XML and Databases

Lecture 2
Memory Representations for XML: Space vs Access Speed

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2009

Reminder

You can freely choose to program your assignments in

→ C / C++, or
→ Java

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

Assignment 1 is due Monday 23:59, **25th of March**!
Submit your code using

```
% give cs4317 ass1 filename.cpp

% give cs4317 ass1 filename.java
```


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

XML size		DOM process size	
81M	$\xrightarrow{x\,2}$	164M	Text only, with one embracing element
52M	$\xrightarrow{x\,13.1}$	680M	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 in 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!

→ Use a **completely different** approach to XML processing (→ SAX)

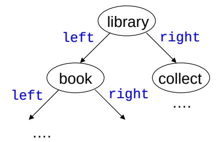
"design your own XML data structure
and fill in with what you need…"

Outline

1. Tree pointer structures

1. Consider *binary trees*

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

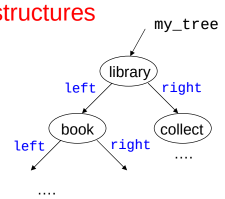


How much memory for n-node binary tree?

1. Tree pointer structures

1. Consider *binary trees*

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



How much memory for n-node binary tree?

```
cadr1: library0        my_tree: cadr1 tadr1 tadr2
cadr2: book0             tadr1: cadr2 tadr3 tadr4
…
```

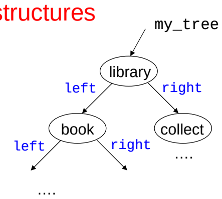
length(label_1) + 1
\+ length(label_2) + 1
\+ … + length(label_n) +1

3 * length(pointer) * n

1. Tree pointer structures

1. Consider *binary trees*

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



How much memory for n-node binary tree?

```
cadr1: library0        my_tree: cadr1 tadr1 tadr2
cadr2: book0             tadr1: cadr2 tadr3 tadr4
…
```

length(label_1) + 1
\+ length(label_2) + 1
\+ … + length(label_n) +1

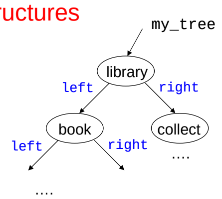
3 * length(pointer) * n

typical: **4 bytes**

Tree pointer structures

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

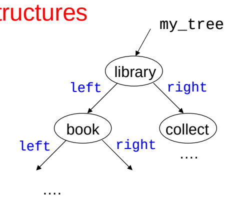
3 * length(pointer) * n

typical: **4 bytes**

Tree pointer structures

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**.

Can easily be optimized:

E.g., store each distinct string only *once*!

length(label_1) + 1
\+ length(label_2) + 1
\+ … + length(label_n) +1

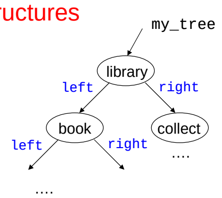
3 * length(pointer) * n

typical: **4 bytes**

Tree pointer structures

1. Consider *binary trees*

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



Serialization to XML

```
<library><book>< … > … </book></library>
^       ^                 ^^       ^
```

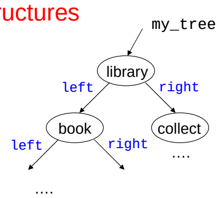
#characters per node: 5 + 2 * Length(label)

→ E.g., one node w. 4-character ASCI label: **13 bytes** (assuming UTF-8!)

Tree pointer structures

1. Consider *binary trees*

```
Type Node {   Byte
  label :  ~~String,~~
  left  :  Node,
  right :  Node
}
```



Often #distinct node labels is small, *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)

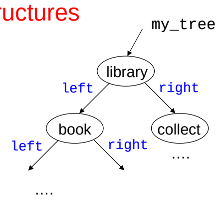
#characters per node: 5 + 2 * Length(label)

→ One node w. 2-character ASCI label: **9 bytes** (assuming UTF-8!)

Tree pointer structures

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 struc, ·256 labels)
= SIZE(n-node binary tree in XML, average label length=2)

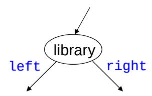
#characters per node: 5 + 2 * Length(label)

→ One node w. 2-character ASCI label: **9 bytes** (assuming UTF-8!)

Tree pointer structures

1. Consider *binary trees*

Plain no attributes, no text nodes, …



Question

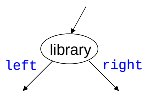
Using a (top-down) pointer structure, as the one above,
how can you implement a **DOM interface**?

```
Node     nodeName     : DOMString
         parentNode   : Node
         firstChild   : Node          leftmost child
         nextSibling  : Node          returns NULL for root elem
         childNodes   : NodeList
```


Tree pointer structures

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**?

```
Node     nodeName     : DOMString
         parentNode   : Node
         firstChild   : Node          leftmost child
         nextSibling  : Node          returns NULL for root elem
         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])
```


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)

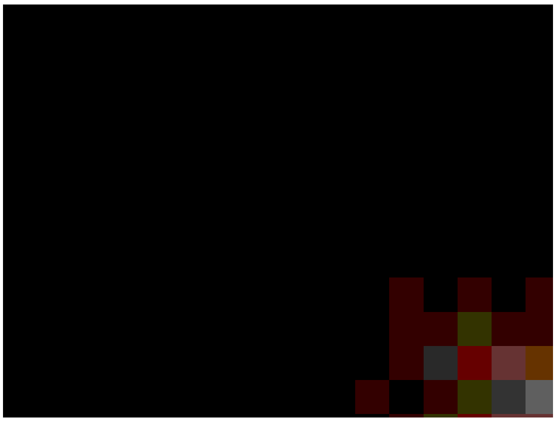
Question

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

```
Node     nodeName     : DOMString
         parentNode   : Node
         firstChild   : Node          leftmost child
         nextSibling  : Node          returns NULL for root elem
         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;
attribute DOMString nodeValue; // raises(DOMException) on setting
                               // raises(DOMException) on retrieval
readonly attribute unsigned short nodeType;
readonly attribute Node parentNode;
readonly attribute NodeList childNodes;
readonly attribute Node firstChild;
readonly attribute Node lastChild;
readonly attribute Node previousSibling;
readonly attribute Node nextSibling;
readonly attribute NamedNodeMap attributes;
readonly attribute Document ownerDocument;
Node insertBefore(in Node newChild, in Node refChild) raises(DOMException);
Node replaceChild(in Node newChild, in Node oldChild) raises(DOMException);
Node removeChild(in Node oldChild) raises(DOMException);
Node appendChild(in Node newChild) raises(DOMException);
boolean hasChildNodes(); Node cloneNode(in boolean deep); };
```

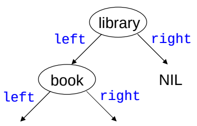

Tree pointer structures

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

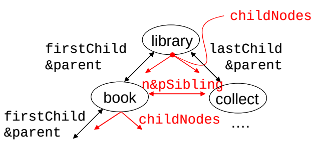
LESSON 1

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

binary trees



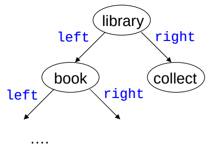
DOM



Tree pointer structures

1. Consider *binary trees*

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



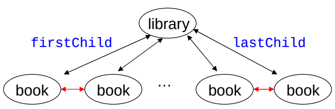
How much memory for n-node binary tree?

How to add *attributes and text nodes* ?

→ e.g., "into the label" …

Tree pointer structures

2. Consider ***unranked*** *trees*



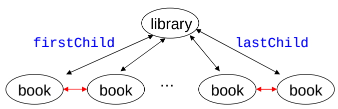
unranked = no a priori bound on #children of a node.

Tree structure of XML: **unranked trees! (not binary)**

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


Tree pointer structures

2. Consider ***unranked*** *trees*



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

Tree structure of XML: **unranked trees! (not binary)**

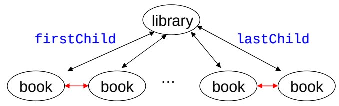
```
Type Node {
  label    :  String,
  children :  List[Node]
}
```

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

25

Tree pointer structures

2. Consider ***unranked*** trees

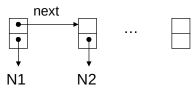


unranked = no a priori bound on #children of a node.

Tree structure of XML: **unranked trees!**

Typically

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

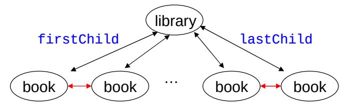


→ How much memory for `List[Node]` of n nodes?

26

Tree pointer structures

2. Consider ***unranked*** trees

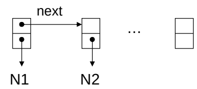


unranked = no a priori bound on #children of a node.

Tree structure of XML: **unranked trees!**

Typically

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



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

27

Tree pointer structures

2. Consider ***unranked*** trees

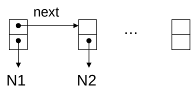
→ In this way, a node of a *binary tree* needs **5 pointers** ☹
(*plus* label info/pointer..)

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

Tree structure of XML: **unranked trees!**

Typically

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



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

28

Tree pointer structures

2. Consider ***unranked*** trees

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

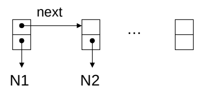
More efficient possibilities:

(1) Use **arrays**. Store #children (e.g., in label). n pointers + (log d) Bits

(2) Encode tree as binary tree.

Typically

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



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

29

Tree pointer structures

2. Consider ***unranked*** trees

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

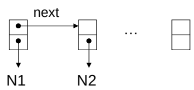
More efficient possibilities:

(1) Use **arrays**. Store #children (e.g., in label). n pointers + (log d) Bits

(2) ➔ Encode tree as binary tree. 🡐

Typically

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



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

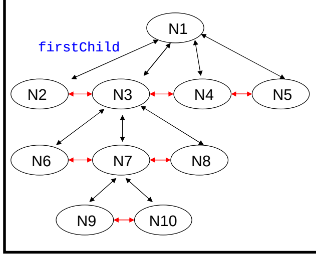
30

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

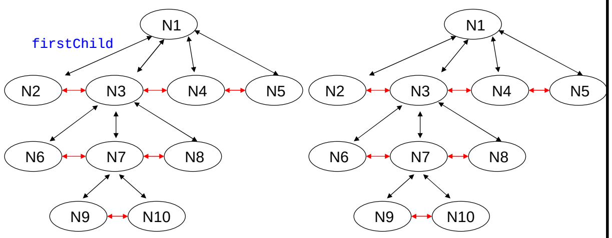


2. Binary Tree Encodings

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

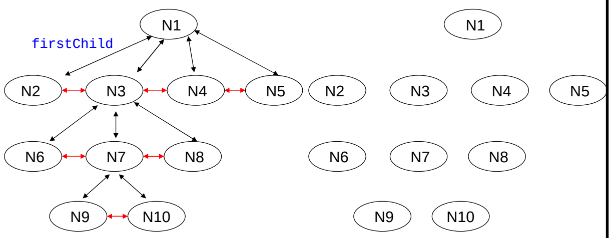


2. Binary Tree Encodings

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

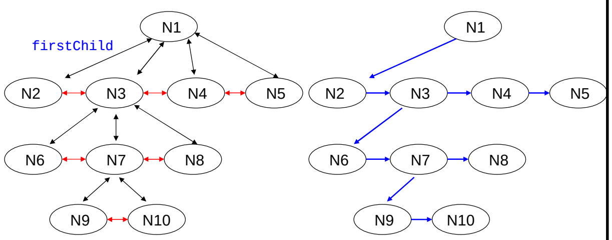


2. Binary Tree Encodings

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

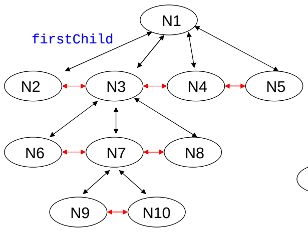


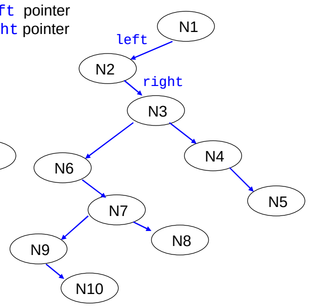
2. Binary Tree Encodings

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



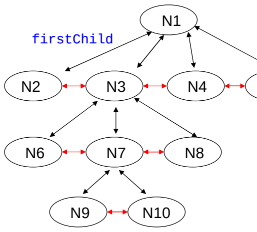


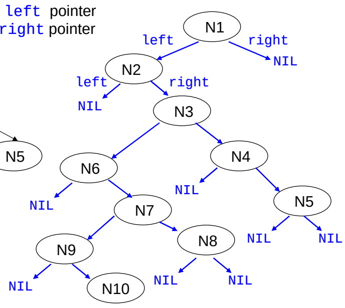
Binary Tree Encodings

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





Binary Tree Encodings

Any unranked tree can be encoded as a binary tree.

n-node unranked tree	"firstChild/nextSibling" encoding → ← decoding	*n-node binary tree*

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.

n-node unranked tree —**"firstChild/nextSibling" encoding**→ *n-node binary tree*

←decoding—

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).

Binary Tree Encodings

Any unranked tree can be encoded as a binary tree.

n-node unranked tree —**"firstChild/nextSibling" encoding**→ *n-node binary tree*

←decoding—

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 :  String,
  left  :  Node,
  right :  Node
}
```


3. Minimal Unique DAGs

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

```
Type Node {
  label :  Byte,
  left  :  Node,
  right :  Node
}
```

binary tree
2 pointers per node

Can we do with even less pointers?

3. Minimal Unique DAGs

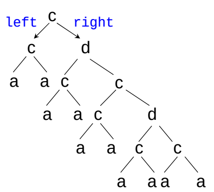
L1: no backward pointers
L2: use binary trees or efficient arrays

```
Type Node {
  label :  Byte,
  left  :  Node,
  right :  Node
}
```

binary tree
2 pointers per node

Can we do with even less pointers?

n-node tree: 2n pointers



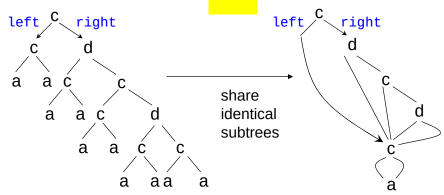
3. Minimal Unique DAGs

```
Type Node {
  label :  Byte,
  left  :  Node,
  right :  Node
}
```

binary tree
2 pointers per node

Can we do with even less pointers?

n-node tree: 2n pointers **YES!** → **D**irected **A**cyclic **G**raph DAG



3. Minimal Unique DAGs

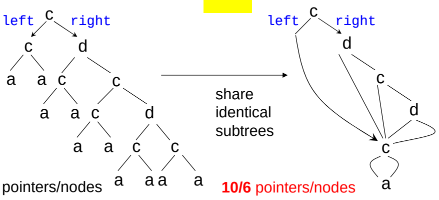
```
Type Node {
  label :  Byte,
  left  :  Node,
  right :  Node
}
```

binary tree
2 pointers per node

"**18** pointers" really means "**18** non-NIL pointers"

Can we do with even less pointers?

n-node tree: 2n pointers **YES!** → **D**irected **A**cyclic **G**raph DAG



18/19 pointers/nodes

10/6 pointers/nodes

3. Minimal Unique DAGs

A DAG representation of a tree has always

→ Less than or equal #nodes than the tree

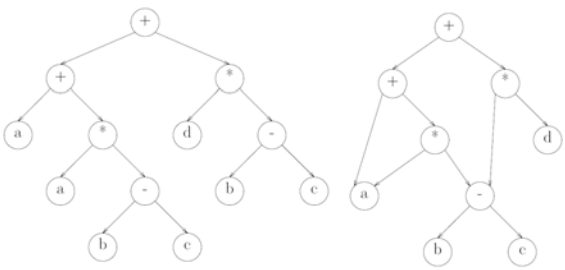
→ Less than or equal #pointers than the tree.

3. Minimal Unique DAGs

Local optimizations

Consider the expression: `a + a * ( b - c ) + ( b - c ) * d`

Tree Directed acyclic graph



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 leafs, expressions
- unique name for each node — its *value number*
- reuse nodes found in hash table

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*

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*

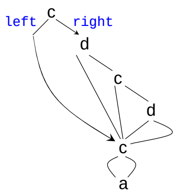
Btw, inside of a DAG, you have "referential completeness"
→ structural equality = equality of pointers ☺

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*.



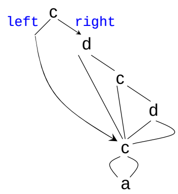
3. Minimal Unique DAGs

→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?



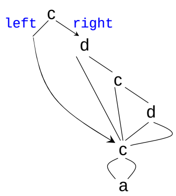
3. Minimal Unique DAGs

→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?



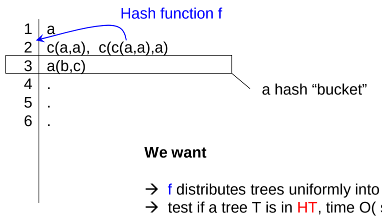
(even while parsing)
→Build a *hash table* of all subtrees seen so far

(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.

3. Minimal Unique DAGs

Hash Table HT



We want

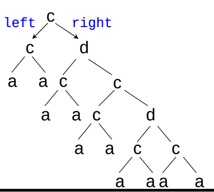
→ f distributes trees uniformly into buckets
→ test if a tree T is in HT, time O(size(T))

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

Minimal Unique DAGs

Example "Parse & DAGify"

```
1: startElement(c)
```



hash	content

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
```

```
<bib>
   <book>
      <author></author>
      <title></title>
   </book>
   <article>
      <author></author>
      <title></title>
   </article>
   <book>
      <price></price>
      <title></title>
   </book>
   <article>
      <price></price>
      <title></title>
   </article>
</bib>
```

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

6

```
<bib>
  <book>
    <author></author>
    <title></title>
  </book>
  <article>
    <author></author>
    <title></title>
  </article>
  <book>
    <price></price>
    <title></title>
  </book>
  <article>
    <price></price>
    <title></title>
  </article>
</book>
```


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~~

6
7

```
<bib>
  <book>
    <author></author>
    <title></title>
  </book>
  <article>
    <author></author>
    <title></title>
  </article>
  <book>
    <price></price>
    <title></title>
  </book>
  <article>
    <price></price>
    <title></title>
  </article>
</book>
```


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~~

6
7
10

```
<bib>
  <book>
    <author></author>
    <title></title>
  </book>
  <article>
    <author></author>
    <title></title>
  </article>
  <book>
    <price></price>
    <title></title>
  </book>
  <article>
    <price></price>
    <title></title>
  </article>
</book>
```


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~~

6
7
10

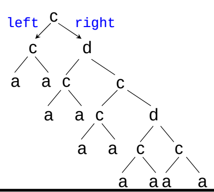
```
<bib>
  <book>
    <author></author>
    <title></title>
  </book>
  <article>
    <author></author>
    <title></title>
  </article>
  <book>
    <price></price>
    <title></title>
  </book>
  <article>
    <price></price>
    <title></title>
  </article>
</book>
```

minimal unique DAG
8 nodes (vs 13 nodes in the original tree)

Minimal Unique DAGs

Example "Parse & DAGify"

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

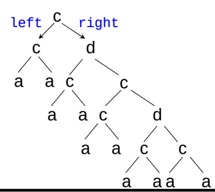


hash	content

Minimal Unique DAGs

Example "Parse & DAGify"

```
1: startElement(c)
2: startElement(c)
3: startElement(a)
```



hash	content

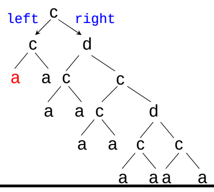
61

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)
                  hashT.insert(p1) }
```



hash	content

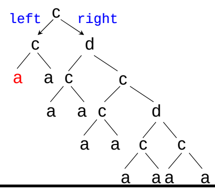
62

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)
                  hashT.insert(p1) }
```



hash	content
1	
2	p1

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

63

Minimal Unique DAGs

Example “Parse & DAGify”

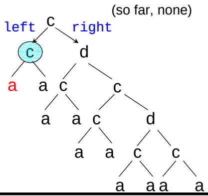
```
1: startElement(c)
2: startElement(c)
3: startElement(a)
4: endElement(a)
```

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

➔ must store children lists: `[ [],[p1] ]`

children of root node (so far, none)

children of c -node (so far, one)



hash	content
1	
2	p1

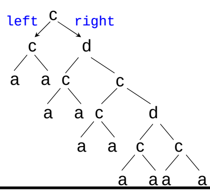
Memory location p1 is a DAG with root node a, and child1-pointer=NULL child2-pointer=NULL

64

Minimal Unique DAGs

Example “Parse & DAGify”

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



hash	content
1	
2	p1

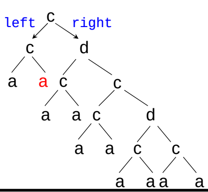
65

Minimal Unique DAGs

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)
                  hashT.insert(p2) }
```



hash	content
1	
2	p1

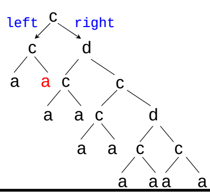
66

Minimal Unique DAGs

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)=p1
2. if(p2==NULL) ~~{ p2=new(“a-node”,NULL,NULL)~~
~~hashT.insert(p2) }~~



hash	content
1	
2	p1

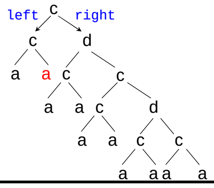
Minimal Unique DAGs

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

➔ store children lists: [[],[p1,p1]]



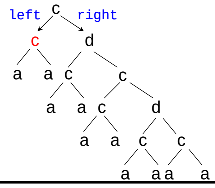
hash	content
1	
2	p1

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)
```

➔ store children lists: [[],[p1,p1]]

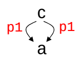


hash	content
1	
2	p1

Minimal Unique DAGs

Example "Parse & DAGify"

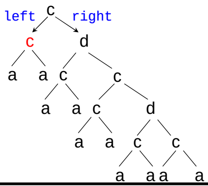
```
1: startElement(c)
2: startElement(c)
3: startElement(a)
4: endElement(a)
5: startElement(a)
6: endElement(a)
7: endElement(c)
```




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

➔ store children lists: [[],[p1,p1]]

➔ use children list!!



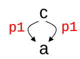
hash	content
1	
2	p1
.	
.	
20201	p

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

Minimal Unique DAGs

Example "Parse & DAGify"

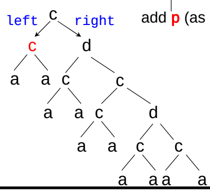
```
1: startElement(c)
2: startElement(c)
3: startElement(a)
4: endElement(a)
5: startElement(a)
6: endElement(a)
7: endElement(c)
```




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

➔ store children lists: [[],[~~p1,p1~~]] **←now update!!**

add p (as last)



hash	content
1	
2	p1
.	
.	
20201	p

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

Minimal Unique DAGs

Example "Parse & DAGify"

New children lists: [[p]]

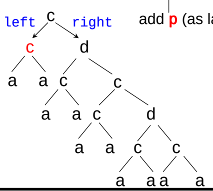
children of root node (so far, one)

```
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)
2. if(p==NULL) { p=new("c-node",p1,p1)
       hashT.insert(p) }
```

➔ store children lists: [[],[~~p1,p1~~]] **←now update!!**

add p (as last)



hash	content
1	
2	p1
.	
.	
20201	p

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

Minimal Unique DAGs

Example "Parse & DAGify"

New children lists: [[p]]

children of root node (so far, one)

```
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)
2. if(p==NULL) { p=new("c-node",p1,p1)
       hashT.insert(p) }
```

hash	content
1	
2	p1
.	
.	
20201	p

→Assume · 100 element names
Example hash function:

```
(#elementName
  + 100 * #elementName(1st child)
  + 100 * 100 * #elemNa(2nd child)
  + 100^3 * elemNa(1st child of 1st ch.)
  + … ) MOD sizeOf(hashT)
```

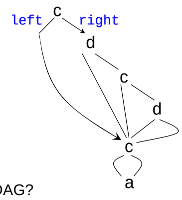
73

Minimal Unique DAGs

→ DOM interfact to the DAG?

`parentNode` / `p&nsibling` as before

→ Updates can be expensive (copying!)



How to attach attribute & text nodes to the DAG?

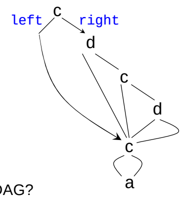
74

Minimal Unique DAGs

→ DOM interfact to the DAG?

`parentNode` / `p&nsibling` as before

→ Updates can be expensive (copying!)



How to attach attribute & text nodes to the DAG?

→Store them seperately in a table.

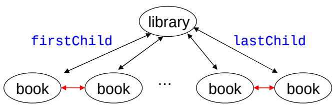
Index by e.g., Node number (in doc-order)
or number of atr/text nodes

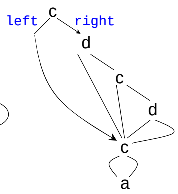
Store index in each DAG node / or compute it online. (pre-traversal)

75

Minimal Unique DAGs

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))

firstChild/nextSibling

76

Minimal Unique DAGs

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

Questions

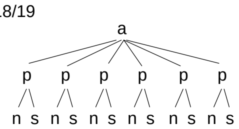
Give a tree for which first is smaller than the second.

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

77

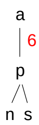
Unranked vs Binary Trees

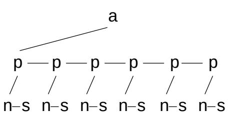
18/19



→ Min. DAG

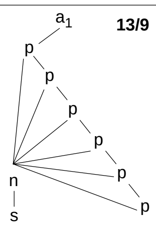
3/4





→ Min. DAG

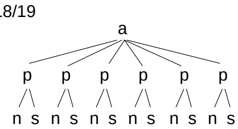
13/9



78

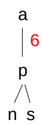
Unranked vs Binary Trees

18/19



→ Min. DAG

3/4

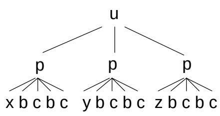


Can it be vica versa? (min bin. DAG is smaller)

YES!!

→ Has **18 edges**

→ DAG of bin.coding only **12 edges**

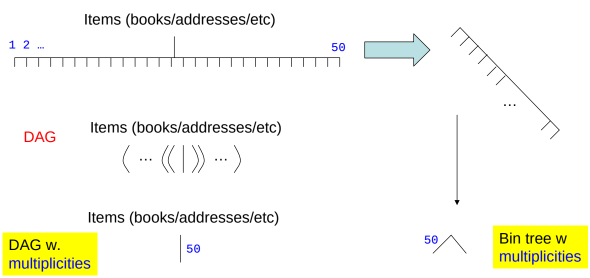


DAG compression is sensible to rank/unrankedness!

Unranked vs Binary Trees

Last comment on binary tree encodings / DAGs

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



3. Minimal Unique DAGs

input file	size of tree	min. binary DAG size		min. unranked mDAG size		BPLEX output size	
SwissProt (457,4 MB)	10,903,568	1,437,445	13.2%	1,100,648	10.1%	311,328	2.9%
DBLP (103.6 MB)	2,611,931	533,183	20.4%	222,754	8.5%	115,902	4.4%
Treebank (55.8 MB)	2,447,727	1,454,494	59.4%	1,301,688	53.2%	519,542	21.2%
1998statistics (657 KB)	28,306	2,403	8.5%	726	2.6%	410	1.4%
catalog-02 (104M)	2,240,231	52,392	2.3%	32,267	1.4%	26,774	1.2%
catalog-01 (11M)	225,194	6,990	3.1%	8,503	2.8%	3,817	1.7%
dictionary-02 (104M)	2,731,764	681,130	24.9%	441,322	16.2%	160,329	5.9%
dictionary-01 (11M)	277,072	77,554	28.0%	46,993	17.0%	20,150	7.3%
JST_snp.chr1 (36M)	655,946	40,663	6.2%	25,047	2.3%	12,858	1.8%
JST_gene.chr1 (11M)	216,401	14,606	6.7%	5,658	2.6%	4,000	1.8%
NCBI_snp.chr1 (190M)	3,642,225	809,394	22.2%	15	<0.1%	59	<0.1%
NCBI_gene.chr1 (24M)	360,350	14,356	4.0%	11,767	3.3%	7,160	2.0%

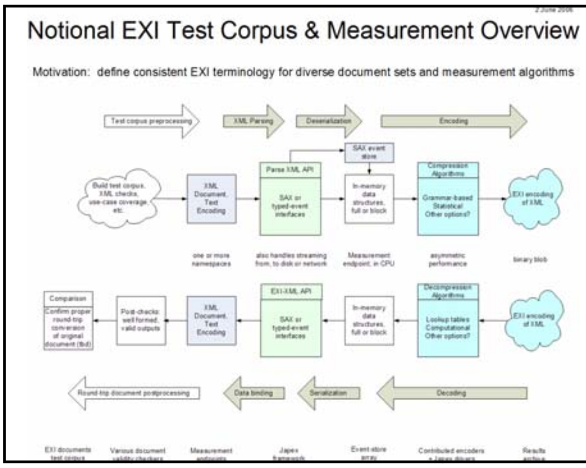
“*Efficient XML*” & Binary XML

W3C working groups

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



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.

Cleary, 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".

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".

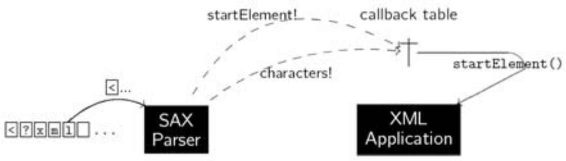
→ SAX

SAX—Simple API for XML

- **SAX**[7] (**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**.

[7] `http://www.saxproject.org/`

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 progress.

SAX Events

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

Event	…reported when seen	Parameters sent
startDocument	<?xml…?>[8]	
endDocument	⟨EOF⟩	
startElement	<t a_1=v_1 … a_n=v_n>	$t, (a_1, v_1), \ldots, (a_n, v_n)$
endElement	</t>	t
characters	*text content*	Unicode buffer ptr, length
comment	<!--c-->	c
processingInstruction	<?t pi?>	t, pi
	⋮	

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

dilbert.xml

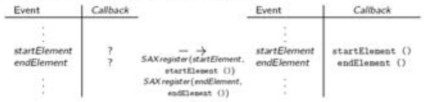
```
<?xml encoding="utf-8"?> ⋆1
<bubbles> ⋆2
  <!-- Dilbert looks stunned --> ⋆3
  <bubble speaker="phb" to="dilbert"> ⋆4
    Tell the truth, but do it in your usual engineering way
    so that no one understands you. ⋆5
  </bubble> ⋆6
</bubbles> ⋆7 ⋆8
```

Event[9] [10]		Parameters sent
$\star_1$	*startDocument*	
$\star_2$	*startElement*	t = "bubbles"
$\star_3$	*comment*	c = "␣Dilbert looks stunned␣"
$\star_4$	*startElement*	t = "bubble", ("speaker","phb"), ("to","dilbert")
$\star_5$	*characters*	buf = "Tell the…understands you.", $len = 99$
$\star_6$	*endElement*	t = "bubble"
$\star_7$	*endElement*	t = "bubbles"
$\star_8$	*endDocument*	

[9] Events are reported in **document reading order** $\star_1, \star_2, \ldots, \star_8$.
[10] **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**:[11]
  1. Before parsing starts, the application **registers function references** in a table in which each event has its own slot:



  2. The application alone decides on the implementation of the functions it registers with the SAX parser.
  3. **Reporting an event** $\star_i$ then amounts to call the function (with parameters) registered in the appropriate table slot.

[11] Much like in event-based GUI libraries.

Java SAX API

In Java, populating the callback table is done via implementation of the SAX `ContentHandler` interface: a `ContentHandler` object represents the callback table, its methods (*e.g.*, `public void endDocument ()`) represent the table slots.

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

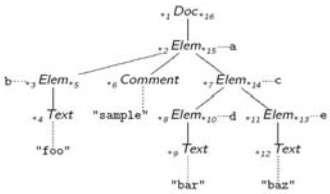
```
content (File f)
  // register the callback,
  // we ignore all other events
  SAXregister (characters, printText);
  SAXparse (f);
  return;

printText ((Unicode) buf, Int len)
  Int i;
  foreach i ∈ 1 ... len do
    print (buf[i]);
  return;
```


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[12]:

Sample XML document

```
*1
<a>*2
  <b>*3 foo*4 </b>*5
  <!--sample-->*6
  <c>*7
    <d>*8 bar*9 </d>*10
    <e>*11 baz*12 </e>*13
  </c>*14
</a>*15 *16
```



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

[12] Sequences of sibling *Char* nodes have been collapsed into a single *Text* node.

Deadline: 6th April

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!)

END
Lecture 2