A Web Browser and Editor

by

Jason A. Wilson

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 28, 1996

Author_____
Department of Electrical Engineering and Computer Science
May 28, 1996

Certified by_____
James Miller
Thesis Supervisor

Accepted by_____
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

A Web Browser and Editor

by

Jason A. Wilson

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 1996, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science

Abstract

I have invented two novel data structures for capturing the structure and contents of a document written in Hypertext Markup Language (HTML). *Interval trees* and *nested interval trees*, the new data structures described in this thesis, permit any portion of an HTML document to be edited without requiring the rendering of the entire document. More importantly, however, interval tree representations of HTML documents are especially well-suited for almost *any* program that dynamically modifies the elements of an HTML document. The applicability of interval trees is not limited to HTML; interval trees provide a uniform representation of any arbitrary information source that can be represented as an SGML document.

Both interval trees and nested interval trees segregate document elements into two groups: semantic tags (HTML markup) and content (raw text). This forced dichotomy between data and meta-data permits efficient access to and modification of any portion of either domain (semantic or content). For example, Web browsers (which are dynamic graphical rendering engines for HTML documents) using interval tree representations may efficiently render subsets of an HTML document. Dynamically changing the contents of a small portion of the document (via a Java script, perhaps) does not force re-formatting the entire document. Graphical HTML document editors, which not only display HTML but also allow dynamic editing of the rendered content, can efficiently locally modify content, semantic tags, or both without invalidating global structures.

I have used interval trees in both an HTML rendering engine and an HTML browser/editor to greatly enhance the performance of both applications. Compared to the most widely used HTML browser currently available (Netscape Navigator™), interval trees are able to both (a) reduce the time required to redraw the document after a screen resize, and (b) dramatically minimize the memory footprint required to store the formatted document.

Thesis supervisor: James Miller
Title: Principle Research Scientist, World Wide Web Consortium

Acknowledgments

First I would like to thank the people who assisted directly with this thesis. My thesis supervisor, Jim Miller, assisted me not only in writing this thesis, but in finding funding when I didn't know how I would pay for my last year at MIT. Jim also introduced me to Dave Ragget, who wrote the original version of Arena (which I borrowed for the name of my editor), and provided the vision for creating a new browser based on a better internal design with a parse tree. Jeff Levine took time on one of his busy nights, to set me up an account on his PC so that I could write code and do measurements, and then logged out when I needed to do timings. Anil Gehi helped me create a title page. I will be forever grateful to Brian LaMacchia for assisting me with the final editing, formatting and printing of this thesis (and many other things not related to this thesis).

My academic advisor, Gerald Jay Sussman, advised me on many matters besides academics, and together with Hal Abelson, gathered together many smart and dynamic people that I worked with as an Undergraduate Researcher. If I tried to list all of these people and all of their help, I'd leave someone out and feel bad. You guys know who you are and I appreciate you all.

One person that I will never be able to thank enough, is Bill Rozas, who took me under his wing and provided inspiration, knowledge, and friendship. By watching and talking to Bill, I learned how to program better and how to use Emacs like a wizard. When Bill left MIT for HP Labs, he got me a summer job working with him on an exciting project, and then on days I didn't have a car, he gave me rides to work.

Special thanks to Anil Gehi, Peter Yao, Jeff Levine, Douglas Decouto, and all of my other friends who made MIT fun.

Most importantly, I would like to thank my family.

Table of Contents

List of Figures

1. Introduction

1.1. WYSIWYG Editing

Many users, especially novices, prefer editing documents when they that are presented on the screen in the same way that the reader will see it. The term WYSIWYG has long described the type of system where What You See (on the screen while you edit the document) is What You Get (when you print the document). In this form, the user sees not only the text of the document but also the implicit structure of the document as indicated by the formatting conventions of the text. Convention allows the reader to see words, sentences, paragraphs and headings in a familiar and understandable form. Computers, on the other hand, do not recognize this implicit structure without explicit programming and without some knowledge of structure they can't help maintain the proper formatting. When the computer knows that something is a paragraph, then it can reformat the lines inside the paragraph according to appropriate formatting conventions such as "no line should extend past the right-margin."

One of the most basic tools for manipulating documents is the "unstructured" text editor. These text editors understand some structure, most notably the lines of text in the document, yet they typically provide little interpretation and impose little restrictions on the text being edited. Thus, a text editor can manipulate an incorrect document just as easily as a correct document because they are both just stream of bytes. A powerful text editor will provide commands that interpret additional structure in the document and perform useful operations, but these commands are typically explicitly invoked by the user and usually understand local structure but not global structure.

The difference between a text editor and a structured editor is that a structured editor takes a more active role in interpreting what is being edited, how it can be manipulated, and how it should be displayed. A structured editor for HTML can manage the structural information (start tags, end tags, and other markup) insuring that the document is always in a legal state. A structured editor can then format the text of the document to indicate the structure of the document in a visual form. Now the user receives immediate and familiar visual feedback after making changes to the document

which makes it much easier to tell if the changes were the right ones. Without a structured editor, the user must know how the document structure is designated, the legal structural formulations, and how to find and fix the errors in the structural information. The user also needs to operate a separate tool for viewing the document in the "finished" form. This additional level of learning complexity prevents many inexperienced users from creating their own HTML documents and publishing them on the Web. For them, WYSIWYG editors are an enabling technology, not merely a convenience. With the popularity of the Web rapidly rising among the novice users and non-computer scientists, creating WYSIWYG HTML editors is an important task.

WYSIWYG editing has a very long history and many WYSIWYG systems, like the system I am using to write this thesis, are available. There are even some editors available specifically for WYSIWYG editing HTML documents like SoftQuad's HotMetal and GNNPress. Unfortunately, little information is available about the internal organization of most of these editing system because they are commercial products and this proprietary information could be used by competitors.

Much of the previous work on WYSIWYG editing isn't a good match for an HTML editor because they do not take advantage of special properties of HTML that make it so suitable for WYSIWYG editing. These systems try to keep the entire document formatted at all times. The price for this is that they have tended to use large amounts of memory to represent documents and slow down as the size of the document increases. For example, the Doc editor [Linton], which represents formatted documents with an object for every character, uses more than 8 bytes per character in the document and "Lilac is by its very nature a space hog." [Brooks pg. 158]

Memory consumption has not been an important consideration for many WYSIWYG systems because they are only meant for editing a few documents at a time. An HTML editor, on the other hand, can also serve as a Web browser if it can keep many documents simultaneously loaded and ready for viewing. This is useful while editing since there are various sources of information, like documentation, that are available as HTML documents, and many more will become available as the popularity of the Web increases. Obviously, memory consumption becomes more important under this usage

pattern. From my own personal experience using text editors, I've found that I typically have anywhere from 10 to 30 documents open at once. Only rarely do I actually modify all these files, but the ability of the text editor I use to view and copy information from documents that aren't changing into a document that I am editing is indispensable.

1.2. Multi-view Editing

A multi-view editor allows the same document to appear in more than one window. This is useful for editing one section of the document while referencing another. Supporting multi-view editing requires additional implementation complexity so that changes in one view are reflected in the other views.

Lilac, Janus [Chamberlin], and other "two-view" systems display both the unformatted source document and the fully formatted document. Other types of views may be desirable such as an outline mode that displays only the headings, or a "narrowed view" [Free Software Foundation] that displays only a portion of the document so that it is easy to keep changes from effecting other parts of the document. Even if only the basic WYSIWYG view is available, the question arises about what happens when one view has a different window size than the other since for HTML this means that the document will be formatted differently in each view.

One way to support multiple views is to have a single shared representation of the document from which each view is generated. When the shared representation is changed, all of the views are incrementally regenerated. Another approach is to use multiple representations and try to keep them in sync as changes occur. Since the first approach shares part of the representation, it has the potential to use much less memory to represent each view. In addition, the first approach allows new types of views to be generated and easily interfaced to the rest of the system. The second approach will not be this flexible unless a common representation for document changes is developed.

1.3. HTML versus SGML

SGML, standardized general markup language, is a standard for creating markup languages like HTML. While SGML is a ten year old international standard (ISO

8879:1986) based on years of experience with markup languages, HTML is younger and still rapidly evolving on multiple fronts. Even the most recent standard for HTML, "Hypertext Markup Language - 2.0" RFC 1866, does not include extensions that are in common use today, like tables. Many of these extensions are unnecessary or violate the spirit of SGML and therefore should probably not become part the of an official standard, but experience shows that *de facto* standards have been very forceful in the past.

SGML provides a way "to build exactly what is required to indicate the internal structure of any type of information in a common tool-independent manner." [Goldfarb] SGML documents are composed out of markup (start tags, end tags, entity references, markup declarations and processing instructions) and the document data (segments of text). The most important types of markup are the start tags and end tags because they describes both the meaning and the structure of the text segments they enclose. An SGML system must parse the document to distinguish the markup from the data in the document. The system can then represent the document internally in any format that is convenient for processing the document.

SGML allows the system to control both the syntax of markup and the allowable set of tags, but HTML provides a single concrete syntax and draws its tags from a fixed set. A SGML document type definition, or DTD, determines the contexts in which each start tag, end tag, and segment of text is allowed. A DTD may permit source documents to legally omit certain start tags and end tags when they can be inferred from context. This saves space in the document and makes it somewhat easier for humans to edit the documents by hand. Luckily, these omissions can be ignored by most of the system since the code that parses the SGML source document can infer tags and automatically insert them where appropriate. At the same time, the parser can insure that the tokens follow the grammar specified by the DTD. Most of this thesis assumes that a parser supplies a stream of tokens where every start tag is eventually terminated by an end tag and that all tags in the document follow the DTD precisely.

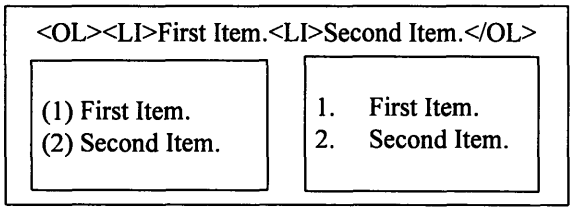

Figure 1-1: Two ways of formatting an HTML ordered list.

One advantage of SGML systems over traditional formatting systems is that the meaning of the document elements is independent of the way those elements are formatted. As Figure 1-1 shows, the formatter can make choices about how the list will be presented which can be used to intelligently format for a particular output medium or the preferences of the user. One reason this is important is that the Web has an active community of blind users who can access the Web with a screen readers and simple text based browser like Lynx, but these users couldn't view HTML documents if it could only be rendered to high-resolution bit-mapped displays and printers [Logue].

1.4. Handling Errors in HTML Documents

One problem that many traditional WYSIWYG systems have not had to contend with is the possibility of errors in the input document. Many existing HTML documents do not conform to the DTD specified by the RFC for HTML 2.0. Not only are there numerous extensions to the standard, but users and even tools often create ill-formed documents. When HTML was first created, most of the processing tools and browsers did not to check that documents followed a DTD or did not inform the user of the errors found in the document. The criteria that most users still use to determine if an HTML document is correct, is to see if it looks right with their favorite browser. Of course, an illegal document could easily appear correctly in one browser and not others. Luckily, this lack of strictness may slowly improve as the quality of the tools available to HTML authors improves.

The indeterminate state of HTML is a crisis for HTML editors because a misinterpretation of the document could cause irreparable damage when the document is overwritten. One possibility, is adding a small number of "error" tags so that many

illegal documents can be represented in a canonical form. For example, any unrecognized tag that is eventually terminated can easily be retained in the parse tree and marked as unknown.

<blink ...>Wow!</blink>

⇔ <unknown tagname="blink" ...>Wow!</unknown>

The user can then be informed that the nonstandard blink tag appears in this document and presented with the option of removing, renaming, or ignoring it.

If an unrecognized tag is not terminated, then it becomes much harder to deal with because it is then unclear when the end tag should be inferred. One possibility is that the start tag was misspelled and its matching end tag occurs later in the document.

<strung>Really? ⇒ Really?

Another common possibility is that like the BR or IMG tags, the unrecognized tag should be closed right away.

<app ...> My first Java Applet!

⇔ <unknown tagname="app" endtag=NO ...></unknown> May first Java Applet!

In this situation, enough information should be retained so that when the document is saved to disk, the unknown tag can be translates back into its previous form with the missing endtag. This way, as long as the context around the tag does not change, the document will still be valid to the tools that understand the unrecognized tag.

Allowing the user to see errors in the context of the formatted document is possible by giving the UNKNOWN tag a special formatted appearance. Rather than forcing the user to fix errors as the document loads and in order, clicking on the UNKNOWN tag in the text could cause a correction menu to appear.

1.5. Character Buffers

Character arrays, like files, are a simple and uniform way to represent information. When an unused region or a gap is added to the array, it becomes easier to add and delete new information from the array. GNU Emacs [Free Software Foundation] has been very successful representing text documents as character arrays with gaps in

them which indicates that this structure might be useful for a structured WYSIWYG editor as well. One of the advantages of character arrays is that many character processing primitives and library routines in the system already operate on arrays of characters.

The gap is like a small storage reservoir. When characters are inserted, the storage space is taken away from the gap, and when characters are deleted, storage space is returned to the gap. If the gap is too small or large, a new character array of a different size is allocated and the characters are copied into it. To be useful, the gap must be positioned at the point of insertion or deletion, but by moving characters around it is possible to "move" the gap between any two characters in the document.

To insert characters into a character buffer, first the gap is moved to the location of insertion. Second, as long as the gap is large enough, characters in the gap may be overwritten with new characters to insert. If the gap is not large enough, the character array must be reallocated with a larger size. Third, the boundaries of the gap are changed so that the newly inserted characters become part of the document.

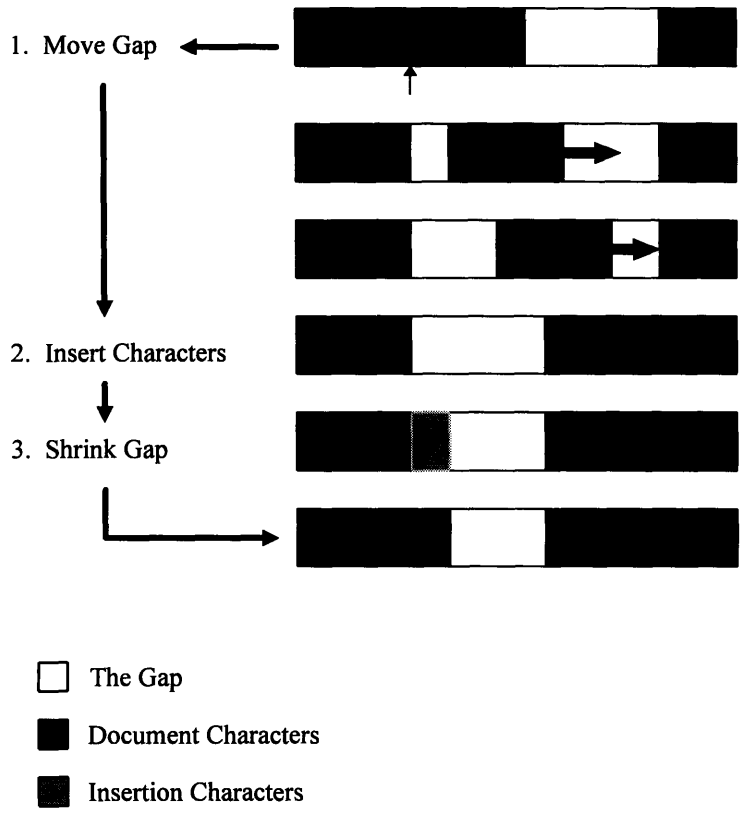

Figure 1-2: Inserting Characters into a Character Array with a Gap

Deleting characters from a character array with a gap in it is also quite easy. The gap is moved to the point of deletion and then grown so that some characters are excluded from the document and become part of the gap.

With small insertions and deletions, the efficiency of the character array depends primarily on how often and how far the gap must be moved (with a big enough gap, the character array does not have to be grown of shrunk very often). If the gap is moved to a random location on every insertion or deletion, then these operations will run in time proportional to the length of the document. Luckily, when users edit text, they tend to edit in local regions so the gap does not have to be moved far very often. When the gap must be moved to a distant point in the document, then the characters can be moved most quickly by moving multiple characters at a time. Such copy loops are typically implemented very efficiently in library routines, with a 32bit computer moving four characters and a 64bit computer moving eight characters at once. Thus, although the character array with a gap in it is not asymptotically as efficient as other data-structures, it

can still be used with documents that are over a megabyte long with only a short pause when the gap is moved long distances.

A character array with a gap in it leads to two natural indexing systems. The first system would index the characters in the document as though the gap did not exist. These index don't change when the gap is moved, but they can change when inserting or deleting characters. The second system indexes characters by their positions in the character array. These indexes change as the gap is moved around (or the character array changes size), but they don't change as characters are inserted or deleted. Before an index from the first system can be used to access the character in the buffer, it must be converted into an index from the second system by adding the length of the gap when the index is larger than the start of the gap.

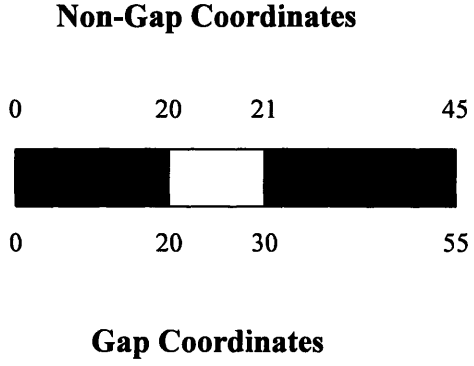

Figure 1-3: Gap and Non-gap indexing schemes

1.6. The Thesis

These are the points of this thesis:

1. Structured HTML editors should employ a uniform representation for internally representing documents that is not biased towards any single way of viewing or formatting the document.

2. Such a representation should take advantage of the fact that the raw data in an HTML document can be stored into a character array with a gap in it, since this is a uniform representation that has worked very well in unstructured editors.
3. The structural information is representable as a tree. With the right construction, this tree can be linked to the unstructured character array so that there is very little penalty for having separated the characters from the structural information.
4. A WYSIWYG HTML editor which employs this representation and only formats what is visible, is a viable alternative to an editor which keeps the document in fully formatted form.

The first point follows, in part, from the desire to support multiple-views. On the other hand, the most important function of a structured WYSIWYG editor is to allow the document to be viewed and edited in the usual formatted form. If this can not be attained with a uniform representation, then a special purpose representation is the only choice. The second point of this thesis will be addressed by the second chapter and further advantages of this representation will appear in the third chapter. The third point of this thesis is addressed by the fourth and fifth chapters. Finally, the fourth point of the thesis is an outcome of this document as a whole.

2. Character Arrays and Character Gap Buffers

2.1. Character Arrays are Convenient and Efficient

The character array is one of the most basic and important data representations: some type of the character array appears in almost every program and programming language. This thesis assumes that character arrays are mutable after construction, so other representation choices are necessary for purely functional programming languages.

A character array is similar to a file in that many different types of information are easily represented inside of character arrays. A character array is usually more efficient than a file since any character in the array can be quickly accessed without making library and operating system calls, or involving the disk except when the character array won't fit in memory. With memory-mapped files, the distinction between character arrays and files becomes blurry, since memory-mapped files can be thought of as persistent character arrays that are shareable between separate programs.

Good implementations of computer programming languages try to optimize the usage of character arrays. For example, C provides string processing library and some of these routines are implemented by hand in assembly language for maximum speed. Another technique of speeding up character arrays, is the use of compiler optimizations like induction variable elimination [Aho], that can speed up programs that march linearly through a character array accessing it with an index. This optimization attempts to use pointers instead of indexes thereby eliminating the step of computing an address.

Even the hardware itself provides mechanisms to allow efficient manipulation of character arrays. For example, the x86 architecture provides special string processing instructions that allowed significant performance improvements on early implementations. On modern micro-processors, such complex instructions have fallen in favor of more general purpose instructions that can be implemented in a single machine cycle, however, RISC processors like the PA-RISC processor from Hewlett Packard, still provide special instructions that help process character arrays at high-speed.

Character arrays with gaps in them retain most of the advantages of character arrays while adding the ability to incrementally add characters inside the character array. Character arrays with gaps are a little harder to interface with library routines since these library routines do not know about the gap. Sometimes, it is easy to break calls into two pieces so that the gap in the character array is excluded from the call. For example, when the part of the character array that the user would like to render is split in the middle by the gap, for some calls it is possible to process each half of the array ("before-gap" and "after-gap") independently. Another option is to move the gap before the call is made; this works well when trying to process small segments of the character array because then the gap does not have to be moved very far. If the number of characters moved is less than the size of the gap and the array isn't modified, then the gap can be restored its previous state with no extra work after the call.

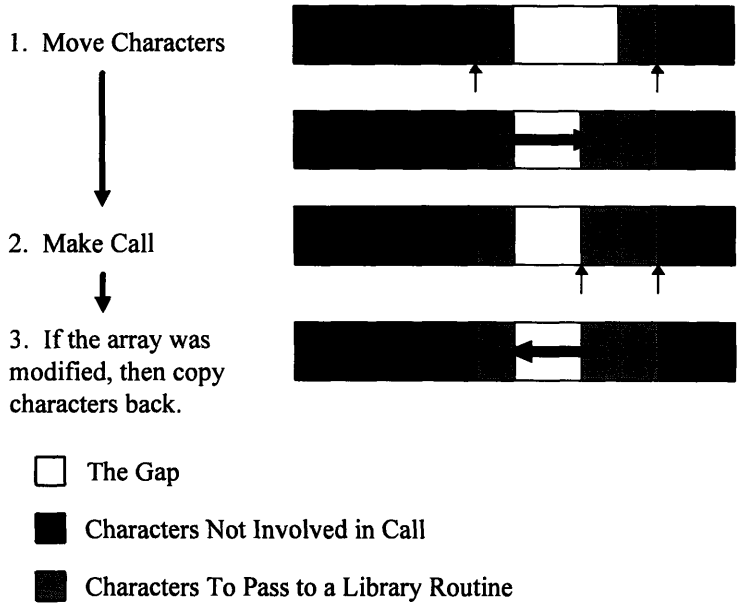

Figure 2-1: Calling Procedures that Expect Contiguous Regions

2.2. One Character Array With Gap vs. Many Character Arrays

Using a single character array with a gap in it has advantages over using many smaller character arrays. A single character saves space when programming with high-level languages where you can't explicitly control the layout of objects in memory as one can in low-level languages like C. For example, Java stores length and type information

as part of character arrays. When alignment costs are added, the storage overhead could rise to almost 3 words per character array. On a 64 bit machine, unless the character arrays are much bigger than (3 words)*(8 chars/word) or 32 characters, the storage costs would be dominated by the overhead costs.

A single character array also avoids problems like memory fragmentation as the number of characters grows and shrinks. One approach to inserting characters into a small character array is to copy the characters into a larger character array on every insertion. This could cause fragmentation problems with languages without copying garbage collectors. When fragmentation occurs, allocating and freeing memory takes longer, and the number of pages in the working set grows. To avoid memory fragmentation, extra space can be kept at the end of some of the small character arrays. Like the gap, this unused section will grow and shrink as characters are inserted and deleted. Unlike a single character array, the extra space is not shared across the entire document. In order to minimize wasted space, some storage reclamation policy is needed. Another disadvantage of this technique is that is presumes small character arrays. If one of these small character arrays becomes large, then excess time will be spent moving characters around because keeping the gap at the end of the character array does not take advantage of the locality of insertions and deletions.

When the document is broken into many character arrays, it's more difficult to make library calls because these calls can span multiple character arrays. Sometimes it is possible to achieve the same function by using multiple calls, but whereas the character array with gap requires at most one extra call with this technique, many separate character arrays means many separate calls. If the overhead of calling the library routine is high, then this can be a big disadvantage. An easy solution is to copy all of the character into a larger array, make the call, and then copy the characters back if necessary, but this takes extra time and space.

2.3. The Character Array for an HTML Document

All of the text of an HTML document can be kept in a single large character array with a gap in it. This corresponds to the source document with the tags and excess

whitespace removed. In addition, named and numeric character entities should be mapped into the characters that they represent.

The character storage area coupled with the document structure serves as the absolute repository of truth about the document. From these elements it is possible to generate the HTML source for a document that will format the same way as this one does (of course some characters are mapped into appropriate named or numeric entities as the source document is generated). However, this source document will not contain comments or unmeaningful whitespace from the original source document. The comments can easily be retained by storing them in the parse tree but it is an open question whether excess whitespace should be maintained. The excess whitespace is unimportant to an editor or browser, but it might be very valuable to the author of the document that edits the HTML source document with a traditional text editor. On the other hand, the output document is the smallest any unnecessary whitespace is removed. A third option is to throw out the original whitespace and to pretty-print the output document adding extra whitespace according to heuristics that produces readable output documents.

3. Formatting HTML Documents

3.1. Overview

From the perspective of the formatter, the tags in the document are instructions about how to transform the text and images into display structures. These display structures allow portions of the document to be render and allow geometric screen coordinates to be mapped back into objects from the document. When the document is modified, some of the display structures must be modified and the screen must be repainted. The modifications and painting must occur quickly enough to support fast typing rates. In addition, repainting must be done carefully or else the screen will flash.

3.2. Formatting Dependencies

Formatters can be characterized by the types dependencies they allows between different elements. Most HTML constructs can be formatted with only a single pass forward through the document because most elements only depend on the formatting of the elements that preceded it in the document. Tables are an important exception because to automatically choose column widths requires collecting information (minimum and maximum width) from every cell in the table. [Reference to Dave's paper on tables] Since this information does not change when the table is subsequently formatted, this additional dependency is easily handled with a single pre-pass over the elements of the table.

Most formatting systems, like TeX [Knuth], deal with cycles of dependencies. For example, to format the table of contents requires knowing the page numbers of sections that have not yet been formatted: a cyclic dependency. The method used to resolve these dependencies is successive approximation: the document is repeatedly reformatted until all of the dependencies are resolved. This can take an unknown number of passes in general. For example, after determining the page number of each section and reformatting the table of contents to include them, the table of contents could grow by a few pages invalidating the other page numbers and forcing parts of the document to be reformatted again. In practice, most dependencies can be resolved with only a few passes

through the document and each pass should be able to reuse work from the previous pass. The successive approximation scheme is similar to the iterative approach employed by many optimizing compilers in order to deal with the cyclic dependencies in the program graphs they are optimizing. Optimizing compilers guarantee termination by employing monotonic functions on finite lattices. Formatters could guarantee termination (on finite sized outputs) by requiring each pass to increase the size of the formatted output.

Each of the actions required to format an HTML document can be separately analyzed for its dependencies. When viewed this way, most of the work of formatting each block level element (the children of the BODY node) does not depend on previous blocks at all, so this work can be done in any convenient order. For example, paragraphs and other blocks can be independently filled to form lines of text because line filling only depends on the width of the screen but determining their absolute vertical offsets requires knowing the heights of all of the previous lines. Formatting from the first to the last block in a document make this very convenient because by the time formatting is started for a block, its vertical position is already known.

3.3. Relative Positioning

Instead of positioning with absolute offsets from the top of the document, blocks can be positioned relative to the screen. With this approach, it is only necessary to format and position enough blocks to fill the screen. Under normal circumstances, the number of blocks that fit on a screen and the size of the largest block are independent of the length of the document. Thus, while formatting the whole document takes time and space proportional to its length, formatting or reformatting just enough blocks to fill the screen will usually take about the same amount of time and space no matter how big or small the document is!

For a browser, this means that when the window is resized, the screen can be updated in a fraction of a second no matter where the user is in the document. In contrast, Netscape's response time for this operation depends critically on how big the document is and where you are in the document. I've measured thirty second delays before the screen is updated when resizing the window at the bottom of a one megabyte file.

For an editor, this means that the whole document does not have to be reformatted (or repositioned) when only a single block is changed, so editing and redisplaying HTML documents of arbitrary size should be possible without any unbearable slow-downs.

These benefits are only possible when the source document meets certain criteria and when the documents are represented internally in a form the allows the blocks to be randomly accessed. If the document is very large and yet composed out of relatively few paragraphs and other top-level blocks, then only formatting one or two of these blocks can still take time proportional to the length of the document. Luckily, the kinds of blocks that are most likely to become very large, like lists, tables and pre-formatted regions, can easily be formatted in sections. Tables require some preprocessing to collect width information but this information doesn't change when the screen is resized. This information can change as insertion and deletions occur, but it may be acceptable to ignore these changes until the user explicitly requests to reformat the table, as is done by other WYSIWYG systems like Microsoft Word. The remaining types of blocks like paragraphs and headers, would be confusing to read if they were much bigger than a single page.

If the unformatted internal representations of documents is smaller than the formatted representation, then space can be saved when editing or browsing large documents. The storage space for the formatting information of non-visible blocks can be recycled and regenerated when needed. In this case, total processing time might increase, but response time is kept uniformly short which is more important. As the user scrolls back and forth through the document, the same blocks are repeatedly reformatted but the user won't notice the extra time because it happens so quickly.

One problem with eliminating absolute positioning is that the scroll bar depends on them. For example, when the scroll bar is centered, this should indicates that height of the document before the current screen is nearly equal to the height of the document below the current screen. Without absolute positions, the scroll bar can not be positioned, unless a metric other than height is utilized. Two logical candidates are either the number of top-level blocks or else the number of characters. These won't have the same

properties as the height but they do give a rough and useful indication of the position in the document.

3.4. The Display List

The renderable structures which are produced by a formatter are commonly called a display list. As the last representation before displaying the document, the display list should provide the full formatting flexibility that the display device provides. The display list for a bit-mapped display should allow the arbitrary placement of multi-font text and images, even if the display list for a character based terminal doesn't support pictures and multiple fonts. As mentioned earlier, another function of the display list is to provide the ability to map screen coordinates to objects in the document. Related to this function is the need to provide the user with a visual feedback of selected text, so the display list should also be useful for highlighting or otherwise temporarily altering the appearance of some of its elements.

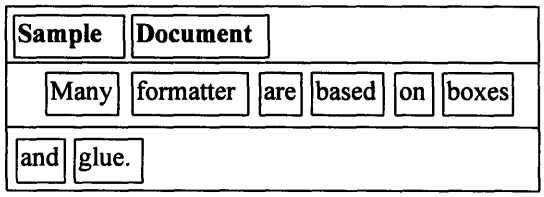

Figure 3-1: Boxes and Glue

The display lists of many formatting systems have been based on the nested box and glue concepts used in TeX. In some systems, characters are the innermost most boxes (Doc) while other systems used words as the inner most type of box (Lilac). In this example, the inner most boxes are words and the spaces between the boxes is filled in by invisible glue elements. With suitable abstraction, tables and images are simply other types of boxes, with tables serving as a container for other boxes.

The hierarchy produced by an HTML 2.0 formatter (no tables) can be vastly simplified. The top-level elements are always lines with nested elements inside of them but those elements do not have children. Display lines have a single baseline which the

Figure 3-2: Display Line Attributes

elements of that line use to control their vertical placement. Display lines also have a maximum ascent, maximum descent, and a maximum vertical line spacing which come directly from the constituent elements of the lines, like text and images. For text, the spacing properties come from the fonts rather than the individual characters. If each display element has an explicit horizontal placement, then glues isn't needed to connect the components of a line.

My first formatter only dealt with lines, text segments and images but the object oriented design allowed additional elements like bullets and radio buttons to be added later. Instead of boxing words or characters, the longest extents of text on the same line with the same style becomes a display item. Each display element has a draw method and methods for getting the ascent, descent, and width of that element. The object oriented design also suggested that each text segment should have a single style (font family, font size, font effects like underlining and bold, color, and so on). After setting the font and color, the text segments could be rendered with a single call to a drawString primitive in Java's graphics library and possibly another call to underline the text. Likewise, each image or other display element could be rendered by calling some Java primitives

The code for creating display lines would have been simpler if each word (or pre-formatted line) was treated as a separate display item since then every object can be treated uniformly as lines are filled. This approach has two problems. First of all, words can be composed up of characters with different styles and maintaining this information inside the display item is as complex as just creating multiple display items each with a single style. Secondly, representing each word separately requires more space and time than using larger chunks.

The space penalties depend on how much of the document is kept in formatted form. If only the visible portion of the document is translated into display items, then the space usage is not as important as in systems like Doc and Lilac, which keep the entire document formatted.

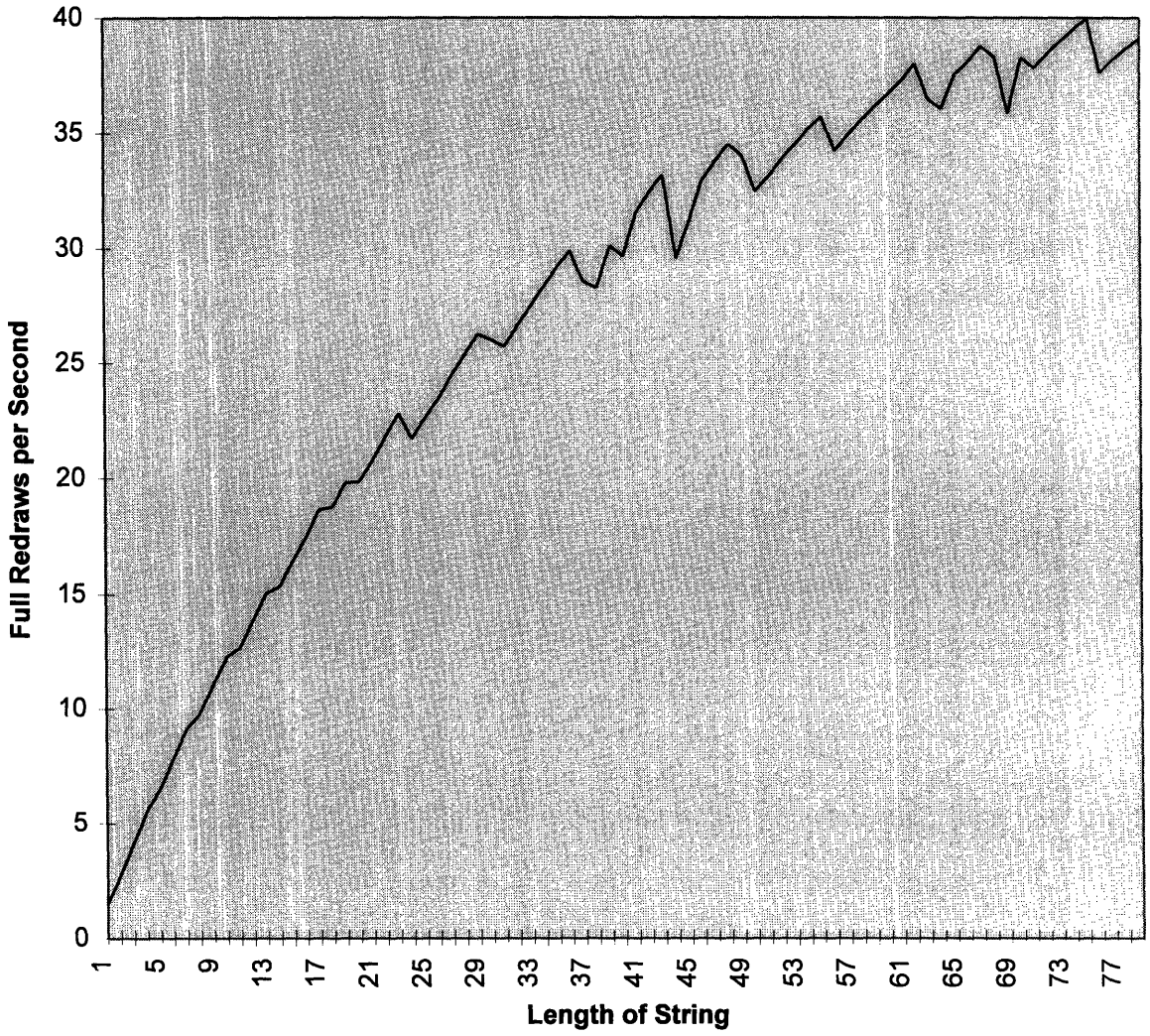

Figure 3-3: Using Larger Strings Increases the Redraw Rate

One of the time penalties arises from the desire to have each display object draw itself independently of the other display objects. This chart shows the maximum conceivable full redraw rate per second attainable by using strings of various lengths. This is interesting because the faster the screen can be painted, the more time there will be for formatting and the other work necessary for incremental changes. This chart shows that finding simple ways to reduce the number of calls made to the basic drawing primitives can have a very big effect on performance especially when the incremental

redisplay algorithm is dumb and repaints the entire screen often. I wrote a program using Sun's Java Developers Kit [JavaSoft] that times how long it takes to set the font, set the color and draw a homogenous string 1000 times to the same location. This is done for strings from one to eighty characters long. Then the maximum number of full redraws per second using a string of each length is calculated, assuming a screen size of 4800 characters (say 80x60). Naturally, this particular size is quite arbitrary since with fonts of various sizes, characters of various widths, user window size preferences, and partially full lines, it's hard to determine the number of characters on the screen. The chart shows that the screen can only be repainted about 6.5 times per second with strings that are 5 characters long. The actual full redraw rate could easily be lower since the timing loop does not include any costs of accessing display elements or calculating screen positions. On the other hand, using strings that are 20 characters long, the screen can be redraw at a rate of 19.8 times per second which is fast enough to keep up with most touch typists.

As stated earlier, display elements are objects that draw themselves and therefore it does not seem appropriate to look inside of them to form larger chunks of text when drawing since this violates the spirit of object-oriented design. Creating an extra level of indirection between the draw methods and the calls to the drawString primitive could be done to recover larger units in a clean way, but this seems more complex than forming larger chunks when the information is more readily available. I also feared that trying to recover large stretches of text might slow everything down. This meant that chunks of text in the same font, not words, were the most logical unit of text to represent in the display items.

3.5. Transforming HTML 2.0 into Display Lines

Once a representation is selected for display lines, the code which creates them from HTML is remarkably simple. Display lines may be generated either incrementally as the document is received from the network (or the file-system), or afterwards by walking over a parse tree. This section will describe the approach I used in my initial formatter which created display lines for the whole document, although much of this description could also apply to a formatter that only formats portions of the document.

There are two kinds of state in the formatter. The first type of state is global to the formatter and is seen by each element as they are walked in order. Examples of this type of state are the current line being filled and absolute screen positions. The second type of state, which I call the style state, is more interesting, because changes only occur when a start tag is encountered. More importantly, when an end tag is encountered, all of these changes are undone. To put it another way, the first type of state change allows an element to see the effects of younger siblings and their children. The other type of state change only depends on the unclosed tags, (the chain of parents from the current block to the root of the parse tree). Thus, all children receive the same style state from their parent.

Style states incorporate information about margins (which allow various kinds of lists to be indented), the font family, font size, font effects, the drawing color, and whether text should be automatically filled. The style state is a useful object that can be embedded inside of multiple display elements.

When a start tag is encountered, any line breaking implied by the tag is done and then the current style state is altered as necessary. When an end tag is encountered, any additional implied line breaking is done and then the style state is reverted to the state in effect when the matching start tag was encountered. This is most easily arranged by storing the states on a stack, since trying to directly undo changes doesn't work very well. For example, when an EM start tag is encountered, the style change turns on the italic effect, but when an EM close tag is encountered, blindly turning off the italic effect is inappropriate since the italic effect may also be required because of an earlier unclosed tag.

A few start tags, like IMG and LI generate display items but most of the display items come from the regions of text between tags. The most complicated code in the formatter was the code for filling text to form lines while making the fewest number of text segments possible. This was simplified by delaying the creation of text display items until the final contents of the line are known. The unfinished line is kept in a buffer that includes not only the character (actually the index of the character), but also the width and the style in effect when the character was added to the buffer. When a character that

is added to the buffer finally overflows the maximum width of the line, or when a tag requests a line break, an appropriate portion of the buffer is "retired," leaving the remaining portion to become part of the next display line. The new display line is then added to the end of display list and is painted in any windows where it could be seen.

To make mapping screen coordinates to objects fast even when there are display lines for an entire document, the display list is represented as an array of display lines sorted by vertical offset from the top of the document. This means that a binary search can quickly find the display line that spans a vertical position. A linear search through the elements of the display line then finds the display item which intersects with a horizontal position. This second search could be speeded up by storing the contents of the lines in an array and using another binary search, but because display lines usually contain just a few items, a linear search works well most of the time. This representation also makes it easy to find the screen lines to paint when part of the window is exposed or when the user scrolls the document window since the first and last line can be found with binary searches.

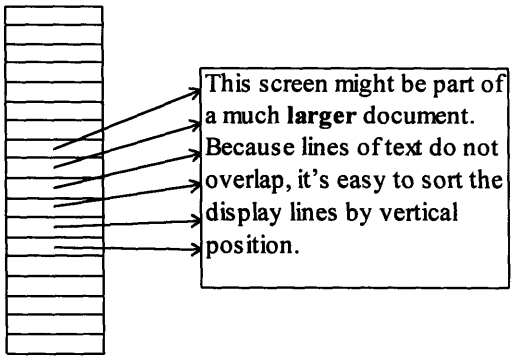

Figure 3-4: Sorting Display Lines

Obviously the approach just described is only sufficient for a simple Web browser. As stressed before, formatting the whole document should be avoided because this saves time and space. More importantly, since display lines do not work well for formatting more interesting structures like tables, the concept of the display line should be generalized before trying to make a better browser. Because tables require two passes through its elements instead of only one, a simple stream of tokens and text is less convenient than a parse tree because it's easy to revisit nodes in a parse tree. The parse tree is also a convenient place to store data collected on the first pass through the table's elements although care must be taken to support multiple views correctly if this is done.

3.6. *Incremental Redisplay*

One of the most fundamental requirements of a WYSIWYG editor is to reformat and redisplay the document after every change so that the user has immediate feedback about how the changes will effect the appearance of the document. I am going to concentrate on the most basic changes, the insertion and deletion of a single character because these are the types of changes that happen most frequently. While a user might tolerate slight delays for large scale changes that occur infrequently, it is very important to give immediate feedback as each character is typed even when the user is typing many characters per second.

What surprises me is that the simplest approach to redisplay almost works: insert the character into the HTML source document, throw out all of the display structures, and then completely reformat and repaint the section of the document that appears on the screen. Netscape appears to format (but not display) documents at the rate of about ten "screens" per second. I measured this by opening a large HTML file (without any images) stored locally on a hard-drive, and timing how long it took for the red gauge in Netscape's status area to become full. Then I counted the number of times that I could press the page down key before reaching the end of the document. Based on the maximum full redraw speed data that I presented earlier, as long as the drawn strings are long enough, at least this same number of screens should be paintable per second. Thus, as a very rough approximation (no images), a typing rate of about five characters per second can be supported with the simple approach provided a suitably fast implementation language is used. This isn't fast enough for touch typists but it should be clear that eliminating a little work would make this a practical approach (especially as computers become faster).

A serious problem with the simple approach is that repainting the entire screen will normally cause the screen to flash annoyingly. A flash occurs when the contents of the screen are erased and then the screen is redrawn. I found that a little flashing is acceptable, but a flash from every key press is annoying. Most systems minimize flashing by not repainting unchanged regions of the screen and by moving around regions of the window when possible since moving does not require erasing the destination

region. The same effect can be achieved by drawing the pixels to an off-screen region and using bit-blit operations to overwrite the contents of the window without first erasing it. A more complicated approach could time the updating of the screen with the motion of the raster beam but this probably isn't appropriate for an editor designed to run under the current windowing operating systems.

How does the time spent trying to reuse portions of the screen compares with the time of drawing to an off-screen region and moving this region unto the screen? Both approaches perform moves except that the screen reuse approach uses many smaller moves instead of one large move and is therefore probably spends more time moving. How does the cost of figuring out what regions can be reused compare with the cost of redrawing every region? Timings suggest that text can be quickly painted when long extents are used, however, images can easily take much longer to draw because of the cost of decoding the image, computing colors and scaling the image. One possibility is painting on-screen images to an off-screen region that is never overwritten. Then the images could always be copied from this region to the other off-screen region where the text is being prepared. This would avoid the costs of decoding the image, but it would add the cost of an extra move and would consume more memory.

While it is easy to initially create the display lines, it is much harder to incrementally modify them in response to small editing changes. This is one reason why the simple approach is so appealing. Display items and fractions of the text display items must be shuffled between display lines while trying to minimize the number of text segments in a line. This requires code that is similar to the code for generating display lines, but different enough that much of it can not be shared. In addition, modifying the display lines directly, means there is no way of knowing what the screen looked like before the change unless that information is retained somewhere. This information is crucial for minimizing the flashing of the screen unless painting is done to an off-screen region as described earlier.

Thus it is very convenient for an incremental redisplay algorithm to generate new display items and throw the old ones away when they are not needed anymore. This implies that display items should be light-weight objects. Here again is another

advantage of the character array. Instead of storing strings inside of display items, the start and end indexes into a single character storage area are stored. This allows lines to be formed without moving characters around or creating new strings. These indexes are gap indexes and therefore must be updated as the gap is moved, but with only a single screen's worth of display lines, this should not take very long.

3.7. Dynamic Formatting and Incremental Updates

To take advantage of light-weight display lines, a single character storage area, and relative positioning, I wrote an new formatter and display engine. The new "formatter" does only a portion of the formatting. As before, it maintains the style states, but now it simply processes the text (removing extra whitespace and mapping entities into characters) that it receives and concatenates it to an output character buffer while associating an appropriate style state with each character. The next chapter will describe data structures which can efficiently implement this association. The new formatter doesn't create any display structures and doesn't fill text for a particular width screen but it does embed newlines into the character buffer between the block level elements since each of these blocks starts on a separate line. Newlines in pre-formatted regions are faithfully copied to the output buffer, but otherwise, newlines from the text are converted into a space character.

The new display engine creates display lines on the fly using the newlines embedded inside the character array to figure out where the lines start. Once it creates enough display lines to fill the screen, it paints them and stops formatting. Each segment of text between the newlines becomes one or more display lines formatted according to the width of the view and the margins in the style state of the first character on each line. The width of each character is determined by calling a width method provided by the style state object. The style state also provides methods for getting ascent, descent, and spacing, as well as for drawing segments of characters.

The display lines created by the display engine are very simple structures because they don't have any children. Instead, they maintain start and end indexes that delimit the

region of characters in the character array that should be painted on this line. They also maintain the width of the screen and the width of the line. Comparing these two number determines if inserting a character into the line will expand it past its maximum allowable width. A line also has a y-offset from the top of the screen, a height, and an ascent. To paint the line or map coordinates to characters, the positions of characters is re-determined, but this can be done very quickly as long as the line isn't hundreds of characters long. Since display lines are so simple, they are cheap to compare.

The dynamic nature of the new display engine made it straight-forward to add incremental update ability. When a key is pressed, the character is first inserted into the character storage area. At this point, when the document is saved, the new character will appear in the output. Since the important state of the document is not kept in the display structures, bugs in the redisplay algorithm are not likely to destroy segments of the document. Next, an attempt is made to insert the character where the cursor is by moving the right hand portion of the line further to the right opening up enough space for the new character, clearing the are underneath the new character, and then drawing the character. If this was done, then the screen line is modified to reflect the change.

In the common case, there will be no more modifications to the screen, and there has been minimal flashing, but it is still necessary to check that things are formatted correctly. Inserting a character can cause words to wrap down to the next line, possibly causing a cascade of changes until an embedded newline is finally reached. Additionally, inserting a space character at the beginning of a line can cause the previous line to change since the first word of the line might now be small enough to fit on the previous line. To verify the formatting, every line starting at the previous line if it exists is reformatted. I currently always reformat the previous line, but if that line ends with a newline or if the inserted character is a non-space character, then this isn't necessary. Formatting stops when the bottom of the screen is reached or when the new display structures start matching up exactly with the old display structures past the point of change. At this point, there is before and after information about every line that has changed, so it is possible to be clever and use move operations in order to make bring the screen up to date with a minimum flashing. I found that clearing and then painting the updated lines

worked and no one complained about excessive flashing. Drawing to an off-screen area and moving onto the screen is probably easier than trying to reuse existing portions of the line.

Even with the order of magnitude slow-down imposed by running the display engine on top of the byte-code interpreter of the Java runtime system, this relatively dumb redisplay algorithm can keep up with my fastest typing rate. The common case has been optimized so that the character is inserted, a few graphics operations are done, and then two lines are reformatted to verify that no other changes are necessary. When this verification fails, it's still usually the case that only a few lines are reformatted and redrawn.

3.8. A Better Dynamic Approach

The new display engine did not deal with tables and images. I imagine handling images by simply treating them as characters in funny fonts. All of the appropriate abstraction is present in the style state object to make this possible. On the other hand, I have already mentioned that since images are more complex to draw than text, trying to reuse images on the screen is probably a very good idea.

Tables are not handled because my model is broken: it is too line centric and works bottom up instead of top-down. Searching for newlines worked because without tables, lines are never nested. Each cell of a table however, can have it's own lines and nested lines.

A better way to do dynamic formatting is to descend down the parse tree like the earlier formatter, and reformat the "dirty" elements. The dirty elements do not have to be completely reformatted since if they contain filled lines of text, the line oriented approach from can reduce the amount of formatting. After reformatting the dirty block, a top-down re-formatter could then reposition the other blocks without actually formatting them. My line oriented implementation sometimes reformatted lines from blocks that haven't changed because a change in position is considered to be a change in the formatting of a line.

A top-down formatter would still find it convenient to use light-weight display lines with the text stored in a single area. Whereas the bottom-up formatter looks for newlines in the document, the top-down formatter can keep information about the extents of text in the parse tree. As the next chapter will show, how this information is represented will have an effect on how quickly the document can be updated after each change.

4. Internal Representations of HTML Documents

4.1. High-Level Structure

The basic requirements for an internal representation of HTML documents are different for browsers and editors. Both browsers and editors need a representation that is compact, can be constructed quickly, and can be accessed by the formatter and display engine before it is fully constructed if portions of the document are to be displayed before the whole document is received. However, a browser only needs to display the document and this leads to some fundamental differences. While an editor requires a representation that can be transformed back into an ordinary SGML source document, a browser's representation only has to include the block and style information needed for formatting the document. A more serious distinction is that an editor mandates fast incremental updates to the internal representation, while a browser only needs to construct a static representation quickly. The point is that any representation that works well for an editor may be more general than a representation suitable for clients like browsers which only wish to examine a source document, not modify it. This excess generality may impose some space and time overheads but a representation suitable for an editor is probably adequate for a browser as well.

An obvious representation for SGML documents is an incrementally constructed parse tree. Each pair of start and end tags becomes a single "tag node" in the parse tree (connected to a single parent and many child nodes) and each segment of text becomes a leaf node in the tree (connected to a parent tag node).

In order to choose the best representation for an HTML document, it is reasonable

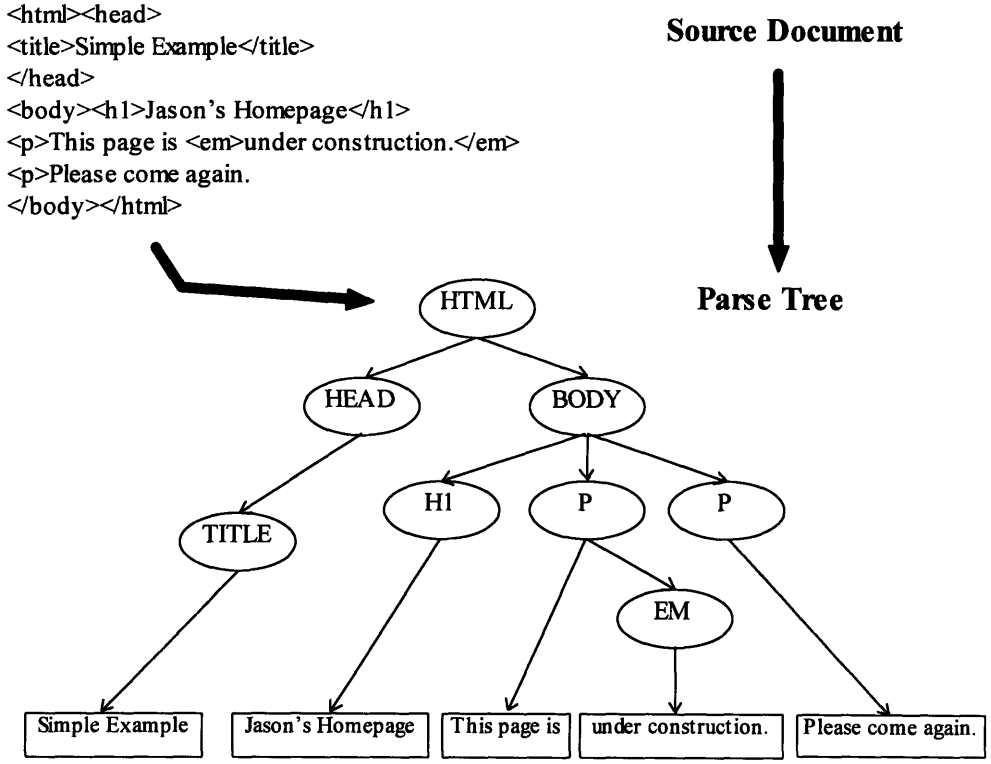

Figure 4-1: High-Level Structure of an HTML Parse Tree

to make some assumption. The nesting of tags in an HTML document is normally quite shallow even though a ridiculous nesting can cause the height of the parse tree to grow linearly with the size of the document. The reference line-mode browser from The World Wide Web Consortium only provides for a maximum depth of 20 nodes and apparently there has been no need to increase this limitation. [Nielsen] On the other hand, as documents become larger, the branching factor can become quite high. For example, a document with 1000 paragraphs and 200 headers could have a branch out factor of over 1200 at the BODY node in the tree! For this reason, it is desirable to store the children as an array, balanced binary tree, or any other structure which supports fast random access.

Retaining the parse tree information as contained in the HTML source document means that it's easy to convert the document back into HTML source but it also does not preclude augmenting the parse tree with additional links in order to capture interesting connections between parse nodes. Dave Ragget suggested that in the case of tables, it might be convenient to link each cell not only to the row (which is the cell's parent in the parse tree) but also to a column structure. There may also be advantages for other HTML tags, such as anchors and forms, if the parse tree is augmented, or additional structures are created that relate to the nodes in the parse tree, but first there is a need to concentrate on the more basic problem of representing the parse tree and text of a document.

4.2. Using a Single Character Array

The high-level description just presented does not use the single character array. One way to accomplish this, is to store start and end indexes in the text nodes. In fact, it is only necessary to store start indexes in the text nodes since the end index is always the starting index of the next node (except for the last node which can be special cased).

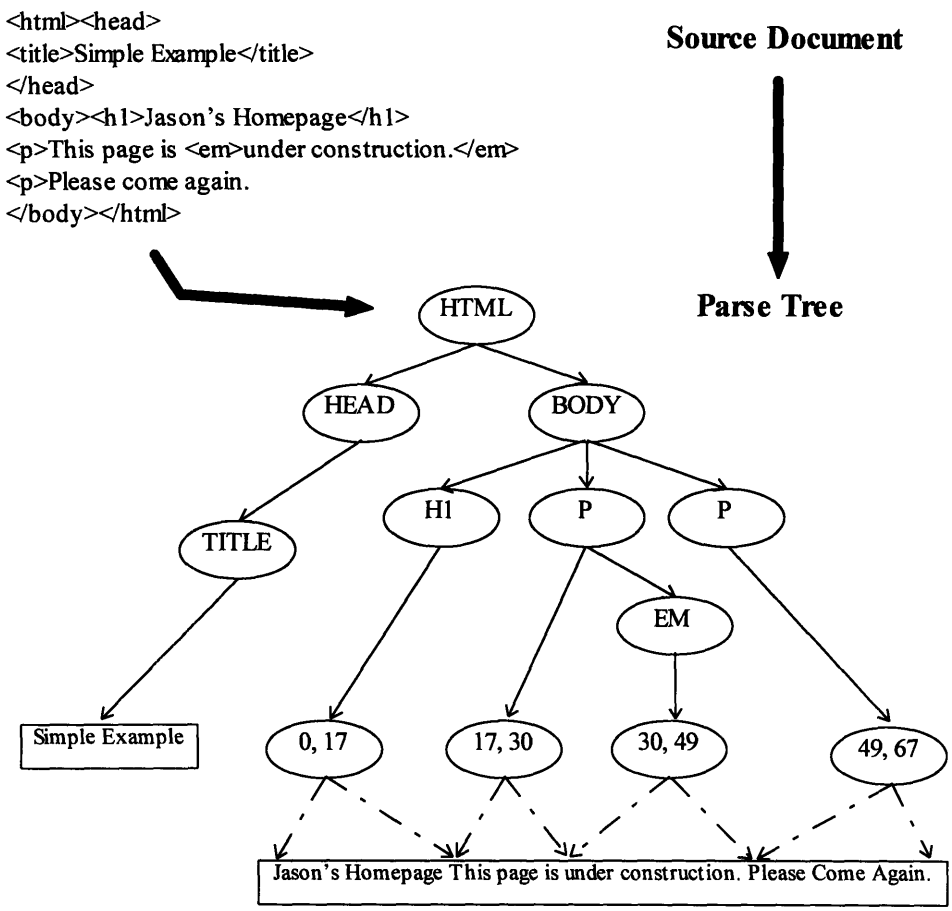

Figure 4-2: Linking the Character Array to the Parse Tree with Start and End Indexes

4.3. Supporting Incremental Updates

Unfortunately, while representing an SGML document as a large character array and a parse tree with start indexes is an appropriate static representation, it doesn't meet an editor's dynamic need for fast incremental updates. When a character is inserted or deleted, this could potentially change all of the start and end indexes stored in the text nodes. Naturally, this is only practical for documents of a certain length. While the current trend on the Web is towards small HTML files because of bandwidth limitations between some servers and clients, it may sometimes be more convenient for the author or

the reader to have a single large document. For example, documentation or books intended to be delivered on a CD-ROM could be stored as large HTML files.

Before considering more elaborate internal representation, I'd like to present a simple representation that isn't really practical because it doesn't meet our storage requirements, but leads to better representations. Again, there is a parse tree and a single large character array, but the start indexes are removed from the parse tree and then a large array of pointers is that links each character (through its index) to a node in the parse tree is added. Now, the parse tree captures the hierarchical arrangement of start and end tags in the document without any reference to the text of the document, and the character array and linkage array describe the relationship of the text to the structure of the document. Naturally, the character array and linkage array can both be kept with a gap in them. Of course while the parse tree and character array are reasonably space efficient, the linkage array consumes one word of memory per character in the document.

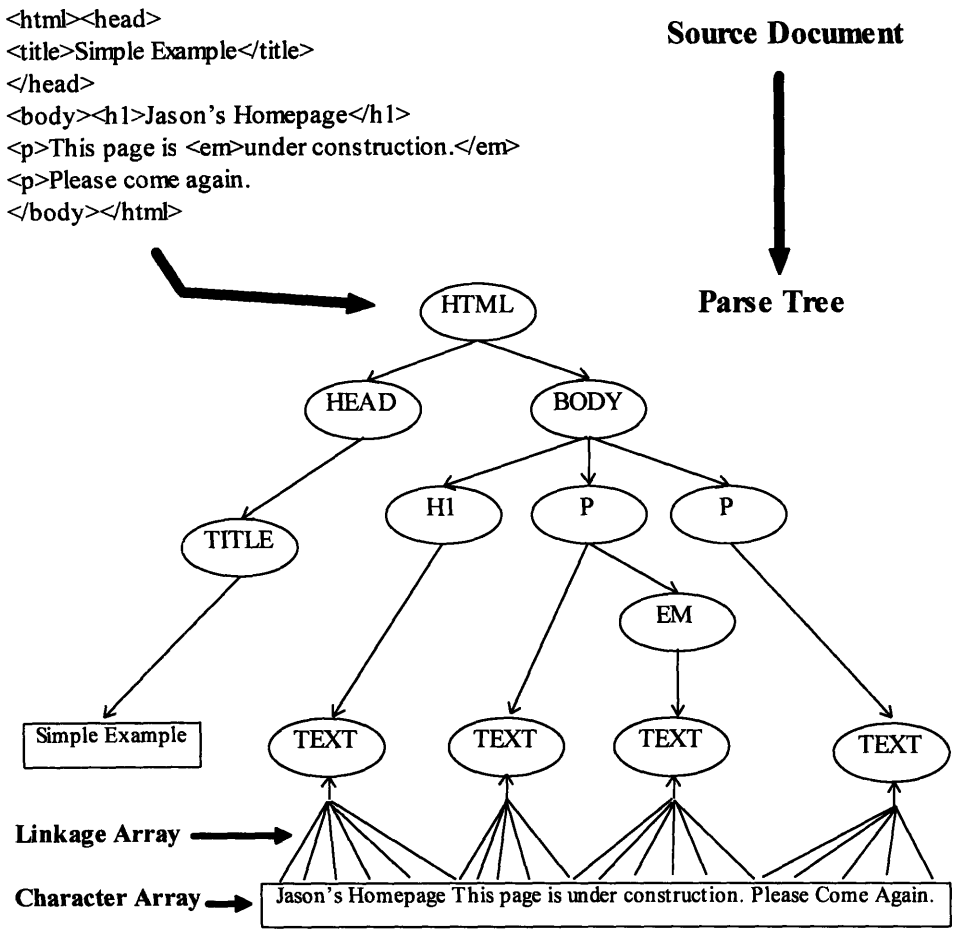

Figure 4-3: Linking the Character Array to the Parse Tree Using A Linkage Array

This representation has another problem besides a large storage cost: a character's index is quickly converted to a node in the parse tree but in the general case it takes a linear search through all of the text to find the region of characters that are children of a node. On the other hand, from a single index which lies within the region, two binary searches will find the starting and ending indexes of that region. These binary searches are not necessarily as fast as other binary searches. The problem of course is that for each trial index in the search, it is necessary to walk from the linkage node all the way up to the root of the parse tree to see if the linkage node is a child of the node of interest. I mentioned before that it is usually appropriate to assume the height of the parse tree is bounded by a constant, but even so, the constant cost is more than just a few machine instructions. The cost of the binary searches can be made O(lg n), and more importantly, the requirement of already having a starting index can be eliminated by monotonically

(but not necessarily consecutively) numbering each node in the parse tree according to a pre-order traversal. Then fast comparisons between linkage nodes and arbitrary parse tree nodes are possible. The reason to number the nodes non-consecutively is to improve the efficiency of inserting nodes once the parse tree is fully constructed since when the nodes are numbered consecutively, they always have to be renumbered in order to maintain the monotonicity requirement, whereas leaving space between indexes allows them to be renumber less often.

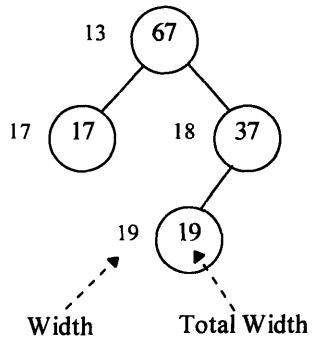

Figure 4-4: An Interval Tree

While the parse tree and character array by themselves are reasonably space efficient, the linkage array consumes about one word of memory per character of text in the document because it neglects the fact that consecutive characters tend to link to the same node. When this is true, the high storage costs of the linkage array can be avoided by representing it using an interval tree, a data-structure I first encountered while examining the text property feature of GNU Emacs 19. (A different kind of interval tree is described in [Cormen].) An interval tree is a balanced binary tree augmented with width information. Each node in the tree stores the sum of its width and the widths of all of the intervals below that node in the tree. To find the width of an interval, subtract the total-width of the left and right children from the total-width of the interval, a constant time operation. The empty tree has a total width of zero.

Starting at the root of the tree, the interval which spans a legal index can be inexpensively found. If the index is less than the total-width of the left child, then recurse down the left tree with the same index. If the index is greater than the total-width of the node minus the total-width of the right child, then the interval spanning the index must lie down the right sub-tree. Before recursing down the right tree, subtract off the total-width of a node minus the total-width of the left node from the search index since that many characters come before the right child. If neither of these conditions is true, then the index lies somewhere within the current interval. As an interval is searched for, its

starting position can also be computed or this can be computed later by walking back towards the root of the tree. When progressing linearly through the interval tree, it is simple to remember the ending index of the previous interval and use this as the starting index for the current interval.

An interval tree can replace the linkage array if the length of each interval correspond to the length of each uninterrupted extent of characters with the same connection to the parse tree. Where previously an array lookup mapped indexes into the character array to a node from the parse tree in (constant time), now a traversal down towards the leaves of an tree (lg (n)) does this mapping. This is much more expensive, but when processing each character in order, this need only be done only once for each interval.

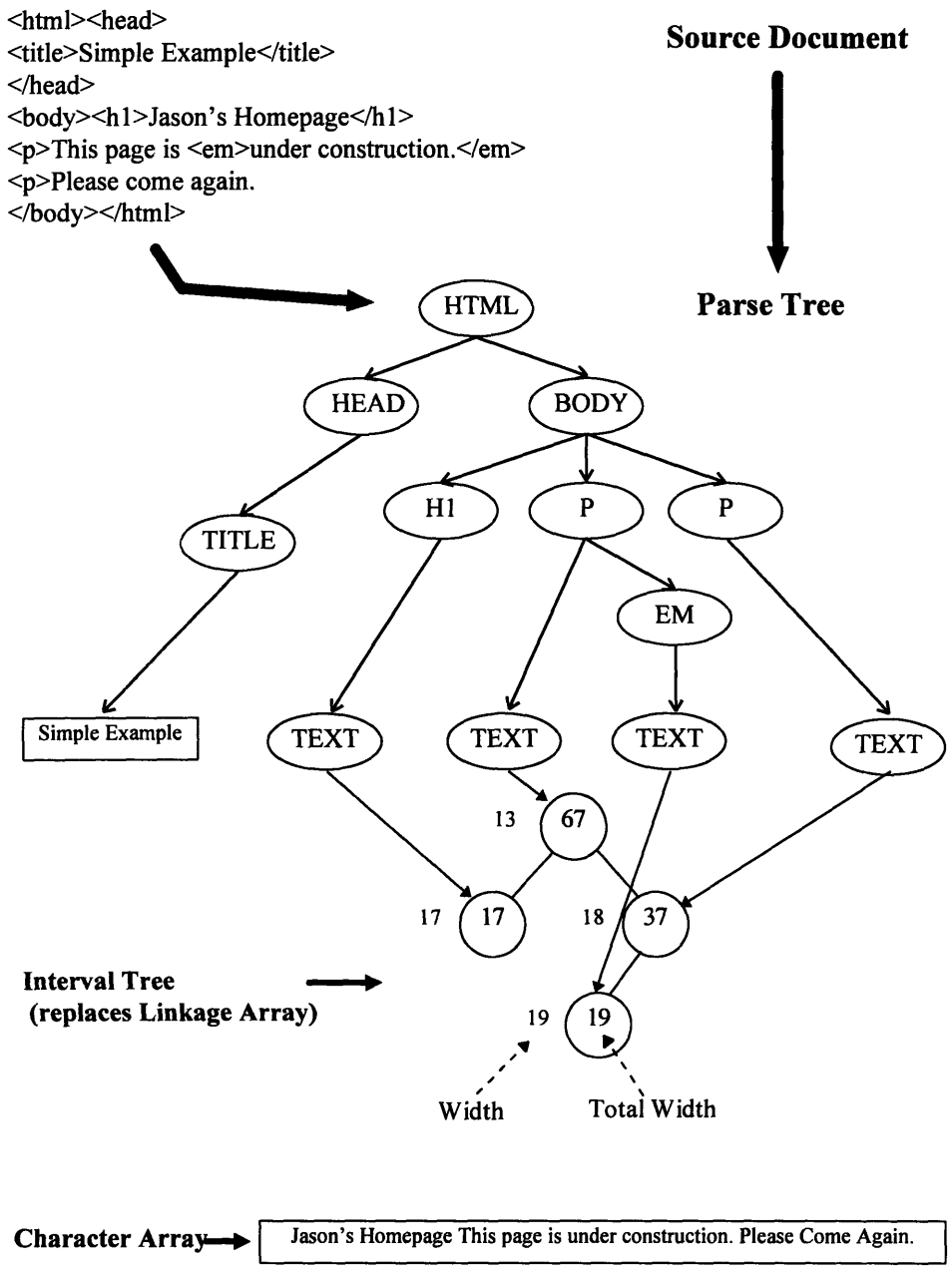

Figure 4-5: Linking the Character Array to the Parse Tree using an Interval Tree

The reason that an interval tree supports incremental updates better than a parse tree with start indexes is that inserting a character only alters the total-widths of a small number of intervals. When the width of an interval is changed, the interval tree is updated with a single upwards pass towards the root of the tree adjusting the total width of each successive parent node by the same delta. Since the tree is balanced and the

height of the tree is O(lg n), this type of change can be done quickly even on large documents. Likewise, adding or removing an interval requires updating all of the parent widths as well as re-balancing the tree, but this can also be done in O(lg n) time even on large documents.

Besides saving space over the linkage array in the usual case, interval trees also provide the important benefit of being able to store intervals in other data-structures, for example, an HTML parse tree. (Some implementations of tree balancing swaps the identity of nodes when a node is deleted [Cormen]. This should be avoided when nodes from the tree are stored into other data structure or when a node has many auxiliary fields.) Having an interval means being able to find its starting or ending index in O(lg n) time, and thus interval trees provides a means of quickly finding the indexes spanned by a parse node without having to number parse tree nodes. Given a node, traverse down to its leftmost text node child and then traverse up the interval tree to find its starting index. Finding the ending index is similar but then rightmost text node is used instead. Assuming that the height of the parse tree is O(lg n), the asymptotic running time is O(lg n) which scales well with the size of source documents.

The HTML editor that I described earlier, written in Java, used a parse tree and interval tree as described above except that I didn't store intervals in the parse tree because I thought I wouldn't need to go from parse nodes to a range of indexes very often. However, the "bottom-up" approach to formatting is flawed and having inexpensive access to start and end indexes from within the parse tree is very advantageous. Still, I learned that interval trees were practical for real-time editing of documents even when they were implemented in Java which imposes an order of magnitude performance hit due to the interpreted implementation of the Java virtual machine. Insertion and deletion of characters into a document can be done at moderate typing speeds doing a redisplay after every change. The fact that interval trees are also used successfully in both Emacs 19 [Free Software Foundation] and Edwin [MIT Scheme Team] suggests that they are a suitably fast representation.

4.4. Combining the Parse Tree and Interval Trees

An interesting possibility is to combine the interval tree and the parse tree. Dave Ragget suggested storing start and end indexes in the parse tree in gap coordinates which change as the gap is moved around but not as insertions and deletions occur. If this is done, the incremental updates would not require much work, but moving the gap around could take much longer since many nodes must be updated. To make this representation practical, the children of each parse node should be kept in an array (or a tree) so that a binary search at each level of the parse tree can be used to find the parse node spanning an index.

A better way of combining the parse tree and interval tree is to use nested interval trees. The children of each parse node are a separate interval tree. Each text and tag node is an element in an interval tree and each tag node is itself subdivided into separate intervals spanned by an interval tree. As I mentioned before, it is desirable to store the children of a node in an array or a tree anyway because these data-structures support random access. Not only does an interval tree provide random access, it gives the ability to quickly find a node spanning an index (the costs are harder to compute when there is more than one interval tree) or find the starting index of a node.

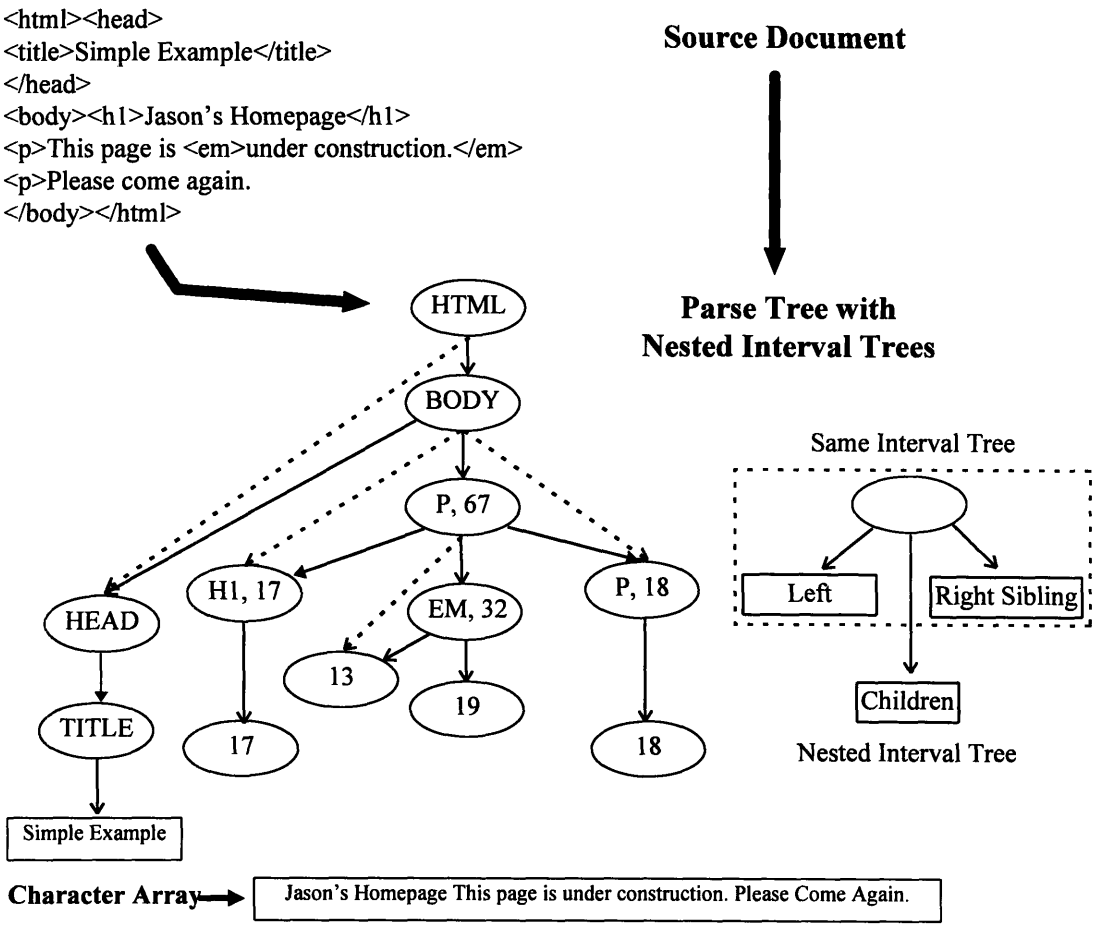

Figure 4-6: Linking the Character Array to the Parse Tree Using a Nested Interval Tree

The hidden advantage of the nested interval representation, is that it can be constructed very quickly because width information doesn't have to be propagated all the way up the tree for every interval. The cost of constructing an n node balanced binary tree is O(n*lg n) where the lg n term represents the balancing costs that can potentially happen at every insertion. This upper-bound overstates the cost however, since not every insertion requires re-balancing. However, when constructing an n node interval tree, the cost is again O(n*lg n) but now the lg(n) term represents the cost of balancing as well as the cost of updating all the parent total-width fields above the newly inserted node. This time, the full lg(n) costs are incurred on every insertion. During incremental editing, this lg(n) cost of inserting a node is acceptable since it isn't likely to add that many nodes at

once but when the initial internal data-structure is built, an unbounded number of nodes needs to be added as quickly as possible.

The problem stems back to the requirement that the data-structure be usable while the document loads. An interval tree can be constructed more quickly by storing the width of each interval in its total-width field but not propagating this information to the root of the tree. After the whole tree is built, a O(n) post-order walk can be used to combine all of the widths into the appropriate total-widths. Of course, constructing the tree this way means that the interval tree can't be accessed until after it is entirely built.

With nested interval trees, there is an easy solution to this problem: simply delay propagating width information until an entire interval tree is finished. For a text interval (which is always a leaf), only correct the interval tree up to it's parent tag node. Since each nested interval tree contains less nodes than an interval tree for all of the intervals in the document, there are fewer nodes to update during this walk. For a tag node interval, delay propagating its width information until all of its children have been added, i.e., when the node's end tag is encountered. When the tag node is finished, only one level of the nested interval trees is updated.

To access the nested interval trees incrementally, it is necessary to somehow find trees that aren't fully constructed. If unfinished nodes are attached to the tree in the rightmost position of the tree, where they will end up anyway, then they are easy to find, but because an unfinished node will not have propagated its total-width upward, it will make its parent node look shorter than it really is, maybe even making it look negative! It might be easier to store the unfinished nodes on a stack and only attach the node and propagate its width information when the node is finally finished. Then, each unfinished node on the stack contains a valid interval tree which might have some fully finished nodes inside of it.

If the partially constructed BODY node is taken off the stack, then it is possible to access all of the fully finished block level elements and start displaying them on the screen, and if the additional level of complexity is merited, the unfinished block one level up on the stack from the body node can be partially formatted and displayed in the same manner before the tree is fully constructed.

4.5. Summary

I have presented some techniques for representing HTML documents that use a parse tree to represent the structure of the document and a single contiguous region of characters to store the text of the document. A single region was chosen because this region can be represented with a character array with a gap in it which allows convenient usage of the primitives and procedures in the system that operate on character arrays. For static uses, the linking was easily achieved by storing start indexes in the parse tree nodes and using an array to store the children of a node. For dynamic uses, the linkage can be achieved using either interval trees or nested interval trees. For dynamic uses, the character array can be kept with a gap in which allows characters to be easily inserted and removed from the character array.

By abstracting out the methods that select children and obtain the starting index, code that examines the static representation can also work with the dynamic representation although with slightly different performance characteristics. On the other hand, code that modifies the structure of a document should use the dynamic representation since the static representation is expensive to update (unless it is only adding information to the end of the document).

5. Time and Space Measurements

5.1. Measuring Time and Space

The recommended representations developed in the previous chapter have good asymptotic running times for basic operations provided that the height of the parse tree is O(lg n) where n is the number of characters in the file. In practice, the height of the parse tree is bounded by a constant so this is not a problem. However, asymptotic running times only give a rough indication of the run-time performance of these data structures because they don't include the constants that determine if these data structures are really practical. For this reason, I will measure and discuss some of the various time costs to determine if these representations are practical for real-world systems.

The asymptotic space usage of all these representations is O(n) where n is the number of characters in the source file. This assumes that the size of each image is bounded by a constant. Of course, the asymptotic space usage is almost completely useless for real world evaluation since almost any internal representation will be O(n). Again, it is the constants that are important. For these representations, the space efficiency depends primarily on the length of the segments of text between tags with longer segments meaning smaller space consumption. When the ratio of text characters to markup characters in the source document is high, these representations consume modest amounts of memory.

I will not include images in the analysis of the space consumption for many reasons. Images can easily make a tiny document into a storage monster because when it comes to storage costs, pictures really are worth a thousand words. Any implementation that expects to perform well on documents of arbitrary sizes, will have to deal with the storage problems of images in a special way. For example, images could be kept on disk but their widths and heights could be kept in memory so that the screen can be formatted and the text portions drawn. Then image can be painted in the right place as they load from disk. Of course some of the images, like the ones visible on screen, should be cached in main memory. Another reason to exclude images is that none of my prototypes

explicitly manages the data for images or implements a policy for retaining them in memory since this is done automatically by the standard Java runtime system.

While collecting data about my own representations is easy, collecting information from other systems is more difficult because I can't modify or analyze the source code of commercial products. Instead, I examine the behavior of these programs as they run and make assumptions about how the behavior translates into time or space usage. This will introduce many additional sources of error and bias, but hopefully the trends will be accurate. To measure timing information from other programs, I found operations that take multiple seconds to complete so I could measure them by hand with a stop-watch. Since most programs perform well on small documents, the only way to expose the time costs are to increase the size of the documents. Of course, increasing the size of the input documents too much eventually causes the physical memory limits of the test machine to be reached and the program slows down from the effects of paging to disk. Because the various programs have different baseline memory footprints and consume memory differently, each program reaches this limitation at a different time. This thesis will not address the performance of programs under paging conditions so I stop measuring programs once they reach the file size where there is excessive paging.

The space costs are even harder than the time costs to measure and requires more assumptions. The tool I use for measuring space is a process viewer, pview, that runs under Windows NT and shows how much heap space has been allocated by a running application. I measured the amount of heap space the application has allocated before and after a big document is loaded and assign the difference as the space used to represent the document. However, if the application has over-allocated memory before loading the test document, then this could under-estimate the amount of storage consumed to represent the test document. In a

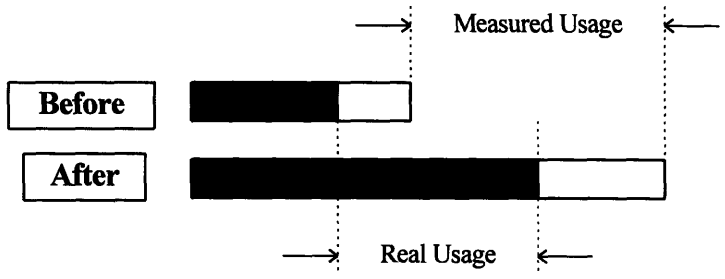

Figure 5-1: The Measured Space Usage May Differ from the Real Space Usage

similar way, if the application over-allocates memory as it loads the test document, then my simple measurement could over-estimate the memory usage.

There really isn't any question that the application will over-allocate memory since allocating memory in chunks is a common way to speed up the storage allocator in the system. The important question is how does the application allocate memory. If the application allocates chunks of memory in small fixed size chunks, then making the test document long enough should eventually overcome the error. However, if the application scales the amount of memory that it allocates, then these results may never be accurate because as the size of the document is scaled, the amount of over-allocated memory could scale along with it. One indication that this measurement method is failing would be a space usage that oscillates or makes big jumps as the size of the document is increased.

5.2. The Test Conditions

The test files for gathering data were generated by duplicating the body of an HTML document containing the Constitution of the United States. The size of the smallest file (a single copy) is 28,563 and the size of the largest file (15 copies) is 427,830. All files are stored on a local hard drive so that network transmission costs can be avoided. There are no images in any of the files and they also contain very little markup besides the headings and paragraph designators, but there are some one line paragraphs and other constructs that increase the amount of markup. Still, these documents are very different from many of the short, image rich, highly ornamental HTML documents in use on the Web.

The timing measurements were conducted using the Linux operating system on a 90 MHz, 16MB computer because this was an easy platform for compiling and modifying the LineMode browser [Nielsen]. The space measurements were run under Windows NT 3.51 using a 90-MHz PC with 32MB of memory.

5.3. Initial Translation

A practical internal representation should be constructible in about the same time as it takes to construct a fully formatted representation of the document. In addition, the

space usage should be similar or better than a fully formatted representation. I wrote code in C for constructing nested intervals trees and embedded this code inside of www, the reference line mode browser developed by the World Wide Web consortium [Nielsen]. I then disabled the code in the line mode browser that formats the document.

The reason I wrote this code in C was to facilitate comparisons with commercial systems that are written in C. The Java code in the editor for parsing HTML files and constructing parse trees and interval trees is terribly slow. Part of this is caused the order of magnitude slow-down caused by running on top of the Java byte-code interpreter. The lexical analyzer and parser are also very inefficient. The lexical analyzer is an interpreter for a mini-language and is not as efficient as a lexer produced by a tool like *lex*. The parser is not very clever and does multiple hash table lookups for each start tag in the document. Finally, the code that translates the text of the document into the character array and interval tree is character oriented and does not take advantage of the fact that the interval tree is constructed incrementally. Since the Java code uses a single interval tree instead of nested interval trees, every change to an interval requires modifying the chain of the parent intervals reaching to the root of the tree. While a single change is accomplished very quickly, the cumulative cost when processing thousands of characters individually becomes very noticeable.

Figure 5-2 shows the time in seconds taken to process the locally stored input file which consisted of between one and 15 copies of the U.S. Constitution. The average, minimum, and maximum times are graphed based on five time trials. The times for the nested interval trees were obtained by timing (with the Linux time command) the modified version of the WWW line-mode browser that constructs nested interval trees. The times for Netscape Navigator 1.2 were obtained on the same machine by manually timing with a stop-watch how long it takes for Netscape's blue-gauge in the status area to fill up and disappear after opening the file or requesting a reload of that file. I presume that this corresponds to Netscape's time to load and process the file into a fully formatted internal representation but this time also includes additional work such as displaying the first screen, updating user interface elements like the gauge, and so on. At around 15

copies of the constitution, or 427830 bytes, the machine (16 megabytes of physical memory) started to page noticeably with Netscape.

Figure 5-2 suggests that the nested interval tree representation can be constructed in a fraction of the time that it takes to build a fully formatted representation. In addition, the time taken to construct nested interval trees scales linearly over this range of input files. This is probably because the input documents are short and bushy. As the size of the file increases, the only interval tree in the document that grows is the tree for the children of the BODY node. This interval tree is only updated once for each top-level block in the document and this happens infrequently and takes little time compared to the other work that is done for each character.

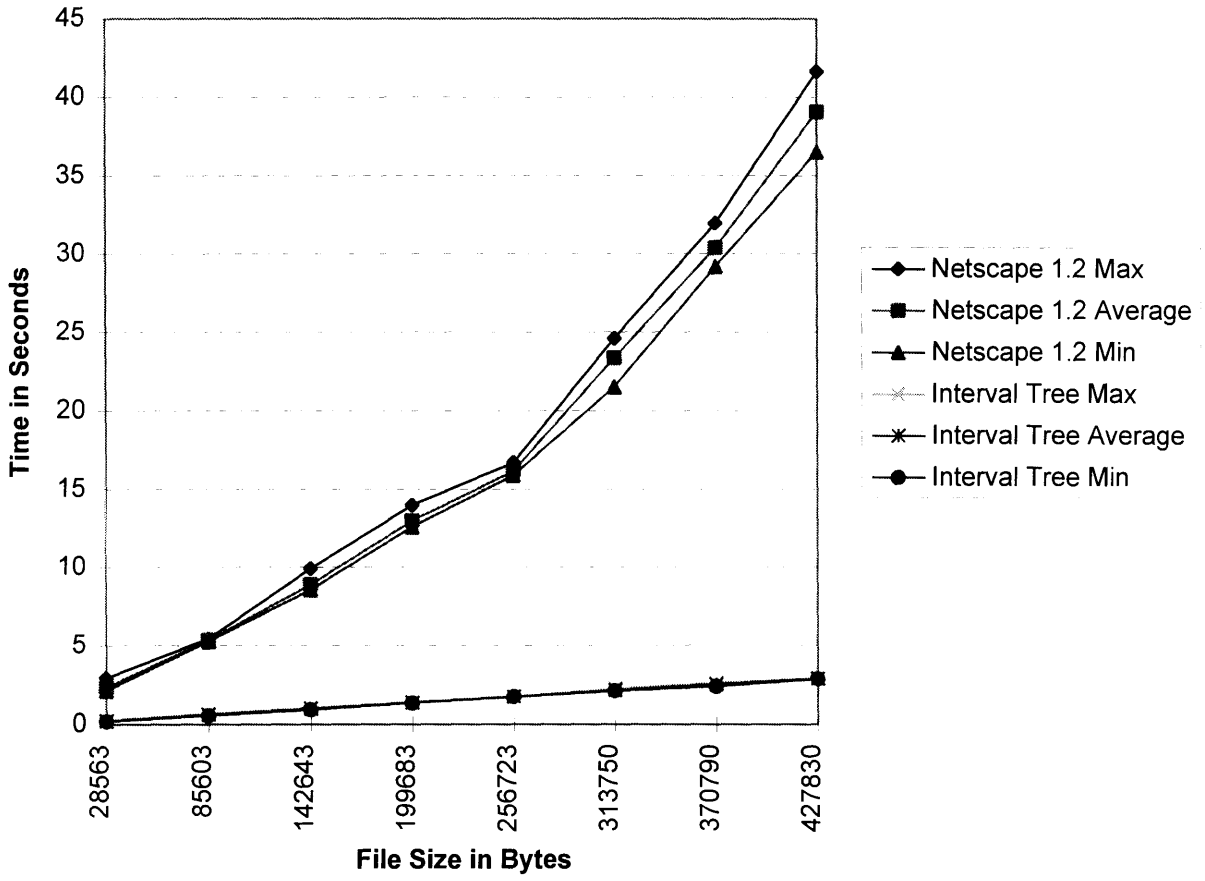

Figure 5-2: Construction Time vs. File Size in Bytes for Netscape and Nested Interval Trees

5.4. Memory Usage

Figure 5-3 shows the space usage for two commercial browsers, two commercial HTML editors, and the space usage for the nested interval trees representation. The space usage for these programs was measured as described above.

The memory usage of the nested interval trees was calculated by adding the length of the source document to the size of the nested interval tree. The size of the source document over-estimates the **minimum** size of the character array since it includes tags,

comments, and excess whitespace, but when the character array contains a large gap for efficiency, these should be roughly equal. The nested interval tree has 1027 nodes for a single copy of the constitution and each node is 36 bytes. These 36 bytes represent 10 fields: up, left, right, children, count, total-length, attributes, tag-number, and the color (for balancing). Since there are only a small number of tags and only one bit is needed for the color, these two fields share a 32 bit word. The space taken by the attributes is not measured but the source document has very few tags with attributes. This graph suggests that there are great space advantages when the document is not kept in a fully formatted form.

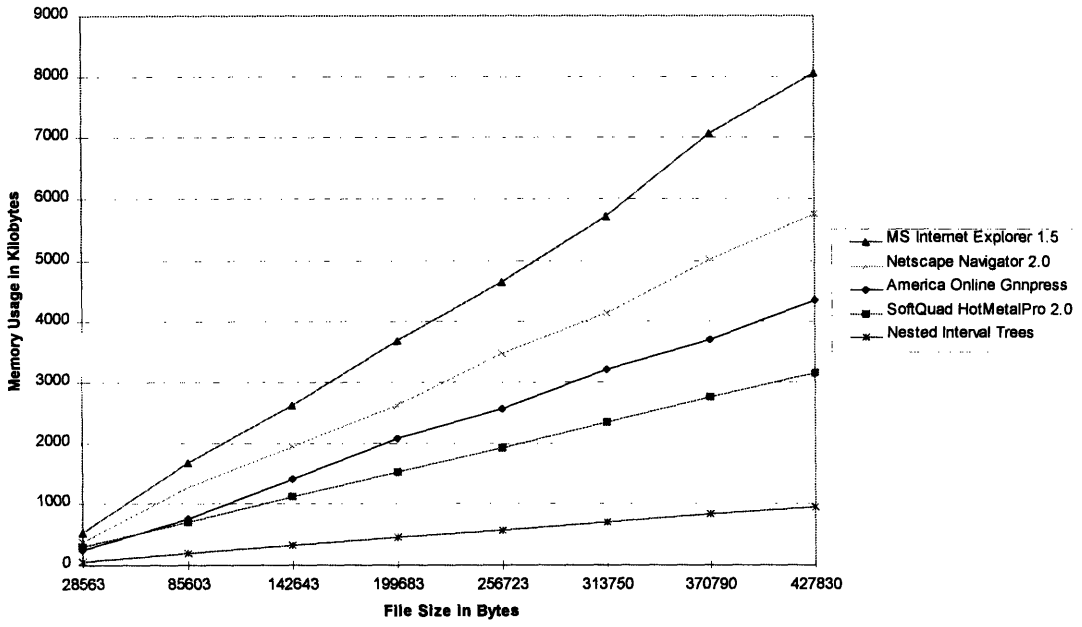

Figure 5-3: Memory Usage vs. File Size in Bytes

5.5. Dynamic Formatting

The dynamic formatter in the editor I wrote used a single interval tree instead of the nested interval trees, so it is not obvious that the performance of dynamic formatter will be as good with nested interval trees because the depth of the tree changes. For hundreds of random access a character at a time, this is probably noticeable, but when

accessing the characters linearly as a formatter does, the cost per character should be comparable.

The editor was written in Java which imposes an order of magnitude slow down factor and yet it still performed well. It is very doubtful that a formatter using nested interval trees written in C could not compete with a formatter written in Java using a single interval tree. After all, the number of nested interval trees is equal to the depth of the parse tree and this never exceeds twenty in practice. Even if each of these nested interval trees was the same size as the single interval tree for the whole document (they should be much smaller), mapping indexes to parse nodes will take a maximum of twenty times longer than with a single interval tree.

6. Conclusions

6.1. Nested Interval Trees and Character Arrays with Gaps

This thesis has developed nested interval trees and demonstrated that when combined with a character array containing a gap, a compact representation for HTML documents that can be constructed quickly is attained. Since elements of a nested interval tree can be accessed randomly and HTML documents have special properties, documents can be formatted in sections rather than being formatted from top to bottom. If only the section on the screen is formatted, then the screen can be resized very quickly for documents of any size and reformatting the document after a change is much easier.

One disadvantage of this representation is that while character arrays with gaps in them are very simple to implement and are very useful, they eventually become too cumbersome as the size of the document increases. Documents that are more than a few megabytes long incur noticeable delays when the gap is moved from one end of the document to the other.

One possibility for future work is to extend the linkage ideas (for example, by using two or more total-width fields) so that more than one region of characters can be used. It may be possible to directly link to a document stored on disk (but this document will contain tags and excess whitespace) and overlay changes in a separate area.

6.2. Dynamic Formatting

The major weakness of this thesis is that the dynamic formatter presented was bottom-up instead of top-down and thus it can only supports lines of filled text and not more interesting constructions like tables. Images can be added, but without additional optimizations, it is not clear how the system would perform if this is done.

This thesis also relies on a very big assumption: that the whole document does not have to be formatted. While this is (currently) true of HTML documents, arbitrary SGML documents might include page numbers and other elements that introduce new formatting dependencies requiring that the whole document be formatted. This would

nullify the space savings of this representation and raise into question whether the document should be kept in both a formatted and an unformatted form.

7. Bibliography

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6).

Kenneth P. Brooks. *A Two-view Document Editor with User-definable Document Structure.* Technical Report SRC 33. Digital Equipment Corp., Systems Research Center. November 1988.

Donald D. Chamberlin *et al. Janus: An Interactive Document Formatter Based on Declarative Tags.* Research Report RJ3366, IBM Research Laboratory, San Jose, California, 1982.

T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* Cambridge: The MIT Press, 1989. ISBN 0-262-03141-8.

Free Software Foundation. *GNU Emacs 19.30 source code.* Available from URL: `ftp://prep.ai.mit.edu/pub/gnu/emacs-19.30.tar.gz`.

Charles F. Goldfarb. *The SGML Handbook.* Oxford: Clarendon Press, 1990.

JavaSoft, a Sun Microsystems Business. *The Java™ Developers Kit (JDK).* Available from URL: `http://www.javasoft.com/java.sun.com/products/JDK/index.html`.

Donald E. Knuth. *The TeXbook.* Addison-Wesley, Reading, Massachusetts, 1994.

T. Berners-Lee and D. Connolly. *Hypertext Markup Language 2.0.* Technical Report RFC 1866. IETF Network Working Group, November 1995. Available from URL: `ftp://ds.internic.net/rfc/rfc1866.txt`.

Mark Linton and Paul Calder. "The Object-Oriented Implementation of a Document Editor". *Proceedings of OOPSLA 1992.* Available from URL: `http://karl.cs.flinders.edu.au/publications/Calder_OOPSLA-92/html/draft.html`

Bob Logue, A Word Is Worth a Thousand Pictures. *Byte*, **21**(4) p. 236.

MIT Scheme Team. *Edwin source code.* Available from URL: `ftp://www-swiss.ai.mit.edu/pub/scheme-7.4/`.

Henrik Frystyk Nielsen *et al. W3C Line Mode Browser source code*, Available from URL: `http://www.w3.org/pub/WWW/LineMode`.

Dave Ragget, *The HTML3 Table Model*, W3C Working Draft, January 1996, Available from URL: `http://www.w3.org/pub/WWW/TR`.

The United States Constitution. Available from URL:
`http://lcweb2.loc.gov:8080/const.html`