```
// if (parent[s] == EMPTY) site s is empty (unoccupied)
private int[] parent;

// A spanning cluster touches both left and right boundaries of the
// lattice. As clusters are merged, we maintain this information in
// the following arrays at the roots. For example, if root of a
// cluster is at site 7 and this cluster touches the left side,
// then touchesLeft[7] == true.
private boolean[] touchesLeft, touchesRght;

public Clusters(int L) {
   this.L = L;
   N = L*L;
   numClusters = new int[N+1];
   order = new int[N];
   parent = new int[N];
   touchesLeft = new boolean[N];
   touchesRght = new boolean[N];
}

public void newLattice() {
   setOccupationOrder(); // choose order in which sites are occupied
   // initially all sites are empty, and there are no clusters
   numSitesOccupied = secondClusterMoment = spanningClusterSize = 0;
   for(int s = 0;s<N;s++) {
      numClusters[s] = 0;
      parent[s] = EMPTY;
   }
   // initially left boundary touchesLeft,
   // right boundary touchesRight
   for(int s = 0;s<N;s++) {
      touchesLeft[s] = (s%L==0);
      touchesRght[s] = (s%L==L-1);
   }
}

// adds site to lattice and updates clusters.
public void addRandomSite() {
   // if all sites are occupied, we can't add anymore
   if(numSitesOccupied==N) {
      return;
   }
   // newSite is the index of random site to be occupied
   int newSite = order[numSitesOccupied++];
   // creates a new cluster containing only the site newSite
   numClusters[1]++;
   secondClusterMoment++;
   // store new cluster's size in parent[]; negative sign
   // distinguishes newSite as a root, with a size value.
   // Positive values correspond to nonroot sites with index
   // pointers.
   parent[newSite] = -1;
   // merge newSite with occupied neighbors. root is index of
   // merged cluster root at each step
   int root = newSite;
   for(int j = 0;j<4;j++) {
```



```
      // neighborSite is jth site neighboring newly added
      // newSite
      int neighborSite = getNeighbor(newSite, j);
      if(neighborSite!=EMPTY&&parent[neighborSite]!=EMPTY) {
         root = mergeRoots(root, findRoot(neighborSite));
      }
   }
}

// gets size of cluster to which site s belongs
public int getClusterSize(int s) {
   return parent[s]==EMPTY ? 0 : -parent[findRoot(s)];
}

// returns size of spanning cluster if it exists, otherwise 0
public int getSpanningClusterSize() {
   return spanningClusterSize;
}

// returns S (mean cluster size); sites belonging to spanning
// cluster is not counted in cluster moments
public double getMeanClusterSize() {
   int spanSize = getSpanningClusterSize();
   // subtract sites in spanning cluster
   double correctedSecondMoment =
        secondClusterMoment-spanSize*spanSize;
   double correctedFirstMoment = numSitesOccupied-spanSize;
   if(correctedFirstMoment>0) {
      return correctedSecondMoment/correctedFirstMoment;
   } else {
      return 0;
   }
}

// given a site index s, returns site index representing the root
// of the cluster to which s belongs
private int findRoot(int s) {
   if(parent[s]<0) {
      return s; // root site (with size -parent[s])
   } else {
      // first link parent[s] to the cluster's root to improve
      // performance (path compression); then return this value
      parent[s] = findRoot(parent[s]);
   }
   return parent[s];
}

// returns jth neighbor of site s; j can be 0 (left), 1 (right),
// 2 (down), or 3 (above). If no neighbor exists because of
// boundary, return value EMPTY. Change this method for periodic
// boundary conditions.
private int getNeighbor(int s, int j) {
   switch(j) {
   case 0:
      return s%L==0 ? EMPTY : s-1;   // left
   case 1:
```