algorithm runs faster on the average than Algorithm 8.6. The decision about whether to use Algorithm 8.6 or the Sherwood algorithm depends on the needs of the particular application. If better average performance is most important, we should use the Sherwood algorithm. If, on the other hand, quadratic performance can never be tolerated, we should use Algorithm 8.6. We stress again the advantage of the Sherwood algorithm over Algorithm 8.5 (Selection). As long as the inputs are uniformly distributed, Algorithm 8.5 also performs better on the average than Algorithm 8.6. However, in some particular application, the inputs may always be ones that approach the worst case in Algorithm 8.5. In such an application, Algorithm 8.5 always exhibits quadratic-time performance. The Sherwood algorithm avoids this difficulty in any application by choosing the pivot item at random according to a uniform distribution.

Next we present the probabilistic (Sherwood) algorithm.

Algorithm 8.7 **Probabilistic Selection**

Problem: Find the kth-smallest key in array S of n distinct keys.

Inputs: positive integers n and k where $k \leq n$, array of distinct keys S indexed from 1 to n.

Outputs: the kth-smallest key in S. It is returned as the value of function *selection3*.

```
keytype selection3 (index low, index high, index k)
{
   if (low == high)
      return S[low];
   else{
      partition3(low, high, pivotpoint);
      if (k == pivotpoint)
         return S[pivotpoint];
      else if (k < pivotpoint)
         return selection3(low, pivotpoint − 1, k);
      else
         return selection3(pivotpoint + 1, high, k);
   }
}

void partition3 (index low, index high,
                 index& pivotpoint)
{
   index i, j, randspot;
   keytype pivotitem;
   randspot = random index between low and high inclusive
              chosen according to a uniform distribution;
   pivotitem = S[randspot];                    // Randomly choose pivoti
   j = low;
   for (i = low + 1; i <= high; i++)
      if (S[i] < pivotitem){
         j++;
         exchange S[i] and S[j];
      }
   pivotpoint = j;
   exchange S[low] and S[pivotpoint];        // Put pivotitem at pivotpoi
}
```

Algorithm 8.7 differs from Algorithm 8.5 only in its random choice of the pivot item. Next we prove that the expected value of the number of comparisons is linear for any input. This analysis must be different from the average-case analysis of Algorithm 8.5 (Selection), because in that analysis we assumed that all values of k are entered with equal frequency. We want to obtain a linear-time expected value for each input. Because each input has a specific value of k, we can't assume that all values of k occur with equal frequency. We show that, independent of k, the expected value is linear.

Analysis of Algorithm 8.7

▶ Expected-Value Time Complexity (Probabilistic Selection)

Basic operation: the comparison of $S[i]$ with *pivotitem* in *partition*.

Input size: n, the number of items in the array.

Given that we are looking for the kth-smallest key in an input of size n, the following table shows the size of the input and the new value of k in the first recursive call for each value of *pivotpoint*.

pivotpoint	*Input Size*	*New Value of k*
1	$n-1$	$k-1$
2	$n-2$	$k-2$
$\vdots$	$\vdots$	$\vdots$
$k-1$	$n-(k-1)$	1
k	0	
$k+1$	k	k
$k+2$	$k+1$	k
$\vdots$	$\vdots$	$\vdots$
n	$n-1$	k