Efficient Processing of Very Large XML Documents in Small Space

by

Matthew K. Meyer

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

2012

©2012

Matthew Kent Meyer

All Rights Reserved

This manuscript has been read and accepted for the Graduate Faculty in computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy

Ira Rudowsky

Date

Chairman of Examining Committee

Ted Brown

Date

Executive Officer

Distinguished Professor Theodore Raphan

Professor Abdullah Tansel

Dr. Mudhakar Srivatsa

The City University of New York

Abstract

Efficient Processing of Very Large XML Documents in Small Space

by

Matthew K. Meyer

Adviser: Professor Ira Rudowksy

The focus of this research was to develop a highly efficient serialized data structure for use in Document Object Model (DOM) query systems of Extensible Markup Language (XML) documents, which are increasingly the dominant model for data representation across the World Web. The space efficiency of the serialized data structure developed is shown to improve query response time and eliminates the need to rebuild the DOM representation each time it is desired to query an existing XML document. Moreover, this serialized data structure enables DOM modeling of extremely large XML documents at a low and fixed memory cost. New algorithms and software tools for working with these data structures have been developed and this allows for compatibility with standard DOM modeling. The structures, tool and techniques presented can be readily adapted for use on the Semantic Web as it becomes a more prominent feature of the internet.

The improved performance of XML database storage and retrieval application developed in this thesis helps close the gap between relational database application performance and XML Document (NXD) database application performance. In addition, the novel DOM querying technique that we have developed in this thesis has potential application on hand held devices, autonomous robots, and other devices which have limited memory.

The research evolved from an examination of three principal limitations that restrict the widespread implementation of Semantic Web technologies. These limitations include: (1) Querying information stored in XML format. Representation in XML format is a hierarchical and queries are less efficient and more time consuming than comparable queries of data stored in relational database format [1] . (2) Complexity of tools and technologies. The suite of technologies, useful ontology's, and tools necessary for creating semantic web technologies are complex and development time is extensive [2]. (3) Insufficient user friendly interfaces. Users with no prior exposure to semantic technologies, such as “Description Logic (DL),” cannot easily utilize the power of semantic search capabilities. In this thesis, the first of the inhibiting factors restricting the growth of the semantic web will be addressed. It will be shown that serializing the data representation of an XML document representation enables significant improvement of XML-DOM query response times. This serialized representation also eliminates the restrictions on documents size imposed by most XML- DOM query systems

Chapters 2 and 3 introduce and motivate the research described in this thesis. Chapters 4, 5 and 6 explain the basic costs inherent in DOM modeling, expand on the influence of DOM tree size and shape on the total cost of DOM modeling, and enumerate why DOM modeling of a XML documents is traditionally considered to take 2 to 5 times the size of the source XML document to store the DOM tree representing the document in memory. Chapters 7 and 8 introduce methods for reducing the size of DOM trees and establish upper and lower bounds for DOM tree size in relation to source XML documents; we establish a practical upper-bound of memory required for

supporting XML DOM querying and show that it is smaller than the 2-5 size limitation described above. Chapter 9 introduces the Minimum DOM Node (MDN), a minimized data structure capable of storing DOM node information. It is also shown how an MDN Array (MDNA) can be used as an alternative to a traditional DOM tree. Chapter 10 examines the memory cost of using an MDNA for DOM modeling. It is also shown that using the MDNA model of an XML document allows DOM querying with a low and fixed allocation of memory. Chapter 11 details the process and costs involved in creating an MDNA from a given source XML document and is a major contribution of this work. Chapter 12 examines the W3C specification for the DOM interface and explains how an MDNA can be used to satisfy the requirements of core DOM node operations. In Chapter 13, a C/C++ implementation of a MDNA parser was developed to prove the concepts presented in this thesis. Chapter 14 examines the future research topics that follow logically from these developments.

Acknowledgements

I would like to acknowledge every organization and individual who has contributed to my financial support during the course of this research. Without the financial support that I have received this dissertation would not exist. I would like to express my deepest gratitude to Professor Ted Brown of the CUNY graduate center and to all Executive Officers of the Computer Science Department of the Graduate Center of CUNY past and present as well to Dr. Aaron Tenenbaum and Professor Yedidyah Langsam, Chairman of the Department of Computer and Information Science at Brooklyn College for their support. The support I received from the CUNY Graduate Center and from Brooklyn College have been invaluable. I would also like to thank Professor Ira Rudowksy and Professor Theodore Raphan for the financial support that they have been able to extend to me and also thank the National Science Foundation and the NSF-GK12 program for the support I received through them as part of a grant awarded to Brooklyn College (Grant No. DGE-0638718).

I would like to thank my advisor Professor Ira Rudowksy as well as Professor Abdullah Tansel and Dr. Mudhakar Srivatsa for their help, advice and for permitting the freedom to pursue my own interests and ideas in deciding on a topic for research.

I extend a heartfelt thanks to Professor Theodore Raphan for his advice, encouragement and most especially for the many hours of his time he has given in hammering this thesis into its present form. His contributions to my research and to this document have been absolutely invaluable and I am truly grateful.

Finally I would like to thank my wife Kelly and my children Tabitha, Maggie and Robert without whom all of this would mean nothing.

Table of Contents

Abstract .... **IV**

List of Illustrations .... **XI**

1. Introduction **1**

2. Background and Motivation **5**

2.1. Semantic Web and XML 5

2.2. Global Problem of Incorporating Semantic Web Technology 6

2.3. XML Data Storage 7

2.4. Querying Data in XML 10

2.5. Space Efficiency of XML Documents 15

3. The Basic Cost of DOM Modeling **20**

3.1. Cost of storing text information 21

3.2. Cost of storing DOM tree structure 21

3.3. Cost of node data structures 22

4. DOM Tree size in relation to XML document size **25**

4.1. A lower bound for the node count of an XML DOM tree created from a source XML document of a given size n (characters) 25

4.2. An upper bound for the node count of an XML DOM tree created from a source XML document of a given size n (characters) 26

5. Node Arrangement and the Effect on DOM Cost **35**

5.1. Vertical (Bar) DOM Tree Arrangements 35

5.2. Horizontal (Beam) DOM Tree Arrangements 37

5.3. Hidden Costs in Traditional DOM 39

6. Reducing the Number and Size of DOM Tree Nodes 41

6.1. Indexing DOM Node Data, with Data Offsets 41

6.2. Reusing Data Indexes 43

7. Upper and Lower Bounds on DOM Tree Size 45

8. New DOM Data Structures 49

8.1. A Simplified Node Object 49

8.2. A minimum DOM (MDN) node structure 51

8.3. The MDN Array (MDNA) 55

9. DOM Methods Cost Comparisons 57

9.1. Cost of a traditional DOM Tree Representation 58

9.2. Cost of a DOM Tree Using Data Indexes 59

9.3. Cost for DOM Using MDN Arrays 61

9.4. Average Cost Using MDN Arrays 64

10. Creating the MDN Array (MDNA) 67

10.1. Creating a MDN Array from an Existing DOM Tree 67

10.2. Creating a MDN Array Directly from the Source XML 68

10.3. Efficiency Issues When Creating the MDN Array 82

11. Supporting Dom Queries with an MDNA 87

11.1. W3C IDL Dom Interface Specifications 88

11.2. W3C DOM Node (Tree) Navigation Operations 90

11.3. W3C DOM Node (Text) Retrieval Operations 92

11.4. W3C Dom Node Modification Operations 93

11.4.1. Updating Operations 94

11.4.2. Deletion Operations 97

11.4.3. Insertion Operations 98

12. C/C++ Implementation of an MDNA Parser 99

12.1. Structure of the C program 100

12.1.1. STACKHandler.c and STACKHandler.h 100

12.1.2. MDNAHandler.c and MDNAHandler.h 101

12.1.3. DOMHandler.c and DOMHandler.h 102

12.1.4. main.c 103

12.2. Flow Diagram of the Program Organization 105

12.3. DOM and Location Path Query Support 106

12.4. Program Performance 109

13. Future Research 110

13.1. Implementation of an MDNA parser device with limited memory 110

13.2. Full support of W3C DOM using a MDNA program written in C++ 110

13.3. Adapt. MNDA to support DOM Level 3 Load and Save Specification 111

13.4. Support for multi-threading during queries 111

13.5. Benchmarking MDNA versus other DOM parsers 112

13.6. Support for XPath and XQuery 112

13.7. Closure under TAX 113

14. Summary and Major Contributions of this Thesis 114

Appendix A: Document Object Model Level 1 Core IDL Interfaces . 116

Appendix B: Logical DOM View (CORE) Level One ……………........... 124

Appendix C: W3C IDL node interface specification version 1.0. 125

Appendix D: Limited API for C MDNA Parser 127

- **Data Structures ……………………………………………………………… 127**
- **DOMHandler.h …………………………………..……………………………… 131**
- **MDNAHandler.h ………………………………………………………..……… 137**
- **STACKHandler.h ………………………………..…………………………… 143**

Bibliography ………………………………………………………….………. 147

List of Illustrations

Fig. 2.1 A simple XML document 10

Fig. 2.2 A hierarchical tree constructed from previous XML document 10

Fig. 2.3 Small section of very large XML document showing a text element of interest 12

Fig. 2.4 Generalized tree structure outline for a very large (XMark generated) XML document 13

Fig. 2.5 A comparison of three common data file types and their size requirements 17

Fig. 3.1 Another simple XML document 20

Fig. 3.2 Logical tree construct based on document shown in Fig. 3.1 22

Fig. 3.3 Generalized DOM tree node structure in C 23

Fig. 4.1(A) Document-centric XHTML document 31

Fig. 4.1(B) XHTML Document rendered as a XML DOM tree 31

Fig. 5.1(A) Text representation of a “bar” XML document 36

Fig. 5.1(B) The resulting logical representation of a "bar" DOM tree 36

Fig. 5.2(A) Text representation of a “beam” model XML document 38

Fig. 5.2(B) Logical node representation of a "beam" DOM tree 38

Fig. 6.1 Generalized DOM tree node structure in C 42

Fig. 6.2 Three related datum, recoverable using one data index 44

Fig. 7.1(A) Simple XML Document 45

Fig. 7.2(B) Associated DOM tree for simple XML document 45

Fig. 8.1 Simplified classical node class 49

Fig. 8.2 A Minimum DOM Node (MDN) 52

Fig. 8.3 An MDN structure with values set 54

Fig. 9.1 A document-centric XML file with a text node that cannot be easily referenced from a node data_location value 57

Fig. 9.2(A) A traditional node structure 59

Fig. 9.2(B) Illustration of the source document (Fig 9.1) rendered as a traditional DOM tree 59

Fig. 9.3(A) A modified node structure using a data_location variable 60

Fig. 9.3(B) Illustration of the source document (Fig 9.1) rendered as a modified DOM tree 60

Fig. 9.4(A) The MDN structure (uses a data_location variable) 62

Fig. 9.4(B) An illustration of the MDNA created from the source document shown in Fig 9.1 62

Fig. 10.1 Structures and variables needed to create the MDN array 70

Fig. 10.2 Source document showing data_location values 73

Fig. 10.3 System state at position 190 in the document 73

Fig. 10.4 'SwissCheese' sort example 76

Fig. 10.5 Illustration of a breaking node and the elements that will be moved to the end of the document because of the breaking node 76

Fig. 10.6 MDNA state after the sort 80

Fig. 10.7 MDNA state after sibling status determined 80

Fig. 10.8 MDNA state while parent_location values are converted to first_child values 82

Fig. 10.9 Stack array size and content after each call to START() function indicating out of place nodes 84

Fig. 11.1 Target XML document showing data_location values and the new inserted node 96

Fig. 12.1 Output from the MDNA parser help screen 104

Fig. 12.2 Flow diagram of MDNA parser program system components . 105

Fig. 12.3 Interactive Mode - Instructions Screen 107

Fig. 12.4 Query Mode - Instructions Screen 108

1. Introduction

Extensible Markup language (XML) has emerged as the Internet standard format for representing and exchanging data [3]. Because the XML standards specification requires that XML documents be readable by humans, XML uses a structured hierarchical data representation schema, which encapsulates each piece of data with an easily readable tag [4]. As a result of this structure, XML has become a de facto standard exchange format for exchanging data between different database and business to business (B2B) applications. It is also becoming the preferred format for heterogeneous data representation and storage, such as XHTML Web pages and documents created by office productivity tools such as Microsoft Office [5]. XML is also a cornerstone technology of the proposed Semantic Web [6].

The popularity and wide-spread use of XML among a diverse set of organizations has engendered a rethinking of the storage and retrieval practices for data. Most early XML storage practices relied on mappings and transformations between XML data trees and relational database tuples within a conventional Relational Database Management Systems (RDBMS). As a consequence, a single business operation might require numerous translations of the data from XML to relational table formats and vice versa at a significant cost in speed, reliability and efficiency of representation. These inefficiencies have become for many organizations a core data management issue [1, 7]. As a result, organizations are increasingly moving towards native XML document storage solutions, which would require the development of a database system that is purely XML based [8]. The goal with Native XML document storage solutions and Native XML databases (NXD) is to eliminate unnecessary duplication of data, as well as the

overhead generated by repeated transformations between two different storage mediums, with all the risk for error that that entails.

The problem of dealing efficiently with large XML documents and large XML document collections is twofold. First, there is a significant storage problem as XML documents tend to be approximately twenty times as large as the same documents represented in either Comma Separated Version (CSV) or Fixed Separated Version (FSV) formats [9], which are alternate methodologies for storing structured information. A second problem of dealing efficiently with documents in XML format is that there are numerous difficulties when attempting to efficiently perform queries, updates and transformations of data in these very large XML documents.

Two models for parsing XML documents exist. The Event-Driven approach exemplified by the Simple Application Program Interface for XML (SAX) parser has efficient parsing performance for single pass operations, but is limited in the face of repeated queries [10]. The SAX parsing methodology is also not an official World Wide Web Consortium (W3C) standard and is difficult for novice programmers to successfully implement [1, 7]. The parsing scheme used in the Documentation Object Model (DOM) is a W3C standard and in addition has two W3C standard query languages (XPath and XQuery) that are in widespread use within NXD's. However, DOM parsing requires the creation of a tree data structure in memory (DOM tree) that is usually 2-5 times greater than the source document that it is based on, since the DOM tree contains additional structure like nodes and links necessary define the tree. This makes DOM modeling of an XML document, as it is normally implemented, impractical for very large XML documents, or large collections of XML documents [8].

Most presently used DOM parsers such as Xerxes or Xindice are unable to handle XML documents larger than 75MB because to transform this size XML document to a DOM tree would require as much as 375 Mbytes [11]. This limitation of DOM parsers inhibits the sue of DOM and advanced languages on very large XML documents as well as on low resource systems such as small hand held devices. Some cost analysis has been done on the space efficiency of XML documents [12], but little work has been done on the breakdown of what makes DOM modeling so expensive in the general case. Alternative methods for storing DOM tree information have been proposed, and compared but all of these systems rely on large costly database management systems to support DOM, XPath or XQuery operations [13]. Simple and very low cost methodology for supporting DOM and XPath queries is explored in this thesis.

In this thesis, we examine the costs associated with traditional XML-DOM tree modeling. We address the question of how to efficiently enable DOM in a small memory space in three ways: (1) We analyze the storage cost of creating a DOM tree from a given XML document and establish upper and lower bounds for DOM tree storage and the number of operations necessary to create the DOM tree in a predefined memory space (2) We derive a fundamentally different, but practical method for reducing the memory size needed for creating and storing DOM trees. (3) We compare traditional methodologies of loading and querying information stored in DOM trees with those developed in this thesis. The methods we developed eliminate excessive nodes and reduce the size of the nodes in the DOM representation without compromising retrieval ability. Moreover the reduced DOM tree constructed is easily to serialize making it a

practical solution for the W3C DOM 3.0 "Load and Save" specification [14].

2. Background and Motivation

2.1 Semantic Web and XML

The World Wide Web (Web) has profoundly altered the way in which we communicate, collaborate and conduct business. Unfortunately, in its current form the Web has limitations that severely affect the sharing of information between applications, platforms and laboratories or organizations, i.e., enterprise boundaries. Most information available on the Web takes the form of text formatted exclusively for the use by humans, which is accessible primarily through keyword based search engines such as Google and Yahoo. A major limitation of existing search engines is that they are extremely sensitive to vocabulary and unable to extract context from complex queries. An inherent lack of formal structure in the Web prevents its content from being easily handled by these search agents [15]. Queries such as (1) finding documents based on words/phrases defined by a specific context; (2) locating information based on complex queries that require a priori background knowledge; (3) locating and using information contained within data repositories; (4) finding and using “web services” are all beyond the capabilities of current search engines.

The Semantic Web is a proposed evolution of the World Wide Web in which the content of web sites have their information and semantics formally structured and defined. A cornerstone of this proposed Semantic Web structure is that all resources, such as words, phrases, pictures, objects, (etc.) are classified within a formal, sharable structure such as XML. These classifications in turn are given additional meaning

through a shared lexicon of relationship information known as ontology. These ontologies may also be implemented using XML. A wide spread adoption of the Semantic Web would provide a common framework to allow data to be shared and reused across application, platform and enterprise boundaries. It would also make it possible for intelligent agents to access, understand and manipulate web content as readily as humans do; without requiring these agents to have human level intelligence [15] [16].

There are a large number of semantic web projects already in place, and more are being added each year [17]. But all successful Semantic Web projects so far have been limited to specific fields and industries. As of yet , the power of semantic technologies have not yet found broad implementation in the Web as a whole, and thus the Semantic Web remains an unrealized goal [18]. The focus of this thesis is to address a key limitation of current semantic web implementation, which is its ability to efficiently handle XML data, particularly large XML documents when memory space is limited.

2.2 Global Problem of Incorporating Semantic Web Technology

Examining the possible reasons that semantic technologies have not been incorporated into more of the Web, two specific deficiencies stand out. The first is creating and implementing a simple ontology, i.e., establishing relationship between objects can be remarkably difficult and time-consuming. Research shows that individuals creating ontologies for first time, even using specialized software (OWL,

Sapphire) frequently fail to make effective ontologies [2]. A second and perhaps more fundamental deficiency is the consistently poor performance of XML parsers, and query systems, which is considered a severe liability in real world applications [1]. Together these deficiencies impact retrieval efficiencies and leads to poor response time and unresolvable queries in semantic systems.

The three most common ontology languages, RDF, RDFS and OWL, all have methodologies to express their contents using XML, which is also commonly used as a data storage and exchange format [19] [20]. Therefore improvements in XML parsing and querying capability would benefit not only retrieval of commonly formatted information but also efficiently handle ontologies stored in XML format. The latter tend to be excessively large documents, because of the tremendous overhead in expressing even simple information when creating ontologies. In this thesis we focus on developing algorithms for improving XML parsing and storage. However these techniques could be used to effectively manage ontologies as well as other data that is has a structured markup language representation.

2.3 XML Data Storage

The dominant model for storing and accessing data in computer systems for the past thirty years has been the relational model [21]. In the relational model, data are typically accessed and viewed in a table format, but this format does not necessarily bear any resemblance to how the data are actually stored in a computer's hardware. Users of a RDBMS do not need to know how the data are physically stored in order to

access it.

When addressing the problem of poor XML parser performance, it is necessary to first understand that XML documents use the hierarchical model of data storage, where data are stored and accessed in a tree-like structure. The hierarchical model requires users to have knowledge of how data are stored and any modification to the stored data requires equivalent changes in the retrieval system.

XML is a structured file format specification that contains both text and meta-data information, or markups, which describes that text. It has been used to create documents to store, and transport atomic units of information that are defined by their markups. The Unicode characters which comprise an XML document can be divided into two types: the meta-data, which is the markup and 2) the text, which is the content. All markup components in an XML document either begin with the character "<" and end with a ">", or begin with the character "&" and end with a ";". Strings of characters which are not markup are assumed to be content. XML documents also have several key logical components defined by markup structures:

Tags from the bulk of the markup content in an XML document and are delimited by the markup symbols "< >". Tags can be divided into three categories. Start-tags have a name surrounded by markup symbols “< >” and an end-tag has a slash preceding the tag name. An example is the library tag. The start-tag is <library>, and the end-tag is </library>. Empty content tags feature a trailing slash under a pair of markup symbols such as <library /> and are a combination of a start and end tag.

Elements are logical units of information in an XML document and are defined by the tags that enclose them. All elements either begin with a start-tag and end with a

matching end-tag, or consist only of an empty-element tag. All data between a start-tag and an end-tag are defined as element content. Element content can include other elements which are then called child elements. An example of an element is <Title>For Whom the Bell Tolls</Title>, where <Title> and </Title and the start and tags and “For Whom the Bell Tolls” is the element content.

Attributes are components that can be added to start tags or empty element tags for purposes of clarity and meta-information storage. They consist of a name/value pair. An example would be <book instock="no">. This signifies that the tag named book should be understood to not be in stock, so that the elements need not be examined as it is not in stock.

An **XML tree** is an abstraction of an XML document. It must have one and only one root element, and all elements must be properly nested. That is, if an element is not the root element, it must be completely enclosed by some other element’s start and end tag. Together, tags, elements and attributes create a logical hierarchical structure, with all elements (except for the root element) having one (and only one) parent element which encloses them. If an element encloses another element, the enclosed element is considered to be the child element of the element that contains it. Elements that share a parent are referred to as sibling elements. Likewise if an element A has a child element B which in turn has a child C, C can be referred to as A's descendant. And if an element A has a parent B and B also has a parent C, then C can be said to be A's ancestor.
An XML document’s logical structure of parents, children and siblings can be represented as a rooted tree of elements. An example document (Fig 2.1.) and related logical tree (Fig 2.2) are shown below help to illustrate this concept.

```xml
<library>
	<book genre="fiction">
		<title lang="en">I Robot</title>
		<author>Isaac Asimov</author>
		<year>1950</year>
		<price>300.00<price>
	</book>
</library>
```

Fig. 2.1 A simple XML document

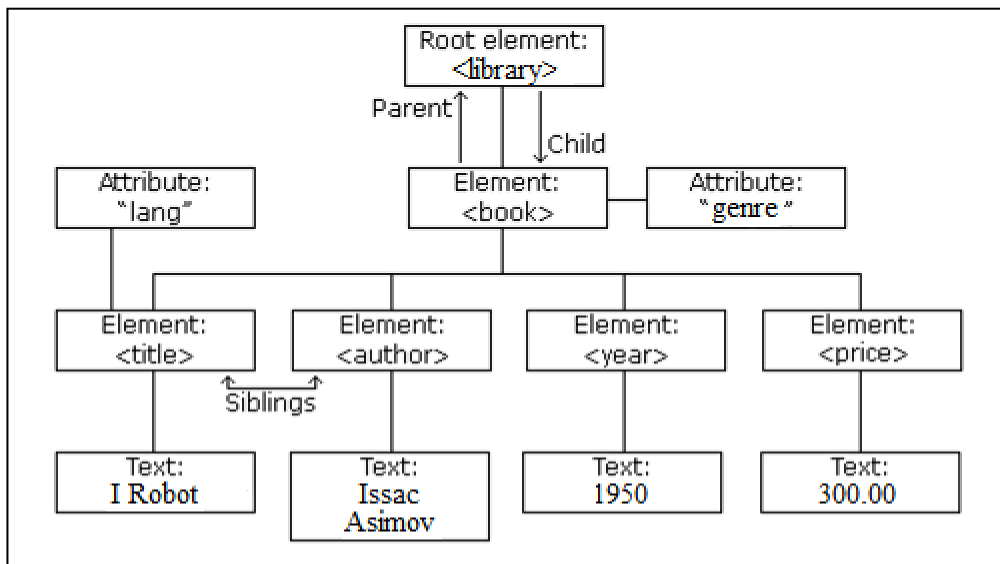


Fig 2.2 A hierarchical tree constructed from the previous XML document

2.4 Querying Data in XML

There are essentially two ways to query or parse an XML document: the event-

based parser approach, more commonly referred to as the SAX (Simple Application Programming Interface (API) for XML) approach and the Document Object Model (DOM) approach [22, 23]. Both have advantages and disadvantages, and most of the major XML parsers support both SAX and DOM. However, there are a few popular and common parsers that only support SAX, and several others that only support their own proprietary API like ElectricXML and XMLPull parser.

In the event-driven or **SAX** approach, the parser reads through the document and calls pre-declared procedures to process atomic units of information such as the tags, attributes, and elements of the document. The major advantage in using SAX is that it can be implemented so that the source document is parsed in sections and does not need to be loaded into memory in its entirety. This allows SAX to perform queries utilizing a small memory space and even parse a document larger than the system's own memory allotment.

The SAX approach is very fast for simple translations, transformations and single pass queries. As currently implemented in SAX 2.0, a SAX reader/parser does not view a XML document as an XML tree (or similar memory contained data structure), but as a stream of events generated by the parser. The parser is able to recognize the following elements/events:

- the start of the document
- the end of the document
- the start tag of an element
- the end tag of an element
- character data

- a processing instruction

A SAX parser, scanning an XML file from start to end, looks for corresponding functions to invoke, as each event is encountered. These event-handling functions (also called callback methods) are defined by the programmer.

An example of the inefficiency of repeated queries using SAX can be seen below in Fig 2.3, which shows a section of a very large XML document used for test purposes. The associated logical tree structure can be seen in Fig 2.4. Supposing that we made repeated requests for the text data (content data) associated with the description of closed auction data. We can express what we are looking for in slash notation to make it clearer: "/site/closed_auctions/closed_auction/annotation/description/text," (Fig 2.4 shaded regions)

Using SAX we would need to define (or override) the functions for handling start and end tags (which will allow us to locate the proper place in the tree) and the character data function (to allow us to return the information we are interested in).

No matter how many times we repeat this query, it will always incur the same (time and space) cost because each query must parse the entire proceeding document (2,035,115 lines) before reaching the section of interest.

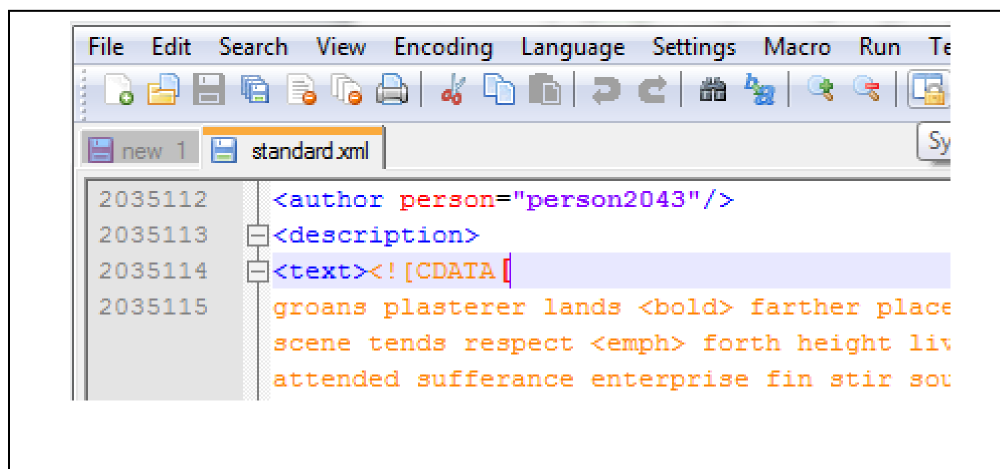


Fig 2.3 Small section of very large XML document showing a text element of interest.

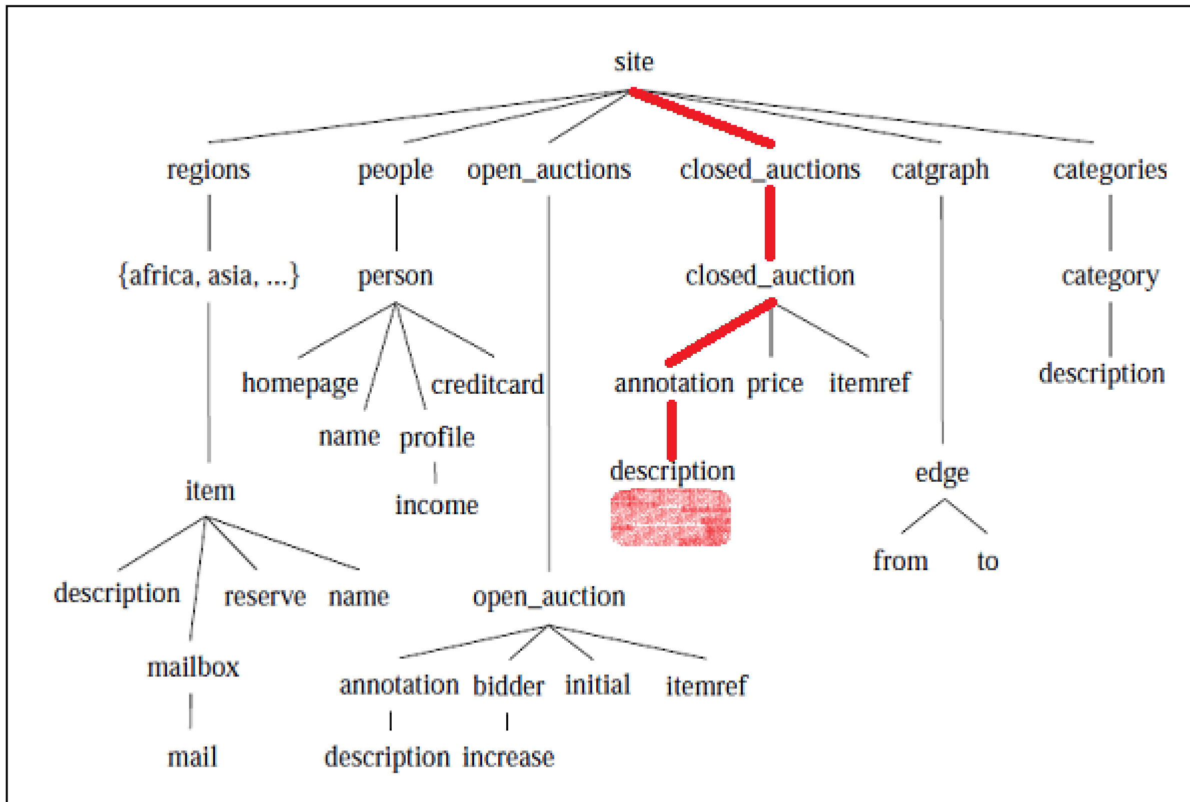


Fig 2.4 Generalized tree structure outline for a very large (XMark generated) XML document. Each node of the tree represents an element in the document. The red line shows the path to the node (element) of interest.

SAX performance breaks down quickly in the face of complex or repeated queries. Each SAX query comprises a complete depth first search (DFS) of the underlying document tree and repeated queries referencing information at the end of the document will gain no benefit from the results of previous queries. Each query will take just as long as previous queries and a critical inefficiency in SAX-based querying.

In contrast to the SAX approach, the **DOM** approach reads and loads an entire XML document into memory as a collection of objects representing the elements of the document. These objects are linked by pointers in memory, which represent the parent

child relationship inherent in the document. Together, these objects and links constitutes a tree structure and often represent a complete duplication of the content of the original document. It may also contain additional links and structures necessary for random access to the underlying data.

The major advantage of the DOM model is that it allows immediate and random access to the XML document’s content. The DOM model also allows support for high level query languages such as XPath and XQuery. This DOM structure also supports and has led to the development of several algebras and calculi that support reasoning over XML trees [24, 25]. The major inefficiency in the DOM approach is that constructing the tree is time-consuming and utilizes large amounts of memory. The tree construct created in memory can be 2-5 times larger than the source document. As a result, most practical applications that support the DOM model are incapable of processing documents larger than a 50-100MB.

Native XML database management systems (NXD's) have been developed to close the performance gap between classical relational database systems and XML database systems. NXD’s use a variety of standard techniques to transform XML documents into collections of DOM trees that can be queried for information and facilitate working with XML in its native format [8, 26]. NXD’s are large programs and frequently use additional approaches to attempt to accelerating query processing when using high-level query languages like XQuery. Two common strategies applied by NXD’s include the use of holistic twig join algorithms [27, 28] which help find specific subtrees (twigs) during querying, and node and tree labeling schemes which help determine the relationships between any two nodes in linear time [29, 30]. NXD’s

facilitate querying XML documents using high level language, however they are invariably large programs that consume considerable resources and represent an added layer of complexity between the document data and the user [31].

The developments in this thesis address utilize simple data structures to provide a program that is confined to a small memory space, but which also provides an interface that adheres to the DOM specification. In addition this program can directly query extremely large XML documents (exceeding 1 GB).

2.5 Space Efficiency of XML Documents

XML documents are characterized by descriptive tags, which are repeated throughout the document. These tags improve readability by humans and facilitate system semantic sharing capabilities by clearly identifying the elements in the text. Thus, XML has a self-describing file structures. However, this XML overhead structure vastly increases file size compared to the actual document content [12]. The overhead of the included tags in the XML format makes it very space inefficient, particularly for regular (homogenous) data sets.

Lawence et al (2004), quantize the extra cost associated with XML documents versus a variety of other file formats. Other common used file structures have externally defined schemas and the tags are absent. A simple file structure for storing data is the comma separated values (CSV) file format, which stores data values between commas and the meaning of each data value is specified by some external header file. Lines in the CSV represent rows of a table, and commas in each line separate what are fields in

each individual table row. This format can be used to store/exchange data in a RDBMS provided that the Schema for the table is also included or sent separately. Likewise, the fixed size values (FSV) format refers to a set of file formats, again in plain text, where all the fields (or all attributes fields in a given column) are of a fixed length. If the data are shorter than the value assigned to a field, the places are held by spaces to keep the length at the size specified.CSV has an "Overhead per Attribute" (OV) cost of 1a where 'a' is the number of attributes in the document. FSV files have an OV cost of (S-D) where S is the Schema size definition (detailing the size of each field) and D is the average size of the data values in each field. In contrast, XML has an OV cost of (2N +5)a where N denotes the average size of tag names in the document.

In Fig. 2.5 we see the comparative size needed to store an identical unit of information, a single book entry for a library, in the 3 different formats previously discussed. It should be noted that both CSV and FSV require an additional external schema file for use with the data file, which is shown, while for the XML document the basic schema information is part of the tags which are incorporated into the file. The inclusion of the schema files for the FSV and CSV representations may make the 3 formats seem comparable in size, particularly when used to store only a single entry. However it should be clear that as the number of book entries grow of that the single schema table needed by the FSV and CSV formats will be all that is needed regardless of how many actual book records are created while the tags used by the XML format will need to be repeated for each and every book entry.

Table Schema for CSV

```
CREATE TABLE bookstore
(Category char(25),
Title char(25),
Author char(25),
Year char(25),
Price char(25) )
```

Data File for CSV

```
cooking,Everyday Italian,Giada De Laurentiis,2005,30.00
```

Table Schema for FSV

```
CREATE TABLE bookstore
(Category char(25),
Title char(25),
Author char(25),
Year char(25),
Price char(25) )
```

Data File for FSV

```
             cooking    Everyday Italian Giada De Laurentiis               2005                30.00
```

XML File (struture/schema is part of file)

```
<bookstore>
<book category="cooking">
<title lang="en">Everyday Italian</title>
<author>Giada De Laurentiis</author>
<year>2005</year>
<price>30.00</price>
</book>
</bookstore>
```

Fig 2.5 A comparison of 3common data file types and their size requirements.

The efficiency difference between CSV and FSV and XML comes about because as data is increased, the number of tags in XML structured files will increase. For CSV and FSV the schema file is fixed and the data are just increased. The CSV and FSV files also cannot leave out NULL values, meaning even if an attribute in a particular row has no data, it must still include a comma or an empty field to delineate its location. By contrast the XML document can be considered to have its Schema built-in (self-documenting), although additional restrictions on what is and is not allowable in the document can also be put into a separate XML Schema Document (XSD). XML

documents can also leave out NULL attributes as their data structure is hierarchical in nature.

The space efficiency of XML dramatically decreases if tag names are long strings. Shortening the tag name length increases space efficiency, but this comes at the expense of human readability. Understanding this tradeoff is critical as XML is designed to be processed by both humans and machines. In general FSV and CSV files are more space efficient than XML files although the degree of efficiency varies wildly depending on factors such as: NULL attribute entries, average tag name length, average field name waste, and whether a document is mostly data (document or text centric) or mostly schema (data-centric).

It is worth noting at this point that Lawrence et al (2004) examine space efficiency principally in terms of using XML for data-exchange and not as a means for native database storage. Nevertheless their cost analysis for attribute overhead costs remain valid, with the addendum that in a database, data is not necessarily restricted to text (character) encoding and thus numerical data can be saved at greatly reduced cost as integers (doubles, etc.) as opposed to as character data. But in the case of character data many databases use CSV and FSV file formats as their native storage solution.

MyISAM is the default storage engine used by MYSQL [32] and uses two files to store the basic information of any given table. For an individual table, the '.frm' file contains information about the table structure — effectively, an internal representation of the 'CREATE TABLE' statement. The '.MYD' file contains the row data with minimal overhead and in the case of pure character data is essentially an FSV file.

It is also necessary to point out that although the data table for the FSV is larger

than the data table for the CSV, in a query situation it will actually be faster to pull individual items of data from the FSV then from the CSV. In order to retrieve the Author value from the 3rd row of a CSV file a query engine would need to parse the document until it had passed 12 commas, whereas in the FSV the query engine could merely calculate the offset (2 rows at 100 characters a row, and then two additional fields at 25 characters a field) of 250bytes and leap to that position.

The research in this project is focused on utilizing the XML format as a native storage mechanism. One way to increase the access efficiency of XML file access is to develop an indexing feature or offset ability, similar to that in FSV for accessing data in XML text files. This is one of the aims of this thesis. Another mechanism for improving the efficiency of utilizing native XML is decreasing the associated costs of DOM modeling, which is the major focus of the work.

3. The Basic Cost of DOM Modeling

To understand why the tree structure in DOM, which represents the XML document, is typically large and occupies about 2-5 times the size of the source document, it is necessary to break down how the DOM tree is usually constructed. The official W3C recommendation for XML DOM only specifies an interface for creating, accessing and manipulating XML files, it does not specify how the underlying data structure to represent the XML document is too be constructed. [23].

XML DOM views each element of the source document, including the text, as a node of the DOM tree. An example of a very simple XML document available on the WWW is shown in Fig.3.1 [33]. The document has been made as small as possible by removing all but one book record, and the name-space declarations that describe the type of XML document and are usually defined above the root element have also been omitted for simplicity.

```
<bookstore>
        <book category="cooking">
                <title lang="en">Everyday Italian</title>
                <author>Giada De Laurentiis</author>
                <year>2005</year>
                <price>30.00</price>
        </book>
</bookstore>
```

Fig 3.1 Another simple XML document

The DOM standard specifies a series of supported and accessible properties for accessing nodes and how they can be accessed using node relationships. For example,

the commands parent, child, and sibling describe the method for accessing particular nodes in relation to other nodes. We now consider why this simple XML representation is expanded to 2-5 times the size of the document when represented as a DOM tree by examining each component of the DOM representation.

3.1 Cost of storing text information

The document (Fig. 3.1) is composed of 168 1 byte (UTF-8) characters. To store the 6 unique element names requires 39 bytes: 33 characters and 6 null terminating characters for each character string: bookstore, book, title, author, year, and price. An additional 25 bytes are needed to store the attribute name and value pairs (category="cooking", lang="en") and 48 bytes are needed to store the actual text values of the nodes (Everyday Italian, Giada De Laurentiis, 2005, 30.00). Thus, a total of 112 bytes is needed to store all possible text information that we might want to retrieve. The additional 70 bytes in the source document that comprise the tag delimiters (< , >, /) are all OV (Overhead) bytes, which are used to indicate the structure of the XML document and are used to help create the links between nodes and are not directly included in the DOM tree implementation. Thus, storing the text of the XML document is not the major contributor of DOM tree size.

3.2 Cost of storing DOM tree structure

The approximately 2-5 increase of DOM size over the XML document size comes

from the cost of implementing the structure of the tree, which expresses the relationships (parent, child, and sibling) between nodes. The associated DOM tree [Fig 3.2] graph [33] of the XML document [Fig. 3.1] has 12 unique nodes. From each of those nodes we would like to be able to access any of the other nodes by following the links of the tree. We therefore would want both a parent and child pointer for each node (22 links). On a 32 bit system, it takes 4 bytes to create a pointer (the links between nodes). We would then need 8 bytes (parent and child pointers) per node to store these links that compose the structure of the tree. Thus, the cost of just representing the links is 176 bytes and is greater than the size of the entire original source document (168 bytes).

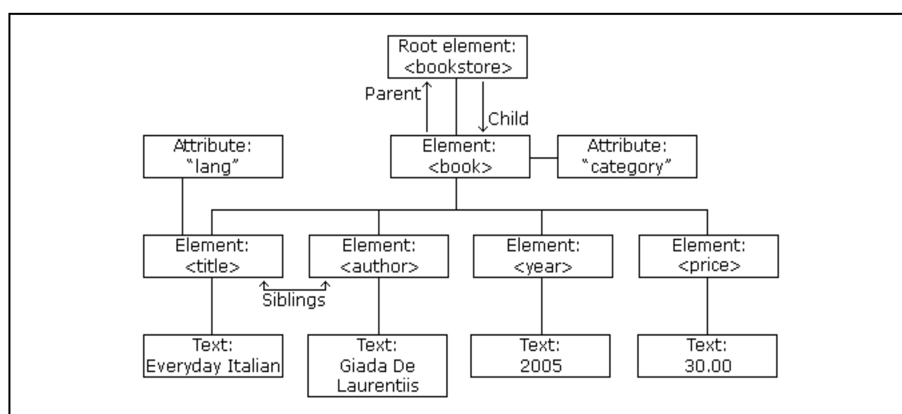


Fig. 3.2 Example logical tree construct based on document in Fig 3.1

3.3 Cost of node data structures

The nodes themselves can add even more cost to the DOM model, depending

on how they are implemented: 1) The W3C IDL interface specification [23] requires each node to have accessors for the following units of information: name, value, attributes. 2) We will not know until we begin parsing the document what the values will be for each required information unit and therefore in the general case will need to reserve space using pointers for values that may in fact not exist. 3) The attributes container used by the nodes is specified to use a mapping between name and value pairs, which will require an additional pointer (as part of the data structure) for each pair. 4) Nodes that have more than one child will need an array structure and a counter so that child nodes can be accessed in order (firstchild(), lastchild() etc.) Below (Figure 3.3.) is an example of a simple structure data declaration in C capable of storing the information necessary for each node according to the W3C DOM model.

```
struct node{
        char     * name;          // Pointer to name of the node or type of node
        char     * value;         // Pointer to value/type of the node. May be set to NULL
        map<char,int>  * attribute_map;       // Pointer to attribute map, may be NULL.

        node    * parent_link;            // Link to the parent
        node    * * child_links;          // Pointer to array of links to children

        int   childcounter;         // Integer for iterating through the child_links array
 } ; // sizeof(node) = 24.
```

Fig. 3.3 Generalized DOM tree node structure in C

The cost for each node using the structure above is 24 bytes. For the 12 nodes used to represent the document, this would require 288 bytes. Thus the total cost for the

DOM data tree would be greater than 544 bytes (112 + 144 + 288), which is 3.2 times the size of our original document of 168 bytes. The attribute maps would add an incremental extra cost in storage. It should be noted that the major cost in storage comes from data structures within the nodes and the associated costs of accessing the links between nodes. Moreover, different types of documents can have significant costs related to the internal arrangements of nodes. These extreme cases will be analyzed as they are useful for establishing upper bounds on DOM tree storage costs.

4. DOM Tree size in relation to XML document size

A XML DOM tree is created based on the form and structure of its source XML document. It should be clear that the number of elements in an XML document will directly affect the size of the DOM tree created to represent that document; each element in the XML document will become a node in the DOM tree. In this section we establish the minimum and maximum number of possible declarations (elements, entity references, etc.) that can be contained within an XML document of a given size n (characters). We use this information to derive lower and upper bounds on the count of the number of nodes that a DOM tree could contain C(n) when it is created from a source XML document of a given size n (characters). Finally we look at arrangements of elements in XML documents and how the shape of the resulting DOM tree can lead to a huge increase in storage costs if the content of the tree nodes are not chosen carefully.

4.1 A lower bound for the node count of an XML DOM tree created from a source XML document of a given size n (characters)

Determining the lower bound for the number of nodes needed to represent an XML document as a DOM tree is a trivial exercise. Assume that we have an XML document, *D*, whose source document is of size n characters. The count of the number of nodes necessary to represent that document as a DOM tree will be denoted by C(n). A lower bound for C(n) is:

$$C(n) = \Omega(1) \qquad \text{(Eq. 4.1)}$$

where $\Omega(1)$ is the usual notation for order 1. This expresses the idea that we cannot have a XML document if the document does not contain at least 1 element. That one element, the root element, would become one node (the root node) or our DOM tree.

4.2 An upper bound for the node count of an XML DOM tree created from a source XML document of a given size n (characters)

The upper bound for the number of nodes needed to represent an XML document as a DOM tree will occur when the source XML document contains the maximum number of elements possible. The source XML document will contain the maximum number of elements possible when each element is created using the fewest number of characters possible. We will show that this upper bound can be given as:

$$C(n) = O\left(\frac{n}{4}\right). \qquad \text{(Eq. 4.2)}$$

This upper bound will occur when an XML source document contains elements that are all created using no less than 4 characters each. As previously defined, an element is an opening and closing tag pair with its contents or an empty element tag that is a concatenation of a start and end tag. There are two categories of XML documents that

we need to examine to establish the fewest characters necessary to create an element: Data-centric and Document-centric XML documents.

Data-Centric XML

Data-centric XML documents are those documents designed to store structured data in records and are similar to a relational database table; it has a set of possible pre-defined fields, which contain records (atomic units of information) that conform to a specific structure. When two different relational databases or two different Business to Business (B2B) applications wish to exchange information they can create a data-centric XML document and may even specify allowable fields using a separate Document Type Definition (DTD) or XML Schema document (XSD). All the previous examples we have seen so far have been data-centric XML documents.

An example of the smallest possible element in a data-centric document is given as:

```
<x/>                (Eq. 4.3)
```

which has 4 characters (n=4) and can be defined by 4 bytes in UTF-8 encoding. This element has a title, x, but has no content or attributes. Such an element could be represented by one node in the DOM tree; again this is 4 characters in our source XML document resulting in the creation of 1 node in our XML DOM tree. In a data-centric document it would be unlikely to find such "empty" elements. Slightly more likely XML

elements include:

<x y="z"/>	(Eq. 4.4)
<x>z</x>	(Eq. 4.5)

Eq. 4.4 is valid XML and is composed of 11 individual characters. Eq 4.4 could be represented in a DOM tree by two nodes one named x composed of 6 characters and the other, an attribute node which could be referred to as y, is composed of 5 characters. Eq. 4.5 is also valid XML and is composed of 8 characters. Eq. 4.5 could also be represented by two nodes: one which is a title node and a child text node containing the character z. Since there eight characters in the string, this results in a cost of 4 characters per node. Therefore, the number of characters is 8 (n=8) and the number of nodes is 2 (C(n)=2), consistent with Eq. 4.2. What needs to be noted in the second example Eq. 4.5 is that the single text node "z" cannot exist without being wrapped in a separate delimitating parent node name x. That is it takes 8 bytes total to create the two distinct nodes.

An exhaustive search of all possible valid combinations of character configurations for XML documents will not reveal any that can result in a valid declaration of an XML element using less than 4 characters. Even single character text elements require parent elements which will require 7 characters to represent. The minimum number of characters that can represent an element in an XML document is 4. Including additional text content in an element and/or increasing element name size will

only increase the element size.

Document-Centric XML

Document-centric is distinct from Data-centric XML and has as its focus, not data, but the document; the text, something pre-existing with its own structure. Document-centric XML can be thought of in terms of markup languages like XHTML, which create valid XML documents, but are not intended primarily as a tool for systemic data exchange. As such, these types of documents provide means to explicitly indicate which of their parts have semantic structure (for example, when a section of a document represents a head section, or a paragraph, a stanza of a poem, a title, a name, or a geographic location) as well as tags which are used to indicate how the data should be presented (bold, italic, etc.). Document-centric XML does not usually have the same repetitive and predictable structure that data-centric XML does; rather, elements representing structure, document features and formatting instructions appear as needed in a given document.

The following two examples represent the smallest possible element configurations in a Document-centric XHTML document:

c	(Eq. 4.6)

<z/>	(Eq. 4.7)

Both examples above would be valid in XHTML. The first example would be rendered as the letter c bolded (**c**) and requires 8 characters. Eq. 4.6 requires two nodes: Seven characters are used to create the node name 'b' and 1 character is used for the text node with content, c. Although the structure of the Document-centric character representation is different from the Data-centric one (Compare Eq. 4.3 and Eq. 4.6), they both require two nodes and establish a bound of 4 characters per node. Eq 4.7 can be thought of as a modified break-tag (<br />) and uses 4 bytes to create a single empty element.

Document-centric allows for mixing formatting and structural tags in ways that would seem to violate existing XML rules. In Fig 4.1A we see an XHML file available online [33] that has a paragraph element (*<p>* ... *</p>*); everything between the start and end paragraph tags is considered to a be the content of the <p> node. At first it would appear that the text child of the p node has a subchild <i>. However text nodes are not allowed to have children. In reality, as shown in 4.1B, when dealing with a DOM representation of the XML file shown in 4.1A, the paragraph node would be considered to have 3 children, two of which are text nodes, the middle node being a formatting node titled 'i'.

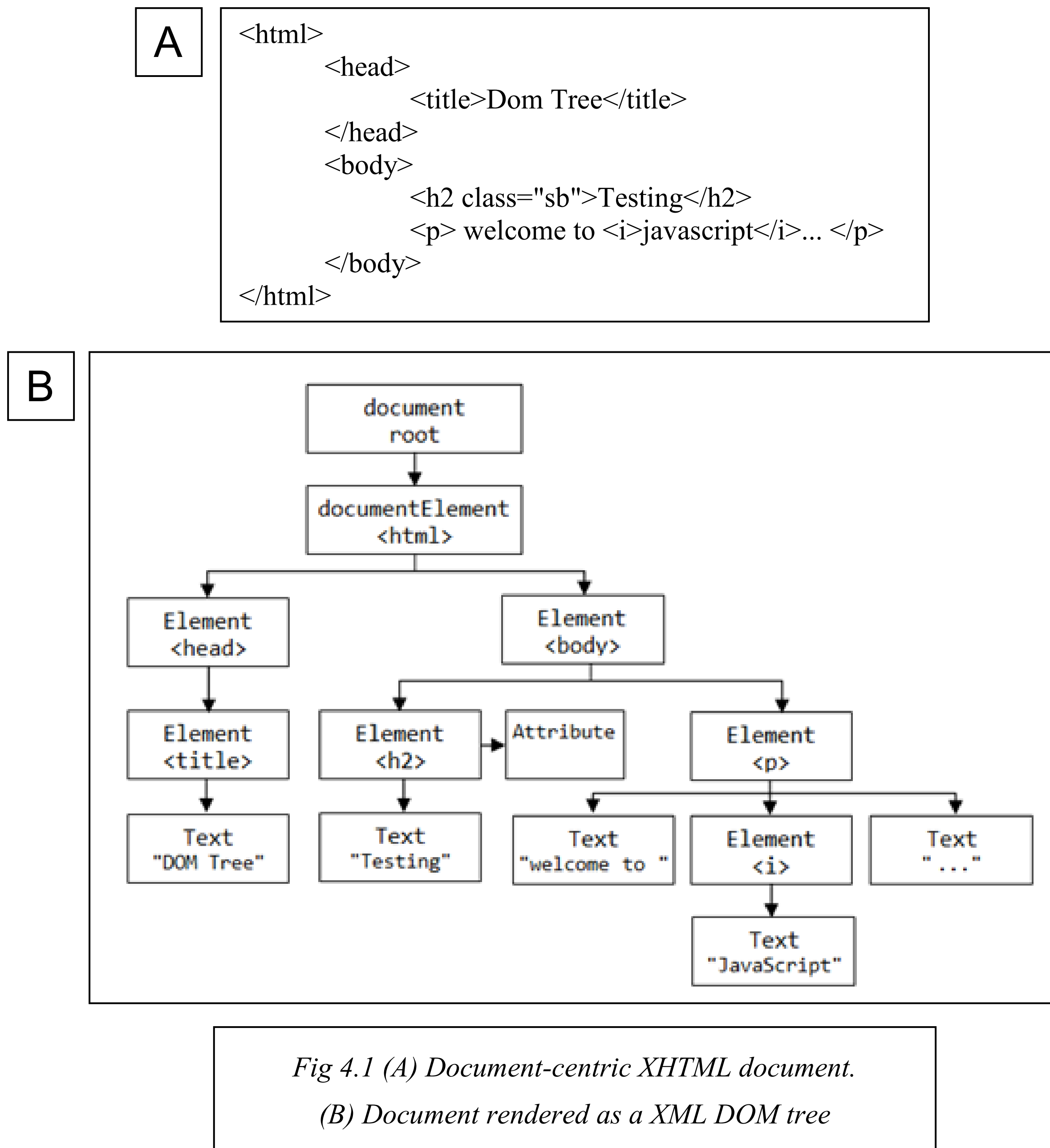


Fig 4.1 (A) Document-centric XHTML document.
(B) Document rendered as a XML DOM tree

The decision to break up the content of the <p> element into 3 children is not a trivial one, and will present complications when we discuss indexing DOM node data

with data offsets (Section 7.2) and reusing data indexes (Section 7.3).

XML Node Types

The XML DOM model specification establishes 12 different node types that can appear in an XML DOM tree. Equations 4.3-4.7 have shown the fewest number of characters that could be used in a source XML document to create the elements that would result in those nodes being created in the XML DOM tree for these types of nodes:

1. ELEMENT_NODE
2. ATTRIBUTE_NODE,
3. TEXT_NODE

This leaves the following types of nodes unrepresented:

4. CDATA_SECTION_NODE,
5. ENTITY_REFERENCE_NODE,
6. ENTITY_NODE,
7. PROCESSING_INSTRUCTION_NODE,
8. COMMENT_NODE,
9. DOCUMENT_NODE,
10. DOCUMENT_TYPE_NODE,
11. DOCUMENT_FRAGMENT_NODE,
12. NOTATION_NODE

In almost all of these cases the distinct node type in question is derived from a unique element type in our source XML document; those unique elements are all established

by using extra delimitating characters beyond what is required to establish a normal element. As an example a comment element is created by including the five extra characters:

<!-- x -->	(Eq. 4.8)

Eq. 4.8 shows an example of a single character comment element created using 10 characters. Those nodes that are not derived from elements that contain extra characters are either singular entities like Document_Node (there can be only one per document, and it is a reference to the source document, so usually quite large) or are "entity references" which are special combinations of characters used to represent Unicode characters and other "escape sequences" in XML. Two examples are:

<	(Eq. 4.9)
~	(Eq. 4.10)

Eq. 4.9 is an entity reference to the less-than (<) symbol and is declared using 4 characters. Eq. 4.10 is another entity reference and is created using 7 characters; it refers to the tilde (~) character. Both Eq 4.9 and Eq 4.10 are valid XML and if found in the source XML document could result in the creation of a node in the XML DOM tree. However all valid entity references required at least 4 characters to declare.

Reviewing this section we can see that it takes at least 4 characters in a source XML document to make a declaration (reference or element) that would result in the

creation of a node in an a representative XML DOM tree. Thus for a 1 gigabyte (1,073,741,824 byte) XML document, that is using UTF-8 encoding (1 byte per character in most cases) there should be no more than 268,435,456 individual nodes in the DOM tree, or no more than ¼ as many nodes as the total size (in bytes) of the source document. However, as mentioned previously the XML standard specifies that XML documents should be easily comprehendible by human beings. It is unlikely that any XML document therefore would consist of only minimal size elements. It should be clear at this point that the formula displayed in Eq. 4.2:

$$C(n) = O\left(\frac{n}{4}\right). \qquad \text{(Eq. 2)}$$

is almost certainly unrealistic for the average case XML document. As part of the future research proceeding logically from this thesis we would like to establish $C_a(n)$: the average number of nodes necessary to represent a given XML document of size n. We are particularly interested in a $C_a(n)$ for very large XML documents. We suspect that the average number of nodes will be significantly smaller then (n/4) for Data-Centric XML and vastly smaller for Document-Centric XML (which will have huge sections of text).

5. Node Arrangement and the Effect on DOM Cost

In the construction of the DOM tree from the XML document, it is not only important to know the maximum number of nodes (elements), but what structural information is stored within each node and the internode structure. This is of particular importance when using common Relational Algebra or Relational Calculus for querying the XML DOM trees.

5.1 Vertical (Bar) DOM Tree Arrangements

One Tree Algebra for XML (TAX) recommends the inclusion of a "pedigree" value in the generalized nodes, which constitute the DOM tree [24]. While leaving the implementation of the pedigree value up to the user, in TAX is made clear that pedigree value should serve as a reference to the complete listing of all of a node's ancestors all the way back to the root of the document. This implementation would be disastrous if applied to a large document with an XML structure shown in Fig 5.1A.

Such an XML structure would create a very tall vertical (bar) DOM tree (Fig 5.1B) with the number of nodes equal to n/7 (n = size of document). An one GB (1,073,741,824 byte) XML document of this form would have at least 153,391,689 nodes, meaning that final leaf node of the tree would have to reference in its pedigree property value all 153,391,688 of its ancestors, with its parent referencing all 153,391,687 of its ancestors, etc.

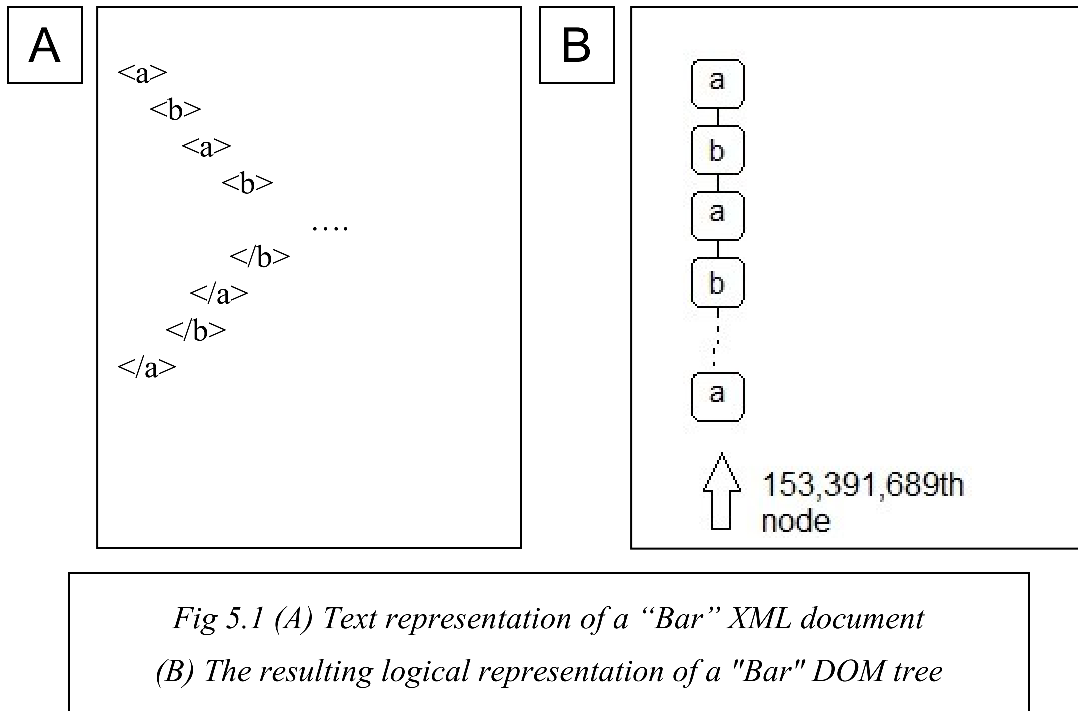


Fig 5.1 (A) Text representation of a "Bar" XML document
(B) The resulting logical representation of a "Bar" DOM tree

The cost of the inclusion of a pedigree value for each node in the beam model thus becomes the summation:

$$\sum_{i=1}^{n} i = \frac{n^2+n}{2} = \Theta(n^2) \qquad \text{(Eq. 5.1)}$$

Using an array of <u>bytes</u> (assuming that a parental reference could be contained in one byte) to record pedigree information, the total cost for storing pedigree information would be 16,434,823 GB (16 TB) for a 1 GB document.

The very large documents that were examined as part of this thesis have an average DOM tree depth of about 10 nodes. If we assume once again a number of

nodes equal to n/7 (near the maximum) and an average pedigree depth over all of the nodes of 5 then we would still need 731.4 megabytes just to store pedigree information (again assuming that a parental reference could be contained in one byte, multiply by four should we need to use standard pointers using 4 bytes).

Although we acknowledge the usefulness of retaining pedigree information within the structure of a DOM systems nodes, particularly as an enabling agent for twig join algorithms, when dealing with very large documents the overhead cost of that inclusion could quickly become impossible to negotiate. As we will see it is possible to retain some pedigree information outside of the nodes; without increasing node size.

5.2 Horizontal (Beam) DOM Tree Arrangements

Another node arrangement that can lead to a very large increase in memory requirements can be seen in Fig 5.2. The document structure shown in Fig 5.2A would create a wide, horizontal (beam) DOM tree (Fig 5.2B) with the number of nodes equal to the upper limit of n/4 (n = size of document). A one GB (1,073,741,824 byte) XML document of this form have could have as many as 268,435,455 nodes.

In many DOM modeling systems, each node structure reserves space for both parent links and child links, even though in the case of a "beam" model DOM tree, all but one of the nodes do not have children and all but one of the nodes have the same parent. Finding a way to eliminate and/or reuse parent and child links (at a typical storage cost of 4 bytes per link) would offer profound cost savings in situation where the XML source would generate a short but wide ("beam" like) DOM tree.

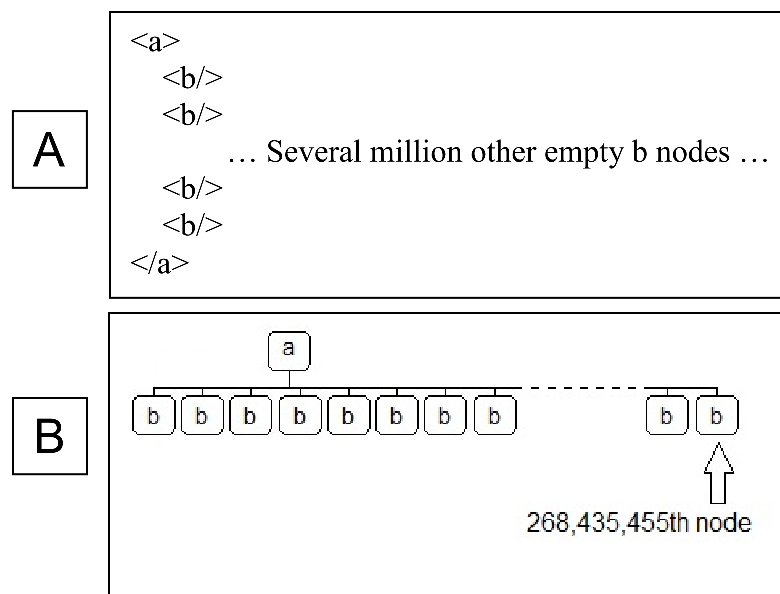


Fig 5.2 (A) Text representation of a "beam" model XML document
(B) Logical node representation of a "beam" DOM tree.

Ancestors Array

To improve efficiency and reduce the size of the data structure that was examined for creating and storing nodes (Fig. 3.3), we can eliminate all node references to parent and child nodes by creating a separate ancestors array. This ancestor array can be used to keep track of the current location in the DOM tree as we navigate it while processing a particular query. The ancestor array will store a list of all nodes that have been traversed to reach the current location and can be followed backwards to the root. That is, as a query moves through the DOM model tree the parent and any siblings of the current node are tracked separately. To put it another way the "pedigree" of the current node will always be known [24].

In a BAR model (tall) DOM tree, the total space that would need to be reserved for storing the ancestor array links for the lowermost leaf node would be roughly equivalent to having each node store its own parent node link. However, in a BEAM (wide) tree scenario the cost would be equivalent to only one pointer.

Our preliminary research indicates that on average DOM model tree depth is less than 10 nodes, even for very large DOM models (10GB+), and in general both data-centric and document-centric XML documents are vastly wider than they are tall.

5.3. Hidden Costs in Traditional DOM

Having identified the information that must be stored in the DOM tree nodes as the primary factor influencing size in DOM modeling, we must carefully consider both what to include in our nodes and HOW to include it to maintain reasonable efficiency. C++ Strings and Vectors might at first seem useful constructs to include within our nodes as they allow quick and easy manipulation of text information and links. However, both of these object-oriented constructs come with hidden overhead costs. Vectors in particular frequently over allocate during construction, that is, although we may request a Vector able to handle an array of 8 items the Vector may in fact be created with 16 array positions. This extra allocated space is normally not a major consideration and keeps the vector from having to go to the stack for more memory if we decide to add one or two more items to our Vector. However, when we are looking at creating millions of nodes, then a few extra bytes in each node add up very quickly, contributing to a large unnecessary overhead.

Strings and Vectors which are created by using classes illustrate a common problem in using object-oriented constructs in any format where we are trying to create an efficient node structure for use in DOM tree modeling. The way that the compiler and program differentiates between multiple class types when tracking function calls on an object is by keeping a hidden translation table of pointers which allows it to correctly resolve function calls made between multiple "related" object types. This hidden translation table is created whenever we invoke classes or objects that use Inheritance and/or Abstract Base Classes. Again, a single extra pointer used to differentiate between object classes is not normally a major consideration but when we are looking at creating millions of nodes based on a class-inheritance chain, then those extra pointers add up very quickly.

Creating a simple and reliable method for working with a representation of a DOM tree that does not rely on classes or advanced data structures was a key goal of this research.

6. Reducing the Number and Size of DOM Tree Nodes

Having looked at the extreme cases and examined situations in node and tree construction to avoid, we can examine several different techniques that will allow us to reduce the overall size of our DOM tree. First we can reduce the number of nodes in the tree by finding ways to prevent the construction of unnecessary nodes. Text and attribute nodes for example can be easily recovered using other references. Reducing the number of nodes in our tree will logically make the tree smaller. Secondly we can examine the problem of creating the smallest possible size node representation that can still be used to create an effective and useable DOM model tree.

6.1 Indexing DOM Node Data, with Data Offsets

When attempting to retrieve text, there is no need to copy the text from the source XML document to the DOM tree nodes. In our previous node proposal, shown again below in (Fig 6.1) we expressed the desire to be able to retrieve the following units of information: name of element, attributes of element, children of element (including text nodes) and parent of element. We then examined a structure that would allow us to store all of that relevant information using pointers. These pointers might remain NULL but could be redirected to dynamically created containers used to hold text values like "name of element".

```
struct node{
        char     * name;          // Pointer to name of the node or type of node
        char     * value;         // Pointer to value of the node. May be set to NULL
        map<char,int>  * attribute_map;       // Pointer to attribute map, may be left NULL.
        node    * parent_link;           // Link to the parent
        node    * * child_links;         // Pointer to array of links to children
        int   childcounter;         // Integer for iterating through the child_links array
 } ; // sizeof(node) = 24.
```

Fig 6.1 Generalized DOM tree node structure in C

We previously showed that in fixed size version (FSV) file format, we can rapidly retrieve units of information simply by calculating the offset within the file. Likewise if we are looking for an element's name (example "book" as in <book>) all we need for each individual element is the offset of that unit of information within the actual text document. This information can then be retrieved from that section of the source XML document. Thus, rather than using pointers to an object that may not in fact exist (text nodes do not have names), we can replace our pointers with integers that represent an offset to the location of the data in the source document. For each node we can then eliminate: 1) the duplication of data that already exists in the original source file and 2) The need for the name, value and attribute map properties of our node data structure. These parameters can be obtained by parsing from the integer offset from the beginning of the document. Caching of the source document can be implemented to achieve rapid retrieval. Simple caching algorithms such as Least Recently Used (LSU) or a variety of simple reinforcement learning techniques (Monte-Carlo) can be used to increase the

likelihood of a cache hit if the source document is divided into sections and individual sections are cached to handle queries.

The pointers we were using in our original node structures were by default 4 bytes in size (on a 32 bit system or 8 bytes on a 64 bit system). Replacing these pointers with unsigned integers (that represent a byte offset within a document) of the same size thus will incur no additional cost in space. Since most element names average 6 characters (7 bytes in UTF-8 in a null terminated string) storing an offset to the element name (rather than a duplicate copy of the name itself) will save on average seven bytes per node by eliminating the duplication of the element name. This offset to the name of an element we will now refer to as the data_index. The data_index can be used to retrieve more than just the element name, but blocks of text as well.

6.2 Reusing Data Indexes

The attributes and 1st text child element of any XML element can both be found by parsing from the start of the element tag itself and looking for the specific characters that delineate their existence (Fig. 6.2).

XML element names cannot contain spaces, so if the data index for an element points to the opening angle bracket ('<') of that element and we begin parsing from that opening angle bracket, as soon as we detect a space character following at least one character of data, we know that the name of the tag is over and anything that follows is part of the attribute set for that tag. Likewise we will know that we are done with the attribute set for that tag when a closing angle bracket is detected (' > '); at that point if

we detect an alpha-numeric string then we know that we have found the first text element child of the original element source element. In this way, we can save all of these attribute descriptors within each node.

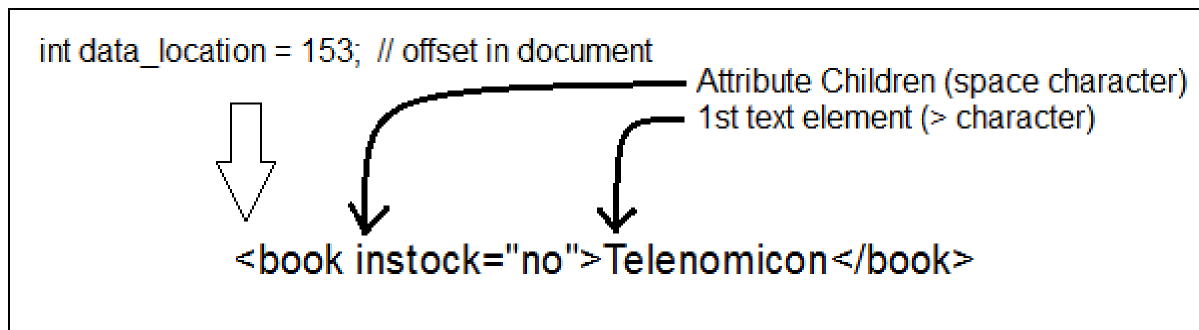


Figure 6.2 Three related datum, recoverable using one data index.

7. Upper and Lower Bounds on DOM Tree Size

We can now consider the smallest possible DOM tree that can be created, given a source XML document of a given size by not duplicating any text inside our DOM tree nodes. We see below the previously discussed simplified XML document, retrieved from the web [33], along with its associated tree [Fig 7.1A and 7.1B].

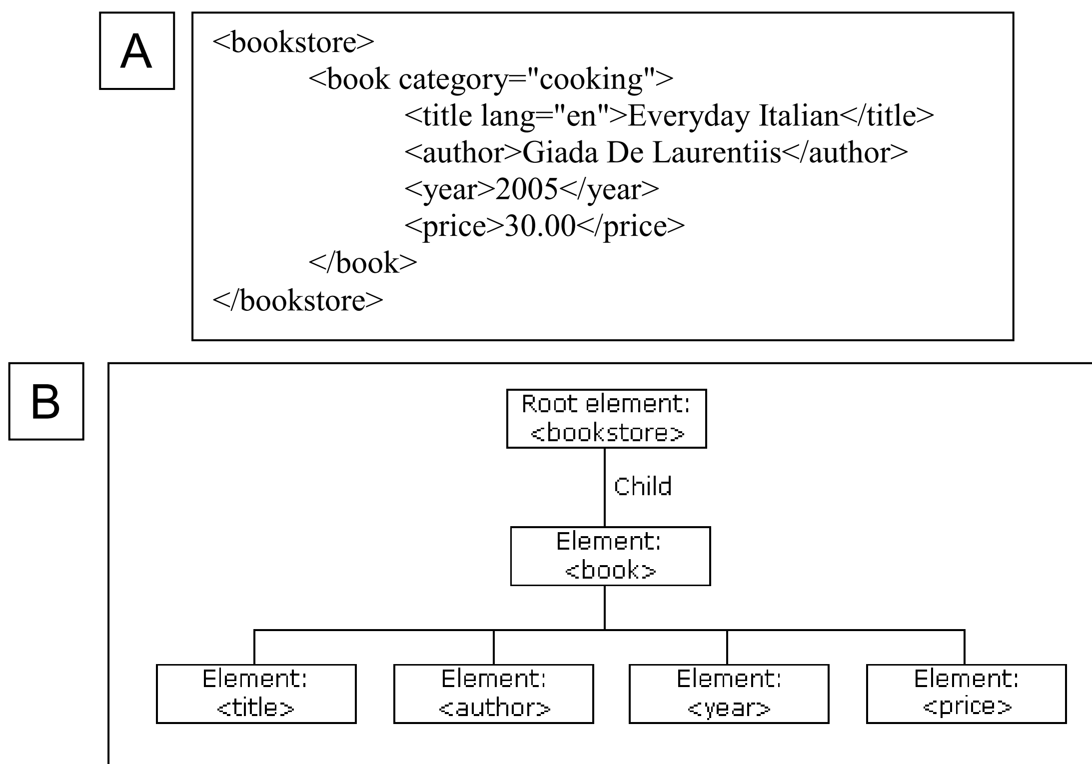


Fig 7.1 (A) Simple XML Document and (B) associated DOM tree.

In order to represent the complete XML DOM tree above we need to be able to represent (1) the nodes and (2) the links between the nodes. We can dramatically

reduce the space required for the nodes if the nodes themselves consist only of a unique data index integer value representing the offset in the source XML document where the elements content can be found. The size of the integer value we use for the data index value will affect the size of the source document that we can manipulate; at a cost of 4 bytes on a 32 bit system we will be able to access documents up to 4GB using the native frame size. At a cost of 8 bytes on a 64 bit system we will be able to access documents up to 16 Exabyte in size again using the native frame size.

For any tree there can be only N-1 links between tree nodes. The question is what it will cost to represent a bidirectional link between two nodes; represent the parent-child relationship between two nodes? Let us assume that we have a way to represent a bidirectional node using the standard frame size for whatever system we are working on (4 bytes for a 32 bit system). The smallest possible memory allotment necessary to represent the largest possible DOM tree based on an XML document of a given size (n) is related to the size of the source document as shown in equations 7.1 and 7.2.

$$\mathrm{C(n)} = \Omega\left(\left(\frac{\mathrm{n}}{4}\right) * 4 + \left(\frac{\mathrm{n}}{4} - 1\right) * 4\right) = \Omega\,(2\mathrm{n} - 4) \qquad \text{(Eq. 7.1)}$$

[in bytes, for documents up to 4 gigabytes]

$$\mathrm{C(n)} = \Omega\left(\left(\frac{\mathrm{n}}{4}\right) * 8 + \left(\frac{\mathrm{n}}{4} - 1\right) * 8\right) = \Omega\,(4\mathrm{n} - 8) \qquad \text{(Eq. 7.2)}$$

[in bytes, for documents up to 16 ExaBytes]

For a document of size n there can be at most n/4 nodes in the document tree. In the simplest possible representation of the tree, each node consists only of a single data index integer (4 or 8 bytes, used to retrieve the text information that belongs to the node) and each bidirectional link (parent-child) is another integer value (another 4 or 8 bytes).

It should be noted at this point that we previously referenced the commonly stated belief that DOM modeling of any XML document requires at least 2n to 5n space in memory to create the DOM tree. We can now state that this belief is clearly wrong. The formula:

$$C(n) = \mathbf{\Omega}(2n - 4) \qquad \text{(Eq. 7.3)}$$

represents the lower bound for the worst possible case (regarding the number of unique nodes in an XML document). That is, the formula states that even if the best possible structural configuration to represent the nodes and the links between the nodes are utilized, then it will cost you roughly 2n to create the document tree if the document has n/4 nodes. BUT, the average XML document is NOT going to contain n/4 nodes and so this formula can be rephrased in terms of an upper bound on the cost of using DOM on all XML documents (again, provided you can create that optimal structure).

$$C(n) = \mathbf{O}(2n - 4) \qquad \text{(Eq. 7.4)}$$

In this thesis we will demonstrate that the UPPER bound for the amount of

memory needed to support DOM querying of any XML document should be 2n for the general case and 3n, should complete caching of the complete XML source document be desired. For the sample set of very large XML documents used in this research the average number of nodes per documents base on document size is substantially lower than our previously established upper bound of n/4. Moreover, the DOM tree we will construct in section 10 will be easy to represent as an array, allowing efficient caching of the DOM tree. This will allow us to model XML documents, regardless of size, as DOM trees using a small fixed memory size (below 1MB) if caching of both the source document and the DOM Node array (defined in Section 9) is supported.

8. New DOM Data Structures

Several principles that will allow for the creation of a node with smaller information content for a DOM tree (size of node) have been considered. In this section, these principles will be applied to obtain DOM tree whose nodes are of the smallest possible size. The arrangement of these nodes such that the tree can still effectively support DOM querying as specified by the W3C will then be considered.

8.1 A Simplified Node Object

A relatively traditional implementation of a DOM node with the inclusion of a data_location integer, uint32_t is a simple way to implement an offset into the source XML document (Fig. 8.1). This implementation would allow for the recovery of information from the source XML document such as title, attribute, and text content of the element rather than storing that information in the node.

```
class node {
   uint32_t      data_location; // Start tag of the element in the source document
   node          * * links;       // Array of all links. links[0] can be parent
   uint32_t      linkcounter;    // Integer for iterating through the links array
 } ; // sizeof(node) = 12bytes.
```

Fig 8.1 Simplified classical node class

A further reduction in the size of the node can be achieved by eliminating the parent link and reserving position 0 in the links array for this purpose. The remainder of the links array will be used as pointers to child nodes. This reduces the memory required to store the tree from a base cost of 24 bytes per node to a base cost of 12 bytes per node. The total cost in bytes, C(n), using this node configuration on a 32-bit system and assuming we have the upper limit (n/4) of possible node is:

$$C(n) = O\left(\left(\frac{n}{4}\right) * 12 + 2\left(\left(\frac{n}{4} - 1\right) * 4\right)\right) = O\,(5n - 8) \qquad \text{(Eq. 8.1)}$$

The first part of the formula for the order of the cost of storing the DOM tree in memory,

$$\left(\frac{n}{4}\right) * 12$$

represents the cost of storing the nodes themselves. The second part of the cost in the formula,

$$2\left(\left(\frac{n}{4} - 1\right) * 4\right)$$

represents the cost of storing the links between the nodes. It is worth noting that this configuration is only reducing the cost of storing the nodes themselves. The cost for storing the pointers between the nodes remains relatively unchanged. The upper bound when using the traditional DOM configuration would be:

$$C(n) = O\left(\left(\frac{n}{4}\right) * 24 + 2\left(\left(\frac{n}{4} - 1\right) * 4\right)\right) = O\,(8n - 8) \qquad \text{(Eq. 8.2)}$$

Thus, the actual memory cost for creating the DOM tree using this method for an average XML document would still be substantially lower than the traditional DOM tree implementation (shown in Fig 6.1).

This arrangement of pointers and tree nodes has an advantage in that it allows for the creation of a traditional flexible tree structure in memory making it easy to update, add and delete nodes and can be made to support the W3C DOM specifications. A major disadvantage of this structure is its dependence on pointers. This dependency requires either direct memory access, which is not available in some high level scripting languages or the use of advanced data structures that may have their own hidden class based costs.

8.2 A minimum DOM Node (MDN) structure

With large XML documents it is very rare that inserts, deletions and updates are allowed to be performed on an individual basis, rather these operations are placed on a queue, i.e, and operation queue, and batched together (performed sequentially) when the system is not being actively mined for information [8, 26]. As result, when working with extremely large XML documents, we can concentrate on an XML DOM tree structure designed principally to support a query system, provided that it allow the inserts, deletions and updating of nodes to be performed later. By making DOM querying our systems focus we can accomplish a dramatic reduction in individual DOM tree node cost and actually meet our theoretical minimum 8 bytes per node. This structure will provide a profound storage savings (See Fig. 8.2) over our previous

structure (Fig. 8.1).

```
struct minDomNode{
      uint32_t   data_location;    // Start of the tag
      uint32_t   first_child;      // First Child & Sibling Status
  } ; // minDomNode = 8.
```

Fig 8.2 A Minimum DOM Node (MDN)

This structure (Fig 8.2.) contains only two 32-bit unsigned integer and no pointers. As with our previous structure this new minimum DOM node (MDN) structure contains both references to text data in the source XML document (*data_location*) as well as some information about the structure of the DOM tree itself (*first_child*). Unlike the previously presented node object for DOM tree representations (Fig. 8.1), of XML the MDN structure will never grow in size. All of the information necessary to represent a complete and navigable DOM tree will be contained either in the nodes themselves, or in the array that contains the nodes.

This structure will allow the utilization of DOM, and the execution of DOM queries for any XML document up to 2 GB. The complete DOM tree will be implemented as a pair of arrays, one containing the MDN structures and the other consisting only of integers. The second array of integers, which will be treated as a stack data structure, is necessary for the purposes of processing queries and will store references to nodes that have already been visited during the processing of a query (see "ancestors array" in 5.2)

A 32 bit unsigned integer can reference up to 4,294,967,296 (4GB) unique

positions in an XML document, this particular structure (Fig. 8.2) is in theory limited to documents of 4GB in size, but practical compiler limitations reduced this to 2GB when implemented. Doubling the unsigned integers contained in the integer and structure arrays to 64 bit unsigned integers would allow us to address document over 16Exabytes in size. This allows for exponentially increasing the accessible document size, while only doubling the size of the arrays stored in memory.

It should be noted that the first unsigned integer in our structure labeled data_location is as before a reference to the offset position in the source document of an individual node's text content (See section 6.1 and 6.2). This single data_location references can also be used to access leaf nodes (like attribute and text nodes, which can be quickly and easily found by parsing from the end of the parent node referenced by the data_location reference (see Fig 6.2). This will eliminate the need to create and store these nodes in our MDN tree.

The unsigned integer labeled first_child serves two purposes: One purpose is to indicate the "sibling status" of the element referred to by the data location variable. This information is contained in the first two bits of the member variable first_child. If the element is a middle child both bits are set to 0 (00). If an element is an only child then both bits are set to 1 (11). If an element is a first child or a last child of its parent then the bits are set to 10 or 01 respectively. The remaining 30 bits of the first_child member variable contain information referring to the position in an array of MDN structures where the first child element of the current element can be found. If no child element exists for the current element then these bits are all set to 0. Accessing just the first two bits, or the last 30 bits, of the first_child variable can be accomplished with a simple bit

mask or bit shift. These kinds of low level operations are well suited for the universally supported C language and could be easily included as an adjunct to any scripting language or a language such as Matlab, which is a language that is optimized for working with arrays and interacts with C/C++.

An example of a MDN structure whose member variables (data_location and first_child) have been set to (100, 32,212,225,479) can be seen in Fig. 8.3. The two values indicate that the content (name, attributes, text content) for the current node can be found at position 100 in the source document, and the current node is an only child whose first child can be found at position 7 in the MDN array.

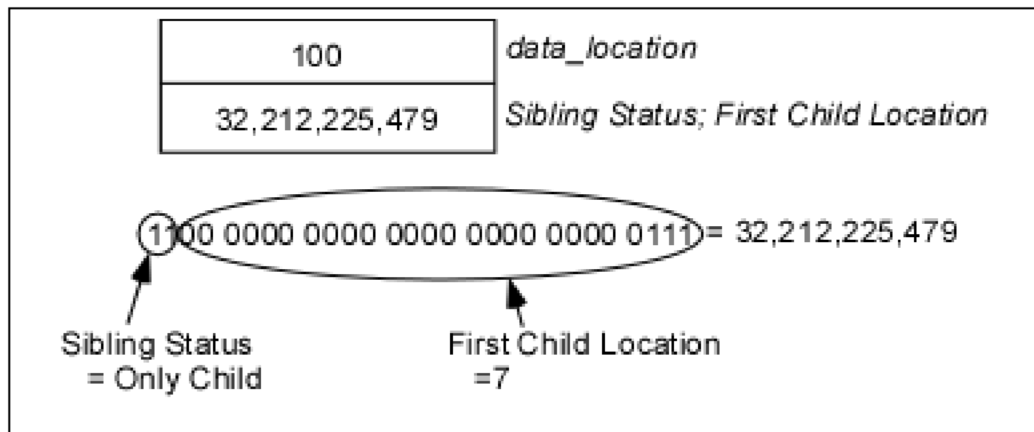


Fig 8.3 An MDN structure with values set

It should be clear that since the first_child member variable is using two of its total of 32 bits to indicate "sibling status," the remaining 30 bits can only reference ¼ as many numbers as the data_location member variable. This is acceptable, because while the data_location member variable is used for referencing a location in the source document, the first_child member variable is used as a reference to a node (or element)

created from the source document. As was established in section 4.2, there can only be ¼ as many nodes in a DOM tree as the size of the source document.

8.3 The MDN Array (MDNA)

Each of the elements in a source XML document will be stored represented by an MDN structure. The complete array of MDN structures (MDNA) will constitute a serialized representation of a traditional DOM tree. The MDN structures in the array will contain some of the information needed to help maintain the same functionality provided by a traditional DOM Tree representation. The remaining information needed for a complete DOM tree representation will come from the ordered arrangement within the array of the MDNs. The MDNA must be constructed according to a set of specific rules so that it becomes a navigable representation of a DOM tree representation of the XML document.

The rules are as follows:

A. For each element in the document that is a text or attribute leaf element in the DOM tree representation, no structure need be created in the MDNA (see section 6).

B. For those elements that are not leaf elements, the MDN structure will be arranged in the array such that the following six rules apply:

1. <u>Position 0 of the Array is reserved</u>: Position 0 will be reserved for a reference to start of the document where DTD and comment information can be found. Any node that has a first_child reference to position 0 will be assumed to have no children.

2. Position 1 of the Array is reserved: Position 1 of the array must contain a reference to the root element of the XML DOM tree.
3. Parents appear before children: In the array, if a node is a parent to any other nodes, then it appears before its children in the MDN array.
4. Siblings nodes are grouped: In the array, if the structure representing an element has siblings (elements that have the same parent), then those siblings are immediately preceding or following it in the array. No non-sibling structure can be found within a group of sibling structures.
5. Siblings nodes are ordered: In the array, a group of structures representing a group of siblings will be arranged in sorted order, such that if an element A would precede sibling element B in a sequential parse of the source document, then the structure representing element A should appear before the structure representing element B in the MDN array.
6. Child nodes are placed as close to parent nodes possible: Whenever possible a child structure should be placed at the end of near the end of the group of siblings that contains the parent structure.
7. Document order breaks ties: When conflicts arise in satisfying rule # 4, initial order in the source document wins. That is, if two or more members of a group of siblings have children, and the children are placed following the group of siblings in the MDN array, then the child element that appears first in the source document will also appear first in the MDN array.

9. DOM Methods Cost Comparison

A specific example using a document centric XML file will be used to compare the costs associated with a traditional DOM tree and DOM MDNA representations. This will highlight the concepts and techniques discussed in previous sections.

Fig 9.1 shows the source document-centric XML file used. The file has a text node that cannot be easily referenced from a node data location value: the text segment after the </k> tag. Text nodes will NOT need to be created for the <c>, <e>, and <h> elements as the text information that they contain will be easily recoverable from the source document by using the data_location variable of the <c>, <e>, and <h> node respectively.

```
<?xml version="1.0"?>
<a>
        <b>
                <c>c_text </c>
                <d>
                        <e>e_text</e>
                </d>
                 <f f_att="f_attvalue" />
        </b>
        <g g_att="g_attvlue">
                 <h>c_text</h>
                 <i i_att="i_attvalue" />
                 <j>
                        <k>k-formatted text</k>
                        not k-formatted text
                </j>
        </g>
</a>
```

Fig 9.1 A document-centric XML file with a text node that cannot be easily referenced from a node data_location value.

A text node will need to be created for the text of the <j> element as parsing from the data_location variable of <j> (looking for text) will not discover that text because the <k> node is in the way.

9.1 Cost of a Traditional DOM Tree Representation

Using a typical pointer-based node scheme (Fig. 9.2A) a traditional DOM tree can be constructed (Fig. 9.2B) in which text and attribute nodes are created as independent objects. It is assumed that none of the space-saving techniques that have been discussed are implemented. Each node contains a duplication of the information contained in the source document as well as pointers that establish parent, child and sibling relationships.

This traditional implementation requires 19 nodes at a cost of approximately 24 bytes per node for a total of 456 bytes just to create the nodes. In addition to the node cost, 36 links between the nodes are required at 32 bytes each for a total of 1152 bytes. The text information from the source document will also require 85 bytes of additional space. Consequently, this traditional DOM tree model of the XML document will require **1693 bytes** at a minimum to represent a document that is only **358 bytes in size**. Thus, the DOM tree model will be at least **4.7 times larger** than the source document.

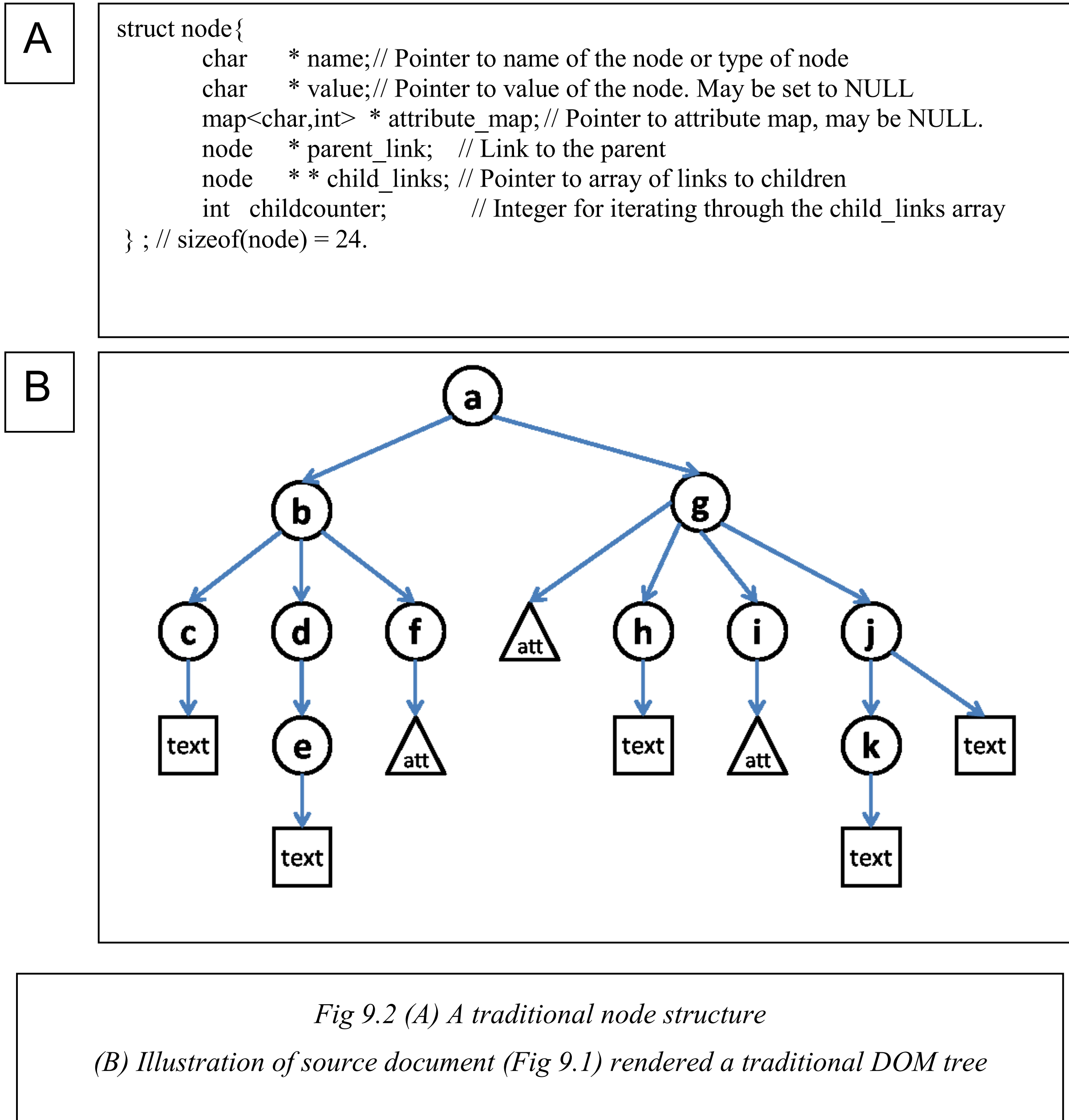


Fig 9.2 (A) A traditional node structure

(B) Illustration of source document (Fig 9.1) rendered a traditional DOM tree

9.2 Cost of a DOM Tree Using Data Indexes

Fig 9.3A shows a node structure that uses a data_location variable. This variable eliminates duplicate information in the nodes that can be readily retrieved from the

source document. Using a data_location variable also allows for the removal of unnecessary text and attributes nodes, which appear in the original DOM tree representation (Fig. 9.3B). It is no longer necessary to have a node representing the text content of the <e> node or the attribute content of the <f> node as in both of those cases the information in question can be quickly and easily recovered by parsing from the start of the parent node (<e> and <f> respectively). It should be noted that it is necessary to preserve the separate text node that is a direct child of the <j> node as that node is a sibling node to the k node and would not be easy to recover from the pointer to the start of the j node.

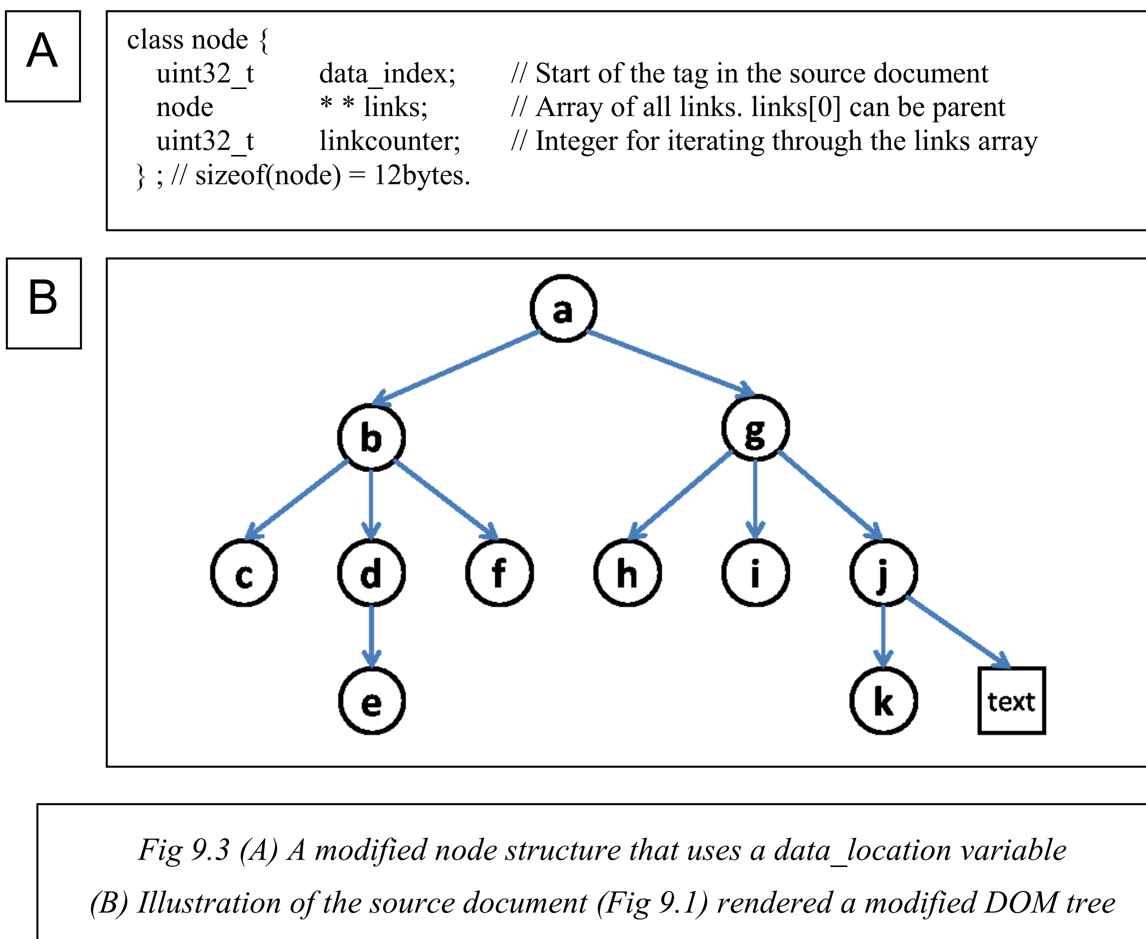
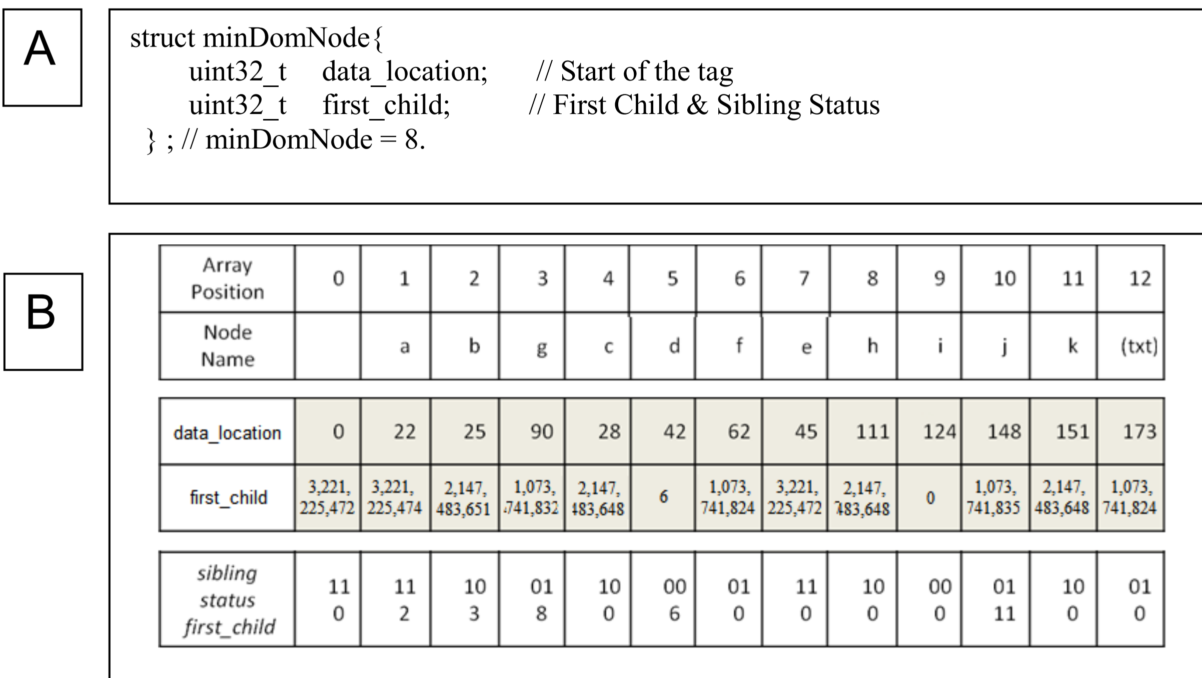


Fig 9.3 (A) A modified node structure that uses a data_location variable
(B) Illustration of the source document (Fig 9.1) rendered a modified DOM tree

This implementation requires 12 nodes at a cost of approximately 12 bytes per node for a total of 144 bytes just to create the nodes. In addition to the nodes, 22 links between the nodes will also need to be created at 32bytes each for a total of 704 bytes. In this implementation all text is left in the source document, so there is no additional cost for storing text. This modified DOM tree model will require a minimum of **848 bytes** to represent a document that is **358 bytes in size**. Even this somewhat optimized DOM tree model is at least **2.4 times larger** than the source document. If a cache is established to help improve querying time performance, the total memory requirement rises to 1206 bytes, which is 3.4 times larger than the source document. This somewhat optimized DOM representation is still significantly smaller than the traditional model (which was 4.7 times larger).

9.3 Cost of a DOM using an MDNA

Fig 9.4A shows the declaration for an MDN structure, which also uses a data_location variable. Fig 9.4B shows a MDN array (MDNA) construct created from the source document (Fig 9.1). The MDNA in figure B, is composed of the structure shown in figure A and is a serialization of the DOM Tree shown in 9.3B. The actual data stored in the array is shown shaded, while the tables above and below the array have been added for illustration and clarification purposes (Fig. 9.4B).



A

```
struct minDomNode{
      uint32_t    data_location;      // Start of the tag
      uint32_t    first_child;         // First Child & Sibling Status
  } ; // minDomNode = 8.
```

B

Array Position	0	1	2	3	4	5	6	7	8	9	10	11	12
Node Name		a	b	g	c	d	f	e	h	i	j	k	(txt)
data_location	0	22	25	90	28	42	62	45	111	124	148	151	173
first_child	3,221, 225,472	3,221, 225,474	2,147, 483,651	1,073, 741,832	2,147, 483,648	6	1,073, 741,824	3,221, 225,472	2,147, 483,648	0	1,073, 741,835	2,147, 483,648	1,073, 741,824
sibling status first_child	11 0	11 2	10 3	01 8	10 0	00 6	01 0	11 0	10 0	00 0	01 11	10 0	01 0

Fig 9.4 (A) The MDN structure that uses a data_location variable
(B) An illustration of the MDNA created from the source document (Fig 9.1).

NOTE: Actual Array contents are shown shaded.

An examination of the shaded portion shows that this implementation will require 13 MDN structures arranged in a MDN array at a cost of 8 bytes per node (see Fig 8.2, in Section 8.2) for a total of ***104 bytes***. The size of the array and the current position in the array can be tracked with two additional integer variables for a cost of ***8 bytes***. A second array (not shown) comprised of integers, tracks the “path to root," which is the sequence of visited nodes while processing a query (see "ancestors array, Section 5.2). This second array need only contain integers (referring to positions in the MDN array) and have a maximum size equal to the depth of the logical XML document tree. In this

case, the depth of the tree is 4, so we will need an array of four integers plus integers to store the size of the array and the current position within the array. This amounts to 6 integers at 4 bytes each for a total of ***24 bytes***. Consequently, this modified DOM tree model will require a minimum of **136 bytes** to represent a document that is **358 bytes in size**. This representation of the XML document is therefore **67% smaller** than the source document. This is a profound savings over the traditional DOM modeling scheme. Even if the entire source XML document is loaded into RAM to help improve query response time, the total memory requirement rises only by 494 bytes, which is just 1.45 times larger than the source document. Small caches to help improve query performance should not significantly increase the representation to source storage capacity ratio.

An upper bound for using MDN arrays is:

$$C(n) = O\left(\left(\frac{n}{4}\right) * 8 + \left(\frac{n}{4}\right) * 4 + 16\right) = O\,(3n + 16) \qquad \text{(Eq. 9.1)}$$

Eq. 9.1 assumes that the minimum number of characters (4) are used to declare all elements in the source document and that the logical arrangement of nodes to represent the document creates the maximum possible depth for the resulting DOM tree (see section 5.1 "bar model" tree) and where n is the size of our source [in bytes, for documents up to 4 gigabytes }

For this assumption, the first part of the formula for the order of the cost of

implementing DOM with a MDN array,

$$\left(\frac{n}{4}\right) * 8$$

represents the cost of storing the nodes themselves. The second part of the cost in the formula,

$$\left(\frac{n}{4}\right) * 4$$

represents the cost of storing an "ancestors array" for a tree that has the maximum possible depth. The final 16 bytes represent the cost of storing four integer variables to keep track of the size and current position within our two arrays.

9.4 Average Cost Using MDN Arrays

Eq. 9.1 represents the upper bound to use the MDN array representation to support DOM queries. It assumes that there is the maximum possible number of nodes in relation to the size of the document and assumes the worst case scenario for node arrangement, the "bar model". In the bar model tree depth is the maximum possible and thus the "ancestors array" is also at its maximum size

In contrast, empirical data suggest that even for very large XML documents, the average depth of an XML documents underlying DOM tree is seldom more than 10. Likewise the average size of the element declarations in a XML document (lower and upper bounds for size were established in Section 4) is closer to 18. That is 18 character bytes are required to indicate an element which will then become a node (or MDN) in the DOM data structure. The example document previously shown in Fig 9.1,

used very small elements (declared using tags where the name of the tag is only one character in size) which are actually relatively unrealistic. Previous research has shown that most tag names in an XML document will consist of at least 6 characters. In addition, most elements will have text content, with an average text field size of 7 [12]. Thus, approximately 24 characters will be needed to declare an element in data-centric XML document.

For traditional DOM tree data structures increasing the size of the element titles and the characters in the text sections will also increase the size of the DOM tree. This is not the case when using the MDN arrays for DOM. In fact the larger the tag names become for a source XML document, the smaller the percentage of the resulting MDN array in relation to that document will result.

Consequently, inserting more realistic values, the average cost of a data-centric document using MDNA modeling should be:

$$C(n) = \Theta\left(\left(\frac{n}{24}\right) * 8 + 56\right) \qquad \text{(Eq. 9.2)}$$

[in bytes, for documents up to 4 gigabytes]

The value $\left(\frac{n}{24}\right)$ in Eq 9.2 represents the cost of storing the nodes themselves. The remaining 56 bytes represent the cost of storing an ancestors array of size 10 (10 integers at 4 bytes) with an additional 16 bytes needed for the four integer variables to keep track of the size and current position within the two arrays.

Eq 9.2 is consistent with previous findings that data-centric XML documents on

average consist of approximately 25% data, 50% schema, 20% overhead, and 5% whitespace; the MDN array stores only schema information. For data-centric documents the averages become 60% data, 25% schema, 10% overhead and 5% whitespace, suggesting that MDN array representations of those types of XML documents will be even smaller in relation to the source document [12].

It is worth reiterating at this point that unlike a traditional DOM tree the MDN is a serialization of a DOM tree. The array is easy to break into logical sections and therefore, as with the source document itself, it is only necessary to have those sections of the MDN array in memory that are currently needed to process a query. DOM querying of XML documents, of any size, using MDN arrays can thus be supported with a low fixed memory allocation. The program detailed in Section 12 of this thesis is able so support core DOM operations on any XML document up to 2GB in size for a total cost in memory (RAM) of less than 700KB.

10. Creating the MDN Array

The MDN Array that has been implemented in this thesis offers considerable savings over traditional DOM structures, but the array must be efficiently created. On the surface, it would seem that creating the MDN Array from an existing tree would be preferable. However, traditional DOM trees lack the data offset information required for creating MDN Array. In this chapter, we consider the building of the array by either mapping from the DOM tree structure with the additional data offset information or creating the array directly from the XML document. We propose that the inclusion of data offset information would be a useful addition to traditional DOM Tree modeling of XML documents, as it would facilitate the creation of MDN arrays. We further show that if such an addition is not included in the DOM Tree structure, then building the MDN Array from the XML document is is more efficient.

10.1 Creating a MDN Array from an Existing DOM Tree

Methods for creating traditional DOM trees from existing XML documents are well established, using an event-driven (SAX, Expat) parser and processing the source document element by element. As each element is discovered (its start tag is encountered) a corresponding node is created and populated with information relating to that element. If an element is found to have children (a start tag for new element is encountered before the closing tag for the current element is encountered) then the element is placed on a stack and its children have their "parent-link" set to the node

currently on top of the stack. When a closing tag for an element is encountered if it matches the node on the top of the stack then that node is removed from the stack. Progressing in this way, a traditional DOM tree can be created. The addition of data offset information could be included or a simplified DOM tree model including only data offset information (see Fig 9.3) could be created. These types of modified DOM trees can then be transformed into a MDN array, following the rules established in Section 8.3. The disadvantage of first creating an enhanced DOM tree from which a MDN array is created is that involves creating 2 complete DOM structures, with logically twice the overhead in terms of space and time required. Moreover, if the source document is very large, it will not be feasible to build the DOM tree in a small space (See “DOM Tree size in relation to XML document size”, Chapter 4).

10.2 Creating a MDN Array Directly from the Source XML

We now consider the building of the MDN array directly from the source document. This approach is complicated by the fact that parsing the source XML document sequentially is equivalent to a Depth First Search (DFS) progression through the associated logical tree. However, the desired final MDN array arrangement will be closer, but not equivalent to, a Breadth First Search (BFS) arrangement of nodes.

The method that we have adopted to build the MDN array directly from the XML source document utilizes Expat, an event-driven XML parser [34]. Expat was originally developed by James Clark in 1998, is currently maintained by Clark Cooper and Fred Drake and is available for a wide variety of platforms. Expat is used by the Apache

HTTP Server, Mozilla browsers, and the Perl, Python and PHP languages for handling XML documents. The reason for using Expat was is that repeated benchmark tests have shown it to be one of the fastest event driven XML parsers. It is also written in **C**, and could easily be incorporated into MDNA parser application. It also has wide support for character encodings ranging from UTF-8 to Unicode [34]. The Expat parser has a skeletal structure for traversing the source document, but requires the implementation of custom event handler functions to process the different parts of a given XML document, such as start tags, end tags and character data sections, as they are encountered by Expat. In addition, it is necessary to create structures to track state and to capture the variables discovered by Expat.

The method described below employs an array of MDN structures and a second array used as a stack to track all open elements. Both arrays utilize size and counter variables that are used with these arrays. A flag variable, justOpened, is used to track whether the last tag encountered was an opening or closing tag. Another variable, outOfPlaceNodes, is used to count the number of "out of place" nodes that are discovered. The count is important for potentially determining the type of sorting routine that would be optimal for a particular XML document. Such a refinement is not critical for the development of the idea in this thesis and could be pursued during future work.

The construction of the MDN array shown in Fig 9.4 (Section 9.3) can be accomplished using the variable shown in Fig 10.1 and in 5 steps. The cost of each step is quantifiable, upper and lower bounds for some steps can be determined and are given.

```
struct minDomNode{
        uint32_t     data_location;       // Start of the tag
        uint32_t     first_child;         // First Child & Sibling Status
  } ; /* Abreviated as mDN below*/

struct minDomNode  mDNArray[n/4]; // n is size of source document
mDNArraySize = n/4;
mDNArrayPos = 0;

int stack[n/4];
stackSize=n/4;
stackPos=0;

int justOpened = 0;

int stackAddSize = 0;              /* Discussed in Section 10.3 */
int outOfPlaceNodes=0;             /* Discussed in Section 10.3 */
```

Fig 10.1 Structures and variables needed to create the MDN array in C.

1. Parse the document and create an array of location and parent references.

Each MDN structure is composed of an array of MDNs, each of which contains two unsigned 32 bit integers. One integer is the data_location and the 2nd integer value initially serves as an offset in the document relative to the PARENT element of the current node. After sorting and other refinements, the second integer will contain a previously defined **first_child** value that contains sibling status information as well as a reference to its first child node position in the MDNA.

It was necessary to implement a minimum 3 event-listeners (START(), END(), and CHAR()) for the parser. These functions capture the information needed to build the MDN array and have the following functionality:

1) START()

- This function is called whenever a start (or opening) tag is encountered.
- When called it will:
    1. Populate the first available uninitialized MDN structure(indicated by the mDNArrayPos variable) in the MDN array with:
        - A number reference, which is a byte offset within the source XML document that indicates the location where the parser discovered the start tag (data_location)
        - The data_location value of the parent node, which has been placed on top of the stack.
    2. Increment mDNArrayPos to access the next node in the array.
    3. Push the data_location value of the current node onto the stack. This value will be used to determine the parent value for any following child nodes.
    4. Update tracking variables, stackPos, stackAddSize, and outOfPlaceNodes
    5. In the case where a text element contains within itself another element, we may need to capture it as a separate node. Therefore, after detecting a START tag, set the just_opened flag variable to true. This indicates that any following text nodes may be ignored.

2) END()

- This function is called whenever an end (or closing) tag is encountered

and will perform the following functions:

1. Pop the stack and decrease the values of stackPos.
2. Set the just_opened variable to false (See 5 of START()).

3) CHAR()

- This function is called whenever a section of character data is encountered and will perform the following functions:
    1. If the just_opened variable is true, then the text encountered will be discoverable using the data_location variable of the last MDN created and no further action needs to be taken.
    2. If the just_opened variable is false, then this text will need to have its own data_location variable and its own MDN. The next available minDomElement structure will be populated with the appropriate values following the same procedure as for the start() function.
- NOTE: Text elements cannot have children, so even if a call to CHAR() creates a MDN, the data_location of that node will not be pushed onto the stack.

Text, attribute, and element nodes are 3 of the 12 possible node types that can be found in an XML document. At this point the MDNA Parser application only tracks these two types of nodes (attributes are recoverable, but not treated as nodes), and this is sufficient to prove the primary functionality of an MDN array. Adding new node types to the array would require only adding and registering new event handler functions for

those respective types.

An example XML document (Fig. 10.2), processed with Expat and the event handler functions described above would initialize the MDN array with its first set of data values. This MDN array would appear as shown in Fig 10.3 when the document encounters the close j (</j>) tag at position 190.

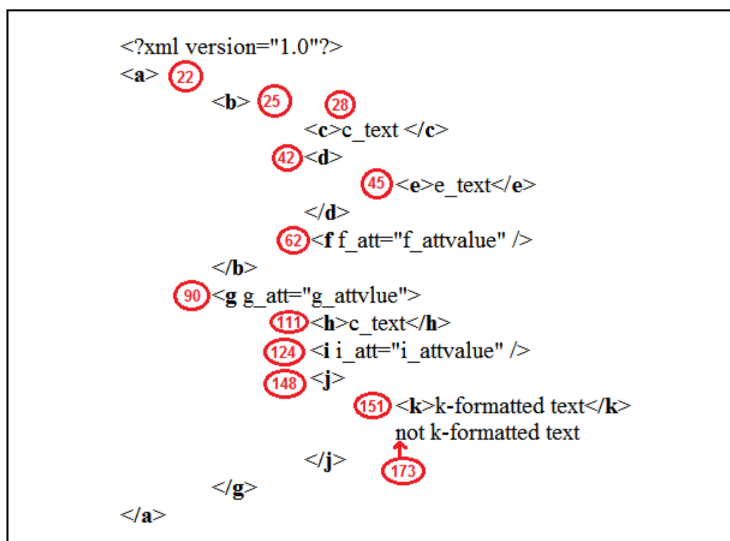


Fig 10.2 Source document showing data_location value

data_location	0	22	25	28	42	45	62	90	111	124	148	151	173
parent_location	0	0	22	25	25	42	25	22	90	90	90	148	148

stack	22	90	148										

mDNArrayPos -> 13
stackPos -> 3
justOpened -> 0

stackAddSize -> 3
outOfPlaceNodes -> 2

Fig 10.3 System state at position 190 in the document

2. Stably sorting the array created in Step 1 by parent values.

The MDNs in the MDN array are not in their proper order nor do they contain proper values. The ordering can be corrected with a stable-sort on the array using the 2nd integer (called first_child, but shown as parent_location in figures 10.3-10.7). The type of sort chosen is important and can vastly effect the time required to create the MDN array, especially when the source XML document is extremely large. As can be seen in Fig 10.3, the document is already highly ordered and many standard sorting algorithms generally do not take advantage of the inherent order. It is also necessary to preserve the order of sibling nodes (nodes that have the same parent value) in association with the order in the source document. Therefore, it is necessary to use a stable sort.

In their 1980 paper "Best sorting algorithm for nearly sorted lists" Cook and Kim defined metrics for comparing the degree to which arrays are already in sorted order. Two qualitative metrics were coined to refer to ordered arrays: The distance that elements must be moved and the number of elements that need to be moved in the sorting process. A highly sorted array is one where both of these metrics are small. A nearly sorted array is where these metrics are extremely small. The most efficient sorting algorithms for small highly sorted lists and nearly sorted lists is the straight Insertion Sort. Quickersort is best for very large nearly-sorted lists [35]. Cook and Kim (1980) went on to further show that that a combination of the Straight Insertion Sort and Quickersort with merging yields a sorting method that performs as well as or better than either Straight Insertion Sort or Quickersort on nearly sorted lists. This approach of using one or more sorts in combination or using one type of sort as a precursor to

another sort has become a common feature in new packaged sorting algorithms, such as Timsort. All these types of sorts using combinations of other sorting algorithms seek to take advantage of the natural properties of a given data set to improve sort time [36].

The MDN array can be very large for large documents, making Insertion Sort untenable. QuickerSort is also not suitable because it is generally not considered stable unless a second array position comparison is implemented, which would double the cost of the sorting algorithm. Therefore, a specific sorting algorithm needed to be implemented for the MDNA, which overcomes many of the difficulties with standard sorts. This sorting algorithm has proven to be highly efficient for this structure, but its generable applicability may not be as efficient as some of the other sorting algorithms.

The MDNA array after it is first created is very highly sorted on the parent_location value. Moreover, the elements that are out of place in relation to the entire array are still highly sorted with relation to each other (sibling relations, and parent-child relations are all correct). We can take advantage of this natural ordering with a sort that traverses the array, moving all out of order values to the end of the array. When a value is moved a hole, in the case of an MDNA node with both values set to 0, is left behind. This process is called recursively on any nodes that are moved until no nodes need to be moved. This process leaves an array with a sparse number of holes, so we refer to this type of sort as a “Swiss Cheese” sort. After this recursive sorting process, the resulting sub lists are merged to give a final sorted array (See figure 10.4).

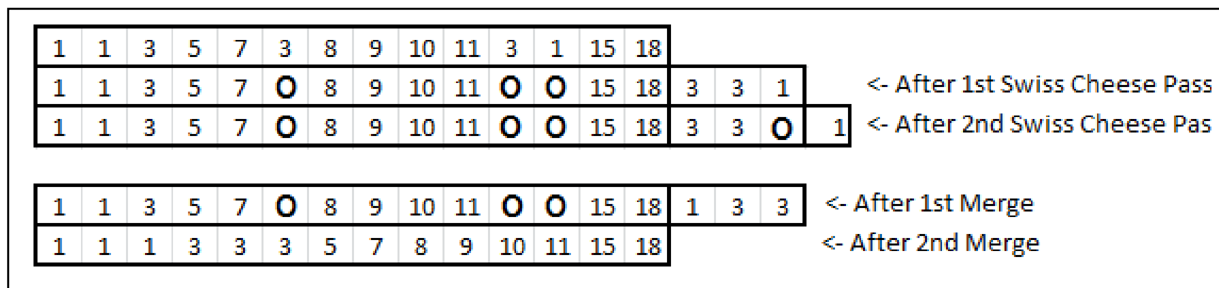


Fig 10.4 'SwissCheese' sort example. An O indicates an empty position

The general applicability of this type of sorting algorithm was also examined. It is disastrous in the case where the list is in reverse sorted order. For this list, it would require n^2 (as opposed to n log n for traditional merge sort) comparisons and n^2 additional space. However for the very highly ordered lists that we are working with (created from a DFS parse of a logical XML DOM tree) this sort seldom requires more than 8 recursive calls even for very large XML documents, making it far more efficient than a traditional Merge Sort. Swiss Cheese sort can be thought of as a variation of the Natural Merge Sort where the primary run is left in place. We can further improve the efficiency of Swiss Cheese sort by combining it with a limited depth Insertion Sort in order to eliminate small "breaking nodes" from the data. In a sequential grouping of siblings (Fig. 10.5, elements marked by curved blue bar), it is possible that only one or two of the sibling will have children.

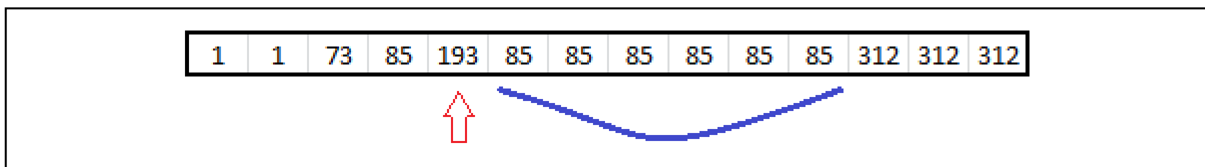


Fig 10.5 Illustration of a breaking node (red arrow) and the elements that will be moved to the end of the document (blue bar) because of the breaking node.

An example is shown in Fig. 10.5 where the fourth element of the array (the first 85) has a child element (193) that breaks up that group of siblings. During the first pass of the Swiss Cheese sort, all of the nodes with the value 85 following the node with the value 193 will be moved to the end of the array and will eventually need to be moved back during a merge. It would be faster and simpler to simply move the 193 value the short distance necessary to put it in its place at the end of the run of 85's. This can be done using a limited depth insertion sort as a pre-processing step.

In a normal Insertion Sort the element that is being added to the end of the sorted array will be repeatedly compared and if necessary swapped with the element in front of it until it is moved forward in the array to its place in the array. We can limit the standard insertion sort so that an element can only be swapped a set number of times. This will mean that if an element would need to be moved a large distance forward in the array in order to reach its spot, it may not in fact be moved far enough. However, by limiting the width (w) of the inner loop of the insertion sort to as little as 1, for a total cost over the entire array of only w * n, we create an insertion sort "window" that will identify large nodes and move them backwards in the array to their proper position. In the example shown, the insertion sort window will pick up the 193 node and move it to the end of the row of 85's as it passes over that section of the MDN array.

It should be noted that the addition of a limited depth Insertion Sort before performing the Swiss Cheese sort vastly improves the performance of the sort in the worst case scenario (where the input array is in reverse sorted order). Using the Swiss Cheese sort procedure only will require o(n^2) additional space as indicated by the summation.

$$n - 1 + n - 2 + \ ... + \ 2 + 1 \ = \sum_{i=1}^{n} n - i = \frac{n\,(n-1)}{2} \qquad \text{(Eq. 10.1)}$$

The inclusion of a precursor limited depth Insertion Sort will reduce the size required (and the resulting merge comparisons needed) by a factor of w where w is the size of the limited depth Insertion Sort window.

$$n - w + n - 2w + \ ... \ + \ 2w + 1w \ = \sum_{i=1}^{\frac{n}{w}-1} n - iw = \frac{n\,(n-w)}{2w} \qquad \text{(Eq. 10.2)}$$

As w approaches n we are of course left with basic Insertion Sort which is $\Theta(n^2)$ in the worst case, but when the insertion depth window (w) is small the cost is *wn*, a linear function.

Thus, the final sort operation has two main parts. The first part uses a limited depth insertion sort to move breaking nodes to their appropriate position, and then move any remaining out of place nodes to the end of the array. This limited depth insertion sort is a precursor to the Swiss Cheese sort. If the insertion sort detects that any nodes are out of place, then the sort calls itself recursively. The second part of the sort merges any lists that were added to the end of the array back into the part of the array that they were removed from.

Normal merge sort has a cost of $\Theta(\, n \log n)$ where the log n value is derived from the number of times the merge sort procedure recursively calls itself; the number of

recursive calls is the number of times it is possible to divide an array of size n in half (log n). With the Swiss Cheese sort the number of recursive calls is dependent on the number of nodes that need to be moved after each SwissCheese comparison pass. With the inclusion of a precursor limited depth insertion sort we can set the upper limit for the complete MDNA sort in the worst case as follows (where w is the depth of the limited depth insertion sort window):

$$w * \left(n + \frac{n(n-w)}{2w}\right) + \left(n + \frac{n(n-w)}{2w}\right) + n = \theta(n^2) \qquad \text{(Eq. 10.3)}$$

The 1st part of the equations [$w * \left(n + \frac{n(n-w)}{2w}\right)$] is the cost for the comparisons used in the limited depth insertion sort procedure. The 2nd part of the equation [$\left(n + \frac{n(n-w)}{2w}\right)$] is the cost for the comparisons used by the Swiss Cheese passes. The 3rd part of the equation [$\left(w * \frac{n}{w}\right) = n$] is the cost for the comparisons used by the merges that that bring the separate sorted lists back together. As w approaches n the short-circuit procedure of the sort means that the sort simply becomes regular Insertion Sort at a cost of $\theta(n^2)$.

Equation 10.3 represents the worst case scenario when the list to be sorted is in reverse sorted order. Using actual XML data and an Insertion Sort window of 10, even very large XML documents (2GB), the recursive Swiss Cheese portion of the MDNA sort rarely needs to call recursively call itself more than 4 times. This makes the average case comparison cost efficiency significantly better than the n log n provided by normal merge or quicksort.

The system state after the sort can be seen below in Fig 10.6.

data_location	0	22	25	90	28	42	62	45	111	124	148	151	173
parent_location	0	0	22	22	25	25	25	42	90	90	90	148	148
stack													

Fig 10.6 MDNA state after the sort

3. Establish sibling status for each MDN structure

Each node of the MDNA structure has two integer values (Sec. 8.2). The second integer value contains two unique pieces of information, one of which is sibling status for that node. In this step of the construction of the MDNA, sibling status is determined by a single pass through the array examining and comparing the parent location information of neighboring elements. Each sibling status value is stored temporarily in the integer array called stack (Fig. 10.7). The possible values are 00 (middle child), 01 (last child), 10 (first child), 11 (only child)

data_location	0	22	25	90	28	42	62	45	111	124	148	151	173
parent_location	0	0	22	22	25	25	25	42	90	90	90	148	148
stack	11	11	10	01	10	00	01	11	10	00	01	10	01

Fig 10.7 MDNA state after sibling status determined

4. **Convert the second integer in each MDN structure from a parent reference to a first child reference.**

After storing the sibling status values for each node on the stack, it is safe to overwrite the integer holding the parent location reference. It is replaced with a reference to the location in the MDNA of the first child of the associated node; 0 if the node has no children at all. Two pointers will be required: one variable to track the node currently being examined, the other to keep track of other nodes that might be the first child of the node.

For each node it must be determined if the node has another node in the array that lists it as its parent. If a node is discovered that does list the current node as a parent, it must be determined if this is the 1st node that lists this node as a parent. Since the array is ordered by parent values searching for the 1st child node of a node can be implemented as a binary search. An additional step is required to determine whether a node that is discovered is in fact the 1st child. The search speed can be improved by limiting the search window to the part of the array that has not been processed. That is, a node cannot have a child that is ahead of it in the array.

Fig 10.8 shows the system state after the node in position 10 has been processed and its first_child variable has been assigned the value 11, which is the location in the MDNA of its first child.

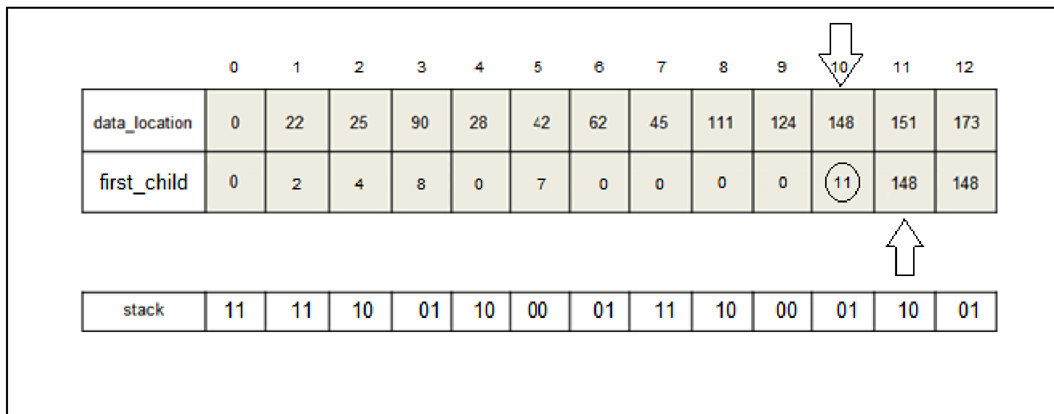


Fig 10.8 MDNA state while parent_location values are converted to first_child values.

5. Insert sibling status values into the first_child structure variable

In the final step, the corresponding value in the stack array is inserted into each node of the MDNA, using a bitmask operation on the first two bit positions of the first_child variable. This will create the data structure seen in Fig21B (Sec. 10.3); a serialized version of the DOM tree where the MDNA contains only references to element location in the source XML file, and information about node relationships (parent, child, sibling).

10.3 Efficiency Issues when Creating the MDNA

Several simple procedures can be taken to dramatically decrease the amount of memory and the amount of time needed to create a MDNA for a given source XML file. Intelligent caching can be used to increase the access and recovery speed when working with an MDNA array and its source XML document. As the size of the source

XML document approaches 2GB it becomes impractical to keep the entire source document in memory. Likewise, as the MDNA array file (and indeed the stack needed by the application becomes very large) it is only necessary to store part of those data structures in memory. Rather than only caching one part of the MDNA file in memory (the default file caching option used by most languages) it will be more efficient to create and maintain an array of memory buffers that will hold different parts of the MNDA file. As an example, when the MDNA is sorted different distinct and widely separated parts of the file will be desired. If there is only one file cache then when a different part of the file is needed, the existing cache will be overwritten, even if the part of the file it currently holds will be needed immediately. By using an array of buffers for the MDNA file (and also for the source XML document and the stack) more than one section of the MDNA file (or the XML file, or the stack) can be held in memory. This increases the likelihood that the part of the file we are looking for will be found in memory. If the part of the file we are looking for is not in a cache buffer we can intelligently decide which buffer to overwrite based on the likelihood that the contents of a given buffer will be desired in the future. Currently the MDNA application decides what cache buffer should be overwritten based on a least recently used algorithm (LRU) [37]. Even more intelligent caching based on document structure and user query habits is possible and is discussed in section 14. A study of how best to organize caching for MDNA and XML document access is a suitable topic for future research.

Very efficient sort and comparisons algorithms for steps 2 and 4 can be intelligently selected to dramatically reduce the run-time cost of creating the MDNA. In section 10.2, in step 1, it was noted that two additional integer values will be maintained

while parsing the XML document for the first time. Those values, stackAddSize and outOfPlaceNodes, will allow the determination of how many nodes are out of order and could be used to intelligently select a specific sorting algorithm based on the size and complexity of the unsorted array. The variable stackAddSize is used within the START() method to keep track of the size of the stack array immediately after a node reference has been added to the stack array. The variable outOfPlaceNodes can be incremented inside the START() function whenever the size (length) of the stack array is found to be greater than the value of stackAddSize (which refers to the height of the stack the LAST time the stack was incremented). Fig 10.9 shows the size and content of the stack array as the source document (Fig 10.2) is processed. The 2 places where the variable stackAddSize is greater than stackSize are indicated with red arrows and indicate the precise nodes that are "breaking nodes"; out of place and will need to be moved in the sort performed in Step.

				45						151	173
		28	42	42	62		111	124	148	148	148
	25	25	25	25	25	90	90	90	90	90	90
22	22	22	22	22	22	22	22	22	22	22	22

Fig 10.9 Stack array size and content after each call to START() function. Red arrows indicate out of place nodes.

In step 2, the MDNA is sorted on the parent values and sorted so that when two nodes have the same parent their order with respect to each other remains unchanged. This "stable sort" requirement makes *n log n* sorting techniques such as Quicksort unusable unless a second comparison (array position, document position) is used in order to maintain sibling order; thus complicating the run time of the sort. By tracking the number and position of the out of place elements, it would be possible to determine when using a highly efficient version of BubbleSort (or some other simple stable sort) would be sufficient to sort the array. In the example above, BubbleSort could be called twice starting both calls from the position of the last out of place node, with the orientation of the sort swaps headed toward the beginning of the array. After only two passes, each of which starts halfway through the array (2n / 2 = n) the array would be properly sorted at a fraction of the 2(n log n) cost on average that a stable quick sort would take.

It should be noted, that a firm upper bound for the number of out of place nodes can be established as (where N is number of nodes in the document):

$$\mathrm{OutOfPlace}(N) = O\left(\left(\frac{N}{2}\right) - 2\right) \qquad \text{(Eq. 10.4)}$$

It was shown earlier that it is possible to detect "breaking nodes". A breaking node can be considered to be "out of order" or we can consider the nodes that follow it to be "out of order" but in either case if we move either the breaking node or the node that follows it we correct the problem. Thus if the majority (more then ½) of the nodes are breaking

nodes it will be better to leave the breaking node where they are and move the other nodes and vice versa. In Equation 10.4 the -2 represents the fact that the root of the document and the first child of any set of sibling cannot be out of order.

Finally, intelligent caching can also be used to increase the access and recovery speed when working with an MDNA array and the source XML document. As the size of the source XML document approaches 2GB it will become impractical to try and keep the entire source document in memory. Likewise as the MDNA array file (and indeed the stack needed by the application becomes very large) it will become necessary to also only store part of those data structures in memory. Currently the file buffers used to cached parts of the source XML, MDNA use a Least Recently Used algorithm to determine which buffers will be overwritten to satisfy new requests. It would be more efficient to use an algorithm to determine whether or not buffer contains elements that are currently part of the DOM branch that is being explored as part of a query.

11. Supporting Dom Queries with an MDNA

The Document Object Model (DOM) is an Application Program Interface (API) defined by a set of World Wide Web Consortium (W3C) recommendation articles. Together these Recommendations describe an object model that is used to store hierarchical documents (like XML and HTML) in memory. The most recently complete standard is the DOM Level 3 Core Recommendation released on April 7th 2004 [23]. The DOM interface was designed so that is independent of any operating system or programming language and is referred to as operating system and programming language neutral. The operational descriptions of the DOM interfaces use an Interface Description Language (IDL), defined and maintained by the Object Management Group. The IDL language has its own terminology constraints. As an example in the IDL language an "attribute" refers to a member variable of an object or structure, and should not be confused with "attribute" as it is used in XML. Below is listed the interface description for the NodeList used in DOM:

```
interface NodeList {
        Node                                    item(in unsigned long index);
        readonly attribute unsigned long        length;
}
```

This interface expressed in Java would look like this:

```
public interface NodeList {
        public Node    item(int index);
        public int       getLength();
}
```

In this section we outline the fundamental interface DOM operations and divide them into general categories as follows: (1) tree navigation operations that allow access to any node within the DOM logical tree, (2) text retrieval operations that acquire information from within the elements of the DOM and (3) tree modification operations that alter the structural arrangement or content of the nodes. This would include operations such as inserting or deleting nodes into the tree or changing the text content such as title, attributes and value associated with a node.

11.1 W3C IDL DOM interface specifications

The public interface of a DOM has the following core interfaces and API calls:

- DOMException,
- ExceptionCode,
- DOMImplementation,
- DocumentFragment,
- Document,
- Node,
- NodeList,
- NamedNodeMap,

- CharacterData,
- Attr,
- Element,
- Text,
- Comment

There are also the following extended interfaces:

- CDATASection,
- DocumentType,
- Notation,
- Entity,
- EntityReference,
- ProcessingInstruction

For a complete listing of the interfaces see appendix A.

A DOM-compliant document parser is one that supports all the DOM API calls (with the results documented) in a specific Level of the W3C DOM Recommendation. It is not required to implement the optional modules or extended interfaces in order to be DOM-compliant. If a DOM parser does not completely implement a feature it should respond "false" to the relevant hasFeature queries. Because this thesis is an exploration of the efficiency of XML DOM structures the focus has not been on full DOM compliance at any level. It is worth noting that a variety of XML parsers in widespread use do not "fully" support the DOM specification at any level (Libxml2 for C++ is one example) and almost no parsers or browsers fully support DOM level 3 at this time.

In this thesis, we have concentrated on proving that a MDN array can be used to support the core operations of the Node and Document interfaces. Almost every interface in the DOM (Core) Level One specification is an extension of the Node interface. A DOM tree is itself as previously described a collection of nodes; the W3C allows for the specification of 12 different sub-types of nodes objects. It is important to note that the DOM specification DOES NOT require the use of object-oriented inheritance in the creation of these node-types, and makes allowances for both "object-oriented" and "flattened" approaches to implementing the DOM specification.

A logical object-oriented UML view of the relationships between the DOM (Core) Level 1 interfaces can be found in appendix B, and it is worth noting that the majority of operations we would need to support are contained within the Node interface, with the bulk of the remainder being found in the Document and Element interfaces. A complete listing of the Node interface can be found in Appendix C.

The remainder of this section addresses the core methods detailed by the Node interface and how they can be implemented using a MDNA.

11.2 W3C DOM Node (Tree) navigation operations

The IDL specifications for node operations and are concerned with changing the “currently selected node” within the DOM tree. Core functions are listed below as a bulleted list; beneath each bulleted function declaration is how the method is actually declared in the IDL interface specification. The bulleted, underlined function names are there simply to add clarity:

- parentNode ()

  readonly attribute Node parentNode;

- childNodes ()

  readonly attribute NodeList childNodes;

- firstChild ()

  readonly attribute Node firstChild;

- lastChild ()

  readonly attribute Node lastChild;

- previousSibling ()

  readonly attribute Node previousSibling;

- nextSibling ()

  readonly attribute Node nextSibling;

- hasChildNodes ()

  Bool hasChildNodes();

Collectively, these operations allow any node in the DOM tree to be accessed from another node within the DOM. These operations are supportable using only the information contained in the MDNA. For example, the firstChild() operation is accomplished by extracting the location of a node's first child node position in the MDNA from the first_child variable of that node, moving to that location within the MDNA and pushing the location of the previous node (the parent) onto the stack. The lastChild() operation is then the firstChild() operation followed by a shift to the right within the MDNA until the sibling status value (extracted from the first_child value)

indicates that the last sibling has been found. The previousSibling() and nextSibling() operations can also be accomplished by shifting left or right within the MNDA if the sibling status value for the selected node indicates that this is possible. The parentNode() operation requires only the recovery of the parent MDNA position from the stack. The childNodes() operation can be accomplished by returning a sub-array of MDNA positions. The hasChildNodes() operation is a simple Boolean operation, which will return a TRUE if the node has a first child variable not equal to zero. Thus, MDNA clearly supports all tree navigation operations, which are part of the DOM 1 specification.

11.3 W3C DOM Node text retrieval operations

The following node operations are concerned with retrieving information that resides within the DOM nodes. It should be noted that because of the compact nature of the MDNA representation of the XML document, both the XML source document and its MDNA representation will be needed to satisfy retival of text requests.

Crucial to some of the operations, such as nodeValue(), is the “type of node” that is being examined. As an example the nodeValue() operation should return NULL for operations on nodes that have the following type: DOCUMENT, DOCUMENT_TYPE, DOCUMENT_FRAGMENT, ELEMENT, ENTITY, ENTITY_REFERENCE and NOTATION. The DLI specification is therefore as follows:

- <u>nodeType()</u>

readonly attribute unsigned short nodeType;

- nodeName()

  readonly attribute DOMString nodeName;

- nodeValue()

  attribute DOMString nodeValue;

- attributes ()

  readonly attribute NamedNodeMap attributes;

- ownerDocument ()

  readonly attribute Document ownerDocument;

For the each of these operations, except ownerDocument(), the first step is to extract from the current MDNA node the data_location variable. Using the data_location variable, the system needs only to find the appropriate line in the source XML document and parse forward from there until the requisite unit of information desired has been recovered. As an example, if the nodeName for an ELEMENT node is desired, it is only necessary to begin parsing in the source XML document from the data_location position which should be a "<" until a ">", "/" or "<space> " character is encountered and then return the resulting string.

The ownerDocument() function returns the root element (document object) for a node.

11.4 W3C DOM Node modification operations

It had previously been suggested that for very large XML documents, insertions, updates, and deletes would not normally be performed on an individual basis, but rather be applied as a group in a batch operation. Nevertheless, the following node operations do allow for normal inserts, updates and deletions on an individual basis. These node operations modify the information that resides within the DOM nodes. Some of the operations that have already been listed as text retrieval operations also allow the text content of a DOM node to be changed. Although they are not relisted here, they are discussed as "updating" operations below.

- insertBefore ()

  Node insertBefore(in Node newChild, in Node refChild) raises(DOMException);

- replaceChild ()

  Node replaceChild(in Node newChild, in Node oldChild) raises(DOMException);

- removeChild ()

  Node removeChild(in Node oldChild) raises(DOMException);

- appendChild ()

  Node appendChild(in Node newChild) raises(DOMException);

- cloneNode ()

  Node cloneNode(in boolean deep);

11.4.1 Updating Operations

There are two possible solutions for implementing update operations using MDNA, whether replacing an entire node using replaceChild() or simply replacing the

content using nodeValue(“new text”). If the replacement content is the same size or smaller than the original content, we can simply overwrite the existing content with the replacement content. However, care must be taken to pad any extra memory with blanks when the contents are smaller than the original contents. For example, in Fig 31, the element at position 45,

<e>e_text</e>

which is defined by the start tag “<e>”, closing tag </e>” and has element content, “e_text”.

This could be replaced with any of the following elements:

<z>e_text</z> (title change)

<e>z_txt </e> (text change with extra space char as padding)

<z>z_txt </z> (title and text change as described above)

This would not require any change to the MDNA as the size (in character bytes) of the replacement text is the same size as the original content.

Replacement text larger than the original content can also be dealt with, but will require changing the MDNA. The node in the MDNA that refers to the updated element (or text) can be changed to point to a new element at the end of the source XML file. The original element will be left in place, but is orphaned as no MDNA node refers to it. However, such a change would destroy the validity of the XML file and therefore we have chosen the strategy of hiding the new element within a comment tag. By including the new element in a comment, the updated source XML document does not accurately represent an updated version of the document. Thus the MDNA together with the

source XML document is a correct representation of the updated document. It should be noted that any query system would of necessity have to operate on both the updated XML document and MDNA to find particular nodes. If it is desired to have a single correct XML source document, all updates would have to be run on the updated XML and MDNA to write a new source XML file and a new MDNA would have to be built offline.

In Fig 11.1, a new node, a replacement for the element at position 28, has been placed in a comment at the end of the document. The node at position 4 in the MDNA (see Fig 10.8) for the document can then be updated so that its data_location value is 214 instead of 28.

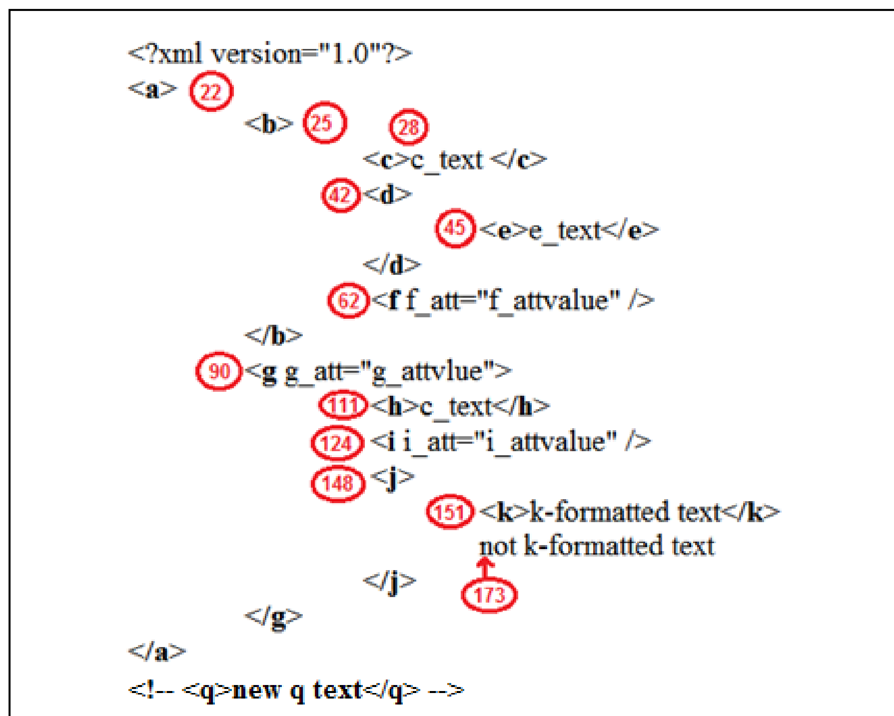


Fig 11.1 Target XML document showing data_location values and the new node

Best practice would suggest that if it was known that an XML document, using an MDNA query system, would be required to support a large number of updates, then it would be advisable to pad all node values (title, attribute, text) to a standard width so that all updates could be done by overwriting existing text rather than adding hidden nodes to the end of the document and modifying node data_location values. This extra padding would not change the size of the MDNA file. This practice is only possible when creating XML documents within the context of combining it with an MDNA representation. However, when an XML document is created by other means, there is no guarantee that padding has been implemented. This could be remedied by implementing a padding routine as a preprocessing of the XML before actually working with it. Any event driven parser such as EXPAT has the capability to perform such a transformation quickly and efficiently, even for very large documents (2-4 Gbytes). The growth of the file size would be more than offset by the efficiency of the MDNA representation in performing queries and operations on the file.

11.4.2 Deletion Operations

Deletion operations for text content can be accomplished easily by simply overwriting the existing text content with blank spaces. Deleting a node (removeChild()) in the DOM, when using an MDNA is as simple as removing any first_child or sibling reference to the node, thus making the node unreachable. A node that is an only child can be removed simply by setting its parent nodes first child reference to 0. If a node is a part of a sibling group, it can be deleted by moving it to the end of its group of siblings, and changing the relevant sibling status value of the node preceding it to that of an end

child. This again, will make the node unreachable, effectively deleting if from the DOM.

As with update operations that change the MDNA (and insertion operations, which are detailed below) after an node is deleted from the MDNA in order to commit the changes a new XML document based on the MDNA would need to be written to file.

11.4.3 Insertion Operations

Insertion operations (insertBefore()) are the most complex operations when working with an MDNA. The text values of the node themselves can be added to the end of the document as was detailed in the updating operations section (11.4.1) but the MDNA nodes will need to be changed as well in order to complete the insertion. Adding a node to a group of siblings will make the first child values of all preceding nodes in the MDNA inaccurate by one. After a node has been inserted all preceding node will need to be updated. One possibility is to insert place holder elements within the source XML as was suggested for padding. Then an insertion would just be an update for a blank node. Better procedures for adapting the MDNA representation for updates and insertions is a subject for future research (See future research).

12. C/C++ implementation of an MDNA parser

As a proof of concept, a complete MDNA DOM Parser command line program has been created using the C programming language. This DOM parser does not attempt to completely support the DOM specification at any level, but does implement the core DOM operations listed in section 12. In addition the MDNA Parser in C supports more complex “tree structure” queries using location paths. The structure of these location path queries has been implemented to be compatible with an abbreviated XPath syntax.

The C/C++ programming language has several features that make it ideal for this application. First, C is a widely used programming language [38], [39] and there are very few computer architectures for which a C compiler does not exist. Many systems that have limited memory would benefit from a memory efficient and fast querying system. These include robotic platforms [40] [41] , microcontrollers [42] and web servers [43]. These systems all support the C programming language. In addition, writing the parser in C allowed for greater control of the exact cost of memory allocation and the identification of good performance metrics for processing very large XML documents.

In should be noted that this C implementation does not support the return of nodes as objects, nor does it support the return of node lists in any of its functions. This limitation is intentional and designed to control the size of the program by prohibiting users from creating and returning large lists of node objects. Unlike the DOM API, the functions in the C MDNA Parser developed here enforce the use of singular “current node” data structure which can be queried, manipulated, returned. All DOM operations are performed on this current node structure in conjunction with the MDNA. There can

only be one “current node” at any time, and any use of tree navigation functions (getParent()) in effect changes the current node. Queries that return multiple nodes are possible, but the query result is a concatenation of the nodes rendered as strings, resulting from processing multiple individual nodes sequentially.

12.1 Structure of the C program

The MDNA program is composed of three core C files: STACKHandler.c, MDNAHandler.c and DOMHandler.c. These three files, together with their associated header files, allow the creation, manipulation and querying of any XML document. At present there is a limit of 2GB on XML documents for which an MDNA can be created and used. This limitation is a result of the limitation of the fread () and fwrite() function in C in combination with the 32 bit size unsigned integers declared in the MDNA nodes. Larger files can easily be supported by adapting the MDNA nodes to use 64 bit integers and enabling large file support (LFS) for fread() and fwrite() functions. LFS is supported by C compilers when a compilation flag is enabled by including a statement such as _FILE_OFFSET_BITS==64 on Linux systems. A complete API for the MDNA Parser program is available online (www.sci.brooklyn.cuny.edu/~meyer/thesis). A partial listing of the API is listed in Appendix D, and brief descriptions of the functionality of each of the individual files are given below as follows:

12.1.1 STACKHandler.c and STACKHandler.h

These two files contain the structures, variables, public functions and static

functions used in the creation and manipulation of a large stack data structure. The stack is implemented as a temporary file. To increase response time for stack operations, the stack functions make use of 2 buffers (SB1 and SB2), which hold parts of the stack that are currently needed. The 2 buffers SB1 and SB2 are structures of type stackFileBuf. The size of the two buffers can be controlled by the user using the stack_SETBUFSIZE() function; the default size for the two buffers is 80 bytes.

The functions provided by STACKHandler.c allow for the implementation of a traditional stack (push, pop, peek) that stores uint32_t variables, while also allowing random access to stack values if necessary. The peek operation on the stack was included for use in building the MDNA, which required random access to stack contents. This operation was not costly for the array data structure utilized to build the stack.

12.1.2 MDNAHandler.c and MDNAHandler.h

These two files contain the structures, variables, public functions and static functions used in the creation and manipulation of an MDNA. The MDNA is maintained as a file that is named based on the source XML file and with the file extension ".mdna". To increase access speed, the file was cached by loading parts of the file into multiple small independent buffers. The number and size of the buffers can be configured by the user using the mdna_SETBUFFERS() function. By default, the system will create 10 MDNA buffers using a total of16KB of RAM. The MDNA buffers are part of a single sequential memory block, but each of the buffer's contents are independent of the contents of the other buffers. A "least recently used" algorithm was used to govern which buffers were overwritten to satisfy queries when the information was not found in

the cache. Additional heuristics, such as maintaining the buffers in a sorted list were also implemented to improve the likelihood of finding the information in a desired buffer quickly.

The functions provided by MDNAHandler.c allow the user to rapidly create an MNDA file from an XML file as well as manipulate the MDNA and extract information (first child, sibling status) from the nodes that make up the MDNA. Additional functions for the tracking of the statistical information (tree depth, cache hits, node count) of the MDNA were also implemented.

12.1.3 DOMHandler.c and DOMHandler.h

These two files contain the structures, variables, public functions and static functions needed to support the DOM functions and other associated Query operations on the MDNA. The DOMHandler files also manage the XML source documents which are still needed to answer text based (title, attributes and value) queries. Similar to the stack and the MDNA cache implementation, the source XML file was also cached using multiple buffers. The number and size of the buffers were made configurable by the user using the dom_SETBUFFERS() function. By default the system will create 10 file buffers using a total of16KB of RAM. As with the MDNA, “least recently used” algorithms were implemented to govern which buffers are overwritten to satisfy queries.

The functions provided by DOMHandler.c allow the user to execute DOM queries using an MDNA on a given source XML file (see section 12). Additional functions also all for the tracking of the statistical information (tree depth, cache hits) of the logical DOM tree. Users can also use the dom_QUERY() function to execute simple “tree

structure" queries using location paths. The structure of these location path queries were implemented to be compatible with abbreviated XPath syntax, and are detailed in the next section.

12.1.4 main.c

The main.c file brings together the stack, MDNA and DOM files into a working program. The main.c files is responsible for calling the open(), create() and close() functions for the stack, MDNA and DOM sections. In addition the main file handles input from the user, including requests to change the buffer numbers and sizes for the stack, MDNA and DOM sections.

The current implementation of the C MDNA parser can take input from the user in two different modes. In "Interactive Mode" the user can use the keyboard to directly call DOM functions to examine the XML file. In "Query Mode" the user can submit tree structure queries to retrieve specific parts of an XML document. Help is available throughout the program (see figure 12.1).

```
This is the MDNAParser, version 1.1
by M. Meyer 2012 (meyer@sci.brooklyn.cuny.edu)

USAGE: MDNAParser(.exe) [ flags <values> ] <xmlFile.xml>

NOTE: User must at minimum provide the name of an XML file to use.

EXAMPLE (1): MDNAParser.exe myXMLfile.xml
EXAMPLE (2): MDNAParser.exe -c -s 10485760 -m 10 104857600 myXMLfile.xml

DEFAULTS: This program by default opens in interactive mode allowing
a user to use the arrow keys,'a' key (attribute) and 'v' key (value)
to manually navigate through the logical DOM tree within the given
source XML file. By default the program will create an MDNA file for the
source XML file ONLY if one does NOT already exist.

FLAGS: The following flags (some flags require values) allow a user to change when
MDNA files are created, the mode (interactive/query) as well as the number and size of
buffers to use in the program.
 -d                  -> Do not create an MDNA file even if one DOES NOT exist
 -c                  -> Create an MDNA file even if one exists already
 -q  "query"         -> Open in query mode and parse the query that follows  -q
 -o  file            -> Redirect all output to the file specified
 -s  value           -> Set the size of the stack buffer to 'value' (in bytes)
 -m  value num       -> Use 'value' bytes and 'num' buffers to buffer the MDNA file
 -x  value num       -> Use 'value' bytes and 'num' buffers to buffer the XML file
```

Fig 12.1 Output from the MDNA parser help screen.

12.2 Flow diagram of the organization of the system

The source XML file, the MDNA file, the temporary stack file and the core operation files interact to create a working program. A flow diagram of that interaction can be seen in Fig 12.2

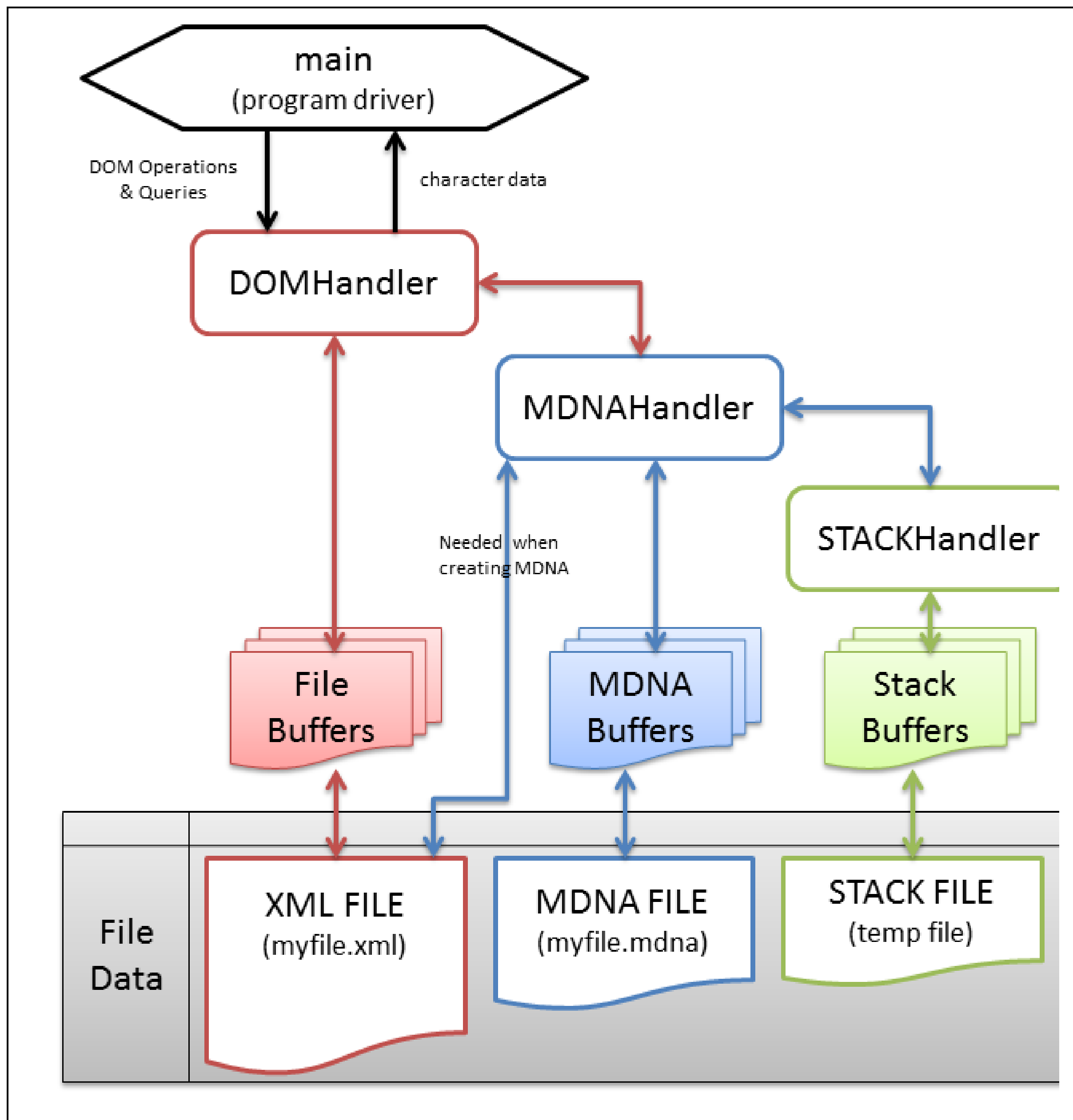


Fig 12.2 Flow diagram of MDNA parser program system components

12.3 DOM and Query Support

At this time the C MDNA parser supports many (but not all) the official W3C DOM Level 1 node functions. The following functions are supported:

- char * dom_nodeName ()

  // Returns the title or name of the current Node.

- char * dom_attributes ()

  // Returns the attributes for the current Node.

- char * dom_nodeValue ()

  // Returns the content for the node; null for some node types.

- int dom_nodeType ()

  // Returns the node type of the current Node.

- int dom_parentNode ()

  // Sets the current node to the parent of the current Node.

- int dom_hasChildNodes ()

  // Returns 1 if the node has children 0 otherwise

- int dom_firstChild ()

  // Sets the current node to the first child of the current Node.

- int dom_lastChild ()

  // Sets the current node to the last child of the current Node.

- int dom_previousSibling ()

  // Sets the current node to the previous sibling of the current Node.

- int dom_nextSibling ()

// Sets the current node to the next sibling of the current Node.

- void dom_removeNode ()

  // Removes the current node, and moves system to the parent of the

  // removed node. It may also make changes to parent and sibling nodes.

Many of these DOM functions can be used directly in "Interactive Mode" which can be entered by running with the MDNA program with a source XML file and no other input.

```
***** ****** ***** ****** ****** ******
* INTERACTIVE MODE
*
* Use the following keys to navigate the DOM tree:
* (Hint: Use NumLock and the number keys for navigation)
*
*   8 -> Go to parent of current node
*   2 -> Go to first child of current node
*   4 -> Go to preceding sibling of current node
*   6 -> Go to next sibling of current node
*
*   t -> Print the title of the current node
*   a -> Print the attributes of the current node
*   v -> Print the value of the current node
*
*   h -> Print the value of the history array
*   p -> Expand the history array to show the "path" back to the root
*   n -> Print the current Node and it's descendants (up to 1024 characters)
*
*   q -> QUIT
***** ****** ***** ****** ****** ******
```

Fig 12.3 Interactive Mode - Instructions Screen

Users can also send queries to the MDNA program using tree location syntax. The query, sent as input to the program when it is called, is evaluated by the program which will then print out the result to the query as a string. The syntax used should look familiar to XPATH users as it is similar to the abbreviated syntax used by that language. It should be clear that the MDNA parser does not completely support the XPath query language, nor was that an intention of this program.

```
***** ****** ***** ****** ****** ******
* QUERY MODE
*
* You must enter a valid query! Queries are composed of tree position
* references linked together in a complete path specification.
*
* There are 4 possible position references:
*
*     /NAME                  -> child node with name NAME
*     /NAME[.='abc']         -> child node with name NAME and text value abc
*     /NAME[@id='123']       -> child node with name NAME and attribute id has value 123
*     /..                    -> Go to the parent of the current node
*
* When a node is found that matches all position references it is returned as a string.
*
* EXAMPLES:
* -q "/PLANT/LIGHT/i[.='Usually']/../.."
* -q "/PLANT/COMMON[.='Hepatica']/i[@id='123']/.."
*
***** ****** ***** ****** ****** ******
```

Fig 12.4 Query Mode - Instructions Screen

12.4 Program Performance

The C MDNA DOM parser program is currently able to work with XML documents up to 2GB in size. This limitation, which can easily be removed, has been explained above. The parser supports the bulk of the W3C Dom 1 operations as well as limited tree location queries sufficient to locate specific nodes by name, attribute and value parameters. The program is able to do all of this on a low and fixed allocation of memory, independent of the size of the source documents and of the MDNA generated to use with the source document. Currently, on the default settings, the program runs in less than 700KB of space regardless of the size of the source document. On the lowest settings the application runs in less than 600KB. Further refinement of the program could still significantly reduce this size.

The MDNA parser compiles and runs on Windows, MAC and Linux machines because it has been implemented in C. It is currently available online at (www.sci.brooklyn.cuny.edu/~meyer/thesis).

13. Future Research

There a number of extensions of this work. Some can easily be implemented while others would require considerable effort. Here we outline a few of the important possibilities for future work.

13.1 Implementation of an MDNA parser on a Microcontroller, hand held device, robot or other small programmable device with limited memory.

The very small system requirements of the MDNA parser make it an excellent candidate for use on systems with very little resources such as hand-held devices, microcontrollers and robots.

The field of robotics shows tremendous promise as an area for deployment of the MDNA parser as the inclusion of the parser as part of a program running a robot would allow an autonomous robot to use an XML file (perhaps located on a removable USB drive) as a portable database for making decisions about navigation or other agent-based decision problems.

13.2 Full support of W3C DOM using a MDNA program written in C++

Complete support of the W3C DOM interface will not be possible without moving to a language which supports objects as the DOM interface requires node objects and node lists objects as return types for some functions. Rewriting the MDNA parser in C++ (with care to avoid hidden costs from class pointers) will allow the creation of a parser that can fully support DOM.

13.3 Development of more efficient algorithms for caching DOM tree information.

Currently the file buffers used to cached parts of the source XML, MDNA and stack file all use a Least Recently Used algorithm to determine which buffers will be overwritten to satisfy new requests. It would be more efficient to use an algorithm to determine whether or not buffer contains elements that are currently part of the DOM branch that is being explored.

13.4 Adaptation of MNDA to support DOM Level 3 Load and Save Specification

The Document Object Model (DOM) Level 3 Load and Save Specification paper (2004) defines a set of interfaces for loading and saving document objects as defined in [DOM Level 2 Core] or newer. Having gone through the cost of building a DOM tree this

interface describes a function set to allow the saving (the serialization) of the DOM tree as a separate file that can later be reloaded.

An MDNA file is a serialization of the trees structure information of a DOM XML representation. This could be quickly and easily modified to meet the W3C document specifications.

13.5 Support for multi-threading during queries

During the execution of a query the MNDA parser accesses the source files as little as possible, principally to load its own buffers. The program is already divided into separate logical components DOMHandler, MDNAHandler and STACKHandler; these separate components could run as individual threads and be used to jointly work on complex queries vastly increasing response rate for very queries within very large XML files. Multiple copies of these individual components could be run in parallel as individual threads and further increase the efficiency of the system.

13.6 Benchmarking MDNA versus other DOM parsers

The MNDA parser has the advantage of being able to support DOM query operations on XML files of almost any size at a low and fixed cost. Although observably very fast when querying simple and small XML files, genuine benchmarking tests against other DOM parsers would be necessary to objectively evaluate the use of MDNA as an adjunct to DOM parsing.

13.7 Support for XPath and XQuery

XPath and XQuery are query languages for selecting nodes from within an XML document. Both languages are popular, but not widely implemented outside of Native Xml Databses (NXDs). However, they are considered more powerful and more useful than simple DOM. In general the XPath is a language used to succinctly pinpoint exact XML nodes in a DOM. XQuery can be considered as a superset of XPath that also provides FLWOR (FOR, LET, WHERE, ORDER, RETURN) syntax, which is SQL-like.

A more robust, C++ based, multi-threaded MDNA parser could be expanded to support the more advanced query languages.

13.8 Closure under TAX

Tree Algebra for XML (TAX) is a formal bulk algebra for applying database-style optimization to XML queries. TAX builds upon relational algebra by considering collections of XML trees instead of relations as the basic unit of manipulation. TAX has only a few operators more than relational algebra despite the increased complexity of working with trees (instead of tuples) and the heterogeneity of most XML files.

Many of the requirements of a TAX based database system (pedigree element, unique tree labels/keys) have analogues when using MDNA (history stack, data_location reference). As part of the process of developing an MDNA parser to support

XPath and XQuery it would be advantageous to simultaneously show that an MDNA parser meets the operational requirements of TAX and is closed under that algebra.

14. Summary and Major Contributions of this Thesis

- Analyzing in depth the costs associated in building traditional DOM trees from XML documents using pointers and linked objects.
    - Detailing costs associated with traditional DOM modeling techniques (node structures, pointers, captured data).
    - Examining how the arrangement of the underlying logical XML element tree can induce vast increases in the size required to accomplish traditional XML DOM modeling of a given XML document (bar and beam models).
    - Examining other less commonly understood features of traditional XML DOM that can also contribute to vast increases in the size a traditional DOM tree (class inheritance pointers, pedigree elements).
    - Establishing upper and lower bounds on the size of a DOM tree based on the maximum possible number of elements within an XML document.
- Demonstrating several methods by which the cost of DOM modeling, in particular the size of the DOM tree can be reduced:
    - Data Indexing: Using document offset values to store the location of information rather than duplicating the information in the DOM tree.
    - Eliminating Nodes: Using a single data_location value to pinpoint the information for several nodes, reducing or eliminating the need to create nodes for XML attribute and text information.

  - Array Tree Storage: Replacing the traditional pointer based DOM tree with an array of nodes that still retains all tree information necessary to support DOM modeling.

- Establishing an alternative array data structure for DOM modeling the Minimum DOM Node Array (MDNA).
  - Detailing the minimum amount of information necessary to store in a node to support Minimum DOM Node (MDN) modeling.
  - Illustrating a methodology for efficiently creating an MDN Array (MDNA).
    - Detailing a new sorting algorithm (Swiss Cheese Sort) of particular use in sorting data that is already very highly sorted.
- Creating a MDNA parser application to demonstrate the validity of the MDNA approach to DOM modeling.
  - Application supports most W3C DOM CORE Level 1 operations.
  - Application supports limited XPath queries in abbreviated syntax form.

Appendix A: Document Object Model Level 1 Core IDL Interfaces

This section contains the OMG IDL definitions for the interfaces in the Core Document Object Model specification, including the extended (XML) interfaces.

```
exception DOMException {
  unsigned short   code;
};

// ExceptionCode
const unsigned short     INDEX_SIZE_ERR     = 1;
const unsigned short     DOMSTRING_SIZE_ERR = 2;
const unsigned short     HIERARCHY_REQUEST_ERR = 3;
const unsigned short     WRONG_DOCUMENT_ERR = 4;
const unsigned short     INVALID_CHARACTER_ERR = 5;
const unsigned short     NO_DATA_ALLOWED_ERR = 6;
const unsigned short     NO_MODIFICATION_ALLOWED_ERR = 7;
const unsigned short     NOT_FOUND_ERR      = 8;
const unsigned short     NOT_SUPPORTED_ERR  = 9;
const unsigned short     INUSE_ATTRIBUTE_ERR = 10;

// ExceptionCode
const unsigned short     INDEX_SIZE_ERR     = 1;
const unsigned short     DOMSTRING_SIZE_ERR = 2;
const unsigned short     HIERARCHY_REQUEST_ERR = 3;
const unsigned short     WRONG_DOCUMENT_ERR = 4;
```

```
const unsigned short     INVALID_CHARACTER_ERR = 5;
const unsigned short     NO_DATA_ALLOWED_ERR = 6;
const unsigned short     NO_MODIFICATION_ALLOWED_ERR = 7;
const unsigned short     NOT_FOUND_ERR      = 8;
const unsigned short     NOT_SUPPORTED_ERR  = 9;
const unsigned short     INUSE_ATTRIBUTE_ERR = 10;

interface DOMImplementation {
  boolean                hasFeature(in DOMString feature,
                                    in DOMString version);
};

interface DocumentFragment : Node {
};

interface Document : Node {
  readonly attribute  DocumentType        doctype;
  readonly attribute  DOMImplementation    implementation;
  readonly attribute  Element             documentElement;
  Element                createElement(in DOMString tagName)
                                       raises(DOMException);
  DocumentFragment        createDocumentFragment();
  Text                  createTextNode(in DOMString data);
  Comment                 createComment(in DOMString data);
  CDATASection            createCDATASection(in DOMString data)
                                         raises(DOMException);
  ProcessingInstruction    createProcessingInstruction(in DOMString target,
                                              in DOMString data)
```

```
                                 raises(DOMException);
  Attr                  createAttribute(in DOMString name)
                              raises(DOMException);
  EntityReference       createEntityReference(in DOMString name)
                                 raises(DOMException);
  NodeList              getElementsByTagName(in DOMString tagname);
};

interface Node {
  // NodeType
  const unsigned short     ELEMENT_NODE                    = 1;
  const unsigned short     ATTRIBUTE_NODE                  = 2;
  const unsigned short     TEXT_NODE                       = 3;
  const unsigned short     CDATA_SECTION_NODE              = 4;
  const unsigned short     ENTITY_REFERENCE_NODE           = 5;
  const unsigned short     ENTITY_NODE                     = 6;
  const unsigned short     PROCESSING_INSTRUCTION_NODE     = 7;
  const unsigned short     COMMENT_NODE                    = 8;
  const unsigned short     DOCUMENT_NODE                   = 9;
  const unsigned short     DOCUMENT_TYPE_NODE              = 10;
  const unsigned short     DOCUMENT_FRAGMENT_NODE          = 11;
  const unsigned short     NOTATION_NODE                   = 12;

  readonly attribute  DOMString          nodeName;
           attribute  DOMString          nodeValue;
                                 // raises(DOMException) on setting
                                 // raises(DOMException) on retrieval
  readonly attribute  unsigned short     nodeType;
```

```
  readonly attribute  Node              parentNode;
  readonly attribute  NodeList          childNodes;
  readonly attribute  Node              firstChild;
  readonly attribute  Node              lastChild;
  readonly attribute  Node              previousSibling;
  readonly attribute  Node              nextSibling;
  readonly attribute  NamedNodeMap      attributes;
  readonly attribute  Document          ownerDocument;
  Node                insertBefore(in Node newChild,
                                   in Node refChild)
                                   raises(DOMException);
  Node                replaceChild(in Node newChild,
                                   in Node oldChild)
                                   raises(DOMException);
  Node                removeChild(in Node oldChild)
                                  raises(DOMException);
  Node                appendChild(in Node newChild)
                                  raises(DOMException);
  boolean             hasChildNodes();
  Node                cloneNode(in boolean deep);
};

interface NodeList {
  Node                item(in unsigned long index);
  readonly attribute  unsigned long     length;
};

interface NamedNodeMap {
```

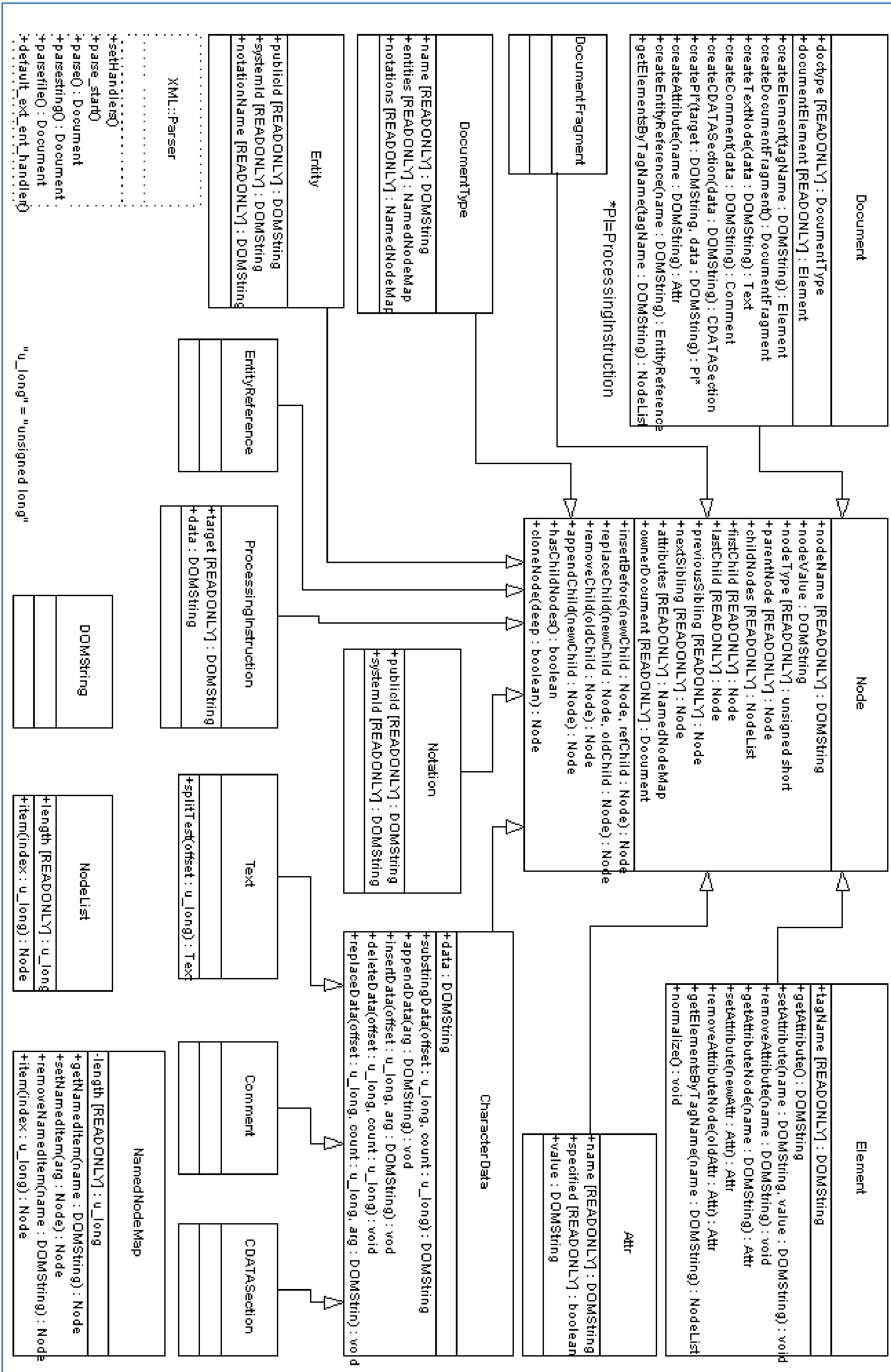
```
  Node                  getNamedItem(in DOMString name);
  Node                  setNamedItem(in Node arg)
                                raises(DOMException);
  Node                  removeNamedItem(in DOMString name)
                                   raises(DOMException);
  Node                  item(in unsigned long index);
  readonly attribute  unsigned long       length;
};

interface CharacterData : Node {
         attribute  DOMString           data;
                              // raises(DOMException) on setting
                              // raises(DOMException) on retrieval
  readonly attribute  unsigned long       length;
  DOMString                substringData(in unsigned long offset,
                                    in unsigned long count)
                                    raises(DOMException);
  void                  appendData(in DOMString arg)
                                 raises(DOMException);
  void                  insertData(in unsigned long offset,
                                 in DOMString arg)
                                 raises(DOMException);
  void                  deleteData(in unsigned long offset,
                                 in unsigned long count)
                                 raises(DOMException);
  void                  replaceData(in unsigned long offset,
                                  in unsigned long count,
                                  in DOMString arg)
```

```
                              raises(DOMException);
};

interface Attr : Node {
  readonly attribute  DOMString           name;
  readonly attribute  boolean             specified;
           attribute  DOMString           value;
};

interface Element : Node {
  readonly attribute  DOMString           tagName;
  DOMString                getAttribute(in DOMString name);
  void                     setAttribute(in DOMString name,
                                        in DOMString value)
                                        raises(DOMException);
  void                     removeAttribute(in DOMString name)
                                           raises(DOMException);
  Attr                     getAttributeNode(in DOMString name);
  Attr                     setAttributeNode(in Attr newAttr)
                                            raises(DOMException);
  Attr                     removeAttributeNode(in Attr oldAttr)
                                               raises(DOMException);
  NodeList                 getElementsByTagName(in DOMString name);
  void                     normalize();
};

interface Text : CharacterData {
  Text                     splitText(in unsigned long offset)
```

```
                    raises(DOMException);
};

interface Comment : CharacterData {
};

interface CDATASection : Text {
};

interface DocumentType : Node {
  readonly attribute  DOMString         name;
  readonly attribute  NamedNodeMap        entities;
  readonly attribute  NamedNodeMap        notations;
};

interface Notation : Node {
  readonly attribute  DOMString         publicId;
  readonly attribute  DOMString         systemId;
};

interface Entity : Node {
  readonly attribute  DOMString         publicId;
  readonly attribute  DOMString         systemId;
  readonly attribute  DOMString         notationName;
};

interface EntityReference : Node {
};
```

```
interface ProcessingInstruction : Node {
  readonly attribute  DOMString          target;
           attribute  DOMString          data;
                                // raises(DOMException) on setting
};
```

Appendix B: Logical DOM View (CORE) Level One



Appendix C: W3C IDL node interface specification v. 1.0.

interface Node {

```
  // NodeType

  const unsigned short      ELEMENT_NODE       = 1;

  const unsigned short      ATTRIBUTE_NODE     = 2;

  const unsigned short      TEXT_NODE          = 3;

  const unsigned short      CDATA_SECTION_NODE = 4;

  const unsigned short      ENTITY_REFERENCE_NODE = 5;

  const unsigned short      ENTITY_NODE        = 6;

  const unsigned short      PROCESSING_INSTRUCTION_NODE = 7;

  const unsigned short      COMMENT_NODE       = 8;

  const unsigned short      DOCUMENT_NODE      = 9;

  const unsigned short      DOCUMENT_TYPE_NODE = 10;

  const unsigned short      DOCUMENT_FRAGMENT_NODE = 11;

  const unsigned short      NOTATION_NODE      = 12;


  readonly attribute  DOMString            nodeName;

           attribute  DOMString            nodeValue;

                                   // raises(DOMException) on setting

                                   // raises(DOMException) on retrieval


  readonly attribute  unsigned short       nodeType;

  readonly attribute  Node                 parentNode;

  readonly attribute  NodeList             childNodes;

  readonly attribute  Node                 firstChild;

  readonly attribute  Node                 lastChild;
```

```
  readonly attribute  Node                  previousSibling;
  readonly attribute  Node                  nextSibling;
  readonly attribute  NamedNodeMap          attributes;
  readonly attribute  Document              ownerDocument;

  Node                      insertBefore(in Node newChild,
                                         in Node refChild)
                                         raises(DOMException);

  Node                      replaceChild(in Node newChild,
                                         in Node oldChild)
                                         raises(DOMException);

  Node                      removeChild(in Node oldChild)
                                        raises(DOMException);

  Node                      appendChild(in Node newChild)
                                        raises(DOMException);

  boolean                   hasChildNodes();

  Node                      cloneNode(in boolean deep);
};
```

Appendix D: Limited API for C MDNA Parser

Data Structures

Here are the data structures with brief descriptions:

<table>
<tr>
<td>domFileBuf</td>
<td>These buffer structures hold sections of the XML file

Data Fields
<table>
<tr><td>uint32_t</td><td>startPos</td></tr>
<tr><td>uint32_t</td><td>size</td></tr>
<tr><td>uint32_t</td><td>age</td></tr>
<tr><td>char *</td><td>buff</td></tr>
<tr><td>uint8_t</td><td>empty</td></tr>
</table>
Detailed Description

To increase access speed parts of the XML file are loaded into buffers. These buffers are part of a single sequential memory block, but each buffers contents are independent of the content of the other buffers.</td>
</tr>
<tr>
<td>mdnaFileBuf</td>
<td>These buffer structures hold sections of the MDNA file

Data Fields
<table>
<tr><td>uint32_t</td><td>startPos</td></tr>
<tr><td>uint32_t</td><td>age</td></tr>
</table>
</td>
</tr>
</table>

<table>
<tr><td></td><td>

MDN *	**buff**
uint8_t	**empty**

Detailed Description

The MDNA is actually a file. To increase access speed parts of the file are loaded into buffers. These buffers are part of a single sequential memory block, but each buffers contents are independent of the content of the other buffers.
</td></tr>
<tr><td>MDN</td><td>

Data Fields

uint32_t	**data_location**
uint32_t	**first_child**
</td></tr>
<tr><td>node</td><td>
This structure is used to capture information relating to the currentNode

Data Fields

uint32_t	**MNDAPosition**
	Location of this node in the MDNA file.
uint32_t	**data_location**
</td></tr>
</table>

		copy of **MDN** data_location value
	uint32_t	**first_child**
		copy of **MDN** first_child value
	int	**type**
		Type of this node (12 possibilities)
	char	**title** [1001]
		Title of this node in the XML file.
	char	**attributes** [20][1001]
		Attributes of this node in the XML file.
	int	**attributeCount**
		Count of attributes of this node in the XML file.
	uint32_t	**textLocation**
		Does this node have an immediate text node.
	char	**value** [4096]
		For capturing text values associated with a node.
	int	**siblingNumber**
		What number child is this (including any text nodes)
	int	**siblingCount**

<table>
<tr><td></td><td><table><tr><td></td><td>How many other siblings are there total (including any text nodes)</td></tr></table></td></tr>
<tr><td>stackFileBuf</td><td>These buffer structures hold sections of the stack file

Data Fields<table><tr><td>uint32_t</td><td>startPos</td></tr><tr><td>uint32_t *</td><td>buff</td></tr></table></td></tr>
</table>

DOMHandler.h File Reference

Header file with structures and functions used in the DOM functions.

Detailed Description

Header file with structures and functions used in the DOM functions.

This file contains the structures and functions used to create and access the contents of an XML file using DOM modeling with an MDNA. NOTE: This application only allows one node at a time to be queried and does not allow the duplication of nodes. At any given moment the system has, as part of its internal state, the currentNode which is the node for which data such as title, value and attributes can be retrieved. Functions like dom_GET_PARENT(), change the internal state of the program by changing the node that is the currentNode.

Warnings:

(1) Do not try to create copies of currentNode. currentNode without the complete value of the stack is useless. The application is designed to be small and fast and free of objects.

(2) The MDNA array cannot hold more than 268,435,456 (hex 0x10000000) **MDN** structures. Going beyond that limit will cause problems. This limit is imposed by fread()/fwrite() which cannot address files larger then 2GB on most systems without Large File Support (LFS). Since LFS support implementation varies across systems and the goal of this program is to be able to work on as may platforms as possible the MDNAHandler does not currently use LFS. The define MAX_ARR_SIZE is used to refer to the maximum size limit

of the **MDN** array. If you change the program to use LFS (you can do this by compiling with _FILE_OFFSET_BITS == 64 on Linux; see http://www.gnu.org/software/libc/manual/html_node/Opening-Streams.html for more details) make sure you update the value for MAX_ARR_SIZE as well.

Author:

Matthew K. Meyer

Data Structures

struct	**domFileBuf**
	These buffer structures hold sections of the XML file.
struct	**node**
	This structure is used to capture and retain information relating to the currentNode.

Macros

#define	**DOM_DEFAULT_BUFFSIZE** 16384
	Default size, in bytes, for memory used to buffer the XML file.
#define	**DOM_MAX_BUFFSIZE** 0x10000000 /* 500MB */
	Maximum size, in bytes, for memory used to buffer the XML file.
#define	**DOM_DEFAULT_BUFFNUM** 10
	Default number of file buffers to create for the XML file.
#define	**DOM_MAX_BUFFNUM** 20

	Maximum number of file buffers to create for the XML file.
#define	**XML_FMT_INT_MOD** "l"
	For EXPAT compatibility.
#define	**SS_FROM_32**(x) ((x) >> 30)
	Extracts sibling status value from an uint32_t var (first_child)
#define	**FC_FROM_32**(y) ((y) & 0x3FFFFFFF)
	Extracts first child value from a uint32_t var (first_child)
#define	**SS_FC_INTO_32**(x, y) (((x) << 30) + (y))
	Combines a sibling status value (0-3) with another uint32_t variable (representing a position in in the **MDN** array) into one value.

Function Documentation

uint32_t	**dom_SETBUFFERS** (uint32_t buffSize, uint32_t buffNum)
	Sets the size and number of buffers to be used when buffering the XML file as part of the query answer process.
uint32_t	**dom_OPEN** (const char *xfn, int cm)
	Opens the XML file (associated with the xml file name given) sets up the necessary buffers and calls appropriate MDNA and stack functions.
uint32_t	**dom_CLOSE** (const char *xfn)
	Closes MDNA file associated with the xml file and the XML file given (if it is

	open) and frees all memory allocated for buffers.
uint32_t	**dom_GET_NODE** (uint32_t nodePos)
	Populates the current node with data from the desired element.
void	**dom_PRINT_CURRENT_NODE** ()
	Helper function to examine internal state of currentNode.
char *	**dom_nodeName** ()
	Returns the title or name of the current Node.
char *	**dom_attributes** ()
	Returns the attributes for the current Node.
int	**dom_attribute_count** ()
	This specifies the number of characters strings in attribute array.
char *	**dom_nodeValue** ()
	Returns the content for the node; null for some node types.
int	**dom_nodeType** ()
	Returns the node type of the current Node.
int	**dom_parentNode** ()
	Sets the current node to the parent of the current Node.
int	**dom_hasChildNodes** ()

	Returns 1 if the node has children 0 otherwise.
int	**dom_firstChild** ()
	Sets the current node to the first child of the current Node.
int	**dom_lastChild** ()
	Sets the current node to the last child of the current Node.
int	**dom_getChild** (int pos)
	Sets the current node to the node specified by pos.
int	**dom_previousSibling** ()
	Sets the current node to the previous sibling of the current Node.
int	**dom_nextSibling** ()
	Sets the current node to the next sibling of the current Node.
void	**dom_removeNode** ()
	Removes the current node from the MDNA and updates appropriate parent and sibling references.
char *	**dom_PATH_TO_ROOT** ()
	Prints out an expanded version of the stack array.
char *	**dom_QUERY** (char *q)
	Processes and returns the answers to simple VERY simple queries.
char *	**dom_NODE_TO_STRING** ()

	Helper function to create string out of node information.

Macro Definition Documentation

#define DOM_DEFAULT_BUFFNUM 10

Default number of file buffers to create for the XML file. Even for very large XML documents tree depth rarely exceeds 10.

#define DOM_DEFAULT_BUFFSIZE 16384

Default size, in bytes, for memory used to buffer the XML file. Set to 16384 (16KB) by default which is 1/2 the standard 32KB L1 Data cache available on most CPU's.

#define DOM_MAX_BUFFNUM 20

Maximum number of file buffers to create for the XML file.

As the number of buffer grows the time it takes to determine that the desired variable is NOT in a buffer grows. We have yet to encounter an XML file where DOM tree depth exceeds 20 and thus limit DOM_MAX_BUFFNUM to 20.

MDNAHandler.h File Reference

Header file with structures and functions used in the MDNA.

Detailed Description

The Minimum DOM Node Array (MDNA) is an array of data structures that allows very large XML documents to be queried using the Document-Object-Model (DOM) model. This file contains the structures and functions used to create and access an MDNA.

Warning:

> The MDNA array cannot hold more than 268,435,456 (hex 0x10000000) **MDN** structures. Going beyond that limit will cause problems. This limit is imposed by fread()/fwrite() which cannot address files larger then 2GB on most systems without Large File Support (LFS). Since LFS support implementation varies across systems and the goal of this program is to be able to work on as may platforms as possible the MDNAHandler does not currently use LFS. The define MAX_ARR_SIZE is used to refer to the maximum size limit of the **MDN** array. If you change the program to use LFS (you can do this by compiling with _FILE_OFFSET_BITS == 64 on Linux; see http://www.gnu.org/software/libc/manual/html_node/Opening-Streams.html for more details) make sure you update the value for MAX_ARR_SIZE as well.

Author:

Matthew K. Meyer

Data Structures

struct	**MDN**
	This is the MDNA node. See thesis for more details.
struct	**mdnaFileBuf**
	These buffer structures hold sections of the MDNA file.

Macros

#define	**MDNA_DEFAULT_BUFFSIZE** 16384
	Default size, in bytes, for memory used to buffer the MDNA file.
#define	**MDNA_MAX_BUFFSIZE** 0x10000000 /* 500MB */
	Maximum size, in bytes, for memory used to buffer the MDNA file.
#define	**MDNA_DEFAULT_BUFFNUM** 10
	Default number of file buffers to create for the MDNA file.
#define	**MDNA_MAX_BUFFNUM** 20
	Maximum number of file buffers to create for the MDNA file.
#define	**MDNA_MAX_ARR_SIZE** 0x10000000 /* 268,435,456 */
	Indicates the maximum number of **MDN** structures the **MDN** array can contain. Default is 268,435,456 **MDN** positions, each of which requires 8 bytes of space making a file of size 2GB. 2GB is the maximum file size that fread() and fwrite() can address without LFS.

#define	**XML_FMT_INT_MOD** "l"
	For EXPAT compatibility.
#define	**SS_FROM_32**(x) ((x) >> 30)
	Extracts sibling status value from an uint32_t var (first_child)
#define	**FC_FROM_32**(y) ((y) & 0x3FFFFFFF)
	Extracts first child value from a uint32_t var (first_child)
#define	**SS_FC_INTO_32**(x, y) (((x) << 30) + (y))
	Combines a sibling status value (0-3) with another uint32_t variable (representing a position in in the **MDN** array) into one value.

Function Documentation

uint32_t	**mdna_SETBUFFERS** (uint32_t bSize, uint32_t bNum)
	Sets the size and number of buffers to be used when buffering the MDNA file.
uint32_t	**mdna_CREATE** (const char *xfn)
	This function creates an MDNA file for the XML file given as input.
uint32_t	**mdna_OPEN** (const char *xfn)
	Opens the MDNA file (associated with the xml file name given) sets up the necessary buffers.

uint32_t	**mdna_CLOSE** (const char *xfn)
	Closes MDNA file associated with the xml file and the XML file given (if it is open) and frees all memory allocated for buffers.
MDN	**mdna_GET** (uint32_t index)
	Returns the **MDN** at position index.
MDN	**mdna_SET** (uint32_t index, **MDN** x)
	Replaces the element at position index with **MDN** x and returns the **MDN** structure formerly at the specified index.
uint32_t	**mdna_FLUSHBUFFS** ()
	Writes the content of the MDNA to file saving them.
uint32_t	**mdna_FC** (uint32_t fc)
	This function extracts the first child value from an uint32 variable and returns it as a integer representing a position in the MDNA array.
uint32_t	**mdna_SS** (uint32_t fc)
	This fuction extracts the sibling status value from the uint32 variable and returns it as a integer.
uint32_t	**mdna_DELETE** (const char *xfn)
	Deletes an MDNA file with the same name as the source XML document (example a.xml would have a file called a.mdna which this function would then delete).

uint32_t	**mdna_GET_NODECOUNT** ()
	Returns the total number of nodes in the MDNA array.
uint32_t	**mdna_GET_TREEDEPTH** ()
	Returns the depth of the logical tree contained in the MDNA.
void	**mdna_PRINT_STATS** ()
	Quick way to check values on all the MDNA variables.
void	**mdna_PRINT_MDNAFILE** ()
	Prints the entire MDNA file to stdout formatting it so that values can be seen and confirmed ... see warning!

Macro Definition Documentation

#define MDNA_DEFAULT_BUFFNUM   10

Default number of file buffers to create for the MDNA file.

Even for very large XML documents tree depth rarley exceeds 10 which is the default setting; allows each individual buffer to hold 204 minimum DOM nodes when MDNA_DEFAULT_BUFFSIZE is left at its default.

#define MDNA_DEFAULT_BUFFSIZE   16384

Default size, in bytes, for memory used to buffer the MDNA file.

Set to 16384 (16KB) by default which is 1/2 the standard 32KB L1 Data cache available on most CPU's, and leaves room for 2048 MDNs

#define MDNA_MAX_BUFFNUM 20

Maximum number of file buffers to create for the MDNA file.

As the number of buffer grows the time it takes to determine that the desired variable is NOT in a buffer grows. We have yet to encounter an XML file where DOM tree depth exceeds 20 and thus limit MAX_BUFFNUM to 20.

STACKHandler.h File Reference

Header file with structures and public functions used in the stack.

Detailed Description

The stack is actually a temporary file. To increase response time stack functions make use of 2 buffers (SB1 and SB2) which hold parts of the stack that are currently needed. The 2 buffers SB1 and SB2 are structures of type **stackFileBuf**. You can use the functions in this file to implement a traditional stack (push, pop) that stores uint32_t variables.

Warning:

> The stack cannot hold more than 536,870,912 (in hex 0x20000000) unit32_t variables. Going beyond that limit will cause unpredictable results. This limit is imposed by fread() & fwrite() which cannot address files larger then 2GB on most systems without enabling Large File Support (LFS). Since LFS support implementation varies across systems and the goal of this program is to be able to work on as may platforms as possible the STACKHandler does not currently use LFS. The define MAX_STACK_SIZE is used to refer to the maximum size limit of the stack. If you change the program to use LFS (as an example you can do this by compiling with _FILE_OFFSET_BITS == 64 on Linux;
> see http://www.gnu.org/software/libc/manual/html_node/Opening-Streams.html for more details) make sure you update the value for MAX_STACK_SIZE as well.

Author:

Matthew K. Meyer

Data Structures

struct	**stackFileBuf**
	These buffer structures hold sections of the stack file.

Macros

#define	**DEFAULT_STACK_SIZE** 80
	Default size, in bytes, for total memory used by the stack.
#define	**MAX_STACK_SIZE** 0x20000000
	Maximum number of uint_32 size variables that this stack can hold.

Function Documentation

void	**stack_SETBUFSIZE** (uint32_t x)
	Allows you to reset the size of the buffers used by the stack, overriding the DEFAULT_STACK_SIZE value.
void	**stack_CREATE** ()
	Creates the temporary stack file and initializes the stack buffers.
void	**stack_DELETE** ()
	Deletes the stack file and frees the stack buffers.
void	**stack_RESET** ()
	Empties the stack of any content and resets all tracking values so that the

	stack can be used again.
void	**stack_PUSH** (uint32_t value)
	Pushes an uint32_t variable onto the stack.
uint32_t	**stack_POP** ()
	Pops an arrayPosition value off the stack.
uint32_t	**stack_PEEK** (uint32_t x)
	Returns a specific arrayPosition value off the stack.
void	**stack_PRINT_STATS** ()
	Prints the current value of all stack variables to stdout.
uint32_t	**stack_GET_TOTALBUFSIZE** ()
	Returns the total size (in bytes) of all stack buffers put together.
uint32_t	**stack_GET_COUNT** ()
	Returns the number of available positions (for variables of type uint32_t) in all of the stack buffers put together.
uint32_t	**stack_GET_POS** ()
	Returns the index of the next free position in the stack array.
uint32_t	**stack_GET_BUFSIZE** ()
	Returns the total size (in bytes) of each individual stack buffer.
uint32_t	**stack_GET_BUFCOUNT** ()

	Returns the number of available positions (for variables of type uint32_t) in each single buffer.
uint32_t	**stack_GET_MAXDEPTH** ()
	Returns the maximum size this that the stack has reached.
uint32_t	**stack_GET_BUFHITS** ()
	Returns a count of the number of times that a stack position was requested and that stack position was found in one of the stack buffers.
uint32_t	**stack_GET_BUFMISSES** ()
	Returns a count of the number of times that a stack position was requested and that stack position was NOT found in one of the stack buffers.
void	**stack_SET_MAXDEPTH** (uint32_t x)
	Allows the variable stackMaxDepth to be set ahead of time.

Macro Definition Documentation

#define MAX_STACK_SIZE   0x20000000

Maximum number of uint_32 size variables that this stack can hold.

This limit depends on whether or not LFS (see detailed description) is enabled when the application is compiled. If LFS is enabled you will want to change the value of

MAX_STACK_SIZE. The default value is 0x20000000 for 536,870,912 total spots.

0x20000000 * 4bytes = 2GB

Bibliography

1. Nicola, M. and J. John, *XML parsing: a threat to database performance*, in *Proceedings of the twelfth international conference on Information and knowledge management*2003, ACM: New Orleans, LA, USA.
2. Rector, A., et al., *OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns* in *Engineering Knowledge in the Age of the SemanticWeb*2004, Springer Berlin / Heidelberg. p. 63-81.
3. Serge Abiteboul, P.B., Dan Suciu, *Data on the web: from relations to semistructured data and XML*2000, New York: Morgan Kaufmann Publishers.
4. Bray, T., et al. *Extensible Markup Language (XML) 1.0 (Third Edition)*. 2004 [cited 2008 April 15]; W3C Recommendation]. Available from: http://www.w3.org/TR/2004/REC-xml-20040204/.
5. Corporation, M. *Microsoft Expands List of Formats Supported in Microsoft Office*. [Press Release] 2008 August 20th, 2010; Available from: http://www.microsoft.com/Presspass/press/2008/may08/05-21ExpandedFormatsPR.mspx.
6. Grigoris Antoniou and F.v. Hramelen, *A Semantic Web Primer*. 2nd ed2008, Cambridge, Massachusetts: The MIT Press. 264.
7. SC Haw, G.R. *A Comparative Study and Benchmarking on XML Parsers*. in *The 9th*

International Conference on Advanced Communication. 2007.

8. Athena, V., *XML Data Stores: Emerging Practices*, C. Barbara and M. Anna, Editors. 2005. p. 62-69.

9. Schmidt, A., et al., *XMark: a benchmark for XML data management*, in *Proceedings of the 28th international conference on Very Large Data Bases*2002, VLDB Endowment: Hong Kong, China. p. 974-985.

10. Yinfei Pan, Y.Z., Chiu, K. , *Hybrid Parallelism for XML SAX Parsing*, in *IEEE International Conference on Web Services, 2008. ICWS '08.* 2008. p. 505-512.

11. Am\, et al., *Projecting XML documents*, in *Proceedings of the 29th international conference on Very large data bases - Volume 29*2003, VLDB Endowment: Berlin, Germany. p. 213-224.

12. Lawrence, R., *The space efficiency of XML.* Information and Software Technology, 2004. **46**(11): p. 753-759.

13. Mathis, C., *Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems*, in *Department of Computer Science,* 2009, University of Kaiserslautern.

14. Johnny Stenback, A.H. *Document Object Model (DOM) Level 3 Load and Save Specification, Version 1.0*. 2004 [cited 2010 August 19th]; Available from: http://www.w3.org/TR/2004/REC-DOM-Level-3-LS-20040407/.

15. Berners-Lee, T., J. Hendler, and O. Lassila, *The Semantic Web.* Scientific American, 2001. **284**(5): p. 28-37.

16. Fensel, D., Hendler, J. Lieberman, H., Wahlster, W. (eds.) *Spinning the Semantic Web. - Bringing the World Wide Web to Its Full Potential.*2003, Cambridge: MIT Press.

17. Feigenbaum, L., et al., *The Semantic Web in Action.* Scientific American Magazine, 2007. **297**(6): p. 90-97.

18. Shadbolt, N., T. Berners-Lee, and W. Hall, *The Semantic Web Revisited.* IEEE INTELLIGENT SYSTEMS, 2006: p. 96-101.

19. Dan Brickley, R.V.G. *RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation*. 2004 [cited 2008 November 20th]; Available from: http://www.w3.org/TR/rdf-schema/.

20. Michael K. Smith, C.W., Deborah L. McGuinness. *OWL Web Ontology Language Guide: W3C Recommendation*. 2004 [cited 2009 February 20th]; Available from: http://www.w3.org/TR/owl-guide/.

21. Codd, E.F., *A Relational Model of Data for Large Shared Data Banks.* Communications of the ACM, 1970. **13**(6): p. 377-387.

22. Erik, W., *XML Technologies Dissected*, 2003. p. 74-78.

23. Arnaud Le Hors, et al. *Document Object Model (DOM) Level 3 Core Specification*. 2004 [cited 2011 Feb 2nd]; Available from: http://www.w3.org/TR/DOM-Level-3-Core/.

24. Jagadish, H., et al., *TAX: A Tree Algebra for XML*, in *Database Programming Languages*, G. Ghelli and G. Grahne, Editors. 2002, Springer Berlin / Heidelberg. p. 149-164.

25. Manukyan, M., *Element Algebra*, in *Advances in Databases and Information Systems*, J. Grundspenkis, et al., Editors. 2010, Springer Berlin / Heidelberg. p. 113-120.

26. Schmidt, A., et al., *Efficient Relational Storage and Retrieval of XML Documents*, in *The World Wide Web and Databases*, G. Goos, et al., Editors. 2001, Springer Berlin / Heidelberg. p. 137-150.

27. Bruno, N., N. Koudas, and D. Srivastava, *Holistic twig joins: optimal XML pattern matching*, in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*2002, ACM: Madison, Wisconsin. p. 310-321.

28. Qin, L., J.X. Yu, and B. Ding, *Twiglist: make twig pattern matching fast*, in *Proceedings of the 12th international conference on Database systems for advanced applications*2007, Springer-Verlag: Bangkok, Thailand. p. 850-862.

29. Tatarinov, I., et al., *Storing and querying ordered XML using a relational database system*, in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*2002, ACM: Madison, Wisconsin. p. 204-215.

30. O'Neil, P., et al., *ORDPATHs: insert-friendly XML node labels*, in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*2004, ACM: Paris, France. p. 903-908.

31. Bourret, R. *XML Database Products*. 2009 February 17, 2009 [cited 2009 January 2nd]; Available from: http://www.rpbourret.com/xml/XMLDatabaseProds.htm.

32. Lentz, A. *MySQL Storage Engine Architecture, Part 2: An In-Depth Look*. MySQL Storage Engine Architecture 2011 28 April 2004 [cited 2011 Jan 12th]; Available from: http://dev.mysql.com/tech-resources/articles/storage-engine/part_2.html.

33. *W3C Schools*. 2012 [cited 2012 6/1/2012]; Available from: http://www.w3schools.com/dom/books.xml.

34. James Clark, C.C., Fred Drake. *The Expat XML Parser*. 2012 [cited 2012; Available from: http://www.libexpat.org/.

35. Cook, C.R. and D.J. Kim, *Best sorting algorithm for nearly sorted lists.* Commun. ACM, 1980. **23**(11): p. 620-624.

36. Hetland, M.L., *Divide, Combine, and Conquer*

Python Algorithms: Mastering Basic Algorithms in the Python Language, 2010, Apress. p. 125-150.

37. Ramakrishna, K., *Caching strategies to improve disk system performance*, J.S. Love and G.W. Bradley, Editors. 1994. p. 38-46.

38. 2012 Wed Apr 13 14:57:11 [cited 2012 5/12/2012]; Programming Language popularity metric]. Available from: www.langpop.com.

39. *TIOBE Software: Tiobe Index*, 2012.

40. Hassenplug, S. *NXT Programming Software*. 2008 2/29/2008 [cited 2012 6/6/2012]; Available from: http://www.teamhassenplug.org/NXT/NXTSoftware.html.

41. A. Delman, A.I., L. Goetz, M. Kunin, Y. Langsam and T. Raphan, *Development of a system for teaching CS1 in C/C++ with*

Lego NXT robots. 2010.

42. Verle, M., *PIC Microcontrollers - Programming in C*2009: mikroElektronika. 336.

43. Foundation, A.S. *Apache HTTP Server Project*. 2012 [cited 2012 6/12/12]; Available from: http://httpd.apache.org/.