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

attribute name

attribute name

```
<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>
```

document node,
i.e. opening tag

XML Document

attribute name

attribute name

```
<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>
```

document node,
i.e. opening tag

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

XML Document

attribute name

attribute name

```
<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>
```

document node,
i.e. opening tag

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

XML Document

attribute name

attribute name

```
<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>
```

document node,
i.e. opening tag

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

XML Document

attribute name

attribute name

```
<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>
```

document node,
i.e. opening tag

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

XML Document

attribute name

attribute name

```
<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>
```

document node,
i.e. opening tag

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

XML Document

attribute name

attribute name

```
<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>
```

document node,
i.e. opening tag

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

XML Document

attribute name

attribute name

```
<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>
```

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

attribute name

attribute name

```
<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>
```

document node,
i.e. opening tag

```
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` ≠ *p*@`b` are tests.

Programs - select node pairs.

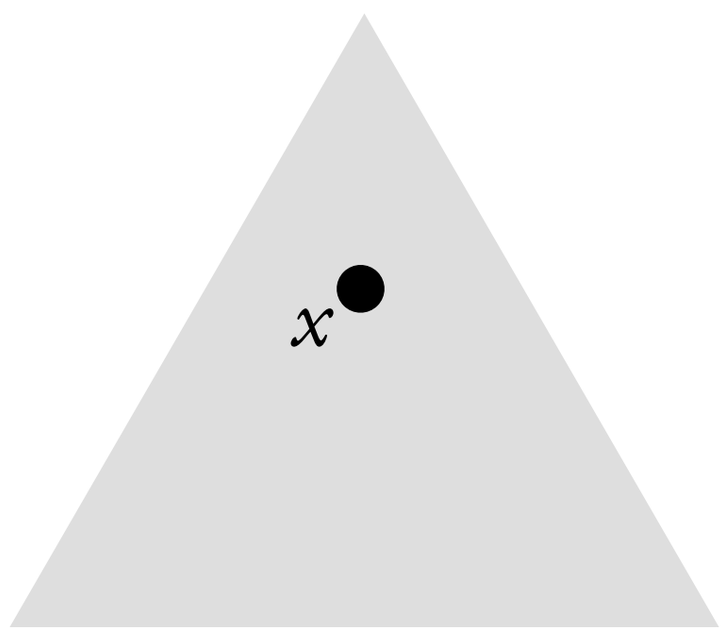
FOXPath

- `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` ≠ *p*@`b` are tests.

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



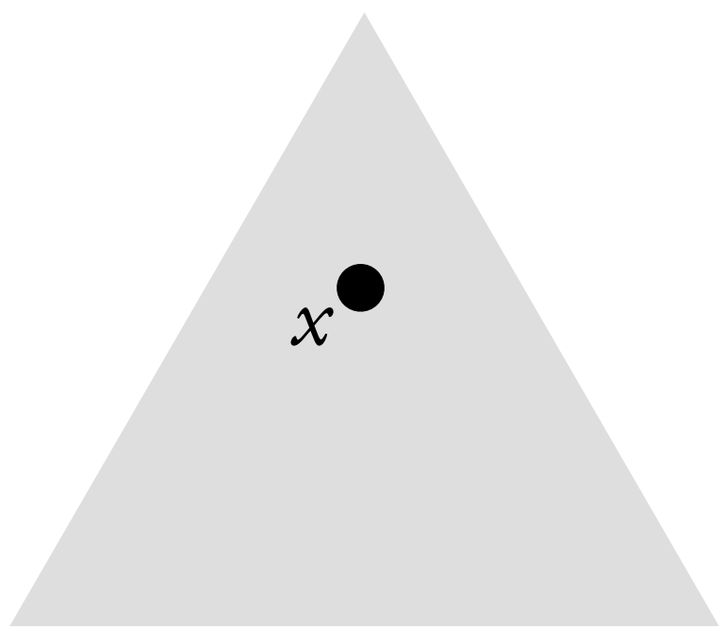
Programs - select node pairs.

- `child`, `parent`, `next-sibling`, `prev-sibling`, `descendant`, etc.
- any regular expression on programs is a program, e.g. `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` ≠ *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.

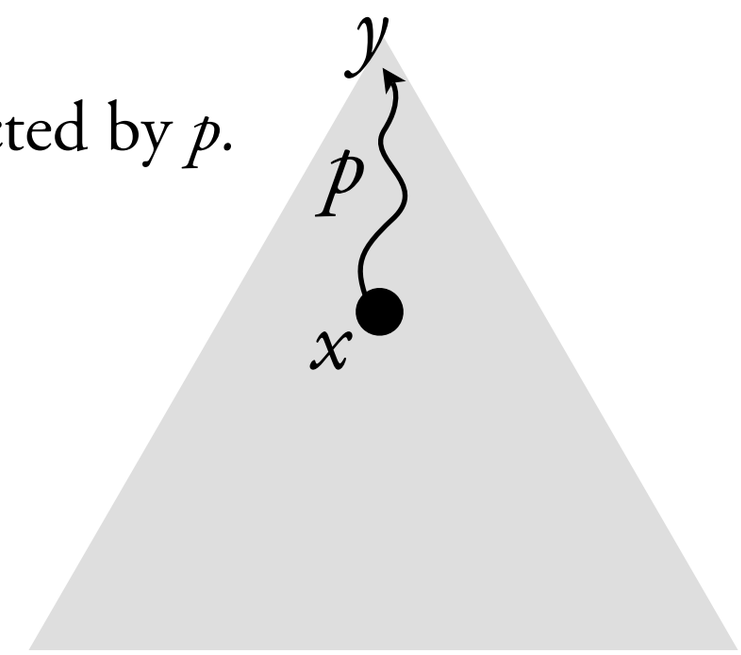
- `child`, `parent`, `next-sibling`, `prev-sibling`, `descendant`, etc.
- any regular expression on programs is a program, e.g. `child`*
- if *t* is a test, then *[t]* is a program that selects *(x,x)* if node *x* satisfies *t*

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` ≠ *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.

FOXPath

- `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*

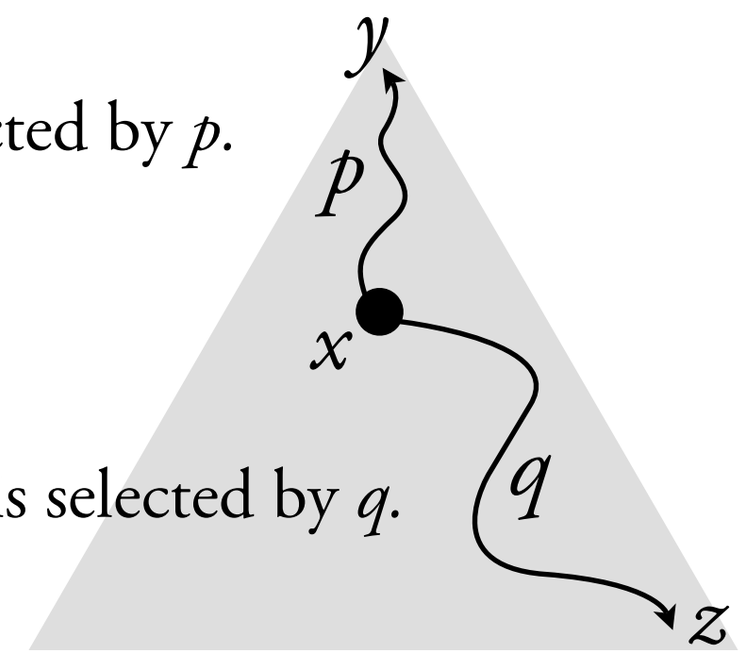
Tests - select single nodes.

- 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` ≠ *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*.

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



Programs - select node pairs.

FOXPath

- `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*

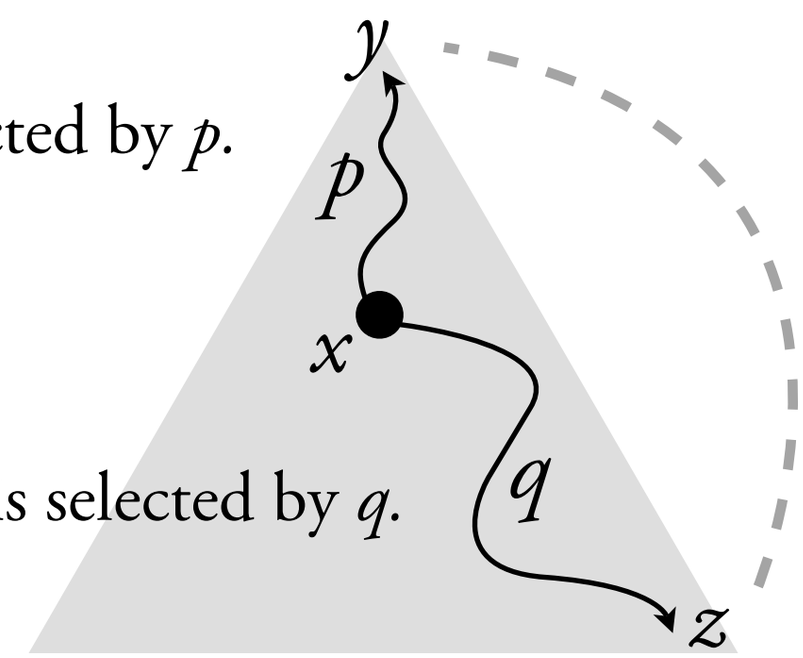
Tests - select single nodes.

- 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` ≠ *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*.

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



the attribute values
y@`a` and *z*@`b` are the same.

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*

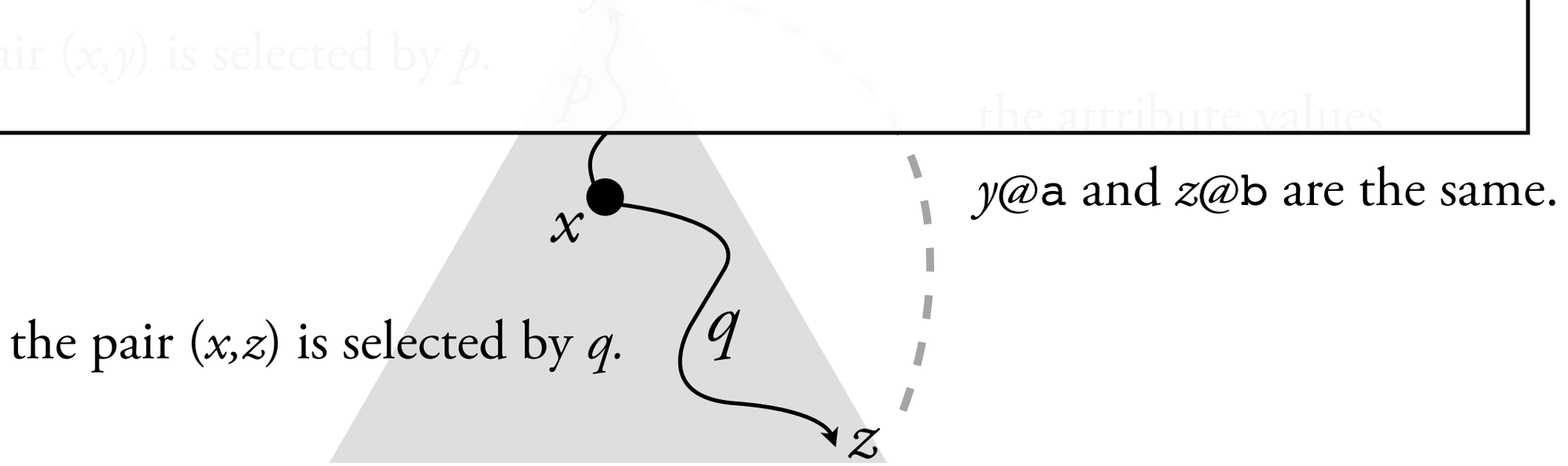
Tests - select single nodes.

- 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 ≠ *p*@b are tests.

Thm.

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

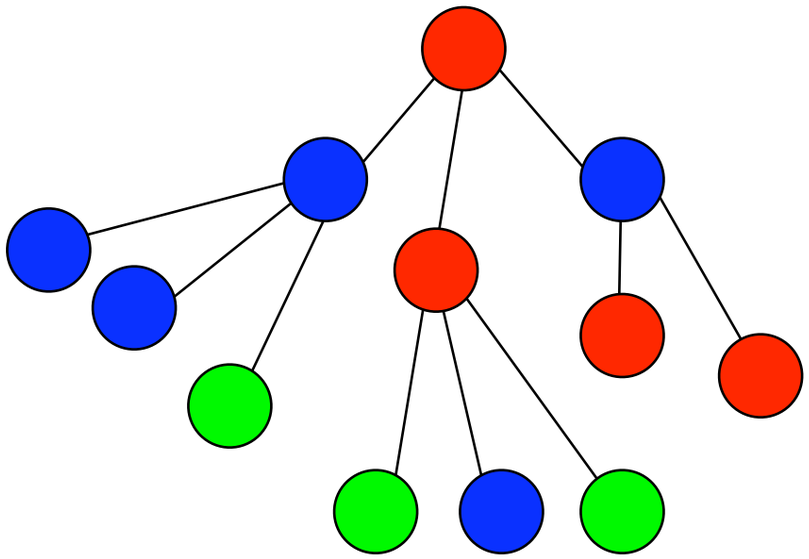
The set of nodes in *d* selected by *t* can be computed in time $O(|d|2^{|t|})$ as well as in time $O(|d|\log(|d|)|t|^2)$



High level overview

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

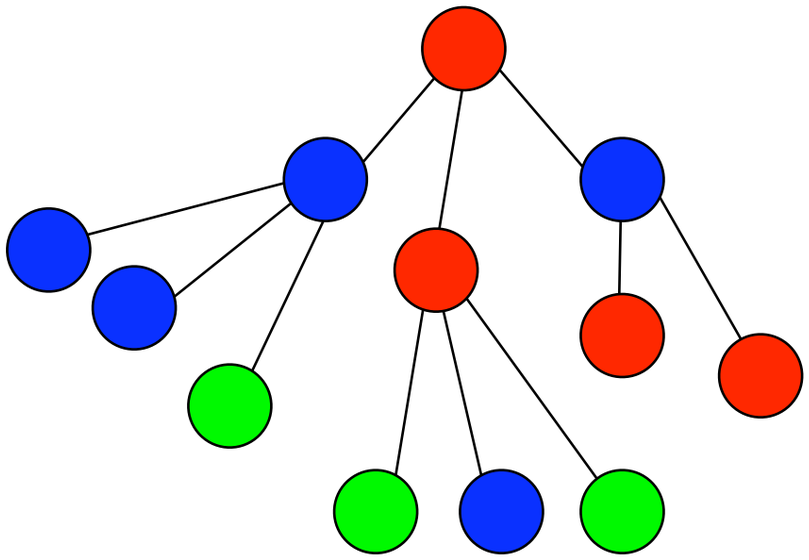
1. decompose trees into classes
   (class = set of nodes with same value of atribute a)

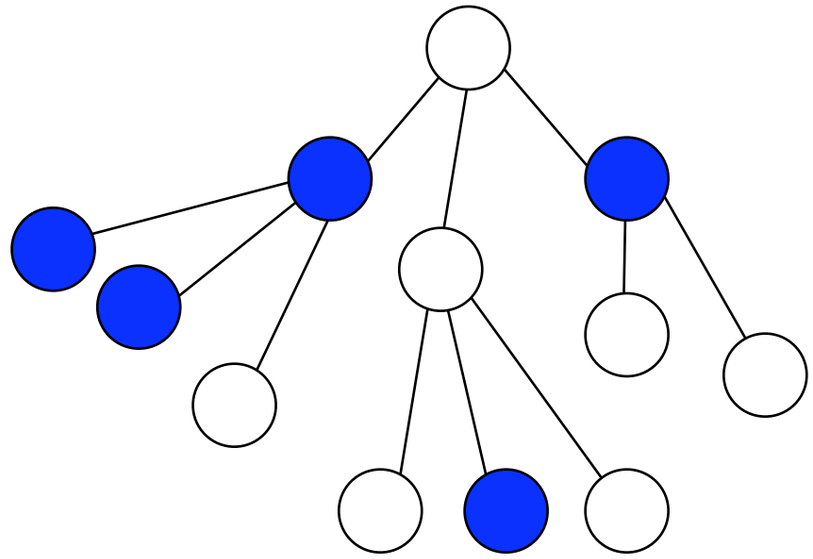


High level overview

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

1. decompose trees into classes
   (class = set of nodes with same value of atribute a)

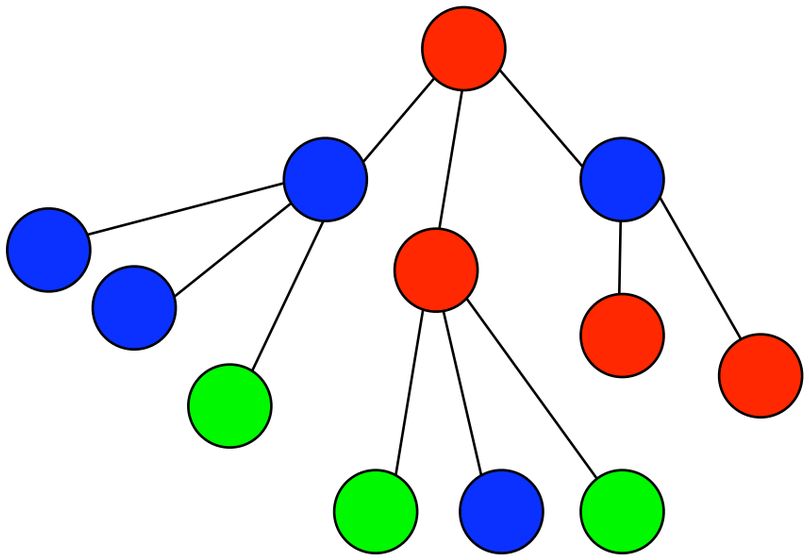


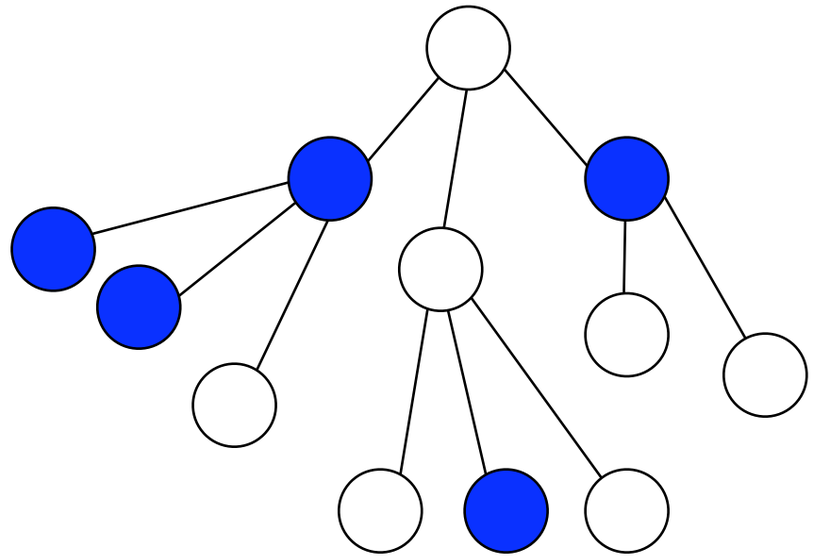


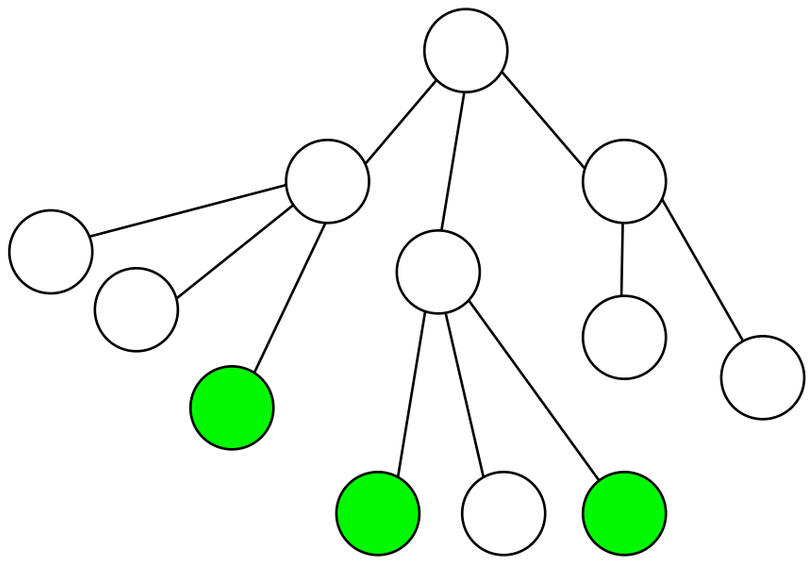
High level overview

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

1. decompose trees into classes
   (class = set of nodes with same value of atribute a)



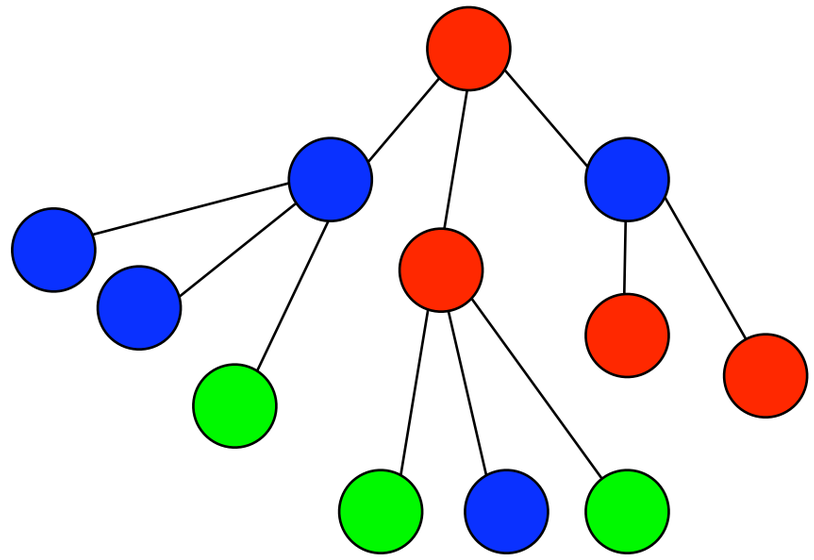


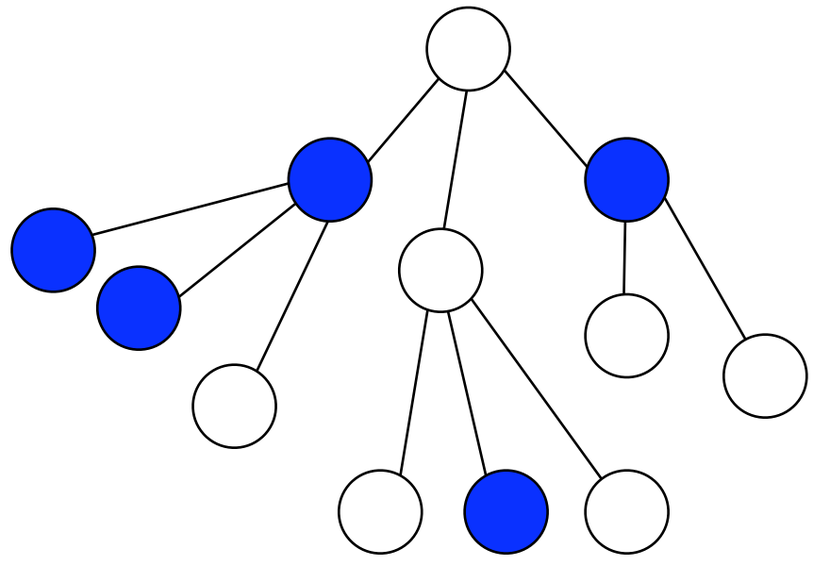


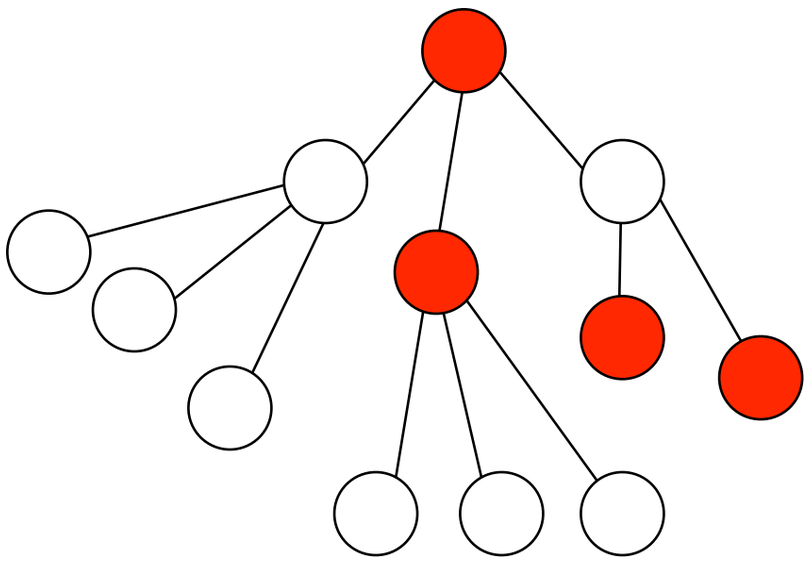
High level overview

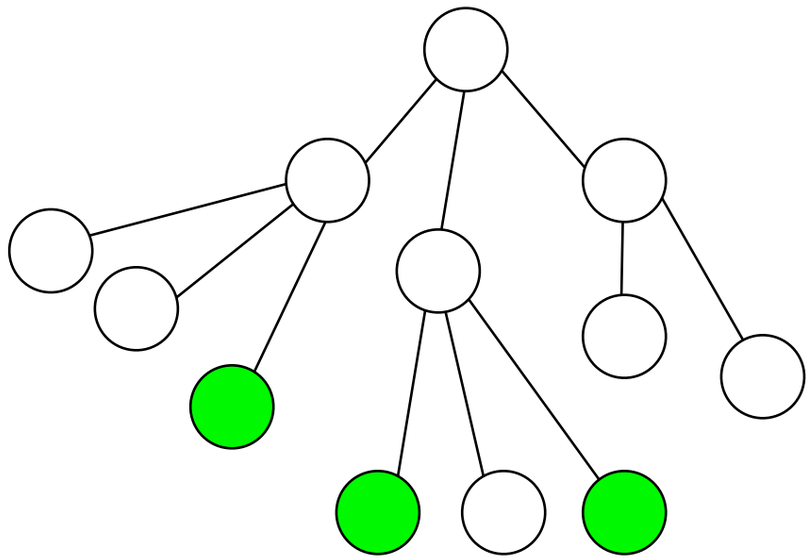
find nodes that satisfy `p@a = q@a`

1. decompose trees into classes
   (class = set of nodes with same value of atribute a)





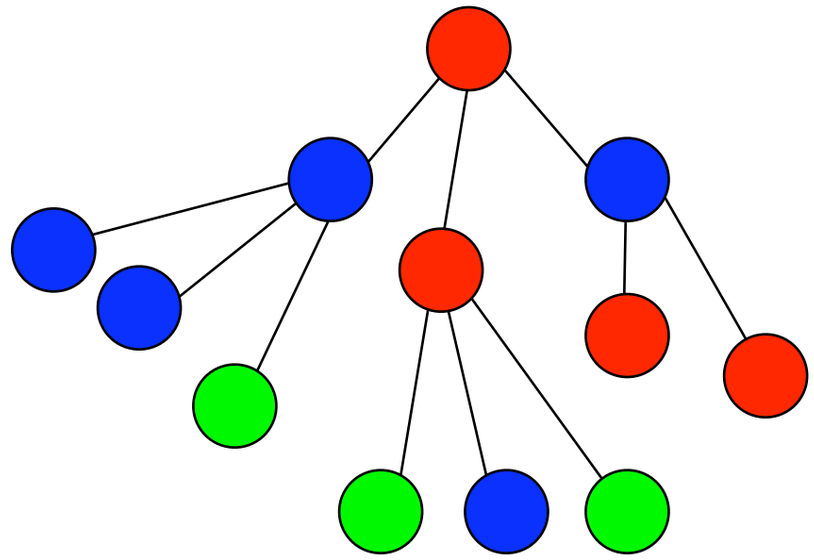


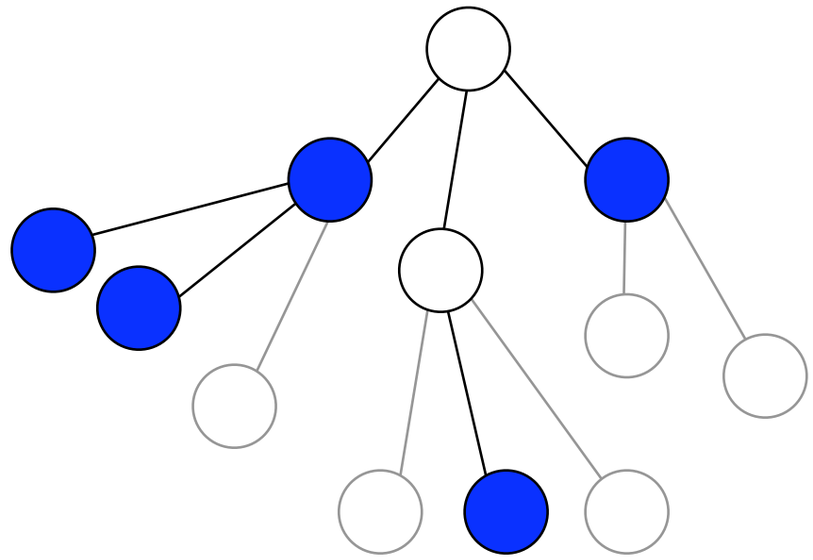


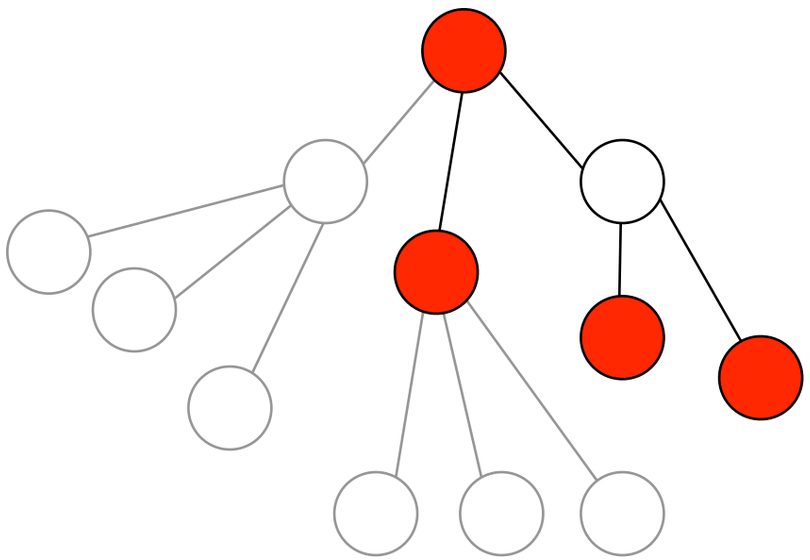
High level overview

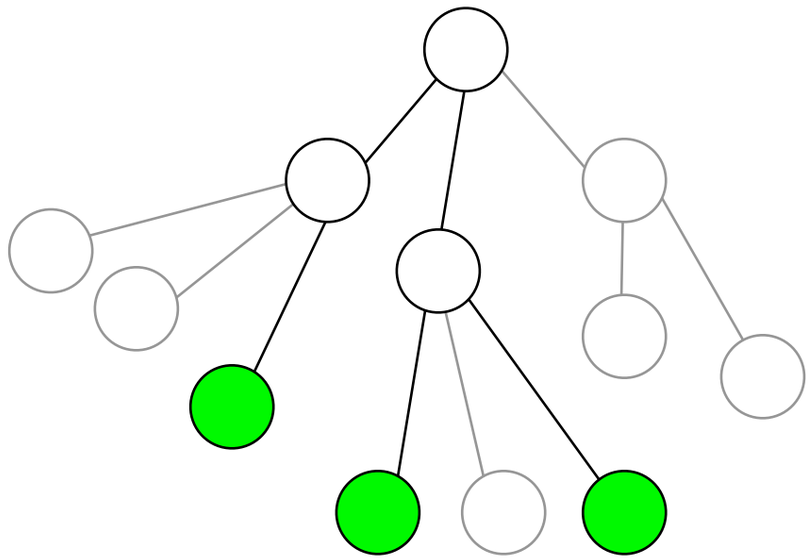
find nodes that satisfy `p@a = q@a`

1. decompose trees into classes
   (class = set of nodes with same value of atribute a)





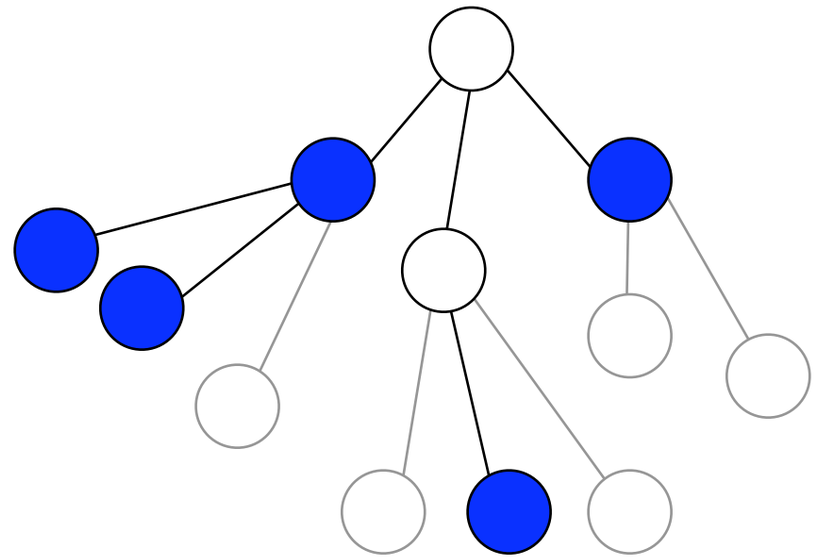


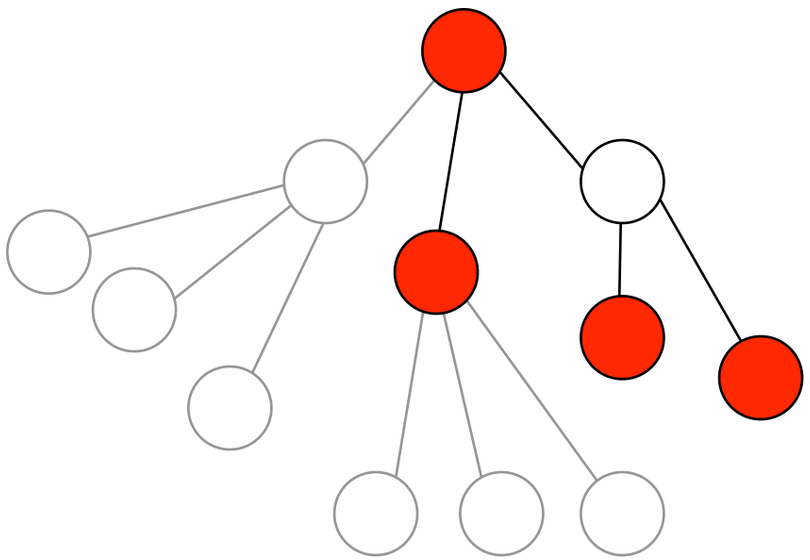


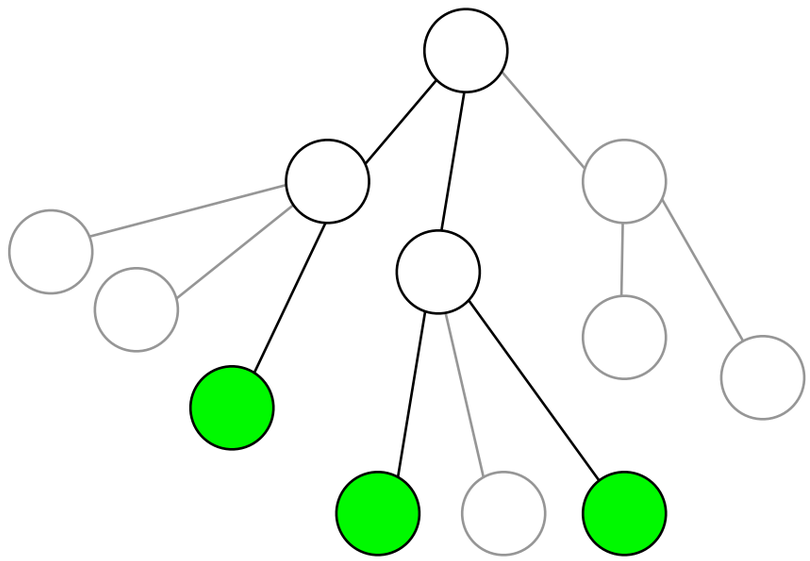
High level overview

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

1. decompose trees into classes
   (class = set of nodes with same value of atribute a)





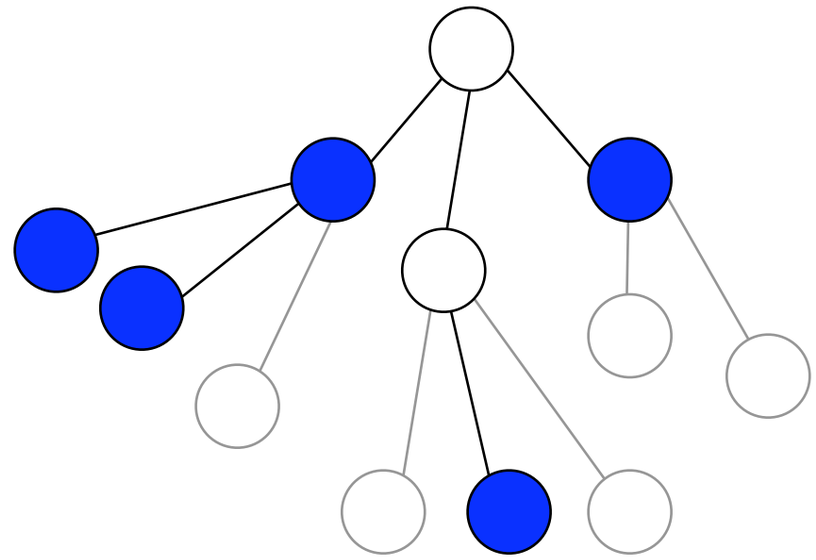


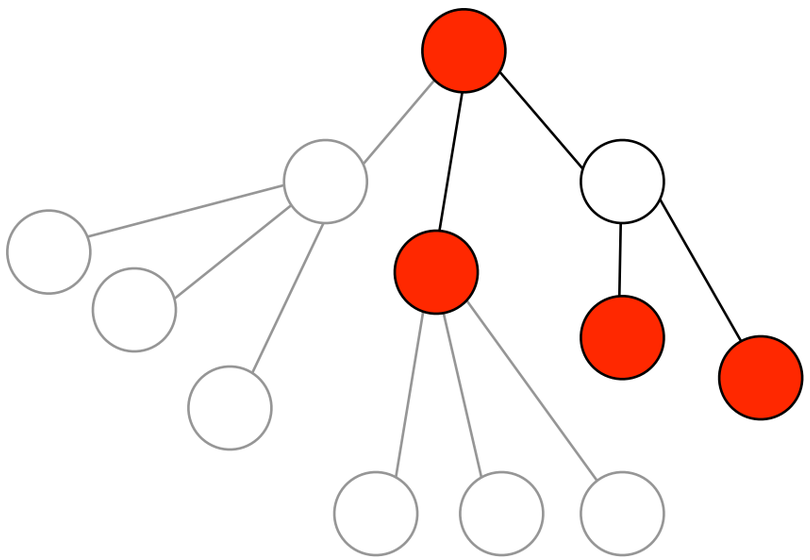
High level overview

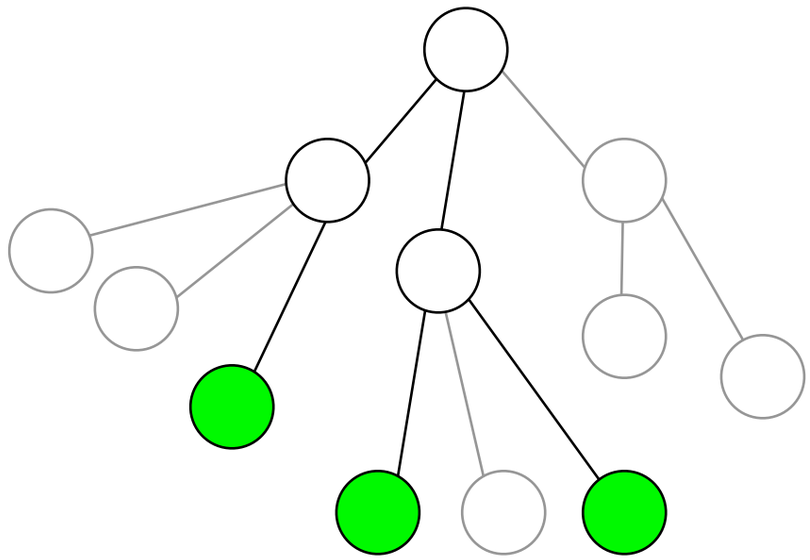
find nodes that satisfy `p@a = q@a`

1. decompose trees into classes
   (class = set of nodes with same value of atribute a)

2. for each class, find nodes that are witnessed by
   that class





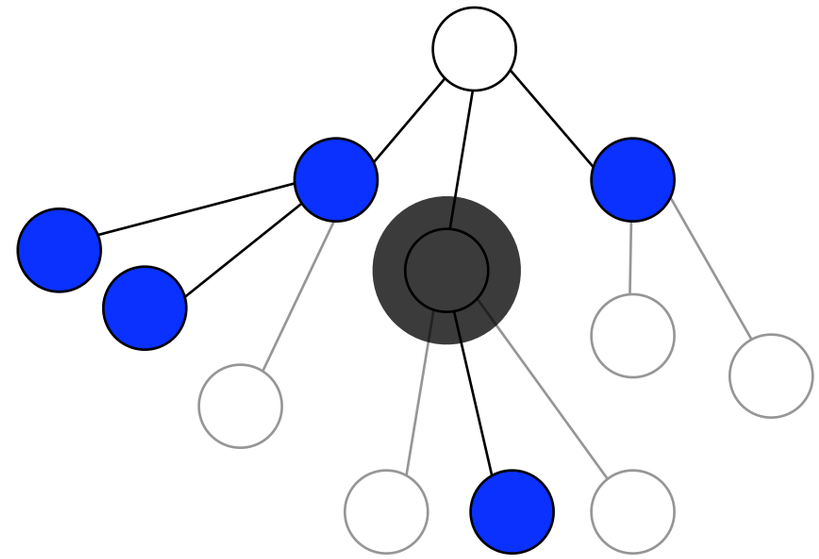


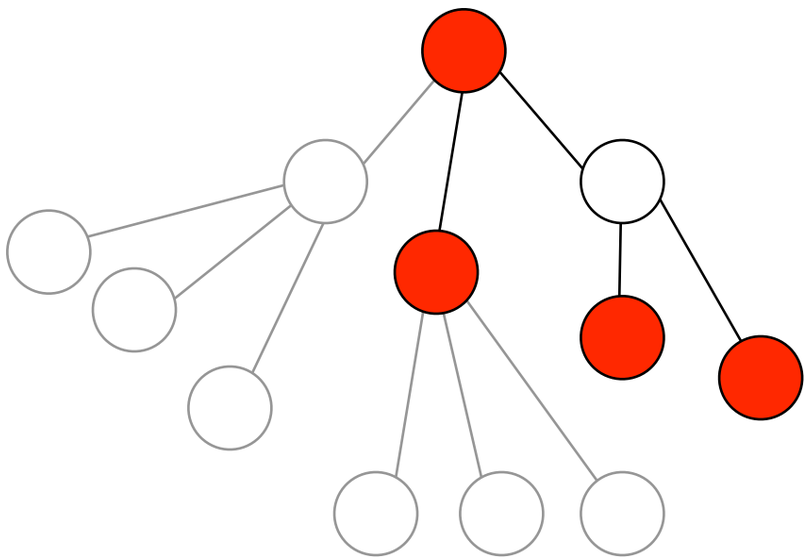
High level overview

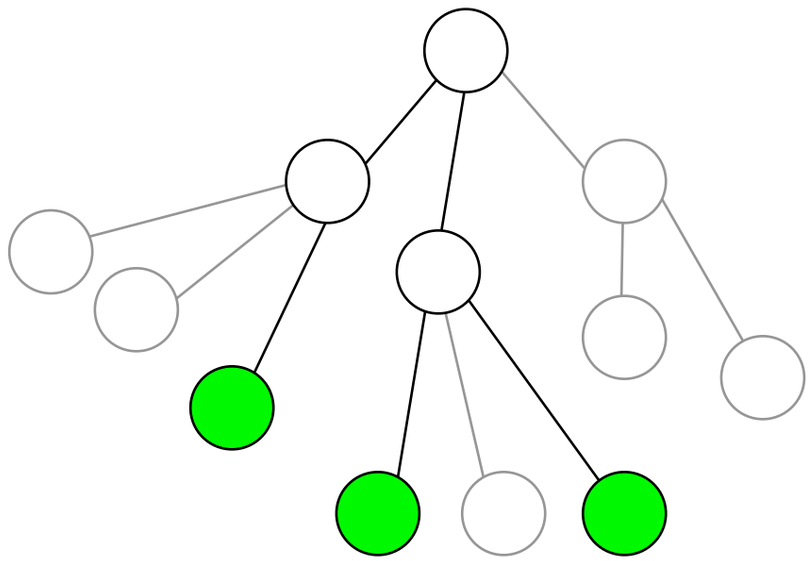
find nodes that satisfy `p@a = q@a`

1. decompose trees into classes
   (class = set of nodes with same value of atribute a)

2. for each class, find nodes that are witnessed by
   that class





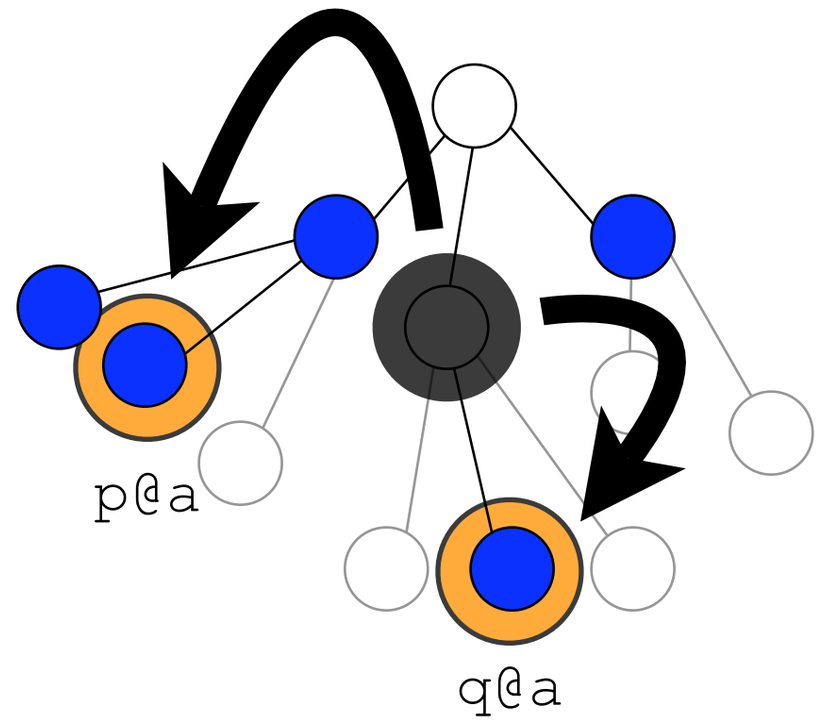


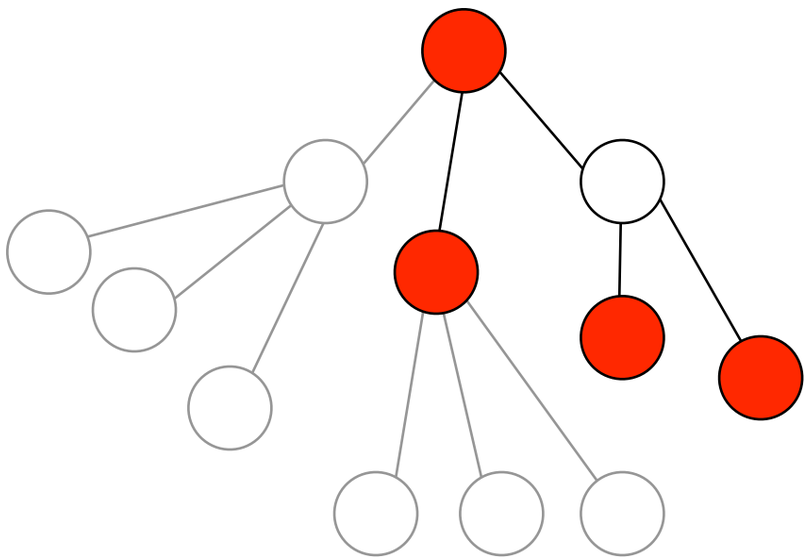
find nodes that satisfy `p@a = q@a`

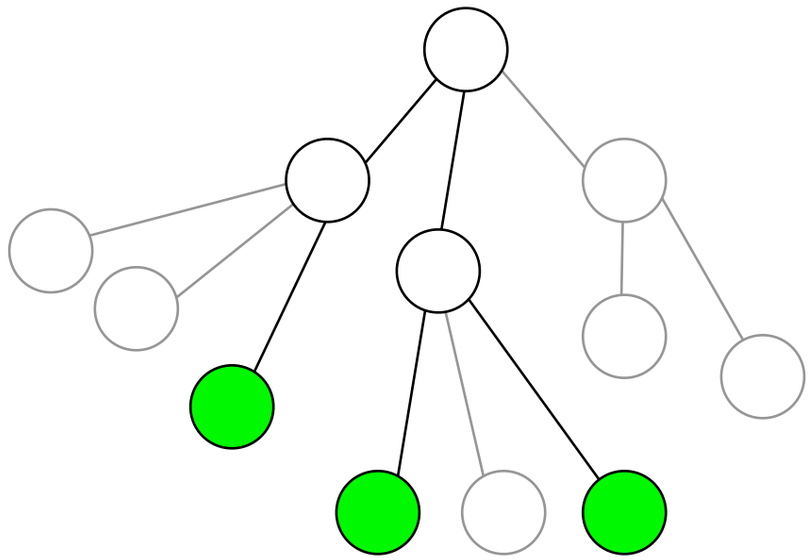
High level overview

1. decompose trees into classes
   (class = set of nodes with same value of atribute a)

2. for each class, find nodes that are witnessed by
   that class





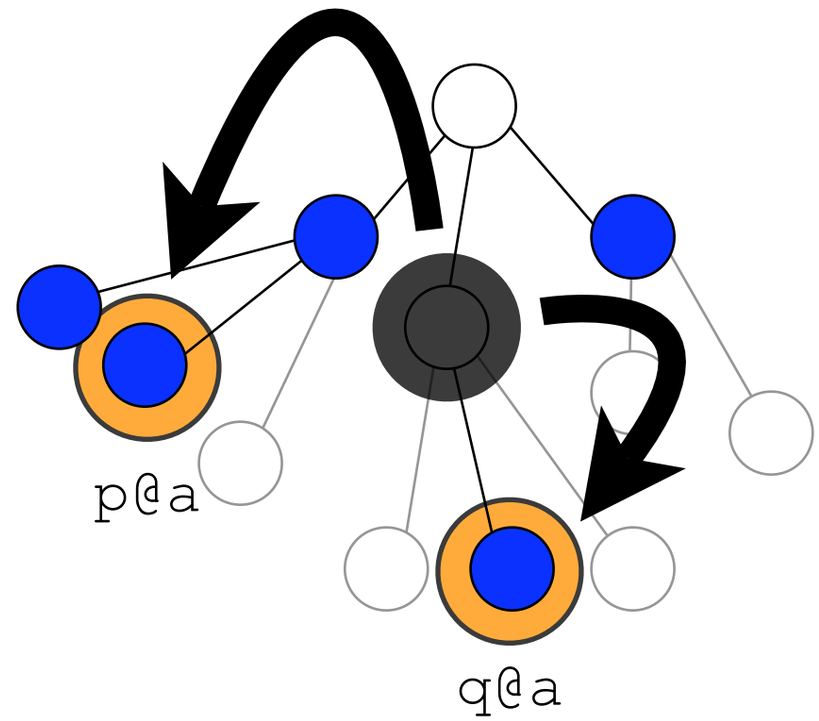


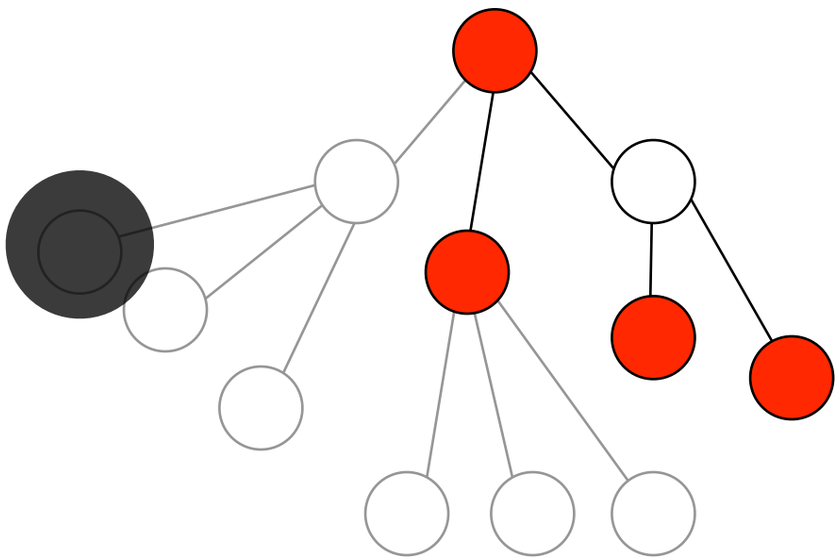
find nodes that satisfy `p@a = q@a`

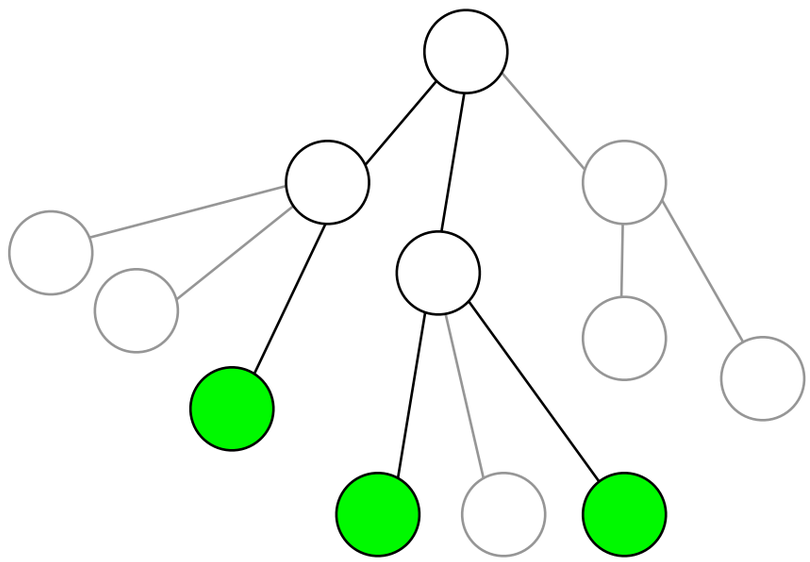
High level overview

1. decompose trees into classes
   (class = set of nodes with same value of atribute a)

2. for each class, find nodes that are witnessed by
   that class





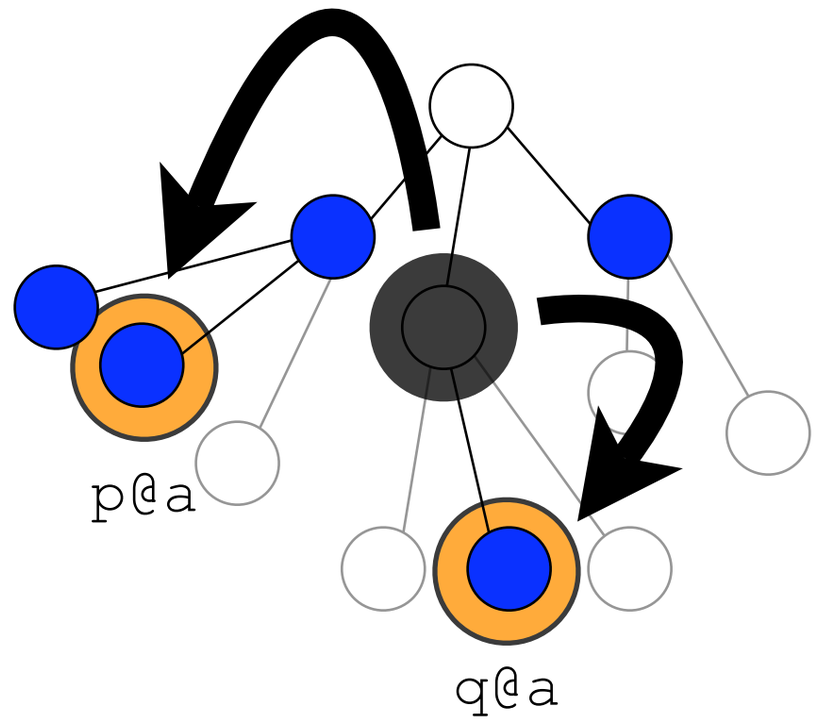


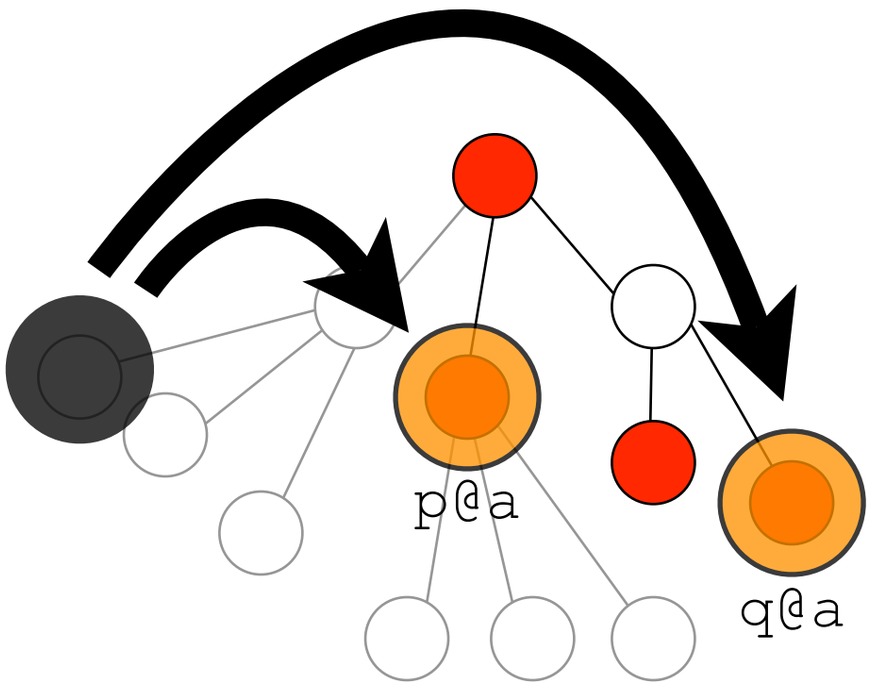
find nodes that satisfy `p@a = q@a`

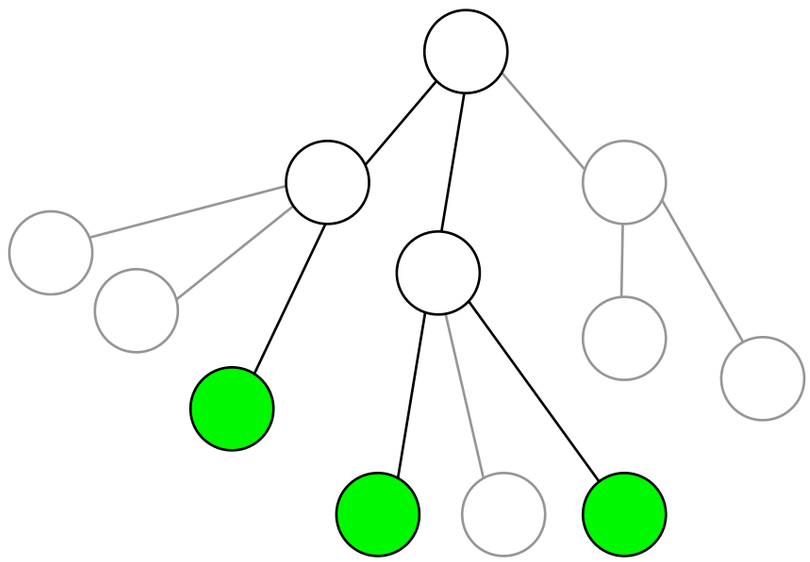
High level overview

1. decompose trees into classes
   (class = set of nodes with same value of atribute a)

2. for each class, find nodes that are witnessed by
   that class





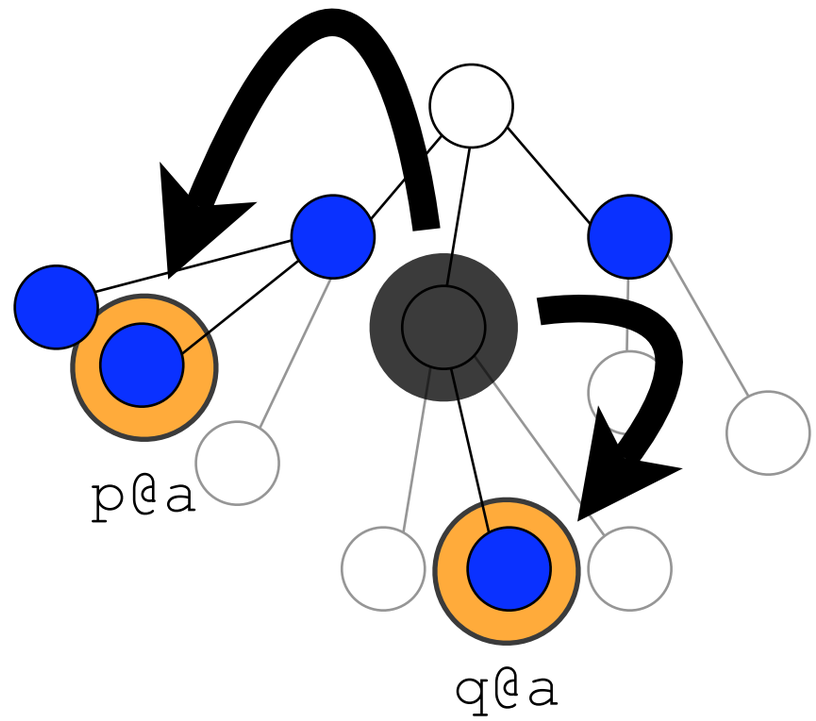


find nodes that satisfy `p@a = q@a`

High level overview

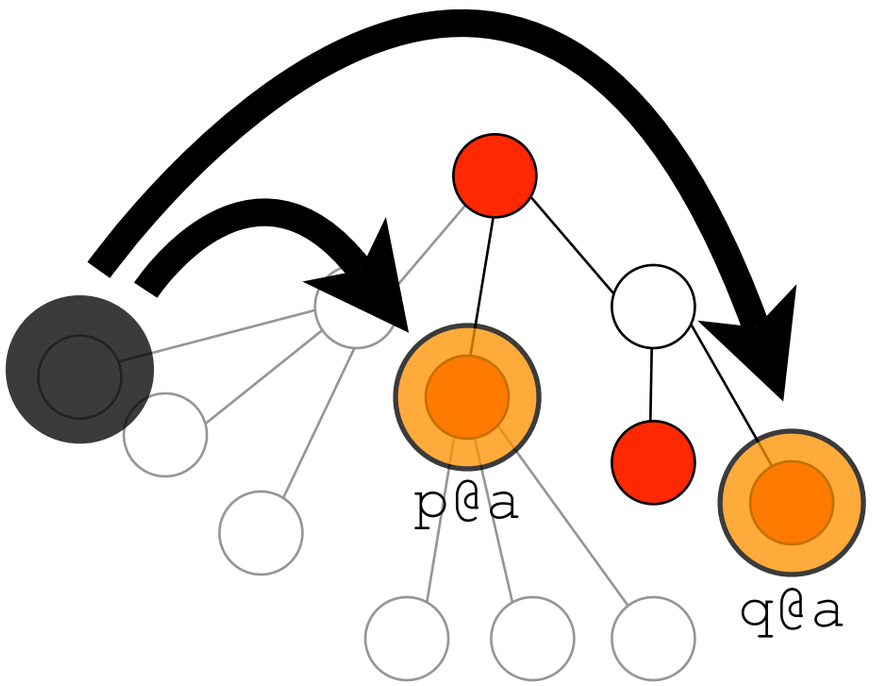
1. decompose trees into classes
   (class = set of nodes with same value of atribute a)

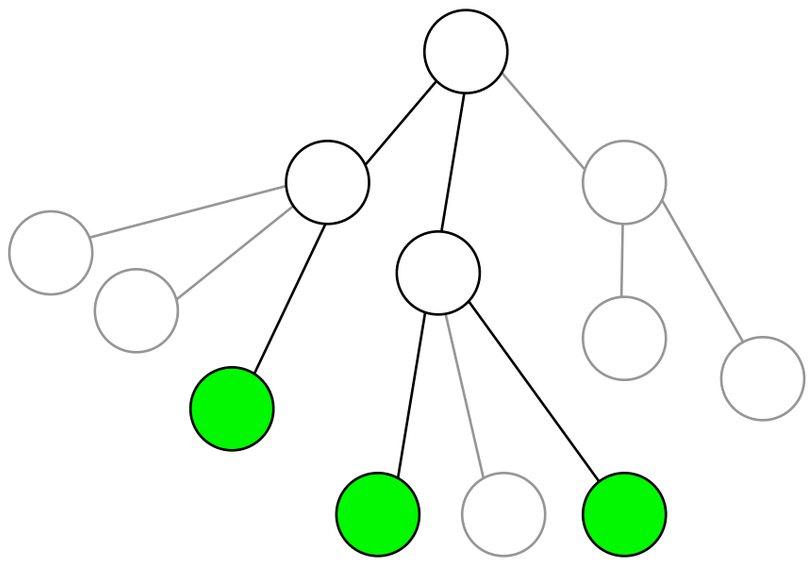
2. for each class, find nodes that are witnessed by
   that class



Goal: avoid repetition

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



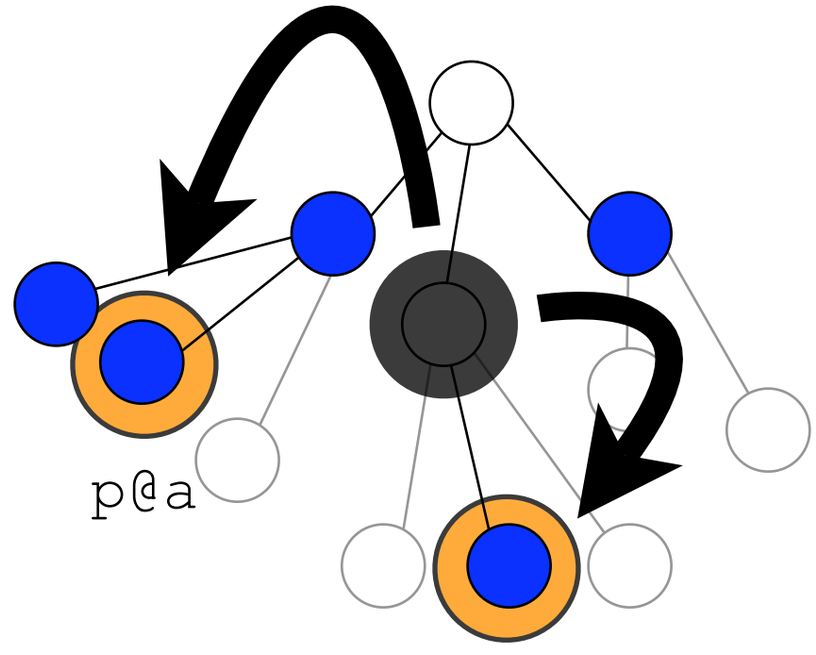


find nodes that satisfy `p@a = q@a`

High level overview

1. decompose trees into classes
   (class = set of nodes with same value of atribute a)

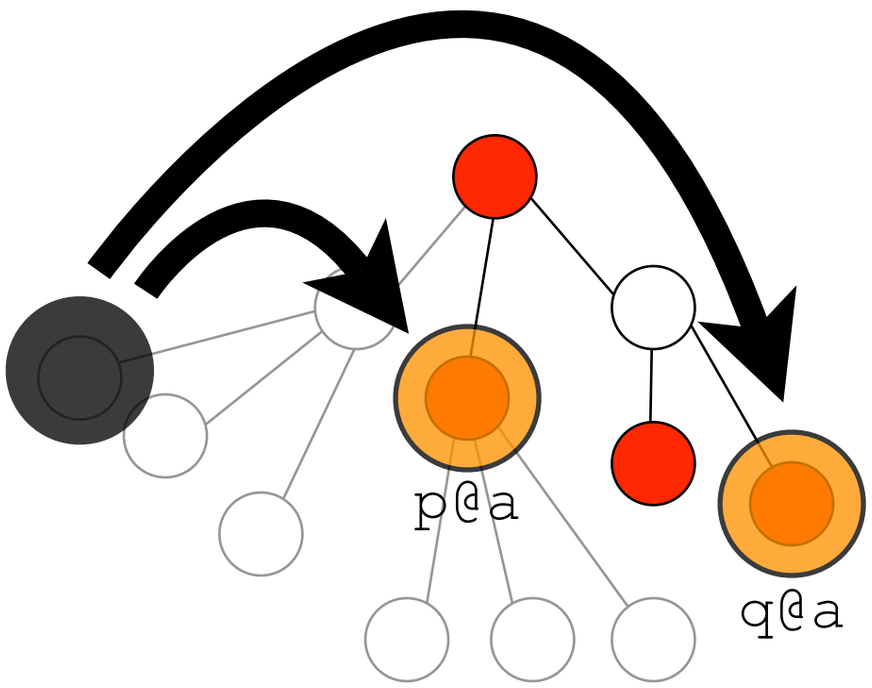
2. for each class, find nodes that are witnessed by
   that class

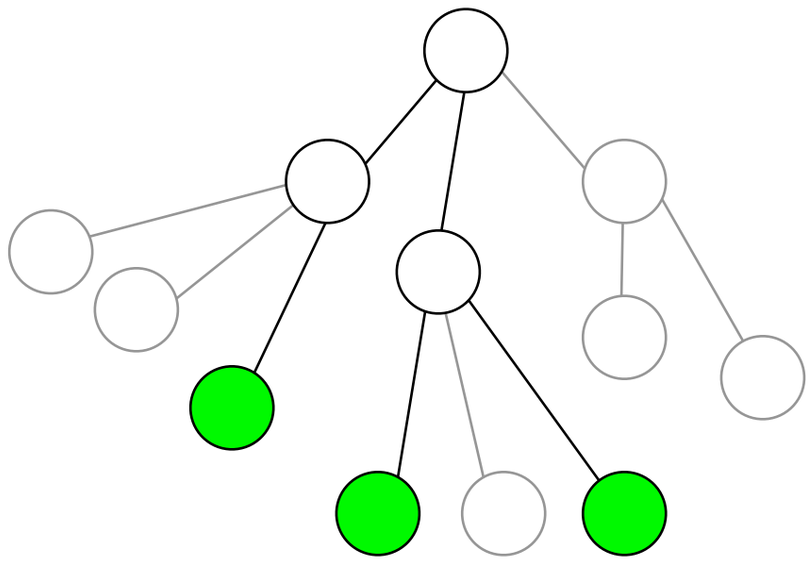


Goal: avoid repetition

– 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 $w=a_1a_2\ldots a_n$
For any infix, membership $a_i\ldots a_j \in L$ can be computed in time $\log n$

L a regular word language.
Do a linear time precomputation on $w=a_1a_2...a_n$
For any infix, membership $a_i...a_j \in L$ can be computed in time *log n*

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

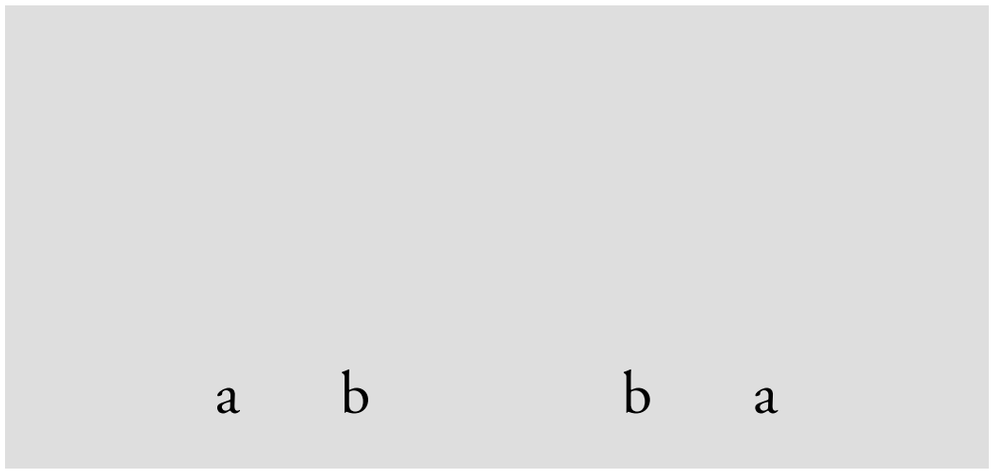
L a regular word language.
Do a linear time precomputation on $w=a_1a_2...a_n$
For any infix, membership $a_i...a_j \in L$ can be computed in time $\log n$

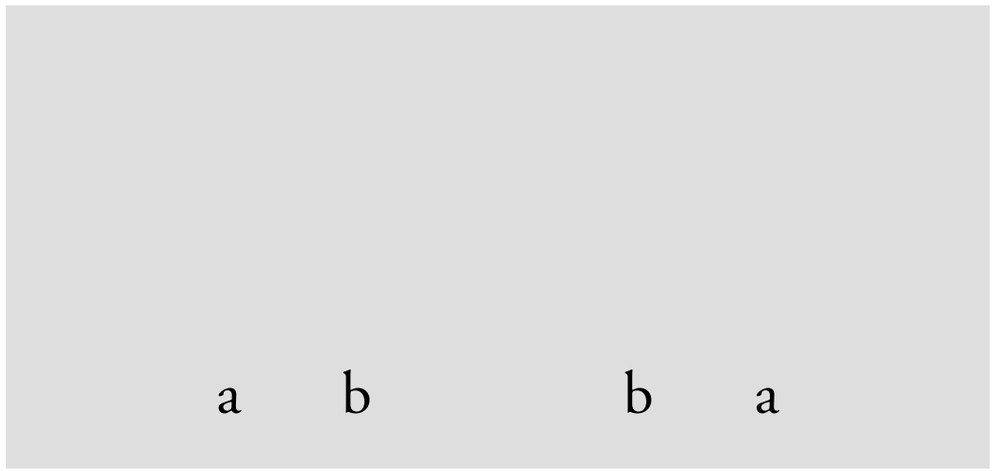
A an automaton recognizing L, and Q its state space.
Each word u induces a tranformation on states $\delta(u) : Q \to Q$.

a b b a a b b a

L a regular word language.
Do a linear time precomputation on $w=a_1a_2...a_n$
For any infix, membership $a_i...a_j \in L$ can be computed in time $\log n$

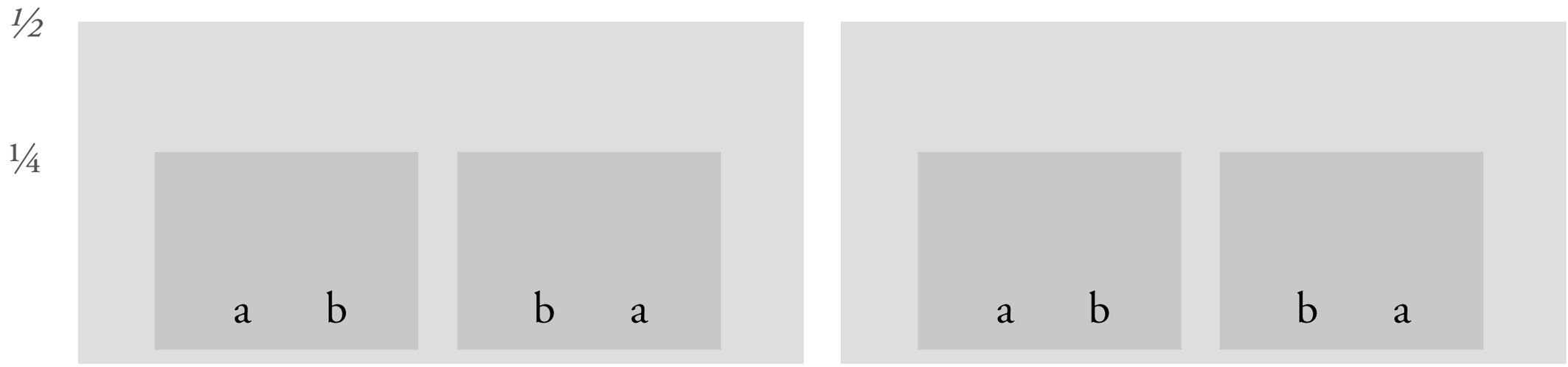
A an automaton recognizing L, and Q its state space.
Each word u induces a tranformation on states $\delta(u) : Q \to Q$.





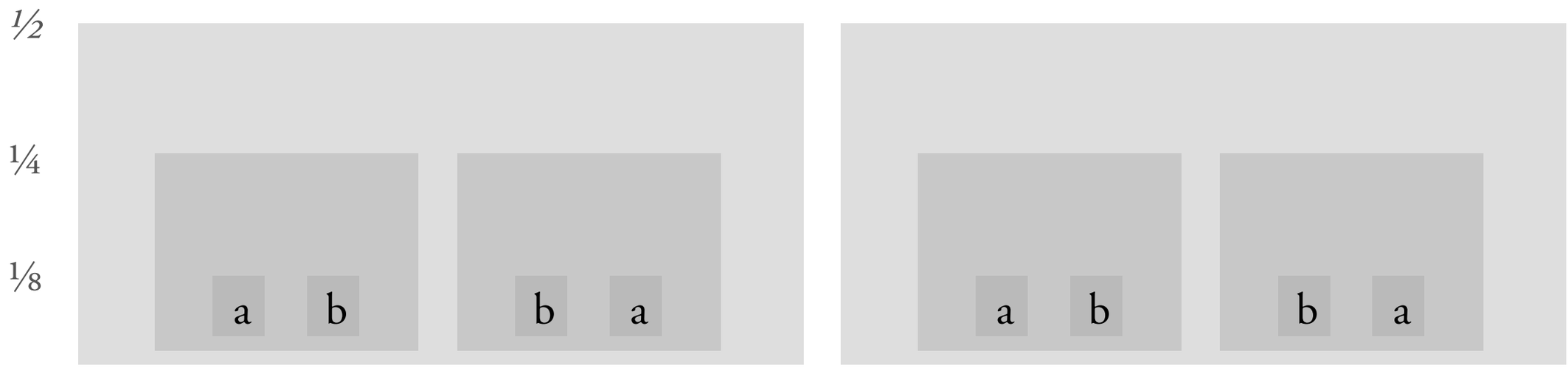
L a regular word language.
Do a linear time precomputation on $w=a_1a_2...a_n$
For any infix, membership $a_i...a_j \in L$ can be computed in time $\log n$

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



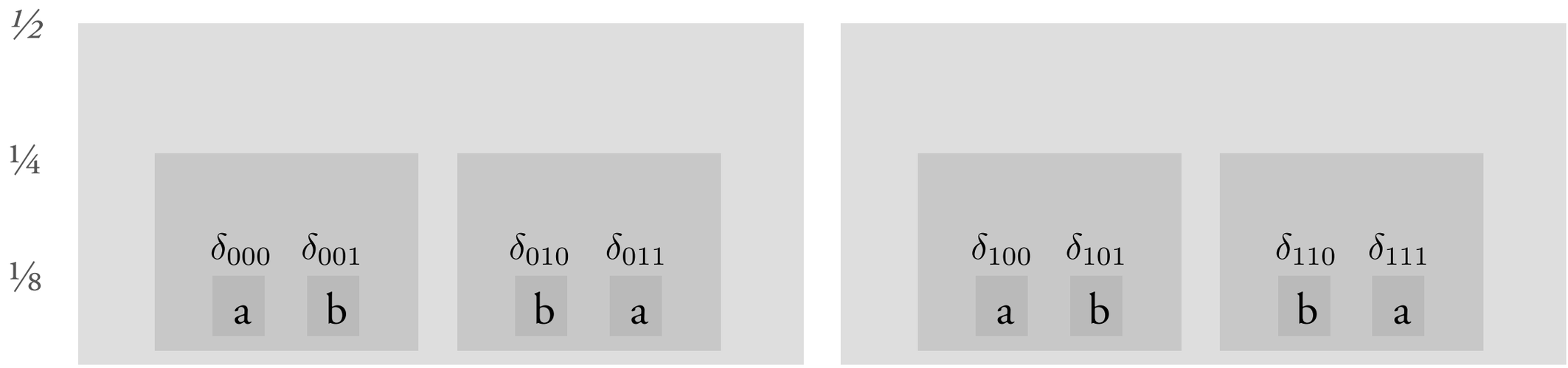
L a regular word language.
Do a linear time precomputation on $w=a_1a_2...a_n$
For any infix, membership $a_i...a_j \in L$ can be computed in time $\log n$

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



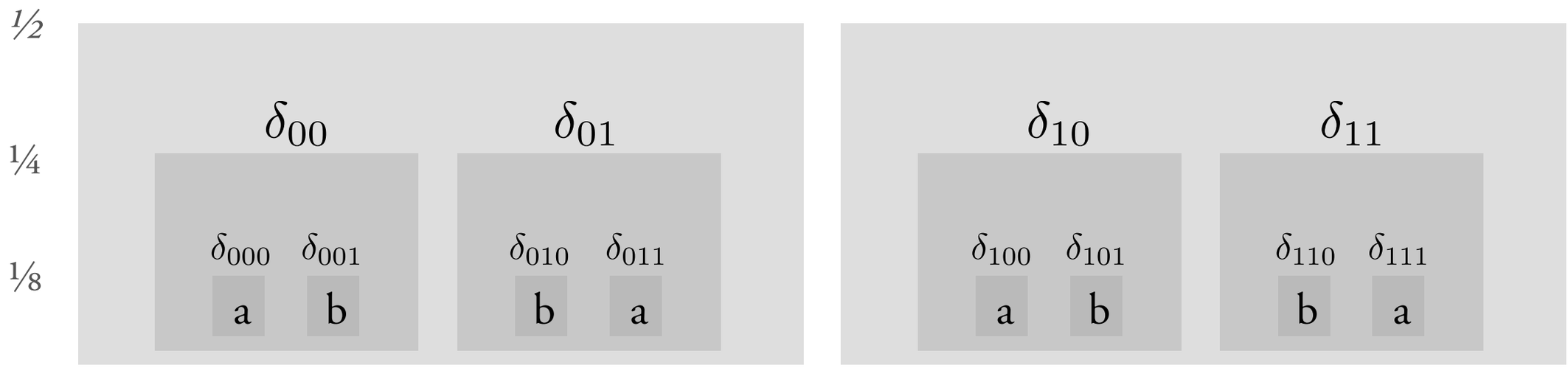
L a regular word language.
Do a linear time precomputation on $w=a_1a_2...a_n$
For any infix, membership $a_i...a_j \in L$ can be computed in time $\log n$

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



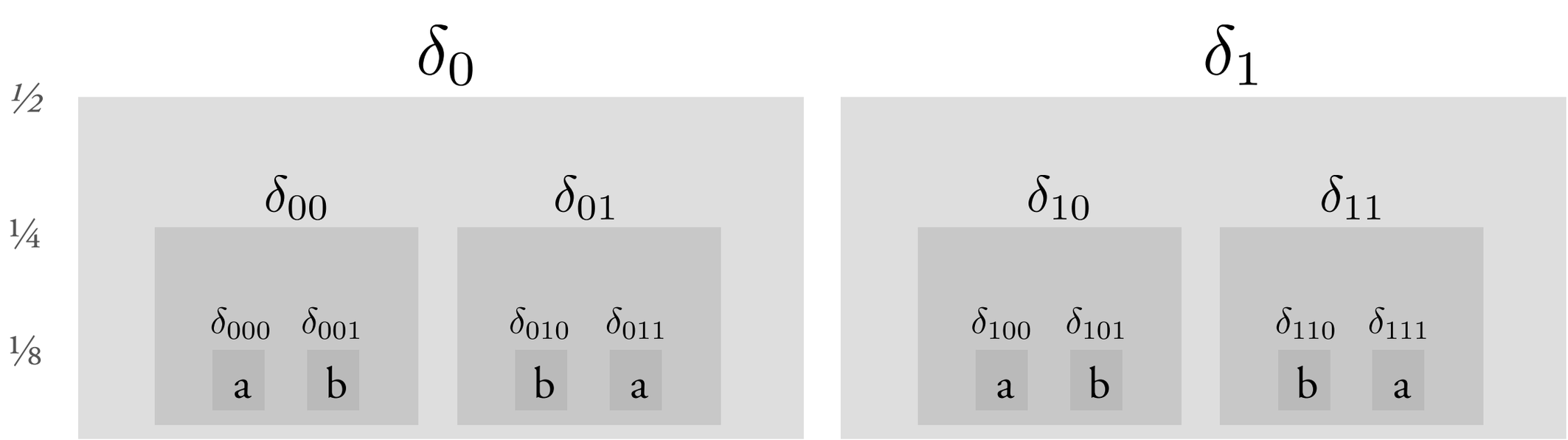
L a regular word language.
Do a linear time precomputation on $w=a_1a_2...a_n$
For any infix, membership $a_i...a_j \in L$ can be computed in time *log n*

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



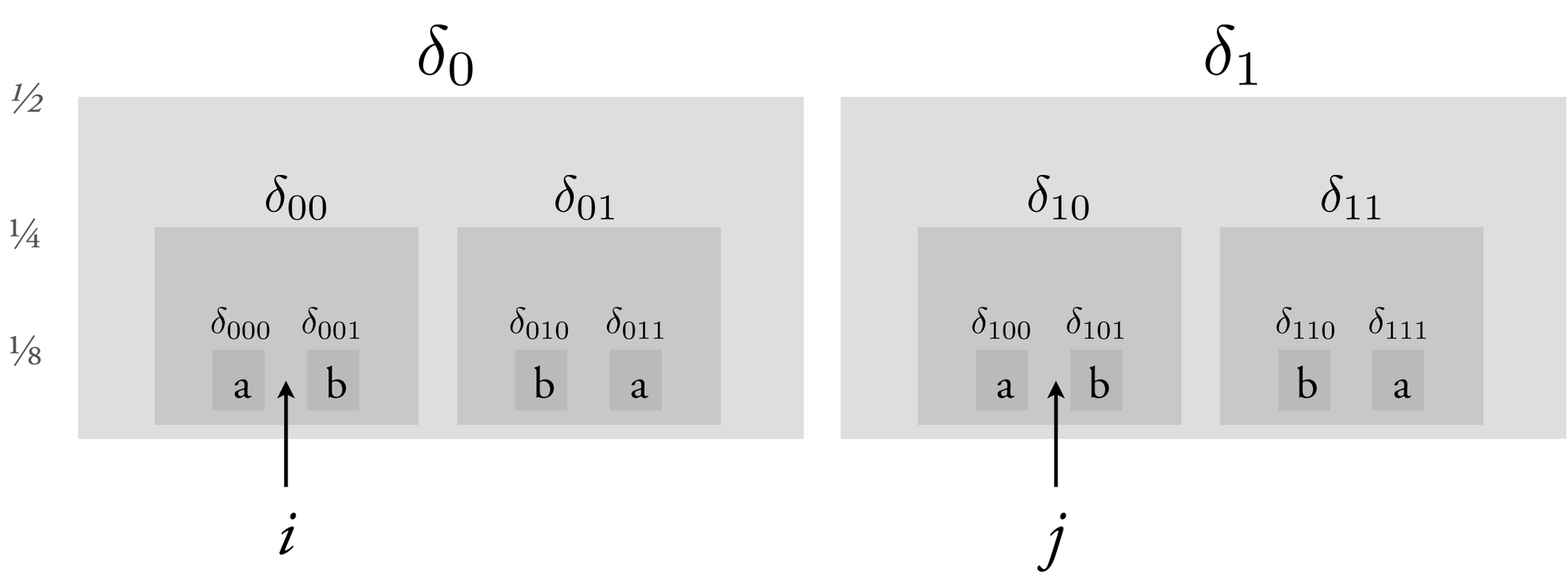
L a regular word language.
Do a linear time precomputation on $w=a_1a_2...a_n$
For any infix, membership $a_i...a_j \in L$ can be computed in time $\log n$

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



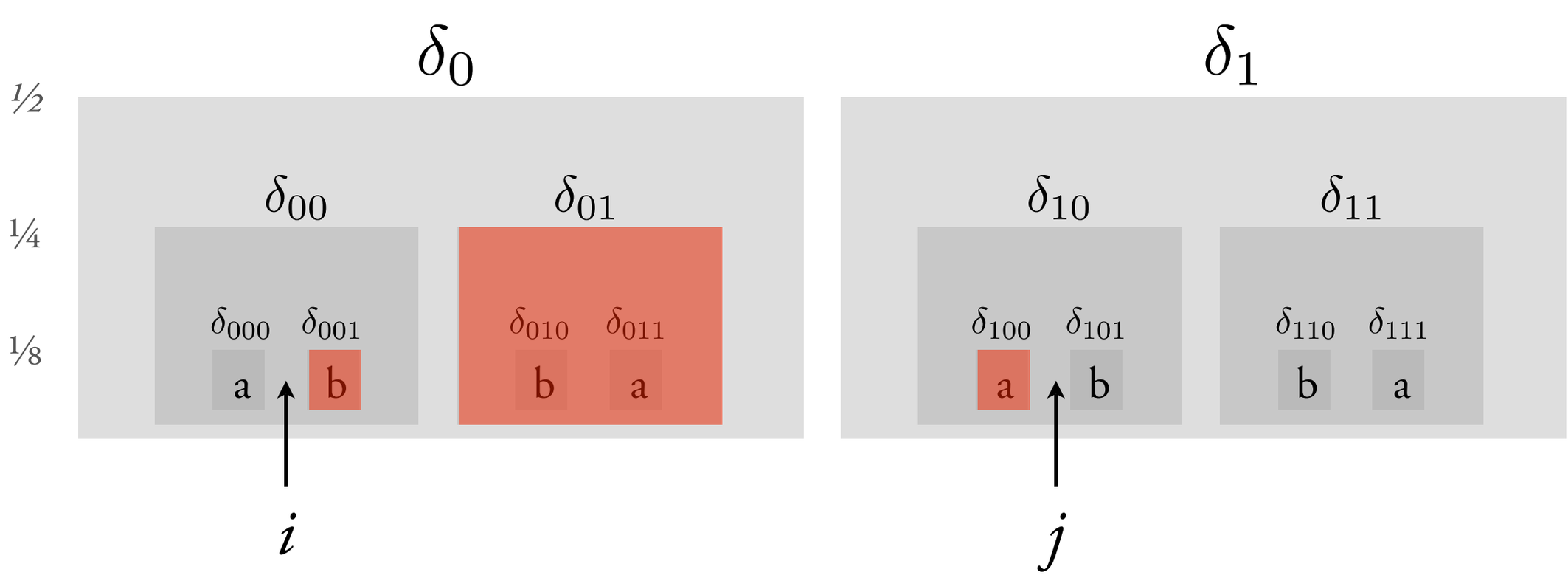
L a regular word language.
Do a linear time precomputation on $w=a_1a_2...a_n$
For any infix, membership $a_i...a_j \in L$ can be computed in time $\log n$

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



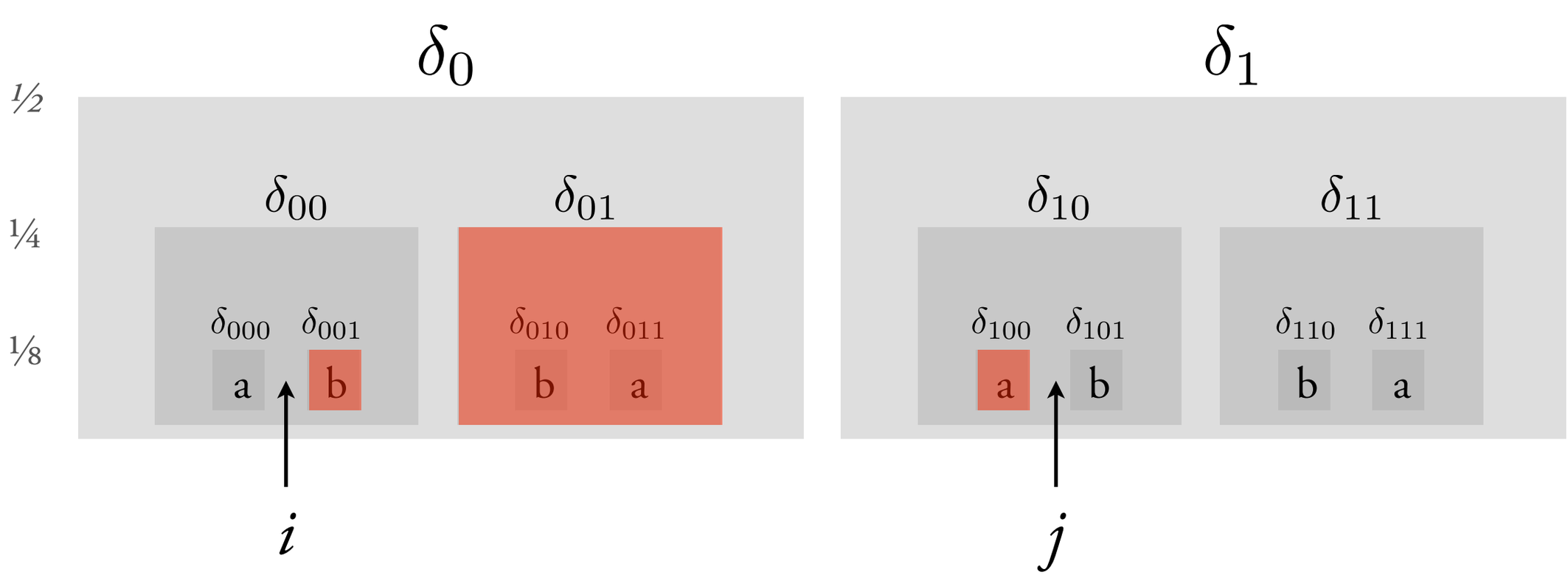
L a regular word language.
Do a linear time precomputation on $w=a_1a_2...a_n$
For any infix, membership $a_i...a_j \in L$ can be computed in time $\log n$

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



L a regular word language.
Do a linear time precomputation on $w=a_1a_2...a_n$
For any infix, membership $a_i...a_j \in L$ can be computed in time *log n*

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



Big news: Simon decomposition does this with constant depth!

Back to XPath evaluation...

To simplify, consider a special case of XPath:

abaabaabbababbababbabbababbababbabbbababbbabbababbabbbabaababbabba

To simplify, consider a special case of XPath:

- words not trees

abaabaabbababbababbabbababbababbabbbababbbabbababbabbbabaababbabba

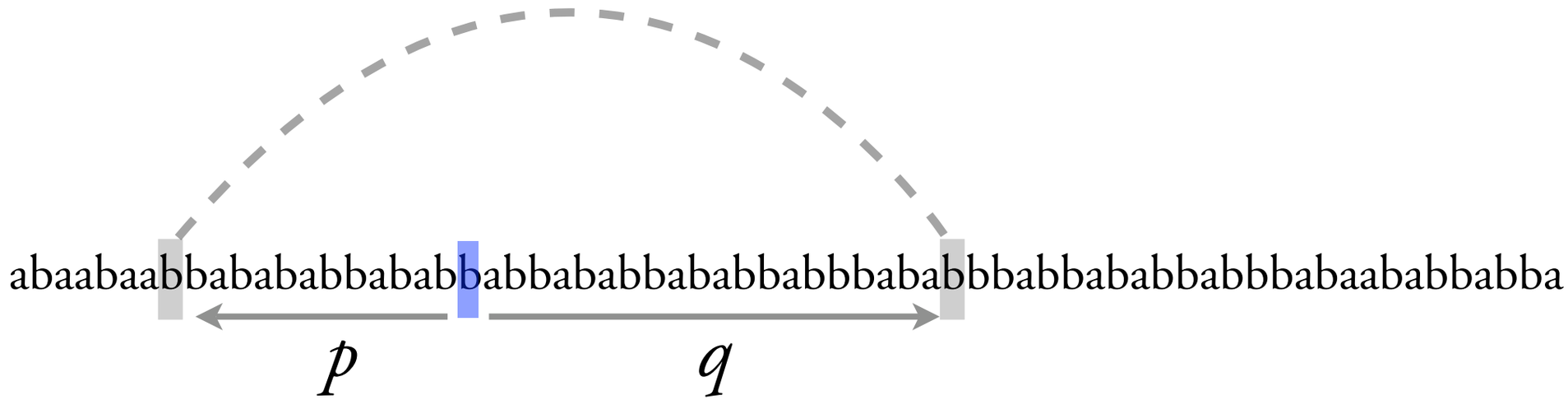
To simplify, consider a special case of XPath:

- words not trees
- a test p@a =q@a

abaabaabbababbababbababbababbababbababbababbababbababbabaababbabba

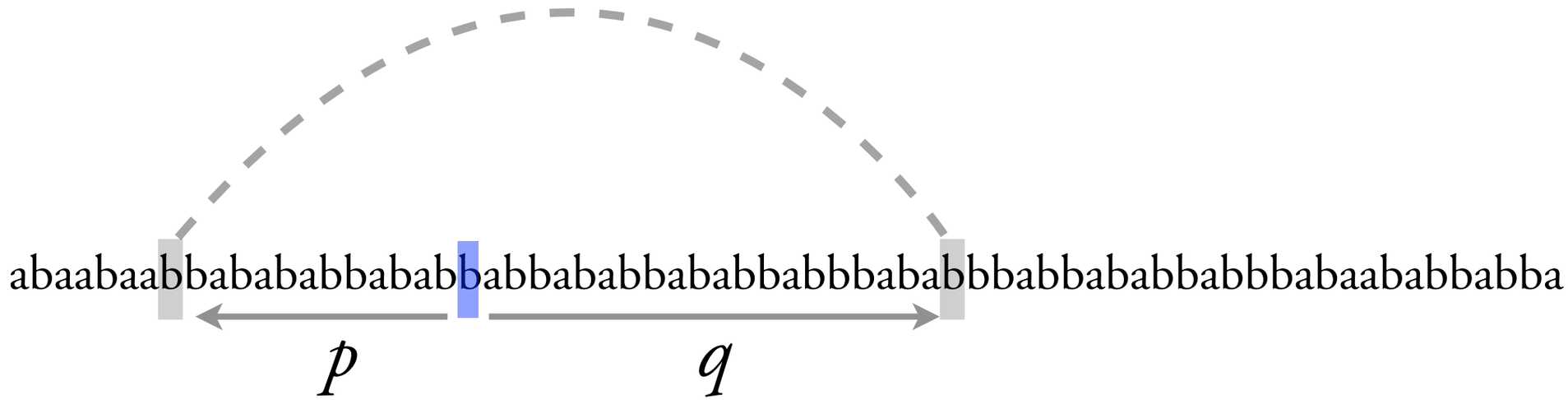
To simplify, consider a special case of XPath:

- words not trees
- a test p@a =q@a



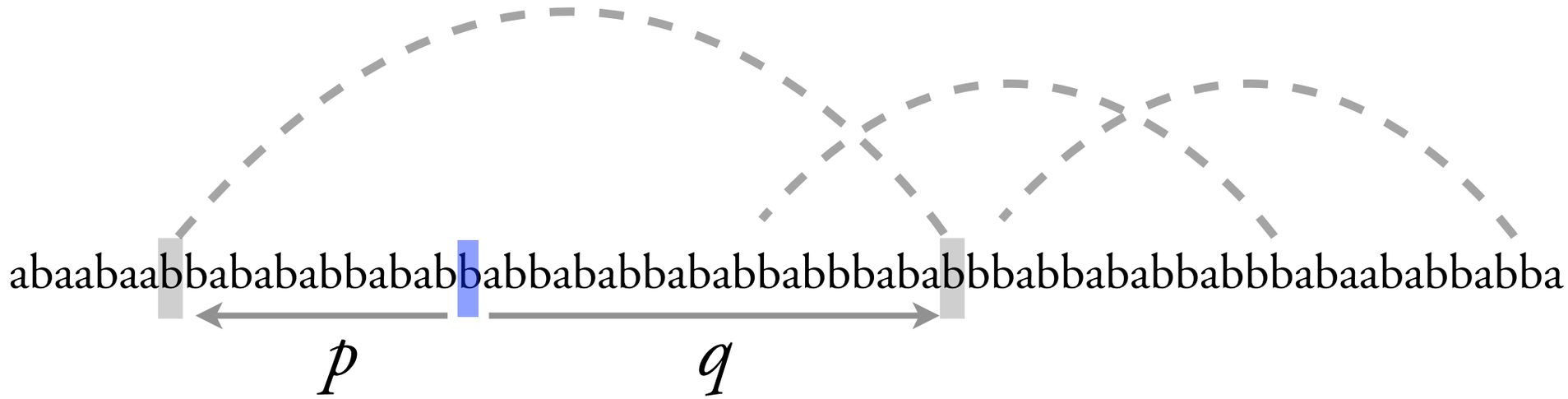
To simplify, consider a special case of XPath:

- words not trees
- a test p@a =q@a



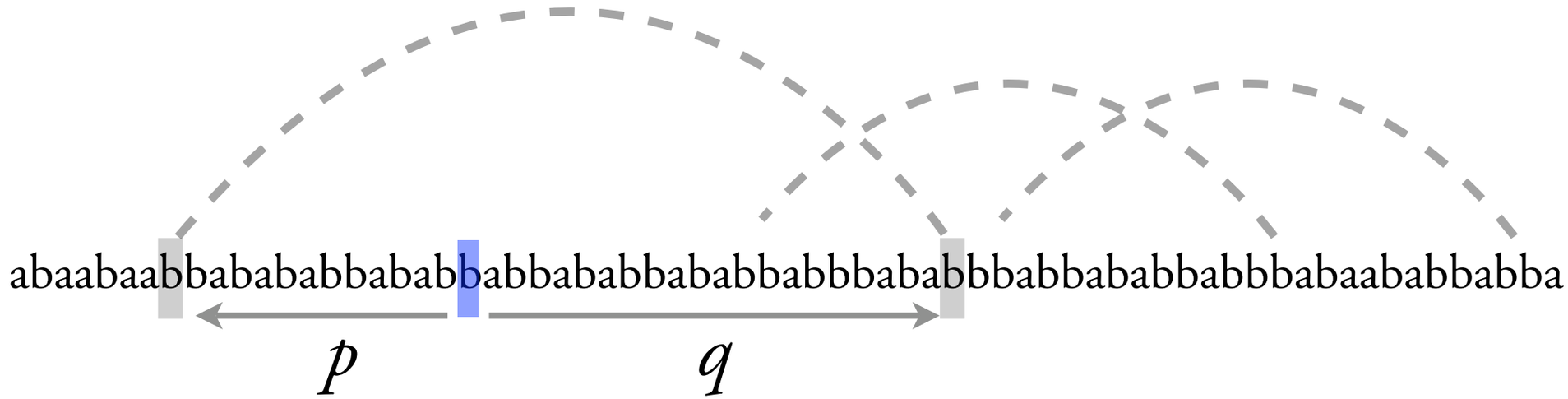
To simplify, consider a special case of XPath:

- words not trees
- a test p@a =q@a
- each attribute value appears exactly twice



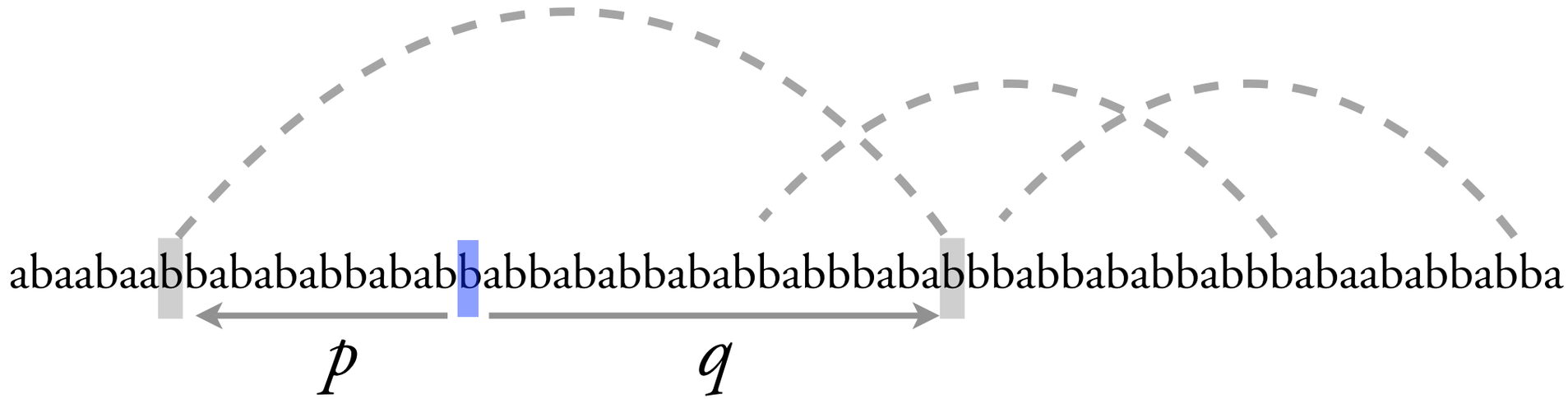
To simplify, consider a special case of XPath:

- words not trees
- a test p@a =q@a
- each attribute value appears exactly twice



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



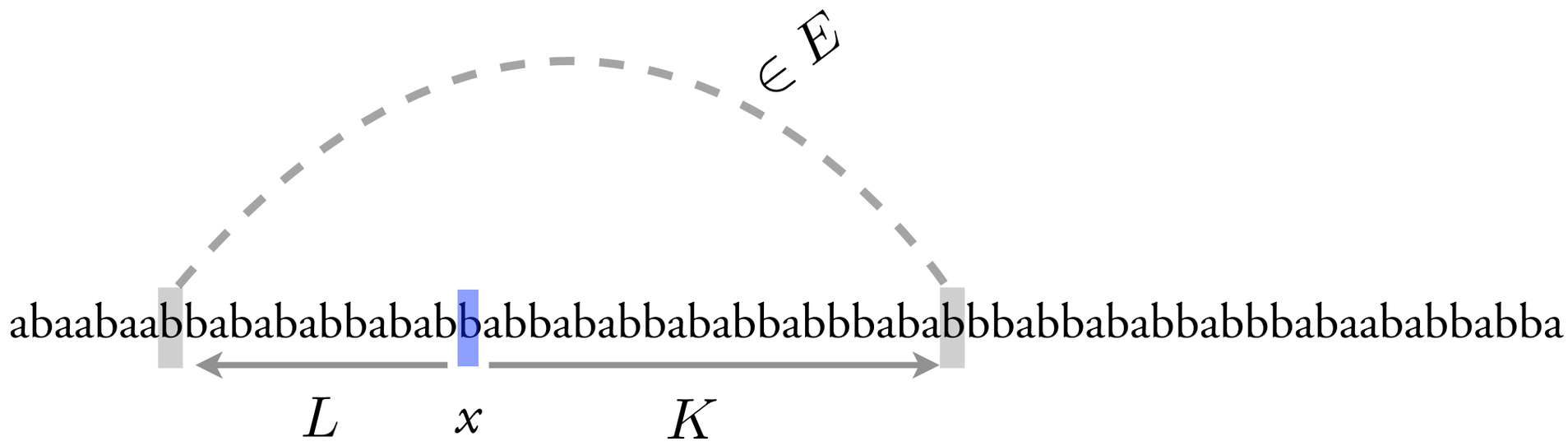
To simplify, consider a special case of XPath:

- 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
- p only goes left, q only goes right

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

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

Output. Set of nodes x with



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
- p only goes left, q only goes right

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

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

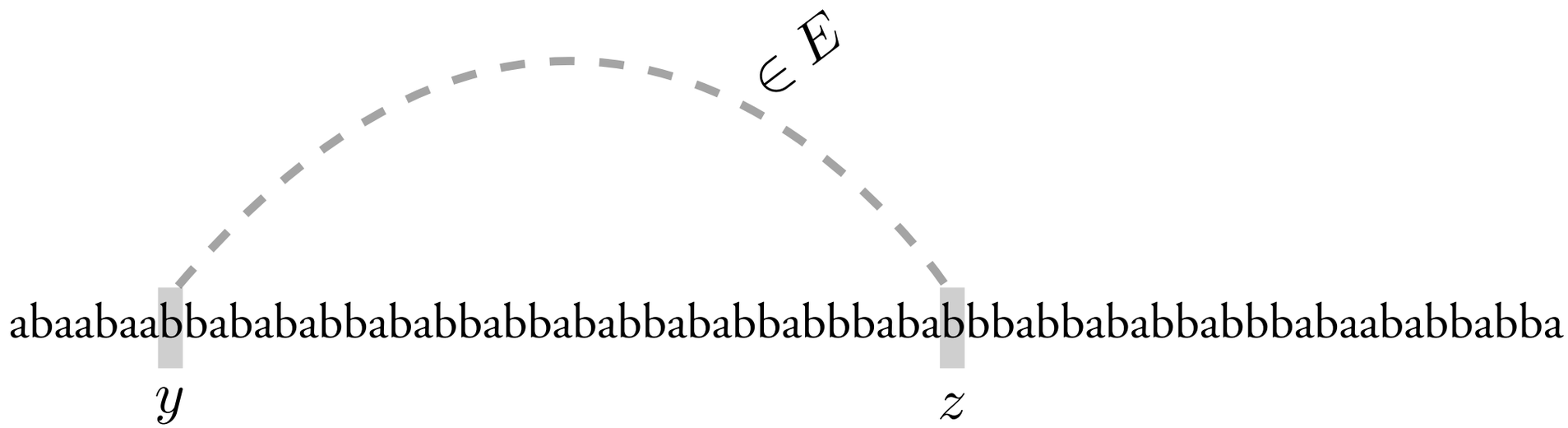
Output. Set of nodes x with

abaabaabbababababbababbabbababbababbabbbababbbabbababbabbbabaababbabba

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

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

Output. Set of nodes x with



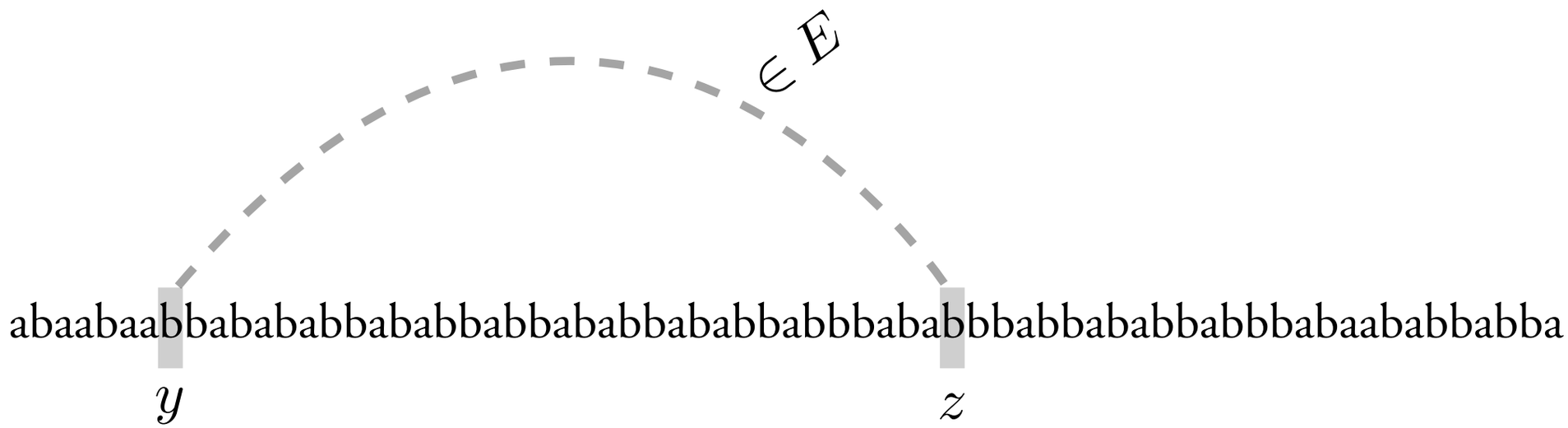
Naive algorithm.

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

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

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

Output. Set of nodes x with



Naive algorithm.

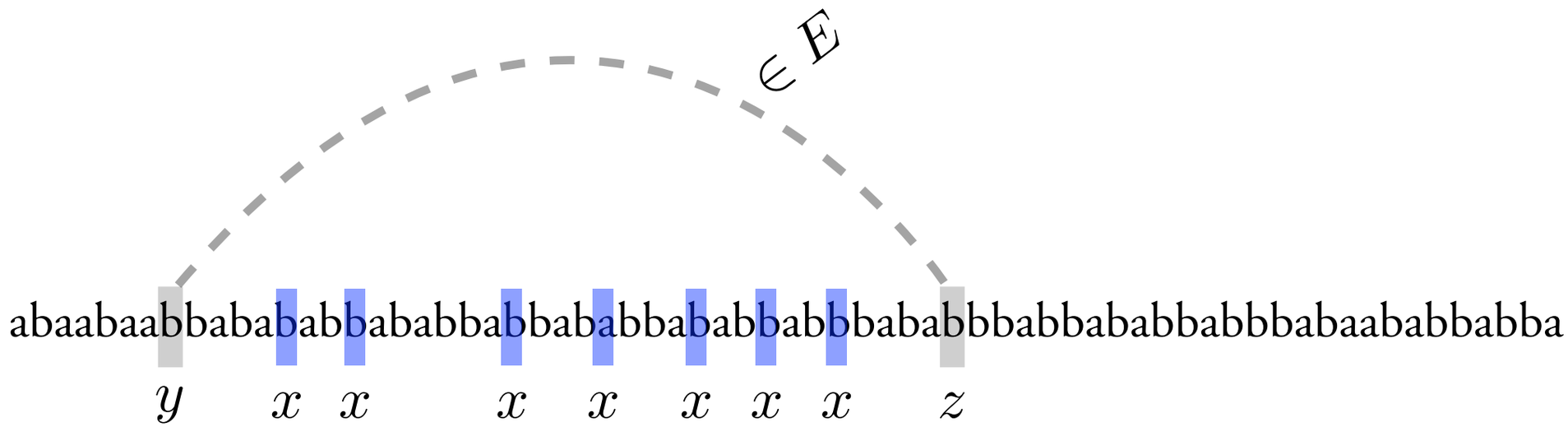
For every match $(y, z) \in E$

Find nodes x such that $w[y..x] \in L$ $w[x..z] \in K$

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

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

Output. Set of nodes x with



Naive algorithm.

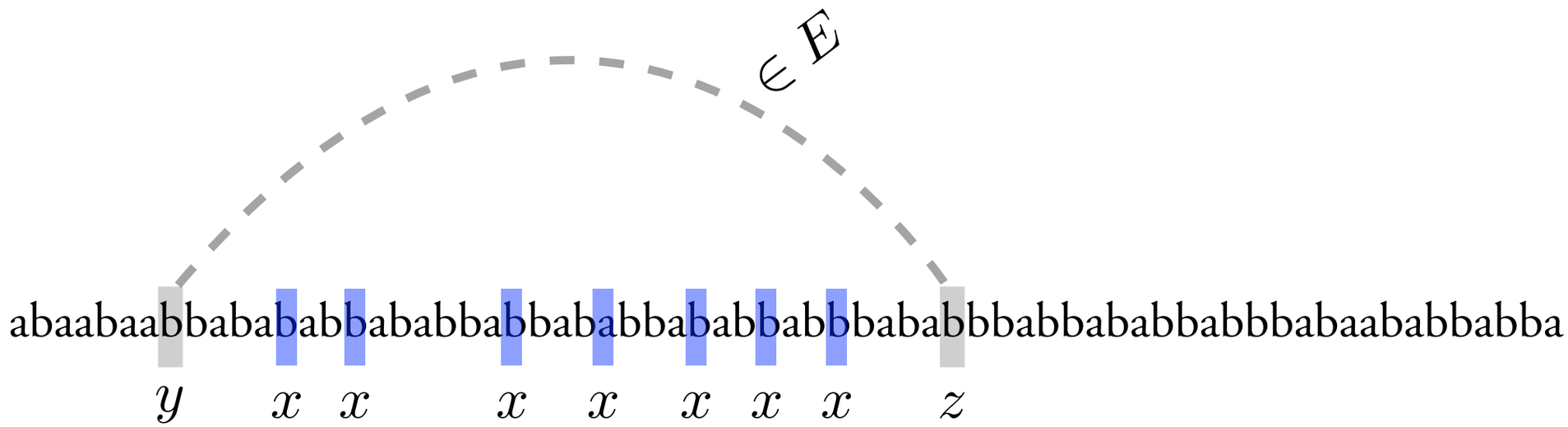
For every match $(y, z) \in E$

Find nodes x such that $w[y..x] \in L$ $w[x..z] \in K$

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

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

Output. Set of nodes x with



Naive algorithm.

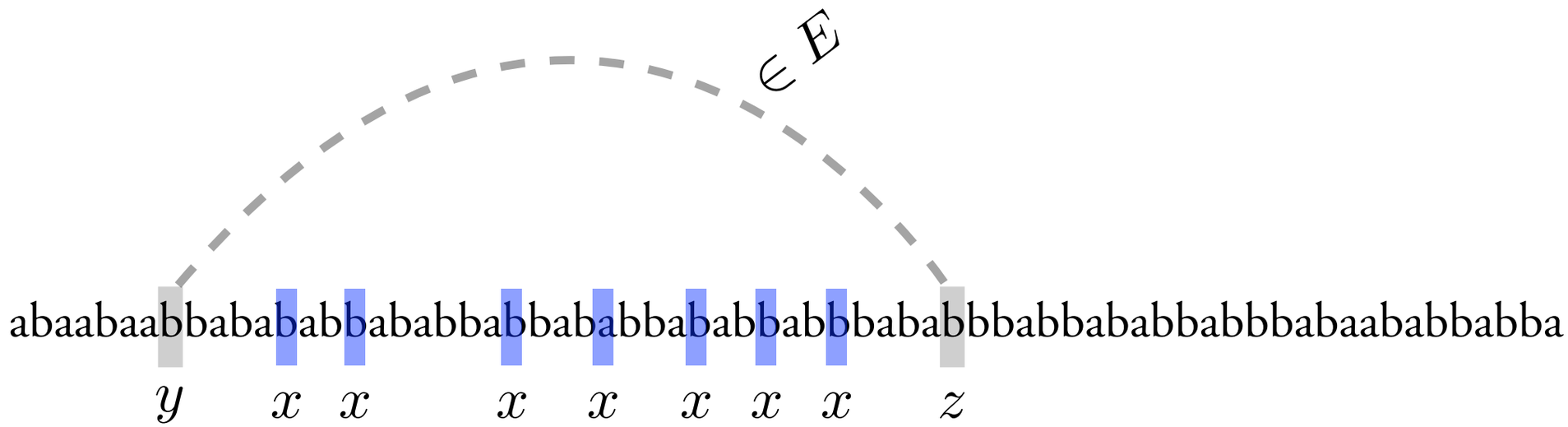
For every match $(y, z) \in E$

Find nodes x such that $w[y..x] \in L$ $w[x..z] \in K$ time $O(n^2)$

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

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

Output. Set of nodes *x* with



Naive algorithm.

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

Find nodes *x* such that $w[y..x] \in L$ $w[x..z] \in K$ time $O(n^2)$

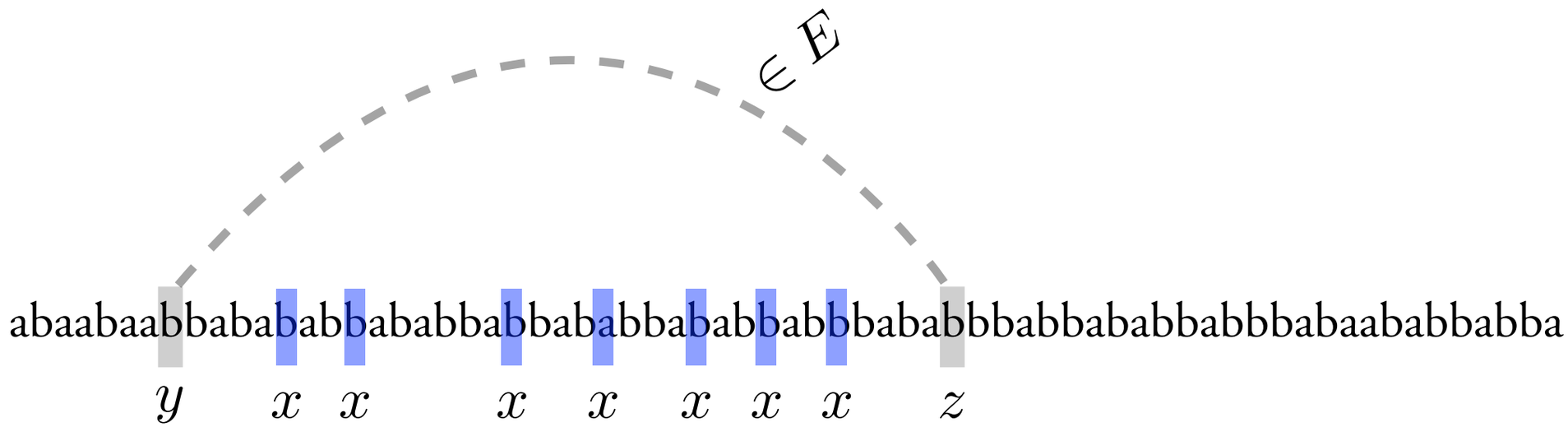
Divide and conquer dynamic algorithm.

Solves the problem in time $O(n \log(n))$

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

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

Output. Set of nodes x with



Naive algorithm.

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

Find nodes x such that $w[y..x] \in L$ $w[x..z] \in K$ time $O(n^2)$

Divide and conquer dynamic algorithm.

Solves the problem in time $O(n \log(n))$

An algorithm that uses the Simon decomposition

Solves the problem in time $O(n)$

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

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

Output. Set of nodes x with

abaabaabbababababbababbabbababbababbabbbababbbabbababbabbbabaababbabba

Divide and conquer dynamic algorithm.

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

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

Output. Set of nodes x with

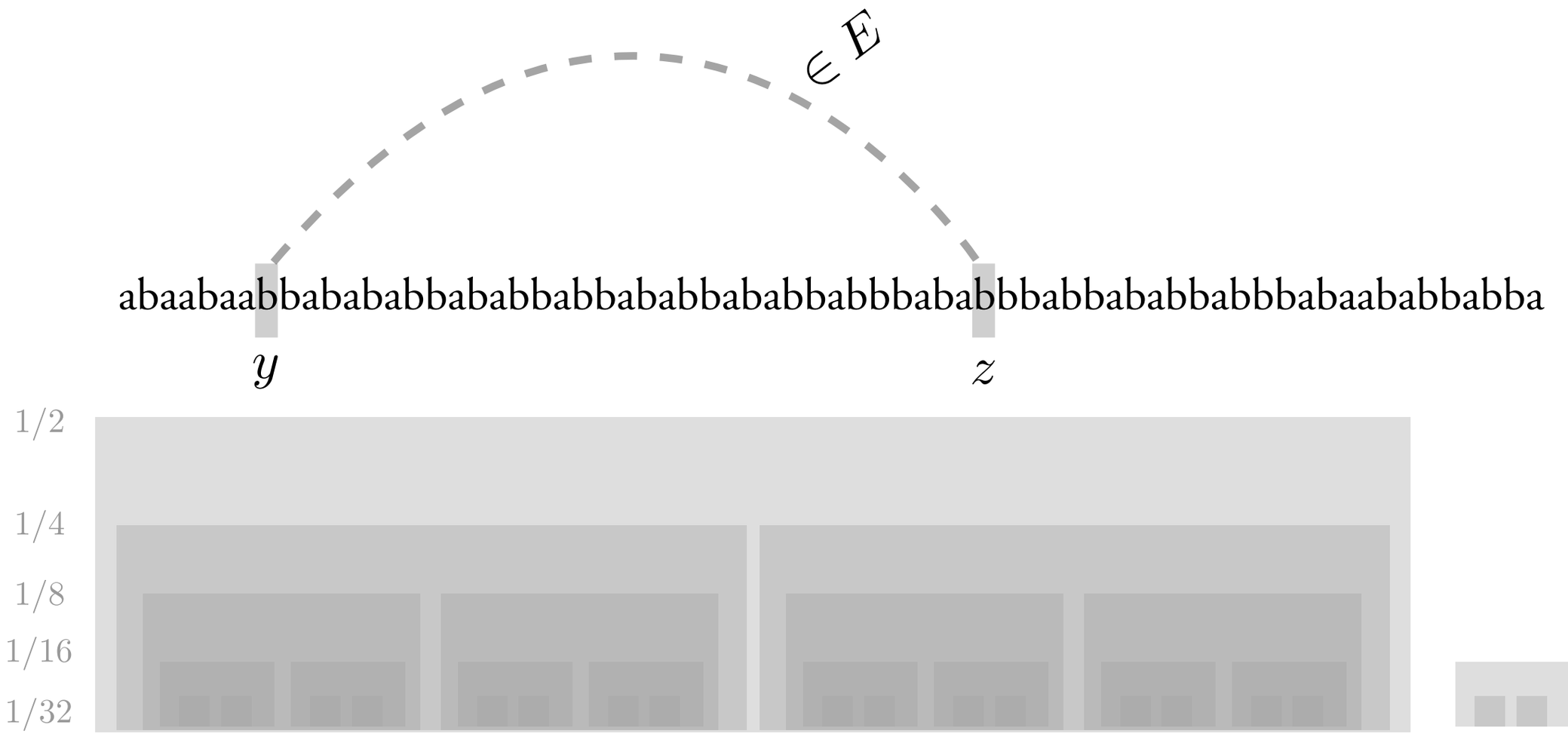
abaabaabbababababbababbabbababbababbabbbababbbabbababbabbbabaababbabba

Divide and conquer dynamic algorithm.

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

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

Output. Set of nodes x with



Divide and conquer dynamic algorithm.

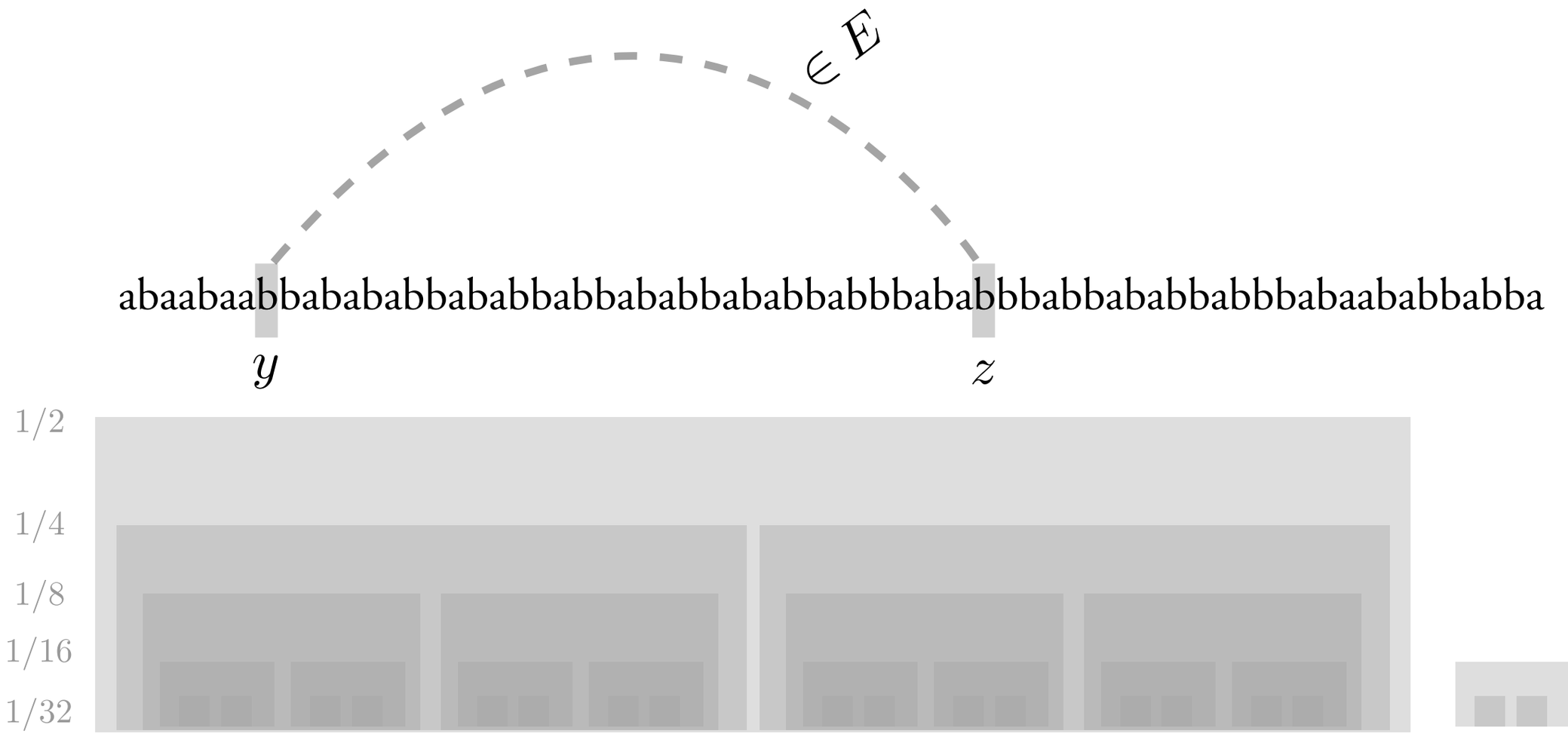
For every match $(y, z) \in E$

Find nodes x such that $w[y..x] \in L$ $w[x..z] \in K$

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

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

Output. Set of nodes x with



Divide and conquer dynamic algorithm.

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

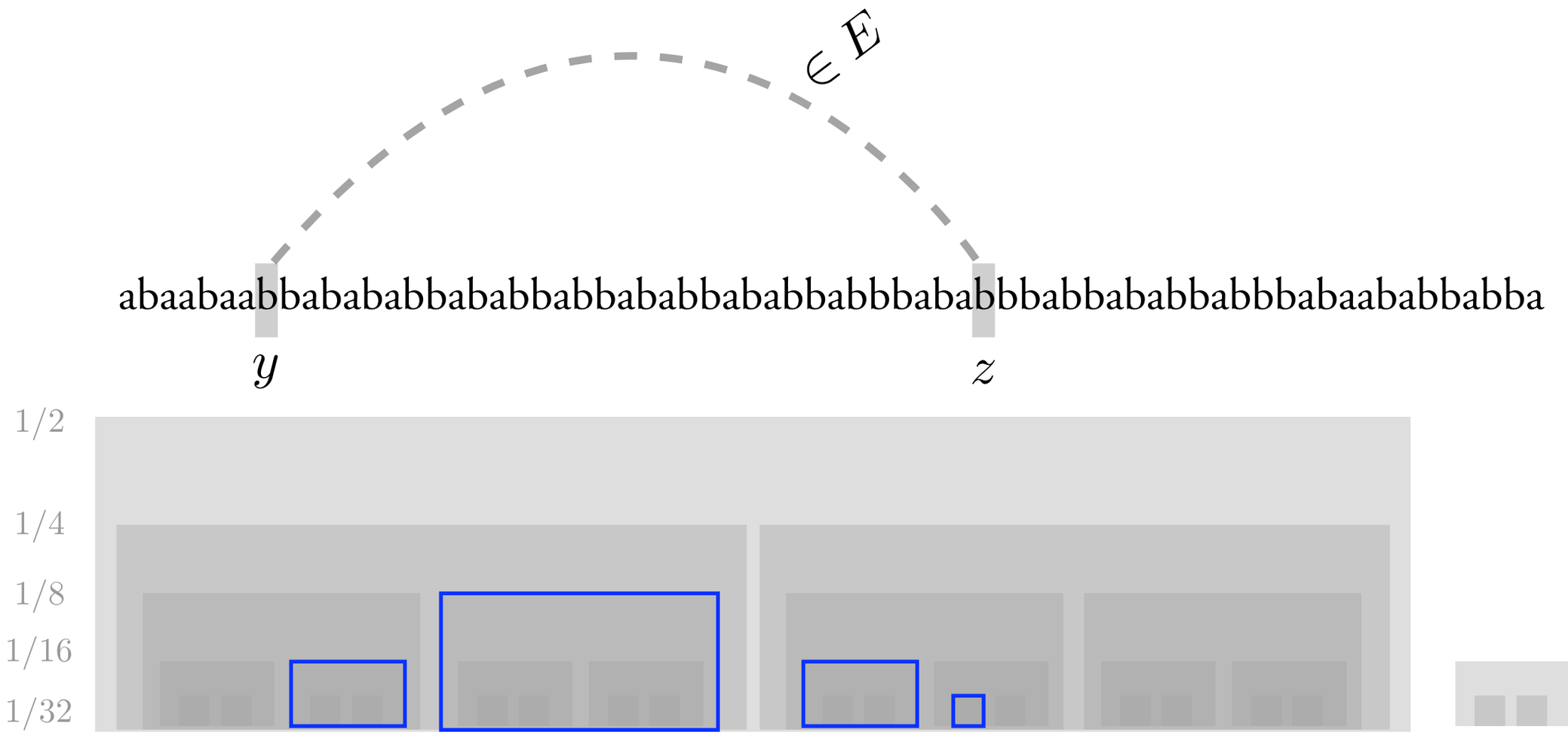
Find nodes x such that $w[y..x] \in L$ $w[x..z] \in K$

but only do logarithmically many operations each time

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

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

Output. Set of nodes x with



Divide and conquer dynamic algorithm.

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

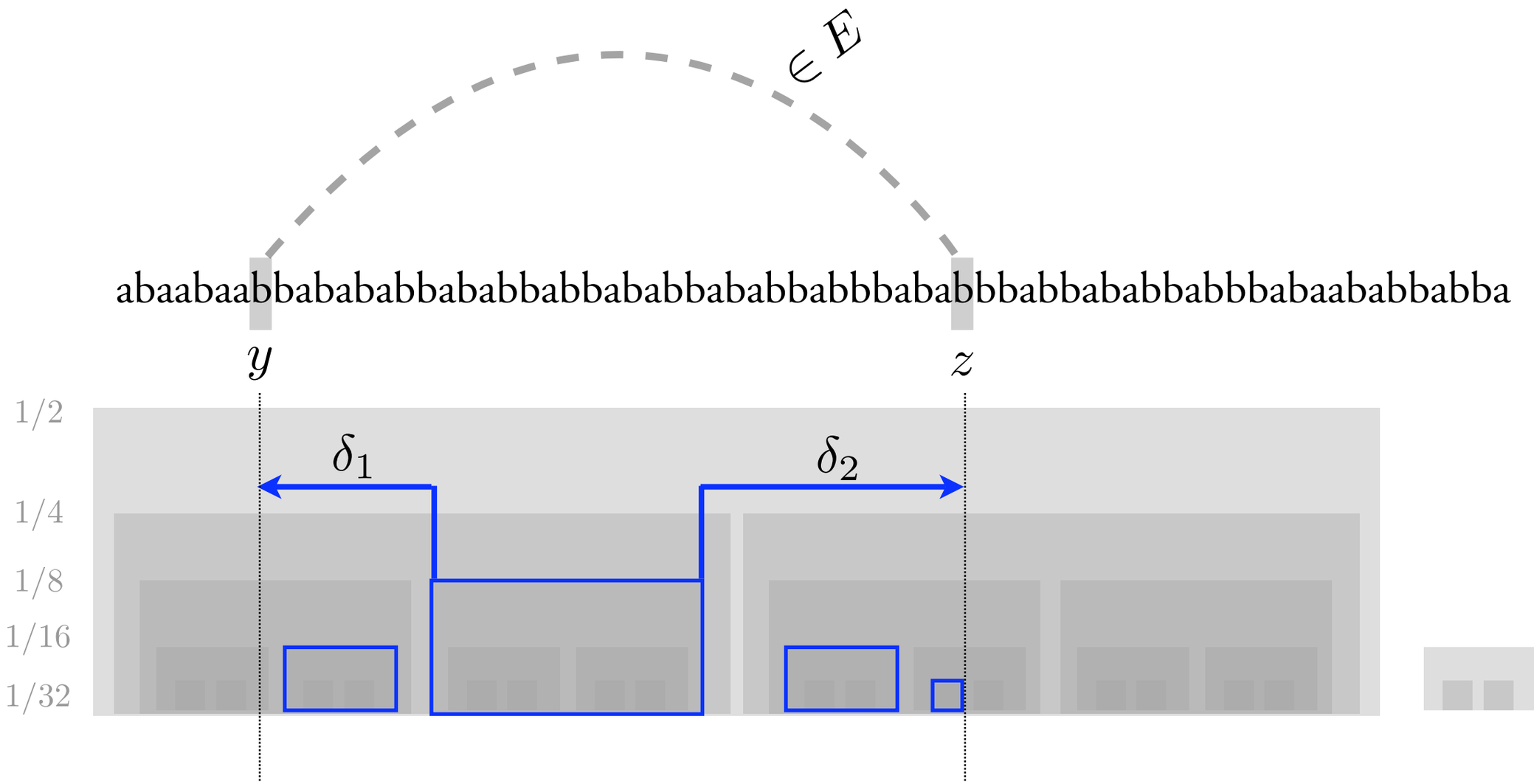
Find nodes x such that $w[y..x] \in L$ $w[x..z] \in K$

but only do logarithmically many operations each time

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

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

Output. Set of nodes x with



Divide and conquer dynamic algorithm.

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

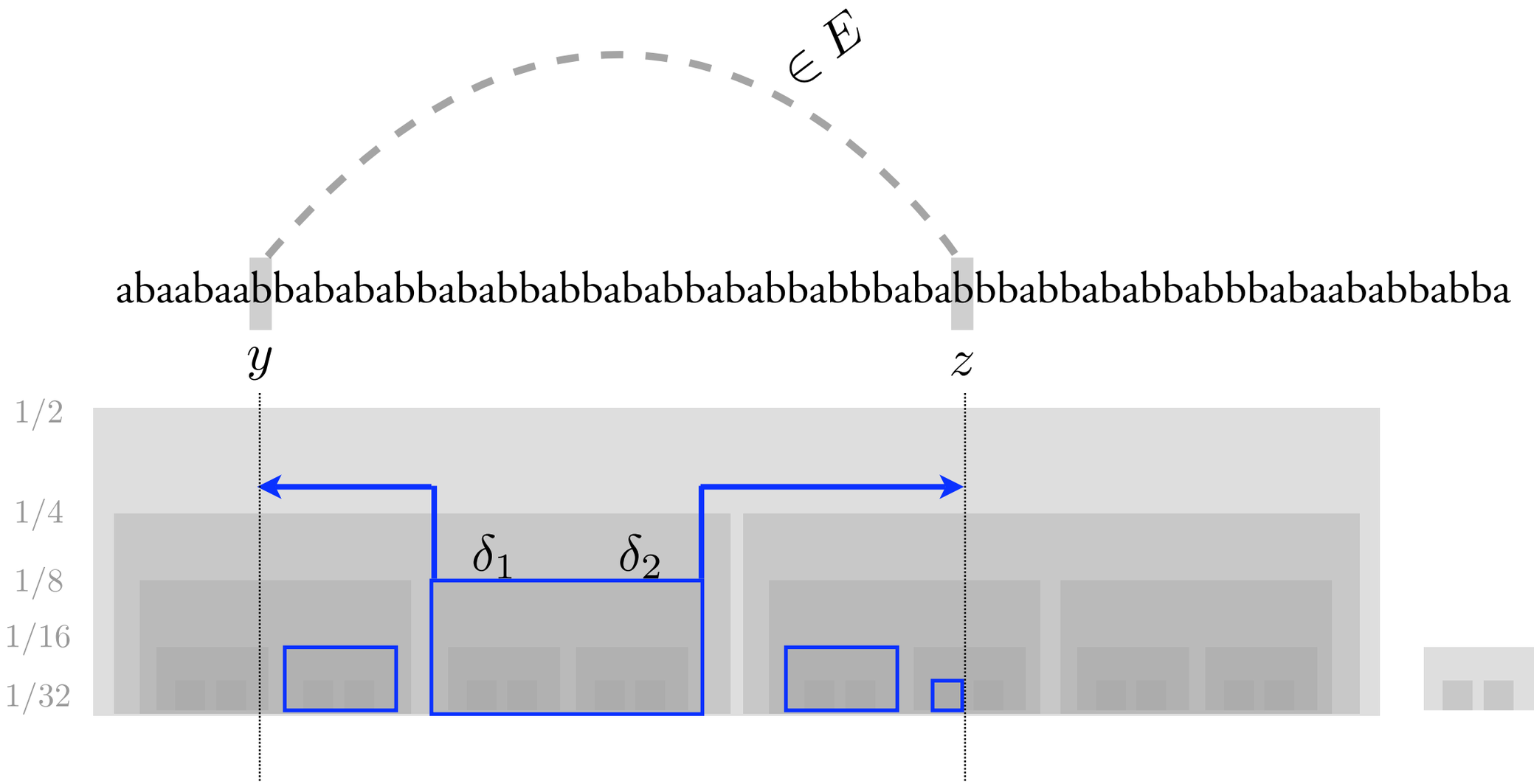
Find nodes x such that $w[y..x] \in L$ $w[x..z] \in K$

but only do logarithmically many operations each time

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

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

Output. Set of nodes x with



Divide and conquer dynamic algorithm.

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

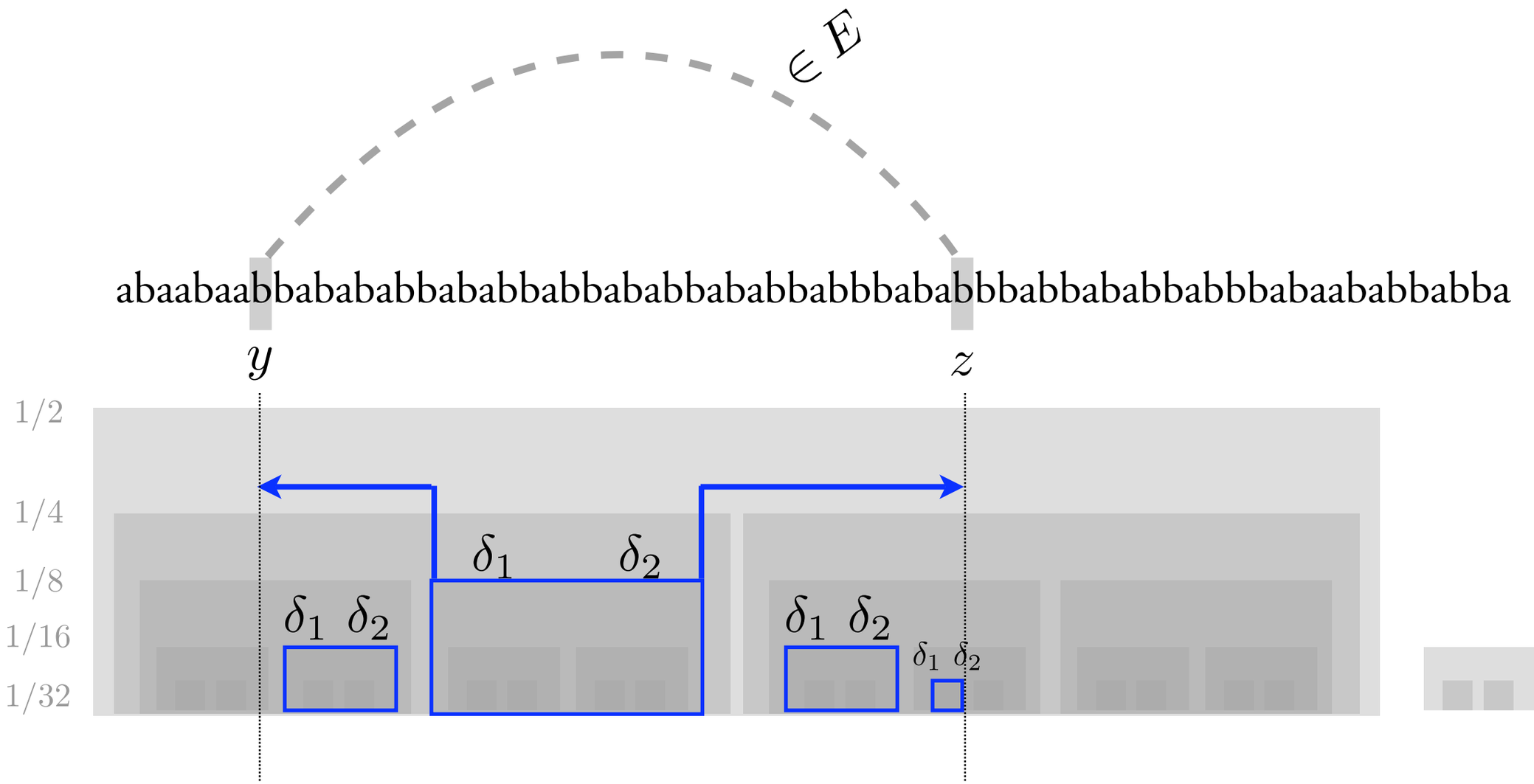
Find nodes x such that $w[y..x] \in L$ $w[x..z] \in K$

but only do logarithmically many operations each time

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

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

Output. Set of nodes x with



Divide and conquer dynamic algorithm.

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

Find nodes x such that $w[y..x] \in L$ $w[x..z] \in K$

but only do logarithmically many operations each time

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

abaabbbababbbbabba | *bbabbbabbbbabbaba*

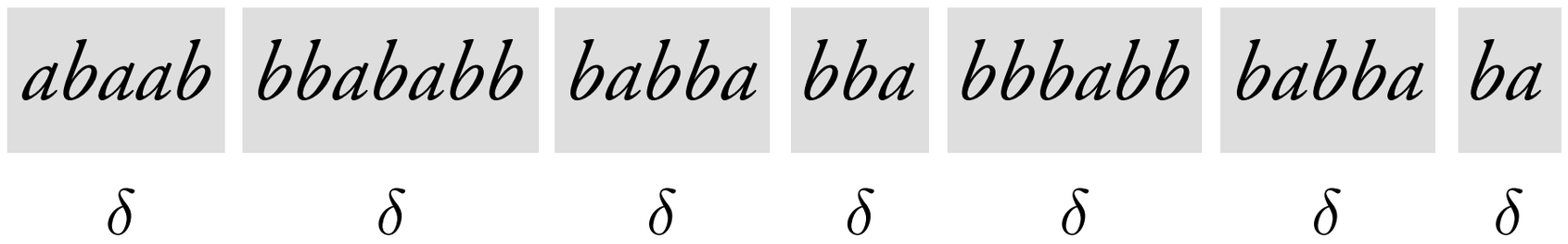
Simon Theorem. For fixed A, there is a splitting depth K, such that every word can be split in depth K down to single letters.

Let A be an automaton with state space Q

Two rules for splitting words.

Rule 1. split into two parts

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



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

abaabbbababbbbabba | *bbabbbabbbbabbaba*

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

abaab | *bbababb* | *babba* | *bba* | *bbbabb* | *babba* | *ba*

δ δ δ δ δ δ δ

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

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.