Chapter 12

Skip Lists

(Revision Date: 21 September 1998)

12.1 Introduction

Skip Lists were developed around 1989 by William Pugh[1] of the University of Maryland. Professor Pugh sees Skip Lists as an alternative to balanced trees such as AVL trees or to self-adjusting trees such as splay trees.

The `contains`, `add`, and `remove` operations on ordinary binary search trees are efficient, $O(\lg n)$, when the input data is random; but less efficient, $O(n)$, when the input data are ordered. Skip List performance for these same operations and for any data set is about as good as that of randomly-built binary search trees – namely $O(\lg n)$

Recall from Chapter 7 that in an ordinary `SortedList`, `contains`, `add`, and `remove` are in $O(n)$ because the list must be scanned node-by-node from the head to find the relevant node. If somehow we could scan down the list in bigger steps ("skip" down, as it were), we would reduce the cost of scanning. This is the fundamental idea behind Skip Lists.

In the simplest terms, Skip Lists are sorted lists with two differences:

1. the nodes in an ordinary list have one 'next' reference. The nodes in a Skip List have many 'next' references (called *forward* references).
2. the number of forward references for a given node is determined probabilistically.

[1]Communications of the ACM, June 1990, pp 668-676

We speak of a Skip List node having *levels*, one level per forward reference. The number of levels in a node is called the *size* of the node.

In an ordinary sorted list, `add`,`remove`, and `contains` operations require sequential traversal of the list. This results in $O(n)$ performance per operation. Skip Lists allow intermediate nodes in the list to be "skipped" during a traversal – resulting in an *expected* performance of $O(\lg n)$ per operation.

12.1.1 The Basic Idea

To introduce the Skip List data structure, let's look at three list data structures that allow skipping, but are not truly Skip Lists. The first of these, shown in Figure 12.1 allows every other node to be skipped in a traversal. The second, shown in Figure 12.2 additionally allows every fourth node to be skipped. The third, shown in Figure 12.3 additionally allows every eighth node to be skipped, but suggests further development of the idea to skipping every 2^i-th node.

For all three pseudo-skip-lists, there is a header node and the nodes do not all have the same number of forward references. Every node has a reference to the next node, but some have additional references to nodes further along the list.

Here is pseudo-code for the `contains` operation on each list. This same algorithm will be used with real Skip Lists later.

```
boolean contains(Object X)
 {
  node = header node
  for (reference level of node from (nodesize - 1) down to 0)
    while (the node referred to is less than X)
        node = node referred to
  if (node referred to has value X)
    return true
  else
   return false
 }
```

We start at the highest level of the header node and follow the references along this level until the value at the node referred to is equal to or larger than `X`. At this point, we switch to the next lower level and continue the search. Eventually, we shall be dealing with a reference at the lowest level of a node. If the next node has the value `X`, then we return true; otherwise we return false.

Skip Every Second Node

Figure 12.1 shows a 16–node list in which every second node has a reference two nodes ahead. The value stored at each node is shown below it (and corresponds in this example to the position of the node in the list). The header node has two levels; it's no smaller than largest node in the list. Node 2 has a reference to node 4, two nodes ahead. Similarly for nodes 4, 6, 8, *etc* – every second node has a reference two nodes ahead.

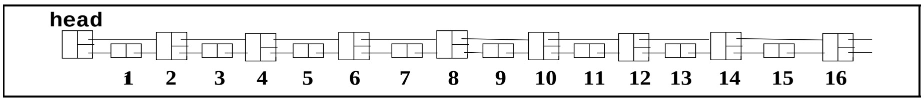

Figure 12.1: Skip Every 2nd Node

It's clear that the `contains` operation does not need to visit each node. It can skip over every other node, then do a final visit at the end. The number of nodes visited is therefore no more than $\lceil \frac{n}{2} \rceil + 1$. For example, the nodes visited in scanning for the node with value `15` would be `2, 4, 6, 8, 10, 12, 14, 16, 15`, a total of $(\lceil \frac{16}{2} \rceil + 1) = 9$.

Follow the algorithm for `contains` on page 184 for this simple example to be sure you understand it thoroughly.

Skip Every Second and Fourth Node

The second example is a list in which every second node has a reference two nodes ahead and additionally every fourth node has a reference four nodes ahead. Such a list is shown in Figure 12.2. The header is still no smaller than the largest node in the list.

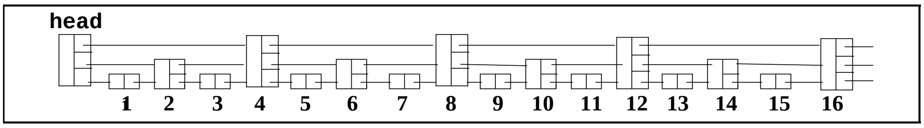

Figure 12.2: Skip Every 2nd and 4th Node

The `contains` operation can now make bigger skips than those for the list in Figure 12.1. Every fourth node is skipped until the search is confined between two nodes of size 3. At this point, as many as three nodes may need to be scanned. It is also possible that some nodes will be visited more than once using the algorithm on page 184. The number[2] of nodes visited is no more than $\lceil \frac{n}{4} \rceil + 3$. As an example, look at the nodes visited in scanning for the node with value of 15. These are `4, 8, 12, 16, 14, 16, 15` for a total of $(\lceil \frac{16}{4} \rceil + 3) = 7$.

[2] This is not the same value given by Pugh who may not have counted multiple visits to a node. Pugh's value is $\lceil \frac{n}{4} \rceil + 2$. This difference in the additive constant does not change the conclusion of $O(\lg n)$ performance for Skip Lists.

Skip Every 2^i-th Node

This final example is for a list in which the skip lengths are even larger. Every 2^i-th node, $i = 1, \ldots \lceil \lg n \rceil$, has a forward reference 2^i nodes ahead. Figure 12.3 shows a short list of this type. Once again, the header is no smaller than the largest node on the list. It is shown arbitrarily large in the Figure. If it were restricted to having just 5 levels, it would be suitable for storing 32 nodes.

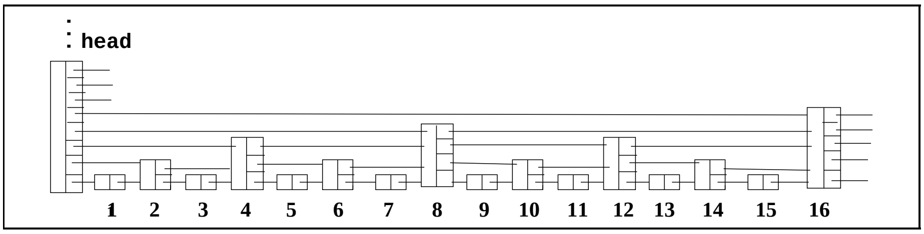

Figure 12.3: Skip Every 2^i-th Node

Look what happens with a search in this list. Working down from the highest level, we first encounter node 16 and have cut the search in half. We then search again, one level down in either the left or right half of the list, again cutting the remaining search in half. We continue in this manner till we find the sought-after node (or not). This is quite reminiscent of binary search in an array and is perhaps the best way to intuitively understand why the maximum number of nodes visited in this list is in $O(\lg n)$.

This data structure is looking pretty good, but there's a serious problem with it for the `add` and `remove` operations. The work required to reorganize the list after an insertion or deletion is in $O(n)$. For example, suppose that the first element is removed in Figure 12.3. Since it is necessary to maintain the strict pattern of node sizes, values from 2 to the end must be moved toward the head and the end node must be removed. A similar situation occurs when a new element is added to the list.

This is where the probabilistic approach of a true Skip List comes into play. A Skip List is built with the same distribution of node sizes, but without the requirement for the rigid pattern of node sizes shown. It is no longer necessary to maintain the rigid pattern by moving values around after a `remove` or `add` operation. Pugh shows that *with high probability* such a list still exhibits $O(\lg n)$ behavior. The probability that a given Skip List will perform badly is very small.

12.2 Skip Lists - the Probabilistic Approach

Figure 12.4 shows the list of Figure 12.3 with the nodes reorganized. The distribution of node sizes is exactly the same as that of Figure 12.3; the nodes just occur in a different pattern. In this example, the pattern would require that 50% of the nodes have just one reference, 25% have just two references, 12.5% have just three references, *etc.* The distribution of node sizes is maintained, but the strict order is not required.

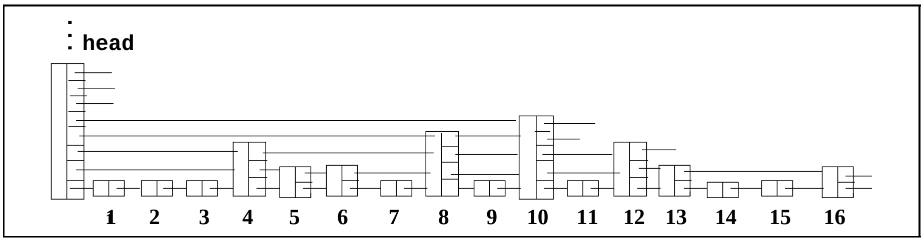

Figure 12.4: A Probabilistic Skip List

Figure 12.4 shows one way this might work out. Of course, this is probabilistic, so there are many other possible node sequences that would satisfy the required probability distribution.

When inserting new nodes, we choose the size of the new node probabilistically. Every Skip List has an associated (and fixed) probability p that determines the distribution of nodes. A fraction p of the nodes that have at least r references also have $r+1$ references. The Skip List does not have to be reorganized when a new element is inserted.

12.2.1 An Aside on Node Distribution

Suppose we have an infinitely-long Skip List with associated probability p. This means that a fraction, p, of the nodes with a forward reference at level i *also* have a forward reference at level $i+1$.

Let L_k be the fraction of nodes having *precisely* k forward references (*i.e.*, L_k is the fraction of nodes of size k). Then, $L_k = pL_{k-1}$ and

$$L_1 = 1 - \sum_{i=2}^{\infty} L_i$$

Since

$$\sum_{i=2}^{\infty} L_i = L_2 + L_3 + /cdots = L_2 + pL_2 + p^2 L_2 + \cdots$$

we can write L_1 in terms of L_2 as

$$L_1 = 1 - L_2 \sum_{i=0}^{\infty} p^i$$

Recall that $\sum_{i=0}^{\infty} p^i$ is the sum of the geometric progression with first term 1 and common ratio p. Thus,

$$\sum_{i=0}^{\infty} p^i = \frac{1}{1-p}$$

Therefore, $L_1 = 1 - \frac{L_2}{1-p}$. Since $L_2 = pL_1$, we can write

$$L_1 = 1 - \frac{pL_1}{1-p} \Rightarrow L_1 = 1 - p$$

EXAMPLE: In the situation shown in Figure 12.4, $p = 0.5$. Therefore, $1 - \frac{1}{2} = \frac{1}{2}$ of the nodes with at least one reference have two references; one-half of those with at least two references have three references, *etc*.

Exercise 3 invites you to work out the distribution for a `SkipList` with associated probability of $\frac{1}{4}$. You should do this exercise to be sure you understand how the distribution is computed.

12.2.2 Choosing a Node Probabilistically

When a new node is needed for a `add` operation, the size of the node is chosen by consulting a random number generator `r`. Here is one way to do this for a Skip List with associated probability `p` and maximum node level `maxLevel`

```
int generateNodeLevel(Random r, Double p, int maxLevel)
{
 int level = 1;

 while (r.nextDouble() < p)
   level++;
 return (level > maxLevel) ? maxLevel : level;
}
```

Note that the level of the new node is independent of the number of nodes already in the Skip List. Each node is chosen only on the basis of the Skip List's associated probability.

When the associated probability is p, the average number of comparisons that must be done for `contains`, is $1 + \frac{\log_{\frac{1}{p}} n}{p} + \frac{1}{1-p}$. For example, for a list of size 65,536, the average number of nodes to be examined is 34.3 for $p = \frac{1}{4}$ and 33 for $p = \frac{1}{2}$. This is a tremendous improvement over `SortedList` for which the average number of comparisons is $\frac{n}{2} = 32768$.

12.2.3 Determining the Header's Level

The level of the header node is the maximum allowed level in the SkipList and is chosen at construction. Pugh shows that the maximum level should be chosen as $\log_{\frac{1}{p}} n$. Thus, for $p = \frac{1}{2}$, the maximum level for a SkipList of up to 65,536 elements should be chosen no smaller than $\lg_2 65536 = 16$.

12.2.4 Performance Considerations

The *expected* time to find an element (and therefore to insert or delete an element) is in $O(\lg n)$. It is possible for the time to be substantially larger if the configuration of node levels is unfavorable for a particular operation. Since the node sizes are generated randomly, it is possible to get a "bad" run of sizes. For example, it is possible that each node will be generated at the same size, producing the equivalent of an ordinary SortedList. A bad run of sizes will result in longer-than expected search (and therefore add or remove) times; the SkipList will simply not be as efficient as expected. Obviously, a bad run will be proportionataly less important in a long SkipList than in a short one. The probability of poor performance decreases rapidly with the number of nodes (i.e., list size).

The probability that an operation will take longer than expected is a function of the probability associated with the list. For example, Pugh calculates that for a list with $p = 0.5$ and 4096 elements, the probability that the actual time will exceed the expected time by a factor of 3 is less than one in 200 million.

The relative time and space performance of a Skip List depends on the probability level associated with the list. Pugh suggests that a probability of 0.25 be used in most cases. If the variability of performance is important, he suggests using a probability of 0.5 (variability decreases with increasing probability). Interestingly, the average number of references per node is only 1.33 when a probability of 0.25 is used. A binary search tree, of course, has 2 references per node, so Skip Lists can be more space-efficient.

12.3 Overview of SkipList Methods

We will examine SkipList methods add, and remove in some detail below. Pseudocode for contains was given on page 184. Exercise 7 invites you to write the implementation of contains.

Insertion and deletion involve searching the SkipList to find the insertion or deletion point, then manipulating the references to make the relevant change.

When inserting a new element, we first generate a node that has had its level selected randomly. The SkipList has a maximum allowed level set at construction time. The number of levels in the header node is the maximum allowed. For convenience in searching, the SkipList keeps track of the maximum level actually in the list. There is no need to search levels above this actual maximum.

12.4 Implementation of Skip Lists

12.4.1 SkipListNode

SkipListNode is derived from SingleListNode. The big difference is that SkipListNode aggregates an array of forward references to other SkipListNodes. The getNext operation of SingleListNode is overridden to return the zero'th element of the array. A SkipListNode "is-A" SingleListNode to which additional functionality has been given.

Here is most of the code for SkipListNode:

```
package SJDS.list;

public class SkipListNode extends SingleListNode
{
   private SkipListNode [] _forward;  //forward references
   private int _numbLevels;  // number of levels in the node
```

The following constructor takes the datum to be stored in the node and the size of the node (number of levels). It constructs the superclass SingleListNode, then constructs the array of forward references. If numbLevels is less than 1, an exception is thrown. Finally, each of the forward references is pointed to null. Note that the forward references are indexed from 0.

```
  SkipListNode(Object datum, int numbLevels)
  {
    super(datum);
    if (numbLevels < 1)
        throw new
           IllegalArgumentException("SkipListNode construction: " +
                                    "size = " + numbLevels);

    _forward = new SkipListNode[numbLevels];
    _numbLevels = numbLevels;
    for (int i = 0; i < numbLevels; i++)
      setForward(i, null);
  }
```

The following constructor calls the above constructor with datum of null.

```
  SkipListNode(int numbLevels)
  {
    this(null, numbLevels);
  }
```

The next node is the one along the lowest-level reference. The following method overrides that in `SingleListNode`.

```
  protected ListNode getNextNode()
  {
    return getForward(0);
  }
```

The following mutator sets the specified forward reference. The accessor below returns the specified forward reference.

```
  protected void setForward(int i, SkipListNode node)
  {
    _forward[i] = node;
  }

  protected SkipListNode getForward(int i)
  {
    return _forward[i];
  }
}    // end SkipListNode class
```

12.4.2 SkipList

`SkipList` is derived from `SortedList`. The major difference is that `SkipList` chooses node sizes probabilistically, while in `SortedList` all nodes are the same size.

```
package SJDS.list;

import SJDS.hchecker.Hchecker;

import java.text.DecimalFormat;
import java.util.Random;
import java.util.Comparator;


public class SkipList extends SortedList
{
  private int _highNodeSize    = 1; // highest-level node in list
  private int _maxNodeSize     = 1; // max-level node allowed in list
  private double _prob         = 0; // probability factor
  private Random _randomizer   = null; // random generator
  public static final int MAX_NODE_SIZE   = 16;  // default value
  public static final double DEFAULT_PROB = 0.25; // default value
  private static final DecimalFormat df = new DecimalFormat("0.000");
```

The following constructor first constructs the superclass `SortedList` with a `SkipListNode` as its prototype node. If the probability value is improper, an exception is thrown. Finally, the randomizer used to determine node size is constructed.

```
public SkipList(Hchecker checker, Comparator comp,
                int maxNodeSize, double prob)
{
  super(checker, new SkipListNode(maxNodeSize), comp);
  _maxNodeSize = maxNodeSize;

  if ( (prob <= 0) || (prob > 1) )
    throw new
      IllegalArgumentException("Skip List construction: " +
                               "probability = " + prob);
  _prob = prob;
  _randomizer = new Random();
}
```

The following constructors take a variety of arguments. The "missing" arguments are set to default values. The "no-argument" constructor, for example, constructs a `SkipList` that has a `LooseChecker` homogeneity checker, a `null` comparator, maximum node size of 16, and probability of 0.25.

```
public SkipList(Hchecker checker, Comparator comp,
                int maxNodeSize)
{ this(checker, comp, maxNodeSize, DEFAULT_PROB); }

public SkipList(Hchecker checker, Comparator comp,
                double prob)
{ this(checker, comp, MAX_NODE_SIZE, prob); }


public SkipList(Hchecker checker, Comparator comp)
{ this(checker, comp, MAX_NODE_SIZE, DEFAULT_PROB); }

public SkipList(Hchecker checker)
{ this(checker, null, MAX_NODE_SIZE, DEFAULT_PROB); }


public SkipList()
{
  this(new LooseChecker(null));
}
```

We override the superclass' `cloneEmpty`, but do not override its `clone`. This means that the `clone` method used is that of the superclass `SortedList`. The nodes in the clone are chosen probabilistically and are unlikely to have the same sizes as those in the original. Exercise 6 invites you to write a `clone` method that produces a clone with the same node sizes as the original.

```
public Object cloneEmpty()
{
  return new SkipList(getHchecker(),
                      getComparator(),
                      getMaxNodeSize(),
                      getProbability());
}
```

The `toString` method uses the private`DecimalFormat` field `df` to present the probability with three decimal places.

```
public String toString()
{
  String st = ((getComparator() == null) ? "null Comparator"
             : (getComparator().toString()));

  return new String("SkipList: size " + size() + "," +
                   getHeader() + "," + getHchecker() +
                   "," + st + ",prob " + df.format(getProbability()));
}
```

The following `add` method is the one required by the `Collection` interface. It just checks homogeneity of the element to be inserted, then calls the `add(Object,int)` method that does all the work.

```
public boolean add(Object X)
     throws ClassCastException
{
  //  Maintain homogeneity of the list
  if ( ! getHchecker().isHomogeneous(X) )
    return false;

  return add(X, randomNodeSize());
}
```

The `findInsertPoint` method overrides that in the superclass `SortedList`. There it returned the node that precedes the new node being inserted. For `SkipList`, it's a bit more complicated. The insertion point is found by traversing the list in the usual "skipping" manner. When the insertion point is found, a "backwards-looking" `SkipListNode` is constructed and returned. This node looks back (toward the head of the list) and has its references set to the closest node of that level. The backwards-looking node is a useful device for helping to stitch the new node into the list. For example, suppose a new element is to be added between nodes 6 and 7 in Figure 12.4. Figure 12.5 shows the `backLook` node in the position between nodes 6 and 7 with its forward references looking back toward the head of the list.

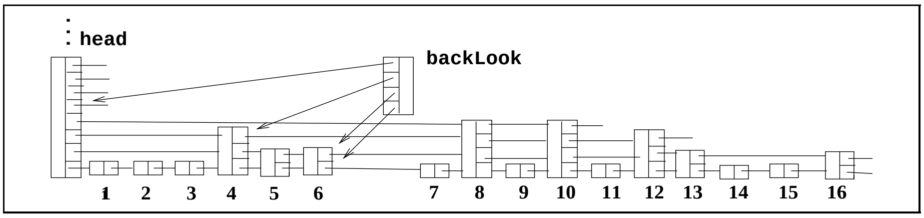

Figure 12.5: Update Node Example

```
  protected ListNode findInsertPoint(Object X, int lookBackSize)
  {
    SkipListNode backLook = new SkipListNode(lookBackSize);
    SkipListNode S = (SkipListNode) getHeader();

    SkipListNode T = null;
    Comparator comp = getComparator();
    int compval = 0;

    for (int i = 0; i < lookBackSize; i++)
      {
backLook.setForward(i, S);
      }

    for (int i = lookBackSize - 1; i >= 0; i--)
      {
while (true)   // scan for the  point
  {
    if (S.getForward(i) == null)
      break;

    Object data = S.getForward(i).getDatum();

    if ( (comp == null) ?
 (((Comparable)X).compareTo(data) >= 0) :
 (comp.compare(data,X) >= 0)  )
      {
S = S.getForward(i);
      }
    else
      break;
  } // while

backLook.setForward(i,S);
      }

    return backLook;
  }
```

The `add(Object, int)` method uses the backLook node returned by `findInsertPoint` to stitch in a new node of the specified size.

```
protected boolean add(Object X, int size)
     throws ClassCastException
{
  SkipListNode update = (SkipListNode) findInsertPoint(X, size);
  SkipListNode newnode = new SkipListNode(X, size );
  newnode.setList(this);

  SkipListNode prev = null;
  SkipListNode next = null;

  // newnode may have more levels than currently in list.
  // If so, adjust current high size.
  int newsize = newnode.getNumbLevels();
  if (newsize > getHighNodeSize())
    setHighNodeSize(newsize);

  for (int i = 0; i < newsize; i++)
    {
      prev = update.getForward(i);
      next = prev.getForward(i);
      newnode.setForward(i, next);
      prev.setForward(i, newnode);
    }

  setCurrentNode(newnode);
  incrementListSize();
  return true;
}
```

The `remove` method also uses `findInsertPoint` to obtain the backLook node. In this case, backLook is used to aid the unstitching.

```
  public boolean remove()
  {
    if ( isEmpty() )
      return false;
    Object X = getCurrentNode().getDatum();

    // update "node" will be at X's position.
    SkipListNode update =
       (SkipListNode) findInsertPoint(X,getHighNodeSize());

    SkipListNode pred = null;
    SkipListNode next = null;
    SkipListNode curr = (SkipListNode) getCurrentNode();

    for (int i = 0; i < getHighNodeSize(); i++)
      {
        pred = update.getForward(i);
        next = pred.getForward(i);

        if (next != curr)  // at level above current node
          break;

        // unlink current node at level i
        next = next.getForward(i);
        pred.setForward(i, next);
      }

    decrementListSize();

    if (isEmpty())
      setCurrentNode(null);
    else if (update.getForward(0) == getHeader())
      // deleted node was first
      setCurrentNode(((SkipListNode)getHeader()).getForward(0));
    else
      setCurrentNode(update.getForward(0));

    return true;
  }
} // end SkipList class
```

12.5 Exercises

1. Figure 12.3 is valid for $p = \frac{1}{2}$. Draw an equivalent figure for $p = \frac{1}{4}$. What distribution of node levels do you expect in a long list of this type? Hint: use the formulas of Section 12.2.1 for L_k in terms of L_{k-1} and for L_1.

2. The expected asymptotic time performance for `SkipList` operations is $O(\lg n)$. There is a non-zero probability that the performance could be as bad as $O(n)$. Draw a 7 element `SkipList`, with `int` data values, that would have such poor performance. Use a maximum node level of 4.

3. A `SkipList` has been constructed using

```
SkipList foo = new SkipList(checker, 16, 0.25, comp)
```

where `comp` is an appropriate `Comparator`.
After inserting 100 items using `add`, what is the expected distribution of node levels in the list (*i.e.* how many of level 1, how many of level 2, *etc.*?)

4. Consider the list of Figure 12.3 in which node levels follow a strict pattern.
   (a) Write pseudo-Java code for `add` in such a list. Concentrate on the insertion operation, don't worry about details like exceptions, etc.
   (b) What "Big Oh" time performance do you expect for your algorithm?

5. The `SkipList` implementation given in this chapter defaults to use of `SkipListNode`, which is a subclass of `SingleListNode`. Recall that a disadvantage of `SingleListNode` is that finding the previous node is in $O(n)$. Define a doubly-linked `SkipListNode` that has a `prev` reference.

6. `SkipList` inherits its `clone` method from `SortedList`, overriding only the `cloneEmpty` method. A clone of a `SkipList` therefore has the same elements as the original and in the same order, but is unlikely to have the same distribution of node sizes. This could result in the clone having different asymptotic performance than the original, an undesirable situation. Write a `clone` method for `SkipList` that ensures that the size of each node in the clone matches that of its corresponding node in the original. *Hint*: as usual, clone an empty list. After that, use `first` and `next` to traverse the original list from the header, determining node size. Use `add(Object,int)` directly to add elements to the clone.

7. Write the `contains` method for `SkipList`. The algorithm is given on page 184.

8. Write an `Iterator` that gives the sizes of the nodes in a `SkipList` rather than the data stored at the nodes. Use it to analyze the node distributions produced with various probability levels. How do the distributions compare to those expected?

9. What maximum node size is appropriate for a `SkipList` suitable for storing up to 65,536 elements and with associated probability $\frac{1}{4}$.