Informatics 1B, 2008

School of Informatics, University of Edinburgh

Data and Analysis

Note 7

Querying XML Documents with XQuery

Alex Simpson

Part II — Semistructured Data

XML

Corpora

How do we extract data from an XML document?

Since an XML document is a text document, one option is to use methods based on text search.

But this ignores the element structure of the document.

XQuery is a dedicated language for forming queries based on the *tree structure* of an XML document

Useful for:

- Performing relational-database-type queries on data published as XML
- Extracting annotated content from marked-up text documents
- All queries that exploit the tree structure of XML

Some features of XQuery

XQuery is a *declarative* language. (As with SQL's "conceptual evaluation", XQuery queries can be given understod via a procedural reading, but this need not coincide with the actual evaluation strategy used in practice.)

As well as using XML documents for its source data, XQuery is capable of producing XML documents as its output. That is, the result of a query can be produced in XML format. Thus, one way of viewing XQuery is as a language for transforming XML documents into XML documents.

(This is analogous to SQL, which takes tables as input to a query and produces a table as output.)

XQuery is compatible with important XML facilities that are beyond the scope of this course, e.g.: namespaces; and XML Schema and its datatypes.

FLWOR expressions

The *FLWOR expression* (pronounced "flower") is the basic query form in Xquery.

- **F**or — iterates over data
- **L**et — binds a variable to a list of data
- **W**here — filters out data that does not satisfy a specified property
- **O**rder — specifies how the data is to be ordered
- **R**eturn — specifies and formats the information to be returned

The next slide displays an illustrative query.

(You should understand this by the end of the lecture, not immediately!)

Example query

```
<WorldLakes> {
for $x in doc("gaz.xml")/Gazetteer/Country
let $y := $x//Feature[@type='Lake']
where $y
order by $x/Name/text()
return
  <Country>
    {$x/Name}
    {for $z in $y/text() return <Lake>{$z}</Lake>}
  </Country>
}
</WorldLakes>
```

Example query (very!) informally

One can read this query as follows:

> For every country **`$x`** in the gazetteer,
>
> bind **`$y`** to the list of all lakes in country **`$x`**,
>
> then, for those countries for which there is at least one lake in **`$y`**,
>
> ordering countries alphabetically by country name,
>
> return (in a specified format) the country name together with the list of all lakes in the country.

The next two slides give a fragment of a source document, followed by a fragment of the corresponding output document.

Note that the output document contains data about Italy, which is not displayed in the source document (due to lack of space on the slide).

```
<Gazetteer>
  <Country>
    <Name>Slovenia</Name>
    <Population>2,020,000</Population>
    <Capital>Ljubljana</Capital>
    <Region>
      <Name>Gorenjska</Name>
      <Feature type="Lake">Bohinj</Feature>
      <Feature type="Mountain">Triglav</Feature>
      <Feature type="Lake">Bled</Feature>
    </Region>
  </Country>
  <!-- data for other countries including Italy here -->
</Gazetteer>
```

```
<WorldLakes>
  <Country>
    <Name>Italy</Name>
    <Lake>Como</Lake>
    <Lake>Maggiore</Lake>
    <Lake>Garda</Lake>
    <Lake>Levico</Lake>
  </Country>
  <Country>
    <Name>Slovenia</Name>
    <Lake>Bohinj</Lake>
    <Lake>Bled</Lake>
  </Country>
</WorldLakes>
```

Path expressions

Path expressions are a crucial component of XQuery.

A *path expression* denotes a *set* of nodes in the document tree.

This set of nodes is returned as a list of nodes (without duplicates) sorted in *document order* (the order in which the nodes appear in the XML document)

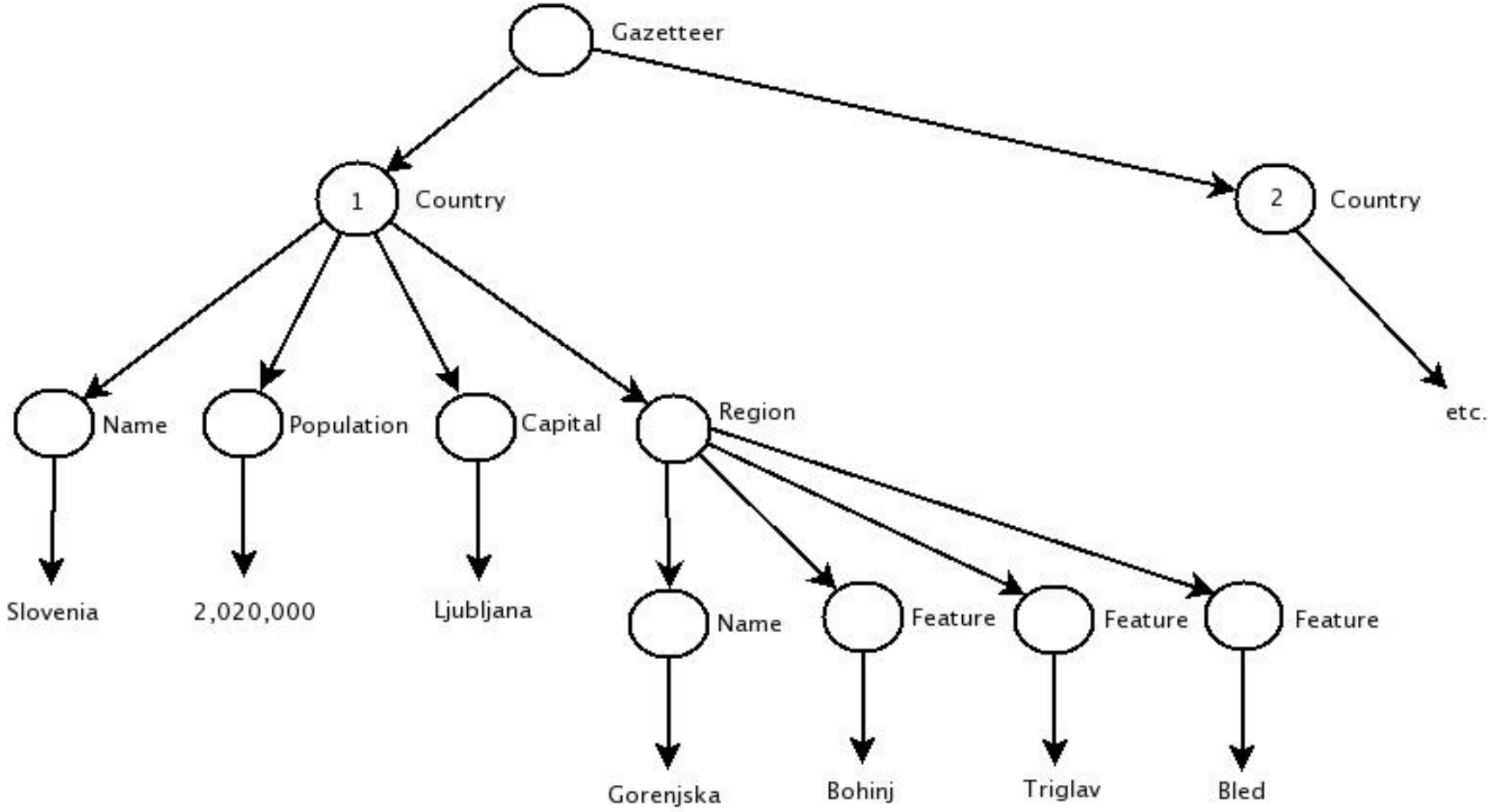
In the example query on slide 7.6, the path expression

```
doc("gaz.xml")/Gazetteer/Country
```

Denotes the set of **`Country`** elements that are nested immediately inside the root element **`Gazeteer`** of the XML document **`gaz.xml`**.
(In fact all the **`Country`** elements in the document are of this form.)

These nodes are numbered, in document order, on the next slide.

gaz.xml


Gazetteer
1
Country
2
Country
Name
Population
Capital
Region
etc.
Slovenia
2,020,000
Ljubljana
Name
Feature
Feature
Feature
Gorenjska
Bohinj
Triglav
Bled

Path expressions (continued)

A path expression can be understood as describing a set of possible paths from the root of the tree.

The set of nodes it defines is the set of all nodes reached as final destinations of the described paths.

Paths are built up step-by-step as the path expression is read from left-to-right.

Each path is constructed by a *context node* that travels over the tree, starting from the root. This moves, according to certain rules, depending on the continuation of the path expression.

Path expressions (building blocks)

We will only consider path expressions built up using the three building blocks below.

- **`/nodeTest`**

  Move the context down the tree to a *child* node passing **`nodeTest`**

- **`//nodeTest`**

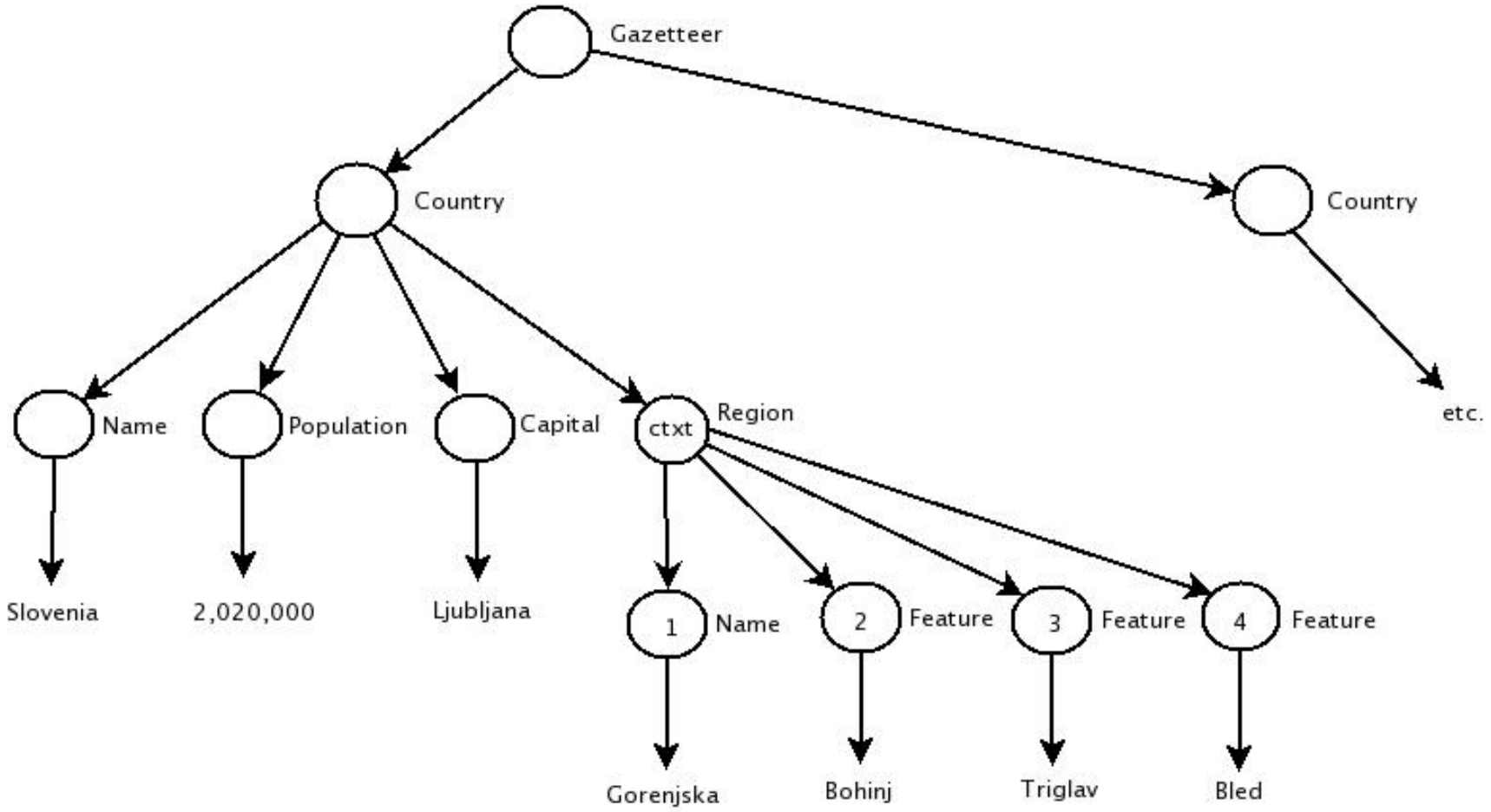
  Move the context down to a *descendent* node passing **`nodeTest`**

- **`/..`**

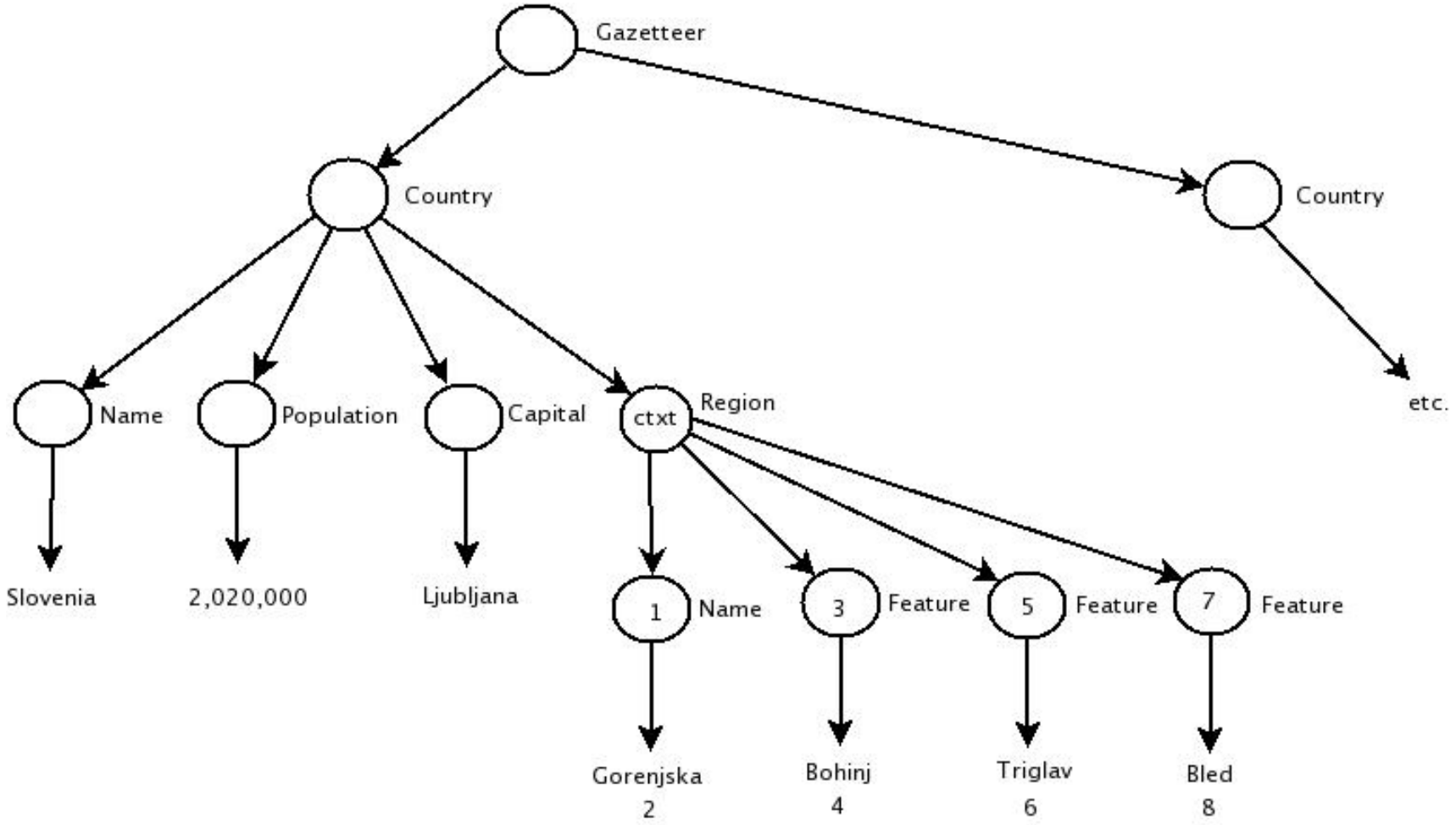
  Move the context up the tree to its (unique) *parent* node

The highlighted tree concepts are illustrated on the next three slides

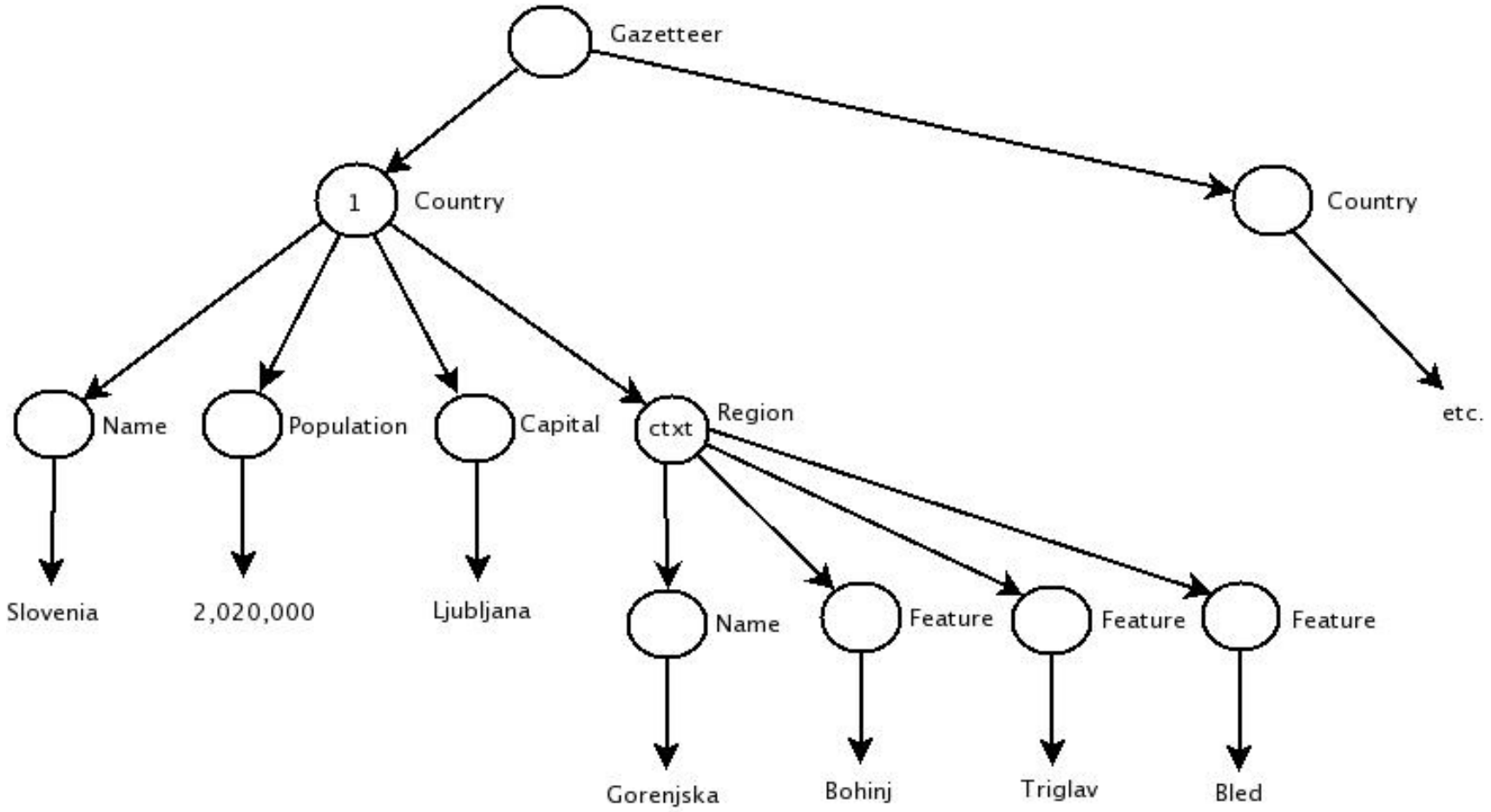
Children nodes, in document order, of the context node “ctxt”



Descendent nodes, in document order, of the context node “ctxt”



The *parent node* of the context node “ctxt”



Path expressions (node tests)

- ***`ElementName`***

  Selects only those nodes that are ***`ElementName`*** elements

- `*`

  Selects all element nodes
  (Warning. Sometimes `*` can have other meanings too, but we will not consider any situation in which this occurs.)

- `text()`

  Selects only those nodes that are character data

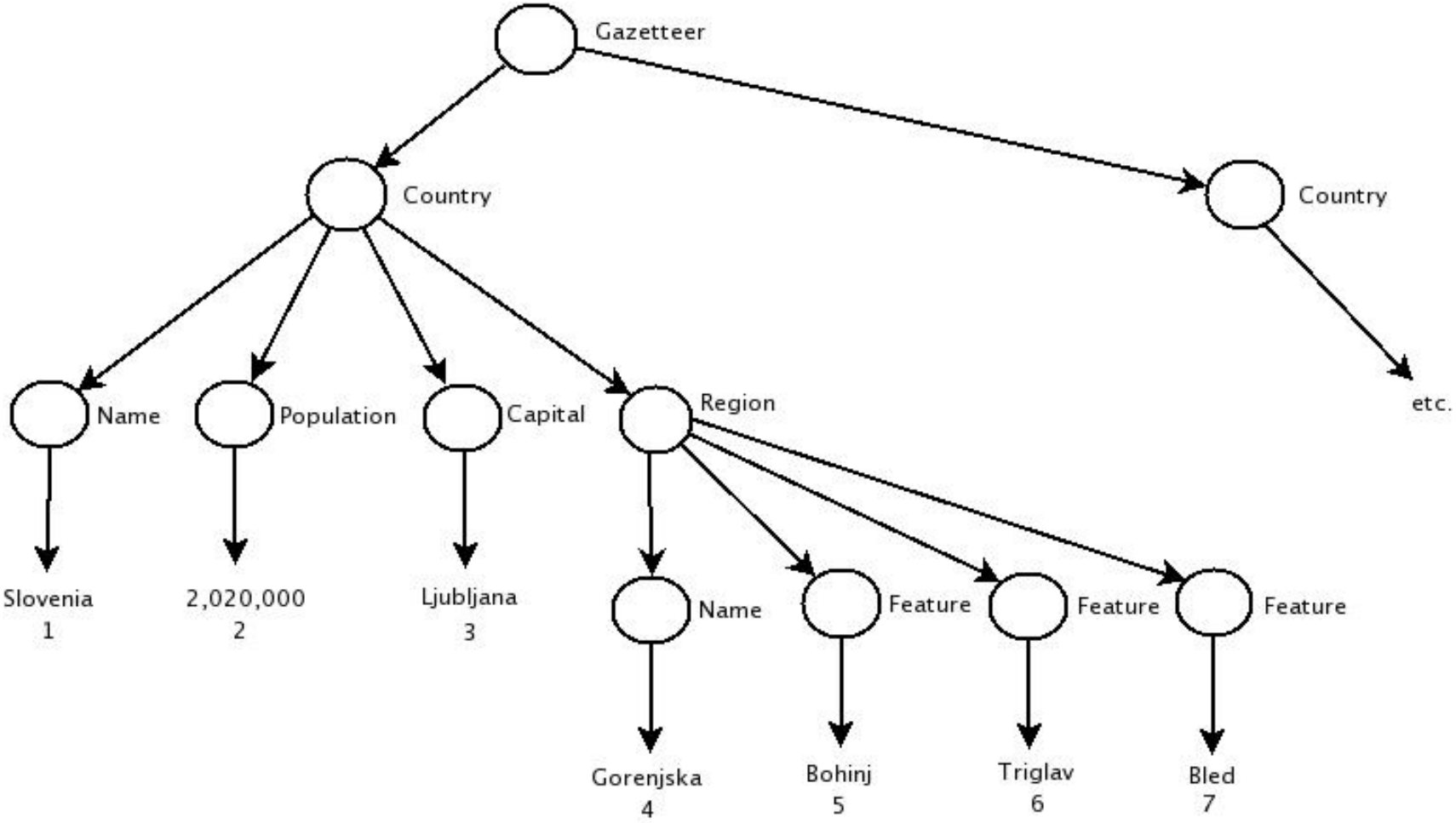
The node test can be followed by a *predicate* in square brackets which further constrains the nodes being selected.

Path expressions (predicates)

We consider just some of the more common examples of predicates.

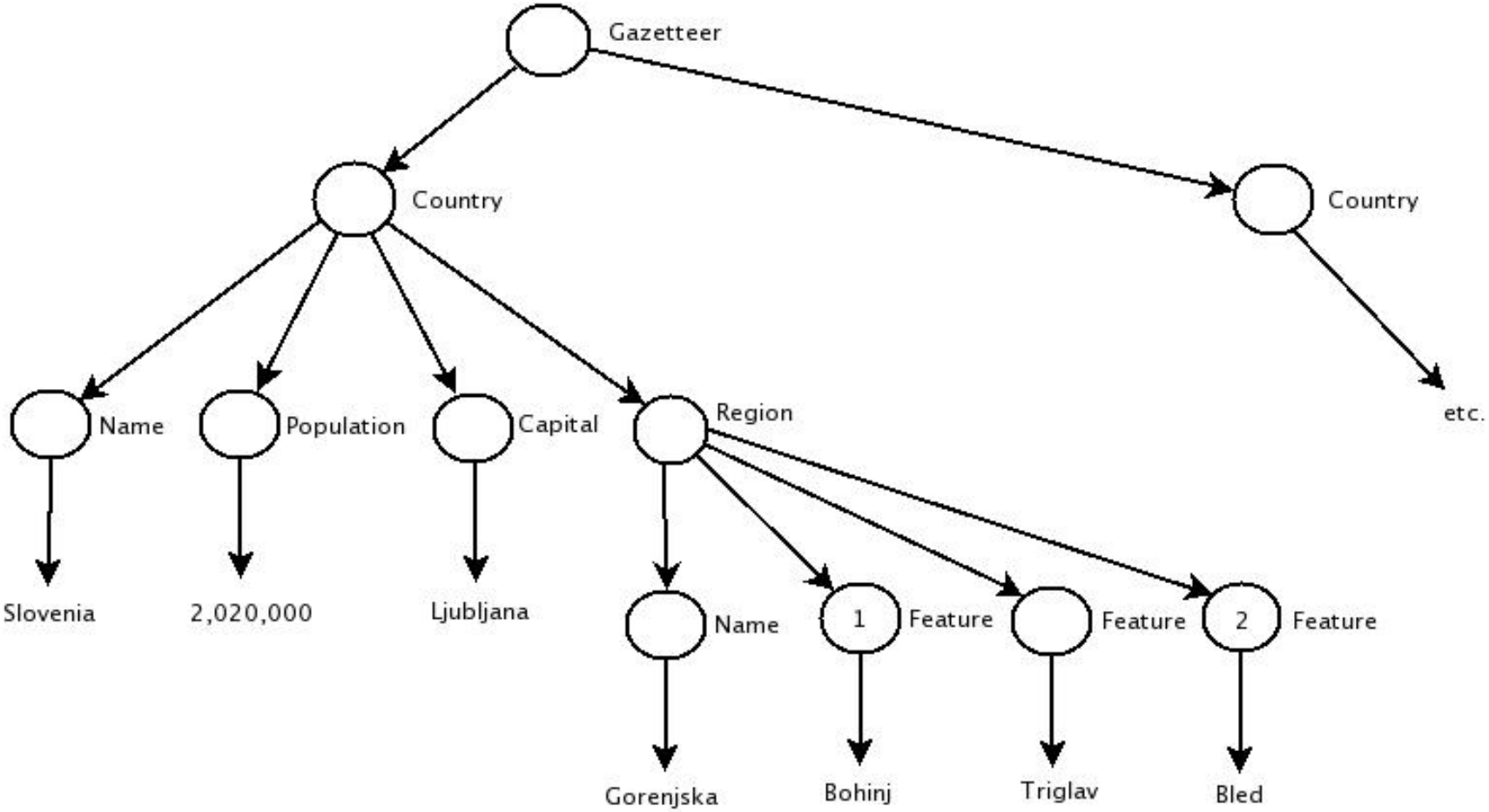
- **`[@attributeName]`**
  Selects only those elements that have an attribute ***`attributeName`***
- **`[@attributeName=value]`**
  Selects only those elements that have an attribute ***`attributeName`*** with value ***`value`***. (Other possible tests include **`!=`** for "not equal to".)
- **`[pathExpression]`**
  Selects only those nodes for which there exists a continuation path (from the current node) matching ***`pathExpression`***.
- **`[pathExpression=value]`**
  Selects those nodes for which there exists a continuation path (from the current node) matching ***`pathExpression`*** such that the final node of the path is equal to *value*.

Example 1: `doc("gaz.xml")//text()`



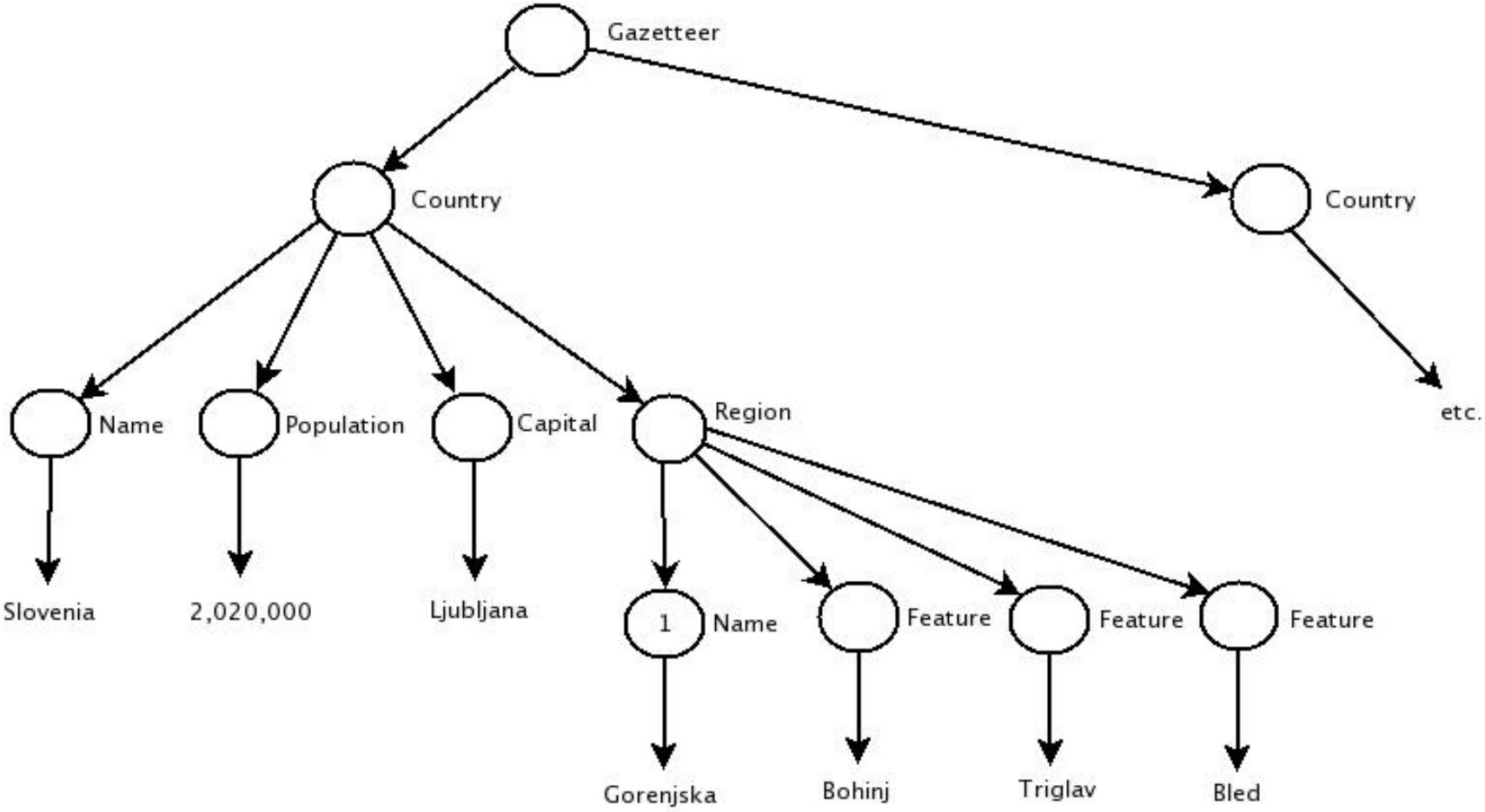
Example 2: All elements that are lakes

```
doc("gaz.xml")//*[@type='Lake']
```



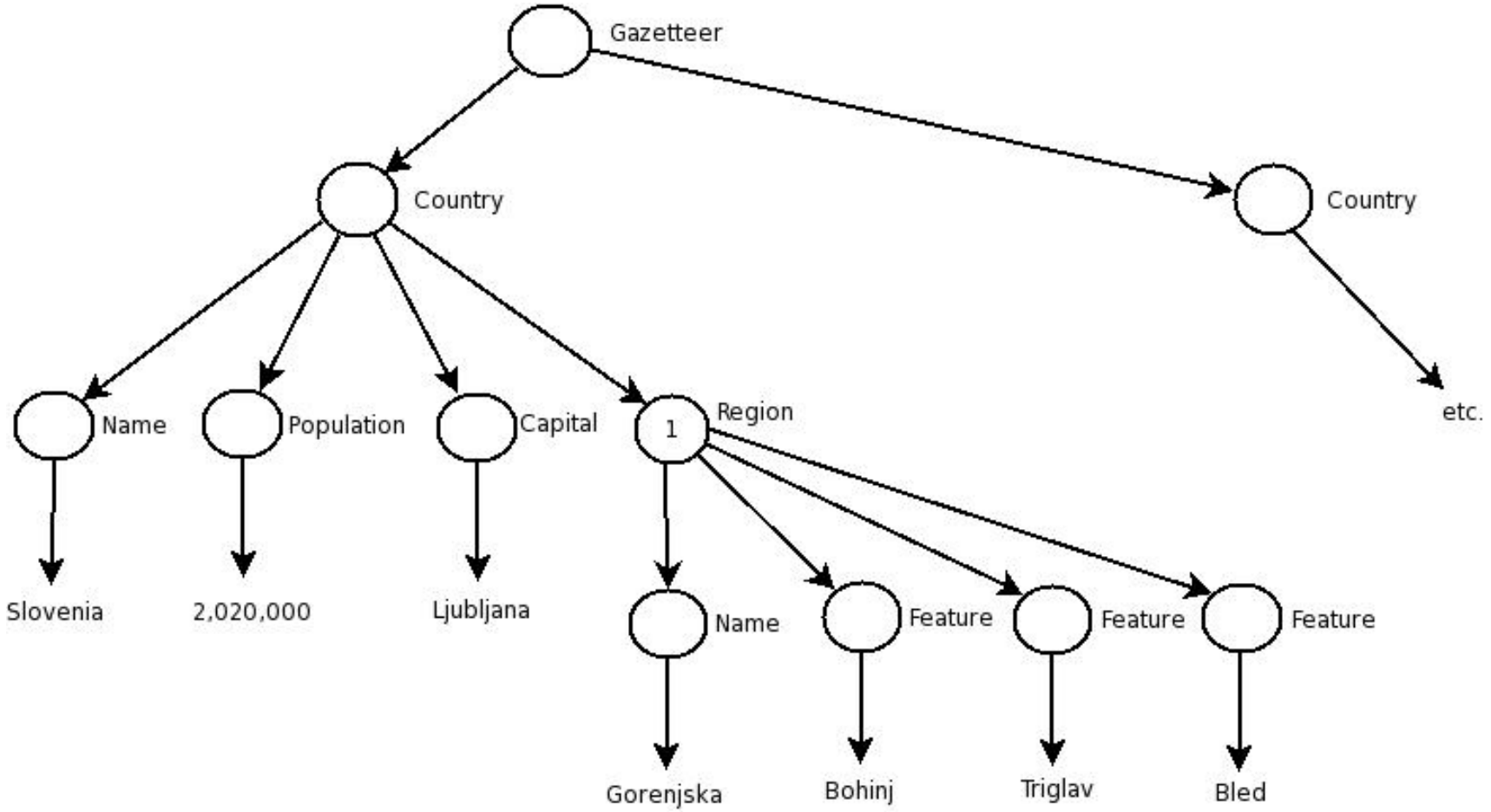
Example 3: All **Name** elements for regions containing lakes

```
doc("gaz.xml")//Feature[@type='Lake']/../Name
```



Example 4: All `Region` elements for Slovenia

```
doc("gaz.xml")//Country[Name/text()='Slovenia']/Region
```



XPath

XQuery is built on XPath 2.0, a rich and expressive language for path expressions.

The path expressions we have considered are all examples of XPath expressions.

However, we have considered only the simplest XPath features, and only their (easy to use) abbreviated form.

Other features in XPath include: navigation based on document order, position and size of context, treatment of namespaces, a rich language of expressions, non-abbreviated syntax.

Evaluating a FLWOR expression

For expressions

The **`for`** expression

`for $x in` ***`path-expression`***

sets the variable **`$x`** to each node in the list returned by the path expression in turn.

Accordingly, in the example,

```
for $x in doc("gaz.xml")/Gazetteer/Country
```

the variable **`$x`** is first set to node 1 in the tree on 7.11, and then to node 2.

Let expressions

The **`let`** expression

`let $y :=` ***`path-expression`***

sets the variable **`$y`** to the entire list of nodes returned by the path expression.

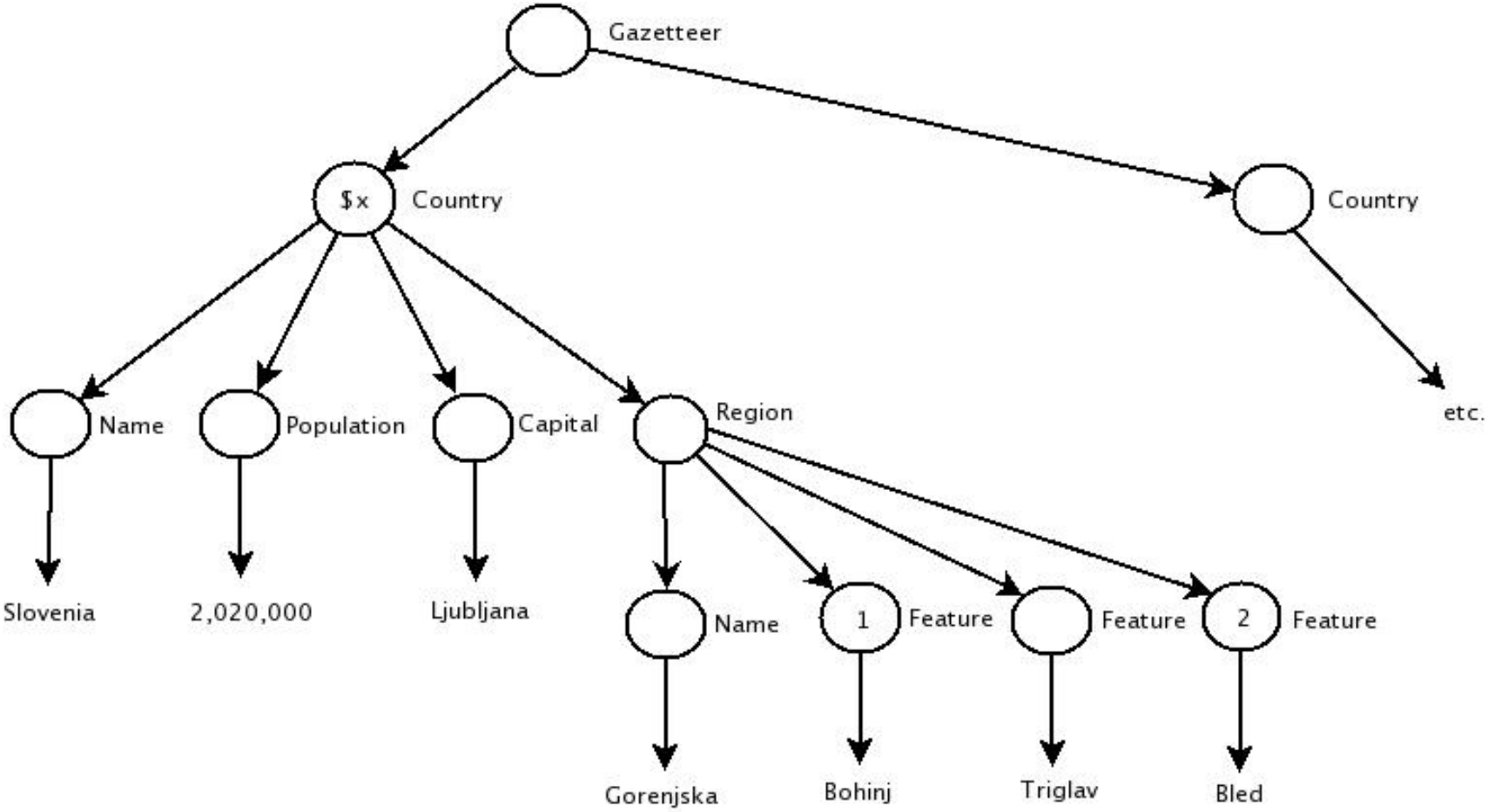
Accordingly, if the variable **`$x`** takes the value of the **`Country`** node for Slovenia, then the **`let`** expression

```
let $y := $x//Feature[@type='Lake']
```

sets **`$y`** to the list consisting of the two **`Feature`** elements for lakes in Slovenia.

This is illustrated on the next slide.

```
$x//Feature[@type='Lake']
```



Return expressions

The **return** expression specifies the data to be returned by the query.

We illustrate by example.

```
for $x in doc("gaz.xml")/Gazetteer/Country/Name
return $x
```

produces a list of as many name elements as countries (assuming each country has a single name in the document), e.g.

```
<Name>Slovenia</Name>, <Name>Italy</Name>
```

Note that this is not in itself an XML document.

To construct an XML document, we use the *document constructor* { . . . }, which allows XQuery to be included within an XML wrapper document, and also replaces separating commas with separating spaces.

Return expressions (continued)

We amend the previous example as follows:

```
<Countries>{
for $x in doc("gaz.xml")/Gazetteer/Country/Name
return $x
}</Countries>
```

This produces

```
<Countries>
<Name>Slovenia</Name> <Name>Italy</Name>
</Countries>
```

The XQuery command inside the document constructor is called an *enclosed expression*

Return expressions (continued)

It is often useful to nest return expressions (indeed general FLWOR expressions), as in a simplified version of the query on 7.6.

```
<WorldLakes> {
for $x in doc("gaz.xml")/Gazetteer/Country
let $y := $x//Feature[@type='Lake']
return
  <Country>
    {$x/Name}
    {for $z in $y/text() return <Lake>$z</Lake>}
  </Country>
} </WorldLakes>
```

The result of this query is shown on the next slide

```
<WorldLakes>
  <Country>
    <Name>Slovenia</Name>
    <Lake>Bohinj</Lake>
    <Lake>Bled</Lake>
  </Country>
  <Country>
    <Name>Italy</Name>
    <Lake>Como</Lake>
    <Lake>Maggiore</Lake>
    <Lake>Garda</Lake>
    <Lake>Levico</Lake>
  </Country>
</WorldLakes>
```

Order and where expressions

The original query on 7.6 contains additional **`where`** and **`order`** clauses.

```
for $x in ...
let $y := ...
where $y
order by $x/Name/text()
return ...
```

The **`where`** clause applies **`$y`** as a test. Since, **`$y`** is a sequence of nodes, the test returns **`false`** if **`$y`** is the empty sequence, and **`true`** otherwise. Only when the test returns **`true`** (i.e. when **`$y`** is non-empty) does execution proceed on to the **`order`** clause. If the test returns **`false`** then execution instead returns to the **`for`** clause for **`$x`** to take its next value.

The **`order`** clause reorders the output by order on the text content of **`$x/Name/text()`**. Since this is text data, the ordering is lexicographic.

and more . . .

For full details on XPath and XQuery see the W3C specifications:

http://www.w3.org/TR/xpath

http://www.w3.org/TR/xquery/

The full languages are vast (and XQuery is something of a minefield!)

Tutorials can be found at:

http://www.w3schools.com/xpath/

http://www.w3schools.com/xquery/

XPath is a particularly useful language to learn, since it is also used in XML Schema and in XSLT (a dedicated language for transforming XML documents)