Ranking and Unranking Trees Using Regular Reductions

by
Pierre Kelsen

Technical Report 93-37
October 1993

Department of Computer Science
University of British Columbia
Rm 201 - 2366 Main Mall
Vancouver, B.C.
CANADA V6T 1Z4

Telephone: (604) 822-3061
Fax: (604) 822-5485

Ranking and Unranking Trees Using Regular Reductions

Pierre Kelsen*

Abstract

We consider the problem of defining a linear order on a set of combinatorial objects so that the following two operations can be performed efficiently: (1) determine the rank of an object in the linear order (*ranking*); (2) compute an object from its rank (*unranking*). Typical applications of such an ordering include testing a program on a random or selected set of input instances and searching for counterexamples of a conjecture involving structured objects.

We reduce the problem of finding such a linear order to the problem of constructing a special mapping on the set of combinatorial objects; we call such a mapping a *regular reduction*. We demonstrate the use of regular reductions by improving several methods for ordering families of rooted trees: we propose ranking and unranking procedures for height-balanced trees on n leaves that run in time $O(n)$ and $O(n \log n)$, respectively, after $O(n^2 \log n)$ preprocessing, improved from $O(n \log^2 n)$ ranking, $O(n \log^3 n)$ unranking, and $O(n^2 \log n)$ preprocessing. For B-trees on n leaves we describe ranking and unranking procedures that both run in time $O(n)$, after $O(n^2)$ preprocessing; previous methods, although running in linear time, are fairly involved and require exponential time (and space) preprocessing. Finally, we propose linear time procedures for ranking and unranking binary trees of bounded height after polynomial time preprocessing; no polynomial time ordering algorithms for this class of trees were previously known.

1 Introduction

We consider the problem of defining a linear order on a set of combinatorial objects so that the following two operations can be performed efficiently: (1) given an object, compute its *rank* in the linear order, i.e., the number of objects that precede it in the linear order; (2) given the rank of an object, construct the object. Tasks (1) and (2) are usually referred to as *ranking* and *unranking*, respectively. Note that we obtain a random combinatorial object by applying the unranking procedure to a random rank and we obtain a list of all combinatorial ojects by invoking the unranking procedure for all ranks. We refer to the problem of finding such a linear ordering as the *ordering problem*.

There are several natural applications for the ordering problem. We may consider the input to an algorithm to be a combinatorial object. Ordering the set of these objects allows us to generate random inputs or certain subsets of possible inputs. This is useful for testing a program or measuring its running time. Another application is the generation of combinatorial objects to test the validity of a conjecture. For instance, one may want to check a conjecture about graphs by generating all graphs of a given size or a subset of them.

*The author was supported by the Natural Sciences and Engineering Research Council of Canada. Address of the author: Department of Computer Science, University of British Columbia, 2366 Main Mall, Vancouver, B.C. CANADA V6T 1Z4; e-mail: kelsen@cs.ubc.ca.

In this paper we shall focus on the ordering problem for rooted trees for two reasons: first, these trees are among the most widely used structured objects in the design of computer algorithms; second, our methods prove particularly effective in this context. Much attention has been devouted to the ordering of trees in the last decade (see e.g. [5, 6, 7, 9, 10, 11, 12, 13, 17, 18]). references). Typically, a given family of trees (e.g., binary trees on n nodes) is ordered by first establishing a correspondence between the trees and certain integer sequences. Ordering algorithms are then developed for these integer sequences. This usually involves assigning a weight to each integer sequence and deriving a recurrence relation for the number of sequences of a given weight. The main drawback of this approach is that finding a suitable encoding may be quite difficult. This difficulty is underlined by the lack of linear time ordering algorithms for many important families of rooted trees.

We propose a new approach that reduces the ordering problem to the problem of finding a special mapping on the set of combinatorial objects; we call such a mapping a *regular reduction*. We exhibit an intimate connection between regular reductions and recurrence relations: every regular reduction yields a recurrence relation from which a solution to the ordering problem can be derived using standard methods. Conversely, we may associate with a wide class of recurrence relations regular reductions. As a result most ordering methods for trees may be reformulated within the framework of regular reductions.

We demonstrate the power of regular reductions for ordering trees by constructing simple regular reductions for several families of trees. The resulting ordering procedures are simple and efficient and improve several results from the literature. For height-balanced trees on n leaves we propose ranking and unranking algorithms that run in time $O(n)$ and $O(n \log n)$, respectively, after $O(n^2 \log n)$ preprocessing, improved from $O(n \log^2 n)$ ranking, $O(n \log^3 n)$ unranking, and $O(n^2 \log n)$ preprocessing. For B-trees on n leaves we describe ranking and unranking procedures that both run in time $O(n)$ after $O(n^2)$ preprocessing; previous methods, although running in linear time, are fairly involved and require exponential time preprocessing. Furthermore, we have linear time procedures for ranking and unranking binary trees with n nodes and height at most h after $O(n^4)$ time preprocessing; no polynomial time ranking or unranking algorithms for this class of trees were previously known.

This paper is organized as follows: in section 2 we formally define the ordering problem and introduce the combinatorial family of set partitions as a typical example. In section 3 we relate our methods to previous work in this area. In section 4 we introduce the concept of a regular reduction and study the connection between regular reductions and recurrence relations. In section 5 we outline the various steps in the solution of the ordering problem. We use regular reductions in section 6 to order height-balanced trees, binary trees of bounded height, and B-trees and obtain the results mentioned above. We conclude in section 7 by summarizing the contributions of this paper and discussing other potential applications for regular reductions.

A preliminary version of this work appeared in [7].

2 Preliminaries

2.1 The Ordering Problem

The following is a rather general statement of the ordering problem: we are given a set S of *combinatorial objects* (e.g., trees), a countable set W of *weights*, and a *weight function* $\omega : S \to W$. We call $\omega(s)$ the *weight* of object $s \in S$ and denote by $S(w)$ the subset of elements in S of weight w. We shall assume that the number of objects of any given weight

is finite. The *ordering problem* is to determine a linear order $<_w$ on each set $S(w)$ so that the following operations can be performed efficiently:

(1) Given an element $s \in S(w)$ compute the rank of s, i.e., the number of objects in $S(w)$ that precede it in the linear order $<_w$; we denote this number by the function $rank(s)$.

(2) Given a number i in the range $0, \ldots, |S(w)| - 1$ return the element $s \in S(w)$ with $rank(s) = i$; let the function $unrank(w, i)$ denote this element.

We note that we can generate an object of $S(w)$ uniformly at random by chosing an integer i uniformly at random from the range $0 \ldots |S(w)| - 1$ and invoking $unrank(w, i)$. We also note that we can generate all objects in $S(w)$ by simply invoking $unrank(w, i)$ for $i = 0, 1, \ldots, |S(w)| - 1$.

Caveat: It is standard practice to measure the complexity of algorithms implementing the functions *rank* and *unrank* under the assumption that numbers representing ranks of elements in S fit into a single memory word (and hence standard RAM instructions on these numbers can be executed in constant time); this is sometimes referred to as the *unit cost criterion* ([1]). One has to bear in mind, however, that these numbers can be quite large. To get a more realistic estimate on the actual running time one may set the cost of a RAM instruction proportional to the number of bits in the largest operand; this is usually referred to as the *logarithmic cost criterion.* In this paper we express the time and space requirement in the unit cost model to facilitate comparisons with earlier results. These bounds are easily converted to the logarithmic cost model.

2.2 An Example: Set Partitions

We shall illustrate various concepts and ideas in this paper using the example of *set partitions.* We choose set partitions instead of trees because they are simpler than trees yet complex enough to illustrate the basic concepts.

For a positive integer n let $[n]$ denote the set $\{1, \ldots, n\}$. A *partition* of $[n]$ is a set of pairwise disjoint nonempty subsets of $[n]$, called *blocks*, whose union is $[n]$. It will be convenient to have a linear order on the blocks of a partition. We define the *canonical order* to be the arrangement of blocks in increasing order of their smallest element. Thus, the blocks of the partition $\{\{3\}, \{2,4\}, \{5,7\}, \{1,6\}\}$ of $[7]$ are, in canonical order, $\{1,6\}$, $\{2,4\}$, $\{3\}$, and $\{5,7\}$.

Let $P(n,k)$ denote the set of partitions of $[n]$ with exactly k blocks. The following recurrence relation for $S(n,k) = |P(n,k)|$ (the *Stirling numbers of the second kind*) is well-known (see e.g. [3]) and will be repeatedly referred to in the sequel:

$$S(n,k) = S(n-1,k-1) + k \cdot S(n-1,k) \tag{1}$$

for $1 < k < n$ with the boundary conditions $S(n,n) = S(n,1) = 1$ for $n \geq 1$. We place set partitions within the general framework of the previous subsection as follows: the set S of combinatorial objects is the union $\cup_{n \geq 1, 1 \leq k \leq n} P(n,k)$; the weight function ω assigns to each partition in S as weight the pair (n,k) where n is the total number of elements it contains and k is the number of blocks it contains. Thus, $P(n,k)$ comprises exactly those elements in S of weight (n,k).

3 Previous Work

In this section we take a brief look at two general models for ordering combinatorial objects that have been proposed in the literature and explain how our techniques relate to these models.

We first review the model of Wilf ([15]). The basic idea of Wilf is to represent combinatorial objects by paths in a digraph, thereby reducing the problem of ordering the combinatorial objects to that of ordering the associated paths. We illustrate the basic ideas with the help of set partitions. Construct a directed graph whose nodes are the pairs (n, k) where $n \geq 1$ and $1 \leq k \leq n$. Connect each vertex (n, k) with $1 < k < n$ by k directed edges (distinguished by unique labels) to $(n-1, k)$ and by a single edge to $(n-1, k-1)$; also for each $n > 1$ connect vertex (n, n) by a single edge to $(n-1, n-1)$ and $(n, 1)$ with a single edge to $(n-1, 1)$. Let $b(n, k)$ denote the number of paths in this directed graph from vertex (n, k) to $(1, 1)$. From the construction of the digraph it follows that $b(n, k)$ satisfies the same recurrence relation as $S(n, k)$. Hence we have $S(n, k) = b(n, k)$ for $n \geq 1$ and $1 \leq k \leq n$. One can establish a simple bijective correspondence between partitions in $P(n, k)$ and paths from (n, k) to $(1, 1)$. Fairly straightforward algorithms for ranking and unranking these paths may thus be used to perform these tasks for the partitions in $P(n, k)$ (see [15]).

An equivalent model is proposed by Williamson ([16]). Again the basic concepts are best illustrated with the example of set partitions. We may interpret recurrence relation 1 as follows: the set $P(n, k)$ may be partitioned into $k+1$ subsets $B_0, B_1, \ldots, B_k$ such that B_0 contains those partitions with n in a block by itself and B_i $(i > 0)$ contains those partitions with n sharing the ith block in canonical order with at least one other element. By identifying B_0 with $P(n-1, k-1)$ and each B_i $(i > 0)$ with $P(n-1, k)$ we may recursively partition each B_i in this way. We thus obtain a sequence of partitions of $P(n, k)$, each partition being a refinement of the preceding one. We can represent this chain of partitions by a tree, the root of the tree being $P(n, k)$ and each leaf corresponding to a single element in $P(n, k)$. By ordering the children of each node we induce an ordering on the leaves. Based on this ordering ranking and unranking may be done in a fairly straightforward way.

Both of these general methods rely on linear recurrence relations (such as equation 1) for the number of objects of a given weight. Although such recurrence relations are well-known for many classical combinatorial families such as permutations, derangements or partitions this is not true for more complex objects such as trees. The standard approach for ordering trees consists of encoding the trees by a restricted family of integer sequences. By associating suitable weights with each sequence, one may be able to derive a recurrence relation for the number of sequences (and hence the number of trees) of a given weight. Once a recurrence relation has been obtained, methods similar to those described above may then be used to solve the ordering problem. For restricted families of trees such as those considered in section 6 the task of finding a suitable encoding may be quite difficult.

In the next section we reduce the ordering problem to the problem of defining a special mapping on the set of combinatorial objects – we call such a mapping a *regular reduction*. We show that for any combinatorial family that admits a regular reduction there is a recurrence relation for the number of objects of a given weight. This will enable us to use methods similar to those described above for ranking and unranking the objects. We demonstrate this approach in section 6 where we exhibit simple regular reductions for several families of trees that yield fast and simple ordering procedures.

4 Regular Trees and Regular Reductions

Up to now we have used the term *combinatorial objects* in a very general sense: in the statement of the ordering problem we only require that each object has a *weight* and that there are a finite number of objects of a given weight. Clearly, in order to design a solution to the ordering problem we need to know more about the structure of the combinatorial family. In this section we restrict the structure of combinatorial families and show that for these restricted families we can solve the ordering problem in a systematic way.

4.1 Terminology

We first define some basic terms used in the sequel. A *rooted tree* is a directed graph $T = (V, E)$ with a distinguished node $r \in V$ such that r has outdegree 0, all other vertices are outdegree 1, and there is a directed path from every vertex to r. In this case we say that r is *reachable* from every vertex. If (v, w) is an edge in T, then v is the *child* of w and w is the *parent* of v. A *leaf* in a rooted tree is a childless node. The *subtree* of T rooted at a node v of T is the tree consisting of all the vertices in T from which v can be reached (including v) and having as edges the edges of T between these vertices. The *depth* of a node in T is the number of edges on the unique path from this node to the root. An *internal* node of T is any node of T other than a leaf.

4.2 Regular Trees

As in section 2 we denote by S the set of combinatorial objects and by $\omega(s)$ the weight of an object $s \in S$. Suppose that the objects in S are being constructed from *simpler* objects using some transformation rules. In the case where each object is built from at most one other object we may associate with the set S a directed graph G: the vertices in G are the combinatorial objects in S; there is an edge (s', s) if s' is constructed from s. Note that each vertex has outdegree at most 1 in G. An interesting special case arises when the digraph is a rooted tree, i.e., there is a unique vertex of outdegree 0 (*the root*) that is reachable from all other vertices (having outdegree 1). By labeling each vertex s in this tree with its weight $\omega(s)$, we obtain a *weighted tree for* (S, ω). This will usually be an infinite tree.

As an example consider the following construction rule for set partitions: from any partition $p \in P(n, k)$ we construct k partitions in $P(n+1, k)$ by inserting $n+1$ in one of the k blocks of p and a single partition in $P(n+1, k+1)$ by creating a new singleton block for $n+1$. An initial portion of the corresponding weighted tree is depicted in figure 1. (Following standard practice we omit the direction of the edges; all edges are assumed to point upwards.) Let T and T' be two weighted trees for (S, ω) and (S', ω'), respectively. We say that T and T' are *isomorphic* if there exists a bijection $h : S \to S'$ such that the following two conditions hold: (1) for any $x, y \in S$, y is a child of x in T if and only if $h(y)$ is a child of $h(x)$ in T'; (2) for any $x \in S$, $\omega(x) = \omega'(h(x))$. A mapping with these properties is called an *isomorphism* from T to T'.

The weighted tree for set partitions whose initial portion is shown in figure 1 exhibits the following important property: if two nodes have the same weight, then the two subtrees rooted at these nodes are isomorphic. If a weighted tree for (S, ω) has this property, then we call the tree a *regular tree* for (S, ω), or just regular tree if S and ω are clear from the context. Let T be a weighted tree for (S, ω). We denote the set of children of a node x by $C(x)$ and the multiset of the weights of these children by $\omega(C(x))$. The following theorem states a simple condition for a weighted tree to be regular.

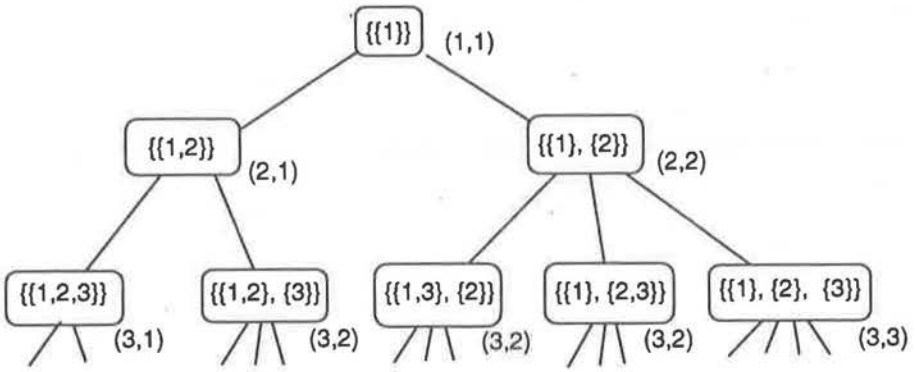

Figure 1: Initial portion of a weighted tree for set partitions. The weight of a node is written next to the node.

Theorem 1 *A weighted tree T for (S, ω) is a regular tree if and only if the multisets $\omega(C(s))$ and $\omega(C(s'))$ are equal for any two nodes s and s' in T with $\omega(s) = \omega(s')$.*

Proof. Throughout this proof $T(s)$ denotes the subtree of T rooted at a node s in T; its vertex set are all nodes in T from which s is reachable. If T is regular and s and s' are two nodes of equal weight in T, then an isomorphism from $T(s)$ to $T(s')$ must map s to s' and the children of s to the children of s'. Since the isomorphism is weight-preserving, it follows that $\omega(C(s))$ and $\omega(C(s'))$ are equal. For the converse assume $\omega(C(s)) = \omega(C(s'))$ if $\omega(s) = \omega(s')$. Fix two nodes s and s' in T with the same weight; hence $\omega(C(s)) = \omega(C(s'))$. We need to show that $T(s)$ and $T(s')$ are isomorphic. We define a bijective mapping h from the vertex set of $T(s)$ to the vertex set of $T(s')$ inductively as follows: $h(s) = s'$ and, if x is a child of a node y in $T(s)$, then $h(x)$ is a child of $h(y)$ of weight $\omega(x)$. Note that these two conditions do not uniquely specify the mapping. The existence of a bijective mapping with these properties follows, however, from our assumption. Furthermore, such a mapping is clearly an isomorphism from $T(s)$ to $T(s')$. Hence T is regular. □

Corollary 1 *Assume each node in a regular tree T has finite indegree. Then there exists a linear order $<_W$ on W such that each weight w has finite rank in $<_W$ and $\omega(f(s)) <_W \omega(s)$ for every $s \in S$ other than the root of T.*

Proof. For $w \in W$, let $d(w)$ denote the maximum depth of a node of weight w in a regular tree T. Since each $S(w)$ is finite, $d(w)$ is indeed well-defined. By setting $w < w'$ if $d(w) < d(w')$, we obtain a partial ordering on W that may be extended into a linear order $<_W$ on W by arbitrarily ordering the weights with same d-value. If $f(s)$ denotes the parent of a non-root node s in T, then theorem 1 implies that $d(\omega(f(s)) < d(\omega(s))$ since every node of weight $\omega(f(s))$ has a child of weight $\omega(s)$ in T. If in addition each node in T has finite indegree (the only case that we shall consider in this paper), then each element in W has finite rank in $<_W$. The result follows. □

Another useful consequence of theorem 1 is the fact that we may extract from a regular tree T a mapping t that assigns to each weight w the multiset $\omega(C(s))$ for an arbitrary $s \in S(w)$. We call the mapping t the *type* of the regular tree. As an example consider the regular tree for set partitions (figure 1): using the notation $m_1 \bullet x_1 + m_2 \bullet x_2 + \ldots + m_k \bullet x_k$ for a multiset containing exactly m_i copies of some object x_i $(1 \leq i \leq k)$ we may write the type of the regular tree for set partitions as $t(n,k) = k \bullet (n+1,k) + 1 \bullet (n+1,k+1)$.

4.3 Regular Reductions

Each rooted tree comes equipped with a natural mapping, namely the mapping that maps each node other than the root to its parent. If the rooted tree is a regular tree for (S,ω), then we call this mapping a *regular reduction* for (S,ω). Regular reductions provide an elegant and convenient characterization of regular trees.

In order to come up with a regular reduction directly (without first constructing the corresponding regular tree), we need a simple criterion for checking whether a given mapping is indeed a regular reduction. Such a criterion is provided by the following theorem.

Theorem 2 *Fix a combinatorial family (S,ω) and a distinguished element $s_0 \in S$. A mapping $f : S\backslash\{s_0\} \to S$ is a regular reduction if and only if the following two conditions are satisfied:*

(i) for any $s \in S$ there exists an integer $i \geq 0$ such that $f^{(i)}(s) = s_0$ (here $f^{(i)}$ denotes the function f composed with itself i times) (finiteness condition)*;*

(ii) for any $s, s' \in S$, if $\omega(s) = \omega(s')$ then the multisets $\omega(f^{-1}(s))$ and $\omega(f^{-1}(s'))$ are equal (regularity condition)*.*

Proof. If f is a regular reduction, then certainly the finiteness and regularity conditions hold. Conversely, if the finiteness condition holds, then the graph with vertex set S and edge set $\{(s, f(s)) : s \in S, s \neq s_0\}$ is a tree rooted at s_0. From the regularity condition and theorem 1 it follows that this tree is regular and hence f is a regular reduction. □

The *type* of a regular reduction is the type of the associated regular tree, i.e., the mapping that assigns to $w \in W$ the multiset of weights $\omega(f^{-1}(s))$ for $s \in S(w)$. As an example consider the family of set partitions. Define a mapping *cut* on $\cup_{n,k} P(n,k)$ as follows: if $p \in P(n,k)$ where $n > 1$ then $cut(p)$ is obtained from p by removing element n from its block and removing the resulting block if it has become empty; if $p = \{\{1\}\}$ then $cut(p)$ is undefined. Thus, if $p = \{\{1,3\},\{2,5\},\{4\}\}$ then $cut(p) = \{\{1,3\},\{2\},\{4\}\}$, $cut^{(2)}(p) = \{\{1,3\},\{2\}\}$, $cut^{(3)}(p) = \{\{1\},\{2\}\}$ and $cut^{(4)}(p) = \{\{1\}\}$. The finiteness condition of theorem 2 is easily checked. For the regularity condition note that if $p \in P(n,k)$ then $\omega(cut^{-1}(p))$ is the multiset $k \bullet (n+1,k) + 1 \bullet (n+1,k+1)$. Thus, f is indeed a regular reduction whose type is given by the last expression. The corresponding regular tree is the tree of figure 1.

4.4 Recurrence relations

The importance of regular reductions in the context of the ordering problem is due to the fact that each regular reduction (and each regular tree) yields a recurrence relation which allows us to use ordering procedures similar to those outlined in section 3.

Fix a regular reduction $f : S\backslash\{s_0\} \to S$ and let T be the corresponding regular tree. As usual we denote by $S(w)$ the set of elements in S of weight $w \in W$; we write $s(w)$ for $|S(w)|$, the number of objects of weight w. For weights $w, z \in W$ let $d(w,z)$ represent the number of nodes of weight w that are children of a node in T of weight z. Also let $<_W$ be a total order on W such that each weight has finite rank in $<_W$ and $\omega(f(s)) < \omega(s)$ for all $s \in S\backslash\{s_0\}$. By corollary 1 such an ordering does exist. The following result elucidates the relationship between regular reductions and recurrence relations.

Theorem 3 *The quantities $s(w)$ satisfy the recurrence relation*

$$s(w) = \sum_{z \in W, z <_W w} d(w,z) \cdot s(z) \tag{2}$$

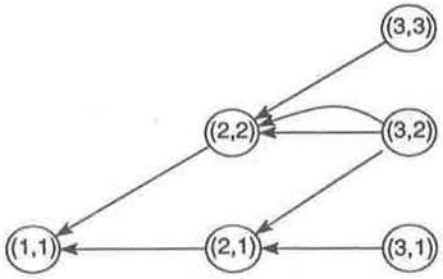

Figure 2: Structure graph for set partitions with respect to reduction *cut*.

for $w \neq \omega(s_0)$, with the boundary condition $s(\omega(s_0)) = 1$.

Proof. For every $s \neq s_0$ we have $\omega(s_0) <_W \omega(s)$, hence the boundary condition. Each node of weight z has $d(w,z)$ children of weight w. Thus, the set of nodes of weight w whose parent has weight z has cardinality $d(w,z)s(z)$. Since these sets are disjoint for different values of z and $\omega(f(s)) <_W w$ for $s \in S(w)$, the claim follows. □

We remark that there may be no closed-form expression for the quantities $d(w,z)$.

We may use recurrence relation 2 to represent S by paths in a directed graph as in Wilf's model (see section 3): the vertex set of the graph is the set W of weights of objects in S. The edge multiset E is given by the formal expression

$$E = \sum_{w \in W \setminus \{\omega(s_0)\}, z <_W w} d(w,z) \bullet (w,z),$$

i.e., each vertex $w \neq \omega(s_0)$ is linked by $d(w,z)$ edges to vertex z for any $z \in W$. We assume that parallel edges are distinguished by unique labels. We call this graph the *structure graph* for (S,ω) with respect to reduction f – structure graph for short if S, ω and f are understood. The number of (labeled) paths from w to $\omega(s_0)$ satisfies the same recurrence relation as the number $s(w)$ of objects of weight w (see recurrence 2). Hence, there is a one-to-one correspondence between elements in $S(w)$ and paths from w to $\omega(s_0)$. Thus, we may rank and unrank the objects in $S(w)$ by performing these tasks for the paths from w to $\omega(s_0)$ as in Wilf's model. We shall describe these procedures in more detail in the next section. Figure 2 shows an initial portion of the structure graph corresponding to regular reduction *cut* defined above on set partitions; the corresponding regular tree is the tree of figure 1.

We have just seen that every regular reduction yields a representation of the combinatorial objects as paths in the structure graph. The converse holds as well: suppose that G is a directed acyclic graph with vertex set W and a distinguished vertex w_0 satisfying the following three conditions: (1) w_0 is the unique vertex of outdegree 0 in G; (2) there is no infinite path starting at a fixed vertex in G; (3) the number of paths from w to w_0 equals $s(w)$. Let $P(w)$ denote the set of paths in G from w to w_0 and let $P = \cup_{w \in W} P(w)$. Let p_0 denote the trivial path consisting of the single vertex w_0. Consider the mapping $g : P \setminus \{p_0\} \to P$ that shortens a path in $P \setminus \{p_0\}$ by removing its first edge. This mapping is clearly a regular reduction for P with a path being assigned as weight its starting vertex. Since $|P(w)| = s(w)$ for all $w \in W$ there exists a bijective correspondence between $P(w)$ and $S(w)$ for $w \in W$. This correspondence together with regular reduction g yields a regular reduction for (S,ω) in the obvious way. We conclude that a combinatorial family satisfying a recurrence relation of the general form given by equation 2 admits a regular reduction. Thus, regular reductions

do indeed characterize a large class of combinatorial families for which the ordering problem can be solved in a systematic way (as shown in the next section).

5 Solving the Ordering Problem

The previous sections suggest the following approach to the ordering problem: first, we construct a regular reduction for the combinatorial family. From the regular reduction we derive a recurrence relation for the number of objects of a given weight. This recurrence allows us to reduce the problem of ordering the combinatorial objects to that of ordering paths in a directed graph, for which we may use the techniques described in section 3. In all examples that we shall consider the recurrence relations are fairly simple and ranking and unranking may be done without explicitly constructing this graph.

In the remainder of this section we describe in some detail the various steps that we have just outlined. In particular we present techniques for simplifying some of the steps.

5.1 Constructing the Regular Reduction

For the following discussion we fix a combinatorial family (S, ω) with weight set W. Our first goal is to construct a regular reduction for (S, ω), i.e., a mapping $f : S \backslash \{s_0\} \rightarrow S$ that satisfies the conditions of theorem 2.

Although many types of rooted trees admit very simple regular reductions (see section 6), there are two useful techniques for simplifying the task of finding a regular reduction. For the first technique we assume that we have constructed a regular reduction for a combinatorial family (S', ω') with weight set W' having the following properties: (1) $S \subseteq S'$; (2) for every $w \in W$ there exists a weight $w' \in W'$ such that $S(w) = S'(w')$. Under these assumptions the problem of ordering the elements in S of weight $w \in W$ reduces to the problem of ordering the elements in S' of some weight $w' \in W'$. We call this technique the *embedding technique.* In subsection 6.4 we shall make use of the embedding technique to order B-trees.

For the second technique assume that we have a mapping $f : S \backslash \{s_0\} \rightarrow S$ that satisfies the finiteness condition of theorem 2 but not the regularity condition. In some cases we may be able to define a new weight function ω' on S such that the following holds: (1) ω' is a *refinement* of ω, i.e., $\omega'(s) = \omega'(s')$ implies $\omega(s) = \omega(s')$ for any $s, s' \in S$; (2) f is a regular reduction for (S, ω'). If these assumptions hold, then we may write $S_\omega(w) = \{s \in S : \omega(s) = w\}$ as a finite union of sets of the form $S_{\omega'}(w) = \{s \in S : \omega'(s) = w\}$. We can thus use ordering procedures for (S, ω') to order (S, ω) in the obvious way: return as the rank of an object in $S_\omega(w)$ the rank of the object in $S_{\omega'}(\omega'(s))$ augmented by the quantity $|\{s' \in S_\omega(w) : \omega'(s') <_W \omega'(s)\}|$. To compute the object s in $S_\omega(w)$ of rank r compute the set $S_{\omega'}(w)$ containing s and return the object in $S_{\omega'}(w)$ of rank $r - |\{s' \in S_\omega(w) : \omega'(s') <_W \omega'(s)\}|$. We call this technique the *refinement technique.* We shall use this technique to order height-balanced trees and binary trees of bounded height (subsections 6.2 and 6.3).

5.2 Deriving the Recurrence Relation

Suppose we have found a regular reduction $f : S \backslash \{s_0\} \rightarrow S$ for (S, ω). We may view f as a mapping that maps vertices other than the root s_0 in a regular tree T for (S, ω) to their parent. Fix a linear order $<_W$ on W having the properties given by corollary 1. By theorem

3 the quantity $s(w) = |S(w)|$ satisfies the recurrence relation

$$s(w) = \sum_{z \in W, z <_W w} d(w,z) \cdot s(z), \tag{3}$$

where $d(w,z)$ is the number of children of weight w of a node of weight z in the regular tree T; equivalently, $d(w,z)$ is the number of elements in S of weight w that f maps to a fixed element of weight z.

In all the examples that we shall consider there is a closed form expression for the coefficients $d(w,z)$ that can be obtained from the type equation in a straightforward manner. As an example consider set partitions. In the last section we introduced the reduction *cut* of type $t(n,k) = k \bullet (n+1,k) + 1 \bullet (n+1,k+1)$. Thus, the coimage of a partition in $P(n,k)$ has either weight $(n-1,k)$ or $(n-1,k-1)$. Furthermore $d((n,k),(n-1,k)) = k$ and $d((n,k),(n-1,k-1)) = 1$. Thus we get the recurrence relation $s(n,k) = k \cdot s(n-1,k) + s(n-1,k-1)$. This is of course the well-known recurrence for Stirling numbers of the second kind (equation 1).

5.3 Ranking and Unranking

Our first task is to define a linear order $<_w$ on each set $S(w)$. The linear order and subsequent ranking and unranking procedures that we present are essentially adaptations of those by Wilf ([15]) to the framework of regular reductions. Let f and $<_W$ be as in the previous subsection. For $s \in S$ and $w \in W$ denote by $S(w,s)$ the set $\{s' \in S(w) : f(s') = s\}$. Note that $|S(w,s)| = d(w,\omega(s))$. Let *index* be a function from S to the nonnegative integers that maps elements in $S(w,s)$ bijectively to the set $\{0,\ldots,|S(w,s)|-1\}$ for each $s \in S$ and $w \in W$. The function *index* induces a linear order on each set $S(w,s)$ in the obvious way.

To define a linear order $<_w$ on $S(w)$, let s and s' be two distinct objects in $S(w)$ with $z = \omega(f(s))$ and $z' = \omega(f(s'))$. We define $<_w$ inductively as follows:

$$\begin{aligned} s <_w s' \text{ iff} \quad & (z <_W z') \text{ or} \\ & (z = z' \text{ and } index(s) < index(s')) \text{ or} \\ & (z = z' \text{ and } index(s) = index(s') \text{ and } f(s) <_z f(s')). \end{aligned} \tag{4}$$

Since $z <_W w$ (see corollary 1), $<_w$ is indeed well-defined. It is helpful to interpret the linear order $<_w$ in terms of the structure graph (see subsection 4.4). For fixed $s \in S$ let k be the unique integer such that $f^k(s) = s_0$. Let

$$t_i = \langle \omega(f^i(s)), \omega(f^{i+1}(s)), index(f^i(s)) \rangle$$

for $0 \le i < k$. In particular $t_0 = \langle \omega(s), \omega(f(s)), index(f(s)) \rangle$. We call the sequence $\langle t_0, t_1, \ldots, t_{k-1} \rangle$ the *reduction sequence* for s. We may regard each triple t_i as a labeled edge in the structure graph from vertex $\omega(f^i(s))$ to $\omega(f^{i+1}(s))$ and the reduction sequence as the path in the structure graph from $\omega(s)$ to $\omega(s_0)$ that represents s. It is not difficult to see that the linear order $<_w$ corresponds to the lexicographic ordering on the reduction sequences for elements in $S(w)$.

Lemma 1 *Let $s \in S(w)$ with $z = \omega(f(s))$. Let $rank(s) = |\{s' \in S(w) : s' <_w s\}|$. Then*

$$rank(s) = \sum_{z' \in W, z' <_W z} d(w,z') \cdot s(z') + index(s) \cdot s(z) + rank(f(s)).$$

Proof. We have $rank(s) = |A| + |B| + |C|$ where $A = \{s' \in S(w) : \omega(f(s')) <_W z\}$, $B = \{s' \in S(w) : \omega(f(s')) = z \wedge index(s') < index(s)\}$ and $C = \{s' \in S(w) : \omega(f(s')) = z \wedge index(s') = index(s) \wedge f(s') <_z f(s)\}$. We have $|A| = \sum_{z' <_W z} d(w, z') \cdot s(z')$, $|B| = index(s) \cdot s(z)$, and $|C| = rank(f(s))$. The claim of the lemma follows. □

For $d(w, z) > 0$ let $g(w, z)$ denote the quantity $\sum_{z' <_W z} d(w, z')s(z')$, i.e., $g(w, z)$ is the number of objects of weight w that map to an object of weight less than z (in $<_W$). We shall assume that the quantities $g(z, z')$ and $s(z)$ have been precomputed for $z, z' \leq_W w$. From lemma 1 we obtain the following recursive implementation of the function *rank*:

function rank (s: object in S): integer;
if $s = s_0$ then $rank := 0$
else
    $w := \omega(s)$;
    $z := \omega(f(s))$;
    $return(g(w, z) + index(s) \cdot s(z) + rank(f(s)))$;
endif;

With the help of lemma 1 it also straightforward to formulate the inverse function $unrank(w, r)$ which returns the unique object s in $S(w)$ with $rank(s) = r$ (assume $0 \leq r < s(w)$). In the following formulation, $obj(s', w, l)$ denotes the unique object $s \in S(w)$ such that $f(s) = s'$ (i.e., $s \in S(w, s')$) and $index(s) = l$.

function unrank (w: weight, r: integer): object in S;
if $w = \omega(s_0)$ then $return(s_0)$
else
    $z :=$ largest weight such that $g(w, z) \leq r$;
    $l := (r - g(w, z))$ div $s(z)$;
    $r' := r - g(w, z) - l \cdot s(z)$;
    $s' := unrank(z, r')$;
    $return(obj(s', w, l))$;
endif;

6 Applications

6.1 Basic Definitions and Notation

Basic definitions concerning rooted trees are given in subsection 4.1. We denote the number of nodes in a rooted tree T by $n(T)$. The *height* of a rooted tree is the maximum number of edges on a path from a leaf to the root; it is denoted by $h(T)$. A *binary tree* is a rooted tree in which every vertex has at most one *left* child and at most one *right* child. We call the subtree rooted at the *left (right) child* of a node v the *left (right) subtree* of v. We denote the left (right) subtree of the root of T by $L(T)$ ($R(T)$). Binary trees are depicted in the usual way with the direction of the edges omitted and the root at the top.

Because we are interested in the structure of the trees rather than the actual labeling, we shall assume that each binary tree has a fixed labeling depending only on the structure of tree, e.g., the vertex set of a tree T is the set $\{1, \ldots, n(T)\}$ and the vertices are numbered level-by-level starting at the root. For the remainder of this section all binary trees are assumed to have such a canonical labeling.

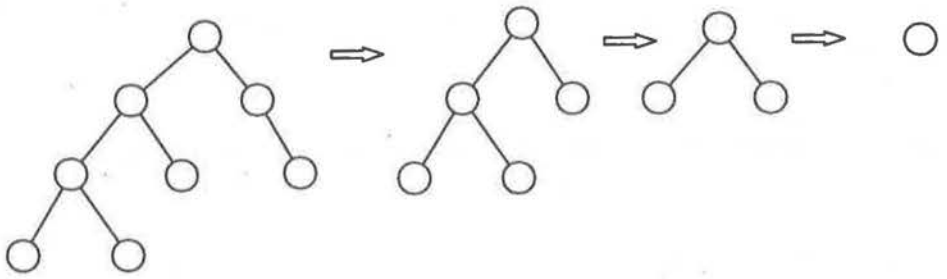

Figure 3: Three applications of $left$; one more application yields the empty tree.

6.2 Height-Balanced Trees

A binary tree is *height-balanced* if at each vertex the height of the left subtree and right subtree differ by at most one. If a subtree is missing it is deemed to be of height -1. Height-balanced trees are also called AVL-trees after their inventors Adel'son-Vel'skii and Landis ([2]). Li ([9]) describes ordering procedures for AVL-trees with n leaves. The ranking and unranking procedures run in time $O(n \log^2 n)$ and $O(n \log^3 n)$ respectively, after a preprocessing phase that takes $O(n^2 \log n)$ time. We derive ranking and unranking algorithms that run in time $O(n)$ and $O(n \log n)$ respectively, after $O(n^2 \log n)$ preprocessing. Besides being faster than those of Li, the algorithms that we propose are also simpler.

The following facts about AVL-trees will be used later: (1) the maximum height of an AVL-tree on n leaves is $\Theta(\log n)$; (2) the number of nodes in an AVL-tree is within a constant factor of the number of leaves in the tree. Fact 1 is an easy corollary of [8, Thm A, p. 453]. Fact 2 follows from the observation that the number of nodes with at least two children is no more than the number of leaves and any node that is a single child must be a leaf.

We use the following notation: $A(n)$ is the set of AVL-trees on n leaves, $A = \cup_{n \geq 0} A(n)$ and $A(n,h)$ is the set of AVL-trees with n leaves and height h. Consider the mapping $left$ that assigns to a non-empty AVL-tree T the left subtree of the root of T (denoted by $L(T)$). Figure 3 depicts a sequence of applications of $left$ to an AVL-tree. Using the refinement technique (see subsection 5.1) we see that $left$ is a regular reduction for (A, ω) where $\omega(T) = (n(T), h(T))$. Indeed $left$ trivially satisfies the finiteness condition of theorem 2. It satisfies the regularity condition since its type is given by

$$t(n,h) = \sum_{i=n_{h-1}}^{N_{h-1}} a(i,h-1) \bullet (n+i,h+1) + \sum_{i=n_h}^{N_h} a(i,h) \bullet (n+i,h+1) + \sum_{i=n_{h+1}}^{N_{h+1}} a(i,h+1) \bullet (n+i,h+2)$$

where $a(n,h) = |A(n,h)|$ and n_h (N_h) denotes the smallest (largest) number of leaves in an AVL-tree of height h. The three terms on the righthand side of the type equation correspond to the cases where the right subtree has height $h-1$, h or $h+1$ (these are the only possible heights since the resulting tree must be height-balanced). From this type equation we obtain the following recurrence relation:

$$a(n,h) = \sum_{i=0}^{n} a(i,h-2)a(n-i,h-1) + \sum_{i=0}^{n} a(i,h-1)[a(n-i,h-2) + a(n-i,h-1)] \quad (5)$$

with boundary conditions $a(n,h) = 0$ for $n \notin \{n_h, \ldots, N_h\}$ and $a(0,-1) = 1$. One could dispose of the first boundary conditions by specifying the precise ranges of i that contribute non-zero terms to recurrence 5; we prefer the current formulation because it is simpler.

Let $W = \{(n', h') : 0 \leq n' \leq n \text{ and } -1 \leq h \leq H_n\}$ where H_n is the maximum height of an AVL-tree on n leaves. By fact 1 we have $H_n = O(\log n)$ and hence $|W| = O(n \log n)$. Let $<_W$ denote the lexicographic order on the weights in W. We precompute two types of quantities: $a(n', h')$ for $(n', h') \in W$ and $g(w, z)$ for $w, z \in W$ (see subsection 5.3). The numbers $a(n', h')$ can be computed from recurrence relation 5 in time $O(n^2 \log n)$ and space $O(n \log n)$. Once these quantities have been computed, the $g(w, z)$ can be computed in a straightforward manner in $O(n^2 \log n)$ time and space.

We define a linear ordering $<_{(n',h')}$ on each set $A(n', h')$ as outlined in subsection 5.3. Let $S(n', h', T)$ denote the subset of trees in $A(n', h')$ that $left$ maps to a fixed AVL-tree T. We define a linear ordering $<_{(n',h',T)}$ on each set $S(n', h', T)$ recursively as follows: $T_1 <_{(n',h',T)} T_2$ for $T_1, T_2 \in S(n', h', T)$ if $h(R(T_1)) < h(R(T_2))$ or $h(R(T_1)) = h(R(T_2))$ and $R(T_1) <_{(n'-n(T),h(R(T_1)))} R(T_2)$. For $T' \in S(n', h', T)$ we denote by $index(T')$ the rank of T' in the ordering $<_{(n',h',T)}$. The linear order $<_W$ and the function $index$ induce a linear order $<_{(n',h')}$ on each $A(n', h')$ as indicated in definition 4 (subsection 5.3).

The functions $rank$ and $unrank$ are now easily adapted from subsection 5.3. In the following formulation of $rank$ we assume that we have precomputed the sizes and heights of all subtrees of T. This is easily done in time $O(n)$ by a postorder traversal of T. (Here we used fact 2 implying that the number of nodes in T is proportional to the number of leaves.) The function $rank$ computes the rank of an AVL-tree T in the linear order $<_{(n(T),h(T))}$.

function rank (T: AVL-tree): integer;
if T is the empty tree then $rank := 0$
else
  $(n, h) := (n(T), h(T))$;
  $(n', h') := (n(L(T)), h(L(T)))$;
  if $(h' = h - 1 \wedge h(R(T)) = h - 1)$ then $d := a(n - n', h - 2)$
    else $d := 0$; /* now $d + rank(R(T)) = index(T)$ */
  endif;
  $return(g((n, h), (n', h')) + (d + rank(R(T))) \cdot a(n', h') + rank(L(T)))$;
endif;

The following function $unrank$ returns the unique AVL-tree in $A(n, h)$ with rank r (in $<_{(n(T),h(T))}$).

function unrank ((n, h): weight; r: integer): AVL-tree;
if $n = 0$ then $return$(empty tree)
else
  search for largest (n', h') such that $g((n, h), (n', h')) \leq r$ (binary search);
  $l := (r - g((n, h), (n', h'))) \text{ div } a(n', h')$;
  $r' := r - g((n, h), (n', h')) - l \cdot a(n', h')$;
  $LT := unrank((n', h'), r')$;
  if $h' = h - 2$ then
    $RT := unrank((n - n', h - 1), l)$
    else if $l \leq a(n - n', h - 2)$ then $RT := unrank((n - n', h - 2), l)$ endif
    else $RT := unrank((n - n', h - 1), l - a(n - n', h - 2))$
  endif;
  $return(T)$ where $L(T) = LT$ and $R(T) = RT$;
endif;

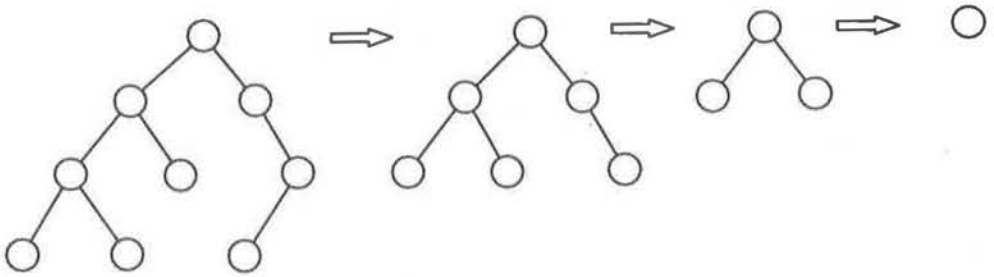

Figure 4: Three applications of *cutlevel*.

Theorem 4 *Functions rank and unrank run in time $O(n)$ and $O(n \log n)$, respectively, where n is the number of leaves in the input (output) tree for rank (unrank).*

Proof. Each execution of the body of rank (excluding the recursive calls) requires time $O(1)$. The number of recursive calls to *rank* is bounded by the number of nodes in T, i.e., it is $O(n)$. The claim for *rank* follows. The execution time of the body of *unrank* is dominated by the time needed in the binary search for (n', h'). The time for this search is $O(\log n)$. Since *unrank* is invoked at most once for each node in the output tree T, the claim follows. □

The above ranking and unranking procedures can easily be extended to ordering procedures for $A(n)$ having the same asymptotic time bounds – $O(n)$ for ranking and $O(n \log n)$ for unranking (see refinement technique, subsection 5.1).

6.3 Binary Trees of Bounded Height

Although there are several efficient algorithms available for ordering binary trees (e.g. [13], [17], [18]), not much is known for the family of binary trees of bounded height, i.e., the family of binary trees with n nodes and height at most h. Lee at al. ([10]) describe an algorithm to generate binary trees of bounded height in constant average time. They pose as an open problem the question of whether there are efficient ranking and unranking algorithms for these trees. We describe a simple regular reduction that yields ranking and unranking algorithms that run in linear time.

Let B denote the family of non-empty binary trees. Our goal is to develop ordering procedures for trees in B with n nodes and height not exceeding h. We shall do so via ordering procedures for (B, ω) where $\omega(T)$ assigns to a binary tree T the triple (n, h, k) where n is the number of nodes in T, h is the height of T, and k is the number of leaves in T of maximum depth. This is an example of an application of the refinement technique (see subsection 5.1). Consider the function *cutlevel* that removes all leaves in T of maximum depth. Figure 4 shows a sequence of applications of *cutlevel* to a binary tree. We note that *cutlevel* is a regular reduction for (B, ω). Its type is given by the equation

$$t(n, h, k) = \sum_{i=1}^{2k} \binom{2k}{i} (n + i, h + 1, i). \tag{6}$$

Let $b(n, h, k)$ denote the number of binary trees of weight (n, h, k). From equation 6 we obtain the recurrence relation

$$b(n, h, k) = \sum_{k/2 \le j < n} \binom{2j}{k} b(n - k, h - 1, j) \tag{7}$$

for $n > 1$ and $1 \leq h, k < n$, with the boundary conditions $b(1,0,1) = 1$, $b(1,h,k) = 0$ for $(h,k) \neq (0,1)$ and $b(n,0,k) = 0$ for $(n,k) \neq (1,1)$. Note that we avoided the problem of determining exactly which $b(n,h,k)$ are nonzero by imposing suitable boundary conditions.

Let $W = \{(n',h',k') : 1 \leq n' \leq n \text{ and } 0 \leq h', k' < n'\}$. We precompute the values $b(n',h',k')$ for $(n',h',k') \in W$. For this we first compute the binomial coefficients $\binom{i}{j}$ for $i < 2n$ and $0 \leq j \leq i$. This takes time and space $\Theta(n^2)$ using the standard recurrence for binomial coefficients. Now the quantities $b(n',h',k')$ can be computed in space $O(n^3)$ and time $O(n^4)$ using recurrence 7. Finally we precompute the quantities $b(n,h) = \sum_k b(n,h,k)$ in time $O(n^3)$ and space $O(n^2)$. Note that $b(n,h)$ is the number of binary trees with n nodes and height h.

Next we define a linear order on each set $B(n,h,k)$. As explained in the last section such an order is uniquely determined by imposing a linear order on the weights and a linear order on trees of the same weight mapped (by *cutlevel*) to the same tree. We define the linear order $<_W$ on W to be the lexicographic order on triples $(n',h',k') \in W$. Now fix a binary tree T of weight (n,h,k). Consider the set S_i of trees of weight $(n+i, h+1, i)$ that map to T. Note that $|S_i| = \binom{2k}{i}$. We may think of constructing a tree $T' \in S_i$ by filling i out of $2k$ "slots" provided by the k leaves at maximum depth in T, where each such leaf provides two slots (one for a right child and one for a left child). Thus, any linear order on i-subsets of $\{1,\ldots,2k\}$ induces a linear order on S_i. If we choose as linear order the lexicographic ordering on the corresponding $2k$-bit vectors, then ranking and unranking of i-subsets can be done in a straightforward way in time $O(k)$ after $O(k^2)$ preprocessing. Impose a linear order on each S_i that corresponds to this subset ordering. Let $index(T')$ denote the rank that the tree T' has in this linear order, i.e., $0 \leq index(T') \leq \binom{2k}{i} - 1$. The function *index* and the linear order $<_W$ yield a linear ordering $<_{(n,h,k)}$ for each $B(n,h,k)$ as described in definition 4 (subsection 5.3).

The functions *rank* and *unrank* are now easily adapted from subsection 5.3. The function rank computes the rank of a tree T in $B(n,h,k)$ with respect to the ordering $<_{(n,h,k)}$.

function rank (T: binary tree): integer;
if $n(T) = 1$ then $rank := 0$
else
    $(n,h,k) := \omega(T)$;
    $(n',h',k') := \omega(cutlevel(T))$;
    $g := 0$;
    for $j := \lceil k/2 \rceil$ to $k' - 1$ do
        $g := g + \binom{2j}{k} b(n-k, h-1, j)$;
    /* now $g = g((n,h,k),(n',h',k'))$; */
    return$(g + index(T) b(n',h',k') + rank(cutlevel(T)))$;
endif;

In the following procedure *unrank* $obj(T',(n,h,k),l)$ denotes the unique tree $T \in B(n,h,k)$ with $cutlevel(T) = T'$ and $index(T) = l$.

function unrank ((n,h,k): weight, r: integer): binary tree;
if $n = 1$ then *return*(tree with single node)
else
    $g := 0$; $j := \lceil k/2 \rceil$;
    while $g + b(n-k, h-1, j) \leq r$ do
        $g := g + b(n-k, h-1, j)$;

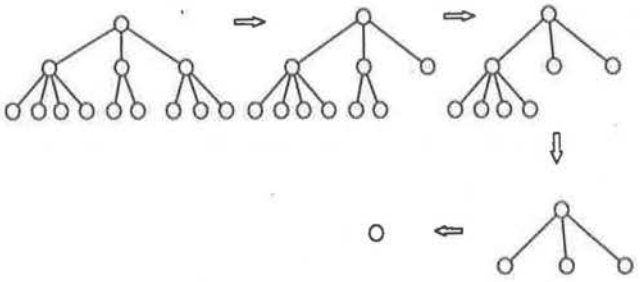

Figure 5: Four applications of *cut* to a B-tree of order 4.

$j := j + 1$;
endwhile;
/* now $(n-k, h-1, j)$ is largest weight such that $g((n,h,k),(n-k,h-1,j)) \leq r$ */
$l := (r - g)$ div $b(n-k, h-1, j)$;
$r' := r - g - l \cdot b(n-k, h-1, j)$;
$T' := unrank((n-k, h-1, j), r')$;
$return(obj(T', (n,h,k), l))$;
endif;

Theorem 5 *The functions rank and unrank run in time $O(n)$.*

Proof. The time required by the statements in the body of the functions rank and unrank (other than the recursive calls) is $O(p)$ where p is the number of leaves in T at levels $h(T)$ and $h(T) - 1$. The claim follows. □

The above ranking and unranking procedures are adapted to the family of trees on n nodes and height at most h in the obvious way without affecting the linear time bound (see refinement technique, subsection 5.1).

6.4 B-trees

A *B-tree* of order m is a rooted tree that satisfies the following three conditions ([4]): (1) the root is either a leaf or has between 2 and m children; (2) all other non-leaf nodes have between $\lceil m/2 \rceil$ and m children; (3) all leaves are at the same level. We are interested in ordering B-trees of order m with n leaves (for fixed m).

Gupta, Lee and Wong ([5]) present ordering algorithms for B-trees that are intimately related to the generation of certain classes of integer partitions. Although the ordering algorithms run in linear time they are fairly involved and they require the construction of an explicit search graph that has exponential size. We describe linear time ranking and unranking procedures for B-trees that are simpler than those of [5]. In particular they only require the precomputation of a table of quadratic size.

A closer look at the algorithms of [5] shows that they are in fact based on the regular reduction that removes all leaves from a B-tree. These algorithms require an explicit search graph because there is no closed form expression for the number of coimages of a given B-tree under this reduction. We remark that there is however a closed form expression for 2-3 trees (B-trees of order 3), leading to fairly simple algorithms for this special case (see [6]). We consider a different reduction *cut*: instead of removing all leaves at once, *cut* removes only the children (leaves) of the rightmost node on the second to last level that is not childless.

Figure 5 depicts a sequence of applications of *cut*. Let B_m denote the set of B-trees of order m. Clearly *cut* cannot be a regular reduction for B_m since the image of a B-tree is usually not a B-tree. We remedy this problem by using the *embedding technique* described in subsection 5.1: let P_m be the smallest set containing B_m and closed under *cut*, i.e., $B_m \subseteq P_m$ and $T \in P_m$ implies $cut(T) \in P_m$. Define weight function ω over P_m as follows: for $T \in P_m$, $\omega(T) = (s, d)$ where s is the number of leaves in T at level $h(T)$ and d is the number of childless nodes in T at level $h(T) - 1$. Let $P_m(s, d)$ denote the set of trees in P_m of weight (s, d). Note that the set of B-trees on n leaves is exactly the set $P_m(n, 0)$.

Lemma 2 *The mapping cut is a regular reduction for (P_m, ω) whose type is given by*

$$t(1,0) = \sum_{i=2}^{m} 1 \bullet (i,0), \tag{8}$$

$$t(s,d) = \sum_{i=\lceil m/2 \rceil}^{m} 1 \bullet (s+i, d-1) \text{ if } d > 0, \tag{9}$$

$$t(s,0) = \sum_{i=\lceil m/2 \rceil}^{m} 1 \bullet (i, s-1) \text{ if } s > 1. \tag{10}$$

Proof. The mapping *cut* trivially satisfies the finiteness condition of theorem 2. We show that it satisfies the regularity condition as well by proving that its type is indeed as given above. The trees in P_m mapped to the single node tree with weight $(1, 0)$ are exactly the trees of height 1 with i children and weight $(i, 0)$ where $2 \leq i \leq m$, hence equation 8. Now consider a tree $T \in P_m(s, d)$ with $d > 0$. The coimages of T are obtained by adding between $\lceil m/2 \rceil$ and m children to the leftmost childless node at level $h(T) - 1$, thus yielding equation 9. Finally, let $T \in P_m(s, 0)$ with $s > 1$. The coimages of T are exactly the trees obtained by adding between $\lceil m/2 \rceil$ and m children to the leftmost leaf in T, hence equation 10. □

Let $b(s, d) = |P_m(s, d)|$ (for fixed m). From the type equations 8–10 we derive the following recurrence relations for $b(s, d)$:

$$\begin{aligned}
b(s,d) &= \sum_{i=\lceil m/2 \rceil}^{m} b(s-i, d+1) \text{ for } s > m, \\
b(m,d) &= \begin{cases} b(d+1,0) + b(m/2, d+1) & \text{if } m \text{ even} \\ b(d+1,0) & \text{if } m \text{ odd} \end{cases} \\
b(s,d) &= b(d+1,0) \text{ for } \lceil m/2 \rceil \leq s < m, \\
b(s,0) &= 1 \text{ for } 1 \leq s < \lceil m/2 \rceil.
\end{aligned}$$

In the asymptotic bounds given below the constant implied by 'O' is independent of both n and m. Let $W = \{(s, d) : 1 \leq s \leq n, 0 \leq d \leq n\}$. We precompute the quantities $b(s, d)$ for $(s, d) \in W$. This takes time and space $O(n^2)$ by using the above recurrences as well as the observation that $b(s+1, d) - b(s, d) = b(s - \lceil m/2 \rceil + 1, d+1) - b(s-m, d+1)$ for $s > m$.

Before we describe the ordering procedures we need to define a linear order $<_{(s,d)}$ on $P_m(s, d)$. We proceed as outlined in subsection 5.3. We let $<_W$ be the lexicographic order on the weights of trees in P_m. Since there is at most one tree of a given weight mapping to a given tree in P_m, we set $index(T) = 0$ for each $T \in P_m$. The linear order on P_m is now given by definition 4 (subsection 5.3).

The following ranking and unranking procedures are straightforward adaptations of their counterparts in subsection 5.3.

function rank (T: tree in P_m): integer;
if T has a single node then $rank := 0$
else
    compute $g((s,d),(s',d'))$;
    $(s,d) := \omega(T)$;
    $(s',d') := \omega(cut(T))$;
    $return(g((s,d),(s',d')) + rank(cut(T)))$;
endif;

In the formulation of the procedure *unrank*, $obj(T',(s,d))$ denotes the unique tree in $P_m(s,d)$ that *cut* maps to T'.

function unrank ((s,d):weight; r: integer): tree in P_m;
if $(s,d) = (1,0)$ then $return$(single node tree)
else
    compute $g((s,d),(s',d'))$ for relevant (s',d');
    $(s',d') :=$ largest weight such that $g((s,d),(s',d')) \leq r$;
    $r' := r - g((s,d),(s',d')$;
    $T' := unrank((s',d'),r')$;
    $return(obj(T',(s,d)))$;
endif;

Theorem 6 *Procedures rank and unrank run in time $O(n)$ (independent of m).*

Proof. The number of recursive calls is $O(n)$ for *rank* and *unrank*. The time spent in the body of *rank* is proportional to the number of nodes removed by *cut*. Similarly, the time spent in the body of *unrank* is proportional to $n(obj(T',(s,d))) - n(T')$. Thus the total time spent by *rank* and *unrank* is proportional to the number of nodes in the input (output) tree, i.e., it is $O(n)$ as claimed (independent of m). □

7 Concluding Remarks

We consider the concept of a regular reduction to be the main contribution of this paper. We have shown how regular reductions may be used in a systematic solution of the ordering problem. We have illustrated this by providing improved algorithms for ordering height-balanced trees, B-trees and binary trees of bounded height. Our algorithms are based on simple regular reductions and have the same simple structure. It would be worthwhile studying applications of regular reductions to ordering other families of trees and other combinatorial objects.

We believe that regular reductions have applications beyond the solution of the ordering problem. One obvious application is the derivation of recurrence relations. A less obvious application concerns the problem of establishing bijective correspondences between different combinatorial families. This application is based on the observation that the type of a regular reduction completely characterizes a combinatorial family and that two families that have the same type equation (up to boundary conditions) can be put into a bijective correspondence. This will be the subject of further research.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The design and analysis of computer algorithms,* Addison-Wesley, 1974.

[2] G.M. Adel'son-Vel'skii and Y.M. Landis, An algorithm for the organization of information, Doklad. Akad. Nauk SSSR, 146 (1962), pp. 263-266; Soviet math. Dokl., 3 (1962), pp. 1259-1262.

[3] C. Berge, Principles of combinatorics, Academic Press, New York, 1971.

[4] R. Bayer and E. McCreight, Organization and maintenance of large ordered indexes, Acta Inform. 1 (1972), pp. 173-189.

[5] U.I. Gupta, D.T. Lee and C.K. Wong, Ranking and unranking of B-trees, Journal of Algorithms 4(1983), pp. 51-60.

[6] U.I. Gupta, D.T. Lee and C.K. Wong, Ranking and unranking of 2-3 trees, SIAM J. Comput. 11(1982), pp. 582-590.

[7] P. Kelsen, Ranking and unranking of trees using regular reductions, M.S. Thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1989.

[8] D.E. Knuth, The art of computer programming, vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.

[9] L. Li, Ranking and unranking of AVL-trees, SIAM J. Comput. 15 (1986), pp. 1025-1035.

[10] C.C Lee, D.T. Lee and C.K. Wong, Generating binary trees of bounded height, Acta Informatica 23, 529-544 (1986).

[11] D. Roelants van Baronaigien and F. Ruskey, Generating t-ary Trees in A-Order, IPL 27 (1988), pp. 205-213.

[12] F. Ruskey, Generating t-ary Trees Lexicographically, SIAM J. Comput., Vol. 7, No. 4, 1978, pp. 424-439.

[13] F. Ruskey and T.C. Hu, Generating binary trees lexicographically, SIAM J. Comput. 6 (1977), pp. 745-758.

[14] F. Ruskey and D. Roelants van Baronaigien, Fast recursive algorithms for generating combinatorial objects, Congressus Numerantium, vol. 41 (1984), pp. 53-62.

[15] H. Wilf, A unified setting for sequencing , ranking and selection algorithms for combinatorial objects, Advances in Mathematics, 24 (1977), pp.281-291.

[16] S.G. Williamson, On the ordering, ranking and random generation of basic combinatorial sets, Lecture Notes in Mathematics, 579, Springer-Verlag, Berlin, 1976.

[17] S. Zaks, Lexicographic generation of ordered trees, Theoretical Computer Science 10 (1980), pp. 63-82.

[18] S. Zaks, Generating trees and other combinatorial objects lexicographically, SIAM J. Comput. 8 (1979), pp. 73-81.