DATA STRUCTURES FOR DISJOINT SETS

An important application of the tree is the representation of sets, where "n" distinct elements are needed to be grouped into a number of disjoint sets.

A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots S_k\}$ of disjoint dynamic sets. Each set is identified by a representative, which is some member of the set. If we have two sets Sx and Sy, $x \neq y$, such that Sx = (3, 4, 5, 6, 7) and Sy = (1, 2) then these sets are called disjoint sets as there is no element which is common in both sets.

DISJOINT-SET OPERATIONS

In the dynamic-set implementations, an object represents each element of a set. Letting x denote an object, we wish to support the following operations.

1. MAKE-SET(x)

creates a new set whose only member (and thus representative) is pointed to by x. Since the sets are disjoint, we require that x not already be in a set.

2. UNION(x,y)

unites the dynamic sets that contain x and y, say Sx and Sy into a new set that is the union of these two sets. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is some member of Sx U Sy Since we require the sets in the collection to be disjoint, we "destroy" sets Sx and Sy removing them from the collection S.

3. FIND-SET(x)

returns a pointer to the representative of the (unique) set containing x

The running times of disjoint-set data structures in terms of two parameters: n, the number of MAKE-SET operations, and m and the total number of MAKE-SET, UNION, and FIND-SET operations. Since the sets are disjoint, each UNION operation reduces the number of sets by one. After n-1 UNION operation, therefore, only one set remains. The number of UNION operations is thus at most n-1. Note also that since the MAKE-SET operations are included in the total number of operations m, we have m >= n.

UNION-FIND ALGORITHM

A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

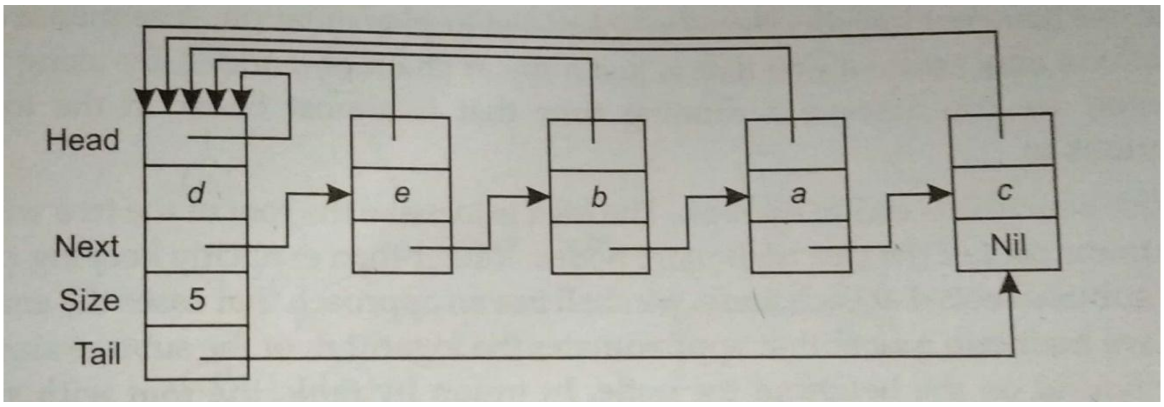
- Find. Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
- Union. Combine or merge two sets into a single set.

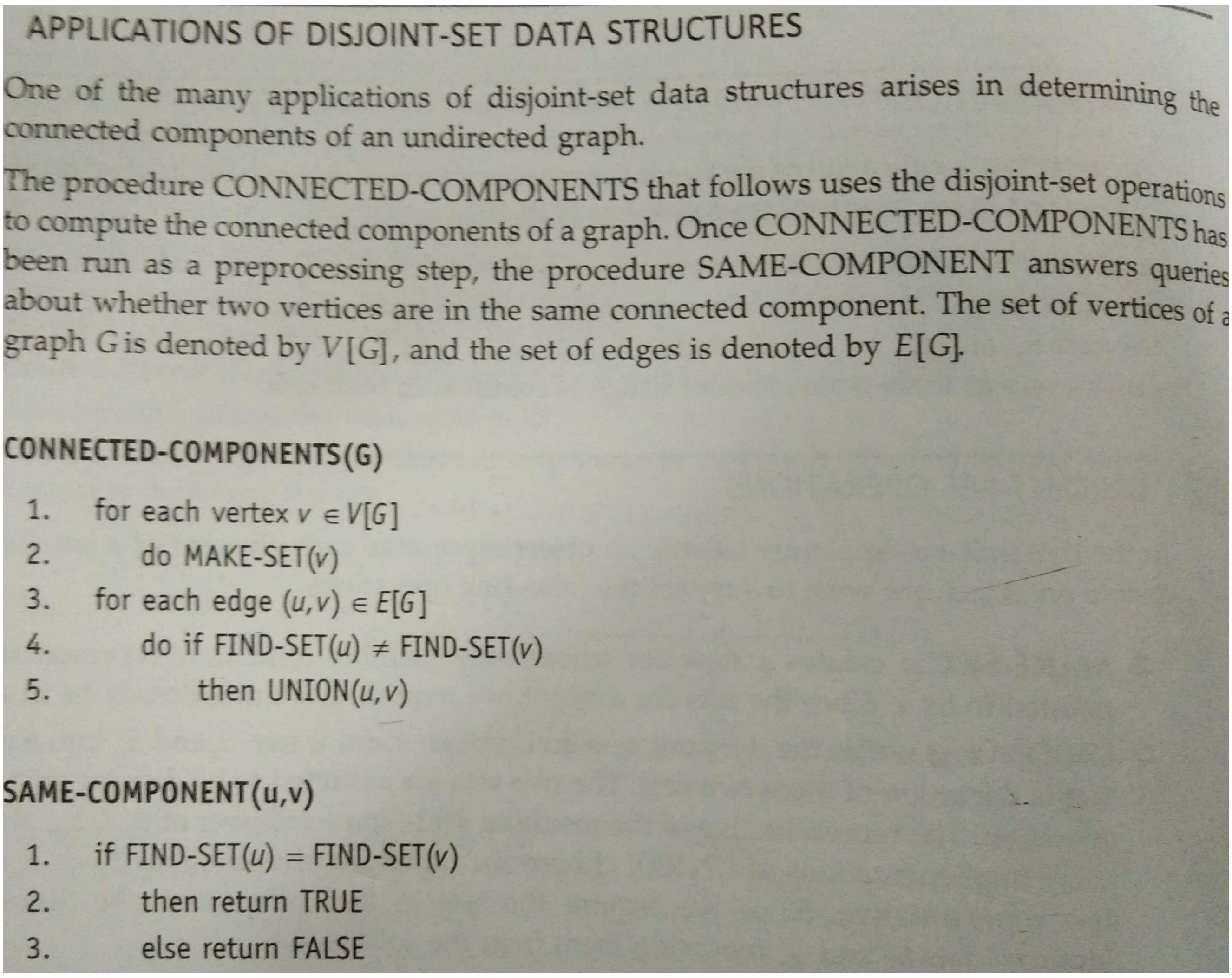
Because it supports these two operations, a disjoint-set data structure is sometimes called a merge-find set.

LINKED-LIST REPRESENTATION OF DISJOINT SETS

A simple way to implement a disjoint-set data structure is to represent each set by a linked list. The first object in each linked list serves as its set's representative. Each object in the linked list contains a set member, a pointer to the object containing the next set member, and a pointer back to the representative. Within each linked list, the objects may appear in any order (subject to our assumption that the first object in each list is the representative)

With this linked-list representation, both MAKE-SET and FIND-SET are easy, requiring O(1) time. To carry out MAKE-SET(x), we create a new linked list whose only object is x. For FIND-SET(x), we just return the pointer from x back to the representative.





APPLICATIONS OF DISJOINT-SET DATA STRUCTURES

One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph.

The procedure CONNECTED-COMPONENTS that follows uses the disjoint-set operations to compute the connected components of a graph. Once CONNECTED-COMPONENTS has been run as a preprocessing step, the procedure SAME-COMPONENT answers queries about whether two vertices are in the same connected component. The set of vertices of a graph G is denoted by $V[G]$, and the set of edges is denoted by $E[G]$.

```
CONNECTED-COMPONENTS(G)
1.   for each vertex $v \in V[G]$
2.        do MAKE-SET($v$)
3.   for each edge $(u,v) \in E[G]$
4.        do if FIND-SET($u$) $\neq$ FIND-SET($v$)
5.              then UNION($u,v$)
```

```
SAME-COMPONENT(u,v)
1.   if FIND-SET($u$) = FIND-SET($v$)
2.        then return TRUE
3.        else return FALSE
```

Example:

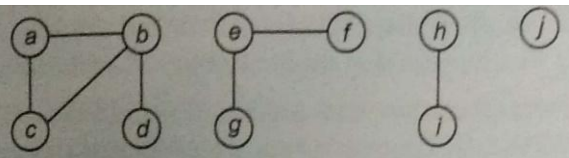


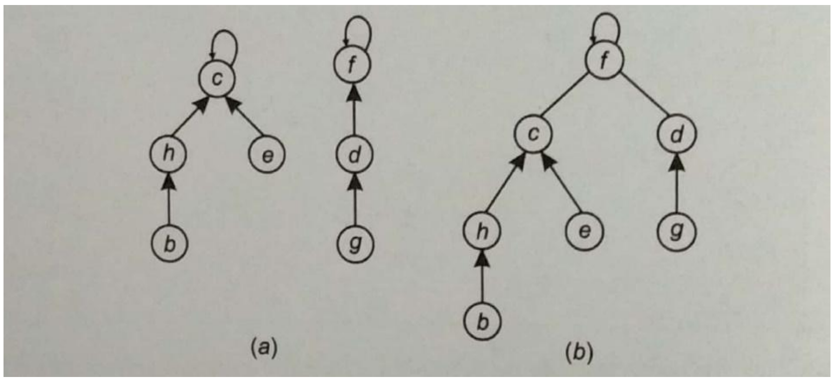
Figure 20.1 A graph with four connected components : $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$, and $\{j\}$. The collection of disjoint sets after each edge is processed.

Edge processed	*Collection of disjoint sets*									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

The procedure CONNECTED-COMPONENTS initially places each vertex v in its own set. Then, for each edge (u,v), it unites the sets containing u and v. After all the edges are processed, two vertices are in the same connected component if and only if the corresponding objects are in the same set. Thus, CONNECTED-COMPONENTS computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component.

DISJOINT-SET FORESTS

In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set. In a disjoint-set forest, each member points only to its parent. The root of each tree contains the representative and is its own parent.



The above figure shows a disjoint-set forest. (a)Two trees representing the two sets of above Figure. The tree on the left represents the set {b, c, e, h}, with c as the representative, and the tree on the right represents the set {d, f, g}, with f as the representative. (b)The result of UNION (e, g).

We perform the three disjoint-set operations as follows.

- A MAKE-SET operation simply creates a tree with just one node.
- We perform a FIND-SET operation by chasing parent pointers until we find the root of the tree. The nodes visited on this path toward the root constitute the find path.
- A UNION operation causes the root of one tree to point to the root of the other.

Heuristics to improve the running time

A sequence of n-1 UNION operations may create a tree that is just a linear chain of n nodes. By using **two heuristic**s, we can achieve a running time that is almost linear in the total number of operations m.

1. union by rank

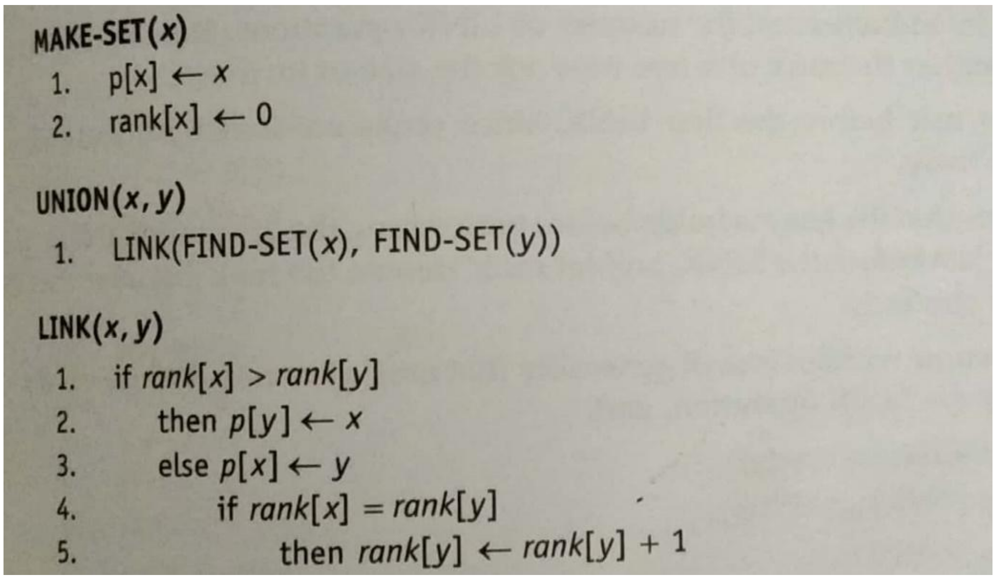
   The idea is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. For each node, we maintain a rank that approximates the logarithm of the subtree size and is also an upper bound on the height of the node. **In union by rank, the root with smaller rank is made to point to the root with larger rank during a UNION operation.**

2. path compression

   We use it during FIND-SET operations to make **each node on the find path point directly to the root.** Path compression does not change any ranks

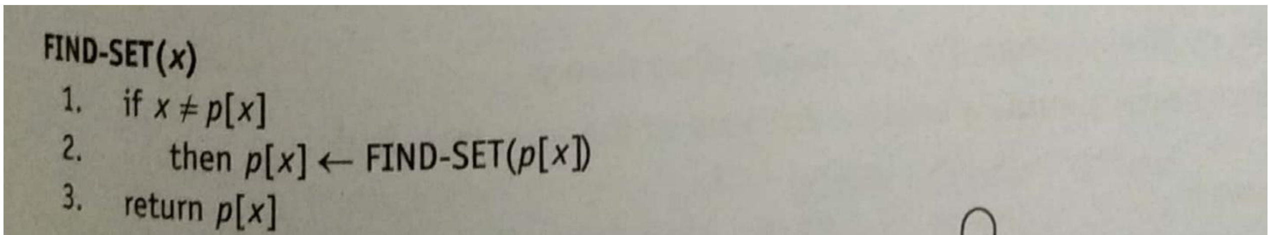
To implement a disjoint-set forest with the union-by-rank heuristic, we must keep track of ranks.

- With each node x, we maintain the integer value rank [x], which is an upper bound on the height of x (the number of edges in the longest path between x and a descendant leaf).
- When a singleton set is created by MAKE-SET, the initial rank of the single node in the corresponding tree is 0.
- Each FIND-SET operation leaves all ranks unchanged.
- When applying UNION to two trees, we make the root of higher rank the parent of the root of lower rank.
- In case of a tie, we arbitrarily choose one of the roots as the parent and increment its rank, we assign the parent of node x by p[x].
- The LINK procedure, a subroutine called by UNION, takes pointers to two roots as inputs.

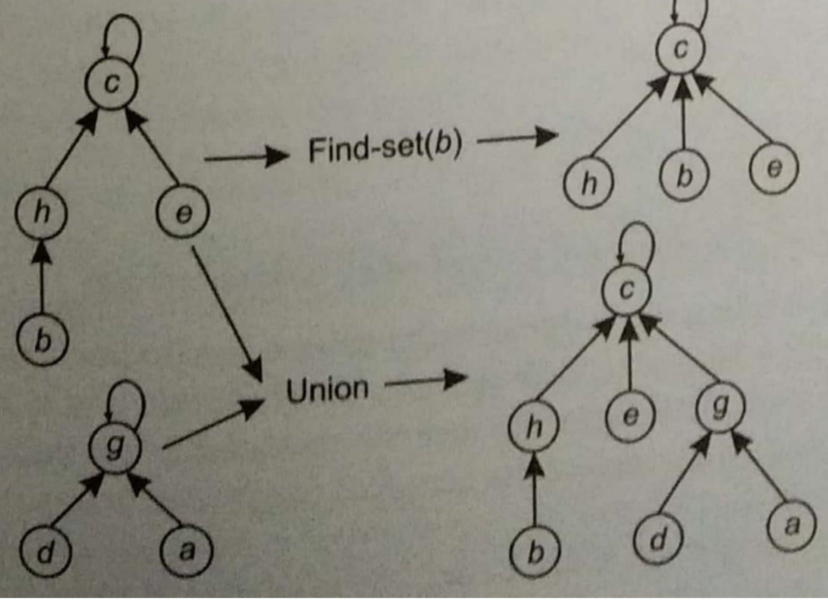


```
MAKE-SET(x)
  1.  p[x] ← x
  2.  rank[x] ← 0

UNION(x, y)
  1.  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
  1.  if rank[x] > rank[y]
  2.      then p[y] ← x
  3.      else p[x] ← y
  4.          if rank[x] = rank[y]
  5.              then rank[y] ← rank[y] + 1
```

The **FIND-SET procedure with path compression** is as follows:



```
FIND-SET(x)
  1.  if x ≠ p[x]
  2.      then p[x] ← FIND-SET(p[x])
  3.  return p[x]
```


c
h
e
b
Find-set(b)
c
h
b
e
g
d
a
Union
c
h
e
g
b
d
a