

String Algorithms and Data Structures

Suffix Tries (and Trees)

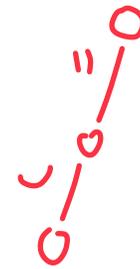
CS 199-225
Brad Solomon

March 2, 2026

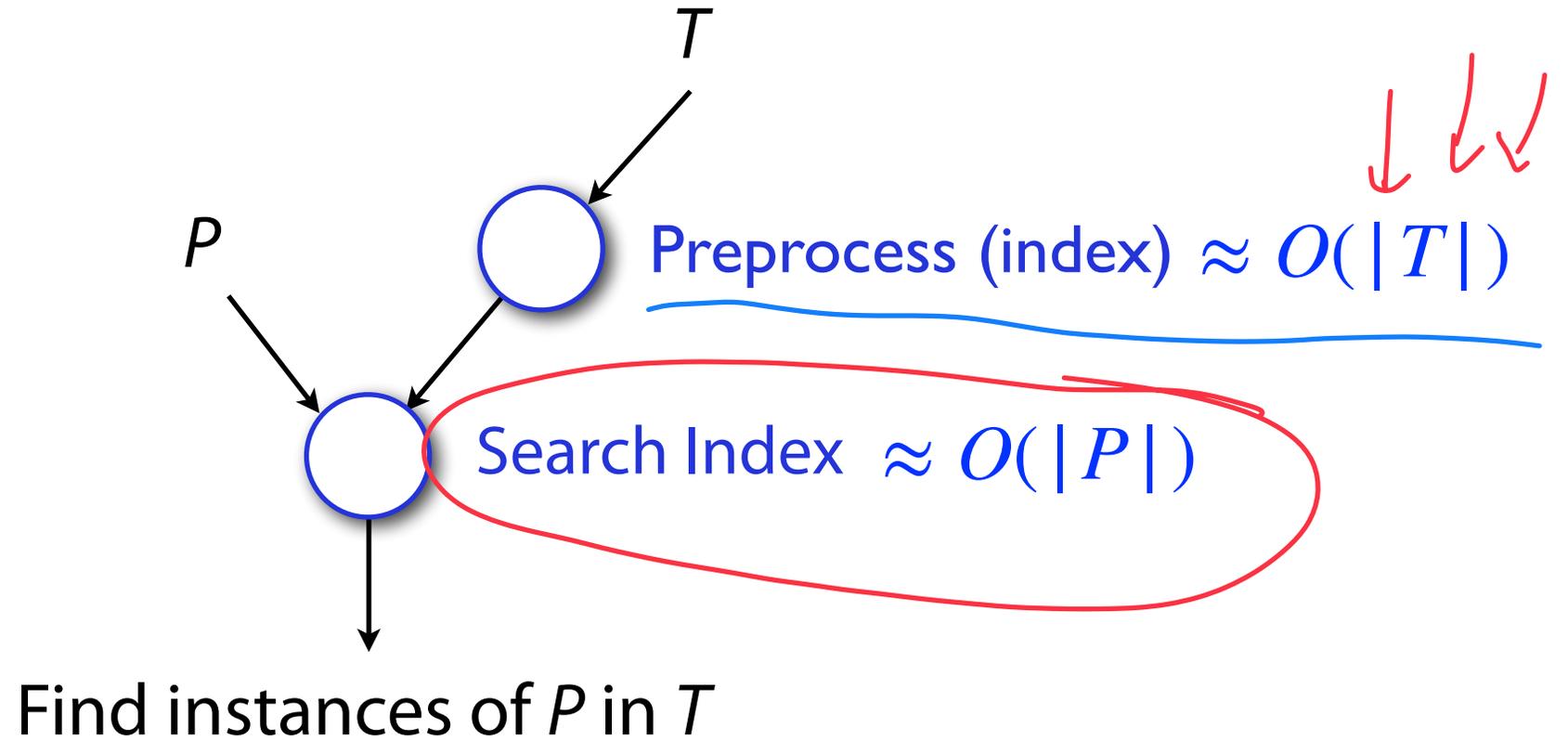


UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science



Exact pattern matching *w/ indexing*



String indexing with Tries

Trie: A rooted tree storing a collection of (key, value) pairs

Keys: Values:

instant 1

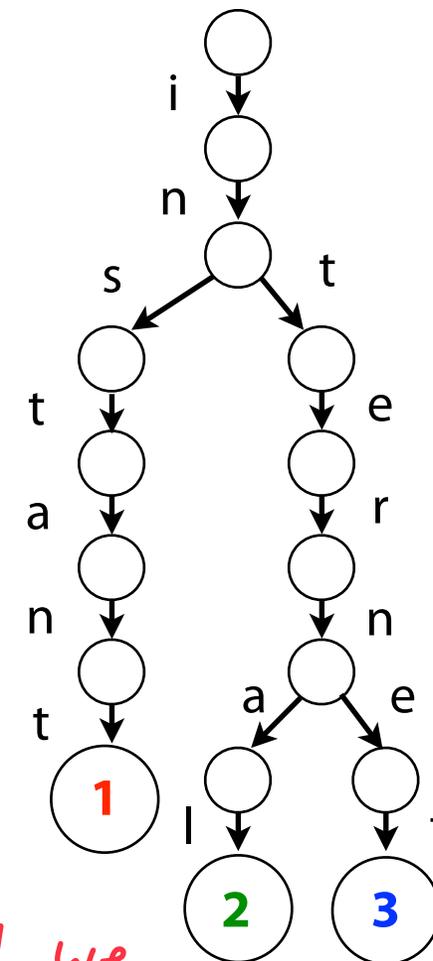
internal 2

internet 3

Each edge is labeled with a character $c \in \Sigma$

For given node, at most one child edge has label c , for any $c \in \Sigma$

Each key is "spelled out" along some path starting at root and each value is stored at the leaf



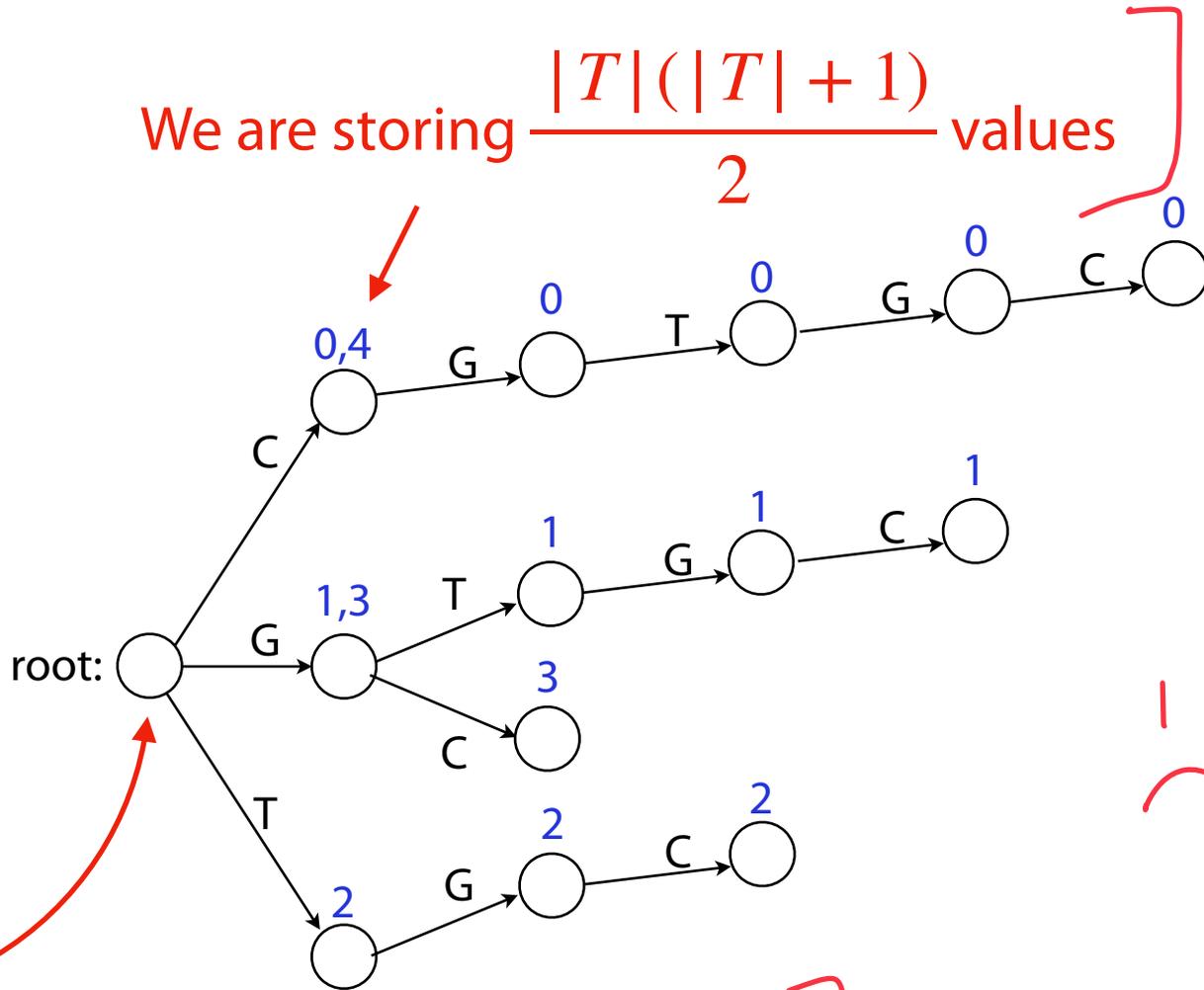
A - array

why we did this

NaryTree build_trie(std::string T)

T: C G T G C

| Key | Value |
|-----|-------|
| C | 0 |
| G | 1 |
| T | 2 |
| G | 3 |
| C | 4 |
| CG | 0 |
| GT | 1 |
| TG | 2 |
| ... | ... |

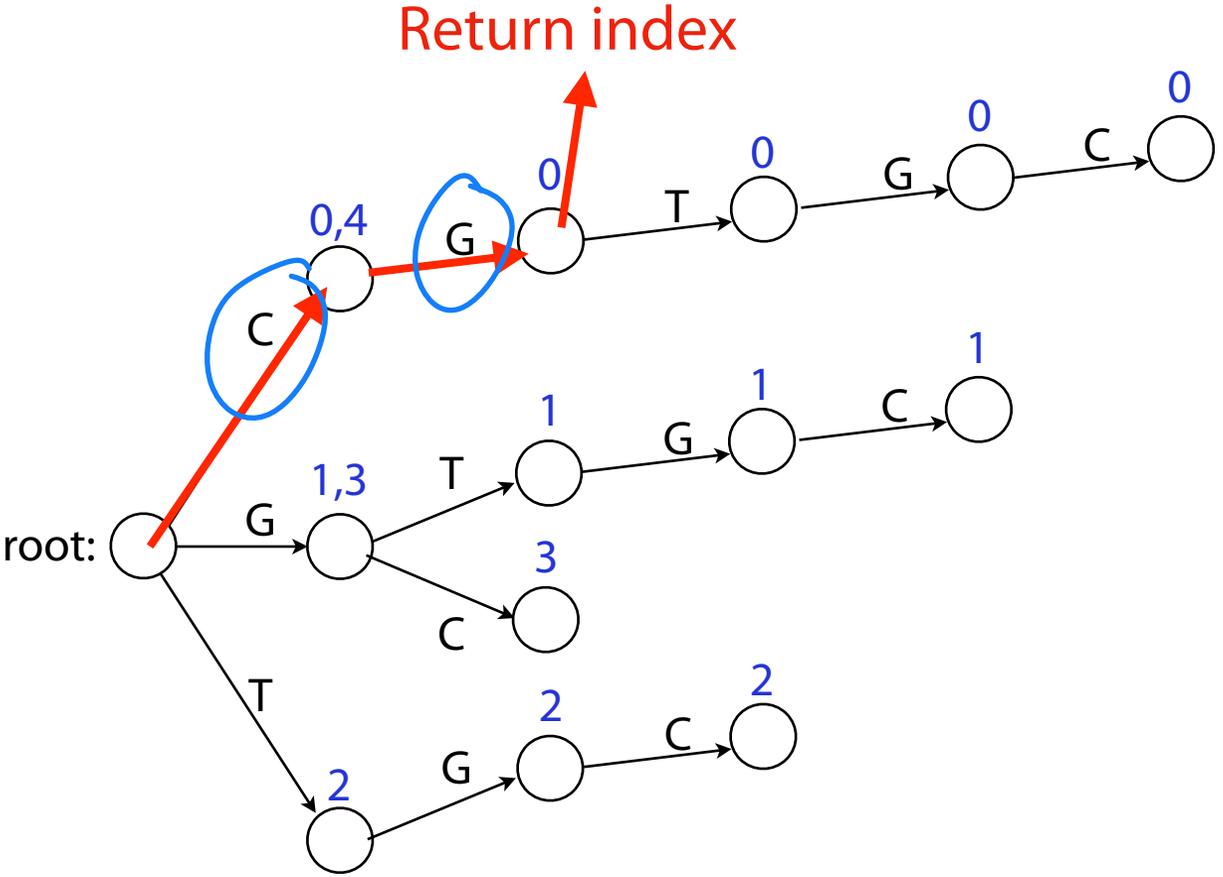


std::vector<int> searchPattern(P)

T:CGTG C

P = "CG"

| Key | Value |
|-----|-------|
| C | 0 |
| G | 1 |
| T | 2 |
| G | 3 |
| C | 4 |
| CG | 0 |
| GT | 1 |
| TG | 2 |
| ... | ... |



We can do exact pattern matching in $O(P)$ time!

Exact pattern matching *w/ indexing*

How can we be more efficient in our preprocessing?



Exact pattern matching *w/ indexing*

How can we be more efficient in our preprocessing?

1) Perform fewer insertions to store T



Runtime to preprocess

2) Store fewer values in index

Storage cost of index



Exact pattern matching *w/ indexing*

How can we be more efficient in our preprocessing?

1) Perform fewer insertions to store T

2) Store fewer values in index

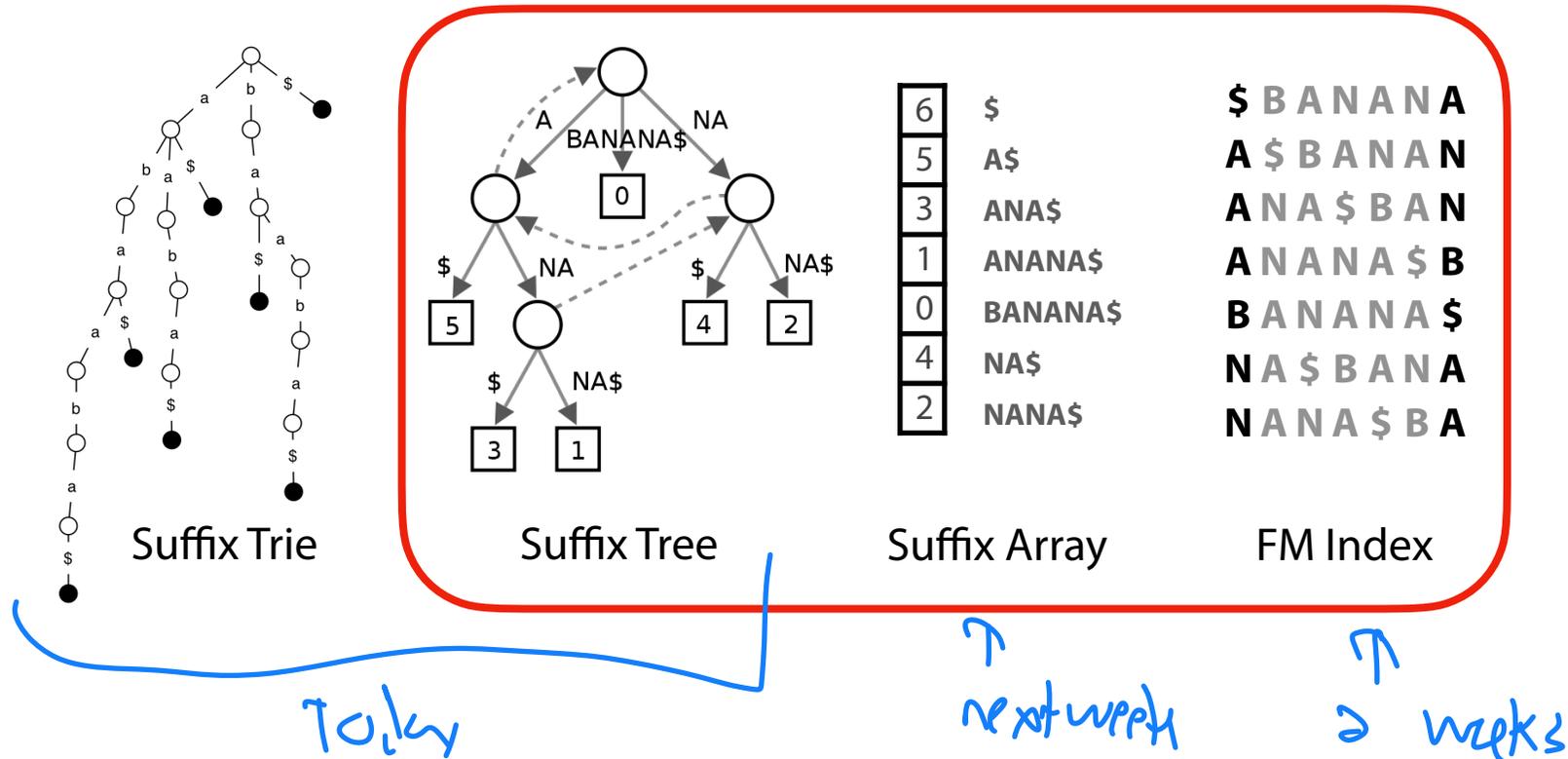
What if we just stored the suffixes?



Exact pattern matching *w/ indexing*

There are many data structures built on **suffixes**

Modern methods still use these today



Suffix Trie

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

T : C G T G C

C G T G C

G T G C

T G C

G C

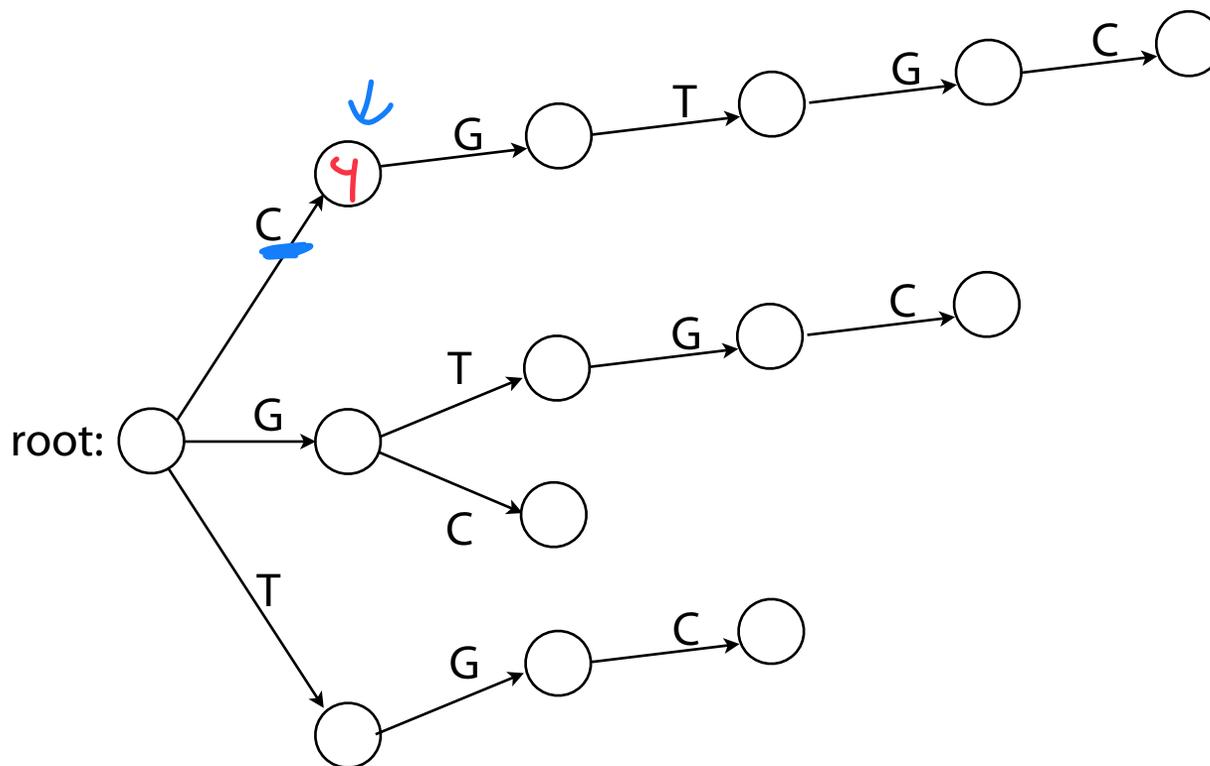
C

T suffixes

Suffix Trie

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

| | Key | Value |
|-----------------|-----|-------|
| T : C G T G C | | |
| C G T G C | | 0 |
| G T G C | | 1 |
| T G C | | 2 |
| G C | | 3 |
| C | | 4 |



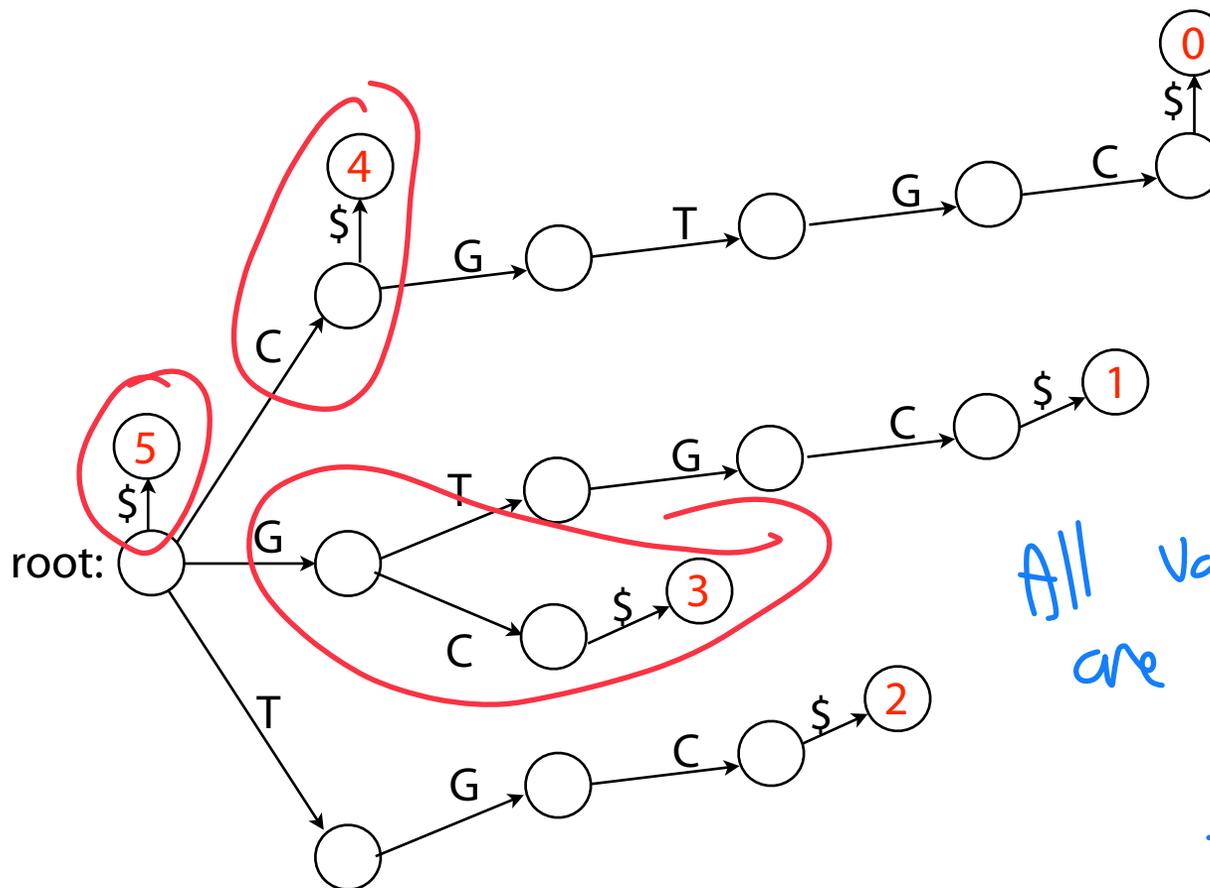
Inserting just T suffixes gets us the same tree*!

Suffix Trie

To prevent loss of information, add a **terminal character**

T: C G T G C \$

| Key | Value |
|---------|-------|
| CGTGC\$ | 0 |
| GTGC\$ | 1 |
| TGC\$ | 2 |
| GCS\$ | 3 |
| C\$ | 4 |
| \$ | 5 |



All values
are leaves
only

Every suffix now ends at a leaf

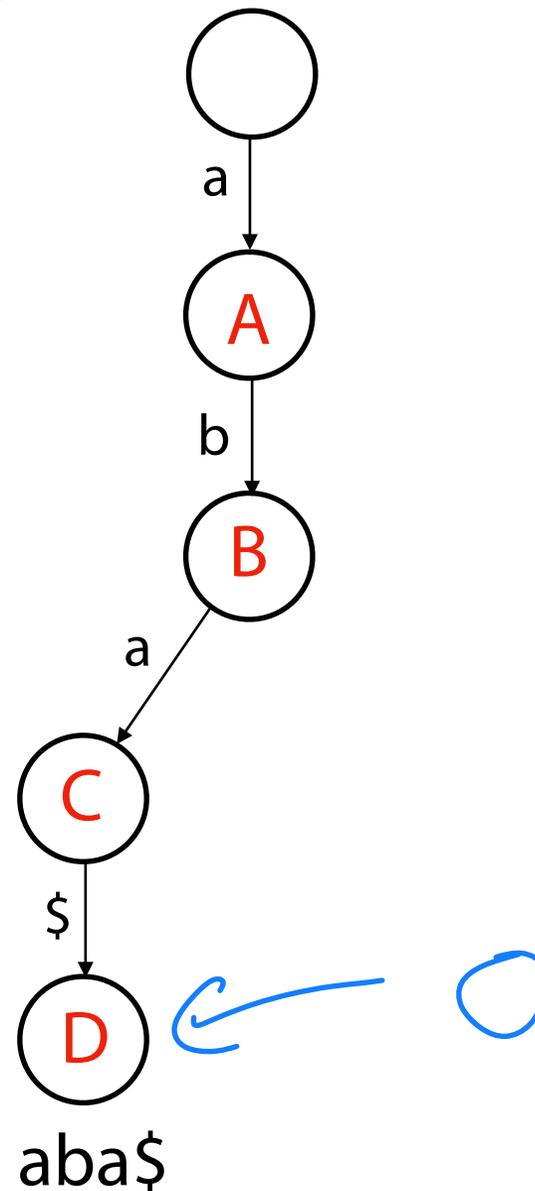
Suffix Trie Construction



Join Code: 225

T: a b a(\$)

| Key | Value |
|-------|-------|
| aba\$ | 0 |
| | 1 |
| | 2 |



Where do I put value?

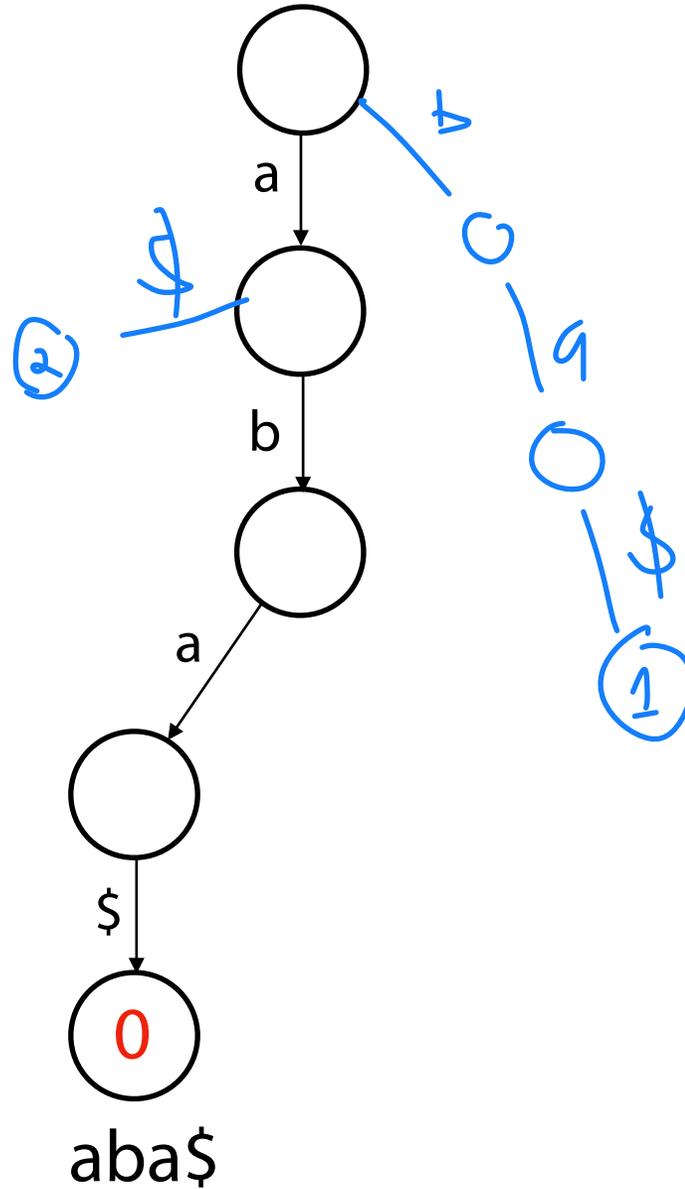
at last position
in string
will always be a leaf

Suffix Trie Construction

T: a b a \$

| Key | Value |
|-------|-------|
| aba\$ | 0 |
| ba\$ | 1 |
| a\$ | 2 |

What are my other keys?



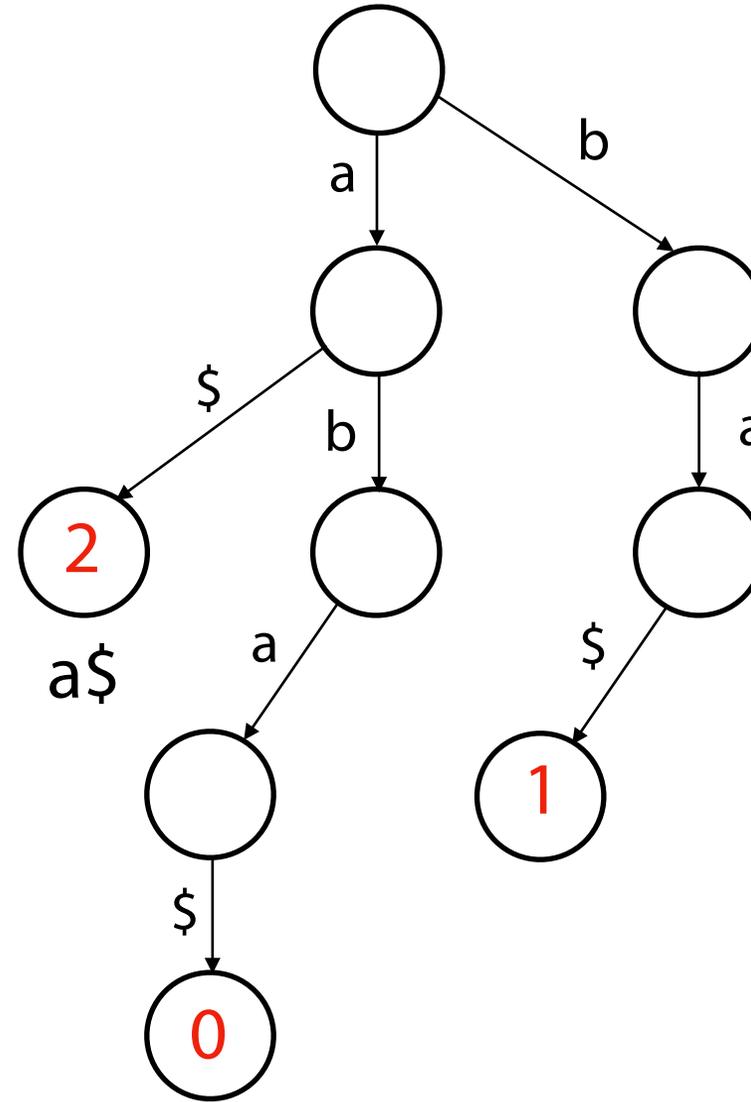
Suffix Trie Construction



Join Code: 225

T: a b a \$

| Key | Value |
|-------|-------|
| aba\$ | 0 |
| ba\$ | 1 |
| a\$ | 2 |



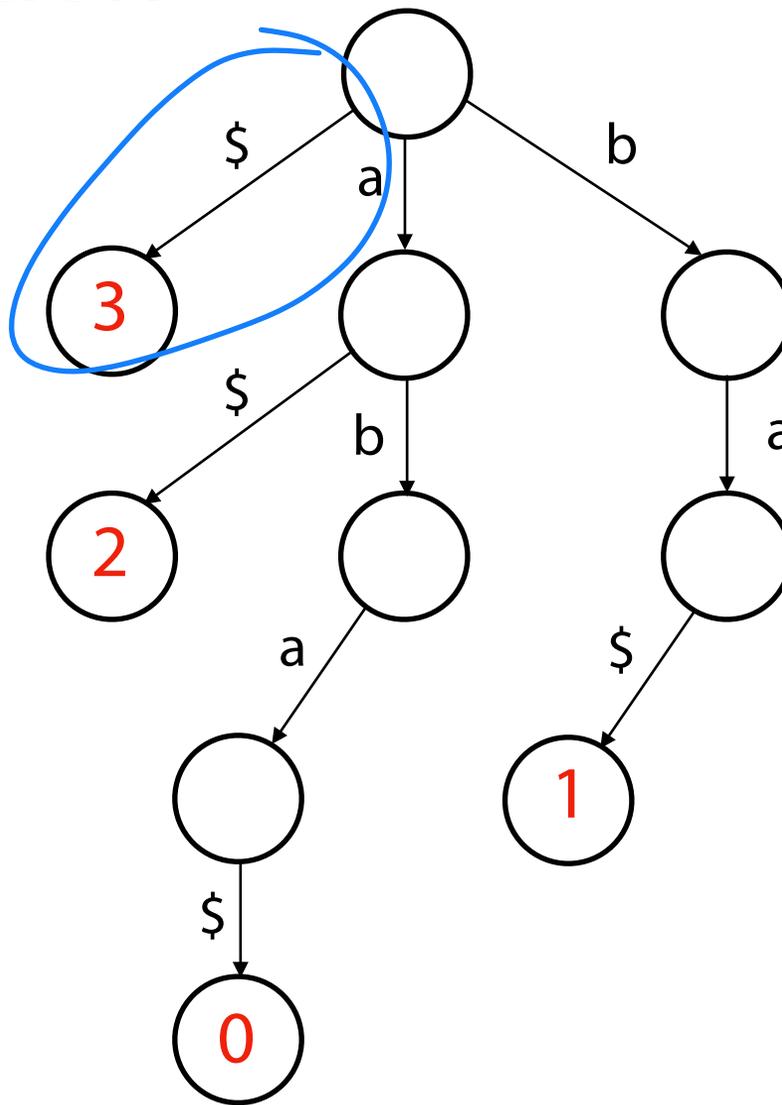
Are we done?

No we are missing the null string

Suffix Trie Construction

$T: a b a \$$

| Key | Value |
|-------|-------|
| aba\$ | 0 |
| ba\$ | 1 |
| a\$ | 2 |
| \$ | 3 |

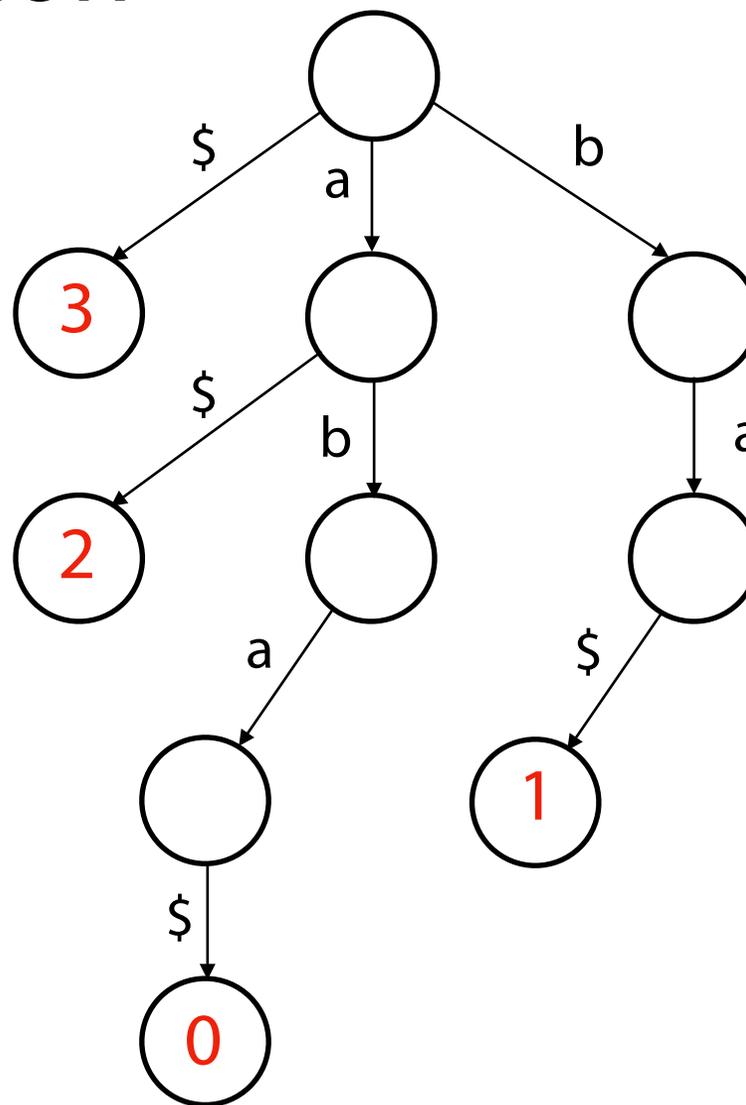


Suffix Trie Construction



T: a b a \$

| Key | Value |
|-------|-------|
| aba\$ | 0 |
| ba\$ | 1 |
| a\$ | 2 |
| \$ | 3 |



Handwritten notes in red ink showing the construction of suffixes: 'a a a \$', 'a / a \$ (3)', 'a / a / a \$ (2)', 'a / a / a / a \$ (2)', 'a / a / a / a / a \$ (2)', 'a / a / a / a / a / a \$ (2)', 'a / a / a / a / a / a / a \$ (2)', 'a / a / a / a / a / a / a / a \$ (2)'. The numbers in parentheses are circled in blue.

Every suffix now ends at a leaf and only leaves store values

NaryTree Modification I and II

stree.h

```
1 class NaryTree
2 {
3     public:
4         struct Node {
5             int index;
6             std::map<std::string, Node*> children;
7
8             Node(std::string s, int i)
9             {
10                if(s.length() > 0 ){
11                    std::string f = s.substr(0,1);
12                    children[f] = new Node(s.substr(1), i );
13                } else {
14                    index = i;
15                }
16            }
17        };
18
19    protected:
20        Node* root;
21    ...
```

used to be vector

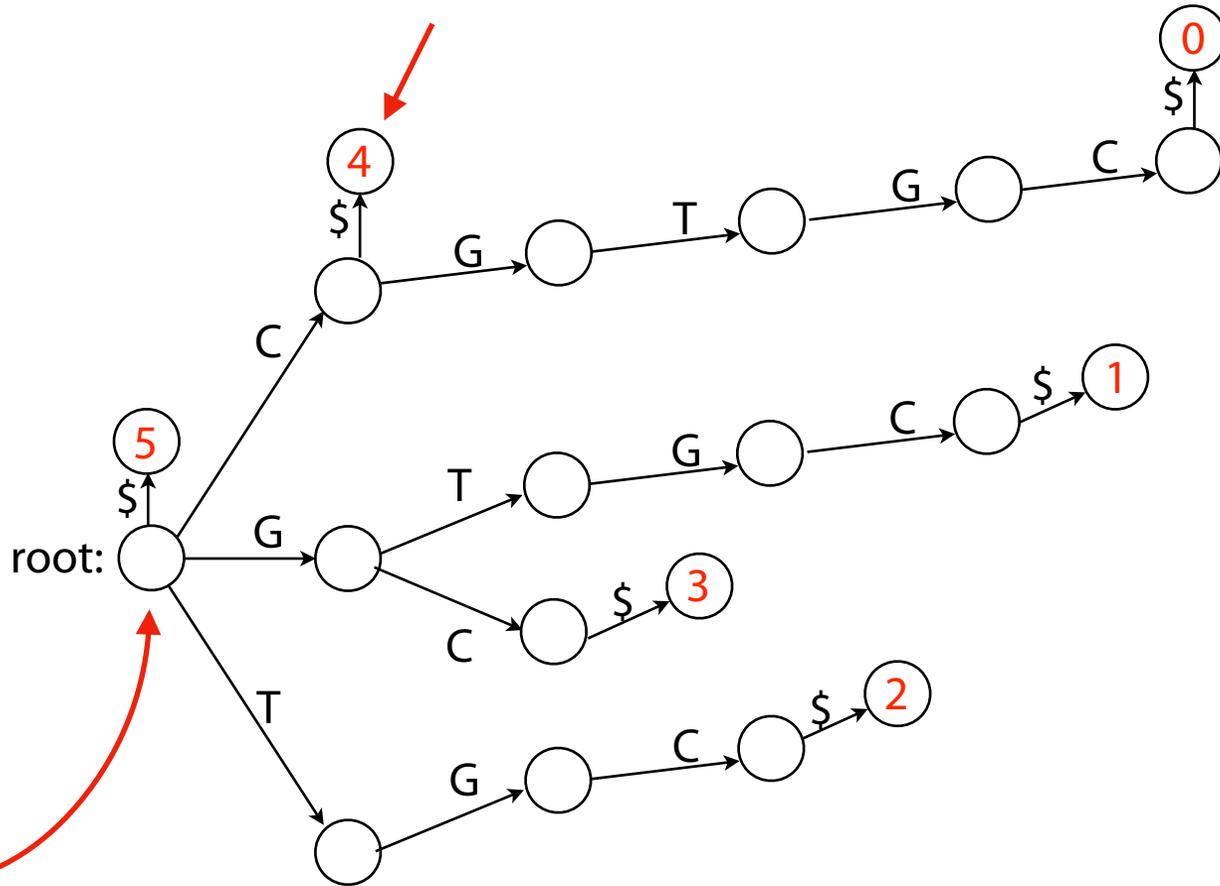
we haven't
seen why
yet!

NaryTree build_strie(std::string T)

add terminal char
T: C G T G C (\$)

Store $|T| + 1$ values

| Key | Value |
|---------|-------|
| CGTGC\$ | 0 |
| GTGC\$ | 1 |
| TGC\$ | 2 |
| GC\$ | 3 |
| C\$ | 4 |
| \$ | 5 |



Perform $|T| + 1$ insertions

Assignment 6: a_stree

Learning Objective:

Use an existing implementation of a suffix trie as a N-ary Tree

Implement exact pattern matching using a suffix trie

Construct a suffix tree from a suffix trie

Suffix Trie Search

Each of T 's substrings is spelled out along a path from the root.

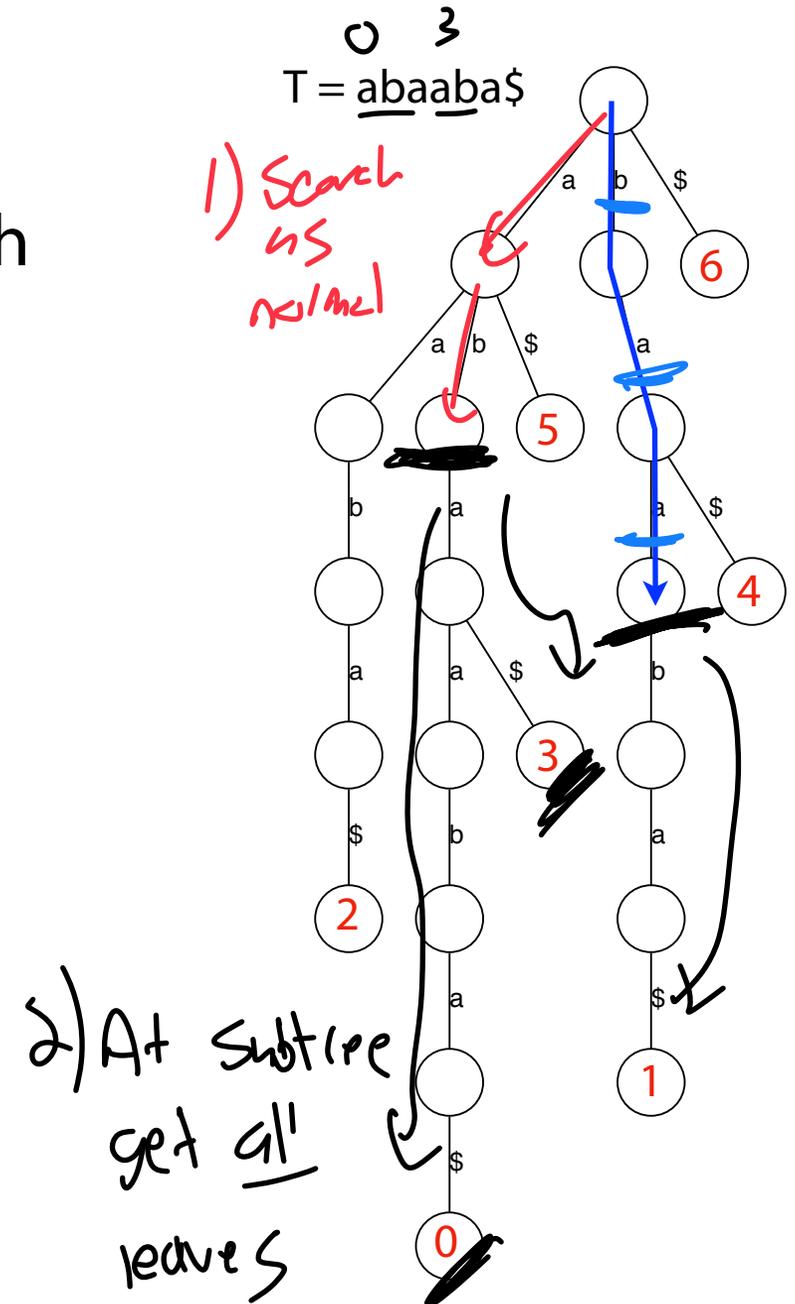
Every **substring** is a **prefix** of some **suffix**

$P = baa$

$\text{Search}(P) = 1$

$P = ab$

$\text{Search}(P) = 0, 3$



Suffix Trie Search

Each of T 's substrings is spelled out along a path from the root.

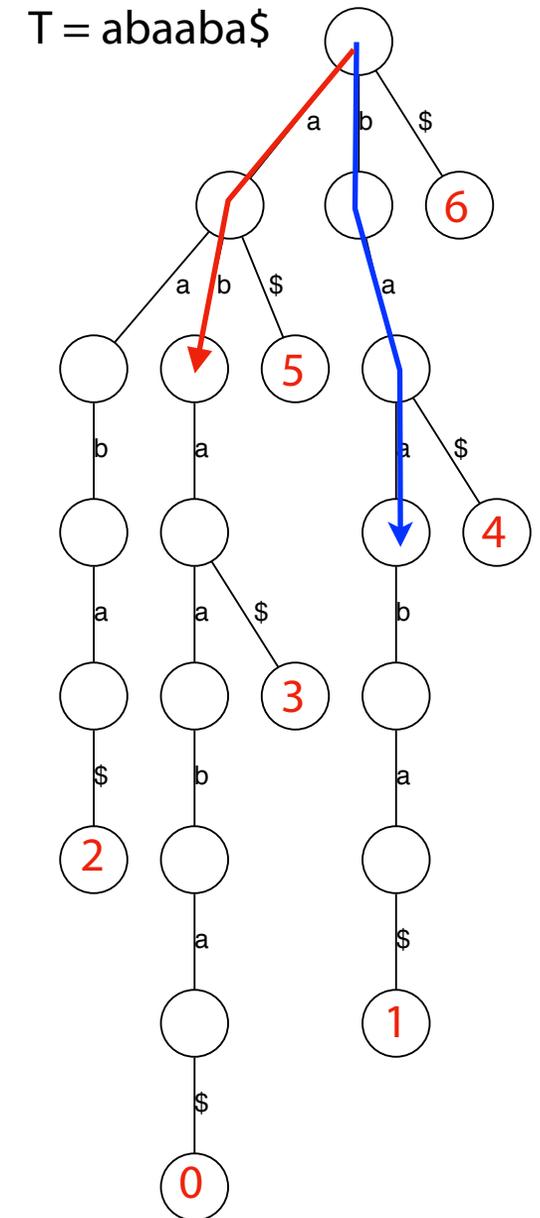
Every **substring** is a **prefix** of some **suffix**

$P = baa$

$\text{Search}(P) = 1$

$P = ab$

$\text{Search}(P) = 0, 3$



Suffix Trie Search

Each of T 's substrings is spelled out along a path from the root.

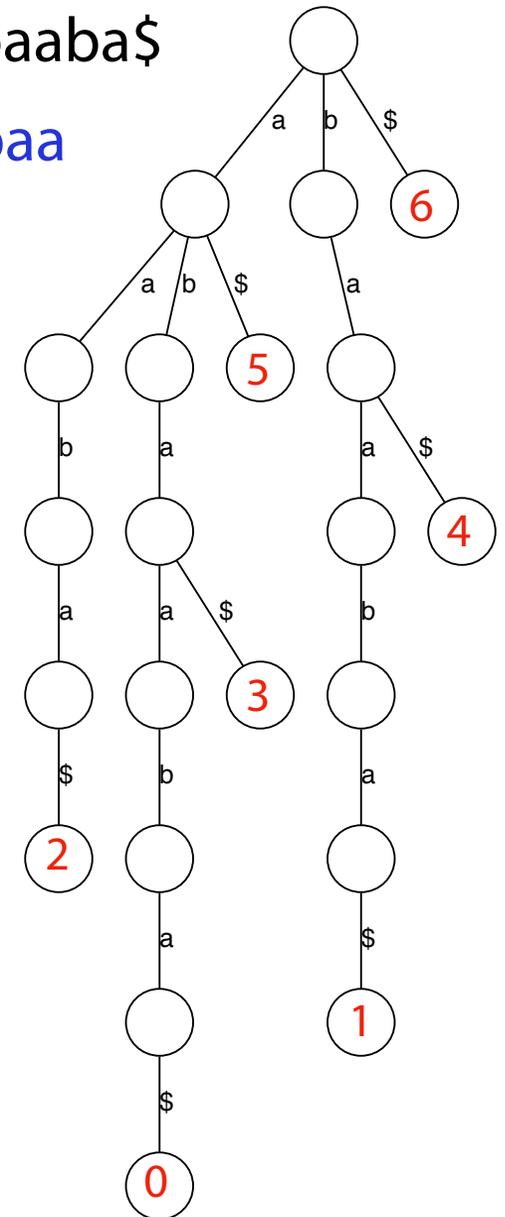
Every **substring** is a **prefix** of some **suffix**

Starting at root:

- (0) If P empty, **return values of all leaves.**
- (1) Try to match front character
- (2) If match, move to appropriate child
 - (2.5) Set pattern equal to remainder
 - (2.5) Go back to (0)
- (3) If mismatch, P is not a substring! ~~X~~

$T = \text{abaaba}\$$

$P = \text{abaa}$



Suffix Trie Search

Each of T 's substrings is spelled out along a path from the root.

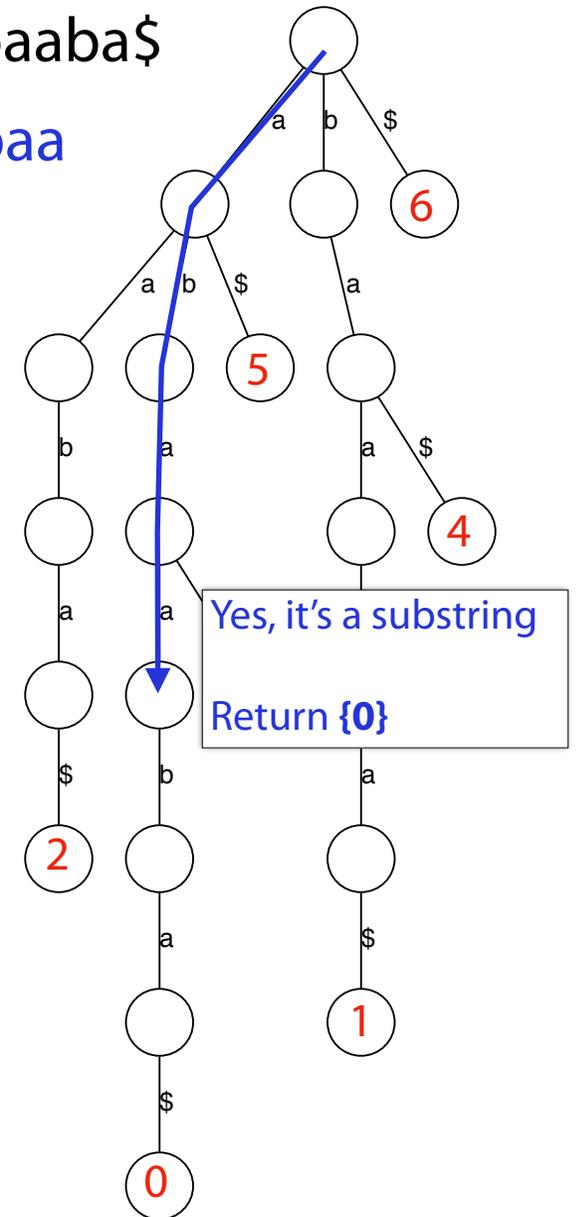
Every **substring** is a **prefix** of some **suffix**

Starting at root:

- (0) If P empty, **return values of all leaves.**
- (1) Try to match front character
- (2) If match, move to appropriate child
 - (2.5) Set pattern equal to remainder
 - (2.5) Go back to (0)
- (3) If mismatch, P is not a substring!

$T = \text{abaaba}\$$

$P = \text{abaa}$



Suffix Trie Search

Each of T 's substrings is spelled out along a path from the root.

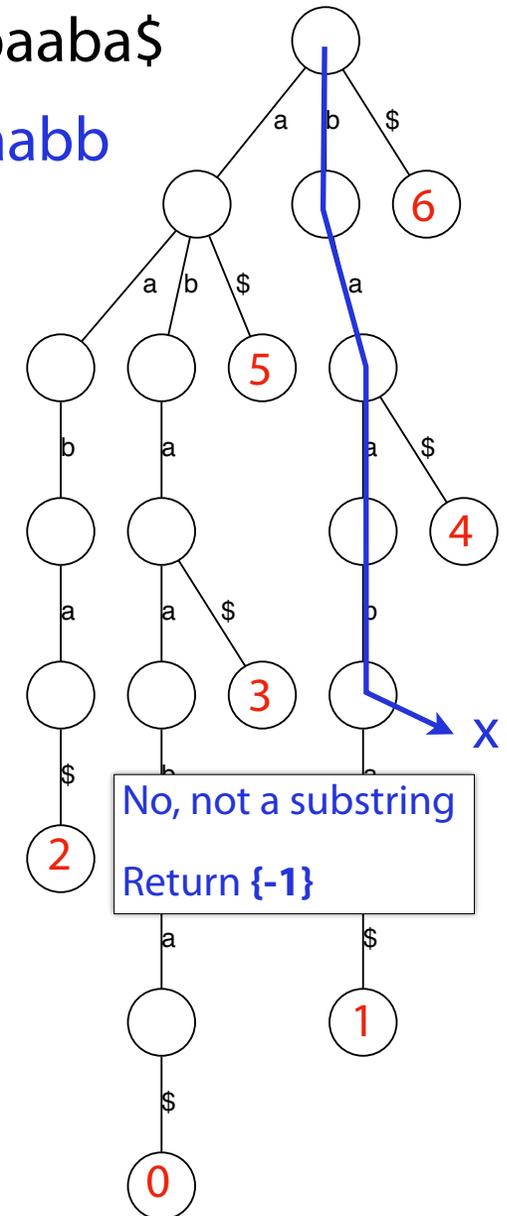
Every **substring** is a **prefix** of some **suffix**

Starting at root:

- (0) If P empty, **return values of all leaves.**
- (1) Try to match front character
- (2) If match, move to appropriate child
 - (2.5) Set pattern equal to remainder
 - (2.5) Go back to (0)
- (3) If mismatch, P is not a substring!

$T = \text{abaaba}\$$

$P = \text{baabb}$



Suffix Trie

How does the suffix trie grow with $|T| = m$?

Is there a class of string where the number of suffix trie nodes grows linearly with m ?

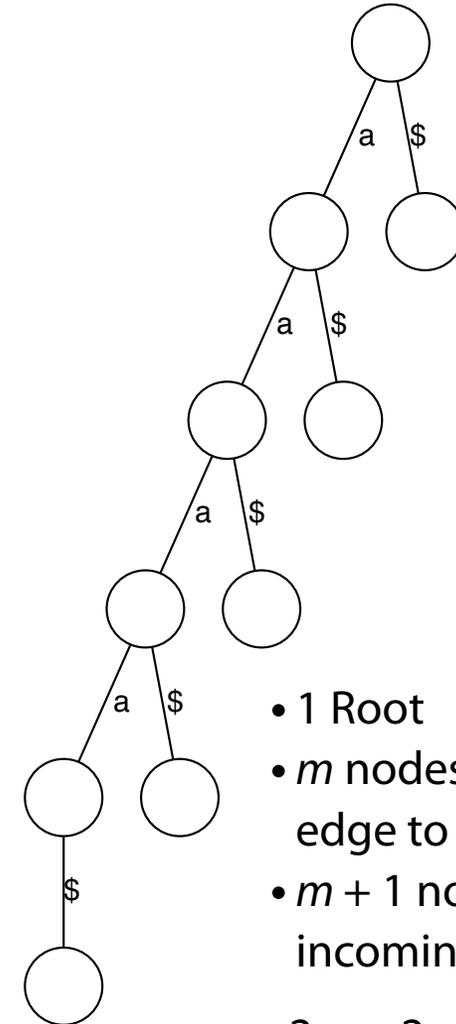
Suffix Trie

How does the suffix trie grow with $|T| = m$?

Is there a class of string where the number of suffix trie nodes grows linearly with m ?

Yes: a string of m a's in a row (a^m)

$T = aaaa\$$



- 1 Root
 - m nodes with "a" edge to parent
 - $m + 1$ nodes with incoming '\$' edge
- $2m + 2$ nodes

Suffix Trie

How does the suffix trie grow with $|T| = m$?

Is there a class of string where the number of suffix trie nodes grows with m^2 ?

Suffix trie: actual growth

Built suffix tries for the first 500 prefixes of a virus genome

Black curve shows how # nodes increases with prefix length

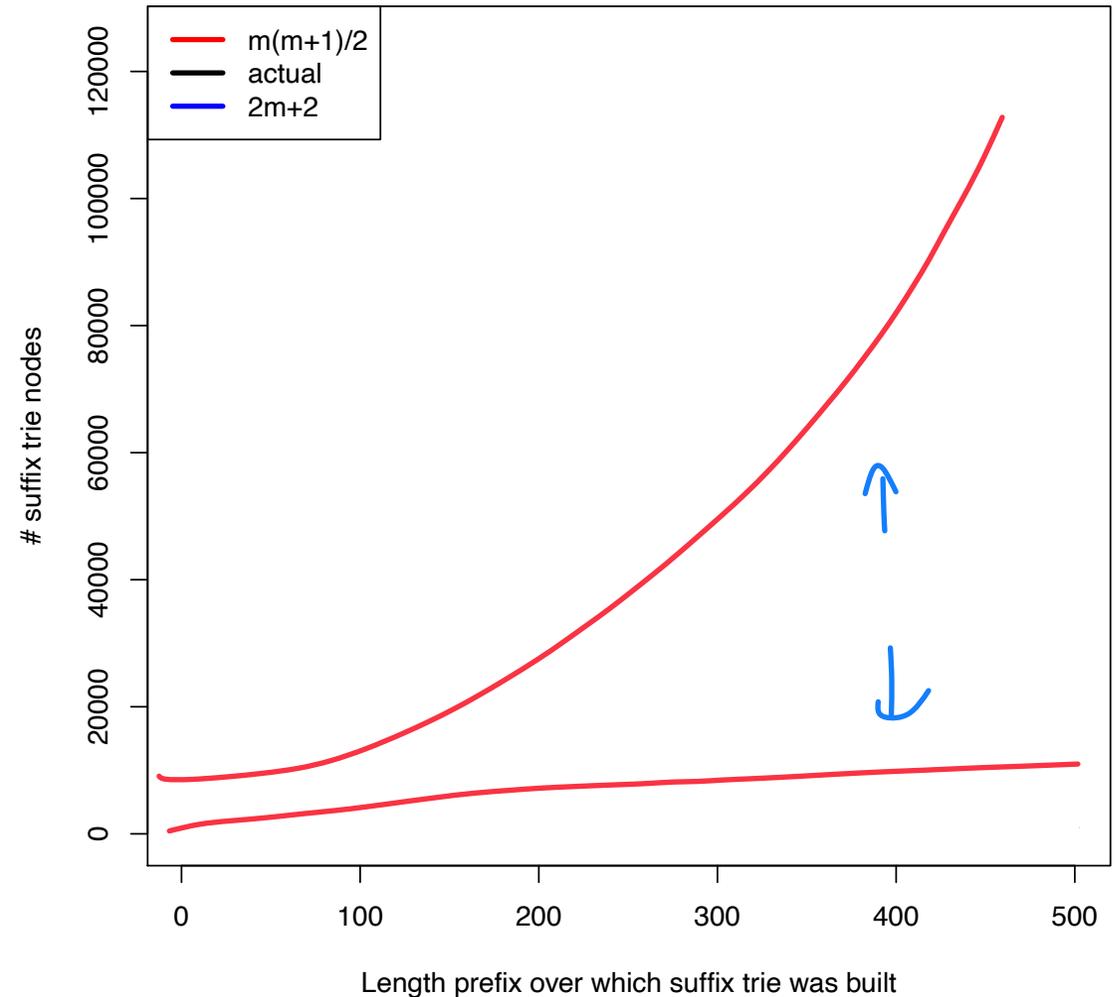


Figure & example by Ben Langmead

Suffix trie: actual growth

Built suffix tries for the first 500 prefixes of a virus genome

Black curve shows how # nodes increases with prefix length

Actual growth *much* closer to worst case than to best!

Search is $O(|A| + |T|^2)$

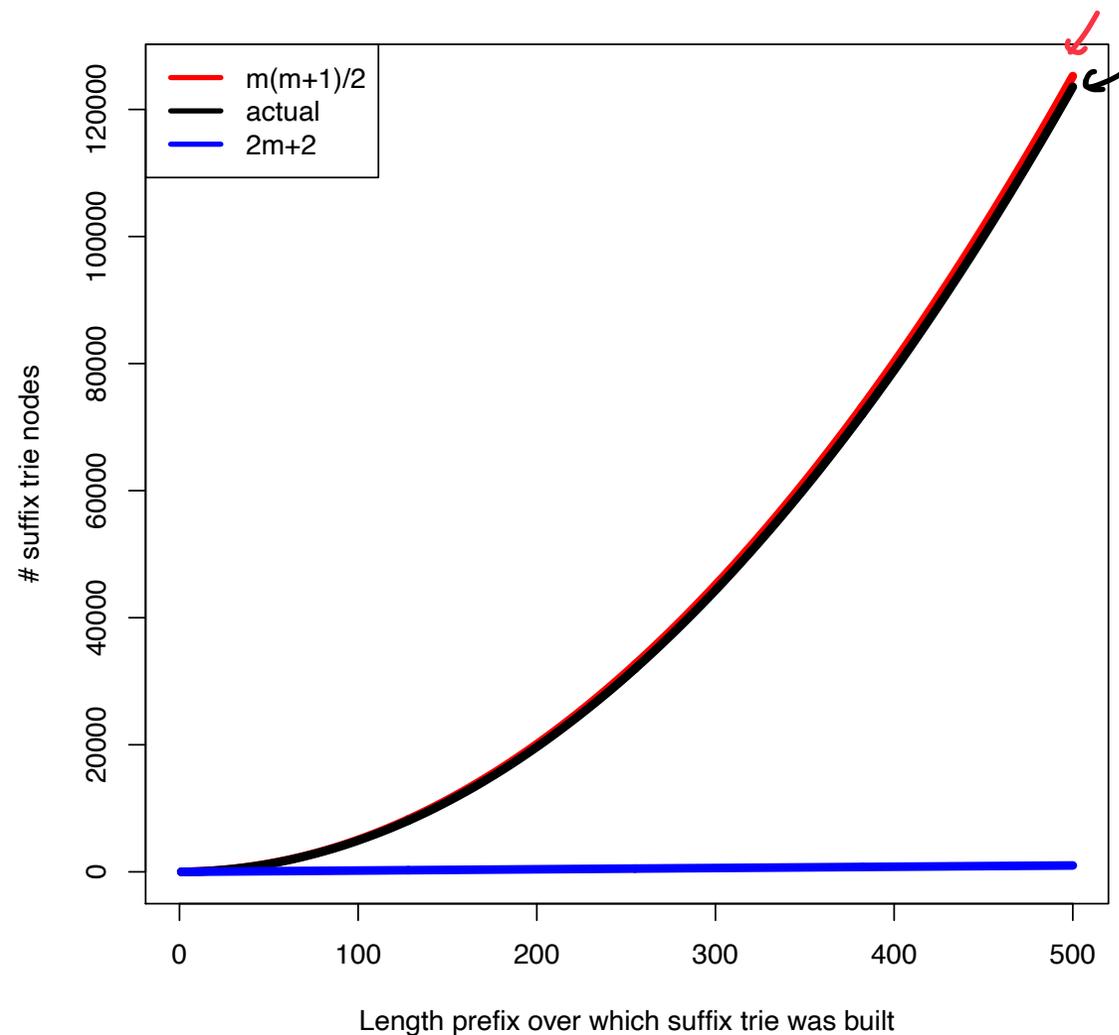


Figure & example by Ben Langmead

Assignment 6: a_stree



Learning Objective:

Use an existing implementation of a suffix trie as a N-ary Tree

Implement exact pattern matching using a suffix trie

Construct a suffix tree from a suffix trie

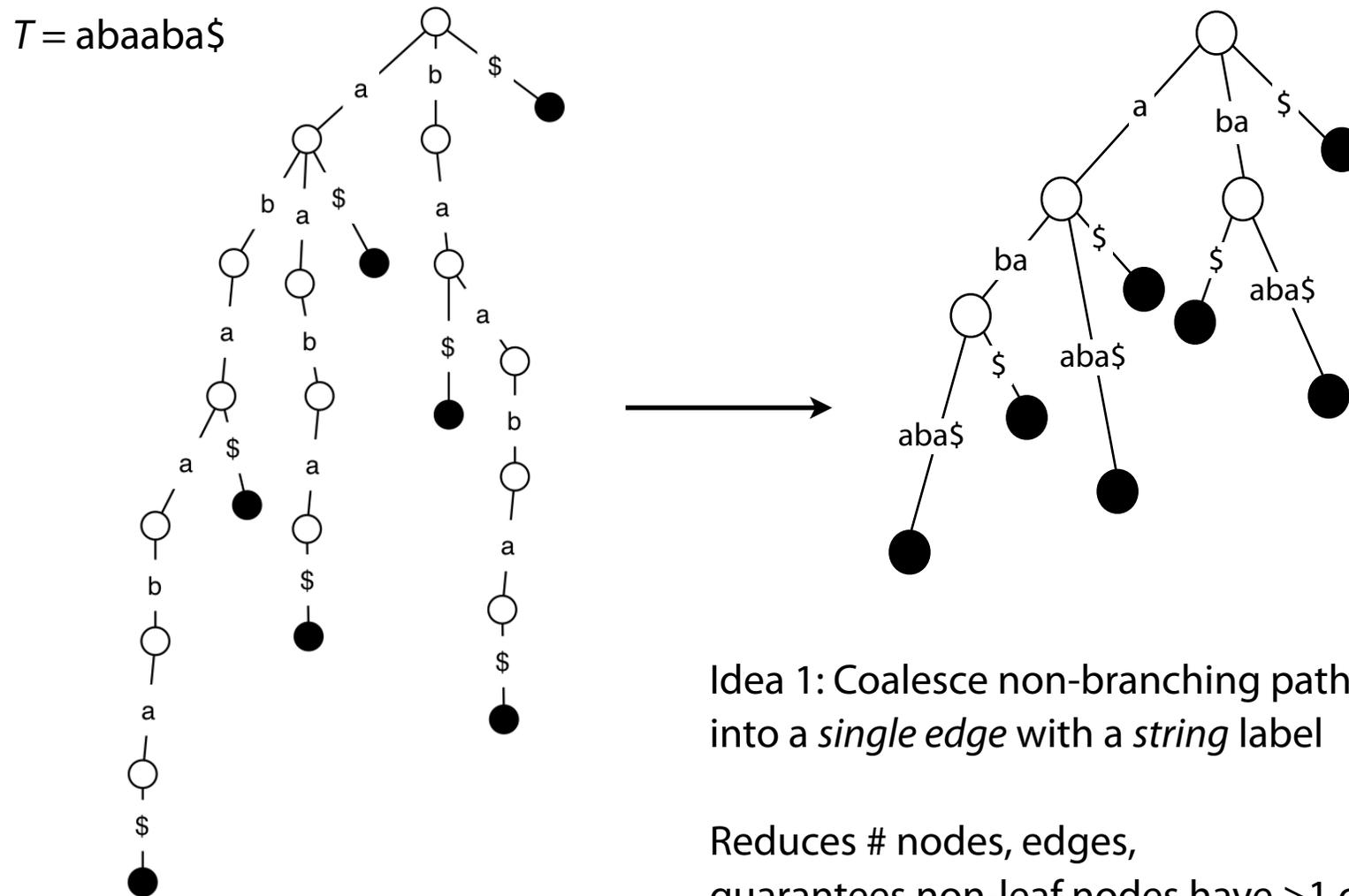


Even worse?

Minor gain in construction

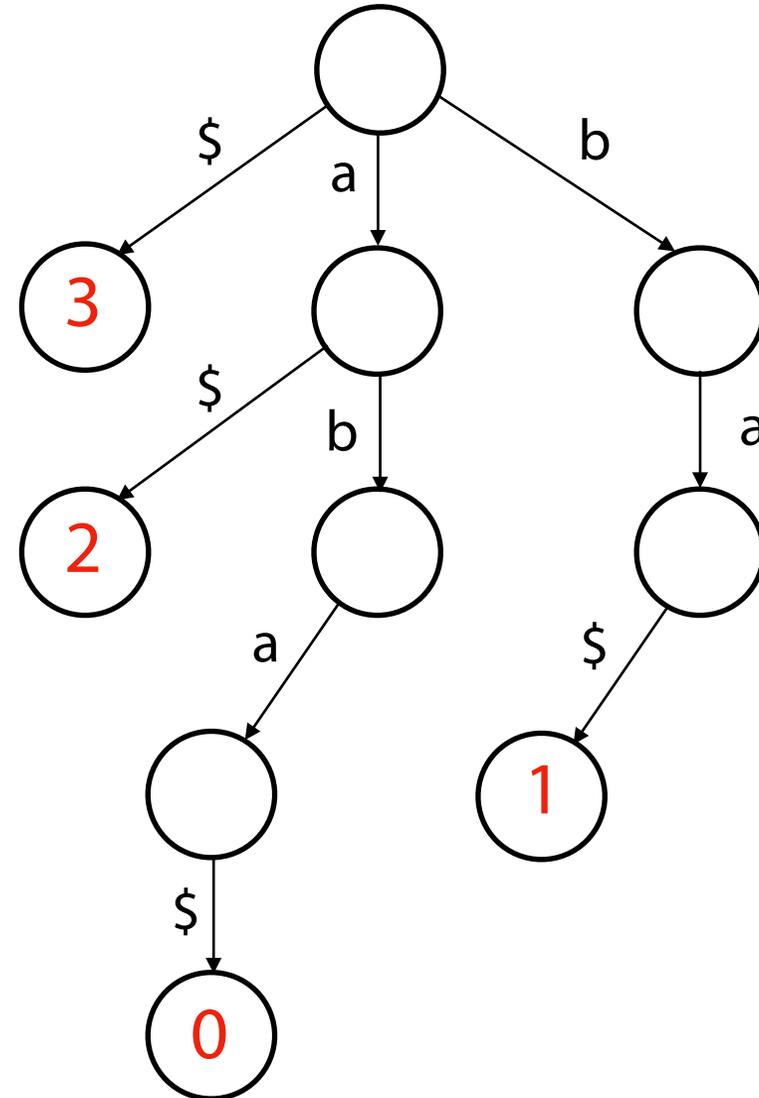
Major loss in search time

Suffix Trie: Making it smaller



Coalescing edges

We want to coalesce paths that don't branch.



Coalescing edges



Join Code: 225

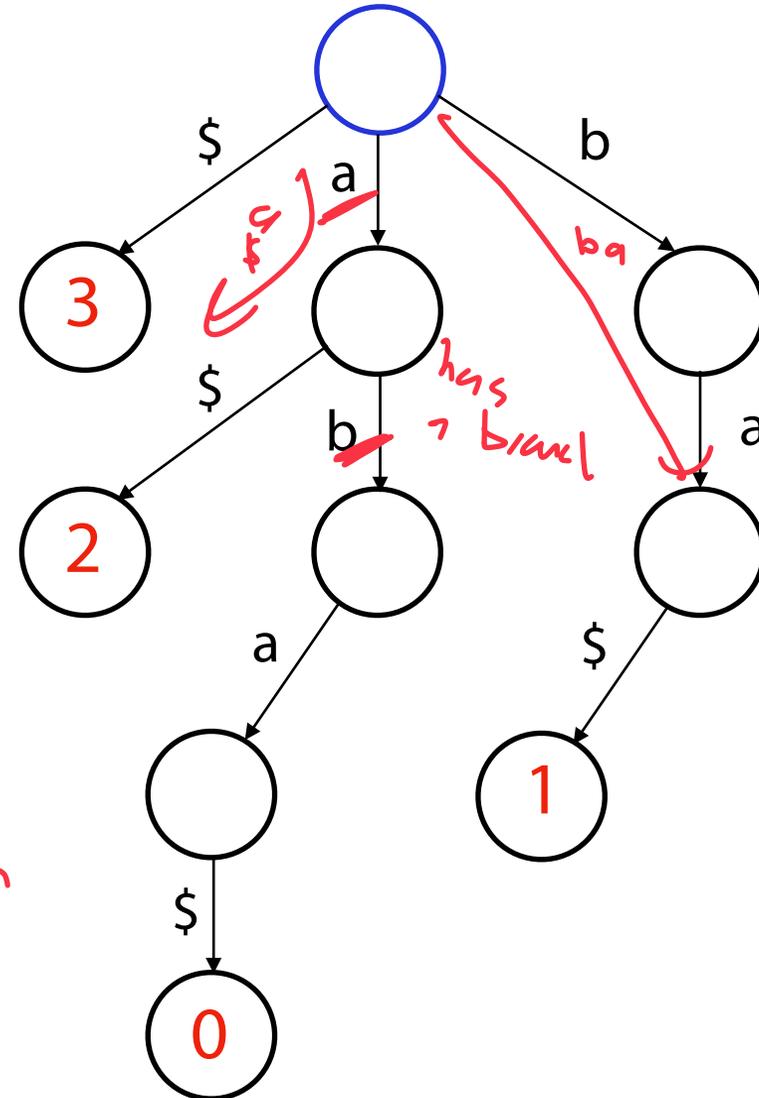
We want to coalesce paths that don't branch.

'Current root' in blue

Coalesce '\$'? No, it's already as small as possible

Coalesce 'a'? No, can't coalesce if node has two children

Coalesce 'b'? Yes



Coalescing edges

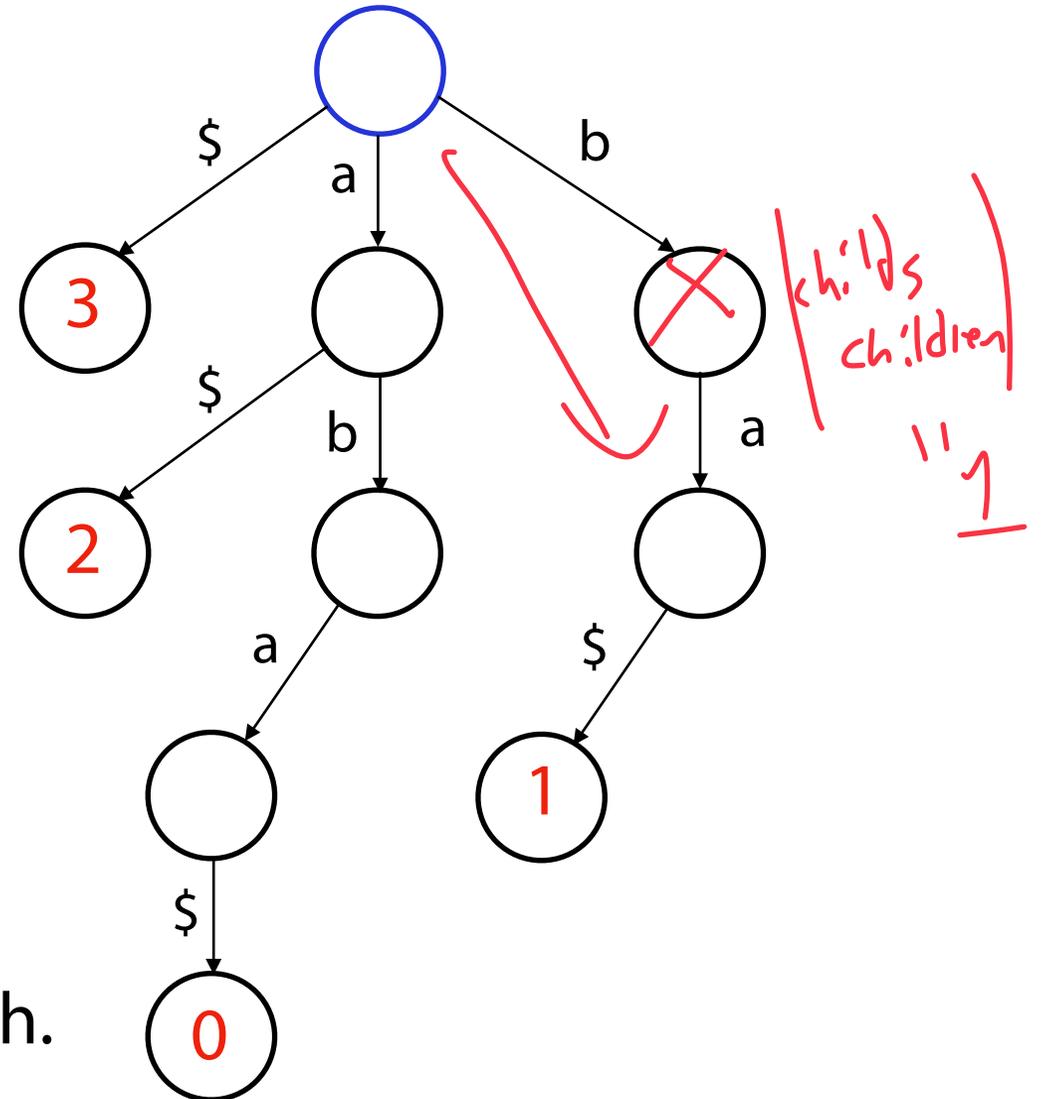
We want to coalesce paths that don't branch.

'Current root' in blue

Coalesce '\$'? **No**, nothing to merge

Coalesce 'a'? **No**, child has a branch

Coalesce 'b'? **Yes**, b->a is the only path.



Coalescing edges

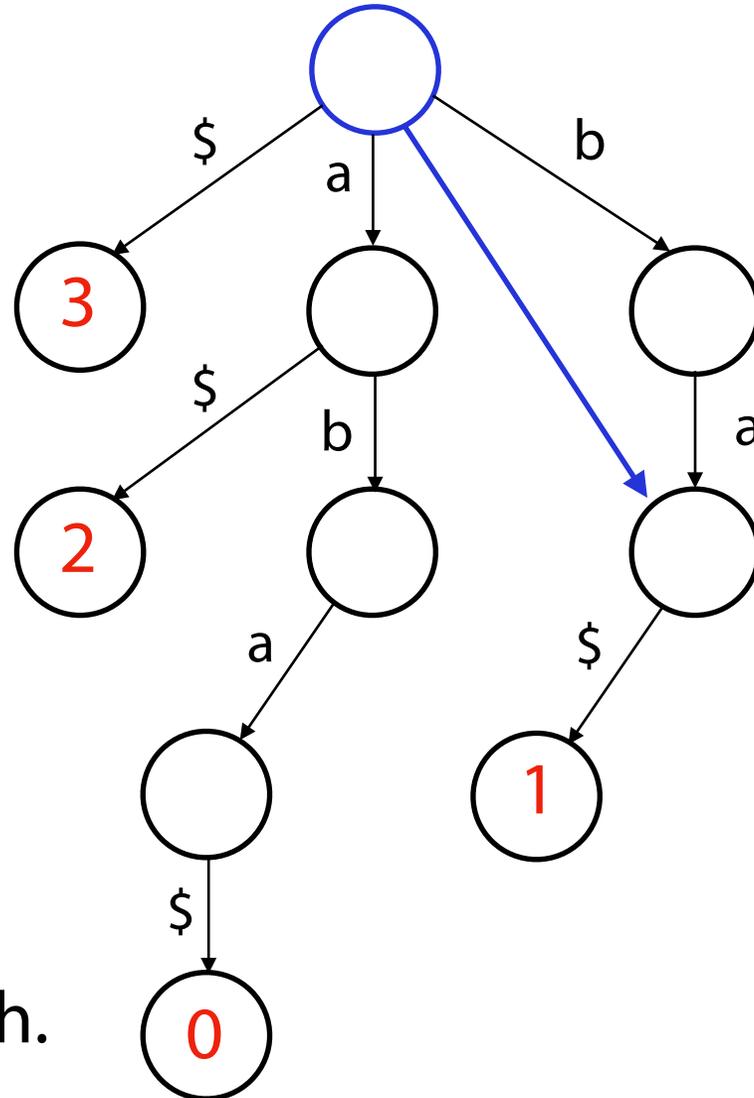
We want to coalesce paths that don't branch.

'Current root' in blue

Coalesce '\$'? **No**, nothing to merge

Coalesce 'a'? **No**, child has a branch

Coalesce 'b'? **Yes**, b->a is the only path.



Coalescing edges

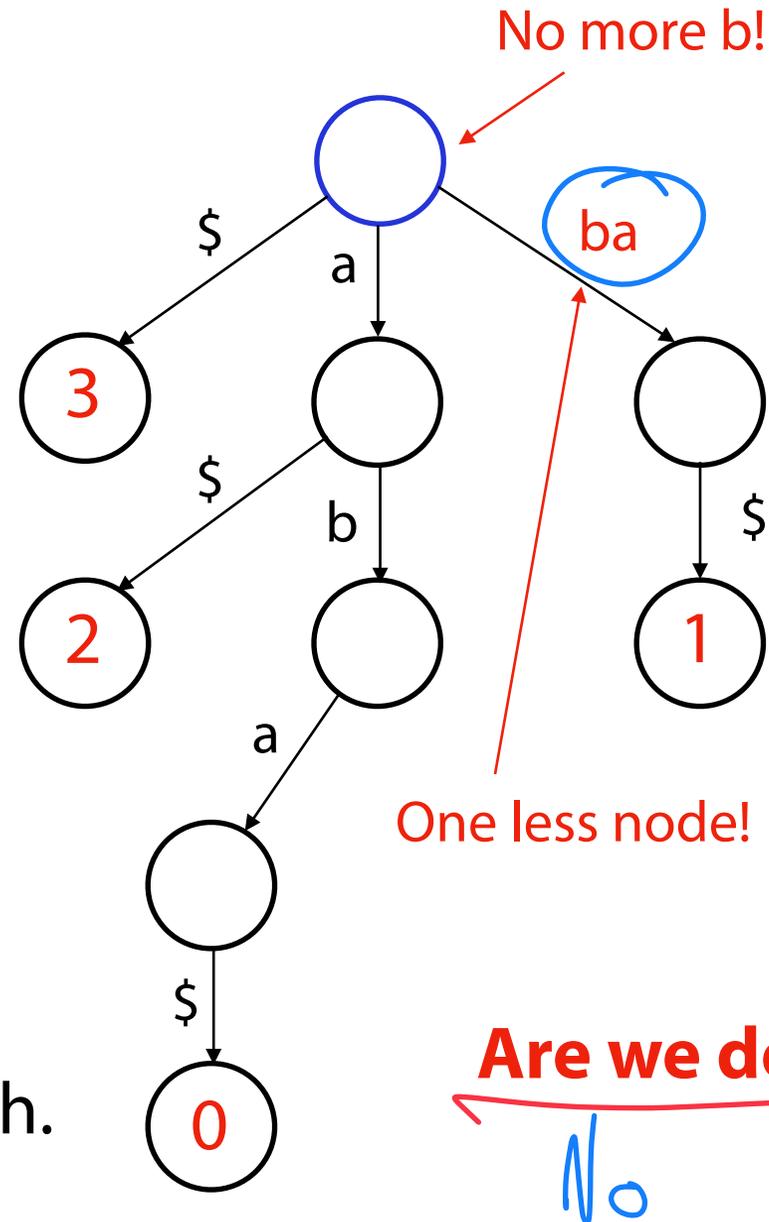
We want to coalesce paths that don't branch.

'Current root' in blue

Coalesce '\$'? **No**, nothing to merge

Coalesce 'a'? **No**, child has a branch

Coalesce 'b'? **Yes**, b->a is the only path.



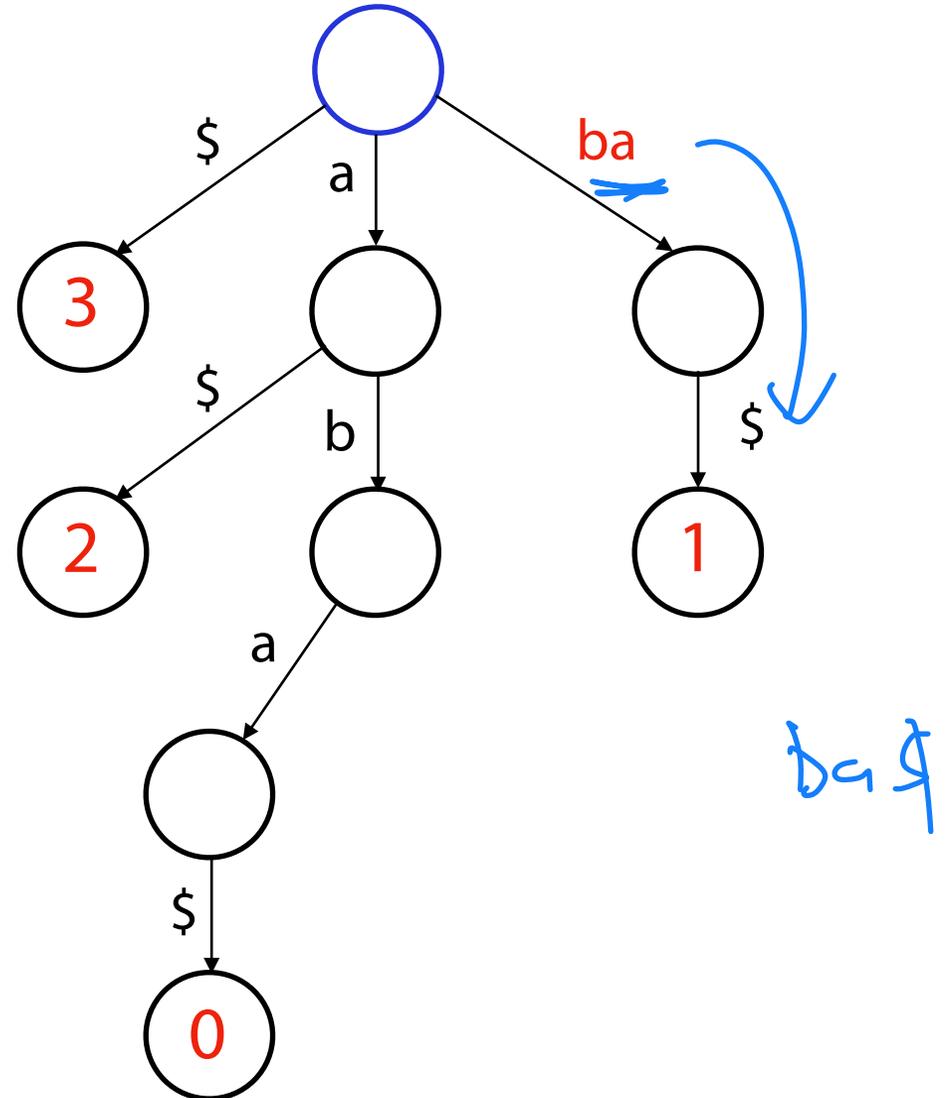
Coalescing edges

We want to coalesce paths that don't branch.

'Current root' in blue

We added a new edge 'ba'!

We might need to coalesce again!



Coalescing Edges



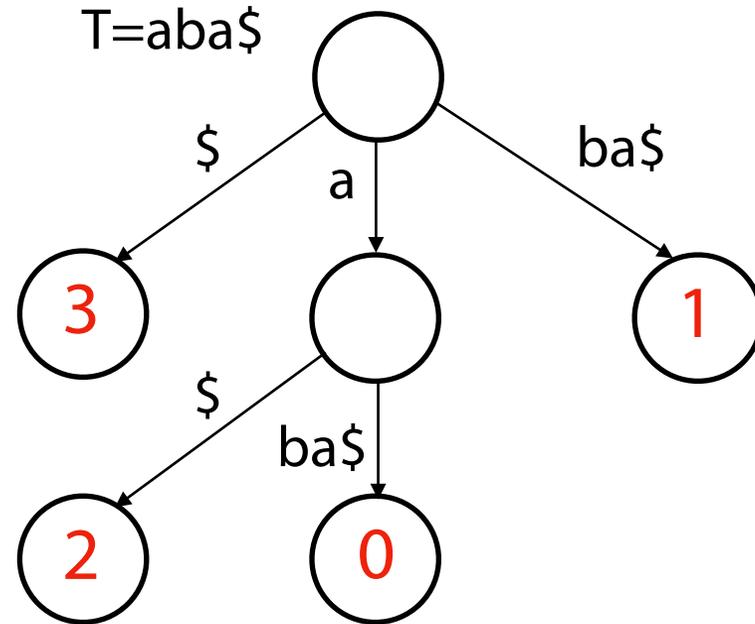
Coalesce all paths that don't branch.

A suffix tree of $|T| = m$ (counting '\$' in T) should have:

m leaves*

$\leq m - 1$ internal nodes

Each internal node ≥ 2 children



How to do this is up to you (and for you to work out)!

Iterator on Dynamic Data

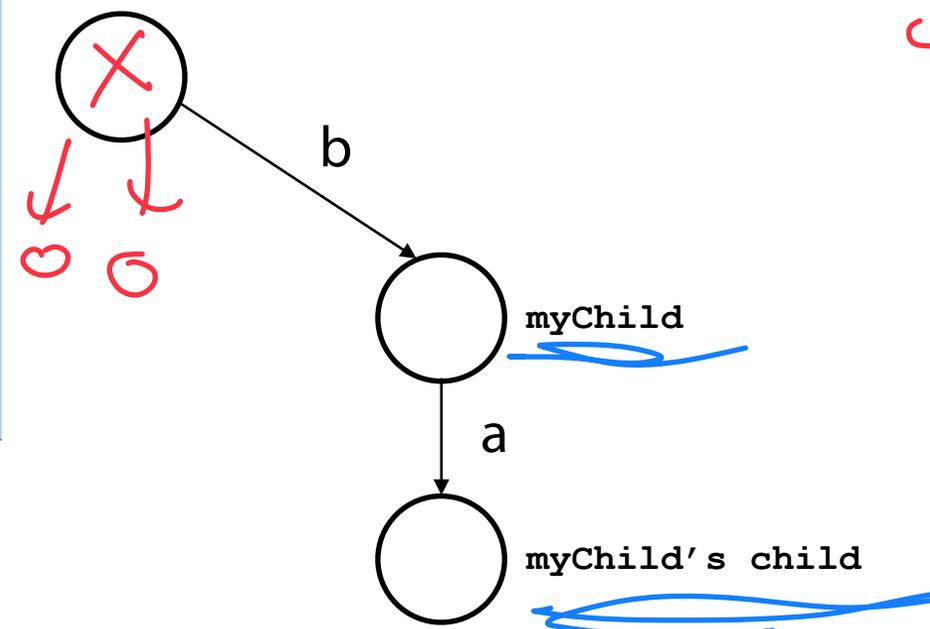
stree.cpp

Use over while loop
iterator

```
1 map<string,Node*>::iterator it = myMap.begin();
2
3 while(it != myMap.end()){
4     Node* myChild = it->second;
5
6
7     if < LOGIC STATEMENT >{
8         Node* temp = < myChild's child >;
9
10        myMap["NewEdge"] = temp;
11
12        delete myChild;
13
14        myMap.erase(it++);
15
16    }
17 }
18
19
20
21
```

children of some node X

maybe only one child?



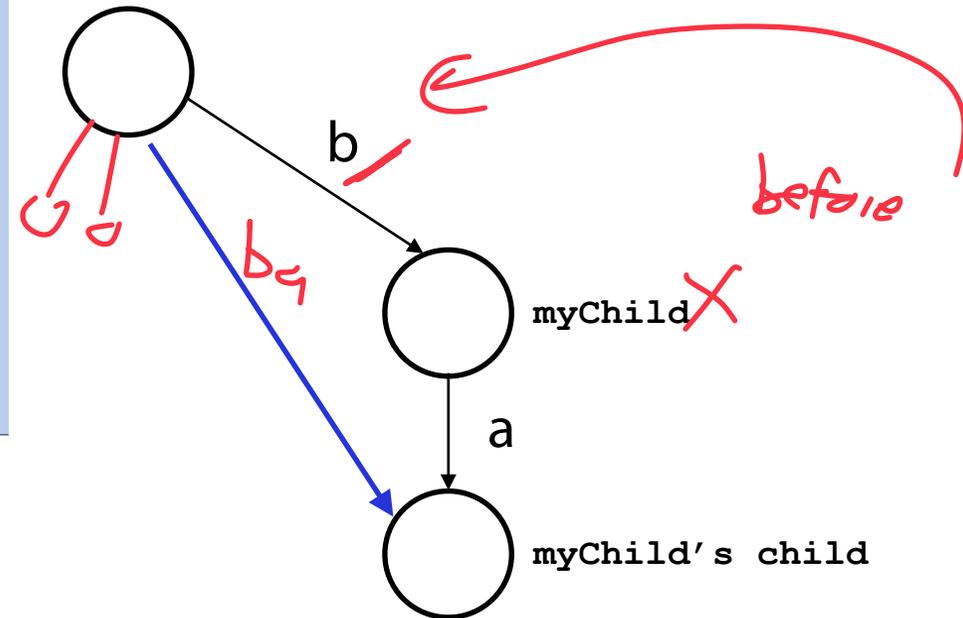
Iterator on Dynamic Data

stree.cpp

```
1 map<string,Node*>::iterator it = myMap.begin();
2
3 while(it != myMap.end()){
4
5     Node* myChild = it->second;
6
7     if < LOGIC STATEMENT >{
8         Node* temp = < myChild's child >;
9
10        myMap["NewEdge"] = temp;
11
12        delete myChild;
13
14        myMap.erase(it++);
15
16    }
17 }
18
19
20
21
```



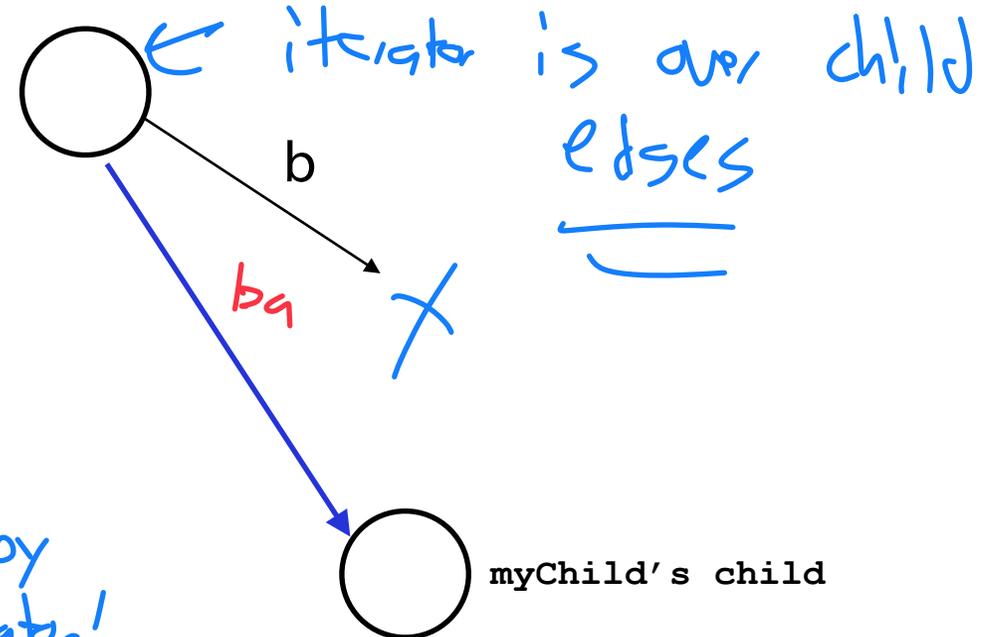
Add new edge



Iterator on Dynamic Data

stree.cpp

```
1 map<string,Node*>::iterator it = myMap.begin();
2
3 while(it != myMap.end()){
4
5     Node* myChild = it->second;
6
7     if < LOGIC STATEMENT >{
8         Node* temp = < myChild's child >;
9
10        myMap["NewEdge"] = temp;
11
12        delete myChild;
13
14        myMap.erase(it++);
15
16    }
17 }
18
19
20
21
```

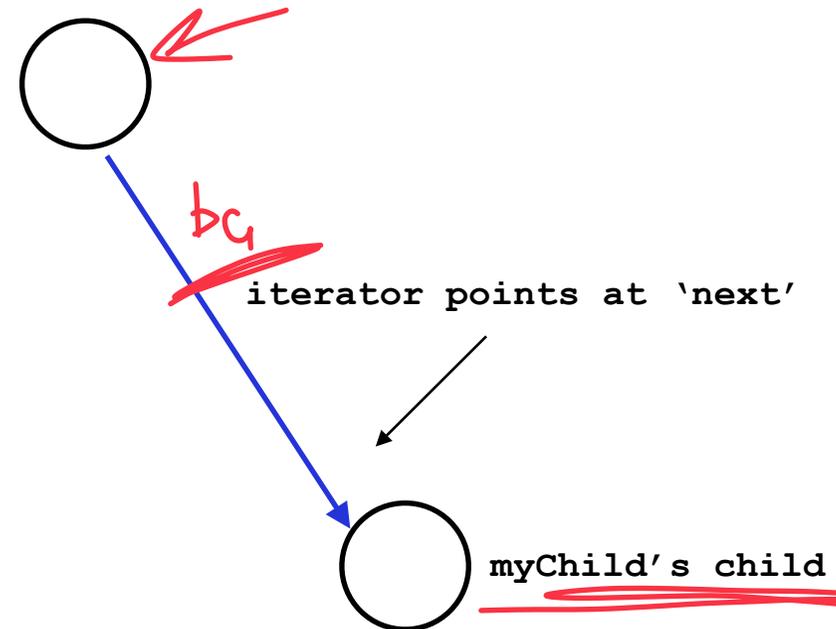


Creates a copy
of my iterator!

Iterator on Dynamic Data

stree.cpp

```
1 map<string,Node*>::iterator it = myMap.begin();
2
3 while(it != myMap.end()){
4
5     Node* myChild = it->second;
6
7     if < LOGIC STATEMENT >{
8         Node* temp = < myChild's child >;
9
10        myMap["NewEdge"] = temp;
11
12        delete myChild;
13
14        myMap.erase(it++);
15
16    }
17 }
18
19
20
21
```

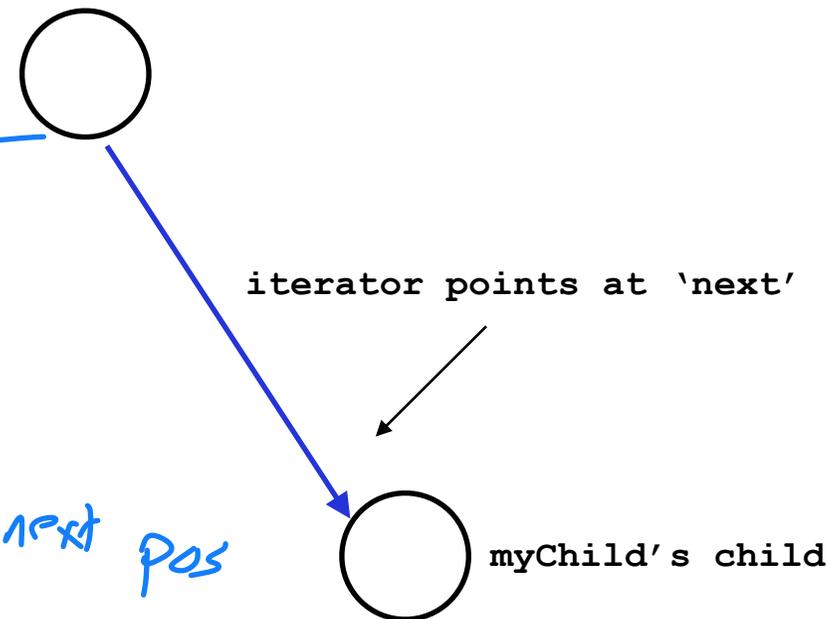


Iterator on Dynamic Data

stree.cpp

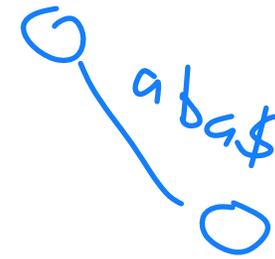
```
1 map<string,Node*>::iterator it = myMap.begin();
2
3 while(it != myMap.end()){
4
5     Node* myChild = it->second;
6
7     if < LOGIC STATEMENT >{
8         Node* temp = < myChild's child >;
9
10        myMap["NewEdge"] = temp;
11
12        delete myChild;
13
14        it = myMap.erase(it);
15
16    }
17 }
18
19
20
21
```

returns iterator at next pos
(:++)



Assignment 6: a_stree

Learning Objective:



9

99

Use an existing implementation of a suffix trie as a N-ary Tree

Implement exact pattern matching using a suffix trie

Construct a suffix tree from a suffix trie

*

Consider: The modified NaryTree code works for both suffix tries and trees. Can you write a search that works for both trees and tries?

↳ suffix tree search is harder!

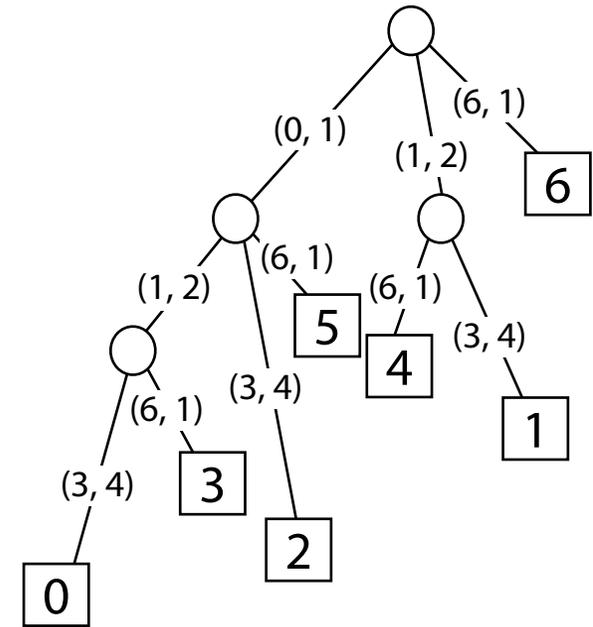
Suffix tree: building

Method 1: build suffix trie, coalesce non-branching paths, relabel edges

$O(m^2)$ time, $O(m^2)$ space

Method 2: build single-edge tree representing longest suffix, augment to include the 2nd-longest, augment to include 3rd-longest, etc (Gusfield 5.4)

$O(m^2)$ time, $O(m)$ space



On-Line Construction of Suffix Trees¹

E. Ukkonen²

Abstract. An on-line algorithm is presented for constructing the suffix tree for a given string in time linear in the length of the string. The new algorithm has the desirable property of processing the string symbol by symbol from left to right. It always has the suffix tree for the scanned part of the string ready. The method is developed as a linear-time version of a very simple algorithm for (quadratic size) suffix *tries*. Regardless of its quadratic worst case this latter algorithm can be a good practical method when the string is not too long. Another variation of this method is shown to give, in a natural way, the well-known algorithms for constructing suffix automata (DAWGs).

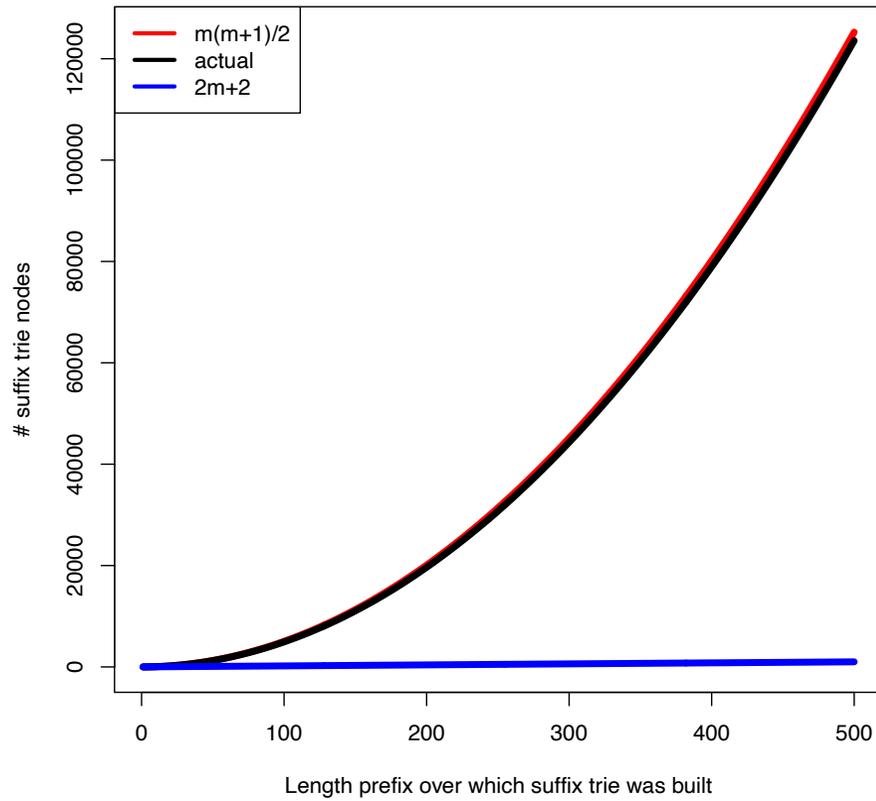
Key Words. Linear-time algorithm, Suffix tree, Suffix trie, Suffix automaton, DAWG.

Canonical algorithm for $O(m)$ time & space suffix tree construction

Won't cover it in class; see Gusfield Ch. 6 for details

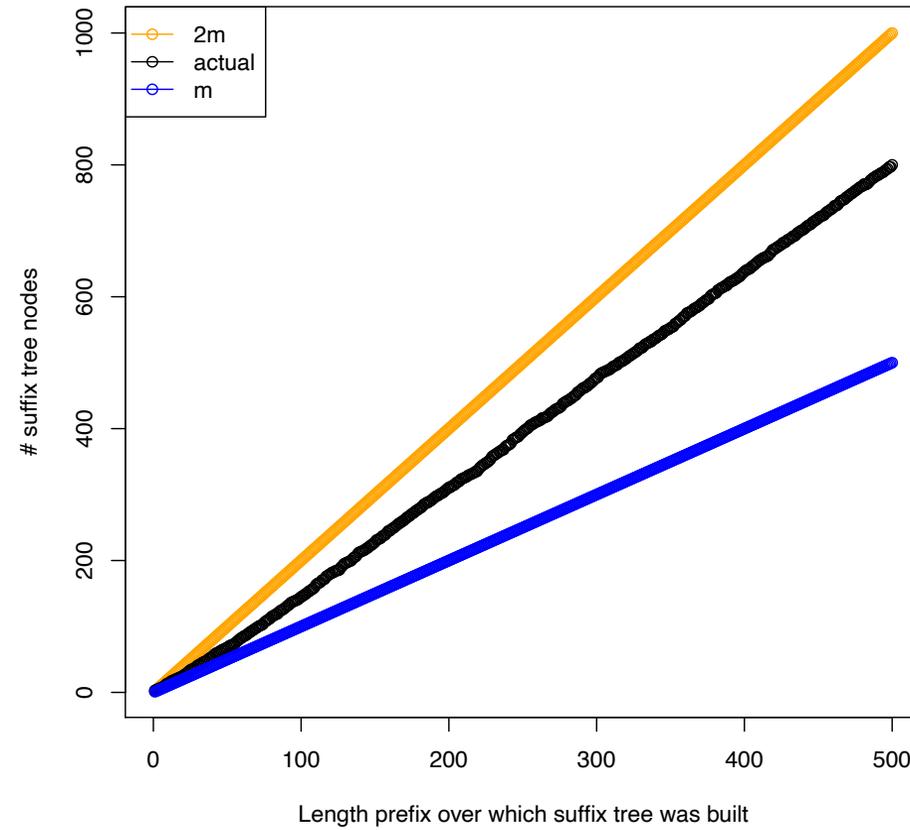
Suffix trie

>100K nodes



Suffix tree

<1K nodes



Suffix Tree



A rooted tree storing a collection of suffixes as (key, value) pairs

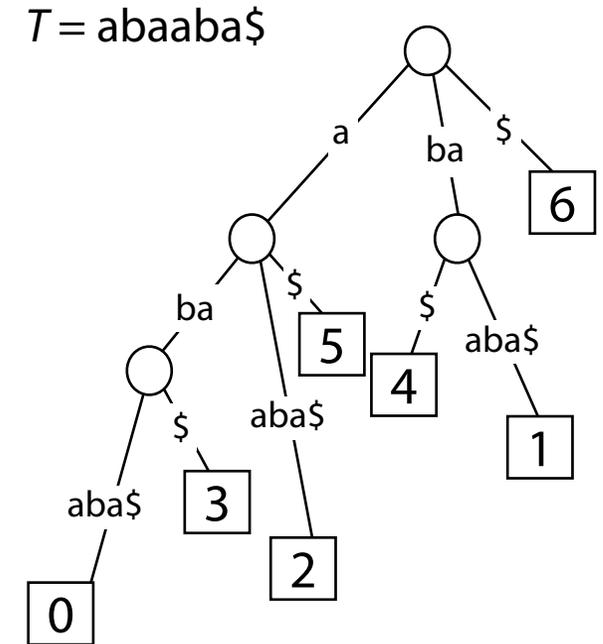
Each key is “spelled out” along some path starting at root

Each edge is labeled with **a string s**

For given node, at most one child edge **starts with character c** , for any $c \in \Sigma$

Each internal node contains >1 children

Each key's value is stored at a leaf



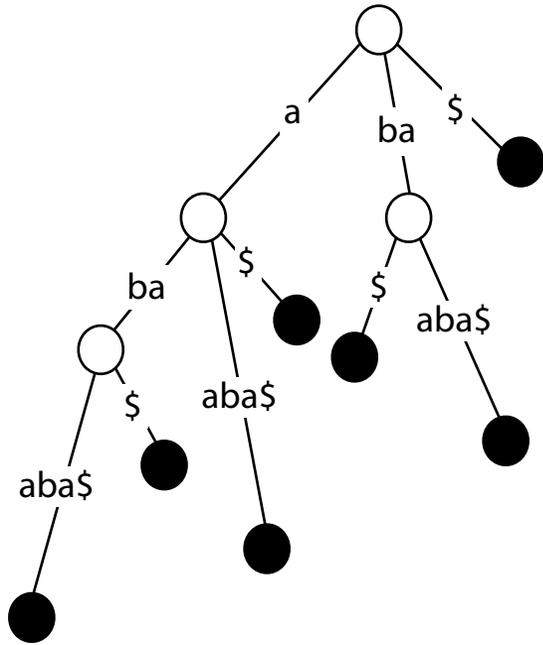


Bonus Slides

Suffix Tree: Size Redux

Sk: p proof!

$T = \text{abaaba}\$$ $|T| = m$



leaves? m

non-leaf nodes (upper-bound)? $m-1$

Let n be # of non-leaf nodes, e be # of edges total

All internal nodes (except root) 'cost' one edge:

$$m = e - (n - 1)$$

Each non-leaf has ≥ 2 child edges: ($e \geq 2n$):

$$m \geq 2n - (n - 1)$$

$$m \geq n + 1$$

$$n \leq m - 1$$

Total storage cost (nodes)?

$O(m)$

Suffix Tree: Size Redux

Problem: We still contain all **suffixes** of a text $|T| = m$

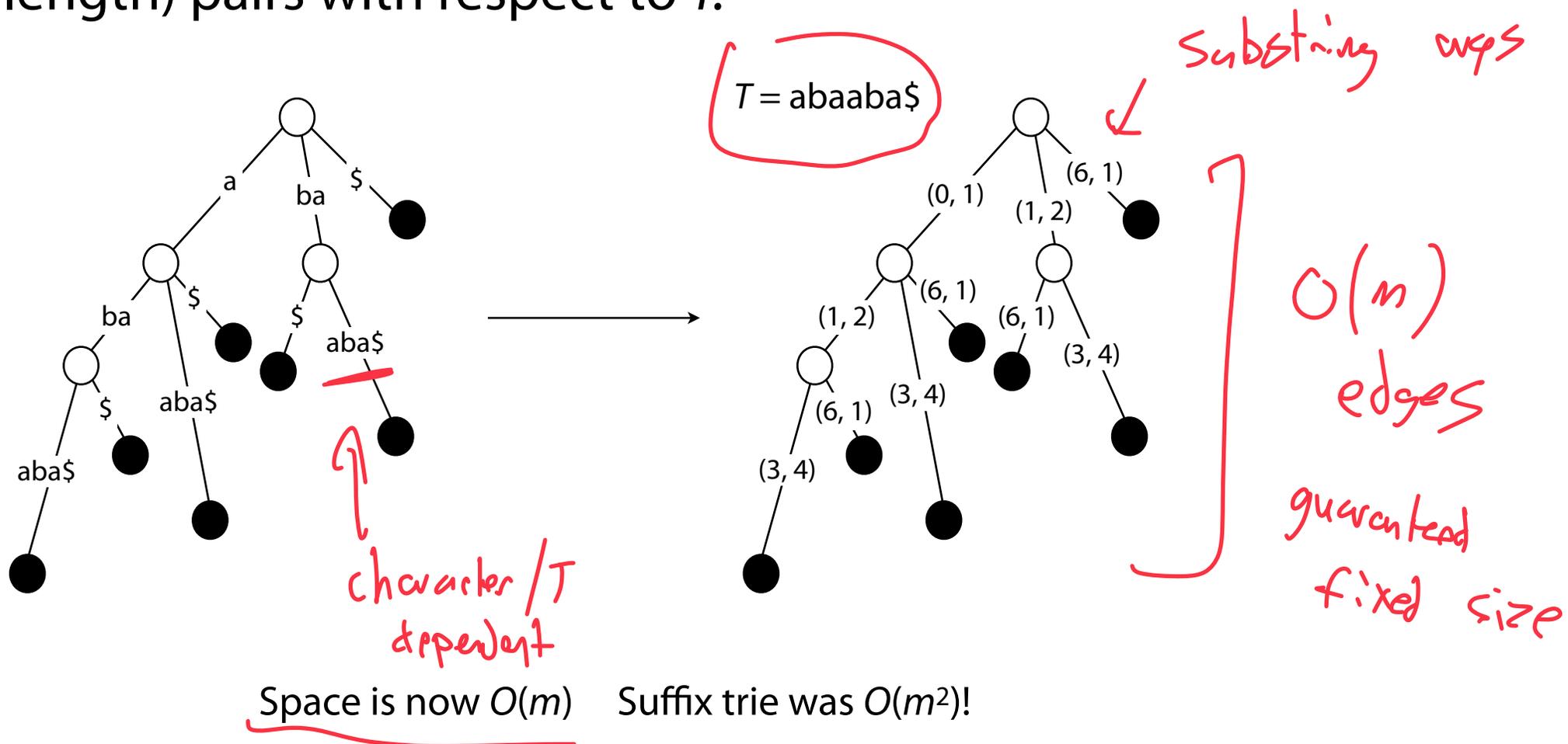
T:
GTTATAGCTGATCGCGGCGTAGCGG\$
GTTATAGCTGATCGCGGCGTAGCGG\$
TTATAGCTGATCGCGGCGTAGCGG\$
TATAGCTGATCGCGGCGTAGCGG\$
ATAGCTGATCGCGGCGTAGCGG\$
TAGCTGATCGCGGCGTAGCGG\$
AGCTGATCGCGGCGTAGCGG\$
GCTGATCGCGGCGTAGCGG\$
CTGATCGCGGCGTAGCGG\$
TGATCGCGGCGTAGCGG\$
GATCGCGGCGTAGCGG\$
ATCGCGGCGTAGCGG\$
TCGCGGCGTAGCGG\$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

$m(m+1)/2$ chars

$O(T^2)$
characters

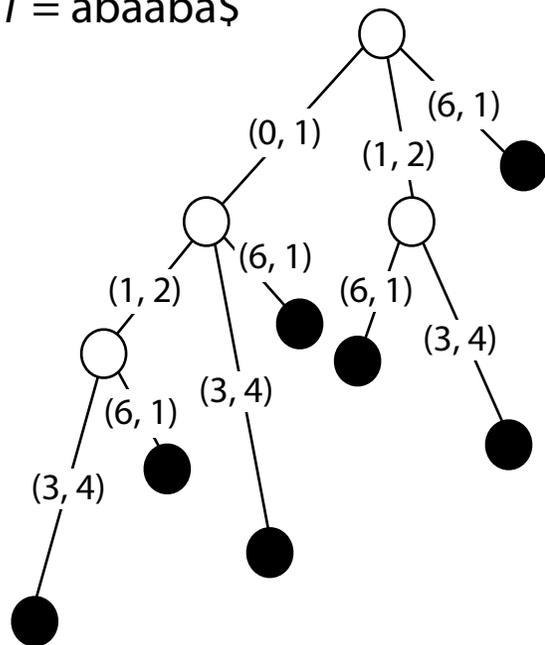
Suffix Tree: Size Redux

Store T itself in addition to the tree. Convert tree's edge labels to (index, length) pairs with respect to T .

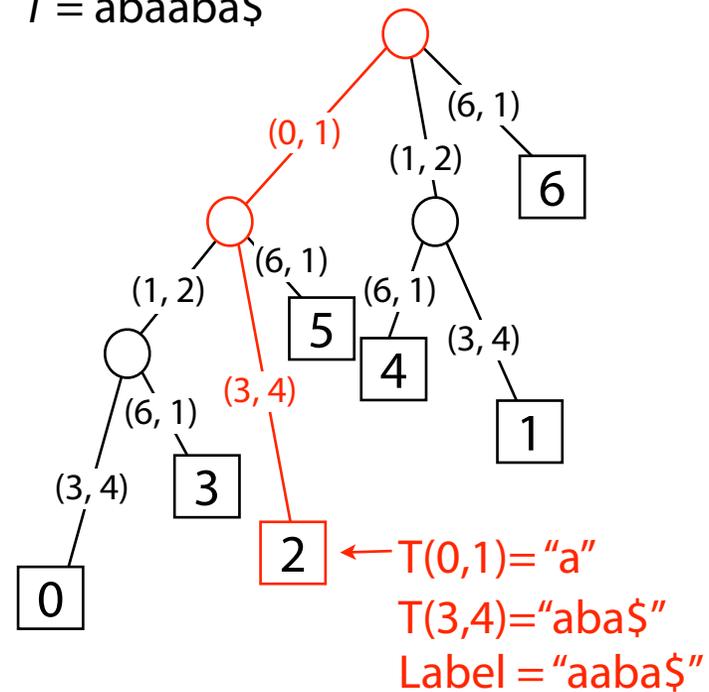


Suffix Tree: Size Redux

$T = \text{abaaba}\$$



$T = \text{abaaba}\$$



Suffix Tree: Size Redux

$T = \text{abaaba}\$$

