CS24 Week 8 Lecture 2

Kyle Dewey

Overview

- Depth-first traversals
- Removing elements from a BST
- Priority queues
- Heaps

Depth-First Traversals

On Using Stacks

- We can cut out the explicit stack by using the call stack implicitly via recursion

```
void traverse(Node* current) {
  if (current != NULL) {
    traverse(current->getLeft());
    traverse(current->getRight());
  }
}
```

Specific Kinds of DFS Traversals

- Depending on when we process the current node, there are three general kinds of DFS traversals:
  - Pre-order: process current first
  - In-order: process current between left and right
  - Post-order: process current after left and right

Pre-Order Traversal

```
void traverse(Node* current) {
  if (current != NULL) {
    process(current);
    traverse(current->getLeft());
    traverse(current->getRight());
  }
}
```

In-Order Traversal

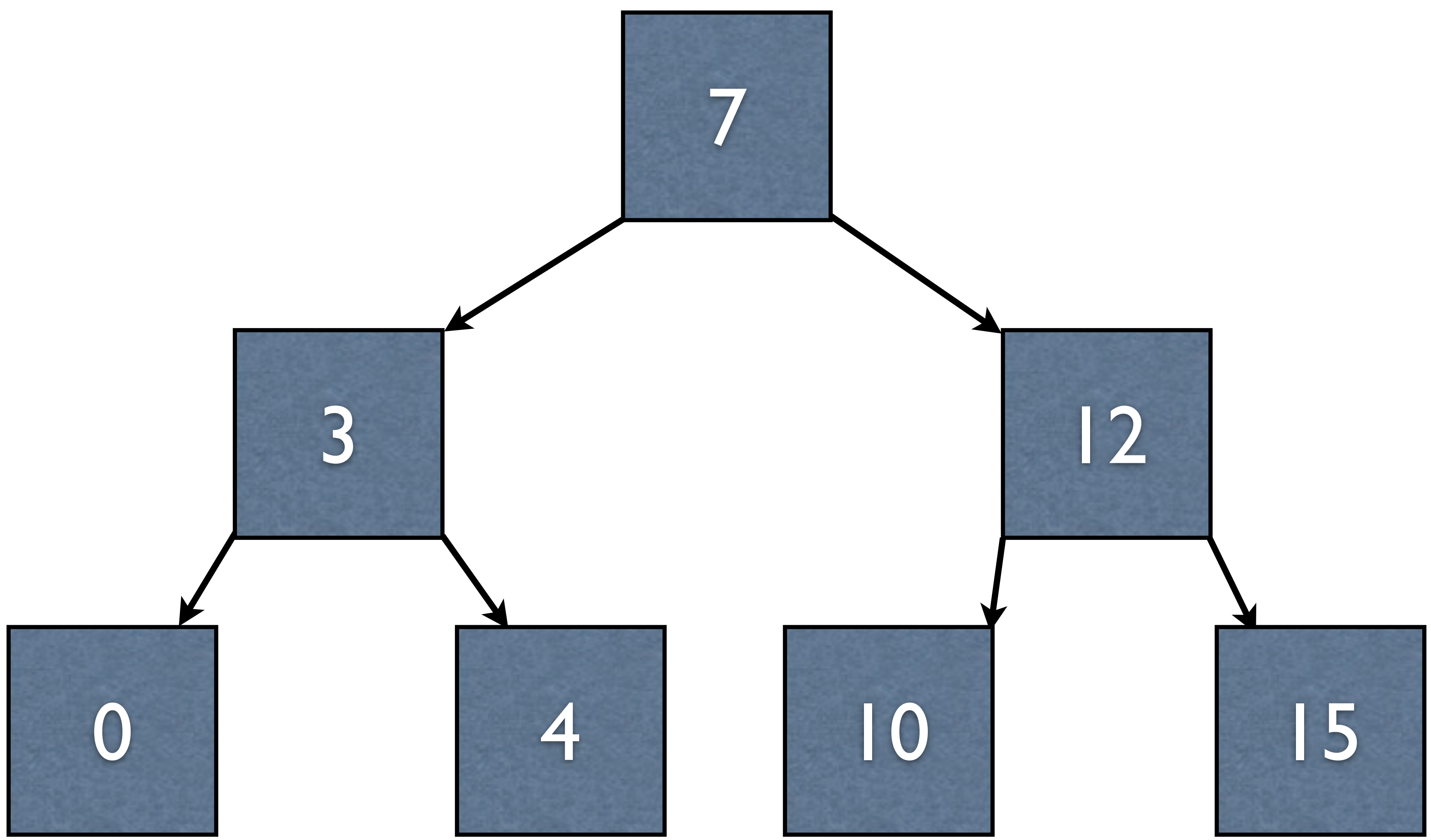
```
void traverse(Node* current) {
  if (current != NULL) {
    traverse(current->getLeft());
    process(current);
    traverse(current->getRight());
  }
}
```

Post-Order Traversal

```
void traverse(Node* current) {
  if (current != NULL) {
    traverse(current->getLeft());
    traverse(current->getRight());
    process(current);
  }
}
```

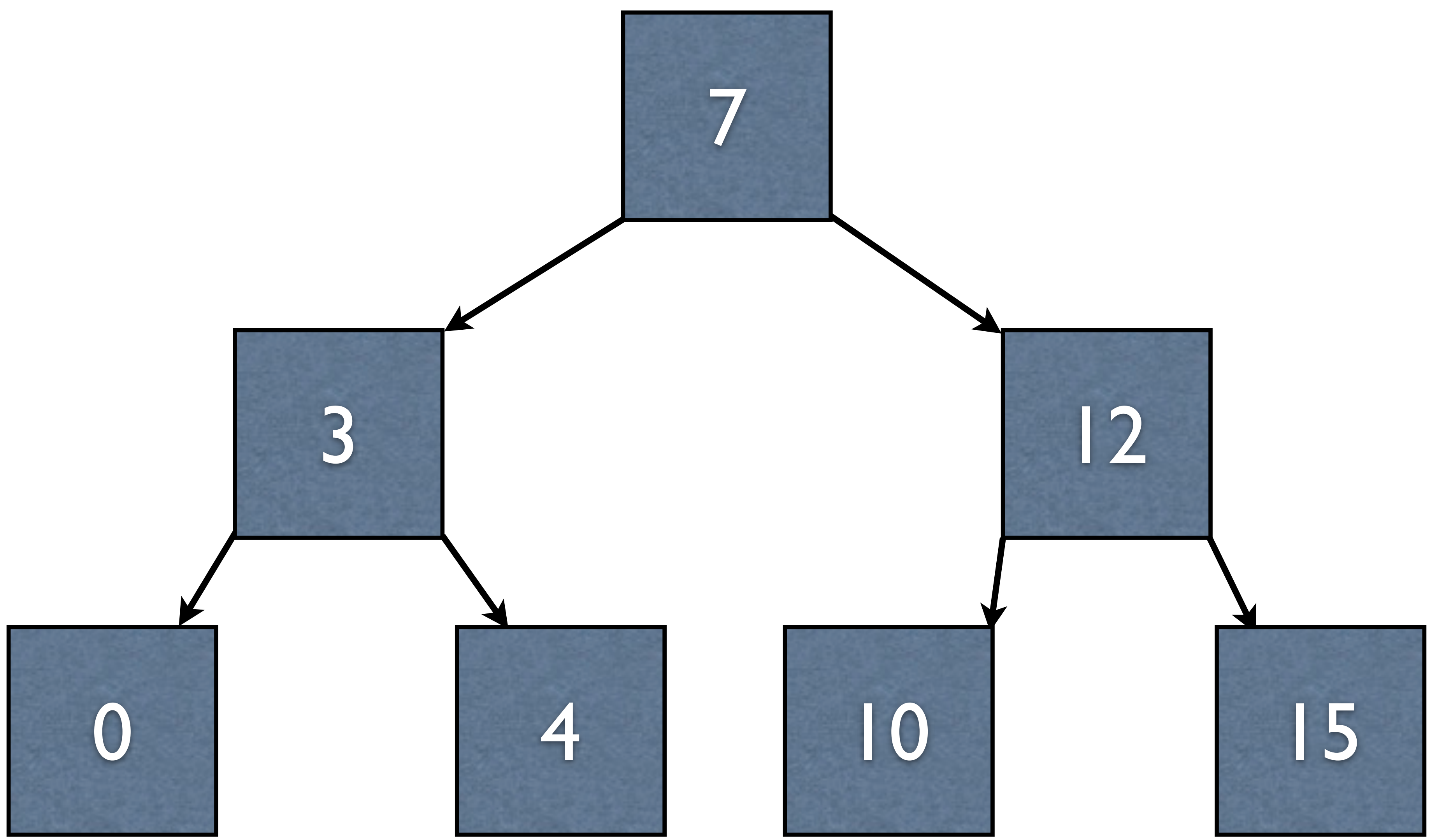
Using Traversals

- Say we want to print out the contents of a binary search tree in sorted order
- What kind of traversal should we use?



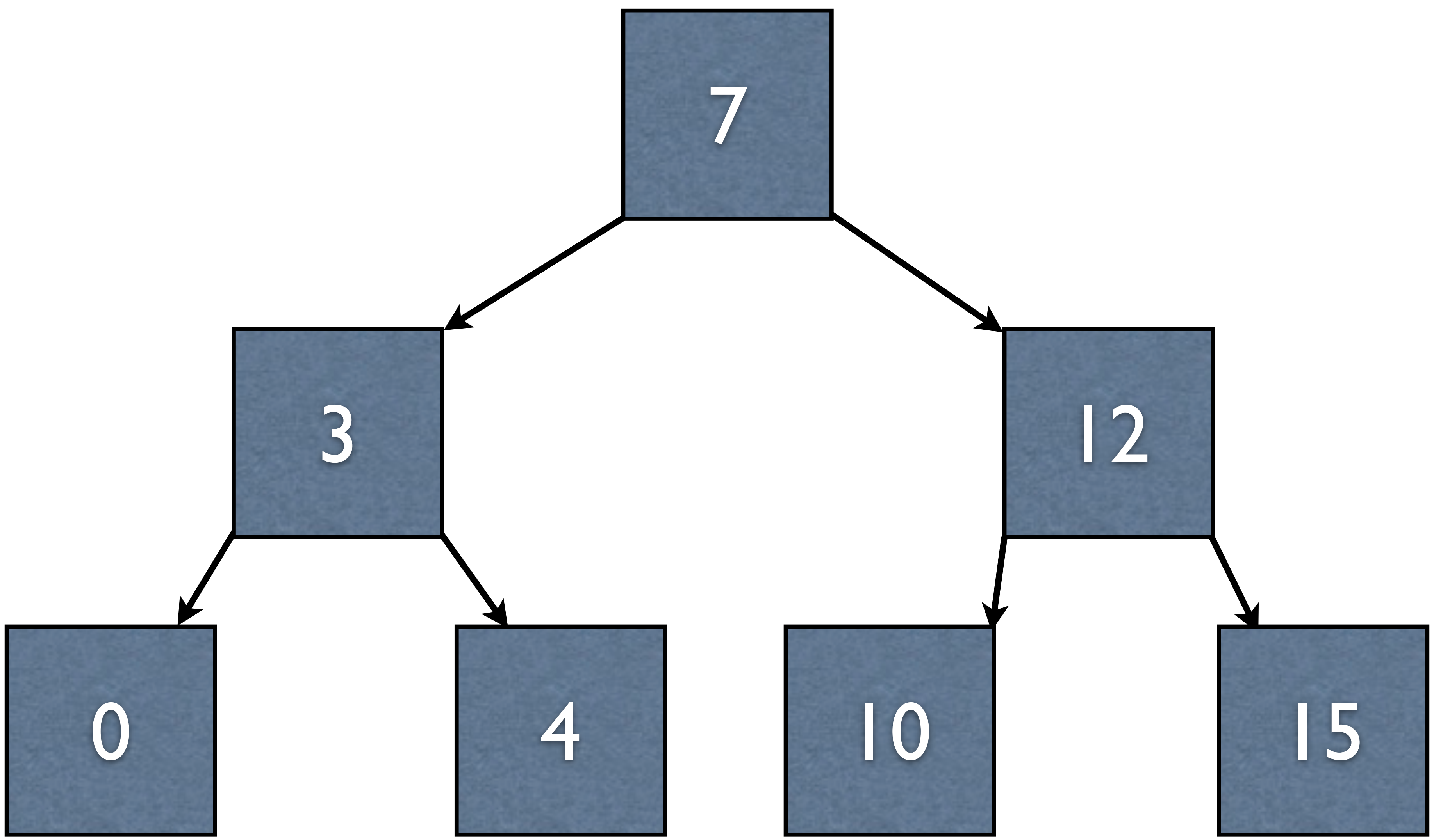
Using Traversals

- Say we want to print out the contents of a binary search tree in sorted order
- What kind of traversal should we use? - in-order



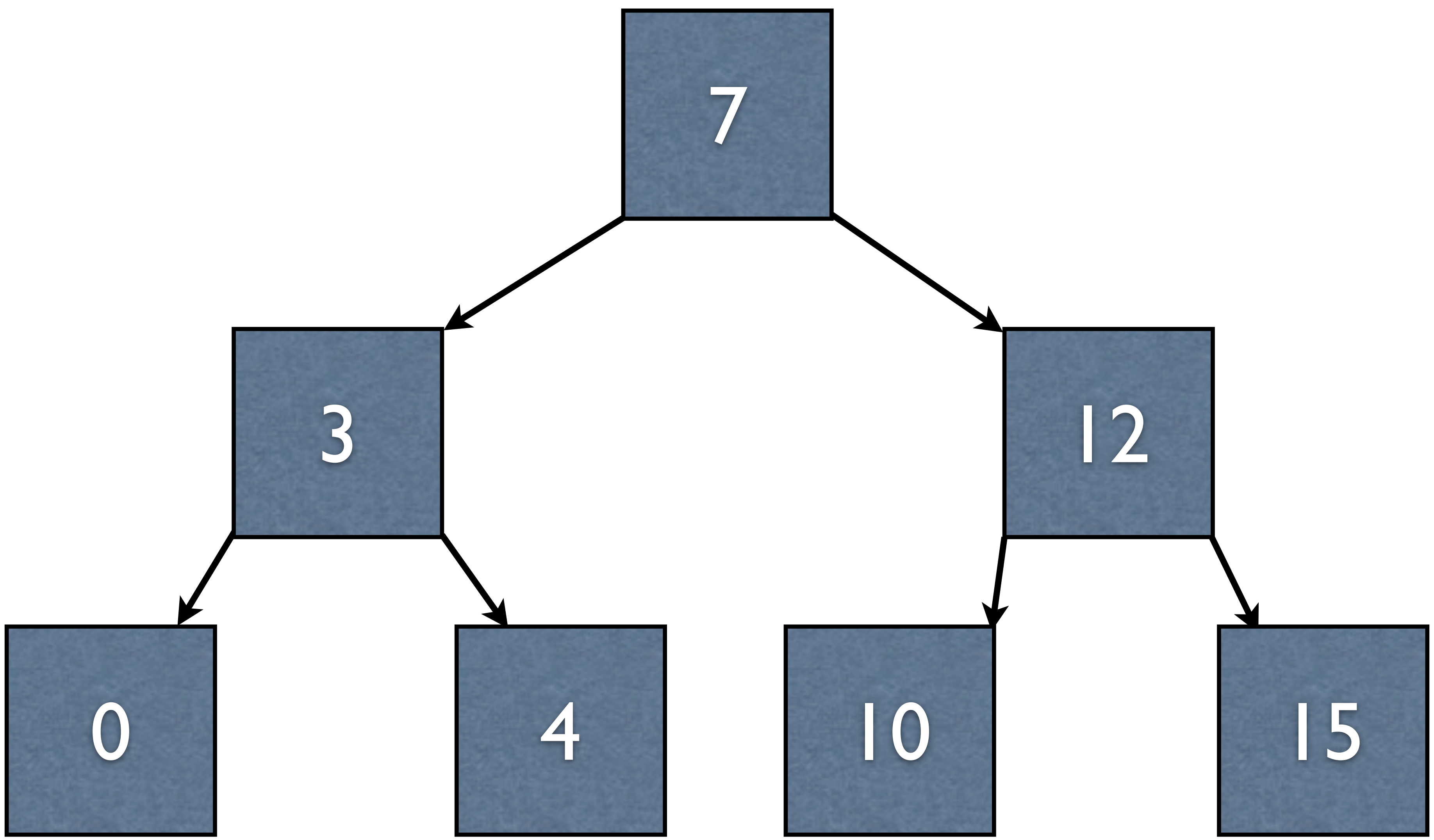
Using Traversals

- Say we want to delete a binary search tree
- Which traversal is best?



Using Traversals

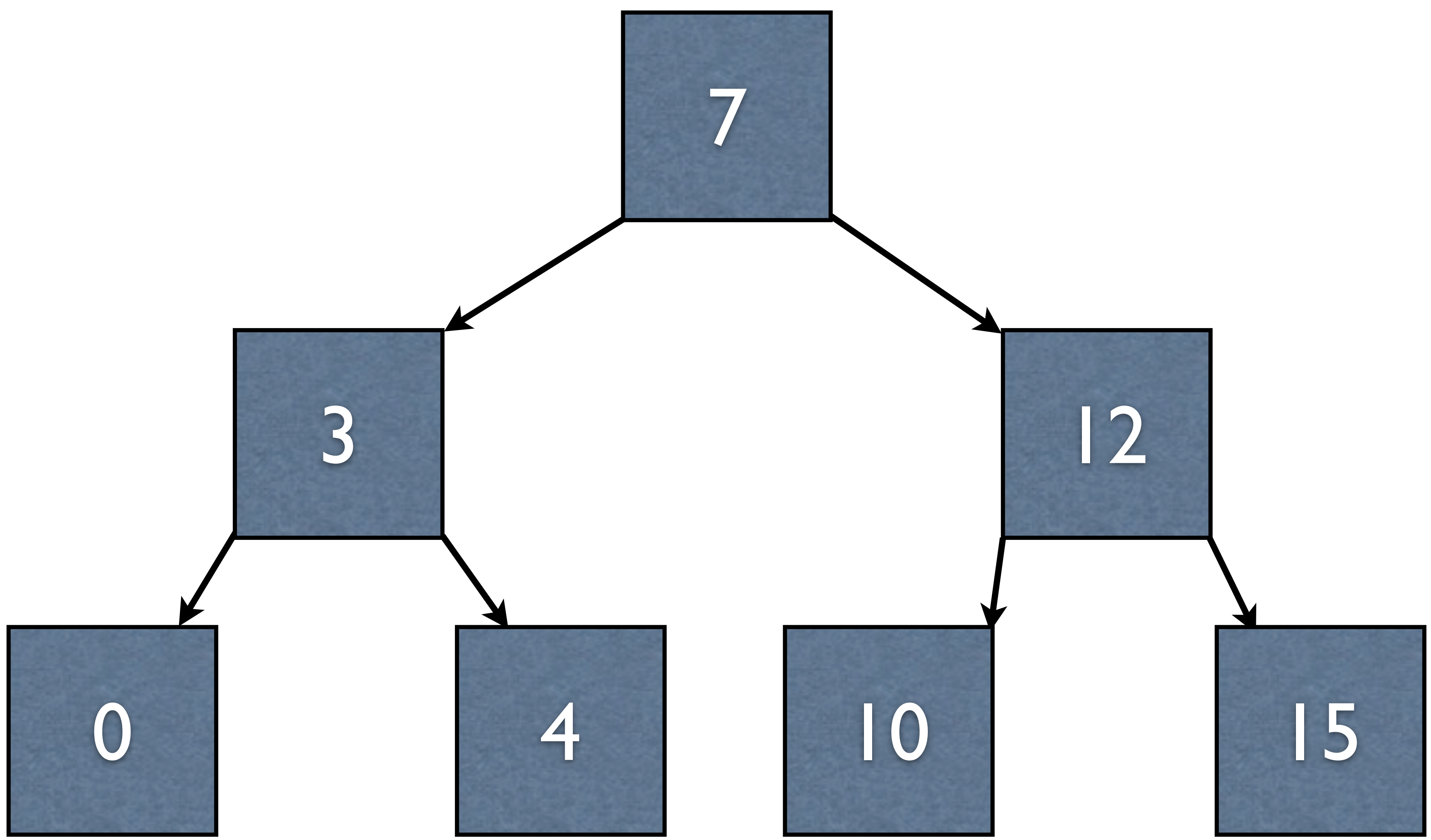
- Say we want to delete a binary search tree
- Which traversal is best? - post-order



Removing BST Elements

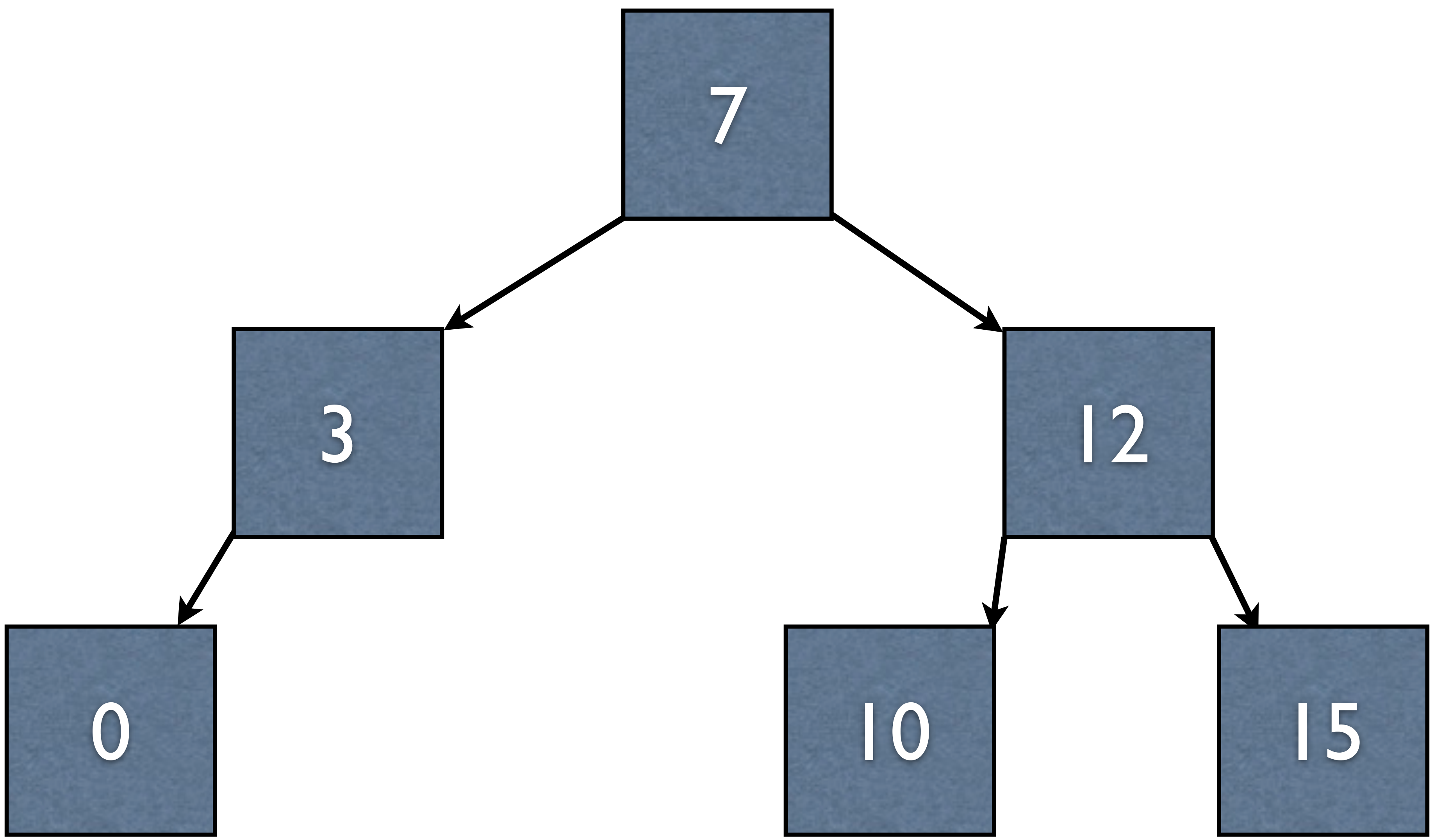
Removing Elements

- Say we want to remove 4. Any problems?



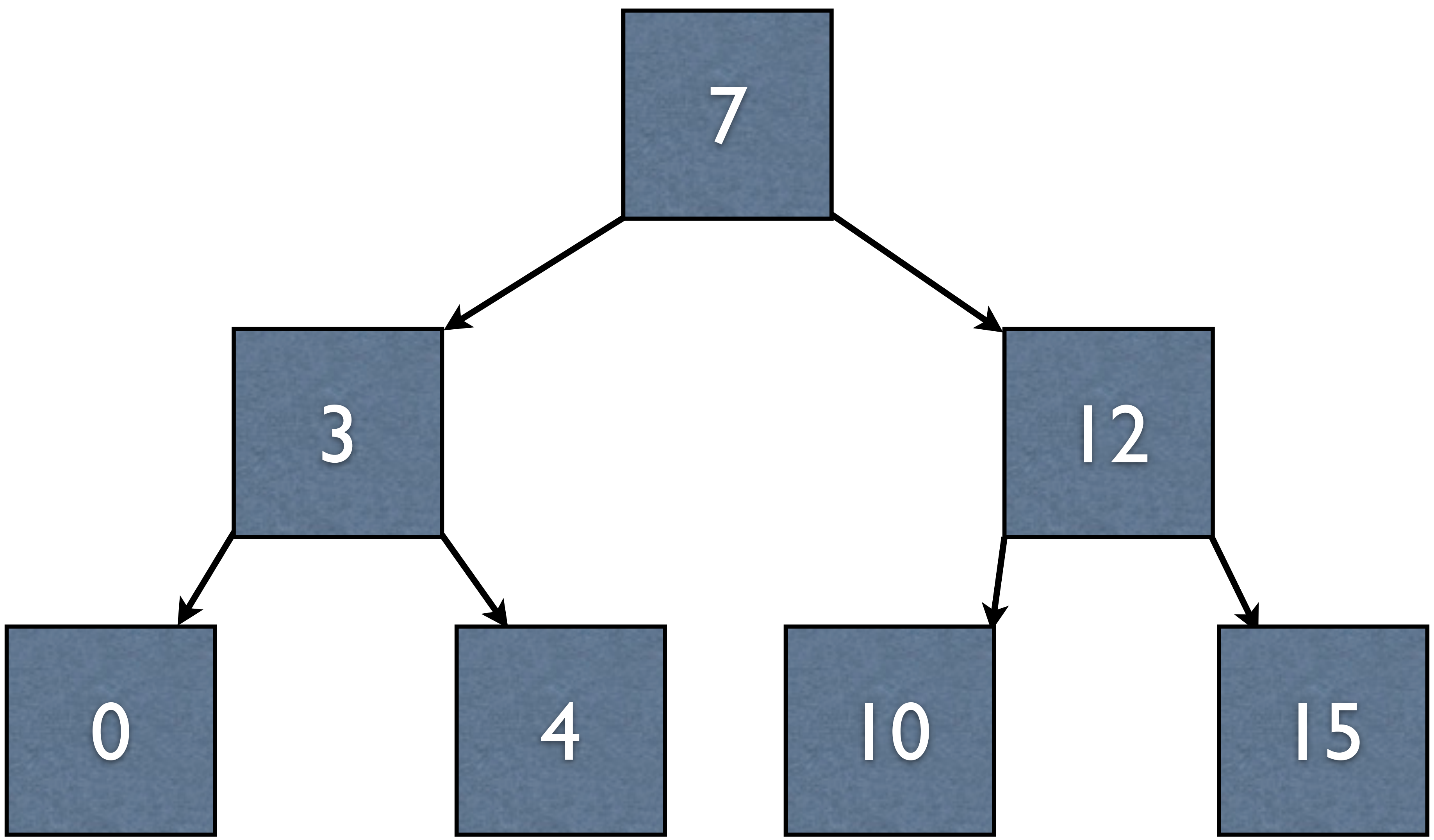
Removing Elements

- Say we want to remove 4. Any problems? - no



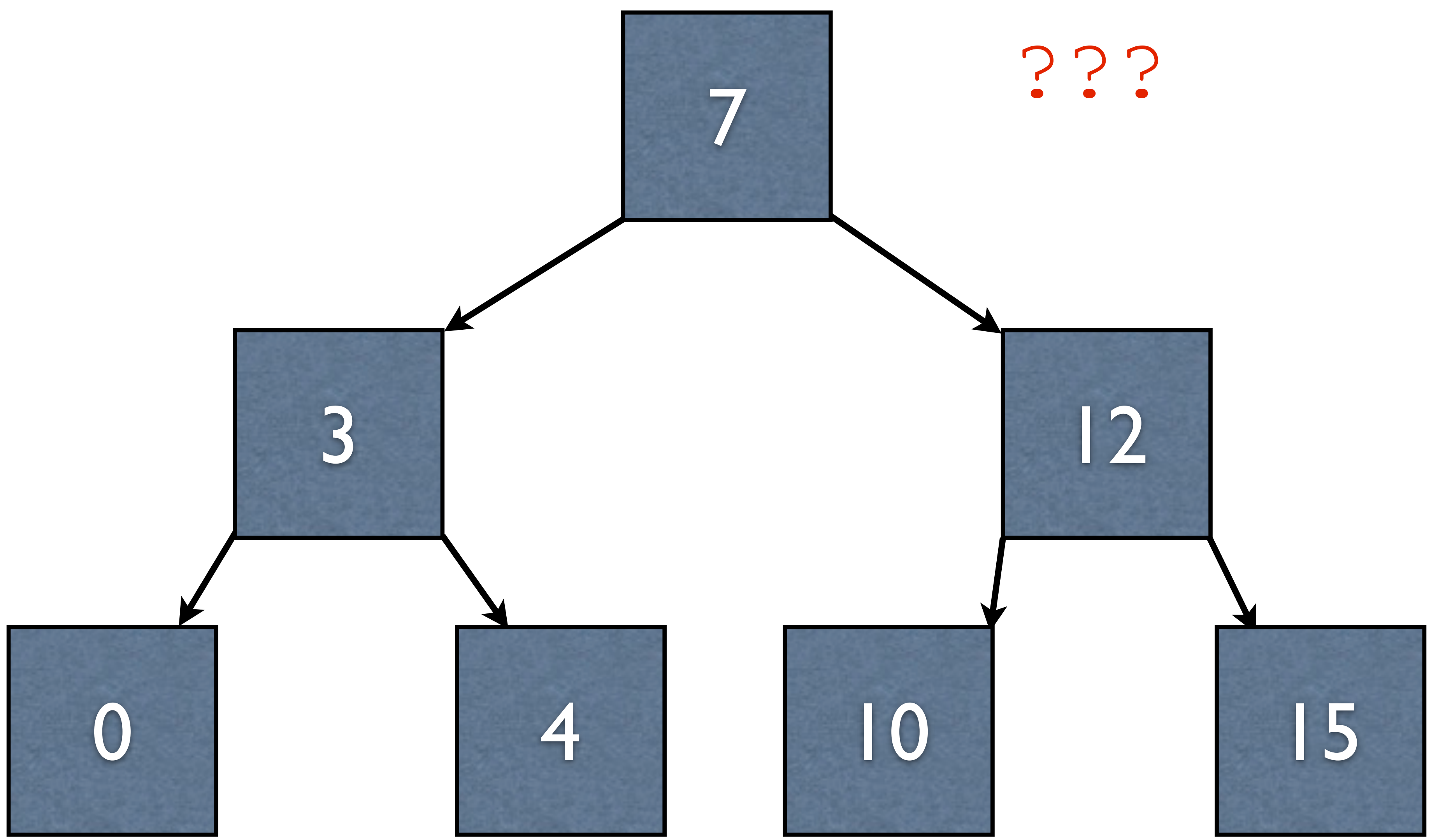
Removing Elements

- Say we want to remove 7 - any problems?



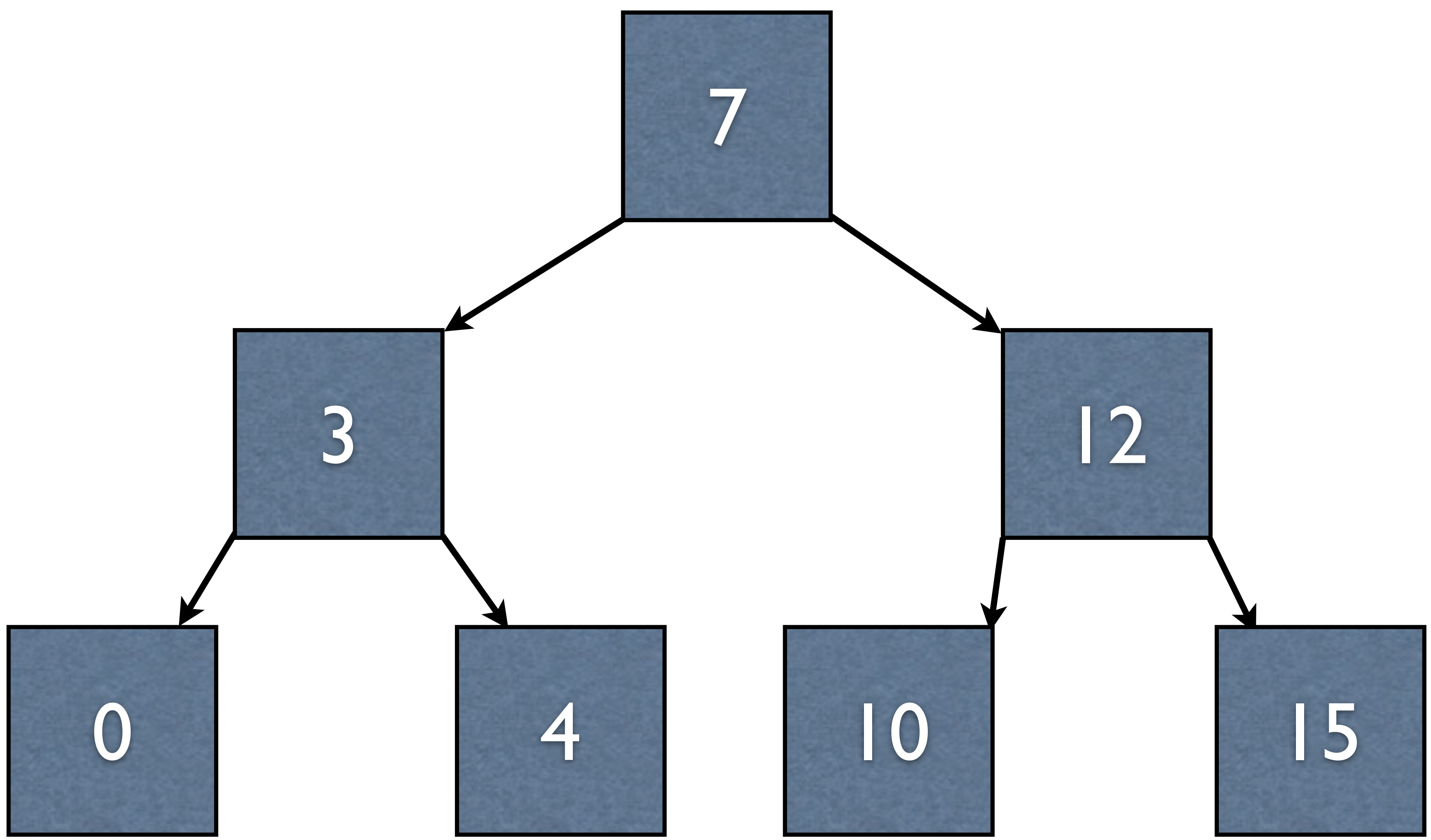
Removing Elements

- Say we want to remove 7 - any problems?
  - Both 3 and 12 cannot be a root



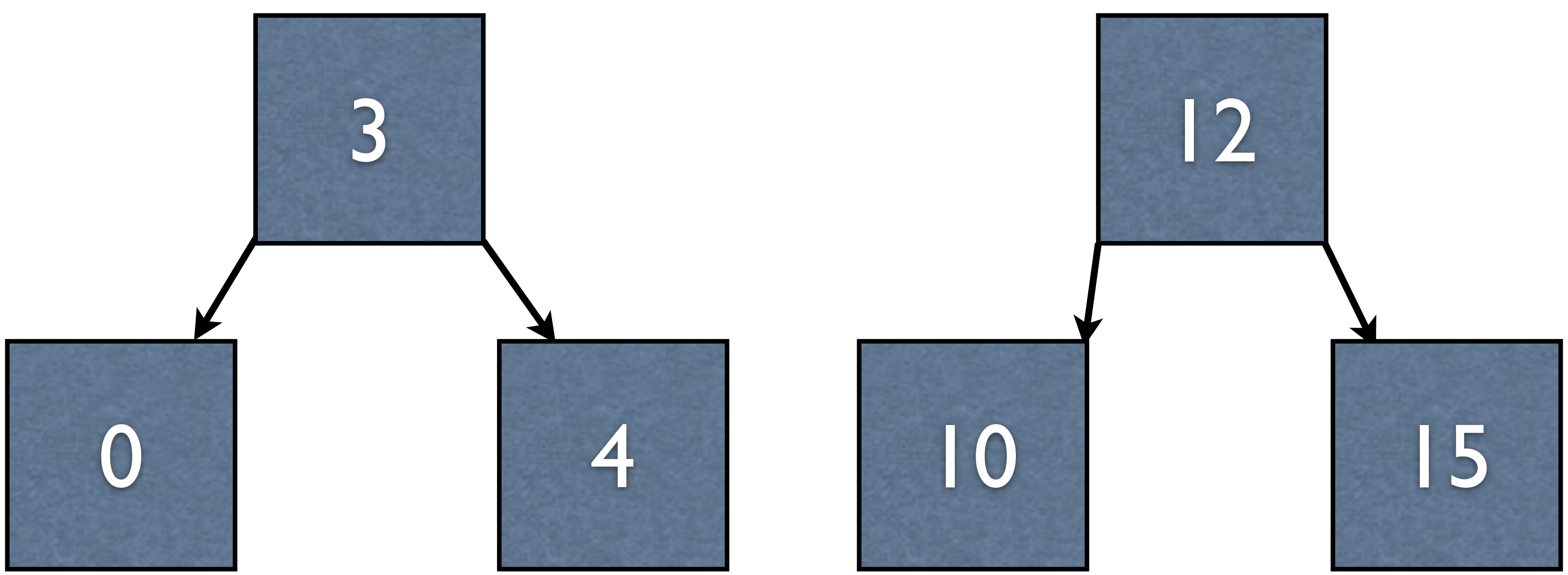
Removing Elements

- Removing 7
- Let's try making 12 a root...



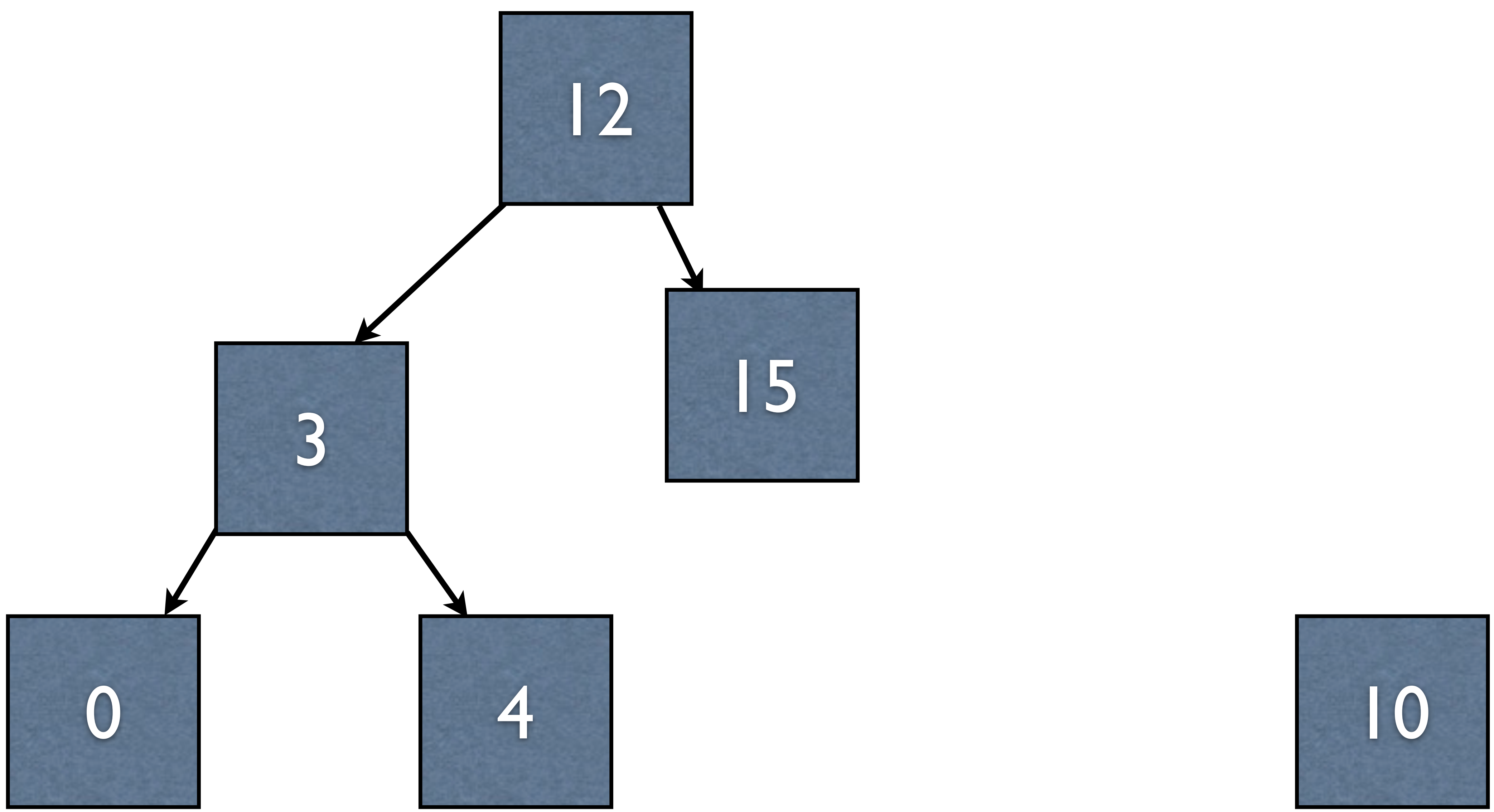
Removing Elements

- Removing 7
- Let's try making 12 a root...



Removing Elements

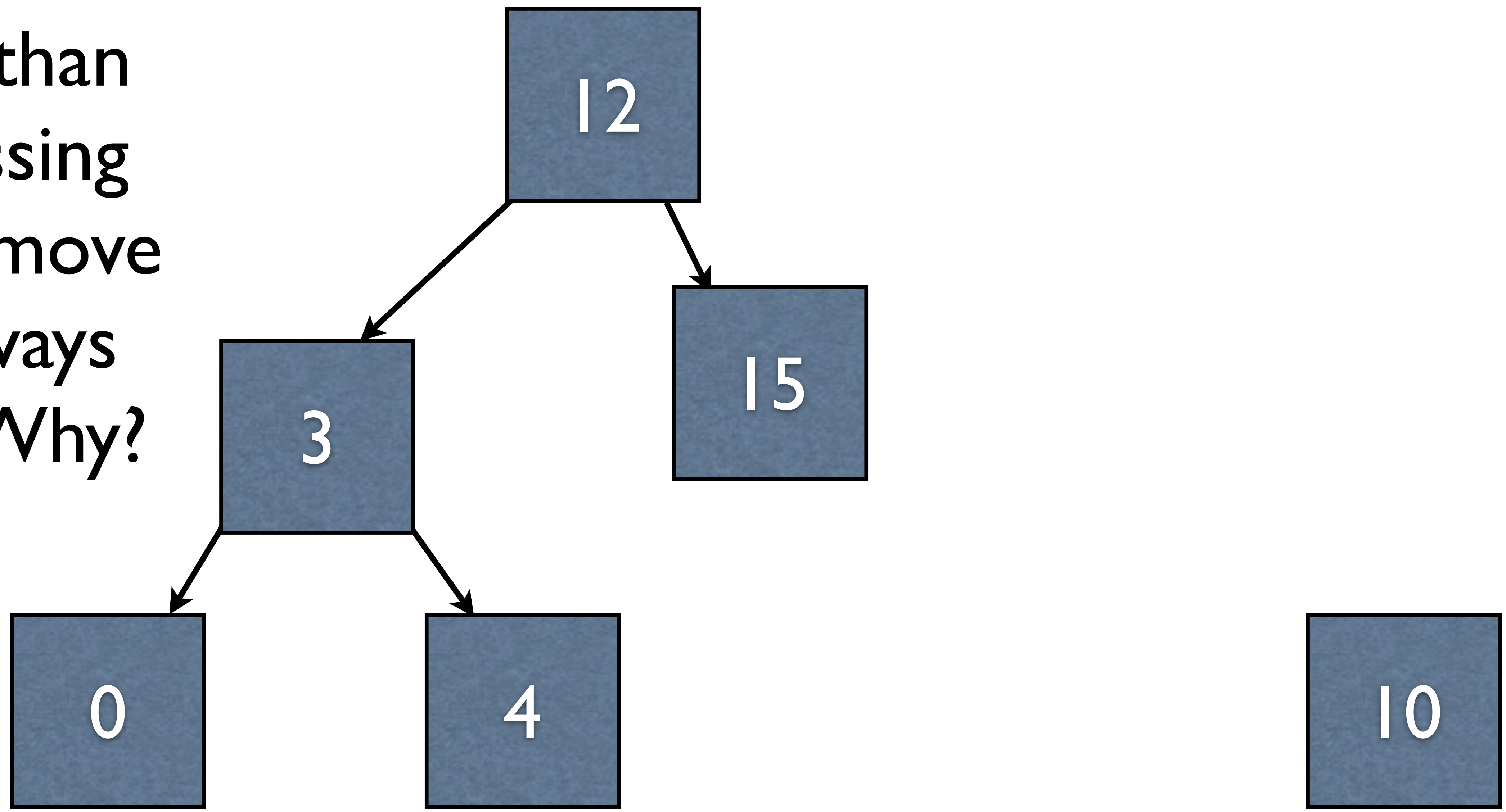
- Removing 7
- Let's try making 12 a root...



Removing Elements

- Removing 7
- Let's try making 12 a root...

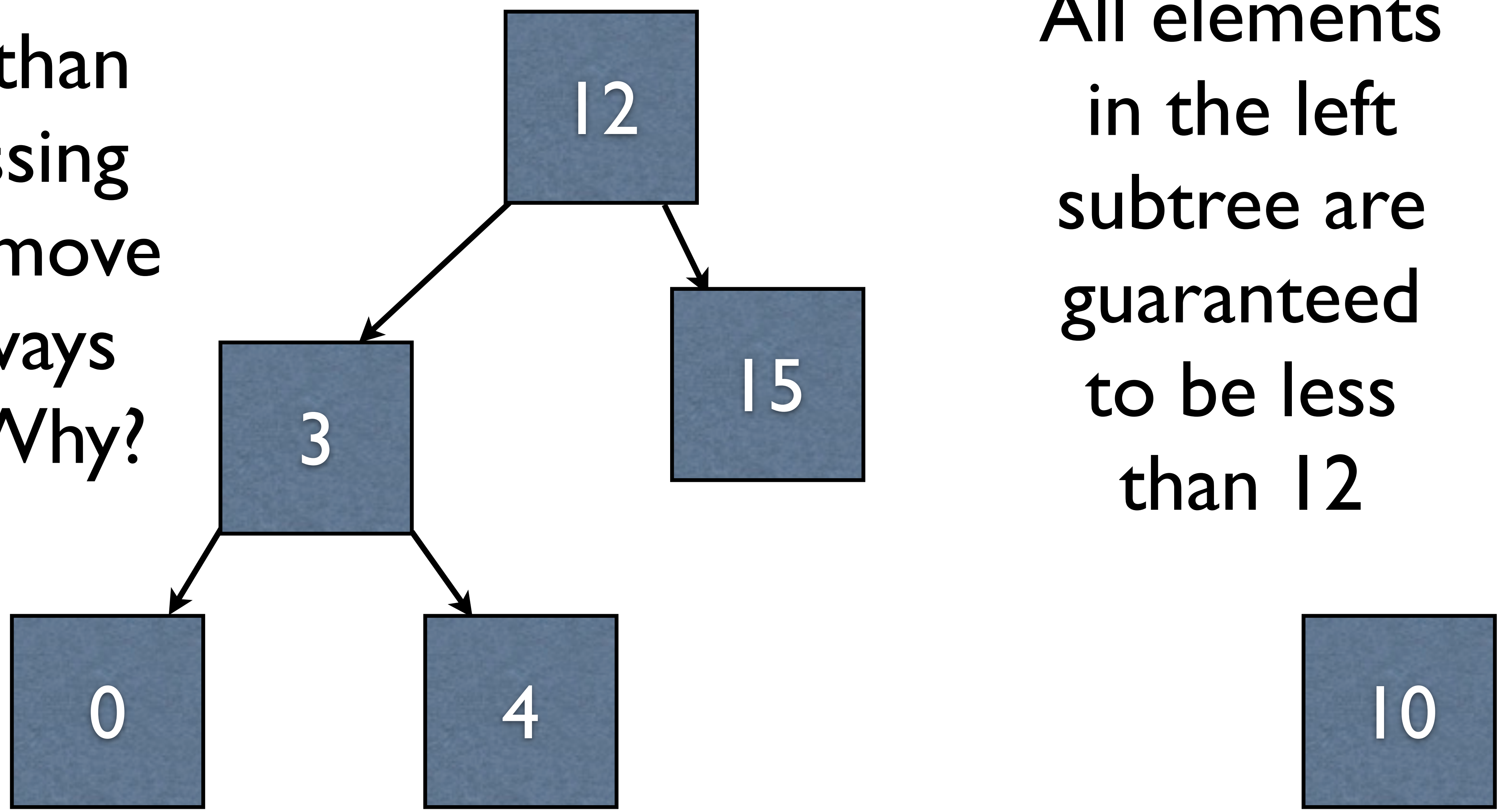
Other than the missing 10, this move will always work. Why?



Removing Elements

- Removing 7
- Let's try making 12 a root...

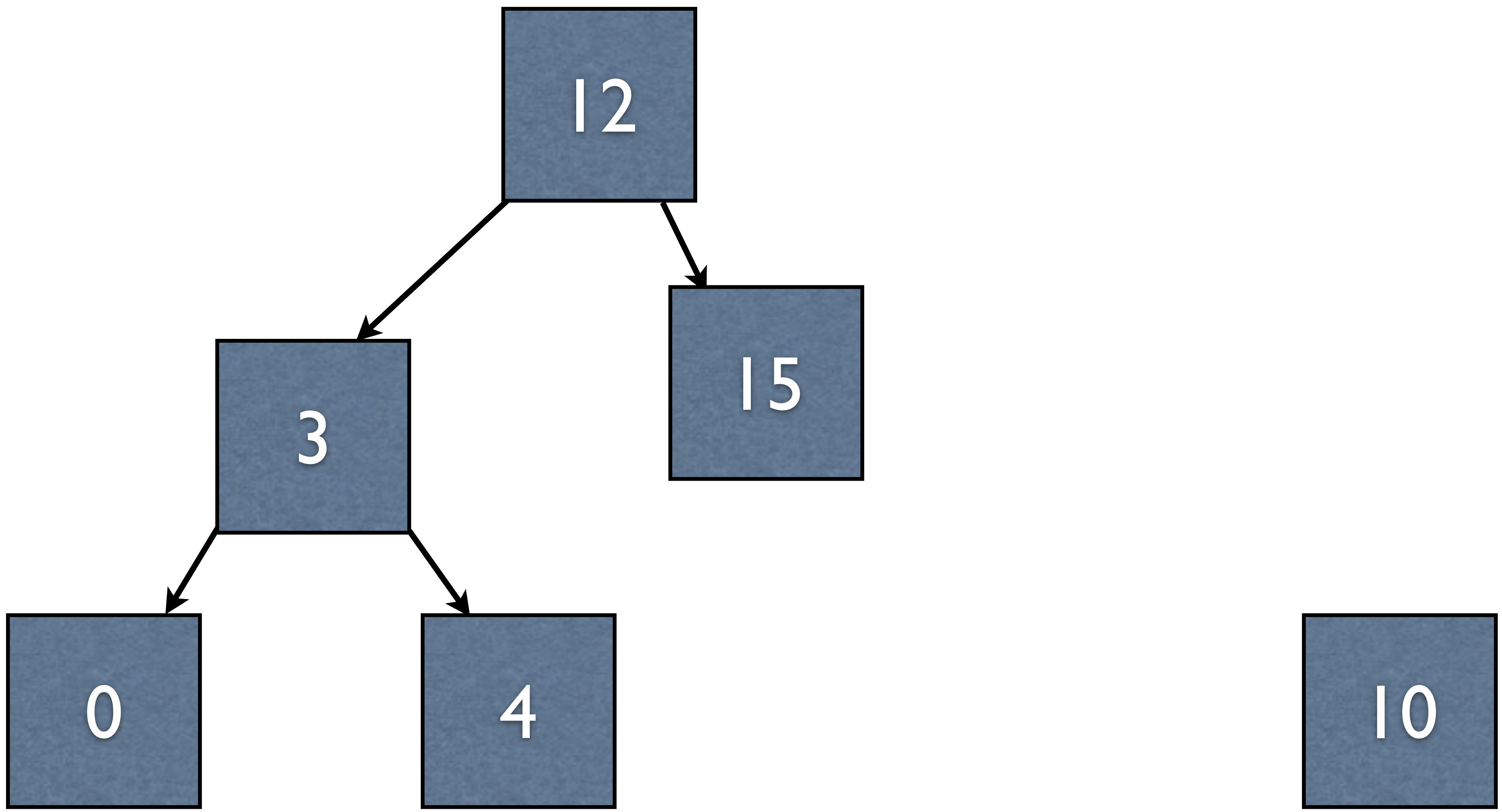
Other than the missing 10, this move will always work. Why?



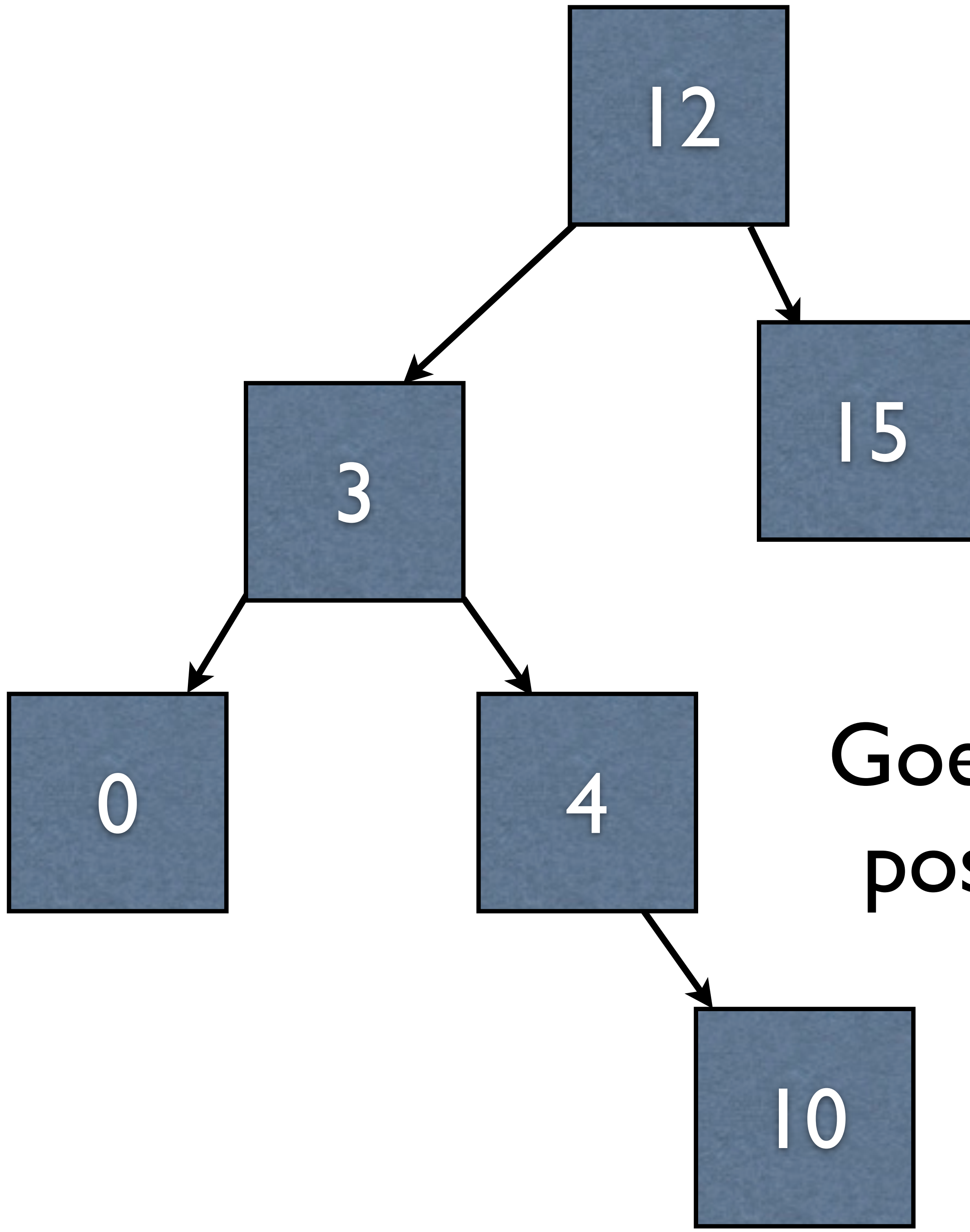
All elements in the left subtree are guaranteed to be less than 12

Removing Elements

- Now we need to put 10 back
- 10 could be an arbitrarily deep subtree
- Always goes into the same position - where?



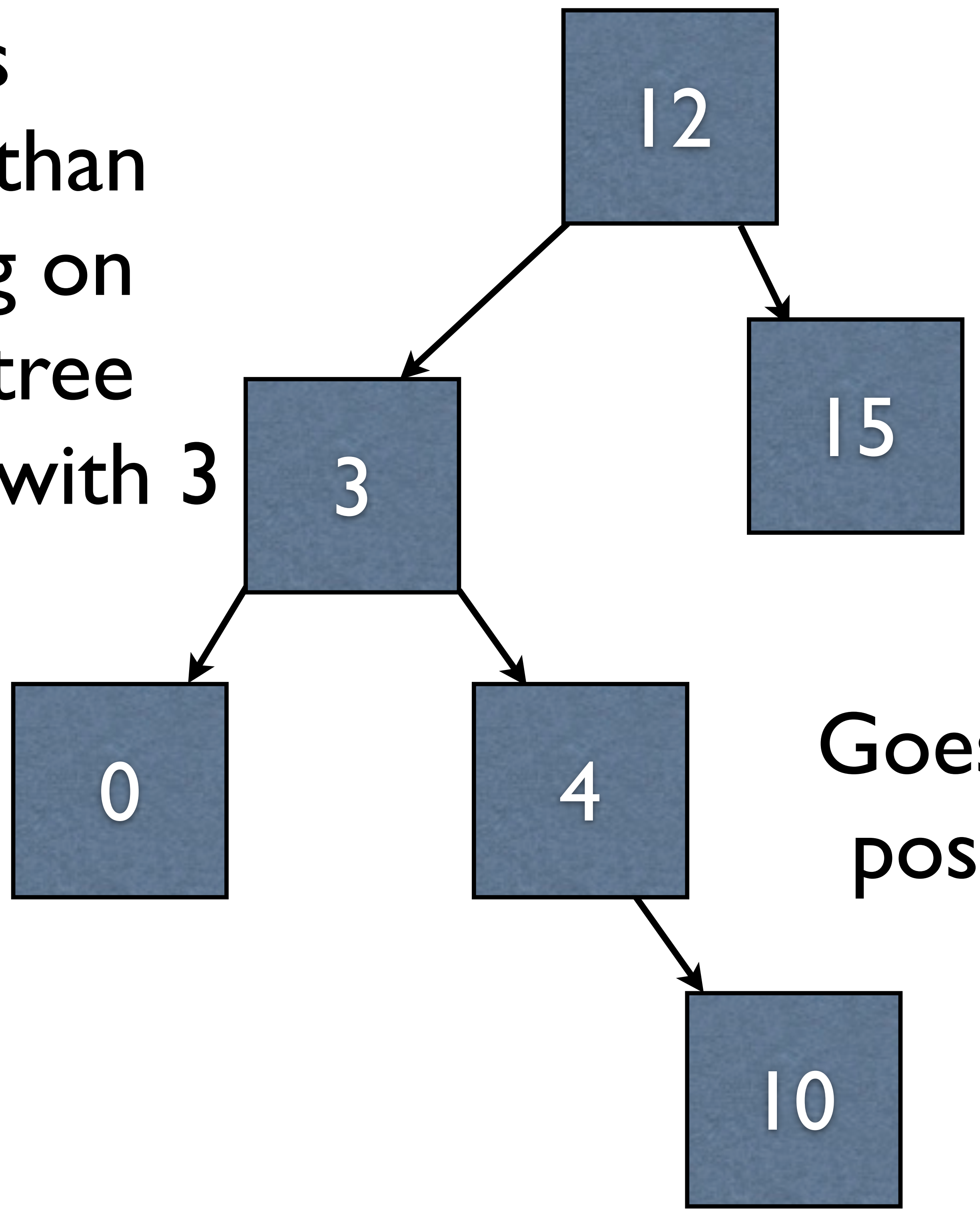
Removing Elements



Goes to the rightmost position here always. Why?

Removing Elements

Guaranteed that 10 is greater than anything on the subtree beginning with 3



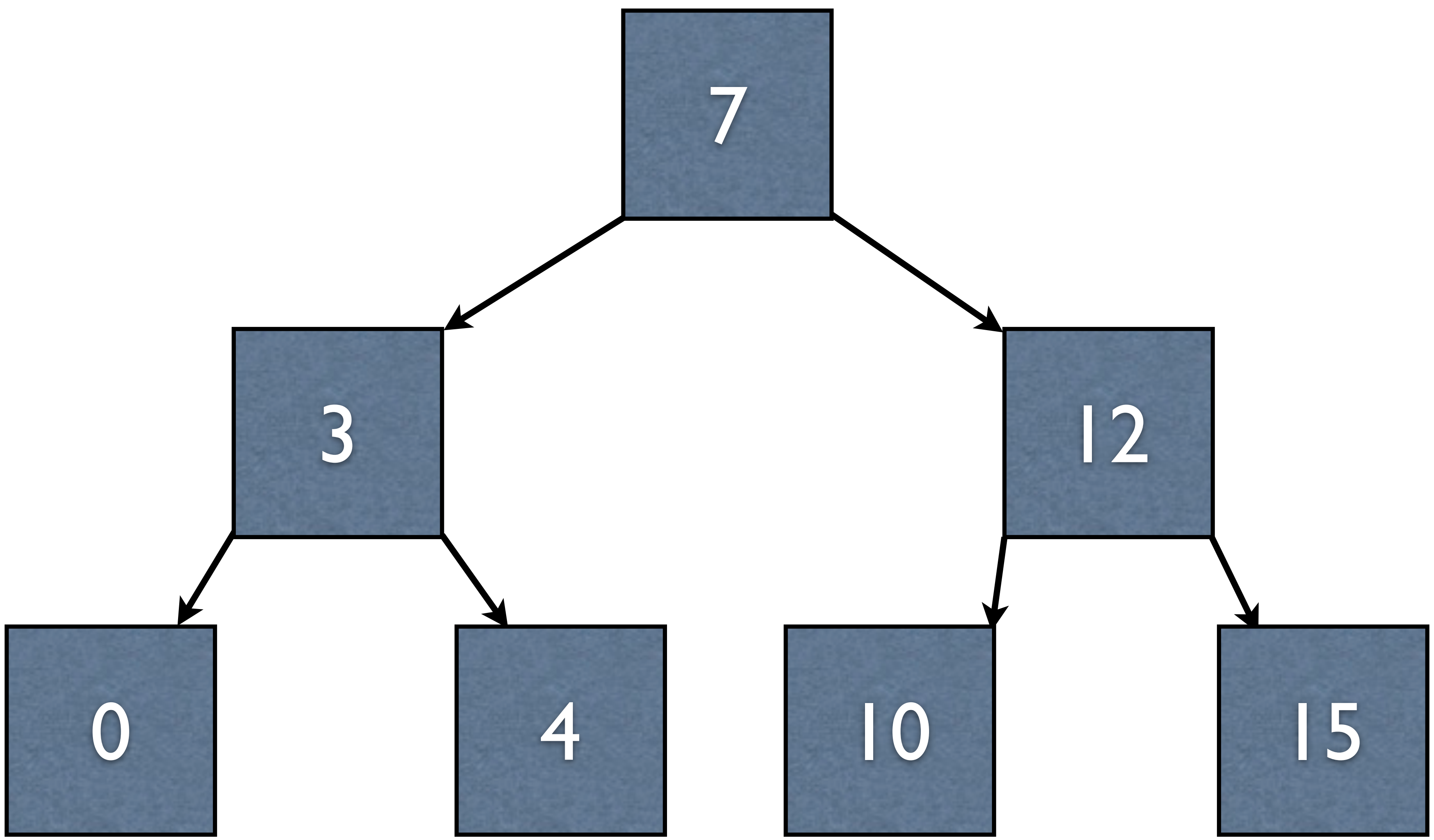
Goes to the rightmost position here always. Why?

Deletion Issues

- Algorithm described prior is somewhat tricky to implement, and easily leads to unbalanced trees
- A better strategy follows

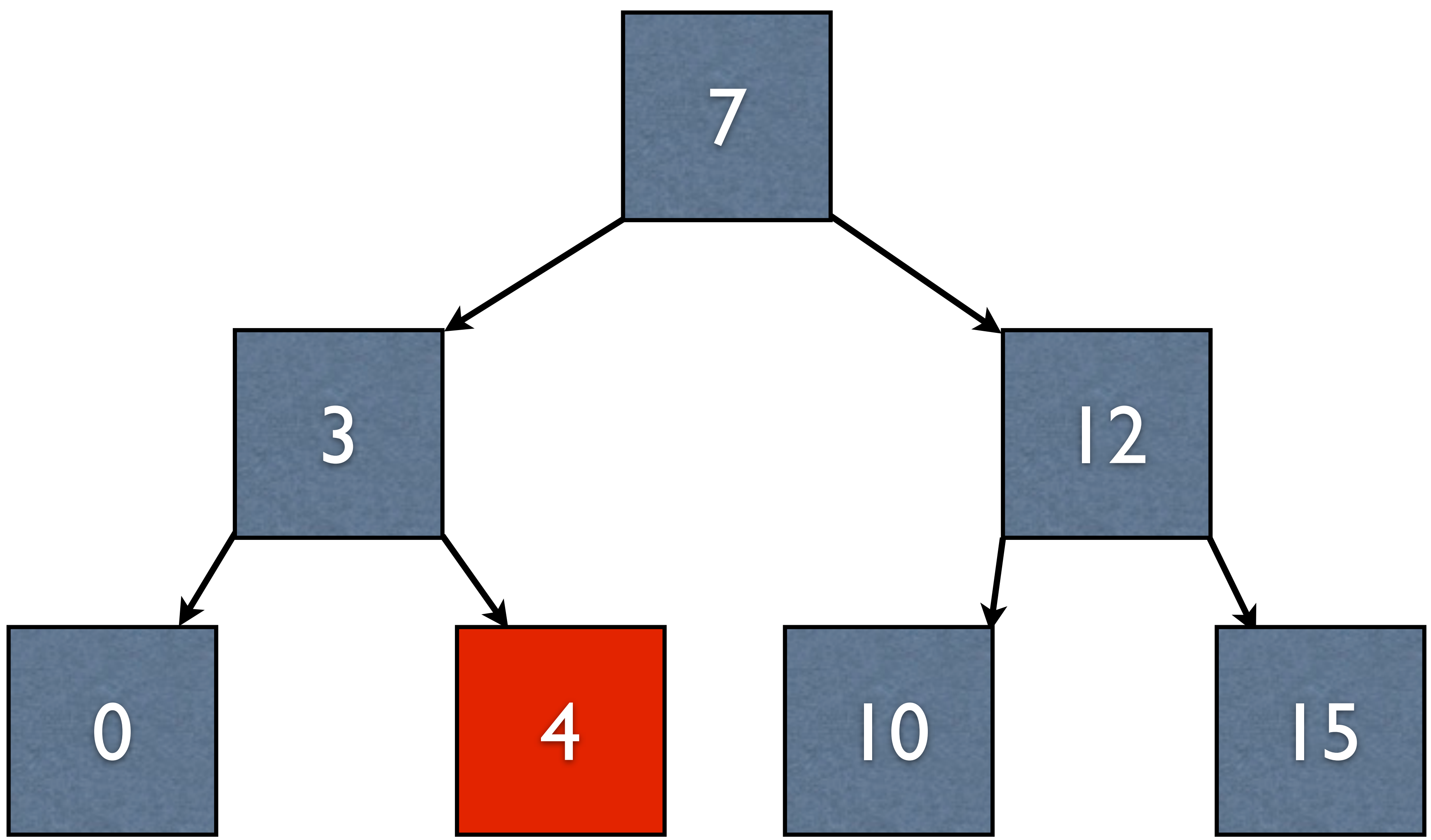
Alternative

- Deleting 7



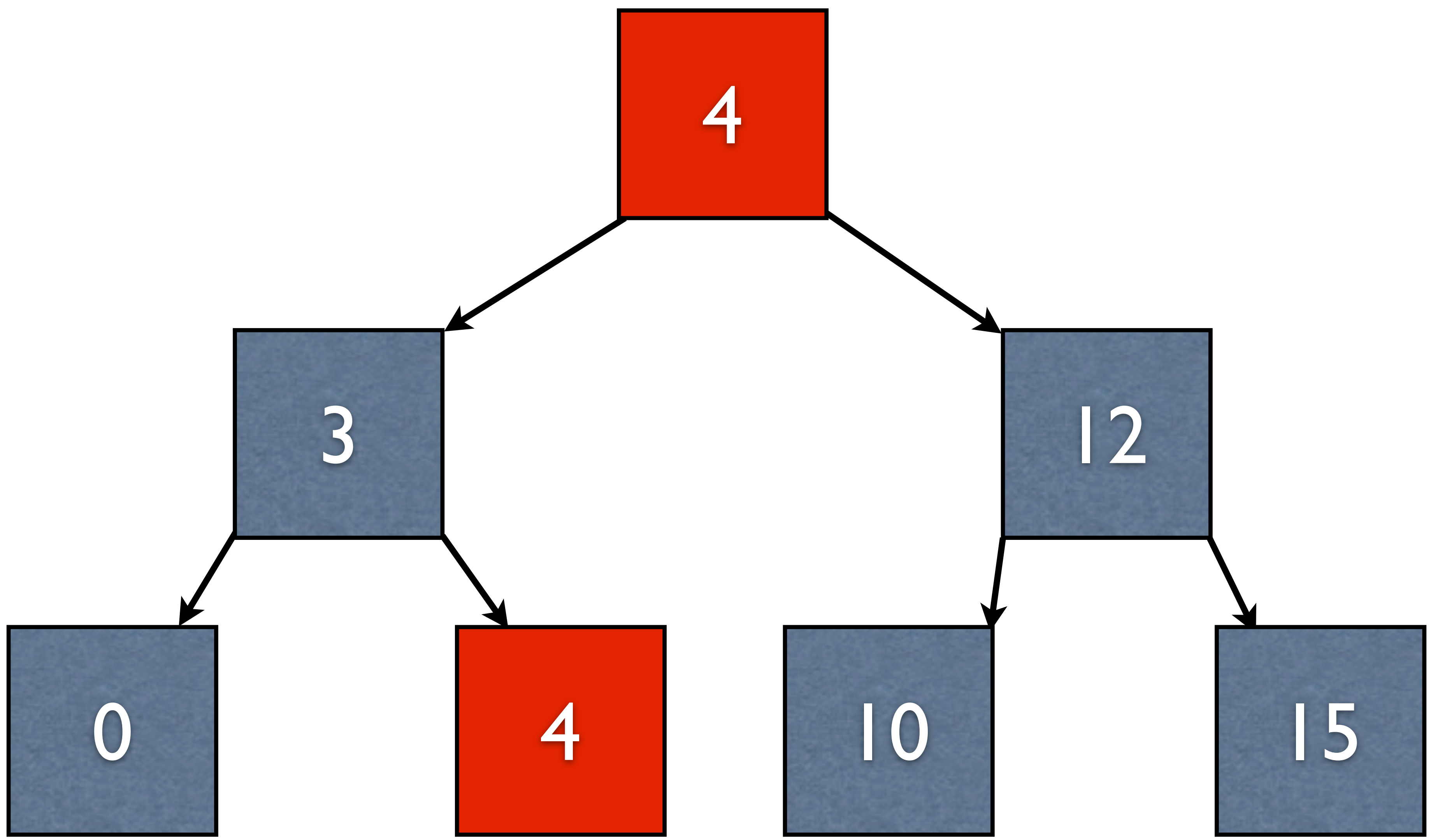
Alternative

- Get the greatest node less than 7 (always on far left subtree)



Alternative

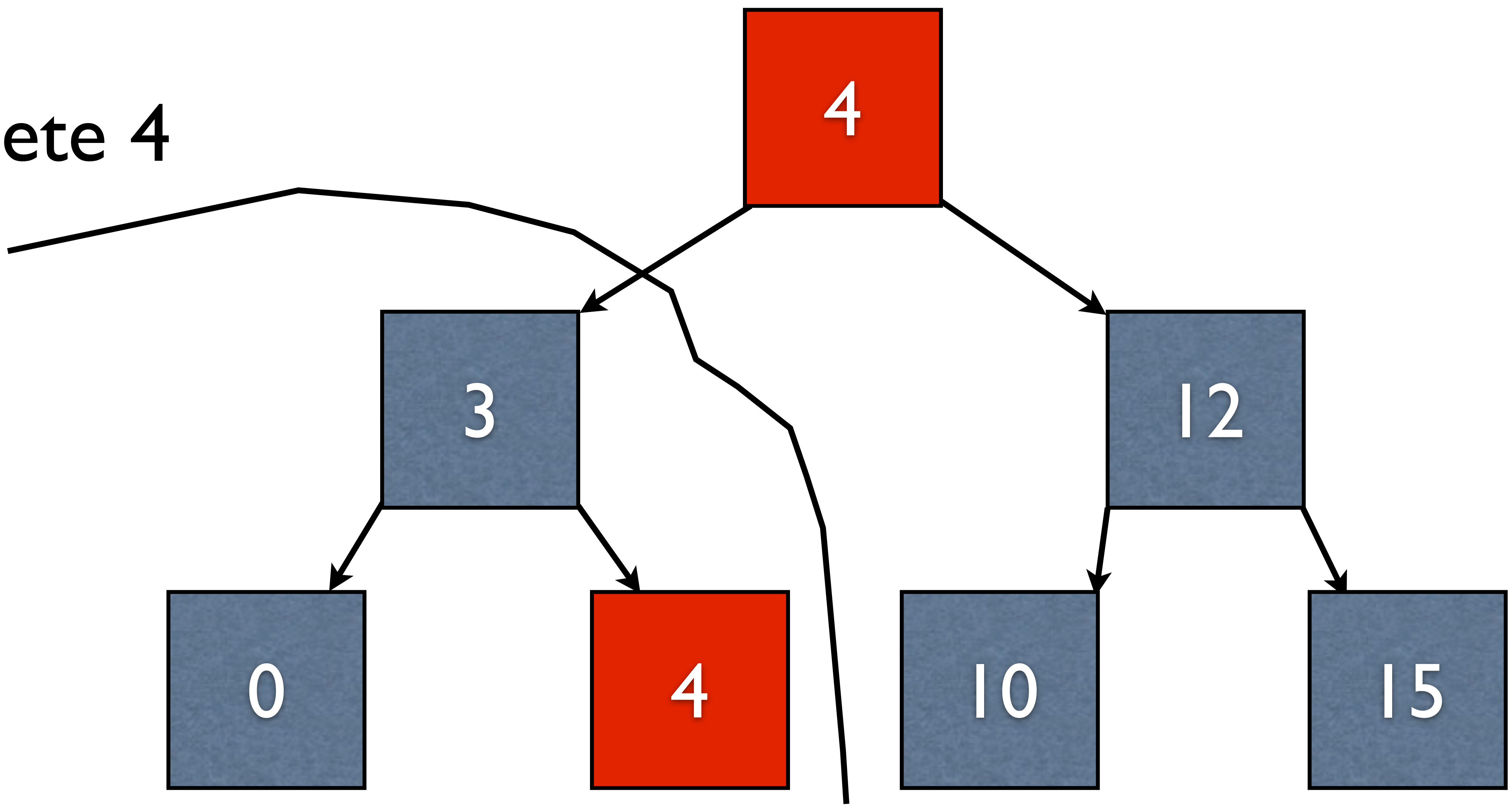
- Copy its value to the node being deleted



Alternative

- Recursively delete the copied element from the left subtree

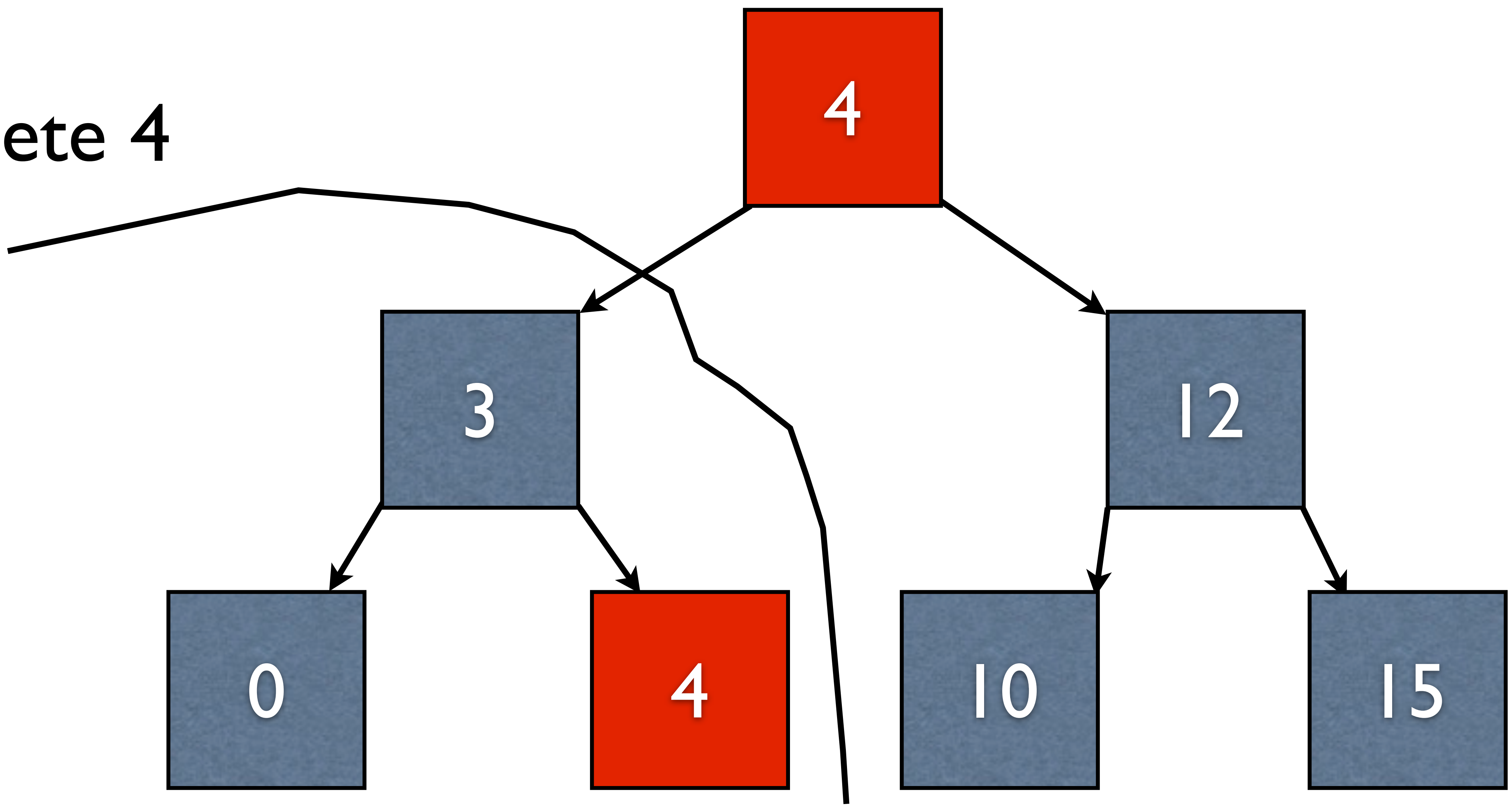
delete 4



Alternative

- We are guaranteed to eventually reach a leaf node (a base case)

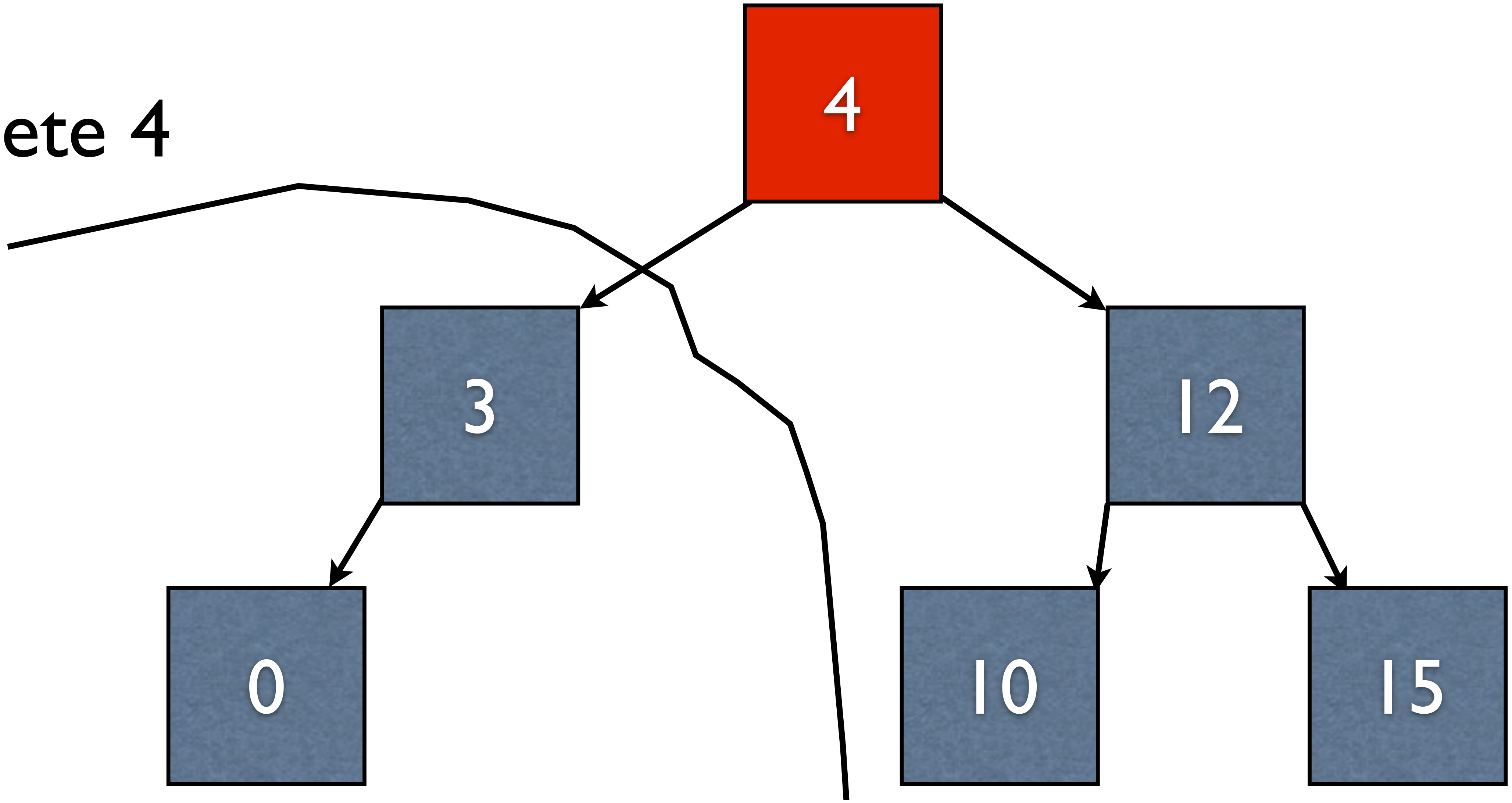
delete 4



Alternative

- We are guaranteed to eventually reach a leaf node (a base case)

delete 4



Priority Queues

Motivation

- Consider a hospital emergency room
- Three patients arrive with specific problems in the following order:
  - Minor cough
  - Light skin irritation
  - Anaphylactic shock
- How can we prioritize them?

Prioritization

- Stack makes no sense in general (whoever gets there last always gets treatment first)
- Queue makes some sense (get treatment in order of arrival)
  - Not good for life-threatening situations
- Need a new data structure to handle this

Priority Queue

- Like a queue, but elements are associated with a given priority
- We always want to dequeue the highest priority element
- How might we implement this?

Implementation #1

- Use a simple linked list
- On dequeue, remove the element from the list with the highest priority
  - Enqueue time complexity?
  - Dequeue time complexity?

Implementation #1

- Use a simple linked list
- On dequeue, remove the element from the list with the highest priority
    - Enqueue time complexity? - `O(1)`
    - Dequeue time complexity? - `O(N)`

Implementation #2

- Using a linked list, keep elements in descending sorted order
- Always dequeue from the front
  - Enqueue time complexity?
  - Dequeue time complexity?

Implementation #2

- Using a linked list, keep elements in descending sorted order
- Always dequeue from the front
  - Enqueue time complexity? - `O(N)`
  - Dequeue time complexity? - `O(1)`

Problems

- Somewhere we have an `O(N)` operation buried
- Any ideas for speeding this up?

Heaps

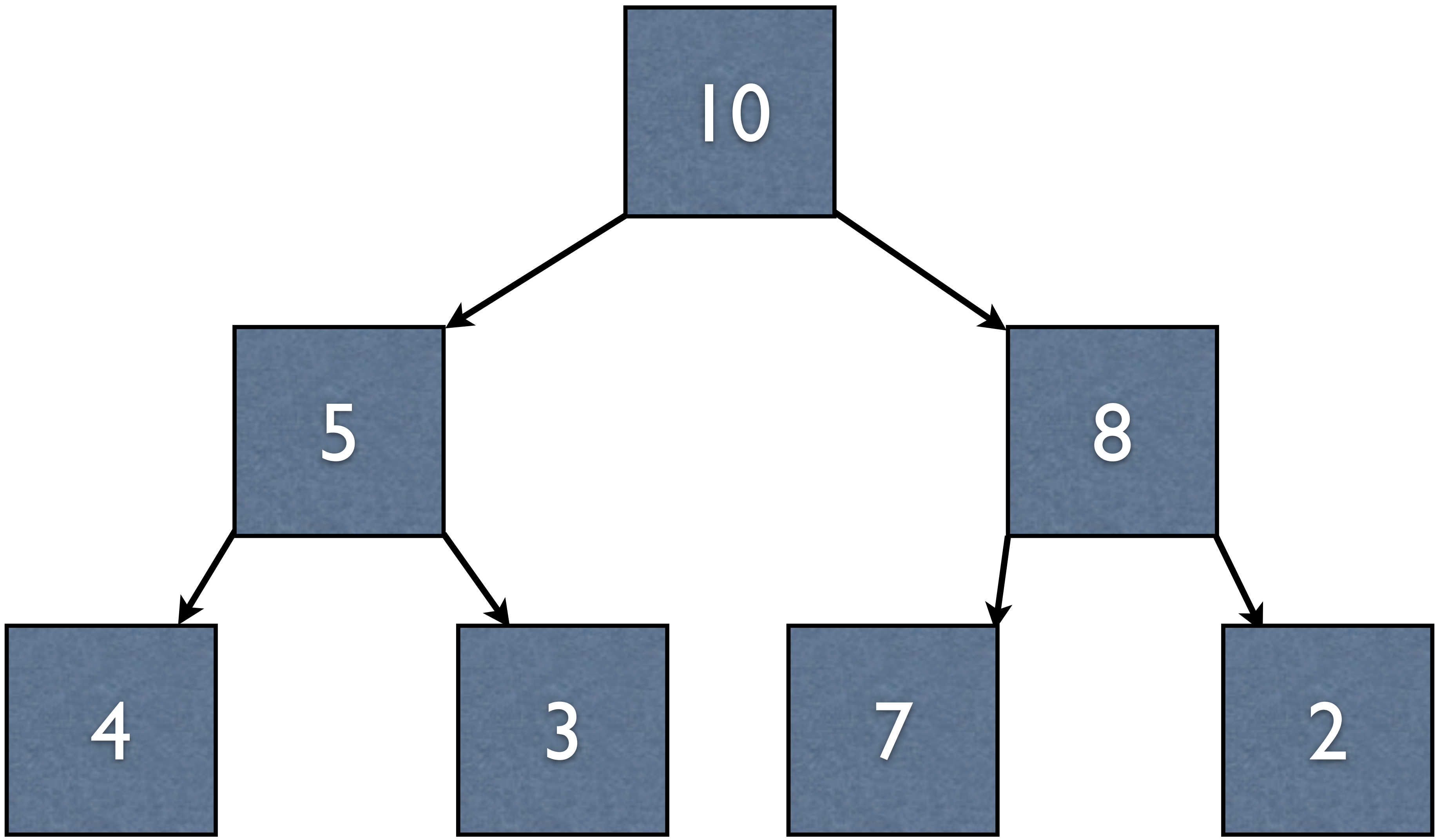
Heap

- **Not** a binary search tree; just a binary tree
- Always have the maximal (or minimal) element at the root
- Support removing the root element in `O(log(N))`, and adding elements in `O(log(N))`

Heap Property

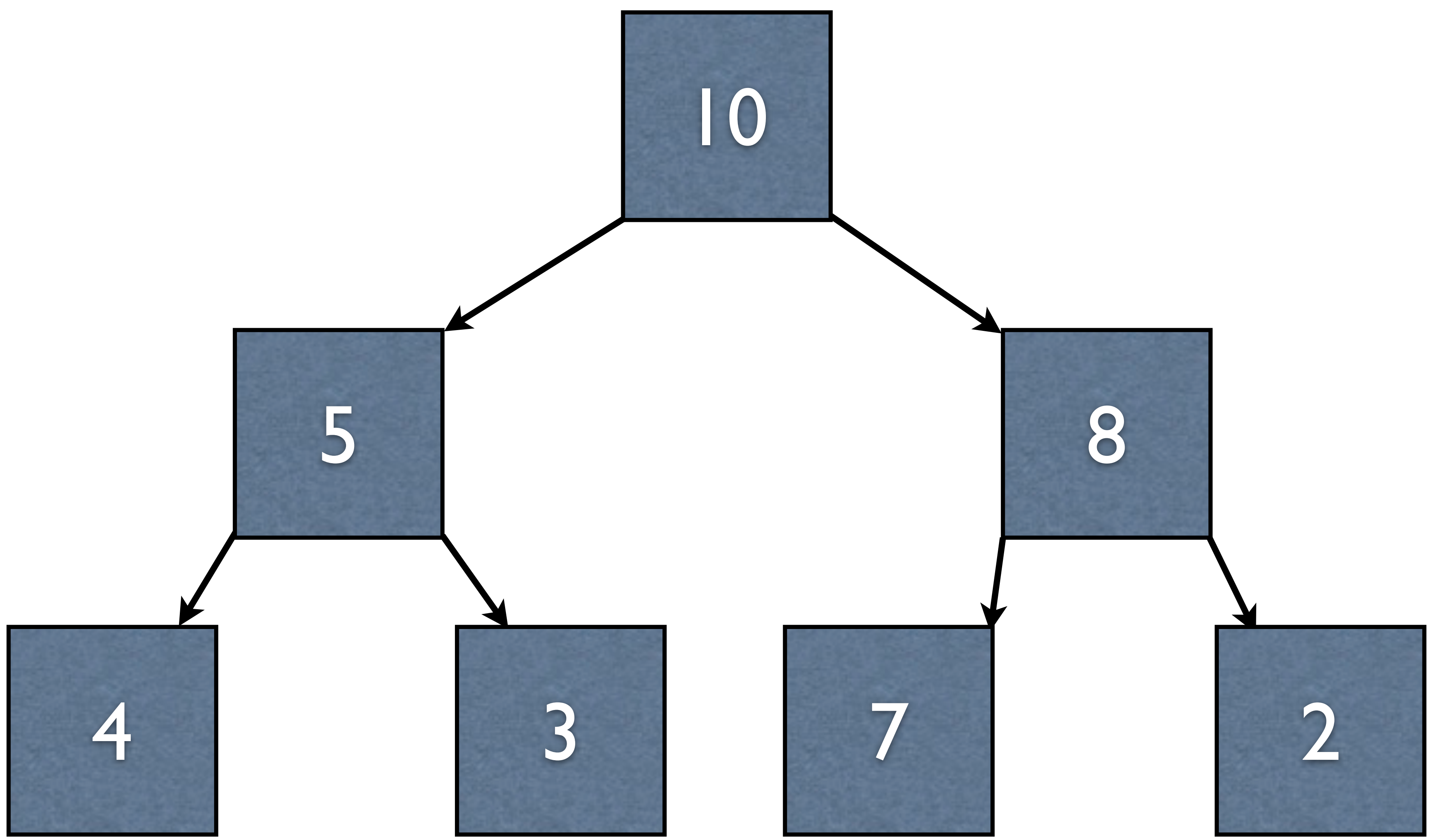
- A binary tree has the heap property if:
    - It is empty
    - Its value is greater than or equal to both of its children, and the children have the heap property

Example



Advantage

- Heaps always have the highest priority element on top, so we always have easy access to it

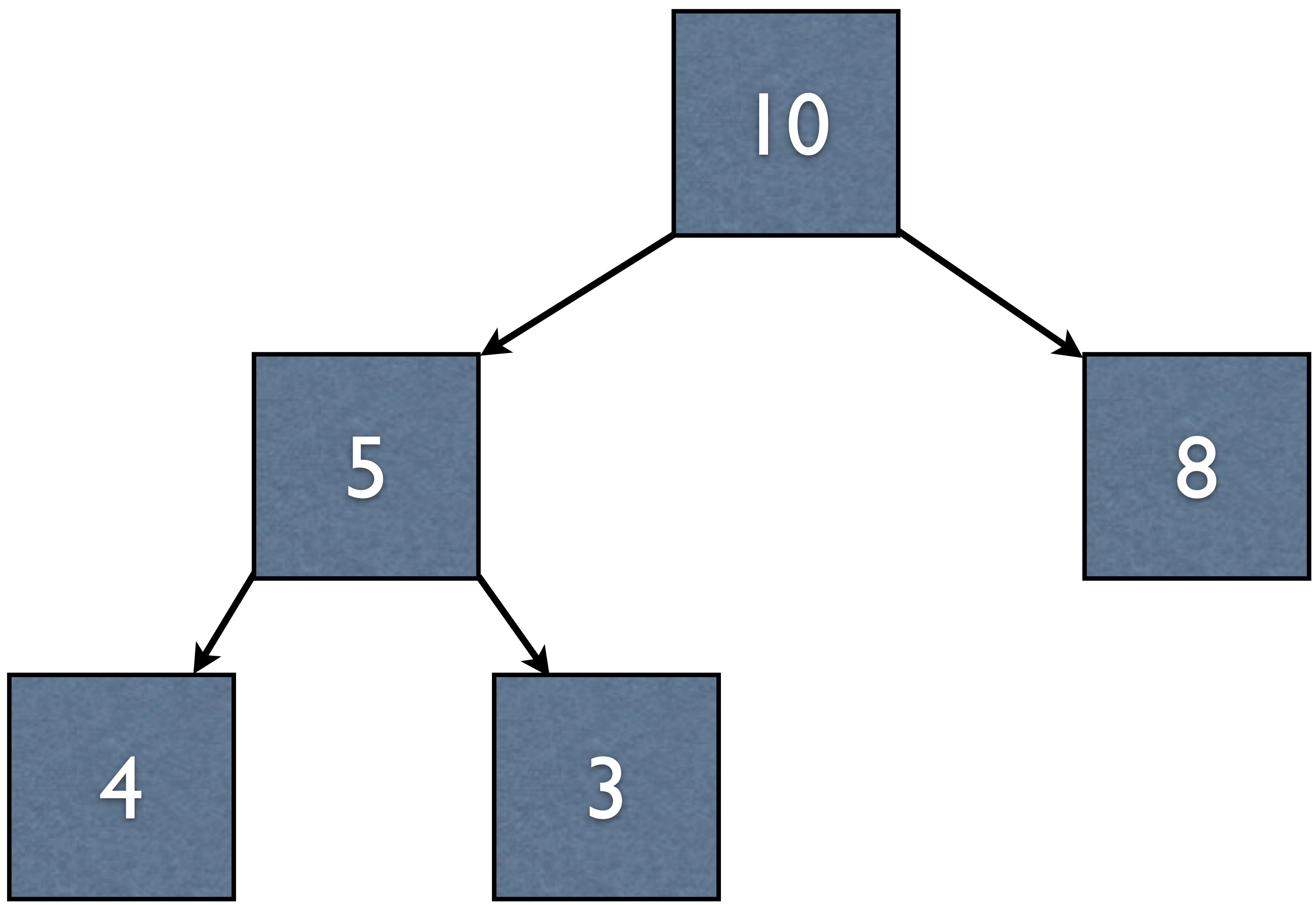


Additional Invariant

- In practice, heaps are always complete
  - What does this mean?

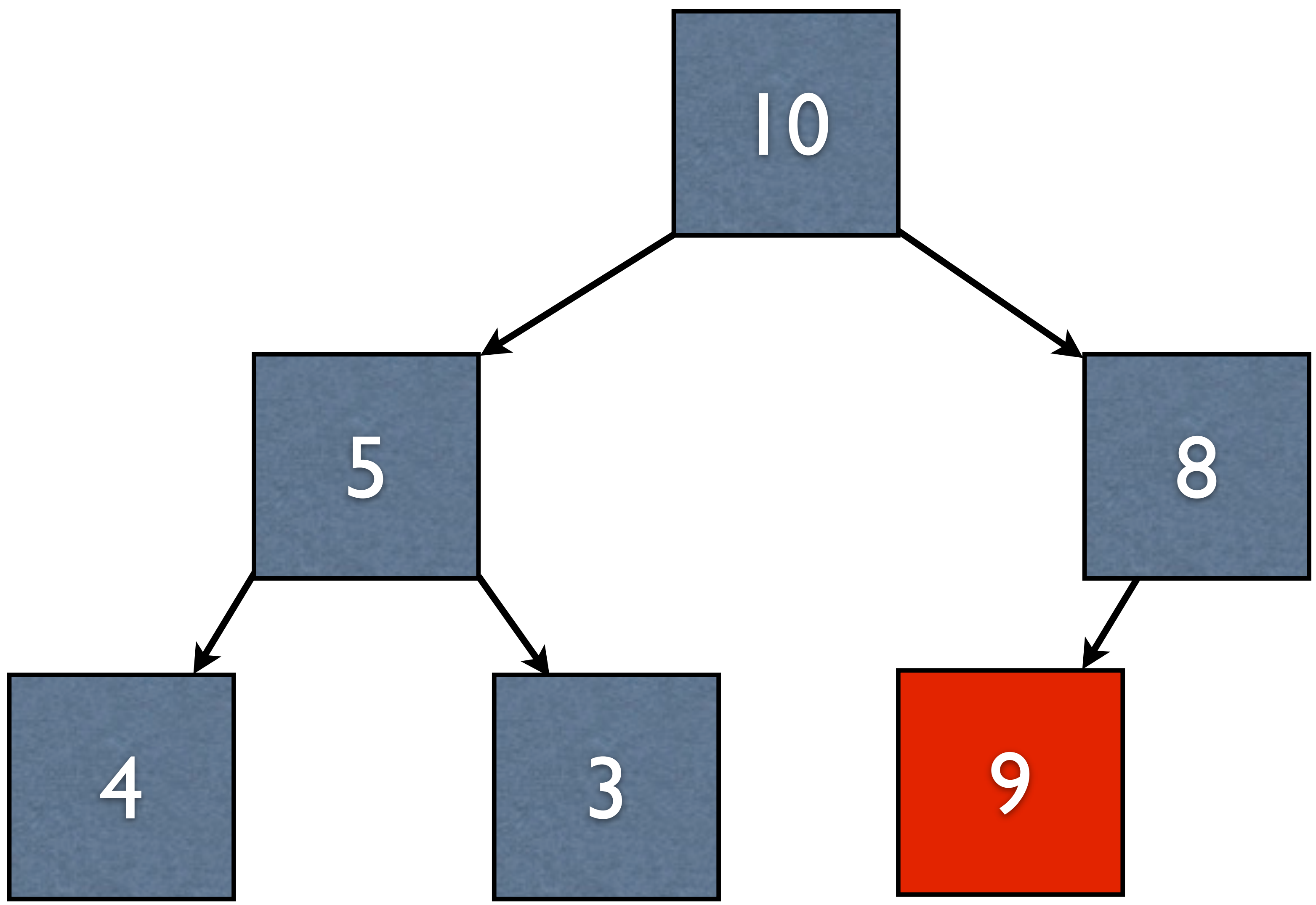
Additional Invariant

- In practice, heaps are always complete
  - What does this mean? - full except for the last row



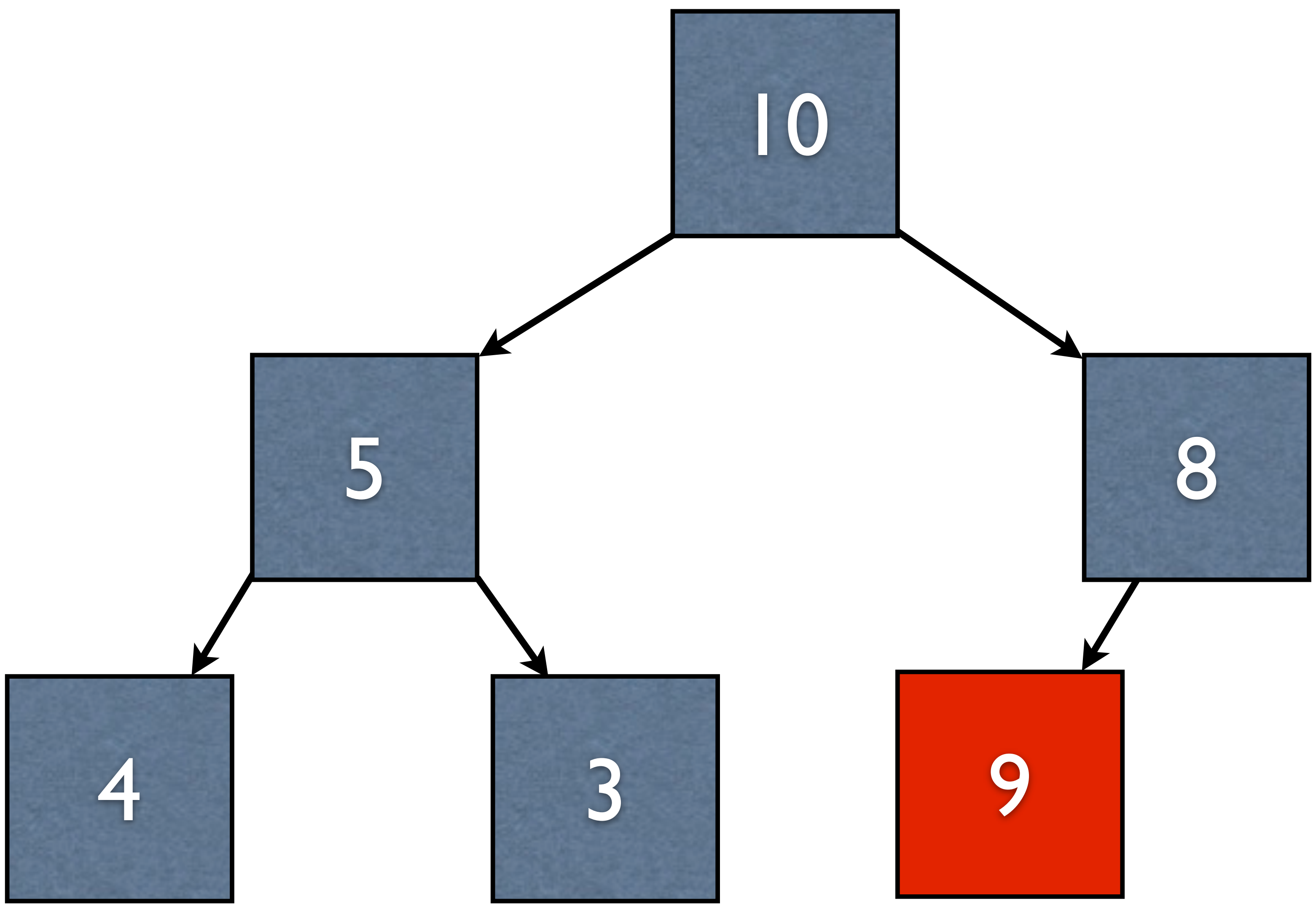
Enqueue

- If the tree is complete, we can enqueue by putting the element on the end
  - Not done yet - could violate heap property



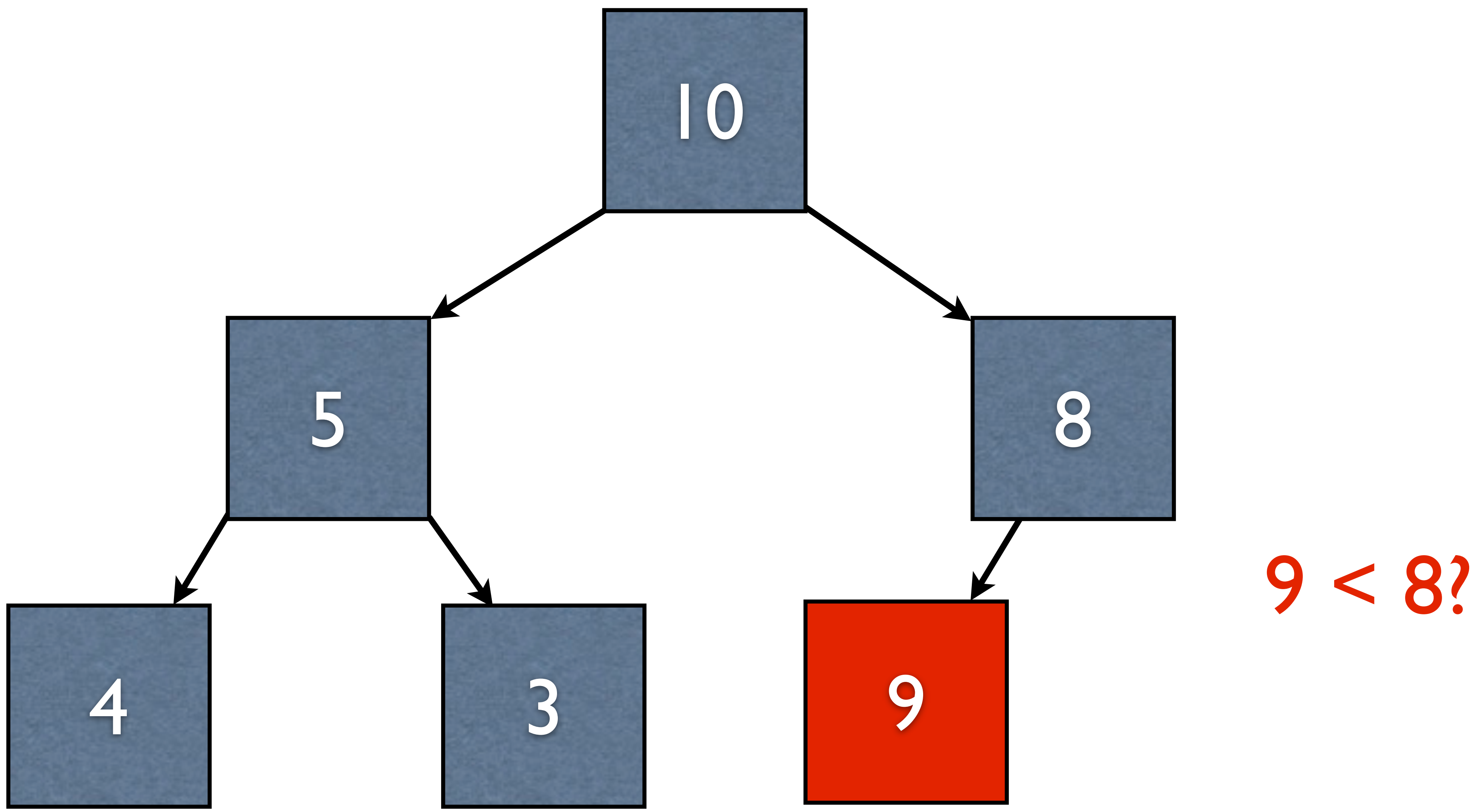
Enqueue

- To restore the heap property, we can *bubble up* - ensure the heap property holds stepwise with parents, and swap if not



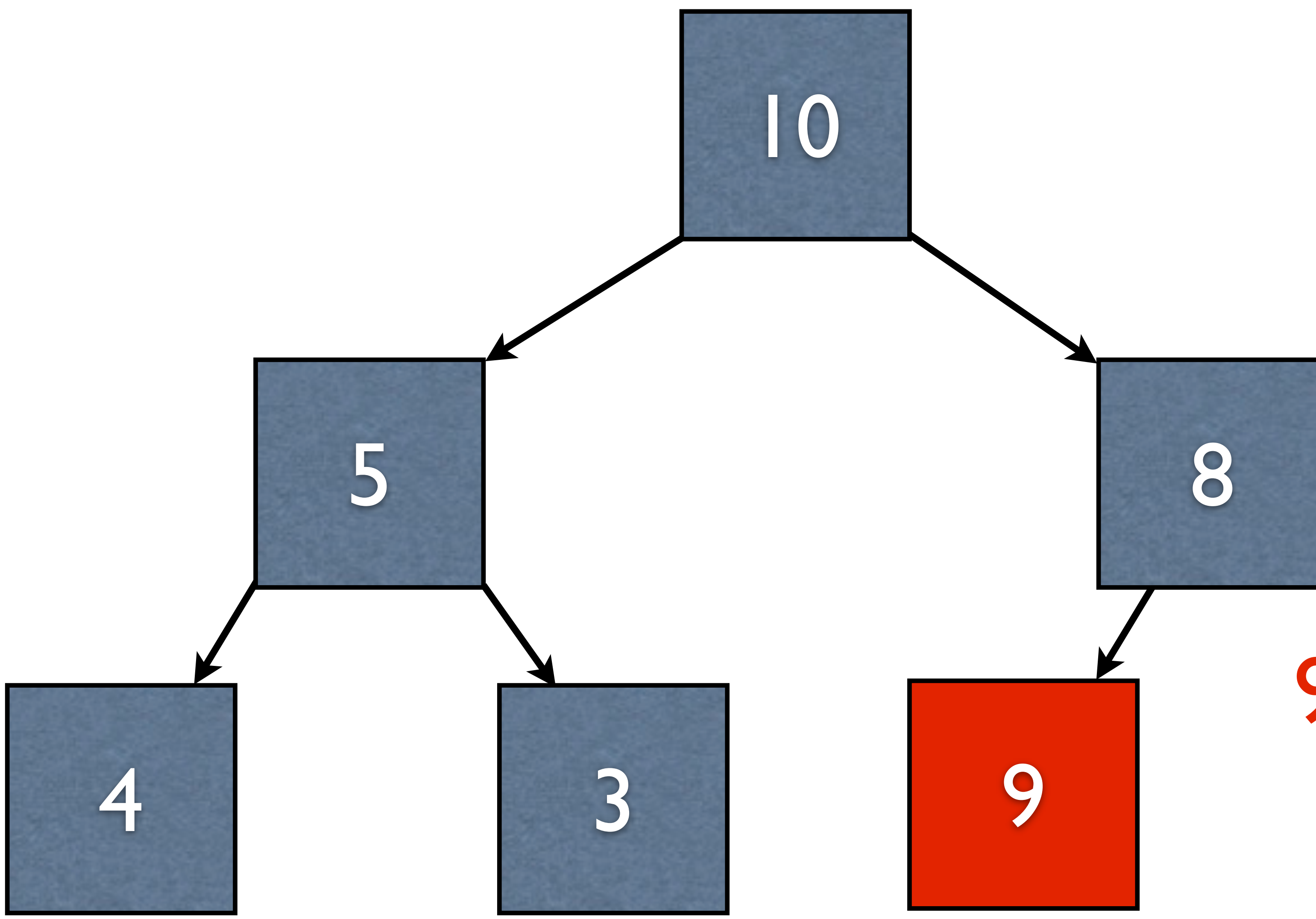
Enqueue

- To restore the heap property, we can *bubble up* - ensure the heap property holds stepwise with parents, and swap if not



Enqueue

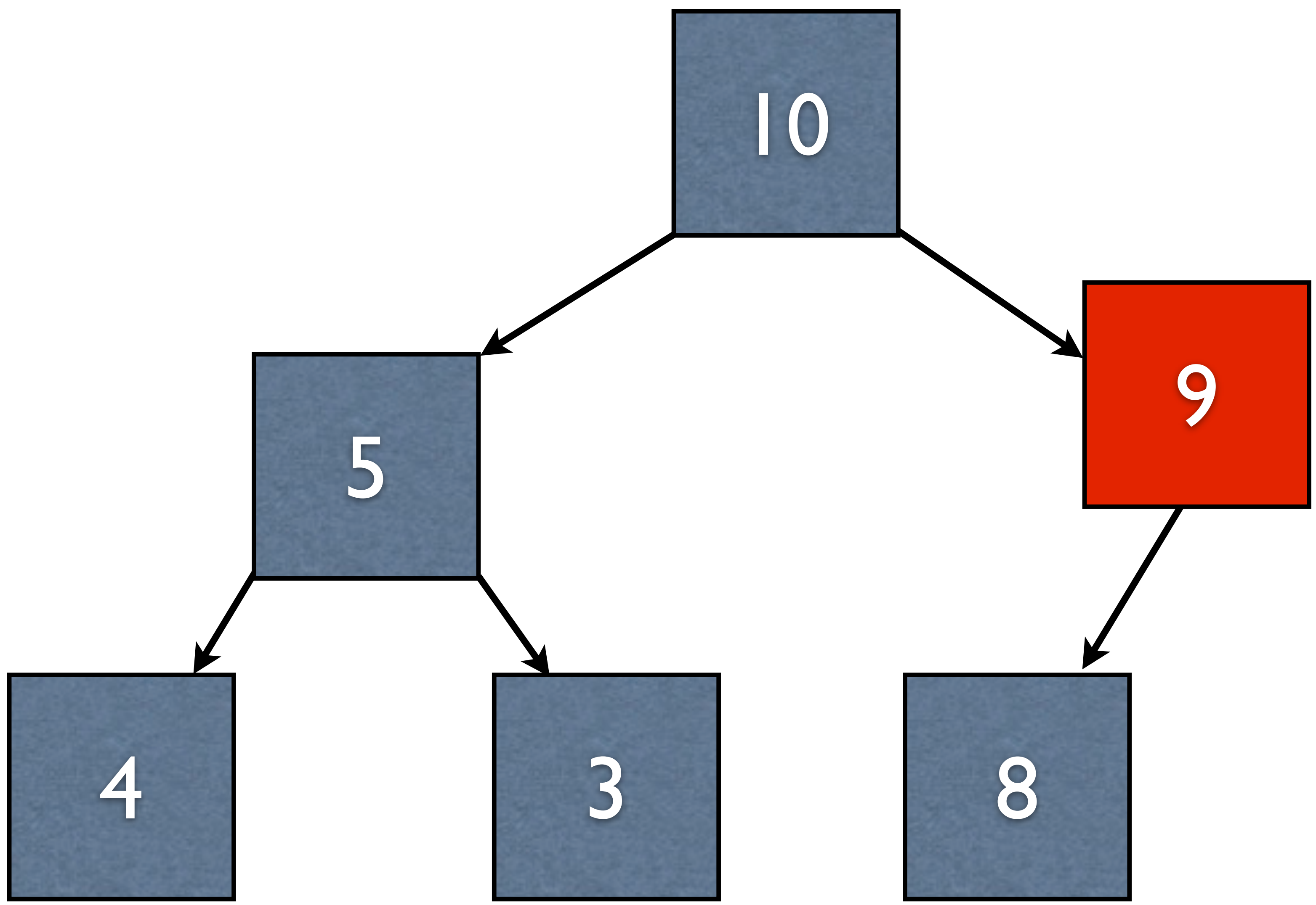
- To restore the heap property, we can *bubble up* - ensure the heap property holds stepwise with parents, and swap if not



9 < 8? - false;
bubble up

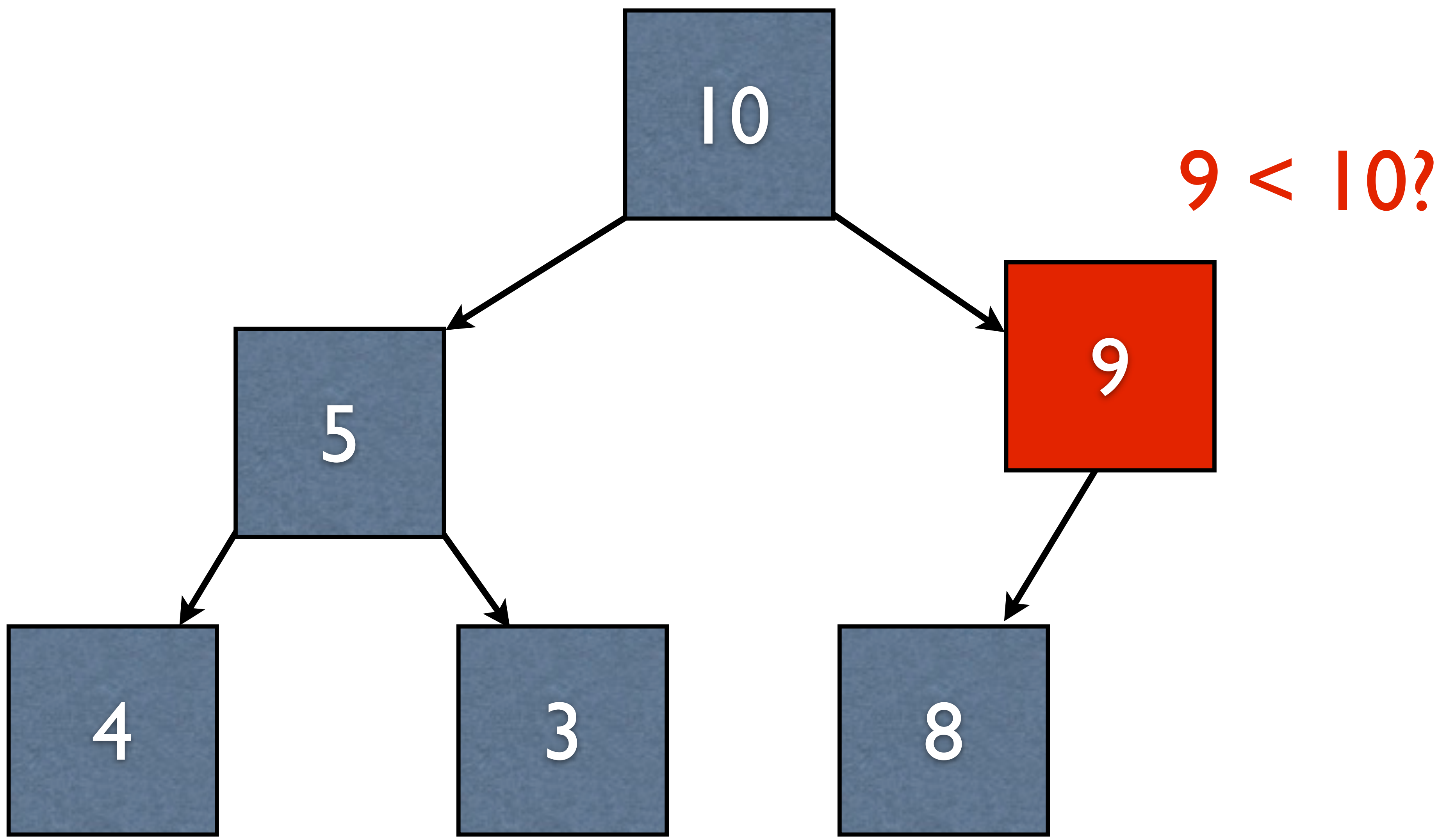
Enqueue

- To restore the heap property, we can *bubble up* - ensure the heap property holds stepwise with parents, and swap if not



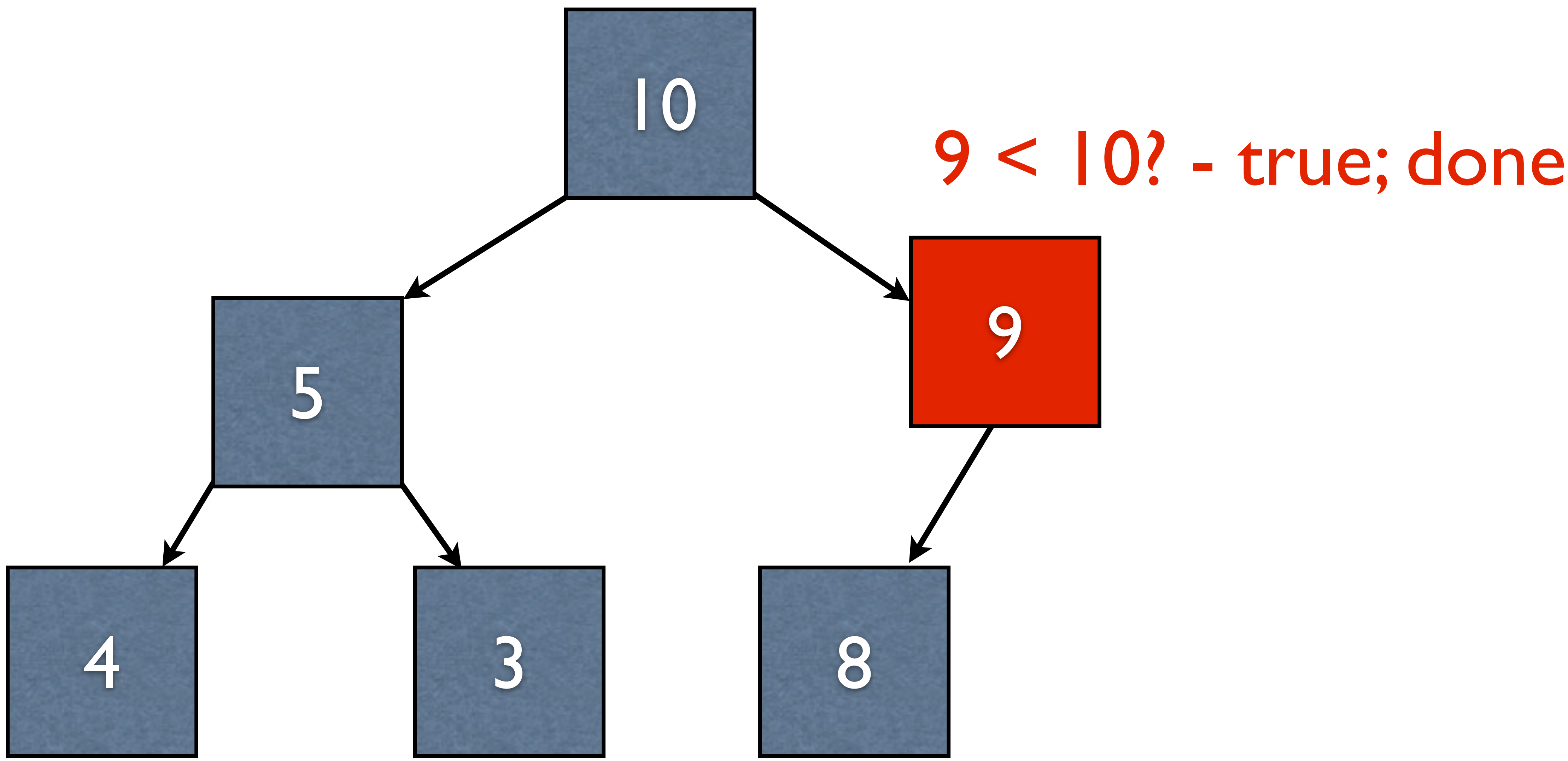
Enqueue

- To restore the heap property, we can *bubble up* - ensure the heap property holds stepwise with parents, and swap if not



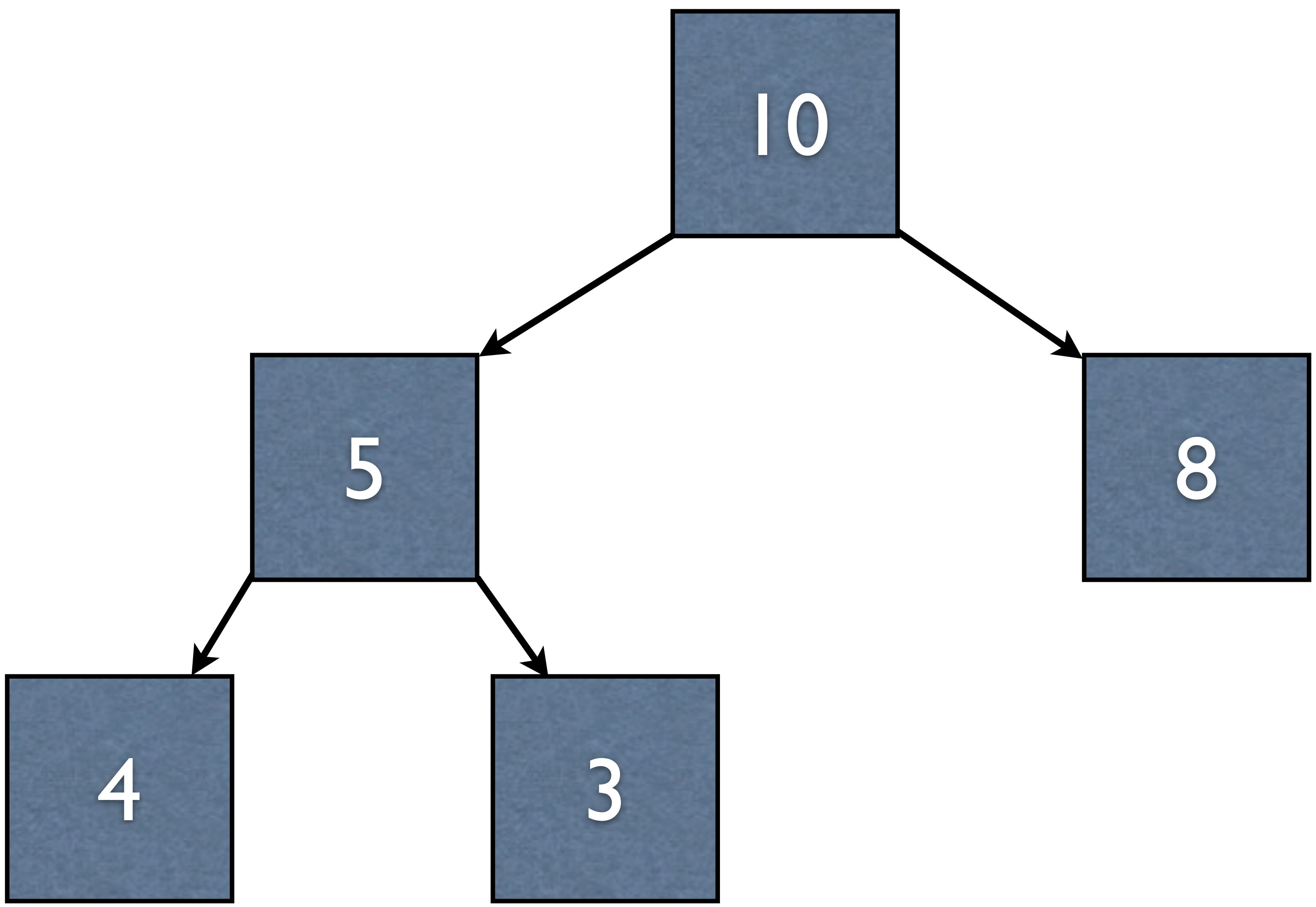
Enqueue

- To restore the heap property, we can *bubble up* - ensure the heap property holds stepwise with parents, and swap if not



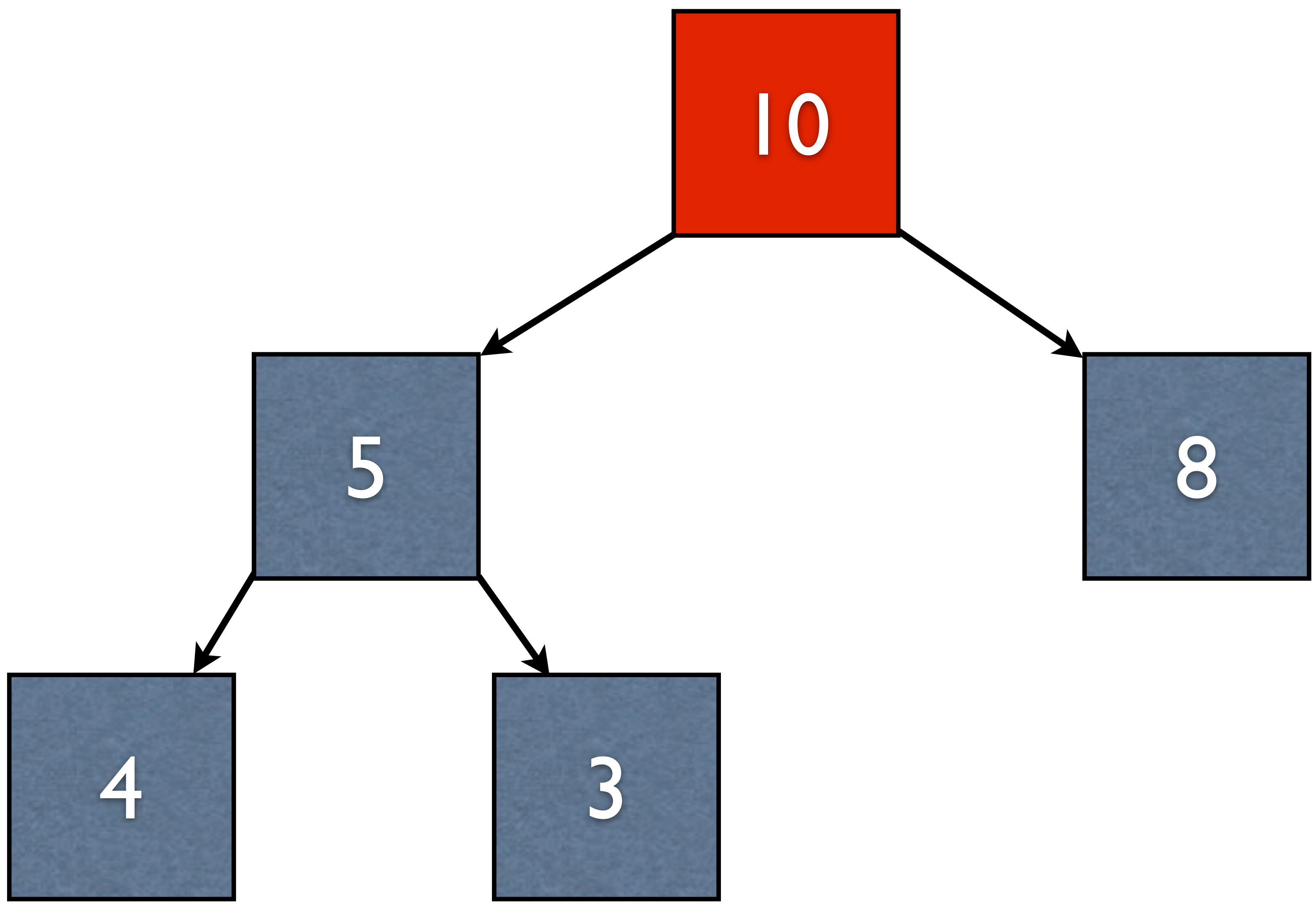
Dequeue

- After getting the element from the top of the tree, we must restore the heap



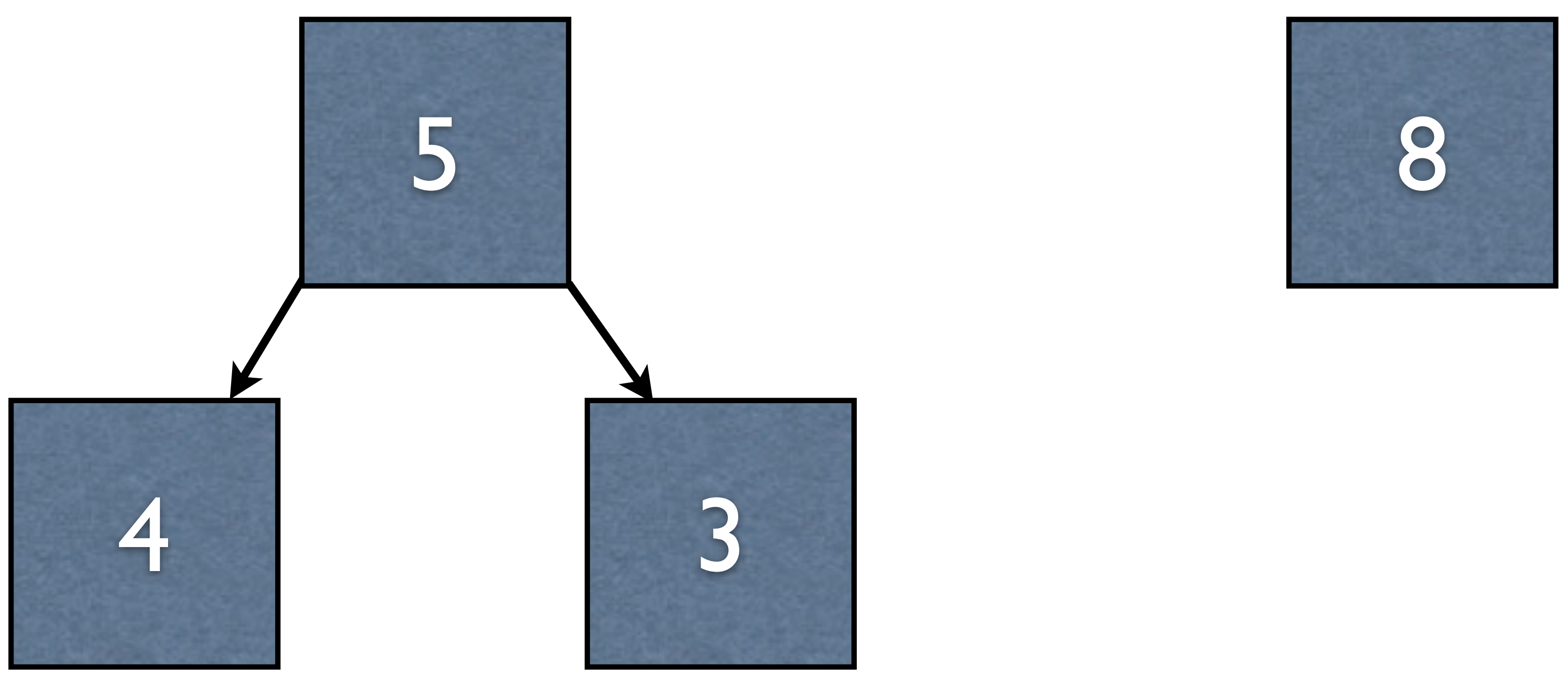
Dequeue

- After getting the element from the top of the tree, we must restore the heap



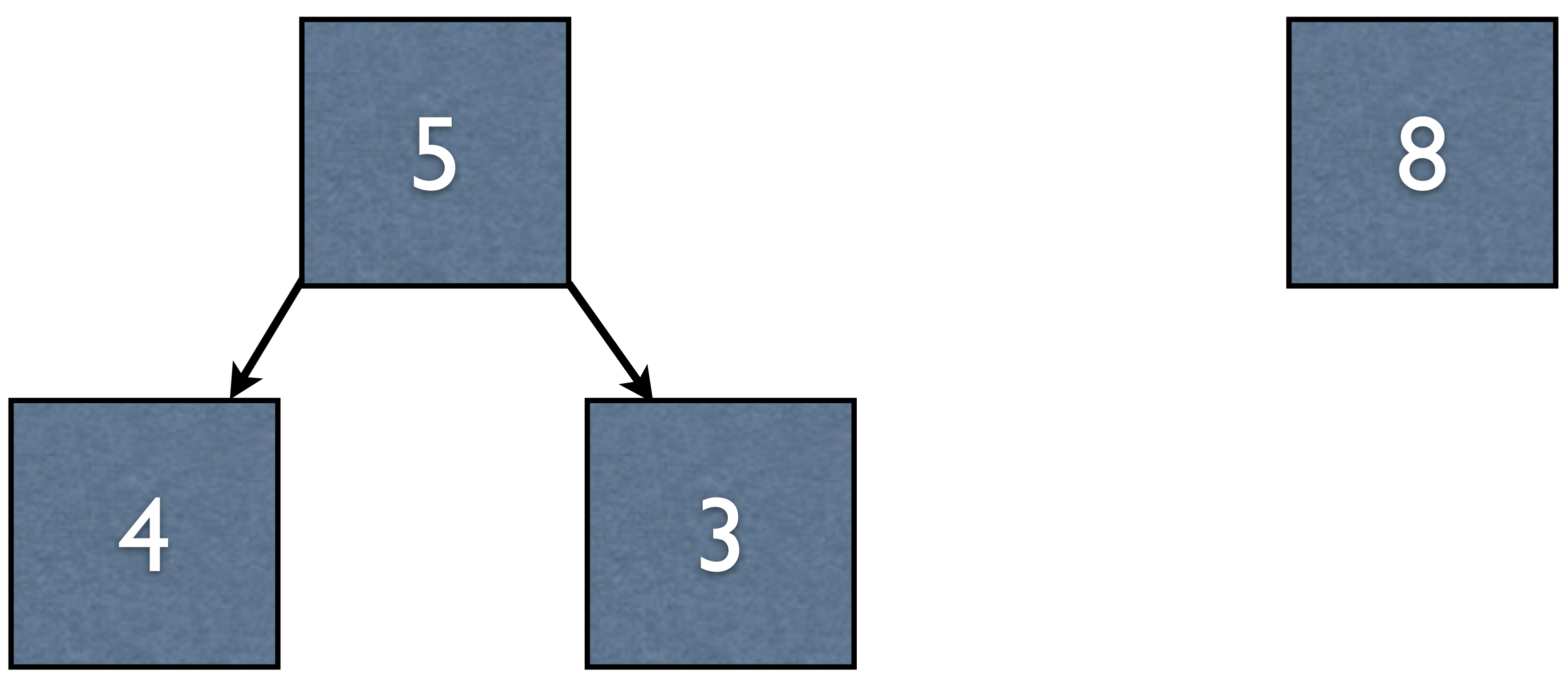
Dequeue

- After getting the element from the top of the tree, we must restore the heap



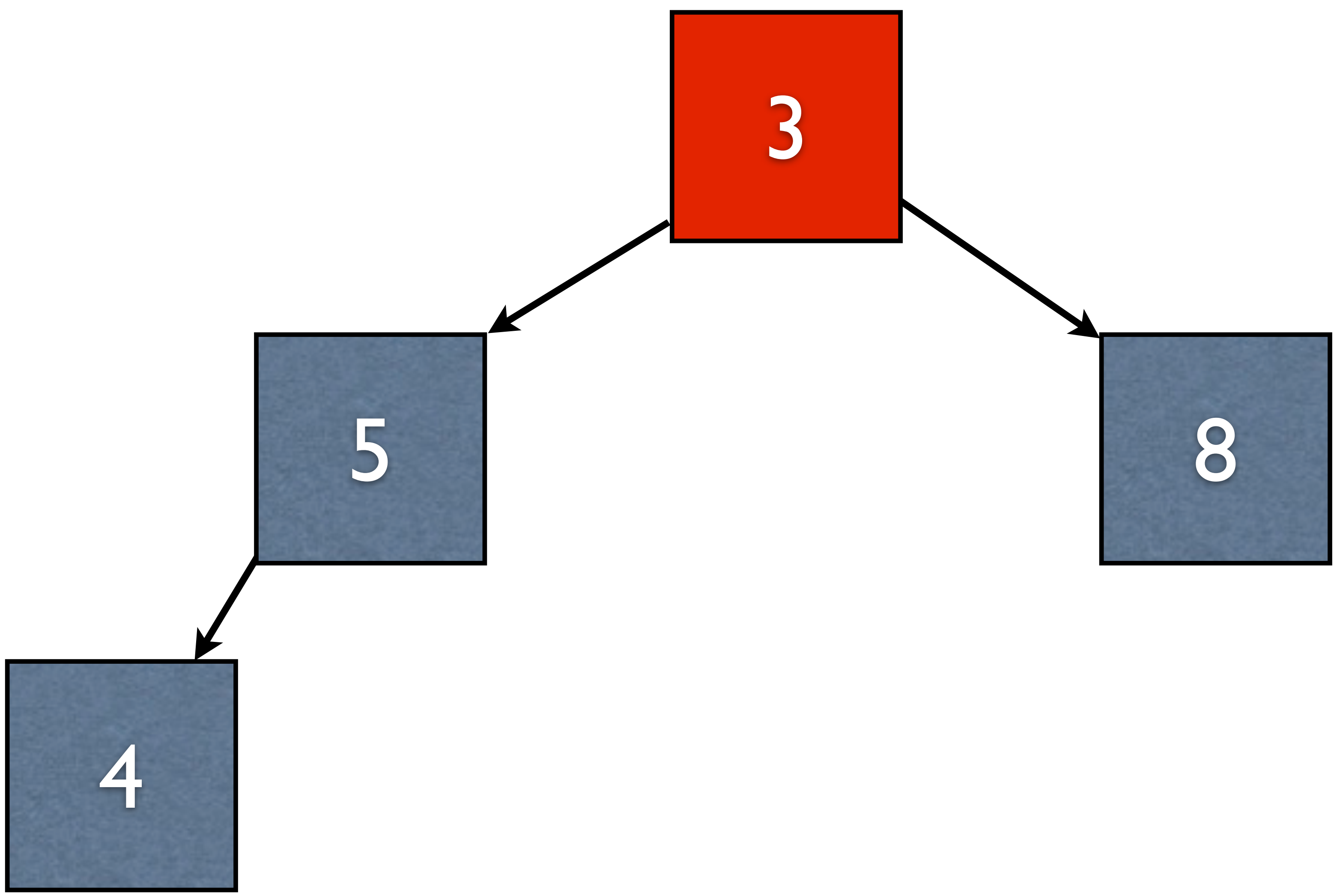
Dequeue

- After getting the element from the top of the tree, we must restore the heap
  - Idea: swap in the last node from the last level



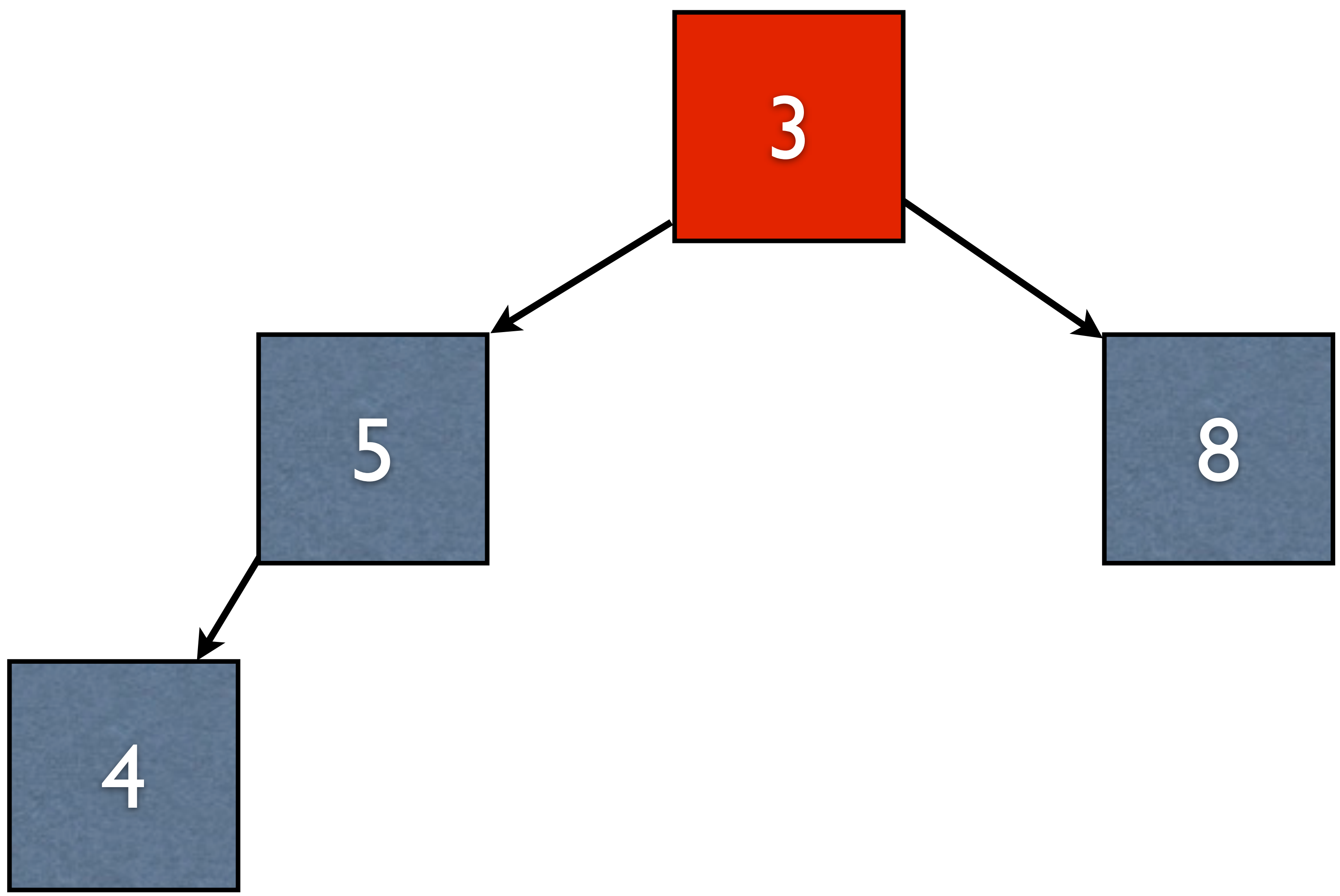
Dequeue

- After getting the element from the top of the tree, we must restore the heap
  - Idea: swap in the last node from the last level



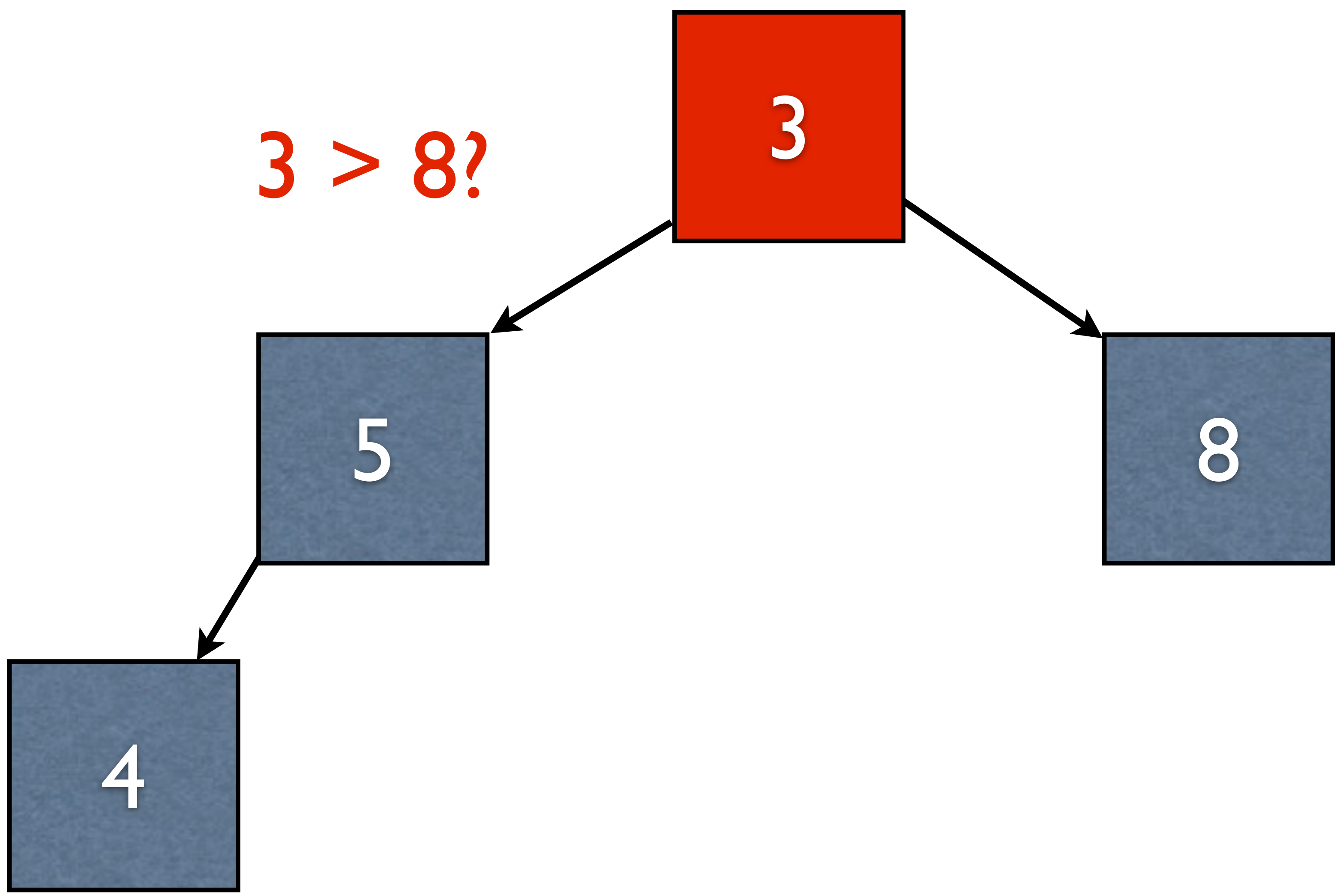
Dequeue

- In order to restore the heap property, we must *bubble down* - swap with the greatest of the children recursively



Dequeue

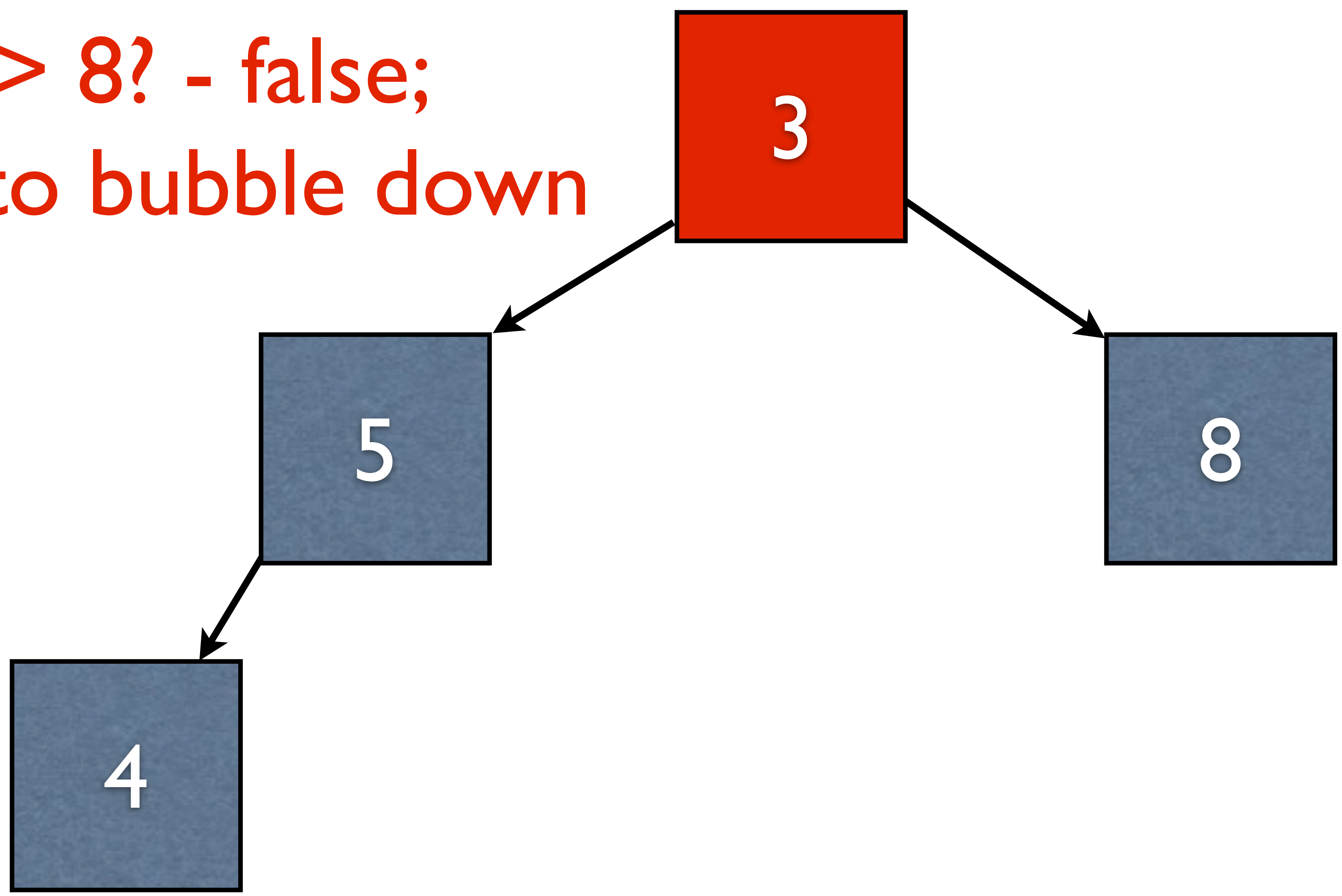
- In order to restore the heap property, we must *bubble down* - swap with the greatest of the children recursively



Dequeue

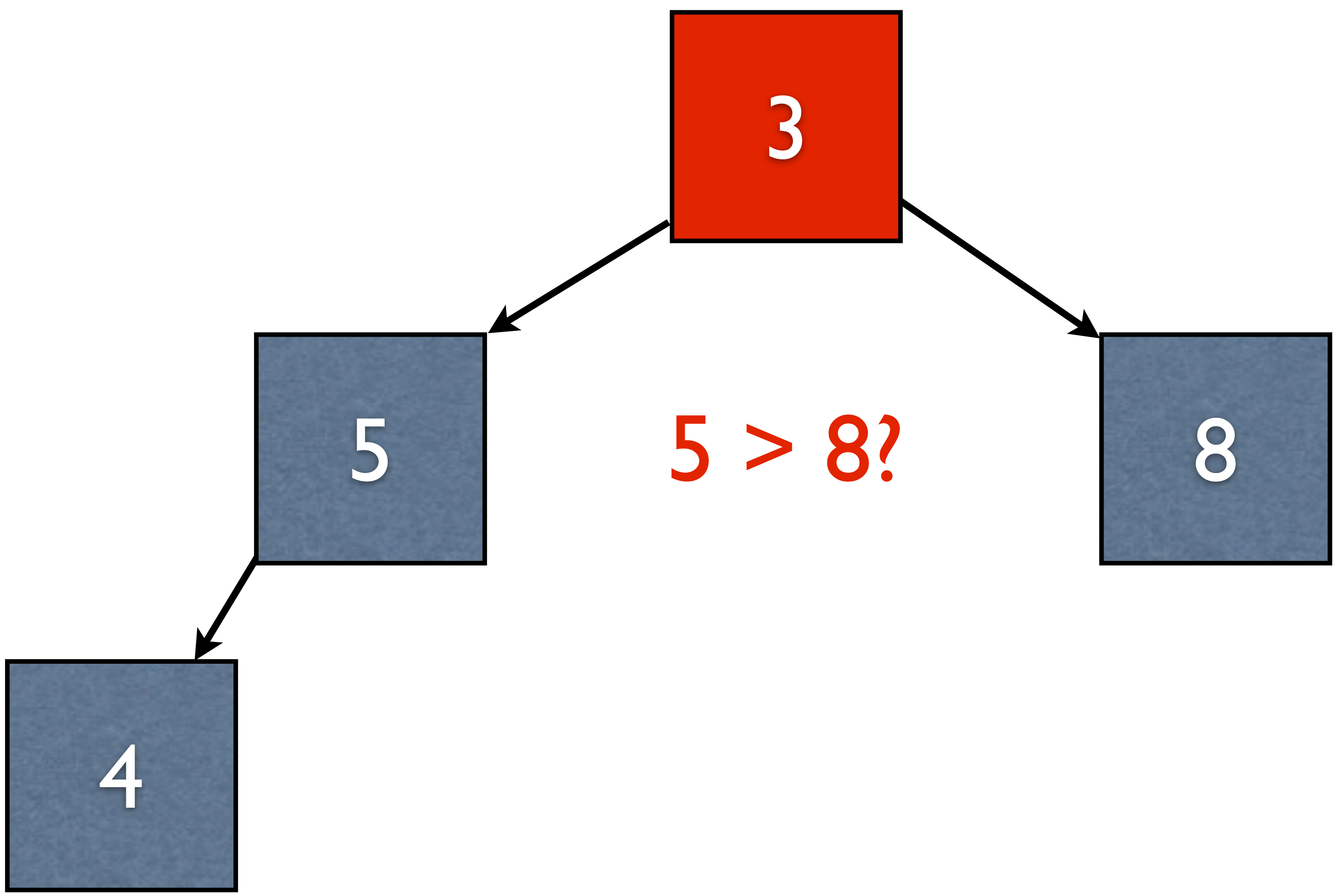
- In order to restore the heap property, we must *bubble down* - swap with the greatest of the children recursively

3 > 8? - false;
need to bubble down



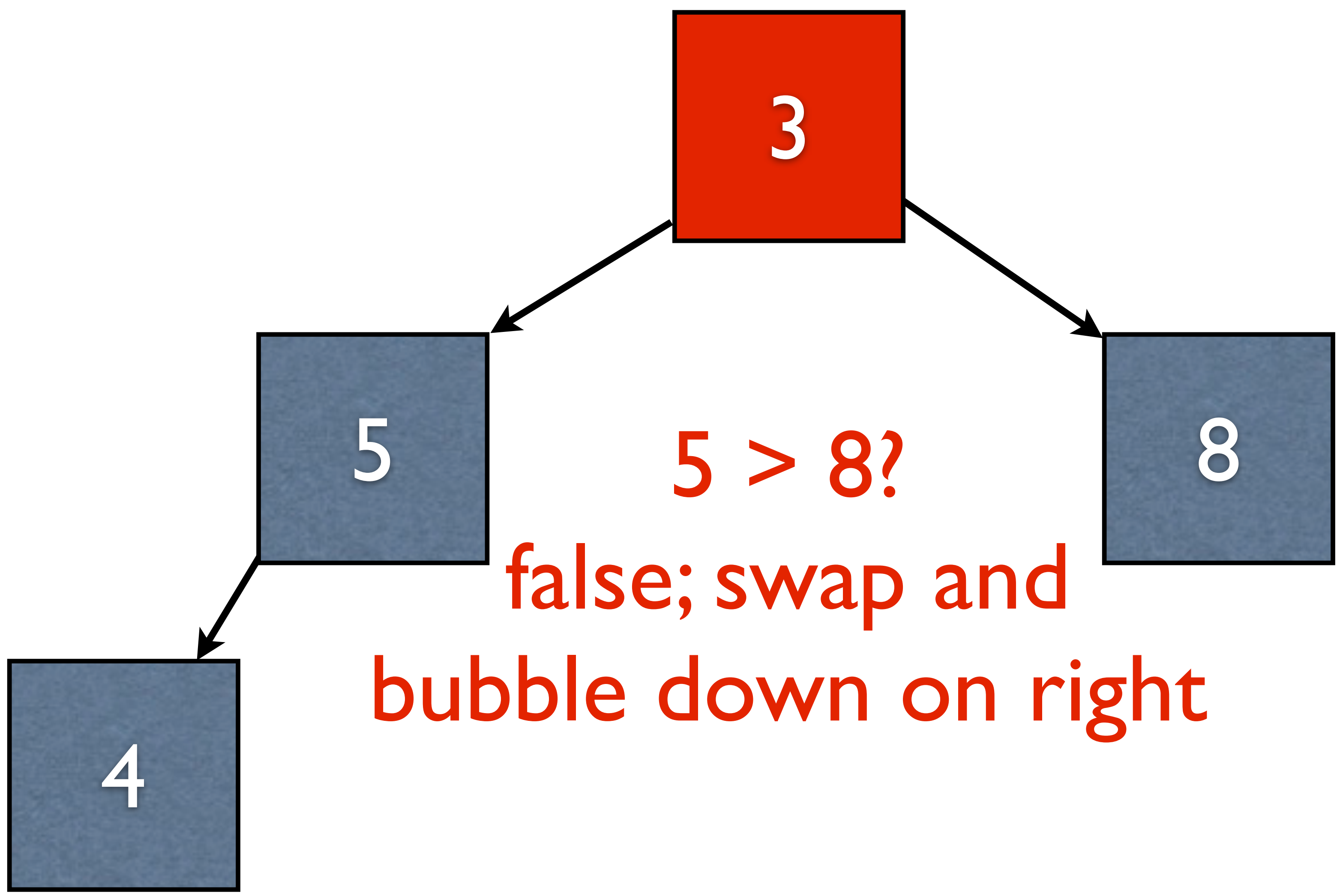
Dequeue

- In order to restore the heap property, we must *bubble down* - swap with the greatest of the children recursively



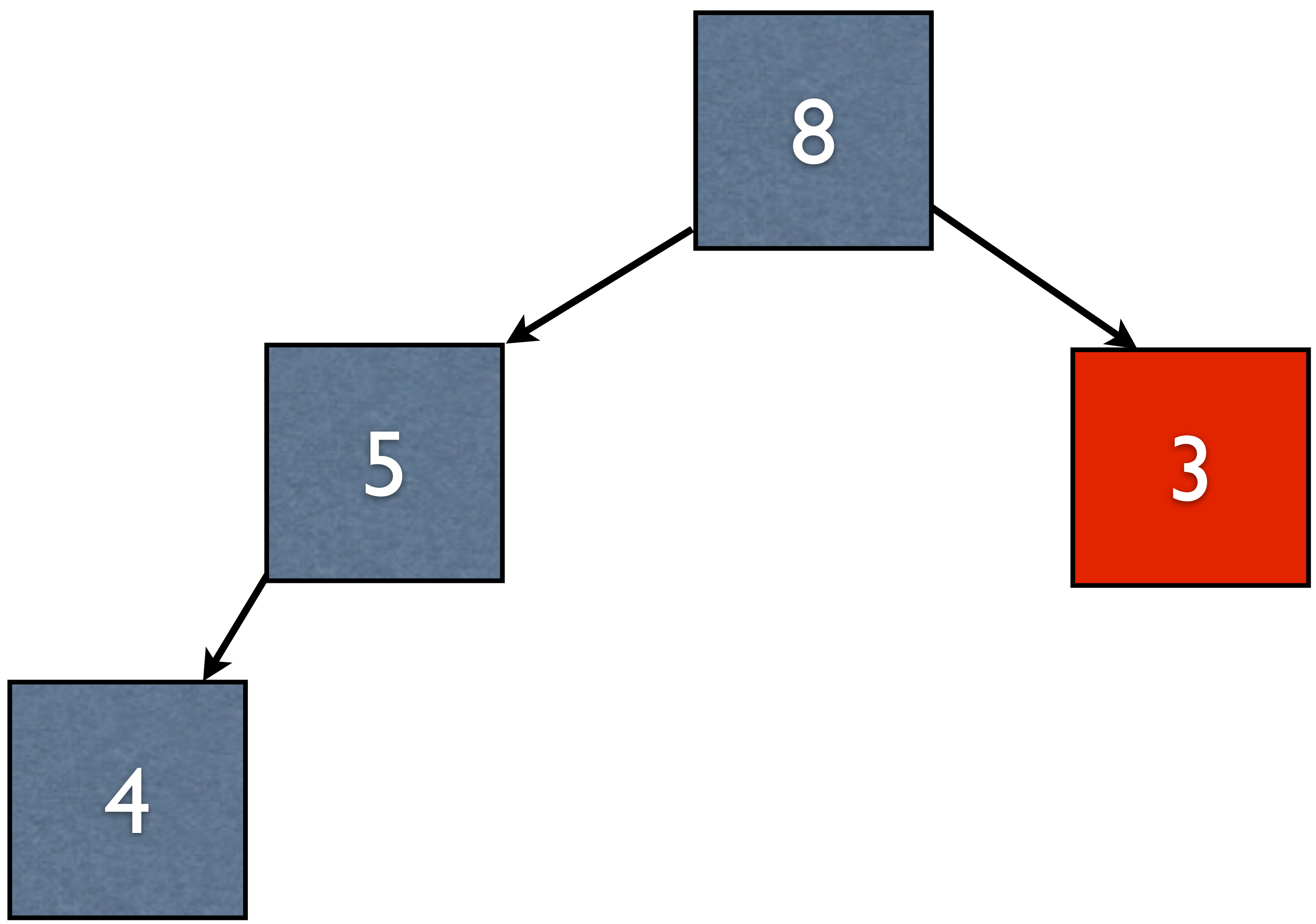
Dequeue

- In order to restore the heap property, we must *bubble down* - swap with the greatest of the children recursively



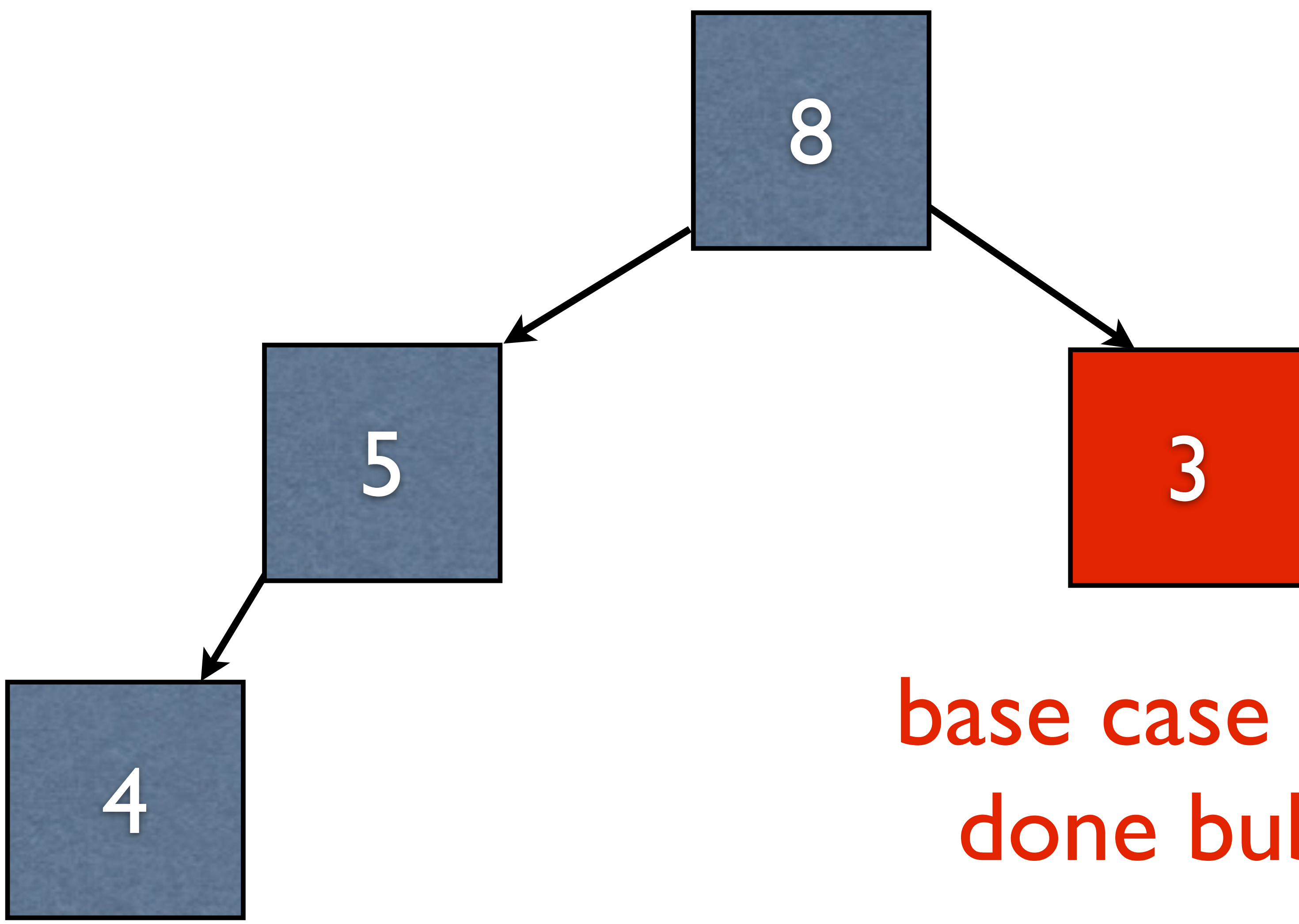
Dequeue

- In order to restore the heap property, we must *bubble down* - swap with the greatest of the children recursively



Dequeue

- In order to restore the heap property, we must *bubble down* - swap with the greatest of the children recursively



base case - no children; done bubbling down

Time Complexity

- Because we force the construction to be complete, we get balanced trees
- Dequeue and enqueue are both `O(log(N))` as a result

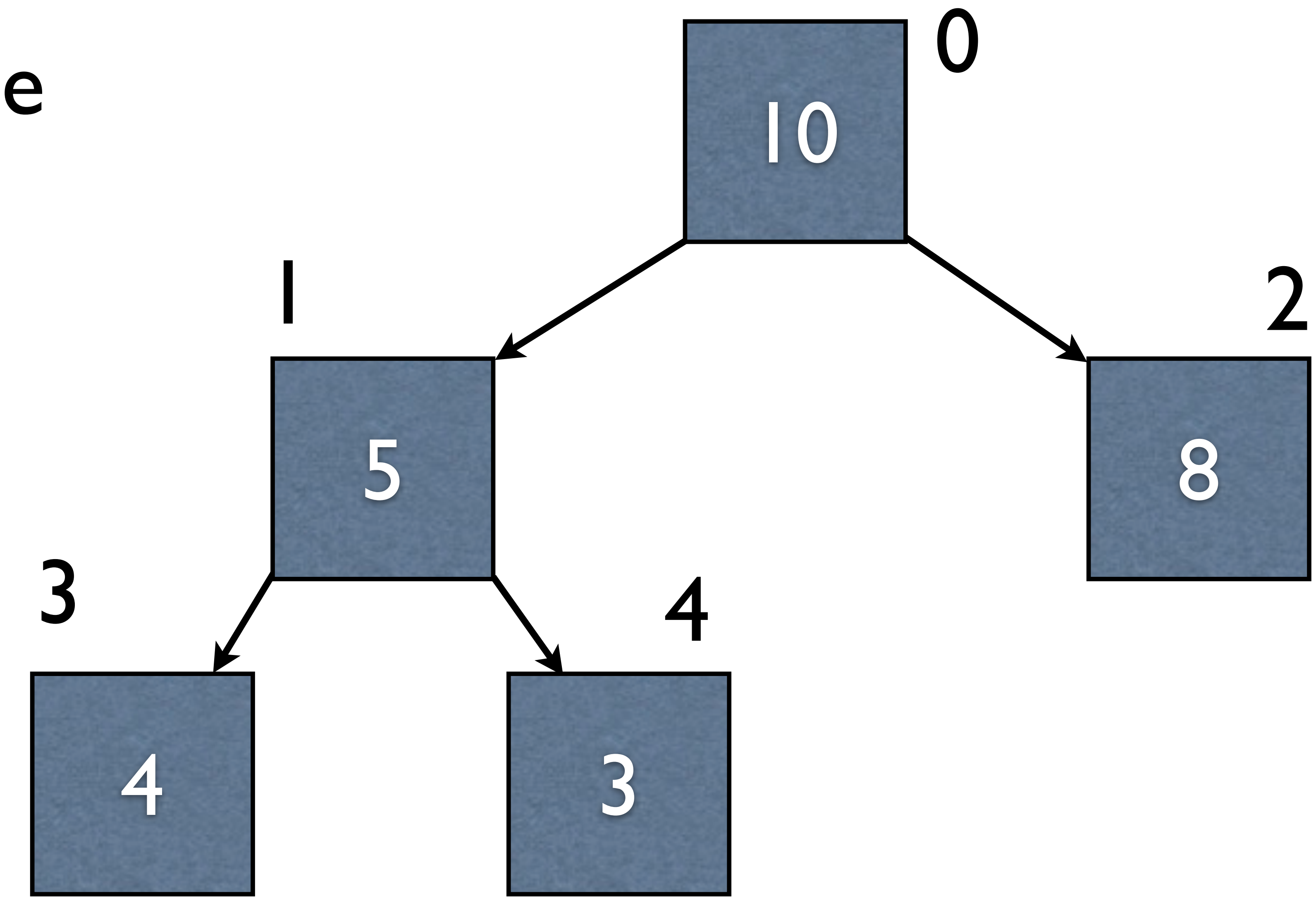
Optimization

- Heaps can be concisely represented with arrays

As Array

10	5	8	4	3

As Tree



Advantages of Arrays

- What sort of advantages does an array representation have?

Advantages of Arrays

- What sort of advantages does an array representation have?
  - Overall simpler
  - Less space consumed for the same data
  - Getting the last node at the last level is just getting the last valid element in the array
  - (Advanced) CPUs are much happier with arrays than trees (i.e., better performance)

Disadvantages of Arrays

- What sort of issues does the array representation have?

Disadvantages of Arrays

- What sort of issues does the array representation have?
  - Adding elements is more difficult; may entail reallocating the whole array
  - In practice, this is very minor compared to all the other advantages