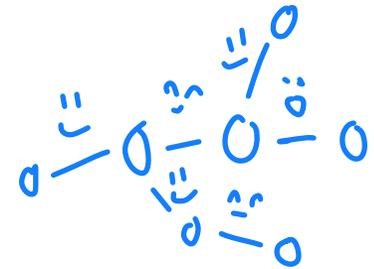# String Algorithms and Data Structures

Tries

CS 199-225
Brad Solomon

February 23, 2026
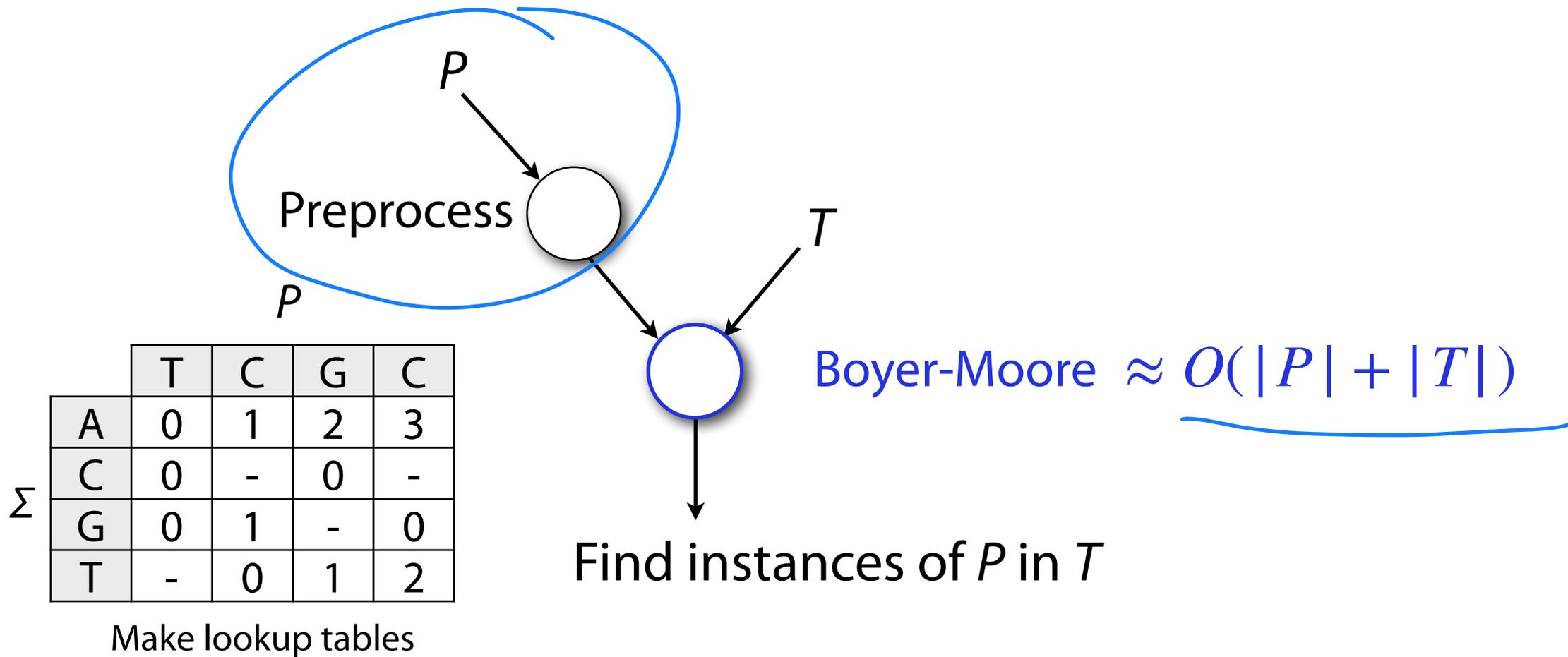
UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN

Department of Computer Science

Exact pattern matching w/ Boyer-Moore

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

Preprocess

P

T

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

Find instances of *P* in *T*

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

P

Make lookup tables

Preprocessing: Live chat streams

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

Preprocessing: Live chat streams

Known queries

Many text entries

!@#$!%

Thanks for the help!

I don't understand that…

Preprocess

You are a !@#$!% teacher

Introducing
AutoMod 2.0
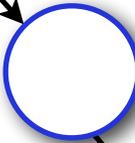
Boyer-Moore
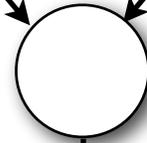
Find instances of *P* in *T*

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

Exact pattern matching *w/ indexing*

Conventionally $T >> P$:

Some large dataset

Many Possible queries

T

P_1 P_2
P_3 P_4

Preprocess (index)

Search Index

Find instances of P in T
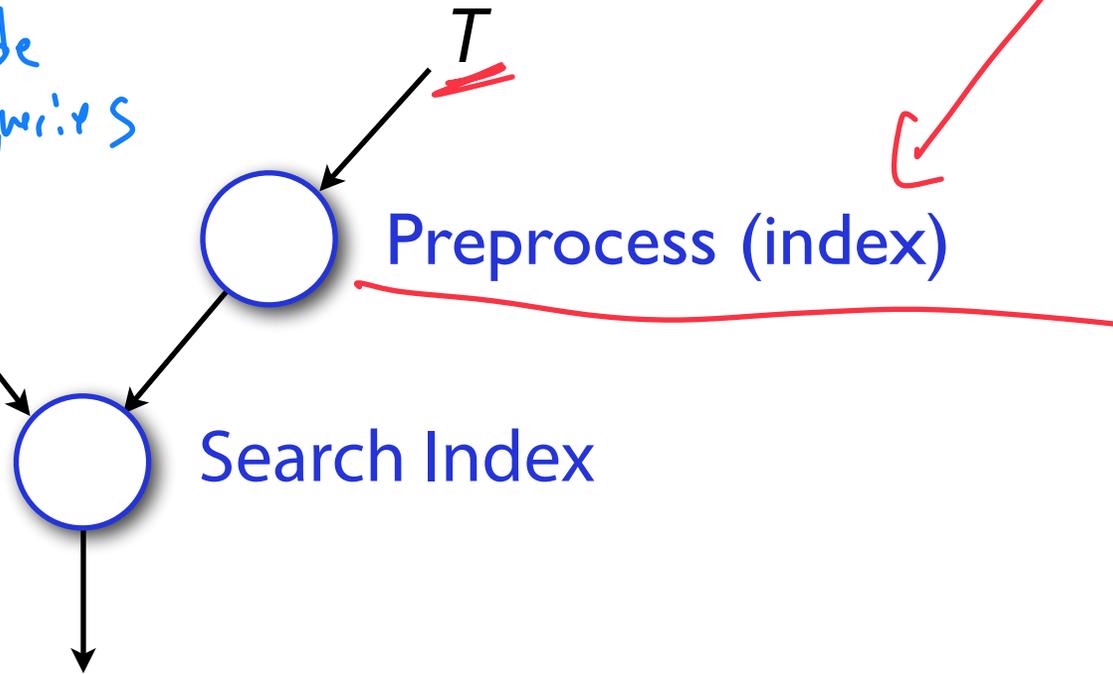
Amortize cost of preprocessing T over many P

Preprocessing: Libraries

Preprocessing: Libraries

preprocess

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

Preprocessing: Libraries
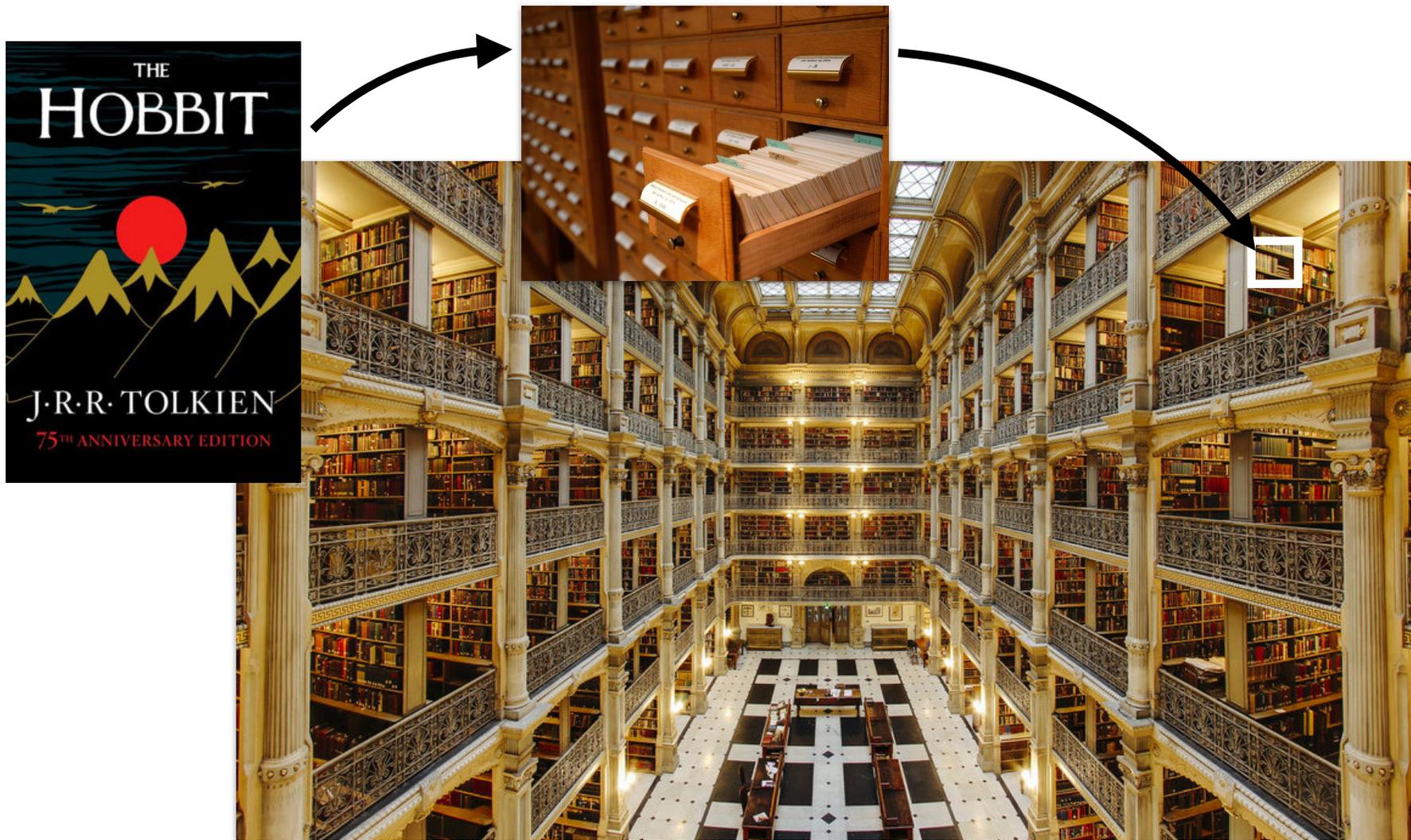
Preprocess the library by *indexing* all the books

Preprocessing: Libraries

THE HOBBIT

Aardvarks Anonymous

By Jim Realman

R.L. STINE
Goosebumps
DON'T GO TO SLEEP
SCHOLASTIC

List of all library books

Preprocess (index)

Search Index

Find instances of *P* in *T*
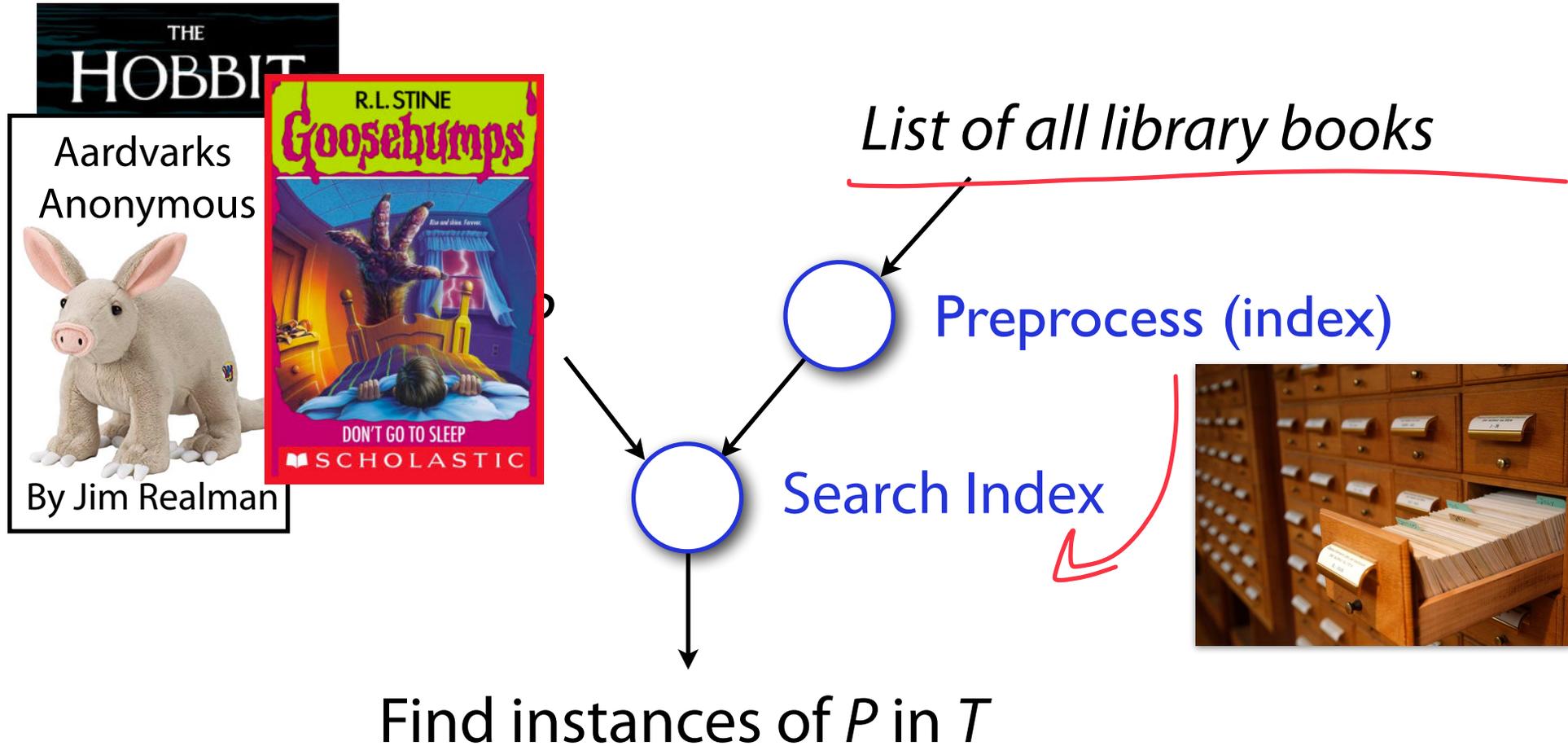
Given full library, built an index once* that is re-used

Preprocessing: Glossaries

Pattern or *text?*

What method of preprocessing is this?

Preprocessing: Glossaries

What method of preprocessing is this?


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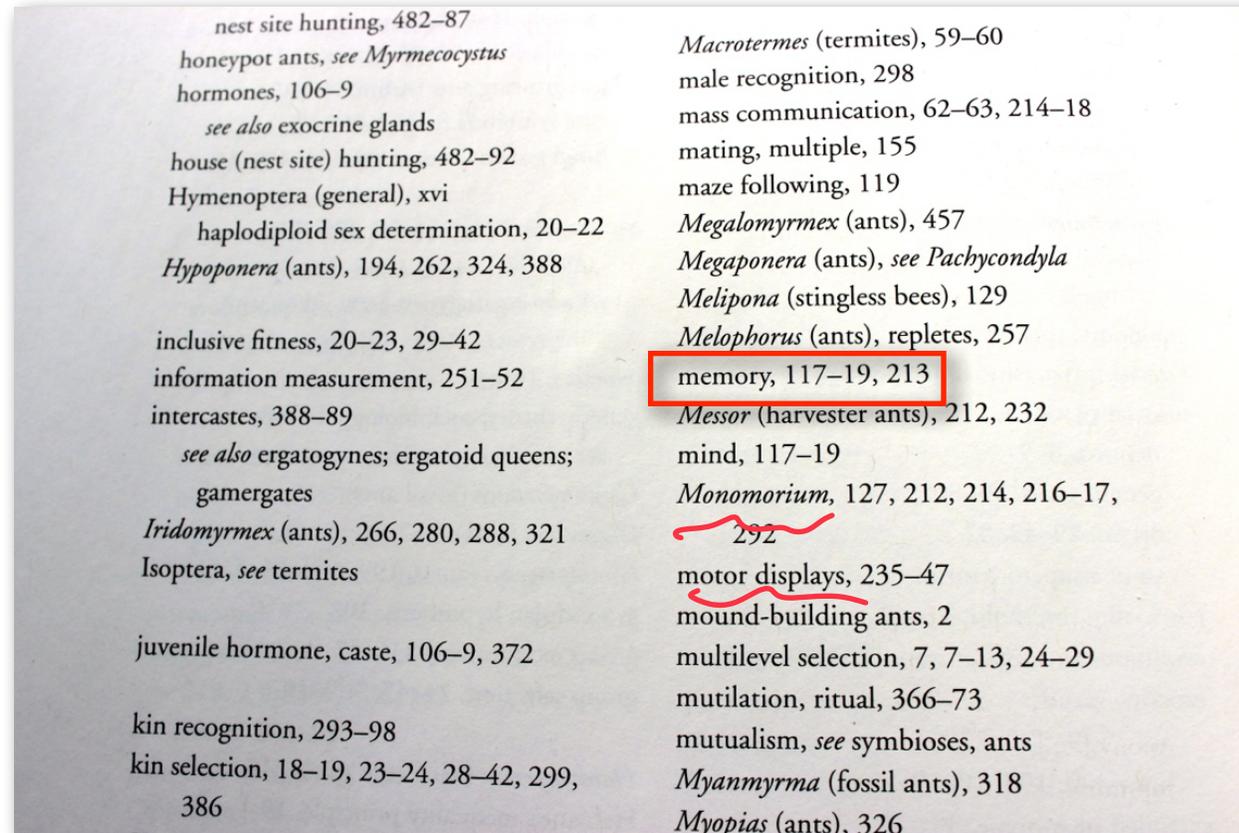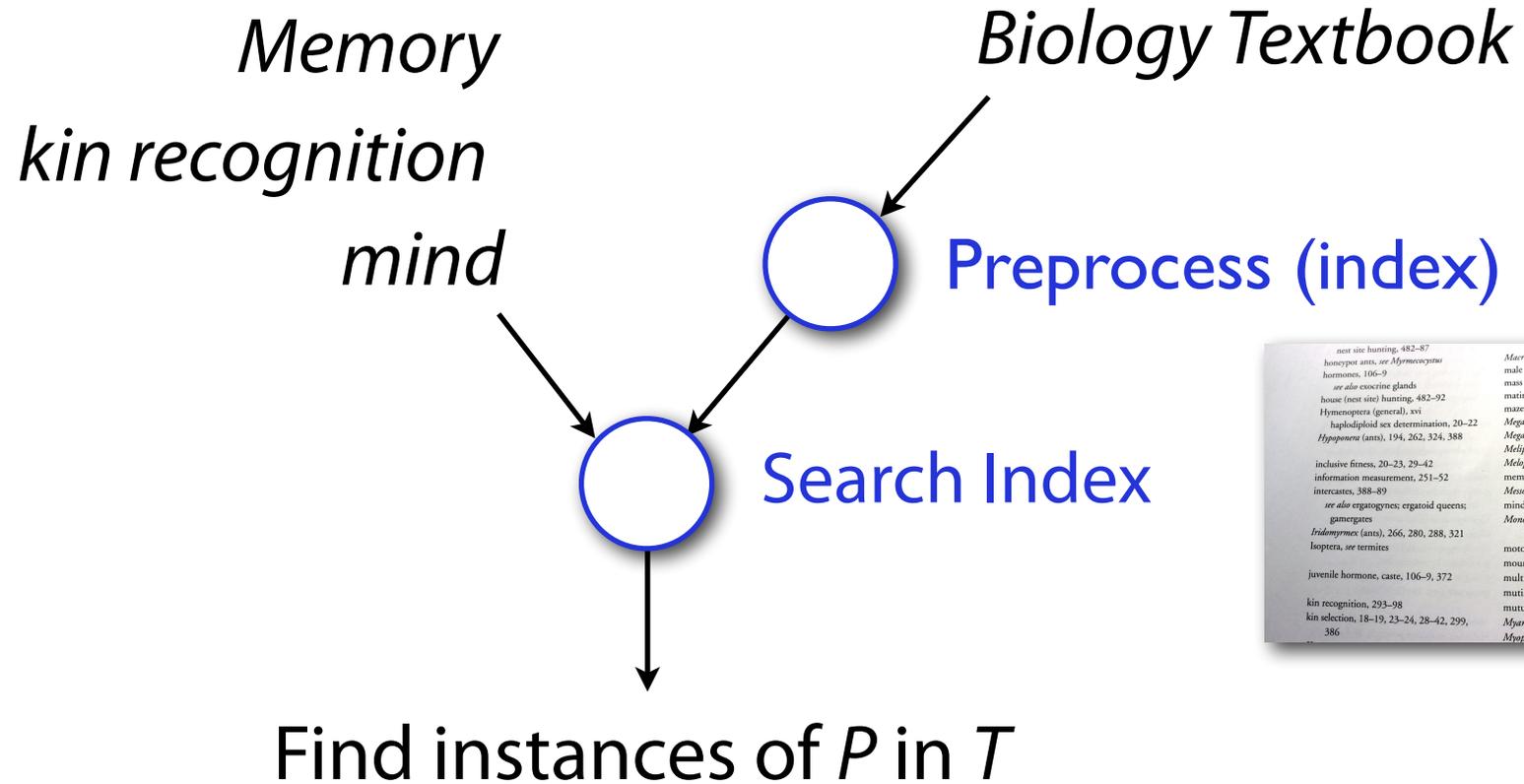
Patterns: Key terms   **Text:** All text in the book

Preprocessing: Glossaries

Memory

kin recognition

mind

Biology Textbook

Preprocess (index)

Search Index

Find instances of *P* in *T*

Glossary built on total contents *T*, useful for multiple *P*

Exact pattern matching *w/ indexing*

$T >> P$

The human genome (~3 billion)

1 Billion reads (~100)

Preprocess (index) $\approx O(|T|)$

Search Index $\approx O(|P|)$

Find instances of *P* in *T*
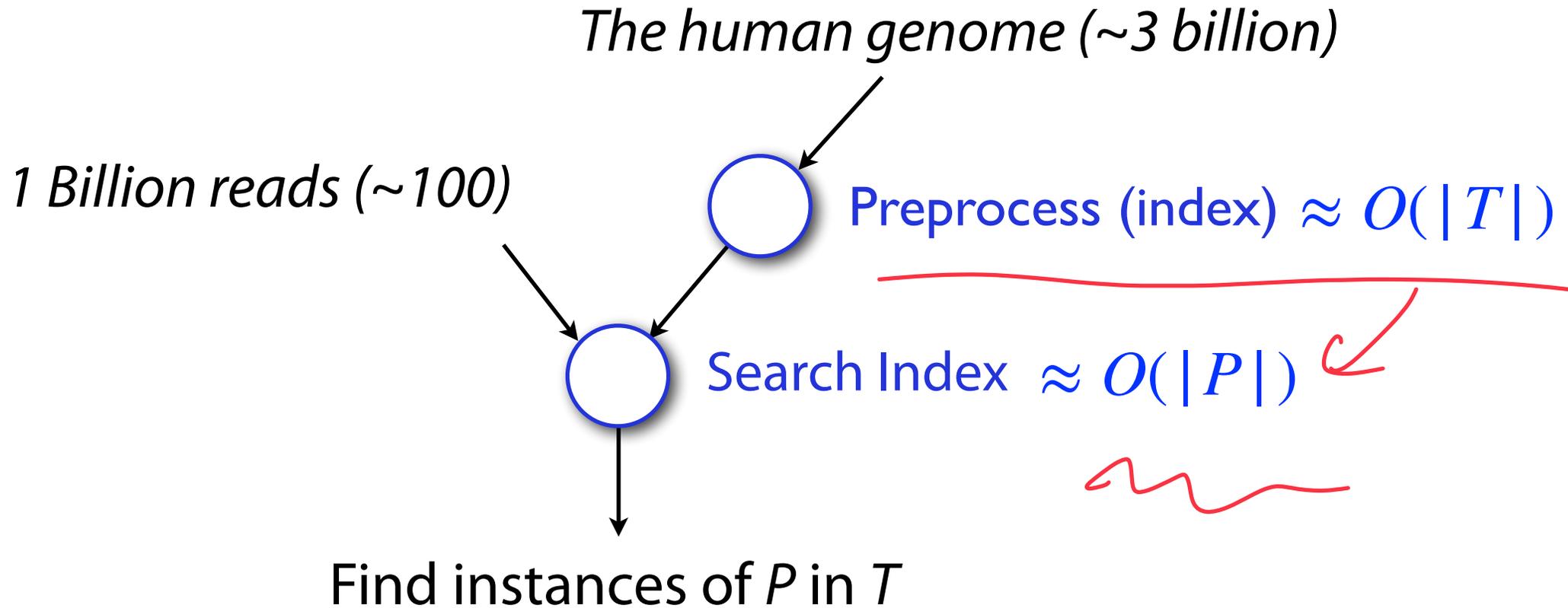
Amortize cost of preprocessing T over many P

Exact pattern matching *w/ indexing*

T

P

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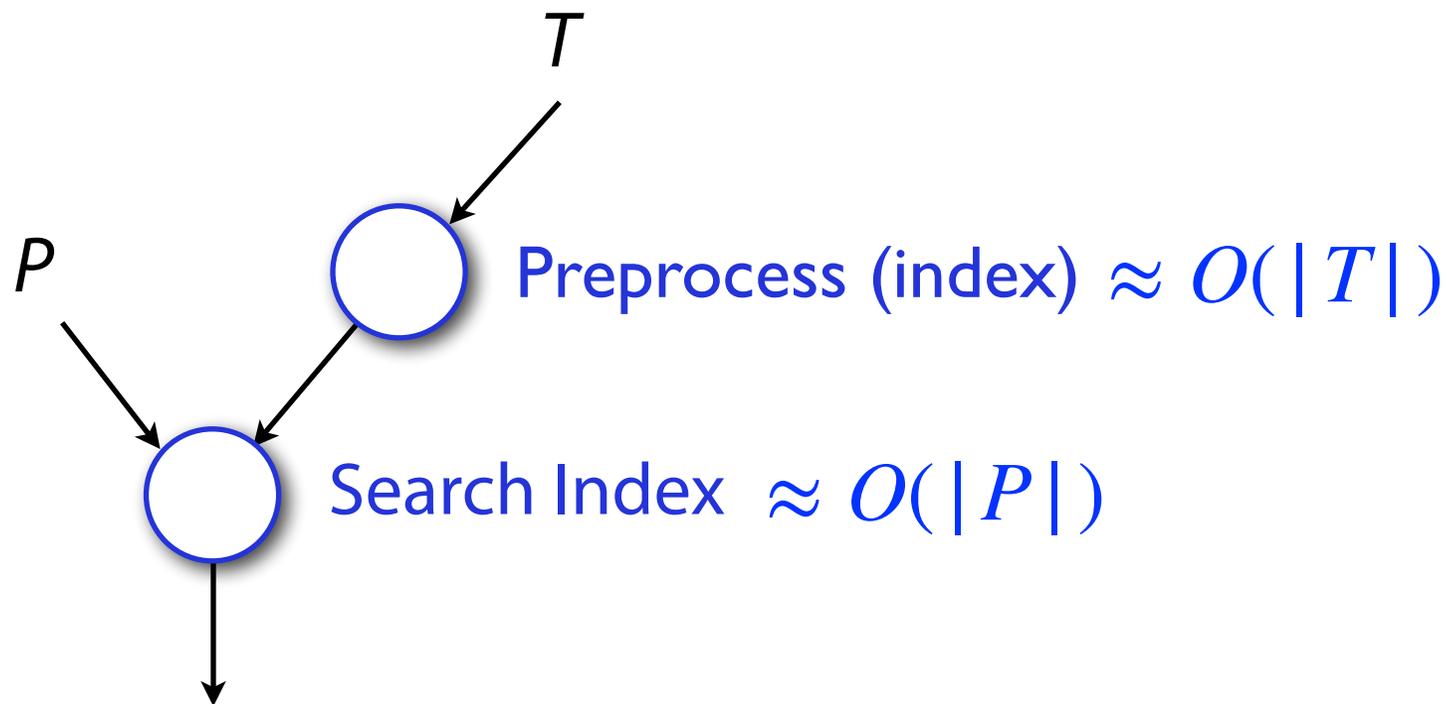
Search Index $\approx O(|P|)$

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: E

Search(*P, T*): nullptr / {-1} / Some sign that 't isnt present

P: C G

Search(*P, T*): pointer to c / index {0}

P: C

Search(*P, T*): {0, 4}

Preprocessing for exact pattern matching

Get every

Substring

Index Position

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
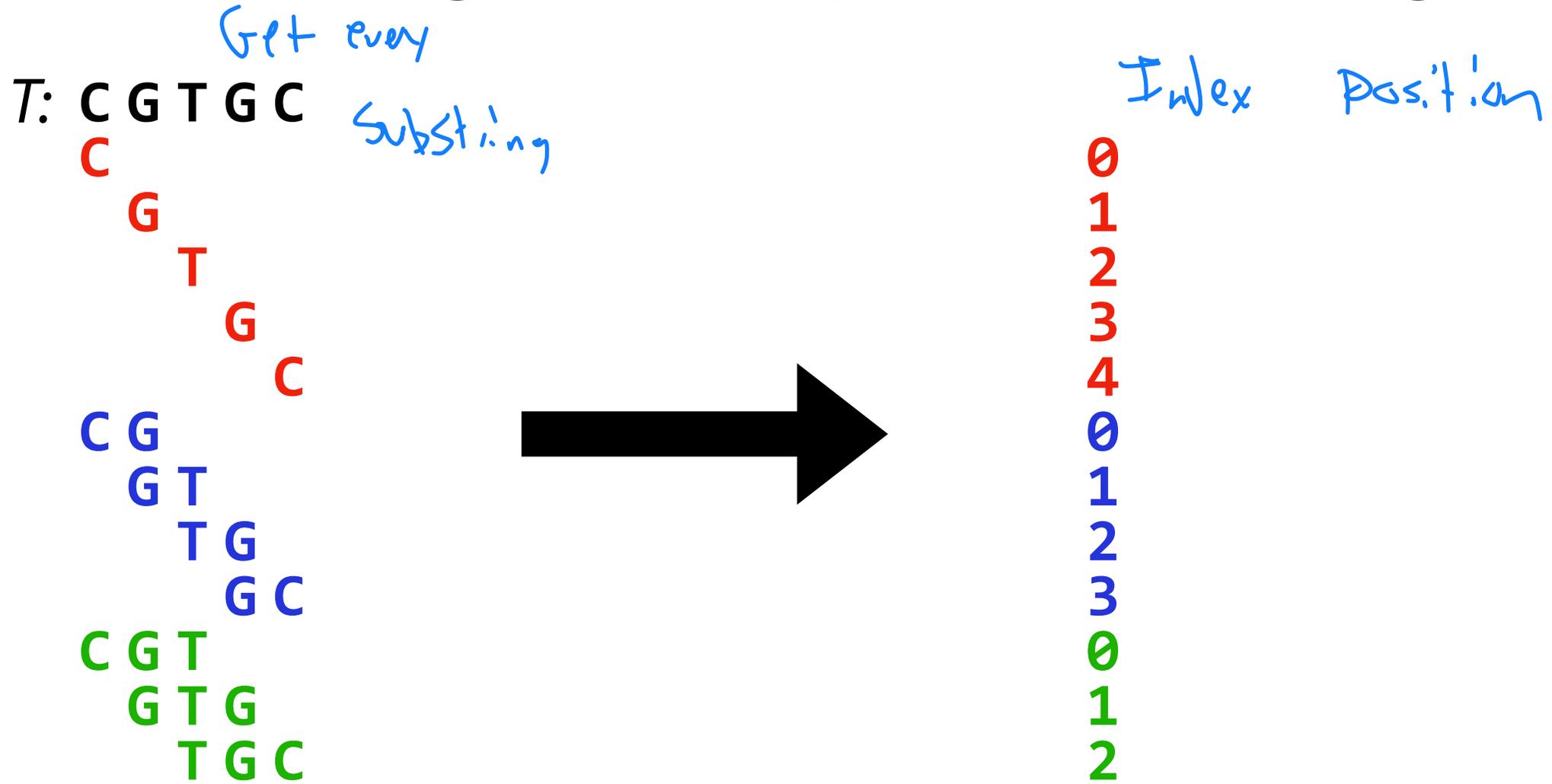
A substring *S*   The position of *S* in *T*
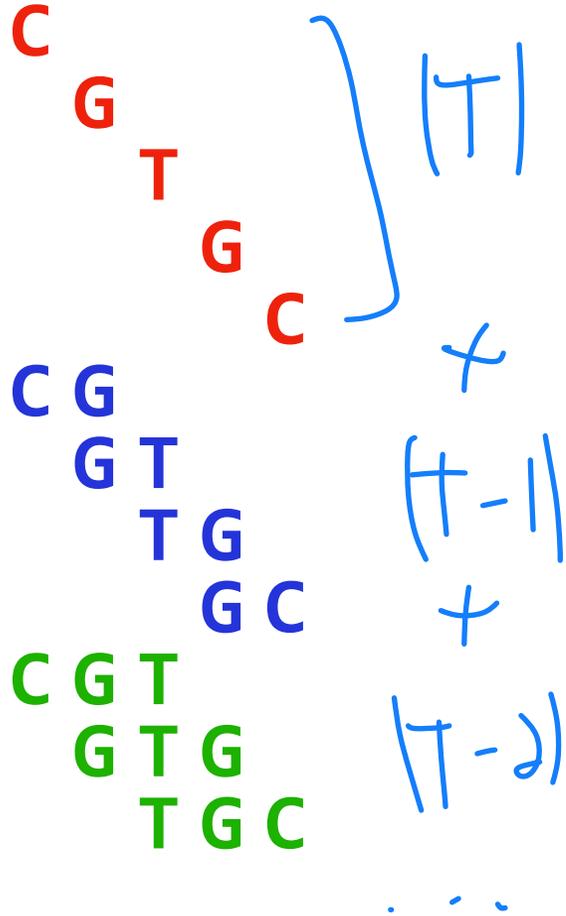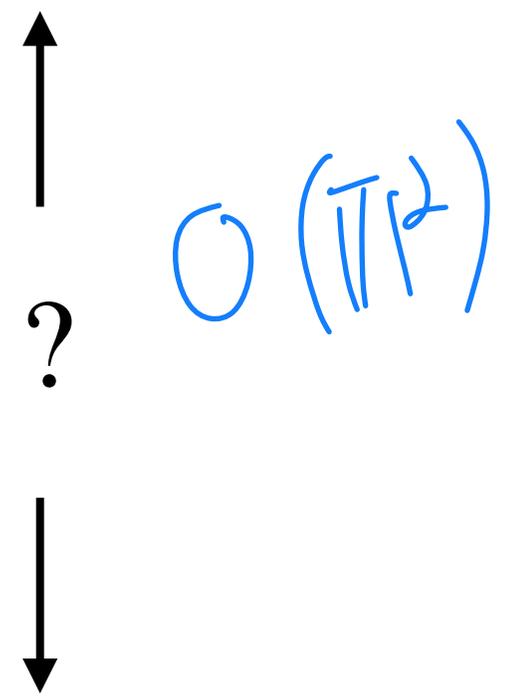
Preprocessing for exact pattern matching

T: **C G T G C**

C
  G
    T
      G
        C

$|T|$

$\times$

C G
 G T
  T G
   G C

$(T-1)$

$\times$

C G T
 G T G
  T G C

$(T-2)$

$\cdots$

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

?

$O(|T|^2)$

Preprocessing for exact pattern matching

T: **C G T G C**

C
  G
    T |T|
      G
        C

C G
  G T
    T G |T-1|
      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
…	…

?

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
…	…

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

$\approx O(T^2)$

Preprocessing for exact pattern matching

Hash Table lookup: $O(1)$ *

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

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

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

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

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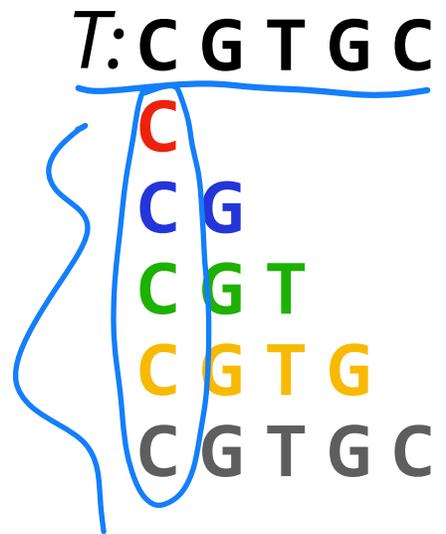
$$\frac{|T|(|T|+1)}{2}$$

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

Preprocessing for exact pattern matching

Strings consist of individual characters!
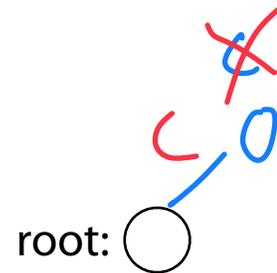
    … and these characters can overlap:

Trie

*by value at **edge***

T: **C G T G C**

C

C G

C G T

C G T G

C G T G C

All Substrings

~~C~~

C   0

root: ◯

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**

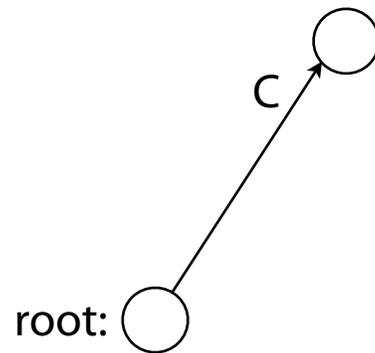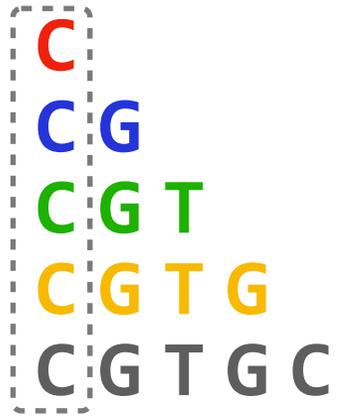**C** **G** **T** **G**

C **G** **T** **G** **C**

root:

c

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

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:

Preprocessing for exact pattern matching

Strings consist of individual characters!
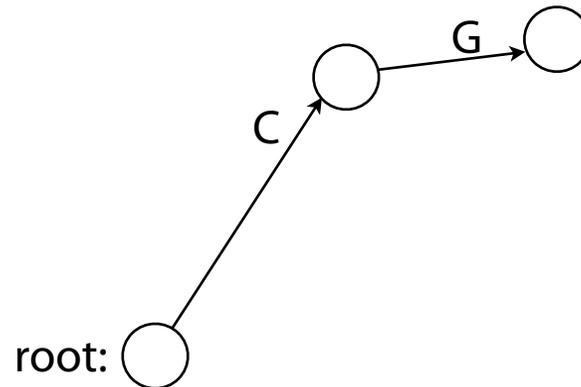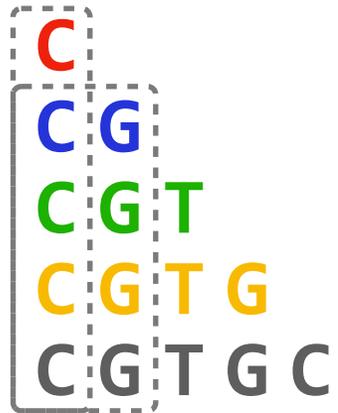
... and these characters can overlap:

Preprocessing for exact pattern matching

Strings consist of individual characters!

… and these characters can overlap:

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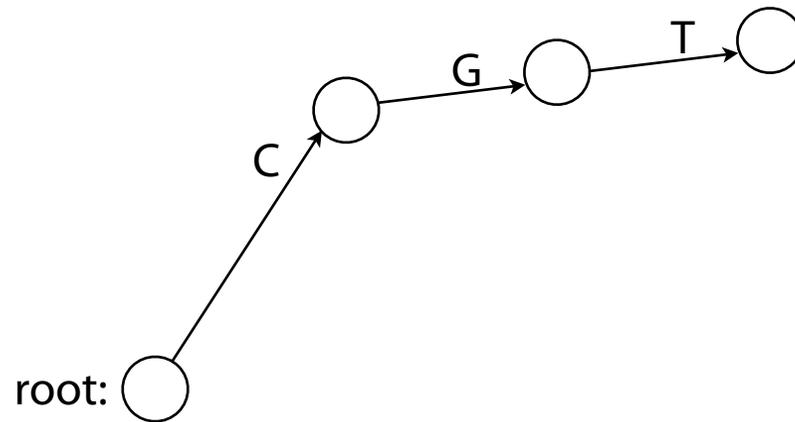**G T G C**

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!
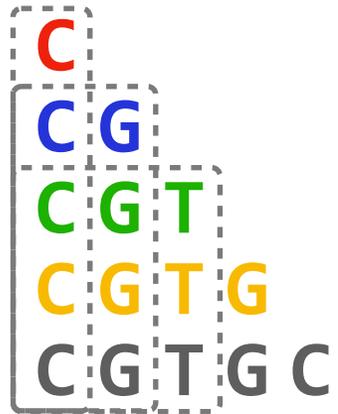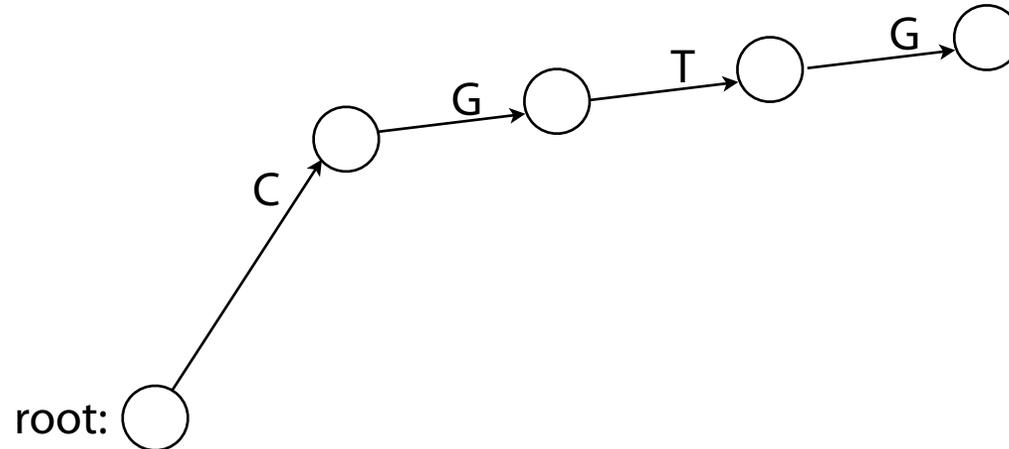
… and these characters can overlap:

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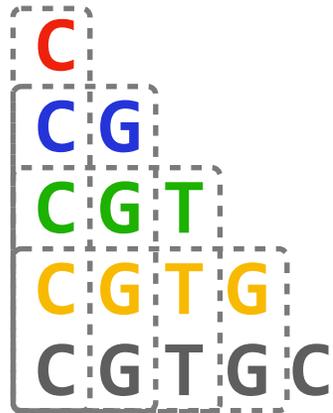
T: **C G T G C**

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:

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!
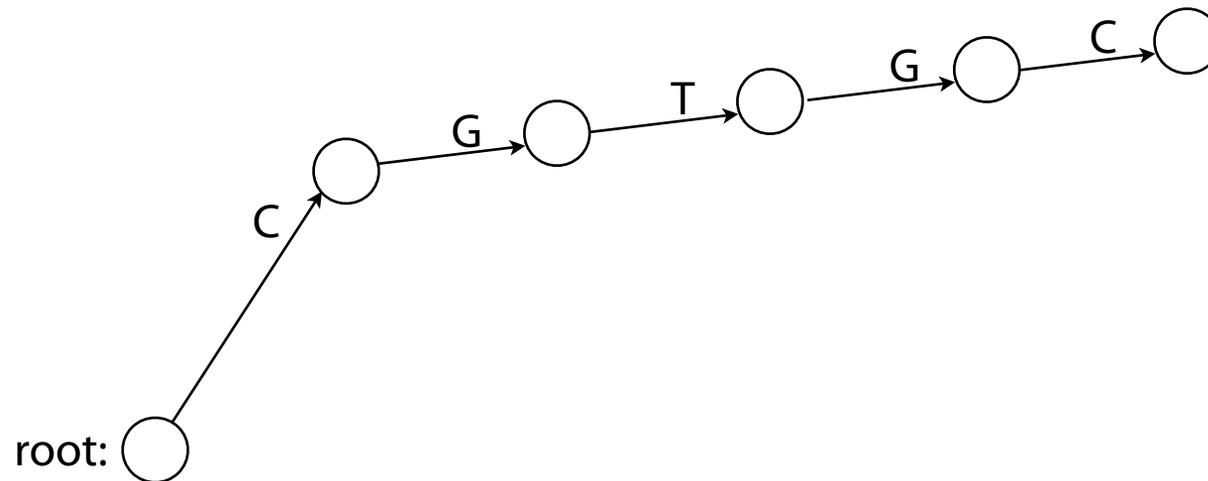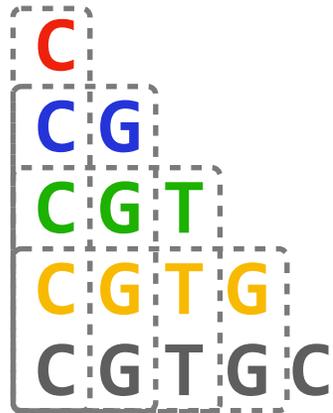
… and these characters can overlap:

T: **C G T G C**

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 $c \in \Sigma$

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 $c \in \Sigma$

root

i

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:

`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:

```
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
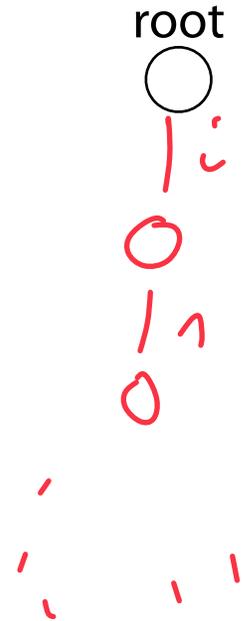
String indexing with Tries

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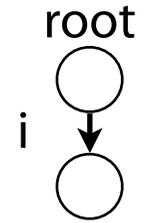
Each edge is labeled with a character $c \in \Sigma$

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:

```
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
```
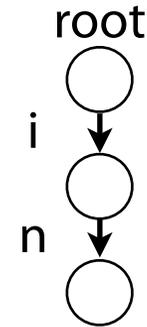
Where should I store the value 1?
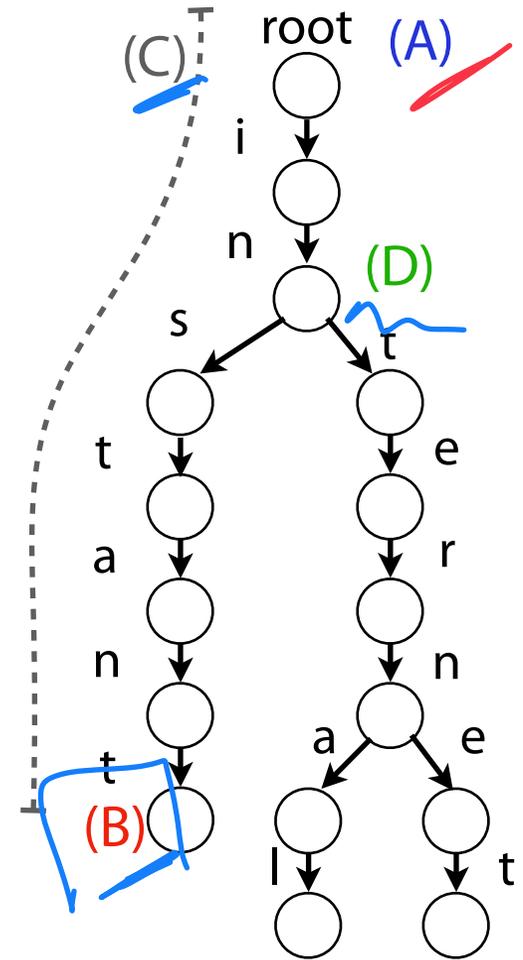
String indexing with Tries

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

Keys: Values:

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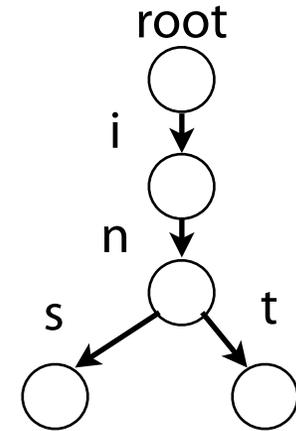
`i n t e r n a l`   2

`i n t e r n e t`   3

Each key's value is stored at the last node in the path
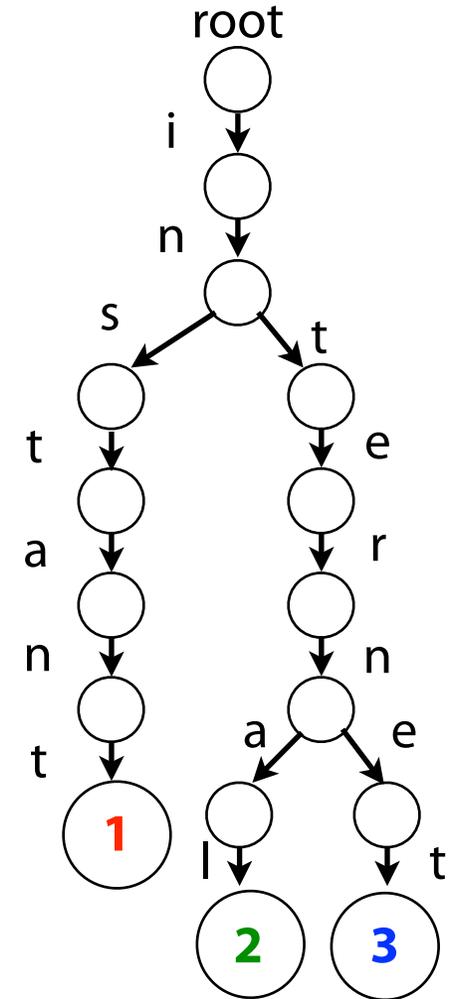
String indexing with Tries

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

Keys: Values:

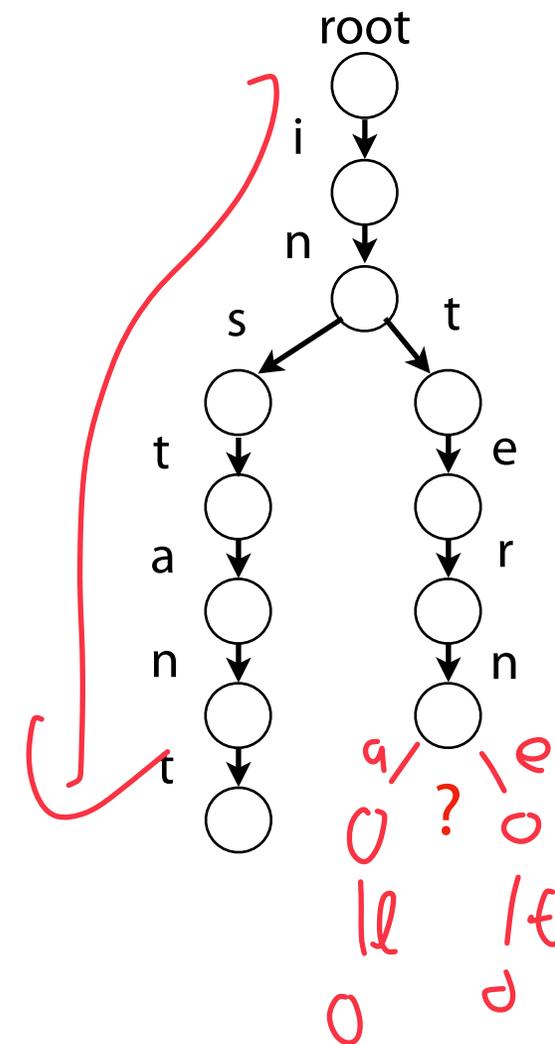`instant`   1

`internal`   2

`internet`   3

`internets` 4

Each key's value is stored at the last node in the path
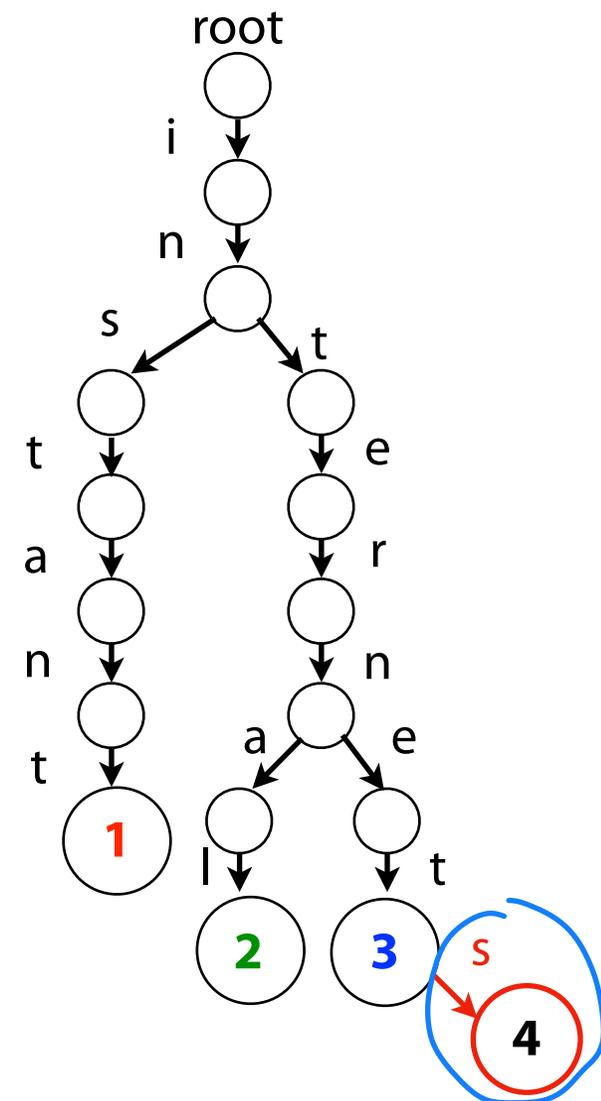
String indexing with Tries

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): ↳ All Substrings

↳ All Locations present

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.
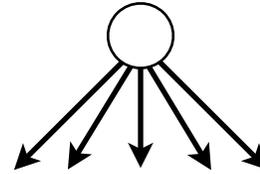
How can we encode this?

N-ary

an array $|\Sigma|$

| N* | N* | N* | | |
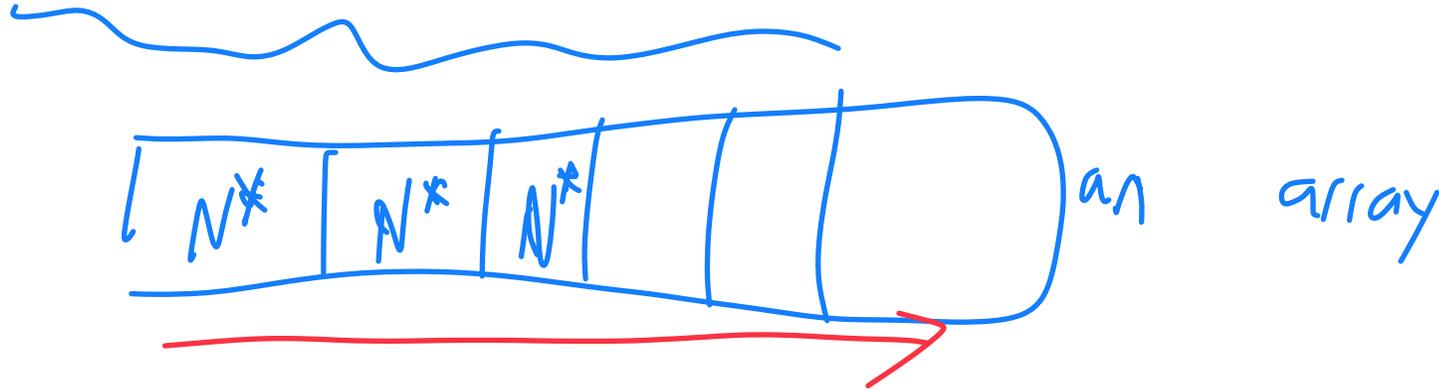
The Node Implementation

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

Each edge is a (potentially NULL) pointer.
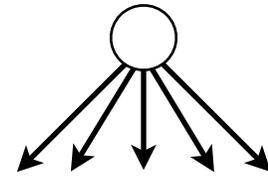
1) Static Array
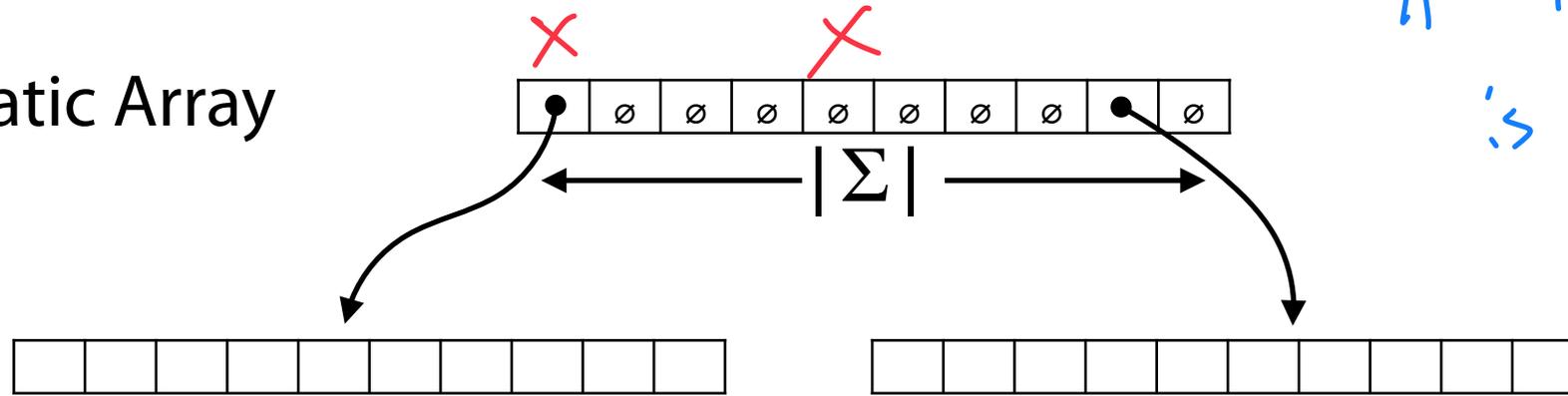
2) Dynamically-sized Dictionary (std::map)
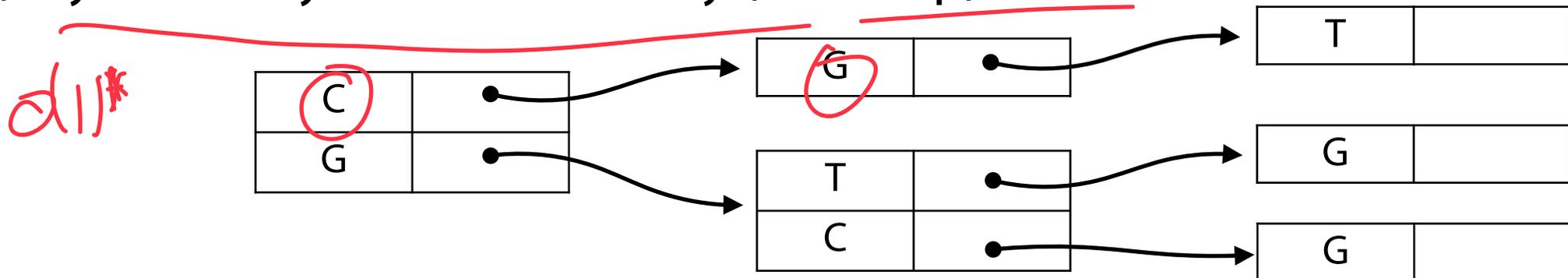
Trie Node Implementation

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;
…
}
```

(handwritten annotations)
my preprocessed answer

edge map of children

Constructor

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 ○

(handwritten annotations) remove front letter — recursive — Base case

Trie Node Implementation

```
 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

C

Trie Node Implementation

NaryTree.h

```cpp
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  }
```

C G T G C

G T G C

T G C

G C

C

G

T

Trie Node Implementation

NaryTree.h

```cpp
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  }
```
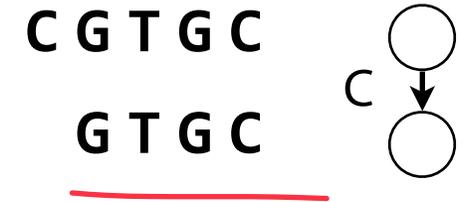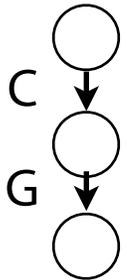
```
C G T G C          ◯
              C    ↓
  G T G C          ◯
              G    ↓
    T G C          ◯
              T    ↓
      G C          ◯
              G    ↓
        C          ◯
```

Trie Node Implementation

NaryTree.h

```cpp
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;
…
}
```

C G T G C

G T G C

T G C

G C

C

i here ->

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
```

```
o
+--A--o           0
          +--B--o
```
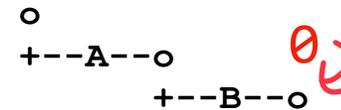
Trie Node Implementation

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

```
o
+--A--o        0
        +--B--o      1
              +--A--o
```
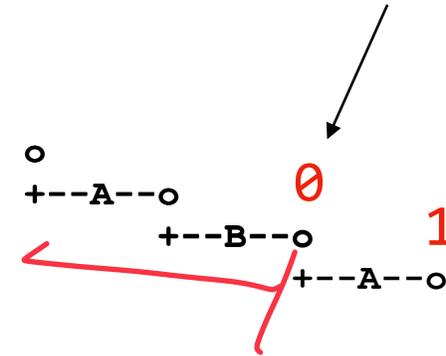
```
struct Node {
        std::vector<int> index;
        std::map<char, Node*> children;
}
```

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
```

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

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
```

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

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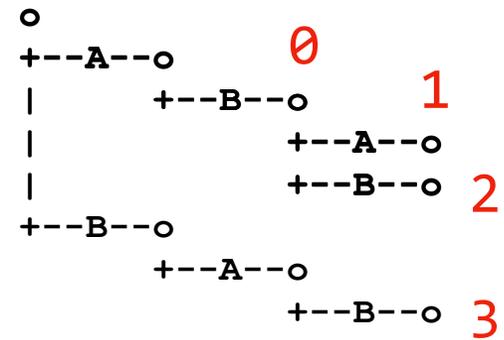
```
o
+--A--o            0
|       +--B--o       1
|              +--A--o
|              +--B--o   2
+--B--o
        +--A--o
        |       +--B--o   3
        +--B--o
                +--B--o   4
```

Trie Node Implementation

NaryTree.h

```cpp
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
```

Base Case for empty Trie

lookup

attempt to follow path

if not make path

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
```

```
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);
}
```

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: `infer`

`infer`

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*:

root

i

oops, no "f"

n

s   t

t   e

a   r

n   n

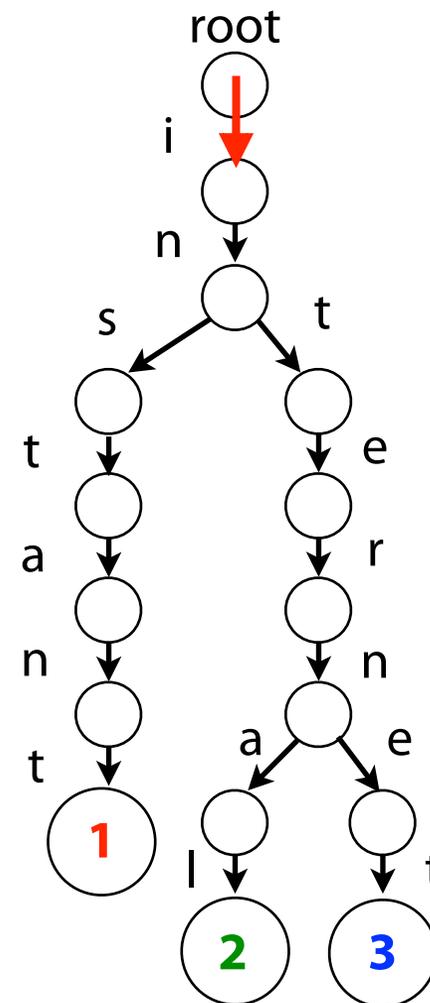t   a   e
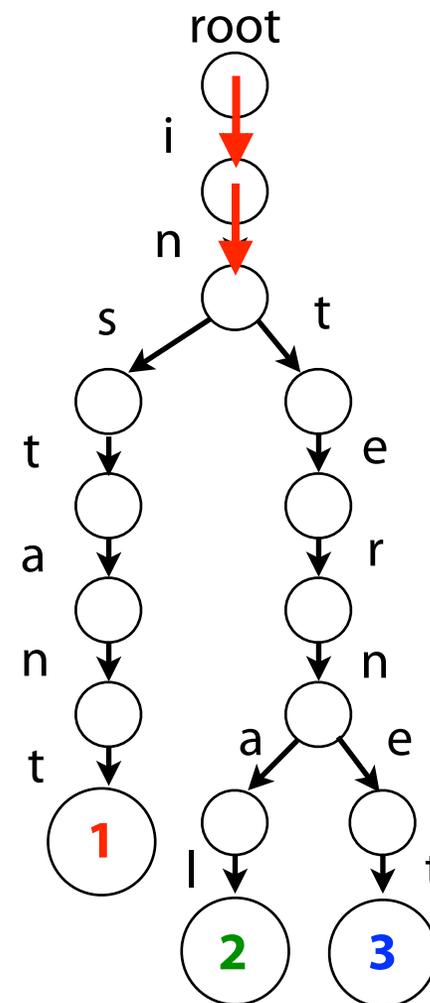
1   l   t

           2   **3**

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

root

oops, no "f"

i

n

s   t

t   e

a   r

n   n

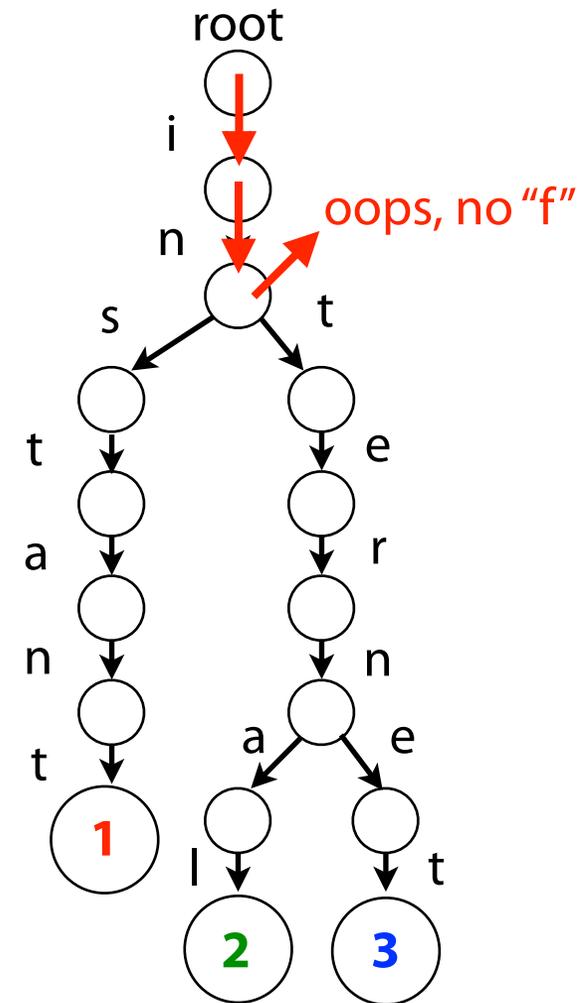t   a   e

1

l   t

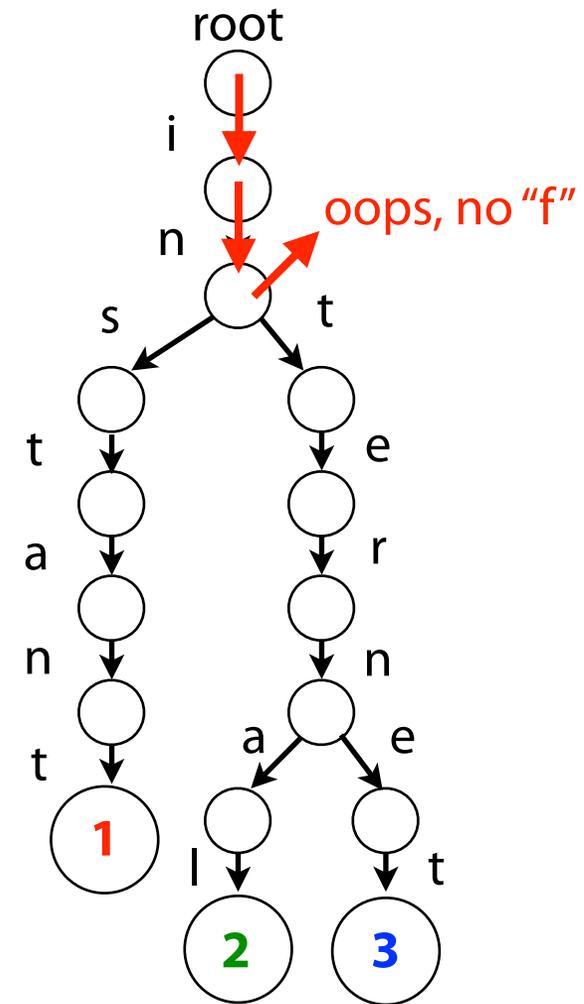**2**   **3**

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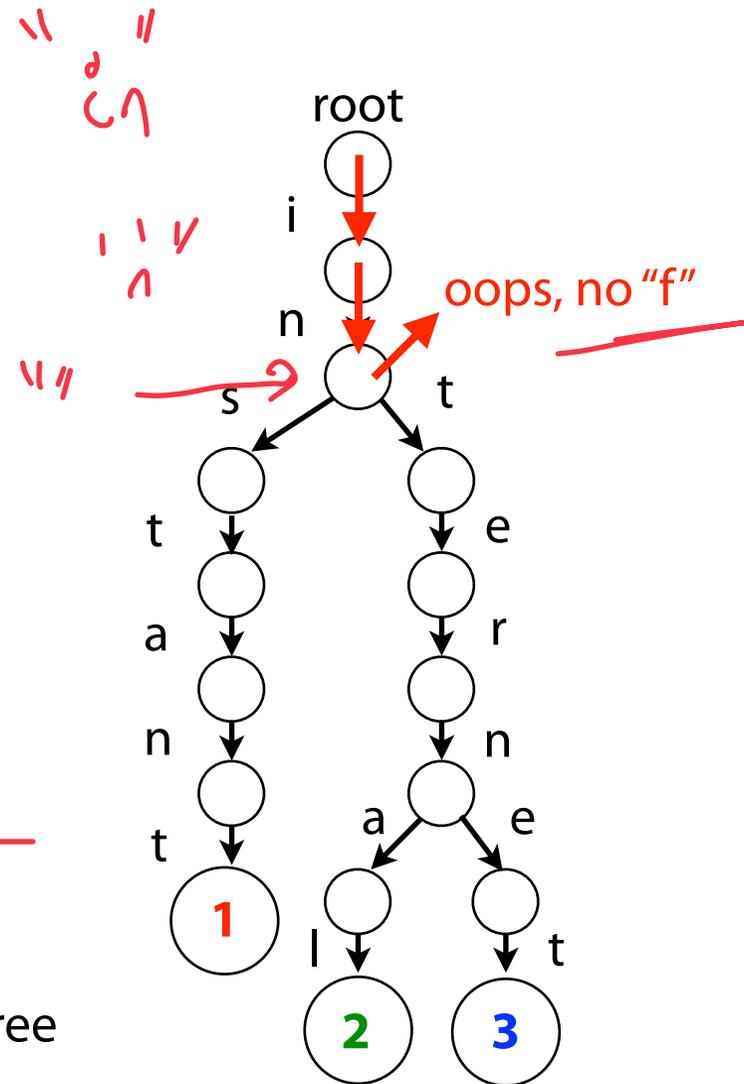
Pattern: `i n f e r`
`i n f e r`
`i n f e r`
`i n f e r`

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!

root

i

n

oops, no "f"

s   t
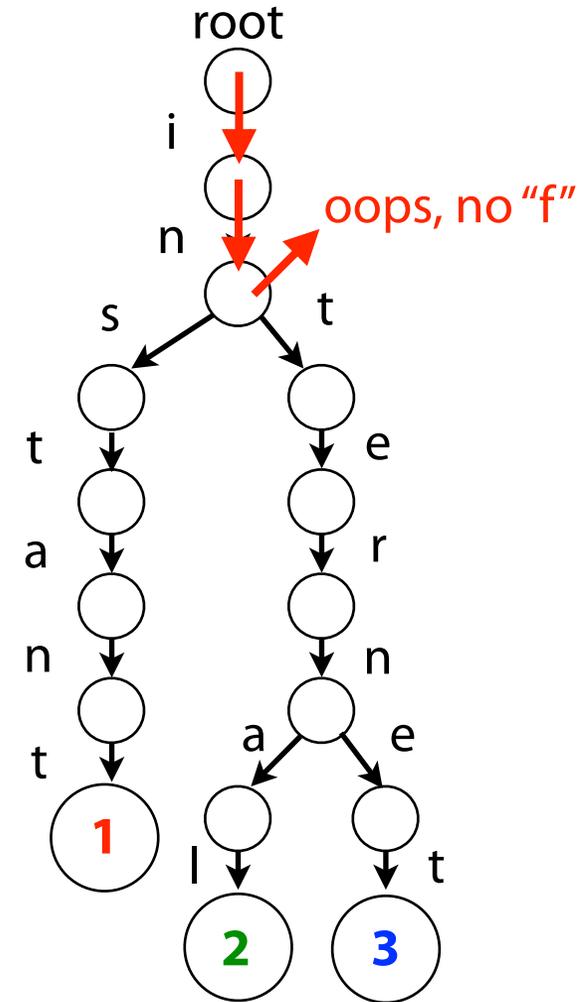
t   e

a   r

n   n

t   a   e

1

l   t

2   **3**

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)

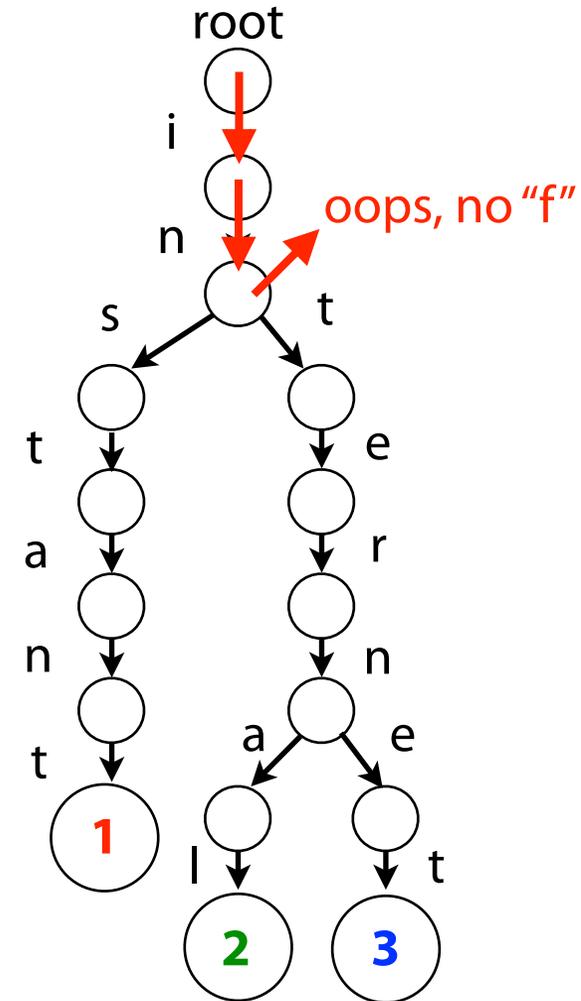(3) If mismatch, *P* is not a key!

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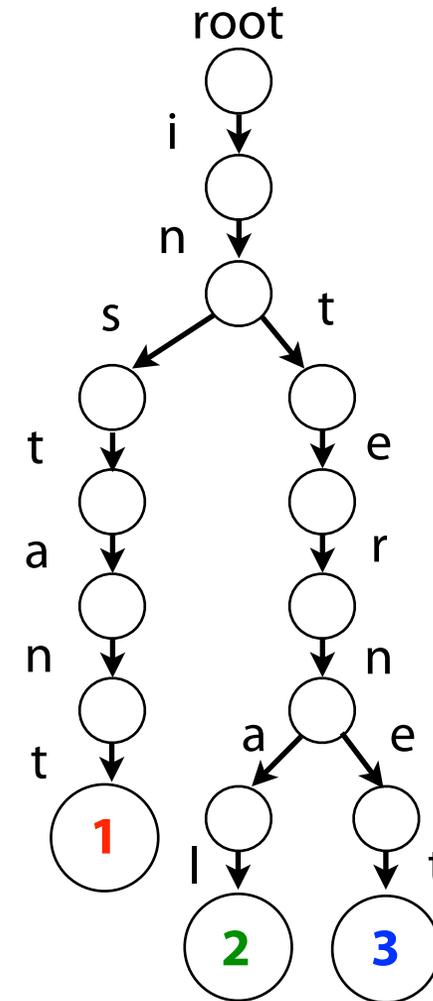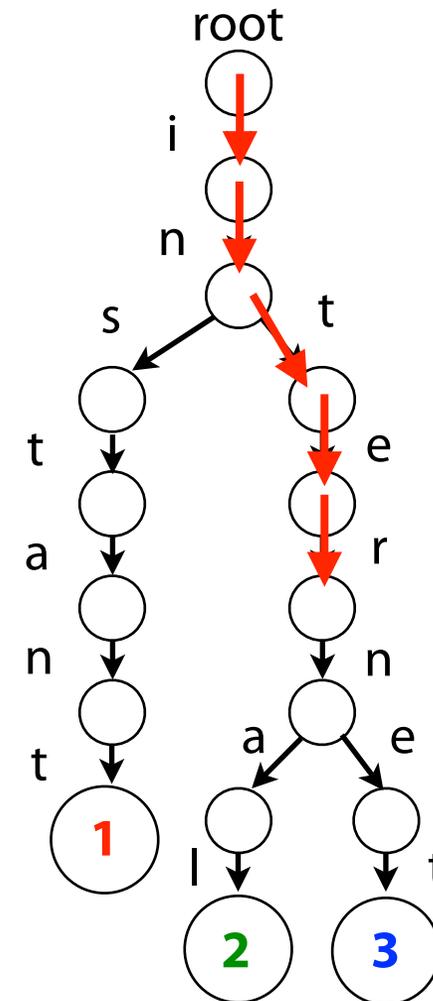
(3) If mismatch, *P* is not a key!

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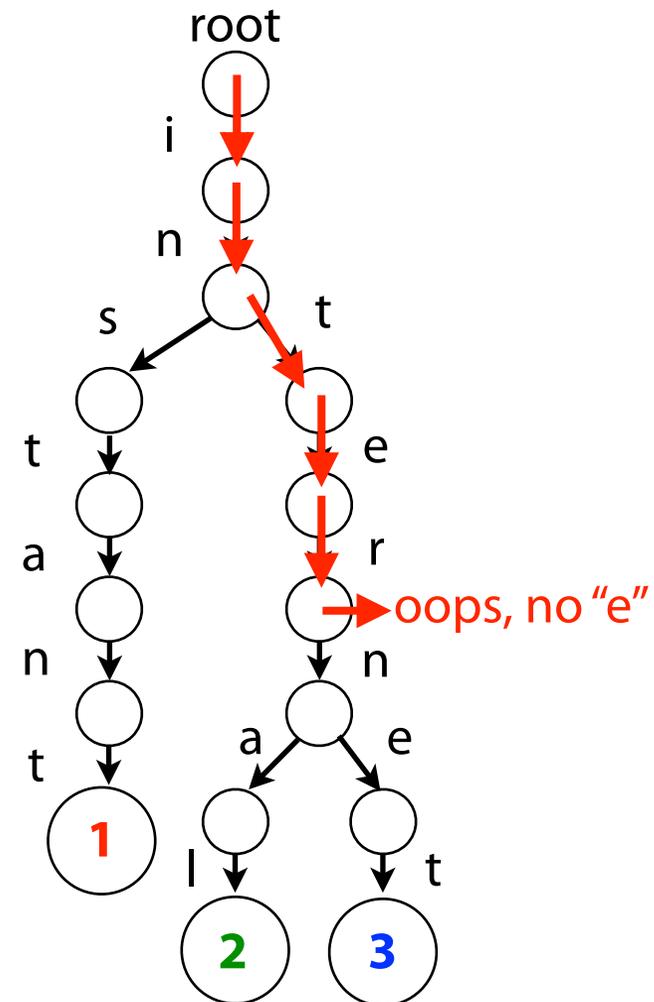
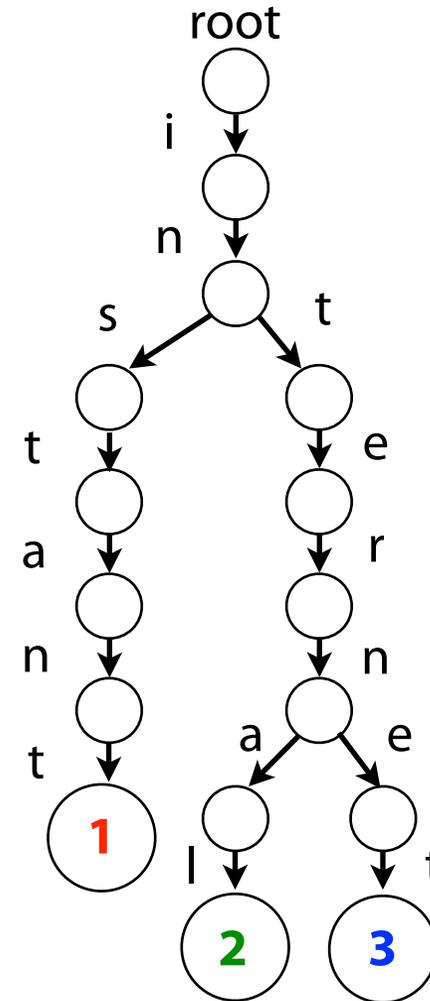   (3) If mismatch, *P* is not a key!

Searching a Trie

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

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

"Insta" is NOT a key!
There's no value here!

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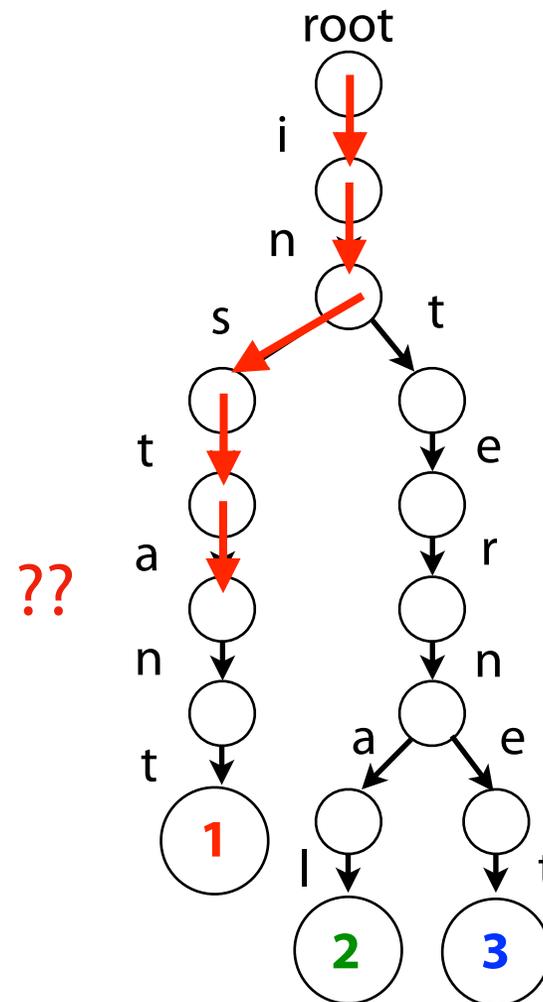
(3) If mismatch, *P* is not a key!

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
```
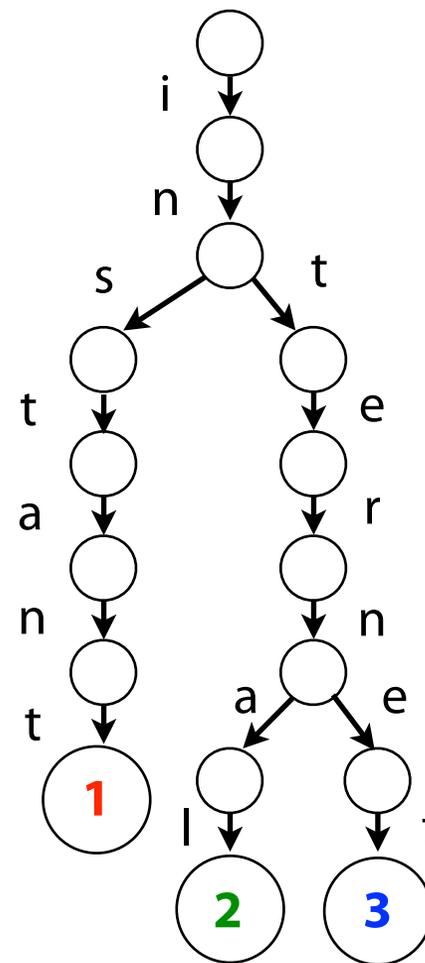
The trie is structured such that:

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
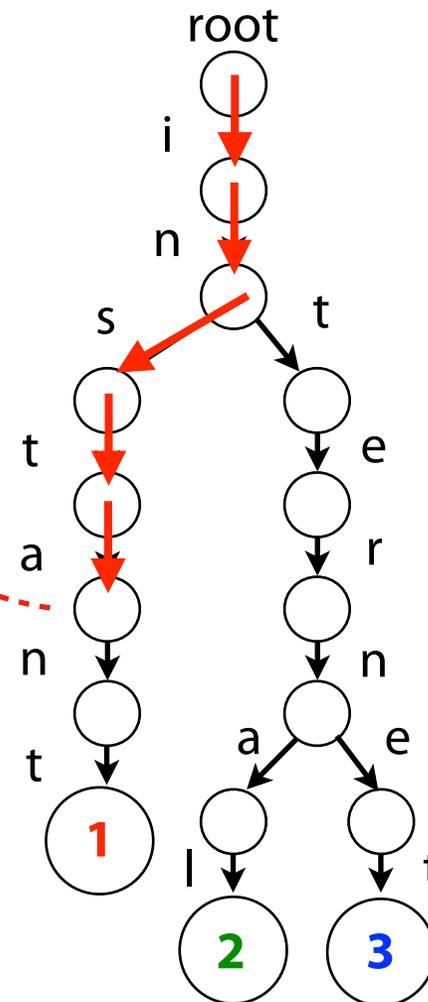
Pattern: `i n s t a`

`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)

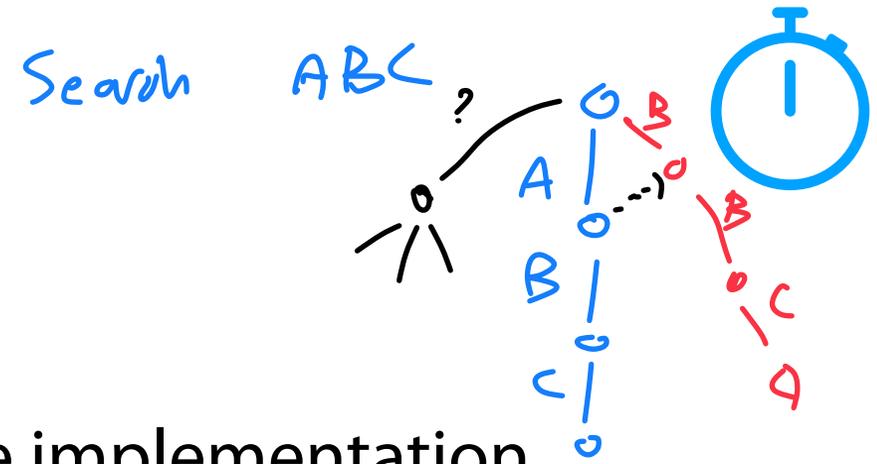(3) If mismatch, *P* is not a key!

return {1}
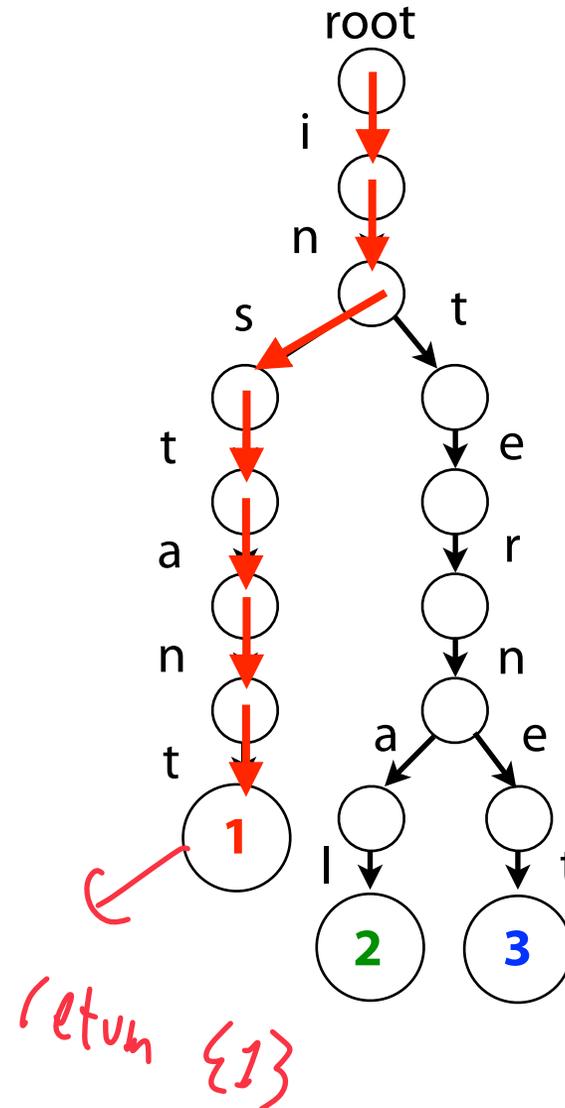
Assignment 5: a_narytree

Learning Objective:

Store all substrings in a trie using NaryTree implementation

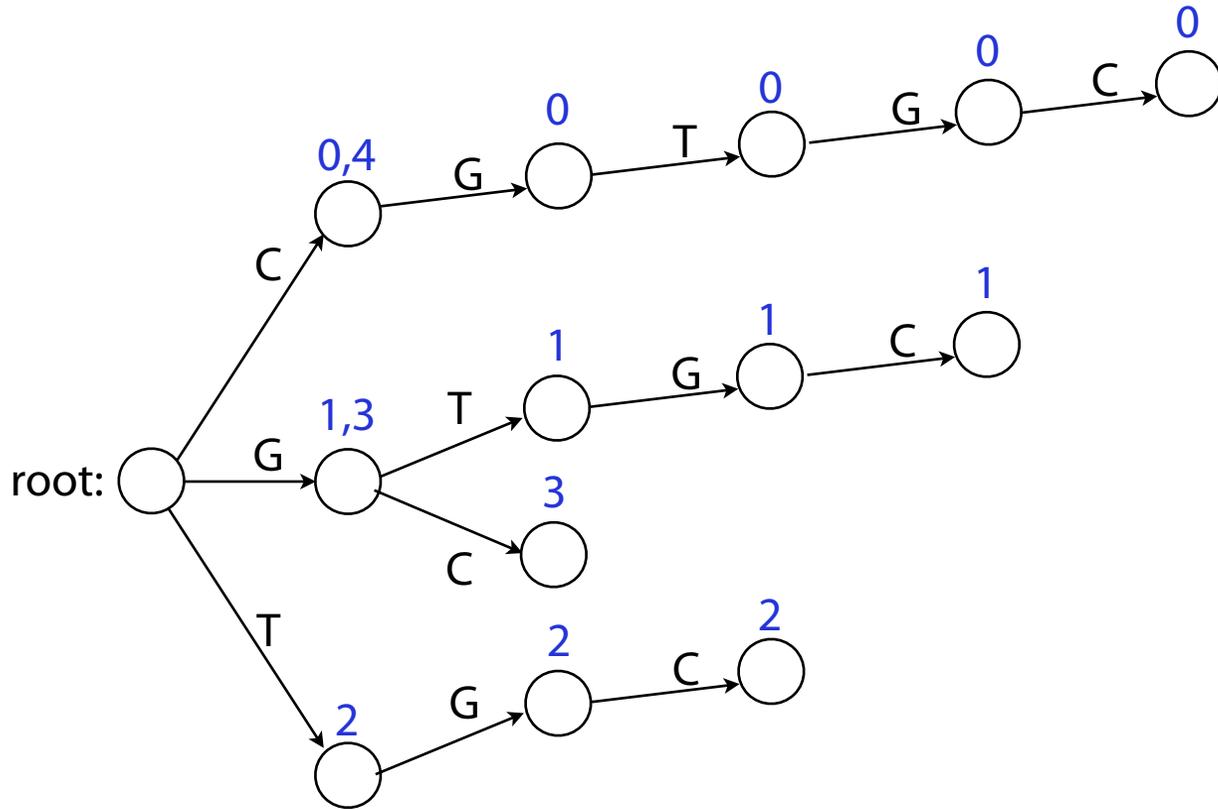**Implement exact pattern matching using this trie**

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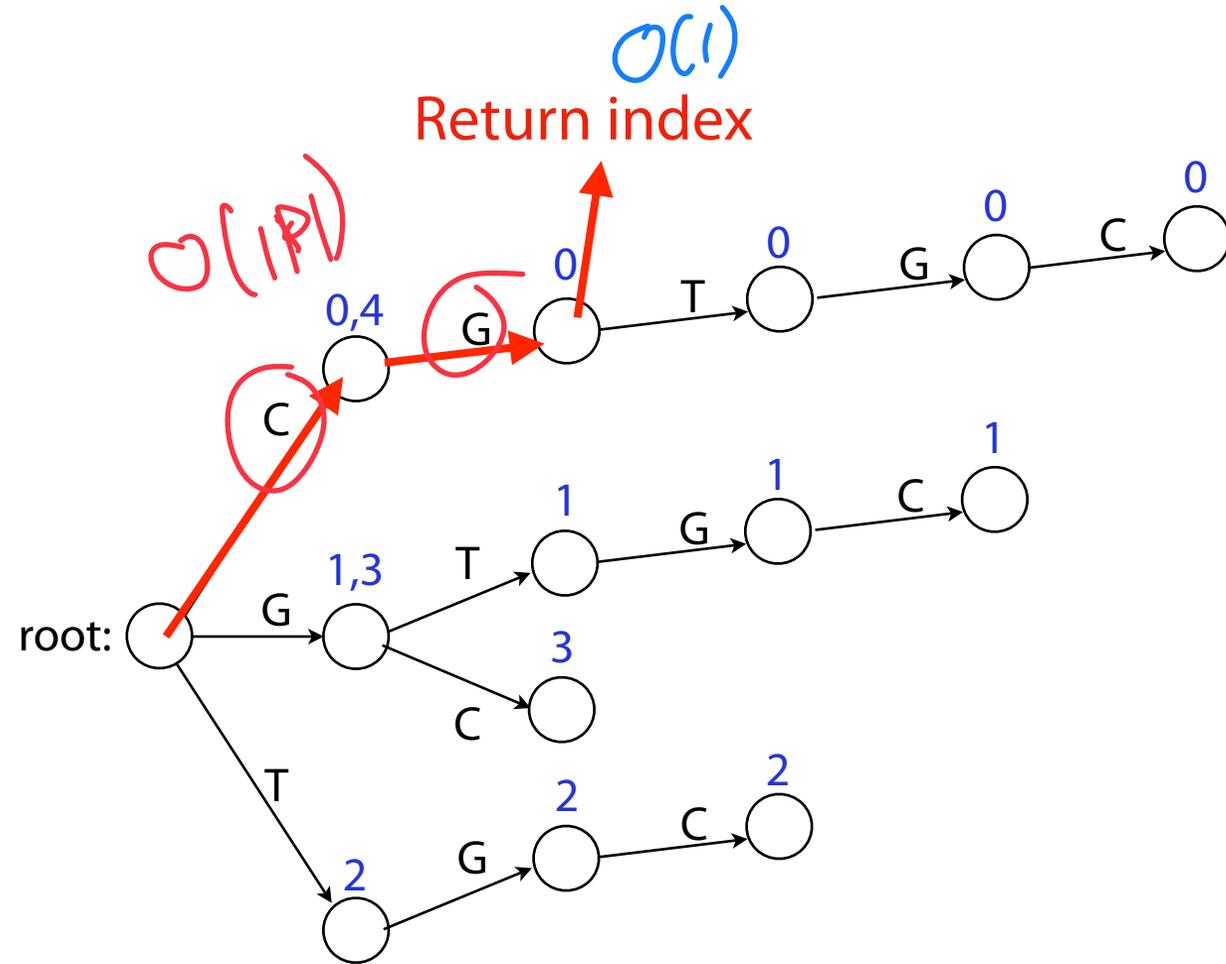
We can do exact pattern matching in $O(P)$ time!

Preprocessing for exact pattern matching

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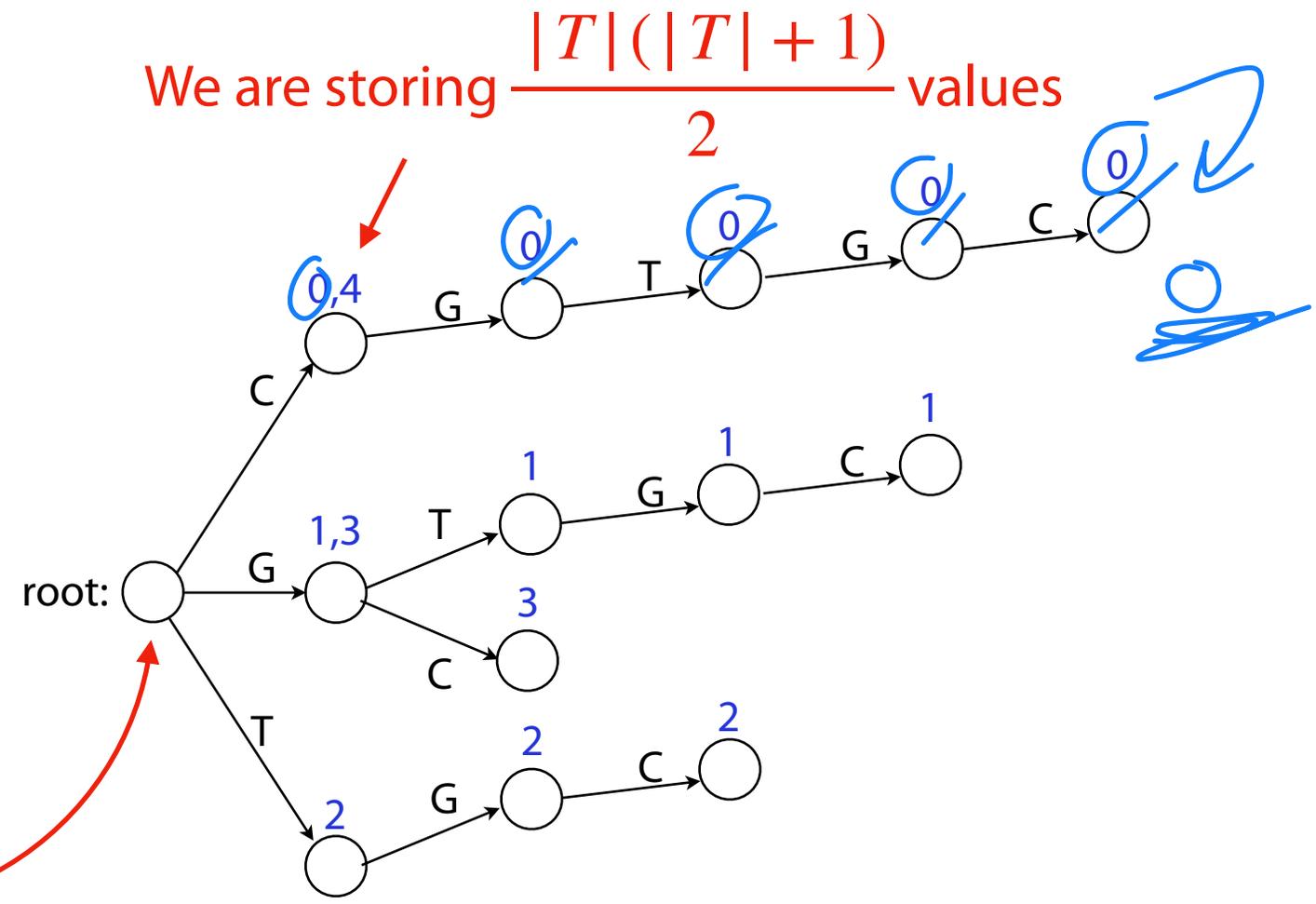
We are storing $\dfrac{|T|(|T|+1)}{2}$ values

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

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

Preprocessing for exact pattern matching

T = abaaba$

If only there was a way…

to insert fewer strings

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!

ab

aba$

why not make all leaves indices