```
begin
    make LIST empty;
    v ← ELEMENT[i];
    while FATHER[v] ≠ 0 do
        begin
            add v to LIST;
            v ← FATHER[v]
        end;
    comment v is now the root;
    print NAME[v];
    for each w on LIST do FATHER[w] ← v
end
```

Fig. 4.18. Executing instruction FIND(i).

```
begin
  wlg assume COUNT[ROOT[i]] ≤ COUNT[ROOT[j]]
    otherwise interchange i and j in
    begin
        LARGE ← ROOT[j];
        SMALL ← ROOT[i];
        FATHER[SMALL] ← LARGE;
        COUNT[LARGE] ← COUNT[LARGE] + COUNT[SMALL];
        NAME[LARGE] ← k;
        ROOT[k] ← LARGE
    end
end
```

Fig. 4.19. Executing instruction UNION(i, j, k).

n	$F(n)$
0	1
1	2
2	4
3	16
4	65536
5	2^{65536}

Fig. 4.20. Some values of F.

```
for i ← 1 until n do
    begin
        j ← FIND(i);
        if j ≤ k then
            begin
                print i "is deleted by the "j"th EXTRACT_MIN instruc-
                  tion";
                UNION(j, SUCC[j], SUCC[j]);
                SUCC[PRED[j]] ← SUCC[j];
                PRED[SUCC[j]] ← PRED[j]
            end
    end
```

Fig. 4.23. Program for off-line MIN problem.

```
begin
    LIST ← (s₁, s₂);
    COLLECTION ← ∅;
    for each s in S₁ ∪ S₂ do add {s} to COLLECTION;
    comment We have just initialized a set for each state in S₁ ∪ S₂;
    while there is a pair (s, s') of states on LIST do
        begin
            delete (s, s') from LIST;
            let A and A' be FIND(s) and FIND(s'), respectively;
            if A ≠ A' then
                begin
                    UNION(A, A', A);
                    for all a in I do
                        add (δ(s, a), δ(s', a)) to LIST
                end
        end
end
```

Fig. 4.25. Algorithm for finding sets of equivalent states, assuming s_1 and s_2 are equivalent.

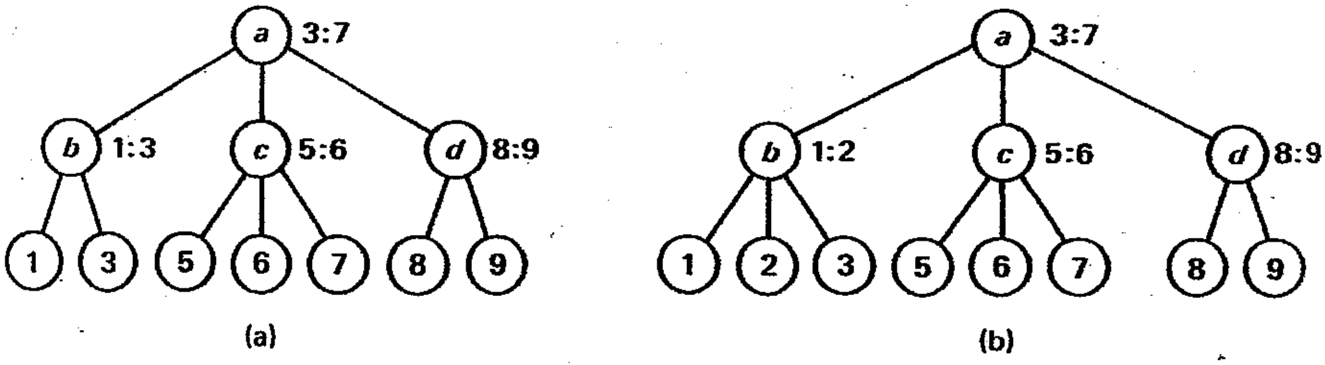

Fig. 4.27 Insertion into a 2–3 tree: (a) tree before insertion; (b) tree after inserting 2.

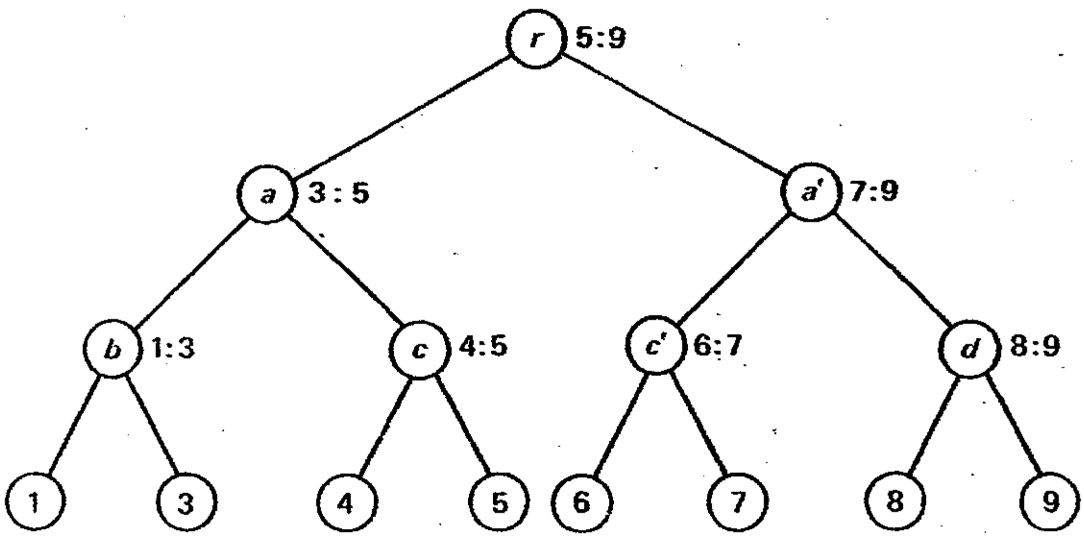

Fig. 4.28 Tree of Fig. 4.27(a), after inserting 4.

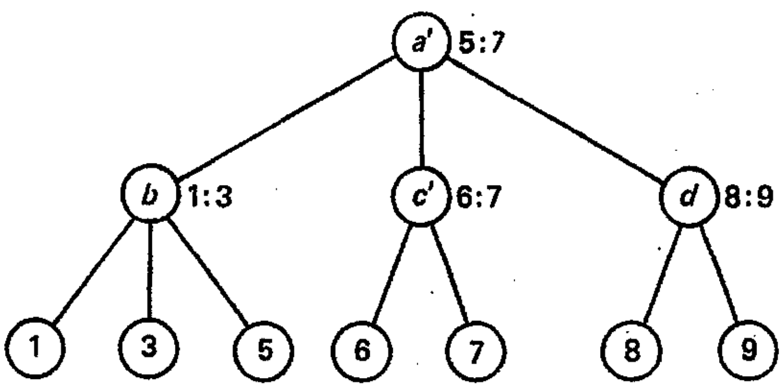

Fig. 4.31 Tree of Fig. 4.28, after removing 4.

```
procedure SEARCH(a, r):
if any son of r is a leaf then return r
else
    begin
        let s_i be the ith son of r;
        if a ≤ L[r] then return SEARCH(a, s_1)
        else
            if r has two sons or a ≤ M[r] then return SEARCH(a, s_2)
            else return SEARCH(a, s_3)
    end
```

Fig. 4.29. Procedure SEARCH.

```
procedure ADDSON(v):
begin
    create a new vertex v';
    make the two rightmost sons of v the left and right sons of v'
    if v has no father then
        begin
            create a new root r;
            make v the left son and v' the right son of r
        end
    else
        begin
            let f be the father of v;
            make v' a son of f immediately to the right of v;
            if f now has four sons then ADDSON(f)
        end
end
```

Fig. 4.30. Procedure ADDSON.

```
procedure IMPLANT(T1, T2):
if HEIGHT(T1) = HEIGHT(T2) then
    begin
        create a new root r;
        make ROOT[T1] and ROOT[T2] the left and right sons of r
    end
else
    wlg assume HEIGHT(T1) > HEIGHT(T2) otherwise
      interchange T1 and T2 and interchange "left" and "right" in
        begin
            let v be the vertex on the rightmost path of T1 such that
                DEPTH(v) = HEIGHT(T1) − HEIGHT(T2);
            let f be the father of v;
            make ROOT[T2] a son of f immediately to the right of v;
            if f now has four sons then ADDSON(f)†
        end
```

† If we wish to have L and M values for the new vertex which ADDSON(f) will create, we must first find the maximum descendant of v by following the path to the rightmost leaf.

Fig. 4.32. Procedure IMPLANT.

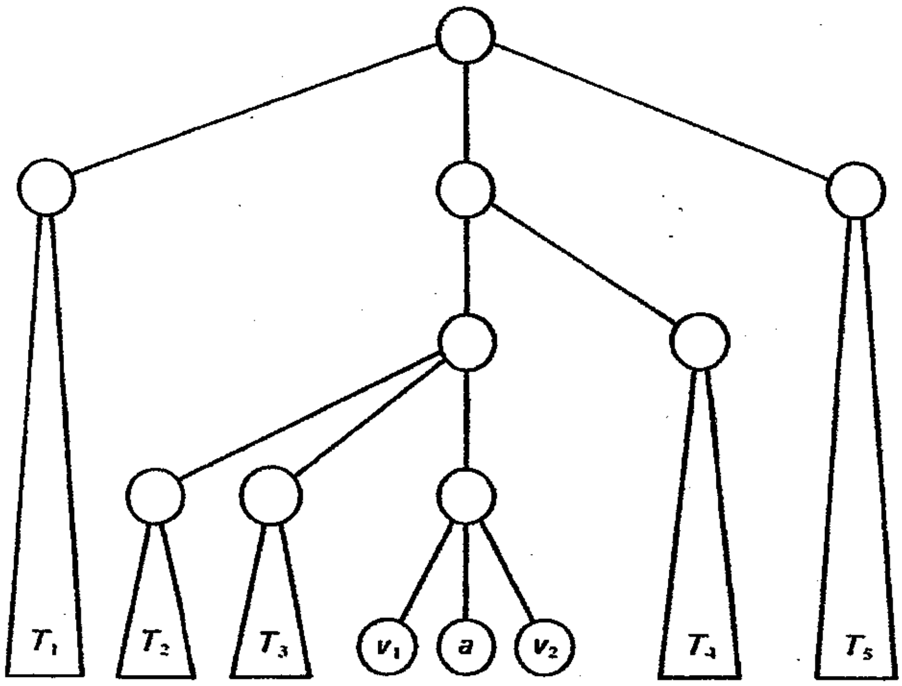

Fig. 4.33 Splitting a 2–3 tree.

procedure DIVIDE(a, T):
begin
    on the path from ROOT[T] to the leaf labeled a remove all vertices except the leaf;
    comment At this point T has been divided into two forests—the *left forest*, which consists of all trees with leaves to the left of and including the leaf labeled a, and the *right forest*, which consists of all trees with leaves to the right of a;
    while there is more than one tree in the left forest **do**
        begin
            let T' and T'' be the two rightmost trees in the left forest;
            IMPLANT(T', T'')†
        end;
    while there is more than one tree in the right forest **do**
        begin
            let T' and T'' be the two leftmost trees in the right forest;
            IMPLANT(T', T'')
        end
end

† The result of IMPLANT(T', T'') should be considered as remaining in the left forest. Similarly, when applied to trees in the right forest, the result of IMPLANT is a tree in the right forest.

Fig. 4.34. Procedure to split a 2–3 tree.

```
begin
    WAITING ← {1, 2, . . . , p};
    q ← p;
    while WAITING not empty do
        begin
            select and delete any integer i from WAITING;
            INVERSE ← f⁻¹(B[i]);
            for each j such that B[j] ∩ INVERSE ≠ ∅ and
              B[j] ⊄ INVERSE do
                begin
                    q ← q + 1;
                    create a new block B[q];
                    B[q] ← B[j] ∩ INVERSE;
                    B[j] ← B[j] − B[q];
                    if j is in WAITING then add q to WAITING
                    else
                        if ||B[j]|| ≤ ||B[q]|| then
                            add j to WAITING
                        else add q to WAITING
                end
        end
end
```

Fig. 4.35. Partitioning algorithm.

Data structure	Type of universe	Instructions permitted	Time to process n instructions on sets of size n	
			Expected time	Worst-case time
1. Hash table	Arbitrary set on which a hashing function can be computed	MEMBER, INSERT, DELETE	$O(n)$	$O(n^2)$
2. Binary search tree	Arbitrary ordered set	MEMBER, INSERT, DELETE, MIN	$O(n \log n)$	$O(n^2)$
3. Tree structure of Algorithm 4.3	Integers 1 to n	MEMBER, INSERT, DELETE, UNION, FIND	$O(nG(n))$ at most	$O(nG(n))$ at most
4. 2–3 trees with leaves unordered	Arbitrary ordered set	MEMBER, INSERT, DELETE, UNION, FIND, MIN	$O(n \log n)$	$O(n \log n)$
5. 2–3 trees with leaves ordered	Arbitrary ordered set	MEMBER, INSERT, DELETE, FIND, SPLIT, MIN, CONCATENATE	$O(n \log n)$	$O(n \log n)$

Fig. 4.36. Summary of properties of data structures.