XML and Databases

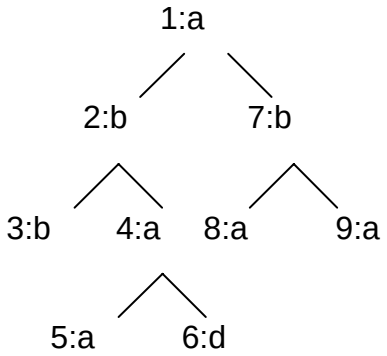
Exam Preparation
Part 2

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2010

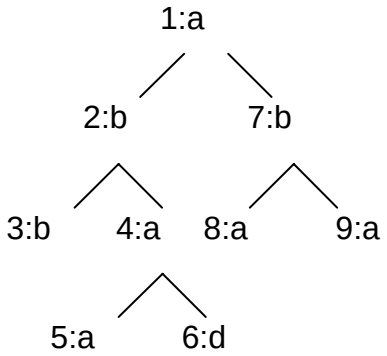
(6)**[4]** For the tree T on the right, write numbers of nodes selected by the following XPath expressions.

a) //a
b) /*//*//a[preceding::a]
c) //*[.//d]
d) /*[not(a and b)]
e) //*[count(.//*)=count(ancestor::*)]
f) //c[position()=last()]
g) /descendant:*[position() mod 2 = count(.//*)]
h) //*[count(*)>1 and not(child::*[not(self::a)])]
i) //*[preceding-sibling::b]



(6)**[4]** For the tree T on the right, write numbers of nodes selected by the following XPath expressions.

a) //a
b) /*//*//a[preceding::a]
c) //*[.//d]
d) /*[not(a and b)]
e) //*[count(.//*)=count(ancestor::*)]
f) //c[position()=last()]
g) /descendant:*[position() mod 2 = count(.//*)]
h) //*[count(*)>1 and not(child::*[not(self::a)])]
i) //*[preceding-sibling::b]

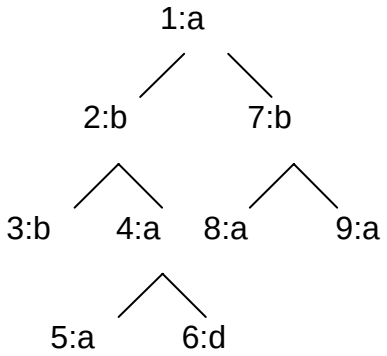


a) //a

Answer: 1,4,5,8,9

(6)**[4]** For the tree T on the right, write numbers of nodes selected by the following XPath expressions.

a) //a
b) /*//*//a[preceding::a]
c) //*[.//d]
d) /*[not(a and b)]
e) //*[count(.//*)=count(ancestor::*)]
f) //c[position()=last()]
g) /descendant:*[position() mod 2 = count(.//*)]
h) //*[count(*)>1 and not(child::*[not(self::a)])]
i) //*[preceding-sibling::b]

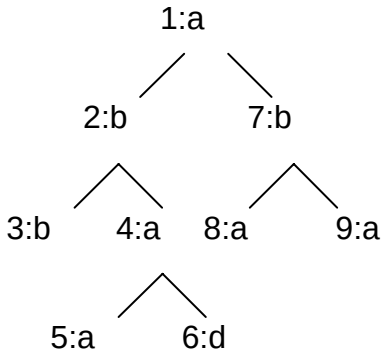


b) /*//*//a[preceding::a]

Answer: 8,9

(6)**[4]** For the tree T on the right, write numbers of nodes selected by the following XPath expressions.

a) //a
b) /*//*//a[preceding::a]
c) //*[.//d]
d) /*[not(a and b)]
e) //*[count(.//*)=count(ancestor::*)]
f) //c[position()=last()]
g) /descendant:*[position() mod 2 = count(.//*)]
h) //*[count(*)>1 and not(child::*[not(self::a)])]
i) //*[preceding-sibling::b]

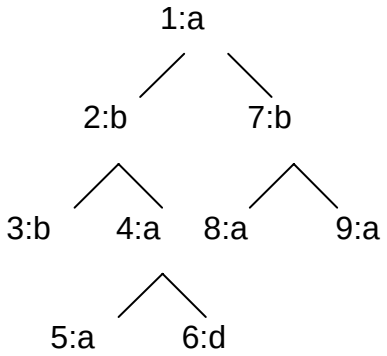


c) //*[.//d]

Answer: 1,2,4

(6)**[4]** For the tree T on the right, write numbers of nodes selected by the following XPath expressions.

a) //a
b) /*//*//a[preceding::a]
c) //*[.//d]
d) /*[not(a and b)]
e) //*[count(.//*)=count(ancestor::*)]
f) //c[position()=last()]
g) /descendant:*[position() mod 2 = count(.//*)]
h) //*[count(*)>1 and not(child::*[not(self::a)])]
i) //*[preceding-sibling::b]

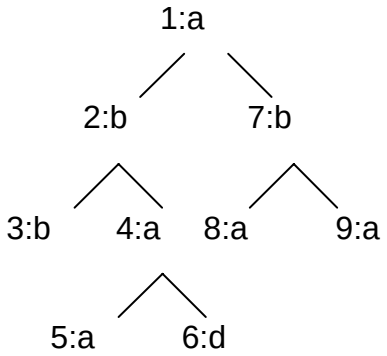


d) /*[not(a and b)]

Answer: 1

(6)**[4]** For the tree T on the right, write numbers of nodes selected by the following XPath expressions.

a) //a
b) /*//*//a[preceding::a]
c) //*[.//d]
d) /*[not(a and b)]
e) //*[count(.//*)=count(ancestor::*)]
f) //c[position()=last()]
g) /descendant:*[position() mod 2 = count(.//*)]
h) //*[count(*)>1 and not(child::*[not(self::a)])]
i) //*[preceding-sibling::b]

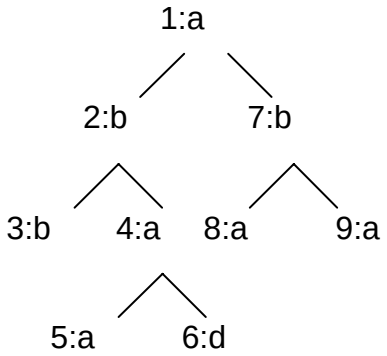


e) //*[count(.//*)=count(ancestor::*)]

Answer: 4

(6)**[4]** For the tree T on the right, write numbers of nodes selected by the following XPath expressions.

a) //a
b) /*//*//a[preceding::a]
c) //*[.//d]
d) /*[not(a and b)]
e) //*[count(.//*)=count(ancestor::*)]
f) //c[position()=last()]
g) /descendant:*[position() mod 2 = count(.//*)]
h) //*[count(*)>1 and not(child::*[not(self::a)])]
i) //*[preceding-sibling::b]

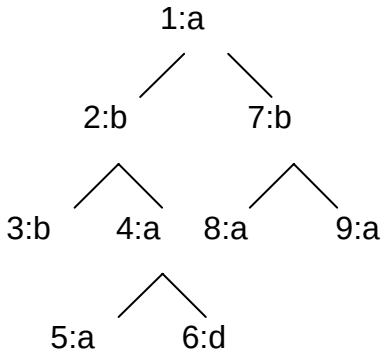


f) //c[position() = last()]

Answer: -

(6)**[4]** For the tree T on the right, write numbers of nodes selected by the following XPath expressions.

a) //a
b) /*//*//a[preceding::a]
c) //*[.//d]
d) /*[not(a and b)]
e) //*[count(.//*)=count(ancestor::*)]
f) //c[position()=last()]
g) /descendant:*[position() mod 2 = count(.//*)]
h) //*[count(*)>1 and not(child::*[not(self::a)])]
i) //*[preceding-sibling::b]

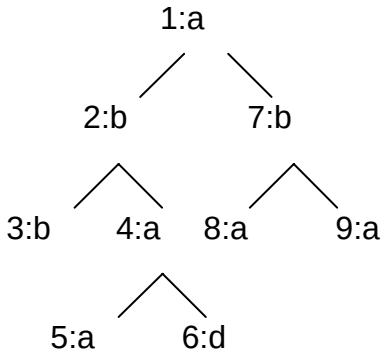


g) /descendant:*[position() mod 2 = count(.//*)]

Answer: 6,8

(6)**[4]** For the tree T on the right, write numbers of nodes selected by the following XPath expressions.

a) //a
b) /*//*//a[preceding::a]
c) //*[.//d]
d) /*[not(a and b)]
e) //*[count(.//*)=count(ancestor::*)]
f) //c[position()=last()]
g) /descendant:*[position() mod 2 = count(.//*)]
h) //*[count(*)>1 and not(child::*[not(self::a)])]
i) //*[preceding-sibling::b]

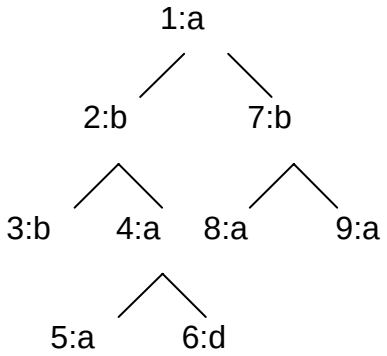


h) //*[count(*)>1 and not(child::*[not(self::a)])]

Answer: 7

(6)**[4]** For the tree T on the right, write numbers of nodes selected by the following XPath expressions.

a) //a
b) /*//*//a[preceding::a]
c) //*[.//d]
d) /*[not(a and b)]
e) //*[count(.//*)=count(ancestor::*)]
f) //c[position()=last()]
g) /descendant:*[position() mod 2 = count(.//*)]
h) //*[count(*)>1 and not(child::*[not(self::a)])]
i) //*[preceding-sibling::b]



i) //*[preceding-sibling::b]

Answer: 4,7

(3)**[4.5]** Given a DAG as
dag(node id)=List(node id’s) and label(node id)=String

(a)write pseudo code that prints in XML format the tree that is represented by the dag. For instance, if 1:a, 2:b[1,1,1] is your dag, then your code should print <a><a><a>

(b) given two dags, dag1 and dag2 (both not necessarily minimal!), write pseudo code that checks whether the trees represented by dag1 and dag2 are equal. Your program should NOT decompress both dags, and then check equality of the strings; instead, your program should run in linear time with respect to the sum of sizes of dag1 and dag2!

---

(3)**[4.5]** Given a DAG as
dag(node id)=List(node id's) and label(node id)=String

(a)write pseudo code that prints in XML format the tree that is represented by the dag. For instance, if 1:a, 2:b[1,1,1] is your dag, then your code should print <a><a><a>

(b) given two dags, dag1 and dag2 (both not necessarily minimal!), write pseudo code that checks whether the trees represented by dag1 and dag2 are equal. Your program should NOT decompress both dags, and then check equality of the strings; instead, your program should run in linear time with respect to the sum of sizes of dag1 and dag2!

(a) Also given:

d.root

(the node-id of the dag's root node)

(3)**[4.5]** Given a DAG as
dag(node id)=List(node id's) and label(node id)=String

(a)write pseudo code that prints in XML format the tree that is represented by the dag. For instance, if 1:a, 2:b[1,1,1] is your dag, then your code should print <a><a><a>

(b) given two dags, dag1 and dag2 (both not necessarily minimal!), write pseudo code that checks whether the trees represented by dag1 and dag2 are equal. Your program should NOT decompress both dags, and then check equality of the strings; instead, your program should run in linear time with respect to the sum of sizes of dag1 and dag2!

(a) Also given:

d.root

(the node-id of the dag's root node)

```
void printNODE(int id){

    print("<" + label(id) + ">");
    List l = dag(id);
    for(int i = 0; i < l.length ; i++){
    printNODE( l.nth(i));
    }

    print("</" + label(id) + ">");
}

void printDAG(DAG d) {
    printNODE(d.root);
}
```

(3)**[4.5]** Given a DAG as
dag(node id)=List(node id's) and label(node id)=String

(b) given two dags, dag1 and dag2 (both not necessarily minimal!), write pseudo code that checks whether the trees represented by dag1 and dag2 are equal. Your program should NOT decompress both dags, and then check equality of the strings; instead, your program should run in linear time with respect to the sum of sizes of dag1 and dag2!

(b) Idea:

Hash table
Key=
nodeID of dag_1
Values=
Sets of nodes of Dag_2

```
void equalNODE(int id1, int id2) {
  Set s = table.find(id1);
  if (s != null && s.member(id2))
       return;    // id1 and id2 point to equal dags.
  if (label(id1) != label(id2))
     throw NotEqualException;
  List l1 = dag(id1);
  List l2 = dag(id2);
  if (l1.length != l2.length)
    throw NotEqualException;

  for(int i = 0; i < l.length; i++)
      equalNode(l1.nth(i), l2.nth(i));
  if (s == null)
     s = new Set;
  s.add(id2);
  table.add(id1, s.add(id2));

  return;
}
```

(4)**[4]** Consider a (pre,post) table: Given a pre-order number x,
the mapping post(x) returns the post-order of the node with pre-order x.
Write pseudo code that, for a node p, prints pre-numbers of
a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute
in an optimal way all the preceding nodes of p_1,…,p_n.

Let pre be numbered

1,2,3,..,MaxPre

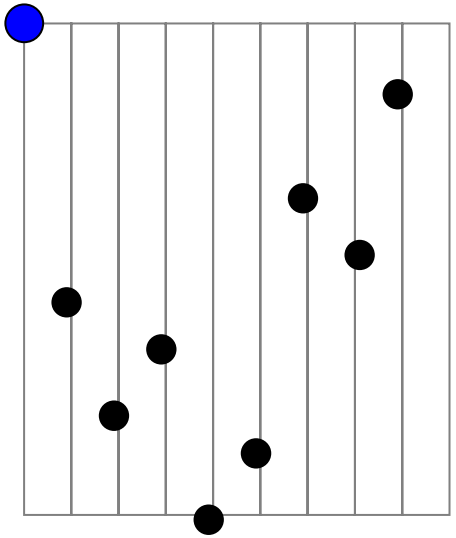
```
void printDescendants(int pre){

    for(int i = pre+1; (i<=MaxPre &&
                        post(i)<post(pre)); i++)
      print(i);
}
```

(4)**[4]** Consider a (pre,post) table: Given a pre-order number x, the mapping post(x) returns the post-order of the node with pre-order x. Write pseudo code that, for a node p, prints pre-numbers of

a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute in an optimal way all the preceding nodes of p_1,…,p_n.

Let pre be numbered

1,2,3,..,MaxPre



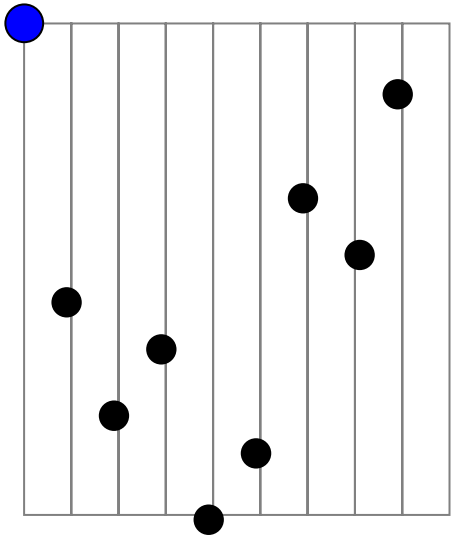
```
void printChildren(int pre){

    int barrier = 0;

    for(int i = pre+1; (i<=MaxPre &&
                post(i)<post(pre)); i++)

      if(post(i)>barrier){
        print(i);
        barrier = post(i)
      }
}
```

(4)**[4]** Consider a (pre,post) table: Given a pre-order number x, the mapping post(x) returns the post-order of the node with pre-order x. Write pseudo code that, for a node p, prints pre-numbers of

a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute in an optimal way all the preceding nodes of p_1,…,p_n.

Let pre be numbered

1,2,3,..,MaxPre



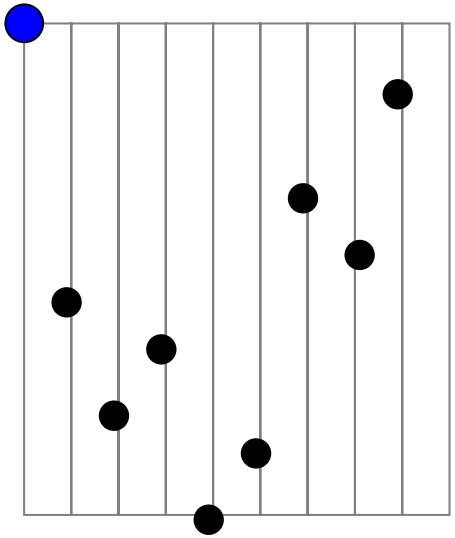
```
void followingSiblings(int pre){

    int barrier = post(pre);

    for(int i = pre+1; i<=MaxPre; i++)

     if(post(i)>barrier){
        print(i);
        barrier = post(i)
      }
}
```

(4)**[4]** Consider a (pre,post) table: Given a pre-order number x, the mapping post(x) returns the post-order of the node with pre-order x. Write pseudo code that, for a node p, prints pre-numbers of

a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute in an optimal way all the preceding nodes of p_1,…,p_n.

Let pre be numbered

1,2,3,..,MaxPre



```
void twoAway(int pre){

    for(int i = 1; i<=MaxPre; i++)

     if(!isChild(i,pre) &&
        !isChild(pre,i))
       print(i);
}
```

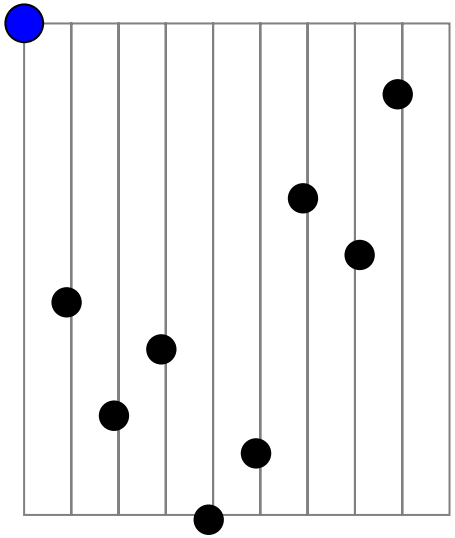
→ Descendants of children
→ (Following plus Preceding) without parent

(4)**[4]** Consider a (pre,post) table: Given a pre-order number x, the mapping post(x) returns the post-order of the node with pre-order x. Write pseudo code that, for a node p, prints pre-numbers of

a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute in an optimal way all the preceding nodes of p_1,…,p_n.

Let pre be numbered

1,2,3,..,MaxPre



```
void twoAway(int pre){

    for(int i = 1; i<=MaxPre; i++)

     if(!isChild(i,pre) &&
        !isChild(pre,i))
       print(i);
}
Boolean isChild(int a, int b){
  for(int i=a--; i>b; i--)
    if(post(i)>post(a)) return false
  return true
}
```

→ Descendants of children
→ (Following plus Preceding) without parent

(4)**[4]** Consider a (pre,post) table: Given a pre-order number x,
the mapping post(x) returns the post-order of the node with pre-order x.
Write pseudo code that, for a node p, prints pre-numbers of
a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute
in an optimal way all the preceding nodes of p_1,…,p_n.

---

f) Cave! It is *not* correct to take the node with largest pre-value (p_n), and to compute the preceding nodes of that!