SPACE EFFICIENT IN-MEMORY REPRESENTATION OF XML DOCUMENTS

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

O'Neil Delpratt BSc (Leicester)

Department of Computer Science

University of Leicester

October 2008

Author's Declaration

I hereby declare that this submission is my own work and that it is the result of work done mainly during the period of registration. To the best of my knowledge, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

Parts of this submission appeared in the following conjoint publications, to each of which I have made substantial contributions:

- Delpratt, O., Rahman, N., and Raman, R. 2006. Engineering the LOUDS Succinct Tree Representation. In *Proc. of 5th Workshop on Experimental Algorithms* (Menorca, Spain, May 24-27, 2006) WEA '06. LNCS, Springer-Verlag, Berlin Heidelberg New York, NY, Vol. 4007, pp. 134-145.
- Delpratt, O., Rahman, N., and Raman, R. 2007. Compressed Prefix Sums. In *Proc. of the Theory and Practice of Computer Science* (Harrachov, Czech Republic, January 20-26, 2007). SOFSEM '07. LNCS, Springer-Verlag, Berlin Heidelberg New York, NY, Vol. 4362, pp. 235-247.
- Delpratt, O., Raman, R., and Rahman, N. 2008. Engineering succinct DOM. In *Proceedings of the 11th international Conference on Extending Database Technology: Advances in Database Technology* (Nantes, France, March 25-29, 2008). EDBT '08, Vol. 261. ACM International Conference Proceeding Series, New York, NY, pp. 49-60.

SPACE EFFICIENT IN-MEMORY REPRESENTATION OF XML DOCUMENTS

O'Neil Delpratt

Abstract

Extensible Markup Language (XML) is a multi-purpose text-based format, used for storage, transmission and manipulation of data. XML documents are often held in main memory and processed via standard interfaces such as the Document Object Model (DOM). However, XML is inherently verbose, and the in-memory representation of XML documents by existing DOM implementations is up to ten times larger than the file size. This is a problem for machines with limited memory, such as mobile devices, where processing even moderately-sized XML documents requires more memory than is available. We focus on in-memory representations of XML documents for situations where space is limited and where rapid processing time is important. We propose a compact representation of XML documents that uses *succinct* or highly space-efficient data structures, that allows XML processing to be executed efficiently.

Succinct data structures use space that approaches the information-theoretic lower bound on the space that is required to represent the data, and support operations upon the representation in constant time. In the context of XML documents, we study and improve succinct representations for ordinal trees by adding features that make them more suitable for use in XML documents. We explore fast and space-efficient representations of the textual data of XML documents. Our basic approach is to concatenate all the textual data in the XML document into a single string, and extract individual textual values by computing the appropriate substring of the concatenated string. Computing the substring requires us to store offsets into the text. The storage of the offsets is surprisingly expensive, if stored naively (as 32 or 64-bit integer values). We give a succinct representation and provide data-aware representations (adapted from work on inverted indices in information retrieval), and show their close connection.

We describe *Succinct DOM* (*SDOM*), which is a DOM implementation that has low, stable and predictable memory usage. We show, via an experimental evaluation, that SDOM is extremely fast. A variant, SDOM-CT, applies BZip-based compression to textual and attribute data, and its space usage is comparable with "query-friendly" XML compressors. Some of these compressors support navigation and/or querying (e.g. subpath queries) of the compressed file. SDOM-CT does not support querying directly, but remains extremely fast: it is several orders of magnitude faster for navigation than query-friendly XML compressors that support navigation (and only a few times slower than popular DOM implementations such as the Apache Foundation's Xerces-C).

Acknowledgements

Firstly, I thank the Lord Jesus Christ, who took me through this course and made the way for me to complete it. He provided me great people to work with, who took interest in my accomplishments. I therefore express my sincere gratitude to them all.

I thank my supervisor Rajeev Raman, who, while I was an undergraduate, sparked the interest and desire that was in me, to start and follow through the journey to complete the PhD. During my period of study, he gave me fantastic support, guidance, help and aided my professional development. He has surely made a great impact upon me, which shall be with me throughout the future. Naila Rahman wrote the code for the succinct data structures that underline SDOM, and I am very grateful for her help in understanding the working of these data structures, for putting in the work to port the code to a newer version of the GNU C++ compiler. Without these two very special people, this thesis would not have been possible. Special thanks to Richard Geary for very useful discussions in the early stages of SDOM.

Clive Page and Tony Linde, from the department of Physics, were my co-supervisors. They offered useful advice and an invaluable external perspective. I express special thanks to all the members of the Computer Science Department at the University of Leicester, notably Rick Thomas for his support throughout the years. Thanks also to the PhD tutor, Fer-Jan de Vries and to Thomas Erlebach for their encouragement, advice and assessing yearly reports presented to them.

I convey my gratitude to my PhD colleagues, Cristóvão Oliveira, Osama El-Hassan, João Abreu and Ahmed Al-Ghamdi for the many useful discussions throughout my time at the University.

My PhD studies were supported by PPARC e-Science Studentship PPA/S/E2003/03749 (co-investigators: Tony Linde, Clive Page, Rajeev Raman and Mike Watson).

Special thanks to my examiners, Peter Wood and Thomas Erlebach, for enduring the work represented to them and their recommendations to steer my thesis to a comprehensible document.

To the special people in my life, my father, Bishop Mark Anderson, my mother, Sharon Anderson, and my brother, Jordan, my gratitude cannot be put into words. They have been a great support in my development; without them, studying at a university would have been a mere thought. Dad has given me the drive to push beyond my limitation and to achieve my greatest potential. I thank my dearest brethren at the Emmanu-‘El Apostolic Church who have been very supportive especially Sister Ellah Kandi and Pastor Samuel Gapara.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

The World Wide Web Consortium (W3C) [79] introduced the *Extensible Markup Language* (XML) specification as a text-based platform-neutral and customizable markup language. The first use of XML was in the late 1990s; it has become a powerful complement to HTML. XML is a multipurpose data format that is well-suited to the representation of complex, hierarchically structured data. Its uses in data exchange, storage and retrieval have reached much further than its creators may have anticipated.

For data exchange, standards exist in Service-Oriented Computing that provide communication between applications and devices based upon XML. These include the Web-Services Description Language (WSDL), Universal Description, Discovery, and Integration (UDDI) and Simple Object Access Protocol (SOAP). In addition, web pages are now represented in XML, such as XHTML.

XML as a storage format represents structured data, such as tables. The data format used in the storage is often standardised, for example, the VO-Table XML format [74] represents scientific data with emphasis on astronomical data (e.g. Astrogrid [74]), and we have the MEDLINE XML format [50] that is used to represent the Medline bibliographic citation database. Word processing applications are now using the XML format as their document representation; these include Microsoft Office 2007 and Open Office. Many companies use XML for their technical documentation based upon the standard format called DocBook [25]. DocBook enables its users to focus on capturing the logical structure of the content, which can then be published in a variety of formats (e.g. HTML, PDF etc).

The retrieval of data in XML is a powerful feature. XML is used in databases, with a number of query languages that have been developed (e.g. XQuery) to provide access and retrieval of the data, just like the SQL standard for traditional database systems.

1.1 XML Processing

A large and growing set of specifications describe the processing of XML documents. We focus on two low-level processing of XML documents; the first is the Simple API for XML (SAX) [60], which provides event-driven functionality used for stream

processing. The second is the Document Object Model (DOM) APIs, which requires a pre-processing phase to construct a representation of the document. DOM and SAX are often used as the underlying engine in many higher-level processors; we will see examples of these later.

The SAX parser provides sequential access to the XML document. It reads the document from the beginning to the end, and the recent data read is provided to the user through call-back event methods. The user is then required to manage the data received, as SAX does not keep track of data that has been read.

The DOM represents XML documents as an in-memory representation; the XML document is parsed, sometimes using a SAX parser, to create the DOM tree structure. The DOM provides access to all parts of the representation of the XML document through the navigation operations.

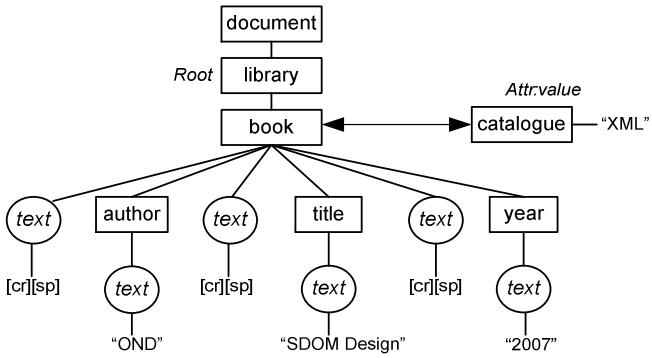
An example of the DOM is given in Figure 1.1, where the XML document in (a), is represented as the DOM document tree in (b). We observe in (b) the square shaped nodes represent the *elements* in the XML document, also the circular nodes represent the element's content and the single attribute in the document is mapped to the library element, where it is defined.

SAX is extremely fast to read the XML document, and has very little memory requirement, whereas DOM has to load the entire document before data can be read. However, DOM provides flexibility of repeated navigation, retrieval and/or update on the document. It is simpler to develop applications using DOM than only using a SAX parser, for the reason that SAX requires the user to supplement the call-back methods and to maintain the data received. Furthermore, in situations where we require repeated navigation upon the XML document SAX is simply not sufficient.

DOM serves as a general-purpose tool that can be used in applications, stand-alone or with other standards such as XPath [71], XSLT [72] and XQuery; these are the high level processors. We observe XSLT processors (such as Xalan [66] and Saxon [61]) rely on the DOM [66] or simpler tree structure representations. The language neutral DOM is supported in most programming languages such as JavaScript, Perl, Java,

```
<library>
  <book catalogue="XML">
   <author>OND</author>
   <title>SDOM Design</title>
   <year>2007</year>
 </book>
...
</library>
```

(a)



(b)

Figure 1.1 – (a) Simple XML document. (b) Corresponding DOM tree.

ActiveX, Python, C/C++, PHP and ASP, etc. In addition, web browsers use DOM parsers to load XML documents, such as the Microsoft IE 5 and higher, which incorporates their XML parser to build the DOM documents.

1.2 Memory Architecture

To help understand the problems associated in representing and processing XML documents, we give an overview of the memory architecture of a computer. The central processing unit (CPU) of a computer receives instructions, decodes them and performs a sequence of operations, given from a program, on data held in its memory. Data is stored in the following types of memory [19]:

- the *hard disk*, which provides a permanent storage of the data, even when the machine is switched off. This type of memory is the largest and the cheapest. Data here is not directly accessible by the CPU, but is first loaded into RAM memory.

- the *Random-Access memory* (RAM), (or main memory), which provides data storage whilst the computer is on. Main memory is much faster than disk, but a lot less in capacity. Main memory is connected to the CPU through a *memory bus*, which has a bandwidth or maximum throughput for transferring data. A software program loads data from the disk to the RAM memory.

- the *cache*, which is an intermediate storage between the CPU and the main memory. Cache is much faster than RAM memory, but is much more expensive

and hence smaller than the main memory. It is used to reduce the average time to access memory. If the CPU requests data stored at a certain memory address, this data and the data stored in nearby memory locations is brought into the cache (up to the cache's maximum size). Due to *locality of reference*, it is normally the case that requests for data from the main memory will be served from the cache [18].

In summary, programs that make better use of fast memory are usually executed much faster. In particular, a program that is designed for use in RAM but uses so much memory that some of its data is stored in virtual memory on disk, may potentially exhibit *thrashing* where data is repeatedly read and written back from RAM to virtual memory. When thrashing occurs, a system will slow down to an extent that it appears to hang.

1.3 XML Bloat

XML is inherently a verbose representation. XML adds *tags* to a flat text file to separate the document into sections, or to indicate the meaning of the text enclosed in the tags. The addition of meta-data (tags) to flat files, can easily triple its size; also, XML files are nearly always much larger than comparable binary formats. This problem with XML documents is what we call '*XML bloat*'. XML bloat becomes a problem for mobile devices that have a very limited memory space, such as PalmTops or PDAs, and increases transmission times and storage/backup costs for PCs and servers.

For a desktop PC or server, XML bloat is still a problem, particularly if the document is processed using DOM. The DOM exacerbates the problem of XML bloat in its tree representation of the document. Existing DOM implementations maintain the entire DOM tree in main memory, as it is faster. However, these implementations suffer from a high memory usage: Table 1.1 illustrates how much larger the in-memory DOM representation (of the standard Xerces-C implementation) is than the (already bloated) XML file. Thus, loading even a moderate size XML file using DOM may lead to thrashing. Thus, solving these problems arising in the storage and processing of XML data is an important research topic in the computing community.

Table 1.1 – Memory usage of representing XML documents in Xerces-C, as a percentage of the original file size.

File	File Size	Xerces-C
Orders.xml	5MB	451%
Lineitem.xml	32MB	399%

1.3.1 XML compression

One way to address the space consumption of XML documents is through data compression. Compression has a number of positive effects: in addition to space saving, better use of memory levels closer to the processor, increased disk and memory bandwidth and reduced (mechanical) seek time**.** Standard text compression like GZip does not compress the XML-specific files as well as XML compressors, such as XMill [48], which achieve very good compression ratios. However, XMill does not support processing operations, such as navigation upon the compressed representation. A number of *query-friendly* XML compressors have recently been developed (see e.g., [3], [10], [14], [30], [48], [52], [55], [64], [75], [80]). The characteristic of a query-friendly compressor is that answering the query involves inspection only of a (usually small) fraction of the XML file, and in principle, only a fraction of the compressed file must be decompressed as well. However, few of these compressors ([10], [30]) support DOM-like navigation, and those that do, are significantly slower than standard DOM implementations.

1.3.2 Our approach

In summary, existing XML compression software partially addresses XML bloat, but little has been done to efficiently support the processing operations, such as navigation of the documents, on an in-memory representation. The objectives of this thesis are as follows:

(a) To develop a space-efficient in-memory representation of XML documents with memory usage, an order of magnitude less than existing DOM implementations.

(b) Fast support for DOM operations at a speed that is comparable to standard DOM implementations.

The basic intuition underlying the approach in this thesis is as follows. The high memory usage of XML DOM implementations is largely due to the use of pointers for maintaining the relationships between nodes in the DOM tree. For example, a Xerces-C node may contain as many as five pointers to other nodes, as shown in Figure 1.2.

These pointers occupy 160 bits, assuming 32-bit pointers. However, an information theoretic argument shows that a tree with n nodes can be represented using just under $2n$ bits. It is not at first sight clear how to represent a DOM tree so compactly while still performing navigation efficiently. The idea in this thesis is to apply the theory of *succinct data structures*. Succinct data structures pioneered by Jacobson [44] show how to represent data using close to the minimum possible space, while performing operations quickly.

1.4 Contributions and Organisation of Thesis

We present a DOM implementation called *Succinct DOM* (*SDOM*) based on succinct data structures. In detail, the contributions of this thesis are as follows:

(a) We study the Xerces-C DOM implementation and determine its space usage costs, and also that of the `TinyTree` DOM implementation in Saxon.

(b) We study several succinct data structures, and give some implementation details that have not previously been published.

(c) We advance the knowledge of succinct data structures in that we have created a strong correlation between succinct tree representations and XML document trees. The succinct tree representations now efficiently support the DOM operations upon the tree.

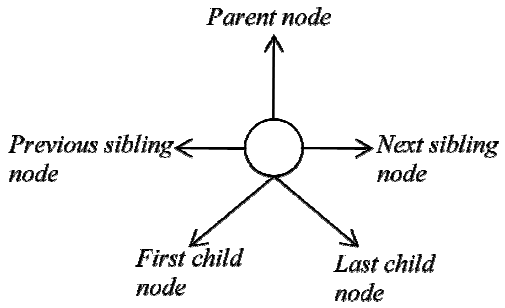


Figure 1.2 – Node representation of a DOM implementation. Arrows represent pointers.

The results show an improved running time and a much reduced space usage to existing succinct tree representations. We detail the introduction of the idea of a partitioned representation of bit-strings to represent tree structures. The experimental evaluation proved our predictions that a partitioned representation of a succinct tree representation does not increase the space usage of a succinct tree (with one exception), and gives improved running times.

(d) We advance the knowledge of representing the textual data in XML documents by investigating the efficient representation and access of the individual textual data. Where existing solutions focus on the textual data compression, we show the importance of compressing the pointers to the individual textual data themselves, which would generally be expensive.

(e) We present a DOM implementation call Succinct DOM (SDOM) that uses succinct and other data structures to represent XML documents. We provide experimental evaluation of SDOM against other DOM implementations and a compressed variant called SDOM-CT against XML compressors. The space usage of SDOM on average was 0.5 times the original file size, whereas the space usage of the DOM implementation, i.e. Xerces on average was 5.7 times larger than the original file size. The space usage of SDOM-CT showed further improvements, which is competitive to the query-friendly compressors.

The rest of the thesis is organised as follows: Chapter 2 gives background details of XML and of the DOM specification. Chapter 3 gives an overview of Xerces-C, details of Saxon's in-memory tree representation, a survey of related XML compressors and

statistics of XML files used in this thesis. In Chapter 4, we define the key algorithmic idea of succinctness, applied to the data structures we use. Chapter 5 gives an experimental study of the succinct tree representations, where we have engineered them further with XML specific requirements. Chapter 6 gives a study of representing the textual data of XML documents efficiently. Chapter 7 presents the main contribution of this thesis with the details of the SDOM implementation, interfaces for other applications and the experimental results of SDOM. Finally, in Chapter 8 we give the closing remarks of the thesis achievements, contributions, and outline future development of SDOM.

Chapter 2
XML Background

Firstly, this chapter introduces basic background knowledge on XML and secondly, details of the DOM specification.

2.1 XML

2.1.1 Markup and Text

An XML document is a text file that is made up of *data values* and *markup*. The data itself is just text. The markup is the description and structure of the data. Markup is composed of *tags*, which consist of a label (characters) inside the symbols '<' and '>'; an example of a tag is `<book>`.

Tags generally appear in pairs, comprising a *starting tag* and *ending tag*. A tag help to distinguish a piece of text from any other piece of text, and often provides information about, or give meaning to the text it contains. For example, in the following element "`<Year>2007</Year>`" we know that the content is probably a numeric value for the year. Elements may contain other elements providing they are properly *nested*; this is to say elements cannot stand alone (unless there is only one element, the root element), and they must be contained within a hierarchy of elements that begins with the root element.

2.1.2 Well-formed and valid XML documents

We categorize the correctness of XML documents into two levels:

- *Well-formed* documents, which obey the necessary and sufficient syntactic condition (defined in the XML specification [79]). The documents contain text and XML tags, which are nested properly (meaning opening and closing tags must match and tag pairs must be contained within outer tags) and data values must appear within an enclosing tag. A *document type definition* (DTD), which we define below, is not compulsory.
- *Valid* documents, which conforms to the above XML syntax and are error checked against a set of rules defined in a DTD or XML Schema, which are

associated with the file. The DTD describes the format of an XML document's markup, such as the tags allowed, what values those tags may contain, definition of entities or attributes allowed, and how the tags relate to each other.

2.1.3 Components of an XML document

An XML document consists of the following components:

(a) *Document prolog* – Is an optional component at the start of the XML document that consists of two parts: the XML declaration, i.e. `<?xml version="1.0">`, and the DTD. Miscellaneous statements may also exist, such as comments or processing instructions.

(b) *Document instance* – This follows the prolog in the document layout and is the main part of the XML document, containing the content of the document. The term instance means (as in object-oriented programming) that the document is an instance of the DTD or an unspecified class if the DTD is not given. The document instance must contain a root element that encloses all other nested elements and data values. We discuss below the subcomponents of the document instance in more detail.

(c) Optionally, processing instructions may appear in the prolog and/or in the document instance.

The document instance includes some or all of the following subcomponents:

- *Elements* – This is a pair of tags, enclosing pairs of tags and/or some simple text, or a single tag with a forward slash at the end (i.e. `<break/>`). The elements are named using an *XML name*. An XML name must begin with a letter, underscore, or a colon. They can contain letters, digits, periods, hyphens, underscores and colon.
- *Attributes* - Elements may contain some named *attributes* associated with them that describe certain properties of the element. They consist of an attribute value pair – the name of the attribute (which is an XML name), then an equal sign, followed by the attribute value enclosed in double (or single)

quotes (an attribute must have a value). For example, the attribute `catalogue` with value `XML` appears within an element as

`<book catalogue="XML"> ... </book>`.

- *Comments* – These appear in either the prolog or the body of the XML document. XML comments are like HTML comments; they can be used for explanatory notes, which are sometimes ignored by applications. They appear in the form `<!-- comment-->`.
- *Entity References* – An entity is understood as a named body of data, usually text. They are often used to represent single characters that cannot be entered on the keyboard. An *entity reference* is a placeholder that represents the entity. Entity references appear in the form of a name, which is preceded by an ampersand (&) and followed by a semicolon (;). There are five predefined entities in XML:
  - `&` ('&' or ampersand)
  - `<` (< or less than)
  - `>` (> or greater than)
  - `&apos;` (apostrophe)
  - `&quot;` (quotation mark)
- *CDATA section* – This markup contains character data with no restrictions of the characters used, and is in the form `<![CDATA[`*`content`*`]]>`. A CDATA section is ideal for inserting arbitrary text e.g. programming code. All characters enclosed in the CDATA section are interpreted as characters, not markup or entity references. A CDATA section may look like:

```
<![CDATA[
for(int i=0; i<=10;i++)
  sum+=i;
]]>
```

- *Processing Instruction* – These appear in either the prolog or the body of the XML document. A processing instruction allows documents to contain instructions for applications, e.g., style-sheets. A processing instruction consists of the string `<?` followed by an XML name, optional white spaces, followed by a list of name-value pairs (similar to an attribute, but the name need not be an XML name), the name is the *target* and the value is the *data*. Finally the string `?>` closes the processing instruction. The XML declaration is not a processing instruction. Example of a processing instruction is a style-sheet declaration connected to the document:

  ```
  <?xml-stylesheet href= “headlines.css” type=“text/css” ?>
  ```

2.1.4 Advanced features of XML documents

Namespaces

Namespaces provide a way to identify unique elements and attributes with the same XML name, but different meaning in the same or different XML documents. For example, if we build an XML document of the courses taught in an educational institution, we may use the tag `module`, to represent courses taught. We would like to integrate this XML document with a document for the Computer Science department that already uses the element name `module` to describe a component of a system development. Using the `module` tag in a combined document causes the problem of ambiguity in the meaning when the tag is used. To avoid this conflict of names *namespaces* are used.

A *namespace* is defined by an attribute with the name `xmlns` in the start element of a tag. When declaring the namespace the syntax is as follows `xmlns:`*`pre`* `=`*`‘URI’`*. The URI uniquely identifies the namespace. The string `pre` is used to prefix any tag name that belongs to the namespace denoted by the URI within the scope of its declaration, thus helping to distinguish between two tags with identical XML names but different meanings. In the example above, the conflicts can be resolved as follows:

```
<university xmlns:de= “http://www.cs.le.ac.uk/department”
xmlns:cs= “http://www.cs.le.ac.uk/systems” >
     ...
     <department name = “computer science”>
          ...
          <de:module = “algorithms” >....</de:module>
          ...
          <cs:module = “SDOM” >....</cs:module>
     </department>
<university>
```

XML tree

The tags and the element content in the document instance form a hierarchical structure, which is logically viewed as an XML tree. We label the nodes with the element names and the data values are stored at the leaves in the tree. For example, see Figure 2.1.

The order of element nodes in the XML tree (in pre order) matches the order of the elements in the document, reading from top to bottom.

```
<bookshop>

  <book catalogue='XML'>
    <author>OND</author>
    <title>SDOM Design</title>
    <year>2007</year>
    <note>Development in
      <code>C++</code>
    </note>
  </book>

  <book catalogue='XML'>
    <author>
      <firstname>J</firstname>
      <surname>Andrews</surname>
    </author>
    <title>DOM processing</title>
    <year>2006</year>
  </book>

</bookshop>
```

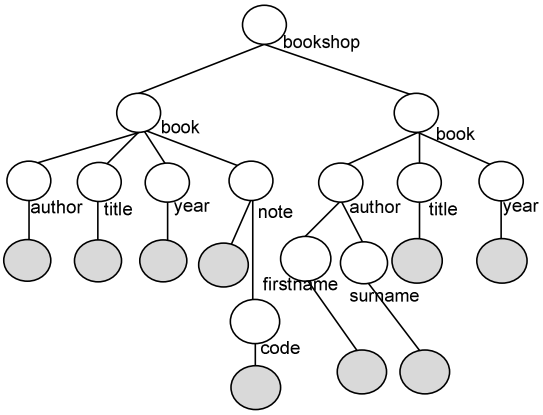


Figure 2.1 – XML tree of the bookshop document. Data values shaded in grey.

2.2 XML Parsing and Processing

Parser: A *parser* is a program that receives input in the form of the characters of a file, which is then analysed against a grammar for that language to check its validity. The parser will often create tokens from the sequence of input characters.

XML Parser: The sequence of outputs received from an XML parser is the markup tags, character data and other data of the XML document. The parser will separate the XML document components (i.e. elements and character data), which is the output to be handled by other programs. An XML parser applies the validity, well-formedness and semantic rules that are given in the DTD or the Schema of the document.

To read and manipulate XML documents one can use the *event-based* parser (i.e. *SAX*) only or use SAX to read the XML document and convert it into an XML DOM object in memory.

The Simple API for XML (SAX), a 'de facto' standard, is an event-driven push model for processing XML. As SAX reads the XML documents in a "stream" manner, it triggers off a series of events. Event handlers must be written to process the data retrieved from these events.

SAX maintains minimal information about the XML document at any one point while parsing, therefore resulting in low memory consumption. Using SAX, we therefore can parse documents that are much larger than the system memory. The disadvantage with SAX is that the document content or its hierarchy is not maintained, during or after the parsing phase, therefore, the content must be handled by an external application. In essence, repeated processing cannot be achieved using SAX without repeated parsing of the document.

DOM implementations represent the XML document as a tree structure in main memory. The tree is constructed using a SAX parser. The tree can be navigated efficiently, but existing DOM implementations exacerbate XML bloat.

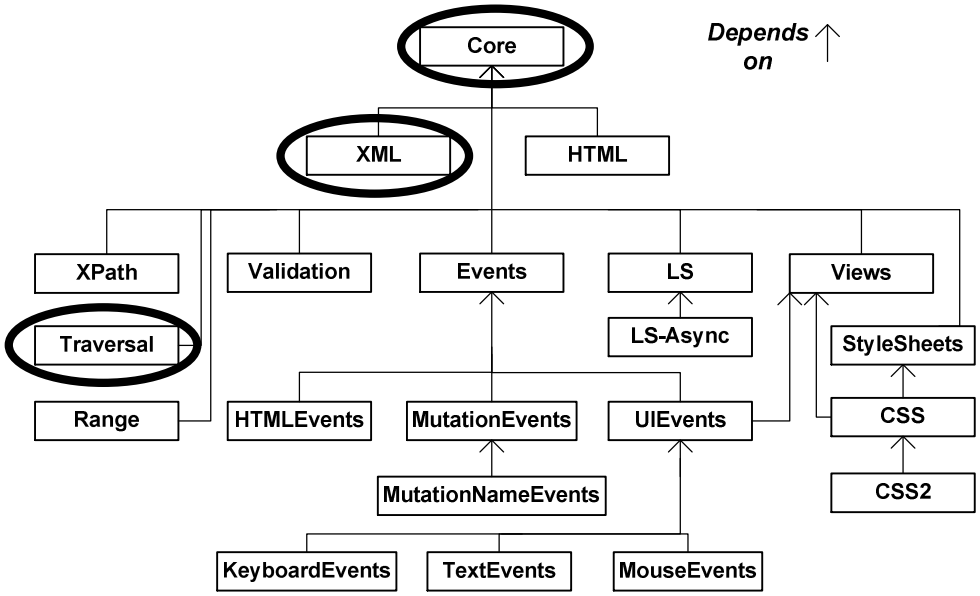


Figure 2.2 – DOM modules defined in the DOM specification [77].

2.3 DOM Architecture and Standards

The DOM [77] is a set of application programming interfaces (APIs) that defines the logical structure, access and manipulation of XML and HTML documents. The DOM APIs are organised into groups that address the same features; these groups are called *modules* and are given a name according to the feature they support. In addition, all APIs are categorised into *levels*, each providing its own operations for the APIs. The levels describe the functionality a user can expect from an application that supports the module(s). In Figure 2.2, we show the hierarchical structure of the modules, where the arrows show the dependences. For each module, level $(i + 1)$ includes the functionality at level i. Some modules begin at level 2 or higher. We describe the modules according to the DOM levels:

- Level 1 – The DOM APIs at this level are divided into two modules, the `Core` and `HTML`. The `HTML` module provides higher-level APIs that are used along with those in the `Core` module for working with HTML documents. The `Core` also contains inherited APIs, which are grouped into what is called the `XML` module.

- Level 2 – The DOM APIs at this level are divided into 14 modules.
- Level 3 – Adds further features for the `XML` module such as abstract schemas, and has six new modules such as `XPath`, `Load`/`Save`, `Validation`, etc.

We show in Figure 2.2 all the modules in DOM as a hierarchical structure: the circled modules in the figure are those we are interested in supporting. The details of these are as follows:

(a) `Core` Module: Contains the fundamental core APIs that should be in all DOM implementations to maintain their conformance to the DOM specification. This module must be supported for others to exist. The APIs contained are as follows: `Node`, `Element`, `Attr`, `CharacterData` (which has the derived APIs `Text` and `Comment`), `DOMImplementation`, `DocumentFragment` and two helper APIs, `NodeList` and `NamedNodeMap`. The `Node` is the base API, which contains functionality common to all nodes, the other APIs inherit their methods and properties from the `Node` API. In level 2, the module is updated with the XML namespace support and further features for the `XML` module.

(b) `XML` Module: Contains the following APIs: `CDATASection`, `DocumentType`, `Notation`, `Entity`, `EntityReference` and `ProcessingInstruction`. This module is an extension of the `Core` module. It deals with XML-specific node types.

(c) `Traversal` Module: Contains the following APIs: `NodeIterator`, `NodeFilter`, `TreeWalker` and `DocumentTraversal`. The first three provide node traversal functionality over a document's nodes. The `DocumentTraversal` provides operations to create instances of `TreeWalker` and `NodeIterator`.

The `Node` interface consists of a number of variables, such as `nodeName`, `nodeValue`, and `attributes`. In addition, the `Node` interface consists of navigation operations upon the DOM tree, which are as follows: `firstChild`, `nextSibling`, `previousSibling`, `parent` and `lastChild`.

In Level 2, the `Core` module has methods to access the node's namespace URI and prefix. Other methods exist such as for comparing document position, node identity check, text content retrieval of all sub-tree nodes and namespace and prefix lookup (within the sub-tree, including the current node).

2.3.1 DOM Node Types

The DOM tree consists of node objects representing the XML document. Each node has a type, which corresponds to the XML component present in the XML document (details of the XML components in Section 2.1.3). The DOM tree begins with a `document` node, which provides a central point to access the entire document. The `document` node can have several child nodes, but must have a single root node, we call the *root* `element` node. The root node corresponds to the root element in the XML document. Other node types exist in DOM, which are given the same name as the XML document components in Section 2.1.3. The type of a node is stored in the variable `nodeType`, there are twelve possible values, depending on the type of a node the Node API variables will differ. The node types are as follows:

- `Element` node: this node type represents the elements within the XML document. Access to a node of this type is provided through the interface `Node` and `Element`.
- `Attribute` node: this node type represents an attribute of an element node. Attributes are not part of the DOM tree, but are accessed through the `NamedNodeMap` interface, which is a variable in the Node interface.
- `Text` node: this node type represents the 'free' textual content of an element. They appear only as leaf nodes in the DOM tree. Access to a node of this type is provided through the interface `Node`, `CharacterData` and `Text`.
- `CDATASection` node: this node type represents the CDATA section in XML documents. The textual body is the data value. Access to a node of this type is provided through the interface `Node`, however the DOM level 2 includes the `CDATASection` interface, which provides direct support.

- `EntityReference` node: this node type represents an entity reference in the XML document. XML allows the user to define their entities in the DTD. Access to a node of this type is provided through the interface `Node`; however, the `XML` module, Level 2 includes the `EntityReference` interface, which provides direct support.
- `Entity` node: this node type represents an entity in an XML document. Access to a node of this type is provided through the interface `Node`, however Level 2 includes the `Entity` interface, which provides direct support. An `Entity` node may be of the following types:
    - *Internal entity*: the definition for this type of entity is within the document's DTD. There are five internal entities predefined in XML, these are special codes to represent the following characters: ampersand, less-than, greater-than, double quote and single quote.
    - *External entity*, which allows the user to integrate entity definitions from other documents.
    - *Parameter entity***,** which can be internal or external entity references and are not expanded in the DTD or the internal subset (main document body).
- `ProcessingInstruction` node: this node type represents a processing instruction in the XML document. They appear as a leaf in the DOM tree.
- `Comment` node: this node type represents a comment in the XML document. The `comment` node is a leaf node in the DOM tree.
- `Document` node: the DOM tree has a single `Document` node. It appears at the root of the tree. The document node object supports the creation of node objects and access to the entire DOM tree.

- `DocumentType` node: this node type represents the DTD in the XML document. This node appears as a child of the `document` node. Only one instance of the `DocumentType` node can exist in a DOM tree.
- `DocumentFragment` node: this node represents a sub-tree inserted to the DOM tree. This node type is used in dynamic implementations of DOM.
- `Notation` node: this node type represents a notation in an XML document. Notations have no parent nodes. They are defined in the DTD.

Table 2.1 shows a summary of the node types in the DOM and shows the corresponding variable details `nodeName` and `nodeValue` for a given node type. In addition, we show for the node types the children allowed, if they have any. Each node type is associated with a special ID number.

2.3.2 Traversal Module

The two main orders of traversal are:

- *Document order*: We navigate the DOM tree from the root node through all first child nodes. Then navigate the right sibling nodes if we are at a leaf node or if we have visited the current node's sub-tree already. This process we repeat until we have reached the right-most leaf node in the tree.
- *Reverse document order*: We navigate the DOM tree from the right-most leaf node through all previous nodes in the tree until we reach the root node. The process is to navigate to the right-most leaf of the sub-tree of each node, then repeat the process on the left sibling of each node, if the node is a leaf or visited already. If there are no more left siblings of the current node, we navigate to the node's parent, then repeat the process with the node's left sibling or at an ancestor node if it has no left sibling node.

Table 2.1 – Summary of the DOM `Node` types. Asterisk (*) indicates maximum of one child node allowed for that node type.

Node No.	Node Type	NodeName	NodeValue	Children Allowed (node no.)
1	`Element`	`Tagname`	`NULL`	1, 3, 4, 5, 7, 8
2	`Attribute`	`Attribute name`	Attribute value	3, 5
3	`Text`	*`"#text"`*	Data value	None
4	`CDataSection`	*`"#CDataSection"`*	Data value	None
5	`EntityReference`	`Entity name referenced`	`NULL`	1, 3, 4, 5, 7, 8
6	`Entity`	`Entity name`	`NULL`	1, 3, 4, 5, 7, 8
7	`ProcessingInstruction`	`Target`	Data value	None
8	`Comment`	*`"#comment"`*	Data value	None
9	`Document`	*`"#document"`*	`NULL`	1*, 7, 8, 10*
10	`DocumentType`	`DocType Name`	`NULL`	None
11	`DocumentFragment`	*`"#documentFragment"`*	`NULL`	1, 3, 4, 5, 7, 8
12	`Notation`	`Notation name`	`NULL`	None

The orders of traversal are applied to the `TreeWalker` or `NodeIterator` interfaces. Details of these are as follows:

- The `NodeIterator` logically views the XML document in a "flat" manner, like an array of nodes that appears in document order. Moving forward in this array (given by the operation `nextNode()`) and backward (given by the operation `previousNode()`) represents document-order and reverse document order traversal, respectively.
- The `TreeWalker` maintains the tree (or sub-tree) structure of the document. The operations of `TreeWalker` are the tree navigations similar to those in the `Node` API, in addition we have `nextNode()`, `PreviousNode()`and `getCurrentNode()`. A call of any navigation operation returns a node to the user, and updates the iterator of the current node held within `TreeWalker`, providing that the node returned is not null.

`TreeWalker` and `NodeIterator` both support the operations `getRoot()`, `getWhatToShow()`, `getFilter()` and `getExpandEntityReferences()`. The last

two operations relate to the `NodeFilter` API, which allows the user to create an object that filters out nodes. The user calls a `NodeFilter`, which is applied to a node in any traversal to determine whether or not the node should be presented in the traversal's logical document. The user can select from thirteen different constant filters, which describes what to show. We only list a few of these because (as their names suggest) they are based upon the DOM node types: SHOW_ALL, SHOW_ELEMENT, SHOW_ATTRIBUTE, SHOW_TEXT, SHOW_CDATA_SECTION, etc.

The documentation of the DOM specification can be found at [77] and [78]. Appendix B details these APIs and their methods and indicates when methods are supported by the SDOM application that we will discuss in Chapter 7.

Chapter 3
Implementations of DOM and XML Compressors

In this chapter, we examine in-memory representations of XML documents that implement the DOM interface. We also include a survey of some XML compressors, some of which are designed to support the DOM `Core` and `XML` modules, whereas others support some navigation resembling the DOM.

In Section 3.1, we focus on the Xerces-C DOM implementation and Saxon's in-memory tree data structure. In Section 3.2, we discuss some of the related work on XML compressors that have in-memory and/or disk-based representations. Finally, in Section 3.3 we describe, and present statistics of, a collection consisting of real-world and synthetically generated XML files that we will use in the experimental evaluation.

3.1 DOM Implementations

The usefulness of the DOM in many applications has led to implementations of the DOM interface in almost all programming languages today, with several in Java and C++. We examine the Xerces DOM implementation, which was developed by the Apache Software Foundation [2]. Implementations are available in either Java or C++; these are called Xerces-J [68] and Xerces-C [67], respectively. Other DOM implementations exist, such as JDOM [45] and dom4j [25], both developed in Java, that implement the DOM interface, as simplified APIs that are less complex and consume less memory, than what DOM offers. We work in the C++ programming language; therefore, we focus mainly on the Xerces-C DOM implementation in our discussions. We abbreviated Xerces-C to just Xerces in the remainder of the thesis.

XQuery and XSLT processors often rely on internal DOM implementations that are optimized for good performance. For example, Saxon [61] for Java and Xalan [66] for C++ use their own interfaces as a plug-in to give access to their data structures, which can without difficulty be wrapped into a DOM node.

We now discuss some of the implementations of DOM mentioned above, beginning with Xerces, followed by Saxon's tree representation.

3.1.1 Xerces

Xerces is a validating XML parser (in version 2.8 at the time of writing of this thesis), which supports the DOM, SAX and SAX2 APIs. We focus on the DOM APIs of the Xerces implementation, which conform to the DOM Level 2 API and contains in addition, a partial implementation of the DOM Level 3 `Core`. More specifically the modules supported are `Core`, `XML`, `Traversal`, `Range` and `Load`/`Save`.

Class Structure

We now discuss Xerces' implementation of the APIs in the `Core`, `XML` and `Traversal` modules. For the `Core` module (and for the entire DOM) the primary API is the `Node` API, this is represented by the class `DOMNodeImpl`. The `DOMNodeImpl` class consists of a single pointer, which points to its parent node (for the nodes that are in the tree, but for other nodes the pointer points to some other associated node, e.g. an attribute node points to the element node where it is declared). The `DOMNodeImpl` also consists of a *special flag* (of type short) indicating certain properties of the node, e.g., a read-only or first child node. We observe that navigation (except the `getParent()` operation) and data retrieval operations are not supported in this class, but implemented by other classes in Xerces (which are derived from `DOMNodeImpl`), we will come back to this later. The other supported DOM APIs in the `Core` are given as follows:

- The `Element` API is implemented in the `DOMElementImpl` class, and the `Element` API, which defines a namespace is implemented in the class `DOMElementNSImpl`. The `Attribute` API is implemented in the `DOMAttrImpl` class, and the `Attribute` API, which defines a namespace is implemented in the class `DOMAttrNSImpl`. The `Text` API is implemented in the `DOMTextImpl` class. The other APIs relating to the node are implemented similarly.
- The `NodeList` API is implemented in the `DOMNodeListImpl` class.
- The `NamedNodeMap` API is implemented in the `DOMNamedNodeMapImpl` class. `NamedNodeMap` contains a vector of nodes (e.g. `DOMTextImpl` objects) and a

pointer to the node, which owns the map. The `DOMAttrMapImpl` is a derived class of `NamedNodeMap`, which provides specific support for attribute nodes, e.g. includes a boolean value indicating default or fully declared attributes.

In Table 3.1, we show the class structure of Xerces, which implements the DOM APIs, such as the APIs in the `XML` module; for example, the `CDataSection` API is implemented in the class `DOMCDATASectionImpl`. The Xerces classes that implement the DOM APIs have a number of pointers and internal class instances as class members, which are shown in Table 3.1. In the following, we discuss the auxiliary classes of Xerces (including `DOMNodeImpl`), which provide navigation support and the data values, depending on the node's position in the tree and its node type information:

a) `DOMNodeImpl`: We have discussed this class already, but we make special mention here because a class representing a DOM node must contain an instance of this class.
b) `DOMParentNode`: In DOM tree, a node, which can have children (see Table 2.1) must contain an instance of this class. For such classes, e.g. `DOMElementImpl`, we need to store a first-child pointer and an instance of `DOMNodeListImpl` class for the `childNode()` operation support. Node objects that cannot have children such as in `DOMTextImpl` class do not store this class instance, and thereby avoid this cost.
c) `DOMChildNode`: A node that is a part of the DOM tree must contain this class instance. This class has pointers to previous-sibling and next-sibling nodes, if they exist, or a null value if any of the siblings do not exist. The `DOMAttributeImpl` is not part of the DOM tree, and therefore it does not contain this class instance.
d) `DOMCharacterData`: A node that stores a data value must contain this class instance. For example, the `DOMTextImpl` and `DOMAttrImpl` have a value, therefore, must contain this class instance. However, the `DOMElementImpl` does not have a data value, therefore, does not contain this class instance.

Table 3.1 – Xerces internal classes, with their class members and memory usage details.

	CLASS MEMBERS			MEMORY USAGE OF CLASS
NODETYPE CLASS DEFINITION	CLASS INSTANCES	POINTERS	VARIABLES	
DOMDOCUMENTIMPL	DOMNODEIMPL, DOMPARENTNODE, DOMNODEIDMAP, VIRTUAL CLASSES ×2	elementID, fActualEncoding, fEncoding, fVersion, fDocumentURI, fDOMConfiguration, fUserDataTable, fRecycleNodePtr, fRecycleBufferPtr, fNodeListPool, fCurrentBlock, fFreePtr, fDocType, fDocElement, fNamePool, fNormalizer, fRanges, fNodeIterators, fMemoryManager	fChanges:int, errorChecking:bool, fStandalone:bool, fFreeBytesRemaining: XMLSize_t	136 BYTES
DOMELEMENTIMPL	DOMNODEIMPL, DOMPARENTNODE, DOMCHILDNODE VIRTUAL CLASS	FATTRIBUTES, FDEFAULTATTRIBUTES, FNAME, FSCHEMATYPE		52 BYTES
DOMELEMENTNSIMPL	VIRTUAL CLASS	FNAMESPACEURI, FLOCALNAME, FPREFIX		68 BYTES
DOMTEXTIMPL	DOMNODEIMPL, DOMCHILDNODE, DOMCHARACTERIMPL, VIRTUAL CLASS			28 BYTES
DOMATTRIBUTEIMPL	DOMNODEIMPL, DOMPARENTIMPL, VIRTUAL CLASS	NAME, FSCHEMATYPE		36 BYTES
DOMCOMMENTIMPL	DOMNODEIMPL, DOMCHILDNODE, DOMCHARACTERDATAIMPL, VIRTUAL CLASS			28 BYTES
DOMENTITYREFIMPL	DOMNODEIMPL, DOMPARENTNODE, DOMCHILDNODE, VIRTUAL CLASS	FNAME, FBASEURI		44 BYTES
DOMCDATASECTIONIMPL	DOMNODEIMPL, DOMPARENTNODE, DOMCHILDNODE, DOMCHARACTERIMPL, VIRTUAL CLASS			44 BYTES
DOMPROCINSTRIMPL	DOMNODEIMPL, DOMCHILDNODE, DOMCHARACTERIMPL, VIRTUAL CLASS	FTARGET, FBASEURI		36 BYTES
DOMENTITYIMPL	DOMNODEIMPL, DOMPARENTNODE, VIRTUAL CLASS	FNAME, FPUBLICID, FSYSTEMID, FNOTATIONNAME FACTUALENCODING, FENCODING, FVERSION, FBASEURI, FREFENTITY	FENTITYREFNODECLONED: BOOL	68 BYTES
DOMDOCUMENTTYPEIMPL	DOMNODEIMPL, DOMPARENTNODE, DOMCHILDNODE, VIRTUAL CLASS	FNAME, FPUBLICID, FSYSTEMID, FINTERNALSUBSET, FENTITIES, FNOTATIONS, ELEMENTS	FINTSUBSETREADING: BOOL, FISCREATEDFROMHEAP:BOOL	68 BYTES

Table 3.2 – Xerces auxiliary classes that appear as class members in Table 3.1. We give the class members and space usage.

	CLASS ATTRIBUTES			MEMORY USAGE OF CLASS
DOM TREE CLASS	**DOM CLASS**	**POINTERS**	**VARIABLES**	
DOMNODEIMPL		OWNER	FLAG:SHORT	8 BYTES
DOMPARENTNODE		OWNERDOCUMENT, FIRSTCHILD, DOMNODELISTIMPL		16 BYTES
DOMNODELISTIMPL	VIRTUAL CLASS	FNODE – ARRAY		8 BYTES
DOMCHILDNODE		PREVIOUSSIBLING, NEXTSIBLING		8 BYTES
DOMCHARACTERDATAIMPL		FDATABUF, FDOC		8 BYTES
*DOMNAMEDNODEMAPIMPL	VIRTUAL CLASS	FNODE, FOWNERNODE		12 BYTES
*DOMATTRMAPIMPL	VIRTUAL CLASS		ATTRDEFAULTS:BOOL	12 BYTES
*DOMBUFFER		FBUFFER, FDOC	FINDEX:INT, FCAPACITY:INT	16 BYTES
*DOMTYPE	VIRTUAL CLASS	NAME, NAMESPACEURI		12BYTES
*DOMNODEIDMAP		**FTABLE, FSIZEINDEX, FDOC	FSIZEINDEX, FSIZE, FNUMENTRIES, FENTRIES	
*DOMCONFIGURATION	VIRTUAL CLASS	FERRORHANDLER, FSCHEMATYPE, FSCHEMALOCATION, FMEMORYMANAGER		20 BYTES
DOMSTRINGPOOL		FDOC, **FHASHTABLE,	FHASHTABLESIZE:INT	
DOMNODEVECTOR		**DATA	ALLOCATEDSIZE:XMLSIZE_T, NEXTFREESLOT: :XMLSIZE_T	

As an example of a node in Xerces, the `DOMElementImpl` class represents an element node in the tree. This class contains instances of the classes `DOMNodeImpl`, `DOMParentNode` and the `DOMChildNode`. In addition, schema type information is stored. A C++ string is stored, which represents the element name information of the node and two pointers to the class `DOMAttrMapImpl` representing attributes and default attributes declared at this element node. A default attribute is given a value by the DTD at the parsing phase if that attribute's value is omitted in the document.

The `DOMElementImpl` class requires 52 bytes. This includes four bytes as overhead because it is derived from the virtual DOM `Element` class. The remaining space comprises (assuming pointers are four bytes) eight bytes for the `DOMNodeImpl` instance, sixteen bytes for the `DOMParentNode` instance, eight bytes for the `DOMChildNode` instance and four further pointers.

We observe that the space usage of other nodes in Xerces can be much less than for an element node, excluding the space for the data value. For example, the `DOMTextImpl` class that represents a text node in the tree only requires 28 bytes. In Table 3.1, we show the classes contained in the `Core` and `XML` modules. We give the space usage of each class. The cost of using a virtual class is included in the space usage costs. Table 3.2 shows the class members of the auxiliary classes in the Xerces implementation of the DOM APIs. We include the space usage of each class which was obtained using the *sizeof* function.

Navigation operations

Navigation in Xerces is very fast. The navigation operations such as `firstChild()`, `parent()`, `nextSibling()` and `previousSibling()` just return a pointer value.

For all nodes in the tree, the last child node appears as the previous sibling pointer of the first child node. Therefore, in the `lastChild()` operation we first dereference the first child node pointer then return its previous sibling pointer. This avoids traversal of the next sibling nodes to get to the last child node. However, this potentially causes the following problem: if we are at the node that is the first child in the tree, a call of the `previousSibling()` operation would return a pointer to its parent's last child node, which is incorrect. Xerces avoids this problem by using a flag (in `DOMNodeImpl`) to indicate in the node instance whether this node is a first child.

NodeType

Node types in Xerces are not explicitly stored. They are represented through the classes representing the node, for example, an `element` node is represented by `DOMElementImpl` class object in memory. Therefore, for the `getNodeType()` operation, the node type of a node is known in the class, and not explicitly stored.

Node name and textual data

Xerces stores node names in two types of classes, which are as follows:

- Classes with namespace support: Contain three pointers to C++ strings, representing the element name. The first pointer points to the namespace URI,

the second to its prefix and the third to its local name. Element nodes with the same node name (including namespaces) point to the same strings.

- Classes without namespace support: Contain a single pointer to a C++ string to represent element name (or a local name). Nodes with the same name point to the same string.

Xerces stores a data value as a pointer to a C++ string (here, nodes with the same text do not point to the same string).

3.1.2 Saxon's TinyTree

Saxon has its own internal tree structure, called `TinyTree` [61]. Besides reducing memory usage, the objective is to minimize the costs of allocating and garbage-collecting Java objects used. The `TinyTree` class contains a collection of arrays to represent the content of one or more XML documents. The arrays in `TinyTree` give a flat view of the tree structure in document order. Table 3.3 shows the arrays in the `TinyTree` class.

The arrays, which are of length n, represent an XML document tree with n nodes. The ith entry in the array stores information relating to the ith node in document order. The following information is maintained for each node: its node type is maintained in the `nodeKind` array, the depth of a node is maintained in the `depth` array, and the tag names (represented by special namecodes) are maintained in the `namecode` array (we discuss namecodes later). A node's next and previous sibling nodes are maintained in the `next` and `prior` arrays, respectively. The `alpha` and `beta` arrays hold different kind of data depending on the type of the node, in the `alpha` array, if the ith node is of type:

- `Text` node, then `alpha[`i`]` contains an offset into the *text buffer* (which contains all the text data of the document, its implementation is described below).
- `Comment` or `processingInstruction` node, then `alpha[`i`]` contains an offset in the *comment buffer* (similar to the text buffer).

- `Element` node, then `alpha[`i`]` contains an index to its first attribute node or -1 if the element has no attribute.

In the `beta` array if the ith node is of type:

- `Text` node, then `beta[`i`]` contains the length of the data value held in the text buffer,
- `Comment` or `processingInstruction` node, then `beta[`i`]` contains the length of the data value held in the comment buffer.
- `Element` node, then `beta[`i`]` contains an index of its first namespace node in the `namespaceCode` array or -1 if the element has no namespace nodes.

The `text` node values are stored in a class called the `LargeStringBuffer`, which is an implementation of the Java `CharSequence` interface. The `LargeStringBuffer` contains the length of the buffer, a Java `List` array called `segments`, which contains a number of `FastStringBuffer` class instances (a `FastStringBuffer` is a character array). Initially the individual textual data values are concatenated into a single string, which is then split into equal size blocks. Each block is held in a `FastStringBuffer` and stored in the `segments` array. The `LargeStringBuffer` also contains an integer array of offsets used to give the position at the start of the block relating to the individual text data. The `LargeStringBuffer` supports the `substring(i,j)` operation which returns the substring from position i to j in the string buffer. The textual data of `comment` and `processingInstruction` nodes are stored in a `FastStringBuffer`.

The attributes are represented in a collection of arrays of length a, where a is the total number of attributes. The `attParent` array maintains information relating the ith attribute to its parent `element` node, and the attribute names are represented by namecodes maintained in the `attCode` array (analogous to the `namecode` array). The attribute values are maintained in a `CharSequence` class.

The namespace information is maintained in arrays of length s, where s is the total number of namespace declarations. The arrays contain the namespace code (these are

the prefix and URI components of the namecode) and an index of the `element` node's first namespace in the array.

We now discuss the use of namecodes in `TinyTree`. These represent, as 32-bit values, the fully qualified names of nodes in such a way that the namespace prefix, namespace URI and local name can easily be retrieved. In the parsing phase, namecodes are built as follows: we have a hash table of all unique tag names as `<localname, URI>` pairs in the document. These are stored in a chained hash table, called the `NamePool`, with 2^{10} buckets, where each bucket is (effectively) limited to hold lists of length 2^{10}. A `<localname, URI>` pair is specified by a 10-bit hash code (specifying the bucket) and a 10-bit offset into the list in that bucket. A further 10 bits are used to encode the namespace prefix.

`TinyTree` does not create all of the above arrays at the outset, but only creates them if the user invokes an operation that needs that array. Therefore, depending on the actual sequence of operations, which are invoked or used by the user, the space usage of `TinyTree` is between 20-30 bytes per node. In Table 3.3, we show the data structures of `TinyTree`, given that we know the count of nodes of an XML document we can easily calculate and confirm the above, space usage per node. Fast construction of the tree representation is crucial; [46] states that the construction can take as long as a subsequent query or transformation.

Saxon's Class Structure

Saxon provides access to the `TinyTree` data structure using classes that must implement the `DocumentInfo` and `NodeInfo` class interfaces, and which are used to interface with other components in Saxon. The class structure of Saxon is similar to Xerces, in that the `TinyNodeImpl` class, which implements the `NodeInfo` interface, in essence represents a DOM `Node`. `TinyNodeImpl` class consists of the `TinyTree` class object, the node number and a `TinyNodeImpl` class instance of its parent node. The `TinyDocumentImpl` implements the `DocumentInfo`, which represents the `document` node.

Table 3.3 – `TinyTree` class members.

TYPE	NAME	SIZE	MEMORY	DETAILS
`arrayList`	`documentList`	5		Default list of documents in
`largeStringBuffer`	`charBuffer`	m	$8m$	m is the total length of text node
`fastStringBuffer`	`commentBuffer`	k	$8k$	k is the total length of comment
`byte`	`nodeKind`	n	$8n$	Node type
`short`	`depth`	n	$16n$	Depth of node in tree
`int`	`Next`	n	$32n$	Node number of next sibling
`int`	`alpha`	n	$32n$	Value depending on node type
`int`	`beta`	n	$32n$	Value depending on node type
`int`	`Namecode`	n	$32n$	Holds the name of the node as a
`int`	`prior`	n	$32n$	Node number of previous sibling
`int`	`typeCodeArray`	n	$32n$	Typecode array for elements if
`int`	`attParent`	a	$32a$	Index of the parent element node
`int`	`attCode`	a	$32a$	Namecode representing the attribute
`charSequence`	`attValue`	a	$8la$	l is the #chars. String value of the
`int`	`attType`	a	$32a$	Type annotations. Created if needed
`int`	`namespaceParent`	s	$32s$	Index of element owning namespace
`int`	`namespaceCode`	s	$32s$	Namespace code used by
`int`	`rootIndex`	8	256	Array holding level 0 root nodes in
`lineNumberMap`	`lineNumberMap`	5	160	
`systemIdMap`	`systemIdMap`	5	160	Created if needed

In Chapter 7, Figure 7.5 shows the class diagram of `TinyTree`. These are designed for use with XPath and XQuery operations. Saxon implements the DOM by wrapper classes of the `NodeInfo` class. The DOM support in Saxon is limited to (a) only read-only DOM operations, (b) only the `Core` and `XML` modules and (c) a separate representation of namespace declarations from the attributes.

Navigation operations

The navigational operations in the DOM, such as `firstChild()`, `nextSibling()`, `previousSibling()`and `lastChild()` are supported indirectly in the `NodeInfo` class using a set of XPath *axis* iterator classes. These classes are customized for use in XPath, but they can support (less efficiently) the DOM. The navigation operations require sequential access to one or more items of the array structures, the details of which are given below:

- The `parent()` operation is simple because an instance of the parent node object is included as a class member of the `DOMNodeImpl` class. However, if

the parent node is not known for node i, then we make a sequence of calls to the `next` array, from `next[`i`]` until to get to the last-sibling node, say node j. A final call to `next[`j`]`stores the index of the parent node, since `next[`j`]`$< j$.

- The `firstChild()` operation is simple. If we are at a node i, the first-child node will appear at position $i + 1$, if `depth[`i`]` $<$ `depth[`$i + 1$`]`.
- The `nextSibling`() operation is simple. If we are at a node i, the index of the next-sibling node is given by `next[`i`]`. If `next[`i`]` $< i$, then it is really the index of the parent from the last sibling. The `previousSibling()` operation is similar, but applied to the `prior` array.
- The `lastChild()` operation is slow. We require a traversal across all children until we reach the last child node. If the count of the children of the node (the degree) is d, then there are d array accesses made in each of the `next` and `nodeKind` arrays. In addition, an instance of the `TinyNodeImpl` class is created for each sibling node, where we check for a null value; in this case, the last child node has been reached. The creation of the `TinyNodeImpl` object is based on the design of the *axis* iterator class.

Saxon supports third-party tree structures as plug-ins; they are required to implement the `DocumentInfo` and `NodeInfo` interfaces. We discuss these interfaces further in Chapter 7.

3.2 XML Compression

We now discuss specialised compressors designed for representing XML documents, which exploit the typical XML characteristics, such as a highly regular structure and the predictive values of the upward path from an element in determining the element. A path p from the root to a leaf node is defined as the sequence of tags $t_1, t_2, \dots, t_j$, where t_1 is the tag of the root node and t_j is the parent of the leaf node (data value). The upward path is the reverse, i.e. $t_j, \dots, t_2, t_1$. In essence, a path gives valuable information to the XML compressors. More generally, some XML compressors use *containers* in grouping textual data elements with similar characteristics, and applying specialized

compression algorithms to each container. Exploiting these characteristics often gain better compression than generic text compressors, such as Gzip [43], which is not able to use the path information in its compression algorithm.

Our survey of XML compressors focuses on in-memory representations, such as BPLEX [10], XBZipIndex [30], DDOM [55] and SEDOM [75], which provide support for tree navigation, with some moving towards DOM support.

We also focus on XML compressors that have a combination of in-memory and/or disk-based data structures, often they store the structure of the document separately to the data values compressed on disk. The use of disk-based data structures for XML compressors is a well-studied area that has produced many interesting results [3], [6], [8], [7], [13], [30], [32], [48], [52], [56], [64], [75] and [80]. In essence, the representations of the XML compressors usually have one of the properties below:

(a) *Local homogeneity*: The final data structure separates the tree structure from the data values. The data values are often grouped and stored into *containers*, according to their paths and tend to be of the same data type, which aids the compression. The paths give valuable information about the data that is stored. XMill [48] was the first to introduce this technique.

(b) *Homomorphic*: The final data structure preserves the document structure with the data values. XGrind [64] was the first to introduce this technique, where the advantages are the parsing/querying can now be made directly upon the compressed documents, also indexes and updates can directly be applied to the compressed document in the same way as applied to the original document. In addition, the compressed document can be checked for validity against a compressed version of the DTD.

In Figure 3.1, we show an example of an XML compressor, which uses the homomorphic property. We have a simple XML document (left) and the compressed representation (right), which is itself, in the form of an XML document with the tag names replaced by some encoding (e.g. dictionary codes) and the data values encoded using some encoder designed by the compressor.

```
<Element1>                              T1
<Element2>valueA</Element2>               T2 encode(valueA)/
<Element3>valueB</Element3>               T3 encode(valueB)/
  <Element4></Element4>                   T4/
</Element1>                             /
```

Figure 3.1 - Left: Simple XML document. Right: Example of Homomorphism.

The compression performance of the XML compressors depends upon the specific XML document that is being compressed, a standard categorisation of XML documents is described below; we use these in what follows:

- *Data-centric* documents have a regular structure of tags as their focus. For example, Figure 2.1 shows the representation of book records in a XML document of a book shop, we observe that data-centric document can be used as a database representation. The tags are usually predefined in a DTD or an XML Schema.
- *Document-centric* XML have the text as the focus. Tags only appear when needed and explicitly indicate when parts of an existing document have structure or meaning, for example, when a section of a document represents a paragraph, or a stanza of a poem.

For the survey of XML compressors, we compare (where possible) the compression ratio and compression time for each compressor. The compression ratio is defined as follows:

$$Compression\ rate = \frac{Size\ of\ compressed\ XML\ data}{Size\ of\ original\ XML\ data}$$

3.2.1 XML Compressors with DOM-like support

DDOM

The dictionary-compression based Document Object Model (DDOM) [55] is java implemented and is locally homogeneous. Two arrays represent the tree structure of the document, with the nodes arranged in document-order. Given an XML document tree with n nodes, which contains e element nodes, the arrays are of length $n + e$. Elements in the XML document are represented by two entries in the arrays, these are the positions where the opening and closing tags would appear. Other node types, such as text and attribute node, are represented by a single entry in the arrays. The first array, called TYPE, uses 8 bits per entry, and maintains the node type information. For an element node, the array entries store a special 8-bit value representing the type, and either opening or closing tag indication. For a node that is not an element, the array entry stores an 8-bit value representing its node type. The second array uses 32 bits per entry, to maintain for each element node an index value to its name in the ELEMENT dictionary (the same value is stored at the positions of the opening and closing tags). For other nodes that have a value, the array maintains the indexes of the data values into their respective dictionaries (discussed below).

`Text` nodes are maintained in dictionaries associated to their parent node (`element` or `attribute`). The index value in the above array is specific to a dictionary given by its parent node. The data values are stored as separate string objects within the dictionaries.

Navigation requires a linear pass over the arrays. In Figure 3.2, we show the DDOM data structure, with a simple XML document. We observe that the elements (e.g. `university`, `department`) are stored in the ELEMENT dictionary array and the textual data are stored in dictionaries according to their parent `element` node.

The column with the hash (#) symbol maintains the references into the dictionary arrays. We observe all textual data under the `<name>` tag are grouped into the same dictionary, for example, in Figure 3.2, even though `<name>` may appear as a descendant

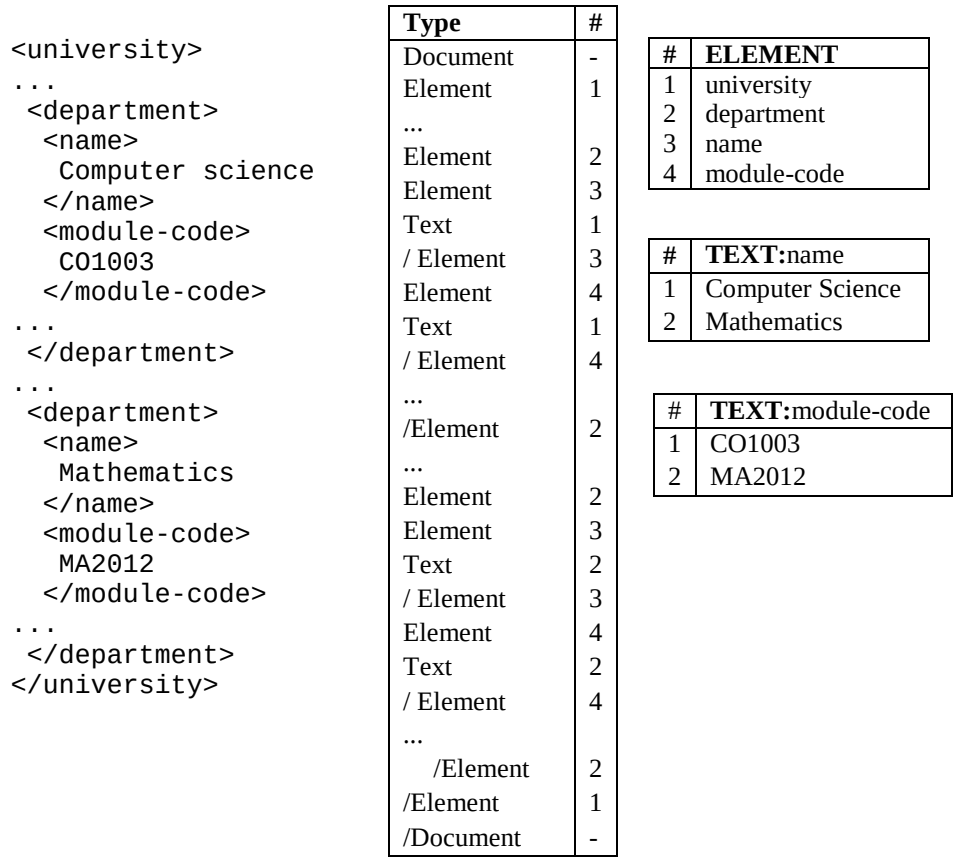


Figure 3.2 - Left: Original XML document. DDOM Centre: Structure arrays, Right: Dictionaries.

of `university` or `department`, they share the same dictionary. A limitation of DDOM is that for different tags, which have the same data value, the data values are duplicated in the different dictionaries. However, it is unusual for an XML document to have such a structure.

The implementation of DDOM supports read-only access on a document once it has been parsed or generated, however new nodes can be added at the end of an existing structure. The structure arrays require 5 bytes per node, plus an extra 5 bytes for each element node in the document. The dictionaries can be very large, especially for XML documents that have few patterns in its content. In general, document centric XML documents will have larger space usage in DDOM than data centric documents. In

addition, highly nested documents have larger space usage since the cost of representing the element nodes in the structure array is paid twice, requiring $10e$ bytes.

DDOM does not explicitly create node objects, only the `document` node and dictionaries exist initially. As in `TinyTree`, the DOM nodes are generated dynamically and contain a reference into the structure array, and are garbage collected once they are no longer needed. The internal DDOM methods work directly on the structure array. DDOM does not directly support query operations, therefore externally a XML Query Language (XQL) engine implemented in Java is required on top of the DOM.

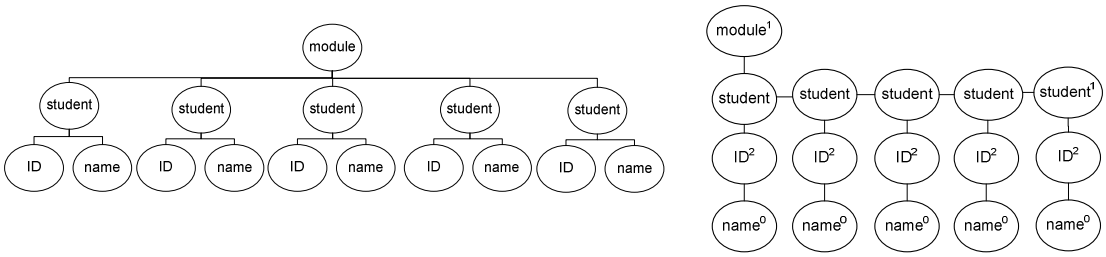
DDOM showed good compression ratios compared to Xerces-J and another DOM implementation called Crimson [21]. The compression rates of DDOM for data-centric XML files were between 20% to 60%. For document centric XML files the approach had minimal impact with compression rates between 70% to 80%. However, DDOM does not solve the problem of XML bloat, as for real life XML documents the space usage is 3 to 4 times the size of the file.

BPLEX

This XML compressor is locally homogeneous. BPLEX has a very compact pointer-based representation of the XML tree structure and represents the data values in string buffers [10]. There are two types of string buffers; the first is for the text nodes, which appear in the tree at the leaf nodes, the second is for attribute values that are associated with the element nodes. For each string buffer we store an array of integers, representing for each text (or attribute) node the location of the text (or attribute) node in the tree relative to other nodes in the buffer. Navigation and queries, such as the Core XPath are supported without full decompression.

The main result of [10] is the very compact tree structure, which we now discuss. Initially an unranked XML tree[1] is transformed into a binary tree. For example, in Figure 3.3 the tree (a) representing an XML document is transformed to the binary tree (b) with the following labels: in the binary tree a node that has no left child (a leaf in the

[1] A tree where nodes have an unbounded list of children (we discuss relationship to the binary trees in Section 4.2.3)



(a) (b)

Figure 3.3 – (a): Unranked tree of XML document. (b): Binary Tree representing the unranked tree.

XML document) is denoted by the superscript 2, and a node with no right child has subscript 1. A last child node that is a leaf is superscripted with a 0.

We observe in the binary tree that there are often repetitions of sub-tree patterns, which can be shared. If a sub-tree occurs more than once, then a pointer is used (to replace the repeated sub-tree) from the parent of the sub-tree to its first occurrence. In this way, a minimal unique directed acyclic graph (DAG) of a tree is obtained. For common XML documents, the minimal DAG is about one-tenth of the size of the binary tree [7]. We then represent the minimal DAG created from the binary tree (Figure 3.3 (b)) by the regular tree grammar that has three productions:

$M \rightarrow module^1(student(A, student(A, student(A, student(A, student^1(A))))))$,

$A \rightarrow ID^2$(B), $B \rightarrow name^0$

The BPLEX algorithm takes as input the DAG and finds a minimal context-free grammar that is half to one-third of the size of the DAG in amortized linear time. It identifies patterns in the grammar, and replaces these by non-terminals representing that pattern. In the above example, we consider the production M, which contains four such patterns of S $= student(A, y_1)$. Thus, given the production $C(y_1) \rightarrow student(A, y_1)$, each of the occurrences can be replaced by the non-terminal C. However, there is one further occurrence of a similar pattern $S' = student^1(A)$, which can be obtained by removing the parameter y_1 from the pattern S. Since A is a first child of S, removing y_1 changes $student$ into $student^1$.

Hence, if a non-terminal appears with no argument then an empty string is used as the marker. With this "overloading" semantics of productions in mind, we therefore have as output from BPLEX:

$$M \to module^1\left(C\big(D(D)\big)\right), D(y_1) \to C\big(C(y_1)\big),\ \ C(y_1) \to student(A, y_1), A \to ID^2(B),$$

$$B \to name^0$$

Construction of BPLEX is a relatively slow process. The algorithm uses a parameter for the maximum number of nodes and production that are examined (window size) in one process. The experiments of [10] show that a window size greater than 100 would give better compression results of BPLEX, but would take several hours for a file size of say 100MB. However, with default parameters (i.e. window size set to 3) BPLEX takes less than a minute. The authors of [10] mention the compression rates of the tree structure for two files in our XML corpus: `SwissProt.xml` 4.1% and `Treebank_e.xml` 34% (for a smaller version of this file). No experiments have been done for a complete XML document representation.

Results from an experimental test of BPLEX in a DOM system have not been provided, and detail of the support of navigation has not been given. However [10] states that a DOM interface can be can be supported. The suggestion is made to store the string buffers more space-efficiently using standard techniques in [3].

SEDOM

SEDOM [75] is a DOM implementation that supports retrieval, update and XPath operations on the document. A schema of the XML document is required to specify the abstract data model of the document. The data structure is locally homogeneous and consists of the following components:

(a) Name index: This stores the unique element names in two arrays. In the first array, the element names are stored as their paths from the root to the element. The schema of the document is represented as a tree structure, which is used to construct the paths; we refer to each entry in the array as an *element class*. If we

have m unique element classes, an index from 1 to m, called the token, represents an element class in (b) below. The sorting of the element classes in the array is given by the order of the paths in the schema tree. The second array of size m stores for each entry a list of indexes where the token is found in (b). The purpose is to find the name of an element class and find its locations in (b).

(b) Framework or document structure: This represents the document tree as a one-dimensional array in document-order, where each entry requires 16 bits to represent a node. The top 3 bits of the 16-bit value indicates one of six different node types, i.e. element class token, `attribute` node container block number, attribute index in the container, `text` node container block number, text node index in the container, and an end symbol for the elements. The other 13 bits are used to indicate (depending on the type of node) a token as in (a), above, or a container block number in (c), below, for `text` or `attribute` nodes.

(c) Compressed containers: Stores the data values arranged into blocks for fast retrieval and updating. Each element class has its own container, however, all attribute values share the same container. This component is stored on disk, whereas the other components are held in main memory.

(d) Container block index: This represents indexing information for each block in the compressed containers. The information stored are as follows: the token of the element class that owns the data values in the container, the block number, the offset of the block in the file, length of the block, and the node type that represents element of the first item in the block. The index information is stored in an array structure organised as a *B+ tree*. A B+ tree represents sorted data in a way that allows for efficient insertion, retrieval of records, each of which is identified by a *key* [4]. The pair consisting of the element class token and the block number are the search keys. The purpose of this index is to provide efficient access to the individual data values in the blocks, and also to locate the element class of the data value.

All navigational operations of the DOM `TreeWalker` API are supported by operating on the framework (part (b)). These are realized using an iterator approach, where the iterator points to a position (token) in the framework. The operations require scanning forwards and backwards (based upon nodes stored in document-order), we describe their running time performance as follows:

- The `firstChild()` operation is extremely fast. If we are at a node with index i, then we require the memory access of index $i + 1$.
- The `nextSibling()`operation is potentially slow. If the sub-tree of the current node is large, then we require the scanning of the nodes in this solutions tree until we reach the end symbol of the current node, the next sibling node will be at the next position in the array. The `previousSibling()` operation is implemented analogously.
- The `lastChild()` operation is slow, since it requires scanning through the array of all child nodes and their sub-tree nodes, until the end symbol for the element (parent node) is reached. Likewise the `parent()` operation requires the scanning through the array of the previous siblings nodes and their sub-trees before we get to the parent node.
- The operations `nextNode()`, `previousNode()`, and attributes retrieval are implemented similarly to the above operations.

Retrieving text values requires retrieving index information from the container block index (see (d)), given we know the desired block index we load the block from disk into a buffer in main memory and then retrieve text value. The retrieval of an element name requires a lookup in the element array (a) for the pair with matching index value from the framework, and then to get the path.

In [75] the experimental evaluation of SEDOM is given with comparison made against pointer-based DOM implementations. The main time delay in the parsing process of SEDOM is due to the compression and disk I/O of the data values. As a

result, accessing node names, node values and the modification of the tree is slower than most DOM implementations, e.g. Xerces, but comparable to XML compressors.

Results of the memory usage of SEDOM and the pointer-based DOM implementations are given only on the data structures that are held in main memory. Therefore, they exclude the textual data containers held on disk. The main memory usage of SEDOM is less than 6.9% of the main memory usage of the pointer-based DOM implementations for the XML files discussed in [75]. In addition, the data structures of SEDOM that represent the structure of the XML documents together are only a factor of 0.1-0.3 of the file sizes of the original XML documents.

We provide a very loose comparison of SEDOM to the pointer-based DOM implementations, which includes the textual data containers uncompressed. The textual values of the XML files in our corpus (see Section 3.3) were on average 46% of the XML file sizes. Therefore, SEDOM would be a factor of 0.56-0.76 of the file sizes of the original XML documents; we assume the memory usage of the structural components of SEDOM detailed in [75] would be the same for the XML documents in our XML corpus. In essence, the memory usage of SEDOM including textual data containers is smaller than the original file sizes, whereas the pointer-based DOM implementations enlarge the original file sizes up to 8.6 times (see Table 7.3); this can be interpreted in [75].

The advantage of SEDOM is that it supports DOM update operations and that it supports documents of size much larger than the pointer-based DOM, given the same memory resources because the framework is much smaller than the tree representation of the pointer-based DOM and that the text value containers are held on disk. However, the only navigational operation for which times are reported in the paper is the `firstChild()` operation, which is around 20 times slower than the pointer-based DOM. Clearly, the other navigation operations would show similar or much worse running times, since several scans in the framework array is much slower than a single pointer access for the pointer-based DOM (e.g. Xerces).

3.2.2 XML Compressors

We now review two specialized XML compressors that achieve excellent compression ratios ([13] [48]), but do not support query operations without the de-compression of the XML document in its entirety.

XMill

XMill [48], a locally homogeneous compressed representation of XML documents shows good compression performance in terms of time and memory usage. There are two main phases during the parsing process:

(a) **Separating Structure from Content**. Start tags are replaced by an integer value relating to the element name in a dictionary array containing all unique elements used in the document, which reduces the space usage as repeated tags are replaced by the codes. The end tags are replaced by the symbol `/`. Attributes are represented as (element) tags with the symbol `@` prefixed to its name in the dictionary array (which distinguishes an attribute name from an element name). Each data value (including attribute values) is replaced by a container ID number from (b). The structure of the XML document in XMill's compressed structure was 1%-3% of the compressed file (for data-centric files), but was approximately 20% of a certain document-centric file (`Treebank.xml`, see Section 3.3).

(b) **Grouping data values**. Each data value is uniquely assigned to one data container. The mapping from data values to containers is resolved by the data value's path and by user parameters to assign containers to specific paths. This allows the user to group into the same containers data values of the same type or with similar patterns of data, hence aids better compression. For example, the data values relating to the path `/Doc/Person/Title` and `/Doc/Book/Title` may have different patterns of data, and should therefore be stored in different containers. The data containers are compressed according to *semantic* compressors on the data; these are built-in encoders, designed to encode specific data type and perform better according to the patterns of the data. For example,

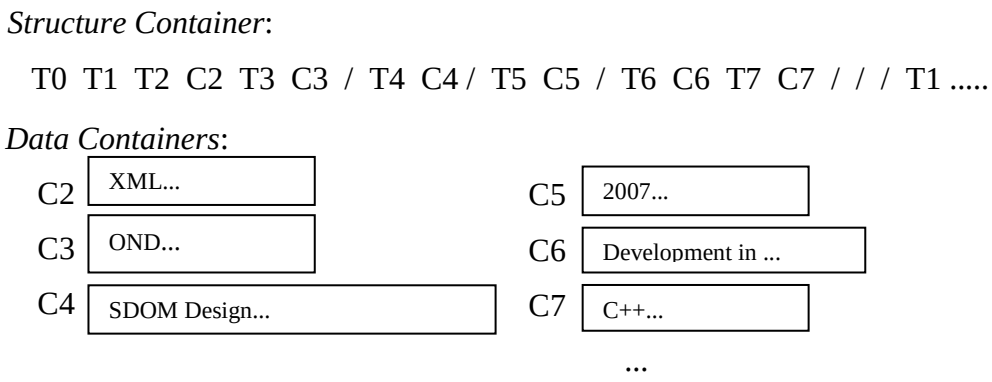


Figure 3.4 - Abstract view of XMill for a single book in the XML document of Figure 2.1.

the *run-length* encodes integers more efficiently (particularly if we have duplicated values) than if, we used another encoder. During compression, default settings are applied, however the user can specify as expressions the encoders to use for the containers, and to link-in other encoders not available in XMill.

Figure 3.4 shows a conceptual view of the compressed document produced by XMill for a simple XML document. The structure of the XML document is maintained by applying dictionary encodings of the tags `bookshop`, `book`, `author`, etc (encoded as `T0`, `T1`, `T3`, etc). The attribute `@catalogue` is encoded as `T2`. The data values are arranged in the containers according to their parent tag.

XMill compared well against the generic compressor GZip. The compressed file was $45\% - 60\%$ the size of GZip for data-centric documents with default settings: using semantic compressors, XMill reduced the size to $35\% - 47\%$ of GZip. For documents with mostly text, XMill was only slightly better.

XMill is one of the first specialised XML compressors. However not supporting querying over the compressed data has limited its application mainly to archival and transmission.

XMLPPM

XMLPPM [13], a locally homogeneous compressed representation of XML documents shows good compression in terms of time and memory by applying the technique of encoding SAX streams and on Prediction by Partial Match (PPM) encoding [17]; this technique is called *Multiplexed Hierarchical Modeling* (MHM).

In summary, the PPM model maintains statistics on previous symbols seen so far in the uncompressed stream. For each symbol, the model is used to estimate a probability of the next symbol in the stream. The PPM compression encodes the symbols using arithmetic coding, although it is possible to use other encodings [13].

The MHM technique handles the SAX event streams received during parsing within four PPM models; one for element and attribute names (`Syms`), one for the element structure (`Elts`), one for attribute values (`Atts`), and the other for text values (`Chars`). The models operate independently, but share access to one underlying arithmetic coder.

Table 3.4 shows the status of the four PPM models when an XML document is converted into a corresponding stream by XMLPPM; for the `Syms` model, the string of the XML name is stored the first time the name appears, and other times a unique byte code is used. The byte code is sent to the `Elts` model to indicate a start tag (e.g. in Table 3.4 the *book* element has the byte code `01`). When an end tag is received, the token `FF` is sent to the `Elts` model representing the end tag. Attribute names (e.g. `catalogue`) are sent to the `Syms` model and the attribute name token (i.e. `OD`) are sent to the `Att` model. Attribute values, such as "`XML`", are sent to the `Atts` model with their corresponding attribute name token, i.e. `OD`. Data values, such as "`OND`", are sent to the `Chars` model.

Table 3.4 –Multiplexed hierarchical modelling in XMLPPM. The Model is a snippet of an XML document in Figure 2.1.

Model	<book	catalogue=	"XML"	>	<author>	OND	</author>	...	</book>
Syms:	book 00	catalogue 00			author 00				
Elts:	01				02	FE	<02>FF		<01>FF
Atts:		<01>OD	XML 00	FF					
Chars:						<02>OND 00			

MHM breaks existing homogeneous property, due to the several models used. To restore the homogeneous property in XML documents, which aids the prediction process, XMLPPM *injects* the enclosing token index (i.e. the token is in the format <nn>) into the corresponding `Elts`, `Att`, or `Chars` model immediately before an `element`, `attribute`, or data value is encoded (see Table 3.4). These tokens indicate to the models that a particular token has been seen, without explicitly encoding or decoding it.

XMLPPM has the benefit over XMill in supporting *online* processing, which means the compressor is able to stream the compressed data to the decoder, rather than requiring the entire compressed data before decompression can begin (i.e. *offline*). [13] claim XMLPPM achieves compression on XML documents that are document-centric and data-centric, 5% and 10-35% better, respectively than the best existing XML compressor (i.e. XMill [13]). However, in XMLPPM the compression time is slower than others.

3.2.3 Query-friendly XML Compressors

We now review eight query-friendly XML compressors.

XBZipIndex

Ferragina et al. [30] present two XML processing tools. The first is an XML compressor called XBZip, which represents the XML document in its compressed format on disk, and requires full decompression before any querying can be done. The other tool, on which we focus, is a query-friendly XML compressor called XBZipIndex. These representations are locally homogeneous and are based on the principles of the Burrows-Wheeler Transform [9]. Their main result is the XBW transform algorithm that represents the tree structure of the XML document better for compression.

We describe the process with an example. The XML document d of a book record (Figure 2.1) corresponds to the tree T given in Figure 3.5. The opening tags such as "`<book>`" are replaced by the string "`<book`". `Attribute` nodes are stored in the tree as `element` nodes, labelled with the symbol `@` at the start of the `attribute` name (i.e. `@catalogue`) to distinguish them from element names. The textual data for `text` nodes or the `attribute` node values creates two nodes in the tree: a special node called the *skip* node with the label '=', and a *content* node, which is the child of the skip node. The label of the content node has the form $\emptyset p$, where p is the textual value, e.g. $\emptyset$2007, and $\emptyset$ is a terminating character in d.

A pre-order traversal of T is carried out to build the arrays in Figure 3.6 (left), representing all nodes. The string S_{last} indicates for each node if it is a last child node (if it is, the value **1** is stored otherwise, a **0** is stored). The string $S_{\propto}$ stores the label of each node in T. The string S_{π} shows the upward path of nodes given from the parent of the node in the tree. The string S_{pcdata} stores a concatenation of the data values in the document.

These strings are the input into the XBW transform, which applies a stable sort according to the S_{π} component, arranging the triplets $\langle S_{last}, S_{\propto}, S_{\pi}\rangle$ in lexicographical order to produce the set $\langle S_{last}, S_{\propto}, S_{\pi}\rangle$ in Figure 3.6 (right). We observe that element nodes (i.e. nodes labelled with '<') appear before attributes (i.e. nodes labelled with

'@'), and in this order appear before text nodes (i.e. node labelled with '='). Given that the number of nodes in the XML tree is $t = n + l$, where n is the number of internal nodes and l the number of leaves, then $S_\propto[1, n]$ contains the labels of the internal nodes and $S_\propto[n+1, t]$ contains the textual values (*pcdata*). Since leaves have no children $S_{last}[i] = 1$ for $i = n+1, \ldots, t$. To avoid the wasteful representation of $S_{last}[n+1, t]$ they split $S_\propto$ and S_{last} into $\langle \hat{S}_{last}, \hat{S}_\propto, \hat{S}_{\text{pcdata}} \rangle$. The output of XBW are the sorted strings $\hat{S}_\propto$, $\hat{S}_{last}$ and $\hat{S}_{\text{pcdata}}$ in Figure 3.6 (bottom). The strings are stored separately, as the tree structure (i.e. $\hat{S}_\propto$ and $\hat{S}_{last}$) and the textual content (i.e. $\hat{S}_{\text{pcdata}}$).

The two main advantages of the output representation of XBW are as follows: firstly, the strings $\hat{S}_{\text{pcdata}}$ and $\hat{S}_\propto$ uphold the locally homogeneous property in the re-arrangement of the textual values in the transform, hence they are highly compressible. Secondly, search and navigation operations over T are greatly simplified.

If we are only interested in a compressed representation of d then the arrays $\hat{S}_\propto$, $\hat{S}_{last}$ and $\hat{S}_{\text{pcdata}}$ are stored compactly as possible. This is done by merging $\hat{S}_{last}$ and $\hat{S}_\propto$ in a single array called $\hat{S}'_\propto$, where we insert after each internal label the special label </ if the previous label corresponded to a 1 bit in $\hat{S}_{last}$. $\hat{S}'_\propto$ and $\hat{S}_{\text{pcdata}}$ are compressed using the general-purpose compressor PpMDI [63]. The compressed representation is called XBZip.

If we are interested in the support of navigation and searching, the $\hat{S}_{last}$ bit-string and the $\hat{S}_\propto$ string are represented with the support of the `RANK` and `SELECT` operations (we will study these in Chapter 4). This representation with query support is called XBZipIndex. Also, to support navigation and searching the array of labels is split into blocks and compressed individually using the Gzip text compressor, and decompression is only applied to a single block instead of the entire array (in order to execute a navigation step). The text content array is much larger and requires a more sophisticated compression tool called the FM-Index that offers substring searching (essential for XPath queries).

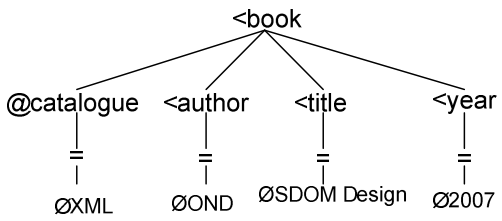


Figure 3.5 - Ordered label tree of a simple XML document

S_{last}	$S_\propto$	S_π
1	<book	*empty string*
0	@catalogue	<book
1	=	@catalogue<book
1	ØXML	=@catalogue<book
0	<author	<book
1	=	<author<book
1	ØOND	=<author<book
0	<title	<book
1	=	<title<book
1	ØSDOM Design	=<title<book
1	<year	<book
1	=	<year<book
1	Ø2007	=<year<book

→

RK	S_{last}	$S_\propto$	S_π
1	1	<book	*empty string*
2	1	=	<author<book
3	0	@catalogue	<book
4	0	<author	<book
5	0	<title	<book
6	1	<year	<book
7	1	=	<title<book
8	1	=	<year<book
9	1	=	@catalogue<book
10	1	ØOND	=<author<book
11	1	ØSDOM Design	=<title<book
12	1	Ø2007	=<year<book
13	1	ØXML	=@catalogue<book

$\hat{S}_{last} = 110001111$

$\hat{S}_\propto = < book = @catalogue < author < title < year ===$

$\hat{S}_{\text{pcdata}} = ØOND ØSDOM\ Design Ø2007 ØXML$

Figure 3.6 – Left: Set *S* after the pre-order visit of T. Right: The set *S* after the stable sort. Bottom:The three arrays $\hat{S}_\propto$, $\hat{S}_{last}$ and $\hat{S}_{\text{pcdata}}$, output of the XBW transform.

XBZip showed the best compression ratio compared to the XML compressors that are not query-friendly (e.g. XMill), however the compressors lie within a 5% absolute difference in their compression ratios. XBZipIndex offers 20% to 30% better compression compared to XPRESS, XQZip.

XQueC

XQueC [3] has a locally homogenous compressed representation of XML documents and supports a large sub-set of XQuery queries upon the compressed representation. XQueC is implemented in Java, and is arranged in the following data structure:

(a) **Node name dictionary**: The unique `element`/`attribute` names are stored in a dictionary. If there are e unique names, then a unique bit-string is assigned to each name requiring $\lceil \log_2 e \rceil$ bits each, called a *tag code* (which is discussed later).

(b) **Tree structure**: The non-text nodes are represented as *node records* stored in a sequence. Each record entry contains its own ID (identifying the node in the tree), the corresponding tag code, a list of IDs of its children and the ID of its parent. A search tree is constructed and stored on top of the sequence of node records, with the ID as the search key, to achieve better query performance. For example, given we are at a node in the tree, to navigate to the parent node record, we retrieve the parent ID in the current node record, which is then used in the search tree to find the parent node record. For each node record there is a pointer to its `attribute` or `text` value in their respective containers in (c).

(c) **Value Containers.** The use of containers is similar to the ideas in XMill, where the data values are stored in containers according to (i) the root to leaf path in the tree structure, which are strongly homogeneous hence highly compressible, and (ii) the type of the data value. XQueC makes the improvement based on XMill by partitioning the containers into *container records* and a compressed value and a pointer to its parent node record are stored in each record. The container records are grouped into lexicographical order (for fast binary searching) and then compressed using a text compression algorithm.

A small tree structure is stored to represent unique paths in the document, this is called the *structure summary*. The leaf nodes in the structure summary point to the corresponding value container. The storing of this structure is somewhat redundant; however, the purpose is to cut down the cost of traversing the whole structure tree in

query operations. Indeed the space usage of the structure summary is about 19% of the original document size.

XQueC primarily focuses on querying; however, we study the potential navigation support on the tree structure, which is given as follows:

- `firstChild()`: for the current node record, the first ID in the sequence of IDs is the first child ID. We then use the search tree to find the node in the sequence of node record.
- `nextSibling()`, `previousSibling()`: we call the `parent()` operation at the current node, then get the ID of the next sibling node (or previous sibling), which is following (or proceeding) the ID of the current node. We then use the search tree to find the node in the sequence of node record.
- `parent()`: the ID of the parent node is given with each node record, therefore we use the search tree with the parent ID to find its node record.

The retrieval of text nodes requires loading and decompressing data from disk, however the frequently accessed data is cached. Queries are efficiently executed as follows: We parse the structure summary for the matching path. We then navigate the tree structure according to the path given by the structure summary. The leaf node of the path in the tree structure points to the `text` value in the data container record. This is then retrieved from the compressed storage structure.

The compression ratio of XQueC is compared with XMill, XPRESS [52] and XGrind [64] (we discuss XPRESS and XGrind later). XMill is clearly better in terms of compression ratio, however, among query-friendly compressors XQueC appears to compresses as well as XPRESS, and better than XGrind. It could even be better if white spaces in the XML document

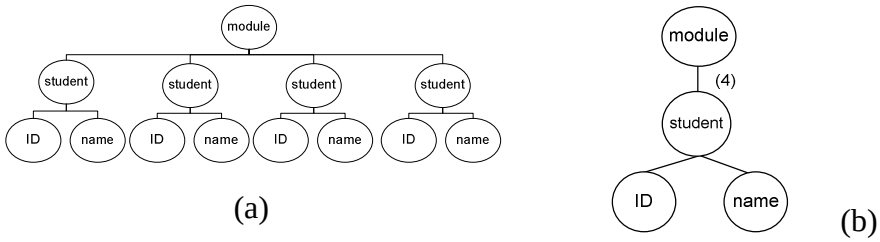


Figure 3.7 – (a): Unranked tree of XML document. (b): Compressed DAG version of (a).

were ignored in the compressed representation, as in XPRESS. Finally, XQueC supports a larger fragment of XPath and XQuery, than XPRESS and XGrind. The query performance of XQueC is much better than the XQuery processor called Galax [33] in most cases.

Path Queries on Compressed XML

Buneman et al. [7] present a query-friendly XML compressor that is locally homogeneous and supports the Core XPath queries. [7] was one of the first to provide compression on the XML tree structure, where the novel *sharing of common subtrees* technique is used to reduce the XML tree to a minimal DAG representation. They further reduce the size of the minimal DAG representation by using multiplicity counters for consecutive equal subtrees (see Figure 3.7 (b)). [10] and [14] later followed with techniques to compress the XML tree structure based upon the minimal DAG representation.

In Figure 3.7, we show the tree structure of an XML document in (a), and the corresponding compressed DAG representation in (b). In the compressed representation the edges that have consecutive sequence of out-edges to the same node are replaced by a single edge and marked with the appropriate cardinality.

The pre-processing phase requires a SAX parser, a stack and a hash table to build the DAG in one parse of the XML document. The DAG is built using pointers to node objects in main memory in a bottom-up approach; the stack is used to track nodes under construction and the hash table is used to keep track of nodes already in the compressed instance that is being created.

It is possible to partially decompress the DAG representation with some of the nodes in the original tree that would be selected by an XPath query. In [7] the data values are not compressed, therefore the original XML document is re-parsed each time we require the access of a data value. Therefore, the compression results shown were only on the XML tree structure and not including the data values. In [7], they claim the compressed tree is about $\frac{1}{10}$ to $\frac{1}{15}$ of the uncompressed XML tree. In regards to the query performance, [7] reports the result that a Core XPath query Q can be evaluated on an XML document represented by a DAG D in time $O(2^{|Q|} \times |D|)$, where $|D|$ is the number of nodes of D.

Query Evaluation on Compressed Trees

Frick et al. [32] study XPath query evaluation on compressed XML tree structures; this is related work to that in [7] where the XML tree structure is compressed as DAG representation. The results are theoretical, i.e. there is no practical implementation developed in this paper. However, they observed that XPath is not easy to define from a theoretical viewpoint; therefore, the authors show that the XPath language can be mapped efficiently to *monadic datalog* language [38]. Frick et al. show that the evaluation problem of queries for monadic datalog on a compressed instance is PSPACE-complete. They then show, again for XPath, there exists an algorithm to solve the monadic datalog evaluation in $O(k \times 2^k \times n)$ time, where k denotes the size of the datalog program and n is the size of the compressed instance.

The complexity results above only consider unary edges in the DAG representation. The inclusion of edge multiplicities can have a practical impact on the compression rate. To avoid the potential problem and to represent the edge multiplicities, an XML tree structure I is initially transformed to binary tree (called $B(I)$), then to a minimized binary tree $M(B(I))$. We show in Figure 3.8 (b) the binary tree representation of a fragment of the XML tree structure (a), where we have a node with degree 7 and the child nodes are duplicate nodes. In the binary tree the node with i children is replaced by an almost complete binary tree of height $2^{\lceil \log(i) \rceil}$. The other unary relations of I can be directly transferred to $B(I)$.

In Figure 3.9, we show the minimisation of $B(I)$. We observe that there is a blow-up in size by a factor that is logarithmic in the maximum edge multiplicity.

We cannot draw upon any practical results in [32] to compare to the other query-friendly XML compressors, but their study shows the correlation of DAG representations and XML trees. Experimental evaluations is shown in [7], [8], [14] and [10].

Vectorizing and Querying Large XML Repositories

Buneman et al. [8] presents a query-friendly XML compressor that is locally homogeneous. The XML tree structures are compressed as DAG representations. This work extends the work done in [7] to support XQuery without decompression, where the compressed representation only did support XPath. The evaluation of queries yields new, usually smaller XML tree structures, which are DAG representations.

The XQuery system of [8] is called *VX*. The compressed XML tree is held in main-memory. As in XMill [48], the data values are represented as containers (also called *vectors*). However, these remain uncompressed and are grouped under their unique path from the root node to the leaf in the XML tree structure. The vectors are held on disk as separate clustered files.

In the experimental evaluation of [8], comparison of VX is made against the SQL Server and a few XQuery systems, including MonetDB [6], Galax [33]. The running times for the queries in VX were competitive if not better than the other XQuery systems for most queries evaluated. For a particular query that required constructing portions of the original XML document VX is better than MonetDB by almost 2.5 orders of magnitude [8]. For the query that required the matching of all data values, MonetDB was significantly better than VX.

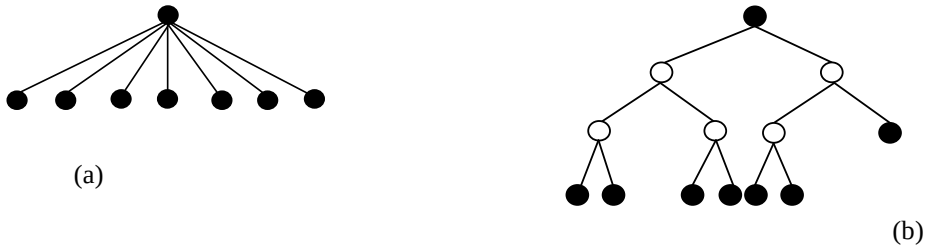


Figure 3.8 – (a) Fragment of an XML tree structure: node has degree 7, of the same node. (b) Binary tree representation of (a).

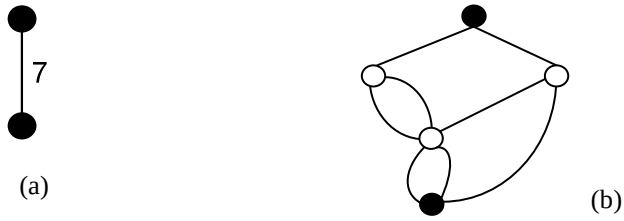


Figure 3.9 – (a) DAG representation of XML tree structure in Figure 3.8 (a). (b) Minimised binary tree of Figure 3.8 (b).

XQZip

XQZip [14] is locally homogeneous in its compressed representation of XML documents and supports a number of XPath queries. As in XMill, the data items are grouped into containers for compression. The new ideas here are a blocking strategy on the data value containers to reduce the amount of data to be decompressed, and a strategy on the tree structure of the XML document to reduce the tree to a DAG. The DAG representation, called the *XML Structure Index Tree (SIT)*, is significantly smaller than the original tree structure.

SIT: We maintain the tree structure of the XML document with all duplicate sub-trees removed. Initially we have a tree with all non-text nodes, such as the element and attribute nodes (we interpret the attribute values as text nodes, which are maintained along with text nodes in containers). Each node is assigned a tree node number that is in

level-order, and a hash ID representing the element or attribute name in the hash table. The attribute names are prefixed by the special symbol @.

Define a tree structure as $T = (V_T, E_T, ROOT)$, where V_T and E_T are the set of nodes and edges, respectively, and $ROOT$ is the unique root of T. A tree node $v \in V_T$ is a triple $v = (eid, nid, ext)$, where $v.eid$ is a Hash ID, $v.nid$ is the unique document order number and $v.ext$ initially is $\{v.nid\}$. $v.ext$ will subsequently contain IDs of removed sub-tree nodes.

Each node is represented as the pair $(v.eid, v.nid)$. The ROOT is uniquely assigned $(0,0)$. Before reduction of the tree, each node is connected using four pointers (to the parent, first-child, next-sibling and previous-sibling nodes). Sibling nodes are ordered according to their Hash ID (eid), i.e. if the children of a node are $B_1, \dots, B_n$, then $v.B_1.eid \leq v.B_2.eid \leq \cdots \leq v.B_n.eid$. If any of the Hash IDs are equal then the nid number is used to break ties. This node ordering accelerates node matching by a factor of two, since two nodes are matched by their eid and on average, only half the children of a given node need be searched. We now discuss the process of reducing the tree structure to a DAG by the following steps:

- **Step 1: Branch and Branch ordering:** A branch is a unique path from the root to a leaf node in the tree T. Given two branches $b_1 = v_0 \to \cdots \to v_1 \to \cdots \to v_p$ and $b_2 = u_0 \to \cdots \to u_1 \to \cdots \to u_p$, where v_p and u_p are leaf nodes of T. We say b_1 is ordered before b_2 if when we scan through the nodes there exists a node where either $b_1.u_i.eid < b_2.v_i.eid$ or eid values are the same but nid values are different with b_1 node lower than b_2 node value.
- **Step 2: SIT-Equivalence:** Two branches $b_1 = v_0 \to \cdots \to v_1 \to \cdots \to v_p$ and $b_2 = u_0 \to \cdots \to u_1 \to \cdots \to u_q$, are SIT-equivalent if $v_i.eid = u_i.eid$ for $0 \leq i \leq p$ and $p = q$. The purpose of this step is to identify identical branches in terms of their eid code. Two sub-trees are SIT-equivalent if they are siblings and all their branches are branch ordered and the ith branch in each are SIT-equivalent, for $0 \leq i \leq m$ and $m = n$, where m and n is the count of branches in the trees, respectively.

- **Step 3: Merge Operator:** This operator performs a merge on two sub-trees, t_1 and t_2 to produce t in place of t_1 in the tree, where t is SIT-equivalent to t_1 and t_2. The effect of the merge operator is that the duplicate SIT-equivalent structure is removed. The nids of the t_1 sub-tree nodes are used in t and the nids of the t_2 sub-tree nodes are added to the ext set of the equivalent node in t.

The construction of the SIT maintains four pointers per node in the tree, pointers to the parent, previous-sibling, next-sibling and first-child node. This in turn maintains the speed of navigation for query evaluation. The space usage of SIT pointers is usually insignificant, as many tree structures of XML documents have repeated patterns of sub-trees, which we now reduce to only a single occurrence. Construction of SIT is achieved using the SAX parser, with time in $O(|V_T|)$ or $O(|SIT||V_T|)$ in the worst case, where SIT is the set of nodes in the SIT tree.

XQZip groups the data values with the same tag/attribute parent into the same data value containers (similar to XQueC). We divide each container into blocks, compress the blocks using GZip and store them on disk. We assign each compressed block an ID, which is given as the highest node ID in the tree structure that is represented in the block. In addition, the starting position of each block is stored in an array, as hash table values. To retrieve a block contained in the compressed containers of a node, we match the node ID with those stored in the blocks. Then we obtain the starting position of the block using a binary search on the array to retrieve the block, which is then decompressed. Other nodes such as processing instruction, comment and namespace handling are not considered in this model; however, the authors mention that extension for these node types is trivial. The advantage of the blocking strategy is that in many query evaluations we only require the decompression of individual blocks hence we avoid full decompression.

XQZip achieves approximately the same compression ratio as XMill. The implementation of XQZip supports most of the core features of XPath, with navigation

support directly focused on the eight XPath axes. The speed of queries was compared against XGrind [64], which they out-performed by a factor of 12.84.

XCQ

XCQ [56] is locally homogenous in its compressed representation of XML documents and supports a number of XPath queries. It makes use of the information provided by the DTD in the compression and query evaluation process.

To compress the XML document the DTD is first represented as a tree, which is used along with a *SAX* parser, this we call the *DSP technique*. The DSP technique has two purposes, (1) to extract structural information from the input XML document that cannot be directly gathered by the DTD during parsing, (2) to group data values according to their paths given by the DTD tree.

A DTD tree is traversed as the document is parsed, so the next event of the SAX parser is expected to match to the next node in the DTD tree. This process constructs the following:

(a) **Structure stream**: The DTD tree is traversed to construct this stream; for each node, if the next node cannot be derived directly from the DTD tree then we output a special symbol, which indicates the node that is the one in the XML document. For example, to keep track of how many times a repetition node group is used in the document a **1** bit is inserted into the structure stream, otherwise a **0** bit is inserted indicating no more repeats. Also for an optional node, where the DTD specifies a choice of $1..x$ nodes, the value i is outputted to the structure stream, representing the existence of the ith child that is used in the document.

(b) **Data streams**: The DTD tree is traversed to construct the data stream. The DTD tree knows when the next node is a data value, which is given as output to a container of data values grouped according to tree paths in the DTD tree. The use of the tree path to group data into containers aids compression, queries and improves upon the simple names used in XMill. These containers

are divided in blocks (discussed below). The blocks are compressed using GZip.

The data streams are represented using the strategies of *Partitioned Path-Based Grouping* (PPG) and *Block Statistics Signature* (BSS). The data is compressed according to paths in a number of streams of blocks. We have a BSS index per block, which summarizes the content in that particular block. For example, if we had a group of numbers in a block then the BSS index would store the minimum, maximum, sum and count of the values present. Similar summaries can be applied to alphabetic data. When querying for a data value in a block stored on disk, the BSS index is first consulted, which will filter out blocks that do not contain the required data, before any block is accessed and searched.

We explain the querying process by example. In Figure 3.7, we show a compressed XML document in XCQ. Given the query:

```
record/book[@catalogue='XML' and year/text()='2007']
```

The query only involves the data streams D0 and D3. We first access the data stream D0 with the path key `records/book/@catalogue`. The entire data stream has to be decompressed since we do not know where the text 'XML' appears. If the word 'XML' appears in blocks 0 and 1 out of many blocks, we then only have to decompress the blocks 0 and 1 in the data stream D3 with path key /records/book/year for the matching text '2007'. Once the block is found with both matches, we then decompress the related blocks in the other data streams to construct the query output to the user.

XCQ can be used as an XML compressor by applying GZip to the structure and data streams into a single file on disk, which compresses better than GZip and XMill (see Figure 13 of [56]).

The compression time of XCQ is slow because there is an initial construction of a tree structure of the DTD and the continual traversal of the DTD tree for the construction of the structure stream. XCQ compression time is slightly longer than XMill.

In contrast, without applying the GZip to the structure stream, XCQ is query-friendly; we are required to decompress at least a single data stream for query matching. Running time performance of XCQ is not assessed [56], but preliminary results are given in their appendix[2].

XGrind

The first of two query-friendly XML compressors we review that use a homomorphic compressed representation is XGrind [64]. The steps required to construct the compressed document are as follows. An initial scan of the document and DTD is required, where statistics are gathered for converting elements and attribute names to dictionary-based codes and to build a set of *non-adaptive context-free* Huffman coders for the data values. The second scan performs the actual encoding of the XML names and data values.

We show in Figure 3.11 a simple XML document and its compressed representation using XGrind. Like XMill, the elements and attribute names are dictionary encoded: each opening tag is encoded by the character '`T`' followed by its unique element ID, i.e. the `book` element is replaced by `T0`. The closing tags are encoded by the character '`/`'. An attribute node is encoded by the character '`A`' followed by its unique attribute ID. The Huffman code of a data value x is denoted by $H(x)$. For attributes that are of an enumerated type, XGrind uses the DTD to assign them a special value that is held in a symbol table.

XGrind claim compression on average is 33.9% of the file size, which is based upon files used in [64]. XGrind supports a variety of common XQL queries, such as `exact-match`, `prefix-match`, `range-match` and `partial-match`. To query the compressed representation the query expression must be converted into a compressed equivalent form. This is achieved by a *lexical analyzer* that replaces the tag names in the query expression to the

[2] XCQ appendix (2005) experimental data of XCQ performance: http://www.cs.ust.hk/~wilfred/XCQ/appndix.pdf

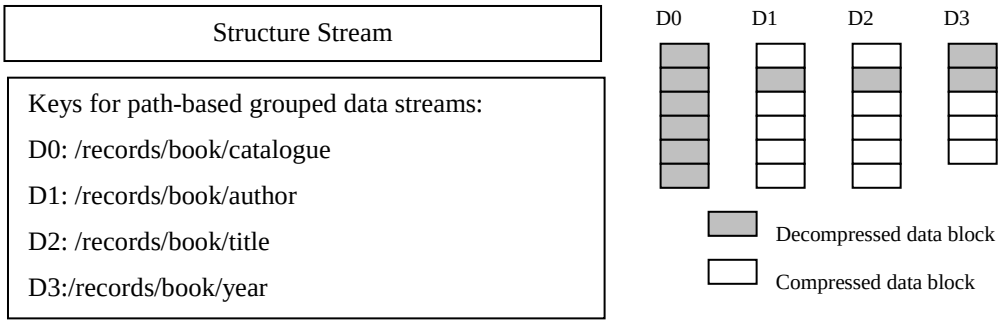


Figure 3.10 – XCQ. Decompressed data blocks when processing query example.

```
<book catalogue='XML'>
<author>OND</author>
<title>SDOM Design</title>
<year>2007</year>
</book>
```

```
T0 A0 H(XML)
  T1 H(OND) /
  T3 H(SDOM Design) /
  T4 H(2007) /
/
```

Figure 3.11 – Left: Example XML document. Right: compressed XGrind representation.

dictionary codes and data value codes used in the compressed representation. A byte-by-byte comparison is made on the compressed document for a pattern matching query. We observe that although for such queries we avoid decompression, the compression ratio of XGrind is much worse than most XML compressors and the construction time is slow since it requires two scans of the XML document.

XPRESS

The second query-friendly XML compressor that is homomorphic in its representation is called XPRESS [52]. The key ideas are the *Reverse Arithmetic Encoding* method which encodes a labelled path of tags into unique intervals in the range [0.0,1.0) and the *automatic type inference* which applies different encoders (e.g. Huffman encoding) to the individual data values, depending on their data type. We examine the Reverse Arithmetic Encoding by the use of a simple example. Given an XML document of a university's organisational structure:

```
<university>
  <faculty>
    <department>
      <module>module1</module>
      ...
    </department>
  </faculty>
</university>
```

Table 3.5 - The interval $[0.65, 0.66)$ is obtained for the simple path `university/department/module`.

Simple path	**Interval**
`module`	[0.5,1.0)
`department/module`	[0.65,0.75)
`university/department/module`	[0.65,0.66)

In the compression phase of XPRESS, we require two scans of the XML document, the first scan performs the gathering of statistics to provide the necessary information in the second scan, which compresses the XML document. Some of the statistics gathered in the first scan are the element tags and their frequency of appearance. In our example above, we have the following element tags: `university`, `faculty`, `department` and `module`. Suppose that the frequency of appearance of `university`, `faculty`, `department` and `module` tags in the document is 0.1, 0.2, 0.2 and 0.5 respectively. Initially we assign each tag name an interval, depending on how frequently they appear in the document. In our example, we give the following intervals: `university` [0.0,0.1), `faculty` [0.1,0.3), `department` [0.3,0.5) and `module` [0.5,1.0). The size of the interval of T is proportional to the frequency of the element T.

Reverse Arithmetic Encoding operates on paths. In Figure 3.4, we show the output steps in the function `reverse_arithmetic_encoding` defined in [52] to obtain the interval $[0.65, 0.66)$ for the path `university/department/module`. We observe that if a simple path P has an interval I, then the interval of all suffix paths of P contains

I, . For example, the interval that represents the element path "`/department/module`" contains the interval that represents the element path: "`/university/department/module`", since "`/department/module`" is a suffix. It is easy to compute sub-path queries with the pattern "$//*/subpath$".

XPRESS stores a compressed XML document by replacing the start tags by the minimum value of the subinterval depending upon the path. In the decompression phase, the tags can be obtained by a binary search of intervals.

Type Inference engine: The data type information received in the parsing phase is passed to a dictionary based type inference engine for encoding. For the integer data types, the type inference engine uses the path information to apply binary encoding and then differential encoding to the set of values.

Data types are categorized into two types for textual data. Firstly, they have the string type, which is for general text data; they apply the Huffman encoding to such data types. Secondly, the enumeration type can handle distinct patterns of the textual data up to 128. If the enumeration count is above 128 then the dictionary encoding is used for such textual data. This selection of the best-suited data value compressor is automatic.

To evaluate queries on the compressed data, a given path is transformed into an interval and then XPRESS scans the compressed file for values in that interval. Integer data values are converted to encoded values for matching in the file without decompressing. Textual data require a partial decompression.

We observe that XPRESS does not cover all queries and navigational support is limited. Improvement is made upon the ideas of XGrind by encoding the tree paths instead of the element tags themselves. This provides direct support for path-based queries. XPRESS supports `exact-match` and `prefix-match` on the compressed data, `partial-match` and `range-match` on decompressed data, and the XPath axes `child`, `descendant` and `attribute`. The support of `range-match` on the compressed data is only for numeric data.

XPRESS claim compression on average is 27% of the file size, which outperforms XGrind especially where data values are integers, enumeration and floating point.

Table 3.6 – Comparison of XML processors and compressors.

XML Tool	Compression ratio	Compression time	XML language support	Navigation support	Update support	Homomorphic
TinyTree	NA	NA	XQuery, XSLT, XPATH, DOM	Yes	No	No
DDOM	NA	Slow	DOM	Yes	No	No
BPLEX	Very good	Slow	DOM	Yes	No	No
SEDOM	Good	Fast	DOM	Yes	Yes	No
XMILL	Very good	Fast	Not supported	No	No	No
XMLPPM	Very Good	Slow	Not supported	No	No	No
XBZipIndex	Very Good	Slow	XPath	Yes	No	No
XQueC	Poor	NA	XPath, XQuery	Yes	No	No
VX	Poor	NA	XPath, XQuery	Yes	No	No
XQZip	Very Good	Fast	XPath 1.0	Yes, slow	No	No
XCQ	Very Good	Fast	XPath 1.0 subset	Yes, slow	No	No
XGRIND	Poor	Slow	XPath subset	Yes, slow	No	Yes
XPRESS	Poor	Slow	XPath subset	Yes, slow	No	Yes

However, on average it is still 20% worse than XMill. For querying, XPRESS takes few seconds to evaluate queries in [52], which is better than XGrind by a factor of 2.83. For navigation support, if a node has many descendants, its sibling will be located quite far away in the (compressed or original) file. In support of the operations `nextSibling`/`previousSibling()`, `next`/`previous()` XGrind or XPRESS may be quite slow.

3.2.4 XML Compressor Summary

We now summarize our discussion of XML compressors, including `TinyTree`, in terms of compression ratio, compression time and functionality. We found that XMill had the best average performance in terms of compression time and compression ratio; however, its lack of query support limits its use. The support of DOM navigational operations by the query-friendly XML compressors was slow, this is because they are designed to support path-based queries, whereas DOM is designed for a much wider scope of applications in-mind and features navigation upon the documents. See Table 3.6 for the full details.

3.3 Statistics of XML documents

We now describe a corpus of XML documents taken from [73] and [74]. Fifteen XML documents have been selected based on their wide range of characteristics and are described in Table 3.7. We also used synthetic XML files that were created using the *xmlgen* data generator [70]. Xmlgen generates a typically well-structured XML document, based upon processes of an auction website. The file sizes range from 0.13MB to 593.6MB and the DOM node count of these files range from 7,000 to 25 million nodes.

In Table 3.8, we show the count of node types, unique tag/attribute names that appears in the DOM trees for the XML documents in our corpus, we also show the count of element and attribute nodes that appear with a namespace. We include the statistics of the synthetic XML files, which we name `XMARK`+[file size]+`MB`. For all our files, the majority of the nodes are `text` nodes (60% of nodes in the tree, on average over all files). The next largest types are the `element` nodes with 38% on average over all files. We have not included `attribute` nodes in these averages, as they are not a part of the tree and not all XML documents use them. However when they are used, there is sometimes a large number of them, e.g. the file `Mondial-3.0.xml` has 104,795 nodes in the document, of these 45% are `attribute` nodes, also in the file `w3c1.xml` 28% of the nodes are `attribute` nodes.

We examine the properties of the DOM tree representations of the XML documents further in Table 3.8. The tree properties we examine in each file are the proportion of leaf nodes to non-leaf nodes, the maximum depth in the tree, which is the maximum number of nodes in any path from the root to a leaf node. We also examine the largest degree of a single node, and observe that there are a large number of leaf nodes, which have node degree zero. The maximum depth of an XML tree generally is quite low. However, there are some documents that have a fairly deep tree, for example, `Treebank_e.xml` has a maximum depth of 37.

Table 3.7 - Description of XML files in our XML corpus taken from [73].

XML Documents:	**Description:**
`Elts.xml`	Describes chemical elements in the periodic table.
`w3c1.xml`	W3C specification documentation
`w3c2.xml`	W3C specification documentation
`Mondial-3.0.xml`	World geographic database integrated from the CIA World Factbook, the International Atlas, and the TERRA database among other sources. From FLORID-Mondial case study
`Partsupp.xml`	Part/Supplier relationship. TPC-H Benchmark, 10 MB version, in XML form. Converted to XML by Zack Ives. From Transaction Processing Performance Council (TPC).
`Orders.xml`	Orders. TPC-H Benchmark, 10 MB version, in XML form. Converted to XML by Zack Ives. From Transaction Processing Performance Council (TPC).
`xCRL.xml`	XML files using the Extensible Customer Representation Language format (xCRL) on customer relationship management
`Votable2.xml`	File created in the VOTABLE XML format defined for the exchange of data.
`Nasa.xml`	Datasets converted from legacy flat-file format into XML and made available to the public. From GSFC/NASA XML Project.
`Lineitem.xml`	Line items. TPC-H Benchmark, 10 MB version, in XML form. Converted to XML by Zack Ives. From Transaction Processing Performance Council (TPC).
`XPATH.xml`	Is not in [73], but uses the LocusXML schema to represent geospatial information in an XML format, it stores annotated human genomic data.
`Treebank_e.xml`	English sentences, tagged with parts of speech. The text nodes have been partially encrypted because they are copyrighted text from the Wall Street Journal. This document has a deep recursive structure. University of Pennsylvania Treebank project.
`SwissProt.xml`	SWISS-PROT is a curated protein sequence database, which strives to provide a high level of annotations (such as the description of the function of a protein, its domains structure, post-translational modifications, variants, etc.), a minimal level of redundancy and high level of integration with other databases. From ExPASy - SWISS-PROT and TrEMBL.
`DBLP.xml`	The DBLP server provides bibliographic information on major computer science journals and proceedings. DBLP stands for Digital Bibliography Library Project. From the DBLP Homepage.
`XCDNA.xml`	A cDNA library of a collection of cloned fragments converted into an XML form.

Table 3.8 – Size and node distribution according to DOM node type of all the XML documents in our corpus. Assume all XML documents have a document node. (EL: Element, ATT: Attribute, ER: EntityReference, ENT: Entity, COM: Comment, DT: DocType, NS: Namespace)

XML FILES	SIZE (MB)	NODES	TREE NODES	EL	ATT	TEXT	ER	ENT	COM	DT	UNIQUE NAMES	ELEM & ATTR WITH NS
Elts	0.13	6927	5992	1897	936	3896	0	0	197	0	22	0
w3c1	0.22	18809	13299	4176	5190	7689	1430	321	1	1	64	4216
w3c2	0.19	16984	12169	3696	4495	7102	1367	321	1	1	63	3736
Mondial-3.0	1.1	104795	57373	22423	47423	34947	0	0	1	0	50	0
Partsupp	2.2	96004	96004	48001	1	48001	0	0	0	0	8	0
Orders	5.1	300004	300004	150001	1	150001	0	0	0	0	12	0
xCRL	8.5	333245	259423	98723	73823	155625	0	0	5073	0	112	0
Votable2	15.6	1991870	1991193	1150175	678	840989	0	0	27	0	29	2
Nasa	23.8	1481852	1425536	476646	56317	948888	0	0	0	0	70	30152
Lineitem	31.6	2045954	2045954	1022976	1	1022976	0	0	0	0	19	0
XPATH	49.8	2522571	2522572	840857	0	1681713	0	0	0	0	42	0
Treebank_e	82	7312613	7312613	2437666	1	4874945	0	0	0	0	251	0
SwissProt	109.5	10599084	8409226	2977031	2189859	5432193	0	0	0	0	99	0
DBLP	127.6	10595379	10191037	3332130	404276	6792148	66756	67	0	1	40	0
XCDNA	593.6	25221153	25221154	8407051	0	16814101	0	0	0	0	210	0
XMARK2MB	2.72	123,582	114404	40600	9178	73803	0	0	0	0	77	0
XMARK4MB	3.94	179,435	166114	58957	13321	107156	0	0	0	0	77	0
XMARK8MB	7.82	361,187	333867	118669	27320	215197	0	0	0	0	77	0
XMARK16MB	15.63	719,454	665188	236322	54266	428865	0	0	0	0	77	0
XMARK32MB	31.23	1,422,486	1315903	467275	106583	848627	0	0	0	0	77	0
XMARK64MB	63.10	2,877,347	2661006	945248	216341	1715757	0	0	0	0	77	0
XMARK128MB	124.8	5,689,748	5261055	1869171	428693	3391883	0	0	0	0	77	0
XMARK256MB	256.9	11,697,794	10816629	3842922	881165	6973706	0	0	0	0	77	0

Table 3.9 - Statistics of XML documents trees for our corpus.

XML FILES	TREE NODES	LEAF	MAX. DEPTH	LARGEST DEGREE
Elts	5992	68%	4	225
w3c1	13299	61%	16	412
w3c2	12169	62%	15	412
Mondial-3.0	57373	78%	6	1911
Partsupp	96004	50%	4	16001
Orders	300004	50%	4	30001
xCRL	259423	80%	19	3113
Votable2	1991193	58%	8	19999
Nasa	1425536	67%	9	4871
Lineitem	2045954	50%	4	120351
XPATH	2522572	67%	6	42075
Treebank_e	7312613	67%	37	112769
SwissProt	8409226	71%	6	100000
DBLP	10191037	67%	7	657717
XCDNA	25221154	67%	8	82237
XMARK2MB	114404	71%	13	1225
XMARK4MB	166114	71%	13	1785
XMARK8MB	333867	71%	13	3571
XMARK16MB	665188	71%	13	7241
XMARK32MB	1315903	71%	13	14281
XMARK64MB	2661006	71%	13	28865
XMARK128MB	5261055	71%	13	57121
XMARK256MB	10816629	71%	13	117299

3.3.1 Textual Data

Table 3.10 shows the statistics of the textual data gathered belonging to attribute values and text nodes of files in our XML corpus. We report the percentage of `text` nodes in the DOM tree, leaf nodes, average length of the individual text data and their total length. For `attribute` nodes, we report the count of nodes, total length of attribute values and average length of each attribute value.

There are a number of observations on the textual data. Firstly, textual data comprises on average 46% of our XML file sizes. However, the proportion varies a lot depending on whether the file is data centric or document-centric. For example, the textual data of the `Treebank_e.xml` data-centric file accounted for 70% of the file size, whereas a document-centric file such as `Lineitem.xml` only accounted for 19%.

Table 3.10 – Statistics on textual data distribution. We report file size, text & attributes node count, % leaf nodes in tree (% of text nodes) and average textual data length. For negligible we use NEG.

		Text Nodes					Attribute Nodes		
File	Size	#nodes	%Leaf	%Text	avg	Text Len.	#nodes	Value Len.	Avg
Elts	128KB	3896	68%	65%	7	25KB	936	14KB	15
w3c1	224KB	7689	61%	58%	15	116KB	5190	36KB	7
w3c2	200KB	7102	62%	58%	16	110KB	4495	26KB	6
Mondial-3	1.1MB	39.3K	78%	61%	11	392KB	47K	296KB	6
Partsupp	2.20MB	48.0K	50%	50%	23	1.1MB	1	NEG	8
Orders	5.1MB	150.0K	50%	50%	10	1.5MB	1	NEG	6
xCRL	8.5MB	155.6K	80%	60%	12	1.8MB	73K	1.2MB	17
Votable2	15.6MB	841.0K	58%	42%	7	5.2MB	678	7KB	10
Nasa	23.8MB	948.9K	67%	67%	16	14.4MB	56K	776KB	14
Lineitem	31.6MB	1.0M	50%	50%	6	6.0MB	1	NEG	8
XPATH	49.8MB	1.7M	67%	67%	8	13.0MB	0	0	0
Treebank_e	82MB	4.9M	67%	67%	12	57.4MB	1	NEG	8
SwissProt	109.5MB	5.4M	71%	65%	7	35.4MB	2.2M	13.2MB	6
DBLP	127.6MB	6.8M	67%	67%	10	64.0MB	404.2K	7.3MB	19
XCDNA	593.6MB	16.8M	67%	67%	16	255.8MB	0	0	0

The average individual data value length over all XML files for text nodes and attribute values was approximately twelve and nine characters, respectively. The range of value length reaches up to twenty-three characters in one file and as low as six characters long in another file.

The proportion of leaf nodes present in all our files on average was 64%. This is expected as we have already noted that there are large counts of text nodes in the XML documents (see Table 3.8), and the fact that they cannot have children in the DOM. It is interesting to note that text nodes comprise 91%-100% of the leaf nodes in our XML files.

In summary, based on the statistics gathered in Table 3.8, Table 3.8 and Table 3.10 we make the following observations:

i. XML documents have many text nodes.

ii. average length of a text node is relatively low.

iii. XML documents often have large attribute node count.

iv. unique XML names are few.

Trees have:

v. few node types that are not of the element, text or attribute node type.

vi. relatively low depth.

vii. large leaf node count.

some nodes with large degrees.

3.4 Summary

We have studied in detail the Xerces XML processor. Xerces provides an almost full implementation of the DOM, but uses a lot of space. We have also studied in detail the Saxon `TinyTree` data structure, which has reduced the space usage of representing the tree to 20-30 bytes per node, even though it is limited to support of read-only operations.

The XML compressor XMill [48] appears to be the best for compression (closely matched by XBZip [30]), and is much better than many generic text compressors. XMill and XBZip minimize storage and transmission time, but querying is not directly supported. We also studied a number of query-friendly XML compressors [3], [10], [12], [30], [52], [55], [56] and [64], which answer queries inspecting only a small fraction of the XML file and in principle only a fraction of the compressed file is decompressed as well. However, only a few of these query-friendly XML compressors offer fast support for DOM-like navigation, for example moving from a tag to its sibling in one operation. Other compressors such as BPLEX [10] or XBZIPIndex [30] do support navigation using the compressed representation. A detailed experimental evaluation focusing on navigation speeds is not presented in either paper.

We have discussed DOM implementations such as DDOM [55] for data centric files and SEDOM [75], which present a space efficient representation supporting both read-only and update operations. Running time performance of the navigational operation is reported only for the `firstChild()` operation, but it is stated in [75], the other

navigation operations give similar response time pattern, even though they are potentially slower, e.g. `lastChild()`.

We tried to get some understanding how best to represent XML documents by examining the characteristics of a range of XML documents in our corpus.

Chapter 4
Succinct Data Structures

Succinct, or highly space-efficient, data structures that support operations rapidly were pioneered by Jacobson [44]. Succinct data structures represent certain data objects; for each kind of object, we begin by giving their *succinct lower bound*, which is an information-theoretic lower bound on the amount of space needed to represent the object. We then discuss the corresponding data structures that use a small amount of space in addition to the succinct bound to support a number of operations upon the data object.

4.1 Information-theoretic lower bounds on space usage

The succinct lower bound on the space required to store an object can be obtained as follows. Suppose that the algorithm knows that the object that it needs to represent is one object from a set S of objects. Then, the algorithm must use $\lceil \lg |S| \rceil$ bits[3] to represent an object from S in the worst case (otherwise, the algorithm would represent two distinct objects from S the same way). We now give examples of succinct lower bounds.

4.1.1 Bit-strings

The set of objects here is all bit-strings of length n. We assume that the algorithm knows that it has to store a bit-string, and also knows the length of the bit-string. Since there are 2^n such bit-strings, taking the logarithm base two of this number, we get:

Proposition 4.1. *The succinct lower bound for representing bit-string of length n bits is n bits.*

Remark – The succinct lower bound of a bit-string indicates that the best possible representation of a bit-string is given by writing the bit-string down itself. In other words, the obvious representation is succinct.

4.1.2 Balanced Parentheses

The set of objects here is all balanced parentheses strings of length $2n$. A balanced parentheses string of length $2n$ is a string which contains n opening and n closing parentheses, and which is balanced, i.e., within any prefix of the string, there are

[3] We use $\lg x$ to denote $\log_2 x$.

always at least as many opening parentheses as closing parentheses. We assume that the algorithm knows it has to store a string of $2n$ balanced parentheses.

((( ))) ( ) ( ) ( ) (( )) ( ) ( ) (( )) (( ) ( ))

Figure 4.1 – The set of balanced parentheses for $n = 3$.

It can be shown that there are $C_n = \frac{1}{n+1}\binom{2n}{n}$ such objects in the set, where C_n is the nth Catalan number. For example, for $n = 3$, we have $C_3 = (1/4) * (6 * 5 * 4)/6 = 5$, and there are five sequences of six balanced parentheses (Figure 4.1). Taking the logarithm base 2 of C_n we obtain:

Proposition 4.2. *The succinct lower bound for representing a balanced sequence of* $2n$ *parentheses is* $2n - O(\log n)$ *bits.*

4.1.3 Ordinal trees

The set of objects here is all *ordinal* trees with n nodes. An ordinal tree is an arbitrary rooted tree where the children of each node are ordered. We assume that the algorithm knows the number of nodes. It can be shown that there are C_{n-1} ordinal trees on n nodes. For example, there are $C_3 = 5$ ordinal trees with $n = 4$ (see Figure 4.2). Since the order of children matters, the third and fourth ordinal trees below are different.

Taking the logarithm base 2 of C_{n-1} we obtain:

Proposition 4.3. *The succinct lower bound for representing an ordinal tree with* n *nodes is* $2n - O(log\, n)$ *bits.*

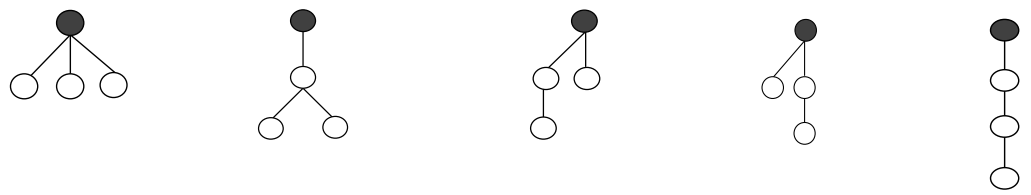


Figure 4.2 – The set of ordinal trees for $n = 4$. Root node is shaded in grey.

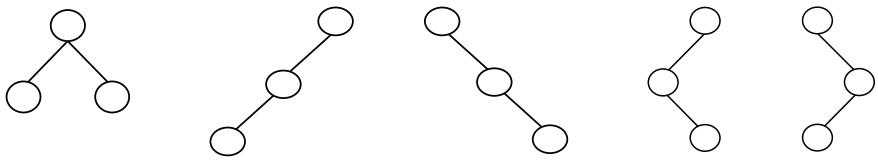


Figure 4.3 - The set of binary trees for $n = 3$.

4.1.4 Binary Tree

The set of objects here is all binary trees with n nodes. A binary tree is a rooted tree with each node having space for a pointer to a left and a right child node; nodes may have only one child node (left or right), both, or no child nodes (leaf node). It can be shown that there are C_n such objects in the set, for example, for $n = 3$, there are $C_3 = 5$ binary trees on three nodes (see Figure 4.3).

Taking the logarithm base two of C_n we obtain:

Proposition 4.4. *The succinct lower bound for representing a binary tree of n nodes is* $2n - O(\log n)$ *bits.*

4.1.5 Prefix-sums

The set of objects here is a sequence $\boldsymbol{x} = (x_1, \ldots, x_n)$ of n positive integers that add up to m. We assume that the algorithm knows n and m. It can be shown that there are $l = \binom{m-1}{n-1}$ such objects in the set. For example, for $n = 3$ and $m = 6$, we have $l = \frac{5!}{2!3!} = 10$ sequences: (1,1,4), (1,4,1), (4,1,1), (1,2,3), (2,1,3), (2,3,1), (1,3,2), (3,1,2), (3,2,1) and (2,2,2). Taking the logarithm base two of l, and using the inequality $\binom{m}{n} \leq \left(\frac{me}{n}\right)^n$ [20] we obtain:

Proposition 4.5. *The information-theoretic lower bound for representing a sequence of n positive integers that add up to m is* $\lceil \lg l \rceil \leq n \lg\left(\frac{m}{n}\right) + n \lg e$ *bits.*

4.1.6 Succinctness vs Data Compression

Succinctness is related to, but distinct from, data compression. It can be applied to many data types such as numeric values, strings, dictionaries, tree, etc. Representing random data, the space usage of a succinct representation is usually good, but it misses out on space savings for regular data, typically captured by compression algorithms, for example, repeated tags in an XML document. In particular, the size of a representation can be estimated quite accurately using the number of input entities in an instance of the data structure.

4.2 Succinct Data Structure

We now describe the succinct data structures to represent the data objects given in Section 4.1. The space usage of these data structures are designed to be close to the corresponding succinct lower bounds and they support the desired operations rapidly. In what follows, we say that $f(n) = o(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = 0$. Thus, a space bound of $n + o(n)$ bits means a space bound of $(1 + \varepsilon_n)\, n$ bits, where ε_n goes to zero as n grows. For each data structure we state the operations it supports, the obvious or naive solution, the succinct solution and its implementation details.

We give the implementation details because the implementations described here are used together in the Succinct DOM application presented in Chapter 7. In addition, some of these implementation details are not given in the published literature, hence the space usage formulas we give in this chapter allow us to verify the space usage costs stated for the DOM application.

4.2.1 Bit-Vector Data structure

The object to be represented is a bit-string x of length n (ref. Section 4.1.1). The operations to be supported are:

- $\mathsf{SELECT}_1(x, i)$: Given an index i, returns the position of the ith **1** bit in x.
- $\mathsf{RANK}_1(x, i)$: Returns the number of **1**s to the left of, and including, position i in x.

$\texttt{SELECT}_0$ and $\texttt{RANK}_0$ are defined analogously for the **0** bits in the bit-string; the operations are collectively referred to as RANK and SELECT. We refer to a data structure that supports (a non-empty subset of) RANK and SELECT operations on a bit-string as a *bit-vector*.

For example, if x = **1 0 0 1 1 0 1 0** then $\texttt{SELECT}_1(x, 4) = 7$ (the fourth **1** is in position seven) and $\texttt{RANK}_1(x, 4) = 2$ (there are two **1**s in positions one to four). These operations are inverse of each other, in that $\texttt{RANK}_1(x, \texttt{SELECT}_1(x, i)) = i$ and $\texttt{RANK}_0(x, \texttt{SELECT}_0(x, i)) = i$ for the index i contained within x. Given $\texttt{RANK}_1(x, i) = j$ we observe that $\texttt{RANK}_0(x, i)$ is computed for free as $i - j$. Hence, by supporting $\texttt{RANK}_1$, we automatically support $\texttt{RANK}_0$ without requiring any additional data structures. Therefore, in this chapter when referring to RANK, it implies both the $\texttt{RANK}_1$ and $\texttt{RANK}_0$ implementation together, even though $\texttt{RANK}_1$ is discussed.

Naive Representation

A naive representation to support $\texttt{RANK}_1$ would be to explicitly store the count of **1**s at each position in the bit-string in an array of length n, with space usage $n \lg n$ bits. As noted above, $\texttt{RANK}_0$ would be automatically supported. For $\texttt{SELECT}_1$ we would explicitly store the position of each **1** in the bit-string in an array of length n_1, where n_1 is the count of **1**s, therefore the space usage is $n_1 \lg n$ bits. Supporting $\texttt{SELECT}_0$ is analogous, but applied to the **0**s. Supporting RANK and SELECT together requires $2n \lg n$ bits.

Succinct Solution

The following is known about a bit-string of length n:

Theorem 4.1. ([51], Chapter 37) *There are bit-vector data structures that use $n + o(n)$ bits to support* SELECT *and* RANK *operations in $O(1)$ time.*

Implementations

We used three bit-vector implementations developed by Naila Rahman; in what follows, we refer to these as CJ, which is based upon ideas of [15], [16], [36] and [44], KNKP, which is based upon ideas of [47], and CNEW, which is based upon ideas of Naila Rahman [23]. CJ and KNKP implementations support the RANK and SELECT operations, but we only describe the SELECT_1 operation (SELECT_0 works analogously). CNEW supports RANK, and either SELECT_0 or SELECT_1, but not both. The implementations and their parameters are aimed at practical performance, and some implementations below even use $O(n)$ bits, rather than $n + o(n)$ bits, from an asymptotic viewpoint. For all the bit-vector implementation we assume $n \leq 2^{32}$.

CJ Implementation

Let n be the size of the given bit-string A. The implementation uses the parameters b, B, s and $L_G = 8s$, which are all powers of two. The bit-string is stored in blocks of size B bits and each block is divided into sub-blocks of size b bits, where $b < B$ and $B \leq 512$. b is limited to the value 8 or 16.

RANK: The implementation of the RANK operation is as follows:

- We use an array D of length n/B, where $D[i]$ stores the number of **1**s in A up to the start of the ith block. The space usage of D is $32(n/B)$ bits.
- Two static lookup tables are created. The first table, which is of size 2^b bytes, maps all combinations of bits in a sub-block by the array index to the count of **1**s in that sub-block. The second table, which is of size $b \times b/8$ bytes, stores a single mask for each position in a sub-block. The total space used by the lookup tables is $2^b + b \times b/8$ bytes.

Therefore, the total space usage of this RANK implementation is $32(n/B) + n$ bits, including the bit-string but not the static lookup tables.

$\mathsf{RANK}_1(A,i)$ is computed as follows: we first retrieve the sum of **1**s for the blocks before the i/B-th block by accessing the array element $D[\lfloor i/B \rfloor]$. If i is not a multiple of B we

add to this value the count of **1**s up to the sub-block where the i-th bit resides by using the first lookup table. In the worst case we would do $B/b - 1$ table lookups. Finally in the sub-block where the i-th bit resides (if i is not a multiple of b), we add the count of **1**s up to and including the i-th bit using the second lookup table to mask out the bits after i in the sub-block. Then we use the first lookup table to count the number of **1**s in the b bits up to i.

SELECT: The $\texttt{SELECT}_1$ operation is implemented as follows:

- For every s-th **1** in A, we store its position in the array E_1. This array requires $32n_1/s$ bits, where n_1 is the number of **1**s in A.
- A *large gap* in A appears where the positions of the is-th **1** and $(i+1)s$-th **1** differ by more than $L_G = 8s$. We define a bit-string R of length n/B bits, which stores at the jth position a **1** bit indicating the start of a large gap somewhere in the jth block in A, and stores a **0** bit otherwise. We add to R data structures to support the RANK operation. The space usage of R is $32(n/B^2) + n/B$ bits. If there is a large gap starting at position is, we store in an array C the position of the **1**s from is up to $(i+1)s - 1$. The space usage of array C is $32 * l_1 * s$ bits, where l_1 is the number of long gaps. Clearly, $l_1 \leq n/L_G$.
- We make use of the RANK lookup tables and introduce two new tables for selecting the ith **1** in their relative sub-block position, these require $b * 2^b$ bytes each.

Since we choose $L_G = 8s$, C takes up at most $32 * s * n/8s = 4n$ bits in the worst case. At first sight this seems costly, however in general we observe this cost is not often paid. However in Chapter 5 we will see an example of a bit-vector derived from an XML document that has a large count of large gaps due to its unique structure. The cost of large gaps is noticeable in the overall space usage.

$\texttt{SELECT}_1(A,i)$ is computed as follows. Letting $z = \lfloor i/s \rfloor$ we get the position of the $\lfloor i/s \rfloor \times s$-th **1** by reading $E_1[z]$. If i is a multiple of s, we return this value. If not, we check if i is within a large gap. If $E_1[z] + L_G < E_1[z+1]$ we know that the ith **1** is

within a large gap. Then we do a simple computation to find the offset p of i in values $is \ldots (i+1)s-1$, namely we return $C[\mathsf{RANK}(R,z)+p]$ as the position of the ith 1. If the ith 1 is not within a large gap we use the RANK array D to check in which block i is located. We now require the use of the RANK lookup tables; in the first lookup table we count the number of 1s in each sub-block up to the sub-block of where the i-th 1 is found. We keep the total of bits before the sub-block of the i-th 1 that is in A. The second lookup table is used to count the number of bits up to and including the i-th 1 within its sub-block, which is achieved by a mask of the bits up to the i-th bit. Then we use the first lookup table to count the bits, which is then added to the total and returned.

KNKP Implementation

Let n be the size of the given bit-string A. This implementation uses the parameters b, B and SB, which are all powers of two. Also, $b < B < SB$, and the implementation assumes $SB \leq 256$. The bit-string is stored in super-blocks of size SB bits and we divide each super-block into blocks of size B bits, and each block is in turn divided into sub-blocks of b bits.

RANK: The implementation of the RANK operation is as follows:

- We use an array D of length n/SB, which stores as running totals, the count of **1**s in A every SB bits. The space usage of D is $32(n/SB)$ bits.
- We use an array d of length n/B, which stores the running totals of the count of **1**s every B bits. At the start of each super-block the totals are initialised to zero. Since $SB \leq 256$ the values in d fit into a byte, therefore the space usage of d is $8(n/B)$ bits.
- We use two static lookup tables. See CJ RANK implementation for details.

The total space usage of this RANK implementation (including the bit-string, but not the static lookup tables) is $32(n/SB) + 8(n/B) + n$ bits.

$\mathsf{RANK}_1(A,i)$ is computed similarly to CJ.

SELECT: The implementation of the SELECT_1 operation is as follows:

- We use a bit-string Q of length n_1, where n_1 is the number of **1**s in A. In Q we store at the ith position a **0** bit, if the ith **1** and $(i-1)$th **1** in A are contained within the same block, and store a **1** bit otherwise. The first bit in Q is always set to **1**. Q is augmented with additional bits to support RANK. The space usage is $32(n_1/(SB)) + 8(n_1/B) + n_1$ bits.
- Conceptually, we use a bit-string P of length n/B, which indicates all-zero and non-zero blocks (i.e. blocks that contain at least one **1**) in A using a **0** bit and **1** bit, respectively. P need not be maintained, we only need to support the SELECT operation on P as follows:
    - We use a bit-string R, which is similar to Q, but applied to P. Its length is the number of **1**s in P and is at most n/B. We define $R[0]$ as 0 if $P[0] = 1$, and 1 otherwise. R is augmented with additional bits to support RANK. If z is the number of **1**s in P (this is the same as the number of non-zero blocks in A), the space usage is $32(z/(SB)) + 8(z/B) + z$ bits. The operation $\mathsf{RANK}(R, i)$ indicates the number of *clumps* before the ith **1** bit in P, where a clump is as a group of contiguous **0**s in P.
    - We use an array called the *clump array* of length c_1, where c_1 is the number of clumps in P. The ith index in the array stores the accumulated count of zeros up to the ith clump in P. The space usage is $32c_1$ bits and in the worse case $c_1 = n/2B$; however in practice c_1 is small, hence the array is often small.

  To compute $\mathsf{SELECT}_1(P, i)$, we first compute $\mathsf{RANK}(R, i)$, which reports the number of clumps before i, then we compute the number j of **0**s in front of the ith **1** bit using the clump array. Then $\mathsf{SELECT}_1(P, i) = i + j$.

The total space usage of SELECT (including the bit-string A) is $n + 32(n_1/(SB)) + 8(n_1/B) + 32(z/(SB)) + 8(z/B) + 32c_1$ bits.

The computation of SELECT is summarised as follows: RANK(Q, y) gives the number of non-zero blocks up to and including the block containing the yth **1** bit of A. SELECT$_1$(P,RANK(Q, y)) then gives the block containing the yth **1** bit. Further scanning of the block is required using lookup tables, to calculate SELECT$_1(A, y)$.

CNEW Implementation

Let n be the size of the given bit-string A. This implementation uses parameters b and B, which are both powers of two. We divide the bit-string into blocks of size B bits (where $B \leq 256$) and further divide the blocks into sub-blocks of size b. We obtain the *extracted* bit-string A' of length n' (cf. [47]) by removing all blocks in A that contain no **1**s (such blocks are called *all-zero* blocks). The blocks that remain in A are called *extracted* blocks.

RANK: The implementation of the RANK operation uses the following data structures:

We use a bit-string P of length n/B. We store at the i-th position a **0** bit if the i-th block in A is an all-zero block, otherwise a **1** bit is stored. We augment this bit-string with additional bits to support RANK (using the CJ implementation).

The original bit-string is not maintained as it can be reconstructed from P and A'. RANK$_1(A, i)$ is computed as follows: we get the count of extracted blocks before i by $y =$ RANK$_1$ $(P, \lfloor i/B \rfloor)$, map position i to its position i' in A' by computing $i' = i - (\lfloor i/B \rfloor - y) \times B$. If i was in an all-zero block (which we can check by looking at $P[\lfloor i/B \rfloor]$) then it does not exist in A', in this case we set $i' = yB - 1$. In each case, we return RANK$_1(A', i')$ as the answer.

The implementation of RANK on A' is done similarly to the CJ implementation, except that the array D is replaced by the following array:

For each block in A', we store the number of **0**s up to the start of the block that was in A in an array R of length n'/B, requiring $32(n'/B)$ bits.

Table 4.1 – Space usage of the three bit-vector implementations used. We denote n and n' as the length of the bit-strings A and A', respectively, where $n \geq n'$. n_0 and n_1 are the count of **1**s and **0**s present in the bit-string, respectively. s, b, B and SB are parameters in the data structures. l_0, l_1 are the number of the zeros and ones large gaps, respectively. In KNKP the term z is the number of extracted blocks in the input bit-string. The terms c_0 and c_1 are the sizes of the clump array.

	CJ	**KNKP**	**CNEW**
Input bit-string	n	n	n'
RANK directory	$32n/B$	$32(n/SB) + 8(n/B)$	$32(n/B^2) + 32n'/B$
SELECT_0 directory	$\frac{32n_0}{s} + \frac{n}{B} + \frac{32n}{B^2} + 32sl_0$	$32(n_1/SB) + 8(n_1/B) + 32(z/SB) + 8(z/B) + 32c_1$	*NA*
SELECT_1 directory	$\frac{32n_1}{s} + \frac{n}{B} + \frac{32n}{B^2} + 32sl_1$	$32(n_1/SB) + 8(n_1/B) + 32(z/SB) + 8(z/B) + 32c_0$	$32(n_1/s) + 8n'/B$

The change is made to help with SELECT_1 as we will see later. We observe that the number of **1**s up to the start of the ith block in A' can easily be calculated from $R[i]$, as the number of **0**s in A' up to the start of the ith block is just $R[i] - (i - y) \times B$.

SELECT: We now come to SELECT_1, for which we store the following information:

- We store the index (in A') of the location of the $is + 1$-st **1**, for $i = 0, 1, \ldots, \lfloor n_1/s \rfloor$, in an array S, where n_1 is the number of **1**s in the bit-string. As each block in A' contains at least a single **1**, adjacent entries in S differ by at most $sB - 1$. The array S requires $32(n_1/s)$ bits.
- An array BC of length n'/B stores 8-bit values that give the count of **1**s from the start of the block to the first pointer from S that lies in the block.

$\mathsf{SELECT}_1(A, i)$ is computed as follows. We first retrieve the position of the $(\lfloor i/s \rfloor \times s + 1)$-st **1** by accessing $S[\lfloor i/s \rfloor] + R[\lfloor i/s \rfloor]$. If this is the required answer it is returned. Otherwise, suppose that $S[\lfloor i/s \rfloor]$ lies in block z. We first search from the start of block z for the block containing the i-th **1** as follows:

- We move to the start of block z, and determine the number of **1**s to the start of block z. This is done by subtracting $BC[z]$ from i (assuming $S[\lfloor i/s \rfloor]$ is the first pointer from S in z, otherwise we further subtract as many multiples of s as necessary).
- We check if the i-th **1** lies in block z. Note that the number of **0**s in block z equals $(R[z+1] - R[z]) \bmod B$ (since any all-zero blocks between blocks z and $z+1$ would contribute exactly B **0**s to $R[z+1]$). From this we calculate the number of **1**s in block z and therefore the number of **1**s up to the start of block $z+1$.

  This allows us to determine whether the i-th **1** is in block z or not. If not, we repeat the process with blocks $z+1, z+2, \ldots$ until the right block is found. In the worst case, s blocks may have to be checked.

Once the block containing the i-th **1** has been found we locate the **1** within the block using table lookup on sub-blocks, as described previously for the CJ implementation.

In Table 4.2, we evaluate the space usage of the formulas for the bit-vector implementations (see formulae in Table 4.1) by giving the parameters certain values. CNEW becomes better than KNKP, when we have many all-zero blocks. The space usage of CJ and KNKP is dependent on the number of large gaps and extracted blocks, respectively, but CNEW has no hidden costs in the worst case.

For the bit-vectors CJ, KNKP and CNEW, the running time of the `RANK` operation is a little slower than a memory access and the `SELECT` operation is 2.5 times slower than `RANK`. In Chapter 5 and 6, we compare the running times of the bit-vectors (where space usage is comparable) for real-life and random bit-strings.

Table 4.2 – Assume a bit-string with $n/2$ **1**s. We show the space usage of the three bit-vector implementations. For CJ and CNEW, the parameter values are $B = 64$, $s = 32$ and $L_G = 256$, and for KNKP we use 256-bit superblocks and 64-bit blocks. Results are based on Table 4.1 formulas.

	CJ	**KNKP**	**CNEW**
Input bit-string	n	n	n'
`RANK` directory	$0.5n$	$0.25n$	$0.5n' + 0.008n$
$\texttt{SELECT}_0$ directory	$0.52n + 1024l_0$	$0.625n + 0.25z + 32c_1$	*NA*
$\texttt{SELECT}_1$ directory	$0.52n + 1024l_1$	$0.625n + 0.25z + 32c_0$	$0.125n' + 0.5n$

1 2 3 4 5 6 7 8 9 10 11 12 13 14
(( ) (( ) (( ))) ( ))

Figure 4.4 - Parentheses string sequence.

4.2.2 Balanced parentheses string

The object to be represented is a balanced parentheses string s (ref. Section 4.1.2), and the operations to be supported are:

- `ENCLOSE`(*s*, *i*): Return the position of the opening parenthesis of the parentheses pair that most immediately encloses the opening parenthesis in position *i* of *s*.
- `FINDOPEN`(*s*, *i*): Return the position of the opening parenthesis that matches the closing parenthesis in the position *i* of s.
- `FINDCLOSE`(*s*, *i*): Return the position of the closing parenthesis that matches the opening parenthesis in position *i* of s.
- `INSPECT`(*s*, *i*): Return the state of the *i*-th parentheses of *s*, which is either an opening or closing parenthesis.

For example in Figure 4.4 the operation $\texttt{ENCLOSE}(s, 7) = 4$, the opening parenthesis at position seven is enclosed by the parentheses opening at position four and closing at

position eleven. In operation $\texttt{FINDCLOSE}(s, 4) = 11$, here the closing parenthesis at position eleven is the matching parenthesis to the opening parenthesis at position four.

Naive Representation

A naive representation to support the above operations would require two arrays of length $2n$ each. In the first array, the values would store the index of the matching closing (opening) parenthesis, depending on whether we are at an opening (closing) parenthesis. Therefore we compute $j = \texttt{FINDCLOSE}(i)$ by returning the value j at array position i. If $i < j$ then j is the closing parenthesis of i, else j is an opening parenthesis of i, and we therefore return null. We compute $j = \texttt{FINDOPEN}(i)$ by accessing the value at index i. If $i > j$ then j is the opening parenthesis of i, else j is a closing parenthesis of i, therefore return null. To compute $\texttt{INSPECT}(i)$ we check the value at the ith index. If this is less than i then we know we are at an opening parenthesis, otherwise a closing parenthesis. In the second array at index i we store the index of the enclosing parenthesis, therefore supporting ENCLOSE. The total space required is $4n \lg n$ bits.

Succinct Solution

The following is known about a balanced parentheses string of length $2n$:

Theorem 4.2. *There are balanced parentheses data structures that use $2n + o(n)$ bits to support the above operations in $O(1)$ time.*

Jacobson [44] first considered this problem and gave an $O(n)$-bit representation. Munro and Raman [54], and later Geary et al. [36], gave $2n + o(n)$-bit representations that support the parentheses operations in $O(1)$ time.

Implementations

The best implementation [36] uses a parameter B, which can be set to 32, 64 or 128. Larger values of B cause the data structure to use less space but run more slowly. The space usage reported in [36] is summarised in Table 4.3. The space usage depends upon the pioneer density (PD), which is a parameter that depends upon the particular parenthesis sequence being represented. We take PD $= 2.4$, as this value was shown to

Table 4.3 – Space usage of implementations of Jacobson's (Jacob) and Geary et al.'s Parentheses DS (New), taken from Figure 6 in [36]. The units are bits per node (parenthesis pair). PD stands for pioneer density.

	PD=4		PD=2.4	
Blocksize	Jacob	New	Jacob	New
32	16.00	8.34	12.80	5.75
64	9.00	4.65	7.40	3.73
128	5.50	3.24	4.70	2.86
256	3.75		3.35	

be realistic for parenthesis sequences derived from real-world XML files (in the worst case, PD = 4). We remark here that although all operations are asymptotically $O(1)$ time, they vary in speed: `INSPECT` is the fastest, `FINDOPEN/FINDCLOSE` are next, and `ENCLOSE` is the slowest, being typically 5-6 times slower than `FINDOPEN`.

4.2.3 Binary Trees

The object to be represented is a binary tree (ref. Section 4.1.4), and the operations to be supported are:

- `LEFT-CHILD`(x): Return the left child of node x, if the node does not exist, then return null.
- `RIGHT-CHILD`(x): Return the right child of node x, if the node does not exist, then return null.
- `PARENT`(x): Return the parent of node x, if no parent exists, then return null.

Naive Representation

A naive representation of a binary tree is to use three pointers per node connecting to its left-child, right-child and parent node. Given we have n nodes in the tree the space usage required is $3n \lg n$ bits.

Succinct Solution

Clark [15] and Jacobson [44] gave a succinct representation of a binary tree with n nodes:

Theorem 4.3. *There are binary tree representations that use* $2n + o(n)$ *bits to support the above operations in* $O(1)$ *time.*

This representation uses a level-order bit-string representation of a binary tree. Consider the binary tree in

Figure 4.5 (a). To form the bit-string representation of this tree, we write a **1** in each node. We then extend the binary tree by replacing all null pointers by pointers to dummy "external" nodes, and we write a **0** in each dummy external node (see

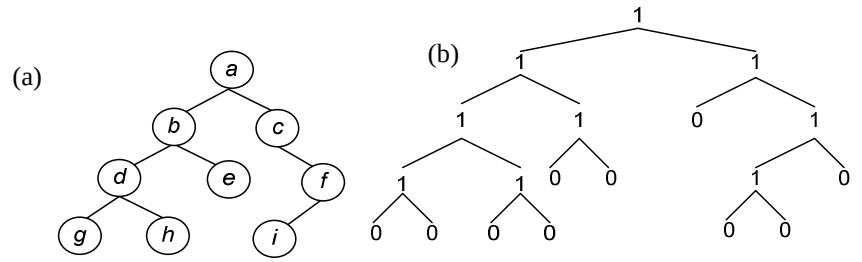
Figure **4.5** (b)). The bit-string is shown in

Figure **4.5** (c). Clearly, this representation requires $2n + 1$ bits.

This representation, like all succinct tree representations, imposes a particular numbering on the nodes of the tree. In this case, a node is numbered by the position in which the corresponding **1** appears in the bit-string. For example, node g is given the number eight. Note that node numbers are integers from 1 to $2n + 1$ and nodes are not numbered consecutively (other succinct tree representations also have node numberings with these properties).

Given this numbering, we use one of the auxiliary structures given in Section 4.2.1 that support `RANK` and `SELECT` operations on the bit-string. We then support the required navigational operations on the tree as follows, where s is the bit-string representing the tree:

- `LEFT-CHILD`$(x) = 2 * \mathsf{RANK}_1(s, x)$
- `RIGHT-CHILD`$(x) = 2 * \mathsf{RANK}_1(s, x)+1$
- `PARENT`$(x) = \mathsf{SELECT}_1(s, \lfloor x/2 \rfloor)$



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	1	1	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0
a	*b*	*c*	*d*	*e*		*f*	*g*	*h*			*i*							

(c)

Figure 4.5 - (a): Binary Tree example, (b): Labelled Extended tree and (c): Bit-string representation.

4.2.4 Ordinal Trees

The object to be represented is an ordinal tree (ref Section 4.1.3) and the operations to be supported are:

- `FIRST-CHILD`(i): Return the first child of the node i,. If the node does not exist, then return null.
- `LAST-CHILD`(i): Return the last child node of the node i. If the node does not exist, then return null.
- `PREVIOUS-SIBLING`(i): Return the previous sibling of the node i. If no previous-sibling exists then return null.
- `NEXT-SIBLING`(i): Return the next sibling node of i. If no next-sibling exists then return null.
- `PARENT`(i): Return the parent node given that we are at the ith node. If we are at the root node then return null.

Naive Representation

A naive representation of the ordinal tree is to use three pointers per node connecting to its first-child, parent and next-sibling node. Given we have n nodes in the tree the space

usage is $3n \lg n$ bits. Given that an ordinal tree can have arbitrary number of children at each node we observe that the `PREVIOUS-SIBLING` and the `LAST-CHILD` operations can be slow, if we were to use only 3 pointers per node. For example, to find the last child node of a node with large degree, we would have to traverse through the first child node and all its sibling nodes, before we reach the last child node. A simple but costly improvement would be to have two more pointers per node, connecting to the last child and previous-sibling nodes.

Succinct Solution

The following is known [44] about an ordinal tree with n nodes:

Theorem 4.4. *There are ordinal tree representations that use $2n + o(n)$ bits to support the above operations in $O(1)$ time.*

The performance bounds in Theorem 4.4 are achieved by a number of data structures. We outline three different representations of an ordinal tree. The numbering of the nodes is given differently in all three representations.

Level-order unary degree sequence representation (`LOUDS`)

The `LOUDS` bit-string (`LBS`) is defined as follows [44]. We begin with an empty string and visit every node in level-order, starting from the root. As we visit a node v with $d \geq 0$ children, we append $\mathbf{1}^d\mathbf{0}$ to the bit-string. Finally, we prefix the bit-string with a $\mathbf{10}$, which is the degree of an imaginary 'super-root,' seen as the parent of the root of the tree (see Figure 4.7). Therefore we get:

Proposition 4.6. *The* `LBS` *of an ordinal tree T with n nodes has n* **1***s and $n + 1$* **0***s.The i-th node of T in level-order is represented twice: as the i-th* **1***, which lies within the encoding of the degree of its parent, and is associated with the edge that attaches it to its parent, and also as the $i + 1$-st* **0***, which marks the end of its own degree sequence.*

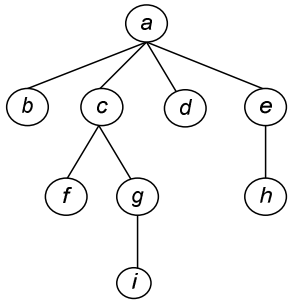


Figure 4.6 – Ordinal tree example.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Ones-based	a		b	c	d	e			f	g			h			i			
Zero-based							a	b			c	d		e	f		g	h	i
	1	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**

Figure 4.7 – The LBS of the ordinal tree of Figure 4.6. Zeros-based and ones-based numberings.

In Figure 4.7, the nodes of the tree in Figure 4.6 are numbered (again, using non-consecutive integers from 1 to $2n + 1$) in one of two different ways suggested by Proposition 4.6:

Ones-based numbering: Jacobson [44] suggested numbering the i-th node in level-order by the position of the i-th **1** bit. This gives a node a number from $\{1, \ldots, 2n + 1\}$. Table 4.4 indicates how the navigational operations might work on the ones-based numbering.

Zeros-based numbering: Geary [34] suggested numbering the i-th node in level-order by the position of the $(i + 1)$-st **0** bit, namely the bit that ends the unary sequence of that node's degree. Table 4.4 indicates how the navigational operations might work on the zeros-based numbering.

Table 4.4 - Navigation operations for zeros-based and ones-based numberings (*A* is the LBS).

Ones-based numbering	***Zeros-based numbering***

parent(x)
  if x = 1 return NULL
  else return
SELECT1(A,RANK0(A,x))
	parent(x)
	  if x = 1 return NULL
	  let y:= SELECT1(A,RANK0(A,x)-1)
	  return SELECT0(A,RANK0(A,y)+1)

first-child(x)
  let y :=
SELECT0(A,RANK1(A,x))+1
  if A[y] = 0 then return NULL
  else return y
	first-child(x)
	  if (A[x-1]=0) then return NULL
	  else
	  let y:=SELECT0(A,RANK0(A,x)-1)+1
	  return
	SELECT0(A,RANK1(A,y)+1)

last-child(x)
  let y
:=SELECT0(A,RANK1(A,x)+1)-1
  if A[y] = 0 then return NULL
  else return y
	last-child(x)
	  if (A[x-1]=0) then return NULL
	  else SELECT0(A,RANK1(A,x)+1)

next-sibling(x)
  if A[x+1] = 0 then return NULL
  else return x+1
	next-sibling(x)
	  let y := SELECT1(A,RANK0(A,x)-1)+1
	  if A[y] = 0 then return NULL
	  else return SELECT0(A,y+1)

previous-sibling(x)
  if A[x-1] = 0 then return NULL
  else return x-1
	previous-sibling(x)
	  let y := SELECT1(A,RANK0(A,x)-1)-1
	  if A[y] = 0 then return NULL else
	  return
	SELECT0(A,RANK0(A,y+1))

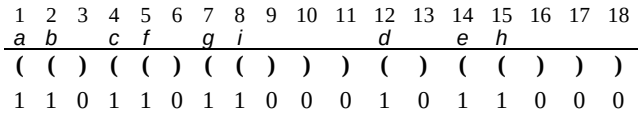


Figure 4.8 - Parentheses string of the ordinal tree in Figure 4.6.

Table 4.5 – Navigation operations for ordinal tree via the balanced parentheses representation. A is the parentheses bit-string and $A[i]$ retrieves the bit at position i in the bit-string A. Let an opening (closing) parenthesis be represented by **1**(**0**) is the bit-string.

Parentheses (node represented by opening parentheses)	

`parent(x)`
`  return ENCLOSE(A, x)`
	`next-sibling(x)`
	`  z:= FINDCLOSE(A, x)`
	`  if A[z+1]=1 then`
	`  return z+1`
	`  else return NULL`

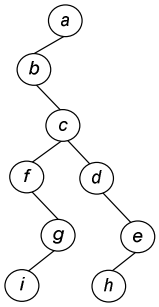
`first-child(x)`
`  if A[x+1]=1 then`
`  return x+1`
`  else return NULL`
	`previous-sibling(x)`
	`  if A[x-1]=0 then`
	`  return FINDOPEN(A, x-1)`
	`  else return NULL`

`last-child(x)`
`  if A[x+1]=1 then`
`  z:=FINDCLOSE(A, x)`
`  return FINDOPEN(A, z-1)`
`  else return NULL`	

Balanced Parentheses representation

The representation of an ordinal tree as a balanced parentheses string is defined as follows. Traverse the tree in depth-first order, and output an opening parenthesis when a node is first encountered and a closing parenthesis once all its descendants have been visited (see Figure 4.8). The bit-string at the bottom of Figure 4.8 encodes the parentheses sequence using the mapping "(" = **1** and ")" = **0**.

Jacobson [44] suggests numbering the i-th node in depth-first order by the position of the i-th "(" parenthesis. This gives a node the number from $\{1, \ldots, 2n\}$. Table 4.6 indicates how the navigational operations might work with this representation. It is possible to represent the nodes by their closing parentheses or by a pair of parentheses positions, but these appear to have no particular advantage.



(a)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	*b*			*c*	*f*	*d*		*g*		*e*	*i*		*h*					
1	1	0	0	1	1	1	0	1	0	1	1	0	1	0	0	0	0	0

(b)

Figure 4.9 – (a): Binary tree equivalent of the ordinal tree in Figure 4.6. (b): its binary tree bit-string.

Binary Tree succinct representation

An ordinal tree can be transformed into a binary tree in a one to one correspondence. The steps required are as follows:

- Ordinal tree: first child ⇔ binary tree: left child
- Ordinal tree: right sibling ⇔ binary tree: right child

We represent the resulting binary tree (Figure 4.9) using the representation of Section 4.2.1. The resulting ordinal tree representation uses $2n + o(n)$ bits and the ordinal tree navigation operations are simulated via the binary tree operations as shown in Table 4.6. As can be seen, the operations `FIRST-CHILD`, `NEXT-SIBLING` and `PREVIOUS-SIBLING` are supported in $O(1)$ time, but `LAST-CHILD` takes $O(d)$ time, where d is the degree of a node, and `PARENT` takes $O(i)$ time, if called at a node that is the i-th child of its parent. The node numbering is by level-order in the binary tree, which is neither depth-first nor level-order in the ordinal tree.

Table 4.6 – The Navigation operations for ordinal tree via binary tree. A is the bit-string. Caps represent the binary tree operations; these operations are PARENT (SELECT call), LEFT-CHILD (RANK call) and RIGHT-CHILD (RANK call).

Ordinal tree via binary tree	

`parent(x)`
`  if x=1 return NULL`
`  while(x mod 2 !=0)`
`  x:=PARENT(A, x)`
`  return PARENT(A, x)`
	`next-sibling(x)`
	`  y:=RIGHT-CHILD(A, x)`
	`  if A[y]=1 then return y`
	`  else return NULL`

`first-child(x)`
`  y:=LEFT-CHILD(A, x)`
`  if A[y]=1 return y`
`  else return NULL`
	`previous-sibling(x)`
	`  if (x mod 2) !=0`
	`  return PARENT(A, x)`
	`  else`
	`  return NULL`

`last-child(x)`
`  y:=LEFT-CHILD(A, x)`
`  if A[y]=0 return NULL`
`  else`
`  while(A[y]!=0)`
`  x:=y`
`  y:= RIGHT-CHILD(A, x)`
`  return x`	

4.2.5 Succinct Prefix sums

The object to be represented is a sequence of positive integers (ref. Section 4.1.5) $\boldsymbol{x} = (x_1, \ldots, x_n)$, where $\sum_{i=1}^{n} x_i = m$. The operation to support is as follows:

- SUM($\boldsymbol{x}, j$): Returns $\sum_{i=1}^{j} x_i$.

For example, if $\boldsymbol{x} = 1, 1, 3, 4, 5$ then $\mathsf{SUM}(\boldsymbol{x}, 3) = 5$.

Naive Representation

A naive representation to support the SUM operation would be to explicitly store each prefix sum value, requiring $n\lceil \lg m \rceil$ bits.

Succinct Solution

We use the following notation. For a sequence $\boldsymbol{x}$, its length is denoted by $|\boldsymbol{x}|$ and, if $|\boldsymbol{x}| = n$ then its components are denoted by $x_1, \ldots, x_n$. The following theorem was essentially shown by Elias [27]:

Theorem 4.5. *A sequence* $\boldsymbol{x}$ *with* $|\boldsymbol{x}| = n$ *and* $\sum_{i=1}^{n} x_i = m$ *can be represented in* $n \lg(m/n) + O(n)$ *bits so that* $\mathsf{SUM}(\boldsymbol{x}, i)$ *can be computed in* $O(1)$ *time.*

The performance bounds are achieved by the following data structure. Let $y_i = \mathsf{SUM}(\boldsymbol{x}, i)$ for $i = 1, \ldots, n$. Let u be an integer, $1 \leq u < \lg m$:

(i) We use a bit-string R of length $n(\lg m - u)$ bits, which stores the *lower-order* $\lg m - u$ bits of each y_i value concatenated together.

(ii) We use a bit-string P of length $n + 2^u$ bits. The multi-set of values formed by the *top-order* u bits is represented by coding the multiplicity of each of the values $0, \ldots, 2^u - 1$ in unary using **0**s, with the **1**s as separators. The unary values are concatenated together (P has n **0**s and 2^u **1**s).

We choose $u = \lfloor \lg n \rfloor$, so $|P| = O(n)$. We augment this bit-string with additional bits to support SELECT_0 (using an implementation from Section 4.2.1). $\mathsf{SUM}(\boldsymbol{x}, j)$ is computed as follows: we first retrieve the lower-order bits represented in R by the substring starting at pointer $z = (j-1) \times (\lg m - u) + 1$ and ending at pointer $y = j(\lg m - u)$. The top-order bits are retrieved by computing $\mathsf{SELECT}_0(j) - j$ on P. The lower and upper order values are concatenated, to give y_j, which is returned in $O(1)$ time.

We now give an example of the prefix-sums solution. Letting $\boldsymbol{x} = 3, 3, 2, 4, 2, 3, 1, 2, 2, 3, 9, 1, 7, 7, 5, 10$, $n = 16$ and $m = 64$. Let the sequence $\boldsymbol{y}$ be the prefix-sums of $\boldsymbol{x}$, that is, $\boldsymbol{y} = 3, 6, 8, 12, 14, 17, 18, 20, 22, 25, 34, 35, 42, 49, 54, 64$. We choose $u = \lfloor \lg 16 \rfloor = 4$, so we take the four top-order bits of the numbers in $\boldsymbol{y}$, see Figure 4.10 (a). We show the multiplicity of the numbers in P in Figure 4.10 (b) and encode them in Figure 4.10 (c). In this example we require $16 + 2^4 = 32$ bits to represent P. Figure 4.10 (d) shows the bit-string R.

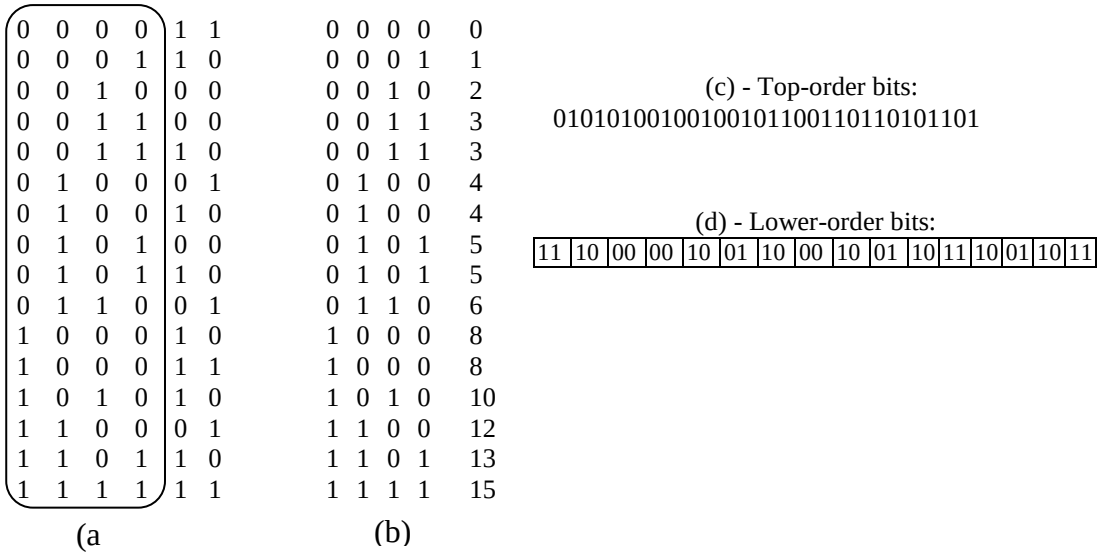


Figure 4.10 – (a) The binary representation of the numbers in $\boldsymbol{y}$. We circle the top-order bits of each number. (b) The multiplicity of the top-order numbers – given indirectly by listing their decimal values. (c) Top-order bits encoded. (d) Lower-order bits of (a) concatenated together.

The $\texttt{SUM}(\boldsymbol{x}, 6)$ operation on the representation in Figure 4.10 is computed follows: $\texttt{SELECT}_0(P, 6) - 6 = 4$, the binary representation of 4 is 0100, which gives the top-order bits of the answer. We now concatenate the lower-order bits in R to the top-order bits, by extracting the substring of R starting at position 11 and ending at position 12, which can be done in $O(1)$ time. We return $0100\ 01 = 17$.

4.3 Summary

We have given the succinct lower bounds for the representation of the following data objects: bit-strings, balanced parentheses, ordinal trees, binary trees and prefix-sums values.

Representing XML documents in succinct representations is an area of research that needs to be explored. In Chapter 5, we provide a study of succinct tree representations and show engineered representation of these to represent XML trees. In Chapter 6, we investigate the problem of the storage retrieval of the textual data in XML documents; here we employ succinct prefix sums representation and other engineered representations to solve the textual retrieval problem. In Chapter 7, we provide a more

comprehensive study of the succinct representation to support a full XML DOM application.

We have described succinct data structures that are able to support operations, usually in $O(1)$ time, upon the data object using space relatively close to the succinct lower bound. In particular, we studied several representations of ordinal trees that were implemented using the balanced parentheses, binary tree and LOUDS tree representation. A careful analysis of the navigational operations for succinct tree representations is provided.

The ideas of `RANK` and `SELECT` operational support on the bit-string in the succinct representations underpin the potential speed improvements of succinct data structures in the area of XML processing compared to other XML processors.

Chapter 5
Engineering Succinct Tree Representations

In this chapter we investigate how best to represent the tree structure of XML documents. We begin with some motivations for the study of succinct tree structures. In Section 5.2, we summarise the basic characteristics of the tree structure of XML documents, and also the DOM operations that impact upon the tree structure, in order to derive requirements for our succinct tree representations. In Section 5.3 and 5.4, we discuss our work on engineering succinct tree representations to support the requirements, and in Section 5.6, we evaluate our implementations empirically. Parts of this chapter were published as [22].

We obtain the *tree structure* of an XML document by removing:

- Non-tree nodes (i.e. `attribute` nodes).
- Textual data (i.e. `text` node, `comment` or `CDATASection` values).
- Element labels and node type information.

For example, the tree structure of the XML document in Figure 5.1 (a) is shown in Figure 5.1 (b). Note that the tree structure of an XML document has the following properties:

(i) It has a root, and the parent-child relationship between two connected nodes is therefore well-defined.

(ii) The number of children of a node is unbounded.

(iii) The order of children matters, since changing the order corresponds to a different document.

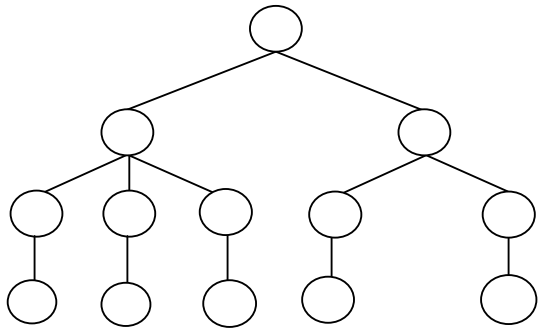
Thus, the tree structure of an XML document can be modelled as an ordinal tree (ref. Section 4.1.3). In this chapter, we use n to denote the number of nodes in the tree structure.

5.1 Motivation

As noted in Chapter 3, existing DOM implementations represent the tree structure using $3-5$ pointers per node, or 96 to 320 bits per node (assuming pointers are 32 or 64 bits

```
<library>
  <book catalogue="XML">
    <author>OND</author>
    <title>SDOM Design</title>
    <year>2007</year>
  </book>
  <book catalogue="XML">
    <author>Jones</author>
    <title>The DOM
Spec</title>
  </book>
</library>
```

(a)



(b)

Figure 5.1 - (a): Example XML document. (b): XML tree structure of (a).

long). In theory, it is possible to represent a tree structure with n nodes using just $2n + o(n)$ bits and support navigation in $O(1)$ time (see Chapter 4). However, a careful investigation of the practical performance of the representations is justified, because:

- Only the parentheses representation has been investigated before. Other, potentially practical, representations like the `LOUDS` and the binary tree representations have not.
- The $o(n)$ term in the space usage is significant in practice. The space usage of the succinct tree implementations in [36] ranges from 2.87 bits to 5.75 bits per node, even though the theoretical space usage is $2n + o(n)$ bits.
- There has been no attempt to study these representations for the specific task of representing the tree structure of XML documents, which have many unique characteristics.

5.2 XML and DOM Characteristics

The characteristics of XML files and the DOM specification are summarised in Chapter 2 and Chapter 3.

5.2.1 DOM functionality

DOM supports navigation with the operations `parent()`, `firstChild()`, `nextSibling()`, `previousSibling()` and `lastChild()`. These are contained in the DOM `Node` API, in addition to the operations to retrieve information associated with each node, for example, the `getNodeName()` and the `getNodeType()` operations returns the name and the type of a node, respectively. The `getNodeValue()` operation returns the textual value of the node that has a value. Traversals are an important navigation operation in DOM: they allow an entire XML document to be read and processed. The two main orders of traversal (described in Chapter 2) are:

- Document order, which corresponds to pre-order.
- Reverse document order.

Traversals can be implemented by a user via the navigational operations provided in the `Node` API. Traversals can be either recursive or non-recursive: the pseudocode of the non-recursive (document order) traversal is shown in Table 5.1.

We recall that a traversal in DOM uses the tree navigational operations, such as `firstChild()`and `nextSibling()`, which are for recursive and non-recursive traversals. A non-recursive traversal requires, in addition, the `parent()` operation.

DOM provides direct support for the tree navigation operations in the `TreeWalker` class, and in addition, has other navigational operations to traverse the tree. We come back to this in Chapter 7. In order to understand the performance of the tree traversals (in our context) using the tree navigation operations, we summarise in Table 5.2 the total number of calls required to traverse a tree recursively and non-recursively. The traversals applied are in document-order and in reverse document-order.

Table 5.1 - Pseudocode for the non-recursive document-order traversal of a tree T.

Non-recursive document-order traversal:

```
Traverse(T){
  current:=root(T)
  direction:=DOWN
  while(current!=NULL){

    switch(direction){
      case DOWN:
           /* Code to process the node's associated
information */

           if(firstChild(current)!=NULL){
             current:= firstChild(current)
           }else
             direction:=HORIZONTAL

      case HORIZONTAL:
           if(nextSibling(current)!=NULL){
               current:= nextSibling(current)
               direction:=DOWN
           }else
               direction:=UP

      case UP:
             current:=parent(current)
             direction:=HORIZONTAL
    }
  }
}
```

Table 5.2 - Count of navigational operations called in the Traversals: document-order (DFO) that is recursive (Rec) or non-recursive (Non-rec), also reverse DFO that is recursive or non-recursive. n is the count of nodes in the tree, and t is the count of non-leaf nodes.

Navigation operation	Traversal			
	DFO		Reverse DFO	
	Rec	Non-rec	Rec.	Non-rec
`firstChild`	n	n	-	-
`nextSibling`	n	n	-	-
`Parent`	-	t	-	t
`previousSibling`	-	-	n	n
`lastChild`	-	-	n	n

5.2.2 XML Document Characteristics

The tree structures of the XML documents we have gathered in our XML corpus, generally have a large count of leaf nodes. On average over all files, about $2/3$ of nodes were leaves, the minimum count of leaves was in `Orders.xml` which had $1/2$ of its nodes as leaves. The file with the most leaves was `xCRL.xml`, which had $4/5$ of its nodes as leaves. In addition, we observe the number of nodes that appear at the last (rightmost) child node of their parent is the same as the number of non-leaf nodes. On average about $1/3$ of the nodes in the tree are non-leaf nodes.

The average count of child nodes in the tree (the degree) of an XML document is clearly just under one. However, we observe in each XML document that there exists at least one node with a large degree relative to the file size. For example, the file `Partsupp.xml` with 96,004 tree nodes, has a node with degree of over 2^{14}. In addition, `XPATH.xml` with 2,522,572 tree nodes, has a node with degree of over 2^{15}.

5.2.3 Requirements

We now derive the set of requirements for the succinct tree structure representation to efficiently represent the XML tree and support its operations.

Requirement 1. As mentioned in Section 5.2.1, associated information is stored for each node in the tree, this we represent using an array of length n. The associated data of the ith node in the tree is stored at index i in the array. The succinct tree representations do not directly support the access to the array of associated information, for the reason that nodes are numbered from 1 to $2n$. Using an array of size $2n$ would mean doubling the space usage. Hence, we have:

R1: Number nodes from 1 to n.

Requirement 2. Locating and retrieving XML data in the DOM document relies upon the navigation operations. Since DOM implementations use a large amount of space, some implementations reduce space costs by not supporting all navigation operations rapidly. However, this is often inappropriate, as DOM is used in a variety of applications with differing traversal patterns, as the following example shows.

The C++ DOM implementation called *Centerpoint* DOM [12] uses less space than Xerces by storing a pointer to the parent, first child and next sibling nodes but not to the last child or previous sibling. However, this comes at a cost of speed for certain operations. The `lastChild()` and `previousSibling()` operations in this implementation are slow. The `lastChild()` operation requires a traversal across all the children. Likewise the `previousSibling()` operation at the ith child requires a traversal across $i - 1$ children; one goes to the parent node and passes through the first child and next sibling nodes until we reach the previous sibling node. In a traversal, at a node with d children, this process takes $O(d^2)$ time in all, which can be very slow if this is a node with large degree (as observed in the example in Section 5.2.2, XML documents have node(s) with large degrees). At the cost of the space usage, Xerces explicitly includes the previous-sibling pointer at each node and includes the last-sibling pointer as a previous-sibling pointer of the nodes first-child, which greatly improves the speed. Hence, we have:

R2: All navigational operations must be fast.

Requirement 3. Traversals are common in DOM. In a document-order traversal of a tree we require n calls to the `firstChild()` and `nextSibling()` operations each (see Table 5.2). For example, calling the `firstChild()` operation in the `LOUDS` representation using the one's based numbering (see Section 4.2.4) requires calls to both `RANK` and `SELECT` and is therefore relatively expensive. Also, `firstChild()` remains equally expensive even if `firstChild()` is called at a leaf, where the answer is null. Given that leaf nodes in XML documents appear $1/2$ to $2/3$ of the time, a fast check for leaf nodes would avoid these operation calls. For the parentheses representation the `nextSibling()` operation is slow, as it requires a call to `FINDCLOSE` operation, but for $1/3$ to $1/2$ the nodes, `nextSibling()` returns null. Hence, we have:

R3.1 Detect quickly if a given node is a leaf.

R3.2 Detect first and last-child nodes quickly.

Having formulated the requirements `R1-R3` we now summarise how well the existing succinct tree representations support these requirements in Table 5.3. We abbreviate, from now on, the ones-based and the zeros-based numbering for the `LOUDS` representation as `LOUDS1` and `LOUDS0`, respectively. We refer to the parentheses and the binary tree representations as `PAREN` and `BT`, respectively.

Table 5.3 shows that the tree representations given in Chapter 4 do not directly support all requirements. In detail:

- `LOUDS`, `PAREN` and `BT` number nodes from 1 to $2n$. To map this to numbers from 1 to n we would need to support `RANK` on the bit-string that represents the tree. For example in Figure 4.8 an opening (closing) parenthesis is denoted by **1**(**0**); node i which is numbered 8 in the 1 to $2n$ numbering is mapped to the 1 to n numbering by $\texttt{RANK}_1(8) = 6$. While `BT` and `LOUDS` anyway need to support `RANK` on the bit-string that represents the tree, up to now there is no need for `PAREN` to support `RANK` (the space bound of Table 4.3 does not

include space for RANK). Thus, for BT/LOUDS there is a time cost to number nodes from 1 to n, and for PAREN there is a space and time cost.

- For R2, we observe that LOUDS and PAREN navigation operations are all $O(1)$ time. For BT the `parent()` and `lastChild()` operations are rather slower. For example, calling the `parent()` operation from the ith sibling requires i SELECT calls, and the `lastChild()` operation requires d RANK calls where d is the degree of the node. However R2 is definitely satisfied by PAREN and LOUDS1/0, and partially by BT (unlike the example of Centerpoint XML, traversals still take $O(d)$ time in the BT representation).
- For R3 (leaf node detection), in LOUDS1, nodes are represented by the **1**s in the degree sequence of their parent node. We require a call of RANK and SELECT to locate the node's own degree sequence, and only then we can detect whether the node is a leaf node. BT requires only a RANK call, and for LOUDS0 and PAREN, the detection of a leaf node is fast (requiring just the check of a single bit). The detection of the first child node in LOUDS1 and PAREN are fast (requiring the check of a single bit), however for LOUDS0, we require a call of RANK and SELECT, and for BT we require a RANK call. The detection of the last child node in PAREN is slow because we require a call to the FINDCLOSE operation. For LOUDS0 and LOUDS1 we require a call of RANK and SELECT, and for BT we require a RANK call.

Table 5.3 – Comparison of the succinct tree representations to support the requirements; we give the operation calls per node. d is a node degree.

Requirement	LOUDS1	LOUDS0	PAREN	BT
R1	RANK	RANK	RANK	RANK
R2	✓	✓	✓	PARENT requires d ×SELECT, LASTCHILD requires d ×RANK
R3.1	RANK and SELECT	✓	✓	RANK
R3.2	✓	RANK and SELECT	FINDCLOSE (FIRST-CHILD is fast)	RANK

5.3 Double Numbering

To address requirement R1, for each representation we number the nodes from 1 to n in the order that they are numbered. For example in Figure 4.7, for LOUDS1 the first node (which is node a) is at position 1, the second node (which is node b) is at position 3, the third node (which is node c) is at position 4, and so on, until the last node, which is the last to appear in level-order. For PAREN this means numbering nodes in document order (pre-order), for LOUDS1 and LOUDS0 this means numbering nodes in level-order, and BT has the node numbering that is neither document-order or level-order. For $i = 1, \ldots, n$, we let $\varphi(i)$ denote the position of the ith node in the bit-string representation of the tree: $\varphi(i)$ is the number of node i as described in Chapter 4.

It is important to maintain the association between i and $\varphi(i)$ for fast navigation, as the tree navigation operations are implemented by operations that use $\varphi(i)$. We apply double numbering to succinct tree representations as follows.

Our new approach, called *double numbering*, numbers the ith node as the pair $\langle i, \varphi(i) \rangle$. In Figure 4.7 according to LOUDS1, node c is indicated by $\langle 3, 4 \rangle$. Our key observation is that for LOUDS1, LOUDS0 and PAREN navigation in DOM with double numbering can be done with very little extra cost. The key property of navigation in DOM is that it always begins at the root (the double numbering of the root is usually

easy to compute). Then, each node is reached by one of the five navigation steps from a previously reached node.

Double numbering in LOUDS

In `LOUDS1` and `LOUDS0`, nodes are numbered from 1 to $2n$ in level-order. However, note that all operations in Table 4.4 use $\varphi(i)$. For example, in `LOUDS1`, if j is the `parent` of i, then $\varphi(j) = \texttt{SELECT}_1(\texttt{RANK}_0(\varphi(i)))$. Recall that:

(a) $\varphi(i)$ in `LOUDS0` equals the position of the ith **0** in the bit-string. Thus, $\varphi(i) = \texttt{SELECT}_0(i) + 1$ and $\texttt{RANK}_0(\varphi(i)) - 1 = i$.

(b) $\varphi(i)$ in `LOUDS1` equals the position of the ith **1** in the bit-string. Thus, $\varphi(i) = \texttt{SELECT}_1(i)$ and $\texttt{RANK}_1(\varphi(i)) = i$.

The key observation is:

Proposition 5.1. *Computing* $y = \texttt{SELECT}_i(x)$, *for* $i = 0$ *or* 1, *also computes* $\texttt{RANK}_0(y)$ *and* $\texttt{RANK}_1(y)$.

Proof. If $y = \texttt{SELECT}_0(x)$ then $\texttt{RANK}_0(y) = x$ and $\texttt{RANK}_1(y) = y - x$. $\texttt{SELECT}_1$ is similar. □

Double numbering in `LOUDS1` works as follows:

- We can calculate φ of the root node, which is $\varphi(1) = 1$.
- We consider the remaining navigation operations in turn:

  To compute `firstChild(<`i`,` $\varphi(i)$`>)`, we compute the position of the first child, i' as $\varphi(i') = \texttt{SELECT}_0(\texttt{RANK}_1(\varphi(i))) + 1$. Noting that $\texttt{RANK}_1(\varphi(i)) = i$, this simplifies to $\varphi(i') = \texttt{SELECT}_0(i) + 1$. Now we use Proposition 5.1 to observe that $\texttt{SELECT}_0(i)$ also essentially computes $\texttt{RANK}_1(\varphi(i')) = i'$.

Table 5.4 - Navigational operations for LOUDS1+ and LOUDS0+ (A is the LBS).

LOUDS1+	*LOUDS0+*

parent(<x, y>)
  if x= 1 then return NULL else
  let x′ := y - x
  let y′ := SELECT1(A, x′)
  return(<x′, y′>)
	parent(<x,y>)
	  if x= 1 then return NULL else
	  let x′ := SELECT1(A, x) - x
	  let y′ := SELECT0(A, x′ + 1)
	  return(<x′, y′>)

firstChild(<x, y>)
  let y′:= SELECT0(A, x) + 1
  if(A[y′]=0) then return NULL
  else
  return <y′- x, y′>
	firstChild(<x, y>)
	  if (A[x-1]=0) then return NULL
	  else
	  let x′:= (SELECT0(A, x) + 1) - x
	  let y′:= SELECT0(A, x′ + 1)
	  return <x′, y′>

lastChild(<x, y>)
  let y′:= SELECT0(A, x+1) - 1
  if(A[y′]=0) then return NULL
  else
  return <y′- x, y′>
	lastChild(<x, y>)
	  if (A[x-1]=0) then NULL
	  else
	  let x′:= y –(x + 1)
	  let y′:= SELECT0(A, x′ + 1)
	  return <x′, y′>

nextSibling(<x, y>)
  if A[y+1] = 0 then return NULL
  else
  return <x + 1, y + 1>
	nextSibling(<x, y>)
	  if A[SELECT1(A, x)+1]=0 then return NULL
	  else
	  let y′:= SELECT0(A, x + 2)
	  return <x + 1, y′>

previousSibling(<x, y>)
  if A[y-1] = 0 then return NULL
  else
  return <x - 1, y - 1>
	previousSibling(<x, y>)
	  if A[SELECT1(A, x)-1]=0 then return NULL
	  else
	  let y′:= SELECT0(A, x + 1)
	  return <x - 1,y′>

- To compute parent(<i, $\varphi(i)$>), we compute the position of the parent, i' as $\varphi(i') =$ $\mathsf{SELECT}_1(\,\mathsf{RANK}_0(\varphi(i))\,)$. Noting that $\mathsf{RANK}_1(\varphi(i)) = i$, this simplifies to $\varphi(i') =$ $\mathsf{SELECT}_1(\varphi(i) - i)$. Now we use Proposition 5.1 to observe that $\mathsf{SELECT}_1(i)$ also essentially computes $\mathsf{RANK}_1(\varphi(i')) = i'$.

- The other navigation operations are similar or much simpler (i.e. `nextSibling()`). We show these (including the operations above) in Table 5.4.

Double numbering in `LOUDS0` works as follows:

- We can calculate φ of the root node, which is $\varphi(1) = 2$.
- Using double numbering, the navigation operations for `LOUDS0` in Table 4.4 are simplified as shown in Table 5.4.

We call the `LOUDS1` and `LOUDS0` variants with double numbering support `LOUDS1+` and `LOUDS0+`, respectively. In support of the requirement `R1`, `LOUDS1+` and `LOUDS0+` are now faster because we avoid the `RANK` calls in the tree navigation operations.

Double numbering in Parentheses

In `PAREN` nodes are numbered from 1 to n in depth-first or document order. Recall that $\varphi(i)$ is the position of the ith opening parenthesis in the bit-string. For example, in Table 5.5 the seventh open parenthesis, which represents node seven, is at position twelve in the bit-string, so $\varphi(7) = 12$. We map the opening (closing) parentheses as **1**(**0**), forming a bit-string, and note that $\texttt{RANK}_1(\varphi(i)) = i$. Note that all operations in Table 4.5 use $\varphi(i)$. For example, if j is the parent of i, then the `parent` operation is computed as $\varphi(j) = \texttt{ENCLOSE}(\varphi(i))$. We now illustrate the use of double numbering through two examples:

- Again, suppose that j is the parent of i, and so $\varphi(j) = \texttt{ENCLOSE}(\varphi(i))$. The parentheses that lie in the bit-string between the open parentheses at $\varphi(j)$ and $\varphi(i)$ comprise the representations of the previous siblings of node i and their descendants. This means that there are an equal number of open and close parentheses between positions $\varphi(j)$ and $\varphi(i)$. Furthermore, the open parentheses that lie in between $\varphi(j)$ and $\varphi(i)$ correspond precisely to the

nodes that lie in between j and i in document order. Thus, $\texttt{RANK}_1(\varphi(j)) = i - (\varphi(i) - \varphi(j) + 1)/2 = j$.

- Another example is for the `nextSibling` operation, if j is the next-sibling of i, then the `nextSibling` operation is computed as $\varphi(j) = \texttt{FINDCLOSE}(\varphi(i)) + 1$:

  The parentheses that lie in the bit-string between the opening and closing parenthesis $\varphi(i)$ and $\varphi(j) - 1$ are the descendant nodes of i. This means that there are an equal number of open and close parentheses between $\varphi(i)$ and $\varphi(j) - 1$. As described for the `parent` operation the open parentheses that lie in between $\varphi(i)$ and $\varphi(j) - 1$ corresponds precisely to the nodes that lie in between j and i in document order. Thus, $\texttt{RANK}_1(\varphi(j)) = i + 1 + (\varphi(j) - \varphi(i) - 1)/2 = j$.

We modify all navigation operations to work with this "double numbering" in an analogous manner in Table 5.6. Observe that the root is node 1, and $\varphi(1) = 1$. Thus, we obtain the double numbering of the root directly, and the double numbering of any node reached from the root via navigation operations is correctly computed by induction. We call the PAREN with double numbering PAREN+.

Table 5.5 – Parentheses sequence representation with double numbering.

$\varphi(i)$	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
i	*1*	*2*		*3*	*4*		*5*	*6*				*7*		
	(	(	)	(	(	)	(	(	)	)	)	(	)	)
	1	1	0	1	1	0	1	1	0	0	0	1	0	0

Table 5.6 - Navigational operations for PAREN+ (double-numbering support). A is the parentheses bit-string and $A[i]$ retrieves the bit at position i in the bit-string A. Let an opening (closing) parenthesis be represented by **1**(**0**).

PAREN+	

parent(<x, y>)
  if x=1 return NULL
  let y' := ENCLOSE(A, y)
  let x' := x - (y - y' + 1)/2
  return (x', y')
	nextSibling(<x, y>)
	  let y'= FINDCLOSE(A, y)+1
	  if A[y']=1 then
	  let x':= x + (y' - y)/2
	  return <x', y'>
	  else return NULL

firstChild(<x, y>)
  if A[y+1]=0 then return NULL
  else
  return <x+1, y+1>
	previousSibling(<x, y>)
	  if A[y-1]=0
	  let y':= FINDOPEN(A, y-1)
	  let x':= x - (y - y')/2
	  return <x', y'>
	  else return NULL

lastChild(<x, y>)
  if A[y+1] = 1 then
  let y' := FINDCLOSE(TP, y)
  y':= FINDOPEN(y'-1)
  let x':= x + (y' - y + 1)/2
  return (x', y')
  else return NULL	

Double numbering in Binary tree representation

In support of requirement R1 we apply double numbering to BT, and number each node as $\langle i, \varphi(i)\rangle$. Hence, $\varphi(i)$ in BT is the position of the ith **1** in the bit-string and $\mathsf{RANK}_1(\varphi(i)) = i$. The numbering that results is, as noted before, neither in document-order or level-order. Note that all operations in Table 4.6 use $\varphi(i)$. For example, if j is the next sibling of i, then $\varphi(j) = 2 \times \mathsf{RANK}_1(\varphi(i))) + 1 = 2i + 1$.

Operations in BT begin with the operation RANK. In other words, once you are at a node, you cannot navigate away from the node without doing a RANK. Hence, we compute the RANK "in advance" since it is going to be needed. We called the BT representation with double numbering BT+. BT+ is never worse than BT. See Table 5.7 for the pseudocode of BT+ (with double numbering).

We now give an example where BT+ is faster than BT for a document-order traversal. At a node x that is a leaf node, we make several successive navigation operation calls even though the answer is null. This happens when we try to go to the first-child only to discover $x$it is a leaf. We then go to the next-sibling of x. In BT one would do the RANK operation for each navigation operation called from x. In BT+ the RANK operation is performed only once by the operation that reached x. Given that the double number of x is $<i, \varphi(i)>$, for the operations `firstChild()` and `nextSibling()` we check the bit at position $i * 2$ and $i * 2 + 1$, respectively. A **0**-bit indicates a leaf node, this is without calling RANK. In a document-order recursive traversal BT requires $2n$ RANK calls, where we have n nodes in the tree and we call `firstChild()` and `nextSibling()` operations at each node. In BT+ we require $n + t$ RANK calls, where t is the number of non-leaf nodes in the tree, where such nodes make up $1/3$ of nodes in the tree. Therefore we have a performance improvement in BT+.

Table 5.7 – Operations of the Binary Tree representations with double-numbering. (*A* is the LBS).

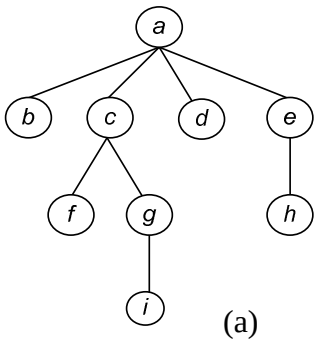
BT+	
parent(<x, y>) let y':= y, let x':= x while(y' mod 2 = 0) <x',y'>:= previousSibling(x',y') x':= y'/2 return <x', $SELECT_1$(A, x')>	nextSibling(<x, y>) let y':= 2*x+1 if A[y']=0 then NULL else return <$RANK_1$(A, y'), y'>
firstChild(<x, y>) let y' := 2*x if A[y'] = 0 then return NULL else return <$RANK_1$(y'), y'>	previousSibling(<x, y>) if (y mod 2)!=0 x':= y/2 y':= $SELECT_1$(A, x') return <x', y'> else return NULL

lastChild(<x, y>)
let y':= 2*x
if A[y'] = 0 then return NULL
else
let x':= $RANK_1$(A, y')
while(nextSibling(<x', y'>)
!=NULL)
<x', y'>:=nextSibling(<x',y'>)
return <x', y'>	

5.4 Optimising LOUDS further

5.4.1 Adding isLeaf bit-string

We make a further optimisation to the LOUDS1+ representation by including a bit-string of length *n*, which differentiates all leaf nodes from non-leaf nodes in the tree. The bit-string of length *n* is defined as follows: each bit represents a single node in the tree, in level-order; where the *i*th bit is set to **1** if the *i*th node in the tree is a leaf node, and to **0**, otherwise. This supports R2 and R3, at the cost of *n* extra bits overall. In a recursive document-order traversal of a tree the required *n* SELECT calls are now reduced to between 1/3 and 1/2. We refer to the LOUDS1+ representation with the isLeaf bit-string as LOUDS1++.



(a)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a		**b**	**c**	**d**	**e**			**f**	**g**			**h**			**i**			
1	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**

(b)

Vertex		*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
Runs0	1	0	1	0	1	0	1	0	0	1
Runs1		1	0	0	0	1	0	1	1	1

(c)

Figure 5.2 – (a) Ordinal tree. (b) LOUDS bit-string of tree in (a). (c) Equivalent partitioned bit-vector.

Table 5.8 – Operations of the partitioned representation. Bit-strings `Runs0` and `Runs1` defined in Section 5.4.2.

Partitioned representation (PLOUDS)	

`parent(x)`
`  let x’:=RANK⁻(Runs1,x)`
`  return SELECT₁(Runs0,x’)`
	`nextSibling(x)`
	`  if Runs1[x]=1 then NULL`
	`else`
	`  return x+1`

`firstChild(x)`
`  if isLeaf(x) then NULL`
`  else`
`  let x’:= RANK⁻(Runs0,x)`
`  return SELECT₁(x’)+1`
	`previousSibling(x)`
	`  if Runs1[x-1]=1 then NULL`
	`else`
	`  return x+1`

`lastChild(x)`
`  if isLeaf(x) then NULL`
`else`
`  let x’:= RANK⁻(Runs0,x)+1`
`  return SELECT₁(x’)+1`	

5.4.2 Partitioned Representation

We now describe a new representation that has the simplicity of `LOUDS1+` and also allows the check of a leaf node in $O(1)$ time. We address requirements `R1`, `R2` (fast navigation operations) and `R3` (indicate leaf and last-child nodes rapidly). The idea is to encode the runs of **0**s and **1**s in the `LOUDS` bit-string (`LBS`) in two separate bit-strings, which we will call `Runs0` and `Runs1`. Specifically, if there are runs of **0**s of length $l_1, l_2, \ldots, l_z$ in the `LBS`, then the bit-string `Runs0` is simply $\mathbf{0}^{l_1-1}\mathbf{1}\mathbf{0}^{l_2-1}\mathbf{1} \ldots \mathbf{0}^{l_z-1}\mathbf{1}$. `Runs1` is defined analogously (see example in Figure 5.2). Noting that the `LBS` begins with a **1** and ends with a **0**, it is clearly possible to reconstruct it from `Runs0` and `Runs1`. `PLOUDS` is simply `Runs0` and `Runs1`, each augmented with directories to support $\texttt{SELECT}_1$ and $\texttt{RANK}^-$ operations, where:

$\texttt{RANK}^-(x)$ returns the number of **1** bits strictly to the left of position x in the bit-vector. ($\texttt{RANK}^-(x) = \texttt{RANK}_1(x - 1)$ except when $x = 1$).

In Table 5.8, we show the navigation operations of `PLOUDS`. Observe that, some operations are now trivial:

- The check of a leaf node requires the check of the previous bit in the `Runs0` bit-string. For a node that appears at position x, it is a leaf if the bit at position $x - 1$ is a **0**. For example, in Figure 5.2 – (a) Ordinal tree. (b) LOUDS bit-string of tree in (a). (c) Equivalent partitioned bit-vector.
- , nodes b, d, f, h and i are leaf nodes by the criterion.
- The `nextSibling()` and `previousSibling()` operations in `PLOUDS` are as simple as they were in `LOUDS`. The `nextSibling()` operation is computed in the `Runs1` bit-string. For a node at position x we check the bit position x, if the bit is a **0** then we return $x + 1$, otherwise return null.

We now observe:

Proposition 5.2. `SELECT` *operations on the* `LBS` *can be simulated by a* $\texttt{SELECT}_1$ *and a* $\texttt{RANK}^-$ *on* `Runs0` *and* `Runs1`.

Proof. We claim that $\texttt{SELECT}_1(\texttt{LBS}, i) = \texttt{SELECT}_1(\texttt{Runs0}, \texttt{RANK}^-(\texttt{Runs1}, i))+i$. Note that $\texttt{RANK}^-(\texttt{Runs1}, i)$ equals the number of completed runs of **1**s before the run that i is in. There must be an equal number of completed runs of **0**s before i. The $\texttt{SELECT}_1$ on `Runs0` then gives the total length of these runs, which is then added to i to give the position of the i-th **1**. $\texttt{SELECT}_0(\texttt{LBS}, i)$ is similar.

Corollary 1. `PLOUDS` *supports the operations* `parent`, `firstChild()` *and* `lastChild()`.

Proof. We look at the implementation of these operations in `LOUDS1`. Due to double-numbering, these operations only have a single `SELECT` call, which can be simulated as in Proposition 5.2.

Proposition 5.3. *The number of* **1***s in* `Runs0` *and* `Runs1` *is equal to the number of non-leaf nodes in the input tree plus one.*

Proof. A run of **1**s in the `LBS` is a node of degree > 0, i.e. a non-leaf node (with the exception of the super-root). The number of **1**s in `Runs1` is the number of runs of **1**s in the `LOUDS` bit-string. The number of runs of **0**s in the `LBS` equals the number of runs of **1**s.

The main advantage of `PLOUDS` is that it requires just $\texttt{SELECT}_1$ and `RANK`, not $\texttt{SELECT}_0$. In addition, the number of **1**s in `Runs0` and `Runs1` is usually small. Therefore, we would normally expect the space usage of `PLOUDS` to be less than `LOUDS1+`. The disadvantage of `PLOUDS` is that to do a `SELECT` call we now must do both `RANK` and `SELECT` calls.

5.5 Comparison of tree representations

We have presented a partitioned version of Jacobson's [44] LOUDS representation, called PLOUDS. Although we will demonstrate experimentally that PLOUDS uses less space than LOUDS, this could be understood on a firmer theoretical basis. It would be interesting to see whether the partitioning idea generalises to other applications.

The idea of double-numbering and fast leaf node checking in PAREN+, LOUDS1+, LOUDS0+, LOUDS1++ and BT+ allows us to meet the requirements (Section 5.2.3) of representing an XML tree, where we gain good tree traversal performance competitive to if not better than standard DOM implementations.

In summary, we show in Table 5.9 the total number of calls to the RANK, SELECT and INSPECT operations in the document-order recursive traversal and non-recursive traversal upon a tree for the LOUDS and BT variants. The INSPECT operation is simply a check of a single bit, hence a memory access. As noted in Chapter 4, RANK is a little slower than a memory access and SELECT is 2.5 times slower than RANK. For LOUDS1, LOUDS0 and BT we require φ in addition to fulfil requirement R1, which we have omitted from the table. Even though, we see an overall improvement in the LOUDS1+, LOUDS0+, LOUDS1++, PLOUDS and BT+ representations because we avoid the call of the extra RANK operation, using in most cases the same space usage.

Further improvements we observe in LOUDS1++ are that the number of SELECT calls are reduced to t in the recursive traversal and to $2t$ in the non-recursive traversal, where t is the number of non-leaf nodes (usually $1/3$ of tree nodes). In BT+ the number of RANK calls have been reduced from $2n$ (that is in BT) to $n + t$.

PLOUDS makes $t \times$ SELECT and $t \times$ RANK calls in a non-recursive traversal. In comparison to the other tree representations (that support double numbering) they appear to be faster at first sight because they do not require both the RANK and SELECT operations in a non-recursive traversal. However in PLOUDS, we observe that there is a potential operational time gain for

Table 5.9 – Total number of RANK and SELECT calls for recursive and non-recursive document-order traversals. Comparison of LOUDS1, LOUDS0, LOUDS1+, LOUDS0+, LOUDS1++ and PLOUDS. n is # nodes and t is # non-leaf nodes in the tree. φ operation call for the tree representations is not included.

Tree Reps	Recursive traversal	Non-Recursive traversal
LOUDS1	$n \times$ SELECT, $n \times$ RANK, $2n \times$INSPECT	$(n+t) \times$ SELECT, $(n+t) \times$ RANK, $2n \times$ INSPECT
LOUDS0	$(2n+2t) \times$SELECT, $(n+2t) \times$RANK, $2n \times$ INSPECT	$(2n+4t) \times$ SELECT, $(n+4t) \times$ RANK, $2n \times$ INSPECT
LOUDS1+	$n \times$ SELECT, $2n \times$ INSPECT	$(n+t) \times$ SELECT, $2n \times$ INSPECT
LOUDS0+	$(2n+t) \times$ SELECT, $2n \times$ INSPECT	$(2n+3t) \times$ SELECT, $2n \times$ INSPECT
LOUDS1++	$t \times$ SELECT, $2n \times$ INSPECT	$2t \times$ SELECT, $2n \times$ INSPECT
PLOUDS	$t \times$ SELECT, $t \times$ RANK, $2n \times$ INSPECT	$2t \times$ SELECT, $2t \times$ RANK, $2n \times$ INSPECT
BT	$2n \times$ RANK, $2n \times$INSPECT	$2n \times$ RANK, $2n \times$INSPECT, $t \times$ SELECT
BT+	$(n+t) \times$ RANK, $2n \times$INSPECT	$2t \times$ RANK, $2n \times$INSPECT, $t \times$ SELECT

computing RANK and SELECT on a bit-string that is half the size (i.e. Runs0 and Runs1 compared to LBS) on larger bit-strings.

The disadvantage of BT and BT+ is that for the operations lastChild() and parent() it can be much slower than the other operations. For example, given we have a node with child degree count h, the cost to go to the last child of a node is $O(h)$ time, making only RANK calls.

Likewise going to the parent node, we make only SELECT calls, in $O(h)$ time, in the worst case when we are at the last-child node. Given that SELECT is a factor 2.5 slower than the RANK operation, we predict that a non-recursive traversal for BT is slow, for documents with large degree nodes.

5.6 Experimental Evaluation

In this section we provide a comparative experimental analysis of the succinct tree representations.

5.6.1 Setup

To test our data structures we obtain ordinal trees from the following six XML files in our XML corpus (Chapter 3): `Mondial-3.0.xml`, `Orders.xml`, `Nasa.xml`, `XPATH.xml`, `Treebank_e.xml` and `XCDNA.xml`. We also tested the data structures on randomly generated XML files. These were obtained by using the algorithm described in [49] to generate random parentheses strings. A random parentheses string was converted to an XML file by replacing the opening and closing parentheses of non-leaf nodes by opening and closing tags. The parentheses for leaf nodes were replaced with short text nodes.

The six XML files selected show a range of file sizes and tree structure. In all cases, the type of each node (`element`, `text` node, etc) was stored as a 4-bit value in an accompanying array; we call this the *node-type* array.

The basic setup of our experiments is outlined in Appendix A. We used the Xerces DOM parser to construct the tree structure bit-strings of the XML documents. The tree representations were tested on the Intel-P4 and Sun-UltraSparc machines. The experiments were to traverse the trees and to count the total number of nodes of a particular type by accessing the node-type array. We tested with four different types of traversal, breadth-first order (BFO), and depth-first order (DFO), which is done both recursively and non-recursively (using the algorithm from Table 5.1), and the reverse recursive depth-first order (RDO), where we first visit the last child at each node and then each of its previous siblings in turn.

We compare the running times of five variants of the `LOUDS` data structures, the two `BT` representations and the two `PAREN` representations. For `RANK` and `SELECT` we use the CJ and KNKP bit-vector implementations detailed in Section 4.2.1, where in the CJ

bit-vector we use the parameters $B = 64$ and $s = 32$, and in the KNKP bit-vector we use the parameters $B = 64$ and $SB = 256$.

5.6.2 Space Usage

In Table 5.10, we summarise the space usage per node of the tree structure representations. We state the space usage per node required for long gaps and the clump array in the CJ and KNKP bit-vectors, respectively. For our XML documents, the `BT` representation has no long gaps for CJ and no clumps in the clump array for KNKP because the pattern of **1**s in the bit-string are densely distributed. Our running time comparisons of the succinct tree representations must be done based on similar space usage; therefore, we use parameters that produce like-for-like space usage. `LOUDS1++` uses the same tree representation as `LOUDS1`, plus a single bit per node for the `isLeaf` bit-string representation. We observe that `PLOUDS` generally uses less space than the other `LOUDS` data structures. When implemented using KNKP its space usage is competitive with the `PAREN`.

We observe that when using CJ in `PLOUDS` a low number of long gaps usually relates to high number of leaves in the document. For example, `Mondial-3.0.xml` has 78% of leaf nodes in its tree structure, and the long gap on average is 0.34 bits per node.

Compared to `Orders.xml`, which has 50% leaf nodes, the long gaps are much higher at 1.6 bits per node. However, for the randomly generated files, that have 50% leaves and have a negligible amount of long gaps, their space usage result per node is consistently higher than most files. For the `BT` representation, we have a consistent space usage for all files, as there are not many long gaps, and we observe in the bit-string the **1**s and **0**s are usually equally distributed, i.e. there are no large runs of only zeros or ones.

Table 5.10 – Space Usage of tree reps. Columns are test file, number of nodes, % leaf node and total space usage of tree representations given per node. `LOUDS0` and `LOUDS1` use the same space usage therefore call them `LOUDS`: For `PLOUDS`, `LOUDS` space per node for the clump data structure using KNKP; space per node to support long gaps using CJ. For `PAREN`: space per node, cf Table 4.3. For negligible we use NEG.

			PLOUDS				LOUDS				BT		PAREN
			KNKP		CJ		KNKP		CJ		CJ	KNKP	
File	Nodes	% leaf	total	clump DS	total	long gap	total	clump DS	total	long gap			
Mondial	57,372	78	3.12	0.07	3.84	0.34	5.11	0.11	5.65	0.55	4.05	4.55	3.73
Orders	300,003	50	3.78	0.03	5.64	1.6	5.07	0.07	5.10	NEG	4.05	4.55	3.73
Nasa	1,425,535	67	3.37	0.05	4.27	0.57	5.09	0.09	5.42	0.33	4.05	4.55	3.73
XPATH	2,522,571	67	3.37	0.04	3.99	0.27	5.08	0.08	5.63	0.53	4.05	4.55	3.73
treebank_e	7,312,612	67	3.37	0.04	3.77	0.06	5.08	0.08	5.10	0.01	4.05	4.55	3.73
XCDNA	25,221,153	67	3.35	0.02	3.80	0.08	5.11	0.11	5.09	0.38	4.05	4.55	3.73
R65K	62,501	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	4.05	4.55	3.73
R250K	250,001	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	4.05	4.55	3.73
R1M	1,000,001	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	4.05	4.55	3.73
R4M	4,000,001	50	3.79	0.04	4.05	NEG	5.08	0.08	5.09	NEG	4.05	4.55	3.73
R16M	16,000,001	50	3.81	0.04	4.05	NEG	5.08	0.08	5.10	NEG	4.05	4.55	3.73

CJ bit-vector analysis on Orders.xml

We observe that `PLOUDS` using CJ uses more space than `LOUDS1+` for the file `Orders.xml`. We now explain this unusual behaviour. In essence, what we observed is that the number of long gaps in the partitioned bit-strings (`PLOUDS`) is relatively large, even though there are no long gaps in the original `LBS`. We first examine the tree structure of `Orders.xml`, which is shown in Figure 5.3.

The root node has degree 30,001. At the next level in the tree these child nodes are arranged in the following pattern: a leaf node followed by a node with nine children.

The nine child nodes themselves each have a single child node. This pattern is repeated 15,000 times. Therefore, the `LOUDS` bit-string representing the tree structure is defined as follows: we insert the bits **10 10** for the super root and the root node. We then insert 30,001 **1**s and a **0** bit representing the node with 30,001 child nodes. For the next level we insert repeatedly (15,000 times) the bit pattern **0 1111111110**, representing a leaf node followed by a node with nine children over the 30,001 nodes. For the next level we insert the bit pattern **10**, repeated 135,000 times, representing a single child node for each node in the groups of nine child nodes.

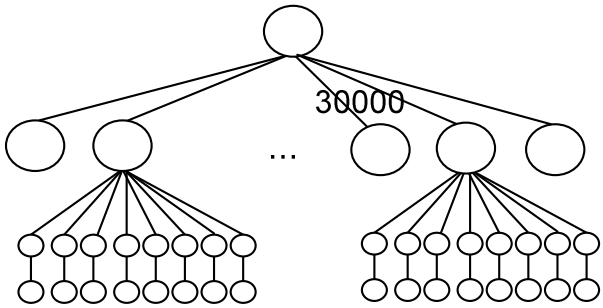
We use the parameters $s = 32$ and $L_G = 256$ for the CJ implementation given in Section 4.2.1. Recall that in the $\texttt{SELECT}_1$ data structure we store explicitly every sth **1**, and a long gap appears where the difference between the is-th and $(i+1)s$-st **1** bit is more than L_G bits. If a long gap appears, we store explicitly the positions of all the **1** bits from is to $(i+1)s - 1$ in an array.

For the `LOUDS1+` bit-string the $\texttt{SELECT}_0$ data structure has one long gap between the s-th and $2s$-th **0** bit, but no more, for the reason that in the second level of the tree representation we have two **0** bits for every nine **1**s. In 256 bits there are approximately forty-six **0**s, therefore no long gaps. For $\texttt{SELECT}_1$ there are no long gaps for the reason that for every 256 bits there are approximately 210 **1**s.

However for the `PLOUDS` bit-strings, which supports $\texttt{SELECT}_1$ our interest is with the `Runs1` bit-string. The first long gap appears between the first and second s-th **1**, where there are 30,000 **0**s. After this we have the pattern of eight **0**s followed by a single **1**, repeated 15,000 times. Within these number of bits the is and $(i+1)s$ **1** bit appear every 288 bits, which is bigger than the long gap size ($L_G = 256$), therefore the number of long gaps (LG) is $135{,}000/288 = 468$.

Therefore as in Table 5.10, we confirm the long gap space usage for `Orders.xml` is as follows:

LGs per node $= (32 \times \#LG \times s)/n = (32 \times 468 \times 32)/300{,}003 = 1.60$ (2 d.p.).



$1010\ 1...1_{30000}\ 01111111110...01111111110_{15000}\ 0\ 101010101010101010...101010101010101010_{135000}\ 0...0_{135000}$

Figure 5.3 – Top: Ordinal tree structure of `Orders.xml`. Bottom: Bit-string representation of Orders.xml (subscripts indicate repetition of sub-string sequence).

In other words, in the partitioned representation of the `LOUDS` bit-string we have found an example where the space usage has increased.

5.6.3 Running Time

The performance measure we report for our succinct data structures is the slowdown relative to Xerces based on the same type of traversal. We first determine which bit-vector to use. Table 5.11 gives the slowdown relative to Xerces of `PLOUDS` using the KNKP and using the CJ for a DFO traversal on a Pentium 4. The CJ based `PLOUDS` outperforms the KNKP based data structure. We saw the same relative performance for `LOUDS1+`, `LOUDS0+` and `BT` traversals. This is not too surprising since the KNKP was designed for sparse bit-vectors; the bit-vectors here are dense. In the remaining experimental results the `LOUDS` and the `BT` data structures use CJ.

Table 5.11 - CJ and KNKP speed comparison

	Mondial	Order	Nasa	XPATH	Treebank	R62K	R250K	R1M	R4M	R16M
KNKP	1.08	2.72	2.01	1.75	2.40	2.11	2.15	2.17	2.24	2.46
CJ	0.96	1.3	1.68	1.42	1.96	1.72	1.78	1.79	1.83	1.98

Table 5.12 – Performance evaluation on Intel-P4. Coloumns are: Test file, slowdown relative to the Xerces for recursive and non-recursive depth-first order (DFO) traversals for `LOUDS1` (L1), `LOUDS1+` (L1+), `LOUDS1++` (L1++) `LOUDS0+` (L0+), `PLOUDS` (PL), BinaryTree (BT), BinaryTree+ (BT+) all using CJ bit-vector and for `PAREN` (Par) and `PAREN+` (Par+). Fastest data structure for each set is in bold font.

File	Intel-P4																	
	DFO recursive									DFO non-recursive								
	L1	L1+	L0+	L1++	PL	Par	Par+	BT	BT+	L1	L1+	L0+	L1++	PL	Par	Par+	BT	BT+
Mondial	1.87	1.24	1.87	**0.75**	0.96	1.28	1.04	1.52	1.08	1.85	1.22	1.92	**0.79**	1.05	1.39	1.11	1.92	1.53
Orders	2.23	1.41	2.29	**1.07**	1.30	1.48	1.24	1.66	1.15	2.29	1.54	2.55	**1.24**	1.62	1.66	1.41	2.10	1.59
Nasa	2.97	2.04	3.19	**1.32**	1.68	2.08	1.67	2.25	1.59	3.09	2.13	3.38	**1.52**	1.95	2.44	2.00	2.96	2.39
XPATH	2.53	1.69	2.66	**1.13**	1.42	1.81	1.44	2.01	1.41	3.02	2.00	3.25	**1.44**	1.87	2.33	1.87	3.01	2.44
treebank	3.29	2.25	3.60	**1.53**	1.96	2.49	2.03	2.37	1.70	3.55	2.49	3.80	**1.87**	2.33	3.27	2.80	3.17	2.55
R65K	2.52	1.82	2.85	**1.45**	1.72	2.19	1.89	1.96	1.53	3.04	2.24	3.28	**1.86**	2.20	2.95	2.67	2.74	2.27
R250K	2.55	1.83	2.93	**1.49**	1.78	2.18	1.90	1.98	1.54	3.09	2.30	3.32	**1.93**	2.24	2.88	2.59	2.75	2.27
R1M	2.56	1.84	2.98	**1.50**	1.79	2.39	2.12	1.98	1.55	3.18	2.36	3.41	**2.01**	2.30	3.28	2.98	2.82	2.34
R4M	2.63	1.88	3.09	**1.56**	1.83	2.41	2.13	2.01	**1.56**	3.30	2.45	3.50	**2.09**	2.35	3.25	2.96	2.88	2.39
R16M	2.76	1.98	3.27	1.67	1.98	2.53	2.20	2.04	**1.58**	3.47	2.63	3.75	**2.31**	2.61	3.23	2.95	2.93	2.45

In Table 5.12, Table 5.13 and Table 5.14, we summarise the performance of the data structures for the DFO and BFO traversals. For the BFO traversal, it required the *queue* data structure of the C++ STL library. The storing of DOM nodes in the queue resulted in some overhead, therefore the DOM in the BFO traversal could not fit XCDNA.xml into the internal memory of the Intel-P4 machine. The data structures are based on the succinct tree representations described in Chapter 4: `LOUDS1`, `LOUDS0`, `PAREN` and `BT`. We observe that double numbering improves the running time operations of the tree representations: Over all files, the `LOUDS1+` variant was a factor of 1.44 better than `LOUDS1` on average. For `PLOUDS` there is an improvement of a factor of 1.60 better than `LOUDS1` on average.

Table 5.13 - Performance evaluation for DFO on Sun-UltraSparc. The setup is the same as in Table 5.12.

File	Sun-UltraSparc																	
	DFO recursive									DFO non-recursive								
	L1	L1+	L0+	L1++	PL	Par	Par+	BT	BT+	L1	L1+	L0+	L1++	PL	Par	Par+	BT	BT+
Mondial	1.52	1.03	1.64	**0.68**	0.92	1.17	0.97	1.43	0.91	1.77	1.18	1.88	**0.81**	1.15	1.40	1.17	2.05	1.52
Orders	1.16	0.75	1.37	**0.62**	0.78	0.97	0.80	1.09	0.69	1.48	0.99	1.73	**0.85**	1.12	1.16	1.01	1.50	1.11
Nasa	1.22	0.81	1.40	**0.57**	0.75	0.98	0.81	1.12	0.71	1.49	1.00	1.65	**0.74**	1.02	1.28	1.08	1.63	1.22
XPATH	1.19	0.77	1.35	**0.55**	0.76	0.96	0.78	1.11	0.70	1.46	0.96	1.61	**0.73**	1.03	1.17	1.00	1.62	1.20
treebank	1.23	0.83	1.38	**0.58**	0.79	1.06	0.89	1.09	0.70	1.48	1.01	1.64	**0.75**	1.05	1.52	1.35	1.59	1.20
XCDNA	1.20	0.78	1.37	**0.57**	0.78	0.96	0.79	1.11	0.71	1.45	0.97	1.62	**0.73**	1.03	1.17	0.99	1.61	1.21
R65K	2.94	2.07	3.35	**1.58**	2.07	2.70	2.38	2.68	1.72	3.72	2.62	4.25	**2.13**	2.83	3.83	3.43	3.64	2.72
R250K	1.29	0.90	1.47	**0.69**	0.91	1.17	1.03	1.13	0.76	1.59	1.11	1.82	**0.91**	1.21	1.64	1.47	1.56	1.17
R1M	1.26	0.88	1.43	**0.68**	0.89	1.31	1.15	1.10	0.74	1.56	1.09	1.78	**0.89**	1.20	1.82	1.66	1.52	1.15
R4M	1.27	0.88	1.45	**0.69**	0.90	1.33	1.18	1.11	0.75	1.56	1.10	1.79	**0.91**	1.21	1.82	1.66	1.53	1.16
R16M	1.26	0.89	1.45	**0.69**	0.90	1.36	1.20	1.11	0.74	1.58	1.10	1.79	**0.93**	1.21	1.79	1.64	1.52	1.15

Table 5.14 - Performance evaluation for BFO on Intel-P4 and Sun-UltraSparc. The setup is the same as in Table 5.12.

File	Intel-P4									Sun-UltraSparc								
	BFO									BFO								
	L1	L1+	L0+	L1++	PL	Par	Par+	BT	BT+	L1	L1+	L0+	L1++	PL	Par	Par+	BT	BT+
Mondial	0.86	0.57	0.80	**0.38**	0.47	0.65	0.55	0.80	0.56	1.13	0.74	1.12	**0.53**	0.65	0.83	0.73	1.09	0.69
Orders	1.26	0.86	1.27	**0.71**	0.83	0.99	0.85	1.05	**0.71**	0.78	0.51	0.85	**0.43**	0.53	0.59	0.54	0.74	0.49
Nasa	1.19	0.81	1.18	**0.57**	0.71	0.94	0.79	1.07	0.78	0.84	0.56	0.88	**0.41**	0.52	0.64	0.57	0.81	0.53
XPATH	1.31	0.87	1.28	**0.62**	0.77	1.03	0.86	1.18	0.85	0.82	0.54	0.86	**0.41**	0.51	0.64	0.57	0.79	0.50
treebank_e	1.12	0.77	1.11	**0.54**	0.67	1.01	0.87	0.98	0.71	0.73	0.50	0.78	**0.36**	0.46	0.66	0.60	0.73	0.48
XCDNA										0.69	0.47	0.71	**0.36**	0.44	0.54	0.49	0.67	0.44
R65K	1.52	1.09	1.61	**0.87**	1.03	1.78	1.56	1.39	1.01	3.00	1.98	3.34	**1.61**	2.04	2.81	2.48	2.78	1.75
R250K	1.48	1.05	1.62	**0.86**	1.00	1.73	1.53	1.37	1.00	1.27	0.83	1.41	**0.68**	0.86	1.23	1.06	1.19	0.74
R1M	1.49	1.05	1.61	**0.86**	1.01	1.95	1.75	1.39	1.01	1.24	0.82	1.37	**0.65**	0.83	1.38	1.22	1.19	0.74
R4M	1.15	0.81	1.24	**0.66**	0.78	1.55	1.39	1.09	0.79	1.09	0.72	1.21	**0.58**	0.74	1.27	1.14	1.07	0.68
R16M	0.96	0.69	1.07	**0.57**	0.66	1.43	1.27	1.02	0.74	0.73	0.49	0.82	**0.40**	0.50	0.97	0.88	0.75	0.50

For PAREN and BT, double numbering speeded up the traversal on average by a factor of 1.17 and 1.37, respectively. Comparing the performance of the basic tree navigation operations we observe that LOUDS1++ is the fastest tree representation.

Note that LOUDS1++ uses n bits more than the other succinct tree representations for the isLeaf bit-string. Therefore, we are required to add one extra bit per node to the LOUDS1 space usage in Table 5.10 to represent LOUDS1++. For both the recursive and non-recursive traversals LOUDS1++ is the fastest. Excluding LOUDS1++ (which requires an extra bit per node) we observe BT+ is the fastest and PAREN+ is almost as fast. For the non-recursive traversal the BT(+) representations suffer on the parent() operation: given that we are at the last-child node we have to navigate through all previous-sibling nodes before we get to the parent node. Over the entire set of files, we observe that PLOUDS was competitive if not better than the other tree representations, if we were to consider the trade-off between the space usage and the running time performance.

5.7 Technical ideas summary

We studied several succinct tree representations and optimised them for DOM support. These optimised representations number the nodes of an n-node tree with integers from 1 to n, and (recall that previous representations numbered nodes non-consecutively with numbers from 1 to $2n$), and have fast implementations for testing whether a node is a leaf. The main new idea introduced was double numbering, and the partitioned representation for the LOUDS bit-string. The idea of the partitioned representation has been applied to bit-strings by [37].

Based on our requirements set out in Section 5.2.3 we chose PAREN+ as the tree structure representation in our DOM application, which we present in Chapter 7. PAREN+ supports the requirements R1-R3, in that it numbers nodes from 1 to n, navigational operations are fast, and the indication of first child nodes is fast. In addition, the direct support of document-order numbering of nodes in PAREN+ is an

advantage and is required in DOM. Such support in the LOUDS variants, which number nodes in level-order, would require additional data structures. The running time of PAREN+ is not as good as some of the other tree representations, however with a lower space usage on average, PAREN+ is still competitive.

Finally, if we were to extend our aims of supporting DOM as in an XML processor application, we would support structural joins and the additional traversal operations such as the *following* and *preceding* operations. These are already supported in PAREN+, but not in the other tree representations.

Chapter 6
Representing Textual Data

In this chapter, we present strategies to efficiently store and access textual data contained in XML documents. There is an abundance of textual data in XML documents: for example, among our test files, `Treebank_e.xml` has 67% of its nodes in the document tree as text nodes. Indeed, in Chapter 3 we saw that textual data made up between 50% and 80% of our documents.

We model the problem of storing textual data in XML documents as follows. Given n (non-empty) strings $s_1, \ldots, s_n$, we wish to store the strings in a data structure so that we can support the operation of returning the ith string, when given the integer i by the "user" (in our case, the "user" will be the SDOM application described in Chapter 7). The strings are numbered consecutively by the "user" and the data structure does not have the freedom to re-order the strings. Our basic approach is to concatenate the strings, and store offsets into the concatenated strings that help us to get the ith string. In order to do this in a space-efficient manner, we introduce the *prefix sums* problem: given a (static) sequence of positive integers $\boldsymbol{x}$, we wish to support the operation: $\mathtt{SUM}(\boldsymbol{x},i)$, which is for the offset i. This problem was described in Section 4.2.5, where a succinct data structure was implemented. We investigate the practical performance of this data structure as well as alternatives.

The chapter is organised as follows. We begin by giving more details of our basic approach, and explaining how the prefix sums problem becomes relevant. Then we give solutions to the prefix sums problem. Next, we describe details of the storage of textual data, and finally we give an empirical evaluation of our approach. Parts of this chapter were published in [23].

6.1 Overview

As noted above, we are given n non-empty strings $s_1, \ldots, s_n$. From an implementation perspective, we assume that the last character of each string is a string terminating character, and for $i = 1, \ldots, n$ we let t_i be the string obtained by removing the string terminating character from s_i. We let $x_i = |s_i|$, $x'_i = |t_i|$, $\boldsymbol{x} = (x_1, \ldots, x_n)$, $\boldsymbol{x'} =$

$(x'_1, \ldots, x'_n)$. Since no string is empty, we have that $x_i \geq 2$, and $x_i' \geq 1$, for all i. By $W(\boldsymbol{x})$ we denote $\sum_{i=1}^{|\boldsymbol{x}|} x_i$.

We consider two basic ways of representing the strings. First, we consider uncompressed text. In this case, we concatenate $s_1, \ldots, s_n$ into a single character array, called A. In addition, we store the numbers $x_1, \ldots, x_n$ in a prefix sums data structure. In order to access the ith string, we:

- compute $j = \texttt{SUM}(\boldsymbol{x}, i)$
- return a pointer to $A[j]$

(the second step is possible since we keep the string terminating character). Next, we consider compressed text. In this case, we let T denote the string which is the concatenation of $t_1, \ldots, t_n$. We store T in a data structure that keeps T in a compressed form, but is rapidly able to answer `subString(`j,k`)` queries, which returns a string that equals the substring of T from positions j to k. In this case, we access the ith string as follows:

- compute $j = \texttt{SUM}(\boldsymbol{x}', i)$
- compute $k = \texttt{SUM}(\boldsymbol{x}', i + 1)$
- return `subString(`j,$k - 1$`)`.

We now explain why we choose this approach. It is important to remember that there are many textual nodes and that the average length of textual data is relatively small (particularly due to the whitespace text nodes). For example, excluding the null terminating character, on average over all files the individual text nodes were approximately 11 characters in length (the average text node length over all files ranges from 6 characters to 23 characters).

We now consider the two naive approaches to the string offset storage problem. We could, for example store the "offset" values $\texttt{SUM}(\boldsymbol{x},1)$, $\texttt{SUM}(\boldsymbol{x},2), \ldots, \texttt{SUM}(\boldsymbol{x},n)$ in an array of integers. This uses up 32 or 64 bits per string. Given that strings are only about

88 bits long on average, the space used by the offsets is a significant portion of the textual data. Since the textual data in turn is a significant portion of the XML document, the offsets would be a significant part of the eventual representation. The other approach is to store each string as a C++ string. This has the disadvantage that somewhere we must store a pointer to this string, which again takes 32 to 64 bits. In addition, using a number of small (dynamically-allocated) chunks of memory would probably lead to memory fragmentation, and hence to even greater memory usage. If the text is stored compressed, then, assuming say a typical 3:1 compression ratio, the compressed size of a text node would be on average just 30 bits, and a naive storage of the offsets/pointers will be even less feasible.

In what follows, therefore, we want to focus on two problems:

- How to store the lengths of the strings in a space-efficient way, so that the `SUM` operation can be supported efficiently.
- How to store the string T in a compressed manner, so as to support the `subString` operation rapidly.

6.2 Prefix Sums Problem

To address the problem of storing the string lengths we engineer several prefix-sums solutions based on two compressibility measures:

(a) The succinct space bound given in Proposition 4.5 is $B(m,n) = \lceil \log_2 \binom{m-1}{n-1} \rceil$ bits, which applies to any sequence $\boldsymbol{x}$ of size n whose elements add up to m;

(b) *Data-aware* measures, which depend on the values in $\boldsymbol{x}$, and can be lower than the succinct bound for some sequences. Appropriate data-aware measures have been studied extensively in the information retrieval (IR) community [76].

We demonstrate a close connection between the data-aware measure that is the best in practice for an important IR application and the succinct bound. As (a) is already defined we now define (b).

6.2.1 Data aware Measures

The data-aware measures are based upon self-delimiting encodings of the individual values x_i; these have been studied extensively in the context of IR applications [76]. The data-aware encodings are designed so that small integers have smaller codes than larger values. This is suitable for our application: as mentioned before the average length of text nodes is relatively small. There are two main families which we discuss; the first is represented by the *Golomb* and *Rice* codes, and the second by the δ and γ codes.

Golomb code

Given an integer parameter $b \geq 1$, the Golomb code of an integer $x > 0$, denoted $G(b,x)$, is obtained by writing the number $q = \lfloor (x-1)/b \rfloor$ in unary (i.e. as $\mathbf{1}^q\mathbf{0}$), followed by $r = x - qb - 1$ in a binary encoding using either $\lfloor \lg b \rfloor$ or $\lceil \lg b \rceil$ bits. If $0 \leq r < 2^{\lceil \lg b \rceil} - b$, then use $\lfloor \lg b \rfloor$ bits to encode r. If $2^{\lceil \lg b \rceil} - b \leq r < b$, then use $\lceil \lg b \rceil$ bits to encode r. If b is a power of two, we can encode each value of r with $\lceil \lg b \rceil$ bits. A Rice code is a Golomb code where b is a power of two.

In Figure 6.1, we show as a binary tree the 'prefix-free' encodings of r when $b = 3$ and $b = 6$. For example, if $x = 9$ and $b = 3$, then $q = 2$ and $r = 2$ because $9 - 1 = 2 * 3 + 2$; so the encoding $G(3,9) = \mathbf{110\ 11}$. We observe that $r = 2$ is encoded as **11** as shown in Figure 6.1 (a).

The first data-aware measure is $GOLOMB(b, \boldsymbol{x}) = \sum_{i=1}^{n} |G(b, x_i)|$, where $|\sigma|$ denotes the length (in bits) of the string σ. In other words, $GOLOMB$ measures how well $\boldsymbol{x}$ compresses by coding each x_i using a Golomb code.

Gamma (γ) code

The γ-code of an integer $x > 0$, $\gamma(x)$, is obtained by writing $\lfloor \lg x \rfloor + 1$ in unary, followed by the value $x - 2^{\lfloor \lg x \rfloor}$ in a field of $\lfloor \lg x \rfloor$ bits, e.g, $\gamma(6)$ = **001 10**. Clearly $|\gamma(x)| = 2\lfloor \lg x \rfloor + 1$ bits. The second data-aware measure of the compressibility of $\boldsymbol{x}$ is $\Gamma(\boldsymbol{x}) = \sum_{i=1}^{n} |\gamma(x_i)|$.

Figure 6.1 – Binary encoding for r values in (a) when $b = 3$ and in (b) when $b = 6$.

Delta (δ) code

The δ-code of an integer $x > 0$, $\delta(x)$, is obtained by writing $\lfloor \lg x \rfloor + 1$ using the γ-code, followed by $x - 2^{\lfloor \lg x \rfloor}$ in a field of $\lfloor \lg x \rfloor$ bits; e.g., $\delta(33)$ = **001 10 00001**. The final data-aware measure of the compressibility of $\boldsymbol{x}$ is $\Delta(\boldsymbol{x}) = \sum_{i=1}^{n} |\delta(x_i)|$.

By the concavity of the log function, it follows that the Γ and Δ measures are maximised when all the x_i's are equal. This gives the following observation:

$$\Gamma(\boldsymbol{x}) = \Delta(\boldsymbol{x}) = O(n \log(m/n)) \tag{6.1}$$

We observe Γ and Δ are never much worse than the succinct bound: recall that $B(m,n) = \left\lceil \log_2 \binom{m-1}{n-1} \right\rceil$ (Chapter 4). Conversely, if the values in $\boldsymbol{x}$ are unevenly distributed, then the Γ and Δ measures are reduced, and may be much less than the succinct bound. This, together with the simple observation that $\Delta(\boldsymbol{x})$ can never exceed $\Gamma(\boldsymbol{x})$ by more than $\theta(n)$ bits, makes the Δ measure asymptotically attractive. However, extensive experiments show in [76] that the Δ, Γ and $GOLOMB$ measures of a sequence arising from a particular IR application were broadly similar, and Γ is often less than Δ; $GOLOMB$ with the choice $b = \lceil (m \ln 2)/n \rceil$ has generally been observed to be the smallest for a particular IR application.

6.2.2 Related Work

There is a large body of related work, which includes:

- Data structures achieving within $O(n)$ bits of the succinct bound were given by many authors (e.g. [27], [37]); the optimal bound was achieved in [57].

- In recent work [41], a new data-aware measure, gap was proposed, where $gap(x) = \sum_{i=1}^{n} \lceil \lg x_i \rceil$. The authors considered, in addition to SUM, a variety of operations including predecessor operations on the set represented by the prefix sums of $\boldsymbol{x}$. Unfortunately, gap is not an achievable measure, i.e. there exist sequences that provably cannot be compressed to gap.
- In [42], Gupta et al. carried out an experimental evaluation on the data-aware data structures. We note that some of the ideas in this chapter are similar to those developed independently in [42].
- Other work [59] implies that $O(1)$-time SELECT is possible if space $gap(x) + o(m)$ bits is used, but the second term can be much larger than gap.
- As our main focus is on the practical performance of these data structures, we look more closely at [42]. In [42], the focus is on RANK queries, while ours is on SELECT, and our data sets are different. Contrary to [42], we uphold the conclusions of [76] that Golomb coding (and hence the succinct bound) are superior to the other gap-aware measures. Although it would be meaningless to draw direct conclusions regarding running times between our work and theirs, in our implementations, only the trivial gap-aware data structures came even close to the succinct data structure.

6.2.3 Succinct Representations and Golomb Codes

The succinct solution given in 4.2.5 is represented using $n \lg(m/n) + O(n)$ bits. We observe that $GOLOMB$ is closely related to the succinct bound when the Golomb parameter b is chosen to be $\Theta(m/n)$. We now show the connection between the succinct and Golomb bounds:

Proposition 6.1. *Let* $c \geq 1/2$ *be any constant, and let* $\boldsymbol{x}$ *be a sequence with* $W(\boldsymbol{x}) = m$ *and* $|\boldsymbol{x}| = n$ *and suppose that* $cm/n > 1$. *Then, taking* $b = \lceil cm/n \rceil$, $|GOLOMB(b, \boldsymbol{x}) - B(m,n)| = O(n)$.

Proof. We use the following inequalities:

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil \text{ for any } x. \tag{6.2}$$

$$\lg(cm/n) \leq \lg b \leq \lg(cm/n) + 1. \tag{6.3}$$

$$n/2c \leq m/b \leq n/c. \tag{6.4}$$

(6.2) follows from the definition. For (6.3) the first inequality is obvious. For the second inequality, take both sides to the power of 2 and restate it as:

$$\lceil cm/n \rceil \leq 2cm/n$$

Since $cm/n > 1$, therefore, $b = \lceil cm/n \rceil < cm/n + 1 \leq 2cm/n$, which shows (6.3). The reasoning for (6.4) is similar to that for (6.3).

We now prove the main proposition. Firstly,

$$GOLOMB(b, x) \leq \sum_{i=1}^{n} (\lfloor (x_i - 1)/b \rfloor + 1 + \lceil \lg b \rceil)$$

The RHS can be greater than the LHS since some x_is will have their binary part coded using $\lfloor \lg b \rfloor$ bits. Now note that:

$$\sum_{i=1}^{n} (\lfloor (x_i - 1)/b \rfloor + 1 + \lceil \lg b \rceil) \leq \sum_{i=1}^{n} (x_i/b + 1 + \lg b + 1)$$

Since $W(x) = m$, we get:

$GOLOMB(b, x) \leq m/b + n(2 + \lg b) \leq n/c + n(3 + \lg(cm/n)) = n \lg(m/n) + (3 + 1/c + \lg c)n$. Thus, $GOLOMB(b, x) - B(m, n) = O(n)$.

Now note that $GOLOMB(b, x) \geq \sum_{i=1}^{n} (\lfloor (x_i - 1)/b \rfloor + 1 + \lfloor \lg b \rfloor)$, and:

$$\begin{aligned} &\sum_{i=1}^{n} (\lfloor (x_i - 1)/b \rfloor + 1 + \lfloor \lg b \rfloor) \\ &\qquad \geq \sum_{i=1}^{n} \left(\frac{x_i - 1}{b} + 1 + \lg b - 2 \right) \\ &\qquad \geq n \lg \left(\frac{cm}{n} \right) + \frac{m - n}{b} - n \geq n \lg(cm/n) + \frac{n}{2c} - 2n \end{aligned}$$

Thus, $B(m, n) - GOLOMB(b, x) = O(n)$.

We conclude that $|B(m,n) - GOLOMB(b,x)| = O(n)$.

Remark – When $c = \ln 2$, we see that $B(m,n) - 2.69n \leq GOLOMB \leq B(m,n) + 3.53n$.

6.2.4 Gamma and Delta Codes

We now consider the compression criteria based on the γ and δ codes. We assume that, given $\gamma(x)$ or $\delta(x)$, we can decode x in $O(1)$ time, provided the code fits in $O(1)$ machine words.

We define the operation ACCESS($\boldsymbol{x}$, i) as returning x_i. We now show:

Proposition 6.2. *A sequence $\boldsymbol{x}$ with $|\boldsymbol{x}| = n$ and $W(\boldsymbol{x}) = m$ can be stored so as to support ACCESS in $O(1)$ time while using $\Gamma(\boldsymbol{x}) + O(n \lg\lg(m/n))$ bits.*

Proof. We form the bit-string σ by concatenating $\gamma(x_1), \ldots, \gamma(x_n)$. We create the sequence $\boldsymbol{o}$, where $o_i = |\gamma(x_i)|$ and store it in the data structure of Theorem 4.5. Evaluating SUM($\boldsymbol{o}$, $i-1$) and SUM($\boldsymbol{o}$, i) gives the start and end points of $\gamma(x_i)$ in $O(1)$ time, and x_i is decoded in $O(1)$ further time. Since $W(\boldsymbol{o}) = \Gamma(x) = O(n \lg(m/n))$, the space used to represent $\boldsymbol{o}$ is $O(n \log\log(m/n))$ bits.

Remark – An obvious optimisation is to remove the unary parts altogether from σ, since they are encoded in $\boldsymbol{o}$, and this is what we do in practice.

A simple prefix-sum data structure is obtained as follows (Lemma 6.1 is similar to one in [42]):

Lemma 6.1. *Given a sequence $\boldsymbol{x}$ with $|x| = n$ and $W(\boldsymbol{x}) = m$, we can store it using $\Gamma(\boldsymbol{x}) + O(n \lg\lg(m/n))$ bits and support SUM in $O(\lg n)$ time.*

Proof. For convenience of description, assume that n is a power of two. Consider a complete binary tree T with n leaves, with the values x_i stored in left-to-right order at the leaves. At each internal node, we store the sum of its two children. We then list the values at the nodes in the tree in level-order (starting from the root), except that for

every internal node, we only enumerate its smaller child. This produces a new sequence of length n, which we denote as *tree*($\boldsymbol{x}$).

For example, in Figure 6.2, $\boldsymbol{x}$ = (3, 4, 6, 2, 6, 5, 3, 3) and *tree(**x**)* = (32, 15, 7, 6, 3, 2, 5, 3). Given *tree(**x**)* and an additional $n - 1$ bits that specify for each internal node, which of the two children was enumerated, we can easily reconstruct all values in nodes on, or adjacent to, any root-to-leaf path, which suffices to answer SUM queries.
The key observation is:

$$\Gamma(tree(\boldsymbol{x})) \leq \Gamma(\boldsymbol{x}) + 2n - 2. \tag{6.5}$$

To prove this, consider a procedure to fill in the values in T bottom up. First, it stores in each node at level 1 the sum of its two children. Let the values stored at level 1 be $y_1, \ldots, y_{n/2}$, and note that $y_i = x_{2i-1} + x_{2i} \leq 2 \times \max\{x_{2i-1}, x_{2i}\}$, so $|\gamma(y_i)| \leq |\gamma(max\{x_{2i-1}, x_{2i}\})| + 2$. If we now delete $max\{x_{2i-1}, x_{2i}\}$ for all i, the total lengths of the γ-codes of the y_is, together with the remaining $n/2$ values at the leaves, is n bits more than $\Gamma(\boldsymbol{x})$. Since the construction of $tree(\boldsymbol{x})$ now essentially recurses on $y_1, \ldots, y_{n/2}$, equation (6.5) follows.

If we store $tree(\boldsymbol{x})$ in the data structure of Proposition 6.2, we have $O(1)$ time access to each of the values in $tree(\boldsymbol{x})$. Together with the bit-string that indicates which nodes are deleted, decoding all the values from a root-to-leaf path, and hence computing SUM, takes $O(\log n)$ time. □

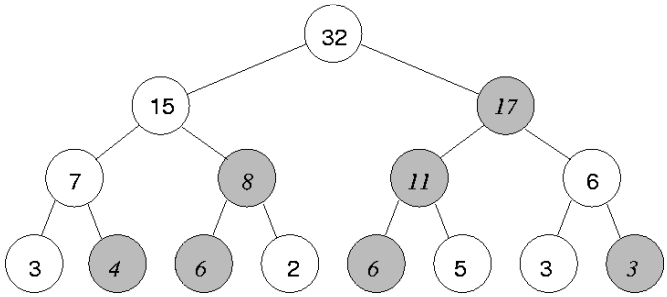


Figure 6.2 - Formation of *tree*($\boldsymbol{x}$); shaded nodes are removed from the output.

6.2.5 Implementation Details

We implemented three prefix-sum data structures: the succinct data structure and two simple

data structures that store γ-codes. A preliminary implementation of Lemma 6.1 was also made. We now discuss some implementation details, assuming a 32-bit machine.

Compacted bit-String data structure

All prefix sum data structures rely on a data structure that stores a bit-string of length n in an integer array of size $\lceil n/32 \rceil$, and supports the following operations:

- `subBitString(`i,j`)`: extracts the substring from positions i to j from the bit-string. We assume that the extracted substring fits into a single word, i.e. $j - i + 1 \leq 32$.
- `getAlignedWord(`i`)`: this returns the substring from positions i to $i + 31$ from the bit-string.

Since these operations are used frequently, the code is carefully optimised. For example, assuming 32-bit integers, we need to compute $\lfloor i/32 \rfloor$ and i mod 32, to determine the integer containing the ith bit, and the offset of the ith bit within the integer. The former is computed using shifts, and the latter by AND with a pre-computed mask. The main reason for separating the functions `subBitString` and `getAlignedWord` is that the former requires a branch statement to separate the cases where the substring is all in one word and where the substring is split across two words, and the latter does not. Since branch mis-predictions are quite expensive, the latter should be faster. In addition, the former also needs to perform division and modulo operations on two indices, while the latter does this only on one index, and has very simple code:

```
getAlignedWord(i)
  let idiv = i >> 5
  imod = i & 0x1F;
  first = seq_BS[idiv];
  second = seq_BS[idiv + 1];
  return(first << imod + second >> (32-imod));
```

Decoding gamma-codes

The low-level representation of the γ-codes is designed to decode quickly. Specifically, we represent $\gamma(x)$ with the unary representation of $\lfloor \lg x \rfloor$ stored reversed in the lower-order bits, and the 'binary' part stored in the next higher-order bits. For example, $\gamma(11) = \mathbf{0001\ 011}$ is stored as **011 1000**. Now suppose that we are given an integer z that contains a γ-code in the lower-order bits, e.g., $z = \ldots \mathbf{011\ 1000}$. We compute $z \,\text{AND}(-z)$ to leave only the 'unary part' of $\gamma(x)$ in the lower-order bits (this is a standard trick).

For example:

z	$\ldots$ **0111000**
$-z$	$\ldots$ **1001000**
$z \,\text{AND}(-z)$	$\ldots$ **0001000**

We then compute the index of the 1 in this word by a lookup table; suppose that the result is k. We then shift z right k positions, and mask out the last $k-1$ bits to obtain the binary part of x.

Succinct prefix sums data structure

This is implemented as described in Section 4.2.5. The lower-order bits are concatenated to form a bit-string, which is then stored in the compacted bit-string data structure described above. In addition, if the text is stored uncompressed, then we have to store a sequence $\boldsymbol{x}$ such that each entry x_i in the sequence is greater than or equal to 2. In this case, we instead store the values $x_i' = x_i - 1$; this reduces the sum of the

values to be stored, and hence potentially the space usage. Note that $\texttt{SUM}(\boldsymbol{x}, i) = \texttt{SUM}(\boldsymbol{x}', i) + i$.

γ-code data structures

We have implemented two simple data structures for the prefix sum problem that target the Γ space bound; these we refer to as *explicit-γ* and *succinct-γ*. For the sequence $\boldsymbol{x} = x_1 \ldots x_n$, we form the bit-string $\boldsymbol{\sigma}$ by concatenating $\gamma(x_1), \ldots, \gamma(x_n)$, and storing it using the compacted bit-string data structure. In addition, these data structures use a parameter $G > 0$. In the explicit-γ data structure we store every G-th prefix sum, as well as offsets into $\boldsymbol{\sigma}$ to the start of the G-th γ-code, explicitly (using 32 bits); in the succinct-γ data structure, these prefix sums and offsets are stored using the succinct data structure. To compute $\texttt{SUM}(\boldsymbol{x}, i)$, we access the appropriate G-th prefix sum, and the corresponding offset, and sequentially scan $\boldsymbol{\sigma}$ from this offset using `getAlignedWord`, which we use to minimise calls to the sub-bitstring operation. The `getAlignedWord` operation retrieves 32 bits of data containing γ-codes. We observe that on average over all files the γ-codes are 5.43 bits long (see Table 6.2), therefore these 32 bits contain on average five γ-codes from the bit-string $\boldsymbol{\sigma}$. These can be decoded "for free" before we need to retrieve more γ-codes.

γ-tree data structures

Finally, we implemented the data structure of Lemma 6.1. Here, we made the following change: we always delete the right child of a node in the tree of prefix sums, rather than the larger child. The advantages are that we do not need to store the additional n bits to indicate which child was deleted, and it also speeds up the navigation down the tree. Let $tree^*(\boldsymbol{x})$ be the sequence obtained by always deleting the right child. We then encode each integer of the $tree^*(\boldsymbol{x})$ sequence using the γ-code. For the γ-codes the unary and binary parts are stored separately, the unary parts are concatenated into a bit-string, which supports $\texttt{SELECT}_1$. The binary parts are concatenated and stored using the compacted bit-string data structure. To retrieve the ith unary value we compute:

```
start = SELECT1(i)
end = SELECT1(i+1)−1
unary = start-end.
```

The binary part is retrieved by the operation `subBitstring(start-i+1, end-(i+1))`. The $\mathtt{SUM}(\boldsymbol{x}, i)$ operation is computed as follows: we decode values from a root-to-leaf path (the leaf where i is stored). To go to a right child node we compute its value by subtracting the left child value from its parent node value. The answer is at the leaf node. If the leaf node is a left child then we go to the node's parent and get the value of its previous-sibling, if there is no previous-sibling node then we traverse up the tree and get the value of the current node's previous-sibling node, and so on if the node's previous-sibling does not exist. If the leaf node is a right child then the answer is the sum of all left child values before the current right child node, in document-order.

6.3 Textual data

We now discuss two alternatives to represent the string T in a compressed manner. These representations support the `subString()` operation discussed in the introduction to this chapter.

FM-Index

The first is using the FM-Index [29], which stores T in a compressed form (it applies a BZip-related substring operation without fully decompressing T). In addition, it also supports the following operation:

- Given a non-empty substring P, count the number of occurrences of P in T, or locate one occurrence of P in T, in time dependent only on the size of P (the null terminating character for each individual string must be left in T if the search functionality is required).

Blocked BZip2

In the other representation, we divide T into blocks of B characters, and compress each block using BZip2 [11]. When the individual string t_i needs to be retrieved, the block(s)

containing it are decompressed. Once a block is decompressed, it is stored in a text block cache of K uncompressed blocks. Then to compute `subString(`j,k`)`we are required to copy the required characters from position j to k in the cache into a new string. However, subsequent accesses to a cached block do not require decompression so long as a block is not evicted from the text block cache because the cache is full (we use a FIFO replacement mechanism). We use $K = 4$ and $B = 16$KB.

The code of BZip2 and FM-Index has been retrieved from [11] and [31], respectively.

6.4 Experimental Evaluation

In this section, we experimentally evaluate the prefix-sums data structures and text data structures. We first describe the basic setup of our experiments. We then present experiments on the prefix-sums data structures, beginning with the evaluation of the compressibility of the test data under certain measures. We then evaluate the space usage and (running time) performance of the prefix-sums implementations. Finally, we evaluate the compression performance of the text data structures.

6.4.1 Basic Setup

The basic setup of our experiments is outlined in Appendix A. The test machines used are the Intel-P4 and Sun-UltraSparc. Our test data are derived from the sixteen files in our XML corpus (see Chapter 3). We use Xerces DOM to extract the data values from these XML files.

In Table 6.1, we show for each file the space usage cost of the textual data and the offsets, assuming the offsets are uncompressed. In addition, we observe that the average cost of the naive representation of the offset values over all files was 40% of the uncompressed textual data size.

6.4.2 Prefix-sums experiments

For the succinct prefix sums data structure we compare three bit-vector implementations (detailed in Section 4.2.1). For the CJ and CNEW implementations, we choose the following parameters: $B = 64$, $s = 32$ and $L_G = 256$. In addition, we include the bit-

vector implementation KNKP (see Section 4.2.1) with the parameters $SB = 256$ and $B = 64$.

Compressibility and Space Usage

Table 6.2 summarises the measures of compressibility, in terms of bits per prefix sum value, using the encoding schemes and using a succinct representation. We omit from the prefix sums experiments the results on the attribute value lengths, as they are less common in our XML documents and provide similar compressibility and running time results to the results on `text` node lengths. In the Golomb codes we use $b = \lceil 0.69m/n \rceil$.

Although gap gives the best measure of compressibility, it cannot be decoded without additional data structures. We see that in practice Γ and Δ are greater than $GOLOMB$ in eleven of our test XML files, and for half our files $GOLOMB$ is at least 29% less than either Γ or Δ; this is in line with many results on compressing inverted lists [76] (however, [42] give examples where Γ and Δ are smallest). Comparing $GOLOMB$ and the succinct bound, in all the cases in Table 6.2 we see that $B - 0.25n \leq GOLOMB \leq B + 0.33n$, which is much closer than what Proposition 6.1 suggested.

Recall that $\Gamma(tree(\boldsymbol{x})) \leq \Gamma(\boldsymbol{x}) + 2|\boldsymbol{x}| - 2$ (Eq. 6.5 in Lemma 6.1). In the best case, $\Gamma(tree^*(\boldsymbol{x})) = \Gamma(tree(\boldsymbol{x})) = \Gamma(\boldsymbol{x})$. In the worst case, we claim that $\Gamma(tree^*(\boldsymbol{x})) \geq 2\,\Gamma(x)$. Consider the sequence $\boldsymbol{x} = a, 1, a, 1, \ldots..$, where a is a value such that $|\Gamma(a+1)| = |\Gamma(a)| + 2$ (for example, $a = 7$ is such a value). For this sequence, $\Gamma(\boldsymbol{x}) = n/2 + (n/2)|\Gamma(a)|$. We now construct just the first level of the tree, by summing pairs of leaves and deleting the ones.

The resulting sequence of numbers (say z) contains $n/2$ a's and $n/2$ $(a+1)$'s and $\Gamma(\boldsymbol{x}) = (n/2)\,|\Gamma(a)| + (n/2)|\Gamma(a+1)| = n + n\,|\Gamma(a)| = 2\Gamma(\boldsymbol{x})$. Since continuing the construction of the tree only increases the size of the numbers, it is clear that $\Gamma(tree^*(\boldsymbol{x})) \geq 2\Gamma(\boldsymbol{x})$.

Table 6.1 – Naive representation of offset values. n' denotes the number of text and attribute nodes (K represents a thousand and M represents a million), cost of storing data values uncompressed, and of a naive representation for the offset values, respectively.

Files	File size	n'	**Uncompressed text**	**Naive offsets**
Elts	128KB	4832	39KB	19KB
w3c1	224KB	12.8K	152KB	50KB
w3c2	200KB	11.6K	136KB	45KB
UNSPC-2.04	1,740KB	58.9K	531KB	230KB
Mondial-3.0	1,081KB	82.4K	688KB	322KB
Partsupp	2,253KB	48.0K	1,088KB	188KB
Orders	5,243KB	150.0K	1,488KB	586KB
xCRL	8,708KB	229.5K	3,079KB	896KB
Votable2	15,927KB	841.7K	5,376KB	3,288KB
Nasa	24,371KB	1.0M	15,530KB	3,927KB
Lineitem	32,326KB	1.0M	6,152KB	3,996KB
XPATH	50,995KB	1.7M	13,314KB	6,569KB
Treebank_e	83,968KB	4.9M	58,757KB	19,043KB
SwissProt	112,129KB	7.6M	49,795KB	29,774KB
DBLP	130,724KB	7.2M	73,077KB	28,111KB
XCDNA	607,881KB	16.8M	261,953KB	65,680KB

Table 6.2 shows $(\Gamma(tree^*(x)) - \Gamma(x))/|x|$ for our sequences. It is interesting to note that this does not go below 1.96, which gives some insight into the distribution of values. Neither does it go above 2.92 nor is typically much smaller showing that always deleting the right child (which is simpler and faster) does not waste space in practice[4].

We now consider the space usage of our data structures. We calculate the space used, in bits per input sequence value, and also the difference between the space used by the data structures and the corresponding compressibility measure (we refer to this as *wasted space*). Table 6.2 summarises the space usage of the various data structures where parameters have been selected such that the wasted space is roughly the same.

[4] Recall that Γ(tree(x)) does not include the $n - 1$ bits needed for decoding x.

Table 6.2 – Compression performance. Compressibility measures: $gap(x)$, $\Delta(\boldsymbol{x})$, $\Gamma(\boldsymbol{x})$, $GOLOMB(b, \boldsymbol{x})$ as (GOL), $B(m, n)$ as (SUC). Tree overhead: $\left(\Gamma\left(\boldsymbol{tree}^*(\boldsymbol{x})\right) - \Gamma(\boldsymbol{x})\right)/|\boldsymbol{x}|$. Space usage: Total space in bits (spac) and wasted space in bits (wast) per prefix value using the succinct prefix sum data structure and using the explicit-γ and succinct-γ data structures. Data structure parameters for explicit-γ and succinct-γ were selected such that wasted space is roughly equal.

File	Text nodes	Compressibility measures					tree ovhd	Space usage					
								Succinct		Explicit-γ		Succinct-γ	
		Gap	Δ	Γ	GOL	SUC		Spac	wast	spac	wast	spac	wast
Elts	3896	2.90	5.53	5.36	3.79	4.04	1.99	7.10	3.07	7.36	2.00	7.89	2.53
w3c1	7689	2.05	4.37	4.34	5.37	5.34	2.54	8.12	2.78	6.34	2.00	7.00	2.65
w3c2	7102	2.00	4.30	4.28	5.40	5.38	2.88	8.19	2.81	6.28	2.00	6.83	2.55
Mondial-3.0	34.9K	3.55	6.87	6.56	4.76	4.90	2.04	7.77	2.88	8.56	2.00	9.13	2.57
UNSPSC-2	39.3K	3.83	7.16	6.71	4.97	4.89	2.42	7.61	2.71	8.71	2.00	9.36	2.65
Partsupp	48.0K	2.53	5.24	5.23	6.14	5.95	1.99	9.36	3.41	7.23	2.00	7.94	2.71
Orders	150.0K	2.56	5.31	4.99	4.83	4.71	2.17	7.67	2.96	6.99	2.00	7.53	2.54
xCRL	155.6K	3.84	7.75	6.96	4.98	4.98	2.03	7.62	2.64	8.96	2.00	9.62	2.65
votable2	841.0K	2.56	5.67	5.28	4.22	4.03	1.97	7.26	3.23	7.28	2.00	7.85	2.57
Nasa	948.9K	3.04	5.58	5.45	5.64	5.39	2.40	8.15	2.76	7.45	2.00	8.11	2.66
Lineitem	1.0M	2.16	4.94	4.55	3.95	3.94	2.10	7.08	3.14	6.55	2.00	7.08	2.52
XPATH	1.7M	3.26	6.41	5.81	4.15	4.37	2.27	7.26	2.89	7.81	2.00	8.38	2.57
Treebank_e	4.9M	3.69	7.08	6.72	4.94	5.01	2.15	7.65	2.64	8.72	2.00	9.25	2.54
SwissProt	5.4M	2.38	5.38	4.64	4.31	4.10	2.25	7.50	3.40	6.64	2.00	7.14	2.50
DBLP	6.8M	1.78	3.88	3.89	5.00	4.67	2.92	8.25	3.58	5.89	2.00	6.45	2.56
XCDNA	16.8M	3.33	6.62	6.18	5.61	5.39	2.29	7.87	2.48	8.18	2.00	8.77	2.59

For the explicit-γ and succinct-γ data structures we used $G = 32$ and $G = 8$, respectively. For these values the space usage in the γ-codes data structures is comparable to the succinct data structure.

Running time performance

The performance measure we report is time in μs for determining a random prefix sum value. Each data point reported is the median of ten runs in which we perform eight million random `SUM` operations. We have again selected parameters such that the wasted space in each data structure is about the same.

Table 6.3 summarises the performance of the data structures. The fastest runtime for each file on the Intel-P4 and on the Sun-UltraSparc machines is shown in bold. The table shows the performance of the succinct data structure using the three different bit-vectors. We see that the performance of the CNEW bit-vector is similar to CJ and better than KNKP. The table also shows the performance of the explicit-γ and succinct-γ data structures using the bit-vector. We see that the explicit-γ data structure out-performs the succinct-γ data structure when the space usage is roughly the same. The performance results are preliminary, but we note that the succinct prefix sums data structure almost always outperforms both the γ-codes data structures. We observed that a single γ-decode is about twenty times faster than a SELECT operation, so improvements in the bit-vector would make succinct-γ more competitive.

We also performed some limited experiments on the relative performance of the data structure of Lemma 6.1. We compared the time for SUM($\boldsymbol{x}$, i), when $\boldsymbol{x}$ is stored as in Lemma 6.1 (but always deleting the right child), versus in a simple bit-string. At $|\boldsymbol{x}| =$ 64, 128, 256, 512 and 1024, the times in μs for the tree were 0.767, 0.91, 1.12, 1.28 and 1.5, and for the bit-string were 0.411, 0.81, 1.57, 3.08 and 6.03. We are not comparing 'like for like', as the tree uses more space. Even then we find that the (logarithmic) tree data structure does not outperform the (linear) bit-string until $|\boldsymbol{x}| > 128$. Unfortunately, the 2 bits per number (at least) wasted by the tree data structure means that explicit-γ with $G = 64$ would be less wasteful in space than the tree, and also faster.

The tree requires two SELECT operations at each node visited, so an approach to speeding-up the tree data structure would be to increase the arity and thereby reduce the height of the tree.

Table 6.3 – Speed evaluation on Intel-P4 and Sun-UltraSparc. Test file, number of text nodes, time in μs to determine a prefix sum value for succinct data structures using CJ, KNKP and CNEW. Time to determine a prefix sum for explicit-γ (Exp) and for succinct-γ (Succ) data structure, both of which are based on the new bit-vector. The best runtime for each file on each platform is in bold.

File	Text nodes	Machine 1 - Pentium 4					Machine 2 - Sun UltraSparc-III				
		Succinct prefix sums			γ-code		Succinct prefix sums			γ-code	
		CJ	KNKP	CNEW	Exp	Succ	CJ	KNKP	CNEW	Exp	Succ
Elts	3896	0.073	0.121	**0.069**	0.189	0.196	0.151	0.222	**0.138**	0.284	0.389
w3c1	7689	0.083	0.134	**0.082**	0.211	0.212	0.158	0.230	**0.138**	0.279	0.389
w3c2	7102	0.082	0.133	**0.079**	0.209	0.216	0.158	0.229	**0.140**	0.279	0.390
Mondial-3.0	34.9K	0.084	0.134	**0.083**	0.211	0.214	0.176	0.240	**0.146**	0.293	0.399
UNSPSC-2	39.3K	0.087	0.136	**0.082**	0.204	0.209	0.176	0.244	**0.149**	0.290	0.401
Partsupp	48.0K	0.084	0.133	**0.080**	0.204	0.213	0.168	0.240	**0.150**	0.284	0.396
Orders	150.0K	**0.081**	0.134	**0.081**	0.197	0.209	0.199	0.270	**0.176**	0.298	0.408
xCRL	155.6K	0.102	0.150	**0.095**	0.206	0.224	0.196	0.270	**0.170**	0.313	0.418
Votable2	841.0K	0.086	0.136	**0.083**	0.204	0.231	0.208	0.298	**0.198**	0.316	0.470
Nasa	948.9K	0.107	0.159	**0.105**	0.222	0.265	0.223	0.321	**0.212**	0.324	0.519
Lineitem	1.0M	0.127	0.180	**0.122**	0.235	0.300	0.215	0.310	**0.207**	0.316	0.481
XPATH	1.7M	**0.113**	0.172	0.115	0.221	0.274	0.218	0.308	**0.203**	0.328	0.510
Treebank_e	4.9M	**0.118**	0.183	0.127	0.243	0.310	**0.241**	0.341	0.244	0.345	0.545
SwissProt	5.4M	**0.273**	0.335	0.275	0.351	0.466	**0.25**	0.36	0.33	0.36	0.57
DBLP	6.8M	0.281	0.344	**0.275**	0.338	0.479	0.26	0.36	**0.24**	0.38	0.56
XCDNA	16.8M	**0.248**	0.306	0.253	0.330	0.403	0.742	0.951	0.733	**0.646**	0.989

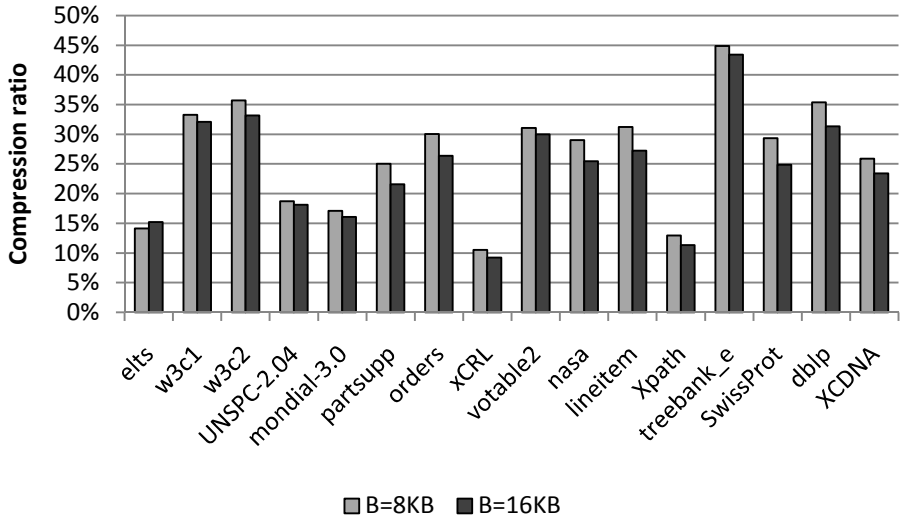


Figure 6.3 – libBZip2-block compression: Textual data of XML documents is arranged in document order.

Table 6.4 – Textual data compression. File names, text + attribute node count (n), uncompressed text data size, compression ratio for BZip, FM-Index in document order, and libBZip2 in document order and path-order. LibBZip2 block size = 8KB.

File	n	Uncompressed text	DocOrder		Doc-order	Path-order
			BZip2	FM-Index	libBZip2	
Elts	4,832	39 KB	12%	13%	14%	11%
w3c1	12,879	152 KB	26%	41%	33%	33%
w3c2	11,597	136 KB	27%	42%	36%	35%
Mondial-3.0	58,941	531 KB	16%	27%	19%	22%
UNSPSC-2	82,370	688 KB	15%	24%	17%	15%
Partsupp	48,002	1,088 KB	17%	27%	25%	22%
Orders	150,002	1,488 KB	20%	30%	30%	22%
xCRL	229,448	3,079 KB	8%	13%	10%	9%
Votable2	841,667	5,376 KB	30%	18%	31%	31%
Nasa	1,005,205	15,530 KB	20%	27%	29%	25%
Lineitem	1,022,977	6,152 KB	21%	27%	31%	21%
XPATH	1,681,713	13,314 KB	10%	16%	13%	9%
Treebank_e	4,874,945	58,757 KB	42%	58%	45%	42%
SwissProt	16,814,101	49,795 KB	18%	20%	29%	17%
DBLP	6,792,148	73,077 KB	25%	30%	35%	28%
XCDNA	5,432,193	261,953 KB	19%	19%	26%	17%

6.4.3 Text DS experiments

Figure 6.3 shows the compression ratio of using a compression library of the Bzip2 data format called *libBzip2* [11] with block size 8KB and 16KB. We observe that the libBZip2 with block size 16KB is generally better than block size 8KB, but not by much. However for the file `Elts.xml` the compression with block size 8KB was better than compression with block size 16KB. Given the small difference of compression ratios between the block size 8KB and 16KB, applications would benefit from the smaller block size because the decompression of the smaller block is faster. In addition, we can access individual data values quicker in the 8KB blocks, especially for a collection of textual values that are small in length and where the text value begins far away from the start of the block. For such a case in a 16KB block, we may have to read double the number of characters than an 8KB block.

In Table 6.4, we show the compression ratio of BZip2 on the textual data in the XML documents, the textual data of each file is arranged in two representations; path-order and document-order. Path-order is where the textual data with the same upward path

from leaf node to root are arranged together in the concatenated file. Document-order is where we concatenate the textual data as we meet them in a document-order traversal of the tree. We also compare the compression ratio of compressing the textual data with FM-Index. We observe that textual data arranged in document-order compresses comparably well to text in path-order. The compression ratio of BZip2 is roughly similar to FM-Index as mentioned earlier (we exclude the fixed cost of the cache in the BZip2 columns in Table 6.4, so FM-Index is better than it seems at first sight).

The compression performance of the two representations are roughly similar. FM-Index allows the searching for arbitrary substrings in hundreds of megabytes within a few milli-seconds [29]. The FM-index is recommended if the string values are not accessed very often, or the access is highly non-local, or the search functionality is desired, but if the strings are accessed frequently with a degree of locality, the blocked BZip2 is recommended.

6.5 Summary

We have shown space-efficient solutions to represent the textual data arising in XML documents, where we are using either FM-Index or blocked BZip2. The experiments show both compression algorithms have a compression ratio that is almost the same, i.e., on average over all files, FM-Index and blocked BZip2 compress the file to 27% and 26%, respectively. We are now able to get good compression ratios on the text data. In addition, the offsets for accessing the individual text data required careful consideration to also represent space-efficiently. We engineered several prefix sums data structures that support the `SUM` operation, answering the queries to retrieve the offset values, which in turn allows us to access individual string values in the text data structures.

We gave compressibility measures for our prefix sums data structures. For our data sets, Golomb encoding and the succinct bound are usually very similar, and they generally use less space than γ and δ encoding. The succinct prefix sums data structure is faster than the γ codes data structures when space usage is comparable. The CNEW

bit-vector has similar or better speed than the other bit-vectors and uses less space in the worst case.

Chapter 7
Succinct DOM

In this chapter, we present our DOM implementation, called *Succinct DOM* (SDOM), bringing together as building blocks the succinct data structures studied in isolation in previous chapters. SDOM is principally suitable for representing large, static XML documents. We currently support almost all read-only operations of the DOM Level 3 Core API.

We analyse the space usage of SDOM compared to Xerces, Saxon's TinyTree and to several XML compressors. In addition, we compare to Xerces the operational performance of traversing a DOM tree, retrieving simple textual data and node type statistics.

We interface the DOM operations with an intermediate representation of the succinct data structures, together with new data structures that are more XML specific. The class structure of SDOM is similar to that of Saxon.

The chapter is organized as follows: We begin by presenting the architecture of the SDOM implementation. Here we discuss each component giving its purpose, operations supported, existing solutions and our own solution. In what follows, we assume that an integer or pointer is 32 bits long. In Section 7.2 we discuss the interface of SDOM to other XML applications. Finally, in Section 7.3, we present the experimental evaluation of SDOM. Parts of this chapter were published as [24].

7.1 SDOM architecture

SDOM consists of 4 core components as shown in Figure 7.1. We see the DOM `document` node, which contains 4 pointers to the SDOM components, these are:

- the succinct tree data structure (DS), henceforth called *`STree`*,
- the *`Namecode`* DS, which stores the XML names for the nodes in the document,
- the *`Text`* DS, which handles the textual data in the document,
- the *`Attribute`* DS, which handles the attribute nodes in the document and their associations to the element nodes.

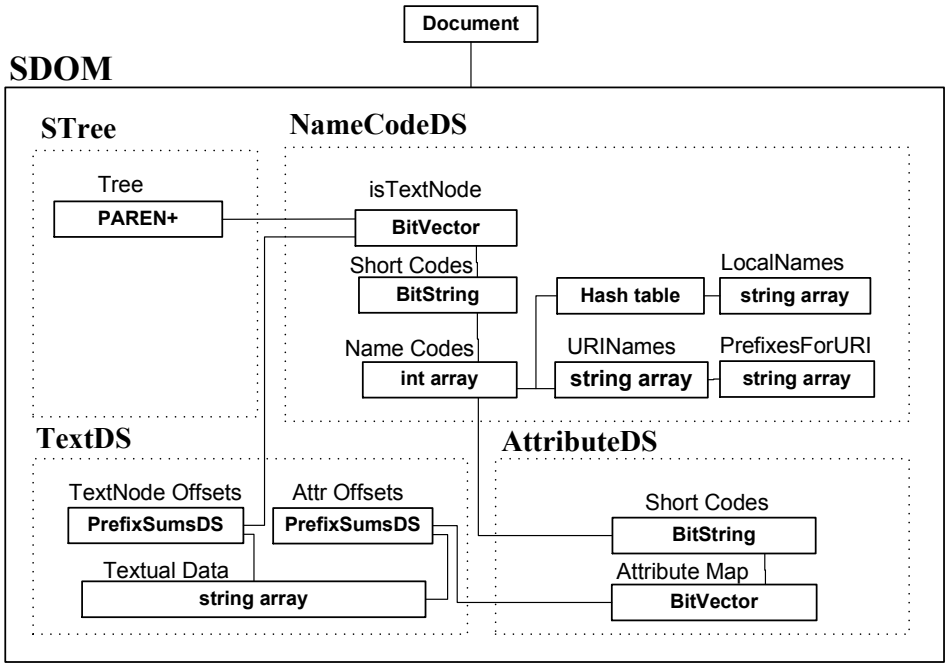


Figure 7.1 - DOM architecture. SDOM stored in the Document node. SDOM components shown with dotted boxes. Connecting lines show relationships between data structures, i.e. compute operations by passing of data in either direction.

The `Text` DS component consists of an uncompressed representation of the textual data. However, a compressed representation can be achieved simply by replacing this sub-component in the Text DS, using a text data structure given in Chapter 6. Henceforth SDOM with compressed text we call *SDOM-CT*.

In what follows we discuss each component in detail, and mention the DOM operations that are directly supported by it. Clearly, some operations rely on more than one data structure, however we give the primary operations here. In Appendix B, we provide a full detailed list of DOM operations (DOM levels 1, 2 and 3), indicating those supported in SDOM.

7.1.1 STree & Node Object

Purpose: Provide support of the navigational operations for the XML tree structure in DOM.

Operations supported: This component primarily supports the operations of the DOM `Node` interface:

- `parent()`
- `firstChild()`
- `lastChild()`
- `nextSibling()`
- `previousSibling()`
- `childNodes()`
- `hasChildNodes()`
- `compareDocumentPosition()`

This component also supports:

- The `TreeWalker` interface. Here we have the same navigation operations as in the `Node` interface, in addition to the `nextNode()`, `previousNode()` and `currentNode()` operations.
- The `item()` and `length()` operations in the `NodeList` helper interface available to the DOM.
- The `following`, `preceding`, `descendant` and `ancestor` axes in XPath.

Existing solution: We discussed in Chapter 3 existing solutions of the XML tree structure, such as Xerces, which represents the tree nodes as objects consisting of several pointers, to the parent, first-child, next-sibling and previous-sibling node. The total cost per node of the pointers is typically 256 bits for an internal node and 128 bits for a leaf node since there are no first-child or child-node list pointers.

Our solution: We use the `PAREN+` representation as the tree structure and recall from Chapter 5 that a node is represented by a double number; the node number i in document order (from 1 to n) and its position $\varphi(i)$ in the succinct tree bit-string representation (from 1 to $2n$), where n is the number of nodes in the tree. Recall that $i = \mathsf{RANK}_0(\varphi(i))$, if we represent '(' by **0** and ")" by **1**. The new `node` objects each contain the integers i and $\varphi(i)$ and a reference to the containing `document` node.

The navigation operation process works as follows: we first access in the node the pointer to the `document` node, then access the `PAREN+` object, which allows us to call the navigation operations of the underlying succinct tree representation which in turn gives the answer as a double number, which is then wrapped in a node object. It is important to remember that, unlike a pointer-based DOM implementation, SDOM does not create all `node` objects in a document when the XML document is parsed but creates a `node` object whenever a navigational operation is invoked on an existing `node` object (the implementation currently does not check if an object has previously been created for the same node). The double number and the `document` node pointer requires 96 bits to represent a node internally. The `document` node pointer is required in SDOM nodes because the `document` node object stores pointers to all SDOM internal components. For example, a navigational operation at a node requires access to the tree representation via the `document` node. Nevertheless, the node representation in SDOM is better than Xerces, which requires several more pointers to represent a node (particularly an internal node). We navigate the tree representation through the navigational operations in the node object. The C++ object must be explicitly freed. As an alternative to avoid the creation of node objects, we recommend the use of the `TreeWalker` class for navigation (see Section 7.2 for details).

The parentheses sequence of the XML document in Figure 7.2 (a) is shown in (b); we identify `element` nodes in circles and `text` nodes in boxes. We ignore for now the storage of the node type information and thus focus only on the structure of the DOM tree, as shown by the parentheses string (c). Nodes are represented by the double numbering encapsulated in a `node` class object.

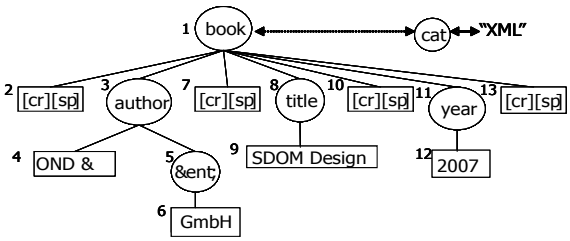
We improve `PAREN+` with the speedup of the primitive operation to go from a node to the next/previous node in document order. This primitive is available in the DOM `TreeWalker` class (see Section 7.2.2), and is also required to iterate along the XPath axes `following` or `preceding`. We define two new operations on the parentheses representation:

- NEXTOPEN(x): To return the position and RANK of the next opening parenthesis given that we are at the opening parenthesis at position x in the bit-string. Formally, NEXTOPEN returns $(i+1, \varphi(i+1))$ if $i < n$ and NULL otherwise, where $x = < i, \varphi(i) >$.
- PREVIOUSOPEN(x): Analogous.

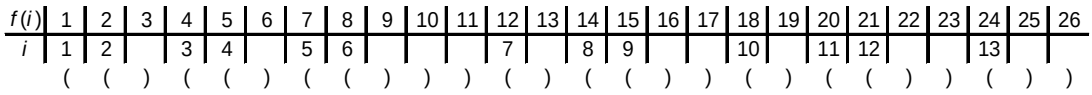
These are implemented straightforwardly by inspecting bits in the parentheses sequence. An individual call to NEXTOPEN (PREVIOUSOPEN) skips over at most d closing (opening) parentheses, where d is the depth of the tree; thus its worst-case time complexity is $O(d)$, but with a small constant. In our experiments (Section 7.3.3), we show that using NEXTOPEN is the fastest option for document-order traversals.

To understand why, we need to understand how going to the next node using the standard navigational operations varies with the location of the current node (we consider document-order traversal, a reverse document order traversal is symmetric). For a non-leaf node, the next node is its first child. The pseudocode for FIRST-CHILD (Table 5.6) shows that this only requires the inspection of a bit in the parentheses sequence, and is consequently very fast. For a leaf node, the next node is its following sibling, and locating it is almost as fast as finding the first child of a non-leaf node, except when the leaf node is the last child of its parent. Note that the number of nodes that are the last child nodes equals the number non-leaf nodes, which is usually a $1/3$ of nodes in the tree. Thus, for at least $1/3$ of the nodes, moving to the next node in document order requires significant computation. A series of alternating parent and next-sibling calls is made, both of which are relatively expensive (generally similar to a few memory accesses). Using NEXT/PREVIOUSOPEN is much faster in this case.

```
<book catalogue="XML">
  <author>OND
  &amp;&ent;</author>
  <title>SDOM Design</title>
  <year>2007</year>
</book>
```



(a) (b)



(c)

Figure 7.2 - (a): Simple XML document fragment. (b): Corresponding DOM tree representation. (c) Parentheses representation of the tree structure with double numbering of nodes. E.g., the 11th node (the element 'year') is at the 20th position in the bit-string. The entity `&ent;` represents the text 'GmbH'.

Finally, we consider non-navigational operations, specifically, comparing the position of two nodes v and w. Note that:

- if v precedes w in document order, both components in the double-numbering for v will be less than their corresponding components in w. Thus, given two nodes, we can check to see if a node precedes another in document-order by just looking at the double-numbering of the two nodes.
- v is an ancestor of w if and only if $\varphi(v) < \varphi(w) <$ `FINDCLOSE`$(\varphi(v))$. Thus, we can check if w is an ancestor of v by a single call to `FINDCLOSE`, since `FINDCLOSE`$(\varphi(w))$ must precede `FINDCLOSE`$(\varphi(v))$ because the XML document is well-formed.

These operations allow us to support the `compareDocumentPosition()` function quickly. The `PAREN+` representation also allows us to compare two nodes v and w by any of the main XPath axes:

- `ancestor`/`descendant`: as above.

- `preceding`: v is in the set of nodes preceding w if $\texttt{FINDCLOSE}(\varphi(v)) < \varphi(w)$.
- `following`: similar to preceding.
- `parent`/`child`: obtained directly from the navigation operations.
- `following-`/`preceding-sibling`: v is a following (preceding) sibling of w if v is after (before) w in document-order and v and w have the same parent.

7.1.2 NameCode Data Structure

Purpose: Store the name and type information of each node in the DOM tree.

Operations supported: This component primarily supports the operations of the DOM `Node` interface:

- `getNodeName()`
- `getTagName()`
- `getPrefix()`
- `getLocalName()`
- `getNodeType()`
- `hasChildNodes()`
- `lookupPrefix()`
- `getPrefix()`

This component also supports:

- The operations `getElementByTagName()`and `getElementByTagNameNS()` in the `Document` interface.
- The operation `getTagName()`in the `Element` interface.

Existing solutions: Xerces stores the node names as three pointers, each to a C++ string, requiring 96 bits per node. For an element node name with a namespace prefix, the first pointer points to the namespace URI, the second to its prefix and the third to its local name. An `element` node without a namespace prefix has a single pointer (instead of the three) to its local name. Nodes with the same name point to the same string.

Node types in Xerces are not explicitly stored. They are represented through the class representing the node (see Section 3.1.1 for details), for example, an `element` node is an instance of the `DOMElementImpl` class derived from the `DOMNode` class. Therefore, for the `getNodeType` operation, the node type of a node is known in its derived class.

In Saxon's `TinyTree` data structure, node names are represented using an Namepool data structure [61] (as discussed in Section 3.1.2), and are mapped to 32-bit integers (name-codes). The name-codes are stored in an array structure of length n, representing the XML document in document-order. Each name-code has three components, which represent the fully qualified name information (prefix, URI and local-name). The node type information is represented in an array called the *nodeKind*, which is of length n, requiring eight bits each. For text nodes, the 32-bit value stored gives an offset into an array containing the textual data.

Our solution: Our solution comprises three parts: `isTextNode` bit-vector, the `Namepool` and the `shortCode` data structure.

Namepool data structure.

Initially in SDOM, the fully-qualified names for elements and attributes are converted into 32-bit name-codes. The data structure for mapping string names to name-codes and back follows Saxon's Namepool data structure closely.

However, the use of a 32-bit name-code is costly, since there tend to be very few distinct name-codes. For example, one of our XML documents `SwissProt.xml` has a total of 5166890 `element` and `attribute` nodes in the document, but only ninety-nine of these are distinct name-codes. In SDOM, to save space, we use an additional level of indirection. Initially, we store each unique element name as 32 bits in an array we call a *name-code table* (these can be decoded as in Saxon using the Namepool data structure).

Our approach begins by splitting the nodes into text nodes and non-text tree nodes. Specifically, we number all text nodes from $1..t$ in document order, and all non-text

tree nodes (mostly `element` nodes, but including `comment` nodes, `entityReference` nodes etc.) from $1..e$, where t and e are the number of text nodes and non-text tree nodes, respectively (note that $t + e = n$). The reason for this split is, in brief, that while the information associated with non-text tree nodes and text nodes can be compressed effectively, the compression methods are rather different. The splitting is done using the `isTextNode` bit-vector, as we now explain.

IsTextNode Bit-Vector

The `isTextNode` is defined as follows: the ith bit is set to **1** if the ith node in document order is a text node, otherwise it is set to **0**. By augmenting the `isTextNode` bit-vector with the RANK operation, we provide a consecutive numbering of text nodes from 1 to t and of non-text tree nodes from 1 to e. For example, if node i is a text node, then RANK_1(`isTextNode`, i) gives the ordinal position of node i among the text nodes, considered in document order, and if node i is a non-text tree node, RANK_0(`isTextNode`, i) gives the ordinal position of node i among the non-text nodes. The CJ bit-vector implementation (discussed in Section 4.2.1) is used to support the RANK operation, therefore the space usage of `isTextNode` is $1.5n$ bits.

Short-code data structure

We then create an "array" of size e. The ith entry of this array is a *short-code* for the ith non-text tree node in document order. A short-code is a positive integer, interpreted as follows:

- If the ith short-code is 12 or less, then the ith node is not an `element` node, and the short-code value gives its node type. The possible node types and their values are: `CDataSection` (4), `entityRef` (5), `processingInstruction` (7), `comment` (8) or `docType` (10). The other node types supported in SDOM (i.e. `Entity` (6), `Notation` (12)) are only present in the XML document prolog (see Section 2.1.3), therefore not in the DOM tree. The

`DocumentFragment` (11) node type is a feature of dynamic DOM implementations, therefore not supported in SDOM.

- If the ith short-code j is 13 or greater, then the ith node is an `element` node, and $j - 13$ is an index into the name-code table, pointing to the entry in this table corresponding to the ith element name.

The short-codes thus take $l = \lceil \log(p + 12) \rceil$ bits each, where p is the number of distinct name-codes in the document. The short-codes are usually much smaller than name-codes. For example, in `SwissProt.xml` $p = 99$ and each short-code is $\lceil \log(99 + 12) \rceil = 7$ bits long. We concatenate all short-codes into a bit-string, using the *compacted bit-string* data structure, described in Section 6.2.5. To extract the ith *short-code* we call `subBitString`$(l \times i, l \times (i + 1) - 1)$.

We now explain some of the design decisions. It is worth bearing in mind that text nodes appear to be the most common kind of node in the tree, and they comprised nearly two-thirds of the nodes in many of our documents (as discussed in Chapter 3).

- We first consider the use of the `isTextNode` bit-vector. The planned representation of textual data (described in Chapter 6) anyway requires text nodes to be numbered consecutively. One could get around this by treating all nodes as text nodes (those without any real textual data could be given a dummy "null" string). This would increase the space usage of the offset data structure. In addition, the short-code array would typically become 2-3 times longer; since short-codes are often 6 bits or more, the savings in the short-code array easily pay for the cost of the `isTextNode` bit-vector.
- Next, we argue that it does not make sense to apply the same separation to other kinds of nodes, e.g. `comment` and `CDataSection` nodes. To do so would require an additional bit-vector of length e, with a space cost of $1.5e$ bits. However, the space savings obtained in the *Short-code* array by removing the `comment` and `CDataSection` nodes would normally be small and would not normally cover the costs of the bit-vector.

Table 7.1 – Pseudocode of DOM Methods, (a): `getNodeType()` and (b): `getNodeName()`.

```
getNodeType(int node_i){                    1
 if(isTextNode[node_i]=1)                   2
   return TEXT                              3
 else                                       4
   x= RANK0(isTextBit, node_i)              5
   scode=getShortCode(x)                    6
   if(scode>12)return ELEMENT               7
   else return scode                        8
}                                           9
```

```
getNodeName(int node_i){                    1
 if(isTextNode[node_i]=0)                   2
  x= RANK0(isTextBit, node_i)               3
 scode=getShortCode(x)                      4
  if(scode>12)                              5
   namecode=decode(scode-13)                6
   return QName(namecode)                   7
 else                                       8
  return undefined                          9
}
```

Table 7.1 shows pseudocode for the get`NodeType()` and the `getNodeName()` operations. The identification of a node type using the get`NodeType()` operation is trivial: we access the `isTextNode` bit-vector. If the *i*th bit is **1**, then the *i*th node is a `text` node (see lines 2-3). Otherwise, the *i*th node is some other node type and we must make use of the short-code array to find this information out (see lines 5-8).

For the `getNodeName()` operation we first map the document-order number to the non-text number in lines 2-3. As x is a number in the range 1 to e, we fetch the xth short-code in the compacted short-codes. If the short-code value is greater than 12 then the node in question is an `element` node and the short-code represents an index into the name-code array (line 6-7). We access the name-code table to output the fully qualified name, using the `QName()` operation, which is supported by the `Namepool`. The node name of a non-text node that is not an `element` node is undefined.

7.1.3 Textual Data Structure

Purpose: Store and retrieve the textual data of individual nodes or groups of nodes within the XML document.

Operations supported: This component primarily supports the DOM operations of the `Node` interface:

- `getNodeValue()`
- `getTextContent()`

This component also supports:

- The `getElementByID()` method of the `Document` interface.
- The `getValue()` method of the `Attribute` interface.
- The `getData()` method of the `ProcessingInstruction` interface.

Existing solutions: Xerces stores the textual data as a pointer to a C++ string. `TinyTree` represents the textual data in an array of strings. As discussed in Section 3.1.2, the `alpha` array provides indexes for the `text`, `attributes` and `comment` nodes into the string buffers.

Our solution: In SDOM, we make the improvement of concatenating the textual data of the XML document into a single C++ array. The textual data for the following node types are stored:

- `Text` – data value associated with the text node
- `Attributes` – attribute node value
- `ProcessingInstruction` – data component of the processing instruction
- `Comment` – content of the comment node
- `CDATASection` – content of the CDATA Section

Recall that in Chapter 6, we gave a data structure for storing a collection of non-empty strings $s_1, \ldots, s_t$, concatenated into a single string, which is either held in a compressed

or uncompressed format. The lengths of the textual nodes are stored in a prefix-sums data structure. The data structure retrieves the ith string.

Given an index i, we have two instances of the textual data structure. The first instance handles the `text` nodes. We assume the `NameCode` data structure numbers the text nodes, given in the DOM tree from 1 to t in document order (using the `isTextNode` bit-vector) and stores the collection of strings $T_1, \ldots, T_t$ in the string data structure of Chapter 6, where T_i is the value of the ith `text` node. The second instance handles the remaining kinds of textual nodes, such as `attribute`, `comment`, `processingInstruction` (target data) and `CDataSection` nodes. We assume the attribute data structure (Section 7.1.4) numbers these nodes from 1 to a (where a is the number of `attribute` nodes, including the other nodes given above).

The reason for doing this (rather than storing all strings in a single instance of the string data structure) is that the other kinds of textual nodes are typically far less numerous than `text` nodes (see statistics in Section 3.3), and appear to have different distributions of lengths. The prefix sums data structure discussed in Chapter 6 represents the lengths of the `text` nodes, and the space usage of the data structure is based on the average length of a text node. We observe the separation of the `text` nodes and `attribute` nodes may have some benefit in the space usage of the prefix sums data structure (see Proposition 4.5); if t' is the number of `attribute` nodes, and m' is their total length, then by the convexity of the log function,

$$(t + t') \log ((m + m')/(t + t')) \geq t \log m/t + t' \log m'/t',$$

so the space consumption of the offsets into the character arrays is always reduced by separately considering the offsets. For example, this avoids the risk that one very large `comment` node raises the average length of all textual nodes in the tree, and thus the space usage of all offsets, if the offsets for `text` nodes and `attribute` nodes (including other nodes given above) were combined.

7.1.4 Attribute Data Structure

Purpose: Provide mapping of `attribute` nodes to the `element` nodes (where they are declared) in the DOM tree. Store the name information of each `attribute` node and the associated node value.

Operations supported: The DOM defines a set of operations to search and access the attribute nodes belonging to an element through the `NamedNodeMap`. The `Node` interface specifies the operation `getAttributes()`, which returns a `NamedNodeMap`. This component primarily supports the DOM operations of the `NamedNodeMap` and the `Attribute` interfaces:

- `item()`
- `length()`
- `getNamedItem()`
- `getNamedItemNS()`
- `getName()`
- `getOwnerElement()`
- `isID()`

This component also supports:

- The `getElementById()` and `getElementByTagNameNS()` methods of the `Document` interface.
- The `getAttributes()` and `hasAttributes()` methods of the `Node` interface.
- The `getAttribute()`, `getAttributeNS()`, `getAttributeNode()`, `getAttributeNodeNS()` and `hasAttribute()` methods of the `Element` interface.
- The `getTarget()` method of the `ProcessingInstruction` interface.
- The `getName()` method of the `DocType` interface.
- The `getNotationName()` method of the `Entity` interface.

Existing Solutions: Xerces represents `attribute` nodes belonging to an `element` node in the `NameNodeMap` class. Each `element` node object has two pointers which are instances of the `NameNodeMap` class: the first pointer is to standard attributes (where

the attributes' name and value are defined), and the second pointer is to the default attributes (where the DTD is required to retrieve the default attribute values, if not defined in the document). These pointers are null for `element` nodes that do not have any attributes. The *`NameNodeMap`* class has a pointer to its owner element object and an instance of a vector. The vector has an array of pointers to instances of the attribute node; in addition, we have two integer variables to maintain the vector. In total the attribute node itself has nine pointers with information such as its name, value and owner document (see Section 3.1.1 for details).

`TinyTree` represents all the attribute nodes in the document using three integer arrays, where each array item represents an attribute in document-order. The first array stores the node number (index) of the element that is the attribute's parent. The second array stores the name-codes of the attribute names. The attribute values are stored in a string array. The navigation of elements to their attribute nodes is supported by the `alpha` array, which provides the mapping from an element node to the index of its first attribute node (see Section 3.1.2 for details).

An important issue that arises is whether to place attribute nodes within the tree structure of the XML document. This approach is taken by a number of XML compressors [14], [30], [48], [55], but appears to be unsuitable for SDOM. This design decision has already been indicated by our definition of "XML document structure" in Chapter 5, and is justified in this section.

Our solution: In SDOM, `attribute` nodes are represented separately from the tree representation. We propose a mapping strategy, which maps `elements` to their `attributes`, and `attribute` names to their values.

We now describe the attribute data structure. Recall from Section 7.1.2 that the `isTextNode` bit-vector numbers non-text tree nodes from 1 to e. We create a sequence of non-negative integers $X = (x_1, \dots, x_e)$ of length e as follows. If the ith non-text tree node is an `element` node, then x_i is the count of attributes it has. If the ith non-text tree

node is any of `processingInstruction`, `CDataSection`, `docType`, `document` or `comment` nodes, we give it a dummy attribute, therefore $x_i = 1$.

Let a be the sum of the x_is (i.e. a is the total number of attributes, including dummy attributes). We now show how to represent X to satisfy the following goals:

(a) All attributes should be numbered from 1 to a, and the attributes associated with a given non-text tree node should be numbered consecutively.

(b) Given a non-text tree node, it should be possible to determine quickly the range of integers that number its (dummy) attributes, if any.

These requirements are met as follows. We consider each non-text tree node in document order, and number all its (dummy) attributes consecutively. The attributes (if any) of the first non-text tree node are numbered starting from one; for any other node, its attributes (if any) are numbered starting from the next available integer. Clearly, all attributes of a node are numbered consecutively, and (a) is satisfied.

For (b), we represent X as a bit-string (called `attr_association`) as follows. Each value x_i is written in unary (e.g. if $x_i = 4$, then x_i is written as **11110**) and concatenated in order (see Figure 7.3 (b) for an example). Note that this bit-string has e **0**s and a **1**s, and it is stored as a bit-vector that supports $\texttt{SELECT}_0$. The attributes of the ith non-text node are numbered from $\texttt{SELECT}_0(i-1) - i + 2$ to $\texttt{SELECT}_0(i) - i$ ($\texttt{SELECT}_0(i) - i$ gives the number of **1**s before the ith **0** in the bit-string). Hence (b) is satisfied.

The attribute names are represented analogously to the element names in the short-code data structure. Initially we create an array of size a, which stores the short-codes of the attribute names and node types of the dummy attributes. The array is then compacted to its final representation as described in Section 7.1.2. The strings $s_1, \ldots, s_a$ are numbered, where s_i is the textual data associated with the ith attribute or dummy attribute node.

```
<root>
  <U a="val" b="val" c="val" />
  <V /> <!-- comment -->
  <W d="val" e ="val">
  <X f="val" g ="val" h="val" i="val">
  <Y j="val">
  <Z />
</root>
```

(**a)**

root				U	V		//			W					X		Y	Z
	a	b	c				com	d	e		f	g	h	i		j		
	1	2	3			4		5	6		7	8	9	10		11		
0	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**

(b)

Figure 7.3 - (a) Example XML document with elements and associated attributes. (b) Bit-string of the attribute representation.

The attribute name and value are accessed via the `Attribute` class in SDOM, which is a derived class of the `Node` class. Since the `attribute` node is not in the DOM tree, calling operations such as `previousSibling()` or `nextSibling()` returns a null value, but for `parent()` it returns the `element` node associated with the `attribute`. In SDOM, an `attribute` node object has four values:

- A pointer to the `document` node to which it belongs. This pointer is required because the attributeDS is referenced in the `document` node object.
- An attribute number in the range 1 to a,
- The double number $< x, y >$ of its parent (the `element` node).

The numbers $< x, y >$ are filled in at the time of creation of the `attribute` node (this can only happen when navigation in the tree to the attribute's parent node is performed). The `attribute` node exists until the user deletes the object.

The operation to retrieve the attribute name is analogous to the `getName()` operation of Table 7.1 for `element` nodes. The operation to retrieve an `attribute` node value is

analogous to the `getNodeValue()` method (Section 7.1.3), except that we have to get the `attribute` node number 1 to a before operations on the `attribute` node are computed.

Implementation of the NamedNodeMap

The `NamedNodeMap` interface represents a collection of nodes that can be accessed by name or by selecting the ith item in the collection. We implemented a specialised `NamedNodeMap` class for `attribute` nodes in SDOM, which contains the following values:

- Pointer to an instance of the parent node (i.e. the `element`), which contains as a class member the double node number and a pointer to the `document` node.
- The number of `attribute` nodes belonging to the `element` node.
- The starting attribute position in the `attr_association` bit-string. This is computed upon the creation of the `NamedNodeMap`.

The operation `item(`i`)` is simple, since upon the creation of the `NamedNodeMap` we know the starting attribute number of the attribute group belonging to a particular element. Given we know its length, the ith attribute will appear at the position `start_position+`i.

We show in Table 7.2 the pseudocode of the DOM node operation `getAttributes()`. In lines, 2 and 3 we show the mapping of the tree node number to the non-text node number and check that the node number is an element node. The `attrNodeCount()`operation (line 4) retrieves the total count of attributes belonging to the element. If this count is greater than zero, we then return a new `NamedNodeMap` instance.

To access of an `attribute` node belonging to an `element` node we create a new node object. We first get the non-text number of the `element` node from 1 to e in the

`isTextNode` bit-vector. Let z be the non-text number of the element node. We find the first attribute belonging to the `element`.

With the call $j = \texttt{SELECT}_0(\text{attr\_association}, z-1) + 1$. To retrieve the ith attribute node we compute $\texttt{RANK}_1(\texttt{attr\_association}, (j-1)+i)$. An instance of the DOM `Attribute` class is returned to the user.

We now discuss alternative representations that include the attributes in the document tree structure. One option is to make `attribute` nodes "special" children of their parent element node (for the sake of concreteness, let us say that if a node has attributes, then they appear before all its "real" children).

An obvious disadvantage of including `attribute` nodes in the tree structure is a slow-down in the navigational operations. For example, if we were to perform a `firstChild()` operation on a node x, then we would need to check that the node that we have reached is not an `attribute` of x, and if it is, then we would need to skip over all its attributes to reach the "real" first child. However, there are disadvantages in terms of space usage as well; depending on how exactly this is done.

We consider two alternative ways to associate attributes with their values. In the first, we store only the `attribute` nodes (not the attribute value) as the first children nodes of their parent nodes (the element nodes where they are defined) in the tree (Figure 7.4 (b)).

Table 7.2 – Pseudocode of Attribute DS interfacing with DOM methods. <i, j> is the double number of the node in the tree.

```
getAttributes(<i,j>){                                              1
  otherNr= RANK0(isTextNode,i)                                     2
  if(getShortCode(otherNr)>12)                                     3
    attCount=attrNodeCount(otherNr)                                4
    if(attCount>0)                                                 5
      return new NamedNodeMap(<i,j>,otherNr,attCount)              6
  return NULL                                                      7
}                                                                  8
```

We filter out the `text` nodes using the `isTextNode` bit-vector as before, therefore we have remaining non-text nodes, along with the attribute nodes.Similar to `isTextNode` we require a bit-vector to number the `attribute` nodes from $1..a$ (to access their textual values including `comment`, `processingInstruction` etc), which we call *`hasTextValue`*. This bit-vector would be of length $e + a$ bits, which is the same length as the `attr_association` bit-vector above. In addition, however, we would need $2a$ bits to store the `attribute` nodes in the tree (two bits per node). Finally, the attribute and element name-codes are stored together in an array, their names may overlap, and hence we use the *`hasTextValue`* bit-vector to identify the attributes nodes from the element nodes.

Another alternative is to store the `attribute` (and `comment`, `processing-Instruction`, etc) values as new `text` nodes in the tree (Figure 7.4 (c)). This would add at least $2a$ nodes to the tree, and hence $4a$ bits overall. The `isTextNode` bit-vector would be modified to filter out the new `text` nodes as well, and number all `text` nodes (original and new) consecutively. Compared to the `attr_association` bit-vector, adding $2a$ nodes to the tree could use less space if a is small.

However, this approach would put all textual data into the same data structure, which can cause an increase in the space usage of the textual data structure, as discussed in Section 7.1.3.

Finally, since attribute and element name-codes would be stored together in an array, the name-codes potentially could overlap for the attributes and elements, as they might have the same name; therefore we would need to identify attributes and element nodes. This we can achieve by numbering the short-codes differently for the attributes and elements even if the name-codes are the same. Potentially the space usage of the name-code data structure would double, in this case.

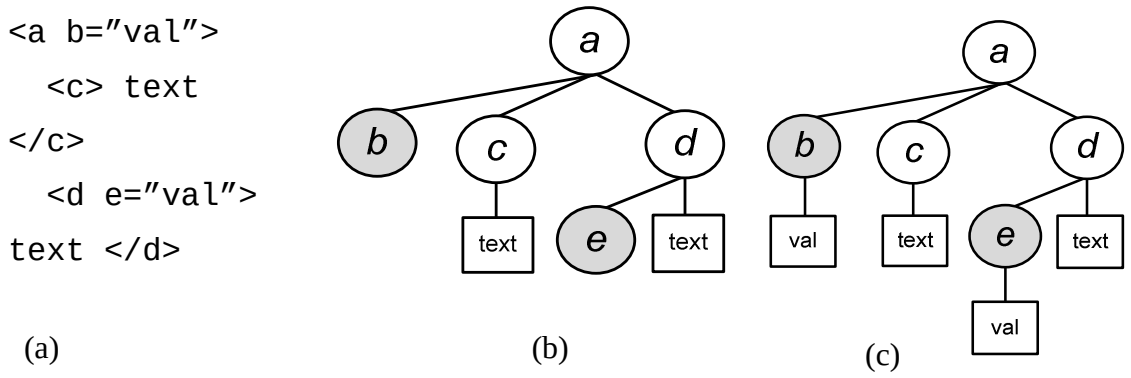


Figure 7.4 – (a) Simple XML document. (b) Tree structure of (a) with attribute nodes (not including textual data) in the tree. (c) Tree structure of (a) with attributes and their values in the tree as nodes.

7.2 SDOM Interface

7.2.1 Class Structure

In this section, we discuss SDOM as an application, which is designed to support DOM and is compatible with XSLT/XQuery processors. We have an intermediate interface which calls the succinct data structures directly, which in turn is called by the DOM operations. The intermediate interface is similar to that used in Saxon [61], which has the *`NodeInfo`* and *`DocumentInfo`* interfaces directly accessing the `TinyTree` data structure. We also support a ported version of the *`NodeInfo`* and *`DocumentInfo`* interfaces in C++, thus allowing SDOM to be a plug-in replacement for `TinyTree`.

In Figure 7.5, we show the class diagram of the `TinyTree` data structure and the interfaces operating directly on `TinyTree`. The class `TinyNodeImpl` (which implements the `NodeInfo`) is used in Saxon's implementation of DOM, as a class member instance of the DOM APIs. The class `TinyDocumentImpl` is also used, as a class member instance in the DOM `Document`. In essence, these class members represent the node, i.e. the `TinyNodeImpl` class consists of a node number and parent pointer. In SDOM, we replace the `TinyNodeImpl` with SDOM's `Node` class (which

consists of two integers for the node and a pointer to the `Document` node), which represents the node object directly. An additional layer implements the `NodeInfo`. The `Document` node provides access to the SDOM data structures. For SDOM, some of the DOM operations directly match those in the `NodeInfo`, for example, the `getNodeKind()` (in the `NodeInfo`), which retrieves the node type information of a node, has the same function as the DOM operation `getNodeType()`. Therefore the DOM `getNodeType()` operation calls directly the `getNodeKind()` operation.

The `getNameCode()` operation retrieves the name-code of a node, operating directly with the SDOM's `NameCode` data structure. This operation is used by the DOM operation `getNodeName()`, where we find in the hash table the matching node name to the name-code.

The navigational operations (with the exception of `parent()`) are not directly supported in `NodeInfo`, but require the use of the `Axis` iterator. In SDOM, we provide direct support of the DOM `node` navigation operations. In addition, we support the `iterateAxis` operations of `NodeInfo`. All the axes are supported in the `iterateAxis`, except the `namespace` axes.

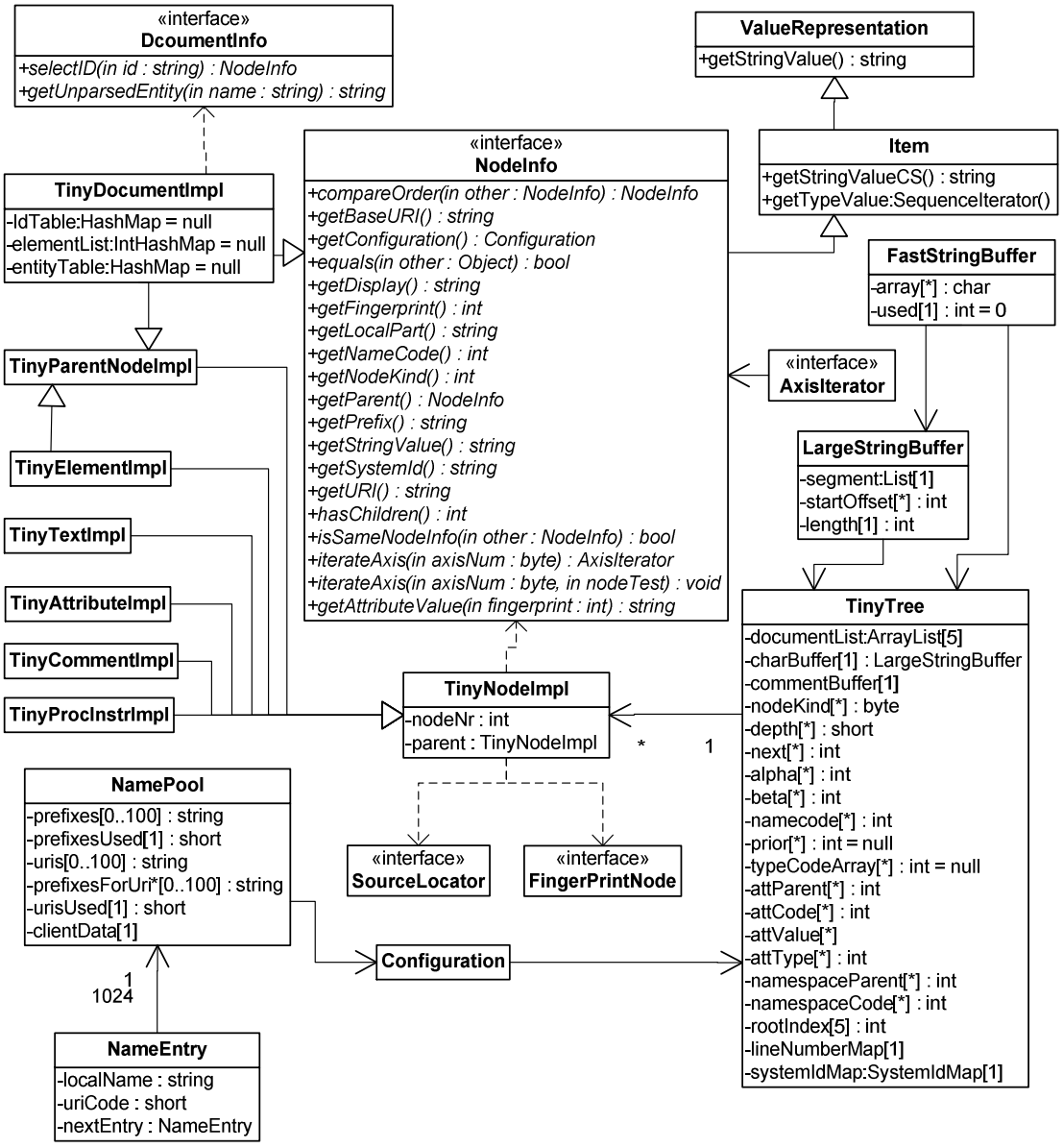


Figure 7.5 – Class Diagram of `TinyTree` and interface classes [61].

7.2.2 DOM TreeWalker Interface

When traversing a document via navigation performed through the `Node` interface, it results in at least one `Node` object being created for each node in the tree (see Section 7.1.1); this collection of `Node` objects will, in many cases, occupy more space than the SDOM representation of the document. To avoid this problem, we recommend the use

of the `TreeWalker` class [78] for navigation; this has an iterator-like behaviour. For example, the `nextNode()` operation in the `TreeWalker` moves `currentNode` pointer to the next node, which is then returned if and only if the next node exists. If the returned value is null, then the `currentNode` remains at the last node visited.

In essence, new `Node` objects are not created by a navigation operation in the `TreeWalker`, but it supports all the navigational operations supported by the `Node` class (our `TreeWalker` implementation does not yet support node filters).

7.3 Experimental Evaluation

In this section we draw comparisons of the space usage and running times between SDOM(-CT), Xerces and `TinyTree` (as `TinyTree` is implemented in Java we did not compare its running times). We also compare our space usage against XML-specific compressors such as XMill, XBZipIndex, XPRESS, XQZip and XGrind (described in Chapter 3). We do not make a detailed comparison with their running times: some do not support queries/navigation (e.g. XMill, XBZip), and those that do, focus on supporting various XPath-like queries rather than navigation, and do not generally report times for navigation. (An exception is [30], where they report navigation operations as taking milliseconds; however, we are several orders of magnitude faster.) The DOM operations supported in SDOM are listed in Appendix B.

7.3.1 Setup

The basic setup of our experiments is outlined in Appendix A. We compare our data structure's running times with Xerces, with testing only done on the Intel-P4 machine. For `RANK` and `SELECT` we use the CJ bit-vector (described in Section 4.2.1), with parameters $B = 64$ and $s = 32$. We used the parentheses implementation of [36] (i.e. `PAREN+`), with parameter $B = 128$.

We tested our SDOM data structure on seven XML files taken from our XML corpus (Section 3.3). Our choice of files gives us a range of typical XML documents (with files

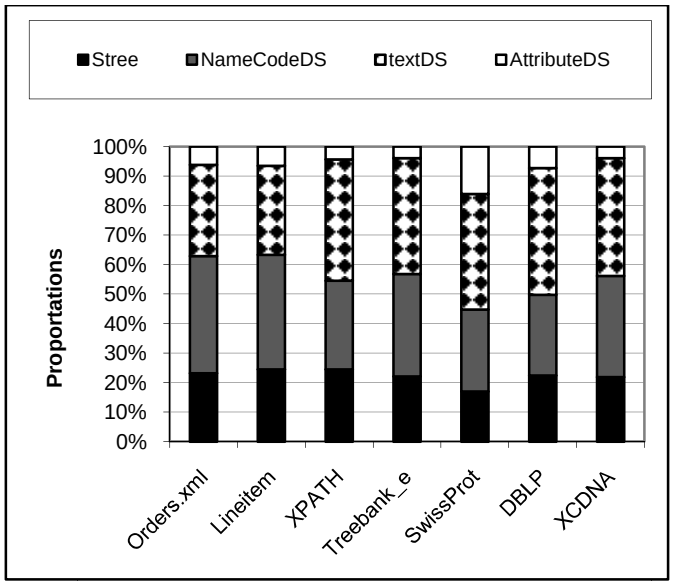


Figure 7.6 - Space usage distribution of SDOM components excluding textual data.

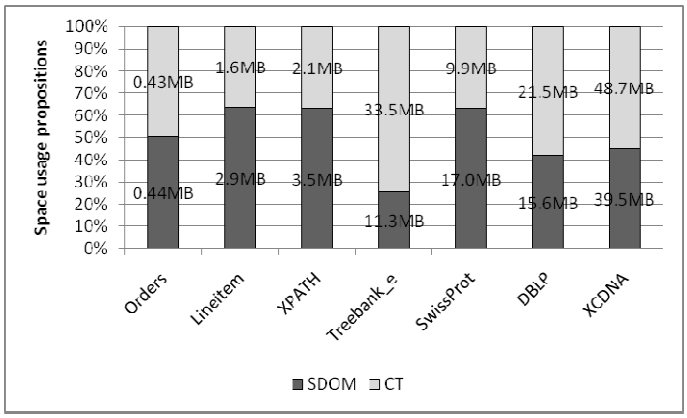


Figure 7.7 - Space usage of SDOM components from Figure 7.6 (shaded in grey) with textual data compressed (shaded in dark-grey).

size ranging from 5MB to 594MB with varied tree structures and textual data). We also ran preliminary tests on a set of synthetically generated files using XMark [70].

7.3.2 Space Usage

The succinct data structures share some static lookup tables (detailed in Section 4.2.1 for bit-vectors and in [36] for the parentheses data structure) with total size

approximately 3.5MB. We have not added this cost in our figures. For relatively large documents, this cost is negligible, and for multiple documents loaded in SDOM, we only pay the cost once.

Figure 7.6 shows the space usage of the SDOM components in their relative proportions (excluding the textual data). Note that with the exception of `Orders.xml` and `Lineitem.xml`, which are not as rich in text nodes as the other files, the textual offset data structure (shaded in black diamonds in Figure 7.6) makes up the largest proportion of the space usage and recall that the succinct representation is four times smaller than the naïve one discussed in Chapter 6.

In addition, the tree structure, despite being very compactly represented, still takes a fourth of the cost of SDOM (excluding the text). The naïve representation, which would require at least $2 \times 32 = 64$ bits per node, would be prohibitive. The `Attribute` data structure is relatively small, as some documents do not have any (or only a few) `attribute` nodes. For documents that have many `attribute` nodes (`SwissProt.xml`) we observe the representation is still relatively small.

Figure 7.7 shows the breakdown of the space usage within SDOM-CT. We see that the compressed text is often smaller than the SDOM components. However we see that for `treebank_e.xml` the compressed text is larger than the SDOM components, which is because the textual data is partially encrypted and therefore does not compress well.

Figure 7.8 compares the space usage of SDOM with other DOM implementations. We observe in Table 7.3 that files that would not easily fit into main memory of our Intel-P4 machine under Xerces, such as `XCDNA.xml` (size 594MB, which Xerces increases by a factor of 4) fit comfortably into the main memory using SDOM.

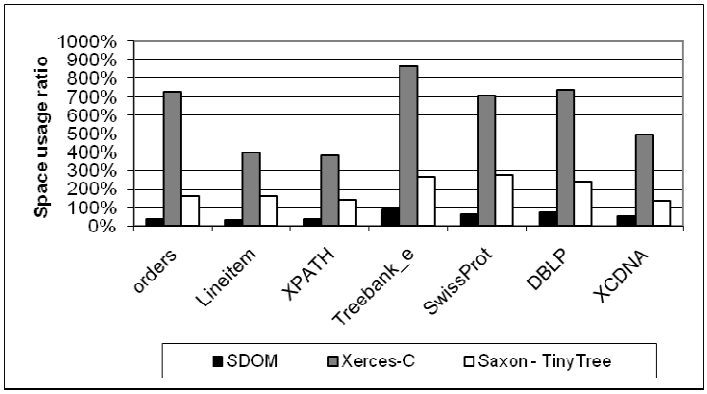


Figure 7.8 - Space usage of DOM implementations compared to original file.

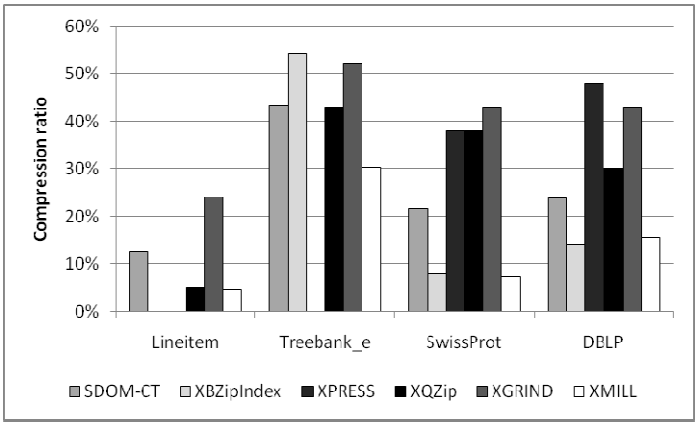


Figure 7.9 - Compression ratio comparisons of the XML compressors.

Figure 7.9 compares SDOM-CT with XML compressors: both query-friendly ones (XPRESS, XQZip, XBZipIndex and XGrind) and a standard compressor, XMill. We quote the results for the other compressors from the papers, and have not re-derived them ourselves (however, for XMill we derived the results using their software). We only show files in Figure 7.9 that are reported by the majority of other compressors. In Figure 7.9 the space usage is expressed as a percentage of the original file size.

Table 7.3 - Space usage of XML representations.

		Uncompressed repn's			Query-friendly compressed representations					
File	Size	SDOM	Xerces	Saxon	SDOM-CT	XBZipIndex	XPRESS	XQZip	XGRIND	XMILL
Orders	5MB	37%	451%	157%	17%	-	-	-	-	12%
Lineitem	32MB	28%	399%	161%	13%	-	-	5%	24%	5%
XPATH	50MB	33%	383%	137%	10%	-	-	-	-	3%
Treebank_e	82MB	84%	866%	266%	43%	54%	-	43%	52%	30%
SwissProt	110MB	60%	704%	272%	22%	8%	38%	38%	43%	7%
DBLP	128MB	68%	737%	240%	24%	14%	48%	30%	43%	15%
XCDNA	594MB	50%	491%	136%	14%	-	-	-	-	8%

In Table 7.3, we show the space usage of the XML processors and XML compressors, where the percentage value is the proportion of the file size. We observe that XMill gives the best compression ratios for all our files (we do not report the results from XBZip, which are similar to XMill); however, XMill does not support query operations upon the compressed representation. We observe that SDOM-CT often gives better compression ratios than the other query-friendly XML compressors.

7.3.3 Running Time

Our tests are based on traversals of XML documents. We always use the SDOM `TreeWalker` interface, and not the SDOM `Node` interface, as discussed in Section 7.2. Even so, there are two different ways of traversing a document in document or reverse document-order in SDOM(-CT):

- Using the `nextNode()` (`previousNode()`) operation.
- Using the standard DOM navigational methods. Over the course of a document-order traversal of an n-node tree, this results in a total of n calls to each of the operations `firstChild()`, `nextSibling()` and `parent()`, thus providing a test that involves a mix of standard navigational operations. The reverse document-order is analogous, using the operations `previousSibling()`, `parent()` and `lastChild()`.

In addition to document-order and reverse document-order, we perform a third kind of traversal, called the *upward path enumeration*, which works as fo*llo*ws. We perform a document-order traversal using the `TreeWalker` navigational methods. When the main iterator reaches a leaf, an auxiliary iterator traverses the entire upward path from the leaf to the root using the DOM `parent` operation.

Along with the traversals, we either gather *basic statistics*, which include the count of `element` and `text` nodes, or p*erform a full test*. In the full test we (i) determine the type of each node. (ii) check whether nodes have associated attributes. (iii) for nodes with attribute data, or `text` nodes, we retrieve the node value, and check to see if the value contains a substring that is unlikely to appear, hence, forcing the substring search to scan the entire text in the node. Each test is repeated several times to obtain stable results (50 times for `Orders.xml`, 10 times for `Lineitem.xml`, `XPATH.xml`, `Treebank.xml`, `DBLP.xml` and `SwissProt.xml`, and 2-5 times for `XCDNA.xml`). The running times are reported as the total of the runs, unless stated otherwise. For `XCDNA.xml` we get the average for a single run of a traversal.

In Figure 7.10, we show the total running times for the traversal of the documents using either the `nextNode()` or the `previousNode()` operations, which corresponds to document-order and reverse document-order, respectively. We observe that SDOM is 40% faster than Xerces, on average. As expected, the gap grows for larger files, e.g. `XCDNA.xml`.

In Figure 7.11, we show the result of a document-order and reverse document-order traversal using the DOM navigational operations; SDOM is always within a factor of two of Xerces, but equals or improves upon Xerces for `XCDNA.xml`. We observe that traversals appear equally fast in document-order or reverse order.

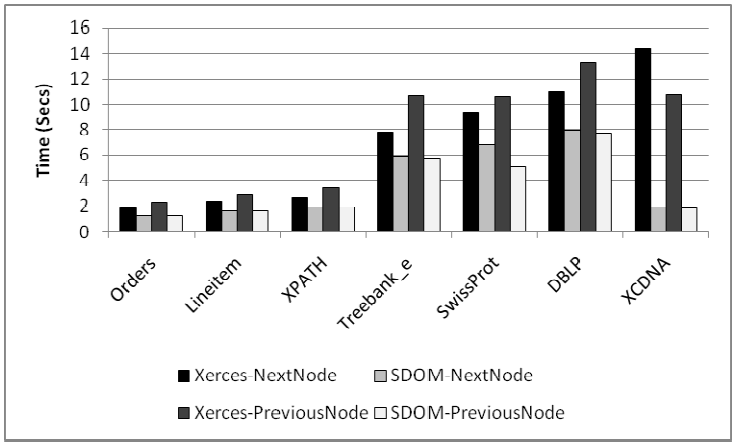


Figure 7.10 - Running times, document-order and reverse document-order traversals gathering basic statistics, of Xerces and SDOM using `nextNode()` and `previousNode()` operations. Average time of a single traversal reported for `XCDNA.xml`.

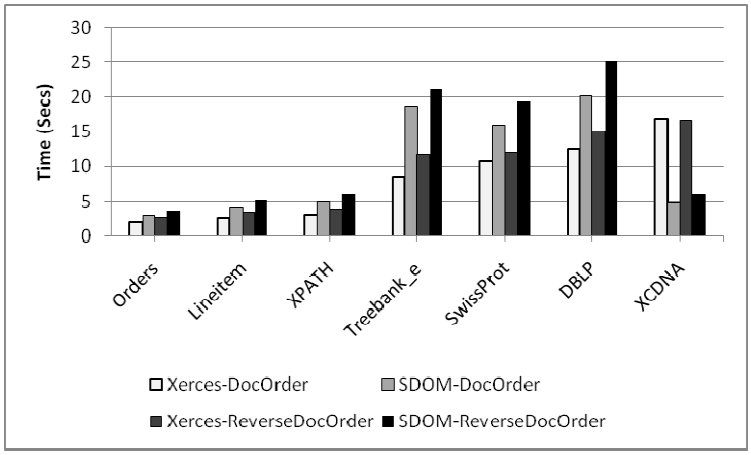


Figure 7.11 – Running times, for document-order and reverse document-order traversals using DOM navigation, with basic statistics for Xerces and SDOM. Average time of a single traversal reported for `XCDNA.xml`.

Table 7.4 – Running times for Xerces and SDOM for 'upward path enumeration'. Time results in seconds. SDOM slowdown wrt Xerces. Average time of a single traversal reported for `XCDNA.xml`

File	#Nodes	%non-leaf nodes	Max depth	Xerces	SDOM	Slowdown
Orders	300003	50%	4	0.08	0.13	1.64
Lineitem	2045954	50%	4	0.55	1.28	2.33
XPATH	2522571	33%	6	0.80	2.52	3.16
Treebank_e	7312613	33%	37	3.22	9.84	3.05
SwissProt	10599084	29%	6	2.71	8.76	3.23
DBLP	10595379	33%	7	2.97	7.90	2.66
XCDNA	25221153	33%	8	24.50	30.72	1.25

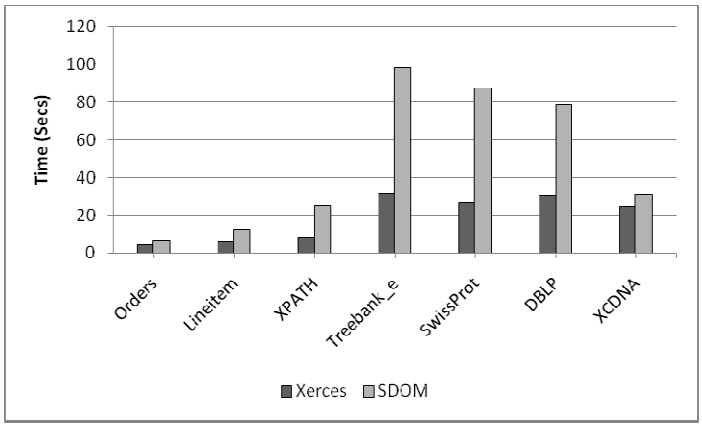


Figure 7.12 – Running times of Xerces and SDOM for 'upward path enumeration' gathering basic statistics. Average time of a single traversal reported for `XCDNA.xml`.

Figure 7.12 and Table 7.4 show the results for an upward path enumeration traversal. This traversal makes a very heavy use of the `parent()` operation which is (relatively) inefficient in SDOM, and may be considered a "worst case" for SDOM. Even here, SDOM on average was only a factor of 2.5 slower than Xerces.

We show the results of a full-test document-order traversal for our XML files in Figure 7.13, and Table 7.5. In Figure 7.13, we observe that even SDOM-CT with the slow DOM navigation is only a few times slower than Xerces, for small files, and for our largest file, the gap starts narrowing rapidly. Particularly noteworthy is the time of SDOM (using the `nextNode()` operation) on `XCDNA.xml`, which is nearly 3.5 times faster than Xerces.

Table 7.5 - Full test using `TreeWalker`. Shows running times in seconds for Xerces using tree navigation operations, and using `nextNode()`, versus SDOM using tree navigation and `nextNode()` and SDOM-CT using tree navigation. Time results in seconds. Average time of a single traversal reported for all files.

File	#Nodes	Xerces TreeNav	Xerces NextNode	SDOM TreeNav	SDOM NextNode	SDOM-CT TreeNav
Orders	300003	0.06	0.05	0.09	0.06	0.22
Lineitem	2045954	0.37	0.32	0.64	0.39	1.22
XPATH	2522571	0.44	0.40	0.82	0.51	1.68
Treebank_e	7312613	1.40	1.25	2.85	1.57	8.51
SwissProt	10599084	1.70	1.48	3.28	2.30	7.78
DBLP	10595379	1.86	1.67	3.56	2.28	10.45
XCDNA	25221153	17.63	16.88	8.50	5.42	27.90

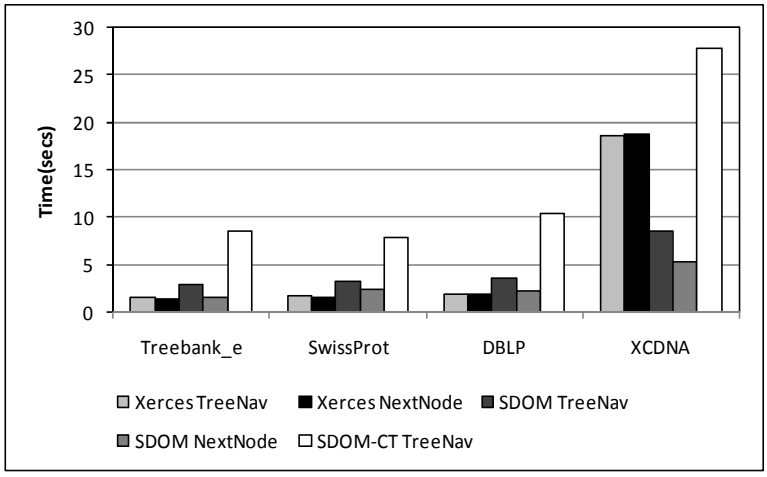


Figure 7.13 – Average running times for DOM full test including examination of attributes and substring test on text and attribute node values.

In Figure 7.14, we show the scalability of SDOM on XMark files with different file sizes. The results are of a document-order traversal on the files. We run the full DOM test using Xerces and SDOM. We observe Xerces is faster, but for a file size of 512MB SDOM is faster than Xerces, as Xerces uses more memory than what is available in main memory, and so (slower) virtual memory is used.

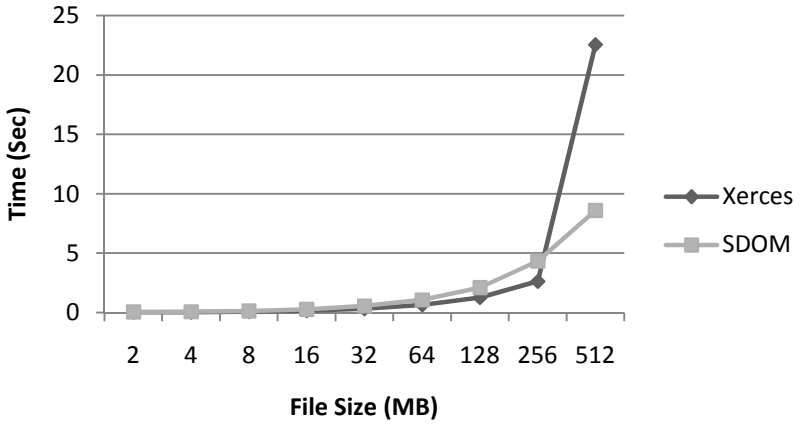


Figure 7.14 - Running times for DOM full test including examination of attributes and substring test on contents of `text` and `attribute` nodes for XMark files (sizes 2MB-512MB). Average times are reported.

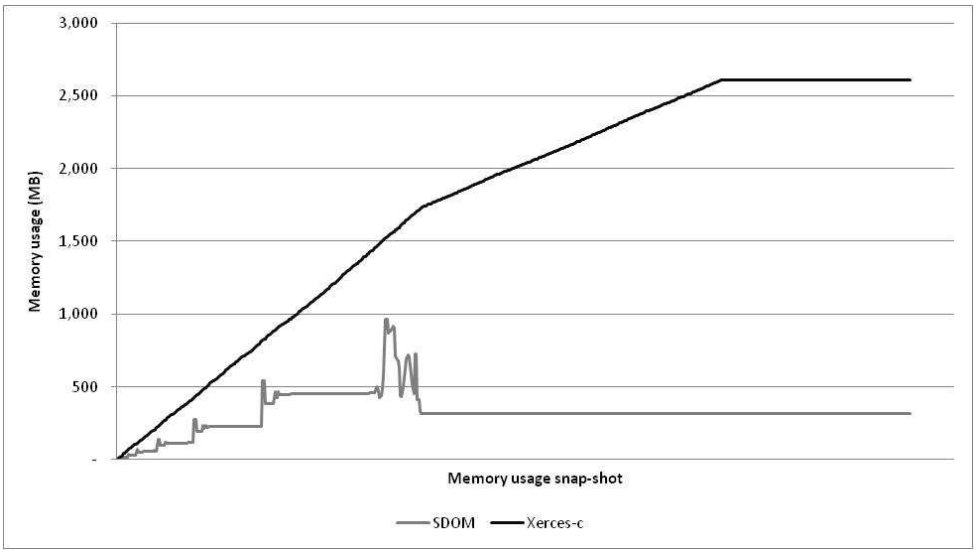


Figure 7.15 – Valgrind Massif profiler [65]: SDOM vs Xerces parsers, using `XCDNA.xml` (594MB).

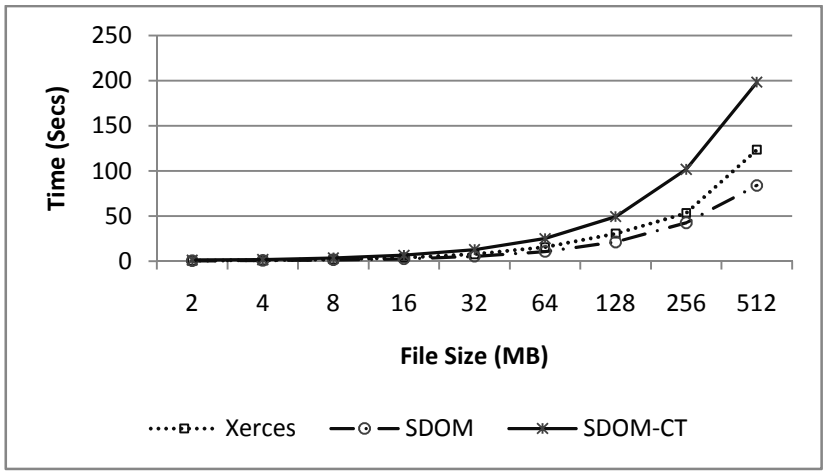


Figure 7.16 – Construction time of SDOM-(CT) vs Xerces using the XMark files.

7.3.4 Pre-processing Performance

We used the Valgrind Massif profiler tool [65] to measure the (heap) memory usage of SDOM compared to Xerces. We show the results of Massif for the XML document `XCDNA.xml`, in Figure 7.15 (the full set of Massif results for all our XML files used in this chapter can be found in [62]); we include the construction phase parsing, although we have not spent any time discussing the parsing of XML documents and the construction of subsequent SDOM representations. We confirm that the estimations made by the formulas in Section 4.1.6 broadly follow the Massif result[5] in Table 7.3. Optimisations still need to be made to the parser of SDOM; therefore, we have some irregular growth during the construction phase causing SDOM space usage to reach above the size of the file before we reach the final state, which is smaller than the file size. We observe Xerces exceeded the main memory of this machine, which has 2GB of RAM and is over four times larger than the XML file.

In Figure 7.16, we show the scalability of the SDOM(-CT) parser on the XMark files, reporting the construction speed. SDOM(-CT) requires a single parse of the XML document to construct the internal data structures. The results include the time for SDOM to create an intermediate representation, which is then converted to the final

[5] We see up to 10% difference on some files.

memory-efficient representation. In the Xerces construction phase time is spent allocating memory for each node in the tree and associated objects. We observe as the file size gets bigger, the main memory begins to run out and the slower virtual memory is used. In Figure 7.16, we observe that SDOM on average is 1.37 times faster than Xerces. In contrast, SDOM-CT on average is 1.80 time slower than Xerces; this slowdown was due to the compression of the text.

7.4 Summary

SDOM is a fast in-memory representation of XML documents with a small memory footprint. The current implementation is close to being a plug-in replacement for a standard DOM implementation in any application that does not require dynamic changes to the XML document, with very little penalty in terms of CPU usage. It is therefore not only suitable for handling moderately large (a few GB) size documents on standard PCs, but may also be useful for enabling the use of XML on devices with limited resources, such as smart cards or handheld computers. SDOM is built upon the succinct data structures introduced in Chapter 4 and engineered in Chapter 5 and 6. There has been a great deal of interest in the algorithms community in the theory of succinct data structures, and implementations of full-text indices that are based upon succinct data structures (see e.g. [29]). These appear to be somewhat unknown to the database community. We believe that the data structures we use could also be applied to other XML compressors.

Our comparison to Xerces is based on textual data that is uncompressed; SDOM uses significantly less space than the original file. A variant, *SDOM-CT*, compresses the textual data, and achieves compression ratios that are competitive with "query-friendly" compressors, but worse than the best XML compressors (see details in Table 7.3). Yet, SDOM-CT compares surprisingly well concerning compression performance, because:

- If one uses BZip and relatives as the compression algorithm, then in most cases, BZip does pretty well even relative to specialized compression algorithms applied to containers.

- When using BZip, the benefit of grouping text is limited in most cases.

The SDOM software library can be downloaded at [62].

Chapter 8
Conclusion

The main objective of this thesis was to represent XML documents space efficiently in memory and efficiently support DOM's navigational and access operations upon the space-efficient representation. We achieved this by using succinct, or highly space-efficient, data structures, which were pioneered by Jacobson [44]. We modified existing succinct data structures for use in representing XML documents, and carefully put the succinct data structures together to create a space-efficient DOM implementation. Our implementation, SDOM has processing speed that is comparable to Xerces, but the space usage is much lower; the space usage of SDOM-CT is comparable to query-friendly XML compressors, but the speed is much faster. The performance of SDOM(-CT) is shown graphically in Figure 8.1. For the query-friendly compressors we estimate the DOM processing time on the graph based upon their conclusions.

8.1 Technical Contributions

We made a number of technical contributions summarised below.

Tree representations

We studied several succinct tree representations and optimised them for DOM support. These optimised representations number the nodes of an n-node tree with integer from 1 to n, and (recall that previous representations numbered nodes non-consecutively with numbers from 1 to $2n$) have fast implementations for testing whether a node is a leaf. The main new idea introduced was double numbering, and the partitioned representation for the `LOUDS` bit-string. The idea of the partitioned representation has been applied to bit-strings by [37].

Textual data structure

Strategies to represent and access efficiently the large amount of textual data in XML documents were studied. The textual data could be represented naively, where the text nodes are concatenated into a string. Alternatively, we could compress the concatenated string using FM-Index [48] or blocked text compressed using BZip2.

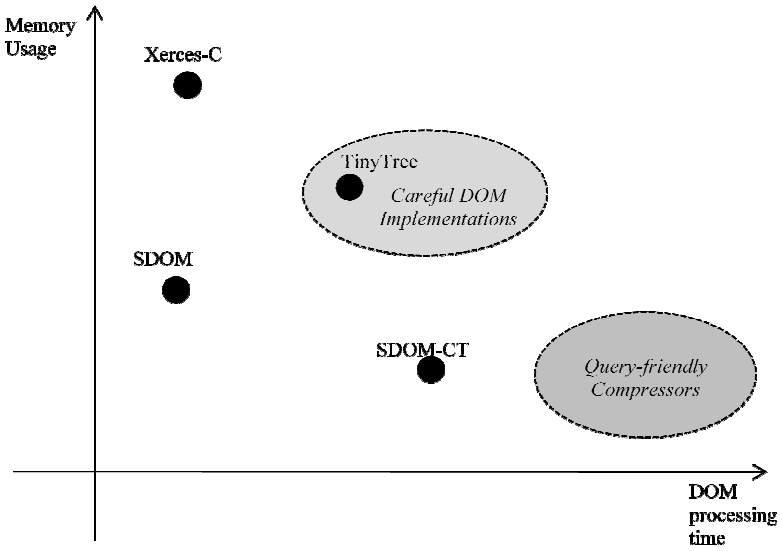


Figure 8.1 – DOM performances graph.

Based upon the experimental evaluation, we found that there was no significant difference in the BZip2 compression of text arranged in document-order compared to text in path-order (the method commonly used by XML compressors such as XMill). As a result, we organise text in document-order in the representations.

Access to the individual text nodes required the storing of offsets into the concatenated string. Storing offsets naively would have a significant space cost, particularly if text is stored compressed.

A careful experimental evaluation of various alternatives to store offsets was done. Our results show significant reduction in space usage costs, with 7-8 bits required per offset, instead of the 32 bits per offset for a naive representation. Accessing an offset was very fast. This was achieved by formulating the offset problem as the prefix-sum problem. The best data structure targeted the succinct prefix sums bound; we showed (both experimentally and mathematically) a close relationship between the succinct bound on prefix sums and the data-aware $GOLOMB$ measure. The $GOLOMB$ measure had been shown to be the best for storing offsets in certain IR applications [23].

SDOM Implementation

We bring together the succinct data structures, including the tree representation (the PAREN+ variant) and the textual data structure as building blocks of the DOM implementation (SDOM). Additional improvements were made in SDOM, which allowed, e.g., faster support for traversals in DOM, efficient implementation of the element-attribute mapping and space-efficient solution to represent fully qualified element names. We include extensive experimental tests on SDOM.

8.2 Future Work

There are a number of tasks and open questions that remain. Firstly, SDOM, as described, can only be used for static documents. Dynamising succinct data structures is an area of active research (see e.g. [58]), but it is far from clear how to implement a full DOM with dynamic operations. Secondly, although loading an XML document is fast (it needs to be – our traversal tests take so little time that reading in the XML file would otherwise be a serious bottleneck in our experiments) and does not take anywhere near the amount of memory required by a standard DOM parser, we have not made a serious attempt at optimising either the speed or the memory usage of parsing. Finally, in addition to the tests that we have performed, it would be very interesting to wrap SDOM in an application such as Xalan or Saxon, and investigate its performance therein.

Appendix A
Experimental Setup

Xerces DOM

We use the Xerces-C v2.8 C++ [67] DOM implementation to parse the XML files, gather statistics and construct the internal data structures.

Data structures

The implemented data structures were in the C++ programming language.

Running Time

The specification of the test machines used for the experiments on the data structures are as follows:

- **Intel-P4**: Dual processor Pentium 4 machine, with 2GB RAM of memory, dual core 3.4 GHz CPUs and a 2MB L2 cache, running Ubuntu 6.06 Linux. The compiler was g++ 3.3.5 with optimisation level 2.
- **Sun-UltraSparc**: Sun UltraSparc-III machine, with 8GB RAM of memory, a 1.2GHz CPU and an 8MB L2 cache, running SunOS 5.9. The compiler was g++ 3.3.2 with optimisation level 2.

Appendix B

DOM methods supported by SDOM

Document

Returns	Method	DOM Level	Related SDOM component
Attr	createAttribute(String name)	1	NA
Attr	createAttributeNS(String namespaceURI, String qualifiedName)	2	NA
CDATASection	createCDATASection(String data)	1	NA
Comment	createComment(String data)	1	NA
Document-Fragment	createDocumentFragment()	1	NA
Element	createElement(String tagName)	1	NA
Element	createElementNS(String namespaceURI, String qualifiedName)	2	NA
EntityReference	createEntityReference(String name)	1	NA
Processing-Instruction	createProcessingInstruction(String target, String data)	1	NA
Text	createTextNode(String data)	1	NA
DocumentType	getDoctype()	1	Document
Element	getDocumentElement()	1	Document
Element	getElementById(String elementId)	2	AttributeDS
NodeList	getElementsByTagName(String tagname)	1	NameCodeDS
NodeList	getElementsByTagNameNS(String namespaceURI, String localName)	2	NameCodeDS
DOM-Implementation	getImplementation()	1	
Node	importNode(Node importedNode, boolean deep)	2	NA
String	getActualEncoding()	3	Document
void	setActualEncoding	3	Document
String	getEncoding	3	Document
void	setEncoding(String enc)	3	NA
void	getStandalone	3	Document
String	getVersion	3	Document
void	setVersion(string version)	3	NA
String	getDocumentURI()	3	Document
bool	getStrictErrorChecking	3	Document
void	setStrictErrorChecking	3	NA
void	renameNode(Node n, String uri, String name)	3	NA
Configuration	getDOMConfiguration()	3	

Node

Returns	Method	DOM Level	Direct SDOM component supported
Node	appendChild(Node newChild)	1	NA
Node	cloneNode(boolean deep)	1	NA
NamedNodeMap	getAttributes()	1	AttributeDS
NodeList	getChildNodes()	1	STree
Node	getFirstChild()	1	STree
Node	getLastChild()	1	STree
String	getLocalName()	2	NameCodeDS
String	getNamespaceURI()	2	NameCodeDS
Node	getNextSibling()	1	STree
String	getNodeName()	1	NameCodeDS
short	getNodeType()	1	NameCodeDS
String	getNodeValue()	1	TextDS
Document	getOwnerDocument()	1	STree
Node	getParentNode()	1	STree
String	getPrefix()	2	NameCodeDS
Node	getPreviousSibling()	1	STree
boolean	hasAttributes()	1	AttributeDS
boolean	hasChildNodes()	1	STree
Node	insertBefore(Node newChild, Node refChild)	1	NA
boolean	isSupported(String feature, String version)	2	Document
void	normalize()	2	NA
Node	removeChild(Node oldChild)	1	NA
Node	replaceChild(Node newChild, Node oldChild)	1	NA
void	setNodeValue(String nodeValue)	1	NA
void	setPrefix(String prefix)	2	NA
short	compareTreePosition(Node other)	3	STree
String	getTextContent() - missing minority nodes	3	TextDS
void	isSameNode(Node other)	3	Not implemented
String	lookupPrefix(String uri, bool usedefault)	3	Not implemented

NodeList

The NodeList here is an interface class. We give details of the class designed for `attribute` nodes.

Returns	Method	DOM Level	Direct SDOM component supported
Int	getLength()	1	STree
Node	item(int index)	1	STree

NamedNodeMap

The namedNodeMap here is an interface class. We give details of the class designed for `attribute` nodes.

Returns	Method	DOM Level	Direct SDOM component supported
Int	getLength()	1	AttributeDS
Node	getNamedItem(String name)	1	AttributeDS
Node	getNamedItemNS(String namespaceURI, String localName)	2	AttributeDS
Node	item(int index)	1	Attribute
Node	removedNamedItem(String name)	1	NA
Node	removedNamedItemNS(String namespaceURI, String localName)	2	NA
Node	setNamedItem(Node arg)	1	NA
Node	setNamedItemNS(Node arg)	2	NA

Attribute

Returns	Method	DOM Level	Direct SDOM component supported
String	getName	1	AttributeDS
Element	getOwnerElement	2	AttributeDS
Bool	getSpecified	1	NA
String	getValue	1	TextDS
void	setValue	1	NA
TyeInfo	schemaTypeInfo	3	Not supported
boolean	isID	3	AttributeDS

Document Type

Returns	Method	DOM Level	Direct SDOM component supported
NamedNodeMap	getEntities()	1	DocType
String	getInternalSubset()	2	Not Supported
String	getName()	1	NameCodeDS
NamedNodeMap	getNotations()	1	DocType
String	getPublicId()	2	DocType
String	getSystemId()	2	DocType

Element

Returns	Method	DOM Level	Direct SDOM component supported
String	getAttribute(String name)	1	AttributeDS
Attr	getAttributeNode(String name)	1	AttributeDS
Attr	getAttributeNodeNS(String namespaceURI, String localName)	2	AttributeDS
String	getAttributeNS(String namespaceURI, String localName)	2	AttributeDS
NodeList	getElementsByTagName(String name)	1	STree, NameCodeDS
NodeList	getElementsByTagNameNS(String namespace URI, String localName)	2	NameCodeDS
String	getTagName()	1	NameCodeDS
boolean	hasAttribute(String name)	1	AttributeDS
boolean	hasAttributeNS(String namespaceURI, String localName)	2	AttributeDS
void	removeAttribute(String name)	1	NA
Attr	removeAttributeNode(Attr oldAttr)	1	NA
void	removeAttributeNS(String namespaceURI, String localName)	2	NA
void	setAttribute(String name, String value)	1	NA
Attr	setAttributeNode(Attr newAttr)	1	NA
Attr	setAttributeNodeNS(Attr newAttr)	2	NA
void	setAttributeNS(String namespaceURI, String qualifiedName, String value)	2	NA

Entity

Returns	Method	DOM Level	Direct SDOM component supported
String	getNotationName()	1	DocType
String	getPublicId()	1	DocType
String	getSystemId()	1	DocType

Notation

Returns	Method	DOM Level	Direct SDOM component supported
String	getPublicId()	1	DcoType
String	getSystemId()	1	DocType

ProcessingInstruction

Returns	Method	DOM Level	Direct SDOM component supported
String	getData()	1	TextDS
String	getTarget()	1	AttributeDS
void	setData(String data)	1	NA

Text

Returns	Method	DOM Level	Direct SDOM component supported
Text	splitText(int offset)	1	NA
bool	iselementContentWhitespace()	3	Not Supported
String	wholeText()	3	Not Supported

DOM-Implementation

Returns	Method	DOM Level	Direct SDOM component supported
Document	createDocument(String uri, String qualifiedName, DocType doctype)	3	NA
DocType	createDocType(String qualifiedName, String pubId, String sysId)	3	NA
bool	hasFeature(String feature, String version)	3	Document

Bibliography

[1] Antoshenkov, G. 1997. Dictionary-based order-preserving string compression. *The VLDB Journal* 6, 1 (Feb. 1997), pp. 26-39. DOI= 10.1007/s007780050031

[2] Apache XML Project. http://xml.apchae.org

[3] Arion, A., Bonifati, A., Manolescu, I., and Pugliese, A. 2007. XQueC: A query-conscious compressed XML database. *ACM Trans. Inter. Tech*. vol. 7, issue 2 (May. 2007), pp. 10. DOI=10.1145/1239971.1239974

[4] B+ Tree. (2008, September 13). In Wikipedia, The Free Encyclopedia. Retrieved 21:09, September 15, 2008, from http://en.wikipedia.org/wiki/B%2B_tree

[5] Benoit, D., Demaine, E. D., Munro, J. I., Raman, R., Raman, V. and Rao, S. S. 2005. Representing trees of higher degree. *Algorithmica*, vol. 43, issue 4, (2005), pp. 275-292. DOI= 10.1007/s00453-004-1146-6

[6] Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., and Teubner, J. 2006. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA, June 27 - 29, 2006). SIGMOD '06. ACM, New York, NY, pp. 479-490. DOI=10.1145/1142473.1142527

[7] Buneman, P., Grohe, M., and Koch, C. 2003. Path queries on compressed XML. In *Proceedings of the 29th international Conference on Very Large Data Bases - Volume 29* (Berlin, Germany, September 09 - 12, 2003). J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, Eds. Very Large Data Bases. VLDB Endowment, pp. 141-152.

[8] Buneman, P., Choi, B., Fan, W., Hutchison, R., Mann, R., and Viglas, S. D. 2005. Vectorizing and Querying Large XML Repositories. In *Proceedings of the 21st international Conference on Data Engineering* (April 05 - 08, 2005). ICDE. IEEE Computer Society, Washington, DC, pp. 261-272. DOI=10.1109/ICDE.2005.150

[9] Burrows, M., Wheeler, D. 1994. A block sorting lossless data compression algorithm. *Technical Report 124*, Digital Equipment Corporation.

[10] Busatto, G., Lohrey, M., and Maneth, S. 2005. Efficient Memory Representation of XML Documents. In *Database Programming Languages, 10th International Symposium, proceedings* (Trondheim, Norway, August 28-29, 2005). DBPL 2005. LNCS, Springer-Verlag, Berlin Heidelberg New York, NY, vol. 3774, pp. 199-216. DOI= 10.1007/11601524_13

[11] BZip2. http://www.bzip.org

[12] CenterPoint DOM Implementation. http://www.cpointc.com/XML/ (note: not delivered anymore as open source)

[13] Cheney, J. 2006. Tradeoffs in XML Database Compression. In Proceedings of the 2006 IEEE Data Compression Conference, (Vancouver, Canada, 2006). DCC 2006. IEEE Computer Society Press, Ls Alamitos, California, 2006, pp. 392-401. DOI=10.1109/DCC.2006.79

[14] Cheng, J., Ng, W., 2004. XQzip: Querying Compressed XML Using Structural Indexing. In *Advances in Database Technology - 9th International Conference on Extending Database Technology, proceedings* (Heraklion, Crete, Greece, March 14-18, 2004). EDBT '04. LNCS, Springer-Verlag, Berlin Heidelberg New York, NY, vol. 2992, pp. 219-236. DOI= 10.1007/b95855

[15] Clark, D. R. 1996. *Compact Pat Trees*. PhD thesis, University of Waterloo.

[16] Clark, D. R. and Munro, J. I. 1996. Efficient suffix trees on secondary storage. In *Proc. of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms* (Atlanta, Georgia, United States, January 28 - 30, 1996). SODA '96. Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, pp. 383-391.

[17] J. Cleary and I. Witten. 1984. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, Vol. 32, issue 4, pp. 396-402.

[18] CPU Cache. (2008, September 13). In Wikipedia, The Free Encyclopedia. Retrieved 21:09, September 15, 2008, from http://en.wikipedia.org/wiki/CPU_cache

[19] Computer Architecture. (2008, September 12). In Wikipedia, The Free Encyclopedia. Retrieved 21:09, September 15, 2008, from http://en.wikipedia.org/wiki/Computer_architecture

[20] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). *Introduction to Algorithms*, first edition, MIT Press and McGraw-Hill. ISBN 0-262-03141-8.

[21] Crimson DOM implementation. http://xml.apache.org/crimson/

[22] Delpratt, O., Rahman, N., and Raman, R. 2006. Engineering the LOUDS Succinct Tree Representation. In *Proc. of 5th Workshop on Experimental Algorithms* (Menorca, Spain, May 24-27, 2006) WEA ’06. LNCS, Springer-Verlag, Berlin Heidelberg New York, NY, Vol. 4007, pp. 134-145. DOI = 10.1007/11764298_12.

[23] Delpratt, O., Rahman, N., and Raman, R. 2007. Compressed Prefix Sums. In *SOFSEM 2007*: *Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science* (Harrachov, Czech Republic, January 20-26, 2007). SOFSEM ’07. LNCS, Springer-Verlag, Berlin Heidelberg New York, NY, Vol. 4362, pp. 235-247. DOI = 10.1007/978-3-540-69507-3_19.

[24] Delpratt, O., Raman, R., and Rahman, N. 2008. Engineering succinct DOM. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology* (Nantes, France, March 25-29, 2008). EDBT '08, vol. 261. ACM International Conference Proceeding Series, New York, NY, pp. 49-60. DOI = 10.1145/1353343.1353354.

[25] DocBook. http://www.docbook.org

[26] DOM4j. http://www.dom4j,org

[27] Elias, P. 1974. Efficient storage retrieval by content and address of static files. *J. ACM*, 21 (1974), pp. 246–260. DOI = 10.1145/321812.321820.

[28] eXist-db. http://exist.sourceforge.net/

[29] Ferragina, P. and Manzini, G. 2001. An experimental study of an opportunistic index. In *Proc. of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms* (Washington, D.C., United States, January 07 - 09, 2001). Society for Industrial and Applied Mathematics, Philadelphia, PA, pp. 269-278.

[30] Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S. 2006. Compressing and Searching XML Data Via Two Zips. In *Proc. of the 15th International World Wide Web Conference* (Edinburgh, Scotland, May 23 - 26, 2006). WWW '06. ACM Press, New York, NY, pp. 751-760.

[31] FM-Index. http://pizzachili.dcc.uchile.cl/indexes/FM-indexV2/

[32] Frick, M., Grohe, M., and Koch, C. 2003. Query Evaluation on Compressed Trees (Extended Abstract). In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science* (June 22 - 25, 2003). LICS. IEEE Computer Society, Washington, DC, pp. 188.

[33] Galax XQuery Implementation. http://www.galaxquery.org/

[34] Geary, R. 2005. Private communication.

[35] Geary, R. F., Raman, R., and Raman, V. 2006. Succinct ordinal trees with level-ancestor queries. *ACM Trans. Algorithms*, vol 2, issue 4 (Oct. 2006), pp. 510-534. DOI=10.1145/1198513.1198516

[36] Geary, R. F., Rahman, N., Raman, R., and Raman, V. 2006. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.* Vol. 368, issue 3 (Dec. 2006), pp. 231-246. DOI=10.1016/j.tcs.2006.09.014

[37] Golynski, A., Grossi, R., Gupta, A., Raman, R., Rao, S. S. On the Size of Succinct Indices. In *Proc 15th Annual European Symposium on Algorithms* (Eilat, Israel, October, 2007). ESA ’07. LNCS, Springer-Verlag, Berlin Heidelberg New York, NY, vol. 4698, pp. 371-382.

[38] Gottlob, G. and Koch, C. 2002. Monadic datalog and the expressive power of Web information extraction languages. *J. ACM* 51, 1 (Jan. 2004), 74-113. DOI= http://doi.acm.org/10.1145/962446.962450.

[39] Grossi, R. and Vitter, J. S. 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proc. 32nd Annual ACM Symposium on Theory of Computing* (Portland, Oregon, United States, May 21 - 23, 2000). STOC '00. ACM, New York, NY, pp. 397-406. DOI=10.1145/335305.335351

[40] Grossi, R., and Vitter, J. S. 2004. Private communication.

[41] Gupta, A., Hon, W., Shah, R., and Vitter, J. S. 2007. Compressed data structures: Dictionaries and data-aware measures. *Theor. Comput. Sci.* Vol. 387, issue 3, (Nov. 2007), pp. 313-331. DOI=10.1016/j.tcs.2007.07.042

[42] Gupta, A., Hon, W. K., Shah, R., and Vitter, J. S. Compressed dictionaries: space measures, data sets, and experiments. 2006. In *Proc. 5th International Workshop on Experimental Algorithms* (Menorca, Spain, May 2006). WEA '06. LNCS, Springer-Verlag, Berlin Heidelberg New York, NY, vol. 4007, pp. 158–169.

[43] Gzip. http://www.gzip.org

[44] Jacobson, G. 1989. Space-efficient static trees and graphs. In *Proc. of the 30th Annual Symposium on the Foundations of Computer Science* (NC, USA, 10/30/1989 - 11/01/1989). FOCS '89. IEEE Computer Society, Washington, DC, vol. 225, Cat.No.89CH2808-4, pp. 549-554, 1989.

DOI=10.1109/SFCS.1989.63533

[45] JDOM. http://www.jdom.org

[46] Kay, Michael. 2006. Optimization in XSLT and XQuery. In Proc. *XMLPrague: a conference on XML* (Prague, Czech Republic, June 17-18, 2006). ITI Series Vol. 2006-294, pp. 29-41, 2006. Link= http://iti.mff.cuni.cz/series

[47] Kim, D. K., Na, J. C., Kim, J. E., and Park, K. 2005. Efficient implementation of rank and select functions for succinct representation. In *Proc. of the 4th Workshop on Experimental Algorithms* (Santorini Island, Greece, May 10-13, 2005). WEA 2005. LNCS, Springer-Verlag, Berlin Heidelberg New York, NY, Vol. 3503, pp. 315-327. DOI= 10.1007/11427186_28.

[48] Liefke, H. and Suciu, D. 2000. XMill: an efficient compressor for XML data. 2000. In *Proc. of the ACM SIGMOD international Conference on Management of Data* (Dallas, Texas, United States, May 15 - 18, 2000). SIGMOD '00. ACM Press, New York, NY, Vol. 29, Issue 2, pp. 153-164. DOI=10.1145/342009.335405

[49] Martin, H. W. and Orr, B. J. 1989. A random binary tree generator. Proc. of the 17th ACM Annual Computer Science Conference, pp. 33–38, 1989.

[50] Medline. http://www.nlm.nih.gov/mesh/gcmdoc2004.html

[51] Mehta, D. P., Sahni, S. (Ed.). 2004. *Handbook of Data Structures and Applications*. Chapman & Hall/CRC publishers. ISBN = 1584884355.

[52] Min, J., Park, M., and Chung, C. 2006. A compressor for effective archiving, retrieval, and updating of XML documents. *ACM Trans. Internet Technol.* 6, 3 (Aug. 2006), 223-258. DOI=10.1145/1151087.1151088.

[53] Munro, J. I. Tables. 2006. In *Proceedings of the 16th Conference on Foundations of Software Technology and theoretical Computer Science* (December 18 - 20, 1996). V. Chandru and V. Vinay, Eds. LNCS, vol. 1180. Springer-Verlag, London, pp. 37-42.

[54] Munro, J. I. and Raman, V. 2002. Succinct representation of balanced parentheses and static Trees. *SIAM J. Comput.* 31, 3 (Mar. 2002), pp. 762-776. DOI =10.1137/S0097539799364092

[55] Neumüller, M. and Wilson, J. N. 2003. Improving XML Processing Using Adapted Data Structures. In *Revised Papers from the Node 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems* (October 07 - 10, 2002). LNCS, Springer-Verlag, Berlin Heidelberg London, vol. 2593, pp. 206-220.

[56] Ng, W., Lam, W., Wood, P. T., and Levene, M. 2006. XCQ: A queriable XML compression system. *Knowl. Inf. Syst.* 10, 4 (Oct. 2006), pp. 421-452. DOI=10.1007/s10115-006-0012-z

[57] Raman, R., Raman, V., and Rao, S. S. 2002. Succinct indexable dictionaries with applications to encoding *k*-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, California, January 06 - 08, 2002). SODA ‘02. Society for Industrial and Applied Mathematics, Philadelphia, PA, pp. 233-242.

[58] Raman, R. and Rao, S. S. 2003. Succinct dynamic dictionaries and trees. In *Proc. Automata, Languages and Programming, 30th International Colloquium* (Eindhoven, The Netherlands, June 30 - July 4, 2003). ICALP ‘03. LNCS, Springer-Verlag, Berlin Heidelberg New York, NY, vol. 2719, pp. 357-368. DOI=10.1007/3-540-45061-0_30

[59] Sadakane, K. and Grossi, R. 2006. Squeezing succinct data structures into entropy bounds. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithm* (Miami, Florida, January 22 - 26, 2006). SODA '06. ACM, New York, NY, pp. 1230-1239. DOI= 10.1145/1109557.1109693

[60] SAX Parser. http://www.saxproject.org/

[61] Saxon. http://saxon.source.forge.net/

[62] SDOM Software Libraries. http://hdl.handle.net/2381/3363

[63] Shkarin, D. 2002. PPM: One Step to Practicality. In *Proceedings of the Data Compression Conference (DCC '02)* (April 02 - 04, 2002). DCC. IEEE Computer Society, Washington, DC, pp. 202.

[64] Tolani, P., Haritsa, J.R. 2002. XGRIND: A Query-Friendly XML Compressor. In *Proc. 18th International Conference on Data Engineering* (February 26 - March 01, 2002). ICDE. IEEE Computer Society, Washington, DC, pp. 225.

[65] Valgring Massif Profiler Tool. http://valgrind.org/

[66] Xalan XSLT Processor The Apache XML project. http://xml.apache.org/xalan-c/

[67] Xerces C++ Parser. http://xerces.apache.org/xerces-c/

[68] Xerces Java 2 Parser. http://xerces.apache.org/xerces2-j/

[69] XIndice. http://xml.apache.org/xindice/FAQ

[70] XMark - XML Benchmark Project. http://monetdb.cwi.nl/xml/

[71] XPath. http://www.w3.org/TR/xpath

[72] XSLT. http://www.w3.org/TR/xslt

[73] UW XML Repository. http://www.cs.washington.edu/research/xmldatasets/www/repository.html

[74] VOTable Documentation. http://www.us-vo.org/VOTable/

[75] Wang, F., Li, J., and Homayounfar, H. 2007. A space efficient XML DOM parser. *Data Knowl. Eng.* 60, 1 (Jan. 2007). Elsevier Science Publishers B. V. Amsterdam, The Netherlands. pp. 185-207. DOI=10.1016/j.datak.2006.01.002

[76] Witten, I., Moffat, A., Bell, I. *Managing Gigabytes, 2e*. Morgan Kaufmann, 1999.

[77] W3C DOM API documentation. 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/

[78] W3C DOM Traversal, 2000. http://www.w3.org/TR/DOM-Level-2-TraversalRange/traversal.html

[79] W3C XML Specification. http://www.w3.org/TR/REC-xml/

[80] Zhang, N., Kacholia, V., and Özsu, M. T. 2004. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proc. 20th international Conference on Data Engineering* (March 30 - April 02, 2004). ICDE. IEEE Computer Society, Washington, DC, pp. 54.