Lecture 17: Tree Removal and Heaps

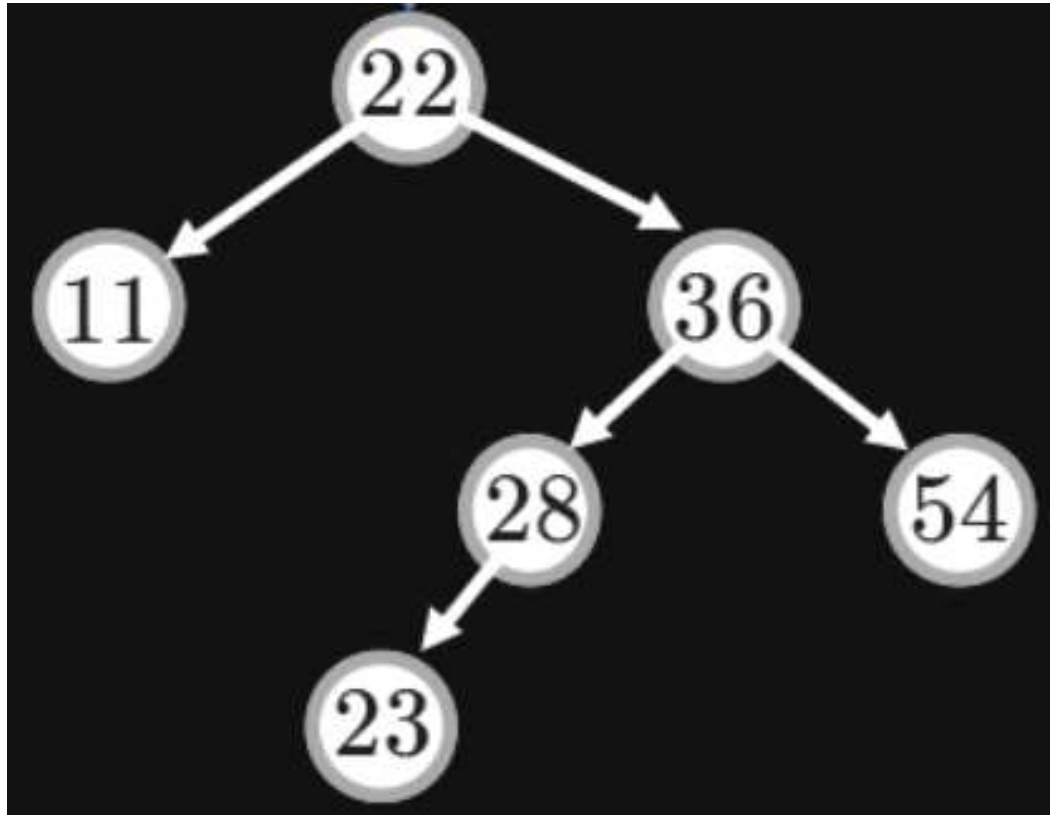
Kaleel Mahmood

Department of Computer Science and Engineering

University of Connecticut

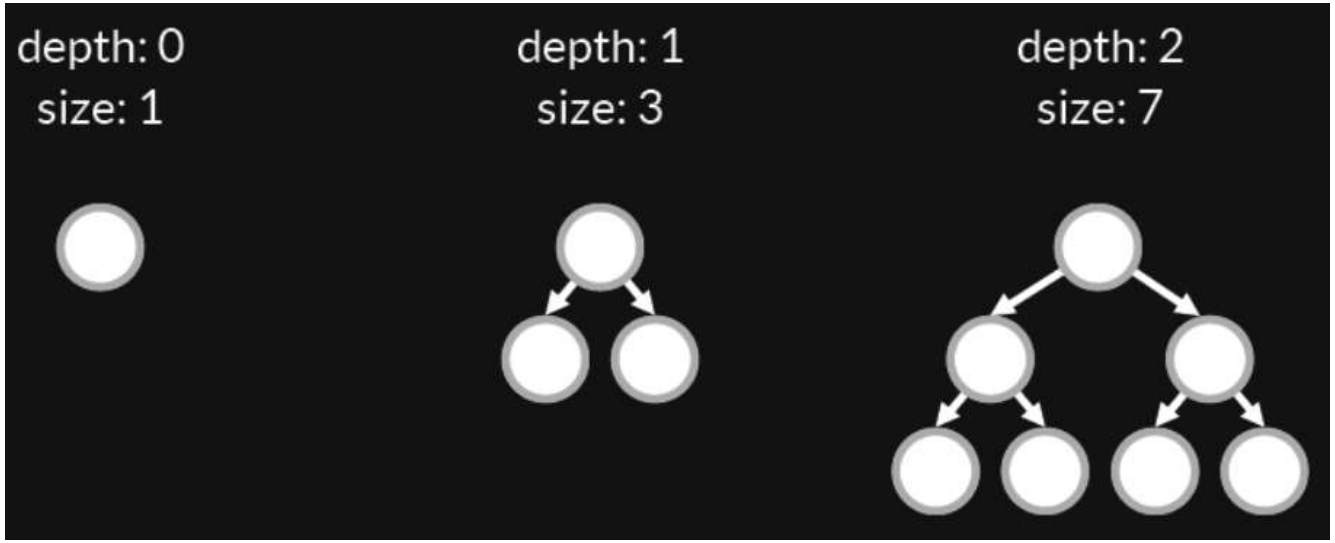
Previously in CSE 1729…

Binary Search Trees



Definition of Depth and Size

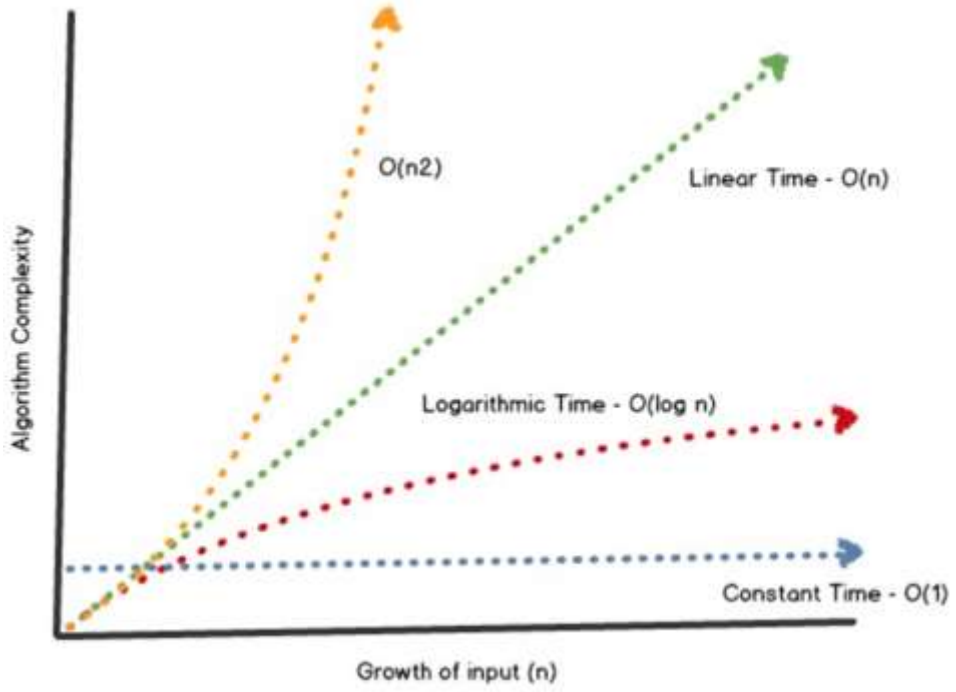
$S_d = 2^{d+1} - 1$ (approximately 2^{d+1})



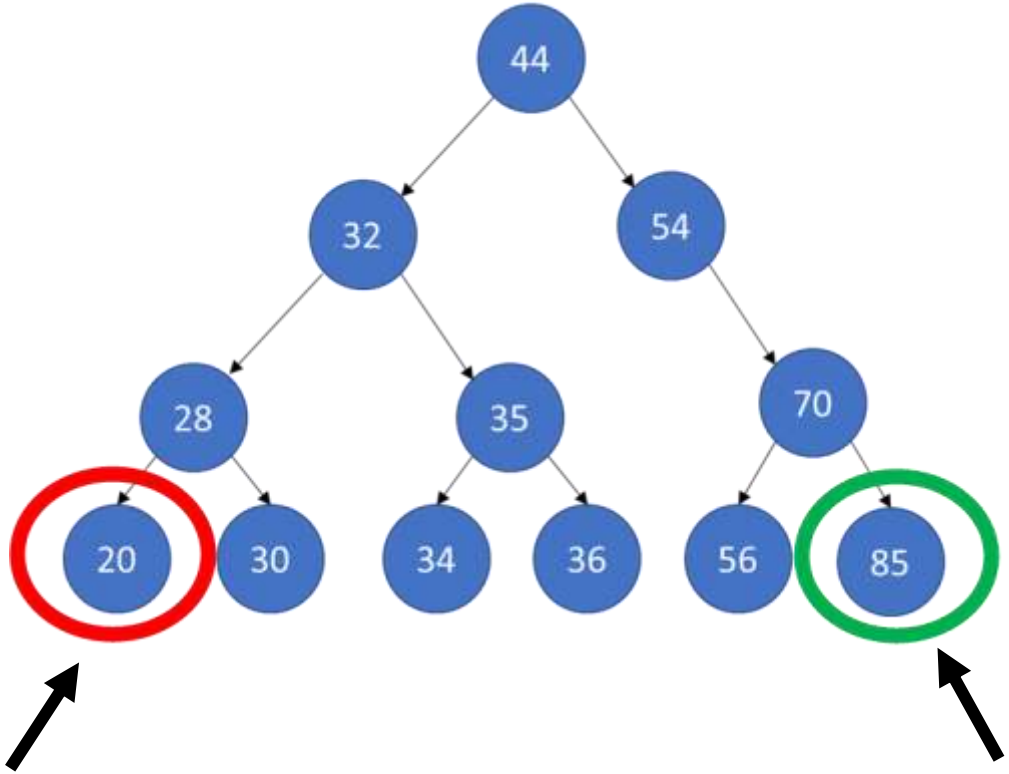
Previously in CSE 1729…

Why do we like Binary Search Trees?

The search complexity grows logarithmically instead of linearly (like a list) as the number of elements in the data structure increases.



Where is the minimum and maximum element in a BST?



Leftmost is the minimum element

Rightmost is the maximum element

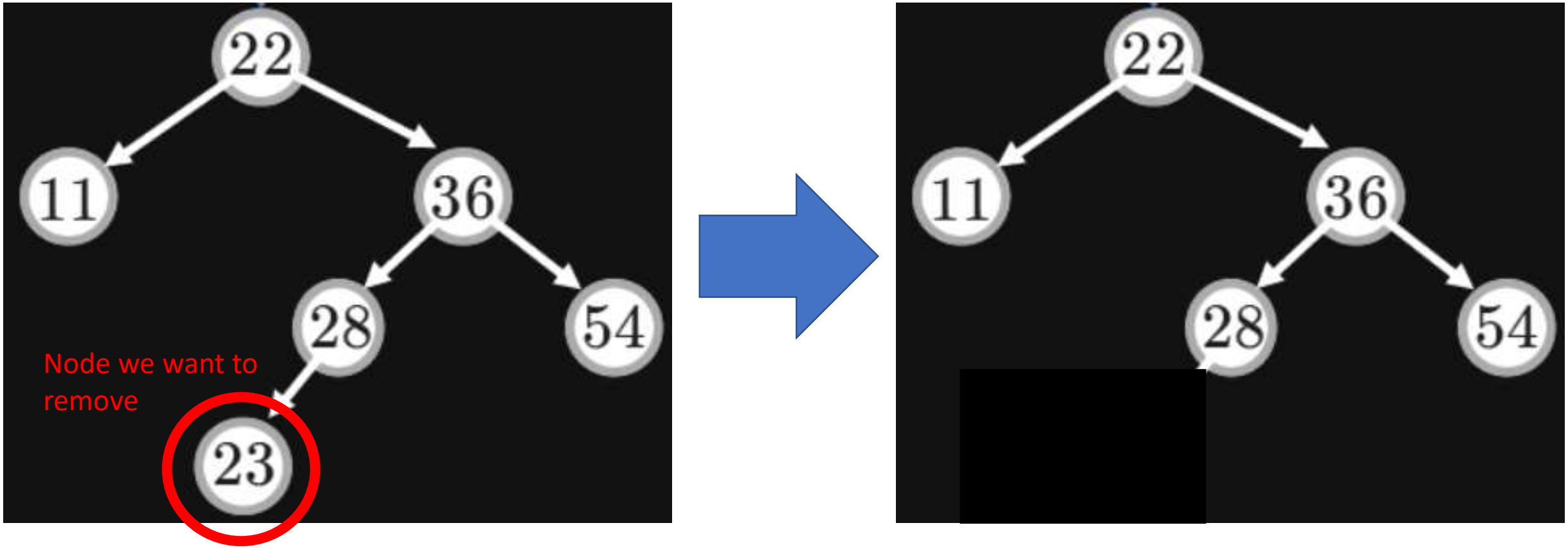
First topic of today's lecture: Tree Removal

REMOVAL

- Problem
  - Remove the value at a position from the tree
- Input
  - A binary search tree + the position of element
- Output
  - A binary search tree that no longer contains the element
- Structural invariant: Binary search tree property preserved
  - For all nodes,
    - all the elements in the left subtree are < root and
    - all elements in the right subtree are > root.

Break it down case by case…

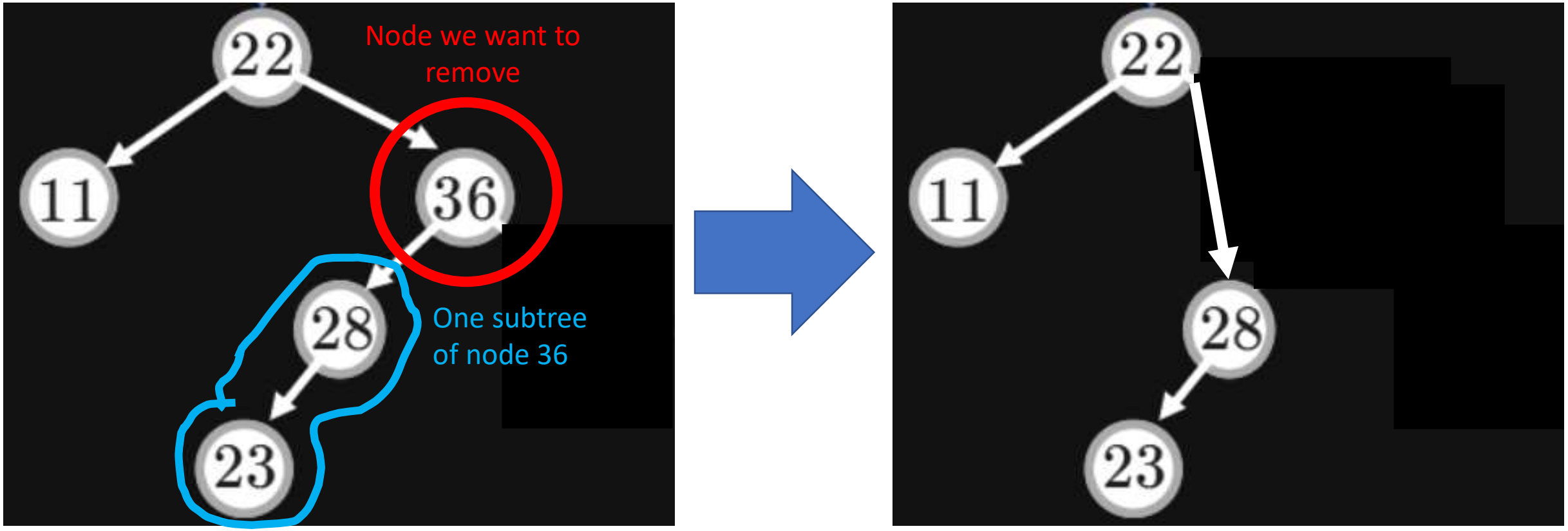
Case 1: The node has no subtrees



Solution: Just remove 23 by replacing it with an empty list.
No other structural changes needed.

Break it down case by case…

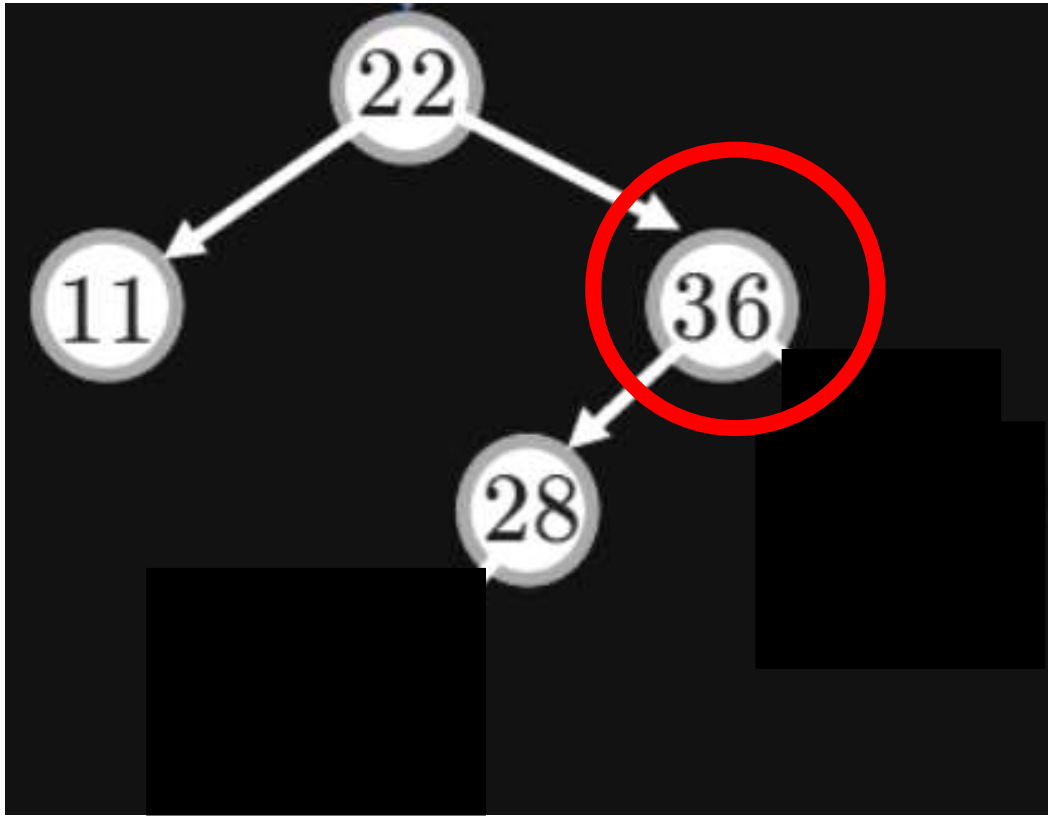
Case 2: There is only one subtree connected to the node



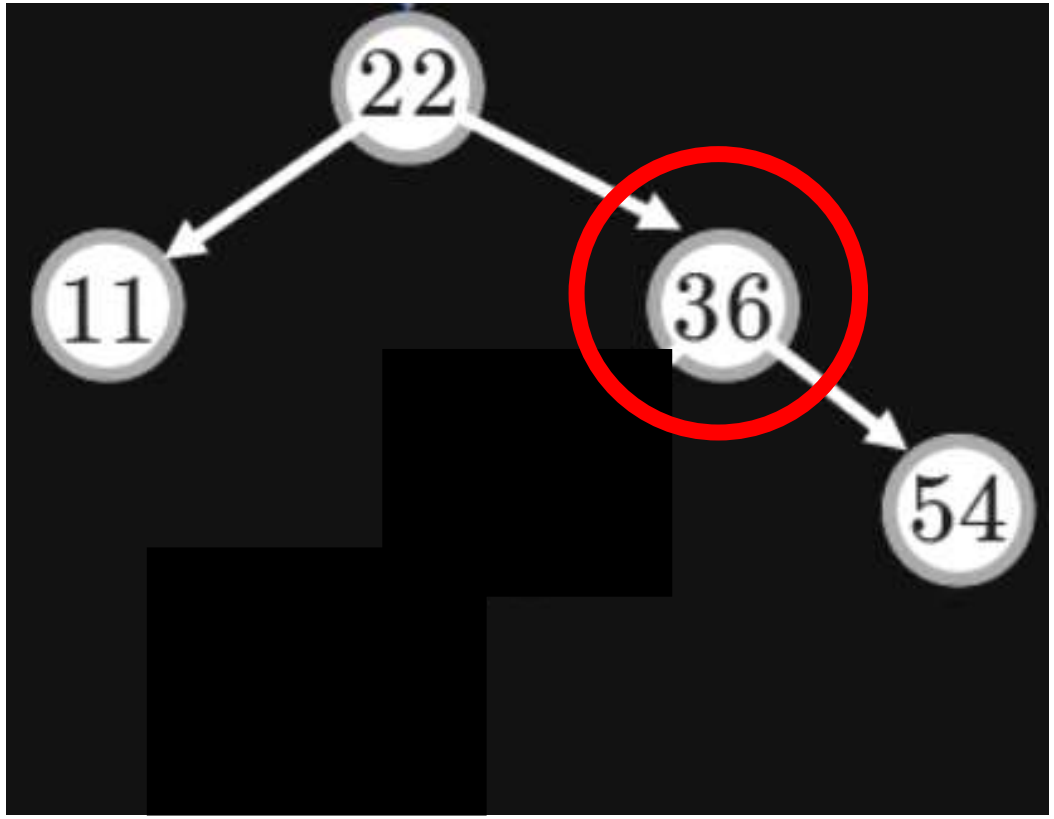
Solution: Remove 36 and the child of 36 becomes the child of 22.

Question: For case 2 does it matter if 36 has left subtree or right subtree for connecting?

Case 2A: Node 36 only has a LEFT subtree.



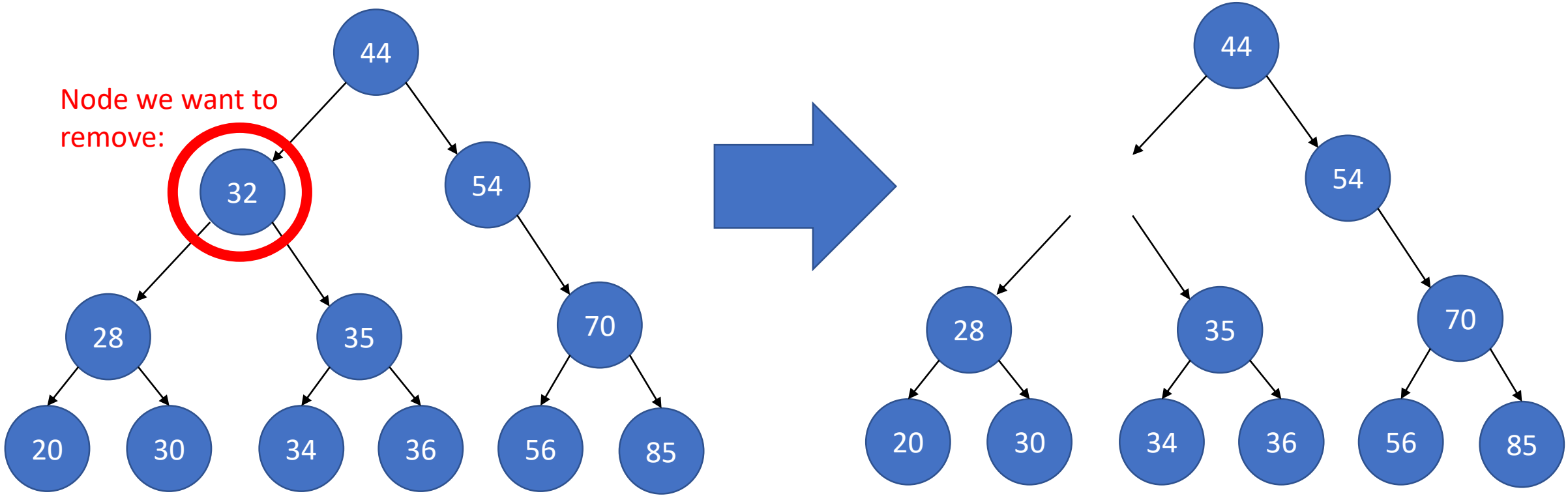
Case 2B: Node 36 only has a RIGHT subtree.



No because in both cases the elements in the subtrees are always greater than the root you are connecting to.

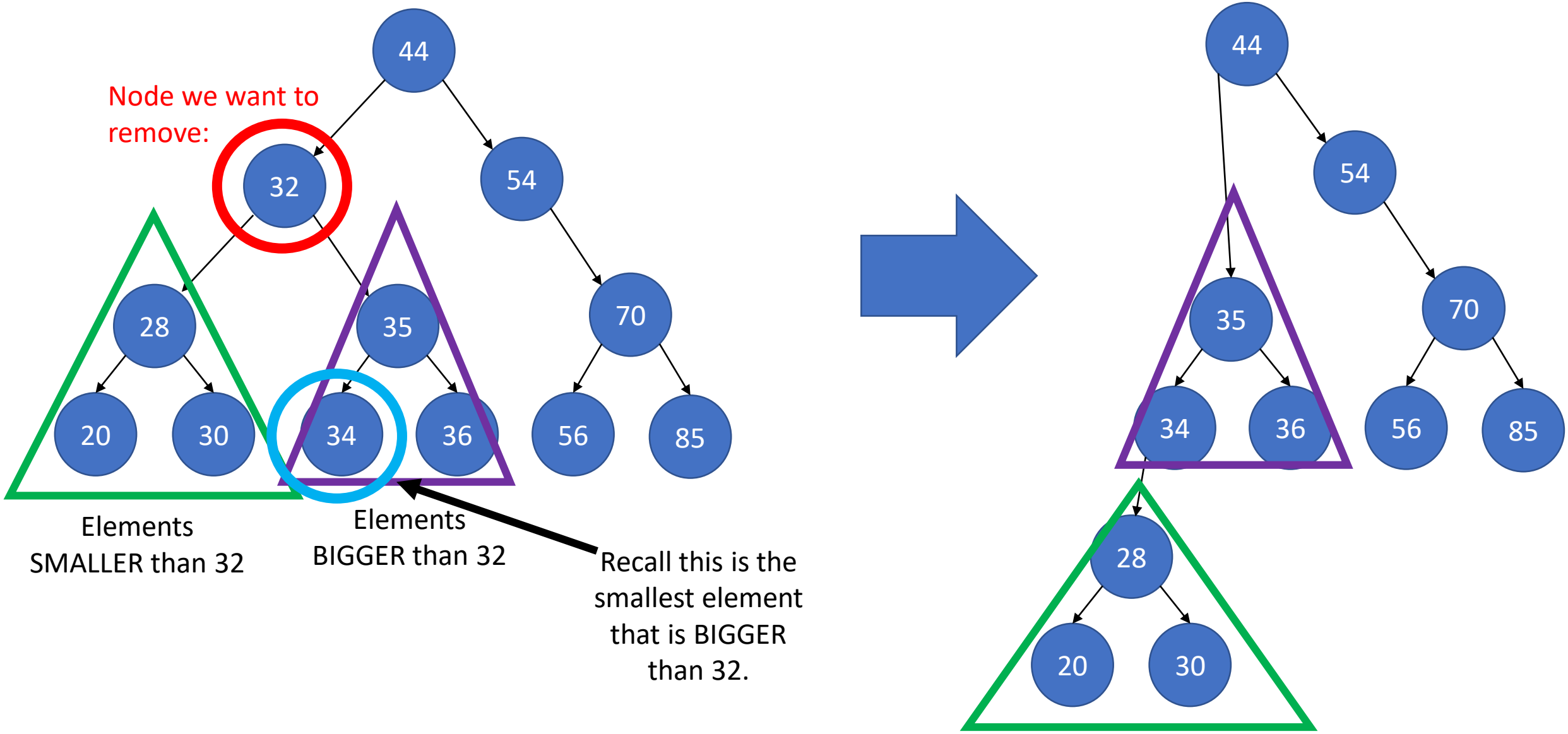
Simple answer: It doesn't matter because the structure of the binary tree is still preserved.

Case 3: The node has both left and right subtrees

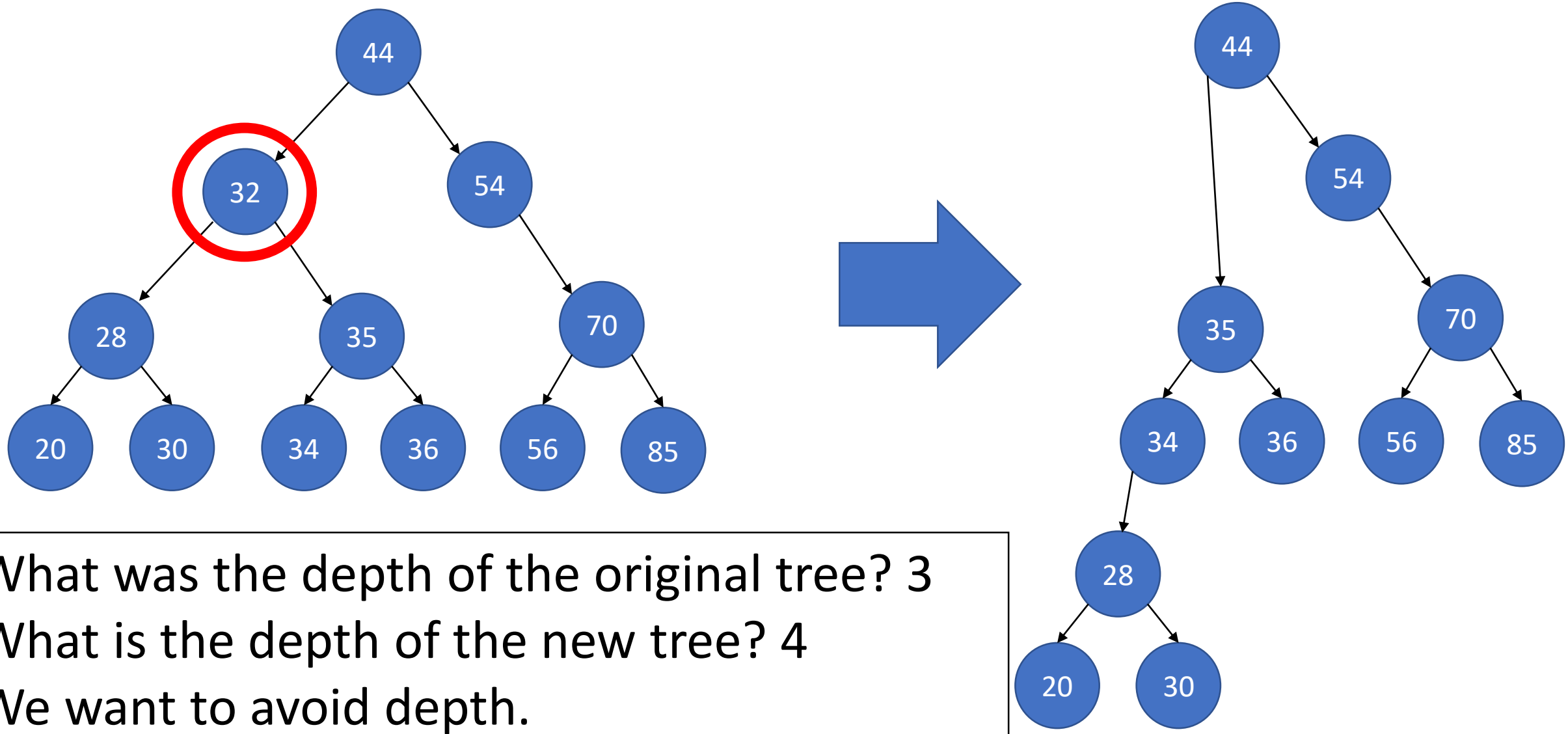


That feeling when you don't know which node to use:

Case 3: Solution A

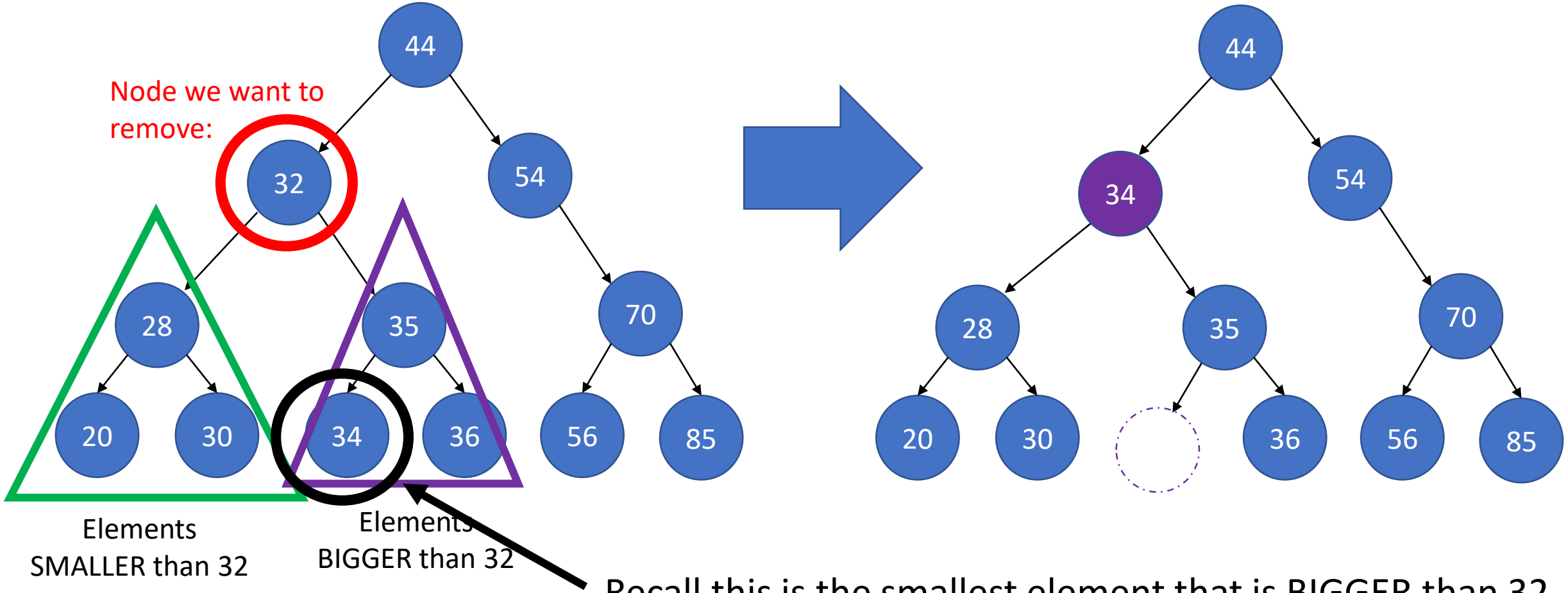


Case 3: Solution A, any problems?



What was the depth of the original tree? 3
What is the depth of the new tree? 4
We want to avoid depth.

Case 3: Solution B



Recall this is the smallest element that is BIGGER than 32. It is the smallest value in the RIGHT subtree. However it is also bigger than all the values in the LEFT subtree. Can we use it as a replacement node?

Summary of Tree Removal

1. The node has no children: Just replace the node with the empty list.
2. The node only has one child: Connect the child to the rest of the tree.
3. The node has two children: Connect the root of the left tree to the smallest element in the right tree.

OR replace the node to be removed with the smallest element in the right tree (which will be a leaf) and remove the smallest element in the right tree (which will follow case 1).

ANOTHER NATURAL TREE STRUCTURE

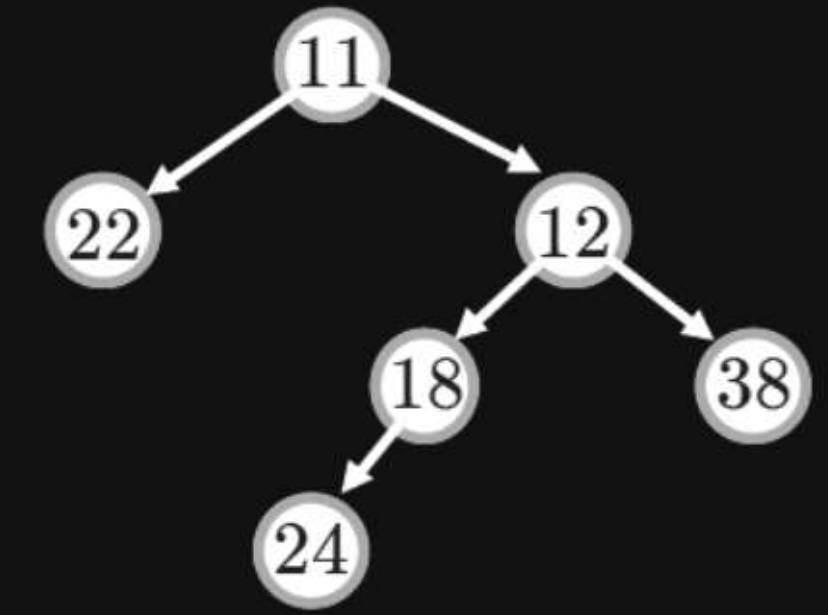
- A *heap* is a tree, with numbers stored at the nodes, with a different property:

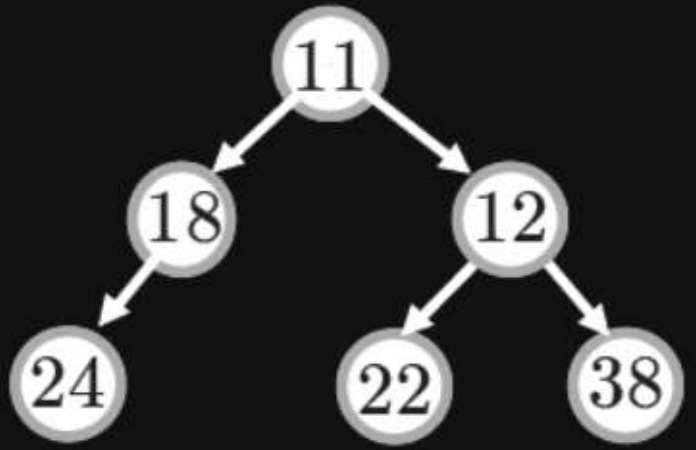
> **Heap Property**: The value of any node is smaller than that of its children.

- Notice that it is easy to determine the minimum element of a heap
  - it's always the root!
- If we can find a way to remove the minimum element and retain the heap property, we could use a heap for sorting. How?
  - I. Build a heap,
  - II. Repeatedly extract the minimum element.

AN EXAMPLE

- Consider the following heap containing the set $\{11, 12, 18, 22, 24, 38\}$
- Note that as with binary search trees, there are many different heaps containing the same set.

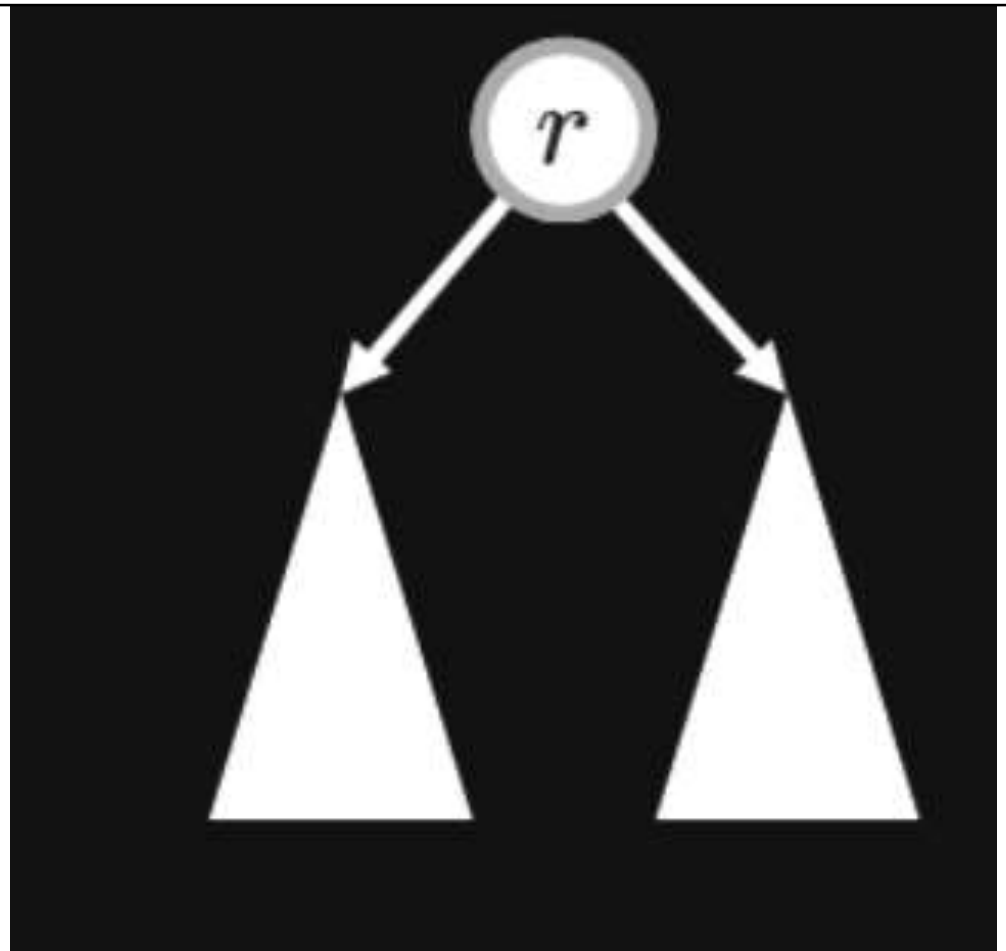




BUILDING BALANCED HEAPS

- Forcing a binary search tree to be balanced requires delicate algorithmic work.
- Heaps guarantee a weaker property: it's easy to construct a balanced heap.
- Idea: *Take turns choosing the insertion subtree.*

Insert an element bigger than r. Where should it go? Well if we inserted in the left subtree last time use the right, if we inserted in the right last then use the left.

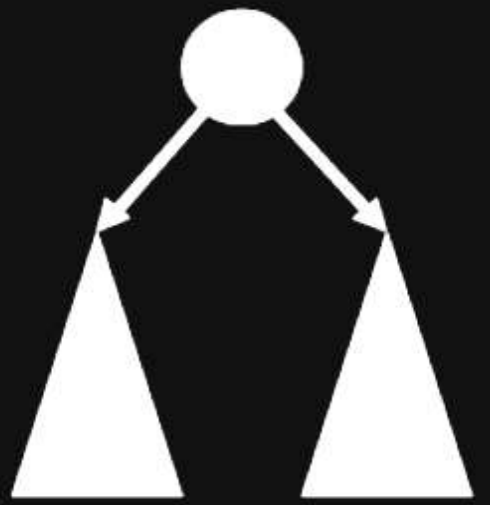


ALTERNATING INSERTION

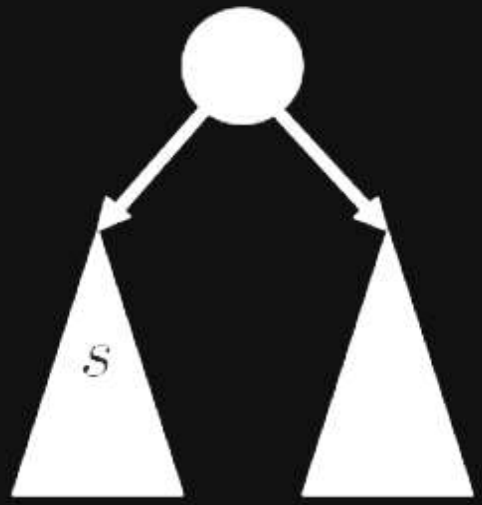
- In general, an insertion will require insertion into a subtree.
- However, *we have the luxury of choosing which tree.*
- If we alternate, half of the elements will be handed down to each child, and the result will be balanced!

- Implementation: Insert into the left tree; exchange them! Why does this maintain the heap property?

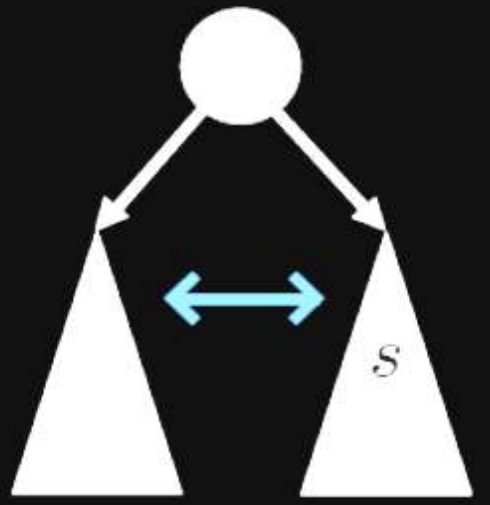
Problem: Insert s into a subtree



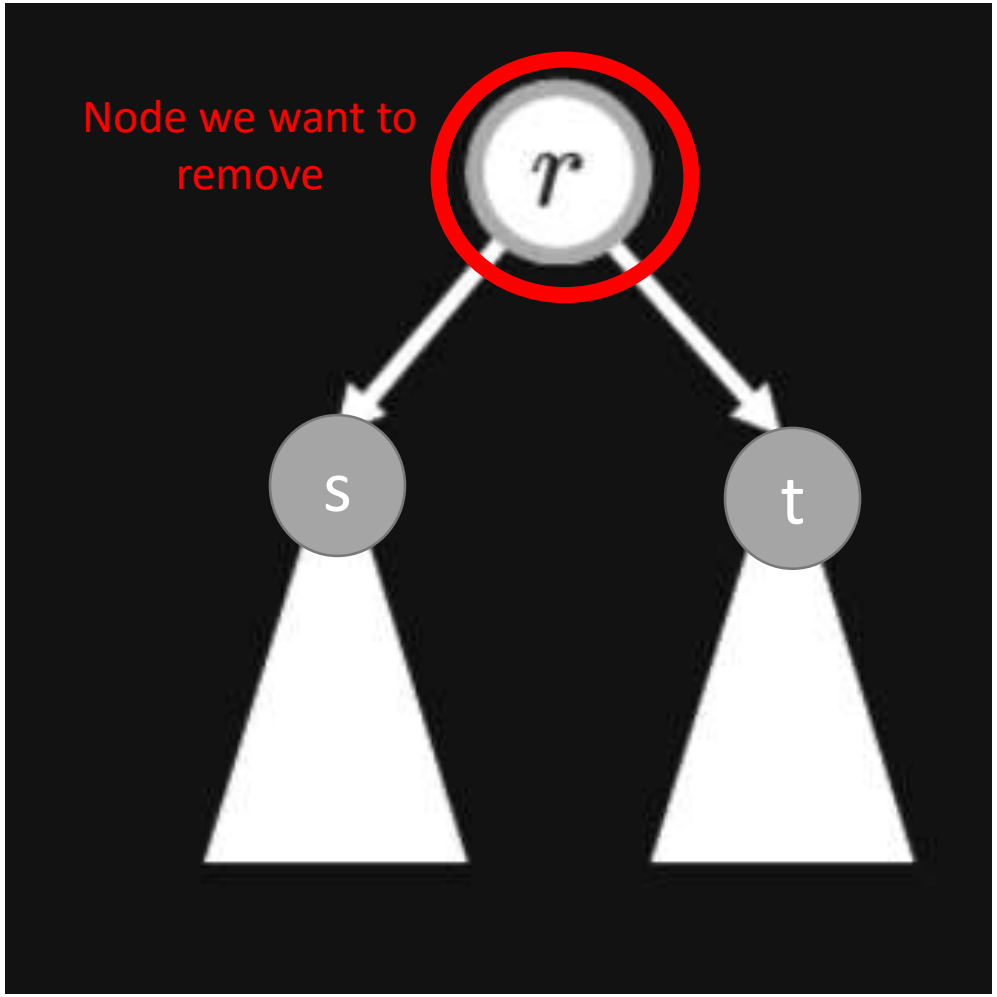
Insert into left tree



swap



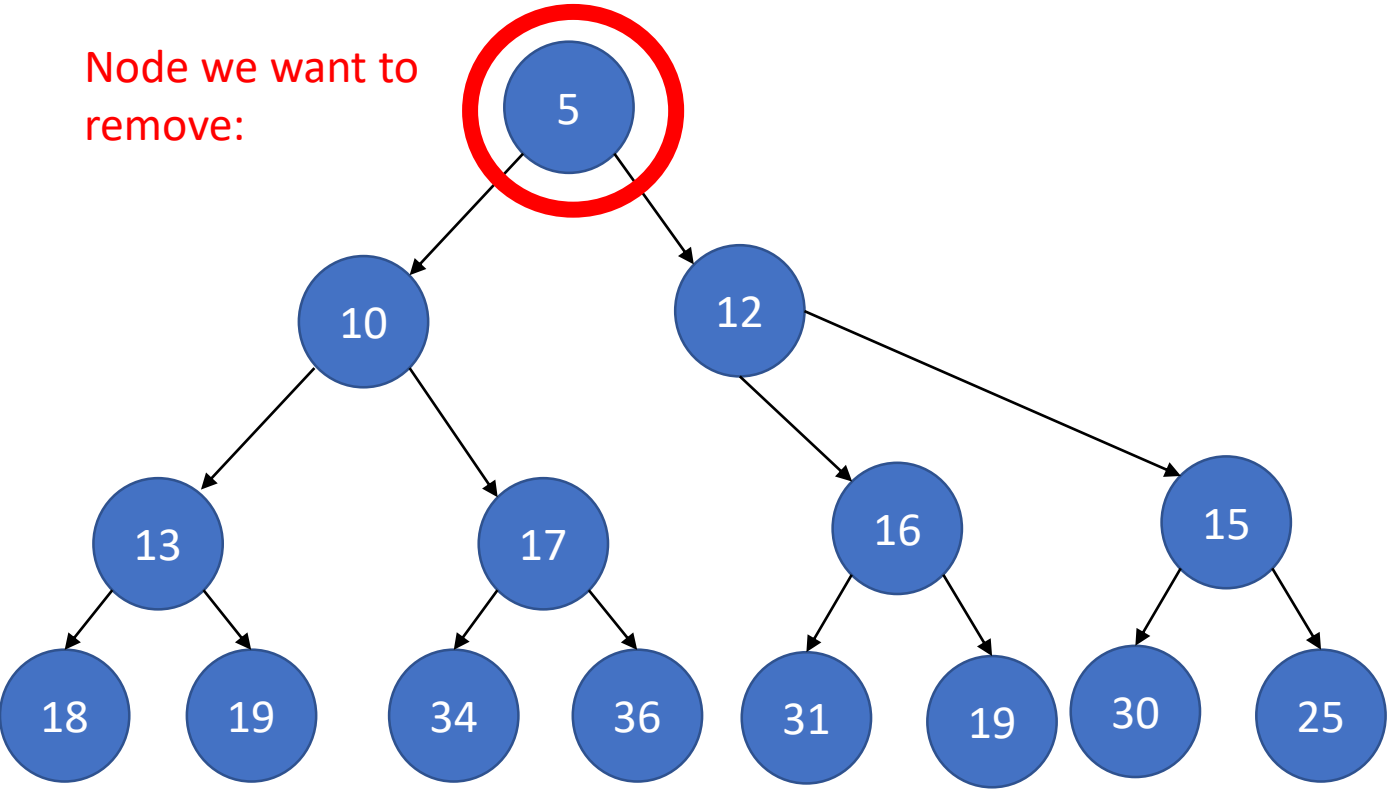
Removing the minimum from a heap. What to do?



Check s and t. The smaller one becomes the root.

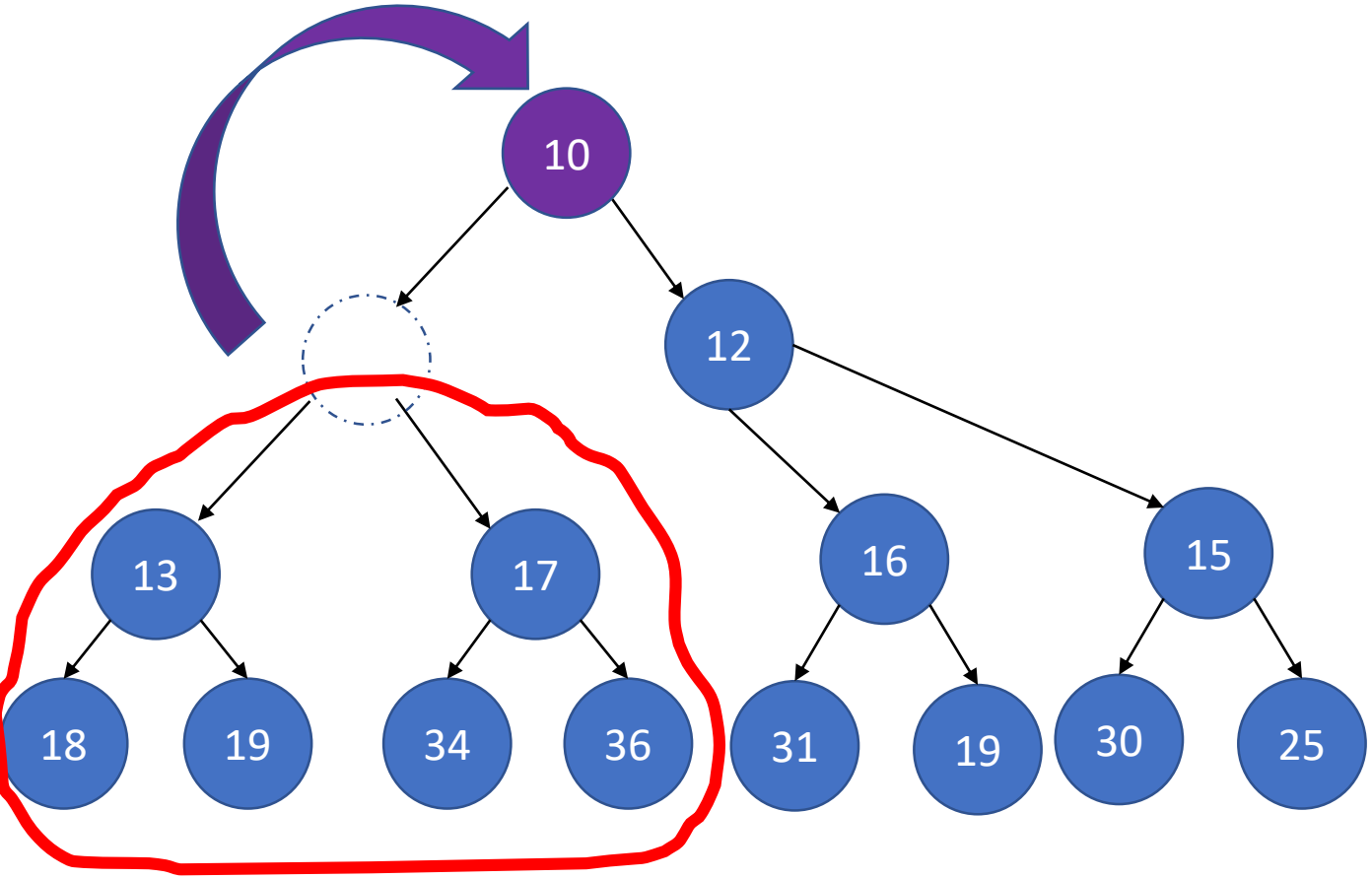
Are we done?

Not done yet! Take a simple example to see the issue…



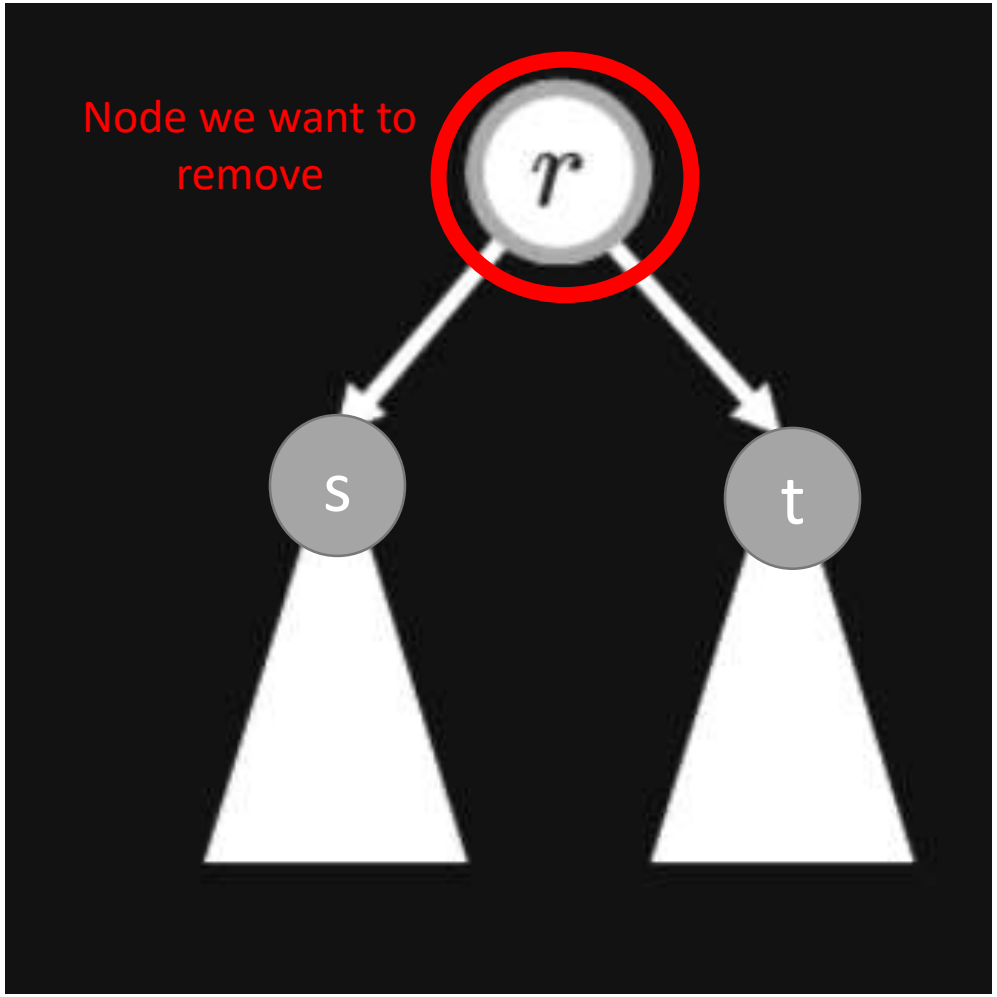
Side note: Is this a balanced heap?
Because it was very frustrating to draw.

Remove the minimum (current root)

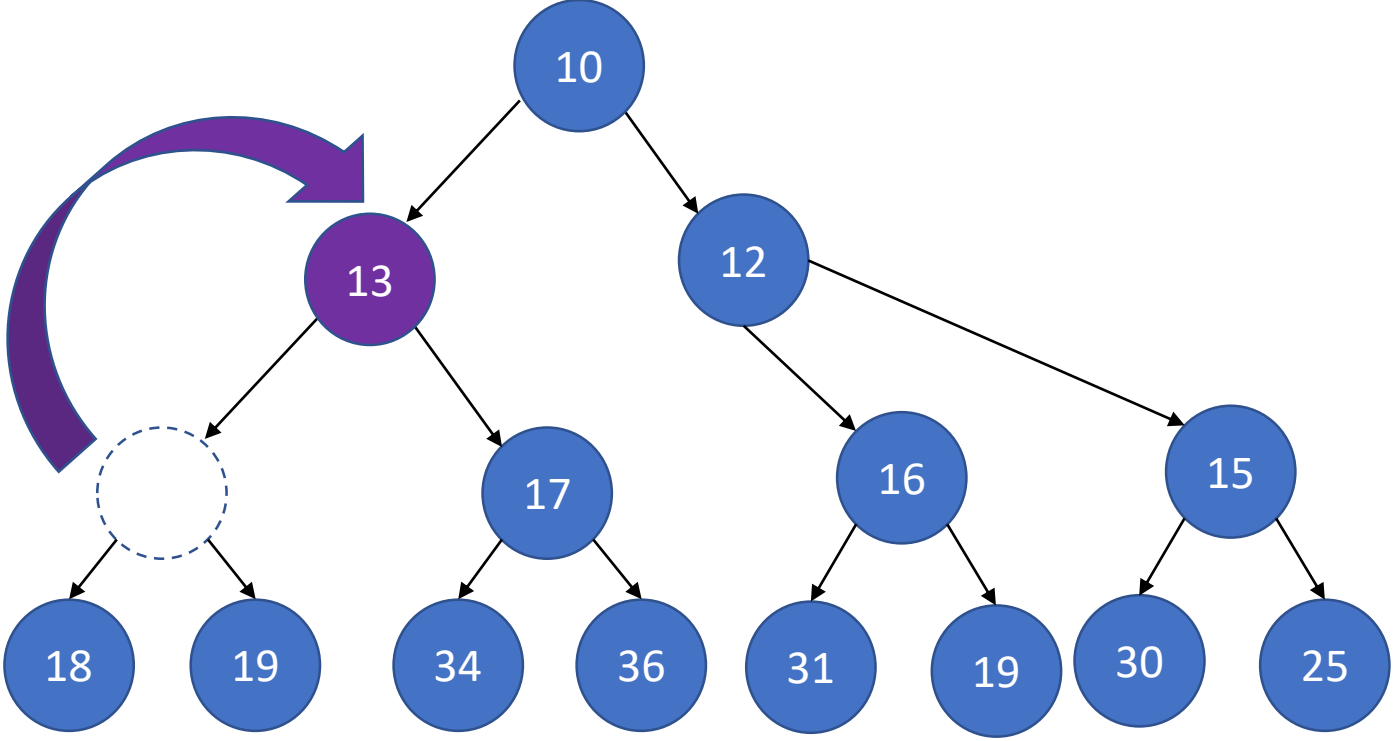


Problem: At least half of the tree is not organized properly…

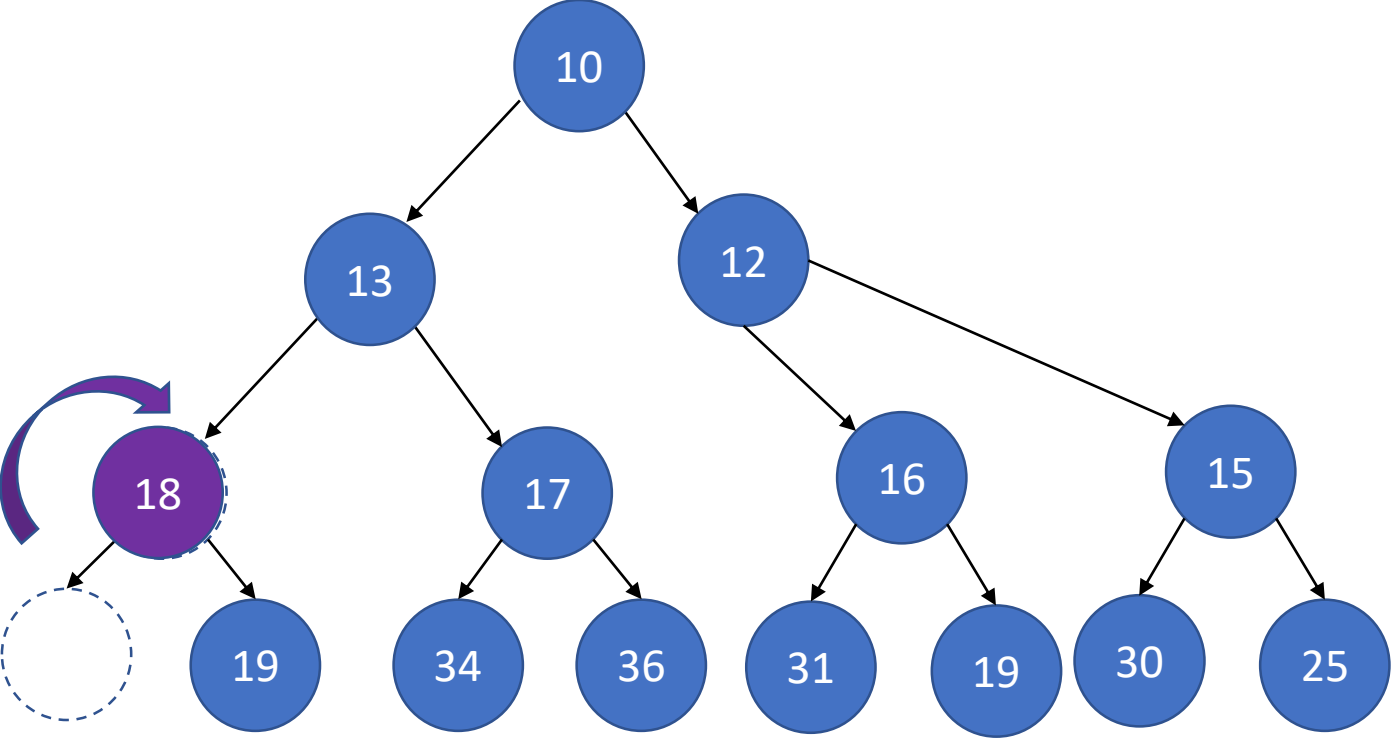
Solution: Reorganize the sub-heap using the exact same logic!



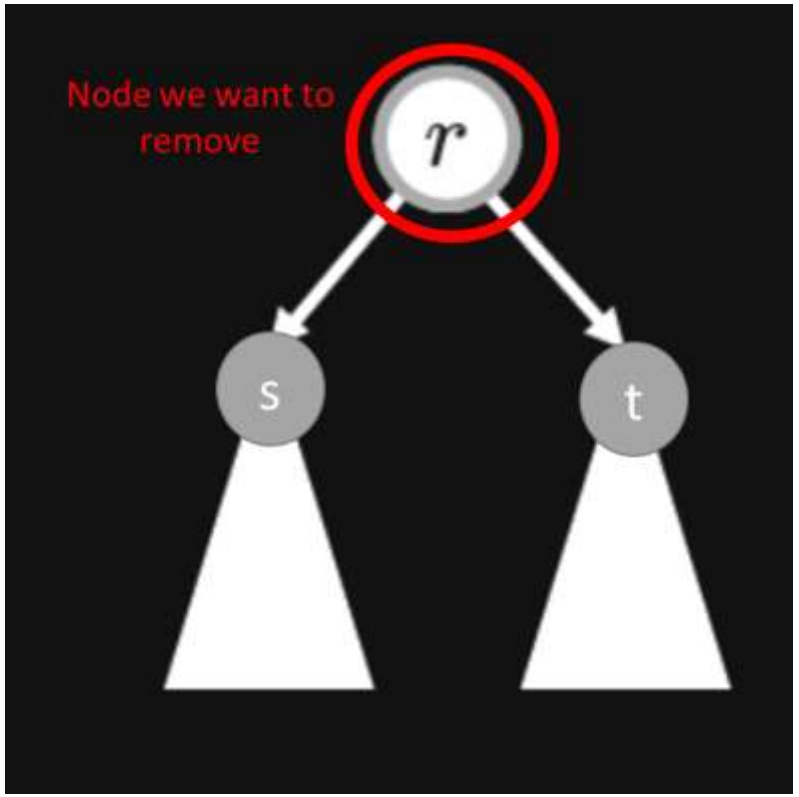
Check s and t. The smaller one becomes the root.



Still have a problem: At least have of the tree is not organized properly…



Repeat the same solution on the next sub-part of the heap…

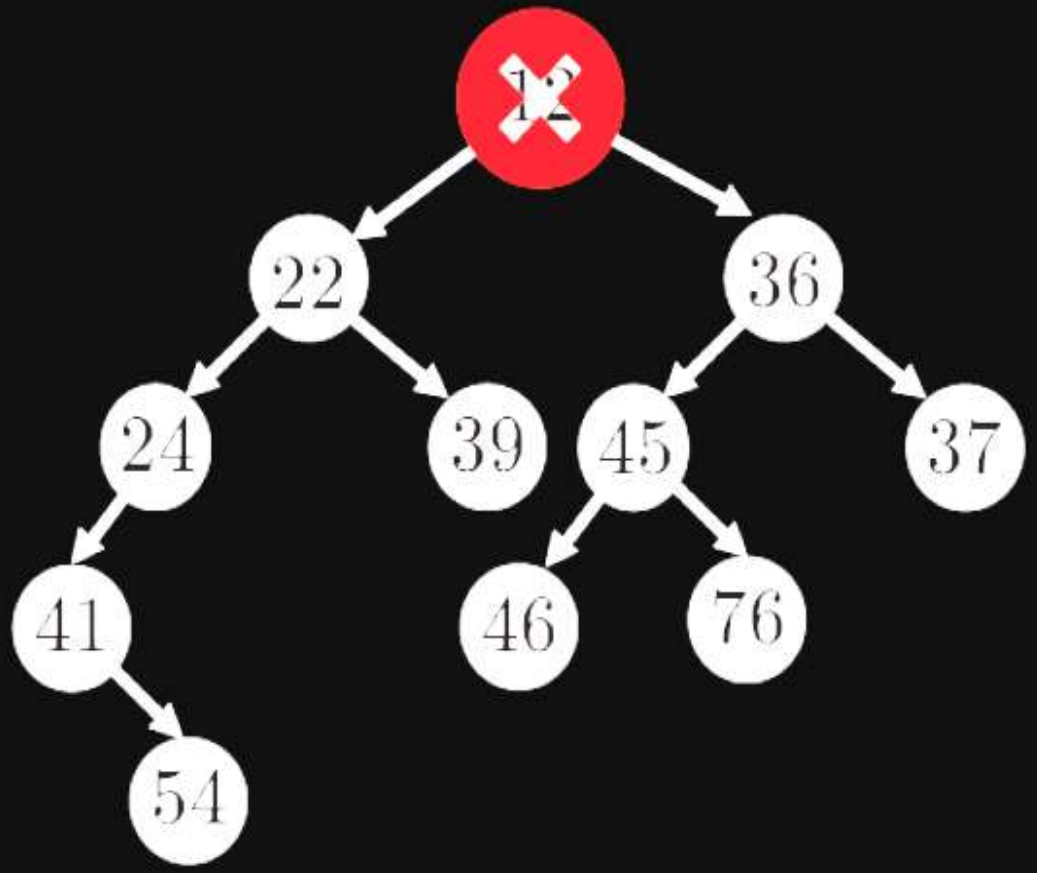


REMOVING THE MINIMUM ELEMENT FROM A HEAP

- Let $\min(H)$ denote the minimum element of a heap H.
- Find minimum value: $\mathrm{Findmin}(H)$. Easy! Return the value of the root.
- Remove minimum value: $\mathrm{Removemin}(H)$: Trickier.
  - Remove the top element. Two heaps result: H_1, H_2.
    - Problem: How do you combine them into a single heap?
  - Suppose $\min(H_1) < \min(H_2)$. Place $min(H_1)$ at top; its children will be H_2 and $\mathrm{Removemin}(H_1)$. A nice recursive rule.
  - (Other cases? What if one of H_i is empty?)

REMOVING THE MINIMUM ELEMENT: ANOTHER VIEW

- Remove the smallest element.
- Promote its smallest child, if it has one.
- Repeat on the subtree of the promoted child.



IMPLEMENTING HEAPS IN SCHEME

- As with trees, we represent a heap as a list containing a value, and two heaps (the "left subheap" and the "right subheap").

```
(define (create-heap v H1 H2)
  (list v H1 H2))
```

- Helper functions for accessing the value and the subheaps:

```
(define (h-min H) (car H))
(define (left H)  (cadr H))
(define (right H) (caddr H))
```

INSERTION

- **Note how the insert & exchange trick works:** We *alternate* insertion into subheaps.
- To implement this: always insert into left subheap, then exchange them.

```
(define (insert x H)
  (if (null? H)
      (create-heap x '() '())
      (let ((child-value (max x (h-min H)))
            (root-value  (min x (h-min H))))
        (create-heap root-value
                     (right H)
                     (insert child-value (left H))))))
```

Let’s break down this code…

First part: If the heap is empty, create a new heap:

```
(define (create-heap v H1 H2)
  (list v H1 H2))
```

```
(define (insert x H)
  (if (null? H)
      (create-heap x '() '())
```

Second part:
Assign the maximum value to “child-value”
This value is either the current element you want to insert (x) or the minimum of the heap depending on which is larger.

```
(define (insert x H)
  [redacted]
    (let ((child-value (max x (h-min H))))
      [redacted]
```

Third part:
Assign the minimum value to “root-value”
This value chosen between the minimum of the heap and current value being inserted (x).

```
(define (insert x H)
        (root-value  (min x (h-min H))))
```

- Basically these two pieces of code together: IF the current element (x) is smaller, it becomes the new root and you need to put the old root somewhere in the heap.
- Otherwise the current element is assigned to the child value.
- In either case the child value is what is going to be inserted into the heap.

```
(define (insert x H)
        (let ((child-value (max x (h-min H)))
              (root-value  (min x (h-min H))))
```

- Make the heap with the root.
- Now here we call create-heap function BUT the left part of the heap is replaced with the right part of the heap to swap.

```
(define (insert x H)
  [redacted]
        (create-heap root-value
                     (right H)
                     (insert child-value (left H))))))
```

Pay attention. Swap between left and right occurring!

```
(define (create-heap v H1 H2)
  (list v H1 H2))
```

- Recursively call the insert function with one side of the heap and the child value.

```
(define (insert x H)
                                    (insert child-value (left H))))))
```

Next Problem:
How to remove the minimum from a heap

```
(define (combine-heaps H1 H2)
  (cond ((null? H1) H2)
        ((null? H2) H1)
        ((< (h-min H1) (h-min H2))
         (create-heap (h-min H1)
                      H2
                      (combine-heaps (left H1) (right H1))))
        (else
         (create-heap (h-min H2)
                      H1
                      (combine-heaps (left H2) (right H2))))))
```

- Then

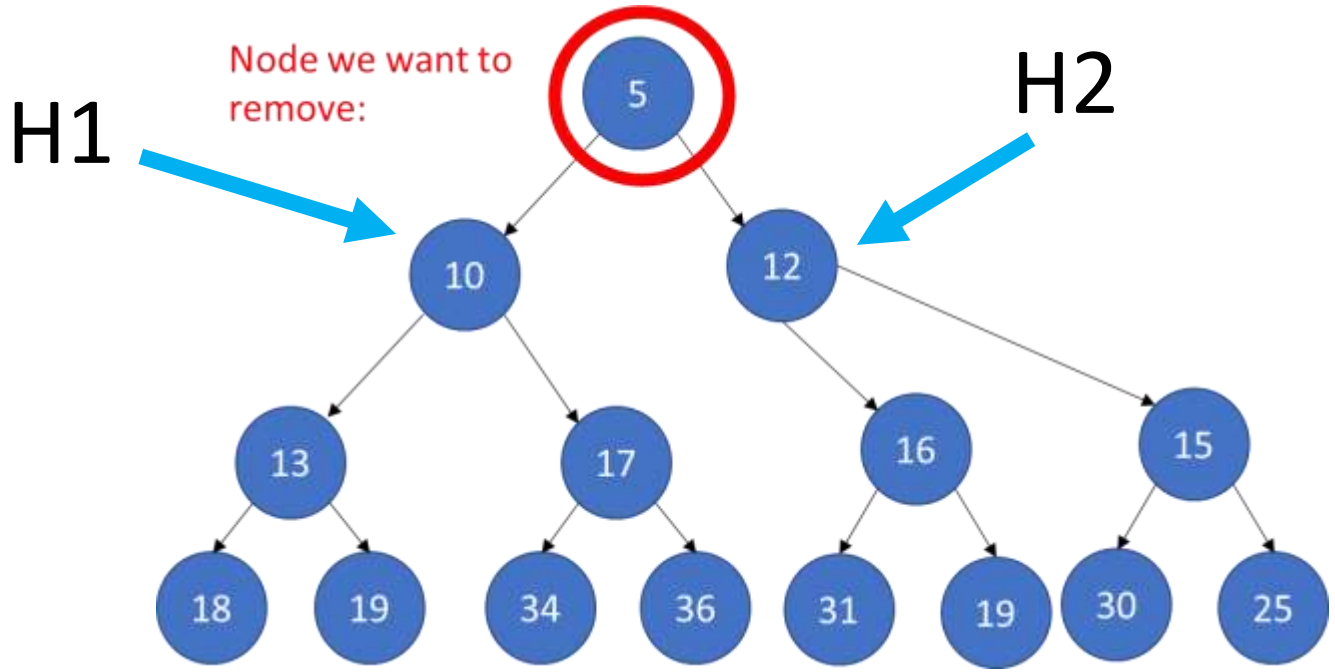
```
(define (remove-minimum H)
  (combine-heaps (left H) (right H)))
```

Let's break down this code…

How to remove the minimum from a heap

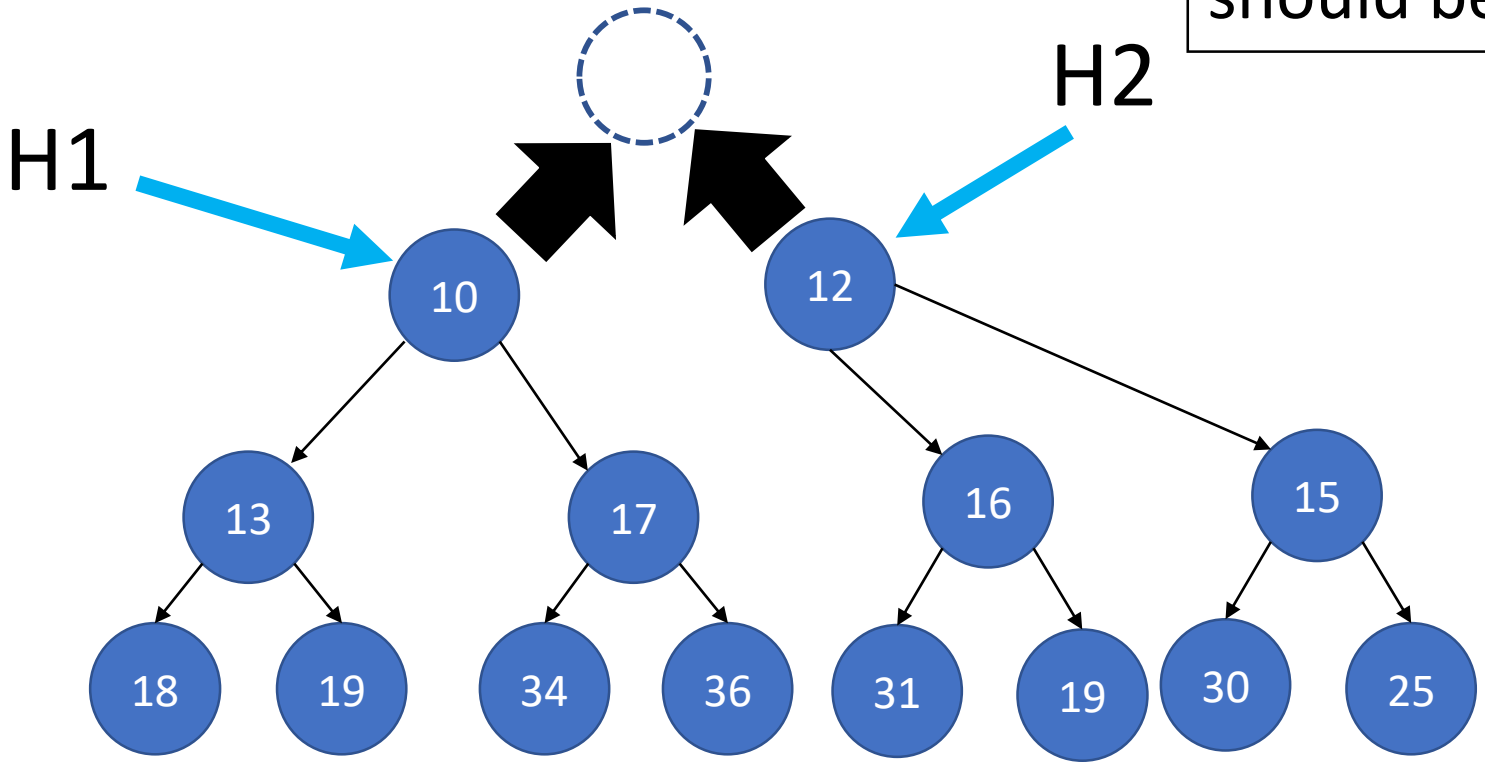
```
(define (create-heap v H1 H2)
  (list v H1 H2))
```

First where is the minimum? Well it is always the top of the heap. So we know it is v. Now recall that we have to combine the two remain parts of the heap to make one heap…

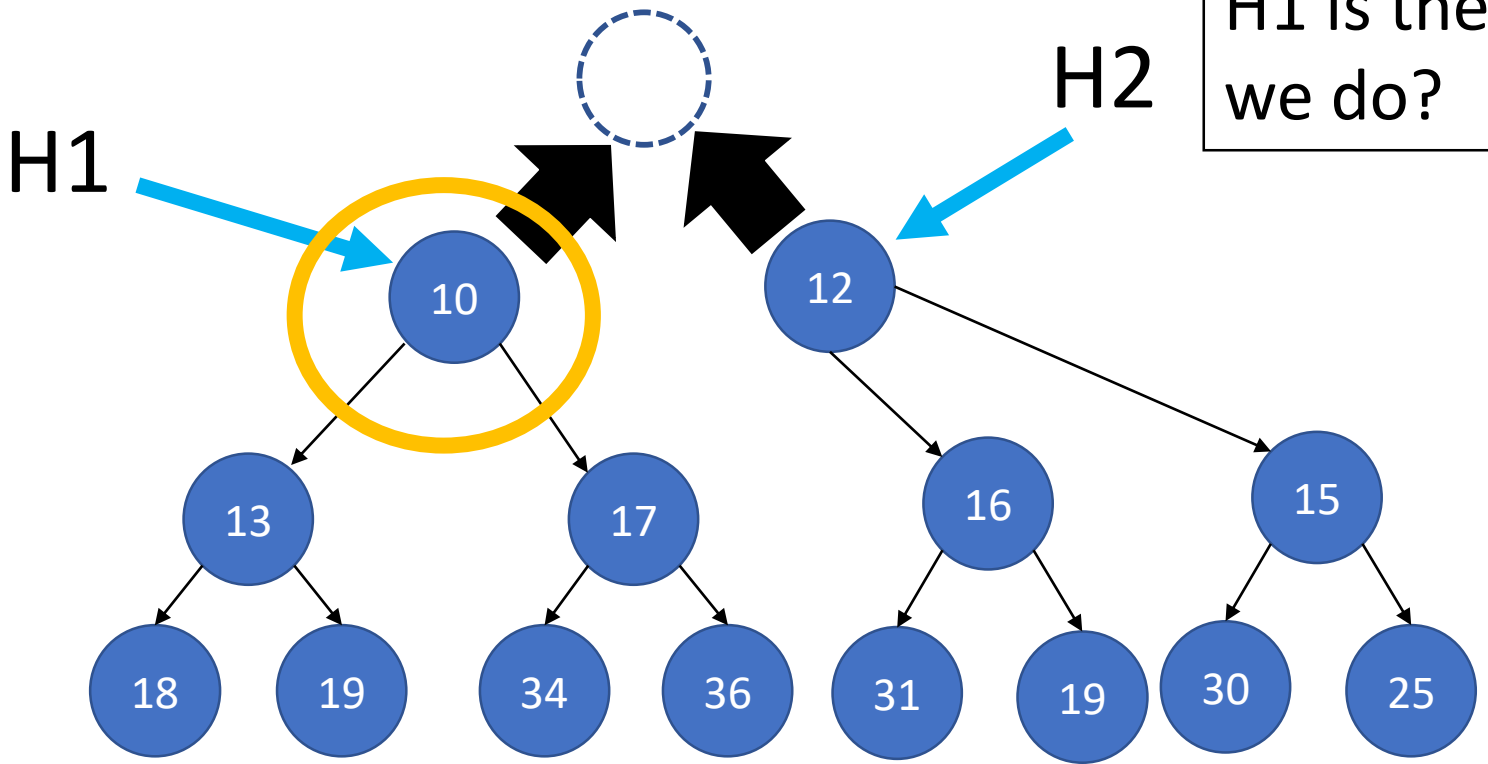


```
(define (combine-heaps H1 H2)
  (cond
    ((< (h-min H1) (h-min H2))
     (else
```

First is just the logic: We removed the top and now we need to figure out which part of the heap should be the new top.

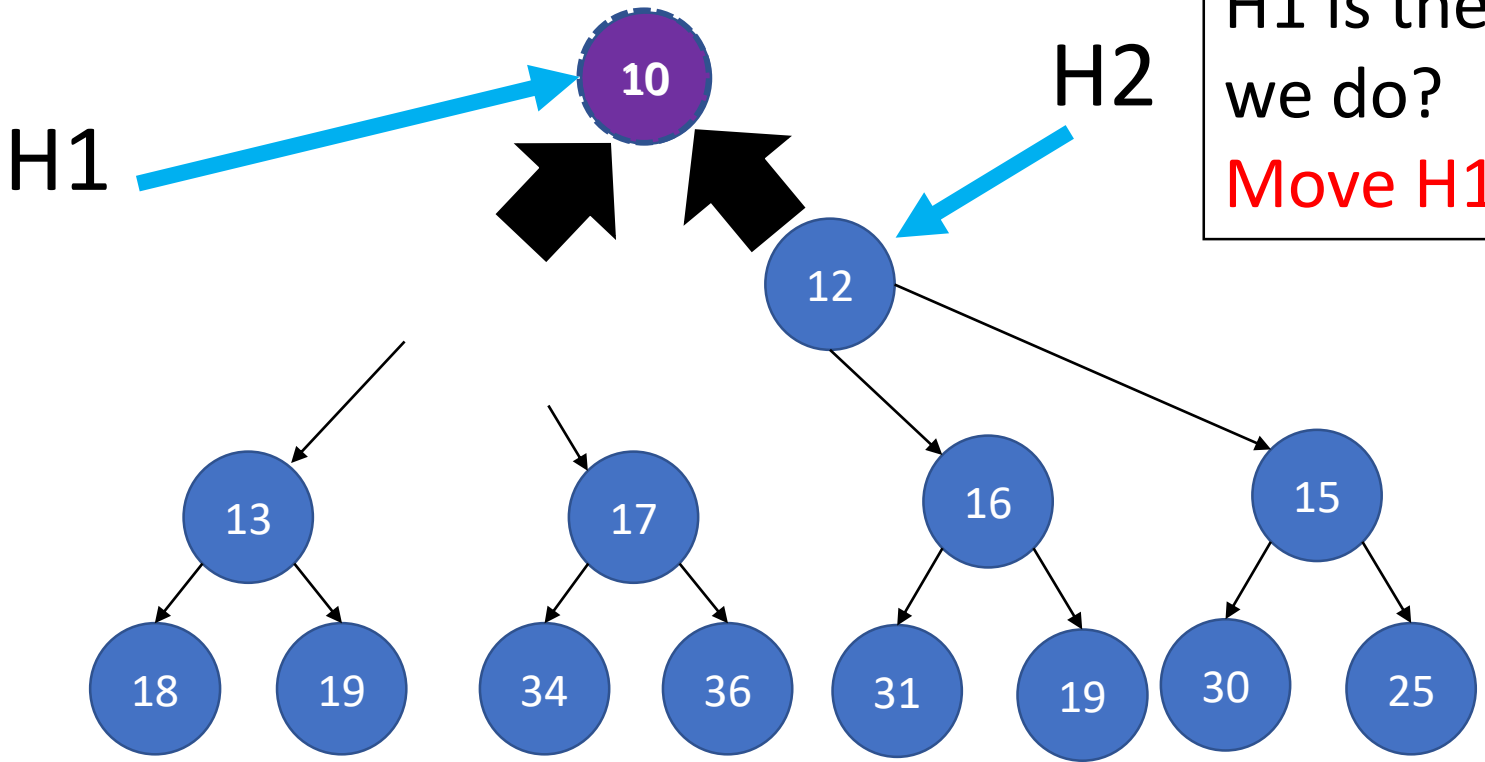


```
(define (combine-heaps H1 H2)
  [redacted]
        ((< (h-min H1) (h-min H2))
         (create-heap (h-min H1)
                      H2
                      (combine-heaps (left H1) (right H1))))
    [redacted]
```



H1 is the top. So what do we do?

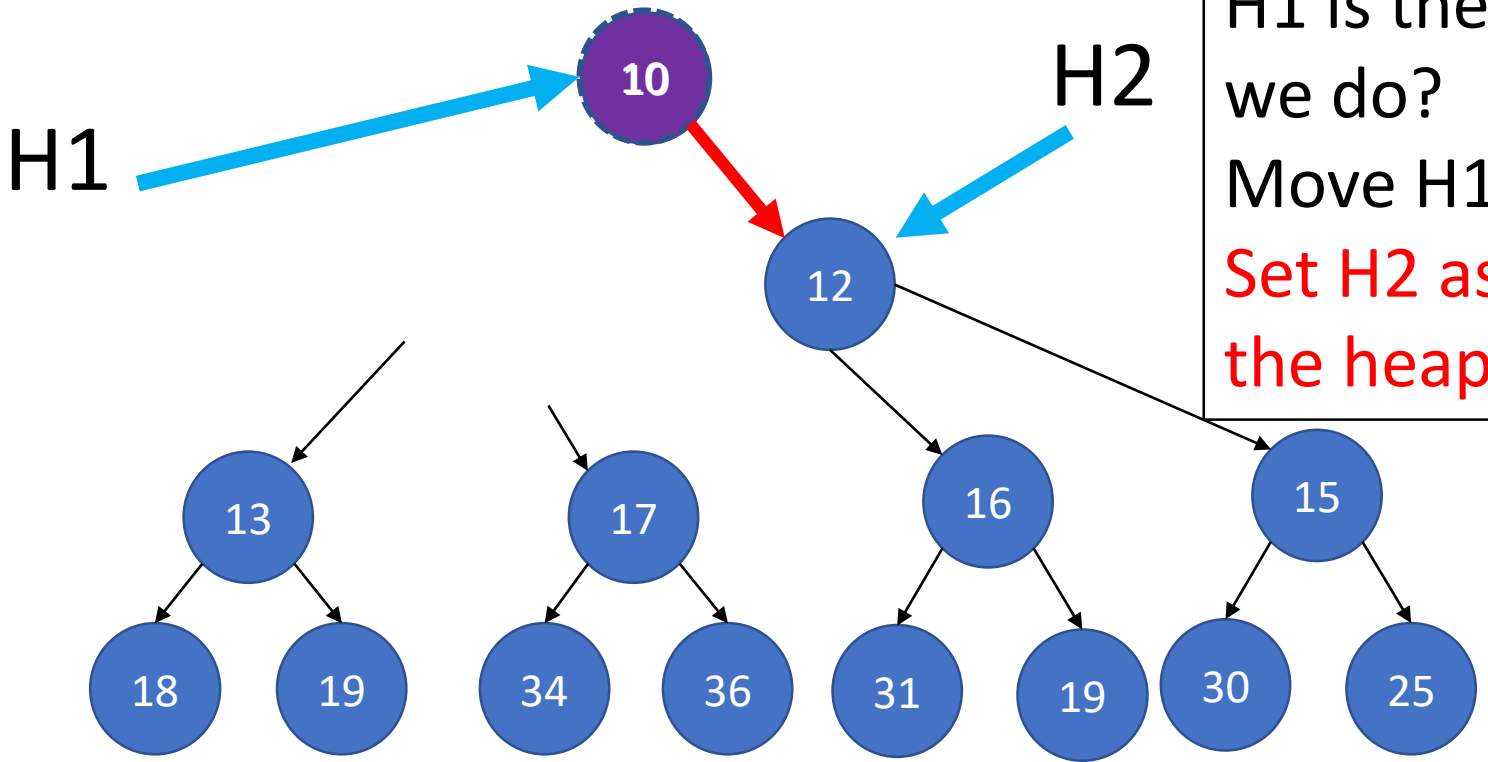
```
(define (combine-heaps H1 H2)
  [redacted]
    (create-heap (h-min H1)
                 H2
                 (combine-heaps (left H1) (right H1))))
  [redacted]
```



H1 is the top. So what do we do?

Move H1 to the top.

```
(define (combine-heaps H1 H2)
  [redacted]
  (create-heap (h-min H1)
               H2
               (combine-heaps (left H1) (right H1))))
  [redacted]
```

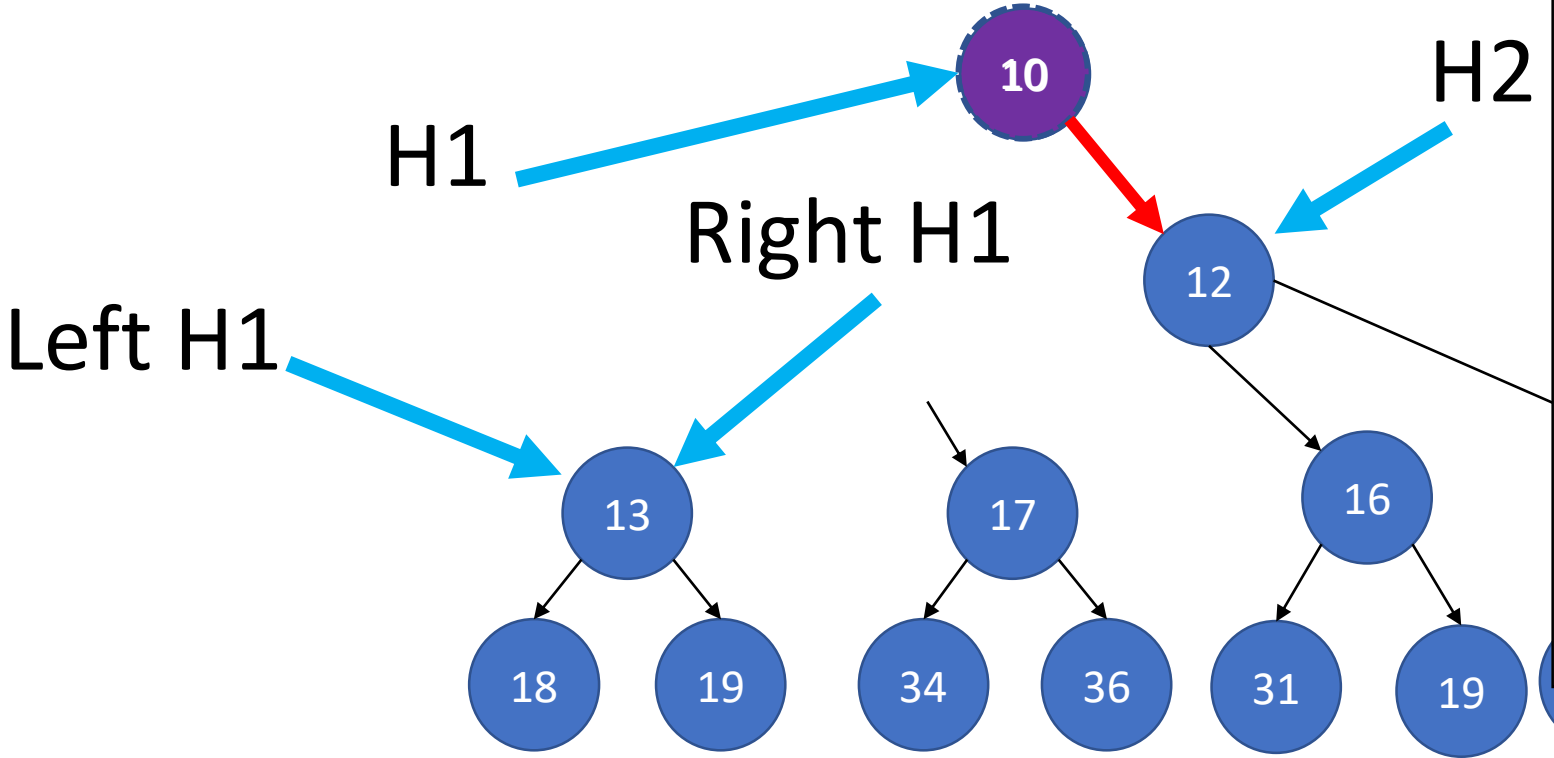


H1 is the top. So what do we do?

Move H1 to the top.

Set H2 as the left part of the heap.

```
(define (combine-heaps H1 H2)
  (create-heap (h-min H1)
               H2
               (combine-heaps (left H1) (right H1))))
```



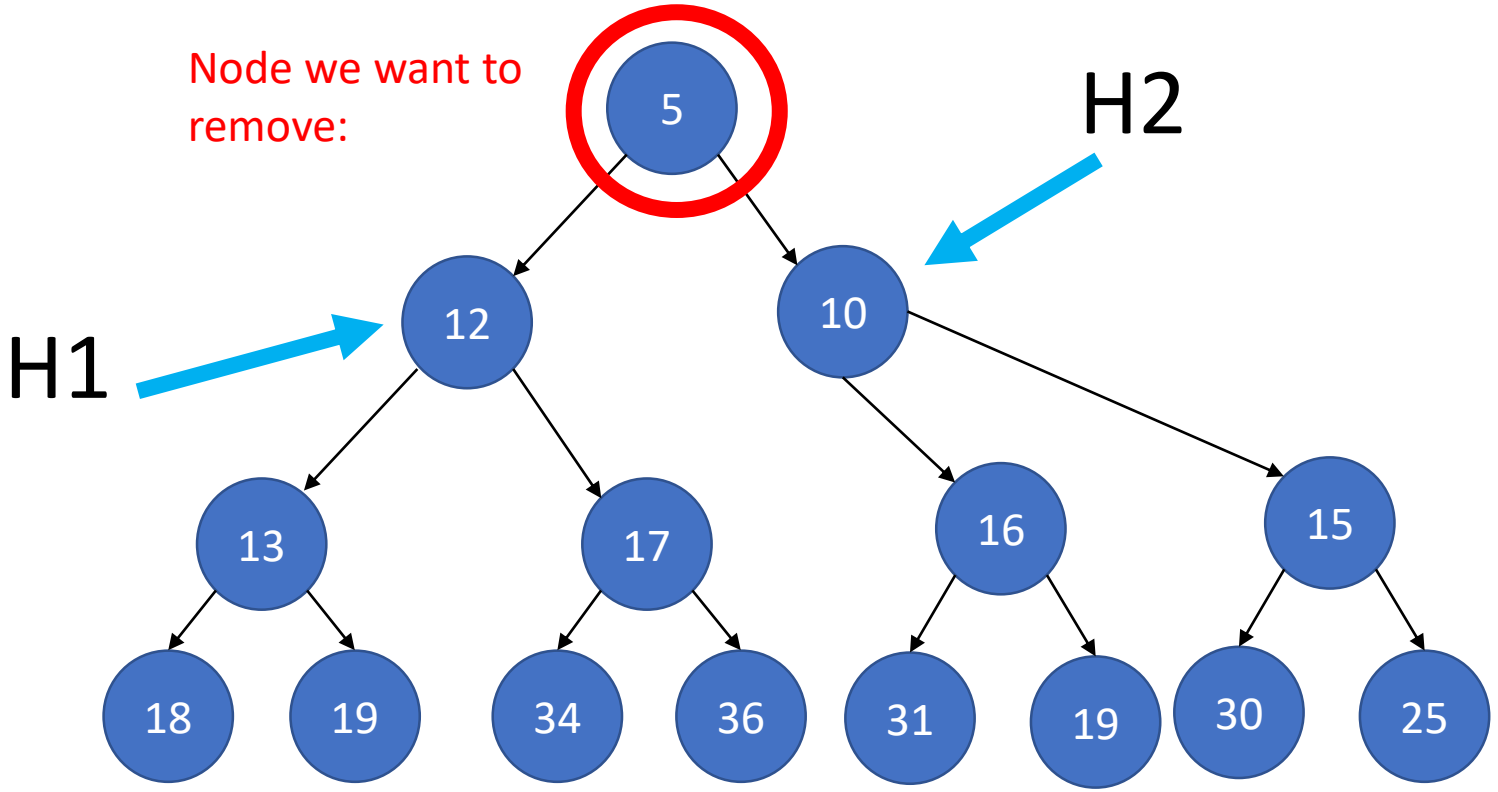
H1 is the top. So what do we do?

Move H1 to the top.

Set H2 as the left part of the heap.

Call combine heap to rebuild the left part of the heap with left(H1) and right (H1)

What would happen if H2 was actually smaller?



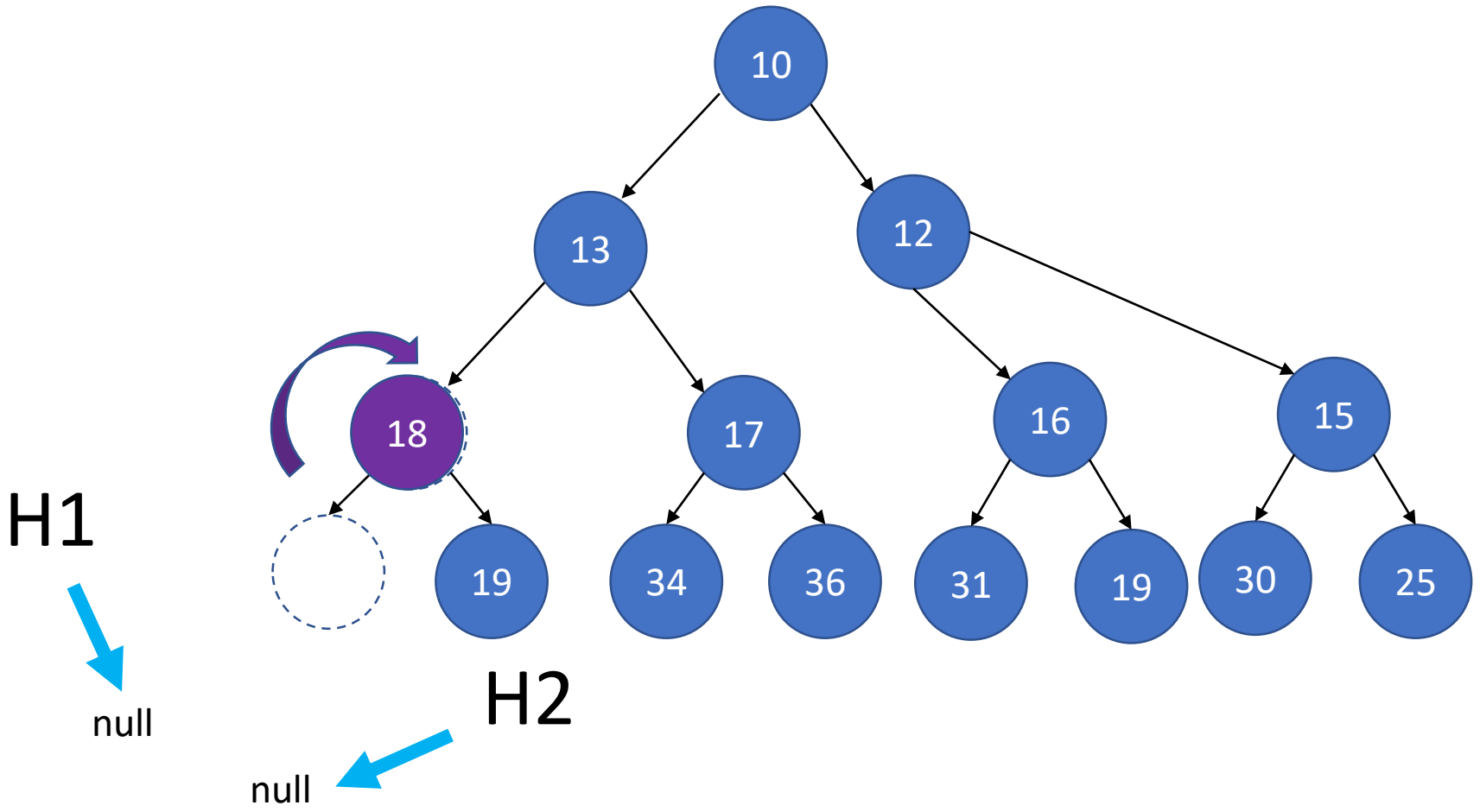
Note I just flipped H1 and H2 from the previous diagram to get this new heap.

What would happen if H2 was actually smaller?

Short Answer: Do the same thing we did before but with H2 instead of H1.

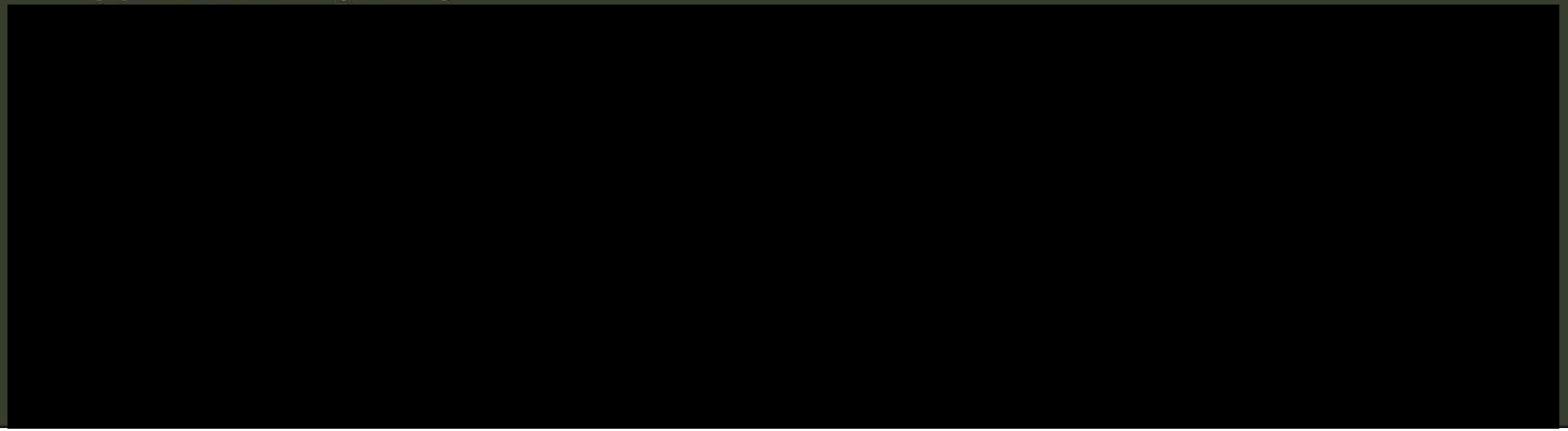
```
(define (combine-heaps H1 H2)
  [redacted]
        (else
         (create-heap (h-min H2)
                      H1
                      (combine-heaps (left H2) (right H2))))))
```

Lastly: What is the base case for this code?



Lastly: What is the base case for this code?

```
(define (combine-heaps H1 H2)
  (cond ((null? H1) H2)
        ((null? H2) H1)
```



The full remove min code. Not so scary now right?

```
(define (combine-heaps H1 H2)
  (cond ((null? H1) H2)
        ((null? H2) H1)
        ((< (h-min H1) (h-min H2))
         (create-heap (h-min H1)
                      H2
                      (combine-heaps (left H1) (right H1))))
        (else
         (create-heap (h-min H2)
                      H1
                      (combine-heaps (left H2) (right H2))))))
```

- Then

```
(define (remove-minimum H)
  (combine-heaps (left H) (right H)))
```

HEAPS IN THE WILD: SORTING

- Heaps are used for sorting: Given n numbers…
  - **Insert** them into the heap (recall: it's easy to keep this balanced).
    - This takes no more than $\sim \log(n)$ time per element: $\sim n \log(n)$ time in total.
  - **Extract-min** n times.
    - This extracts the elements in sorted order.
    - Each extract min takes time no more than the depth of the tree $\sim \log(n)$.
    - So…this also takes about $\sim n \log(n)$ time.
- This gives a fast sorting algorithm:
  - Approximately $n \cdot log(n)$ operations to sort n elements.
  - In many settings, you cannot do better!

HEAPS IN THE WILD: PRIORITY QUEUES

- Maintain a collection of elements in setting where you only need to find the smallest. (Often called a *priority queue*.)
- For example: In an computing system, you may wish to maintain a collection of processing jobs that need to be completed, with priorities. Then there are two basic operations required:
  - Insert a new job in the system (with a particular priority, given by a number), and
  - Find, extract (and complete!) the job with the current highest priority.
- A heap is perfect for this!

Figure Sources

- https://miro.medium.com/max/1200/1*U4dZWeXgNNrYaedRCuzTIg.png
- https://timberworksva.com/wp-content/uploads/2019/06/2.-Tree-removal-3-e1561686697970-1080x675.jpg
- https://i.kym-cdn.com/entries/icons/original/000/027/916/hamster.jpg
- https://i.imgur.com/rGQN1Em.png