XML and Databases

Exam Preperation
Discuss Answers to last year's exam

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2008

(1)For each of the following, **explain why it is not well-formed XML** (is a WFC or the XML grammar violated?)

---

a) <author></author><title></title>

b) <author><title></author></title>

c) <info temp='25C'>content</info>

d)
```
<!DOCTYPE greeting [
    <!ELEMENT greeting (#PCDATA)>
    <!ENTITY e1 "&e2; e3">
    <!ENTITY e2 "&e3;">
    <!ENTITY e3 "&e2;">
]>
<greeting> &e1; </greeting>
```

e) <a at1="blah" at<2="foo"> 1 < 5

f) <a b3="a" b2="b" b1="a" b2="5"/>

g) <a><c><c/></c><c/>ab&e;

(1)For each of the following, **explain why it is not well-formed XML** (is a WFC or the XML grammar violated?)

---

a) `<author></author><title></title>` → XML grammar (*cannot* be derived by grammar!)

b) `<author><title></author></title>`

c) `<info temp='25C'>content</info>`

d)
```
<!DOCTYPE greeting [
    <!ELEMENT greeting (#PCDATA)>
    <!ENTITY e1 "&e2; e3">
    <!ENTITY e2 "&e3;">
    <!ENTITY e3 "&e2;">
]>
<greeting> &e1; </greeting>
```

e) `<a at1="blah" at<2="foo"> 1 < 5 </a>`

f) `<a b3="a" b2="b" b1="a" b2="5"/>`

g) `<a><b><c><c/></c><c/>ab&e;</b></a>`

(1)For each of the following, **explain why it is not well-formed XML** (is a WFC or the XML grammar violated?)

---

a) <author></author><title></title> → XML grammar

b) <author><title></author></title> → WFC

c) <info temp='25C'>content</info>

d)
```
<!DOCTYPE greeting [
    <!ELEMENT greeting (#PCDATA)>
    <!ENTITY e1 "&e2; e3">
    <!ENTITY e2 "&e3;">
    <!ENTITY e3 "&e2;">
]>
<greeting> &e1; </greeting>
```

e) <a at1="blah" at<2="foo"> 1 < 5

f) <a b3="a" b2="b" b1="a" b2="5"/>

g) <a><c><c/></c><c/>ab&e;

(1)For each of the following, **explain why it is not well-formed XML** (is a WFC or the XML grammar violated?)

---

a) <author></author><title></title> → XML grammar

b) <author><title></author></title> → WFC

c) <info temp='25C'>content</info> → XML grammar (should be "25c"; -is actually OK..!)

d)
```
<!DOCTYPE greeting [
    <!ELEMENT greeting (#PCDATA)>
    <!ENTITY e1 "&e2; e3">
    <!ENTITY e2 "&e3;">
    <!ENTITY e3 "&e2;">
]>
<greeting> &e1; </greeting>
```

e) <a at1="blah" at<2="foo"> 1 < 5

f) <a b3="a" b2="b" b1="a" b2="5"/>

g) <a><c><c/></c><c/>ab&e;

(1)For each of the following, **explain why it is not well-formed XML** (is a WFC or the XML grammar violated?)

---

a) <author></author><title></title> → XML grammar

b) <author><title></author></title> → WFC

c) <info temp='25C'>content</info> → XML grammar (should be "25c"; -is actually OK..!)

d) <!DOCTYPE greeting [→ WFC
    <!ELEMENT greeting (#PCDATA)>
    <!ENTITY e1 "&e2; e3">
    <!ENTITY e2 "&e3;">
    <!ENTITY e3 "&e2;">
]>
<greeting> &e1; </greeting>

Well-formedness constraint: No Recursion
A parsed entity *MUST NOT* contain a recursive reference to itself, either directly or indirectly.

e) <a at1="blah" at<2="foo"> 1 < 5

f) <a b3="a" b2="b" b1="a" b2="5"/>

g) <a><c><c/></c><c/>ab&e;

(1)For each of the following, **explain why it is not well-formed XML** (is a WFC or the XML grammar violated?)

---

a) <author></author><title></title> → XML grammar

b) <author><title></author></title> → WFC

c) <info temp='25C'>content</info> → XML grammar (should be "25c"; -is actually OK..!)

d) → WFC

```
<!DOCTYPE greeting [
    <!ELEMENT greeting (#PCDATA)>
    <!ENTITY e1 "&e2; e3">
    <!ENTITY e2 "&e3;">
    <!ENTITY e3 "&e2;">
]>
<greeting> &e1; </greeting>
```

e) <a at1="blah" at<2="foo"> 1 < 5 → XML grammar

f) <a b3="a" b2="b" b1="a" b2="5"/>

g) <a><c><c/></c><c/>ab&e;

```
[41] Attribute   ::=   Name Eq AttValue
[5]       Name   ::=   (Letter | '_' | ':') (NameChar)*
[84] Letter      ::=   [a-zA-Z]
```

(1)For each of the following, **explain why it is not well-formed XML** (is a WFC or the XML grammar violated?)

---

a) `<author></author><title></title>` → XML grammar

b) `<author><title></author></title>` → WFC

c) `<info temp='25C'>content</info>` → XML grammar (should be "25c"; -is actually OK..!)

d) → WFC

```
<!DOCTYPE greeting [
    <!ELEMENT greeting (#PCDATA)>
    <!ENTITY e1 "&e2; e3">
    <!ENTITY e2 "&e3;">
    <!ENTITY e3 "&e2;">
]>
<greeting> &e1; </greeting>
```

e) `<a at1="blah" at<2="foo"> 1 < 5 </a>` → XML grammar

f) `<a b3="a" b2="b" b1="a" b2="5"/>` → WFC

g) `<a><b><c><c/></c><c/>ab&e;</b></a>`

Well-formedness constraint: Unique Att Spec
An attribute name *MUST NOT* appear more than once in the same start-tag or empty-element tag.

(1)For each of the following, **explain why it is not well-formed XML** (is a WFC or the XML grammar violated?)

---

a) `<author></author><title></title>` → XML grammar

b) `<author><title></author></title>` → WFC

c) `<info temp='25C'>content</info>` → XML grammar (should be "25c"; -is actually OK..!)

d) → WFC

```
<!DOCTYPE greeting [
    <!ELEMENT greeting (#PCDATA)>
    <!ENTITY e1 "&e2; e3">
    <!ENTITY e2 "&e3;">
    <!ENTITY e3 "&e2;">
]>
<greeting> &e1; </greeting>
```

e) `<a at1="blah" at<2="foo"> 1 < 5 </a>` → XML grammar

f) `<a b3="a" b2="b" b1="a" b2="5"/>` → WFC

g) `<a><b><c><c/></c><c/>ab&e;</b></a>` → WFC

Well-formedness constraint: Entity Declared
...the Name given in the entity reference *MUST* match that in an *entity declaration* that...

(2) Show sequences of Unicode characters for which
a) UTF-8 needs more space than UTF-16

b) UTF-16 needs more space than UTF-8
together with the corresponding UTF codes and their lengths.

c) Explain how to binary sort a sequence of UTF-8 characters.
   Use pseudo code if appropriate.

---

(2) Show sequences of Unicode characters for which
a) UTF-8 needs more space than UTF-16

b) UTF-16 needs more space than UTF-8
together with the corresponding UTF codes and their lengths.

c) Explain how to binary sort a sequence of UTF-8 characters.
   Use pseudo code if appropriate.

---

a) U+FFFF

```
   UTF-8:   11101111 10111111 10111111   = 24bits
  UTF-16:   11111111 11111111   = 16bits
```

(2) Show sequences of Unicode characters for which
a) UTF-8 needs more space than UTF-16

b) UTF-16 needs more space than UTF-8
together with the corresponding UTF codes and their lengths.

c) Explain how to binary sort a sequence of UTF-8 characters.
   Use pseudo code if appropriate.

---

a) U+FFFF

```
   UTF-8:   11101111 10111111 10111111   = 24bits
  UTF-16:   11111111 11111111   = 16bits
```

b) U+00

```
   UTF-8: 00000000   = 8bits
  UTF-16: 00000000 00000000   = 16bits
```

(2) Show sequences of Unicode characters for which
a) UTF-8 needs more space than UTF-16

b) UTF-16 needs more space than UTF-8
together with the corresponding UTF codes and their lengths.

c) Explain how to binary sort a sequence of UTF-8 characters.
   Use pseudo code if appropriate.

---

a) U+FFFF

```
   UTF-8:   11101111 10111111 10111111   = 24bits
  UTF-16:   11111111 11111111   = 16bits
```

b) U+00

```
   UTF-8: 00000000   = 8bits
  UTF-16: 00000000 00000000   = 16bits
```

c)

To binary compare two characters, simply start from the highest bit!
In this way, for characters with different lengths in UTF-8, after
$\leq$ 4 bits we will be done!

- In case UTF-8 lengths are same → normal binary compare..

(3) Show an element node with mixed content, using the XML Information Set.
Assume that for a node M, Type(M) is it’s type, i.e.,
is one of DOC, ELEM, ATTR, or CHAR.
Using the Infoset, show pseudo code that, given a node N,

a) returns all ancestors of the node

b) returns the previous sibling of the node.

---

(3) Show an element node with mixed content, using the XML Information Set.
Assume that for a node M, Type(M) is it's type, i.e.,
is one of DOC, ELEM, ATTR, or CHAR.
Using the Infoset, show pseudo code that, given a node N,

a) returns all ancestors of the node

b) returns the previous sibling of the node.

```
localname(e1)  = "elem"
children(e1)   = [e2,c1]
localname(e2)  = "elem"
children(e2)   = []
parent(e2)     = e1
code(c1)       = U+00
parent(c1)     = e1
attributes(e1) = []
attributes(e2) = []
```

(3) Show an element node with mixed content, using the XML Information Set.
Assume that for a node M, Type(M) is it's type, i.e.,
is one of DOC, ELEM, ATTR, or CHAR.
Using the Infoset, show pseudo code that, given a node N,

a) returns all ancestors of the node

b) returns the previous sibling of the node.

```
localname(e1)  = "elem"
children(e1)   = [e2,c1]
localname(e2)  = "elem"
children(e2)   = []
parent(e2)     = e1
code(c1)       = U+00
parent(c1)     = e1
attributes(e1) = []
attributes(e2) = []
```

a)
```
getAncestors(Node n): NodeSet
{
  NodeSet result=NULL;

  if(n.type!=DOC){
    for(; n=n->parent ; n.type!="DOC") Add(n,result);
    Add(n,result);
  }
  return result;
}
```

(3) Show an element node with mixed content, using the XML Information Set.
Assume that for a node M, Type(M) is it's type, i.e.,
is one of DOC, ELEM, ATTR, or CHAR.
Using the Infoset, show pseudo code that, given a node N,

a) returns all ancestors of the node

b) returns the previous sibling of the node.

```
localname(e1)  = "elem"
children(e1)   = [e2,c1]
localname(e2)  = "elem"
children(e2)   = []
parent(e2)     = e1
code(c1)       = U+00
parent(c1)     = e1
attributes(e1) = []
attributes(e2) = []
```

```
a)  getAncestors(Node n): NodeSet
    {
      NodeSet result=NULL;

      if(n.type!=DOC){
        for(; n=n->parent ; n.type!="DOC") Add(n,result);
        Add(n,result);
      }
      return result;
    }
```

```
b)  getPrevSib(Node n): Node
    {
      NodeList l=NULL;
      if(n.type!="DOC")
      {
        Node parent=n->parentNode();
        l=n->children;
        if(l==NULL) return NULL;
        s=first(l);
        if(s==l) return NULL;
        while(s->next!=n)
          s=s->next();
      }
      return s;
    }
```

(4) Using DOM, give pseudo code that determines the average depth of the XML tree. The average depth of <a/> is 1.

---

(4) Using DOM, give pseudo code that determines the average depth of the XML tree. The average depth of <a/> is 1.

```
int total=0;
int count=0;

call calcAverage(root, 1);

return total/count;

void calcAverage(Node n, int depth)
{
  NodeList children = n->childList();
  if(children->isEmpty())
  {
    total += depth;
    count++;
    return;
  }
  else for each Node c in children calcAverage(c, depth+1);
}
```

(5) Explain in detail, using an example, why hashing is useful for finding the minimal DAG of a tree.
Why are updates more expensive on a DAG than on a tree?
Give an example that clearly explains this.

---

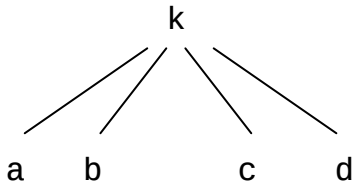
(5) Explain in detail, using an example, why hashing is useful for finding the minimal DAG of a tree.
Why are updates more expensive on a DAG than on a tree?
Give an example that clearly explains this.

When computing the minimal DAG, we need to determine whether a given subtree has occurred already. If we keep a table of pointers to the subtrees that have occurred already, then to check for a given subtree whether or not it occurs in the table takes in the worst case
(#of trees in the table) x (# nodes in subtree)
which, in the worst case, is quadratic to the size of the input tree!

With hashing, we only need
(# trees in the hash bucket) x (# nodes in the subtree).

Thus, if a bucket has only a constant number of trees, on average, then the complexity goes from quadratic to linear!

Example tree:

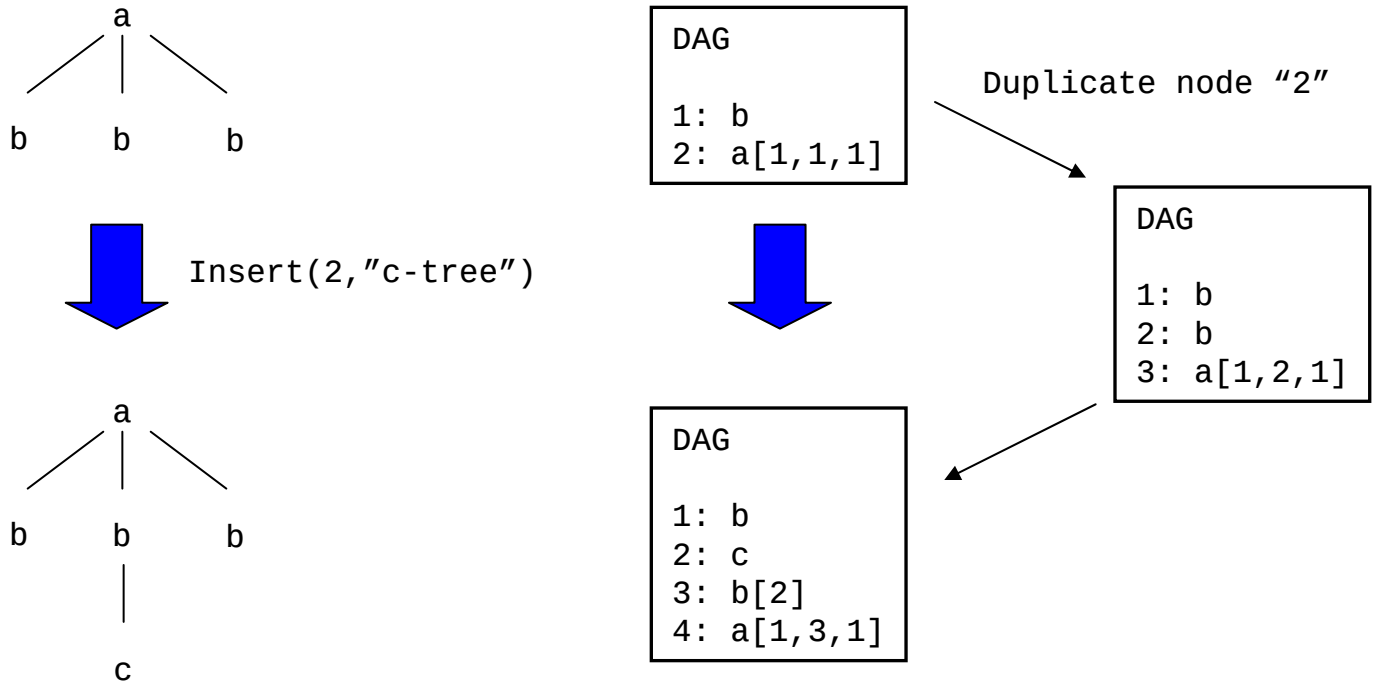



```
Seen:=NULL
Seen:=seen + ”a-tree”
Contains(Seen, “b-tree”)?   ← needs 1 comparison
Seen:=Seen + “b-tree”
Contains(Seen, “c-tree”)?   ← needs 2 comparisons
Seen:=Seen + “c-tree”
Contains(Seen, “d-tree”)?   ← needs 3 comparisons
Seen:=Seen + “d-tree”
```

```
Assume
hash(“a-tree”)=1
hash(“b-tree”)=2
hash(“c-tree”)=3
hash(“d-tree”)=4

Then we need no
(tree) comparisons
whatsover!
```

(5) Explain in detail, using an example, why hashing is useful for finding the minimal DAG of a tree.
Why are **updates** more expensive on a DAG than on a tree?
Give an example that clearly explains this.



Inserting a single new child required adding two rows and changing one existing one.

(the shared 2nd child "b-subtree" of the a-node must be duplicated first, before the c-child can be added.

(6) Give the PRE/POST table for the tree
<a><c/><c><d/><d></d></c><d/><c><d/></c><d/>

b) Give pseudo code that computes the POST order of a tree in an iterative way, i.e., without any recursive calls(!). You can use firstChild(n), nextSibling(n), and parent(n) for a node n.

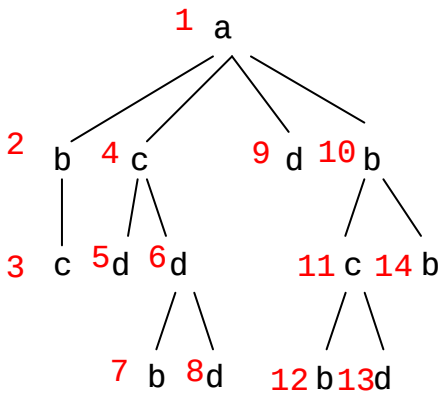
Using the PRE/POST-encoding, explain how to obtain
c) the ancestors of a node
d) the last child of a node
e) the maximal depth of the subtree at a node.

---

(6) Give the PRE/POST table for the tree
<a><c/><c><d/><d></d></c><d/><c><d/></c><d/>

b) Give pseudo code that computes the POST order of a tree in an iterative way, i.e., without any recursive calls(!). You can use firstChild(n), nextSibling(n), and parent(n) for a node n.

Using the PRE/POST-encoding, explain how to obtain
c) the ancestors of a node
d) the last child of a node
e) the maximal depth of the subtree at a node.



PRE	POST	label
1	14	a
2	2	b
3	1	c
4	7	c
5	3	d
6	6	d
7	4	b
8	5	b
9	8	d
10	13	b
11	11	c
12	12	b
13	10	d
14	12	d

(6) Give the PRE/POST table for the tree
<a><c/><c><d/><d></d></c><d/><c><d/></c><d/>

b) Give pseudo code that computes the POST order of a tree in an iterative way, i.e., without any recursive calls(!). You can use firstChild(n), nextSibling(n), and parent(n) for a node n.

Using the PRE/POST-encoding, explain how to obtain
c) the ancestors of a node
d) the last child of a node
e) the maximal depth of the subtree at a node.

```
b)  int i=1;
    Node n=root;
    repeat{
      while(firstChild(n)!=NULL) n=firstChild(n);
      post(i)=n;
      i++;
      while(nextSibling(n)==NIL){
        n=parent(n);
        if(n==NULL) break;
        post(i)=n;
        i++;
      }
      n=nextSibling(n);
     }
```

(6) Give the PRE/POST table for the tree
<a><c/><c><d/><d></d></c><d/><c><d/></c><d/>

b) Give pseudo code that computes the POST order of a tree in an iterative way, i.e., without any recursive calls(!). You can use firstChild(n), nextSibling(n), and parent(n) for a node n.

Using the PRE/POST-encoding, explain how to obtain
c) the ancestors of a node
d) the last child of a node
e) the maximal depth of the subtree at a node.

---

c) Given (pre, post) of a node, its **ancestors** are all nodes with pre-value < pre and post-value > post.

(6) Give the PRE/POST table for the tree
<a><c/><c><d/><d></d></c><d/><c><d/></c><d/>

b) Give pseudo code that computes the POST order of a tree in an iterative way, i.e., without any recursive calls(!). You can use firstChild(n), nextSibling(n), and parent(n) for a node n.

Using the PRE/POST-encoding, explain how to obtain
c) the ancestors of a node
d) the last child of a node
e) the maximal depth of the subtree at a node.

---

c) Given (pre, post) of a node, its
ancestors are all nodes with pre-value < pre
and post-value > post.

d) If there ia a node with pre-value > pre and with
post-value=post-1, then that is the last child of (pre, post)

(6) Give the PRE/POST table for the tree
<a><c/><c><d/><d></d></c><d/><c><d/></c><d/>

b) Give pseudo code that computes the POST order of a tree in an iterative way, i.e., without any recursive calls(!). You can use firstChild(n), nextSibling(n), and parent(n) for a node n.

Using the PRE/POST-encoding, explain how to obtain
c) the ancestors of a node
d) the last child of a node
e) the maximal depth of the subtree at a node.

---

c) Given (pre, post) of a node, its
ancestors are all nodes with pre-value < pre
and post-value > post.

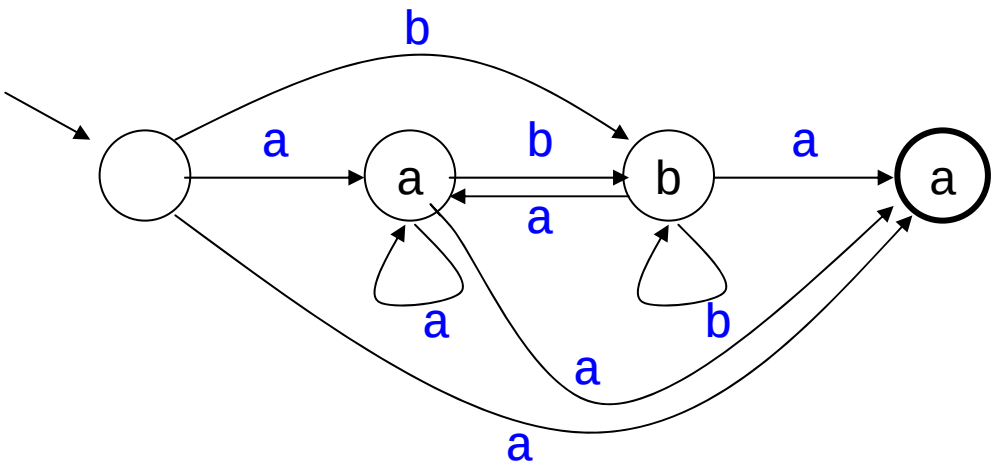
d) If there ia a node with pre-value > pre and with
post-value=post-1, then that is the last child of (pre, post)

e)
```
int maxDepth(int pr){
size(int p): int{
  int s=0;
  for(int pr2=p+1; post(pr2)<post(p); pr2++) s++
  return s;
}
int D=0; int u,L;
L=pr+size(pr)-post(pr);
for(int pr2=pr+1; post(pr2)<post(pr); pr2++){
  u=pr2+size(pr2)-post(pr2)-L;
  if(u>D) D=u;
return D;
}
```

(8) Show the Glushkov automaton for the regular expression E=(a | b)*a.
Is this expression 1-unambiguous? Explain!
Give a deterministic automaton for the same expression.
Is E2=(b*a(a|b))*a equivalent to E? Is it 1-unambiguous?
Show a 1-unambiguous expression that is equivalent to a(a | b)*.

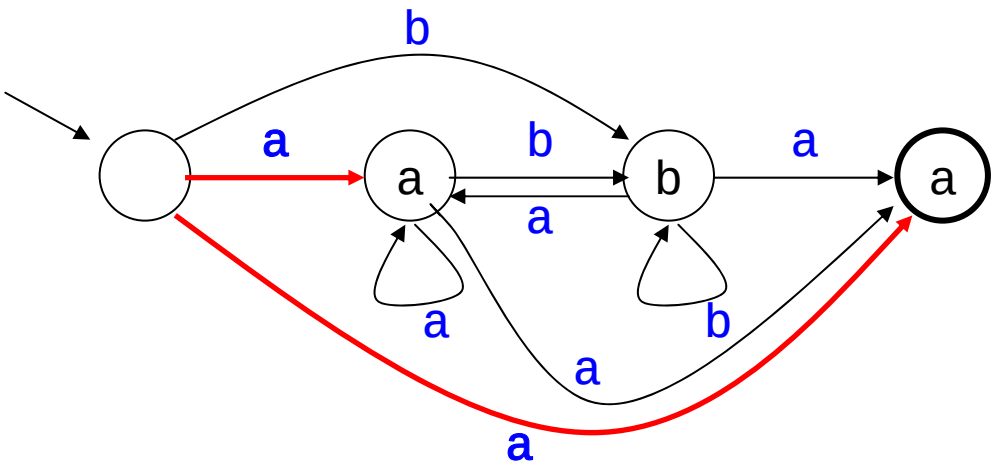
---

(8) Show the Glushkov automaton for the regular expression E=(a | b)*a.
Is this expression 1-unambiguous? Explain!
Give a deterministic automaton for the same expression.
Is E2=(b*a(a|b))*a equivalent to E? Is it 1-unambiguous?
Show a 1-unambiguous expression that is equivalent to a(a | b)*.



Deterministic??

(8) Show the Glushkov automaton for the regular expression E=(a | b)*a.
Is this expression 1-unambiguous? Explain!
Give a deterministic automaton for the same expression.
Is E2=(b*a(a|b))*a equivalent to E? Is it 1-unambiguous?
Show a 1-unambiguous expression that is equivalent to a(a | b)*.

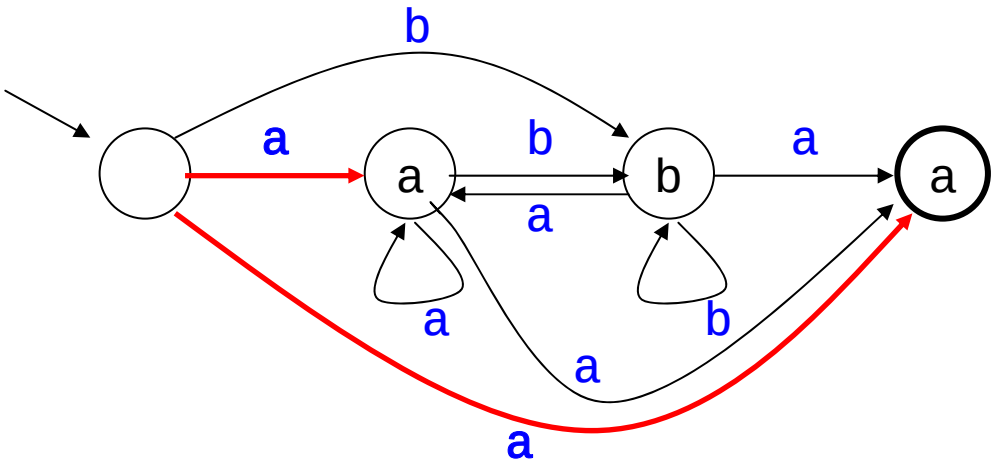


Deterministic??

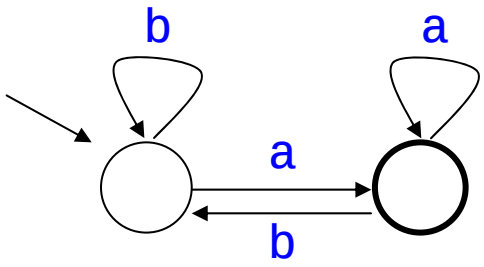
→ no!

Thus, E is not 1-unambiguous.

(8) Show the Glushkov automaton for the regular expression E=(a | b)*a.
Is this expression 1-unambiguous? Explain!
Give a deterministic automaton for the same expression.
Is E2=(b*a(a|b))*a equivalent to E? Is it 1-unambiguous?
Show a 1-unambiguous expression that is equivalent to a(a | b)*.



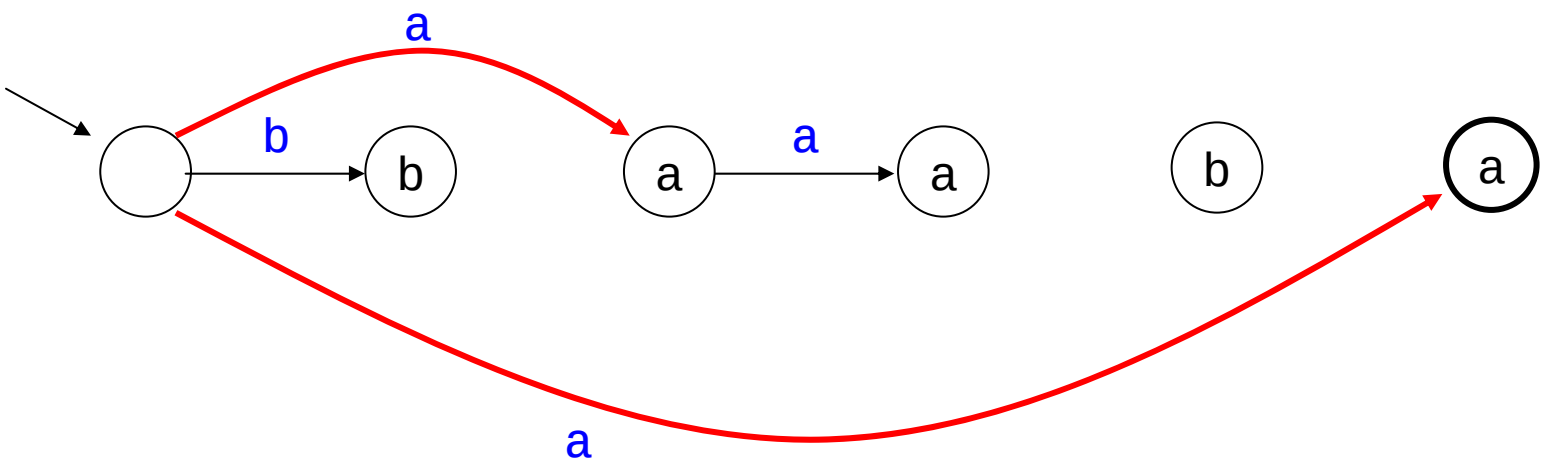
Deterministic automaton:



(8) Show the Glushkov automaton for the regular expression E=(a | b)*a.
Is this expression 1-unambiguous? Explain!
Give a deterministic automaton for the same expression.
Is E2=(b*a(a|b))*a equivalent to E? Is it 1-unambiguous?
Show a 1-unambiguous expression that is equivalent to a(a | b)*.

NOT equivalent to E!
The string "ba" is matched by E, but **NOT** by E2.

E2 is NOT 1-unambiguous:



(8) Show the Glushkov automaton for the regular expression E=(a | b)*a.
Is this expression 1-unambiguous? Explain!
Give a deterministic automaton for the same expression.
Is E2=(b*a(a|b))*a equivalent to E? Is it 1-unambiguous?
Show a 1-unambiguous expression that is equivalent to a(a | b)*.

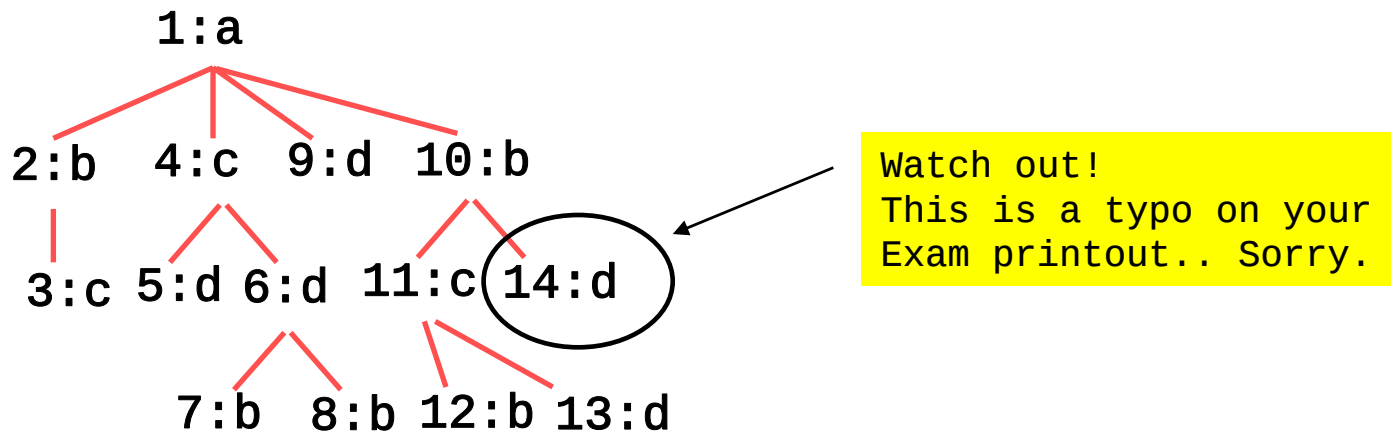
The expression b*a(b*a)*
is

→ equivalent to E
→ 1-unambiguous.

(9) For the tree given in 6, write XPath expressions that

a) select all b nodes
b) select all b nodes that have a c-child
c) select all b nodes that have no c-children
d) select the right most c-node
e) select all nodes that have a c-parent

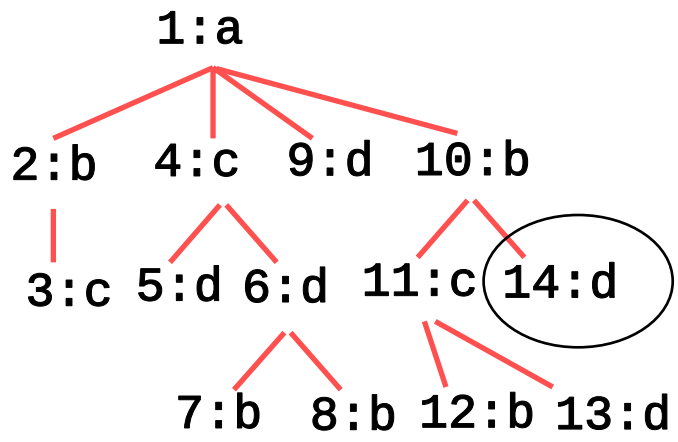
```
<a><b><c/></b><c><d/><d><b/><b/></d></c><d/><b><c><b/><d/></c><d/>
</b></a>
```



(9) For the tree given in 6, write XPath expressions that

a) **select all b nodes**
b) select all b nodes that have a c-child
c) select all b nodes that have no c-children
d) select the right most c-node
e) select all nodes that have a c-parent

<a><c/><c><d/><d></d></c><d/><c><d/></c><d/>

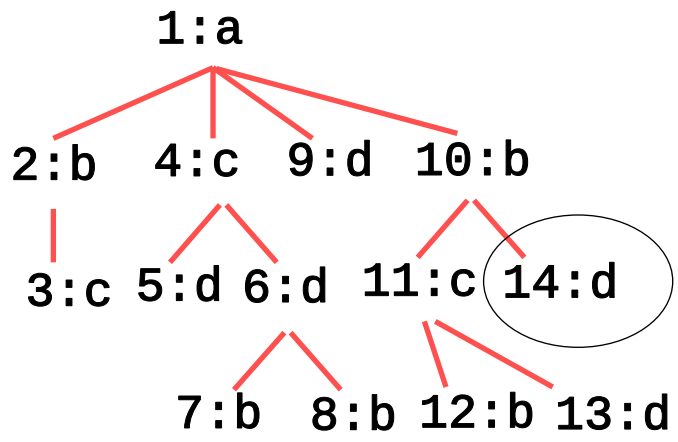


a) //b

(9) For the tree given in 6, write XPath expressions that

a) select all b nodes
b) select all b nodes that have a c-child
c) select all b nodes that have no c-children
d) select the right most c-node
e) select all nodes that have a c-parent

```
<a><b><c/></b><c><d/><d><b/><b/></d></c><d/><b><c><b/><d/></c><d/>
</b></a>
```



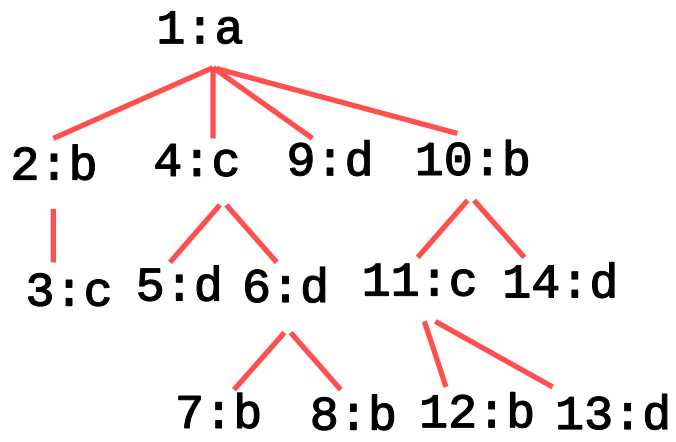
a) //b

b) //b[c]

(9) For the tree given in 6, write XPath expressions that

a) select all b nodes
b) select all b nodes that have a c-child
c) **select all b nodes that have no c-children**
d) select the right most c-node
e) select all nodes that have a c-parent

```
<a><b><c/></b><c><d/><d><b/><b/></d></c><d/><b><c><b/><d/></c><d/>
</b></a>
```



a) //b

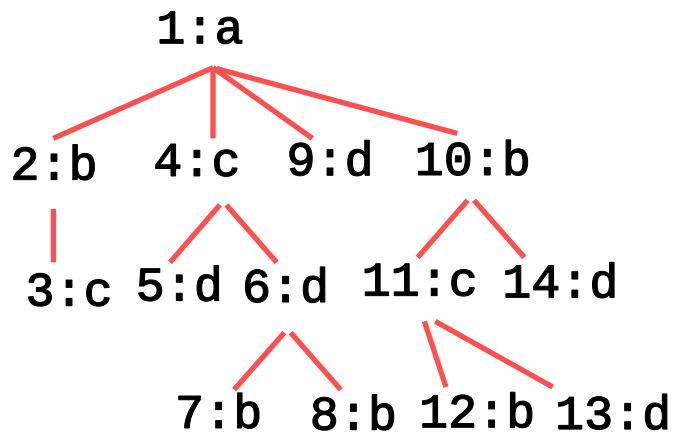
b) //b[c]

c) //b[not(c)]

(9) For the tree given in 6, write XPath expressions that

a) select all b nodes
b) select all b nodes that have a c-child
c) select all b nodes that have no c-children
d) select the right most c-node (node no 11)
e) select all nodes that have a c-parent

```
<a><b><c/></b><c><d/><d><b/><b/></d></c><d/><b><c><b/><d/></c><d/>
</b></a>
```



a) //b

b) //b[c]

c) //b[not(c)]

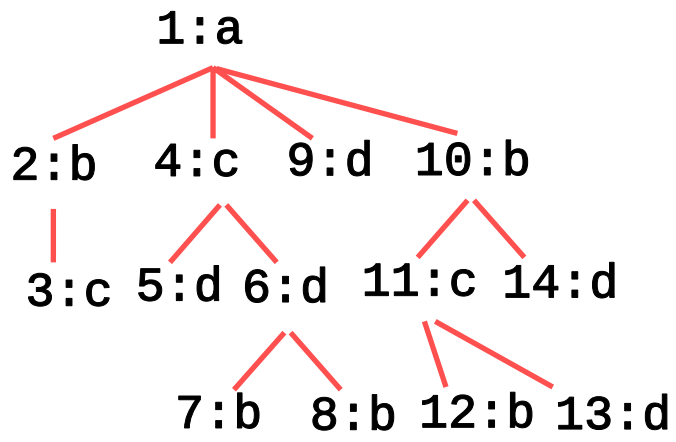
d) //c[b]

or **//c[b and d]**

(9) For the tree given in 6, write XPath expressions that

a) select all b nodes
b) select all b nodes that have a c-child
c) select all b nodes that have no c-children
d) select the right most c-node (node no 11)
e) select all nodes that have a c-parent

```
<a><b><c/></b><c><d/><d><b/><b/></d></c><d/><b><c><b/><d/></c><d/>
</b></a>
```



a) //b

b) //b[c]

c) //b[not(c)]

d) //c[b]

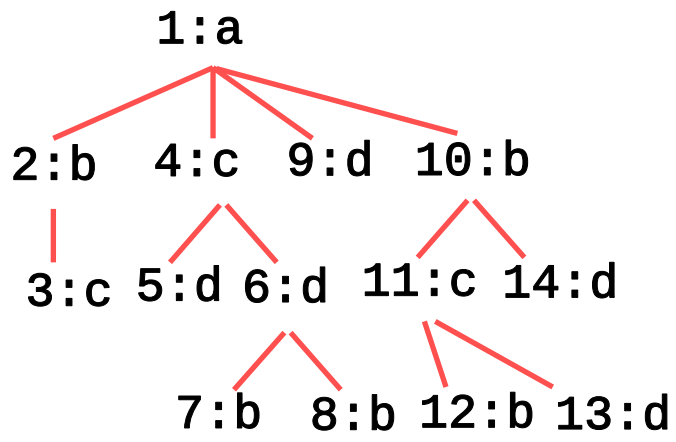
or //c[b and d]

e) //*[parent::c]

or **//c/***

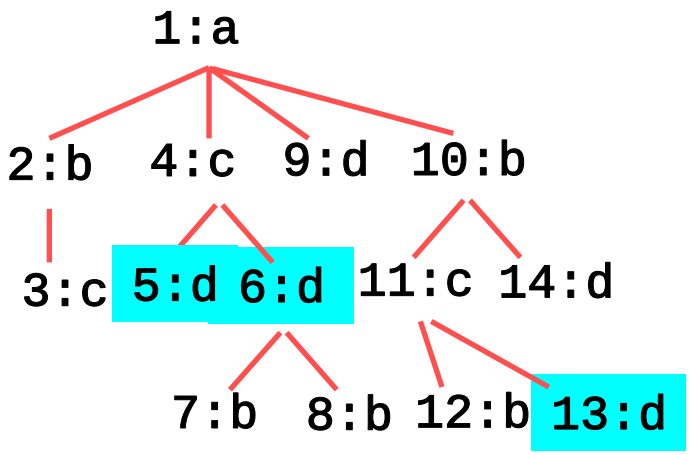
(10) Below is a tree corresponding to the document in (6).
Show the sequences of node numbers that are selected by the following queries.

a) //c//d
b) //*[a or b]
c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]
e) //c//d/preceding::*//d



(10) Below is a tree corresponding to the document in (6).
Show the sequences of node numbers that are selected by the following queries.

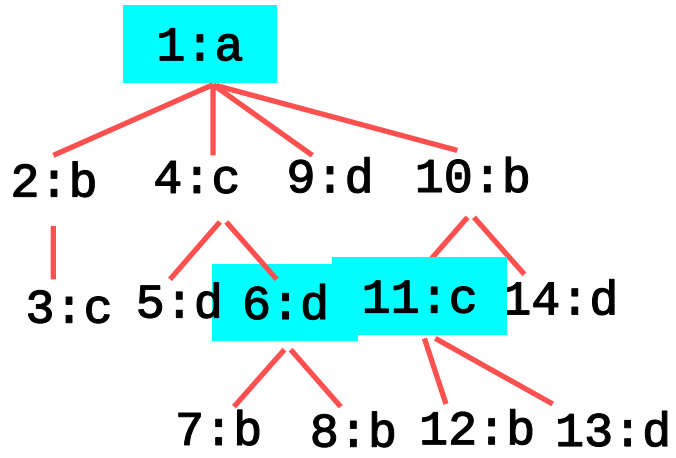
a) //c//d
b) //*[a or b]
c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]
e) //c//d/preceding::*//d



a) 5, 6, 13

(10) Below is a tree corresponding to the document in (6).
Show the sequences of node numbers that are selected by the following queries.

a) //c//d
b) //*[a or b]
c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]
e) //c//d/preceding::*//d

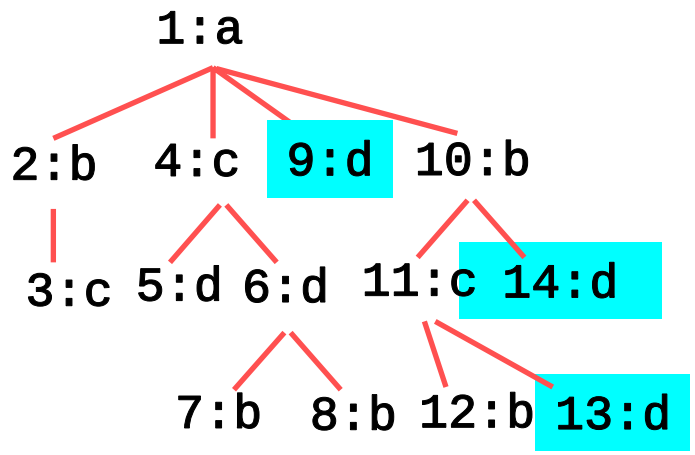


a) 5, 6, 13

b) 1, 6, 11

(10) Below is a tree corresponding to the document in (6).
Show the sequences of node numbers that are selected by the following queries.

a) //c//d
b) //*[a or b]
c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]
e) //c//d/preceding::*//d



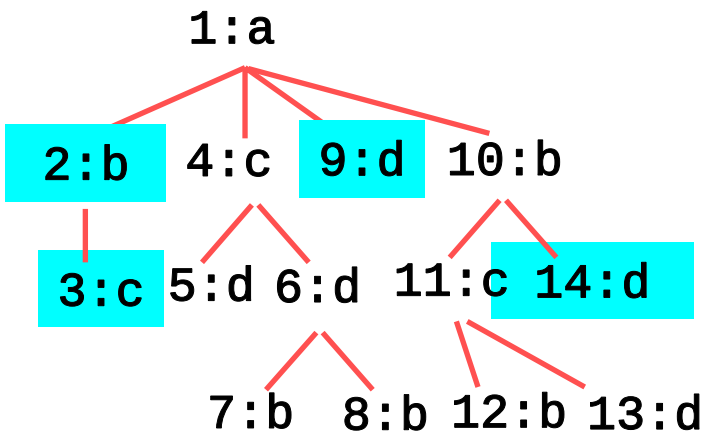
a) 5, 6, 13

b) 1, 6, 11

c) 9, 13, 14

(10) This a tree corresponding to the XML in (6).
Show the sequences of node numbers that are selected by the following queries.

a) //c//d
b) //*[a or b]
c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]
e) //c//d/preceding::*//d



a) 5, 6, 13

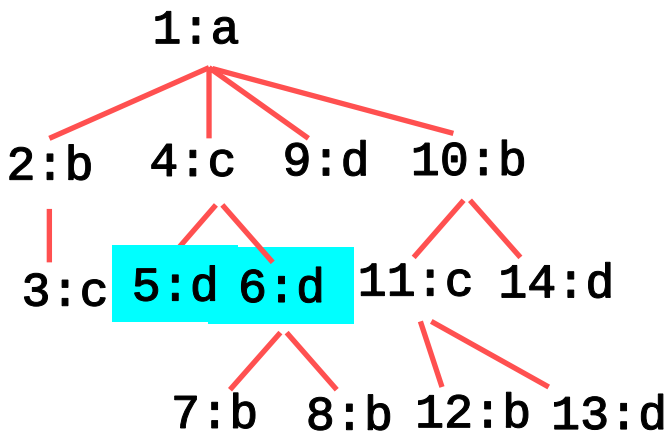
b) 1, 6, 11

c) 9, 13, 14

d) 2, 3, 9, 14

(10) This a tree corresponding to the XML in (6).
Show the sequences of node numbers that are selected by the following queries.

a) //c//d
b) //*[a or b]
c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]
e) //c//d/preceding::*//d



a) 5, 6, 13

b) 1, 6, 11

c) 9, 13, 14

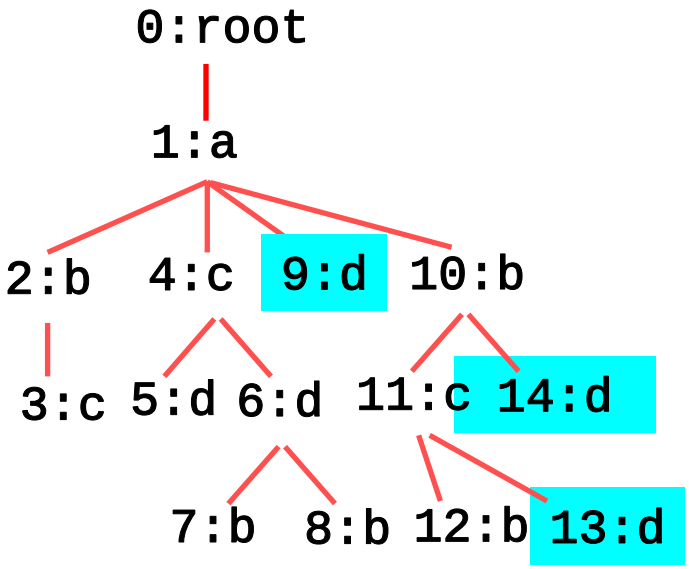
d) 2, 3, 9, 14

e) 5, 6

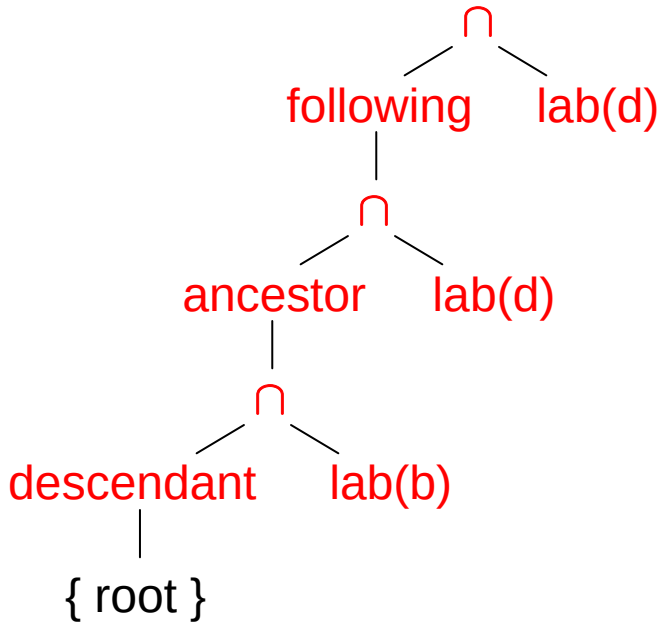
(10) This a tree corresponding to the XML in (6).

c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]

For **query c)** show in detail how the **Core-XPath evaluation algorithm** computes the answer to this query.
Do the same for query d).



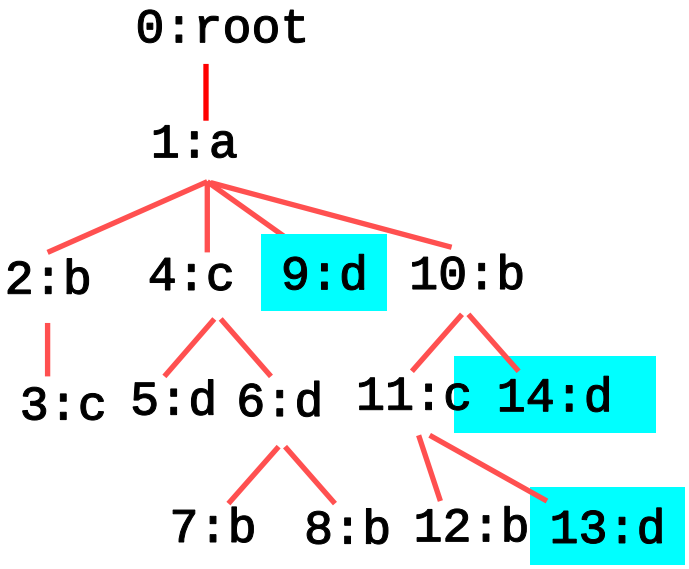
lab(a) = { 1 }
lab(b) = { 2, 7, 8, 10, 12 }
lab(c) = { 3, 4, 11 }
lab(d) = { 5, 6, 9, 13, 14 }



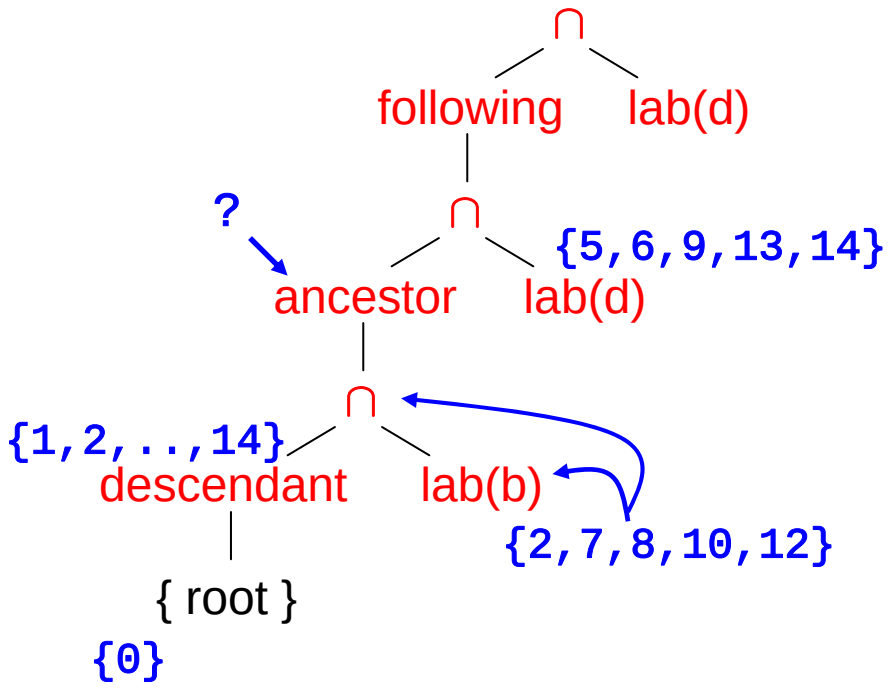
(10) This a tree corresponding to the XML in (6).

c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]

For **query c)** show in detail how the **Core-XPath evaluation algorithm** computes the answer to this query.
Do the same for query d).



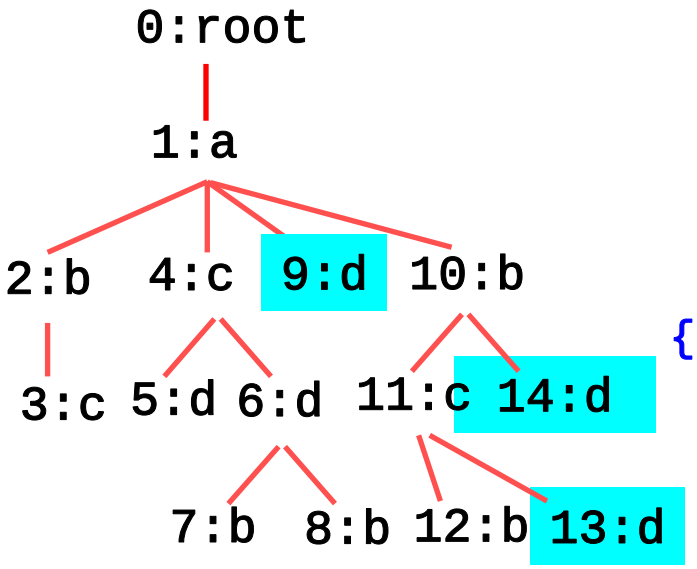
lab(a) = { 1 }
lab(b) = { 2, 7, 8, 10, 12 }
lab(c) = { 3, 4, 11 }
lab(d) = { 5, 6, 9, 13, 14 }



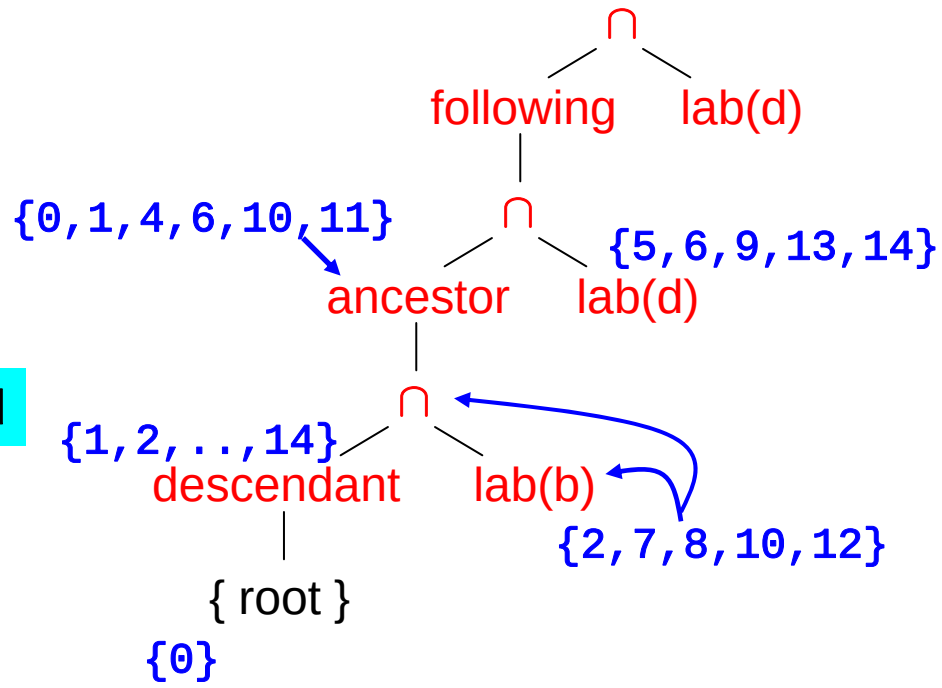
(10) This a tree corresponding to the XML in (6).

c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]

For **query c)** show in detail how the **Core-XPath evaluation algorithm** computes the answer to this query.
Do the same for query d).



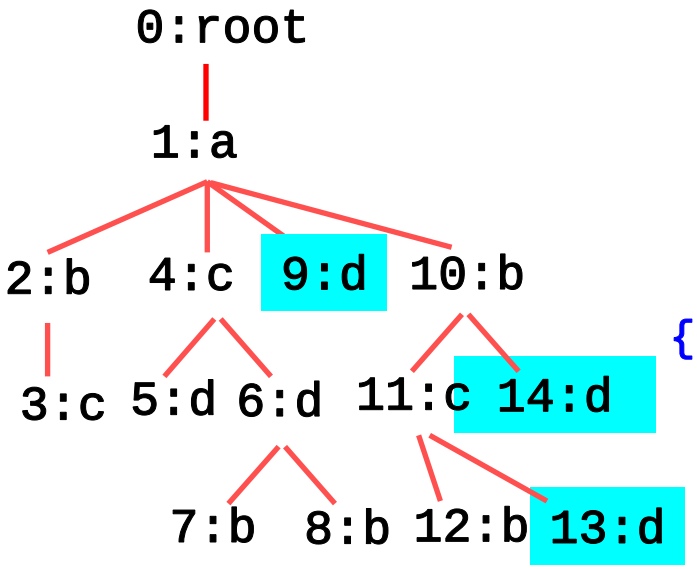
lab(a) = { 1 }
lab(b) = { 2, 7, 8, 10, 12 }
lab(c) = { 3, 4, 11 }
lab(d) = { 5, 6, 9, 13, 14 }



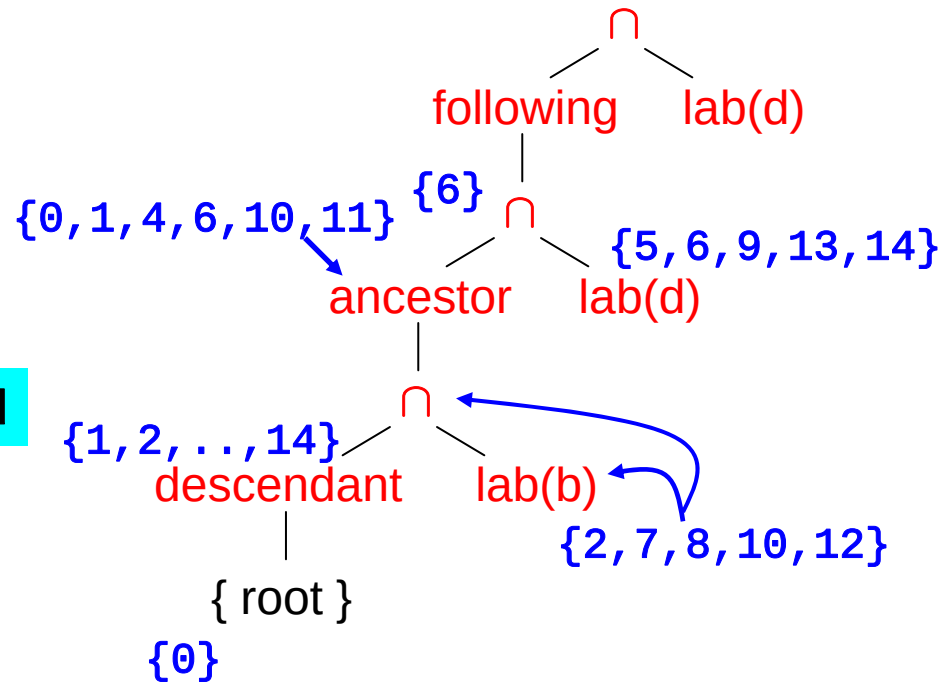
(10) This a tree corresponding to the XML in (6).

c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]

For **query c)** show in detail how the **Core-XPath evaluation algorithm** computes the answer to this query.
Do the same for query d).



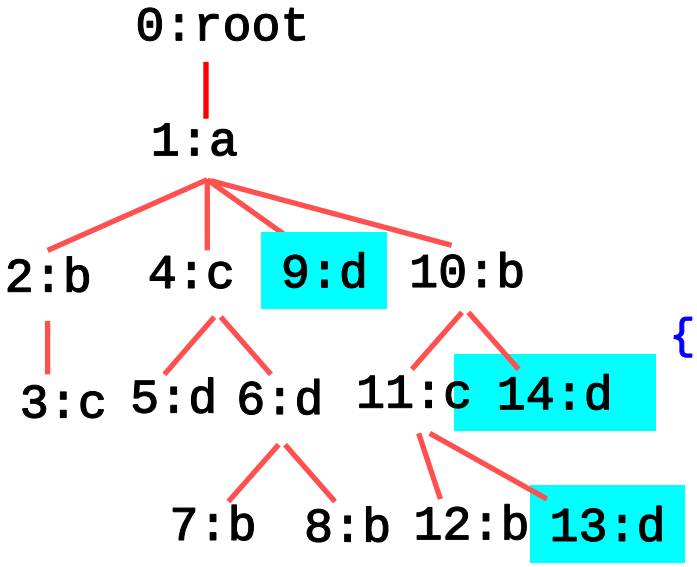
lab(a) = { 1 }
lab(b) = { 2, 7, 8, 10, 12 }
lab(c) = { 3, 4, 11 }
lab(d) = { 5, 6, 9, 13, 14 }



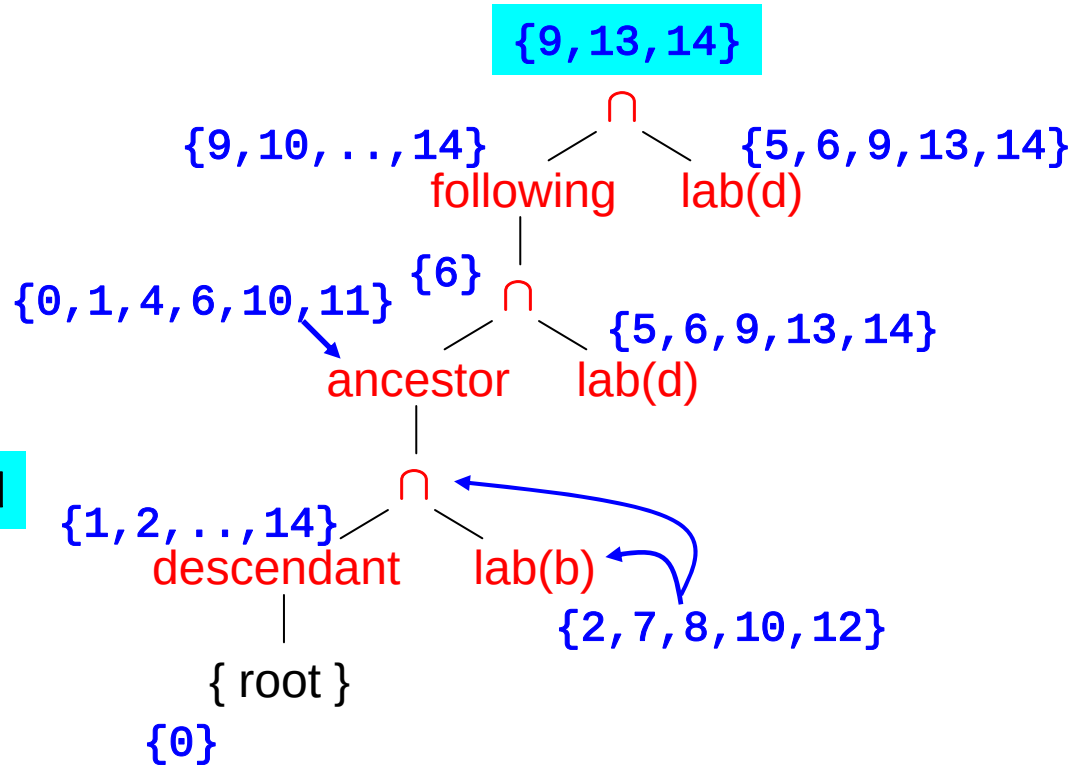
(10) This a tree corresponding to the XML in (6).

c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]

For **query c)** show in detail how the **Core-XPath evaluation algorithm** computes the answer to this query.
Do the same for query d).



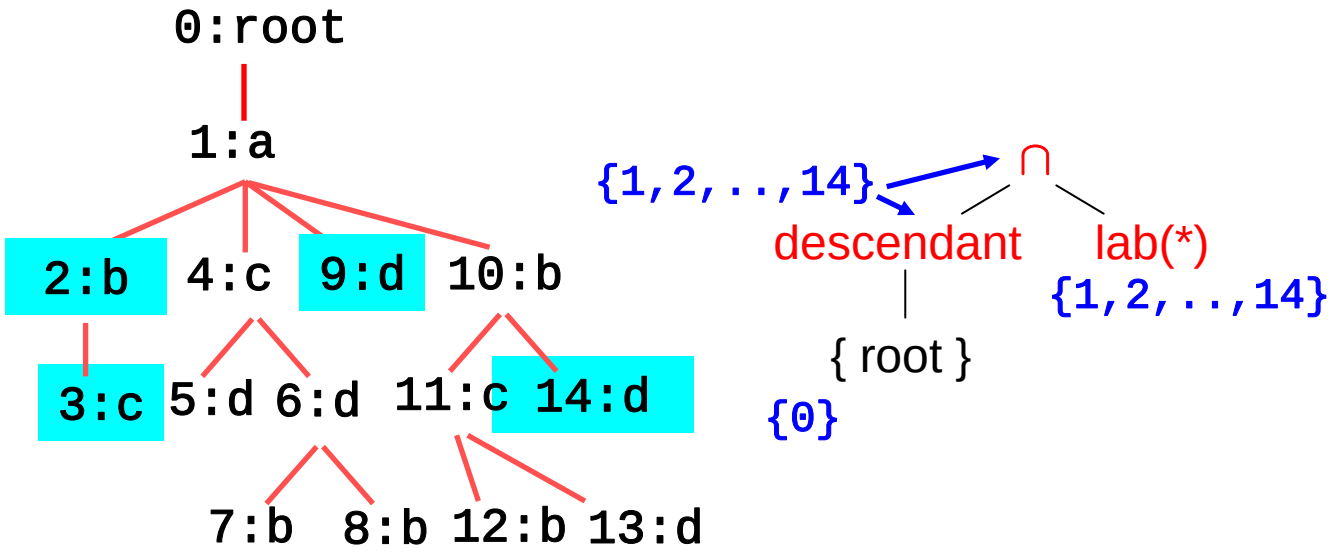
lab(a) = { 1 }
lab(b) = { 2, 7, 8, 10, 12 }
lab(c) = { 3, 4, 11 }
lab(d) = { 5, 6, 9, 13, 14 }



(10) This a tree corresponding to the XML in (6).

c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]

For query c) show in detail how the **Core-XPath evaluation algorithm** computes the answer to this query.
Do the same for **query d)**.

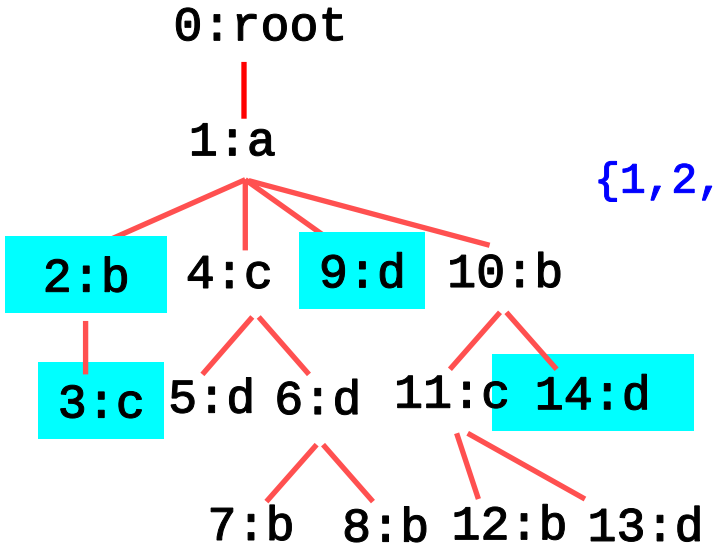


lab(a) = { 1 }
lab(b) = { 2, 7, 8, 10, 12 }
lab(c) = { 3, 4, 11 }
lab(d) = { 5, 6, 9, 13, 14 }

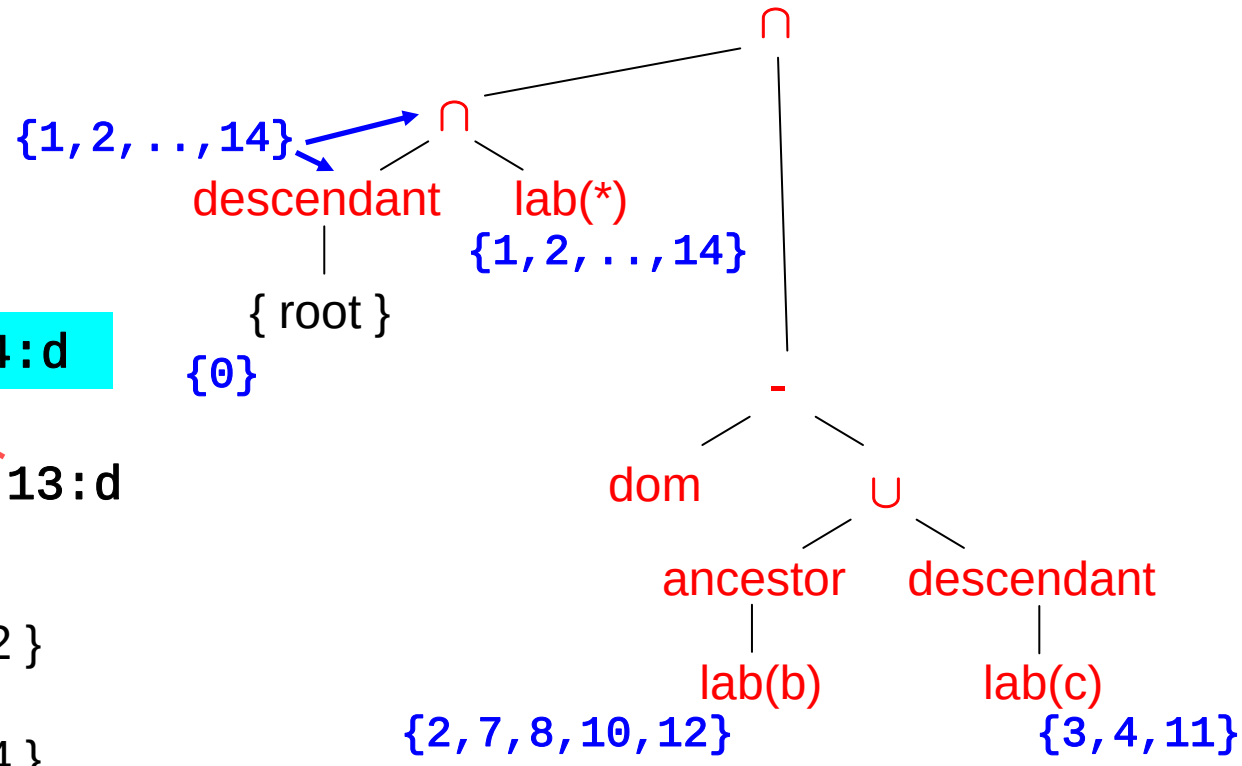
(10) This a tree corresponding to the XML in (6).

c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]

For query c) show in detail how the **Core-XPath evaluation algorithm** computes the answer to this query.
Do the same for **query d)**.



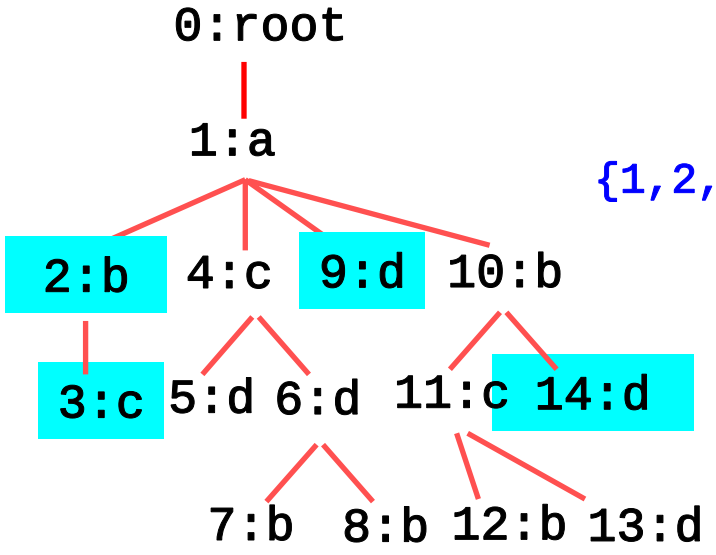
lab(a) = { 1 }
lab(b) = { 2, 7, 8, 10, 12 }
lab(c) = { 3, 4, 11 }
lab(d) = { 5, 6, 9, 13, 14 }



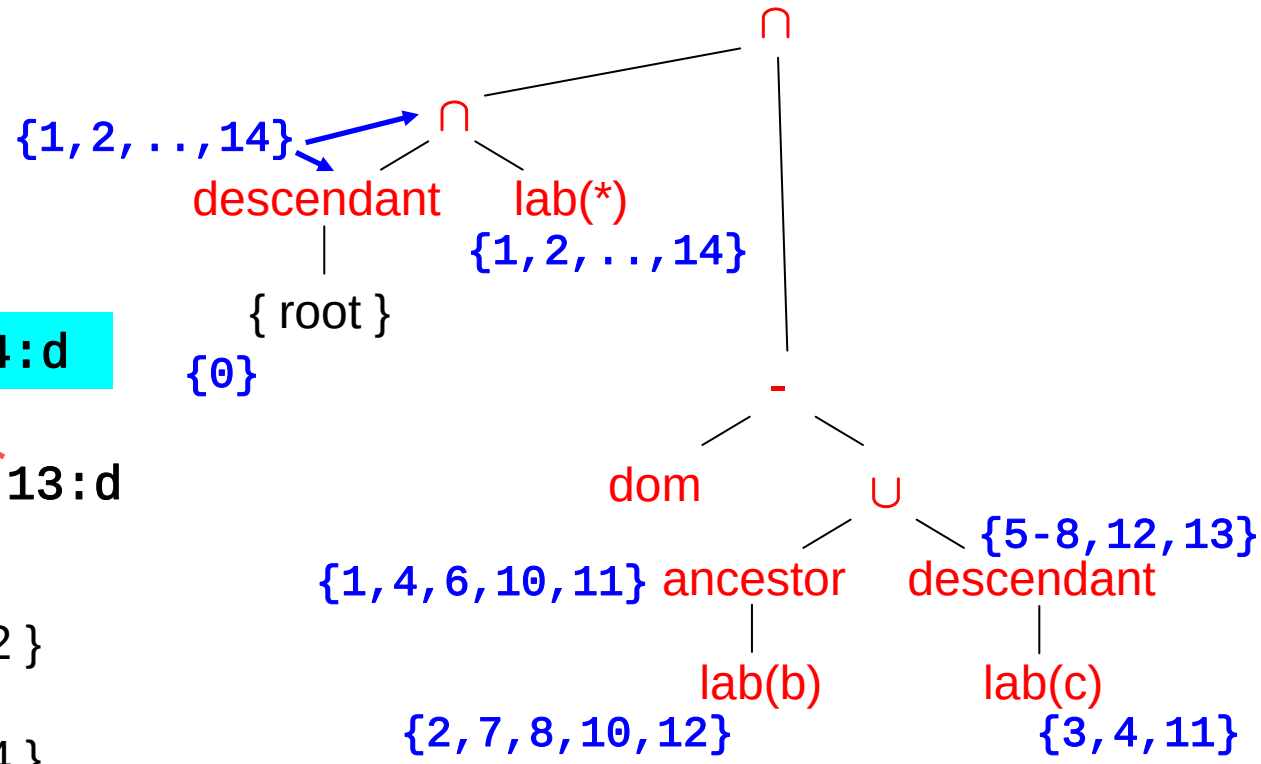
(10) This a tree corresponding to the XML in (6).

c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]

For query c) show in detail how the **Core-XPath evaluation algorithm** computes the answer to this query.
Do the same for **query d)**.



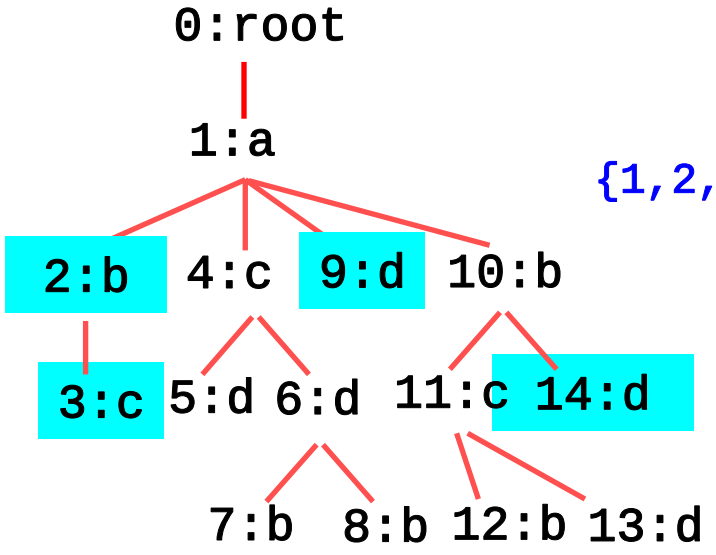
lab(a) = { 1 }
lab(b) = { 2, 7, 8, 10, 12 }
lab(c) = { 3, 4, 11 }
lab(d) = { 5, 6, 9, 13, 14 }



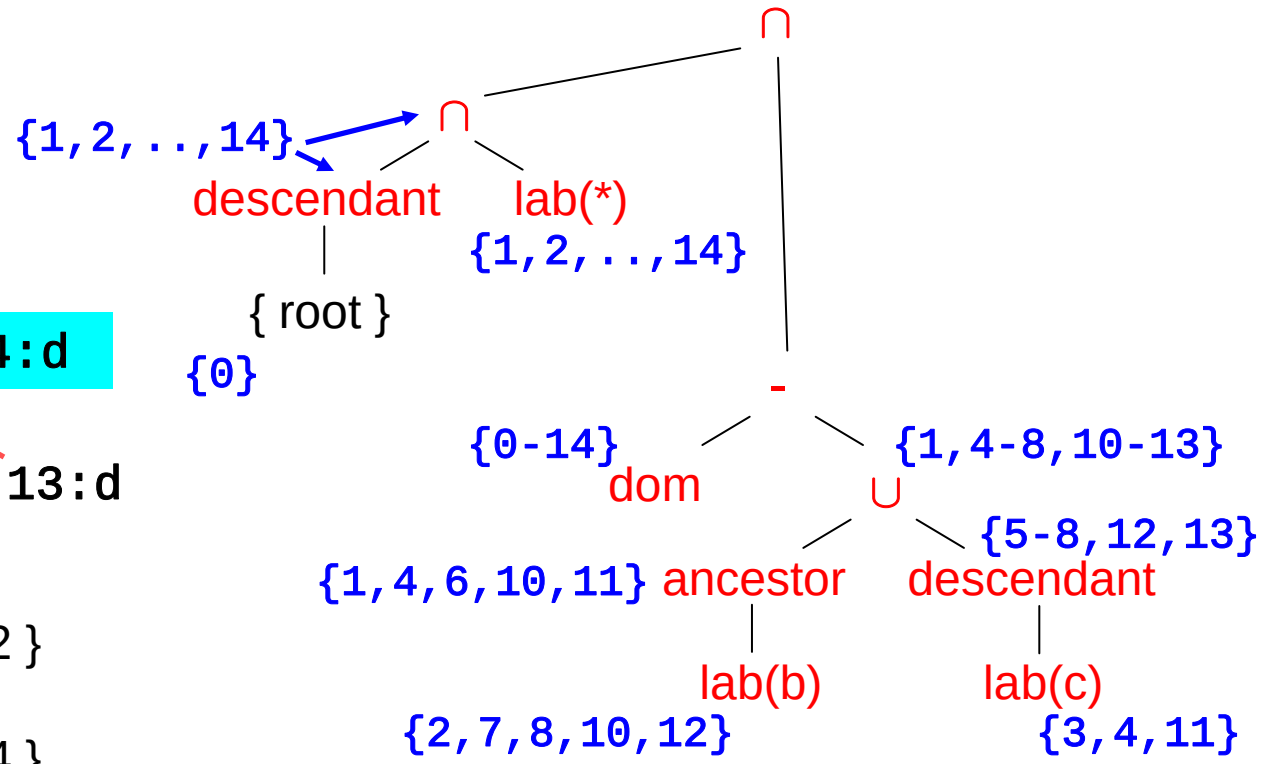
(10) This a tree corresponding to the XML in (6).

c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]

For query c) show in detail how the **Core-XPath evaluation algorithm** computes the answer to this query.
Do the same for **query d)**.



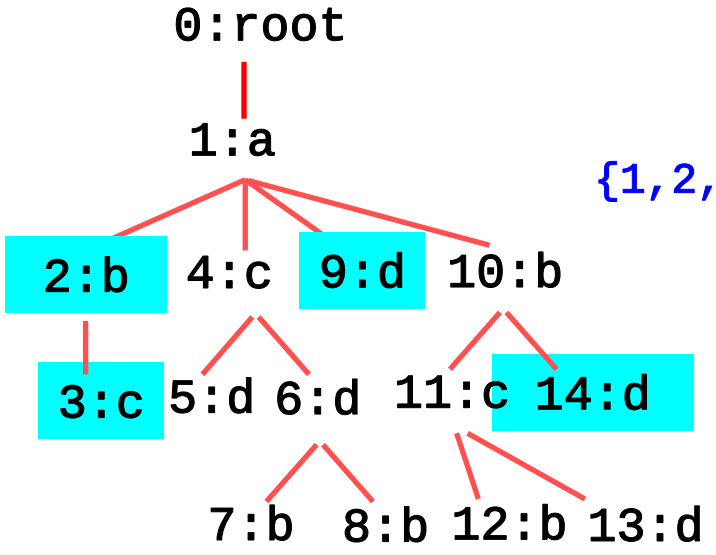
lab(a) = { 1 }
lab(b) = { 2, 7, 8, 10, 12 }
lab(c) = { 3, 4, 11 }
lab(d) = { 5, 6, 9, 13, 14 }



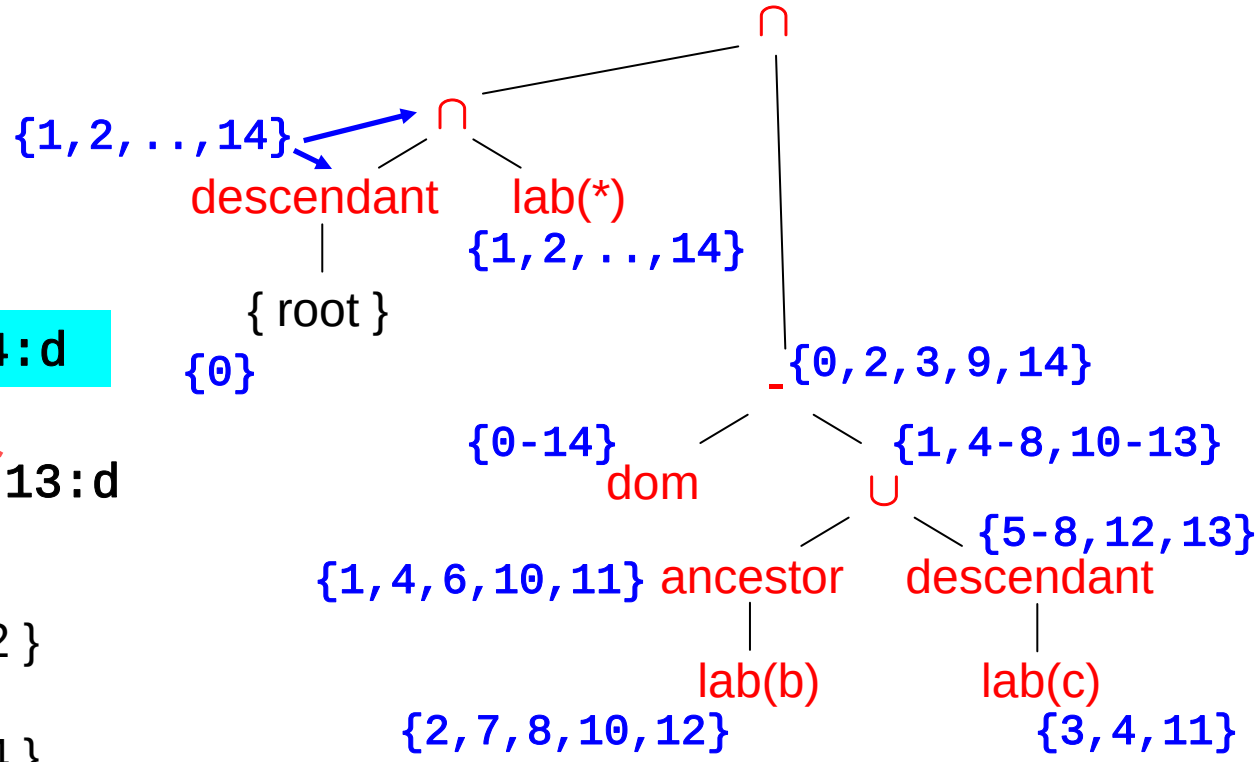
(10) This a tree corresponding to the XML in (6).

c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]

For query c) show in detail how the **Core-XPath evaluation algorithm** computes the answer to this query.
Do the same for **query d)**.



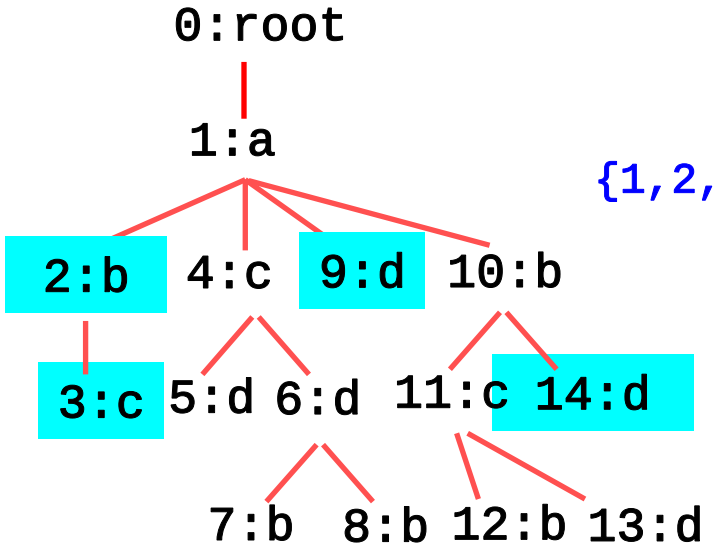
lab(a) = { 1 }
lab(b) = { 2, 7, 8, 10, 12 }
lab(c) = { 3, 4, 11 }
lab(d) = { 5, 6, 9, 13, 14 }



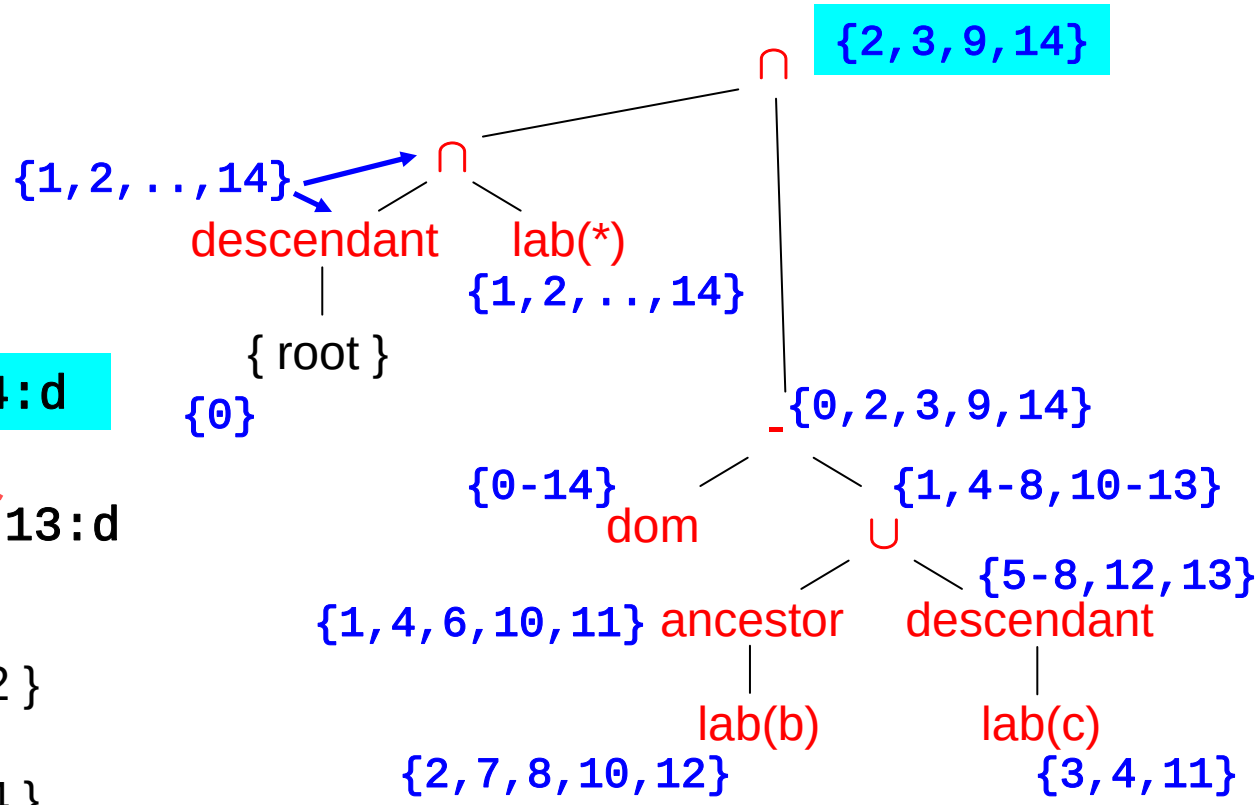
(10) This a tree corresponding to the XML in (6).

c) //b/ancestor::d/following::d
d) //*[not(.//b | ./ancestor::c)]

For query c) show in detail how the **Core-XPath evaluation algorithm** computes the answer to this query.
Do the same for **query d)**.



lab(a) = { 1 }
lab(b) = { 2, 7, 8, 10, 12 }
lab(c) = { 3, 4, 11 }
lab(d) = { 5, 6, 9, 13, 14 }



(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether

p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in

q=a[.//b[c//*//d]/*[c/d]]

---

(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
  p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
  q=a[.//b[c//*//d]/*[c/d]]

---

q1 = /b//b
q2 = //b

or

q1 = //b
q2 = //*

(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
q=a[.//b[c//*//d]/*[c/d]]

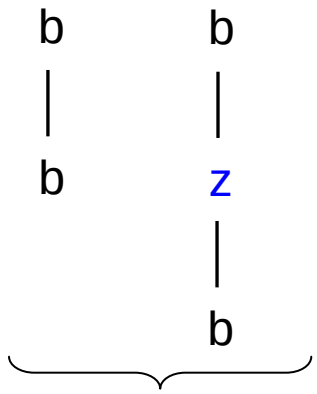
---

q1 = /b//b
q2 = //b

Canonical model:

Replace in q1 // by
$\leq$ N+1 many /z/

0, here

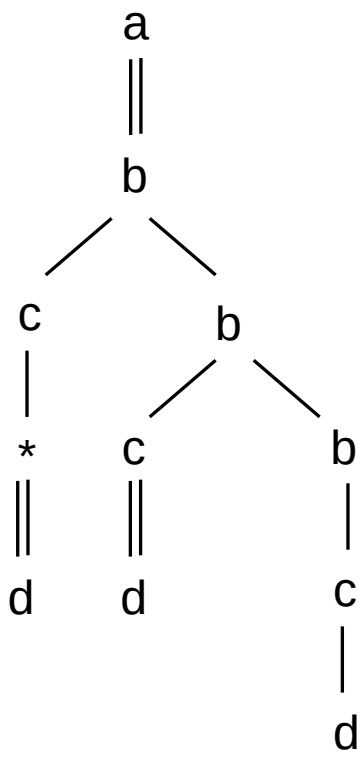


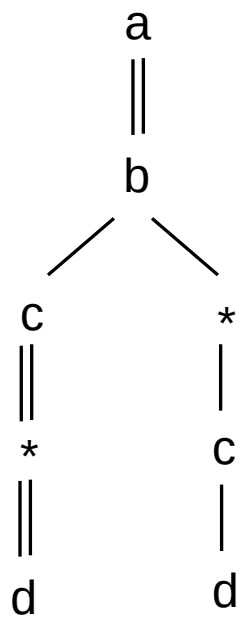
Do they match q2? → yes!!

or

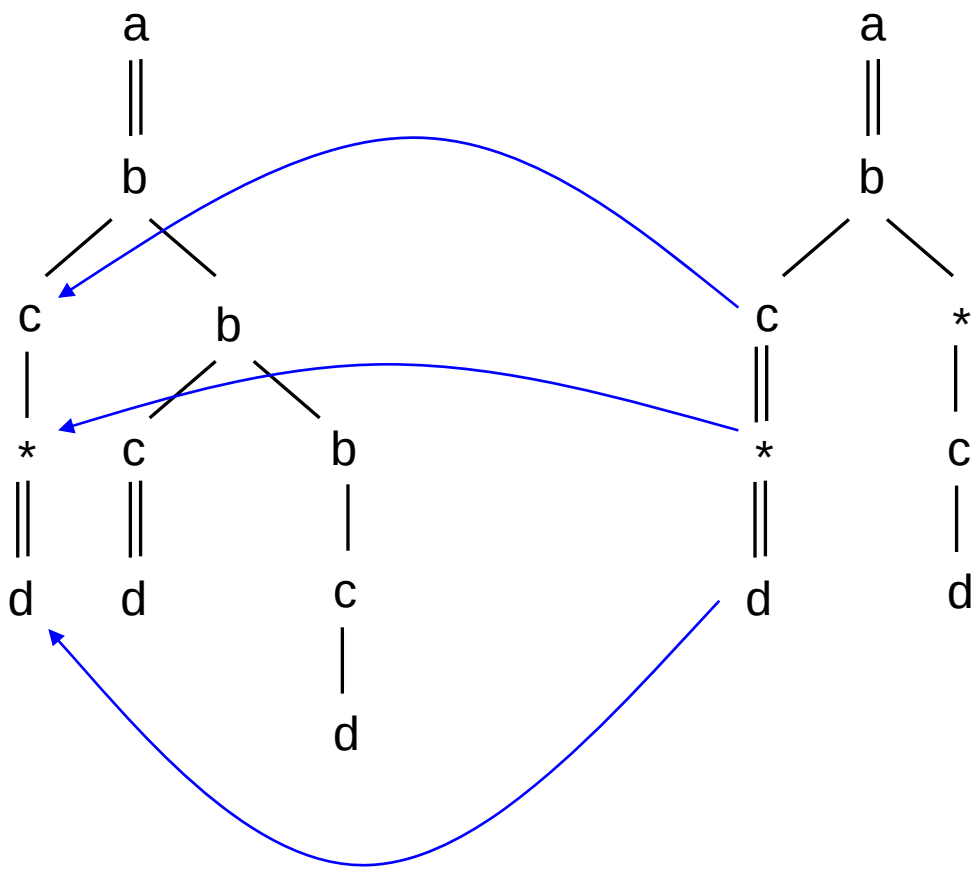
q1 = //b
q2 = //*

(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
  p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
  q=a[.//b[c//*//d]/*[c/d]]

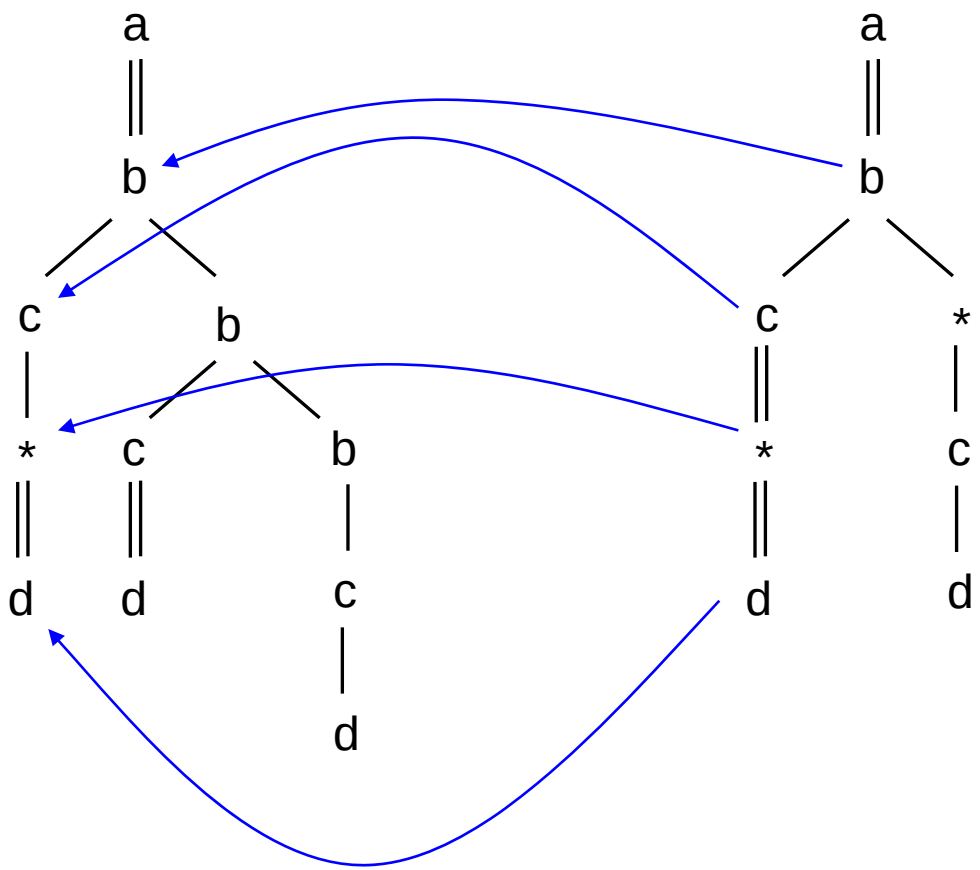




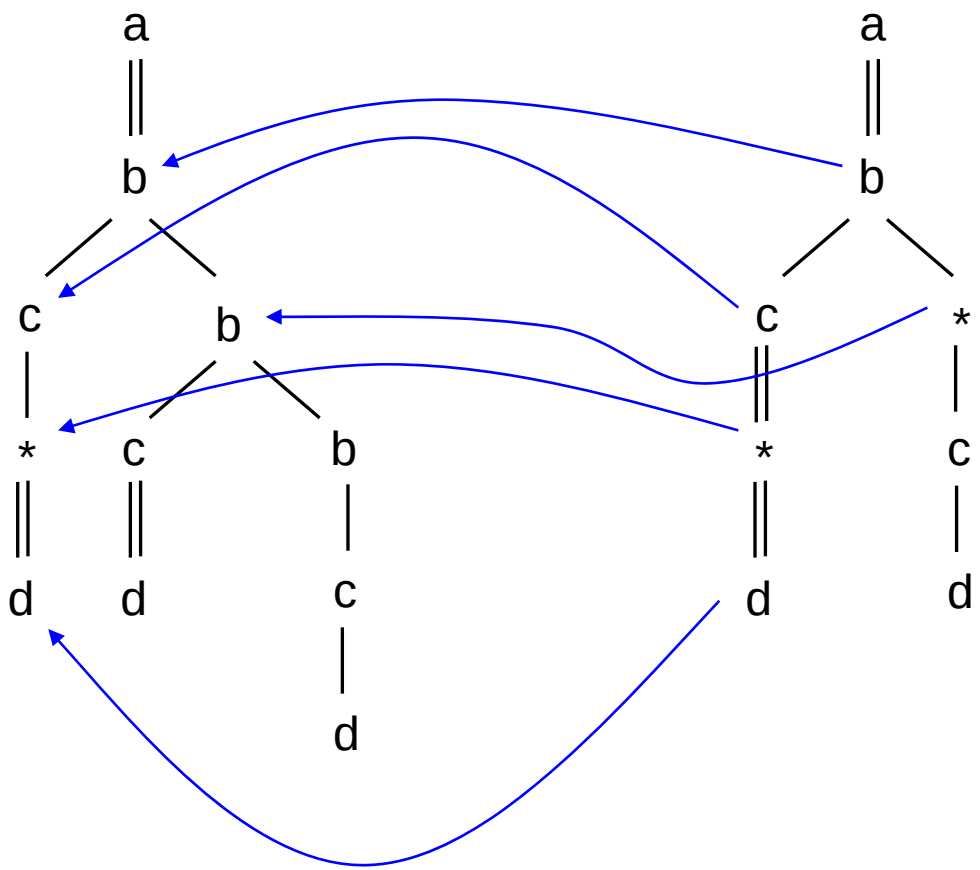
(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
  p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
  q=a[.//b[c//*//d]/*[c/d]]



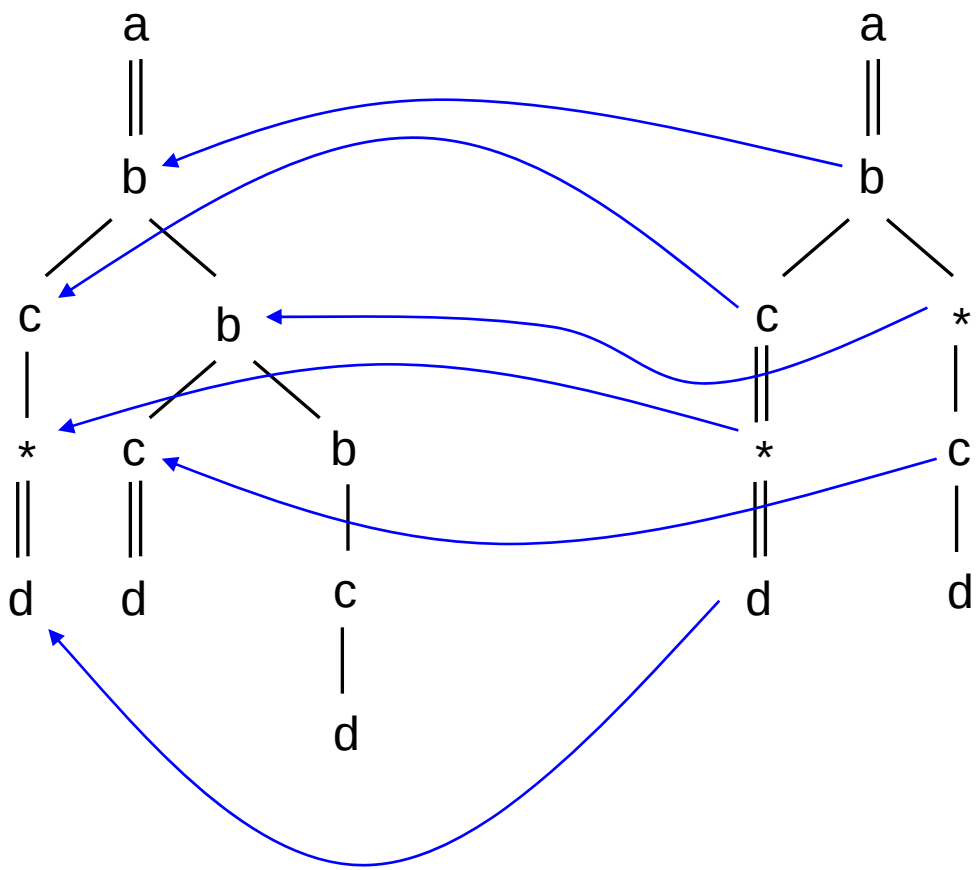
(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
q=a[.//b[c//*//d]/*[c/d]]



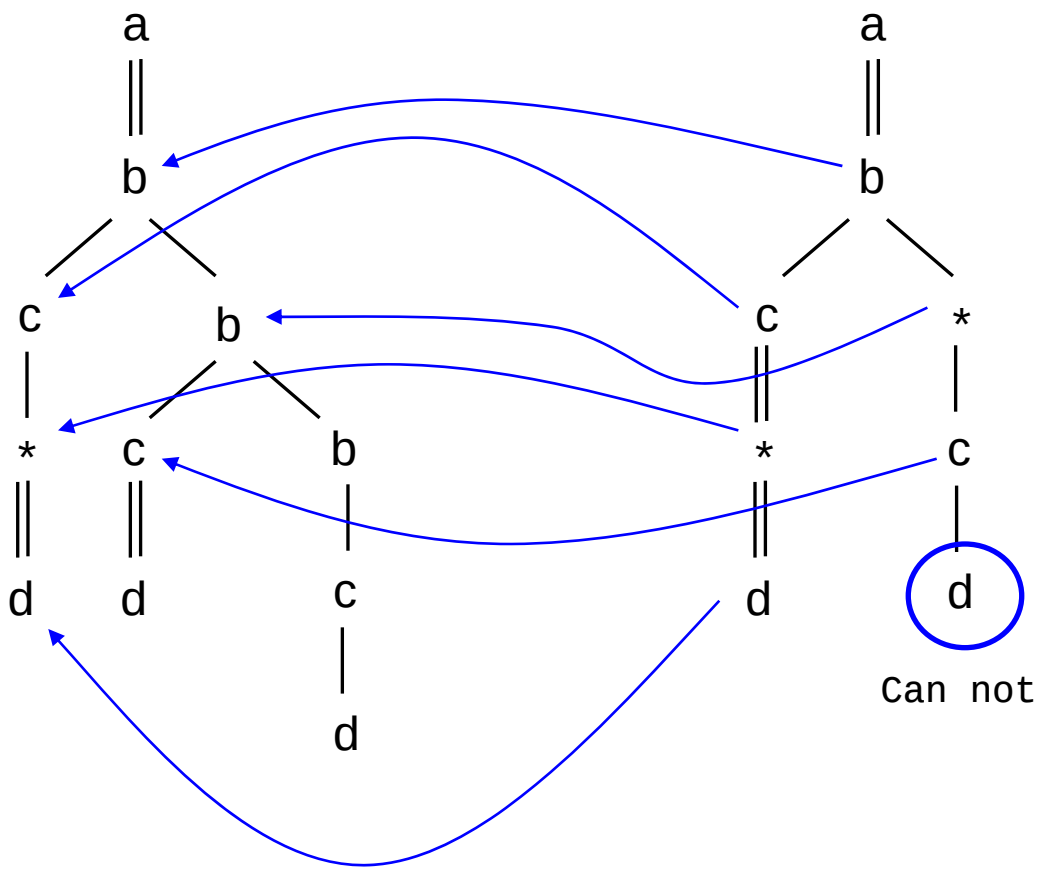
(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
  p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
  q=a[.//b[c//*//d]/*[c/d]]



(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
q=a[.//b[c//*//d]/*[c/d]]

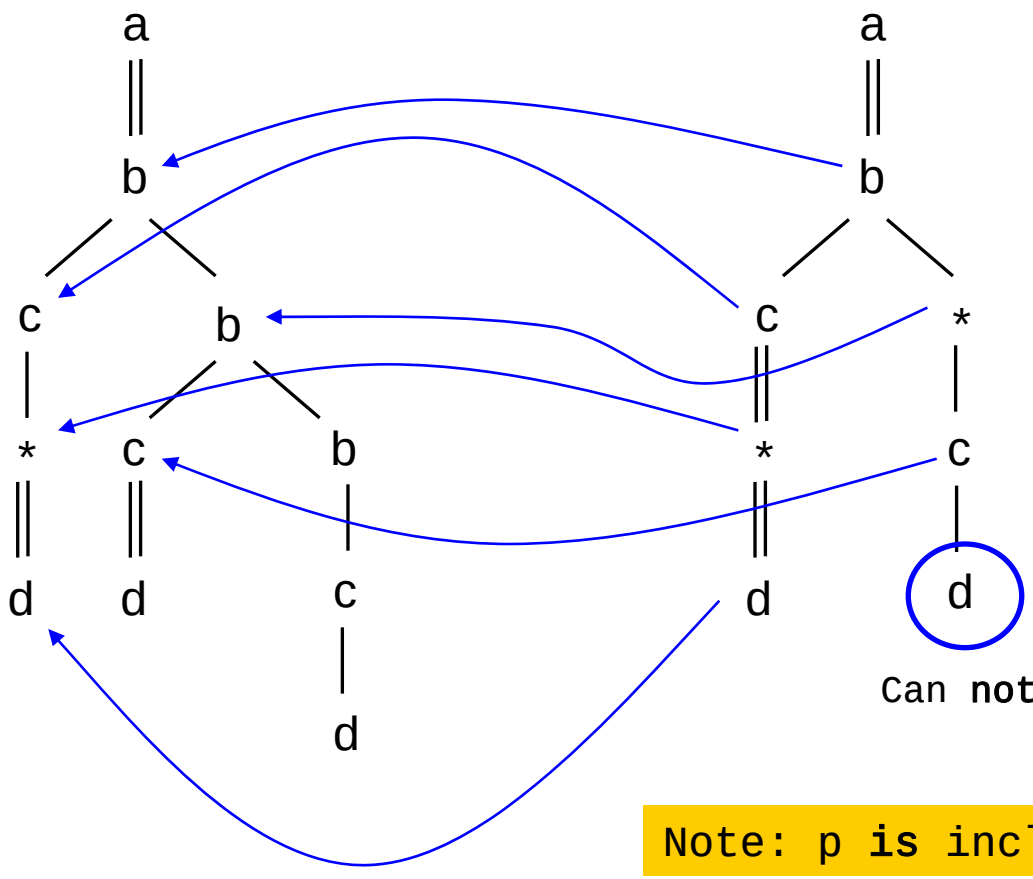


(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
  p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
  q=a[.//b[c//*//d]/*[c/d]]



Can not be mapped.

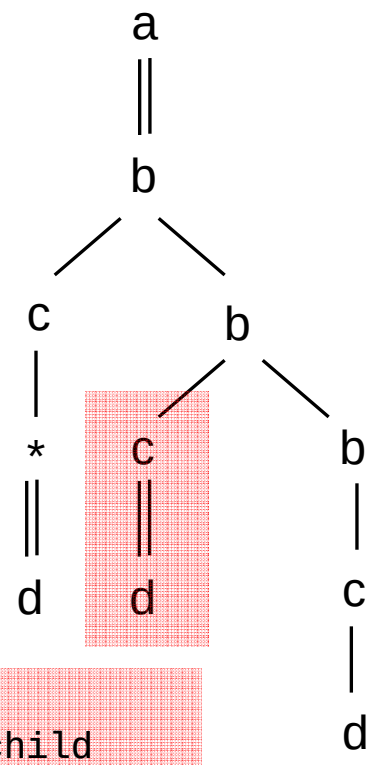
(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
q=a[.//b[c//*//d]/*[c/d]]



Can **not** be mapped.

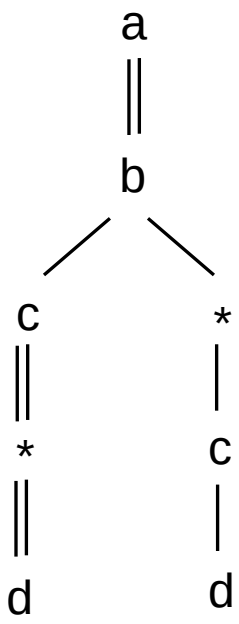
Note: p **is** included in q!!

(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
q=a[.//b[c//*//d]/*[c/d]]



Case 1:
c has d-child

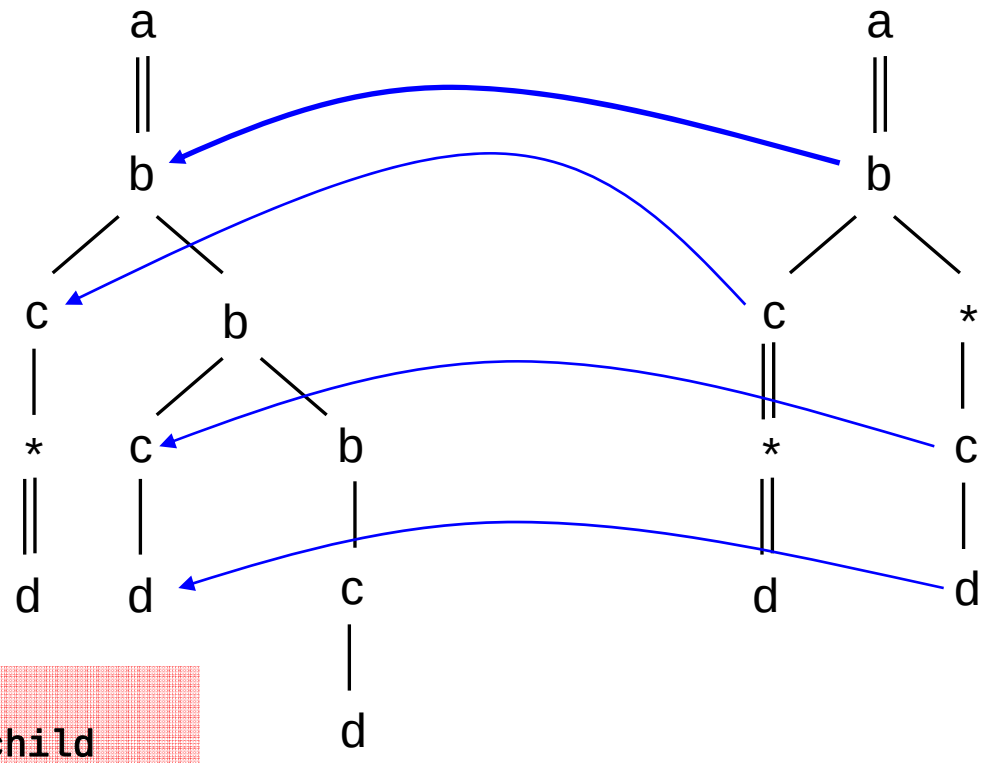
Case 2:
c has no d-child



Why?

Note: p **is** included in q!!

(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
q=a[.//b[c//*//d]/*[c/d]]



→ OK!!

p included in q, for **Case 1**!

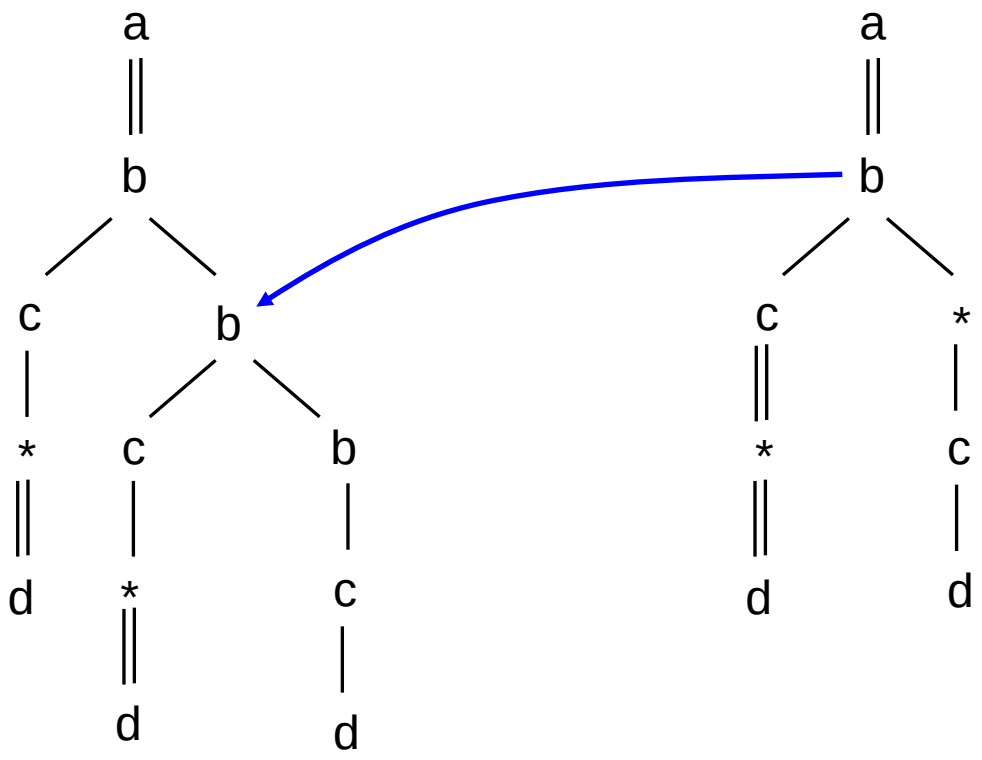
Case 1:
c has d-child

Case 2:
c has no d-child

Why?

Note: p **is** included in q!!

(11) Show an example of XPath queries q1,q2 such that they are not equivalent, but q1 is included in q2.
Show that q1 is included in q2 using one of the methods discussed.
Use the homomorphism technique to test whether
p=a[.//b[c/*//d]/b[c//d]/b[c/d]] is included in
q=a[.//b[c//*//d]/*[c/d]]



Also OK!!

p included in q, for **Case 2**!

Case 2:
c has no d-child

Why?

Note: p **is** included in q!!

(12) Construct a DTD such that 10a) is included in 10e),
and another DTD such that 10e) is included in 10a).
[Very easy!!]

10a) //c//d
10e) //c//d/preceding::*//d

---

(12) Construct a DTD such that 10a) is included in 10e),
and another DTD such that 10e) is included in 10a).
[Very easy!!]

10a) //c//d
10e) //c//d/preceding::*//d

For the first part:
Any DTD so that c-nodes do NOT have d-descendants! ☺

For the second part:
Any DTD, so that all d-nodes are c-descendants.

(14) Given a PRE/POST/SIZE table, show SQL queries for the XPath queries

a) /*
b) /a/b/*
c) //a/*//b
d) //a/following-sibling::b

---

(14) Given a PRE/POST/SIZE table, show SQL queries for the XPath queries

a) /*
b) /a/b/*
c) //a/*//b
d) //a/following-sibling::b

```
SELECT DISTINCT r1.pre FROM doc_tbl r1
WHERE r1.pre=1
ORDERED BY r1.pr
```

(14) Given a PRE/POST/SIZE table, show SQL queries for the XPath queries

a) /*
b) /a/b/*
c) //a/*//b
d) //a/following-sibling::b

```
SELECT DISTINCT r4.pre FROM doc_tbl r1, r2, r3, r4
         WHERE r1.pre=0
           AND r2.pre>r1.pre
           AND r2.post<r1.post
           AND (r2.pre-r2.post+r2.size)=(r1.pre-r1.post+r1.size)+1
           AND r2.tag="a"
           AND r3.pre>r2.pre
           AND r3.post<r2.post
           AND (r3.pre-r3.post+r3.size)=(r2.pre-r2.post+r2.size)+1
           AND r3.tag="b"
           AND r4.pre>r3.pre
           AND r4.post<r3.post
           AND (r4.pre-r4.post+r4.size)=(r3.pre-r3.post+r3.size)+1
ORDERED BY r4.pre
```

Recall: level(n) = pre(n)–post(n)+size(n)

(14) Given a PRE/POST/SIZE table, show SQL queries for the XPath queries

a) /*
b) /a/b/*
c) //a/*//b
d) //a/following-sibling::b

```
SELECT DISTINCT r4.pre FROM doc_tbl r1, r2, r3, r4
         WHERE r1.pre=0
           AND r2.pre>r1.pre
           AND r2.post<r1.post
           AND r2.tag="a"
           AND r3.pre>r2.pre
           AND r3.post<r2.post
           AND (r3.pre-r3.post+r3.size)=(r2.pre-r2.post+r2.size)+1
           AND r4.pre>r3.pre
           AND r4.post<r3.post
           AND r4.tag="b"
ORDERED BY r4.pre
```

Recall: level(n) = pre(n)–post(n)+size(n)

(14) Given a PRE/POST/SIZE table, show SQL queries for the XPath queries

a) /*
b) /a/b/*
c) //a/*//b
d) //a/following-sibling::b

```
SELECT DISTINCT r4.pre FROM doc_tbl r1, r2, r3, r4
         WHERE r1.pre=0
           AND r2.pre>r1.pre
           AND r2.post<r1.post
           AND r2.tag="a"
           AND r3.pre<r2.pre                                          parent of
           AND r3.post>r2.post                                        a-node
           AND (r3.pre-r3.post+r3.size)=(r2.pre-r2.post+r2.size)-1
           AND r4.pre>r2.pre                                  →after a
           AND r4.post<r3.post                                →before parent
           AND (r4.pre-r4.post+r4.size)=(r2.pre-r2.post+r2.size) →same level as a
           AND r4.tag="b"
ORDERED BY r4.pre
```

Recall: level(n) = pre(n)–post(n)+size(n)