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

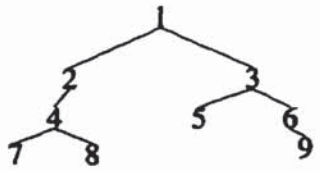

Figure 2.1: Sample Binary Tree

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

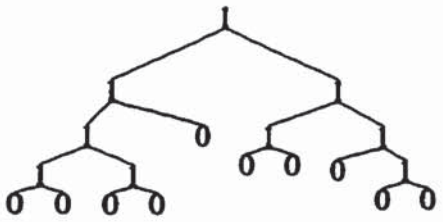

Figure 2.2: Labelled Extended Tree

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 $\textbf{select}(6 = \lfloor \frac{13}{2} \rfloor) = 7$ 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

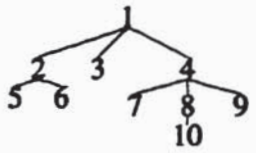

Figure 2.3: Sample General Tree

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

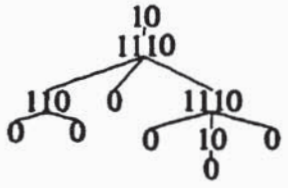

Figure 2.4: Unary Labelling of Sample Tree

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 **rank**(i) for every i a multiple of $\lceil \lg n \rceil^2$.* A second auxiliary directory contains **rank'**(j) for j a multiple of $\lceil \lg n \rceil$ within each subrange where **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: **rank**(x) is calculated by locating the correct first auxiliary directory entry, at position $\lfloor \frac{x}{\lceil \lg n \rceil^2} \rfloor$, and the correct second level entry, at position $\lfloor \frac{x}{\lceil \lg n \rceil} \rfloor$. Adding these two values gives the rank of the first bit in a $\lceil \lg n \rceil$ sized range. The final component of **rank**(x) could be computed by scanning x mod $\lceil \lg n \rceil$ 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 $\left\lceil n^{\frac{1}{c}} \right\rceil \lceil \lg \lceil \lg n \rceil \rceil$ bits. Hence the storage required in addition to the original bit map is $\frac{2n \lg \lg n}{\lg n} + O(\frac{n}{\lg n})$. $\mathcal{QED}$

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. *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:

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

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
$2\,\textbf{rank}(\textbf{child}(x,1))$. 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 $0 \leq i \leq n$, $v_2(i+1) + v_2(n-i) = v_2(n+1) + k$ 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

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

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. *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

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

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:

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

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 100101011010101011. 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

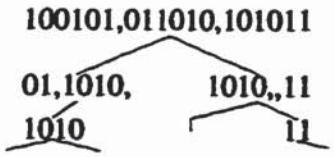

Figure 2.5: Tree Encoding

```
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 $\left\lfloor \frac{n+1}{4} \right\rfloor$. 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.