**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 ∈ V:
   makeset(u)

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

### 5.1.4 A data structure for disjoint sets

#### Union by rank

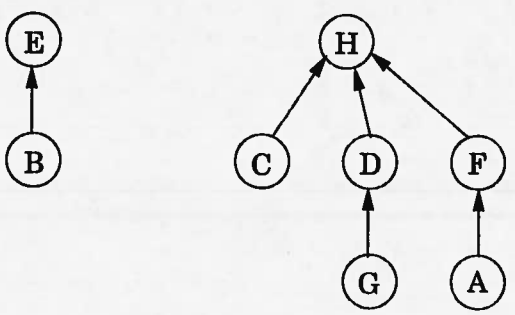
One way to store a set is as a directed tree (Figure 5.5). Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree. This root element is a convenient *representative*, or *name*, for the set. It is distinguished 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)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x):  x = π(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*.

```
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):
    π(r_y) = r_x
else:
    π(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

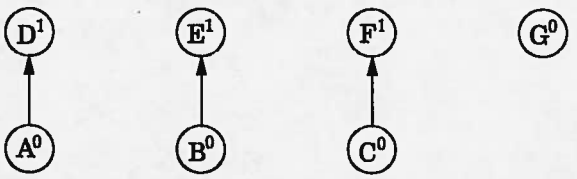
**Property 3** *If there are $n$ elements overall, there can be at most $n/2^k$ nodes of rank $k$.*

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`.

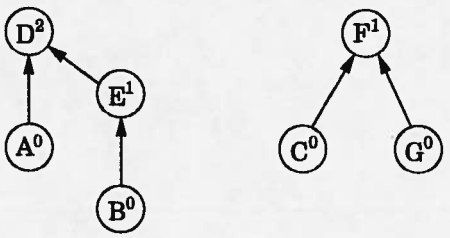
**Figure 5.6** A sequence of disjoint-set operations. Superscripts denote rank.

After `makeset`$(A)$, `makeset`$(B)$, ..., `makeset`$(G)$:

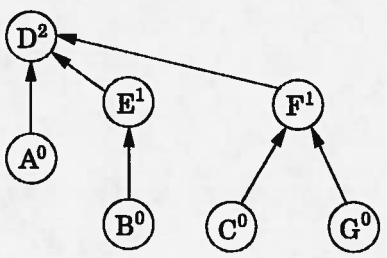

After `union`$(A, D)$, `union`$(B, E)$, `union`$(C, F)$:




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




After `union`$(B, G)$:



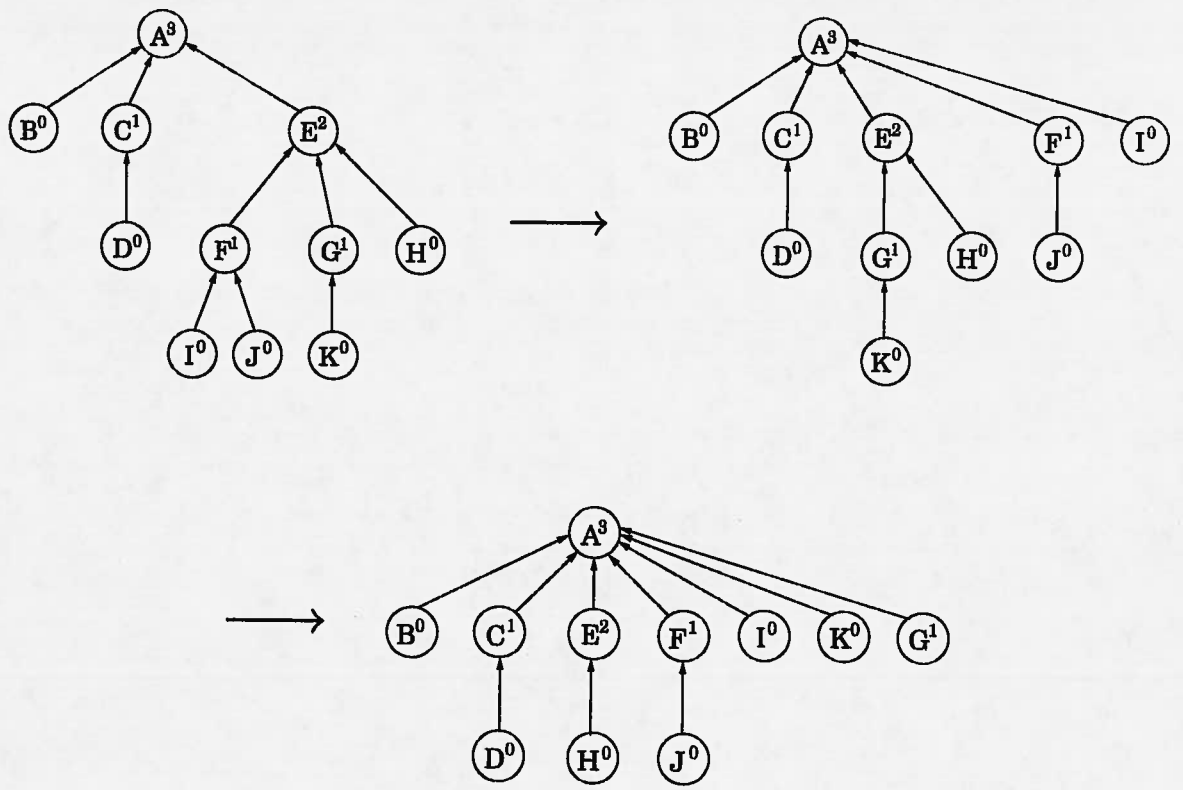

### Path compression

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

**Figure 5.7** The effect of path compression: `find`($I$) followed by `find`($K$).




little more careful to maintain our data structure in good shape. As any housekeeper knows, a little extra effort put into routine maintenance can pay off handsomely in the long run, by forestalling major calamities. We have in mind a particular maintenance operation for our union-find data structure, intended to keep the trees short— during each `find`, when a series of parent pointers is followed up to the root of a tree, we will change all these pointers so that they point directly to the root (Figure 5.7). This *path compression* heuristic only slightly increases the time needed for a `find` and is easy to code.

```
function find(x)
if x ≠ π(x): π(x) = find(π(x))
return π(x)
```

The benefit of this simple alteration is long-term rather than instantaneous and thus necessitates a particular kind of analysis: we need to look at *sequences* of `find` and `union` operations, starting from an empty data structure, and determine the average time per operation. This *amortized cost* turns out to be just barely more than $O(1)$, down from the earlier $O(\log n)$.

Think of the data structure as having a "top level" consisting of the root nodes, and below it, the insides of the trees. There is a division of labor: `find` operations (with or without path

compression) only touch the insides of trees, whereas `union`'s only look at the top level. Thus path compression has no effect on `union` operations and leaves the top level unchanged.

We now know that the ranks of root nodes are unaltered, but what about *nonroot* nodes? The key point here is that once a node ceases to be a root, it never resurfaces, and its rank is forever fixed. Therefore the ranks of all nodes are unchanged by path compression, even though these numbers can no longer be interpreted as tree heights. In particular, properties 1–3 (from page 139) still hold.

If there are $n$ elements, their rank values can range from 0 to $\log n$ by Property 3. Let's divide the nonzero part of this range into certain carefully chosen intervals, for reasons that will soon become clear:

$$\{1\}, \{2\}, \{3,4\}, \{5,6,\ldots,16\}, \{17,18,\ldots,2^{16}=65536\}, \{65537,65538,\ldots,2^{65536}\}, \ldots$$

Each group is of the form $\{k+1, k+2, \ldots, 2^k\}$, where $k$ is a power of 2. The number of groups is $\log^* n$, which is defined to be the number of successive $\log$ operations that need to be applied to $n$ to bring it down to 1 (or below 1). For instance, $\log^* 1000 = 4$ since $\log\log\log\log 1000 \leq 1$. In practice there will just be the first five of the intervals shown; more are needed only if $n \geq 2^{65536}$, in other words never.

In a sequence of `find` operations, some may take longer than others. We'll bound the overall running time using some creative accounting. Specifically, we will give each node a certain amount of pocket money, such that the total money doled out is at most $n \log^* n$ dollars. We will then show that each `find` takes $O(\log^* n)$ steps, plus some additional amount of time that can be "paid for" using the pocket money of the nodes involved—one dollar per unit of time. Thus the overall time for $m$ `find`'s is $O(m \log^* n)$ plus at most $O(n \log^* n)$.

In more detail, a node receives its allowance as soon as it ceases to be a root, at which point its rank is fixed. If this rank lies in the interval $\{k+1, \ldots, 2^k\}$, the node receives $2^k$ dollars. By Property 3, the number of nodes with rank $> k$ is bounded by

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}.$$

Therefore the total money given to nodes in this particular interval is at most $n$ dollars, and since there are $\log^* n$ intervals, the total money disbursed to all nodes is $\leq n \log^* n$.

Now, the time taken by a specific `find` is simply the number of pointers followed. Consider the ascending rank values along this chain of nodes up to the root. Nodes $x$ on the chain fall into two categories: either the rank of $\pi(x)$ is in a higher interval than the rank of $x$, or else it lies in the same interval. There are at most $\log^* n$ nodes of the first type (do you see why?), so the work done on them takes $O(\log^* n)$ time. The remaining nodes—whose parents' ranks are in the same interval as theirs—have to pay a dollar out of their pocket money for their processing time.

This only works if the initial allowance of each node $x$ is enough to cover all of its payments in the sequence of `find` operations. Here's the crucial observation: each time $x$ pays a dollar, its parent changes to one of higher rank. Therefore, if $x$'s rank lies in the interval $\{k+1, \ldots, 2^k\}$, it has to pay at most $2^k$ dollars before its parent's rank is in a higher interval; whereupon it never has to pay again.