**Reminder**

You can freely choose to program your assignments in

- C / C++ (xerces-c library not installed on cse, will be fixed shortly)
- Java

However, your code must compile with gcc / g++, javac, as installed on CSE linux systems!

**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>81 M</td>
<td>164 M</td>
</tr>
<tr>
<td>52 M</td>
<td>680 M</td>
</tr>
</tbody>
</table>

Text only, with one embracing element

Treebank, deep tree structure with short texts

... 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 it with what you need..."
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!

Outline

1. Tree Pointer Structures
2. Binary Tree Encodings
3. Minimal Unique DAGs
4. How to use SAX

How much memory for n-node binary tree?

\[
\text{length(label}_1\text{)} + 1 + \text{length(label}_2\text{)} + 1 + \ldots + \text{length(label}_n\text{)} + 1
\]

3 \times \text{length(pointer)} \times n

typical: 4 bytes
1. Consider binary trees

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

How much memory for n-node binary tree?

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

length(label_1) + 1
+ length(label_2) + 1
+ ... + length(label_n) + 1

3 * length(pointer) * n

Typical: 4 bytes

Can easily be optimized:
- E.g., store each distinct string only once!

Serialization to XML

<library><book>< ... > ... </book></library>

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

Often #distinct node labels is small: \( \leq 100 \).
- Fits in one Byte
Then, only 9 bytes per node.

MEM(n-node binary tree pointer struc, \#256 labels)

= SIZE(n-node binary tree in XML, average label length=2)

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

Nice
- Following pointers is fast!
- \( \Rightarrow \) much higher access speed!
- (than on doc seen as string)
- E.g., at root, get right-child.

Often #distinct node labels is small: \( \leq 100 \).
- Fits in one Byte
Then, only 9 bytes per node.

MEM(n-node binary tree pointer struc, \#256 labels)

= SIZE(n-node binary tree in XML, average label length=2)

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

(node, [parent(node)::parent(parent(node)):: ... ::root-node])
Tree pointer structures

Access speed of parentNode should be approx same, as in a native DOM.
→ What about access speed of nextSibling?

What is the run-time size of our “binary DOM-tree” data structure? (WC/average)

Question

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

Node

nodeName : DOMString

parentNode : Node

firstChild : Node

nextSibling : Node

childNodes : NodeList

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

(Node, (parent(Node)::parent(parent(Node)):: .. ::root-node))

interface Node { // NodeType const unsigned short ELEMENT_NODE = 1; const unsigned short ATTRIBUTE_NODE = 2; const unsigned short TEXT_NODE = 3; const unsigned short CDATA_SECTION_NODE = 4; const unsigned short ENTITY_REFERENCE_NODE = 5; const unsigned short ENTITY_NODE = 6; const unsigned short PROCESSING_INSTRUCTION_NODE = 7; const unsigned short COMMENT_NODE = 8; const unsigned short DOCUMENT_NODE = 9; const unsigned short DOCUMENT_TYPE_NODE = 10; const unsigned short DOCUMENT_FRAGMENT_NODE = 11; const unsigned short NOTATION_NODE = 12; readonly attribute DOMString nodeName; // raisers(DOMException) on setting attr DOMString nodeValue; // raisers(DOMException) on retrieval! readonly attribute unsigned short nodeType; readonly attribute Node parentNode; readonly attribute Node firstChild; readonly attribute Node lastChild; readonly attribute Node previousSibling; readonly attribute Node nextSibling.

How much memory for n-node binary tree?

How to add attributes and text nodes?

Text: can be in the label, starting with a special character (pb: 2 unused pointers)
→ Attributes: pointer to an aux. Data-Structure (expensive)
→ Attributes encoded as children with special labels

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

2. Consider unranked trees

Tree structure of XML: unranked trees!

Typically

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

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

Typically

N1 N2

→ How much memory for List[Node] of n nodes? 2^n pointers

Tree structure of XML: unranked trees!

Typically

N1 N2

→ How much memory for List[Node] of n nodes? 2^n pointers

More efficient possibilities:

(1) Use arrays. Store #children. n pointers + (log d) Bits

(2) Encode tree as binary tree.

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

In this way, a node of a binary tree needs 5 pointers (plus label info pointer...)

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

Typically

N1 N2

→ How much memory for List[Node] of n nodes? 2^n pointers

In this way, a node of a binary tree needs 5 pointers (plus label info pointer...)

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

Typically

N1 N2

→ How much memory for List[Node] of n nodes? 2^n pointers
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
Binary Tree Encodings

Any unranked tree can be encoded as a binary tree.

n-node unranked tree → "firstChild/nextSibling" encoding → n-node 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).

LESSON 2
- ... against memory hunger ...
- Use binary trees instead of unranked trees. (... or use efficient arrays)
  - Fast child-m access
  - Expensive to update (insert/delete)

Tree Pointer Structures

Question

Give a datatype for binary trees which stores only non-NIL pointers. Then, n-node tree: <n pointers

```
Type Node {
    label : Byte, binary tree
    left : Node, 2 pointers per node
    right : Node
}
```

Can we do with even less pointers?

n-node tree: 2n pointers

3. Minimal Unique DAGs

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

Can we do with even less pointers?

YES!
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.

Can we do with even less pointers?

n-node tree: 2n pointers \rightarrow Directed Acyclic Graph DAG

Share identical subtrees

18/19 pointers/nodes

10/6 pointers/nodes

3. Minimal Unique DAGs

Local optimizations

- Common subexpressions (CSE)
  - portion of expressions
  - repeated multiple times
  - computes same value
  - can reuse previously computed value

- Directed acyclic graph (DAG)
  - program representation
  - nodes can have multiple parents
  - no cycles allowed
  - exposes common subexpressions

Building a DAG for an expression

- maintain hash table for leaf expressions
- unique name for each node — its value number
- reuse nodes found in hash table

Btw, inside of a DAG, you have “referential completeness” structural equality = equality of pointers.

3. Minimal Unique DAGs

Consider the expression: \( a + a \times (b - c) + (b - c) + d \)

Tree

\[ \begin{align*}
    &+ \\
    &+ \\
    &- \\
    &- \\
    &d \\
    &a \\
    &b \\
    &c
\end{align*} \]

Directed acyclic graph

\[ \begin{align*}
    &+ \\
    &+ \\
    &- \\
    &- \\
    &d \\
    &a \\
    &b \\
    &c
\end{align*} \]

3. Minimal Unique DAGs

(minimal) DAGs have 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

(minimal) DAGs have 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
3. Minimal Unique DAGs

- 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?

(even while parsing)
- 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

We want
- f distributes trees uniformly into buckets
- test if a tree T is in HT, time O(size(T))

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

Example: "Parse & DAGify"

```
hash  content
1: startElement(c) 2: startElement(c)
```

Minimal Unique DAGs

Example: "Parse & DAGify"

```
hash  content
1: startElement(c) 2: startElement(c) 3: startElement(a)
```
Example: "Parse & DAGify"

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

\[
\text{Memory location } p1 \text{ is a DAG with root node } a, \text{ and child1-pointer}=NULL, \text{ child2-pointer}=NULL
\]

\[
\text{hash content}
\]

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

\[
\text{children of root node (so far, none)}
\]
\[
\text{children of c-node (so far, one)}
\]
\[
\text{children of c-node (so far, one)}
\]

\[
\text{must store children lists: } [1], [p1]
\]

\[
\text{hash content}
\]

1. p2=hashT.find(a)=p1
2. if(p2==NULL) { p2=new("a-node",NULL,NULL)
   hashT.insert(p2) }

\[
\text{hash content}
\]
Example "Parse & DAGify"

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

→ store children lists: [ [], [p1, p1] ]

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

Example "Parse & DAGify"

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

→ store children lists: [ [], [p1, p1] ] → use children list!!

Example "Parse & DAGify"

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

→ store children lists: [ [], [p1, p1] ] → now update!!

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

Example "Parse & DAGify"

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

→ store children lists: [ [], [p1, p1] ]

Assume ≤ 100 element names

Example hash function:

\[
\text{sizeOf(hashT)} = 100 + \text{#elementName(1st child)} + 100 + \text{#elementName(2nd child)} + 100 + \text{#elementName(1st child of 1st ch.)} + \ldots
\]
DOM interface to the DAG?

How to attach attribute & text nodes to the DAG?

Updates can be expensive (copying!)

How to 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 / or compute it online (pre-traversal)

What about unranked vs binary DAGs?

More precisely, size of minimal-unique-unranked-DAG(Tree) vs size of minimal-unique-binary-DAG(KCnS-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.

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

YES!

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)

Items (books/addresses/etc)

DAG

Items (books/addresses/etc)

DAG w. multiplicities

Bin tree w. multiplicities

3. 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 us the freedom of what exactly to store, and how.

Efficient XML & Binary XML

W3C working groups

→ 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...!

Efficient XML & Binary XML

W3C working groups

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

The Figure on the next slide is from the "EXI Measurement Note" -- new version of the note came out 25 July 2007...!

3. 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 us 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!

... 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 the parser to:
- Give low-level access to the data.
- Bracket by bracket,
text-node by text-node.

In document order.

SAX — Simple API for XML

SAX is, unlike DOM, not a W3C standard, but has been developed jointly by members of the XML-DEV mailing list (ca. 1998).

SAX processor use constant space, regardless of the XML input. SAX processor is in fact more efficient than DOM, as it does not maintain an intermediate tree data structure.

You want the parser to:
- Give low-level access to the data.
- Bracket by bracket,
text-node by text-node.

In document order.

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>Parameters sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>startDocument</td>
<td>event (optional)</td>
</tr>
<tr>
<td>startElement</td>
<td>elementName, attributes</td>
</tr>
<tr>
<td>endElement</td>
<td>elementName</td>
</tr>
<tr>
<td>text</td>
<td>text content</td>
</tr>
<tr>
<td>comment</td>
<td>comment content</td>
</tr>
<tr>
<td>processingInstruction</td>
<td>target, content</td>
</tr>
</tbody>
</table>

N.B. Event startDocument is sent even if the optional XML text declaration was not be reading XML.

SAX and DOM

XML and Entainment

XML and XML

XML and XML

XML and XML
For Assignment 2, you only need to register `startElement` and `endElement`. 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!)

---

**SAX Callbacks**

To provide an efficient and tight coupling between the SAX front end and the application back end, 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 `e`, then amounts to calling the function (with parameters) registered in the appropriate table slot.

---

**SAX and the XML Tree Structure**

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

---

**Sources**

- SAX: [http://www.saxproject.org](http://www.saxproject.org)
- Java API: [https://java.net/projects/xom](https://java.net/projects/xom)