String Algorithms and Data Structures

Tries

CS 199-225
Brad Solomon

October 3, 2022

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Exact pattern matching w/ Boyer-Moore

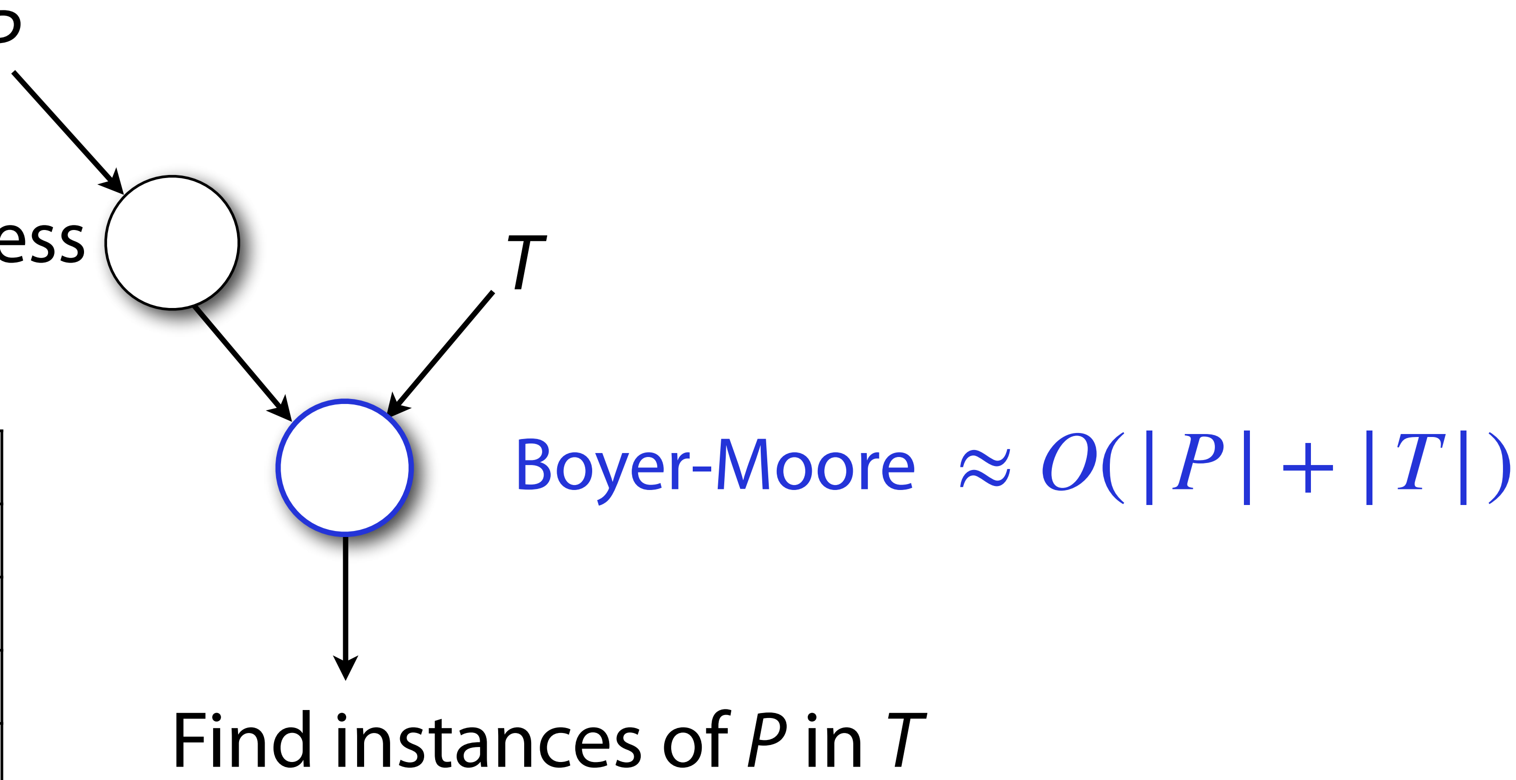
As seen in HW: sub-linear time *in practice*

P

		T	C	G	C
Σ	A	0	1	2	3
	C	0	-	0	-
	G	0	1	-	0
	T	-	0	1	2

Make lookup tables

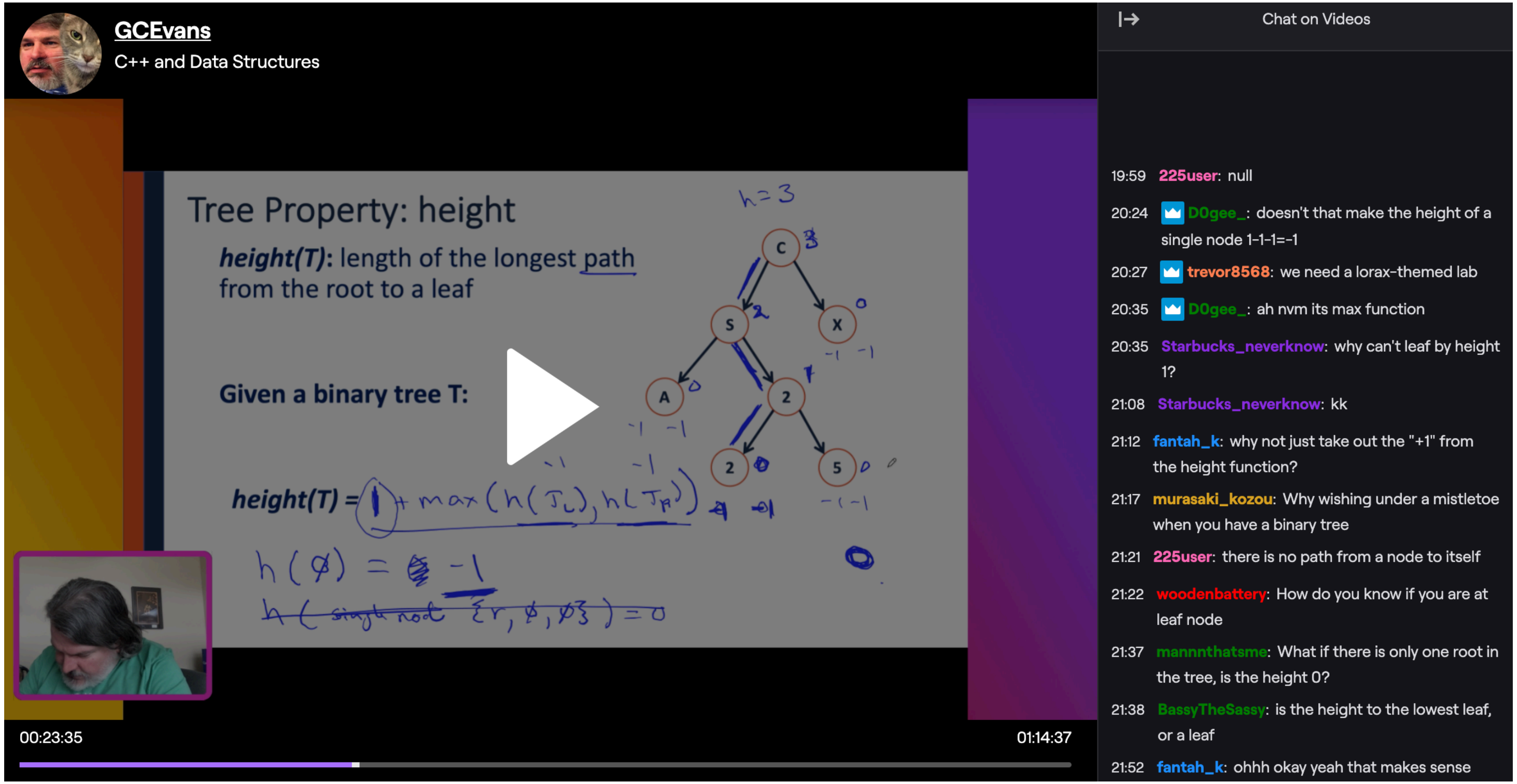
Preprocess



Boyer-Moore $\approx O(|P| + |T|)$

Find instances of *P* in *T*

Preprocessing: Live chat streams



19:59 225user: null

20:24 D0gee_: doesn't that make the height of a single node 1-1-1=-1

20:27 trevor8568: we need a lorax-themed lab

20:35 D0gee_: ah nvm its max function

20:35 Starbucks_neverknow: why can't leaf by height 1?

21:08 Starbucks_neverknow: kk

21:12 fantah_k: why not just take out the "+1" from the height function?

21:17 murasaki_kozou: Why wishing under a mistletoe when you have a binary tree

21:21 225user: there is no path from a node to itself

21:22 woodenbattery: How do you know if you are at leaf node

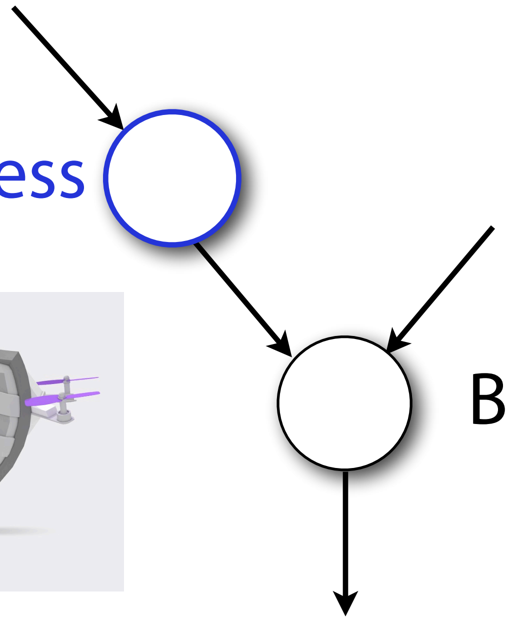
21:37 mannnthatsme: What if there is only one root in the tree, is the height 0?

21:38 BassyTheSassy: is the height to the lowest leaf, or a leaf

21:52 fantah_k: ohhh okay yeah that makes sense

Patterns: banned phrases **Text:** Chat messages

Preprocessing: Live chat streams



!@#$!%

Preprocess

Thanks for the help!

I don’t understand that…

You are a !@#$!% teacher

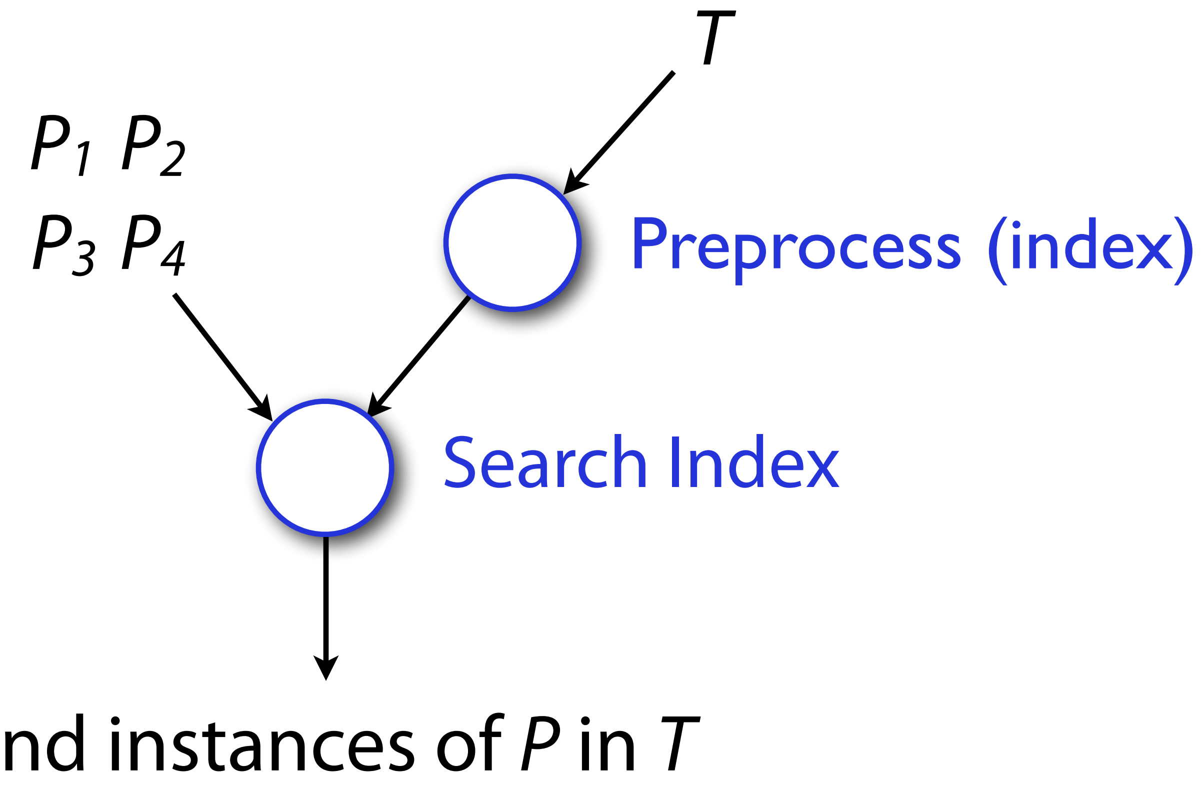
Boyer-Moore

Find instances of *P* in *T*

Amortize cost of preprocessing *P* over many *T*

Exact pattern matching *w/ indexing*

Conventionally *T* >> *P*:

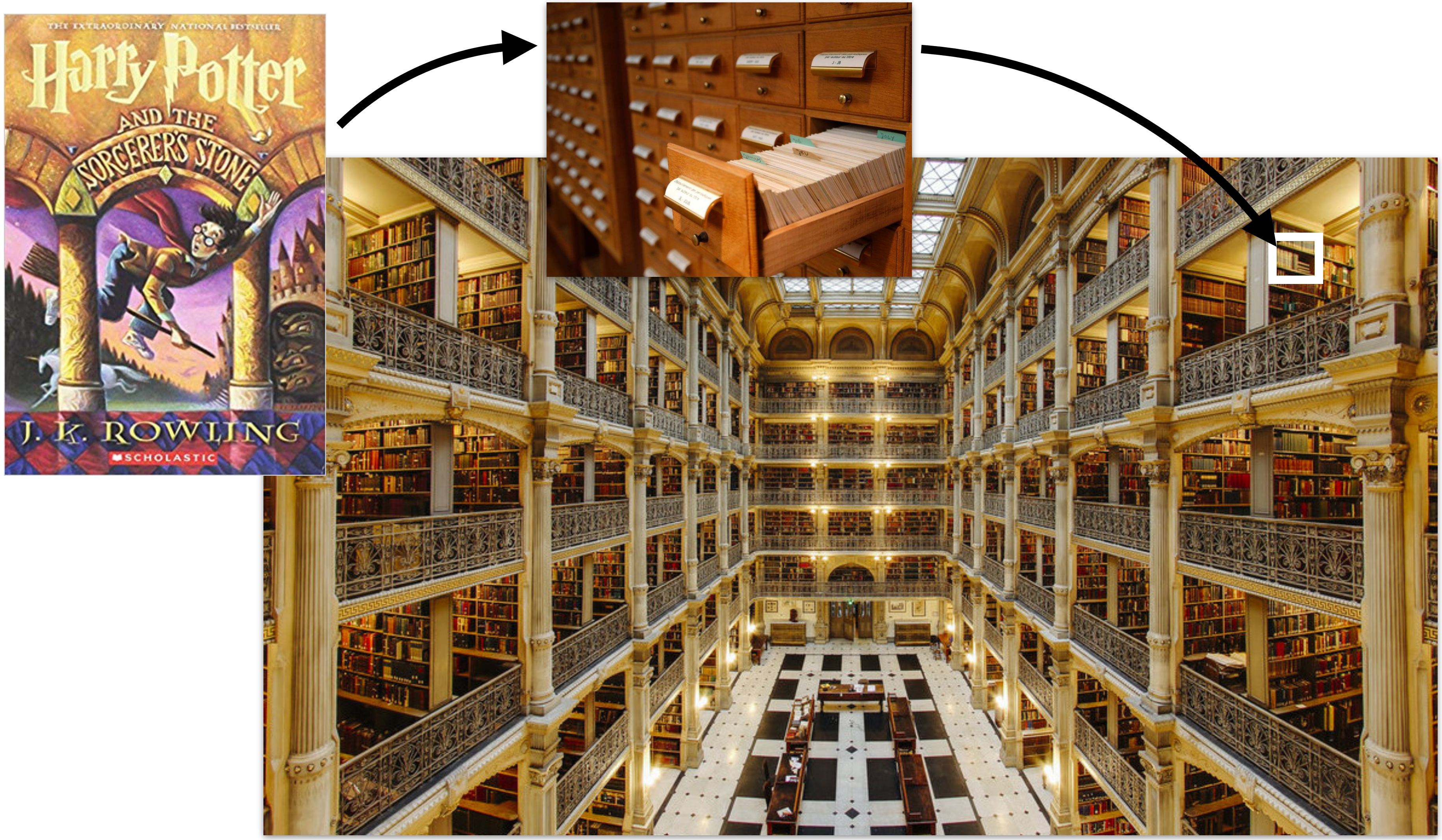


Amortize cost of preprocessing *T* over many *P*

Preprocessing: Libraries

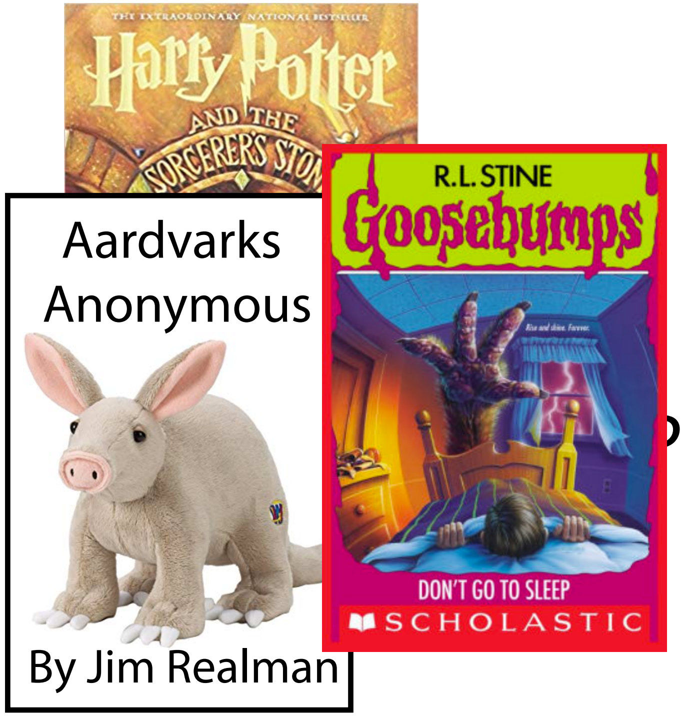
Patterns: Book of interest **Text:** All books in library

Preprocessing: Libraries

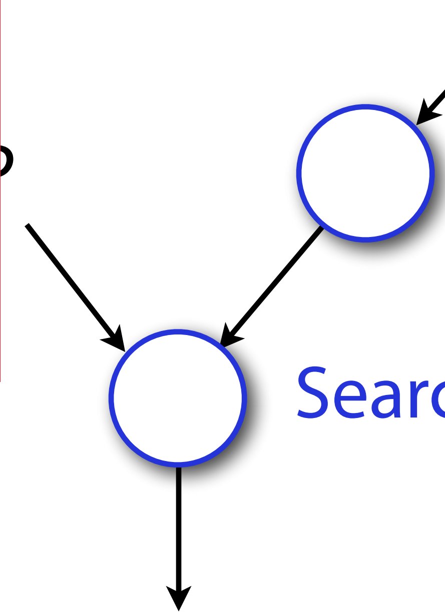


Preprocess the library by *indexing* all the books

Preprocessing: Libraries



List of all library books



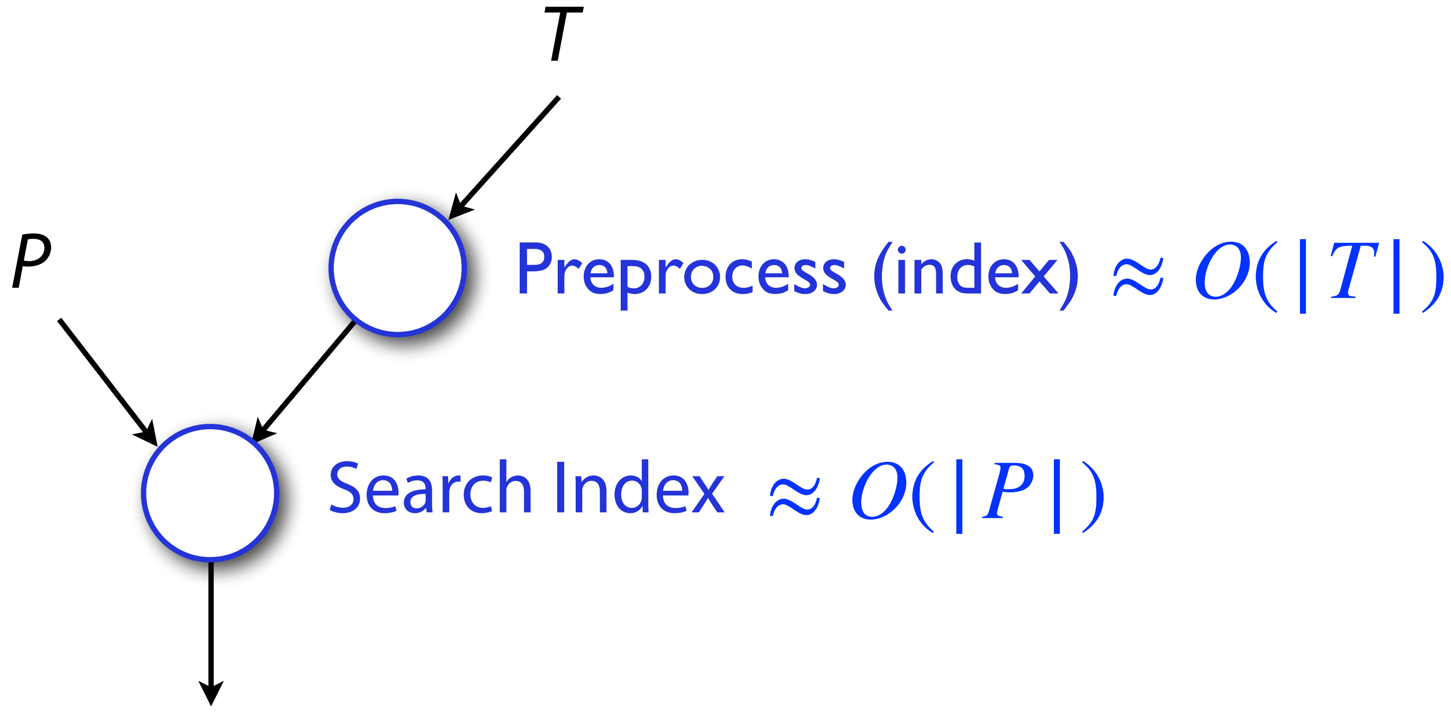
Preprocess (index)

Search Index

Find instances of *P* in *T*

Given full library, built an index once* that is re-used

Exact pattern matching *w/ indexing*


T
P
Preprocess (index) $\approx O(|T|)$
Search Index $\approx O(|P|)$

Find instances of *P* in *T*

What information from *T* do we need to search for *P*?

Preprocessing for exact pattern matching

T: **C G T G C**

P:

Search(*P*, *T*):

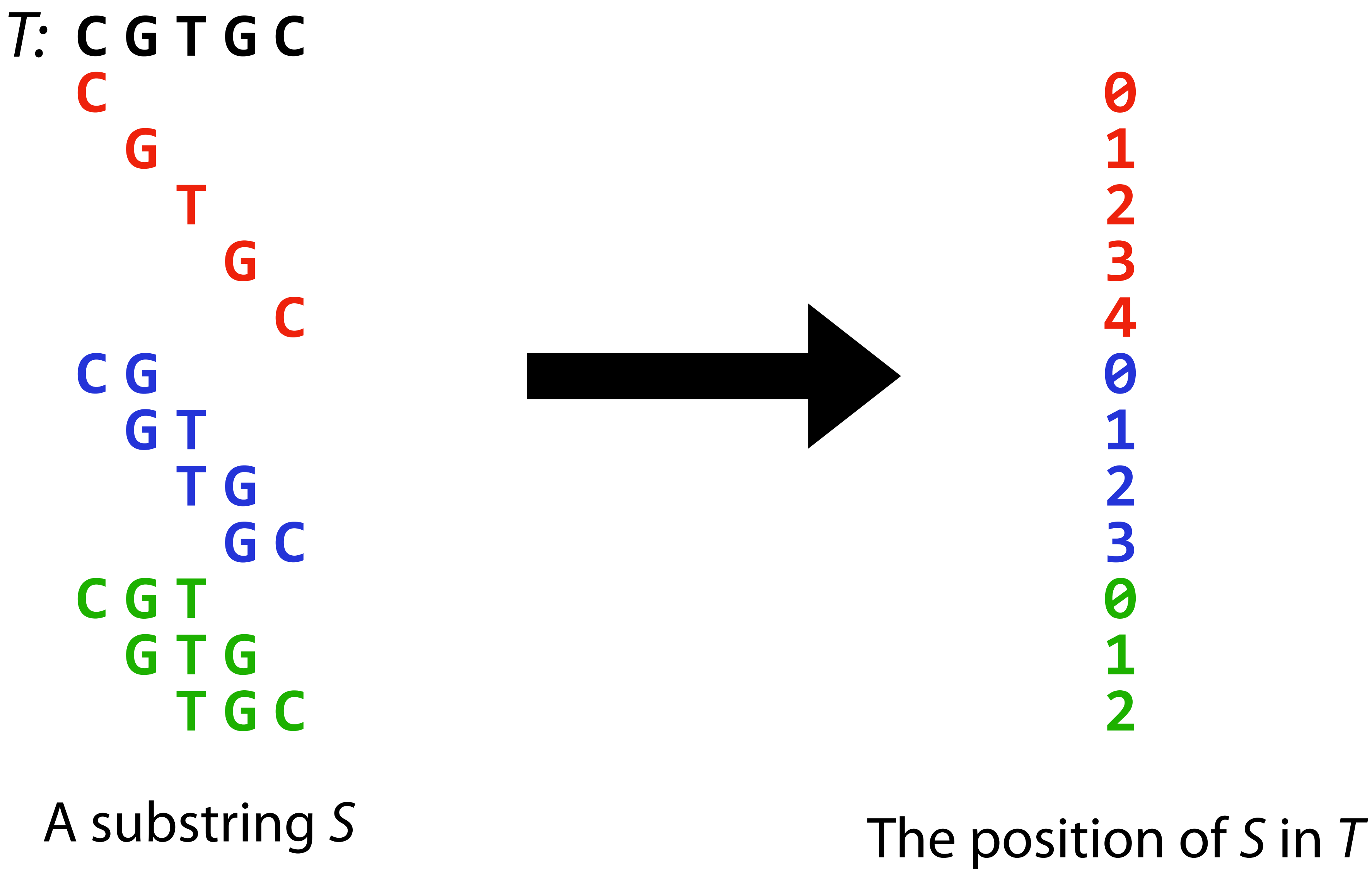
P:

Search(*P*, *T*):

P:

Search(*P*, *T*):

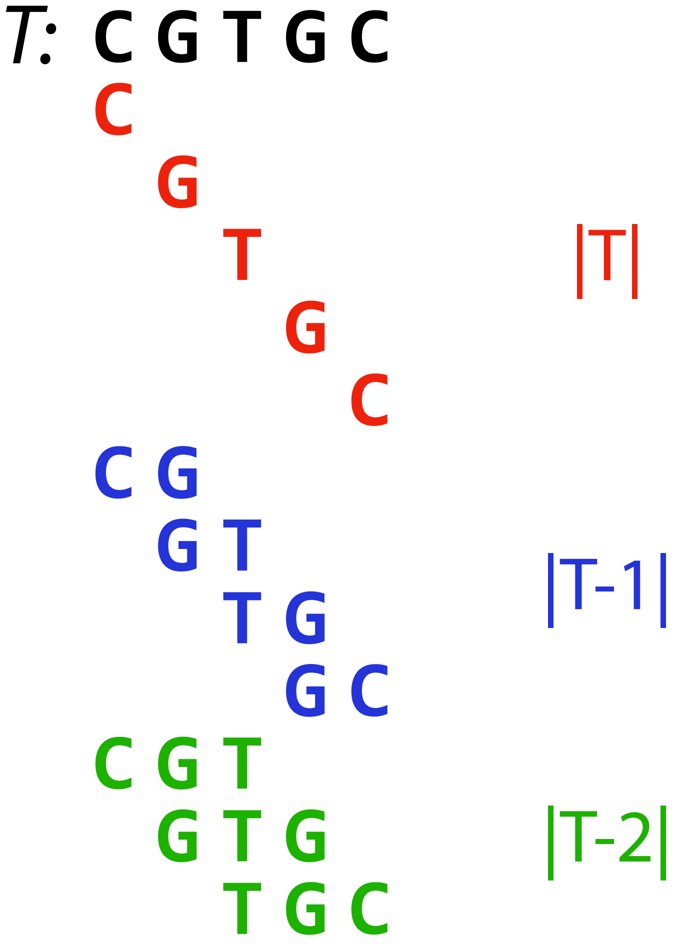
Preprocessing for exact pattern matching



T: C G T G C

C	0
G	1
T	2
G	3
C	4
C G	0
G T	1
T G	2
G C	3
C G T	0
G T G	1
T G C	2
A substring *S*	The position of *S* in *T*

Preprocessing for exact pattern matching



T: C G T G C

C

G

T

G

C

|T|

C G

G T

T G

G C

|T-1|

C G T

G T G

T G C

|T-2|

Key	Value
C	0
G	1
T	2
G	3
C	4
CG	0
GT	1
TG	2
…	…

?

Preprocessing for exact pattern matching

T: **C G T G C**

C
G
T |T|
G
C

C G
G T |T-1|
T G
G C

C G T
G T G |T-2|
T G C

Key	Value
C	0
G	1
T	2
G	3
C	4
CG	0
GT	1
TG	2
…	…

$$\frac{|T|(|T|+1)}{2}$$

Preprocessing for exact pattern matching

Key	Value
C	0
G	1
T	2
G	3
C	4
CG	0
GT	1
TG	2
…	…

Because our keys are strings, this is sometimes possible!

$$\frac{|T|(|T|+1)}{2}$$

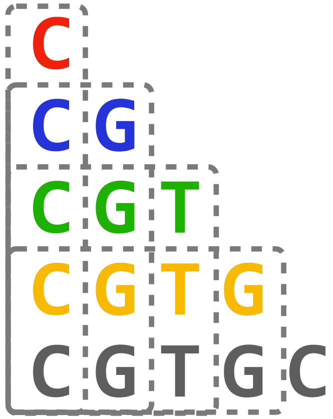
We want to search in $O(|P|)$ without $O(|T|^2)$ space!

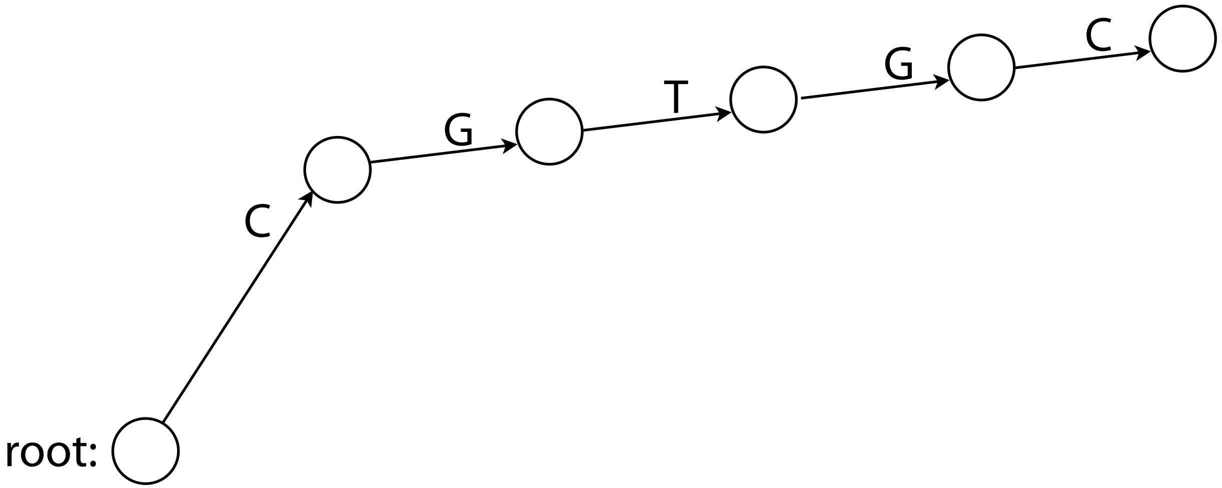
Preprocessing for exact pattern matching

Strings consist of individual characters!

… and these characters can overlap:

T: **C G T G C**





Preprocessing for exact pattern matching

Strings consist of individual characters!

… and these characters can overlap:

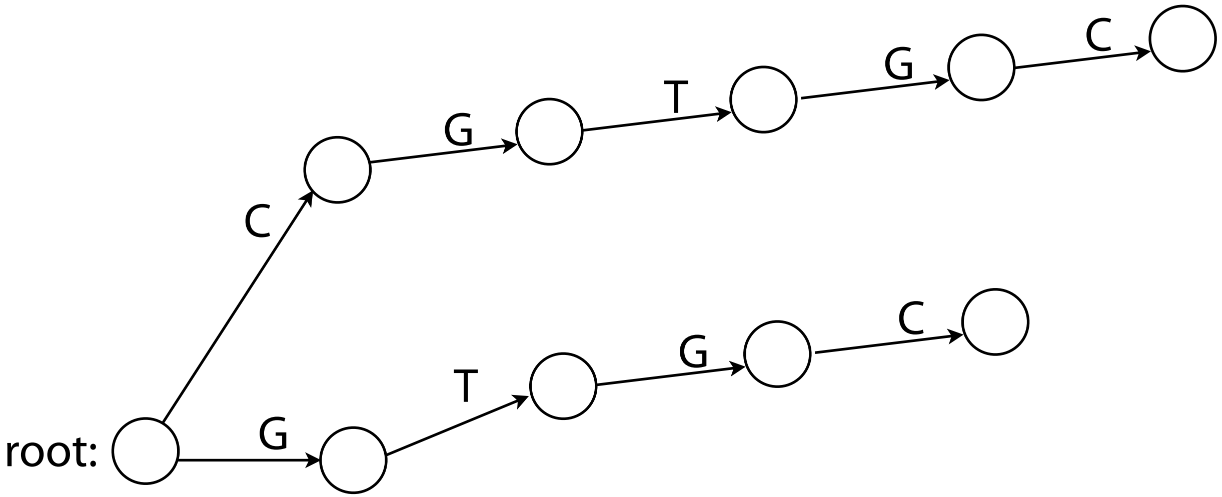
T: **C G T G C**

G

G T

G T G

G T G C



Preprocessing for exact pattern matching

Strings consist of individual characters!

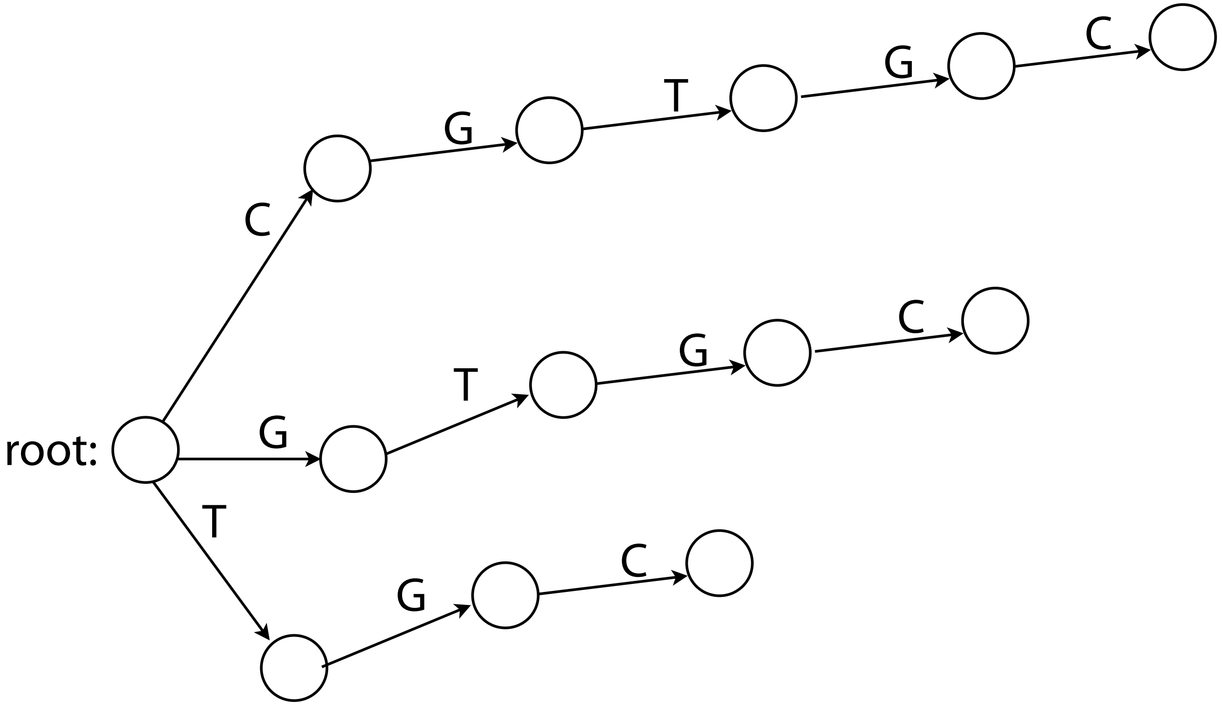
… and these characters can overlap:

T: **C G T G C**

T

T G

T G C



Preprocessing for exact pattern matching

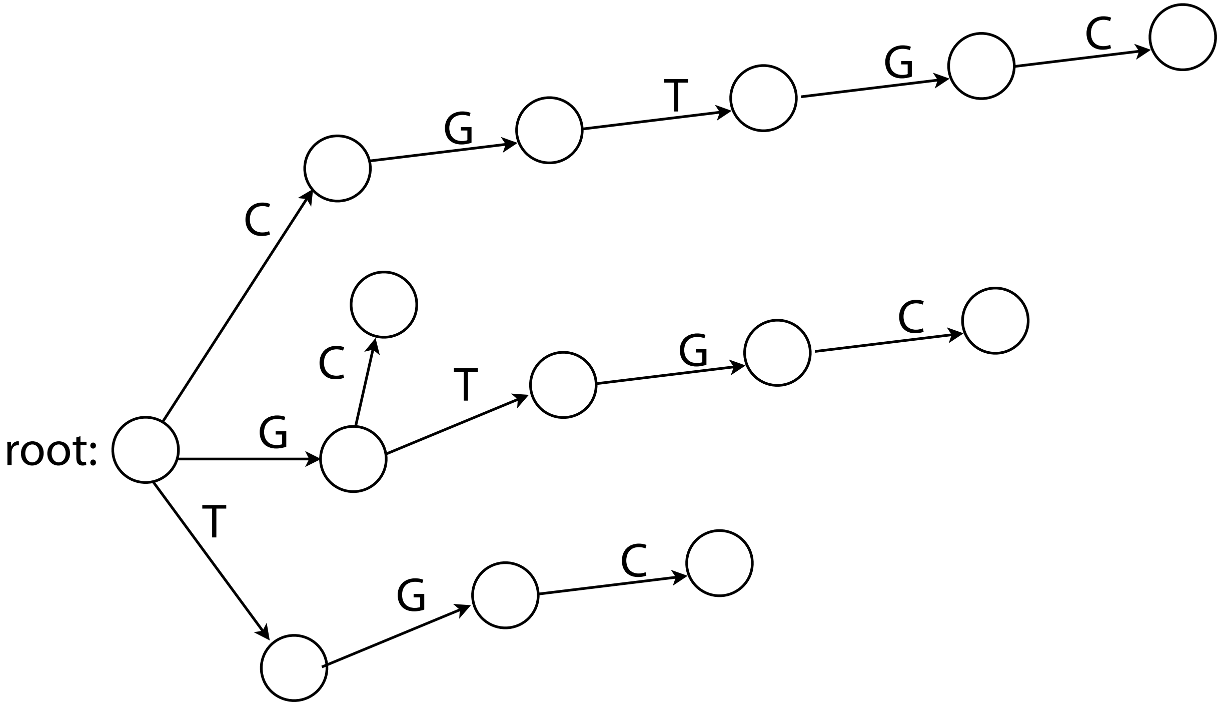
Strings consist of individual characters!

… and these characters can overlap:

T: **C G T G C**

G

G C



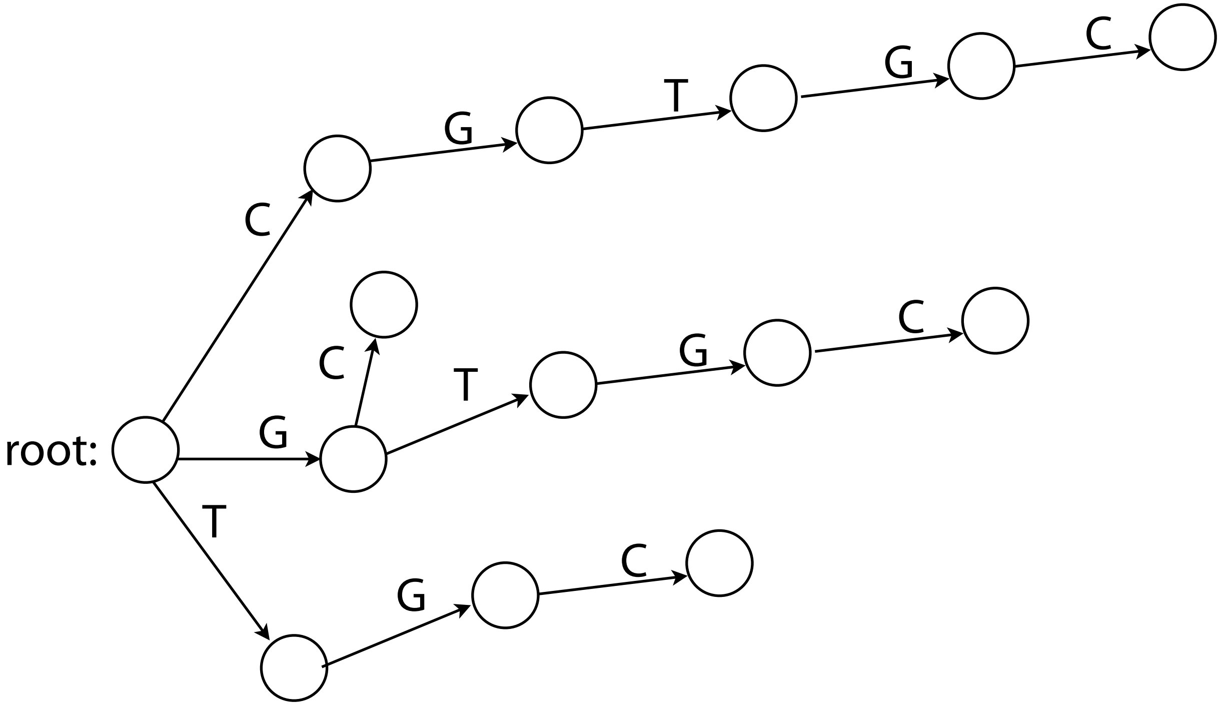
Preprocessing for exact pattern matching

Strings consist of individual characters!

… and these characters can overlap:

T: **C G T G C**

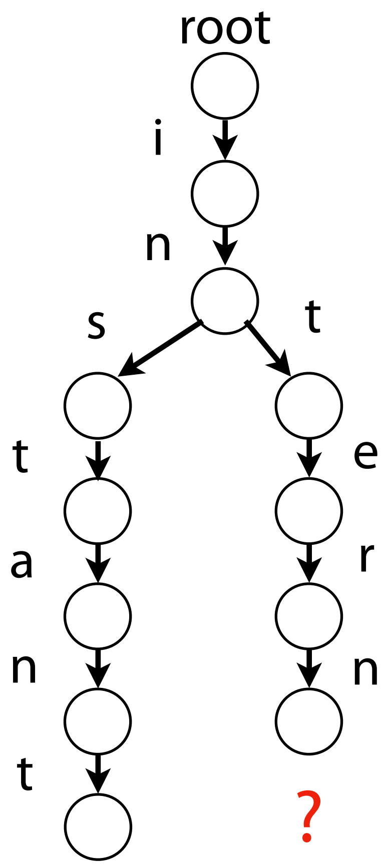
C



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 $c \in \Sigma$

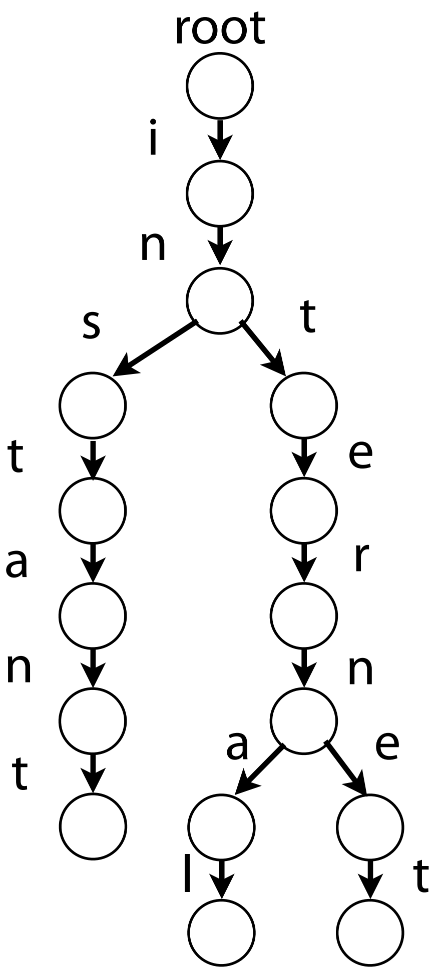
For given node, at most one child edge has label c, for any $c \in \Sigma$

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

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* ∈ *Σ*

For given node, at most one child edge has label *c*, for any *c* ∈ *Σ*

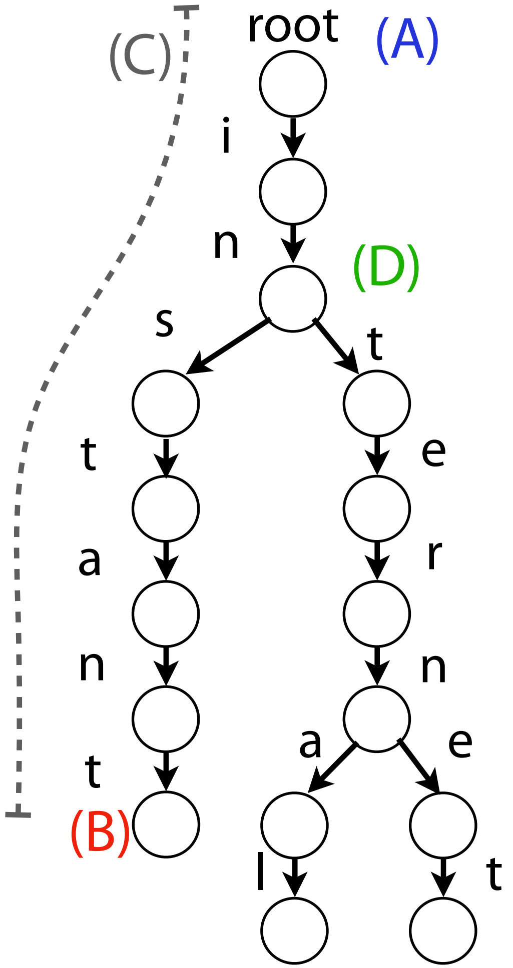
Each key is “spelled out” along some path starting at root

String indexing with Tries

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

Keys:	Values:
`instant`	1
`internal`	2
`internet`	3

Where should I store the value 1?

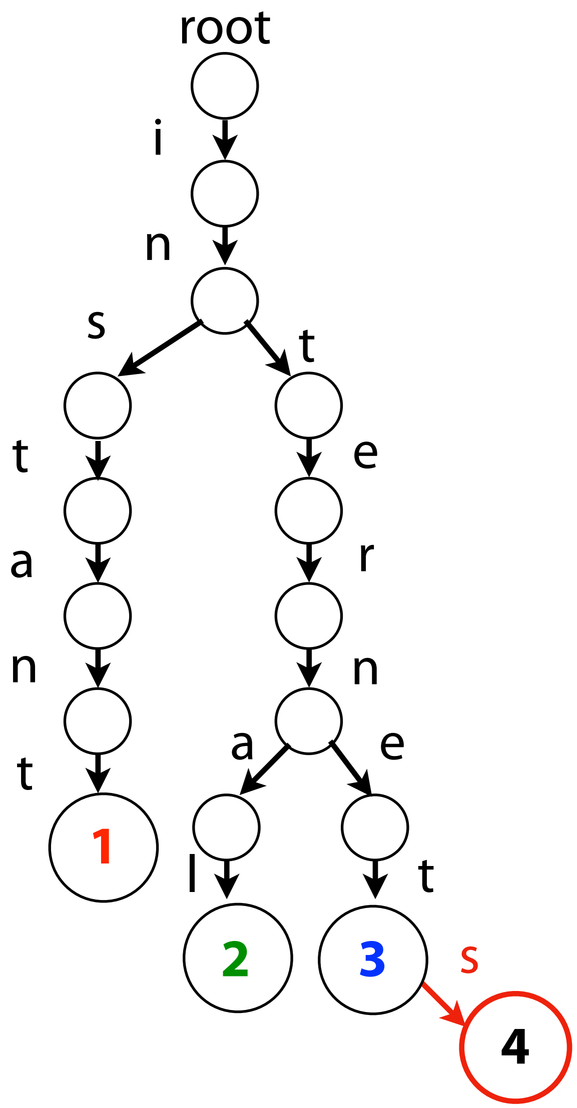


String indexing with Tries

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

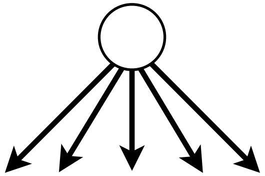
Keys:	Values:
instant	1
internal	2
internet	3
internets	4

Each key's value is stored at the last node in the path



The Node Implementation

Each node in my trie has $\leq |\Sigma|$ edges!

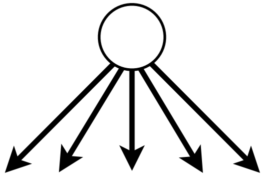


Each edge is a (potentially NULL) pointer.

How can we encode this?

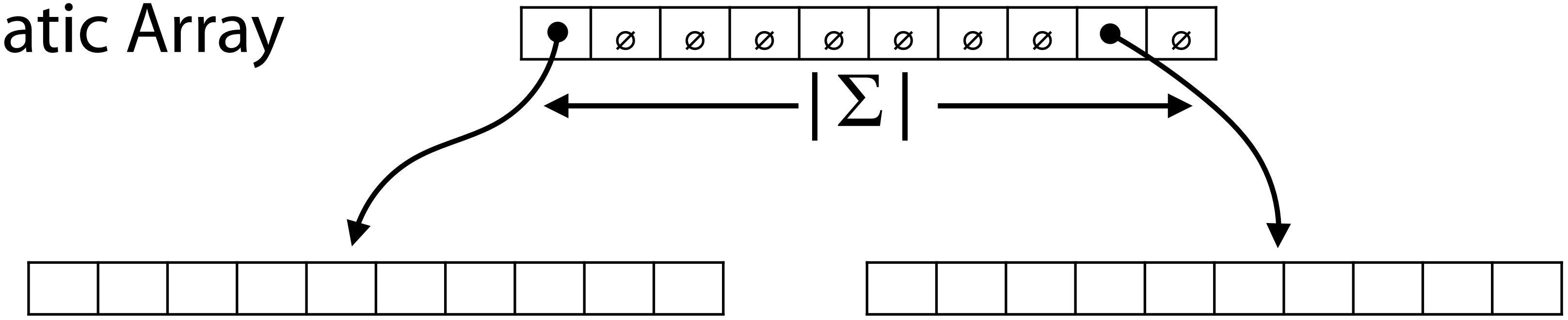
The Node Implementation

Each node in my trie has $\leq |\Sigma|$ edges!

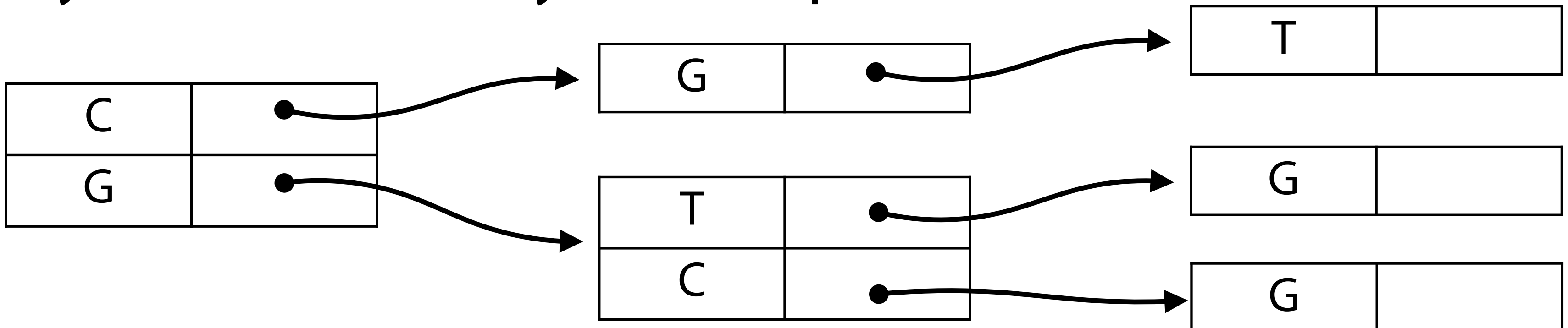


Each edge is a (potentially NULL) pointer.

1) Static Array



2) Dynamically-sized Dictionary (std::map)



Trie Node Implementation

NaryTree.h

```
class NaryTree
{
  public:
    struct Node {
      std::vector<int> index;
      std::map<char, Node*> children;

      Node(std::string s, int i)
      {
        if(s.length() > 0 ){
          children[s[0]] = new Node(s.substr(1), i);
        } else {
          index.push_back(i);
        }
      }
    };
  protected:
    Node* root;
…
}
```

Trie Node Implementation

NaryTree.h

```
class NaryTree
{
  public:
    struct Node {
      std::vector<int> index;
      std::map<char, Node*> children;

      Node(std::string s, int i)
      {
        if(s.length() > 0 ){
          children[s[0]] = new Node(s.substr(1), i);
        } else {
          index.push_back(i);
        }
      }
    };
  protected:
    Node* root;
…
}
```

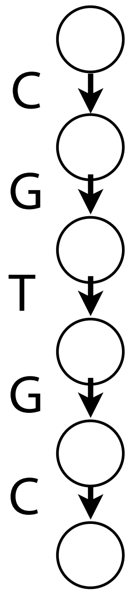
CGTGC

GTGC

TGC

GC

C



i here ->

What if we have more than one string?

Trie Node Implementation

main.cpp

```
NaryTree myT;
myTree.print();

myTree.insert("AB",0);
myTree.print();

myTree.insert("ABA",1);
myTree.print();

myTree.insert(“ABB",2);
myTree.print();

myTree.insert("BAB",3);
myTree.print();

myTree.insert(“BBB",4);
myTree.print();
```

```
x
```

Trie Node Implementation

main.cpp

```
NaryTree myT;
myTree.print();

myTree.insert("AB",0);
myTree.print();

myTree.insert("ABA",1);
myTree.print();

myTree.insert(“ABB",2);
myTree.print();

myTree.insert("BAB",3);
myTree.print();

myTree.insert(“BBB",4);
myTree.print();
```

```
o
+--A--o      0
      +--B--o
```

Trie Node Implementation

main.cpp

```cpp
NaryTree myT;
myTree.print();

myTree.insert("AB",0);
myTree.print();

myTree.insert("ABA",1);
myTree.print();

myTree.insert(“ABB",2);
myTree.print();

myTree.insert("BAB",3);
myTree.print();

myTree.insert(“BBB",4);
myTree.print();
```

Former leaf node, still holds value

```
o
+--A--o    0
     +--B--o    1
          +--A--o
```

```cpp
struct Node {
        std::vector<int> index;
        std::map<char, Node*> children;
}
```

Trie Node Implementation

main.cpp

```
NaryTree myT;
myTree.print();

myTree.insert("AB",0);
myTree.print();

myTree.insert("ABA",1);
myTree.print();

myTree.insert(“ABB",2);
myTree.print();

myTree.insert("BAB",3);
myTree.print();

myTree.insert(“BBB",4);
myTree.print();
```

```
o
+--A--o      0
      +--B--o      1
            +--A--o
            +--B--o  2
```

Trie Node Implementation

main.cpp

```
NaryTree myT;
myTree.print();

myTree.insert("AB",0);
myTree.print();

myTree.insert("ABA",1);
myTree.print();

myTree.insert(“ABB",2);
myTree.print();

myTree.insert("BAB",3);
myTree.print();

myTree.insert(“BBB",4);
myTree.print();
```

```
o
+--A--o      0
|     +--B--o     1
|           +--A--o
|           +--B--o 2
+--B--o
      +--A--o
            +--B--o 3
```

Trie Node Implementation

main.cpp

```
NaryTree myT;
myTree.print();

myTree.insert("AB",0);
myTree.print();

myTree.insert("ABA",1);
myTree.print();

myTree.insert(“ABB",2);
myTree.print();

myTree.insert("BAB",3);
myTree.print();

myTree.insert(“BBB",4);
myTree.print();
```

```
o
+--A--o     0
|     +--B--o     1
|           +--A--o
|           +--B--o 2
+--B--o
      +--A--o
      |     +--B--o 3
      +--B--o
            +--B--o 4
```

Trie Node Implementation

NaryTree.h

```
void NaryTree::insert(const std::string& s, int i)
{
    insert(root, s, int i);
}

void NaryTree::insert(Node*& node, const std::string & s, int i)
{
    // If we're at a NULL pointer, we make a new Node
    if (node == NULL) {
        node = new Node(s, i);
    } else {
        if(s.length() > 0 ){
            if(node->children.count(s[0]) > 0){
               insert(node->children[s[0]],s.substr(1), i);
            }else{
                node->children[s[0]] = new Node(s.substr(1), i);
            }
        } else{
            node->index.push_back(i);
        }

    }
}
```

Assignment 5: a_narytree

Learning Objective:

Store all substrings in a trie using NaryTree implementation

Implement exact pattern matching using this trie

Consider: How many insertions are we doing for each string?
Is there a better or faster way to do this?

Trie Node Implementation

main.cpp

```
NaryTree myT;

myTree.insert("AB",0);

myTree.insert(“AB",2);

myTree.print();
```

```
o
+--A--o     ??
      +--B--o
```

Trie Node Implementation

main.cpp

```
NaryTree myT;

myTree.insert("AB",0);

myTree.insert(“AB",2);

myTree.print();
```

```
o
+--A--o    0,2
      +--B--o
```

```
struct Node {
    std::vector<int> index;
    std::map<char, Node*> children;
}
```

```
if(s.length() > 0 ){
    if(node->children.count(s[0]) > 0){
        insert(node->children[s[0]],s.substr(1), i);
    }else{
        node->children[s[0]] = new Node(s.substr(1), i);
    }
} else{
    node->index.push_back(i);
}
```

Searching a Trie

Given *P*, search the trie for keys and return values

Pattern: `i n f e r`

`i n f e r`

`i n f e r`

`i n f e r`

Lets break that down using *recursion*:

Starting at root:

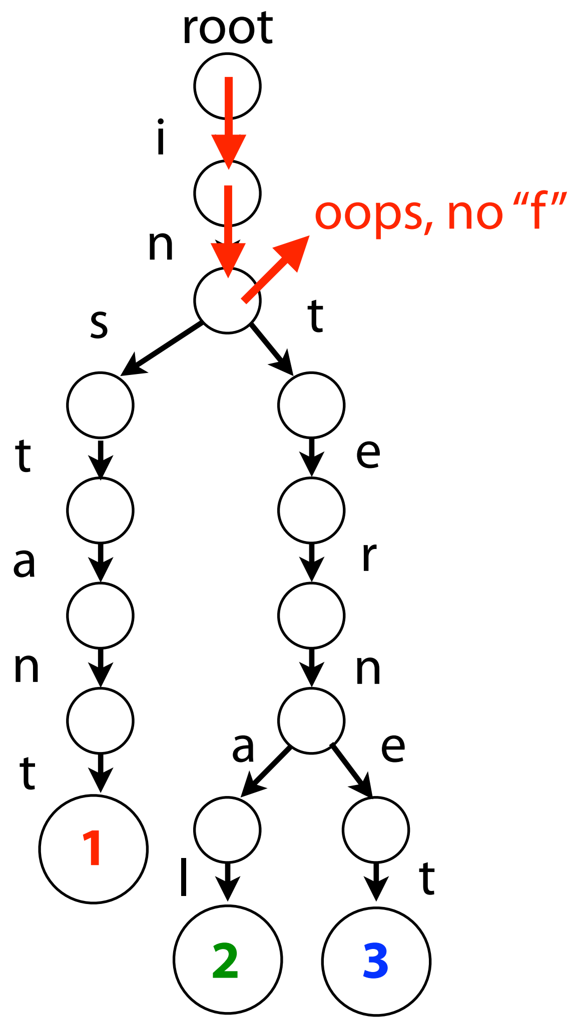
(1) Try to match front character

(2) If match, move to appropriate child

(2.5) Set pattern equal to remainder

(2.5) Go back to (1)

(3) If mismatch, *P* is not a key!



Searching a Trie

Given *P*, search the trie for keys and return values

Pattern: `i n t e r e s t i n g`

`i n t e r e s t i n g`

`i n t e r e s t i n g`

Lets break that down using *recursion*:

Starting at root:

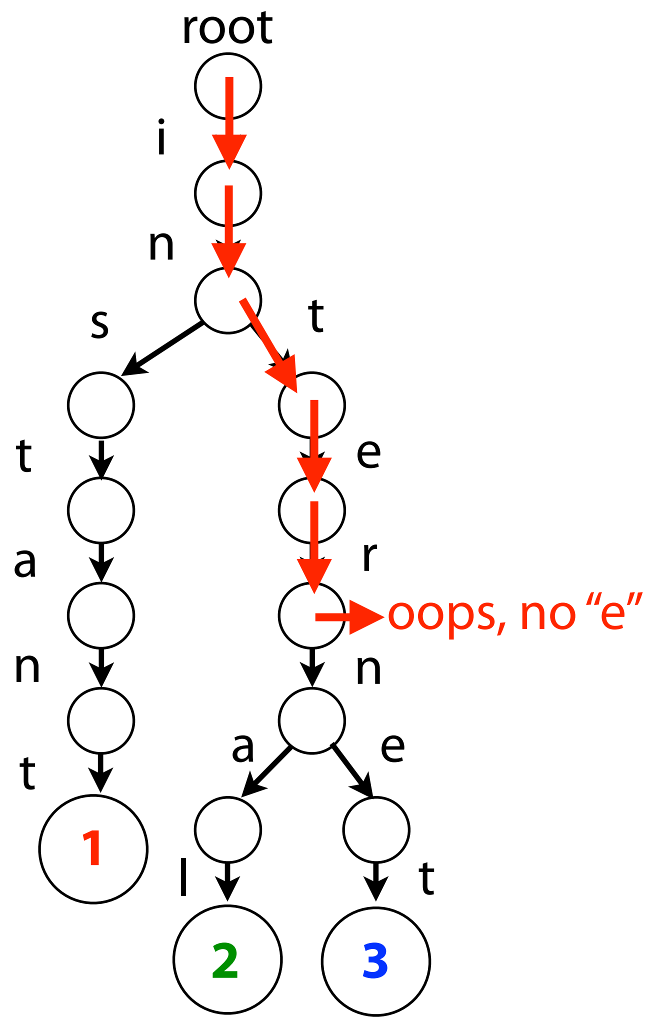
(1) Try to match front character

(2) If match, move to appropriate child

(2.5) Set pattern equal to remainder

(2.5) Go back to (1)

(3) If mismatch, *P* is not a key!



oops, no "e"

Searching a Trie

Given *P*, search the trie for keys and return values

Pattern: `i n s t a`

`i n s t a`

Lets break that down using *recursion*:

Starting at root:

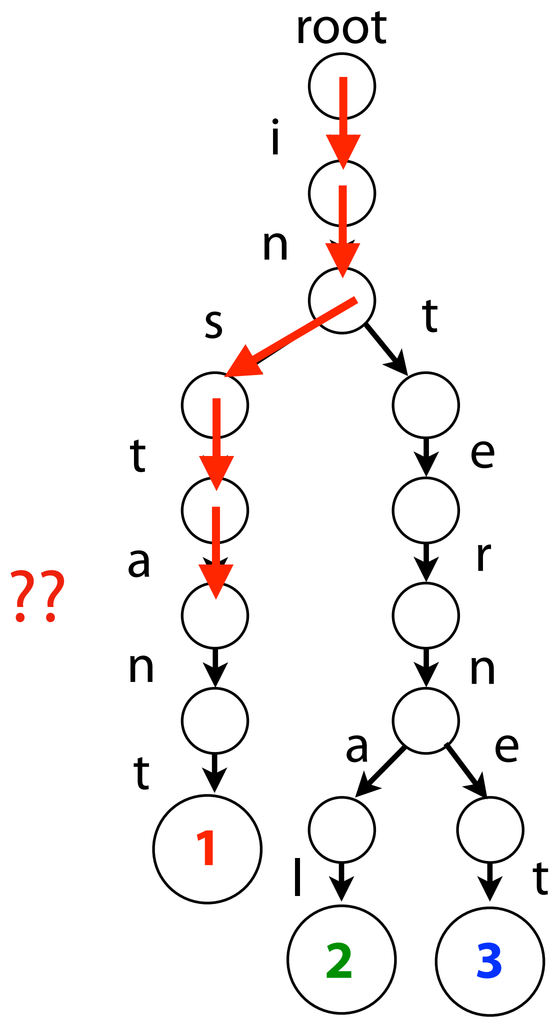
(1) Try to match front character

(2) If match, move to appropriate child

(2.5) Set pattern equal to remainder

(2.5) Go back to (1)

(3) If mismatch, *P* is not a key!



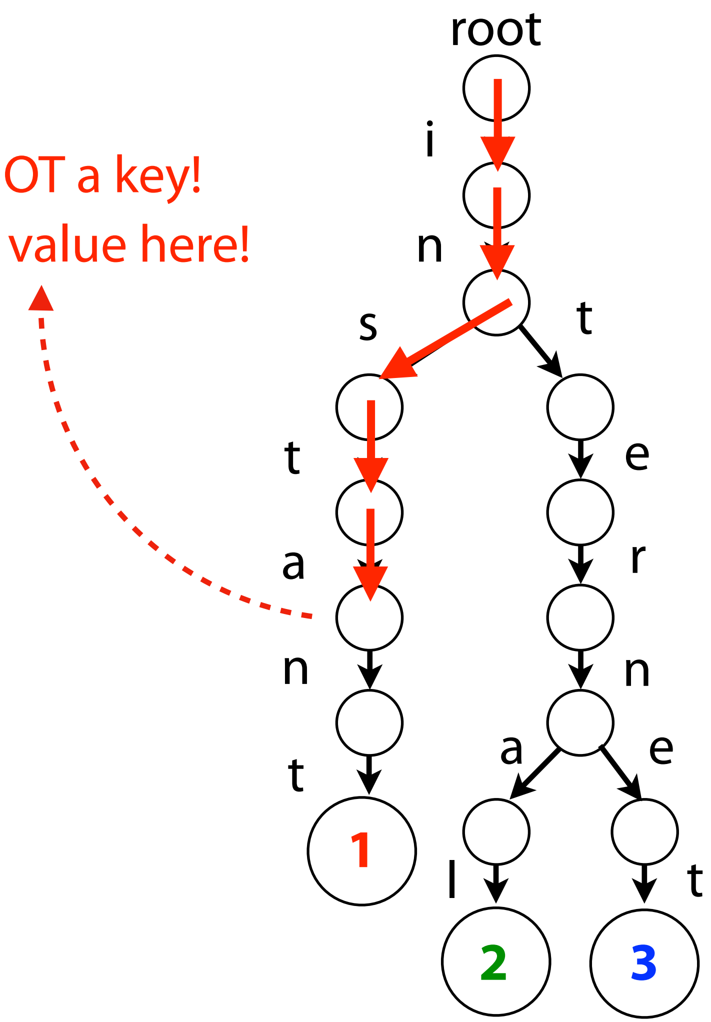
Searching a Trie

Given *P*, search the trie for keys and return values

Pattern: `i n s t a`

`i n s t a`

“Insta” is NOT a key!
There’s no value here!



Lets break that down using *recursion*:

Starting at root:

(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 (1)

(3) If mismatch, *P* is not a key!

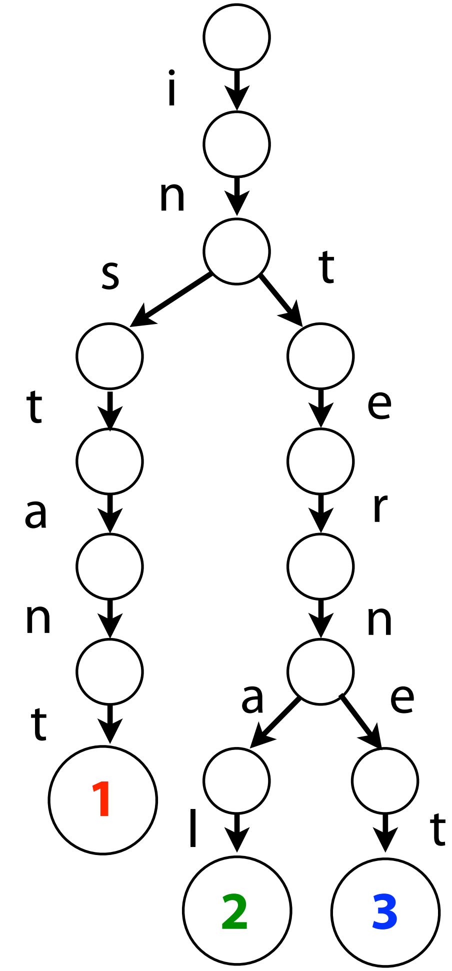
String indexing with Tries

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

Keys:	Values:
`instant`	1
`internal`	2
`internet`	3

The trie is structured such that:

- 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
- Each key's value is stored at the last node in the path



Searching a Trie

Given *P*, search the trie for keys and return values

Pattern: insta

insta

Lets break that down using *recursion*:

Starting at root:

(0) If we have no 'front' char, check value

(0.5) If no value, *P* is not a key!

(0.5) If value, *P* is a key, return value(s).

(1) Try to match front character

(2) If match, move to appropriate child

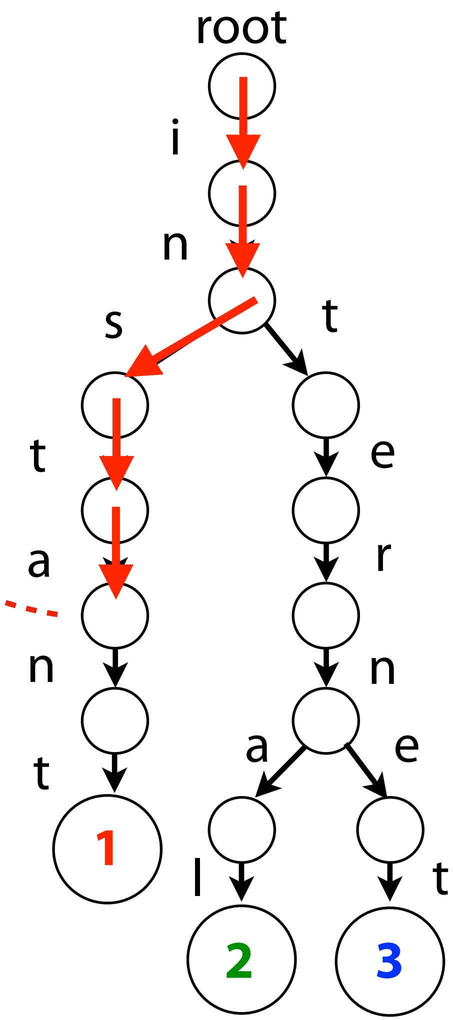
(2.5) Set pattern equal to remainder

(2.5) Go back to (1)

(3) If mismatch, *P* is not a key!

"Insta" is NOT a key!

There's no value here!



Searching a Trie

Given *P*, search the trie for keys and return values

Pattern: `i n s t a n t`

`i n s t a n t`

Lets break that down using *recursion*:

Starting at root:

(0) If we have no 'front' char, check value

(0.5) If no value, *P* is not a key.

(0.5) If value, *P* is a key, return value(s).

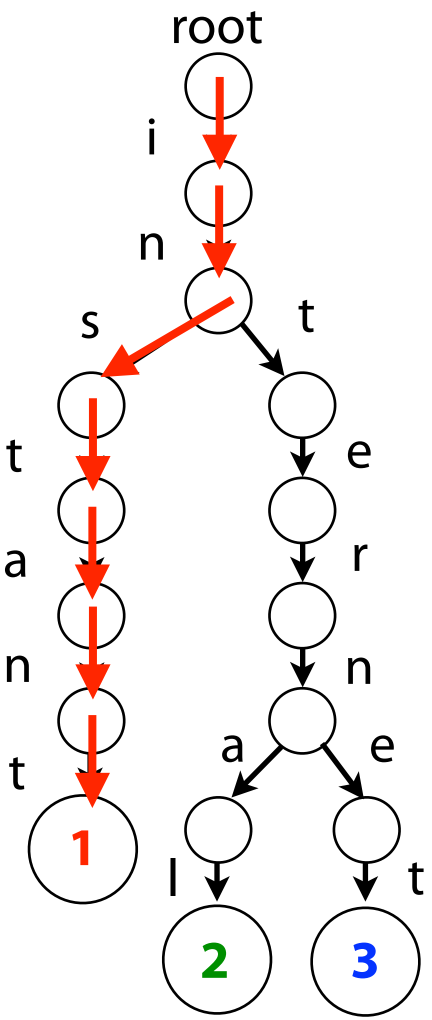
(1) Try to match front character

(2) If match, move to appropriate child

(2.5) Set pattern equal to remainder

(2.5) Go back to (1)

(3) If mismatch, *P* is not a key!



Assignment 5: a_narytree

Learning Objective:

Store all substrings in a trie using NaryTree implementation

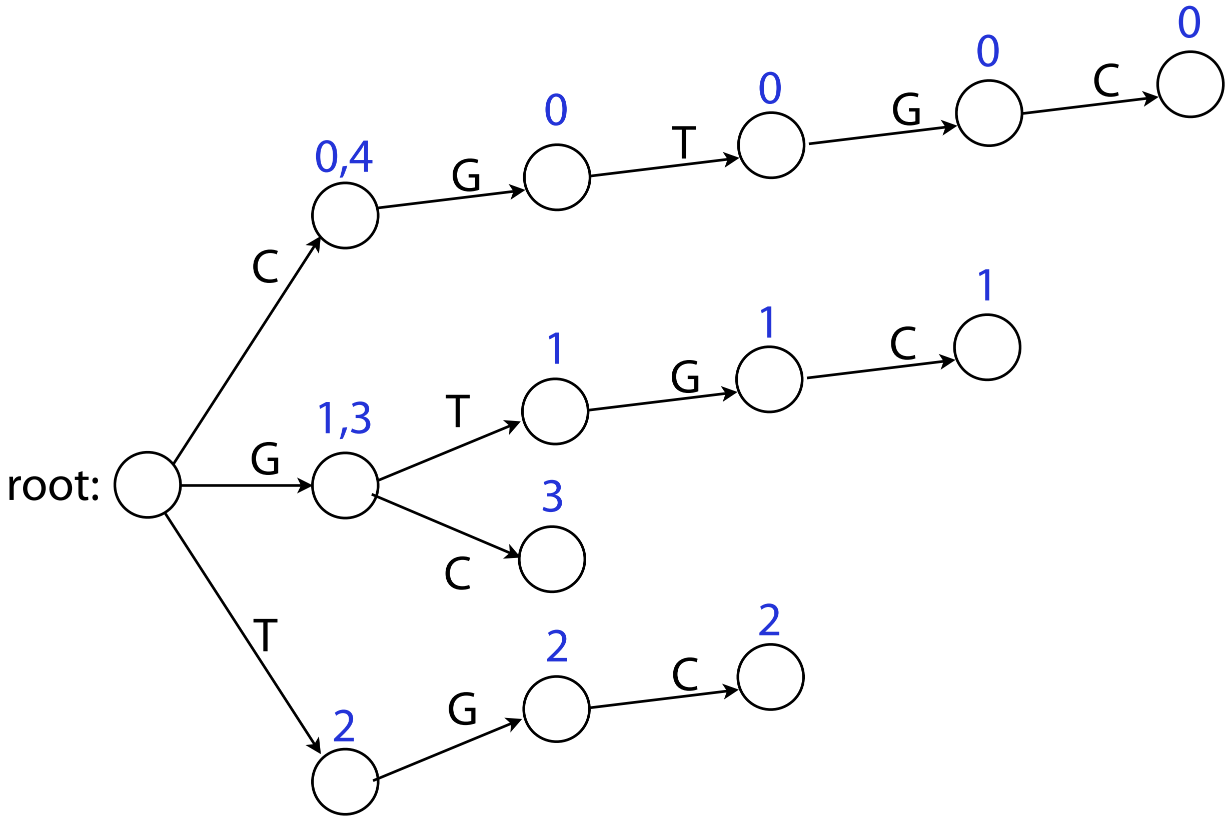
Implement exact pattern matching using this trie

Consider: How could we search the trie if we are only allowed to store one value in each node [instead of a vector of them]?

Preprocessing for exact pattern matching

T: **C G T G C**

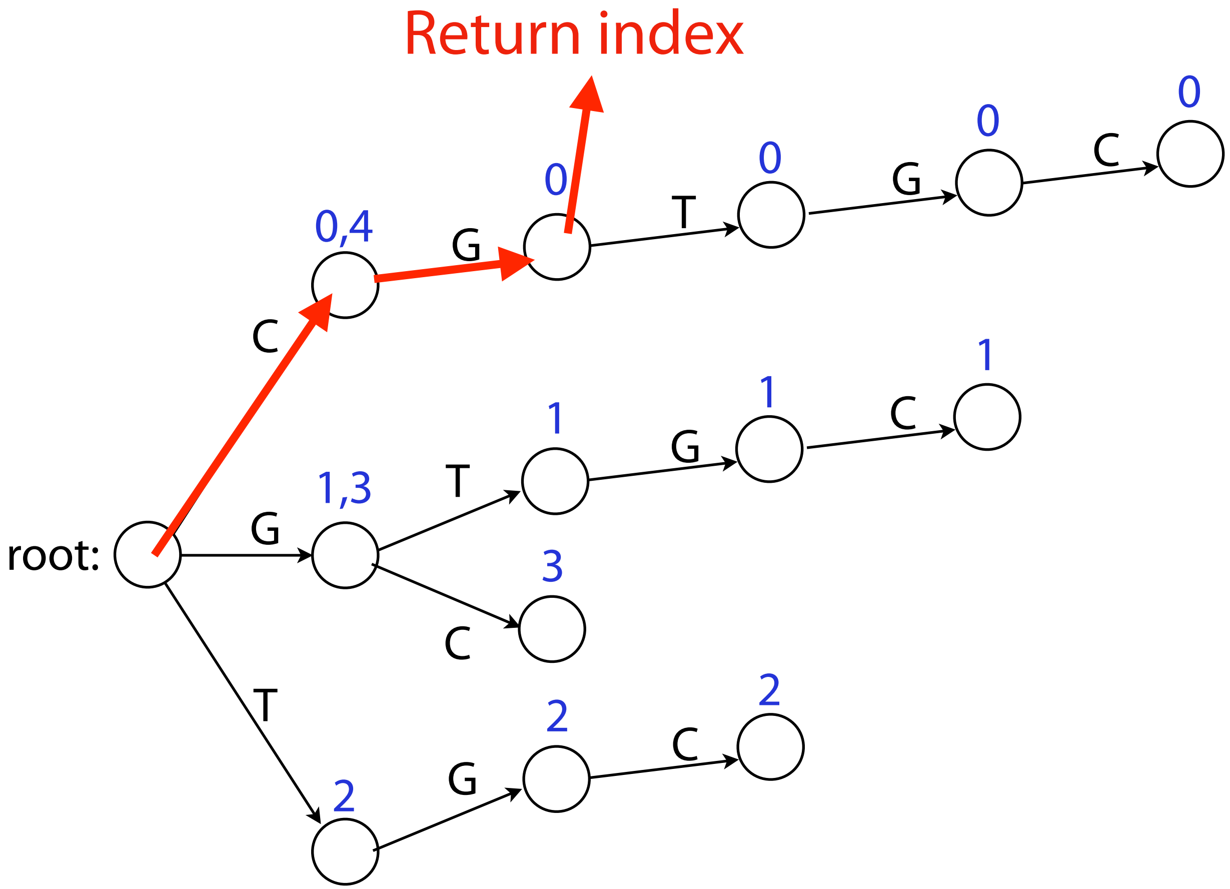
Key	Value
C	0
G	1
T	2
G	3
C	4
CG	0
GT	1
TG	2
…	…



Preprocessing for exact pattern matching

T: **C G T G C**

Key	Value
C	0
G	1
T	2
G	3
C	4
CG	0
GT	1
TG	2
…	…



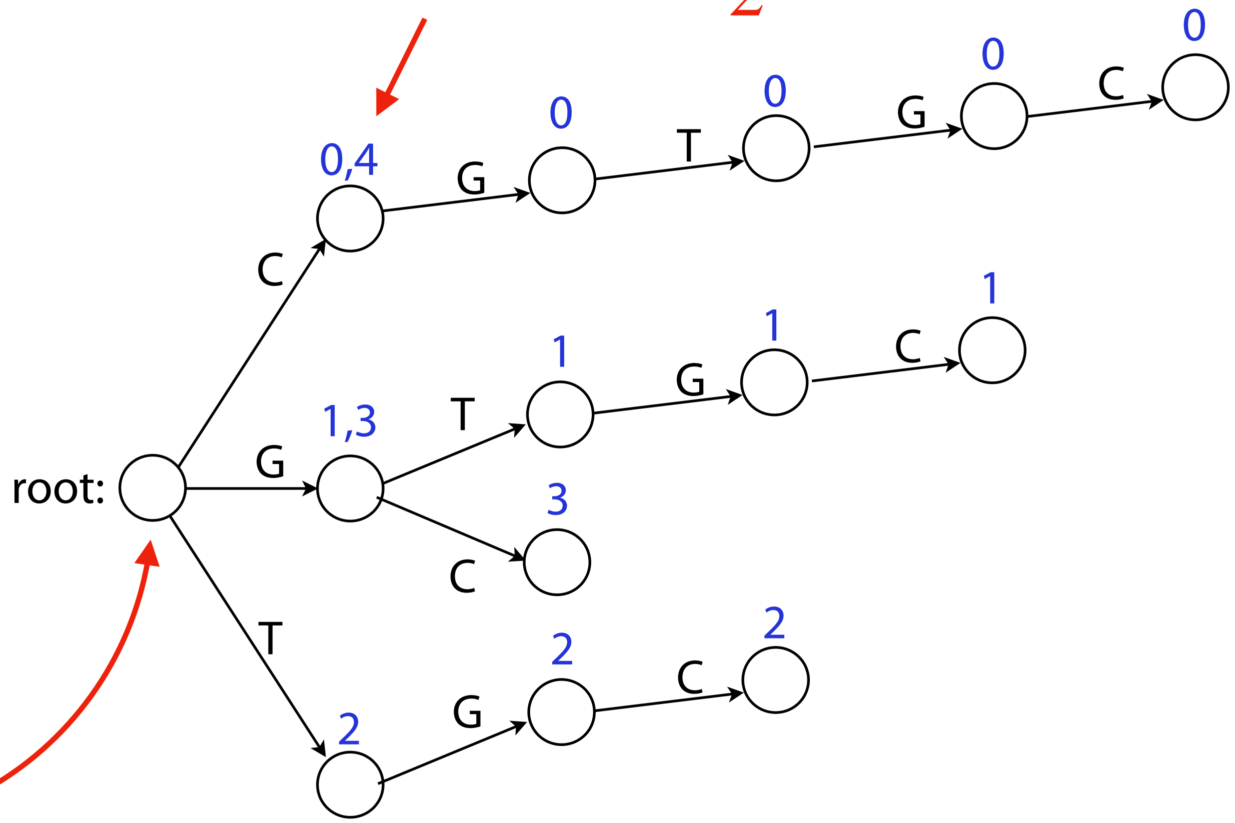
We can do exact pattern matching in $O(P)$ time!

Preprocessing for exact pattern matching

T: **C G T G C**

Key	Value
C	0
G	1
T	2
G	3
C	4
CG	0
GT	1
TG	2
…	…

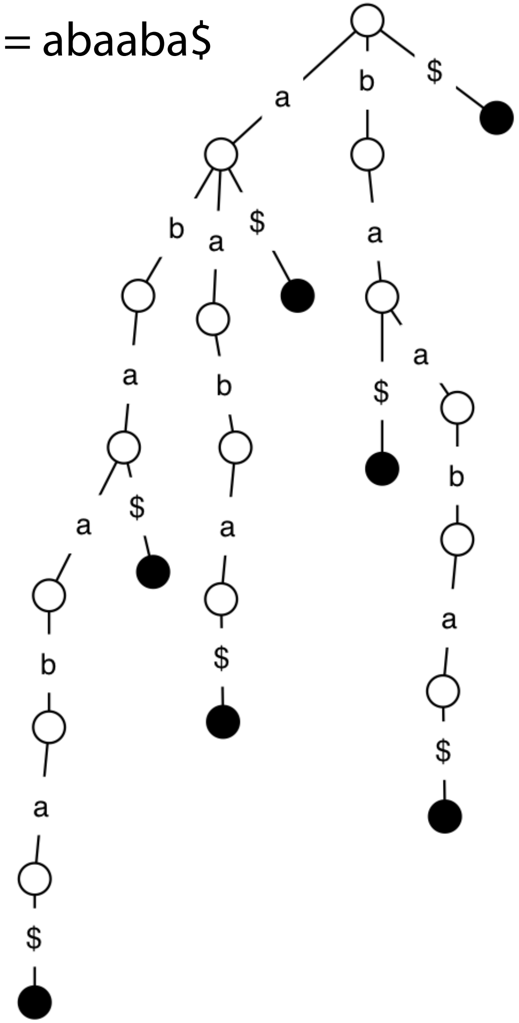
We are storing $\dfrac{|T|(|T|+1)}{2}$ values



We had to do $\dfrac{|T|(|T|+1)}{2}$ insertions

Preprocessing for exact pattern matching

T = abaaba$



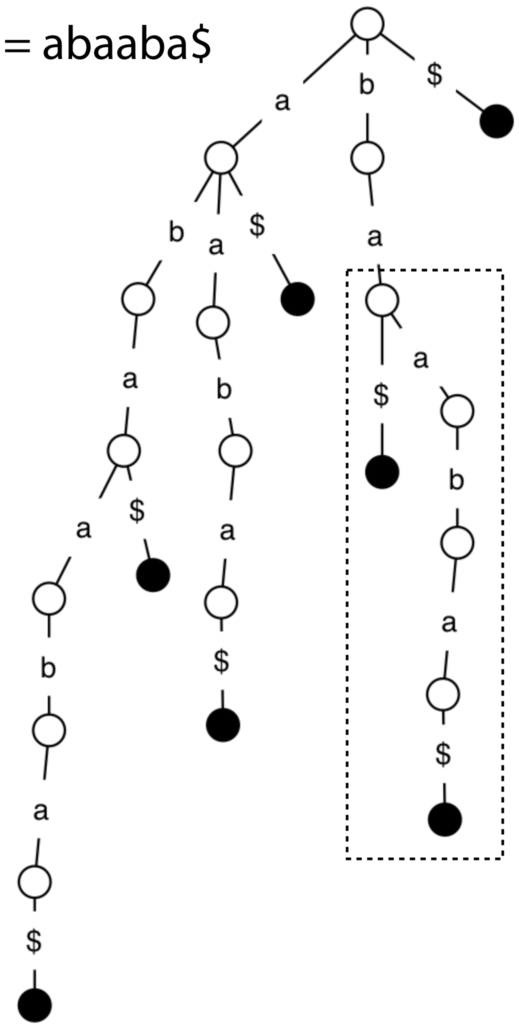
If only there was a way…

to insert fewer strings

to store fewer values

Preprocessing for exact pattern matching

T = abaaba$



If only there was a way…

to insert fewer strings

to store fewer values

to be even more efficient!