Data structures are often needed to provide organization for large sets of data.

However, traditional approaches offer a tradeoff between insertion/deletion and search performance:

Contiguous storage (e.g., a sorted array):

- worst/average search cost $\Theta(\log N)$ where N is the number of data elements

- insert/delete cost $\Theta(N)$

Linear linked storage (e.g., a linked list):

- worst/average search cost $\Theta(N)$

- insert/delete cost $\Theta(1)$

Balanced binary trees (e.g., an AVL tree):

- worst/average search cost $\Theta(\log N)$

- insert/delete cost $\Theta(\log N)$

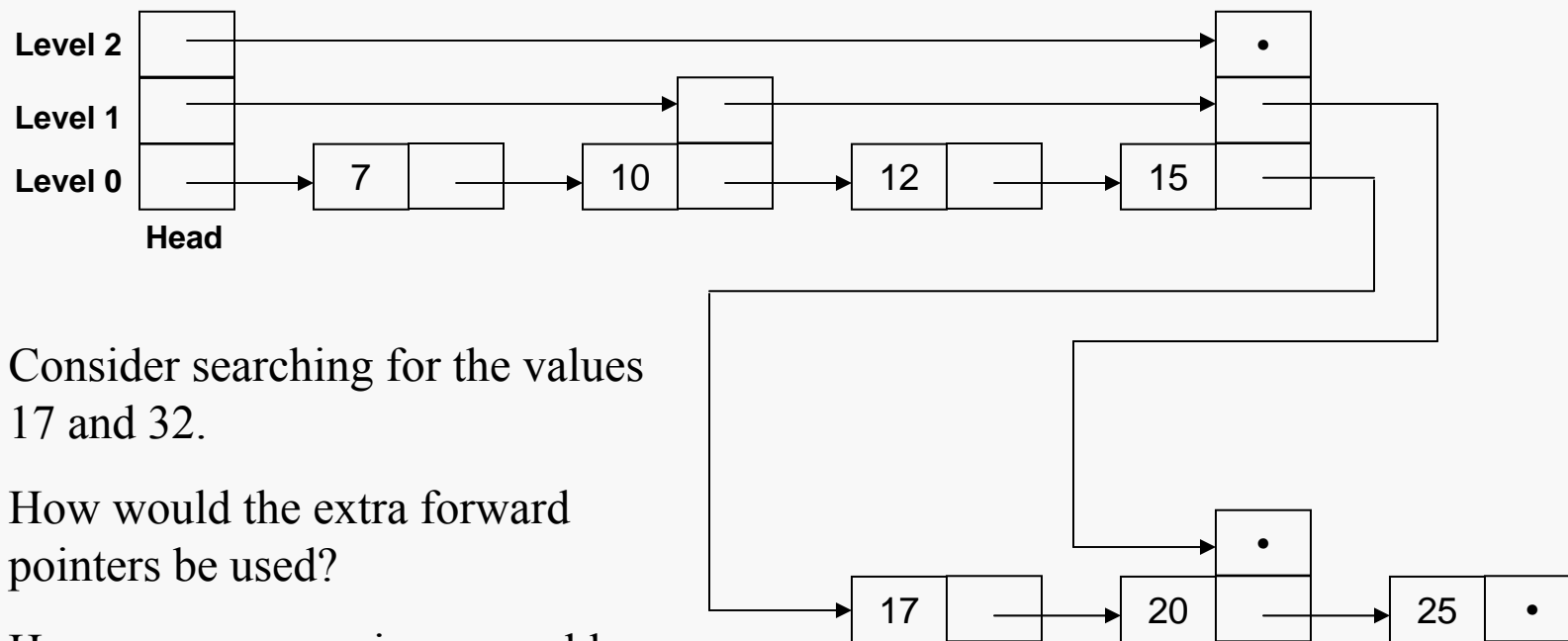> **Unfortunately, balanced binary tree implementations are quite complicated.**
>
> **We'd like similar performance with less complexity…**

Linear linked structures are relatively simple to implement, and well-understood.

We can improve search costs by adding some additional pointers to selected nodes to allow "skipping" over nodes that can safely be ignored.

Consider:
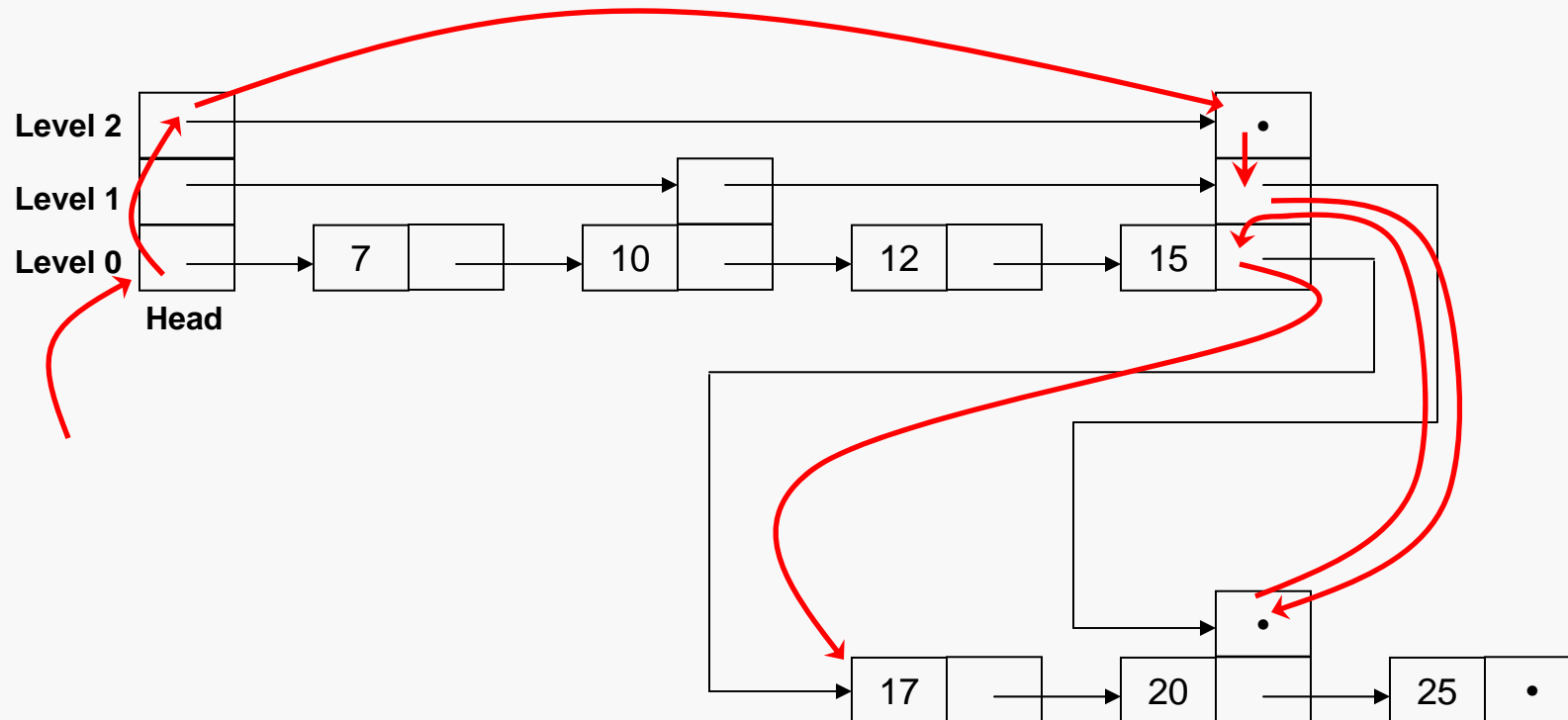

Consider searching for the values 17 and 32.

How would the extra forward pointers be used?

How many comparisons would be required?

"Skip lists: a probabilistic alternative to balanced trees", CACM, W. Pugh, 1990

Consider searching for the value 17:

By using the extra pointers, we can jump over the first half of the list, and then determine that the value does not lie within the fourth quarter of the list. A careful count of operations doesn't show any advantage for this tiny list, but…

We can view the list as consisting of a hierarchy of parallel, intersecting sub-lists or levels.
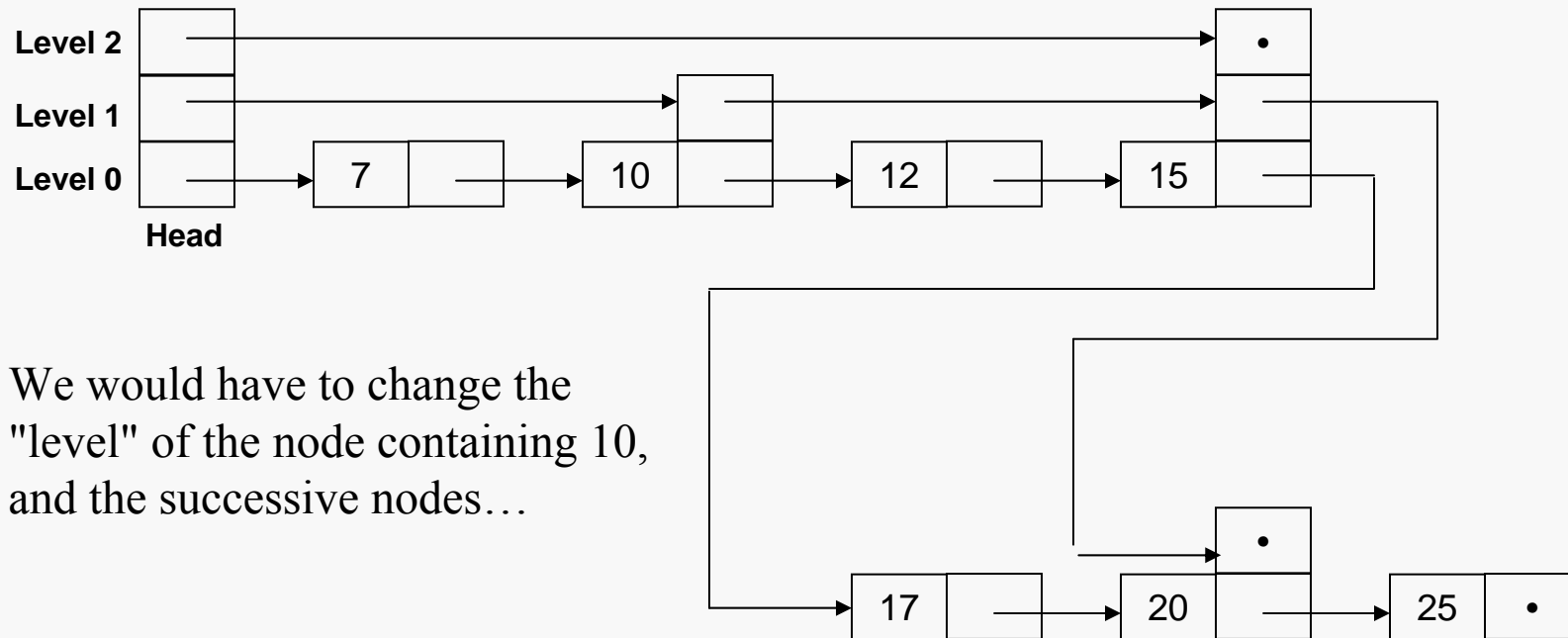
Level 1 is a subset of level 0, level 2 a subset of level 1, and so forth.

By convention, we say each node belongs to level K-1 if it contains K forward pointers.
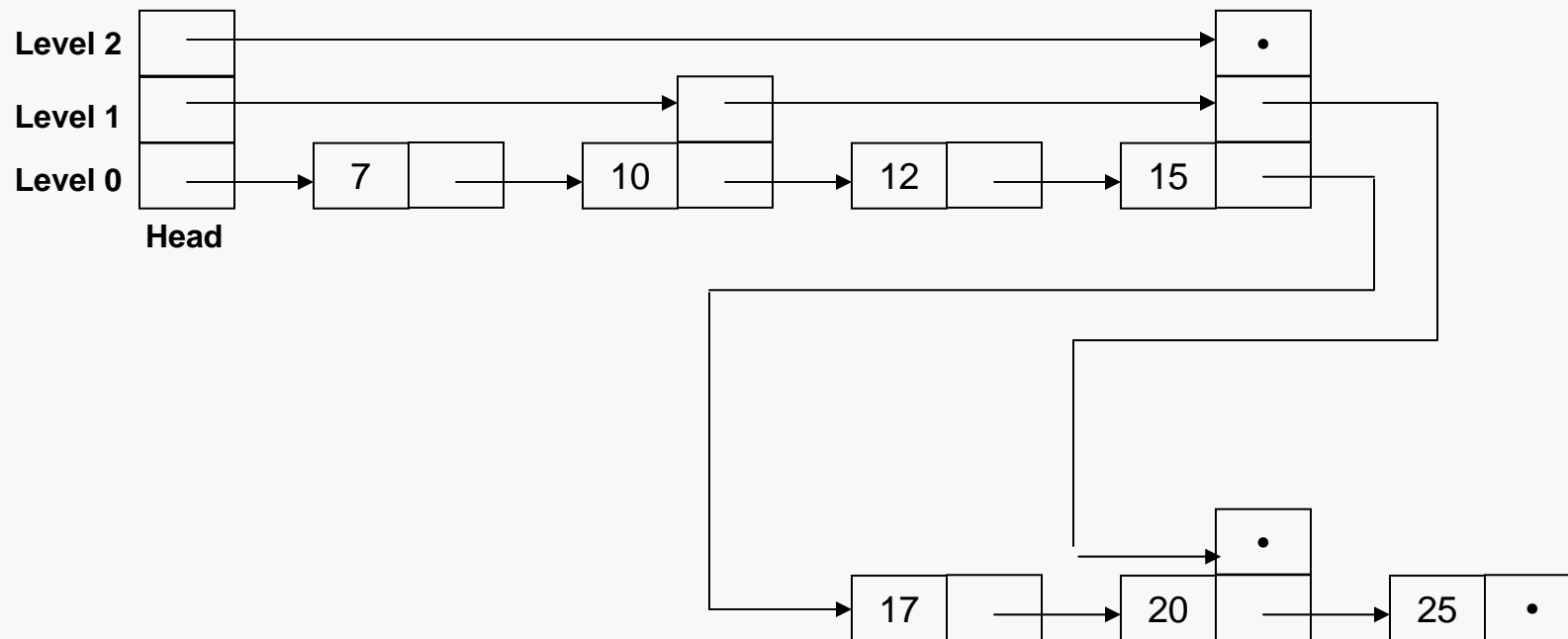
Unfortunately, inserting new nodes into this "ideal" skip list structure is very expensive.

Consider inserting the value 8:

We would have to change the "level" of the node containing 10, and the successive nodes…

… must the structure be so regular in order to provide improved performance?

In the "ideal" skip list:

- 1/2 the nodes are in level 0, 1/4 in level 1, 1/8 in level 2, and so forth.

- nodes in level 0 point to the next node; nodes in level 1 to the next and second-next node; nodes in level 2 to the next, second-next and fourth-next nodes; and so forth

- the level of a node is determined entirely by its position in the list

We can reduce the cost of insertion by:

- selecting a <u>random</u> level for the new node, so that the proportion of level 0, level 1, etc., nodes is roughly preserved

- level 0 nodes point to the next node;

  level 1 nodes also point to the next node that belongs to level 1 or higher;

  level 2 nodes also point to the next node that belongs to level 2 or higher;

  … and so forth

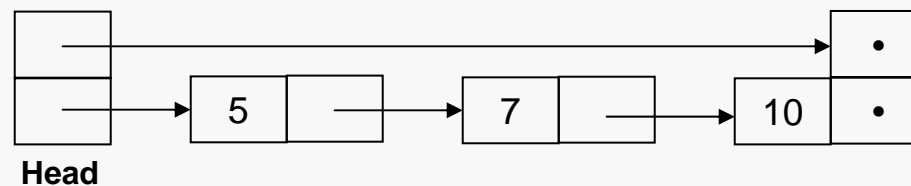Empty list configured to provide 1 level:

Head

Insert the value 10; assume that level 1 is selected:

10

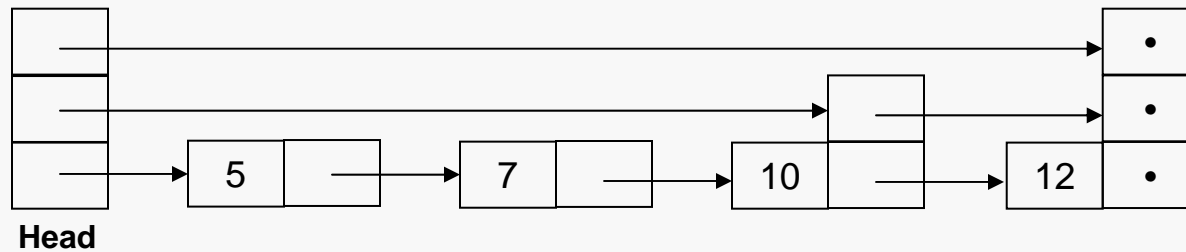**Head**
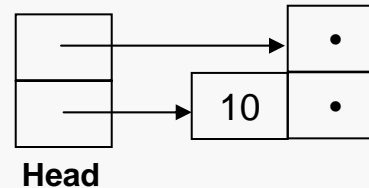
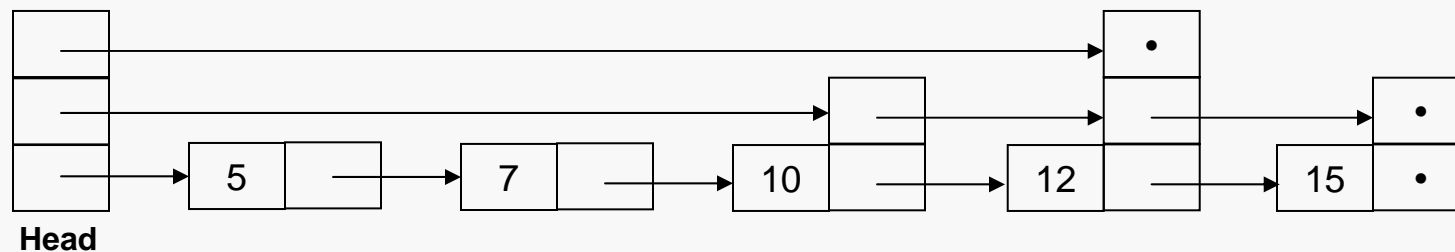Insert the value 5 at level 0 and then insert the value 7 at level 0:

5   7   10

Head

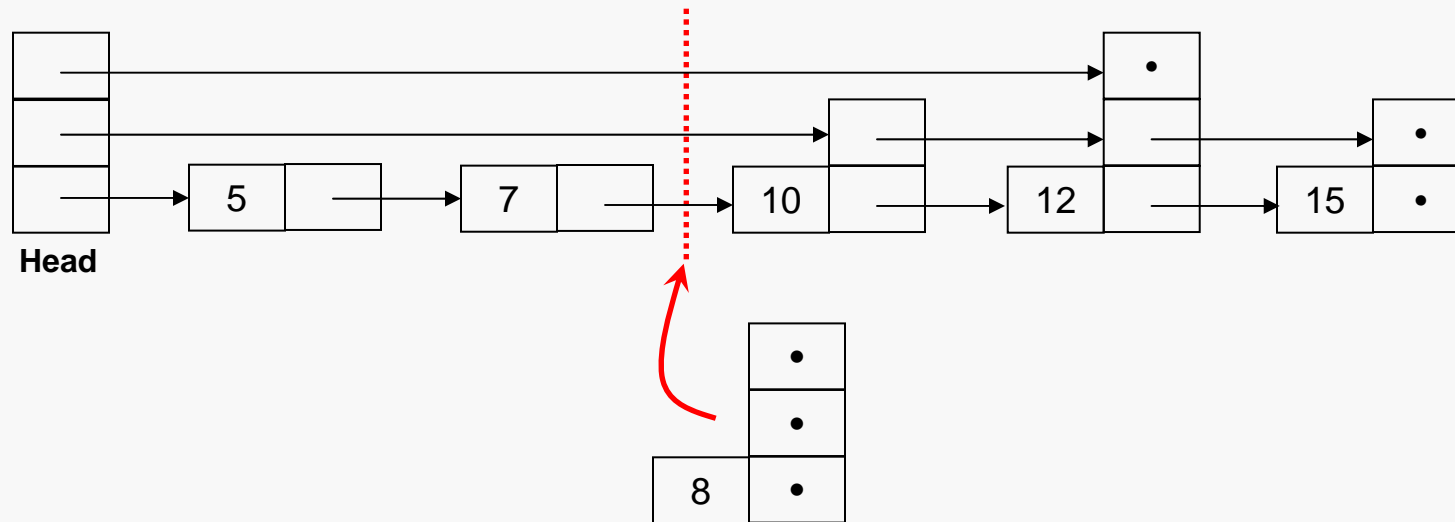Insert the value 12 at level 2:

Head

And then insert the value 15 at level 1:

Head

Consider searching for a few values, say 7 and 15. In the former case, there's no advantage over a simple list; in the latter, there's a substantial advantage.

Consider inserting the value 8 into the skip list below, and assume that the new node is assigned to level 2:

The first step is to search for the largest value already in the list that is less than or equal to the new value. The new node will become the level-0 successor of that existing node.

To complete the insertion, we must modify pointers to and from the new node.

But which pointers need to be modified?

Precisely the pointers that break the dashed line and are at a level containing the new node.

During the search phase, we must remember the nodes that contain pointers that "go past" the insertion point for the new node.

To do this, we need an array that can hold up to $N + 1$ node pointers, where N is the maximum number of levels among the existing list nodes.
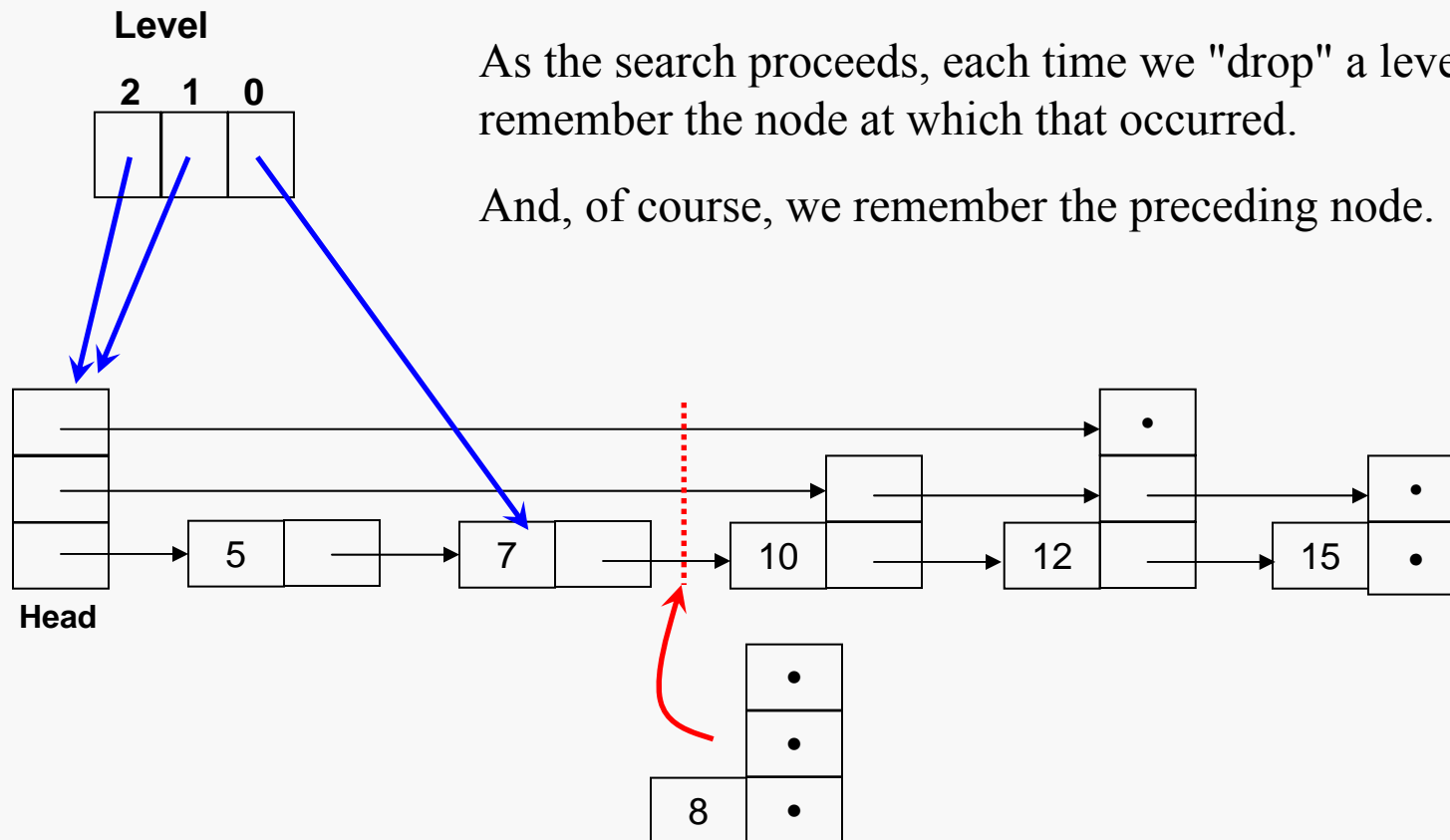
As the search proceeds, each time we "drop" a level we remember the node at which that occurred.

And, of course, we remember the preceding node.

At this point we have sufficient information to perform the necessary pointer updates.
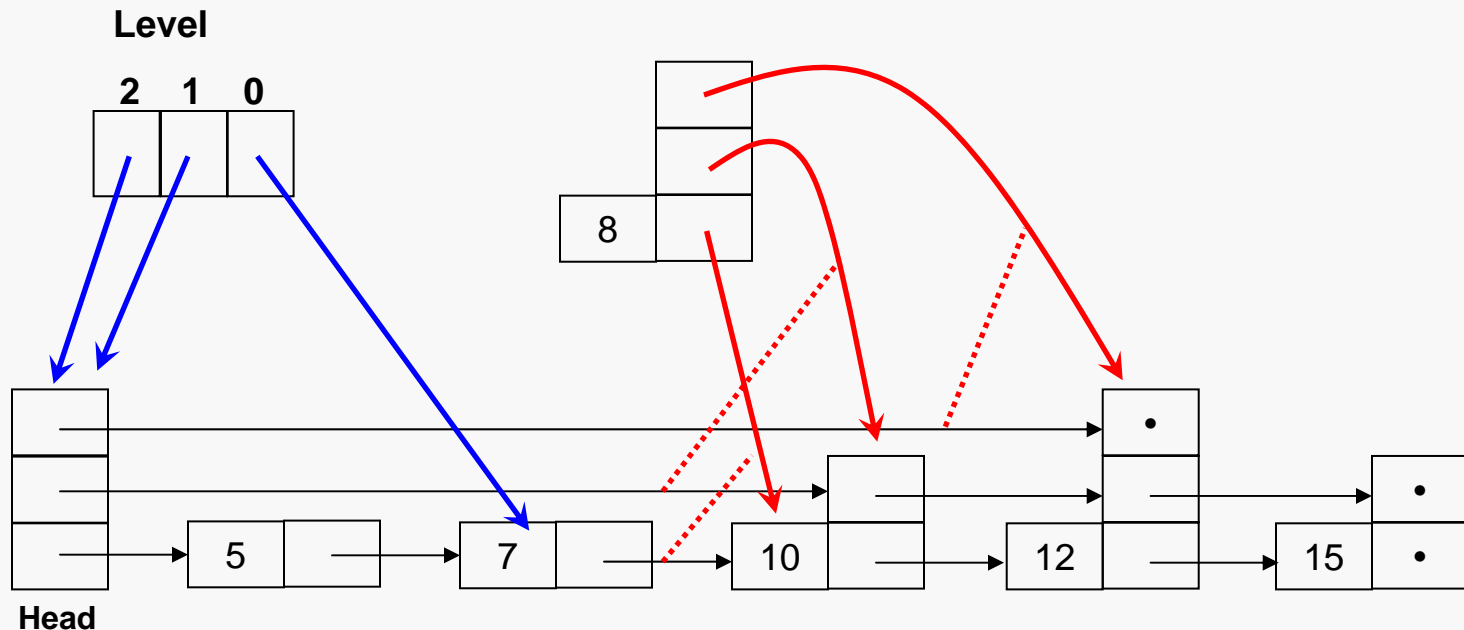
The pointers <u>from</u> the new node are set using the pointers identified earlier, taking into account the levels.

Then, the pointers <u>to</u> the new node are set, again taking levels into account:
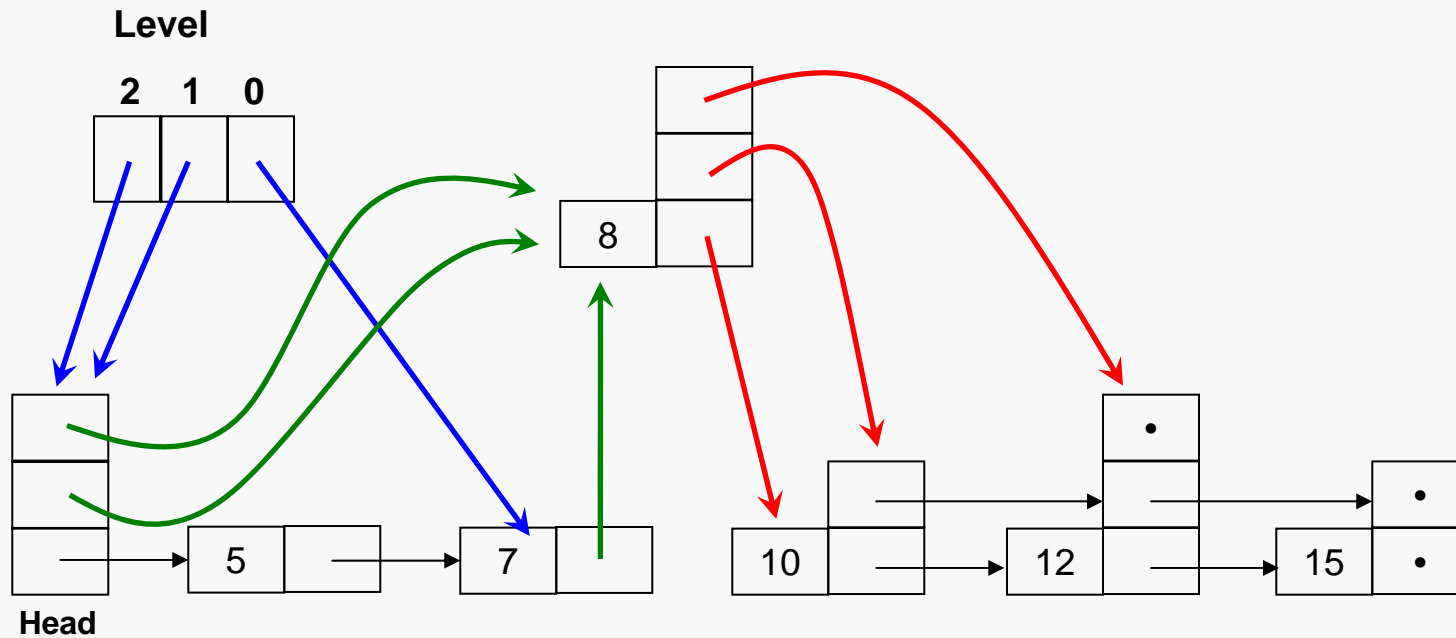
In practice, resetting the pointers <u>from</u> and <u>to</u> the new node would be interleaved so only one pass through the list of "passed" pointers is needed.

Given a data value (that may or may not occur in the list):

```
Node* Update[MaximumLevel]          ; for remembering "pass" nodes
Node* x := &HeadNode;


for i := MaximumLevel downto 0 do

    ; go as far as possible on current level
    while x->forward[i]->keyField < searchKey do
      x := x->forward[i]
    endwhile

    ; search level will now change
    Update[i] := x                  ; remember node for updating later
    ; drop to previous level (via for loop)
endfor


x := x->forward[0]                  ; predecessor is in next Node
. . .
```

Adapted from Pugh, 1990

The first part of the algorithm has found the predecessor node, now it's time for the physical insertion:

```
. . .
if x->keyField = searchKey then
    ; take action for case of duplicate key value
else
    Lvl := randomLevel()      ; select random level for new node
    if Lvl > MaximumLevel then  ; adjust if list needs new level
        for i := MaximumLevel + 1 to Lvl do
            Update[i] := &HeadNode
        endfor
        MaximumLevel := Lvl
    endif
    ; create new node
    x := makeNode(Lvl, searchKey, Data)
    ; patch it into the list, updating "pass" pointers
    for i := 0 to MaximumLevel do
        x->forward[i] := Update[i]
        Update[i]->forward[i] := x
    endfor
endif
```

Each new node may be given a randomly-chosen level, but that will almost certainly be disastrous because it will lead to lots of unnecessary levels.

There are a number of schemes for restricting the level selection without sacrificing the degree of randomness that is necessary to achieve a high probability of good performance.

One approach is to "cap" the number of levels; for example if we allow no more than 32 levels then in principle the list can store 2^{32} data values. A small cap is not desirable.

Another approach is "don't worry, be happy". Trust a sensible random scheme to work out well enough.

Another is to cap the increase, say allow the addition of at most one new level to the list.

In any case, we do want some sort of exponential distribution:

```
int randomLevel() {

    int Level;

    for (Level = 0; rand() % 2 == 0; Level++);

    return Level;

}
```

It should be intuitively clear that the "ideal" skip list would give essentially the same search performance as a binary search; i.e., the average number of comparisons would be $\Theta(\log N)$.

But what about the "practical" skip list? Clearly the irregular pattern will not necessarily be terribly similar to the "ideal" distribution.

In the paper cited on slide 2, Pugh proves that, with high probability, the average number of comparisons would be $\Theta(\log N)$ for the "practical" skip list as well.

That is, there is no guarantee that a skip list will always (or ever) provide highly efficient search, but in all likelihood a skip list will provide search costs that are at least roughly competitive with balanced binary tree structures.

Why consider using a skip list? Primarily because the implementation is far simpler than that of a good balanced binary tree.

Given a data value (that may or may not occur in the list):

```
Node* x := &HeadNode;
for i := MaximumLevel downto 0 do
    ; go as far as possible on current level
    while x->forward[i]->keyField < searchKey do
      x := x->forward[i]
    endwhile
    ; drop to previous level (via for loop)
endfor
x := x->forward[0]    ; step to next Node
if x->keyField = searchKey then
    return x->Data
else
    return failure
endif
```

The search logic here assumes some relational operators have been supplied for the key type and some convention for returning an indication of failure.

Adapted from Pugh, 1990

Logically, deleting a node should be the opposite of insertion. The same basic principles apply:

- first we must find the node that precedes the node to be deleted, if any, remembering the nodes containing "pass" pointers

- if the succeeding node contains the targeted key value

    - update the "pass" pointers that point to the target node

    - delete the targeted node

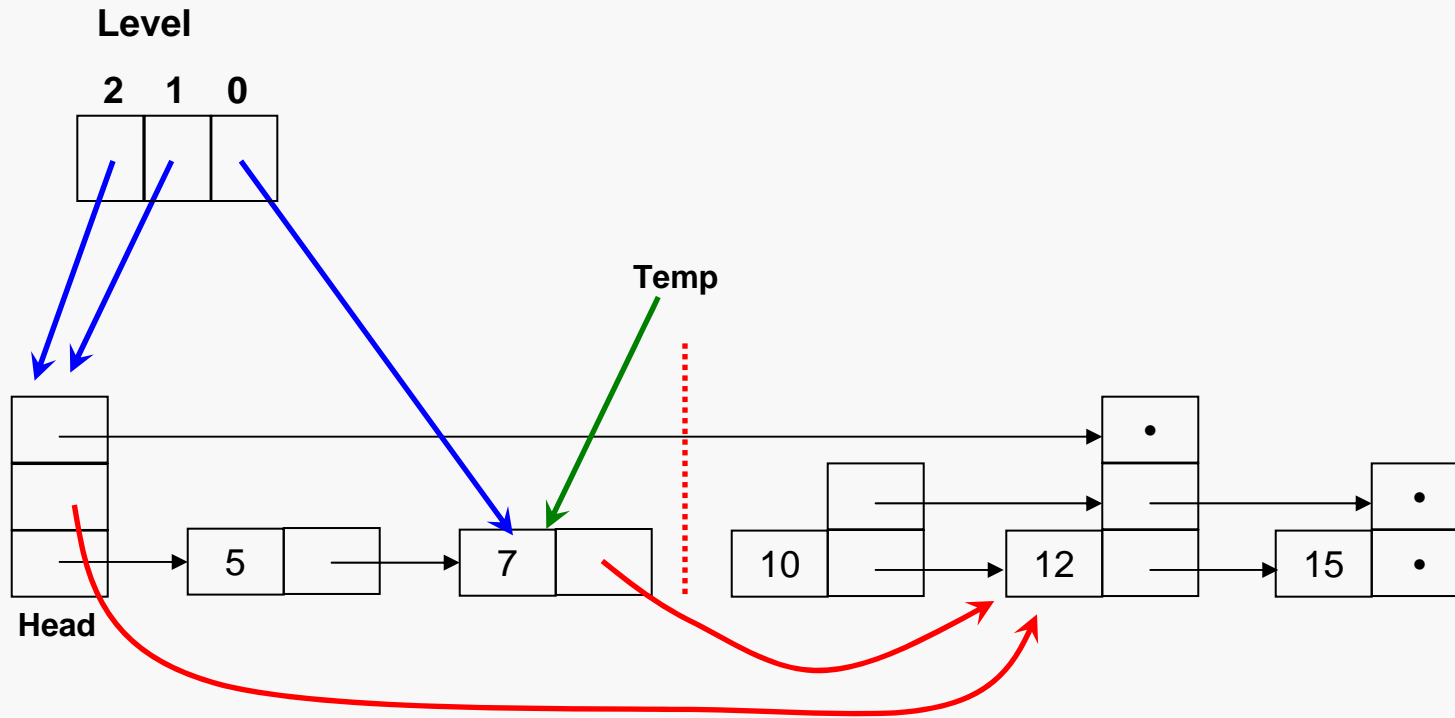   - if necessary, adjust the head node to reduce the number of levels in the list

Consider deleting the value 10 from the list below:

Level

2 1 0

Temp

5   7   10   12   15

Head
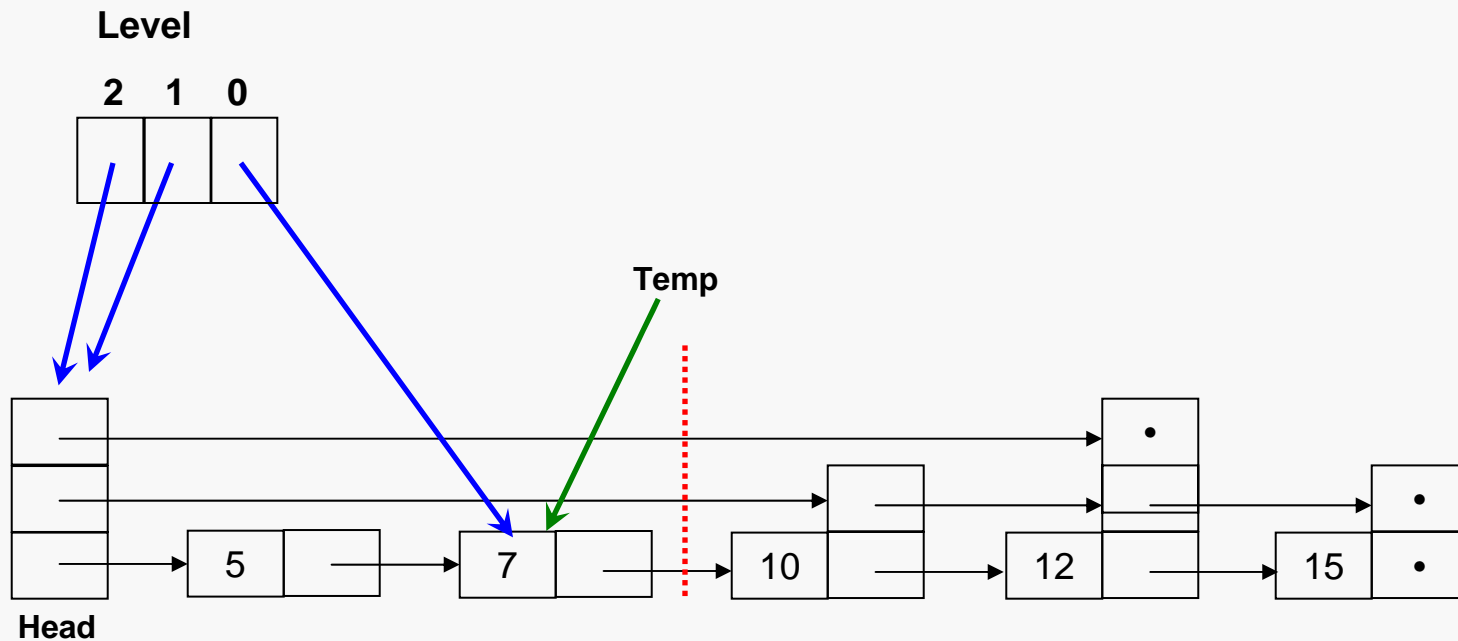
First we must find the preceding node and identify the nodes whose pointers may need to be reset.

Next we must reset the pointers to the targeted node "around" it:

Level

2 1 0

Temp

| 5 | | | 7 | | | 10 | | | 12 | | | 15 | • |

Head

Finally, we must deallocate the targeted node.

Insertion and deletion both begin with a search, which will probably have cost $\Theta(\log N)$.

Following the search, the number of pointers that must be updated is no more than twice the number of levels in the list, which should not be much more than log(N) if level assignments to new nodes have been done intelligently.

The remaining work (reducing the head structure, etc.) are essentially constant cost…

…aside from the node deletion, which involves a system call. However, deletions are generally less expensive than allocations.

So, it is reasonable to expect that the cost of insertion and deletion will be $\Theta(\log N)$.