Figure 5.4 Kruskal's minimum spanning tree algorithm.

procedure kruskal(G, w)
Input: A connected undirected graph $G = (V, E)$ with edge weights $w_e$
Output: A minimum spanning tree defined by the edges $X$

for all $u \in V$:
    makeset(u)

$X = {}$
Sort the edges $E$ by weight
for all edges $(u, v) \in E$, in increasing order of weight:
    if find($u$) $\neq$ find($v$):
        add edge $(u, v)$ to $X$
        union($u, v$)

5.1.4 A data structure for disjoint sets

Union by rank

One way to store a set is as a directed tree (Figure 5.5). Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree. This root element is a convenient representative, or name, for the set. It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

In addition to a parent pointer $\pi$, each node also has a rank that, for the time being, should be interpreted as the height of the subtree hanging from that node.

procedure makeset($x$)
    $\pi(x) = x$
    rank($x$) = 0

function find($x$)
    while $x \neq \pi(x)$: $x = \pi(x)$

Figure 5.5 A directed-tree representation of two sets $\{B, E\}$ and $\{A, C, D, F, G, H\}$.
return $x$

As can be expected, makeset is a constant-time operation. On the other hand, find follows parent pointers to the root of the tree and therefore takes time proportional to the height of the tree. The tree actually gets built via the third operation, union, and so we must make sure that this procedure keeps trees shallow.

Merging two sets is easy: make the root of one point to the root of the other. But we have a choice here. If the representatives (roots) of the sets are $r_x$ and $r_y$, do we make $r_x$ point to $r_y$ or the other way around? Since tree height is the main impediment to computational efficiency, a good strategy is to make the root of the shorter tree point to the root of the taller tree. This way, the overall height increases only if the two trees being merged are equally tall. Instead of explicitly computing heights of trees, we will use the rank numbers of their root nodes—which is why this scheme is called union by rank.

```plaintext
procedure union($x$, $y$)
  $r_x$ = find($x$)
  $r_y$ = find($y$)
  if $r_x$ = $r_y$: return
  if rank($r_x$) > rank($r_y$):
    $\pi(r_y) = r_x$
  else:
    $\pi(r_x) = r_y$
    if rank($r_x$) = rank($r_y$): rank($r_y$) = rank($r_y$) + 1
```

See Figure 5.6 for an example.

By design, the rank of a node is exactly the height of the subtree rooted at that node. This means, for instance, that as you move up a path toward a root node, the rank values along the way are strictly increasing.

**Property 1** For any $x$, rank($x$) < rank($\pi(x)$).

A root node with rank $k$ is created by the merger of two trees with roots of rank $k - 1$. It follows by induction (try it!) that

**Property 2** Any root node of rank $k$ has at least $2^k$ nodes in its tree.

This extends to internal (nonroot) nodes as well: a node of rank $k$ has at least $2^k$ descendants. After all, any internal node was once a root, and neither its rank nor its set of descendants has changed since then. Moreover, different rank-$k$ nodes cannot have common descendants, since by Property 1 any element has at most one ancestor of rank $k$. Which means

**Property 3** If there are $n$ elements overall, there can be at most $n/2^k$ nodes of rank $k$.

This last observation implies, crucially, that the maximum rank is $\log n$. Therefore, all the trees have height $\leq \log n$, and this is an upper bound on the running time of find and union.
**Figure 5.6 A sequence of disjoint-set operations. Superscripts denote rank.**

After `makeset(A), makeset(B), …; makeset(G):

```
A⁰   B⁰   C⁰   D⁰   E⁰   F⁰   G⁰
```

After `union(A, D), union(B, E), union(C, F):

```
D¹   E¹   F¹   G¹
   A⁰   B⁰   C⁰
```

After `union(C, G), union(E, A):

```
D²   E¹   F¹
   A⁰   C⁰   G⁰
   B⁰
```

After `union(B, G):

```
D²   E¹   F¹
   A⁰   B⁰   G¹
   C⁰   C⁰
```

**Path compression**

With the data structure as presented so far, the total time for Kruskal’s algorithm becomes $O(|E| \log |V|)$ for sorting the edges (remember, $\log |E| \approx \log |V|$) plus another $O(|E| \log |V|)$ for the `union` and `find` operations that dominate the rest of the algorithm. So there seems to be little incentive to make our data structure any more efficient.

But what if the edges are given to us sorted? Or if the weights are small (say, $O(|E|)$) so that sorting can be done in linear time? Then the data structure part becomes the bottleneck, and it is useful to think about improving its performance beyond $\log n$ per operation. As it turns out, the improved data structure is useful in many other applications.

But how can we perform `union`’s and `find`’s faster than $\log n$? The answer is, by being a