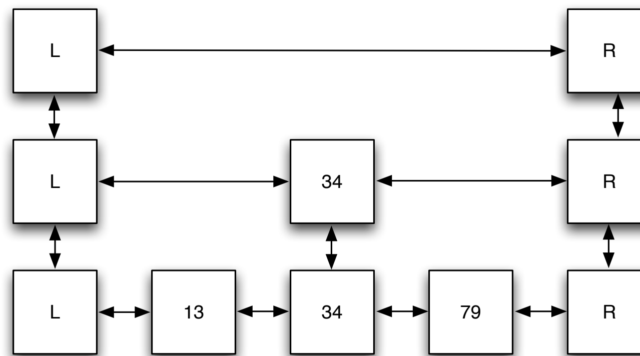
Question 1. *Suppose you have a biased coin that, when flipped, takes on heads with unknown probability p and tails with probability $1 - p$. Show how to use this coin to construct a string of n independent bits such that each bit is equally likely to be a 0 or a 1. In other words, show how to use this coin to construct an algorithm that, when run, behaves like an unbiased coin. What is the expected running time of your algorithm as a function of p?*

Question 2 (Inspired by Dave Moore '10)**.** *Imagine a procedure* $\textsc{Random}(a, b)$ *that, when called, returns an integer between a and b inclusively and uniformly at random. That is, each integer in the range $[a, b]$ is equally likely to appear on a call to* $\textsc{Random}(a, b)$*. Now, suppose you have a fair coin. Describe an implementation of* $\textsc{Random}(a, b)$ *that is only allowed to flip this coin (i.e., it can't use any other source of randomness). What is the expected running time of your procedure, as a function of a and b?*

Question 3. *Suppose you have the same biased coin as in Question 1, however, this time you don't wish to construct an algorithm to produce an unbiased coin (i.e. a coin with probability 1/2 of coming up heads) but rather another unbiased coin with some given probability q of coming up heads. Show how to use your coin to construct an algorithm that, when run, behaves like a biased coin with probability q of returning heads. You may assume that q is a rational number. Can your solution to Question 2 help?*

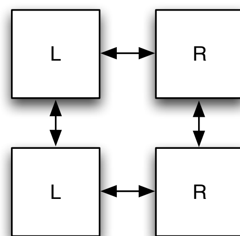
Question 4 (JE)**.** *Suppose you have a set of n complementary nuts and bolts where the size of each pair is unique. Consider the following randomized algorithm for choosing the largest bolt. Draw a bolt uniformly at random from the set of n bolts, and draw a nut uniformly at random from the set of n nuts. If the bolt is smaller than the nut, discard the bolt, draw a new bolt uniformly at random from the unchosen bolts, and repeat. Otherwise, discard the nut, draw a new nut uniformly at random from the unchosen nuts, and repeat. Stop either when every nut has been discarded, or every bolt except the one in your hand has been discarded. What is the exact expected number of nut-bolt tests performed by this algorithm? Prove your answer is correct. [*Hint: What is the expected number of unchosen nuts and bolts when the algorithm terminates?*]*

Question 5 (Extra Credit)**.** *This question asks you to implement a skip list in Java. Recall that a skip list is a collection of linked lists, organized by level. Every item in the list appears at level 0. Items appear at successive levels with geometrically diminishing probability. Our skip list will take advantage of two boundaries along the left and right edges of the skip list, as well as a boundary above the highest level of the skip list. These boundaries will make finding, inserting, and removing items easier. Here's a picture of a skip list with values 13, 34, and 79.*

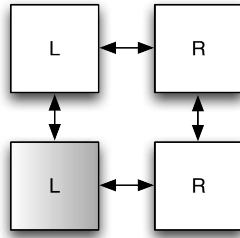


Addition

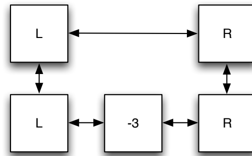
Recall that after an item is inserted at level 0, it is promoted to level 1 with probability 1/2. The process then repeats itself. In general, an item at level i is promoted to level $i + 1$ with probability 1/2. However, an item is never promoted above the top boundary. In fact, we always have an upper boundary that never contains any egers. If an item is ever promoted to the upper boundary, we immediately take away its coin (so that it can't be promoted anymore), and add a new upper boundary. For example, say we insert the value -3 into an empty skip list.



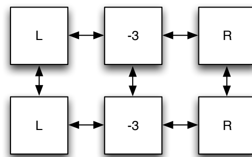
Notice that level 0 has an upper boundary. We first find the correct node on level 0 (this node should have the largest value not exceeding -3).



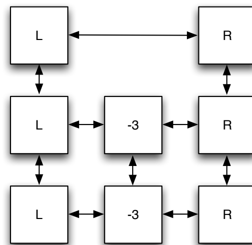
We then insert a new node with value -3 and update the appropriate pointers.



Next, we flip a coin. Say it ends up heads. Then we promote -3 to the next level.

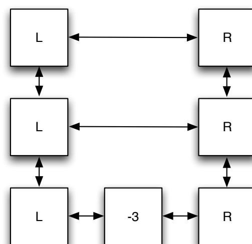


However, since -3 has reached the top boundary, we steal its coin away and no longer allow it promotion. Furthermore, we add a new upper boundary.



Removal

Removing a node is similar. However, beware of superfluous upper boundaries. That is, we should never see a skip list like this.



Implementation

You should implement a skip list as described above. Use `turnin` *to turn in your* `SkipList.java` *file. Please note and adhere to the following:*

- *Your skip list should be a Java class called* `SkipList`. *It should be placed inside a file called* `SkipList.java`.
- *Your skip list is generic—it should be parameterized by some type* `T` *which is* `Comparable`. *The type* `T` *is the type of the values stored in the skip list.*
- *Starter code is provided on the website. It contains a sparse skeleton of the* `SkipList` *class as well as moderate skeletons of three inner private classes* `Node`, `LeftBoundaryNode`, *and* `RightBoundaryNode`. *Feel free to use the starter code or ignore it. However, your final code* **must** *adhere to the constraints outlined herein. Note that the starter code uses a* `Node` *class with pointers in all four directions.* `null` *is the default value of these pointers. The default* `value` *of a node is also* `null`.
- *Do not place your code inside a specific package.*
- *Your* `SkipList` *class should* not *contain a* `main` *method when you turn it in. That said, you will find a* `main` *method helpful when testing.*
- *Your* `SkipList` *class should support the following methods:*
  1. `public boolean add(T i)` *where the return value indicates if the addition was successful (duplicate elements are not allowed).*
  2. `public boolean remove(T i)` *where the return value indicates if the removal was successful.*
  3. `public boolean contains(T i)`
  4. `public T predecessor(T i)` *– returns the largest item less than or equal i in the skip list. Note that i may not be in the skip list. This is called* `floor` *in Java's* `TreeSet`.
  5. `public T successor(T i)` *– return the smallest item greater than or equal to i in the skip list. Note that i may not be in the skip list. This is called* `ceiling` *in Java's* `TreeSet`.
  6. `public int size()` *– return the number of items in level 0 (boundary nodes don't count). Do not return the number of nodes in the skip list.*
- *Duplicate items are not permitted in the skip list.*
- *Use the* `nextBoolean()` *method of the* `Random` *class for coin flips.*
- *The starter code seeds the random number generator with 47. Your code should match the given example output if you maintain this seed.*

Running your completed code with the following `main` *method*

```
public static void main(String[] args) {
  SkipList<Integer> l = new SkipList<Integer>();
  System.out.println(l);
  l.add(new Integer(-1000));
  System.out.println(l);
  l.add(new Integer(3000));
  System.out.println(l);
}
```

should produce the following output

```
assonance:SkipLists heeringa$ java -Xms512m -Xmx1024m SkipList
[ L ] -- [ R ]
[ L ] -- [ R ]

[ L ] -- [ R ]
[ L ] -- [ -1000 ] -- [ R ]
[ L ] -- [ -1000 ] -- [ R ]

[ L ] -- [ R ]
[ L ] -- [ -1000 ] -- [ R ]
[ L ] -- [ -1000 ] -- [ 3000 ] -- [ R ]
```

The `-Xms512m -Xmx1024m` *arguments to the java virtual machine tell it that the minimum heap size should be 512 MB and the maximum heap size should be 1024 MB.*

Experiments

William Pugh, the creator of skip lists, reports that the randomized data structure offers comparable if not better performance than red-black trees. We should test this claim (albeit modestly) using your implementation and the Java `TreeSet` *class which is (conveniently) implemented using a red-black tree. Use the following syntax to create a* `TreeSet` *of integers:*

```
TreeSet<Integer> set = new TreeSet<Integer>();
```

Notice that we must use the Java Object type `Integer` *rather than the primitive* `int`*. To add an integer, use:*

```
set.add(new Integer(-3));
```

More information about the Java `TreeSet` *class is available at* `http://download.oracle.com/javase/6/docs/api/java/u`

Perform the following experiments (you probably want to disable the JIT by using `-Xint` *as an additional flag to the Java interpreter:*

1. *Insert 50,000 random integers into your skip list. Insert the same integers into a Java* `TreeSet`*. Report on the total and average insertion time for both data structures. Draw conclusions. You may find* `System.currentTimeMillis()` *helpful. Recall that duplicates are not allowed, so your sets will likely have size less than 50,000.*
2. *Now insert an additional random 50,000 integers into your skip list. Insert the same integers into the* `TreeSet`*. Both data structures should have the same size. Now delete the first 50,000 integers (some of these deletions may not be possible which is okay since your remove method will report that no deletion occurred). Report on the total and average removal time for both data structures. Draw conclusions.*
3. *Perform one well motivated experiment of your own design. Describe it in detail and report on the results.*