Compact Pat Trees

by

David Clark

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1996

0-612-21335-8

Abstract

Given a text string $S = s_1 s_2 s_3 ... s_n$, we want to preprocess S such that given a pattern $P = p_1 p_2 p_3 ... p_m$, we can find $\{i | s_i .. s_{i+m-1} = P\}$ as efficiently as possible. Suffix trees are a data structure solution to this problem. Unfortunately, when n is large, the storage required by a suffix tree can be prohibitive. This thesis presents several related new representations for a close relative of the suffix tree, the PAT tree, that retain the functionality of suffix trees while requiring a fraction of the storage used by current methods.

Acknowledgements

There are many people without whose support this thesis would not exist, however two stand out. First is my wife Liane. In addition to providing love and support, she read drafts of this thesis and provided many useful comments. I also thank Professor Ian Munro, my supervisor, for working with me on this project and supplying many of the ideas that in some form or another made it into this thesis.

Darrell Raymond, Tim Snider, Elizabeth Blake, Professor Frank Tompa and Professor Gaston Gonnet, members of the Electronic Oxford English Dictionary project past and present, provided a link to the practical side of text handling. Professor Frank Tompa was my advisor during the first portion of my program and his support is gratefully acknowledged. I thank Professor George Labahn and Professor Keith Geddes of the Symbolic Computation Group for financial support. I also thank George Labahn for providing some advice along the way.

Professors Alberto Mendelzon, Ron Read, Ian Munro, Gord Cormack and Frank Tompa, my thesis committee members, provided many valuable comments on my work and this thesis.

During my stay at UW, I made many friends who have helped me in one way or another to complete this thesis. I particularly want to thank my office partner Lori Case and my bridge partners Andrej Brodnik, Glenn Paulley and Alfredo Viola.

Finally, I want to thank my parents for their support and encouragement over the years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The electronic storage and retrieval of information in large bodies of text, such as genetic databases, newspaper archives, dictionaries, encyclopedias and other reference works, requires the use of searching systems that are efficient both in time and storage requirements. Because of their large size, suffix trees[47], a standard text searching technique, have given way to other smaller but less effective data structures such as suffix arrays[35][22]. This thesis presents a new representation for a close relative of the suffix tree, the PAT tree, that retains the functionality of a suffix tree while requiring a fraction of the storage used by current methods. We also present methods for managing this structure on secondary storage for static and dynamic documents.

1.1 Overview

This chapter gives a brief introduction to some of the concepts necessary to understanding the remainder of the thesis and the motivations for our work. The second chapter reviews and improves some earlier results of Jacobson[27][26] required for our work and discusses some other applications of these results. The following three chapters present compact string searching indices for several classes of documents. In the conclusions, we summarize our results and discuss future work. The appendix contains a glossary of terms used in this thesis with short definitions and references to longer definitions occurring in the text. The first significant occurrence of any term defined in the glossary will be printed in italics.

1.2 Document Searching

We consider three different classes of documents in this thesis:

- static text in primary storage, such as a static compression dictionary,
- static text on secondary storage, such as an encyclopedia, dictionary or other reference work,
- dynamic text on secondary storage, such as electronic news or a genetic database.

Given a document, there are several types of queries that may be needed:

- search for a character sequence, e.g. find all occurrences of "qu" in a paper,
- search for a word, e.g. find all occurrences of the word "witch" in the works of Shakespeare; this may or may not exclude variations on the word such as witches or witching.
- search for a phrase, e.g. find all occurrences of "from M. E." in a dictionary,
- search for a regular expression, e.g. find all occurrences of "C(CA)*CT" in a genetic database.

In each case, the query may require the exact locations of all the matches, the approximate locations of the matches or simply a count of the number of occurrences of the pattern. An approximate location for a match might take the form of a range of character positions or, in the case of structured documents, a logical component of the document such as a chapter or paragraph that contains the match. When searching structured documents, it may also be desirable to restrict searching to particular components of the document (e.g. find the string "compact" in a title).

In order to answer these queries, we can either work directly on the input document using string searching algorithms, or we can preprocess the text and build an index that allows us to answer queries more quickly. String searching algorithms like those of Knuth, Morris and Pratt[30], Boyer and Moore[7] and other automata based methods can provide answers to the queries above in time linear in the document size. However, when the text is large and we expect multiple queries, it is often worthwhile building an index that allows faster searching than is possible with these algorithms. The index may restrict the type of queries that can be efficiently handled and the set of possible match points. A *word index* restricts matches to whole words, so a word index search for "the" would not return the occurrences inside "their" or "other." We also use the term word index for a *suffix index*, as explained later, where the matches are restricted to start at the beginning of a word but where no restriction is placed on the end of the match. Indices capable of finding matches starting at any character are called *character indices.* Each possible match start in the text is referred to as an *index point.* The storage requirements of most text indices are proportional to the number of index points in the text so word indices tend to be much smaller than character indices. While the ratio of words to characters varies with the language, writing style, author, and encoding, we use the ratio of 1:5 in this thesis based on our experience with our test documents.* Word indices may also exclude very common words like "and," "the," and "to," because they convey little information and the exclusion of these *stop words* can significantly reduce the size of the index. Finally, various mappings may be performed on the text and query prior to indexing and searching. *Stemming* is the removal of word suffixes, and occasionally prefixes, in an attempt to reduce a word to a canonical form. Stemming is used when the exact form of the word is not important. Stemming allows "witching" and "witches" to match the word query "witch." *Case conversion*, the mapping of all characters to lower case, is used when the capitalization of the word in the text is not important. Finally, the mapping of punctuation and other special characters, except those critical to the encoding, to blanks is commonly used during index building and searching. Case conversion and punctuation removal are irrelevant to our work, however they are used for all the experimental results we quote. These assumptions are made to more closely approximate real world searching conditions. Stemming is not used because the structures considered in this thesis are more appropriate when searching for arbitrary strings than when searching for words. While capable of efficiently answering word queries, these structures are more appropriate when searching for phrases as well as specialized applications where word searching is inappropriate, such as searching genetic information. Word searching is generally better handled by more specialized but limited structures such as those discussed in the next section.

1.3 Searching Methods

Several types of indices are known for efficiently answering some of the queries above, including:

*The actual average was near 5.5 but the exact ratio is not critical to our work

- Inverted Word Lists[16]: In an inverted word list, each distinct word in the text is stored in a secondary search structure with a list of all the occurrences of the word. Each occurrence may be a specific location or a more general location such as a paragraph or section number. Searching is performed by looking up the words in the query in the secondary structure and returning the associated word list. Inverted word lists handle word queries very well. Phrase searching is handled through multiple word queries and then combining and filtering the results. Regular expression matching is not supported. While inverted word lists appear to require one pointer per index point, the actual index is frequently much smaller because repeated references to a single region of text are often not stored, although a count may be kept. Witten, Bell and Neville[49] report on a word list implementation which requires approximately 30% of the original text size. However, that implementation made such extensive use of data compression that performance problems became apparent. While it is possible that recent advances in processor speed make this approach feasible, a more reasonable ratio seems to be 50-70% of the original text size for the index.

- Signature Files[16]: A signature file is constructed by first assigning a unique bit pattern, called a signature, to each word in the entire document. Then the input document is broken up into smaller pieces, such as paragraphs or sections, that we will refer to as sub-documents. A signature for each sub-document is constructed by bitwise "or"-ing the signatures for every word in the sub-document.[†] Given a query word or words, searching is performed by computing the signature for the query by "or"-ing the signatures for its components and then checking the set bits of the query's signature against the signatures of all the sub-documents using a linear

[†]Here we speak of the superimposed coding variety of signature files. Faloutsos and Christodoulakis[17] discuss this and several other variants having similar properties.

scan. Any sub-document that has a set bit corresponding to each set bit in the query's signature is a good candidate for a match. Because the bit patterns of the words may overlap, some sub-documents may match all the set bits of the query without containing the query words. For example, if we assign the words "to," "be," and "or" the signatures 010, 101 and 110 respectively, then the signature for "to be" is 111 and will match the signature for "or" in its set bits without containing the word "or." These false results are expected and must be filtered out by explicitly checking the query against each candidate sub-document. A careful selection of signatures keeps the number of false results small. The main advantages of signature files are their small size relative to other indices and the ease of updating the index to reflect changes to the document. Faloutsos and Christodoulakis determined that the size of the signature file should be approximately 10% of that of the text[17]. Signature files effectively handle word and phrase queries on moderately large documents but do not support regular expression searching. The linear behaviour of signature files makes them less effective for searching very large documents.

- Suffix Trees[47][36] and their derivatives are explained later in this chapter.

This thesis concentrates on a specific instance of the last type of index because it is capable of efficiently answering all of the queries given above and others.

1.4 Sample Documents

For presenting empirical results on text searching structures, we use four documents as test cases:

- Holmes, an ASCII encoded extract from the works of Sir Arthur Conan Doyle,

Name	#Characters	#Index Points
Holmes	238551	43745
Bible	5553621	1202504
OED	545578702	108687644
XIII	924430	924430

Table 1.1: Sample Documents

- Bible, an SGML[43] encoded version of the King James version of the Bible,
- OED, an SGML encoded version of the Oxford English Dictionary, Second Edition[39],
- XIII, ASCII encoded genetic information, the nucleic acid sequence for chromosone XIII from S. cerevisiae reported by the Sanger Centre.

In the first three cases we use word indices where word breaks occur at blanks, punctuation and SGML tags. The final document, XIII, is searched using a character index over the characters A,C,G and T. Various properties of these documents are shown in Table 1.1.

1.5 Preliminaries

We first clarify a few terms used in the remainder of the thesis. Throughout, the logarithm to the base 2 of x is denoted $\lg x$ and the natural logarithm is denoted $\ln x$. We use $\log x$ when the base of the logarithm is not crucial and can be taken to be any constant greater than 1. When the base is important, it will be placed in a subscript, as in $\log_b(x)$. Ceiling, $\lceil x \rceil$, is the smallest integer greater than or equal to x and floor, $\lfloor x \rfloor$, is the largest integer less than or equal to x. When discussing the representation of a number in a base other than ten, we place the value in parentheses and use a subscript to indicate the base, as in $13 = (15)_8$. We will also use Iverson's convention for representing conditional expressions where [..] is one if the contents are true, zero otherwise(see Graham *et al.*[25]).

When analysing the performance of our data structures and algorithms we will make use of order notation. Briefly, f is $O(g)$ if there exist positive constants k and n_o such that $f(n) < kg(n)$ for all $n > n_o$. Similarly f is $\Omega(g)$ if g is $O(f)$, f is $\Theta(g)$ if f is $O(g)$ and f is $\Omega(g)$ and finally f is $o(g)$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ [25].

When reporting the sizes of objects in main memory or on secondary storage, we use the suffixes k,m, and g to denote multipliers of 2^{10}=1024, 2^{20}=1048576 and 2^{30}=1073741824 respectively.

1.5.1 Trees

The trees dealt with in this thesis fall into two categories:

- finite, rooted, and ordered *general trees*, and
- finite rooted *binary trees*.

A general tree T (we omit the "finite, rooted" and "ordered" qualifiers for the remainder of the thesis) is formally defined as a non-empty, finite set of nodes such that there is one distinguished node called the *root* of the tree, and the remaining nodes are partitioned into $m \geq 0$ disjoint sub-trees $T_1, T_2, T_3 ... T_m$ where the order of the sub-trees is significant[41]. Nodes having no sub-trees are called *leaves* or *external nodes*. Nodes with sub-trees are called *internal nodes*. The *degree* of a node is the number of sub-trees of the node.[‡] The roots of the sub-trees of a node are called the *children* of the node. Multiple nodes that are the children of a

[‡]Some authors use the term "out-degree" for the number of sub-trees, reserving "degree" for the total number of edges incident on the node.

common node are called *siblings* (the roots of T_1, T_2, T_3 and T_m are all siblings). Conversely, if a node has a child then it is referred to as the *parent* of the child node. We define the *height* of a node recursively, with leaf nodes having height one and internal nodes having a height one greater than the maximum of the heights of their children. The height of a node is the length of the longest path from the node to a leaf in its sub-tree. The height of a tree is the height of its root. Trees are frequently depicted pictorially by placing the nodes in the plane and then drawing lines between nodes and their children. The connections betweens nodes and their children are called *edges* in both the concrete diagram and the abstract tree definition. Figure 1.1 shows a drawing of a sample tree. All drawings of trees in this thesis place tree roots higher on the page than their sub-trees.

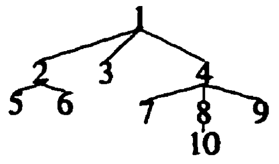

Figure 1.1: Sample Tree

When computing with trees, we will primarily be interested in three operations:

parent(n) given (a representation of) a node n, computes the representation of the parent of that node, and

degree(n) given a node n, computes its degree, and

child(n, i) given a node n and an integer i returns the i'th child of n.

A binary tree consists of a distinguished node called the root. The root may have a left sub-tree and/or a right sub-tree each of which must itself be a binary tree. The important difference between a binary tree and a general tree where each node has a degree of at most two is that each sub-tree of a binary tree node is labelled left or right - even if there is only one of them. The general tree terminology of degree, parent, child, leaf and internal node is also used with binary trees. Figure 1.2 shows a drawing of a binary tree. In some portions of

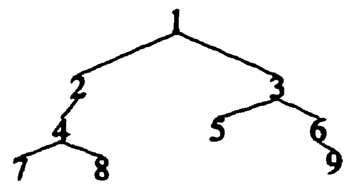

Figure 1.2: Sample Binary Tree

this thesis we will deal with a restricted form of binary tree in which every internal node has exactly two sub-trees. We call such trees *strictly binary trees*. When working with binary trees, we retain the **parent** and **degree** operations but replace the **child** operation with two operations: **leftchild** and **rightchild**.

For both general and binary trees we may extend the definition to include a degenerate empty tree as consisting of no nodes.

1.5.2 Machine Model

We use the *MBRAM* model of computation allowing indirect addressing of memory and basic mathematical and bit-wise boolean operations. The MBRAM model incorporates an accumulator, an infinite set of memory registers each with a unique integer address and each capable of holding a value which may be interpreted as either an integer or a finite bit sequence, and a finite control containing a program made up of instructions from the following categories:

- flow control instructions including conditional and unconditional branches,
- input/output instructions,

- load/store instructions that transfer data between the accumulator and the memory registers,
- arithmetic and bitwise boolean operations that operate on the contents of the accumulator and memory.

The main distinguishing feature of the RAM based models is the ability to perform indirect loads and stores. A memory load (store) can load from (store to) a memory register whose address is taken from a second register. Indirect memory operations allow the RAM to perform table lookup and pointer operations efficiently. The ability to perform multiplication, division and bitwise boolean operations separates the MBRAM from the basic RAM model. This description is based on that of van Emde Boas[15].

To measure the performance of a program, we assign a cost to each instruction and define the cost of the program as the sum of the costs of all the instructions executed by the program. All of our analyses will be done using a unit cost approach where each instruction has cost one as opposed to the logarithmic cost model where the cost of an instruction is based on the length of the operands in bits. The use of a unit cost approach allows easier comparison to existing work, but can result in unreasonable results if care is not taken (for further information, see [15]). In order to avoid these problems, we will ensure that at no time will any computation use integers greater than n^c or bit sequences of length greater than $c \lg n$ where n is a reasonable measure of the problem size and c is a small constant. Usually n is the document size but occasionally it will be the number of nodes in a tree. Such a restriction allows us to obtain a bound on the logarithmic cost simply by multiplying the unit cost by $c \lg n$. We will refer to the MBRAM model using the unit cost approach as a *wide bus* model to distinguish it from one where bit operations are counted.

Offset	Suffix	Unique Prefix
1	*abccabca$*	*abcc*
2	*bccabca$*	*bcc*
3	*ccabca$*	*cc*
4	*cabca$*	*cab*
5	*abca$*	*abca*
6	*bca$*	*bca*
7	*ca$*	*ca$*
8	*a$*	*a$*

Table 1.2: Suffixes of *abccabca$*

1.6 Suffix Trees and Related Structures

The structures presented in this section have been discussed, with minor variations, by several authors and indeed have acquired several names in the process. In addition, some names have been used to refer to several distinct methods of searching. We will attempt to keep the terminology simple by choosing one name for each structure and refer only parenthetically to other names.

Given a text string $S = s_1s_2s_3...s_n$, where each s_i is a member of an alphabet Σ, we want to preprocess S such that given a pattern, $P = p_1p_2p_3...p_m(p_i \in \Sigma)$, the set $\{i|s_i..s_{i+m-1} = P\}$ can be found as efficiently as possible. By a *suffix* of S we mean any substring of S ending in the final position, i.e. $s_i...s_n$ for some value of i between one and n. In order to ensure that each suffix occurs exactly once in the text, a special character "$," not in Σ, is appended to S. The string $S =$ *abccabca$* will be used for many of our examples. The suffixes of S are shown in Table 1.2. Each suffix has a minimal prefix that distinguishes it from the other suffixes. This prefix appears in the third column of Table 1.2. If S did not end in $, the strings *ca* and *a* could not uniquely identify the last two suffixes of S

because they occur elsewhere in the string. In order to locate all occurrences of *bc*, it is sufficient to search for suffixes that start with that character string. While this search problem may not seem easier than the original problem, many existing structures for searching a set of keys can be used to solve it. We refer to any index that operates by searching the suffixes of the text as a *suffix index*.

Given a set of unique string keys, a trie[20] is a search tree in which each leaf contains one of the strings and each edge has a single character label. A string occurs in the sub-tree rooted at a node if and only if the concatenation of the labels on the path from the root to the node is a prefix of the string. Each internal node has children for each continuation of its prefix that leads to one of the keys. Depending on the implementation, the construction continues until either there is only one key remaining or until the end of the key is reached. Structurally, a trie is a general tree where each node has degree at most m, where m is the size of the alphabet, and each edge has a label drawn from Σ such that no two edges below the same node share a label.

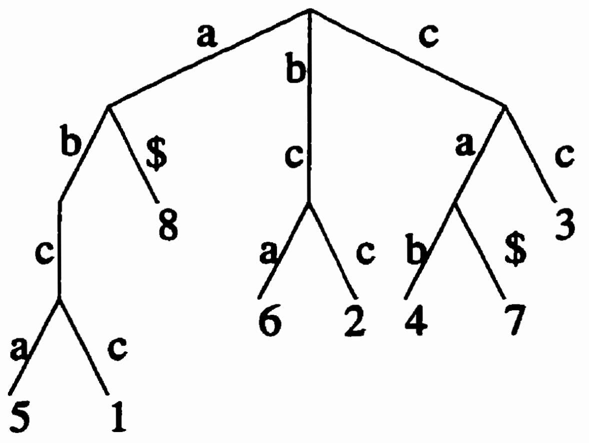

Figure 1.3: Suffix Trie

The *suffix trie*[47](also called position tree, non-compact suffix tree and FuTrie) of S is a trie built on the unique prefixes of the suffixes of S. The suffix trie for the example string is shown in Figure 1.3. Each leaf is labelled with the suffix offset. In the suffix trie, the labels along any root-leaf path can be concatenated to obtain the unique identifier of the suffix stored in the leaf. Similarly, the labels along any root-vertex path can be concatenated to form a maximal prefix shared by the suffixes of all the leaves in the sub-tree rooted at the node. Using the suffix trie, it is possible to search for any pattern by traversing the trie until either the end of the pattern is encountered or the search encounters a leaf. If the end of the pattern is encountered, any leaf in the sub-tree rooted at the last node visited is a match. If a leaf is encountered before the end of the pattern, then the remainder of the pattern must be checked against the appropriate suffix. The cost of performing this search is $O(m+q)$ where m is the size of the pattern and q is the size of the answer. The dependence of the search cost on the size of the answer can be avoided if we are permitted to return the root of the sub-tree of answers instead of some other representation of the answer set. By convention, the dependence on the answer size is omitted in the remainder of this thesis. In most cases, the conversion of the node to a list of suffix offsets can be performed in linear time (the suffix trie being a notable exception).

While a suffix trie allows $O(m)$ searching, it can require $\Theta(n^2)$ nodes: consider the suffix trie for $S = a^n b^n a^n b^n \$$ (for $n = 3$ the trie is shown in Figure 1.4). In general, this trie has $(n+1)^2$ internal nodes so it is possible for suffix tries to require $\Theta(n^2)$ space to index a string of size $O(n)$.

Suffix trees (also called compact suffix trees, prefix trees, subword trees, position trees and OrTries)[47][36], reduce the storage requirements by removing some or all of the degree one nodes in the suffix trie. For the example string, $S = abccabca\$$, the suffix tree obtained by merging all degree one nodes is shown in Figure 1.5. This structure allows efficient searching, $O(m)$ time, and uses only $O(n)$ storage if the node labels are stored as pointers into the text. The form of suffix tree shown here is due to McCreight, while Weiner retained up to n degree

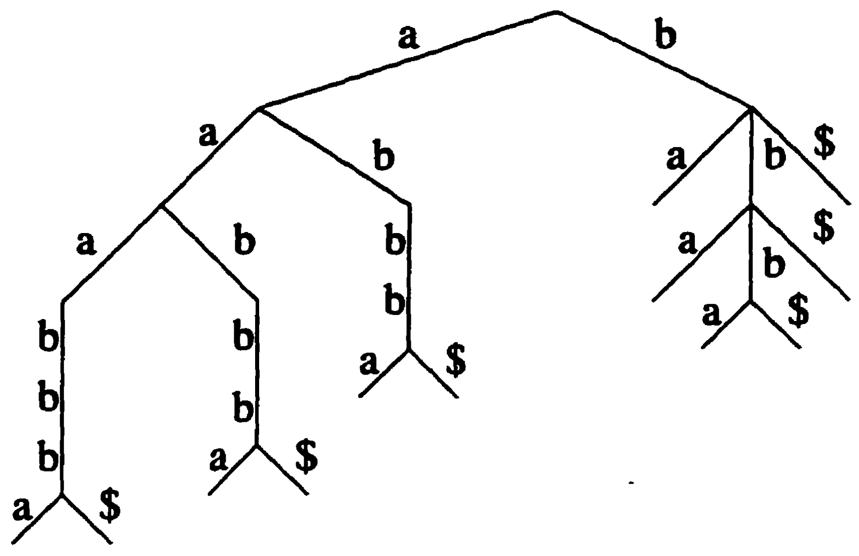

Figure 1.4: $\Theta(n^2)$ Suffix Trie

one nodes. Both Weiner and McCreight showed $O(n)$ time algorithms for the construction of their respective forms of the suffix tree. Another linear time construction algorithm relating suffix tries and Directed Acyclic Word Graphs (DAWGs), a data structure briefly discussed later, is presented by Chen and Seiferas[10].

The final suffix tree structure considered here is obtained by using the *PATRICIA* searching method of Morrison[38] to search the suffixes. Given a set of unique string keys and a binary encoding of the letters in $\Sigma \cup \{\$\}$, a PATRICIA tree is a search tree in which each leaf contains one of the strings and each internal node is labelled with the position of a bit that distinguishes the keys in the left sub-tree from those in the right sub-tree. We use the first bit that is not identical in all the keys in a sub-tree to partition the keys into the left and right sub-trees. The use of PATRICIA for suffix searching is implicit in Morrison's paper and is also discussed by Knuth[29], Sedgewick[42] and Gonnet *et al.*[22]. Following Gonnet *et al.*, we use the term *PAT tree* for the PATRICIA tree applied to suffix searching.

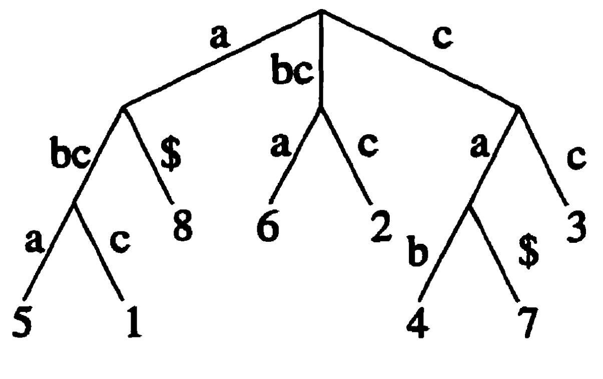

Figure 1.5: Suffix Tree

Using the encoding $a = 00$, $b = 01$, $c = 10$ and $\$ = 11$, the PAT tree for the example string is shown in Figure 1.6.

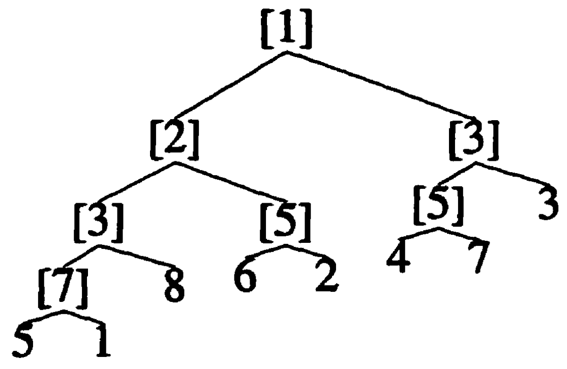

Figure 1.6: PAT Tree, [k] is the node's bit offset

A PAT tree is searched by generating the binary encoding of the pattern and then traversing the tree. At each internal node, the bit offset is used to select a bit from the pattern. Based on the bit value, the search continues with either the left or the right child of the node. Because the search can skip bits in the pattern, the termination of the search is more complex than that of simple suffix trees. If the search terminates at a leaf node, then the pattern must be compared to the leaf suffix to see if it matches. If the end of the pattern is encountered before a leaf, then a representative suffix from the current sub-tree must be chosen and compared to the pattern. The representative matches the pattern if and only if all of the suffixes in the sub-tree match the pattern. In practice, the offset information stored in each node is a skip value one less than the difference between the offset value of the node and its parent (with an implicit parent offset of 0 for the root). The actual offset is accumulated as the tree is traversed. Because this structure is central to this thesis, the pseudo-code for searching it is shown in Figure 1.7. Provided care is taken to ensure that locating a sample for

```
node = root
offset = 0
bit_pattern = encode(pattern)
bit_length = |bit_pattern|
while (node is not a leaf and offset + node.skip <= bit_length)
   offset = offset + node.skip + 1
   if bit_pattern[offset] is set then node = rightchild(node)
   else node = leftchild(node)
   endif
end
sample = any sub-leaf of node
if (pattern = S[sample]) return node
else return NOTFOUND
endif
```

Figure 1.7: PAT Tree Search

comparison can be performed efficiently, the search cost of PAT trees is the same as that of suffix trees, $O(m)$.

1.7 Properties of Suffix Trees and PAT Trees

Each PAT tree is a strictly binary tree in that each node has exactly zero or two children. In a strictly binary tree the number of leaves is exactly one more than the number of internal nodes, so a PAT tree has one fewer internal nodes than there are index points in the text. Note that there is an isomorphism between the strictly binary trees with $2n+1$ nodes and the binary trees on n nodes obtained by removing all the leaf nodes from the strictly binary tree. This isomorphism allows us to represent the structure of a PAT tree more succinctly in the following chapters.

A second useful property of PAT trees is the expected height of the tree. The technical report of Szpankowski[46] contains many results on the expected case behaviour of tries and suffix trees. We use his notation for the remainder of this section. Szpankowski shows that, subject to some apparently reasonable conditions on the text, the height of a suffix tree on a text of length n, H_n, satisfies $\lim_{n\to\infty} \frac{H_n}{\ln n} = \frac{1}{h_3}$ *almost surely*, where $h_3 = -\lim_{n\to\infty} \frac{\ln \max P(X_1^n)}{n}$ is a parameter of the model that generated the text. In this expression, $P(X_1^n)$ is the probability of a particular character sequence of length n occurring and the maximum is taken over all such strings. The criteria for this result to hold essentially state that:

1. The sequence of characters in the underlying string is drawn using a stationary ergodic model under which a character's probability distribution can depend on the previous characters.
2. No sequence of characters fully implies any following character.
3. The influence of any given character sequence on the probability distributions of characters occurring later in the string decreases rapidly with the distance between the two sets of characters (strong mixing condition).

While written text is not random, the conditions above constitute a realistic model for reasoning about the behaviour of documents. In order to apply this result to PAT trees we must replace character sequences by bit sequences.

Provided we use an efficient encoding of the alphabet the conditions on the character strings imply similar conditions on the bit strings so the same result will hold for PAT trees. The efficient coding of the alphabet will also be required for compact storage of the PAT tree.

Another parameter studied by Szpankowski is the expected height of a suffix trie. This value is of interest to us because it is linearly related to the maximum offset occurring in the PAT tree. He shows that the height of a suffix trie, reusing H_n, satisfies $\lim_{n\to\infty} \frac{H_n}{\ln n} = \frac{1}{h_2^{(1)}}$ almost surely where $h_2^{(1)} = -\lim_{n\to\infty} \frac{\ln \sum_{X_1^n} P^2(X_1^n)}{2n}$ is another parameter of the model generating the text. These two results will be used throughout this thesis to derive logarithmic bounds on many properties of our structures.

1.8 Storage Requirements

Computer representations of suffix based structures require the use of pointers and text offsets. For the purposes of comparison, it is useful to assume that each of these require lg n bits. Manber and Myers performed an analysis of various possible representations of suffix trees and determined that approximately 17 bytes per index point were used in the most compact representations[35]. Under the assumption that their system was capable of handling documents of at most 2^{32} characters, 17 bytes equates to 4.25 lg n for the texts considered in their report. This result agrees with the "4n to 5n words" reported by Gonnet *et al.*[22]. The use of PAT trees can reduce the storage requirements to approximately $2 + 3\lg n$ bits per index point if the obvious implementation of a node as two words, an integer skip, and bit flags indicating if the words contain pointers to other nodes or suffix offsets is used. On the OED, this implementation results in a word index roughly twice the size of the text - a suffix tree would be three times the size. If all the characters in a document are indexed, the size of each of these indices will be increased by a factor of approximately five as discussed earlier.

The storage costs of suffix trees have long been a cause for concern. Several researchers have developed other suffix based structures that tradeoff the searching abilities of suffix trees against their storage costs. These include,

- *Suffix Arrays*[35][22]: Due to the large storage requirements of suffix trees Manber and Myers and Gonnet *et al.* independently developed the suffix array (Gonnet *et al.* used the term PAT array) structure. The suffix array structure is simply the list of suffix offsets sorted by the lexicographic ordering of the suffixes to which they refer. The suffix array for the example string is: $\boxed{5}\boxed{1}\boxed{8}\boxed{6}\boxed{2}\boxed{4}\boxed{7}\boxed{3}$. In practice, suffix arrays are searched using a binary search requiring $O(m \lg n)$ time because each comparison can require $O(m)$ character comparisons. Manber and Myers, however, give a secondary structure that can boost the searching speed to $O(m + \lg n)$. Both Manber and Myers and Gonnet *et al.* have found it necessary to add an auxiliary structure to the base suffix array to speed searching on secondary storage[35][45]. These auxiliary structures range in size from 25% to 100% of the size of the suffix array. Barbosa *et al.*[5] investigated methods to speed searching on secondary storage by modifying the pivot element(s) chosen during the binary search to reduce the time taken to perform the disk accesses.

- Directed Acyclic Word Graphs (DAWGs): DAWGs are obtained by merging edge isomorphic sub-trees in a suffix trie[10]. The number of nodes in the DAWG is linear in the size of the text. While DAWGs can detect the occurrence of phrases in a document, they cannot determine the locations of the occurrences or associate any data with each occurrence so they are not appropriate for most text searching applications.

- Bucketed Suffix Trees[4]: A bucketed tree allows multiple suffix offsets to occur in each leaf. Searching is performed by using the tree to determine the bucket and then using brute force or binary search within the buckets. The storage savings and performance cost vary with the size of the buckets but the storage cost is comparable to that of suffix arrays because the suffix offsets are still stored.

- Prefix B-Trees[6]: A Prefix B-Tree extracts common prefixes of keys and uses minimal separators of the keys to increase the branch factor of each node. While not developed for suffix searching, Prefix B-Trees are a competitive approach to the problem. The SB-Tree, discussed later, appears to subsume the Prefix B-Tree for suffix searching and so will be used for any comparisons.

- PaTries[44],[37]: PaTries are similar in approach to the structures presented here but, because of the lack of an efficient tree encoding and the use of a non-optimal partitioning scheme, are unlikely to be competitive on either processor time or secondary storage accesses.

- LC-Tries[2]: By storing a compressed trie in main memory, Andersson and Nilson produce a compact pre-index for a suffix array on secondary storage. The method is unlikely to scale well to very large text but could be applied recursively to improve performance. Such a recursive application would have much in common with the PaTrie.

- SB-Trees[19]: Essentially a B-tree on the suffixes where each node in the B-tree node contains a PATRICIA tree that distinguishes the keys in that node. While the storage requirements of the SB-tree are subject to many performance tradeoffs, the use of a compact tree representation allows implementations requiring from 3 to 12.3 bytes per node for documents of up to 2^{40} characters. The smaller representations incur performance penalties to achieve their small size. Direct comparison of the SB-tree to our structure is difficult because of the many space-time tradeoffs, however an empirical study is being considered and will be reported on later.

The remainder of this thesis presents several compact representations for PAT trees with storage requirements comparable to the structures above but offering greater efficiency or functionality. Our interest in these structures was initially motivated by the problem of efficiently searching text on CD-ROM. Due to the high performance penalties incurred when using suffix arrays on these devices, we started searching for alternative methods of searching text. During the investigation it became clear that the primary motivation for suffix arrays was the high storage cost for suffix trees. This lead to an investigation of suffix tree representations and hence to compact PAT trees.

Chapter 2

Compact Trees and Tries

In this chapter, we present techniques for constructing traversable compact representations of trees and tries. We first review some tree representations from Guy Jacobson's thesis and then extend these methods to the MBRAM model of computation. We also show how these methods can be used to construct a compact traversable representation for tries. Finally, we present a second compact traversable representation for binary trees that is again based on earlier work by Jacobson but with improved space efficiency.

Jacobson[27][26] presents several compact representations of binary trees and unlabelled general trees with efficient, in terms of bit accesses, implementations for the selection of the parent and children of a node. We first review two such representations and then extend Jacobson's results to the wide bus model of computation we are using. Using these tools we develop a new representation for tries by constructing a representation for trees with edge labels and efficient selection of an edge based on its label. This last result is based on simple hashing so the performance claims are probabilistic, although low fill ratios make reasonable performance likely. Finally we present an improved version of another of Jacobson's encodings. These representations will be used in the following chapters to represent the tree structure of a PAT tree.

We are interested in compact representations of various types of trees and tries that allow common tree traversal operations to operate directly on the compact form of the tree. The operations we will be interested in include:

- selecting the left or right child of a binary tree or selecting one of the children of a node in a general tree based on ordinal number, or, in the case of a trie, edge label,
- locating the parent of a node,
- determining the size of the sub-tree rooted at a node.

In each case, we require that the operations be performed in a constant number of operations on $\lg n$ size objects so they will operate in constant time on the MBRAM model. For the purposes of this chapter, we let n represent the number of nodes in the trees being discussed instead of a document size. Jacobson's thesis[27] presents two different methods of efficiently representing and traversing trees: rank/select directories and a recursive encoding for binary trees.

2.1 Rank/Select Representations

Jacobson[27][26] defines two new operations, *rank* and *select*, on bit-maps that can be efficiently implemented and are crucial in manipulating his compact bit map representations of trees and other structures. The operations are defined as:

- **rank**(x) computes the number of ones preceding (to the left of) and including the bit in position x,
- **select**(x) computes the position of the x'th one in the bit map.

Note that **rank(select(x))** $= x$ and **select(rank(x))** $= x$ if the x'th bit is a one. Also define **rank0** and **select0** as performing the analogous operations of counting or finding zeroes instead of ones.

2.1.1 Binary Trees

The number of binary trees on n vertices is denoted C_n and called the n'th *Catalan* number, $C_n = \frac{1}{n+1}\binom{2n}{n}$[41]. A compact encoding of a binary tree structure should require about $\lg C_n$ bits. Using Stirling's approximation to the logarithm of the factorial function, $\lg C_n$ can be shown to be approximately $2n$ (see Section 2.3 for a full derivation). The survey papers of Mäkinen[34] and Katajainen and Mäkinen[28] present many techniques for representing binary trees that attain the $2n$ bound, however none provide the functionality required.

For representing binary trees, Jacobson starts with a level order binary tree encoding. Consider the tree in Figure 2.1. To form the level order encoding first

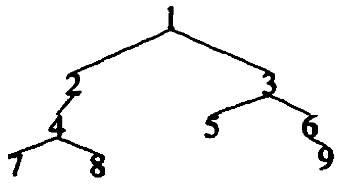

Figure 2.1: Sample Binary Tree

extend the tree by adding new leaf nodes below each leaf or non-full internal node in the original tree. Then assign a 1 to each node that exists in the original tree and a 0 to each leaf in the extended tree, as in Figure 2.2. Note that the extended tree is a strictly binary tree in that all internal nodes have degree two. The level order encoding of the tree is created by performing a level order traversal of the extended tree and recording the labels on the nodes encountered. The level order encoding of the tree in Figure 2.1 is

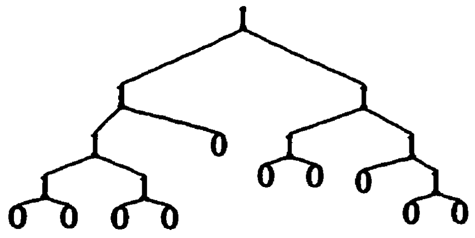

Figure 2.2: Labelled Extended Tree

1	1	1	1	0	1	1	1	1	0	0	0	1	0	0	0	0	0	0
1	2	3	4		5	6	7	8				9						

where the node numbers appear below their position in the encoding. The bits comprising the encoding are broken into segments of five solely for ease of reading. This encoding occurs in the construction of Zaks' sequences[50] and was also studied by Lee *et al.*[31]. This level order encoding requires $2n+1$ bits to represent a tree on n nodes, so it is near-optimal, but it does not appear to support the efficient location of the parent or children of a node. Jacobson noted that using the **rank()** and **select()** operations, the parent and child operations can be computed as

$$\begin{aligned}\textbf{leftchild}(x) &= 2\,\textbf{rank}(x)\\ \textbf{rightchild}(x) &= 2\,\textbf{rank}(x)+1\\ \textbf{parent}(x) &= \textbf{select}\left(\left\lfloor\frac{x}{2}\right\rfloor\right)\end{aligned}$$

where each function takes the offset of the node and returns the offset of the child or parent. If the bit at the offset returned by the child operations is zero, then that child is not present in the tree. As an example, node 4 is located at offset 4, so its right child is at offset $9 = 2*4+1$. Similarly, the parent of the node at offset 13, node 9, is at $\textbf{select}(6 = \left\lfloor\frac{13}{2}\right\rfloor) = 7$ which is the offset of node 6. Formulas similar to the equations above occur in the implicit representation of complete binary trees used by Williams in Heapsort[48] where the special form of the tree reduces rank and select to identity functions.

2.1.2 General Trees

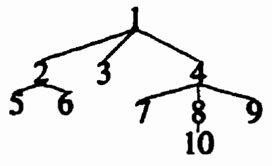

Figure 2.3: Sample General Tree

The representation of general trees, such as the one in Figure 2.3, provides another use for **rank()** and **select()**. By observing the well known isomorphism with the binary trees obtained by mapping a node's first child to its left child and its right sibling to its right child, one can determine that there are C_n such trees on n nodes and so $2n$ bits are again sufficient. However, using this isomorphism to represent such trees results in a sequential scan of the children of a node in order to locate the encoding of a particular child. Instead, Jacobson uses an encoding from Read[40] that is again based on a level order traversal of the nodes. The encoding is obtained by labelling each node with the unary encoding of its child count using ones for the count and a zero for the terminator. In order to represent the empty tree, an extra "super-root" is added above the real root of the tree. Figure 2.4 shows the tree from Figure 2.3 using this encoding. Again generate the

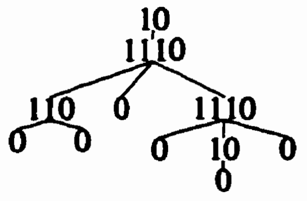

Figure 2.4: Unary Labelling of Sample Tree

bit representation for the tree using a level order traversal of the tree. The bit string for Figure 2.4 is

1	0	1	1	1	0	1	1	0	0	1	1	1	0	0	0	0	1	0	0	0
-		1				2			3	4				5	6	7	8		9	10

Note that each node has one '1' bit, found in its parent's label, and one '0' bit, terminating its label, associated with it plus an extra '0' bit for the super-root so $2n + 1$ bits are used in encoding.

Again, efficient implementations of the traversal operations are not obvious with this representation, but Jacobson shows that the **rank()** and **select()** operations can be used to implement these operations. In this case, if we represent a node by the offset of the corresponding one bit in the parent's label, then:

$$\begin{aligned}\textbf{degree}(x) &= \textbf{select0}(\textbf{rank}(x)+1) - \textbf{select0}(\textbf{rank}(x)) - 1\\ \textbf{child}(x,i) &= \textbf{select0}(\textbf{rank}(x)) + i\\ \textbf{parent}(x) &= \textbf{select}(\textbf{rank0}(x))\end{aligned}$$

where children are numbered starting from 1. For convenience, we refer to the rank of the set bit representing a node as the rank of the node for both general and binary level order encoded trees. This convention allows us to associate a unique integer in range $1..n$ with each node. In Figures 2.1 and 2.3, a node's label is equal to its rank.

2.2 Implementing Rank and Select

Jacobson[27][26] presented implementations of **rank()** and **select()** that are efficient in terms of the number of bits accessed. However, his implementation of **select()** requires non-constant time when run under the MBRAM model of computation. In this section we review Jacobson's implementation of **rank()**, which runs in constant time on an MBRAM, and present a new implementation of **select()** that runs in constant time on an MBRAM.

2.2.1 Jacobson's Rank Implementation

Given a string of n bits, Jacobson constructs a two-level auxiliary directory structure allowing constant time computation of the rank function The first auxiliary directory contains $\mathbf{rank}(i)$ for every i a multiple of $\lceil \lg n \rceil^2$.* A second auxiliary directory contains $\mathbf{rank}'(j)$ for j a multiple of $\lceil \lg n \rceil$ within each subrange where $\mathbf{rank}'$ computes the rank within the subranges of size $\lceil \lg n \rceil^2$.

Theorem 2.1 (Jacobson) *Rank can be performed on an MBRAM in constant time using* $\frac{2n \lg \lg n}{\lg n} + O(\frac{n}{\lg n})$ *bits of extra space.*

Proof: $\mathbf{rank}(x)$ is calculated by locating the correct first auxiliary directory entry, at position $\left\lfloor \frac{x}{\lceil \lg n \rceil^2} \right\rfloor$, and the correct second level entry, at position $\left\lfloor \frac{x}{\lceil \lg n \rceil} \right\rfloor$. Adding these two values gives the rank of the first bit in a $\lceil \lg n \rceil$ sized range. The final component of $\mathbf{rank}(x)$ could be computed by scanning $x \bmod \lceil \lg n \rceil$ bits in the bit string. Using table lookup, however, this scan can be performed in constant time. We simply retain a table which for each possible bit pattern of length $\frac{\lg n}{c}$, for some integer $c > 1$ ($c = 2$ suffices), gives the number of 1's in the pattern. After masking out unwanted trailing bits, our final term is found by adding at most c entries in the table. There are approximately $n^{\frac{1}{c}}$ entries in this table.

Each of the $\left\lceil \frac{n}{\lceil \lg^2 n \rceil} \right\rceil$ entries in the first auxiliary directory requires $\lceil \lg n \rceil$ bits. Each of the $\left\lceil \frac{n}{\lg n} \right\rceil$ entries in the second level auxiliary directories requires

*Jacobson actually uses $\ln n \lg n$.

$2\lceil \lg \lceil \lg n \rceil \rceil$ bits because they contain values less than $\lceil \lg n \rceil^2$. The total storage requirement, ignoring floors and ceilings, for these directories is $\frac{n}{\lg n} + \frac{2n \lg \lg n}{\lg n}$. The final table requires fewer than $\lceil n^{\frac{1}{c}} \rceil \lceil \lg \lceil \lg n \rceil \rceil$ bits. Hence the storage required in addition to the original bit map is $\frac{2n \lg \lg n}{\lg n} + O(\frac{n}{\lg n})$. *QED*

2.2.2 Implementation of Select on an MBRAM

While Jacobson's ranking function operates in constant time on a MBRAM, his implementation of **select()** requires $\Theta(\log \log n)$ time because it includes a binary search on a region of $\lg^2 n$ bits. In this section, we present a new solution that operates in constant time under a wide bus model. As in the rank case, we use a multi-level auxiliary directory structure and find that the final case can be scanned using a constant number of table lookups. Our goal is to use $O\left(\frac{n}{\lg \lg n}\right)$ extra bits to store the auxiliary directory structure. In order to achieve this goal, we will ensure that for the ranges of length r in the auxiliary directory structures we will use $\left\lfloor \frac{r}{\lg \lg n} \right\rfloor$ bits of storage. This condition also allows us to index into these directories to locate the appropriate bit sequences for each range.

Theorem 2.2 *Select can be performed on an MBRAM in constant time using* $\frac{3n}{\lceil \lg \lg n \rceil} + O(n^{\frac{1}{2}} \lg n \lg \lg n)$ *bits of extra space.*

Proof: We use three levels of auxiliary directory structures to compute select. The first auxiliary directory records the position of every $\lceil \lg n \rceil \lceil \lg \lg n \rceil$'th one bit. This directory requires $\lceil \lg n \rceil$ bits for each entry and has $\left\lfloor \frac{n}{\lceil \lg n \rceil \lceil \lg \lg n \rceil} \right\rfloor$ entries so at most $\left\lfloor \frac{n}{\lceil \lg \lg n \rceil} \right\rfloor$ bits are used. Let r be the size of a subrange between two values in the first auxiliary directory and consider the sub-directory for this range. Note that during traversal operations r is easily computed and does not need to be stored explicitly. We are willing to spend $\left\lfloor \frac{r}{\lceil \lg \lg n \rceil} \right\rfloor$ bits on this range in the second level of directories. As with the inference of the value r, we can also infer the location of this block of bits. To explicitly record the $\lceil \lg n \rceil \lceil \lg \lg n \rceil$ possible answers in that range requires $\lceil \lg n \rceil^2 \lceil \lg \lg n \rceil$ bits. If

$$r \geq \lceil \lg n \rceil^2 \lceil \lg \lg n \rceil^2$$

then

$$\left\lfloor \frac{r}{\lceil \lg \lg n \rceil} \right\rfloor \geq \lceil \lg n \rceil^2 \lceil \lg \lg n \rceil$$

and we have sufficient storage to explicitly record the answers.

If instead

$$r < (\lceil \lg n \rceil \lceil \lg \lg n \rceil)^2 \tag{2.1}$$

we re-subdivide the range and record the position, relative to the start of the range, of each $\lceil \lg r \rceil \lceil \lg \lg n \rceil$'th one bit in the second level auxiliary directory. Each entry requires $\lg r$ bits and there are at most $\left\lfloor \frac{r}{\lceil \lg r \rceil \lceil \lg \lg n \rceil} \right\rfloor$ entries so again this takes at most $\left\lfloor \frac{r}{\lceil \lg \lg n \rceil} \right\rfloor$ bits.

Let r' be the size of a subrange between values in the second level auxiliary directory. To explicitly record the relative positions of all the possible answers requires $\lceil \lg r' \rceil \lceil \lg r \rceil \lceil \lg \lg n \rceil$ bits. If

$$r' \geq \lceil \lg r' \rceil \lceil \lg r \rceil \lceil \lg \lg n \rceil^2$$

then

$$\left\lfloor \frac{r'}{\lceil \lg \lg n \rceil} \right\rfloor \geq \lceil \lg r' \rceil \lceil \lg r \rceil \lceil \lg \lg n \rceil$$

so there is sufficient space to record all the answers in a third level of auxiliary directories.

In the final case we have

$$r' < \lceil \lg r' \rceil \lceil \lg r \rceil \lceil \lg \lg n \rceil^2 . \tag{2.2}$$

From equation 2.1 we obtain

$$\lg r < 2 (\lg(\lceil \lg n \rceil) + \lg(\lceil \lg \lg n \rceil))$$

so $\lg r < 4 \lceil \lg\lg n \rceil$ by observing that $\lg \lceil \lg\lg n \rceil < \lg \lceil \lg n \rceil$ and $\lg \lceil \lg n \rceil \leq \lceil \lg\lg n \rceil$. In addition we know $r' < r$ so $\lg r' < \lg r$ and equation 2.2 implies $r' < 16 \lceil \lg\lg n \rceil^4$. Because $(\lg\lg n)^4$ is asymptotically smaller than $\lg n$ we know that we can perform select on a range of $\lceil \lg\lg n \rceil^4$ bits using a constant number of operations on regions of size $\lceil \lg n \rceil$ bits. Computing select on a small range of bits is again performed using table lookup. Again let c be an integer greater than one. For each possible bit pattern of length $\frac{\lg n}{c}$ and each value i in the range $1..\frac{\lg n}{c}$ we record the position of the i'th one in the bit pattern and, in a separate table, the number of ones in the bit pattern. To compute select on a small range we scan the range using the second table until we know which subrange contains the answer and use the first table to compute the answer. At most a constant number of subranges can be considered.

Select(k) is performed by locating the pair of first auxiliary directory entries bracketing the desired value starting at $\frac{k}{\lceil \lg n \rceil \lceil \lg\lg n \rceil} \lceil \lg n \rceil$ and from these computing r. If $r \geq (\lceil \lg n \rceil \lceil \lg\lg n \rceil)^2$ the storage in the second level auxiliary directory is treated as an array and the correct answer is read off from the correct entry. Otherwise a similar search is performed in the second level auxiliary directory resulting in either a scan of a small number of bits or reading the answer from the third level auxiliary directory and then summing the results from all the levels. The critical point is that we know where the appropriate directory bits at each level are located and how to interpret them based on the value of k and the preceding directory levels. The storage used for the auxiliary directories is $\frac{3n}{\lceil \lg\lg n \rceil}$ and the storage used for the lookup tables is $n^{\frac{1}{c}} \left(\frac{1}{c} \lg n \lg\lg n + \lg\lg n \right)$ so the extra storage for auxiliary directories is $\frac{3n}{\lceil \lg\lg n \rceil} + O(n^{\frac{1}{c}} \lg n \lg\lg n)$ which matches the statement of the theorem given c is at least 2. $\mathcal{QED}$

This shows that **select**() can be implemented in constant time under a wide bus model using asymptotically negligible storage.

Theorem 2.3 *A binary tree on n nodes can be represented in $2n + o(n)$ bits and support* **parent**, **leftchild** *and* **rightchild** *operations in constant time on an MBRAM. Similarly, a general tree on n nodes can be represented in $2n + o(n)$ bits and support the* **child**, **parent** *and* **degree** *operations in constant time on an MBRAM.*

Proof: Using Jacobson's rank and our select directory structures on the level order encodings from Sections 2.1.1 and 2.1.2, the result follows. $\mathcal{QED}$

While asymptotically small, the extra storage required by the rank and select directories is significant when considering trees of a size comparable to modern computer memories. For $n = 2^{16}$, the extra storage required by the auxiliary directories is slightly larger than the bit maps. However, the size of the directories can be reduced by storing less frequent samples in the directories and performing larger linear scans. These reductions are necessary in the next section.

The rank/select based operations on the level order encodings of binary and general trees do not appear to directly support the inclusion of fixed sized fields of different sizes in the leaves and internal nodes of the trees. Such fields are required in many applications, including the tries covered in the next section. In an application where both leaves and internal nodes require fields and these fields are approximately the same size, we can store the field values in a separate array indexed by the rank of the corresponding node. However, when the field sizes for the two classes of nodes are not equal, this method may waste too much storage to be practical. In order to solve this problem, we associate a separate bit vector, indexed by the ranks of the nodes, that distinguishes internal nodes from leaves using a one bit for a leaf and a zero bit for an internal node. Using the **rank**() function on this bit vector we can map any leaf to a number in the range $1..L$, where L is the number of leaves, and then store the leaf data in a separate array. **rank0**() can be used to perform the analogous task for internal nodes. This adds

$n + o(n)$ bits or about one bit per node to the total storage requirements.

2.3 Compact Tries

Now we consider the compact implementation of a trie. The primary differences between a general tree and a trie are:

- the addition of edge labels in the range $1 \ldots m$ such that no two edges from a node have the same label (this also places a bound of m on the degree of a node but this bound is not significant to us), and
- a new operation **triechild**(x, i) that returns the child of x with the label i (as opposed to **child** which returns the i'th child).

As with binary trees, we first determine the asymptotic number of bits needed to represent a trie.

Theorem 2.4 *The number of bits required to represent an order m trie on n nodes is at least, asymptotically in n,*

$$n\,(m \lg m - (m-1)\lg(m-1)) + O(\log n). \tag{2.3}$$

Proof: The number of order m tries on n nodes satisfies the recurrence relation:

$$\begin{aligned} C_0^{(m)} &= 1 \\ C_n^{(m)} &= \sum_{n_1+n_2+\ldots+n_m=n-1} C_{n_1}^{(m)} C_{n_2}^{(m)} \ldots C_{n_m}^{(m)}. \end{aligned}$$

The numbers $C_n^{(m)}$ are called the *Fuss-Catalan* numbers and can be shown to equal $\frac{1}{mn+1}\binom{mn+1}{n}$ (cf. [25]). For $m = 2$ the superscript is omitted and we obtain the Catalan numbers of Section 2.1.1. To determine the minimum number of bits needed to represent a trie, we need to compute $\lg C_n^{(m)}$. Convert the binomial to factorials, simplify some terms, and expand the logarithm to obtain:

$$\lg((mn)!) - \lg(n!) - \lg((mn-n)!) - \lg(mn-n+1). \tag{2.4}$$

Stirling's approximation to $\ln(x!)$ is $x\ln x - x - \frac{\ln x}{2} + \sigma + O\left(\frac{1}{x}\right)$[25]. Substituting this approximation into formula 2.4, dropping low order terms (σ and the order term), converting from ln to lg, and cancelling some terms we obtain:

$$mn\lg(mn) - \frac{\lg(mn)}{2} - n\lg n + \frac{\lg n}{2} - (m-1)n\lg((m-1)n) + \frac{\lg(mn-n)}{2} - \lg(mn-n+1).$$

In order to determine the number of bits needed asymptotically in n, we place any terms not at least linear in n in an order term. After expanding the logs and cancelling two $mn\lg n$ terms of opposite sign, we obtain:

$$n\,(m\lg m - (m-1)\lg(m-1)) + O(\log n).$$

QED

Note that binary trees are the same as 2-ary tries, so, for $m = 2$, this result confirms our previous goal of $2n$ bits per node for binary trees. To obtain the asymptotic number of bits required for general m, rewrite formula 2.3 as:

$$n\left(\lg m + \lg\left(\frac{m^{m-1}}{(m-1)^{m-1}}\right)\right) + O(\log n).$$

As $m \to \infty$ the second log term approaches $\lg e$ so the number of bits required to represent a trie is approximately $\lg m + \lg e$ bits per node where $\lg e \approx 1.44$. We will not actually attain this goal but will be satisfied with $\lg m + c$ bits per node in the trie provided c is small. As before we want to obtain traversal operations in constant time on an MBRAM. However, we will have to be satisfied with an expected constant cost instead of a deterministic one.

Our approach to representing a trie compactly builds on the previous structure for general trees and adds edge labels and hash tables for rapidly locating a labelled child. By using the **rank()** function, the fill ratio of the hash tables can be kept near 50% without adding greatly to the storage requirements. In addition to the above requirements, in a typical application each leaf of a trie is labelled with a data element so, as discussed in the previous section, we require one more ranked bit map to map leaves to data elements. This structure adds about one bit per node to the storage cost.

Recall from Section 2.1.2 that, in the general tree encoding, a node is represented by the position of the corresponding one in its parent node's label. This convention allows us to number the nodes during a level order traversal of the tree and obtain each node's number using the **rank()** function. We move the edge labels to their destination node and place the labels in a simple array indexed from 1 to n. The label for a node can then be obtained using **rank**. The remaining step is to provide a mapping from the node labels to the ordinal number of a child. We could simply order the children of a node according to their label value and use a binary search to obtain logarithmic time traversal operations. In order to obtain constant time operations we reserve 2 bits per child in each node of the trie and use them to store a hash table.

The $2k$ bits available for a hash table for k children will be split into two pieces: a bit map and a ranking directory for the bit map. Each child will be inserted into the hash table using a hash value based on its label and the hash table size. We do not concern ourselves with the exact hash functions used or the details of the collision resolution strategy. The hash table is stored in the bit map with a one bit representing a full slot and a zero bit representing an empty slot. The children of a node are sorted according to their final position in the hash table. Using the **rank()** function on a hash table position we can obtain the ordinal number of the child. Another use of **rank()** on the main tree representation obtains the node number of the child and its label. If we let t_k be the number of bits required to build a ranking directory on a k bit bitmap then the fill ratio of the hash table is

$\frac{k}{2k-t_k}$. By modifying the **rank** construction we can still obtain constant time but ensure that $t_k < \frac{k}{4}$. The modifications needed reduce the number of samples stored in the directories and use larger linear scans. With this change, the hash tables are at most $\frac{2}{3}$ full and for large k the fill ratio approaches $\frac{1}{2}$ because t_k is $o(k)$. This ensures we can search the hash table in constant expected time[23]. Because we reserve exactly 2 bits per child we can store all the hash tables in a separate bit map with each node's hash table starting at offset 2 **rank**(**child**(x, 1)). The **degree**() operation can be used to compute the size of the hash table and hence the correct hash function.

Theorem 2.5 *A static trie on an alphabet of size m with n nodes can be represented in $\lg m + 4 + o(1)$ bits per node and provide the* **parent** *operation in constant time and the* **triechild** *operation in constant expected time.*

Proof: The storage required for the underlying tree representation is $2n + o(n)$ bits. The node labels require $n \lg m$ bits and the hash tables require $2n$ bits. Summing up, the total storage used is $\lg m + 4 + o(1)$ bits per node. *QED*

While higher than the previously obtained optimum, it is within our goal of $\lg m + c$. The parent operation is unchanged from the general tree and so operates in constant time. The **triechild** operation requires a search through a hash table where each step in the search requires a constant number of rank and select operations. The expected number of steps is constant so the overall running time is constant.

It is worth noting that if m is very small relative to n, specifically $m \lg m \leq c \lg n$ for some small constant c, then we do not need the hash tables and can obtain constant time operations while saving two bits per node. For such m and n, we can scan the labels for all of a node's children in a constant number of word operations. For example, for each value i in $1..m$ we can have table, loc_i, of all bit

Name	#Index Points	#Nodes	Index Size (bytes)
Holmes	43745	95512	235726
XIII	924430	1728510	3931554
Bible	1202504	4187104	9175535

Table 2.1: Suffix Trie Sizes

patterns of length $\frac{\lg n}{2}$ that gives the location of any aligned occurrence of i in the bit pattern. Using these tables, we can search for a particular child using $2c$ table lookups. In practice, the application of some simple bitwise boolean and arithmetic operations can replace the table lookups. The packing of multiple values in a single word and then using word operations to perform parallel computations on the original values is called "word-size parallelism" and is further discussed by Brodnik[8].

The only comparably compact representation for a trie that we are aware of is the Bonsai structure of Darragh *et al.*[13] which is stated to require $\frac{5}{4}(6 + \lg m)$ bits per node. It appears, however, that the 6 hides some non-constant but slowly growing terms. The $\frac{5}{4}$ factor is based on an 80% full hash table. For large n the structure developed here will be significantly smaller than the Bonsai structure. However, the Bonsai structure allows a limited number of insertions, with a small probability of failure, and so a direct comparison is not really meaningful.

While we are not recommending the structure above for general text searching (our solution for that problem lies in the next chapter), we list the estimated index sizes for three of our test documents in Table 2.1. Using techniques similar to those developed in the next chapter, the trie representation here can also be used to develop a compact suffix tree representation for main memory.

2.4 Recursive Encoding: Binary Trees Revisited

The previous section introduced an asymptotically optimal encoding for binary trees that provides leftchild, rightchild and parent operations. In this section, we provide a slightly less space efficient encoding that provides the leftchild, rightchild and sub-tree size operations. When working with PAT trees, the sub-tree size is the size of the query result and so is a useful unit cost operation, particularly when the search result is large. The new representation is very similar to one also developed by Jacobson[27], although ours uses a more efficient prefix code to obtain a smaller representation.

The tree encoding represents each tree as a bit string

Header	Left Sub-tree Encoding	Right Sub-tree Encoding

where the header contains two fields:

- a single bit indicating which of the two children has fewer nodes with an arbitrary choice made in the case of a tie, and,
- a prefix coded integer indicating the size of the smaller child.

To represent an integer i, we concatenate the unary encoding of $\lfloor \lg(i+1) \rfloor$ with the binary encoding of $i+1$. This constructs a *prefix code* such that no code value is a prefix of any other code value and so we can, in a left to right scan of the data, determine when we have the complete encoding of an integer. The first few code values are shown in Table 2.2. In order to ensure that the operations of locating the encodings of the left and right sub-trees of a node can be efficiently implemented, each tree encoding is padded out to the length of the longest encoding of a tree of the same number of nodes.

0	1	6	00111
1	010	7	0001000
2	011	8	0001001
3	00100	9	0001010
4	00101	10	0001011
5	00110	11	0001100

Table 2.2: Integer Prefix Code

The integer prefix code requires $2\lceil\lg(i+2)\rceil - 1$ bits, so the size of the encoding of a tree on n vertices satisfies

$$\begin{aligned} B(0) &= 0 \\ B(1) &= 1 \\ B(n) &= \max_{i=0..\lfloor(n-1)/2\rfloor} B(i) + B(n-i-1) + 2\lceil\lg(i+2)\rceil . \end{aligned} \tag{2.5}$$

The initial values for the recurrence are based on the fact that we do not need to encode the structure of trees with zero or one nodes. The ability to solve this recurrence is the primary requirement when choosing a prefix code for use in this type of tree encoding. We will see that the closed form solution to formula 2.5 is of the form $B(n) = 3n - f(n)$ where $f(n)$ is $O(\lg n)$. Before proving this closed form, a few lemmas are needed. We use the notation $(x)_2$ to denote the base 2 representation of x and $v_2(x)$ to denote the number of ones in this representation.

Lemma 2.1 *For $n > 0$ and $0 \le i \le n$, $v_2(i+1) + v_2(n-i) = v_2(n+1) + k$ where k is the number of carries that occur when adding $i+1$ and $n-i$ in base two.*

Proof: Each carry that occurs during the addition of $(i+1)_2$ and $(n-i)_2$ requires two one bits and produces another. If k carries occur, we have $v_2(i+1) + v_2(n-i) + k$ ones available and we require exactly one bit for each one in $(n+1)_2$ and two for each carry so

$$v_2(i+1)+v_2(n-i)+k=v_2(n+1)+2k$$

or

$$v_2(i+1)+v_2(n-i)=v_2(n+1)+k.$$

$\mathcal{QED}$

Lemma 2.2 *If n is even, $min_{i=0..n/2-1}v_2(i+1)+v_2(n-i)+\lfloor\lg(n-i)\rfloor$ occurs at $i=0$ and so is $1+v_2(n)+\lfloor\lg(n)\rfloor$.*

Proof: This result is a simple corollary of the previous lemma. Because n is even there are no carries when adding 1 and n so $i=0$ minimizes the first two terms. For $n=2^k-2$, the lg term is constant over the range of i values so $i=0$ still the produces minimum total. For other values of n, the lg term takes on one of two values over the range of i: $\lfloor\lg(n)\rfloor$ and $\lfloor\lg(n)\rfloor-1$. However, those values of i resulting in the smaller value necessarily require a carry when adding $(i+1)_2$ and $(n-i)_2$ so the sum of the first two terms increases by at least one and this increase offsets the saving in the lg term. $\mathcal{QED}$

Lemma 2.3 *If n is odd, $\min_{i=0..(n-1)/2}v_2(i+1)+v_2(n-i)+\lfloor\lg(n-i)\rfloor+[i \text{ is odd}]$ occurs at $i=2^k-1$ where*

$$k=\begin{cases} j-1 \text{ if } n=2^j-1 \\ \text{the number of trailing 1 bits in the binary representation of } n \text{ otherwise}\end{cases}$$

and so is $2+v_2(n)+\lfloor\lg n\rfloor-k$.

Proof: First consider $n=2^j-1$. At least one carry must occur when adding $(i+1)_2$ and $(n-i)_2$ because representing $n+1$ requires more bits than either of these two terms. Setting $i=2^{j-1}-1$ results in exactly one carry and so minimizes the first two terms. $\lfloor\lg(n-i)\rfloor$ is constant over the range of i values so

$i = 2^{j-1} - 1$ optimizes the first three terms. Selecting an even value of i requires two carries when adding $(i+1)_2$ and $(n-i)_2$, one in the least significant bit and another in the most significant bit and so cannot lessen the total. Setting $i = 2^{j-1} - 1$ in the sum and using the relationships $v_2(2^j) = 1$, $v_2(n+1) = v_2(n) - k + 1$ and $\lfloor \lg(n-i) \rfloor = \lfloor \lg n \rfloor$ produces the final result.

Now consider the more general case, setting $i = 2^k - 1$ results in no carries and so minimizes first two terms. As in the proof of Lemma 2.2 the lg term takes on two successive values but choosing i sufficiently large that the smaller of the two values occurs necessarily results in a carry that offsets the saving in the lg term. Finally, as with the first case, making i even results in a carry in the low order bit that offsets any savings in the last term. Setting $i = 2^k - 1$ in the sum and using the same relationships that occurred in the first case yields the final total. $\mathcal{QED}$

We are now able to state and prove the closed form solution to formula 2.5.

Theorem 2.6 *$B(n) = 3n + 2 - 2\lfloor \lg(n+1) \rfloor - 2v_2(n+1) - [n \text{ is odd}]$, where v_2 denotes the number of ones in the binary representation of its argument.*

Proof: Recall the recurrence relation for B:

$$\begin{aligned} B(0) &= 0 \\ B(1) &= 1 \\ B(n) &= \max_{i=0..\lfloor (n-1)/2 \rfloor} B(i) + B(n-i-1) + 2\lceil \lg(i+2) \rceil . \end{aligned}$$

The proof proceeds by induction. The base cases, $B(0)$ and $B(1)$, satisfy the equation for B. Assuming the formula is correct for $0..n-1$, the recurrence relation gives:

$$\begin{aligned} B(n) = \max_{i=0..\lfloor \frac{n-1}{2} \rfloor} \; & 3i + 2 - 2\lfloor \lg(i+1) \rfloor - 2v_2(i+1) - [i \text{ is odd}] + 3(n-i-1) \\ & +2 - 2\lfloor \lg(n-i) \rfloor - 2v_2(n-i) - [n-i-1 \text{ is odd}] \\ & +2\lceil \lg(i+2) \rceil . \end{aligned}$$

Simplifying and using the fact that $\lceil \lg(i+2) \rceil = \lfloor \lg(i+1) \rfloor + 1$, obtain

$$\begin{aligned} B(n) = \quad & 3n + 3 + \max_{i=0..\lfloor \frac{n-1}{2} \rfloor} -2v_2(i+1) - 2v_2(n-i) \\ & -2\lfloor \lg(n-i) \rfloor - [i \text{ is odd}] - [n-i-1 \text{ is odd}]. \end{aligned} \tag{2.6}$$

Consider even and odd values of n:

- If n is even, then if i is odd, $n-i-1$ is even and vice-versa so the last two terms in formula 2.6 always sum to minus one. Bring the -2 outside the max and replace the max with a min and simplify to obtain:

  $$3n + 2 - 2 \min_{i=0..\frac{n}{2}-1} v_2(i+1) + v_2(n-i) + \lfloor \lg(n-i) \rfloor .$$

  Using Lemma 2.2, this simplifies to

  $$3n - 2v_2(n) - 2\lfloor \lg n \rfloor .$$

  In this case, the closed form gives

  $$\begin{aligned} B(n) &= 3n + 2 - 2v_2(n+1) - 2\lfloor \lg(n+1) \rfloor - [n \text{ is odd}] \\ &= 3n + 2 - 2(v_2(n) + 1) - 2\lfloor \lg n \rfloor \\ &= 3n - 2v_2(n) - 2\lfloor \lg n \rfloor . \end{aligned}$$

  so the equations are equal.

- If n is odd, then $n-i-1$ is odd iff i is odd so the last two terms of formula 2.6 become $2[i \text{ is odd}]$. As before we bring the -2 outside the max and replace the max with a min to correct the sign change to obtain:

  $$B(n) = 3n + 3 - 2 \min_{i=0..\frac{n-1}{2}} v_2(i+1) + v_2(n-i) + \lfloor lg(n-i) \rfloor + [i \text{ is odd}]$$

  which by Lemma 2.3 gives

  $$B(n) = 3n + 3 - 2(2 + v_2(n) + \lfloor \lg n \rfloor - k)$$

where k is as defined in the lemma. In both of the cases of Lemma 2.3, we obtain $v_2(n) + \lfloor \lg n \rfloor = v_2(n+1) + \lfloor \lg(n+1) \rfloor + k - 1$. Using this formula, we obtain:

$$\begin{aligned} B(n) &= 3n + 3 - 2\,(2 + v_2(n+1) + \lfloor \lg(n+1) \rfloor + k - 1 - k) \\ &= 3n + 1 - 2v_2(n+1) - 2\lfloor \lg(n+1) \rfloor . \end{aligned}$$

Which is equal to the closed form, given that n is odd.

QED

n	$B(n)$	n	$B(n)$	n	$B(n)$	n	$B(n)$
0	0	8	16	16	38	24	60
1	0	9	18	17	40	25	62
2	2	10	20	18	42	26	64
3	4	11	24	19	46	27	68
4	6	12	26	20	48	28	70
5	8	13	28	21	50	29	72
6	10	14	30	22	52	30	74
7	14	15	36	23	58	31	82

Table 2.3: $B(n)$

From the closed form equation, it is clear that $B(n) < 3n$ so the total storage requirement for the binary tree information is less than three bits per node. Table 2.3 shows the value of B for small n. Using this representation, the tree structure for the tree in Figure 2.1 on page 25 is represented by the bit string 10010101101010101011. Figure 2.5 shows this tree with each sub-tree labelled with its description.

The simple formula for $B(n)$ allows efficient implementation of the operations of fetching the left and right children of a node. The left child is found immediately

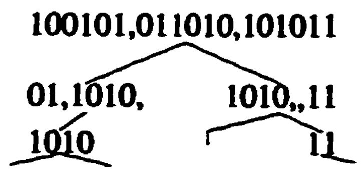

Figure 2.5: Tree Encoding

```
rightchild(node,size) =
   small_child = read bit at position node
   child_size = read prefix code at position node + 1
   children = node+2*ceil(lg(child_size+2))
   if (small_child = '1') then
      return (children+B(child_size)), size-child_size-1)
   else return (children+B(size-child_size-1),child_size)
end
```

Figure 2.6: Pseudo-code for Rightchild

following the prefix code of the integer giving the size of the smaller tree and the right child can be found immediately after the description of the left child whose size can be computed based on the number of nodes in the left sub-tree which can in turn be computed given we know which sub-tree is smaller, the size of the smaller sub-tree and the size of the overall tree. The pseudo-code for the rightchild operation is found in Figure 2.6. The pseudo-code for leftchild does not add $B(leftchildsize)$ to "children" and reverses the cases of the if statement. Each of these operations require and return both the location of the node in the bit stream and the size of the sub-tree rooted at the node.

A natural question to ask is "can we change this encoding to obtain $2n$ bits per node?" In his thesis, Jacobson investigates the use of optimal prefix codes for this application and determines that while it is theoretically possible to obtain a bound of less than 2.5 bits per node using more compact prefix codes, this approach has a lower bound of about 2.3 bits per node. We also want to mention that some slight improvements can be made by increasing the number of base cases in formula 2.5. If, for example, we add the conditions $B(2) = 1$, $B(3) = 3$ and $B(4) = 4$ (using $B(n) = \lceil \lg C_n \rceil$) to the recurrence then $B(n)$ is reduced by $\lfloor \frac{n+1}{4} \rfloor$. This change reduces the asymptotic requirement to 2.75 bits per node. Further increases in the number of base cases appear to reduce the requirements further although we have yet to determine a closed form for these cases.

In this chapter we have provided some extensions and improvements to the results presented by Jacobson[27][26] that will be useful in our PAT representation as well as other tree based structures. We have also provided further demonstration of the usefulness of rank and select by using them to construct a representation for static tries requiring $\lg m + 4 + o(1)$ bits per node.

Chapter 3

Static Text on Primary Storage

In this chapter, we combine the compact tree representations of Chapter 2 with compact representations of the skip values and suffix offsets to produce a representation for the PAT tree that is little larger than the representation of a suffix array. Finally, we present some empirical results showing the effectiveness of the new structure.

The information stored in the PAT tree can be broken into three categories:

- the tree structure,
- the skip values,
- the suffix offsets in the leaves.

By efficiently storing each class of information, Compact PAT Trees (CPTs) match the storage efficiency of other suffix based search structures while retaining the functionality of PAT trees.

3.1 Choosing a Tree Representation

In order to implement the tree operations for PAT trees, the encoding of the tree structure must provide the following functionality:

- efficient selection of the left and right children of a node,
- support for the inclusion of constant size fields for each internal node, the skip, and another constant size field for each leaf, the suffix offset. Given a node or leaf, we must be able to efficiently determine the field values.

In each case, we require that the operations be performed in a constant number of operations on $\lg n$ size objects so they will operate in constant time on the MBRAM model. In addition, the following three operations are useful in some cases but are not critical to our work:

- given a node, locate its parent,
- given a node, efficiently retrieve the suffix offset field information from some leaf descended from the node,
- given a node, determine the size of the sub-tree rooted at the node.

The parent operation is not used during the PAT tree search traversal but can be used to conserve memory during update operations. The efficient retrieval of the suffix information from a sub-tree is used during the final step of the search procedure and must be performed efficiently. If necessary, we can afford to traverse the tree downward until we hit a leaf and then retrieve its suffix information. Given the almost sure expected logarithmic depth of the PAT tree, the cost of the downward traversal will not effect the asymptotic cost of a search. Determining the sub-tree size is useful because it will be a good approximation to the number of matches to a query.

Chapter 2 presented two compact representations for binary trees that can be used to encode the structure of the PAT tree and still support the required operations. Before deciding between these representations, we observe that, because the PAT tree is a strictly binary tree, we need only represent the structure of the tree made up of the internal nodes. The positions of the leaves, the suffix offsets, of the PAT tree are implied by the positions of the leaves and degree one nodes of the tree of internal nodes. This change allows us to reduce the size of the tree we need to encode from $2n - 1$ nodes to $n - 1$ nodes, where n is the number of index points.

As discussed in the previous chapter, the rank/select representation for binary trees of Section 2.1.1 does not directly support the requirement for fixed sized fields in the leaves of the tree. However, we observe that in the level order encoding of the tree of internal nodes, the ones correspond exactly to the internal nodes and the zeroes correspond to leaves of the original PAT tree (see Figure 3.1). Because of this property, we can store the node labels and suffix offsets in

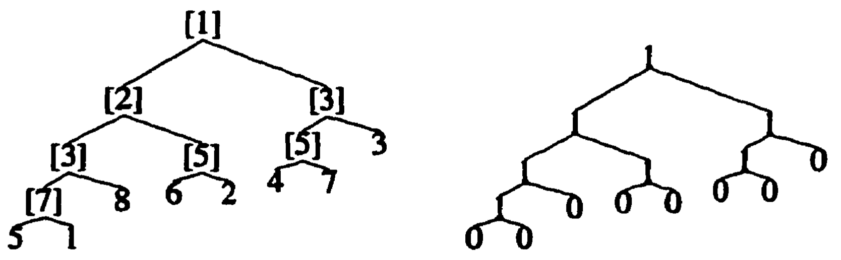

Figure 3.1: PAT tree and Labelled Extended Tree of Internal Nodes

two arrays referenced by **rank()** and **rank0()** respectively.

If we instead use the recursive encoding for binary trees from Section 2.4 to represent the structure of a PAT tree, the node labels can be stored either in the bit stream for the tree encoding or in a separate array created during an in-order traversal of the tree. During the downward traversal the index for the current node can be computed by tracking the number of nodes in each left sub-tree that is skipped. A similar method allows us to store the suffix offsets at the leaves in an array and track our current position in that array. Retrieval of the test suffix pointer is very efficient because we can simply retrieve the first entry of the current sub-array.

The trade-off to be considered in determining which representation to choose is one of size, an extra bit per node, versus functionality, the ability to quickly determine the answer size. For the static text case in primary storage the extra functionality is well worth the extra bit in the representations so we use the recursive encoding. If used to represent other PATRICIA based structures that require the parent operation, such as the "blind trie" used in the SB-tree of Ferragina and Grossi[18], the rank/select encoding would be more appropriate. In deciding whether to embed the skip values and suffix offsets in the bit sequence for the tree encoding or place them in separate arrays, we choose to place the skip values in the tree encoding and use a separate array for the suffix offsets. This layout improves the access locality of the searching procedure and allows aligned accesses into the suffix offset array which should improve performance.

3.2 Storing the Skip

Compressing the skip information requires an understanding of the distribution of the skip values. For the purpose of analysing the skips, temporarily assume the suffixes are strings of independent uniformly sampled bits with 0 and 1 having equal probability. Consider an internal node with k leaves in its sub-tree, then the probability that the skip value of the node is greater than l is the same as the probability that k random bit strings match in their first $l+1$ bits. This value is easily seen to be $2^{-(l+1)(k-1)}$. From this value, we see that we can expect the majority of the skip values to be zero and that the likelihood of higher values decreases geometrically. Figure 3.2 shows the skip distributions for the four sample documents and illustrates the rapid decrease in the likelihood of large skip

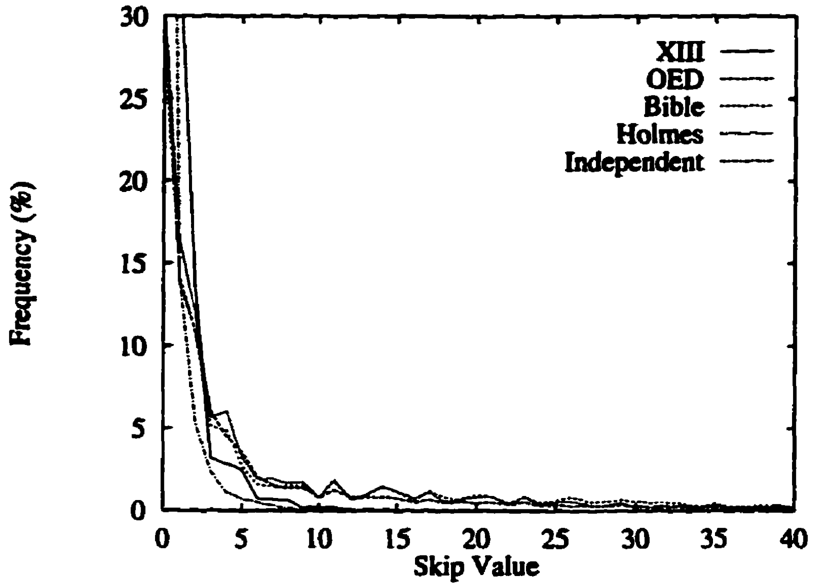

Figure 3.2: Frequency of Skip Values in the Sample Documents

values when a compact alphabet code is used. The use of a compact code for the alphabet is discussed later in this section. The curve labelled "Independent" in Figure 3.2 shows the results for a PATRICIA tree on independent pseudo-random keys generated using a uniform model for the characters.

The low likelihood of large skip values leads to a simple method of compactly encoding the skip values. We reserve a small fixed number of bits to hold the skip value for each internal node and introduce a strategy to resolve problems caused by skip values that overflow this field. We handle overflow by inserting a new node and a leaf into the tree and distributing the skip bits from the original node across the skip fields of the new and the original node. Figure 3.3 illustrates an overflow

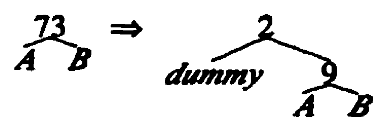

Figure 3.3: Overflow Nodes

situation where 5 bits have been reserved for the skip information. The actual skip value of 73 is encoded as $(29)_{32}$ and stored in the skip fields of the original node and the new *overflow node.* The dummy leaf node must have some special key value that allows it to be easily recognized (typically all 0s or all 1s). If needed, multiple overflow nodes and leaves can be inserted for extremely large skip values.

When traversing the tree, simply checking for a single leaf with the dummy value is sufficient to determine if the skip should be checked or the bits concatenated to obtain the true skip value. The use of this overflow handling mechanism has one slight drawback in that the sub-tree size is no longer the exact size of the answer. However, the sub-tree size is still an upper bound on the size of the answer and in practice a good estimate of the size.

How many bits should be reserved for the skip field? If we let N be the set of internal nodes in the PAT tree, then the expected number of overflow nodes when using a k bit skip field is, by the previous approximation,

$$\sum_{m \in N} \sum_{t=1}^{\frac{\lg n}{k}} \frac{1}{2^{(l_m-1)(2^{tk}+1)}}$$

where l_m is the number of leaves below node m. The inner sum is dominated by the value at $t = 1$ so we will ignore the other values. Let r be the storage required for representing one node in the tree, $r = c + k + \lg n$ where $c = 3$ if we use the recursive encoding. The storage requirement when using k skip bits is then approximately

$$r\left(n - 1 + \sum_{m \in N} \frac{1}{2^{(l_m-1)(2^k+1)}}\right) + \lg n.$$

The value of k we are interested in is the smallest value such that the expected storage at k bits is less than that at $k + 1$ bits. So we want the smallest k that satisfies:

$$r\left(n - 1 + \sum_{m \in N} \frac{1}{2^{(l_m-1)(2^k+1)}}\right) + \lg n < (r+1)\left(n - 1 + \sum_{m \in N} \frac{1}{2^{(l_m-1)(2^{k+1}+1)}}\right) + \lg n.$$

Expand the products and simplify to obtain:

$$\sum_{m \in N} \frac{r}{2^{(l_m-1)(2^k+1)}} - \frac{r+1}{2^{(l_m-1)(2^{k+1}+1)}} < n-1.$$

Further simplify the sum:

$$\sum_{m \in N} \frac{1}{2^{(l_m-1)(2^k+1)}} \left(r - \frac{r+1}{2^{(l_m-1)2^k}} \right) < n-1.$$

The value $\frac{r+1}{2^{(l_m-1)2^k}}$ is much smaller than r and so will not be significant in the sum. This observation allows us to approximate the inequality by

$$r \sum_{m \in N} \frac{1}{2^{(l_m-1)(2^k+1)}} < n-1. \tag{3.1}$$

The exact value of the sum depends on the shape of the tree, and k, but it is clearly less than $\frac{n-1}{2^{2^k+1}}$ because l_m is at least two for every node m. Now consider a perfectly balanced binary tree on an even number of leaves. There are $\frac{n}{2}$ nodes with exactly two leaves below them so the sum, for this tree, will be at least $\frac{n}{2^{2^k+2}}$. The sum for the worst case tree will be somewhere between these two bounds. Entering either of these bounds in equation 3.1 gives an equation of the form (here we use the upper bound):

$$r\frac{n-1}{2^{2^k+1}} < n-1$$

which is true if $r < 2^{2^k+1}$. Insert the definition of r to obtain $c + k + \lg n < 2^{2^k+1}$. Take logarithms twice and ignore small values (assume c and k are small) to derive that $k \approx \lg\lg\lg n$. The optimal k derived using the lower bound gives the same result so the optimal value of k for the worst case tree is about $\lg\lg\lg n$. Note that for some tree shapes the optimal value may be much smaller.

Theorem 3.1 *Under the assumption of independent suffixes drawn using a symmetric model, the expected size of the Compact PAT Tree can be made less than* $3\frac{1}{2} + \lg n + \lg\lg\lg n + O\left(\frac{\lg\lg\lg n}{\lg n}\right)$ *bits per index point. We achieve this size by setting the skip field size to* $\lg\lg\lg n$.

Proof: Under the given assumptions and using the skip field size above, the expected number of internal nodes is,

$$n - 1 + \sum_{m \in N} \sum_{t=1}^{\frac{\lg n}{\lg\lg\lg n}} \frac{1}{2^{(l_m - 1)(2^{t \lg\lg\lg n} + 1)}}.$$

Set l_m to 2 for all nodes and simplify the outer sum to obtain the bound:

$$n - 1 + (n-1) \sum_{t=1}^{\frac{\lg n}{\lg\lg\lg n}} \frac{1}{2^{2^{t \lg\lg\lg n} + 1}}.$$

Multiply this bound by the node storage cost and add $\lg n$ to account for the fact that there is one more leaf than internal nodes. After simplifying, obtain:

$$(n-1)\left(1 + \frac{1}{2} \sum_{t=1}^{\frac{\lg n}{\lg\lg\lg n}} \frac{1}{2^{(\lg\lg n)^t}}\right)(3 + \lg\lg\lg n + \lg n) + \lg n.$$

Divide by n to obtain the per index point cost and ignore some asymptotically insignificant terms to get:

$$\left(1 + \frac{1}{2} \sum_{t=1}^{\frac{\lg n}{\lg\lg\lg n}} \frac{1}{2^{(\lg\lg n)^t}}\right)(3 + \lg\lg\lg n + \lg n).$$

Expand the product and the sum to obtain:

$$3 + \lg\lg\lg n + \lg n + \frac{3 + \lg\lg\lg n}{2}\left(\frac{1}{\lg n} + \frac{1}{2^{(\lg\lg n)^2}} + \ldots\right) + \frac{\lg n}{2}\left(\frac{1}{\lg n} + \frac{1}{2^{(\lg\lg n)^2}} + \ldots\right).$$

Finally expand the last term to obtain:

$$3\frac{1}{2} + \lg\lg\lg n + \lg n + \frac{3 + \lg\lg\lg n}{2\lg n} + \frac{3 + \lg\lg\lg n + \lg n}{2}\left(\frac{1}{2^{(\lg\lg n)^2}} + \frac{1}{2^{(\lg\lg n)^3}} \ldots\right).$$

which satisfies the statement of the theorem. *QED*

It is useful to note that the final order term in Theorem 3.1 is not just asymptotically negligible but that for reasonable values of n, for example $n = 2^{16}$, it is significantly less than one.

There are two approximations in the argument above that deserve serious consideration. The first is the assumption that the bit strings in the suffixes are independent. This is clearly false as the strings are, potentially overlapping, sub-strings of a single string. However this approximation does not seem inappropriate because the underlying string is large and for all but the few nodes near the root we expect the suffixes below a node to be sparse in the underlying string. This position is further supported by work by Szpankowski showing that the expected depth of a trie does not change when moving from independent to dependent bit strings[34]. The second, more serious, problem above is the assumption that the binary string is generated by a uniform symmetric random process. This is not a good model of written text or other large documents. Consider, for example, English text coded in ASCII, in which the high order bit of each byte will be zero. In addition, the codes 0..31 are unlikely to occur in the text. While this is a real weakness in the approach, we can take some steps to alleviate it:

- use of a compact character code. Instead of ASCII, we use a character code where all bits are active for our search engines. We are considering the use of a data compression model incorporating digrams for a future version, but it is unlikely the performance will be adequate given the large amount of data handled during index construction. It is not clear that the cost of performing even the translation to the compact code during the indexing phase is worth the slight reduction in the number of overflow nodes. We use this translation in our test system because it is easily incorporated in the other translations (case conversion, merging multiple spaces) that are required during searching and index construction.
- increasing the skip field size. By adding an extra 2 or 3 bits to the skip size we can dramatically reduce the number of overflow nodes present in the tree.

The final pseudo-code for searching a compact PAT tree is shown below:

```
size = n              // size of the current sub-tree
index = 1             // index of first leaf of the current
                      // sub-tree in the suffix array
test_bit = 0          // accumulated skip values
bit_pattern = encode(pattern)
while (size > 0)
   save = <size, index>
   if (size > 1)
      smaller = read one bit
      sizeofsmaller = read prefix code
      skip = read k bit integer
      // loop to handle overflow nodes
      while(smaller = '0' and sizeofsmaller = 0 and
            suffixes[index]=dummy) do
         skip = skip*2^k
         size = size - 1
         index = index + 1
         if (size > 1)
            smaller = read one bit
            sizeofsmaller = read prefix code
         else
            smaller = sizeofsmaller = 0
         endif
         skip = skip + read k bit integer
      end
      if (smaller = '1')
         leftsize = sizeofsmaller
         rightsize = size - sizeofsmaller - 1
      else
         rightsize = sizeofsmaller
         leftsize = size - sizeofsmaller - 1
      endif
   else
      leftsize = rightsize = 0
```

```
            skip = read k bit integer
         endif
         test_bit = test_bit + skip + 1
         if test_bit  > |bit_pattern|
            <size,index> = save    // restore values
            exit the loop
         endif
         if bit_pattern[testbit] = '1'  then
            skip B(leftsize) + leftsize*k bits
            size = rightsize
            index = index + leftsize+1
         else
            size = leftsize
         endif
      end
      // skip any dummy nodes before doing the test
      while (suffixes[index] = dummy) do
         index = index + 1
         size = size - 1;
      end
      // compare the first suffix against the pattern
      if (pattern = suffixes[index]) return index,size
      else return NOTFOUND
      endif
```

The special handling of trees with zero or one internal nodes in the pseudo-code occurs because the structure of such small trees does not need encoding so only the skip value is stored. We typically use a skip field size of 5 or 6 depending on the document size. Even these larger sizes result in a very small index requiring about 10 bits per index point to represent the trie. For storing the skip values, we simply add a third constant size field to the tree header and modify $B(n)$ appropriately by adding n times the size of the skip field.

3.3 Suffix Offsets

The suffix offsets take up the bulk of the storage used by the CPT and other suffix based structures. While the suffix offsets do not easily admit compression, they can be stored much more compactly if we are willing to make some sacrifices on performance. To achieve this storage reduction, we use a technique also used in Shang's PaTries[44]. If l low order bits in the suffix offsets are omitted from the CPT structure, nl bits are saved in the final index. In order to perform searching using these truncated offsets, a scan through all suffixes starting in a range of 2^l characters must be made each time we require an exact suffix offset. Because the bits tested along the root-leaf path uniquely determine the suffix, the exact suffix offset can always be determined by running each suffix in the range through the PAT tree and selecting only the one that ends at the correct leaf. Alternatively, a simple string search for the query can be made at all the index points in the range and any matches reported. The string searching method has the advantage of only loading text segments once but special care must be taken when handling multiple matches in a block of 2^l characters. If we let H be the height of the CPT, these changes incur an additive $2^l H$ cost in the searching time and a $2^l H$ multiplicative factor on the conversion of a node to its list of leaf offsets because each suffix tested has to be traced through the CPT to a leaf. More importantly, proximity based queries such as "find word A within 50 characters of word B" cannot be answered without referring to the text because the suffix offset values may not have enough precision to determine if two matches are sufficiently close to each other. It is worth noting that suffix arrays cannot use truncated suffix offsets because they require the exact offset values to guide the searching procedure. The ability of the CPT to operate with inexact suffix offsets also allows its efficient use on text files compressed on a block by block basis without requiring complicated address translation mechanisms.

3.4 Empirical Results

Text(size)	Skip Size	#Overflow Nodes	Index Size (kbytes)
Holmes	4	5486	151
(233k)	5	1580	144
	6	242	145
XIII	2	123127	3197
(903k)	3	40509	3063
	4	19250	3111
Bible	4	264903	5373
(5.3m)	5	127150	5032
	6	48407	4887
	7	19377	4923

Table 3.1: Index Sizes for Sample Documents

Table 3.1 shows the number of overflow nodes and the resulting index sizes for three of the sample documents (the OED is too large for construction of an index in primary storage on the machines available for these experiments). From this table we see that the optimal skip sizes are 5 for Holmes, 3 for XIII, and 6 for the Bible. Notice also that the size of the final index for non-optimal values is still close to the optimal size. These sizes are based on full suffix pointers. If we allow the truncation of 8 bits then the index sizes for the examples will drop by about 45k, 960k, and 1250k bytes respectively.

3.5 Comparison to Other Structures

The storage cost of this structure is significantly less than that of previous representations for PAT trees and suffix trees with the exception of suffix arrays.

Suffix arrays offer comparable performance in main memory for straight searching. However, they incur a logarithmic performance penalty when simulating suffix tree operations such as those used for regular expression matching[4]. The new structure also has the advantage of quickly determining an approximate answer size.

Chapter 4

Static Text on Secondary Storage

In this chapter, we adapt the Compact Pat Tree from the preceding chapter for use on secondary storage by partitioning the tree into disk block sized pieces. We first discuss the characteristics of secondary storage as they affect our design and then discuss some existing tree partitioning algorithms. Next we present a new optimal tree partitioning algorithm that is more appropriate to our application. Finally, we give some empirical results demonstrating the effectiveness of the partitioned Compact Pat Tree.

Searching methods for large text databases must be concerned with more than asymptotic time requirements; storage requirements and the number of secondary storage accesses are also critical. If the index requires k bytes per index point in the text, character indices will be k times the size of the document while word indices will be about $\frac{k}{5}$ times the size of document. For large documents the storage cost quickly becomes prohibitive as k gets large. Similarly, while the asymptotic operation count of the algorithm is important, the number of accesses to and the amount of storage transferred from secondary storage are likely to have a far greater effect on the performance, and even the feasibility, of the index.

The time taken to access secondary storage can be broken into two components:

- the overhead time necessary to initiate and terminate a read or write operation. We refer to this time as the *seek time* of the device but it also includes other factors such as rotational latency. In most of this thesis we assume the seek time of the device is constant because we are not interested in optimizing the placement of data on the physical device.
- the time taken to transfer data to or from the device, referred to as the *transfer time* of the device. This component of the time is dependent on the amount of data transferred.

We assume the secondary storage operates on a *block* basis where each read or write operation transfers an integral number of contiguous blocks. We also assume the *block size*, which we label P, is fixed by the physical device and software drivers and is given to us as a parameter of the problem. For the moment, we ignore the transfer time because for current magnetic media the transfer time of one or two blocks is small when compared to the seek time. Later, when searching data stored on CD-ROM, we explicitly consider the transfer time. For our empirical testing we use block sizes of 1k, 2k, 4k and 8k bytes.

The first portion of this thesis dealt with controlling the storage requirements of PAT trees and, as a side-effect, reduced the amount of data we will need to transfer from secondary storage. In the second portion of this thesis we concentrate on controlling the number of accesses to secondary storage during searches and updates.

4.1 Partitioned Compact PAT Trees

In order to control the number of accesses to secondary storage required during CPT operations, we *partition* the tree into connected components each of which fits in a disk block. We call each component a *page* because of the similarity of this problem to the problem of efficiently laying out a tree or other data structure in a paged virtual memory system[21]. If the disk block size is such that it can hold two internal nodes then the PAT tree of Figure 1.6 could be partitioned as shown in Figure 4.1. In this case we need to perform three accesses to secondary

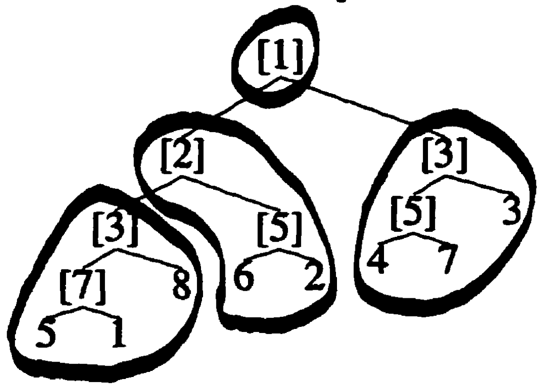

Figure 4.1: Tree Partition

storage to reach leaf 1, 5 or 8 from the root. The alternative partitioning in Figure 4.2 can reach any leaf in two accesses and so might be preferred.

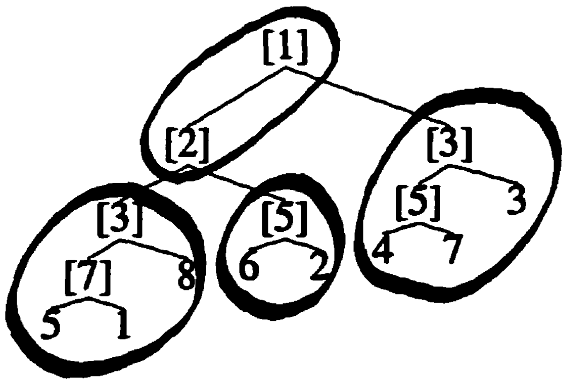

Figure 4.2: Alternative Tree Partition

Two possible criteria for choosing one partitioning over others are:

- the number of pages accessed when traversing from the root to a leaf, averaged over all the leaves, and
- the maximum number of pages accessed when traversing from the root to any leaf.

We call page partitionings that minimize these measures *average case optimal* and *worst case optimal* respectively. Let c_i be the number of pages accessed to reach the i'th leaf (under some ordering of the leaves). Then these partitionings minimize $\sum_i c_i$ and $\max c_i$ respectively. Implicit in these measures is the assumption that we consider all leaves equally important. Lukes[32] and Gil and Itai[21] consider more general cases where nodes and edges can have weights associated with them.

The partitionings considered here are restricted such that each page holds a connected portion of the tree. Gil and Itai use the term *convex* to describe such partitionings and show that loosening this restriction does not allow for better average case partitioning[21]. Because of this restriction, each page will itself be a tree and can be stored using the CPT structure from Chapter 2. The only change required to the CPT structure for storing the pages is that the leaf data may now point to either a suffix in the text or a sub-tree page so an extra bit is required to distinguish these two cases. We let the value p denote the number of internal nodes in the largest sub-tree we can place in a block. Using the representation from Chapter 3, $p \approx \frac{P-\lg n}{\lg n+\lg\lg\lg n+4}$. The restriction to connected sub-trees allows us to refer to the root of the sub-tree in a page as the root of the page. In addition we will refer to the page containing the sibling node of a page's root as the page's sibling. Note that in some cases a page's root and its sibling may be the same page (consider the rightmost internal node of Figure 4.2).

Lukes[32] presents a dynamic programming method for finding an average case optimal partitioning in $O(np^2)$ time. A related method for finding a worst case optimal partitioning in $O(np)$ time is reported in Carlisle *et al.*[9]. Unfortunately both of these methods require np words of storage to compute the partitioning and so are not practical for trees of the size we are considering. Gil and Itai[21] develop a similar dynamic programming method for the average case that operates in much less memory. However, their algorithm performs multiple passes over the tree and so is unlikely to be efficient enough for our purposes. Additionally, these dynamic programming methods do not adapt well to the dynamic trees needed in the next chapter. Carlisle *et al.*[9] also discuss a top down greedy heuristic that is conceptually simple and works well on some classes of trees but can require $\Theta(\log n)$ extra page accesses on average to reach any leaf.

In the remainder of this chapter we present a new bottom up greedy algorithm for constructing a worst case optimal partitioning of a binary tree and demonstrate its use on the CPT.

4.2 Partitioning Algorithm

Define the *page height* of a node in a partitioned tree as the maximum number of pages that need to be read when traversing from the node to any leaf in its sub-tree and the page height of a page as the page height of its root. In each case we include the current page in the page height count. For any given assignment of nodes to pages, also define the *local page size* of a node as the number of descendents of that node that are on the same page as the node, plus one for the original node. The page height and local page size of a node may be defined for a partial partitioning provided the node and all of its descendents have been placed on pages.

We present a partitioning algorithm that starts by assigning each leaf its own page and a page height of one. Working upward, we apply the rule in Figure 4.3 at each node.

```
if both children have the same page height
    if the sum of the local page sizes of the children is less than p,
        merge the pages of the children and add the node
        set the page height of the node to that of the children
    else
        close off the pages of the children
        create a new page for the current node
        set the page height of the node to that of the children plus one
else
    close off the page of the child with the lesser height
    if the local page size of the remaining child is less than p,
        add the node to the child's page
        set the page height of the node to match the child
    else
        close off the page of the remaining child
        create a new page for the node
        set the page height of the node to that of the child plus one
```

Figure 4.3: Tree Partitioning Rules

Theorem 4.1 *A worst case optimal convex partitioning of a binary tree can be computed in linear time, irrespective of the page size.*

Proof: Using induction on the tree height, we show that the rule in Figure 4.3 produces a worst case optimal partitioning of the tree such that no other optimal partitioning has a smaller root page and moreover that this holds for each sub-tree. The basis case, $k = 1$, consists of a tree with a single node and so is trivial. Assume the statement for $1..k-1$ and then consider the root of a tree of height k. There are several possible cases:

1. The root has only one child. In which case either the root fits on the topmost page of the child or it does not.

   - Root fits (the local page size of the child is less than p): Place the root in the topmost page. Any partitioning of smaller page height or top most page must contain a partitioning for the child of larger page height that violates the induction hypothesis for $k-1$.

   - Root does not fit (the local page size of the child is p): Create a new page for the root. Clearly there cannot be a partitioning with fewer than one vertex in the topmost page so any violation must be on the page height constraint. The existence of partitioning of lesser page height would imply a partitioning at height $k-1$ with room for the new root but the partitioning of the height $k-1$ sub-tree was completely full and also had smallest topmost page amongst all optimal partitioning so this situation cannot occur.

2. Next consider the case where the root has two children that differ in page height. By the rules above, the child of lesser page height is closed off. The root is placed in the topmost page of the other child if at all possible, and on a new page if not. There are two cases that are argued exactly as case 1 above. Case 1 is actually a specialization of case 2 so this is not surprising.

3. Finally assume the root has two children each of equal page height. Under the rules above the new partitioning is formed by merging the topmost pages of the two children and adding the root if the combined page is not too large. If the combined page is too large, the topmost pages of both children are closed and a new page is started for the root.

- **Root fits (sum of children's local page sizes is less than p): the page height of the new partitioning is the same as that of the children so the existence of a partitioning of lesser page height would violate the induction hypothesis for $k-1$. If there is a partitioning of the same page height but smaller topmost page then it must contain a height $k-1$ partitioning for one of the two children that violates the induction hypothesis.**
- **Root does not fit (sum of children's local page sizes is at least p): Again the topmost page has size one so no other partitioning of the same page height can have a smaller topmost page. If there is a partitioning of smaller page height then as before, it must contain a partitioning for one of the two children that violates the induction hypothesis.**

The "moreover" part holds because we never go back and invalidate the optimality of the partitioning of sub-trees.

The rule in Figure 4.3 performs a constant amount of work at each node and so can be applied in linear time. $\mathcal{QED}$

Based on Theorem 4.1 we will refer to a partitioning resulting from the rules in Figure 4.3 as the "optimal bottom up partitioning" of a tree. The optimal bottom up partitioning is optimal in the sense that it minimizes the number of pages accessed in the worst case root-leaf traversal. However, it can produce a large number of very small pages. This problem results from the automatic closing off of a page if its sibling has a greater page height. Because we do not worry about aligning pages on block boundaries in the static text case, these small pages do not cause serious problems. However, it is still worthwhile running a post-processing pass that merges small pages into their parent whenever possible because each page has some small amount of storage overhead. The results reported later in this chapter include the use of such a pass. We will have to return to this problem

in the next chapter where small pages can cause storage management problems.

In order to judge the overall performance of data structures using the optimal bottom up partitioning, we want to bound the page height in terms of the number of nodes and the tree height, H. Before proving the bound, two simpler results are needed.

Lemma 4.1 *In an optimal bottom up partitioning, each sub-tree in a tree encoded in a page of page height $k > 1$ contains at least one node having children with page height $k - 1$.*

Proof: If all its children have page height $k - 2$ or lower, split off the sub-tree into its own page and obtain a partitioning with a smaller root node. The difference in page heights allows us to make this change without increasing the page height of the root. $\mathcal{QED}$

Lemma 4.1 allows the simple observation that, under an optimal bottom up partitioning, all nodes in a page have the same page height.

Lemma 4.2 *While on a root-leaf path of pages in an optimal bottom up partitioning, the leaves within a page height k page where $k > 1$ either have one child page with page height $k - 1$ containing p nodes or two child pages with page height $k - 1$ containing a total of at least p nodes.*

Proof: Each leaf node is a sub-tree so by Lemma 4.1, it contains at least one page height $k - 1$ child. If neither of the conditions are met, then the parent would have been moved in with either or both of the children and a partitioning with a smaller root node obtained for that sub-tree. $\mathcal{QED}$

Theorem 4.2 *Let $0 \leq t < 1$ be an arbitrary constant. The page height of the worst case optimal partitioning of a tree is bounded above by*

$$1 + \left\lceil \frac{H}{p^t} \right\rceil + \left\lceil \frac{1}{1-t} \log_p n \right\rceil \tag{4.1}$$

where H is the height of the tree and n is the number of nodes in the tree.

Proof: Our proof is based on the optimal bottom up partitioning. Given such a partitioning, we construct a sequence of pages on a deepest path, in the page sense, such that at each stage we either divide the number of nodes in the current sub-tree by $\lceil p^{1-t} \rceil$ or reduce the height (in the node sense) of the sub-tree by $\lceil p^t \rceil$. At each point in the construction we consider either a single page or a pair of sibling pages. Start the construction at the root page and select any node in the page that has children at page height $k-1$ and consider its page height $k-1$ children. By Lemma 4.2, we know that there are at least p nodes in these child pages. Because there are p nodes, one of the following two conditions must be met:

1. there are at least $\lceil p^{1-t} \rceil$ children pointing to child pages with page height $k-2$, in which case we select the child whose page height $k-2$ children have the smallest portion of the entire sub-tree, or

2. there is at least one node pointing to pages at page height $k-2$ such that the length of the path from the root of the page to the node has length at least $\lceil p^t \rceil$. Select that node's page height $k-2$ children for the next step.

If neither of these conditions are met, then we could not be dealing with p nodes. Case one can only occur $\left\lceil \log_{\lceil p^{1-t} \rceil} n \right\rceil$ times and case two can only occur $\left\lceil \frac{H}{\lceil p^t \rceil} \right\rceil$ times. Add one for the root page, remove the inner ceilings, and simplify the log to obtain an upper bound on the length of the path constructed. Because this path is a deepest path in the page sense, the bound also applies to the page height of the tree. $\mathcal{QED}$

Two corollaries can be obtained by selecting specific values of t. Choosing $t = \frac{1}{2}$, we obtain a bound of the form $1 + \left\lceil \frac{H}{\sqrt{p}} \right\rceil + \left\lceil 2 \log_p n \right\rceil$ which is interesting for its simplicity. Choosing $t = 1 - \frac{\lg \lg p}{\lg p}$, the bound takes the form

$1 + \left\lceil \frac{H}{p} \lg p \right\rceil + \left\lceil \frac{\lg p}{\lg \lg p} \log_p n \right\rceil$ which is interesting because it is approximately $\lg p$ times the sum of the two trivial lower bounds of $\frac{H}{p}$ and $\log_p n$.* In order to demonstrate that these logarithmic terms are necessary, we describe a method for constructing a tree with a worst cast optimal page height within a constant factor of this bound. The general structure of the tree consists of a root node and then K levels each made up of uniform groupings of nodes which we call clusters. In Figure 4.4 we show the root and the first two layers of clusters. Each cluster

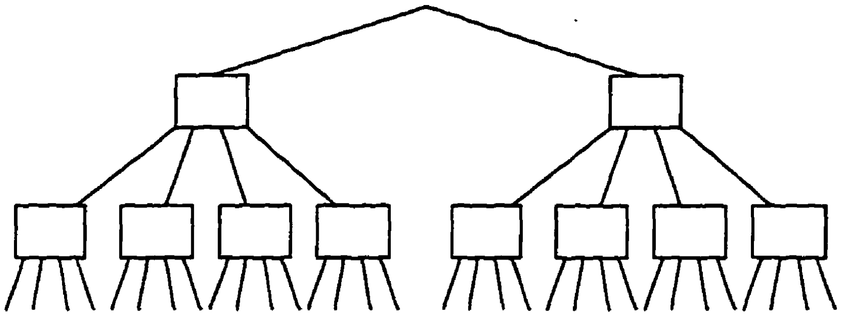

Figure 4.4: Tree Construction

takes the form of a perfectly balanced sub-tree of height $\lg \lg p$ with $\frac{\lg p}{2}$ strands below it as shown in Figure 4.5 (for this construction to work p should be of the form 2^{2^k}). The length of each strand is chosen to make the total number of nodes slightly more than $\frac{p}{2}$. If there are K levels of clusters then the worst case optimal partitioning of this tree has page height $K + 1$ and places each cluster on its own page. In addition we obtain,

$$H = 1 + K\left(\frac{p - \lg p}{\lg p} + \lg \lg p\right) \approx \frac{Kp}{\lg p}$$

*The value of t chosen here is an approximation to the solution of $p^{1-t} = \frac{1}{1-t}$ which would make the multipliers equal. The exact solution is $t = 1 - \frac{\omega(\ln p)}{\ln p}$ where ω satisfies $\omega(x)\, e^{\omega(x)} = x$. The value of t used here can be obtained by substituting in the first term of the asymptotic expansion of ω found in [12]. We use the slightly less optimal binary logarithm in order to simplify the construction that follows.

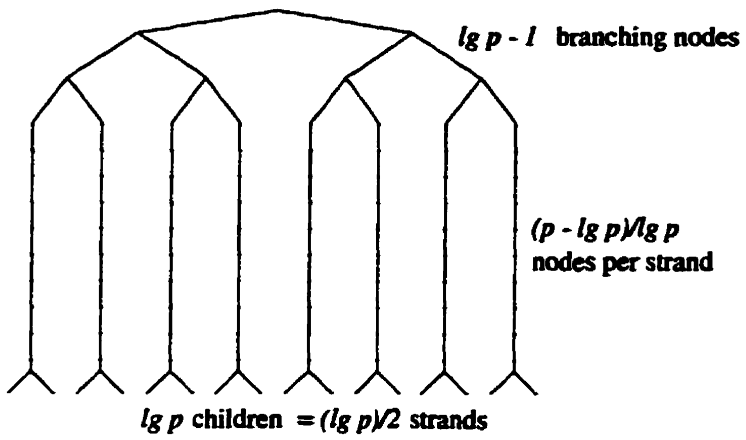
lg p - 1 branching nodes
(p - lg p)/lg p nodes per strand
lg p children = (lg p)/2 strands

Figure 4.5: Node Cluster

$$\begin{aligned}
\#clusters &= \lg^{k-1} p + \lg^{k-2} p \ldots + 1 \approx \lg^{k-1} p \\
n &= \frac{p + \lg p}{2} \#clusters \approx \frac{p}{2} \lg^{k-1} p.
\end{aligned}$$

Using these approximations, we note $\frac{H}{p} \lg p \approx K$ and $\log_p n \approx \frac{(k-1)\lg\lg p}{\lg p}$. Placing these approximations in the upper bound and removing the floors and ceilings, we obtain $1 + K + K - 1$ or $2K$ so this tree has actual page height about one half the upper bound.

This construction leads to the following theorem:

Theorem 4.3 *Given $p > 16$ and $\frac{\lg\lg p}{\lg p} < t < 1$, there exists a family of trees T_k of monotone increasing size such that for large k the ratio of the page height bound to the actual worst case optimal page height is bounded above by a small constant.*

Proof: First we handle the case $t \leq 1 - \frac{2}{\lg p}$ using the cluster construction from the previous discussion. Construct clusters containing $\lceil \frac{p^{1-t}}{2} \rceil$ strands with $\lceil p^t \rceil - 1$ nodes each. If the total number of nodes is not greater than $\frac{p}{2}$, then add a single node to each of one or two strands until this bound is met. Because of the constraints on p and t, the total number of nodes per cluster will now be more than $\frac{p}{2}$ and at most p. Using k layers of clusters and a single root node in its own page, we obtain an actual page height of $k+1$ and the bound is approximately

$$1 + \frac{k}{p^t}(p^t + \lg p^{1-t}) + \frac{1}{1-t}\log_p p^{1+(1-t)(k-1)}$$

which is approximately $k\left(2 + \frac{(1-t)\lg p}{p^t}\right)$ if we ignore terms not linearly dependent on k Note that t is bounded away from zero so p^t is at least $\lg p$.

Now consider $t > 1 - \frac{2}{\lg p}$. In this case we construct a chain of length kp. The bound is then $1 + \frac{kp}{p^t} + \frac{1}{1-t}\log_p(kp)$. Let $\epsilon = t - (1 - \frac{2}{\lg p})$ and note that $p^{\frac{2}{\lg p}} = 4$ so the bound becomes $1 + \frac{4k}{p^\epsilon} + \frac{1}{1-t}(1 + \log_p k)$ which is $\frac{4k}{p^\epsilon} + O(\log k)$. Finally note $1 < p^\epsilon < 4$ to obtain the result. $\mathcal{QED}$

This theorem shows that the bound in Theorem 4.2 is not excessively pessimistic.

In order to understand exactly how the bound in Theorem 4.2 varies with H and n, we select the value of t that minimizes the continuous version of formula 4.1:

$$1 + \frac{H}{p^t} + \frac{1}{1-t}\log_p n. \tag{4.2}$$

Differentiating this formula and finding a zero of the resulting formula, we obtain $t = 1 - \frac{2\omega\left(\sqrt{\frac{p\ln n}{4H}}\right)}{\ln p}$ where $\omega(x)e^{\omega(x)} = x$. Given $p \geq 2$, this value is between zero and one because $H \geq \lg n$. Substituting this value of t into 4.2, we obtain

$$1 + \frac{H}{p}p^{\frac{2\omega\left(\sqrt{\frac{p\ln n}{4H}}\right)}{\ln p}} + \frac{\ln p}{2\omega\left(\sqrt{\frac{p\ln n}{4H}}\right)}\log_p n.$$

Simplifying using $e^{\omega(x)} = \frac{x}{\omega(x)}$, converting p to $e^{\ln p}$ and finally expressing $\log_p n$ in terms of the natural logarithm, we obtain:

$$1 + \left(\frac{1}{2\omega\left(\sqrt{\frac{p\ln n}{4H}}\right)} + \frac{1}{4\omega^2\left(\sqrt{\frac{p\ln n}{4H}}\right)}\right)\ln n \tag{4.3}$$

so we see that, unsurprisingly, it is the ratio of H to $\ln n$ that is critical to the upper bound. Given H and n, we can use formula 4.3 to obtain an approximate bound on the page height of a partitioned tree (a trivial lower bound on the page height is $\max\left(\frac{H}{p}, \log_p n\right)$).

For the special case of the PAT tree, we can apply the results of Szpankowski mentioned in Chapter 1 to obtain the almost sure convergence of the bound to

$$1 + \left(\frac{1}{2\omega\left(\sqrt{\frac{ph_3}{4}}\right)} + \frac{1}{4\omega^2\left(\sqrt{\frac{ph_3}{4}}\right)} \right) \ln n. \tag{4.4}$$

Corless *et al.*[12] show that

$$\omega(x) = \ln x - \ln\ln x + O\left(\frac{\log\log x}{\log x}\right).$$

From this expansion, we obtain

$$\frac{1}{\omega(x)} = \frac{1}{\ln x} + O\left(\frac{\log\log x}{\log^2 x}\right).$$

Substituting this expression into formula 4.4, simplifying and placing some small terms in the order term (assume p is much larger than h_3), we obtain the almost sure convergence of the bound for partitioned PAT trees to

$$1 + \left(1 + O\left(\frac{\log\log p}{\log p}\right)\right) \log_p n.$$

Note that this bound does not apply to the CPT because of the presence of overflow nodes. However, it does give good reason to be optimistic about the performance of a partitioned CPT.

4.3 Compact PAT Trees for Secondary Storage

The CPT for secondary storage is obtained by applying optimal bottom up partitioning to the structure from Chapter 3 and using that CPT representation for each page on secondary storage. As noted earlier, we add one bit to each leaf to indicate if it is a suffix offset or a pointer to another page. When discussing the partitioning of the CPT, we have to stress that we deal only with the tree formed by the internal nodes and, as in Chapter 3, allow the position of the internal nodes to be implicit in that structure.

For the static case, we do not concern ourselves with aligning pages on block boundaries. In practice, the lack of alignment results in increased data transfer time but no increase in seek time. Bin packing heuristics could be used to place the pages in blocks if the increased data transfer time presents problems. However, we have not seen such a problem. The encoding of each page consists of

- header: containing the number of nodes on the page,
- tree: a compact encoding of the tree structure with the skip bits as discussed in Chapter 3,
- offsets and pointers: an array containing the leaf labels of the tree, each of which is either a suffix offset or a sub-page location. Each array entry has a single bit indicating which case is occurring. Because they both address approximately the same amount of data, we assume suffix offsets and page locations require the same number of bits.

4.3.1 Compact Suffixes On Secondary Storage

The technique of dropping the low order bits discussed by Shang[44] and in Chapter 3 has some new implications for text stored on secondary storage. It results in a smaller index and the cost of performing the search to determine the actual values for each offset is small relative to the time needed to access secondary storage. In addition, the reduced per node storage results in a higher branching factor within the index. The higher branching factor may lower the page height of the resulting tree although the partitioning algorithm makes this unlikely. Another side-effect of suffix pointer truncation results from our requirement that suffix offsets and sub-page pointers be the same size. Truncating suffix offsets requires a similar truncation in the disk block addresses. In our test system, we achieve the page pointer truncation by ensuring each page starts at an offset that is a multiple of 2^l where l is the number of bits we need to truncate to make these sizes equal. Aligning pages in this way may result in some waste storage that must be traded against the savings from the suffix offset truncation. For word indices on documents capable of being managed on current computers, l is usually the same as the number of bits truncated in the text. For a character index, l needs to be increased by approximately two bits because the page pointers have to address about four or five times as much data as the suffix offsets.

4.4 Empirical Results

Table 4.1 shows the index sizes for several indices on the test documents when using full suffix pointers. When producing the empirical results for indices on secondary storage, the optimal skip sizes from Chapter 3 were used for Holmes, XIII, and the Bible. For the OED, a skip size of six was chosen somewhat arbitrarily. In each case, the height of the tree matches the number of accesses to secondary storage needed to perform a search if the root page is held in memory because one further access is required to locate the test suffix. Table 4.2 shows the reduced sizes of the 4k page size indices above when some suffix bits are truncated. The number of bits truncated in each case was chosen to make the size of each leaf an integral number of bytes for performance reasons. While the savings due to suffix truncation appear modest when compared to Table 4.1, such a direct comparison is not really meaningful. The values in Table 4.1 are based on

Text	Page Size	Depth	#Pages	Index Size
Holmes	1K	2	178	144k
	2K	2	85	144k
	4K	2	40	144k
	8K	2	19	144k
XIII	1K	3	4770	3083k
	2K	3	2365	3073k
	4K	2	1179	3068k
	8K	2	596	3066k
Bible	1K	3	10781	4938k
	2K	3	5459	4913k
	4K	3	2985	4901k
	8K	2	1397	4894k
OED	1K	5	1285521	542m
	2K	4	698923	541m
	4K	4	374414	540m
	8K	3	195994	539m

Table 4.1: Static Index Sizes

the use of 18, 20 and 23 bit sized fields for the suffix offsets. In practice, an integral number of bytes would be reserved for each leaf, significantly increasing the index size. The indices in Table 4.2 already use an integral number of bytes for each leaf and so would not increase in a practical system.

4.5 Adapting to CD-ROM

The disk block size of a CD-ROM is 2048 bytes of application data plus about three hundred bytes of error detection and correction information (see [14] for an

Text	Truncation	Index Size	Reduction
Holmes	3	134k	7%
XIII	5	2845k	7%
Bible	8	4211k	14%
OED	7	489m	9%

Table 4.2: Effects of Suffix Truncation

overview of the properties of CD-ROM). Using this block size when searching on CD-ROM can lead to poor performance because of the very high seek time of these devices. Single spin CD-ROM drives can require up to 1 second to locate a particular block. The average performance is about 0.75 seconds. These same drives have a transfer rate of about 150k bytes per second so transferring a block requires about .014 seconds. While faster CD-ROM drives are available, the ratio of seek time to transfer rate remains roughly constant.

For the purposes of searching data on CD-ROM, it does not seem reasonable to spend fifty times longer locating pages than transferring actual data from the disk. In order to better balance these costs of searching, when applying the CPT structure to data on CD-ROM, a much larger page size is used during partitioning. An apparently reasonable guideline is to use as much time transferring data as is used locating the data to transfer. Equating these two values leads us to use page sizes of between 100k and 150k bytes. The use of 100k byte pages for searching the OED document results in a tree of height two. If the root page is held in memory, searches can progress with one read of the index at a cost of roughly twice the seek time and one smaller read of the text to acquire the test sample. Even with a slow drive, the first response to a query will be available in less than three seconds. Common multi-spin drives should achieve sub-second response time on word or phrase queries.

4.6 Comparison to Other Structures

The PaTrie structure of Shang[44] and Merrett and Shang[37] is similar in approach to the structure presented here. It is not clear which of the two structures will be more compact because that will likely depend on the text. However, the improved partitioning algorithm used here will result in superior performance on secondary storage. Empirically, Shang shows the PaTrie requires an average of between 5 and 7 accesses to secondary storage when using 1k byte pages for searching a text with 100 million index points. However, the PaTrie required as many as 46 disk accesses for some suffixes in that document. The compact PAT tree used a maximum of 5 accesses when searching a comparable text (the OED). When using 8k byte pages, the CPT structure required only 3 accesses for searching the OED document. The compact PAT tree is also less processor intensive and so should perform better on large page sizes such as those used for searching CD-ROM. Because it scans every bit in the encoding when traversing a block, the PaTrie is unlikely to extend well to larger page sizes. Shang notes that the PaTrie is processor bound when using 1k byte pages. Use of the efficient tree encodings from Chapter 2 could remedy this weakness of the PaTrie.

Another comparable structure is the SB-tree of Ferragina and Grossi[18]. On a 100 million index point document, they report using 6 accesses to secondary storage using an index requiring about 8.25 bytes per index point. While this appears excessively large when compared to the CPT, it must be remembered that the SB-tree has guaranteed logarithmic height while the PAT tree, and hence the CPT, has logarithmic height almost surely under certain text models. Further performance tradeoffs can be used to reduce the size of the SB-tree so it is comparable to that of the CPT, but at a cost in accesses to secondary storage. The SB-tree is also likely to be processor intensive but again this problem can be easily alleviated using the efficient tree encodings from Chapter 2.

Chapter 5

Dynamic Text on Secondary Storage

In this chapter, we extend the CPT structure to support changes to the underlying document or document set. In addition to detailing the changes to the PAT tree we show that the partitioning method of the previous chapter allows changes to the tree with only local changes to the page structure. Finally, we address the problem of placing the pages of the CPT structure into blocks on secondary storage.

There are two models of updates to the text that we consider:

- insertion, deletion, or replacement of a character in a single indexed document, or
- insertion or deletion of an entire document from an index on a set of documents.

The latter is the External Dynamic Substring Search (EDSS) problem formalized by Ferragina and Grossi[18]. In both cases the changes to the text result in the insertion or deletion of multiple suffixes to or from the PAT tree.

Simple character changes can result in a number of suffix insertions and deletions proportional to the maximal offset occurring in the PAT tree. For example, when changing the third character of the example string, *abccabca$*, to a *b*, we must delete the suffixes *cc*, *bcc* and *abcc* and replace them with *bcab*, *bb*, and *abb*. To determine the exact set of suffixes affected, we start at the first index point preceding the change point and scan backward. Each index point encountered is searched in the PAT tree until a leaf is reached. If the final offset value that occurs in the search is at or beyond the change point then the suffix is removed and replaced with the new suffix starting at the index point. If the leaf is encountered before the change point, then the tree position of the suffix and any suffixes to the left are not effected by the change to the string. The character change model is easily extended to include the insertion, deletion, or replacement of sub-strings of the original string. For such operations all suffixes starting inside the changed region are first inserted, deleted or replaced and then the suffixes preceding the change point are handled in the same manner as during a simple character change. Another basic operation that is frequently used with suffix indices is that of prepending new characters to the index string. The prepend operation is of interest because updating a suffix index after a prepend operation only requires the insertion of the new suffixes. It does not require updating the index for any existing suffixes and so can be more efficiently implemented. An issue we will not address is the nature of suffix offsets under the character change model. If a simple offset is used then insertion or deletion of the first character in the text requires an update of every offset in the CPT structure. Majster and Reiser[33] suggest the use of a "Dewey Decimal" numbering scheme for string positions that solves this problem but do not address the implementation of such a scheme in a practical system.

Under the EDSS model, the text being searched is made up of multiple documents, $\Delta = \{\delta_1, \delta_2, ...\delta_m\}$, stored on secondary storage. The model allows the insertion of new documents into Δ or the removal of existing documents. Updates to the documents can be modeled as a delete followed by an insert. When using a suffix based index, including the CPT, insertion or deletion of a document results in the insertion or deletion of the suffixes corresponding to each index point in the document. A minor issue for the multiple document model is the use of end markers to ensure the uniqueness of each suffix. In order to retain the uniqueness of suffixes across documents, the end marker must be different for each document. In our test system, we use the string "\$<doc>\$" where \$ is the previous unique end marker and <doc> is a unique identifier for the document (e.g. its name).

Even when indexing a single dynamic document, the EDSS model is likely to be more relevant than the single document model. Documents managed on secondary storage tend to be large and often can be broken down into sub-documents each of which fits in primary storage. For example, a dictionary or encyclopedia can be broken into entries, a scientific journal into papers and fiction or non-fiction books into chapters, sections, or paragraphs. The breakdown into sub-documents is necessary to allow users and programs to efficiently manage the documents. In practice, the lack of such a natural breakdown will result in the use of an "unnatural" breakdown such as one based on disk blocking. For this reason, we use the EDSS model in our empirical testing.

The preceding discussion shows that higher level document operations can be implemented using suffix operations, so we will only concern ourselves with the insertion and deletion of suffixes from the CPT structure for the remainder of this chapter.

5.1 Updating PAT Trees

Before considering the effect of suffix insertions and deletions on the partitioned CPT, it is worthwhile reviewing the effect of such changes on the underlying PATRICIA tree.

Adding a suffix to the tree requires the addition of a new internal node having as one of its children a leaf containing the new suffix's offset. In terms of the tree of internal nodes, the new node may either split an edge between two internal nodes of the PATRICIA tree or be inserted below an existing leaf. To insert a new suffix into a PATRICIA tree, the following steps are taken:

1. search the tree using the new suffix until we reach a leaf, call the suffix located at this leaf the test suffix,
2. determine the first bit position at which the new suffix and the test suffix differ,
3. if the suffixes first differ at a bit beyond the bits tested during the traversal, then replace the leaf with an internal node having the test suffix and the new suffix as children,
4. otherwise find the edge on the root-leaf path that skips over the bit position where the suffixes differ and split the edge by inserting a new node having as children the new suffix and the old sub-tree reached by the edge,
5. set the offset or skip values of the new node and, in the second case, the old sub-tree appropriately.

For example, if we prepend an a to the example string, $S = abccabca\$$, and then insert the new suffix *aabcc...* into the PAT tree we first search the new suffix in the existing tree terminating at leaf 1. We then determine that the new suffix and the suffix at offset one differ in their fourth bits. This bit is skipped on the edge linking the nodes labelled [3] and [7] so the new suffix is inserted on that edge. The new suffix has a zero in the fourth bit position so it will be the left child of the new node. The resulting PAT tree is shown on the right of Figure 5.1.

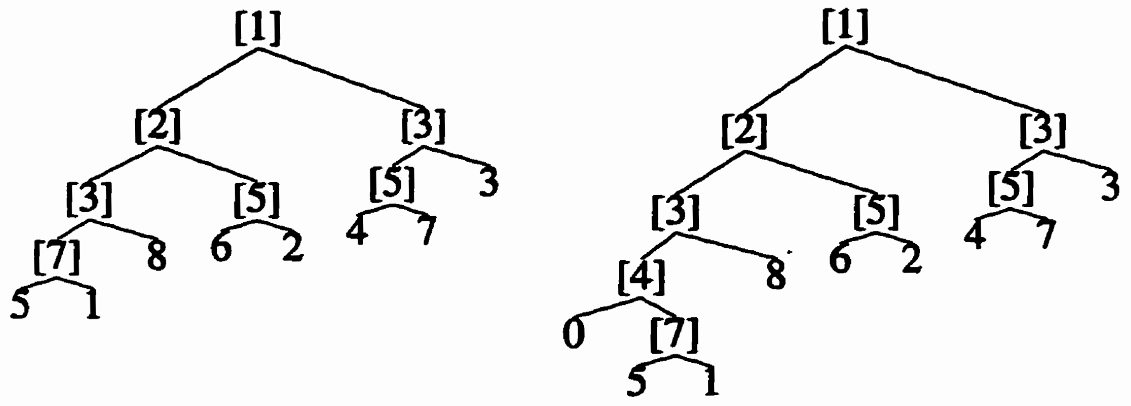

Figure 5.1: Original and Updated PAT Trees

Deletion of a suffix is more easily handled:

1. search the suffix in the PAT tree to locate the suffix's leaf,

2. remove the suffix and its parent by replacing the parent with a straight through edge,

3. if the ex-sibling is not a leaf, then update its skip value (if offsets are used then no change is needed).

A more detailed discussion of the updating procedures can be found in Sedgewick's discussion of PATRICIA[42]. When the keys being inserted are known to be consecutive suffixes, more efficient methods of implementing these update operations on suffix trees are known, see McCreight[36], but they require the maintenance of extra data in each node.

To adapt the PATRICIA update methods to the partitioned CPT we need to handle three new problems:

- overflow nodes,
- maintaining the worst case optimal partitioning,
- handling truncated suffix offsets.

Neither overflow nodes nor truncated suffix offsets cause significant problems when updating the CPT structure.

5.2 Dynamic Optimal Bottom Up Partitioning

The optimal bottom up partitioning rule is based on information local to a sub-tree, so the effects of repartitioning are limited to the path upward from the point we insert or delete a node to the root. Because of this locality, we can efficiently maintain the optimality of the partitioning in the presence of update operations. This differs considerably from the other partitioning strategies discussed in the previous chapter where a local change to a tree can have global side-effects on the partitioning.

Because we operate on the tree of internal nodes of the PAT tree, the two binary tree operations of interest to us are the insertion of a new node either as a leaf or by splitting an edge and the deletion of a leaf or straight through node (a node with only one child). We refer to the location of the new node or the parent of the node removed as the *change point* in the tree. We handle the insertion and deletion of nodes separately.

Theorem 5.1 *An optimal bottom up partitioning of a binary tree can be updated to reflect a node insertion operation by updating at most $2H + 1$ page definitions, where H is the page height of the partitioned tree prior to the operation.*

Proof: The optimal bottom up partitioning rules are based solely on information local to a sub-tree and its sibling. Therefore, only pages containing nodes on the path upward from the change point to the root or those nodes' siblings can possibly change. For the same reason, within such pages only the decisions made at nodes on that path can change. In the remainder of the proof we consider what can happen when reapplying the optimal bottom up partitioning rules to the nodes along this path. The actual implementation of the insertion operation for a partitioned tree closely follows the proof.

At each stage in the recomputation of the tree partitioning, we have a node, i, that we are considering adding to a page and we have to determine how its addition changes the page definitions. Initially, i is the new node being inserted and the page is either the page containing i's parent if i is a leaf or the page containing i's child otherwise (the only real problem is when i is inserted in an edge between two pages - in that case we select the lower page). Set h_i, the page height of the node being inserted, to the page height of i's child or to one if i is a leaf.

Let h_{page} be the page height of the current page. There are two cases that have to be considered immediately:

1. If $h_i < h_{page}$ then i's sibling either has page depth h_{page} or has a lesser page depth but is on its own full page. According to the partitioning rules, i should be placed on its own page. Once i is placed on its own page, then neither the page height nor the local page size of any already existing node are changed so the remainder of the optimal bottom up partitioning is identical to the previous partitioning.

2. In the more general case, $h_i = h_{page}$, we logically add i to the page definition (we say logically because this addition may cause the page size to exceed p).

In the second case, we have to continue reapplying the partitioning rules upward. First consider nodes other than the root of the page. For these nodes, the page height is unchanged so no partitioning decision based on page height will change. Additionally, the local page sizes for one child of these nodes has increased by exactly one but each of these was at most $p - 2$ before because these pages where later augmented with the node and its parent. So the increased local page size is still less than p and no partitioning decision based on page size will change either. Therefore the first node for which the partitioning decisions can change is the root of the page. Two possible situations arise from application of the partitioning rules to the root of the page:

1. If the root of the page still fits on the page, the rules allow us to keep it in the page definition. The local page size and page height of any nodes further up the path to the root of the tree are unchanged. Because these values do not change, the page definitions from the old partitioning are valid.

2. If the root of the page no longer fits on the page, then its children and their sub-trees must be placed on their own pages (if they are not already). We then have to decide where the rules place the old page root. If it is the root of the entire CPT, then they simply place it in its own page and the reapplication of the partitioning rules is complete. In general though, we consider the effect of its insertion into its parent's page. After incrementing its page depth by one, the old root becomes i and we iteratively apply the rules above to determine how its addition changes higher level page definitions.

Based on this argument we see that only the definitions of the pages containing children of the original page roots can change during an insert. As well, a new page definition may be required to hold the root of the entire CPT. There are at most $2H + 1$ such pages so the result follows. *QED*

Theorem 5.2 *An optimal bottom up partitioning of a binary tree can be updated to reflect a node deletion operation by updating at most H page definitions and removing at most another $H - 1$ page definitions, where H is the page height of the partitioned tree prior to the operation.*

Proof: As with insert, we only need to consider the reapplication of the rules to nodes on the path upward from the change point to the root. While the details are different, the structure of the argument is the same as the insert case. At each stage we have a page that has had a node removed from it and we consider what happens as we reapply the partitioning rules on the path upward from the change point. Initially, this page is the one that used to contain the deleted node.

There are three possible cases,

1. The remaining page definition is empty and the decision to place the node on its own page was based on its sibling's page height (i.e. the sibling has a greater page height). In this case, we remove the page definition. The removal of the node and its page does not have any effect on the decisions made at its parent because of the difference in page heights. The page height and local page size of all other nodes remain unchanged so the page definitions from the original partition are still valid.

2. The remaining page definition is empty and the decision to place the node on its own page was based on the total of the local page sizes for it and its sibling. If the sibling now has local page size $p - 1$ (it cannot be less), then the partitioning rules require the parent to be placed on the sibling's page.

We remove the parent from its page definition and are again in the position of having a page from which we have removed a node and have to apply these same arguments to the parent page to determine how the remainder of the partitioning might change. Otherwise the sibling's local page size is p, so the parent cannot be added to the page definition and, as before, the page height and local page size of all other nodes remain unchanged so the page definitions from the original partition are still valid.

3. There are nodes remaining on the page.

In the last case, we have to consider what happens as we reapply the partitioning rule within the page. Any decisions to create pages made when applying the rules to nodes from the change point up to and including the root of the page had to be based on the page height of the nodes because decisions based on local page size break off pages for both children. The page height of the nodes is unchanged so these decisions will not change when the rules are reapplied. Therefore, the parent of the root of the current page is the first node on the upward path where the partitioning decisions might change. Note that if this node does not exist then we are at the root of the tree and the reapplication of the rules is complete. Otherwise, we have to consider the two cases that can occur at this node:

1. If the partitioning decision is unchanged then the page height and local page size of all nodes further up the path to the root of the tree are also unchanged so the page definitions from the old partitioning can be used for the remainder of the upward path.

2. If the partitioning decision changes to move the root's parent (and possibly its sibling if it has the same page height) on to the current page, then we remove that node from the parent page and update the current page definition.

In the second case, we are again left with a page (the parent page) from which we have removed a node and we iteratively apply the arguments above.

We have shown that the only nodes whose partitioning decisions can change are the parents of the roots of the pages. For each of these we may have to update the definition of both the root and its parent's page. As well, we may have to delete the sibling page definition. At most H page definitions can be updated and another $H-1$ definitions may be removed. $\mathcal{QED}$

These two theorems lead immediately to the following result:

Theorem 5.3 *An optimal bottom up partitioning of a binary tree can be maintained under node insertion and deletion operations by updating at most* $2H+1$ *page definitions per operation, where* H *is the page height of the partitioned tree prior to the operation.*

Theorem 5.3 is quite pessimistic on the performance of the repartitioning process. As we will see, for the well balanced PAT tree, on average about one page is updated for each node operation. However, it is not difficult to construct trees such that an insert will require the full $2H+1$ page updates or a delete will require H page updates and the removal of $H-1$ page definitions.

5.3 Dynamic Compact Pat Trees

The dynamic CPT is obtained by combining the PAT tree insertion operations from Section 5.1 and the repartitioning procedure implicit in the proof of Theorem 5.3. We have to stress that, as in Chapter 4, we perform the partitioning operations in terms of the tree made up of the internal nodes of the CPT.

5.3.1 Insertion of a Suffix

When inserting a new suffix into the CPT, we first locate the page containing the portion of the tree that must be changed by executing a search for the suffix (this search must fail because suffixes are unique due to the end marker) and then determine the bit position at which the suffix located at the last leaf encountered and the new suffix differ. The page containing the portion of the tree that skips over that bit position is loaded into memory and the new suffix inserted into it. In the case that the edge that contains the bit position spans a page boundary then the child page is used.

The insertion of a suffix can result in one of three conditions:

- an increase of one in the number of internal nodes in the page,
- no change in the number of internal nodes in the page,
- an increase of more than one in the number of nodes in the page.

The latter two conditions are only possible due to the use of overflow nodes. In the second case, we can simply write the page back to secondary storage. This cannot invalidate the partitioning because the page heights of all nodes and size of the pages are unchanged. The third case can be handled using multiple invocations of the first and is sufficiently rare that we do not concern ourselves with it.

The steps performed after insertion are:

1. if the suffix pointer is inserted adjacent to another suffix pointer*, then we have to consider the page height of the page into which it is being inserted. If the height of the page is greater than one then the new node and both leaves are split off into a new page and replaced in the page with a pointer

*Equivalently, we are inserting a leaf in the tree of internal nodes.

to the location of this page. Otherwise we simply add the new node and leaf to the page. If the updated page fits in a block we write it back and we are done. Otherwise we have to repartition the page.

2. if the suffix pointer is inserted with an internal node as its sibling, then we add the new internal node and leaf. Again, if the updated page fits in a block we write it back and are done. Otherwise we skip to the repartitioning phase.

If the updated page is too large to fit on a block, then we have to adjust the partitioning of the page and its parent page. The repartitioning process operates by placing the sub-tree children of the root in their own pages and then either placing the root on a new page or inserting it in its parent's page depending on their relative page heights. If the root is inserted in its parent's page then that page may become too large and the process continues recursively. The steps to be followed are:

1. remove the root of the page and write its non-leaf children to secondary storage replacing them with pointers to where they were written.
2. set the page height of the root node to one more than that of its children.
3. if the page's root node is the root of the entire CPT then write it to its own page and mark this page as the root page,
4. else if the page height of the root's parent is the same as the updated page height of the root, then insert the root in the parent. If the updated page fits in a block then write it back, otherwise recursively repartition the parent page.

5. otherwise the page height of the root's parent is greater than the updated page height of the root so we write the root to its own page and update the sub-page pointer in the parent page.

5.3.2 Deletion of a Suffix

During the deletion of a suffix, the suffix is first searched through the CPT to locate its leaf. That leaf and its parent are then removed and the sibling's skip value updated. Note that if the sibling is on another page, then updating its skip value requires a read and write of secondary storage. In practice, this extra I/O can be avoided if the cumulative offset for the root of each page is maintained in each page. We do not, however, currently use this optimization.

After deleting a suffix pointer from a page, there are again several cases:

- a decrease of one in the number of nodes in the page,
- a decrease of more than one in the number of nodes in the page,
- no change in the number of nodes in the page.

As with insertion, we only consider the first case.

The repartitioning in the case of a deletion attempts to merge the current page with its parent node and, if it has the same page height, its sibling page. If the combined structure fits in a block then their pages are combined and the resulting page written back. The parent node is then removed from its old page and that page is then considered for repartitioning. The steps performed after a deletion are:

1. if the current page is the root page then we are done,

2. else if the page height of the current page and the page height of the sibling page are equal and the sum of their sizes plus one for their parent is less than or equal to p, then merge the pages and remove their parent node from the parent page. Recursively repartition the parent page.

3. else if there was only one node in the page, then remove the page and replace the parent page's pointer to this page with the remaining leaf,

4. else if the page height of the current page is greater than that of its sibling then move the parent node of the page from the parent page to the current page and recursively repartition the parent page.

5. otherwise no further adjustment is necessary and the page can be written back.

For a balanced tree, the splitting and merging of pages discussed here is exactly analogous to the splitting and merging of nodes during B-tree updates[42].

5.3.3 Block Layout on Secondary Storage

While the optimal bottom up partitioning rule produces a worst case optimal partitioning, it can also produce a large number of small pages. In practice, many of these pages result from sub-trees whose sibling tree has a higher page height. In such cases, the partitioning rule adds the parent into the child with the higher page height and "closes off" the other child irrespective of its size. If each page is assigned its own disk block, these small pages will result in very low storage utilization and high storage requirements. For example, when using a page size of 8k bytes, the OED required 195994 pages so the naive dynamic index would require 1.5g bytes or nearly three times the storage of the static index. Moreover, there is no guarantee that this multiplier does not get much worse.

To alleviate the problem of small pages, we place multiple pages on each block of secondary storage. While there are many possible approaches to this problem, the most general being Dynamic Bin Packing[11], we adopt a simple approach based on primary storage management techniques. We maintain free lists for storage regions of size $2^{k_0}, 2^{k_0+1}, ..2^{\lg P}$ and manage these using a "binary buddy"[1] style splitting and coalescing strategy within each block. During allocation, if there are no free regions of the appropriate size we recursively allocate from the next larger region and split the resulting region, using half for the page and leaving half in the free page list. If we ever run out of free regions of the largest size, a new block is allocated at the end of the index file. When a region is released, the free list is searched for the remaining half needed to return it to the next larger size. If the "buddy" region is found in the free list then the merged region is recursively freed to the next larger region class. While we could set k_0 to 1 and guarantee each allocated region is at least half full, this interferes with the savings from suffix truncation and so is not worthwhile.

5.3.4 Suffix Offsets

In the static case, the bits in truncated suffix offsets were recovered during searching by scanning a range of 2^l characters and finding the unique suffix that when searched in the CPT ended at the correct leaf. The same technique can be used to locate the correct suffix during the insertion operation.

An alternate method for locating the suffix is based on the observation that the correct test suffix is that suffix in the range which matches the suffix being inserted for the greatest number of bits. To see this, let i be the bit at which this suffix differs from the suffix being inserted. Consider any other suffix starting in the range. It must first differ from the suffix being inserted and the candidate test suffix at some bit j where $j < i$. From the properties of the PAT tree, bit j must be tested at some internal node on the path from the root to the candidate test suffix. Moreover the suffix being inserted must match the test candidate at bit j otherwise the search would have taken the other branch at that node and not end at this leaf.

The decision of which of these two criteria to use will depend on the relative speed of string comparisons and tree operations in an actual implementation. During deletions of suffixes, we start with the exact suffix offset and can use the suffix string to search the CPT and stop at the unique leaf that needs deletion.

As in the static case, any bits truncated from the suffix offsets also have to be truncated from the page pointers. We accomplish this by setting the value of k_0 in the storage manager to the number of bits we need to truncate from the page pointers. All pages are then aligned on a multiple of 2^{k_0} and so we can truncate the low order bits.

5.3.5 Practicalities

Many of the update operations require knowledge of the page height of sub-pages to make repartitioning decisions. Instead of retrieving the sub-pages to locate this information, we add one further element to the page pointer objects: the page height of the page being pointed to. In practice, two bits are sufficient for this field because it can be stored as a difference in page height between a page and its parent, shifted by one to make use of zero. This requires a further increase of k_0 by two.

Finally, we have said nothing about the order of suffix insertions and deletions under the EDSS model. If we assume the suffixes being updated are sparse in the lexicographically ordered set of the indexed suffixes, then little can be done to save reading a height one page per suffix (assuming updates occur near the leaves of the tree). However, by lexicographically sorting the suffixes and inserting or deleting them in order, we can avoid reading higher level pages more than once each. This ordering can also be used to merge multiple page repartitioning and resizing operations. We do not currently make use of this optimization in our test system.

5.4 Empirical Results

Page size	Leaf Size	2^{k_0}	# Blocks	Index Size
unaligned	4		195994	565m
8k	4	8k	195994	1531m
8k	4	32	96241	752m
8k	3	256	77914	609m

Table 5.1: Improved Block Allocation

The results of using the block allocation structure of Section 5.3.3 on the CPT index for the static OED is shown in Table 5.1. The first row shows the number of pages and the index size for a text index using 4 byte pointers but no alignment of objects on secondary storage. The second row shows the index size when using naive page alignment. The remaining lines show the number of blocks used for two leaf sizes using 8k pages. In the latter two cases, bits had to be truncated from the suffixes and k_0 had to be increased to allow for the smaller leaf objects. These results have to be interpreted with care because they do not include storage lost to external fragmentation (blocks not currently allocated at all) which would occur in a real system.

For a more realistic test of the dynamic CPT, we gathered an assortment of 161 documents, primarily English text, ranging in size from 21k to 1411k bytes. We then inserted all but one of these documents into a CPT using the algorithms described here. During the insertion of the last document, we recorded the number of page updates performed during the suffix insertions. The final document contained 14482 index points and its insertion required 14597 writes of secondary storage for an average of 1.01 writes per index point. The final index size was 67m bytes on a total text size of 50m bytes using 4 byte leaf objects.

5.5 Comparison to Other Structures

As with the static case, the most comparable structures are the PaTrie and the SB-tree. Little data is available about the dynamic PaTrie so direct comparison is not possible. However, it seems likely that the concerns of the static case will continue into the dynamic case.

The SB-tree, being based on a B-tree, makes the transition to the dynamic case quite easily. Here it has two advantages: its guaranteed height and its guaranteed storage utilization. The latter property avoids the more complicated storage management required by the CPT. These must be traded against the greater functionality and empirically fewer secondary storage accesses of the CPT.

Chapter 6

Conclusions

This chapter presents a short overview of the results already discussed in this thesis and presents some areas for future research. Chapter 2 provided an overview of traversable compact tree representations and extended these methods to the MBRAM model and trees with edge selection based on a label such as a trie. In Chapters three, four and five, we have shown several representations for PAT trees that use significantly less storage than previous methods. In particular, we have presented:

- a new representation for static PAT trees in primary storage that allows efficient searching with an expect storage cost of less than $3\frac{1}{2} + \lg n + \lg\lg\lg n$ bits per node for random text. Empirically we show the representation works well for real world data.
- a new representation for static PAT trees in secondary storage that we have shown empirically is little larger than a suffix array and offers significantly better performance than that offered by suffix arrays, partitioned PaTries or SB-trees.

- methods for managing the structure mentioned above that allow us to efficiently handle updates to documents on secondary storage.

Each of these structures represents a significant advance in our ability to search large bodies of textual data efficiently.

6.1 Applications

As stated in the introduction, our initial motivation for this work was the searching of large documents on CD-ROM. CD-ROM differs significantly from magnetic disk in that the cost of seek operations is extremely high when compared to magnetic disk. When searching a document with a size comparable to the capacity of a CD-ROM (approximately 600 million bytes at the time of writing), a suffix array will require approximately 45 random seeks on the disk at a cost of roughly 3/4 of a second each. Empirically, we see the structure presented in Chapter 4 searching files of this size in 3 disk accesses resulting in a 15 fold performance improvement. In addition to its uses in string processing, we believe the CPT structure will find uses in computational biochemistry where it will allow fast searching of even longer strings of genetic information.

Finally, we believe the structure presented in Chapter 5 is an extremely practical solution to the efficient phrase searching of large dynamic documents. It is yet to be seen if the added functionality is sufficient to make it a rival of inverted word lists in the large text database field, but it is clearly more suitable for many applications because of its ability to handle string, phrase and regular expression searching. Of particular interest is the searching of genetic information because such data does not admit a word breakdown. PAT trees have also been found useful for solving several other problems in text and data processing. In particular (see [24] unless otherwise noted):

- Searching picture data. By converting pixel data to "semi-infinite spirals," Gonnet proposed the use PAT trees to create an index capable of searching pictures.

- Regular Expression searching of text. Baeza-Yates and Gonnet show that PAT trees can be used to perform regular expression searches in sub-linear expected time[4].

- Finding longest repeated strings. The deepest internal node in the suffix tree has as its children the occurrences of the longest repeated substrings in the text. By attaching a single bit indicating which of the two children is deeper to each internal node a longest repeated sub-string can be efficiently located. In addition, given any string, the longest repeated continuation of that string can also be located. By adding a second bit to each node (to indicate two children of equal height), all of the longest repeated sub-strings can be located.

- Finding the most common continuation of a string. If each internal node is labelled with its size, suffix trees can be used to find the most common continuation of any string. With some work, similar techniques can be applied to PAT trees. This work is closely related to work by Gonnet *et al.*. that uses the PAT tree to generate random text that is "similar" to an input text.

The survey by Apostolico[3] gives an overview of several other uses of suffix trees, most of which can be extended to PAT trees. There is no reason to believe that the structures reported here cannot be applied to these and other problems with equally successful results.

6.2 Future Work

In addition to investigating the extensions above, we want to investigate the use of compact tree structures within other types of tree based structures. The representations developed here are directly applicable to implementing PATRICIA when the bits tested are monotone increasing and the differences in the offset of the test bits are, with high probability, small. Other similar structures may handle related cases. Shang has investigated some applications of related ideas to spatial data structures. An interesting problem remaining from Chapter 2 is the compact and efficient representation of dynamic binary or general trees.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, 1983.

[2] A. Andersson and Stefan Nilsson. Efficient implementation of suffix trees. *Software – Practice and Experience*, 25(2):129–141, February 1995.

[3] A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words, NATO ASI Series F, Computer and System Sciences, Vol. 12*, pages 87–96. Springer Verlag, Berlin, New York, 1985.

[4] R. A. Baeza-Yates and G. H. Gonnet. Efficient text searching of regular expressions. *ICALP'89, Lecture Notes in Computer Science 372*, pages 46–62, 1989.

[5] E. F. Barbosa, G. Navarro, R. Baeza-Yates, C. Perleberg, and N. Ziviani. Optimized binary search and text retrieval. *Algorithms – ESA '95, Third Annual European Symposium, Lecture Notes in Computer Science 979*, pages 311–326, September 1995.

[6] R. Bayer and K. Unterauer. Prefix B-trees. *ACM Transactions on Database Systems*, 2(1):11–26, March 1977.

[7] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[8] A. Brodnik. Searching in constant time and minimum space. Technical Report CS-95-41, Department of Computer Science, University of Waterloo, 1995.

[9] M. Carlisle, J. I. Munro, and R. Sedgewick. Data layout for memory locality. unpublished manuscript, 1993.

[10] M. T. Chen and J. Seiferas. Efficient and elegant subword-tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words,NATO ASI Series F, Computer and System Sciences, Vol. 12*, pages 97–107. Springer Verlag, Berlin, New York, 1985.

[11] E. G. Coffman Jr., M. R. Garey, and D. S. Johnson. Dynamic bin packing. *SIAM Journal on Computing*, 12(2):227–258, May 1983.

[12] R. M. Corless, G. H. Gonnet, D. E. G. Hare, and D. Jeffrey. On Lambert's ω function. Technical Report CS-93-03, Department of Computer Science, University of Waterloo, 1993.

[13] J. J. Darragh, J. G. Cleary, and I. H. Witten. Bonsai: A compact representation of trees. *Software - Practice and Experience*, 23(3):277–291, March 1993.

[14] J. Einberger. CD ROM characteristics. In S. Ropieque, editor, *CD ROM, Optical Publishing*, pages 31–42. Microsoft Press, Redmond, 1987.

[15] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*, pages 1–66. Elsevier, Amsterdam, New York, Oxford, Tokyo, 1990.

[16] C. Faloutsos. Access methods for text. *Computing Surveys*, 17(1):49–74, March 1985.

[17] C. Faloutsos and S. Christodoulakis. Description and performance analysis of signature file methods. *ACM Transactions on Office Information Systems*, 5(3):237–257, July 1987.

[18] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. *Proc.* 27^{th} *ACM Symposium on the Theory of Computing*, pages 693–701, 1995.

[19] P. Ferragina and R. Grossi. Fast string searching on secondary storage: Theoretical developments and experimental results. *Proc.* 7^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 373–382, 1996.

[20] E. H. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–500, September 1960.

[21] J. Gil and A. Itai. Packing trees. *Algorithms – ESA '95, Third Annual European Symposium, Lecture Notes in Computer Science 979*, pages 113–127, September 1995.

[22] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval*, pages 66–82. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[23] G. H. Gonnet. *Handbook of Algorithms and Data Structures.* Addison-Wesley, London, 1984.

[24] G. H. Gonnet. Efficient searching of text and pictures (extended abstract). Technical Report OED-88-02, Centre for the New OED, University of Waterloo, 1988.

[25] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, New York, 1989.

[26] G. Jacobson. Space efficient static trees and graphs. *Proc.* 30th *Symposium on Foundations of Computer Science*, pages 549–554, October 1989.

[27] G. Jacobson. Succinct static data structures. Technical Report CMU-CS-89-112, Carnegie Mellon University, 1989.

[28] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *Int. Journal of Foundations of Computer Science*, 1(4):425–447, December 1990.

[29] D. Knuth. *The Art of Computer Programming: Sorting and Searching, Volume 3.* Addison-Wesley, Reading Mass., 1973.

[30] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[31] C. C. Lee, D. T. Lee, and C. K. Wong. Generating binary trees of bounded height. *Acta Informatica*, 23:529–544, 1986.

[32] J. A. Lukes. Efficient algorithm for partitioning of trees. *IBM Journal of Research and Development*, 18(3):217–224, 1974.

[33] M. E. Majster and A. Reiser. Efficient on-line construction and correction of position trees. *SIAM Journal on Computing*, 9(4):785–807, November 1980.

[34] E. Mäkinen. A survey on binary tree codings. *The Computer Journal*, 34(5):438–443, 1991.

[35] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[36] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, April 1976.

[37] T. H. Merrett and H. Shang. Trie methods for representing text. Technical Report SOCS-93.5, McGill University, 1993.

[38] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the Association for Computing Machinery*, 15(4):514–524, October 1968.

[39] *The Oxford English Dictionary, Second Edition.* Clarendon Press, Oxford, 1989.

[40] R. C. Read. The coding of various kinds of unlabelled trees. In R. C. Read, editor, *Graph Theory and Computing*, pages 153–182. Academic Press, New York, 1972.

[41] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Englewood Cliffs, 1977.

[42] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading Mass., 1995.

[43] *(ISO 8879) Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML).* International Organization for Standardization (ISO), 1986.

[44] H. Shang. *Trie Methods for Text and Spatial Data Structures on Secondary Storage.* PhD thesis, McGill University, 1995.

[45] T. Snider. S vectors. unpublished manuscript, 1993.

[46] W. Szpankowski. Suffix trees revisited (un)expected asymptotic behavior. Technical Report CSD-TR-91-063, Purdue University, 1991.

[47] P. Weiner. Linear pattern matching algorithm. *Proc. 14th IEEE Symposium Switching Theory and Automata Theory*, pages 1–11, October 1973.

[48] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the Association for Computing Machinery*, 7(6):347–348, 1964.

[49] I. H. Witten, T. C. Bell, and C. G. Nevill. Models for compression in full-text retrieval systems. *Data Compression Conference*, pages 23–32, April 1991.

[50] S. Zaks. Lexicographic generation of ordered trees. *Theoretical Computer Science*, 10(1):63–82, January 1980.

Glossary

This glossary contains definitions for many of the topics discussed in this thesis. In those cases where the full definition is too large to include here a page reference to the main text is given.

[...]	Iverson's convention. The expression [...] has a value of one if the contents of the expression are true and zero otherwise.
$(x)_b$	The representation of x in base b.
$\lceil x \rceil$	The smallest integer greater than or equal to x.
$\lfloor x \rfloor$	The largest integer less than or equal to x.
almost sure	An event is almost sure if it occurs with probability one. A sequence of jointly distributed random variables, R_n, converges almost surely to R if $P\left(\lim_{n\to\infty} R_n = R\right) = 1$.
average case optimal	A tree partitioning is average case optimal if it minimizes the number of pages accessed when travelling from the root to a leaf, averaged over all leaves.

binary tree	All binary trees in this thesis are finite and rooted. A binary tree consists of a distinguished node called the root. The root may have a left sub-tree and/or a right sub-tree each of which must itself be a binary tree. See page 8 for more information.
block	A unit of data that can be transferred to or from secondary storage in a single atomic operation.
block size	The size of a block. In Chapter 4 we use P to denote the block size.
case conversion	The conversion of all strings to either upper or lower case during searching. Case conversion is frequently used because in many applications the capitalization of a word is not important.
Catalan number	$C_n = \frac{1}{n+1}\binom{2n}{n}$ is the number of binary trees with n nodes.
character index	An index that allows searching for matches that start and end at any character position.
convex partition	A partition of a tree such that each page contains a connected portion of the tree. All tree partitions considered in this thesis are convex.
degree	The degree of a tree node is the number of sub-trees of the node.
external node	A tree node is an external node if it has no children. External nodes are also called leaves.
Fuss-Catalan number	$C_n^{(m)} = \frac{1}{mn+1}\binom{mn+1}{n}$ is the number of m-ary tries with n nodes.

general tree	All trees in this thesis are finite, rooted and ordered. A general tree T is formally defined as a non-empty, finite set of nodes such that there is one distinguished node called the *root* of the tree, and the remaining nodes are partitioned into $m \geq 0$ disjoint sub-trees $T_1, T_2, T_3 ... T_m$ where the order of the trees is significant. See page 8 for more information.
height	The height of a node is the length of the longest path from the node to a leaf in its sub-tree. The height of a tree is the height of its root. Also see page height.
index point	A position in an indexed text that is a possible query result.
internal node	A tree node is an internal node if it has at least one child.
leaf	A tree node is a leaf if it has no children. Leaves are also called external nodes.
$\lg x$	The base 2 logarithm of x.
$\ln x$	The natural logarithm of x.
$\log x$	A logarithm of x where the base is greater than one but otherwise unspecified.
$\log_b x$	The base b logarithm of x.
local page size	In a partitioned tree, the local page size of a node is the number of nodes in the subtree rooted at the node that are on the same page as the node.
MBRAM	A machine model allowing table lookup and pointer dereferencing operations as well as basic arithmetic operations. See page 10 for a full definition.

overflow node	A node inserted into the CPT to handle a skip field that requires more bits than are available in a single node. See page 52 for more information.
page	One contiguous section of a tree grouped as part of a partition.
page height	In a partitioned tree, the page height of a node is the maximum number of pages that occur on any path from the node to a leaf in its sub-tree. The page height of a partitioned tree is the page height of the root of the tree.
parent	The parent of a tree node is the unique node having that node as a child.
partition	A breakdown of a tree into contiguous pieces, called pages, each of which can be represented in a single block.
PATRICIA tree	A searching structure exploiting the binary encoding of the keys. Given a set of unique keys and a binary encoding of the keys, a PATRICIA tree is a search tree in which each leaf contains one of the keys and each internal node is labelled with the position of a bit that distinguishes the keys in the left sub-tree from those in the right subtree. We use the first bit that is not identical in all the keys in a sub-tree to partition the keys into the left and right sub-trees.
PAT tree	A suffix index created by using a PATRICIA tree to search the binary encodings of the suffixes of the text. See Figure 1.6 on page 16 for an example of a PAT tree.

prefix code	A code such that no code value is a prefix of any other code value.
rank(x)	A function that computes the number of 1s to the left of and including position x in a bitmap. We also refer to the rank of the bit representing a node in a level order encoded tree as the rank of the node.
root	The root is the unique node of a tree having all other nodes in the tree as descendents. See page 8 for a discussion of trees.
seek time	For a disk based media the seek time is the time taken to move the read/write head to the correct location for a transfer. In this thesis we include other factors such as rotational latency in the seek time.
select(x)	A function that computes the location of the x'th 1 in a bitmap.
sibling	A tree node is a sibling of another node if they have the same parent.
skip value	The label on an internal node of a PATRICIA or PAT tree indicating the number of bits to skip to obtain the bit tested at the node.
stemming	The reduction of words to a canonical form for the purposes of searching. Word suffixes such as "s," "ing" and "ed" and occasionally prefixes are often removed during searching if the exact form of the word is not important for the application.

stop word	Any common word that is ignored during searching. Words such as "to," "the," "as" and the various conjugations of "to be" are typically ignored during searching because they convey little information compared to other words in a query. Ignoring stop words in the index can significantly reduce the size of an index.
strictly binary tree	A binary tree is strictly binary if every internal node has exactly two children.
suffix	A substring of a string that extends from a given character position to the end of the string.
suffix array	A suffix index created by lexicographically sorting all suffixes of the text and storing the ordered list of suffix offsets.
suffix index	An index that operates by searching the set of suffixes of the document.
suffix tree	A suffix index created by removing all degree one (straight through) nodes of a suffix trie. See Figure 1.5 on page 16.
suffix trie	A suffix index created by building a trie on the set of suffixes of the text. See Figure 1.3 on page 13.
transfer time	The time taken actually transferring data to or from secondary storage, not including the seek time.
trie	A trie is a search tree in which each leaf contains one string key and each edge has a single character label. A key occurs in the sub-tree rooted at an internal node if and only if the concatenation of the labels on the path from the root to the node are a prefix of the key. Each internal node has children for each continuation of its prefix that leads to one of the keys.

$v_2(x)$	The number of ones in the binary representation of x.
wide bus	A model of computation that assigns a cost of one to base operations on blocks of bits of size $\log n$.
word index	An index that restricts query answers to start and end on a word boundary, or, in the case of a suffix index, an index that restricts query answers to start on a word boundary.
worst case optimal	A tree partitioning is worst case optimal if it minimizes the maximum number of pages accessed when travelling from the root to any leaf.