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?

... but first: Memory efficient tree traversals using e.g. DOM?
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
\[
\text{Traverse( } n \text{ Node) } \{
\text{ print(n); For m in childNodes(n) Traverse(m) }
\}
\]

\[
\begin{array}{c}
\text{Memory need proportional to the height of the XML doc tree.}
\end{array}
\]

\[
\begin{array}{c}
\text{Should be fine. Usually height (XML doc) is small. } \leq 15
\end{array}
\]

Problematic
2nd recursion on children

\[
\begin{array}{c}
\text{TR(n:Node) } \{
\text{ print(n); if (m=firstChild(n))!=NIL then TR(m); if (m=nextSibling(n))!=NIL then TR(m) }
\}
\]

\[
\begin{array}{c}
\text{Should be fine. Usually height (XML doc) is small. } \leq 15
\end{array}
\]
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 doc tree.

Should be fine. Usually height (XML doc) is small. \((\leq 15)\)

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

Binary Tree Encoding
Recall “firstChild/nextSibling” encoding.
The “firstChild” becomes the left pointer
The “nextSibling” becomes the right pointer

```
<table>
<thead>
<tr>
<th>ID</th>
<th>fc:ns:lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2: a</td>
</tr>
<tr>
<td>2</td>
<td>3: b</td>
</tr>
<tr>
<td>3</td>
<td>4: 9: a</td>
</tr>
<tr>
<td>4</td>
<td>5: 5: c</td>
</tr>
<tr>
<td>5</td>
<td>6: 6: d</td>
</tr>
<tr>
<td>6</td>
<td>7: c</td>
</tr>
<tr>
<td>7</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Question
What is the max recursion depth on this tree?

Recall binary tree (firstChild/nextSibling) encoding.
The “firstChild” becomes the left pointer
The “nextSibling” becomes the right pointer

```
<table>
<thead>
<tr>
<th>ID</th>
<th>fc:ns:lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2: a</td>
</tr>
<tr>
<td>2</td>
<td>3: b</td>
</tr>
<tr>
<td>3</td>
<td>4: 9: a</td>
</tr>
<tr>
<td>4</td>
<td>5: 5: c</td>
</tr>
<tr>
<td>5</td>
<td>6: 6: d</td>
</tr>
<tr>
<td>6</td>
<td>7: c</td>
</tr>
<tr>
<td>7</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Question
In the binary tree, what corresponds to this number?

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
Tree Traversals

- Both `Traverse` and `TR` can be executed on the `fcnns-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.

Other Traversals

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

Memory need proportional to max. nodes on one level

- Can be HUGE!!!

Other Traversals

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

- Realized by `Traverse & TR`

Other Traversals

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

- Realized by `Traverse & TR`

Other Traversals

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

- Realized by `Traverse & TR`

Pre-Order

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

1) recursively

```
if(!) 
memory need: O(max_height)
```

```
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: $O(\text{max\_height})$

How to compute Pre-order:
(2) iteratively
$\rightarrow$ Memory need?

$i=1$;
$n=\text{root}$;
$\text{pre}(i)=n$;
while (firstChild($n$) $\neq \text{NIL}$)
{  $n=$ firstChild($n$);
    $\text{pre}(++i)=n$;
}

How to compute Pre-order:
(2) iteratively
$\rightarrow$ Memory need?

$i=1$;
n $\neq \text{root}$;
$\text{pre}(i)=n$;
while (firstChild($n$) $\neq \text{NIL}$)
{  $n=$ firstChild($n$);
    $\text{pre}(++i)=n$;
}

How to compute Pre-order:
(2) iteratively
$\rightarrow$ Memory need?

$i=1$;
n $\neq \text{root}$;
$\text{pre}(i)=n$;
while (firstChild($n$) $\neq \text{NIL}$)
{  $n=$ firstChild($n$);
    $\text{pre}(++i)=n$;
}

pre(1)
pre(2)
pre(3)
pre(4)
We saw how to compute Pre-order (and Post and In)

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

\[
\begin{align*}
i & = 1; \\
n & = \text{root}; \\
\text{pre}(i) & = n; \\
\text{repeat} \{ \\
\quad \text{while} (\text{firstChild}(n) \neq \text{NIL}) \\
\quad \quad \{ \\
\quad \quad \quad n = \text{firstChild}(n); \\
\quad \quad \quad \text{pre}(++i) = n; \\
\quad \quad \}\ \\
\quad \text{while} (\text{nextSibling}(n) \neq \text{NIL}) \\
\quad \quad \{ \\
\quad \quad \quad n = \text{parent}(n); \\
\quad \quad \quad \text{pre}(++i) = n; \\
\quad \quad \}\ \\
\quad \text{n = nextSibling(n); } \\
\quad \text{pre}(++i) = n; \\
\}\ \text{if} (n = \text{NIL}) \text{ then break; }
\end{align*}
\]

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

Can you do it with constant memory?

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).

Do you see how to do Post- and In-order iteratively? No recursion. Needs constant memory (only one pointer).

From pre() we can compute 

- \( \text{PreFollowing}(n) = \{ \text{nodes } m \text{ with } \text{pre}(m) > n \} \)
- \( \text{PrePreceding}(n) = \{ \text{nodes } m \text{ with } \text{pre}(m) < n \} \)
**Pre-Order**

From $\text{pre}(\ )$ we can compute

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

**XML to RDBMS Encoding**

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.

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

**Hard requirements:**

- $\xi$ is required to reflect document order and node identity.
  - Otherwise: cannot enforce XPath semantics, cannot support $\ll$ and $\gg$, cannot support node construction.
- $\xi$ 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.
- $\xi$ is able to encode any well-formed schema-less XML fragment (i.e., $\xi$ is "schema-oblivious", see below).
  - Otherwise: cannot process non-validated XML documents, cannot support arbitrary node construction.

**Soft requirements** (primarily motivated by performance concerns):

- Data-bound operations on trees (potentially delivering/copying lots of nodes) should map into efficient database operations.
  - XPath location steps (12 axes)
- 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:
1. Large (unparsed) text block
2. Schema-based encoding
3. Adjacency-based encoding

**Dead Ends**

No good...

**Questions** Why is (1) a dead end?

---

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

XML database database (excerpt)

```
<address>
  <street>13 Main St</street>
  <zip>12340</zip>
  <city>Miami</city>
</address>
```

**Issues:**
1. Number of encoding tables depends on nesting depth.
2. Empty element c encoded by NULL, empty element b encoded by absence of ϑ (will need outer join on column b).
3. NULL encodes absence of attribute, NULL encodes absence of element.
4. Document order/identity of b elements only implicit.

**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**

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

---

**Schema-based relational encoding: table person**

<table>
<thead>
<tr>
<th>id</th>
<th>first</th>
<th>last</th>
<th>street</th>
<th>zip</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>John</td>
<td>Foe</td>
<td>13 Main St</td>
<td>12340</td>
<td>Miami</td>
</tr>
<tr>
<td>1</td>
<td>Mike</td>
<td>Bart</td>
<td>42 Kings Rd</td>
<td>54321</td>
<td>New York</td>
</tr>
</tbody>
</table>

**Adjacency-based encoding of XML fragments**

<table>
<thead>
<tr>
<th>id</th>
<th>parent</th>
<th>tag</th>
<th>text</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>NULL</td>
<td>0</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>b</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>NULL</td>
<td>&quot;fo&quot;</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>a</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>c</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**Resulting relational encoding**
**Pre/Post Encoding**

- Add **POST** order

- **CREATE VIEW descendant AS**
  ```sql```
  ```
  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.

<table>
<thead>
<tr>
<th>Post</th>
<th>Pre</th>
<th>Value</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>9</td>
<td>NULL</td>
<td>e</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>elem</td>
<td>a</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>text</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>NULL</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>NULL</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>NULL</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>NULL</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>NULL</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>NULL</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>text</td>
<td>i</td>
</tr>
</tbody>
</table>

**Questions**

- How to do
  - lastChild
  - parent
  - childNodes

Can you find corresponding SQL queries?

---

**END Lecture 3**