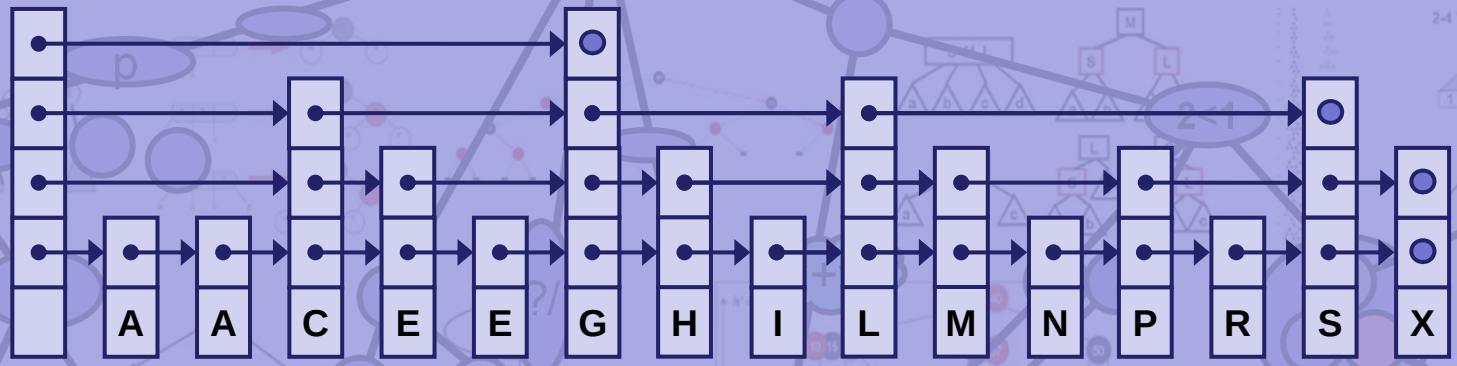
Skip List

ACM ICPC Maraton
Prague 2015



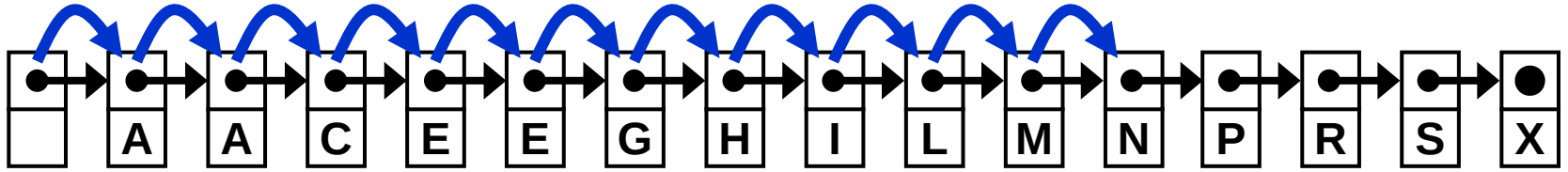
Marko Berezovský

To read

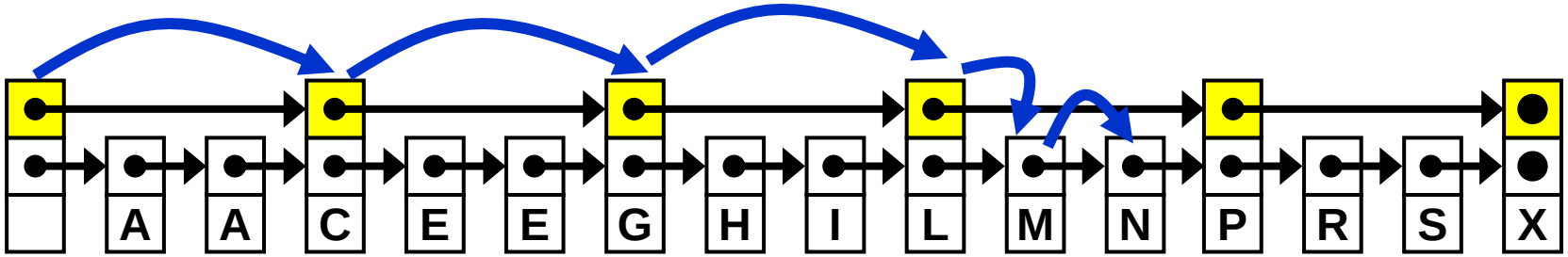
- Robert Sedgewick: *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching, Third Edition*, Addison Wesley Professional, 1998
- William Pugh. *Skip lists: A probabilistic alternative to balanced trees*. Communications of the ACM, 33(6):668–676, 1990.
- William Pugh: *A Skip List Cookbook* [http://cglab.ca/~morin/teaching/5408/refs/p90b.pdf]
- Bradley T. Vander Zanden: [http://web.eecs.utk.edu/~huangj/CS302S04/notes/skip-lists.html]

Problem: **Find(N)** in your list.

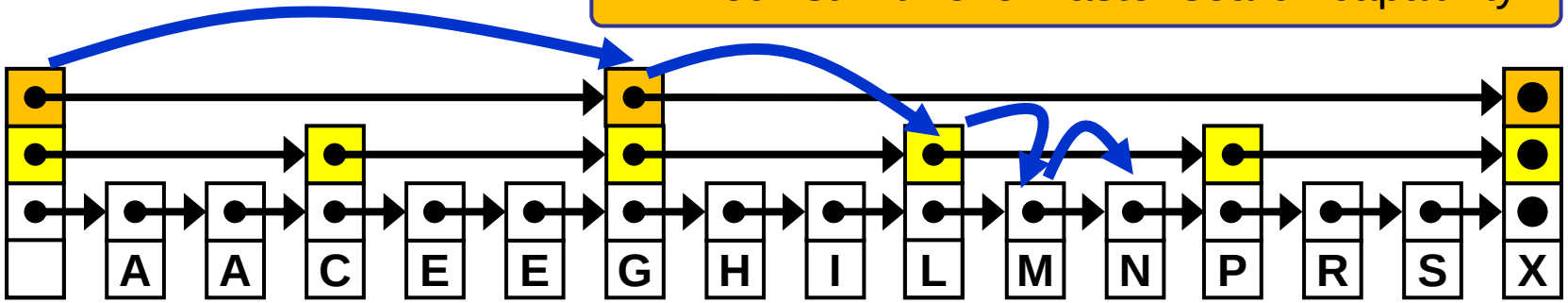
A regular linked list

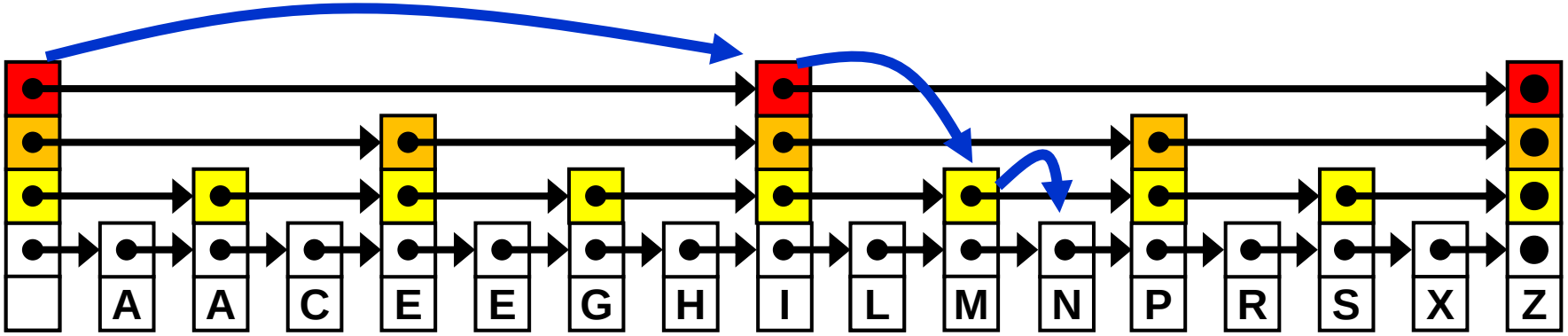


A linked list with faster search capability



A linked list with even faster search capability





A linked list with **log(N)** search capability.
Note the shape **similarity** to a **balanced binary search tree**.

Problem:
Subsequent Insert/Delete operations would destroy this favourable list shape.
The cost of restauration is huge -- $\Theta(N)$.

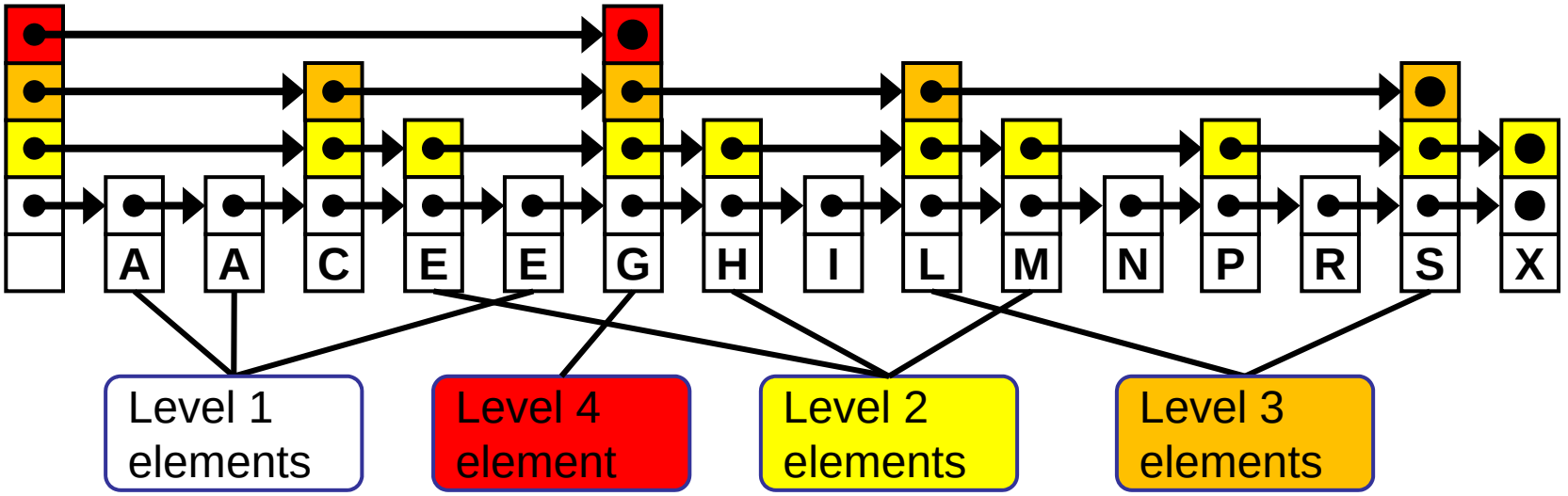
Solution:
Create a randomized shape, roughly similar to the optimal one.
Random deviations from the nice shape in the long run nearly cancel each other resulting again in a nearly favourable list shape.

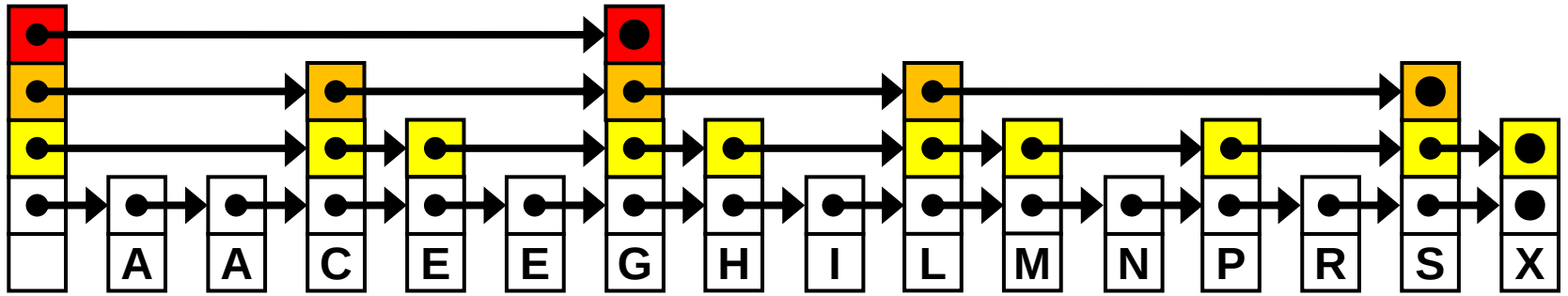
A skip list is an ordered linked list where each node contains a variable number of links, with the k-th link in the node implementing singly linked list that skips (forward) the nodes with less than k links.

[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 is a list element that has ***k*** forward pointers.
the *j-th* pointer points to the next level *j* element.





The Skip List data structure contains also:

-- **Header:** A dummy skip list node with the initial set of forward pointers
-- **Level:** The current number of levels in the skip list.
-- **MaxLevel:** The maximum number of levels to which a skip list can grow

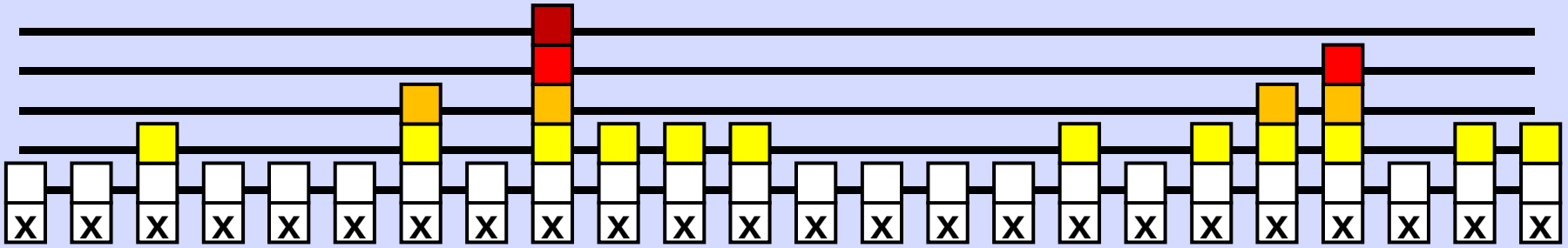
Basic randomness

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

Flip a coin until it comes up tails.
We 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.

Sixpence of Queen Elizabeth I, struck in 1593 at the Tower Mint. [wikipedia.org]



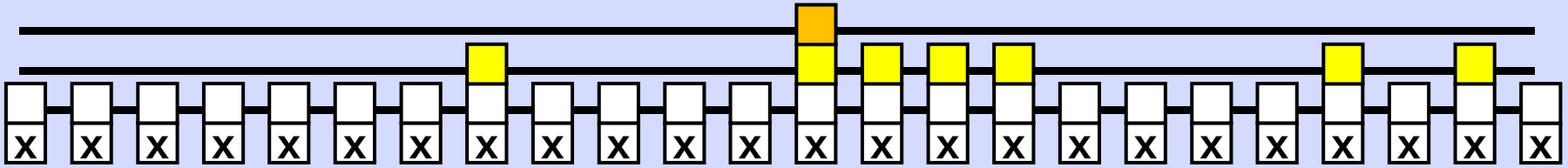
Example of an experimental independent levels calculation (p = 0.5, see bellow) **.**

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

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.



Example of an experimental independent levels calculation with p = 0.33.

Choosing a Random Level

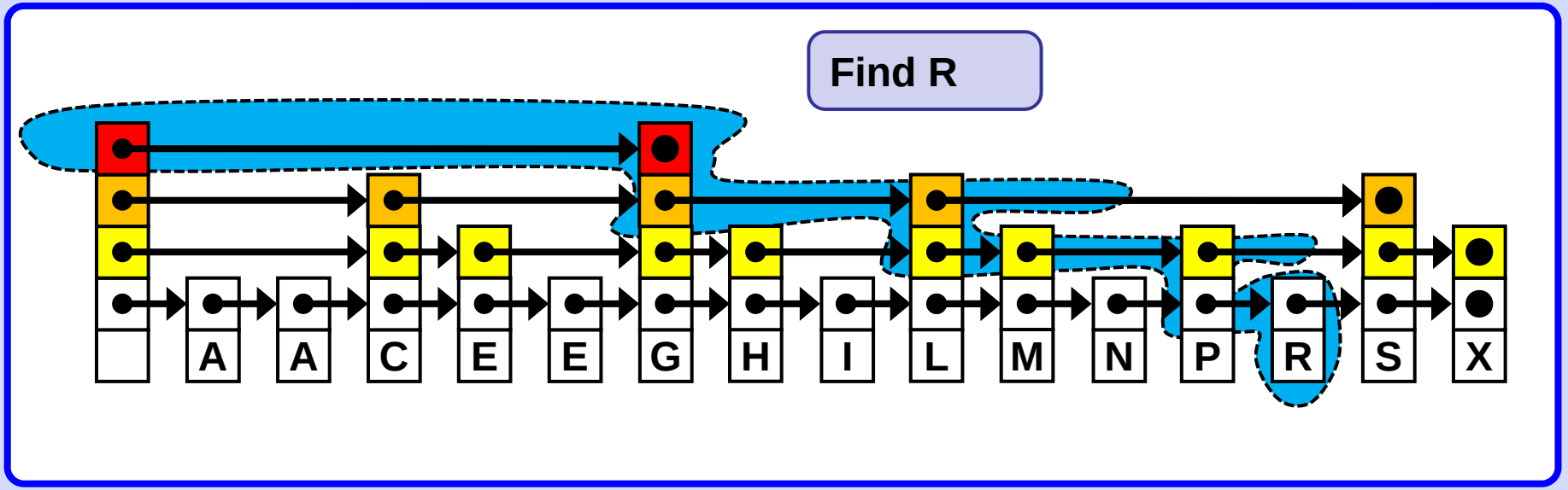
A level is chosen for an element in effect by flipping a coin that has probablility *p* of coming up heads. We keep flipping until we get "tails" or until the maximum number of levels is reached.

```
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

We scan through the top list until we find the search key or a node with a smaller key that has a link to a node with a larger key.

Then, we move to the second-from-top list and iterate the procedure, continuing until the search key is found
or a search miss happens on 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.

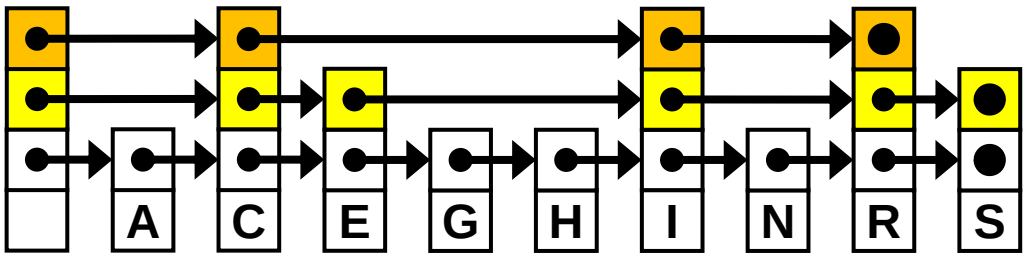
```
Node search(List list, int searchKey) {
  Node x = list.header;

  // loop invariant: x.key < searchKey
  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;
}
```

Keeping the code simple

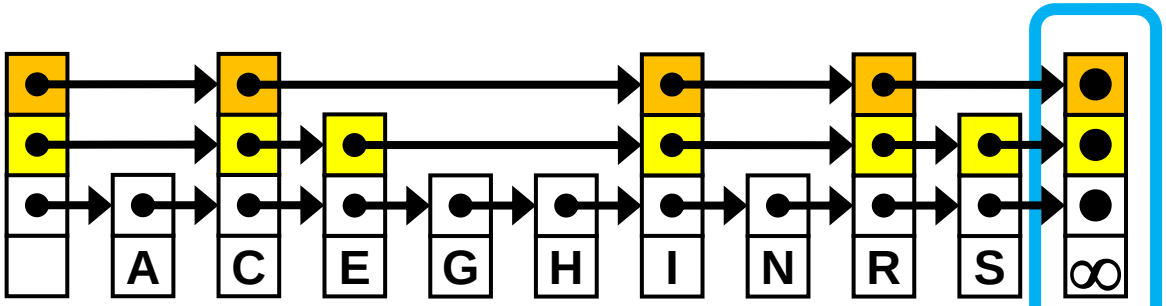
```
while(x.forward[i].key < searchKey) // x.forward[i] == null?
```

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

The level of the sentinel is the same as the whole list level.

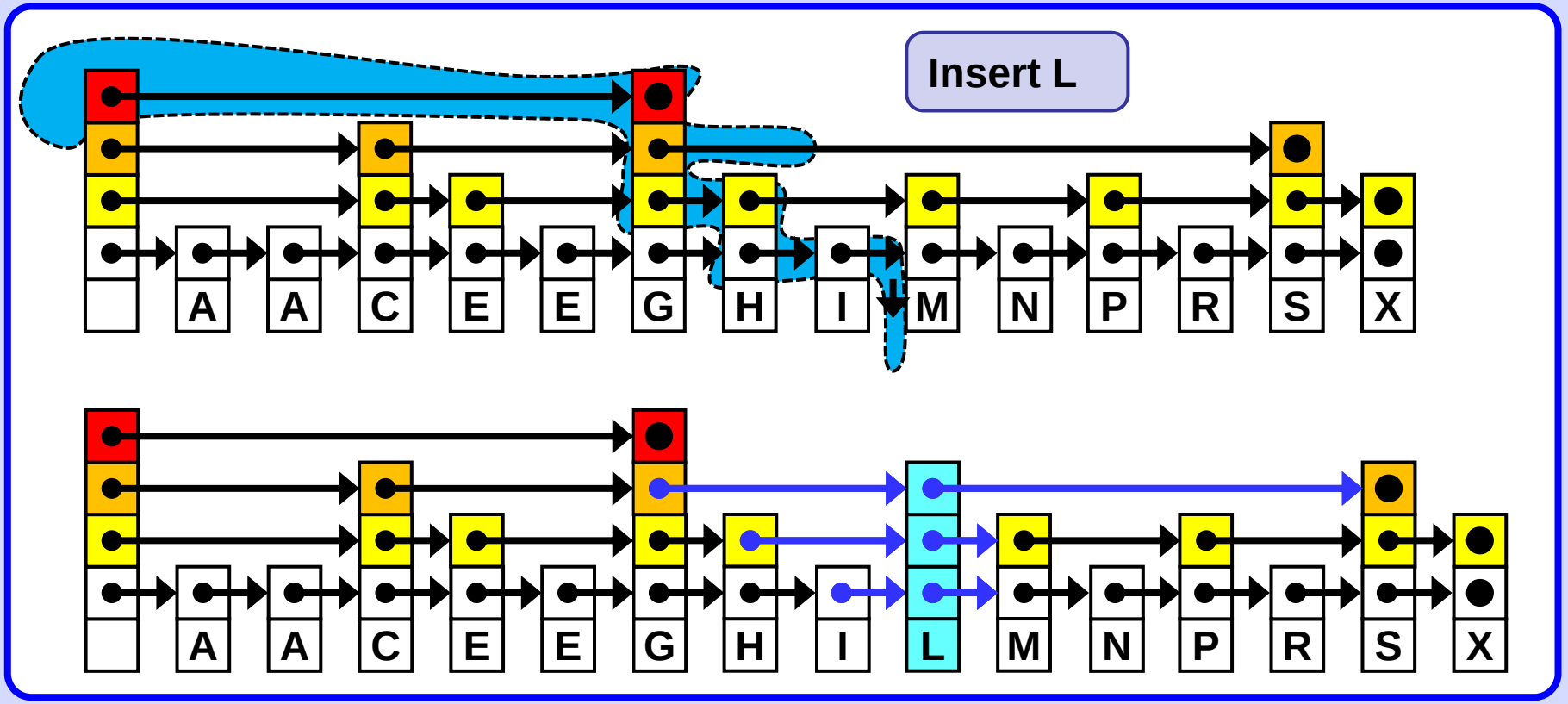


Note that in the other diagrams the **sentinel** is not displayed to spare space in the presentation.

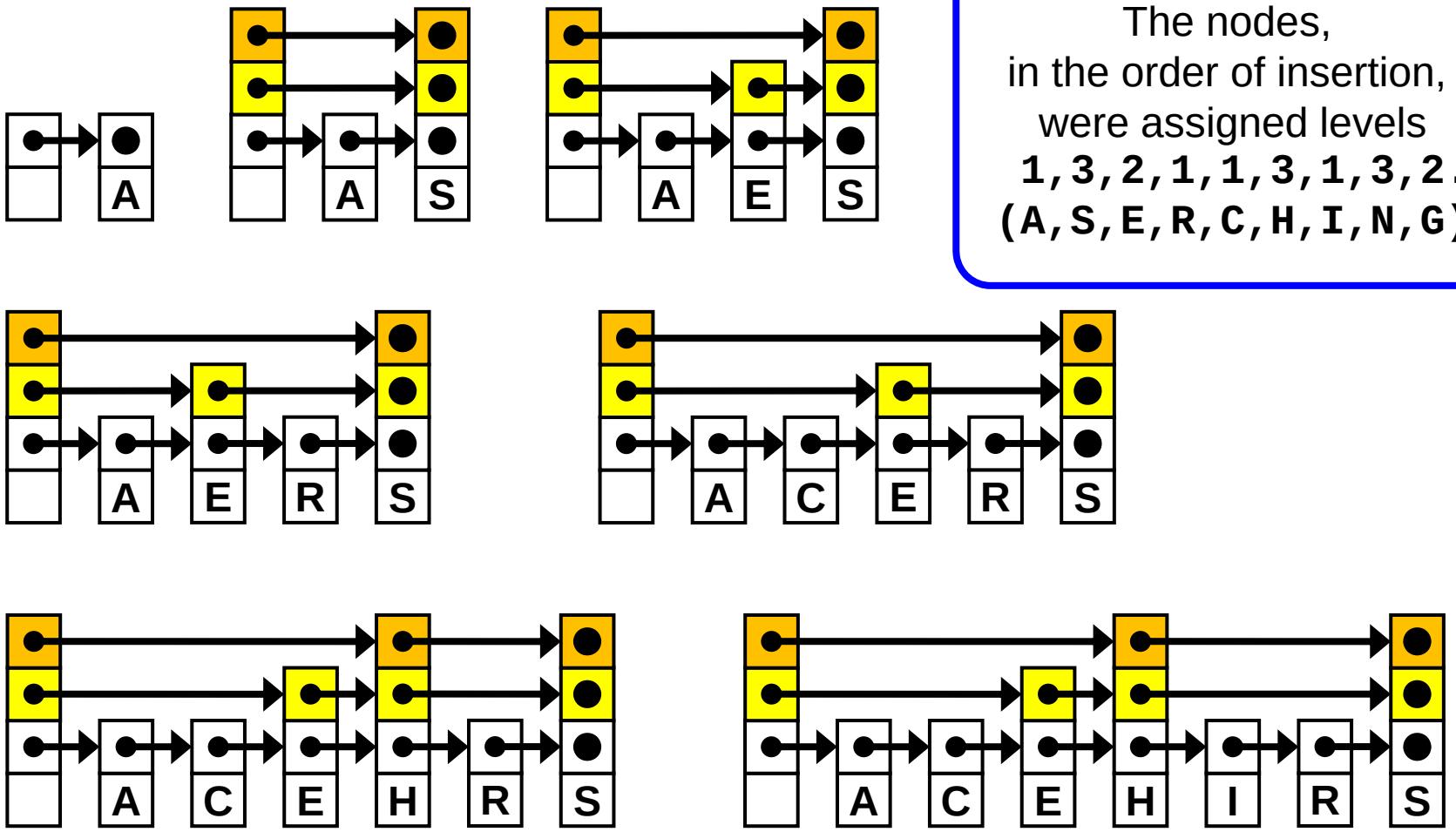


Insert

Find the place for the new element.
Assign to it its level ***k*** computed by flipping the coin.
Insert the element into each of those ***k*** lists, starting at the bottom.



Insert A, S, E, R, C, H, I, N, G.

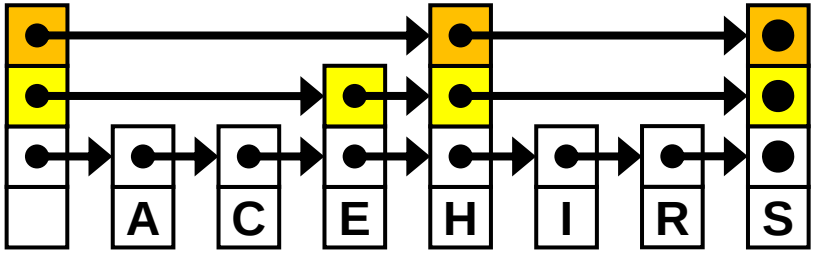


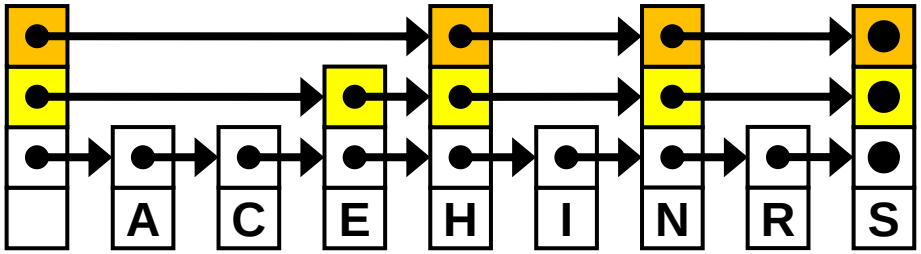
The nodes,
in the order of insertion,
were assigned levels
1,3,2,1,1,3,1,3,2.
(A,S,E,R,C,H,I,N,G)

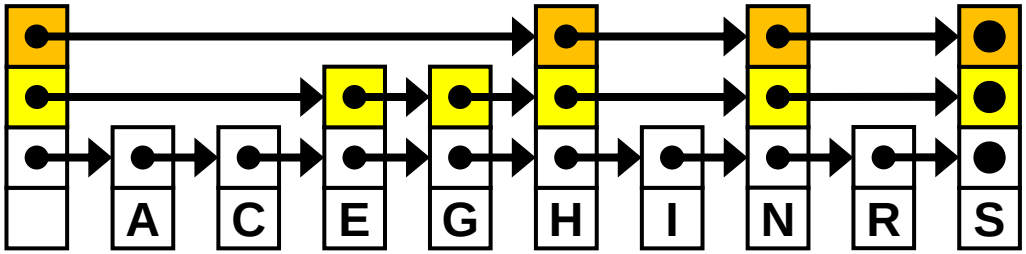
continue...

.. continued

Insert A, S, E, R, C, H, I, N, G.

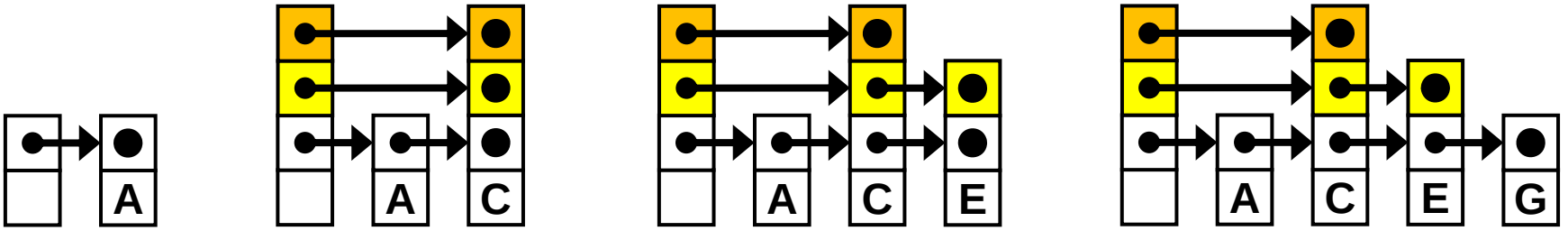


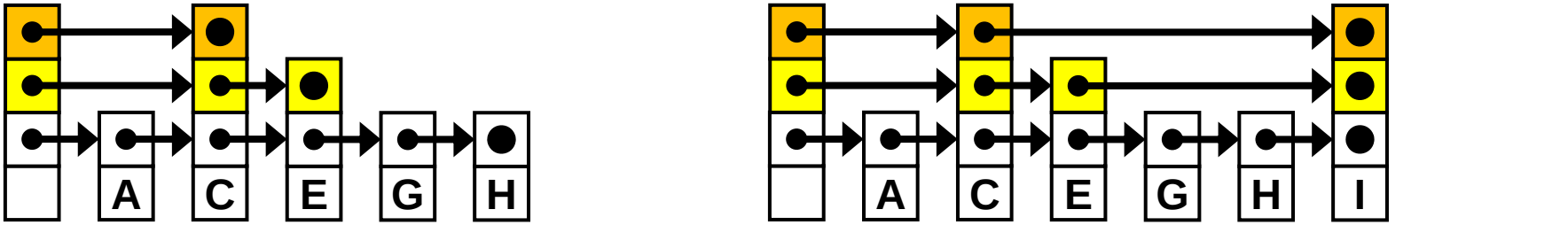


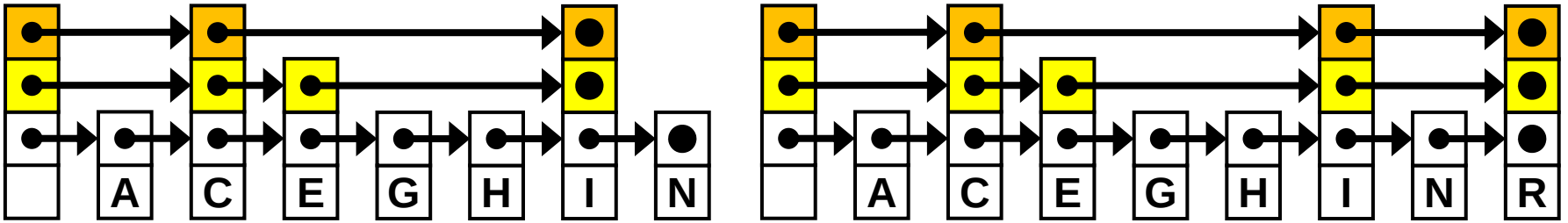


The nodes,
in the order of insertion,
were assigned levels
1,3,2,1,1,3,1,3,2.
(A,S,E,R,C,H,I,N,G)

Insert A, C, E, G, H, I, N, R, S. (Same values, different order)

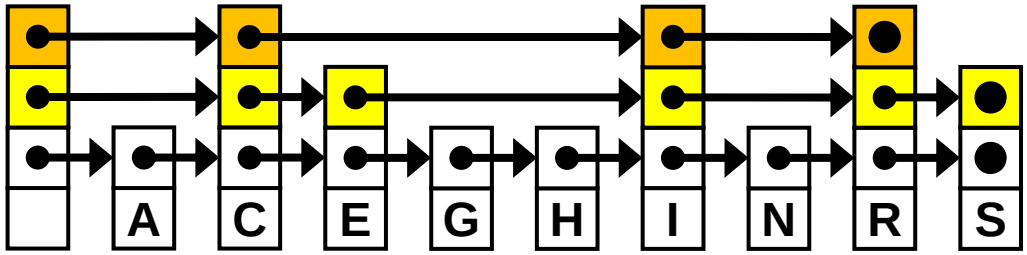






continue...

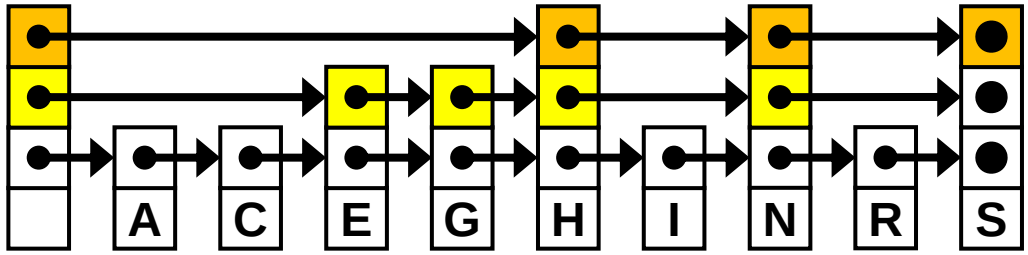
.. continued



The nodes were inserted in sorted order.

The nodes,
in the order of insertion,
were assigned levels
1,3,2,1,1,3,1,3,2.
(A,C,E,G,H,I,N,R,S)

The result of the previous example



The nodes were inserted in random order.

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)

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

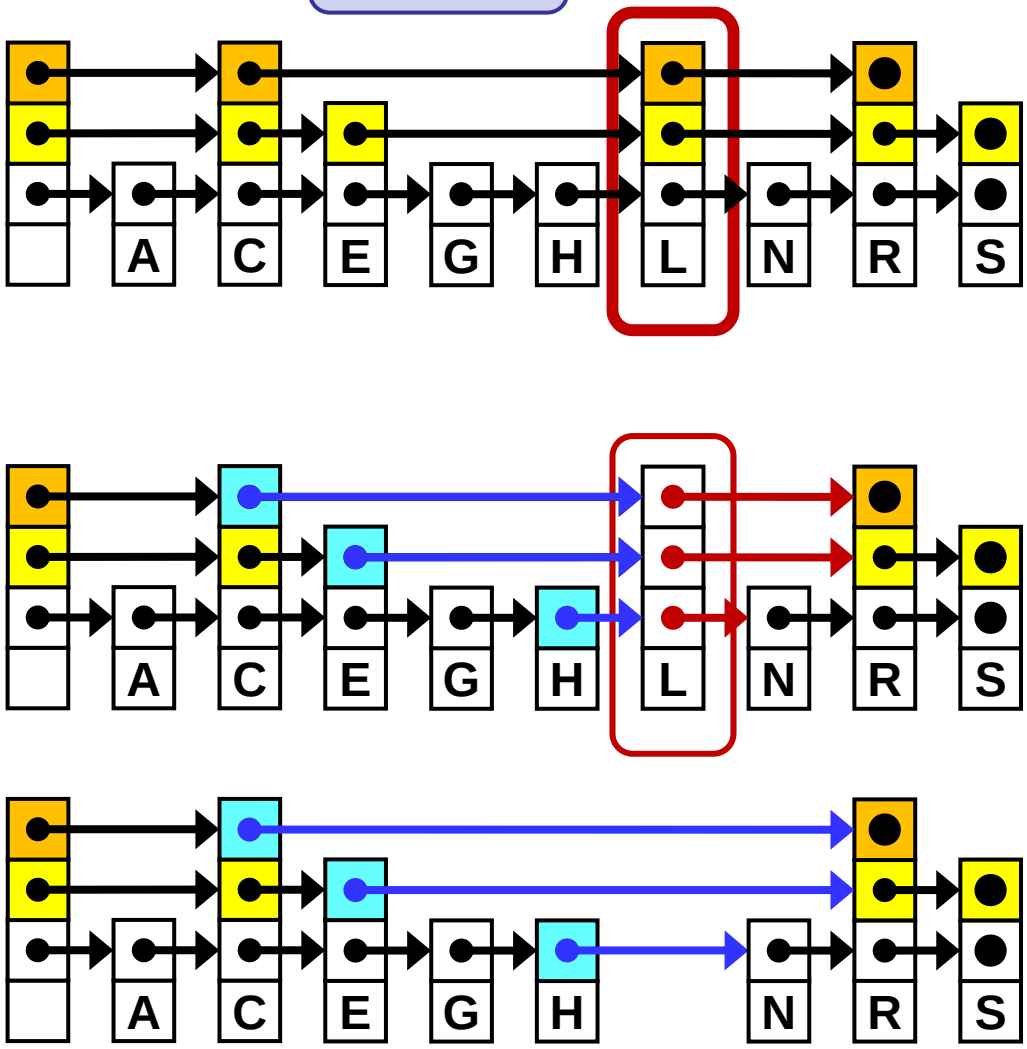
```
// Utilise update which is a (vertical) array
// of pointers to the elements which will be
// predecessors of the new element.

void insert(List list, int searchKey, Data newValue){

  Node x = list.header;
  for (int i = list.level; i >= 1; i--) {
    while (x.forward[i].key < searchKey)
       x = x.forward[i];
    //note: x.key < searchKey <= x.forward[i].key
    update[i] = x;
  }

  x = x.forward[1];
  if (x.key == searchKey)
       x.value = newValue;
  else { // key not found, do insertion here:
```

continue...

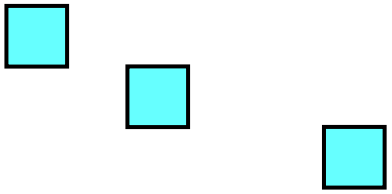
.. continued

```
... else { // key not found, do insertion here:
  int newLevel = randomLevel(list);

  /* If the 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, we will increase the level of the
  list by at most one on each insertion. */
  if (newLevel > list.level) {
    newLevel = list.level + 1; list.level = newLevel;
    update[newLevel] = list.header;
  }

  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
```

Delete L



registered in the **`update`** array



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 finds 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.

```
// update is an array of pointers to the
// predecessors of the element to be deleted.
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 ...
```

continue...

(**) 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.

.. continued

```
    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 compexity

The average number of links in a randomized skip list with parameter p is
$(p/(p-1)) \cdot N$

The average number of comparisons in **search** and **insert**
in a randomized skip list with parameter p is on average

$(p \log_p (N)) / 2 = \log(N) * p / (2 \log (p))$

Experimental time comparisons:

	Search		Insert		Delete	
Skip list	0.051	**(1.0)**	0.065	**(1.0)**	0.059	**(1.0)**
AVL tree	0.046	(0.91)	0.100	(1.55)	0.085	(1.46)
2-3 tree	0.054	(1.05)	0.210	(3.2)	0. 21	(3.65)
Splay tree	0.490	(9.6)	0.510	(7.8)	0.53	(9.0)

Times in ms on some antiquitated HW [Pugh, 1990]

Notes on compexity

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.

erikdemaine.org/

- Erik Demaine's presentation at MIT
  http://videolectures.net/mit6046jf05_demaine_lec12/

- Robert Sedgewick: *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching, Third Edition*, Addison Wesley Professional, 1998

- William Pugh: *Skip lists: A probabilistic alternative to balanced trees*. Communications of the ACM, 33(6):668–676, 1990.

- William Pugh: *A Skip List Cookbook* [http://cglab.ca/~morin/teaching/5408/refs/p90b.pdf]

- Bradley T. Vander Zanden: [http://web.eecs.utk.edu/~huangj/CS302S04/notes/skip-lists.html]