CHAPTER 9

Rooted Plane Trees

Introduction

The most important recursive definition in computer science may be the definition of an *rooted plane tree* (RP-tree) given in Section 5.4. For convenience and emphasis, here it is again.

> Definition 9.1 Rooted plane trees *An* **RP-tree** *consists of a set of vertices each of which has a (possibly empty) linearly ordered list of vertices associated with it called its* **sons** *or* **children**. *Exactly one of the vertices of the tree is called the* **root**. *Among all such possibilities, only those produced in the following manner are RP-trees.*
>
> - *A single vertex with no sons is an RP-tree. That vertex is the root.*
> - *If $T_1, \ldots, T_k$ is an ordered list of RP-trees with roots $r_1, \ldots, r_k$ and no vertices in common, then an RP-tree T can be constructed by choosing an unused vertex r to be the root, letting its ith child be r_i and forgetting that $r_1, \ldots, r_k$ were called roots.*

In Example 7.9 (p. 206) we sketched a proof that this definition agrees with the one given in Definition 5.12. Figure 9.1 illustrates the last stage in building an RP-tree by using our new recursive definition. In that case $k = 3$.

We need a little more terminology. The RP-trees $T_1, \ldots, T_k$ in the definition are called the *principal subtrees* of T. A vertex with no sons is called a *leaf.*

The fact that a general RP-tree can be defined recursively opens up the possibility of recursive algorithms for manipulating general classes of RP-trees. Of course, we are frequently interested in special classes of RP-trees. Those special classes which can be defined recursively are often the most powerful and elegant. Here are three such classes.

- In Example 7.14 (p. 213) we saw how a local description could be associated with a recursive algorithm. In Example 7.16 (p. 214) a local description was expanded into a tree for the Tower of Hanoi procedure. These local descriptions are simply recursive descriptions of RP-trees that describe the algorithms. A leaf is the "output" of the algorithm. In these two examples, a leaf is either a permutation or the movement of a single washer, respectively. In other words:

  A recursive algorithm
  contains
  a local description
  which is
  a recursive definition of some RP-trees.

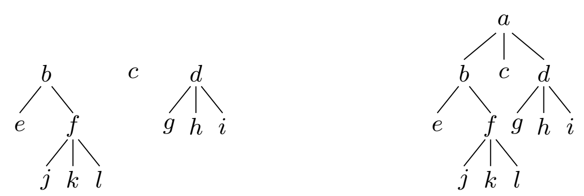


Figure 9.1 The last stage in recursively building an RP-tree. Left: The trees T_1, T_2 and T_3 with roots b, c and d. Right: The new tree T with root a.

In Section 9.1, we'll look at some recursive algorithms for traversing RP-trees.

- Compilers are an important aspect of computer science. Associated with a statement in a language is a "parse tree." This is an RP-tree in which the leaves are the parts of the language that you actually see and the other vertices are grammatical terms. "Context free" grammars are recursively defined and lead to recursively defined parse trees. In Section 9.2 we'll briefly look at some simple parse trees.

- Those RP-trees in which each vertex has either zero or two sons are called *full binary RP-trees*. We'll study them in Section 9.3, with emphasis on ranking and unranking them. (Ranking and unranking were studied in Section 3.2.) Since the trees are defined recursively, so is their rank function.

Except for a reference in Example 9.12 (p. 263), *the sections of this chapter can be read independently of one another.*

9.1 Traversing Trees

A *tree traversal algorithm* is a systematic method for visiting all the vertices in an RP-tree. We've already seen a nonrecursive traversal algorithm in Theorem 3.5 (p. 85). As we shall soon see a recursive description is much simpler. It is essentially a local description.

Traversal algorithms fall into two categories called "breadth first" and "depth first," with depth first being the more common type. After explaining the categories, we'll focus on depth first algorithms.

The left side of Figure 9.2 shows an RP-tree. Consider the right side of Figure 9.2. There we see the same RP-tree. Take your pencil and, starting at the root a, follow the arrows in such a way that you visit the vertices in the order

$$a\ b\ e\ b\ f\ j\ f\ k\ f\ l\ f\ b\ a\ c\ a\ d\ g\ d\ h\ d\ i\ d\ a. \qquad 9.1$$

This manner of traversing the tree diagram, which extends in an obvious manner to any RP-tree T, is called a *depth-first traversal* of the ordered rooted tree T. The sequence of vertices (9.1) associated with the depth-first traversal of the RP-tree T will be called the *depth-first vertex sequence* of T and will be denoted by $\mathrm{DFV}(T)$. If you do a depth-first traversal of an RP-tree T and list the edges encountered (list an edge each time your pencil passes its midpoint in the diagram), you obtain the *depth-first edge sequence* of T, denoted by $\mathrm{DFE}(T)$. In Figure 9.2, the sequence $\mathrm{DFE}(T)$ is

$$\{a,b\}\ \{b,e\}\ \{b,e\}\ \{b,f\}\ \{f,j\}\ \{f,j\}\ \{f,k\}\ \{f,k\}\ \{f,l\}\ \{f,l\}\ \{b,f\}$$
$$\{a,b\}\ \{a,c\}\ \{a,c\}\ \{a,d\}\ \{d,g\}\ \{d,g\}\ \{d,h\}\ \{d,h\}\ \{d,i\}\ \{d,i\}\ \{a,d\}.$$

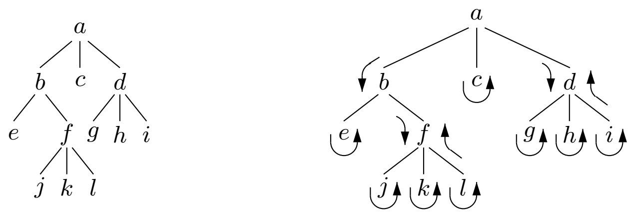


Figure 9.2 Left: An RP-tree with root a. Right: Arrows show depth first traversal of the tree.

The other important linear order associated with RP-trees is called *breadth-first order*. This order is obtained, in the case of Figure 9.2, by reading the vertices or edges level by level, starting with the root. In the case of vertices, we obtain the *breadth-first vertex sequence* (BFV(T)). In Figure 9.2, BFV(T) $= abcdefghijkl$. Similarly, we can define the *breadth-first edge sequence* (BFE(T)).

Although we have defined these orders for trees, the ideas can be extended to other graphs. For example, one can use a breadth first search to find the shortest (least number of choices) route out of a maze: Construct a decision tree in which each vertex corresponds to an intersection in the maze. (More than one vertex may correspond to the same intersection.) A vertex corresponding to an intersection already encountered in the breadth first search has no sons. The decisions at an intersection not previously encountered are all possibilities of the form "follow a passage to the next intersection."

Example 9.1 Data structures for tree traversals Depth-first and breadth-first traversals have data structures naturally associated with their computer implementations.

BFV(T) can be implemented by using a queue. A *queue* is a list from which items are removed at the opposite end from which they are added (first in, first out). Checkout lines at markets are queues. The root of the tree is listed and placed on the queue. As long as the queue is not empty, remove the next vertex from the queue and place its sons on the queue. You should be able to modify this to give BFE(T)

DFV(T) can be implemented by using a stack. A *stack* is a list from which items are removed at the same end to which they are added (last in, first out). They are used in computer programming to implement recursive code. (See Example 7.17 (p. 216).) For DFV, the root of the tree is listed and placed on the stack. As long as the stack is not empty, remove the vertex that is on the top of the stack from the stack and place its sons, in order, on the stack so that leftmost son is on the top of the stack.

When a vertex is added to or removed from the data structure, you may want to take such action; otherwise, you have will have traversed the tree without accomplishing anything.

Depth First Traversals

In a tree traversal, we often want to process either the vertices or edges and do so either the first or last time we encounter them. If you process something only the first time it is encountered, this is a *preorder traversal*; if you list it only the last time it is encountered, this is a *postorder traversal*; This leads to four concepts:

PREV(T)	*preorder vertex sequence;*
POSTV(T)	*postorder vertex sequence;*
PREE(T)	*preorder edge sequence;*
POSTE(T)	*postorder edge sequence.*

Here's the promised recursive algorithm for depth first traversal of a tree. The sequences PREV, POSTV, PREE and POSTE are initialized to empty. They are "global variables," so all levels of the recursive call are working with the same four sequences.

```
Procedure DFT(T)
    Let r be the root of T
    Append vertex r to PREV                 /* PREV */
    Let k be the number of principal subtrees of T
    /* By convention, the For loop is skipped if k = 0. */
    For i = 1, 2, ..., k
        Let T_i be the ith principal subtree of T
        Let r_i be the root of T_i
        Append edge {r, r_i} to PREE        /* PREE */
        DFT(T_i)
        Append edge {r, r_i} to POSTE       /* POSTE */
        Append vertex r to POSTV            /* POSTV */
    End for
    Return
End
```

For example, in Figure 9.2, PREV$(T) = abefjklcdghi$ and POSTV$(T) = ejklfbcghida$. Our pseudocode is easily modified to do these traversals: Simply cross out those "`List`" lines whose comments refer to traversals you don't want.

When a tree is being traversed, the programmer normally does not want a list of the vertices or edges. Instead, he wants to take some sort of action. Thus "`Append vertex` v …" and "`Append edge` $\{u, v\}$ …" would probably be replaced by something like "`DoVERTEX`(v)" and "`DoEDGE`(u, v)," respectively.

Example 9.2 Reconstructing trees Does a sequence like PREV(T) have enough information to reconstruct the tree T from the sequence? Of course, one might replace PREV with other possibilities such as POSTV.

The answer is "no". One way to show this is by finding two trees T and U with PREV$(T) =$ PREV(U) — or using whatever other possibility one wants in place of PREV. The trouble with this approach is that we need a new example for each case. We'll use the Pigeonhole Principle (p. 55) to give a proof that is easily adapted to other situations. Given a set V of n labels for the vertices, there are $n!$ possible sequences for PREV since each such sequence must be a permutation of V. Let B be the set of these permutations and let A be the set of RP-trees with vertex set V. Then PREV : $A \to B$ and we want to show that two elements of A map to the same permutation. By the Pigeonhole Principle, it suffices to show that $|A| > |B|$. We already know that $|B| = n!$ There are n^{n-2} trees by Example 5.10 (p. 143). Since many RP-trees may give the same tree (root and order information is removed), we have $|A| \geq n^{n-2}$. We need to show that $n! < n^{n-2}$ for some value of n. This can be done by finding such an n or by using Stirling's Formula, Theorem 1.5 (p. 12). We leave this to you. □

Example 9.3 Graph traversal and spanning trees One can do depth first traversal of a graph to construct a lineal spanning tree a concept defined in Definition 6.2 (p. 153). The following algorithm finds a lineal spanning tree with root r for a connected simple graph $G = (V, E)$. The graph is a "global variable," so changes to it affect all the levels of the recursive calls. When a vertex is removed from G, so are the incident edges. The comments refer to the proof of Theorem 6.2 (p. 153). We leave it to you to prove that the algorithm does follow the proof as claimed in the comments.

```
/* Generate a lineal spanning tree of G rooted at r. */
LST(r)
     If (no edges contain r)
          Remove r from G
          Return the 1 vertex tree with root r
     End if
     /* f = {r,s} is as in the proof. */
     Choose {r,s} ∈ E
     Remove r from G, saving it and its incident edges
     /* S corresponds to T(A) in the proof. */
     S = LST(s)
     Restore r and the saved edges whose ends are still in G
     /* R corresponds to T(B) in the proof. */
     R = LST(r)
     Join S to R by an edge {r,s} to obtain a new tree T
     Remove r from G
     Return T
End
```
□

Example 9.4 Counting RP-trees When doing a depth first traversal of an unlabeled RP-tree, imagine listing the direction of each step: a for away from the root and t for toward the root. What can we see in this sequence of a's and t's?

Each edge contributes one a and one t since it is traversed in each direction once. If the tree has n edges, we get a $2n$-long sequence containing n copies of a and n copies of t, say $s_1, \ldots, s_{2n}$. If $s_1, \ldots, s_k$ contains d_k more a's than t's, we will be a distance d_k from the root after k steps because we've taken d more steps away from the root than toward it. In particular $d_k \geq 0$ for all k.

Thus a tree with n edges determines a unique sequence of n a's and n t's in which each initial part of the sequence contains at least as many a's as t's. Furthermore, you should be able to see how to reconstruct the tree if you are given such a sequence. Thus there is a bijection between the trees and the sequences. It follows from the first paragraph of Example 1.13 (p. 15) that the number of n-edge unlabeled RP-trees is C_n, the Catalan number.

Since a tree with n vertices has $n - 1$ edges, it follows that the number of n-vertex unlabeled RP-trees is C_{n-1}. By Exercise 9.3.12 (p. 266), it follows that the number of unlabeled binary RP-trees with n leaves is C_{n-1}. Thus the solution to Exercise 9.3.13 provides a formula for the Catalan numbers. □

Exercises

9.1.1. Write pseudocode for recursive algorithms for PREV(T), PREE(T) and POSTE(T).

9.1.2. In the case of decision trees we are only interested in visiting the leaves of the tree. Call the resulting sequence DFL(T) Write pseudocode for a recursive algorithm for DFL(T). What is the connection with the traversal algorithm in Theorem 3.5 (p. 85)?

9.1.3. Write pseudocode to implement breadth first traversal using a queue. Call the operations for adding to the queue and removing from the queue `INQUEUE` and `OUTQUEUE`, respectively.

9.1.4. Write pseudocode to implement PREV(T) using a stack. Call the operations for adding to the stack and removing from the stack `PUSH` and `POP`, respectively.

9.1.5. Construct a careful proof that the algorithm in Example 9.3 does indeed construct a lineal spanning tree.

9.1.6. In contrast to Example 9.2, if PREV(T) = PREV(U) *and* POSTV(T) = POSTV(U), then $T = U$. Another way to state this is:

$$\text{Given PREV}(T) \text{ and POSTV}(T)\text{, we can reconstruct } T. \tag{9.2}$$

The goal of this exercise is to prove (9.2) by induction on the number of vertices.

Let $n = |T|$, the number of vertices of T. Let v be the root of T. Let PREV(T) $= a_1, \ldots, a_n$ and POSTV(T) $= z_1, \ldots, z_n$.

If $n > 1$, we may suppose that T consists of the RP-trees $T_1, \ldots, T_k$ joined to the root v. Let U be the RP-tree with root v joined to the trees $T_2, \ldots, T_k$. (If $k = 1$, U is just the single vertex v.)

(a) Prove (9.2) is true when $n = 1$.

(b) Prove that a_2 is the root of T_1

(c) Suppose $z_t = a_2$. (This must be true for some t since PREV and POSTV are both permutations of the vertex set.) Prove that POSTV(T_1) $= z_1, \ldots, z_t$.

(d) With t as above, prove that PREV(T_1) $= a_2, \ldots, a_{t+1}$.
Hint. How many vertices are there in T_1?

(e) Prove that PREV(U) $= v, a_{t+2}, \ldots, a_n$ and POSTV(U) $= z_{t+1}, \ldots, z_n$.

(f) Complete the proof.

9.1.7. In Example 9.2, we proved that a tree cannot be reconstructed from the sequence PREV. The proof also works for POSTV.

(a) Find two different RP-trees T and U with PREV(T) = PREV(U).

(b) Find two different RP-trees T and U with POSTV(T) = POSTV(U).

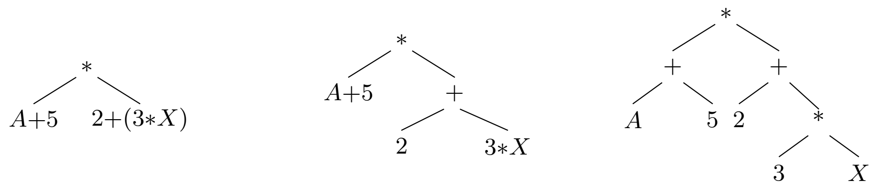
9.1.8. Use Exercise 9.1.6 to write pseudocode for a recursive procedure to reconstruct a tree from PREV and POSTV.

9.1.9. If T is an unlabeled RP-tree, define a sequence $D(T)$ of ±1's as follows. Perform a depth first traversal of the tree. Whenever an edge is followed from father to son, list a $+1$. Whenever an edge is followed from son to father, list a -1.

(a) Let $T_1, \ldots, T_m$ be unlabeled RP-trees and let T be the tree formed by joining the roots of each of the T_i's to a new root to form a new unlabeled RP-tree. Express $D(T)$ in terms of $D(T_1)$, ..., $D(T_m)$.

(b) IF $D(T) = s_1, \ldots, s_n$, show that n is twice the number of edges of T and show that $\sum_{i=1}^k s_i \ge 0$ for all k with equality when $k = n$.

*(c) Let $\vec{s} = s_1, \ldots, s_n$ be a sequence of ±1's. Show that $\vec{s}$ comes from at most one tree and that it comes from a tree if and only if $\sum_{i=1}^k s_i \ge 0$ for all k, with equality when $k = n$.



(a) The first step. (b) An intermediate step. (c) The final result.

Figure 9.3 Interpreting the expression $(A + 5) * (2 + (3 * X))$.

9.2 Grammars and RP-Trees

Languages are important in computer science. Compilers convert programming languages into machine code. Automatic translation projects are entangled by the intricacies of natural languages. Researchers in artificial intelligence are interested in how people manage to understand language. We look briefly at a simple, small part of all this: "context-free grammars" and "parse trees." To provide some background material, we'll look at arithmetic expressions and at simple sentences.

Unfortunately, we'll be introducing quit a bit of terminology. Fortunately, you won't need most of it for later sections, so, if you forget it, you can simply look up what you need in the index at the time it is needed.

Example 9.5 Arithmetic expressions Let's consider the meaning of the expression $(A + 5) * (2 + (3 * X))$.

It means $A + 5$ times $2 + (3 * X)$, which we can represent by the RP-tree in Figure 9.3(a). We can then interpret the subexpressions and replace them by their interpretations and so on until we obtain Figure 9.3(c). We can represent this recursive procedure as follows, where "*exp*" is short for "*expression*".

```
INTERPRET(exp)
     If (exp has no operation)
          Return the RP-tree with root exp
               and no other vertices.
     End if
     Let exp = (exp_l op exp_r).
     Return the RP-tree with root op,
          left principal subtree INTERPRET(exp_l) and
          right principal subtree INTERPRET(exp_r).
End
```

Now suppose we wish to evaluate the expression, with a procedure we'll call `EVALUATE`. One way of doing that would be to modify `INTERPRET` slightly so that it returns values instead of trees. A less obvious method is to traverse the tree generated by `INTERPRET` in POSTV order. Each leaf is replaced by its value and each nonleaf is replaced by the value of performing the operation it indicates on the values of its two sons.

To illustrate this, POSTV of Figure 9.3(c) is $A\ 5\ +\ 2\ 3\ X\ *\ +\ *$. Thus we replace the leaves labeled A and 5 with their values, then the leftmost vertex labeled $+$ with the value of $A + 5$, and so on. POSTV of a tree associated with calculations is called *postorder notation* or *reverse Polish notation.* It is used in some computer languages such as Forth and PostScript® and in some handheld calculators. □

Example 9.6 Very simple English Languages are very complicated creations; however, we can describe a rather simple form of an English sentence as follows:

(a) One type *sentence* consists of the three parts *noun-phrase*, *verb*, *noun-phrase* and a period in that order.

(b) A *noun-phrase* is either a *noun* or an *adjective* followed by a *noun-phrase*.

(c) Lists of acceptable words to use in place of *verb*, *noun* and *adjective* are available.

This description is a gross oversimplification. It could lead to such sentences as "**big brown small houses sees green green boys.**" The disagreement between subject and verb (plural versus singular) could be fixed rather easily, but the nonsense nature of the sentence is not so easily fixed. If we agree to distinguish between grammatical correctness and content, we can avoid this problem. (As we shall see in a little while, this is not merely a way of defining our difficulty out of existence.) □

Let's rephrase our rules from the last example in more concise form. Here's one way to do that.

(a) *sentence* → *noun-phrase verb noun-phrase* **.**

(b) *noun-phrase* → *adjective noun-phrase* | *noun*

(c) *adjective* → **big** | **small** | **green** | ···

noun → **houses** | **boys** | ···

verb → **sees** | ···

These rules along with the fact that we are interested in sentences are what is called a "context-free grammar."

Definition 9.2 Context-free grammar *A* **context-free grammar** *consists of*

1. *a finite set S of* **nonterminals***;*
2. *a finite set T of* **terminals***;*
3. *a* **start symbol** $s_0 \in S$*;*
4. *a finite set of* **productions** *of the form* $s \to x_1 \ldots x_n$ *where* $s \in S$ *and* $x_i \in S \cup T$*. We allow* $n = 0$*, in which case the production is "*$s \to$*."*

*We'll distinguish between terminal and nonterminal symbols by writing them in the fonts "***terminal***" and "nonterminal." If we don't know whether or not an element is terminal, we will use the nonterminal font. (This can happen in a statement like,* $x \in S \cup T$*.)*

In the previous example, the start symbol is *sentence* and the productions are given in (a)–(c).

The productions of the grammar give the structural rules for building the language associated with the grammar. This set of rules is called the *syntax* of the language. The grammar is called context-free because the productions do not depend on the context (surroundings) in which the nonterminal symbol appears. Natural languages are not context-free, but many computer languages are nearly context-free.

Any string of symbols that can be obtained from the start symbol (*sentence* above) by repeated substitutions using the productions is called a sentential form.

Definition 9.3 The language of a grammar *A sentential form consisting only of terminal symbols is a* **sentence***. The set of sentences associated with a grammar G is called the* **language** *of the grammar and is denoted $L(G)$.*

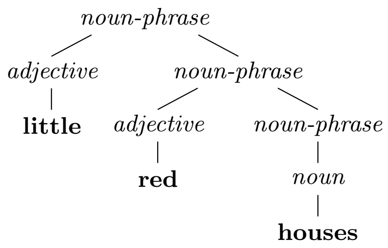


Figure 9.4 The parse tree for the sentence **little red houses**.

Processes that obtain one string of symbols from another by repeated applications of productions are called *derivations*. To indicate that **little red** *noun* can be derived from *noun-phrase*, we write

$$\textit{noun-phrase} \quad \stackrel{*}{\Rightarrow} \quad \textbf{little red } \textit{noun}.$$

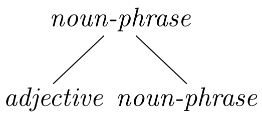
Thus

$$\textit{sentence} \quad \stackrel{*}{\Rightarrow} \quad \textbf{big brown small houses sees green green boys}.$$

We can represent productions by RP-trees where a vertex is the left side of a production and the leaves are the items on the right side; for example,

$$\textit{noun-phrase} \quad \to \quad \textit{adjective noun-phrase}$$

becomes



The collection of productions thus become local descriptions for trees corresponding to derivations such as that shown in Figure 9.4. We call such a tree a *parse tree*. The string that was derived from the root of the parse tree is the DFV sequence of the tree. A sentence corresponds to a parse tree in which the root is the start symbol and all the leaves are terminal symbols.

How is Figure 9.3 related to parse trees? There certainly seems to be some similarity, but all the symbols are terminals. What has happened is that the parse tree has been squeezed down to eliminate unnecessary information. Before we can think of Figure 9.3 as coming from a parse tree, we need to know what the grammar is. Here's a possibility, with *exp* (short for *expression*) the start symbol, **number** standing for any number and **id** standing for any *identifier*; i.e., the name of a variable.

$$\begin{array}{rcl} \textit{exp} & \to & \textit{term} \;\mid\; \textit{term op term} \\ \textit{term} & \to & (\,\textit{exp}\,) \;\mid\; \textbf{id} \;\mid\; \textbf{number} \\ \textit{op} & \to & + \;\mid\; - \;\mid\; * \;\mid\; / \end{array}$$

A computer language has a grammar. The main purpose of a compiler is to translate grammatically correct code into machine code. A secondary purpose is to give the programmer useful messages when nongrammatical material is encountered. Whether grammatically correct statements make sense in the context of what the programmer wishes to do is beyond the ken of a compiler; that is, a compiler is concerned with syntax, not with content.

Compilers must use grammars backwards from the way we've been doing it. Suppose you are functioning as a general purpose compiler. You are given the grammar and a string of terminal symbols. Instead of starting with the start symbol, you must start with the string of terminals and works backwards, creating the parse tree *from* the terminals at the leaves *back to* the start symbol at the root. This is called *parsing*. A good compiler must be able to carry out this process or something like it quite rapidly. For this reason, attention has focused on grammars which are quickly parsed and yet flexible enough to be useful as computer languages.

When concerned with parsing, one should read the productions backwards. For example,

$$\textit{term} \;\rightarrow\; (\ \textit{exp}\ ) \;\mid\; \textbf{id} \;\mid\; \textbf{number}$$

would be read as the three statements:

- If one sees the string "(*exp*)", it may be thought of as a *term*.
- If one sees an **id**, it may be thought of as a *term*.
- If one sees a **number**, it may be thought of as a *term*.

Example 9.7 Arithmetic expressions again Our opening example on arithmetic expressions had a serious deficiency: Parentheses were required to group things. We would like a grammar that would obey the usual rules of mathematics: Multiplication and division take precedence over addition and subtraction and, in the event of a draw, operations are performed from left to right.

We can distinguish between factors, terms (products of factors) and expressions (sums of terms) to enforce the required precedence. We can enforce the left to right rule by one of two methods:

(a) We can build it into the syntax.

(b) We can insist that in the event of an ambiguity the leftmost operation should be performed.

The latter idea has important ramifications that make parsing easier. You'll learn about that when you study compilers. We'll use method (a). Here's the productions, with *exp* the start symbol.

$$\begin{aligned}
\textit{exp} &\;\rightarrow\; \textit{term} \;\mid\; \textit{exp} + \textit{term} \;\mid\; \textit{exp} - \textit{term}\\
\textit{term} &\;\rightarrow\; \textit{factor} \;\mid\; \textit{term} * \textit{factor} \;\mid\; \textit{term} \,/\, \textit{factor}\\
\textit{factor} &\;\rightarrow\; \textbf{(}\ \textit{exp}\ \textbf{)} \;\mid\; \textbf{id} \;\mid\; \textbf{number}
\end{aligned}$$

As you can see, even this a language fragment is a bit complicated. □

By altering (4) of Definition 9.2 (p. 254), we can get other types of grammars. A more general replacement rule is

4′. a finite set of *productions* of the form $v_1 \ldots v_m \rightarrow x_1 \ldots x_n$ where $v_i, x_i \in S \cup T$, $m \geq 1$ and $n \geq 0$.

This gives what are called *phrase structure grammars*. Grammars that are this general are hard to handle.

A much more restrictive replacement rule is

4″. a finite set of *productions* of the form $s \rightarrow \mathbf{t_1} \ldots \mathbf{t_n}$ or $s \rightarrow \mathbf{t_1} \ldots \mathbf{t_n} r$ and where $r,s \in S$, $n \geq 0$ and $\mathbf{t_i} \in T$.

This gives what are called *regular grammars*, which are particularly easy to study.

Example 9.8 Finite automata and regular grammars In Section 6.6 we studied finite automata and, briefly, Turing machines. If $\mathcal{A}$ is a finite automaton, the *language* of $\mathcal{A}$, $L(\mathcal{A})$, is the set of all input sequences that are recognized by $\mathcal{A}$. With a proper definition of recognition for Turing machines, the languages of phrase structure grammars are precisely the languages recognized by Turing machines. We will prove the analogous result for regular grammars:

Theorem 9.1 Regular Grammars *Let L be a set of strings of symbols. There is a regular grammar G with $L(G) = L$ if and only if there is a finite automaton $\mathcal{A}$ with $L = L(\mathcal{A})$. In other words, the languages of regular grammars are precisely the languages recognized by finite automata.*

Proof: Suppose we are given an automaton $\mathcal{A} = (S, I, f, s_0, A)$. We must exhibit a regular grammar G with $L(G) = L(\mathcal{A})$. Here it is.

1. The set of nonterminal symbols of G is S, the set of states of $\mathcal{A}$.
2. The set of terminal symbols of G is I, set of input symbols of $\mathcal{A}$.
3. The start symbol of G is s_0, the start symbol of $\mathcal{A}$.
4. The productions of G are all things of the form $s \to it$ or $s \to j$ where $f(s,i) = t$ or $f(s,j) \in A$, the accepting states of $\mathcal{A}$.

Clearly G is a regular grammar. It is not hard to see that $L(G) = L(\mathcal{A})$.

Now suppose we are given a regular grammar G. We must exhibit an automaton $\mathcal{A}$ with $L(\mathcal{A}) = L(G)$. Our proof is in three steps. First we show that there is a regular grammar G' with $L(G') = L(G)$ and with no productions of the form $s \to t$. Second we show that there is a regular grammar G'' with $L(G'') = L(G')$ in which the right side of all productions are either empty or of the form $\mathbf{i}s$, that is $\mathbf{i}$ is terminal and s is nonterminal. Finally we show that there is a nondeterministic finite automaton $\mathcal{N}$ that recognizes precisely $L(G'')$. By the "no free will" theorem in Section 6.6, this will complete the proof.

First step: Suppose that G contains productions of the form $s \to t$. We will construct a new regular grammar G' with no productions of this form and $L(G) = L(G')$. (If there are no such productions, simply let $G' = G$.) The terminal, nonterminal and start symbols of G' are the same as those of G. Let R be the right side of a production in G that is not simply a nonterminal symbol. We will let $s \to R$ be a production in G' if and only if

- $s \to R$ is a production in G, or
- There exist $x_1, \ldots, x_n$ such that $s \to x_1$, $x_1 \to x_2$, $\cdots$, $x_{n-1} \to x_n$ and $x_n \to R$ are all productions in G.

You should be able to convince yourself that $L(G') = L(G)$.

Second step: We now construct a regular grammar G'' in which the right side of every production is either empty or has exactly one terminal symbol. The terminal symbols of G'' are the same as those of G'. The nonterminal symbols of G'' are those of G' plus some additional ones which we'll define shortly.

Let $s \to \mathbf{i}_1 \ldots \mathbf{i}_n t$ be a production in G'. If $n = 1$, this is also a production in G''. By the construction of G', we cannot have $n = 0$, so we can assume $n > 1$. Let σ stand for the right side of the production. Introduce $n-1$ new states (s, σ, k) for $2 \le k \le n$. Let G'' contain the productions

- $s \to \mathbf{i}_1(s, \sigma, 2)$;
- $(s, \sigma, k) \to \mathbf{i}_k(s, \sigma, k+1)$, for $1 < k < n$ (There are none of these if $n = 2$.);
- $(s, \sigma, n) \to \mathbf{i}_n t$.

Let $s \to \mathbf{i}_1 \ldots \mathbf{i}_n$ be a production in G'. (An empty right hand side corresponds to $n = 0$.) If $n = 0$ or $n = 1$, let this be a production in G''; otherwise, use the idea of the previous paragraph, omitting t. You should convince yourself that $L(G'') = L(G')$.

Third step: We now construct a nondeterministic finite automaton $\mathcal{N}$ that recognizes precisely $L(G)$. The states of $\mathcal{N}$ are the internal symbols of G'' together with a new state a, the start state is the start symbol, and the input symbols are the terminal symbols. Let a be an accepting state of $\mathcal{N}$. Let $s \to R$ be a production of G''. There are three possible forms for R:

- If R is empty, then s is an accepting state of $\mathcal{N}$.
- If $R = \mathbf{i}$, then $(s, \mathbf{i}, a)$ is an edge of $\mathcal{N}$.
- if $R = \mathbf{i}t$, then $(s, \mathbf{i}, t)$ is an edge of $\mathcal{N}$.

You should convince yourself that $\mathcal{N}$ accepts precisely $L(G'')$. $\square$

Exercises

9.2.1. Draw the trees like Figure 9.3(c) to interpret the following expressions.

(a) $(((1+2)+3)+4)+5$

(b) $((1+2)+(3+4))+5$

(c) $1+(2+(3+(4+5)))$

(d) $(X+5*Y)/(X-Y)$

(e) $(X*Y-3)+X*(Y+1)$

9.2.2. How might the ideas in Example 9.5 be modified to allow for unary minus as in $(-A)*B$?

9.2.3. Write pseudocode for the two methods suggested in Example 9.5 for calculating the value of an arithmetic expression.

9.2.4. Using the grammar of Example 9.7, construct parse trees for the following sentences.

(a) $(X+5*Y)/(X-Y)$

(b) $(X*Y-3)+X*(Y+1)$

9.2.5. Add the following features to the expressions of Example 9.7.

(a) *Unary minus.*

(b) *Exponentiation* using the operator $**$. Ambiguities are resolved in the reverse manner from the other arithmetic operations: $\mathbf{A} ** \mathbf{B} ** \mathbf{C}$ is the same as $\mathbf{A} ** (\mathbf{B} ** \mathbf{C})$.

(c) *Replacement* using the operator **:=**. Only one use of the operator is allowed.

(d) *Multiple replacement* as in

$$\mathbf{A} := \mathbf{4} + \mathbf{B} := \mathbf{C} := \mathbf{5} + \mathbf{2} * \mathbf{3},$$

which means that C is set equal to $5+2*3$, B is set to equal to C and A is set equal to $4+B$.

9.2.6. Let G be the grammar with the start state s and the productions

(i) $s \to \mathbf{x}t$ and $s \to \mathbf{y}t$;

(ii) $t \to R$ where R is either empty or one of $+\mathbf{x}t$, $+\mathbf{y}t$ or $-\mathbf{x}t$.

(a) Describe $L(G)$.

(b) Follow the steps in the construction of the corresponding nondeterministic finite automaton; that is, describe G', G'' and $\mathcal{N}$ that were constructed in the proof.

(c) Continuing the previous part, construct a deterministic machine as done in Section 6.6 corresponding to $\mathcal{N}$.

(d) Can you construct a simpler deterministic machine to recognize $L(G)$?

*9.3 Unlabeled Full Binary RP-Trees

We'll begin with a review of material discussed in Examples 7.9 (p. 206) and 7.10. Roughly speaking, an unlabeled RP-tree is an RP-tree with the vertex labels erased. Thus, the order of the sons of a vertex is still important. A tree is "binary" (resp. "full binary") if each nonleaf has at most (resp. exactly) two sons. Figure 9.5 shows some unlabeled full binary RP-trees. Here is a more precise pictorial definition. Compare it to Definition 9.1 for (labeled) RP-trees.

Definition 9.4 Unlabeled binary rooted plane trees *The following are* **unlabeled binary RP-trees**. *Roots are indicated by $\bullet$ and other vertices by $\circ$.*

(i) *The single vertex $\bullet$ is such a tree.*

(ii) *If T_1 is one such tree, so is the tree formed by (a) drawing T_1 root upward, (b) adding a $\bullet$ above T_1 and connecting $\bullet$ to the root of T_1, and (c) changing the root of T_1 to $\circ$.*

(iii) *If T_1 and T_2 are two such trees, so is the tree formed by (a) drawing T_1 to the left of T_2, both root upward, (b) adding a $\bullet$ above them and connecting it to their roots, and (c) changing the roots of T_1 and T_2 to $\circ$'s.*

If we omit (ii), the result is **unlabeled full binary RP-trees**.

These trees are often referred to as "unlabeled ordered (full) binary trees." Why? To define a binary tree, one needs to have a root. Drawing a tree in the plane is equivalent to ordering the children of each vertex. Sometimes the adjective "full" is omitted. In this section, we'll study unlabeled ordered full binary trees.

We can build all unlabeled full binary RP-trees recursively by applying the definition over and over. To begin with there are no trees, so all we get is a single vertex by (1) of the definition. This tree can then be used to build the 3 vertex full binary RP-tree shown in the next step of Figure 9.5. Using the two trees now available, we can build the three new trees shown in the right hand step of Figure 9.5. In general, if we have a total of t_n trees at step n, then $t_1 = 1$ (the single vertex) and $t_{n+1} = 1 + (t_n)^2 + 1$ (either use the single vertex tree or join two trees T_1 and T_2 to a new root).

Example 9.9 Counting and listing unlabeled full binary RP-trees How many unlabeled full binary RP-trees are there with n leaves? How can we list them?

As we shall see, answers to these questions come almost immediately from the recursive definition. It is important to note that

Definition 9.4 provides *exactly one way* to produce every unlabeled full binary RP-tree.

If there were more than one way to produce some of the trees from the definition, we would not be able to obtain answers to our questions so easily, if at all.

We begin with counting. Let b_n be the desired number. Clearly $b_0 = 0$, since a tree has at least one leaf. Let's look at how our definition leads to trees with n leaves.

According to the definition, an unlabeled full binary RP-tree will be either a single vertex, which contributes to b_1, or it will have exactly two principal subtrees, both of which are unlabeled full binary RP-trees. If the first of these has k leaves, then the second must have $n-k$. By the Rules of Sum and Product,

$$b_n = \sum_{k=1}^{n-1} b_k b_{n-k} \qquad \text{if } n > 1. \tag{9.3}$$

Using this we can calculate the first few values fairly easily:

$$b_1 = 1 \qquad b_2 = 1 \qquad b_3 = 2 \qquad b_4 = 5 \qquad b_5 = 14 \qquad b_6 = 42 \qquad b_7 = 132.$$

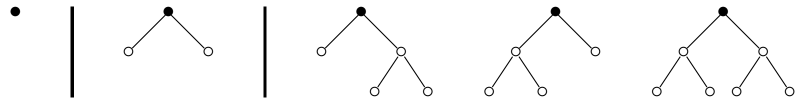


Figure 9.5 The first three stages in building unlabeled full binary RP-trees recursively. A ∘ is a vertex of a previously constructed tree and a • is the root of a new tree.

Notice how the recursion came almost immediately from the definition.

So far, this has all been essentially a review of material in Examples 7.9 and 7.10. Now we'll look at something new: listing the trees based on the recursive description. Here's some pseudocode to list all binary RP-trees with n leaves.

```
/* Make a list of n-leaf unlabeled full binary RP-trees */
Procedure BRPT(n)
     If (n = 1), then Return the single vertex tree
     Set List empty
     For k = 1, 2, ···, n − 1:
          /* Get a list of first principal subtrees. */
          S_L = BRPT(k)
          /* Get a list of second principal subtrees. */
          S_R = BRPT(n − k)
          For each T_1 ∈ S_L:
              For each T_2 ∈ S_R:
                   Add JOIN(T_1, T_2) to List
              End for
          End for
     End for
     Return List
End
```

The procedure JOIN(T_1, T_2) creates a full binary RP-tree with principal subtrees T_1 and T_2. The outer `for` loop is running through the terms in the summation in (9.3). The inner `for` loop is constructing each of the trees that contribute to the product $b_k b_{n-k}$. This parallel between the code and (9.3) is perfectly natural: They both came from the same recursive description of how to construct unlabeled full binary RP-trees. □

What happened in this example is typical. Given a recursive description of how to *uniquely* construct all objects in some set, we can provide both a recursion for the number of them and pseudocode to list all of them. It is not so obvious that such a description usually leads to ranking and unranking algorithms as well. Rather than attempt a theoretical explanation of how to do this, we'll look at examples. Since it's probably not fresh in your mind, it would be a good idea to review the concepts of ranking and unranking from Section 3.2 (p. 75) at this time.

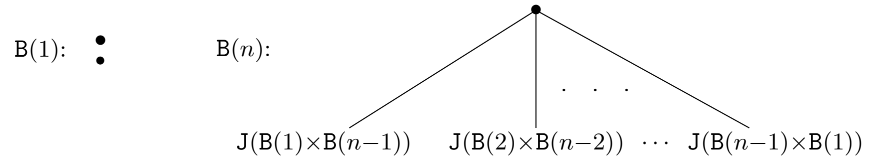


Figure 9.6 The local description for n-leaf unlabeled full binary RP-trees. We assume that $n > 1$ in the right hand figure. `B` stands for `BRPT` and `J(C,D)` $= \{\texttt{JOIN}(c,d) \mid c \in \texttt{C},\ d \in \texttt{D}\}$, with the set made into an ordered list using lexicographic order based on the orderings in `C` and `D` obtained by lex ordering all the pairs $(\text{RANK}(c), \text{RANK}(d))$, with $c \in \texttt{C}$ and $d \in \texttt{D}$.

Example 9.10 A ranking for permutations We'll start with permutations since they're fairly simple.

Suppose that S is an n-set with elements $s_1 < \cdots < s_n$. The local description of how to generate $L(S)$, the permutations of S listed in lex order, is given in Figure 7.2 (p. 213). This can be converted to a verbal recursive description: Go through the elements $s_i \in S$ in order. For each s_i, list all the permutations of $S - \{s_i\}$ in lex order, each preceded by "s_i,". (The comma after s_i is not a misprint.)

How does this lead to RANK and UNRANK functions? Let $\sigma\colon \underline{n} \to \underline{n}$ be a permutation and let $\text{RANK}(s_{\sigma(1)}, \ldots, s_{\sigma(n)})$ denote the rank of $s_{\sigma(1)}, \ldots, s_{\sigma(n)}$. Since the description is recursive, the rank formula will be as well. We need to start with $n = 1$. Since there is only one permutation of a one-element set, $\text{RANK}(\sigma) = 0$.

Now suppose $n > 1$. There are $\sigma(1) - 1$ principal subtrees of $L(S)$ to the left of the subtree

$$s_{\sigma(1)}, L(S - \{s_{\sigma(1)}\}),$$

each of which has $(n-1)!$ leaves. Thus we have

$$\text{RANK}(s_{\sigma(1)}, \ldots, s_{\sigma(n)}) \;=\; (\sigma(1)-1)(n-1)! + \text{RANK}(s_{\sigma(2)}, \ldots, s_{\sigma(n)}).$$

You should also be able to see this by looking at Figure xrefLexOrderLocal.

As usual the rank formula can be "reversed" to do the unranking: Let $\text{UNRANK}(r, S)$ denote the permutation of the set S that has rank r. Let $q = r/(n-1)!$ with the remainder discarded. Then

$$\text{UNRANK}(r, S) \;=\; s_{q+1},\ \text{UNRANK}\big(r - (n-1)!q,\ S - \{s_{q+1}\}\big). \quad \square$$

Example 9.11 A ranking for unlabeled full binary RP-trees How can we rank and unrank `BRPT`(n), the n-leaf unlabeled full binary RP-trees?

Either Definition 9.1 or the listing algorithm `BRPT` that we obtained from it in Example 9.9 can be used as a starting point. Each gives a local description of a decision tree for generating n-leaf unlabeled full binary RP-trees. The only thing that is missing is an ordering of the sons in the decision tree, which can be done in any convenient manner. The listing algorithm provides a specific order for listing the trees, so we'll use it. It's something like lex order:

- first by size of the left tree (the outer loop on k),
- then by the rank of the left tree (the middle loop on $T_1 \in S_L$), 9.4
- finally by the rank of the right tree (the inner loop on $T_2 \in S_R$).

The left part of the figure is not a misprint: the top • is the decision tree and the bottom • is it's label: the 1-leaf tree. Carrying this a bit further, Figure 9.7 expands to local description for 2 and 3 leaves.

Expanding this local description for any value of n would give the complete decision tree in a nonrecursive manner. However, we have learned that expanding recursive descriptions is usually unnecessary and often confusing. In fact, we can obtain ranking and unranking algorithms directly from the local description, as we did for permutations in the preceding example.

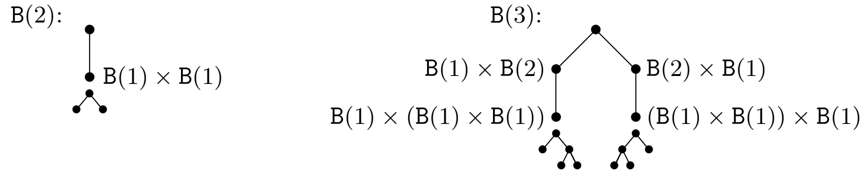


Figure 9.7 An expansion of Figure 9.6 for $n = 2$ and $n = 3$. The upper trees are the expansion of Figure 9.6. The lower trees are the full binary RP-trees that occur at the leaves.

Let's get a formula for the rank. Since our algorithm for listing is recursive, our rank formula will also be recursive. We must start our recursive formula with the smallest case, $|T| = 1$. In this case there is only one tree, namely a single vertex. Thus $T = \bullet$ and $\mathrm{RANK}(T) = 0$.

Now suppose $|T| > 1$. and let T_1 and T_2 be its first and second principal subtrees. (Or left and right, if you prefer.) We need to know which trees come before T in the ranking. Suppose Q has principal subtrees Q_1 and Q_2 and $|Q| = |T|$. The information in (9.4) says that the tree T is preceded by precisely those trees Q for which $|Q| = |T|$, and either

- $|Q_1| < |T_1|$ OR
- $|Q_1| = |T_1|$ AND $\mathrm{RANK}(Q_1) < \mathrm{RANK}(T_1)$ OR
- $Q_1 = T_1$ AND $\mathrm{RANK}(Q_2) < \mathrm{RANK}(T_2)$.

The number of trees in each of these categories is

- $\displaystyle\sum_{k<|T_1|} b_k b_{n-k}$, where terms were collected by $k = |Q_1|$,
- $\mathrm{RANK}(T_1) \times b_{|T_2|}$, and
- $1 \times \mathrm{RANK}(T_2)$.

Hence

Theorem 9.2 Rank of Unlabeled Full Binary RP-Trees *The rank of an unlabeled full binary RP-tree with n leaves is 0 if $n = 1$ and otherwise is*

$$\mathrm{RANK}(T) \;=\; \sum_{k<|T_1|} b_k b_{n-k} + \mathrm{RANK}(T_1) b_{|T_2|} + \mathrm{RANK}(T_2), \qquad 9.5$$

where T_1 and T_2 are the first and second principal subtrees of T and b_k is number of k-leaf unlabeled full binary RP-trees.

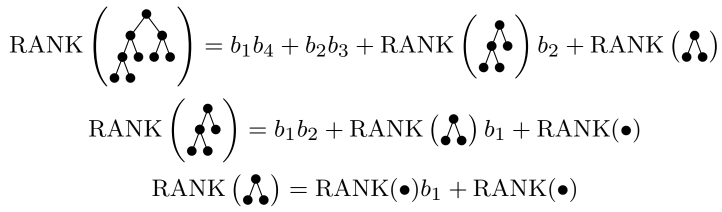


Figure 9.8 A recursive rank calculation for an unlabeled full binary RP-tree with 5 leaves.

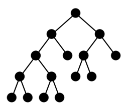


Figure 9.9 The 8-leaf unlabeled full binary RP-tree of rank 250.

Figure 9.8 shows how (9.5) is used to compute rank. Each of the equations given there is a special case of (9.5). The first equation gives the rank of the tree we are interested in. The other two equations give ranks that are needed because of the recursive nature of (9.5). One can now work from the bottom up using $\mathrm{RANK}(\bullet) = 0$ to get ranks of 0, 1 and 8, respectively.

As always, unranking uses a greedy algorithm. Let $\mathrm{UNRANK}(R, n)$ denote the n-leaf full binary RP-tree with rank R. Let's compute $\mathrm{UNRANK}(250, 8)$, the 8-leaf full binary RP-tree with rank 250. Being greedy, we want T_1, the left principal tree to have as many leaves as possible. We have

$$b_1b_7 + b_2b_6 + b_3b_5 + b_4b_4 \;=\; 227$$

and

$$b_1b_7 + b_2b_6 + b_3b_5 + b_4b_4 + b_5b_3 \;=\; 227 + 28 > 250.$$

Thus the first principal tree, T_1, has five leaves and the second, T_2, has three. Now we want $\mathrm{RANK}(T_1)$ to be as large as possible. Since $\mathrm{RANK}(T_1)b_3 + \mathrm{RANK}(T_2) = 250 - 227 = 23$ and $23/b_3 = 23/2 = 11.5$, T_1 has rank 11 and T_2 has rank $23 - 11b_3 = 1$. Thus $T_1 = \mathrm{UNRANK}(11, 5)$ and $T_2 = \mathrm{UNRANK}(1, 3)$. We'll compute T_2 first. We have $b_1b_2 = 1$ and $b_2b_1 = 1$, so the first principal subtree of T_2 has two leaves and the second has one. Since there is only one 2-leaf tree and only one 1-leaf, we are done with T_2. Since $b_1b_4 + b_2b_3 + b_3b_2 = 9$, the first principal subtree of T_1 has four leaves and rank 2 while the second has one leaf. Since $b_1b_3 = 2$, both principal subtrees of this 4-leaf tree have two leaves. Putting this all together, we get the tree shown in Figure 9.9. □

Example 9.12 Computing the rank without recursion In Example 9.11 (p. 261) we proved the recursive formula (9.5) for the rank of an unlabeled full binary RP-tree. This formula can be implemented as it stands by a recursive computer program; however, recursive procedures can be inconvenient for hand calculations. We can implement the formula by a depth first postorder vertex traversal of the tree that we want to rank.

The last time we visit a vertex, we simply record the rank and number of leaves in the subtree which has that vertex as its root. If we are at a leaf, the rank is 0 and the number of leaves is 1. Suppose we have reached some tree T that is not a leaf. In the notation of (9.5) we have available $\mathrm{RANK}(T_1)$, $|T_1|$, $\mathrm{RANK}(T_2)$ and $|T_2|$. From this we can compute $\mathrm{RANK}(T)$ using (9.5) because $n = |T_1| + |T_2|$. The values of the b_i's can be computed ahead of time and written in a table. Figure 9.10 applies this idea to the tree in Figure 9.9. Rather than work in depth first postorder, we can simply do the vertices depth by depth, starting at the lowest depth. □

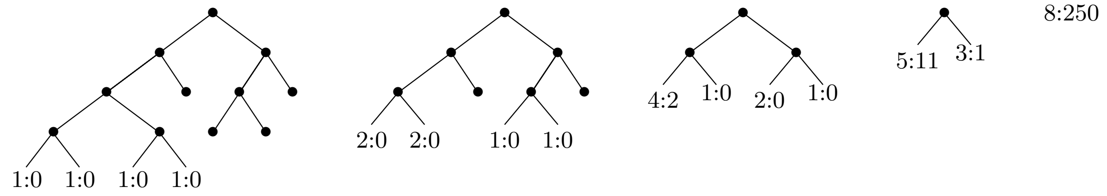


Figure 9.10 Computing the rank of the tree in Figure 9.9. As we move up in the tree, we replace a vertex v with $L\colon R$ where R is the rank of the subtree rooted at v and L is how many leaves it has. When information is no longer needed, we discard it to keep the figures from getting cluttered.

Example 9.13 Calculating statistics for RP-trees When RP-trees are used as data structures, items of data may be stored at the leaves and an "action" such as finding an item in a list may involve finding the appropriate leaf by traversing the path from the root to the leaf. How fast can data in such a tree be accessed?

The tree is being used as a decision tree and the number of decisions needed to reach the leaf equals the length of the path. Thus, the time needed to find an item this way is usually nearly proportional to the length of the path traversed. Given an RP-tree T, the length of the longest path from the root to a leaf, $m(T)$, say, and the average over all leaves of the lengths of the paths, $\mu(T)$, say, are therefore important measures of how good the tree is for storing data. Worst case (i.e., longest) time is proportional to $m(T)$ and average time to $\mu(T)$.

Given a particular tree T, we could calculate $m(T)$ and $\mu(T)$. Suppose we are told that the algorithm for creating the data structure constructs a random tree from some class; e.g., the set of n-leaf unlabeled full binary RP-trees. How can we get information about $\mu(T)$ in this case? Here are some possible approaches assuming we are dealing with unlabeled full binary RP-trees.

- Average over all: We might try to compute the average value of $\mu(T)$ over all such trees, provided we have a way to list all of them. Call the average value $\mu(n)$. We could then compute $\mu(T)$ for each tree T and average the results to obtain $\mu(n)$. If we are lucky enough to have an unranking algorithm, we can list all the trees using $\mathtt{UNRANK}(i,n)$ for $i = 0, 1, \ldots, b_n - 1$. Unfortunately, b_n is much too large for realistic values of n, so the program would take too long to run.
- Generate some at random and average: Another method is to generate n-leaf unlabeled full binary RP-trees at random by the method mentioned at the start of Section 3.2: Choose a random integer in $[0, b_n)$, unrank it and study the result. Repeat this procedure many times to get a good estimate. Choosing many elements from a set at random and studying them is known as the *Monte Carlo method* or *Monte Carlo simulation*. Studying it would take us too far afield.
- Use generating functions: We might try to find a theoretical tool. Indeed, we'll be able to compute $\mu(n)$ using generating functions (Exercise 11.2.16 (p. 329)). Theoretical methods can be wonderful when they work, but they have a nasty habit of not working when we change the problem. For example, our method will not give us the average of $m(T)$, the length of the longest path to the root. It can be estimated theoretically, but it is far more difficult than determining the average of $\mu(T)$.

Suppose we are looking at structures where we don't have an unranking algorithm and we can't afford to list all of them. It appears that we must use generating functions. This is not necessarily the case. Suppose that we have an unranking algorithm for a set that contains the one we are interested in and is not too much larger. We can use that algorithm, rejecting completely those structures that lie outside the set of interest. Here is a general pseudocode procedure for this method.

```
Procedure ESTIMATE(setsize,ncases,parameters)
     Initialize
     /* Loop to generate ncases examples. */
     For i = 1, 2, ..., ncases:
          /* Need a case in set of interest. */
          Set needcase to true
          While needcase is true:
               Choose a random integer j ∈ [0, setsize)
               T = UNRANK(j, parameters)
               If T is okay, then set needcase to false
          End while
          Store information about T as desired
     End for
     Finish it up: calculate and output concluding information
End
```

We will not study such algorithms in this text. □

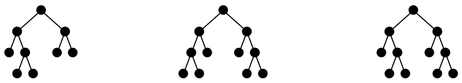
Exercises

9.3.1. Using Definition 9.1, recursively construct *all* unlabeled RP-trees with at most 4 vertices. Note that you are not supposed to simply list them. You should iterate the definition and retain all trees of at most 4 vertices that arise. When the list does not change during an iteration, it is complete.
Note: You are asked for all trees, not just the binary ones.

9.3.2. Using Definition 9.4, recursively construct all unlabeled full *binary* RP-trees with at most 4 leaves.

9.3.3. Construct a table of b_n for $n \le 10$.

9.3.4. Compute the ranks of the unlabeled full binary RP-trees shown here.



9.3.5. Construct the unlabeled full binary RP-trees with eight leaves whose ranks are 100, 200, 300 and 400.

9.3.6. Prove that a full binary RP-tree with n leaves has $n-1$ other vertices.

9.3.7. We are interested in the unlabeled full binary RP-tree with n leaves and rank $b_n/2$; i.e., the tree just past the middle of the list. Call the tree $\mathcal{M}_n$.

(a) Construct $\mathcal{M}_3$, $\mathcal{M}_5$ and $\mathcal{M}_7$.

(b) Conjecture and prove the nature of $\mathcal{M}_n$ when n is odd.

(c) Conjecture and prove the nature of $\mathcal{M}_n$ when n is even.

9.3.8. Provide a recursive method for calculating the rank of a decreasing function in lex order.

9.3.9. Use equivalence relations to provide a formal definition for unlabeled RP-trees in terms of labeled RP-trees.

9.3.10. Provide a recursive method for calculating the lex order rank of a permutation.

9.3.11. Let $**$ stand for the binary operation of exponentiation. How parentheses are placed in the expression $x{*}{*}y{*}{*}z$ effects the answer. Thus $3{*}{*}(2{*}{*}3) = 3^{2^3} = 3^8 = 1458$ while $(3{*}{*}2){*}{*}3 = (3^2)^3 = 3^6 = 729$. We would like to generate all ways of parenthesizing $x_1 ** \cdots ** x_n$. This can be done by *first* selecting the *last* $**$ operation to be performed as in

$$(x_1 ** \cdots ** x_k) ** (x_{k+1} ** \cdots ** x_n),$$

and then proceeding recursively on $x_1 ** \cdots ** x_k$ and $x_{k+1} ** \ldots ** x_n$. (If $k = 1$ we have simply (x_1) on the left.) The recursion stops when every innermost pair of parentheses contains just one number as in (x_i). Call this final result a "parenthesizing."

(a) Show that if you remove the x_i's from a parenthesizing, it is possible to tell where they belong. Thus all we need are the parentheses.

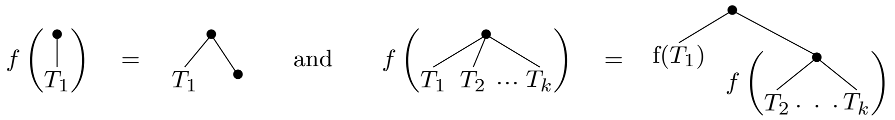
(b) Show that the set of possible parentheses patterns leads to a tree that looks the same as that in Figure 9.6 with • replaced by ( ) and $\texttt{JOIN}(A, B)$ interpreted as (AB).

(c) It follows from (b) that there is a simple correspondence between the parenthesized expressions and unlabeled full binary RP-trees. Describe it.

9.3.12. If T_i are unlabeled RP-trees, let $[T_1, \ldots, T_k]$ denote the unlabeled rooted RP-tree in which the ith edge from the root leads to the root of T_i. In particular, $[\,]$ is the tree •. We define a map f from the unlabeled RP-trees to the unlabeled full binary RP-trees recursively as follows. Let $f\big([\,]\big) = [\,] = \bullet$ and

$$f\big([T_1, \ldots, T_k]\big) = \Big[f(T_1),\ f\big([T_2, \ldots, T_k]\big)\Big] \qquad 9.6$$

when $k > 0$. Pictorially,



(a) Show that f is a bijection between n-vertex unlabeled RP-trees and n-leaf unlabeled full binary RP-trees.

(b) Use the above correspondence to find a procedure for ranking and unranking the set of all n-vertex unlabeled RP-trees. Provide a local description like Figure 9.6.

*9.3.13. In this exercise you will obtain a formula for b_n by proving a simple recursion. You might ask "How would I be expected to discover such a result?" Our answer at this time would be "Luck and/or experience." When we study generating functions, you'll have a more systematic method.

Let $\mathcal{L}_n$ be the set of n-leaf unlabeled full binary RP-trees with one leaf marked and let $\mathcal{V}_n$ with the set with one vertex marked.

(a) Prove that $\mathcal{L}_n$ has nb_n elements and that $\mathcal{V}_n$ has $(2n-1)b_n$ elements.

(b) Consider the following operation on each element of $\mathcal{L}_{n+1}$. If x is the marked leaf, let f be its father and b its brother. Remove x and shrink the edge between f and b so that f and b merge into a single vertex. Prove that each element of $\mathcal{V}_n$ arises exactly twice if we do this with each element of $\mathcal{L}_{n+1}$.

(c) From the previous results, conclude that $(n+1)b_{n+1} = 2(2n-1)b_n$.

(d) Use the recursion just derived to obtain a fairly simple formula for b_n.

Notes and References

Books on combinatorial algorithms and data structures usually discuss trees. You may wish to look at the references at the end of Chapter 6. Grammars are discussed extensively in books on compiler design such as [1].

1. Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley (1977).