Order Number 8918056

Succinct static data structures

Jacobson, Guy Joseph, Ph.D.

Carnegie-Mellon University, 1988

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Succinct Static Data Structures

Guy Jacobson

January 1989

CMU-CS-89-112

Submitted to Carnegie Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored in part by an Amoco Fellowship, and in part by the National Science Foundation under contract number CCR-8352081.

Computer Science Department

THESIS

SUCCINCT STATIC DATA STRUCTURES

Guy Joseph Jacobson

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Merrick L. Furst — MAJOR PROFESSOR — 9/9/88 — DATE

A. Habermann — DEPARTMENT HEAD — 30 Jan '89 — DATE

APPROVED:

A. G. Jordan — PROVOST — 2/24/89 — DATE

Abstract

Data *compression* is when you take a big chunk of data and crunch it down to fit into a smaller space. That data is put on ice; you have to un-crunch the compressed data to get at it. Data *optimization*, on the other hand, is when you take a chunk of data *plus* a collection of operations you can perform on that data, and crunch it into a smaller space while retaining the ability to perform the operations efficiently.

This thesis investigates the problem of data optimization for some fundamental *static* data types, concentrating on linked data structures such as trees. I chose to restrict my attention to static data structures because they are easier to optimize since the optimization can be performed off-line.

Data optimization comes in two different flavors: *concrete* and *abstract*. Concrete optimization finds minimal representations within a given implementation of a data structure; abstract optimization seeks implementations with guaranteed economy of space and time.

I consider the problem of concrete optimization of various pointer-based implementations of trees and graphs. The only legitimate use of a pointer is as a reference, so we are free to map the pieces of a linked structure into memory as we choose. The problem is to find a mapping that maximizes overlap of the pieces, and hence minimizes the space they occupy.

I solve the problem of finding a minimal representation for general unordered trees where pointers to children are stored in a block of consecutive locations. The algorithm presented is based on weighted matching. I also present an analysis showing that the average number of cons-cells required to store a binary tree of n nodes as a minimal binary DAG is asymptotic to $n/(\frac{\pi}{8}\lg n)^{1/2}$.

Methods for representing trees of n nodes in $O(n)$ bits that allow efficient tree-traversal are presented. I develop tools for abstract optimization based on a succinct representation for ordered sets that supports ranking and selection. These tools are put to use in a building an $O(n)$-bit data structure that represents n-node planar graphs, allowing efficient traversal and adjacency-testing.

Succinct Static Data Structures

[Thesis Summary]

Guy Jacobson

Computer Science Department
Carnegie-Mellon University
Pittsburgh Pennsylvania 15213

Abstract

Data *compression* is when you take a big chunk of data and crunch it down to fit into a smaller space. That data is put on ice; you have to un-crunch the compressed data to get at it. Data *optimization*, on the other hand, is when you take a chunk of data *plus* a collection of operations you can perform on that data, and crunch it into a smaller space while retaining the ability to perform the operations efficiently.

This thesis investigates the problem of data optimization for some fundamental *static* data types, concentrating on linked data structures such as trees. I chose to restrict my attention to static data structures because they are easier to optimise since the optimization can be performed off-line.

Data optimisation comes in two different flavors: *concrete* and *abstract*. Concrete optimization finds minimal representations within a given implementation of a data structure; abstract optimization seeks implementations with guaranteed economy of space and time.

I consider the problem of concrete optimisation of various pointer-based implementations of trees and graphs. The only legitimate use of a pointer is as a reference, so we are free to map the pieces of a linked structure into memory as we choose. The problem is to find a mapping that maximizes overlap of the pieces, and hence minimises the space they occupy.

I solve the problem of finding a minimal representation for general unordered trees where pointers to children are stored in a block of consecutive locations. The algorithm presented is based on weighted matching. I also present an analysis showing that the average number of cons-cells required to store a binary tree of n nodes as a minimal binary DAG is asymptotic to $n/(\frac{\pi}{8}\lg n)^{1/2}$.

Methods for representing trees of n nodes in $O(n)$ bits that allow efficient tree-traversal are presented. I develop tools for abstract optimization based on a succinct representation for ordered sets that supports ranking and selection. These tools are put to use in a building an $O(n)$-bit data structure that represents n-node planar graphs, allowing efficient traversal and adjacency-testing.

1 What is data *optimization*?

Small is beautiful. It is good when a piece of data can be made smaller. It is a bad, however, when this reduction in size is accompanied by a reduction in accessibility as well, but this is the usual compromise made in classical data compression. Sometimes such a compromise is unacceptable.

The job of an optimizing compiler is to take a specification of operations to be performed on data and produce a functionally equivalent specification that is somehow *better* than the original. An equivalence between the original operations and the optimized operations is necessary; given the same data, the two versions must behave identically. An optimizing compiler is absolutely uncompromising in this regard.

I call transformations that make data smaller, while preserving important functionality, *data optimizations*. A compiler must be adamant about its optimization, because the computer is hard-wired for a certain set of operations. A fixed computer program that accesses a large static external data structure also assumes a particular concrete representation for the data it accesses. The analogy of a program optimizer is a data optimizer, which reduces the size of external data structures in a way that is transparent to the program.

My thesis systematically examines the problems of data optimization for some basic data types and combinatorial objects. Special attention is devoted to the optimization of linked data structures, since these data structures have been traditionally neglected in the study of data compression. I place emphasis both on constructing and analyzing provably efficient algorithms *and* on the practical issues of real-world implementation.

Data optimization is much easier when we can sit back and do it off-line. I have therefore restricted my attention to static data structures. Extending the work I present to dynamic structures (where possible) would be the subject of another thesis.

2 Concrete optimization

The transparent transformation that reduces the size of our data can only be possible if we know how the program is going to access the data. Thinking of the data structure as a data type with a particular set of query operations already implemented, we can change the data so that the program does not see any difference. I call this type of transformation *concrete* data optimization, since the program that accesses the data is considered wholely immutable: the low-level operations are have *concrete* implementations. But because we know the specifications of these operations, we have the

freedom to change the data, as long as we do it in such a way that the behavior of any program that uses the operations does not change. Concrete optimization is most successful when there are many equivalent (from the limited point of view of the program) patterns of bits in memory that represent the same data object. We are then free to optimize by choosing a succinct pattern.

Problems of concrete optimization are optimization problems in the classical sense. We are given a concrete representation scheme for our abstract data type, along with a collection of routines that access the data in the given scheme. Our task is to devise an algorithm that accepts an object of the given type and finds a succinct representation within the scheme. Ideally, we strive for an efficient algorithm that finds minimal representations. Sometimes we have to settle for an algorithm that finds close-to-minimal representations, or one that produces provably succinct representations in the average case.

2.1 A concrete model of linked data structures

With linked data structures, there are many different, but equivalent, patterns of bits that represent a particular object. Let's adopt a simple but general model for this class of structures. Our linked data structures consist of a collection of *nodes*. Each node occupies a contiguous block of memory. The nodes do not necessarily have a fixed size or layout. The nodes contain one or more pointers to other nodes, and they may contain other data as well. We are free to do as we please with the other information within a node, but we may only move from node to node by dereferencing a pointer.

The pointers are simply absolute addresses in memory. The specifications of the abstract data type do not permit arbitrary manipulations of these pointers; the operations may only dereference them. Because the program is not allowed to use the numerical values of the pointers, the mapping of the nodes of the linked structure to memory is up to us. The standard scheme for representing a linked structure partitions memory so that each node of the structure occupies a distinct block. The nodes do not overlap. But we are free choose a mapping of the nodes to memory locations that *does* overlap, to minimize space. When our chosen mapping allows two nodes of our linked data object to share the same memory locations, we save space.

3 Abstract optimization

The other type of data optimization allows the optimizer some control over the lowest level access primitives of the abstract data type. Here, the abstract specifications of the operations are fixed,

but their implementation is up to the optimizer. I call this *abstract* data optimization.

When doing abstract optimization, we actually design the format of the data structure. (In concrete optimization, this format is already fixed.) Additionally, we must implement the primitive operations of the data type. It is naturally desirable that our implementation isn't much slower than an implementation that uses a natural, but less space-efficient, format.

This is the paradigm of abstract optimization:

- We start with the specification for a static abstract data type C. (We will abuse notation slightly and also use C to refer to the set of all objects of type C.) Typically, there will be a *natural* implementation of C whose performance is satisfactory in execution time, but wasteful of space.
- We choose a natural size parameter n, which partitions the class into subclasses C_n.
- Combinatorially, we determine the number of elements in C_n as a function of n. This computation suggest a *canonical* implementation that simply maps each member of C_n into a different integer from 1 to $||C_n||$, represented in binary. While this implementation is optimal in space, it does not support the desired operations efficiently in time.
- We devise a new representation for C, and implementations of the primitive operations, that has the space-efficiency of the canonical implementation, and the time-efficiency of the natural implementation. This is the real optimization step.

The quality of the optimization depends on how closely, in the last step, we are able to simultaneously approach the space- and time-efficiencies of the canonical and natural implementations. We may allow ourselves a reduction in performance by a constant factor, especially in time-efficiency. When it is not possible to be efficient in both time and space simultaneously, we can explore the tradeoffs involved.

3.1 An abstract model of linked data structures

Linked data structures are *linked* because there are *pointers* that associate the nodes with each other. In natural implementations (as in the concrete model proposed earlier) these pointers are absolute addresses: integer indices into memory. The most natural size parameter is often the number of nodes[1] n. A structure with n nodes will have at least $O(n)$ pointers, and each pointer

[1] Although some structures with more than $O(1)$ pointers per node may be more naturally represented by the total number of pointers

needs to be able to address at least n different locations. To have this much addressing power, we need to store about $\lg n$ bits per pointer. This means that the natural implementation will occupy $O(n \log n)$ bits in memory. For many significant families of linked data structures, this is much more space than the information-theoretic bound of $\lg \|C_n\|$.

Trees are a good example of such data structures. The number of unlabeled trees with n nodes is bounded by k^n (with k depending on the exact variety of tree we are talking about), so the number of bits required to store a tree is only linear in the number of nodes in the tree. It seems like a great waste to use $O(n \log n)$ bits when $O(n)$ will do. In fact, there is a large literature on encoding trees economically as strings of bits. But this literature devotes itself only to the encoding and decoding of trees to bits. No suggestion of performing the usual tree-traversal operations directly on these efficient encodings is found therein. The design of efficient encodings for trees that allow speedy traversal is a basic goal of abstract optimization for linked structures.

How can we overcome the $\log n$ bits-per-pointer barrier? For some types of linked structures, we cannot. General graphs, for example, require $O(m \log n)$ bits (where m is the number of edges) by a simple counting argument. But for others (like trees) this barrier can be overcome. Two possible approaches are:

1. Take advantage of the special form of the data structures involved to reduce the space for the pointers. Even if we need to address n different locations, we can use the classical techniques of data compression (like entropy-coding) to reduce the total space. Remember that the *total* space used is the quantity of interest: we can amortize a few expensive pointers if most of them are cheap.

2. Do away with the need for pointers entirely. Use a radically different encoding that is both space-efficient and traversable.

4 Related work

Make frequent utterances terse at the expense of making infrequent ones verbose: this is a basic concept of classical data compression. Huffman coding, for example, takes a string of tokens over a finite alphabet and produces a string of bits whose length is close to the entropy (in the information-theoretic sense) contained in the tokens. As an abstract data optimization technique, Huffman coding (and other related types of entropy coding) only efficiently support the feeble operation of sequentially accessing the tokens starting from the beginning. Furthermore, entropy

coding techniques only apply when the number of tokens in the stream is much greater than the number of distinct token values. When the natural units of data are chosen from a very large alphabet and may not occur more than a few times in the entire data structure (as is the case with linked data structures), these methods fail.

4.1 Concrete optimization

A well known example of concrete data optimization is the finite-state machine minimization algorithm due to Huffman and Moore. A finite state machine is a kind of labeled graph, so finding an equivalent machine with fewer states is a concrete data optimization of a linked data structure.

Often it is convenient to structure a large database hierarchically, as a tree. If the database is static, and there are choices in the layout of the tree that affect the storage requirements, we can perform a concrete optimization to save space.

A *trie* is a hierarchical data structure that allows relatively efficient lookup of records with multiple keys. Nodes in the trie represent subsets of the records. The root represents the entire set, and the leaves represent individual records. The sets represented by the children of a node n form a partition of the set represented by n into equivalence classes under equality of a particular key.

When all the records have the same set of keys, we are free to choose which key to use to partition each node. The total size of the resulting trie will depend on these choices. A number of authors have investigated the problem of minimizing the space required to store tries. Although Comer and Sethi prove that the problem is NP-Complete, Comer exhibits a simple heuristic that seems to performs well on average.

This shows another way of demonstrating the effectiveness of a particular concrete data optimization. Even though an efficient algorithm to solve the optimization problem exactly could not be found, Comer was able to get good results from his heuristic for plausible input distributions. He also shows that there are classes of tries for which his heuristic does not perform well.

4.2 Abstract optimization

Work that is allied to abstract data compression falls into two broad categories:

1. Design of space-efficient data structures.

2. Enumeration/Encoding of combinatorial objects.

Abstract data compression unites these two categories for abstract data types that are also combinatorial objects. Because many data types differ only in their *dynamic* properties (for example, a static stack, queue or list is merely a sequence), the useful static data structures are are relatively few in number.

The data-structure Designers are concerned with being efficient in time as well as space, but they generally do not account for the space they use very strictly. They usually count the space used in words rather than bits. They do not strive to achieve the optimum space-efficiency derived from information theory—they merely seek to improve previous results.

On the other hand, the Encoders are acutely aware of the minimum number of bits required to represent objects of a given size. But they do not consider how to operate efficiently on these representations directly, without first converting them back into a natural representation.

Their succinct representations usually can be classified into three categories:

canonical This is the best we can hope for. This is a mapping from C_n into the integers $1 \ldots \|C_n\|$. The resulting integer is then encoded as a $\lg \|C_n\|$-bit binary number. This type of representation is always *possible*, the trick is to compute the mapping and its inverse efficiently.

asymptotically optimal This is a little worse than canonical. This is a mapping from C_n into a bit string of length $\lg \|C_n\| \cdot (1 + o(1))$. Some wasted space is allowed in this type of representation, but as n grows, the fraction of waste must vanish.

linear This maps C_n into bit strings whose length is $O(\log \|C_n\|)$. We may have to settle grudgingly for this.

In abstract optimization, we will not insist on canonical representations, but we do value asymptotic optimality.

5 Cost metrics for abstract optimization

Suppose we have an abstract class C_n of static data objects, (for example, the set of trees with n nodes) and a set of operations S (like `car` and `cdr`) that examine a data object but do not modify it. Each member of C_n can be viewed as a collection of partial functions, one function corresponding to each operation in S. The domain and range of these functions can be either predefined data

types (like `integer` or `boolean`) or they can be *indices*. These indices are meant to be the abstract analogues of pointers; they can only be used and returned by operations in S.

In simple tree example cited, we would have a domain `node` referring to nodes of the tree, and the following abstract operations:

- a function of no arguments (a constant) `root` returning `node`.
- two functions, `car` and `cdr`, mapping `node` to `node`.
- a function `null` mapping `node` to `boolean`.

An *implementation* of an abstract class provides a mapping from elements of C_n into a read-only memory, and a program for each operation in S that references this memory. An implementation also provides a mapping between elements of the index domains and small pieces of memory. All of these mappings are strictly internal to the implementation, and cannot be referenced by a program that makes use of the data type.

The abstract data types I study have natural implementations that use too much space. Optimization means making something better. A *better* implementation of these data types has the same functionality as the natural implementation, but uses less space. The trick of abstract optimization is to trim the fat in the data without slowing down the operations too much. How much space has been saved? How much slower is the optimized implementation? To provide meaningful answers to such questions, we need to have a model of computation that defines precise cost metrics for space and time, and that is realistic about computers' capabilities.

5.1 Space metrics

The (worst-case) space cost of an implementation is simply the maximum length of any of the bit-strings representing an element in C_n. This is a strictly log-cost accounting of space. Since space-efficiency is the primary concern here, I cannot afford to be sloppy and measure space in *words*, which hold an unspecified amount of information. It is always possible to make use of all the bits in a computer word.

Bits are universal. While it *is* possible to buy a computer that does more work per unit time, it is *not* possible to buy a computer that stores more per bit. In other words, the time required for a given operation can only be bounded by a functional form, whereas the space required can be bounded absolutely. It would be foolish to use any metric for space other than bits.

5.2 Time metrics

The choice of metric for time is not so clear-cut. The unit-cost model of computation is a popular accounting metric for time, and with good reason. This model usually has the most realistic correspondence to observed running time. The pitfalls of the unit-cost model when numbers get large are well known. Less obvious, but just as nasty, are the architecture-specific shortcomings of this metric. The unit-cost model assumes some kind of word-size bit parallelism exists within the circuits of a computer. When the logarithms of the numbers involved stay below the word size, it is reasonable to expect to perform *certain* operations with this degree of parallelism. But the circuits in any given computer are fixed, so we may be out of luck when we try to coerce a computer into performing a particular word-size operation for which it is ill-suited.

Let's look at a specific example of this phenomenon. Suppose a critical step of an algorithm involves counting the number of 1 bits in a certain binary number. Let's assume that the typical number n we are dealing with is small enough to fit in a single computer word. How much time should we account for this bit-counting operation? If we get to choose, we can use a CDC computer with a `bit-count` instruction. It would seem reasonable then to assess a cost of one to bit counting. But many other computers lack an explicit `bit-count` instruction. Should we loop through testing the bits of the number n, and charge $\lg n$? Should we use a clever sequence of $\lg \lg n$ shifts, masks, and additions to compute the bit-count of n? Or should we break each word into k chunks, keep a table of size $\sqrt[k]{n}$ of precomputed bit-counts, and charge k (generally the most efficient scheme in practice)? Further complications arise if our computer doesn't have a multi-place shift instruction. The best implementation of this operation, and hence its accounting, depends on the architecture of our computer.

To avoid these processor-specific pitfalls, I restrict my attention to metrics based on bus transactions between the processor and the static data. I will charge for, and only for, each reference the processor makes to the data. The processor is allowed to perform arbitrary computations at no cost with the data it has already has in hand. I am measuring I/O complexity.

By varying the bandwidth of the buses and the costs of transactions, we get different metrics. Two that I especially like, and use extensively in the thesis are:

data-bits model We assume that the data bus from memory to the processor is only one bit wide. We simply count the number of bit-accesses performed to get the time cost.

wide-bus model We assume that the data bus is $\lg N$ bits wide, where N is the size of the memory. We can fetch $\lg N$ bits from consecutive memory locations at unit cost.

In both models we assume that the address bus is sufficiently wide to address any bit contained therein.

The data-bits model has simplicity as its main advantage. There is a strong analogy between accounting time as bit-accesses here and accounting the time used by sorting algorithms in element comparisons. If nothing else, our time metric gives a very reasonable lower bound on achievable asymptotic performance, as long as the bandwidth of the data bus is fixed. Two disadvantages of this model are that it does not reflect the inherent word-size parallelism in the data buses of real computers, and that it does not take account of the bounded bandwidth of the *address* bus.

The wide-bus model, on the other hand, is more realistic about the inherent parallelism in the buses of computers. Still, there is something a bit disturbing about the size of the bus growing along with the size of the data.

6 Thesis outline

I conclude this summary with a chapter-by-chapter outline of the results presented in the technical chapters of the thesis.

Chapter 2: Treats concrete optimization of trees and other linked data structures. Optimization under various common implementations are considered. The major results are:

- A polynomial-time algorithm, based on weighted matching, that finds a minimal representation of a general unordered tree when the pointers to children are stored in a block of consecutive locations.
- An analysis showing that the average number of cons-cells required to store a binary tree of n nodes as a minimal binary DAG is asymptotic to $n/(\frac{\pi}{8}\lg n)^{1/2}$.

Chapter 3: Presents a formal model for abstract optimization. An argument for a particular choice of metrics for space and time is put forth.

Chapter 4: Discusses a class of recursive representations for trees in linear space. The $\log n$ bits-per-pointer barrier is broken by a using a variable-length encoding for the

pointers and amortizing the space over the whole tree. An optimal representation in this class is identified, and its efficiency is partially analyzed. The conclusion is that no representation in this class could be asymptotically optimal.

Chapter 5: I develop a number of general tools for abstract optimization based on efficient data structures for ordered sets and parenthesis balancing. The data structure for ordered sets support the operations rank and `select`. Three applications of these tools are presented:

1. Random-access Huffman coding: how to prepare an index that lets us find the mth symbol in a file of n Huffman-coded symbols efficiently. The extra space required for the index is $o(n)$.
2. Trees in asymptotically optimal space: this addresses the same problem as chapter 4, but obtains an optimal constant factor.
3. Planar graphs in linear space: how to store a planar graph on n nodes using only $O(n)$ bits. The operations of adjacency testing and searching (neighbor enumeration) are efficiently supported.

Acknowledgements

First, let me thank the members of my committee. Merrick Furst, my advisor, stood up for me long after most advisors would have given up. Danny Sleator helped steer my wayward brain back on track when I steered from the true course. His guidance and his friendship are very valuable to me. Thanks also go to Bob Sedgewick, my outside member, and to Rick Statman, who was there when I needed him.

I thank my parents, who gently prodded me to finish, and all my friends here at Carnegie Mellon, who made it so easy for me to stay.

But most of all, I thank my wonderful wife WenLing for her encouragement and her patience.

Contents

1 Introduction 1

1.1 What is data *optimization*? . . . 1
1.2 Concrete optimization . . . 2
1.3 Abstract optimization . . . 3
1.4 Related work . . . 5
1.5 Thesis outline . . . 9

2 Concrete Optimized Trees 11

2.1 A taxonomy of trees . . . 12
2.2 Binary trees as cons-cells . . . 13
2.3 An average case analysis . . . 14
2.4 Binary trees as cdr-coded lists . . . 25
2.5 General trees as binary trees . . . 26
2.6 Unordered trees as cdr-coded lists . . . 28
2.7 Implementation details . . . 32
2.8 Matching . . . 36
2.9 An application: English lexicon tree . . . 37
2.10 Future work . . . 38

3 Abstract Optimization 40

3.1 Space metrics . . . 41
3.2 Time metrics . . . 42
3.3 Binary trees: an example . . . 43

4 Abstract Optimized Trees 45

4.1 Introduction 45

4.2 Encoding trees in linear space 47

4.3 Lowering the constant factor 52

4.4 Practical concerns 63

4.5 Conclusions 70

5 Techniques of abstract optimization 73

5.1 Ranking and selection 73

5.2 Random-access Huffman coding 81

5.3 Trees in optimal linear space 82

5.4 Planar graphs in linear space 87

List of Figures

2.1 Splicing trees 16
2.2 Graph of S_n for $1 \leq n \leq 1000$ 18
2.3 Graph of K_{nm} and K'_{nm} for $n = 1000, m \leq 20$ 21
2.4 Graph of K_{nm} and K'_{nm} for $n = 1000, m > 20$ 21
2.5 A straightforward translation of a tree into an array. 29
2.6 Reducing the space for tree storage. 30
2.7 The space-minimization algorithm: an example. 33
2.8 Two types of augmenting paths 37

4.1 Balanced parentheses as trees 46
4.2 Recursive layout of a binary tree 48
4.3 Left leaning tree 49
4.4 Layout of a binary tree with `Left-child-first?` bits 50
4.5 Layout of an n-node binary tree with generalized R_n 53
4.6 Construction bounding the lower limit of B_n/n 58
4.7 Layout of a binary tree using `child-empty?` bits 66
4.8 Layout of a binary tree to allow upward traversal 67

5.1 A two-level directory for set ranking. 76
5.2 A binary tree and its implicit bitmap. 83
5.3 Level-order binary marked representation. 83
5.4 Level-order unary degree sequence representation. 86
5.5 A structure to balance parentheses 91
5.6 One-page graphs as balanced parentheses 92

Chapter 1

Introduction

1.1 What is data *optimization*?

Small is beautiful. It is good when a piece of data can be made smaller. It is bad, however, when this reduction in size is accompanied by a reduction in accessibility as well, but this is the compromise made in classical data compression. Sometimes such a compromise is unacceptable.

The job of an optimizing compiler is to take a specification of operations to be performed on data and produce a functionally equivalent specification that is somehow *better* than the original. An equivalence between the original operations and the optimized operations is necessary; given the same data, the two versions must behave identically. An optimizing compiler is absolutely uncompromising in this regard.

I call transformations that make data smaller, while preserving important functionality, *data optimizations*. A compiler must be adamant about its optimization, because the computer is hard-wired for a certain set of operations. A fixed computer program that accesses a large static external data structure also assumes a particular representation for the data it accesses. The analogue of a program optimizer is a data optimizer, which reduces the size of external data structures in a way that is transparent to the program.

My thesis systematically examines problems of data optimization for some basic data types and combinatorial objects. Special attention is devoted to the optimization of linked data structures, since these data structures have been traditionally neglected in the study of data compression. I place emphasis both on constructing and analyzing provably efficient algorithms *and* on the practical issues of real-world implementation.

Data optimization is much easier when we can sit back and do it off-line. I have therefore re-

stricted my attention to static data structures. Extending the work I present to dynamic structures (where possible) would be the subject of another thesis.

1.2 Concrete optimization

The transparent transformation that reduces the size of our data can only be possible if we know how the program is going to access the data. Thinking of the data structure as a data type with a particular set of query operations already implemented, we can change the data so that the program does not see any difference. I call this type of transformation *concrete* data optimization, since the program that accesses the data is considered wholely immutable: the low-level operations are have *concrete* implementations. But because we know how these operations are implemented, we have the freedom to change the data, as long as we do it in such a way that the behavior of any program that uses the operations does not change. Concrete optimization is most successful when there are many equivalent (from the limited point of view of the program) patterns of bits in memory that represent the same data object. We are then free to optimize by choosing a succinct pattern.

Problems of concrete optimization are optimization problems in the classical sense. We are given a concrete representation scheme for our abstract data type, along with a collection of routines that access the data in the given scheme. Our task is to devise an algorithm that accepts an object of the given type and finds a succinct representation within the scheme. Ideally, we strive for an efficient algorithm that finds minimal representations. Sometimes we have to settle for an algorithm that finds close-to-minimal representations, or one that produces provably succinct representations in the average case.

1.2.1 A concrete model of linked data structures

With linked data structures, there are many different, but equivalent, patterns of bits that represent a particular object. Let's adopt a simple but general model for this class of structures. Our linked data structures consist of a collection of *nodes*. Each node occupies a contiguous block of memory. The nodes do not necessarily have a fixed size or layout. The nodes contain one or more pointers to other nodes, and they may contain other data as well. We are free to do as we please with the other information within a node, but we may only move from node to node by dereferencing a pointer.

The pointers are simply absolute addresses in memory. The specifications of the abstract data type do not permit arbitrary manipulations of these pointers; the operations may only dereference

them. Because the program is not allowed to use the numerical values of the pointers, the mapping of the nodes of the linked structure to memory is up to us. The standard scheme for representing a linked structure partitions memory so that each node of the structure occupies a distinct block. The nodes do not overlap. But we are free choose a mapping of the nodes to memory locations that *does* overlap, optimizing to minimize space. When our chosen mapping allows two nodes of our linked data object to share the same memory locations, we save space.

1.3 Abstract optimization

The other type of data optimization allows the optimizer some control over the lowest level access primitives of the abstract data type. Here, the abstract specifications of the operations are fixed, but their implementation is up to the optimizer. I call this *abstract* data optimization.

When doing abstract optimization, we actually design the format of the data structure. (In concrete optimization, this format is already fixed.) Additionally, we must implement the primitive operations of the data type. It is naturally desirable that our new implementation isn't much slower than an implementation that uses a natural, but less space-efficient, format.

This is the paradigm of abstract optimization:

- We start with the specification for a static abstract data type C. (We will abuse notation slightly and also use C to refer to the set of all objects of type C.) Typically, there will be a *natural* implementation of C whose performance is satisfactory in execution time, but wasteful of space.
- We choose a natural size parameter n, which partitions the class into subclasses C_n.
- Combinatorially, we determine the number of elements in C_n as a function of n. This computation suggest a *canonical* implementation that simply maps each member of C_n into a different integer from 1 to $||C_n||$, represented in binary. While this implementation is optimal in space, it does not support the desired operations efficiently in time.
- We devise a new representation for C, and implementations of the primitive operations, that has the space-efficiency of the canonical implementation, and the time-efficiency of the natural implementation. This is the real optimization step.

The quality of the optimization depends on how closely, in the last step, we are able to simultaneously approach the space- and time-efficiencies of the canonical and natural implementations. We

may allow ourselves a reduction in performance by a constant factor, especially in time-efficiency. When it is not possible to be efficient in both time and space simultaneously, we can explore the tradeoffs involved.

1.3.1 An abstract model of linked data structures

Linked data structures are *linked* because there are *pointers* that associate the nodes with each other. In natural implementations (as in the concrete model proposed earlier) these pointers are absolute addresses: integer indices into memory. The most natural size parameter is often the number of nodes[1] n. A structure with n nodes will have at least $O(n)$ pointers, and each pointer needs to be able to address at least n different locations. To have this much addressing power, we need to store about $\lg n$ bits per pointer. This means that the natural implementation will occupy $O(n \log n)$ bits in memory. For many significant families of linked data structures, this is much more space than the information-theoretic bound of $\lg \|C_n\|$.

Trees are a good example of such data structures. The number of unlabeled trees with n nodes is bounded by k^n (with k depending on the exact variety of tree we are talking about), so the number of bits required to store a tree is only linear in the number of nodes in the tree. It seems like a great waste to use $O(n \log n)$ bits when $O(n)$ will do. In fact, there is a large literature on encoding trees economically as strings of bits. (R. C. Read[41] provides a good review of these.) But this literature devotes itself only to the encoding and decoding of trees to bits. No suggestion of performing the usual tree-traversal operations directly on these efficient encodings is found therein. The design of efficient encodings for trees that allow speedy traversal is a basic goal of abstract optimization for linked structures.

How can we overcome the $\log n$ bits-per-pointer barrier? For some types of linked structures, we cannot. General graphs, for example, can be shown to require $O(m \log n)$ bits (where m is the number of edges) by a simple counting argument. But for others (like trees) this barrier can be overcome. Two possible approaches are:

1. Take advantage of the special form of the data structures involved to reduce the space for the pointers. Even if we need to address n different locations, we can use the classical techniques of data compression (like entropy-coding) to reduce the total space. Remember that the *total*

[1] Although some structures with more than $O(1)$ pointers per node may be more naturally represented by the total number of pointers

space used is the quantity of interest: we can afford a few expensive pointers if most of them are cheap.

2. Do away with the need for pointers entirely. Use a radically different encoding that is both space-efficient and traversable.

1.4 Related work

Make frequent utterances terse at the expense of making infrequent ones verbose: this is a basic concept of classical data compression. Huffman coding, for example, takes a string of tokens over a finite alphabet and produces a string of bits whose length is close to the entropy (in the information-theoretic sense) contained in the tokens. As an abstract data optimization technique, Huffman coding (and other related types of entropy coding) only support the feeble operation of sequentially accessing the tokens starting from the beginning. Furthermore, entropy coding techniques only apply when the number of tokens in the stream is much greater than the number of distinct token values. When the natural units of data are chosen from a very large alphabet and may not occur more than a few times in the entire data structure (as is the case with linked data structures), these methods fail.

1.4.1 Examples from the literature

The term *data optimization* is my own, but researchers have sought useful space-efficient data structures since the dawn of computers. Let me now review examples of previous work that is related in subject or style to the work presented in this thesis. These examples are grouped by data type.

Sets and sequences

The most fundmental data types are collections, ordered and unordered: sets and sequences. The problem of abstract optimization of subsets of $[0 \ldots N]$ with membership queries was first addressed by Minsky and Papert[37, pages 215-225]. They give informal bounds on the time required to answer membership queries with various amounts of available space. Elias[12], improves and tightens these bounds. Later, Elias and Flower[14] investigate more complex retrieval problems with sets of natural numbers. Yao[50] considers representations in which a set of cardinality n is maintained by a table of n or $n+1$ values from $[0 \ldots N]$. He concludes that keeping a sorted table of set elements is

asymptotically optimal when the table has n items, but becomes suboptimal when the table size increases slightly.

Quite a few authors, including Fredman[17] and Tarjan and Yao[45], have investigated perfect hashing. This abstract optimization stores a sparse set with access times within a constant factor of those for a straightforward representation (a bitmap), using only a little more space than is informationally necessary. Recently, Fiat *et al.*[16] investigated how non-oblivious hashing, where unsuccessful probes determine future probing strategy. This technique can store sparse sets with n members using only $n+O(\log n)$ memory locations, and still support $O(1)$ worst-case lookup times. These hashing schemes account for the space they use in memory locations rather than bits, and assume that a memory location is large enough to hold any member of the set. Of course, hashing is not an efficient technique when our sets are not sparse. If we are interested in membership queries only, very dense sets can be stored efficiently as a bitmap.

Linked structures: concrete optimization

A well known example of concrete data optimization is the finite-state machine minimization algorithm due to Huffman[24] and Moore[38]. A finite state machine is a kind of labeled graph, so finding an equivalent machine with fewer states is a concrete data optimization of a linked data structure.

Often it is convenient to structure a large database hierarchically, as a tree. If the database is static, and there are choices in the layout of the tree that affect the storage requirements, we can perform a concrete optimization to save space.

A *trie* can be viewed as a hierarchical data structure that allows efficient lookup of records with multiple keys. Nodes in the trie represent subsets of the records. The root represents the entire set, and the leaves represent individual records. The sets represented by the children of a node n form a partition of the set represented by n into equivalence classes under equality of a particular key.

When all the records have the same set of keys, we are free to choose which key to use to partition each node. The total size of the resulting trie will depend on these choices. A number of authors have investigated the problem of minimizing the space required to store tries. Although Comer and Sethi[8] prove that the problem is NP-Complete, Comer[7] exhibits a simple heuristic that seems to performs well on average.

This shows another way of demonstrating the effectiveness of a particular concrete data opti-

mization. Even though an efficient algorithm to solve the optimization problem exactly could not be found, Comer was able to get good results from his heuristic for plausible input distributions. He also shows that there are classes of tries for which his heuristic does not perform well.

Comer's analysis of the on-average performance of his heuristic is based on simulation. It would have been better if he made a precise mathematical statement of why his heuristic was good.

Linked structures: abstract optimization

Although linked structures are not usually designed for space efficiency (and are less commonly used for static data), a number of authors have studied ways to reduce the space to store them. Work that is allied to abstract data compression falls into two broad categories:

1. Design of space-efficient data structures.
2. Enumeration/Encoding of combinatorial objects.

Abstract data compression unites these two categories for abstract data types that are also combinatorial objects. Because many data types differ only in their *dynamic* properties (for example, a static stack, queue or list is merely a sequence), the useful static data structures are are relatively few in number.

The data-structure Designers are concerned with being efficient in time as well as space, but they generally do not account for the space they use very strictly. They usually count the space used in words rather than bits. They do not strive to achieve the optimum space-efficiency derived from information theory—they merely seek to improve previous results.

On the other hand, the Encoders are acutely aware of the minimum number of bits required to represent objects of a given size. But they usually do not consider how to operate efficiently on these representations directly, without first converting them back into a natural representation.

A good example of a linked data structure designed to be space-efficient is the the *compressed trie* of Maly[35]. His tries are multiway trees with constant branching degree m at the internal nodes. Instead of storing m pointers at each internal node as is common, Maly stores only m bits, plus about one pointer's worth of extra space. This achieves significant compression of the trie with only a small penalty in execution time. The space required to store a trie of n nodes is reduced from $O(nm \log n)$ bits down to $O(n \cdot (m + \log n))$.

Chazelle[6] designed an efficient data structure to store a static range tree. A range tree is itself a data structure that represents a permutation Π of $1 \ldots n$ in a way that facilitates answering

queries about the size of the set $\{\Pi(x) \leq s \mid x \leq t\}$ as a function of the two parameters s and t (this is useful in range counting). Previous implementations of range trees required $O(n \log n)$ words of space to achieve polylog query times; Chazelle's implementation requires only $O(n)$ words. Of course, since there are $O(n \log n)$ bits of information in a permutation on n elements, each word must hold $O(\log n)$ bits. Although he worries about packing bits into words and simulating bit operations in weak machine models (and I don't), Chazelle's work is very similar in spirit to the work presented in this thesis.

There is a large body of literature on the succinct encoding of linked data structures, particularly trees. A good summary of this literature can be found in Read[41]. The tree encoders have different goals, but they all deal with trees as atomic data objects. That is, their encodings do not reflect the internal structure of the tree, and do not support operations within the tree. Once a tree is encoded, a question such as "Where is node x?" may have no satisfactory answer. Even if we can find an answer for *that* question, a question like "Where are the children of x?" cannot be answered efficiently.

For other linked structures, the literature is considerably more sparse. Turán found a way to encode planar graphs of n nodes in $12n$ bits. His encoding does not allow any useful internal operations, such as moving from one node to adjacent nodes, or testing adjacency.

Other structures

Finite groups are very basic combinatorial objects. Jerrum[26] found a representation for permutation groups on n elements needing only $O(n^2)$ space. His scheme, which supports efficient membership testing, is an an improvement over the previously known $O(n^3)$ representation of Furst *et al.*[18]. This reduction in space is an abstract optimization.

Another example of a very practical abstract data optimization is S.C. Johnson's[34] scheme for storing a sparse two-dimensional table used to store the transition tables in the LEX lexical analyzer generator. This scheme allows storage of the table in (almost) as little space as other sparse-array schemes, while allowing direct access times close to those for full two-dimensional arrays.

Boolean functions, represented as circuits, are important in the design and simulation of computer hardware. Often these circuits have a concise hierarchical description, even when they contain a great many gates. Appel[4] showed how to use a static hierarchical description, along with a single dynamic bit for each wire, to simulate a circuit efficiently. Previous simulators had required one or more pointers (with a logarithmic number of bits each) in their representation of the circuit. For

n-gate circuits with small hierarchical descriptions of size m, Appel's scheme optimizes the storage required from $n \log n$ bits down to m, without incurring a penalty in simulation time.

1.5 Thesis outline

Let me conclude this introduction with a chapter-by-chapter outline of the results presented in the technical chapters of the thesis.

Chapter 2: Treats concrete optimization of trees and other linked data structures. Optimization under various common implementations are considered. The major results are:

- An analysis showing that the average number of cons-cells required to store a binary tree of n nodes as a minimal binary DAG is asymptotic to $n/(\frac{\pi}{8} \lg n)^{1/2}$.
- A polynomial-time algorithm, based on weighted matching, that finds a minimal representation of a general unordered tree when the pointers to children are stored in a block of consecutive locations.

Chapter 3: Presents a formal model for abstract optimization. An argument for a particular choice of metrics for space and time is put forth.

Chapter 4: Discusses a class of recursive representations for trees in linear space. The $\log n$ bits-per-pointer barrier is broken by a using a variable-length encoding for the pointers. An optimal representation in this class is identified, and its efficiency is partially analyzed. The conclusion is that no representation in this class could be asymptotically optimal.

Chapter 5: I develop a number of general tools for abstract optimization based on efficient data structures for ordered sets and parenthesis balancing. The data structure for ordered sets support the operations `rank` and `select`. Three applications of these tools are presented:

1. Random-access Huffman coding: how to prepare an index that lets us find the mth symbol in a file of n Huffman-coded symbols efficiently. The extra space required for the index is $o(n)$.

2. **Trees in asymptotically optimal space: this addresses the same problem as chapter 4, but obtains an optimal constant factor.**
3. **Planar graphs in linear space: how to store a planar graph of n nodes using only $O(n)$ bits. The operations of adjacency testing and searching (neighbor enumeration) are efficiently supported.**

Chapter 2

Concrete Optimized Trees

This chapter explores the possibility of saving space in pointer-based implementations of static trees. Every implementation of a class of static trees has an associated optimization problem: given a static tree t, find the shortest sequence of words in memory that represents t. For some implementations of trees, the optimization problem is easy. For others it is provably intractable. For yet others the problem is of intermediate difficulty: an efficient optimization algorithm exists, but it is not obvious. In this chapter such problems are examined and classified.

The trees discussed in this chapter are very simple. They are unlabeled, with no extra information stored in the nodes. The only operations we implement are:

- an operation to move from a node to its children (and to iterate through the children in the case of general trees)
- an operation to test if a node is `nil`, a special value indicating that we are no longer in the tree.

Binary trees are considered first. A simple minimization algorithm for binary trees stored as cons-cells is presented, followed by an average case analysis of the space saved thereby. General trees, both ordered and unordered, are then considered. The major result is an algorithm to find a minimal representation for unordered trees where pointers to children are stored in consecutive memory locations. Finally, a practical application of concrete optimization is given: a compact representation of a lexicon of English words, structured as a tree.

2.1 A taxonomy of trees

Trees are important in combinatorics, where they are graphs of a simple type, and in computer science, where they are the most natural structures for representing hierarchical data. They come in many combinatorial varieties, and each variety has several common implementations.

The most basic definition of a tree is an acyclic, connected graph. We call this a *free* tree. If we choose one of the nodes to be the *root*, then we call the tree *oriented* or equivalently *rooted*. The term oriented is used because the choice of root induces a natural orientation on the edges of the tree. Sometime we consider all the edges to be oriented toward the root, but more often we consider them to be oriented away from the root. This kind of tree models hierarchical data. We will deal only with oriented trees in this chapter, because the common representations of free trees model them as oriented trees with a root chosen arbitrarily.

Another dichotomy in the family of trees arises from the labeling of the nodes in the tree. The nodes in the tree may be *unlabeled*; in this case they are intrinsically fungible. Or there may be some information stored at each node. Perhaps this information is just sufficient to distinguish or label every node. The optimizations described in this chapter become less useful as the nodes become more fundamentally distinct (although they will still work). We will ignore the information stored in the nodes of the tree, and treat all trees as unlabeled.

Within the class of unlabeled oriented trees there is another important distinguishing feature: the relationship of the children to each other. If the children form an unordered set, then we say that the tree is *unordered*. If they form a sequence, we say that the tree is *ordered*. Another possibility is that the children are labeled with distinct elements of a k-element set. These are the k-ary trees. The most important subcase is $k = 2$, the *binary* trees.

To summarize, this chapter will deal with static, oriented, unlabeled trees in three principle varieties:

1. binary trees
2. ordered trees of general degree
3. unordered trees of general degree

For each variety, a number of common implementations will be discussed.

2.2 Binary trees as cons-cells

Let's begin with binary trees in their most common implementation, as cons-cells. For each node in the tree, we reserve a record in memory big enough to hold two pointer fields. For historical reasons, we will call these records cons-cells, with the two pointer fields called `car` and `cdr`. The `cdr` holds a pointer to the cell reserved for the left son (or a special value `nil` if there is no left son), and the `car` holds a pointer to the right son. This is all very simple, and navigating among the cons-cells is just a matter of dereferencing pointers.

The usual way to lay out a binary tree T as a collection of cons-cells is to reserve a cell for each node of the tree. But if two subtrees within T are isomorphic, we can save space by using the same collection of cells for both subtrees, since we are not allowed to perform such operations as testing two nodes for equality. If we share space wherever possible, we will find a minimal cons-cell representation of a binary tree. This leads to an obvious algorithm:

1. Find all isomorphic subtrees within T.
2. Allocate a cons-cell for each isomorphism class.
3. Assign values to the pointers.

Steps 2 and 3 are straightforward. Step 1 is easy to do inefficiently by linear search, comparing each subtree with each other subtree. We can make use of a simple dictionary and a postorder tree-walk to make this step fast. We will assign each node t an integer label, giving the isomorphism class of the subtree rooted at t. Our dictionary D is a two-dimensional array, initially all zeroes.

```
label (t: node): integer
        if t = nil
          return 0
        i ← label(left child of t)
        j ← label(right child of t)
        v ← D_ij
        if v = 0
          D_ij ← x
          x ← x + 1
        return D_ij
x ← 1
label(root)
```

Algorithm LABEL: find and label isomorphic subtrees

The integer variable x holds the number of the next label to be assigned. Nil nodes get label 0. Using a real two-dimensional array for D would require $O(n^2)$ space, although the execution time is only $O(n)$. Since D sparse, we can implement it as a balanced tree (with a $\log n$ slowdown in speed) or as a hash table (and accept expected linear time rather than worst-case linear time) using only $O(n)$ space.

2.3 An average case analysis

How much space can we save by using the algorithm presented above? One answer is simply "as much as possible," since we are find a minimum representation. In those trees where there is a great deal of shared structure, the savings is large; but in those with little or no sharing, there is no savings. To get an idea of how much savings we will realize a priori, we need to choose an expected distribution of trees. Then we can compute the on-average savings.

This section presents an average-case analysis of the compression factor achieved by merging isomorphic subtrees after choosing an n-node binary tree from a uniform distribution. We show that this expected compression factor grows without bound as the number of nodes increases; in other words, the expected number of cells needed to store an n-node binary tree in this fashion is $o(n)$.

How many cons cells are needed, on the average, to store a tree of n nodes? The answer, as we have previously observed, depends on the amount of shared structure within the particular tree. We have the following obvious lemma:

Lemma 2.1 *The minimum number of* cons *cells needed to store a binary tree T is equal to the number of non-isomorphic subtrees of T.*

This follows directly from step 2 of the minimization algorithm.

Let $\mathcal{T}_n$ be the set of all n-node binary trees. The direct way to obtain the average number of cells S_n used to store a member of $\mathcal{T}_n$ is to evaluate the following sum and quotient:

$$S_n = \frac{1}{||\mathcal{T}_n||} \sum_{t \in \mathcal{T}_n} ||\{s \mid t \text{ contains } s \text{ as a subtree}\}|| \tag{2.1}$$

Noting that a given subtree s can occur at most once per tree, we use the technique of inverting the summation:

$$S_n = \frac{1}{||\mathcal{T}_n||} \sum_{m=1}^{n} \sum_{s \in \mathcal{T}_m} ||\{t \in \mathcal{T}_n \mid t \text{ contains } s \text{ as a subtree}\}|| \tag{2.2}$$

The motivation for this inversion will become clear shortly. Summations over trees are hard to deal with, so let us try to replace the inner sum of equation 2.2 with a more tractable one. Another lemma will prove useful:

Lemma 2.2 *If s_1 and s_2 are both m-node binary trees, then the number of n-node trees containing s_1 as a subtree is the same as the number of n-node trees containing s_2 as a subtree.*

Proof. Consider the mapping $\mathbf{M}_{s_1 \leftrightarrow s_2} : \mathcal{T}_n \mapsto \mathcal{T}_n$ that takes an n-node tree t and replaces all subtrees of t isomorphic to s_1 by copies of s_2, and all subtrees of t isomorphic to s_2 by copies of s_1. Because s_1 and s_2 both have m nodes, neither can be a proper subset of the other, and so $\mathbf{M}_{s_1 \leftrightarrow s_2}$ must be a bijection. Since $\mathbf{M}_{s_1 \leftrightarrow s_2}$ maps trees in $\mathcal{T}_n$ containing s_1 to trees containing s_2 and vice versa, the trees containing copies of s_1 must be equinumerous with those containing s_2.

This lemma shows that the number of n-node trees containing a particular m-node tree depends only on m, and not on the shape of the particular m-node tree. We can count the total for all m-node trees by computing the number for any particular m-node tree, and multiplying by the total number of m-node trees.

The number of n-node binary trees is C_n, the nth Catalan number:

$$C_n = \binom{2n}{n} \frac{1}{n+1} \tag{2.3}$$

Using lemma 2.2, we can write equation 2.2 like this:

$$S_n = \frac{1}{C_n} \sum_{1 \le m \le n} C_m \cdot ||\{t \in \mathcal{T}_n \mid t \text{ contains a particular } m\text{-node subtree}\}|| \tag{2.4}$$

We must find the size of the set in this sum. Let us give it a name:

$$A_{nm} = ||\{t \in \mathcal{T}_n \mid t \text{ contains a particular } m\text{-node subtree}\}|| \tag{2.5}$$

2.3.1 A generating function for A_{nm}

It is easy to get a generating function for the first index of A_{nm}. Let us start by counting the number of trees B_{nm} that do *not* contain a particular m-node subtree. Clearly $B_{nm} = C_n - A_{nm}$. Define $b_m(x)$ to be the generating function of B_{nm} over n:

$$b_m(x) = \sum_n B_{nm} \cdot x^n \tag{2.6}$$

Observe that the values of B_{nm} obey the Catalan recurrence, for most values of n:

$$B_{nm} = \sum_{i=0}^{n-1} B_{im} \cdot B_{n-1-i,m}, \quad \text{for } n \neq m \tag{2.7}$$

but when $n = m$, exactly one tree is excluded:

$$B_{mm} = \sum_{i=0}^{m-1} B_{im} \cdot B_{m-1-i,m} - 1 \tag{2.8}$$

From this pair of equations, we see that $b_m(x)$ satisfies:

$$xb^2{}_m(x) = b_m(x) + x^m - 1 \tag{2.9}$$

Using the quadratic formula (and observing that $b_m(0) = 1$ for $m > 0$ to discard the spurious root) we obtain:

$$b_m(x) = \frac{1}{2x}(1 - \sqrt{1 - 4x + 4x^{m+1}}) \tag{2.10}$$

A combinatorial solution

We could simply expand equation 2.10 directly and subtract from the Catalan numbers to get an expression for A_{nm}. An equivalent, but more direct method to obtain such an expression is to use the principle of inclusion-exclusion.

Crudely, we may estimate the number of n-node trees containing a particular m-node tree by observing that such trees consist of any $(n - m)$-node tree, with our desired m-node tree spliced in at one of its external nodes, as shown in figure 2.1. Since the $(n - m)$-node tree has $n - m + 1$

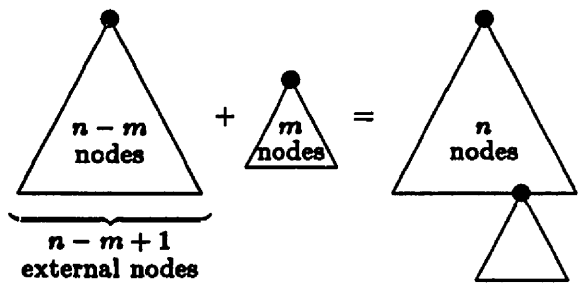


Figure 2.1: Splicing trees

external nodes, this shows that

$$A_{nm} \approx (n - m + 1)C_{n-m} \tag{2.11}$$

This approximation over-counts those n-node trees in which our desired m-node subtree occurs more than once. In any case, note that that $(n-m+1)C_{n-m}$ is an upper bound on A_{nm}; we will make use of this later.

To refine our approximation, we can subtract off the number of trees in which the desired m-node subtree occurs at least twice. We can form such trees by starting with an arbitrary $(n-2m)$-node tree, and splicing in two copies of the m-node subtree at two of the $n-2m+1$ external nodes. Subtracting, we get:

$$A_{nm} \approx (n-m+1)C_{n-m} - \binom{n-2m+1}{2} C_{n-2m} \tag{2.12}$$

This new approximation now *under-counts* those trees with three or more copies of the m-node subtree. Therefore, this quantity must be a lower bound on A_{nm}.

We can continue in this way, alternately adding and subtracting terms, to get an exact form for A_{nm}. Let R_j be the set of all n-node trees t such that the jth node of t (numbering the nodes in any predetermined order) is the root of the particular m-node subtree we seek. Let I_j denote the sum of the sizes of all j-tuple intersections of the R_j's. Then by inclusion-exclusion,

$$A_{nm} = \sum_{j=1}^{\lfloor n/m \rfloor} (-1)^{j+1} I_j \tag{2.13}$$

The upper bound of this sum is simply the maximum number of copies of the m-node subtree that could possibly fit into an n-node tree. The argument in the preceding paragraph demonstrates that

$$I_j = \binom{n-jm+1}{j} C_{n-jm} \tag{2.14}$$

and so

$$A_{nm} = \sum_{j=1}^{\lfloor n/m \rfloor} (-1)^{j+1} \binom{n-jm+1}{j} C_{n-jm} \tag{2.15}$$

We can now combine this sum with equation 2.4 to get an expression for the average space needed to store a tree of n-nodes:

$$S_n = \frac{1}{C_n} \sum_{m=1}^{n} \sum_{j=1}^{\lfloor n/m \rfloor} (-1)^{j+1} \binom{n-jm+1}{j} C_{n-jm} C_m \tag{2.16}$$

2.3.2 Asymptotic space requirements

Getting an exact closed form for the double sum in equation 2.16 seems to be hard. It is almost as enlightening (and more feasible) to study the asymptotic behavior of this sum as n grows large. To get a feel for the behavior of S_n, a graph for $1 \leq n \leq 1000$ is included in figure 2.2. (It is hard to calculate exact values for large n, since very large binomial coefficients are involved.) What can we conclude from this graph? Not too much. In the region shown, it looks a bit like a line with slope 1/2, but it would be hasty to make an asymptotic judgement. We will make good use of a

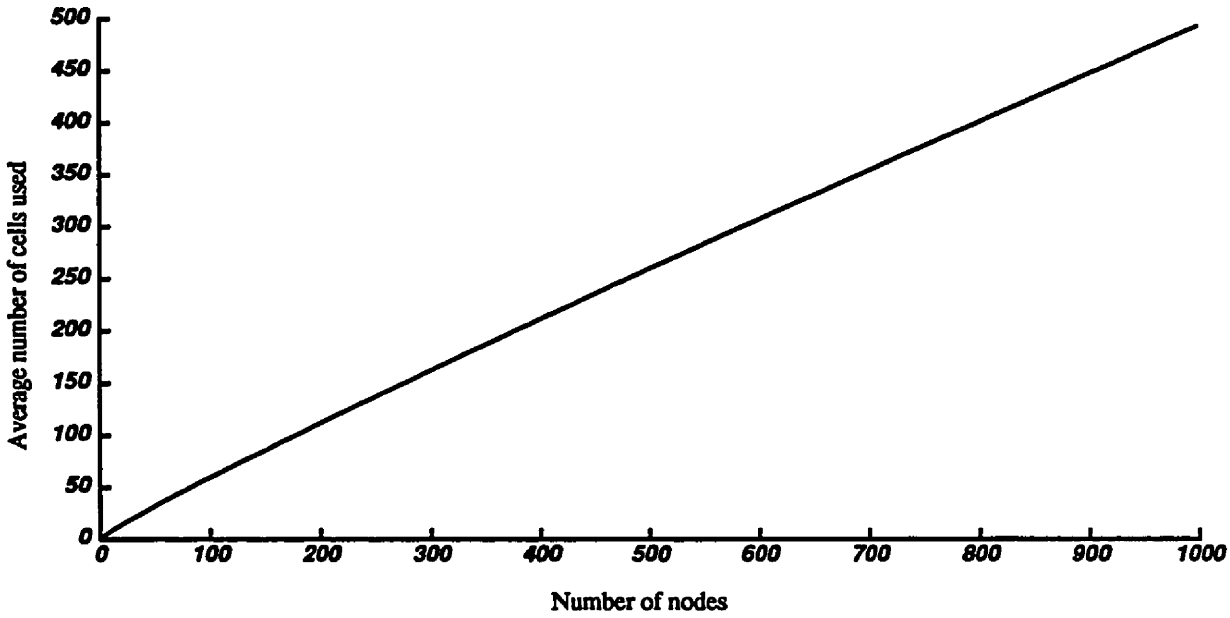


Figure 2.2: Graph of S_n for $1 \leq n \leq 1000$

graphical insight a little later on.

To gain a better understanding of S_n, let's break down the outer sum of equation 2.16. Say that the *size* of a node n in a tree is number of nodes in the subtree rooted at n. The outer sum of equation 2.16 totals up the number of cells used by nodes of various sizes from 1 (the leaves) to n (the root). Let us rearrange that equation slightly:

$$S_n = \sum_{m=1}^{n} \frac{C_m}{C_n} \cdot A_{nm} \tag{2.17}$$

Term the summed quantity $K_{nm} = (C_m/C_n) \cdot A_{nm}$. This quantity counts the expected number

of cells used to store those nodes of size m. Since many nodes have small sizes, K_{nm} will be relatively large when m is small. But since there are a fixed number of different trees with m nodes, and each can cost us only one cell no matter how many times it occurs, K_{nm} must ultimately decrease again as m get very small.

Consider the following two inequalities on A_{nm}:

$$\begin{aligned} A_{nm} &\leq C_n \\ A_{nm} &\leq (n-m+1)C_{n-m} \end{aligned} \tag{2.18}$$

The first inequality expresses the fact that the number of n-node trees containing a particular m-node subtree tree cannot exceed the total number of n-node trees. The second one we recognize from equation 2.12. Let us now proceed to identify the range of values for m where each inequality is dominant, and *dissect* the sum of equation 2.17 in the style of Knuth and Green[30, page 48].

The formula for the Catalan numbers given in equation 2.3, together with Stirling's approximation, yields the following:

$$C_n = \frac{1}{\sqrt{\pi}} 4^n n^{-3/2} + O(4^n n^{-5/2}) \tag{2.19}$$

The "crossover point" of A_{nm} where the second inequality begins to dominate the first in the minimum occurs where $C_n = (n-m+1)C_{n-m}$. By the approximation in equation 2.19 this comes about when

$$\begin{aligned} \frac{1}{\sqrt{\pi}} 4^n n^{-3/2} &= (n-m+1)\frac{1}{\sqrt{\pi}} 4^{n-m}(n-m)^{-3/2} \\ 4^m &= (n-m+1)(n-m)^{-3/2} n^{3/2} \\ m &= \frac{1}{2}\lg n + \frac{1}{2}\lg\left[1+\frac{1}{n-m}\right] + \frac{1}{4}\lg\left[1+\frac{m}{n-m}\right] \end{aligned} \tag{2.20}$$

z This shows that when n is large, the crossover comes just past $m = \frac{1}{2}\lg n$. For concreteness, call $p = \left\lceil \frac{1}{2}\lg n \right\rceil$ this crossover point. Let us give names to the regions on both sides of p. Call the domain of $m < p$ the *saturated* regime. For values of m in the saturated regime, almost all of the C_m different m-node trees are likely to occur as a subtree of an average n-node tree. The domain of $m \geq p$ is the *unique* regime. In this regime, each m-node subtree of an average n-node tree is unlikely to be found elsewhere in the tree; the sharing of structure here is small.

The two inequalities on A_{nm} lead directly to the following inequalities on K_{nm}, obtained by multiplying through by C_m/C_n:

$$K_{nm} \leq C_m \tag{2.21}$$

$$K_{nm} \leq (n-m+1)\frac{C_{n-m}C_m}{C_n}$$

So we can bound K_{nm} by a function K'_{nm} defined as follows:

$$K'_{nm} = \begin{cases} C_m & m < p \\ (n-m+1)C_{n-m} & m \geq p \end{cases} \tag{2.22}$$

We will soon show that K'_{nm} is not too much bigger than K_{nm} for most values of m. The quality of the approximation of K_{nm} by K'_{nm} can be seen in figures 2.3 and 2.4, showing the breakdown of space used in trees of 1000 nodes by node size. Note the very sharp transition in both K'_{nm} and K_{nm} around $m = p$. In figure 2.4, the difference between the two values is unobservably small. The value of S_n is equal to the sum of the K_{nm} over m; graphically, this means that S_n is the area under the curves shown.

The saturated regime

Let us first get a bound on the total space required in the saturated regime.

Using K'_{nm} as our upper bound, we get

$$\begin{aligned} \sum_{m=1}^{p-1} C_m &\leq \sum_{m=1}^{p-1} C_m C_{p-1-m} \\ &\leq C_p \end{aligned}$$

Remembering that $p = \left\lceil \frac{1}{2} \lg n \right\rceil$ and employing the approximation for C_n from equation 2.19, we can see that the total average space used by nodes in the saturated regime is $O(n \log^{-3/2} n)$.

The unique regime

The graphs of figures 2.4 and 2.3 suggest that the bulk of the space used is in the lower part of the unique regime. We will determine the space used by the following steps:

1. Approximate K_{nm} by K'_{nm}.
2. Find a closed form for the sum of K'_{nm} in this regime.
3. Approximate the sum using Stirling's approximation.
4. Bound the difference between K_{nm} and K'_{nm}.

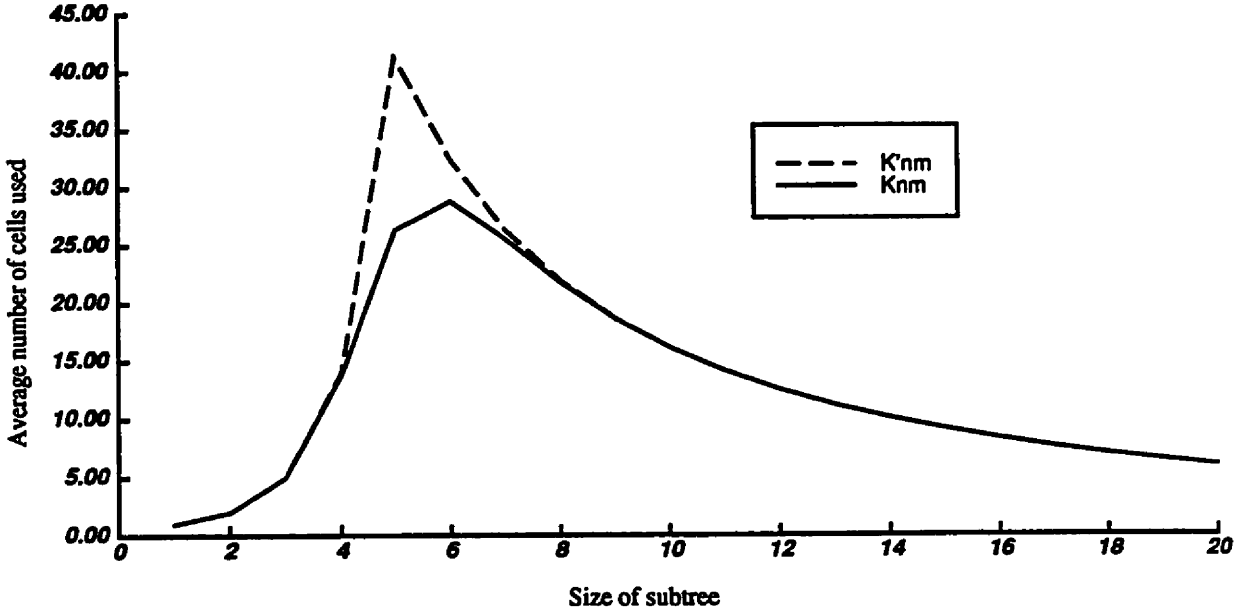


Figure 2.3: Graph of K_{nm} and K'_{nm} for $n = 1000, m \leq 20$

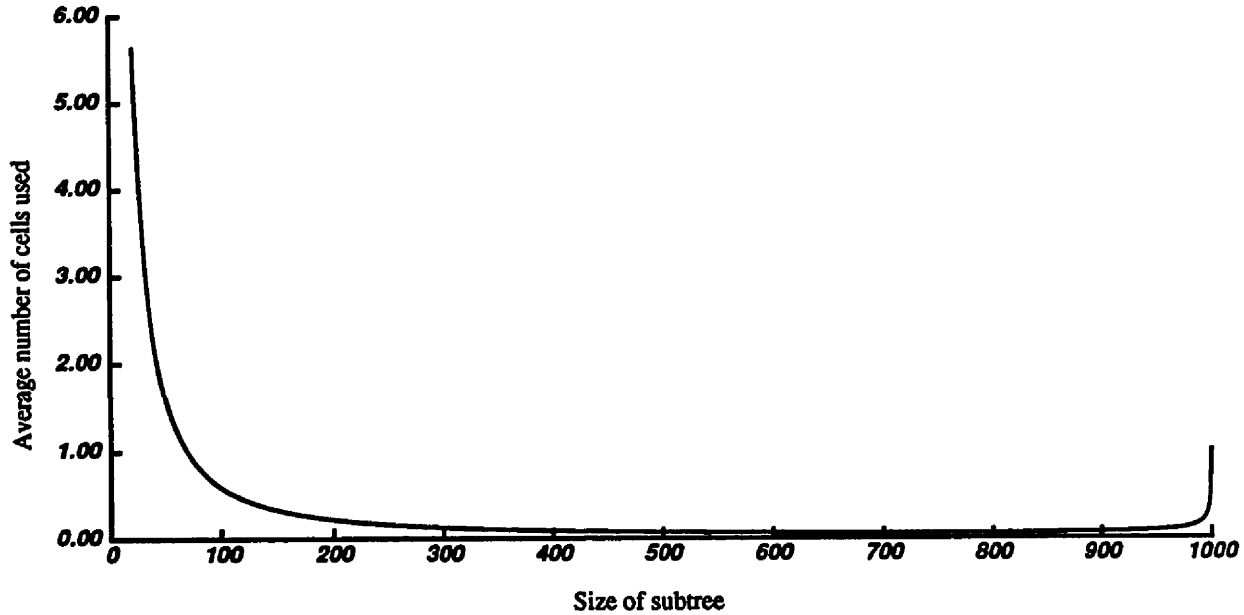


Figure 2.4: Graph of K_{nm} and K'_{nm} for $n = 1000, m > 20$

The first step is to replace the sum $\sum_{m=p}^{n} K_{nm}$ by $\sum_{m=p}^{n} K'_{nm}$. Looking at the graphs, we would guess that this isn't too bad an approximation; later we shall make this rigorous.

While K_{nm} is difficult to sum, K'_{nm} is simple. We have:

$$\begin{aligned}
\sum_{m=p}^{n} K'_{nm} &= \frac{1}{C_n} \sum_{m=p}^{n} (n-m+1) C_{n-m} C_m \\
&= (n+1) \binom{2n}{n}^{-1} \sum_{m=p}^{n} \frac{1}{m+1} \binom{2(n-m)}{n-m} \binom{2m}{m} \\
&= \binom{2n}{n}^{-1} [2(n-p)+1] \binom{2p}{p} \binom{2(n-p)}{n-p} \\
&= [2(n-p)+1] \binom{n}{p}^2 \binom{2n}{2p}^{-1} \qquad (2.23)
\end{aligned}$$

by a variety of combinatorial identities. Notice that this formula gives the expected number of cells needed to store those nodes of size p or greater in a tree of n nodes when we do *not* merge isomorphic subtrees. We can check this by plugging $p = 1$ into equation 2.23:

$$[2(n-1)+1] \binom{n}{1}^2 \binom{2n}{2}^{-1} = n$$

As expected, we learn that n cells are required to store trees of n nodes in the normal way.

Now, applying Stirling's approximation in the range where $p \ll n$ to equation 2.23

$$[2(n-p)+1] \binom{n}{p}^2 \binom{2n}{2p}^{-1} = \frac{2}{\sqrt{\pi}} n p^{-1/2} + O(n p^{-3/2}) \qquad (2.24)$$

We can now plug in $p = \left\lceil \frac{1}{2} \lg n \right\rceil$, to get

$$\sum_{m=p}^{n} K'_{nm} = \sqrt{\frac{8}{\pi}} n \lg^{-1/2} n + O(n \log^{-3/2} n) \qquad (2.25)$$

This confirms our suspicion that the bulk of the space is used here.

So far we have shown that $S_n = O(n \log^{-1/2} n)$. To find a good lower bound for S_n, it remains to bound $K'_{nm} - K_{nm}$ in the unique regime.

As mentioned earlier, the approximation in equation 2.12 is also an inequality:

$$A_{nm} \geq (n-m+1) C_{n-m} - \binom{n-2m+1}{2} C_{n-2m} \qquad (2.26)$$

Multiplying by C_m/C_n, and admitting $m \geq p$, we get

$$K_{nm} \geq K'_{nm} - \binom{n-2m+1}{2} C_{n-2m} C_m / C_n \tag{2.27}$$

So $\binom{n-2m+1}{2} C_{n-2m} C_m / C_n$ is a bound on $K'_{nm} - K_{nm}$. We would like to show that the sum of this quantity in the unique regime is not too large. We will call this quantity D_{nm}. First, let us observe that D_{nm} is a decreasing function of m. To see how quickly it gets small, consider the term ratio $D_{nm}/D_{n,m+1}$:

$$\begin{aligned} \frac{D_{nm}}{D_{n,m+1}} &= \frac{\binom{n-2m+1}{2} C_{n-2m} C_m}{\binom{n-2m-1}{2} C_{n-2m-2} C_{m+1}} \\ &= \underbrace{\frac{2m+4}{2m+1}}_{>1} \cdot \underbrace{\frac{2(n-2m)-1}{n-2m-1}}_{>2} \cdot \underbrace{\frac{2(n-2m)-3}{n-2m-2}}_{>2} \\ &> 4 \end{aligned} \tag{2.28}$$

The values of D_{nm} get smaller from term to term by a factor of at least 4. This makes it easy to get a bound on the sum:

$$\begin{aligned} \sum_{m=p}^{n} D_{nm} &< \sum_{m=p}^{n} D_{np} 4^{p-m} \\ &< D_{np} \sum_{m=p}^{n} 4^{p-m} \\ &< \frac{4}{3} D_{np} \end{aligned}$$

The whole sum is simply a constant times the value at $m = p$. We once again enlist the aid of Stirling's approximation, setting $p = \left\lceil \frac{1}{2} \lg n \right\rceil$

$$\begin{aligned} D_{np} &= \binom{n-2p+1}{2} C_{n-2p} C_p / C_n \\ &= O(n \log^{-3/2} n) \end{aligned}$$

This demonstrates that our overestimate of K_{nm} by K'_{nm} does not contribute to the asymptotically dominating term in the sum, which is $O(n \log^{-1/2} n)$. We previously showed that the entirety of the space used by the saturated regime is $O(n \log^{-3/2} n)$, so it doesn't contribute either. Putting all the pieces together, we can now conclude that:

$$S_n = \sqrt{\frac{8}{\pi}} n \lg^{-1/2} n + O(n \log^{-3/2} n) \tag{2.29}$$

2.3.3 Conclusions

The main result of this section is that the compression factor realized by merging isomorphic subtrees grows without bound, when trees are chosen at random under a uniform distribution. This factor grows rather slowly, only $O(\sqrt{\log n})$. For a random binary tree with a million nodes, the expected compression factor is a little under three.

An information-theoretic bound

We can compare this growth rate in the compression ratio to an information-theoretic bound. Since each of the C_n trees are equally likely, we will need an average of at least $\lg C_n$ *bits* to encode each one. Applying the approximation from equation 2.19, this means that *any* representation must average around $2n$ bits per tree. The pointers we use must have enough bits to address all the different nodes in the tree; therefore each pointer is about $\lg n$ bits long. So we must have at least $\Omega(n/\log n)$ pointers (and as many cells, up to a constant factor), to produce the required minimum number of bits. This gives an upper bound of $O(\log n)$ on the compression factor.

A distribution favoring balanced trees

The analysis was done assuming a uniform distribution of n-node binary trees. While this distribution is a reasonable choice, it does not lead to very balanced trees. The "average" binary tree of n nodes is a lean and scraggly fellow whose internal path length is $O(\sqrt{n})$ (see Knuth[29, section 6.2.2]). One of the reasons that trees are important in computer science is that (when they are reasonably balanced) they provide access to elements starting from the root in logarithmic time. This means that we might want to redo this analysis using a distribution of trees that favors balance. We can expect the savings to be greater when trees are balanced because there are more very small subtrees. The perfectly balanced tree of $2^m - 1$ nodes occupies only m cells, for example.

One particular non-uniform distribution of trees comes to mind as a candidate for analysis. Imagine that our n-node trees are binary search trees formed by inserting n keys in random order into an initially empty tree. Ignoring the values of the keys, and looking only at the shape of the resulting trees, we get a realistic distribution of n-node binary trees that is much more balanced than the uniform distribution. The average internal path length of such trees is only $O(\log n)$ (see Knuth[29, section 6.2.2, page 427]). To analyze this distribution, we would start by considering the $n!$ different possible permutations of the keys as equally likely. We would sum the space used in

each permutation, and divide the total by $n!$ to get the average.

One difficulty of modifying the analysis presented in this section to deal with this distribution is that lemma 2.2, which says that the number of n-node trees containing a particular m-node tree is independent of the shape of the m-node tree, doesn't help here. The *number* of n-node trees may be independent of which m-node tree we are looking for, but the total *probability* under this distribution is not. Balanced m-node trees will more likely be found as subtrees than unbalanced ones. Still, it should be possible to analyze the average space used under this distribution, and this analysis is an open problem. It seems inevitable that less space would be needed, on average, for trees under this distribution. In fact, the information-theoretic bound of $O(\log n)$ for the compression factor doesn't even apply here, so the compression could conceivably be superlogarithmic.

Who is using the space?

Another useful by-product of the way the asymptotic analysis was done is that it gives us a good understanding of where in the tree the space is going. The sharp peak in the distribution of space around size $m = 1/2 \lg n$ means that the almost all savings in space (that is, merging of isomorphic subtrees) occurs in trees of that size or smaller. We can imagine that there is no savings of space for nodes whose size lies in the unique regime, and nearly total saving in the saturated regime. This style of accounting for the space by node size may also be useful in the analysis proposed in the previous paragraph.

2.4 Binary trees as cdr-coded lists

When trees are static, there is an easy way to reduce the space by almost a factor of two. If we arrange the tree so that the right son of a node t is stored in the memory location immediately following t, then we no longer need cdr pointers. The cdr is right there in the next memory cell. This technique (which is similar enough to the cdr-coding of ZETALISP[48] that we will appropriate that name for it) needs only one pointer per node (pointing to the left son), plus something more to tell us when a node has a node has no right son. We can either reserve one extra bit from each memory cell for this purpose, or reserve one extra cell containing a special value following each node with no left son.

We cannot apply the algorithm from the previous section to trees that are cdr-coded. Notice that the same node cannot be the cdr of two non-isomorphic nodes, as it can in the cons-cell

representation.

It is still quite easy to optimize space, though. Let's call a group of nodes stored in consecutive locations, where each one's cdr is in the next cell, a *block*. A block's end is flagged either by a bit, or by a cell containing a special value. The space used by a particular representation of a tree is the total size of all the blocks.

Two blocks can *nest*, and share the same space, when one forms a terminal sequence of another. Let's call those nodes that are left children of their parents *blockheads*, because these nodes would appear at the beginnings of blocks if there were no nesting. If a subtree rooted at node t is isomorphic to the subtree rooted at another node s, then the block containing t may be nested in the block containing s *if and only if* node t is a blockhead. This observation makes it clear how to modify the algorithm of the previous section:

1. Group the nodes of T into blocks.
2. Find all isomorphic subtrees within T.
3. Allocate k cells for each block of size k whose blockhead is not isomorphic to any other node in T.
4. Assign values to the pointers.

All these steps are straightforward. In step 3, if we encounter a group of blocks that are isomorphic, each of k nodes, we allocate exactly one block of size k for them.

2.5 General trees as binary trees

In general trees, each node may have an arbitrary number of children. There is a well-known way to represent an ordered general tree as a binary tree: we build a binary tree where "left son" and "right son" mean "first child" and "next sibling" respectively in the original tree. After translating an ordered general tree into a binary tree in this way, we can then use either the cons-cell representation or the cdr-coded representation, described earlier in this chapter, to complete a concrete implementation. The optimization problems for general trees stored in this fashion are the same as before, so the same algorithms apply.

It is worth noting that the cdr-coded representation for binary trees, when applied to general trees, can be understood as follows: Each node t of the general tree is represented by a block of d

cells (where d is the branching degree of t). The d cells store pointers to the blocks that represent the d children of node t. This explanation is more easily understood than saying that we first translate the general tree into a binary tree, and then represent the binary tree as a cdr-coded list.

But what about *unordered* general trees, where we do not require that the children of a given node occur in a particular order? Let us restrict our attention to methods that store an unordered tree by first fixing an order for the children at each node, and subsequently use one of the methods already described. We could simply try each possible order for the children in all combinations to find the one with the smallest space requirement, but that would be quite expensive. Can we do better?

2.5.1 Unordered trees as cons-cells

If we try to represent a tree in a minimum number of cons-cells, we run into intractability; this minimization problem is NP-Hard. I now show how to reduce VERTEX COVER(see Garey and Johnson[20, pages 53–56]), to this problem. Given a graph $G = (V, E)$ we construct a unordered tree T of uniform depth 3 where T can be stored in $k + \|V\| + 2\|E\| + 1$ cons-cells if and only if G has a vertex cover of size k.

There is exactly one node in T at depth 0, the root r. Let each edge $e \in E$ have an associated node at depth 1 in T called t_e. Each t_e itself has exactly two children (at depth 2). Let the nodes in V be numbered from $1 \ldots \|V\|$. If edge $e = \langle i, j \rangle$, then let the two children of node t_e have i and j children respectively (leaves at depth 3).

How many cells are needed to store T? We need one cell for the root. The leaves, though numerous, require exactly $\|V\|$ cells chained together as a list with `nil car` fields. This linked list is shared by all the leaves. We also need exactly one cell for every node at depth 1 of T. This accounts for another $\|E\|$ cells, for a total of $\|V\| + \|E\| + 1$ so far for the nodes at depths 0, 1 and 3. All that remains to be determined is the number of cells required by nodes at depth 2 in T.

Now t_e has exactly two children at depth 2 in T. When represented by cons-cells, one of these children's cells (call it the *head*) will point, via its `cdr` field, to its `next-sibling` in the other child's cell (call it the *tail*). Since all the edges in G are all distinct, the head cells are never shared. But tail cells all have `nil cdr` fields and may be shared. The number of head cells at depth 2 is simply $\|E\|$. But the number of tail cells depends on how we chose which of the two children of the t_e would become the head and which the tail. Each such choice effectively orients the edge e. The VERTEX COVER problem can be rephrased as follows: given an undirected graph G and a positive

integer k, can the edges of G be oriented so that there are at most k vertices in G with nonzero in-degree? If (and only if) G has a vertex cover of size k, we will use only k tail cells. The total number of cells needed by nodes at depth 2 is $k + \|E\|$.

This completes the demonstration that T can be represented in $k + \|V\| + 2\|E\| + 1$ cells if and only if G has a vertex cover of size k.

2.6 Unordered trees as cdr-coded lists

If we store our tree as a cdr-coded list, the optimization problem becomes tractable, but the algorithm is not obvious. I shall go into this problem in some detail, presenting an algorithm to find a minimal representation. Let me start with a complete concrete specification.

Each node t of the tree has an associated block of cells, which stores pointers to the children of t in consecutive locations. All the blocks are concatenated into one big array of memory cells, and the pointers are just integer indices that indicate where in the array the children's blocks start. We will use the special value 0 as a block terminator to mark the end of the children. (We could just as well use a bit flag in each cell for this purpose; the extra 0 cell will make explanation easier.)

Given this representation, it is straightforward to translate a tree into an array of integers. (See figure 2.5 for an example) The total number of cells needed to store an n-node tree is $2n - 1$. Each node has a cell containing a 0 marking the end of its children (accounting for n array locations), and each of the nodes (except the root) is referred to by exactly one cell, accounting for the $n - 1$ locations filled with non-zeroes.

2.6.1 Minimizing space

The straightforward method of creating an array from a tree is wasteful in its allocation of space. The optimization algorithm presented in section 2.4 cannot be applied, but only because the children of a node do not have a fixed order, so two blocks may or may not be nestable, depending on what ordering of children we choose.

The order in which the indices of the children of a tree are stored within a block is up to us. By choosing a favorable ordering of the children, we make it possible for certain blocks to be nested in other blocks. When each child of t is isomorphic to a child of s, t can be nested in s if we order the children of s so that those that are isomorphic to children of t are stored after the others. But arranging the children of s so that t can be nested therein may make it impossible to nest some other node u in s.

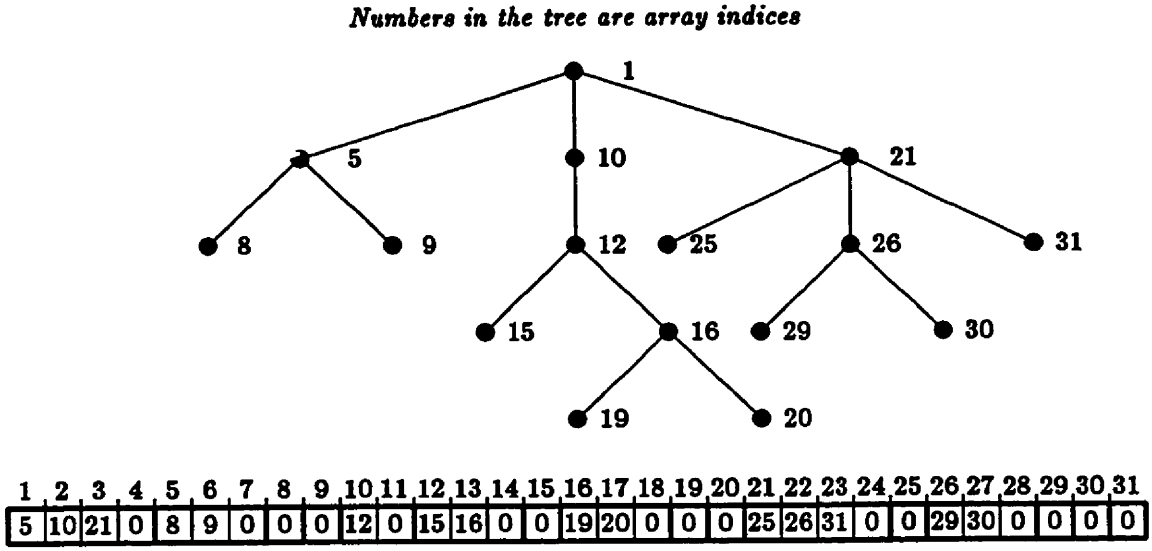


Figure 2.5: A straightforward translation of a tree into an array.

Note that the nodes $t_1, t_2, \ldots, t_k$ can be recursively nested, as long as each of the children of t_i is isomorphic to a child of t_{i+1}, for $1 \leq i < k$. This nesting will produce a savings of $\sum_{i=1}^{k-1}(c_{t_i}+1)$ locations. Figure 2.6 shows the same tree from figure 2.5 represented as an array of minimum size. How did we decide how to order the children at each node to maximize savings through nesting? I will soon show how weighted matching can be used to find the best ordering.

2.6.2 Some formal notation

If the subtree rooted at t and subtree rooted at s are isomorphic, write $t \cong s$. If for each child t' of t there exists a child s' of node s such that $t' \cong s'$, then t is *nestable* in s, written $t \preceq s$. If $t \preceq s$ but not $t \cong s$, then t is *strictly nestable* in s, written $t \prec s$. If either $s \preceq t$ or $t \preceq s$, informally say that t and s are *nestable*. Finally, a collection of nodes that are all mutually comparable under nestability will be called *mutually nestable*, and a partition of a collection of nodes into mutually nestable subcollections is a *nesting* partition.

Note that nestability is a reflexive and transitive relation, and that strict nestability is asymmetric and transitive.

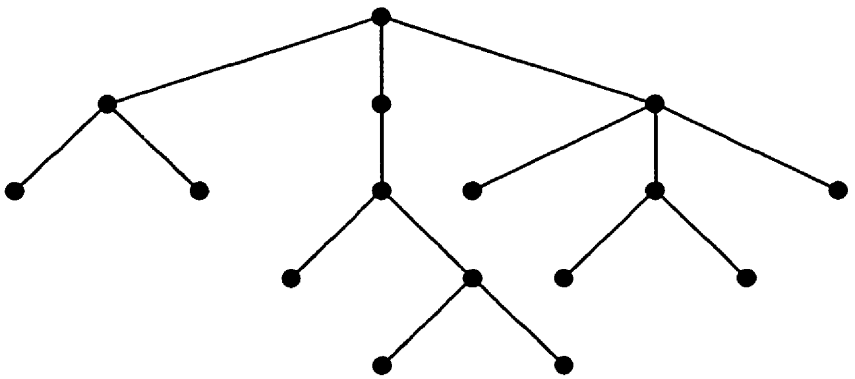
2.6.3 How to nest?

A nesting partition of the nodes in a tree places constraints on the ordering of the children. If we choose some nesting partition π, and then nest those blocks that are in the same part under π, the resulting array will have a fixed size

$$N(\pi) = 2n - 1 - \sum_{\substack{t \\ \text{where } t \text{ is} \\ \text{nested in} \\ \text{something} \\ \text{else}}} (c_t - 1) \tag{2.30}$$

That is, the space for all of the blocks that are nested in other blocks (the sum in this equation) is saved. Maximizing this sum necessarily minimizes the space used.

Original tree:



Original array:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
5	10	21	0	8	9	0	0	0	12	0	15	16	0	0	19	20	0	0	0	25	26	31	0	0	29	30	0	0	0	0

Minimal array:

1	2	3	4	5	6	7	8	9	10	11	12	13
5	8	10	0	4	4	0	11	0	4	4	5	0

Figure 2.6: Reducing the space for tree storage.

2.6.4 A matching method

A nesting partition of the nodes in a tree is just a covering of the nestability relation by *chains*. (A chain is just a sequence of elements $e_1, e_2, \ldots, e_k$ such that $e_1 \preceq e_2 \preceq \cdots \preceq e_k$.) If the *weight* of a chain $n_1 \preceq n_2 \preceq \ldots \preceq n_k$ is defined as $c_{n_k} + 1$, then the nesting partition sought is the covering by chains of minimum total weight. Minimum covering of a transitive relation by chains is a well-known problem. Dilworth[10] showed that it can be solved by bipartite matching. The weighted version of the problem here can similarly be solved by weighted bipartite matching (see Dantzig[9]).

From a tree, construct a weighted bipartite graph $G = (X, Y, E)$ where E is a function from $X \times Y$ to the natural numbers. For each node t, create a pair of vertices $x_t \in X$ and $y_t \in Y$. Let $E(x_s, y_t)$ be $c_s + 1$ if $s \preceq t$ and $s \neq t$, and zero otherwise. A matching in G (using only edges with positive weight) corresponds directly to a covering of $\preceq$ by chains, and vice versa. Matching x_s to y_t corresponds to $\ldots \preceq m \preceq n \preceq \ldots$ appearing in some chain. The weight of edge $E(x_s, y_t)$ in G is the savings realized by nesting s in t.

The total weight of a matching in G is the total savings (over the straightforward method) in space realized by the corresponding nestings in the tree, as was seen in equation 2.30. A maximum matching therefore corresponds to maximum savings, and hence minimum space. Now we can sketch an algorithm for translating a tree into an array of minimum size:

1. Determine which pairs of subtrees are isomorphic.
2. Determine which pairs of nodes are nestable.
3. Form the graph G described above, and use a standard algorithm (the Hungarian Method of Kuhn[33], for example) to find a maximum weighted bipartite matching in G.
4. Now consider the graph $G' = (T, E)$ formed by identifying the vertices x_n and y_n in G. The edges in the matching found in the previous step form a covering of G' by chains. Partition the nodes placing each chain in its own partition.
5. For each partition, order the children of each node in a proper nesting order. Allocate space for the partitions and assign array indices to all nodes.

Each step of the above algorithm can be done in polynomial time (in the size of the tree structure). The one-to-one correspondence between matchings in the graph G and nesting partitions

of the nodes in the tree guarantees that the algorithm will find the minimum size array that can store the nodes. An example of this algorithm is graphically illustrated in figure 2.7, using the same tree as the other figures. The details of the implementation will now be discussed.

2.7 Implementation details

The previous section offered only a sketch of techniques and a demonstration of a polynomial algorithm for the problem at hand. Since the algorithm proposed is supposed to be *practical* for very large problem sizes, more must be said about the nitty-gritty issues of implementation. The steps in the algorithm will now be addressed in turn.

2.7.1 Subtree isomorphism

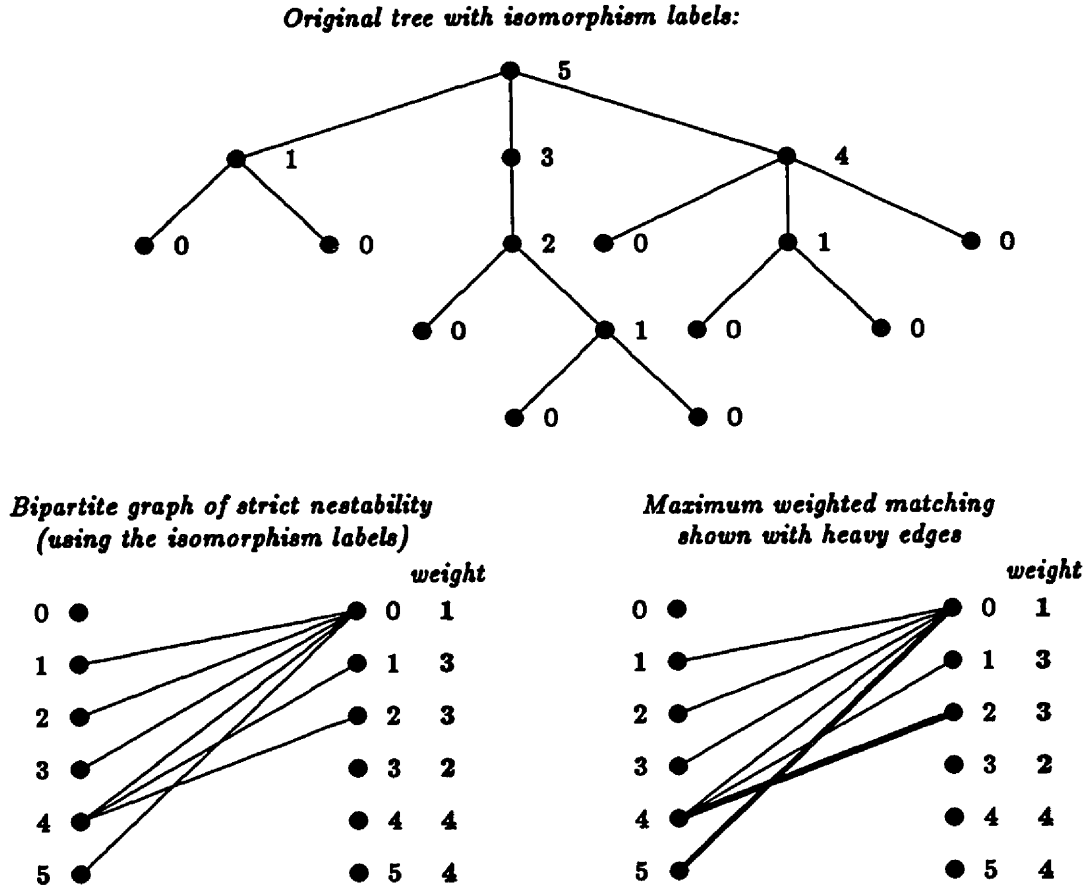
Identifying isomorphic subtrees is slightly harder for general unordered trees than it is for binary trees. Nonetheless, we can still adapt the algorithm LABEL for general trees. Instead of a being indexed by a pair of integers (that is, a two-dimensional array) the dictionary D must now be indexed by a multiset of integers:

```
ulabel (t: node): integer
        if t = nil
          return 0
        S ← ∅
        for each child t' of t
            S ← S + {ulabel(t')}
        v ← D_S
        if v = 0
          D_S ← x
          x ← x + 1
        return D_S

x ← 1
ulabel(root)
```

Algorithm ULABEL: find and label isomorphic subtrees

Although we can no longer use a simple array to store D, we can still implement D efficiently. Aho[1, pages 84–86] describes an algorithm for comparing two unordered trees similar to ULABEL, that uses sorting. If we sort the multiset S, we can build D as a search tree keyed on sorted multisets. This would mean a total time of $O(n \log n)$ for ULABELing a tree of n nodes.



Resulting minimum-size array:

1	2	3	4	5	6	7	8	9	10	11	12	13
5	8	10	0	4	4	0	11	0	4	4	5	0

Figure 2.7: The space-minimization algorithm: an example.

Instead, we could maintain D as a hash table keyed on multisets. Using an associative and commutative hash function to combine the integers in S would remove the need for sorting, and we could perform ULABEL in expected time $O(n)$. This is similar to a method described by Miller[36].

Nodes that are the roots of isomorphic subtrees are mutually nestable. It can never be a mistake (in the later matching phase) to put such isomorphic nodes in the same nesting partition. This is so because whenever $s \cong t$, the partition

$$\pi = \cdots\{\ldots, s, \ldots\}\cdots\{\ldots, t, \ldots\}\cdots$$

can be replaced with the partition

$$\pi' = \cdots\{\ldots, s, t, \ldots\}\cdots$$

requiring the same number or fewer array locations; that is, $N(\pi') \leq N(\pi)$. Therefore it simplifies the rest of the description to transform a tree into a directed acyclic graph G by first identifying isomorphic subtrees, and operating directly on G. If the labels assigned by algorithm ULABEL are $0 \ldots x$, then the corresponding graph $G = (V, E)$ will have vertices $V = \{v_0 \ldots v_x\}$, and edges $E = \{\langle v_y, v_z\rangle \mid y \text{ is the label of a child of a node with label } z\}$. Each vertex of G represents an entire class of isomorphic nodes. A nesting partition of the vertices of G directly implies a nesting partion of the nodes in the original tree.

2.7.2 Computing nestability

This step is likely to be the computational bottleneck of the whole operation. The following simple algorithm takes the directed graph $G = (V, E)$ (constructed by the tree isomorphism step described in the previous section), and produces the relation $R_{\preceq} \subseteq V \times V$, where $u \preceq v$ means that for every vertex w, $\langle u, w\rangle \in E \Rightarrow \langle v, w\rangle \in E$ (the successors of u are a subset of the successors of v).

```
R_≤ ← ∅
foreach u ∈ V
        foreach v ∈ V
                flag ← true
                foreach w ∈ V
                        flag ← flag ∧ (⟨u,w⟩ ∉ E ∨ ⟨v,w⟩ ∈ E)
                if flag then R_≤ ← R_≤ ∪ {⟨u,v⟩}
```

Algorithm NEST: Compute nesting relation $R_{\preceq}$

Note that this presentation of algorithm NEST is just a thinly disguised boolean matrix multiplication. Let $\mathbf{M}^\dagger$ denote the transpose of a square boolean matrix $\mathbf{M}$, and $\overline{\mathbf{M}}$ denote its element-wise complement. Then the matrix $\mathbf{R}$ of the relation $R_\preceq$ is related to the adjacency matrix $\mathbf{G}$ of the graph G by the equation $\overline{\mathbf{R}} = \mathbf{G}\overline{\mathbf{G}}^\dagger$. It might seem appealing to use standard algorithms for computing products of boolean matrices quickly (for example, see Aho[pages 242-247][1]) to expedite computing nestability, since computing $\mathbf{M}\overline{\mathbf{M}}^\dagger$ for an arbitrary boolean matrix $\mathbf{M}$ is as hard (up to a constant factor) as multiplying two boolean matrices $\mathbf{A}$ and $\mathbf{B}$. Letting

$$\mathbf{M} = \begin{pmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \overline{\mathbf{B}}^\dagger \end{pmatrix} \text{ means that}$$

$$\mathbf{M}\overline{\mathbf{M}}^\dagger = \begin{pmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \overline{\mathbf{B}}^\dagger \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{1} \\ \overline{\mathbf{A}}^\dagger & \mathbf{B} \end{pmatrix} = \begin{pmatrix} \mathbf{A}\overline{\mathbf{A}}^\dagger & \mathbf{AB} \\ \overline{\mathbf{B}}^\dagger\overline{\mathbf{A}}^\dagger & \overline{\mathbf{B}}^\dagger\mathbf{B} \end{pmatrix}$$

However, the matrix $\mathbf{G}$ is not arbitrary here, since it was constructed by identifying isomorphic subtrees of a tree. In particular, the number of edges in G (as well as the number of vertices) is bounded by the number of nodes in the original tree. Cases where G and $R_\preceq$ are both fairly sparse should be handled much more efficiently, since they will arise most often in practice. This means that fast *general* matrix multiplication algorithms should not be used here; they can only hope to approach quadratic running times (from the obvious cubic implementation). Algorithm NEST can be made to run in time $O(\|V\| \cdot \|E\|)$ as follows: store G using sorted adjacency lists, so that the inner loop "**foreach** $w \in V \ldots$" is actually implemented as "**foreach** $w \in V$ such that $\langle u, w\rangle \in E$ flag $\leftarrow$ flag $\wedge \langle v, w\rangle \in E$." This means that the running time will be $O(n^2)$, where n is the number of trees in the original structure. (This observation follows directly from a simple method for multiplying sparse matrices.)

Actually G could be a multigraph, since a node could have two or more isomorphic children. This does not add any inherent extra difficulty to the problem of computing nestability.

There is another method for computing nestings that is useful when the maximum number of children of a tree is small (and hence the graph G is of bounded out-degree). If every vertex in G has at most k successors, then the relation $R_\preceq$ can be computed in expected time $O(\|V\| \cdot 2^k + \|R_\preceq\|)$. Let the hash table H be a map from multisets of vertices in V to sets of vertices in V; $H: \mathbf{N}^V \mapsto 2^V$. Initially $H(S) = \emptyset$ for all multisets $S \in V$.

The following algorithm computes the nesting relation $R_\preceq$ in two passes through the vertices. In the first pass, all subsets of the successors of each vertex are entered in the hash table; the second pass looks up each vertex by the full set of its successors.

```
R_≤ ← ∅
foreach u ∈ V
        let S = {v | ⟨u,v⟩ ∈ E}
        foreach S' ⊆ S
                H(S') ← H(S') ∪ {u}
        foreach u ∈ V
                let S = {v | ⟨u,v⟩ ∈ E}
                foreach v ∈ H(S)
                        R_≤ ← R_≤ ∪ {⟨u,v⟩}
```

Algorithm NEST2: Compute $R_{\preceq}$ when out-degree is bounded

Of course, the exponential dependence on k makes algorithm NEST2 unusable even in sparse graphs with any vertices of high out-degree. A practical algorithm for general sparse graphs with subquadratic running time is yet to be found. Research into this area might also yield faster algorithms for multiplying sparse boolean matrices and computing transitive closures of sparse graphs.

2.8 Matching

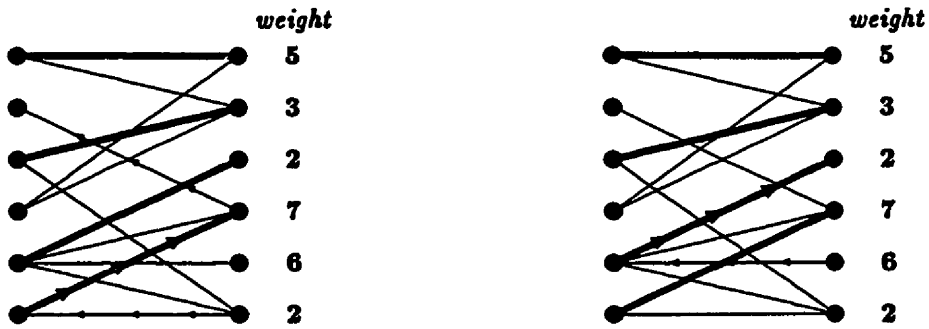
This step might seem potentially costly, but in practice it should be quick. Once the relation $R_{\preceq}$ is found, a weighted bipartite matching problem must be solved in order to find the minimum weighted covering of $R_{\preceq}$ by chains. The weights on the edges of the bipartite graph are not general here, since for one of the parts, all edges leaving a given vertex in that part have the same weight. This fact simplifies finding a maximum matching using an augmenting path method (see Papadimitriou[39, chapter 6], or Tarjan[44, chapters 8 and 9]) over the case where the weights are general.

Let the bipartite graph $G = (X, Y, E)$ have positive weights associated with the vertices in X. Then the augmenting paths in G all have one end at some unmatched vertex $x \in X$ and follow unmatched and matched edges in E alternately. These paths are of two types:

1. an alternating path ending at an unmatched vertex in Y. These paths are the exactly the augmenting paths in the unweighted graph; and in the weighted graph they increase the weight of the matching by the weight of the initial vertex x.

2. an alternating path ending at a matched vertex $x' \in X$ where the weight of x' is less than the weight of x. These paths do not increase the cardinality of the set of matched edges, but they do increase the weight of the matching by the difference in weight between x and x'.

The diagram of figure 2.8 illustrates these two types of paths.



Type 1 augmenting path (value 2) *Type 2 augmenting path (value 4)*

(matched edges are heavy)

Figure 2.8: Two types of augmenting paths

The augmenting paths can be found by depth-first or breadth-first search, so each one might take as long as $O(\|E\|)$ time. If n is the number of nodes in the original tree, then the number of times an augmenting path will be found must be bounded by $2n - 1$, since each path causes a savings of at least one array location in the final representation. The number of edges here in G might be as high as $O(n^2)$ (this is unlikely, though), so in the worst case, it might take $O(n^3)$ steps to do the matching.

Conceivably, the special form of the weights allows an asymptotically more efficient scheme (like that of Dinic[11] when applied to unit networks; see Even[15] or Hopcroft[22]), but the extra complication introduced would rarely be worth the trouble. The graph G is probably sparse in practice, and the time to find augmenting paths is likely to be almost constant, so the method described above should exhibit nearly linear behavior anyway.

2.9 An application: English lexicon tree

In 1983, Andrew Appel and I wrote a program to play Scrabble[3]. To represent the legal English words, we used a tree structure. Nodes in the tree were labeled with letters of the alphabet, and the labels along the paths from the root to specially marked nodes formed the set of acceptable words. This data structure (a cdr-coded list stored in an array) was chosen because it was convenient for move generation.

Unlike the trees discussed earlier in this chapter, the trees in this application have some extra information associated with each node (letters of the alphabet). We can still apply the results of this chapter, as long as we change our notion of isomorphism slightly. To be considered isomorphic, two subtrees must agree not only in structure, but also in the information found at the nodes.

The word list we used contained roughly 94,000 words. Using the straightforward translation, the array would have contained about 180,000 locations. We discovered that by merging isomorphic subtrees, the number of array locations is reduced to about 60,000. This is an amazing savings, especially when you consider that there is no loss in search efficiency, and no need to change the Scrabble program at all.

This is how the matter stood, until I developed and programmed the the optimization algorithm for unordered trees stored as cdr-coded lists described in this chapter. Using this new program, I found that the minimum number of cells needed to store the tree is about 51,000. Again, the savings here is free, since no modification to the program is required, and the search runs just as fast.

I also tried representing the lexicon with cons-cells, after choosing an alphabetical ordering of the children. I found that about 43,000 cells are needed to store the same English lexicon. Of course, each cell requires two pointers rather than one, so this is not as efficient a representation as the cdr-coded one for this application.

2.10 Future work

2.10.1 Analysis of other implementations

This chapter presented an average-case analysis of the cons-cell representation of binary trees. Of course, that analysis also applies to the cons-cell representation of general ordered trees. What about cdr-coded representations? The techniques of analysis presented earlier probably can be applied here as well, when dealing with binary or ordered trees.

Analyzing the average-case space requirements for *unordered* trees would be much more difficult. Computing the expected weight of the maximum matching requires a more sophisticated approach than has been presented here.

2.10.2 Concrete optimization of graphs

What about other linked structures? Although this chapter discussed only trees, the techniques presented here can be applied to other linked structures. When a graph is stored as an adjacency list, the tree-minimization algorithms described in this chapter can be applied directly. Only the first step, where isomorphic nodes are labeled, needs to be changed. In graphs, isomorphism labeling can be performed efficiently by a partition refinement finite-state machine minimization algorithm (for example, see Hopcroft[31, pages 189–196]).

Chapter 3

Abstract Optimization

In this chapter I present a framework for studying abstract optimization of static data structures, and discuss metrics for measuring the computational resources involved.

Suppose we have an abstract class C_n of static data objects, and a set of operations S that examine a data object but do not modify it. Each member of C_n can be viewed as a collection of partial functions, one function corresponding to each operation in S. The domain and range of these functions can be either predefined data types (like `integer` or `boolean`) or they can be *indices*. These indices are meant to be the abstract analogues of pointers; they can only be used and returned by operations in S.

An *implementation* of an abstract class provides a mapping from elements of C_n into a read-only memory, and a program for each operation in S that references this memory. An implementation also provides a mapping between elements of the index domains and small pieces of memory. All of these mappings are strictly internal to the implementation, and cannot be referenced by a program that makes use of the data type.

The abstract data types I study have natural implementations that use too much space. Optimization means making something better. A *better* implementation of these data types has the same functionality as the natural implementation, but uses less space. The trick of abstract optimization is to trim the fat in the data without slowing down the operations too much. How much space has been saved? How much slower is the optimized implementation? To provide meaningful answers to such questions, we need to use a model of computation that defines precise cost metrics for space and time, and that is realistic about computers' capabilities.

3.1 Space metrics

The (worst-case) space cost of an implementation is simply the maximum length of any of the bit-strings representing an element in C_n. This is a strictly log-cost accounting of space. Since space-efficiency is the primary concern here, I cannot afford to be sloppy and measure space in *words*, which hold an unspecified amount of information. It is always possible to make use of all the bits in a computer word.

Bits are universal. While it *is* possible to buy a computer that does more work per unit time, it *is not* possible to buy a computer that stores more per bit. In other words, the time required for a given operation can only be bounded by a functional form, whereas the space required can be bounded absolutely. It would be foolish to use any metric for space other than bits.

We wish to represent the elements of C_n in such a way that the maximum length of an element, measured in bits, is a slowly-growing function of n. How should we measure the space-performance of our representation-length function? Information theory provides us with an excellent yardstick. We know that there are $||C_n||$ different elements in C_n. It follows that there some objects must have be at least $\lg ||C_n||$ bits; otherwise, the objects could not all have distinct representations. This absolute lower bound on the size of any possible representation gives us something to shoot for. Let's classify our succinct representations into one of three categories, in decreasing order of desirability:

canonical is the best we can hope for. This is a mapping from C_n into the integers $1 \ldots ||C_n||$. The resulting integer is then encoded as a $\lg ||C_n||$-bit binary number.

asymptotically optimal is a little worse than canonical. This is a mapping from C_n into a bit string of length $\lg ||C_n|| \cdot (1 + o(1))$. Some wasted space is allowed in this type of representation, but as n grows, the fraction of waste must vanish.

linear maps C_n into bit strings whose length is $O(\log ||C_n||)$.

These categories are based on the functional form of the extra space required over $\lg ||C_n||$ bits. In abstract optimization, I will not insist on canonical representations, but I do strive for asymptotic optimality.

3.2 Time metrics

The choice of metric for time is not so clear-cut. The unit-cost model of computation is a popular accounting metric for time, and with good reason. This model usually has the most realistic correspondence to observed running time. The pitfalls of the unit-cost model when numbers get too large are well known. Less obvious, but just as nasty, are the architecture-specific shortcomings of this metric. The unit-cost model assumes some kind of word-size bit parallelism exists within the circuits of a computer. When the logarithms of the numbers involved stay below the word size, it is reasonable to expect to perform *certain* operations with this degree of parallelism. But the circuits in any given computer are fixed, so we may be out of luck when we try to coerce a computer into performing a particular word-size operation for which it is ill-suited.

Let me provide a specific example of this phenomenon. Suppose a critical step of an algorithm involves counting the number of 1 bits in a binary number. Assume that the typical number n we are dealing with is small enough to fit in a single computer word of w bits . How much time should we account for this bit-counting operation? If we get to choose, we can use a computer with a `bit-count` instruction. It would seem reasonable then to assess a cost of one to bit counting. But many computers lack an explicit `bit-count` instruction. Should we loop through the bits of the word testing for ones, and charge w? Should we use a clever sequence of $\lg w$ shifts, masks, and additions to compute the bit-count of n? Or should we break each word into k chunks, keep a table of size $2^{w/k}$ of precomputed bit-counts, and charge k (generally the most efficient scheme in practice)? Further complications arise if our computer doesn't have a multi-place shift instruction. The best implementation of this operation, and hence its accounting, depends on the architecture.

To avoid these processor-specific pitfalls, I restrict my attention to metrics based on bus transactions between the processor and the static data. I will charge for, and only for, each reference the processor makes to the data. The processor is allowed to perform arbitrary computations at no cost with the data it has already has in hand. I am measuring I/O complexity.

By varying the bandwidth of the buses and the costs of transactions, we get different metrics. Two that I like particularly well, and use extensively in the thesis are:

data-bits model assumes that the data bus from memory to the processor is only one bit wide. We simply count the number of bit-accesses performed to get the time cost.

wide-bus model assumes that the data bus is $\lg N$ bits wide, where N is the size of the memory. We can fetch $\lg N$ bits from consecutive memory locations at unit cost.

In both models we assume that the address bus is sufficiently wide to address any bit contained in memory.

The data-bits model has simplicity as its main advantage. There is a strong analogy between accounting time as bit-accesses here and accounting the time used by sorting algorithms in element comparisons. If nothing else, our time metric gives a very reasonable lower bound on achievable asymptotic performance, as long as the bandwidth of the data bus is fixed. Two disadvantages of this model are that it does not reflect the inherent word-size parallelism in the data buses of real computers, and that it does not take account of the bounded bandwidth of the *address* bus.

The wide-bus model, on the other hand, is more realistic about the inherent parallelism in the buses of computers. Still, there is something a little disturbing about the size of the bus growing along with the size of the data.

3.3 Binary trees: an example

Let me make the model developed in this chapter clear by giving an example. Let our class C_n be the set of binary trees with n nodes, and let the set of operations S be:

- a function of no arguments (a constant) `root` returning `node`.
- two functions, `car` and `cdr`, mapping `node` to `node`.
- a function `null` mapping `node` to `boolean`.

The type `node` is an index type. This is the abstract analog of the pointers that occur in the natural implementation. Note that there is no information stored in the tree; the only operations we can perform are moving from a node to its children and testing if a node exists.

3.3.1 A natural implementation

A natural implementation uses a `cons` cell (a pair of pointers) for each node. Externally, each node is referenced by a pointer to its `cons` cell.

Each pointer requires about $\lg n$ bits, since there are n different nodes. The operations is S can be performed by dereferencing a pointer. This takes time $O(\log n)$ in the data-bits model, and time $O(1)$ in the wide-bus model.

The space used for each `cons` cell is about $2 \lg n$ bits. So the total space for an n-node tree is $O(n \log n)$. We could cut this space nearly in half by storing the `cdr`s in consecutive memory

locations, so we only need pointers to the cars. But this does not change the basic $O(n \log n)$ space performance.

3.3.2 A canonical implementation

The number of different binary trees of n-nodes is C_n, the famous Catalan numbers. They are defined by:

$$C_n = \frac{1}{n+1}\binom{2n}{n} \tag{3.1}$$

Let us represent a tree by an integer in the range $1 \ldots C_n$. This canonical representation would take $\lceil \lg C_n \rceil$ bits. Considering the combinatoric meaning of the *choose* operator, it is clear that $\binom{2n}{n}$ is strictly less than 2^{2n}. The number of bits used by the canonical representation is thus bounded above by $2n$. Of course, there is no obvious way to represent individual nodes in such a representation to allow efficient performance of the operations in S.

3.3.3 Optimization

Do we really need to store $O(n \log n)$ bits to obtain the time-performance of the natural implementation? No. The next chapter describes a scheme that uses linear space. The trick employed there is to use a variable-length code for the pointers. Although the largest pointers are $O(\log n)$ bits long, the average pointers are only $O(1)$ bits. Summed over the entire tree, the total size of all the pointers is $O(n)$.

Another solution for this problem is presented in chapter 5. The representation given there uses asymptotically optimal space, and doesn't use anything resembling a pointer.

Chapter 4

Abstract Optimized Trees

This chapter addresses the problem of representing static trees in space proportional to their information content, without sacrificing more than a constant factor in time efficiency (of the basic traversal operations) over conventional pointer methods. A family of recursive representations with this desirable economy is exhibited, parameterized by prefix codes for the natural numbers. A simple and practical representation is described and analyzed. Next, properties of the optimal representation in the family are presented and partially analyzed. A discussion of practical considerations and extensions to these techniques concludes the chapter.

4.1 Introduction

How much information is encoded in the shape of a tree? The number of distinct unlabeled trees with n nodes grows functionally as k^n (times a subexponential factor), for some constant k. Knuth[29, section 2.4.4.4] gives the value for k for several classes of trees. The information content of a tree of n nodes is about $\log k^n$, or cn for some c. It should therefore be possible to represent a tree using only a constant number of bits per node.

This is easy to do, in a number of ways. We could simply assign each distinct tree a canonical number. This would be optimal in terms of space, although it is difficult to see how we could perform tree-traversal operations using this representation. The local structure of the tree is not reflected anywhere in such numbers.

Let's start by considering n-node binary trees, since they are easy to analyze and are equivalent to general ordered trees with $n + 1$ nodes using the well-known trick of tilting the general tree by 45 degrees (see Knuth[29, section 2.3.2]). One way to represent both binary and general trees in

a linear number of bits is to use the structural correspondence between ordered trees and strings of balanced parentheses. This correspondence is best described recursively: the empty tree is represented by the empty string, and other trees are represented by an open '(', followed by the concatenations of the representations of the tree's children, followed by a close ')', as in figure 4.1. (This is just the LISP S-expression for the tree.) Trees of n nodes are represented by strings of

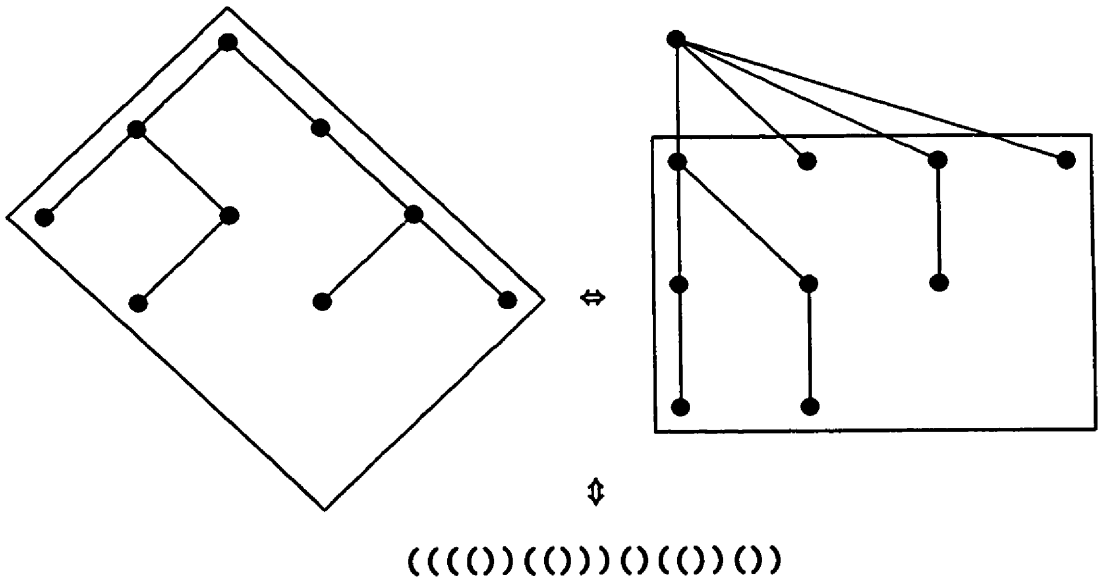


Figure 4.1: Balanced parentheses as trees

length $2n$. Since each character in the string is either a '(' or a ')', we could use 0's and 1's to encode the n-node tree with $2n$ bits.

Although this representation is not informationally perfect, (since an unbalanced string of parentheses does not represent any tree), it is quite efficient. As we use this scheme to represent ever larger trees, the fraction of bits wasted vanishes. The number of balanced strings of $2n$ parentheses is given by the nth Catalan number $C_n = \binom{2n}{n}\frac{1}{n+1}$. By Stirling's approximation, $C_n = 4^n \cdot \Theta(n^{-3/2})$. Taking logarithms base 2, $\lg C_n = 2n + o(n)$, so two bits per node is the best possible asymptotic bound on the storage needed.

Contrast the representation as a string of parentheses with the common linked (pointer) representation for such trees. Every node in the tree, except perhaps the leaves, stores one or more pointers to children. Since there are n nodes in the tree (each having a unique address), the pointers

must be at least $\log n$ bits wide. So this scheme takes $\Omega(n \log n)$ bits, although the cost is linear in the unit cost model of space.

The $\log n$ blowup in space is compensated by the superior speed of the linked representation for accessing parts of the tree. The two common tree-traversal primitives `Right-child` and `Left-child` (equivalently, `First-child` and `Next-sibling`, or `car` and `cdr`) can be accomplished in constant time under the unit cost measure, and in logarithmic time under the log-cost measure. To move around in a tree structure represented as a string of balanced parentheses, it will sometimes be necessary to scan through a large fraction of the string, matching parentheses, to find the `Right-child`. The worst-case cost of this operation is $\Omega(n)$, much worse than the cost in the linked representation.

Another representation that uses just two bits per node is *marked preorder sequential*, described by Smith[42, page 225]. Storing the nodes in preorder, we keep a pair of bits called `Left-child-empty?` and `Right-child-empty?` for each one. The shape of the tree can be reconstructed from this information, but locating the positions of the children in such a tree still requires a linear scan through the data.

Is it possible to enjoy representational efficiency without sacrificing too much in speed? The answer is yes, and a class of representations achieving this is developed in the next section.

4.2 Encoding trees in linear space

Although the space to store a pointer in the linked representation is $\Omega(\log n)$, most of those bits do not carry their weight in information. When the nodes in a tree are laid out in preorder, the left child of a node immediately follows its parent in memory, and the right child is probably not too far ahead, since most of the nodes in a (balanced) tree are near the bottom level . If we were to represent pointers as relative addresses rather than absolute addresses, the numbers stored in the pointers would be smaller, so they could, on average, be represented with fewer bits. Near the top of the tree, the relative pointers will still be large (around $\log n$ bits), but if we are clever, the total size of all the pointers summed over the whole tree will be only $O(n)$. Let's represent our binary tree recursively as a string of bits as shown in figure 4.2

The header block tells us the relative sizes of the left and right subtrees; it is followed by the encoding of the left subtree concatenated with the encoding of the right subtree. The empty tree is represented by the empty string. A position in the tree is represented by an index into this

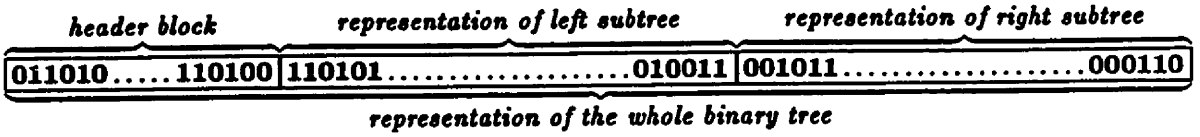


Figure 4.2: Recursive layout of a binary tree

bit string (the start of the block corresponding to the desired subtree), along with the size of the current subtree (so we know how large the block is).

A test for emptiness can be performed by checking if the stored size of the subtree is zero. To move to the left child, simply skip the header block, adjust the size, and position the new index at the end of the header. To move to the right child, take the same action, except that the new position is computed by adding the size of the left subtree (somehow computed from the header) to the position at the end of the header.

The header encodes the number of nodes in the left subtree as an integer n, which functions as a relative pointer to the right subtree. To find the size in bits of the left subtree, we compute a function B_n, the maximum number of bits in the representation of any n-node tree. We will insist that all n-node trees occupy *exactly* B_n bits, padding them if needed. This allows us to store the sizes of our blocks as the number of nodes encoded therein, rather than as a count of bits.

To make this scheme work, we have to know when we are done reading through the header. Since the number of nodes in a tree is unbounded, we obviously cannot put a fixed ceiling on the number of bits in any code that represents these numbers. However, there are well-known methods of encoding integers as self-terminating bit strings (called *prefix codes* by Elias[13]), that use a logarithmic number of bits; that is, the representation of n takes only $\Theta(\log n)$ bits. The methods in the literature are concerned mainly with the asymptotic efficiency of the representations (for example, see Stout[43]), and strive to minimize functional form of the representation achieving $\lg n + \Theta(\log \log n)$, $\lg n + \lg \lg n + \Theta(\log \log \log n)$ and beyond. Here, though, it will be important to pay close attention to the representation of very small integers, since many of the subtrees will be small. Let us assume for the moment that we use any prefix code that achieves logarithmic succinctness.

Although the numbers representing the sizes of the largest subtrees of an n-node tree have $\log n$ bits, we will show that the large number of small subtrees causes the total number bits

used, summed over the entire tree, to be only linear in n. To see the basic reason for this, let us (erroneously) suppose that the worst-case occurs when our trees are balanced, complete binary trees. If B_n denotes the space required for a tree of n nodes, then, informally,

$$\begin{aligned} B_n &= O(\log n) + 2B_{\lfloor \frac{n}{2} \rfloor} \\ &= \sum_{i=0}^{\log n} 2^i \cdot O(\log(n/2^i)) \\ &= O(n) \end{aligned} \tag{4.1}$$

so at each lower level of the tree, the relative pointers (really, the encodings of the sizes of the subtrees) require one fewer bit, leading to amortized constant space per node.

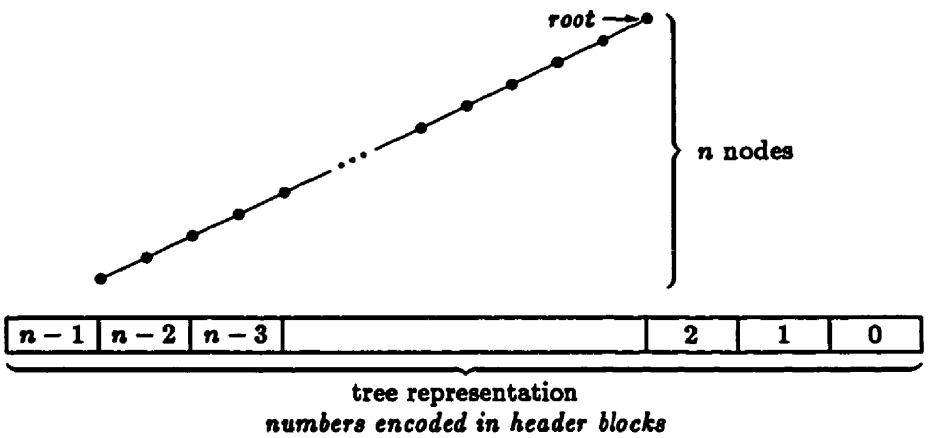


Figure 4.3: Left leaning tree

Unfortunately, this analysis does not apply to arbitrary trees. Consider a tree which leans all the way to the left, shown in figure 4.3. For this tree, the size of the relative pointer doesn't decrease very much from one level to the next, and the representation described above will take $\Omega(n \log n)$ bits. We can fix the representation by adding a `Left-child-first?` bit for every node telling which of its subtrees has more nodes, and encoding the smaller subtree first, shown in figure 4.4. This guarantees that the size of the pointers must shrink by at least one each level, since the smaller subtree has fewer than half as many nodes as the whole tree.

Now we will get a linear space representation as long as we use a prefix code for the natural numbers that uses $\Theta(\log n)$ bits to encode the integer n. The choice of which code we use has

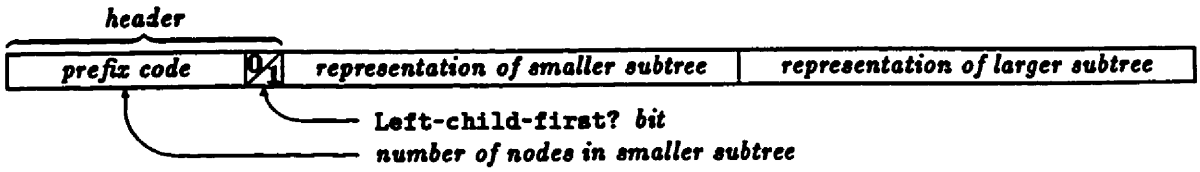


Figure 4.4: Layout of a binary tree with `Left-child-first?` bits

an important effect on the asymptotic constant of linearity. To access a child, we must examine $O(\log n)$ consecutive bits to read in a prefix code (and process those bits). This takes $O(\log n)$ time in the data bits model, but only $O(1)$ time in the wide bus model. This representation scheme is therefore as fast as a pointer representation, up to a constant factor. A complete analysis of this scheme with a particular prefix code comprises the next section.

4.2.1 Analysis of a practical encoding

Let's use the following very simple prefix code, with encoding function R taking the natural numbers to binary strings. Define

$$\begin{aligned} R(0) &= \mathbf{0} \\ R(n>0) &= \mathbf{1} \cdot [n \bmod 2] \cdot R(\lfloor n/2 \rfloor) \end{aligned} \tag{4.2}$$

here $\cdot$ denotes concatenation, and $n \bmod 2$ is either a **0** or a **1** depending on whether n is even or odd. Let $r(n)$ denote the length in bits of $R(n)$:

$$\begin{aligned} r(0) &= 1 \\ r(n>0) &= r(\lfloor n/2 \rfloor) + 2 \end{aligned} \tag{4.3}$$

So in particular, $R(0) = \mathbf{0}$, $R(1) = \mathbf{110}$, $R(2) = \mathbf{10110}$, $R(3) = \mathbf{11110}$, and $R(4) = \mathbf{1010110}$. This encoding amounts to taking the standard binary representation of n (which, of course, is not a prefix code) reversing it, and shuffling it with a binary string of the form $\mathbf{1}^k\mathbf{0}$. To decode a bit string, begin reading the bits until you encounter a **0** in an odd position. The bits in even positions, reading backwards, form the standard binary encoding of the integer.

This is not a particularly efficient encoding of the integers; it asymptotically requires twice as many bits as the most efficient schemes, and it even wastes some codes (for example, **100** doesn't begin the code for any integer). We use this encoding because it is easy to analyze.

To simplify the analysis, define the *integer logarithm* function $l(n)$ on the natural numbers to be $\lfloor \lg(n+1/2) \rfloor$. (This is really just $\lfloor \lg n \rfloor$, except that it is defined to be -1 for $n = 0$.) This function has the property that

$$l(\lfloor n/2 \rfloor) = l(n) - 1 \quad \text{for all integral } n > 0 \tag{4.4}$$

Note that the code length function $r(n) = 2l(n) + 3$.

The representation scheme for trees is as follows:

$$\begin{aligned} \mathcal{R}(\textit{empty tree}) &= \epsilon \\ \mathcal{R}(T) &= R(\textit{number of nodes in smaller subtree}) \\ &\quad \cdot[\texttt{Left-child-first? bit}] \\ &\quad \cdot\mathcal{R}(\textit{smaller subtree}) \\ &\quad \cdot\mathcal{R}(\textit{larger subtree}) \end{aligned} \tag{4.5}$$

Let B_n be the maximum number of bits needed to store a tree of n nodes. $B_0 = 0$; for positive n,

$$B_n = \max_{0 \le k \le \lfloor (n-1)/2 \rfloor} [r(k) + 1 + B_k + B_{n-1-k}] \tag{4.6}$$

(The $n-1$'s in this important equation are due to the fact that the subtrees can have at most $n-1$ nodes, since there is one node at the root.)

Theorem 4.1 *The space B_n for a tree of n nodes is given by:*

$$B_n = 4n - 2l(n) - 2 \tag{4.7}$$

Proof by induction on n. $B_0 = 4 \cdot 0 - 2 \cdot (-1) - 2 = 0$, verifying the base case. From equation 4.6,

$$\begin{aligned} &\max_{0 \le k \le \lfloor (n-1)/2 \rfloor} [r(k) + 1 + B_k + B_{n-1-k}] \\ &= \max_{0 \le k \le \lfloor (n-1)/2 \rfloor} \left[\begin{array}{l} (2l(k)+3) + 1 + (4k - 2l(k) - 2) \\ \quad + (4(n-1-k) - 2l(n-1-k) - 2) \end{array} \right] \\ &= \max_{0 \le k \le \lfloor (n-1)/2 \rfloor} [4n - 4 - 2l(n-1-k)] \\ &= 4n - n - 2 \cdot \left[\min_{0 \le k \le \lfloor (n-1)/2 \rfloor} l(n-1-k) \right] \\ &= 4n - 4 - 2l(n - 1 - \lfloor (n-1)/2 \rfloor) \end{aligned} \tag{4.8}$$

$$
\begin{aligned}
&= 4n - 4 - 2l(\lfloor n/2 \rfloor) \\
&= 4n - 4 - 2(l(n) - 1) \qquad (4.9) \\
&= 4n - 2l(n) - 2 \\
&= B_n
\end{aligned}
$$

demonstrating the induction case. Equation 4.8 follows from the monotonicity of the integer logarithm function $l(n)$, and equation 4.9 follows from equation 4.4. □

The limit of B_n/n as n grows without bound is 4 (since $l(n)$ is $o(n)$), so this scheme stores trees in about four bits per node.

This representation for binary trees is quite practical and easy to implement. The decoding of the relative pointers from the bits strings is straightforward, as is the computation of B_n from n. The space required by this scheme is asymptotically only a factor of two worse than the space required by *any* possible representation. The time required is within a constant factor of that used by the standard pointer representation.

Because this representation uses an inefficient scheme to encode the integers, it is logical to look for schemes using more efficient integer encodings, to better approach the asymptotic bound of two bits per node. Any logarithmically succinct prefix code leads directly to a representation scheme, based on the solution to the recurrence in equation 4.6 with a different function $r(n)$. For practical purposes, it is desirable that B_n be efficiently computable, since this computation must be performed in traversing the tree. The encoding above yields a particularly simple form for B_n; for some other representations I have examined it is more difficult to find a simple formula. Nevertheless, since the model of computation used here only counts bit-accesses to the data, we can assume that B_n can be computed for free given any chosen encoding of the integers. This at least gives a lower-bound for those schemes where B_n is easy to compute.

4.3 Lowering the constant factor

Imagine the following quasi-mechanical search for an efficient representation:

> For each prefix code R for the natural numbers, plug the code length function r for that encoding into equation 4.6 and solve the recurrence to obtain a formula for B_n. Find the asymptotic limit of B_n/n and minimize.

There are two problems with this approach:

1. How can you enumerate all the valid prefix codes of the natural numbers? There are infinitely many of them.

2. Given a particular code length function r, how can you solve the recurrence of equation 4.6 to obtain B_n/n, or even the asymptotic value of B_n/n?

In order to find an efficient representation, let's generalize our scheme slightly. Instead of employing one fixed prefix code R to encode all the relative pointers, use a family of codes R_n. We intend to use the code R_n to encode the relative pointer at the root of all trees containing n nodes. (Remember that we keep track of the number of nodes in the current tree anyway, to tell when the tree is empty.) If a tree has n nodes, the number of nodes in a subtree is between 0 and $n-1$ inclusive, so R_n need only map $[0 \ldots n-1]$ into strings of bits. This generalized scheme also allows us to do without the `Left-child-first?` bit; that is, R_n can directly encode the number of children in the left subtree in a tree of m nodes, as shown in figure 4.5. The new R_n simply assigns short codes to the integers near 0 *and* those near $m-1$.

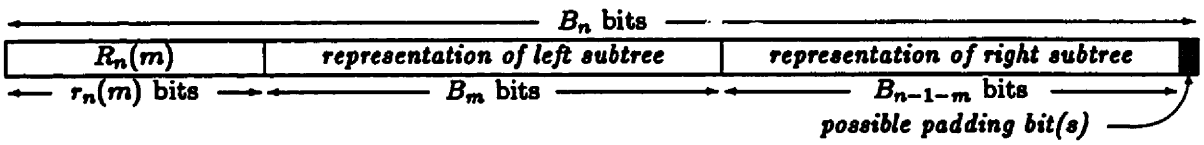


Figure 4.5: Layout of an n-node binary tree with generalized R_n

Call representations in the new scheme *variable encodings*, to distinguish them from the *uniform encodings* in the old scheme for representing trees.

First, note that variable encodings strictly generalize uniform encodings; any uniform encoding R' can be regarded as a variable encoding R_n that doesn't change as n grows (except inasmuch as the extra `Left-child-first?` bit is subsumed in the new R_n). Therefore representations using variable encoding must be at least as good as those using uniform encoding. Under variable encodings, the recurrence of equation 4.6 becomes:

$$B_n = \max_{0 \le k < n} [r_n(k) + B_k + B_{n-1-k}] \tag{4.10}$$

where r_n is, analogously, the length of R_n.

At first it would seem that this added generality makes the search for efficient representations even more difficult. The space of possible encoding functions is now indexed by two variables,

rather than one. However, this generality actually simplifies the search tremendously, as we will show in the next section.

4.3.1 The optimal variable encoding

Now that we allow different encodings of the relative pointers in different size trees, a wonderful thing happens. The size of n-node trees, B_n, now depends only on the values of B_k for $k \leq n$, and on the function r_n, whereas before it depended on the uniform function r, which had to be the same for all n. This independence of B_n from the encodings R_m $(m \neq n)$ together with equation 4.10 means that the principle of optimality holds. For B_n to be minimal, each B_k must also be minimal for $k < n$. This suggests the following dynamic programming algorithm which computes optimal (minimal) values of B_n, along with encoding functions that achieve those values:

$B_0 \leftarrow 0$
for $n \leftarrow 1 \ldots \infty$
    $B_n \leftarrow \infty$
    for each encoding R_n of $[0 \ldots n-1]$
        $r_n \leftarrow \|R_n\|$
        $b \leftarrow \max_{0 \leq k < n} [r_n(k) + B_k + B_{n-1-k}]$
        $B_n \leftarrow \min[B_n, b]$

Algorithm A

For the rest of this section, we will use the notation B_n to mean the optimal value of B_n over all variable encodings. Note again that this B_n is an absolute lower bound on all possible uniform encodings, for reasons mentioned earlier.

The search over all possible encodings in the algorithm above seems daunting. Actually, no search is required at all. From the algorithm, $r_n(k)$ can be as large as $B_n - B_k - B_{n-1-k}$; that is, the encoding of the integer k in variable encoding R_n can have as many as $B_n - B_k - B_{n-1-k}$ bits. (Of course, we don't know what B_n is, yet.) Because R_n is a prefix code, the set of strings $\{R_n(k) \mid 0 \leq k < n\}$ must have the *prefix property*: no $R_n(k_1)$ may be a proper prefix of another $R_n(k_2)$. We can find an encoding R_n of $[0 \ldots n-1]$ with respective lengths $[r_n(0) \ldots r_n(n-1)]$ with the prefix property exactly when the values of $r_n(k)$ satisfy Kraft's[32] inequality for noiseless encoding (better explained by Gallager[19, page 47]):

$$1 \geq \sum_{k=0}^{n-1} 2^{-r_n(k)} \tag{4.11}$$

now, using $r_n(k) \leq B_n - B_k - B_{n-1-k}$, substituting:

$$1 \geq \sum_{k=0}^{n-1} 2^{B_k + B_{n-1-k} - B_n}$$

multiplying by 2^{B_n}

$$2^{B_n} \geq \sum_{k=0}^{n-1} 2^{B_k + B_{n-1-k}}$$

now taking logarithms and minimizing B_n (remembering that B_n is constrained to be an integer) we obtain the following recurrence:

$$B_n = \left\lceil \lg \sum_{k=0}^{n-1} 2^{B_k + B_{n-1-k}} \right\rceil \tag{4.12}$$

A search-free dynamic programming algorithm to compute the values of B_n follows directly from the recurrence of equation 4.12:

```
B_0 ← 0
for n ← 1...∞
    t ← 0
    for k ← 0...n − 1
        t ← t + 2^(B_k + B_{n−1−k})
    B_n ← ⌈lg t⌉
```

Algorithm B

It still remains to construct prefix codes R_n that achieve these values of B_n. We can work backwards, starting from the B_n computed by algorithm B. In constructing prefix code R_n, assign the integer k a *weight* $W_k = 2^{B_k + B_{n-1-k} - B_n}$. Because we chose the B_n to satisfy Kraft's inequality, we know that $\sum W_k \leq 1$. Now, use Huffman's[23] classic algorithm to find a prefix code of minimum total weight. A well-known property of such minimal-weight codes is that $W_k < 2^{-r_n(k)+1}$ (see Gallager[19, page 50]); in our case, this means that $r_n(k) \leq B_n - B_k - B_{n-1-k}$, which is the bound we need.

These algorithms to compute B_n and R_n given n are all that is needed (at least in theory) to implement the optimal variable encoding of binary trees. To make the optimal scheme practical, it is necessary that values of B_n and decodings via R_n be computed quickly. Although our model of computation doesn't charge us for these computations, algorithm B above takes $O(n^2)$ time to compute B_n.

n	B_n	n	B_n
1	0	600	1360
2	1	700	1589
3	3	800	1818
4	5	900	2047
5	7	1000	2276
6	9	2000	4567
7	11	3000	6859
8	13	4000	9151
9	15	5000	11443
10	17	6000	13736
20	38	7000	16028
30	60	8000	18320
40	82	9000	20613
50	105	10000	22905
60	127	20000	45830
70	150	30000	68756
80	173	40000	91681
90	195	50000	114607
100	218	60000	137533
200	446	70000	160460
300	674	80000	183386
400	903	90000	206312
500	1132	100000	229238

Table 4.1: Some values of B_n

It might seem that the computations involving t in algorithm B would be expensive to perform because of the extremely large size of the numbers involved, when n is large. The trick is to store t as a floating point number with binary exponent and mantissa stored in separate integer variables. If the mantissa is maintained as a normalized fixed-point fraction $1/2 < m \leq 1$, with the exponent e an integer, then $t = m \cdot 2^e$ can be maintained as an invariant of the inner loop with only a constant number of (unit cost) operations. The number of bits in the mantissa need only be a few bits larger than the difference between the longest and shortest code in the range of R_n. This difference is logarithmic in n, and in practice, the mantissa can be stored in a single 32-bit machine word for values of n into the hundreds of thousands. The value of the exponent e at the end of the loop is B_n.

Maybe there is a simple closed form for B_n (or at least a more efficient algorithm than algorithm B), but such a formula has eluded me. The next section discusses some of the properties of B_n and the recurrence of equation 4.12.

4.3.2 The function B_n

To give a feel for B_n, selected values of B_n, computed by algorithm B, are tabulated in table 4.1. It appears that B_n/n is about 2.29 as B_n approaches infinity, which would mean that the optimal algorithm (and hence any algorithm using a uniform or variable encoding) has about a 15% overhead asymptotically. But what can be *proven* about the asymptotic form of B_n?

Consider the recurrence of equation 4.12 without the ceiling function:

$$B'_n = \lg \sum_{k=0}^{n-1} 2^{B'_k + B'_{n-1-k}} \tag{4.13}$$

Removing the ceiling function amounts to allowing us to express the relative pointers with a fractional number of bits. If we define $C_n = 2^{B'_n}$, equation 4.13 becomes:

$$\begin{aligned} \lg C_n &= \lg \sum_{k=0}^{n-1} C_k \cdot C_{n-1-k} \\ C_n &= \sum_{k=0}^{n-1} C_k \cdot C_{n-1-k} \end{aligned} \tag{4.14}$$

Now observe that equation 4.14 is a defining recurrence for the Catalan numbers encountered earlier. So C_n are the Catalan numbers. If we *could* somehow encode the relative pointers using fractional bits with no waste (some kind of arithmetic coding comes to mind) , our encoding would be absolutely perfect. I don't know how to do this, and it is an interesting open problem in this area.

The analysis of equation 4.12 is made very difficult by the presence of the ceiling function. However, the monotonicity of the value of B_n with respect to values of $B_k, k < n$, together with the fact that $\lceil x \rceil < x + 1$ allows us to write the following inequality:

$$B_n < 1 + \lg \sum_{k=0}^{n-1} 2^{B_k + B_{n-1-k}} \tag{4.15}$$

which allows us to show the following:

Theorem 4.2 *The value of $B_n < 3n$, for positive n.*

Proof Let B'_n be the function that makes the inequality 4.15 an equality. Clearly $B_n < B'_n$. Let C'_n be $2^{B'_n}$. Then

$$C'_n = 2\sum_{k=0}^{n-1} C'_k \cdot C'_{n-1-k}$$

Now the claim that $C'_n = 2^n C_n$, where C_n are the Catalan numbers, is easily verified by induction, using this equation and the recurrence of equation 4.14. So $B'_n = n + \lg C_n$, and it was argued earlier by Stirling's approximation that $\lg C_n = 2n - o(n)$. So $B'_n = n + 2n - o(n)$, and $B_n < 3n$, completing the proof. □

This theorem, giving a bound of three bits per node, seems rather weak, observing the values in table 4.1. The rest of this section will examine techniques for finding bounds on the asymptotic limit of B_n/n. First, observe that is is possible, a priori, that B_n/n does not approach a limit as $n \to \infty$. We know that in the limit, $2 < B_n/n < 3$, but perhaps B_n is like the function $f(n) = 2^{\lceil \lg n \rceil}$, which takes on values between n and $2n$, but has no limit $\lim_{n\to\infty} f(n)/n$. Let's call $\inf\{k \mid kn > B_n \text{ a.e.}\}$ the *upper limit* of B_n/n, and $\sup\{k \mid kn < B_n \text{ a.e.}\}$ the *lower limit* of B_n/n. Then B_n/n has a true limit if the lower limit equals the upper limit. This equality is indeed the case, and will be proved shortly. But first, a word about the lower limit.

The monotonicity of B_n gives an easy technique to estimate the lower limit of B_n/n. For any value of n we can exhibit an n-node tree T requiring B_n bits (this is trivial; they *all* require B_n bits). We can take k copies of T and build a tree T', with a right-leaning spine of k nodes, where each node of the spine has a copy of T as its left child, as shown in figure 4.6.

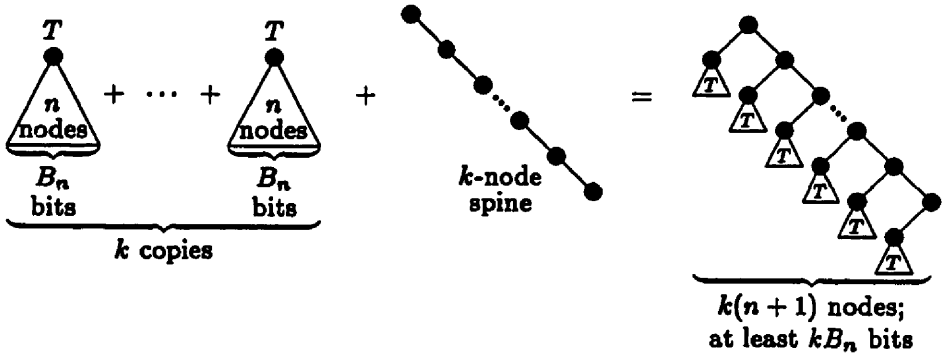


Figure 4.6: Construction bounding the lower limit of B_n/n

The tree T' has $k(n+1)$ nodes, and since it contains k copies of T, it must occupy more than kB_n bits. Since our choice of k was arbitrary, there exists an infinite set of trees (one for each k) where the number of bits per node is greater than $B_n/(n+1)$. So for every n, the value of $B_n/(n+1)$ is a lower bound on the lower limit of B_n/n. (This intuitive proof directly translates into a symbolic one, using equation 4.10 and doing induction on both n and k.) From the values in table 4.1 we now know that the lower limit is at least 2.2935.

The fact that $B_n/(n+1)$ is always less than the lower limit of B_n/n implies that the lower limit is equal to the upper limit, as we now demonstrate:

Theorem 4.3 $\lim_{n\to\infty} B_n/n$ *exists.*

Proof: Suppose that the upper limit u and the lower limit l of B_n/n were unequal. Choose some u' such that $l < u' < u$. Now $u = \inf\{k \mid kn > B_n \text{ a.e.}\}$, so for our $u' < u$, there exists an infinite sequence $\{n_1, n_2, \ldots\}$ where $u'n_i < B_{n_i}$. We also know that for any n, $B_n/(n+1) < l$. Combining these two inequalities, we have $u'n_i < l(n_i+1)$ for the infinite sequence of n_i's. Thus $u'/l < (n_i+1)/n_i$. As i increases without bound, the ratio on the right goes to 1; therefore $u'/l \leq 1$. But we chose u' to be greater than l, hence contradiction. Our assumption that $u > l$ must be false, proving the theorem. □

It seems to be much more difficult to get upper bounds on the limit of B_n/n. The bound of 3 shown in theorem 4.2 can be improved using a variation of the technique used in the proof given earlier. Consider a sequence B'_n defined as follows:

$$B'_n = \begin{cases} B_n & n \leq N \\ 1 + \lg \sum_{k=0}^{n-1} 2^{B'_k + B'_{n-1-k}} & n > N \end{cases} \tag{4.16}$$

Here N is a parameter of the sequence. The idea is that the new sequence B'_n is defined to be identical to B_n on all values up to B'_N, and obeys the defining recurrence for B_n for the values beyond N, with the ceiling operator replaced by increment. We know that $B'_n \geq B_n$ from the monotonicity of B_n. Note that the sequence B' in theorem 4.2 is a special case of the sequence B' here, with $N = 0$. Since we have discarded the troublesome ceiling function, the sequence B' should prove more tractable. We can compute the asymptotic value of B'_n/n for increasing values of N; this will provide an upper bound on the limit of B_n/n.

To simplify our analysis, let's define $C'_n = 2^{B'_n}$. So

$$C'_n = \begin{cases} 2^{B_n} & n \leq N \\ 2 \cdot \sum_{k=0}^{n-1} C'_k C'_{n-1-k} & n > N \end{cases} \tag{4.17}$$

Now consider the generating function $c(x)$ for the C'_n:

$$c(x) = \sum_{n \geq 0} C'_n x^n \tag{4.18}$$

squaring and multiplying by x, we find

$$\begin{aligned} xc^2(x) &= \sum_{n>0} \left[\sum_{k=0}^{n-1} C'_k C'_{n-1-k} \right] x^n \\ &= \sum_{n=1}^{N} \left[\sum_{k=0}^{n-1} 2^{B_k + B_{n-1-k}} \right] x^n \\ &\quad + \sum_{n>N} \left[\sum_{k=0}^{n-1} C'_k C'_{n-1-k} \right] x^n \end{aligned}$$

from equation 4.17, the second part of this sum can be written:

$$\begin{aligned} \sum_{n>N} \left[\sum_{k=0}^{n-1} C'_k C'_{n-1-k} \right] x^n &= \frac{1}{2} \sum_{n>N} C'_n x^n \\ &= \frac{1}{2} \left[c(x) - \sum_{n=0}^{N} C'_n x^n \right] \\ &= \frac{1}{2} \left[c(x) - \sum_{n=0}^{N} 2^{B_n} x^n \right] \end{aligned}$$

therefore

$$2xc^2(x) = c(x) + \sum_{n=0}^{N} \left[2 \left(\sum_{k=0}^{n-1} 2^{B_k + B_{n-1-k}} \right) - 2^{B_n} \right] x^n \tag{4.19}$$

Equation 4.19 is just a quadratic equation in $c(x)$ and can be solved using the quadratic formula (remembering that $c(0) = C'_0 = 1$):

$$c(x) = \frac{1}{4x} \left[1 - \sqrt{8xP(x) + 1} \right] \tag{4.20}$$

where $P(x)$ is the Nth degree polynomial defined by:

$$P(x) = \sum_{n=0}^{N} \left[2 \left(\sum_{k=0}^{n-1} 2^{B_k + B_{n-1-k}} \right) - 2^{B_n} \right] x^n \tag{4.21}$$

The coefficients of $P(x)$ are simply computed from known values of B_n. Observe that $P(x)$ is the series $p(x)$ truncated to $N+1$ terms:

$$\begin{aligned} p(x) &= \sum_{n\geq 0}\left[2\left(\sum_{k\geq 0} 2^{B_k+B_{n-1-k}}\right) - 2^{B_n}\right]x^n \\ &= -1 + x + 2x^2 + 2x^3 + 8x^4 + 40x^5 + 192x^6 + \cdots \end{aligned} \tag{4.22}$$

The key to finding the asymptotic behavior of B'_n lies in the analysis of equation 4.20. Calculating the exact expansion of $c(x)$ seems laborious, since it involves finding the expansion for the square root of ugly polynomials. However, we are only interested in bounding the values of C'_n asymptotically. Let us first observe that $C'_n = -D_{n+1}/4$ for $n > 0$, where D_n is the expansion of the simpler

$$\sqrt{8xP(x)+1} = \sum_{n\geq 0} D_n x^n$$

So it suffices to find asymptotic bounds on the magnitude of D_n. Imagine factoring the polynomial $8xP(x)+1$ into a product of the form:

$$(1 - x/r_1)(1 - x/r_2)\cdots(1 - x/r_{N+1})$$

where the r_k are the complex roots of the polynomial. Then

$$\sqrt{8xP(x)+1} = (1 - x/r_1)^{\frac{1}{2}}(1 - x/r_2)^{\frac{1}{2}}\cdots(1 - x/r_{N+1})^{\frac{1}{2}}$$

Now the expansion of $(1 - x/r)^{1/2}$ is, from the binomial theorem:

$$\sum_{n\geq 0}\binom{\frac{1}{2}}{n}\left(-\frac{1}{r}\right)^n x^n$$

The radius of convergence of this series is simply r. Since $\sqrt{8xP(x)+1}$ is a product of factors of the form $(1 - x/r)^{1/2}$, the region of convergence of its expansion will include the intersection of the regions of convergence of the factors (see Kemp[28, page 92, theorem 4.8]. Therefore the series $\sum_{n\geq 0} D_n x^n$ converges for any value of x where $|x|$ is less than the smallest root of $8xP(x)+1$. For the series to converge, it must also be true that $\lim_{n\to\infty} D_n x^n = 0$. So for any y less than the smallest root of $8xP(x)+1$, $|D_n| = o((1/y)^n)$. Choosing r as the smallest root, we see that $\lg|D_n| = (-\lg r)\cdot n + o(n)$. This implies that $B'_n = \lg|C'_n| = (-\lg r)\cdot n + o(n)$, also.

Since $B_n \leq B'_n$, the value of $-\lg r$ is an upper bound on the limit of B_n/n. This suggests the following procedure for finding such bounds:

N	$\Delta(8xP(x)+1)$	smallest root	bound on B_n/n
0	$1-8x$	.12500	3.0000
1	$+8x^2$	.14645	2.7716
2	$+16x^3$	.15772	2.6645
3	$+16x^4$	.16021	2.6420
4	$+64x^5$	.16204	2.6256
5	$+320x^6$	.16374	2.6105
10		.16916	2.5636
25		.17637	2.5033
50		.17806	2.4896
100		.17830	2.4876

Table 4.2: Upper bounds on the limit of B_n/n

for $N \leftarrow 0 \ldots \infty$
let r be the smallest root of
$$8\sum_{n=0}^{N}\left[2\left(\sum_{k=0}^{n-1}2^{B_k+B_{n-1-k}}\right)-2^{B_n}\right]x^{n+1}+1=0$$
then the limit of $B_n/n < -\lg r$

Algorithm C

Using a symbolic algebra package, I computed the polynomial $8xP(x)+1$ for a number of values of N, and found its smallest root. The result of these calculations[1] are tabulated in table 4.2. I have a strong reason to suspect that the values of the bounds given in the table are converging to about 2.487, since for large values of N there is a root which is just slightly larger than the smallest one. Since the polynomial for $N+1$ is strictly greater than the one for N at all positive x, the limit value of the smallest root (it must converge, since it is monotonically increasing and bounded from above) lies between these two roots. It is not clear that the bound derived in this way converges to the limit of B_n/n (this seems unlikely, in view of the values in table 4.1). We have only proved that the optimal encoding asymptotically requires less than 2.5 bits per node.

The optimal algorithm is interesting from a theoretical standpoint, because it provides a lower bound on the storage needed by any representation of this class. It is also amusing to note that the difficulty of making the optimal algorithm practical lies in the difficulty of the analysis of its

[1] I would have liked to go further, but the polynomials were getting too big and ill-conditioned for the symbolic algebra package to handle.

performance. Often in computer science, a simple algorithm will have a very complex analysis, but that complexity has no direct impact on the implementor. Here, though, the complexity of deriving an easily-evaluated closed form for B_n makes the implementation of the optimal algorithm impractical. The algorithm is only practical if it is easy to analyze!

4.4 Practical concerns

In the first part of this chapter, a basic method for storing trees in linear space was presented. Now, variations on that method of a practical nature will be discussed.

4.4.1 Other operations on trees

The family of representations developed in this chapter was designed only to make the common tree-traversal operations `car`, `cdr`, and `null` efficient. As a bonus, certain other operations dealing with the size of subtrees and tree numberings can also be implemented efficiently. Obviously, obtaining the number of nodes in a given subtree is free, since we are storing it anyway. We can also keep track of the preorder, postorder, or inorder numbers of the nodes as we descend, so it is also cheap to provide operations to query for them. The following relations make it clear how to maintain these numbers as we descend in the tree, using the `size(T)` function counting the number of nodes in the subtree rooted at `T`:

```
preorder(car(T)) = preorder(T) + 1
preorder(cdr(T)) = preorder(T) + size(car(T)) + 1

inorder(car(T)) = inorder(T) - size(cdr(car(T)) - 1
inorder(cdr(T)) = inorder(T) + size(car(cdr(T)) + 1

postorder(car(T)) = postorder(T) - size(cdr(T)) - 1
postorder(cdr(T)) = postorder(T) - 1
```

Furthermore, we can with reasonable efficiency access nodes by their preorder, inorder, or postorder numbers, using the above relations. To access a node at depth d by number, it takes $O(d \log n)$ time (bit accesses) in a tree of n nodes. Since there are only $O(n)$ bits in the entire data structure, the time required is also bounded by $O(n)$, which is better when the tree is very unbalanced and we are accessing a node at depth $d > n/\log n$.

Note that while moving from one node to a child may take $\Theta(\log n)$ time, the total time to traverse the entire tree is only $O(n)$, since we need to scan each bit in the data only once. More generally, the time required to fully explore a subtree of m nodes with a root at depth d is $O(m + d \log n)$, when the whole tree has n-nodes. Another useful observation is that we can fully explore an inorder subrange of tree from the node with inorder number m_1 to the node with number m_2 (at depths d_1 and d_2) in time $O((m_2 - m_1) + (d_1 + d_2) \log n)$. This bound is derived from the number of bits we need to look at to scan all nodes in the range.

Another operation that can be efficiently performed using this scheme is testing whether two pointers (meaning indices into the string of bits) refer to the same subtree. This operation was taken for granted here, and it simply involves comparing the two indices for equality. But the fact that we can easily keep track of preorder and postorder numbers means that we can also test if one node is an ancestor of another. Node a is an ancestor of node b if and only if it has a smaller preorder number and a larger postorder number than b, or equivalently (see Aho[2, page 82]), when

```
postorder(a) - size(a) < postorder(b) < postorder(a)
```

If we resolve to always store the lexicographically smaller subtree first (rather than the storing the one with fewer nodes first, and breaking ties at random), then we can also check two subtrees for structural isomorphism in time proportional to the number of nodes they contain by checking their bit strings for equality. If we wish to check two trees for similarity (isomorphism where we disregard the distinction between left and right children), we check their bit strings for equality, ignoring the `Left-child-first?` bits.

4.4.2 An improvement that obviates dynamic counts

Suppose we don't really need to perform any of these operations that make use of the count of the nodes in the current subtree. Can we avoid keeping track of this count while traversing the tree? In our current scheme, there is a need to keep this value on hand during tree traversal: without it we cannot tell when we have reached a leaf, and we would go "off the trolley," marching down into another part of the bit string. Happily, it is possible to modify our scheme to avoid maintaining this dynamic node count while still keeping the basic operations efficient.

We now wish to represent a position in the tree as a simple index into the string of bits. As before, the position will be the head of a recursively defined block of bits. Conceptually, the simplest modification to the old scheme we could adopt would be to add a bit to the header telling if the

subtree stored second (the larger subtree) is empty. (We can always tell if the subtree stored first is empty by decoding the node count in the header.) But if the larger subtree is empty, then the smaller subtree is empty too, and the current tree is a leaf node, so this new bit is only set at the leaves. If our encoding of the integers R has some unused codes we can, instead of reserving a new bit in *all* the headers, choose the shortest unused code (this is **100** in the scheme examined in section 4.2.1) and assign it to encode the leaf nodes. If we run across this special code when decoding the header, we know we are at a leaf, so this change is sufficient to allow us to use the four bit per node scheme. Except that now, the scheme requires more than four bits per node. By an analysis too similar to that done in section 4.2.1 to merit inclusion, we find that with the special leaf encoding, $B_n = \lceil 9n/2 \rceil - 2l(n) - 2$, which is exactly $\lceil n/2 \rceil$ bits more than we used to need. The asymptotic constant of proportionality has risen to 4.5 bits per node.

As an alternative to modifying the four bit per node scheme by adding a special leaf code, there is a perhaps more direct (and more efficient) way to avoid the need for dynamic node counts. We can cross-breed the *marked preorder sequential* scheme (mentioned in the introduction) with our scheme. We will keep our recursive preorder layout; for each node, we will have the first two bits in the header be the `Left-child-empty?` and `Right-child-empty?` flags telling which of the children are missing. The rest of the header will *only* be present when these first two bits are **00**, indicating that both children are present. The rest of the header consists, as before, of an encoding of the number of nodes in the smaller of the two subtrees, plus the `Left-child-first?` bit. The encodings of the subtrees follow the header, with the smaller subtree first. A leaf node is encoded very simply as **11**. All this is depicted in figure 4.7

Notice that we never need to define $R(0)$ here, since the second part of the header is only present when both subtrees are non-empty. Our analysis is made very simple by the following choice of R:

$$\begin{aligned} R(1) &= \mathbf{1} \\ R(n>1) &= \mathbf{0} \cdot R(\lfloor n/2 \rfloor) \cdot [n \bmod 2] \end{aligned}$$

so the corresponding representation length function r is defined by:

$$\begin{aligned} r(1) &= 1 \\ r(n>1) &= r(\lfloor n/2 \rfloor) + 2 \end{aligned}$$

This representation is simple to understand. A positive integer n is represented by its standard binary representation, prefixed by the string $0^{\lfloor \lg n \rfloor}$. To decode, count the number of leading 0's

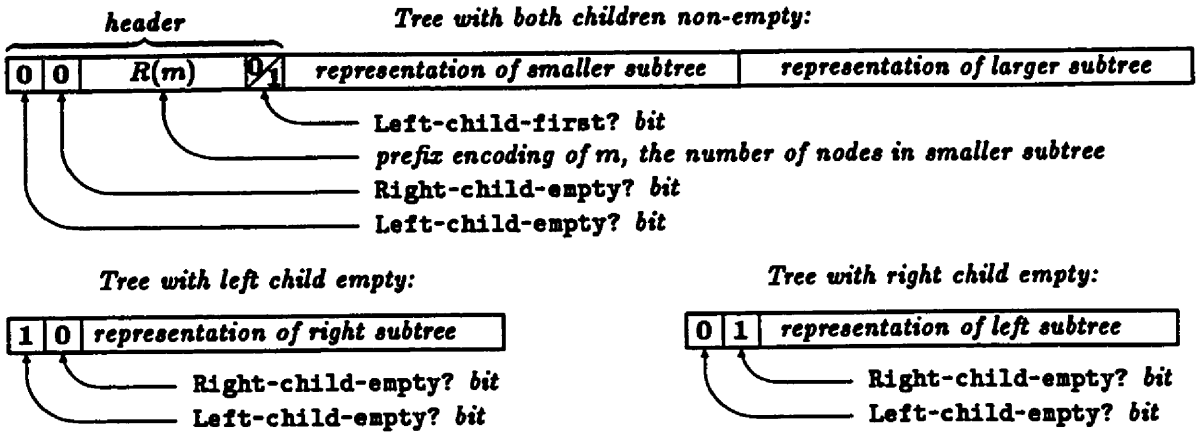


Figure 4.7: Layout of a binary tree using child-empty? bits

until you come to the first 1. After that, read as many more digits as you encountered leading 0's. Note that $r(n) = 2l(n) + 1$. The recurrence for B_n is

$$B_n = \max_{0 \le k \le \lfloor (n-1)/2 \rfloor} [r(k) + 3 + B_k + B_{n-1-k}] \tag{4.23}$$

Note that this recurrence is correct even in the boundary case of $k = 0$, where the second part of the header is omitted, because (even though $R(0)$ is undefined) $r(0) = 2l(0) + 1 = -1$ and $B_0 = 0$, so the quantity in the brackets is $-1 + 3 + 0 + B_{n-1} = B_{n-1} + 2$.

The new $r(n)$ is exactly 2 less than the old one from equation 4.3. This is exactly compensated by the extra 2 in the bracketed quantity in equation 4.23 compared to equation 4.6. Therefore, the same analysis can be recycled, giving $B_n = 4n - 2l(n) - 2$ here as well.

We have recovered the asymptotic half bit per node painlessly. We can use this improved scheme without keeping track of the number of nodes in the current subtree (but we can still keep track of this value if we choose to).

4.4.3 Moving up

So far, we have concerned ourselves only with tree traversal operations that move *downward* in the tree. Sometimes we want to move *upward*, back to our parent. In the representation presented here, as in the standard pointer representation, we may choose to keep a stack of nodes encountered on our downward path, and move upward by simply popping the stack. In the standard representation, it can sometimes be advantageous to store an additional explicit static upward pointer at each node, to avoid the dynamic space overhead of maintaining a stack.

This option is available to us in our linear space representation as well, at a cost of increasing the number of bits stored per node. We modify the scheme presented in the last section that allowed us to traverse the tree downward without maintaining a dynamic node count. Two more bits are added to the first part of the header: `I-am-only-child?` and `I-am-first-child?`. If there is a second part to the header, we replicate this second part and include it between the encodings of the children. The general (neither child empty) case of this complex layout is shown in figure 4.8.

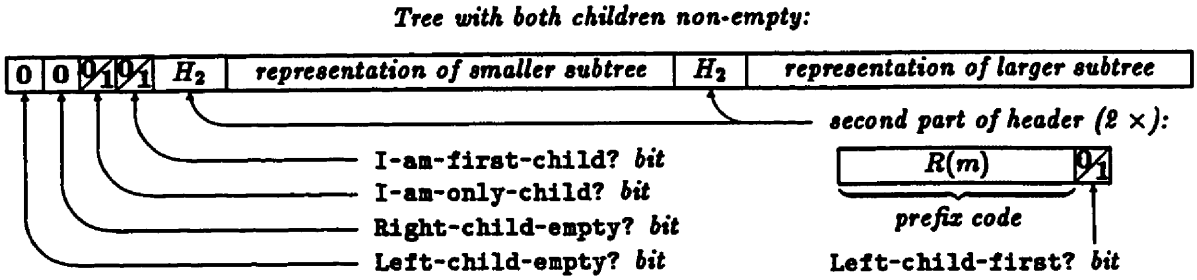


Figure 4.8: Layout of a binary tree to allow upward traversal

The basic idea here is that string of bits immediately preceding our current block now gives us an idea of how many bits backward we need to skip to get to the start of our parent's block.

It is also necessary to use a new encoding function R. Since we will be scanning through the encoded integer both forwards and backwards, we need an encoding that is self-terminating in both directions. Here is one such encoding: $R(1) = 0$; form $R(n > 1)$ by taking the standard $(\lfloor \log n \rfloor + 1)$ digit binary representation of n, removing the most-significant bit (a 1) and shuffling the remainder into $10^{\lfloor \log n \rfloor - 1}1$. To decode (in either direction), read the first bit. If it is a 0 the number encoded is 1. Otherwise, continue reading bits, looking for a 1 in an even position marking the end of the number. The bits from the odd positions (with the leading 1 bit restored) are the standard binary encoding of the number (when taken in order of increasing position in the bit string). This R has the same length function as the previous encoding: $r(n) = 2l(n) + 1$.

Traversing down in the tree is accomplished as before, except that when moving to the larger child, we must skip the bits in the duplicated header as well. To move up, there are three cases, based on the values of the bits `I-am-only-child?` and `I-am-first-child?`:

1x: In this case, our parent has only a one-part header. Skipping back four bits will take us to the beginning of our parent's block.

01: We are the smaller child, stored first. Scan backwards through the whole two-part header of our parent, ending up at the beginning of our parent's block.

00: We are the larger child, stored second. Scan backwards, reading the duplicate copy of the second part of our parent's header. Decode the number of nodes n in our smaller sibling from this, then skip backwards B_n bits. We are now at the beginning of our sibling's block; the **01** case now applies.

The first part of the header is now twice as large; the second part is always present in duplicate. Therefore this representation will take exactly twice as much space as the scheme presented in the previous section: $B_n = 8n - 4l(n) - 4$.

One unfortunate consequence of this upward-capable scheme is that it doesn't allow us to maintain the dynamic count of nodes in the current tree. The problem is that when we move up from a smaller child to its parent, we have to determine how many nodes are in its larger sibling. This quantity is not to be found anywhere; in fact, the absence of this potentially large number is directly necessary for the space-linearity of our scheme. Perhaps there is some clever modification that would allow us to maintain the dynamic node counts while moving upwards, but I haven't found one.

4.4.4 Other types of trees

In this chapter, we have so far concentrated on two equivalent types of unlabeled trees: binary trees and general ordered trees. The classes of labeled trees and of unlabeled unordered trees (both oriented and free) will now be briefly considered.

In labeled trees, we assume that the nodes of the tree are are all distinguishable irrespective of their position in the tree. We can think that each node in an n-node tree bears a distinct integer label from 1 to n. It can be shown that the number of oriented labeled trees with n nodes is n^{n-1}, and it simply follows that the number of unoriented labeled trees is n^{n-2} (see Knuth[29, section 2.3.4.4]). Taking logarithms, the information content of such trees is $\Theta(n \log n)$. A standard pointer representation, using $O(n)$ pointers of $O(\log n)$ bits each, (plus an additional field of $\lg n$ bits storing the label of each node) achieves this informational limit to within a constant factor, so the techniques expounded in this chapter do not help. This $\Theta(n \log n)$ bound applies even when we bound the degree of the nodes in our trees. The conclusion is that for labeled trees, we might as well use the standard pointer representation.

Within the class of unordered trees (those where the children of a node form a multiset, rather than a list) there are two main types: the oriented (or rooted) trees, and the free trees. The only difference between these two is that the oriented trees have a distinguished node called a root, and the edges of the tree are considered to be oriented toward (or away from) the root. The number of oriented trees of n nodes is at most n times as great as the number of free trees of n nodes, since each free tree plus a choice of root equals an oriented tree. The information content of an n-node free tree is therefore at most $\lg n$ bits less than that of an n-node oriented tree, so the asymptotic number of bits per node required for the two are the same. Let us consider all unordered trees to be rooted, since this only engenders the small extra cost of $\lg n$ bits.

The unordered trees of n nodes are no more numerous than the ordered ones, since the order of the children in each unordered tree can be fixed arbitrarily, giving distinct ordered trees. We can certainly store an unordered tree by arbitrarily ordering the children of each node, and then using the techniques described earlier in this chapter. What loss in efficiency do we incur by doing this? The number of unordered trees with n nodes was shown to be $k^n \cdot \Theta(n^{-3/2})$ by Pólya[40], and Knuth[29, section 2.3.4.4, exercise 4] gives the value of $k = 2.95576$. The number of bits stored per node must be asymptotically $\lg 2.95576 = 1.56353$, compared to 2.0 for ordered trees. There is less information here, by a constant factor. Since our methods for storing binary trees in linear space are already suboptimal by a constant factor, the situation is no different when the method is applied to unordered trees (except that the constant is worse).

Rather than use an arbitrary ordering of the children, we can choose some canonical ordering, and take advantage of the constraints implicit in the chosen ordering. This can yield a smaller constant factor. For example, if we are dealing with unordered binary trees (unordered trees of maximum degree 2), we do not need to store the `Left-child-first?` bits, and we do not need two bits for `Left-child-empty` and `Right-child-empty`. We can represent leaves by a single bit, and have one additional bit in the non-leaves to tell if the tree has both children non-empty. An analysis of this scheme reveals an asymptotic performance of $3\frac{1}{3}$ bits per node, using an encoding of the integers with $r = 2l(n) + 1$. It is very likely that a little more research into this area will yield more efficient schemes than this.

When representing unordered general trees (as binary trees) we should arrange the children of each node in lexicographic order. As we proceed from a node to its `Next-sibling` (its `cdr`), we know that the number of nodes in the `First-children` (the `cars`) will be non-decreasing. While traversing downward, we can remember both the number of nodes n in the current subtree, and

a lower bound on the number of nodes m in the `First-child` of that subtree. Just as was done in the optimal variable encoding, we can choose a different encoding function R_{nm} for each n and m (mapping $[m \ldots n-1]$ into bit strings), and find a minimal value of B_{nm}. (Observe that for such trees, the `First-child` will always contain fewer nodes than the `Next-sibling`, unless the `Next-sibling` is empty.) The complete analysis of this construction is an interesting open problem.

4.5 Conclusions

When are the schemes presented in this chapter really useful? There are a few conditions to be met:

- The trees are static.
- The amount of extra information stored at a node is small compared to the size of a standard pointer.
- Economy of space is more critical than speed.

The first condition is obvious. Updating the structures described in this chapter would require a great deal of bit copying. If the amount of extra information to be stored per node is much larger than the size of a pointer, then the savings realized by our scheme will only be a fraction of the total space needed, and the extra complexity and slowness are probably not worth it. Finally, it is clear that our scheme will be substantially slower than a standard pointer implementation. Computers are fast at doing indirect addressing with word-size pointers, and slow at extracting information from memory a bit at a time. This is more a reality of current hardware than a necessity, since a fast hardware decoder of variable length integers could be devised. Ultimately, the bottleneck will be the fixed bandwidth of the channel between the processor and the memory.

A few more words should be said about the encoding of the extra information in the nodes. If each node has a constant number of bits of extra information, we have two choices:

1. We can store the per-node information directly in the bit stream, right after (or before) the header. If we have k extra bits, we simply adjust the function B_n to be $B_n + kn$.
2. We can store the per-node information in a separate array, indexed by one of the tree numberings.

Which of these methods we choose is mostly a matter of style.

4.5.1 Open Problems

The research described in this chapter is really just a beginning; many interesting problems remain open. Foremost among these is the question of whether or not it is possible to achieve the information-theoretic limit of two bits per node, while maintaining the efficiency of traversal operations. An affirmative answer to this question is shown in the next chapter, using a radically different scheme.

Perhaps the method of arithmetic coding can be used to recover the fractional bits lost in the optimal variable encoding. It is difficult to see how to make this idea work, since we would have to "jump" into fractional bit positions.

Another important question is: what is the optimum number of bits per node under uniform encodings? Assume we do *not* keep track of the number of nodes in the current subtree, and only keep an index into the stream of bits. Variable encoding schemes require us to maintain the dynamic node count. Perhaps it is easier to show lower bounds in this weaker model.

Here is another puzzler: Is there a fast way of calculating the optimal B_n? If this could be done in logarithmic time, it would be an important step in making the optimal algorithm practical. Even a somewhat faster (better than the $O(n^2)$ algorithm derived from the recurrence) would be nice, just to allow calculation of more values of B_n. The convoluted form of the defining recurrence suggests that an $O(n \log n)$ algorithm based on the *FFT* may be possible.

If an efficient means of calculating general values B_n eludes us, it may still be possible to calculate values for all *practical* values of n (less than a billion, say) without storing a large table. There may be small circuits to compute B_n. For example, the values of B_n (for small n), can be described as the floor of a piecewise-linear function with a relatively small number of segments. Witness table 4.3, which shows that for $n \leq 100000$, only 46 linear segments are needed to approximate B_n. A table such as this is a useful first step in making the optimal algorithm practical. We would still need a scheme for finding a satisfactory R_n (one that could be quickly decoded) to use such a table.

Finally, it still remains to get a good upper bound on the limit of B_n/n. The upper bound from the sequence of "delayed roundup" generating functions appears to converge to a value above $\lim_{n\to\infty} B_n/n$. What is the reason for this?

Interval	$B_n = \lfloor \ldots \rfloor$	*Interval*	$B_n = \lfloor \ldots \rfloor$
$2 \le n \le 22$	$2.033333 \cdot n - 2.566667$	$13961 \le n \le 15519$	$2.292492 \cdot n - 19.476375$
$23 \le n \le 48$	$2.224747 \cdot n - 6.113636$	$15520 \le n \le 18312$	$2.292516 \cdot n - 19.843489$
$49 \le n \le 94$	$2.254032 \cdot n - 7.322581$	$18313 \le n \le 19946$	$2.292532 \cdot n - 20.135994$
$95 \le n \le 152$	$2.273864 \cdot n - 8.971591$	$19947 \le n \le 21515$	$2.292541 \cdot n - 20.319023$
$153 \le n \le 244$	$2.282202 \cdot n - 10.164103$	$21516 \le n \le 24065$	$2.292548 \cdot n - 20.472911$
$245 \le n \le 551$	$2.286002 \cdot n - 10.999136$	$24066 \le n \le 26027$	$2.292555 \cdot n - 20.629614$
$552 \le n \le 717$	$2.288976 \cdot n - 12.503472$	$26028 \le n \le 28570$	$2.292567 \cdot n - 20.921727$
$718 \le n \le 855$	$2.289597 \cdot n - 12.917241$	$28571 \le n \le 33919$	$2.292577 \cdot n - 21.212190$
$856 \le n \le 1213$	$2.290382 \cdot n - 13.550759$	$33920 \le n \le 37412$	$2.292584 \cdot n - 21.465772$
$1214 \le n \le 1481$	$2.290957 \cdot n - 14.212939$	$37413 \le n \le 40570$	$2.292590 \cdot n - 21.665376$
$1482 \le n \le 1735$	$2.291283 \cdot n - 14.676910$	$40571 \le n \le 43082$	$2.292594 \cdot n - 21.817455$
$1736 \le n \le 2911$	$2.291647 \cdot n - 15.297429$	$43083 \le n \le 45987$	$2.292598 \cdot n - 22.008348$
$2912 \le n \le 3233$	$2.291779 \cdot n - 15.640428$	$45988 \le n \le 48393$	$2.292601 \cdot n - 22.141789$
$3234 \le n \le 3733$	$2.291980 \cdot n - 16.260426$	$48394 \le n \le 55628$	$2.292605 \cdot n - 22.313719$
$3734 \le n \le 4137$	$2.292049 \cdot n - 16.508203$	$55629 \le n \le 58902$	$2.292608 \cdot n - 22.489890$
$4138 \le n \le 4979$	$2.292144 \cdot n - 16.887125$	$58903 \le n \le 63003$	$2.292611 \cdot n - 22.683609$
$4980 \le n \le 5636$	$2.292227 \cdot n - 17.291437$	$63004 \le n \le 69421$	$2.292614 \cdot n - 22.846136$
$5637 \le n \le 7456$	$2.292299 \cdot n - 17.688497$	$69422 \le n \le 73891$	$2.292617 \cdot n - 23.074457$
$7457 \le n \le 8636$	$2.292368 \cdot n - 18.187833$	$73892 \le n \le 76953$	$2.292619 \cdot n - 23.187472$
$8637 \le n \le 9874$	$2.292401 \cdot n - 18.461952$	$76954 \le n \le 87957$	$2.292621 \cdot n - 23.327815$
$9875 \le n \le 11064$	$2.292434 \cdot n - 18.789478$	$87958 \le n \le 91183$	$2.292621 \cdot n - 23.400569$
$11065 \le n \le 12736$	$2.292457 \cdot n - 19.029091$	$91184 \le n \le 96972$	$2.292624 \cdot n - 23.598943$
$12737 \le n \le 13960$	$2.292481 \cdot n - 19.323283$	$96973 \le n \le 100362$	$2.292625 \cdot n - 23.727124$

Table 4.3: Piecewise-linear B_n, for $2 \le n \le 100362$

Chapter 5

Techniques of abstract optimization

This chapter develops some basic tools for abstract data optimization. Several applications are presented, including schemes for storing trees in optimal space (improving the results from chapter 4), doing random-access Huffman coding, and storing planar graphs in linear space (with searching). We begin by exhibiting a space-efficient data structure to represent ordered sets.

5.1 Ranking and selection

Ordered sets are a most fundamental data type. Given a static subset of $1 \ldots n$, it is trivial to design a data structure that supports membership testing in optimal space; a simple bit-vector will do. If the set is sparse, with m elements chosen from $1 \ldots n$ where $m \ll n$, we desire to store the set in $\lg \binom{n}{m}$ bits, which is roughly $m \lg \frac{n}{m}$. Various hashing techniques allow us to approach this limit.

What if we desire a richer set of set operations? Two very useful operations on a subset S of $1 \ldots n$ are:

`rank(`m`)` Returns the number of elements in S less than or equal to m.

`select(`m`)` Returns the mth smallest element in S.

These are inverses of each other, in the sense that `rank(select(`m`))` $= m$, for $1 \leq m \leq ||S||$, and `select(rank(`m`))` $= m$, for $m \in S$. These operations can, of course, be performed directly when a bit-map implementation is used, but that would be very inefficient. We generally must perform a linear scan through the bits to rank and select, so the worst-case cost of these operations is $O(n)$.

Ranking and selection are basic operations that can be used to implement a variety of useful functions on ordered sets. For example, let $j, k \leq n$, and $m \in S$:

`rangecount(`j`,`k`)` Returns the number of elements in S in the interval $j \ldots k$.
This is `rank(`k`) - rank(`$j-1$`)`.

`next(`j`)` Returns the smallest element in S greater than j.
This is `select(rank(`j`)+1)`.

`prev(`j`)` Returns the largest element in S less than j.
This is `select(rank(`$j-1$`))`.

`skip(`m`,`j`)` Returns the element in S that comes j positions after m in a sorted list.
This is `select(rank(`m`)+`j`)`.

One way to add the operations of ranking and selection to a bit-map implementation of a set data type is to augment the bit-map with an auxiliary structure which we shall call a *directory*. This data structure will help make the additional operations efficient.

The term *directory* is taken from Elias[12], where he examines a similar problem: efficient ranking and selection in multisets (which he calls *inventories*). For multisets, there is a pleasing symmetry between ranking and selection, which Elias exploits. However, his scheme is only efficient in the average case. The number of bit-inspections required for any particular operation may be large, but when averaged over all possible inputs he gets logarithmic performance. This average-case efficiency is not good enough for us, since we plan to use ranking and selection as tools. Once they are incorporated into another algorithm, it will be difficult to describe the distribution of inputs to rank and select in a meaningful way. Still, Elias's construction is the inspiration for the two-level directory structure we develop later in section 5.1.1.

Simply storing all the precomputed values of `rank(`m`)` and `select(`m`)` would produce a kind of directory. Since the range values are $1 \ldots n$, we need about $\lg n$ bits per value stored. So the space for this would be $O(n \log n)$, which is unacceptable. The term *directory* implies that the auxiliary data is not too large compared to the bit-map itself. We know that there is a great deal of "fat" in this representation, since the values don't change much from one to the next.

5.1.1 Ranking directories

To achieve good performance with small directories, we will have to be a little more sophisticated. For now, let's restrict our attention to the problem of creating a directory to make ranking efficient. We will add in extra information to facilitate selection later.

One-level directories

Rather than storing *all* the precomputed values of the rank operation, we will store only a fraction of them. The other values can be reconstructed by interpolation, by counting 1 bits in a small region of the bit-map. If we store some of the n values, with equal spacing k between each stored value, we can compute rank(m) by computing $\lfloor m/k \rfloor$, doing one table lookup, and then scanning through at most k bits of the bit-map adding up 1's to get the desired answer. This requires about $(n/k)\lg n$ bits in the directory (which is organized as an n/k element array of numbers, each of $\lg n$ bits), and does $\lg n + k$ bit accesses in the worst case to compute ranks. The choice of k produces a trade-off of space for time. The bits are chopped up into consecutive blocks of size k, and the information in the directory limits our inspection of the bits from the bit-map to a single block.

Choosing $k = \lg n$ gives a scheme that uses $O(n)$ space (in bits) and takes $O(\log n)$ time (in bit-accesses). The time used is within a constant factor of optimal, as is the space. But we would really like a scheme that uses $1 + o(1)$ times the minimal number of bits; since we are retaining the n bits in the bit-map, we want the space for the directory to be $o(n)$.

If we choose k to grow faster than $\log n$, say $\log^2 n$, we need only $O(n/\log n)$ bits, but the time increases to $O(\log^2 n)$. (We can choose any monotonic unbounded $f(n)$, set $k = f(n)\log n$, and achieve $O(n/f(n))$ space and $O(f(n)\log n)$ time). This gets the space down to where we want it, but now the time grows too quickly.

Two-level directories

The directory schemes proposed above are *one-level* schemes. We know how many bits in positions less than m are 1's (except for those in the same block as m) with a single lookup in the directory. Since the maximum number of 1 bits in a block is k (which is small) the values in the directory still don't change too much from one to the next. This suggests using a multi-level directory to recoup some of the space lost to this redundancy. As long as the number of levels is bounded by a constant, we need only inspect $O(\log n)$ bits of the directory. If the final, smallest block size is

$\lg n$, this will lead to a total time of $O(\log n)$.

Let's consider two-level directories. The first-level directory is simply a one-level directory, with block size j. Each block is treated as a independent subset of $1 \ldots j$, with its own directory (with block size k) forming the second level directory. This is shown in figure 5.1. To find rank(m), we

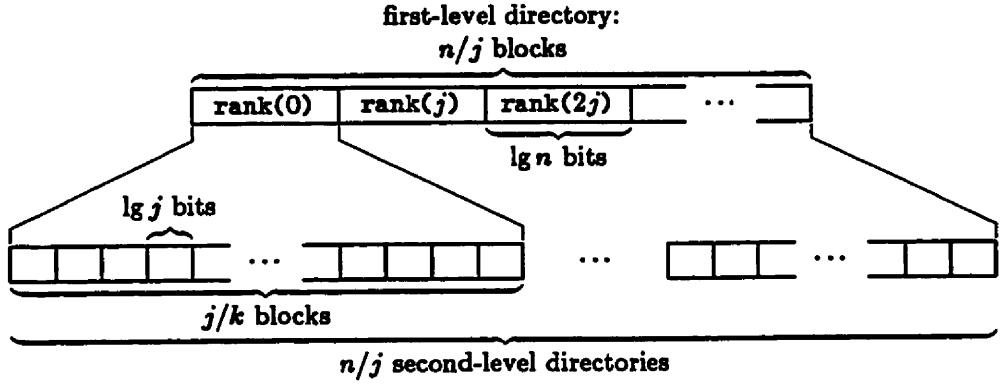


Figure 5.1: A two-level directory for set ranking.

first compute the first-level block number $b_1 = \lfloor m/j \rfloor$. We look up the value of rank($j \cdot b_1$) in the first-level directory, a table of n/j numbers each of $\lg n$ bits. Then we proceed to the appropriate second-level directory. We compute the second-level block number $b_2 = \lfloor (m \bmod j)/k \rfloor$. Then we look at element number b_2 in the second level directory; this will be rank($k \cdot b_2$) in the subrange $(b_1 \cdot j) \ldots (b_1 \cdot j + b_2 \cdot k)$. Adding this value to the value from the first-level directory gives the number of 1 bits in the whole set, except for those in the same second-level block as m. These last few bits (at most k) can be scanned directly in the bit-map and added in to get the total value of rank(m).

The extra space required by this scheme is as follows: $(n/j) \cdot \lg n$ bits for the first-level directory; n/j second-level directories at $(j/k) \cdot \lg j$ bits each for a total of $(n/k) \cdot \lg j$ bits. The number of bits accessed is: $\lg n$ in the first-level directory; $\lg j$ in the second level; and at most k in the bit-map itself. The total time is therefore $O(\log n + k)$. Choosing $k = \lg n$ to make the total time $O(\log n)$, the total space used is $n \cdot [(\lg n)/j + (\lg j)/(\lg n)]$. This space is at a minimum when $j = \lg n \cdot \ln n$. The space needed at this value of j is $2n \ln \ln n / \ln n + O(n \log^{-1} n)$ bits, which is $O(n \log \log n / \log n) = o(n)$. Since the extra space for the directory becomes a vanishing fraction of the space for the bit-map itself, the two-level directory scheme achieves the time and space bounds we seek simultaneously.

Since two levels outdo one, it is tempting to try using the same scheme with more levels to get better results. However, this doesn't lead to improvement over the $1 + O(\log \log n / \log n)$ bit per element ratio that the two-level scheme realizes. Observe that the bulk of the space in any multi-level directory will be found in the bottom level. If the block size at the bottom level is $k(n)$ and the block size at the penultimate level is $j(n)$, the total space used by the bottom level will be $(n/k(n)) \cdot \lg j(n)$. We know that $j(n) > k(n)$, and we require $k(n) = O(\log n)$ to achieve the time bound of $O(\log n)$ bit-accesses. The number of bits in the directories per element must therefore be $\Omega(\log \log n / \log n)$. This is not to say, however, that some fundamentally different scheme could not achieve better performance.

5.1.2 Selection directories

Now that we know how to construct a succinct directory to make ranking efficient, we would like to do the same thing for selection. First, note that a directory that does ranking in time $t(n)$ can be used to do selection (with no additional space) in time $t(n) \cdot \lg n$, by binary search. If we seek the mth element in the set, we start by doing `rank`$(\lfloor n/2 \rfloor)$ and compare the returned value with m to determine in which half of $1 \ldots n$ the value of `select`(m) lies. After $\lg n$ such bisections, we will know the value of `select`(m) exactly.

While this is better than no directory at all, it doesn't get us down to the bound of $O(\log n)$ bit inspections we are after. The ranking directories we built require $O(\log n)$ time per operation, so this binary search technique will only get us down to $O(\log^2 n)$.

Another line of attack is to do what we did when building the ranking directories: keep a table of precomputed values of `select`(m) for m a multiple of some suitably chosen j. Then to find `select`(m), we can look up the value of `select`$(j \cdot \lfloor m/j \rfloor)$, and begin scanning the bit-map, starting at the returned position until $m \bmod j$ more 1 bits are encountered. The problem with this idea is that the number of bits we need to scan through in the bit-map may be very large in the worst case (where the set is sparse).

Putting the ideas together

Neither of the two ideas proposed above is powerful enough to get us the $O(\log n)$ time bound we are after by itself. But if we skillfully combine them, we can make things work.

Assume we have the optimal two-level ranking directory of the previous section available to us. If we knew which second-level block contained `select`(m), we could compute the rank of the first

element in that block, and look through at most $\lg n$ bits in the bit-map to find the value we want. This means that once we locate the second-level block containing `select(`m`)`, we need only look at $O(\log n)$ more bits to get the exact value.

What if we knew which *first-level* block contained `select(`m`)`? We could compute the rank of the first element in that block, subtract from m, and do binary search to find the second-level block containing `select(`m`)`. The binary search would require only $\lg j$ bisections, and each bisection would require us to inspect a number with only $\lg j$ bits, for a total of $\lg^2 j$ bit-inspections. We chose $j = \lg n \cdot \ln n$, so this works out to $O((\log \log n)^2) = O(\log n)$ time. This, together with the result of the previous paragraph, shows that if we could locate the first-level block containing `select(`m`)` in $O(\log n)$ time, we could compute the exact value in $O(\log n)$ time.

We are still left with the problem of finding the correct first-level block. A binary search of the whole first-level directory would be too slow. But we can use a table of precomputed values of `select(`m`)` to find a subarray of the first-level directory to start the binary search. Furthermore, if we know that the values in that subarray are in the range $a \ldots (a + b)$, we need only inspect the $\lg b$ least significant bits of the numbers in the subarray. The other bits can be deduced from the value of a.

In our precomputed table of `select(`m`)` we will store all values where m is a multiple of $j = \lfloor \lg n \cdot \ln n \rfloor$. If we want to find `select(`m`)` for m not a multiple of j, we know that the answer lies between `select(`$j \cdot \lfloor m/j \rfloor$`)` and `select(`$j \cdot \lceil m/j \rceil$`)`, both of which can be obtained via table lookup. Dividing these upper and lower bounds by j, we get a pair of values that bound b_1, the first-level block that contains `select(`m`)`. We can use these indices to define the subarray of the first-level directory on which we start the binary search. We know that in this initial subarray, there are at most $3j$ elements of the set (at most j in the first block, at most j in the last block, and at most j in between).

At this point we run into a small problem. The upper and lower bounds we get out of the table might be quite far apart, if the set is very sparse in this region. Luckily, there is a simple fix for this problem. We will prepare a compressed ranking directory consisting of the values in the first-level ranking directory with duplicates removed. Also, we prepare a two-way index between the compressed and non-compressed ranking directories. The index will store, for each value in the non-compressed ranking directory, the *unique* position in the compressed directory where that value occurs, and for each value in the compressed directory, the *first* position in the non-compressed directory where that value occurs.

With the aid of the compressed ranking directory and the two-way index, we can use the bounds from the table of select values to start the binary search with a small subarray. After finding the upper and lower bounds from this table, we use the non-compressed to compressed index to find a subarray of the compressed ranking directory. Since there are at most $3j$ elements in the subarray, and the subarray is *strictly* increasing, the subarray is at most $3j$ long. We can then perform binary search through this subarray of the compressed directory with only $\lg 3j = O(\log \log n)$ bisections, and use the compressed to non-compressed index to find b_1, the true first-level block number. As we remarked earlier, we do not need to inspect all of the bits of the numbers in the compressed directory to do the binary search either. If we read the first number in its entirety, then we only need to look at the least significant $\lg 3j$ bits of the others, since we know the other values cannot differ from the first value by more than $3j$. Therefore each bisection can be performed using only $O(\log \log n)$ time, and the total time for the binary search is $O((\log \log)^2) = O(\log n)$.

This completes the demonstration that `select` (m) can be carried out in $O(\log n)$ time. But we have been pretty free and easy with the space, adding new structures as needed. How much did we actually use?

Extra space for the selection directory

First we have the table of precomputed values of `select`. There are $n/j = n/(\lg n \cdot \ln n)$ of these at $\lg n$ bits each, for a total of $n/\ln n$ bits here. Then there is the compressed ranking directory, which cannot be bigger than the non-compressed first-level directory, weighing in at $n/\ln n$ bits. Finally, there is the two-way index. Once again, each of these structures is $n/\ln n$ bits. The total additional space used by the selection directory (not counting the ranking directory) is $O(n/\log n)$. Recall that we previously showed that the two-level ranking directory used $O(n \log \log n/\log n)$ bits. Thus the bit-map itself, the ranking directory and the selection directory come to $n \cdot [1 + O(\log \log n/\log n)] = n \cdot [1 + o(1)]$.

A summary of selection

Here is a summary of the steps performed in computing `select(`m`)`. (Remember that j is defined to be $\lfloor \lg n \cdot \ln n \rfloor$.)

1. If j divides m, then we can find `select(`m`)` by table lookup.

2. Otherwise, we do two lookups to obtain a lower bound l of `select`$(j\lfloor m/j \rfloor)$ and an upper bound u of `select`$(j\lceil m/j \rceil)$.

3. The subarray of the (non-compressed) ranking directory we want is from locations $\lfloor l/j \rfloor$ to $\lceil u/j \rceil$ inclusive. We use table lookup to find the appropriate range l' to u' in the compressed ranking directory.

4. We read the value s stored in the compressed ranking directory at location l'. We know that between l' and u', all the values are between s and $s + 3j$.

5. Using s, l' and u', we do a binary search through a subarray of the compressed ranking directory. We examine *only* the least significant $\lg 3j$ bits of each number. This yields an index into the compressed ranking directory of the first-level block holding `select`(m).

6. We map the index into the compressed directory into the true (non-compressed) first-level block number using a table lookup into the two-way index.

7. We do a binary search through the second-level index to find which second-level block contains `select`(m).

8. Finally, we scan through the bits of the proper second-level block in the bit-map until we find the right **1** bit. The address of this bit is the value of `select`(m).

Discussion

The construction used in building the selection directory is indeed ugly. But now that we have this construction, we can forget about its dirty internal details and freely use selection and ranking as tools for data optimization. All the other set operations mentioned in the first section of this chapter are now at our disposal.

There are still several ways in which these rank/select structures could be improved:

- The number of extra bits per universe element goes as $\log\log n / \log n$. While this quantity does vanish as n grows without bound, it does so quite slowly. Even for very large n, it doesn't even half as n squares! We would rather use less extra space. An asymptotic value of something like $n^{1-\epsilon}$ for some positive ϵ would be better.

- The analysis was performed using the data bits model, which counts time in bit-accesses. A careful examination of the construction of the ranking directory shows that we only examine a constant number of consecutive strings of bits, and each string examined is only $O(\log n)$ bits long. Therefore ranking can be done in constant time in the wide-bus model, where aligned fetches of $\lg n$ bits only cost us one unit. The same is not true of the construction of the selection directory. The binary search steps may access a non-constant number of consecutive bit-strings. So if we employ a selection directory, we do not have the desirable property of being constant-cost under the wide-bus model. Is there a way to reorganize the selection directory to achieve constant-cost under the wide-bus model?

Remember that if we improve the performance of the rank/select directory, all the other operations using ranking and selection benefit.

5.2 Random-access Huffman coding

For a simple first application of ranking and selecting, consider the following problem (which has nothing to do with linked data structures): We are given a file of Huffman coded symbols. We would like to prepare a directory to make random access into the unencoded file efficient; that is, given an index m, find the mth symbol in the original file. We would like to use a vanishing proportion of extra space for this directory.

This sounds very much like a selection problem. Let us begin by preparing a selection directory for the set of positions in the (binary) Huffman-coded that begin new symbols in the original file. If the encoded file is of n bits long, we only use $o(n)$ extra bits for the directory. This means that if Huffman-coding achieves some compression factor over a fixed-codeword-length encoding, we can (given long enough files) achieve the same compression factor and enjoy random-access to the symbols.

Of course, there is a big problem with the proposed solution. To do selection, we need to store the original set bit-map as well as the directory. This would double the storage required, which is unsatisfactory. We can get as far as computing the second-level block in which the desired codeword begins without storing the bit-map of start positions. But we cannot scan through the second-level block (the final step in selection) because the blocks are out of synch with the codewords; we do not know how many bits of this block are part of the the last codeword beginning in a previous block.

We do not need to store all $\lg n$ bits of the second-level blocks to recover this synchronizing information. It suffices to store, for each of the $n/\lg n$ blocks, the number of bits in the block that belong to a codeword that begins in a previous block. These numbers are at most $\lg n$, so we only need $\lg\lg n$ bits each, and only $(n/\lg n)\cdot(\lg\lg n) = o(n)$ extra bits for all of them.

With the select directory (minus the set bit-map) and the synchronizing table, we can find the start position of the mth symbol in $O(\log n)$ time, using only $o(n)$ extra bits.

5.3 Trees in optimal linear space

In chapter 4 we discussed a general scheme for storing trees in linear space while allowing efficient traversal. The methods described, while simple and practical, failed to achieve the optimal asymptotic ratio of two bits per node. Now we shall describe a method, employing rank/select directories, that achieves this optimum.

First, let us consider the case of binary trees.

5.3.1 Level-order binary marked

When a binary tree is very balanced, we can implicitly represent the tree as addresses in an array. The root is given the address 1. A node whose address is m has a left child with address $2m$ and a right child with address $2m+1$. The "information" at the nodes of the tree can be stored in the array. This scheme is an efficient choice to represent heaps (see Aho[1, page 87]) since there is no need for explicit pointers, and no wasted space.

Since we are not interested in the information stored at the nodes, but we *are* interested in trees with imperfect balance, we can use the implicit addresses to index an array of bits saying which nodes are present in the tree and which are not. This implicit-bitmap representation of binary trees is shown in figure 5.2. This representation is great for searching (we can do `car` and `cdr` in a single bit access!) but it has an obvious drawback: unless the tree is extremely well balanced, the number of bits needed will be huge. If the deepest node is at depth d, we will need between 2^d and $2^{d+1}-1$ bits. This is simply unacceptable.

The bit-string corresponding to an unbalanced tree will be full of zeroes that indicate missing nodes. But once we know that some node is not in the tree, it is redundant to store zeroes telling us that its children are not in the tree. We can save great deal of the space by leaving out the bits for non-nodes whose parents are also non-nodes. If there is a **0** at location m and a **0** at

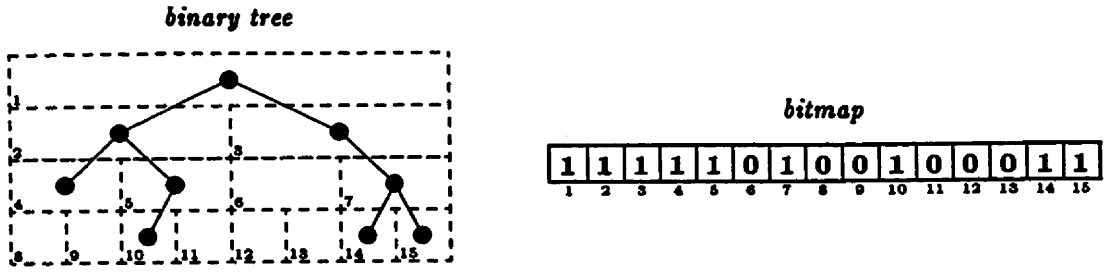


Figure 5.2: A binary tree and its implicit bitmap.

location $\lfloor m/2 \rfloor$, simply cross out the former. This compressed representation is depicted in figure 5.3. Notice that this representation can also be obtained from the original tree as follows:

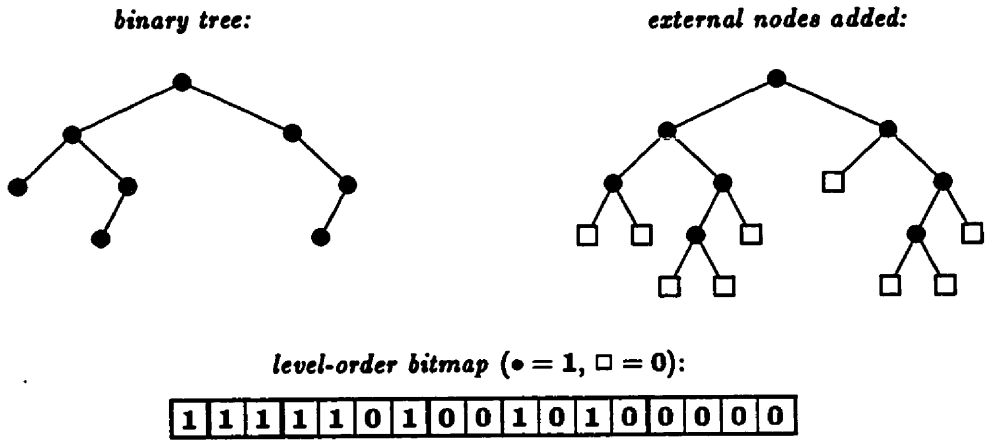


Figure 5.3: Level-order binary marked representation.

1. Mark all the nodes of the tree with 1 bits.
2. Add external nodes to the tree, and mark them all with 0 bits.
3. Read off the bits marking the nodes of the tree in (left-to-right) level-order.

This construction makes it easy to see that the original tree can be reconstructed from the string of bits formed. Each such bit string is therefore associated with a unique tree.

How many bits are there in these level-order mark bit strings? There are n **1** bits (the internal nodes) and $n+1$ **0**'s, for a total of $2n+1$ bits. This representation is therefore asymptotically optimal. But in the compression process, we lost the ability to navigate in the tree by simple index arithmetic. How can we regain the ability to efficiently traverse the tree?

Ranking to the rescue

Suppose we were to represent (internal) node m by the index of where its **1** bit appeared in the level-order mark bit string. Now, consider the bit string as the bit-map of the set of indices of the (internal) nodes. We can build a ranking directory for this set. Each **1** bit on level d corresponds to a node with two children (some of which may be external nodes) on level $d+1$, and these two children will correspond to two adjacent bits in the part of the string where the level $d+1$ nodes appear. Also, left-to-right ordering is maintained from one level to the next: If two nodes, a and b, are on the same level, and a's **1** bit is to the left of b's, then the adjacent pair of bits corresponding to the children of a will occur before b's pair in the string.

This leads to a very simple algorithm to compute `car`(m) and `cdr`(m), for (internal) node m.

$$\begin{array}{|l} \texttt{car}(m) \leftarrow 2 \cdot \texttt{rank}(m) \\ \texttt{cdr}(m) \leftarrow 2 \cdot \texttt{rank}(m) + 1 \end{array}$$

Algorithm LOB

There is a strong similarity to the implicit addressing scheme discussed earlier. The node m is `nil` exactly when the mth bit of the string is a **0**, since this indicates an external node. The root node has index 1.

The string itself occupies $2n+1$ bits, and the ranking directory occupies $o(n)$ bits, so the total space required is $2n+o(n)$. This is asymptotically optimal linear space. The tree-traversal operations do a single `rank`, so they require time $O(\log n)$ time under the data-bits model (but only constant time under the wide-bus model). This improves the results of chapter 4.

If we were also to keep a selection directory, we could find `parent`(m) efficiently too. This is because

$$\texttt{parent}(m) = \texttt{select}(\lfloor m/2 \rfloor)$$

Now, we turn our attention to general rooted trees with ordered children. We will use both ranking and selection, together with another $2n$ bit string scheme (again based on level-order) to represent such trees.

5.3.2 Level-order unary degree sequence

A rooted, ordered tree can be represented by giving the branching degree sequence in any of a number of standard orderings of the nodes. (The (branching) degree of a node in a rooted tree is simply the number of children it has.) Suppose we write down the degree sequence of a tree, ordering the nodes in the left-to-right level order employed in the previous section. This sequence of n positive integers uniquely identifies the tree. Now let us encode these positive integers with the simplest possible binary prefix code (the "unary" code):

$$\begin{aligned} R(0) &= \mathbf{0} \\ R(k>0) &= \mathbf{1} \cdot R(k-1) \end{aligned} \tag{5.1}$$

The integer d is represented by the string $\mathbf{1}^d\mathbf{0}$. Let us take the sequence of degrees encoded in this fashion and simply concatenate them together to form a bit string. Since the codes are prefix codes, we still can easily find the unique tree associated with a string.

The number of 1 bits in this string is n. Every node except the root is a child of another node, so the number of 0 bits is $n-1$. The total length of the string is thus $2n-1$ bits. We will say that each node is associated with exactly one 0 bit and (except for the root) one 1 bit. To maintain the "one 1 per node" property, let us add a fake super-root node to the top of the tree, whose only child is the root. Now each node has a unique 1 bit associated with it, and the string is only two bits longer.

This bit-string scheme has much in common with the level-order marked binary scheme described in the previous section. Figure 5.4 depicts a tree and its level-order unary degree sequence.

An almost identical scheme is described by Read[41, pages 173–175]. This "bottom-up valency code," as he terms it, is just one of several correspondences between ordered trees of n nodes and binary strings of length about $2n$ that he gives. But Read is only interested in encoding trees as strings and subsequently decoding them; he does not consider the possibility of performing search operations directly on these bit strings, as we do.

We will represent a node m by the index of its corresponding 1 bit in the string, as in the previous section. Let us build ranking and selection directories for the bit-string *and* its bitwise complement. This will allow us the additional freedom to select the mth element *not* present in the

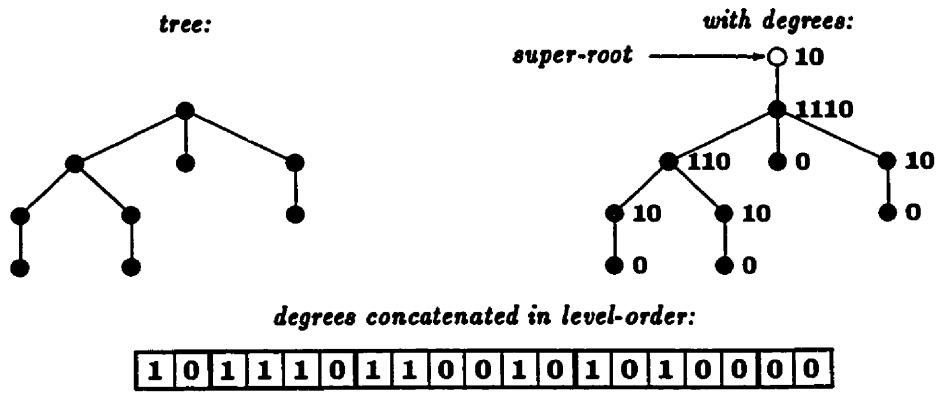


Figure 5.4: Level-order unary degree sequence representation.

set (the mth 0 bit). We will use the notation `rank0` and `select0` to refer to the set-complemented operations.

With this representation, we shall be able to support a rich collection of tree-traversal operations. In particular, we will be able to efficiently move up in the tree and enjoy random access to children.

We can test if m is `nil`, as before, by inspecting the mth bit of the bit-map. The operation `next-sibling(`m`)` is simply an increment of m. In fact, we can find the sibling j after m by incrementing m by j. This also allows us to access previous siblings, and access children by number (provided we can find `first-children` efficiently). Here is the full set of available operations on (non-`nil`) node m:

```
first-child(m) ← select0(rank(m)) + 1
next-sibling(m) ← m + 1
parent(m) ← select(rank0(m))
```

Algorithm LOUDS

To make random access to children truly useful, we need to know how many children a given node m has. This is easily computed as:

`select0(rank(`m`)+1) − first-child(`m`)`

Once again, we have improved the results of chapter 4 by making the linear constant optimal while retaining logarithmic time in bit-accesses. In this case, we have also provided a more flexible

repertoire of tree-traversal operations. However, since we make heavy use of selection directories, we cannot claim constant-time performance in the wide-bus model for this case.

5.4 Planar graphs in linear space

Up until now, we have avoided abstract optimization of graphs. This seems like an obvious generalization of optimization of trees, but there is a fundamental difference.

A pointer-based representation of a tree is very wasteful of information; we have seen that it uses $O(n \log n)$ bits where $O(n)$ suffice. But for general graphs, a pointer-based adjacency-list representation is reasonably efficient (for graphs that are not very dense). For a (labeled) graph of n nodes and m edges, we need $\lg \binom{n}{m}$ bits. Unlabeled graphs require about $\lg(n!)$ fewer bits, but even then the number of bits needed is $\Omega(m \log m)$. For sparse graphs, where $m = O(n)$, the adjacency list representation is within a constant factor of optimal. Itai and Rodeh[25] show that an adjacency list is close to optimal even for graphs that are fairly dense.

The problem is that there are just too many graphs to be able to represent them all succinctly. What if we restrict our attention to a particular class of graphs whose number is simply exponential in the number of nodes? Then the possibility of a linear-space (in bits) representation exists. One important class of graphs with this property is the class of planar graphs. Planar graphs come up frequently in computer science, and it would be useful to have a space-efficient encoding for them.

Turán[46] gives such an encoding. His encoding stores a graph of n nodes in $12n$ bits. It is possible to encode graphs into trees (and vice-versa) with a simple construction in linear time. But there is no easy way to do useful graph-traversal on the encoded form of the graph. Soon we will show how to do this. Let us first briefly summarize Turán's elegant construction.

5.4.1 Turán's construction

We start by embedding the graph G in the plane. This is accomplished in linear time by a number of well-known algorithms. Next we choose an arbitrary rooted spanning tree T of G. Let the planar dual of G be G'. Then the edges in G' that do not cross an edge in T form a spanning tree T' of G'. Observe that each edge in G is either in the spanning tree T or crosses a unique edge in T'. If we could represent T and T' and the relationship between them in linear space, we would have a linear space encoding.

As we saw in chapter 4, trees have a natural representation as balanced strings of parentheses. The tree T can be encoded with one type of parenthesis, and the tree T' with another. These two

balanced strings of parentheses can then be "shuffled" together in such a way that the relationship between T and T' is encoded.

The resulting string, encoded in binary, is the encoding of G. Each edge in G is either in T or crosses one in T', and thus accounts for one pair of parentheses in the final string. Since there are two different types of parenthesis, there are four distinct symbols, so each parenthesis requires two-bits. Finally, since planar graphs have at most $3n - 6$ edges, the number of bits used is:

$$\frac{3n \text{ edges}}{\text{planar graph}} \cdot \frac{2 \text{ parentheses}}{\text{edge}} \cdot \frac{2 \text{ bits}}{\text{parenthesis}} = \frac{12n \text{ bits}}{\text{planar graph}}$$

5.4.2 Searching and testing adjacency

Turán's encoding gives linear space, but it does not allow efficient searching. With a linked representation, we could iterate through the neighbors of a node with only $O(\log n)$ bit inspections per neighbor. We really want an encoding that permits efficient searching. Additionally, we would like to support adjacency testing with similar efficiency.

Recently, Kannan *et al.*[27] show how to implicitly represent planar graphs to allow adjacency testing with $O(\log n)$ bit-inspections. Their method makes use of the bounded arboricity of planar graphs. They decompose a planar graph into (at most) three edge-disjoint spanning trees (using a famous theorem of Nash-Williams), and then represent each tree separately. Although they still need $O(n \log n)$ bits for the whole graph and they cannot search efficiently, the beauty of their data structure lies in its implicitness: the graph is fully described by the set of its node indices.

The decomposition of planar graphs into three trees seems like an attractive idea for us, too, since we already know how to represent trees efficiently. The problem is that we can't afford the space required to cross-reference the node indices from one tree to another. Kannan *et al.*'s scheme applies to the full class of graphs with bounded arboricity, not just to planar graphs. There are too many graphs of arboricity two to represent each one in $O(n)$ bits. Still, the idea of edge-decomposing graphs will prove effective for our problem. We simply need a decomposition that admits a global indexing scheme for the nodes. Later, we will show how the decomposition of a graph into pages fits our needs.

5.4.3 Parentheses balancing

We will show a structure that implements the operations of searching and adjacency testing with the desired $O(\log n)$ bit-accesses-per-operation, in a number of bits proportional to n. Heavy use will be made of ranking and selection. We will also need one other tool: a parenthesis balancer.

The construction of Turán shows how useful trees, and ultimately strings of balanced parentheses, can be for representing planar graphs. We will not make use of his construction, but we will make use of balanced parentheses. Let's first describe and build the tool.

As we mentioned early in chapter 4, a balanced parenthesis string is not a good representation for a tree if we desire to do tree-traversal efficiently, because we might have to scan through a large fraction of the string to find a matching parenthesis (much as the text editor I am using does). If I type an unmatched "}" right now, the editor will pause noticeably before informing me that there is no matching open parenthesis. Making (static) parenthesis balancing efficient is a job for data optimization.

Given a static balanced string of n parentheses, we wish to build, in space linear in n, a directory that will make the following operation efficient:

> Find the position in the string of the close (open) parenthesis that matches the open (close) parenthesis in position m.

Obviously it suffices to solve the restricted problem of finding close parentheses that match open ones, because we can build a backwards directory to find the open parentheses that match close ones.

As a primitive first cut, we could simply try to store, for each index m, a pointer to the matching parenthesis. This obviously fails, since each pointer stored needs to be $\lg n$ bits in length.

The next refinement is to break the string into blocks of size $\lg n$. We can afford to spend $\lg n$ time checking if a parenthesis has a match within its own block, so we only need to store pointers for parentheses whose matches lie outside their blocks (let's call these the *far* parentheses). The block in which the parenthesis that matches a far parenthesis will be called the *matching block* of that far parenthesis.

We will record the *nesting depth* at the beginning of each block with only $\lg n \cdot n/\lg n = n$ bits. Now we can store the pointers for far parentheses as the block numbers of the matching blocks instead of exact indices. To find the match for a far parenthesis, we look up the depth at the beginning of its own block and then scan through that block to find the depth of the far parenthesis. We next look up the matching block, look up the matching block's initial nesting depth, and scan through the parentheses in the matching block until the depth is the same as that of the far parenthesis we started with. We have then found the matching parenthesis with only $O(\log n)$ work.

This is a start, but we are not yet there. It may be the case that almost all of the parentheses are far; witness this deeply nested string:

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

While there may be many far parentheses, two far parenthesis that are near each other in the string are likely to have matches that are nearby each other. This fact is the key to a succinct representation of the far pointers.

Let us take our table of far pointers (the block numbers of the matching blocks of the far pointers) and compress it by removing consecutive duplicates. How will we find which entry in the pointer table contains the block number of the matching parenthesis for a particular far parenthesis? Easy. We build the set (via a bit-map) of indices of those far parentheses whose matching block is different from that of the previous far parenthesis in the string (let us call such far parentheses *pioneer* parentheses). Also, we construct a ranking directory for that set. Now, to find the appropriate location in the table where the pointer for a far parenthesis with index m, we just compute $\mathbf{rank}(m)$ in the set of pioneers.

Now we have a structure that can balance parentheses in $O(\log n)$ time. A picture of this hairy beast is included as figure 5.5.

The table of nesting depths, the set (bitmap) of pioneer parenthesis, and its ranking directory add up to only $2n + o(n)$ extra bits. But what about the compressed pointer table? If there are more than $O(n/\log n)$ pioneers, this table could grow to be more than linear. Any individual block can contribute $\lg n$ pointers into the table. Let us now prove that the total size of this table is not too large.

Theorem 5.1 *The number of pioneer parentheses in a balanced string divided into b blocks is at most $2b - 3$.*

Proof: Imagine a graph with a node for each block of the string. Lay these nodes out on the plane in a straight line in order of the blocks. Create an edge between two nodes if there is a far parenthesis in one that points into the other. The number of edges in such a graph is at least as great as the number of pioneers in the string, since every pioneer can be mapped to a different edge. But since our string of parentheses is balanced, none of the edges can cross, and the graph is outer-planer (that is, it can be embedded in the plane with all the vertices on a single face). Therefore the number of edges is at most $2b - 3$, by a property of outer-planar graphs. □

Since each pointer in the table is lg n bits wide, this means the entire table is at most $[2(n/\lg n) - 3] \cdot \lg n = 2n + o(n)$ bits.

This completes the description of the linear-space parenthesis balancer. We have built a structure of $5n + o(n)$ bits that balances parentheses strings of length n in $O(\log n)$ time in bit-accesses (and constant time under the wide-bus model, since we only use ranking and not selection). Of course, an obvious open problem is to reduce the constant factor to less than 5 (ideally to 1).

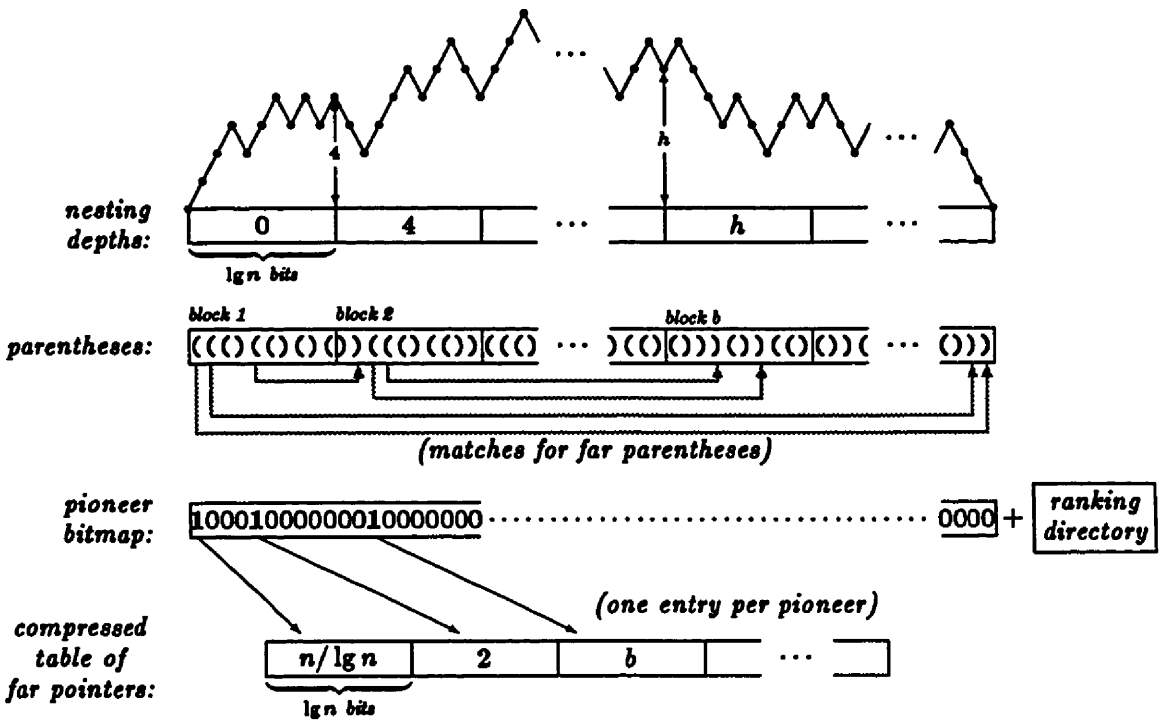


Figure 5.5: A structure to balance parentheses

5.4.4 Bounded pagenumber graphs

Instead of describing how to use the tools we have built to efficiently represent planar graphs, we will actually show how to represent a larger class of graphs, of which planar graphs are a subclass. This larger class is the class of *bounded pagenumber graphs*. These are the graphs that have k-page book embeddings, where k is a parameter of the class. Let us first define the term *book embedding*, following Benrhart[5].

A *k-page book embedding* of a graph $G = \langle V, E \rangle$ is a *printing order* of V (a permutation specifying the ordering of the nodes along the spine of a book), plus a partition of E into k *pages*. The edges on a given page must not intersect, and all pages share the same printing order of the nodes.

The *pagenumber* (or *book-thickness*) of a graph G is the minimum number of pages in any book embedding of G. Let $\mathcal{G}_k$ be a class of graphs all of whose pagenumber is bounded by k. Given a particular graph $G \in \mathcal{G}_k$, and a correct k-page embedding of G, we will show how to visit neighbors and test adjacency in G. For G with n nodes, we will use only $O(\log n)$ bit-inspections per operation, using a representation of G with total number of bits linear in n (for fixed k). The number of bits used will actually be $O(kn)$.

Let us first build up a representation of G. To make things simple, we will start by showing a linear-space representation for one-page graphs (these are exactly the outer-planer graphs) and then generalize.

One-page graphs in linear space

The edges on a given page of the graph all lie to one side of the nodes (which are on the spine) and may not cross. If we lay our "book" so so that the spine is horizontal (as shown in figure 5.6) we observe that the nesting structure of the edges is just that of a balanced string of parentheses.

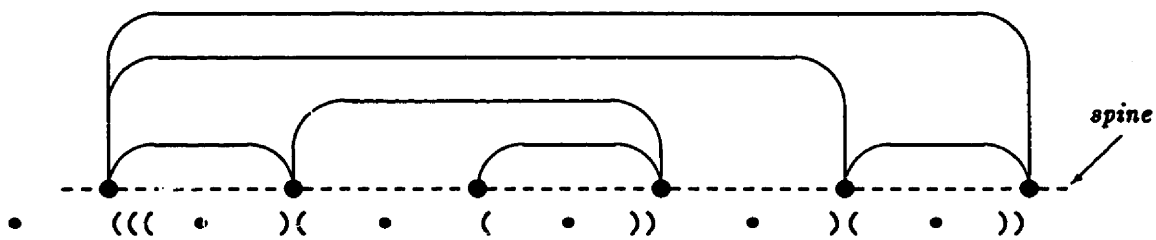


Figure 5.6: One-page graphs as balanced parentheses

We start with a string of n node symbols each, denoted by •. For each edge $\langle u, v \rangle$ on the page, we insert a '(' just before the $(u+1)$st node symbol and a ')' just after the vth node symbol. (Note that the parentheses in the string remain balanced after each such insertion.) The final result is a string over a 3 symbol alphabet, containing n node symbols and at most $2n-3$ each open and close parentheses (from the properties of outer-planer graphs discussed earlier). The parentheses between the mth and the $m+1$st node symbol correspond to the set of edges out of node m.

Let us further encode this 3-symbol string into a pair of strings of bits. First, we will record the sets of positions occupied by node symbols as a bit map. Term this the *node map*. Next, delete the node symbols and record the remaining parenthesis string in binary. These two strings allow reconstruction of the original graph, and use at most $[n+2(2n-3)]+[2(2n-3)] = 9n-12 = O(n)$ bits.

These two bit strings are at the heart of the linear space representation of G. Additionally, we will construct and employ the following optimization tools:

- A matcher for the parentheses string.
- A rank/select directory for the node-map and its complement.

These tools will require extra space, but the total storage used will still be $O(n)$.

We shall use the natural numbering provided by the printing order as our indices for the nodes. When iterating through the edges leaving a given node, we will store an index into the parenthesis string as edge indices.

Searching

With these structures, searching around in G is little more than matching the parentheses. Each edge in G is associated with a pair of matching parentheses. To follow an edge, given the index of one of its associated parentheses, simply find the matching parenthesis. We will also make use of two simple macros: `node-to-edge` converts a node number into an index into the parenthesis string where the edges (parentheses) out of that node start, and `edge-to-node` takes an index into the parenthesis string and finds the number of the node whose block contains that edge (parenthesis):

$$\texttt{node-to-edge}(m) = \texttt{rank0}(\texttt{select}(m)+1)$$
$$\texttt{edge-to-node}(e) = \texttt{rank}(\texttt{select0}(e))$$

Now we can write the algorithm to visit the neighbors of a node given by its index number:

```
e ← node-to-edge(m)
while edge-to-node(e) = m
        e' ← paren-match(e)
        visit edge-to-node(e')
        e ← e + 1
```

Algorithm NEIGHBORS: visit the neighbors of node m

This algorithm performs only a constant number of rank, select, and parenthesis matching operations between the nodes it visits. Therefore it can be used to search with only $O(\log n)$ bit-inspections per edge examined.

Testing adjacency

Another frequently desired operation is adjacency testing: is there an edge between node u and node v? This operation is not efficiently implemented by adjacency-lists, but with our data structure, it is cheap.

First, we need an easy theorem. Say that an edge $\langle u, v\rangle$ out of node v is a *forward* edge if u comes after v in the printing order, and a *backward* edge otherwise. Also, let us define the length of an edge $\langle u, v\rangle$ to be $|u - v|$, where u and v are printing order numbers.

Theorem 5.2 *Let graph $G = \langle V, E\rangle$ be embedded in one page, and let u and v be two nodes from V given by printing order number, with u before v. Then $e = \langle u, v\rangle \in E$ if and only if either e is the longest forward edge out of u* or *e is the longest backward edge out of v.*

Proof: The *if* part is trivial. To show the *only if*, use proof by contradiction. Assume that $e = \langle u, v\rangle \in E$, but that the longest forward edge out of u is not e (so it must be $\langle u, v'\rangle$, for some v' that lies ahead of v), and that the longest backward edge out of v is not e (it must be $\langle u', v\rangle$, for a u' before u). But the edge $\langle u, v'\rangle$ would have to cross the edge $\langle u', v\rangle$, violating our stipulation that G be embedded in one page. □

Each edge $\langle u, v\rangle$ is a forward edge out of one of the nodes it impinges upon, and a backward edge out of the other. The forward edges correspond to the open parentheses in our string, and the backward edges correspond to the close parentheses. The edges out of a given node correspond to a contiguous block of the parenthesis string: first the backward edges appear in order of increasing length, and then the forward edges appear in order of decreasing length. This ordering of the edges out of a node makes it easy to locate the longest forward and backward edges, and theorem 5.2 tells us that we only need to look at these longest edges to determine adjacency in one-page graphs.

To determine where the block of backward edges (close parentheses) ends and the block of forward edges (open parentheses) begins, we will need to keep an additional rank/select directory for the parenthesis string. This will permit us to find the next-open-paren in the string efficiently, using the techniques described in section 5.1, and add only linear extra storage. Here is the

algorithm to check the adjacency of node u with node v, assuming u precedes v in the printing order:

```
e ← next-open-paren(node-to-edge(u))
e' ← paren-match(e)
f ← next-open-paren(node-to-edge(v)) − 1
f' ← paren-match(f)
if edge-to-node(e) = u and edge-to-node(e') = v
  or edge-to-node(f') = u and edge-to-node(f) = v
    return TRUE
  else
    return FALSE
```

Algorithm ADJ: Test if $\langle u, v \rangle$ is an edge.

Like NEIGHBOR, algorithm ADJ performs only a constant number of rank, select and match operations, and so it requires only $O(\log n)$ time.

Graphs of more than one page

In the previous section, we showed how to represent a one-page graph of n nodes in $O(n)$ bits allowing searching and adjacency testing in $O(\log n)$ time. The generalization to multi-page graphs is direct. If graph G is a k-page graph, we simply represent each of its pages (these are one-page graphs) separately. All the pages share the same printing order, so node indices are common to all pages. To visit all the neighbors of a particular node m, go through each of the k pages in turn, executing algorithm NEIGHBOR. To test two nodes for adjacency, simply use ADJ to resolve the question for each page, and take the OR of the results.

So we can represent any $G \in \mathcal{G}_k$ (the class of k-page graphs) in $O(kn)$ bits, with searching and adjacency testing in $O(k \log n)$ time. For any sub-class of graphs with bounded pagenumber, this becomes $O(n)$ space and $O(\log n)$ time. Yannakakis[49] gives a linear-time algorithm that embeds any planar graph in four pages. Since we have shown the linear-space result for any class of graphs with bounded pagenumber, it follows for planar graphs as well.

Room for improvement

The linear factor used by our planar-graph representation leaves much room for improvement. The constructions given were chosen to maximize clarity, rather than minimize this factor. If the reader

was strict in accounting the bits used, she will have counted 64 bits per node in the graph (ignoring the sublinear terms). This is in contrast with Turán's construction, which uses only 12 bits per node (but does not allow efficient searching). Most of the overhead comes from the inefficiency of the parenthesis matcher, which takes 4 extra bits per parenthesis *per direction*. If these extra bits could be eliminated, we would need only 16 bits per node, still 4 short of Turán's bound.

I really don't know if 12 bits per node is asymptotically optimal for planar graphs. I believe that the formula for the (asymptotic) number of nonisomorphic planar graphs on n nodes is not known.

Bibliography

[1] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. (1974). *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA.

[2] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. (1983). *Data Structures and Algorithms.* Addison-Wesley, Reading, MA.

[3] APPEL, A. W., AND JACOBSON, G. J. (1988). "The World's Fastest Scrabble Program." *Communications of the ACM* 31(5):572–585.

[4] APPEL, A. W. (1988). "Simulating Digital Circuits with One Bit Per Wire." *IEEE Transactions on CAD/ICAS* 7(9).

[5] BERNHART, F., AND KAINEN, P. C. (1979). "The book thickness of a graph." *Journal of Combinatorial Theory B* 27:320–331.

[6] CHAZELLE, B. (1985). "Slimming down data structures; a functional approach to algorithm design." *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science*, pages 165–174.

[7] COMER, D., AND SETHI, R. (1977). "The complexity of trie index construction." *Journal of the ACM* 24(3):428–440.

[8] COMER, D. (1981). "Analysis of a heuristic for full trie minimization." *ACM Transactions on Database Systems* 6(3):513–537.

[9] DANTZIG, G. B., AND FULKERSON, D. R. (1954). "Minimizing the number of tankers to meet a fixed schedule." *Naval Research Logistics Quarterly* 1:217–222.

[10] DILWORTH, R. P. (1950). "A decomposition theorem for partially ordered sets." *Annals of Mathematics* 51:161–166.

[11] DINIC, E. A. (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation." *Soviet Math. Dokl.* 11:1277–1280.

[12] ELIAS, P. (1974). "Efficient Storage and Retrieval by Content and Address of Static Files." *Journal of the ACM* 21(2):246–260.

[13] ELIAS, P. (1975). "Universal Codeword Sets and Representations of the Integers." *IEEE Transactions on Information Theory* IT-21(2):194–203.

[14] ELIAS, P., AND FLOWER, R. A. (1975). "The Complexity of Some Simple Retrieval Problems." *Journal of the ACM* 22(3):367–379.

[15] EVEN, S., AND TARJAN, R. E. (1975). "Network flow and testing graph connectivity." *SIAM Journal of Computing* 4:507–518.

[16] FIAT, A., NAOR, J., SCHMIDT, P., AND SIEGEL, A. (1988). "Non-oblivious Hashing." *Proceedings of the 20th ACM Symposium on Theory of Computing*, pages 367–376.

[17] FREDMAN, M. L., KOLMÓS, J., AND SZEMERÉDI, E. (1982). "Storing a Sparse Table with $O(1)$ Worst Case Access Time." *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, pages 165–169.

[18] FURST, M., HOPCROFT, J., AND LUKS, E. (1981). "Polynomial-time algorithms for permutation groups." *Proceedings of the 21st IEEE Symposium on Foundations of Computer Science*, pages 36–41.

[19] GALLAGER, R. G. (1980). *Information Theory and Reliable Communication.* John Wiley & Sons, New York, NY.

[20] GAREY, M. R., AND JOHNSON, D. S. (1979). *Computers and Intractability.* Freeman, San Francisco, CA.

[21] VAN DEN HERIK, H. J., AND HERSCHBERG, I. S. (1986). "A Data Base on Data Bases." *ICCA Journal* 2(1):29–34.

[22] HOPCROFT, J. E., AND KARP, R. M. (1973). "An $n^{5/2}$ algorithm for maximum matching in bipartite graphs." *SIAM Journal of Computing* 2:225–231.

[23] HUFFMAN, D. A. (1952). "A method for the construction of minimum-redundancy codes." *Proceedings of the IRE* 40:1098–1101.

[24] HUFFMAN, D. A. (1954). "The synthesis of sequential switching circuits." *Journal of the Franklyn Institute* 257:3–4, 161–190, 275–303.

[25] ITAI, A., AND RODEH, M. (1980). "Representations of Graphs." *Acta Informatica* 17:215–219.

[26] JERRUM, M. (1982). "A Compact Representation for Permutation Groups." *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, pages 126–133.

[27] KANNAN, S., NAOR, M., AND RUDICH, S. (1988). "Implicit Representations of Graphs." *Proceedings of the 20th ACM Symposium on Theory of Computing*, pages 334–343.

[28] KEMP, R. (1984). *Fundamentals of the Average Case Analysis of Particular Algorithms.* B. G. Teubner, Stuttgart, and John Wiley & Sons, Chichester.

[29] KNUTH, D. E. (1973). *The Art of Computer Programming.* Addison-Wesley, Reading, MA.

[30] KNUTH, D. E., AND GREEN, D. H. (1981). *Mathematics for the Analysis of Algorithms.* Birkhäuser, Boston, MA.

[31] KOHAVI, Z., AND PAZ, A. (editors) (1971). *Theory of Machines and Computations.* Academic Press, New York, NY.

[32] KRAFT, L. K. (1949). "A Device for Quantization, Grouping, and Coding Amplitude Modulated Pulses." M.S. Thesis, Electrical Engineering Department, Massachusetts Institute of Technology, Cambridge, MA.

[33] KUHN, H. W. (1955). "The Hungarian method for the assignment problem." *Naval Research Logistics Quarterly* 2:83–97.

[34] LESK, M. E. (1975). *LEX — a lexical analyzer generator.* CSTR 39, Bell Laboratories, Murray Hill, NJ.

[35] MALY, K. (1976). "Compressed Tries." *Communications of the ACM* 19(7):409–415.

[36] MILLER, G. L., AND REIF, J. H. (1985). *Parallel tree contraction and its application.* Technical report 18-85, Center for Research in Computing Technology, Harvard University, Cambridge, MA.

[37] MINSKY, M., AND PAPERT, S. (1969). *Perceptrons.* MIT Press, Cambridge, MA.

[38] MOORE, E. F. (1956). "Gedanken experiments on sequential machines." in *Automata studies*, pages 129–153. Princeton University Press, Princeton, NJ.

[39] PAPADIMITRIOU, C. H., AND STEIGLITZ, K. (1982). *Combinatorial Optimization: algorithms and complexity.* Prentice Hall, Englewood Cliffs, NJ.

[40] PÓLYA (1937). "Kombinatorische Anzahlbestimmungen für Gruppen, Graphen, und chemische Verbindungen." *Acta Math.* 68:145–253. Translated as *Combinatorial Enumeration of Groups, Graphs and Chemical Compounds*, by R. C. Read (1987), Springer-Verlag, New York, NY.

[41] READ, R. C. (1972). "The coding of various kinds of unlabeled trees." in *Graph Theory and Computing*, pages 153–182. Academic Press, New York, NY.

[42] SMITH, H. F. (1987). *Data Structures: Form and Function.* Harcourt Brace Jovanovich, San Diego, CA.

[43] STOUT, Q. F. (1980). "Improved Prefix Encodings of the Natural Numbers." *IEEE Transactions on Information Theory* **IT-26**(5):607–609.

[44] TARJAN, R. E. (1983). *Data Structures and Network Algorithms.* SIAM, Philadelphia, MA.

[45] TARJAN, R. E., AND YAO, A. C. (1979). "Storing a Sparse Table." *Communications of the ACM* **22**(11):606–611.

[46] TURÁN, G. (1984). "Succinct Representations of Graphs." *Discrete Applied Math* 8:289–294.

[47] WEINER, P., (1973). "Linear pattern matching algorithms." *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11.

[48] WEINREB, D., AND MOON, D. (1981). *Lisp Machine Manual, Fourth Edition.* Massachusetts Institute of Technology, Cambridge, MA.

[49] YANNAKAKIS, M. (1986). "Four pages are necessary and sufficient for planar graphs." *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 104–108.

[50] YAO, A. C. (1981). "Should Tables be Sorted?" *Journal of the ACM* **28**(3):615–628.