

XML and Databases

Lecture 3

XML into RDBMS

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2009

Lecture 3

XML Into RDBMS

Memory Representations

Facts

- DOM is easy to use, but memory heavy.
in-memory size usually **5-10 times larger** than size of original file.
- SAX is very flexible.
Using arrays or binary trees wo backward pointers,
in-memory size is **approx. same** as size of original file.
- Can be further improved using DAGs/sharing-Graphs
& coding/compression for data values.

Memory Representations

Facts

- DOM is easy to use, but memory heavy.
in-memory size usually **5-10 times larger** than size of original file.
- SAX is very flexible.
Using arrays or binary trees wo backward pointers,
in-memory size is **approx. same** as size of original file.
- Can be further improved using DAGs/sharing-Graphs
& coding/compression for data values.

TODAY

- ➔ How can we map XML into a **relational DB**?

Memory Representations

Facts

- DOM is easy to use, but memory heavy.
in-memory size usually **5-10 times larger** than size of original file.
- SAX is very flexible.
Using arrays or binary trees wo backward pointers,
in-memory size is **approx. same** as size of original file.
- Can be further improved using DAGs/sharing-Graphs
& coding/compression for data values.

TODAY

- ➔ How can we map XML into a **relational DB**?
- ... but first: **Memory efficient** *tree traversals* using e.g. DOM?

Traversals and Pre/Post-Encoding

1. Pre-Order Traversal (recursively)
2. Post-Order
3. Pre-Order (iteratively)
4. Into RDBMS with Pre/Post-encoding

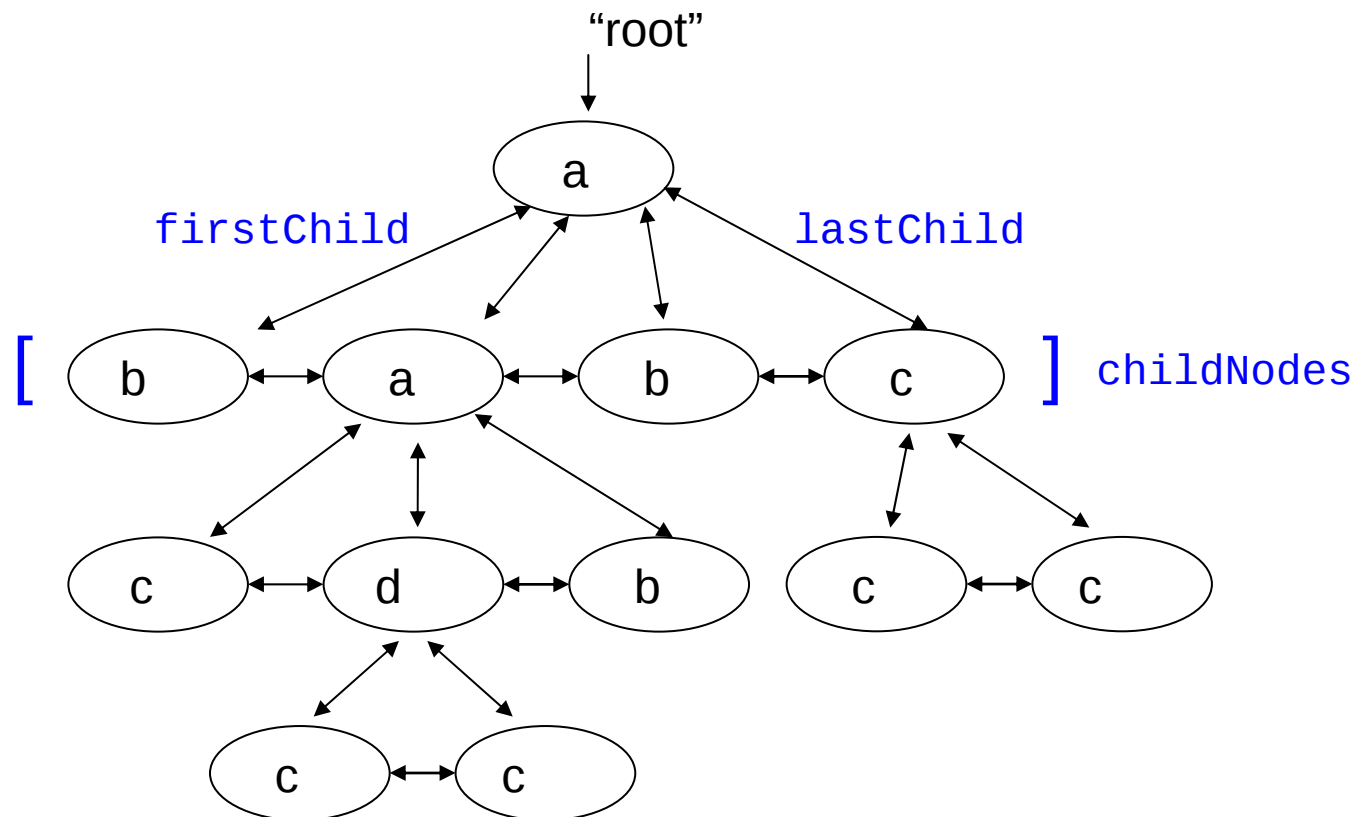
TODAY

➔ How can we map XML into a relational DB?

... but first: **Memory efficient** *tree traversals* using e.g. DOM?

Tree Traversals

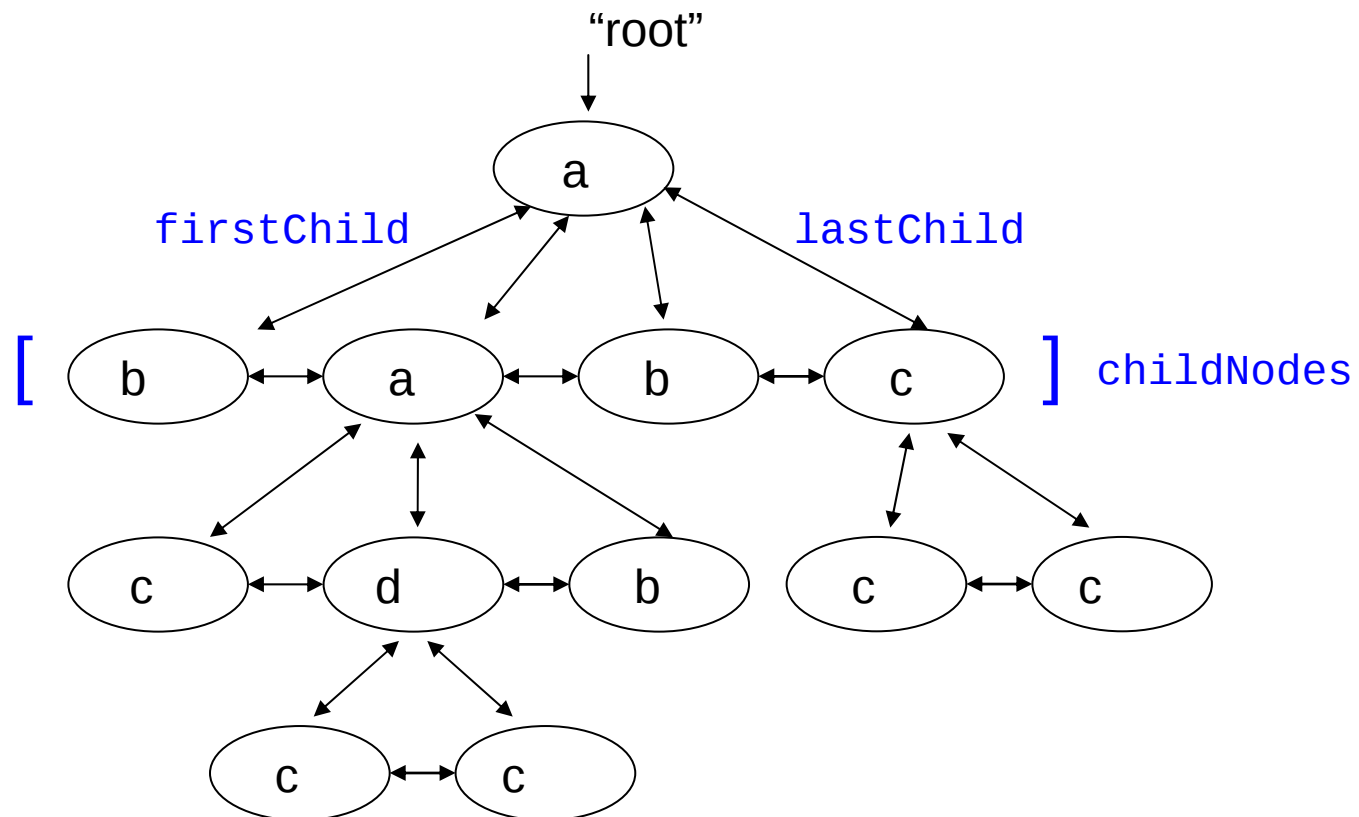
Start at root node; want to visit every node.



Tree Traversals

Start at root node; want to visit every node.

(1) *recursively*



Tree Traversals

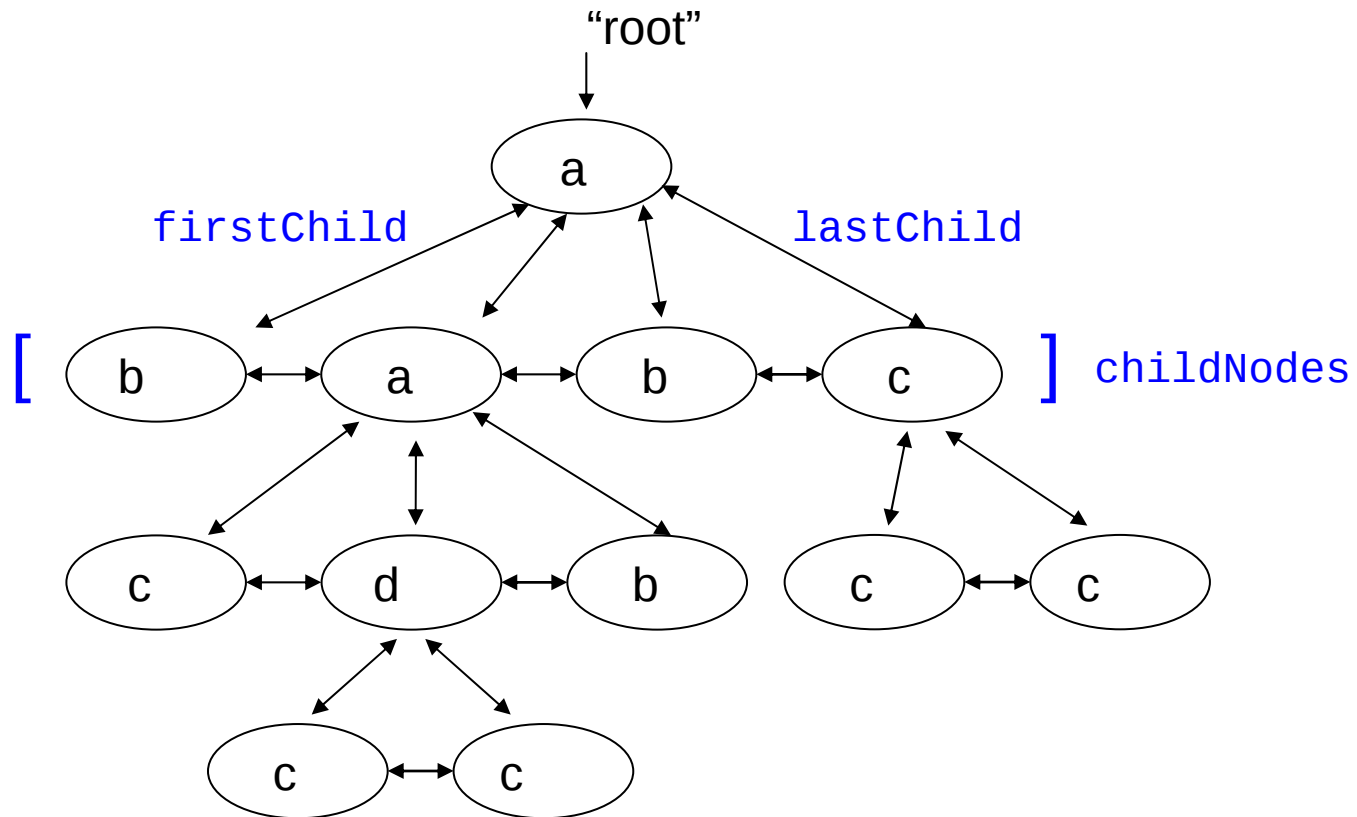
Start at root node; want to visit every node.

(1) *recursively*

```

 Traverse(n:Node){
   print(n);
   For m in childNodes(n) Traverse(m)
 }

```



Tree Traversals

Start at root node; want to visit every node.

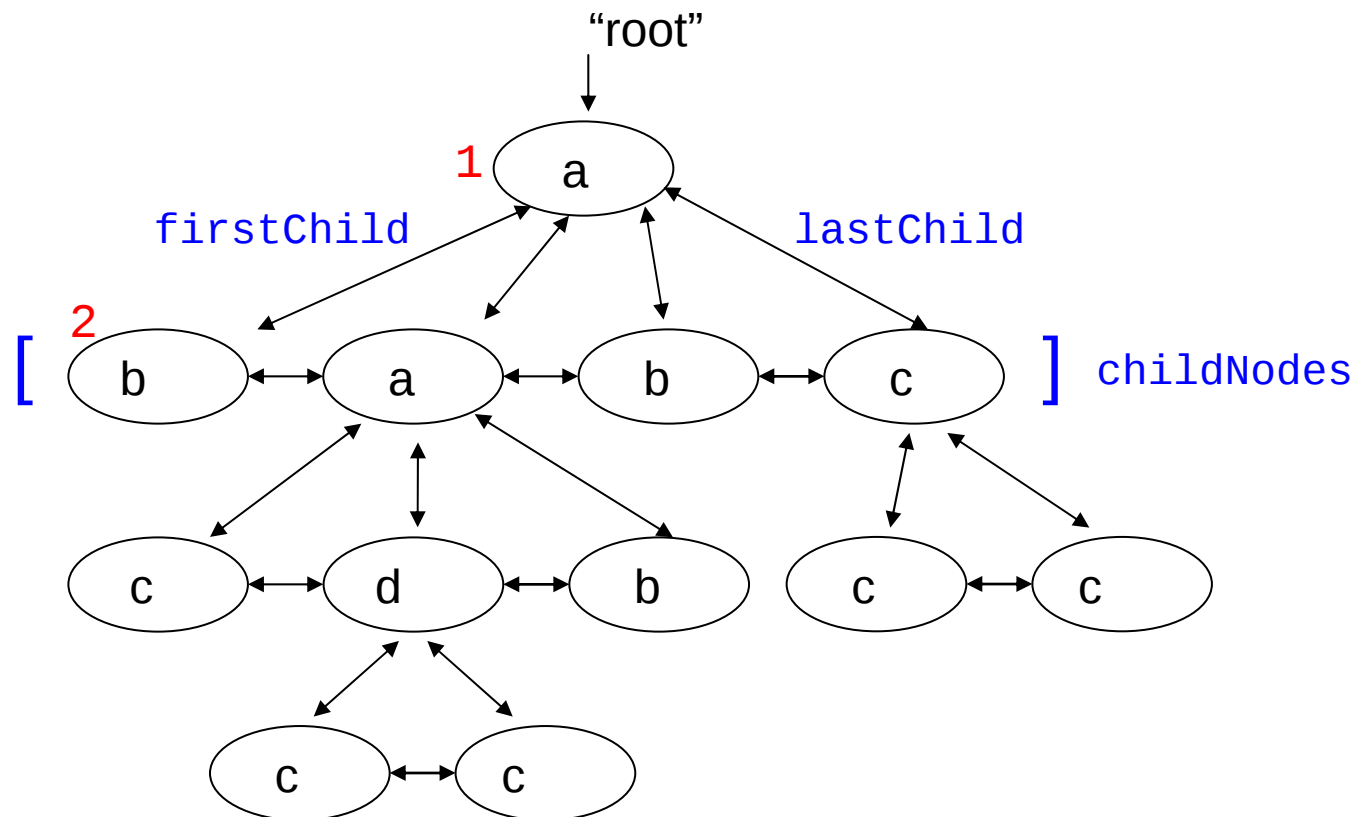
Tr(**1**)
 pr(**1**)
 Tr(**2**)
 pr(**2**)

(1) *recursively*

```

  Traverse(n:Node){
    print(n);
    For m in childNodes(n) Traverse(m)
  }

```



Tree Traversals

Start at root node; want to visit every node.

(1) *recursively*

```

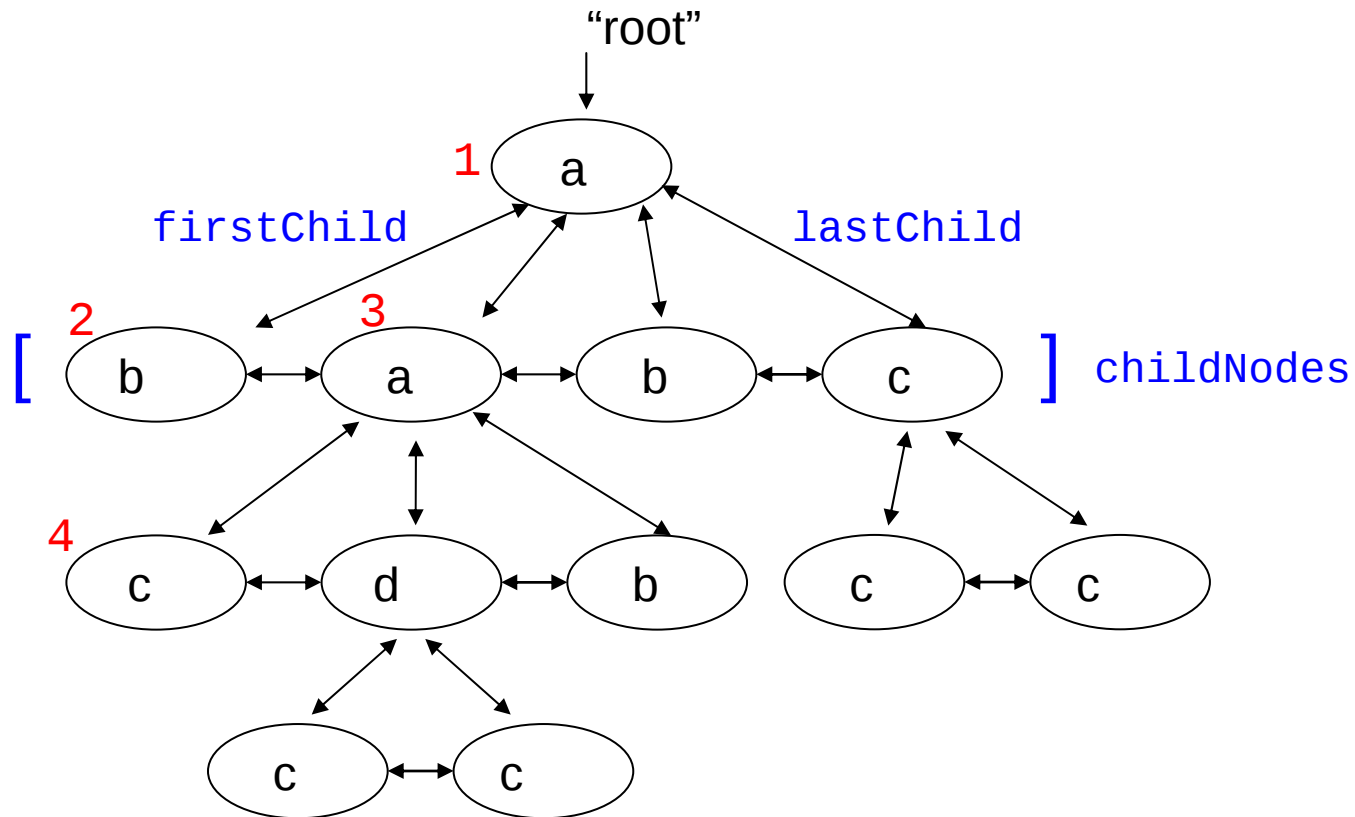
 Traverse(n:Node){
   print(n);
   For m in childNodes(n) Traverse(m)
 }

```

```

Tr(1)
  pr(1)
  Tr(2)
    pr(2)
    Tr(3)
      pr(3)
      Tr(4)
        pr(4)

```



Tree Traversals

Start at root node; want to visit every node.

(1) *recursively*

```

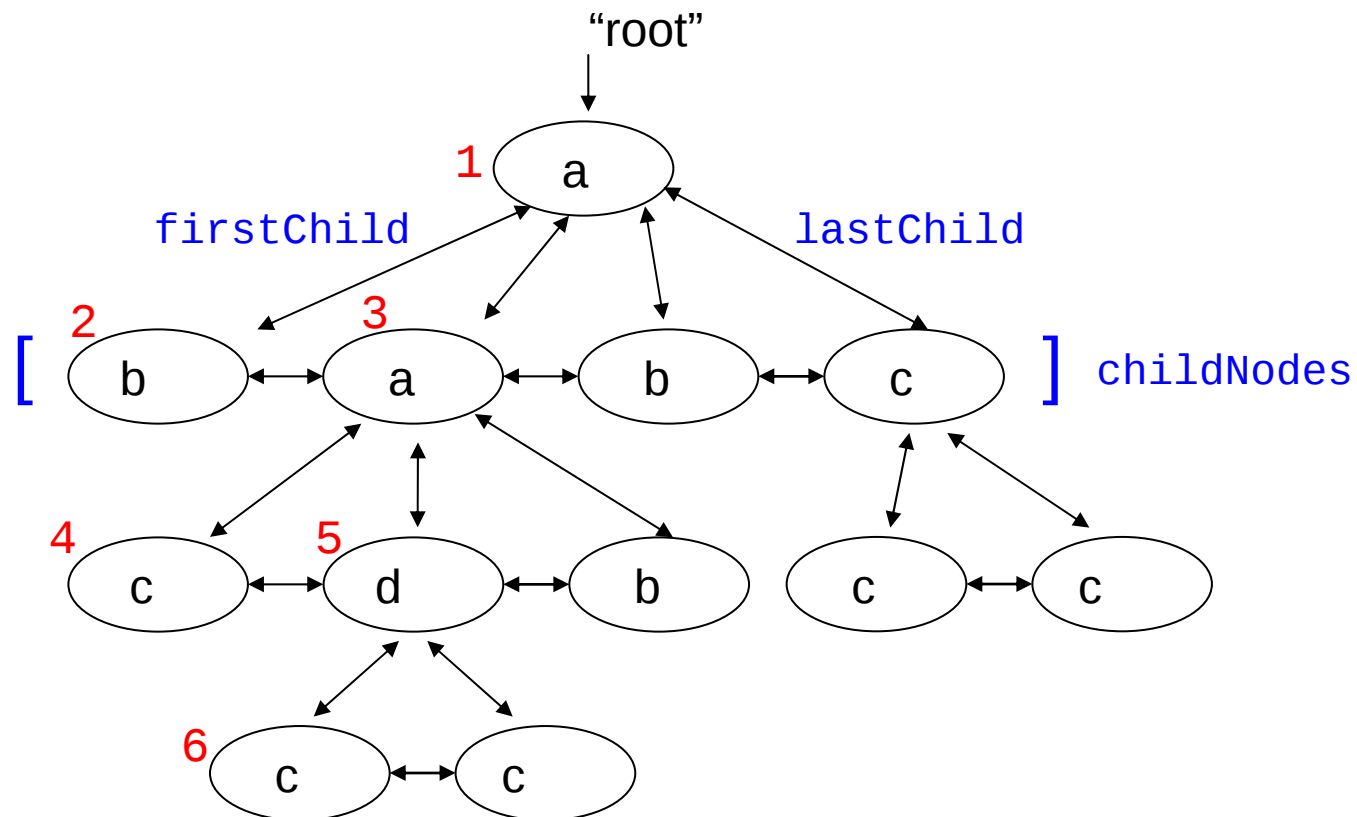
 Traverse(n:Node){
   print(n);
   For m in childNodes(n) Traverse(m)
 }

```

```

Tr(1)
  pr(1)
Tr(2)
  pr(2)
Tr(3)
  pr(3)
Tr(4)
  pr(4)
Tr(5)
  pr(5)
Tr(6)
  pr(6)

```



Tree Traversals

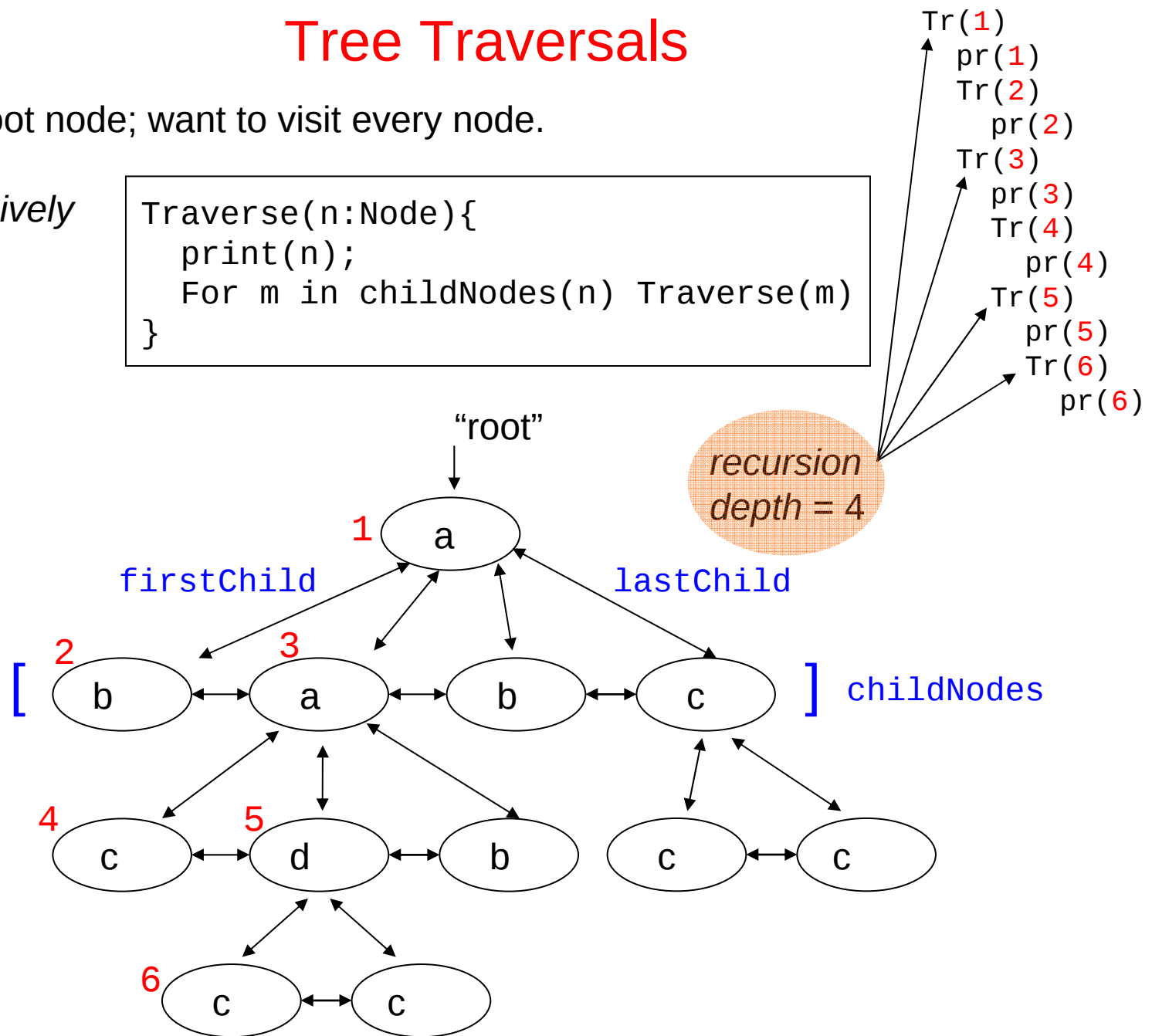
Start at root node; want to visit every node.

(1) *recursively*

```

Traverse(n:Node){
  print(n);
  For m in childNodes(n) Traverse(m)
}

```



Tree Traversals

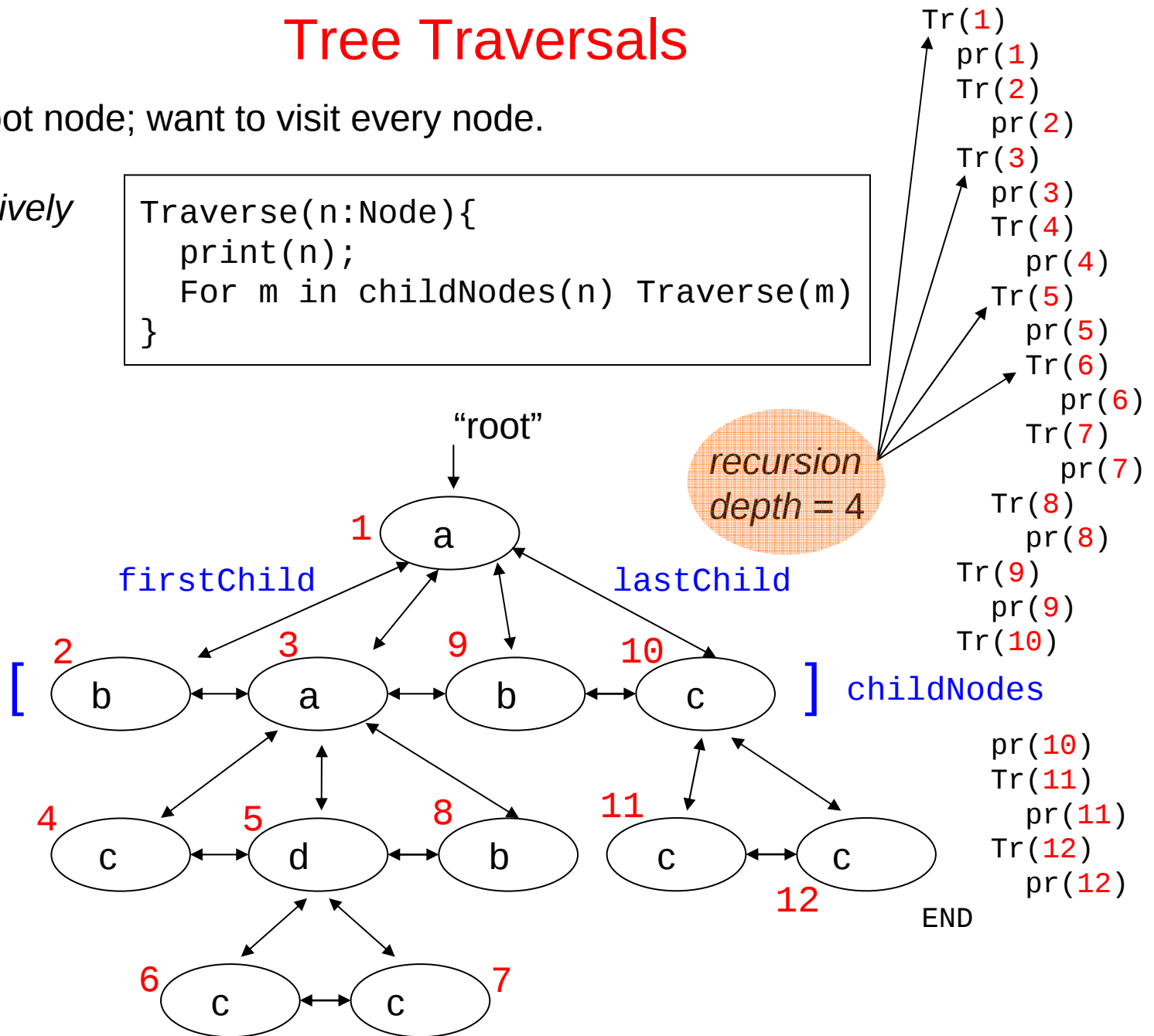
Start at root node; want to visit every node.

(1) *recursively*

```

 Traverse(n:Node){
   print(n);
   For m in childNodes(n) Traverse(m)
 }

```



Tree Traversals

Start at root node; want to visit every node.

(1) *recursively*

```
Traverse(n:Node){  
    print(n);  
    For m in childNodes(n) Traverse(m)  
}
```

Memory need
proportional
to the
height of the
XML tree.

→ Should be fine.

Usually **height** (XML doc) is small. (≤ 15)

Tree Traversals

Start at root node; want to visit every node.

(1) *recursively*

```

Traverse(n:Node){
  print(n);
  For m in childNodes(n) Traverse(m)
}

```

Memory need
proportional
to the
height of the
XML tree.

→ Should be fine.

Usually **height** (XML doc) is small. (≤ 15)

Problematic

2nd recursion on children

```

TR(n:Node){
  print(n);
  if(m=firstChild(n))!=NIL then Tr(m);
  if(m=nextSibling(n))!=NIL then Tr(m)
}

```

Tree Traversals

Start at root node; want to visit every node.

(1) *recursively*

```

Traverse(n:Node){
  print(n);
  For m in childNodes(n) Traverse(m)
}

```

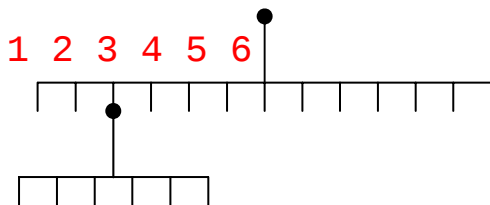
Memory need
proportional
to the
height of the
XML tree.

→ Should be fine.

Usually **height** (XML doc) is small. (≤ 15)

Problematic

2nd recursion on children



```

TR(n:Node){
  print(n);
  if(m=firstChild(n))!=NIL then Tr(m);
  if(m=nextSibling(n))!=NIL then Tr(m)
}

```

Tree Traversals

Start at root node; want to visit every node.

(1) *recursively*

```

 Traverse(n:Node){
   print(n);
   For m in childNodes(n) Traverse(m)
 }

```

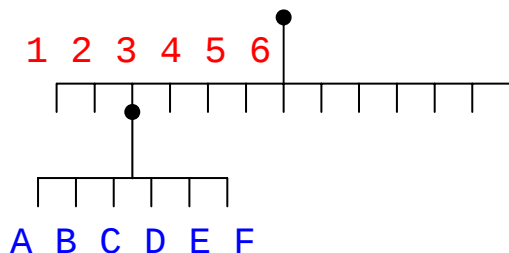
Memory need
proportional
to the
height of the
XML tree.

→ Should be fine.

Usually **height** (XML doc) is small. (≤ 15)

Problematic

2nd recursion on children



```

 TR(n:Node){
   print(n);
   if(m=firstChild(n))!=NIL then TR(m);
   if(m=nextSibling(n))!=NIL then TR(m)
 }

```

```

 TR(1);pr(1)
   TR(2);pr(2)
     TR(3);pr(3)
       TR(A);pr(A)
         TR(B);pr(B)

```

Tree Traversals

Start at root node; want to visit every node.

(1) *recursively*

```

Traverse(n:Node){
  print(n);
  For m in childNodes(n) Traverse(m)
}

```

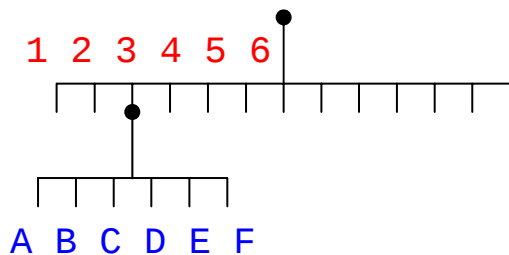
Memory need
proportional
to the
height of the
XML tree.

→ Should be fine.

Usually **height** (XML doc) is small. (≤ 15)

Problematic

2nd recursion on children



```

TR(n:Node){
  print(n);
  if(m=firstChild(n))!=NIL then Tr(m);
  if(m=nextSibling(n))!=NIL then Tr(m)
}

```

```

TR(1);pr(1)
  TR(2);pr(2)
    TR(3);pr(3)
      TR(A);pr(A)
        TR(B);pr(B)

```

Memory need
proportional to max. length of
(firstChild | nextSibling)*-path

Tree Traversals

Start at root node; want to visit every node.

(1) *recursively*

```

Traverse(n:Node){
  print(n);
  For m in childNodes(n) Traverse(m)
}

```

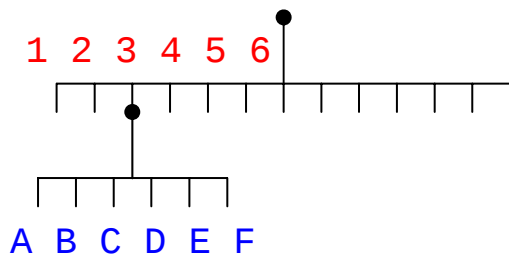
Memory need
proportional
to the
height of the
XML tree.

→ Should be fine.

Usually **height** (XML doc) is small. (≤ 15)

Problematic

2nd recursion on children



```

Tr(n:Node){
  print(n);
  if(m=firstChild(n))!=NIL then Tr(m);
  if(m=nextSibling(n))!=NIL then Tr(m)
}

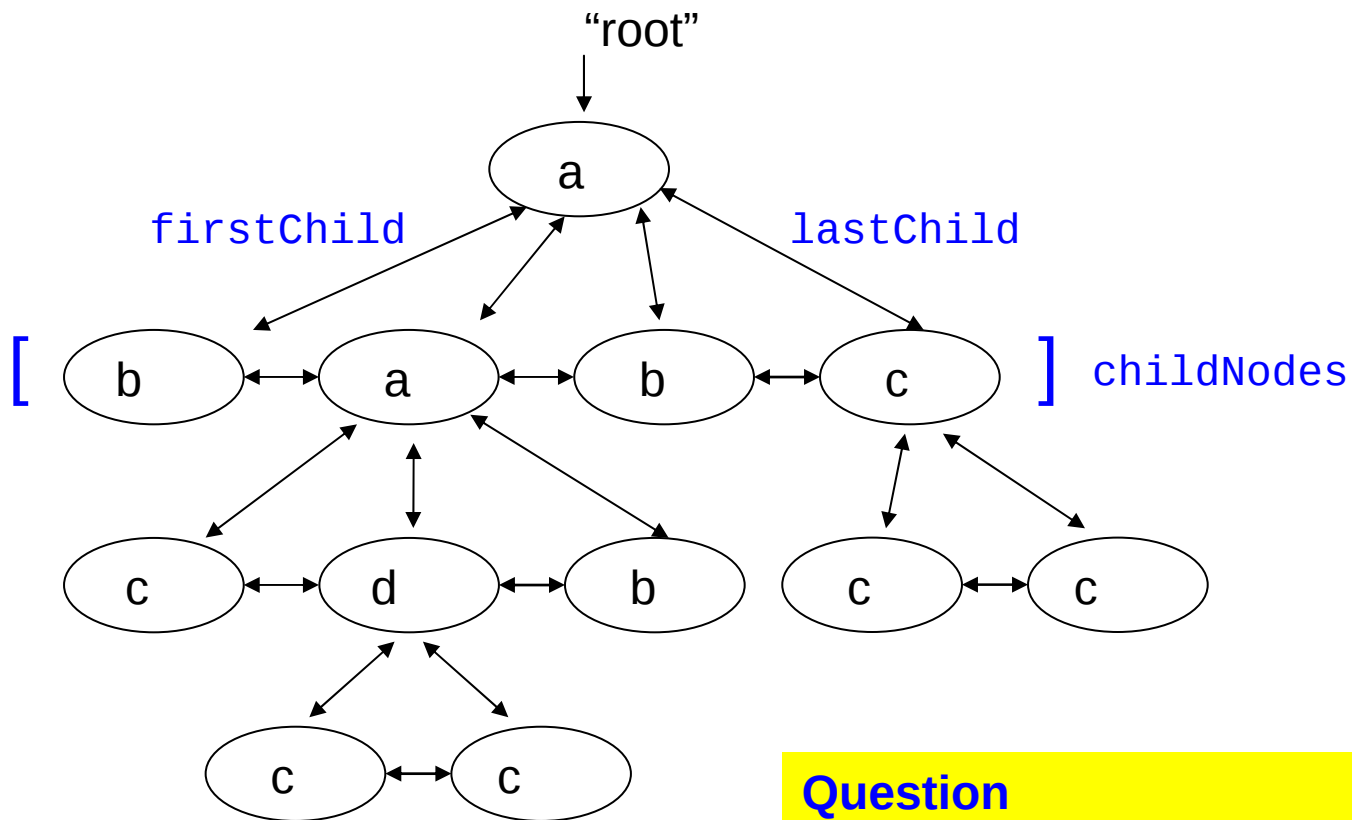
```

```

TR(1);pr(1)
  TR(2);pr(2)
    TR(3);pr(3)
      TR(A);pr(A)
        TR(B);pr(B)

```

Can be
HUGE!!!
=size(tree)
Memory need
proportional to max. length of
(firstChild | nextSibling)*-path



Question

What is the max recursion depth on this tree?

```
Tr(n:Node){
  print(n);
  if(m=firstChild(n))!=NIL then Tr(m);
  if(m=nextSibling(n))!=NIL then Tr(m)
}
```

Memory need proportional to max. length of (firstChild | nextSibling)*-path

Can be HUGE!!!
=size(tree)

Tree Traversals

Start at root node; want to visit every node.

(1) *recursively*

```

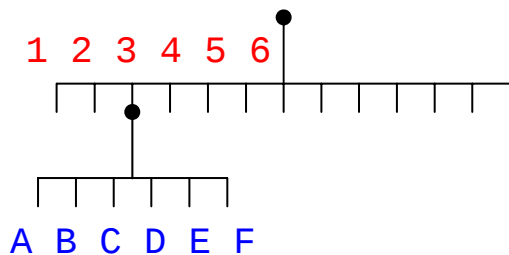
    Traverse(n:Node){
      print(n);
      For m in childNodes(n) Traverse(m)
    }
```

Memory need
proportional
to the
height of the
XML tree.

→ Should be fine.
Usually **height** (XML doc) is small. (≤ 15)

Problematic

2nd recursion on children



Recall binary tree (firstChild/nextSibling) encoding.

Question

In the binary tree, what corresponds to this number?

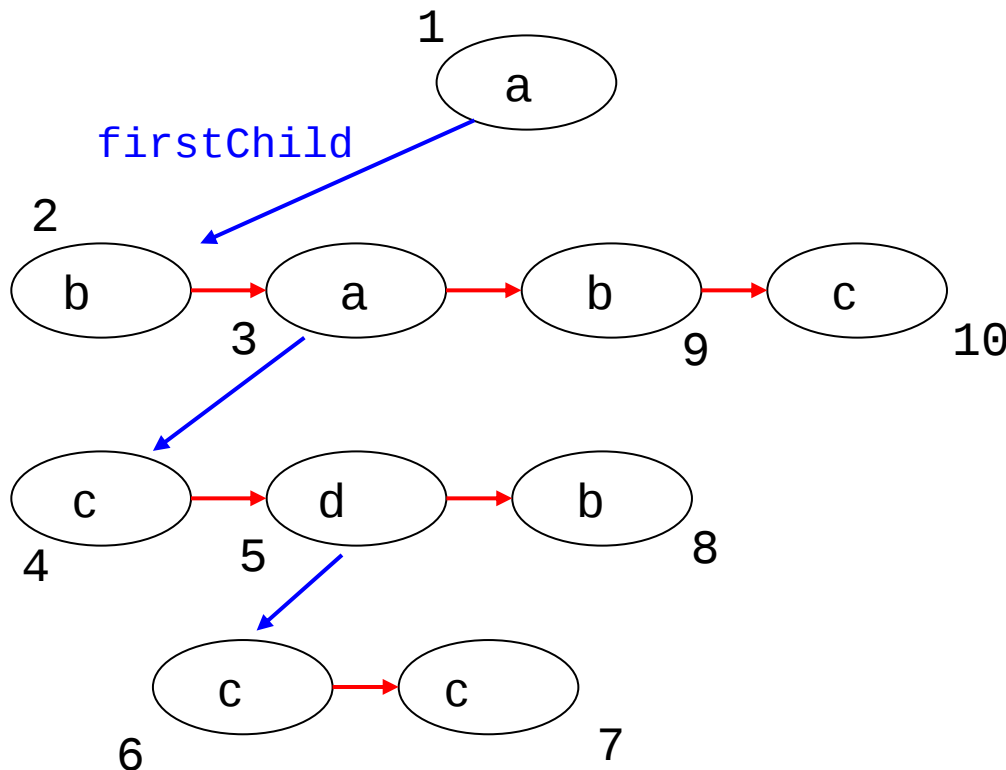
TR(1);pr(1)
TR(2);pr(2)
TR(3);pr(3)
TR(A);pr(A)
TR(B);pr(B)

max. length of
(firstChild | nextSibling)*-path

Binary Tree Encoding

Recall “firstChild/nextSibling” encoding.

The “firstChild” becomes the **left** pointer
 The “nextSibling” becomes the **right** pointer

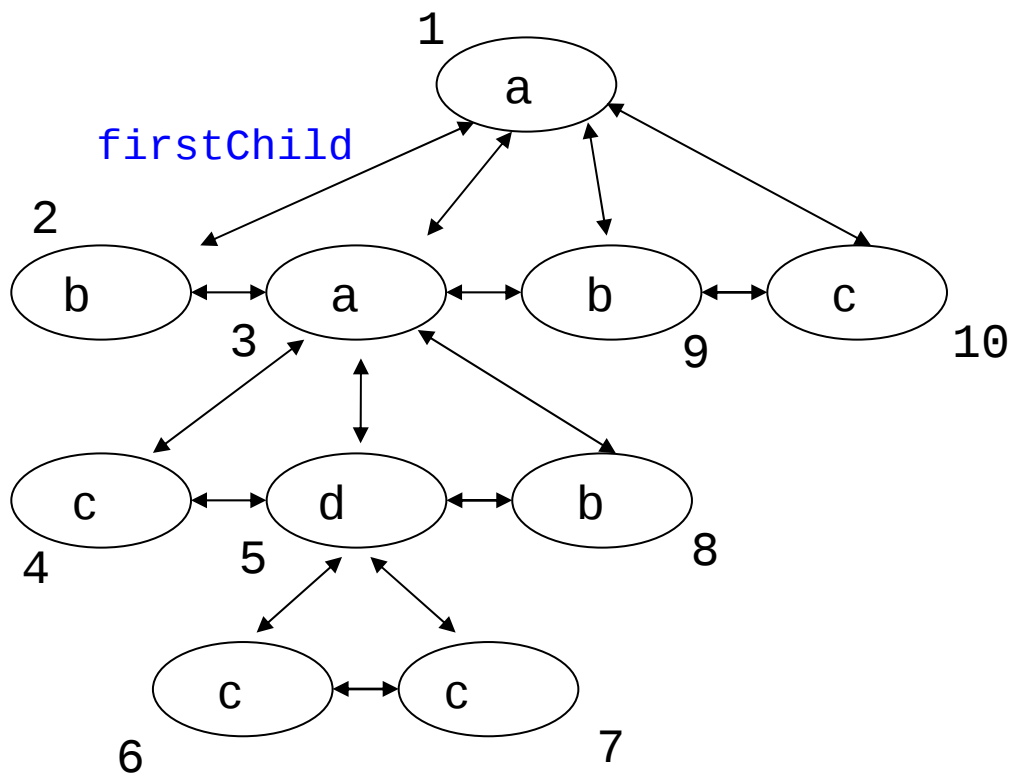


per Node
 2 pointers/IDs
 + label info

ID	fc:ns:lab
1	(2:-:a)
2	(-:3:b)
3	(4:9:a)
4	(-:5:c)
5	(6:8:d)
6	(-:7:c)
7	(-:-:c)
8	(-:-:b)
9	(-:10:b)
10	(-, -:c)

Tree Traversals

→ both, Traverse and TR can
be executed on the
fc/ns-binary tree encoding.



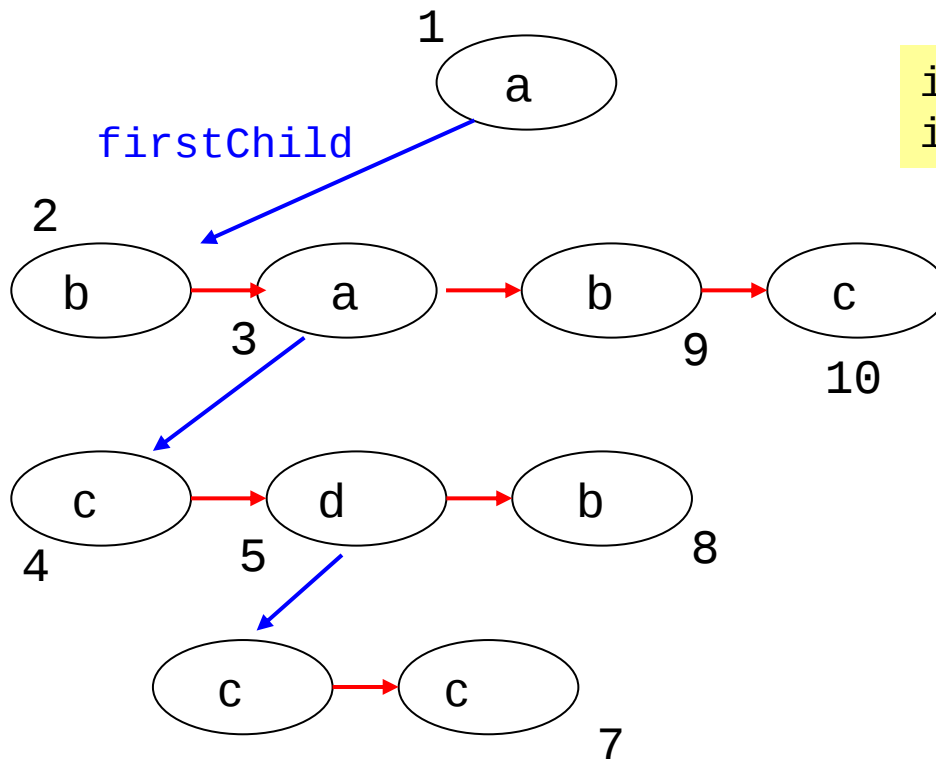
Tree Traversals

→ both, Traverse and Tr can be executed on the **fc/ns**-binary tree encoding.

```
For m in childNodes(n) Traverse(n)
```



```
if(m=n->left) != NIL
{ while(m=n->right) != NIL Traverse(m)
}
```



```
if(m=firstChild(n)) != NIL then TR(m);
if(m=nextSibling(n)) != NIL then TR(m)
```



```
if(m=n->left) != NIL then TR(m)
if(m=n->right) != NIL then TR(m)
```

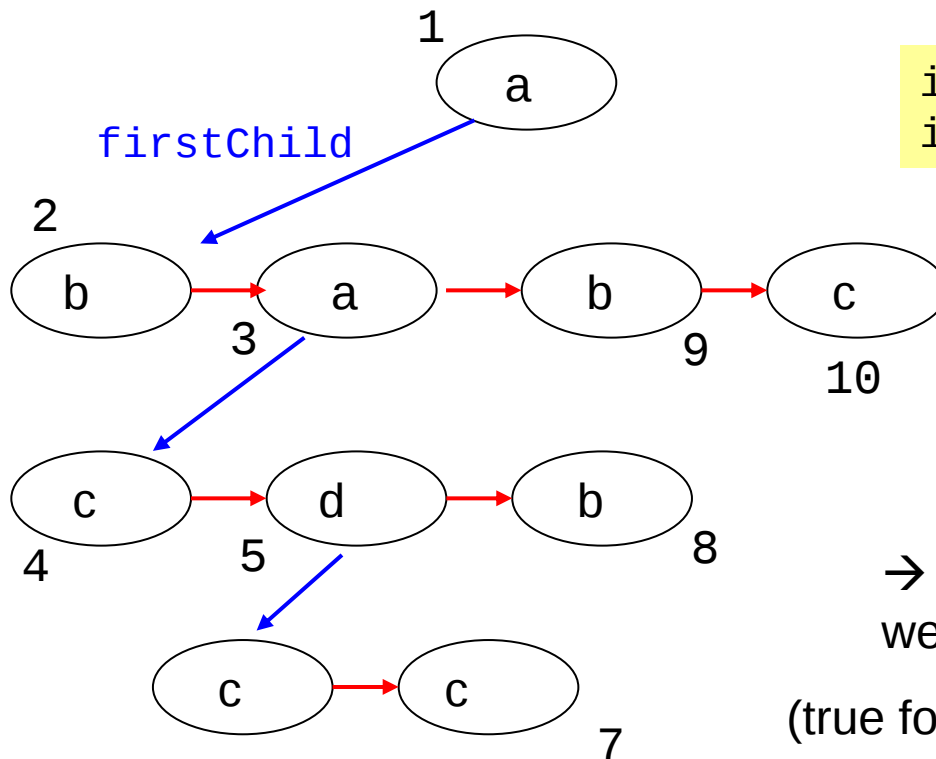
Tree Traversals

→ both, Traverse and Tr can be executed on the **fc/ns**-binary tree encoding.

```
For m in childNodes(n) Traverse(n)
```



```
if(m=n->left) != NIL
{ while(m=n->right) != NIL Traverse(m)
}
```



```
if(m=firstChild(n)) != NIL then Tr(m);
if(m=nextSibling(n)) != NIL then Tr(m)
```



```
if(m=n->left) != NIL then TR(m)
if(m=n->right) != NIL then TR(m)
```

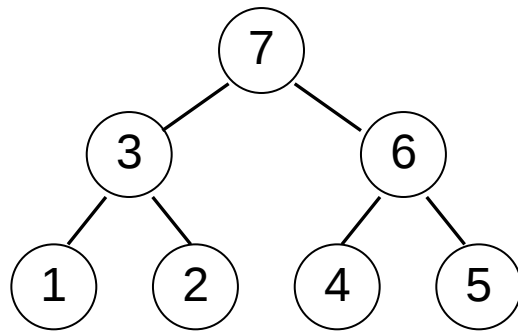
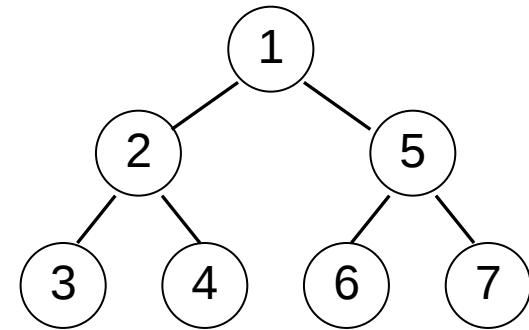
→ Recursion takes care of the fact that we do not have parent pointers. 😊

(true for both, unranked & binary tree)

Other Traversals

We discussed the **Pre-order** of the tree.
(or, in XML-jargon: *document-order*).

Realized by Traverse & TR

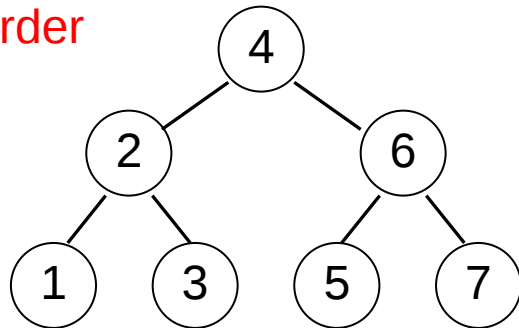


Post-order of a tree =

1. Traverse left subtree in post-order
2. Traverse right subtree in post-order
3. Visit the root

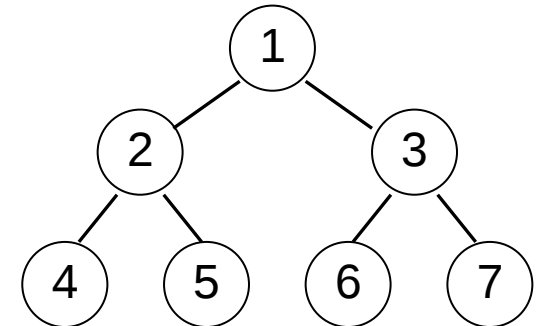
→ Reverse Polish Notation

In-order



→ Binary Search Tree (increasing)

Breadth-First
(left-to-right)
"level-order"



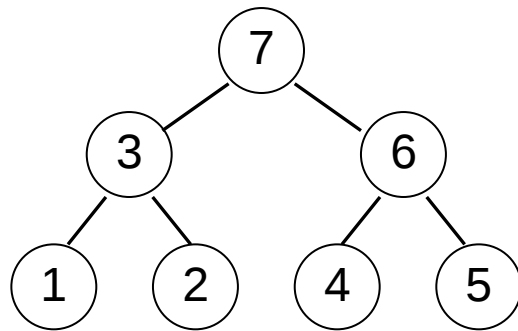
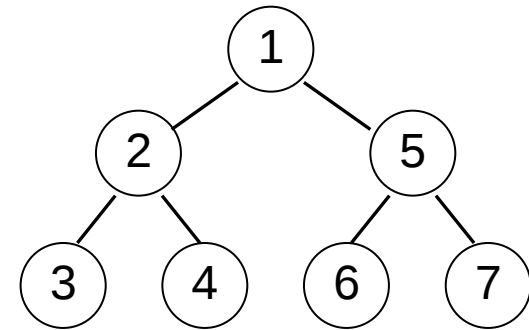
```

q.enq(root);
while(NOT q.empty){ Visit(q.deq);
  If(q->Left!=NIL) q.enq(q->Left)
  If(q->Right!=NIL) q.enq(q->Right)}
  
```

Other Traversals

We discussed the **Pre-order** of the tree.
(or, in XML-jargon: *document-order*).

Realized by Traverse & TR

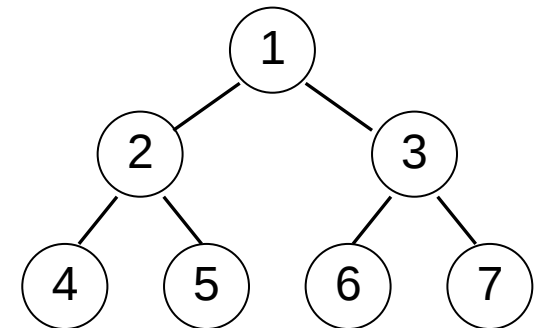


Post-order of a tree =

1. Traverse left subtree in post-order
2. Traverse right subtree in post-order
3. Visit the root

→ Reverse Polish Notation

Breadth-First
(left-to-right)
“level-order”



Memory need
proportional
to

max. #nodes on one level



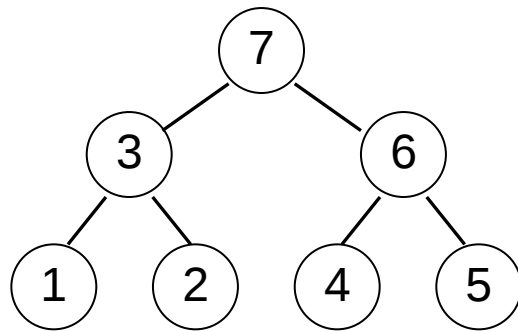
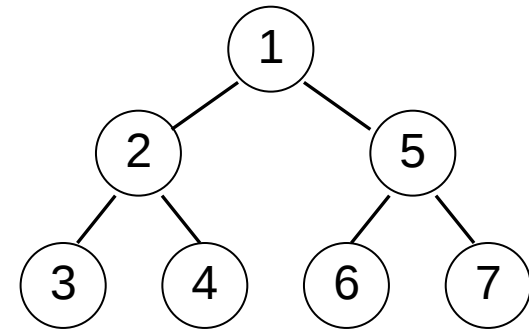
```

q.enq(root);
while(NOT q.empty){ Visit(q.deq);
    If(q->Left!=NIL) q.enq(q->Left)
    If(q->Right!=NIL) q.enq(q->Right)}
  
```

Other Traversals

We discussed the **Pre-order** of the tree.
(or, in XML-jargon: *document-order*).

Realized by Traverse & TR

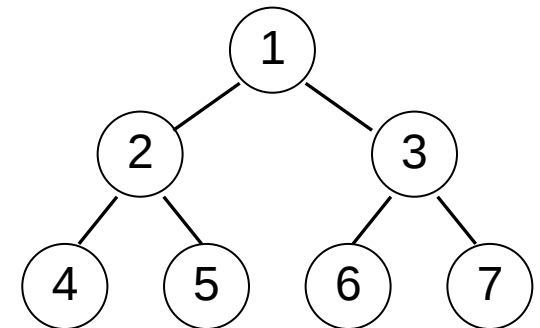


Post-order of a tree =

1. Traverse left subtree in post-order
2. Traverse right subtree in post-order
3. Visit the root

→ Reverse Polish Notation

Breadth-First
(left-to-right)
"level-order"



Memory need
proportional
to

max. #nodes on one level

Can be
HUGE!!!



```

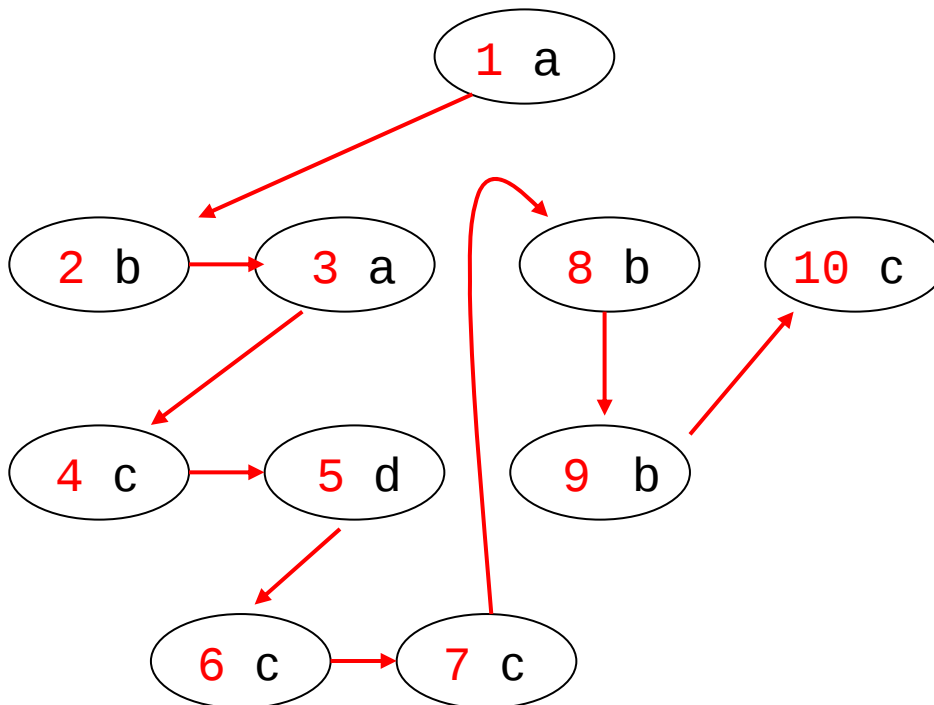
q.enq(root);
while(NOT q.empty){ Visit(q.deq);
  If(q->Left!=NIL) q.enq(q->Left)
  If(q->Right!=NIL) q.enq(q->Right);}
  
```

Pre-Order

We saw how to compute
Pre-order (and **Post** and **In**)

(1) recursively

memory need: $O(\text{max_height})$

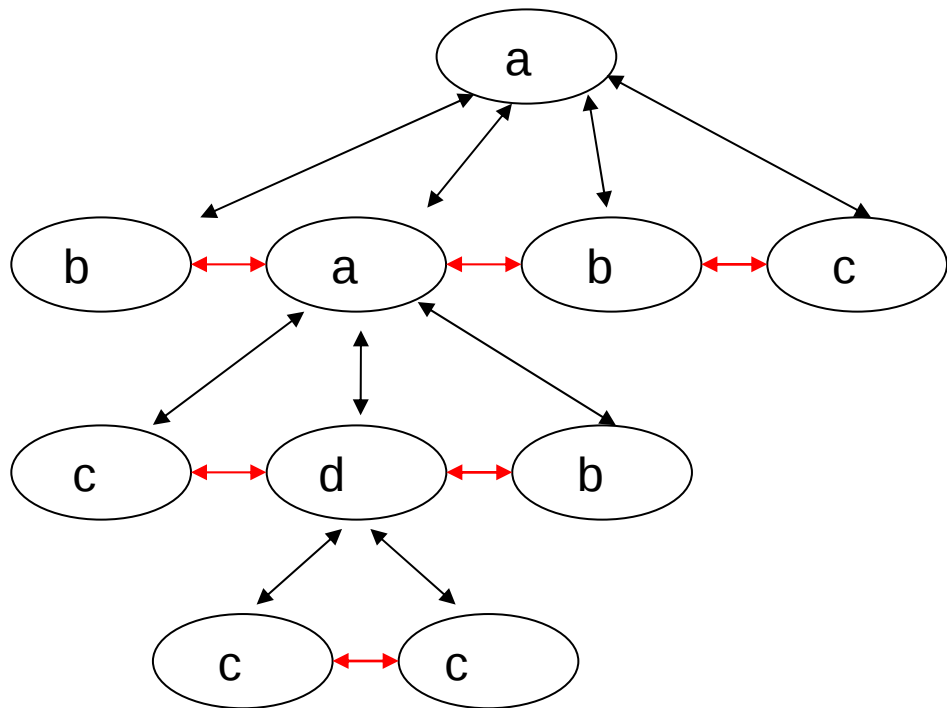


Pre-Order

We saw how to compute
Pre-order (and **Post** and **In**)

(1) recursively

memory need: $O(\text{max_height})$



How to compute Pre-order

(2) *iteratively*

→ Memory need?

```

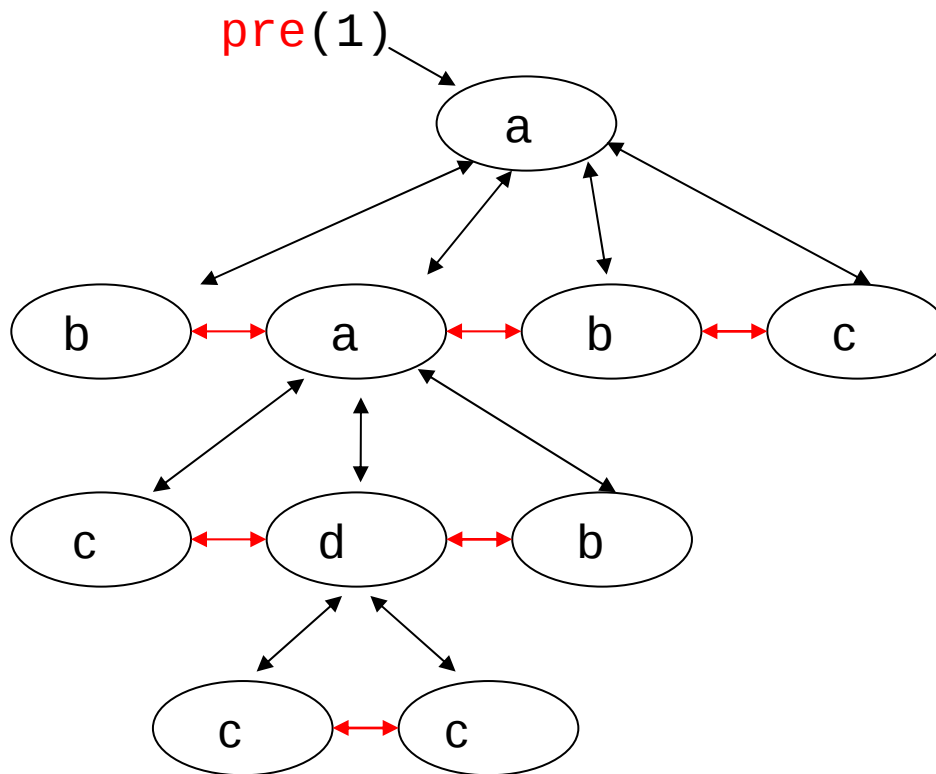
i=1;
n=root;
pre(i)=n;
while(firstChild(n)!=NIL)
{
  n=firstChild(n);
  pre(++i)=n;
}
  
```

Pre-Order

We saw how to compute
Pre-order (and **Post** and **In**)

(1) recursively

memory need: $O(\text{max_height})$



How to compute Pre-order

(2) *iteratively*

→ Memory need?

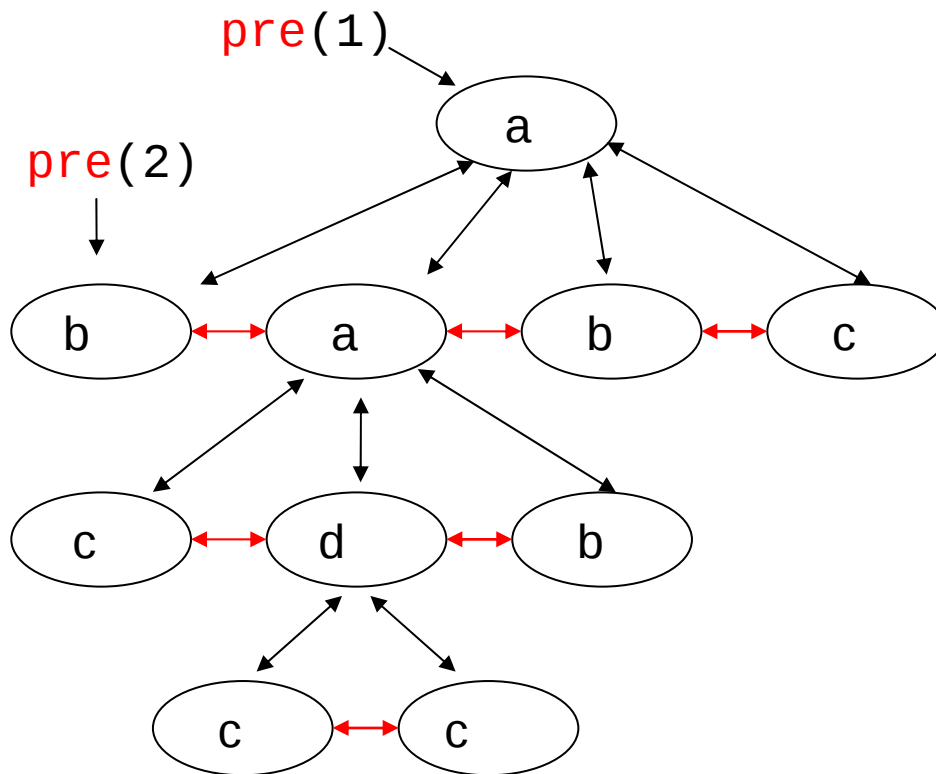
```
i=1;
n=root;
pre(i)=n;
while(firstChild(n)!=NIL)
{
  n=firstChild(n);
  pre(++i)=n;
}
```

Pre-Order

We saw how to compute
Pre-order (and **Post** and **In**)

(1) *recursively*

memory need: $O(\text{max_height})$



How to compute Pre-order

(2) *iteratively*

→ Memory need?

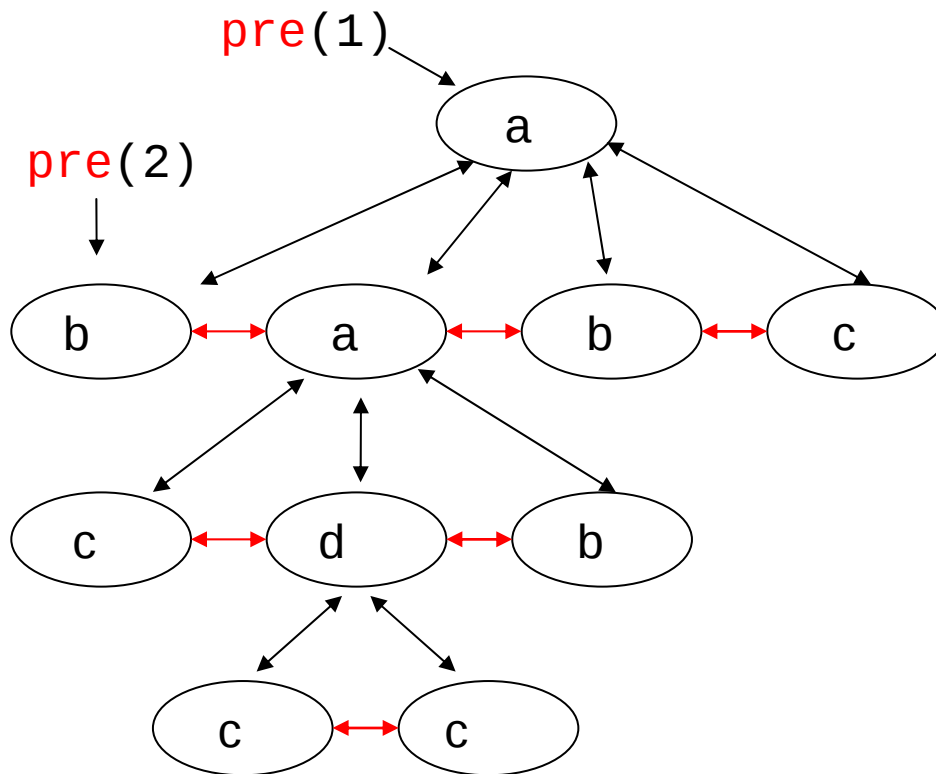
```
i=1;
n=root;
pre(i)=n;
while(firstChild(n)!=NIL)
{
  n=firstChild(n);
  pre(++i)=n;
}
```

Pre-Order

We saw how to compute
Pre-order (and **Post** and **In**)

(1) *recursively*

memory need: $O(\text{max_height})$



How to compute Pre-order

(2) *iteratively*

→ Memory need?

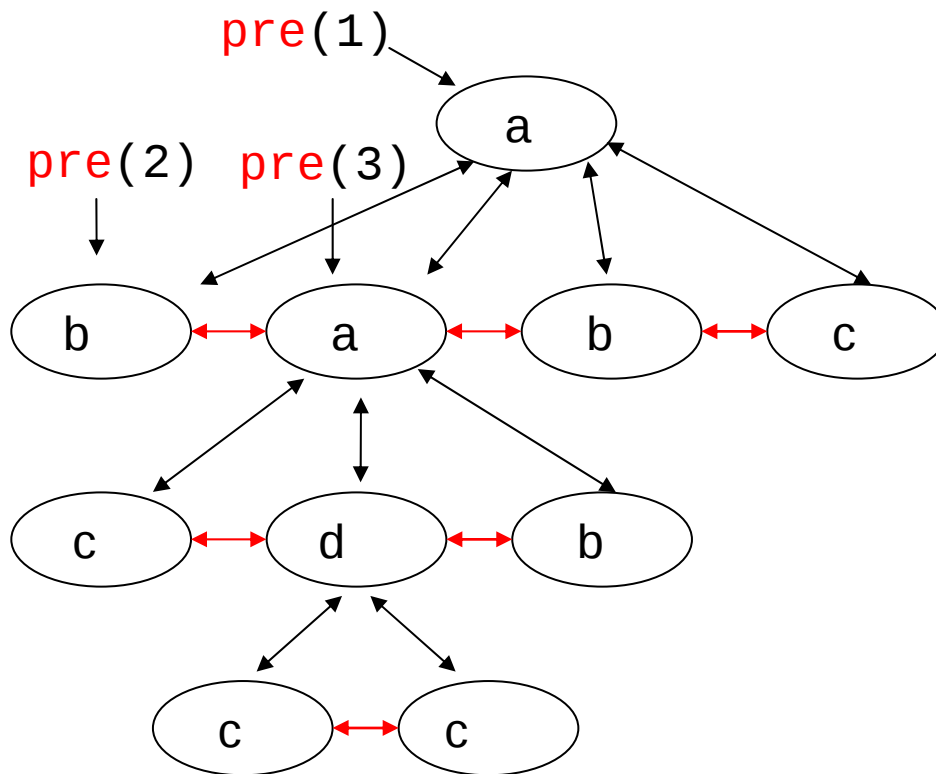
```
i=1;
n=root;
pre(i)=n;
repeat {
    while(firstChild(n)!=NIL)
    { n=firstChild(n);
      pre(++i)=n;
    }
    while(nextSibling(n)=NIL)
    { n=parent(n);}
    n=nextSibling(n);
    pre(++i)=n;
}
```

Pre-Order

We saw how to compute
Pre-order (and Post and In)

(1) *recursively*

memory need: $O(\text{max_height})$



How to compute Pre-order

(2) *iteratively*

→ Memory need?

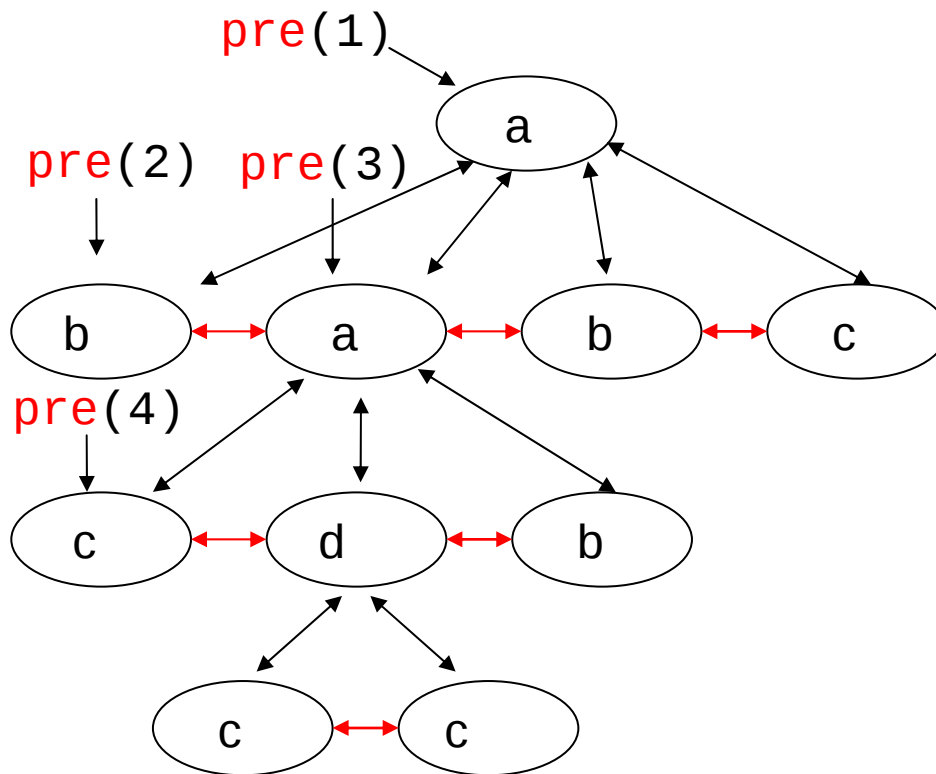
```
i=1;
n=root;
pre(i)=n;
repeat {
    while(firstChild(n)!=NIL)
    {   n=firstChild(n);
        pre(++i)=n;
    }
    while(nextSibling(n)=NIL)
    {   n=parent(n);}
    n=nextSibling(n);
    pre(++i)=n;
}
```

Pre-Order

We saw how to compute
Pre-order (and Post and In)

(1) *recursively*

memory need: $O(\text{max_height})$



How to compute Pre-order

(2) *iteratively*

→ Memory need?

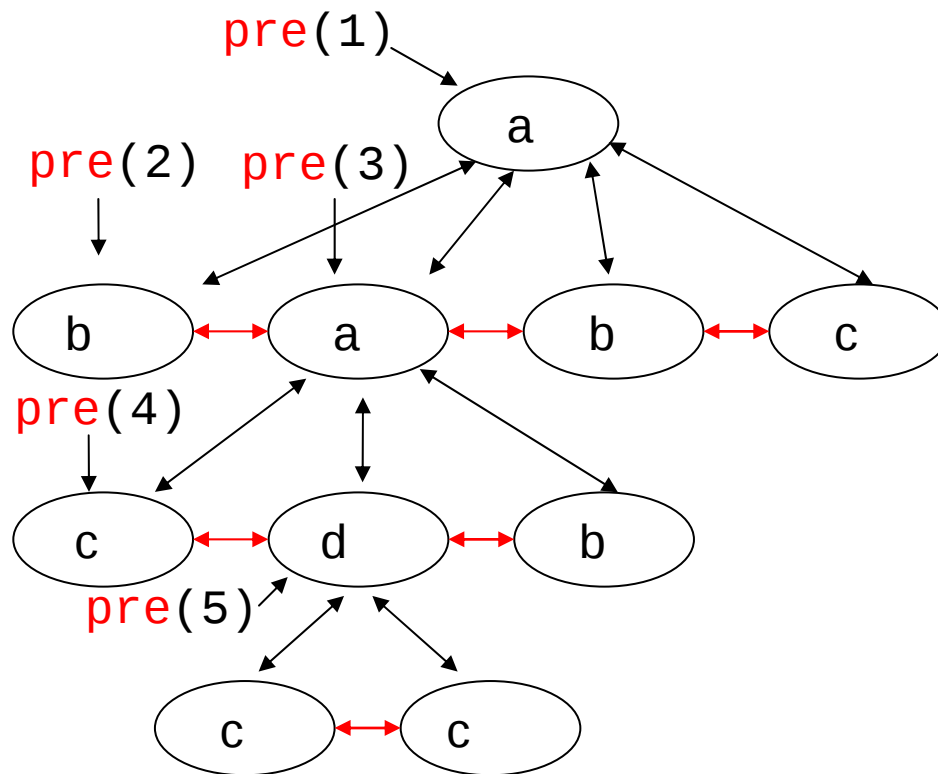
```
i=1;
n=root;
pre(i)=n;
repeat {
    while(firstChild(n)!=NIL)
    { n=firstChild(n);
      pre(++i)=n;
    }
    while(nextSibling(n)=NIL)
    { n=parent(n);}
    n=nextSibling(n);
    pre(++i)=n;
}
```

Pre-Order

We saw how to compute
Pre-order (and Post and In)

(1) *recursively*

memory need: $O(\text{max_height})$



How to compute Pre-order

(2) *iteratively*

→ Memory need?

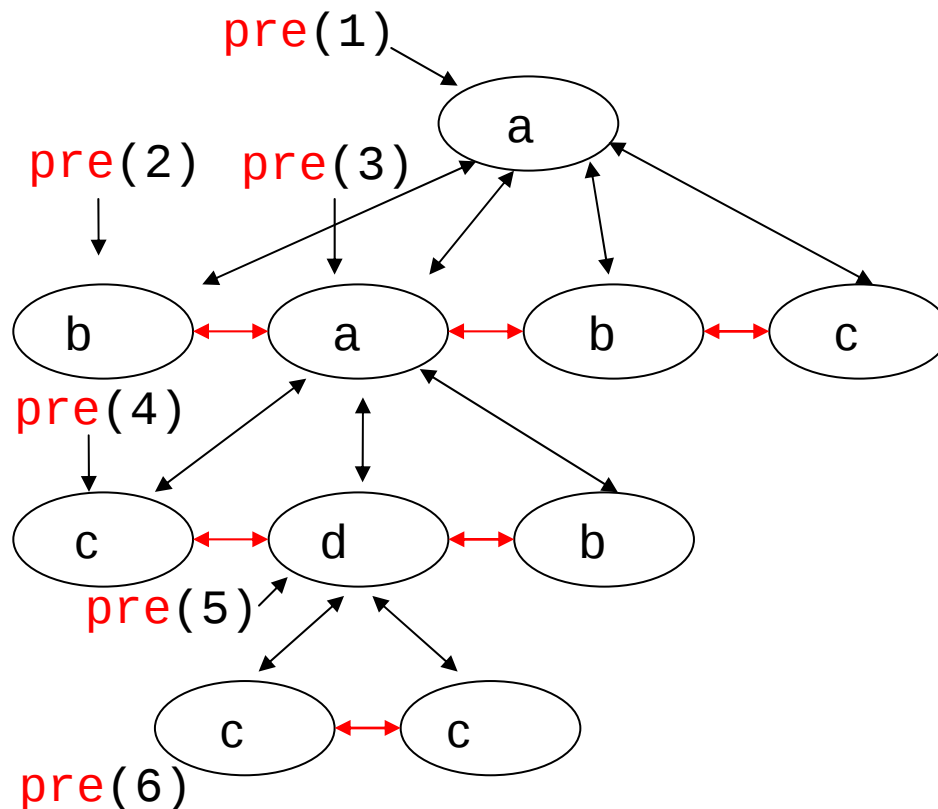
```
i=1;
n=root;
pre(i)=n;
repeat {
    while(firstChild(n)!=NIL)
    {   n=firstChild(n);
        pre(++i)=n;
    }
    while(nextSibling(n)=NIL)
    {   n=parent(n);}
    n=nextSibling(n);
    pre(++i)=n;
}
```

Pre-Order

We saw how to compute
Pre-order (and **Post** and **In**)

(1) *recursively*

memory need: $O(\text{max_height})$



How to compute Pre-order

(2) *iteratively*

→ Memory need?

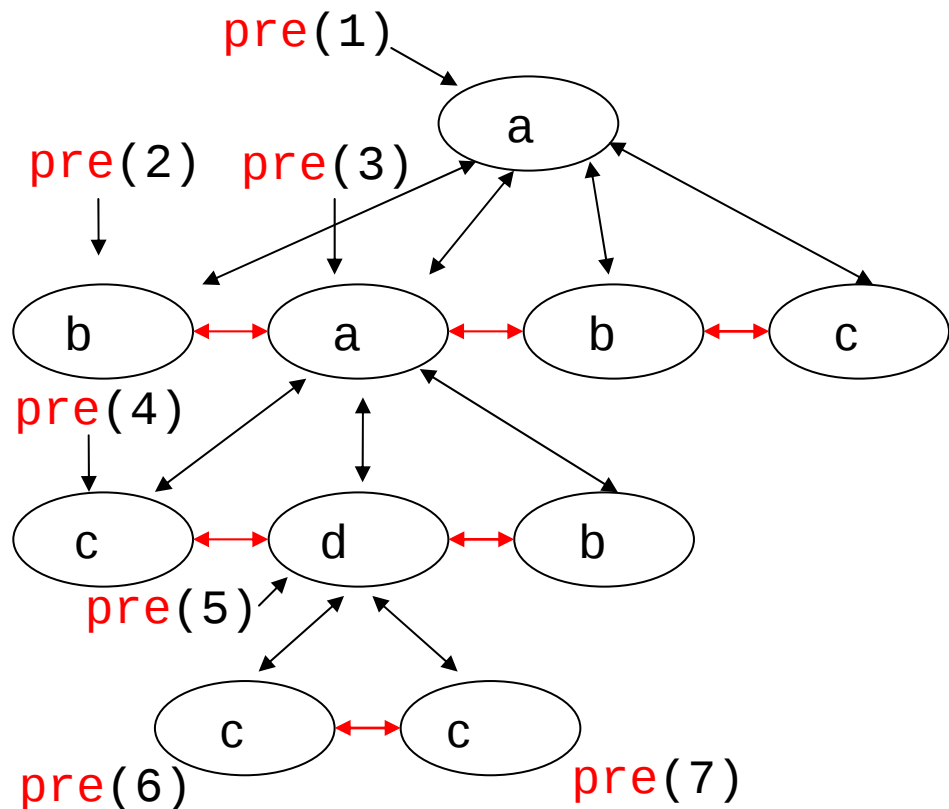
```
i=1;
n=root;
pre(i)=n;
repeat {
    while(firstChild(n)!=NIL)
        { n=firstChild(n);
          pre(++i)=n;
        }
    while(nextSibling(n)=NIL)
        { n=parent(n); }
    n=nextSibling(n);
    pre(++i)=n;
}
```

Pre-Order

We saw how to compute
Pre-order (and **Post** and **In**)

(1) *recursively*

memory need: $O(\text{max_height})$



How to compute Pre-order

(2) *iteratively*

→ Memory need?

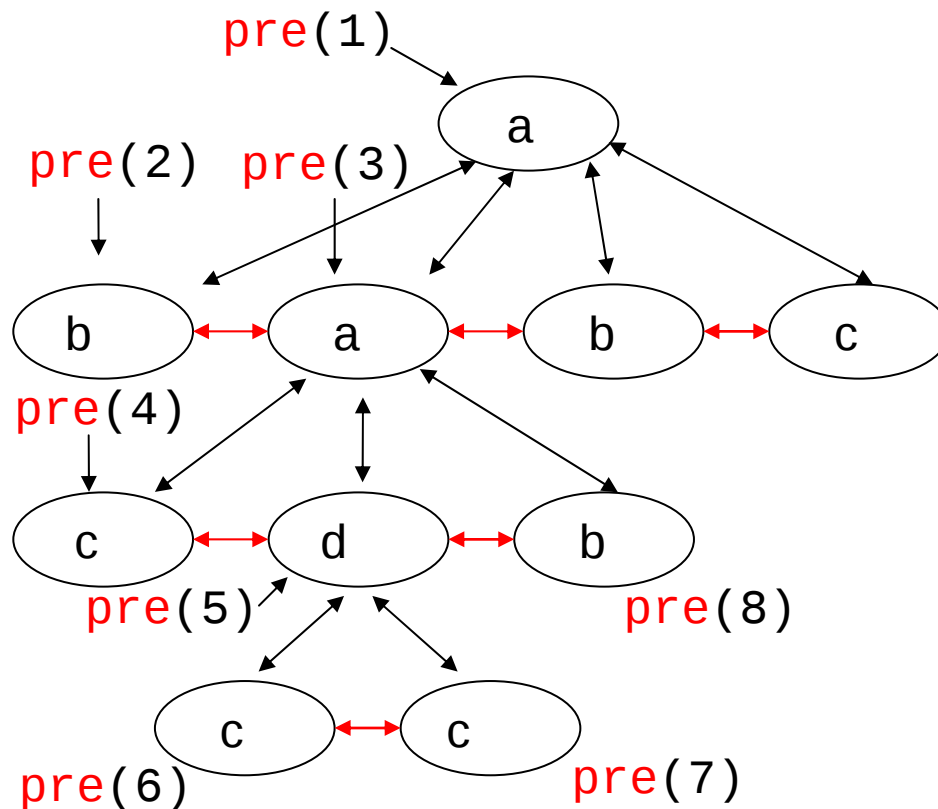
```
i=1;
n=root;
pre(i)=n;
repeat {
    while(firstChild(n)!=NIL)
    {   n=firstChild(n);
        pre(++i)=n;
    }
    while(nextSibling(n)=NIL)
    {   n=parent(n);}
    n=nextSibling(n);
    pre(++i)=n;
}
```

Pre-Order

We saw how to compute
Pre-order (and Post and In)

(1) *recursively*

memory need: $O(\text{max_height})$



How to compute Pre-order

(2) *iteratively*

→ Memory need?

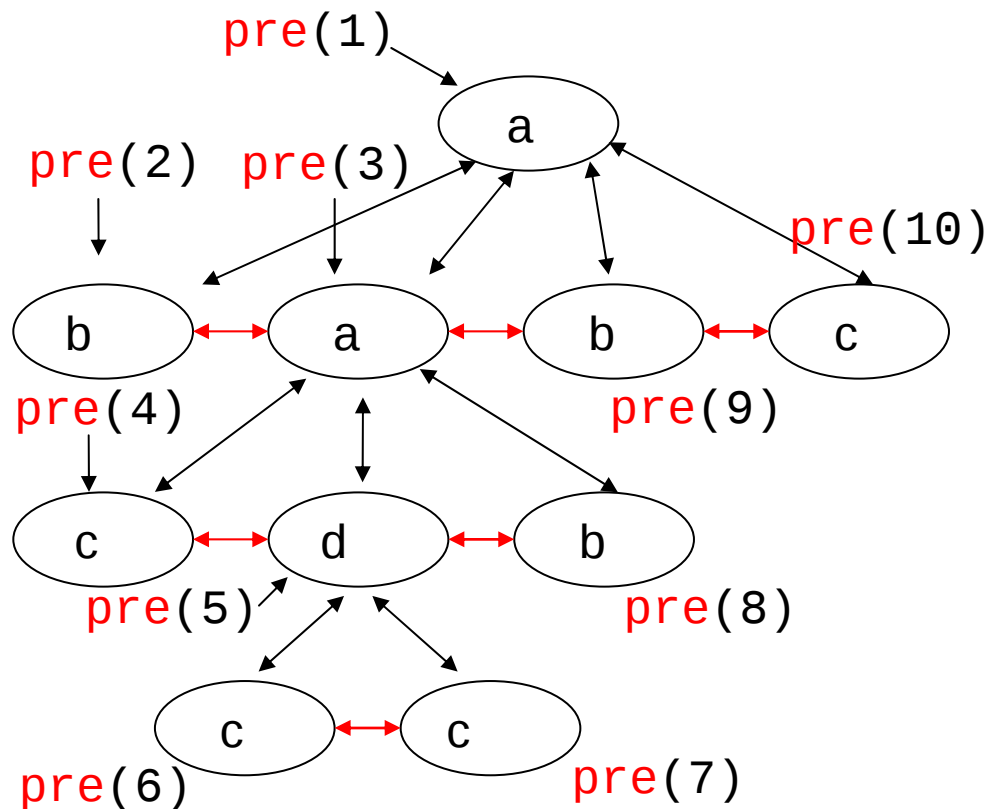
```
i=1;
n=root;
pre(i)=n;
repeat {
    while(firstChild(n)!=NIL)
    {   n=firstChild(n);
        pre(++i)=n;
    }
    while(nextSibling(n)=NIL)
    {   n=parent(n);}
    n=nextSibling(n);
    pre(++i)=n;
}
```

Pre-Order

We saw how to compute
Pre-order (and Post and In)

(1) *recursively*

memory need: $O(\text{max_height})$



How to compute Pre-order

(2) *iteratively*

→ Memory need?

```
i=1;
n=root;
pre(i)=n;
repeat {
  while(firstChild(n)!=NIL)
    { n=firstChild(n);
      pre(++i)=n;
    }
  while(nextSibling(n)=NIL)
    { n=parent(n);
      n=nextSibling(n);
      pre(++i)=n;
    }
}
```

if(n=NIL) then break; ☺

Pre-Order

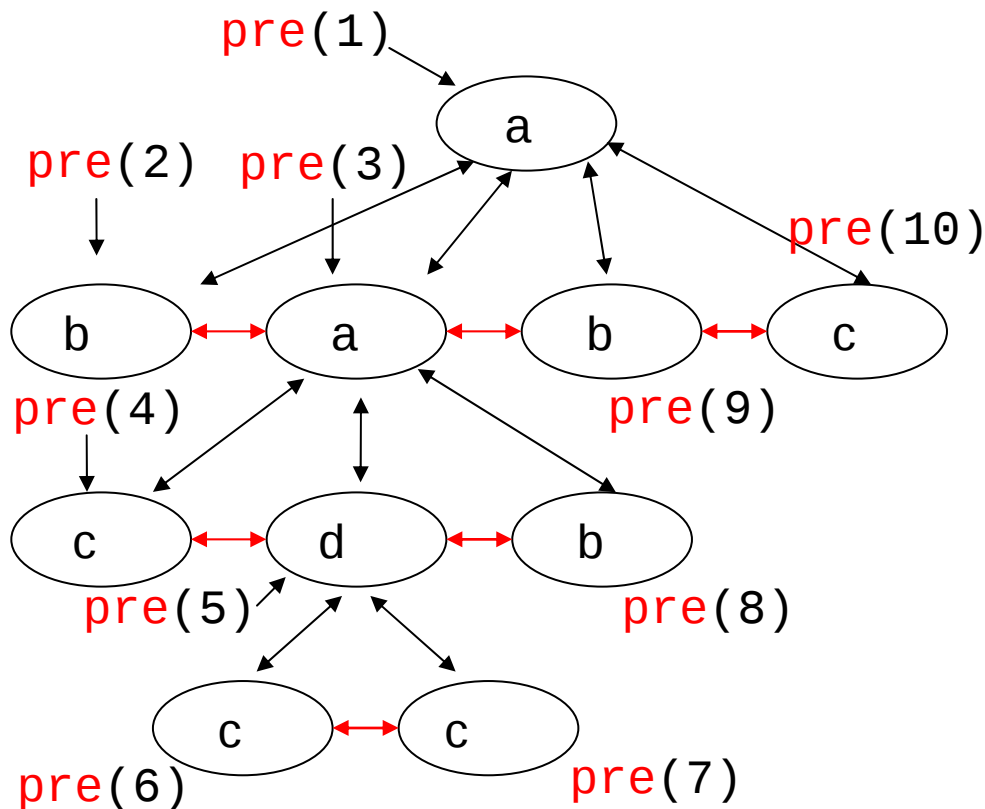
How to compute Pre-order

(2) *iteratively*

→ Memory need?

No recursion!

Needs constant memory!
(only one pointer)



```

i=1;
n=root;
pre(i)=n;
repeat {
    while(firstChild(n)!=NIL)
        { n=firstChild(n);
          pre(++i)=n;
        }
    while(nextSibling(n)=NIL)
        { n=parent(n);
          n=nextSibling(n);
          pre(++i)=n;
        }
}

if(n=NIL)then break; ☺

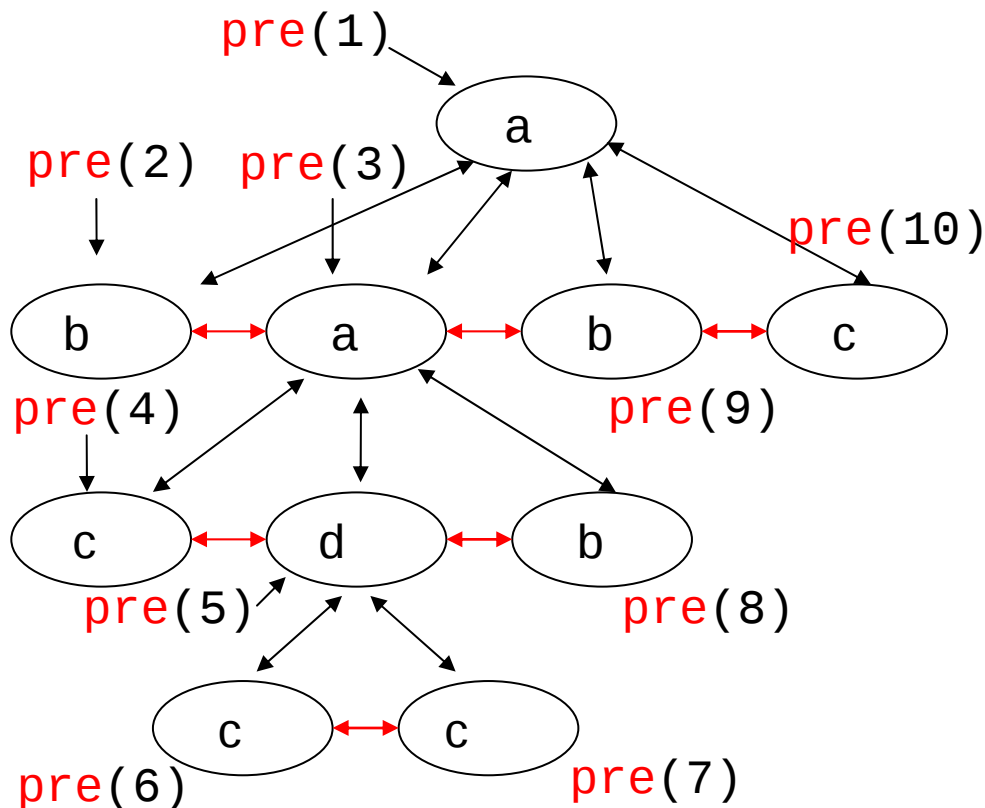
```

Pre-Order

Question

Given a *binary tree*, (top-down, no parent)
how much memory do you need
to compute **pre**?

No recursion!
Needs constant memory!
(only one pointer)



```

i=1;
n=root;
pre(i)=n;
repeat {
    while(firstChild(n)!=NIL)
        { n=firstChild(n);
          pre(++i)=n;
        }
    while(nextSibling(n)=NIL)
        { n=parent(n); }
    n=nextSibling(n);
    pre(++i)=n;
}

if(n=NIL)then break; ☺

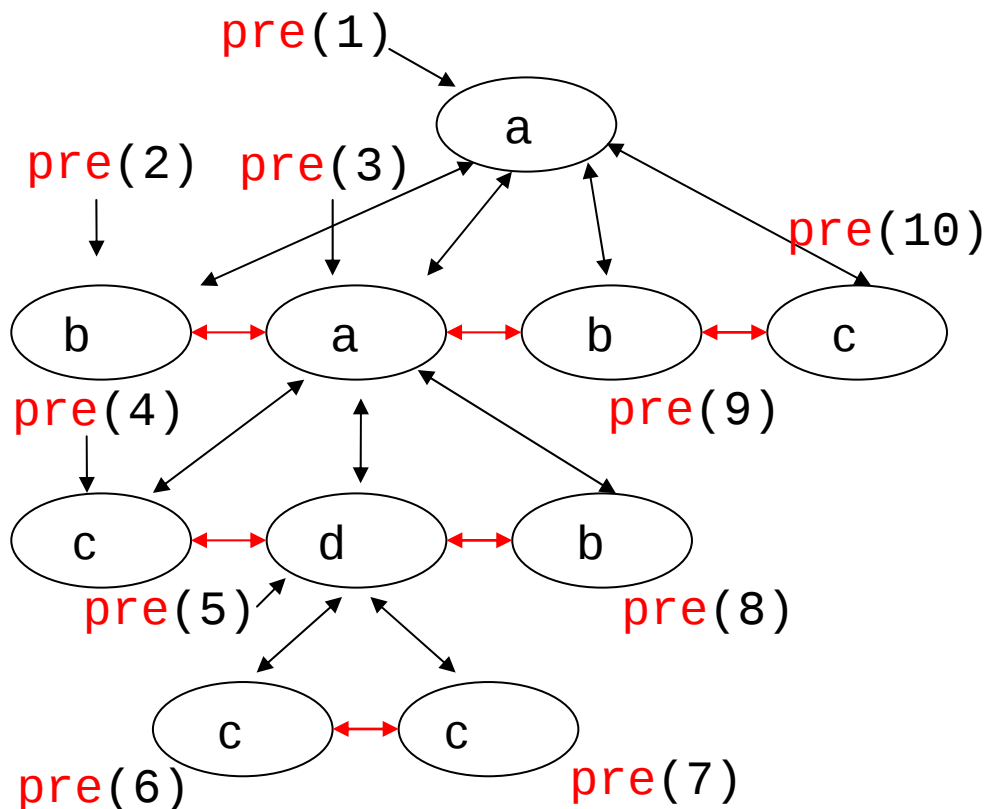
```

Pre-Order

Question

Given a *binary tree*, (top-down, no parent)
how much memory do you need
to compute **pre**?

→ Can you do it w. **constant memory**?



No recursion!
Needs *constant memory*!
(only one pointer)

```
i=1;
n=root;
pre(i)=n;
repeat {
  while(firstChild(n)!=NIL)
    { n=firstChild(n);
      pre(++i)=n;
    }
  while(nextSibling(n)=NIL)
    { n=parent(n); }
  n=nextSibling(n);
  pre(++i)=n;
}

if(n=NIL)then break; ☺
```

Pre-Order

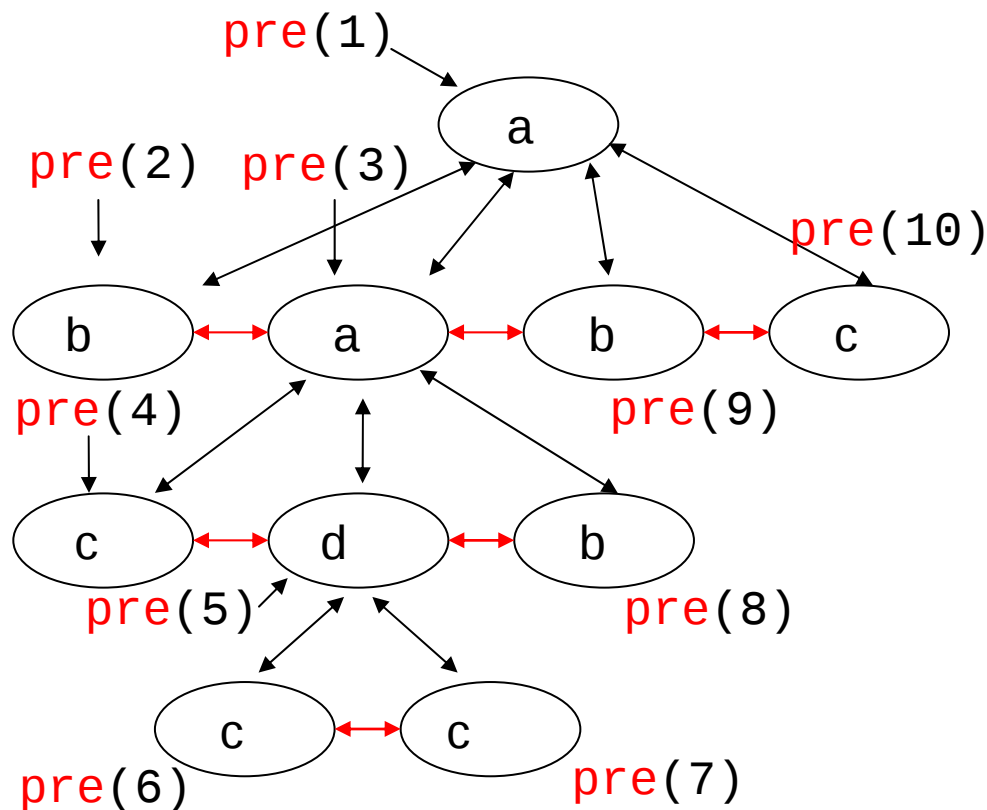
Question

Fun (MS ji)

How much memory you need to check for cycles, in a single-linked (pointer) list?

No recursion!

Needs constant memory!
(only one pointer)



```
i=1;
n=root;
pre(i)=n;
repeat {
    while(firstChild(n)!=NIL)
    {   n=firstChild(n);
        pre(++i)=n;
    }
    while(nextSibling(n)=NIL)
    {   n=parent(n);
        n=nextSibling(n);
        pre(++i)=n;
    }
}
```

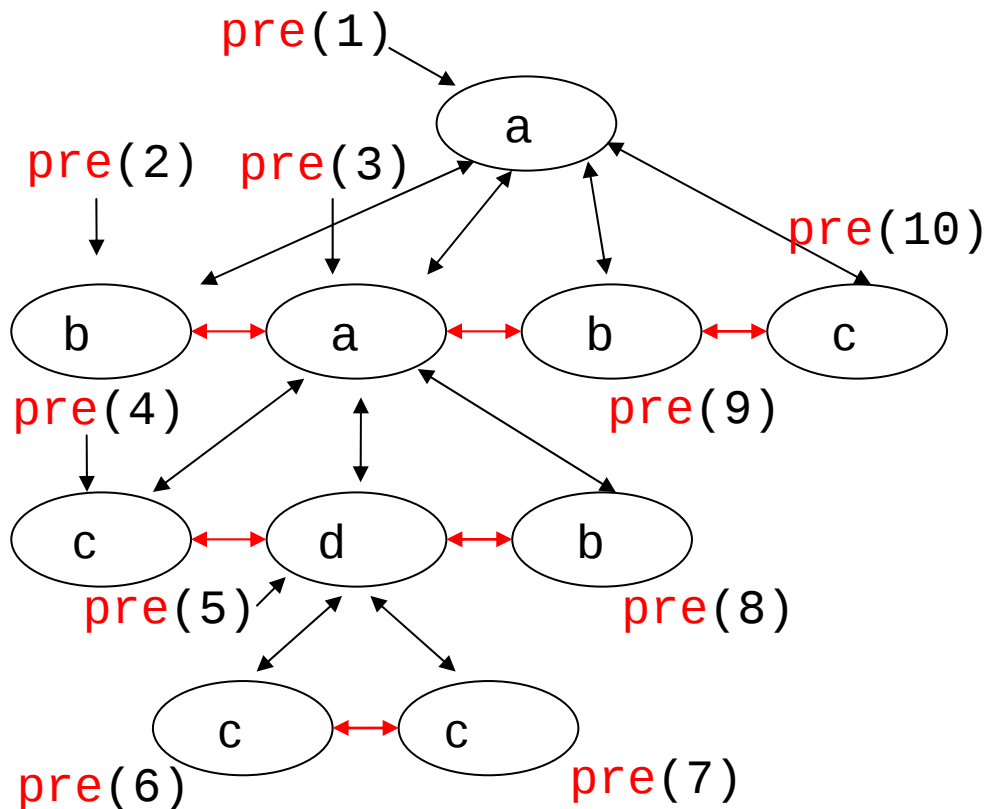
if(n=NIL)then break; ☺

Pre-Order

Question

Do you see how to do **Post-** and **In-orders** iteratively?

No recursion!
Needs *constant memory!*
(only one pointer)



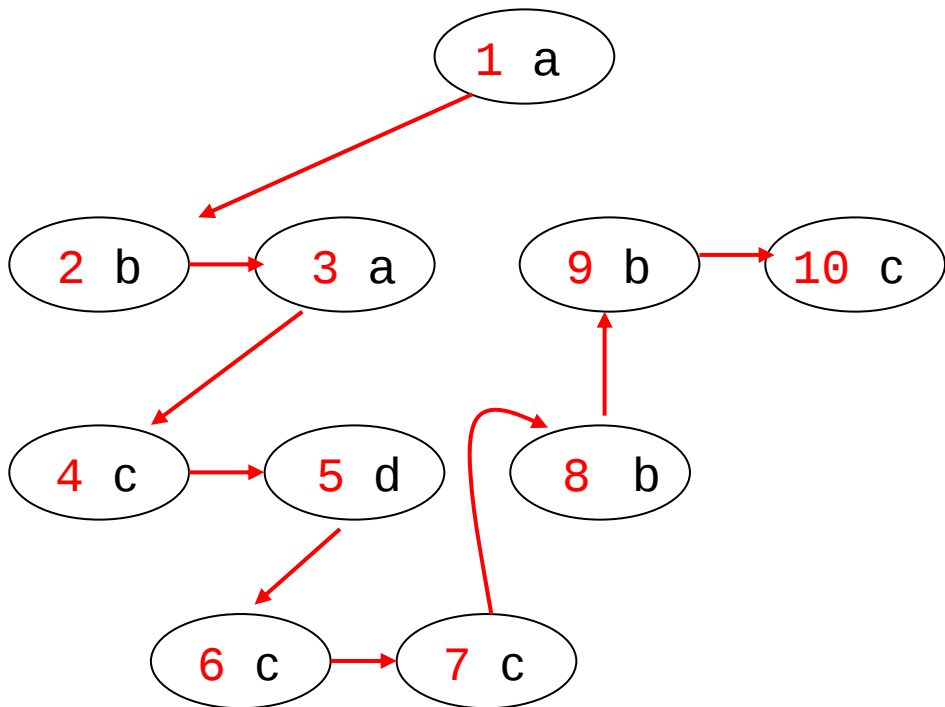
```
i=1;
n=root;
pre(i)=n;
repeat {
    while(firstChild(n)!=NIL)
        { n=firstChild(n);
          pre(++i)=n;
        }
    while(nextSibling(n)=NIL)
        { n=parent(n);
          n=nextSibling(n);
          pre(++i)=n;
        }
}

if(n=NIL)then break; ☺
```

Pre-Order

From `pre( )`

we can compute $\rightarrow \text{PreFollowing}(n) = \{ \text{nodes } m \text{ with } \text{pre}(m) > n \}$
 $\rightarrow \text{PrePreceding}(n) = \{ \text{nodes } m \text{ with } \text{pre}(m) < n \}$



`pre`

 1
 2
 3
 4
 5
 6
 7
 8
 9
 10

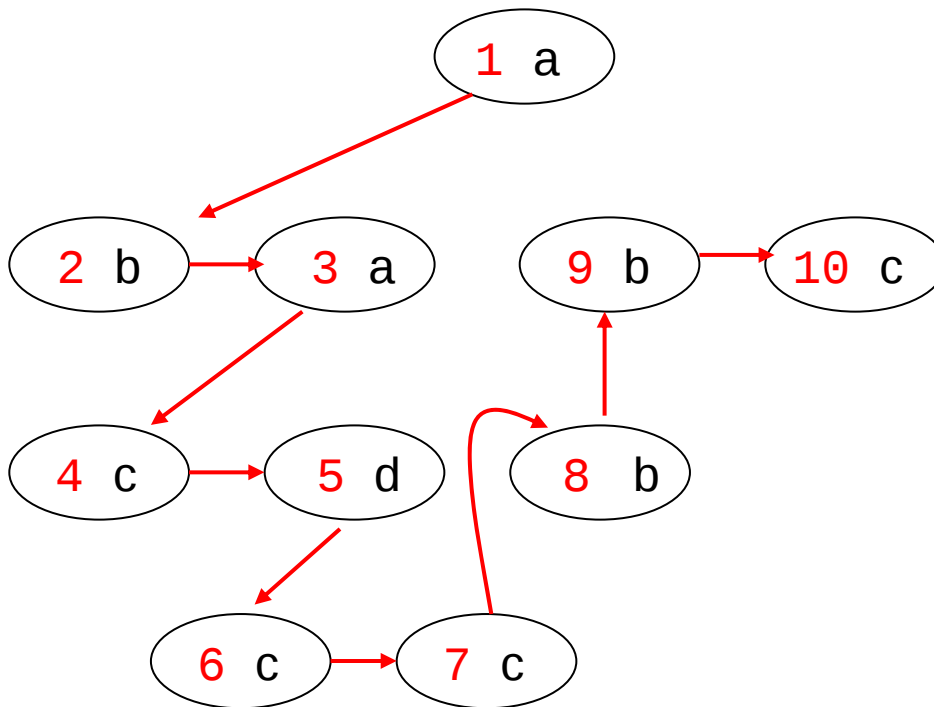
Pre-Order

What is this?

From `pre( )`

we can compute

→ `PreFollowing( n )` = { nodes *m* with `pre(m) > n` }
 → `PrePreceding( n )` = { nodes *m* with `pre(m) < n` }



`pre`

 1
 2
 3
 4
 5
 6
 7
 8
 9
 10

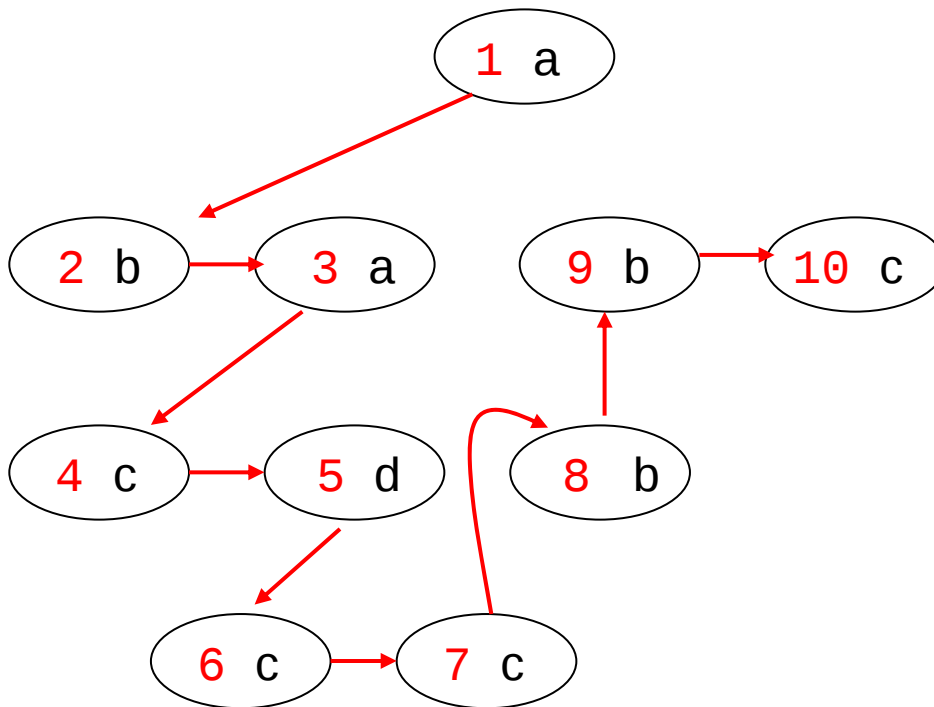
Pre-Order

From `pre( )`

we can compute

→ `PreFollowing( n )` = { nodes `m` with `pre(m) > n` }
 → `PrePreceding( n )` = { nodes `m` with `pre(m) < n` }

What is this?



pre	foll	prec
1	2-10	-
2	3-10	1
3	4-10	1-2
4	5-10	1-3
5	6-10	1-4
6	7-10	1-5
7	8-10	1-6
8	9-10	1-7
9	10	1-8
10	-	1-9

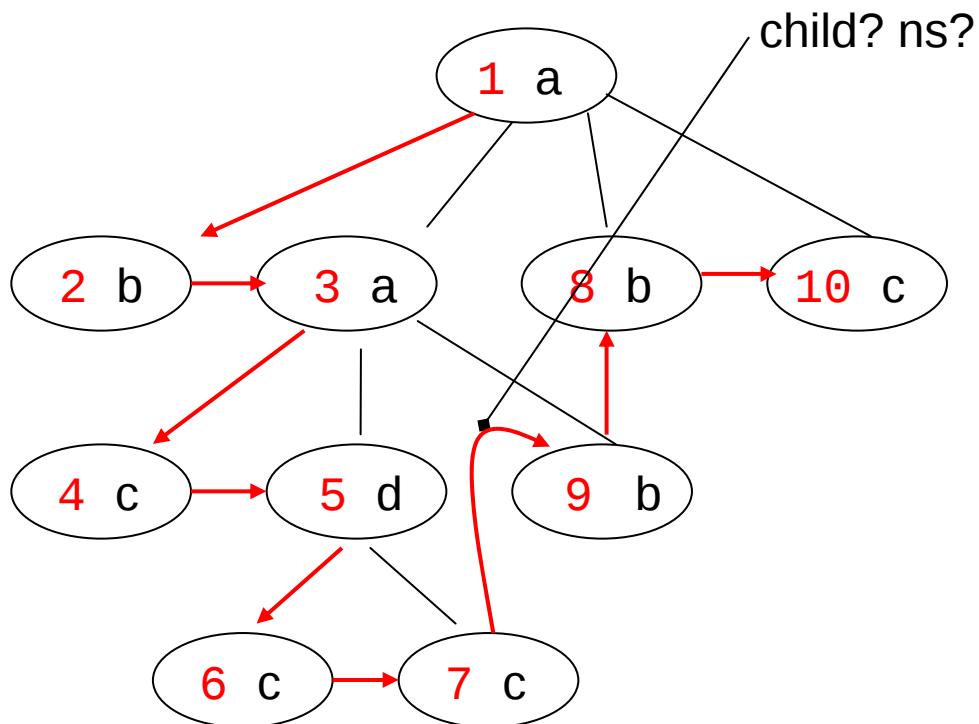
Pre-Order

...useful??

From `pre( )`

we can compute $\rightarrow \text{PreFollowing}(n) = \{ \text{nodes } m \text{ with } \text{pre}(m) > n \}$
 $\rightarrow \text{PrePreceding}(n) = \{ \text{nodes } m \text{ with } \text{pre}(m) < n \}$

Not “tree (navigation) complete” ☹



`pre`

 1
 2
 3
 4
 5
 6
 7
 8
 9
 10

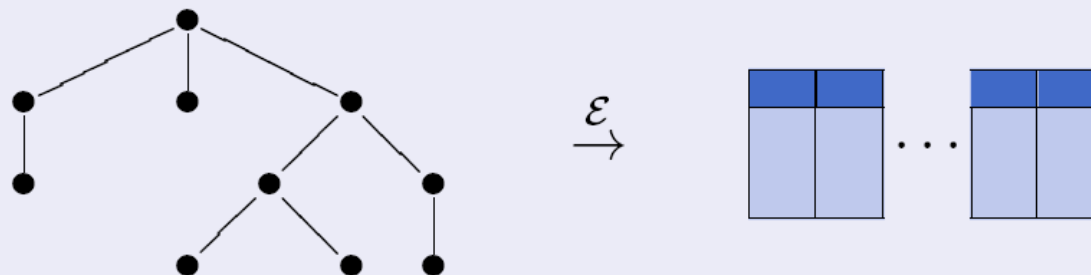
XML to RDBMS Encoding

Relational XML processors (2)

Our approach to **relational XQuery processing**:

- The XQuery data model—ordered, unranked trees and ordered item sequences—is, in a sense, alien to a relational database kernel.
- A **relational tree encoding** $\mathcal{E}$ is required to map trees into the relational domain, *i.e.*, tables.

Relational tree encoding $\mathcal{E}$



What makes a good (relational) (XML) tree encoding?

Hard requirements:

- ① $\mathcal{E}$ is required to reflect **document order** and **node identity**.
 - ▶ *Otherwise*: cannot enforce XPath semantics, cannot support $\ll$ and is , cannot support node construction.
- ② $\mathcal{E}$ is required to encode the **XQuery DM node properties**.
 - ▶ *Otherwise*: cannot support XPath axes, cannot support XPath node tests, cannot support atomization, cannot support validation.
- ③ $\mathcal{E}$ is able to encode any well-formed **schema-less** XML fragment (*i.e.*, $\mathcal{E}$ is “**schema-oblivious**”, see below).
 - ▶ *Otherwise*: cannot process non-validated XML documents, cannot support arbitrary node construction.

What makes a good (relational) (XML) tree encoding?

Soft requirements (primarily motivated by performance concerns):

- ④ **Data-bound operations** on trees (potentially delivering/copying lots of nodes) should map into efficient database operations.
 - ▶ *XPath location steps* (12 axes)
- ⑤ Principal, recurring **operations imposed by the XQuery semantics** should map into efficient database operations.
 - ▶ *Subtree traversal* (atomization, element construction, serialization).

For a relational encoding, “database operations” always mean “table operations” ...

XML to RDBMS Encoding

pre() is not enough

Other possibilities:

- large (unparsed) text bloc
- Schema-based encoding
- Adjacency-based encoding

Dead Ends
Not good...

Good possibility: use **Pre-** and **Post-order** of a node!

Dead end #2: Schema-based encoding

XML address database (excerpt)

```

<person>
  <name><first>John</first><last>Foo</last></name>
  <address><street>13 Main St</street>
    <zip>12345</zip><city>Miami</city>
  </address>
</person>
<person>
  <name><first>Erik</first><last>Bar</last></name>
  <address><street>42 Kings Rd</street>
    <zip>54321</zip><city>New York</city>
  </address>
</person>

```

Schema-based relational encoding: table person

<u>id</u>	first	last	street	zip	city
0	John	Foo	13 Main St	12345	Miami
1	Erik	Bar	42 Kings Rd	54321	New York

Dead end #2: Schema-based encoding

Irregular hierarchy

```

<a no="0">
  <b><c>X</c></b>
</a>
<a no="1">
  <b><c>Y</c></b>
</a>
<a><b/></a>
<a no="3"/>

```

A relational encoding

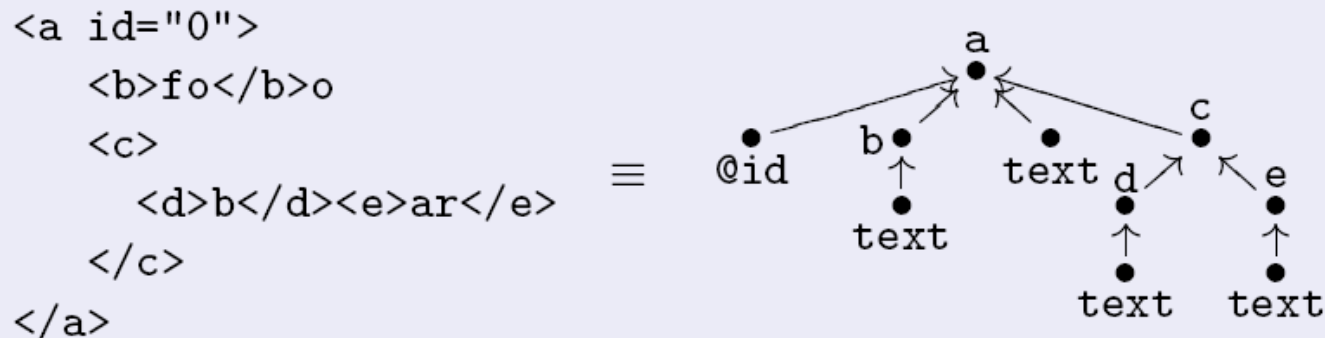
<u>id</u>	@no	b	<u>id</u>	b	c
0	0	α	1	α	X
3	1	β	2	α	NULL ^c
5	NULL ^a	γ	4	β	Y
6	3	NULL ^b			

Issues:

- Number of encoding tables depends on nesting depth.
- Empty element c encoded by NULL^c, empty element b encoded by absence of γ (will need *outer join* on column b).
- NULL^a encodes absence of attribute, NULL^b encodes absence of element.
- Document order/identity of b elements only implicit.

Dead end #3: Adjacency-based encoding

Adjacency-based encoding of XML fragments



Resulting relational encoding

<u>id</u>	parent	tag	text	val
0	NULL	a	NULL	NULL
1	0	@id	NULL	"0"
2	0	b	NULL	NULL
3	2	NULL	"fo"	NULL
4	0	NULL	"o"	NULL
5	0	c	NULL	NULL
⋮				

Dead end #3: Adjacency-based encoding

- **Pro:**

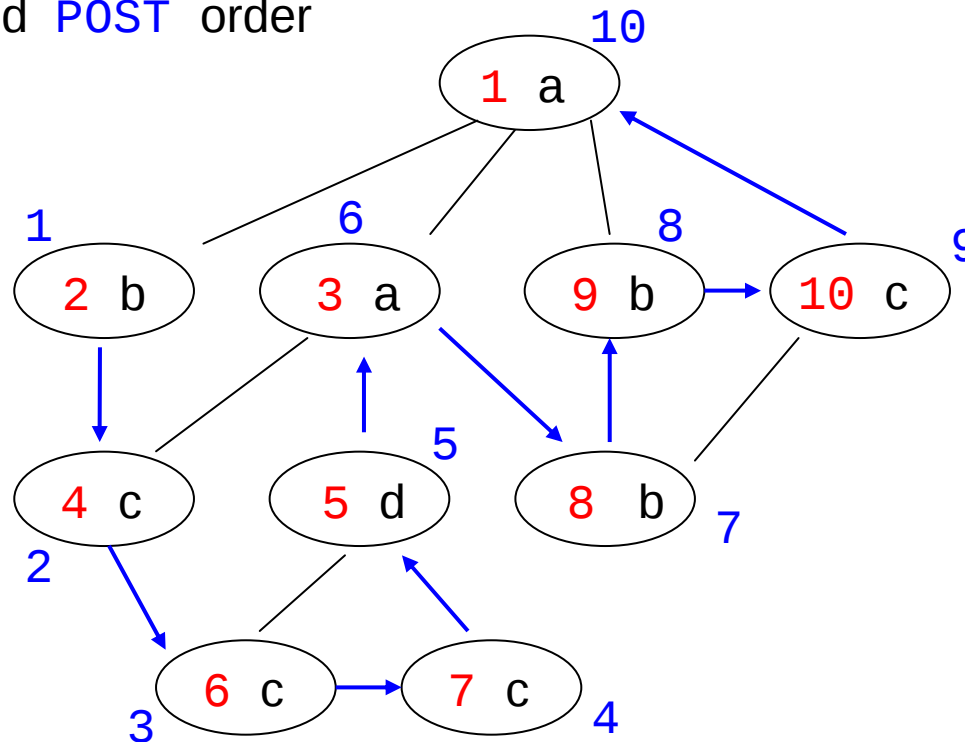
- ▶ Since this captures all adjacency, kind, and content information, we can—in principle—**serialize the original XML fragment**.
- ▶ **Node identity** and **document order** is adequately represented.

- **Contra:**

- ▶ The XQuery processing model is not well-supported: subtree traversals require **extra-relational** queries (**recursion**).
- ▶ This is completely parent–child centric. How to support descendant, ancestor, following, or preceding?

Pre/Post Encoding

➔ Add **POST** order



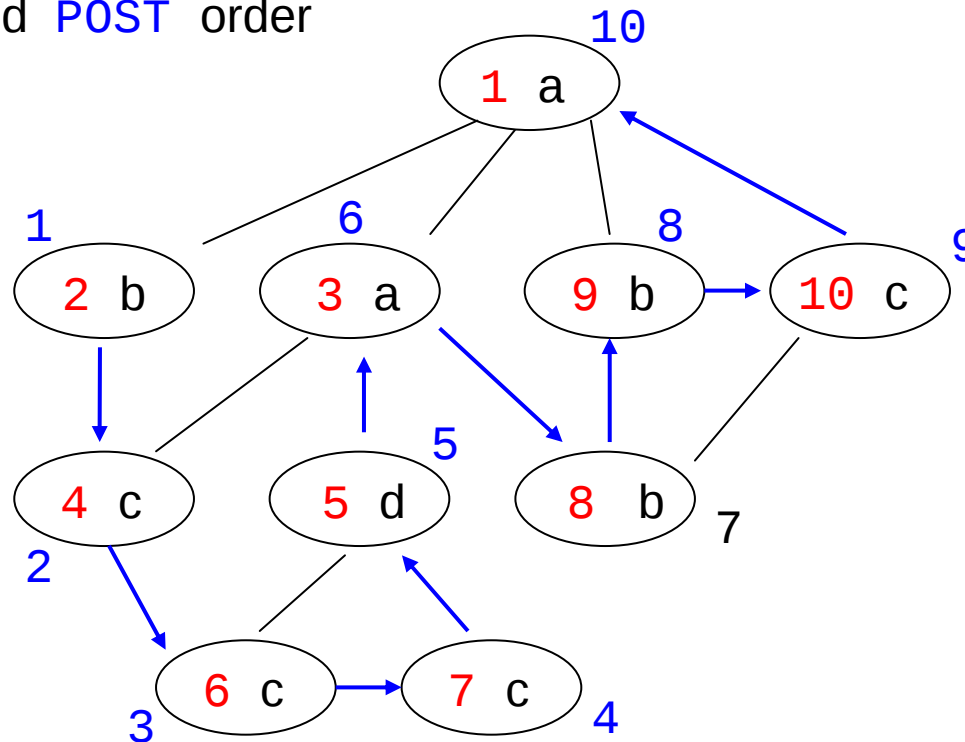
PRE	POST	lab
1	10	a
2	1	b
3	6	a
4	2	c
5	5	d
6	3	c
7	4	c
8	7	b
9	8	b
10	9	c

```

CREATE VIEW descendant AS
SELECT r1.pre,r2.pre FROM R r1, R r2
WHERE r1.pre<r2.pre
      AND r1.post>r2.post
  
```

Pre/Post Encoding

➔ Add **POST** order



PRE	POST	lab
1	10	a
2	1	b
3	6	a
4	2	c
5	5	d
6	3	c
7	4	c
8	7	b
9	8	b
10	9	c

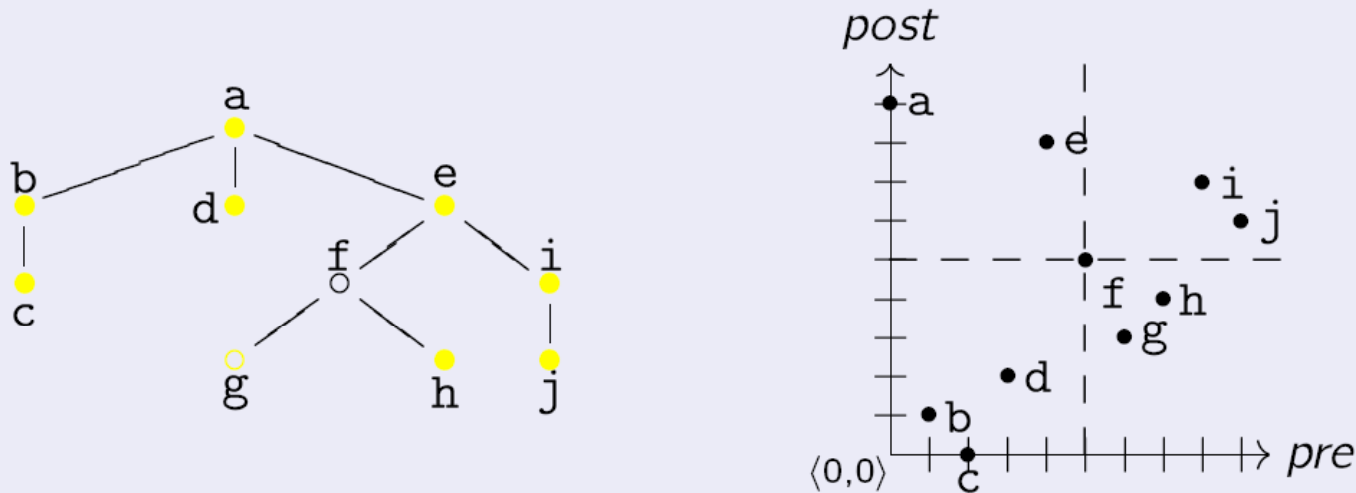
```

CREATE VIEW descendant AS
SELECT r1.pre, r2.pre FROM R r1, R r2
WHERE r1.pre < r2.pre
      AND r1.post > r2.post
  
```

“structural join”

XPath axes in the pre/post plane

Plane partitions $\equiv$ XPath axes, $\circ$ is arbitrary!



Pre/post plane regions $\equiv$ major XPath axes

The **major XPath axes** descendant, ancestor, following, preceding correspond to rectangular **pre/post plane windows**.

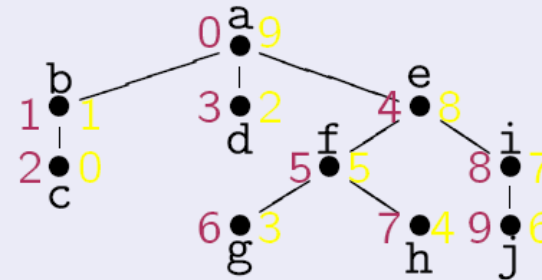
XPath Accelerator encoding

XML fragment *f* and its skeleton tree

```

<a>
  <b>c</b>
  <!--d-->
  <e><f><g/><?h?></f>
    <i>j</i>
  </e>
</a>

```



Pre/post encoding of *f*: table accel

<i>pre</i>	<i>post</i>	<i>par</i>	<i>kind</i>	<i>tag</i>	<i>text</i>
0	9	NULL	elem	a	NULL
1	1	0	elem	b	NULL
2	0	1	text	NULL	c
3	2	0	com	NULL	d
4	8	0	elem	e	NULL
5	5	4	elem	f	NULL
6	3	5	elem	g	NULL
7	4	5	pi	NULL	h
8	7	4	elem	i	NULL
9	6	8	text	NULL	j

Pre/Post Encoding

Straightforward how to compute, for a given node,

- descendants
- ancestors
- following
- preceding

Questions

How to do

- lastChild
- parent
- childNodes

Can you find corresponding SQL queries?

END

Lecture 3