

String Algorithms and Data Structures

Tries

CS 199-225

Brad Solomon

February 23, 2026

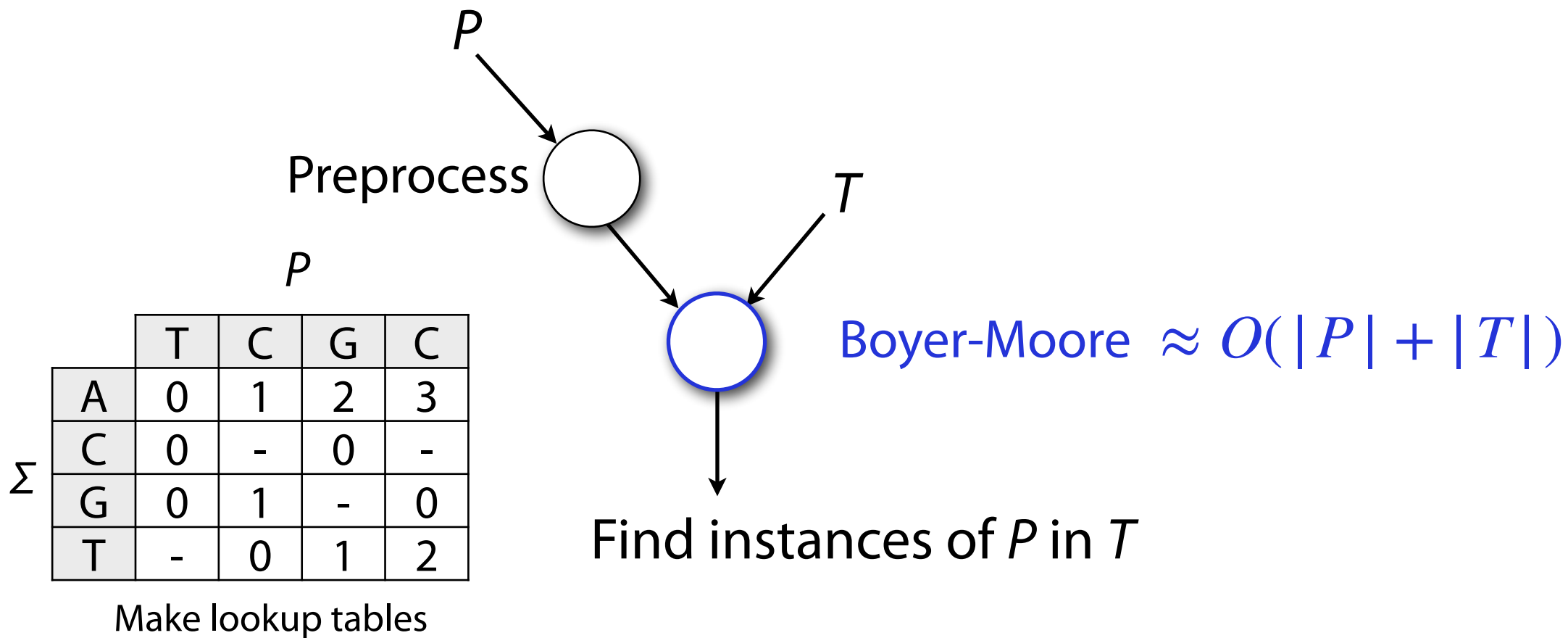


UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science

Exact pattern matching w/ Boyer-Moore

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



Preprocessing: Live chat streams

GCEvans
C++ and Data Structures

Tree Property: height
 $height(T)$: length of the longest path from the root to a leaf

Given a binary tree T:

$height(T) = 1 + \max(h(T_L), h(T_R))$

$h(\emptyset) = -1$
 $h(\text{single node } \{r, \emptyset, \emptyset\}) = 0$

00:23:35 01:14:37

Chat on Videos

19:59 **225user**: null

20:24 **DOgee_**: 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 **DOgee_**: 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 **mannthatsme**: What if there is only one root in the tree, is the height 0?

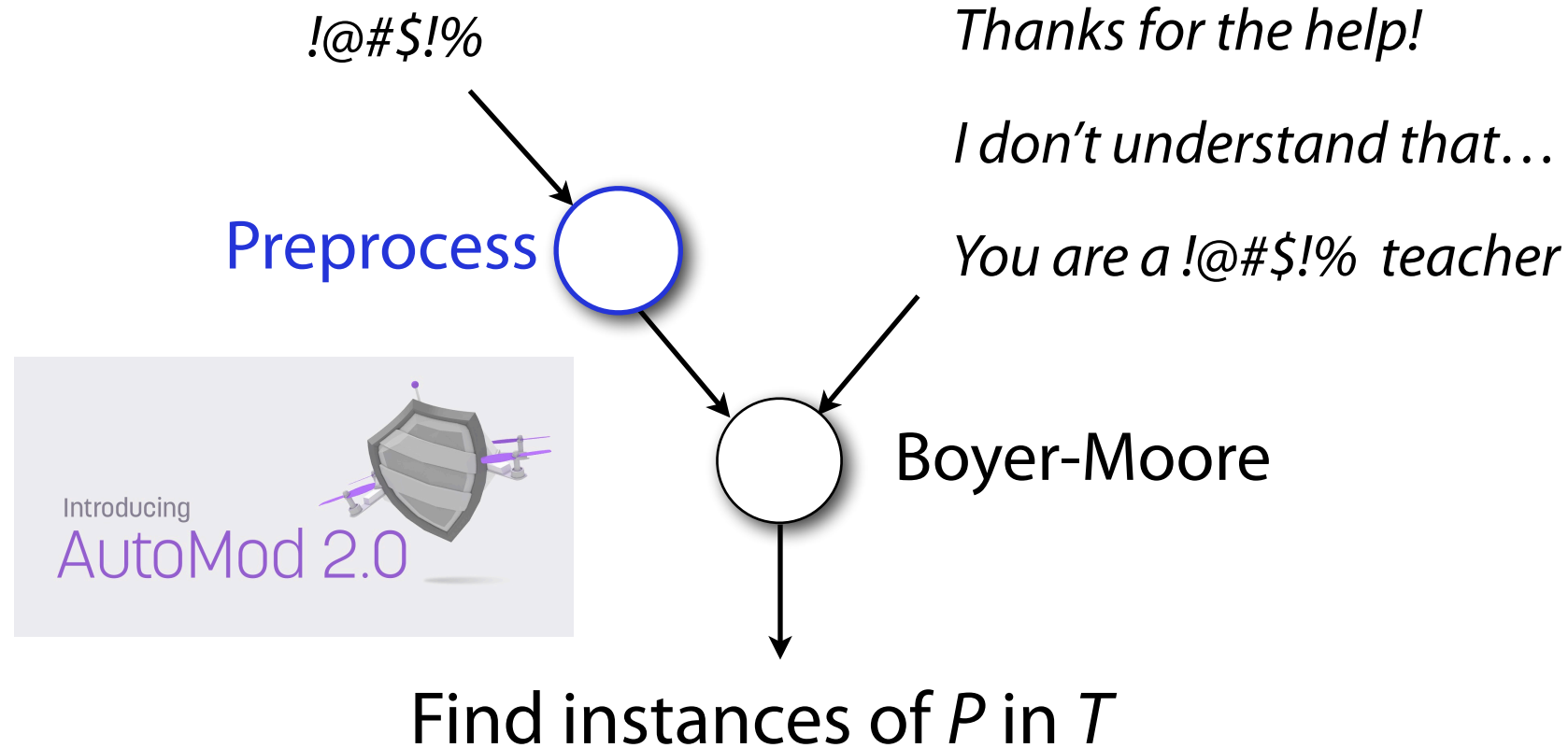
21:38 **BassyTheSassy**: is the height to the lowest leaf, or a leaf

21:52 **fantah_k**: ohhh okay yeah that makes sense

Patterns: banned phrases

Text: Chat messages

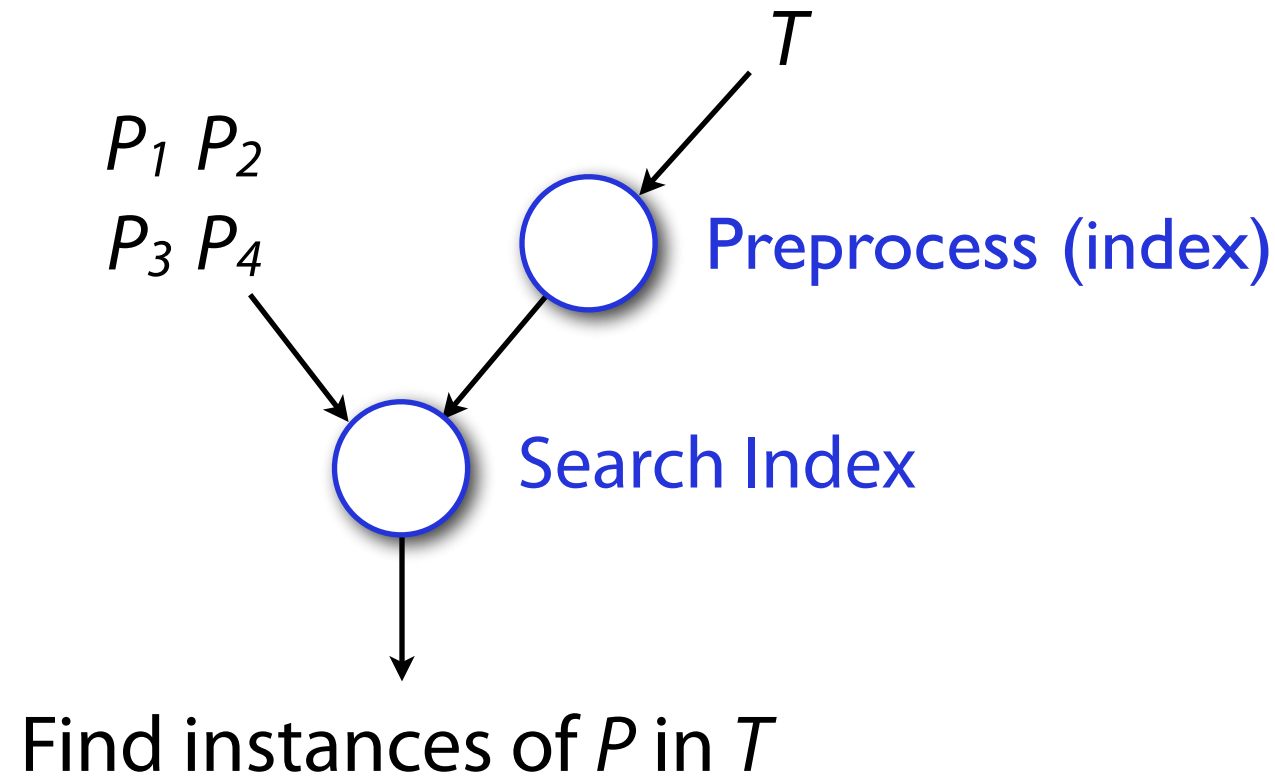
Preprocessing: Live chat streams



Amortize cost of preprocessing P over many T

Exact pattern matching *w/ indexing*

Conventionally $T \gg P$:



Amortize cost of preprocessing T over many P

Preprocessing: Libraries



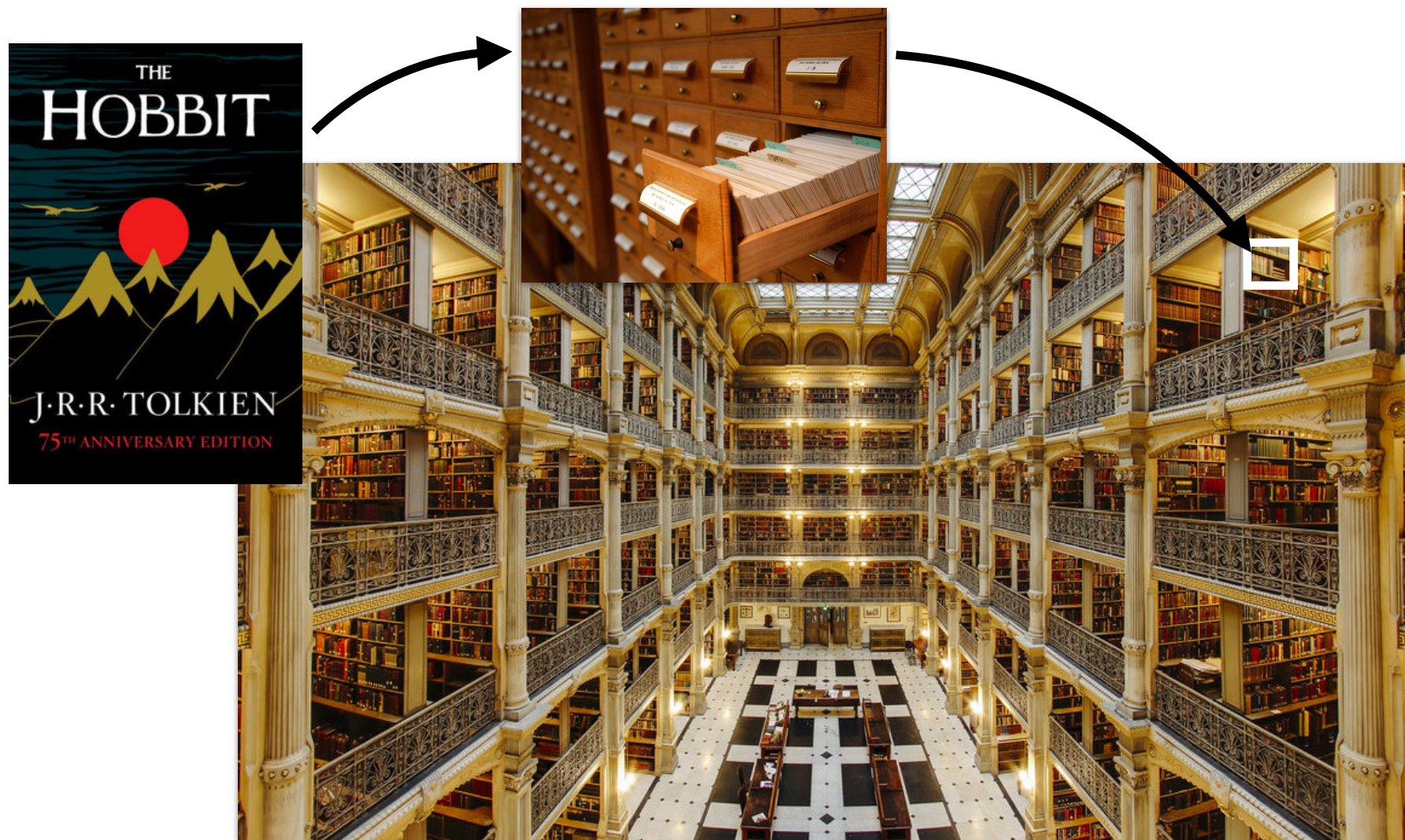
Preprocessing: Libraries



Patterns: Book of interest

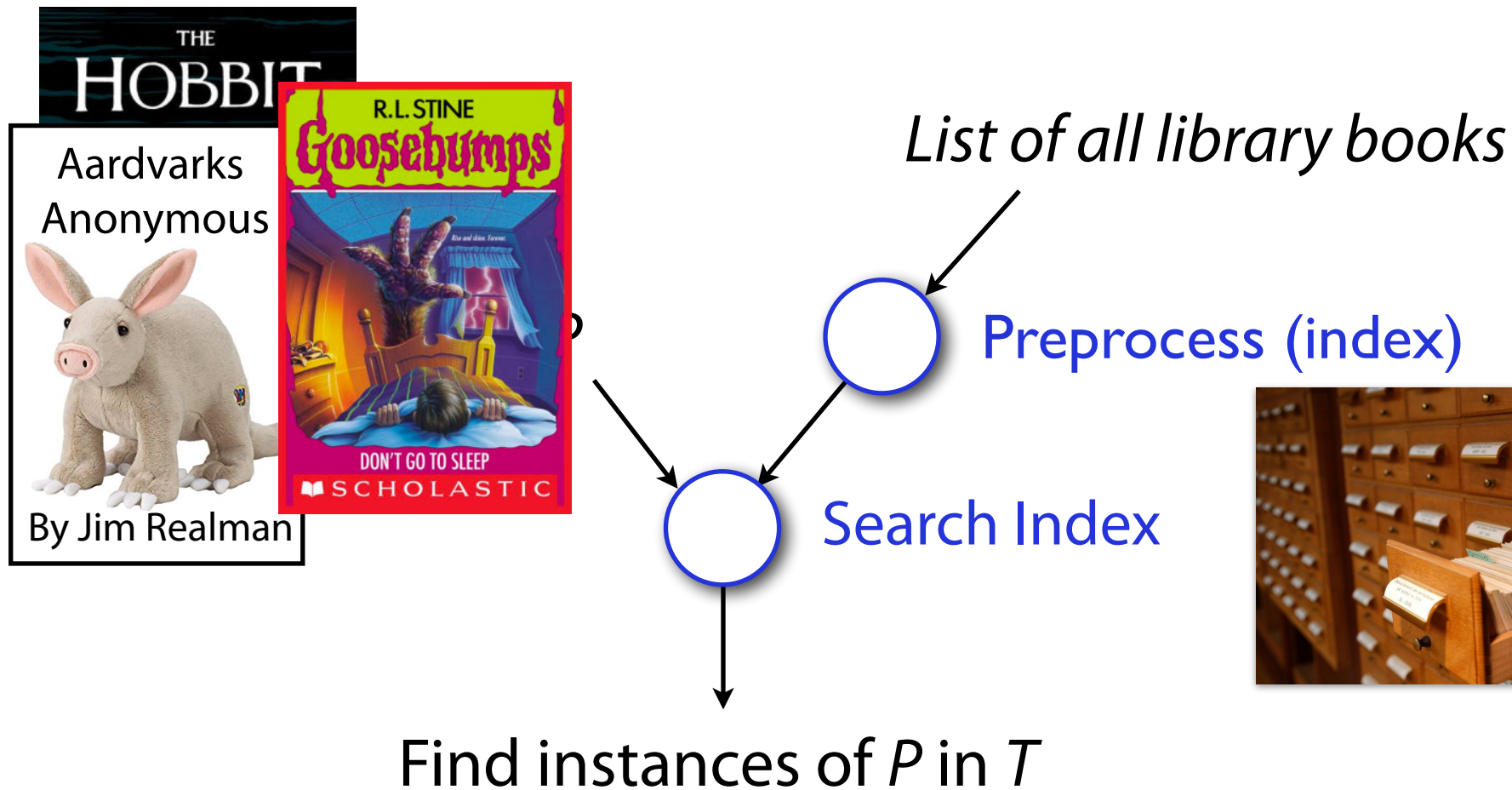
Text: All books in library

Preprocessing: Libraries



Preprocess the library by *indexing* all the books

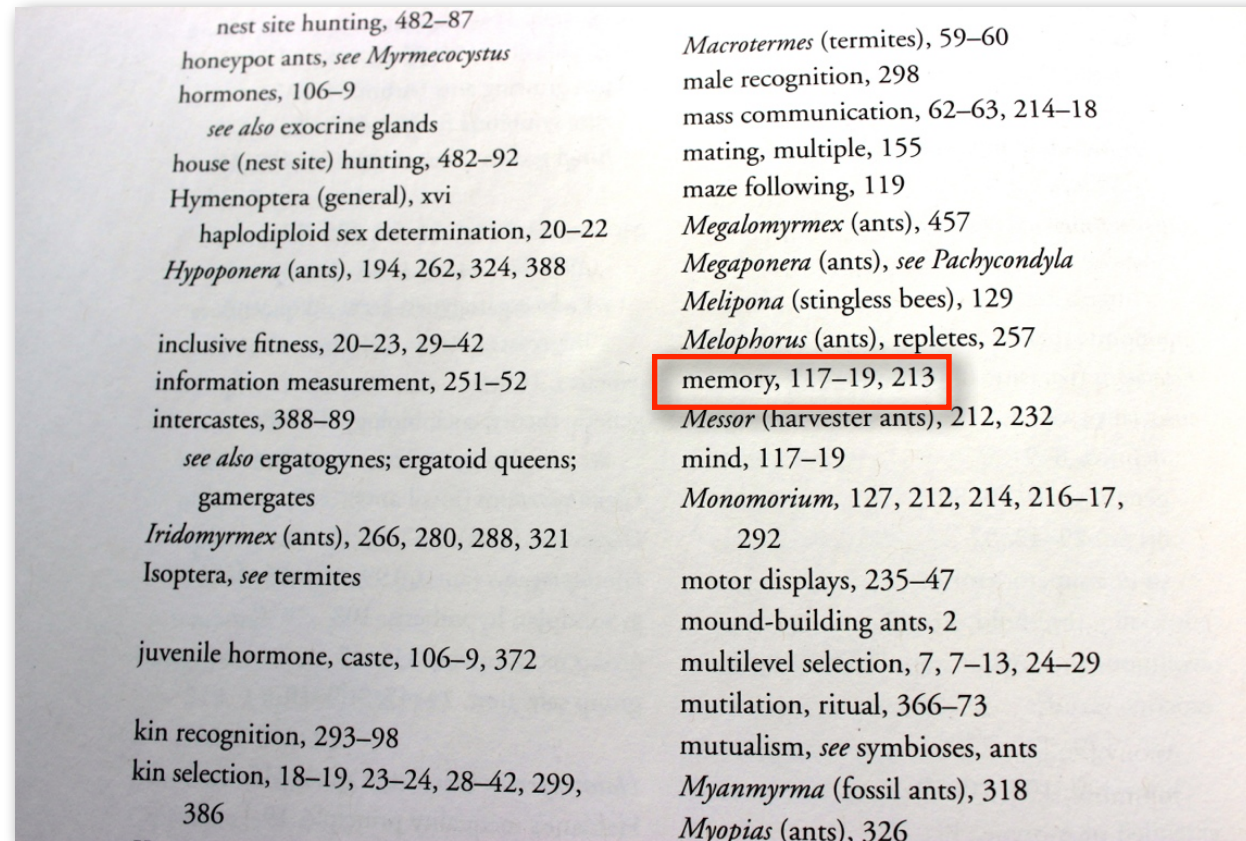
Preprocessing: Libraries



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

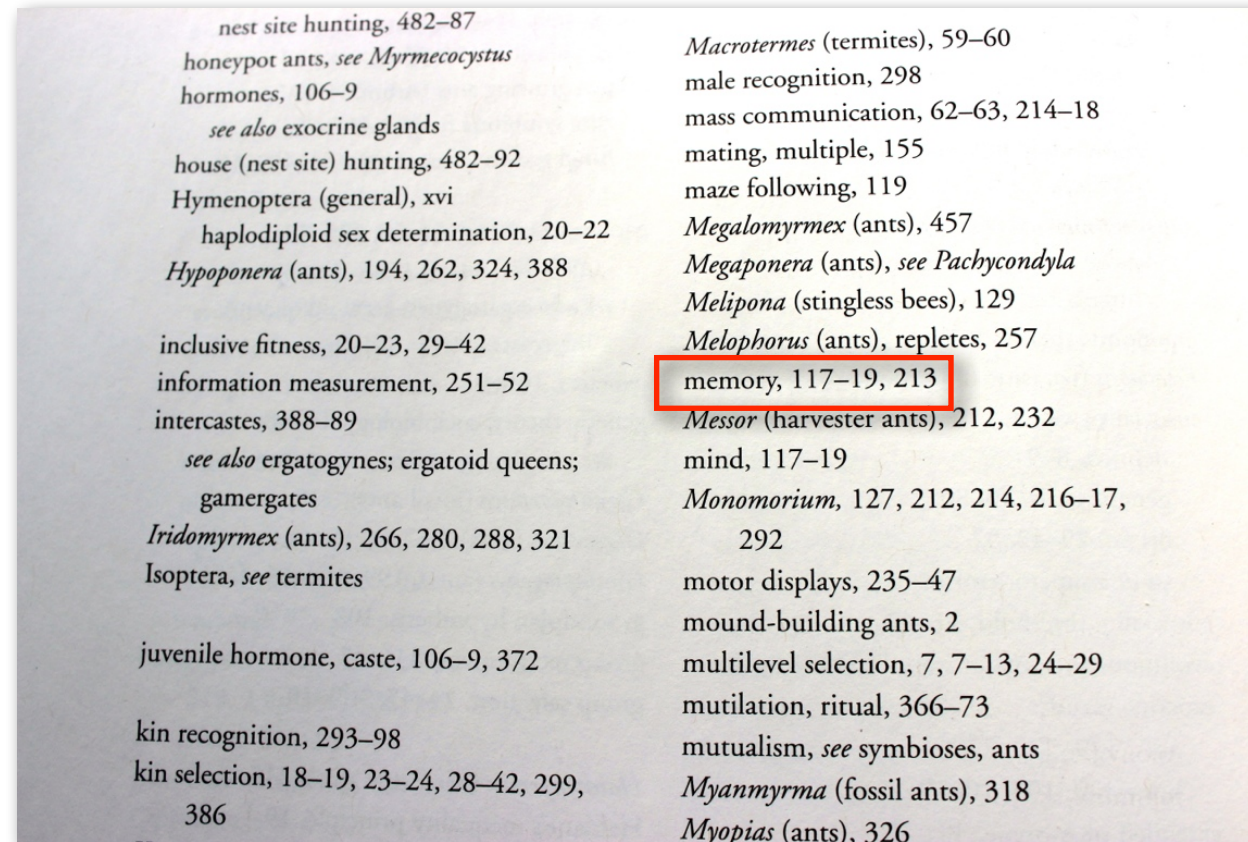
Preprocessing: Glossaries

What method of preprocessing is this?



Preprocessing: Glossaries

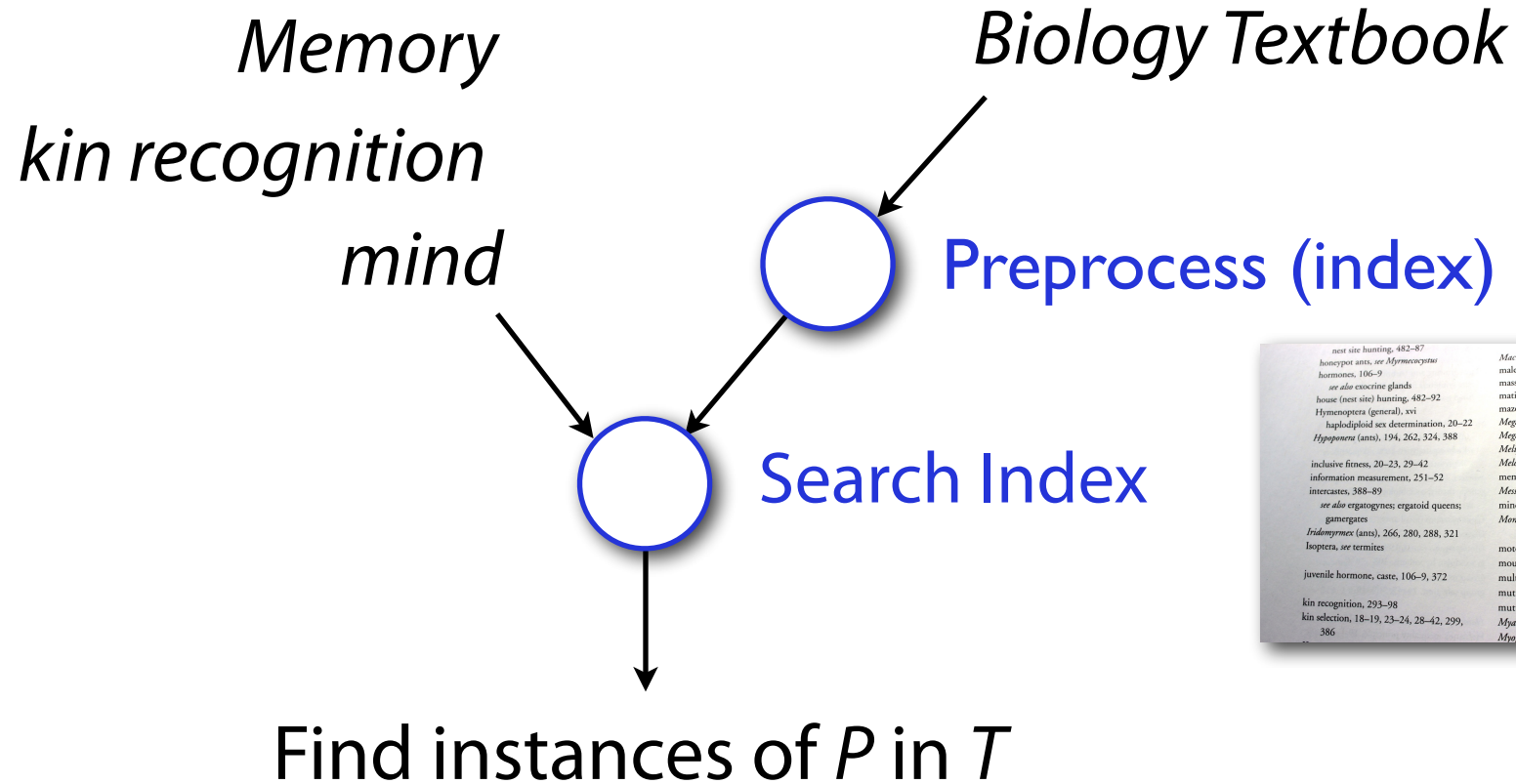
What method of preprocessing is this?



Patterns: Key terms

Text: All text in the book

Preprocessing: Glossaries

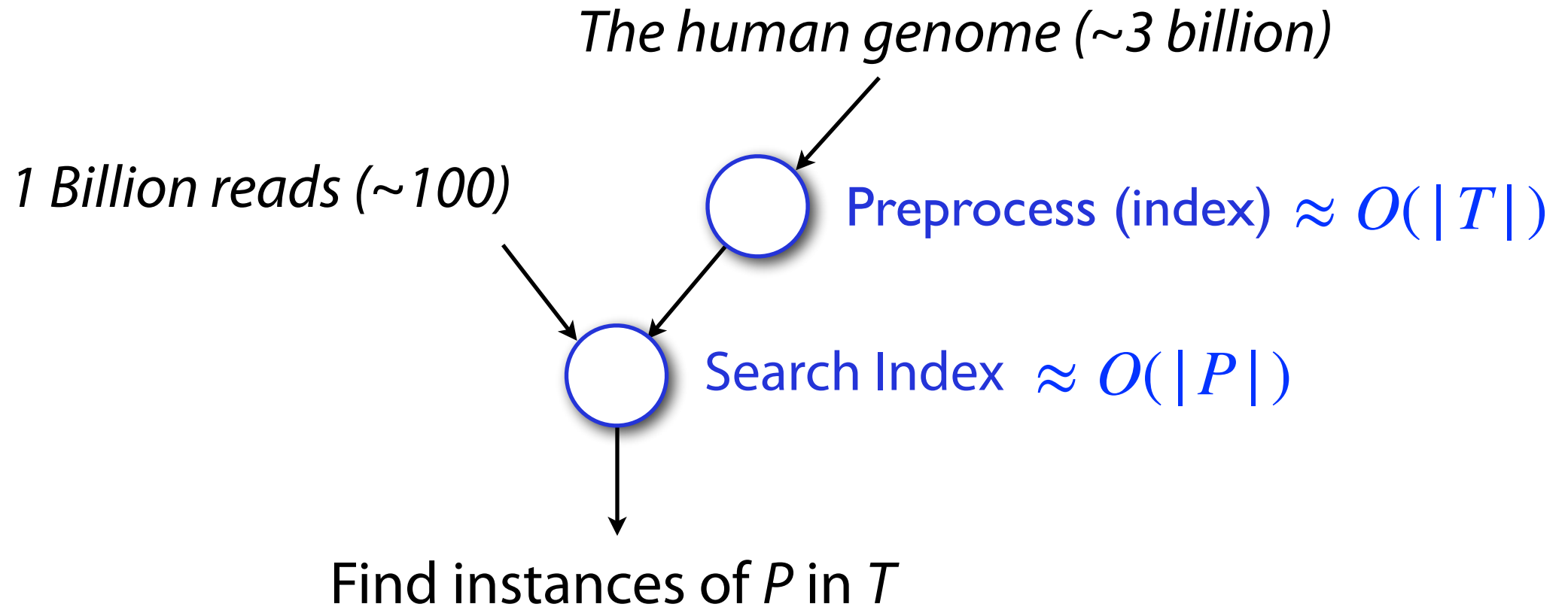


nest site hunting, 482-87	Macrotermes (termites), 59-60
honeypot ants, <i>see Myrmecocystus</i>	male recognition, 298
hormones, 106-9	mass communication, 62-63, 214-18
<i>see also</i> exocrine glands	mating, multiple, 155
house (nest site) hunting, 482-92	maze following, 119
Hymenoptera (general), xvi	<i>Megalomyrma</i> (ants), 457
haplodiploid sex determination, 20-22	<i>Megaponera</i> (ants), <i>see Pachycondyla</i>
<i>Hypoponera</i> (ants), 194, 262, 324, 388	<i>Melipona</i> (stingless bees), 129
inclusive fitness, 20-23, 29-42	<i>Melophorus</i> (ants), repletes, 257
information measurement, 251-52	memory, 117-19, 213
intercastes, 388-89	<i>Messor</i> (harvester ants), 212, 232
<i>see also</i> ergatogynes; ergatoid queens;	mind, 117-19
gamergates	<i>Monomorium</i> , 127, 212, 214, 216-17,
<i>Iridomyrmex</i> (ants), 266, 280, 288, 321	292
Isoptera, <i>see</i> termites	motor displays, 235-47
juvenile hormone, caste, 106-9, 372	mound-building ants, 2
kin recognition, 293-98	multilevel selection, 7, 7-13, 24-29
kin selection, 18-19, 23-24, 28-42, 299,	mutilation, ritual, 366-73
386	mutualism, <i>see</i> symbioses, ants
	<i>Myrmomyrma</i> (fossil ants), 318
	<i>Myrpicia</i> (ants), 326

Glossary built on total contents T , useful for multiple P

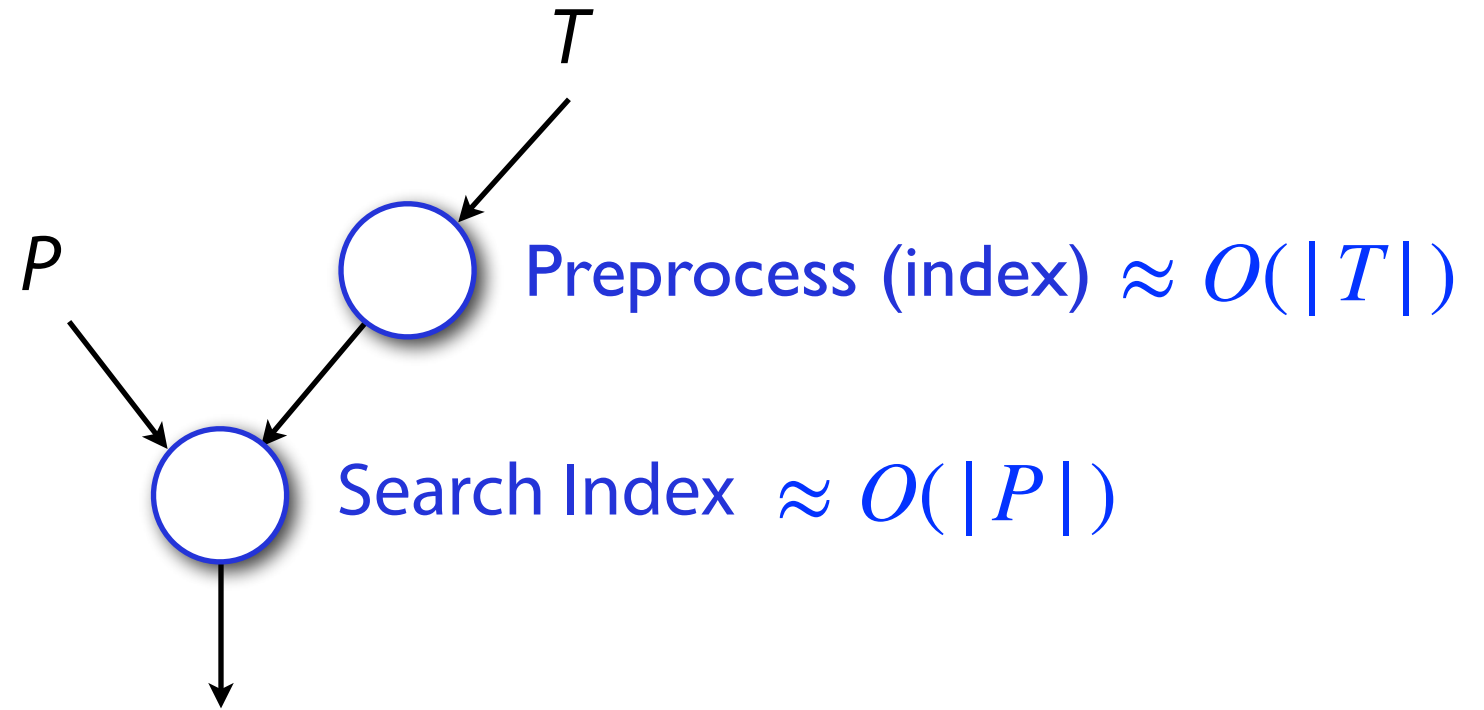


Exact pattern matching *w/ indexing*



Amortize cost of preprocessing T over many P

Exact pattern matching *w/ indexing*



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
G
T
G C
C G
G T
T G
G C
C G T
G T G
T G C



0
1
2
3
4
0
1
2
3
0
1
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

C G
G T
T G
G C

C G T
G T 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

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
 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
...	...

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

↑
↓

Preprocessing for exact pattern matching

Hash Table lookup: $O(1)$ *

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

Cost: $O(|T|^2)$ space

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

Tradeoff: Target search goal is $O(|P|)$ without $O(|T|^2)$ space!

Preprocessing for exact pattern matching



Because our keys are strings, this is sometimes possible!

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

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

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

C

C G

C G T

C G T G

C G T G C

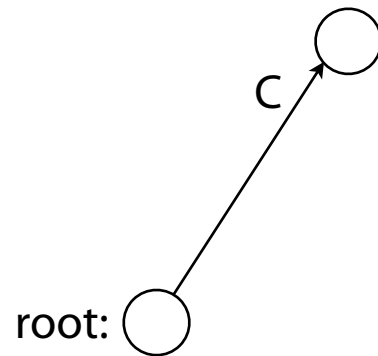
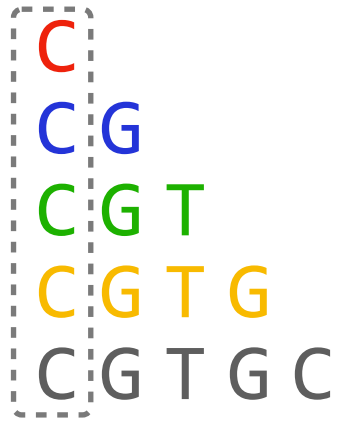
root: ○

Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

T: C G T G C

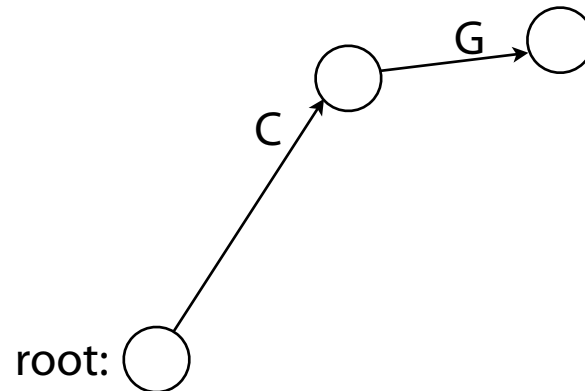
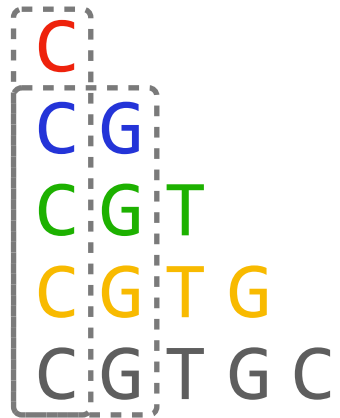


Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

T: C G T G C

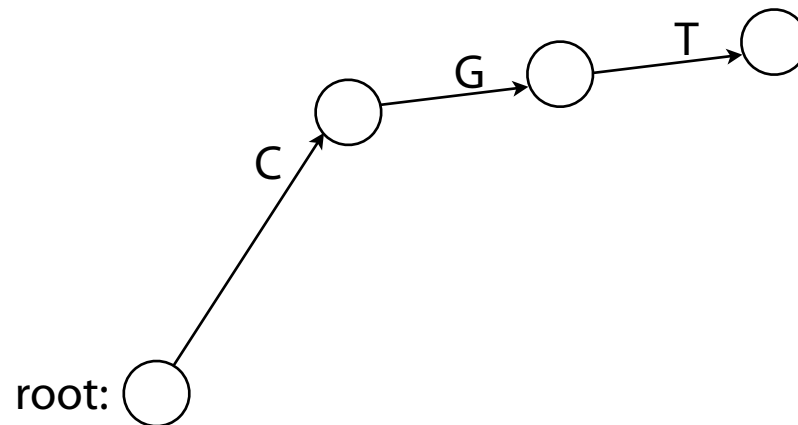
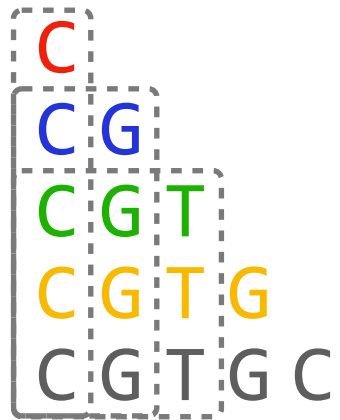


Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

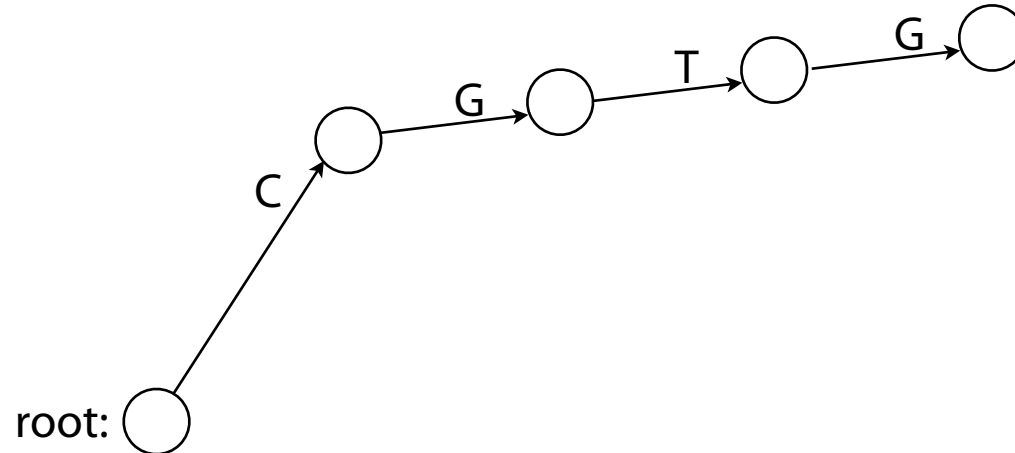
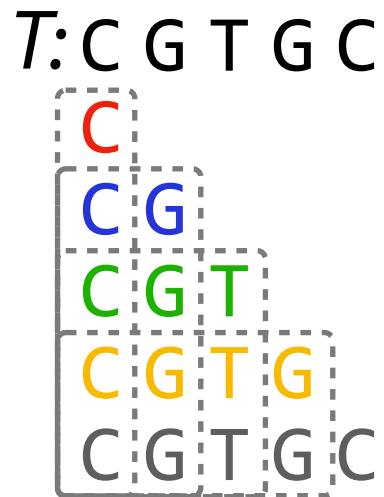
T: C G T G C



Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

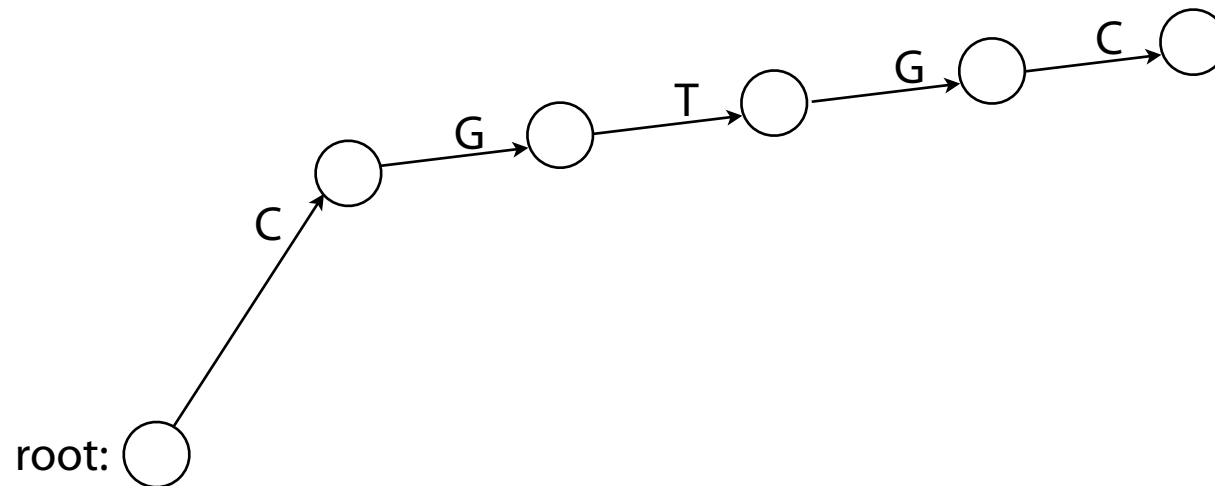
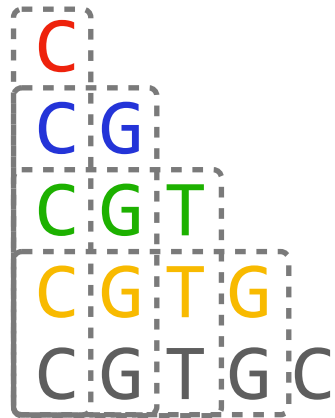


Preprocessing for exact pattern matching

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:

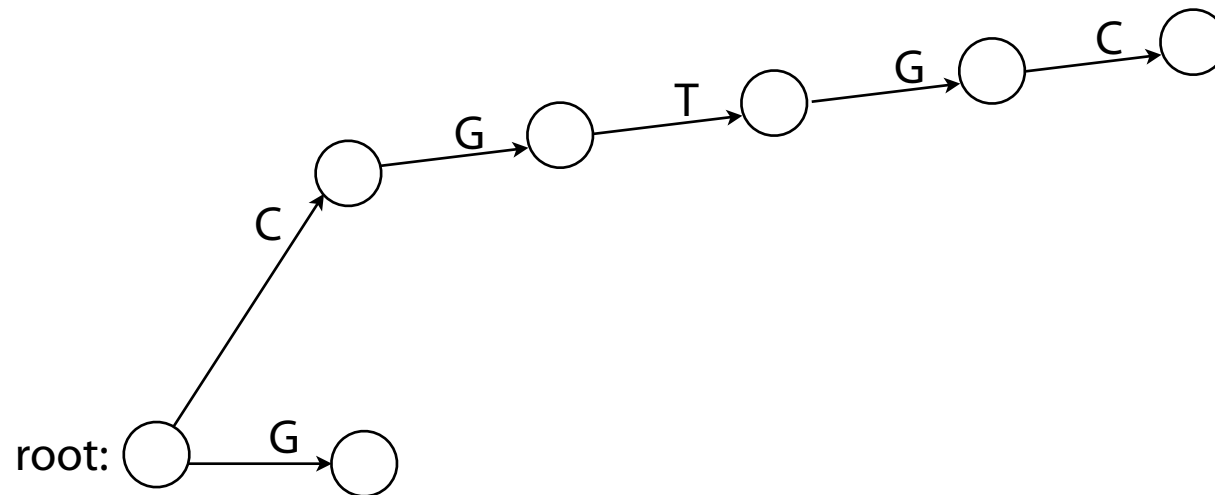
T: C G T G C

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:

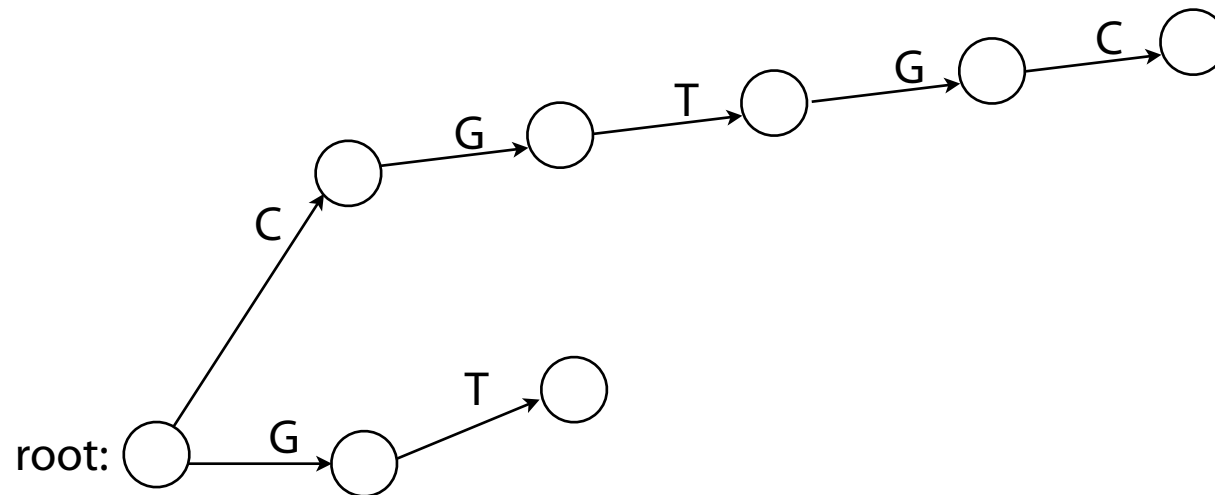
T: C G T G C

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:

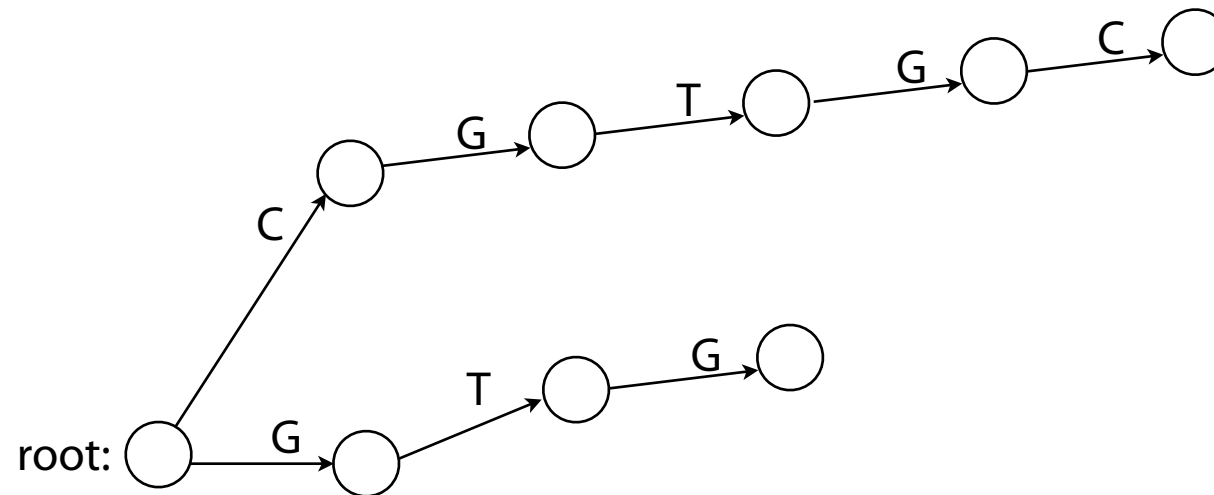
T: C G T G C

G

G T

G T G

G T G C

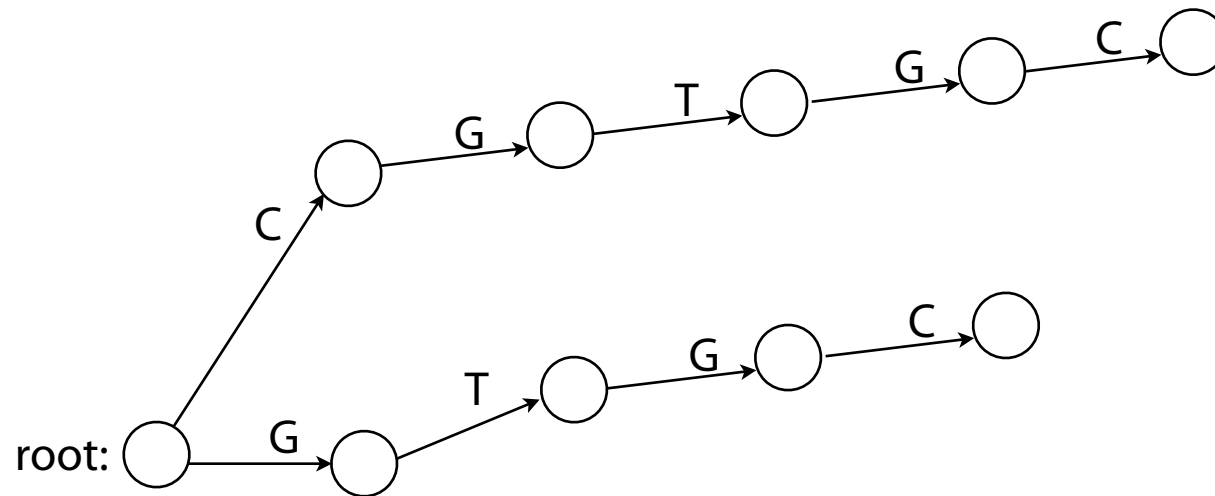


Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

T: C G T G C
 G
 G T
 G T G
 G T G C

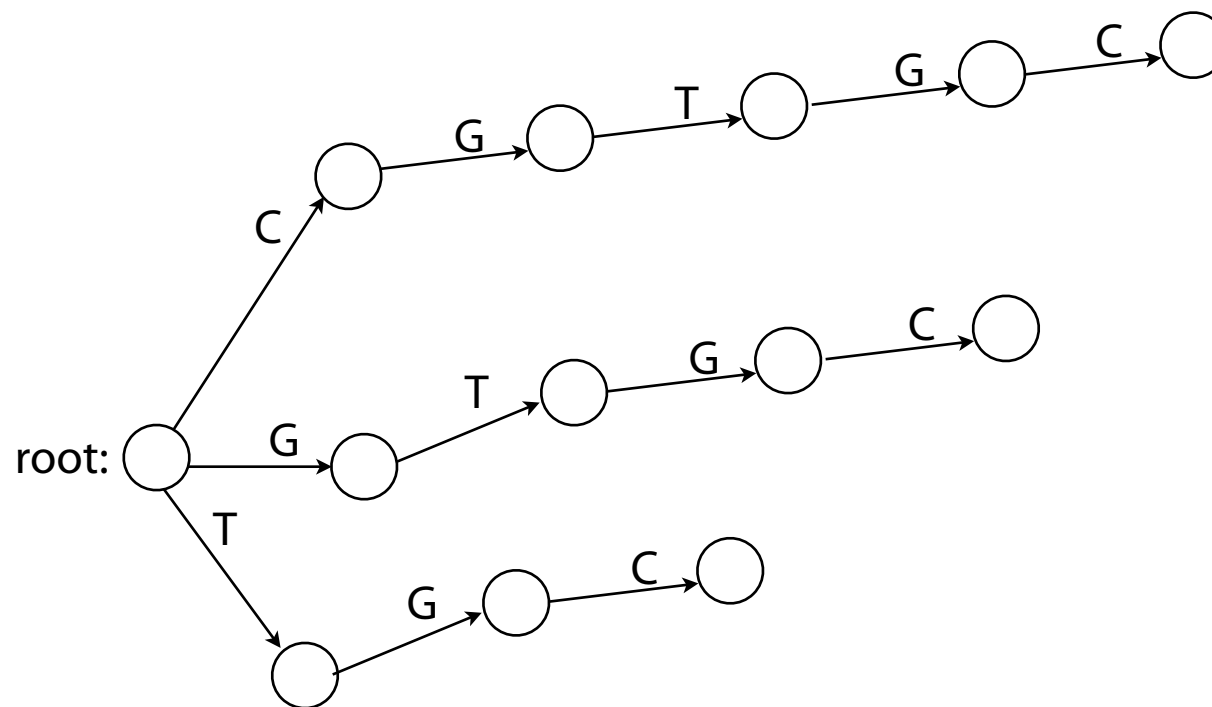


Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

T: C G T G C
 T
 T G
 T G C

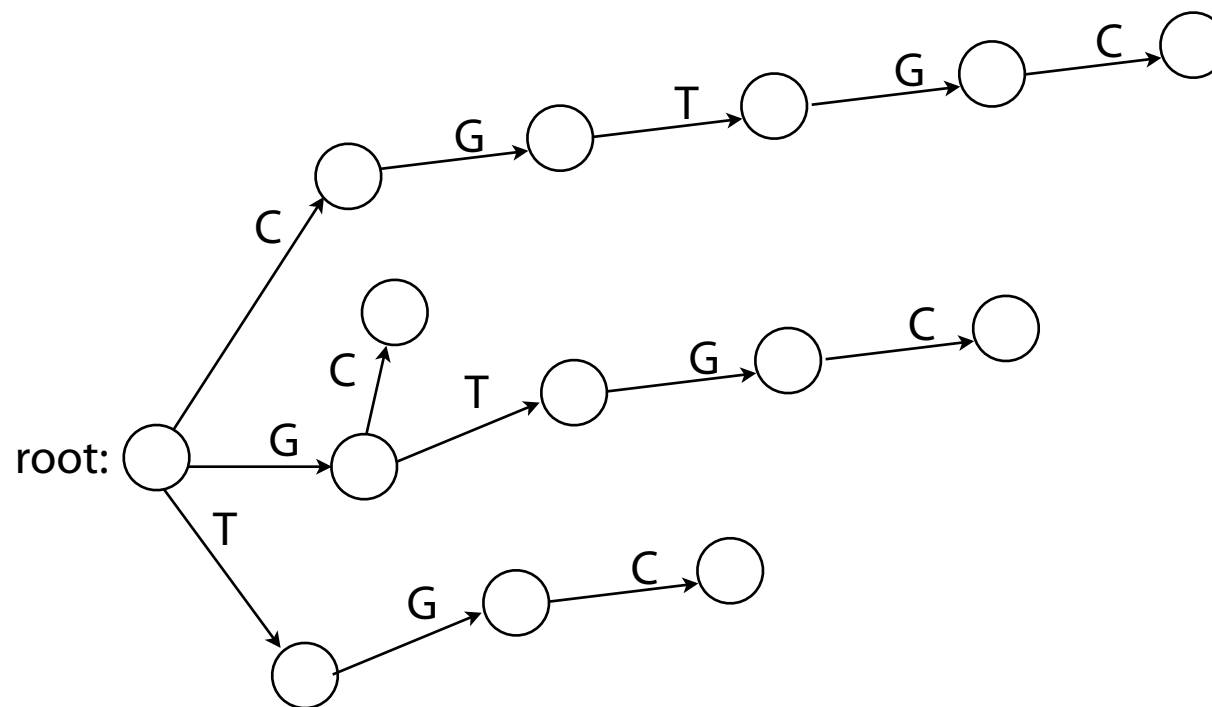


Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

T: C G T G C
 G
 G C

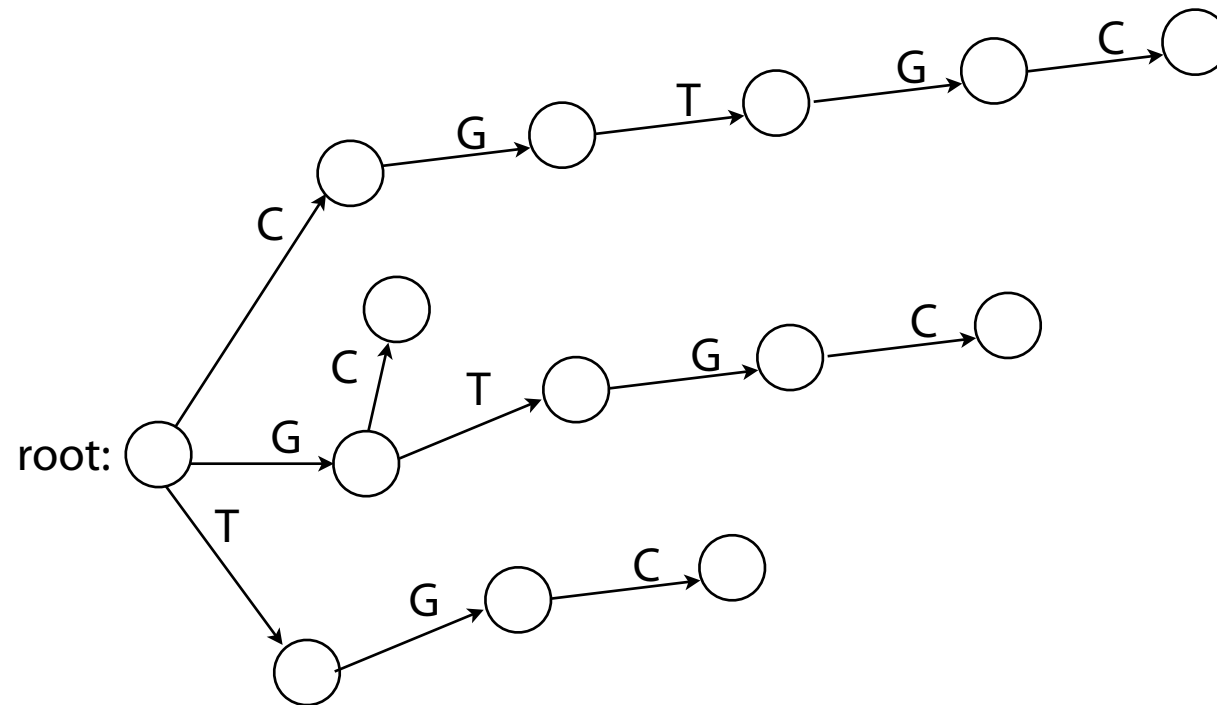


Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

T: C G T G C
C



String indexing with Tries

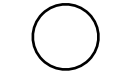
Trie: A rooted tree storing a collection of (key, value) pairs

Keys: Values:

`i n s t a n t` `1`

`i n t e r n a l` `2`

`i n t e r n e t` `3`

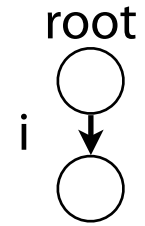
root


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

String indexing with Tries

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

Keys:	Values:
<code>i n s t a n t</code>	<code>1</code>
<code>i n t e r n a l</code>	<code>2</code>
<code>i n t e r n e t</code>	<code>3</code>



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

String indexing with Tries

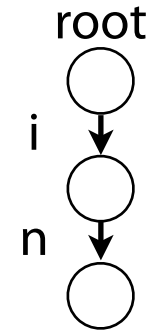
Trie: A rooted tree storing a collection of (key, value) pairs

Keys: Values:

`i n s t a n t` `1`

`i n t e r n a l` `2`

`i n t e r n e t` `3`



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

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

String indexing with Tries

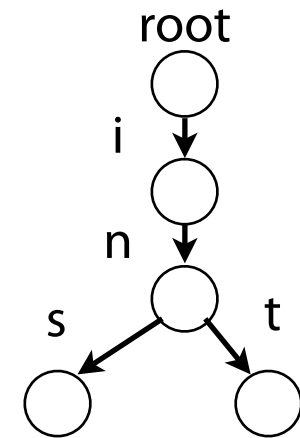
Trie: A rooted tree storing a collection of (key, value) pairs

Keys: Values:

`i n s t a n t` `1`

`i n t e r n a l` `2`

`i n t e r n e t` `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:

`i n s t a n t` `1`

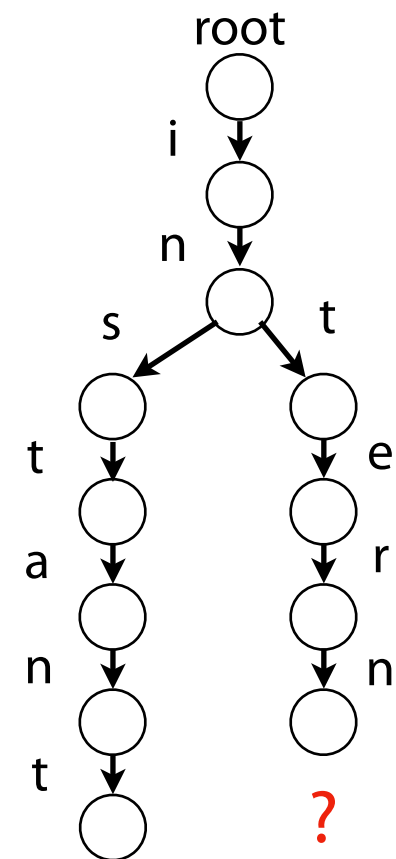
`i n t e r n a l` `2`

`i n t e r n e t` `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

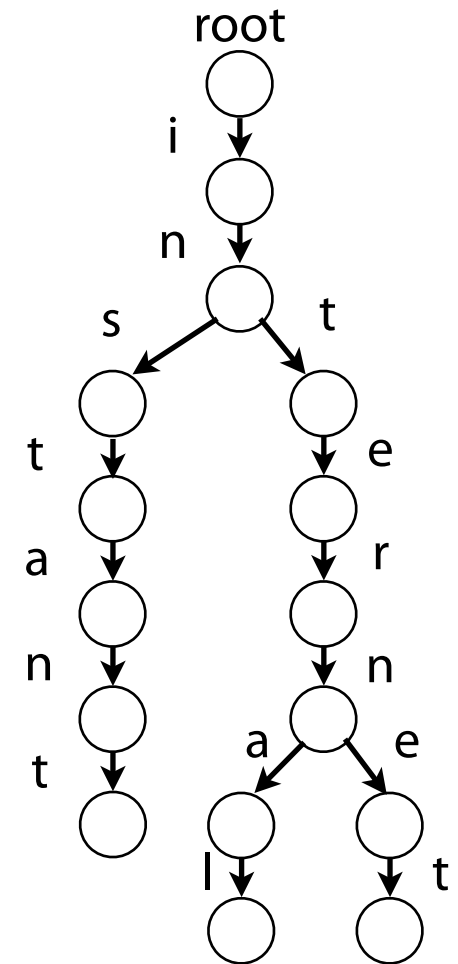
internal 2

internet 3

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

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

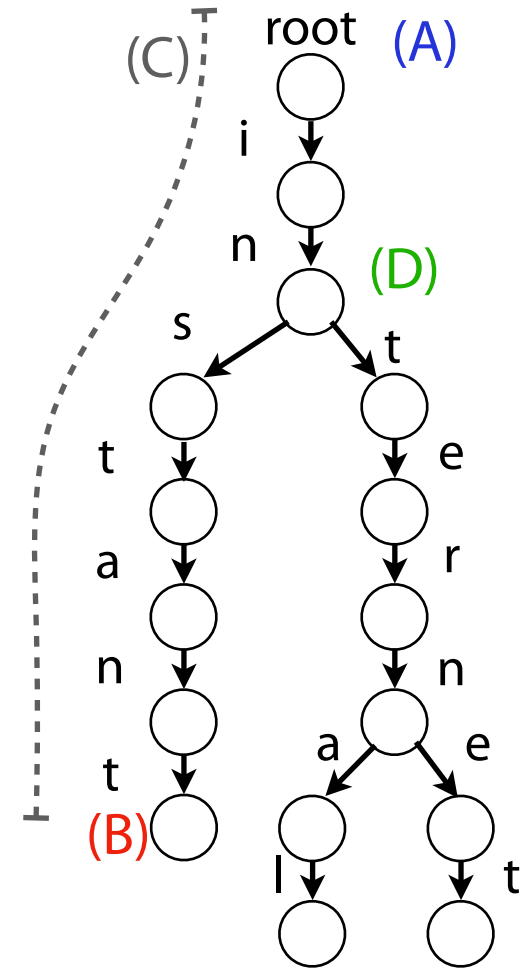


Join Code: 225

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

Keys:	Values:
<code>i n s t a n t</code>	1
<code>i n t e r n a l</code>	2
<code>i n t e r n e t</code>	3

Where should I store the value 1?

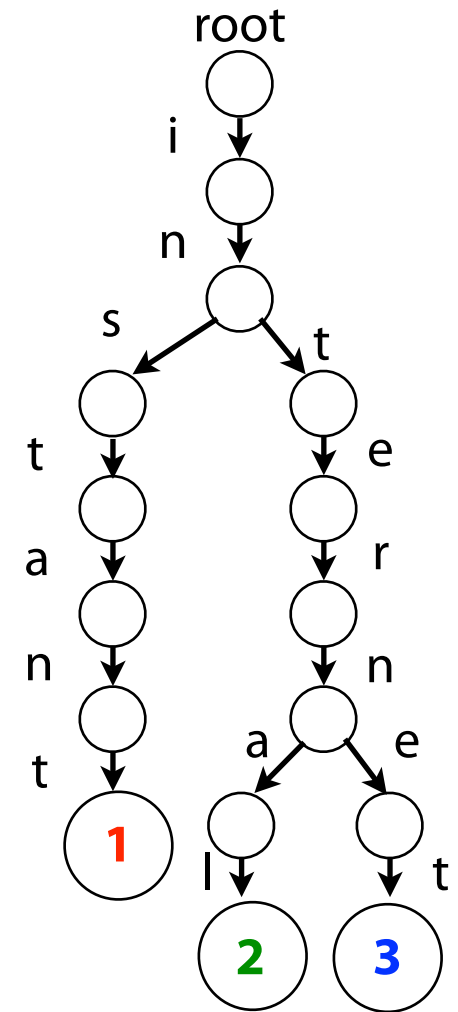


String indexing with Tries

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

Keys:	Values:
<code>i n s t a n t</code>	1
<code>i n t e r n a l</code>	2
<code>i n t e r n e t</code>	3

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



String indexing with Tries

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

Keys: Values:

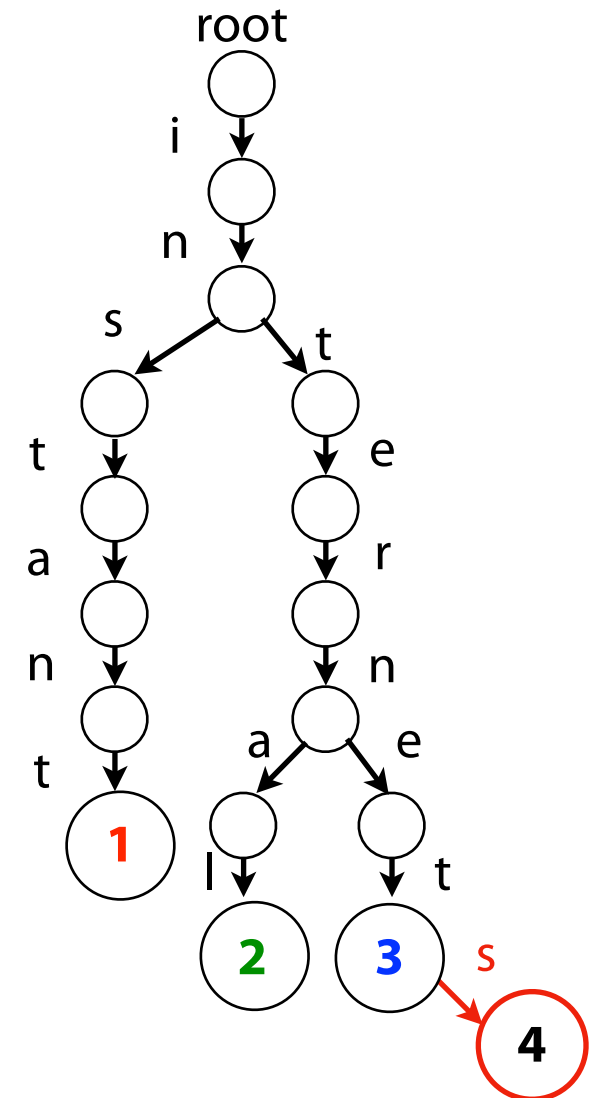
`i n s t a n t` `1`

`i n t e r n a l` `2`

`i n t e r n e t` `3`

`i n t e r n e t s` `4`

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





String indexing with Tries

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

We will use this to store our preprocessed index of T = C G T G C

Keys:

Values (Index Location):

C 0 , 4

G 1 , 3

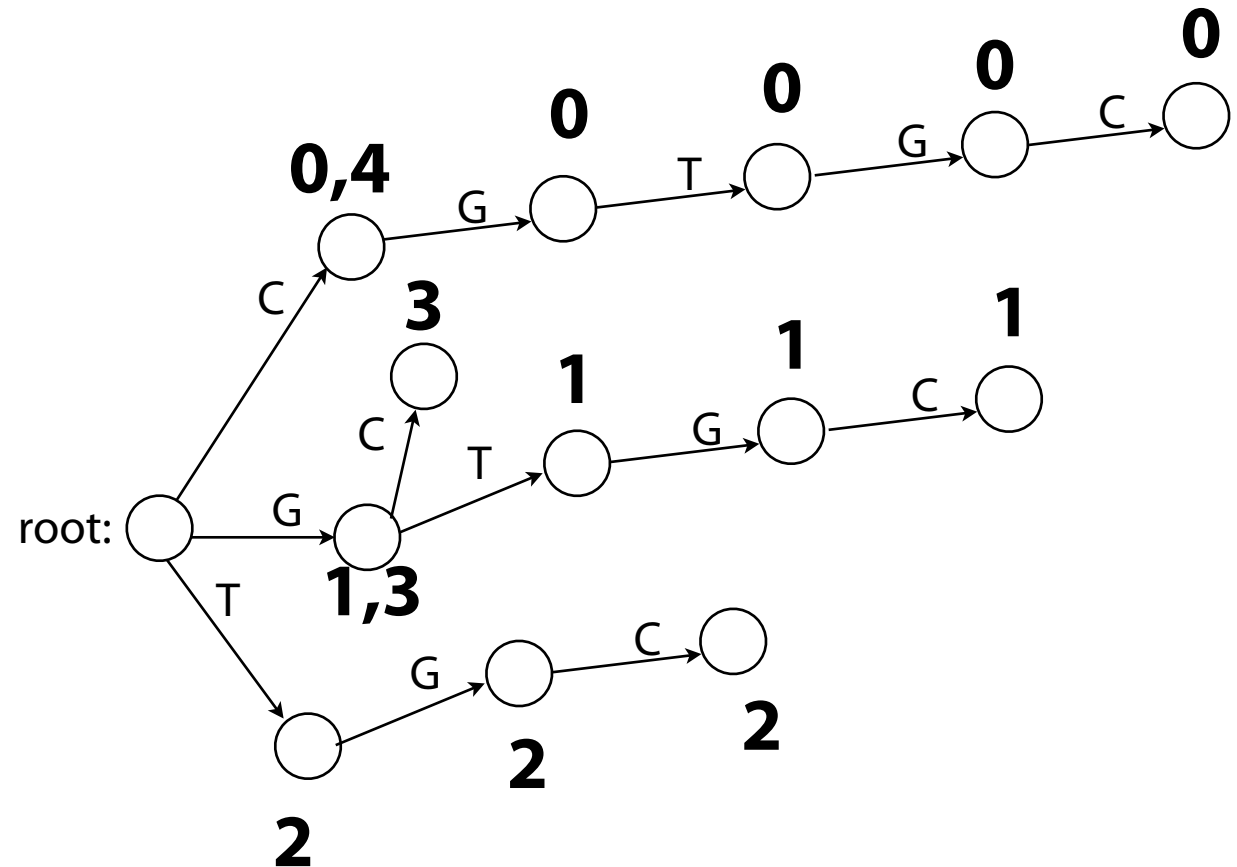
T 2

C G 0

G T 1

T G 2

G C 3

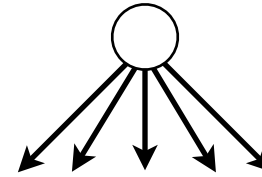


The Node Implementation

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

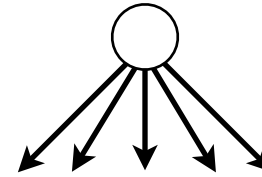
Each edge is a (potentially NULL) pointer.

How can we encode this?



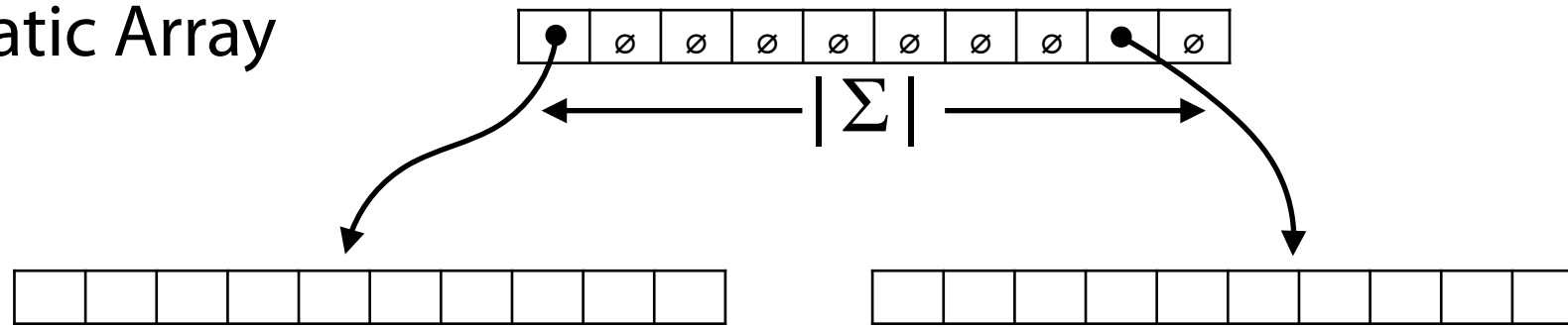
The Node Implementation

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

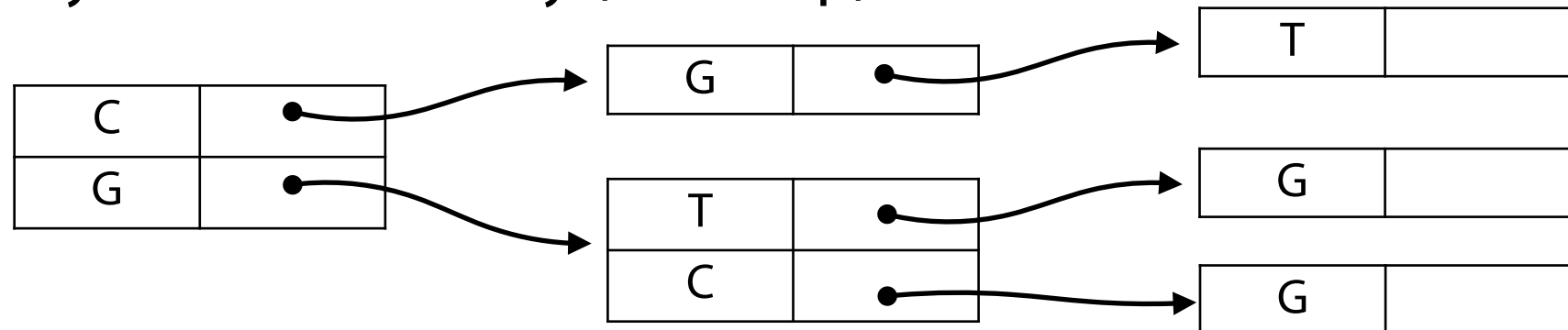


Each edge is a (potentially NULL) pointer.

1) Static Array



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



Trie Node Implementation

NaryTree.h

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

Trie Node Implementation

NaryTree.h

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

C G T G C

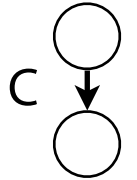


Trie Node Implementation

NaryTree.h

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

C G T G C
G T G C



Trie Node Implementation

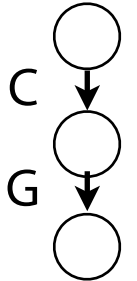
NaryTree.h

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

C G T G C

G T G C

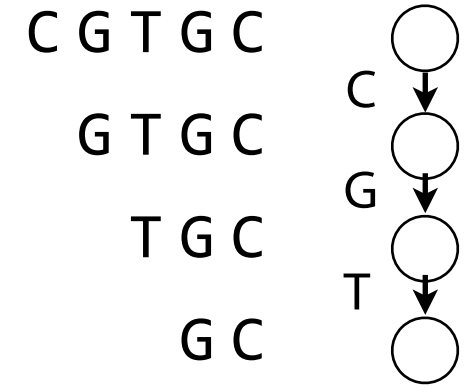
T G C



Trie Node Implementation

NaryTree.h

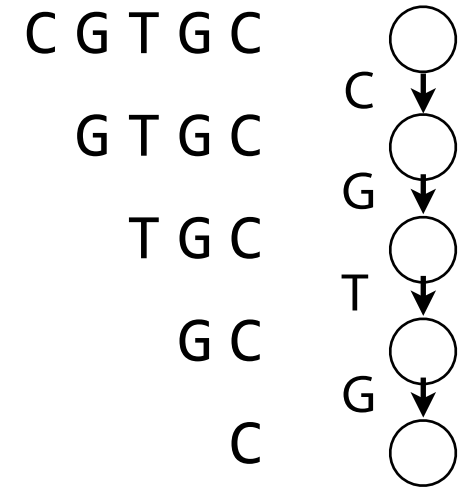
```
1 class NaryTree
2 {
3     public:
4         struct Node {
5             std::vector<int> index;
6             std::map<char, Node*> children;
7
8             Node(std::string s, int i)
9             {
10                if(s.length() > 0 ){
11                    children[s[0]] = new Node(s.substr(1),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



Trie Node Implementation

NaryTree.h

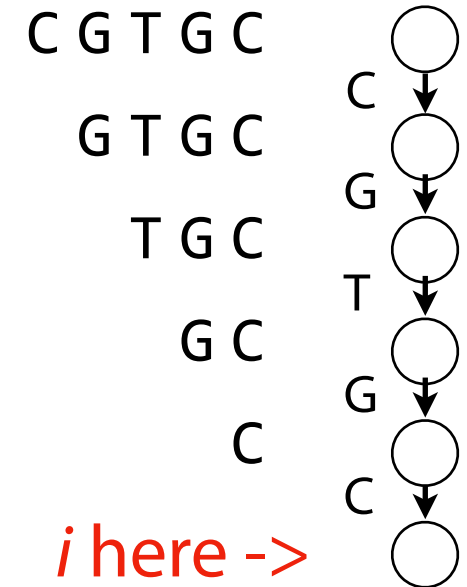
```
1 class NaryTree
2 {
3     public:
4         struct Node {
5             std::vector<int> index;
6             std::map<char, Node*> children;
7
8             Node(std::string s, int i)
9             {
10                if(s.length() > 0 ){
11                    children[s[0]] = new Node(s.substr(1),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



Trie Node Implementation

NaryTree.h

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



What if we have more than one string?

Trie Node Implementation

main.cpp

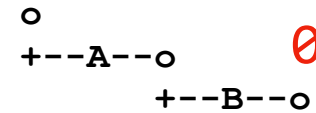
```
1 NaryTree myT;  
2 myTree.print();  
3  
4 myTree.insert("AB",0);  
5 myTree.print();  
6  
7 myTree.insert("ABA",1);  
8 myTree.print();  
9  
10 myTree.insert("ABB",2);  
11 myTree.print();  
12  
13 myTree.insert("BAB",3);  
14 myTree.print();  
15  
16 myTree.insert("BBB",4);  
17 myTree.print();  
18  
19  
20  
21
```

x

Trie Node Implementation

main.cpp

```
1 NaryTree myT;  
2 myTree.print();  
3  
4 myTree.insert("AB",0);  
5 myTree.print();  
6  
7 myTree.insert("ABA",1);  
8 myTree.print();  
9  
10 myTree.insert("ABB",2);  
11 myTree.print();  
12  
13 myTree.insert("BAB",3);  
14 myTree.print();  
15  
16 myTree.insert("BBB",4);  
17 myTree.print();  
18  
19  
20  
21
```

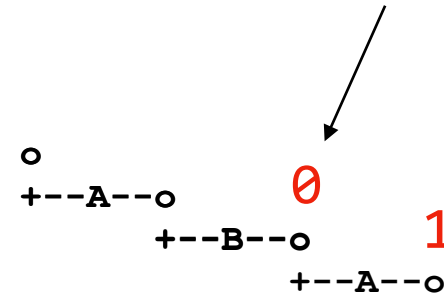


Trie Node Implementation

main.cpp

```
1 NaryTree myT;  
2 myTree.print();  
3  
4 myTree.insert("AB",0);  
5 myTree.print();  
6  
7 myTree.insert("ABA",1);  
8 myTree.print();  
9  
10 myTree.insert("ABB",2);  
11 myTree.print();  
12  
13 myTree.insert("BAB",3);  
14 myTree.print();  
15  
16 myTree.insert("BBB",4);  
17 myTree.print();  
18  
19  
20  
21
```

Former leaf node, still holds value

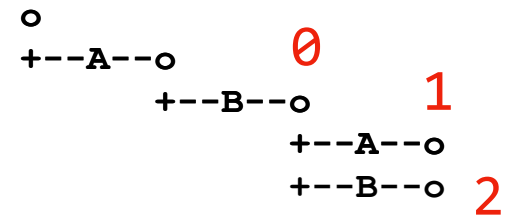


```
struct Node {  
    std::vector<int> index;  
    std::map<char, Node*> children;  
}
```

Trie Node Implementation

main.cpp

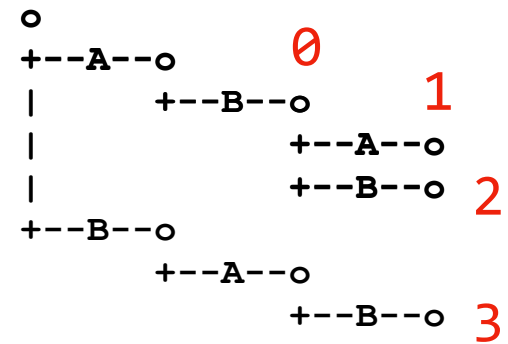
```
1 NaryTree myT;  
2 myTree.print();  
3  
4 myTree.insert("AB",0);  
5 myTree.print();  
6  
7 myTree.insert("ABA",1);  
8 myTree.print();  
9  
10 myTree.insert("ABB",2);  
11 myTree.print();  
12  
13 myTree.insert("BAB",3);  
14 myTree.print();  
15  
16 myTree.insert("BBB",4);  
17 myTree.print();  
18  
19  
20  
21
```



Trie Node Implementation

main.cpp

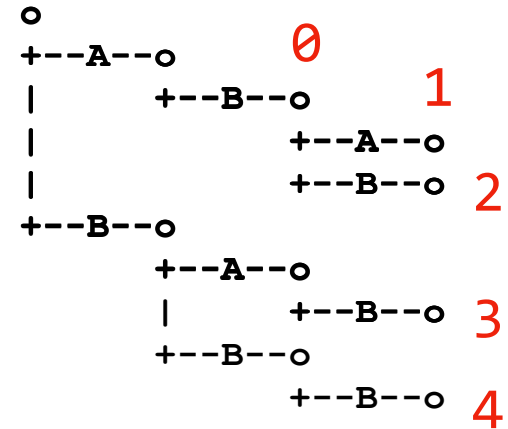
```
1 NaryTree myT;  
2 myTree.print();  
3  
4 myTree.insert("AB",0);  
5 myTree.print();  
6  
7 myTree.insert("ABA",1);  
8 myTree.print();  
9  
10 myTree.insert("ABB",2);  
11 myTree.print();  
12  
13 myTree.insert("BAB",3);  
14 myTree.print();  
15  
16 myTree.insert("BBB",4);  
17 myTree.print();  
18  
19  
20  
21
```



Trie Node Implementation

main.cpp

```
1 NaryTree myT;  
2 myTree.print();  
3  
4 myTree.insert("AB",0);  
5 myTree.print();  
6  
7 myTree.insert("ABA",1);  
8 myTree.print();  
9  
10 myTree.insert("ABB",2);  
11 myTree.print();  
12  
13 myTree.insert("BAB",3);  
14 myTree.print();  
15  
16 myTree.insert("BBB",4);  
17 myTree.print();  
18  
19  
20  
21
```



Trie Node Implementation

NaryTree.h

```
1 void NaryTree::insert(const std::string& s, int i)
2 {
3     insert(root, s, int i);
4 }
5
6 void NaryTree::insert(Node*& node, const std::string & s, int i)
7 {
8     // If we're at a NULL pointer, we make a new Node
9     if (node == NULL) {
10        node = new Node(s, i);
11    } else {
12        if(s.length() > 0 ){
13            if(node->children.count(s[0]) > 0){
14                insert(node->children[s[0]],s.substr(1), i);
15            }else{
16                node->children[s[0]] = new Node(s.substr(1), i);
17            }
18        } else{
19            node->index.push_back(i);
20        }
21    }
22 }
23 }
24
25
```

Trie Node Implementation

main.cpp

```
1 NaryTree myT;  
2  
3 myTree.insert("AB",0);  
4  
5 myTree.insert("AB",2);  
6  
7 myTree.print();  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21
```

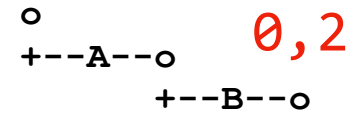
```
o  
+--A--o   ??  
      +--B--o
```

Trie Node Implementation

main.cpp

```
1 NaryTree myT;  
2  
3 myTree.insert("AB",0);  
4  
5 myTree.insert("AB",2);  
6  
7 myTree.print();  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21
```

```
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);  
}
```



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

Assignment 5: a_narytree



Learning Objective:

Store all substrings in a trie using NaryTree implementation

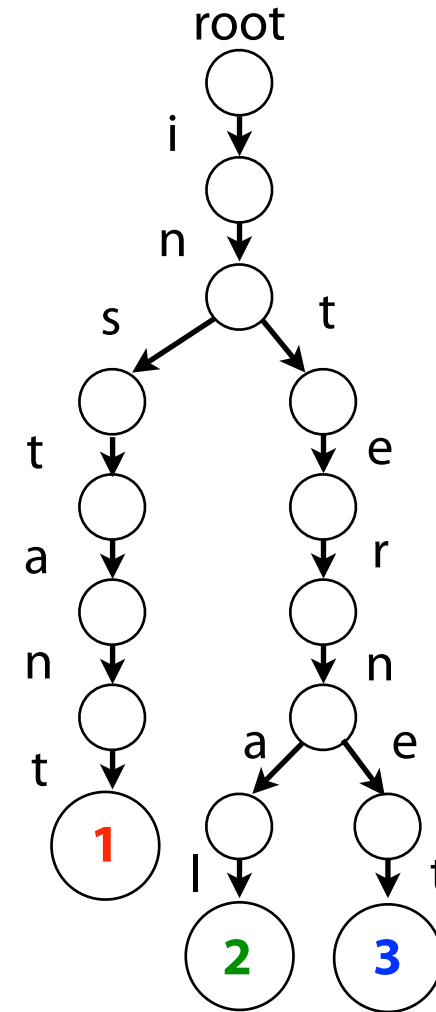
Implement exact pattern matching using this trie

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

Searching a Trie

Given P , search the trie for keys and return values

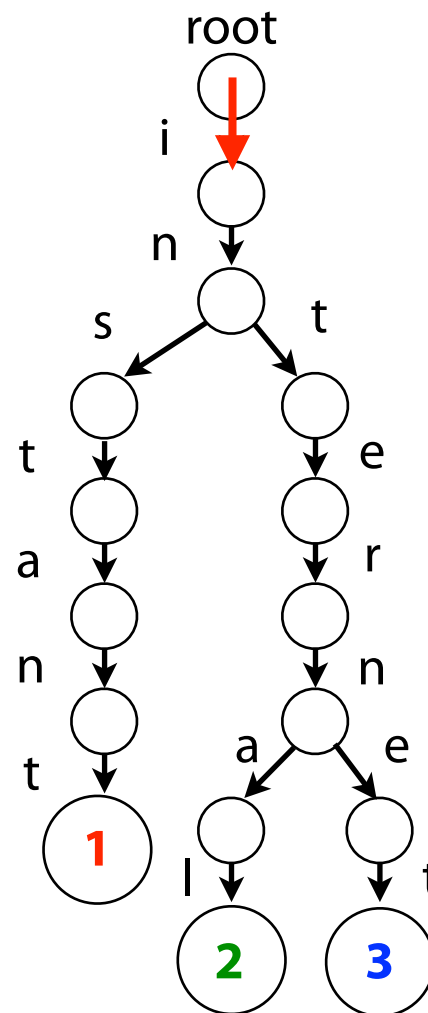
Pattern: `i n f e r`



Searching a Trie

Given P , search the trie for keys and return values

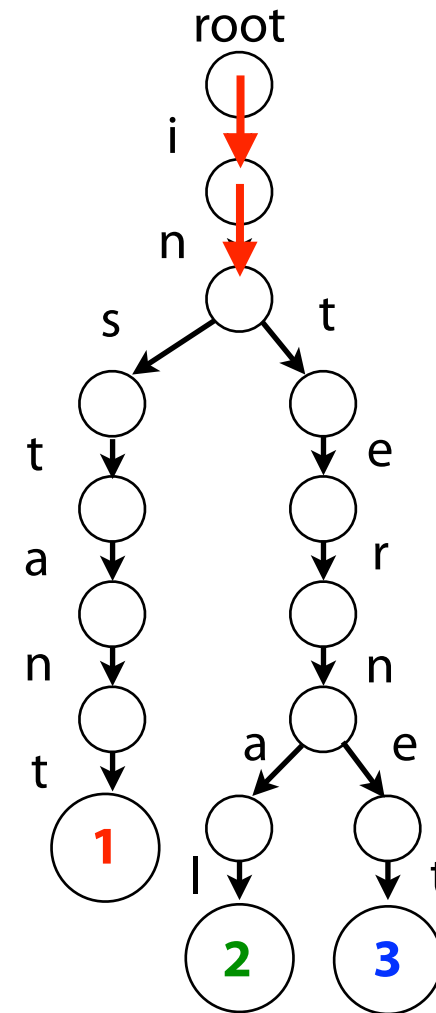
Pattern: `i n f e r`
`i n f e r`



Searching a Trie

Given P , search the trie for keys and return values

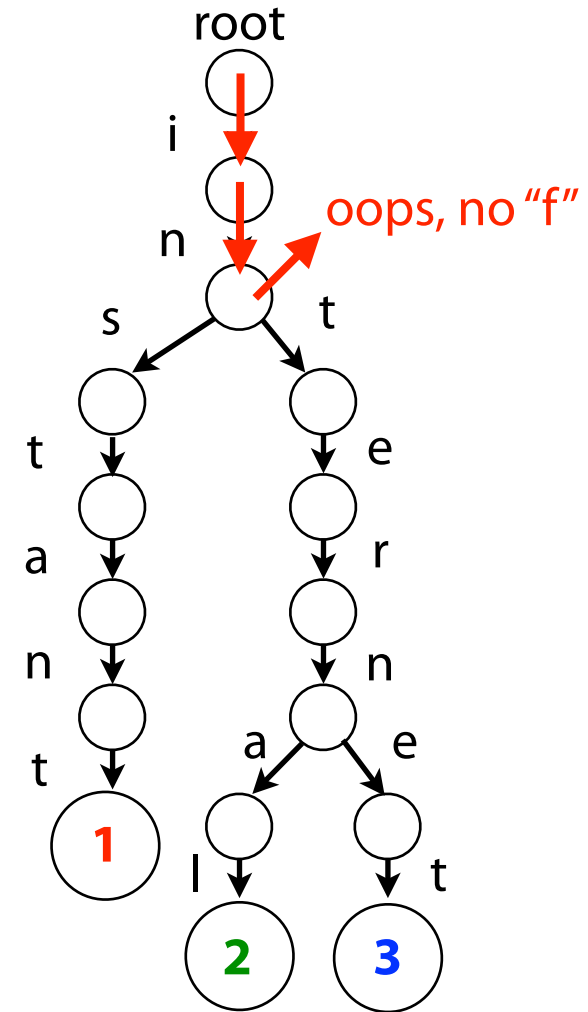
Pattern: `i n f e r`
`i n f e r`
`i n f e r`



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

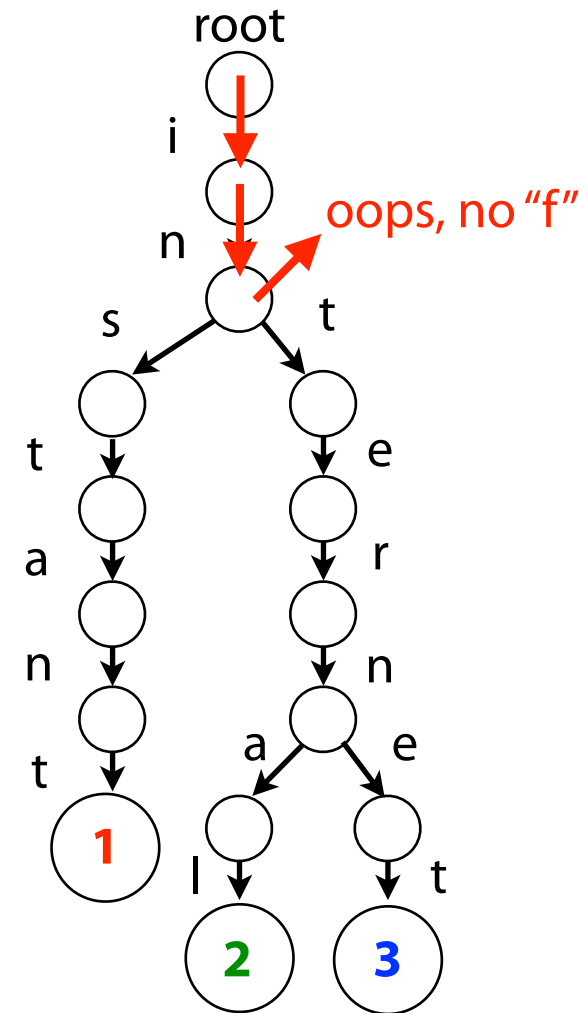


Searching a Trie

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*:



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

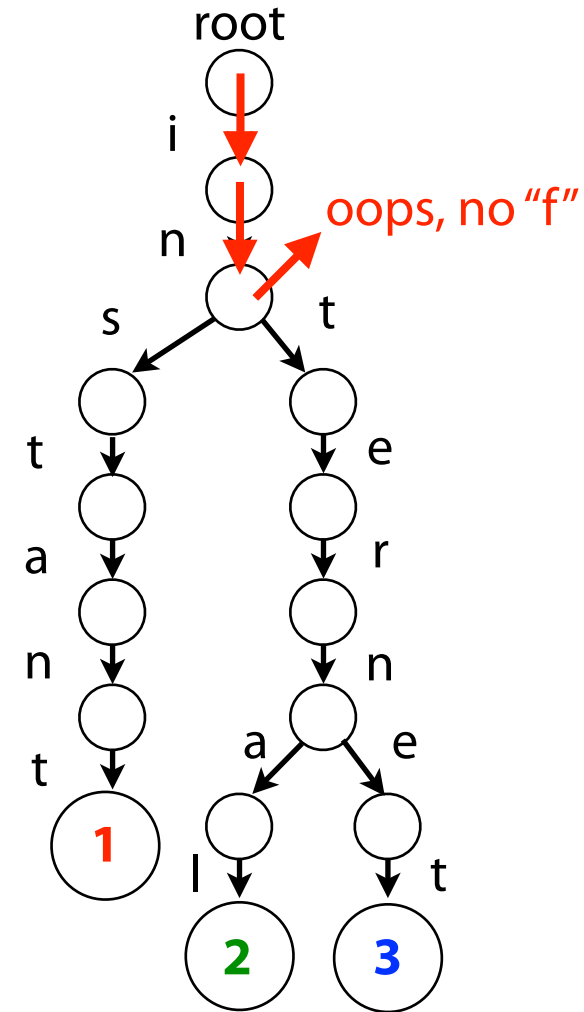
Lets break that down using *recursion*:

Base Case:

- Pattern is null [Implied we matched all characters]
- Front character doesn't exist as child edge

Recursive Step:

- Search for pattern minus front letter in front letter subtree

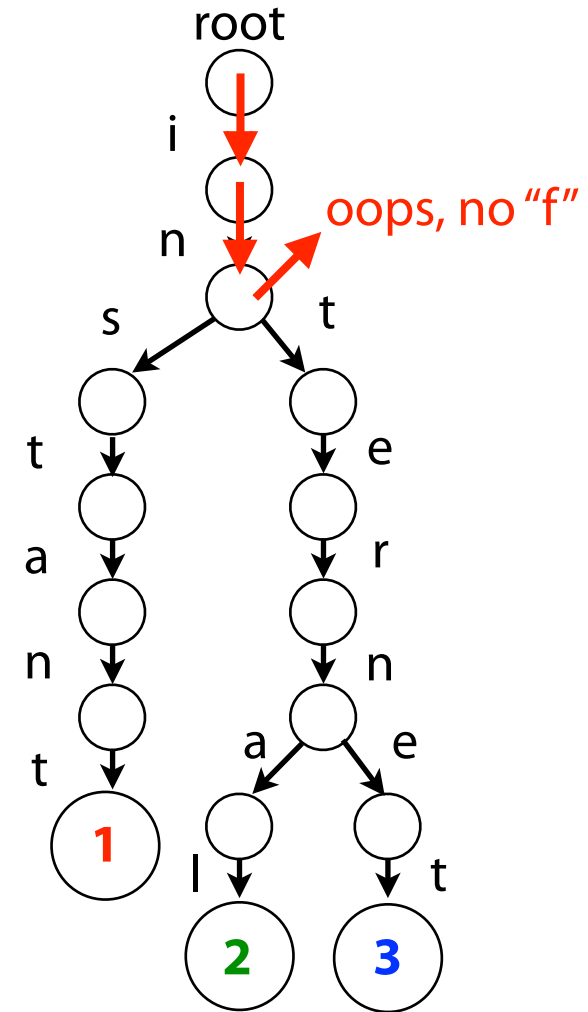


Searching a Trie

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 *pseudocode*:



Searching a Trie

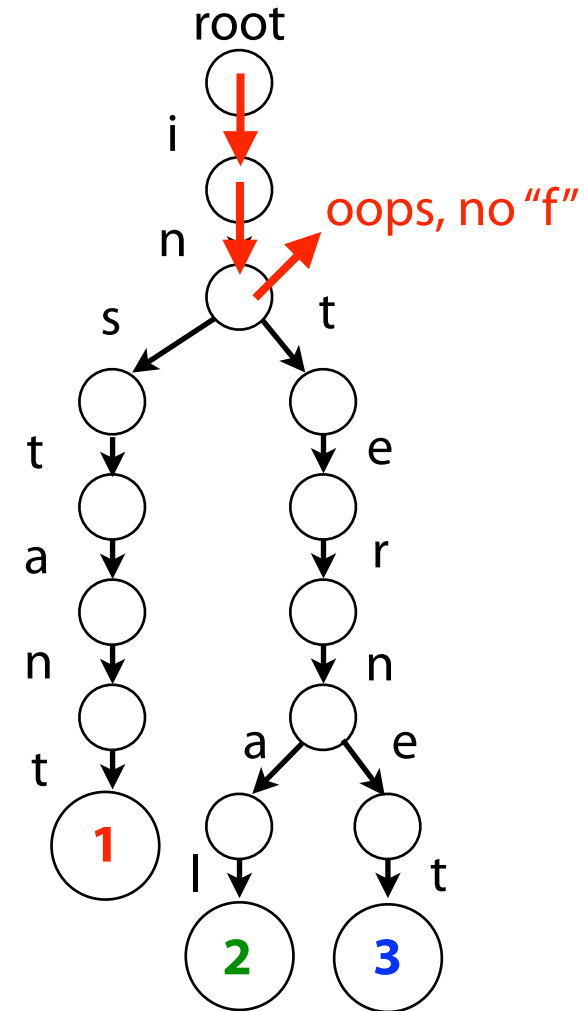
Given P , search the trie for keys and return values

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 *pseudocode*:

Starting at root:

- (1) Try to match front character
- (2) If match, recurse 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

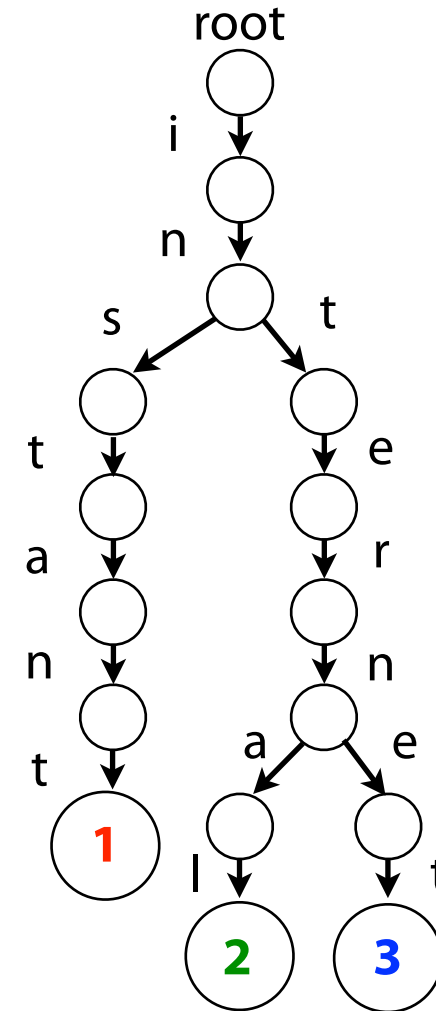
Given P , search the trie for keys and return values

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

Lets break that down using *pseudocode*:

Starting at root:

- (1) Try to match front character
- (2) If match, recurse 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

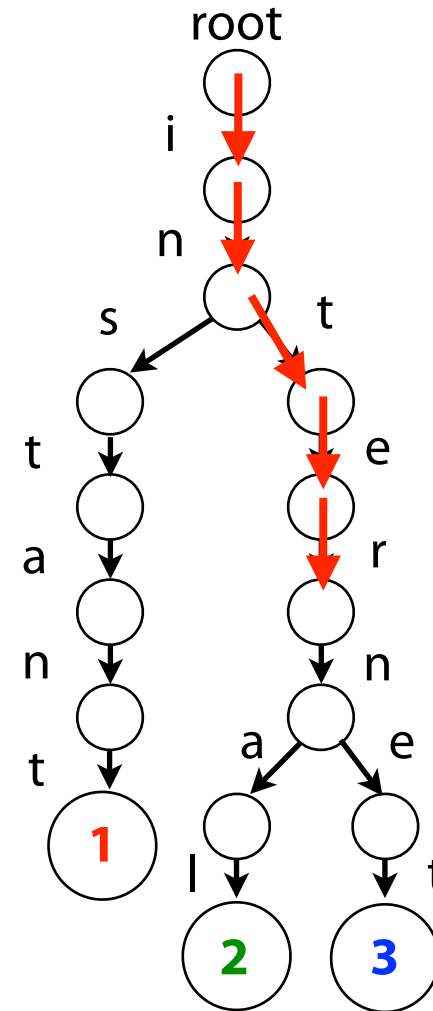
Given P , search the trie for keys and return values

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

Lets break that down using *pseudocode*:

Starting at root:

- (1) Try to match front character
- (2) If match, recurse 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

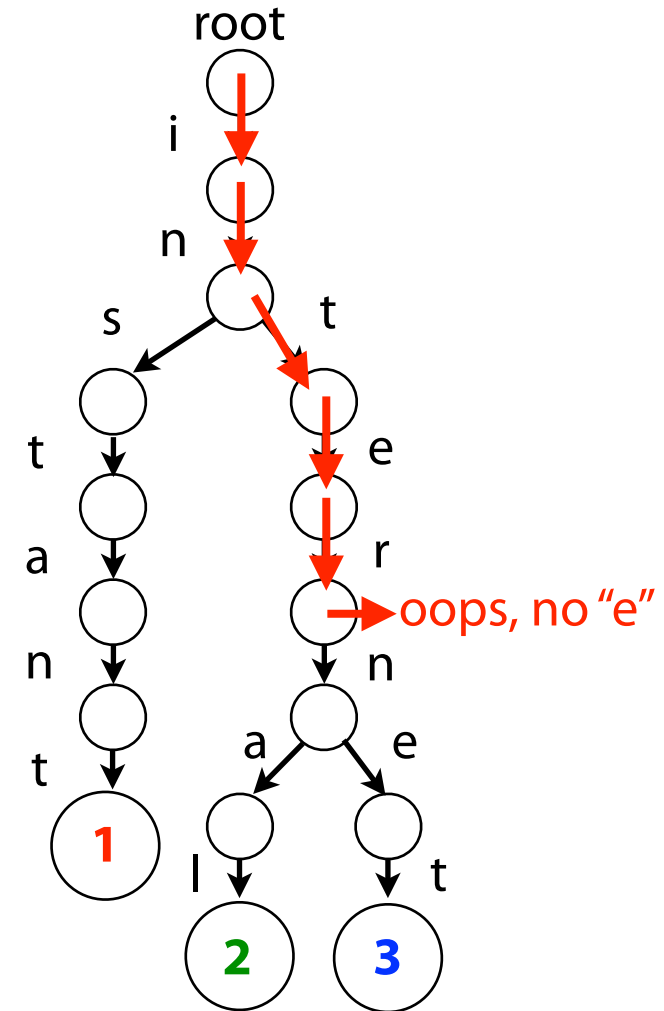
Given P , search the trie for keys and return values

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 *pseudocode*:

Starting at root:

- (1) Try to match front character
- (2) If match, recurse 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

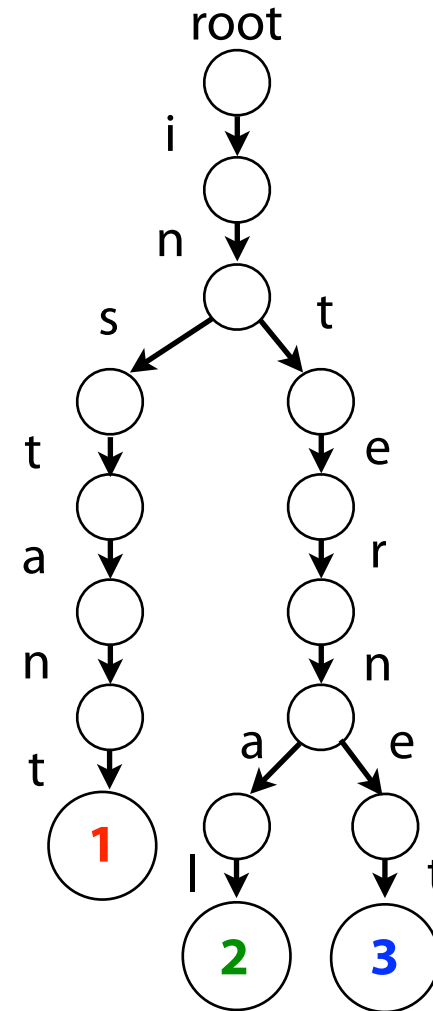
Given P , search the trie for keys and return values

Pattern: `i n s t a`

Lets break that down using *pseudocode*:

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

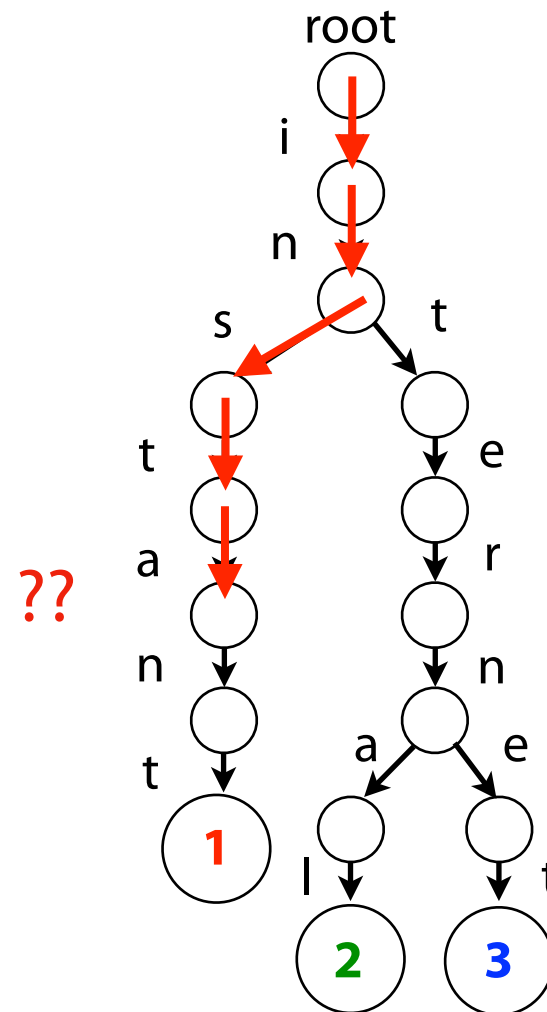
Given P , search the trie for keys and return values

Pattern: `i n s t a`
`i n s t a`

Lets break that down using *pseudocode*:

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

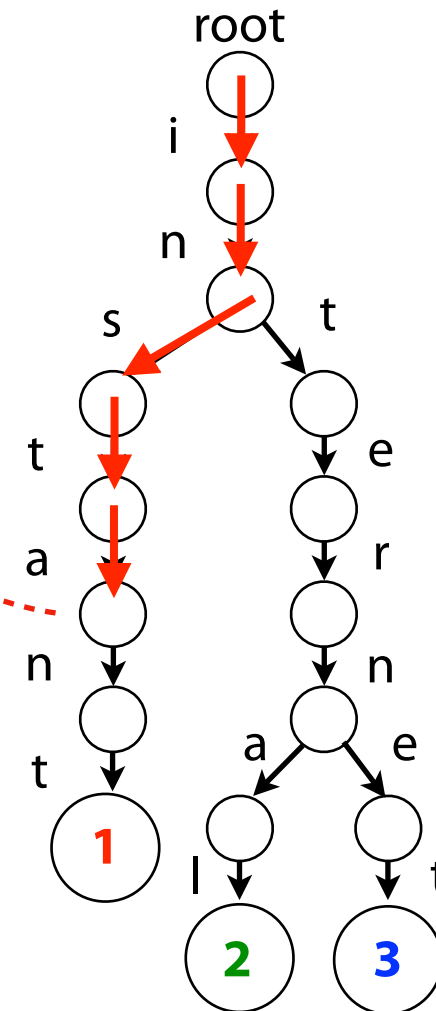
Pattern: `i n s t a`
`i n s t a`

Lets break that down using *pseudocode*:

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!

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



String indexing with Tries

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

Keys:

Values:

i n s t a n t	1
i n t e r n a l	2
i n t e r n e t	3

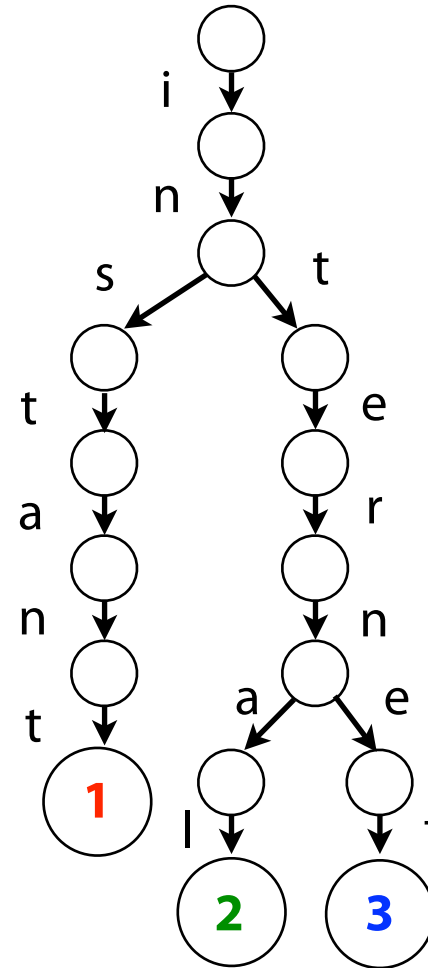
The trie is structured such that:

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: i n s t a

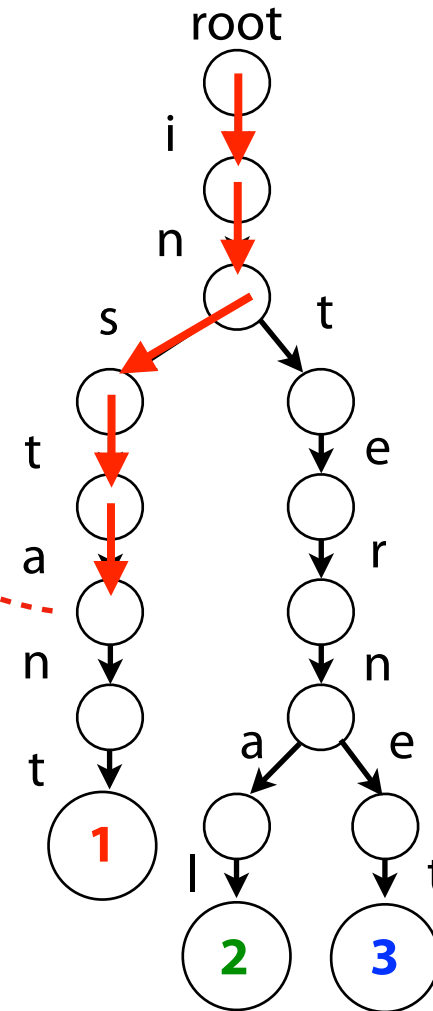
i n s t a

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

Lets break that down using *pseudocode*:

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!





Searching a Trie

Given P , search the trie for keys and return values

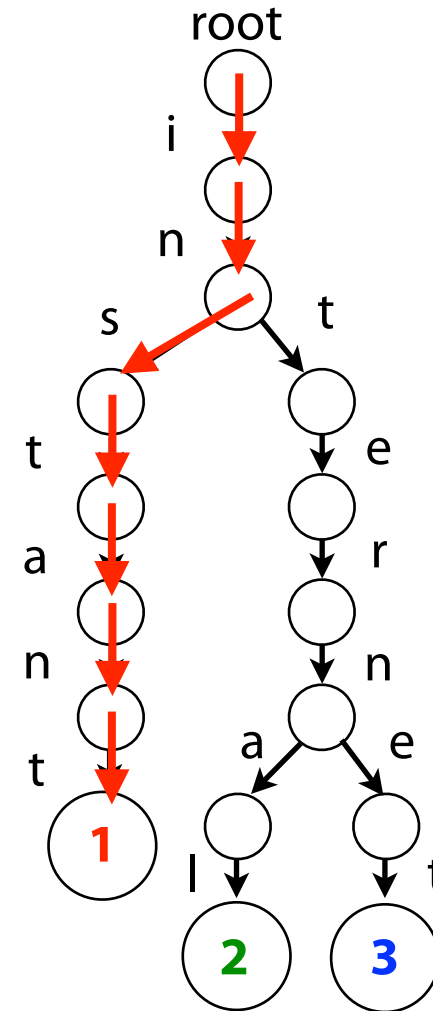
Pattern: `i n s t a n t`

`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

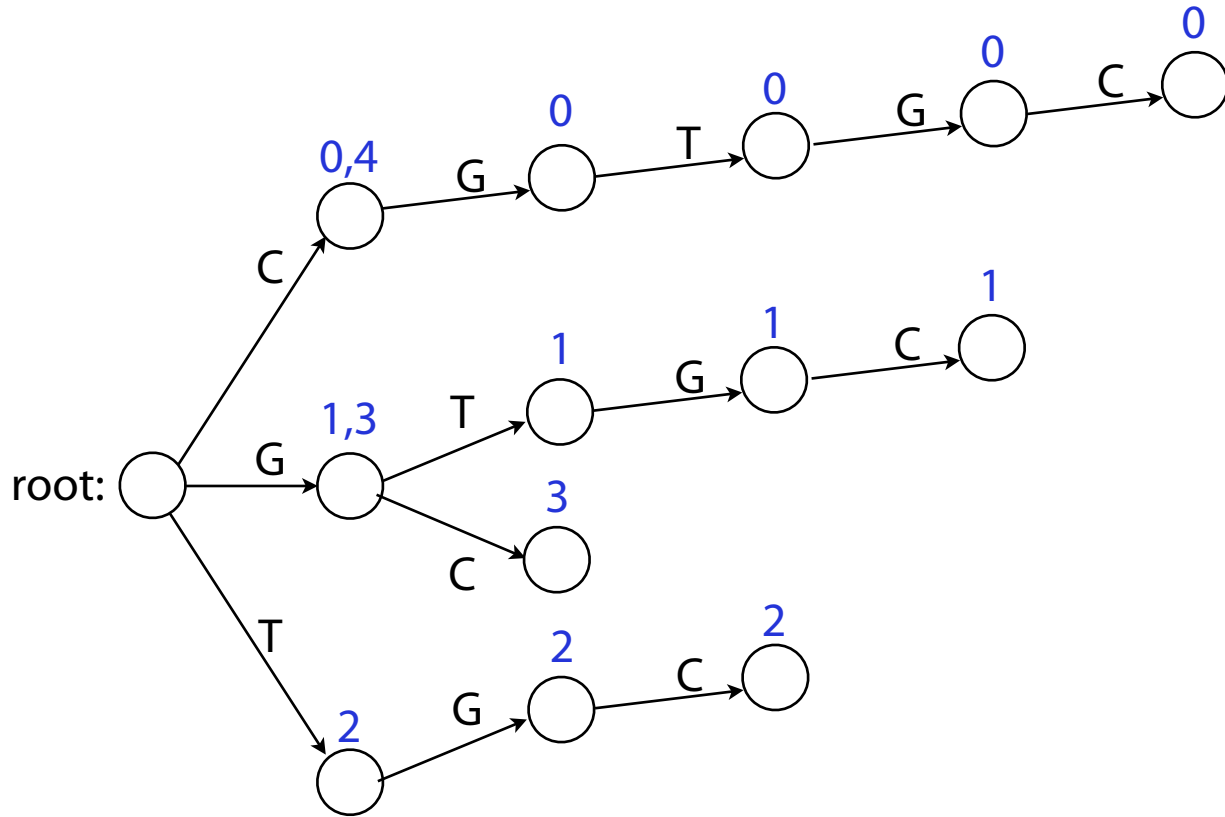
Implement exact pattern matching using this trie

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

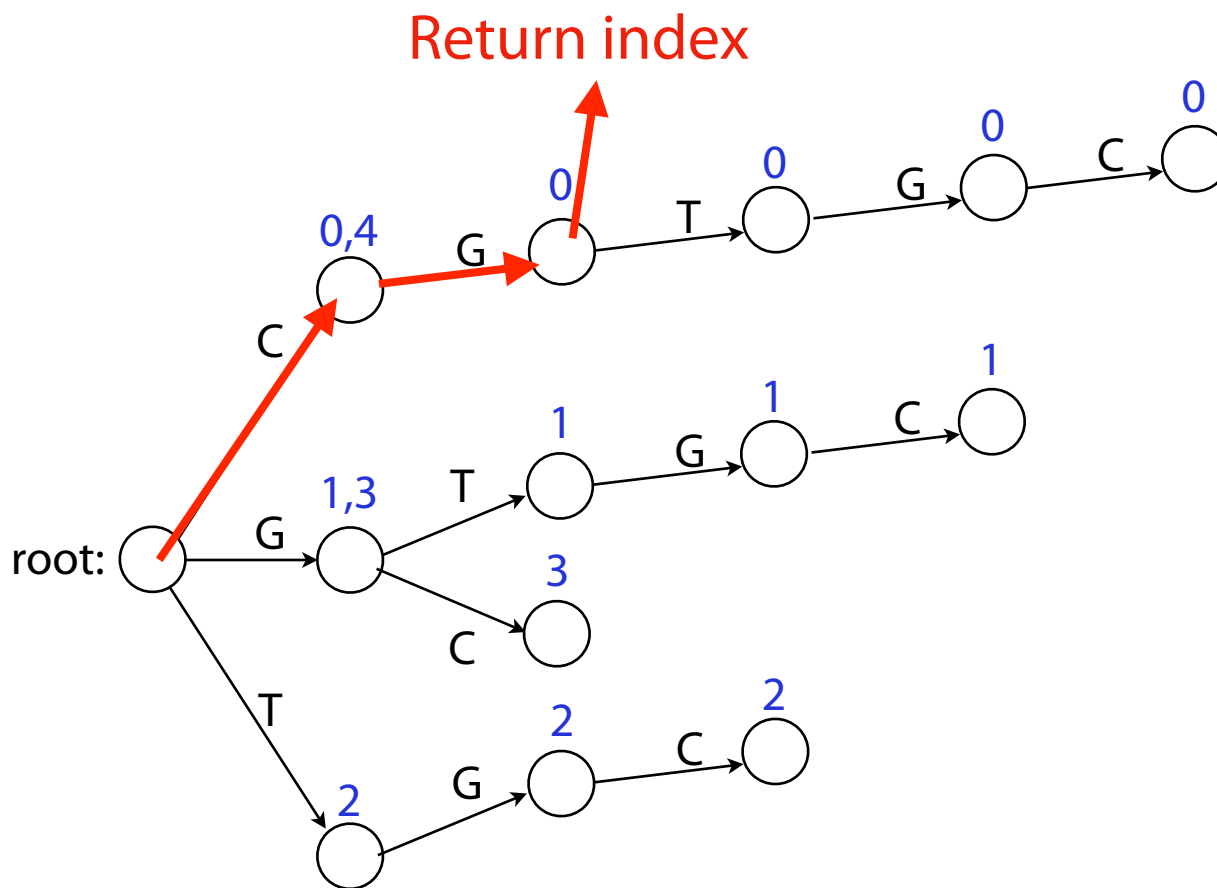
Key	Value
C	0
G	1
T	2
G	3
C	4
CG	0
GT	1
TG	2
...	...



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
...	...

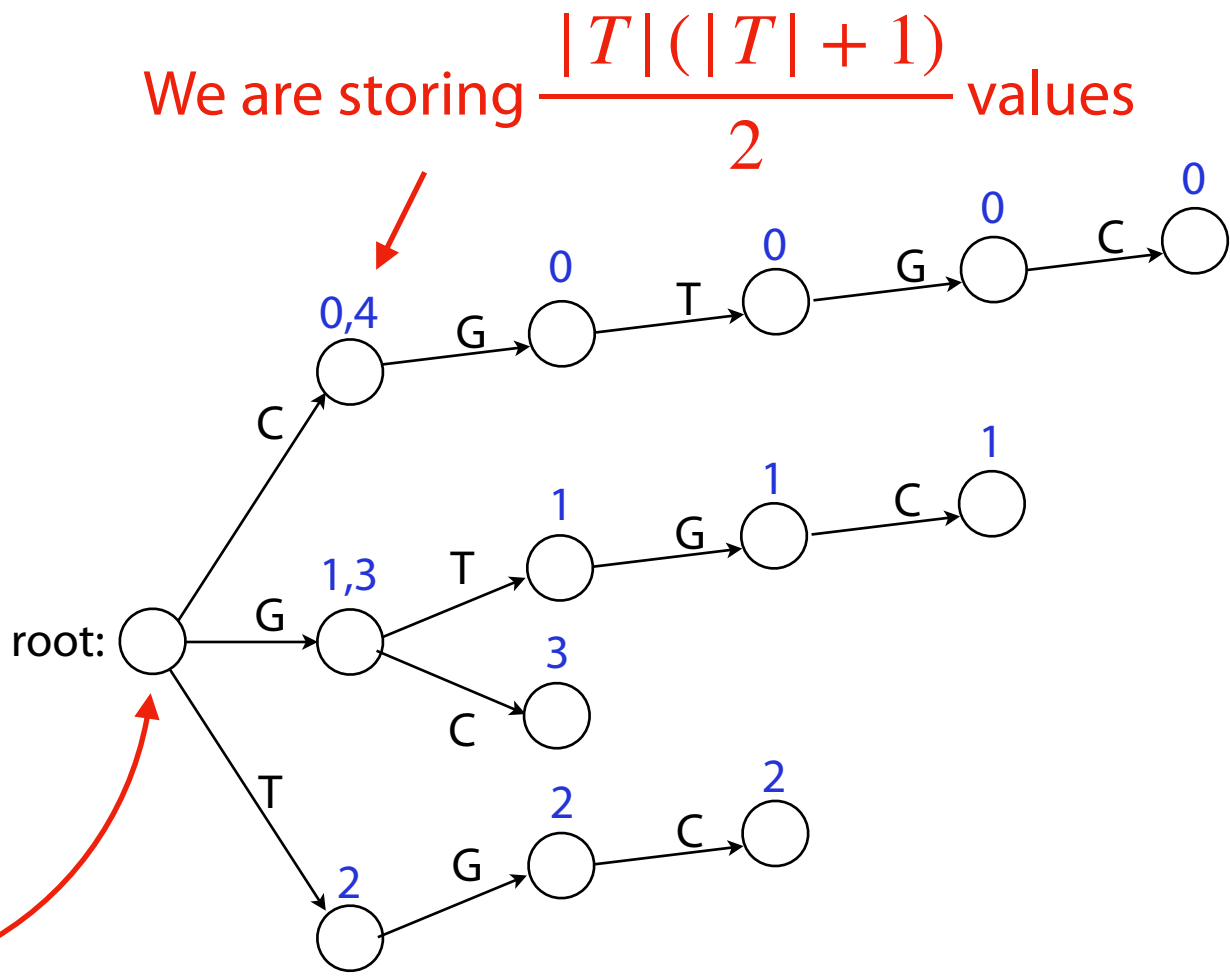


We can do exact pattern matching in $O(P)$ time!

Preprocessing for exact pattern matching

T: C G T G C

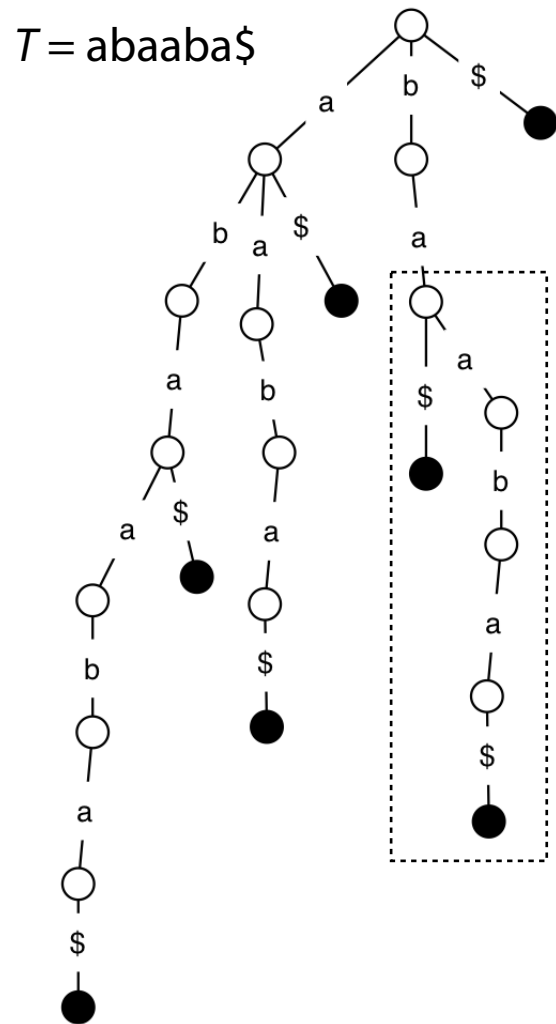
Key	Value
C	0
G	1
T	2
G	3
C	4
CG	0
GT	1
TG	2
...	...



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

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

Preprocessing for exact pattern matching



If only there was a way...

to insert fewer strings

to store fewer values

to be even more efficient!

