1. *Precompute the occupation order.* One way to occupy the sites is to choose sites at random until we find an unoccupied site. However, this procedure will become inefficient when most of the sites are already occupied. Instead, we store the order in which the sites are to be occupied in `order[]` and generate the order by randomly permuting the integers from 0 to $N-1$ in method `setOccupationOrder`. For example, `order[0]` $= 2$ means that we occupy site 2 first.
2. *Add sites according to predetermined order.* When a new site is added, we check all its neighbors to determine if the new site is an isolated cluster (all neighbors empty) or if it joins one or more existing clusters.
3. *Determine the clusters.* The clusters are organized in a tree-like structure, with one site of each cluster designated as the *root*. All sites in a given cluster, other than the root, point to another site in the same cluster, so that the root can be reached by recursively following the pointers. The "pointers"[1] are stored in the `parent` array. To join two clusters, we add a pointer from the root of the smaller cluster to the root of the larger one.

In the following, we use `order` $= \{2, 6, 8, 4, 5, \ldots\}$ to illustrate the method (see Figure 12.9.

(i) Because `order[0]` $= 2$, we first occupy site 2 and set `parent[2]` $= -1$. The negative sign distinguishes site 2 as a root. The size of the cluster is stored as `-parent[root]`. In this case `-parent[2]` $= 1$, and because no other sites are occupied, there is no possibility of merging clusters.

(ii) For our example, `order[1]` $= 6$, and we initially set `parent[6]` $= -1$. We then consider the neighbor sites of site 6. Sites 5 (left), 7 (right), and 10 (up) are unoccupied (`parent[5]` = `parent[7]` = `parent[10]` = `EMPTY`), but site 2 (down) is occupied. Hence, we need to merge the two clusters, and we set `parent[6]` $= 2$ and `parent[2]` $= -2$. That is, the value of `parent[6]` points to site 2, and the value of `parent[2]` indicates that site 2 is the root of a cluster of size 2.

(iii) The next sites to be occupied are 8 and 4, as shown in Figure 12.9. These two sites form a size 2 cluster as before. We have `parent[8]` $= -1$ and then `parent[4]` $= 8$ and `parent[8]` $= -2$. We see that the value of each element of the `parent` array has three functions: nonroot occupied sites contain the index for the site's parent in the cluster tree; root sites are negative and equal to the negative of the number of sites in the cluster; and unoccupied sites have the value `EMPTY`.

(iv) We next add site 5 and set `parent[5]` $= -1$. From Figure 12.9 we see that we have to merge two clusters. We (arbitrarily) check the left neighbor of site 5 first, and hence we first merge the cluster of size 1 at site 5 with the cluster at site 8 of size 2. Hence, we set `parent[5]` $= 8$ and `parent[8]` $= -3$. We next check the right neighbor of site 5 and find that we need to merge two clusters again with root sites at 8 and 2. Because the cluster at site 8 is bigger, we set `parent[2]` $= 8$ and `parent[8]` $= -5$.

[1]We use the term "pointer" as it is used by Newman and Ziff, that is, a link to an array index. A true pointer stores a memory address and does not exist in Java.

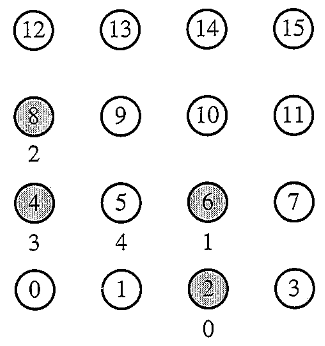

Figure 12.9 Illustration of the Newman–Ziff algorithm. The `order` array is given by {2, 6, 8, 4, 5, ...}; the number below a site denotes the order in which that site was occupied. When site 5 is occupied, we have to merge the two clusters as explained in the text.

Listing 12.2 Implementation of Newman–Ziff algorithm for identifying clusters.

```
package org.opensourcephysics.sip.ch12;
public class Clusters {
   static private final int EMPTY = Integer.MIN_VALUE;
   public int L;                   // linear dimension of lattice
   public int N;                   // N = L*L
   public int numSitesOccupied;    // number of occupied lattice sites
   public int[] numClusters;       // number of clusters of size s, n_s

   // secondClusterMoment stores sum s^2 n_s, where sum is over
   // all clusters (not counting spanning cluster)
   // first cluster moment, sum s n_s equals numSitesOccupied.
   // mean cluster size S defined as
   // S = secondClusterMoment/numSitesOccupied
   private int secondClusterMoment;

   // spanningClusterSize, number of sites in a spanning cluster; 0 if
   // it doesn't exist. Assume there is at most one spanning cluster
   private int spanningClusterSize;

   // order[n] gives index of nth occupied site; contains all numbers
   // from [1...N], but in random order. For example, order[0] = 3
   // means we will occupy site 3 first. An alternative to using order
   // array is to choose sites at random until we find an unoccupied
   // site.
   private int[] order;

   // parent[] array serves three purposes: stores the cluster size
   // when the site is the root.  Otherwise, it stores the index of
   // the site's "parent" or is EMPTY. The root is found from an
   // occupied site by recursively following the parent array.
   // Recursion terminates when we encounter a negative value in the
   // parent array, which indicates we have found the unique cluster
   // root.
   // if (parent[s] >= 0) parent[s] is the parent site index
   // if (0 > parent[s] > EMPTY) s is the root of size -parent[s]
```