List arrays and Skiplists

CMSC 420

Central question

- How can I do binary search in a sorted linked list?

Central question

- How can I do binary search in a sorted linked list?
- Naive solution has complexity $\mathcal{O}(n)$, which does not improve on linear search☹

Central question

- How can I do binary search in a sorted linked list?
- Naive solution has complexity $\mathcal{O}(n)$, which does not improve on linear search☹
  - When you have n nodes, you scan $\frac{n}{2}$ nodes to find the middle element.
  - When you have $\frac{n}{2}$, you scan $\frac{n}{4}$….
  - When you have $\frac{n}{4}$, you scan $\frac{n}{8}$….
  - …
  - When you have $1 = \frac{n}{n} = \frac{n}{2^{\log_2 n}}$, you scan $1 = \frac{n}{\log_2 n}$ nodes….

Central question

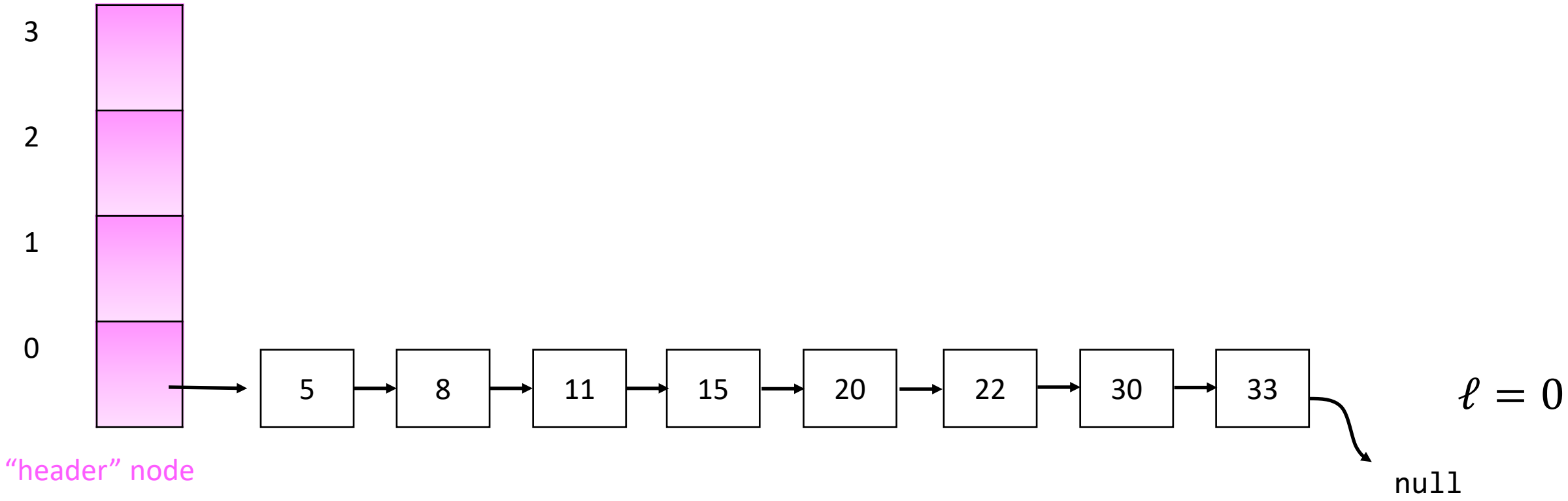
- How can I do binary search in a sorted linked list?
- Naive solution has complexity $\mathcal{O}(n)$, which does not improve on linear search☹
  - When you have n nodes, you scan $\frac{n}{2}$ nodes to find the middle element.
  - When you have $\frac{n}{2}$, you scan $\frac{n}{4}$….
  - When you have $\frac{n}{4}$, you scan $\frac{n}{8}$….
  - …
  - When you have $\frac{n}{2^{\log_2 n}}$, you scan $\frac{n}{n} = 1$ nodes….

Total #nodes scanned = $\frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{\log_2 n}} = n\left(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{\log_2 n}}\right) = \mathcal{O}(n)$

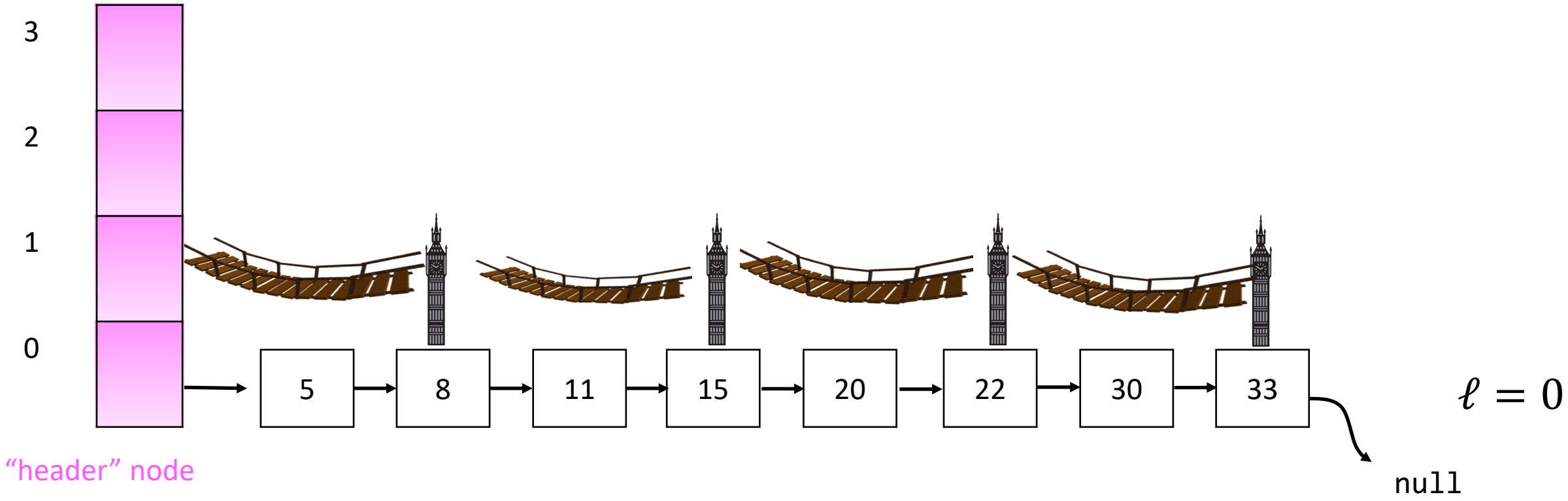
Central question

- Idea to improve complexity: Consider every node to be a "tower" consisting of a certain number of levels, different for each node.
  - The levels themselves will be connected by linked lists
    - Think of them as bridges!
  - The "tallest" tower will have height $\log_2 n$!
  - First level will be the original list: connects every $2^0 = 1$ nodes.
  - Second level will connect every $2^1 = 2$ nodes.
  - ….
  - i^{th} level will connect every 2^i nodes.

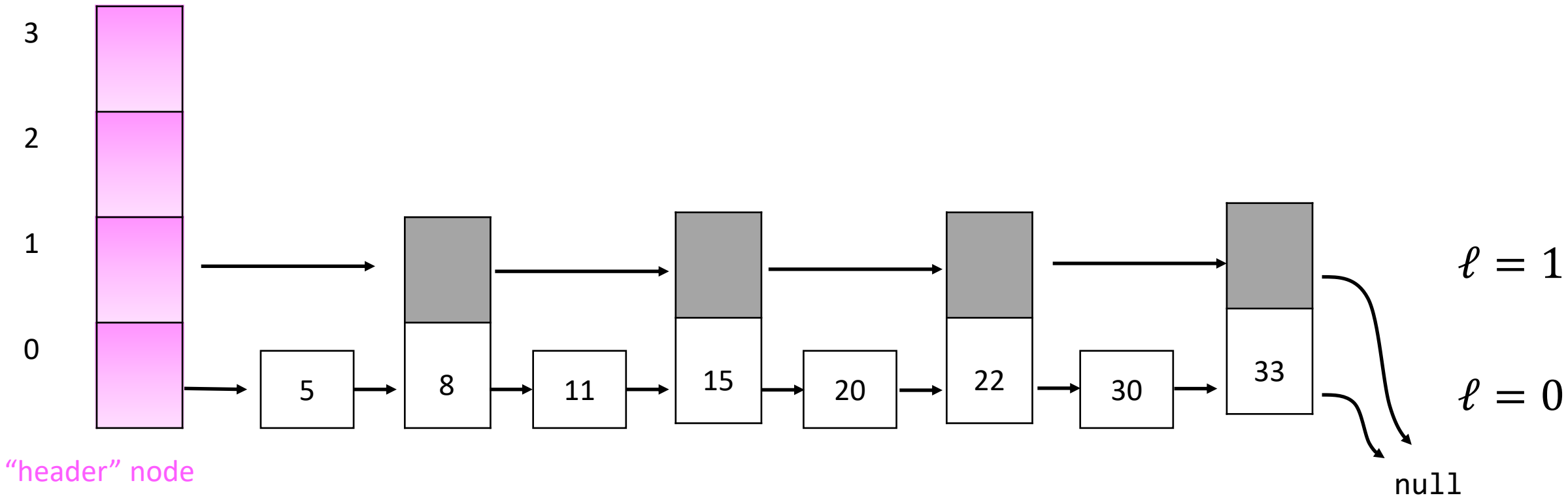
List Arrays



List Arrays

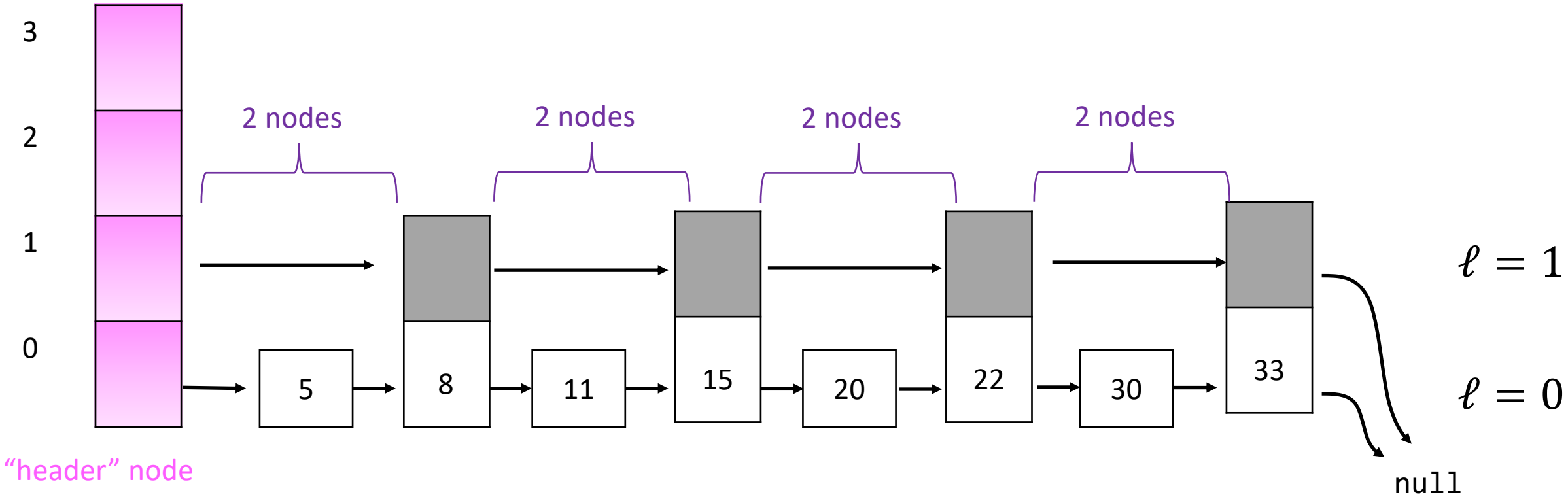


List Arrays

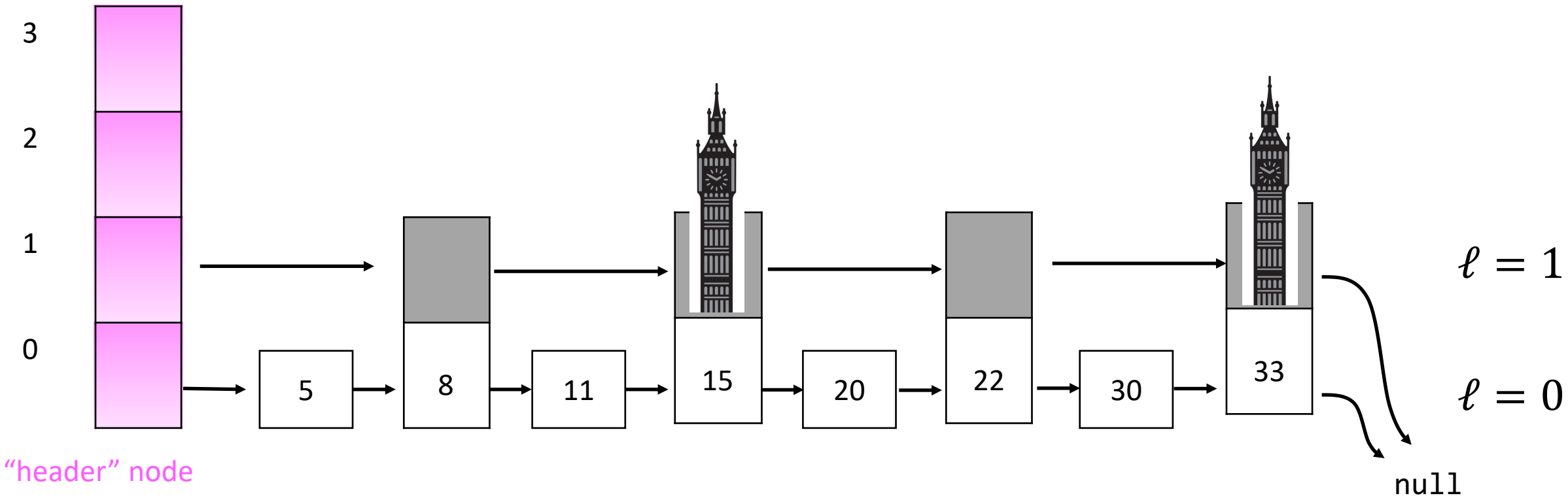


List Arrays

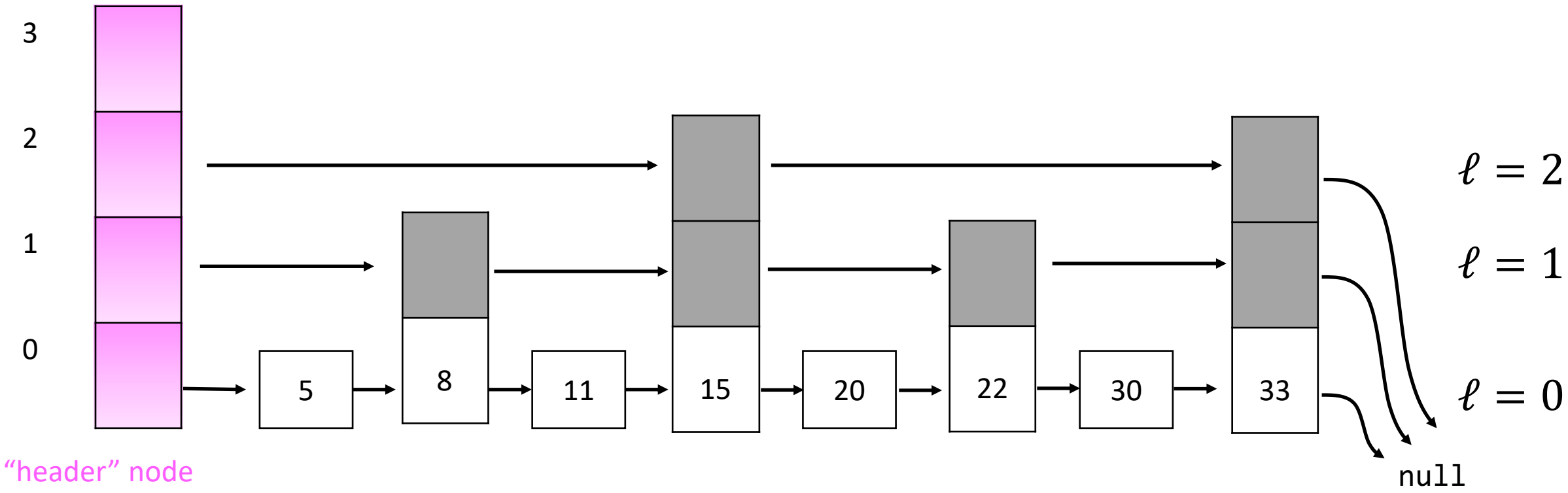
Second level ($\ell = 1$) connects every 2^1 = 2 nodes!



List Arrays

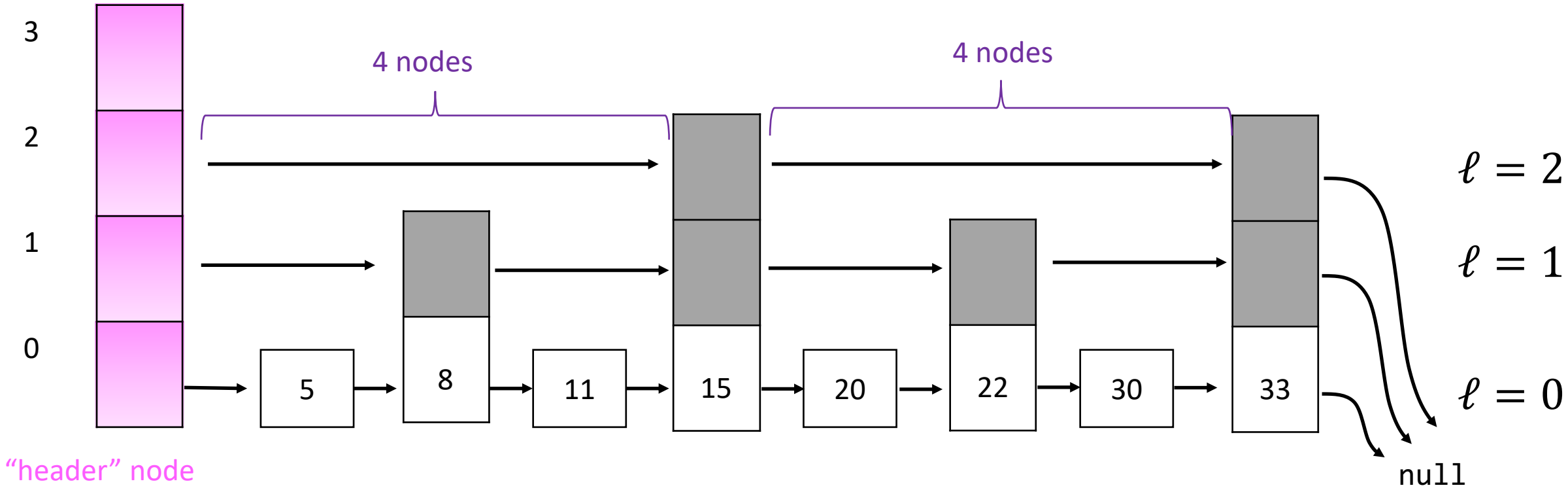


List Arrays

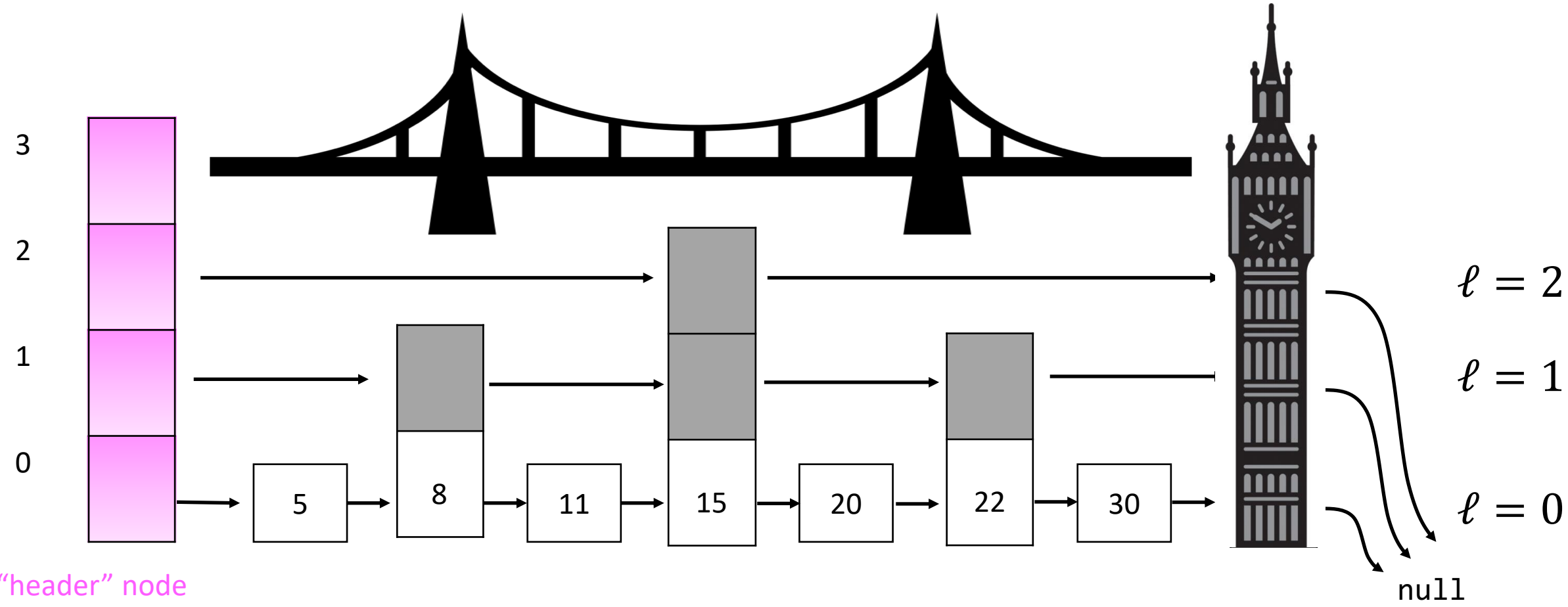


List Arrays

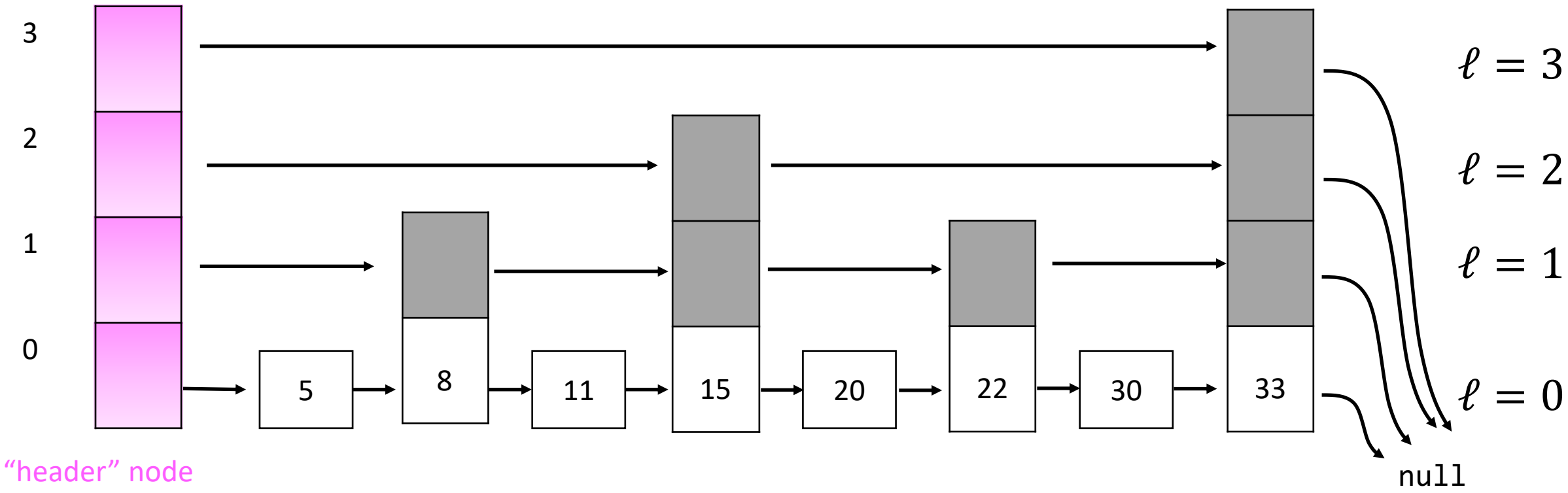
Third level ($\ell = 2$) connects every 2^2 = 4 nodes!



List Arrays

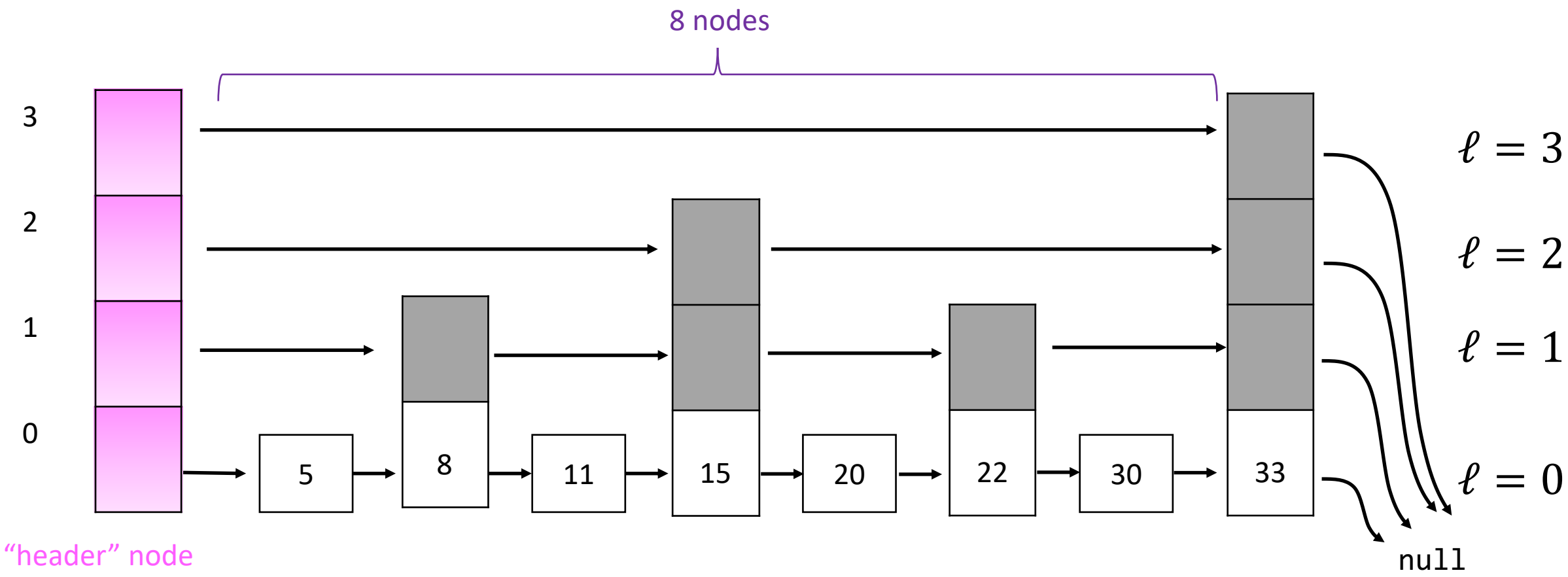


List Arrays



List Arrays

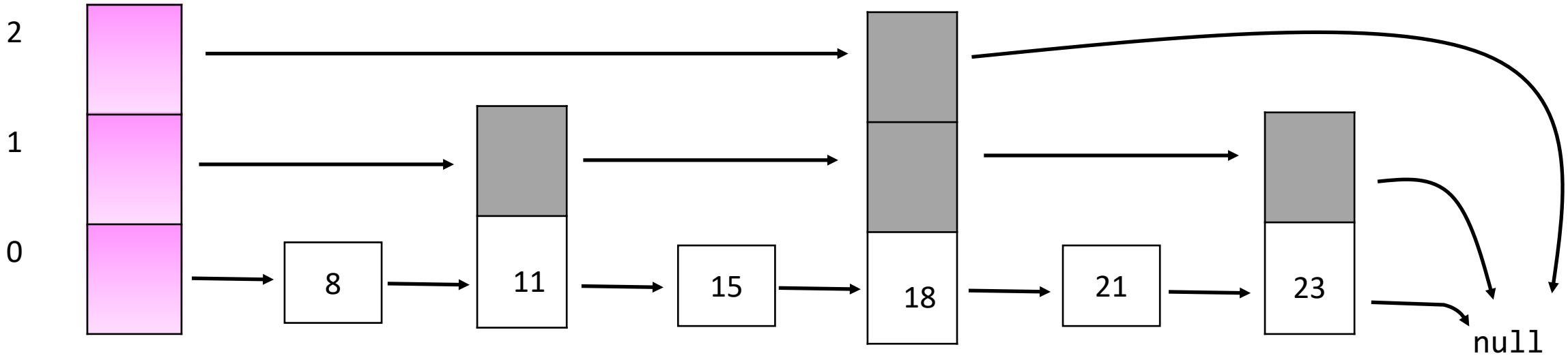
Fourth level ($\ell = 3$) connects every 2^3 = 8 nodes!



Extension to other lengths

- What if my list has a length that is not a power of 2?
- No problem: we then have $\lceil \log_2 n \rceil$ levels. Example:

$$(n = 6) \Rightarrow (\lceil \log_2 n \rceil = 3)$$

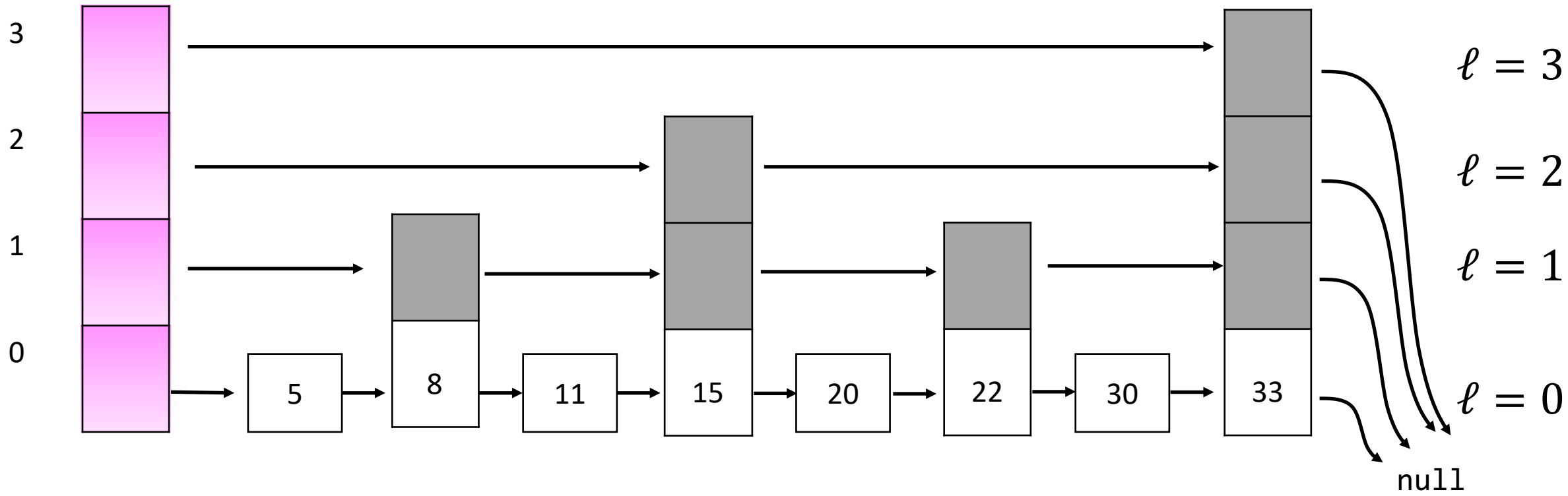


Searching a list array

- Every one of the individual arrays has the data element stored at position 0.
- Searching works as follows:
  1. Begin at `header[levels - 1]`. Traverse the current level until:
     a) You find the element (search **hit**, congrats)
     b) You find a tower whose element is greater, or:
     c) You hit `null`.
  2. If either (b) or (c) happens, descend to the level below and goto (1).
     - If there is no level below, you have a search **miss**. At this point you should consider how this makes you feel.

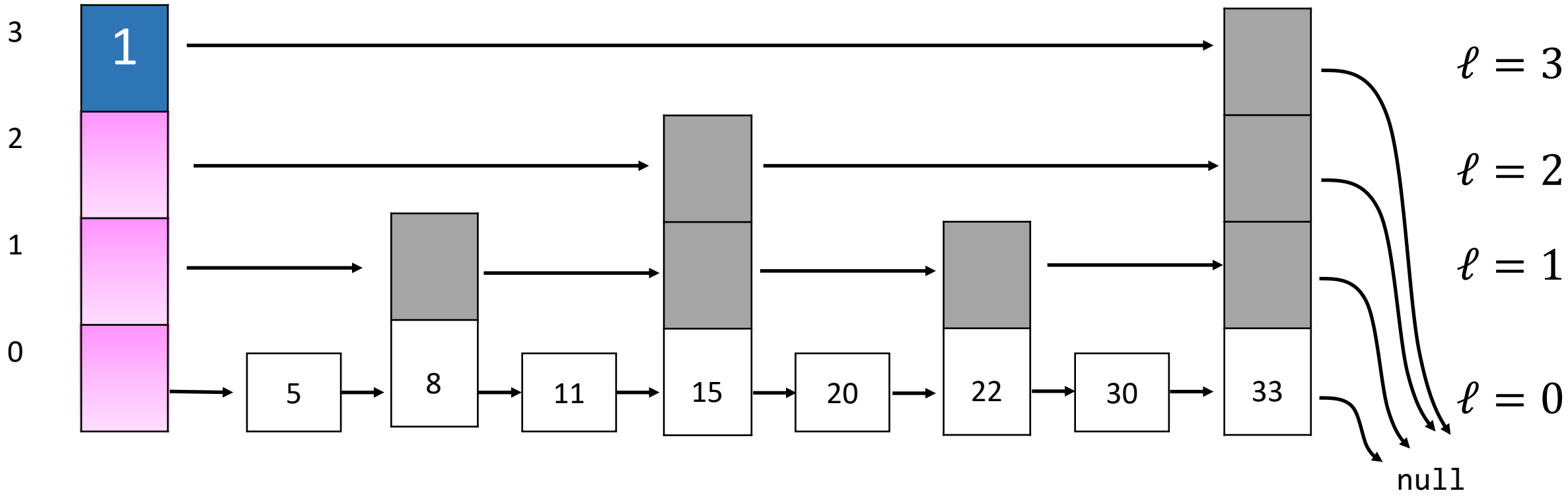
Search example

Search for “20” – should be a successful search



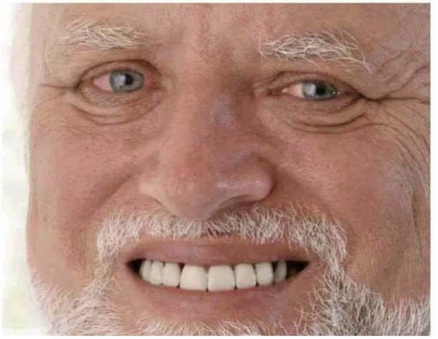
Search example

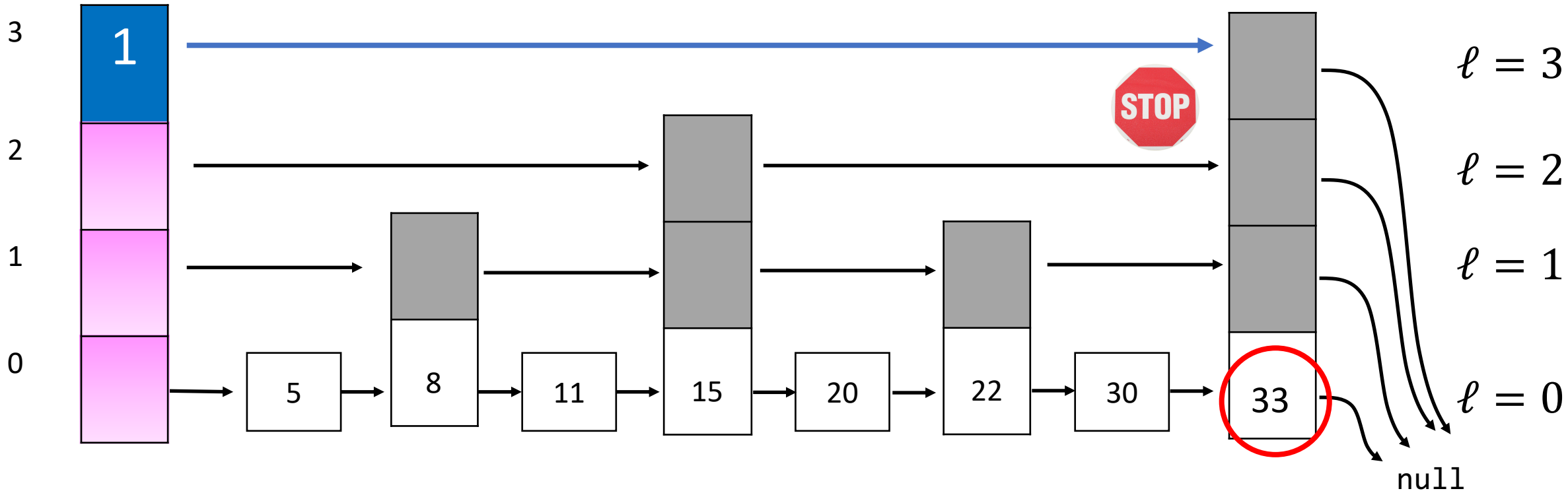
Start at header[levels-1]…



Search example

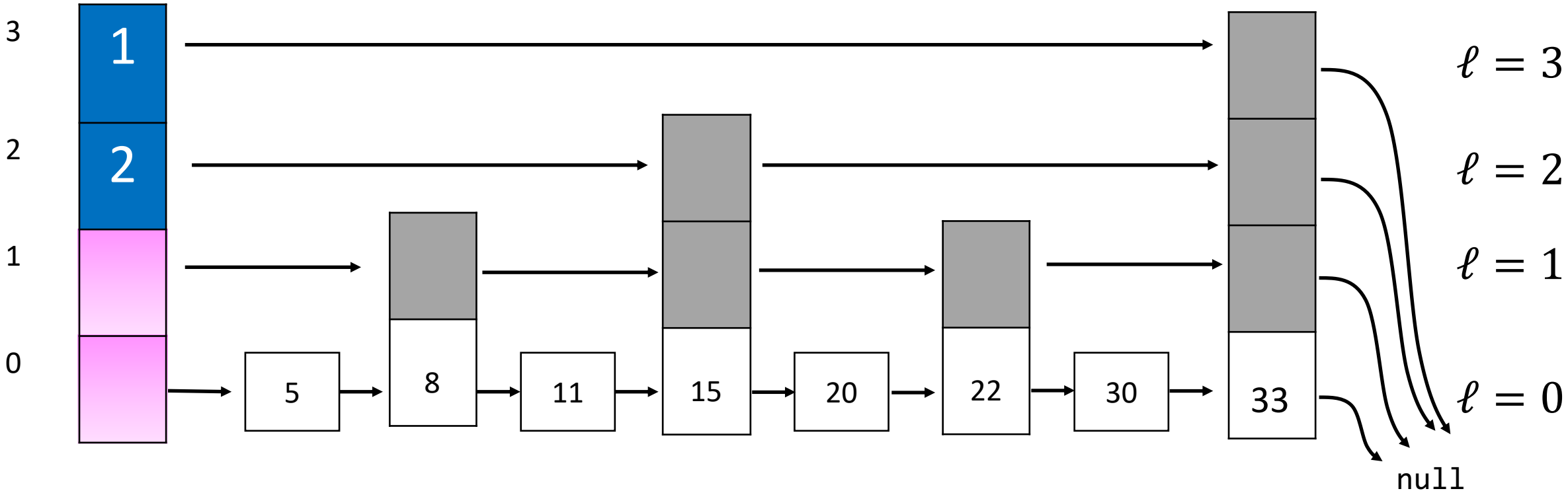
The pointer points to a tower with element greater than 20





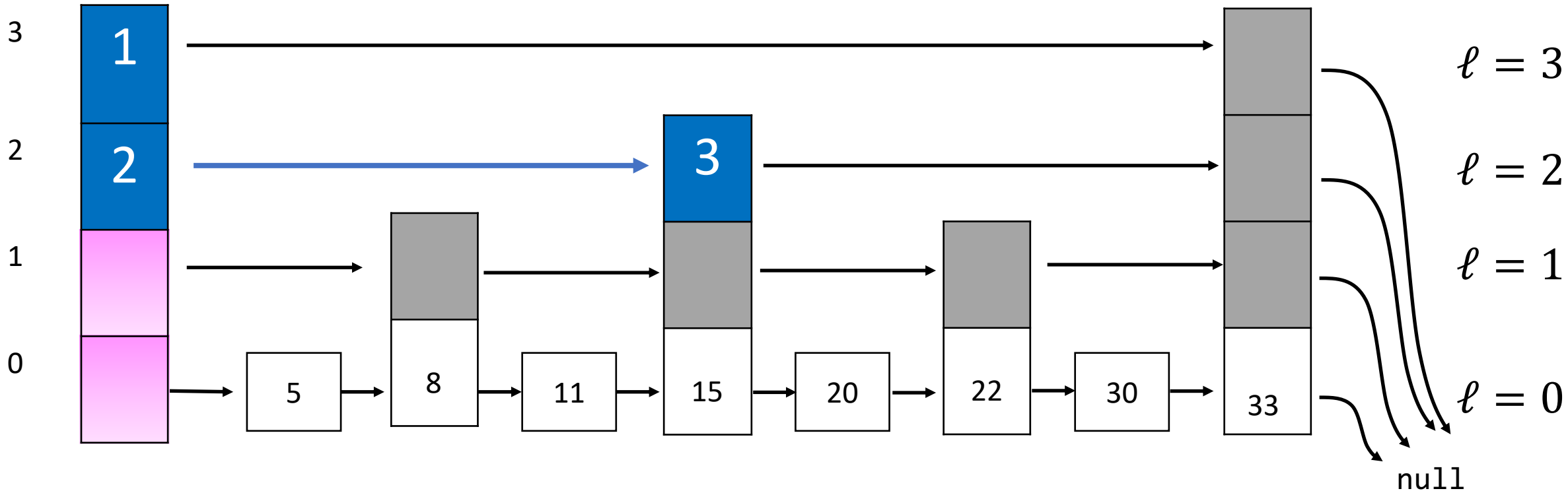
Search example

Solution: decrease your level!



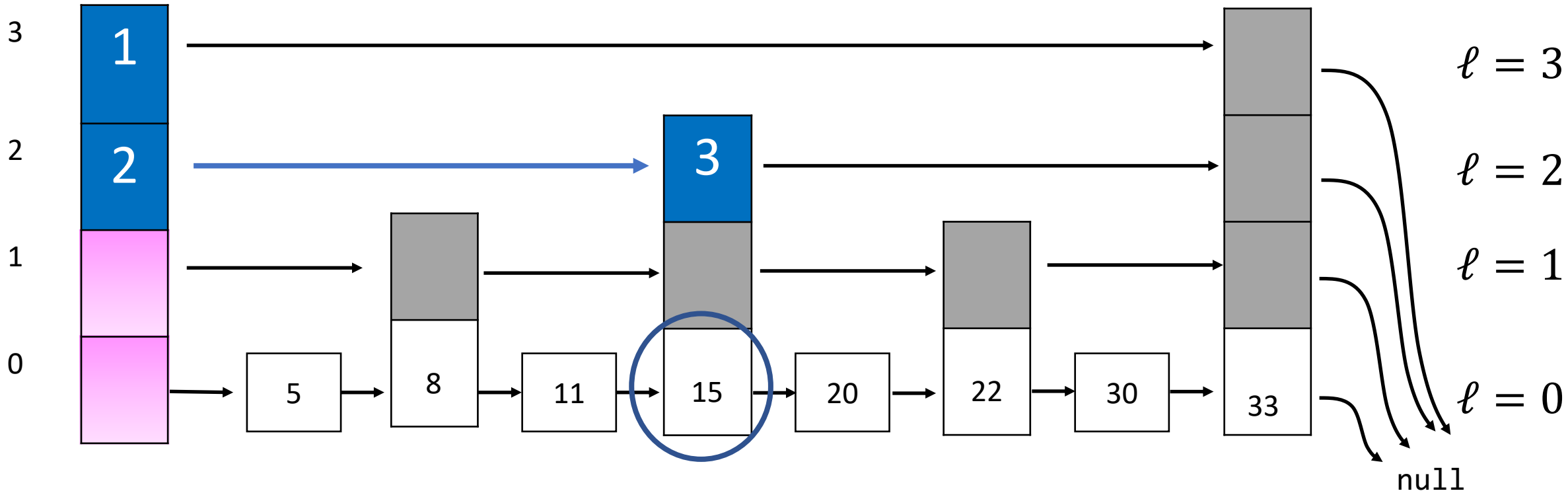
Search example

Dereference pointer.



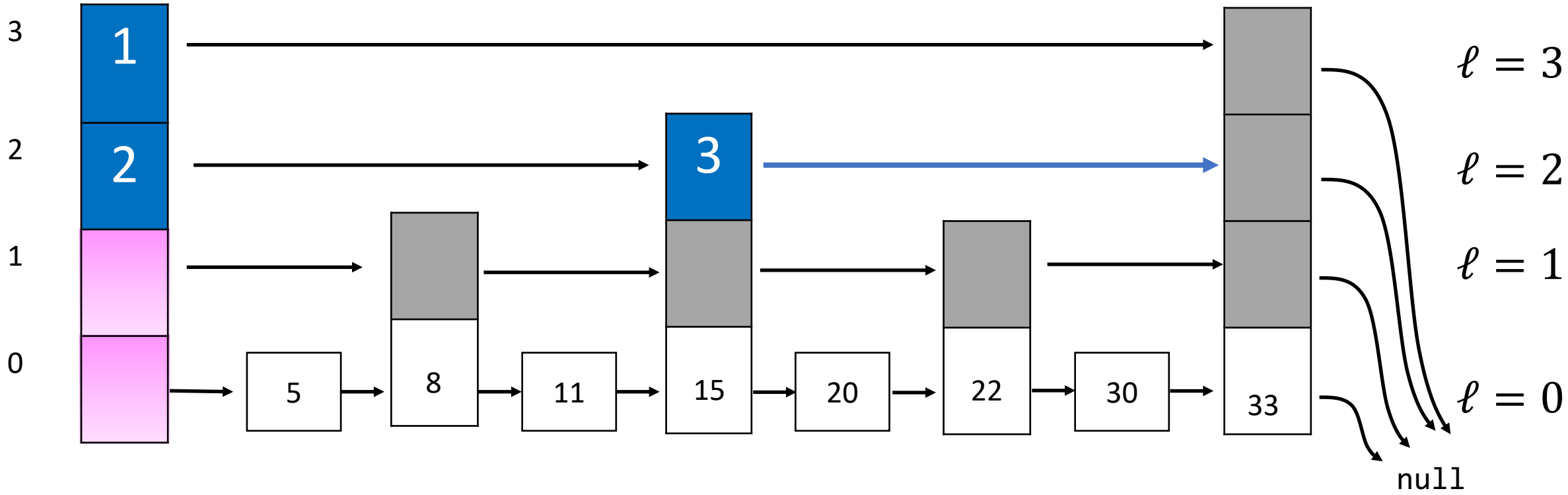
Search example

*Key (15) **smaller** than desired key (20)*



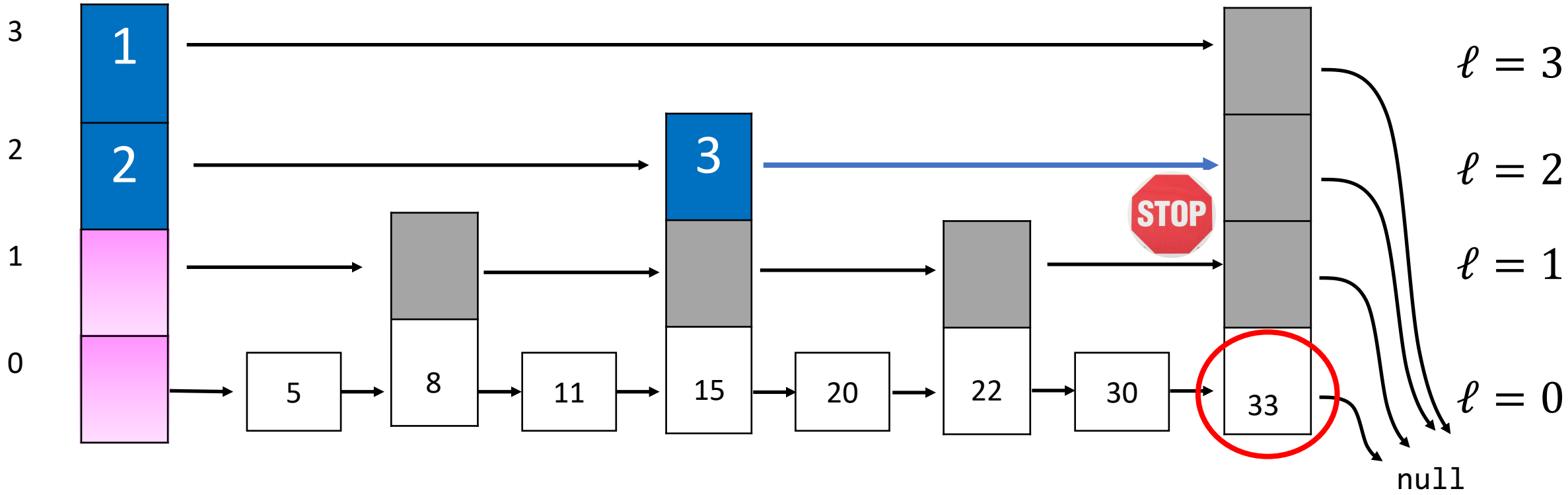
Search example

So keep going on the same level!



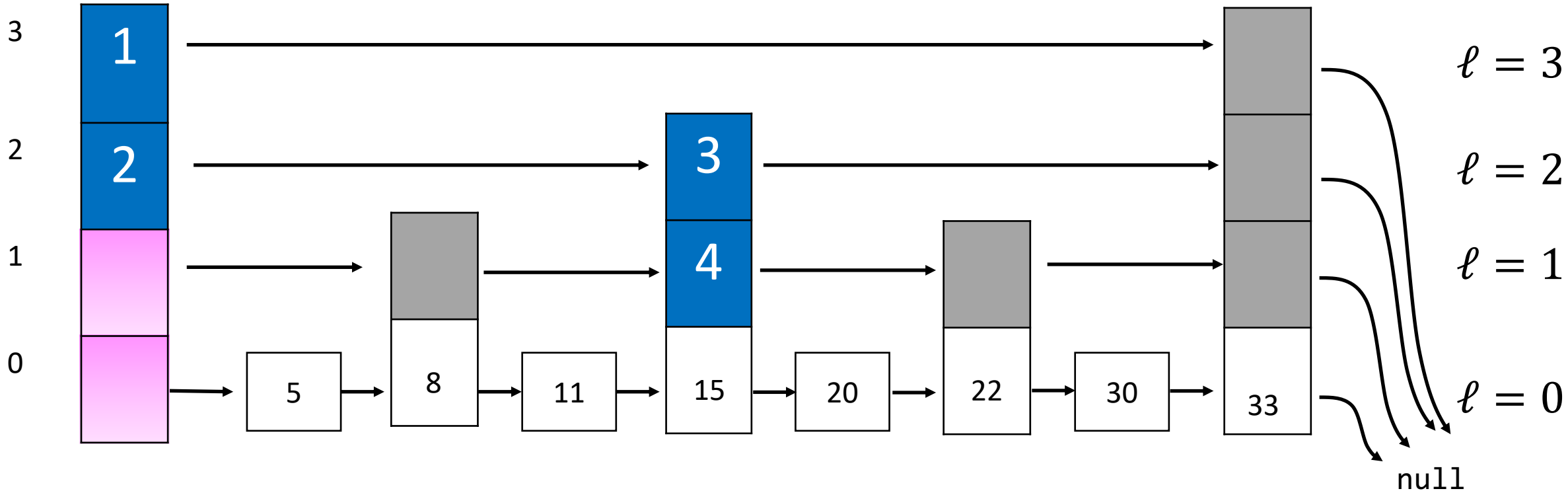
Search example

Contained key (33) **greater** than sought key (20) again!



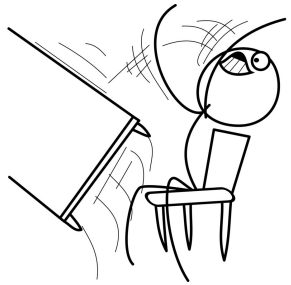
Search example

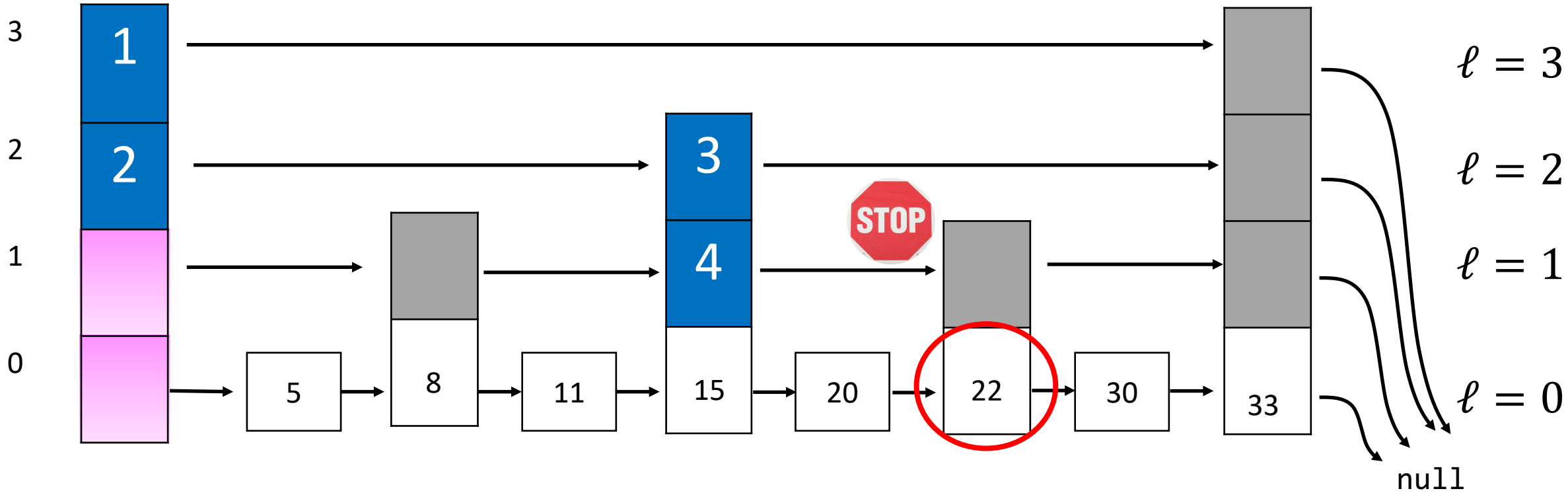
So descend a level again!



Search example

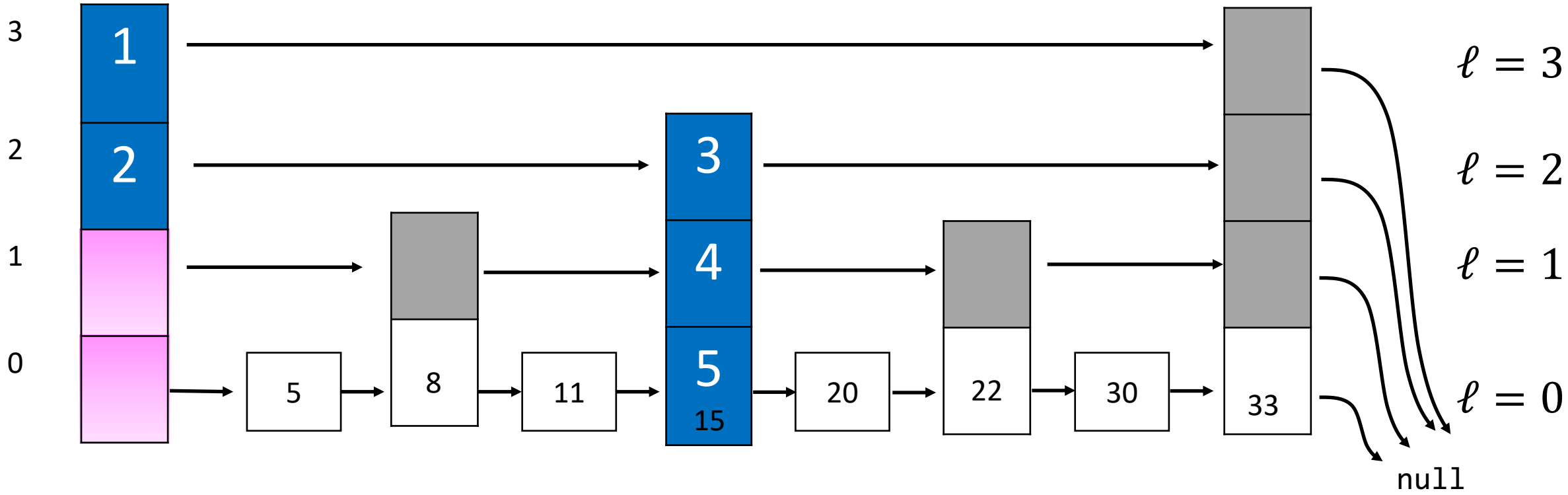
Dereferencing pointer leads us to a tower with a key too big…





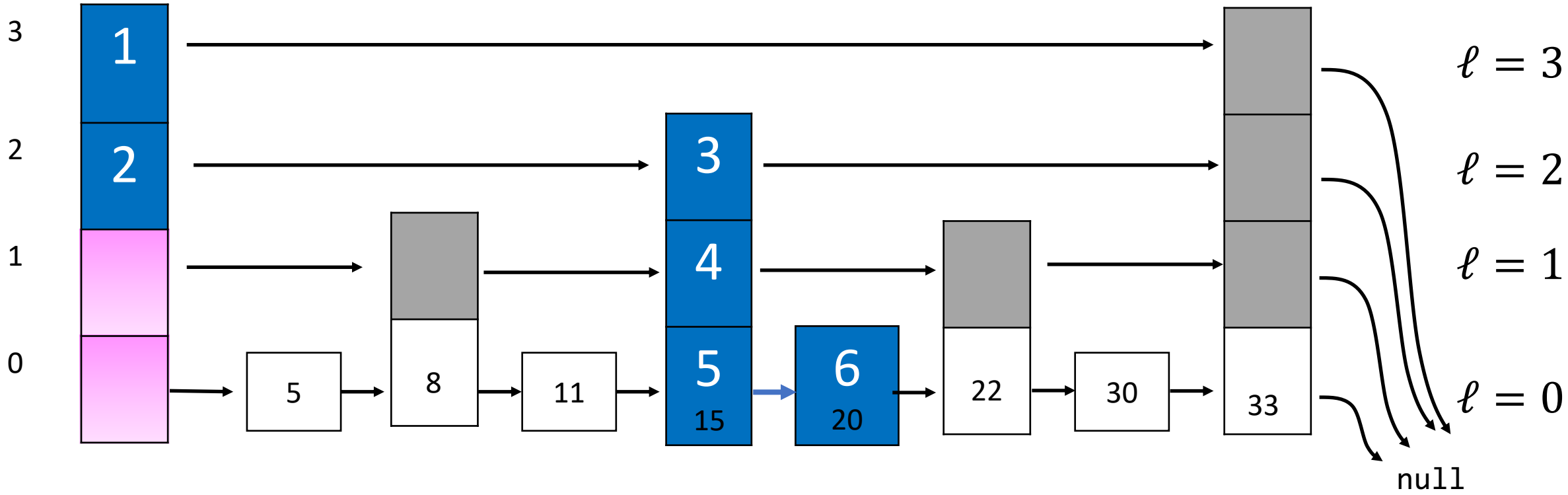
Search example

So descend another level…



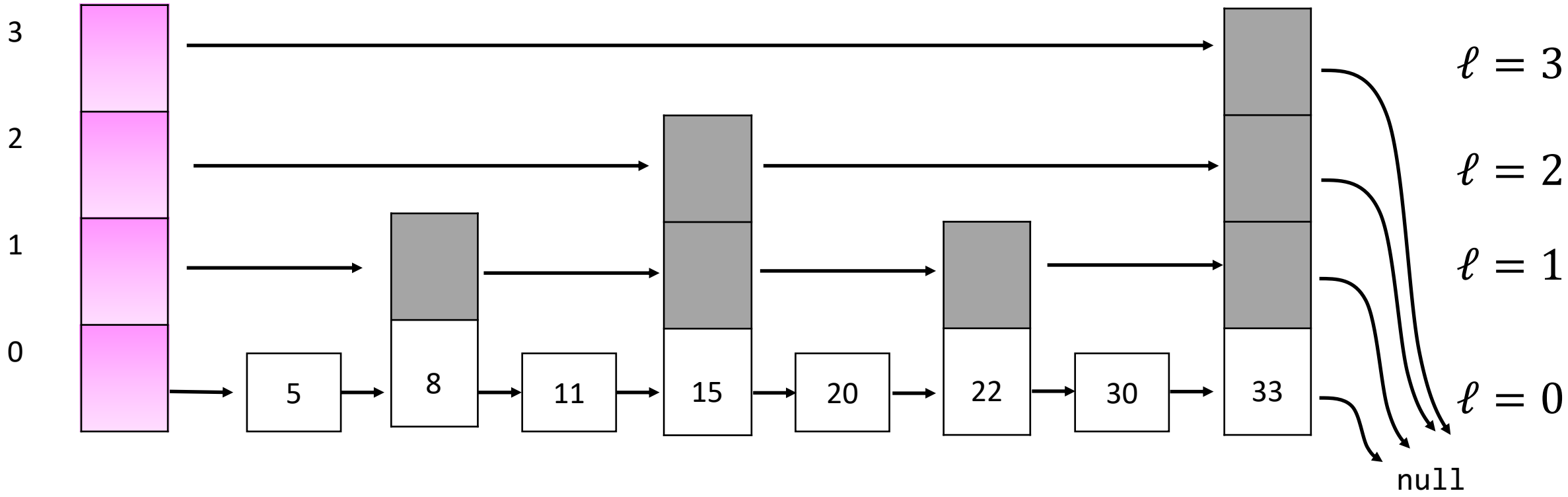
Search example

Final pointer dereference gives us our key. Search **hit.**

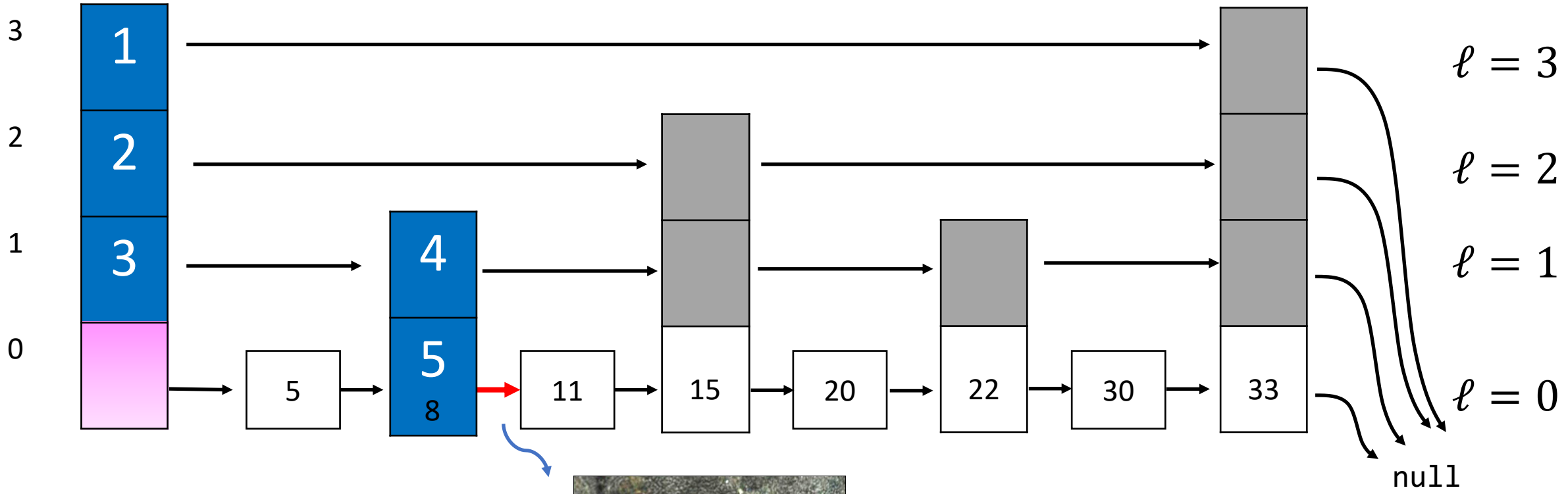


Another search example – for you!

- For the same list array, sequentially mark the nodes traversed by searching for 9 (nine), which should fail!



Solution



Space analysis

- To make matters simpler, assume that n is some power of 2.
  - So, $n/2$ will be an integer, as will be $n/4 = n/2^2$, $n/8 = n/2^3$, etc
- Alongside the list level, we have $\log_2 n + 1$ levels.
- The list level (0^{th} level) has n nodes.
- The 1st level has $n/2$ nodes.
- The 2nd level has $n/4$ nodes.
- …
- …
- Level # $\log_2 n$ has 1 nodes

Total # nodes:

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 = n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots \frac{1}{n}\right) =$$

$$= 2^{-(\log_2 n - 1)}$$

$$= n(2^0 + 2^{-1} + 2^{-2} + \cdots + 2^{-\log_2 n})$$

Sum of geometric progression with $a_0 = 2^0 = 1$, $r = 1/2$ and $\log_2 n + 1$ terms!

Unpacking the sum…

- $2^0 + 2^{-1} + 2^{-2} + \cdots + 2^{-(\log_2 n)} = \frac{2^0\left[1-(1/2)^{\log_2 n+1}\right]}{1-1/2} = 2\left[1 - (1/2)^{\log_2 n+1}\right]$
- $(1/2)^{\log_2 n+1} = \frac{1}{2} \cdot \frac{1}{2^{\log_2 n}} = \frac{1}{2n}$
- Substituting the green stuff onto the red stuff yields:
- $2^0 + 2^{-1} + 2^{-2} + \cdots + 2^{-(\log_2 n)} = 2\left(1 - \frac{1}{2n}\right) = \frac{2n-1}{n}$
- Note that this tends to 2 as n goes to infinity, which is no surprise since the geometric series $\sum_{i=0}^{n} 2^{-i}$ converges to $\frac{1}{1-r} = \frac{1}{1-1/2} = 2$

Final answer

- Num nodes = $n\left(2^0 + 2^{-1} + 2^{-2} + \cdots + 2^{-\log_2 n}\right) =$

$$\cancel{n}\frac{2n-1}{\cancel{n}} = 2n - 1$$

- We also have the "header" node which contributes $\log_2 n + 1$ nodes…

- Final answer: $2n - 1 + \log_2 n + 1 = 2n + \log_2 n = \mathcal{O}(n)$

Search efficiency analysis

- We have a sorted linked list ℓ with n elements and we build a list array A on top.
- We want to search for an element e.
- Unit cost (what n counts): dereferencing a pointer.
- What is the computational complexity of searching for e in ℓ using A?

Search efficiency analysis

- We have a sorted linked list ℓ with n elements and we build a list array A on top.
- We want to search for an element e.
- Unit cost (what n counts): dereferencing a pointer.
- What is the computational complexity of searching for e in ℓ using A?

$O(n)$

$O(\log_2 n)$

$O(\log_2 (\log_2 n))$

Something else (what?)

Search efficiency analysis

- We have a sorted linked list ℓ with n elements and we build a list array A on top.
- We want to search for an element e.
- Unit cost (what n counts): dereferencing a pointer.
- What is the computational complexity of searching for e in ℓ using A?

$O(n)$

$O(\log_2 n)$

$O(\log_2 (\log_2 n))$

Something else (what?)

Now let's determine the constant…

Search efficiency analysis

- We have a sorted linked list ℓ with n elements and we build a list array A on top.
- We want to search for an element e.
- Unit cost (what n counts): dereferencing a pointer.
- What is the maximum number of links we will scan to succeed or fail the search for e in ℓ using A?

$\log_2 n$

$1.5 \log_2 n$

$2 \log_2 n$

Something else (what?)

Search efficiency analysis

- We have a sorted linked list ℓ with n elements and we build a list array A on top.
- We want to search for an element e.
- Unit cost (what n counts): dereferencing a pointer.
- What is the maximum number of links we will scan to succeed or fail the search for e in ℓ using A?

$2\log_2 n + 1$

Also acceptable

$\log_2 n$

$1.5\log_2 n$

$2\log_2 n$

Something else (what?)

At every level, we visit at most 2 nodes (why?) and at level 0 we might have one additional pointer dereference.

Huge issue with list arrays

- Inserting a new element or deleting from it requires re-building the entire list array ☹
- Pay $O(n)$ cost to do so.
- For the insertion and/or deletion of n elements, the cost ends up being quadratic, which is unacceptable.
- Conclusion: a list array is **not** a good dictionary because of sub-par insertion and deletion.
  - But for ***immutable*** sorted lists, they work pretty well!

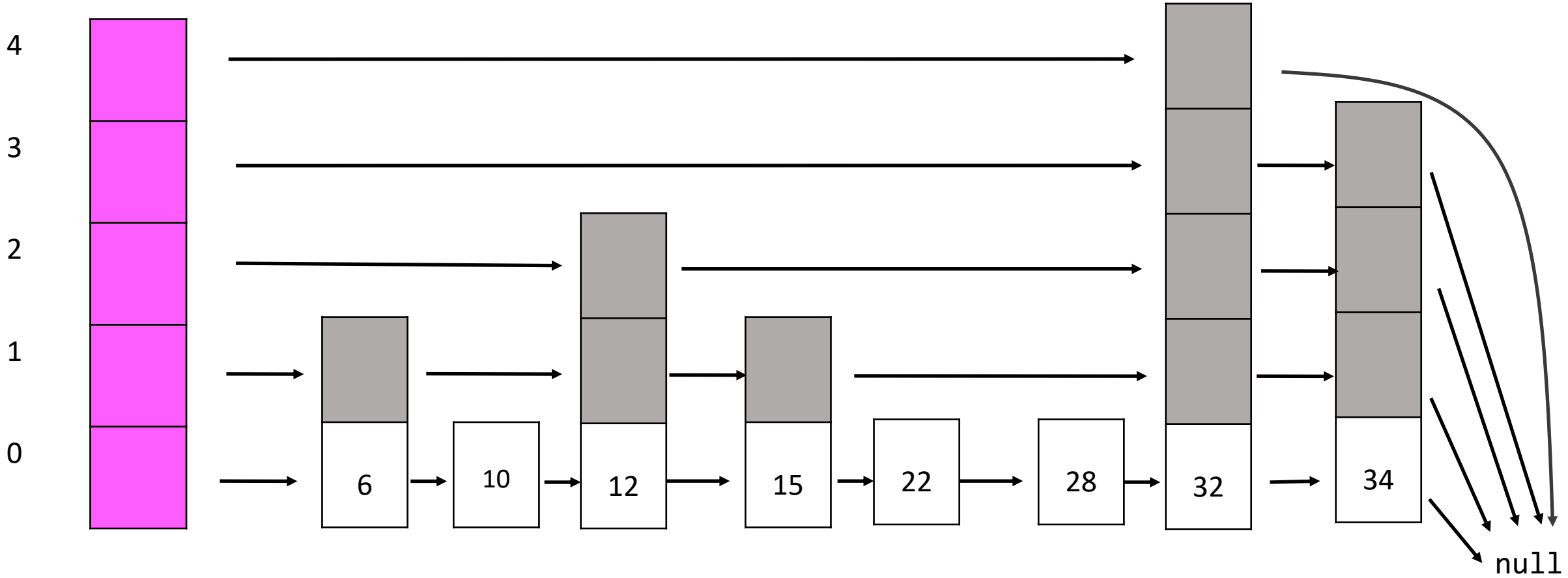
Binary Search “vs” list arrays

Binary Search	List array
Used in contiguous storage (arrays)	Used in discontiguous storage (linked lists)
Requires up to $\log_2 n$ comparisons	Requires up to $2 \log_2 n + 1$ comparisons
Doesn’t need extra space	Needs $O(n)$ extra space
Doesn’t require adjustments after mutations	Needs to be re-built after every mutation

Skiplist: relaxing the constraint

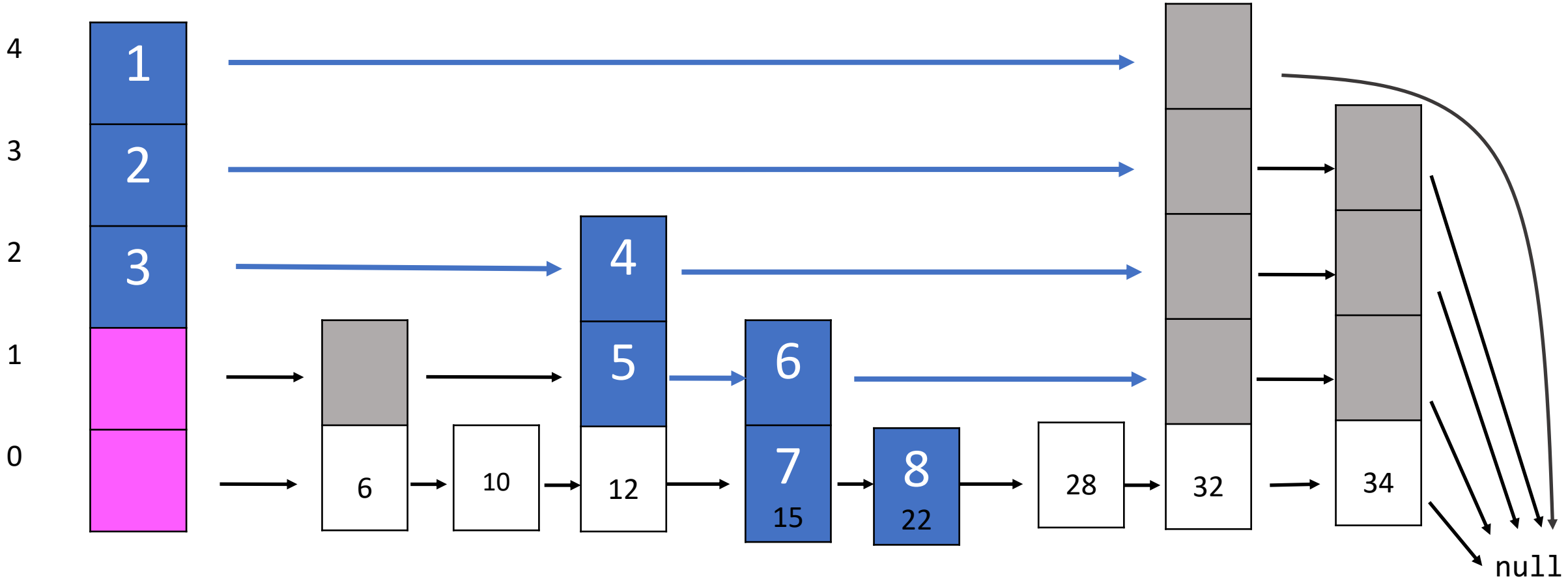
- Observation: in a list array, half ($^1/_2$) of the "towers" that extend to level i also extend to level $i + 1$. Therefore:
  - $^1/_2$ of the list's keys are represented in level 1
  - $^1/_4$ are represented in level 2
  - ….
  - $(^1/_2)^i$ are represented in level i.
- Solution: probabilistically build the towers by randomly sampling the new height of an inserted node before actually inserting it!
- Use a geometric distribution: $P(level = i) = \left(\frac{1}{2}\right)^{i+1}$

Skiplist example



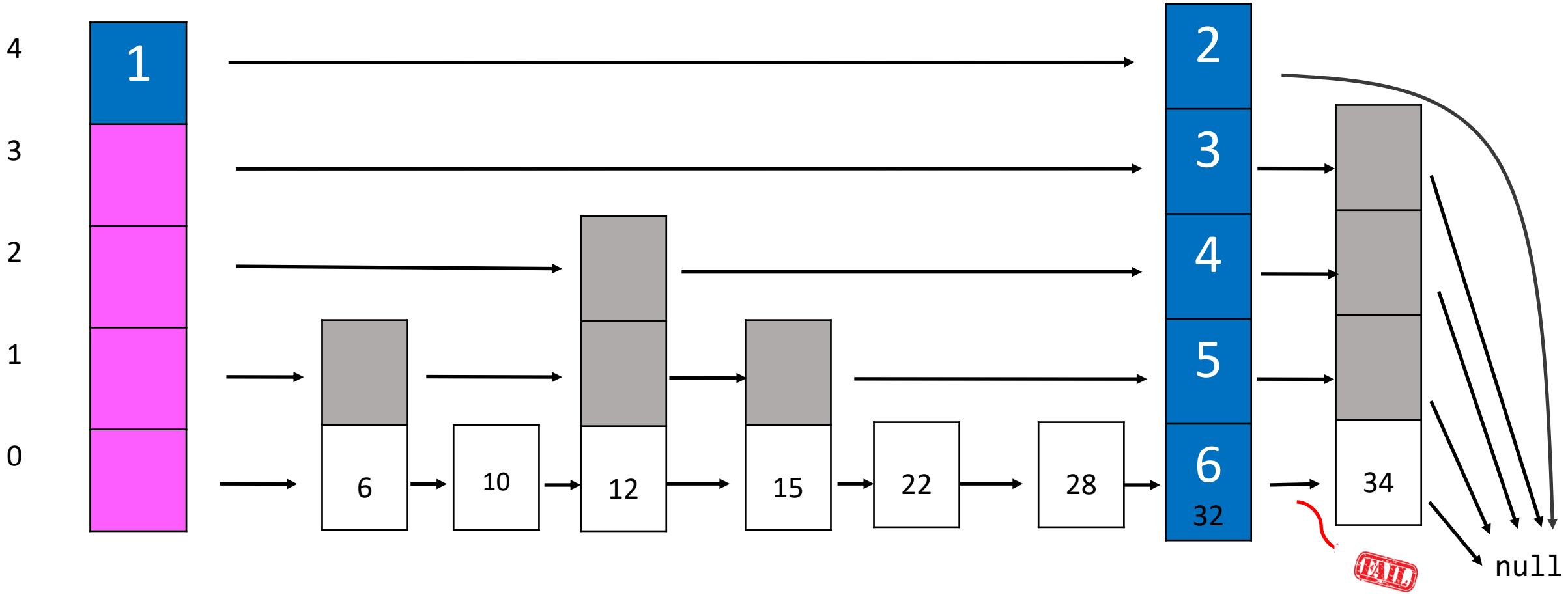
Skiplist example

Searching for 22 (should be a search ***hit***)



Skiplist example

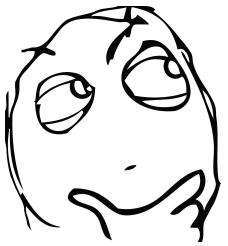
Searching for 33 (should be a search ***miss***)

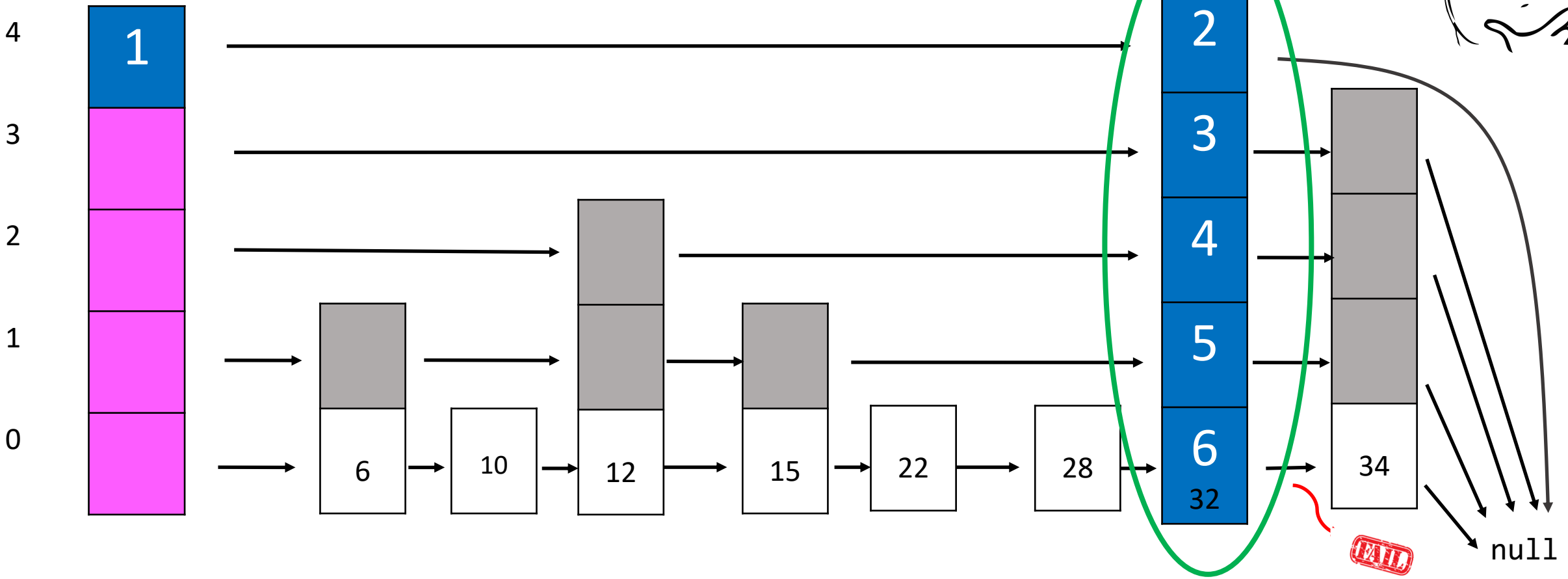


Skiplist example

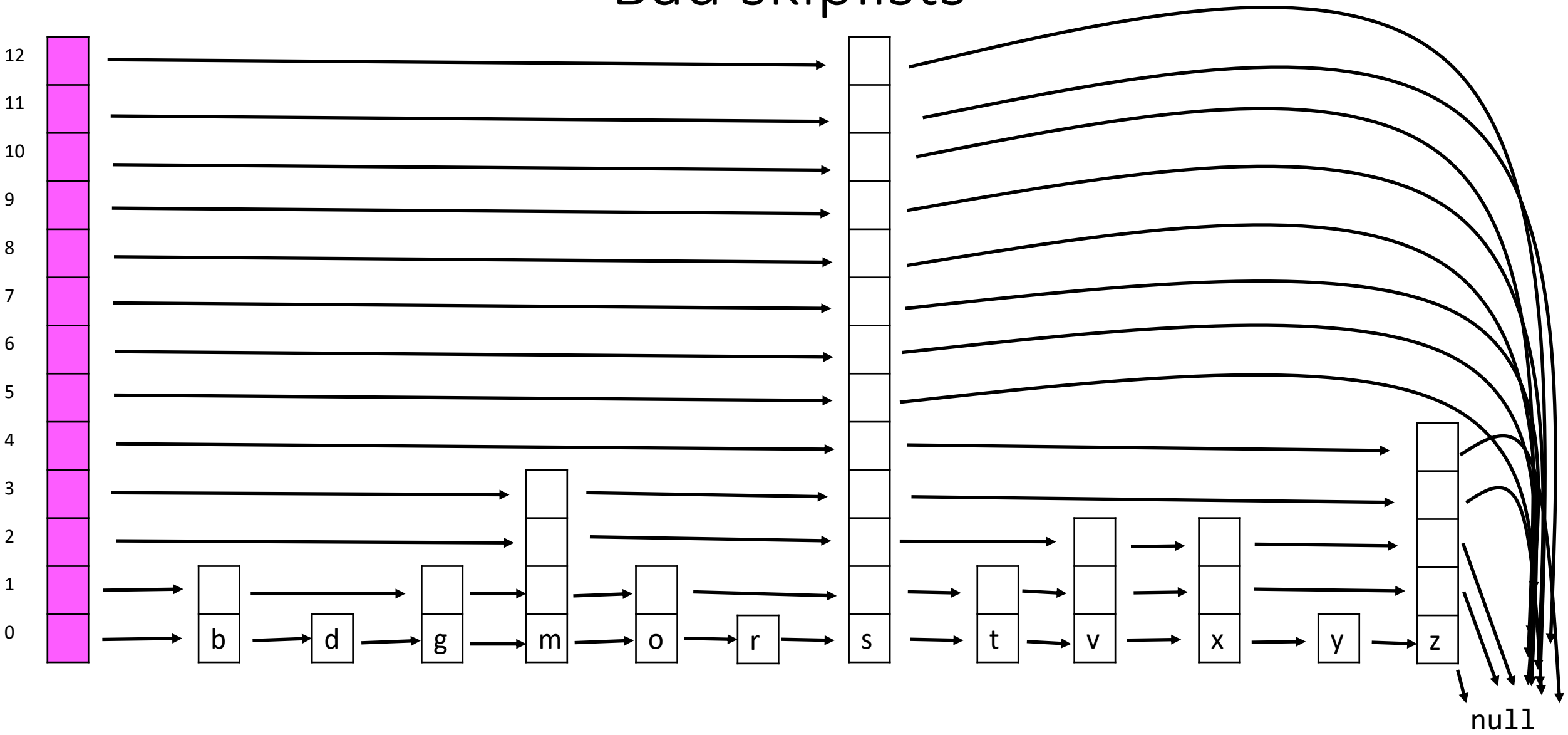
Searching for 33 (should be a search ***miss***)

It certainly looks like the height of this tower is slowing us down…

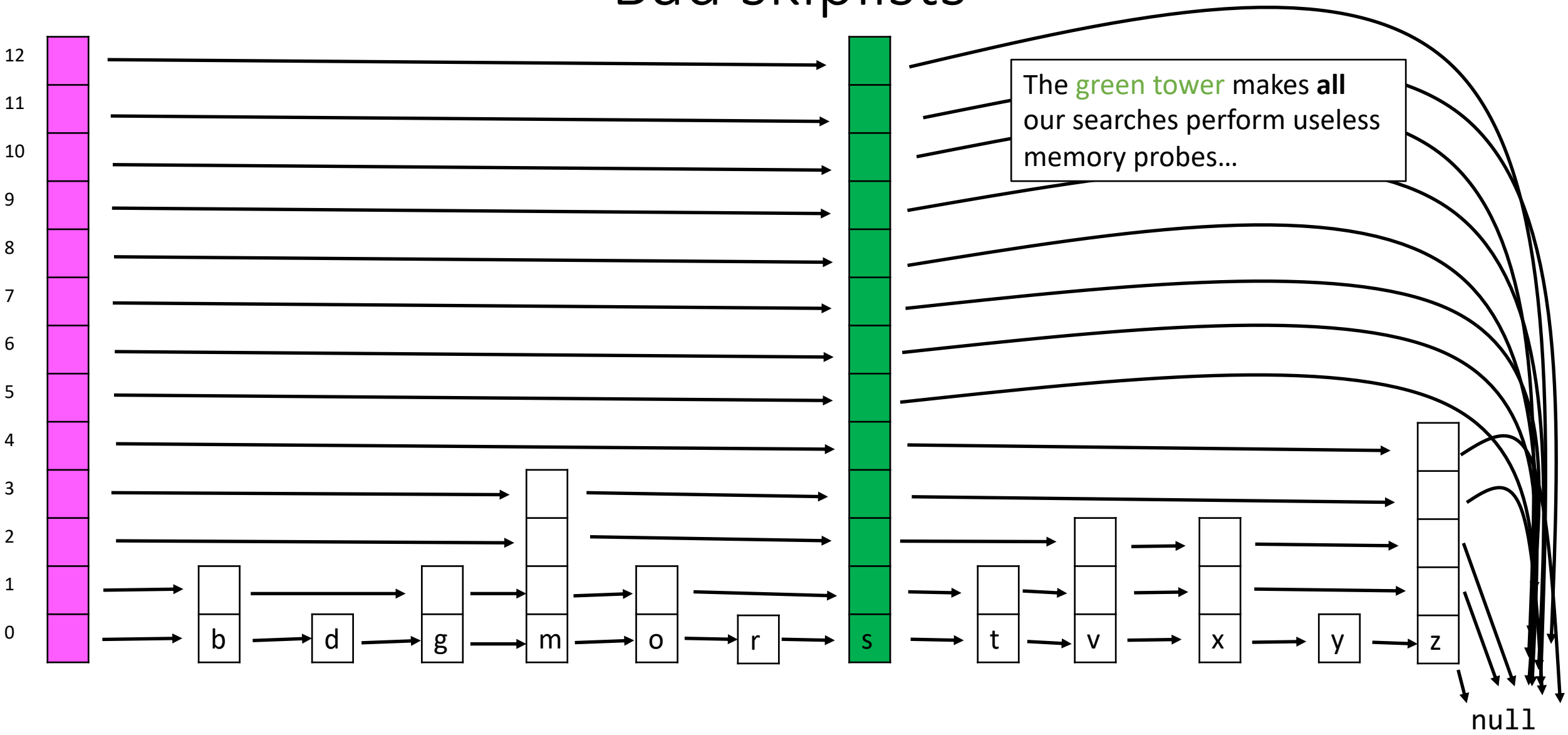




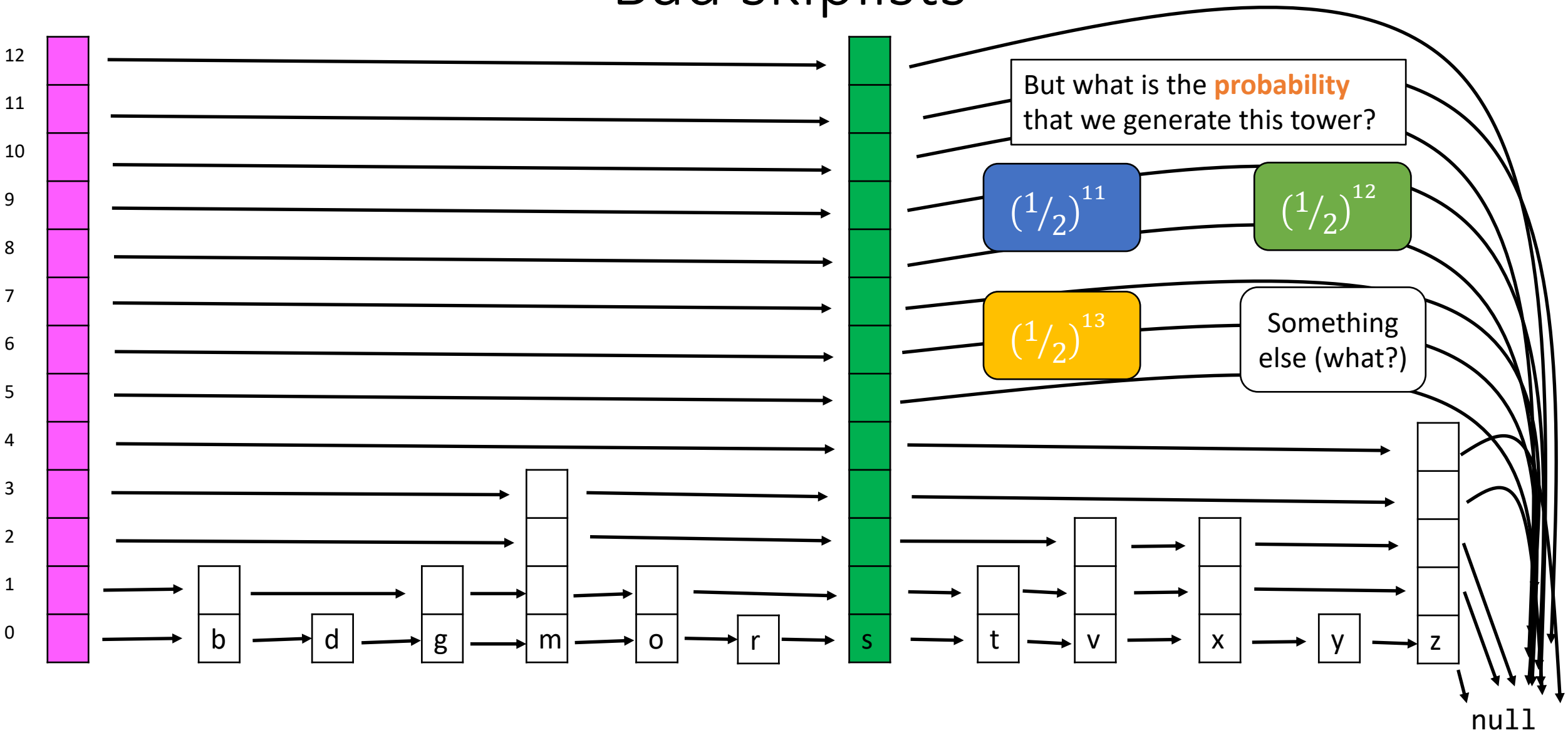
Bad skiplists



Bad skiplists



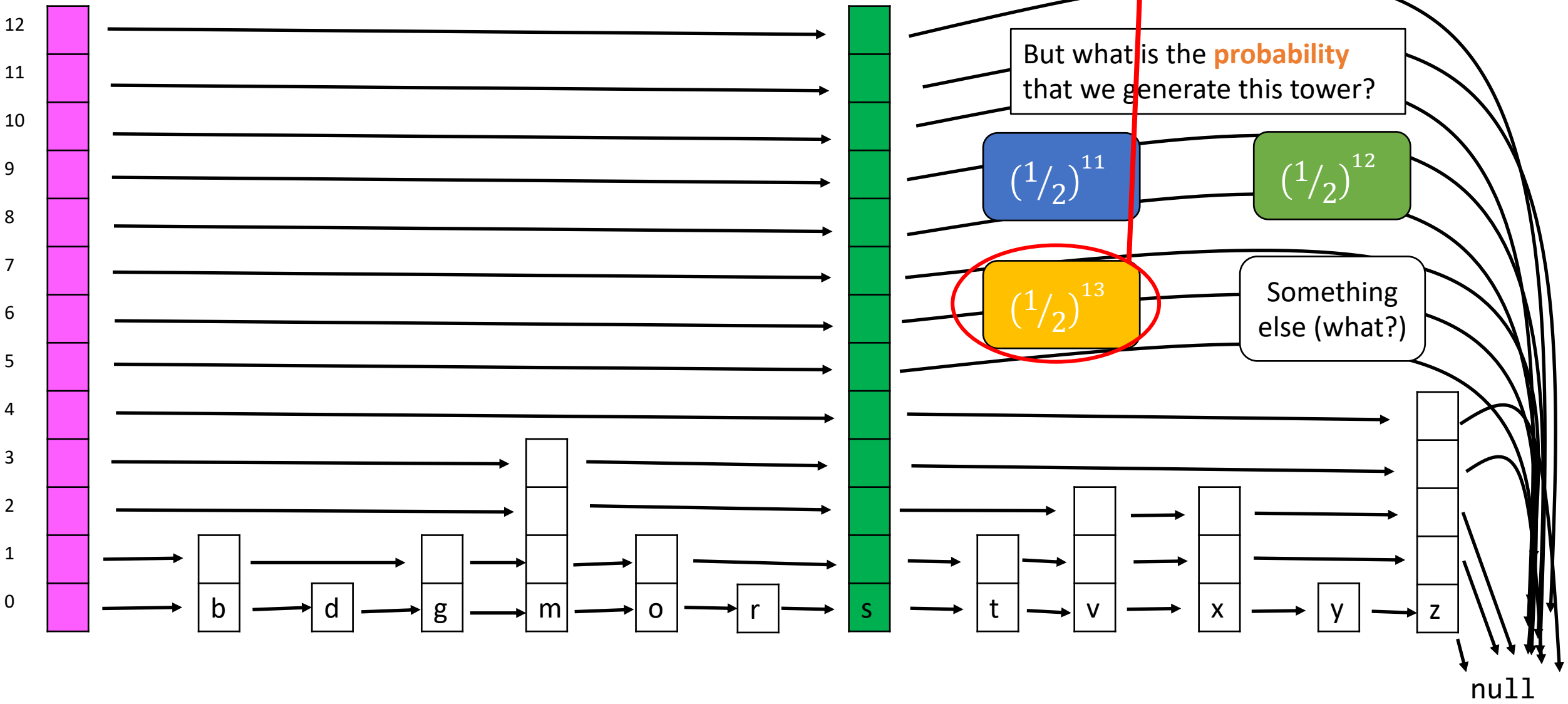
Bad skiplists



We have 12 flips to make the tower have 13 levels and another to keep it there!

Bad skiplists

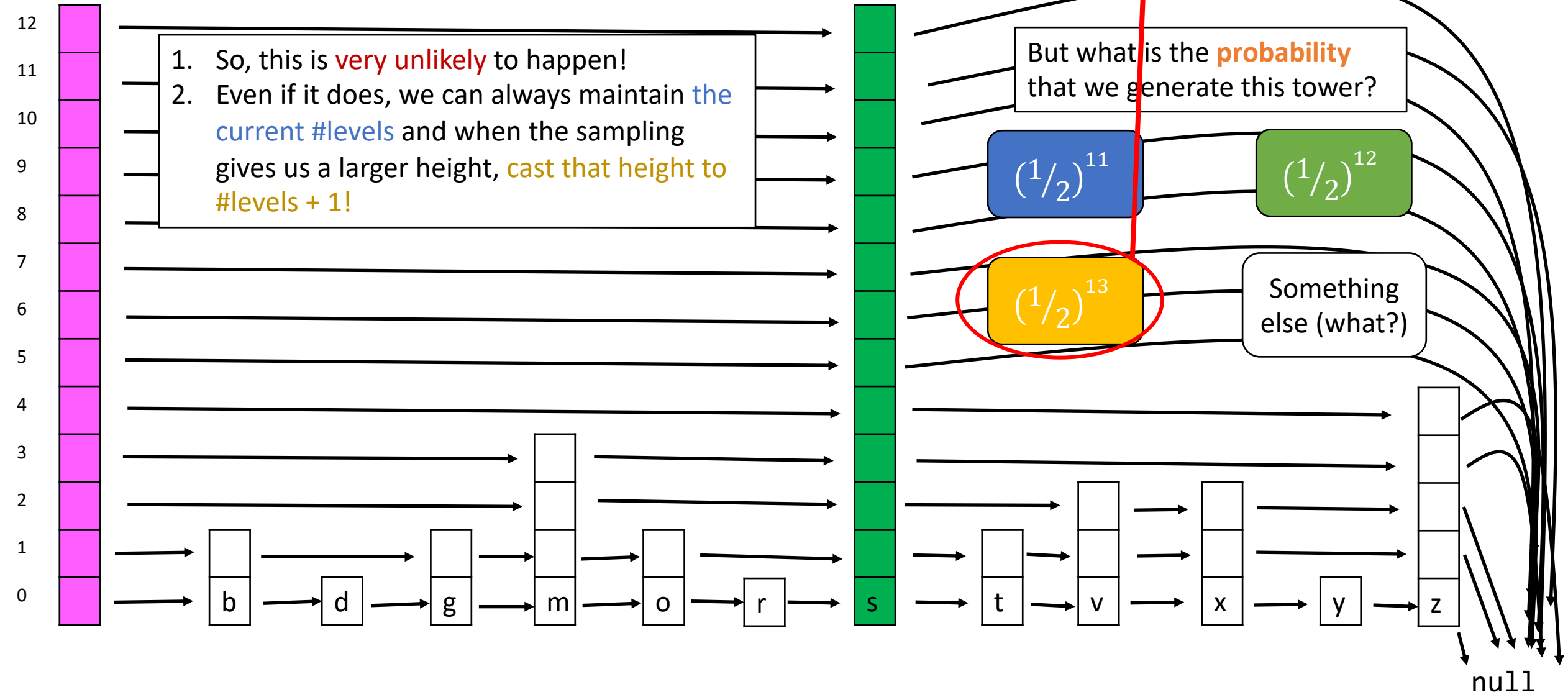
= 0.012207031%



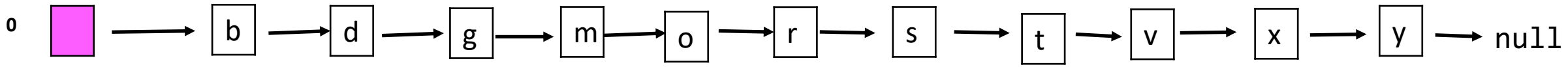
We have 12 flips to make the tower have 13 levels and another to keep it there!

= 0.012207031%

Bad skiplists

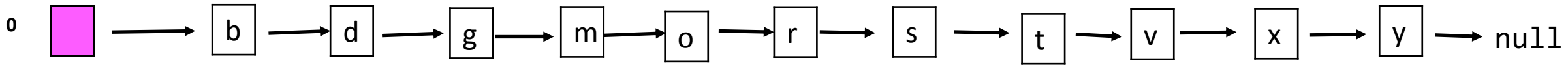


Bad skiplists



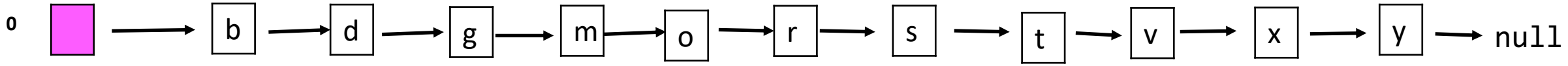
Bad skiplists

- Degenerate skiplist: is equivalent to the linked list of its elements.



Bad skiplists

- Degenerate skiplist: is equivalent to the linked list of its elements.
- But how likely is such a skiplist?



Bad skiplists

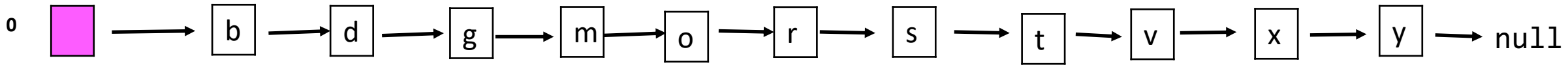
- Degenerate skiplist: is equivalent to the linked list of its elements.
- But how likely is such a skiplist?

$(1/2)^{11}$

$(1/2)^{12}$

$(1/2)^{13}$

Something else (what?)



Bad skiplists

- Degenerate skiplist: is equivalent to the linked list of its elements.
- But how likely is such a skiplist?

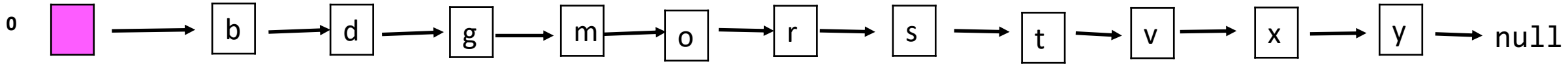
Very unlikely!

$(1/2)^{11}$

$(1/2)^{12}$

$(1/2)^{13}$

Something else (what?)



Insertion

- Insertion into a skiplist follows the “search and splice” rule. Steps:

Insertion

- Insertion into a skiplist follows the “search and splice” rule. Steps:
  1. Randomly sample a height for the new node using a geometric distribution.

Insertion

- Insertion into a skiplist follows the "search and splice" rule. Steps:
  1. Randomly sample a height for the new node using a geometric distribution.
  2. "Search" for the key in order to find the appropriate position to insert it into. While you're descending into the skiplist, maintain a `levels`-sized 1D array called `update` which will hold the pointers from the towers that we descended from at every level.

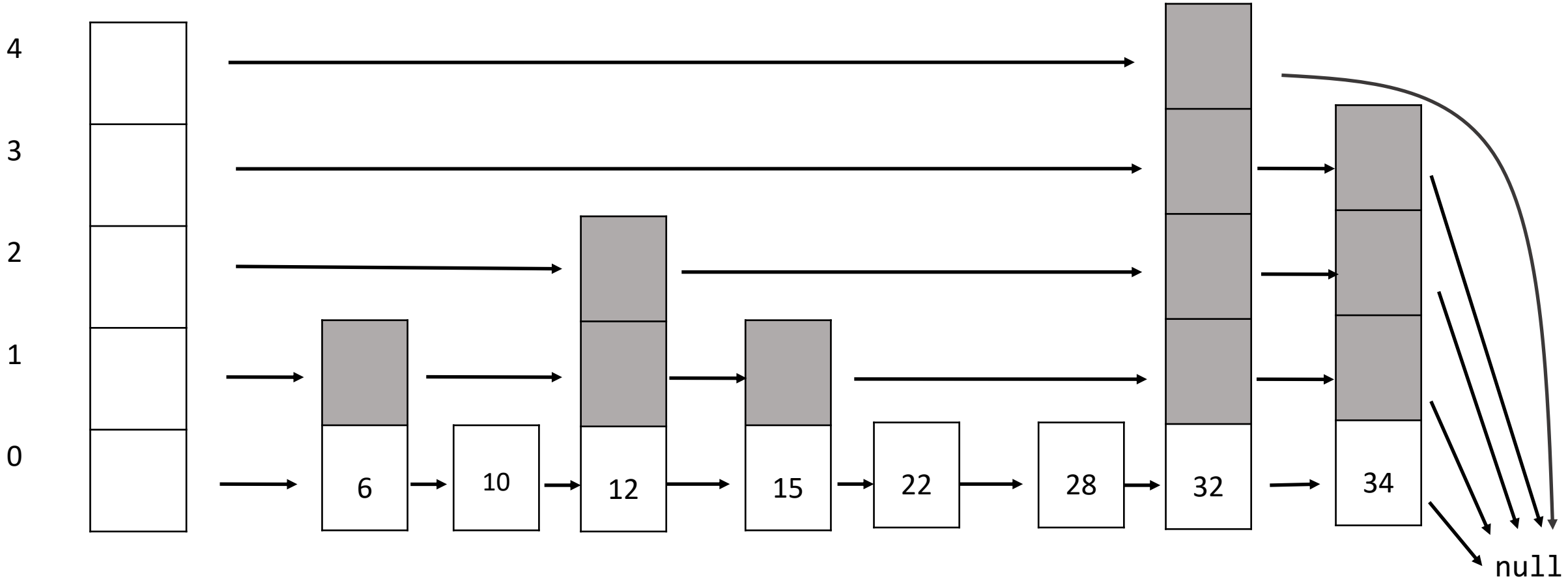
Insertion

- Insertion into a skiplist follows the "search and splice" rule. Steps:
  1. Randomly sample a height for the new node using a geometric distribution.
  2. "Search" for the key in order to find the appropriate position to insert it into. While you're descending into the skiplist, maintain a `levels`-sized 1D array called `update` which will hold the pointers from the towers that we descended from at every level.
  3. When you find the position, **splice**: insert the tower in place by using `update` to update the relevant pointers.

Insertion

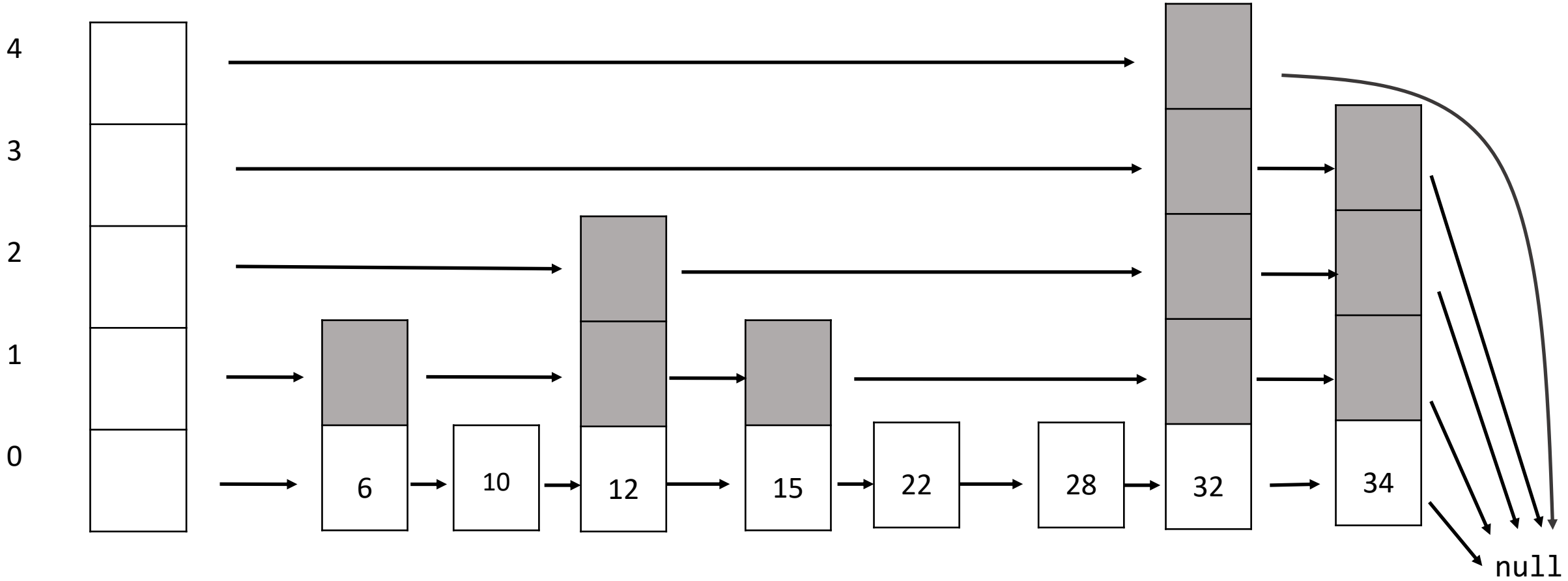
- Insertion into a skiplist follows the "search and splice" rule. Steps:
  1. Randomly sample a height for the new node using a geometric distribution.
  2. "Search" for the key in order to find the appropriate position to insert it into. While you're descending into the skiplist, maintain a `levels`-sized 1D array called `update` which will hold the pointers from the towers that we descended from at every level.
  3. When you find the position, **splice**: insert the tower in place by using `update` to update the relevant pointers.
- We employ zero-indexing: the list level is at level 0, one level above is level 1, etc.

Insertion Example



Insertion Example

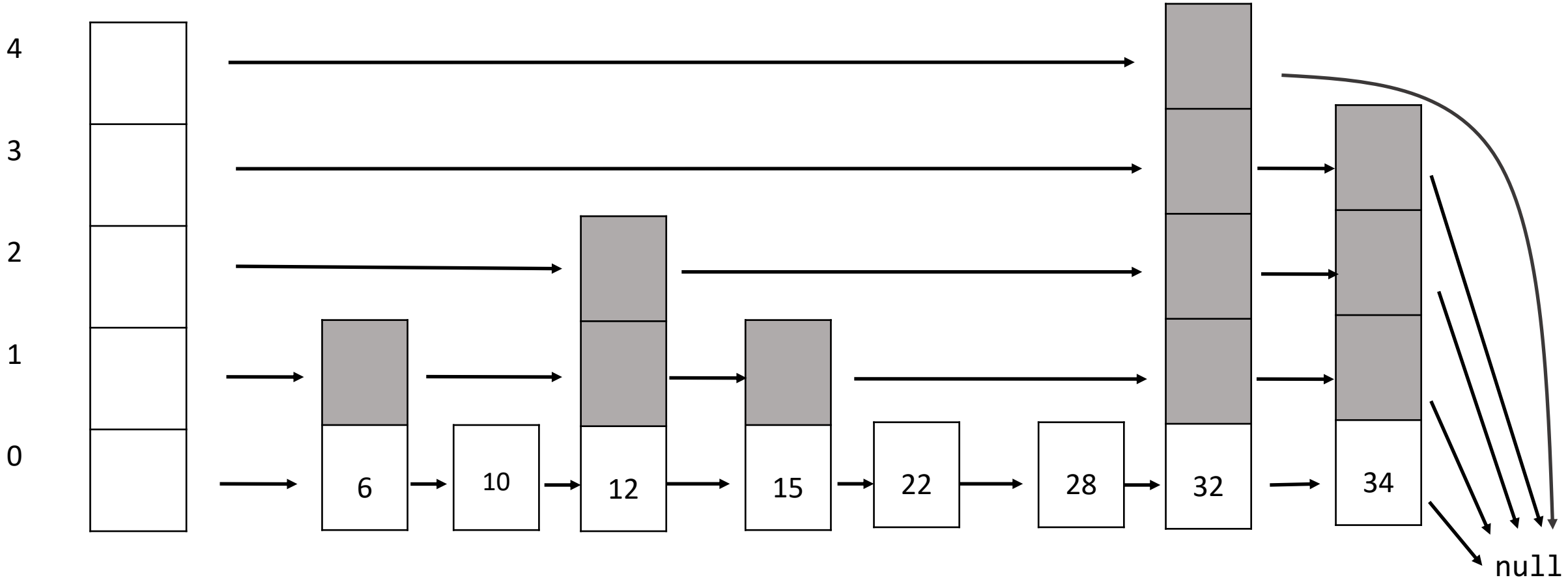
Suppose we want to insert 25.



Task: Insert 25

Insertion Example

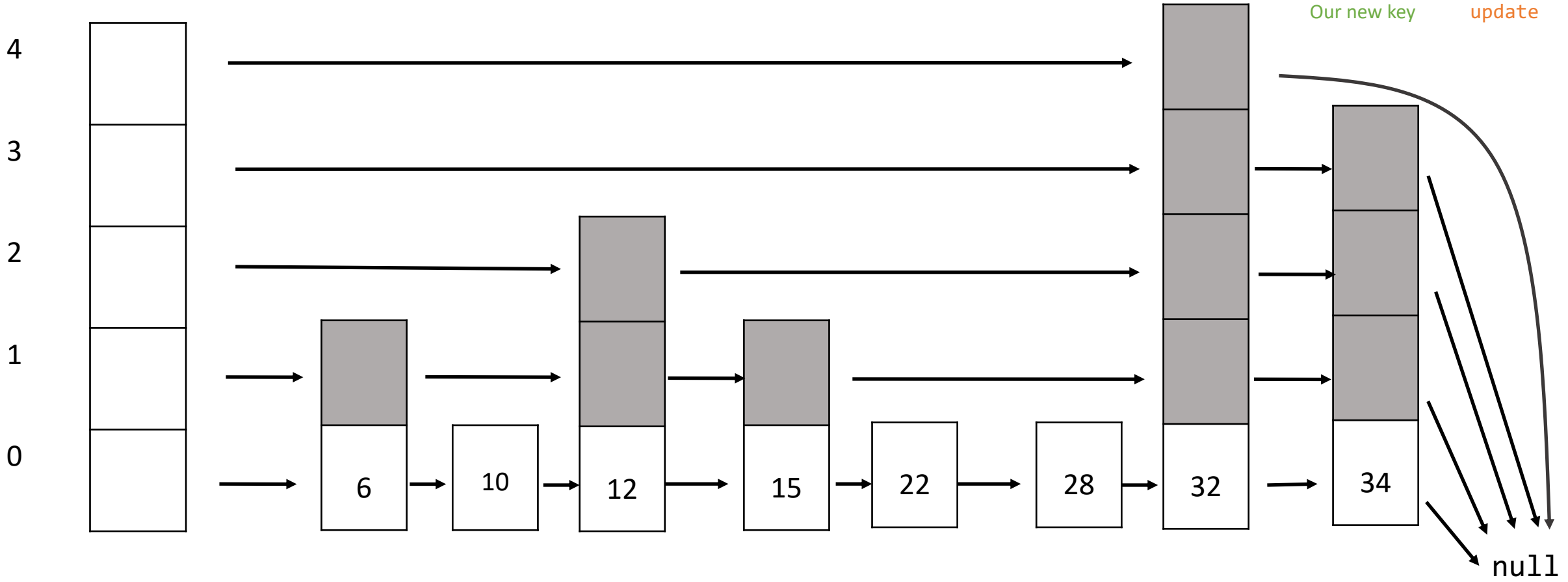
Step 1: Sample a #levels “on top” of the key. Suppose we sample 3 (probability?)



Task: Insert 25

Insertion Example

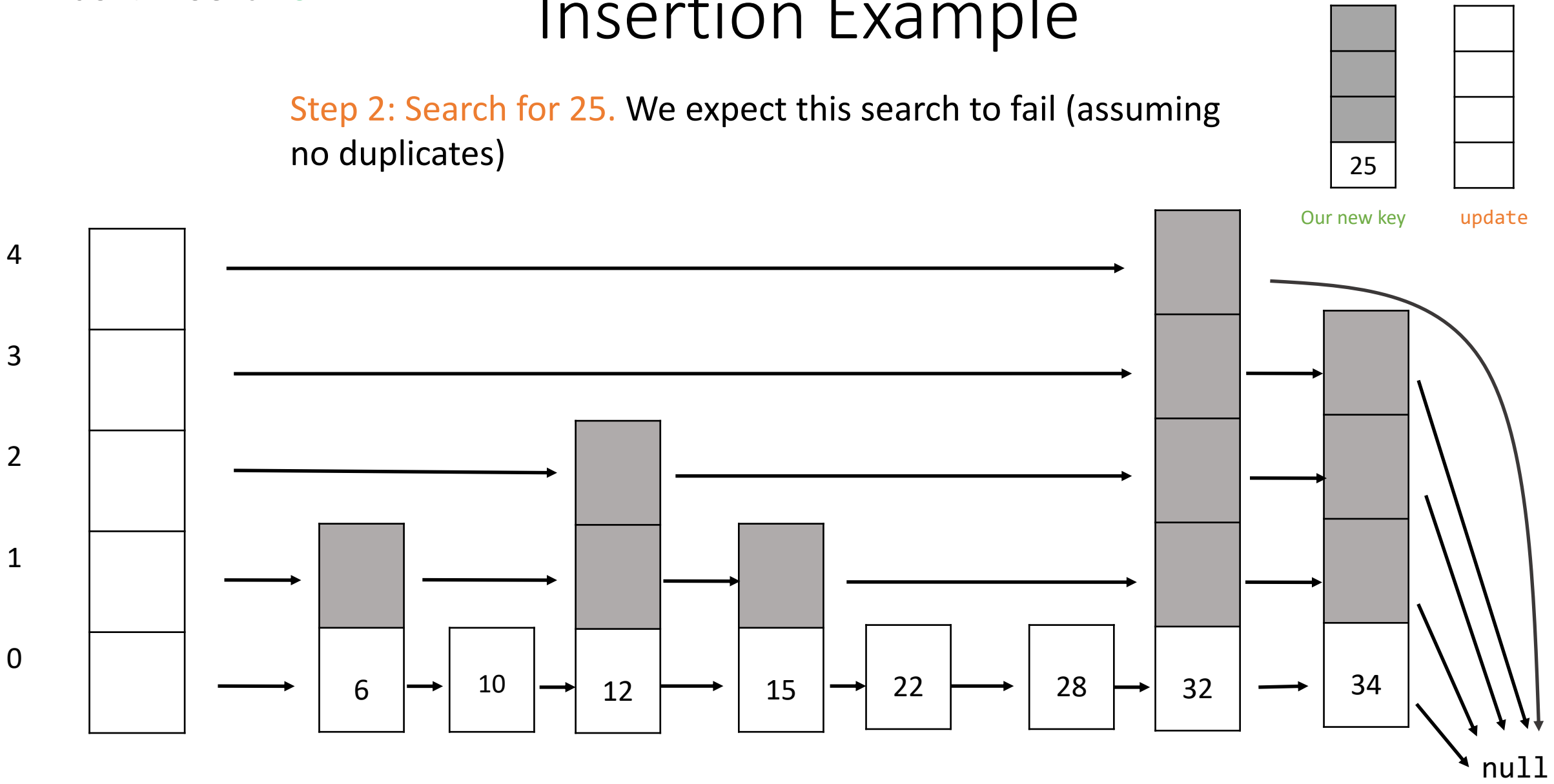
Step 1: Sample a #levels “on top” of the key. Suppose we sample 3 (probability?)



Task: Insert 25

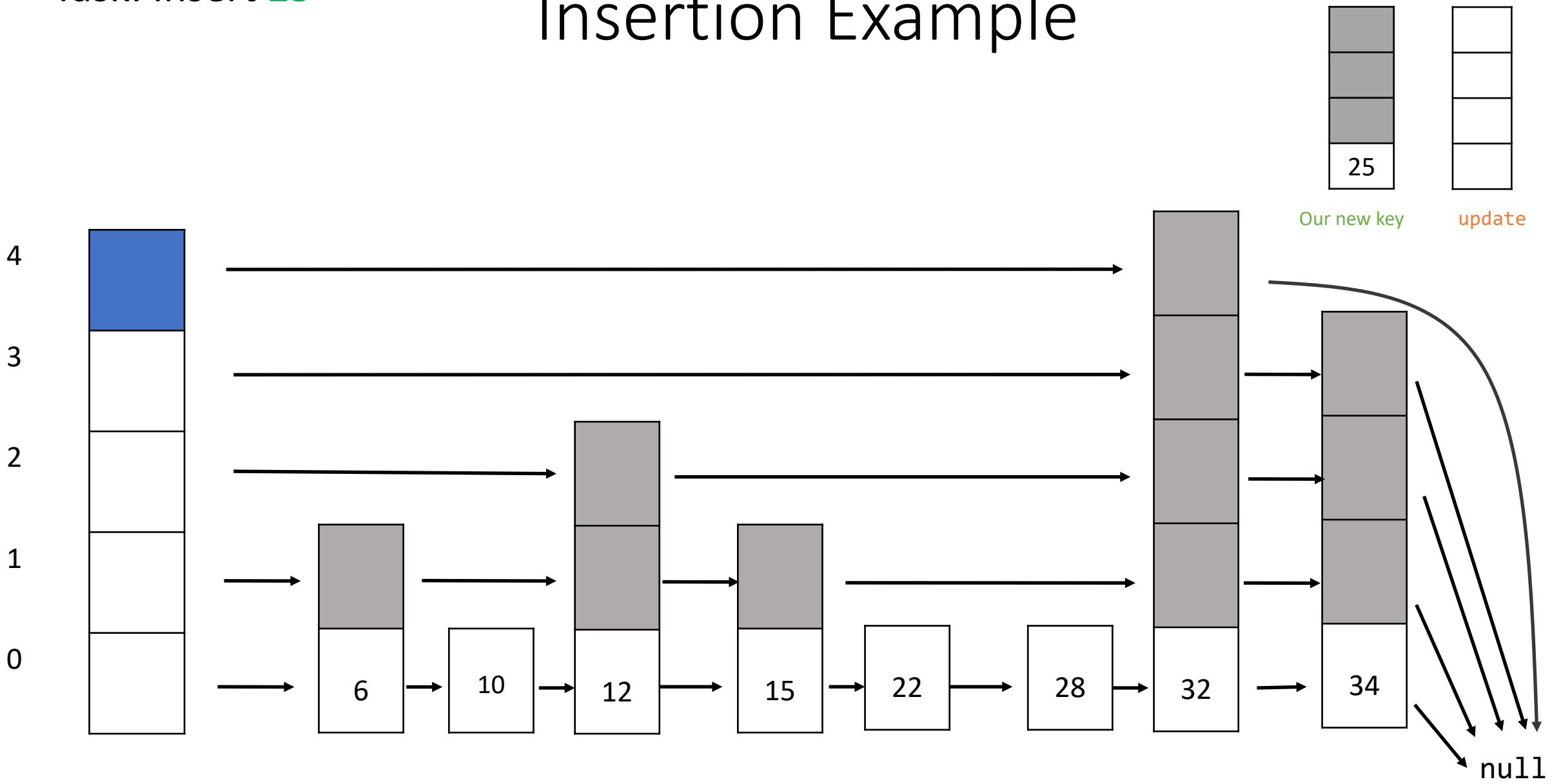
Insertion Example

Step 2: Search for 25. We expect this search to fail (assuming no duplicates)



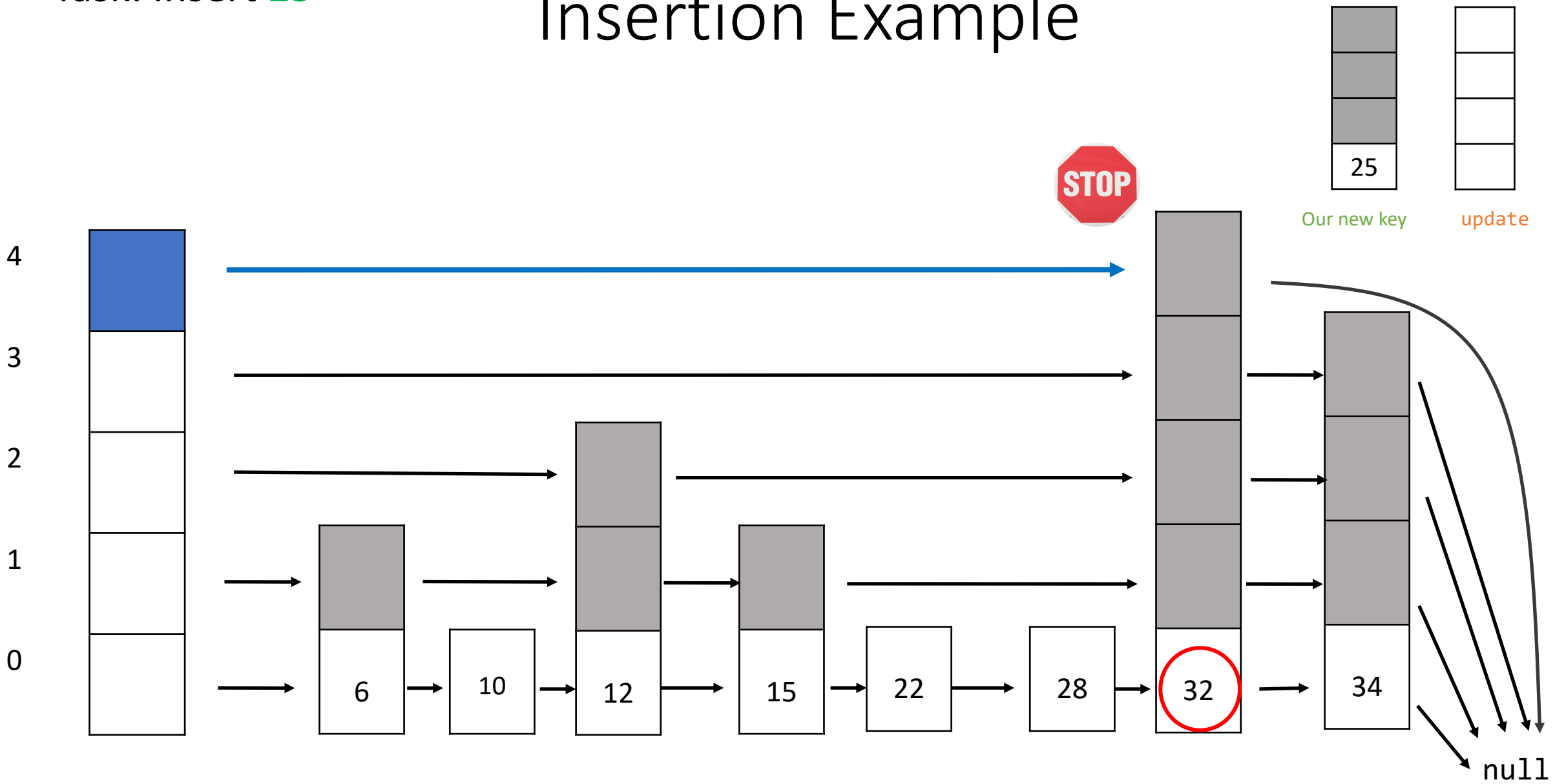
Task: Insert 25

Insertion Example



Task: Insert 25

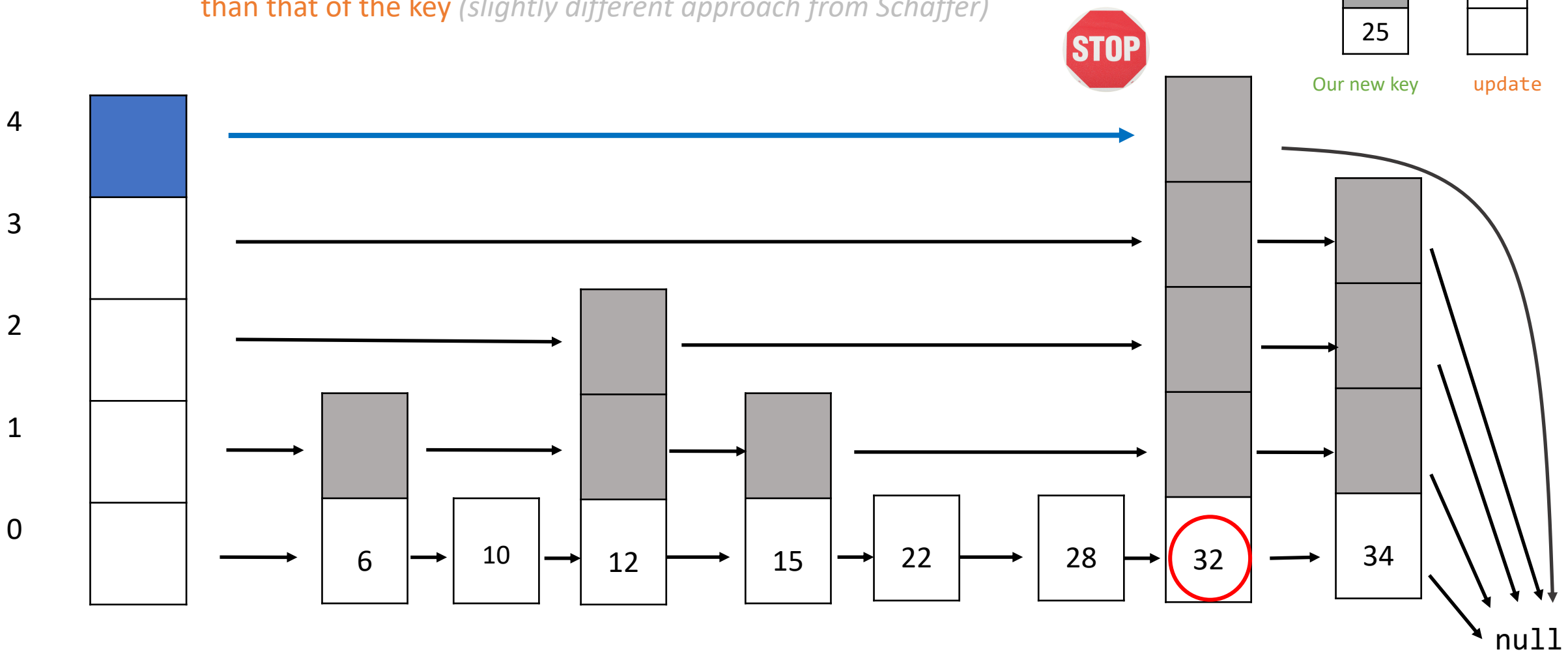
Insertion Example



Task: Insert 25

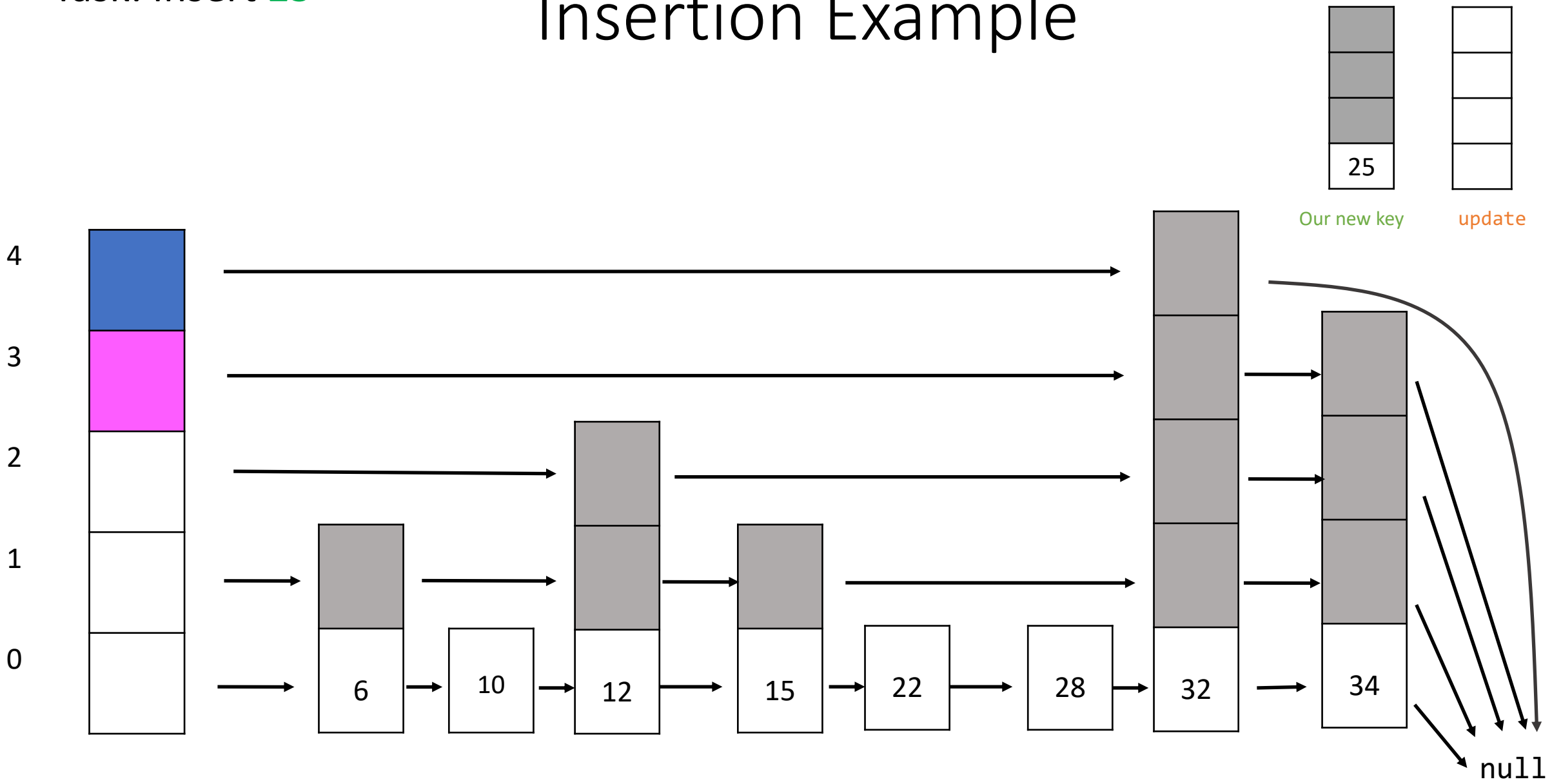
Insertion Example

We don't have to touch `update` yet, since the current #levels is 4, greater than that of the key *(slightly different approach from Schaffer)*



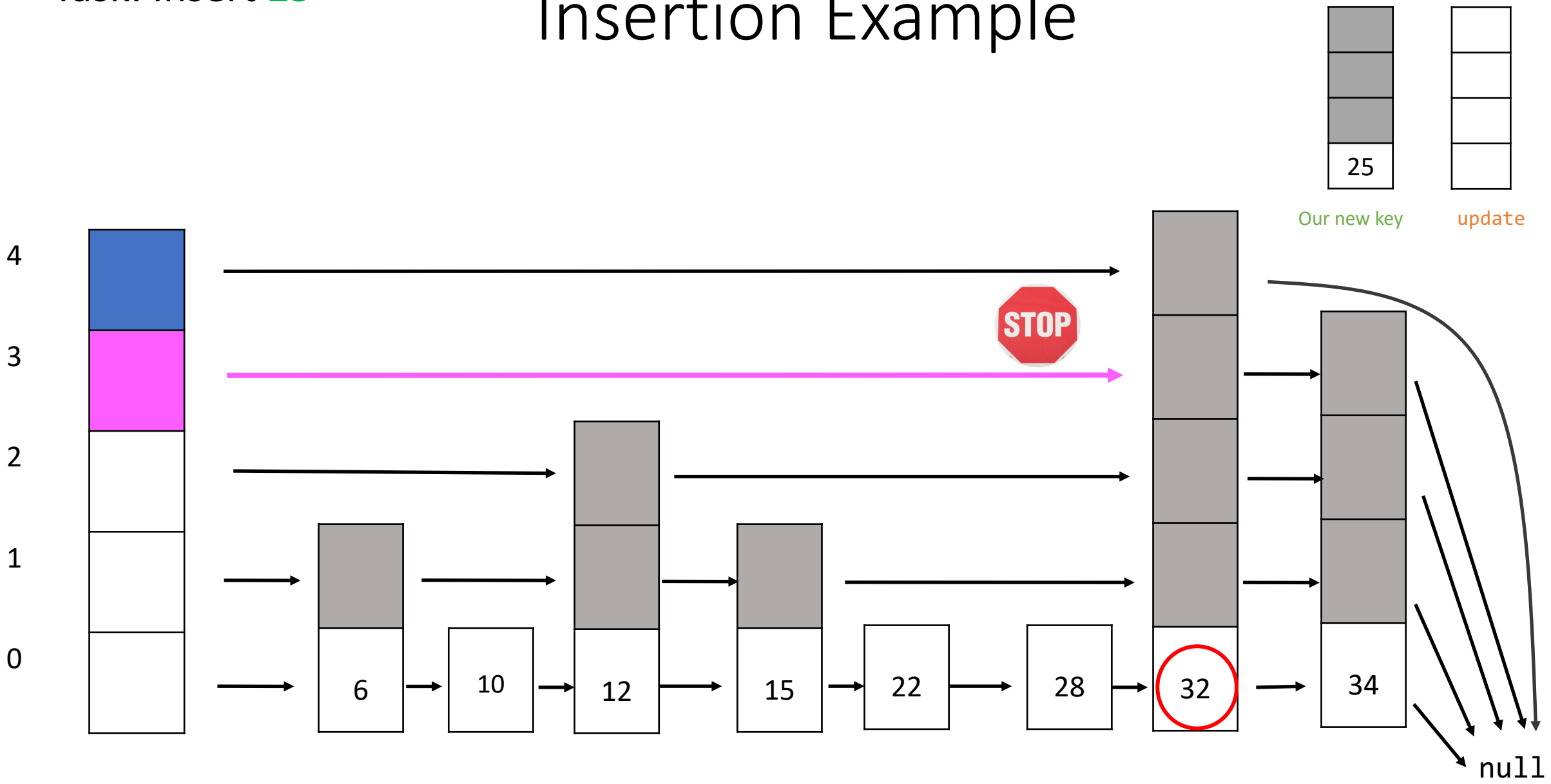
Task: Insert 25

Insertion Example



Task: Insert 25

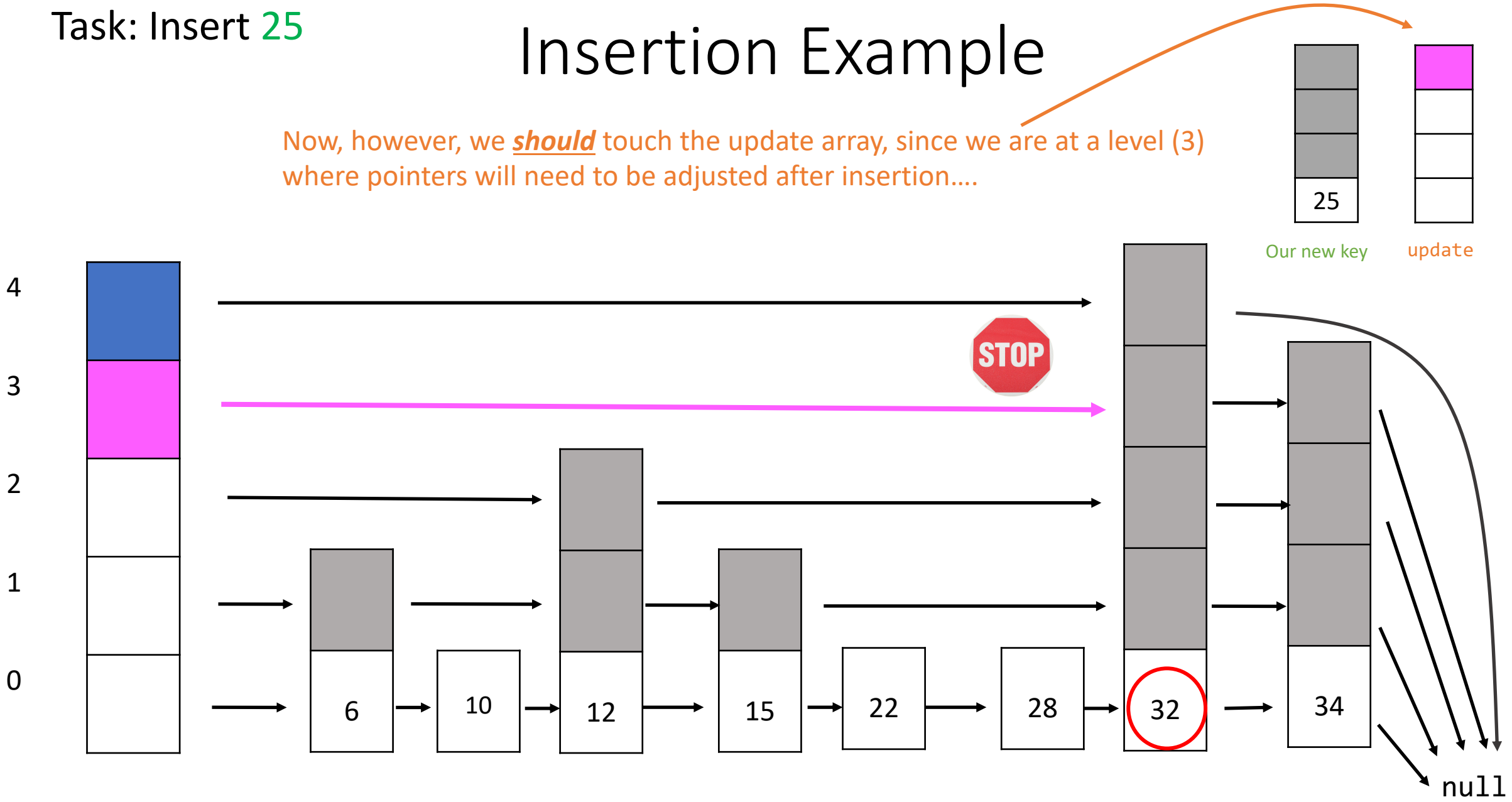
Insertion Example



Task: Insert 25

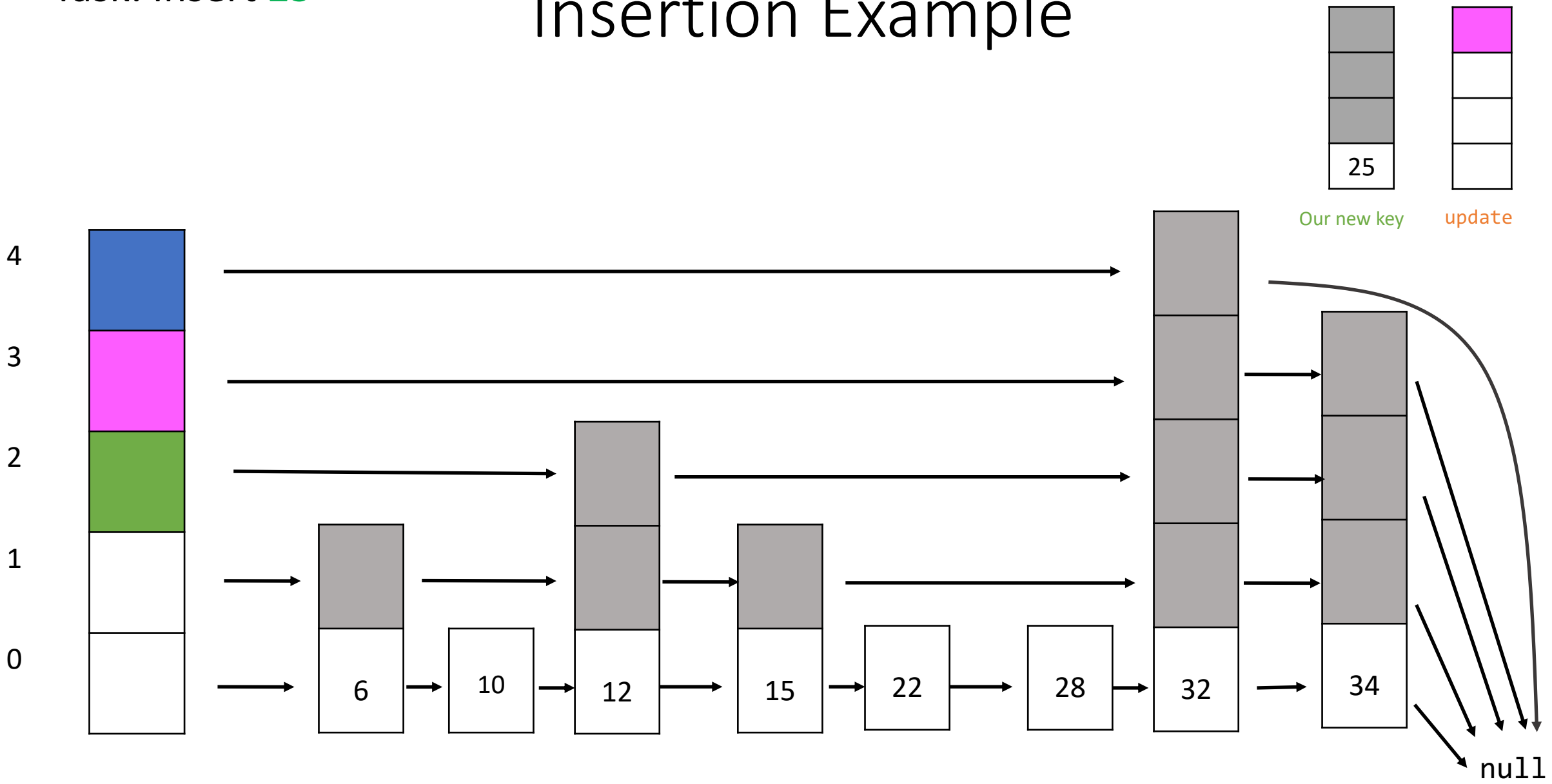
Insertion Example

Now, however, we ***should*** touch the update array, since we are at a level (3) where pointers will need to be adjusted after insertion….



Task: Insert 25

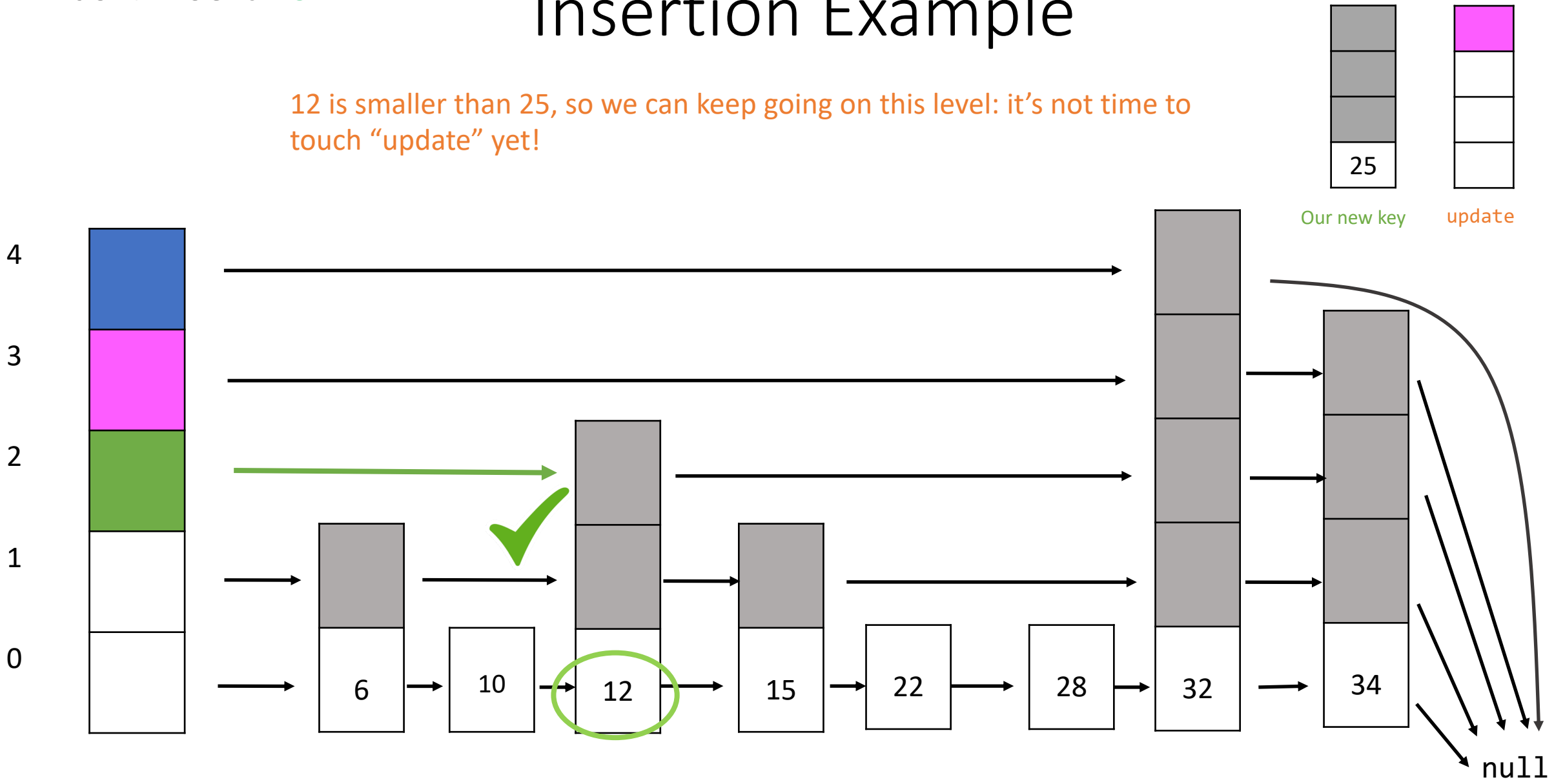
Insertion Example



Task: Insert 25

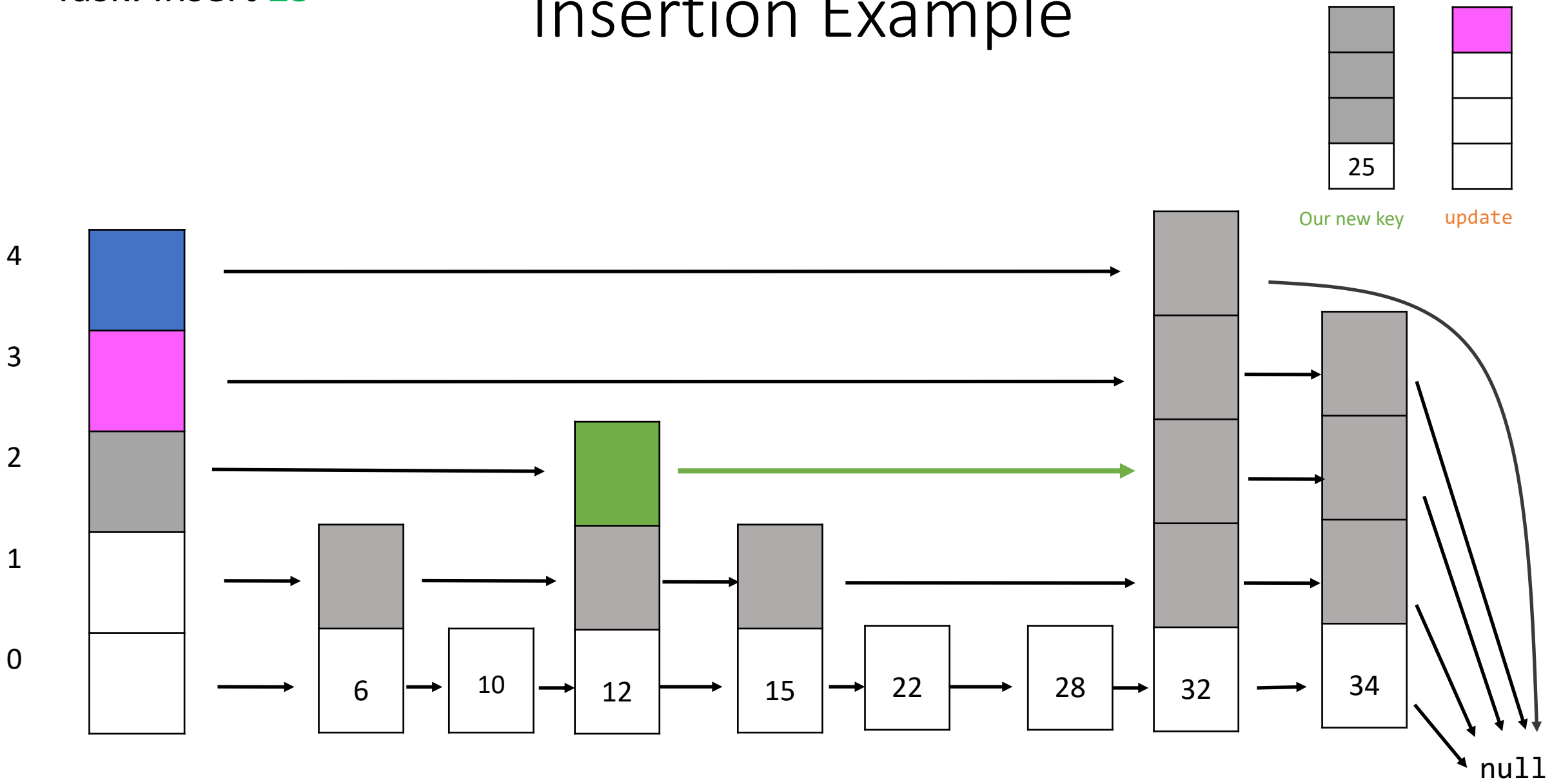
Insertion Example

12 is smaller than 25, so we can keep going on this level: it's not time to touch "update" yet!



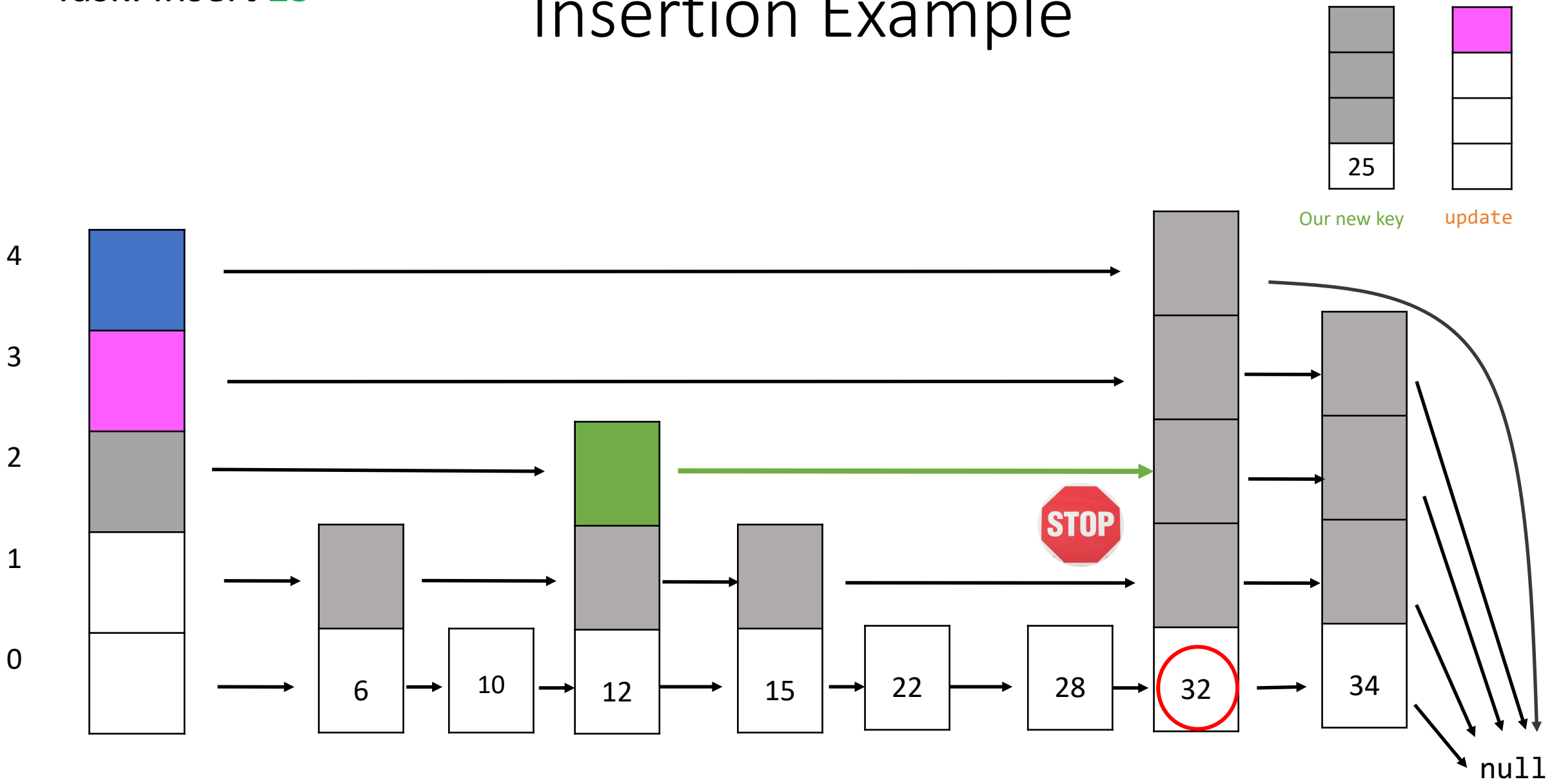
Task: Insert 25

Insertion Example



Task: Insert 25

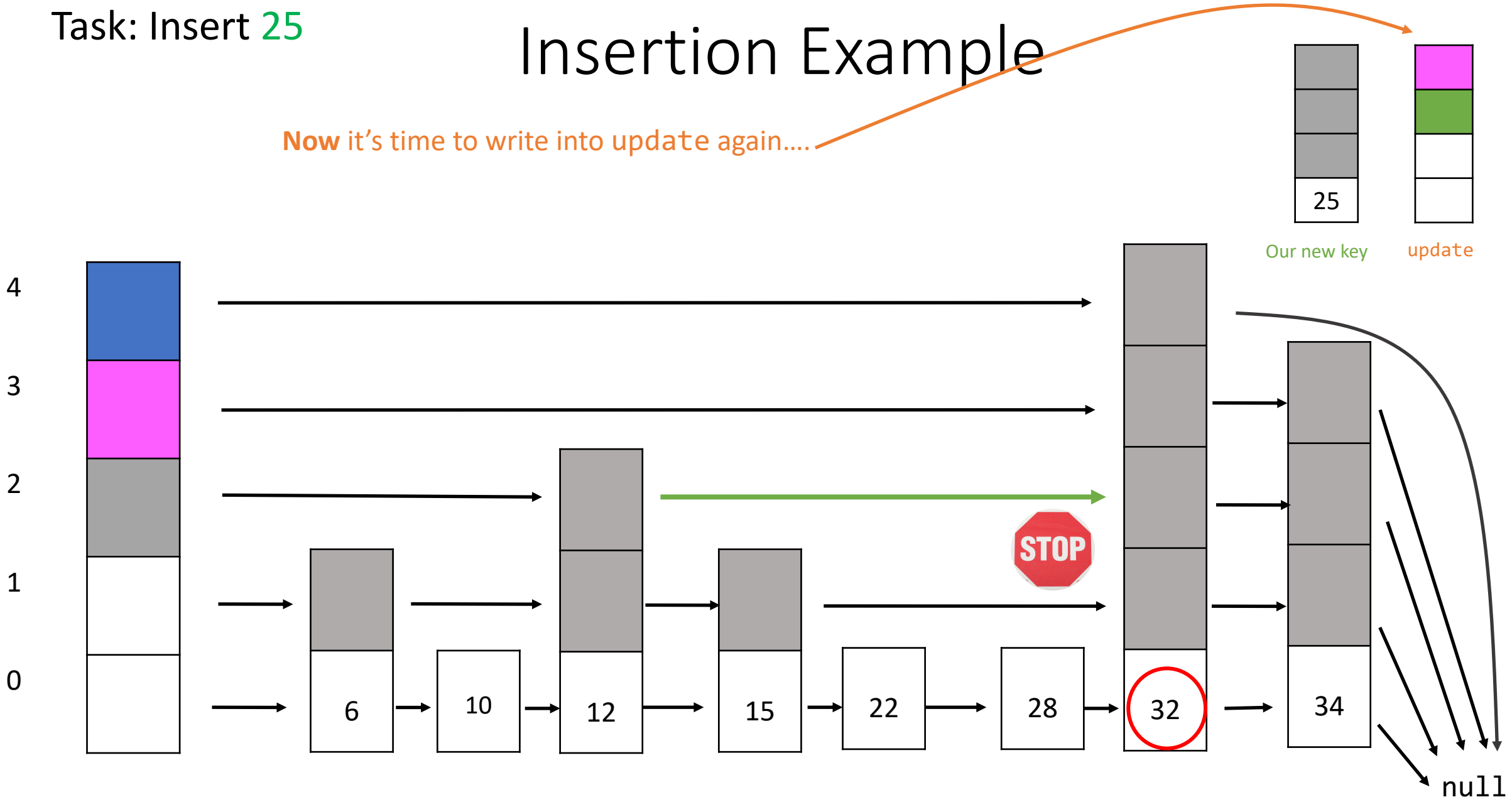
Insertion Example



Task: Insert 25

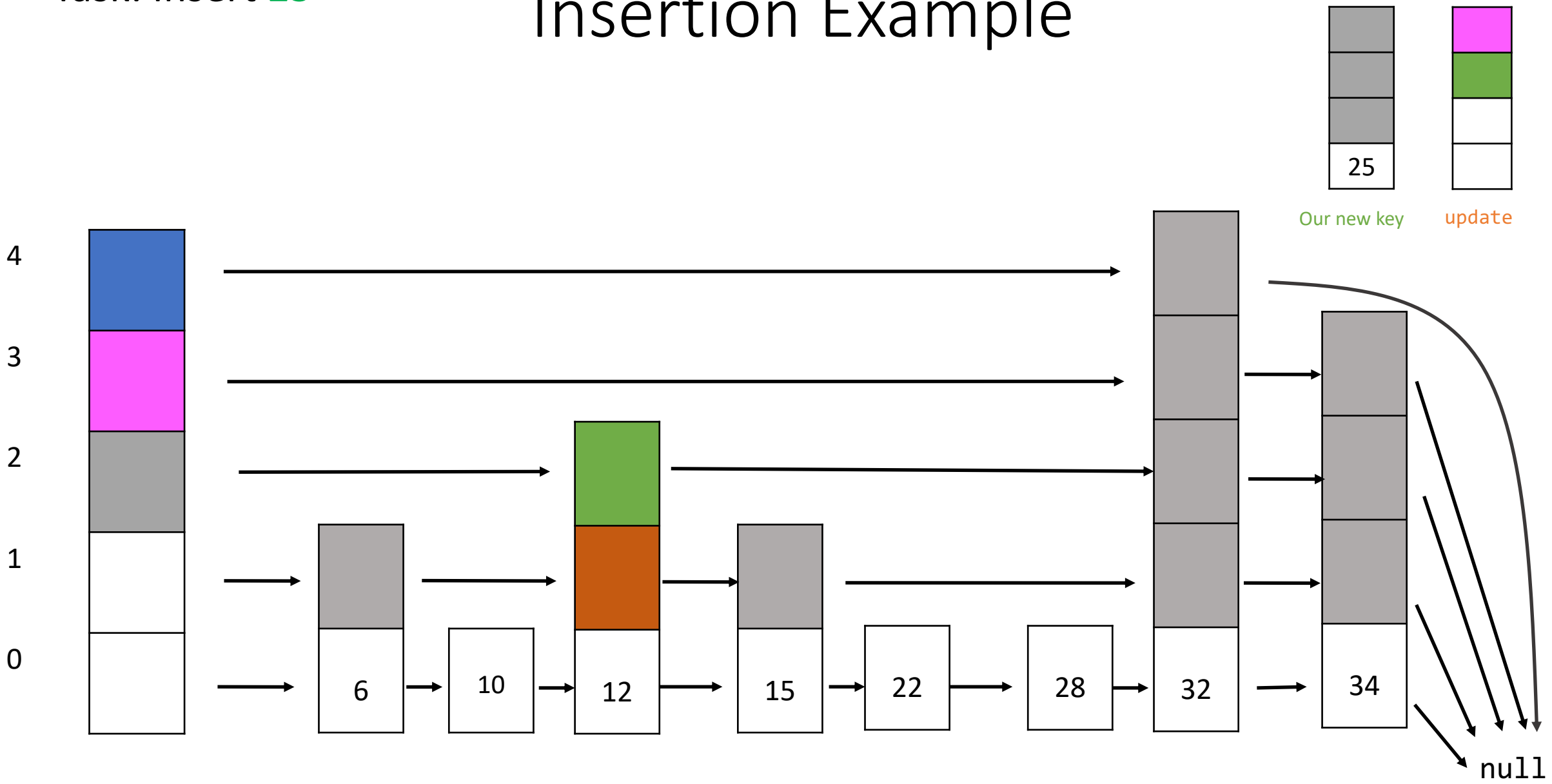
Insertion Example

Now it's time to write into update again….



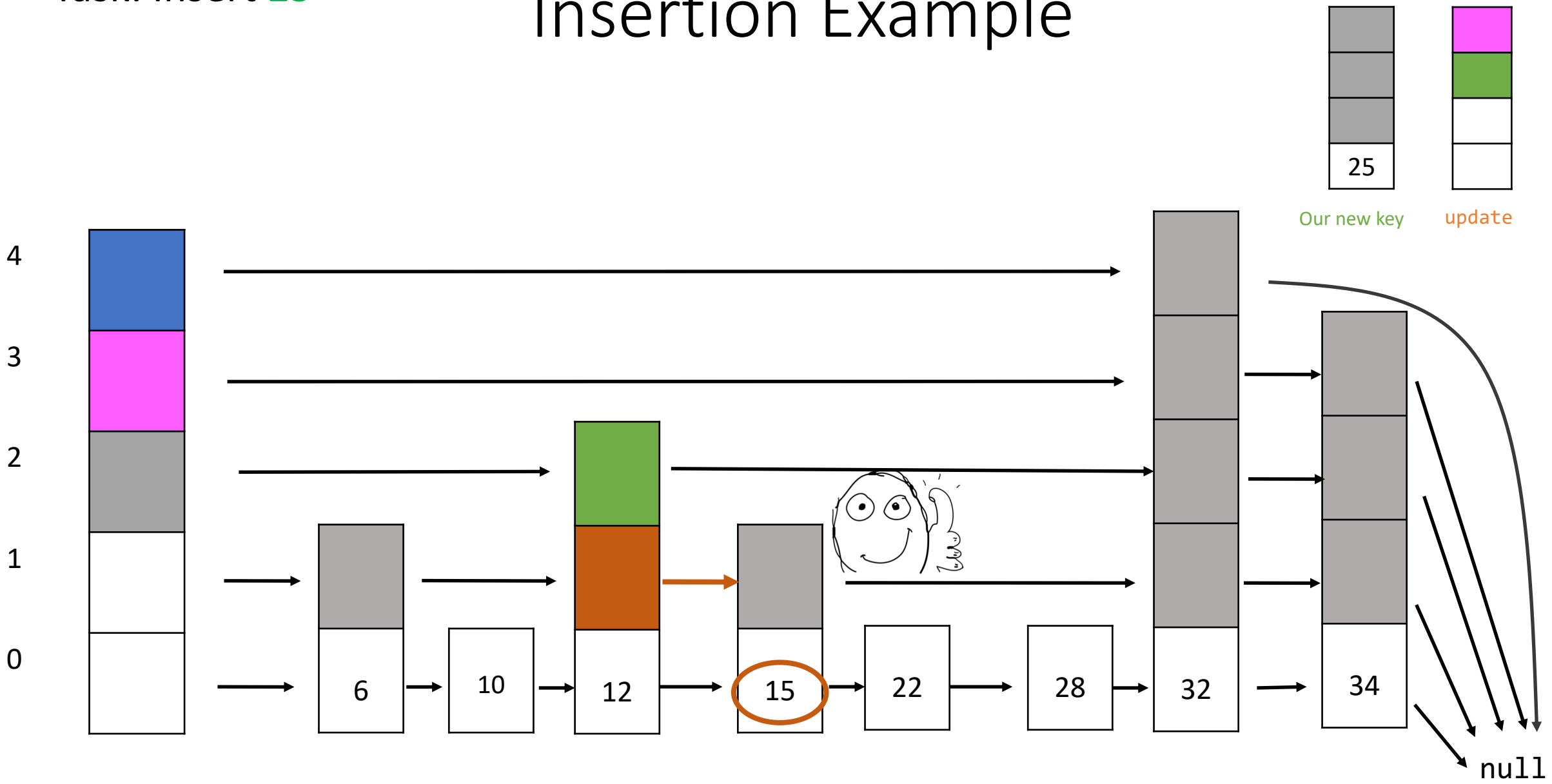
Task: Insert 25

Insertion Example



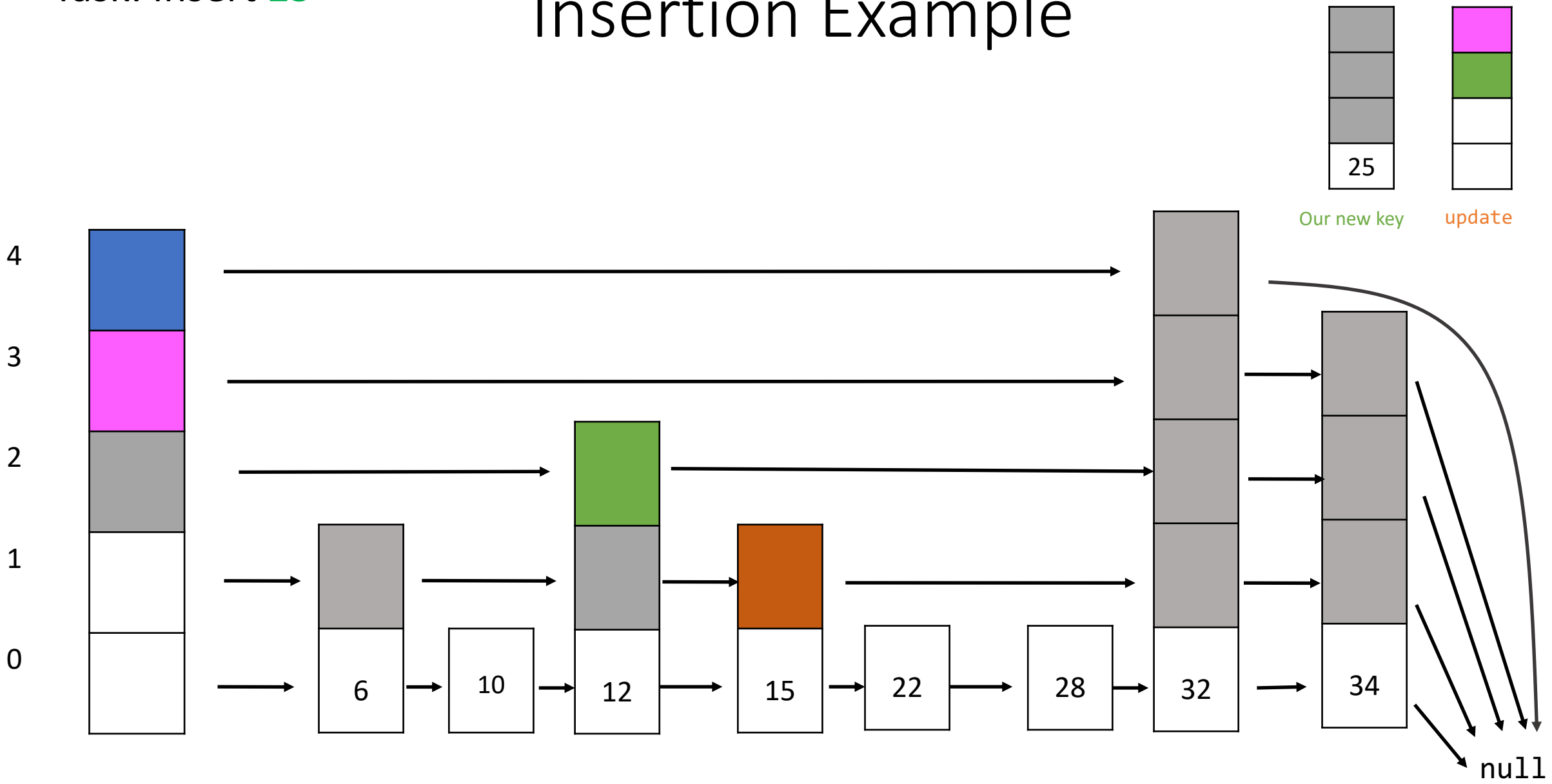
Task: Insert 25

Insertion Example



Task: Insert 25

Insertion Example



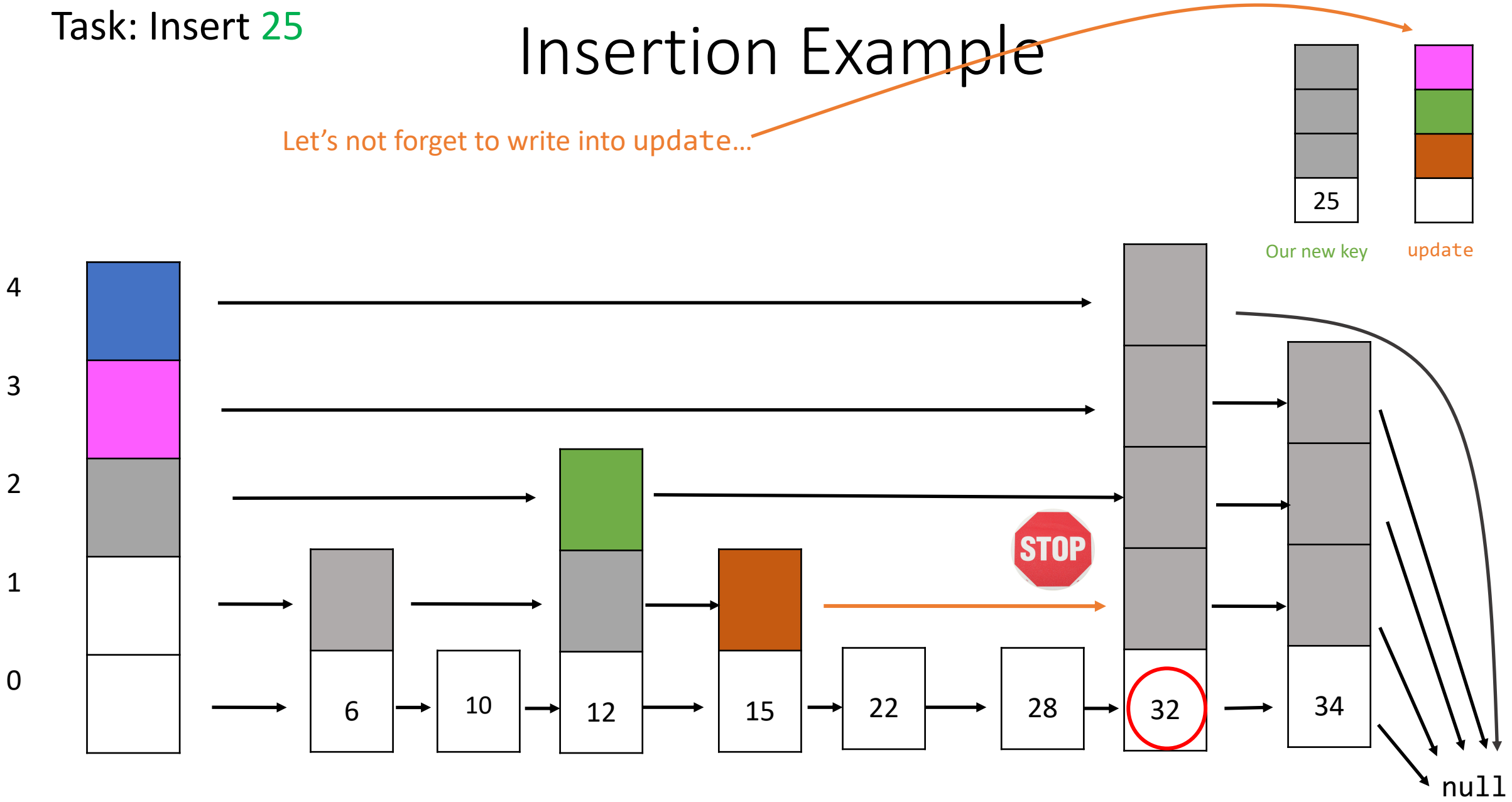
Task: Insert 25

Insertion Example

Task: Insert 25

Insertion Example

Let's not forget to write into update…



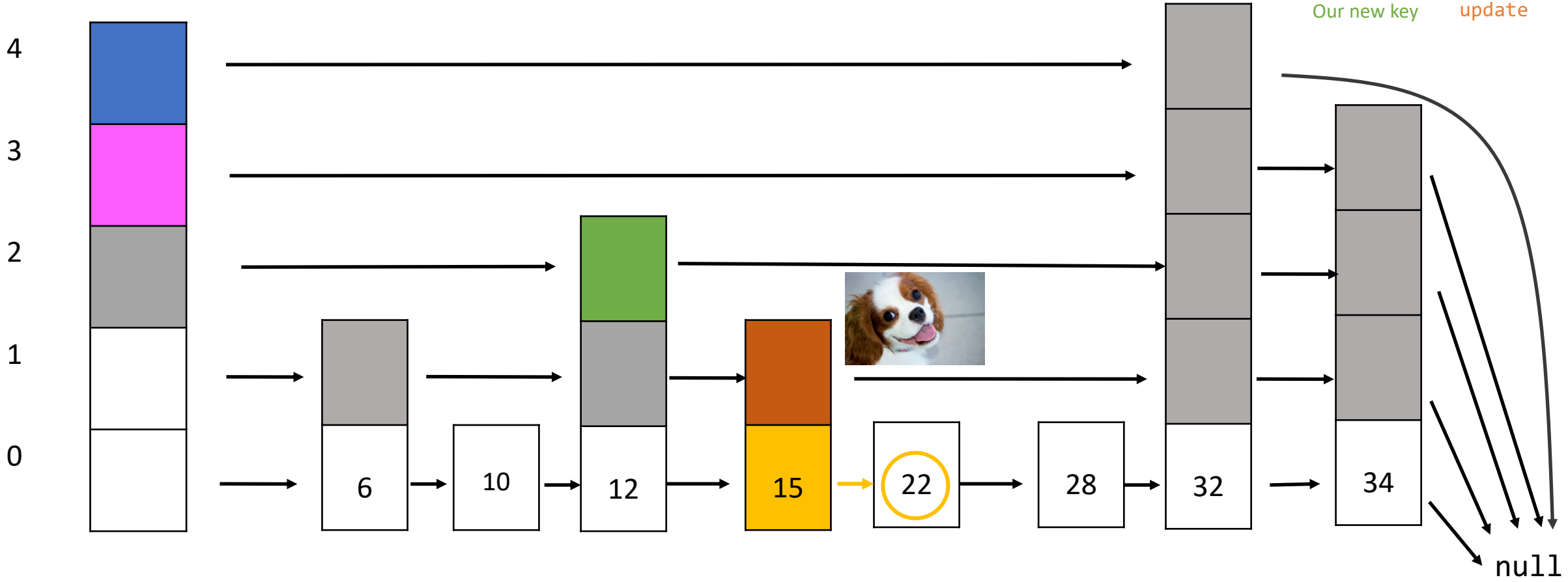
Task: Insert 25

Insertion Example

Task: Insert 25

Insertion Example

22 < 25, keep going….

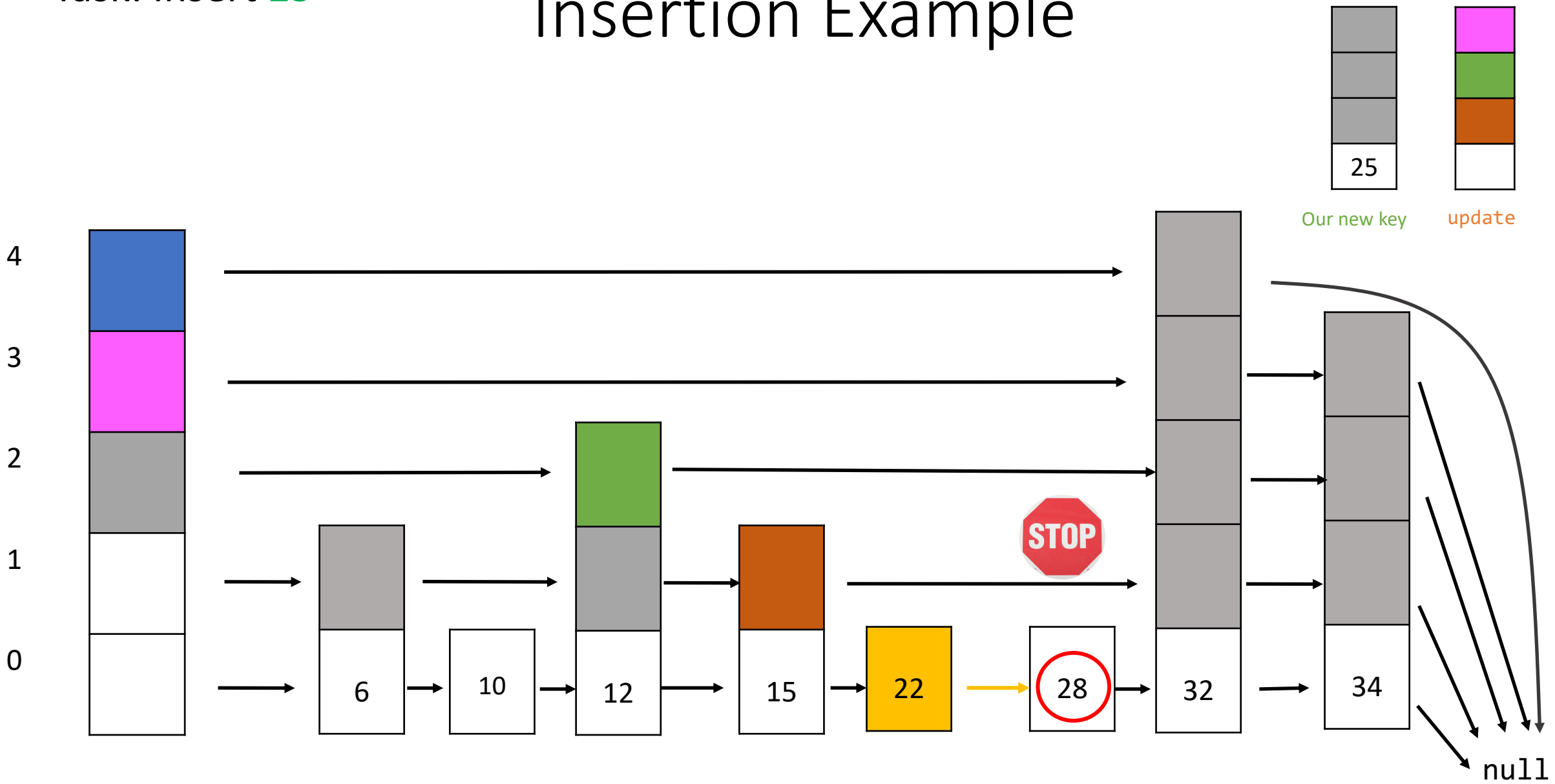


Task: Insert 25

Insertion Example

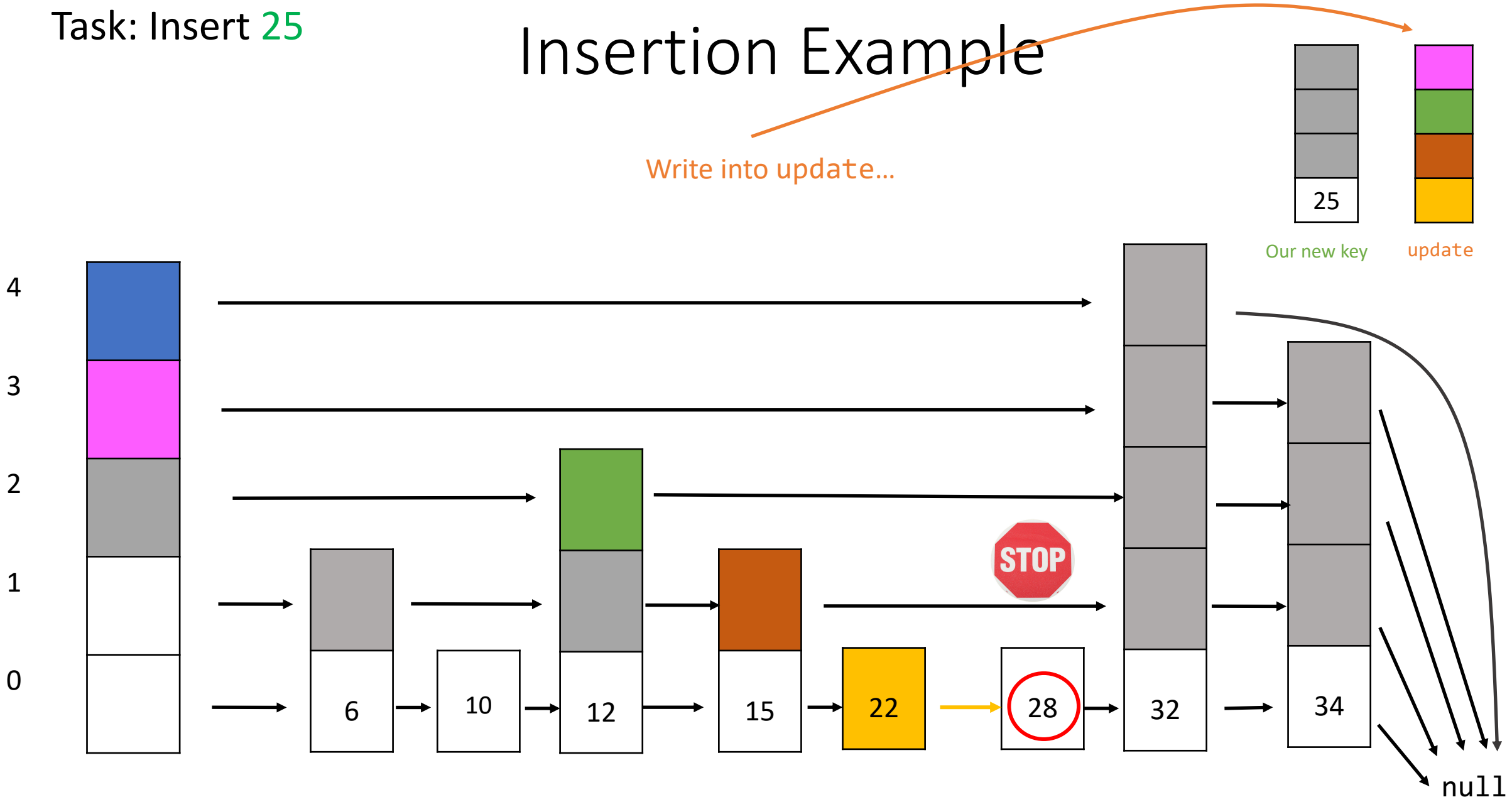
Task: Insert 25

Insertion Example



Task: Insert 25

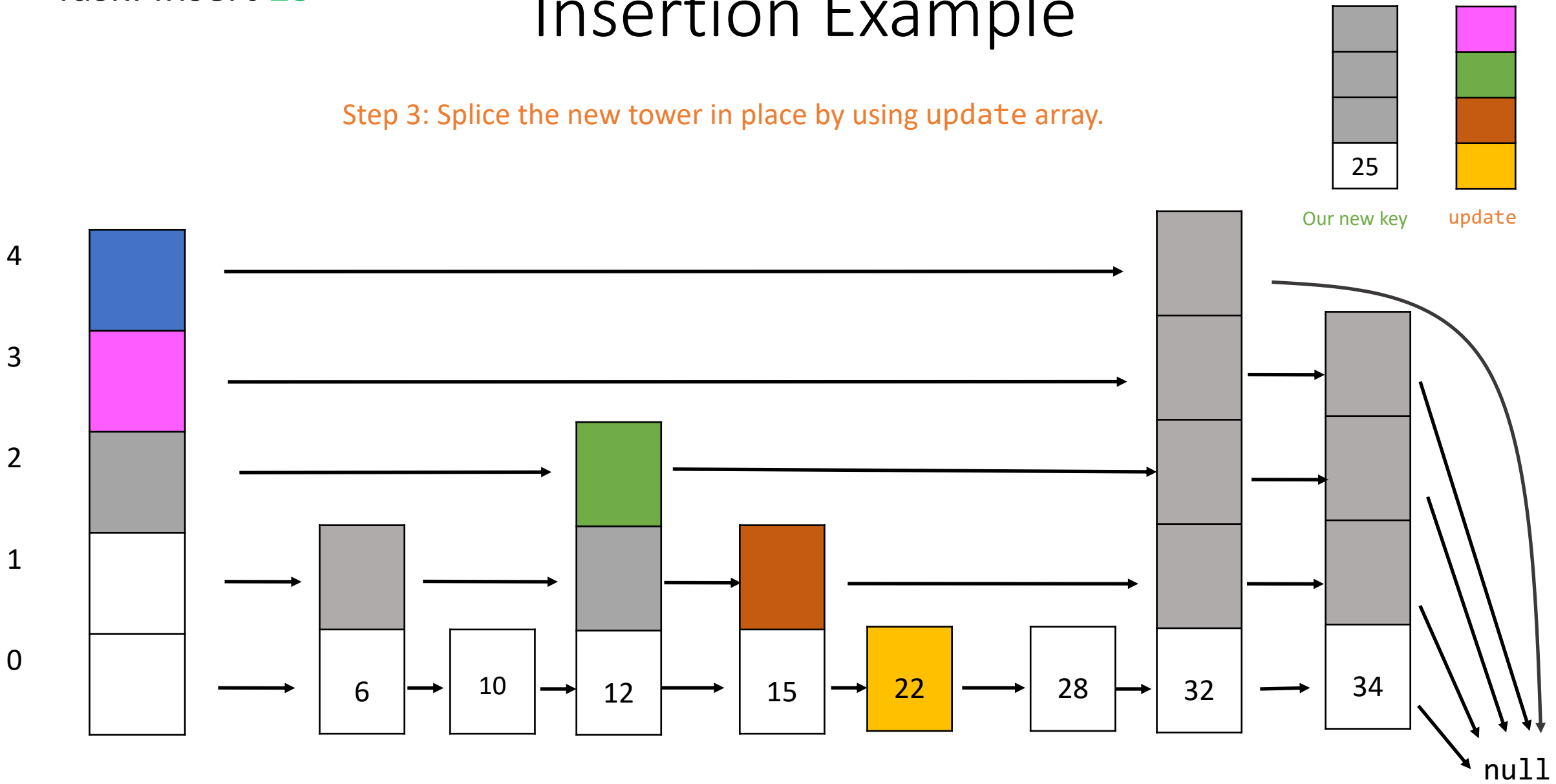
Insertion Example



Task: Insert 25

Insertion Example

Step 3: Splice the new tower in place by using `update` array.

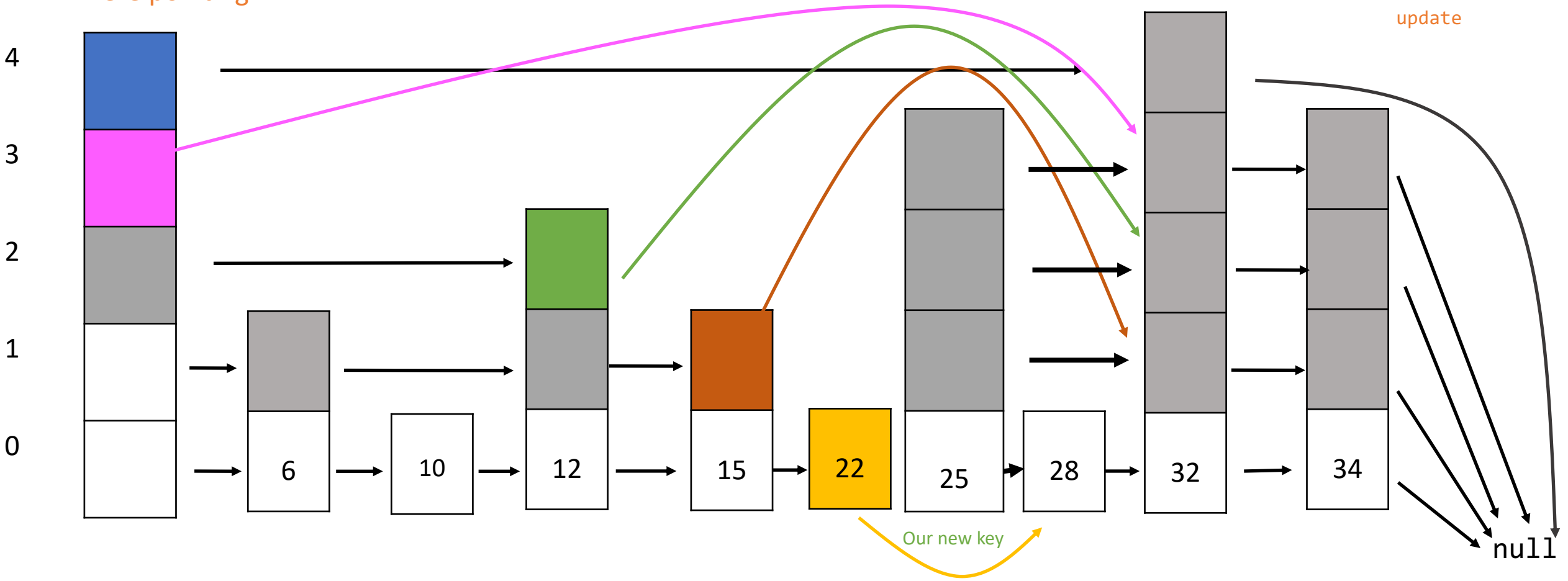


Task: Insert 25

Insertion Example

Step 3: Splice the new tower in place by using `update` array.

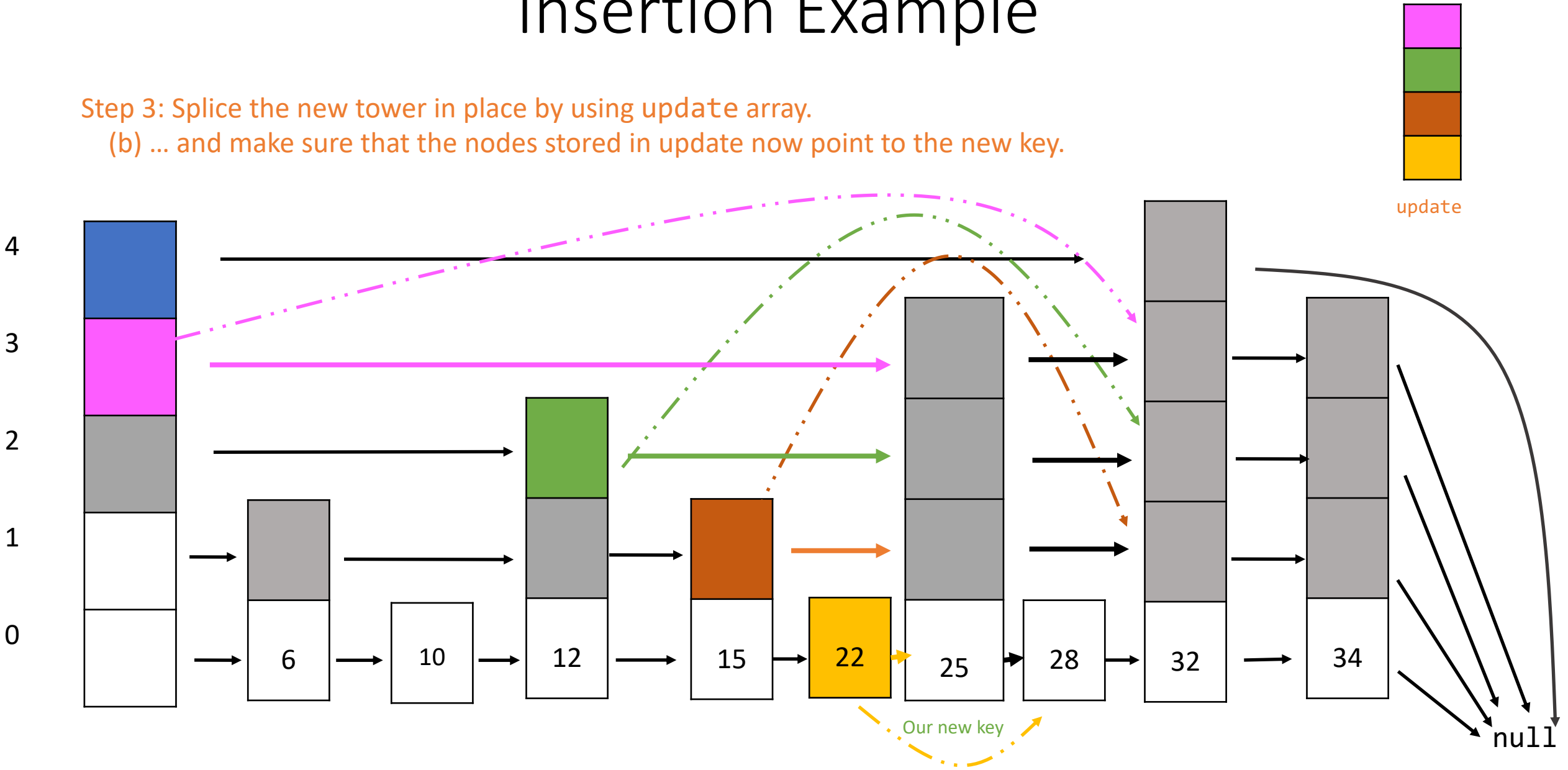
(a) Make all the forward pointers of the new key point wherever the nodes stored in `update` were pointing….



Task: Insert 25

Insertion Example

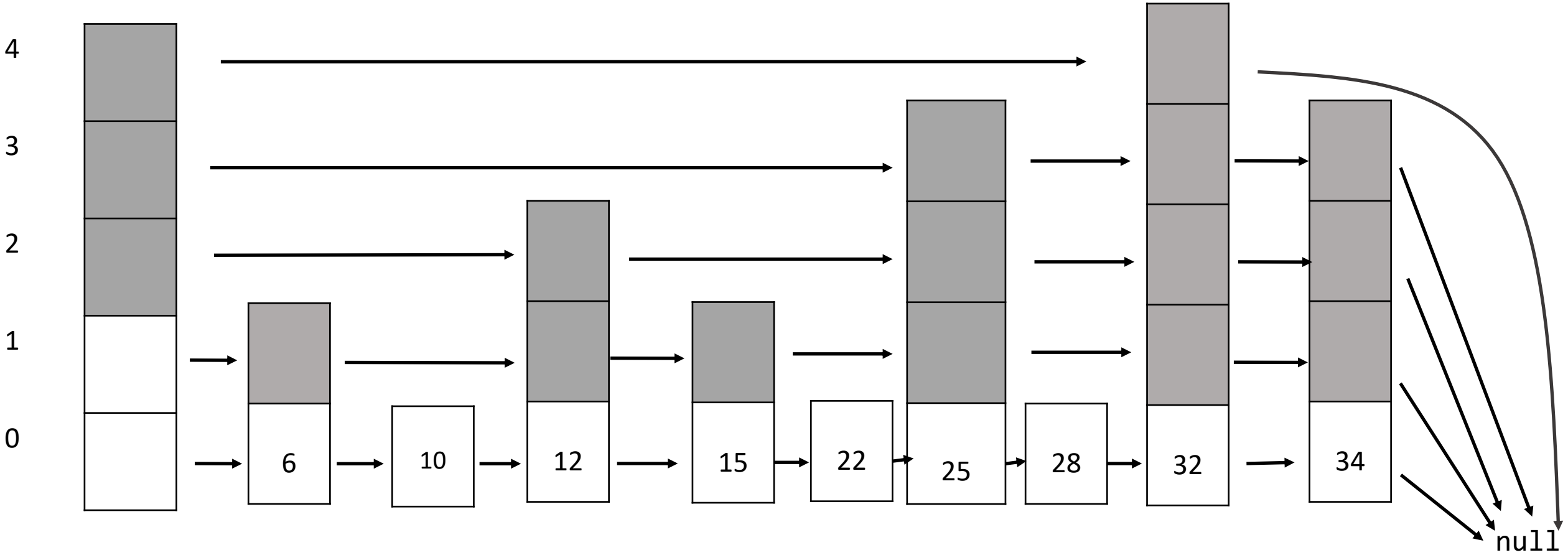
Step 3: Splice the new tower in place by using `update` array.
(b) … and make sure that the nodes stored in update now point to the new key.



Task: Insert 25

Insertion Example

Done! ☺



Live insertion demo!

- Using only the pseudocode from Shaffer's book (p. 519 in the Java edition) and a coin, let's see if we can insert some ints into an initially empty Skiplist!

```
/** Insert a record into the skiplist */
public void insert(Key k, E newValue) {
  int newLevel = randomLevel();  // New node's level
  if (newLevel > level)          // If new node is deeper
    AdjustHead(newLevel);        //   adjust the header
  // Track end of level
  SkipNode<Key,E>[] update =
        (SkipNode<Key,E>[])new SkipNode[level+1];
  SkipNode<Key,E> x = head;        // Start at header node
  for (int i=level; i>=0; i--) { // Find insert position
    while((x.forward[i] != null) &&
          (k.compareTo(x.forward[i].key()) > 0))
      x = x.forward[i];
    update[i] = x;               // Track end at level i
  }
  x = new SkipNode<Key,E>(k, newValue, newLevel);
  for (int i=0; i<=newLevel; i++) {      // Splice into list
    x.forward[i] = update[i].forward[i]; // Who x points to
    update[i].forward[i] = x;            // Who y points to
  }
  size++;                        // Increment dictionary size
}
```

Figure 16.4 Implementation for the Skip List `Insert` function.

```
/** Pick a level using a geometric distribution */
int randomLevel() {
  int lev;
  for (lev=0; DSutil.random(2) == 0; lev++); // Do nothing
  return lev;
}
```

Skiplist insertion demo!

```
/** Insert a record into the skiplist */
public void insert(Key k, E newValue) {
  int newLevel = randomLevel();  // New node's level
  if (newLevel > level)          // If new node is deeper
    AdjustHead(newLevel);        //   adjust the header
  // Track end of level
  SkipNode<Key,E>[] update =
        (SkipNode<Key,E>[])new SkipNode[level+1];
  SkipNode<Key,E> x = head;         // Start at header node
  for (int i=level; i>=0; i--) { // Find insert position
    while((x.forward[i] != null) &&
          (k.compareTo(x.forward[i].key()) > 0))
      x = x.forward[i];
    update[i] = x;               // Track end at level i
  }
  x = new SkipNode<Key,E>(k, newValue, newLevel);
  for (int i=0; i<=newLevel; i++) {      // Splice into list
    x.forward[i] = update[i].forward[i]; // Who x points to
    update[i].forward[i] = x;            // Who y points to
  }
  size++;                          // Increment dictionary size
}

/** Pick a level using a geometric distribution */
int randomLevel() {
  int lev;
  for (lev=0; DSutil.random(2) == 0; lev++); // Do nothing
  return lev;
}
```

Worksheet exercise!

Deletion

- Deletion works very much like insertion.
- Two differences:
  1. We don't have a need to sample a height.
  2. We use the information in the ***update*** array differently: instead of connecting nodes "before" the new tower to it and connecting the new tower to subsequent towers, we remove the existing pointers.

Analysis

- Searching for an appropriate position dominates the cost for both insertions and deletions.
- Unit cost: visiting a node on our way to the appropriate position.
- Therefore, the length of the path that connects the sought key with `header[levels-1]` dominates our complexity.
- It can be proven (e.g check out "Analysis of expected search cost" here) that, with very high probability, this length is bounded above by a scalar multiple of $\log_2 n$.

Analysis

- Searching for an appropriate position dominates the cost for both insertions and deletions.
- Unit cost: visiting a node on our way to the appropriate position.
- Therefore, the length of the path that connects the sought key with `header[levels-1]` dominates our complexity.
- It can be proven (e.g check out "Analysis of expected search cost" here) that, with very high probability, this length is bounded above by a scalar multiple of $\log_2 n$.
- So, with very high probability, searching a key k in a skiplist ℓ is $\mathcal{O}(\log_2 n)$!

Why skiplists?

- Excellent distributed performance.
  - Locked **and** lock-free implementations exist
  - E.g check out Java's `ConcurrentSkipList`
- Compare with AVL trees and Red-Black trees where locking entire paths or even the entire tree might be necessary because of successive rotations.
- Local updates ensure that you only need to lock at most two towers per insertion, and only at the very end.
- Header node immutable; never needs to be locked.

Why skiplists?

- Insensitivity in insertion order.
  - AVL + Red-Black trees might make balance guarantees, but a bad insertion order can affect performance of insertions!
- Ease of implementation
  - No recursion overhead: core of search algorithm is a double loop (while inside a for)
- Linearity of data and local updates allow for distribution of data across nodes (very good horizontal scalability)
  - Compare with AVL and Red-Black trees where storing different subtrees in different nodes might make it hard to propagate rotations from one subtree to the other.

Disadvantages of skiplists

- $O(n)$ storage overhead.
- Probabilistic instead of deterministic guarantees
  - Compare with AVL and Red-Black trees which offer guarantees of $\log_2 n$ and $2 \cdot \log_2 n$ respectively.
- Worst-case search constant bigger than AVL *(AVL-1)* Trees and B-Trees, on par with Red-Black trees.