ECS32B

Introduction to Data Structures

AVL Trees

Heaps

Lecture 24

Announcements

- Have a good holiday weekend, nothing is due next week.
- Homework 6 will be posted when you get back.
- A sample final will be posted the last week of class.
- Last lecture will mostly be final exam preparation.

AVL trees

Three ideas we need to understand to work with AVL trees:

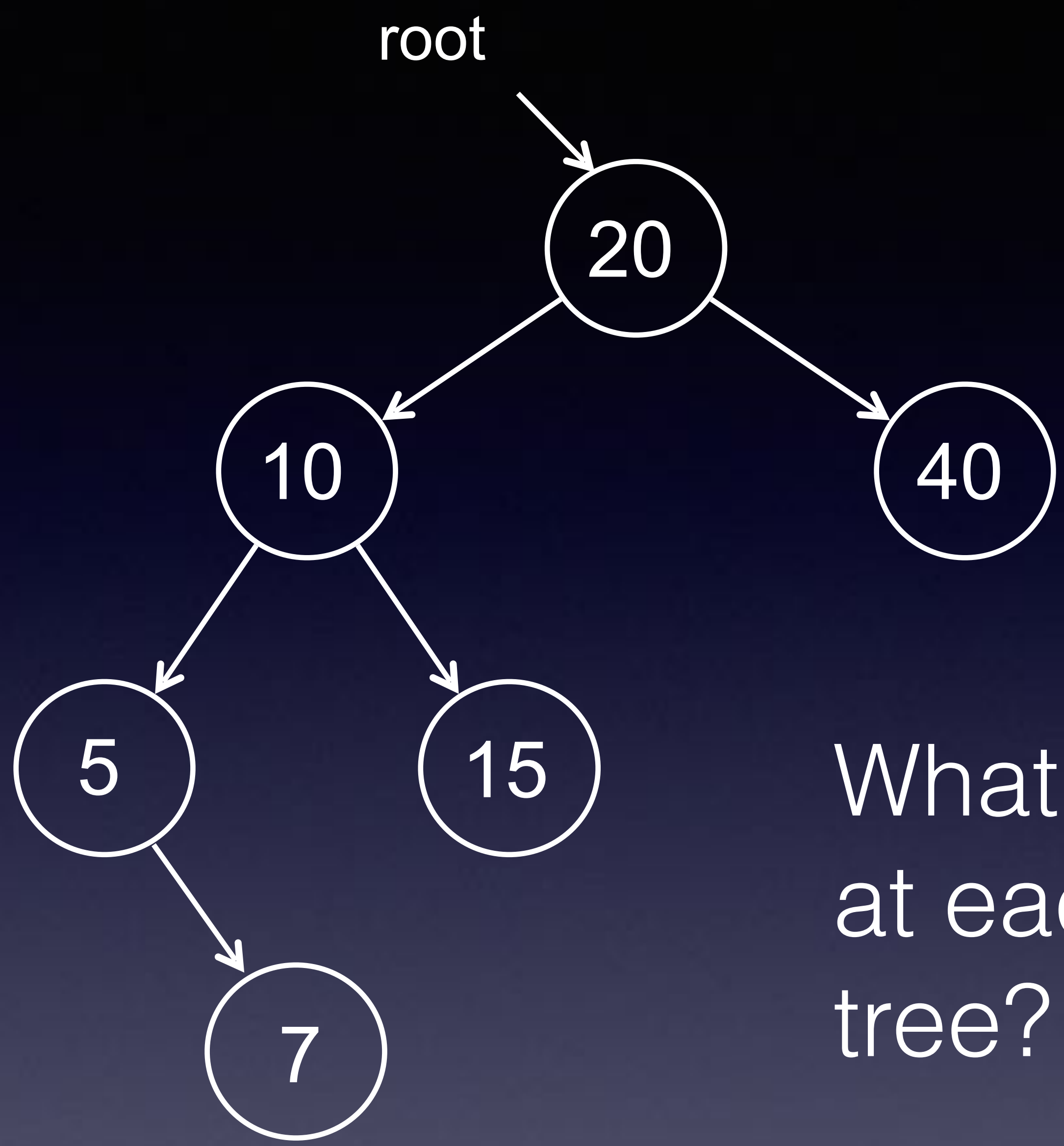
height **balance** **rotation**

AVL trees maintain **balance** by tracking the difference in **height** of each subtree.

If the balance is in the nominal range -1 to 1 for all nodes the height is as most 1.44 $\log_2(n)$ and therefore search is O(logn).

If the balance ever goes outside of the nominal range, **rotation** is applied to bring the balance back into range.

Rotate Right



What's the balance at each node of this tree?

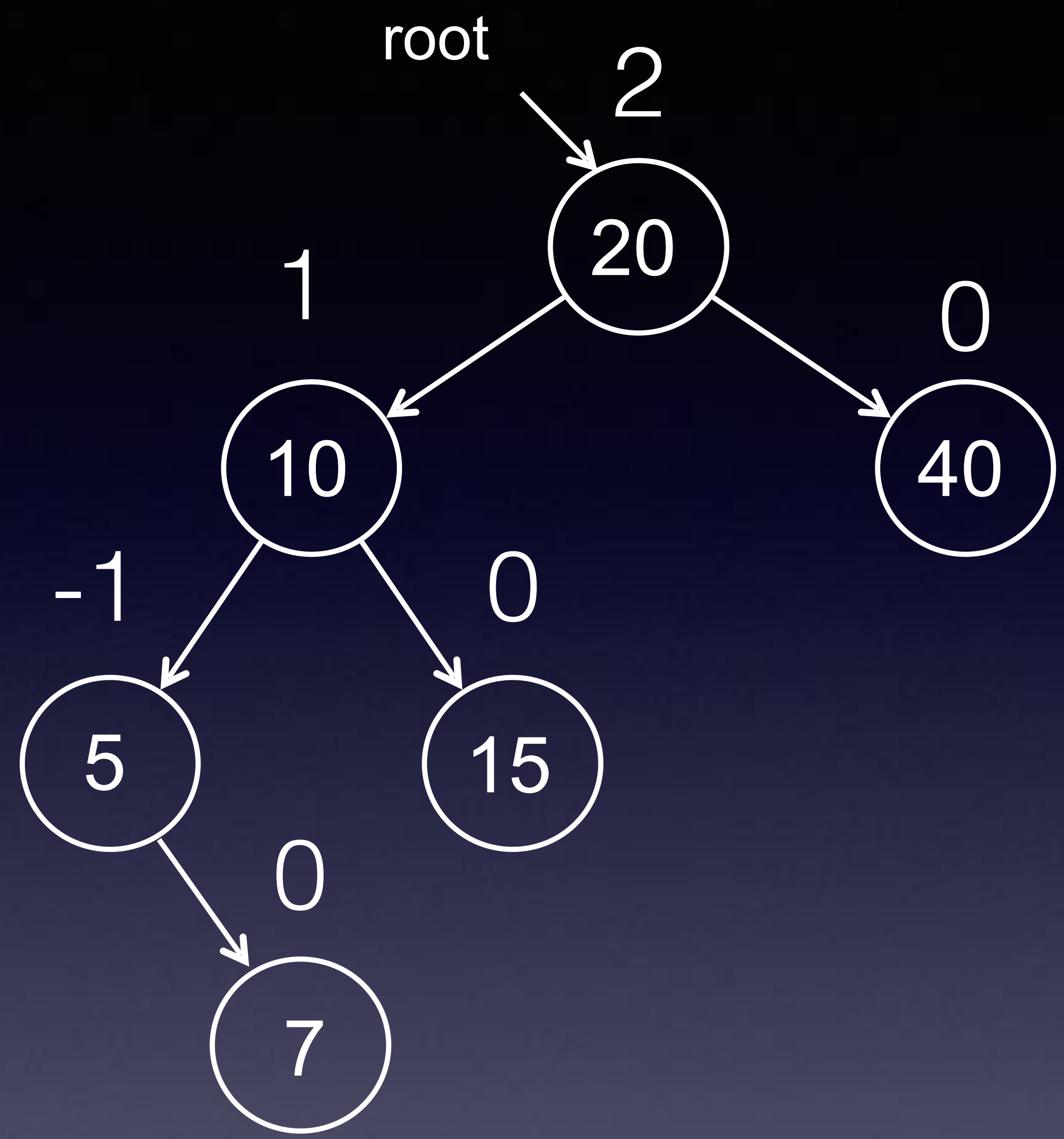
It's not magic. It's just code.

Here's the algorithm for right rotation:

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

Note: the notation "root->left" is short for "the left pointer from the root node".

Rotate Right



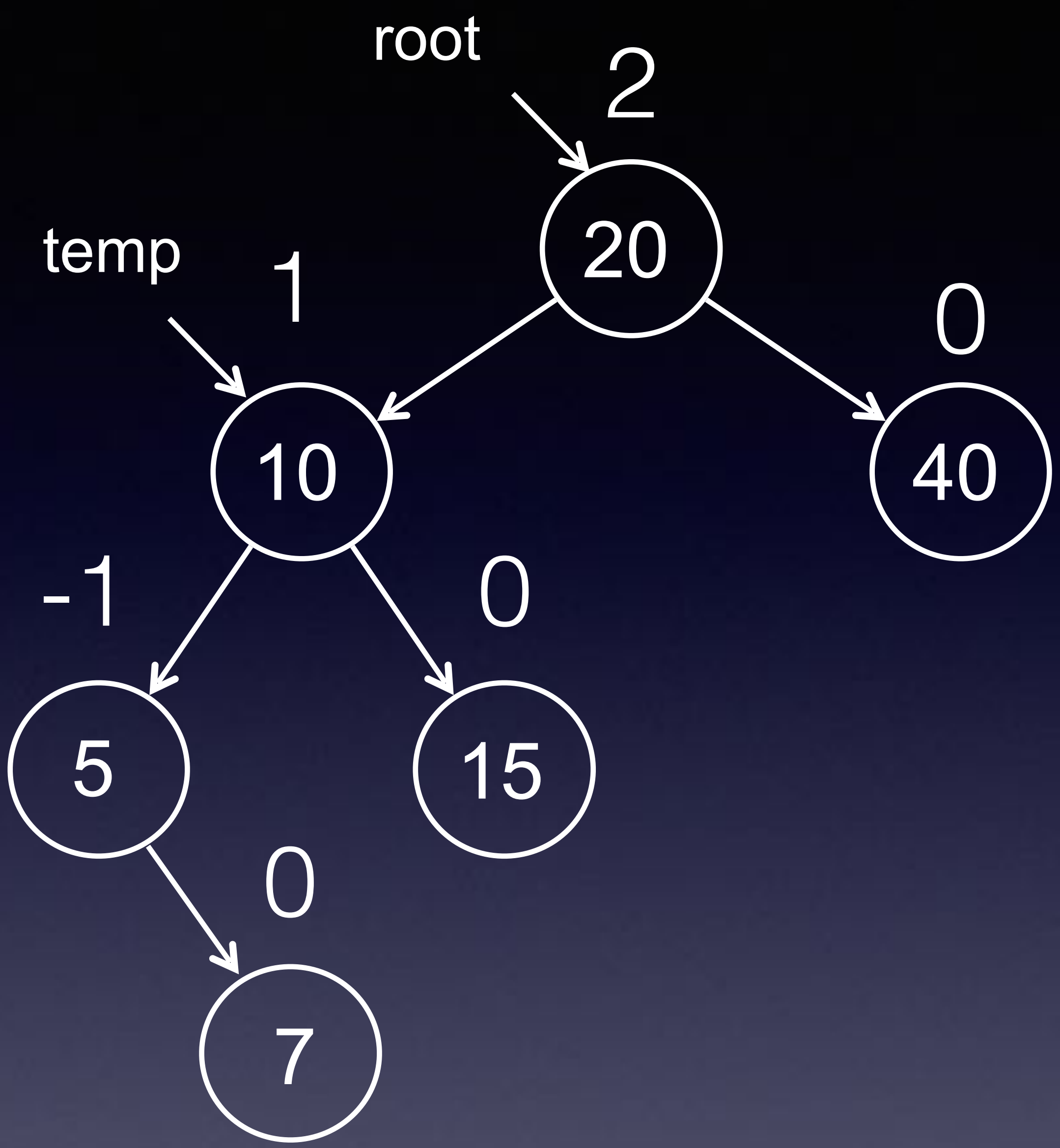
It's not magic. It's just code.

Here's the algorithm for right rotation:

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

Note: the notation "root->left" is short for "the left pointer from the root node".

Rotate Right



It's not magic. It's just code.

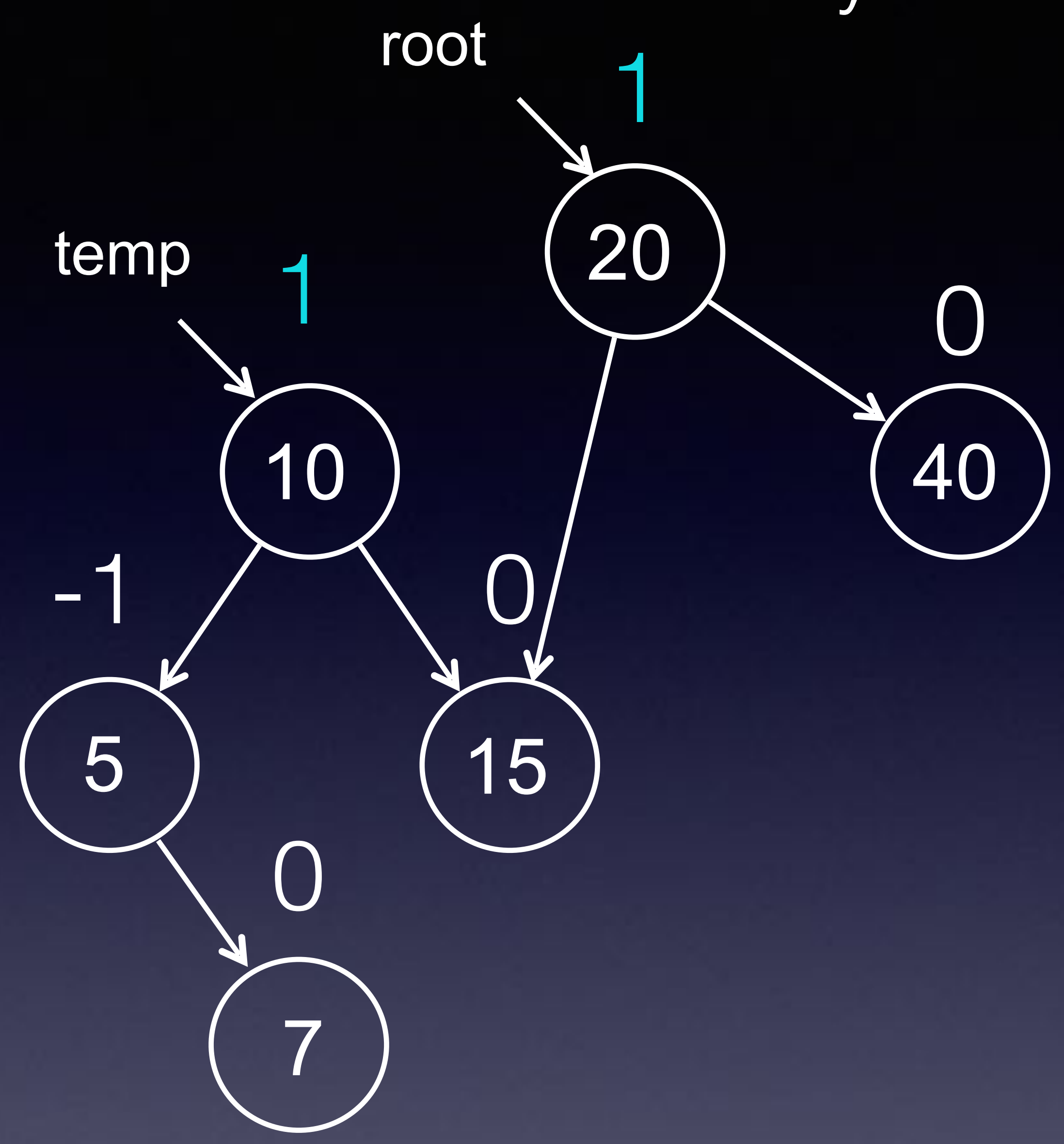
Here's the algorithm for right rotation:

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

Note: the notation "root->left" is short for "the left pointer from the root node".

Rotate Right

Notice balance changed at only two nodes: root and temp



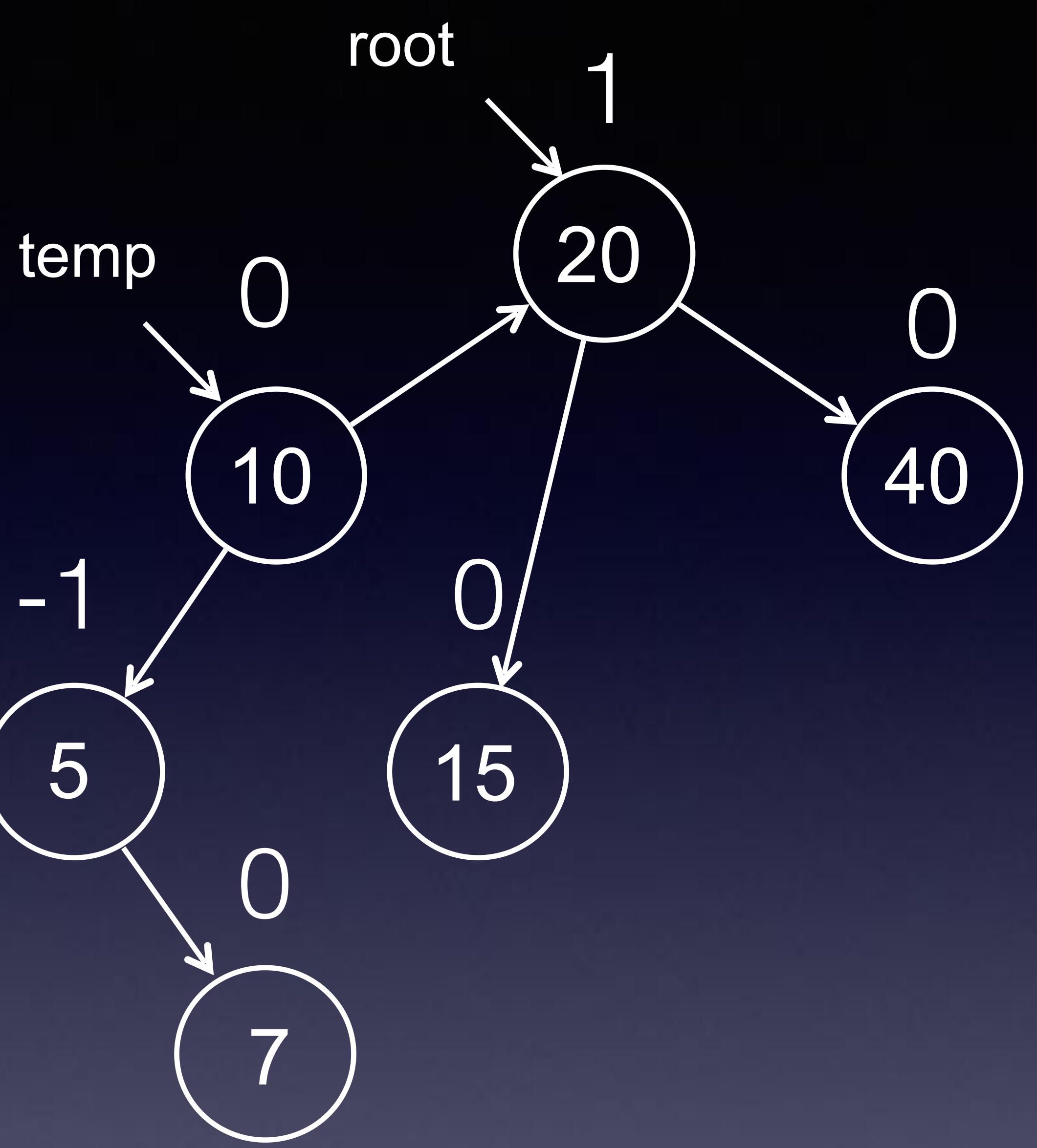
It's not magic. It's just code.

Here's the algorithm for right rotation:

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

Note: the notation "root->left" is short for "the left pointer from the root node".

Rotate Right



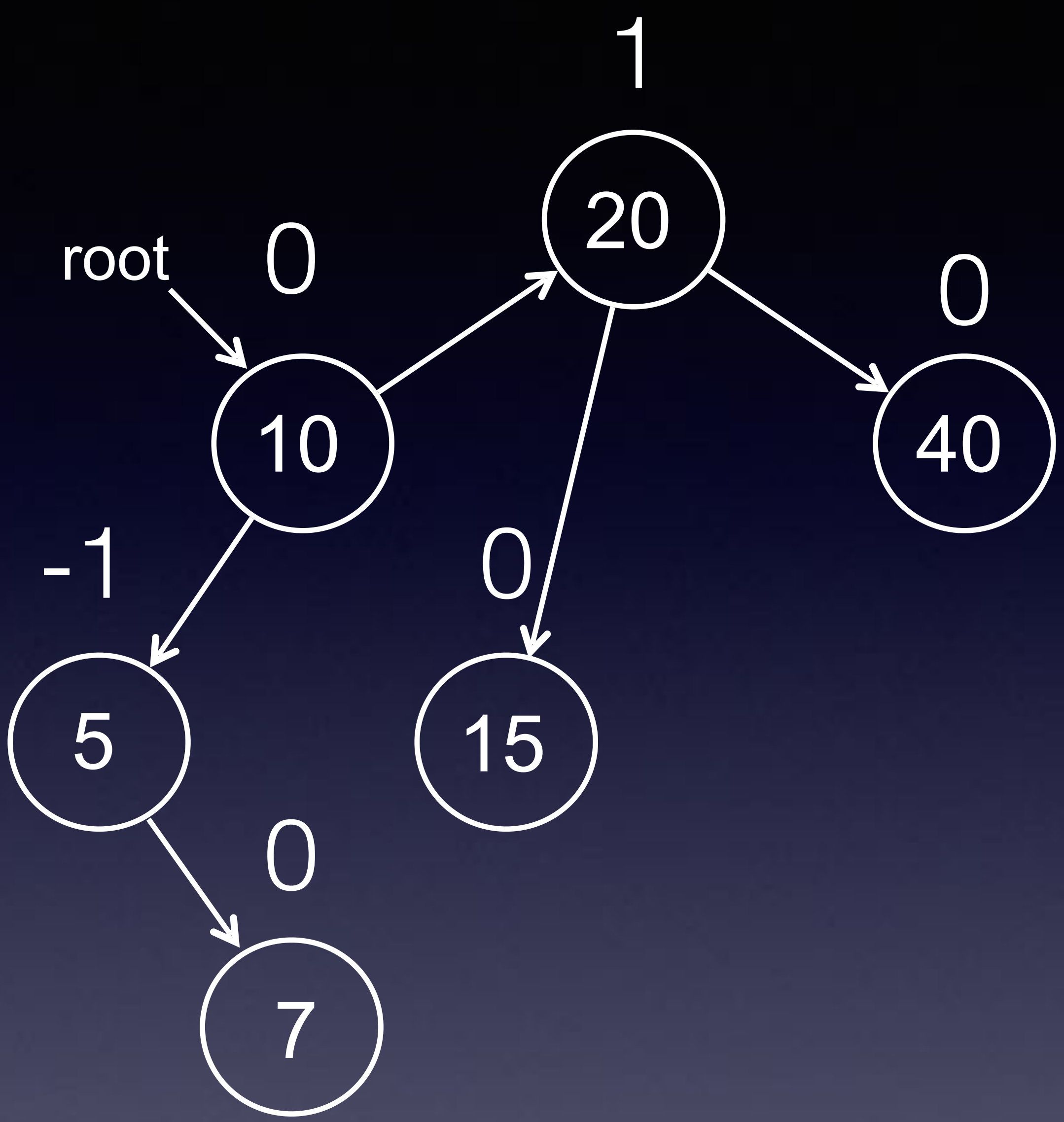
It's not magic. It's just code.

Here's the algorithm for right rotation:

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

Note: the notation "root->left" is short for "the left pointer from the root node".

Rotate Right



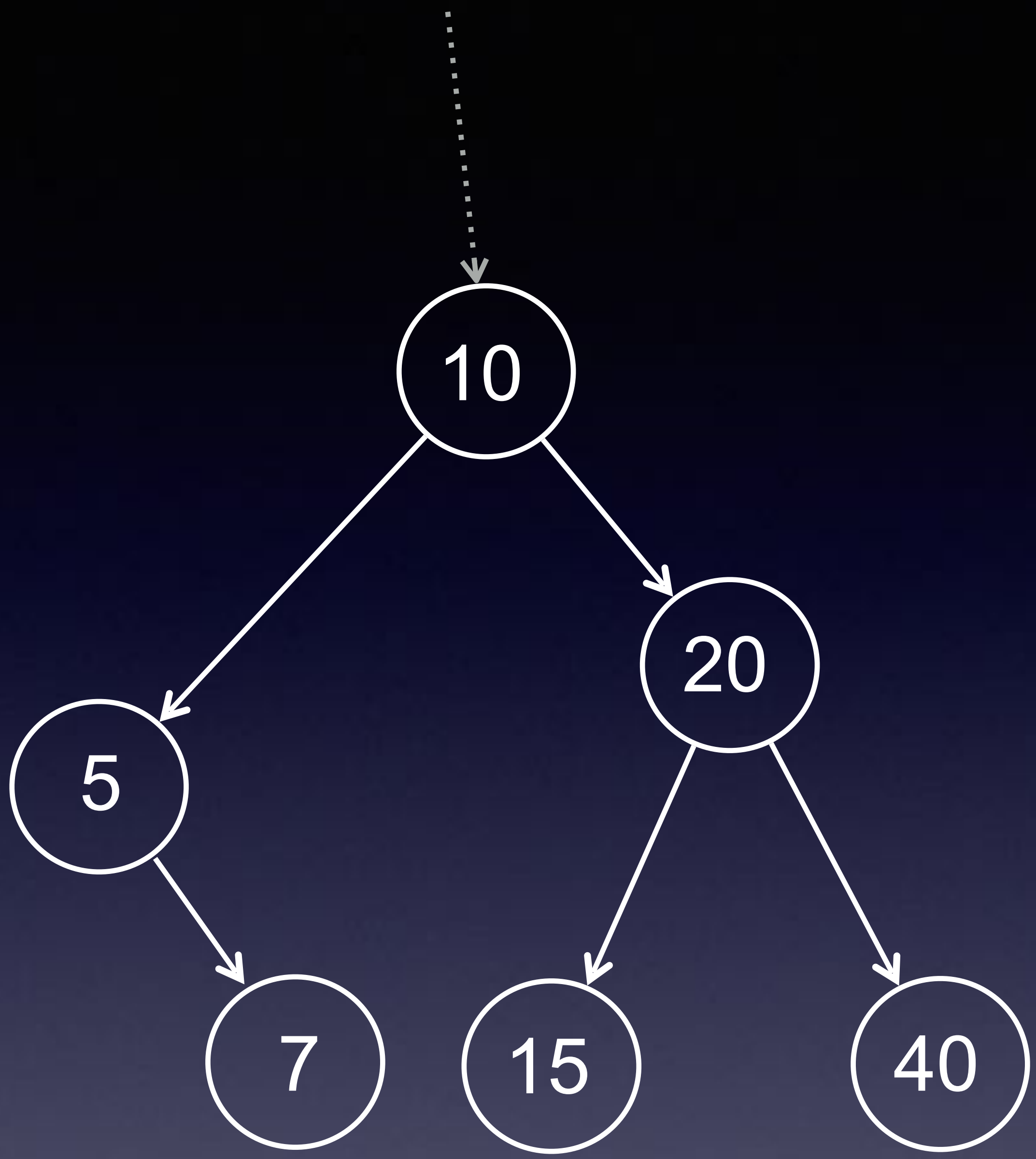
It's not magic. It's just code.

Here's the algorithm for right rotation:

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

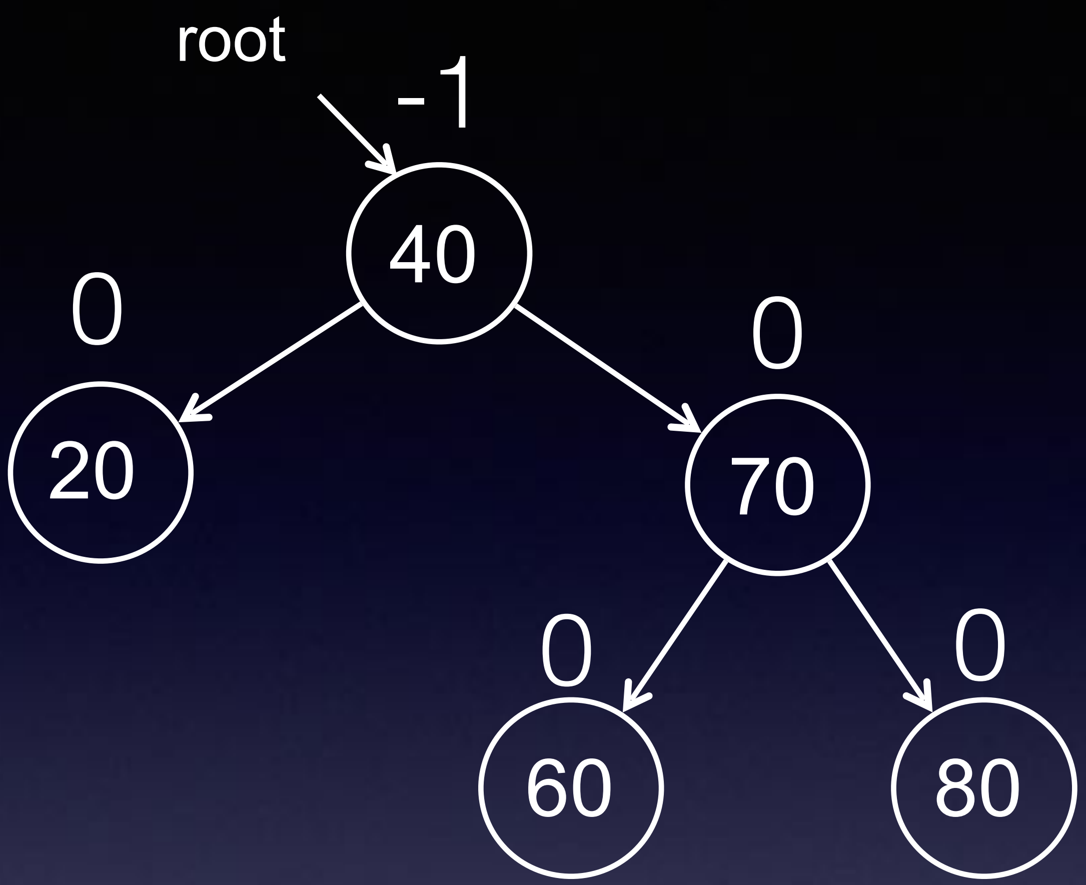
Note: the notation "root->left" is short for "the left pointer from the root node".

Rotate Right



Note that this tree could be a subtree of some larger tree, and this rotation is happening "under" some other part of the tree.

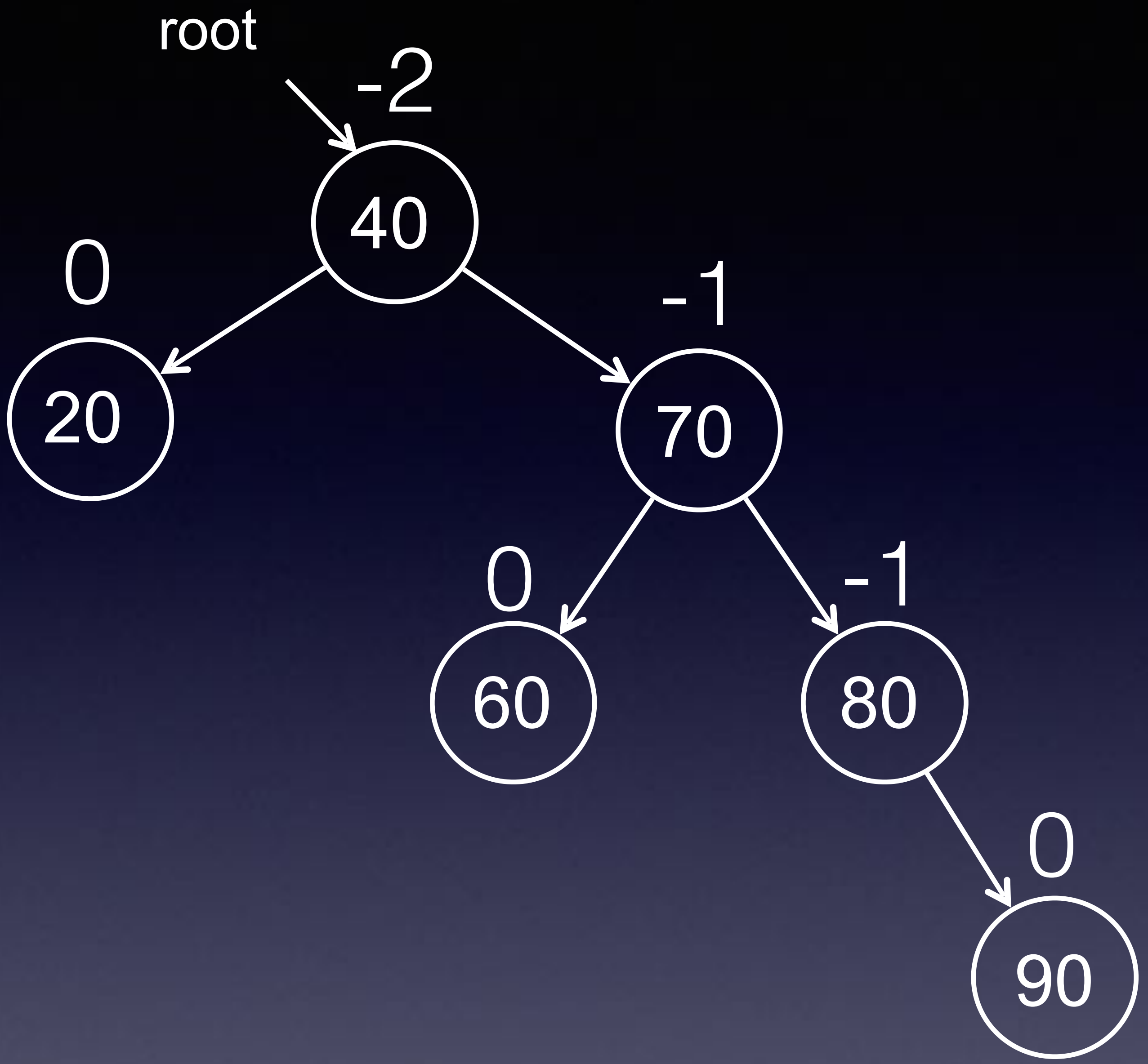
Rotate Left



Rotate left is just the mirror image of rotate right….

1. Remember the value of root->right (temp = root->right)
2. Set root->right to value of temp->left
3. Set temp->left to root
4. Set root to temp

Rotate Left

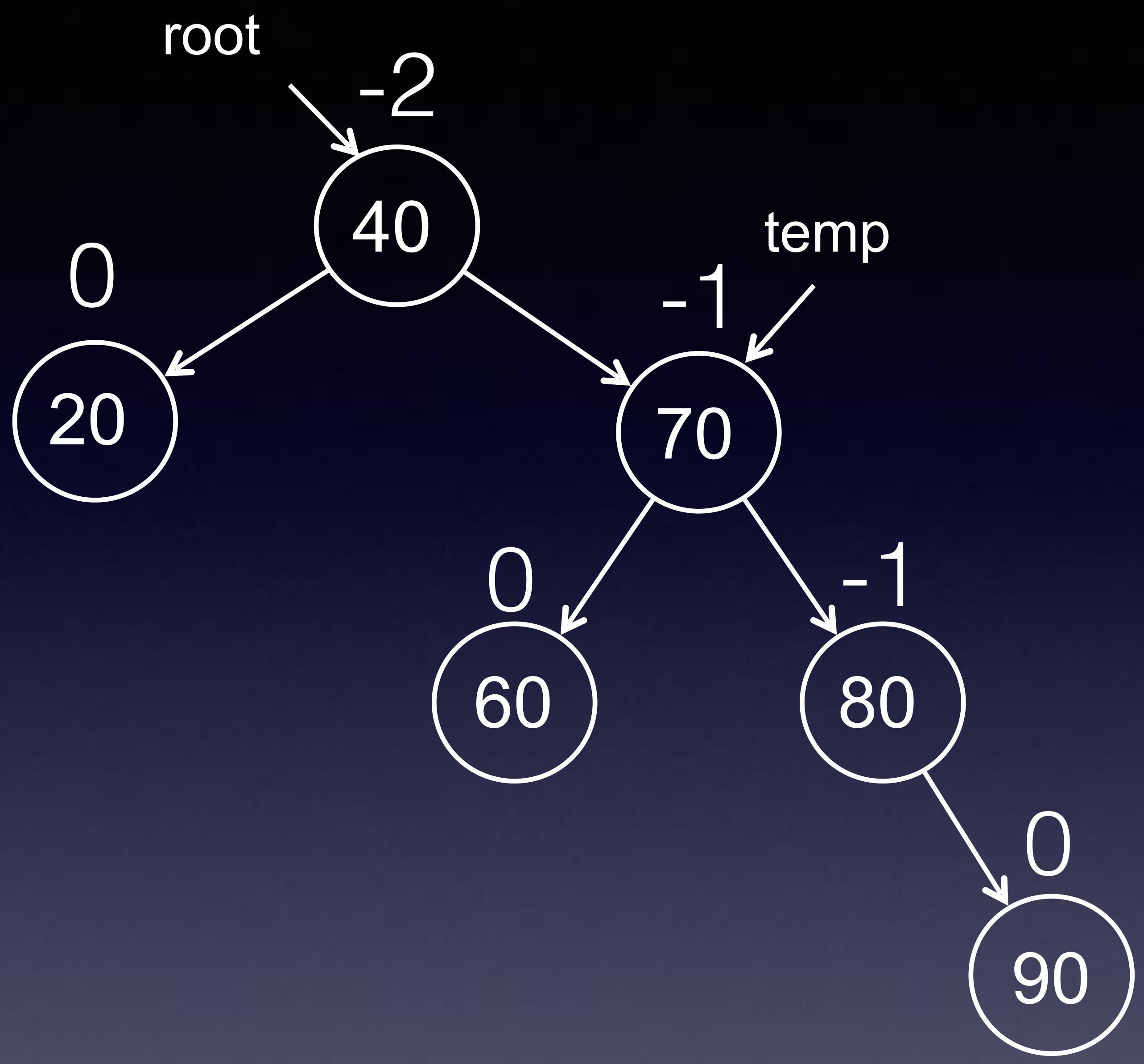


Rotate left is just the mirror image of rotate right….

1. Remember the value of root->right (temp = root->right)
2. Set root->right to value of temp->left
3. Set temp->left to root
4. Set root to temp

Adding 90 puts the balance at root outside the -1 to 1 range.
Positive balance says tree is right-heavy so rotate left.

Rotate Left

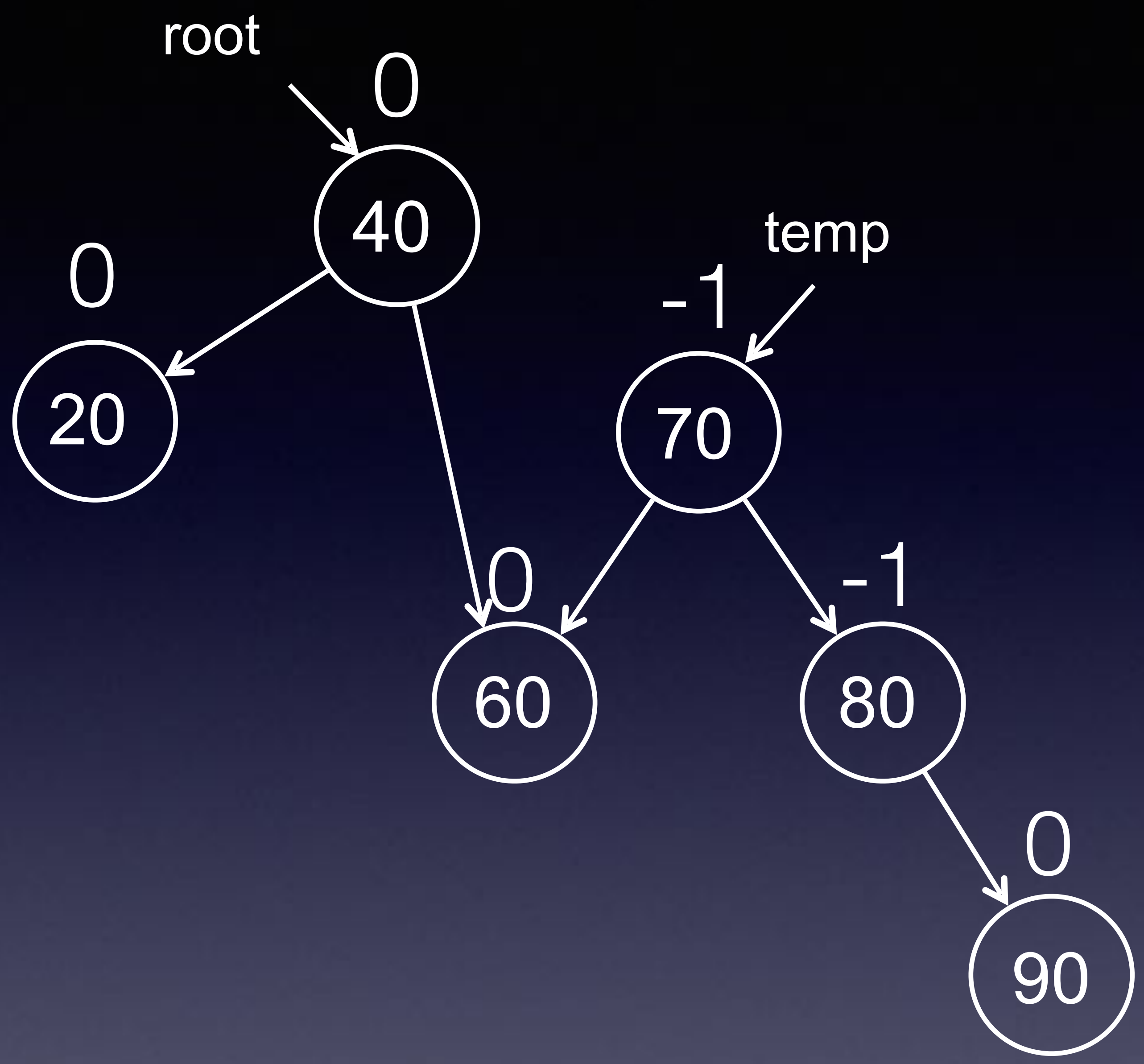


Rotate left is just the mirror image of rotate right….

1. Remember the value of root->right (temp = root->right)
2. Set root->right to value of temp->left
3. Set temp->left to root
4. Set root to temp

Adding 90 puts the balance at root outside the -1 to 1 range.
Positive balance says tree is right-heavy so rotate left.

Rotate Left

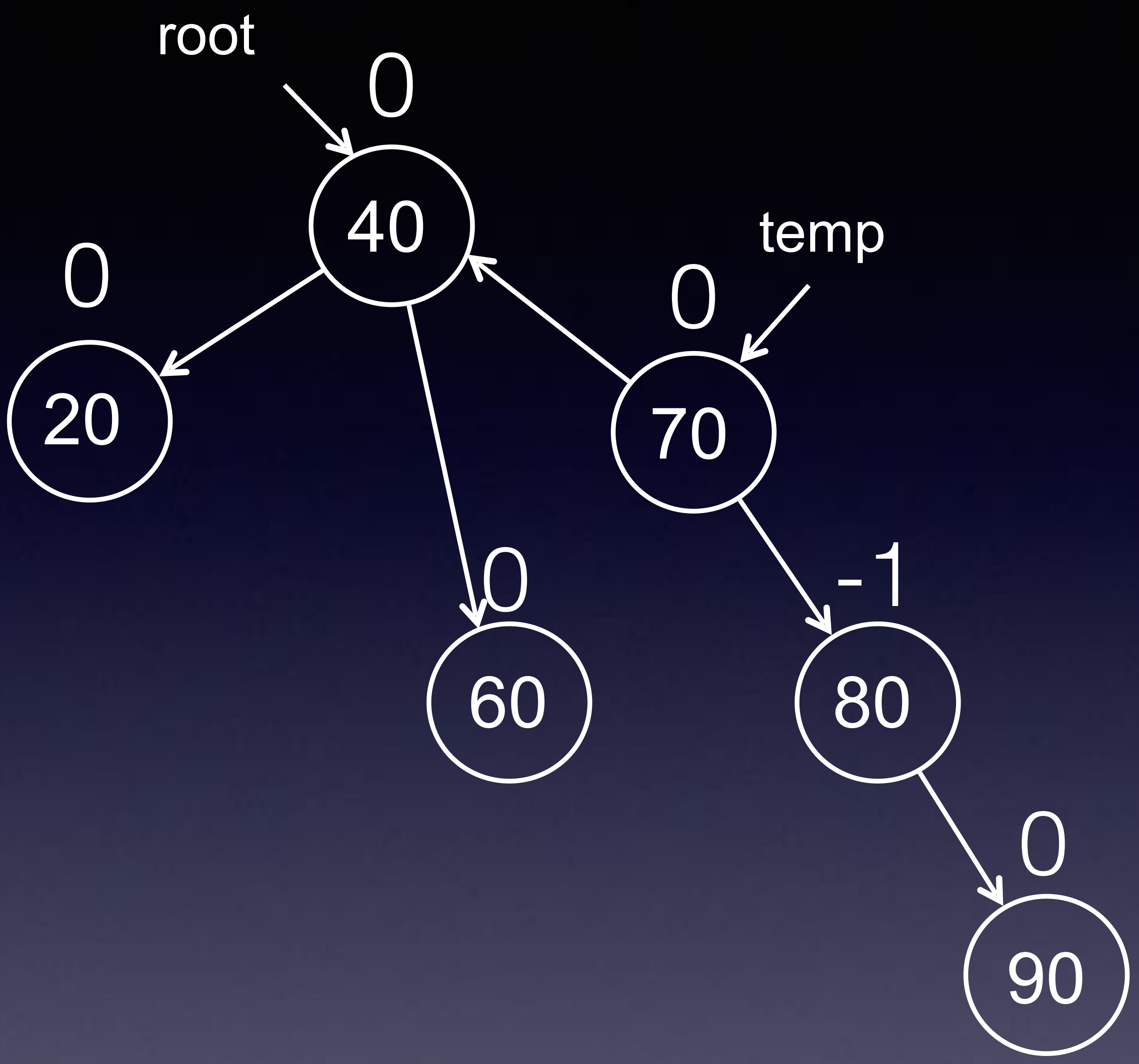


Rotate left is just the mirror image of rotate right….

1. Remember the value of root->right (temp = root->right)
2. Set root->right to value of temp->left
3. Set temp->left to root
4. Set root to temp

Adding 90 puts the balance at root outside the -1 to 1 range. Positive balance says tree is right-heavy so rotate left.

Rotate Left

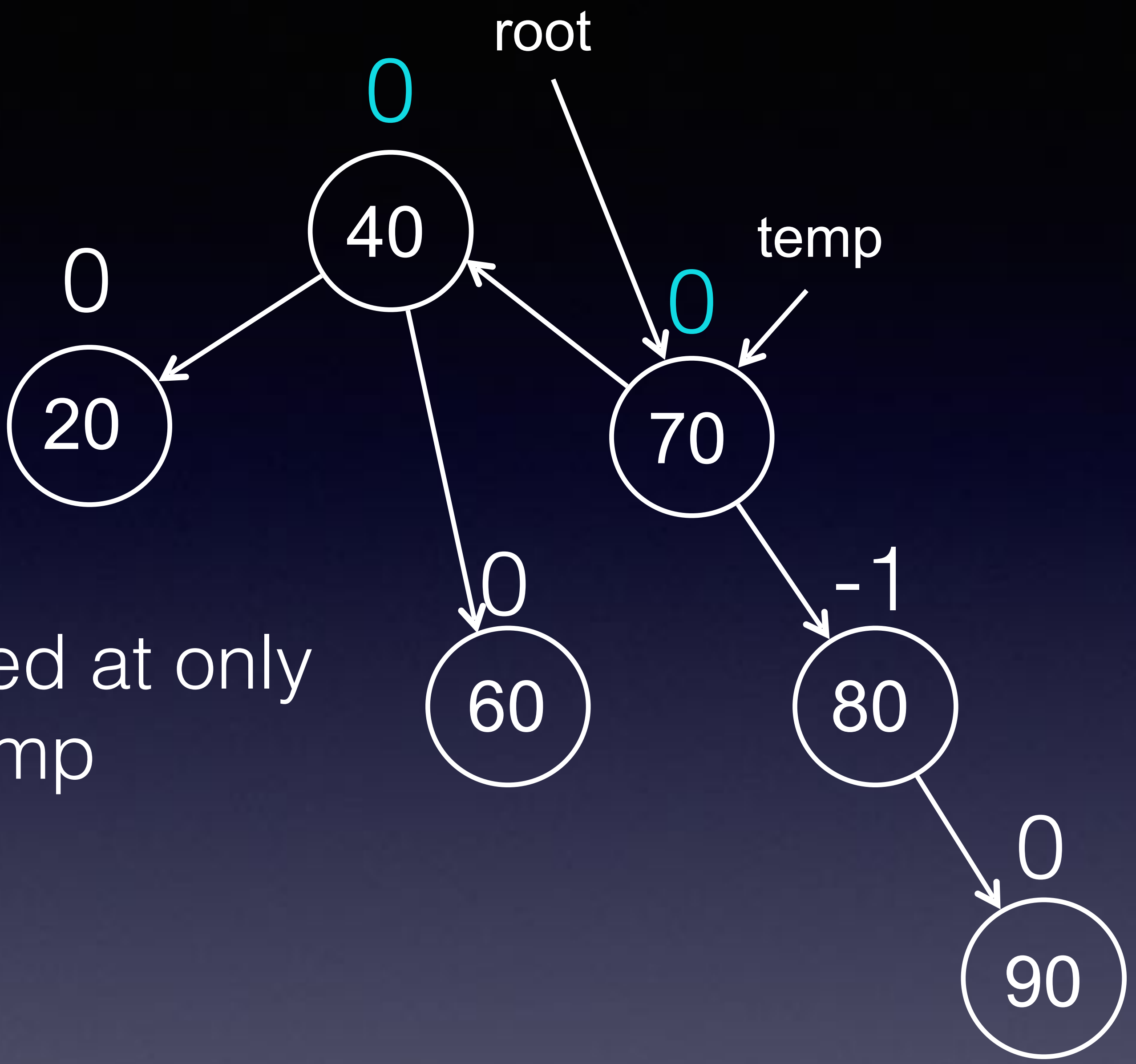


Rotate left is just the mirror image of rotate right….

1. Remember the value of root->right (temp = root->right)
2. Set root->right to value of temp->left
3. Set temp->left to root
4. Set root to temp

Adding 90 puts the balance at root outside the -1 to 1 range.
Positive balance says tree is right-heavy so rotate left.

Rotate Left



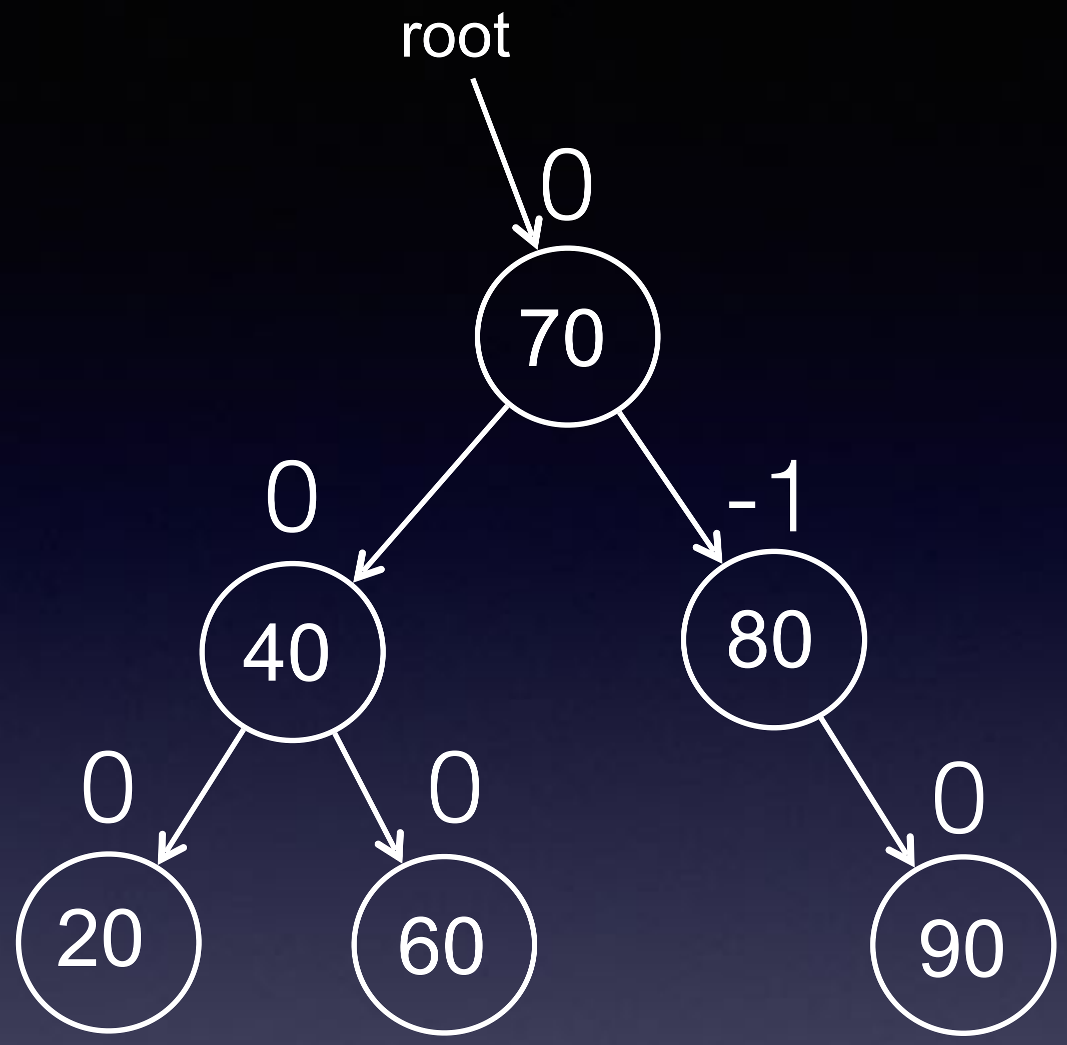
Notice balance changed at only two nodes: root and temp

Rotate left is just the mirror image of rotate right….

1. Remember the value of root->right (temp = root->right)
2. Set root->right to value of temp->left
3. Set temp->left to root
4. Set root to temp

Adding 90 puts the balance at root outside the -1 to 1 range.
Positive balance says tree is right-heavy so rotate left.

Rotate Left



Rotate left is just the mirror image of rotate right….

1. Remember the value of root->right (temp = root->right)
2. Set root->right to value of temp->left
3. Set temp->left to root
4. Set root to temp

Adding 90 puts the balance at root outside the -1 to 1 range.
Positive balance says tree is right-heavy so rotate left.

The AVL algorithm

An AVL tree is a balanced binary search tree in which **each node has a balance value** that is equal to the difference between the heights of its left and right subtrees: $h_L - h_R$.

A node in the tree is balanced if it has a **balance** value of 0.

A node is **left-heavy** if it has a **positive** balance.

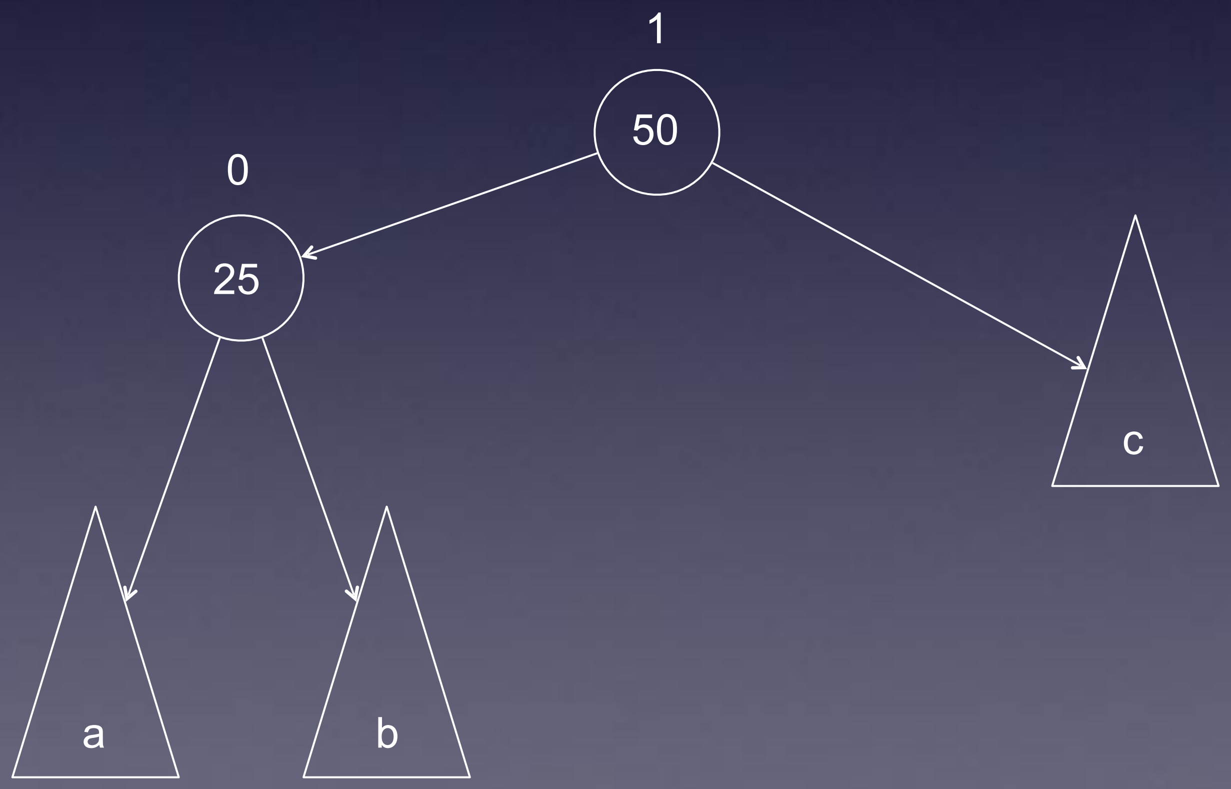
A node is **right-heavy** if it has a **negative** balance.

Rebalancing is done when traversing back along the insertion path and observing a balance of -2 or +2.

AVL trees

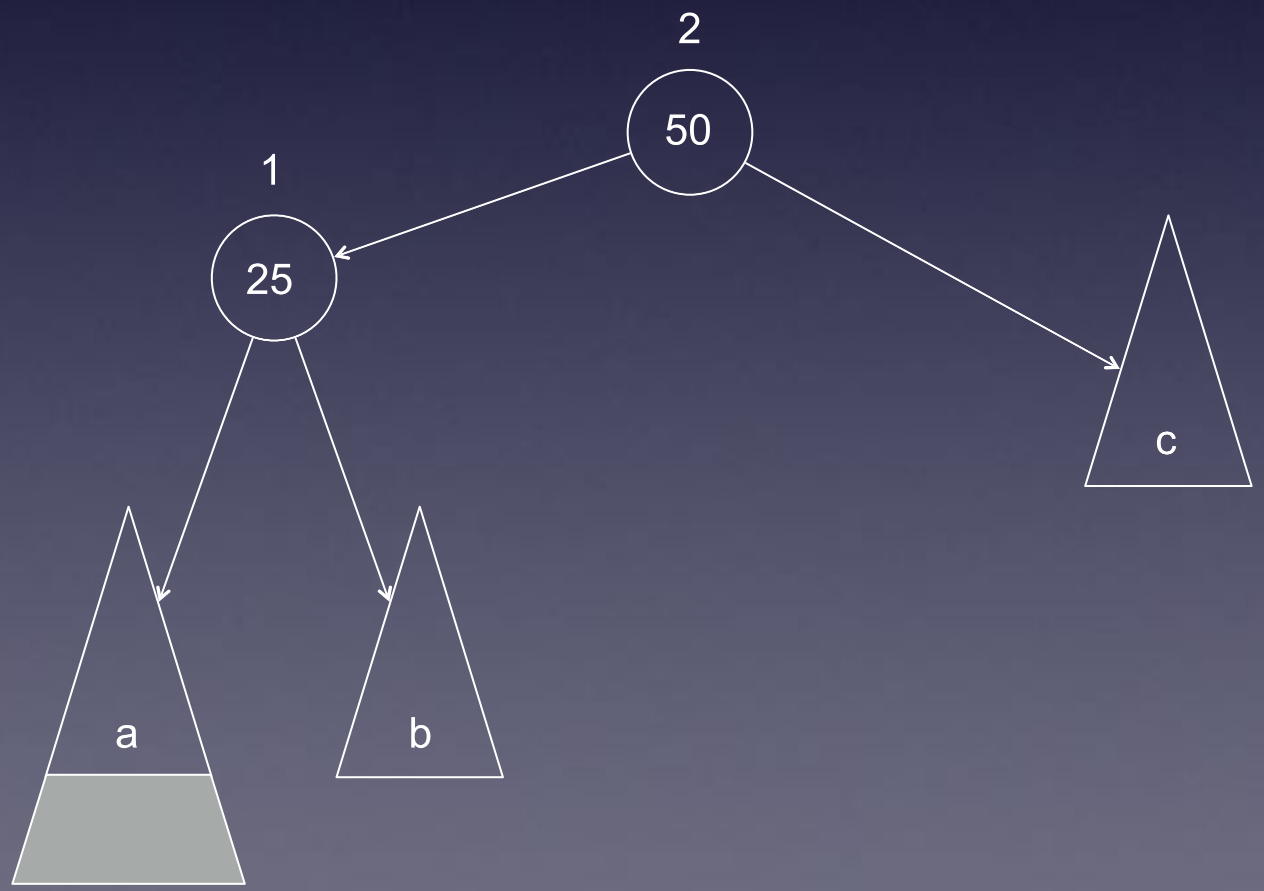
The AVL algorithm looks for **four cases** of unbalanced trees:

The **Left-Left** tree (parent and left child nodes are both left-heavy, parent balance is +2, child balance is +1). Fix by rotating right around the parent.



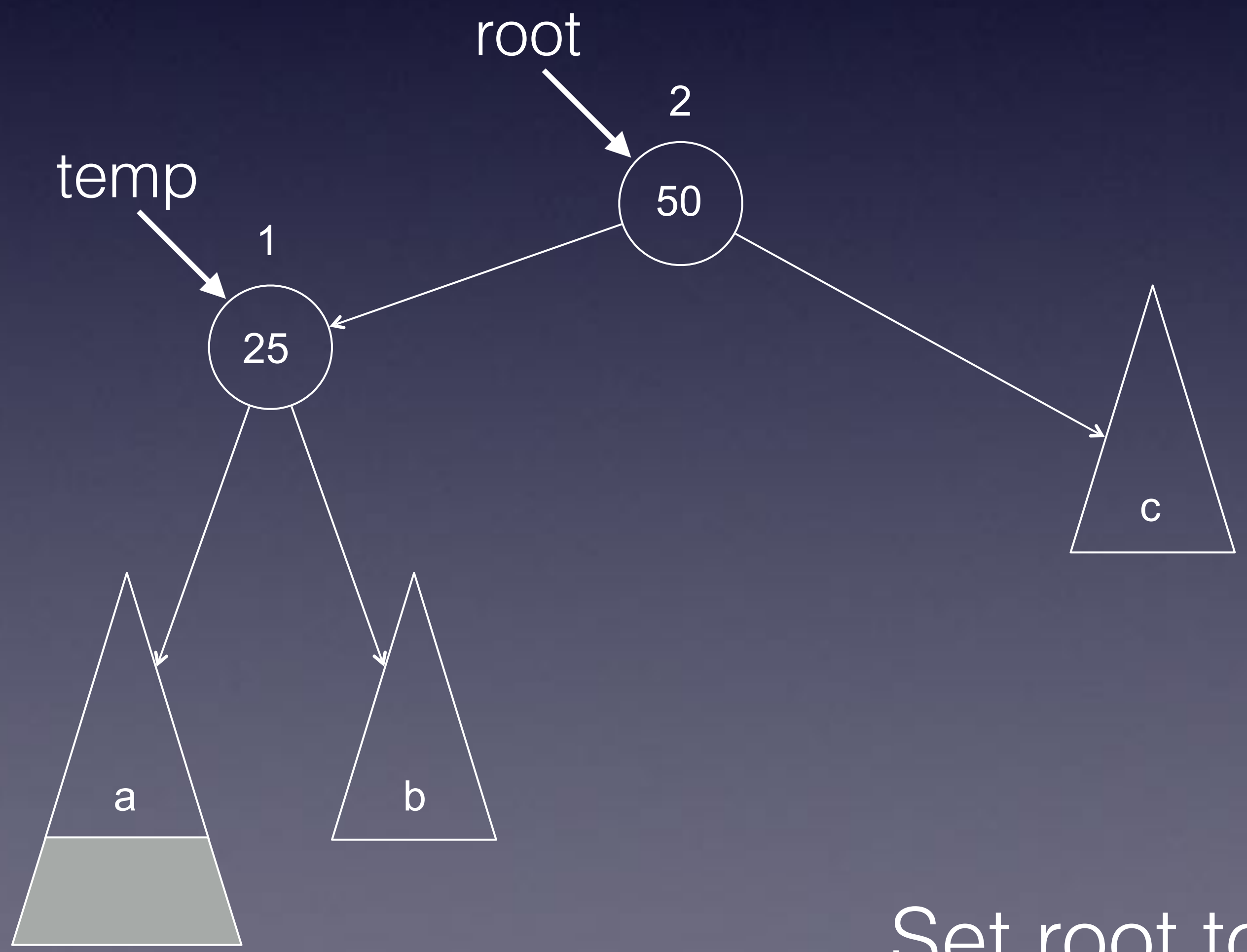
AVL trees

The **Left-Left** tree (parent and left child nodes are both left-heavy, parent balance is +2, child balance is +1). Fix by rotating right around the parent.



AVL trees

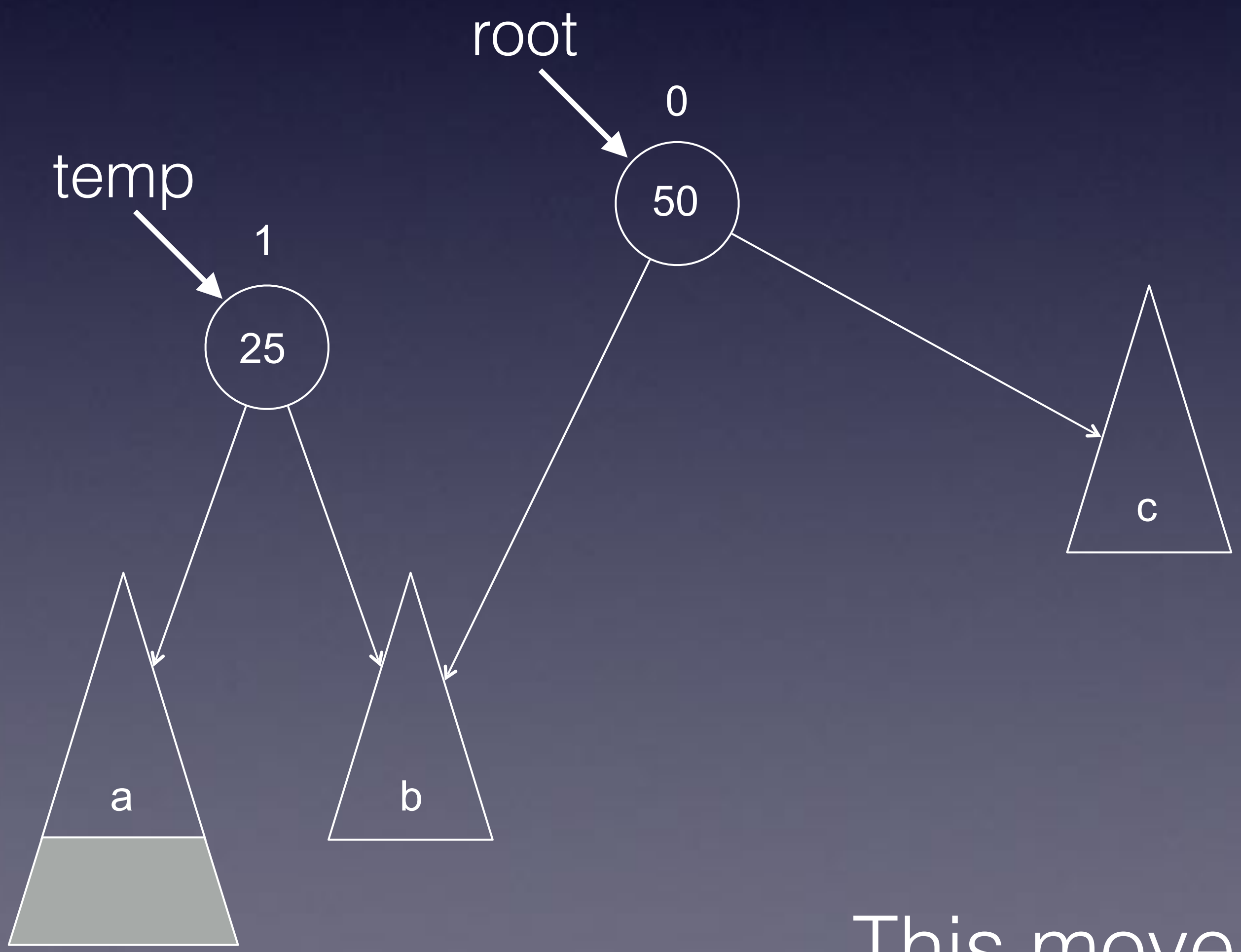
The **Left-Left** tree (parent and left child nodes are both left-heavy, parent balance is +2, child balance is +1). Fix by rotating right around the parent.



Set root to parent node and temp to root's left subtree

AVL trees

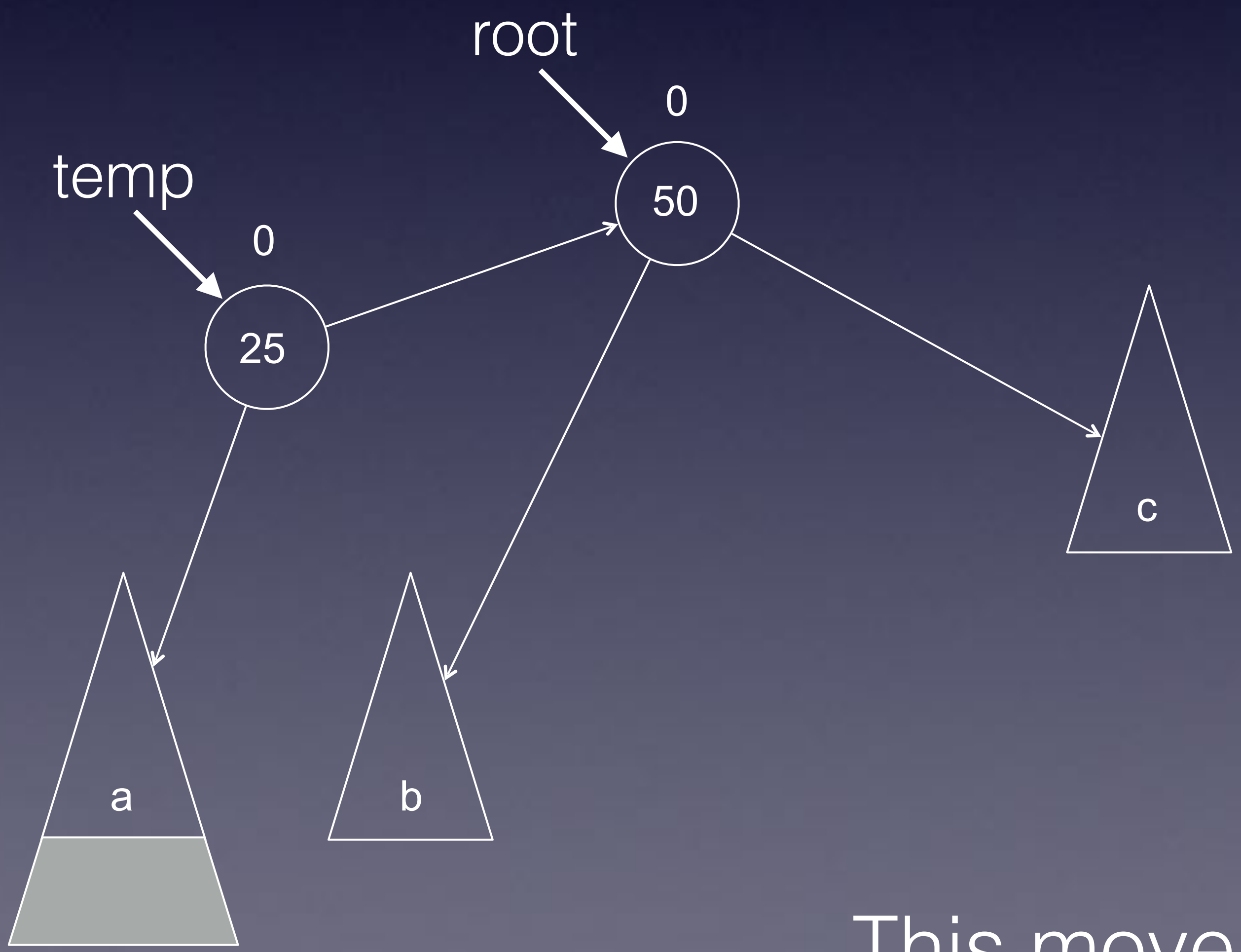
The **Left-Left** tree (parent and left child nodes are both left-heavy, parent balance is +2, child balance is +1). Fix by rotating right around the parent.



This move reduces the root's left subtree height by two

AVL trees

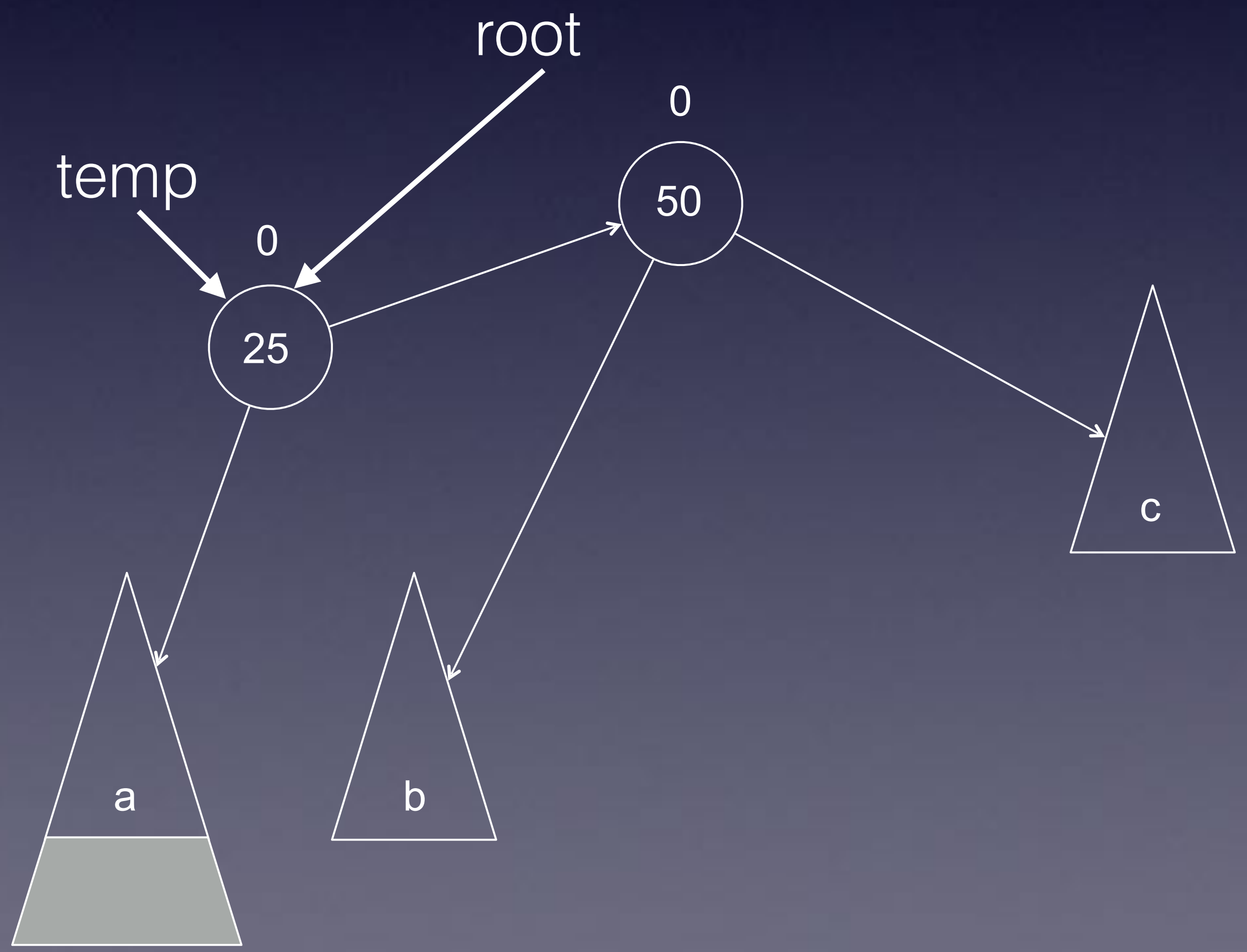
The **Left-Left** tree (parent and left child nodes are both left-heavy, parent balance is +2, child balance is +1). Fix by rotating right around the parent.



This move increases temp's right subtree height by one

AVL trees

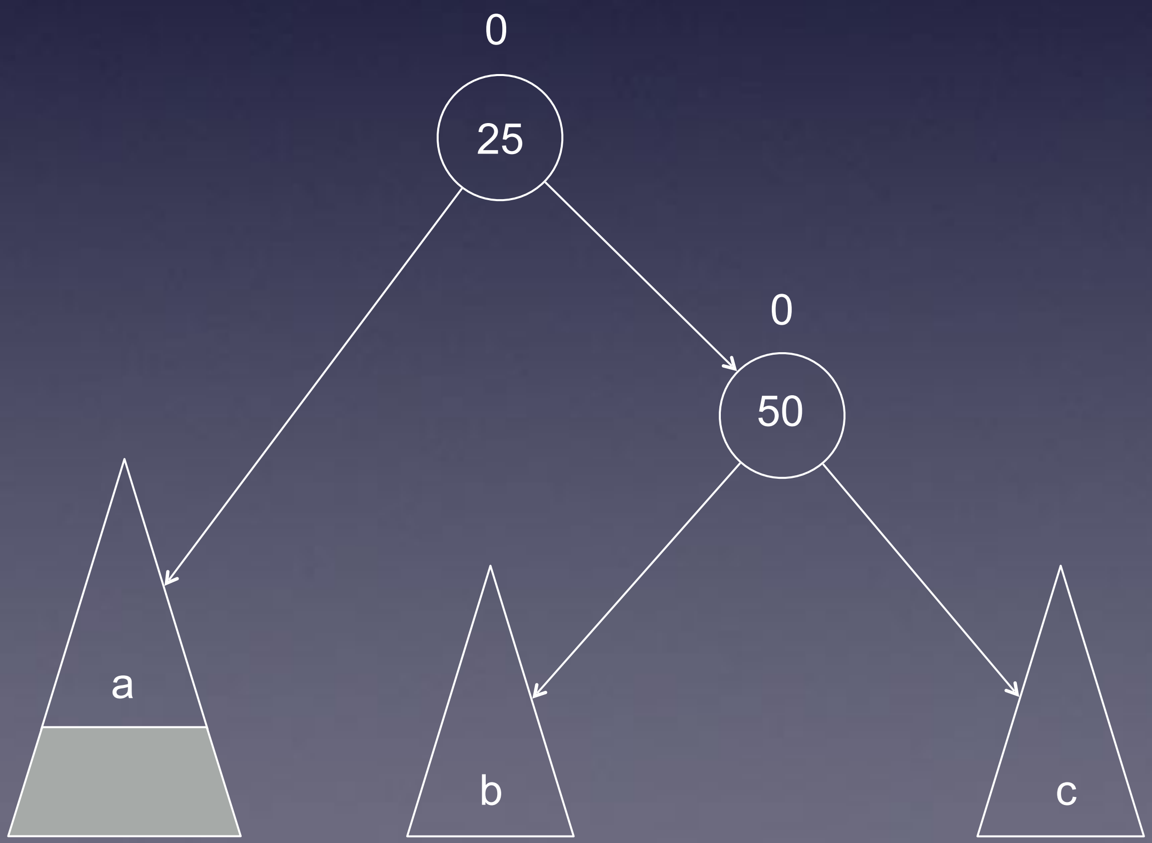
The **Left-Left** tree (parent and left child nodes are both left-heavy, parent balance is +2, child balance is +1). Fix by rotating right around the parent.



Set root to temp

AVL trees

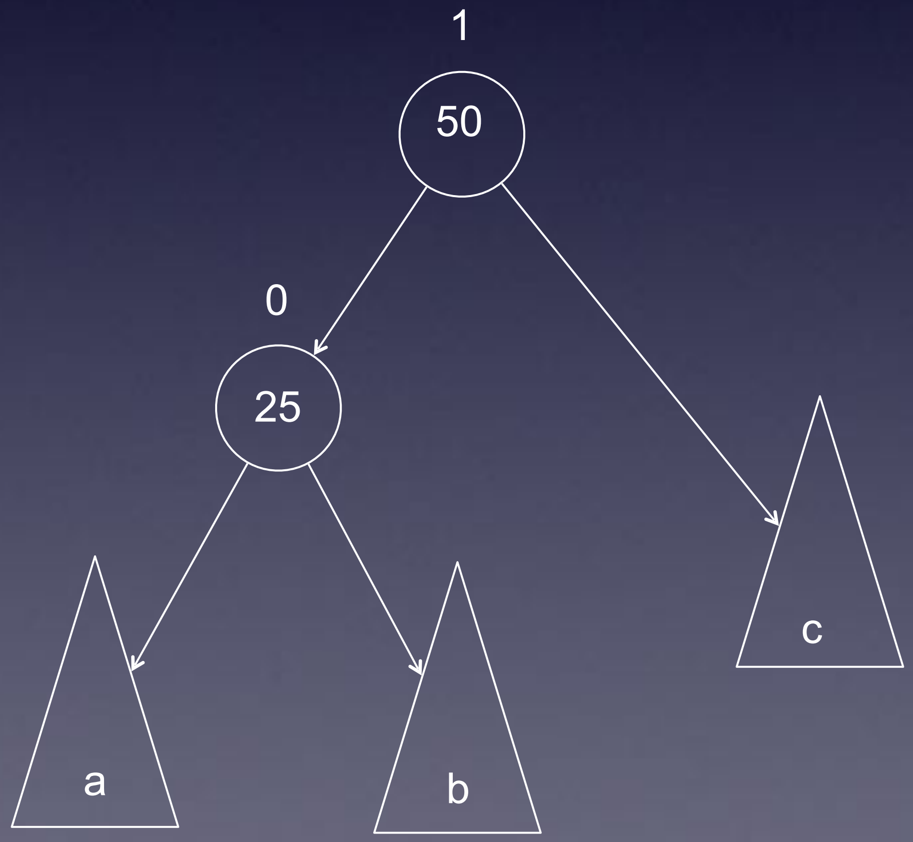
The **Left-Left** tree (parent and left child nodes are both left-heavy, parent balance is +2, child balance is +1). Fix by rotating right around the parent.



Done!

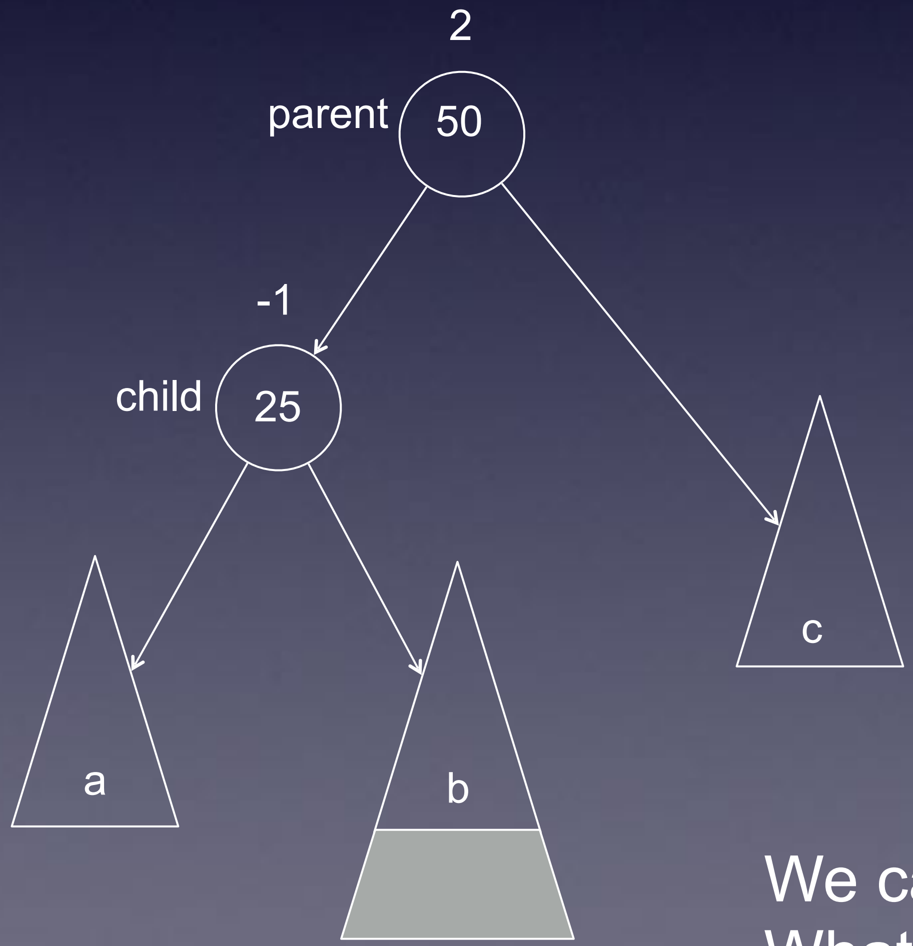
AVL trees

The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



AVL trees

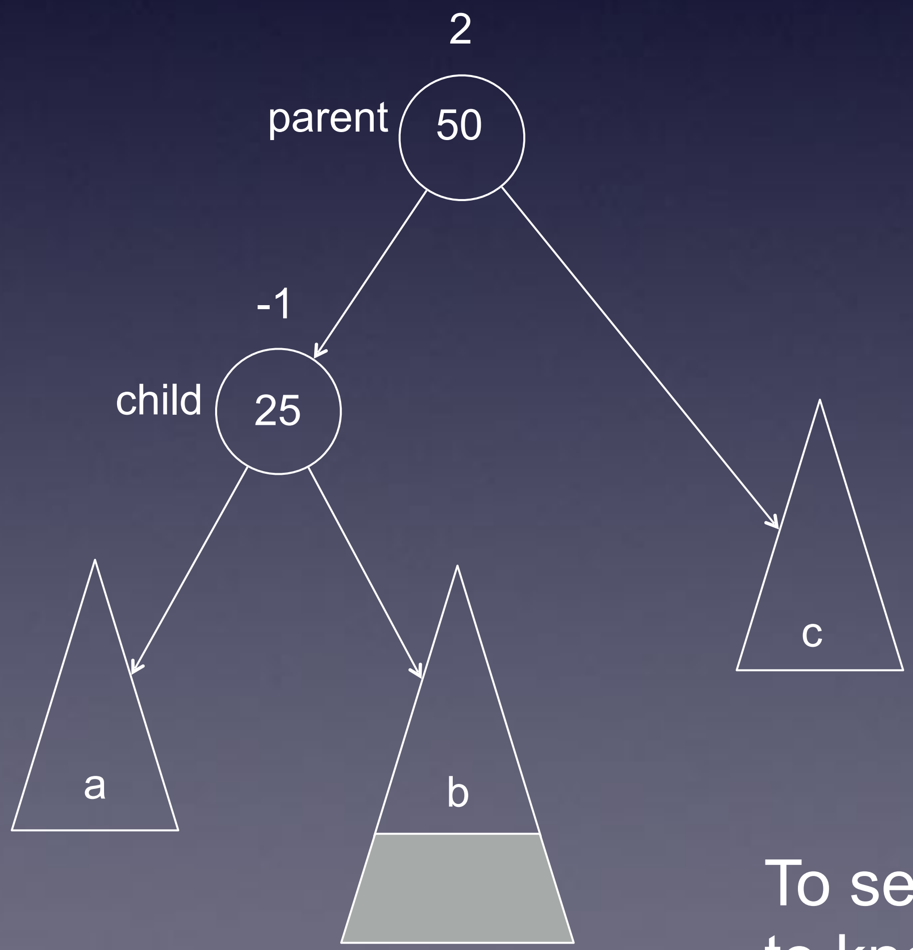
The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



We can’t simply fix this by rotating right.
What happens to the balance at the child?

AVL trees

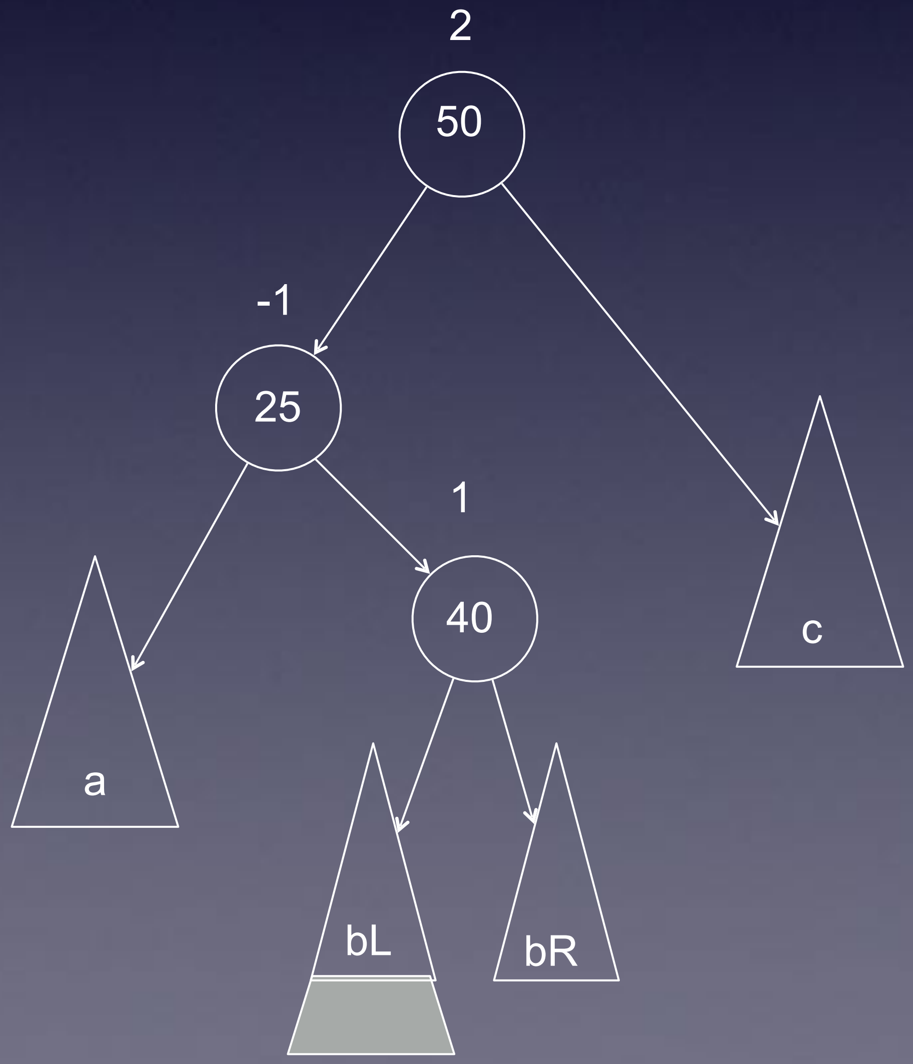
The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



To see the left rotation details, we need to know where the new node was inserted in subtree b. Here is one possibility:

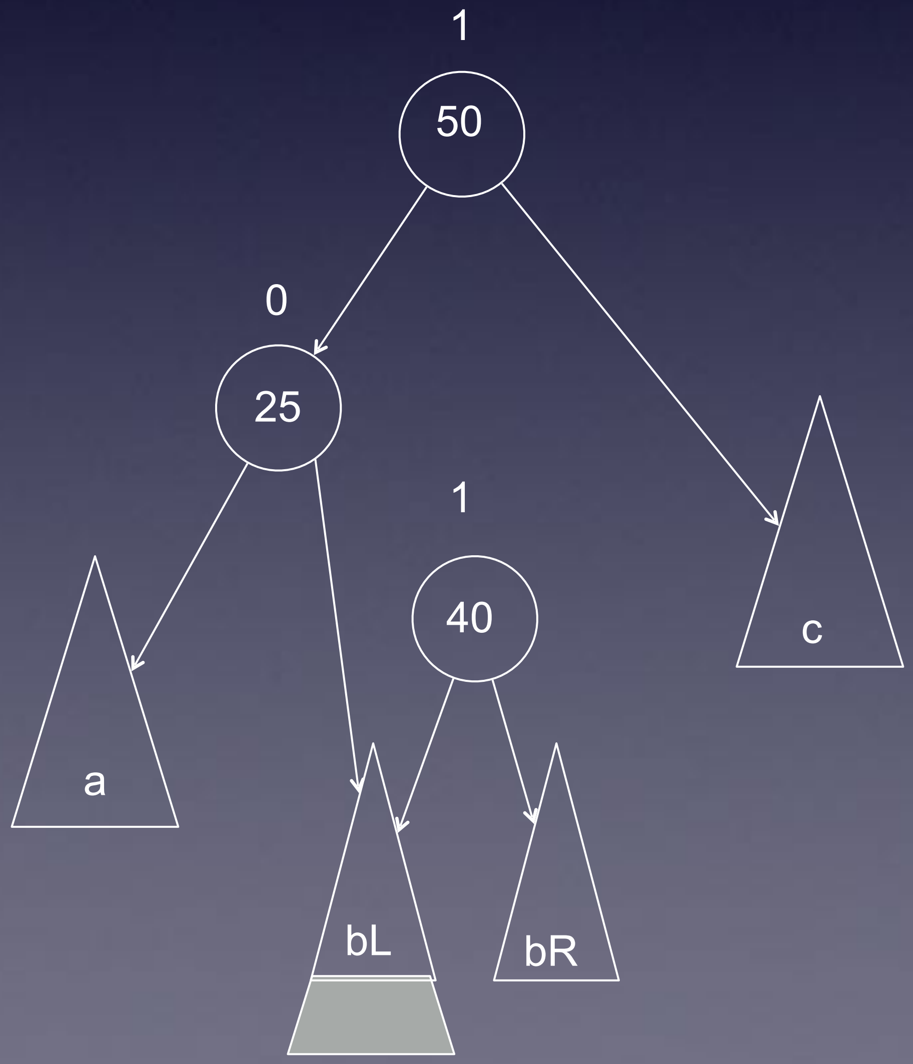
AVL trees

The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



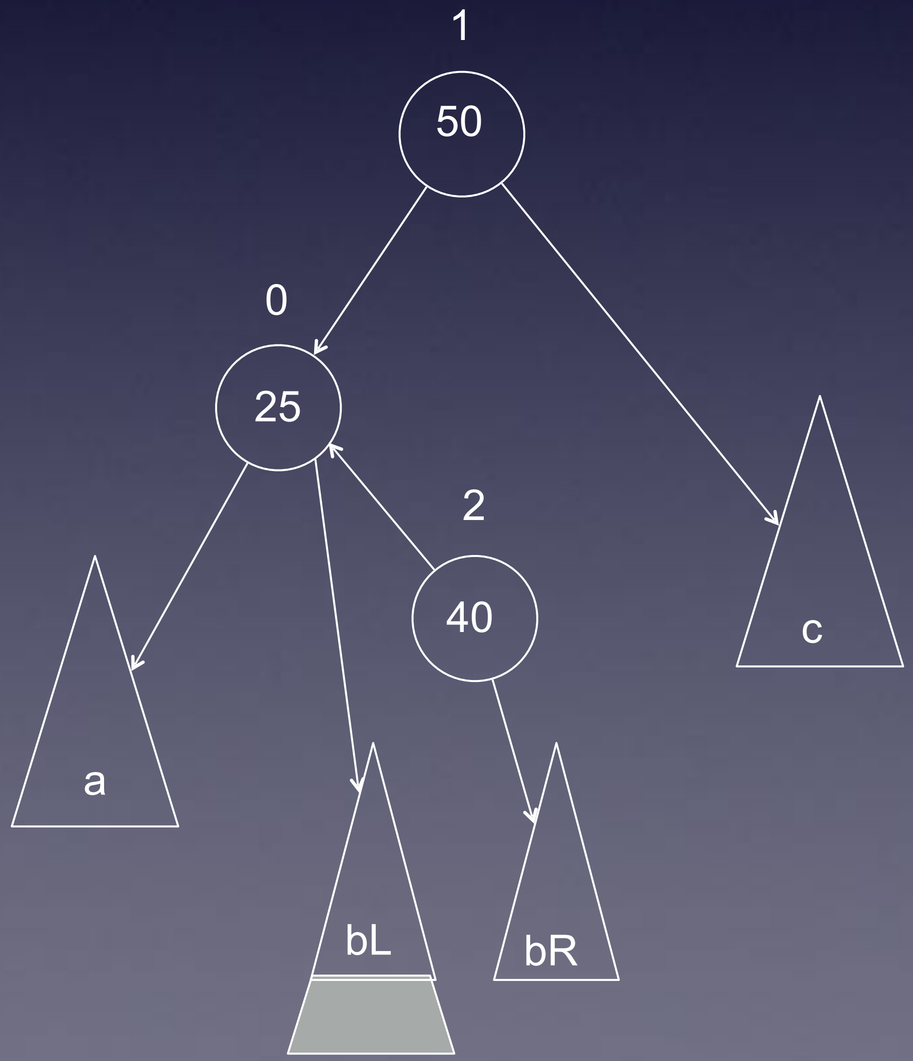
AVL trees

The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



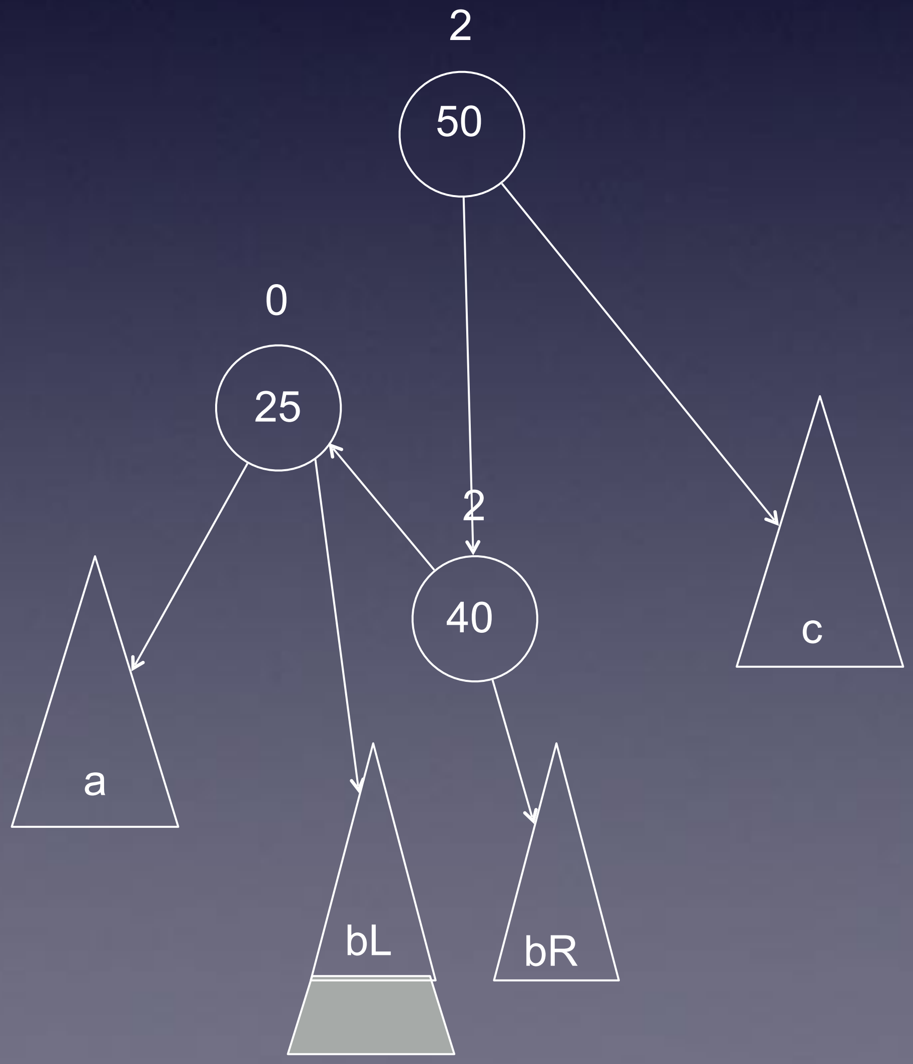
AVL trees

The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



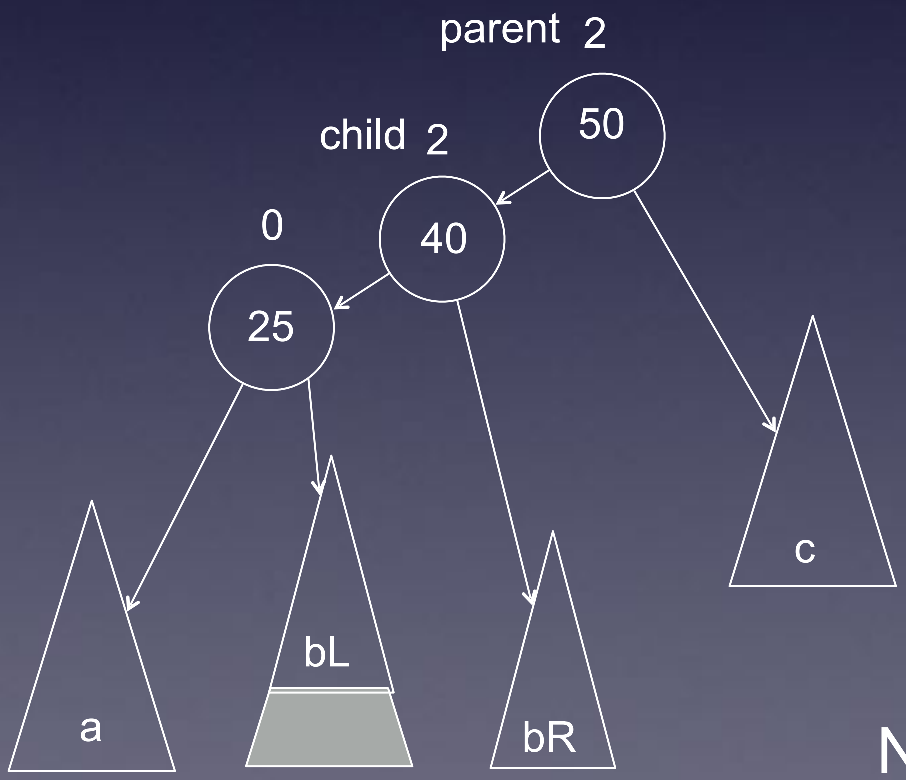
AVL trees

The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



AVL trees

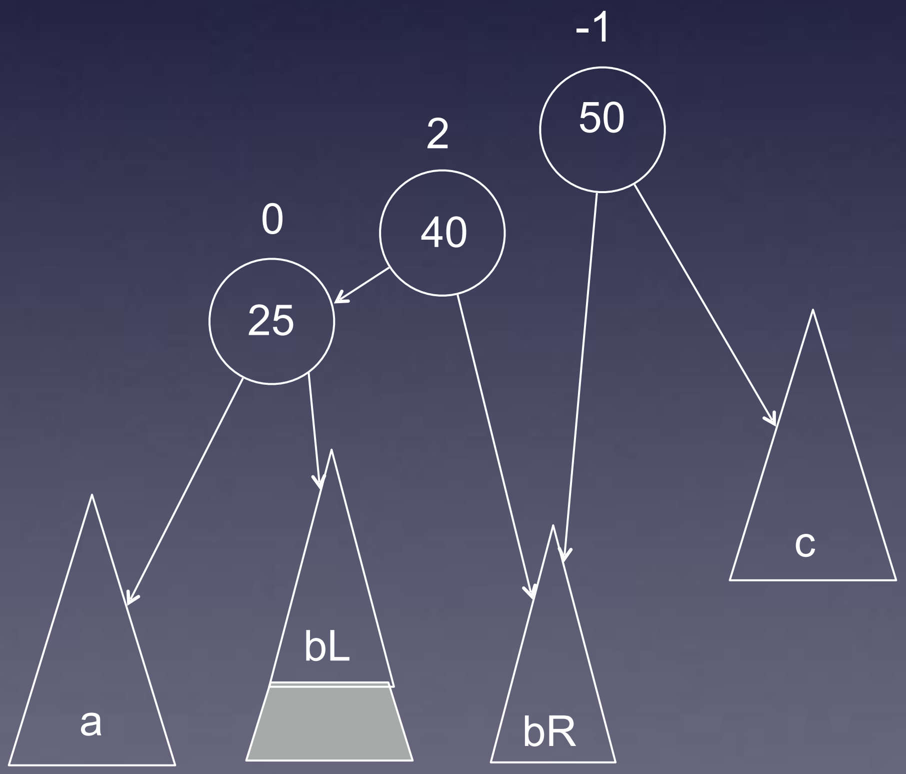
The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



Now we can do that right rotation around parent.

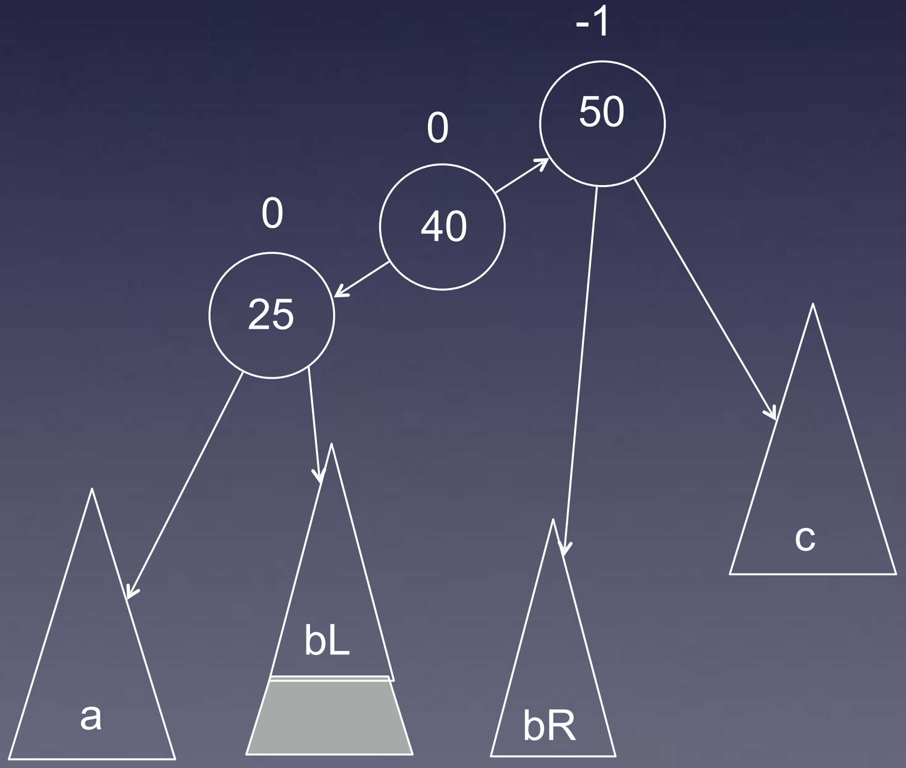
AVL trees

The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



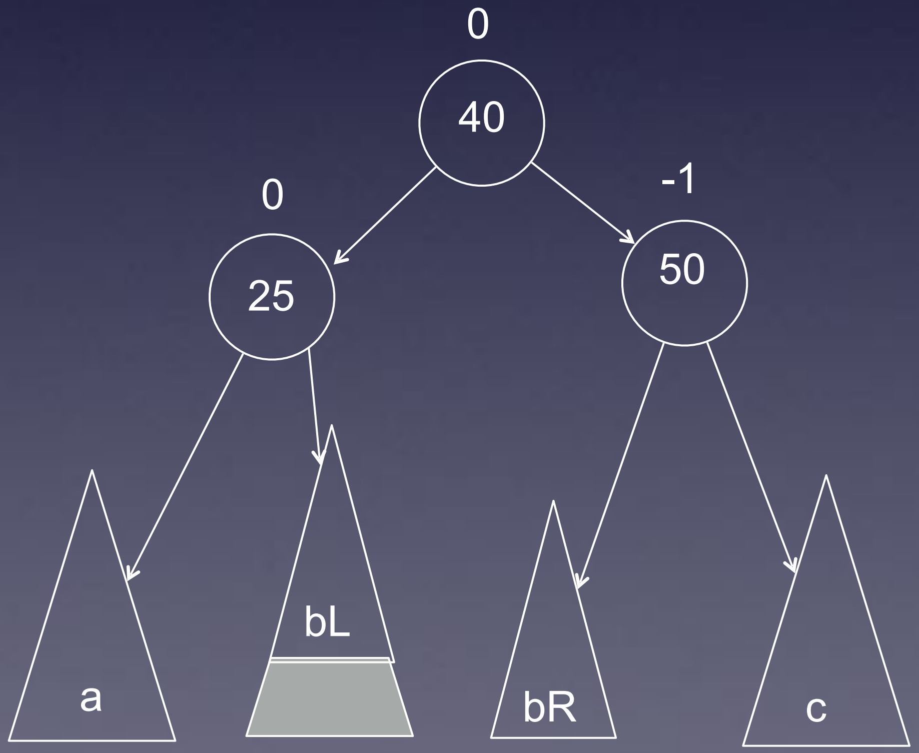
AVL trees

The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



AVL trees

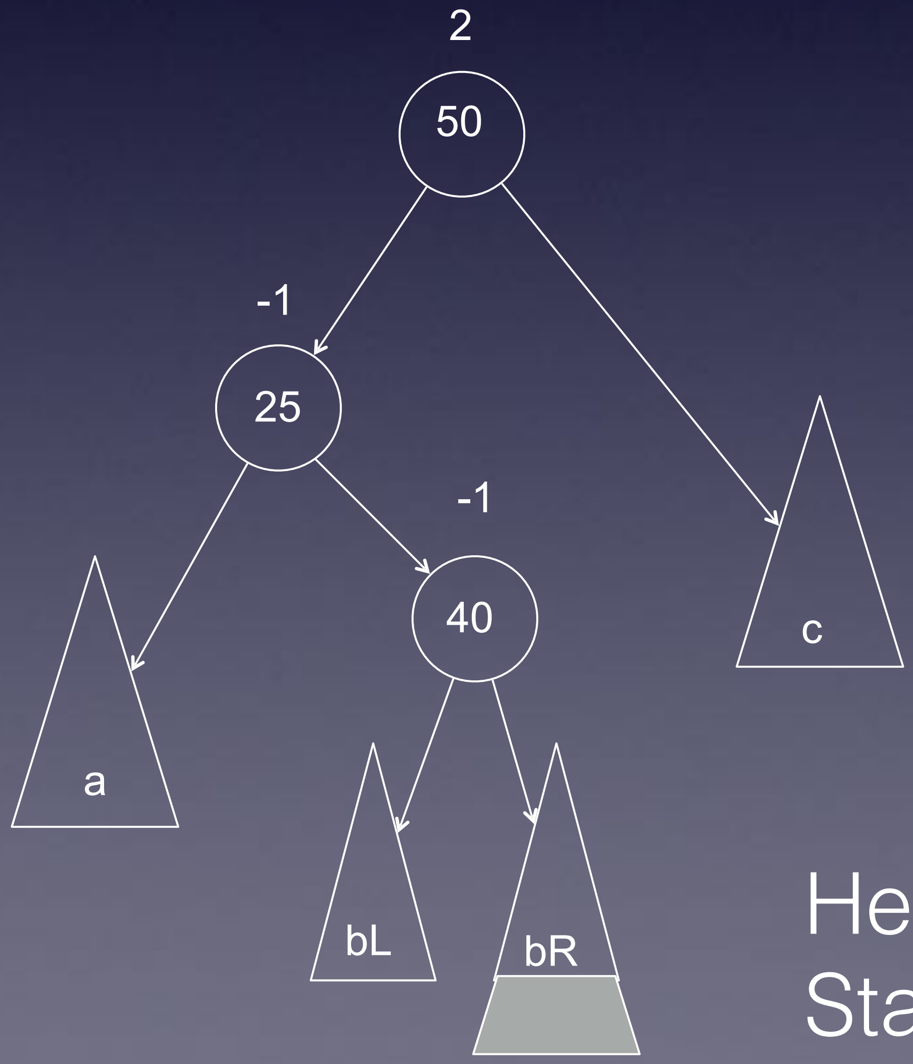
The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



Done!

AVL trees

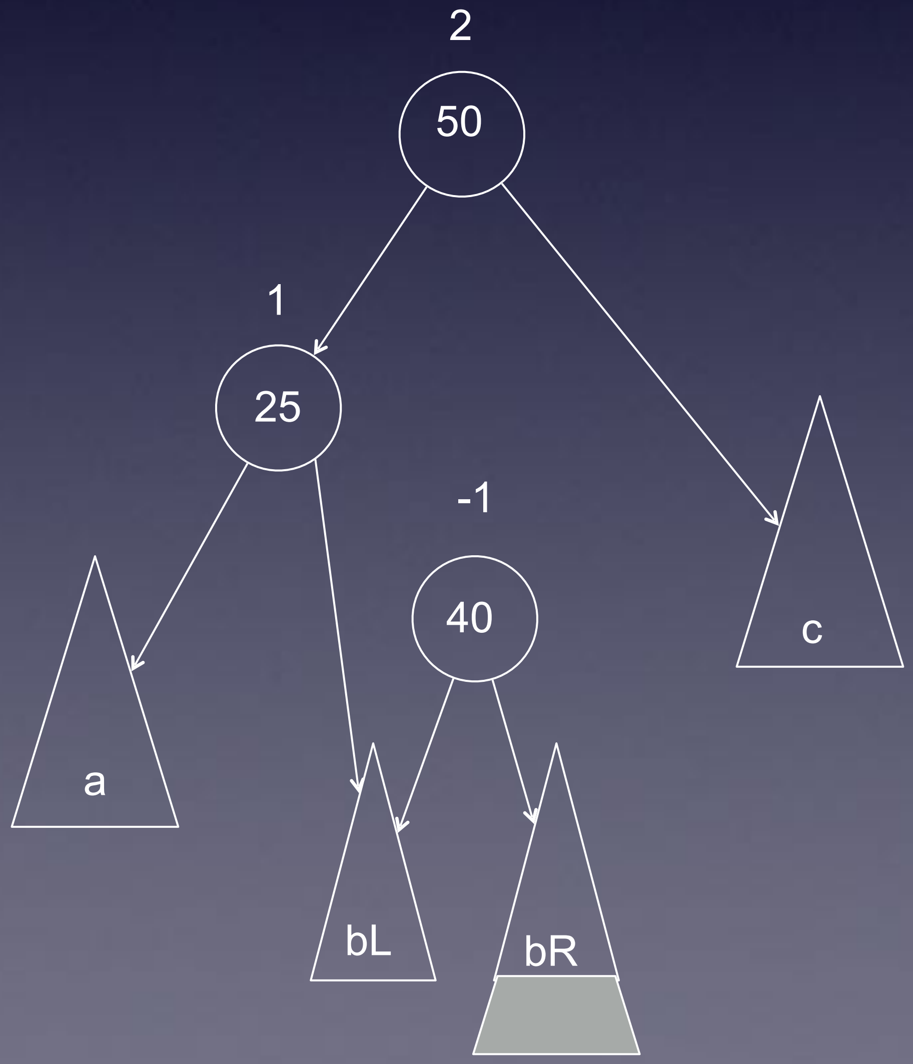
The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



Here is the other possibility.
Start by rotate left around child.

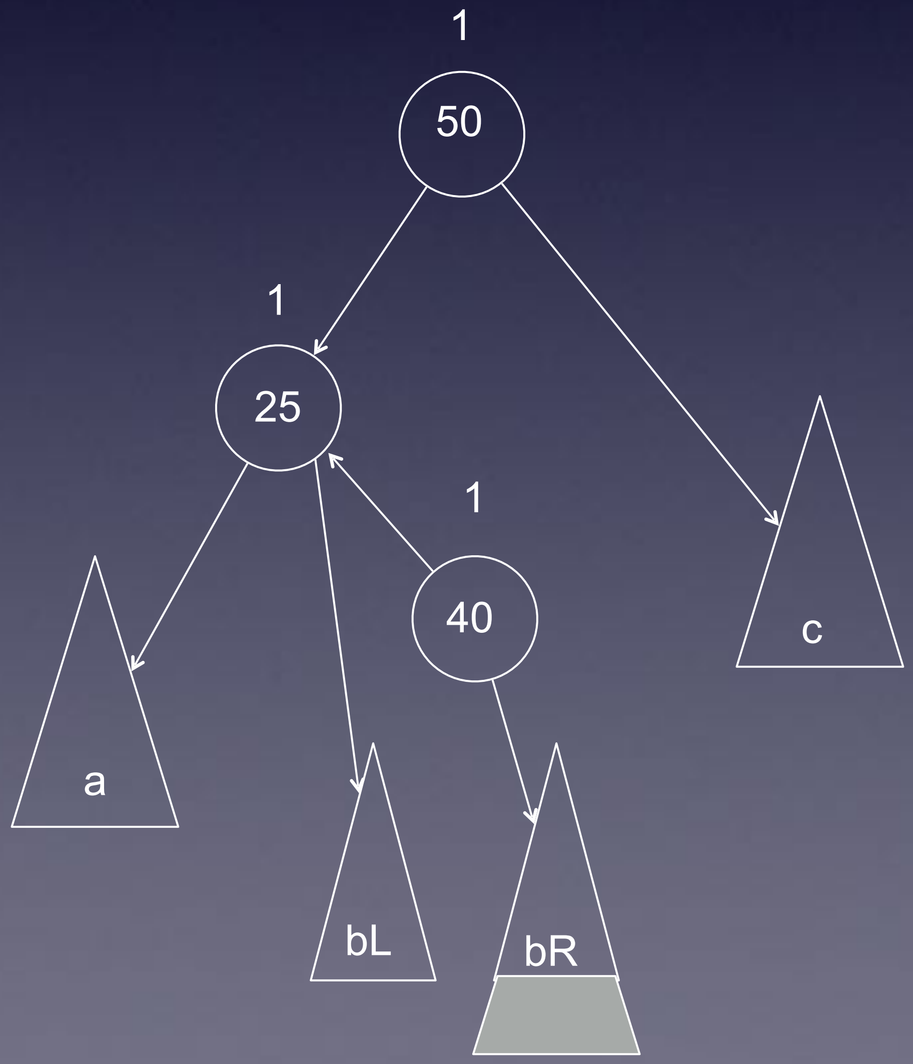
AVL trees

The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



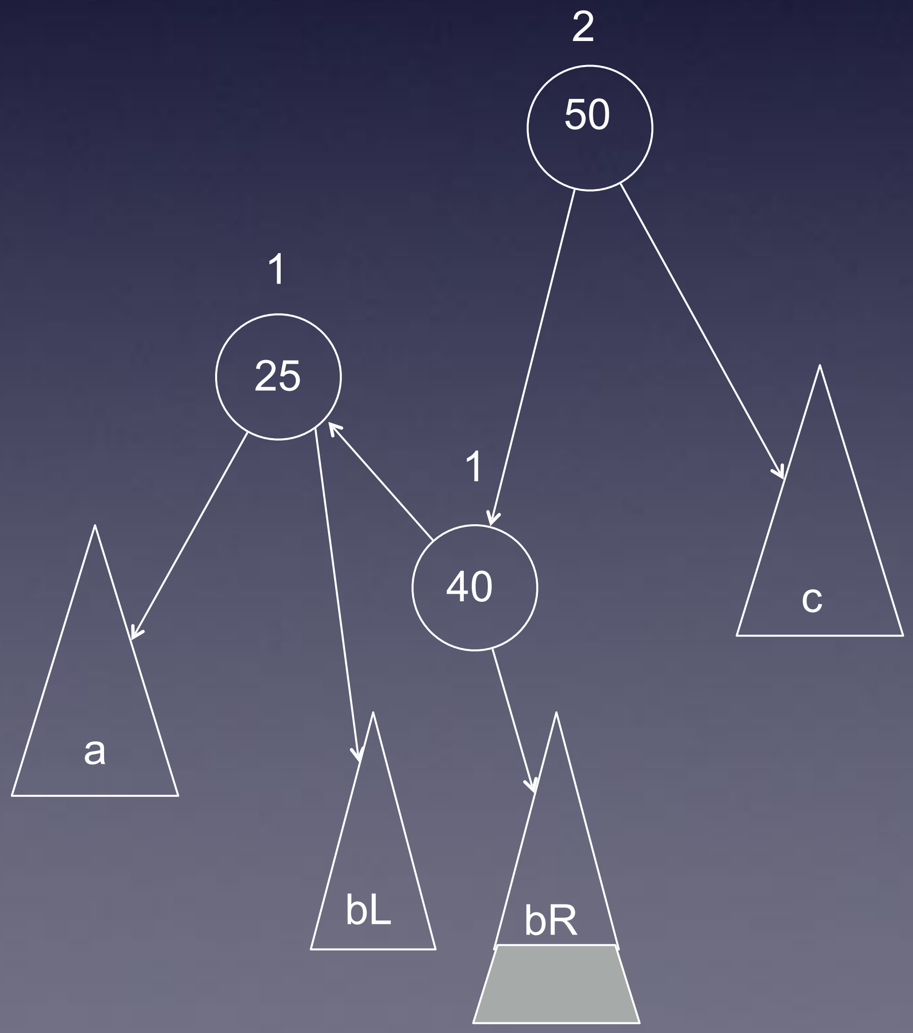
AVL trees

The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



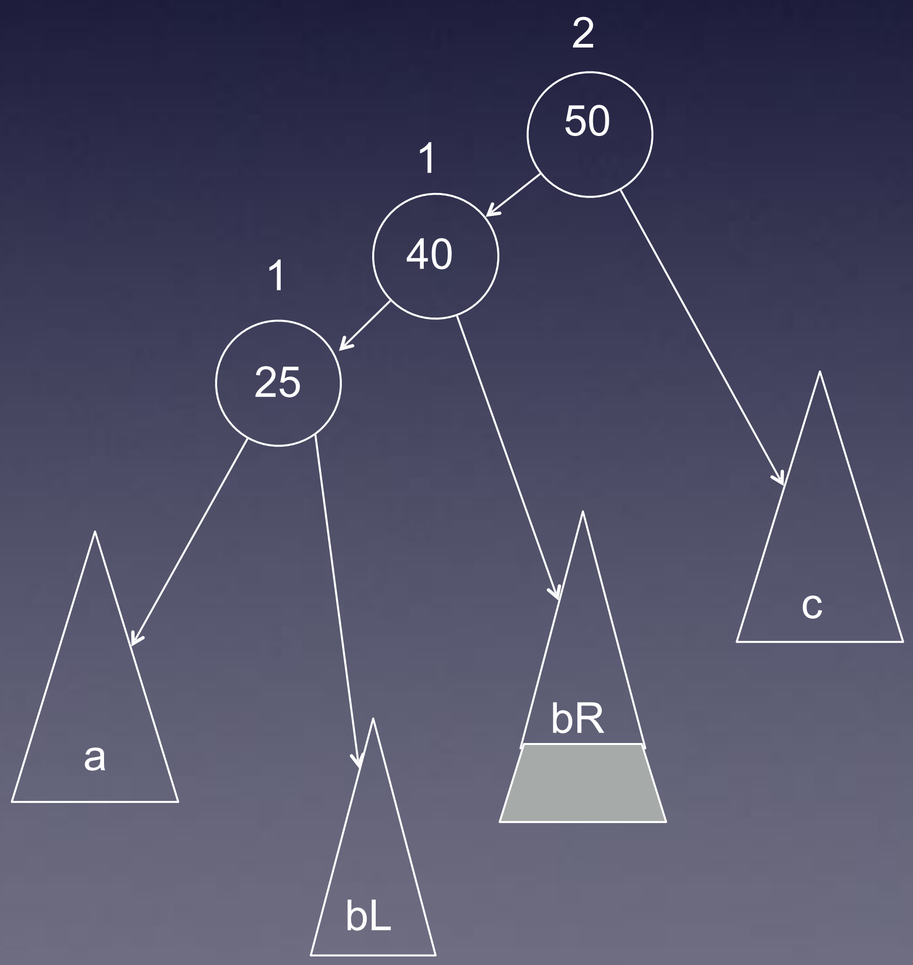
AVL trees

The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



AVL trees

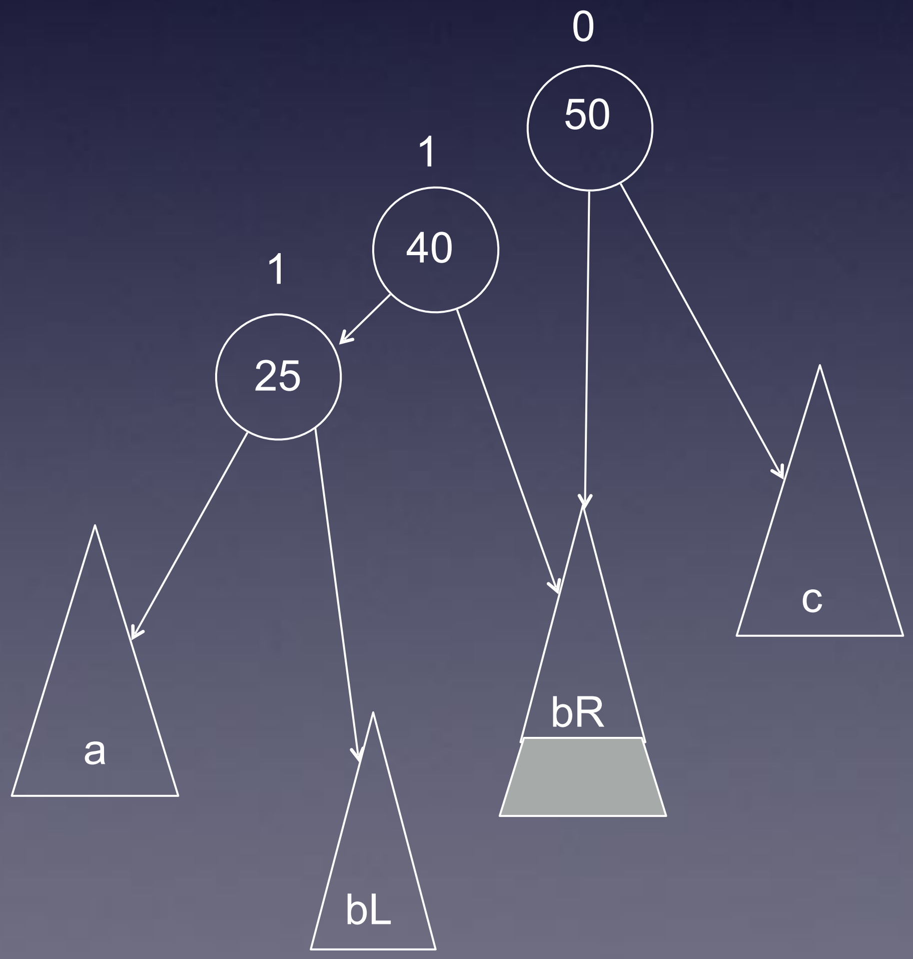
The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



Now right rotation around parent

AVL trees

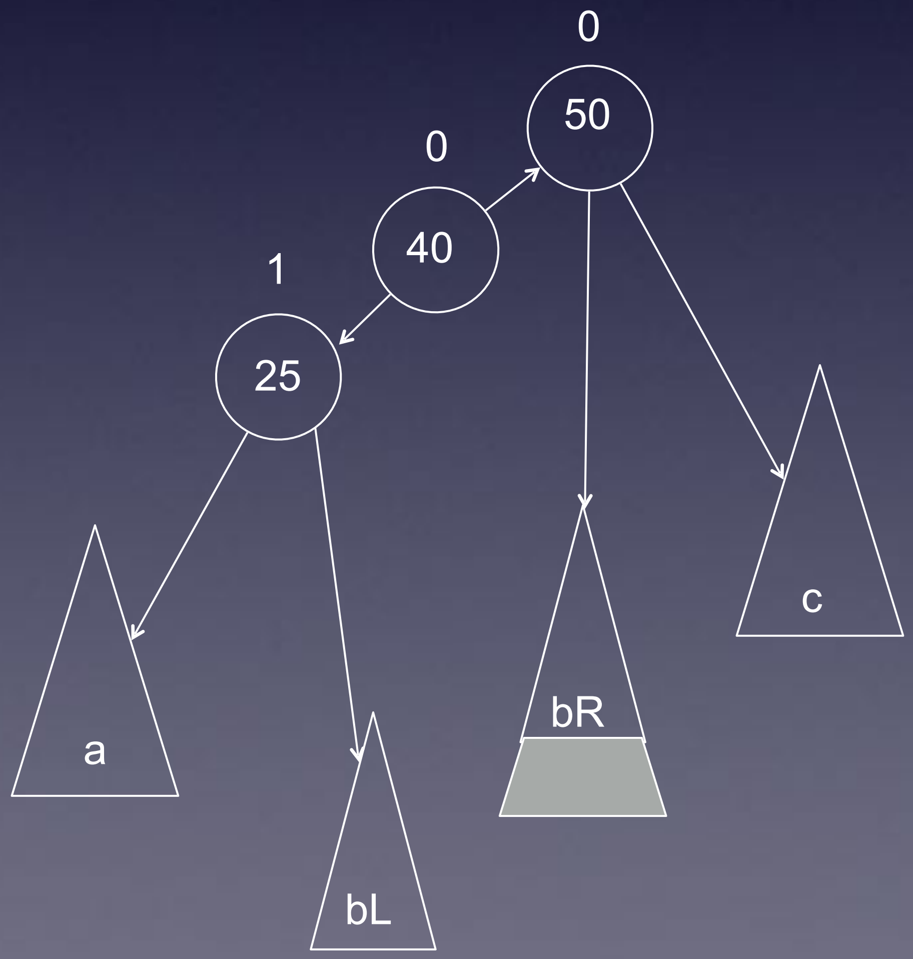
The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



Now right rotation around parent

AVL trees

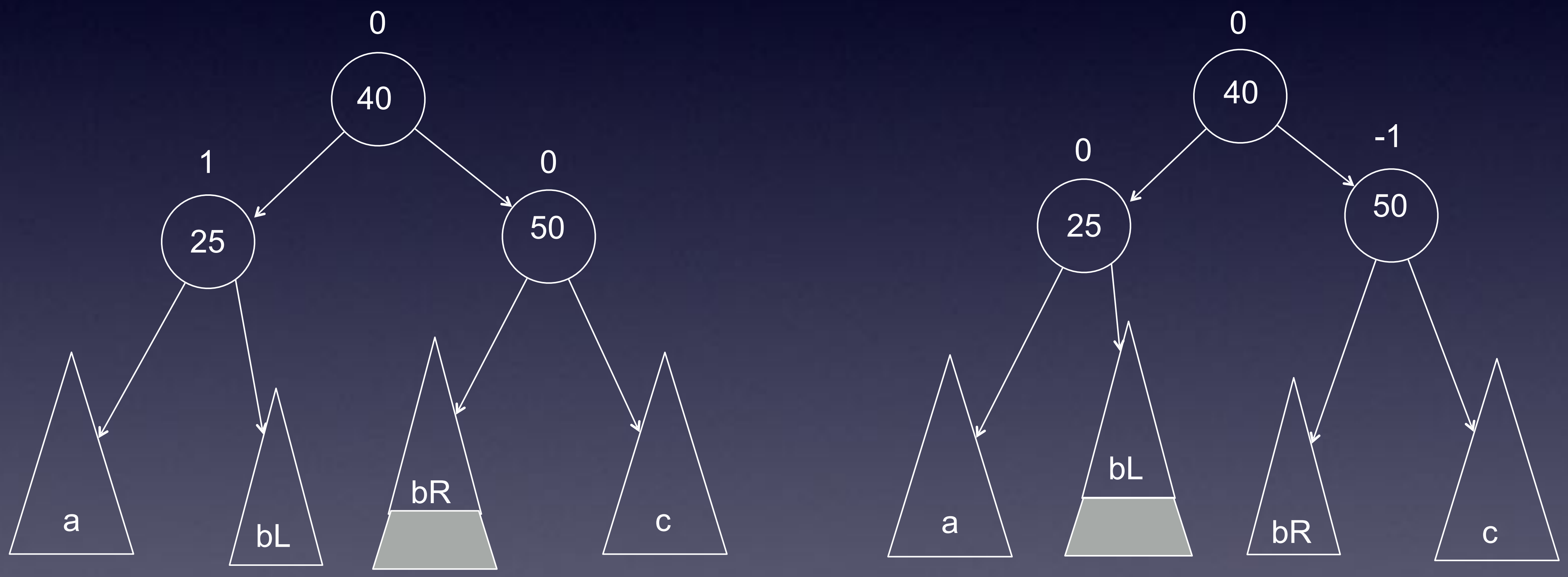
The **Left-Right** tree (parent and is left heavy and left child is right heavy, parent balance is +2, left child balance is -1). Fix by rotating left around the child then right around the parent.



Now right rotation around parent

AVL trees

Compare to the result of adding the new node to the left half of subtree b from earlier:



The steps were exactly the same.
Only the final balance is different.

AVL trees

The remaining two cases of unbalanced trees are just mirror images of the ones we've seen:

The **Right-Right** tree (parent and right child nodes are both right-heavy, parent balance is -2, child balance is -1). Fix by rotating left around the parent.
This is the mirror image of Left-Left.

The **Right-Left** tree (parent is right heavy and right child is left heavy, parent balance is -2, child balance is +1). Fix by rotating right around the right child then left around the parent. This is the mirror image of Left-Right.

Rule of thumb: If simply rotating around the parent won't work, you first need a rotation around the child that will become parent.

Priority Queues

Say you're at the **grocery store**.

You put a few items in your cart and head for checkout.

There are a couple of other sparsely-laden carts ahead of you, but ahead of those is the guy whose cart has enough food for the **next five Thanksgiving dinners**. Should you and the other sparsely-laden carts go ahead of him?

Priority Queues

You click ‘send’ on an email to your significant other to say you’ll meet them in 15 minutes at your favorite pub.

But just before you clicked that button, another person using the same email system clicked ‘send’ on the **uncompressed file of all the digital video he’s taken since 2007**.

Whose email should go first?

Priority Queues

You put your one page to-do list in the printer queue right after the student in the next office hit the 'print' button on **her 957-page doctoral dissertation with annotated bibliography**.

Which one should get printed first?

Solution

We want to solve these problems by searching the queue for “little jobs” that can be done before the “big jobs”.

If that dissertation in the printer queue holds up the small print requests of 50 other people, you have 50 unhappy people.

Alternatively, if those 50 jobs go first, then at most you have 1 unhappy person.

And that person isn’t expecting speedy printing anyway, so she probably won’t notice.

Priority Queues

If the jobs/requests are being fed into a **traditional queue**, you can extract the "small jobs" by performing a **sequential search** on the data structure.

As the queue gets bigger, it takes more time to search for the "small jobs" ... time that could be spent on processing the jobs.

There is a faster way of getting at the "small jobs".

The **heap data structure** will help us get there. We'll use it as a queue, but it's not a linear data structure...it's a tree.

Heaps

A **heap** is a **binary tree** with an **order property** and a **structure property**.

The **heap structure property**: A heap is a **complete binary tree**, this a tree in which each level has all of its nodes. The only exception is the bottom level of the tree which must be filled in from left to right.

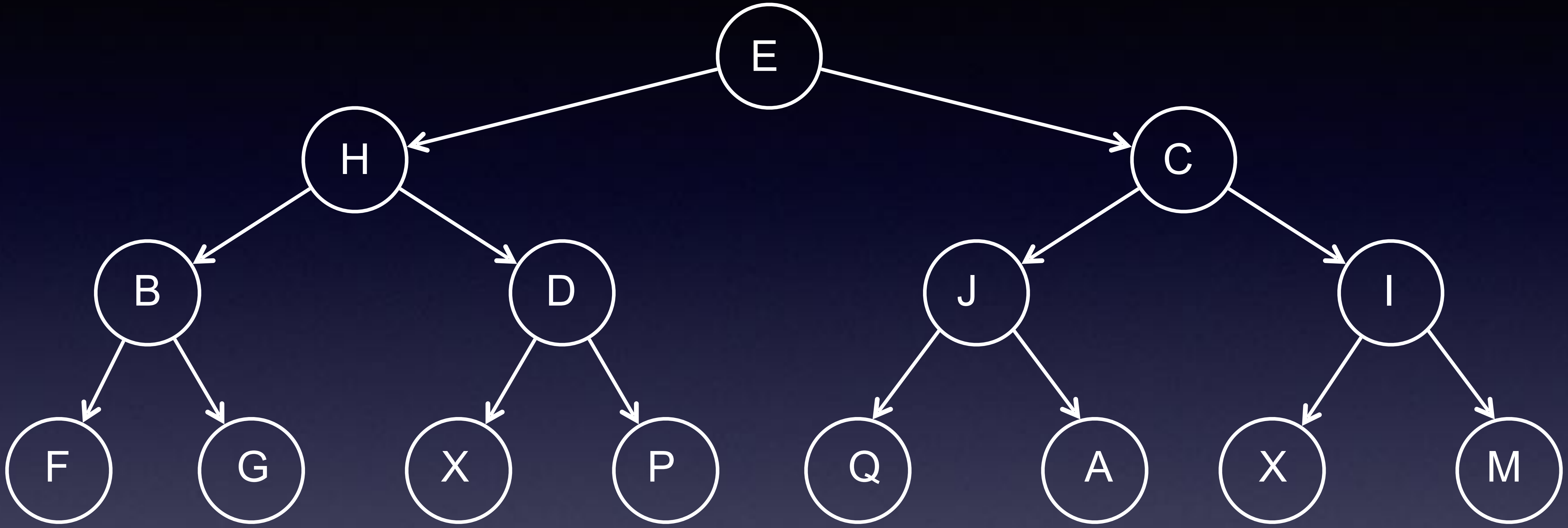
The **heap order property**: If A is a parent node of B, then the value (or key) at A is **ordered** with respect to the value (or key) of B, and the same ordering applies throughout the heap.

Heap Order Property

If the values (or keys) of **parent nodes are always greater than or equal to those of the child nodes** and the largest value (or key) is at the root node, we have a **max heap**.

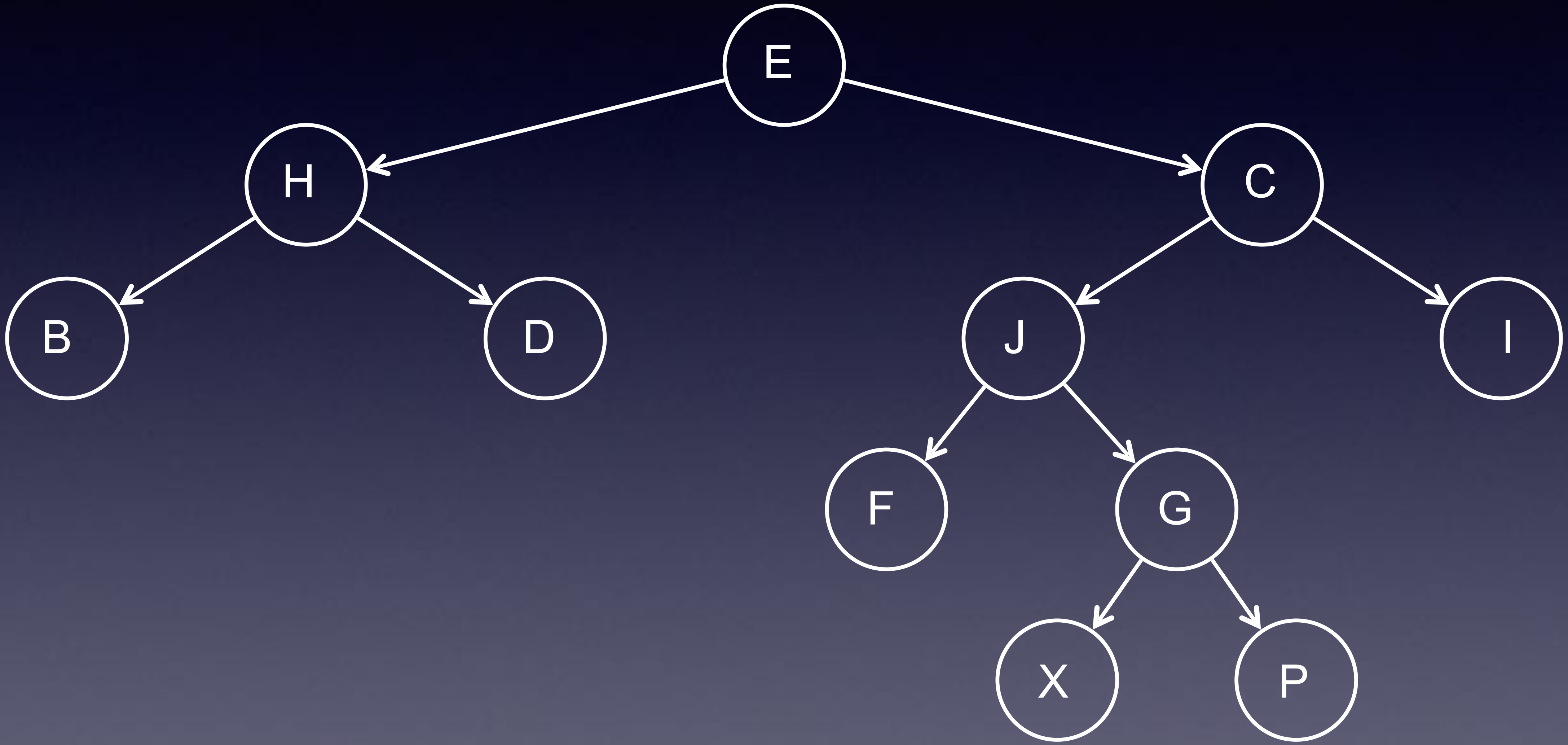
If the values (or keys) of **parent nodes are always less than or equal to those of the child nodes** and the smallest value (or key) is at the root node, we have a **min heap**.

Complete Binary Tree



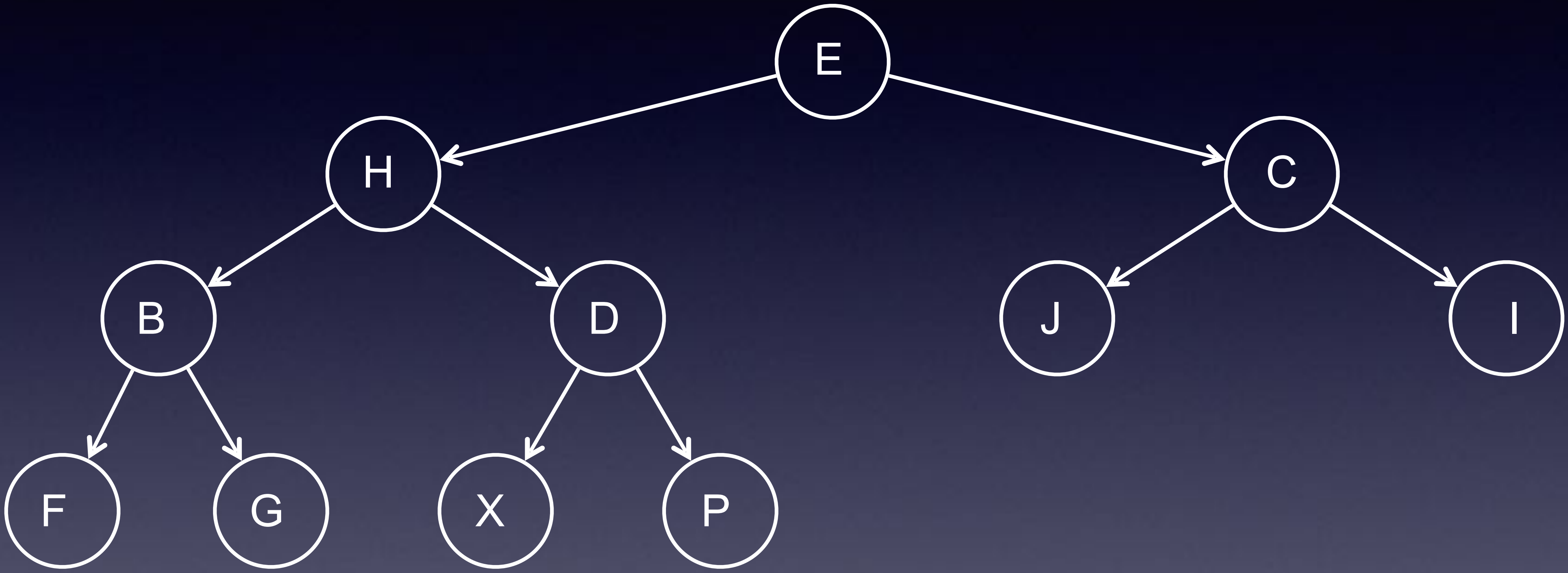
Complete? Yes. Each level has all of its nodes

Complete Binary Tree



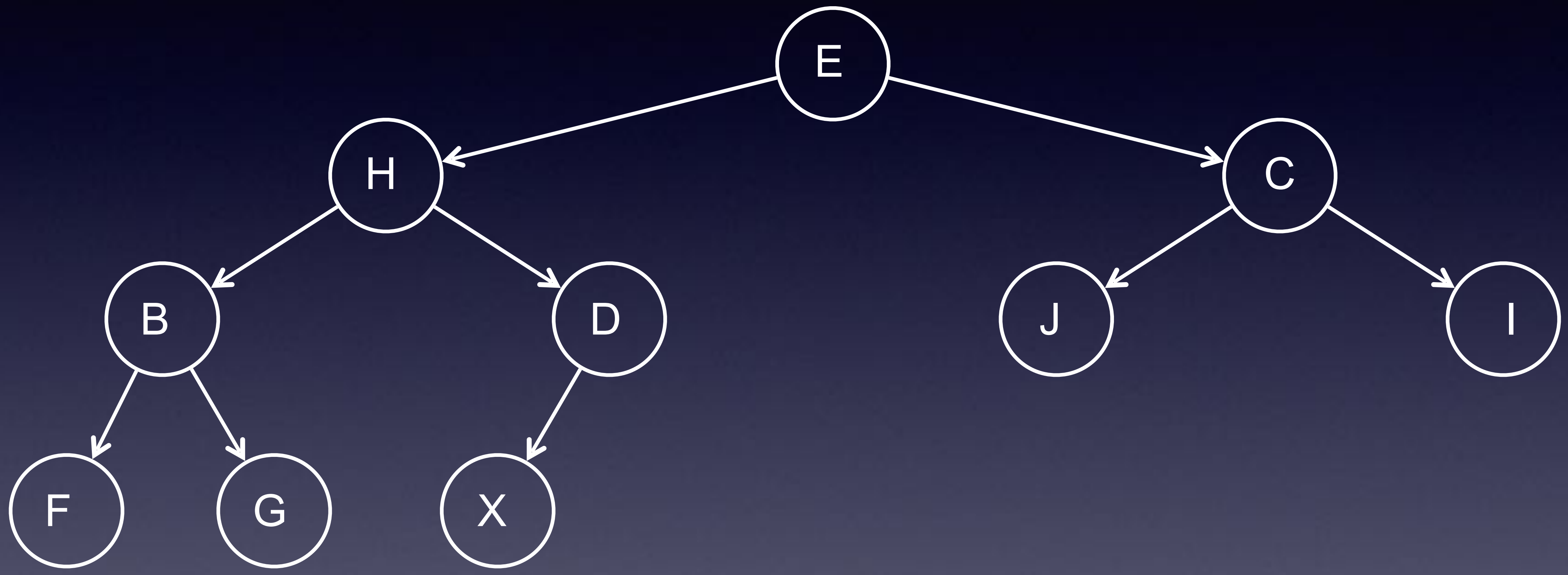
Complete? No

Complete Binary Tree



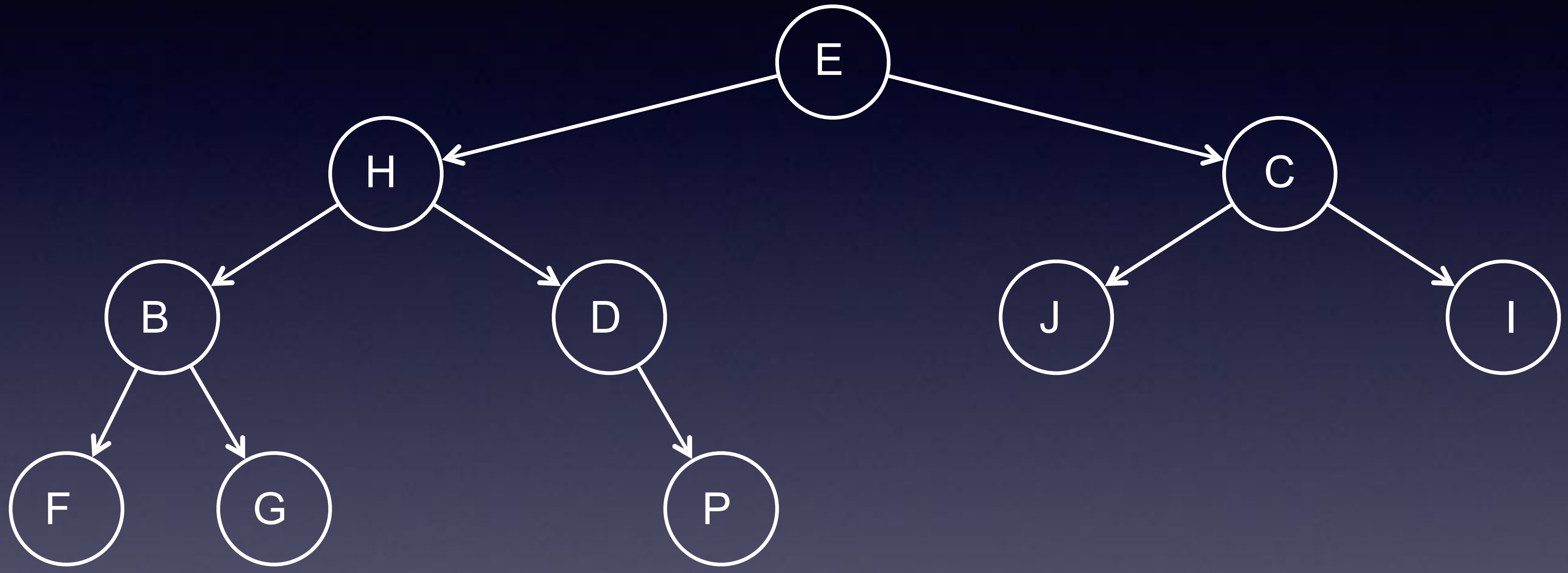
Complete? Yes. Each level has all of its nodes with the exception of the last which is filled in from left to right.

Complete Binary Tree



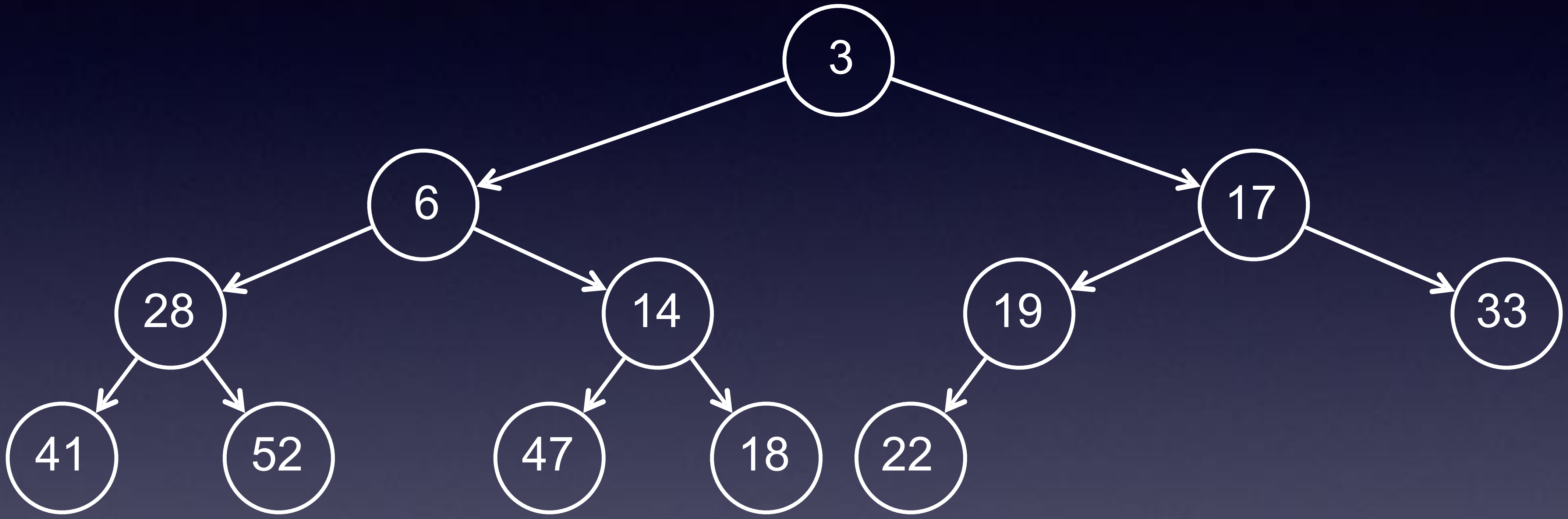
Complete?

Complete Binary Tree



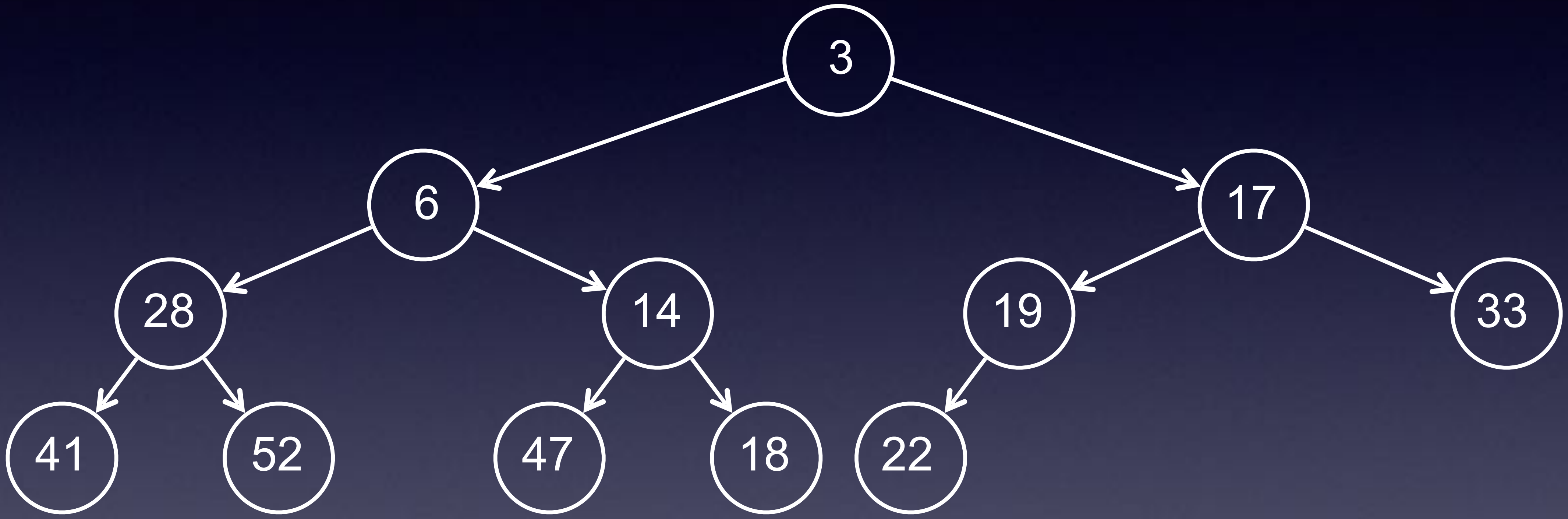
Complete?

Heaps



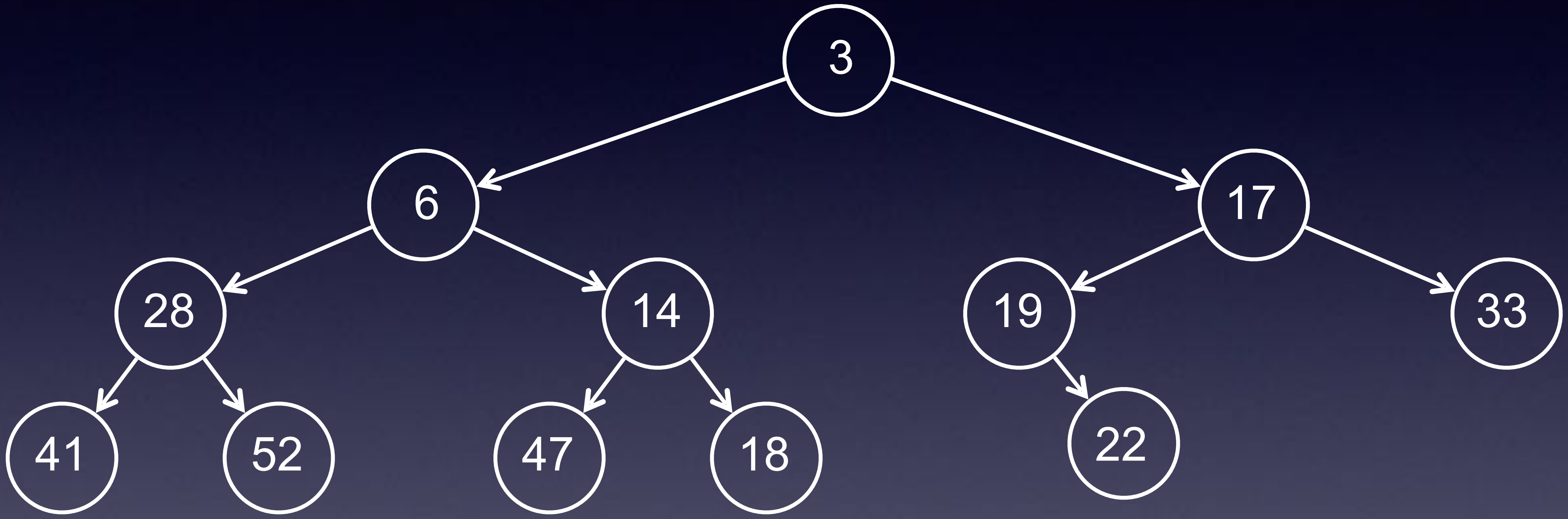
Here is a **min heap**

Heaps



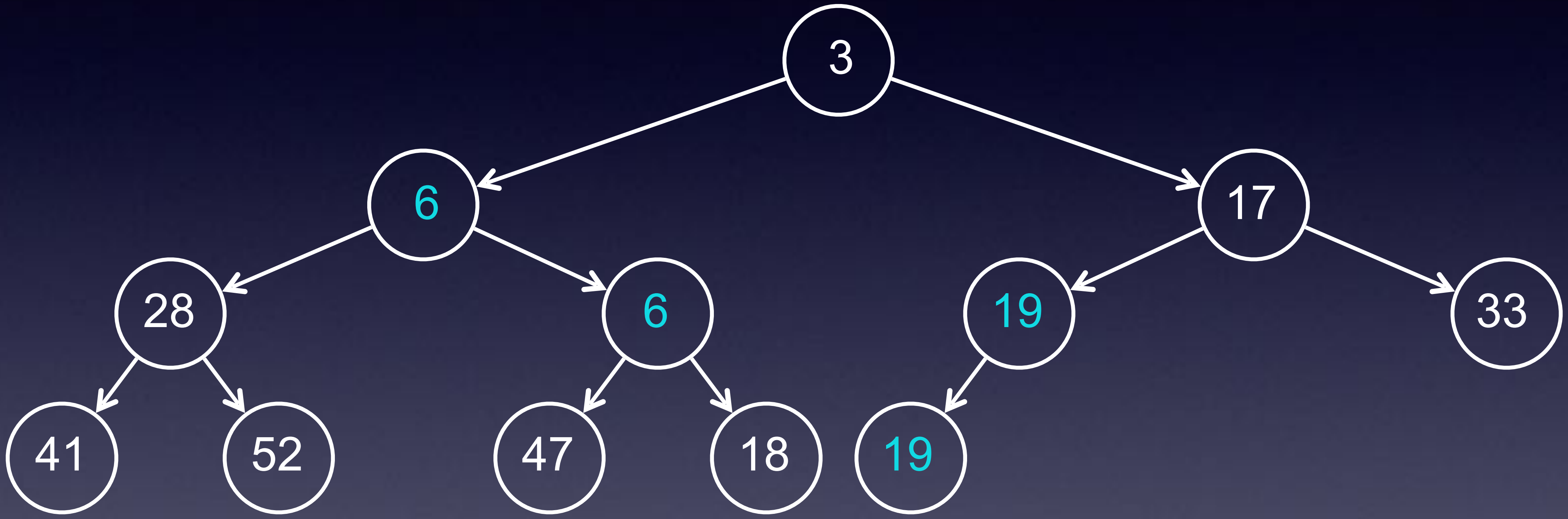
There is a top-to-bottom ordering, but there is no side-to-side ordering as we would see in a binary search tree.

Heaps



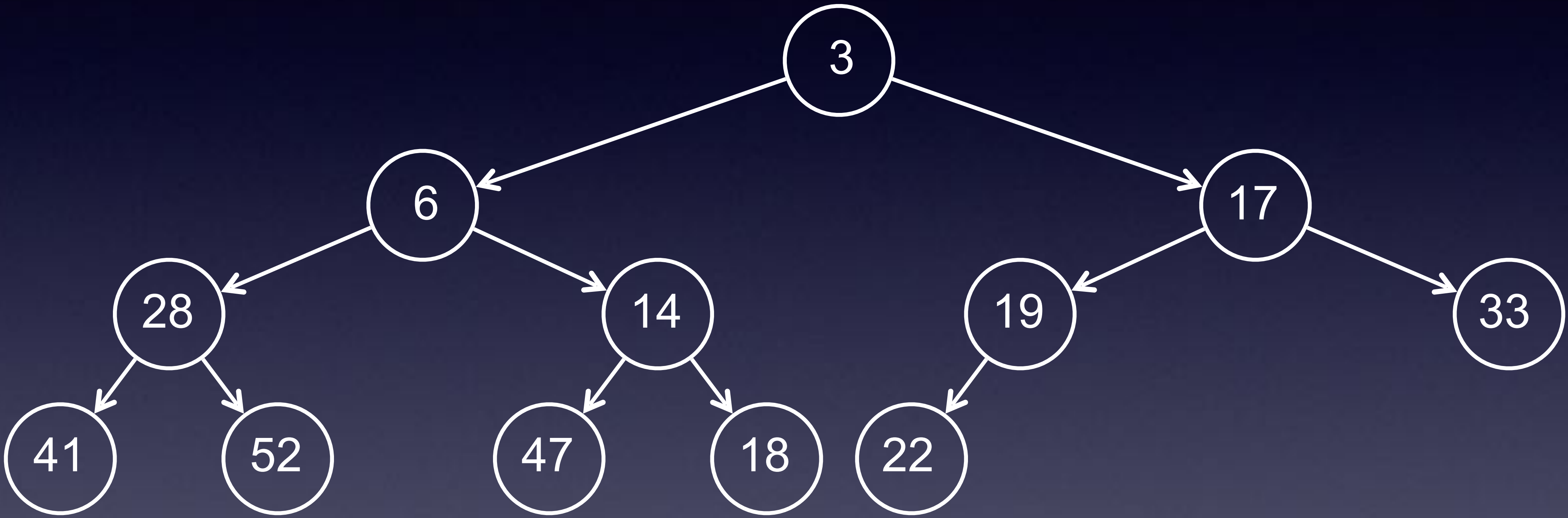
This is not a heap. Why?

Heaps



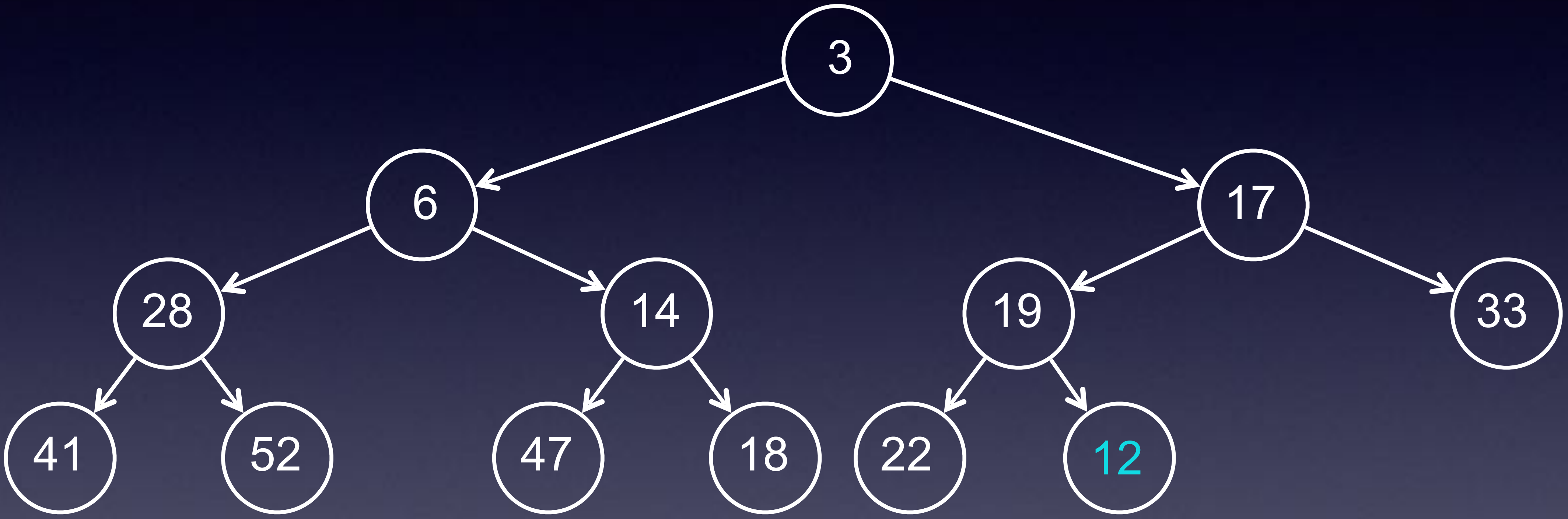
A heap may contain the same values in different locations

Insertion in a heap



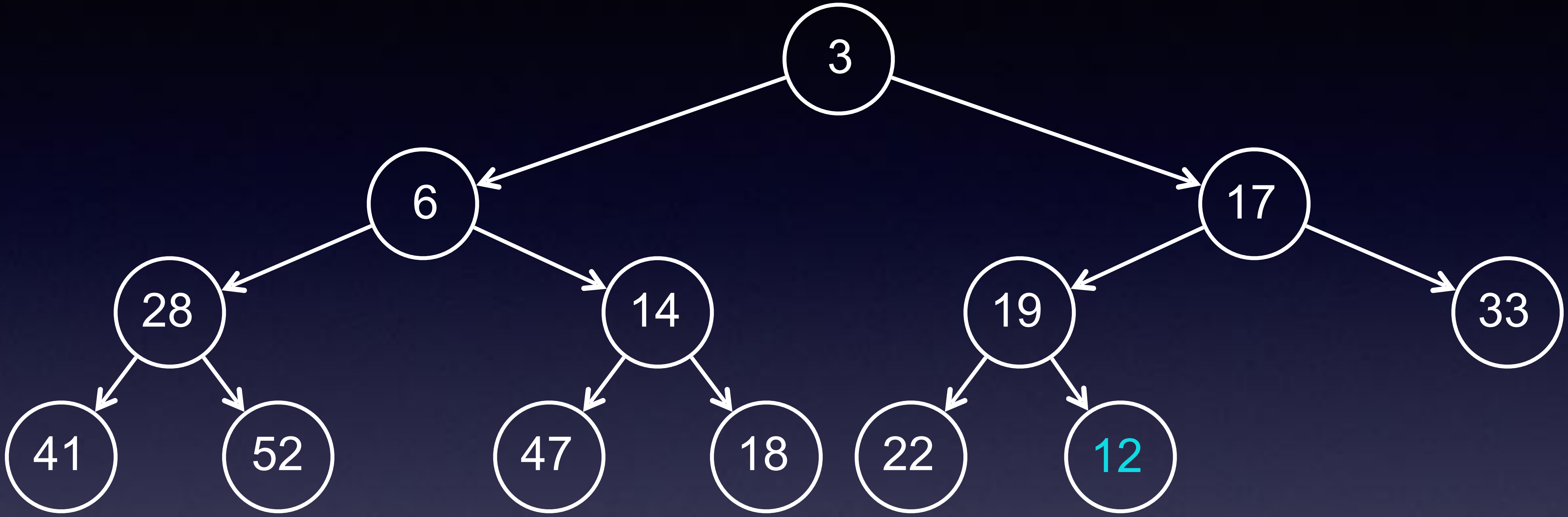
Inserting a new item into a min heap starts with adding the item to the heap at the next available location in the complete binary tree.

Insertion in a heap



Inserting a new item into a min heap starts with adding the item to the heap at the next available location in the complete binary tree.

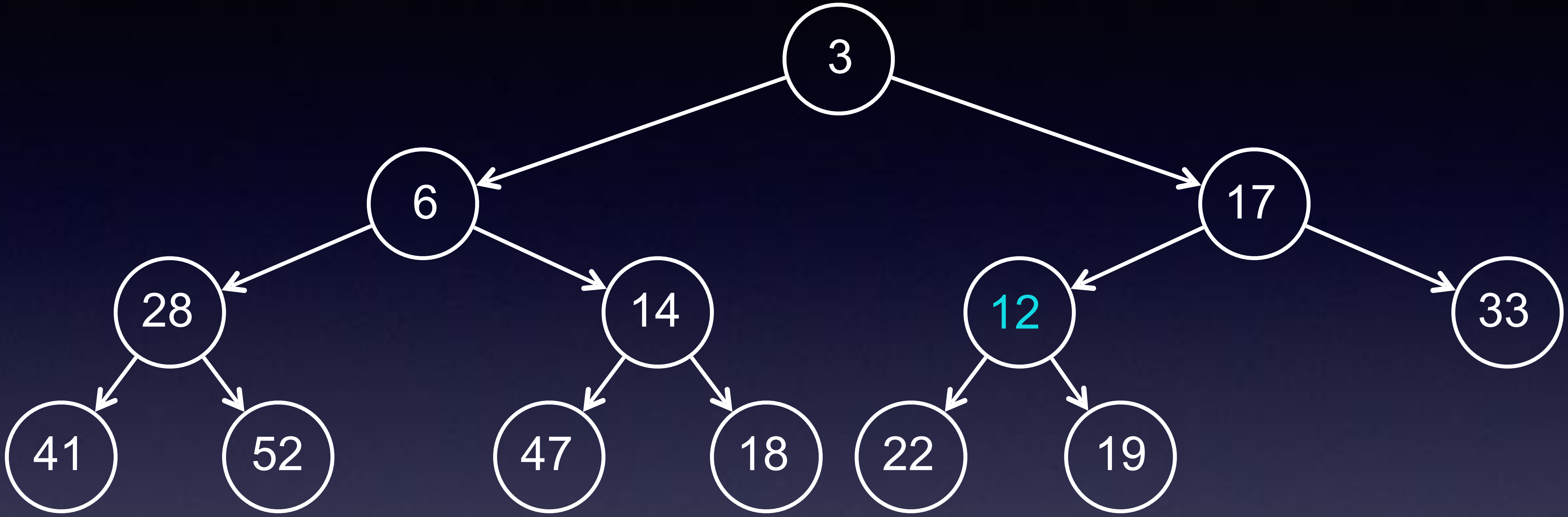
Insertion in a heap



Algorithm for insertion into a min heap:

```
1. insert the new item in the next position at the bottom of the heap
2. while new item is not at the root and new item is smaller than its parent
3.     swap the new item with its parent, moving the new item up the heap
```

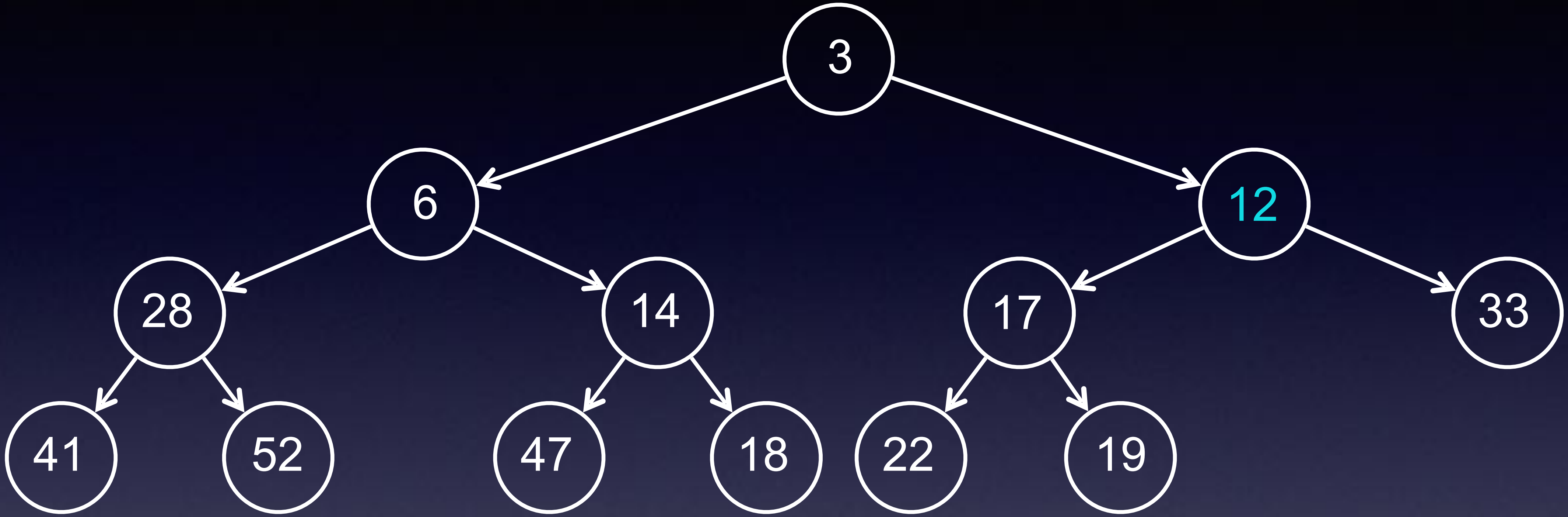
Insertion in a heap



Algorithm for insertion into a min heap:

```
1.  insert the new item in the next position at the bottom of the heap
2.  while new item is not at the root and new item is smaller than its parent
3.      swap the new item with its parent, moving the new item up the heap
```

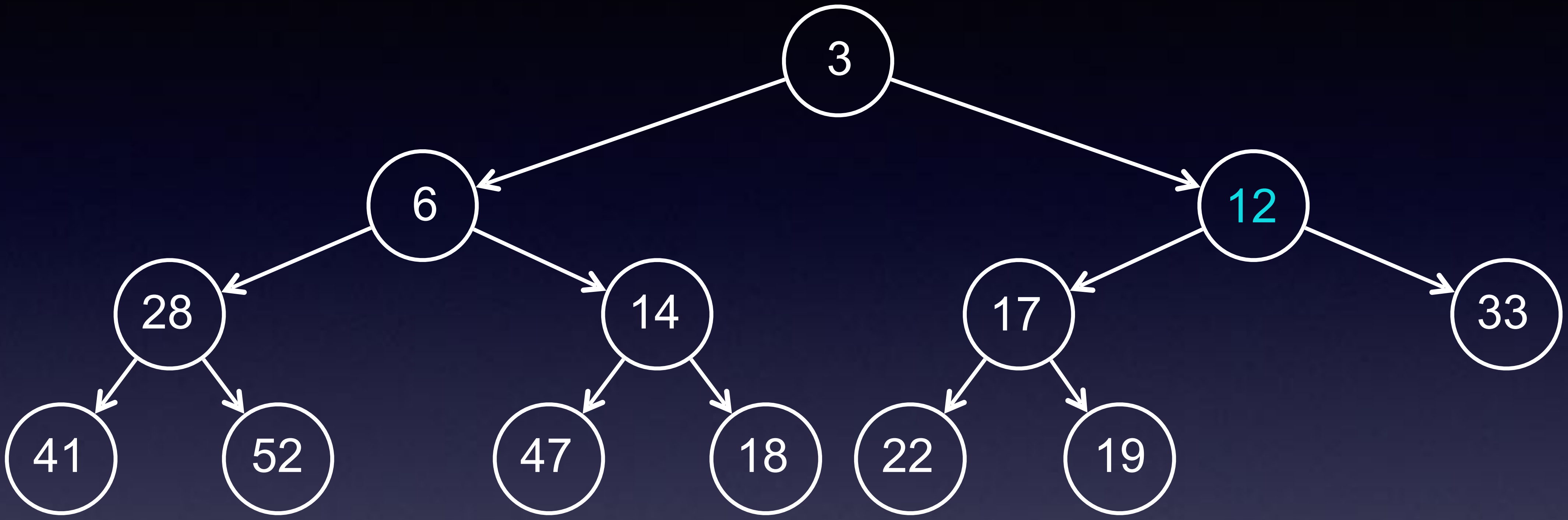
Insertion in a heap



Algorithm for insertion into a min heap:

```
1. insert the new item in the next position at the bottom of the heap
2. while new item is not at the root and new item is smaller than its parent
3.     swap the new item with its parent, moving the new item up the heap
```

Insertion in a heap

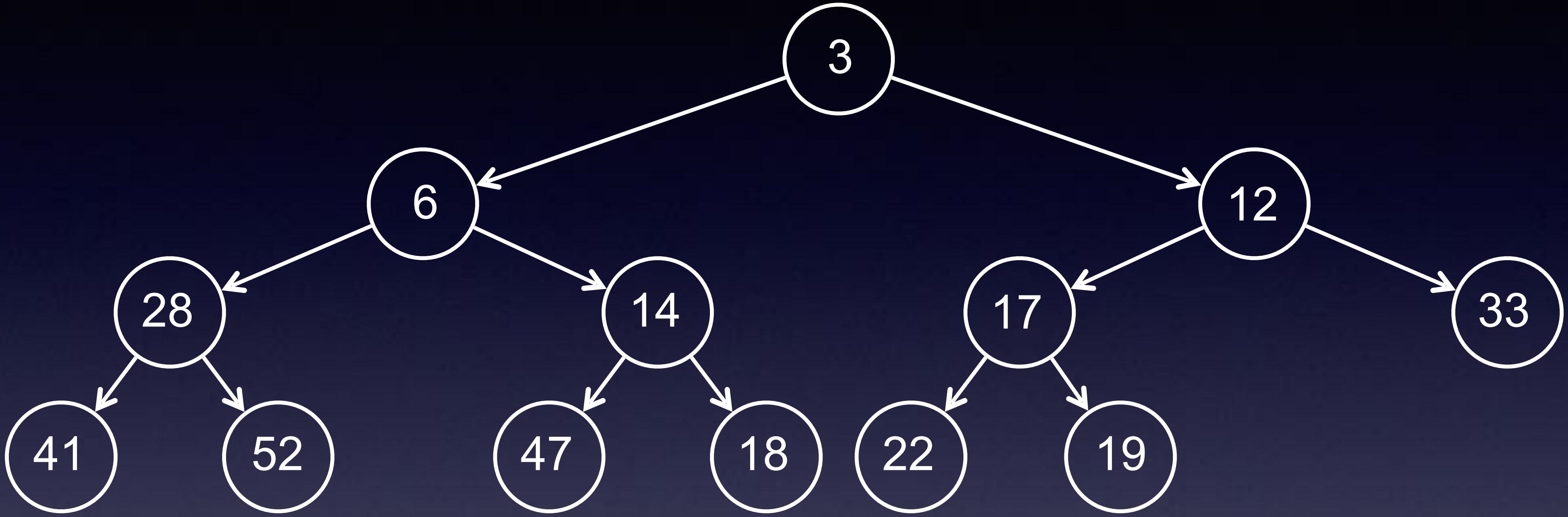


Algorithm for insertion into a min heap:

```
1. insert the new item in the next position at the bottom of the heap
2. while new item is not at the root and new item is smaller than its parent
3.     swap the new item with its parent, moving the new item up the heap
```

Our book calls this percUp (for percolate up).

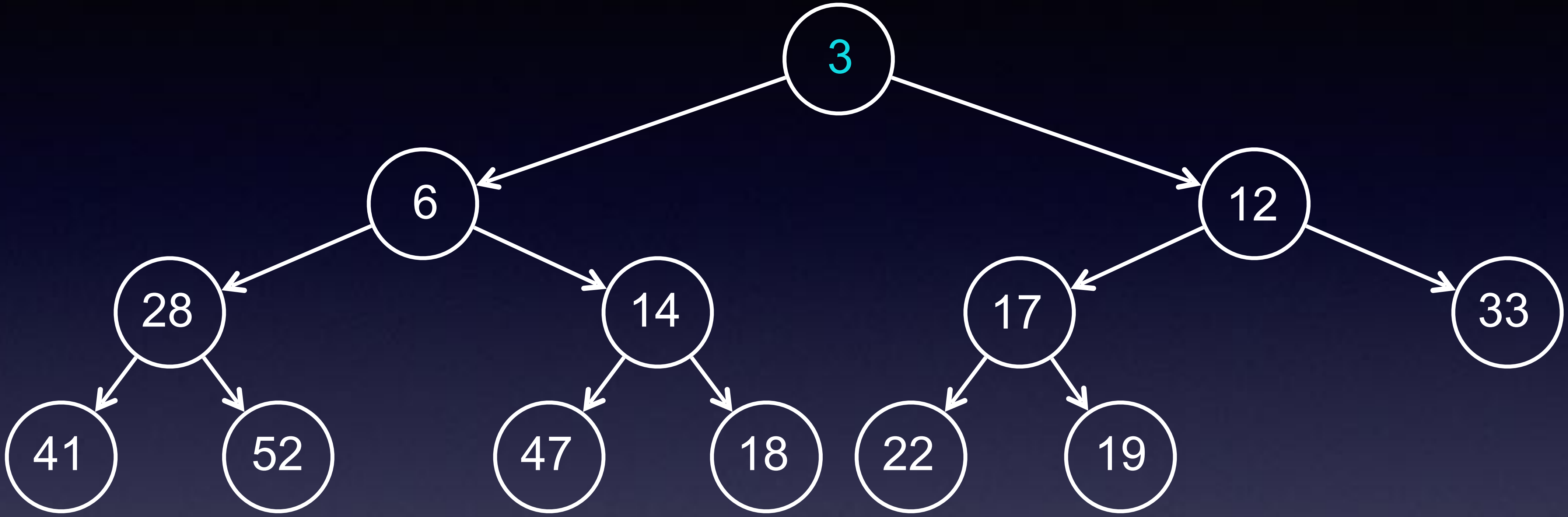
Deletion from a heap



Algorithm for deleting the root from a min heap:

```
1. replace the item at the root node with the last item in the heap (LIH)
2.  while item LIH has children and item LIH is larger than at least one child
3.      swap item LIH with the smaller of its children, moving LIH down the heap
```

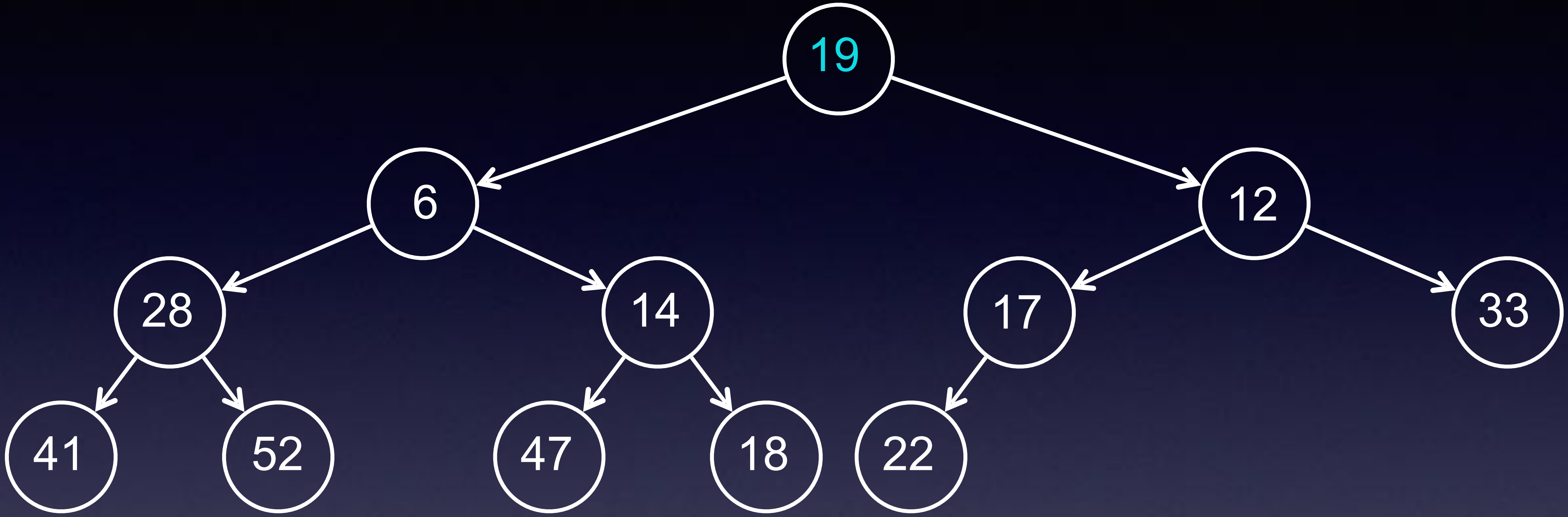
Deletion from a heap



Algorithm for deleting the root from a min heap:

```
1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3.     swap item LIH with the smaller of its children, moving LIH down the heap
```

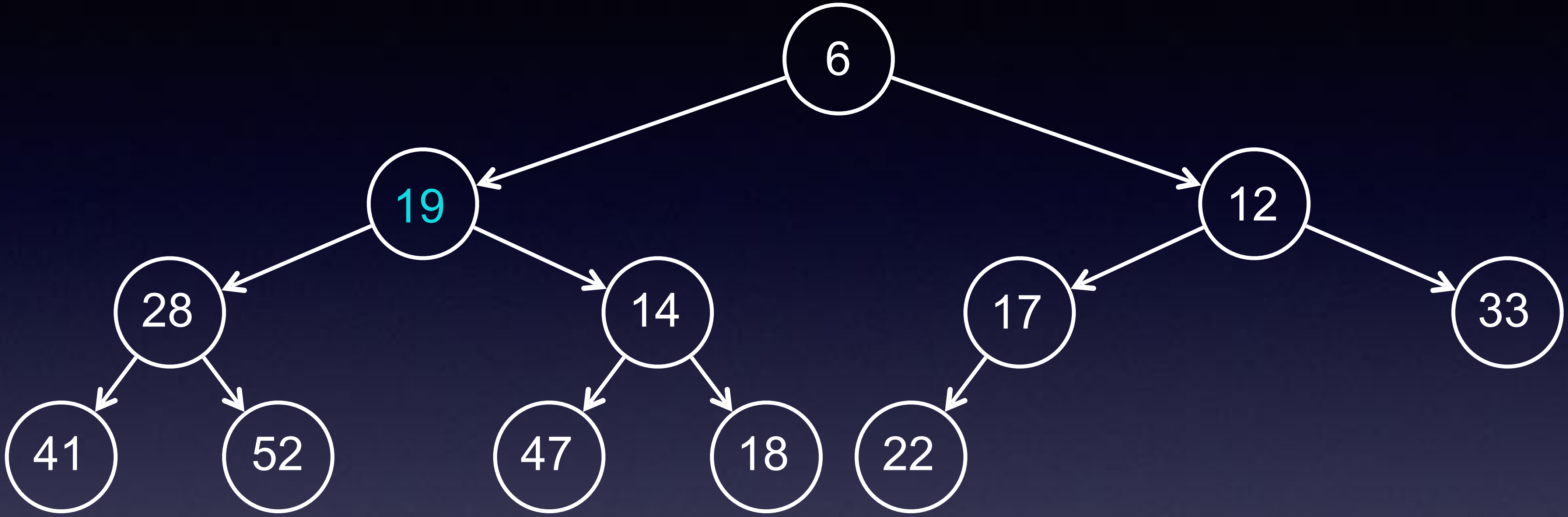
Deletion from a heap



Algorithm for deleting the root from a min heap:

```
1. replace the item at the root node with the last item in the heap (LIH)
2.  while item LIH has children and item LIH is larger than at least one child
3.      swap item LIH with the smaller of its children, moving LIH down the heap
```

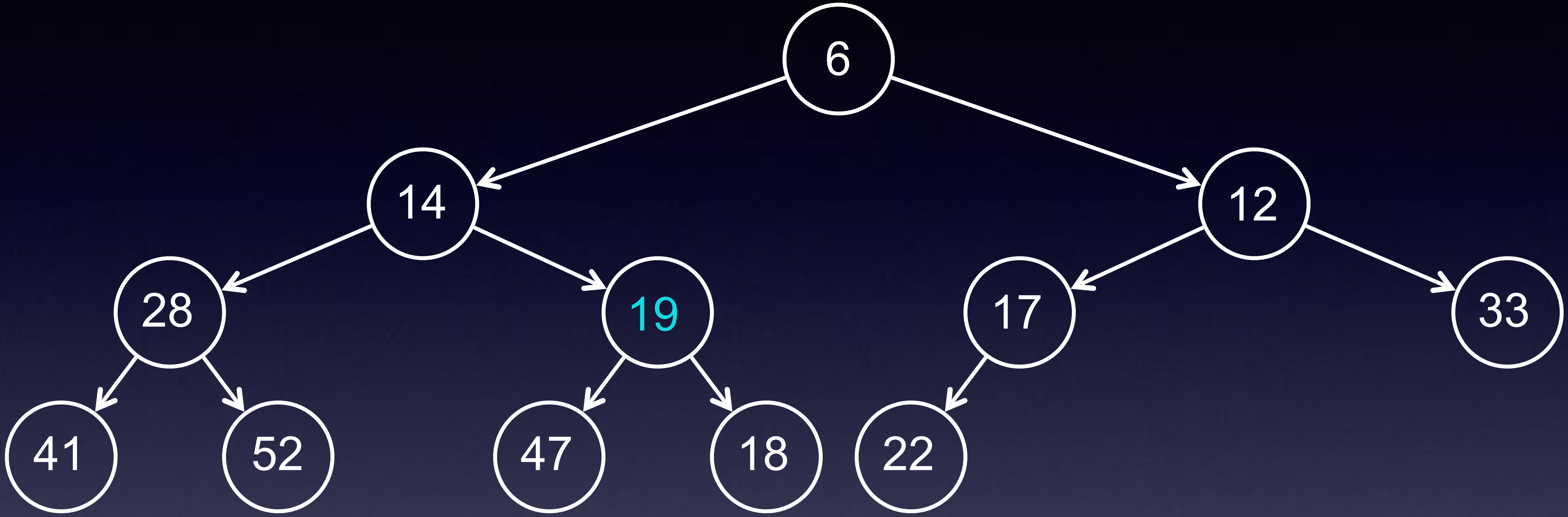
Deletion from a heap



Algorithm for deleting the root from a min heap:

1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3. swap item LIH with the smaller of its children, moving LIH down the heap

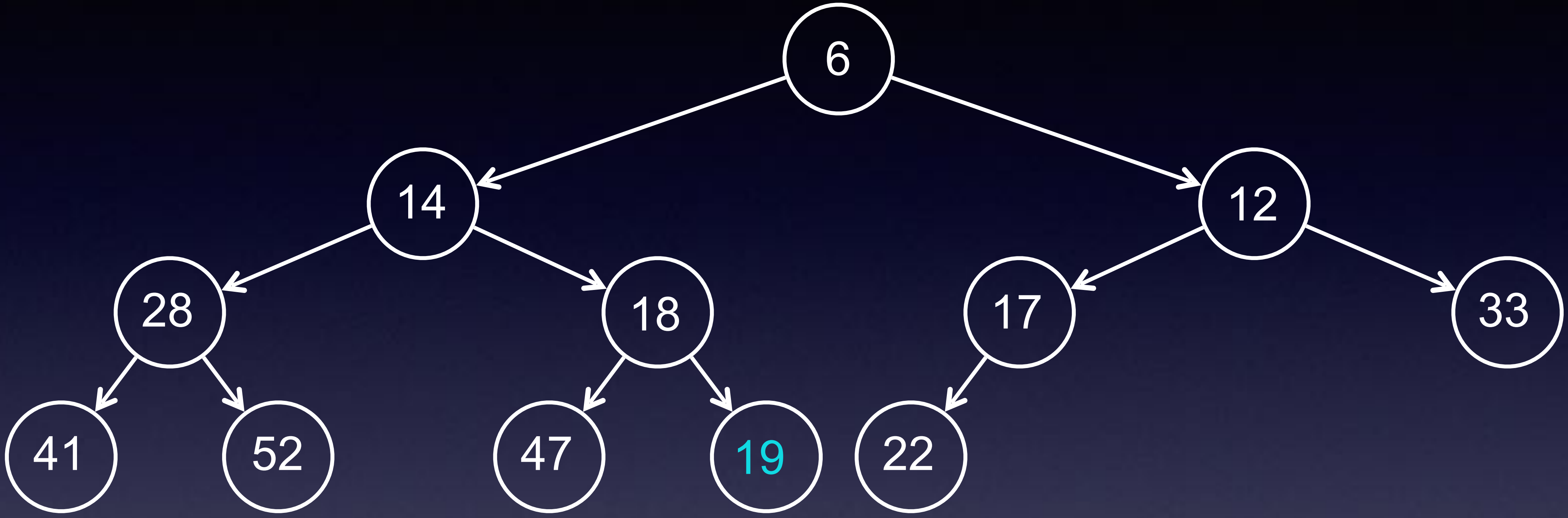
Deletion from a heap



Algorithm for deleting the root from a min heap:

```
1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3.     swap item LIH with the smaller of its children, moving LIH down the heap
```

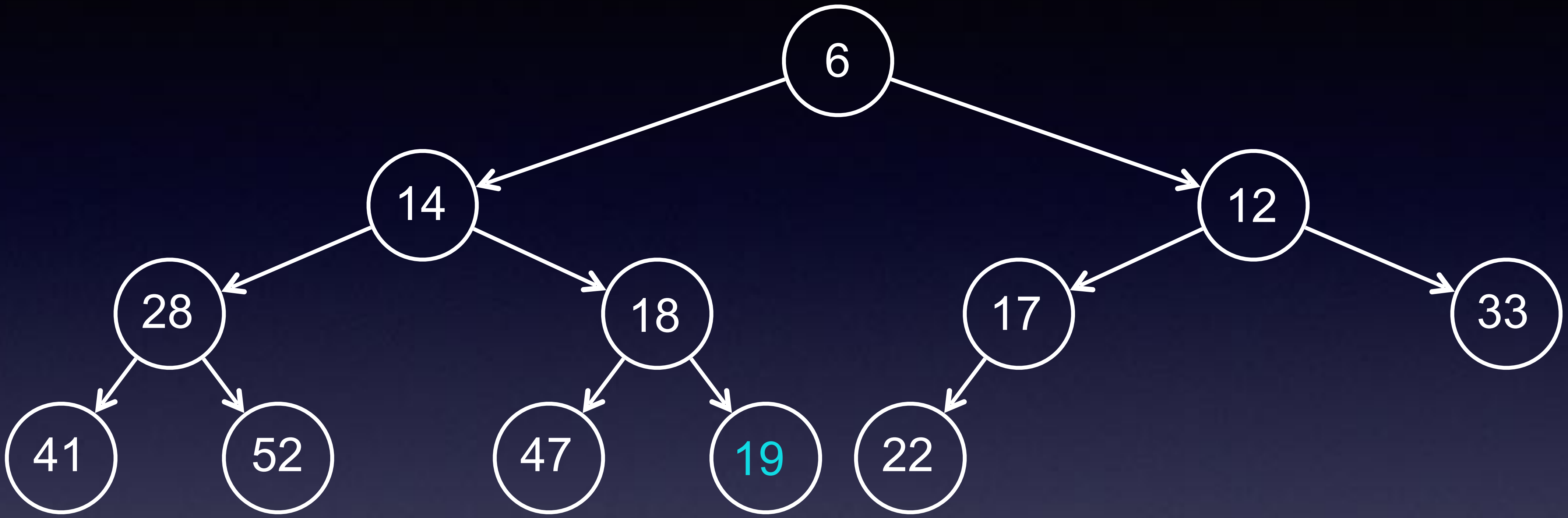
Deletion from a heap



Algorithm for deleting the root from a min heap:

1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3. swap item LIH with the smaller of its children, moving LIH down the heap

Deletion from a heap

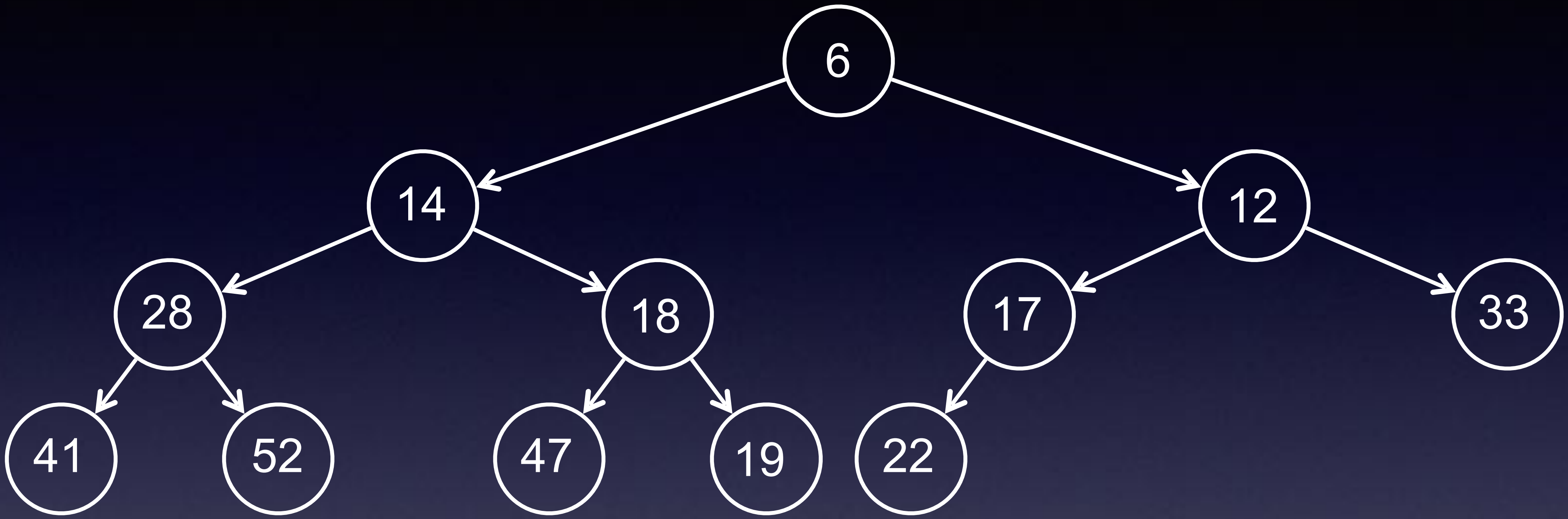


Algorithm for deleting the root from a min heap:

```
1. replace the item at the root node with the last item in the heap (LIH)
2.  while item LIH has children and item LIH is larger than at least one child
3.      swap item LIH with the smaller of its children, moving LIH down the heap
```

Our book calls this percDown (for percolate down).

Deletion from a heap

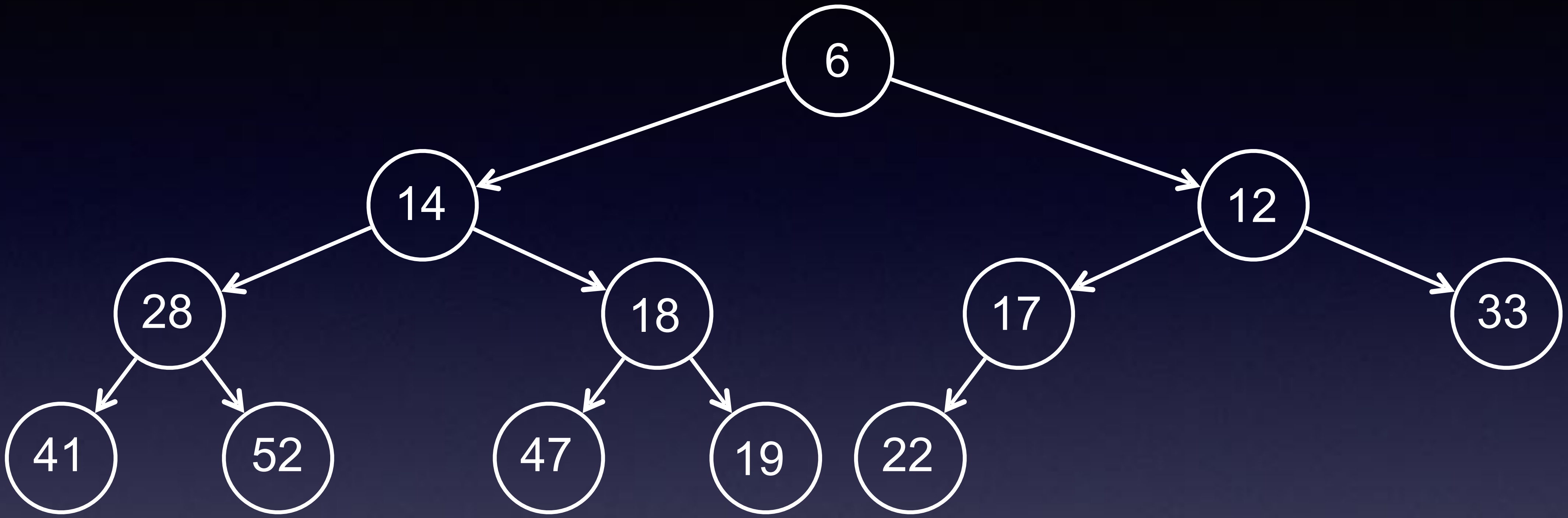


Usually, deletion is from the root. Why? That answer is coming.

What is the complexity of deletion?
Note that swapping two values is constant number of steps.
So what's the worst case for the number of swaps as a function of n?

Is this also the complexity of insertion?

Deletion from a heap



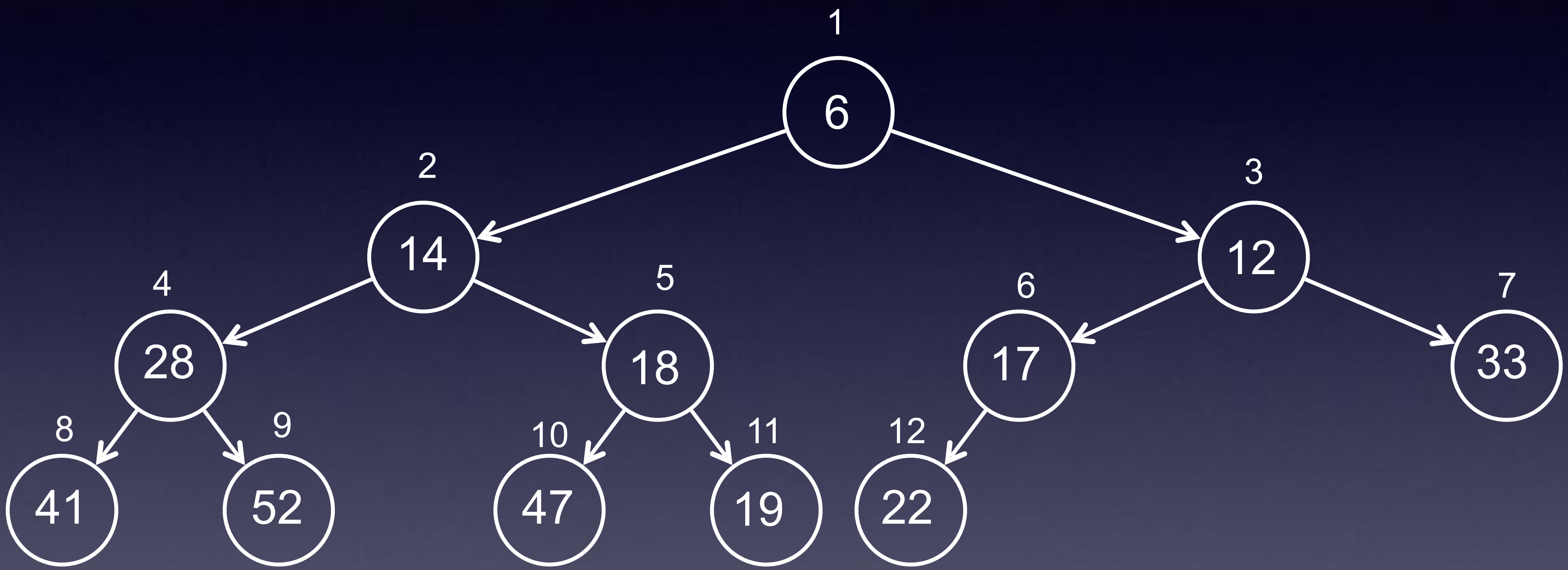
Usually, deletion is from the root. Why? That answer is coming.

What is the complexity of deletion?
Note that swapping two values is constant number of steps.
So what’s the worst case for the number of swaps as a function of n? O(log n)

Is this also the complexity of insertion? Yes

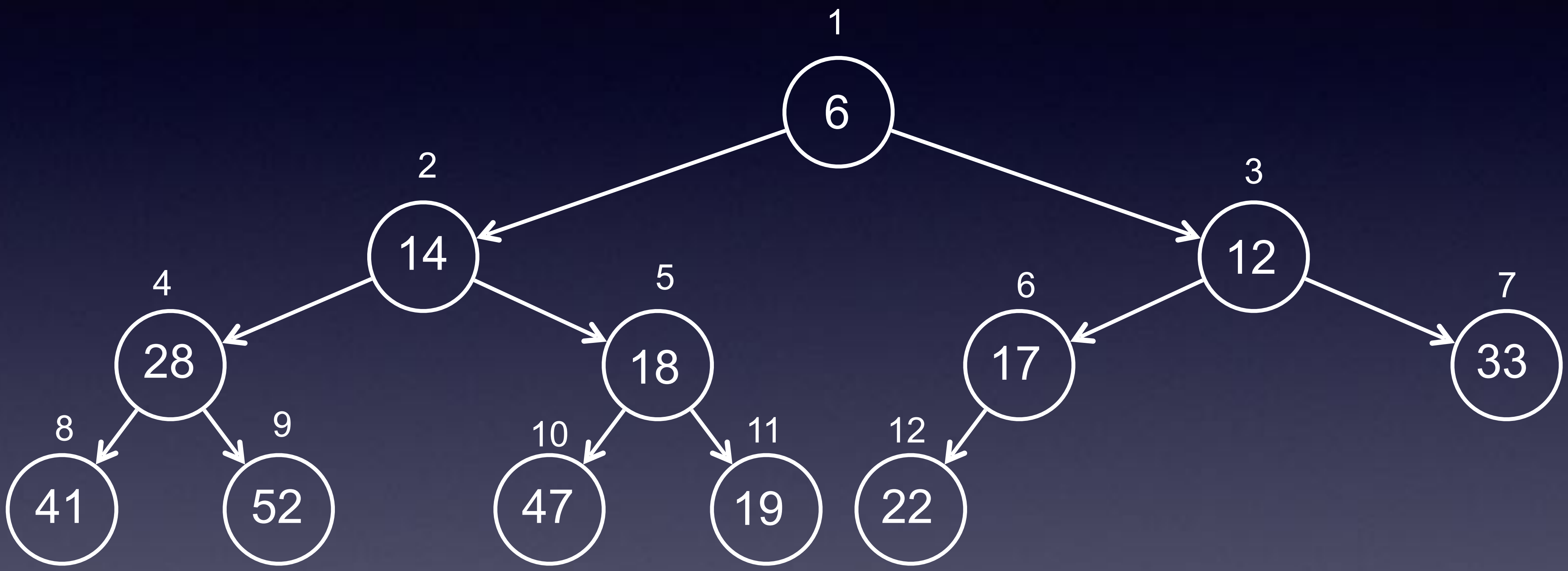
Implementing a heap

If we number the nodes in a complete binary tree by level, and left to right within a level, we get:



Implementing a heap

If we number the nodes in a complete binary tree by level, and left to right within a level, we get:

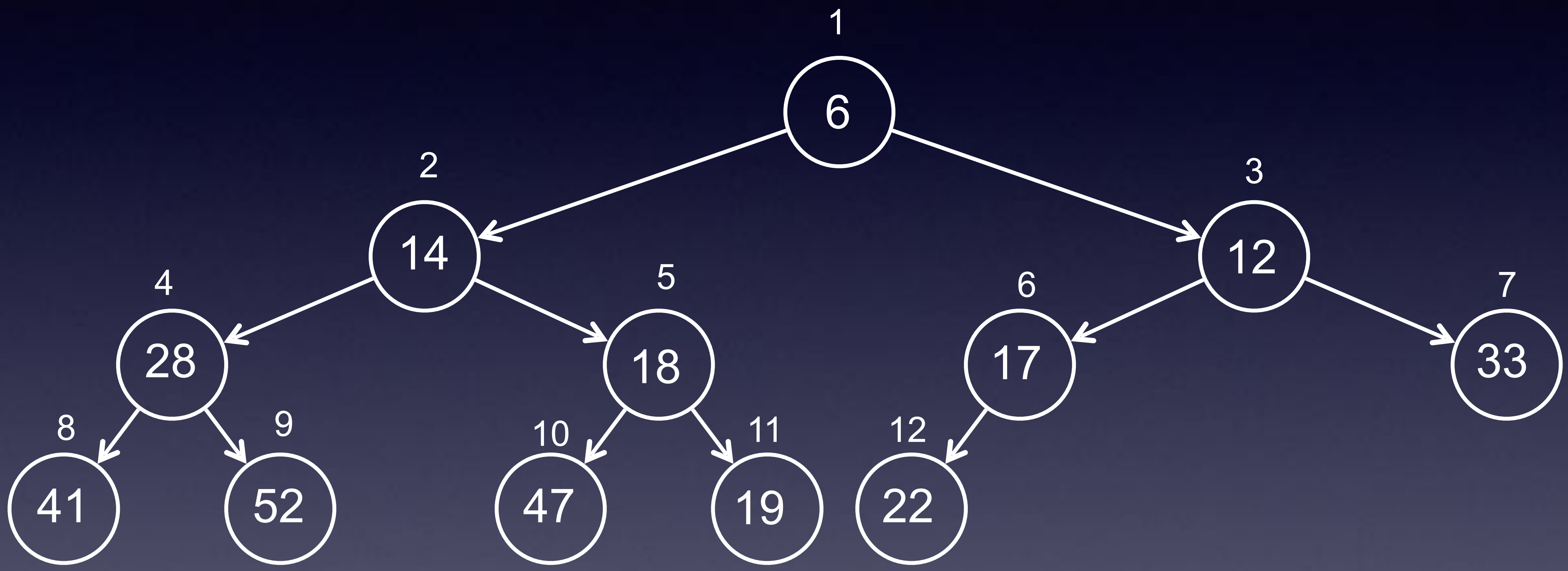


Notice that a node at index k,

The **left child** of k is at index 2k
The **right child** of k at index 2k + 1
The **parent** of k at index k // 2

Implementing a heap

If we number the nodes in a complete binary tree by level, and left to right within a level, we get:



While other types of trees are implemented as linked list structures, a **complete binary tree can be** represented in contiguous storage (e.g. a Python list) using the numbering or indexing described above ...

Implementing a heap

0	6	14	12	28	18	17	33	41	52	47	19	22			
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15

Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: 2k

we can find the right child of k at index: 2k + 1

we can find the parent of k at index: k // 2

(Note: in this implementation, the 0th item in the list is never used. It's just there to make the arithmetic simpler. The books convention is to set it to 0.)