containing s and s' in COLLECTION, and call the resulting set A.
2. On completion of the program, the sets in COLLECTION represent a partition of $S_1 \cup S_2$ into blocks of states that must be equivalent. M_1 and M_2 are equivalent if and only if no block contains both an accepting state and a nonaccepting state.

The running time of the algorithm (as a function of $n = \|S_1\| + \|S_2\|$, the number of states) is dominated by the set union algorithm. There can be at most $n - 1$ UNION's since each UNION instruction reduces the number of sets in COLLECTION by one, and there were only n sets to begin with. The number of FIND's is proportional to the number of pairs placed on LIST. This number is at most $n \times \|I\|$ since with the exception of the initial pair (s_1, s_2), a pair is placed on LIST only after a UNION instruction. Thus the time required to determine whether M_1 is equivalent to M_2 is $O(nG(n))$, assuming $\|I\|$ is a constant.

4.9 BALANCED TREE SCHEMES

There are several important classes of problems which are similar to the UNION–FIND problem but which apparently force us to fall back on techniques that require $O(n \log n)$ time (worst case) to process a sequence of n instructions. One such class of problems involves processing a sequence of MEMBER, INSERT, and DELETE instructions when the universe of possible elements is much larger than the number of elements actually used. In this case we cannot access an element by directly indexing into an array of pointers. We must use either hashing or a binary search tree.

If n elements have been inserted, the hashing method has an average access time which is constant but a worst-case time which is $O(n)$ per access. Using a binary search tree gives an expected access time of $O(\log n)$ per access. However, a binary search tree can also give a poor worst-case access time if the set of names is not static. If we simply add names to the tree without some mechanism to keep the tree balanced, we may eventually end up with a tree of n elements which has a depth close to n. Thus the worst-case performance of a binary search tree can be $O(n)$ per operation. The techniques of this section can be used to reduce the worst-case performance to $O(\log n)$ steps per operation.

Another class of problems requiring $O(n \log n)$ time is the on-line processing of sequences of n instructions containing the operations INSERT, DELETE, and MIN. Still a third such class of problems arises if we need to represent ordered lists and have the capability of concatenating and splitting lists.

In this section we shall present techniques that will allow us to process, on-line, sequences containing important subsets of the seven fundamental

4.10 DICTIONARIES AND PRIORITY QUEUES

In this section we shall consider the basic operations required to implement dictionaries and priority queues. Throughout this section we shall assume that elements are assigned to the leaves of a 2–3 tree in left-to-right order, and $L[v]$ and $M[v]$ (the "largest" descendant functions described in the previous section) are present at each nonleaf v.

To insert a new element a into a 2–3 tree we must locate the position for the new leaf l that will contain a. This is done by trying to locate element a in the tree. Assuming the tree contains more than one element, the search for a terminates at a vertex f such that f has either two or three leaves as sons.

If f has only two leaves l_1 and l_2, we make l a son of f. If $a < E[l_1]$,† we make l the leftmost son of f and set $L[f] = a$ and $M[f] = E[l_1]$; if $E[l_1] < a < E[l_2]$, we make l the middle son of f and set $M[f] = a$; if $E[l_2] < a$, we make l the third son of f. The L or M values of some proper ancestors of f may have to be changed in the latter case.

Example 4.9. If we insert the element 2 into the 2–3 tree of Fig. 4.27(a), we get the 2–3 tree of Fig. 4.27(b). □

Now suppose f already has three leaves, l_1, l_2, and l_3. We make l the appropriate son of f. Vertex f now has four sons. To maintain the 2–3 property, we create a new vertex g. We keep the two leftmost sons as sons of f, but change the two rightmost sons into sons of g. We then make g a brother of vertex f by making g a son of the father of f. If the father of f had two sons, we stop here. If the father of f had three sons, we must repeat this procedure recursively until all vertices in the tree have at most three sons. If the root is given four sons, we create a new root with two new sons, each of which has two of the four sons of the former root.

Example 4.10. If we insert element 4 into the 2–3 tree of Fig. 4.27(a), we find that the new leaf labeled 4 should be made the leftmost son of vertex c. Since vertex c already has three sons, we create a new vertex c'. We make leaves 4 and 5 sons of c, and leaves 6 and 7 sons of c'. We now make c' a son of vertex a. However, since vertex a already has three sons, we create another vertex a'. We make vertices b and c sons of the old vertex a, and vertices c' and d sons of the new vertex a'. Finally, we create a new root r and make a and a' sons of r. The resulting tree is shown in Fig. 4.28. □

Algorithm 4.4. Insertion of a new element into a 2–3 tree.

Input. A nonempty 2–3 tree T with root r and a new element a not in T.

Output. A revised 2–3 tree with a new leaf labeled a.

† $E[v]$ is the element stored at leaf v.

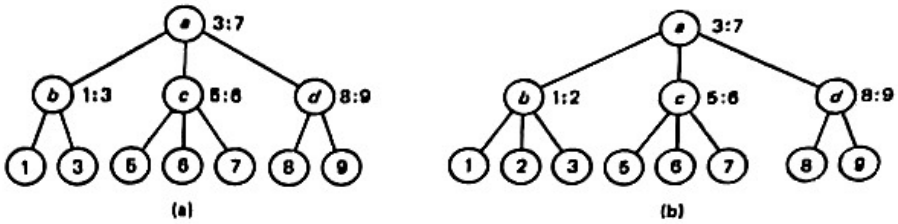

Fig. 4.27 Insertion into a 2–3 tree: (a) tree before insertion; (b) tree after inserting 2.

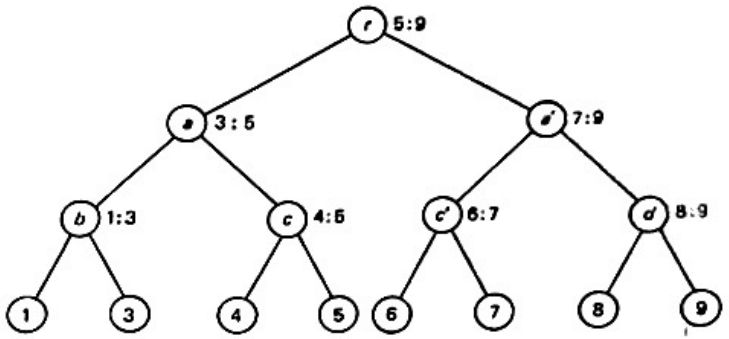

Fig. 4.28 Tree of Fig. 4.27(a), after inserting 4.

Method. We assume T has at least one element. To simplify description of the algorithm we have omitted the details of updating L and M at various vertices.

1. If T consists of a single leaf l labeled b, then create a new root r'. Create a new leaf v labeled a. Make l and v sons of r', making l the left son if $b < a$, otherwise, making l the right son.
2. If T has more than one vertex, let f = SEARCH(a, r), where SEARCH is the procedure in Fig. 4.29. Create a new leaf l labeled a. If f has two sons labeled b_1 and b_2, then make l the appropriate son of f. Make l the left son if $a < b_1$, the middle son if $b_1 < a < b_2$, the right son if $b_2 < a$. If f has three sons, make l the appropriate son of f and then call ADDSON(f) to incorporate f and its four sons into T. ADDSON is the procedure in Fig. 4.30. Adjust the values of L and M along the path