4

Trees

4.1 Allowing Multiple Successors

The graphs of the previous chapter were a generalization of the linear lists to structures without order. We now turn to another generalization, still keeping some order, but a more relaxed one. The main motivation may be the fact that a linear list is ultimately not able to support both efficient searches and updates. To get the possibility to perform a binary search, the elements have to be stored sequentially, but then inserts and deletes may require $\Omega(n)$ steps. Using a linked list, on the other hand, may reduce the update time to $O(1)$, but at the price of losing the ability to locate the middle elements needed for the binary search, so the time complexity can be $\Omega(n)$.

The problem is that the constraint of every record having only a single successor, as depicted on the left-hand side of Figure 4.1, might be too restrictive. This chapter explores the structures one obtains by allowing each element to be directly followed by possibly more than one other element, as on the right-hand side of Figure 4.1. Because of the repeated branching, such structures are known as *trees*. Formally, the definition of a tree is general enough to encompass a broad family of recursive structures, and we shall deal with several such families in this and the following chapters.

Definition 4.1. A *tree* is a set of elements called *nodes* for which:

(i) there is a special element in the set, called the *root*;
(ii) the other elements can be partitioned into m subsets, each of which is a tree.

Actually, almost any set can then be considered to be a tree. Refer for example to the nodes in the left part of Figure 4.2, 5 might be chosen as the root, and the others can be partitioned into $m = 3$ subsets, as shown. The convention is

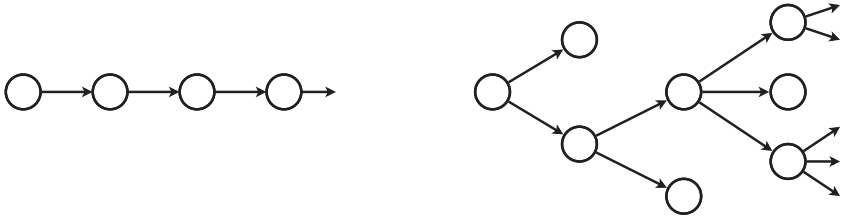


Figure 4.1. Linear lists versus tree structure.

to draw such a tree in levels top down, the root being placed on the top in level 0, and the m subtrees following in some order below the root, starting with the next level. This yields the equivalent tree on the right side of Figure 4.2. Accordingly, the vocabulary of genealogic trees has been adopted, for example, 3 is the *parent* of 4, and 3, 9, and 10 are *siblings* and the *children* of 5. Nodes without children, like 9 or 7, are called *leaves*, and the others, like 8 or 5, are called *internal* nodes.

A tree will be the natural choice if one has to represent a set that is intrinsically hierarchic, like a book, which can be subpartitioned into parts, and then into chapters, sections, sentences, etc.; or a university, partitioned into faculties, departments, and so on. The interesting aspect, however, of using trees, is that they might be the preferred choice even in other situations, which have *a priori* nothing to do with a recursive structure. Nonetheless, the properties of trees we shall see may turn them into the optimal solution for many problems.

Keeping the definition of a tree in its general form, with no limits imposed on the number m of subtrees, may be appropriate in many situations, but the processing is more complicated. As alternative, one defines a *binary tree* as follows:

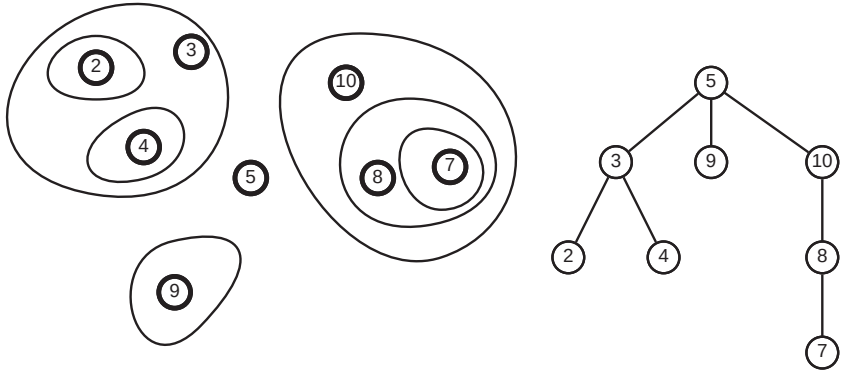


Figure 4.2. A tree structure.

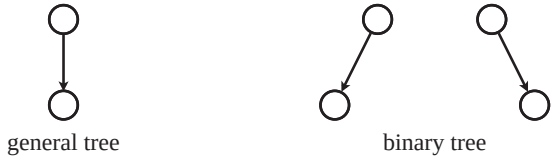


Figure 4.3. General and binary trees with two nodes.

Definition 4.2. A *binary tree* is a set of elements which is:

(i) either empty, or
(ii) consists of a special element, called the *root*, and two subsets that are themselves binary trees and are called *left subtree* and *right subtree*, respectively.

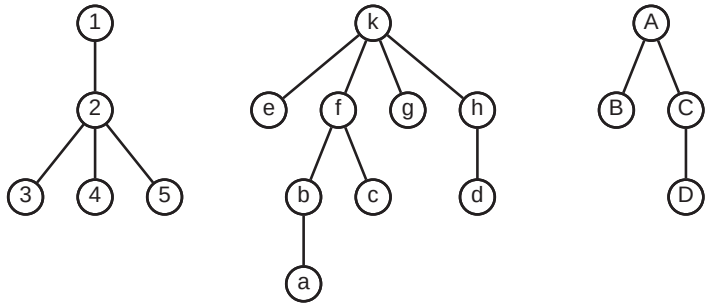
Note that a binary tree is therefore *not* a special case of a general tree. First, a general tree is never empty as it contains at least a root. Second, the subtrees of a general tree are not ordered, while we do differentiate between a left and a right child of a node in a binary tree. In particular, there is only a single general tree with two nodes, as shown in Figure 4.3, but there are two different binary trees with two nodes: both have a root, but the second node can be either a left, or a right child, and this will be reflected in our drawings.

4.2 General versus Binary Trees

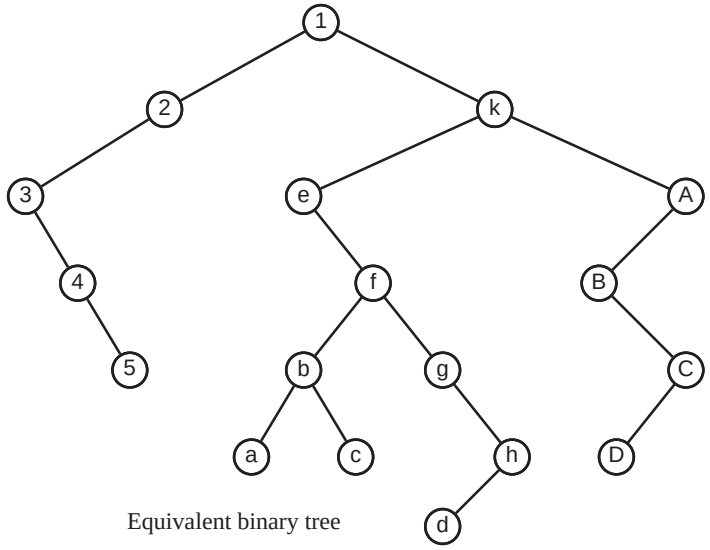
Limiting the number of children of a node to at most 2 has the advantage of using only two pointers for each node. General trees may be handled by transforming them first into equivalent binary trees, as follows. Consider an ordered *forest* $\mathcal{F}$, which is a sequence of general trees; we shall construct an equivalent binary tree $\mathcal{T}$, by using the same set of nodes, but reordering them differently, so that only two pointers, to a left and a right child, are needed for each node.

The root of $\mathcal{T}$ will be the root of the leftmost tree in $\mathcal{F}$. The left child of a node in $\mathcal{T}$ will be the first (leftmost) child of that node in $\mathcal{F}$, and the right child of a node v in $\mathcal{T}$ will be the sibling immediately to the right of v in $\mathcal{F}$, if there is such a sibling at all. The roots of the trees in $\mathcal{F}$ are special cases, and are considered as siblings, as if they were the children of some virtual global root of the forest. This procedure yields the binary tree in the lower part of Figure 4.4 if we start with the forest in the upper part of the figure.

This construction process is reversible, implying that the function relating a given forest with its equivalent binary tree is a bijection. It follows that the number of different such forests with n nodes is equal to the number of different binary trees with n nodes.



Forest of 3 general trees



Equivalent binary tree

Figure 4.4. Equivalence between binary trees and ordered forests of general trees.

Background Concept: Number of Binary Trees

To evaluate the number of such forests or binary trees, it is convenient to use the following compact representation of trees, which is also useful when the information describing a tree has to be stored or transmitted. A tree T with root R and subtrees $T_1, T_2, \ldots, T_k$, will be represented by the recursively defined sequence $(\mathrm{R}\ T_1\ T_2 \cdots T_k)$. For example, the sequence corresponding to the tree of Figure 4.2 is

$$\Big(5\ \big(3\ (2)\ (4)\big)\ \big(9\big)\ \big(10\ (8\ (7))\big)\Big).$$

If one is only interested in the structure of the tree, and not in the values of its nodes, the values can be eliminated and one is left with a sequence

of opening and closing parentheses, as in the preceding example:

$$(\, (\, (\,) \, (\,) \,) \, (\,) \, (\, (\, (\,) \,) \,) \,).$$

The sequence for a forest is just the concatenation of the sequences of the corresponding trees, so $(\,) \, (\,)$ represents two trees, each consisting of a single node. Notice that not every sequence of parentheses is appropriate. Since every opening parenthesis has a matching closing one, their number must be equal. Denoting a sequence of x opening parentheses and y closing parentheses as a (x, y)-sequence, the sequence corresponding to a forest with n nodes has thus to be an (n, n)-sequence.

That this is not a sufficient condition can be seen when considering, e.g., the sequence $S = (\, (\, (\,) \, (\,) \,) \,) \, (\, (\,)$, which is not properly nested. So the following nesting condition has to be added: when scanning the sequence left to right, at each point, the number of opening parentheses seen so far is not smaller than the number of closing parentheses. Let us call *good* (n, n)-sequences those satisfying the nesting condition. To evaluate their number, we consider the complementing set of *bad* (n, n)-sequences.

In any bad (n, n)-sequence, the number of closing parentheses exceeds the number of opening ones, so there must be a leftmost position in a left to right scan at which this happens for the first time. In the example of the sequence S, it is at position 7, and the violating closing parenthesis is emphasized. Consider then the function f transforming such bad (n, n)-sequences by inverting all the parentheses following this first occurrence. For our example, the resulting sequence would be $f(S) = (\, (\, (\,) \, (\,) \,) \,) \,) \,) \, (\, (\,$, in which all inverted parentheses are emphasized. The resulting sequence will always be an $(n-1, n+1)$-sequence, and since the function is reversible by applying it again, that is, $f^{-1} = f$, the function f is a bijection, implying that the number of bad (n, n)-sequences is equal to the total number of $(n-1, n+1)$-sequences, which is $\binom{2n}{n-1}$. We can thus conclude that

$$\begin{array}{ccccc} \text{number of} & & \text{number of} & & \text{total number} & & \text{number of} \\ \text{forests with} & = & \text{good } (n,n)\text{-} & = & \text{of } (n,n)\text{-} & - & (n-1,n+1)\text{-} \\ n \text{ nodes} & & \text{sequences} & & \text{sequences} & & \text{sequences} \end{array}$$

$$= \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n}.$$

These numbers are known as *Catalan* numbers, the first of which are $1, 2, 5, 14, 42, \ldots$. Asymptotically, the nth Catalan number is $O(\frac{4^n}{n^{3/2}})$.

Returning to the equivalence between ordered forests and binary trees, it seems that we can always restrict ourselves to deal with binary trees, which are more convenient to handle, since a general tree can be transformed into an equivalent binary one. Indeed, most of our discussion on trees will assume that the trees are binary. It should however be noted that a certain price has been payed for this convenience: some direct connections that existed in the general tree have been replaced by more expensive alternatives. For example, in the forest of Figure 4.4, only two links were needed to get from node k to node d, whereas in the equivalent binary tree, this path is now of length 5.

4.3 Binary Trees: Properties and Examples

Similarly to what we saw for graphs in the previous chapter, there is a need to explore all the nodes of a tree in a systematic way. We shall use the term *visit* to describe such an exploration. Any visit consists of a well-defined order in which all the nodes are processed, and of some action to be taken at each of the nodes. In our examples, this action will be simply to print the value of the node.

By convention, given a binary tree, its left subtree is visited before the right one, but there are three possibilities for the visit of the root: in a top down processing, the root will be processed before dealing, recursively, with the subtrees, while in applications requiring a bottom up approach, the root will be visited after the visit of the subtrees has been completed. An interesting third alternative is to visit the root after the left subtree but before the right one. The orders corresponding to these three visit strategies are known, respectively, as *pre-order*, *post-order*, and *in-order*.

Consider the general binary tree depicted in the upper drawing of Figure 4.5. The recursive visit procedures and the results produced by applying them on this tree are given in Figure 4.6.

If every internal node has exactly two children, the tree is called *complete*, as the one in the left lower part of Figure 4.5. The tree in the upper part of this figure is not complete, because the internal nodes labeled B, C, G and E have only a single child each. Instead of defining a complete tree as a special case of a general binary tree, one could have given it a recursive definition of its own, only slightly different from Definition 4.2:

Definition 4.3. A *complete binary tree* is a set of elements which is:

(i) either a single element, or
(ii) consists of a special element, called the *root*, and two subsets that are themselves complete binary trees and are called *left subtree* and *right subtree*, respectively.

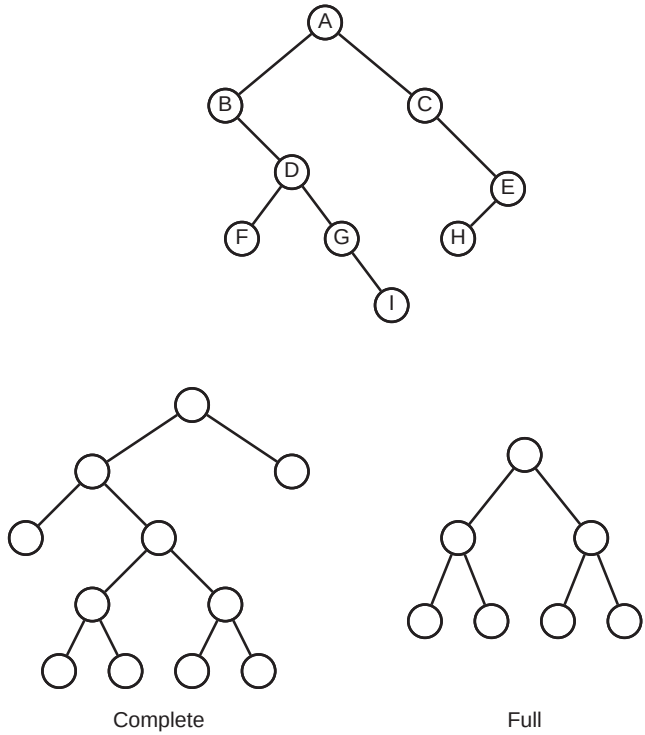


Figure 4.5. General, complete, and full binary trees.

Since this is a recursive definition, it covers actually a broad range of similar structures. Consider, for instance, the definition of an arithmetic expression including only binary operators, like $+, -, \times, /, \uparrow, \ldots$ (as before, we use the symbol $\uparrow$ for exponentiation).

Definition 4.4. An *arithmetic expression* is a string of elements which is:

(i) either a single element, called an *operand*, or
(ii) consists of a special element, called the *operator*, and two substrings that are themselves arithmetic expressions.

Pre-order(T)	In-order(T)	Post-order(T)
visit $root(T)$	In-order$(left(T))$	Post-order$(left(T))$
Pre-order$(left(T))$	visit $root(T)$	Post-order$(right(T))$
Pre-order$(right(T))$	In-order$(right(T))$	visit $root(T)$
A B D F G I C E H	B F D G I A C H E	F I G D B H E C A

Figure 4.6. Visiting a binary tree.

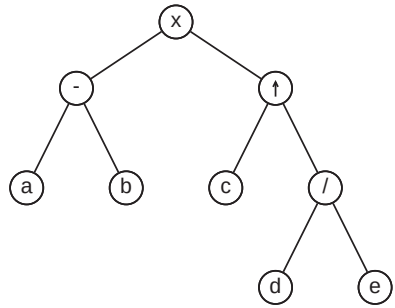


Figure 4.7. Complete binary tree of an arithmetic expression.

In fact, the conclusion is that such an expression can be considered as a complete binary tree, which may be useful for certain applications. As example, refer to the expression

$$(a - b) \times c \uparrow (d/e)$$

and to the corresponding binary tree given in Figure 4.7. Visiting this tree in the orders discussed earlier yields

For pre-order	For in-order	For post-order
$\times - a\ b \uparrow c\ /\ d\ e$	$((a - b) \times (c \uparrow (d\ /\ e)))$	$a\ b - c\ d\ e\ /\ \uparrow \times$

which can be recognized as prefix, infix, and postfix notation, respectively, of the given expression. These notations have been discussed in Section 2.3.1 about stacks. As mentioned, parentheses are needed for in-order to change the operator priorities, when necessary.

We now turn to the problem of formally proving certain properties of trees. The proof of many properties dealt with in discrete mathematics is often based on *induction*. This works fine for mathematical identities, once they are known, like $\sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2$, but it is not trivial to extend this technique also to the more involved structure of a binary tree. Indeed, there are several options on which to base the induction: the number of nodes, the number of leaves, the depth of the tree, etc. One also has to show that every possible tree is covered by the inductive procedure. As alternative, one can exploit the recursive structure of a tree to devise a type of proof called *induction on the structure* of the tree. The following example will illustrate this approach.

The *depth* of a node in a tree is the level on which it appears, where the level of the root is defined as 0. In many applications, our interest is focused on the leaves of the tree rather than the internal nodes, and in particular on the depths of these leaves. The exact shape of the tree might be less important, and a tree is often characterized by the sequence of depths of its leaves. For example, these

sequences would be 3 3 4, 1 2 4 4 4 4, and 2 2 2 2 for the three binary trees of Figure 4.5, and each sequence is given in nondecreasing order, rather than in the order of the leaves themselves.

We would like to recognize those sequences that correspond to complete binary trees. A sequence may not correspond to any tree, like 1 1 1, since no binary tree can have more than two leaves on level 1. Other sequences may correspond to binary trees, but not to complete ones, like 3 3 4.

Theorem 4.1. Given is a complete binary tree with leaves on levels $\ell_1, \ell_2, \ldots, \ell_n$, respectively. Then

$$\sum_{i=1}^{n} 2^{-\ell_i} = 1.$$

Proof by induction on the structure of the tree. For $n = 1$, the only leaf must also be the root, so its depth is 0, and we have $\sum_{i=1}^{n} 2^{-\ell_1} = 2^0 = 1$.

For $n > 1$, the tree consists of a root and nonempty left and right subtrees. Without loss of generality, we may assume that there is some k, $1 \le k < n$, such that the leaves in the left subtree are indexed $1, \ldots, k$, while those in the right subtree are indexed $k + 1, \ldots, n$. Denote the depth of a leaf in one of the subtrees by d_i, we then have that $\ell_i = d_i + 1$. It follows that

$$\begin{aligned}\sum_{i=1}^{n} 2^{-\ell_i} &= \sum_{i=1}^{k} 2^{-\ell_i} + \sum_{i=k+1}^{n} 2^{-\ell_i} \\ &= \sum_{i=1}^{k} 2^{-(d_i+1)} + \sum_{i=k+1}^{n} 2^{-(d_i+1)} \\ &= \frac{1}{2}\sum_{i=1}^{k} 2^{-d_i} + \frac{1}{2}\sum_{i=k+1}^{n} 2^{-d_i} = \frac{1}{2} + \frac{1}{2} = 1,\end{aligned}$$

where we have used the inductive assumption for each of the two subtrees. ∎

4.4 Binary Search Trees

One of the main applications of binary trees is the processing of a set of records, as explained in the previous chapters. Operations that need to be supported are the efficient search for a given record in the set, as well as insertion and deletion. This leads to the definition of *binary search trees*, in which every record is assigned to one of the nodes of the tree, and can be identified by its value stored in the node. For the simplicity of the discussion, we shall assume

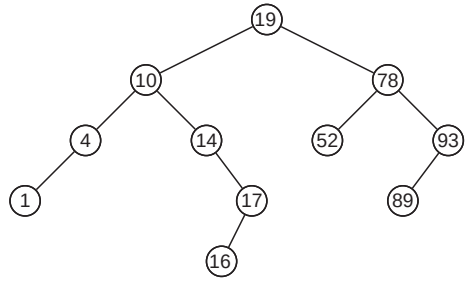


Figure 4.8. Binary search tree.

that all the values in the tree are different from each other. The special property of binary search trees is:

Definition 4.5. In a binary search tree, for every node x with value v_x, the values in the left subtree of x are smaller than v_x and the values in the right subtree of x are larger than v_x.

Here and subsequently, we shall use the convention of denoting by v_x the node containing the value x. An example of a binary search tree can be seen in Figure 4.8. Scanning the tree using inorder will list the elements ordered by increasing values. This property can also be used to induce an order in any binary tree. For example, if one considers the tree in the lower part of Figure 4.4 as a binary search tree, the implied order of its elements would be

3 4 5 2 1 e a b c f g d h k B D C A.

4.4.1 Search

Searching for an element in a binary search tree always starts at the root. If the element is not found there, we know already if it is smaller or larger than the value stored in the root. In the former case, the search continues recursively in the left subtree, and in the latter case in the right subtree.

For example, suppose one looks for an element with value 12 in the tree of Figure 4.8. The value 12 is smaller than 19 stored in the root, so the next comparison is with the root of the left subtree, storing 10; 12 is larger, hence one continues to v_{14}; 12 is smaller than 14, so we should continue to the left child of v_{14}, but there is no such left child. We conclude that v_{12} is not in the tree.

The formal recursive procedure searching for an element v_x in the subtree rooted by T is given in Figure 4.9. It should be invoked by Search(*root*, x), and it returns a triplet (*indic*, R, *side*). If a node v_x with value x is in the tree, then

```
Search(T, x)
    v ⟵ value(T)
    if x = v then return (found, T, 0)
    else  if x < v then
              if left(T) = NIL then          return (not-found, T, left)
              else                           Search(left(T), x)
          else  // here x > v
              if right(T) = NIL then         return (not-found, T, right)
              else                           Search(right(T), x)
```

Figure 4.9. Recursive search for x in tree rooted by T.

the indicator *indic* = found and R points to v_x (the third element *side* is not used in this case). Otherwise, *indic* = not-found, R is a pointer to a node w which should be the parent of v_x, and *side* = left or right, showing whether v_x should be a left or right child of w.

The number of comparisons and thus the complexity of the search procedure is clearly dependent on the level in the tree at which v_x, in case of a successful search, or w, for an unsuccessful one, are found. We have, however, no control over the depth of the tree or the level at which a certain node will appear. In the worst case, for example, if the nodes are created by increasing order of their values, the depth of the tree will be $n-1$, which is clearly unacceptable. The best scenario would be a *full* binary tree, defined as one having all its leaves on the same level, like the right tree in the lower part of Figure 4.5. There is regretfully no consensus about the use of the terms *full* and *complete*, and some books invert their definitions.

In a full binary tree with n nodes, the depth of a node is at most $\lfloor \log_2 n \rfloor$, and this will thus be the bound on the search time. We shall see in the following chapters what actions can be taken to ensure that the depth of the binary search tree will be logarithmic in n even for the worst case.

4.4.2 Insertion

To insert an element v_x with value x into a tree rooted at T, one has first to check that there is no node with value x already in the tree. One may thus search for x using the procedure of Figure 4.9, which returns a pointer to a node R that should be the parent node of v_x, and indicates whether v_x is a left or right child of R. The formal procedure is in Figure 4.10.

Note that any newly inserted element creates a new leaf in the tree. The complexity of inserting an element depends on the depth at which the corresponding leaf will be inserted, which is at most the depth of the tree. As for the search, insertion might thus require $\Omega(n)$ steps.

```
Insert(T, x)
    (indic, R, side) ⟵ Search(T, x)
    if indic = not-found then
        create node v_x
        value(v_x) ⟵ x
        left(v_x) ⟵ right(v_x) ⟵ NIL
        side(R) ⟵ v_x
```

Figure 4.10. Insertion of v_x in tree rooted by T.

4.4.3 Deletion

While insertion into and deletion from linear lists could be treated more or less symmetrically, this is not the case when dealing with search trees. Indeed, a node is always inserted as a leaf, whereas one might wish to delete any node in the tree, possibly also an internal one.

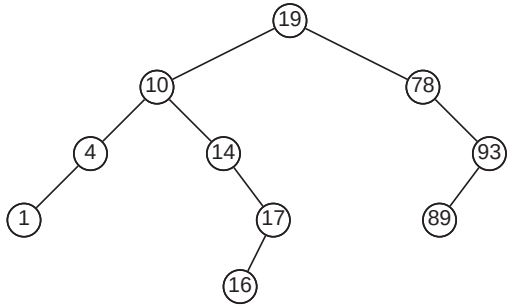
The deletion procedure gets as parameters a pointer T to the root of the tree and the value x of the element to be deleted. The process starts by checking whether there is a node v_x with value x in the tree, and where it is located. If such an element is found, the process distinguishes between three cases, according to the number of children N of v_x:

(i) If $N = 0$, v_x is a leaf, and it can be removed by changing the pointer to v_x, if it exists, into a NIL-pointer, without further action. If v_x is the root, we are in the special case of a tree with a single node, and this tree becomes empty after the deletion.

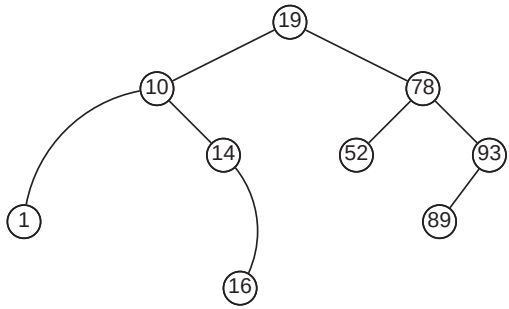
(ii) If $N = 1$, v_x is deleted by directly connecting its parent node, if it exists, to the only child of v_x. If v_x is the root, its only child becomes the new root.

(iii) If $N = 2$, we have to deal with the problem that there is a single incoming pointer to v_x, but there are two outgoing pointers. Simply eliminating the node would thus disconnect the tree. To circumvent the problem, the node v_x will not be deleted, only the value x it contains will be erased and replaced by some other value y that does not disturb any possible subsequent searching process. This value y should be the value of one of the nodes v_y of the tree, and the only possible candidates for y are thus the immediate predecessor or successor of x in the ordered sequence of all the elements stored in the tree. Let us choose the successor. After replacing x by y, the node v_y should be deleted, which can be done recursively.

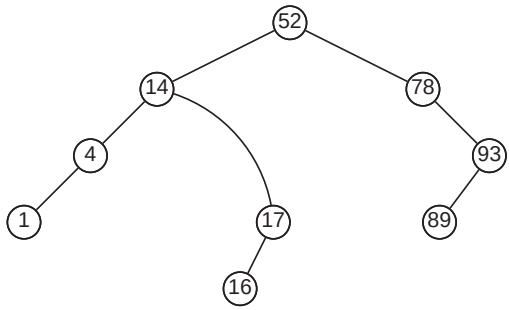
The successor v_y of a node v_x which has a right child is found as follows: it has to be in the right subtree of v_x, since this subtree includes the elements with smallest values that are larger than x, and it has to be the element with smallest



Deleting 52



Deleting 4 and 17



Deleting 10 and 19

Figure 4.11. Deletion from a binary search tree.

```
Delete(T, x)
    (indic, R, dum) ⟵ Search(T, x)
    if indic = found then
        case 0: R is a leaf, then    // just erase the leaf
            if R is the root then erase it    // case of empty tree
            else if R is a left child then    left(parent(R)) ⟵ NIL
            else                              right(parent(R)) ⟵ NIL

        case 1: R has 1 child, then    // bypass the node, then erase
            if left(R) = NIL then     child ⟵ right(R)
            else                      child ⟵ left(R)
            if R is the root then     root ⟵ child
            else if R is a left child then    left(parent(R)) ⟵ child
            else                              right(parent(R)) ⟵ child

        case 2: R has 2 children, then    // search for successor C to replace R
            C ⟵ right(R)
            while left(C) ≠ NIL       C ⟵ left(C)
            value(R) ⟵ value(C)
            Delete(C, value(C))
```

Figure 4.12. Deletion of v_x from tree rooted by T.

value within this subtree. The way to get from v_x to v_y is thus to take one step to the right, and then repeatedly steps to the left, as long as possible. Refer, for example, to the tree in the lower part of Figure 4.4, assuming that the order of its elements' values is indeed given by an inorder traversal, then the successor of v_{e} in this order is v_{a}, and the successor of v_{k} is v_{B}. Note that the successor is not necessarily a leaf, as can be seen in the latter example, but it cannot have a left child. This is the reason that deleting v_y instead of v_x will not cause a loop of recursive calls, since it is invoked in the case $N = 2$, but the deletion of the successor involves one of the cases $N = 0$ or $N = 1$.

Figure 4.11 brings examples of all the cases for the deletion process. For each of them, the starting point is the tree of Figure 4.8. In the upper example in Figure 4.11, the element to be deleted is the one with value 52, which is a leaf. The examples in the middle part of Figure 4.11 correspond to $N = 1$: deleting v_4 is a case in which two *left* pointers are combined, while the deletion of v_{17} replaces a *right* and a *left* pointer by a single *right* pointer. Finally, the tree in the lower part of Figure 4.11 shows two instances of the case $N = 2$: the successor of v_{10} is the root v_{14} of its right subtree, and v_{14} itself has only one child; and the successor of v_{19} is v_{52}, which is a leaf.

The formal algorithm for deleting an element v_x from a tree rooted at T is given in Figure 4.12. A technical remark: one could save the special treatment involving the *root* by the addition of a sentinel element carrying no value, as already mentioned earlier in Sections 2.4.1 and 2.4.2. Instead of reserving a

root pointer to access the tree, there would be a root node v_{root} without value, but having the entire tree as, say, its right subtree. For large enough trees, the relative overhead of such an element would be negligible, and the processing of all the cases would then be uniform.

The complexity of deleting an element v_x from a binary search tree is thus not bounded by the level at which v_x appears in the tree, since it might involve the successor of v_x at a lower level. However, even in the worst case, the complexity is still bounded by the depth of the tree.

Exercises

4.1 Draw the 5 different forests as well as the 5 different binary trees with 3 nodes, and show how they correspond using the equivalence displayed in Figure 4.4.

4.2 Draw the binary tree whose in-order and post-order traversals are, respectively, B G L F I K A C N H J E M D and L F G K C N A I J D M E H B. What is the pre-order traversal of this tree? Is it possible to determine the tree on the basis of its pre-order and post-order traversals alone? Show how to do it, or give a counterexample.

4.3 A complete *ternary* tree is a tree in which each internal node has exactly 3 children. Show by induction on the structure of the tree that a complete ternary tree with n leaves has $(n-1)/2$ internal nodes.

4.4 Given is a sequence of n ordered integer sequences $A_1, \ldots, A_n$, of lengths $\ell_1, \ldots, \ell_n$, respectively. We wish to merge all the sequences into a single ordered sequence, allowing only pairs of adjacent already ordered sequences to be merged. The number of steps needed to merge a elements with b elements is $a+b$. If $n=2$, there is only one possibility to merge A_1 with A_2, but for $n=3$, one could merge $A_1(A_2A_3)$ or $(A_1A_2)A_3$. The corresponding times would be $(\ell_2+\ell_3)+(\ell_1+\ell_2+\ell_3)$ or $(\ell_1+\ell_2)+(\ell_1+\ell_2+\ell_3)$. For $n=4$, there are 5 possibilities.

Find a connection between the different possibilities to merge n sequences under the given constraint and binary trees. How many possibilities are there to merge ten sequences?

4.5 Given are two binary search trees with k and ℓ nodes, respectively, such that $k+\ell = 2^n-1$ for some $n>2$. All the 2^n-1 elements are different. Describe an algorithm running in $O(k+\ell)$ time, that merges the two trees into one full binary search tree.