

# XPath Evaluation in Linear Time

Mikołaj Bojańczyk, Paweł Parys  
Warsaw University

**Goal:** find the nodes in an XML document  $d$  that satisfy an XPath unary query  $q$ .

We consider a fragment of XPath called FOXPath.

**Previous algorithms:**

- exponential in the document size
- quadratic in the document size (Benedikt, Koch)

**We give two algorithms:**

- linear in the document size:  $O(2^{|q|} \cdot |d|)$
- good combined complexity:  $O(|q| \cdot |d| \cdot \log(|d|))$

# XML Document

```
<document>
```

```
  <team name="Borussia">
```

```
    <player name="Kuba"></player>
```

```
    <player name="Frei"></player>
```

```
  </team>
```

```
  <team name="Schalke">
```

```
    <player name="Kuranyi">
```

```
  </team>
```

```
  <team name="Poland">
```

```
    <player name="Kuba"></player>
```

```
    <player name="Boruc"></player>
```

```
  </team>
```

```
</document>
```

# XML Document

```
<document>
  <team name="Borussia">
    <player name="Kuba"></player>
    <player name="Frei"></player>
  </team>

  <team name="Schalke">
    <player name="Kuranyi">
  </team>

  <team name="Poland">
    <player name="Kuba"></player>
    <player name="Boruc"></player>
  </team>
</document>
```

attribute name

attribute name

document node,  
i.e. opening tag

# XML Document

```
<document>
```

attribute name

attribute name

```
<team name="Borussia">
```

```
<player name="Kuba"></player>
```

```
<player name="Frei"></player>
```

```
</team>
```

```
<team name="Schalke">
```

```
<player name="Kuranyi">
```

```
</team>
```

```
<team name="Poland">
```

```
<player name="Kuba"></player>
```

```
<player name="Boruc"></player>
```

```
</team>
```

```
</document>
```

document node,  
i.e. opening tag

# XML Document

```
<document>
  <team name="Borussia">
    <player name="Kuba"></player>
    <player name="Frei"></player>
  </team>

  <team name="Schalke">
    <player name="Kuranyi">
  </team>

  <team name="Poland">
    <player name="Kuba"></player>
    <player name="Boruc"></player>
  </team>
</document>
```

attribute name

attribute name

document node,  
i.e. opening tag

XPath query: “select teams that share a player with another team”

# XML Document

```
<document>  
  <team name="Borussia">  
    <player name="Kuba"></player>  
    <player name="Frei"></player>  
  </team>  
  
  <team name="Schalke">  
    <player name="Kuranyi">  
  </team>  
  
  <team name="Poland">  
    <player name="Kuba"></player>  
    <player name="Boruc"></player>  
  </team>  
</document>
```

attribute name

attribute name

document node,  
i.e. opening tag

XPath query: “select teams that share a player with another team”

# XML Document

```
<document>
  <team name="Borussia">
    <player name="Kuba"></player>
    <player name="Frei"></player>
  </team>

  <team name="Schalke">
    <player name="Kuranyi">
  </team>

  <team name="Poland">
    <player name="Kuba"></player>
    <player name="Boruc"></player>
  </team>
</document>
```

attribute name

attribute name

document node,  
i.e. opening tag

XPath query: “select teams that share a player with another team”

# XML Document

```
<document>
  <team name="Borussia">
    <player name="Kuba"></player>
    <player name="Frei"></player>
  </team>

  <team name="Schalke">
    <player name="Kuranyi">
  </team>

  <team name="Poland">
    <player name="Kuba"></player>
    <player name="Boruc"></player>
  </team>
</document>
```

attribute name

attribute name

document node,  
i.e. opening tag

XPath query: “select teams that share a player with another team”

# XML Document



XPath query: “select teams that share a player with another team”

# XML Document



XPath query: “select teams that share a player with another team”

# XML Document



XPath query: “select teams that share a player with another team”

# XML Document



XPath query: “select teams that share a player with another team”

# XML Document



`child[player]@name = sibling[team]/child[player]@name`

XPath query: "select teams that share a player with another team"

## Programs - select node pairs.

- child, parent, next-sibling, prev-sibling, descendant, etc.
- any regular expression on programs is a program, e.g. `child*`
- if  $t$  is a test, then  $[t]$  is a program that selects  $(x,x)$  if node  $x$  satisfies  $t$

## Tests - select single nodes.

- any tag name  $a$  is a test that selects nodes with this tag.
- boolean operations: *or*, *and*, *not*
- if  $p, q$  are programs, and  $a, b$  attribute names, then  $p@a=q@b$  and  $p@a \neq p@b$  are tests.

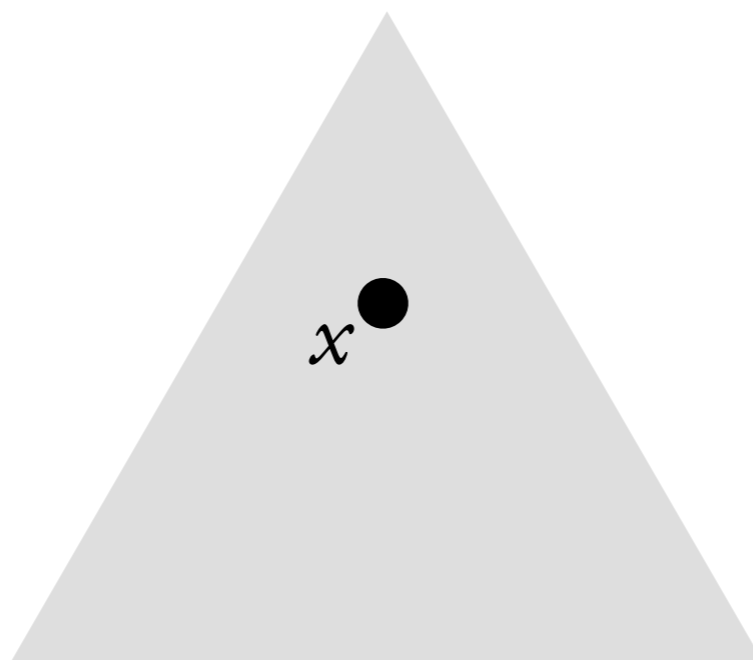
## Programs - select node pairs.

- child, parent, next-sibling, prev-sibling, descendant, etc.
- any regular expression on programs is a program, e.g.  $\text{child}^*$
- if  $t$  is a test, then  $[t]$  is a program that selects  $(x,x)$  if node  $x$  satisfies  $t$

## Tests - select single nodes.

- any tag name  $a$  is a test that selects nodes with this tag.
- boolean operations: *or*, *and*, *not*
- if  $p, q$  are programs, and  $a, b$  attribute names, then  $p@a=q@b$  and  $p@a \neq p@b$  are tests.

A node  $x$  is selected by  $p@a=q@b$  if



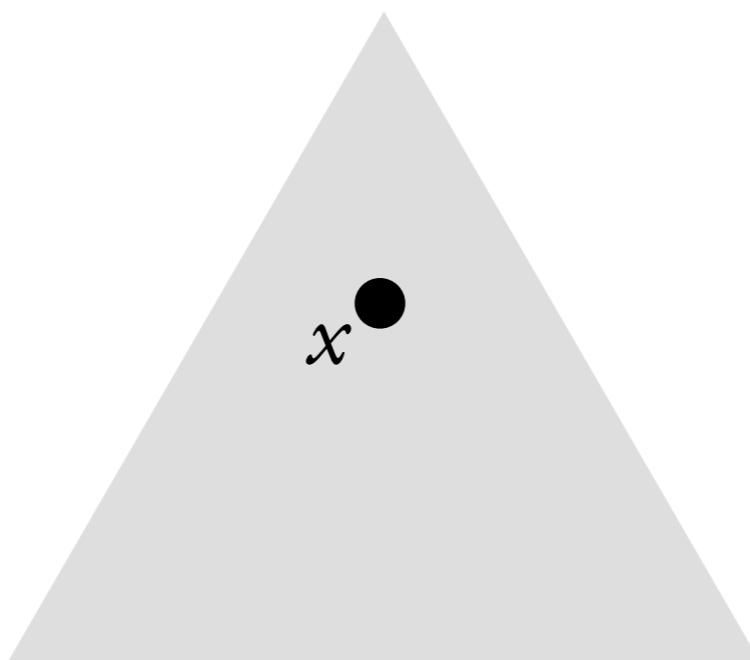
## Programs - select node pairs.

- child, parent, next-sibling, prev-sibling, descendant, etc.
- any regular expression on programs is a program, e.g.  $\text{child}^*$
- if  $t$  is a test, then  $[t]$  is a program that selects  $(x,x)$  if node  $x$  satisfies  $t$

## Tests - select single nodes.

- any tag name  $a$  is a test that selects nodes with this tag.
- boolean operations: *or*, *and*, *not*
- if  $p, q$  are programs, and  $a, b$  attribute names, then  $p@a=q@b$  and  $p@a \neq p@b$  are tests.

A node  $x$  is selected by  $p@a=q@b$  if  
there are some nodes  $y$  and  $z$  such that



## Programs - select node pairs.

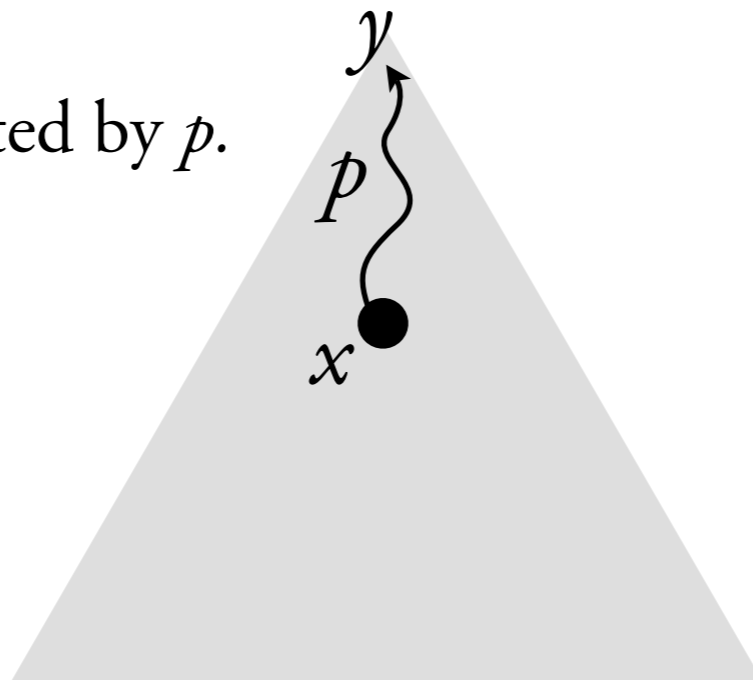
- child, parent, next-sibling, prev-sibling, descendant, etc.
- any regular expression on programs is a program, e.g.  $\text{child}^*$
- if  $t$  is a test, then  $[t]$  is a program that selects  $(x,x)$  if node  $x$  satisfies  $t$

## Tests - select single nodes.

- any tag name  $a$  is a test that selects nodes with this tag.
- boolean operations: *or*, *and*, *not*
- if  $p, q$  are programs, and  $a, b$  attribute names, then  $p@a=q@b$  and  $p@a \neq p@b$  are tests.

A node  $x$  is selected by  $p@a=q@b$  if  
there are some nodes  $y$  and  $z$  such that

the pair  $(x,y)$  is selected by  $p$ .



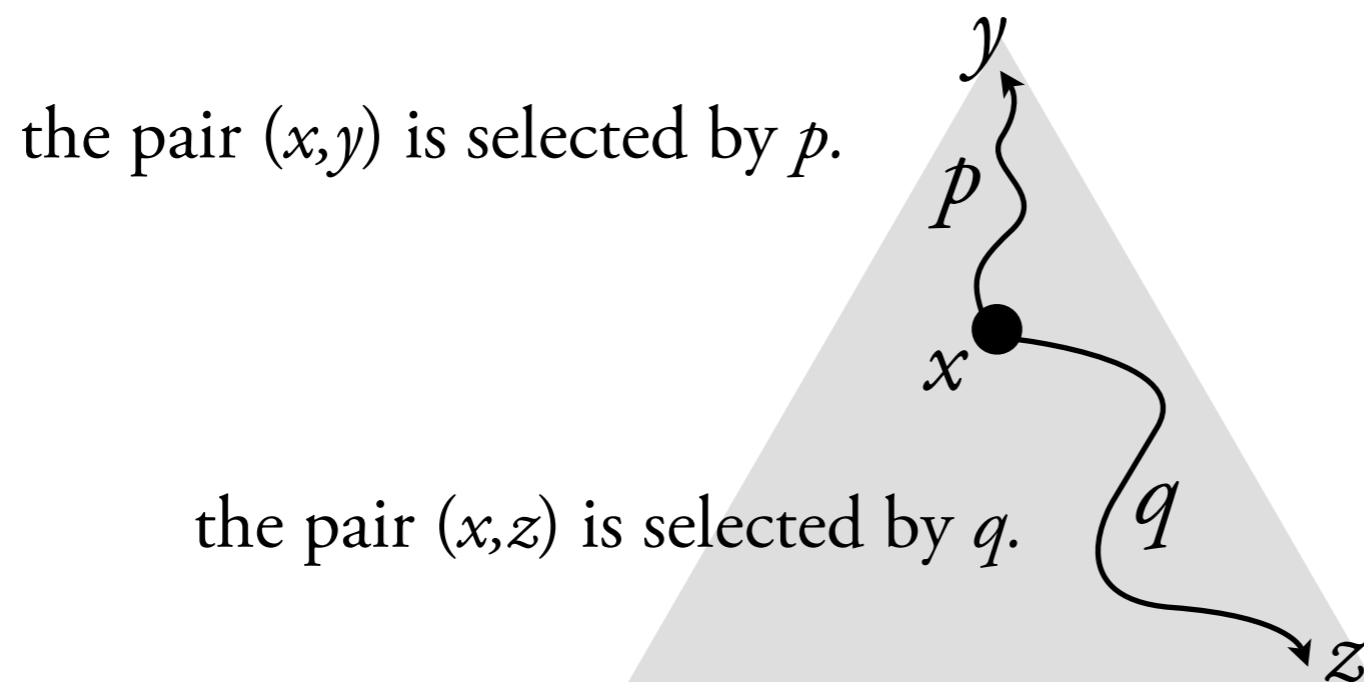
## Programs - select node pairs.

- child, parent, next-sibling, prev-sibling, descendant, etc.
- any regular expression on programs is a program, e.g.  $\text{child}^*$
- if  $t$  is a test, then  $[t]$  is a program that selects  $(x,x)$  if node  $x$  satisfies  $t$

## Tests - select single nodes.

- any tag name  $a$  is a test that selects nodes with this tag.
- boolean operations: *or*, *and*, *not*
- if  $p, q$  are programs, and  $a, b$  attribute names, then  $p@a=q@b$  and  $p@a \neq p@b$  are tests.

A node  $x$  is selected by  $p@a=q@b$  if  
there are some nodes  $y$  and  $z$  such that



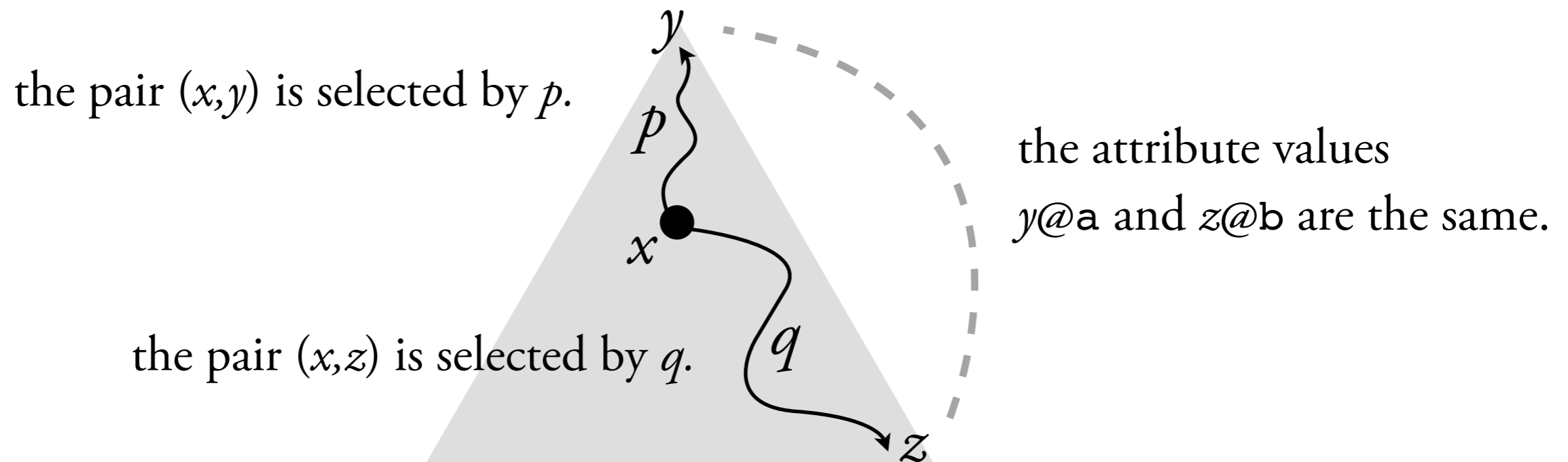
## Programs - select node pairs.

- child, parent, next-sibling, prev-sibling, descendant, etc.
- any regular expression on programs is a program, e.g.  $\text{child}^*$
- if  $t$  is a test, then  $[t]$  is a program that selects  $(x,x)$  if node  $x$  satisfies  $t$

## Tests - select single nodes.

- any tag name  $a$  is a test that selects nodes with this tag.
- boolean operations: *or*, *and*, *not*
- if  $p, q$  are programs, and  $a, b$  attribute names, then  $p@a=q@b$  and  $p@a \neq p@b$  are tests.

A node  $x$  is selected by  $p@a=q@b$  if  
there are some nodes  $y$  and  $z$  such that



## Programs - select node pairs.

- child, parent, next-sibling, prev-sibling, descendant, etc.
- any regular expression on programs is a program, e.g.  $\text{child}^*$
- if  $t$  is a test, then  $[t]$  is a program that selects  $(x,x)$  if node  $x$  satisfies  $t$

## Tests - select single nodes.

any tag name  $a$  is a test that selects nodes with this tag.

boolean operations: *or*, *and*, *not*

if  $p$  and  $q$  are programs, and  $a, b$  attribute names, then  $p@a=q@b$  and  $p@a \neq p@b$  are tests.

### Thm.

Let  $t$  be an FOXPath test and  $d$  an XML document.

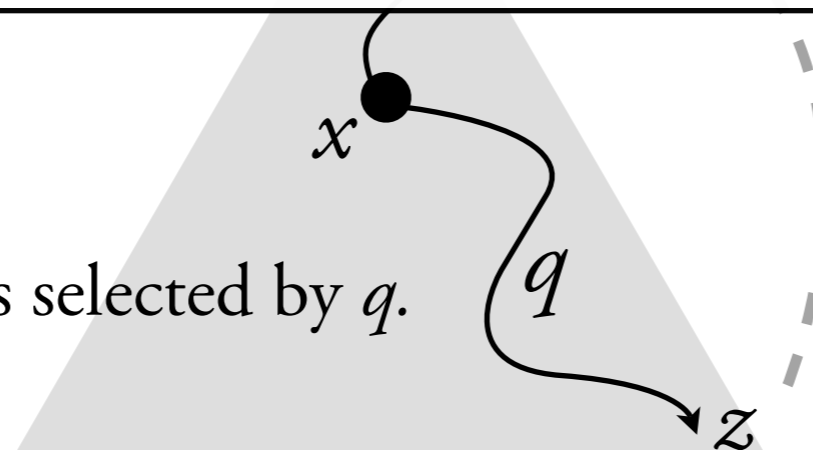
The set of nodes in  $d$  selected by  $t$  can be computed in time  $O(|d|2^{|t|})$  as well as in time  $O(|d|\log(|d|)|t|^2)$

the pair  $(x,y)$  is selected by  $p$ .

the attribute values

$y@a$  and  $z@b$  are the same.

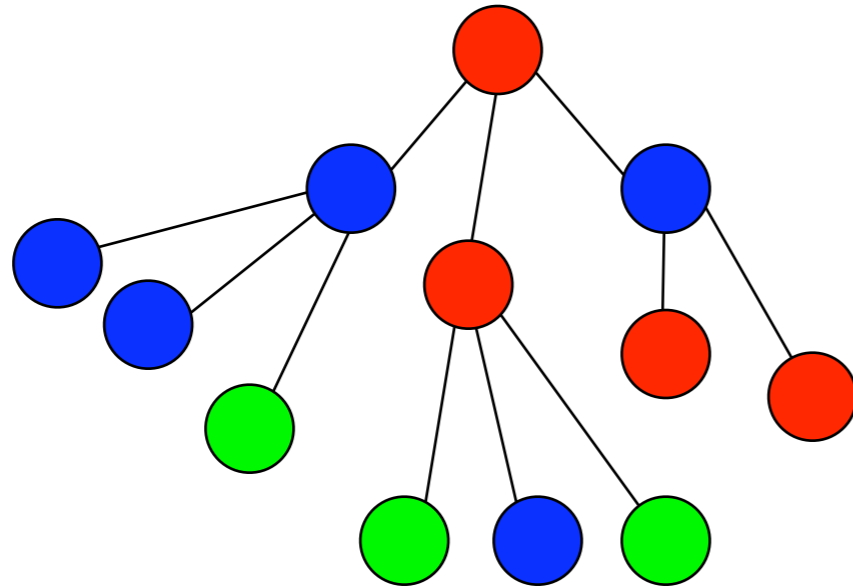
the pair  $(x,z)$  is selected by  $q$ .



find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes

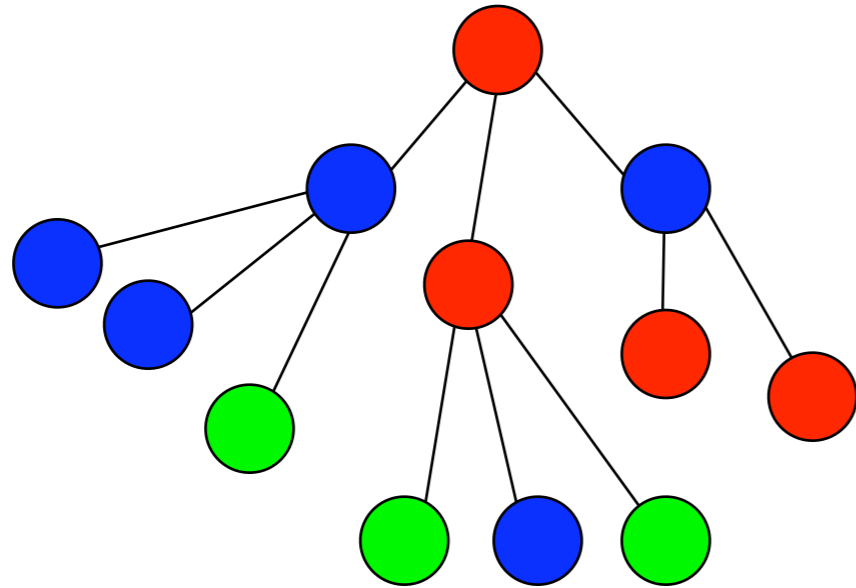
(class = set of nodes with same value of attribute a)



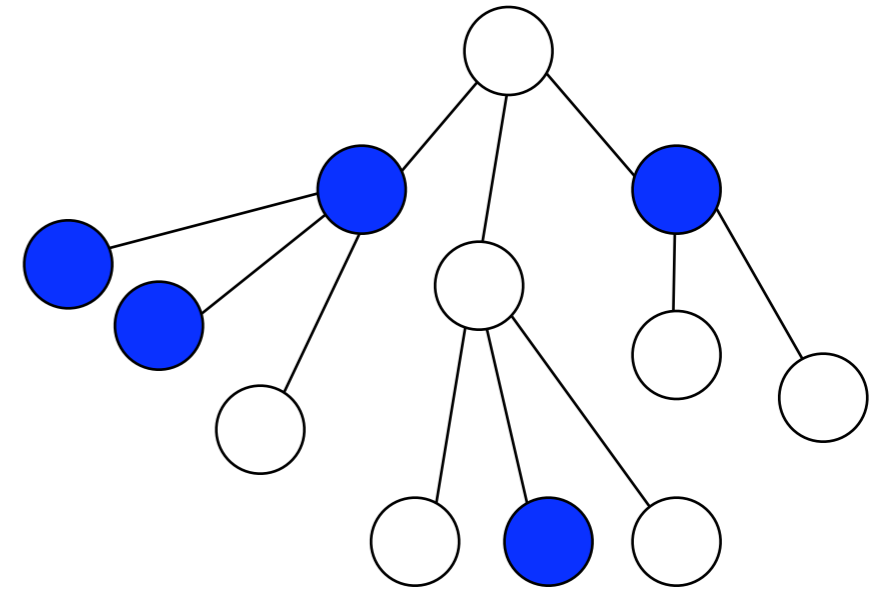
find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes

(class = set of nodes with same value of attribute a)



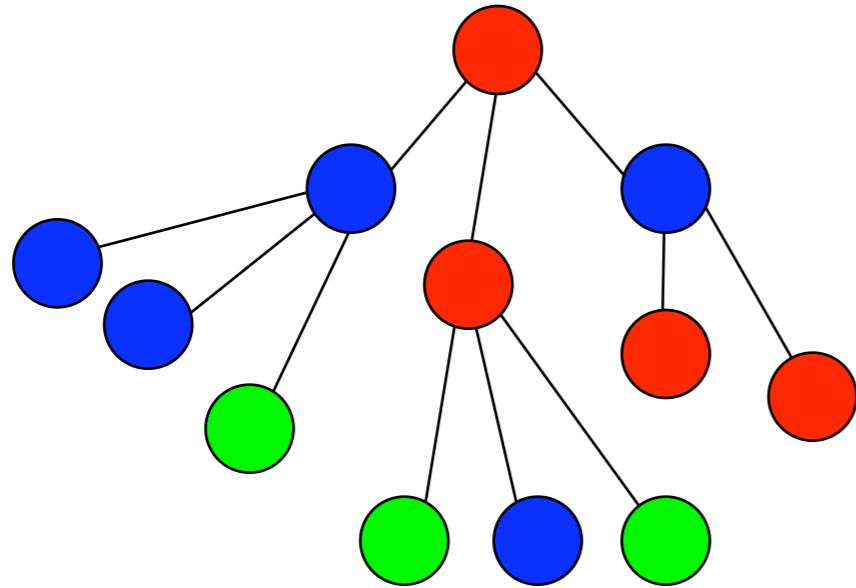
# High level overview



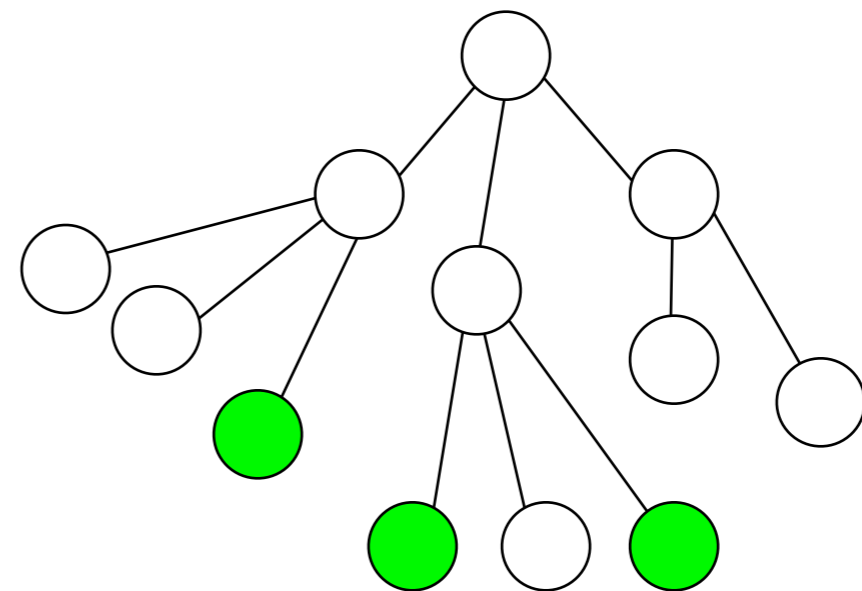
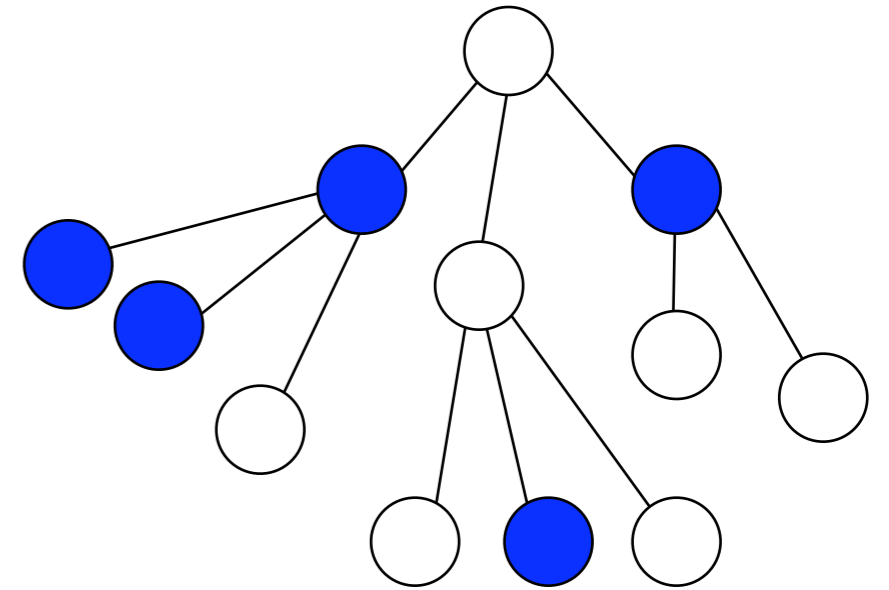
find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes

(class = set of nodes with same value of attribute a)



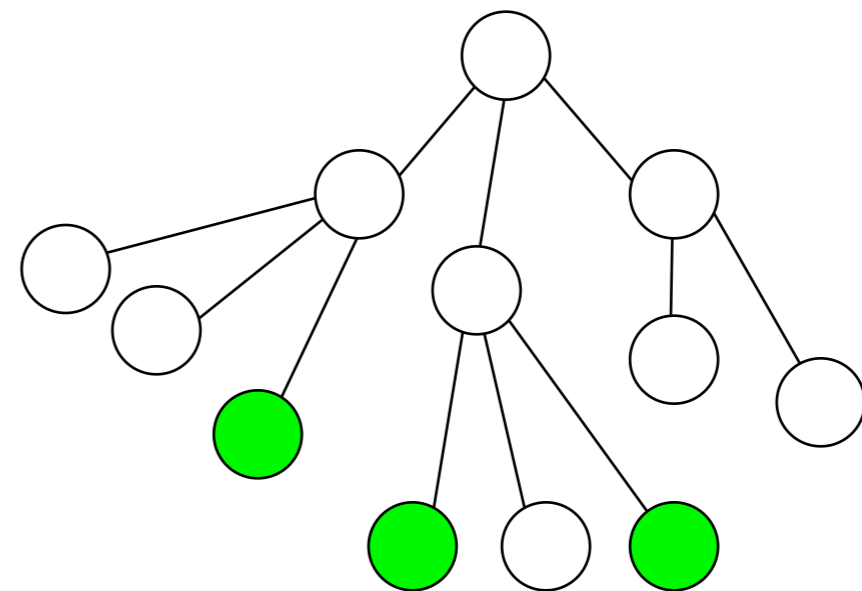
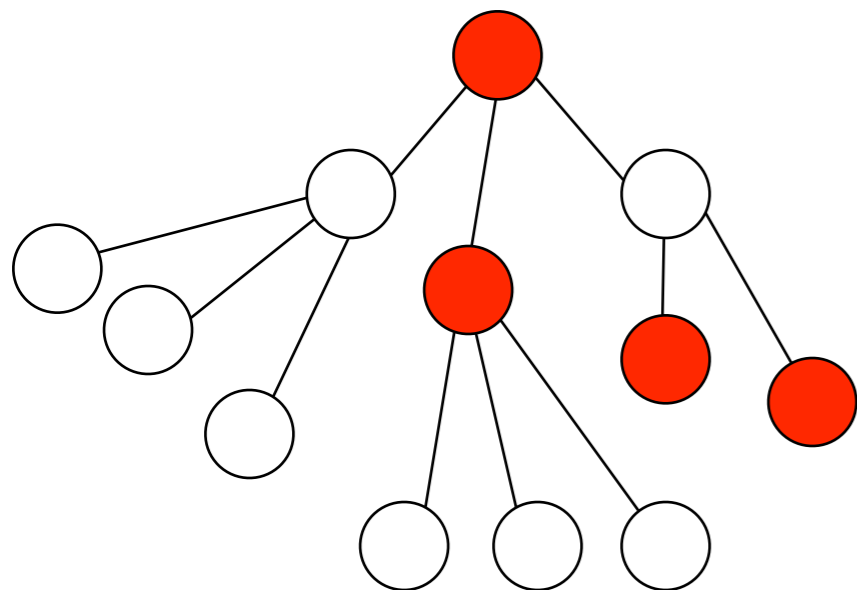
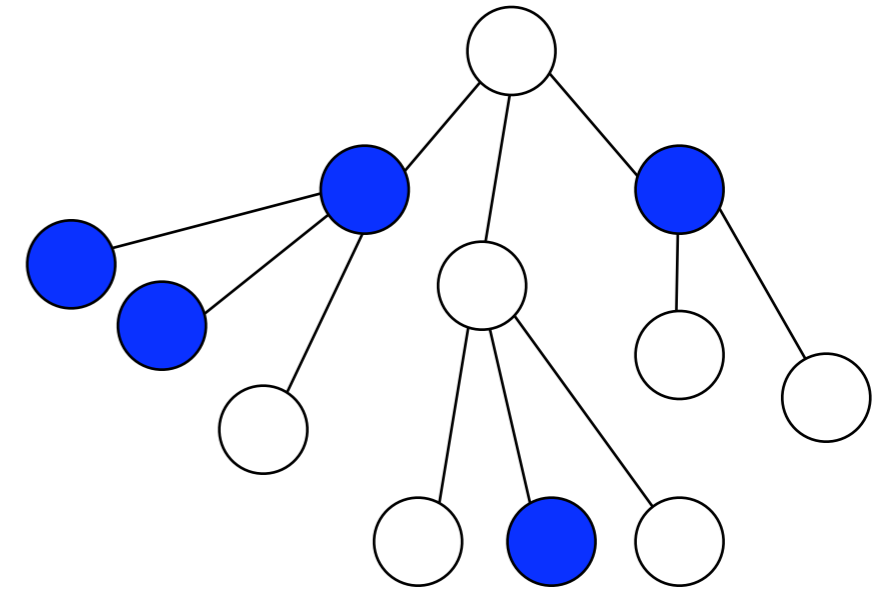
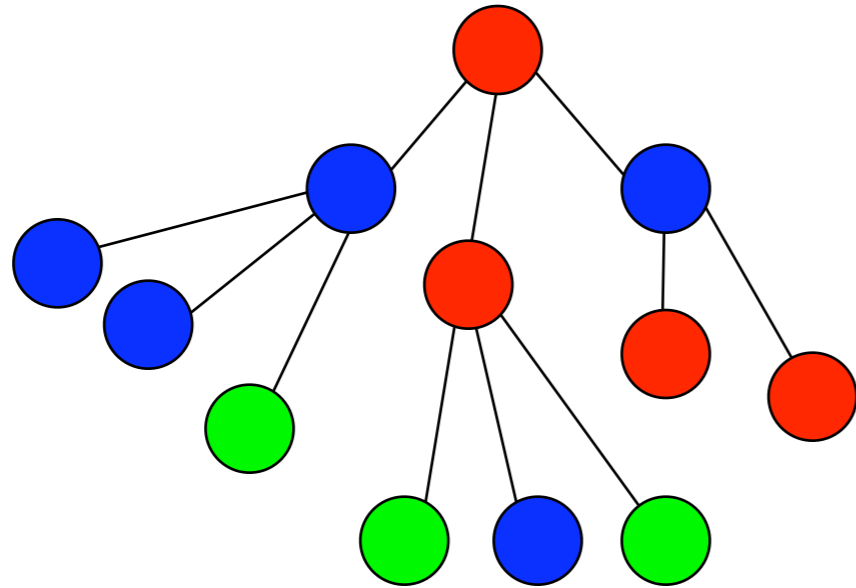
# High level overview



find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes

(class = set of nodes with same value of attribute a)

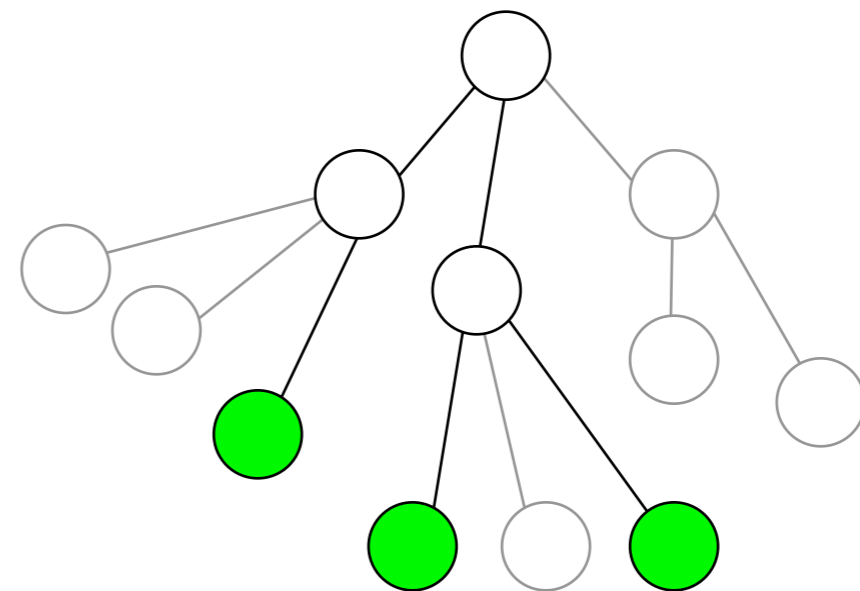
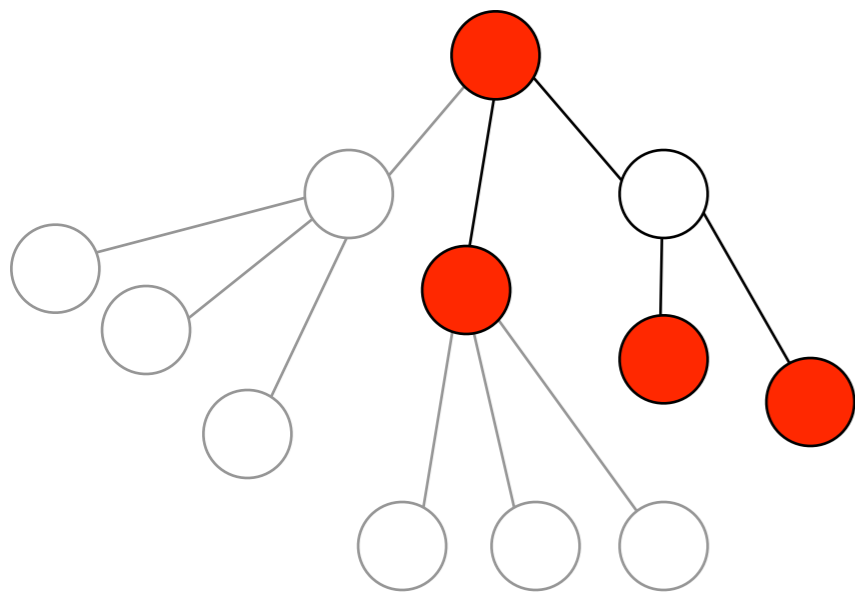
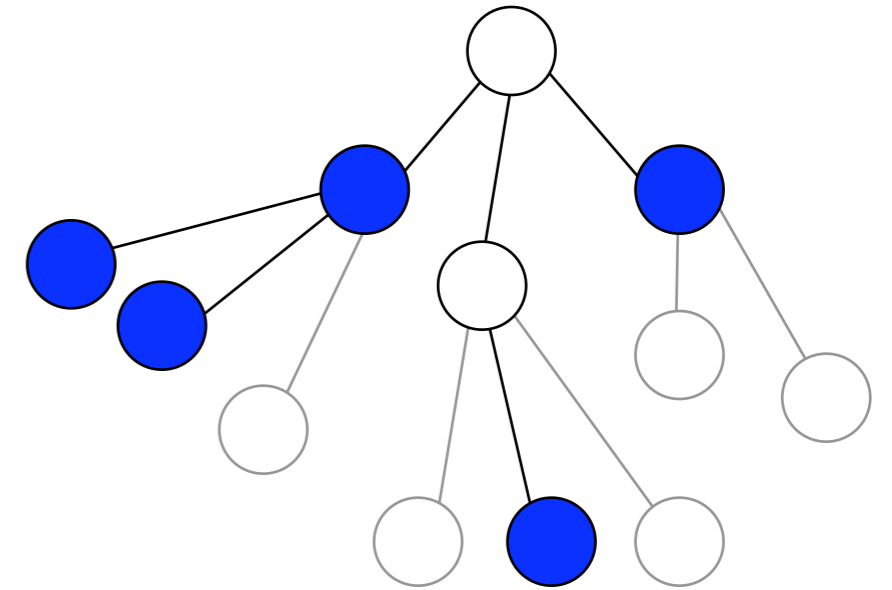
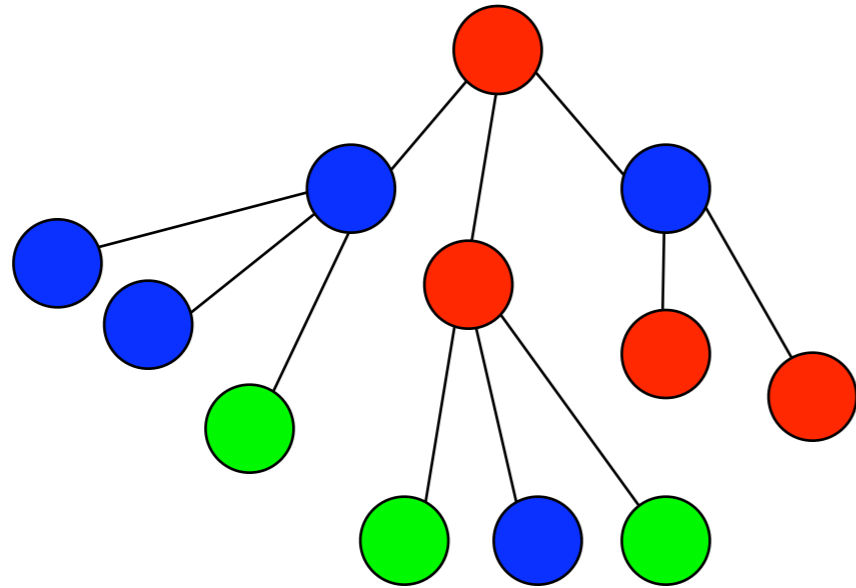


High level overview

find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes

(class = set of nodes with same value of attribute a)



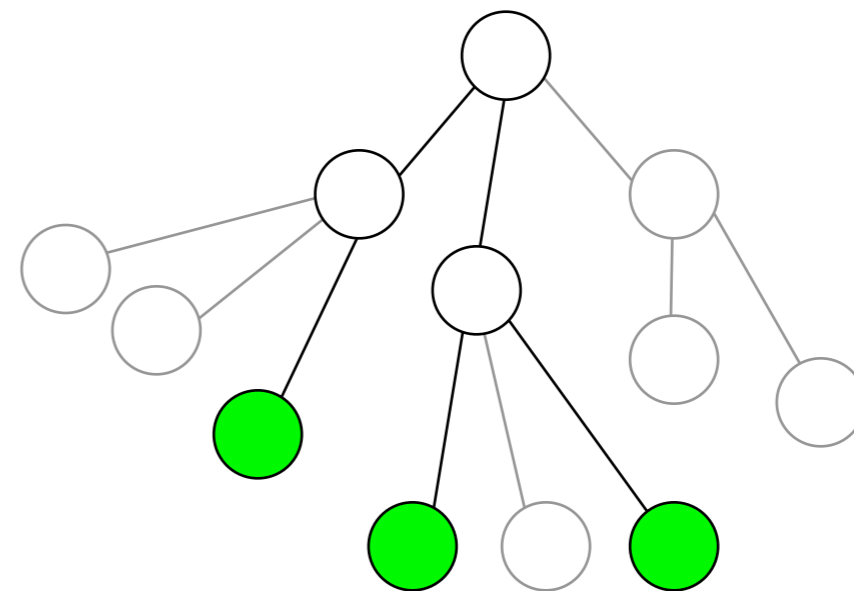
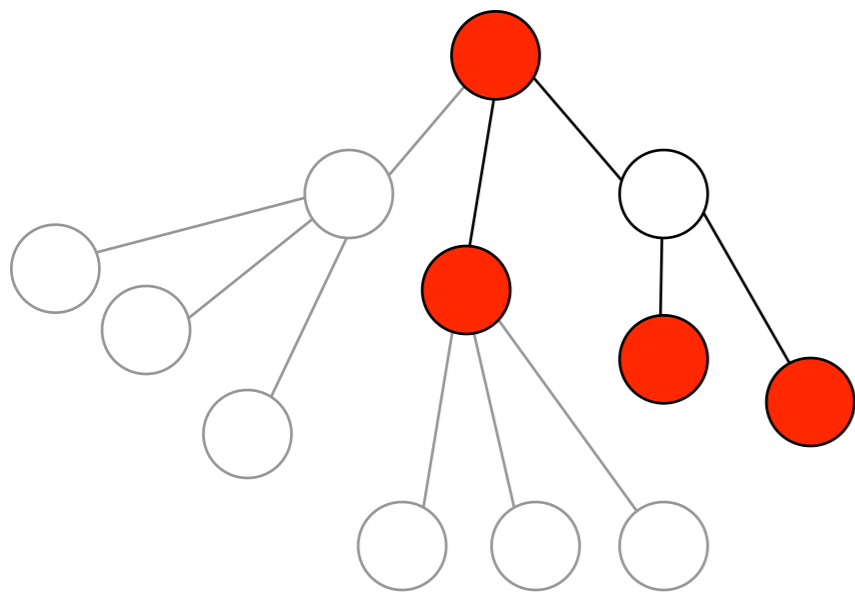
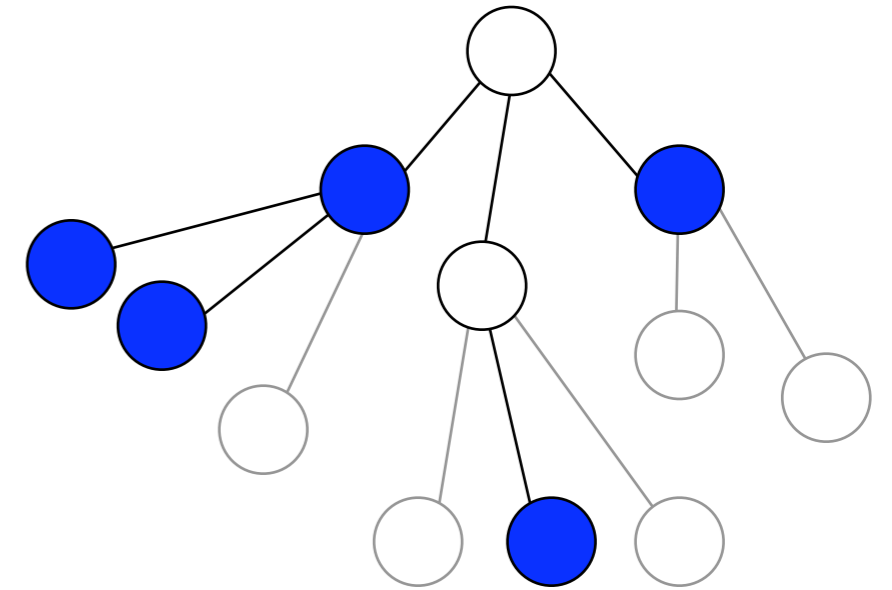
High level overview



find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes  
(class = set of nodes with same value of attribute a)
2. for each class, find nodes that are witnessed by that class

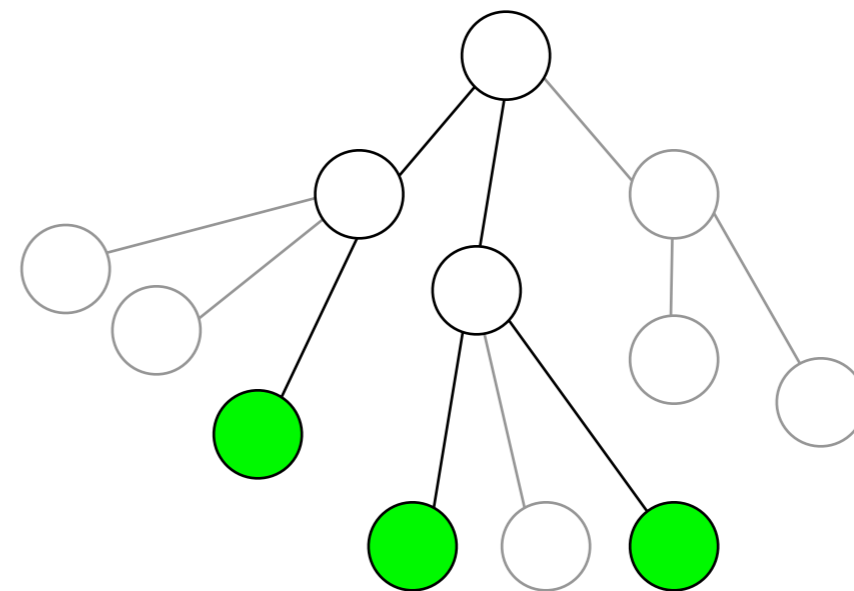
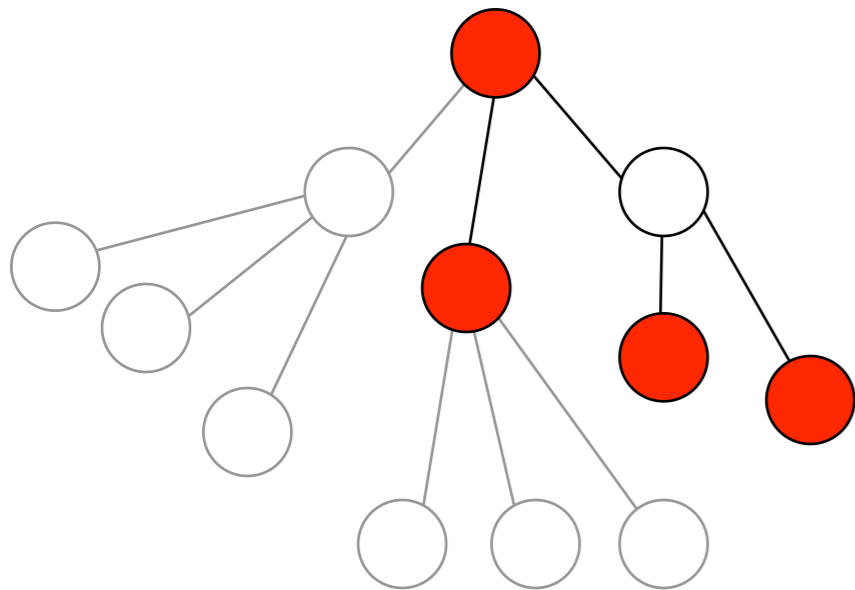
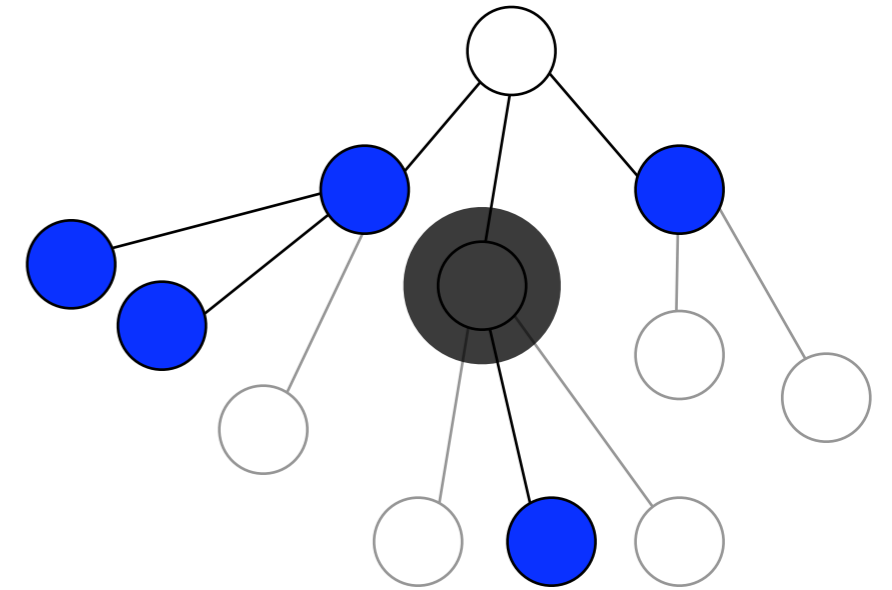
## High level overview



find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes  
(class = set of nodes with same value of attribute a)
2. for each class, find nodes that are witnessed by that class

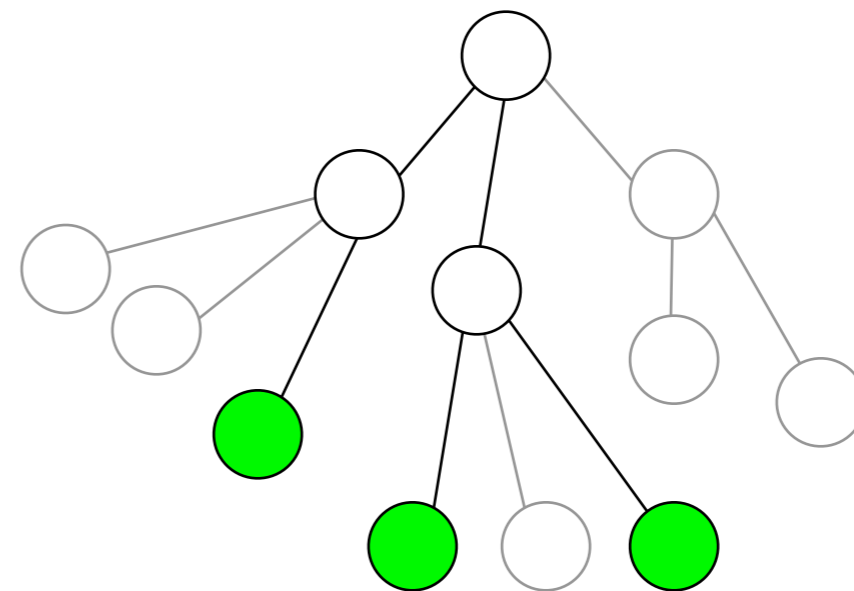
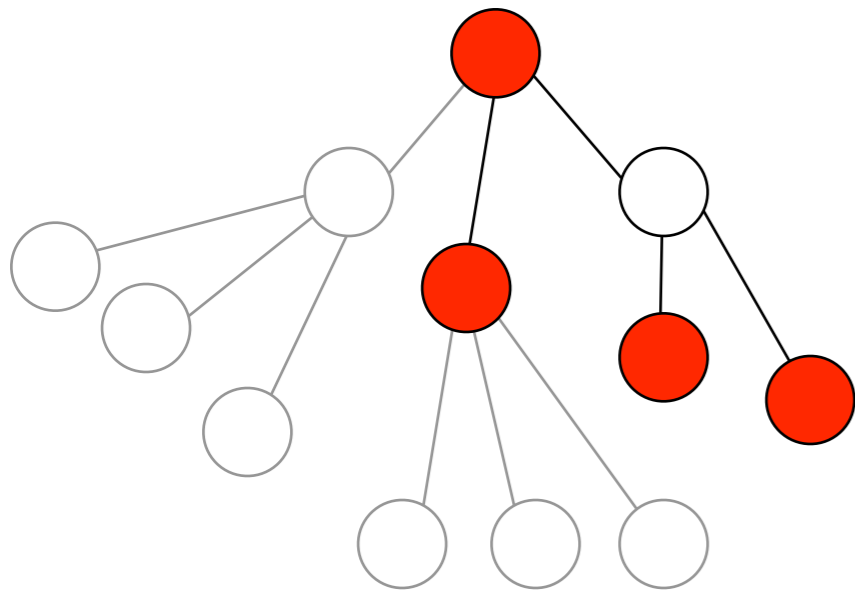
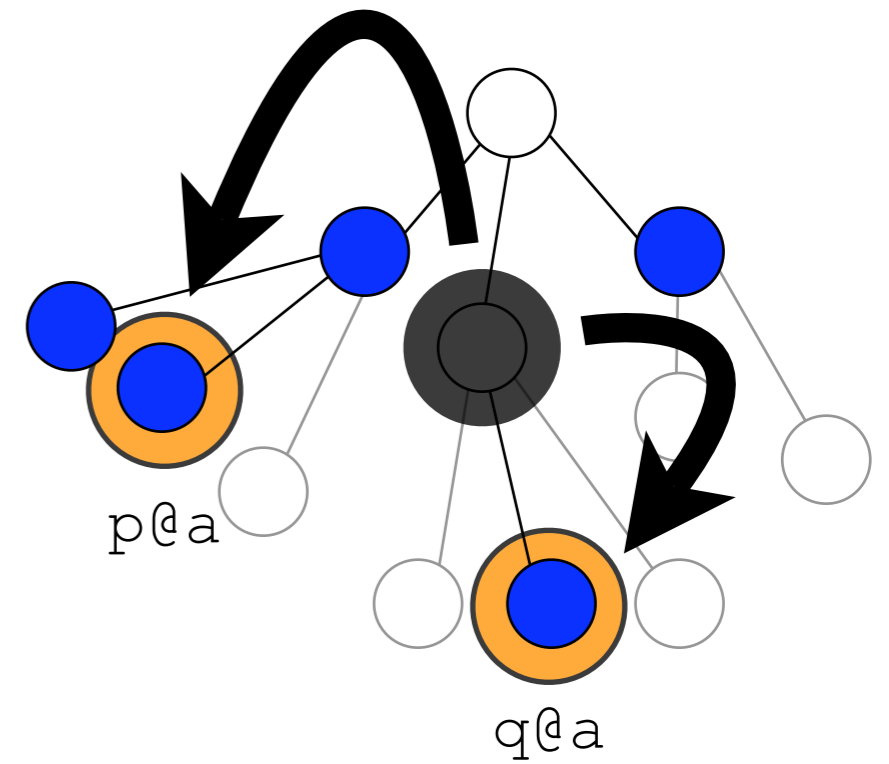
## High level overview



find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes  
(class = set of nodes with same value of attribute a)
2. for each class, find nodes that are witnessed by that class

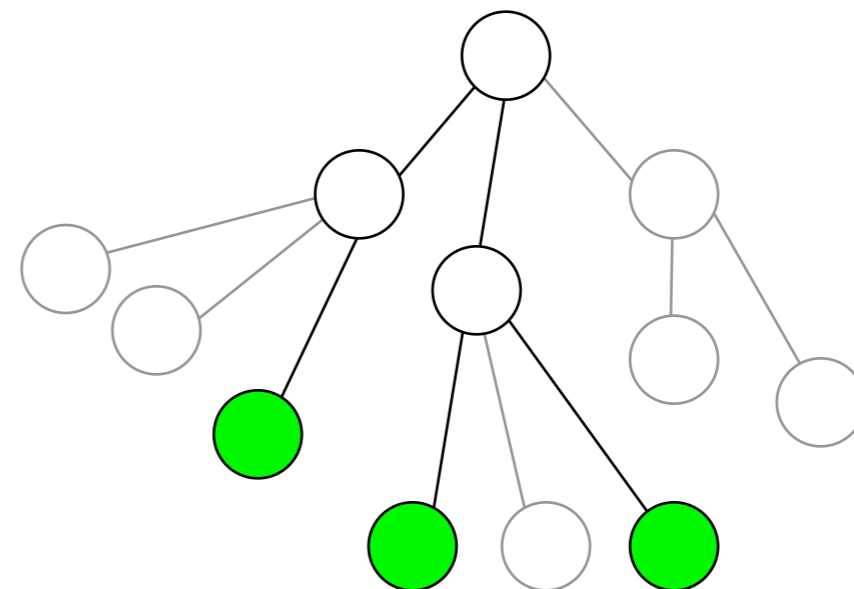
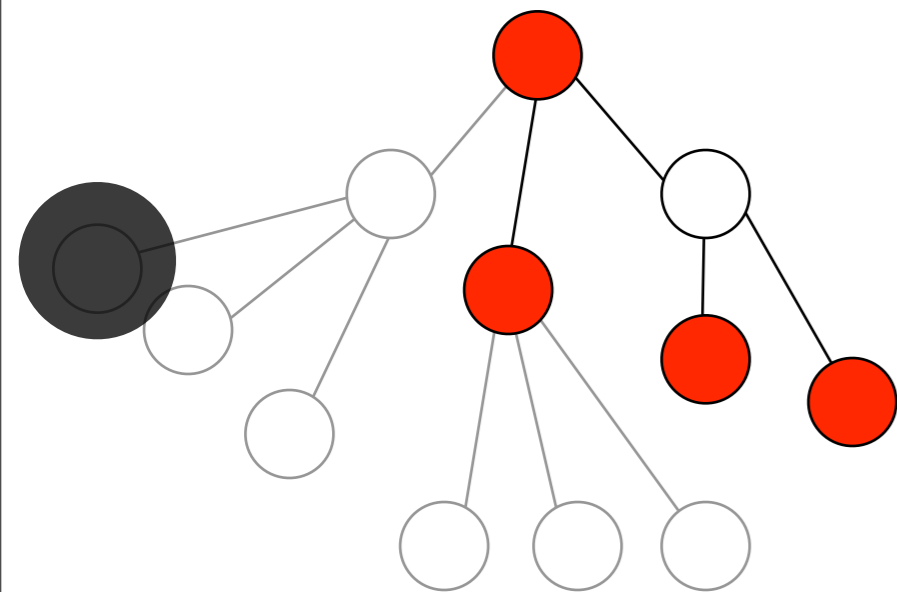
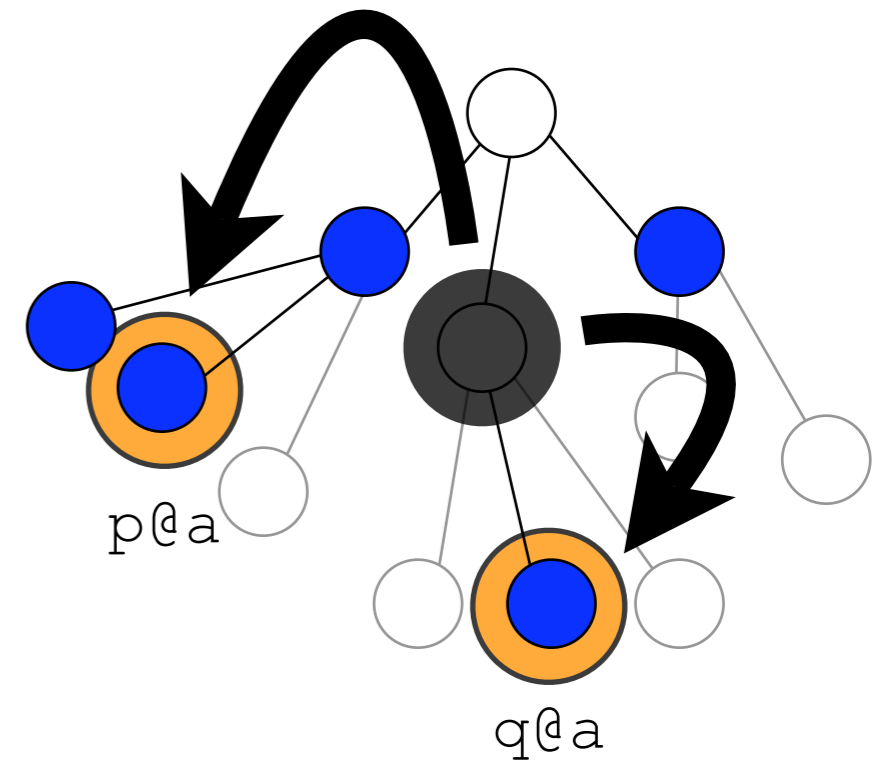
## High level overview



find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes  
(class = set of nodes with same value of attribute a)
2. for each class, find nodes that are witnessed by that class

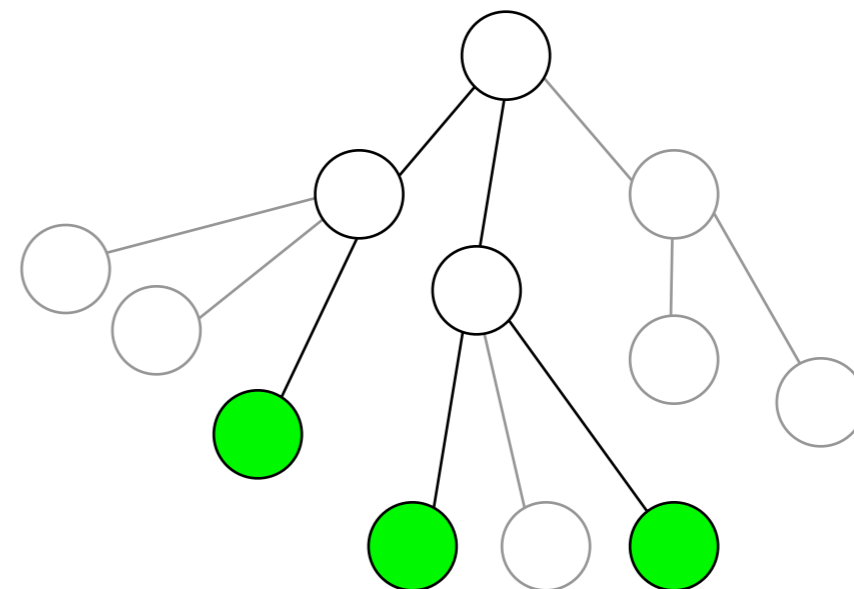
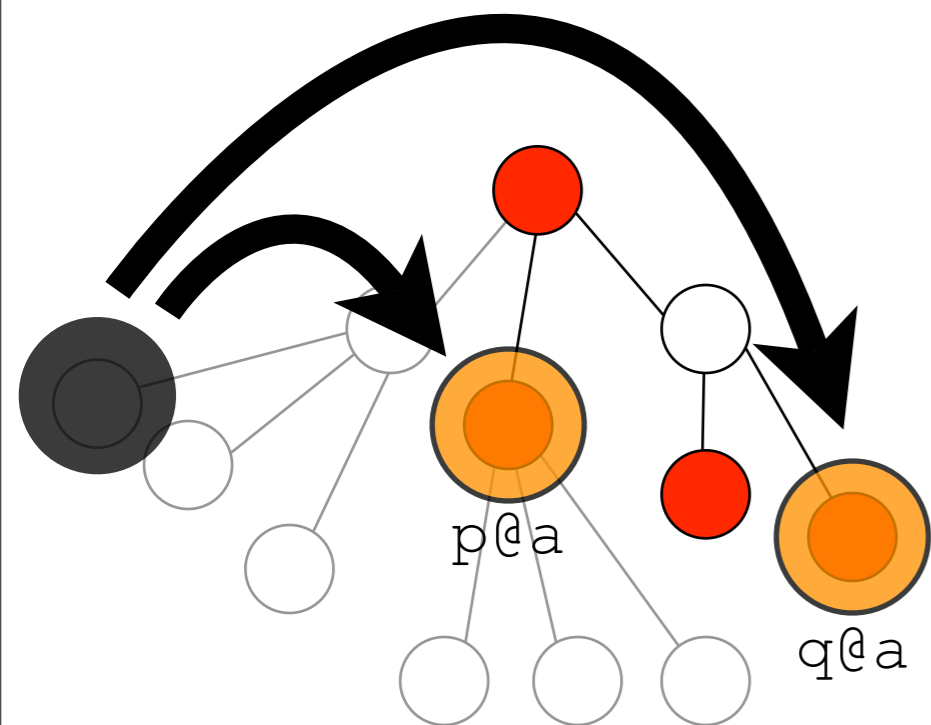
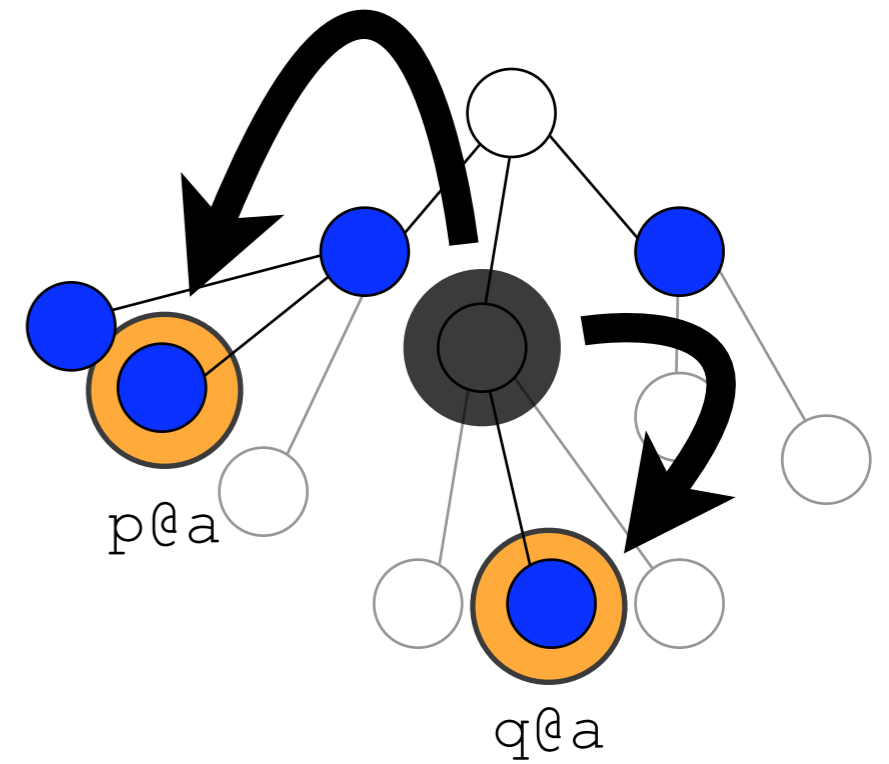
## High level overview



find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes  
(class = set of nodes with same value of attribute a)
2. for each class, find nodes that are witnessed by that class

## High level overview



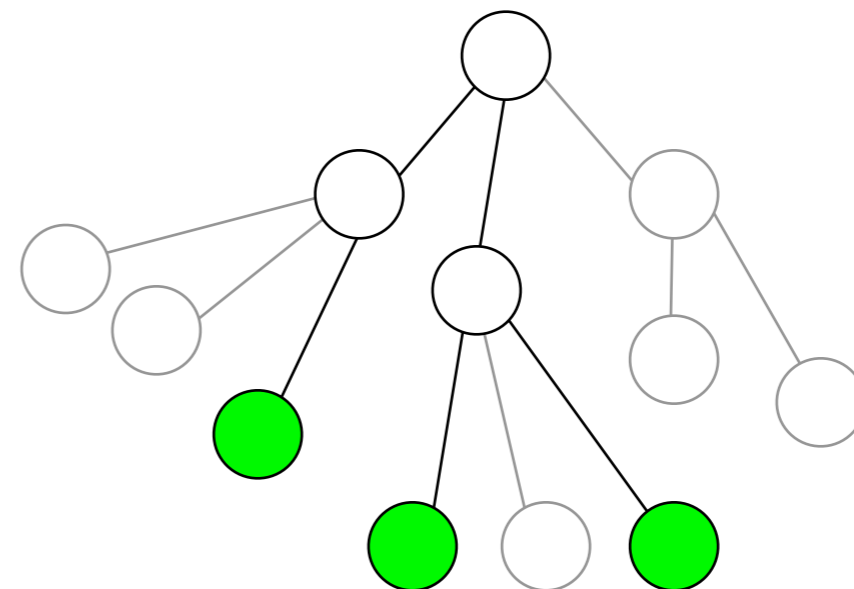
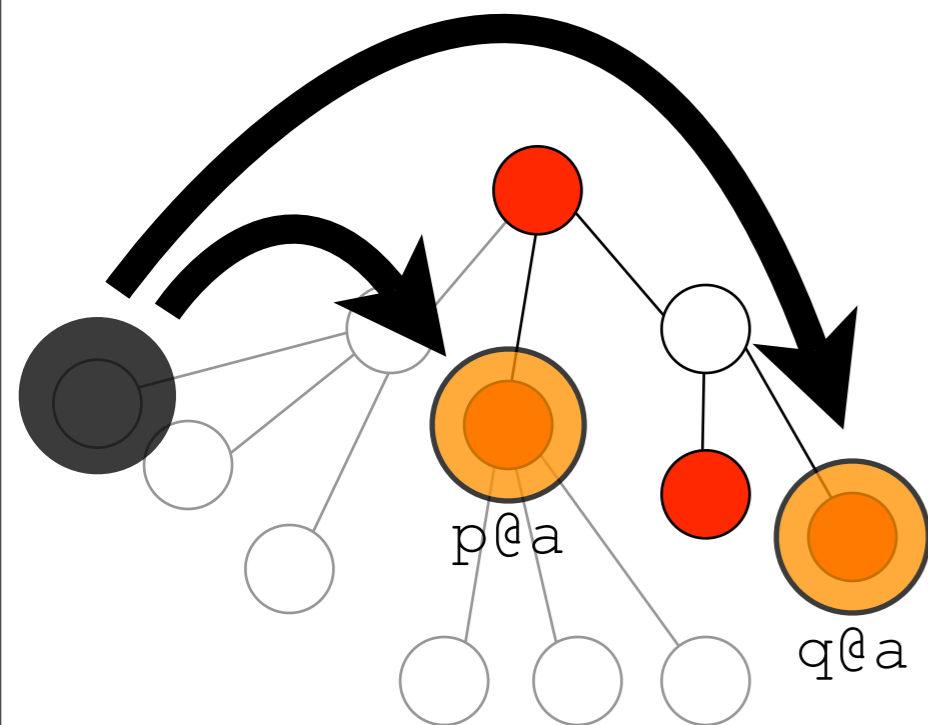
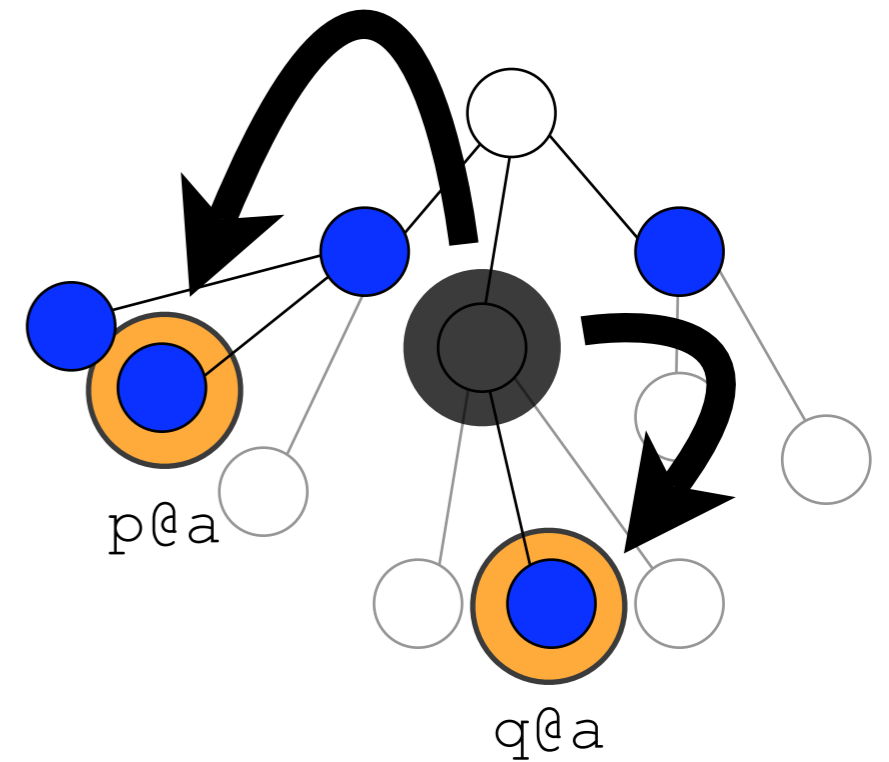
find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes  
(class = set of nodes with same value of attribute a)
2. for each class, find nodes that are witnessed by that class

**Goal: avoid repetition**

- do a constant number of operations per node
- or at least logarithmic

High level overview



# High level overview

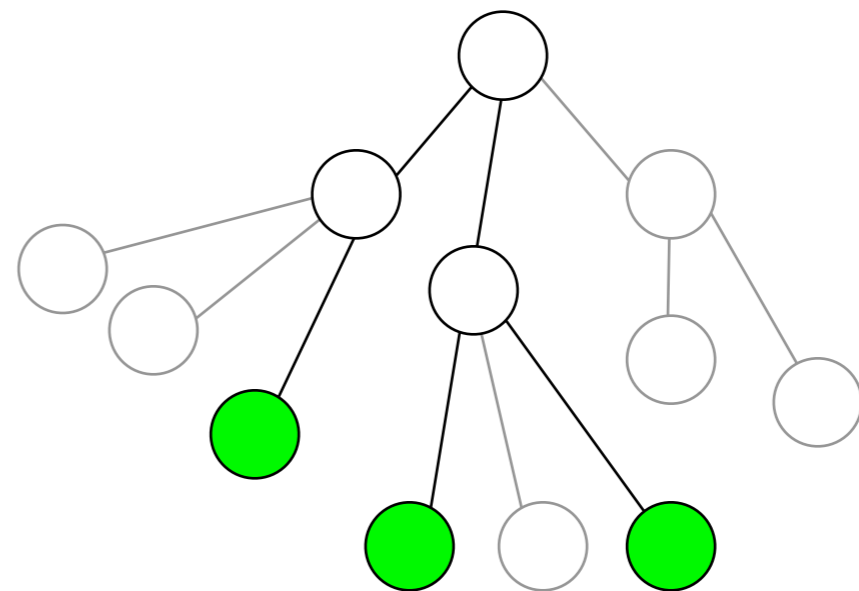
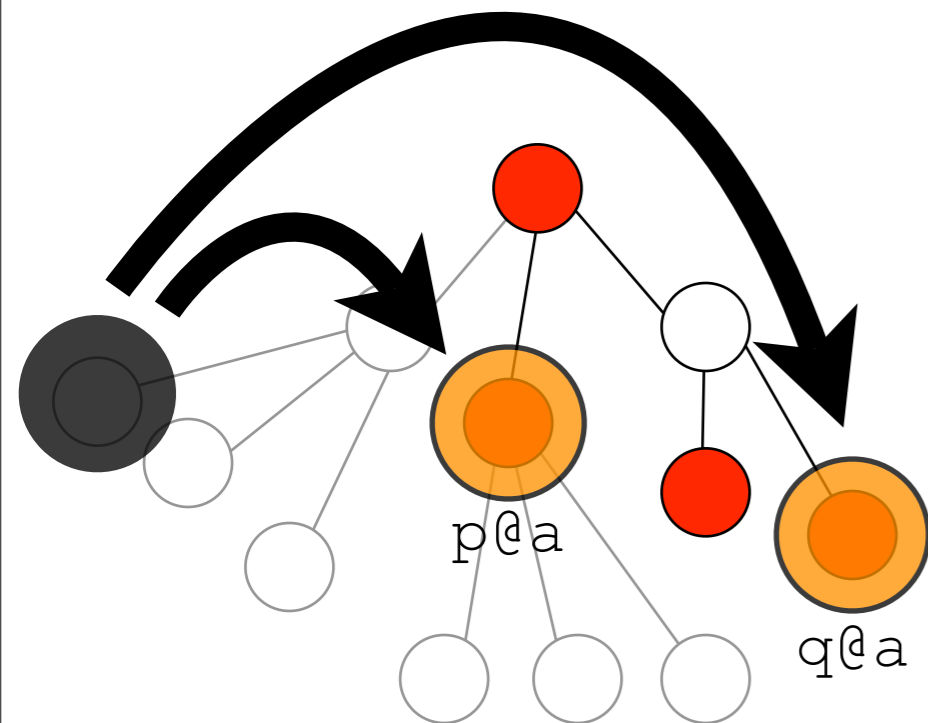
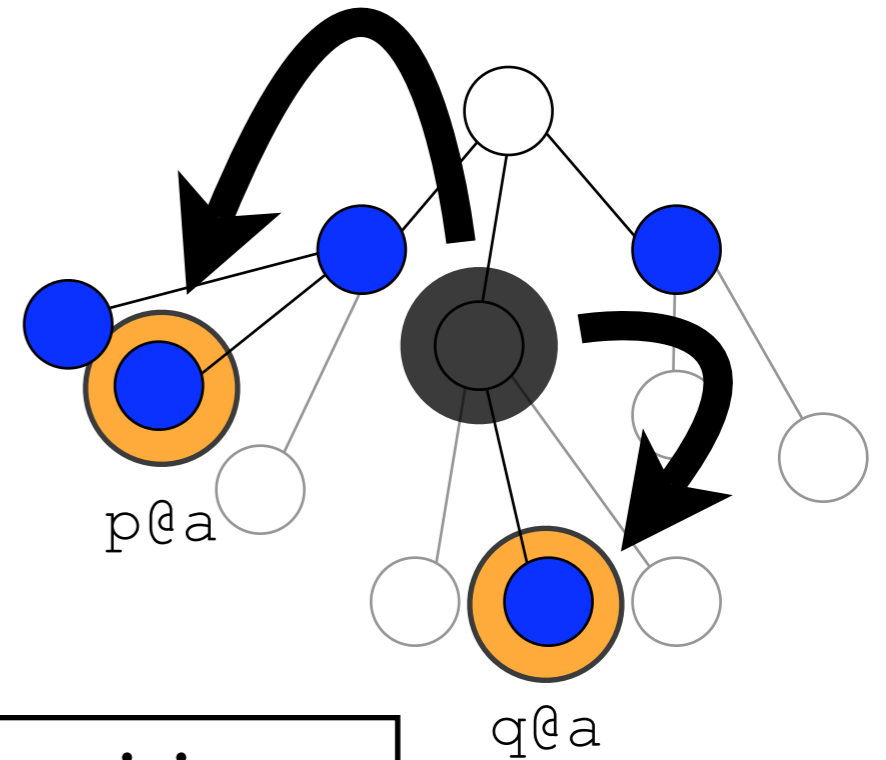
find nodes that satisfy  $p@a = q@a$

1. decompose trees into classes  
(class = set of nodes with same value of attribute a)
2. for each class, find nodes that are witnessed by that class

Goal: avoid repetition

- do a constant number of operations per node
- or at least logarithmic

Using Simon decompositions,  
a fancy algebraic result



What is the Simon decomposition?

$L$  a regular word language.

Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$L$  a regular word language.

Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$A$  an automaton recognizing  $L$ , and  $Q$  its state space.

Each word  $u$  induces a transformation on states  $\delta(u) : Q \rightarrow Q$ .

$L$  a regular word language.

Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$A$  an automaton recognizing  $L$ , and  $Q$  its state space.

Each word  $u$  induces a transformation on states  $\delta(u) : Q \rightarrow Q$ .

a b

b a

a b

b a

$L$  a regular word language.

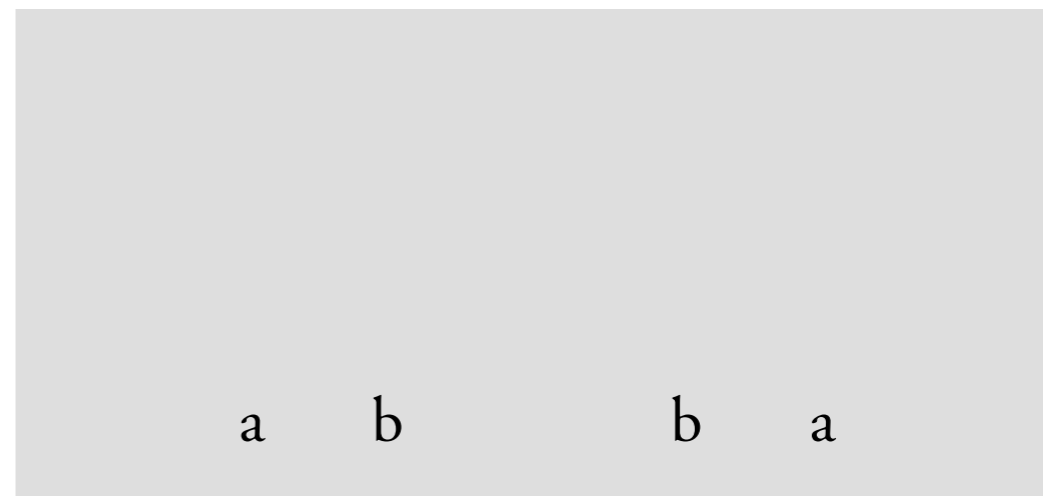
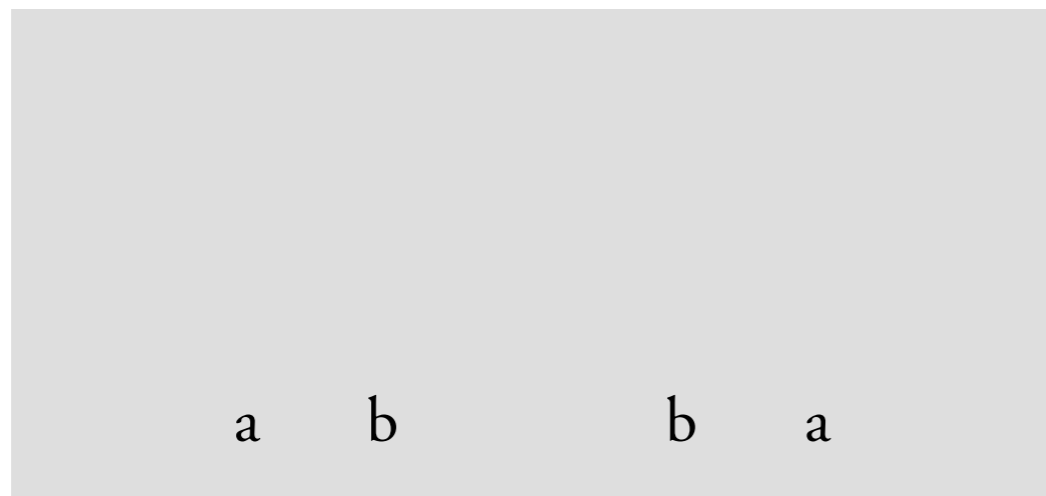
Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$A$  an automaton recognizing  $L$ , and  $Q$  its state space.

Each word  $u$  induces a transformation on states  $\delta(u) : Q \rightarrow Q$ .

$\frac{1}{2}$



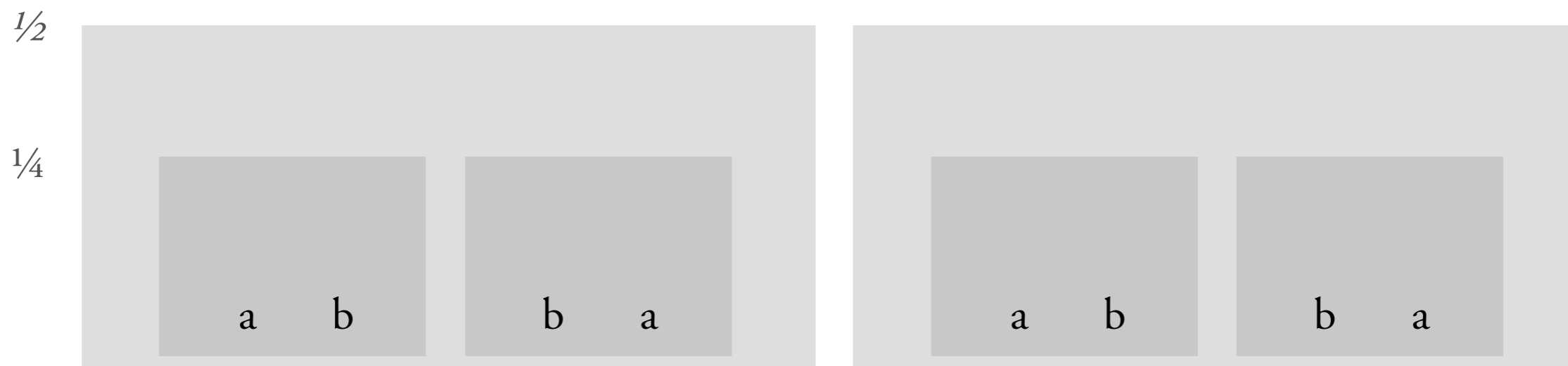
$L$  a regular word language.

Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$A$  an automaton recognizing  $L$ , and  $Q$  its state space.

Each word  $u$  induces a transformation on states  $\delta(u) : Q \rightarrow Q$ .



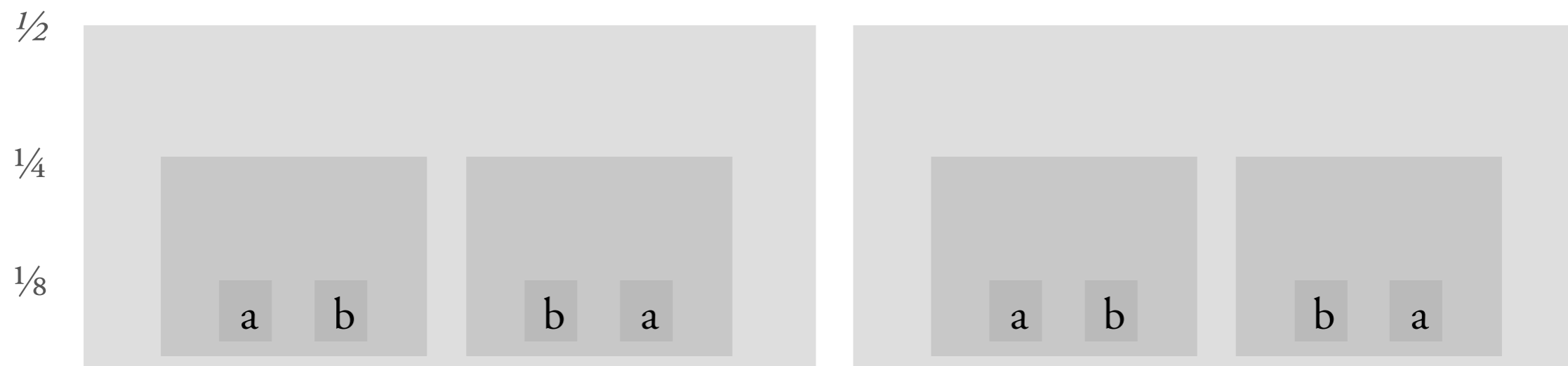
$L$  a regular word language.

Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$A$  an automaton recognizing  $L$ , and  $Q$  its state space.

Each word  $u$  induces a transformation on states  $\delta(u) : Q \rightarrow Q$ .



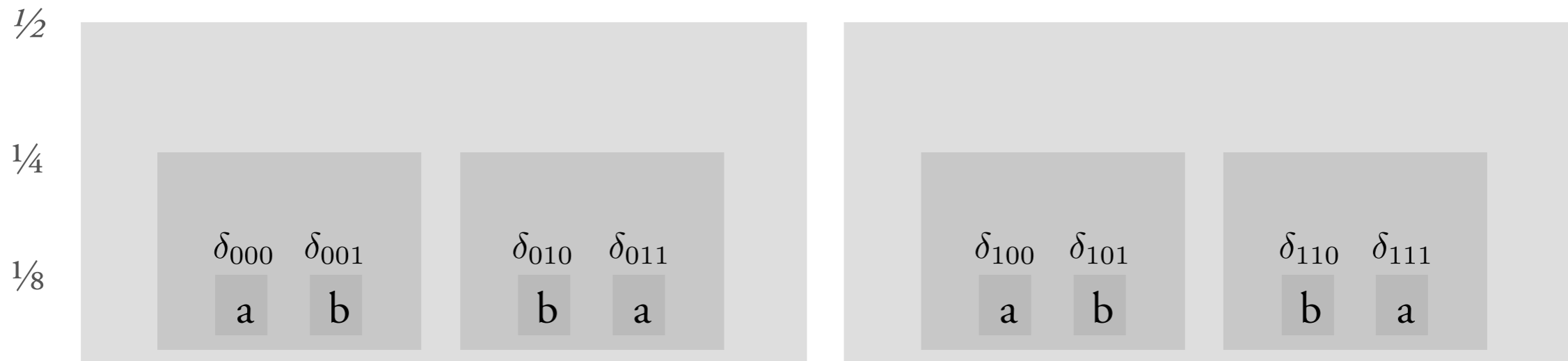
$L$  a regular word language.

Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$A$  an automaton recognizing  $L$ , and  $Q$  its state space.

Each word  $u$  induces a transformation on states  $\delta(u) : Q \rightarrow Q$ .



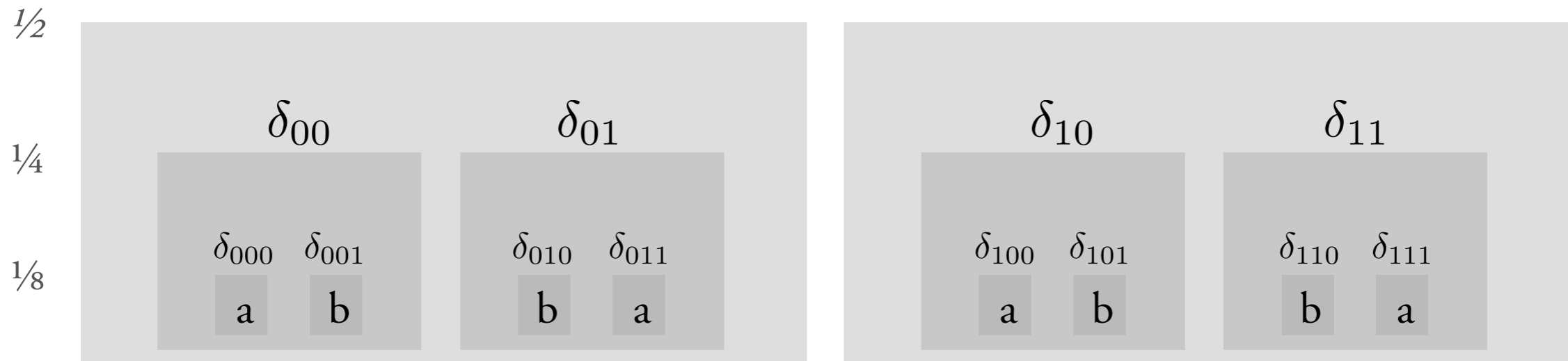
$L$  a regular word language.

Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$A$  an automaton recognizing  $L$ , and  $Q$  its state space.

Each word  $u$  induces a transformation on states  $\delta(u) : Q \rightarrow Q$ .



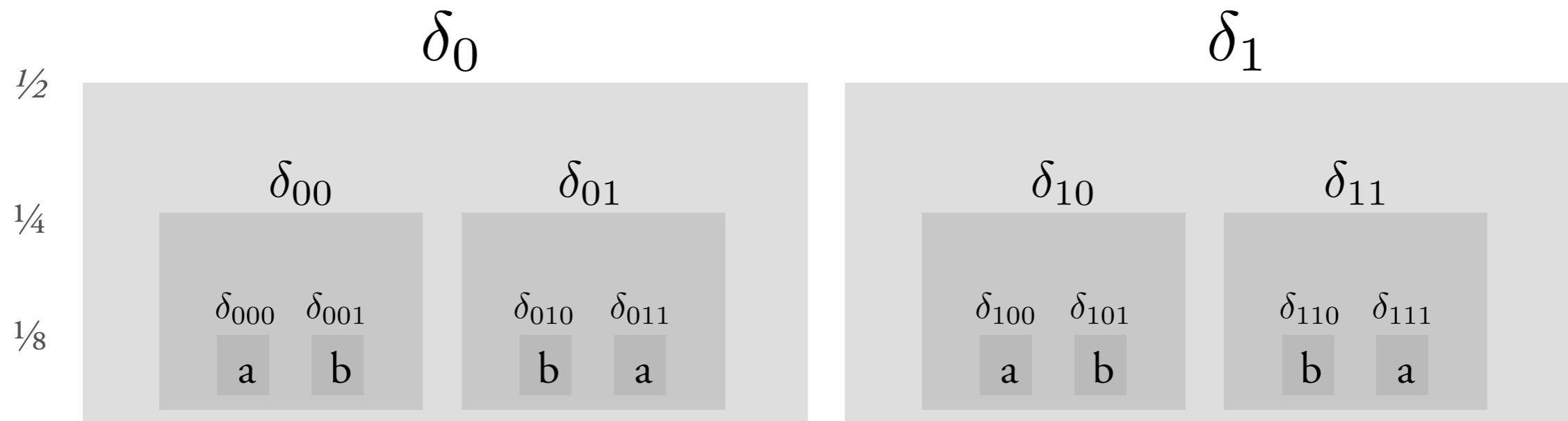
$L$  a regular word language.

Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$A$  an automaton recognizing  $L$ , and  $Q$  its state space.

Each word  $u$  induces a transformation on states  $\delta(u) : Q \rightarrow Q$ .

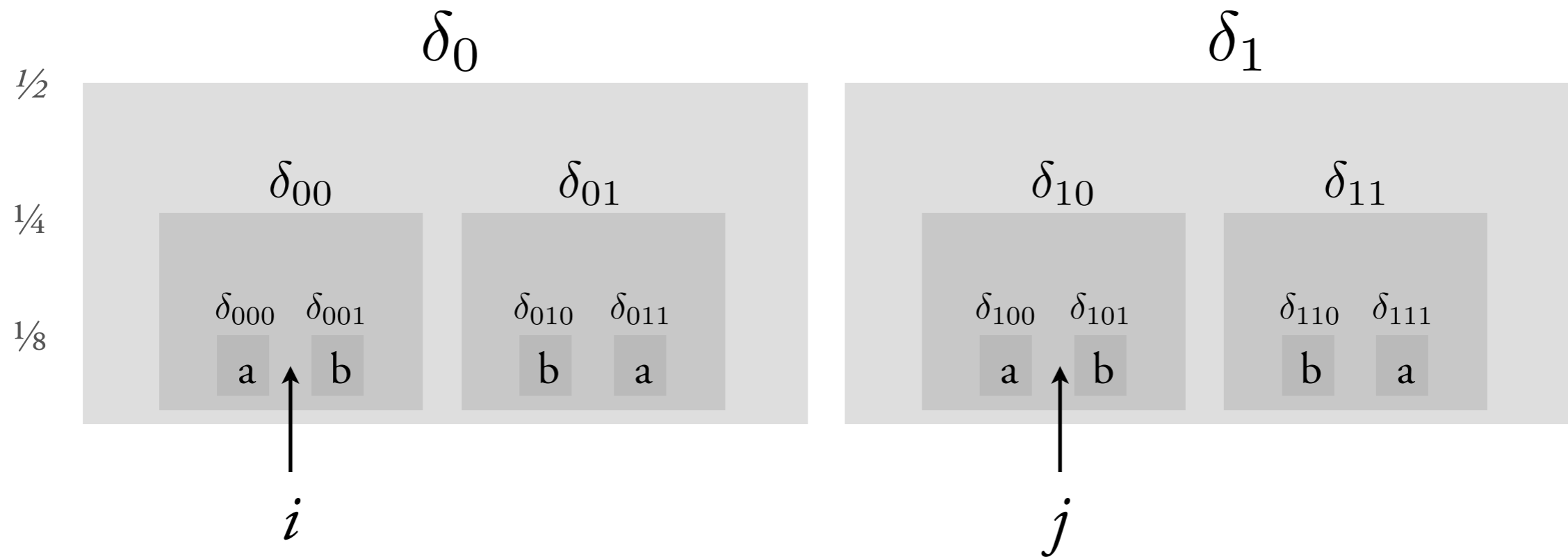


$L$  a regular word language.

Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$A$  an automaton recognizing  $L$ , and  $Q$  its state space.  
Each word  $u$  induces a transformation on states  $\delta(u) : Q \rightarrow Q$ .

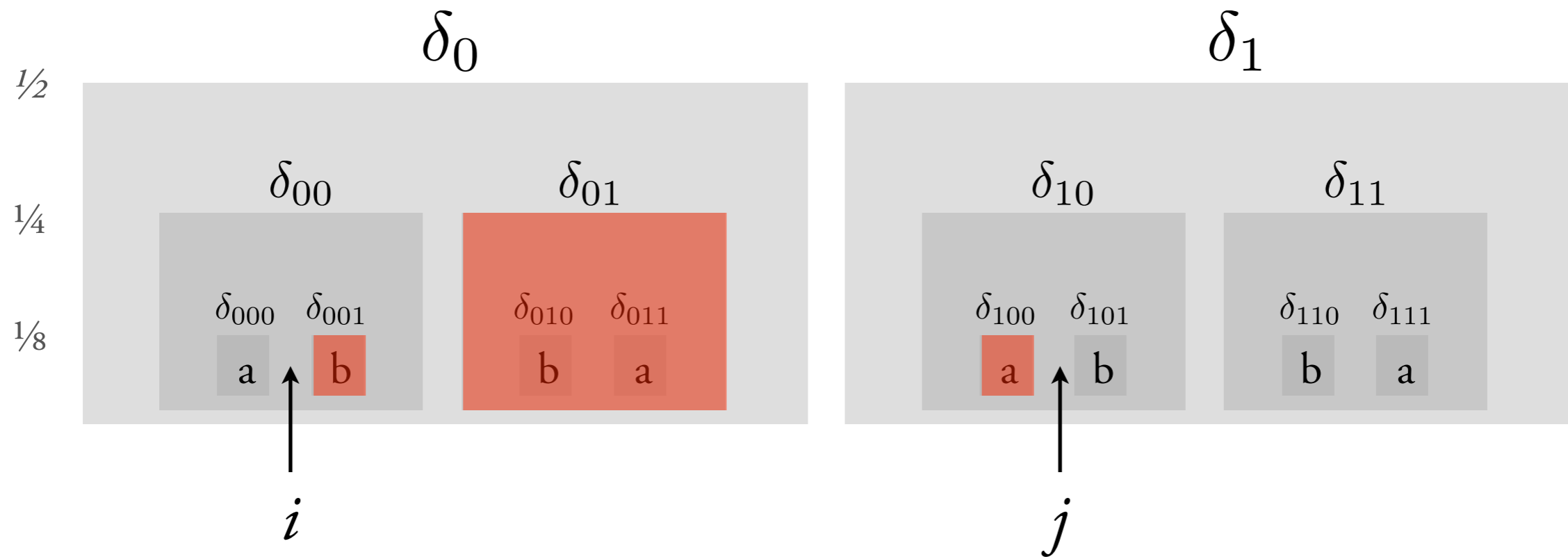


$L$  a regular word language.

Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$A$  an automaton recognizing  $L$ , and  $Q$  its state space.  
Each word  $u$  induces a transformation on states  $\delta(u) : Q \rightarrow Q$ .

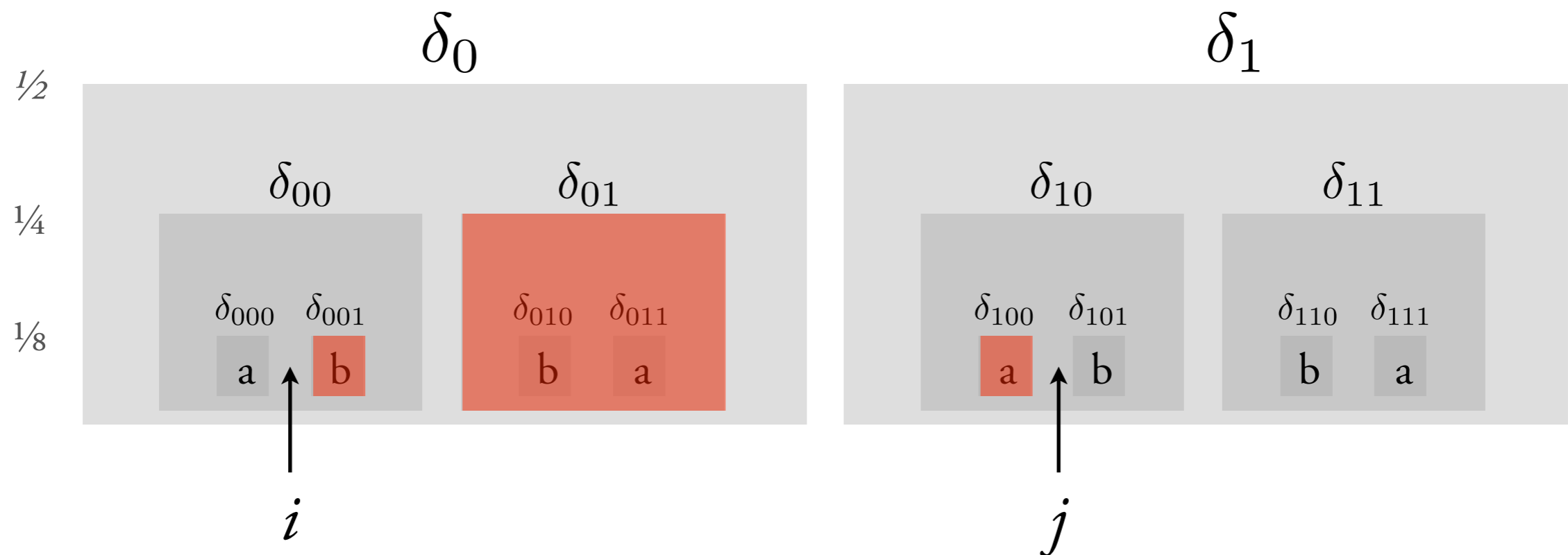


$L$  a regular word language.

Do a linear time precomputation on  $w = a_1 a_2 \dots a_n$

For any infix, membership  $a_i \dots a_j \in L$  can be computed in time  $\log n$

$A$  an automaton recognizing  $L$ , and  $Q$  its state space.  
Each word  $u$  induces a transformation on states  $\delta(u) : Q \rightarrow Q$ .



Big news: Simon decomposition does this with constant depth!

Back to XPath evaluation...

To simplify, consider a special case of XPath:

abaabaabbabababbababbababbababbbababbbabbababbabbabaababbabba

To simplify, consider a special case of XPath:

- words not trees

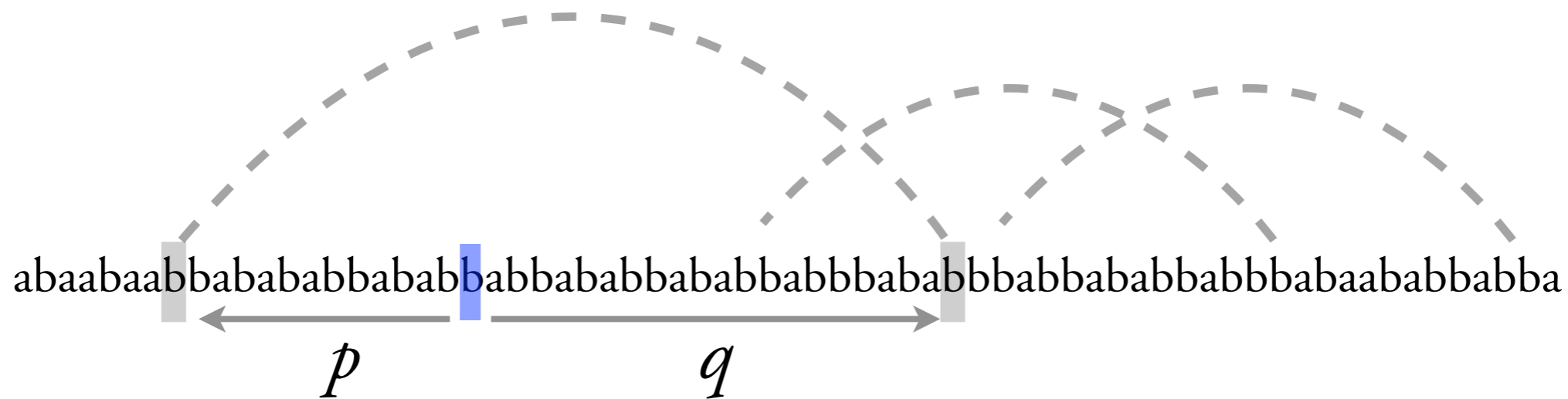












To simplify, consider a special case of XPath:

- words not trees
- a test  $p@a = q@a$
- each attribute value appears exactly twice
- programs  $p, q$  have no nested tests, except label tests





**Problem.** Fix a set of tag names  $\Sigma$  and regular word languages  $L, K \subseteq \Sigma^*$

**Input.**  $a_1 \cdots a_n \in \Sigma^*$   $E \subseteq \{1, \dots, n\}^2$  matching

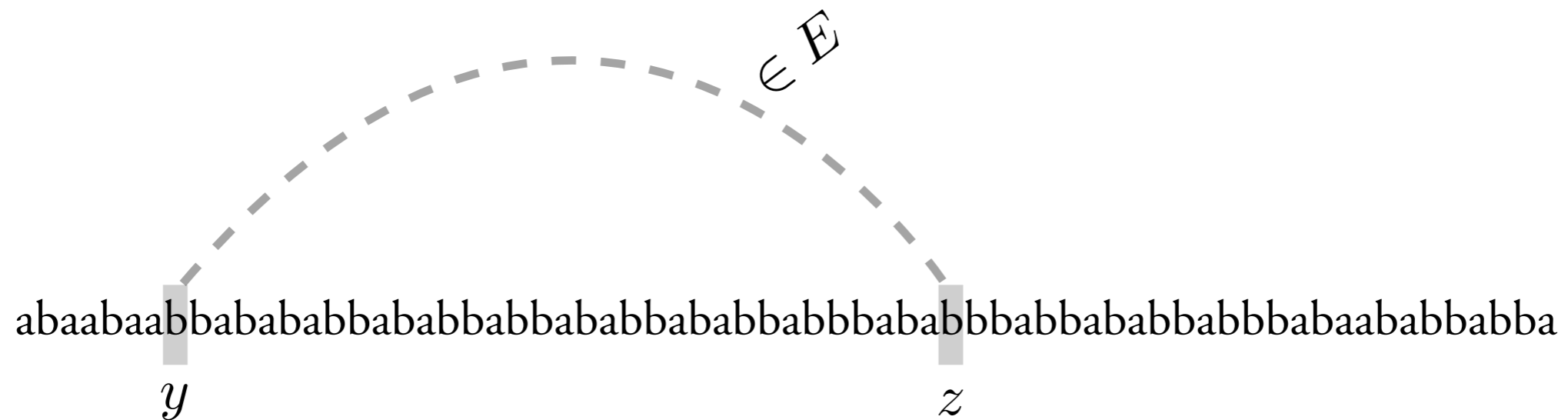
**Output.** Set of nodes  $x$  with

abaabaabbabababbababbababbababbbababbbabbababbabbabaababbabba

**Problem.** Fix a set of tag names  $\Sigma$  and regular word languages  $L, K \subseteq \Sigma^*$

**Input.**  $a_1 \cdots a_n \in \Sigma^*$      $E \subseteq \{1, \dots, n\}^2$  matching

**Output.** Set of nodes  $x$  with



**Naive algorithm.**

For every match  $(y, z) \in E$











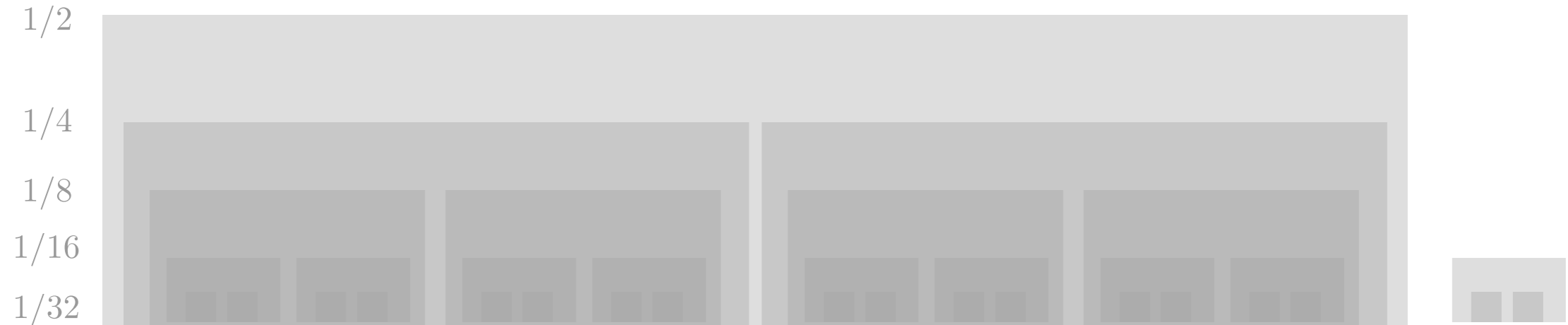


**Problem.** Fix a set of tag names  $\Sigma$  and regular word languages  $L, K \subseteq \Sigma^*$

**Input.**  $a_1 \cdots a_n \in \Sigma^*$   $E \subseteq \{1, \dots, n\}^2$  matching

**Output.** Set of nodes  $x$  with

abaabaabbabababbababbababbababbbbababbbbababbababbbbabababbbbabababbbbababaababbabba



**Divide and conquer dynamic algorithm.**













# Summary

# Summary

- We evaluate XPath queries with linear time data complexity, improving previous quadratic algorithms.  
(the constant is exponential in the query, because we use semigroup theory)

# Summary

- We evaluate XPath queries with linear time data complexity, improving previous quadratic algorithms.  
(the constant is exponential in the query, because we use semigroup theory)
- Works for both unary and binary queries

# Summary

- We evaluate XPath queries with linear time data complexity, improving previous quadratic algorithms.  
(the constant is exponential in the query, because we use semigroup theory)
- Works for both unary and binary queries
- Semigroups a good tool for efficient query evaluation

# Summary

- We evaluate XPath queries with linear time data complexity, improving previous quadratic algorithms.  
(the constant is exponential in the query, because we use semigroup theory)
- Works for both unary and binary queries
- Semigroups a good tool for efficient query evaluation

## Future work

# Summary

- We evaluate XPath queries with linear time data complexity, improving previous quadratic algorithms.  
(the constant is exponential in the query, because we use semigroup theory)
- Works for both unary and binary queries
- Semigroups a good tool for efficient query evaluation

## Future work

- Preliminary results indicate that semigroups can be avoided, and the constant becomes polynomial in the query.

# Summary

- We evaluate XPath queries with linear time data complexity, improving previous quadratic algorithms.  
(the constant is exponential in the query, because we use semigroup theory)
- Works for both unary and binary queries
- Semigroups a good tool for efficient query evaluation

## Future work

- Preliminary results indicate that semigroups can be avoided, and the constant becomes polynomial in the query.
- We want to investigate more of XPath, and other languages

Let  $A$  be an automaton with state space  $Q$   
Two rules for splitting words.

Let  $A$  be an automaton with state space  $Q$   
Two rules for splitting words.

**Simon Theorem.** For fixed  $A$ , there is a splitting depth  $K$ , such that every word can be split in depth  $K$  down to single letters.

Let  $A$  be an automaton with state space  $Q$

Two rules for splitting words.

Rule 1.

split into two parts

*abaabbbababbbabba* *bbabbbabbbabba*

**Simon Theorem.** For fixed  $A$ , there is a splitting depth  $K$ , such that every word can be split in depth  $K$  down to single letters.

Let  $A$  be an automaton with state space  $Q$

Two rules for splitting words.

Rule 1. split into two parts

*abaabbbababbabba* *bbabbbabbbabbaba*

Rule 2. split into many parts, each with the same transformation  $\delta : Q \rightarrow Q$

*abaab* *bbababb* *babba* *bba* *bbbabb* *babba* *ba*  
 $\delta$   $\delta$   $\delta$   $\delta$   $\delta$   $\delta$   $\delta$

**Simon Theorem.** For fixed  $A$ , there is a splitting depth  $K$ , such that every word can be split in depth  $K$  down to single letters.

Let  $A$  be an automaton with state space  $Q$

Two rules for splitting words.

Rule 1. split into two parts

*abaabbbababbabba* *bbabbbabbbabbaba*

Rule 2. split into many parts, each with the same transformation  $\delta : Q \rightarrow Q$

*abaab* *bbababb* *babba* *bba* *bbbabb* *babba* *ba*  
 $\delta$   $\delta$   $\delta$   $\delta$   $\delta$   $\delta$   $\delta$

$$\delta \circ \delta = \delta$$

**Simon Theorem.** For fixed  $A$ , there is a splitting depth  $K$ , such that every word can be split in depth  $K$  down to single letters.