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 w/o 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?

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?
Start at root node; want to visit every node.

```
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

\[ \text{Traverse}(n: \text{Node}) \{ \
\text{print}(n); \\
\text{for } m \text{ in childNodes}(n) \text{ Traverse}(m) \\
\} \]

Memory need proportional to the height of the XML tree.

⇒ Should be fine. Usually \( \text{height} \) (XML doc) is small. \( \leq 15 \)

Problematic
2nd recursion on children

\[ \text{Problematic} \]

\[ \text{2nd recursion on children} \]

\[ \text{Problematic} \]

\[ \text{2nd recursion on children} \]
Tree Traversals

Start at root node; want to visit every node.

(1) Recursively

\[
\text{Traverse}(n: \text{Node}) = \{ \\
\quad \text{print}(n) ; \\
\quad \text{For } m \in \text{childNodes}(n) \text{ Traverse}(m) \\
\} 
\]

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

Memory need proportional to the height of the XML tree.
Æ Should be fine. Usually height (XML doc) is small. (≤ 15)

Problematic

2nd recursion on children

1 2 3 4 5 6
A B C D E F

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?

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) 3 b)
2 [3:3 b)
3 [4:9 a)
4 [5:5 c)
5 [6:8 d)
6 [7:-:c)
7 [-:-c)
8 [-:-b)
9 [-:10 b)
10 [1:-:-c)

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

Binary Tree Encoding

Recall "firstChild/nextSibling" encoding.

Per Node
2 pointers/IDs + label info

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

Both Traverse and Tr can be executed on the fcns-binary tree encoding.

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

- Traverse left subtree in post-order
- Traverse right subtree in post-order
- Visit the root

Reverse Polish Notation

Pre-Order

Recursive

- Visit the root
- Traverse left subtree in pre-order
- Traverse right subtree in pre-order

Memory need proportional to max. nodes on one level

Breadth-First (left-to-right)

- Level-order

Other Traversals

We discussed the Pre-order of the tree.

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

Binary Search Tree (increasing)

Other Traversals

Post-order of a tree =

- Traverse left subtree in post-order
- Traverse right subtree in post-order
- Visit the root

Reverse Polish Notation

Breadth-First (left-to-right)

- Level-order

Memory need proportional to max. nodes on one level

Pre-Order

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

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

Memory need proportional to max. nodes on one level

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

True for both, unranked & binary tree
How to compute Pre-order iteratively

We saw how to compute Pre-order (and Post and In) recursively
memory need: \(O(\text{max\_height})\)

```c
i = 1;
n = root;
pre[1] = n;
while (firstChild(n) != NULL)
    { n = firstChild(n);
      pre[++i] = n;
    }
```

```c
i = 1;
n = root;
pre[0] = n;
while (firstChild(n) != NULL)
    { n = firstChild(n);
      pre[++i] = n;
    }
```

We saw how to compute Pre-order (and Post and In) iteratively
memory need: \(O(\text{max\_height})\)

```c
i = 1;
n = root;
pre[1] = n;
repeat {
    while (firstChild(n) != NULL)
        { n = firstChild(n);
          pre[++i] = n;
        }
    while (nextSibling(n) != NULL)
        { n = parent(n);
          n = nextSibling(n);
          pre[++i] = n;
        }
}
```

```c
i = 1;
n = root;
pre[1] = n;
repeat {
    while (firstChild(n) != NULL)
        { n = firstChild(n);
          pre[++i] = n;
        }
    while (nextSibling(n) != NULL)
        { n = parent(n);
          n = nextSibling(n);
          pre[++i] = n;
        }
}
```
We saw how to compute Pre-order (and Post and In)

(1) recursively
memory need: $O(\text{max height})$

(2) iteratively
Memory need?

\begin{verbatim}
  i = 1;
  n = root;
  pre(1) = 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;
    }
  }
\end{verbatim}
Question

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

```c
i = 1; 
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;
```

Can you do it with constant memory?

Question

Pre-Order

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

Pre-Order

Fun (MS ji)

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

```c
i = 1; 
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;
```

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

Question

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

Pre-Order

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

$PrePreceding(n) = \{ \text{nodes m with } pre(m) < n \}$

Pre-Order

What is this?

```c
pre 
\[ \begin{array}{cccccccccc} 
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array} \]
```
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 $\xi$ is required to map trees into the relational domain, i.e., tables.

Relational tree encoding $\xi$

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

Hard requirements:
1. $\xi$ is required to reflect document order and node identity.
2. $\xi$ is required to encode the XQuery DM node properties.
3. $\xi$ is able to encode any well-formed schema-less XML fragment (i.e., $\xi$ is "schema-oblivious", see below).

Soft requirements (primarily motivated by performance concerns):
1. Data-bound operations on trees (potentially delivering/copying lots of nodes) should map into efficient database operations.
2. Principal, recurring operations imposed by the XQuery semantics should map into efficient database operations.

For a relational encoding, “database operations” always mean “table operations”...
XML to RDBMS Encoding

The \texttt{pre()} function is not enough for encoding.

Other possibilities:
- Large (unparsed) text block
- Schema-based encoding
- Adjacency-based encoding

<table>
<thead>
<tr>
<th>Dead Ends</th>
<th>Not good...</th>
</tr>
</thead>
</table>

Good possibility: use \texttt{Pre- and Post-order} of a node!

---

### Dead end #2: Schema-based encoding

#### Irregular hierarchy:

```
<book><title><title>My Title</title></title>
<author><name><name>John</name></name><name><name>Smith</name></name></author>
<address><street><street>13 Main St</street><street>12340</street></street><city>City</city></address>
```

#### A relational encoding:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>30</td>
<td>City</td>
</tr>
<tr>
<td>2</td>
<td>Smith</td>
<td>25</td>
<td>City</td>
</tr>
</tbody>
</table>

### Dead end #3: Adjacency-based encoding

#### Adjacency-based encoding of XML fragments:

```
<book>
  <title>My Title</title>
  <author><name>John</name><name>Smith</name></author>
  <address><street>13 Main St</street><city>City</city></address>
</book>
```

#### Resulting relational encoding:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
<td>30</td>
<td>City</td>
</tr>
<tr>
<td>2</td>
<td>Smith</td>
<td>25</td>
<td>City</td>
</tr>
</tbody>
</table>

#### Pre/Post Encoding

```
CREATE VIEW descendant AS
SELECT r1.pre, r2.pre FROM R r1, R r2
WHERE r1.pre < r2.pre
AND r1.post = r2.pre;
```
Pre/Post Encoding

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 Accelerator encoding

XML fragment f and its skeleton tree

Pre/post encoding of f: table accel.

Questions

Can you find corresponding SQL queries?

END Lecture 3