Lecture 2

XML into Memory

Problem with DOM

- Uses massive amounts of memory.
- Even if application touches only a single element node, the DOM API has to maintain a data structure that represents the whole XML input document.

Example

<table>
<thead>
<tr>
<th>XML size</th>
<th>DOM process size</th>
</tr>
</thead>
<tbody>
<tr>
<td>81M</td>
<td>× 2</td>
</tr>
<tr>
<td>52M</td>
<td>× 13</td>
</tr>
</tbody>
</table>

- Usually: more than 10-times blow up!!

To remedy the memory hunger of DOM …

Preprocess (i.e., filter) the input XML document to reduce its overall size.

- Use an XPath/XSLT processor to preselect interesting document regions.
- CAVE: no updates on the input XML document are possible.
- CAVE: make sure the XPath/XSLT processor is not implemented on top of DOM!

- Use a completely different approach to XML processing (⇒ SAX).
  “design your own XML data structure and fill in with what you need…”
1. Tree pointer structures

Type Node {
  label : String,
  left : Node,
  right : Node
}

1. Consider binary trees

How much memory for n-node binary tree?

length(label_1) + 1
+ length(label_2) + 1
+ … + length(label_n) + 1
3 * length(pointer) * n

Typical: 4 bytes

Æ

Whatever is needed for the labels
PLUS 12 bytes per node.

Can easily be optimized.
E.g., store each distinct string only once!
1. Consider binary trees

**Type Node {**
  label : String,
  left  : Node,
  right : Node
**}**

Serialization to XML

```xml
<library><book><....</book></library>
```

- #characters per node: 5 + 2 * Length(label)
- E.g., one node w. 4-character ASCII label: 13 bytes (assuming UTF-8!)

Often #distinct node labels is small, *100. → Fits in one Byte
Then, only 9 bytes per node.

- MEM(n-node binary tree pointer struct, *256 labels) = SIZE(n-node binary tree in XML, average label length=2)

- #characters per node: 5 + 2 * Length(label)
- One node w. 2-character ASCII label: 9 bytes (assuming UTF-8!)

Nice

Following pointers is fast!

- much higher access speed (than on doc seen as string.)

E.g.

at root, get right-child.

Often #distinct node labels is small, *100. → Fits in one Byte
Then, only 9 bytes per node.

- MEM(n-node binary tree pointer struct, *256 labels) = SIZE(n-node binary tree in XML, average label length=2)

- #characters per node: 5 + 2 * Length(label)
- One node w. 2-character ASCII label: 9 bytes (assuming UTF-8!)

1. Consider binary trees

Plain no attributes, no text nodes, ...

**Question**

Using a (top-down) pointer structure, as the one above, how can you implement a DOM interface?

Access speed of parentNode should be approx same, as in a native DOM.

What about access speed of nextSibling?

**Question**

Using a (top-down) pointer structure, as the one above, how can you implement a DOM interface?

At run-time a node is represented as a pointer, PLUS a stack of
pointers of all its ancestors.

At run-time a node is represented as a pointer, PLUS a stack of
pointers of all its ancestors.
Tree pointer structures

To slash memory hunger (of, e.g., DOM…)

LESSON 1

→ Avoid all backward pointers (build them online, dynamically)

1. Consider binary trees

   Type Node {
   label : String,
   left  : Node,
   right : Node
   }

   How much memory for n-node binary tree?

   How to add attributes and text nodes?

   → e.g., “into the label” ...

2. Consider unranked trees

   Type Node {
   label : String,
   children : List[Node]
   }

   How much memory for List[Node] of n nodes?

Tree pointer structures

To slash memory hunger (of, e.g., DOM…)

LESSON 1

→ Avoid all backward pointers (build them online, dynamically)

1. Consider binary trees

   Type Node {
   label : String,
   left  : Node,
   right : Node
   }

   How much memory for n-node binary tree?

   How to add attributes and text nodes?

   → e.g., “into the label” ...

2. Consider unranked trees

   Type Node {
   label : String,
   children : List[Node]
   }

   How much memory for List[Node] of n nodes?
Tree pointer structures

2. Consider unranked trees

unranked = no a priori bound on #children of a node.

Tree structure of XML: unranked trees!

Typically

Type Node {
    label : String,
    children : List[Node]
}

→ How much memory for List[Node] of n nodes?

2n pointers

More efficient possibilities:
(1) Use arrays. Store #children (e.g., in label).
(2) Encode tree as binary tree.

Typically

n pointers + (log d) Bits

2. Binary Tree Encodings

Any unranked tree can be encoded as a binary tree.

Popular encoding: “firstChild/nextSibling” encoding.

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

2. Binary Tree Encodings

Any unranked tree can be encoded as a binary tree.

Popular encoding: “firstChild/nextSibling” encoding.

The “firstChild” becomes the left pointer
The “nextSibling” becomes the right pointer
2. Binary Tree Encodings

Any unranked tree can be encoded as a binary tree.

Popular encoding: "firstChild/nextSibling" encoding.

The "firstChild" becomes the left pointer
The "nextSibling" becomes the right pointer

Questions

- Time overhead for simulating lastChild access, on the binary encoding?
- Can you think of other binary tree encodings?
- How to simulate preceding-sibling?
Binary Tree Encodings

Any unranked tree can be encoded as a binary tree.

Good Property of the firstChild/nextSibling encoding:

- XML types (e.g., DTD, XML Schema, Relax NG) are preserved when going from unranked to binary (and vice versa).

Tree Pointer Structures

Question

Give a data type for binary trees which stores only non-NIL pointers.

Then, n-node tree: mn pointers

Type Node {
  label : String,
  left : Node,
  right : Node
}

3. Minimal Unique DAGs

L1: no backward pointers
L2: use binary trees or efficient arrays

Can we do with even less pointers?

n-node tree: 2n pointers

L1: no backward pointers
L2: use binary trees or efficient arrays

Can we do with even less pointers?

n-node tree: 2 pointers per node

+ Fast child access
- Expensive to update (insert/delete)

Directed Acyclic Graph DAG

share identical subtrees
3. Minimal Unique DAGs

A DAG representation of a tree has always
→ Less than or equal #nodes than the tree
→ Less than or equal #pointers than the tree.

A (minimal) DAG has many applications!

→ CSE (Common Subexpression Elimination) for efficient evaluation of expressions
   (do “term graph” rewriting, instead of term rewriting)

→ Model checking with BDDs
   Binary Decision Diagrams for efficient evaluation of logic formulas

→ Efficient XML query evaluation

Btw, inside of a DAG, you have “referential completeness”
→ structural equality = equality of pointers ☺
Every tree has a minimal, unique DAG!

The DAG is at most exponentially smaller than the tree.

Building the minimal unique DAG is easy!
Can be done in (amortized) linear time.

How?

(build a hash table of all subtrees seen so far)

(we don’t want to compare many trees, node by node, later on.)

Question Give a simple hash function that works for the tree above.

Hash Table HT

Hash function f

a hash "bucket"

We want

f distributes trees uniformly into buckets

test if a tree T is in HT, time O(size(T))

Example “Parse & DAGify”

1: startElement(c)

hash content

Example “Parse & DAGify”

1: bib [2,3,4,5]
2: book [6,7]
3: article [8,9]
4: book [10,11]
5: article [12,13]
6: author
7: title
8: author
9: title
10: price
11: title
12: price
13: title

Paragraphs

Example “Parse & DAGify”

1: bib [2,3,4,5]
2: book [6,7]
3: article [8,9]
4: book [10,11]
5: article [12,13]
6: author
7: title
8: author
9: title
10: price
11: title
12: price
13: title
Minimal Unique DAGs

1: bib \([2, 3, 4, 5]\)
2: book \([6, 7]\)
3: article \([8, 9]\)
4: book \([10, 11]\)
5: article \([12, 13]\)
6: author
7: title
8: author
9: title
10: price
11: title
12: price
13: title

\[
\begin{align*}
\text{Minimal Unique DAG} & \quad 8 \text{ nodes} & \quad (\text{vs 13 nodes in the original tree})
\end{align*}
\]
Minimal Unique DAGs

Example: "Parse & DAGify"

1. startElement(c)
2. startElement(c)
3. startElement(a)
4. endElement(a)

1. p1=hashT.find(a)
2. if(p1==NULL) { p1=new("a-node",NULL,NULL)
3. hashT.insert(p1) }

Memory location p1 is a DAG with root node a, and child1-pointer=NULL, child2-pointer=NULL

-> must store children lists: [1,[p1] ]

children of root node (so far, none)
children of node (so far, one)

Memory location p1 is a DAG with root node a, and child1-pointer=NULL, child2-pointer=NULL

Example: "Parse & DAGify"

1. startElement(c)
2. startElement(c)
3. startElement(a)
4. endElement(a)
5. startElement(a)
6. endElement(a)

1. p2=hashT.find(a)
2. if(p2==NULL) { p2=new("a-node",NULL,NULL)
3. hashT.insert(p2) }

Memory location p2 is a DAG with root node a, and child1-pointer=NULL, child2-pointer=NULL
Minimal Unique DAGs

Example "Parse & DAGify"

1. startElement(c)
2. startElement(c)
3. startElement(a)
4. endElement(a)
5. startElement(a)
6. endElement(a)
7. endElement(c)

1. p = hashT.find(a) = p1
2. if (p == NULL) { p = new("c-node", p1, p1)
3. hashT.insert(p) }

store children lists: [ ] [ ]

Hash | Content
--- | ---
1 | p1
2 | p1

Memory location p is a DAG with root node c, and child1-pointer = p1, child2-pointer = p1

New children lists: [ ] [ ]

Hash | Content
--- | ---
1 | p1
2 | p1

New children lists: [ ] [ ]

Hash | Content
--- | ---
1 | p1
2 | p1

Assume - 100 element names
Example hash function:

hash = #elementName + 100 * #elementName(1st child) + 100 * 100 * #elementName(2nd child) + 100^3 * elementName(1st child of 1st ch.) + ... MOD sizeOf(hashT)
Minimal Unique DAGs

→ DOM interface to the DAG?
perspective / sibling as before
→ Updates can be expensive (copying!)

How to attach attribute & text nodes to the DAG?

Æ

DOM interface to the DAG?
perspective / sibling as before
→ Updates can be expensive (copying!)

How to attach attribute & text nodes to the DAG?

Æ

Store them separately in a table.
Index by e.g., Node number (in doc-order)
or number of attr/text nodes
Store index in each DAG node /
compute it online. (pre-traversal)

What about unranked, vs binary DAGs?

More precisely,

What about size of minimal-unique-unranked-DAG( Tree ) vs size of minimal-unique-binary-DAG( fCnS-enc( Tree ) )

Questions
Give a tree for which first is smaller than the second.
Give a tree for which the second is smaller than the first.

Unranked vs Binary Trees

Can it be vica versa? (min bin. DAG is smaller)

YES!

Æ

Has 18 edges
→ DAG of bin. coding only 12 edges

DAG compression is sensible to rank/unrankedness!
Unranked vs Binary Trees

Last comment on binary tree encodings / DAGs

YES: the binary trees become very "regular" (deep, to the right)

---

DAG w. multiplicities

Items (books/addresses/etc)

DAG w. multiplicities

Bin tree w. multiplicities

3. Minimal Unique DAGs

<table>
<thead>
<tr>
<th>input file</th>
<th>size of tree</th>
<th>size binary DAG set</th>
<th>size unique binary DAG set</th>
<th>XML size</th>
<th>XML size unique set</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryXML (577 MB)</td>
<td>10,961.388</td>
<td>1,929,245</td>
<td>23.2%</td>
<td>1,306,849</td>
<td>91.1%</td>
</tr>
<tr>
<td>DB (102.6 MB)</td>
<td>4,611.927</td>
<td>1,213,310</td>
<td>26.4%</td>
<td>222,554</td>
<td>8.5%</td>
</tr>
<tr>
<td>Elephant (58 MB)</td>
<td>2,847.727</td>
<td>1,454,454</td>
<td>68.9%</td>
<td>1,391,488</td>
<td>85.3%</td>
</tr>
<tr>
<td>XML-escape (176 MB)</td>
<td>2,040.062</td>
<td>2,403.4</td>
<td>0%</td>
<td>729.0</td>
<td>2.0%</td>
</tr>
<tr>
<td>XML-escape (241 MB)</td>
<td>2,751.346</td>
<td>681,192</td>
<td>24.9%</td>
<td>461,322</td>
<td>16.3%</td>
</tr>
<tr>
<td>XML-escape (221 MB)</td>
<td>2,751.346</td>
<td>681,192</td>
<td>24.9%</td>
<td>461,322</td>
<td>16.3%</td>
</tr>
<tr>
<td>XML-escape (156 MB)</td>
<td>237,372</td>
<td>1,713,464</td>
<td>38.6%</td>
<td>454,355</td>
<td>11.7%</td>
</tr>
<tr>
<td>XML-escape (112 MB)</td>
<td>405.546</td>
<td>40,565</td>
<td>0.2%</td>
<td>25,047</td>
<td>0.2%</td>
</tr>
<tr>
<td>XML-escape (166 MB)</td>
<td>3,042,220</td>
<td>830,304</td>
<td>22.2%</td>
<td>15</td>
<td>&gt;0.1%</td>
</tr>
<tr>
<td>XML-escape (246 MB)</td>
<td>3,042,220</td>
<td>14,504</td>
<td>0.4%</td>
<td>11,707</td>
<td>3.3%</td>
</tr>
</tbody>
</table>

“Efficient XML” & Binary XML

W3C working groups

→ Efficient XML Interchange Working Group (EXI)
http://www.w3.org/EXI/

→ XML Binary Characterization Working Group
http://www.w3.org/XML/Binary/

The Figure on the next slide is from the “EXI Measurement Note” — new version of the note came out 25 July 2007…!

Notional EXI Test Corpus & Measurement Overview

Motivation: define consistent EXI terminology for diverse document sets and measurement protocols

Minimal Unique DAGs

Assignment 2
build a minimal DAG for a tree (given in XML)

For simplicity, ignore attributes and text values.
→ only consider element nodes.

Build the DAG, while parsing the XML!

Construct a hash table which stores all (complete) distinct subtrees seen so far.

Clearly, we do not want to parse into DOM, and then pull things out of there.

Instead, we need a more flexible parser that gives as the freedom of what exactly to store, and how.

How to use SAX

Remember one of the promises of XML…

You never need to write a parser again!
How to use SAX

Remember one of the promises of XML…
You never need to write a parser again!

… but, of course if you want to build up your own (e.g. memory-efficient)
data structure, you need to “talk” to the parser.

You want to tell the parser:
- Give me low level access to the data:
  - Bracket by bracket,
  - Text-node by text-node.
- In “document order”.

SAX—Simple API for XML

- SAX (Simple API for XML) is, unlike DOM, not a W3C standard,
  but has been developed jointly by members of the XML-DEV mailing list (ca. 1998).
- SAX processors use constant space, regardless of the XML input
document size.
  - Communication between the SAX processor and the backend XML
    application does not involve an intermediate tree data structure.
  - Instead, the SAX parser sends events to the application whenever a
    certain piece of XML text has been recognized (i.e., parsed).
  - The backend acts on/ignores events by populating a callback
    function table.

Sketch of SAX’s mode of operations

- A SAX processor reads its input document sequentially and once
  only.
- No memory of what the parser has seen so far is retained while
  parsing. As soon as a significant bit of XML text has been
  recognized, an event is sent.
- The application is able to act on events in parallel with the parsing
  process.

SAX Events

- To meet the constant memory space requirement, SAX reports
  fine-grained parsing events for a document:

<table>
<thead>
<tr>
<th>Event</th>
<th>…reported when seen…</th>
<th>Parameters sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>startDocument</td>
<td>&lt;xml…&gt;</td>
<td>file path, length</td>
</tr>
<tr>
<td>startElement</td>
<td>&lt;f &gt;…&lt;/f &gt;</td>
<td>f, (f1, f2, …, fn)</td>
</tr>
<tr>
<td>endElement</td>
<td>&lt;/f &gt;</td>
<td>f</td>
</tr>
<tr>
<td>characters</td>
<td>text content</td>
<td>Unicode buffer ptr, length</td>
</tr>
<tr>
<td>comment</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>processingInstruction</td>
<td>&lt;w? w? ?&gt;</td>
<td>c, pid</td>
</tr>
</tbody>
</table>

N.B.: Event startDocument is sent even if the optional XML text declaration
should be missing.

Events shown:

<xml encoding="utf-8" ?>
<dibelle>
    <speaker="?">?
    <dibelle>?
    </speaker>
</dibelle>

N.B.: Some events suppressed (white space).
SAX Callbacks

- To provide an efficient and tight coupling between the SAX frontend and the application backend, the SAX API employs function callbacks. Before parsing starts, the application registers function references in a table in which each event has its own slot.

  - The application alone decides on the implementation of the functions it registers with the SAX parser.
  - Reporting an event $i$ then amounts to calling the function (with parameters) registered in the appropriate table slot.

In that way, you automatically receive only element nodes.

Of course you can use SAX for other things than building up a data structure.

E.g.

- answer path queries while parsing (on a "stream") (low memory consumption)

Example: Reimplement content.cc shown earlier for DOM (find all XML text nodes and print their content) using SAX (pseudo code):

```java
// register the callback
// we ignore all other events
SAXRegister (characters, printText);
SAXparse ();
return;
```

SAX and the XML Tree Structure

- Looking closer, the order of SAX events reported for a document is determined by a pre-order traversal of its document tree:

N.B. An Elem [Doc] node is associated with two SAX events, namely startElement and endElement [startDocument, endDocument].

For Assignment 2, you only need to register startElement and endElement. In that way, you automatically receive only element nodes.

Deadline: 6th April