XML and Databases

Lecture 6

Node Selecting Queries: XPath 1.0

Sebastian Maneth

NICTA and UNSW

CSE@UNSW   -- Semester 1, 2010

Outline

1. XPath Data Model: **7 types of nodes**
2. Simple Examples
3. Location Steps and Paths
4. Value Comparison, and Other Functions

XPath

- → Query language to select (a sequence of) nodes of an XML document
- → W3C Standard
- → **Most important XML query language**: used in many other standards such as XQuery, XSLT, XPointer, XLink, …
- → Supported by *every modern web browser* for Java Script processing!
- → Cave: version 2.0 is considerably more expressive than 1.0
  We study XPath 1.0

Terminology: Instead of XPath “query” we often say *XPath expression*.

(An expression is the primary construction of the XPath grammar;
it matches the production Expr of the XPath grammar.)

Outline - Lectures

1. Introduction to XML, Encodings, Parsers
2. Memory Representations for XML: Space vs Access Speed
3. RDBMS Representation of XML
4. DTDs, Schemas, Regular Expressions, Ambiguity
5. XML Validation using Automata

6. Node Selecting Queries: **XPath**

7. Tree Automata for Efficient **XPath** Evaluation, Parallel Evaluation
8. .**XPath** Properties: backward axes, containment test
9. Streaming Evaluation: how much memory do you need?
10. **XPath** Evaluation using RDBMS

XPath

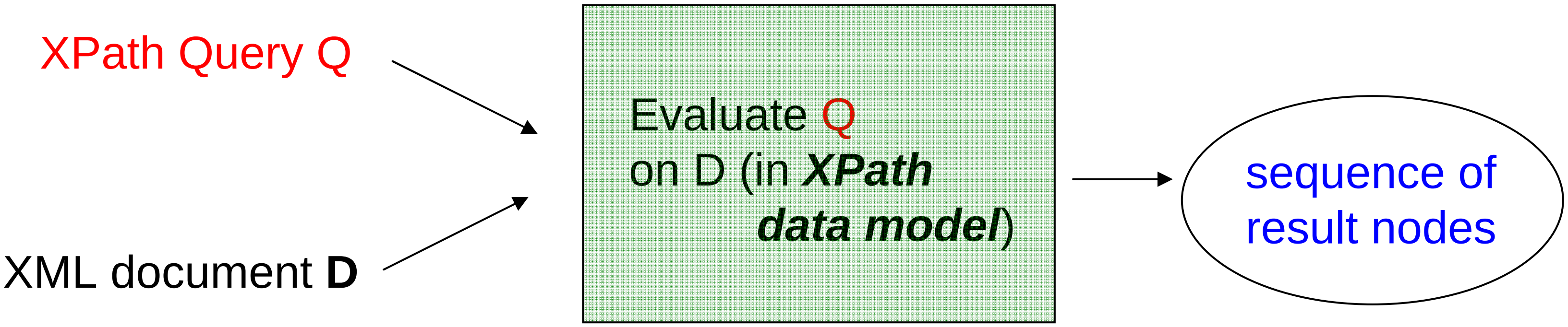
11. XSLT – stylesheets and transform
12. XQuery – XML query language
13. Wrap up, Exam Preparation, Open questions, etc

Outline - Assignments

1. Read XML, using DOM parser. Create document statistics.
2. SAX Parse into memory structure: Tree and DAG

3. Map XML into RDBMS → 29. April
4. **XPath evaluation** → 17. May
5. **XPath** into SQL Translation → 31. May

XPath Data Model



XPath Query Q

XML document **D**

Evaluate Q on D (in ***XPath data model***)

sequence of result nodes

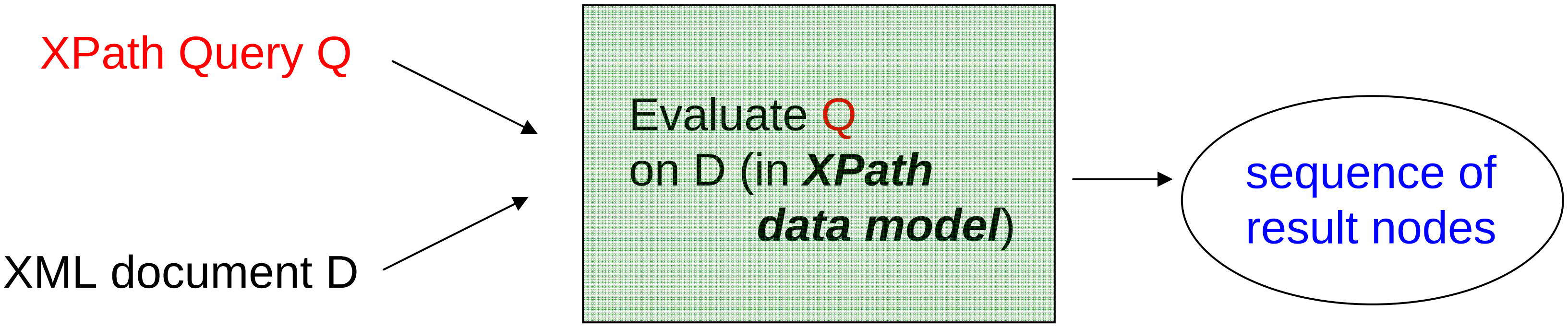
Document **D** is modeled as a **tree**.

THERE ARE SEVEN TYPES OF NODES in the XPath Data Model:

7 node types

- → root nodes
- → element nodes
- → text nodes
- → attribute nodes
- → namespace nodes
- → processing instruction nodes
- → comment nodes

XPath Data Model



Document D is modeled as a **tree**.

THERE ARE SEVEN TYPES OF NODES in the XPath Data Model:

7 node types

- → root nodes
- → element nodes
- → text nodes
- → attribute nodes
- → namespace nodes
- → processing instruction nodes
- → comment nodes

for rest of lecture:
this is ALL you need
to know about
XML nodes! ☺

XPath Data Model

5.2.1 Unique IDs
An element node may have a unique identifier (ID).
→ Value of the attribute that is declared in the DTD as type ID.
→ No two elements in a document may have the same unique ID.
→ If an XML processor reports two elements in a document as having the same unique ID (which is possible only if the document is invalid)
then the second element in doc. order must be treated as **not** having a unique ID.

NOTE: If a document has no DTD, then no element will have a unique ID.

→ root nodes
→ element nodes
→ text nodes
→ attribute nodes
→ namespace nodes
→ processing instruction nodes
→ comment nodes

for rest of lecture:
this is ALL you need
to know about
XML nodes! ☺

XPath Data Model

Document D is modeled as a **tree.**

For each node a **string-value** can be determined. (sometimes part of the node, sometimes computed from descendants, sometimes expanded-name:
local name + namespace URI)

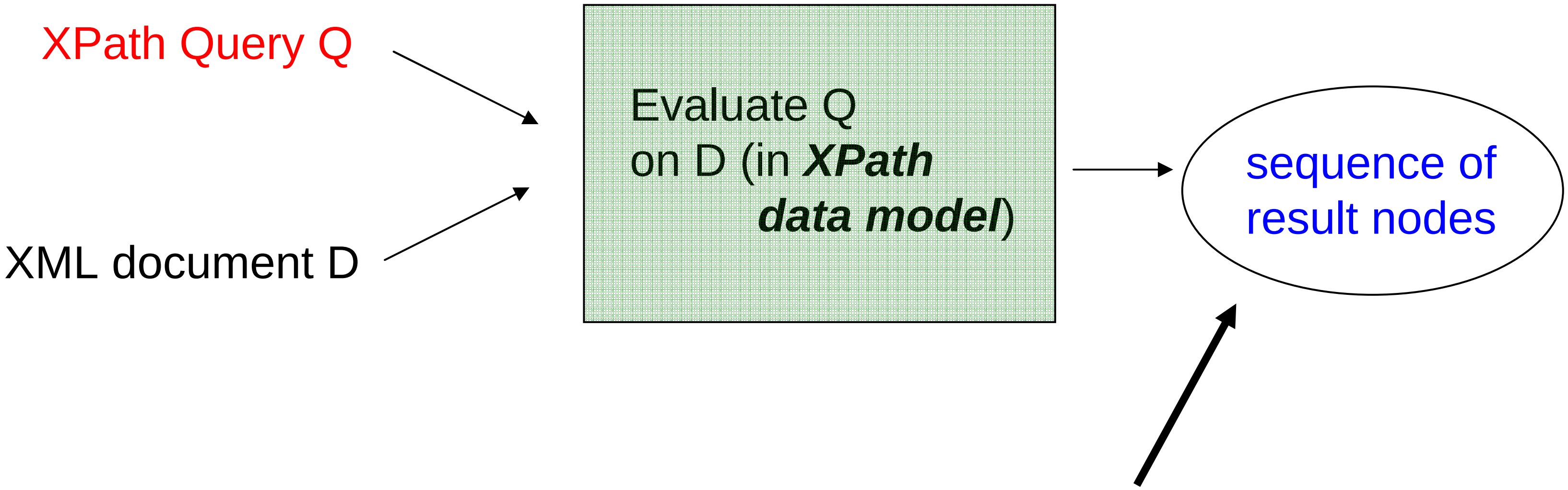
There is an order, **document order**, defined on all nodes. → corresponds to the position of the first character of the XML representation of the node, in the document (after entity expansion)

→ Attribute and namespace nodes appear
before the children of an element.

→ Order of attribute and namespace nodes is *implementation-dependent*

Every node (besides root) has
exactly one parent (which is a root or an element node)

XPath Result Sequences



XPath Query Q

XML document D

Evaluate Q on D (in ***XPath data model***)

sequence of result nodes

→ Ordered in **document order**
→ Contains **no duplicates**

Simple Examples

In abbreviated XPath syntax.

`Q0: /`

Selects the document root
(always the parent of the document element)

Document:

```
<bib>
  <book>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
    <title>Foundations of Databases</title>
    <year>1995</year>
  </book>
  <book>
    <author>Ullmann</author>
    <title>Principles of Database and Knowledge Base Systems</title>
    <year>1998</year>
  </book>
</bib>
```

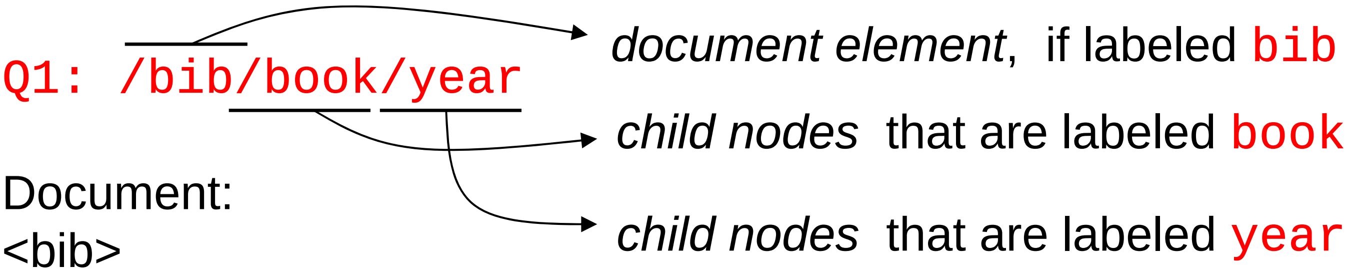
document root is virtual and invisible, in this example.

If <?xml version="1.0"?> is present, then it is returned (as first entry) in the result of `Q0`.

Note XPath Evaluators usually return the full subtree of the selected node.

Simple Examples

In abbreviated syntax.



Q1: /bib/book/year

document element, if labeled bib

child nodes that are labeled book

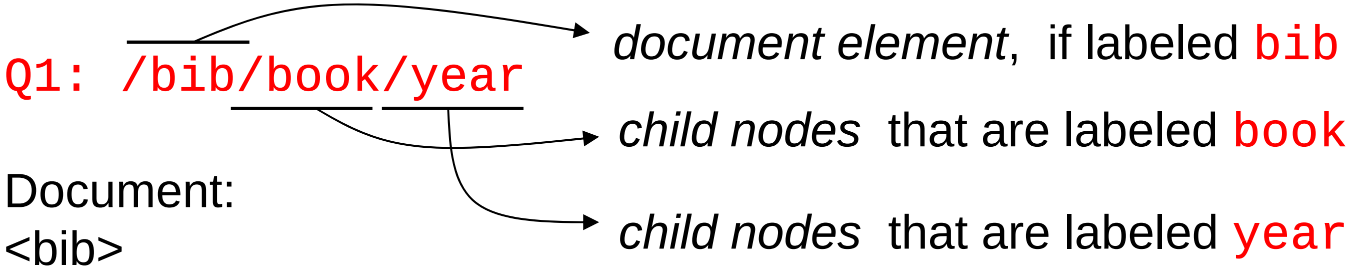
child nodes that are labeled year

Document:

```
<bib>
  <book>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
    <title>Foundations of Databases</title>
    <year>1995</year>
  </book>
  <book>
    <author>Ullmann</author>
    <title>Principles of Database and Knowledge Base Systems</title>
    <year>1998</year>
  </book>
</bib>
```

Simple Examples

In abbreviated syntax.



Q1: /bib/book/year

document element, if labeled bib

child nodes that are labeled book

child nodes that are labeled year

Document:

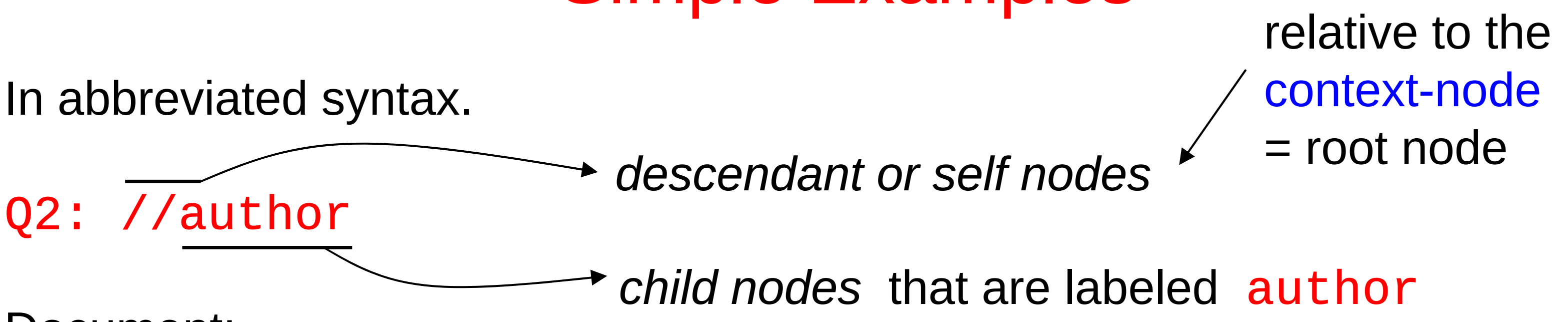
```
<bib>
  <book>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
    <title>Foundations of Databases</title>
    <year>1995</year>
  </book>
  <book>
    <author>Ullmann</author>
    <title>Principles of Database and Knowledge Base Systems</title>
    <year>1998</year>
  </book>
</bib>
```

Result of query Q1 =
(element) nodes N1, N2

subtree at N1 is <year>1995</year>
and subtree at N2 is <year>1998</year>

Simple Examples

In abbreviated syntax.



Q2: //author

descendant or self nodes

relative to the context-node = root node

child nodes that are labeled author

Document:

```
<bib>
 <book>
  <author>Abiteboul</author>
  <author>Hull</author>
  <author>Vianu</author>
  <title>Foundations of Databases</title>
  <year>1995</year>
 </book>
 <book>
  <author>Ullmann</author>
  <title>Principles of Database and Knowledge Base Systems</title>
  <year>1998</year>
 </book>
</bib>
```

// is short for /descendant-or-self::node()/

For example, //author is short for /descendant-or-self::node()/child::author

Simple Examples

In abbreviated syntax.

relative to the
context-node
= root node

`Q2: //author`

Descendant or self nodes
that are labeled `author`

Document:

```
<bib>
  <book>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
    <title>Foundations of Databases</title>
    <year>1995</year>
  </book>
  <book>
    <author>Ullman</author>
    <title>Principles of Database and Knowledge Base Systems</title>
    <year>1998</year>
  </book>
</bib>
```

Result of query `Q2` =
sequence of (element) nodes
(N1, N2, N3, N4)

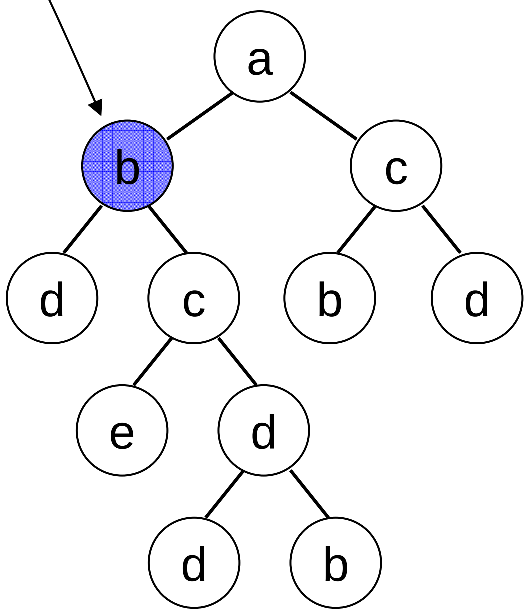
// is short for /descendant-or-self::node()/
For example, //author is short for /descendant-or-self::node()/child::author

Simple Examples

In abbreviated syntax.

Q3: $\overline{\texttt{/a/b}}\underline{\texttt{//d}}$

“b-child of a-doc. element”

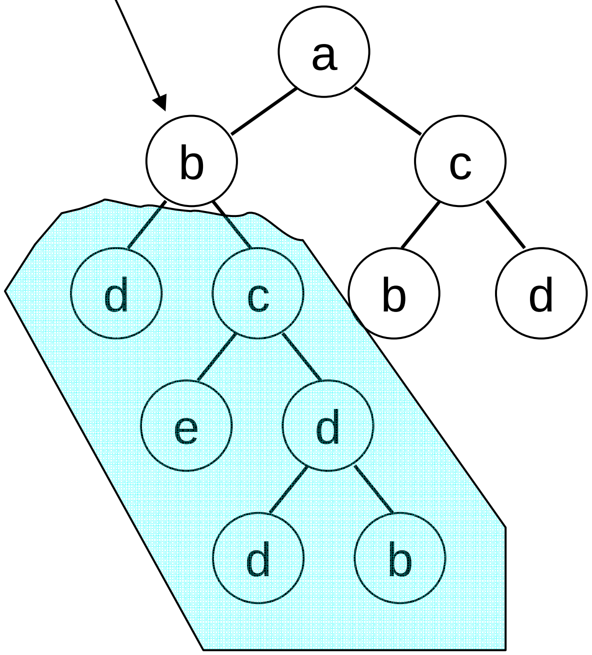


Simple Examples

In abbreviated syntax.

`Q3: /a/b//d`

“b-child of a-doc. element”



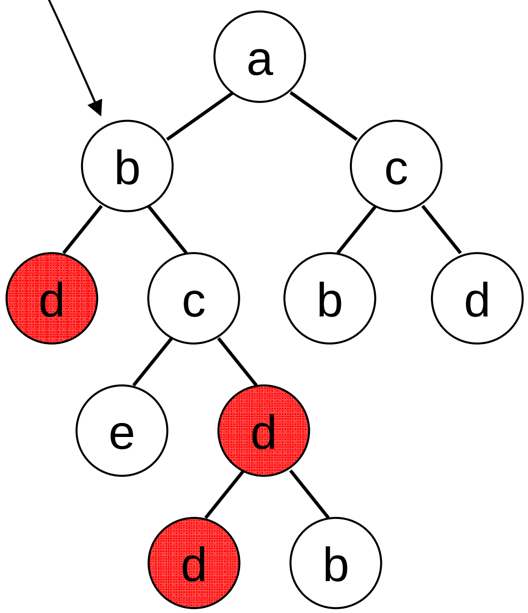
ALL d-nodes
in these subtrees

Simple Examples

In abbreviated syntax.

`Q3: /a/b//d`

“b-child of a-doc. element”

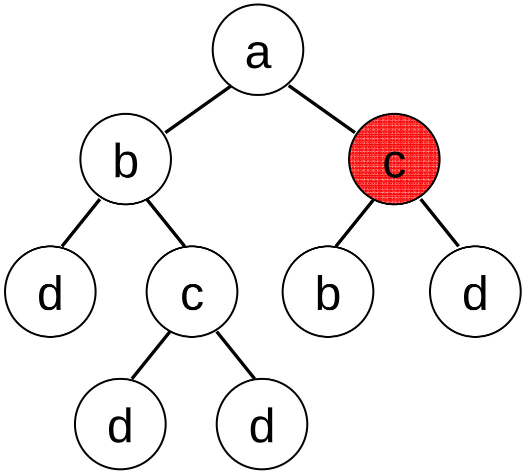


ALL d-nodes
in these subtrees

Simple Examples

In abbreviated syntax.

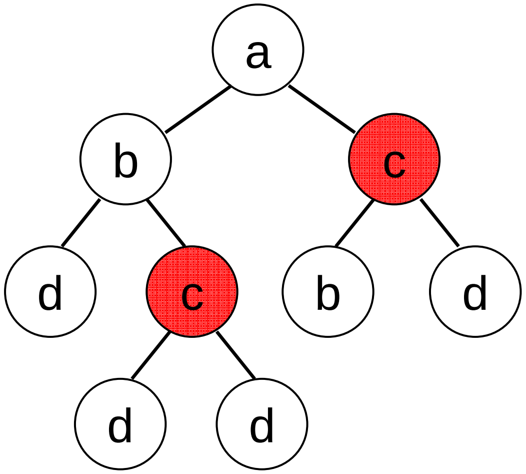
`Q4: /*/c`



Simple Examples

In abbreviated syntax.

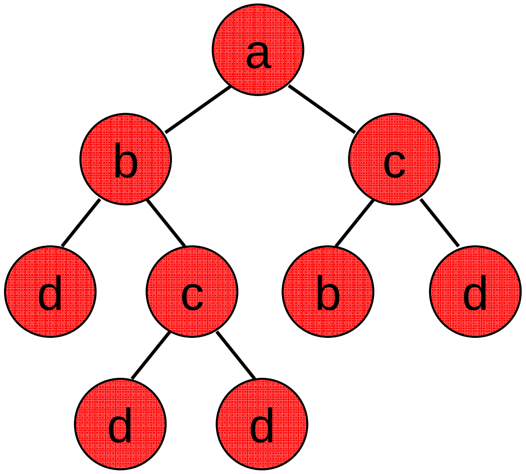
`Q5: //c`



Simple Examples

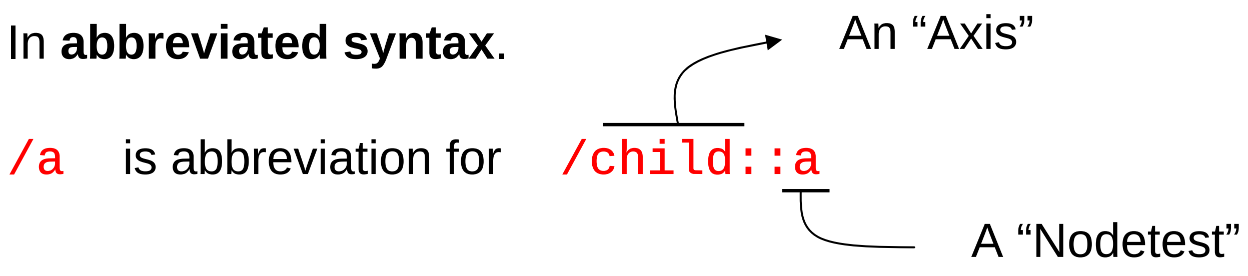
In abbreviated syntax.

`Q6: //*`



Abbreviations, so far

In **abbreviated syntax**.



`/a` is abbreviation for `/child::a`

`//a` is abbreviation for `/descendant-or-self::node()/child::a`

→ Child and descendant-or-self are only 2 out of **12 possible axes**.

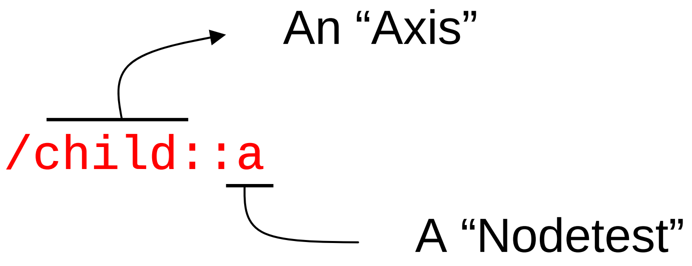
An “axis” is a sequence of nodes. It is evaluated relative to a context-node.

Other axes:
→ `descendant`
→ `parent`
→ `ancestor-or-self`
→ `ancestor`
→ `following-sibling`
→ `preceding-sibling`
→ `attribute`
→ `following`
→ `preceding`
→ `self`

Abbreviations, so far

In **abbreviated syntax**.

`/a` is abbreviation for `/child::a`



`//a` is abbreviation for `/descendant-or-self::node()/child::a`

`//` is abbreviation for `/descendant-or-self::node()/`

`.` is abbreviation for `self::node()`

`..` is abbreviation for `parent::node()`

→ Child and descendant-or-self are only 2 out of **12 possible axes**.

An “axis” is a sequence of nodes. It is evaluated relative to a context-node.

Other axes:

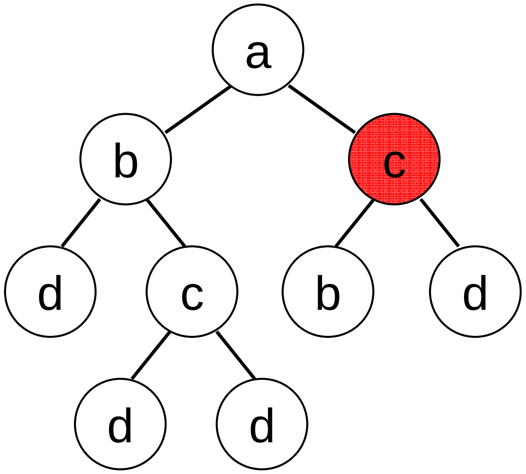
- → `descendant`
- → `parent`
- → `ancestor-or-self`
- → `ancestor`
- → `following-sibling`
- → `preceding-sibling`
- → `attribute`
- → `following`
- → `preceding`
- → `self`

Examples: Predicates

In abbreviated syntax.

`Q7: //c[./b]`

“*has b-child*” (context-nodes are all c-nodes…)

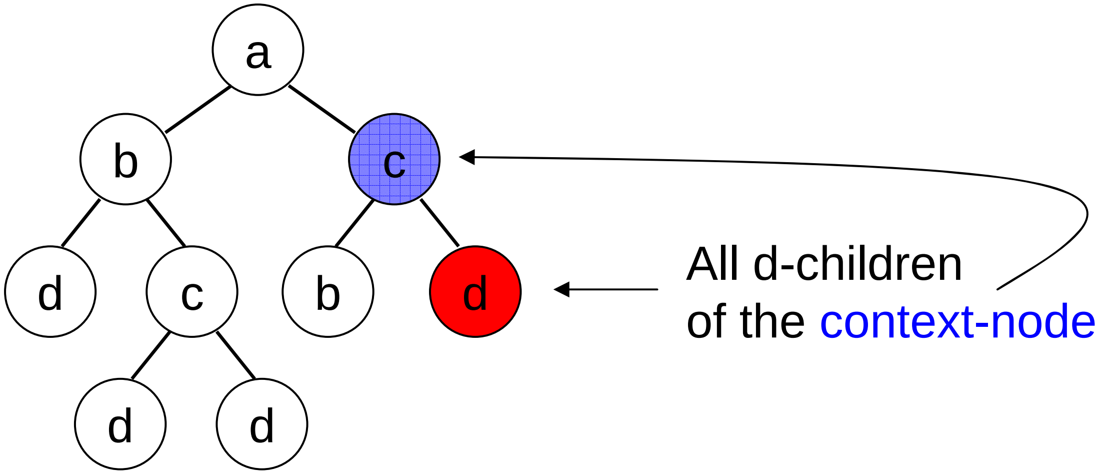


Examples: Predicates

In abbreviated syntax.

```
Q8: //c[./b]/d
```

“*has b-child*”



All d-children
of the context-node(s)

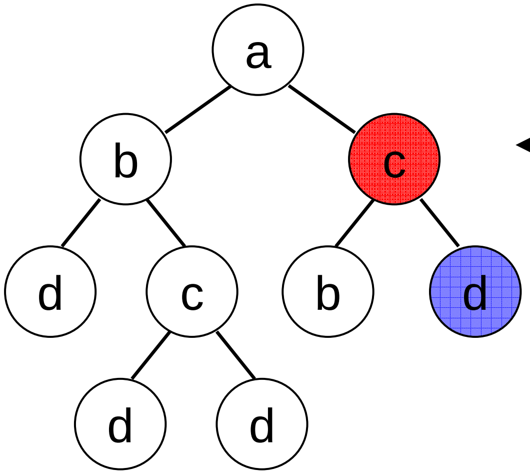
Examples: Predicates

In abbreviated syntax.

Q9: //c[./b]/d/..

"*has b-child*"

select **parent(s)** of context-node(s)



parent(s) of the context-node(s)

Q9 selects c-nodes that "*have a b-child AND a d-child*"

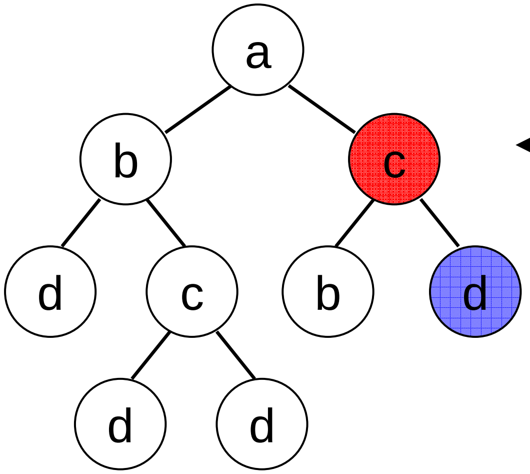
Examples: Predicates

In abbreviated syntax.

`Q9: //c[./b]/d/..`

“*has b-child*”

select **parent(s)**
of context-node(s)



parent(s)
of the context-node(s)

`Q9` selects c-nodes that “*have a b-child AND a d-child*”

More direct way: `//c[./b and ./d]`

(same as
`//c[./b]`
on *this* tree..!)

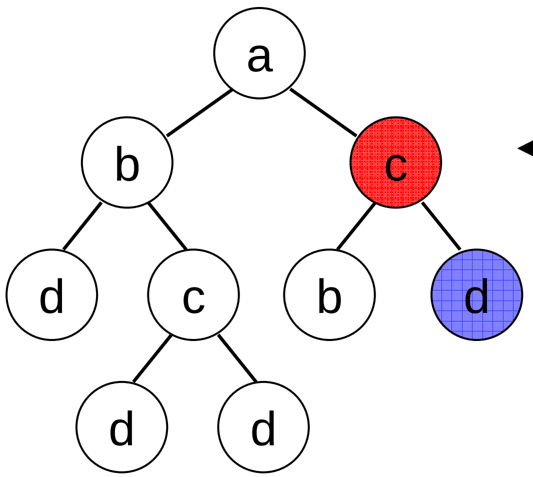
Examples: Predicates

In abbreviated syntax.

`Q9: //c[b]/d/..`

“*has b-child*”

select **parent(s)**
of context-node(s)



parent(s)
of the context-node(s)

`Q9` selects c-nodes that “*have a b-child AND a d-child*”

(same as
`//c[./b]`
on *this* tree..!)

More direct way: `//c[b and d]`

We do not need `“./b”` → `self::node()/child::b` equivalent to `b`

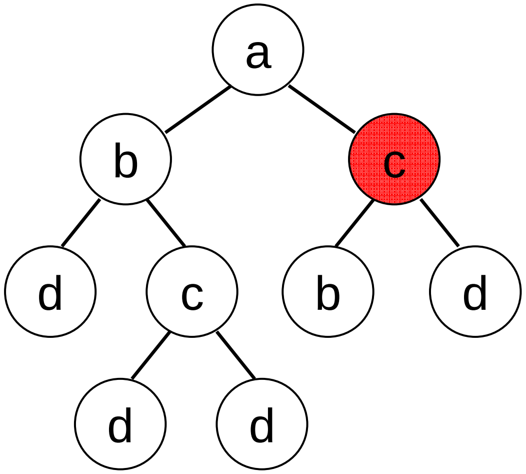
Examples: Predicates (or “Filters”)

In abbreviated syntax.

```
//c[b and d]
```

evaluates to **true/false**

A “*Filter*”



c-nodes that “*have a b-child AND a d-child*”

Examples: Predicates (or “Filters”)

In abbreviated syntax.

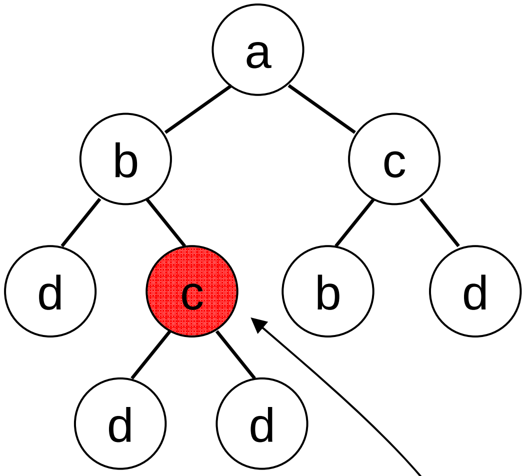
`//c[b and d]`

`b and d` evaluates to **true/false**

A “*Filter*”

Question

How to only select
the other c-node?



Can use “`not( … )`” in a filter!

`//c[not(b)]`

“does not have a b-child”

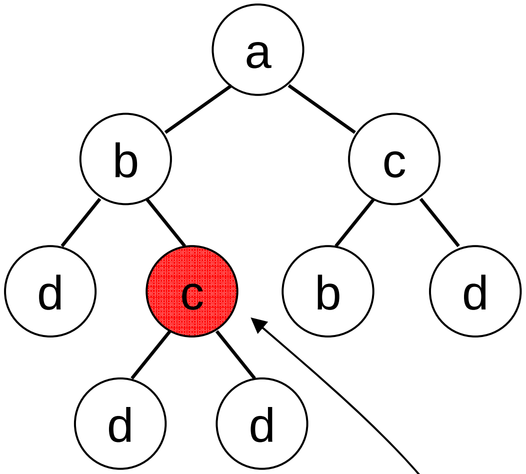
Examples: Predicates

In abbreviated syntax.

```
//c[b and d]
```

evaluates to **true/false**

A “*Filter*”



Question

How to only select the other c-node?

Many more possibilities, of course:

```
//c[parent::b]
//c[../../b]
//c[../d]
```

```
CAVE: what does
//c[../b] give??
```

Can use “`not( … )`” in a filter!

```
//c[not(b)]
```

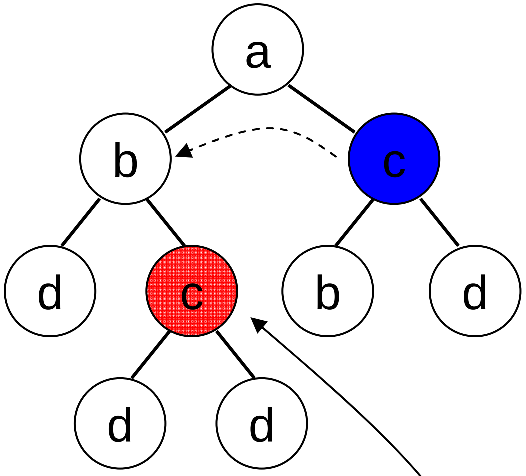
Examples: Predicates

In abbreviated syntax.

```
//c[b and d]
```

evaluates to **true/false**

A “*Filter*”



Can use “`not( … )`” in a filter!

```
//c[not(b)]
```

Question

How to only select
the other c-node?

Many more
possibilities, of course:

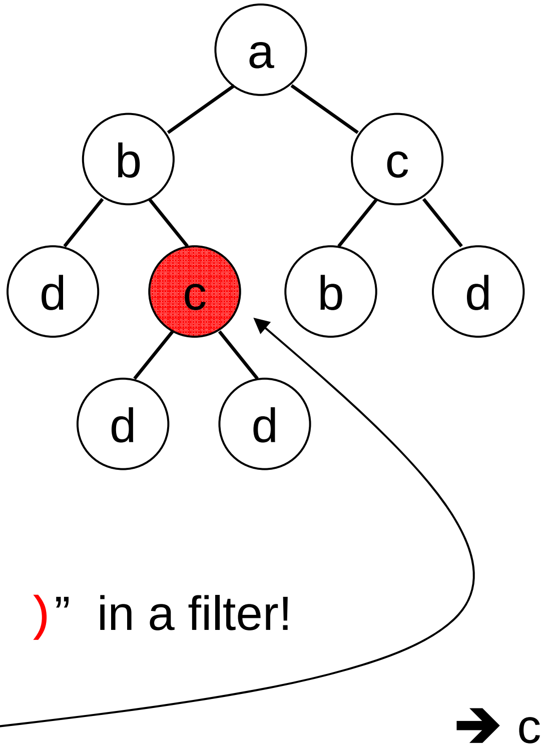
```
//c[parent::b]
//c[../../b]
//c[../d]
```

CAVE: what does
`//c[../b]` give??

Examples: Predicates

In abbreviated syntax.

$$//c\underbrace{[b \text{ and } d]}_{\text{A "}\textit{Filter}\text{"}}$$ evaluates to **true/false**



Question

How to only select
the other c-node?

Many more
possibilities, of course:

```
//c[parent::b]
//c[../../b]
//c[../d]
```

Can use "`not( … )`" in a filter!

```
//c[not(b)]
```

➔ can you say
"c-node that has only d-children"?

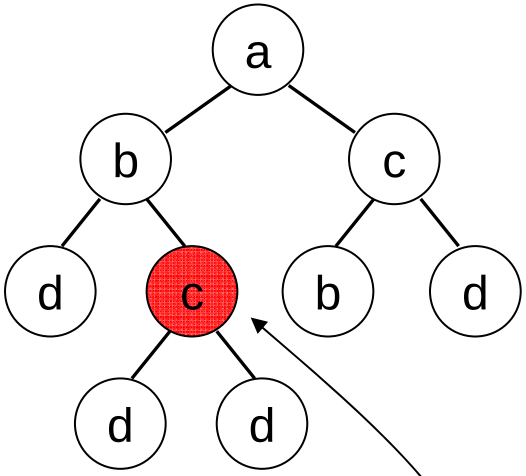
Examples: Predicates

In abbreviated syntax.

`//c[b and d]`

evaluates to **true/false**

A “*Filter*”



Question

How to only select
the other c-node?

Many more
possibilities, of course:

```
//c[parent::b]
//c[../../b]
//c[../d]
```

Can use “`not( … )`” in a filter!

`//c[not(b)]`

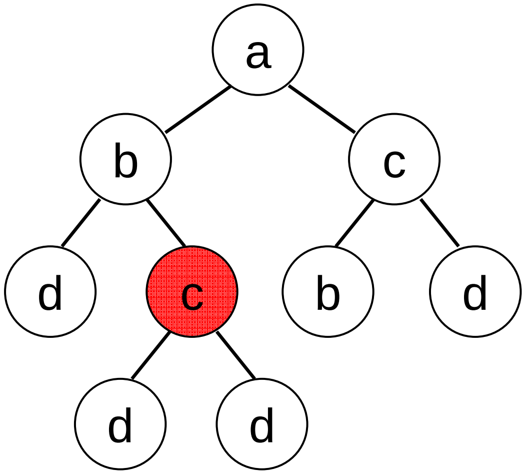
➔ can you say
“c-node that has only d-children”?

YES! needs a bit of logic… `//c[not(child::*[not(self::d)])]`

Examples: Predicates

In abbreviated syntax.

`//c[not(b)]` same as .. on this tree `//c[not(child::*[not(self::d)])]`



“not the case that
all children are not labeled d”

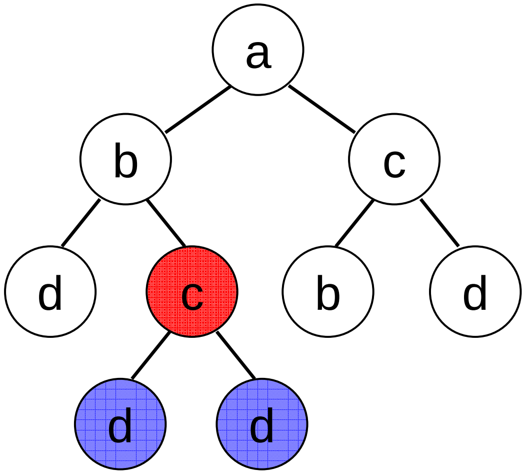
holds if and only if

“all children are labeled d”

Examples: Predicates

In abbreviated syntax.

`//c[not(b)]` same as .. on this tree `//c[not(child::*[not(self::d)])]`



“not the case that
all children are not labeled d”

holds if and only if

“all children are labeled d”

Duplicate elimination

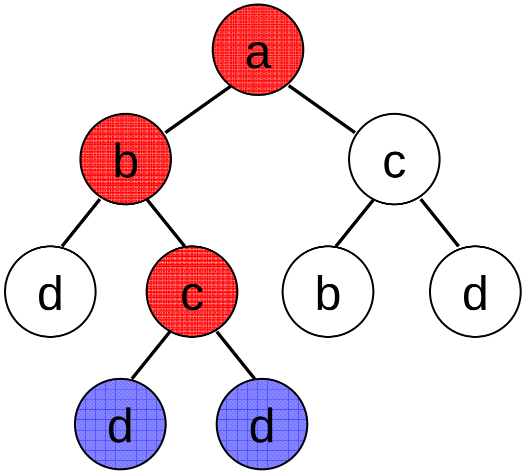
`//c[not(b)]/d/..`

context-nodes
for parent selection (/..)

Examples: Predicates

In abbreviated syntax.

`//c[not(b)]` same as .. on this tree `//c[not(child::*[not(self::d)])]`



“not the case that all children are not labeled d”

holds if and only if

“all children are labeled d”

Duplicate elimination

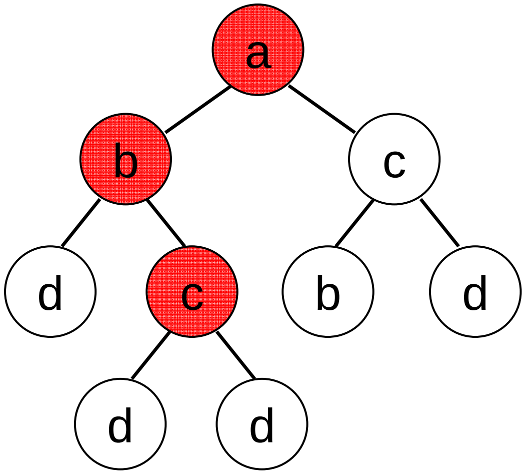
context-nodes
for ancestor selection

`//c[not(b)]/d/ancestor::*`

Examples: Predicates

In abbreviated syntax.

`//c[not(b)]` same as ..
on this tree `//c[not(child::*[not(self::d)])]`



"not the case that
all children are not labeled d"

holds if and only if

"all children are labeled d"

maybe
→ `//*[.//c[not(b)]]`

Duplicate elimination

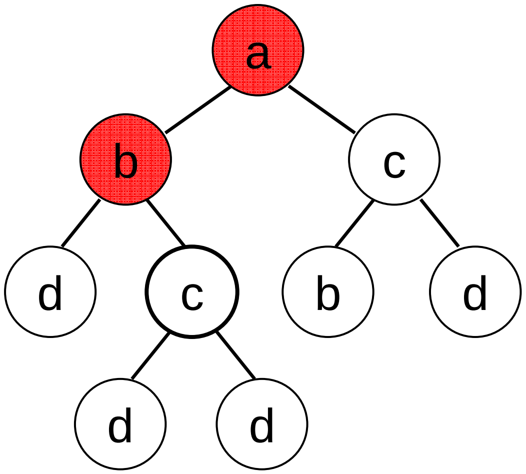
`//c[not(b)]/d/ancestor::*`

Equivalent one, *without use of* `ancestor`??

Examples: Predicates

In abbreviated syntax.

`//c[not(b)]` same as ..
on this tree `//c[not(child::*[not(self::d)])]`



“not the case that
all children are not labeled d”

holds if and only if

“all children are labeled d”

maybe
→`//*[.//c[not(b)]]`
No.. ☹

How to select the c-node?

Duplicate elimination

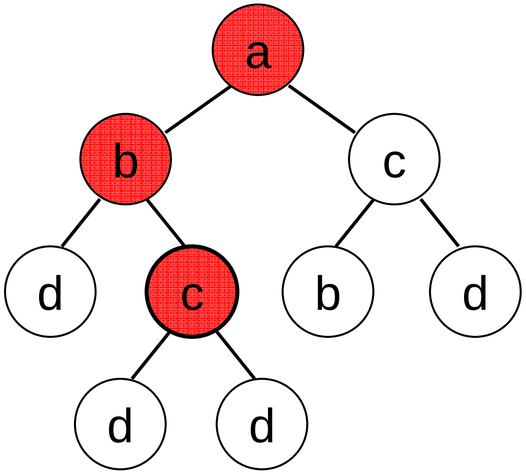
`//c[not(b)]/d/ancestor::*`

No use of `ancestor`?

Examples: Predicates

In abbreviated syntax.

`//c[not(b)]` same as .. on this tree `//c[not(child::*[not(self::d)])]`



“not the case that
all children are not labeled d”

holds if and only if

“all children are labeled d”

maybe
→`//*[.//c[not(b)]]`
No.. ☹

How to select the c-node?

Duplicate elimination

`//c[not(b)]/d/ancestor::*`

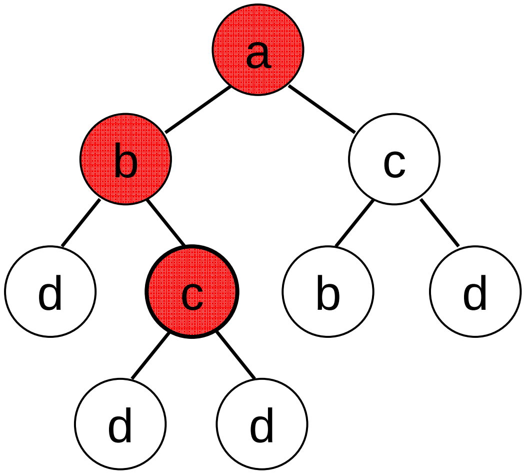
No use of `ancestor`?

→`//*[descendant-or-self::c[not(b)]]`

Examples: Predicates

In abbreviated syntax.

`//c[not(b)]` same as ..
on this tree `//c[not(child::*[not(self::d)])]`



“not the case that
all children are not labeled d”

holds if and only if

“all children are labeled d”

Duplicate elimination

`//c[not(b)]/d/ancestor::*`

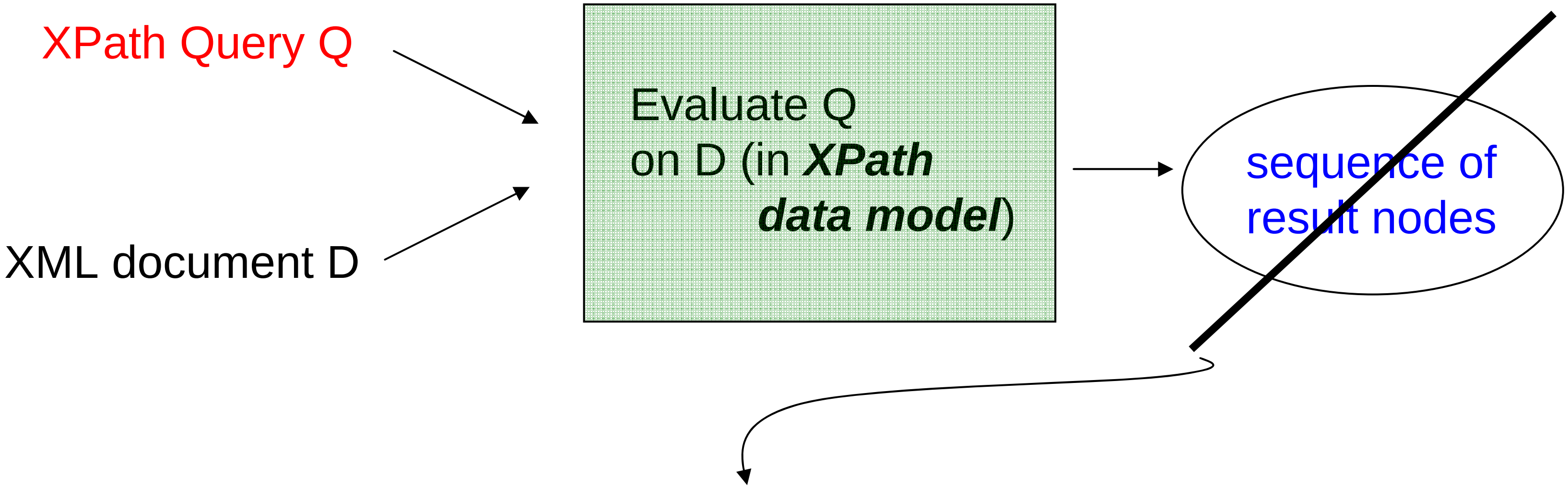
maybe
→`//*[.//c[not(b)]]`
No.. ☹

How to select the c-node?

`//*[.//c[not(b)] or not(child::*[not(self::d)]) and ./*]`

“only d-children” “has child (not leaf)”

More Details



XPath Query Q

XML document D

Evaluate Q on D (in ***XPath* *data model***)

~~sequence of result nodes~~

NOT correct (at least not for intermediate expr's)

An expression evaluates to an object, which has one of the following **four basic types**

- node-set (an unordered collection of nodes w/o duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)

Location Steps & Paths

→ A Location Path is a sequence of Location Steps

➔ Initial Context
will be is root node

Location Paths

[1] LocationPath ::= RelativeLocationPath
| AbsoluteLocationPath

[2] AbsoluteLocationPath ::= '/' RelativeLocationPath?
| AbbreviatedAbsoluteLocationPath

[3] RelativeLocationPath ::= Step
| RelativeLocationPath '/' Step
| AbbreviatedRelativeLocationPath

Location Steps

[4] Step ::= AxisSpecifier NodeTest Predicate*
| AbbreviatedStep

[5] AxisSpecifier ::= AxisName '::‘
| AbbreviatedAxisSpecifier

Location Steps & Paths

→ A Location Path is a sequence of Location Steps

→ A Location Step is of the form

axis :: nodetest [Filter_1] [Filter_2] … [Filter_n]

Filters (aka predicates, (filter) expressions)
→ evaluate to **true/false**
→ XPath queries, evaluated with
context-node = current node

Boolean operators: **and, or**

Empty string/sequence are converted to **false**

Location Steps & Paths

→ A Location Path is a sequence of Location Steps

→ A Location Step is of the form

axis :: nodetest [Filter_1] [Filter_2] … [Filter_n]

Filters (aka predicates, (filter) expressions)
evaluate to **true/false**

nodetest: * or node-name (could be expanded →namespaces) or

- → text()
- → comment()
- → processing-instruction(ln)
- → node()

Example `child::text()` “*select all text node children of the context node*”

→ the nodetest node() is true for any node.

`attribute::*` “*select all attributes of the context node*”

Location Steps & Paths

→ A Location Path is a sequence of Location Steps

→ A Location Step is of the form

axis :: nodetest [Filter_1] [Filter_2] … [Filter_n]

Filters (aka predicates, (filter) expressions) evaluate to **true/false**

nodetest: * or node-name (could be expanded →namespaces) or

→ text()
→ comment()
→ processing -instruction(ln)
→ node()

12 Axes

Forward Axes:

```
→ self
→ child
→ descendant-or-self
→ descendant
→ following
→ following-sibling
```

In doc order

Backward Axes:

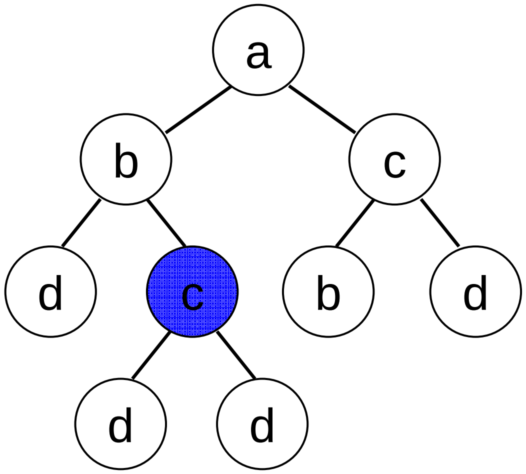
```
→ parent
→ ancestor
→ ancestor-or-self
→ preceding
→ preceding-sibling
```

reverse doc order

```
→ attribute
```

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

- → self
- → child
- → descendant-or-self
- → descendant
- → following
- → following-sibling

In doc order

Backward Axes:

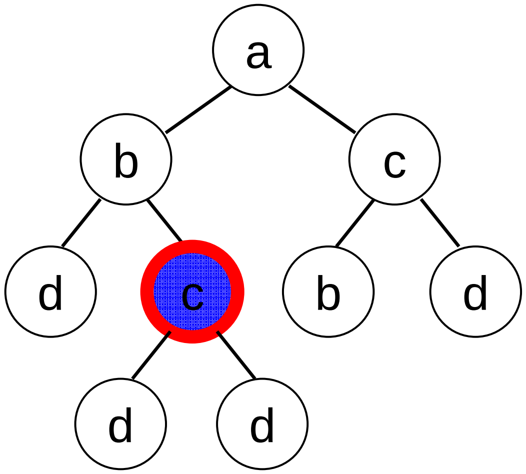
- → parent
- → ancestor
- → ancestor-or-self
- → preceding
- → preceding-sibling

reverse doc order

→ attribute

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

- ➔ self
- → child
- → descendant-or-self
- → descendant
- → following
- → following-sibling

In doc order

Backward Axes:

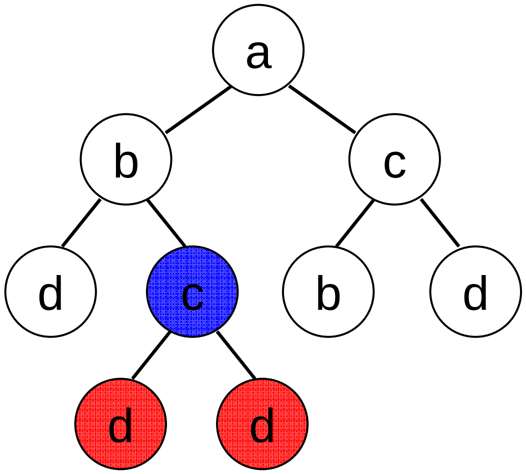
- → parent
- → ancestor
- → ancestor-or-self
- → preceding
- → preceding-sibling

→ attribute

reverse doc order

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

- → self
- → **child**
- → descendant-or-self
- → descendant
- → following
- → following-sibling

In doc order

Backward Axes:

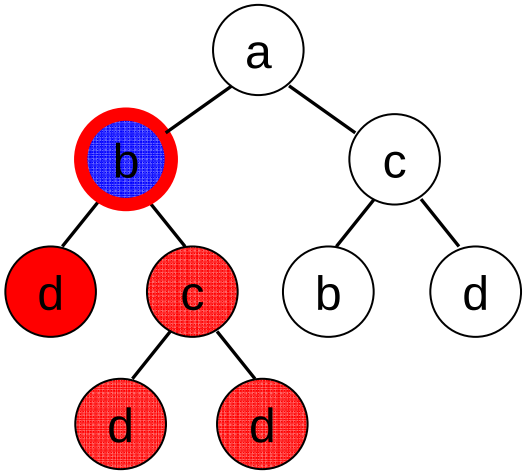
- → parent
- → ancestor
- → ancestor-or-self
- → preceding
- → preceding-sibling

→ attribute

reverse doc order

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

→ self
→ child
→ descendant-or-self
→ descendant
→ following
→ following-sibling

In doc order

Backward Axes:

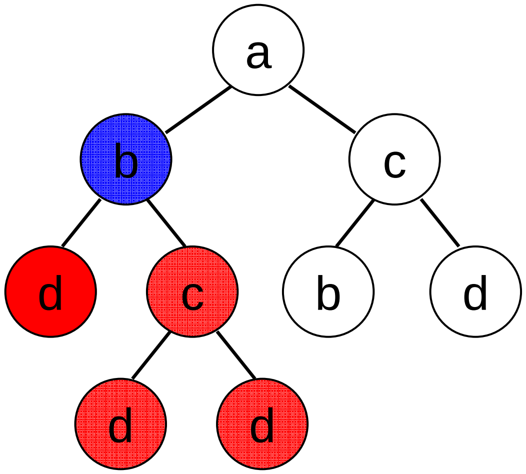
→ parent
→ ancestor
→ ancestor-or-self
→ preceding
→ preceding-sibling

→ attribute

reverse doc order

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

- → self
- → child
- → descendant-or-self
- → descendant
- → following
- → following-sibling

In doc order

Backward Axes:

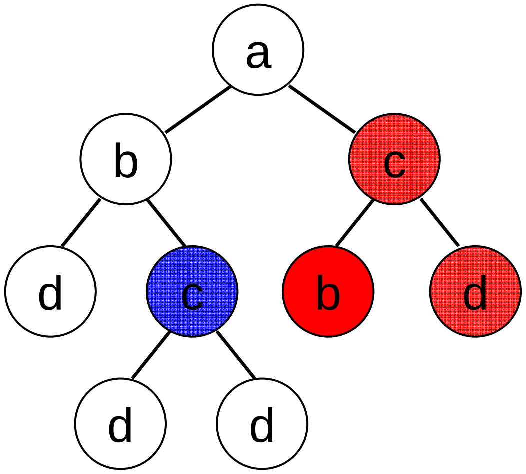
- → parent
- → ancestor
- → ancestor-or-self
- → preceding
- → preceding-sibling

→ attribute

reverse doc order

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

→ self

→ child

→ descendant-or-self

→ descendant

→ following

→ following-sibling

In doc order

Backward Axes:

→ parent

→ ancestor

→ ancestor-or-self

→ preceding

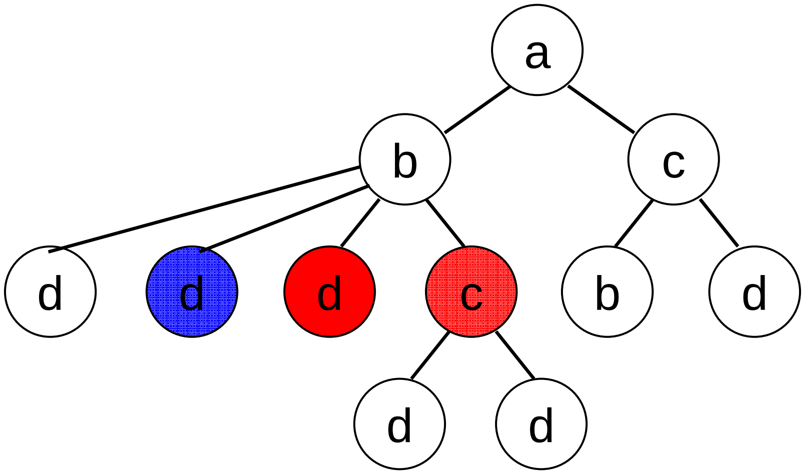
→ preceding-sibling

→ attribute

reverse doc order

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

→ self
→ child
→ descendant-or-self
→ descendant
→ following
→ following-sibling

In doc order

Backward Axes:

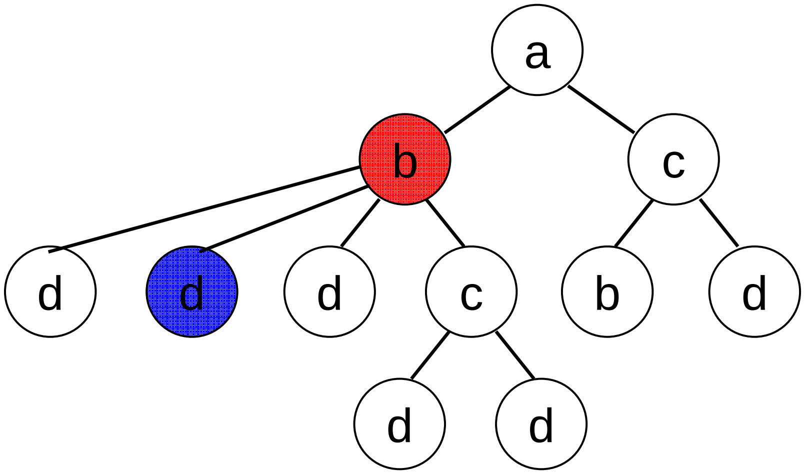
→ parent
→ ancestor
→ ancestor-or-self
→ preceding
→ preceding-sibling

→ attribute

reverse doc order

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

- → `self`
- → `child`
- → `descendant-or-self`
- → `descendant`
- → `following`
- → `following-sibling`

In doc order

Backward Axes:

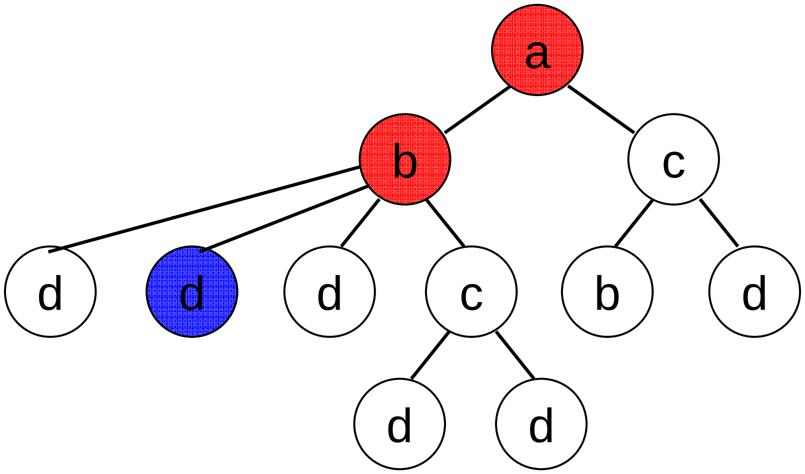
- → `parent`
- → `ancestor`
- → `ancestor-or-self`
- → `preceding`
- → `preceding-sibling`

→ `attribute`

reverse doc order

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

- → self
- → child
- → descendant-or-self
- → descendant
- → following
- → following-sibling

In doc order

Backward Axes:

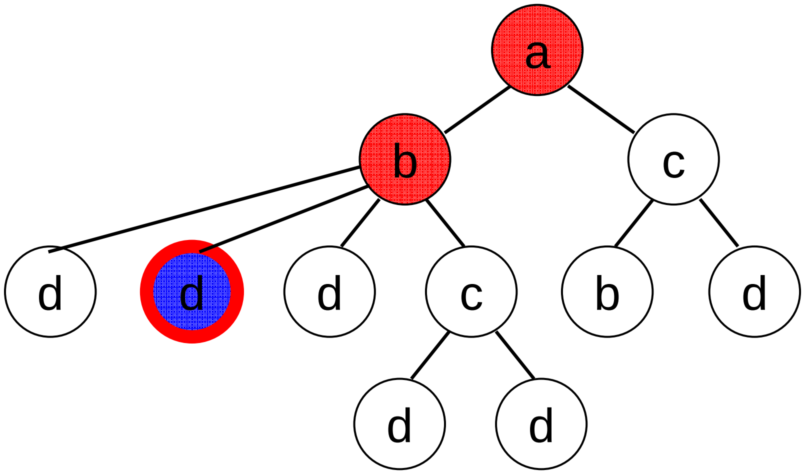
- → parent
- → ancestor
- → ancestor-or-self
- → preceding
- → preceding-sibling

→ attribute

reverse doc order

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

- → self
- → child
- → descendant-or-self
- → descendant
- → following
- → following-sibling

In doc order

Backward Axes:

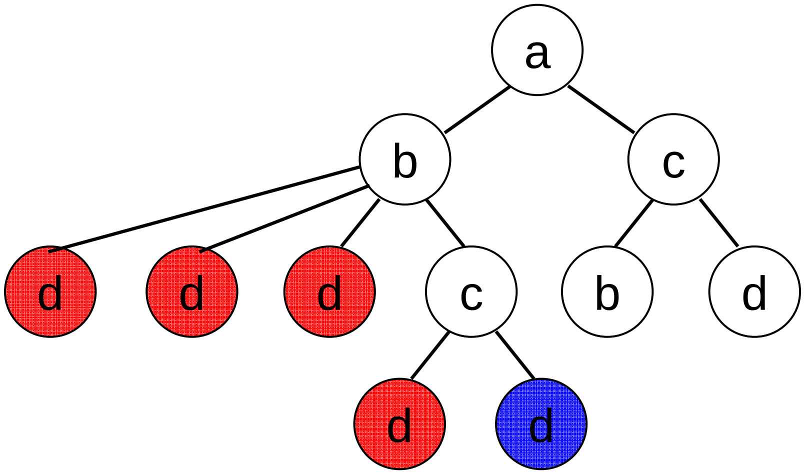
- → parent
- → ancestor
- → ancestor-or-self
- → preceding
- → preceding-sibling

reverse doc order

→ attribute

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

- → self
- → child
- → descendant-or-self
- → descendant
- → following
- → following-sibling

In doc order

Backward Axes:

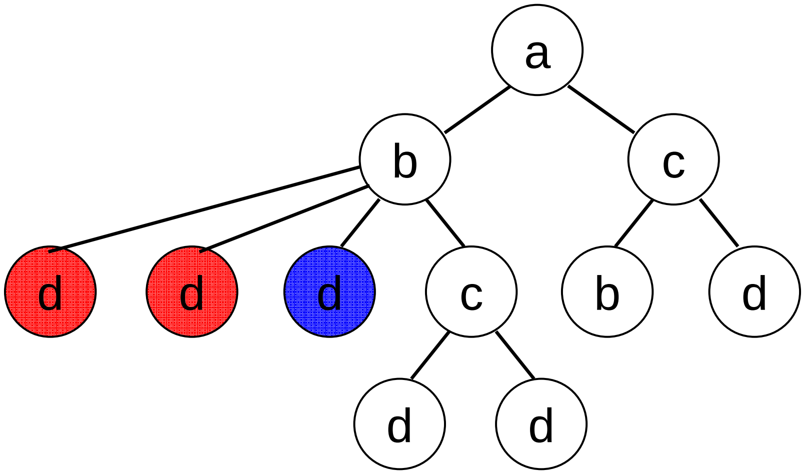
- → parent
- → ancestor
- → ancestor-or-self
- → preceding
- → preceding-sibling

→ attribute

reverse doc order

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

- → self
- → child
- → descendant-or-self
- → descendant
- → following
- → following-sibling

In doc order

Backward Axes:

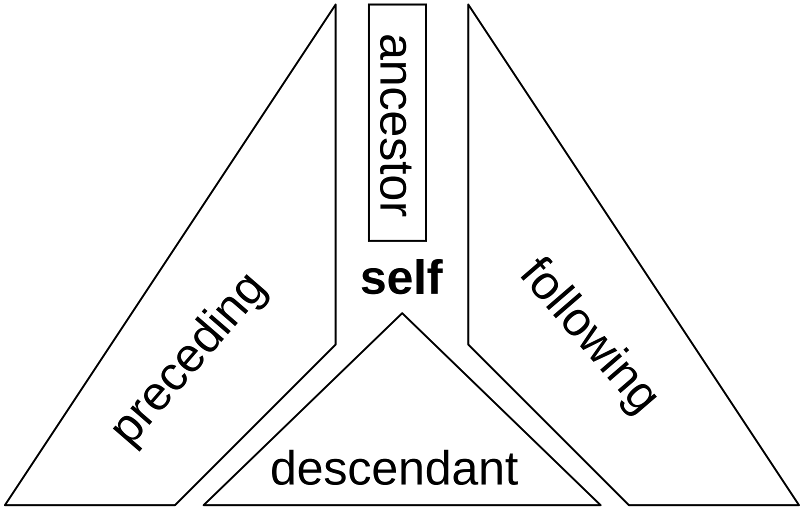
- → parent
- → ancestor
- → ancestor-or-self
- → preceding
- → preceding-sibling

→ attribute

reverse doc order

Location Steps & Paths

Axis = a sequence of nodes (is evaluated relative to context-node)



Forward Axes:

→ `self`
→ `child`
→ `descendant-or-self`
→ `descendant`
→ `following`
→ `following-sibling`

In doc order

Backward Axes:

→ `parent`
→ `ancestor`
→ `ancestor-or-self`
→ `preceding`
→ `preceding-sibling`

reverse doc order

→ `attribute`

Location Path Evaluation

Context of an XPath evaluation:

(1) context-node
(2) context position and size (both non-negative integers)
(3) set of variable bindings (= mappings from variable names to values)
(4) function library (= mapping from function names to functions)
(5) set of namespace declarations

(btw: context position is $\leq$ context size)

Application determines the Initial Context.

If path starts with “/”, then Initial Context has

→ context-node = root node
→ context-position = context-size = 1

Location Path Semantics

→ A Location Path **P** is a sequence of Location Steps

a_1 :: n_1 [F_1_1] [F_1_2] … [F_1_n1]
/ **a_2** :: n_2 [F_2_1] [F_2_2] … [F_2_n2]

/ **a_m** :: n_m [F_m_1] [F_m_2] … [F_m_nm]

S0 = initial sequence of context-nodes

(1) (to each) context-node N in S0, apply axis **a_1**: gives sequence S1 of nodes
(2) remove from S1 any node M for which

- → test n_1 evaluates to false
- → any of filters F_1_1,…,F_1_n1 evaluate to false.

Apply steps (1)&(2) for step 2, to botain from S1 the sequence S2

3,	S2	S3
…	…	…
m	S{m-1}	Sm

= result of **P**

No Looking Back

Backward Axes are not needed!!

→ possible to rewrite most XPath queries into equivalent ones that **do not use backward axes**.

Very nice result!

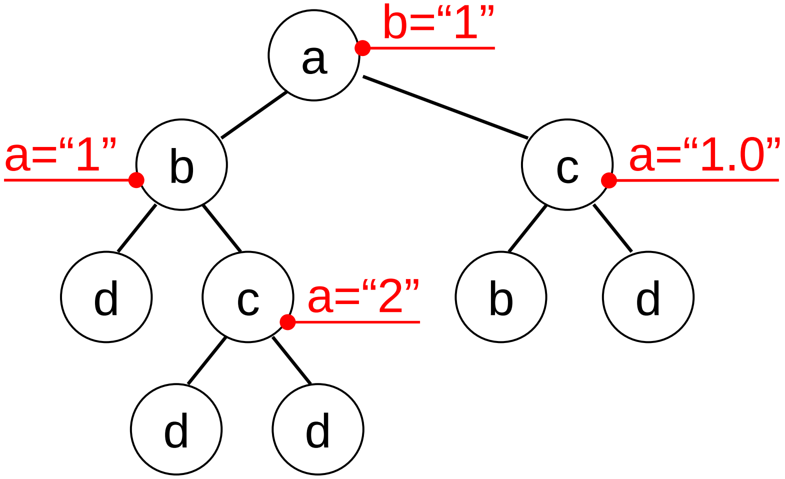
Can you see how this could be done?

→ We saw an example of removing ancestor axis. But, of course the rewritten query must be the same ON EVERY possible tree!!

Questions *how much larger* does the query get, when you remove all backward axis?
Is this *useful* for efficient query evaluation?!

Attribute Axis

How to
→ test attribute nodes



Examples

`//attribute::*`

Result:
b="1"
a="1"
a="2"
a="1.0"

Remember, these are just NODEs.

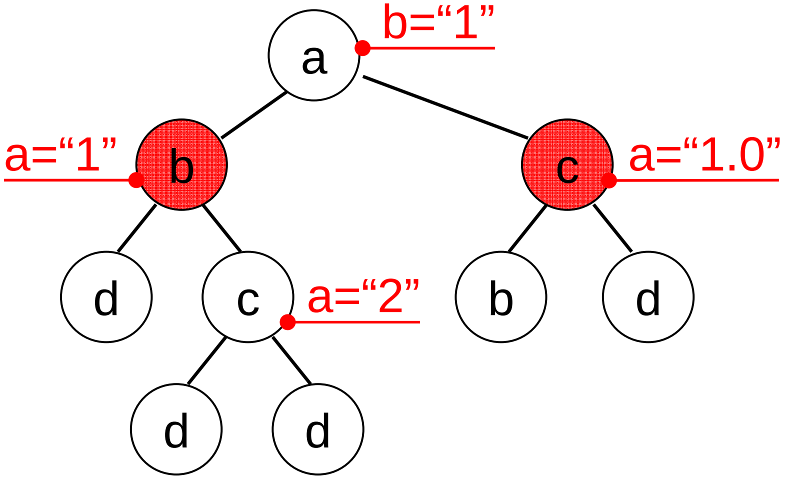
//attribute::*/.   gives same result

And //attribute::a/.. gives

<b a="1"><d/><c a="2"><d/><d/></c></b>
<c a="2"><d/><d/></c>
<c a="1.0"><b/><d/></c>

Attribute Axis & Value Tests

How to
→ test attribute **values**



Examples

`//*[attribute::a=1]`

(selects the two red nodes)

Attribute Axis & Value Tests

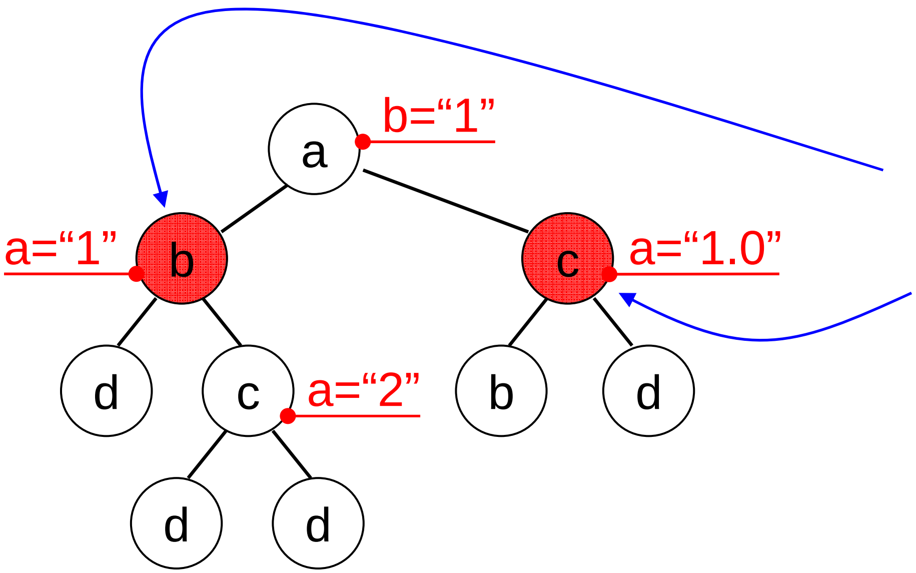
How to
→ test attribute **values**

Examples

number (float) comparison

`//*[attribute::a=1]`

(selects the two red nodes)



Watch out

//*[attribute::a=“1”] only gives

//*[attribute::a=“1.0”] only gives

string comparison

Attribute Axis & Value Tests

How to
→ test attribute **values**

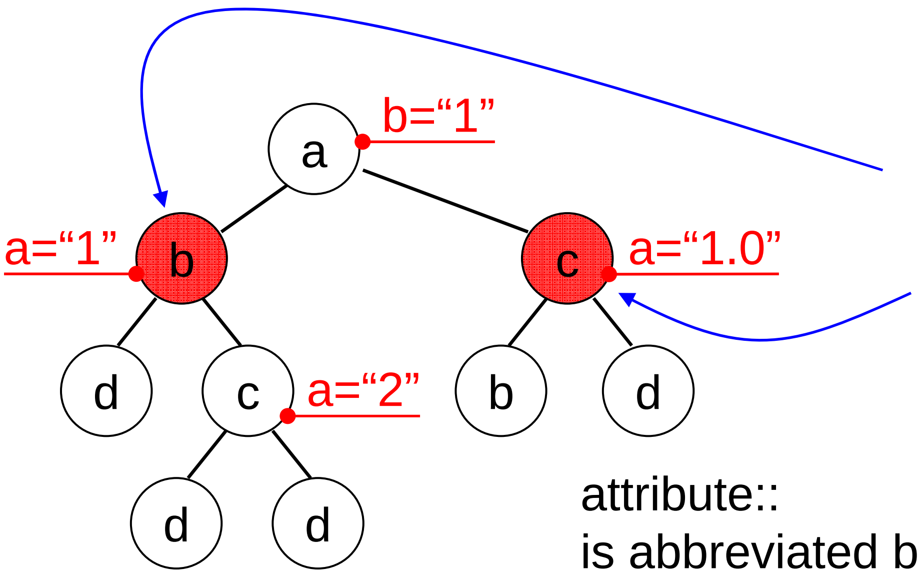
Examples

number (float) comparison

`//*[attribute::a=1]`

(selects the two red nodes)

Watch out



//*[attribute::`a="1"`] only gives

//*[~~attribute::~~`a="1.0"`] only gives

@

string comparison

attribute::
is abbreviated by @

Attribute Axis & Value Tests

How to
→ test attribute **values**

Examples

number (float) comparison

`//*[attribute::a=1]`

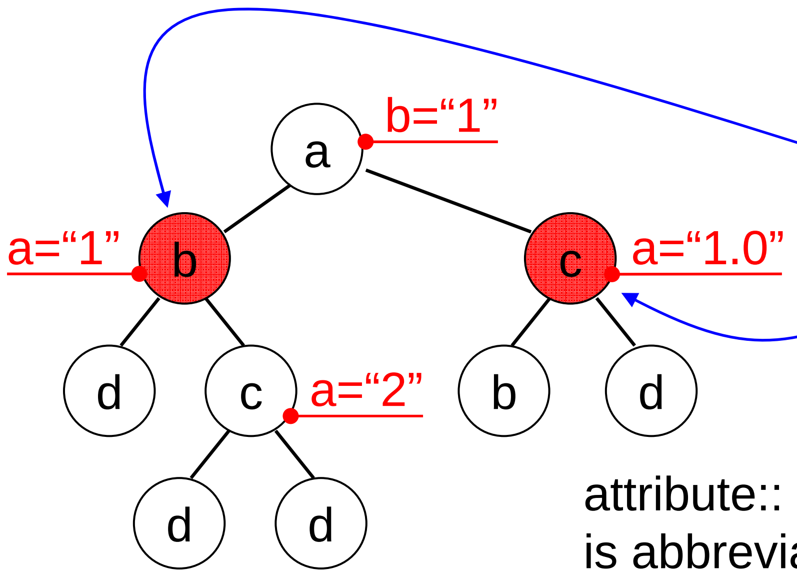
(selects the two red nodes)

Watch out

//*[attribute::a="1"] only gives

//*[~~attribute::~~a="1.0"] only gives
@

string comparison



attribute::
is abbreviated by @

//*[@a!="1"] selects both c-nodes
//*[@a>1] selects only left c-node
//*[@a=//@b] selects what?? (hint: "=" is string comp. here)

Tests in Filters

- or
- and
- =, !=
- <=, <, >=, >

Boolean **true**
coerced to a float 1.0

The operators are all left associative.
For example, 3 > 2 > 1 is equivalent to (3 > 2) > 1, which evaluates to **false**.

But, 3 > 2 > 0.9 evaluates to **true**. Can you see why?

For two strings u,v

u<=v
u<v
u>=v
u>v

Always return **false**!
→ Unless both u and v are numbers.

["1.0">="1"] evaluates to **true**.

Text Nodes

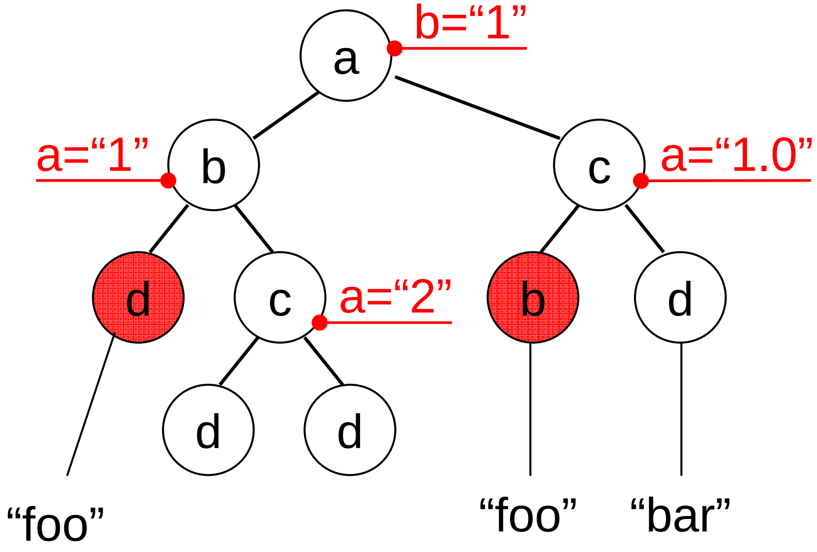
How
→ test text nodes & values

//text()

Result:
foo
foo
Bar

//*[text()=“foo”]

Result: the two red nodes



Question:

What is the result for
//*[text()=//b/text()]

Useful Functions (on Booleans)

→ `boolean(object):boolean` (“boolean” means {**true/false**})

Converts argument into **true/false**:

a number is **true** if it is not equal to zero (or NaN)
a node-set is **true** if it is non-empty
a string is **true** if its length is non-zero
- for other objects, conversion depends on type

→ `not(true)=false,  not(false)=true`
→ `true():boolean`
→ `false():boolean`

→ `lang(string):boolean`
Returns **true** if language specified by `xml:lang` attributes is same as string

Useful even for use with self-axis:

```
child::*[self::chapter or self::appendix]
```

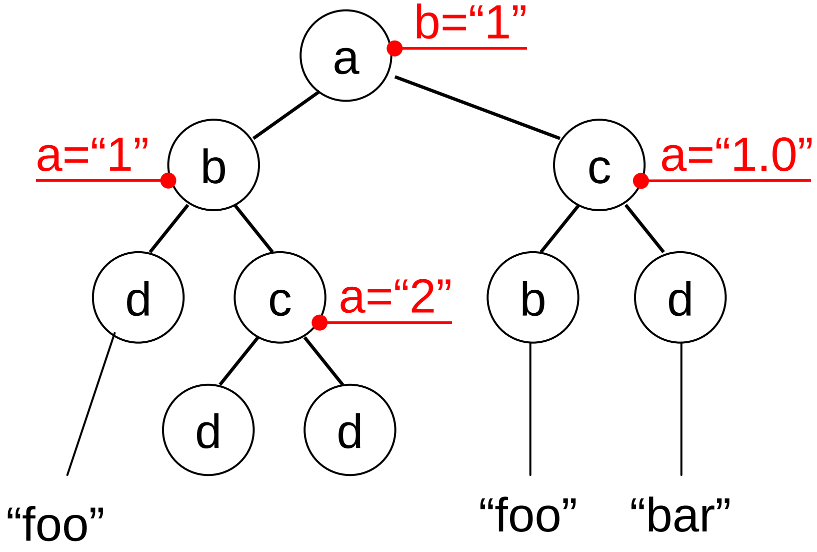
chapter or appendix children of context node

Useful Functions (on Node Sets)

→ `count`
Counts number or results

```
/a[count(//*[text()=//b/text()])=2]
```

What is the result?



Useful Functions (on Node Sets)

→ `count`

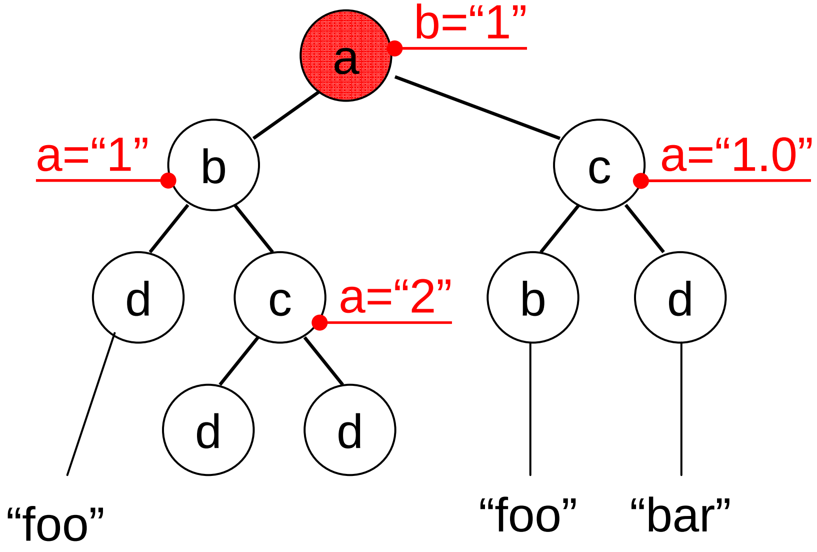
Counts number or results

```
/a[count(//*[text()=//b/text()])=2]
```

What is the result?

Same result as:

```
/a[count(//*[text()="foo"])
  > count(//*[text()="bar"])]
```



Useful Functions (on Node Sets)

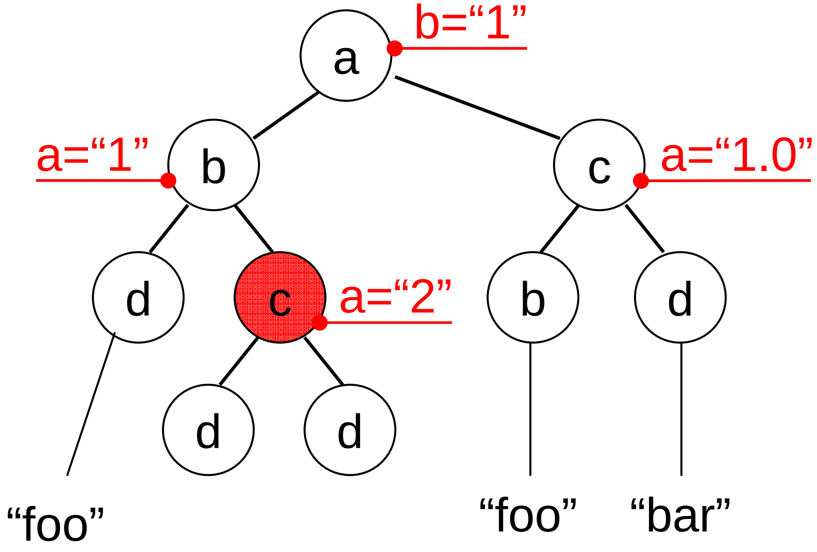
→ `count`
Counts number or results

```
/a[count(//*[text()=//b/text()])=2]
```

What is the result?

Same result as:

```
/a[count(//*[text()="foo"])
  > count(//*[text()="bar"])]
```



What is the result for:

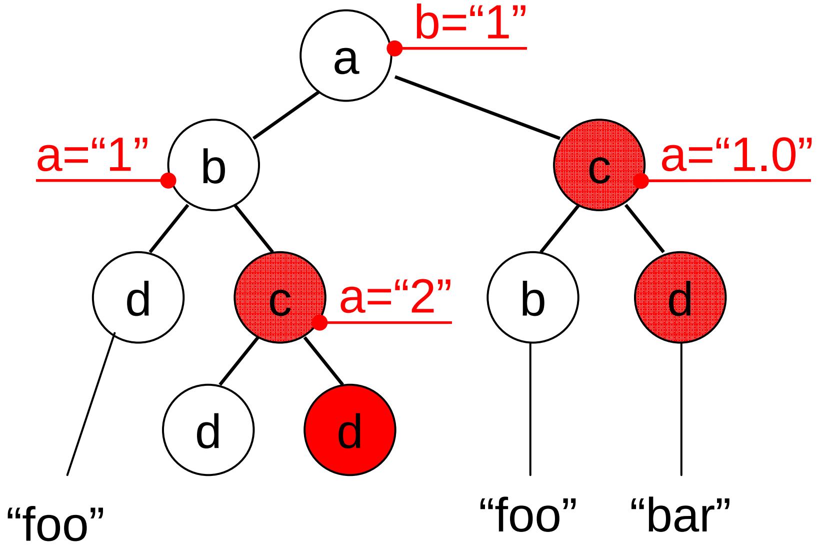
```
//c[count(b)=0]
```

(same as `//c[not(b)]`)

Useful Functions (on Node Sets)

→ `last()`
returns contex-size from the evaluation context

→ `position()`
Returns context-position from the eval. context

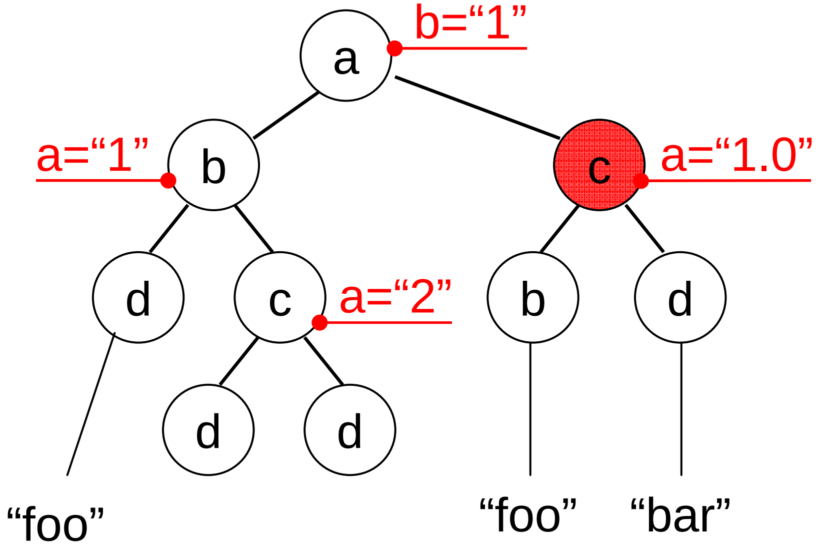


`//*[position()=2]`

Useful Functions (on Node Sets)

→ `last()`
returns contex-size from the evaluation context

→ `position()`
Returns context-position from the eval. context



```
//*[position()=2]
```

```
//*[position()=2 and ../../a]
```

Same as

```
//*[position()=2 and ./b]
```

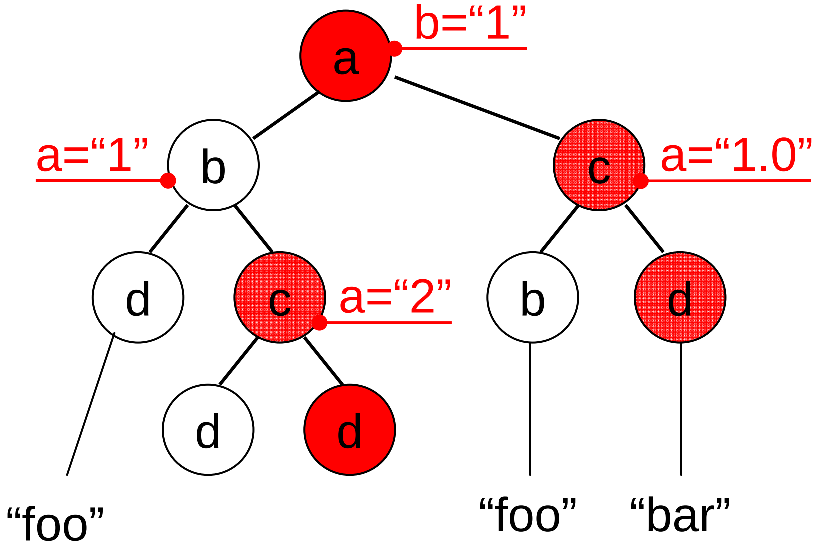
Useful Functions (on Node Sets)

→ `last()`

returns contex-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



```
//*[position()=2]

//*[position()=2 and ../../a]
Same as
//*[position()=2 and ./b]


//*[position()=last()]
```

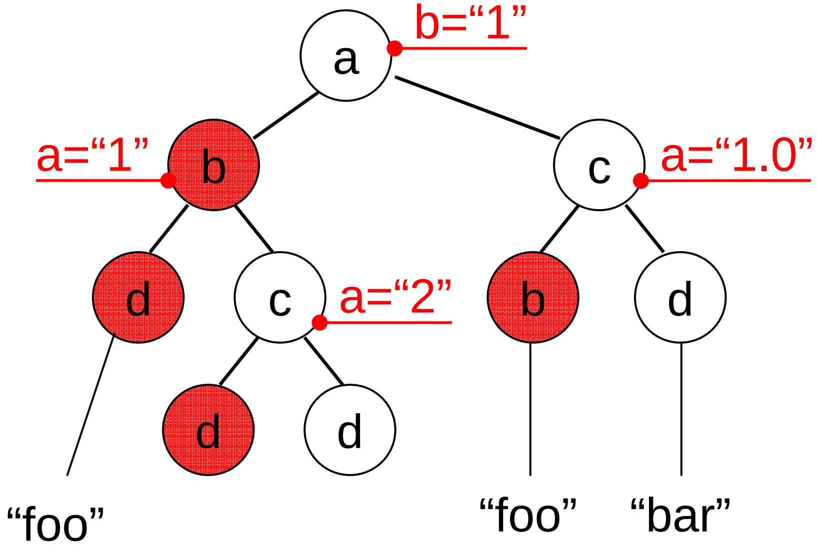
Useful Functions (on Node Sets)

→ `last()`

returns contex-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



```
//*[position()=2]
```

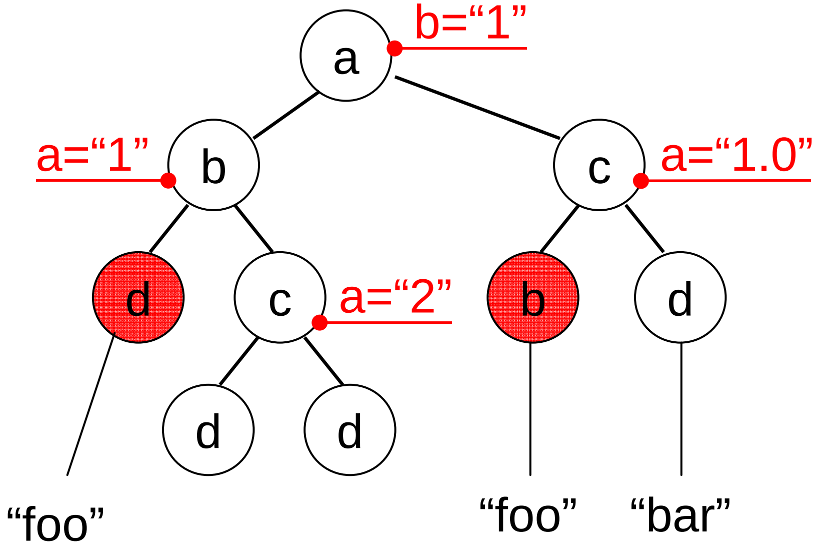
```
//*[position()=2 and ../../a]
Same as
//*[position()=2 and ./b]
```

```
//*[position()=last()-1]
```

Useful Functions (on Node Sets)

→ `last()`
returns contex-size from the evaluation context

→ `position()`
Returns context-position from the eval. context

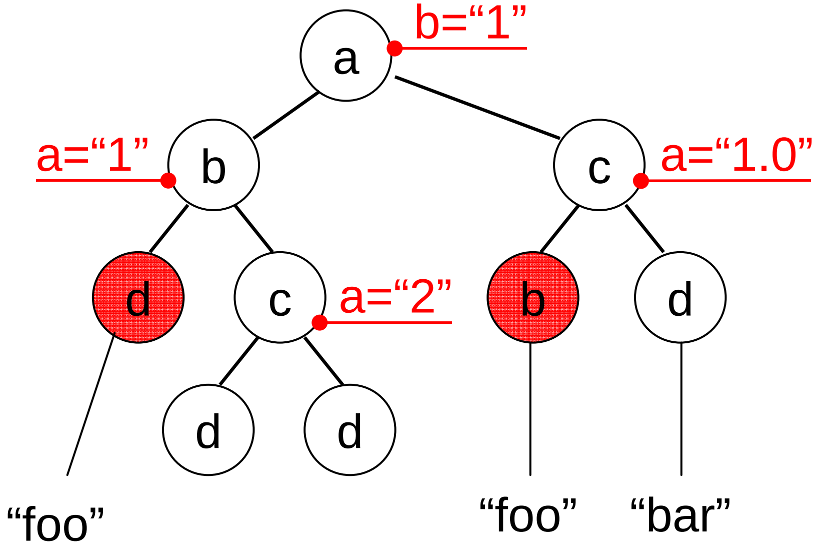


```
//*[position()=2]

//*[position()=2 and ../../a]
Same as
//*[position()=2 and ./b]

//*[position()=last()-1
     and ./text()="foo"]
```

Useful Functions (on Node Sets)

→ `last()`
returns contex-size from the evaluation context

→ `position()`
Returns context-position from the eval. context



```
//*[position()=2]

//*[position()=2 and ../../a]
Same as
//*[position()=2 and ./b]


//*[position()=last()-1
     and ./text()="foo"]
```

Useful:

`child::*[self::chapter or self::appendix][position()=last()]`

selects the last chapter or appendix child of the context node

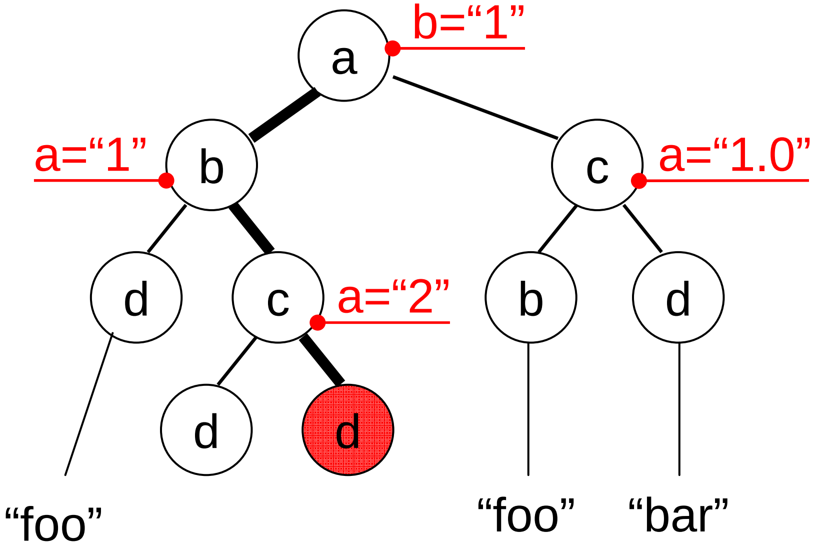
Useful Functions (on Node Sets)

→ `last()`

returns contex-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



```
//*[position()=2]

//*[position()=2 and ../../a]
Same as
//*[position()=2 and ./b]


//*[position()=last()-1
     and ./text()="foo"]
```

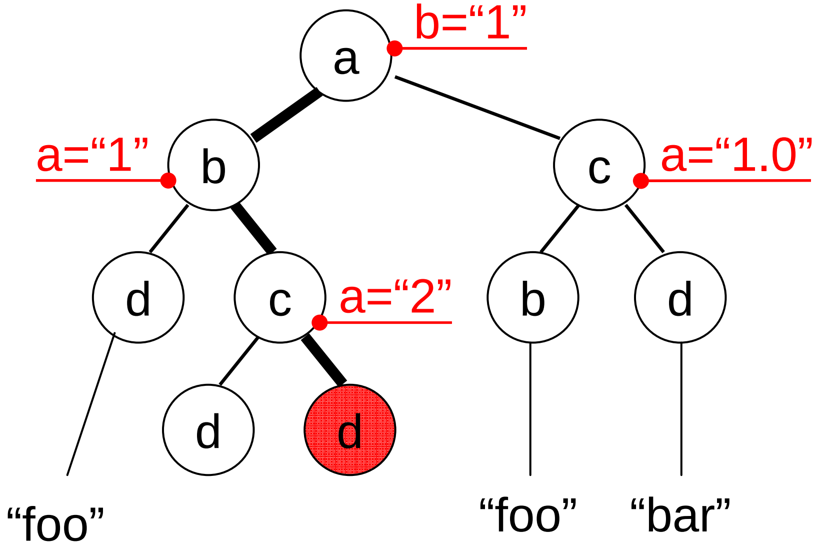
```
*/*[position()=1]/*[position()=2]/*[position()=2]
```

→ allows absolute location of any node (a la Dewey)

Useful Functions (on Node Sets)

→ `last()`
returns contex-size from the evaluation context

→ `position()`
Returns context-position from the eval. context



```
//*[position()=2]

//*[position()=2 and ../../a]
Same as
//*[position()=2 and ./b]


//*[position()=last()-1
    and ./text()="foo"]
```

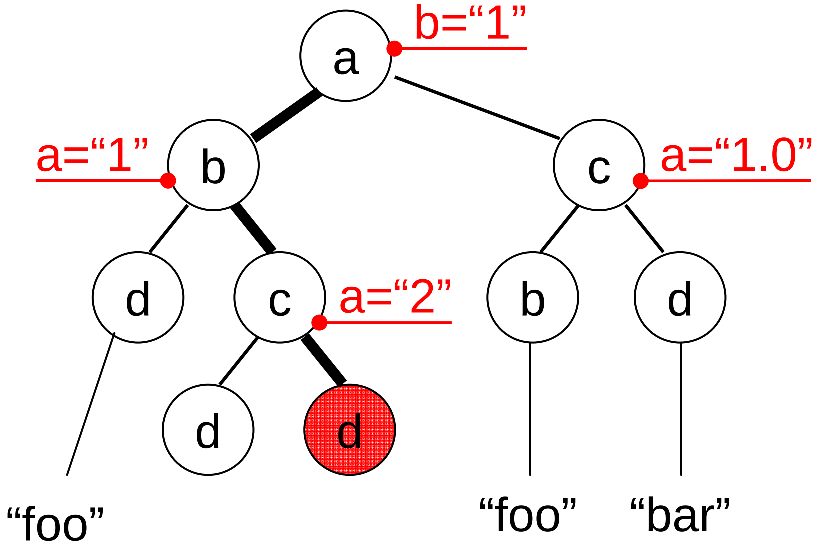
```
*/*[position()=1]/*[position()=2]/*[position()=2]
```

Abbreviation: `*/*[1]/*[2]/*[2]`

Useful Functions (on Node Sets)

→ `last()`
returns contex-size from the evaluation context

→ `position()`
Returns context-position from the eval. context



```
//*[position()=2]

//*[position()=2 and ../../a]
Same as
//*[position()=2 and ./b]


//*[position()=last()-1
    and ./text()="foo"]
```

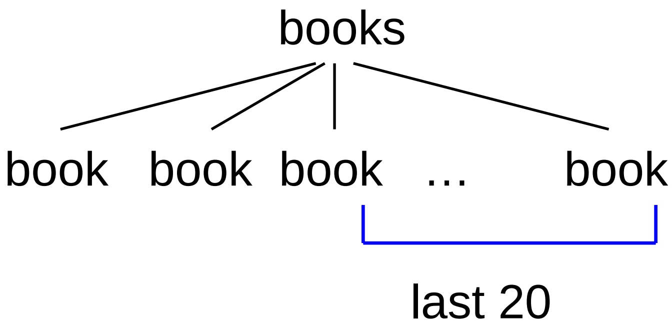
```
*/*[position()=1]/*[position()=2]/*[position()=2]
```

Abbreviation: `*/*[1]/*[2]/*[2]` **→What is result for** `//*[./*[2]/*[2]]`

Useful Functions (on Node Sets)

→ `last()`
returns contex-size from the evaluation context

→ `position()`
Returns context-position from the eval. context



How do you select the
last 20 book-children of books?

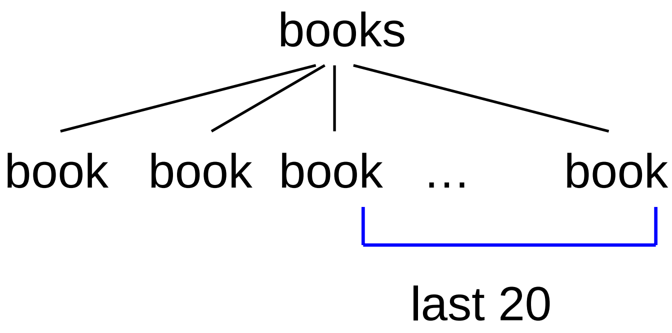
Useful Functions (on Node Sets)

→ `last()`

returns contex-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



How do you select the
last 20 book-children of books?

```
/books/book[position()>last()-20]
```

Useful Functions (on Node Sets)

→ `last():number`
returns contex-size from the evaluation context

→ `position():number`
eturns context-position from the eval. Context

→ `id(object):node-set`
`id(“foo”)` selects the element with unique ID foo

→ `local-name(node-set?):string`
returns the local part of the expanded-name of the node

→ `namespace-uri(node-set?):string`
returns the namespace URI of the expanded-name of the node

→ `name(node-set?):string`
returns a string containing a QName representing the expanded-name of the node

Useful Functions (on Node Sets)

XPath 2.0 has much clearer comparison operators!!

Nodes have an identity

```
<a>
  <b>tt</b>
  <b>tt</b>
</a>
```

Different nodes!

```
//a[*[1]=*[2]]
```

~~gives *empty result*.~~

Sorry.
This is **wrong.**
Equality ("=") is based on string value of a node!

→ Gives also a-node

But:

```
//a[contains(*[1],*[2])]
```

gives the a-node.

string-value ("tt") **is** contained in "tt"

Useful Functions (on Node Sets)

XPath 2.0 has much clearer comparison operators!!

Careful with equality ("=")

```
<a>
  <b>
    <d>red</d>
    <d>green</d>
    <d>blue</d>
  </b>
  <c>
    <d>yellow</d>
    <d>orange</d>
    <d>green</d>
  </c>
</a>
```

Sorry.
This is **wrong.**
Equality ("=") is based on string value of a node!

→ Gives also a-node

`//a[b/d = c/d]` selects a-node!!!

there exists a node in the node set for `b/d`
with same string value as a node in node set `c/d`

Useful Functions (on Node Sets)

XPath 2.0 has much clearer comparison operators!!

Careful with equality (“=“)

```
<a>
  <b>
    <d>red</d>
    <d>green</d>
    <d>blue</d>
  </b>
  <c>
    <d>yellow</d>
    <d>orange</d>
    <d>green</d>
  </c>
</a>
```

Sorry.
This is **wrong.**
Equality (“=“) is based on string value of a node!

→ Gives also a-node

`//a[b/d = c/d]` selects a-node!!!

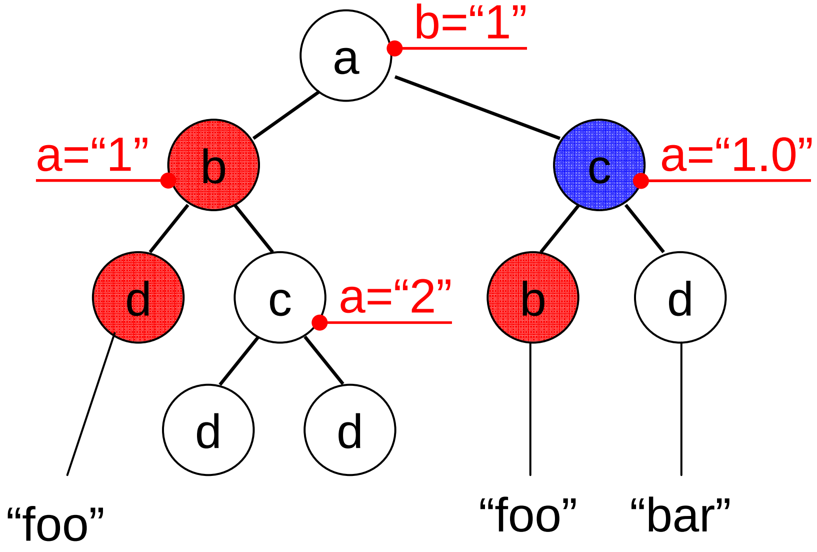
there exists a node in the node set for `b/d`
with same string value as a node in node set `c/d`

→ `What about //a[b/d != c/d]`

Useful Functions (Strings)

The string-value of an element node is the concatenation of the string-values of all text node descendants in document order.

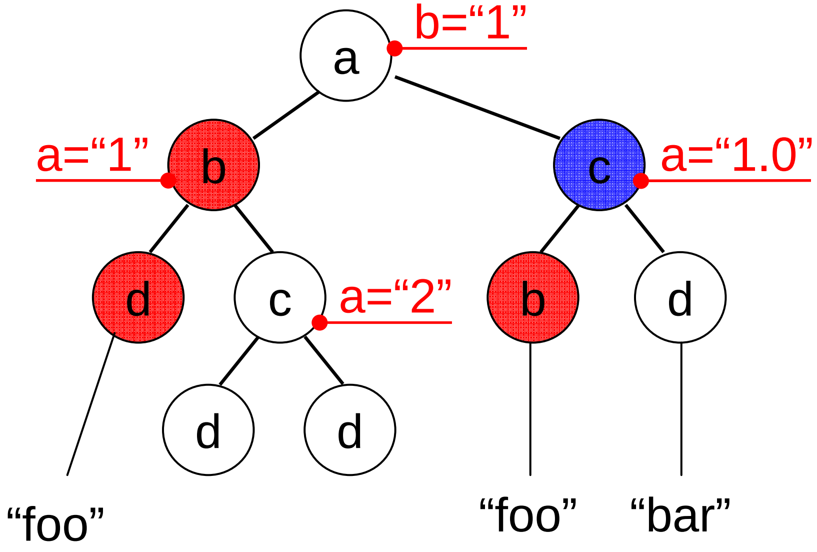
```
//*[.="foo"]
//*[.=“foobar”]
```



Useful Functions (Strings)

The string-value of an element node is the concatenation of the string-values of all text node descendants in document order.

```
//*[.="foo"]
//*[.=“foobar”]
```



- → concat(st_1, st_2,…, st_n) = st_1 st_2 … st_n
- → startswith(“abcd”,”ab”) = true
- → contains(“bar”,”a”) = true
- → substring-before("1999/04/01","/") = 1999.
- → substring-after("1999/04/01","19") = 99/04/01
- → substring("12345",2,3) = “234”
- → string-length(“foo”) = 3

What is the result to this: `//*[contains(.,"bar")]`

Useful Functions (Strings)

The string-value of an element node is the concatenation of the string-values of all text node descendants in document order.

```
//*[.="foo"]
//*[.=“foobar”]
```

→ normalize-space(“   foo   bar a “) = “foo bar a”

→ translate(“bar","abc","ABC") = BAr

returns the first argument string with occurrences of characters in the second argument string replaced by the character at the corresponding position in the third argument string

NOTE: The **translate** function is not a sufficient solution for case conversion in all languages

Useful Functions (Numbers)

Operators on Numbers
`+,-,*,div,mod`

→ `number(object):number`

Converts argument to a number

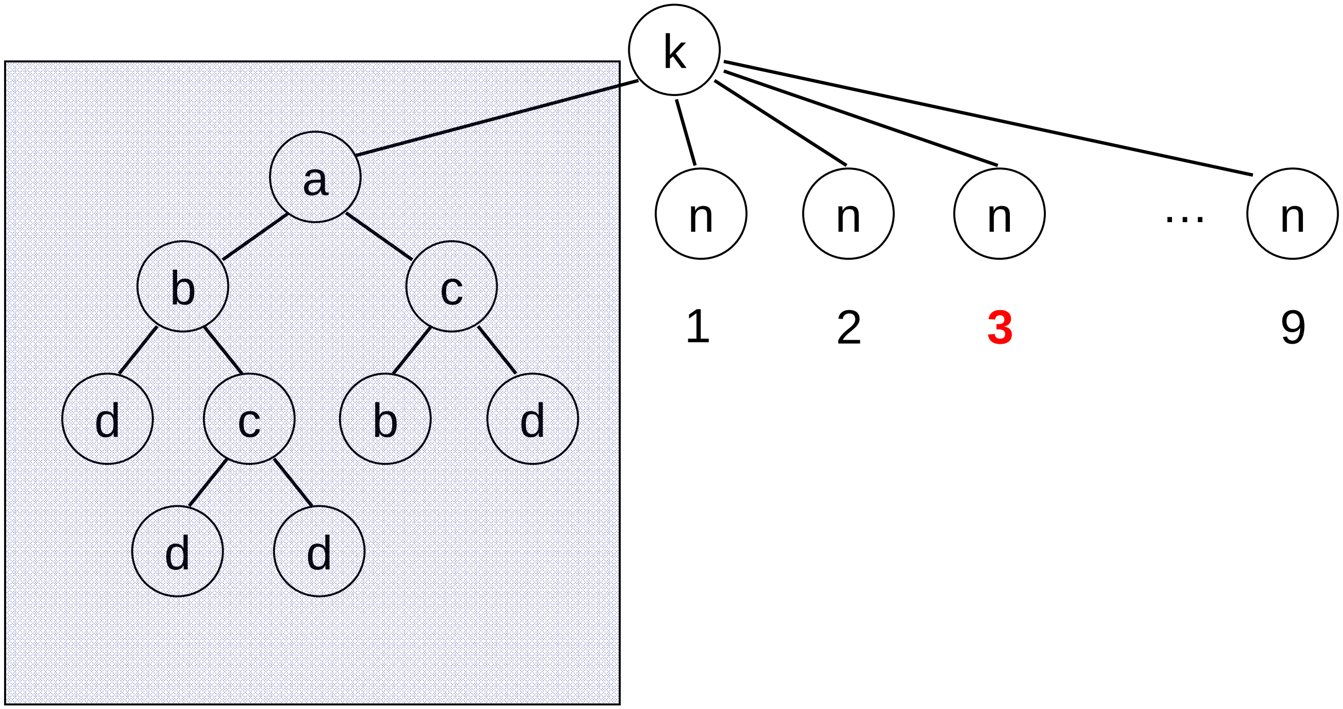
- the boolean true is converted to 1, false is converted to 0
- a string that consists of optional whitespace followed by an optional minus sign followed by a Number followed by whitespace is converted to the IEEE 754 number that is nearest to the mathematical value represented by the string.

→ `sum(node-set):number`
returns sum, for each node in the argument node-set, of the result of converting the string-values of the node to a number

→ `floor(number):number`
returns largest integer that is not greater than the argument
→ `ceiling(number):number`
returns the smallest integer that is not less than the argument
→ `round(number):number`
returns integer closest to the argument. (if there are 2, take above:
`round`(0.5)=1 and `round`(-0.5)=0

Display Number Result…

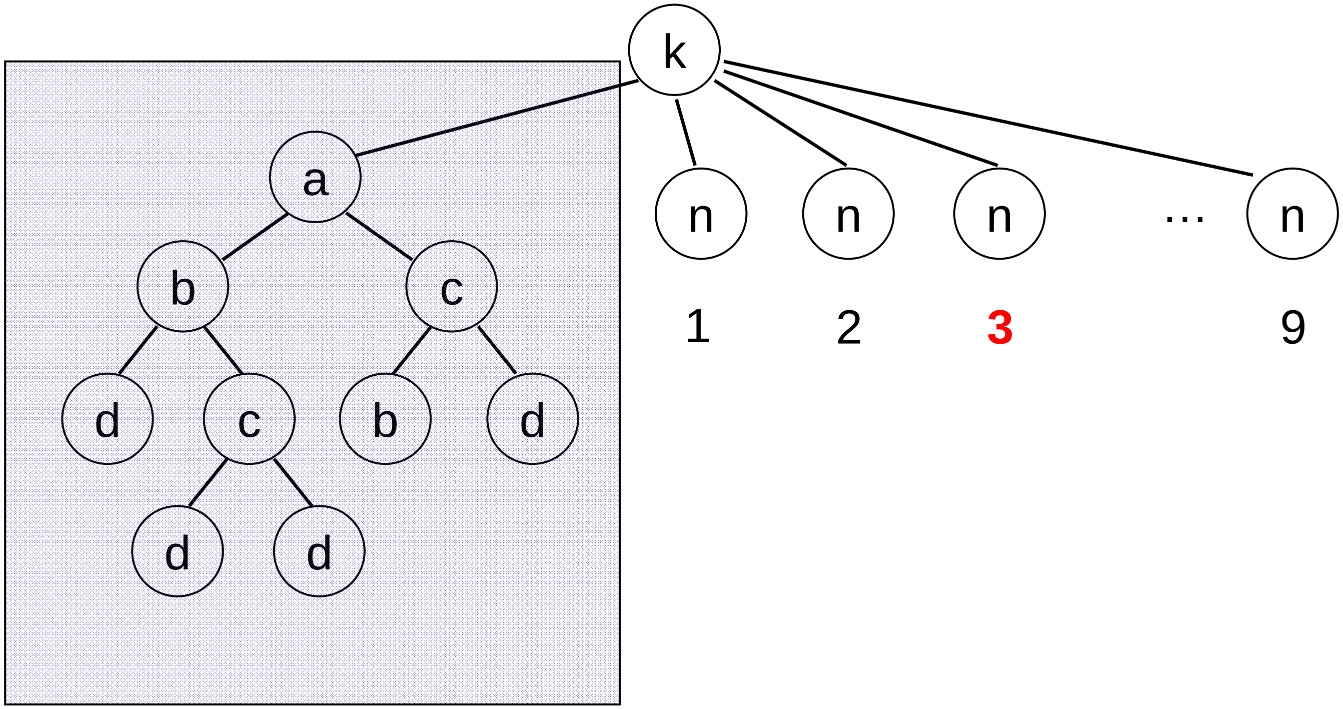
```
//*[text()=7 mod (count(//b)+2))]/text()
```



Use http://b-cage.net/code/web/xpath-evaluator.html

Display Number Result…

```
//*[text()=7 mod (count(//b)+2))]/text()
```



Similar for arbitrary large numbers / booleans, node-sets… Try it… ☺

XPath Query Evaluation

How to implement?

How expensive? complexity?

What are the most difficult queries?

Next time

Efficient Algorithms: *which queries run how fast*?

First, focus on *navigational queries*: only /, //, label-test, [filters]

(techniques for
value comparison/queries already well-known from rel. DB's…)

means year **2003**…

Experiments with current systems

Next 4 slides from
Georg Gottlob and Christoph Koch "XPath Query Processing".
Invited tutorial at DBPL **2003**
http://www.dbai.tuwien.ac.at/research/xmltaskforce/xpath-tutorial1.ppt.gz

context node

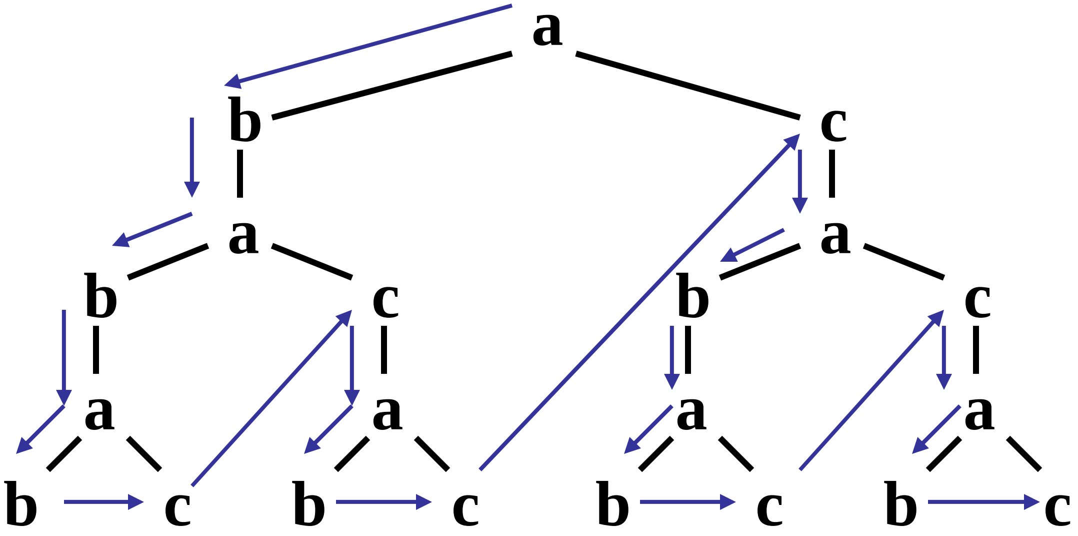
$$P[\![\pi_1/\pi_2]\!](x) := \bigcup_{y \in P[\![\pi_1]\!](x)} P[\![\pi_2]\!](y)$$

```
procedure process-location-step($n_0$, $Q$)
/* $n_0$ is the context node;
   query $Q$ is a list of location steps */
begin
   node set $S$ := apply $Q$.first to node $n_0$;
   if ($Q$.tail is not empty) then
      for each node $n \in S$ do
         process-location-step($n$, $Q$.tail);
end
```

Document:

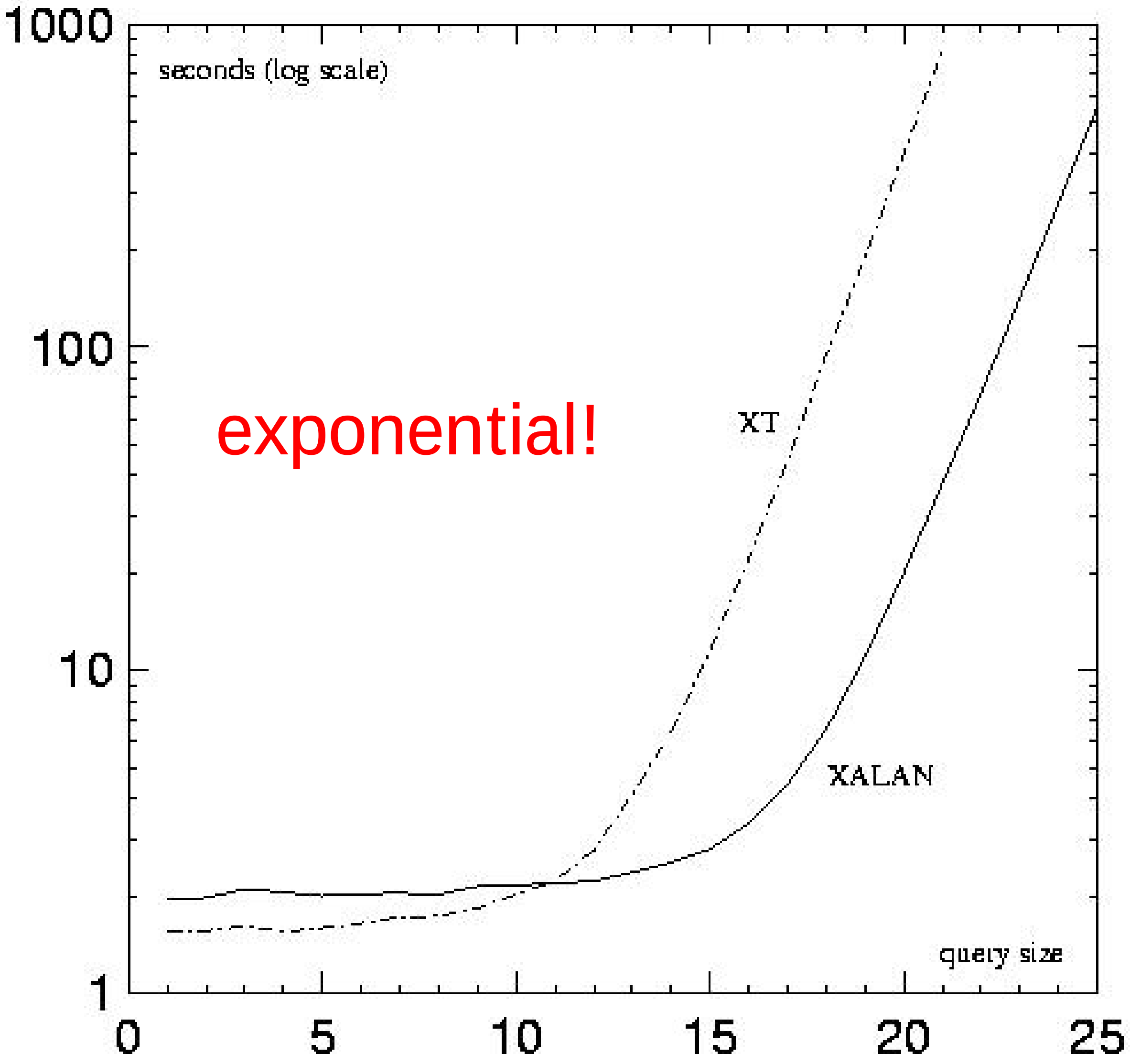
<a> <b/> <c/> </a>

Xpath Query (relative to a):
child::*/parent::*/child::*/
parent::*/child::*



Tree of nodes visited is of size $O(|D|^{|Q|})$!!!

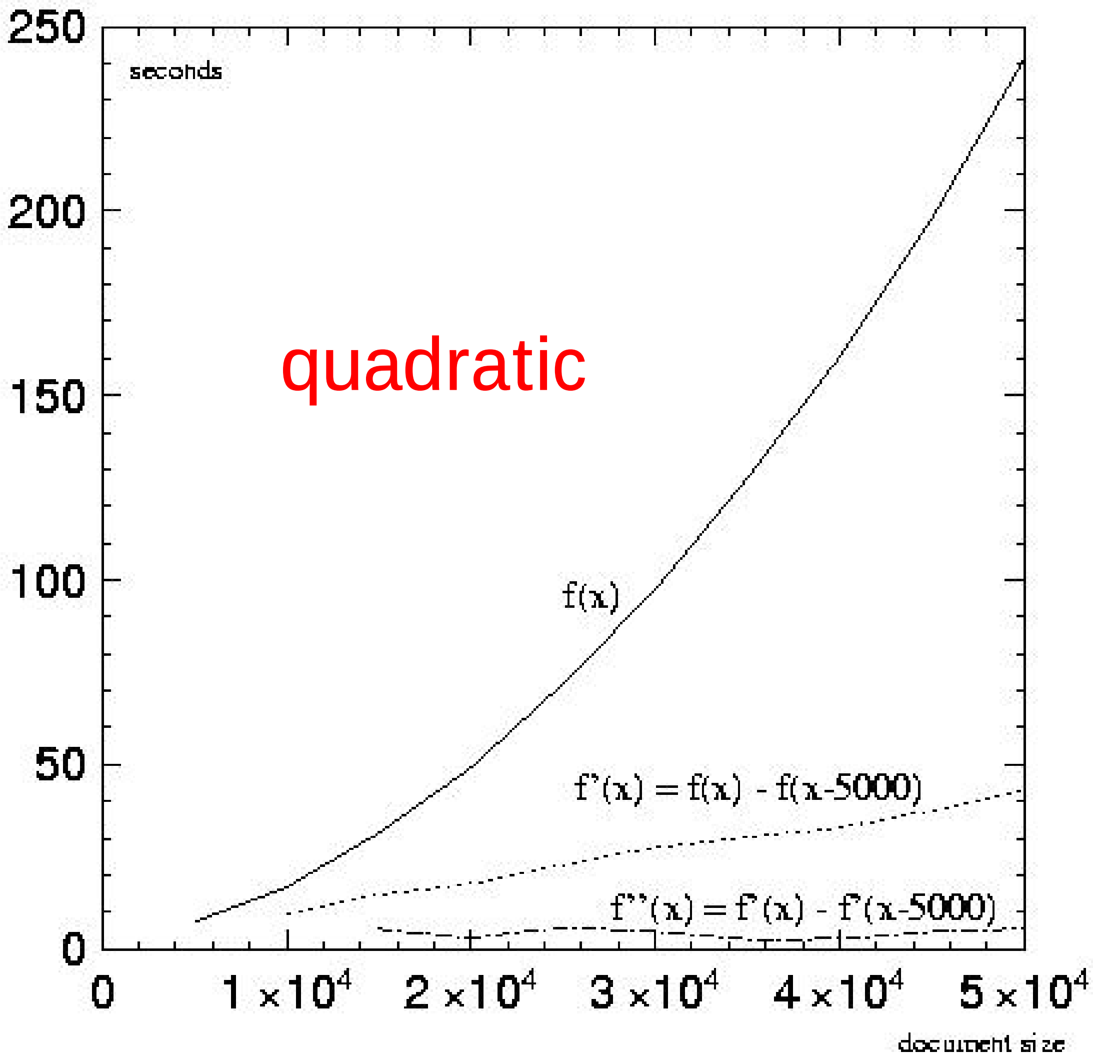
Document:
<a><b/><b/></a>



exponential!

Core Xpath on Xalan and XT

Queries: a/b/parent::a/b/...parent::a/b



quadratic

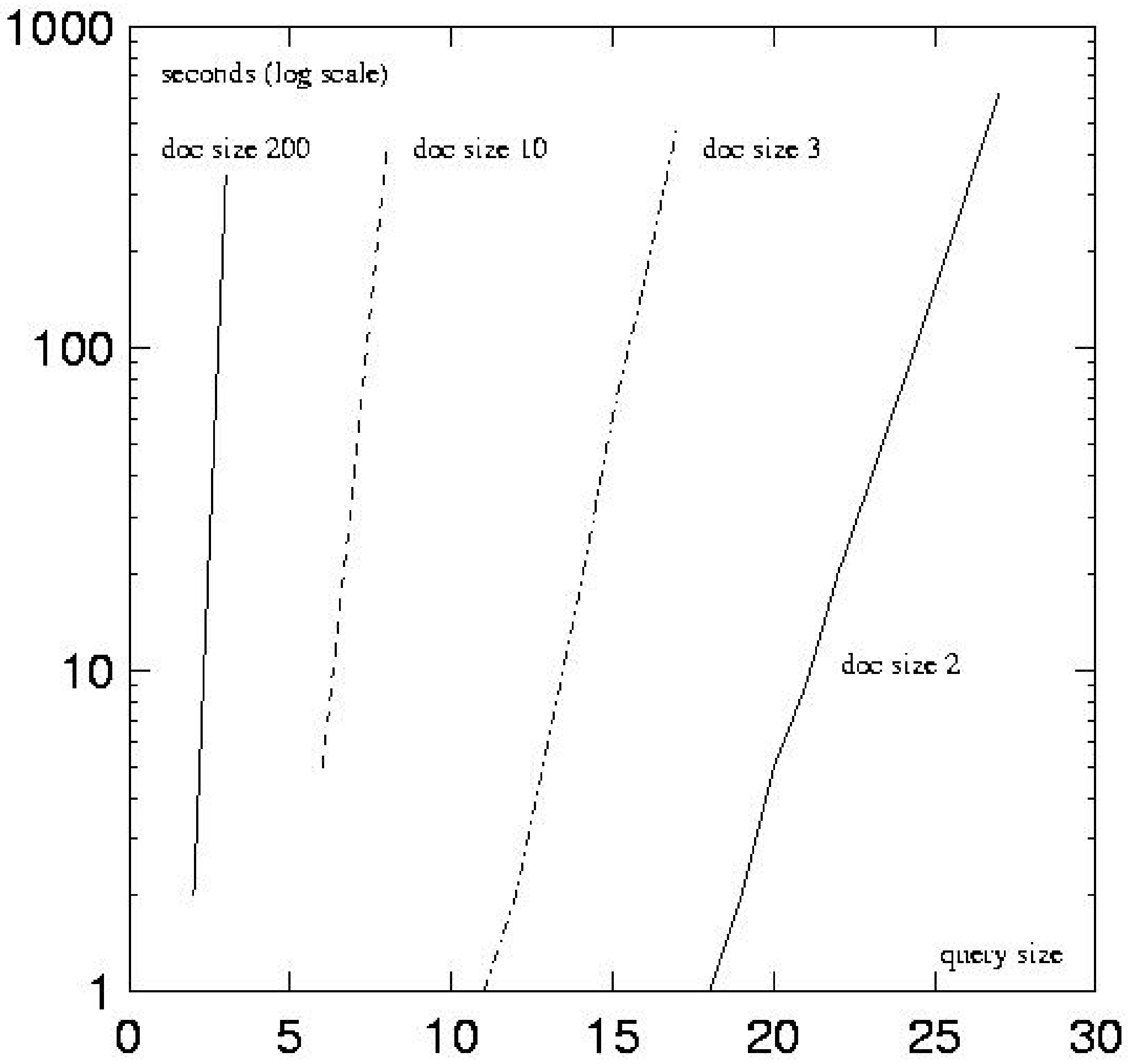
Core Xpath on Microsoft IE6:

polynomial combined complexity,
quadratic data complexity

Full XPath on IE6:

Exponential combined complexity!

Exponential query complexity



XPath Query Evaluation

Static Methods (used, e.g., for Query Optimization…)

Given Xpath queries Q1, Q2:

→ Is result set of Q1 included in result set of Q2?

→ Are result sets equal?

→ Is their intersection empty?

for all possible documents

(probably we will look at this in Lecture 8 or 9)

Simple Examples

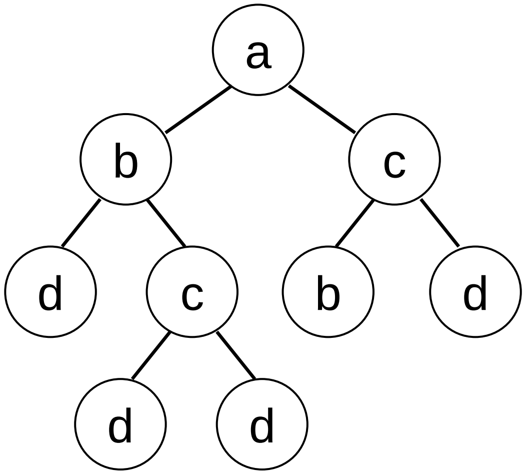
Is

```
//c[count(d)=count(*)]
```

equivalent to

```
//c[not(child::*[not(self::d)])]
```

on all possible trees?



END
Lecture 6