To read

Problem: \textbf{Find}(Q) in your list.
Difficulty:
Subsequent Insert/Delete operations would destroy this favourable list shape. The cost of restauration is huge -- $\Theta(N)$.

Solution:
Create a randomized shape, roughly similar to the optimal shape. Random deviations from the nice shape in the long run nearly cancel each other. The result is a list shape with properties relatively close to the optimal properties.
A skip list is an ordered linked list where each node contains a variable number of links, with the k-th link in the node implementing singly linked list that skips (forward) the nodes with less than k links. [Sedgewick]

Each element points to its immediate successor (= next element). Some elements also point to one or more elements further down the list.

A level $k$ element has $k$ forward pointers. the $j$-th pointer points to the next element in level $j$.
while(x.forward[i].key < searchKey) // x.forward[i] != null

There is a **sentinel** with infinite key value at the tail of the list.

The level of the sentinel is the same as the whole list level.
The list may be implemented as circular with the header serving as the sentinel.

Note that the skip list is displayed as linear in this presentation (with separate sentinel) to keep pictures less cluttered.
A skip list data structure contains also:

-- **Header**  
A node with the initial set of forward pointers.

-- **Sentinel**  
Optional last node with $\infty$ value, it is the header in circular list.

-- **Level**  
The current number of levels in the skip list.

-- **MaxLevel**  
The maximum number of levels to which a skip list can grow.

-- **Update[ ]**  
Auxiliary array with predecessors of an inserted/deleted element  
see Insert and Delete operations.

A skip list element contains:

-- **Key**  
Search key.

-- **Value**  
(Optional, not discussed here, allowing associative structure.)

-- **Forward[ ]**  
Array of pointers to the following skip list elements.

The header and the sentinel are of the same type.
Basic randomness

The level of an element is chosen by flipping a coin.

Flip a coin until it comes up tails. Count one plus the number of times the coin came up heads before it comes up tails.

This result represents the level of the element.
Experiment with Lehmer generator

$$X_{n+1} = 16807 \times X_n \mod 2^{31} - 1$$

seed = 23021905  // birth date of Derrick Henry Lehmer

| Level | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ...
|-------|---|---|---|---|---|---|---|---|---|---
| Number of nodes | Expected | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 1/2 | 1/4 | ...
| Actual | 60 | 36 | 17 | 5 | 7 | 1 | 1 | 1 | 0 | ... |

Coin flipping:

$$(X_n \gg 16) \& 1$$

Head = 1

128 nodes

Pokročilá Algoritmizace, A4M33PAL, ZS 2012/2013, FEL ČVUT, 12/14
More general randomness

Choose a fraction $p$ between 0 and 1.
Rule: Fraction $p$ of elements with level $k$ pointers will have level $k+1$ elements as well.

On average: 
- $(1-p)$ elements will be level 1 elements,
- $(1-p)^2$ elements will be level 2 elements,
- $(1-p)^3$ elements will be level 3 elements, etc.

This scheme corresponds to flipping a coin that has $p$ chance of coming up heads, $(1-p)$ chance of coming up tails.

Example of an experimental independent levels calculation with $p = 0.33$. 
Choosing a Random Level

A level is chosen for an element in effect by flipping a coin that has probability $p$ of coming up heads. We keep flipping until we get "tails" or until the maximum number of levels is reached.

```c
int randomLevel( List list ) {
    // random() returns a random value in [0..1)
    int newLevel = 1;
    while( random() < list.p )    // no MaxLevel check!
        newLevel++;
    return min( newLevel, list.MaxLevel ); // efficiency!
}
```
Search

Scan through the top list until the current node either contains the search key or it contains a smaller key and a link to a node with a larger key.

Then, move to the second-from-top list and iterate the procedure, continuing forward and downward until the search key is found or a search mismatch happens at the bottom level.
Search

Start with the coarsest grain list and find where in that list the key resides, then drop down to the next less coarse grain list and repeat the search.

```java
Node search( List list, int searchKey ) {
    Node x = list.header;

    // loop invariant: x.key < searchKey, strict ineq!!
    for( int i = list.level; i >= 1; i-- )
        while( x.forward[i].key < searchKey )
            x = x.forward[i];

    // x.key < searchKey <= x.forward[1].key
    x = x.forward[1];

    if( x.key == searchKey ) return x;
    else return null;  // not found
}
```
Insert

Find the place for the new element. Compute its level $k$ by flipping the coin. Insert the element into first $k$ lists, starting at the bottom list.
The array `update[]` is an auxiliary array supporting Insert / Delete operations.

`update[k]` points to that element in the list whose level `k` pointer points to the inserted (or deleted) element, (= predecessor in the `k`-th level).

Note that in many cases, when the level of the inserted/deleted element is 1, only `update[1]` will be used.
The nodes, in the order of insertion, (A, S, E, R, C, H, I, N, G) were assigned levels 1, 3, 2, 1, 1, 3, 1, 3, 2.

[Sedgewick]

The nodes, in the order of insertion, were assigned levels 1, 3, 2, 1, 1, 3, 1, 3, 2.

[Sedgewick]

The nodes, in the order of insertion, were assigned levels 1, 3, 2, 1, 1, 3, 1, 3, 2.

[Sedgewick]
The nodes, in the order of insertion, were assigned levels 1, 3, 2, 1, 1, 3, 1, 3, 2. (A, S, E, R, C, H, I, N, G)
Insert A, C, E, G, H, I, N, R, S. (Same values, different order) The nodes \((A, C, E, G, H, I, N, R, S)\) were assigned levels \(1, 3, 2, 1, 1, 3, 1, 3, 2\).
The nodes were inserted in sorted order.

The result of the previous example

The nodes were inserted in random order.

The shapes of the lists are different, the probabilistic properties are the same.

The nodes, in the order of insertion, were assigned levels 1, 3, 2, 1, 1, 3, 1, 3, 2.

The nodes, in the order of insertion, were assigned levels 1, 3, 2, 1, 1, 3, 1, 3, 2.
// update[k] .. predecessor at level k

define insert(List list, int searchKey, Data newValue){
  Node x = list.header;
  for( int i = list.level; i >= 1; i-- ){
    // invariant: x.key < searchKey <= x.forward[i].key
    while( x.forward[i].key < searchKey )
      x = x.forward[i];
    update[i] = x;
  }
  x = x.forward[1];       // expected position

  if( x.key == searchKey )
    x.value = newValue;  // associative structure
  else {
    continue...  // key not found, do insertion:
  }
}
... else { // key not found, do insertion here:
    int newLevel = randomLevel( list );
    /* If newLevel is greater than the current level
    of the list, knock newLevel down so that it is only
    one level more than the current level of the list.
    In other words, increase the level of the list
    by at most 1 in each insert operation. */
    if( newLevel > list.level ) {
        if( list.level < list.MaxLevel ) list.level++;
        newLevel = list.level;
        update[newLevel] = list.header; // sentinel
    }
    // finally, physical insertion:
    Node x = makeNode( newLevel, searchKey, newValue );
    for( int i = 1; i <= newLevel; i++ ) {
        x.forward[i] = update[i].forward[i];
        update[i].forward[i] = x;
    }
}} // of insert
Skip list - Delete

Delete L

Deleting in a skip list is like deleting the same value independently from each list in which forward pointers of the deleted element are involved.

The algorithm registers the element's predecessor in the list, makes the predecessor point to the element that the deleted element points to, and finally deletes the element. It is a regular list delete operation.

```c
void delete(List list, int searchKey) {
    Node x = list.header;
    for (int i = list.level; i >= 1; i--) {  
        while (x.forward[i].key < searchKey)  
            x = x.forward[i];
        update[i] = x;
    }
    x = x.forward[1];
    if (x.key == searchkey) {  // go delete ...}  
}
```
If the element to be deleted is a level $k$ node, break out of the loop when level $(k+1)$ is reached. Since the code does not store the level of an element, we determine that we have exhausted the levels of an element when a predecessor element points past it, rather than to it.

```c
for (int i = 1; i <= list.level; i++) {
    if (update[i].forward[i] != x) break; //(**)
    update[i].forward[i] = x.forward[i];
}
destroy_remove(x);

/* if deleting the element causes some of the highest level list to become empty, decrease the list level until a non-empty list is encountered. */
while ((list.level > 1) && (list.header.forward[list.level] == list.header))
    list.level--;
} // deleted
Choosing \( p \)

One might think that \( p \) should be chosen to be 0.5. If \( p \) is chosen to be 0.5, then roughly half our elements will be level 1 nodes, 0.25 will be level 2 nodes, 0.125 will be level 3 nodes, and so on. This will give us

- on average \( \log(N) \) search time and
- on average 2 pointers per node.

However, empirical tests show that choosing \( p \) to be 0.25 results in

- roughly the same search time
- but only an average of 1.33 pointers per node,
- somewhat more variability in the search times.

There is a greater chance of a search taking longer than expected, but the decrease in storage overhead seems to be worth it sometimes.
Notes on size and complexity

The average number of links in a randomized skip list with parameter \( p \) is

\[ N \cdot \frac{1}{1 - p} \]

The average number of key comparisons in search and insert in a randomized skip list with parameter \( p \) is on average

\[ -\log_p(N) / 2p = \log(N) \cdot (-1) \cdot (2p \cdot \log(p))^{-1} = \log(N) / (2p \cdot \log(1/p)) \]

**Experimental time comparisons:**

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip list</td>
<td>0.051</td>
<td>(1.0)</td>
<td>0.065</td>
</tr>
<tr>
<td>AVL tree</td>
<td>0.046</td>
<td>(0.91)</td>
<td>0.100</td>
</tr>
<tr>
<td>2-3 tree</td>
<td>0.054</td>
<td>(1.05)</td>
<td>0.210</td>
</tr>
<tr>
<td>Splay tree</td>
<td>0.490</td>
<td>(9.6)</td>
<td>0.510</td>
</tr>
</tbody>
</table>

Times in ms on some antiquated HW [Pugh, 1990]
Notes on complexity

The probabilistic analysis of skip lists is rather advanced. However, it can be shown that the expected times of search, insert, delete are all $O(\lg n)$.

The choice of $p$ determines the variability of these search times.

Intuitively, decreasing $p$ will increase the variability since it will decrease the number of higher-level elements (i.e., the number of "skip" nodes in the list).

The Pugh paper contains a number of graphs that show the probability of a search taking significantly longer than expected for given values of $p$. For example, if $p$ is 0.5 and there are more than 256 elements in the list, the chances of a search taking 3 times longer than expected are less than 1 in a million. If $p$ is decreased to 0.25, the chances rise to about 1 in a thousand.
Array-like property -- random element access

Supplement each forward pointer with its "length" = 1 + number of the list elements it skips.

A k-th list element can be accessed in expected $O(\log n)$ time.

Search, Insert, Delete are analogous to the "plain" variant. The length of the affected pointers has to be updated after each Insert or Delete. Asymptotic complexity remains the same in all cases -- $O(\log n)$.
- Erik Demaine's presentation at MIT
  http://videolectures.net/mit6046jf05_demaine_lec12/


**Also:**  `java.util.concurrent.ConcurrentSkipListSet<E>`