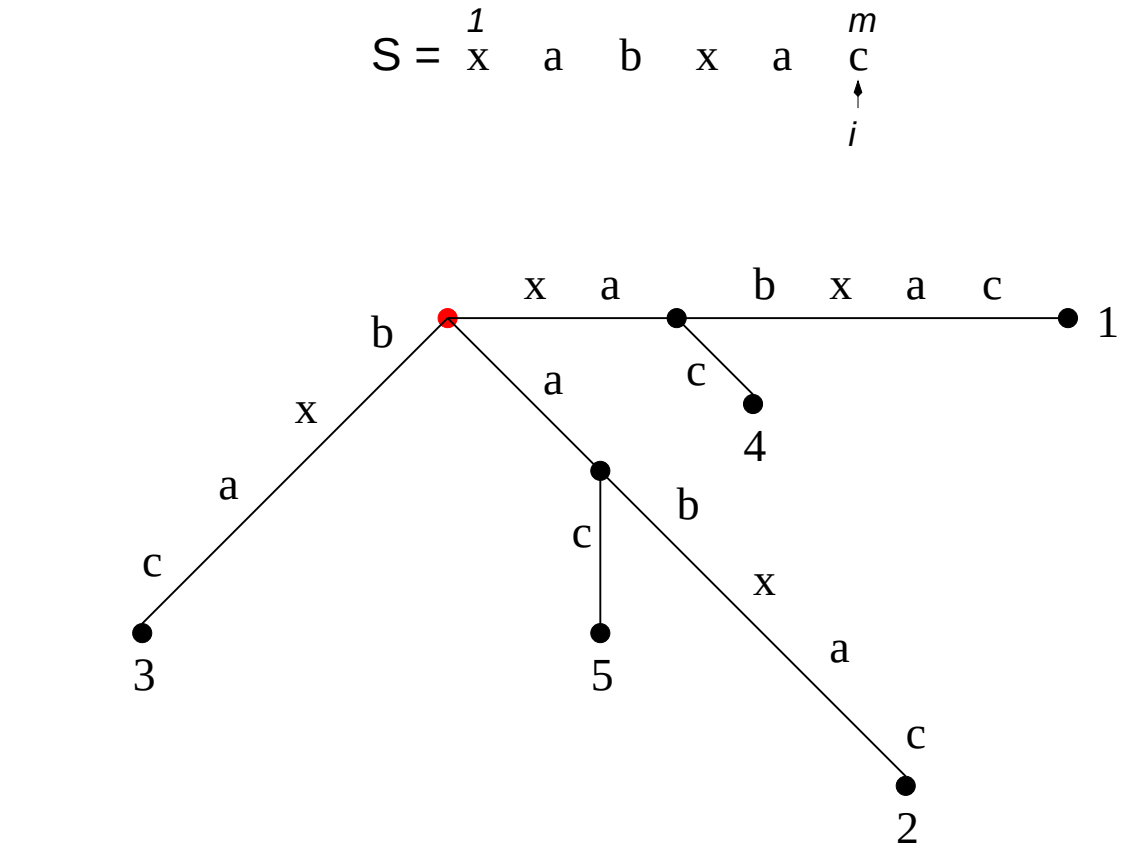
Example: building a suffix tree



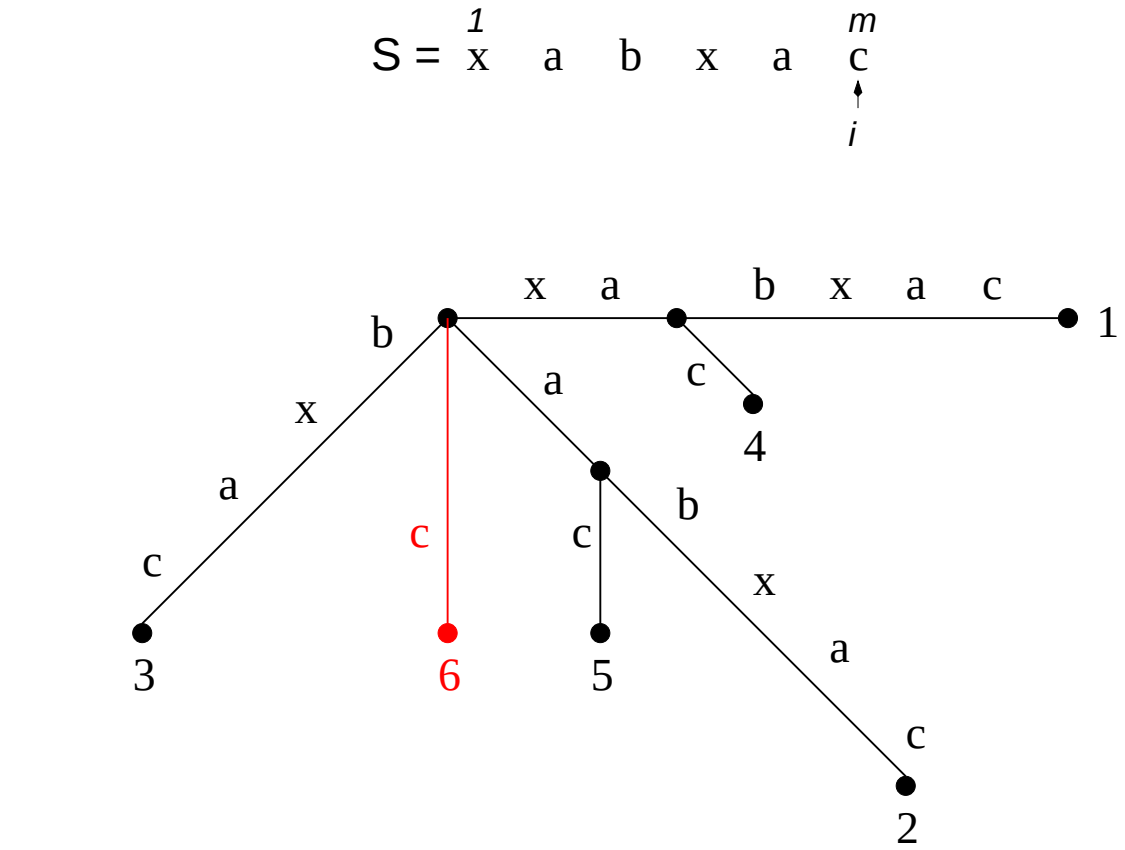
Example: building a suffix tree



Example: building a suffix tree



Example: building a suffix tree



Naïve algorithm to build a suffix tree in $\mathcal{O}(m^2)$

{ Initialization }
    Create a new tree, enter the single edge $S[1..m]\$$

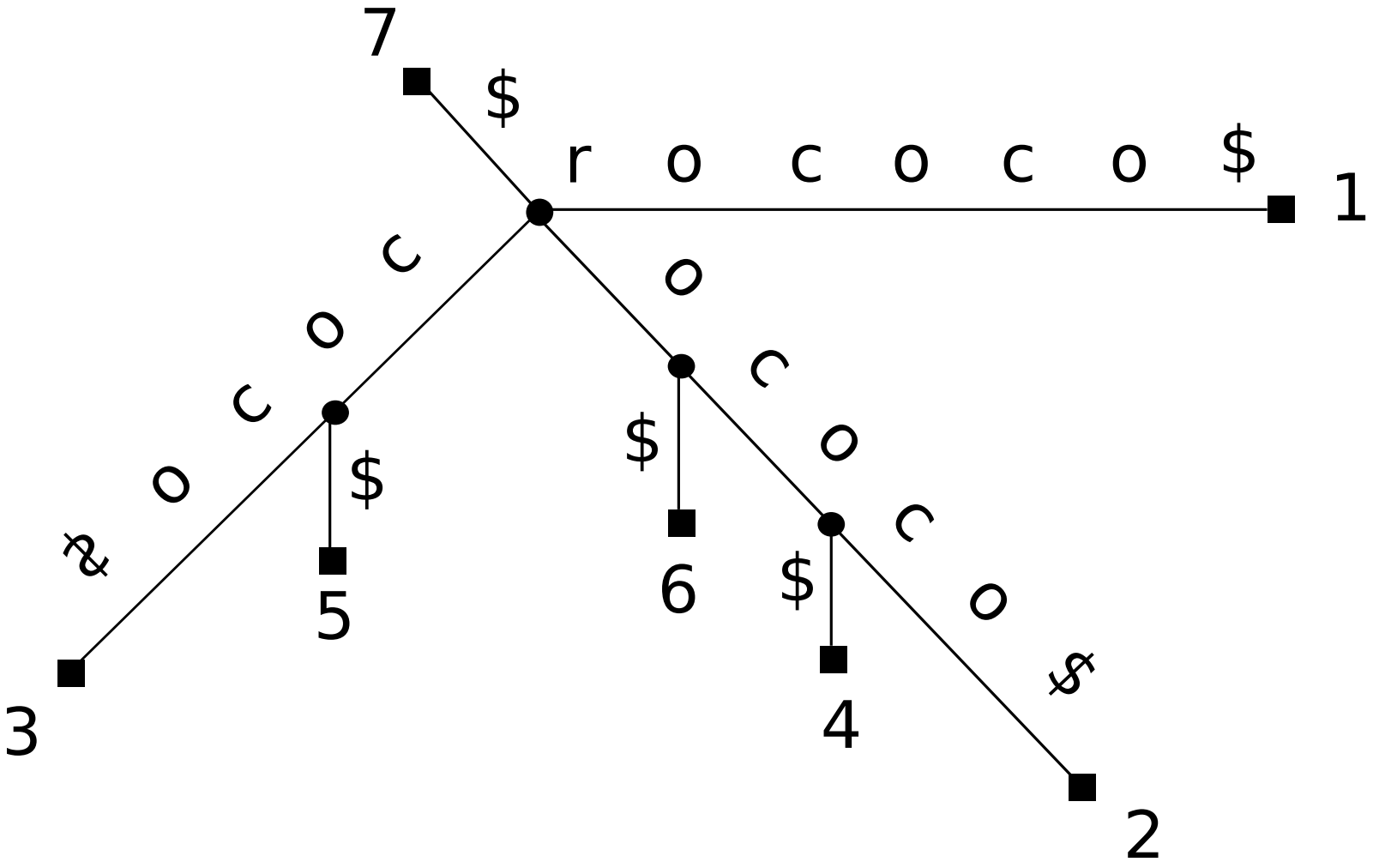
Naïve algorithm to build a suffix tree in $\mathcal{O}(m^2)$ (cont.)

{ Successively add $S[i..m]\$$ to the growing tree $\mathcal{T}$ }

for i from 2 to m do
find the longest match for $S[i..m]$ in $\mathcal{T}$
Let's call $S(j)$ the position of the mismatch

if $S(j)$ was found at a node, say w, then
add a new child to w labeled $S[j..m]\$$
else $S(j)$ is in the middle of an edge,
say (u, v), then insert a new node w:
replace (u, v) by (u, w) and (w, v),
where (u, w) correspond to the portion
of (u, v) that matched $S[i..j]$ and
(w, v) the remaining part. Finally,
insert a new edge (w, i) labelled $S[j..m]$.

Example: rococo



Exercises

Build by hand a suffix tree for some of these words: molecule, allele, rococo, tarantara, tartar, repetitive, murmurs, mathematic, banana and monotonous.

Size of the tree: memory usage

The total number of nodes is $\mathcal{O}(|S|)$. Consider the naïve algorithm. The $n = |S|$ suffixes are added one by one. When a suffix is added, it forces the creation of at most one internal node (sometimes, the algorithm only necessitates adding a branch out of an existing node). Therefore, the maximum number of internal is $\mathcal{O}(|S|)$, and the number of leaves is $\mathcal{O}(|S|)$. Hence, the total number of nodes is $\mathcal{O}(|S|)$.

The total number of nodes is $\mathcal{O}(|S|)$, does this mean that the space requirement will be $\mathcal{O}(|S|)$ also?

The presentation suggests that the labels on the arcs of the tree are strings themselves.

Since there are $\mathcal{O}(|S|)$ edges, each of them labeled with a string $\mathcal{O}(|S|)$ long, implies $\mathcal{O}(|S|^2)$ space!

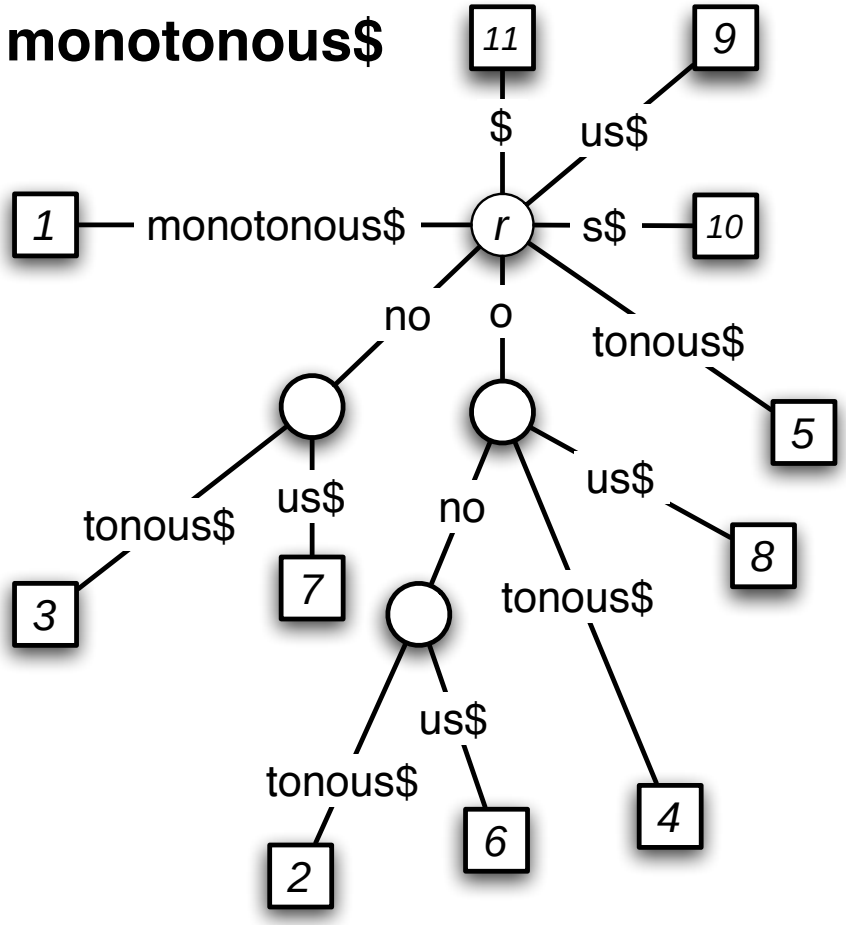
[Hackers, can you do better?]

Edge label compression

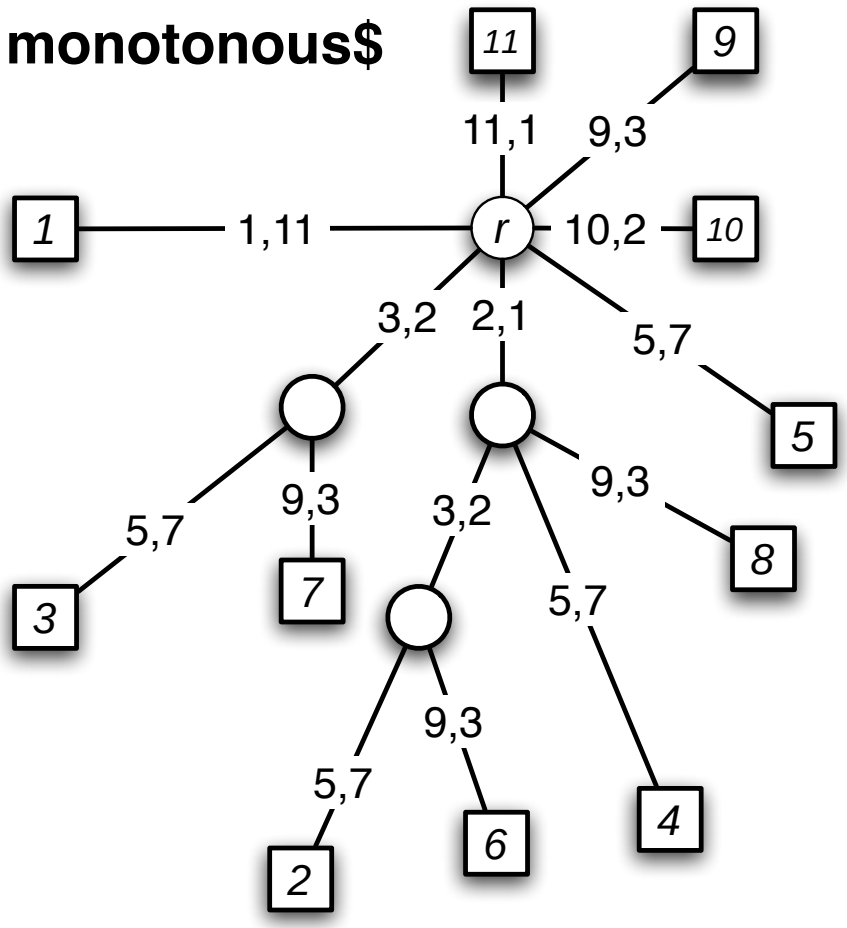
We know that the total number of nodes for the suffix tree of a terminated string is $|S| - 1$, and therefore the number of edges is $|S| - 2$, representing each edge with two numbers, **start and ending position of the label within the original string**, allows us to use a constant amount of space per label, and therefore the space requirement is linear.

In practice, this can cause a lot of paging, if the string and tree cannot be fitted together in main memory.

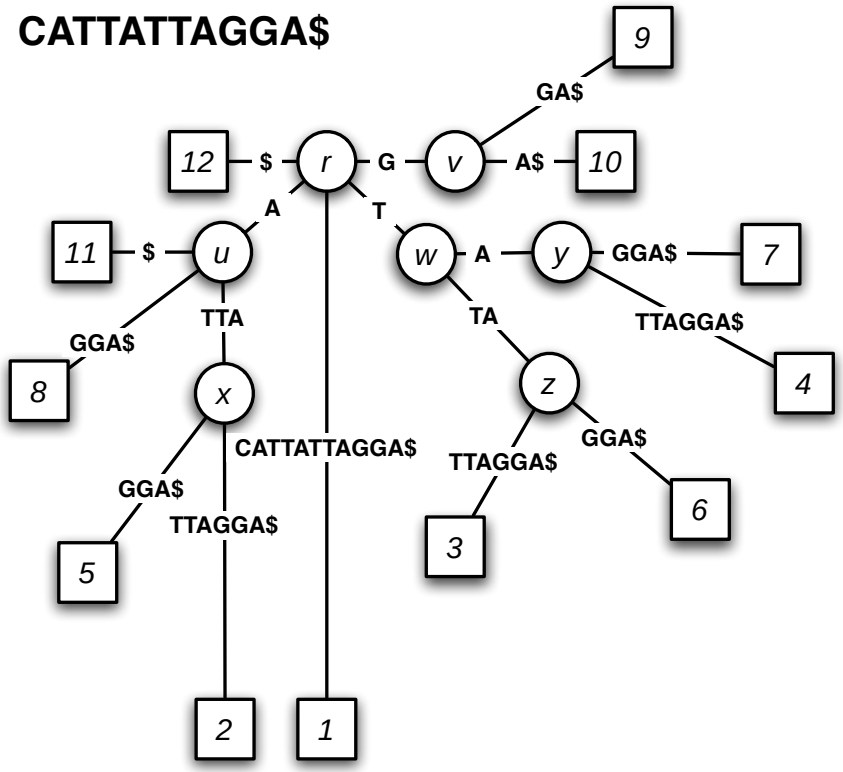
Edge label compression (cont.)



Edge label compression (cont.)

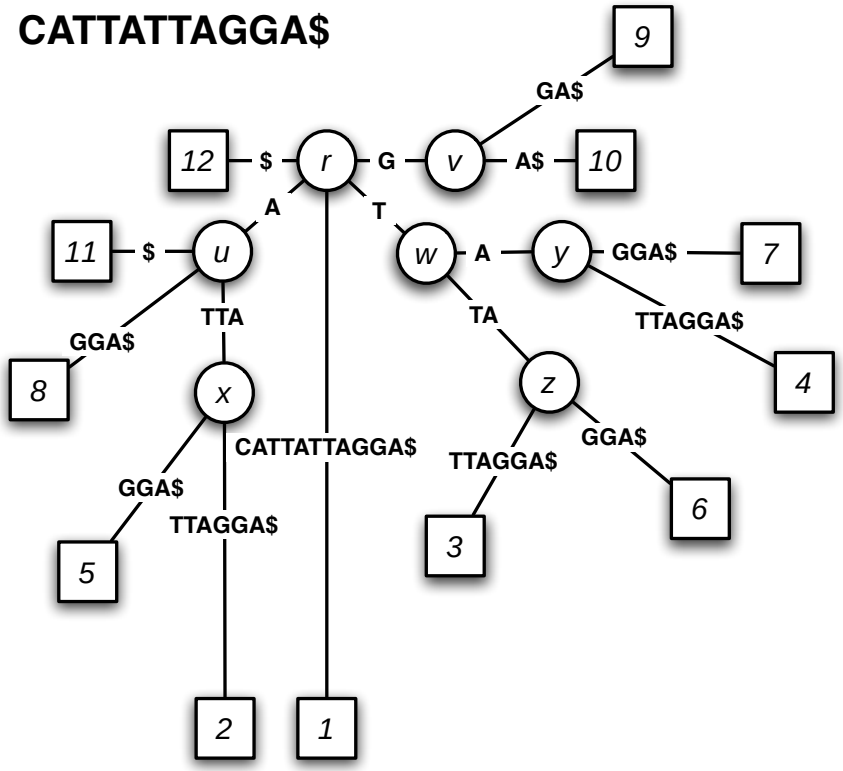


Edge label compression (cont.)



The **path label** of a node is the concatenation of all the **edge labels** on the unique path from the root to that node, e.g. `path-label(z) = TTA`.

Edge label compression (cont.)



The **string depth** of a node is the length of its path label, e.g. `string-depth(z) = length(path-label(z)) = 3`.

Linear Time Construction

Several algorithms exist for the linear time construction of suffix trees. Ukkonen's algorithm is often considered the method of choice.

Ukkonen (1995); this linear algorithm also allows for *online* left-to-right processing and is conceptually easier to understand than the previous two methods.

The presentation of the linear time algorithm is beyond the scope of the course. The library presented in the next few slides has an implementation of Ukkonen's algorithm.

Note: In the lecture notes, the convention used in most textbooks and publications denoting the first index of a string by 1 is used, but for the the Java implementation, the first index is 0.

Suffix Tree Library

On the course web site, you will find a Java library that implements a suffix tree data structure.
It was developed by **Daniela Cernea** in 2003 for her Honours project.
The next slides present an overview of the classes involved.
Creating a suffix tree:

```
SuffixTree tree = new SuffixTree( "acgt" );
```

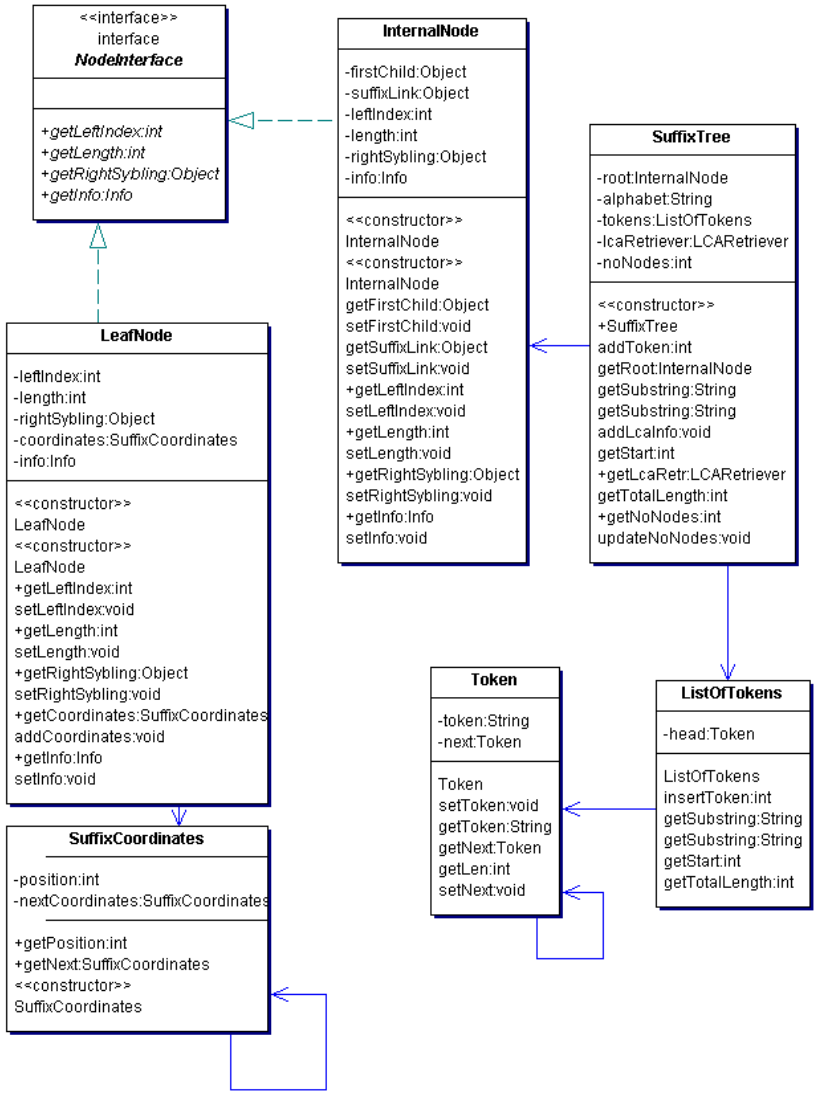
where "acgt" is the alphabet.
For adding strings to the tree, we will need a builder:

```
TreeBuilder builder = new TreeBuilder( tree );
```

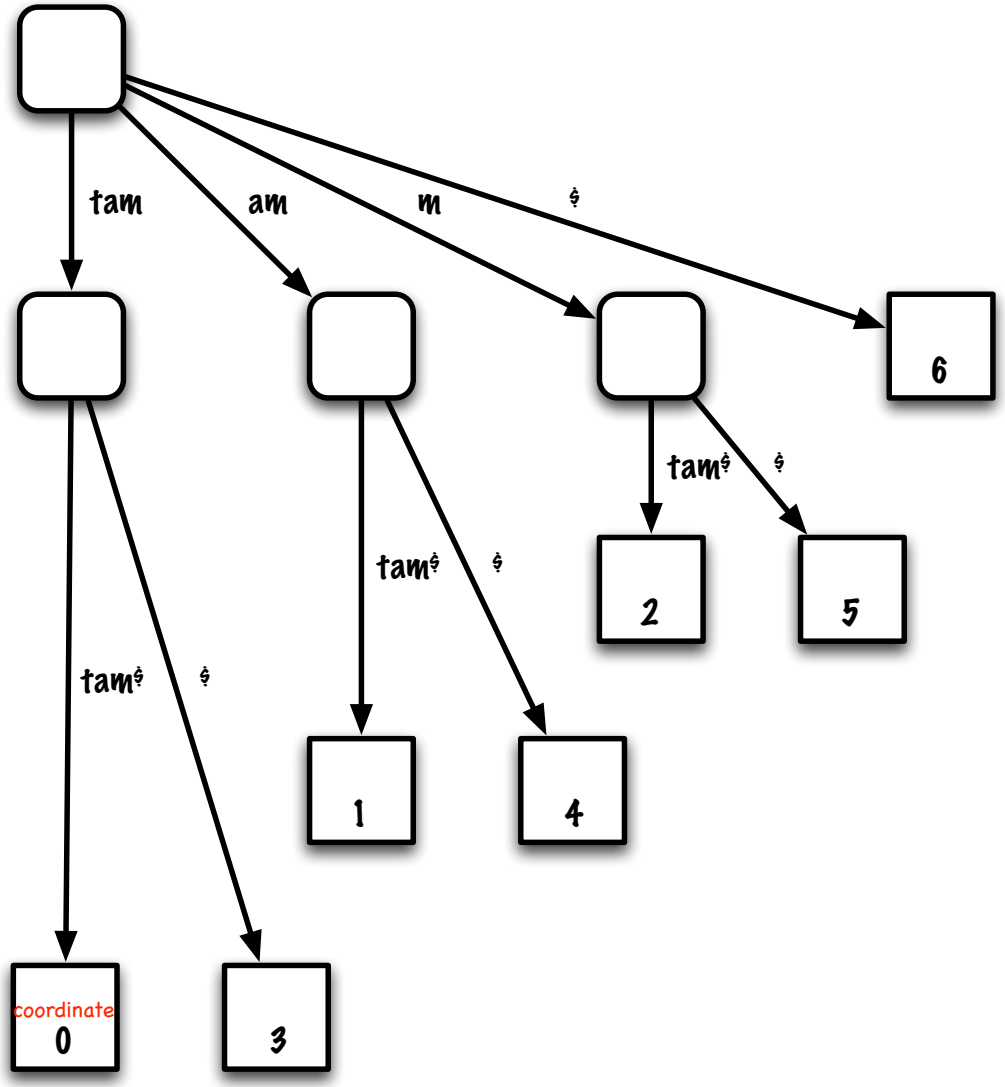
The method `addToken` is used to insert a string into an existing tree:

```
builder.addToken( "cattattagga" );
```

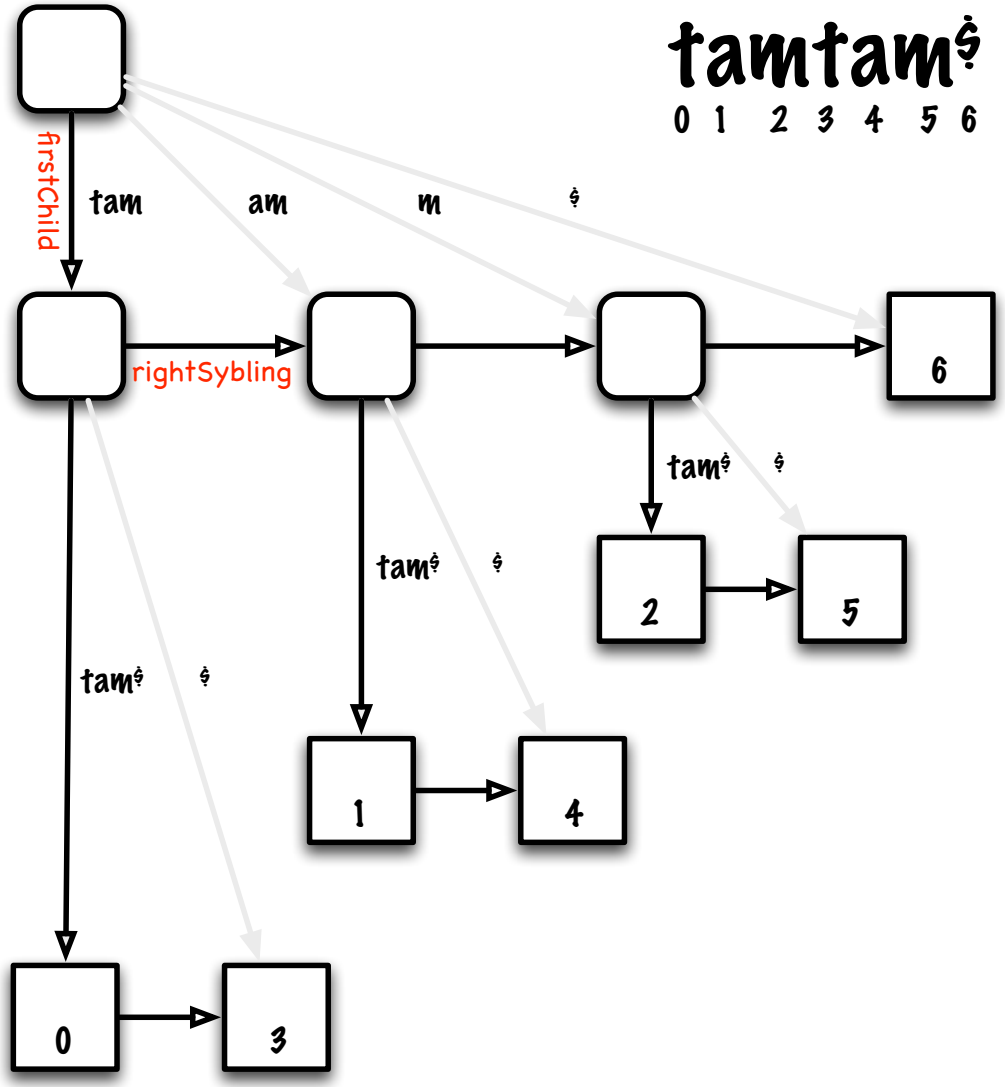
Suffix Tree Library (cont.)



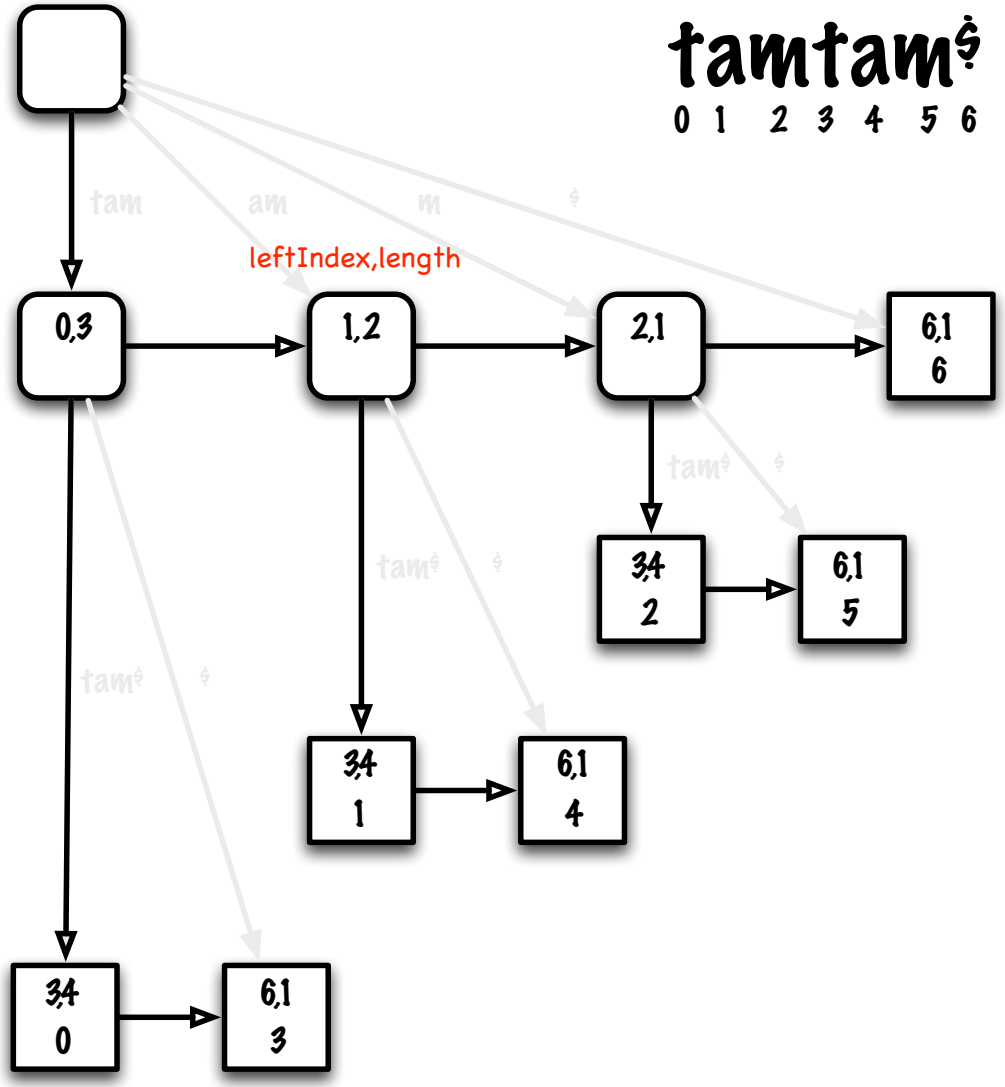
Suffix Tree Library (cont.)



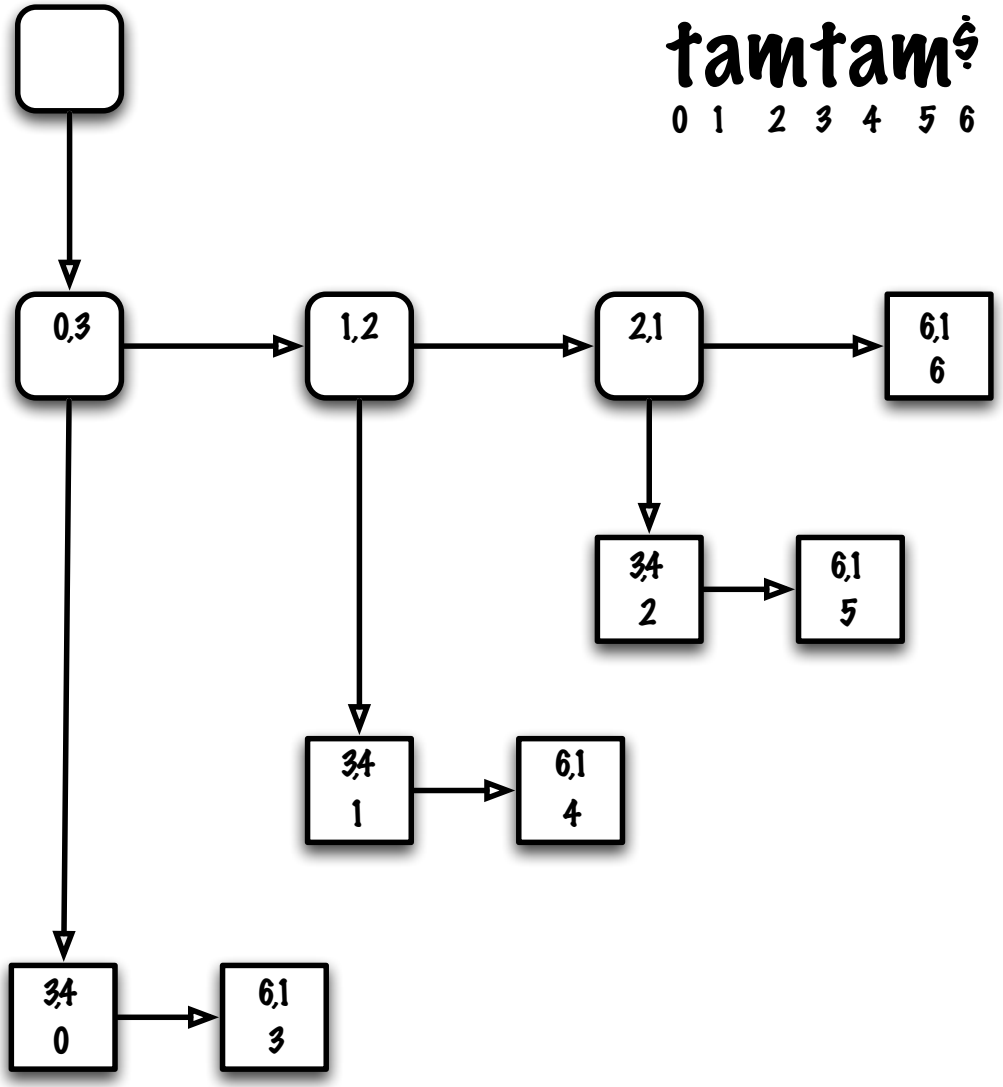
Suffix Tree Library (cont.)



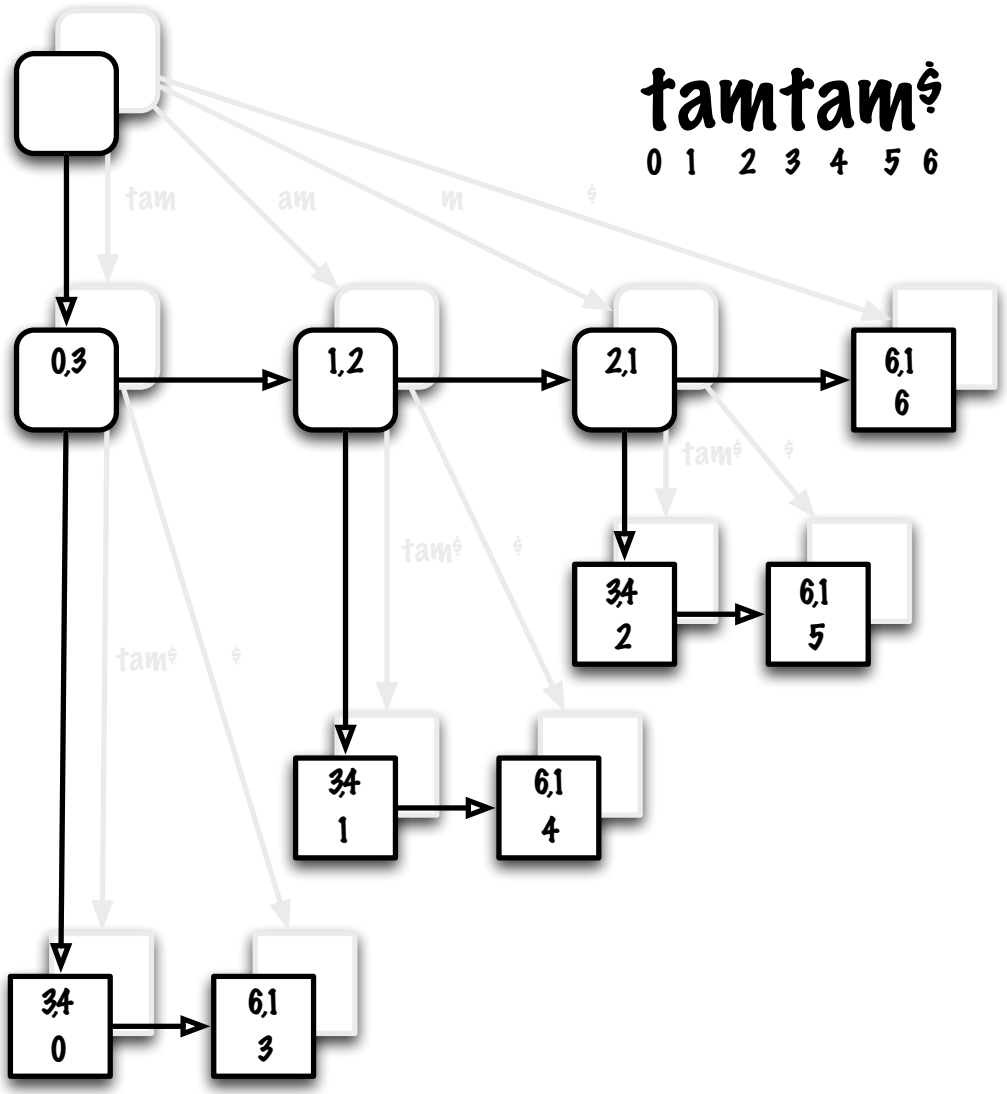
Suffix Tree Library (cont.)



Suffix Tree Library (cont.)



Suffix Tree Library (cont.)



Suffix Tree Library

Most suffix tree algorithms involve traversing the tree.
The simplest, and most informative, algorithm traversing a tree simply prints its content.
Here is how it is used:

```
SuffixTree tree = new SuffixTree( "impst" );

TreeBuilder builder = new TreeBuilder( tree );

builder.addToken( "mississippi$" );

PrintTree printer = new PrintTree( tree );

printer.prettyPrint();
```

Suffix Tree Library (cont.)

```
<node root>
  <node label=s>
    <node label=si>
      <leaf label=ssippi$ pos=2/>
      <leaf label=ppi$ pos=5/>
    </node>
    <node label=i>
      <leaf label=ssippi$ pos=3/>
      <leaf label=ppi$ pos=6/>
    </node>
  </node>
  <node label=p>
    <leaf label=pi$ pos=8/>
    <leaf label=i$ pos=9/>
  </node>
  <leaf label=mississippi$ pos=0/>
  <node label=i>
    <node label=ssi>
```

Suffix Tree Library (cont.)

```
        <leaf label=ssippi$ pos=1/>
        <leaf label=ppi$ pos=4/>
      </node>
      <leaf label=ppi$ pos=7/>
      <leaf label=$ pos=10/>
    </node>
    <leaf label=$ pos=11/>
</node>
```

Suffix Tree Library (cont.)

```
public class TreePrinter {

    private SuffixTree tree;

    public TreePrinter( SuffixTree tree ) {
        this.tree = tree;
    }

    public void prettyPrint() {
        prettyPrint( (NodeInterface) tree.getRoot(), "" );
    }
```

Suffix Tree Library (cont.)

```
    private void prettyPrint( NodeInterface node, String depth ) {
        String info;

        if ( node == tree.getRoot() ) {
            info = "root";
        } else {
            info=tree.getSubstring( node.getLeftIndex(),node.getLength() );
        }

        if ( node instanceof InternalNode ) {

            System.out.println( depth + "<node " + info + ">" );
            NodeInterface child = (NodeInterface) ( (InternalNode) node ).getFirstChild();
            prettyPrint( child, depth + "  " );
            System.out.println( depth + "</node>" );

        } else {

            String coord = ( (LeafNode) node ).getCoordinates();
            System.out.println( depth+"<leaf "+info+" pos="+coord+"/>" );

        }
        NodeInterface right = (NodeInterface) node.getRightSybling();
       if ( right != null ) {
            prettyPrint( right, depth );
        }
    }
}
```

Suffix tree for tamtam$

```
<node root>
  <node label=tam>
    <leaf label=tam$ pos=0>
    <leaf label=$ pos=3>
  </node>
  <node label=m>
    <leaf label=tam$ pos=2>
    <leaf label=$ pos=5>
  </node>
  <node label=am>
    <leaf label=tam$ pos=1>
    <leaf label=$ pos=4>
  </node>
  <leaf label=$ pos=6>
</node>
```

$\Rightarrow$ Printed by **TreePrinter**.

Suffix arrays

Nowadays, suffix arrays are used rather than suffix trees.
Given an input sequence S of length $|S| = n$.
Each suffix is represented by its starting position (an integer), a suffix array lists all the suffixes in lexicographic order.
Uses $\mathcal{O}(n)$ space; with small constant!
$\log_2 n$ bits are needed to represent a position.
32 bits (4 bytes) are enough to represent sequences up to 4 Gbytes long.
$4 \times n$ bytes are needed to represent a sequence of size n for $n < 4$ Gbytes.
For the same size of input, the most compact suffix tree requires $28 \times n$ bytes.

Suffix arrays

Manber U et Myers G (1990) *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*: 319 – 327.
Manber U and Myers G (1993) *SIAM J on Computing* **22**(5):935–948.
Until very recently directly constructing a suffix array was costly, $\mathcal{O}(n \log n)$**.**
Building in $\mathcal{O}(n)$ **time.**
Kärkkäinen J et Sanders P (2003) In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03), LNCS 2719*, 943-955. (Skew algorithm)

Suffix arrays

Bottom up traversal,
Abouelhoda M et al. (2003) WABI 2002, *LNCS 2452* :449-463.
Top down traversal,
Abouelhoda M et al. (2002) SPIRE 2002, *LNCS 2476* :31-43.
More and more, suffix trees are become a didactic or conceptual tool.
Mohamed Ibrahim Abouelhoda, Stefan Kurtz and Enno Ohlebusch (2004) Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms* **2**(1):53–86.

Software using suffix trees/arrays

REPuter: bibiserv.techfak.uni-bielefeld.de/reputer
S. Kurtz, J. V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, R. Giegerich: REPuter: The Manifold Applications of Repeat Analysis on a Genomic Scale. Nucleic Acids Res., 29(22):4633-4642, 2001.

VMATCH: www.vmatch.de

MUMMER: mummer.sourceforge.net
A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg (1999) Alignment of Whole Genomes. Nucleic Acids Research, 27:11 (1999), 2369-2376.

Seed and its suffix array library

One of the software systems developed by research group contains a suffix array library written in C:

Seed: bio.site.uottawa.ca/software/seed
Mohammad Anwar, Truong Nguyen and Marcel Turcotte (2006) Identification of consensus RNA secondary structures using suffix arrays. BMC Bioinformatics, 7:244.

References

- Aluru S. (2006) Handbook of Computational Molecular Biology. Chapman & Hall/CRC, §5, 6 and 7. (QH 324.2 .H357 2006)
- Gusfield, D. (1997) Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge Press, §1, 2, 5 and 6. (MRT General QA 76.9 .A43 G87 1997)
- Jones N.C. and Pevzner P.A. (2004) An Introduction to Bioinformatics Algorithms, MIT Press, pp 320–332. (QH324.2 b.J66 2004)

Please don't print these lecture notes unless you really need to!