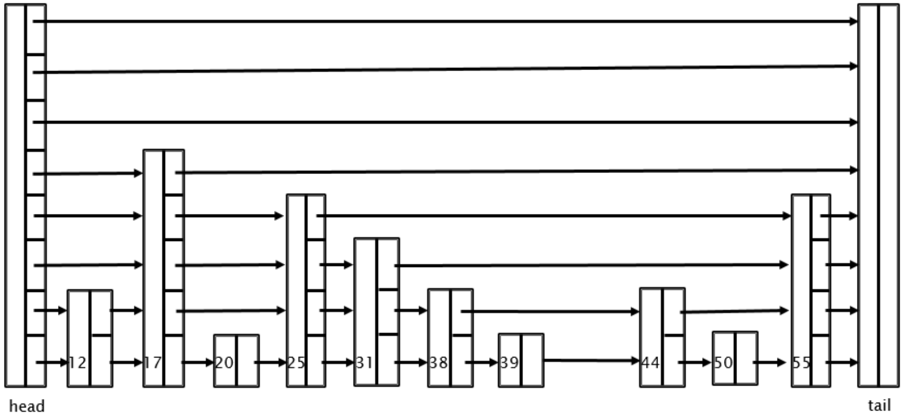


CSSE 230 Day 25

Skip Lists

After today, you should be able to …
… explain the idea of probabilistic skip lists
… implement skip list insertion and deletion

Announcements

- I will be off campus for much of Weds – Monday.
- Thursday and Friday's classes are on Binary Heaps and Heap Sort.
- They can be done:
  - As normal (I will be in class and there will be worktime in class to ask questions)
  - As self-study (completed quiz packet will be graded and count as attendance)

Skip Lists

An alternative to balanced trees

Sorted data.

Random.

Expected times are O(log n).

An alternative to AVL trees

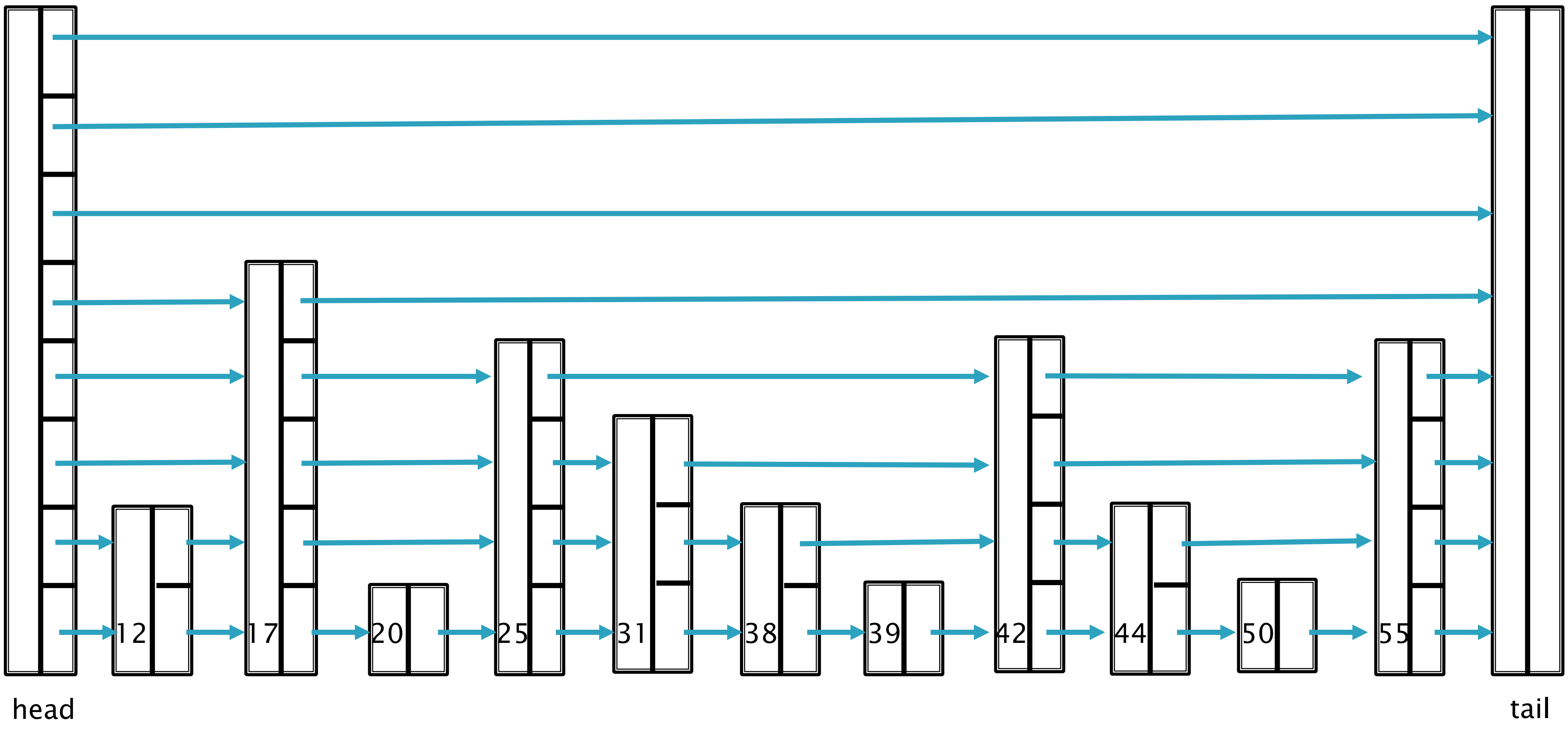
- Indexed lists.
  - One-level index.
  - 2nd-level index.
  - 3rd-level index
  - log-n-level index.
- Problem: insertion and deletion.

Remember the problem with keeping trees *completely* balanced"?

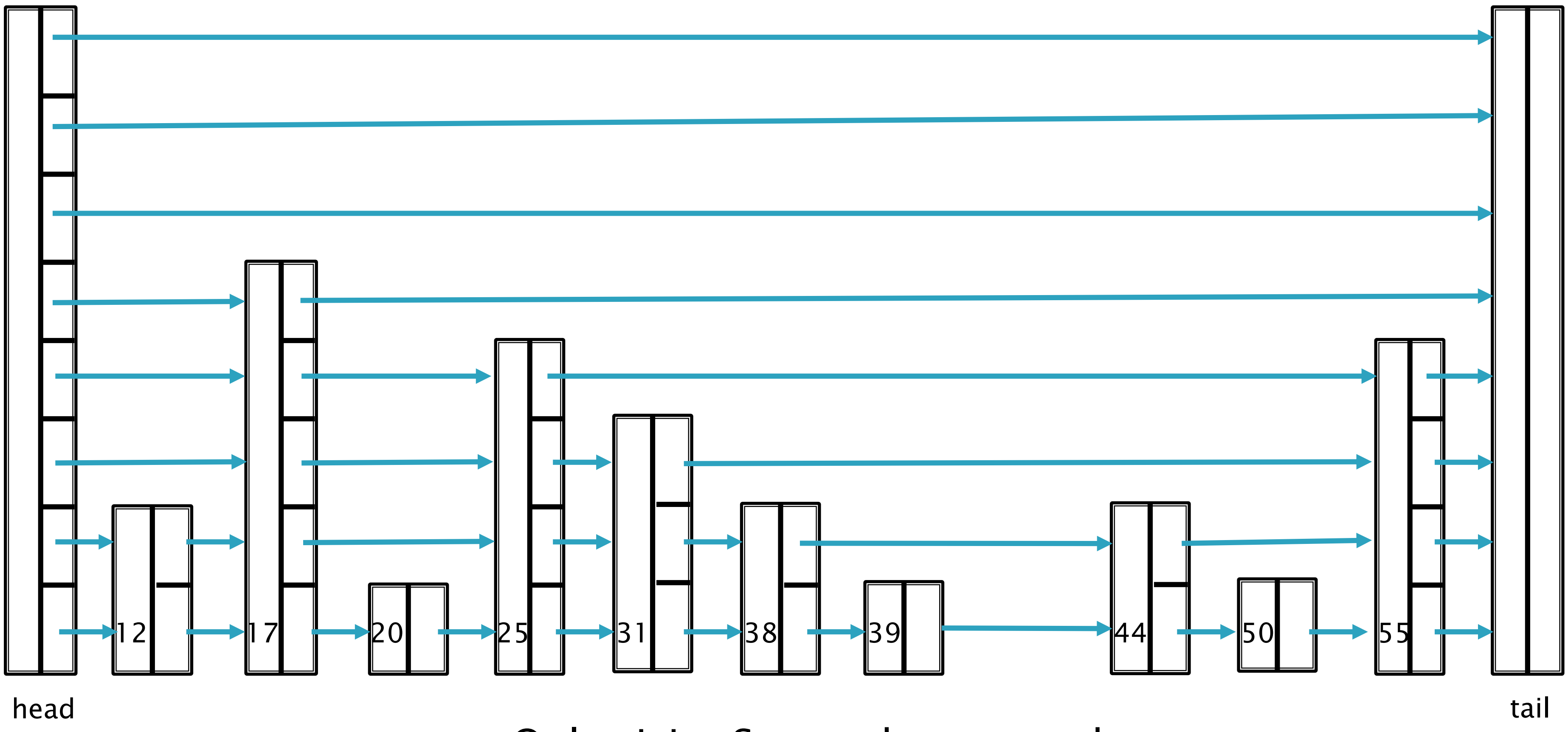
- Solution: Randomized node height: Skip lists.
  - Pugh, 1990 CACM.
- http://www.cs.umd.edu/class/spring2002/cmsc420-0401/demo/SkipList2/
  - Applet, certain browsers may reject

Note that we can iterate through the list easily and in increasing order

SkipList representation:
Each node has a list of links

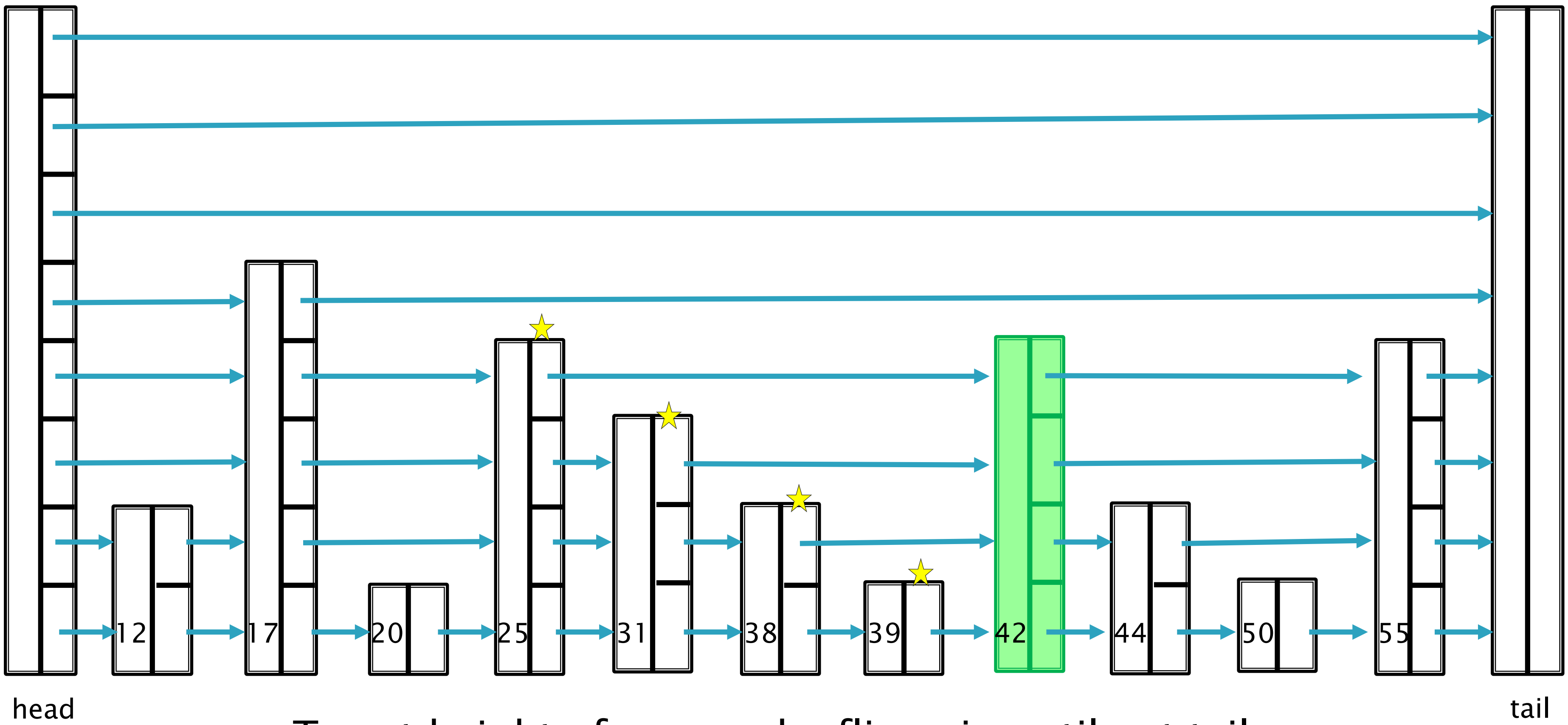


Search for 50:
Start at top, look ahead, and work down.



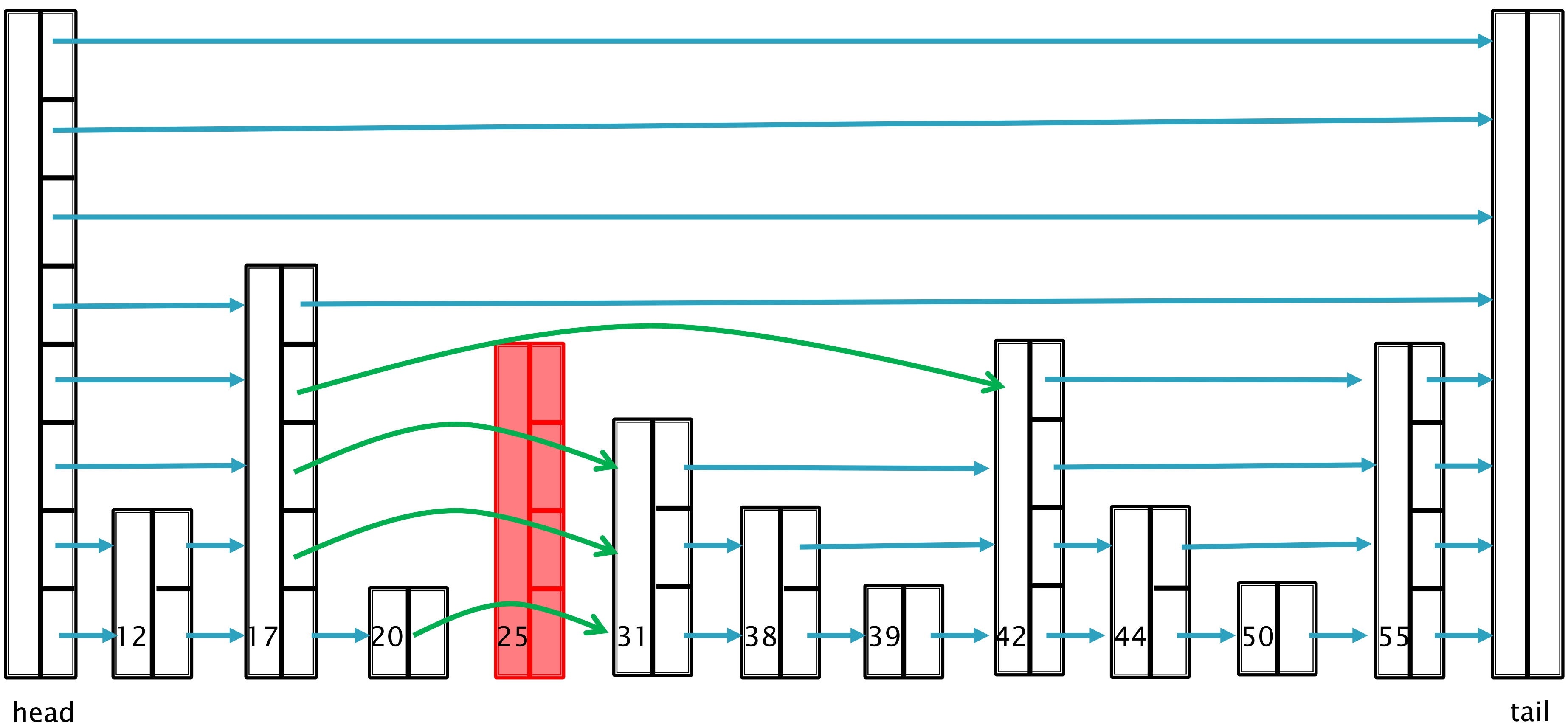
Only visits 6 non-dummy nodes

Insert 42:
Make new node. Find list of previous nodes. Then update links.



To set height of new node: flip coin until get tails

Delete 25:
Find list of previous nodes. Then update links.



Next slides show an alternative representation we won't use, but with more detail

- Uses a bit more space.
- Michael Goodrich and Roberto Tamassia.

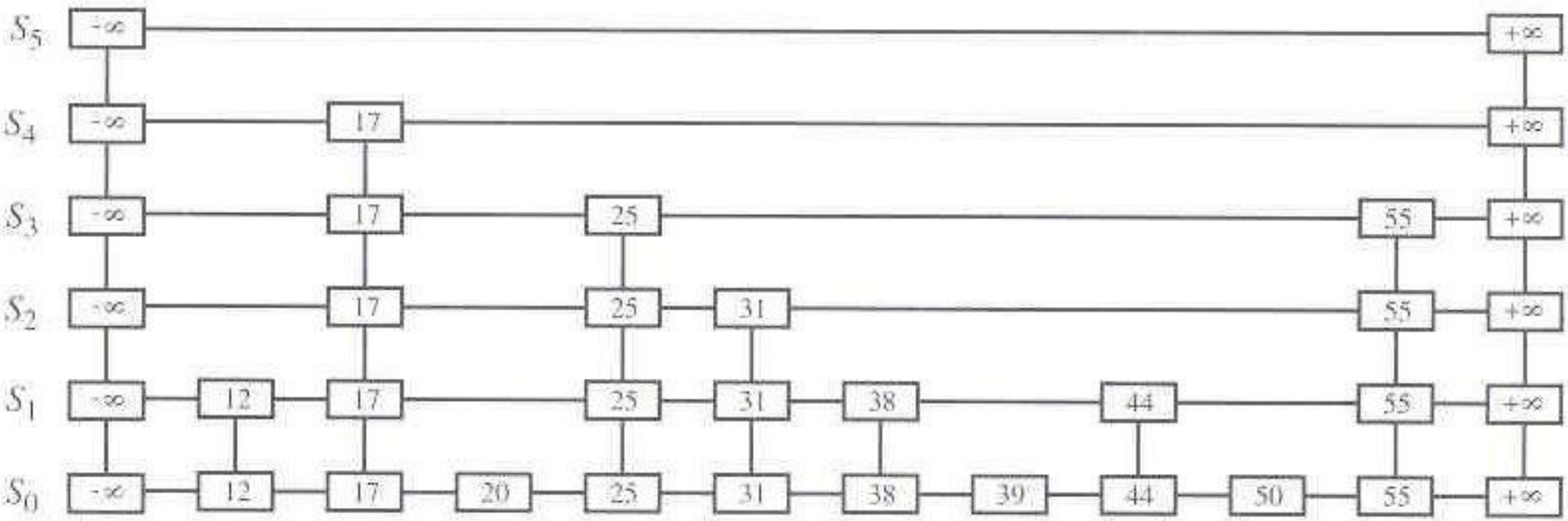


Figure 8.9: Example of a skip list.

Methods

after(p): Return the position following p on the same level.

before(p): Return the position preceding p on the same level.

below(p): Return the position below p in the same tower.

above(p): Return the position above p in the same tower.

Search algorithm

1. If S.below(p) is null, then the search terminates—we are ***at the bottom*** and have located the largest item in S with key less than or equal to the search key k. Otherwise, we ***drop down*** to the next lower level in the present tower by setting $p \leftarrow S$.below(p).
2. Starting at position p, we move p forward until it is at the right-most position on the present level such that key(p) $\leq k$. We call this the ***scan forward*** step. Note that such a position always exists, since each level contains the special keys $+\infty$ and $-\infty$. In fact, after we perform the scan forward for this level, p may remain where it started. In any case, we then repeat the previous step.

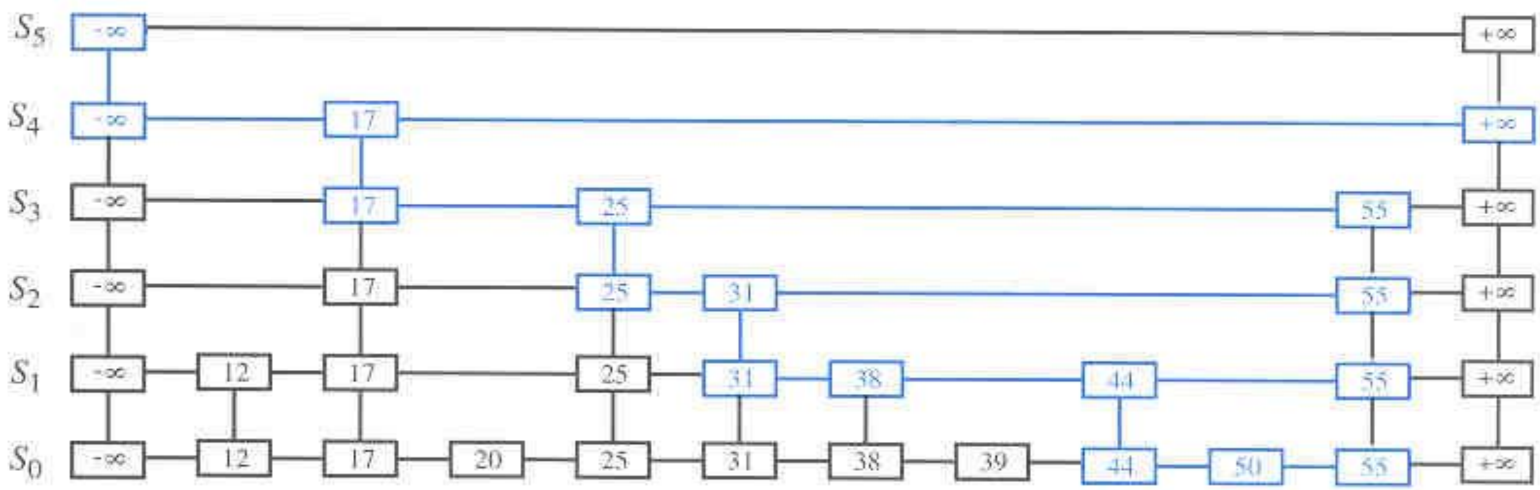
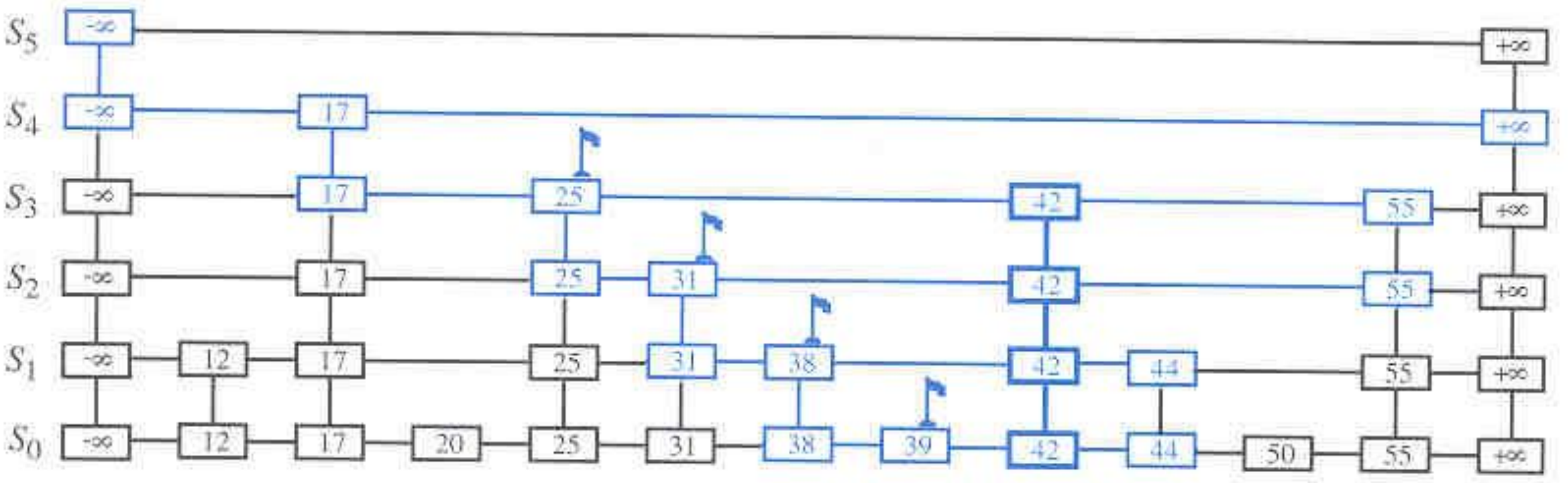
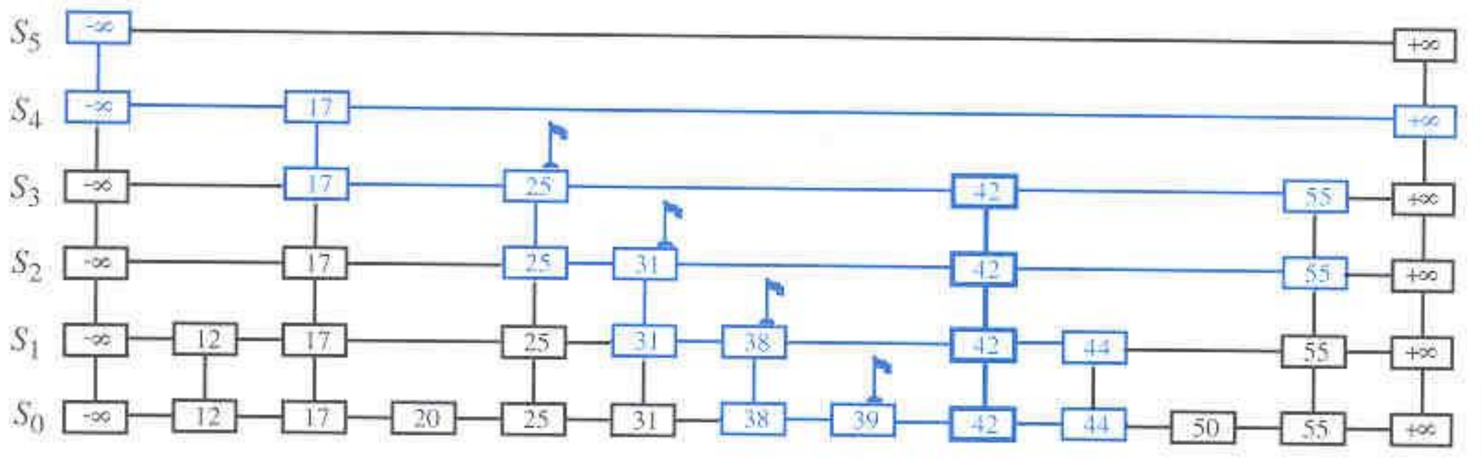


Figure 8.10: Example of a search in a skip list. The positions visited when searching for key 50 are highlighted in blue.

Insertion diagram



Insertion algorithm



Algorithm SkipInsert(k, e):

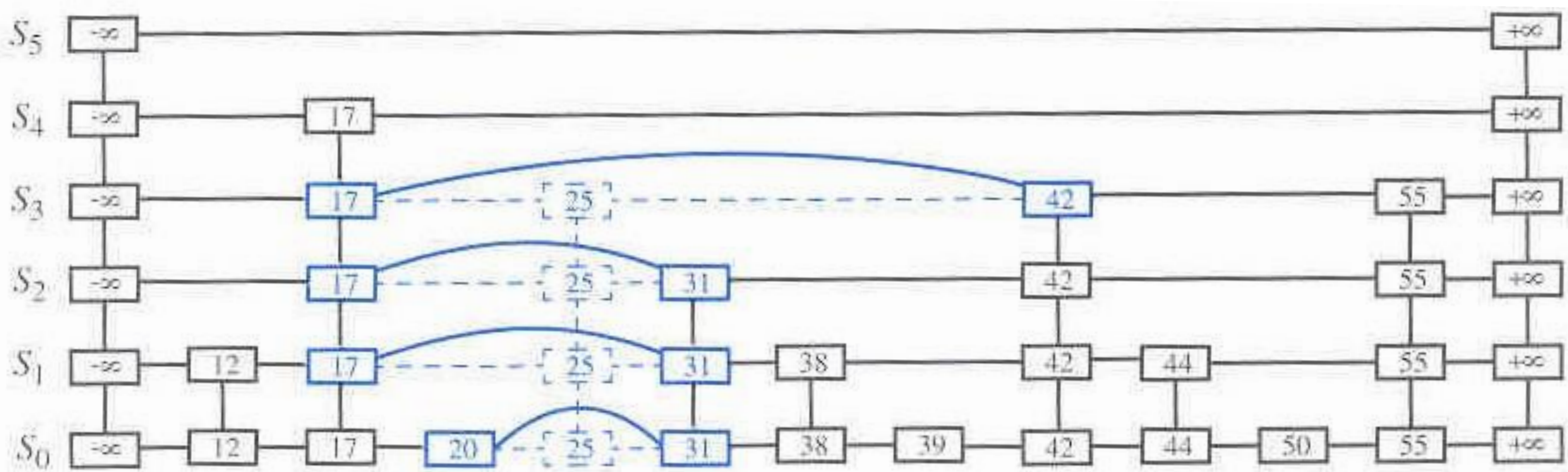
Input: Item (k, e)

Output: None

```
p ← SkipSearch(k)
q ← insertAfterAbove(p, null, (k, e))          {we are at the bottom level}
while random() < 1/2 do
   while above(p) = null do
      p ← before(p)            {scan backward}
   p ← above(p)            {jump up to higher level}
   q ← insertAfterAbove(p, q, (k, e))          {insert new item}
```

Code Fragment 8.5: Insertion in a skip list, assuming random() returns a random number between 0 and 1, and we never insert past the top level.

Remove algorithm



(sort of) Analysis of Skip Lists

- No guarantees that we won't get O(N) behavior.
  - The interaction of the random number generator and the order in which things are inserted/deleted *could* lead to a long chain of nodes with the same height.
  - But this is **very** unlikely.
  - ***Expected*** time for search, insert, and remove are O(log n).