Informatics 1, 2009

School of Informatics, University of Edinburgh

Data and Analysis

Part II

Semistructured Data

Alex Simpson

Recommended reading

[DMS] covers the main Part II topics, but rather superficially.

For a more in-depth treatment see:

[XWT] An Introduction to XML and Web Technologies
A. Møller and M. Schwartzbach
Addison Wesley, 2006

"A superb summary of the main Web technologies. It is broad and deep giving you enough detail to get real work done. Eminently readable with excellent examples and touches of humour. This book is a gem."

Prof. Philip Wadler, University of Edinburgh

Part II — Semistructured Data

Recommended reading: Chapter 2 of [XWT]

pp. 227–231 of [DMS]

Background

Relational databases record data in tables conforming to relational schemata. This imposes rigid structure on data

In many situations, it is useful to structure data in a less rigid way; for example:

- when the data needs to be made publicly available in a standard and easily readable data format;
- when we wish to *mark up* (i.e. annotate) existing unstructured data (e.g. text) with additional information (e.g. semantic information);
- when the data possesses a natural hierarchical structure and/or the structure of the data we wish to record varies from item to item.

Semistructured data

Semistructured data imposes a loose structure on data, hence the choice of terminology.

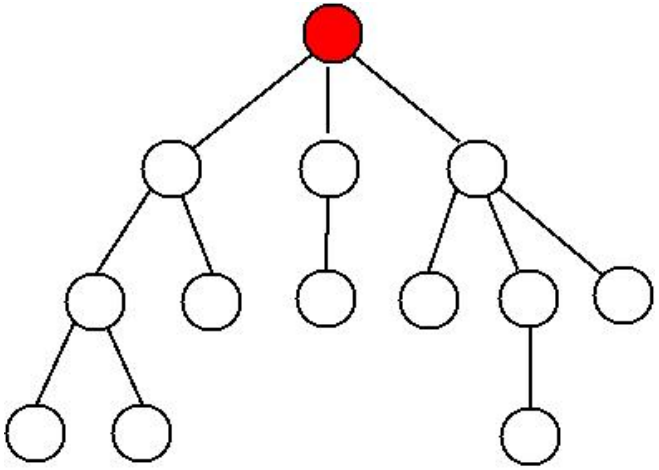
The principal structure imposed on data is that of a *tree*.

Before seeing how trees are used to structure data, we review basic terminology for talking about trees.

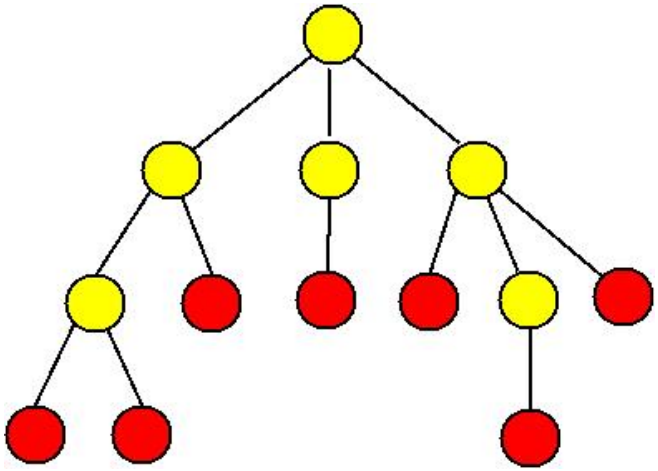
Recall, a *tree* consists of a set of *nodes*, amongst which there is a unique *root node*. For every node in the tree, there is a unique path from the root node to the node.

Nodes separate into two disjoint classes: *leaves* and *internal nodes*.

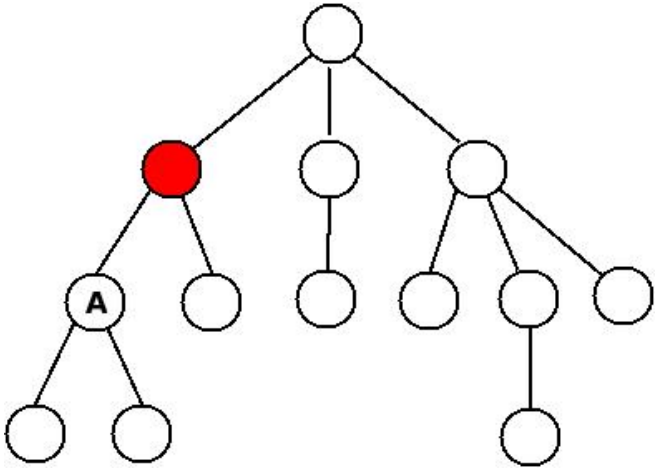
Every node other than the root has a unique *parent* node. Every internal node has a nonempty set of *children* nodes.



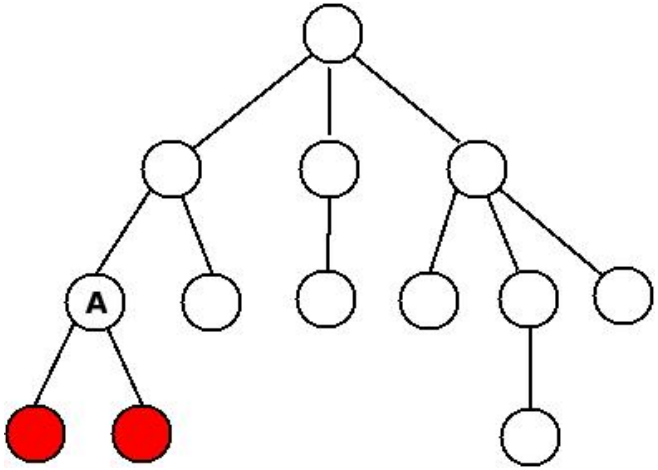
Root node



Leaves and internal nodes



Parent of A



Children of A

Semistructured data models

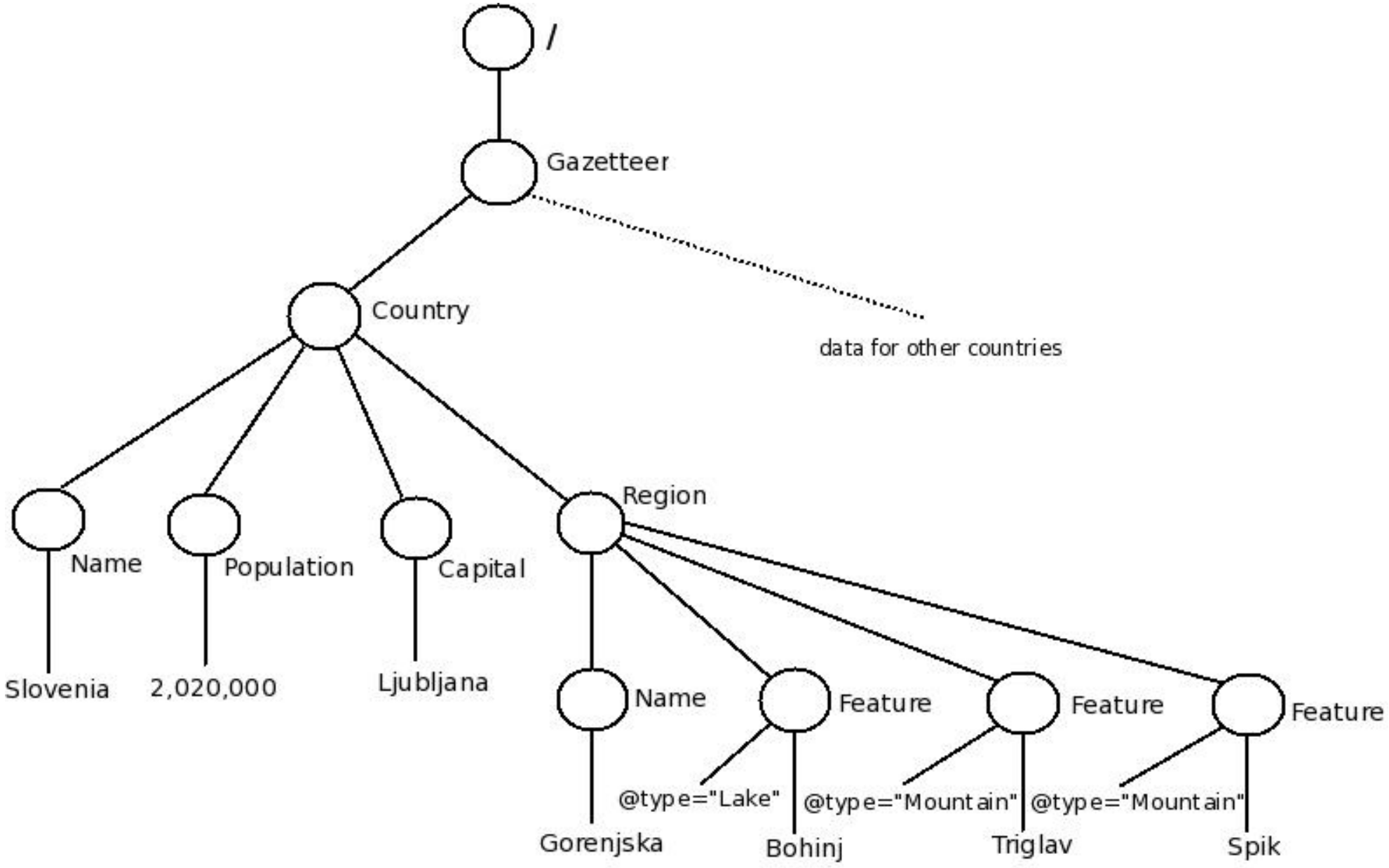
Data is incorporated into a tree structure using a *semistructured data model*.

There are several different such data models.

We shall use the *XPath data model* (chosen because its structure corresponds exactly to XML).

The next slide illustrates an example of data structured according to the XPath data model.

The chosen example, a fragment of a gazetteer, is given because it is one that is naturally accommodated within a hierarchical tree-based structure.


/
Gazetteer
Country
data for other countries
Name
Population
Capital
Region
Slovenia
2,020,000
Ljubljana
Name
Feature
Feature
Feature
@type="Lake"
@type="Mountain"
@type="Mountain"
Gorenjska
Bohinj
Triglav
Spik

Types of node in the XPath data model

Root node. This is the root of the tree. It is labelled **`/`**.

Element nodes. These are nodes labelled with *element names*, which serve the purpose of categorising the data below them. In the example, the element names are: **`Gazetteer`**, **`Country`**, **`Name`**, **`Population`**, **`Capital`**, **`Region`**, and **`Feature`**. In the XPath data model, internal nodes other than the root are always element nodes.

The root node is required to have a single element node as child, called the *root element* (since it is root in the tree of all element nodes). In the example, the root element is **`Gazetteer`**.

Text nodes. These are leaves of the tree where textual information is stored. In the example, the text strings **`"Slovenia"`**, **`"2,020,000"`**, **`"Ljubljana"`**, **`"Gorenjska"`**, **`"Triglav"`**, **`"Bohinj"`** and **`"Špik"`** appear at text nodes.

Attribute nodes

Attribute nodes are leaves of the tree in which an *attribute* associated with the parent element node is assigned a value. In the example, we use the `@` symbol to identify attributes. There is a single attribute **`type`**, it is associated with the **`Feature`** element, and it is assigned the text values **`"Lake"`** and **`"Mountain"`**.

In the XPath data model, attribute nodes are treated differently from other nodes.

Although the parent of an attribute node is an element node, when we talk about the children of this parent node, attribute nodes are not considered to be amongst them.

Since this can be confusing, explicit warnings will be given in situations in which confusion might arise.

Understanding the tree

The meaning of the data at a text node depends on the element nodes that appear along the path from the root of the tree to the leaf, and on the values of the attributes to this node.

For example, the path to **`Bohinj`** is

`/Gazetteer/Country/Region/Feature/`

and the value of the **`type`** attribute of the associated **`Feature`** element is **`"Lake"`**. This tells us that Bohinj is a feature in a region in a country in the gazetteer, and that the type of feature is a lake.

Note that to get further information (such as the name of the country, Slovenia), we need to extract it by following another path within the relevant ancestor element (in this case, the **`Country`** element).

Similarly, the meaning of an element node depends on the path to the node from the root of the tree.

For example, the element **`Name`** is used in two different ways.

A path **`/Gazetteer/Country/Name/`** leads to a text node containing the name of a country.

A path **`/Gazetteer/Country/Region/Name/`** leads to a text node containing the name of a region.

XML is a text-based language for presenting exactly the same tree-structured information as the XPath data model.

Extensible Markup Language (XML)

This is a *markup language*, that is it provides a mechanism, based on *elements* (also called *tags*), for annotating (*marking up*) ordinary text with additional information.

It was developed in the mid 1990's from the Standard General Markup Language (SGML) and Hypertext Markup Language (HTML).

XML has a simple text-based format which provides a convenient basis for making data widely available, e.g. over the web. Indeed, XML has become the *de facto* standard for publishing data on the web.

The next slide presents the gazetteer example in XML format.

The content and structure are identical to that of the tree presented earlier. Only the format is different.

```
<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="Mountain">Špik</Feature>
    </Region>
  </Country>
  <!-- data for other countries here -->
</Gazetteer>
```

XML Elements

Elements (also called *tags*) are the building blocks of XML documents.

The start of the content of an element **`elm`** is marked with the *start tag* **`<elm>`**, and the end of the content is marked with the *end tag* **`</elm>`**.

Elements must be *properly nested*. Thus,

```
<Country><Region> ... </Region></Country>
```

is legal, whereas

```
<Country><Region> ... </Country></Region>
```

is illegal.

Elements are case sensitive, so **`REGION`** would be different from **`Region`**.

The content of the **`Capital`** element

```
<Capital>Ljubljana</Capital>
```

is the text string **`"Ljubljana"`**.

The content of the **`Region`** element consists of one **`Name`** element together with three **`Feature`** elements in sequence.

The *root element* **`Gazetteer`** encloses all information in the document.

Although there are no such examples in the example document, the content of an element may be empty, e.g.,

```
<elm></elm>
```

Such *empty elements* can be abbreviated using a single hybrid tag:

```
<elm/>
```

Attributes

An element can have descriptive attributes that provide additional information about the element. For example,

```
<Feature type="Mountain"> ... </Feature>
```

sets the attribute **type** of the given **Feature** element to have value **Mountain**.

Note that attribute values are enclosed in quotation marks (either double or single quotes).

It is possible for one element to have several different attributes, with values defined in sequence within the start tag, e.g.

```
<elm attr1="value1" attr2="value2"> ... </elm>
```

Relating XML and the tree model

The existence of a root element together with the proper nesting of elements ensures that every XML document carries a tree structure in a natural way:

- Each element of the XML document corresponds to an individual element node of the tree.
- The root element of the XML document corresponds to the root element (but *not* the root node) of the tree.
- The text content of an individual XML element corresponds to a child text node of the corresponding element node in the tree.
- An attribute definition in an element's start tag corresponds to a child attribute node of the corresponding element node in the tree.

Comments and processing instructions

Comments can be inserted anywhere in an XML document. Comments start with `<!--` and end with `-->`. They can contain arbitrary text apart from the string `--`.

The full XPath data model also contains *comment nodes* which correspond to XML comments. We have do not consider such nodes in our tree model for two reasons:

1. Simplicity.
2. We have included all the types of node that should be used to store data. Comments should instead be used as aids to the interpretation of the data represented.

XML and the XPath data model also allow *processing instructions* to be included. These are beyond the scope of this course.

Unicode

An XML document is a text document written in *Unicode*.

Unicode is a universal code for "text characters", currently supporting around 100,000 different characters.

The Unicode characters contain the standard ASCII character set, but also all "characters" in human use worldwide. (The majority of the 100,000 assigned characters are Chinese!)

Each character has an assigned *code point*, which is a number between 0 and 1,114,112.

The actual digital representation of Unicode text depends on a choice of encodings of Unicode character sequences as byte streams. Common choices of encoding are: UTF-8, UTF-16, UTF-32, ISO-8859-1.

Well-formed documents

An XML document is *well-formed* if it conforms to three guidelines:

- It starts with an XML declaration. (Our example gazetteer document does not!) A suitable such declaration would be:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  ```

  This declares the XML version, and states that UTF-8 character encoding is to be used for Unicode. (Such declarations are not examinable. In Data & Analysis, we are interested in the *content* of a document not in its declaration.)

- It has a root element that contains all other elements.

- All elements are properly nested.

These are minimal requirements on a document. Often there will be other constraints we wish to impose.

Part II — Semistructured Data

Recommended reading: §§4.1–4.3 of [XWT]

§7.4.2 of [DMS]

Structuring XML

In a given XML application area, there is often an intended structure that an XML document should possess.

For example, in the **`Gazetteer`** example, we expect the various elements to respect the natural hierarchy:

- the **`Country`** elements are inside **`Gazetteer`**;
- the **`Name`** (of the country), **`Population`**, **`Capital`** and **`Region`** elements are inside **`Country`**;
- and the **`Name`** (of the region) and **`Feature`** elements are inside **`Region`**.

Moreover, the **`Feature`** elements assign a suitable value to the attribute **`type`**.

Schema languages for XML

In relational databases, a *schema* specifies the format of a relation (table).

A *schema language* for XML is a language designed for specifying the format of XML documents.

The use of a schema language has two main advantages over giving an informal specification (cf. the informal and partial specification of the **`Gazeteer`** format on the previous slide):

- It is precise.
- It can be machine checked if an XML document satisfies (*validates*) a given schema specification.

If an XML document $\boldsymbol{X}$ has the format specified by a given schema $\boldsymbol{S}$ then we say that $\boldsymbol{X}$ is *valid* with respect to $\boldsymbol{S}$.

Document Type Definitions

The *Document Type Definition (DTD)* mechanism is a basic schema language for XML.

The language is simple, commonly used, and has been an integrated feature of XML since its inception.

DTD's allow one to specify:

- The elements and entities that can appear in a document.
- What the attributes of the elements are.
- The relationship between different elements including the order of appearance and how they are nested.

We illustrate DTD's by giving an example DTD for a gazetteer format, which validates the XML document on slide II:14.

Example DTD

```
<!ELEMENT Gazetteer (Country+)>
<!ELEMENT Country (Name,Population,Capital,Region*)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Population (#PCDATA)>
<!ELEMENT Capital (#PCDATA)>
<!ELEMENT Region (Name,Feature*)>
<!ELEMENT Feature (#PCDATA)>
<!ATTLIST Feature type CDATA #REQUIRED>
```

Understanding the example DTD

```
<!ELEMENT Gazetteer (Country+)>
```

This states that the **Gazetteer** element consists of one or more **Country** elements.

```
<!ELEMENT Country (Name,Population,Capital,Region*)>
```

This states that a **Country** element consists of: one **Name** element, followed by one **Population** element, followed by one **Capital** element, followed by zero or more **Region** elements.

```
<!ELEMENT Name (#PCDATA)>
```

This states that the **Name** element contains text. The keyword **#PCDATA** abbreviates “parsed character data”.

```
<!ELEMENT Region (Name,Feature*)>
```

This states that a **Region** element consists of: one **Name**, followed by zero or more **Feature** elements.

```
<!ELEMENT Feature (#PCDATA)>
```

This states that the **Feature** element has text content.

```
<!ATTLIST Feature type CDATA #REQUIRED>
```

This states that the **Feature** element has an attribute **type**, and that the value of the attribute should be a text string (**CDATA** abbreviates "character data"). Moreover, it is *required* that every **Feature** element in the document must assign a value to the **type** attribute.

General format of element declarations

An *element declaration* has the structure:

```
<!ELEMENT elementName (contentType)>
```

There are four possible content types:

1. **EMPTY** indicating that the element has no content, i.e. it is an *empty element* as defined on slide II:16.
2. **ANY** indicating that any content is permitted.

   Nevertheless elements that appear within the element content must themselves be declared by corresponding element declarations.
3. **#PCDATA** indicating text content.

   (In fact this is an instance of a more general *mixed content* format, which we shall not consider further.)

4. A *regular expression* of element names.

Regular expressions were introduced in Inf1 Computation and Logic.

DTD's make use of the following format for regular expressions.

- Any element name is a regular expression.
  (The element names are the *alphabet* for the regular expressions.)
- ***exp1, exp2***: first ***exp1*** then ***exp2*** in sequence.
- ***exp****: zero or more occurrences of ***exp***.
- ***exp?***: zero or one occurrences of ***exp***.
- ***exp+***: one or more occurrences of ***exp***.
- ***exp1|exp2***: either ***exp1*** or ***exp2***.

General format of attribute declarations

The attributes of an element are declared separately to the element declaration. The general format is:

```
<!ATTLIST elementName (attName attType default)+>
```

This declares a list of at least one attribute for the element ***elementName***.

For each entry in the list:

- ***attName*** is the attribute name
- ***attType*** is a type for the value of the attribute.
- ***default*** specifies whether the attribute is required or optional, and may specify a default value for the attribute.

We shall consider only the following attribute types:

- *String type:* **`CDATA`** means that the attribute may have any text string as its value.
- *Enumerated type:* **`(`**s_1 **`|`** s_2 **`|...|`** s_n**`)`** means that the attribute must take one of the strings s_1, s_2, ..., s_n as its value.

And the following default options.

- *Required:* **`#REQUIRED`** means that the attribute must be explicitly assigned a value in every start tag for the element.
- *Optional:* **`#IMPLIED`** means it is optional whether a value is assigned to the attribute or not.
- *Default:* A fixed string can be specified as the default value for the attribute to take if no explicit value is given in the element's start tag.

A variation on the example

Consider replacing the attribute declaration in the example DTD with the following declaration.

```
<!ATTLIST Feature type (Mountain|Lake|River) "Mountain">
```

With this new (but not with the original) declaration:

```
<Feature>Ben Nevis</Feature>
```

would be a valid **Feature** element. The **type** attribute would be given the default (and correct) value **Mountain**.

The element below is not valid with respect to the new DTD (although it is valid for the original DTD)

```
<Feature type="Castle">Eilean Donan</Feature>
```

because **Castle** is not one of the specified values for **type**.

Document type declaration

A *document type declaration* can appear in an XML document between the XML declaration and the root element. It links the XML document to a DTD schema intended to specify the structure of the document.

The usual format of a document type declaration is:

```
<!DOCTYPE rootName SYSTEM "URI">
```

where ***rootName*** is the name of the root element, and ***URI*** is the *Uniform Resource Indicator* of the intended DTD.

An alternative (illustrated on the next slide) is to include the DTD within the XML document itself, using an *internal declaration*

```
<!DOCTYPE rootName [DTD]>
```

Example internal document type declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Gazetteer [
<!ELEMENT Gazetteer (Country+)>
<!ELEMENT Country (Name,Population,Capital,Region*)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Population (#PCDATA)>
<!ELEMENT Capital (#PCDATA)>
<!ELEMENT Region (Name,Feature*)>
<!ELEMENT Feature (#PCDATA)>
<!ATTLIST Feature type CDATA #REQUIRED>
]>
<Gazetteer>...</Gazetteer>
```

Limitations of DTD's

One of the strengths of the DTD mechanism is its essential simplicity.

However, it is inexpressive in several important ways, and this severely limits its usefulness. For example, three weaknesses are:

- Elements and attributes cannot be assigned useful types.
- It is impossible to place constraints on data values.
- There are restrictions on how character data and elements can be combined (they can only be combined as *mixed content*), and there are also undesirable technical restrictions on the forms of regular expression allowed when declaring the structure of elements.

These issues and others have been dealt with through the development of more powerful, but more complex, XML format languages, such as XML Schema (which lie beyond the scope of Data & Analysis.)

Publishing relational data as XML

A common application of XML is as a format for publishing data from relational databases.

The benefit of XML for this is that its simple text format makes the data easily readable and transferable across platforms.

The generality and flexibility of the XML format means that there are many ways to translate relational data into XML.

We illustrate one simple approach using example data from previous lectures (cf. slide I:99).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE UniversityData [
<!ELEMENT UniversityData (Students,Courses,Takes)>
<!ELEMENT Students (mn,name,age,email)*>
<!ELEMENT Courses (code,name,year)*>
<!ELEMENT Takes (mn,name,mark)*>
<!ELEMENT mn (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT code (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT mark (#PCDATA)>
]>
```

```
<UniversityData>
  <Students>
    <mn>s0456782</mn> <name>John</name>
      <age>18</age> <email>john@inf</email>
    <mn>s0412375</mn> <name>Mary</name>
      <age>18</age> <email>mary@inf</email>
    <mn>s0378435</mn> <name>Helen</name>
      <age>20</age> <email>helen@phys</email>
    <mn>s0189034</mn> <name>Peter</name>
      <age>22</age> <email>peter@math</email>
  </Students>
  <Courses>
    <code>inf1</code><name>Informatics 1</name><year>1</year>
    <code>math1</code><name>Mathematics 1</name><year>1</year>
  </Courses>
  <Takes>
    <mn>s0412375</mn><code>inf1</code><mark>80</mark>
    <mn>s0378435</mn><code>math1</code><mark>70</mark>
  </Takes>
</UniversityData>
```

Efficiency

Relational database systems are optimised for storage efficiency.

As we have seen, the XML version of relational data is extremely verbose.

Nevertheless, XML can still be stored efficiently using *data compression* (which can be optimised for XML).

Furthermore, once published XML data has been downloaded, it can be converted back to relational data so it can be stored efficiently in a local database system.

Converting XML to back to relational data has the benefit of enabling the data to be queried ising relational database technology (i.e., SQL).

An interesting alternative is to apply newer technology for directly querying XML.

Part II — Semistructured Data

Recommended reading:

§§3.1–3.4 of [XWT]

pp. 948–949 of [DMS] (superficial coverage only)

On-line XPath tutorial: **`http://www.w3schools.com/xpath/`**

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.

A better alternative is to use a dedicated language for forming queries based on the *tree structure* of an XML document

This has many uses, for example:

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

XQuery and XPath

XQuery is a powerful declarative query language for extracting information from XML documents.

However, the XQuery language is too complex for this course. (See [XWT] for further information.)

XPath is a sublanguage of XQuery, used specifically for navigating XML documents using *path expressions*.

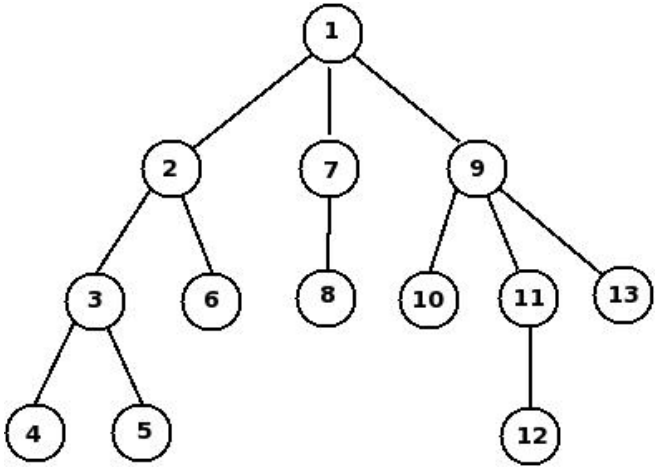
XPath can be viewed as a rudimentary query language in its own right.

It is also an important component of many XML application languages other than XQuery (e.g., XML Schema, XSLT, XLink, XPointer).

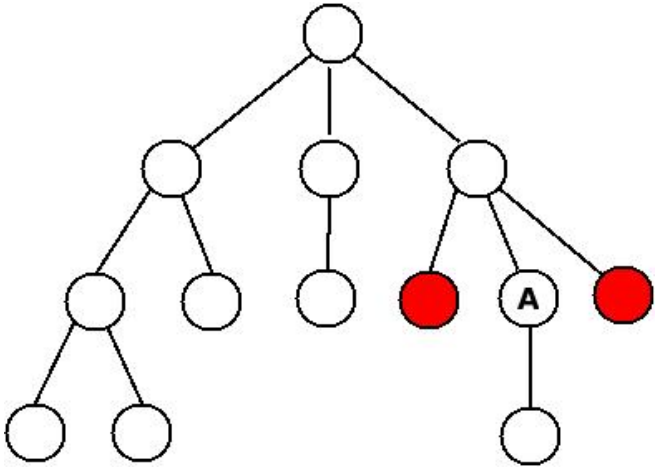
Location paths

A *location path* (a.k.a. *path expression*) retrieves a *set* of nodes from an XML document tree.

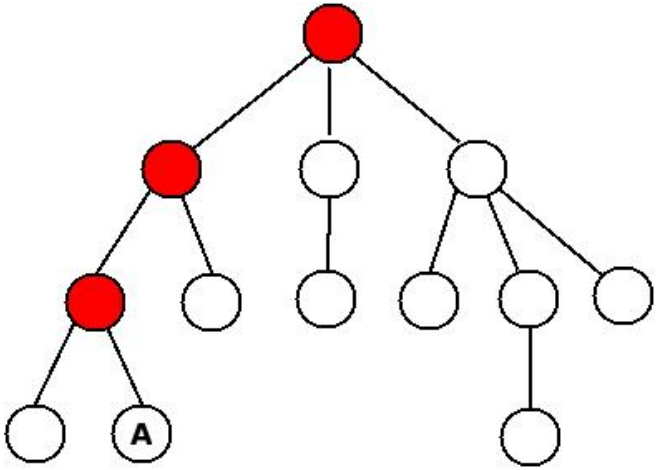
- The location path describes a set of possible paths from the root of the tree.
- The set of nodes retrieved is the set of all nodes reached as final destinations of the described paths.
- This set of nodes is returned as a list of nodes (without duplicates) sorted in *document order* (the order in which the nodes appear in the XML document)



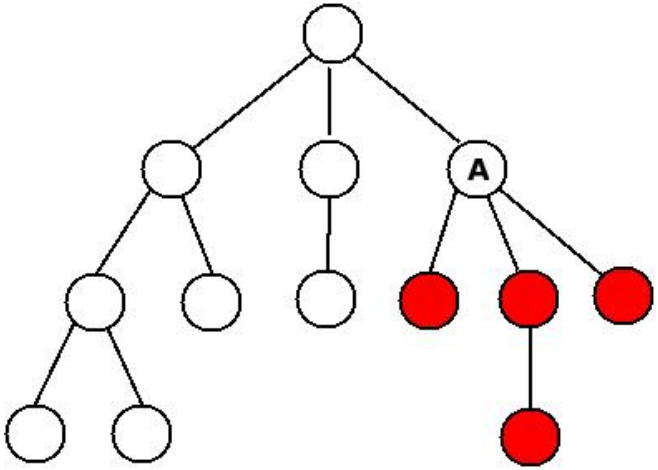
Document order



Siblings of A



Ancestors of A



Descendants of A

Example location paths

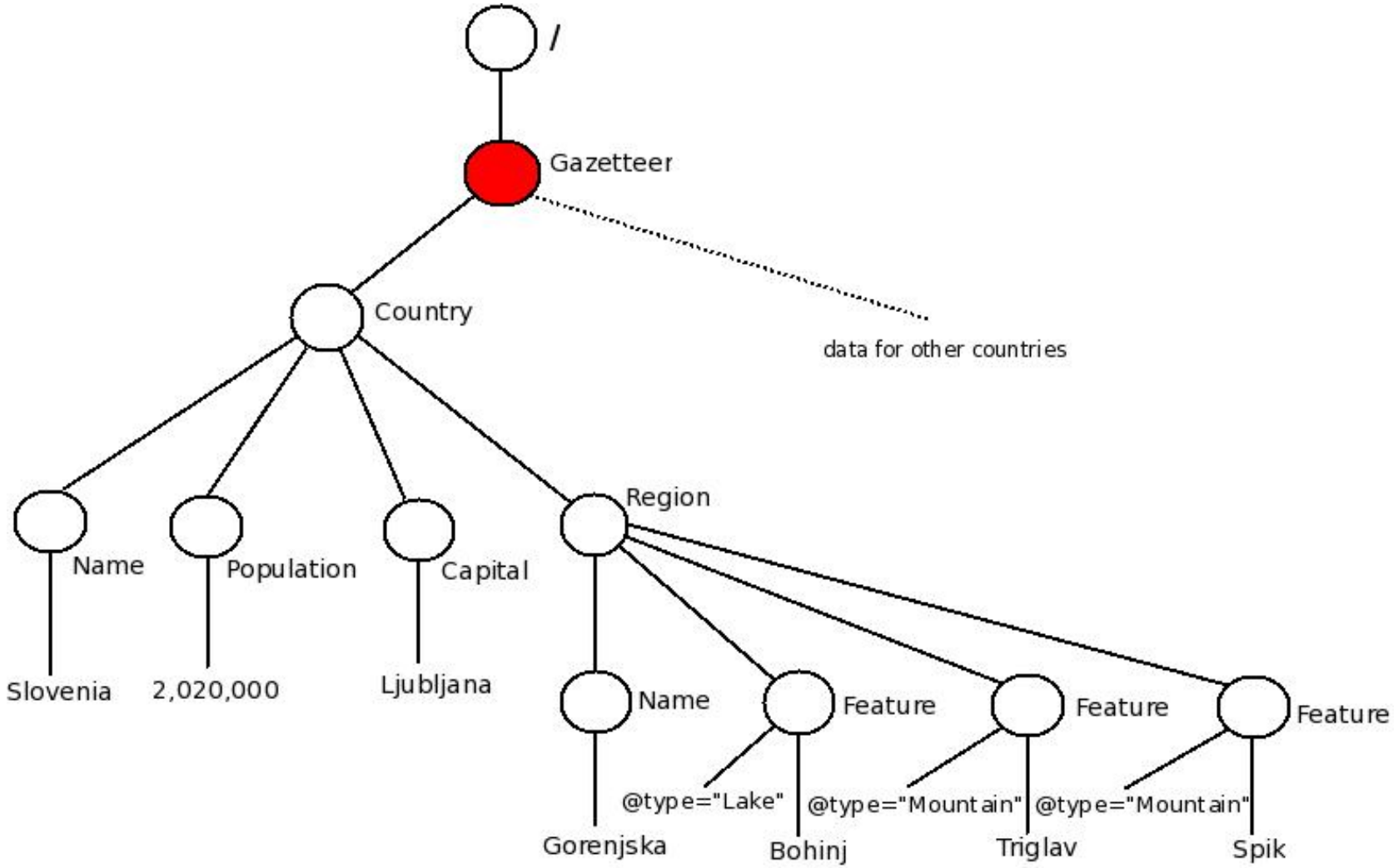
The next few slides illustrate a selection of location paths. Each is given twice: above using the full XPath syntax, and below using a convenient abbreviated syntax.

In each case, the retrieved nodes are highlighted in red. These nodes will be returned as a list in document order.

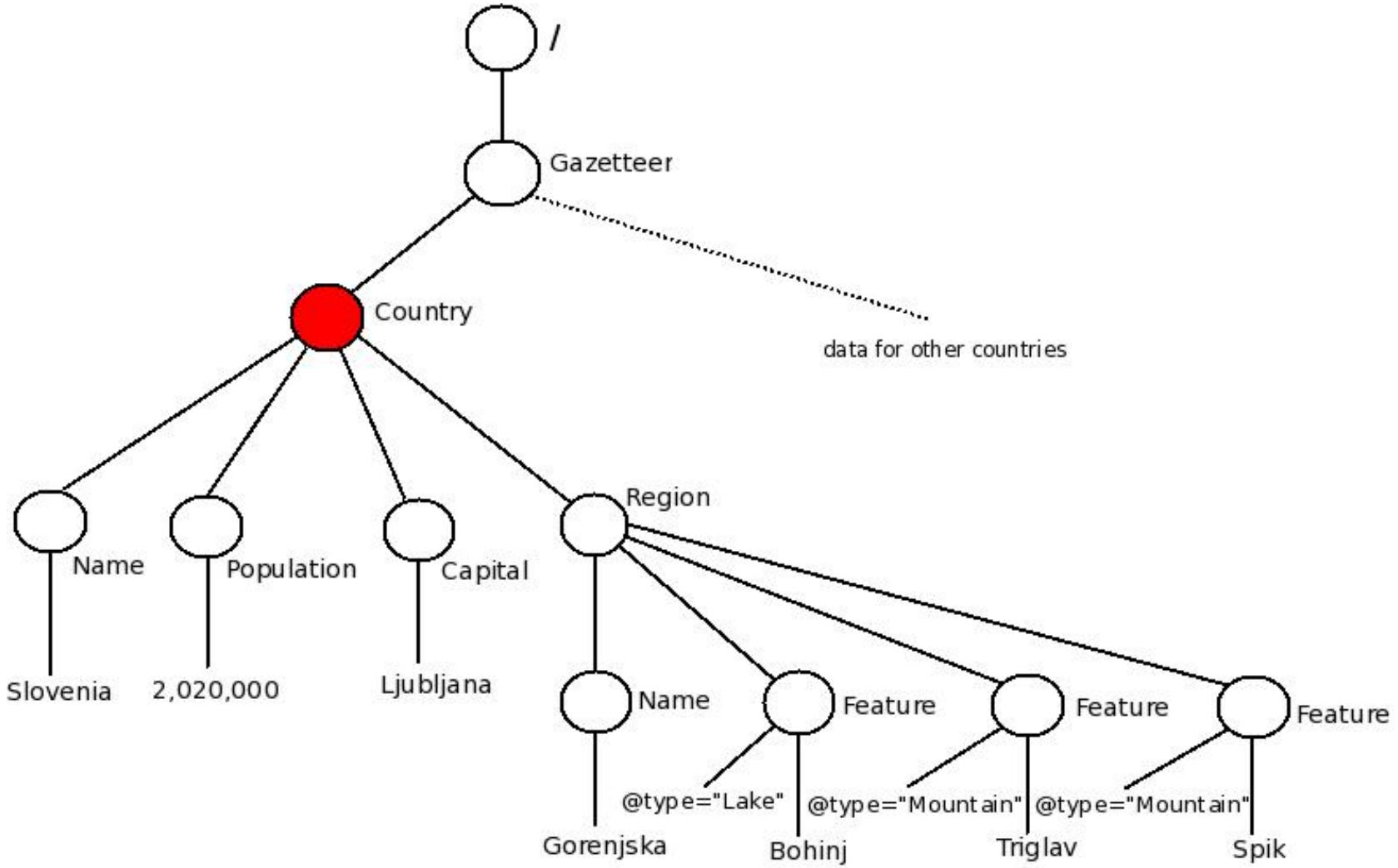
Paths are built up step-by-step as the location path is read from left-to-right.

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

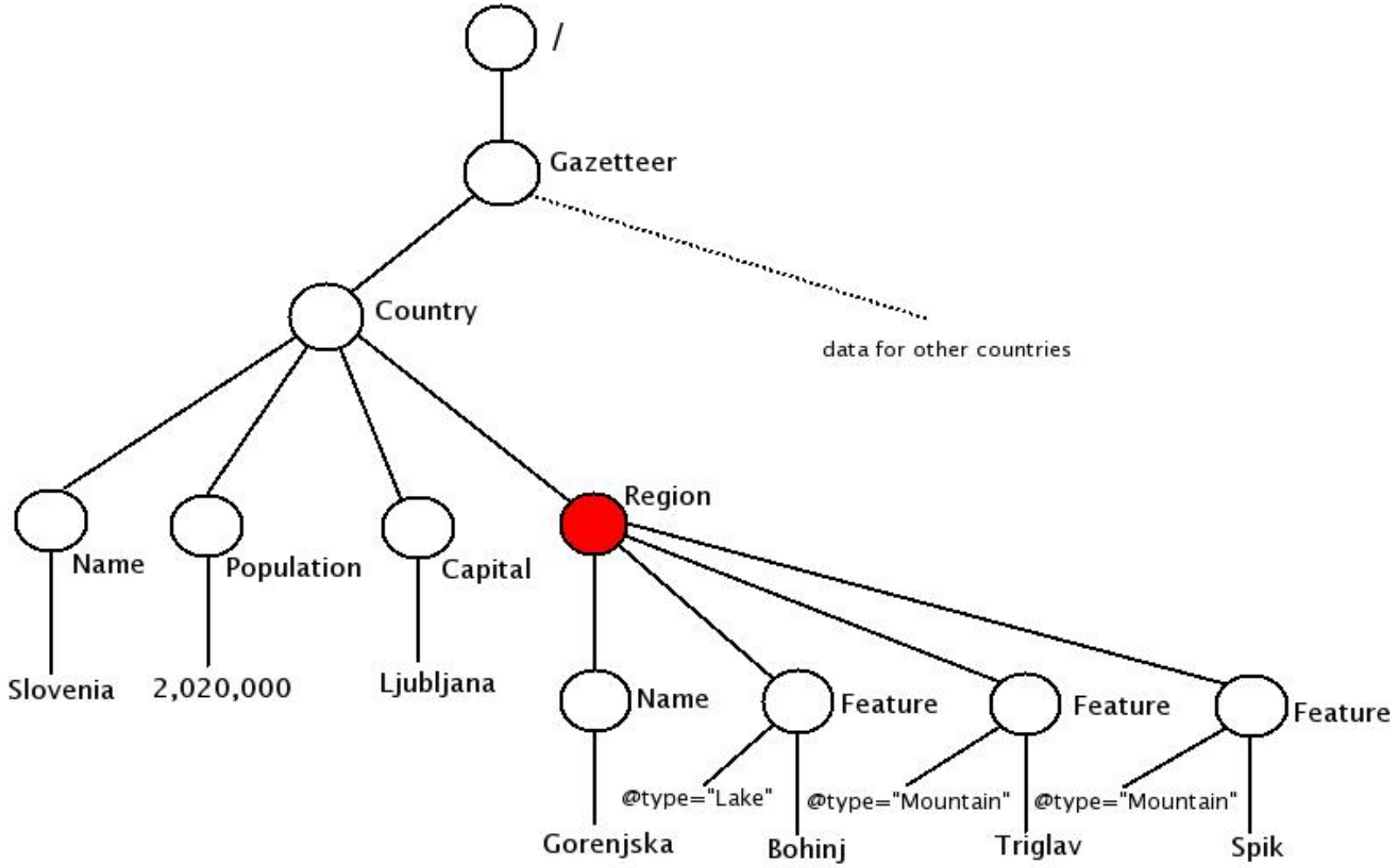
The slash `/` at the start of a location path indicates that the starting position for the context node is the root node.



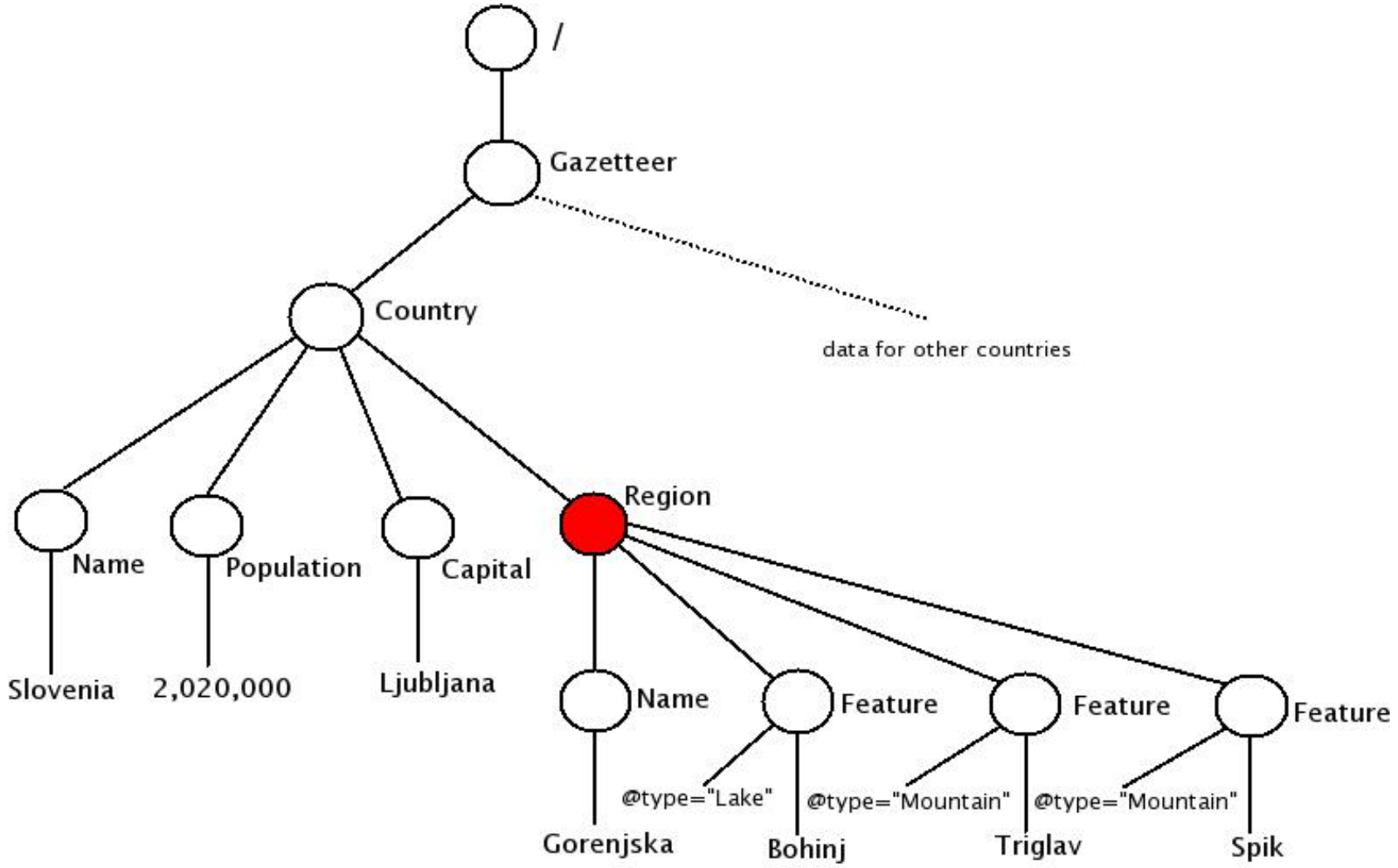

```
/child::Gazetteer
/Gazetteer
```



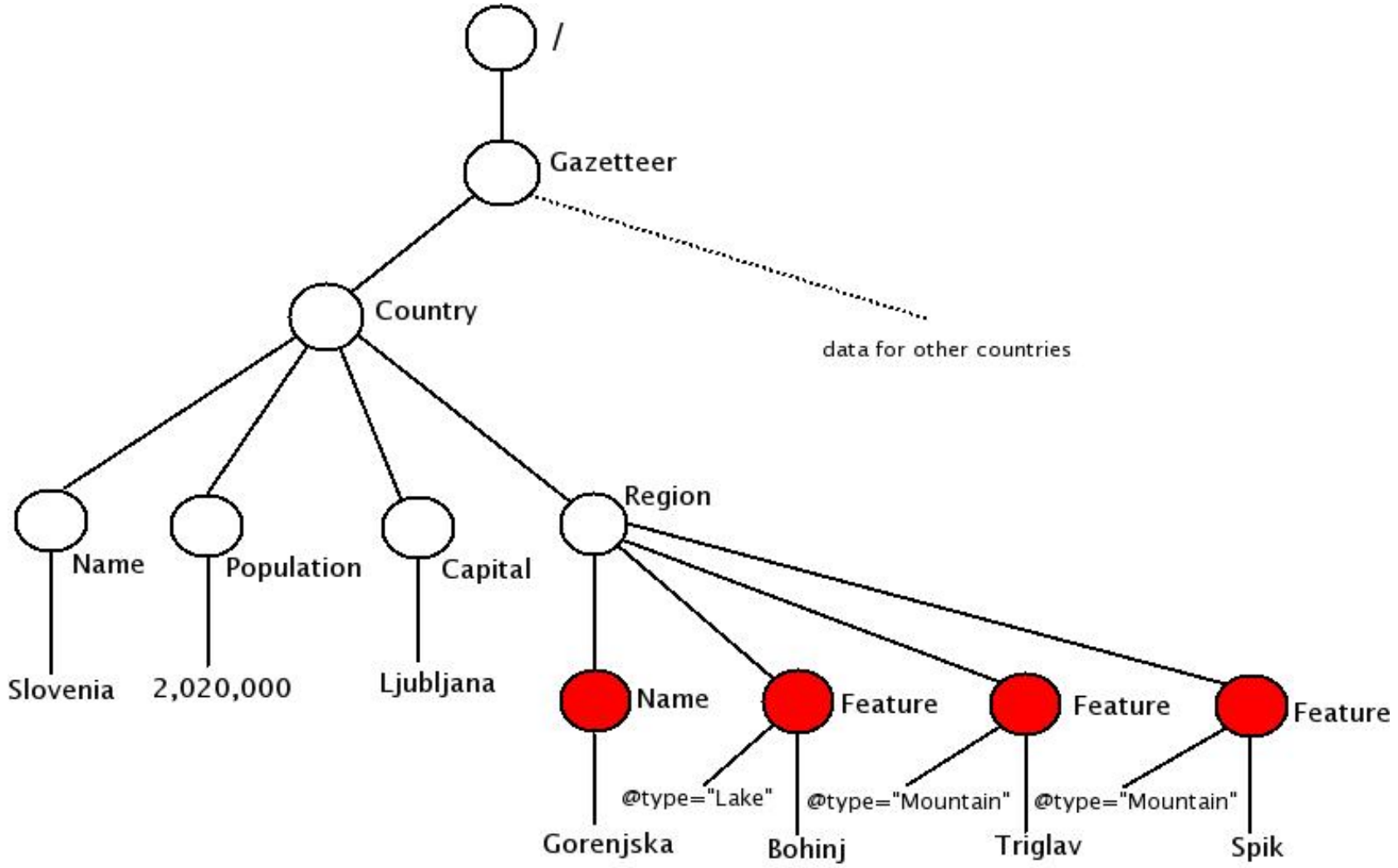

```
/child::Gazetteer/child::Country
/Gazetteer/Country
```



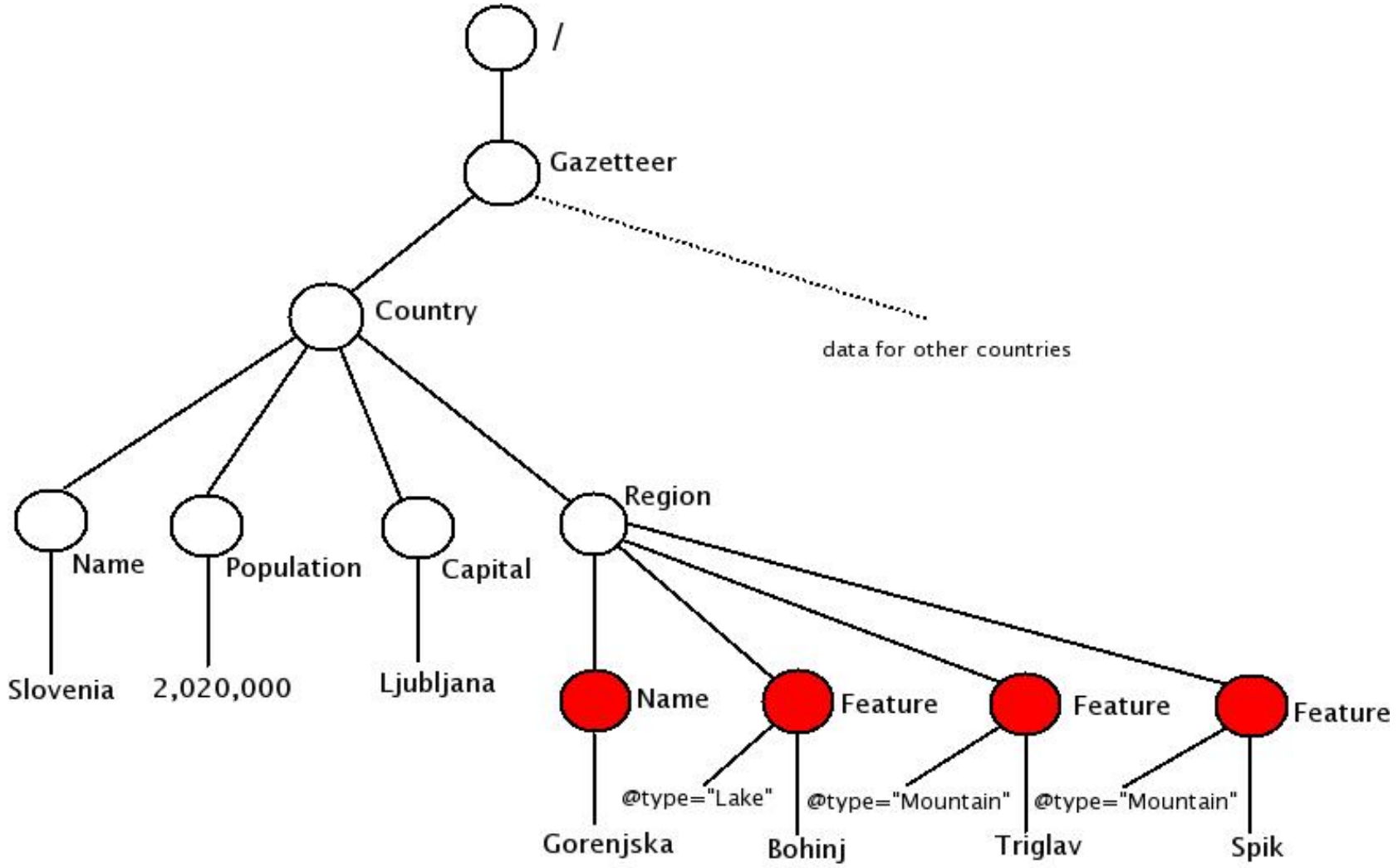

```
/child::Gazetteer/child::Country/child::Region
/Gazetteer/Country/Region
```



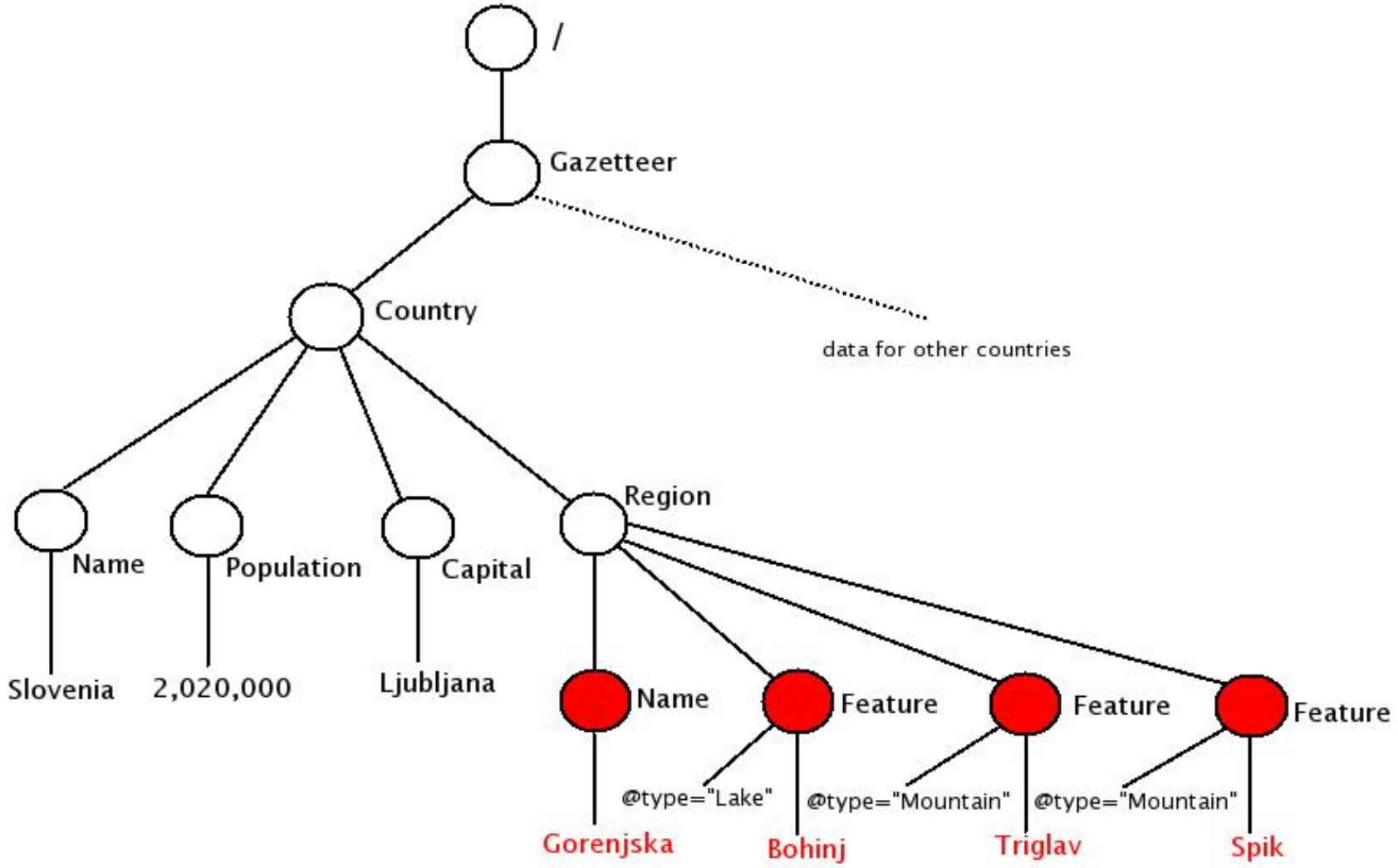

```
/descendant::Region
//Region
```



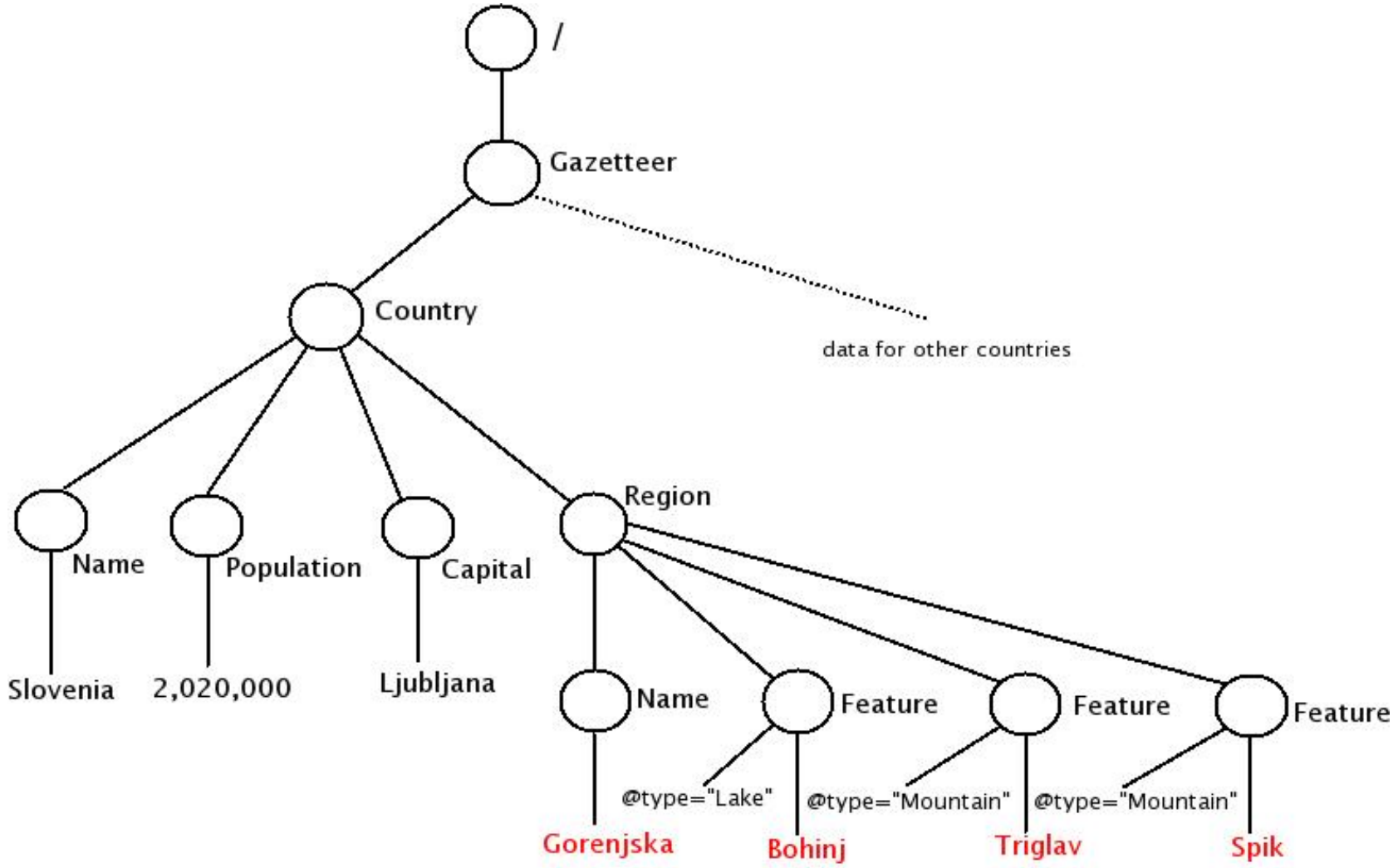

```
/descendant::Region/child::*
//Region/*
```



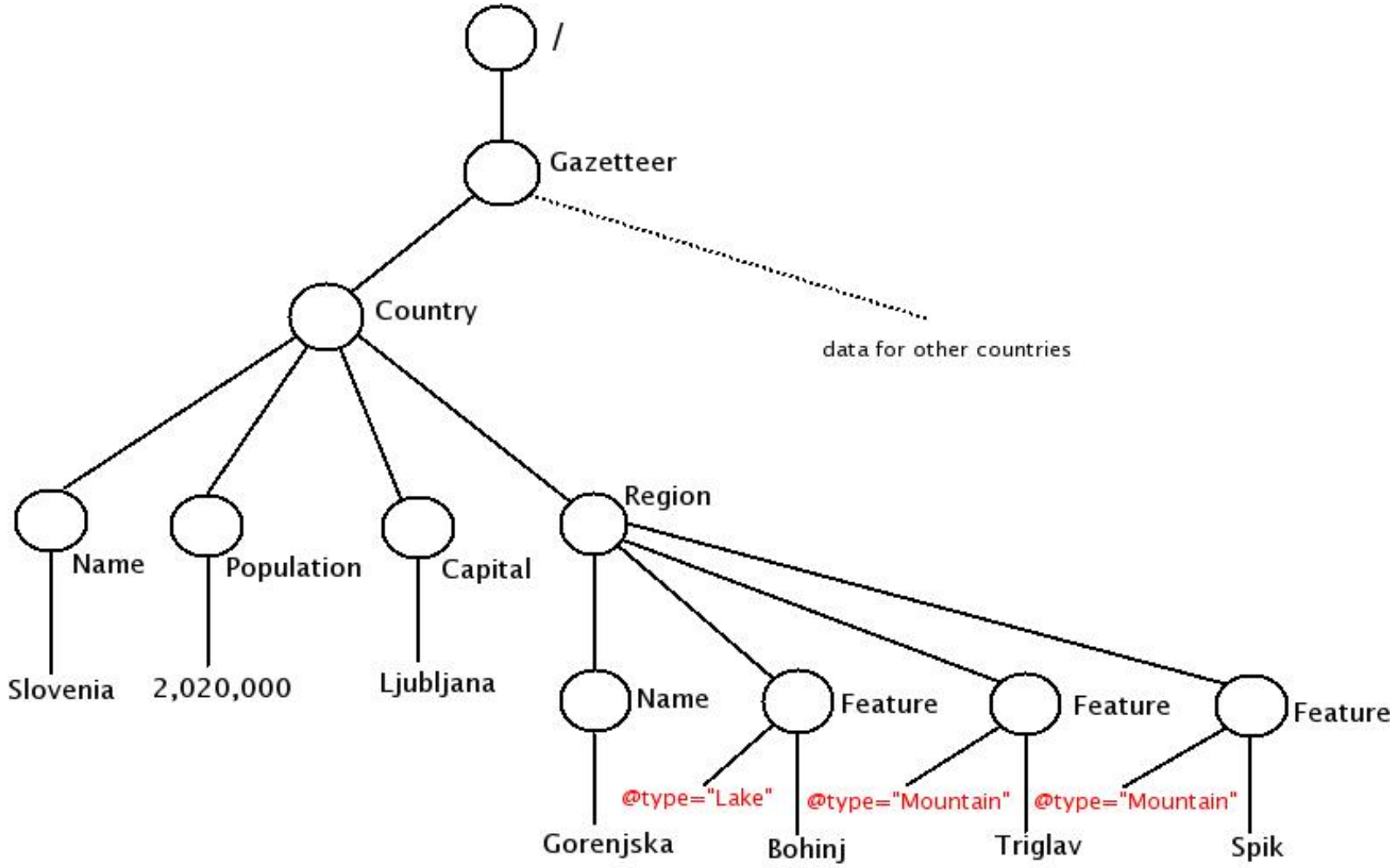

```
/descendant::Region/descendant::*
//Region//*
```




```
/descendant::Region/descendant::node()
//Region//node()
```




```
/descendant::Region/descendant::text()
//Region//text()
```




```
/descendant::Feature/attribute::type
//Feature/@type
```

General unabbreviated syntax of location paths

A *location path* is a sequence of *location steps* separated by a **/** character.

A *location step* has the form

axis*::*nodeTest predicate*

- The *axis* tells the context node which way to move.
- The *node test* selects nodes of an appropriate type from the tree.
- The optional *predicates* supply conditions that need to be satisfied for the path to be allowed to count towards the result.

N.B., the previous examples contained only axes and node tests.

A selection of axes

- `child`: the children of the context node (remember, an attribute node does not count as a child node)
- `descendant`: the descendants of the context node (again, an attribute node does not count as a descendant).
- `parent`: the unique parent of the context node (where the context node must not be the root node).
- `attribute`: all attribute nodes of the context node (which must be an element node).
- `self`: the context node itself (this is useful in connection with abbreviations).
- `descendant-or-self`: the context node together with its descendants.

A selection of node tests

Node tests filter the nodes selected by the current axis according to the type of node.

- **`text()`** : selects only character data nodes.
- **`node()`** : selects all nodes.
- **`*`** : if the axis is **`attribute`** then all attribute nodes are selected; for any other axis, all element nodes are selected.
- ***`name`*** : selects the nodes with the given name.

  The names used for node tests in the earlier examples were: **`Gazetteer`**, **`Country`**, **`Region`**, **`Feature`** and **`type`**.

Predicates

The node test in a location step may be followed by zero, one or several *predicates* each given by an expression enclosed in square brackets.

Common examples of predicates are:

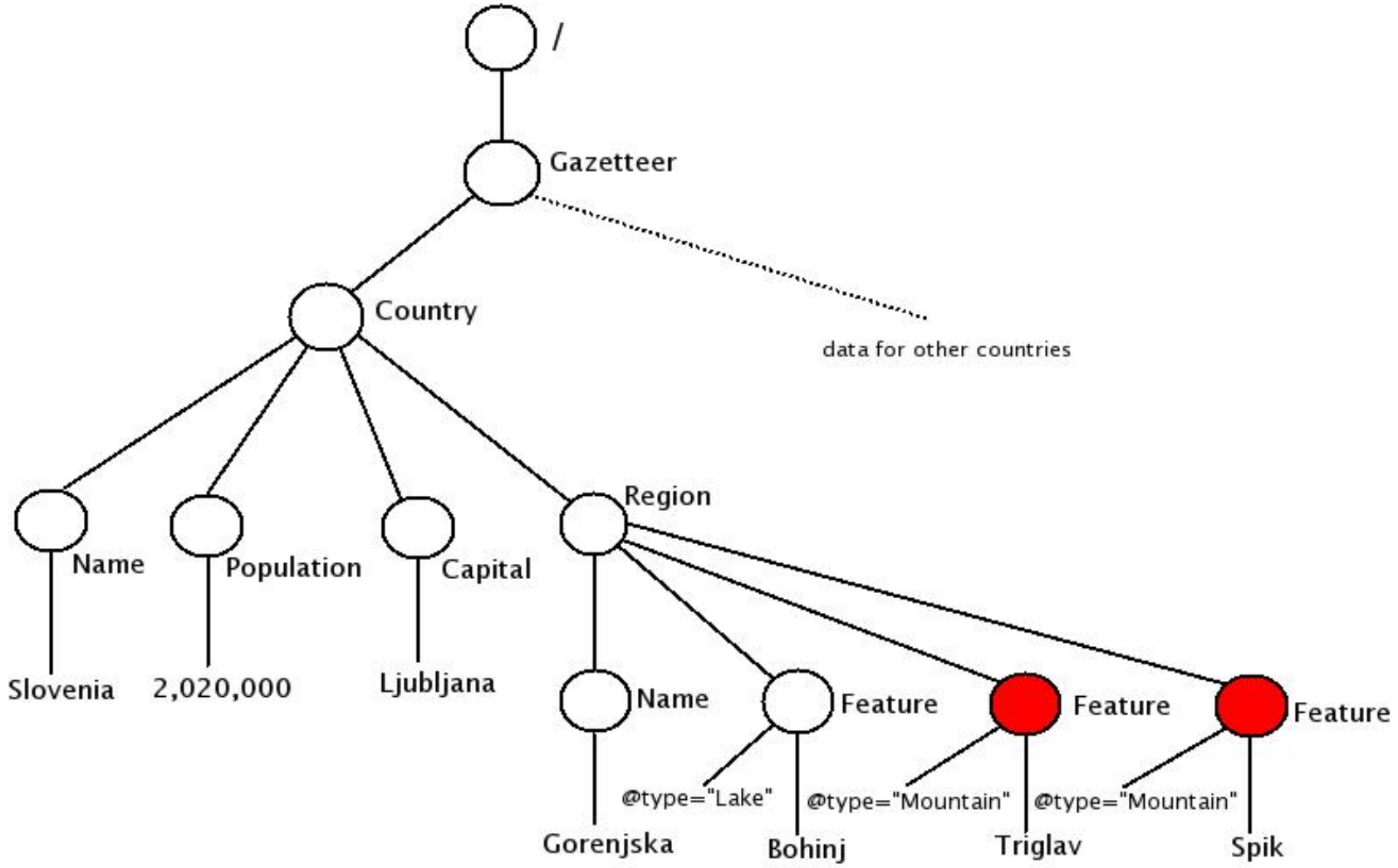
- **`[ locationPath ]`**

  This selects only those nodes for which there exists a continuation path (from the current node) matching ***`locationPath`***.

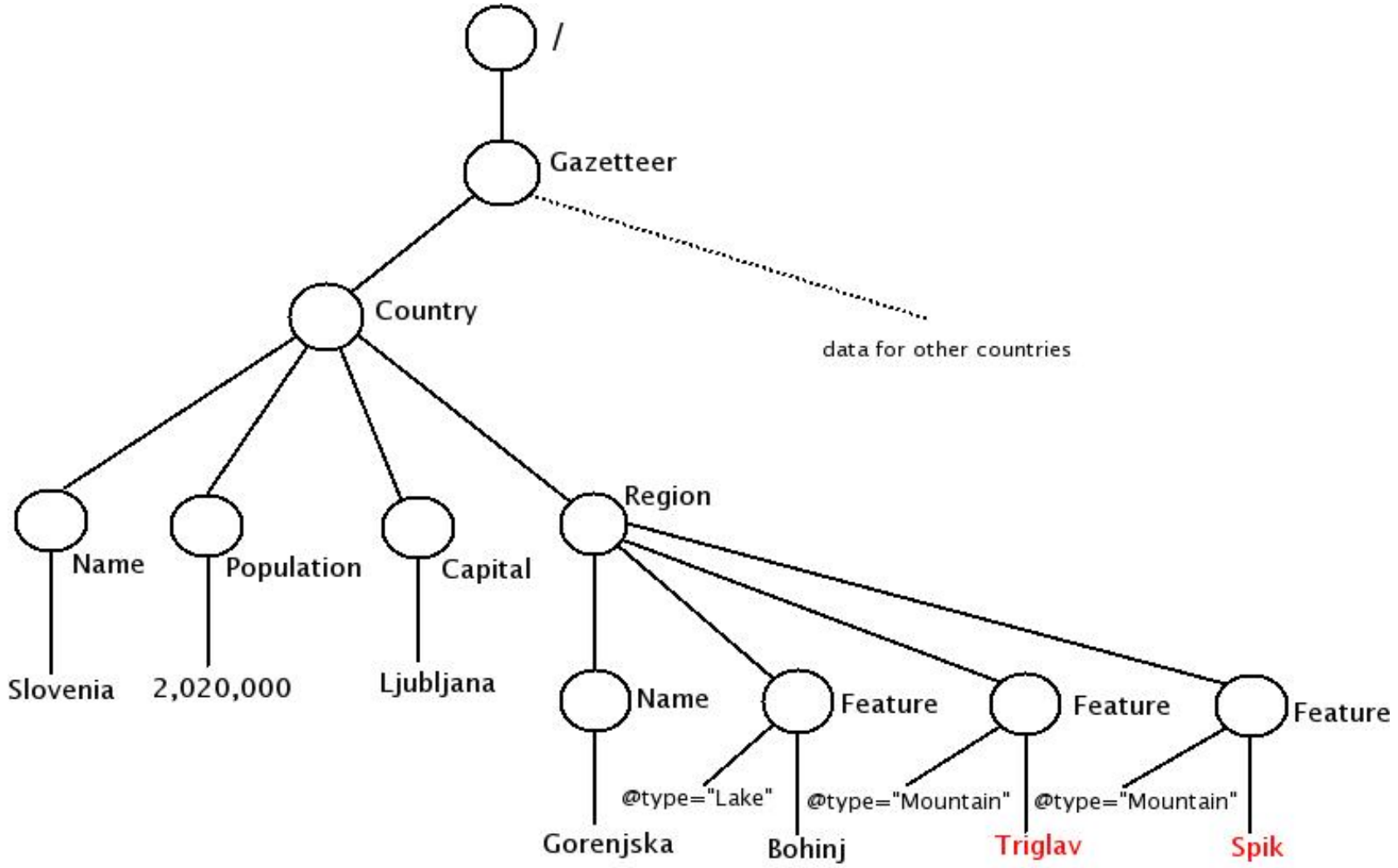
- **`[ locationPath = value ]`**

  Selects those nodes for which there exists a continuation path matching ***`locationPath`*** such that the final node of the path is equal to ***`value`***.

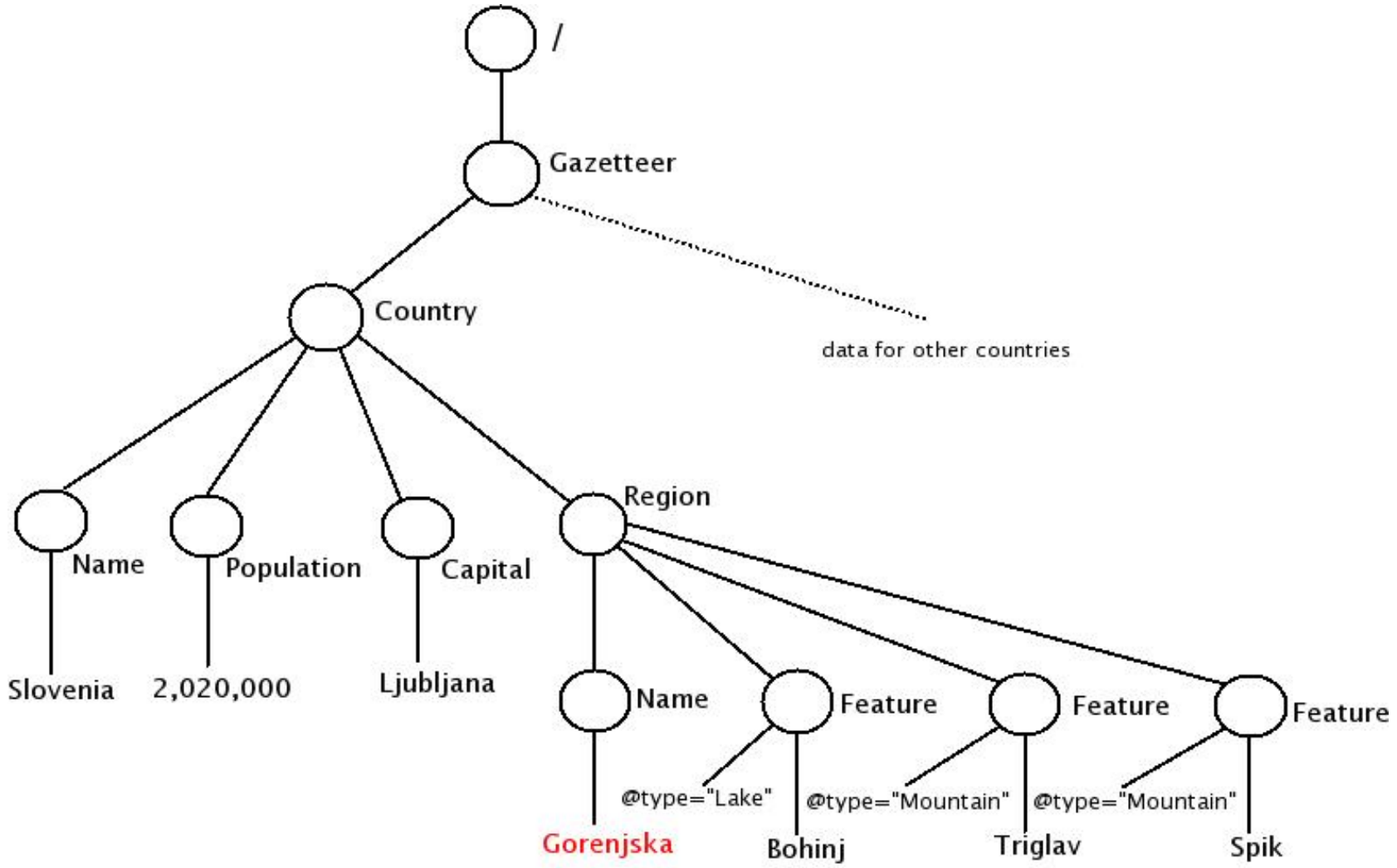
The full syntax of XPath predicate expressions is rather powerful, but beyond the scope of the course.




```
/descendant::Feature[attribute::type='Mountain']
//Feature[@type='Mountain']
```




```
/descendant::Feature[attribute::type='Mountain']/child::text()
//Feature[@type='Mountain']/text()
```




```
//Feature[@type='Mountain']/../Name/text()
```

XPath as a query language

The previous examples illustrate XPath as a rudimentary query language.

The queries formulated are:

- Slide II: 60 : Find every feature element for which the feature is a mountain.
- Slide II: 61 : Find the name of every mountain.
- Slide II: 62 : Find the name of every region in which there is a mountain.

The last query was given only in abbreviated form. The full version is more cumbersome:

```
/descendant::Feature[attribute::type='Mountain']/
  parent::*/child::Name/child::text()
```

Abbreviated syntax

The abbreviated syntax is more economical and often (but not always!) more intuitive.

The XPath abbreviations are:

- The syntax **`child::`** may be omitted from a location step altogether. (The child axis is chosen as default.)
- The syntax **`@`** is an abbreviation for: **`attribute::`**
- The syntax **`//`** is an abbreviation for:

  `/descendant-or-self::node()/`
- The syntax **`..`** is an abbreviation for: **`parent::node()`**
- The syntax **`.`** is an abbreviation for: **`self::node()`**

Queries and alternatives

Consider again the last query above:

Find the name of every region in which there is a mountain.

An alternative location path for this is:

```
//Region[Feature/@type='Mountain']/Name/text()
```

Similarly, consider:

Find the name of countries containing a feature called Everest.

Two queries for this are:

```
//Feature[text()='Everest']/../../Name/text()
```

```
//Country[.//Feature/text()='Everest']/Name/text()
```

One subtle point

A subtle point with XPath is illustrated by the second solution above to:

Find the name of countries containing a feature called Everest.

While the given query (repeated below) is correct,

```
//Country[.//Feature/text()='Everest']/Name/text()
```

the following (natural) attempt would be incorrect:

```
//Country[//Feature/text()='Everest']/Name/text()
```

The problem is that the location path `//Feature/text()` starts with a `/` character, and this means that XPath interprets this path as starting at the root node, whereas the path needs to start at the current node.

The omission of a necessary '`.`' character at the start of a predicate expression is a common source of errors in XPath.

More on XPath

In practice, when using XPath, one often needs to prefix the location path with a pointer to the given XML document; e.g.,

```
doc("gazetter.xml")//Feature[@type='Mountain']/text()
```

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

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

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

A tutorial can be found at:

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