# XML and Databases

**Lecture 13**
*Update Languages for XML*

Sebastian Maneth
NICTA and UNSW

*CSE@UNSW -- Semester 1, 2010*

# Outline

1. Update Languages for XML

   → XQuery Update Facility: delete,insert,replace,rename,remove
   → type issues
   → snapshot semantics

2. The physical site

   → how to update a DAG?
   → how to update PRE/POST encoding?
   → other storage schemes?

# XML Updates -- History

Updates = write operations, e.g., *delete, insert, replace, rename*, etc

Want to have Update Language, i.e., a formalism for "update programs".

---

Currently, there is **no** accepted standard XML Update Language

→ XUpdate (XML:DB, working draft from 9/2000)

→ XQuery! (by the implementors of the Galax XQuery engine)

→ XQuery Update Facility (W3C Candidate Recommendation, 09 June 2009)

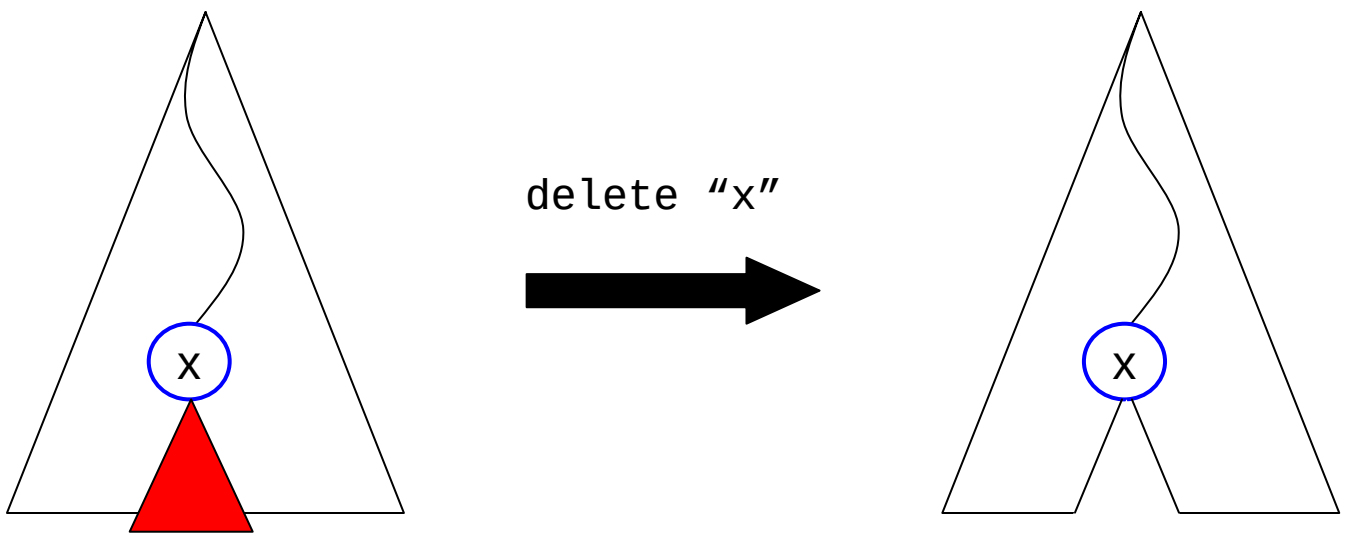
plus lots of other smaller projects…

# XML Updates

**Note**
Every node has an "identity" = a unique identifier.
Also: there may be attributes of type "ID"!

**Example** updates for XML data

(1) *delete subtree* rooted at node x



# XML Updates

**Example** updates for XML data

(1) *delete subtree* rooted at node x

Use XPath to specify the nodes x to be deleted.

Explicit examples

Delete the last author of the first book in a given bibliography.

```
do delete fn:doc("bib.xml")/books/book[1]/author[last()]
```

Delete all email messages that are more than 365 days old.

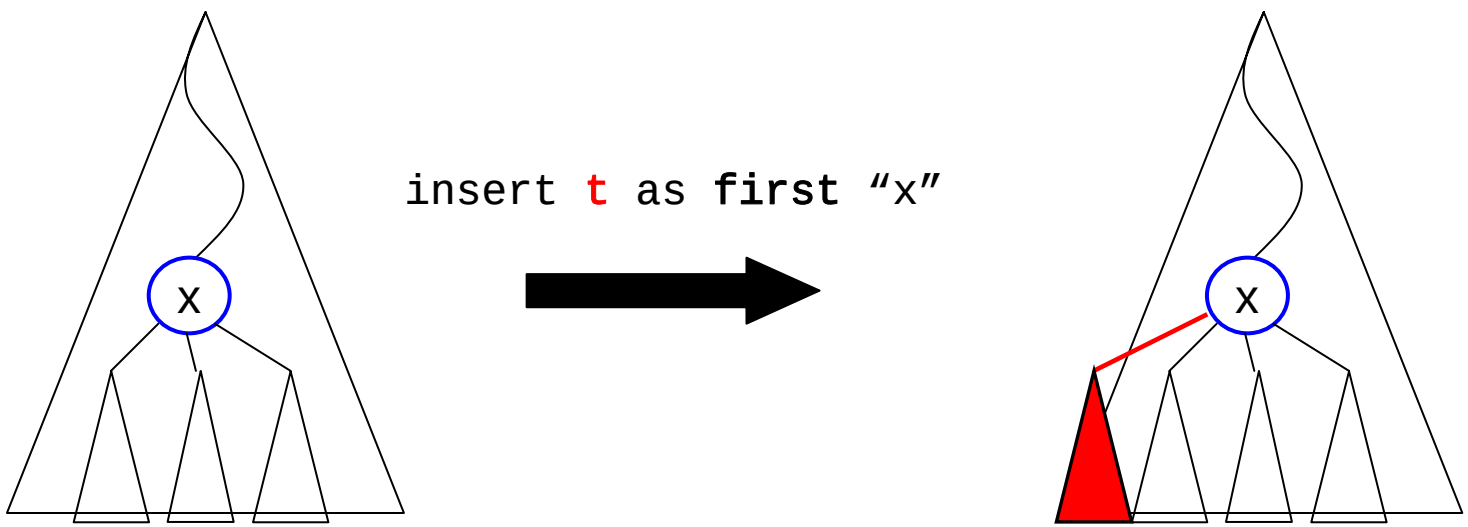
```
do delete /email/message[fn:currentDate()-date >
                                    xs:dayTimeDuration("P365D")]
```

# XML Updates

**Note**
Every node has an "identity" = a unique identifier.
Also: there may be attributes of type "ID"!

**Example** updates for XML data

(2) *insert subtree "t"* as **first** of node x

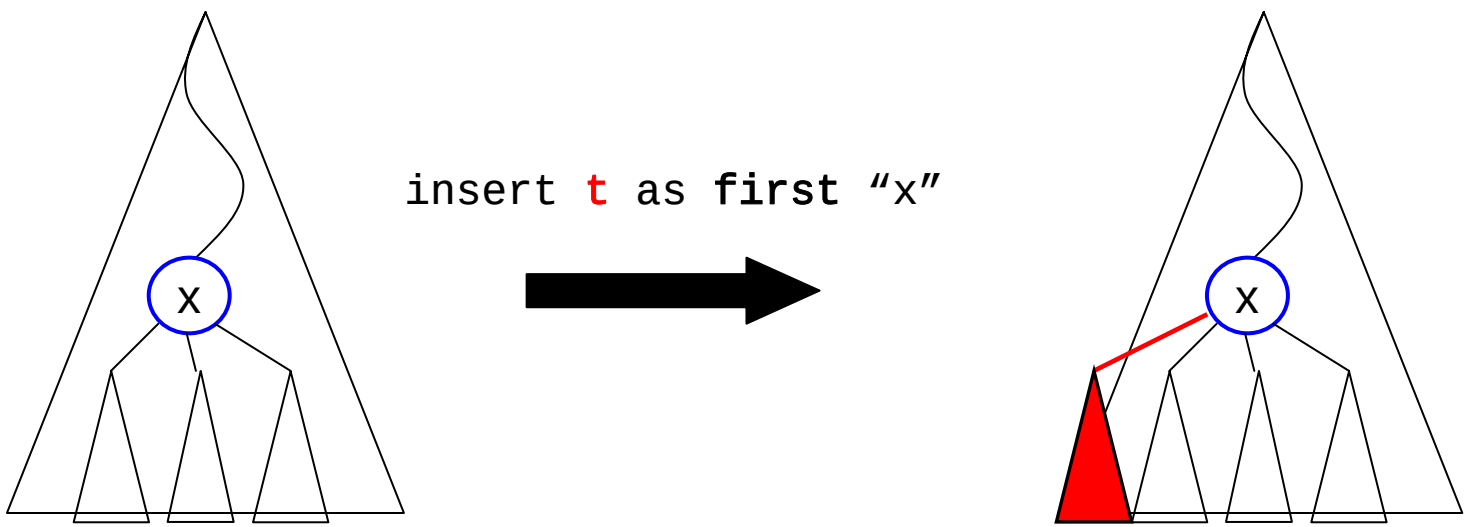


# XML Updates

**Note**
Every node has an "identity" = a unique identifier.
Also: there may be attributes of type "ID"!

**Example** updates for XML data

(2) *insert subtree "t"* as **first** of node x



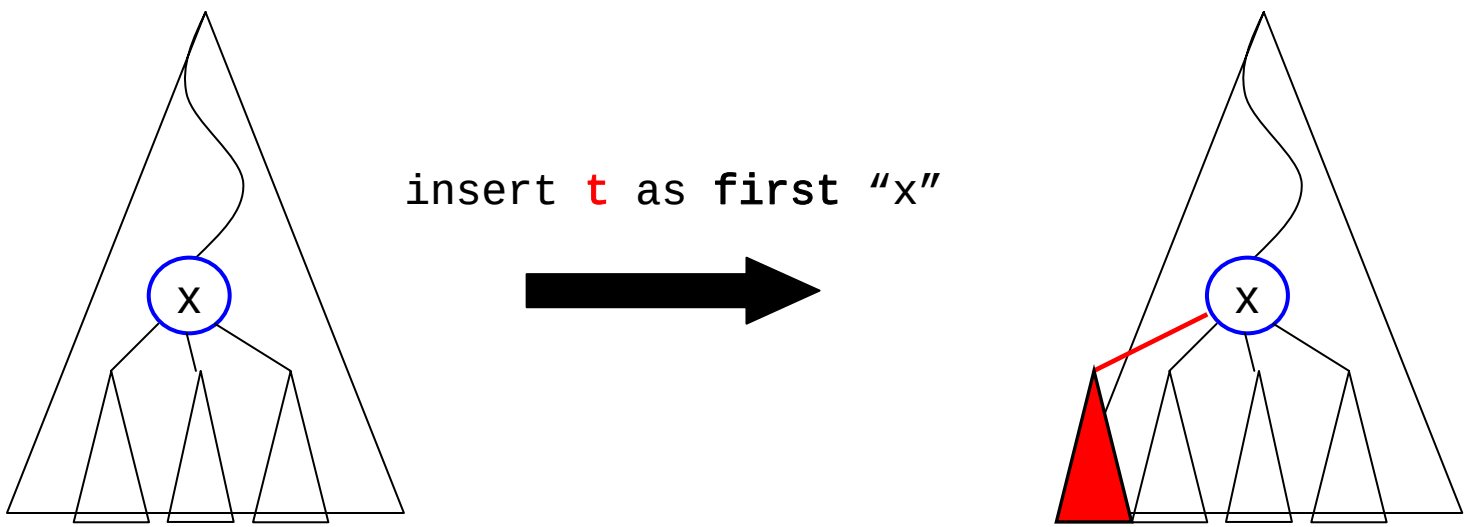

**Question** Can **t** be arbitrary?
For which **t** should the insert *fail*?

# XML Updates

**Note**
Every node has an "identity" = a unique identifier.
Also: there may be attributes of type "ID"!

**Example** updates for XML data

(2) *insert subtree "t"* as **first** of node x



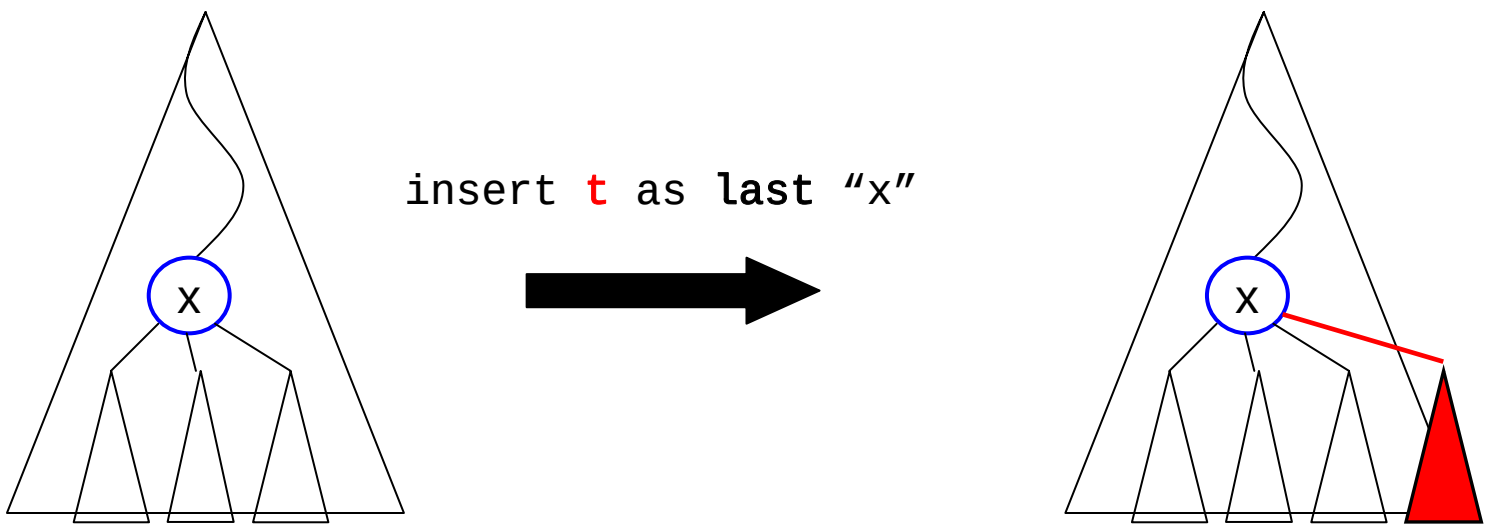

**Question** Can t be arbitrary?
For which t should the insert *fail*?

➔ non-unique values of ID-attributes!

# XML Updates

**Example** updates for XML data

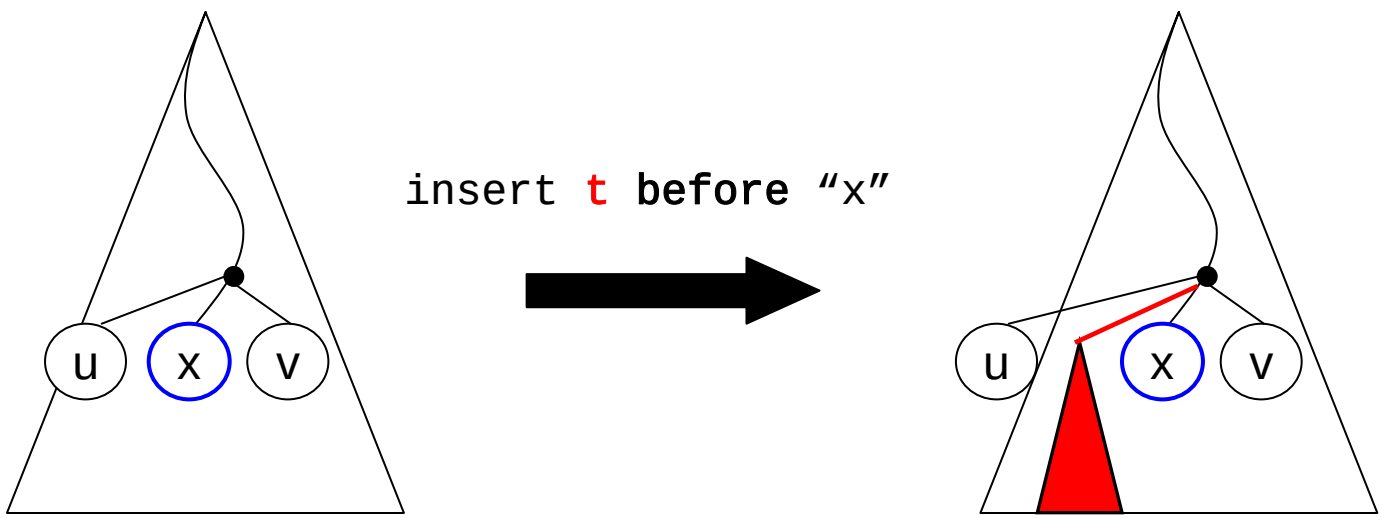
(3) *insert subtree "t"* as **last** of node x



# XML Updates

**Example** updates for XML data

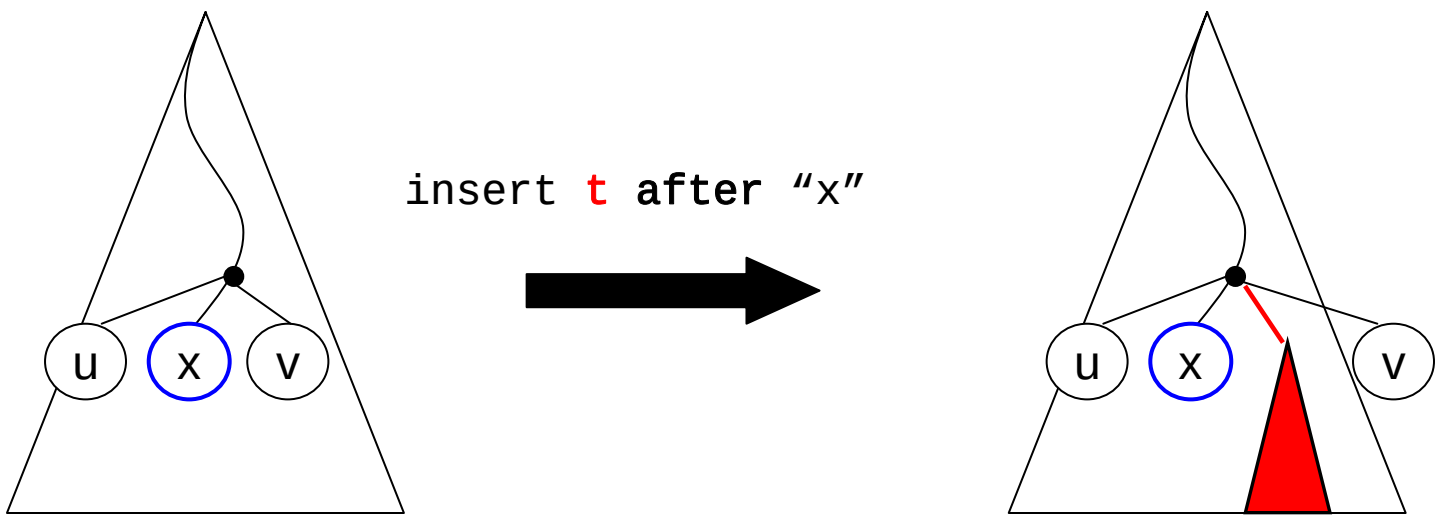
(4) *insert subtree "t"* **before** node x



# XML Updates

**Example** updates for XML data

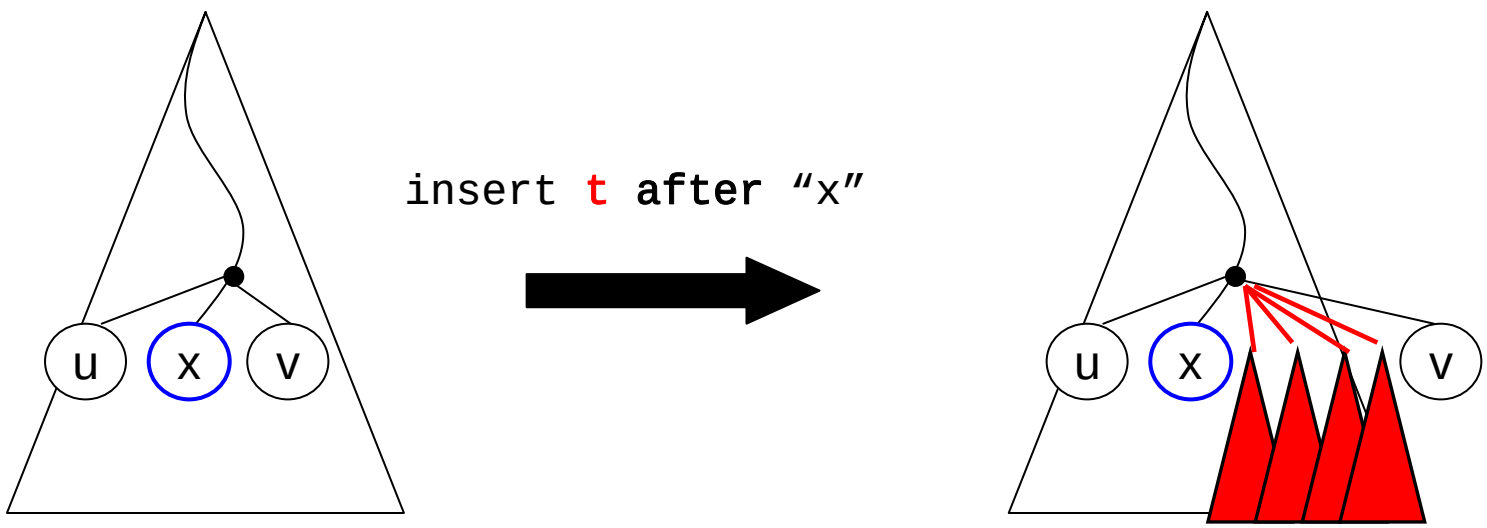
(5) *insert subtree "t"* **after** node x



# XML Updates

**Example** updates for XML data

(5) *insert subtree "t"* **after** node x




All **insert** operations: "subtree t" can easily be generalized to a sequence of subtrees ( t_1, t_2, t_3, …. t_n )

# XML Updates

**Example** updates for XML data

(5) *insert subtree "t"* **after** node x

Explicit examples

Insert a year element after the publisher of the first book.

```
do insert <year>2005</year>  after
   fn:doc("bib.xml")/books/book[1]/publisher
```

Navigating by means of several bound variables, insert a new police report into the list of police reports for a particular accident.

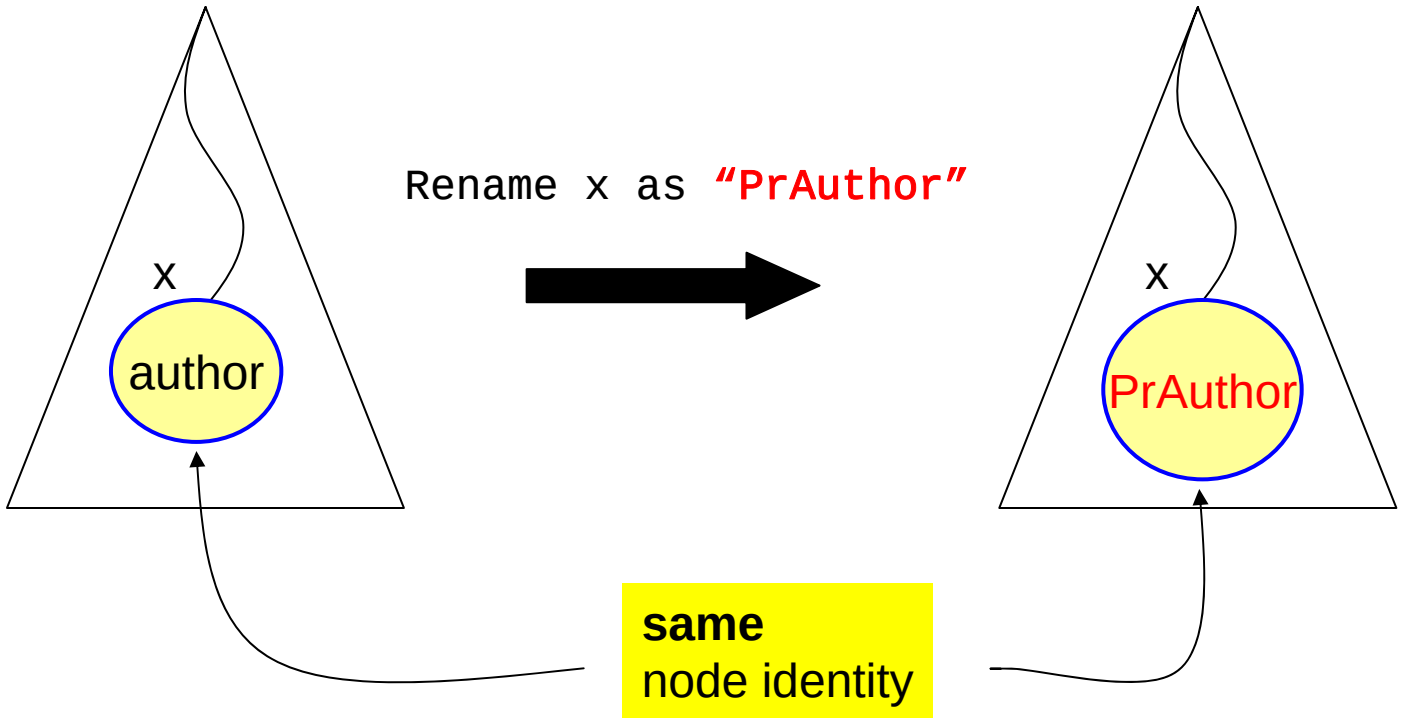
```
do insert $new-police-report
    as last into fn:doc("insurance.xml")/policies
       /policy[id = $pid]
       /driver[license = $license]
       /accident[date = $accdate]
       /police-reports
```

# XML Updates

**Example** updates for XML data

**Note**
The rename operation preserves node identity!

(6) *rename* node x as *name*



# XML Updates

**Note**
The rename operation preserves node identity!

**Example** updates for XML data

(6) *rename* node x as *name*

Explicit examples

Rename the first author element of the first book to `principal-author`.

```
do rename fn:doc("bib.xml")/books/book[1]/author[1]
          as "principal-author"
```

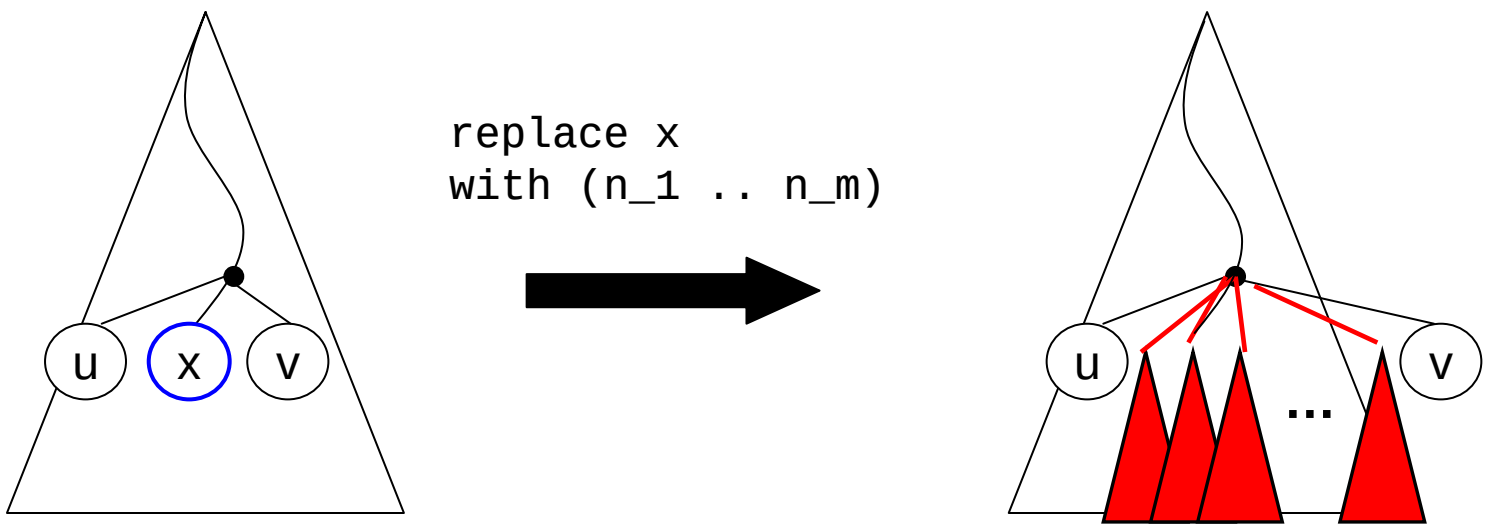
Rename the first `author` element of the first book to the QName that is the value of the variable `$newname`.

```
do rename fn:doc("bib.xml")/books/book[1]/author[1]
          as $newname
```

# XML Updates

**Example** updates for XML data

(7) *replace* node x with ( n_1 n_2 n_3 … n_m )



# XML Updates

**Example** updates for XML data

(7) *replace* node x with ( n_1 n_2 n_3 … n_m )

Explicit examples

Replace the publisher of the first book with the publisher of the second book.

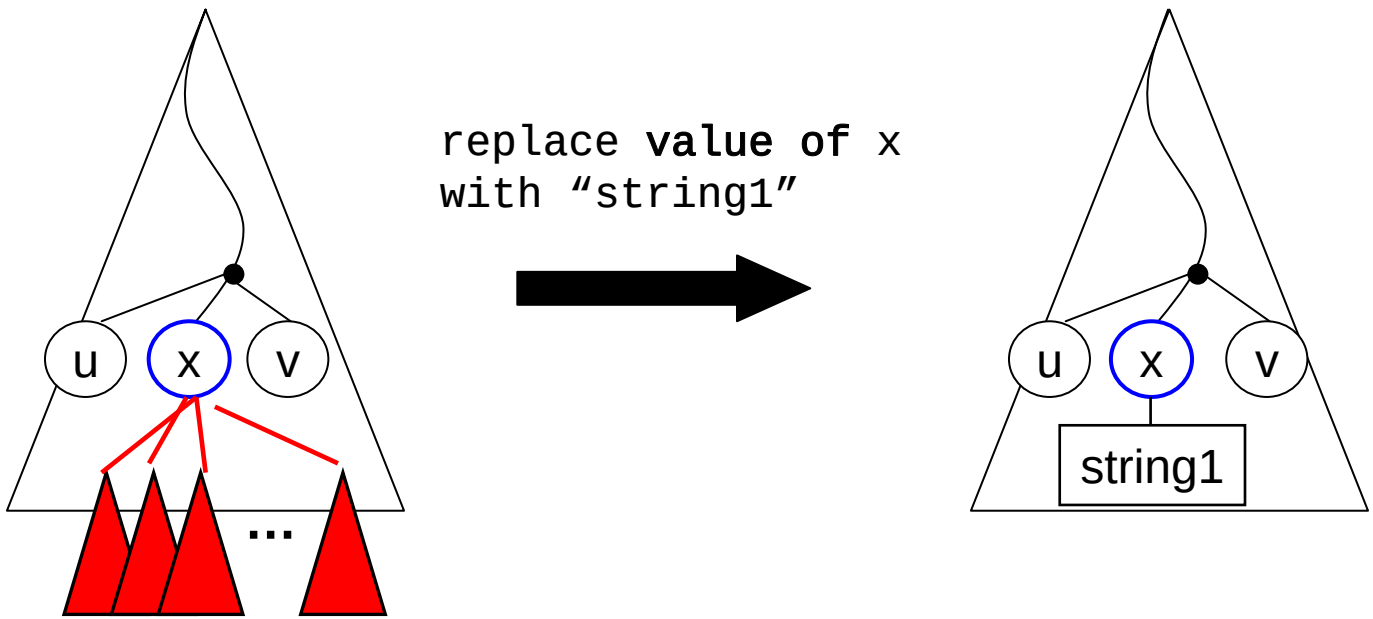
```
do replace  fn:doc("bib.xml")/books/book[1]/publisher

      with  fn:doc("bib.xml")/books/book[2]/publisher
```

# XML Updates

**Note**
The replace-value-of op. preserves node identity!

**Example** updates for XML data

(8) *replace value of* node x with "some string"




→ If x is a text-node, then text-content of x becomes "string1"
→ If x is an attribute node, then attribute value becomes "string1"

# XML Updates

**Note**
The replace-value-of op. preserves node identity!

**Example** updates for XML data

(8) *replace value of* node x with "some string"

Explicit examples

Increase the price of the first book by ten percent.

```
do replace value of

     fn:doc("bib.xml")/books/book[1]/price

     with fn:doc("bib.xml")/books/book[1]/price * 1.1
```

# XML Updates

**Questions**

→ What about the different node **types**

Can I insert an attribute node at any position?
Can I replace an attribute node by an element node, or vice versa?
etc

---

→ Do we really need so many different operations?
Which operation can be **simulated** by other ones?

---

→ How to *generalize the target*, from a node to an XPath expression?
(bulk updates, using one operation)

**Semantical issues:** doc changes after first update,
this might affect the subsequent updates! How to deal with this?

# Snapshot Semantics

```
for $e in //a insert as first <a></a>
```

Semantics of this
on the document <a></a> ??

```
insert <phone>02 83060405</phone>
as last into //address/name[text()="Jonny Pizzicato"]
for $e in //phone
  rename $e as "telephone"
```

---

**Snapshot Semantics**

- ➔ Each update operation is logically applied to a separate snapshot of the original document.
- ➔ Updates are applied independently from each other to the original document. They don't see each others' effects.

- ➔ The order of the update operations is irrelevant.

# Type Issues

do delete <u>TargetExpr</u>

must eval. to a sequence (n_1…n_m) of nodes.
Otherwise: Type Error!

Semantics for all n_i, append `upd:delete(n_i)` to *pending update list*

# Type Issues

do delete <u>TargetExpr</u> ← must eval. to a sequence of nodes. Otherwise: Type Error!

do insert <u>SourceExpr</u> (as (first | last) into) | before | after <u>TargetExpr</u>

↓ evaluates to → Otherwise: Type Error

| n_1 n_2 … n_p | u_1 u_2 … u_p |
|---|---|
| $alist | $clist |

→ TargetExpr must evaluate to *single node* (called $target)
→ If before/after then $target must have a parent node ($parent)

as first/last

```
upd:insertAttributes($target,$alist);
upd:insertIntoAsLast($target,$clist)
```

} append to *pending update list*

before/after

```
upd:insertAttributes($parent,$alist)
upd:insertBefore($target,$clist)
```

} append to *pending update list*

# Type Issues

do delete TargetExpr

must eval. to a sequence of nodes.
Otherwise: Type Error!

must eval. to a sequence of attribute nodes followed by non-att nodes

do insert SourceExpr (as (first | last) into) | before | after TargetExpr

do replace TargetExpr with ExprSingle

evaluates to → Otherwise: Type Error

| n_1 n_2 … n_p | u_1 u_2 … u_p |
|---|---|
| $alist | $clist |

→ TargetExpr must evaluate to *single node* (called $target) and must have a parent ($parent)

If $target is element, text, comment, or PI node, then

```
upd:insertAttributes($parent,$alist);
upd:insertBefore($target,$clist)
upd:delete($target)
```

} append to *pending update list*

# Type Issues

do delete TargetExpr

must eval. to a sequence of nodes.
Otherwise: Type Error!

must eval. to a sequence of attribute nodes followed by non-att nodes

do insert SourceExpr (as (first | last) into) | before | after TargetExpr

do replace TargetExpr with ExprSingle

evaluates to → Otherwise: Type Error

| n_1 n_2 … n_p | u_1 u_2 … u_p |
|---|---|
| $alist | $clist |

→ TargetExpr must evaluate to *single node* (called $target) and must have a parent ($parent)

If $target is attribute node, then

```
upd:insertAttributes($parent,$alist);
upd:insertBefore($parent,$clist)
upd:delete($target)
```

} append to *pending update list*

# Ambiguity

If $target is element, text, comment, or PI node, then

do replace TargetExpr with ExprSingle

is the same as

do insert ExprSingle before TargetExpr
do delete TargetExpr

Many more data-dependent ambiguities

insert as last = insert as first, if there are no children
insert as first = insert before on the first child, if that exists
insert as last = insert after on the last child, if that exists
…

# Challenges: Physical Updates

**Questions**

→ How to do updates on a DAG?

What will be different?
Are incremental updates possible?

---

→ How to do updates on a PRE/POST-encoding?

What will be different?
Are incremental updates possible?

# XUpdate: Text node updates

Obviously, the kind of $c$ determines the overall impact on the updated tree and its encoding.

## XUpdate: replacing text by text

```
<a>
  <b id="0">foo</b>
  <b id="1">bar</b>
</a>
```

$\Downarrow$

```
<xupdate:update select="//b[@id = 1]">
  foo
</xupdate:update>
```

```
<a>
  <b id="0">foo</b>
  <b id="1">foo</b>
</a>
```

- New content $c$: a **text node**.

# XUpdate: Text node updates

Translated into, *e.g.*, the XPath Accelerator representation, we see that

- Replacing text nodes by text nodes has **local impact** only on the *pre*/*post* encoding of the updated tree.

## XUpdate statement leads to local relational update

| *pre* | *post* | ··· | *text* |
|---|---|---|---|
| 0 | 4 | | NULL |
| 1 | 1 | | NULL |
| 2 | 0 | | foo |
| 3 | 3 | | NULL |
| 4 | 2 | | bar |

⇒

| *pre* | *post* | ··· | *text* |
|---|---|---|---|
| 0 | 4 | | NULL |
| 1 | 1 | | NULL |
| 2 | 0 | | foo |
| 3 | 3 | | NULL |
| 4 | 2 | | foo |

- Similar observations can be made for updates on comment and processing instruction nodes.

# XUpdate: Structural updates

## XUpdate: inserting a new subtree

```
<a>
  <b><c><d/><e/></c></b>
  <f><g/>
     <h><i/><j/></h>
  </f>
</a>
```

```
<xupdate:update select="/a/f/g">
  <k><l/><m/></k>
</xupdate:update>
```

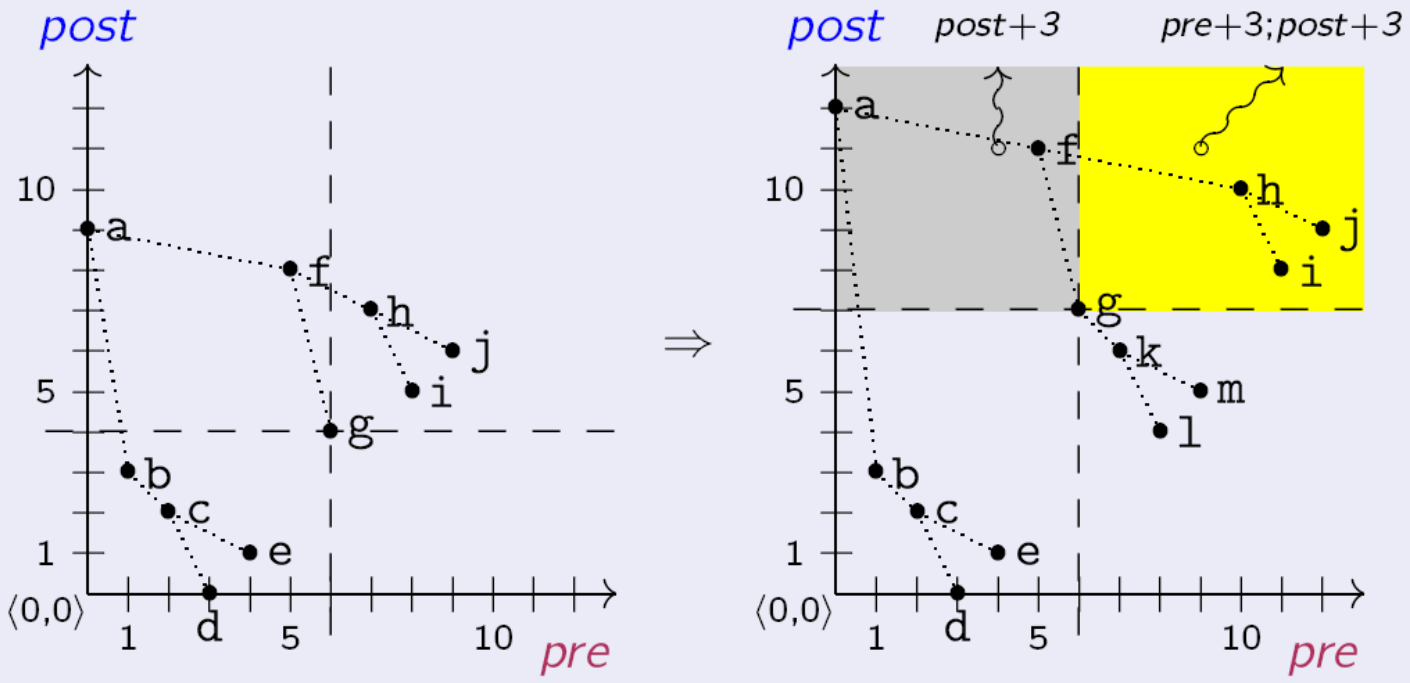
⇓

```
<a>
  <b><c><d/><e/></c></b>
  <f><g><k><l/><m/></k></g>
     <h><i/><j/></h>
  </f>
</a>
```

**Question:** What are the effects w.r.t. our structure encoding. . . ?

# XUpdate: Global impact on encoding

## Global shifts in the *pre*/*post* Plane



# XUpdate: Global impact on *pre*/*post* plane

## Insert a subtree of $n$ nodes below parent element $v$

1. $post(v) \leftarrow post(v) + n$
2. $\forall v' \in v/\texttt{following::node()}$:
   $pre(v') \leftarrow pre(v') + n; post(v') \leftarrow post(v') + n$
3. $\forall v' \in v/\texttt{ancestor::node()}$:
   $post(v') \leftarrow post(v') + n$

## Cost (tree of $N$ nodes)

$$\underbrace{O(N)}_{②} + \underbrace{O(\log N)}_{③}$$

## Update cost

③ is not so much a problem of cost but of **locking.** Why?

# Updates and fixed-width encodings

**Theoretical result [Milo *et.al.*, PODS 2002]**

There is a sequence of updates (subtree insertions) for any persistent[49] tree encoding scheme $\mathcal{E}$, such that $\mathcal{E}$ **needs labels of length** $\Omega(N)$ to encode the resulting tree of $N$ nodes.

- **Fixed-width** tree encodings (like XPath Accelerator) are inherently **static.**

  $\Rightarrow$ Non-solutions:
  - **Gaps** in the encoding,
  - encodings based on **decimal fractions**.

[49] A node keeps its initial encoding label even if its tree is updated.

# A variable-width tree encoding: ORDPATH

Here we look at a particular variant of a hierarchical numbering scheme, optimized for updates.

- The **ORDPATH** encoding (used in MS SQL Server™) assigns node labels of **variable length.**

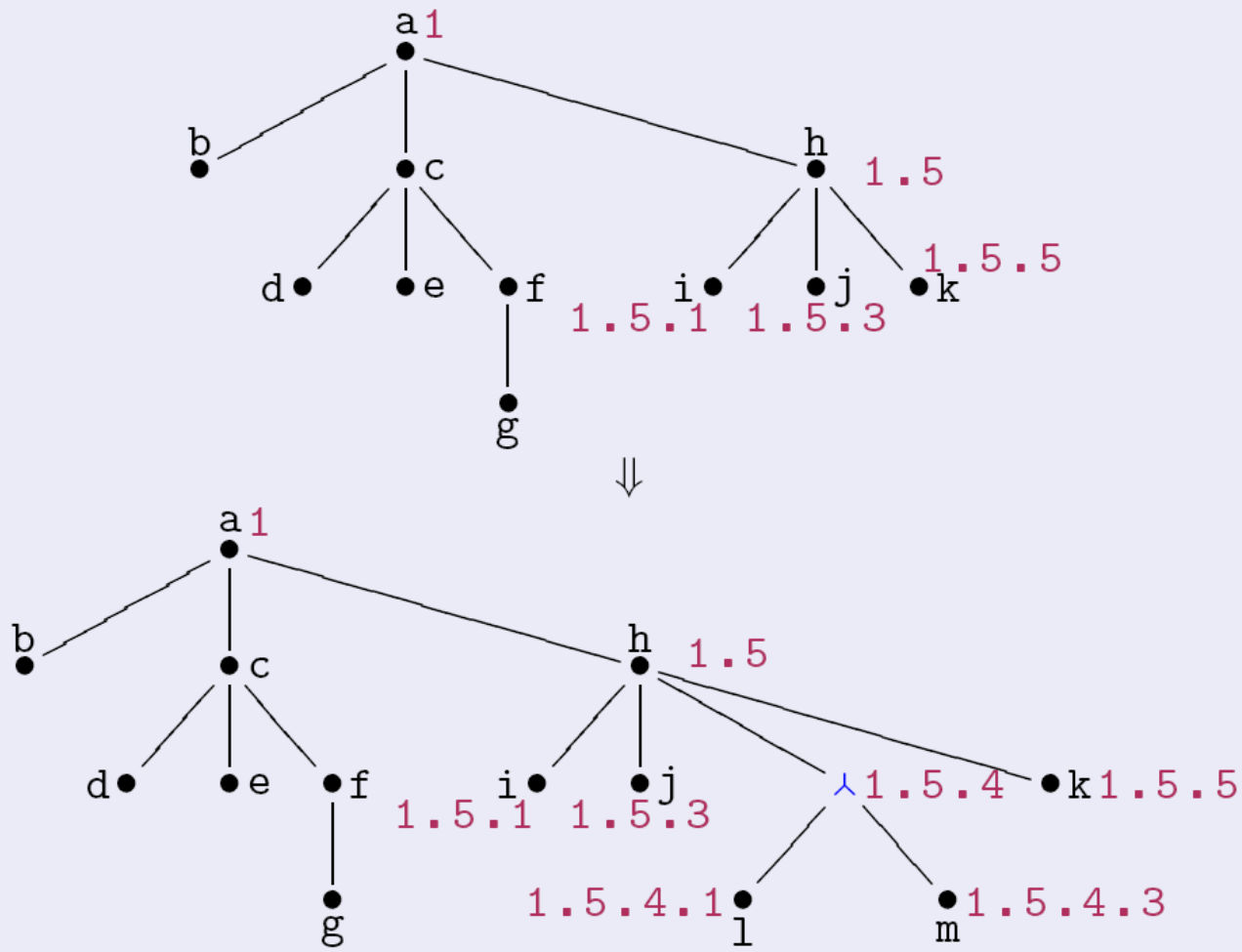
## ORDPATH labels for an XML fragment

1. The fragment root receives label 1.
2. The $n$th ($n = 1, 2, \ldots$) child of a parent node labelled $p$ receives label $p\,.\,(2 \cdot n - 1)$.

- Internally, ORDPATH labels are not stored as .-separated ordinals but using a prefix-encoding (similarities with Unicode).

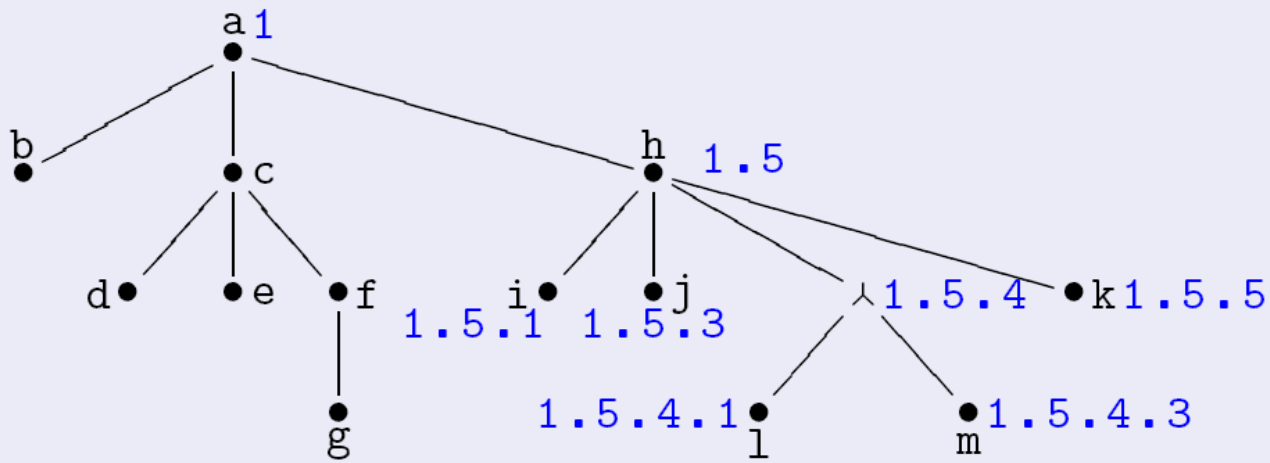
# ORDPATH: Insertion between siblings (Example)

## Insertion of (<l/>, <m/>) between <j/> and <k/>



# ORDPATH: Insertion between siblings

## ORDPATH: Insertions at arbitrary locations?




Determine ORDPATH label of new node $v$ inserted

1. to the right of <k/>,
2. to the left of <i/>,
3. between <j/> and <l/>,
4. between <l/> and <m/>,

# Processing XQuery and ORDPATH

Is ORDPATH a suitable encoding $\mathcal{E}$?

Mapping core operations of the XQuery processing model to operations on ORDPATH labels:

### $v$/parent::node()

1. Let $p$ . $m$ . $n$ denote $v$'s label ($n$ is odd).
2. If the rightmost ordinal ($m$) is even, remove it. Goto ②.

In other words: the carets (⅄) do not count for ancestry.

### $v$/descendant::node()

1. Let $p$ . $n$ denote $v$'s label ($n$ is odd).
2. Perform a lexicographic index range scan from $p$ . $n$ to $p$ . $(n+1)$—the *virtual following sibling* of $v$.

# ORDPATH: Variable-length node encoding

- Using (4 byte) integers for all numbers in the hierarchical numbering scheme is an obvious waste of space!
- Fewer (and variable number of) bits are typically sufficient;
- they may bear the risk of running out of new numbers, though. In that case, even ORDPATH cannot avoid *renumbering*.
  - In principle, though, *no bounded* representation can absolutely avoid the need for renumbering.
- Several approaches have been proposed so as to alleviate the problem, for instance:
  - use a variable number of bits/bytes, akin to Unicode,
  - apply some (order-preserving) hashing schemes to shorten the numbers,
  - ...

# ORDPATH: Variable-length node encoding

- For a 10 MB XML sample document, the authors of ORDPATH observed label lenghts between 6 and 12 bytes (using Unicode-like compact representations).
- Since ORDPATH labels encode **root-to-node** paths, node labels share **common prefixes.**

**ORDPATH labels of <l/> and <m/>**

```
1.5.4.1
1.5.4.3
```

⇒ Label comparisons often need to inspect encoding bits at the far right.

- MS SQL Server™ employs further path encodings organized in **reverse** (node-to-root) order.
- **Note:** Fixed-length node IDs (such as, *e.g.*, preorder ranks) typically fit into CPU registers.

END
Lecture 13