Maintaining balanced binary search trees to support dynamic dictionary operations like search, insert, delete is often quite cumbersome. Rotations/double rotations/color change in Red-Black trees/node splitting and joining in B-trees, are not easy to remember. Challenge is to remember $O(\log n)$-time for each operation.

As opposed to trees, linked lists are relatively easier to maintain.
but suffer from the disadvantage of much higher time per operation, namely linear.

Can we combine the advantages of lists and trees?

We will maintain a sorted list to support fast searches.
This supports logn search time
but requires a very strong invariant
i.e., every alternate (every K-th bit)
must be "promoted" to the level above

New "randomized" invariant

Every element is promoted to the
next level with probability \( \frac{1}{2} \)

(Independently)

What is the "expected" gap between
two promoted elements?

Recall that the gap determines
the maximum traversals within a level

The expected gap is 2

What is the expected height? \( \Theta(\log n) \)
What is the expected length of the search path?

Look at a search path in a backward manner.

\[ E[L] = E[l_1 + l_2 + \cdots + l_k] \]

\[ = \sum E[l_i] = 2 \cdot k \]

\( k \) can be bounded by \( \log n \) (by choice).

Then the expected number of elements in level \( k \) is \( \left( \frac{1}{2} \right)^k \cdot \eta = O(1) \) for \( k = \log n \).

So search time is \( 2k + \) Expected of element in top level.
Skip List: a randomized data structure for dynamic dictionary

by William Pugh in 1988

Insertion: Search for the position in L0 and promote by coin-tossing
Expected # of promotions = 2

Deletion: Reverse the process

Time for insert/delete = search time + # copies

Can be applied to "Concealable Queues"

Split and Union of lists/sets

L_1 \leftarrow L_2 \rightarrow L_1

All elements of L_2 \geq L_1,
What is the expected height of the skip list if we want to bound the total # elements in the topmost list by say 10.