UNIT-IV

Basic Traversal & Search Techniques: Techniques for Binary Trees and Graphs, Connected Components and Spanning Tress, Bi-Connected Components and DFS.

Sets and Disjoint set Union: Introduction, Union and Find operations.

Techniques for Binary Trees:

When the search necessarily involves the examination of every vertex in the object being searched, it is called a traversal.
There are many operations that we want to perform on binary trees.
When traversing a binary tree, we want to treat each node and its sub trees in the same fashion. If we let L,D,and R stand for moving left, printing the data, and moving right when at a node,then there are six possible combinations of traversal: LDR, LRD, DLR, DRL, RDL, and RLD.If we adopt the convention that we traverse left before right,then only three traversals remain:LDR, LRD, and DLR.
To these we assign the names In-order, Post-order, and Pre-order

```
treenode= record
{
Typedata;// Type isthe datatype of data,
 treenode*lchild; treenode*rchild;
}
```

```
1.Algorithm In-Order(i)
{
if t!= 0 then
{
In-0rder (t->lchild)
Visit(i);
In-0rder (t->rchild)
}
}
```

Uses of In-order Traversal:

- In the case of binary search trees (BST), In-order traversal gives nodes in non-decreasing order.
- To get nodes of BST in non-increasing order, a variation of In-order traversal where In-order traversal is reversed can be used.
- In-order traversal can be used to evaluate arithmetic expressions stored in expression trees.

```
2.Algorithm Pre-Order(i)
{
if t!= 0 then
{
Visit(i);
Pre-0rder (t->lchild)
Pre-0rder (t->rchild)
}
}
```

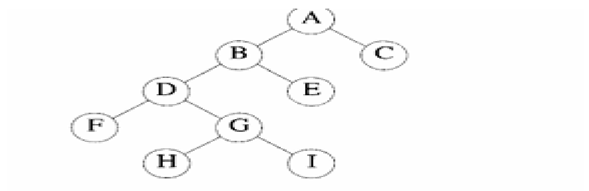
Uses of Preorder Traversal:

- Pre-order traversal is used to create a copy of the tree.
- Pre-order traversal is also used to get prefix expressions on an expression tree.

```
3.Algorithm Post--Order(i)
{
if t!= 0 then
{
Post-0rder (t->lchild)
Post-0rder (t->rchild)
Visit(i);
}
}
```

Uses of Post-order Traversal:

- Post-order traversal is used to delete the tree. See the question for the deletion of a tree for details.
- Post-order traversal is also useful to get the postfix expression of an expression tree.
- Post-order traversal can help in garbage collection algorithms, particularly in systems where manual memory management is used.



In order: FDHGIBEAC
Pre order: ABDFGHIEC
Post order: FHIGDEBCA

Techniques for graphs:

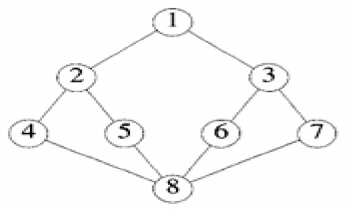
A fundamental problem concerning graphs is the reachability problem. .In its simplest form it requires us to determine whether there exists a path in the given graph G = (V, E) such that this path starts at vertex v and ends at vertex u.
Two search methods for this

1. Breadth first search (BFS) or Level order traversal
2. Depth First Search (DFS)

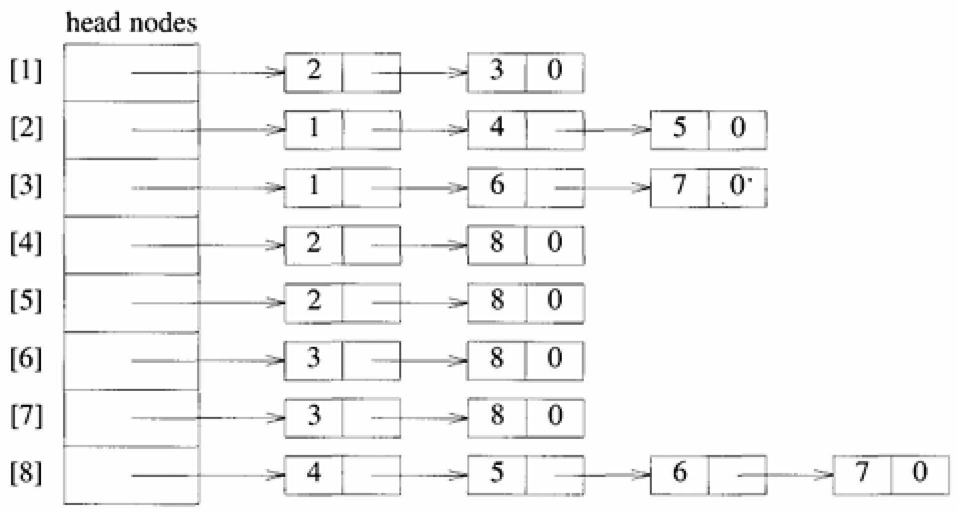
Breadth first search (BFS):

In breadth first search we start at a vertex v and mark it as having been reached(visited). The vertex v is at this time said to be unexplored. A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from v are visited next.These are new unexplored vertices. Vertex v has now been explored.The newly visited vertices haven't been explored and are put onto the end of a list of unexplored vertices. The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left.The list of unexplored vertices operates as a queue and can be represented using any of the standard queue representations

```
Algorithm  BFS(v)
{
    u :=v;
    visited[v]=1;
    repeat
    {
        for all vertices w adjacent from u do
        {
            if (visited[w]= 0) then
            {
                Add w to q;
                visited[w] :=1;
            }
        }
        if q is empty then return;
         Deleteu from q;
    . }
    until(false)
}
```



BFSsequence:1   2   3   4   5   6   7   8   :



(c) Adjacency list for *G*

Figure 6.4 Example graphs and adjacency lists

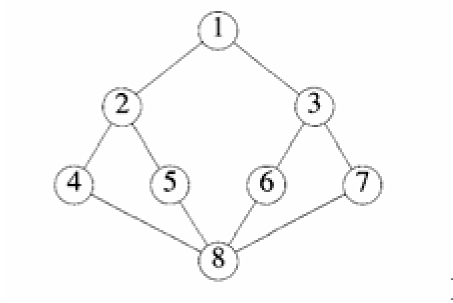
If BFS is used on a connected undirected graph G, then all vertices in G get visited and the graph is traversed .However, if G is not connected, then at least one vertex of G is not visited. A complete traversal of the graph can be made by repeatedly calling BFS each time with a new unvisited starting vertex. The resulting traversal algorithm is known as breadth first

Depth First Search and Traversal:

A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At this time the exploration of the new vertex u begins. When this new vertex has been explored, the exploration of v continues. The search terminates when all reached vertices have been fully explored.

```
Algorithm DFS(v)
{
visited[v] :=1;
for each vertex w adjacent from v do
{
 if (visited[w]= 0) then DFS(w);
}
}
```

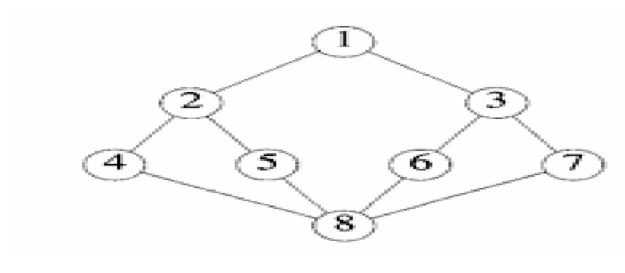
Fig: Algorithm: Depth first search of a graph

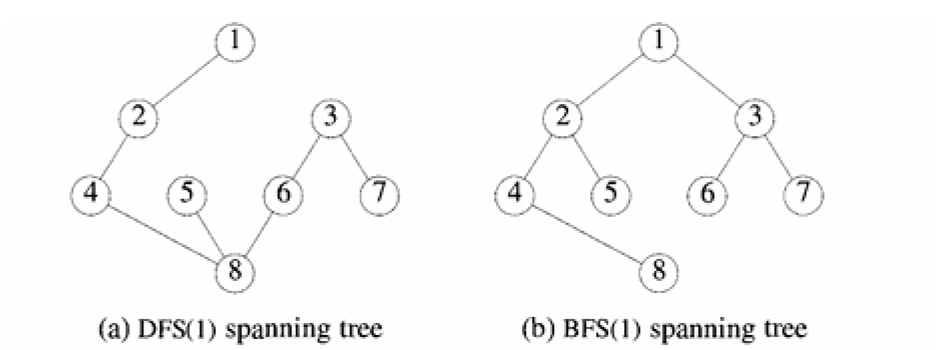


DFS sequence :1 2 4 8 5 6 3 7

Connected Components and Spanning Tress:

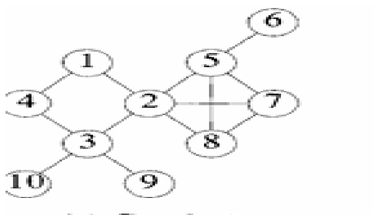
If G is a connected undirected graph, then all vertices of G will get visited on the first call to BFS (Algorithm 6.5).If G is not connected, then at least two calls to BFS will be needed. Hence, BFS can be used to determine whether G is connected.



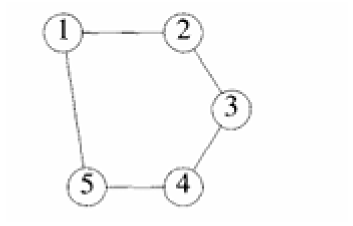

(a) DFS(1) spanning tree (b) BFS(1) spanning tree

Biconnected components and DFS:

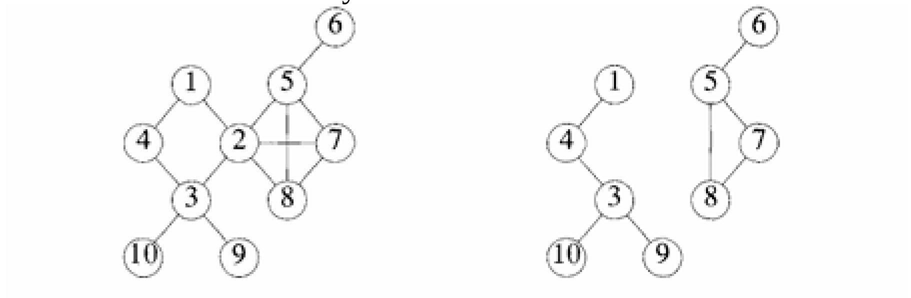
Here we consider Undirected graph. A vertex v in a connected graph G is an articulation point if and only if the deletion of vertex v together with all edges incident to v disconnects the graph into two or more nonempty components
A graph G is biconnected if and only if it contains no articulation points. The graph of


(a) Graph G

is not biconnected.

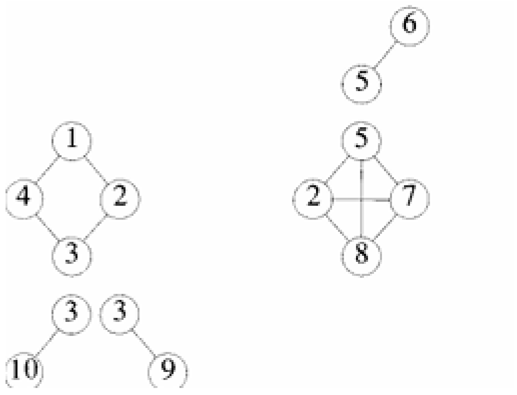


The graph is biconnected. The presence of articulation points in a connected graph can be an undesirable feature in many cases.


(a) Graph G (b) Result of deleting vertex 2

Two biconnected components can have at most one vertex in common and this vertex is an articulation point.

Since every biconnected component of a connected graphG contains at least two vertices (unlessG itself has only onevertex),it follows that the Wj of line5 exists. Example6.4Usingthe above scheme to transform the graph of Figure 6.6(a) into a biconnected graph requires us to add edges(4,10) and (10,9) (corresponding to the articulation point 3),edge(1,5) (corresponding articulation point 2),and edge(6,7) (corresponding to point 5).



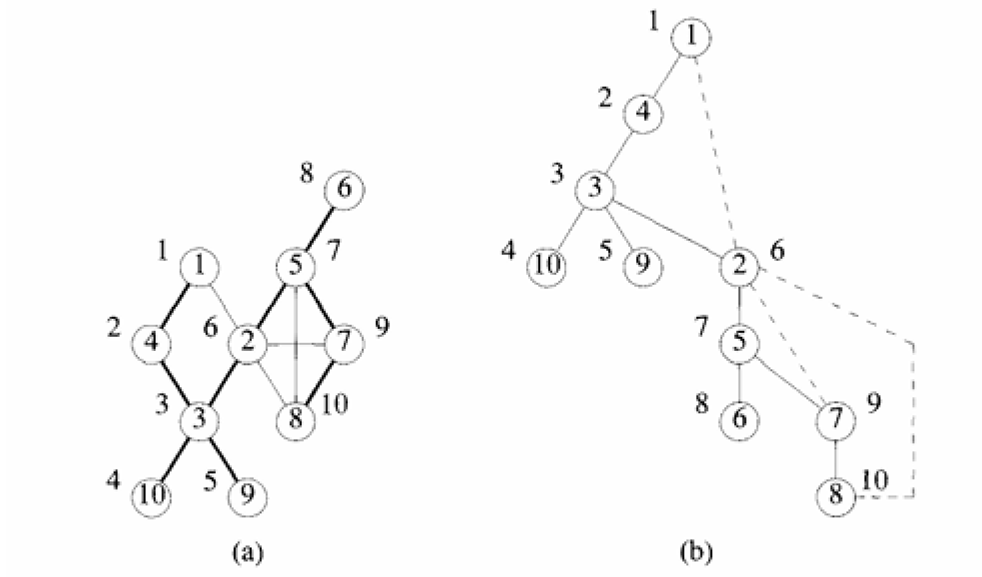


Figure 6.9 A depth first spanning tree of the graph of Figure 6.6(a)

Figure6.9(a) and (b) shows a depth first spanning tree of the graph of the above Figure. In each figure there is a number outside each vertex. These numbers correspond to the order in which a depth first search visits these vertices and are referred to as the depth first numbers (dfns) of the vertex. Thus, dfn[l]= 1,dfn[4] = 2,dfn[6] = 8,and soon. In Figure6.9(b) solid edges form the depth first spanning tree. These edges are called tree edges. Broken edges(i.e., all the remaining edges)are called back edges.

For each vertex it, define L[u] as follows:

L[u] =min {dfn[u], min {L[w} / w is achild of u},min {dfn[w] /(u,w) is a back edge}}

It should be clear that L[u]is the lowest depth first number that can be reached from it using a path of descendents followed by at most one back edge. From the preceding discussion it follows that if u is not the root, then it is an articulation point iff it has a child w such that L[w] >dfn[u].

Example6.5 For the spanning tree of Figure6.9(b) the L values are L[l:10]= {1,1,1,1,6,8,6,6,5,4}. Vertex3 is an articulation point as child10 has L[10] = 4 and dfn[3]=

3.Vertex2 is an articulation point as child 5 has L[5]= 6 and dfn[2] = 6.The only other articulation point is vertex5; child6 has L[6]= 8 and dfn[5] = 7.

Sets and Disjoint set Union: Introduction, Union and Find operations.

Introduction :

Set:Collection of finite set of elements
Example : S= {1,2,3,4,5}
Disjoint set: if Si and Sj are two sets, then there is no element that is in both Si and Sj .For example,when n = 10,the elements can be partitioned into three disjoint sets, ,Si= {1,7,8,9}, and Sj= {3,4,6}.
Figure2.17shows one possible sets. In this representation, S2= {2,5,10}, representation for these each set is represented as a tree. Notice that for each set we have linked the nodes from the children to the parent,rather than our usual method of linking from the parent to the children. There as on for this change in linkage becomes apparent when we discuss the implementation

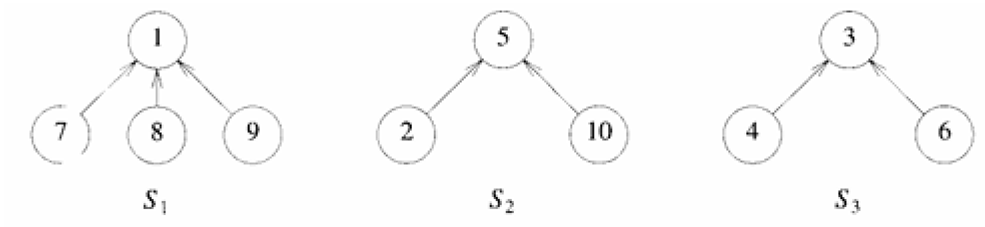


Figure 2.17 Possible tree representation of sets

The operations we wish to perform on these sets are:
1. **Disjoint set union**: If Si and Sj are two disjoint sets, then their union SiUSj= all elements x such that x is in Si or Sj.Thus S1US2 = {1, 7, 8,9,2,5,10}. Since we have assumed that all sets are disjoint, we can assume that following the union of Si and Sj,the sets Si and Sj do not exist independently; that is,they are replaced by SiUSj in the collection of sets.
2.**Find(i)**:Given the element i, find the set containing i. Thus,4 is in set S3,and 9 is in set S.

Union and Find Operations:

Let us consider the union operation first. Suppose that we wish to obtain the union of Si and S2 (from Figure2.17). Since we have linked the nodes from children to parent, we simply make one of the trees a sub tree of the other.S1US2 could then have one of the representations

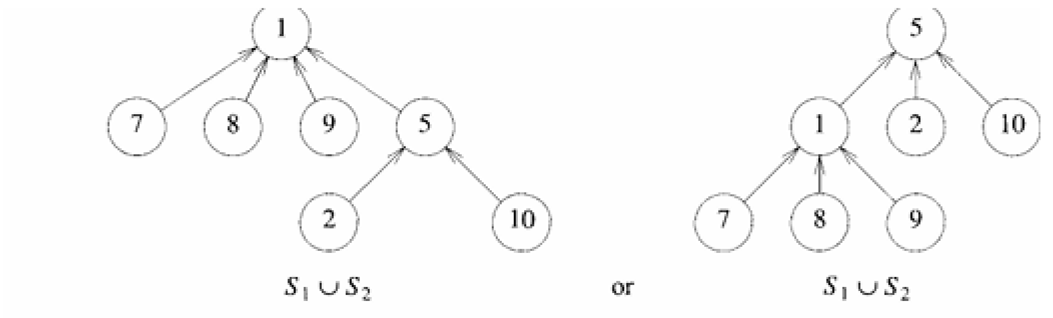


Figure 2.18 Possible representations of $S_1 \cup S_2$

Field of one of the roots to the other root.This can be accomplished easily if, with each set name,we keep a pointer to the root of the tree representing that set.If, in addition,each root has a pointer to the set name,then to determine which set an element is currently in, we follow parent links to the root of its tree and use the pointer to the set name.The data representation for Si,S2,and S3may then take the form shown in Figure2.19. In presenting the union and find algorithms, we ignore the set names and identify just by the roots of the trees representing them.Thissimplifie

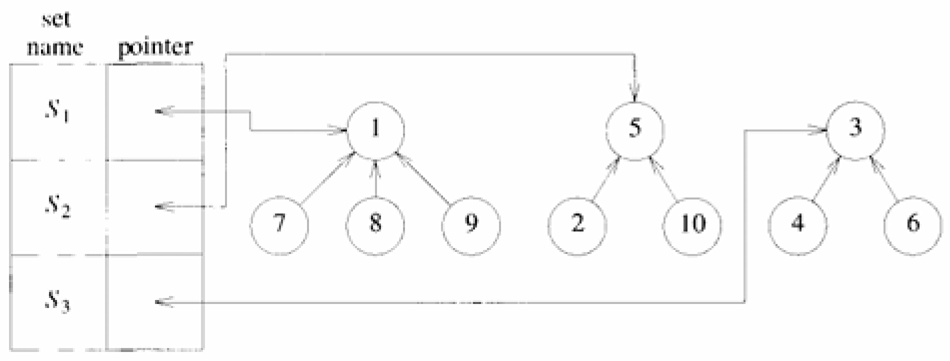


Figure 2.19 Data representation for S_1, S_2, and S_3

Element i is in a tree with root j, and j has a pointer to entry k in the set name table,then the set name is just name[k].If we wish to unite sets Si, and Sj,then we wish to unite the trees with roots FindPointer(S'j) and FindPointer(/Sj).Here FindPointer is a function that takes a set name and determines the root of the tree that represents it. This is done by an examination of the [set name,pointer]table.In many applications the set name is just the element at the root.The operation of Find(i)now becomes: Determine the root of the tree containing element i.The function Union(i, j) requires two trees with roots i and j be joined. Also to simplify, assume that the set elements are the numbers1through n. Since the set elements are numbered1through n, we represent the tree nodes using an array p[1:n],where n is the maximum number of elements. The ith element of this array represents the tree node that contains element i. This array element gives the parent pointer of the corresponding tree node.Figure2.20showsthis representation of the sets Si,S2,and S3of Figure2.17

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

Figure 2.20 Array representation of S_1, S_2, and S_3 of Figure 2.17

We can now implement Find(i) by following the indices,starting at i until we reach a node with parent value .For example,Find(6) starts at 6 and then moves to 6'sparent,3.Since p[3]is negative,we have reached the root.The operation Union(i,j) is equally simple.We pass in two trees with roots % and j. Adopting the convention that the first tree becomes a sub tree of the second, the statement p[i]:=j; accomplishes the union.

```
Algorithm SimpleUnion(i, j)
{
    p[i] := j;
}

Algorithm SimpleFind(i)
{
    while (p[i] ≥ 0) do i := p[i];
    return i;
}
```

Union(l,2), Union(2,3), Union(3,4), Union(4,5),... , Union(n)=(1,n) Find(l),Find{2),..., Find{n) This sequence results in the degenerate tree of Figure2.21.

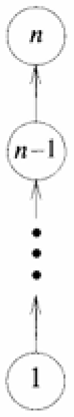


Figure 2.21 Degenerate tree

Definition:[Weighting rule for Union(i,j)]If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i; otherwise make i the parent of j

When we use the weighting rule to perform the sequence of set unions given before,we obtain the trees of Figure2.22. In this figure,the unions have been modified so that the input parameter values correspond to the roots of the trees to be combined. To implement the weighting rule,we need to know how many nodes there are in every tree. To do this easily, we maintain a count field in the root of every tree.If i is a root node,then count[i] equals the number of nodes in that tree. Since all nodes other than the roots of trees have a positive number in the p field,we can maintain the count in the p field of the roots as a negative number

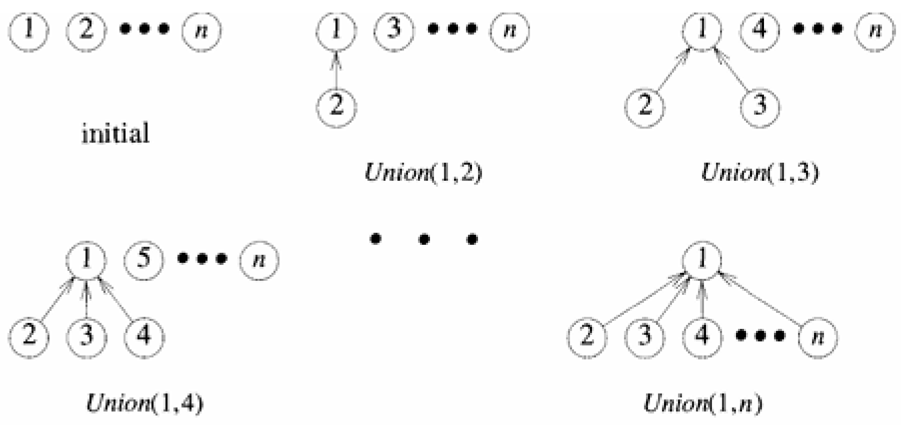


Figure 2.22 Trees obtained using the weighting rule

```
Algorithm WeightedUnion($i, j$)
// Union sets with roots $i$ and $j$, $i \neq j$, using the
// weighting rule. $p[i] = -count[i]$ and $p[j] = -count[j]$.
{
    $temp := p[i] + p[j]$;
    if ($p[i] > p[j]$) then
    { // $i$ has fewer nodes.
        $p[i] := j$; $p[j] := temp$;
    }
    else
    { // $j$ has fewer or equal nodes.
        $p[j] := i$; $p[i] := temp$;
    }
}
```

Algorithm 2.14 Union algorithm with weighting rule

Definition:[Collapsingrule]:If j is a node on the path from i to its root and p[i] root[i], then set p[j] to root[i]. CollapsingFind(Algorithm 2.15) incorporates the collapsing rule

Consider the tree created by Weighted Union on the sequence of unions of Example2.4.Now process the following eight finds: Find(8),Find(8),..., Find(8) If Simple Find is used,each Find(8) requires going up three parent link fields for a total of 24 moves to
Process all eight finds. When CollapsingFind is used, the first Find(8) requires going up three links and then resetting two links. Note that even though only two parent links need to be reset,CollapsingFind will reset three(the parent of 5 is reset to 1).Each of the remaining

seven finds requires going up only one link field.The total cost is now only 13 moves.

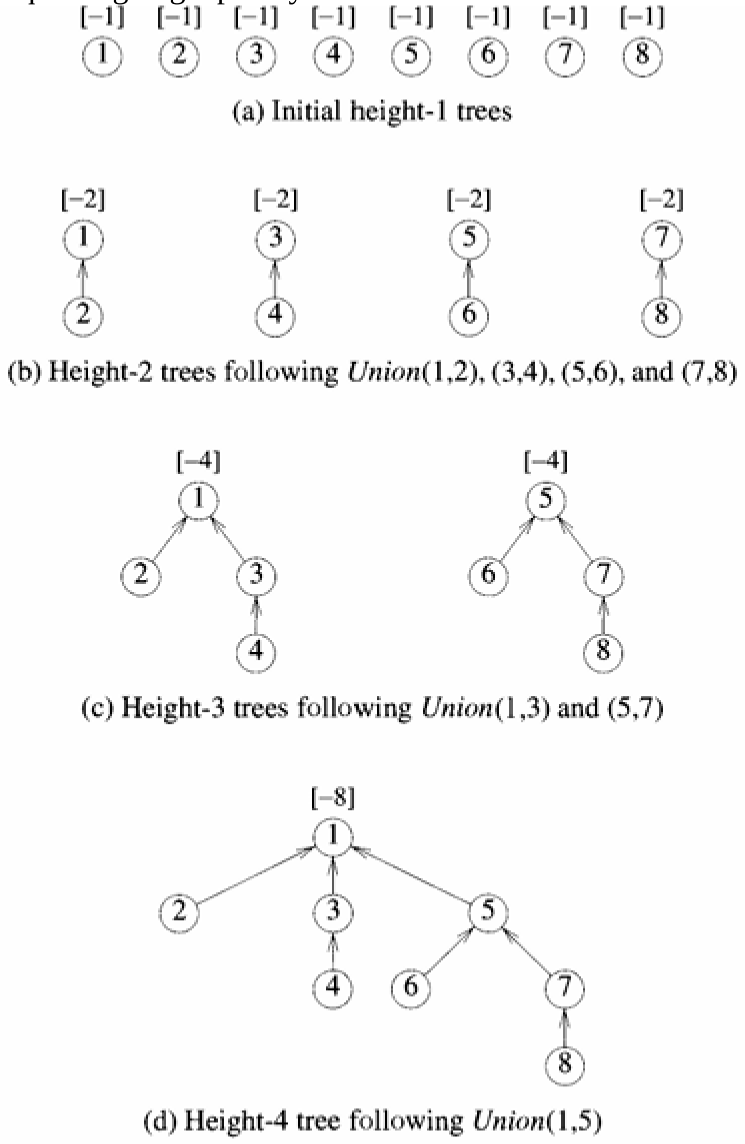

[−1] [−1] [−1] [−1] [−1] [−1] [−1] [−1]
(a) Initial height-1 trees
[−2] [−2] [−2] [−2]
(b) Height-2 trees following *Union*(1,2), (3,4), (5,6), and (7,8)
[−4] [−4]
(c) Height-3 trees following *Union*(1,3) and (5,7)
[−8]
(d) Height-4 tree following *Union*(1,5)

Figure 2.23 Trees achieving worst-case bound

```
Algorithm CollapsingFind(i)
// Find the root of the tree containing element i. Use the
// collapsing rule to collapse all nodes from i to the root.
{
    r := i;
    while (p[r] > 0) do r := p[r]; // Find the root.
    while (i ≠ r) do  // Collapse nodes from i to root r.
    {
        s := p[i]; p[i] := r; i := s;
    }
    return r;
}
```

Algorithm 2.15 Find algorithm with collapsing rule