CPSC 221

Basic Algorithms and Data Structures

June 5, 2015

Administrative stuff

Exam marking mostly done.

Theory assignment 2 coming out this weekend.

Binary search trees

Binary trees are cool, but what we are really interested in is a class of binary trees called binary search trees.

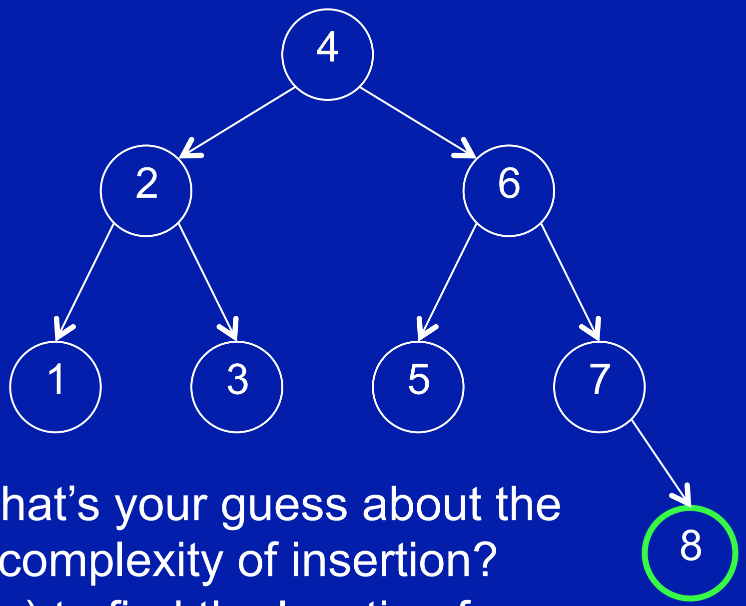
A binary search tree is a binary tree in which every node is

- empty or
- the root of a binary tree in which all the values in the left subtree are less than the value at the root, and all the values in the right subtree are greater than the value at the root.

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

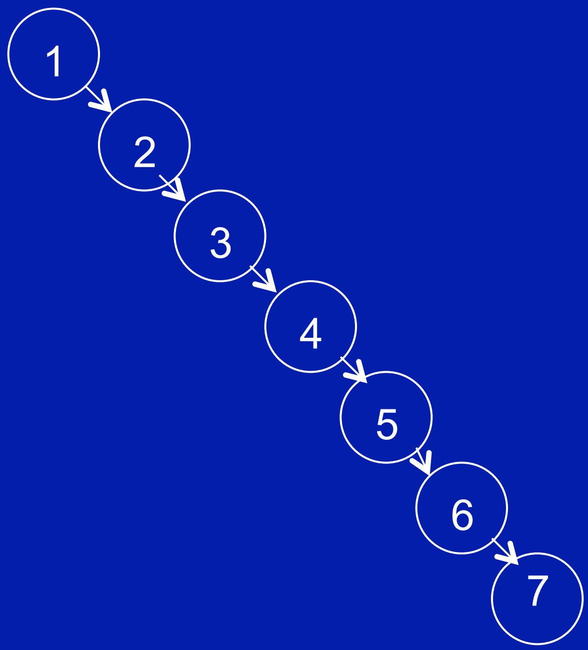


So what's your guess about the time complexity of insertion? O(lg n) to find the location for insertion plus some fixed time to create the new node and add the link.

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```



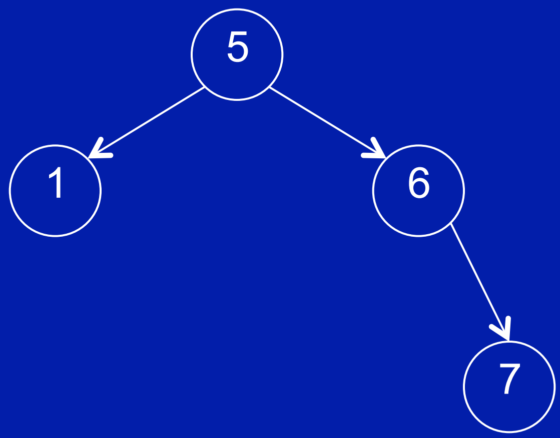
Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert
1 2 3 4 5 6 7 in that order?
Do we have an issue here?

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```



Can we also use the smallest value in the right subtree? Sure.
Delete 4.
What if 5 had children? Then the effects would ripple down from there. More on that next time.

Tree terminology

There is some vocabulary associated with trees that you should know. Some of it will be used here in class, but anything that we don't use here could easily find its way into other computer science courses.

Tree terminology

To keep things simple, we have been talking about binary trees as if a node contains only a value and two pointers to its subtrees.

In reality, a node will have a key (usually unique) to identify the associated value(s) or data contained in the node (and, of course, the pointers to the subtrees). It is the key that is used in searching. This may come up in future examples.

For example:

Student number:	98765432	← key
Student name:	Bubba Hackmeister	
Year:	3	← data/values
Program:	BCS	

Tree terminology

The path from node N_1 to node N_k is a sequence of nodes $N_1, N_2, ..., N_k$ where N_i is the parent of N_{i+1}.

The length of the path is the number of edges in the path.

The depth or level of a node N is the length of the path from the root to N. The depth of the root is 0.

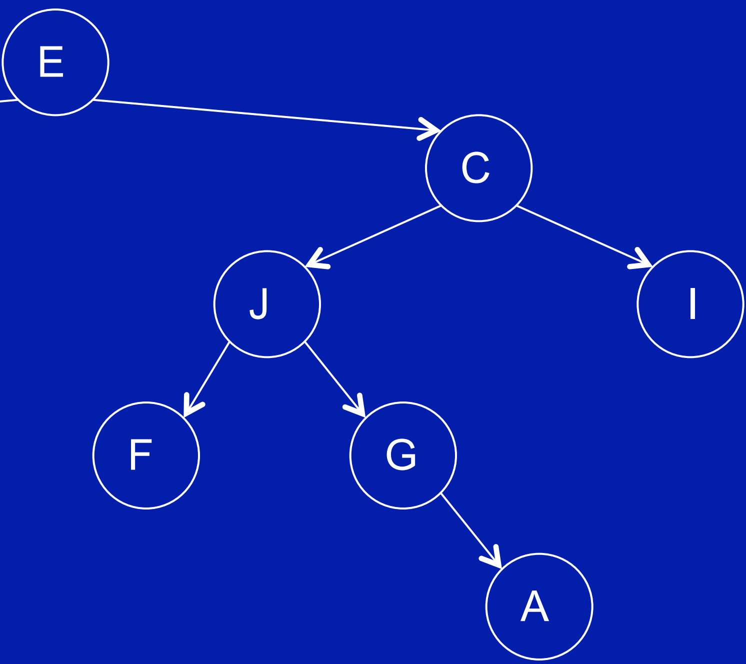
The height of a node N is the length of the longest path from N to a leaf node (a node with no child). The height of a leaf node is 0.

Tree terminology

The height of a tree is the height of its root node. The height of the empty tree is -1. The height of a tree with only a single node is 0.

The number of nodes in a binary tree of height h is at least h + 1 and no more than $2^{h+1} - 1$.

Tree terminology



Height of tree:

Depth of node containing B:

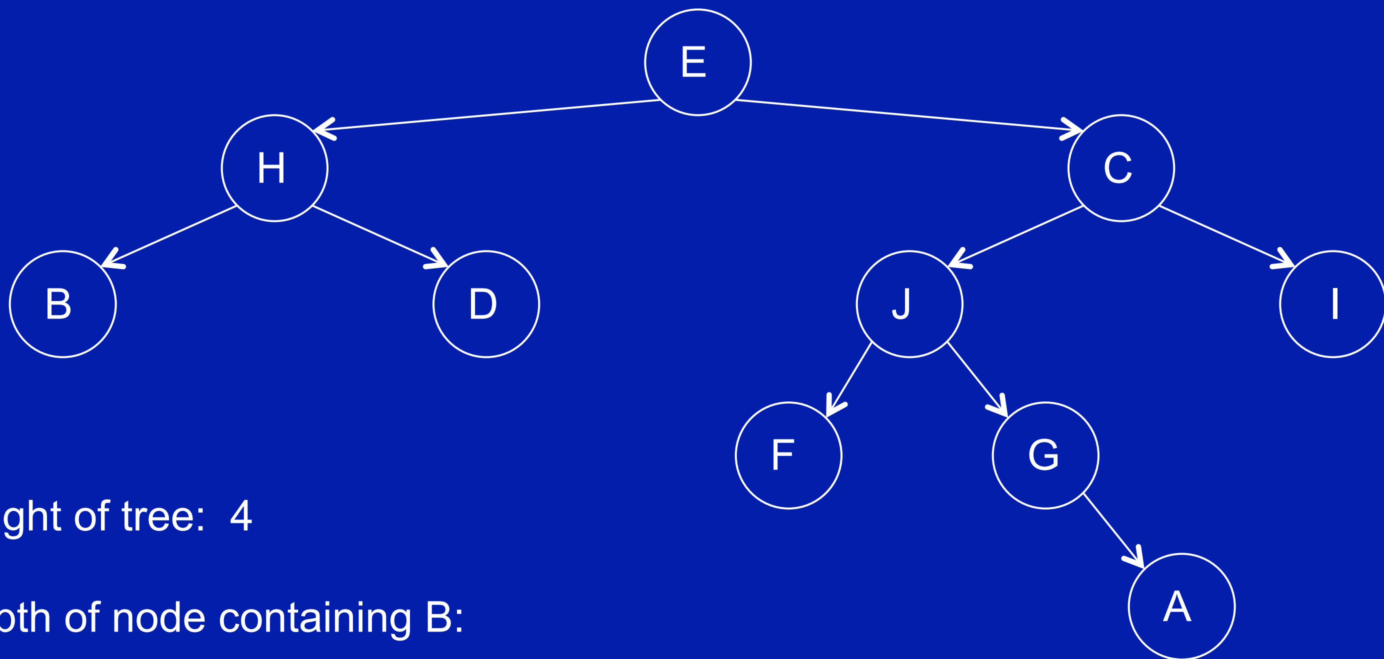
Height of node containing B:

# of nodes in this tree:

# of leaves (i.e., terminal or external nodes):

# of non-leaf (i.e., internal) nodes:

Tree terminology



Height of tree: 4

Depth of node containing B:

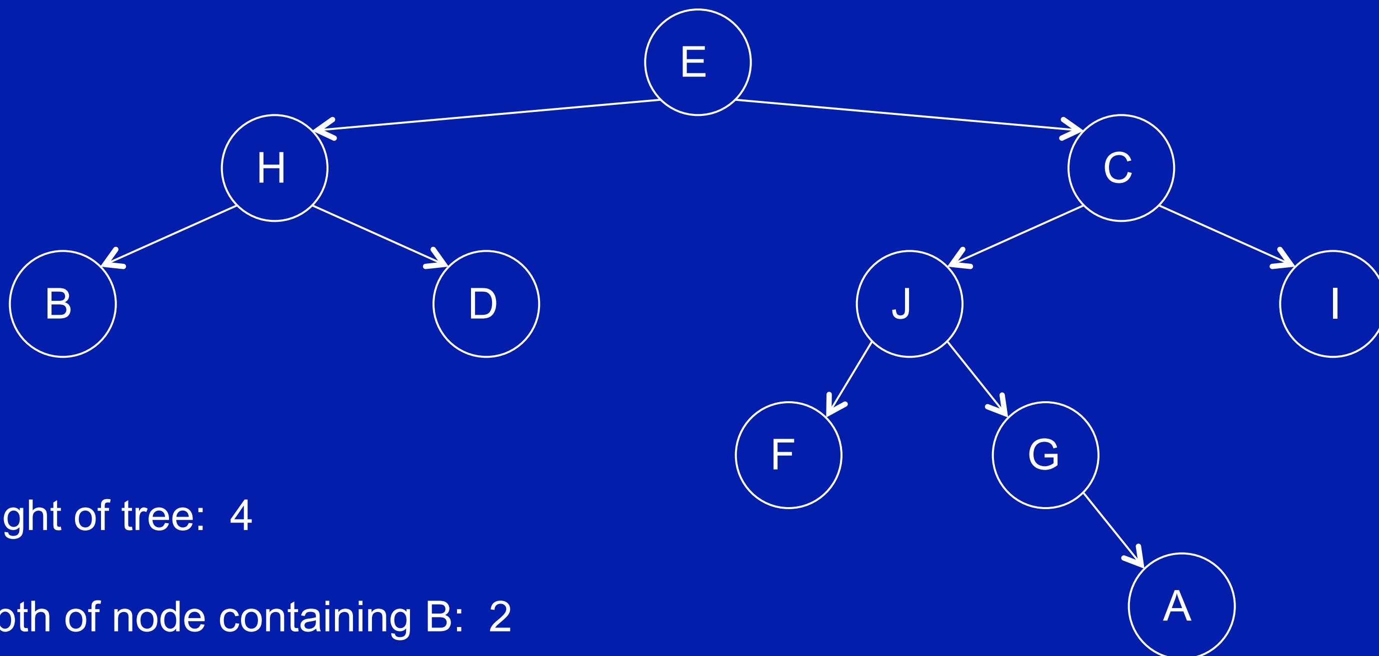
Height of node containing B:

# of nodes in this tree:

# of leaves (i.e., terminal or external nodes):

# of non-leaf (i.e., internal) nodes:

Tree terminology



Height of tree: 4

Depth of node containing B: 2

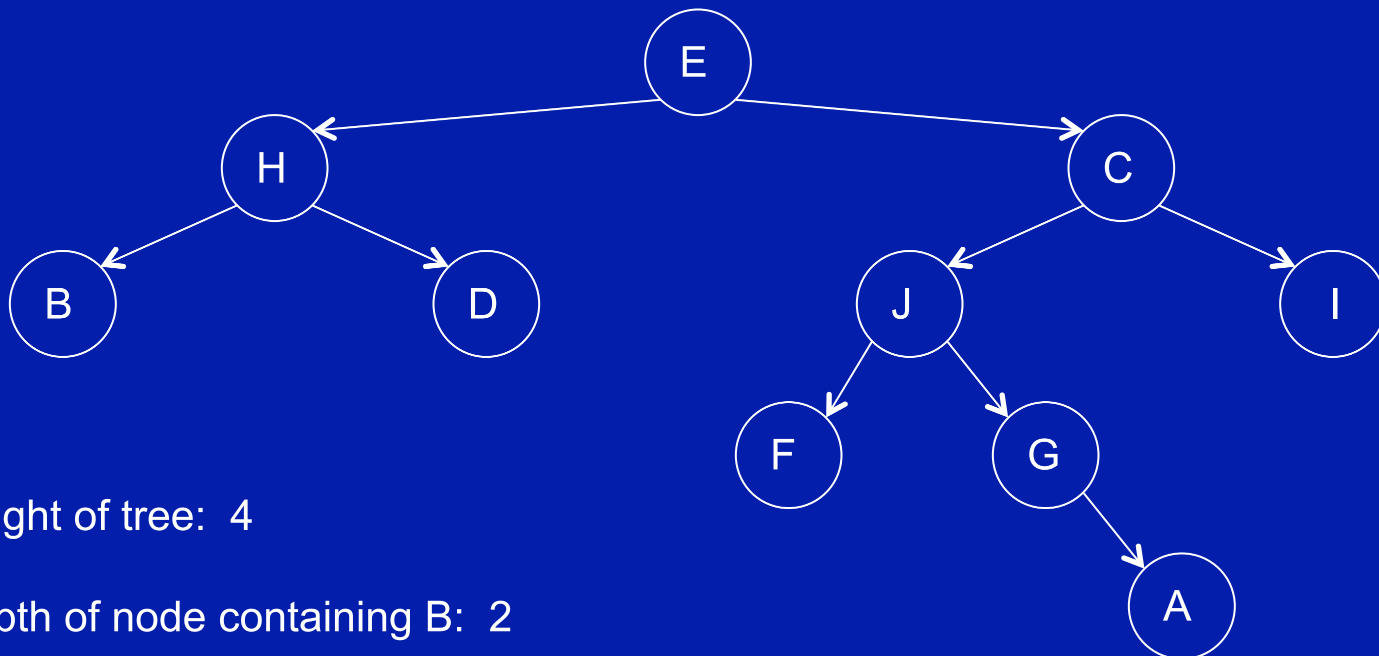
Height of node containing B:

\# of nodes in this tree:

\# of leaves (i.e., terminal or external nodes):

\# of non-leaf (i.e., internal) nodes:

Tree terminology



Height of tree: 4

Depth of node containing B: 2

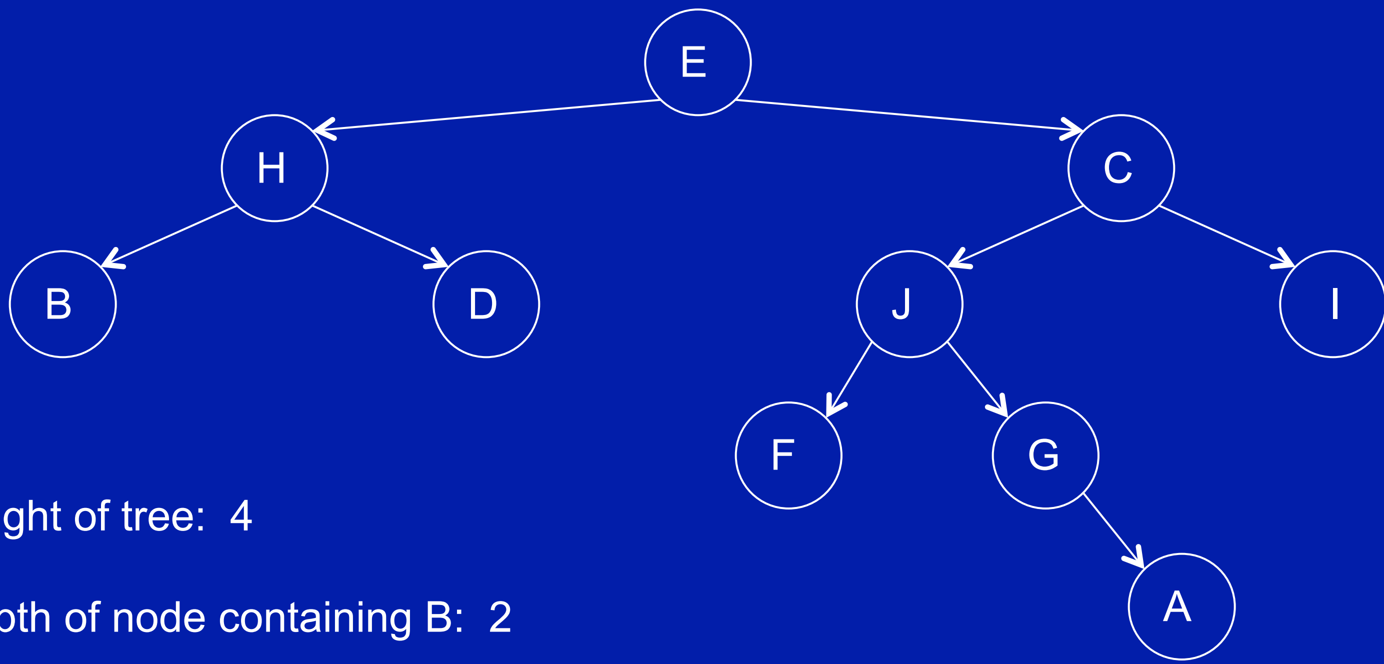
Height of node containing B: 0

# of nodes in this tree:

# of leaves (i.e., terminal or external nodes):

# of non-leaf (i.e., internal) nodes:

Tree terminology



Height of tree: 4

Depth of node containing B: 2

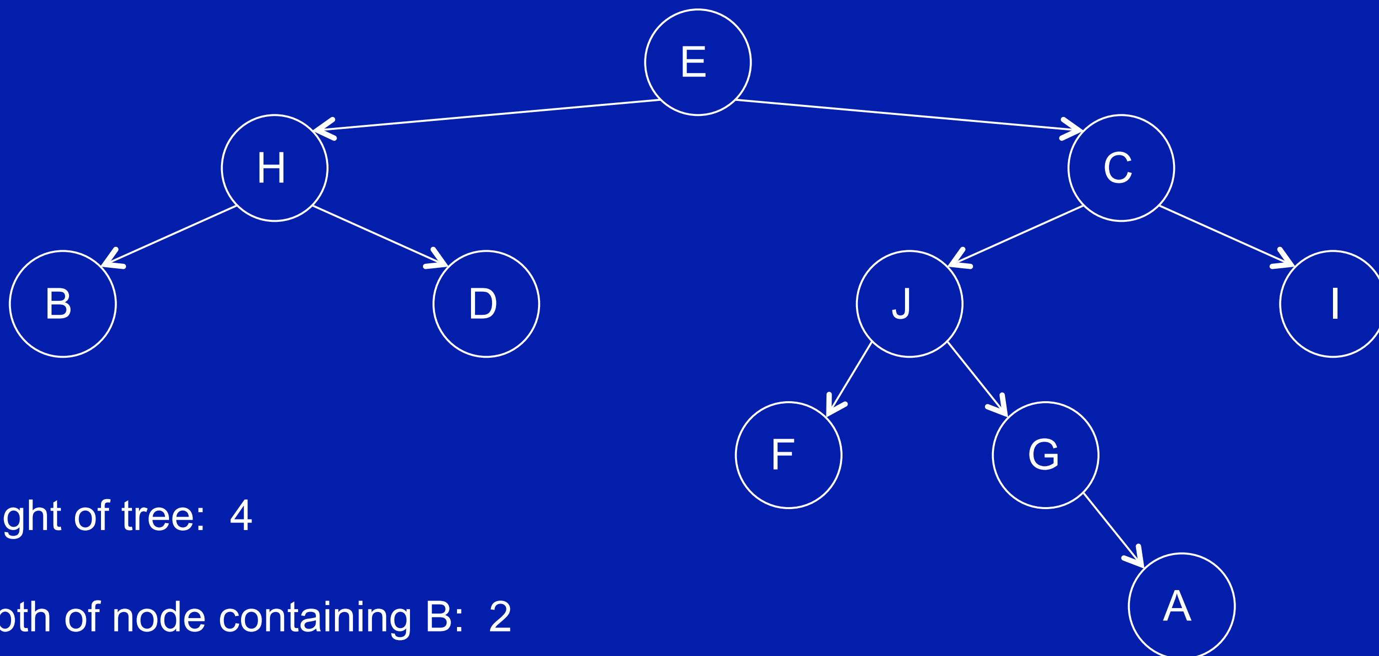
Height of node containing B: 0

# of nodes in this tree: 10

# of leaves (i.e., terminal or external nodes):

# of non-leaf (i.e., internal) nodes:

Tree terminology



Height of tree: 4

Depth of node containing B: 2

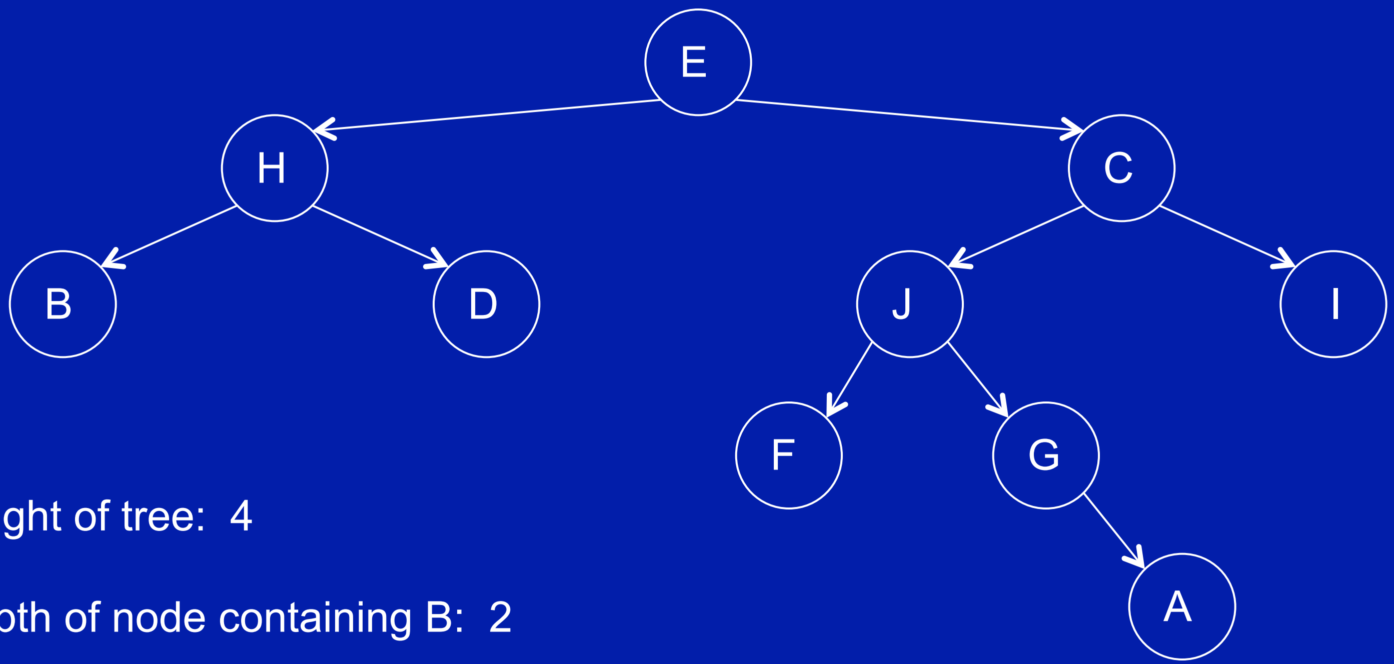
Height of node containing B: 0

# of nodes in this tree: 10

# of leaves (i.e., terminal or external nodes): 5

# of non-leaf (i.e., internal) nodes:

Tree terminology



Height of tree: 4

Depth of node containing B: 2

Height of node containing B: 0

# of nodes in this tree: 10

# of leaves (i.e., terminal or external nodes): 5

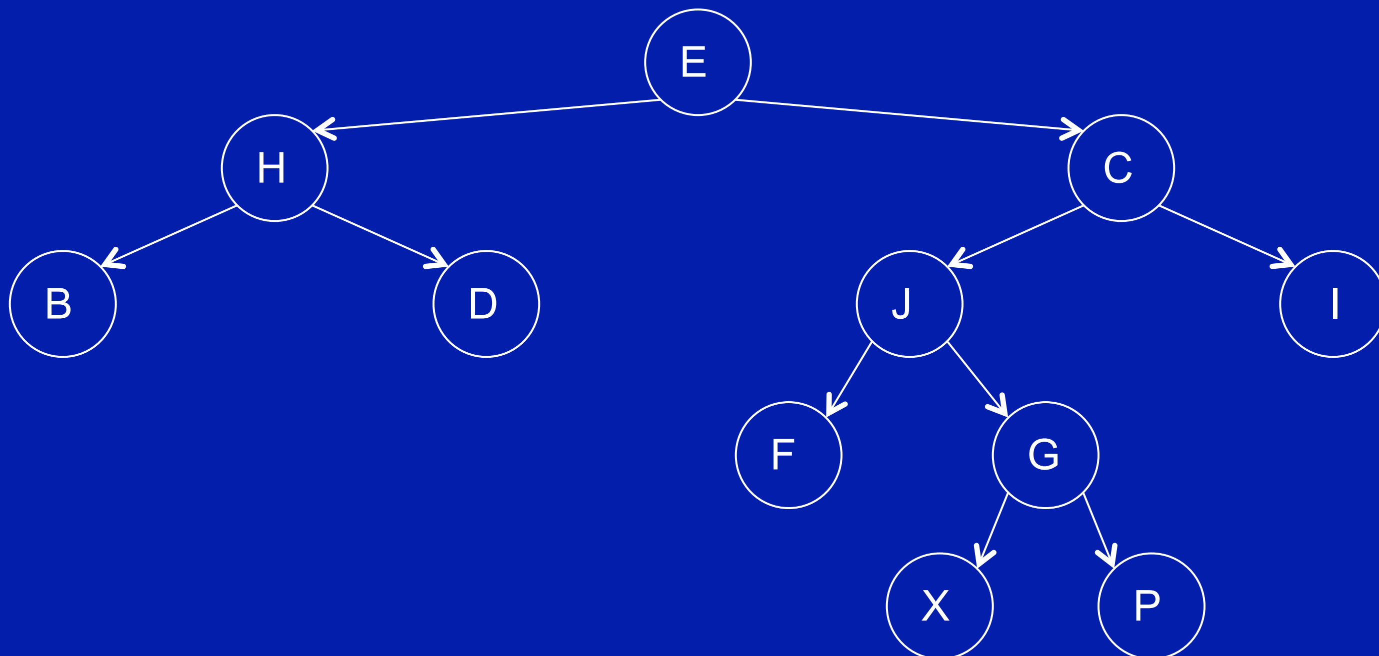
# of non-leaf (i.e., internal) nodes: 5

Tree terminology

A full binary tree is a binary tree in which each node has exactly 0 or 2 children. Each internal node has exactly 2 children, and each leaf has 0 children.

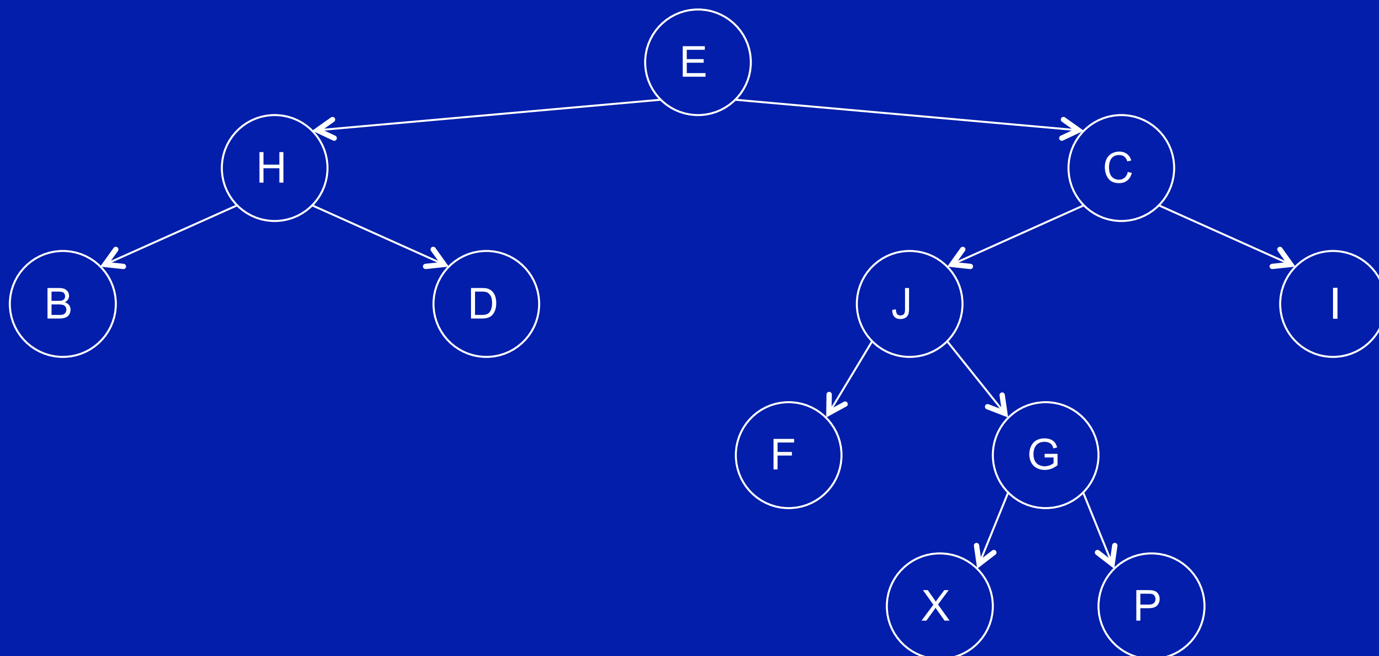
A complete binary tree is a binary tree of height h in which leaves appear only at two adjacent levels, depth h – 1 and depth h, and the leaves at the very bottom (depth h) are in the leftmost positions.

Tree terminology



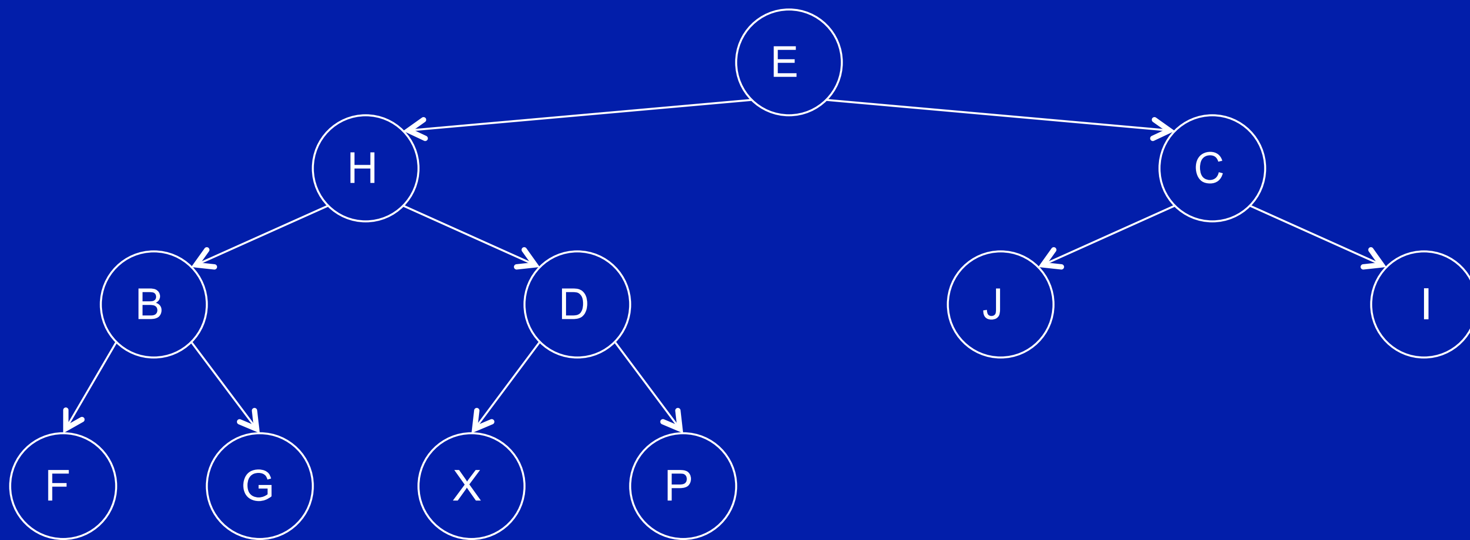
Full?
Complete?

Tree terminology



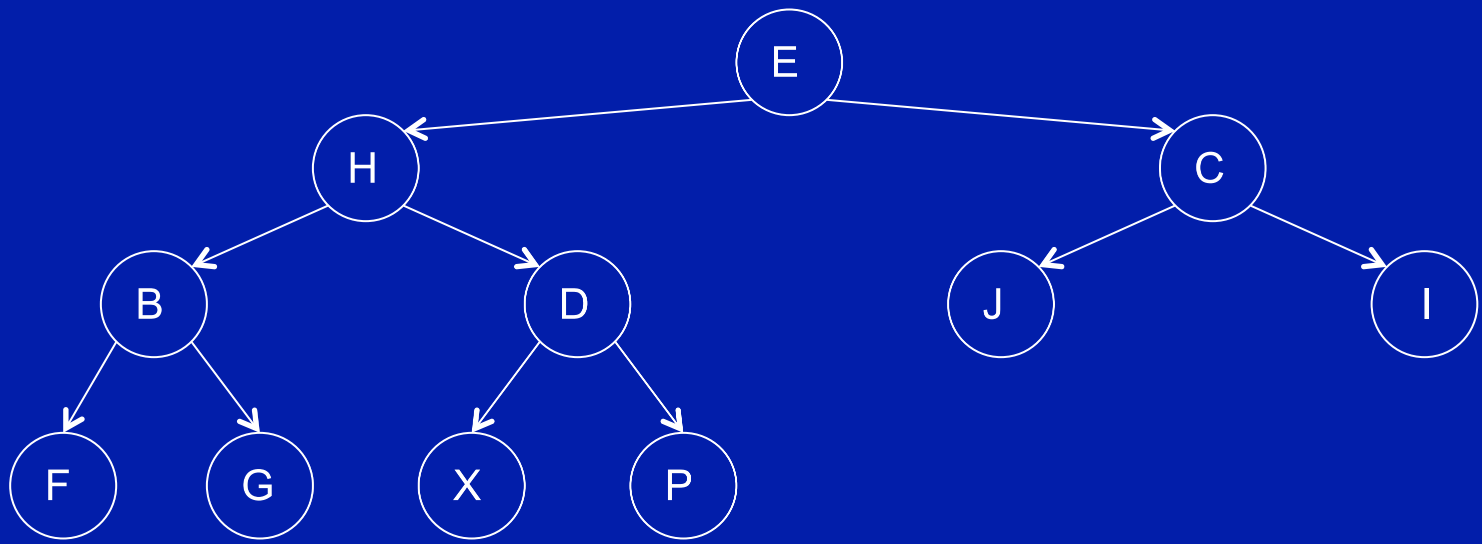
Full? Yes
Complete? No

Tree terminology



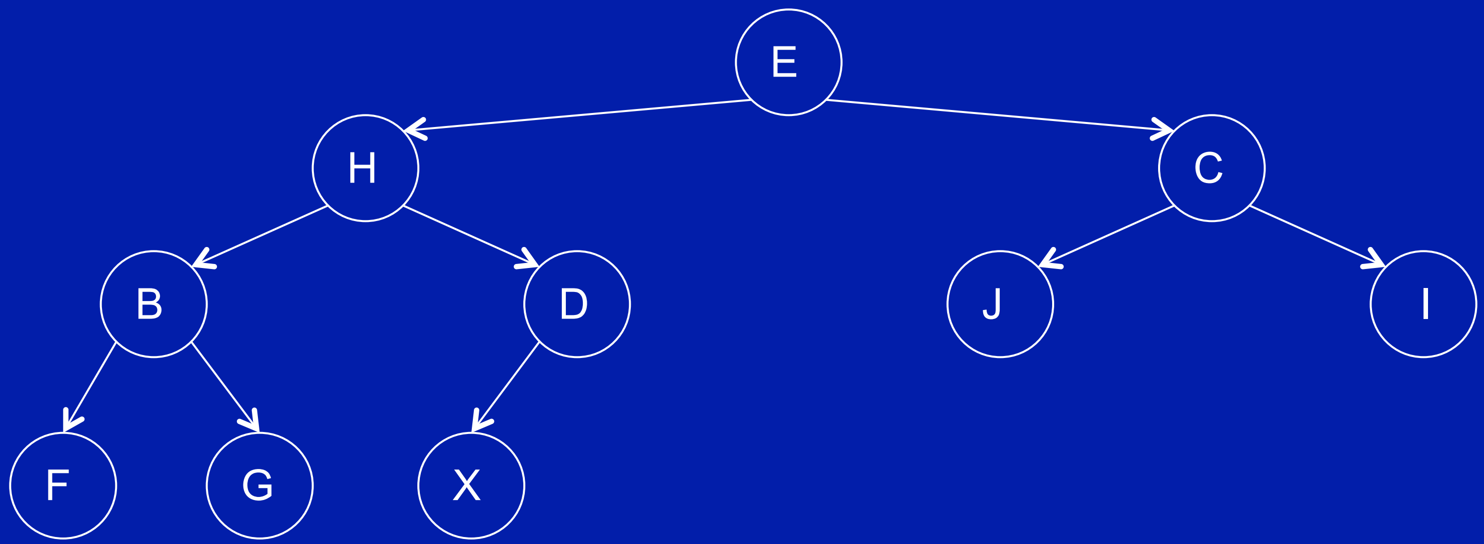
Full?
Complete?

Tree terminology



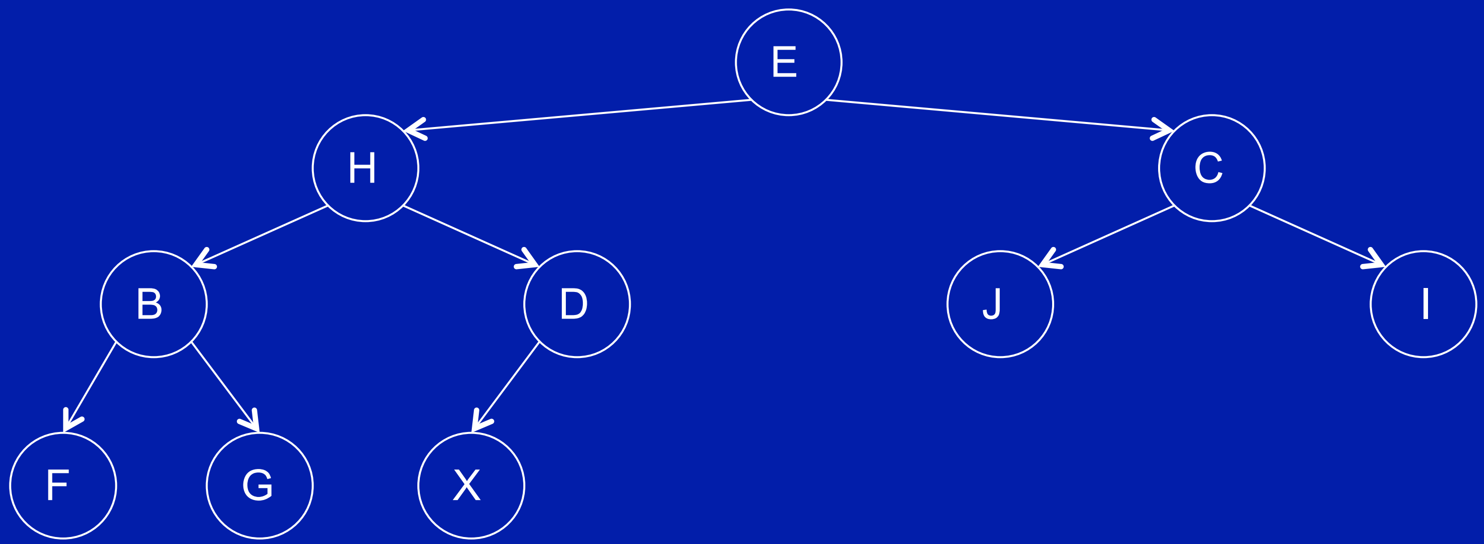
Full? Yes
Complete? Yes

Tree terminology



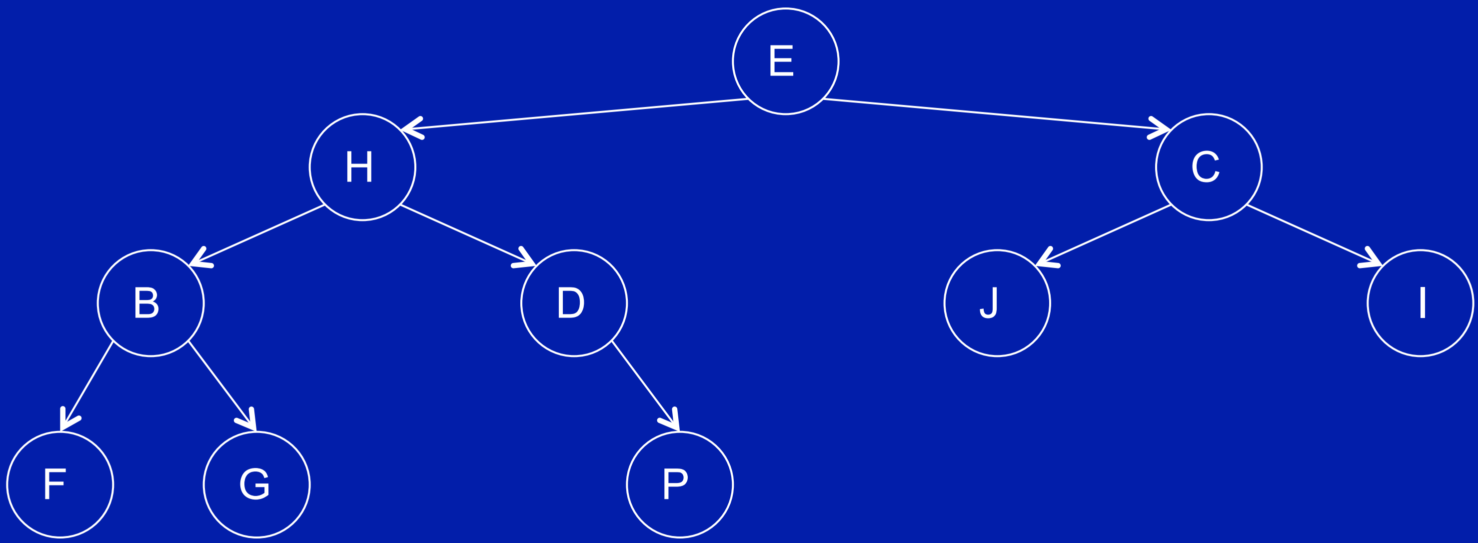
Full?
Complete?

Tree terminology



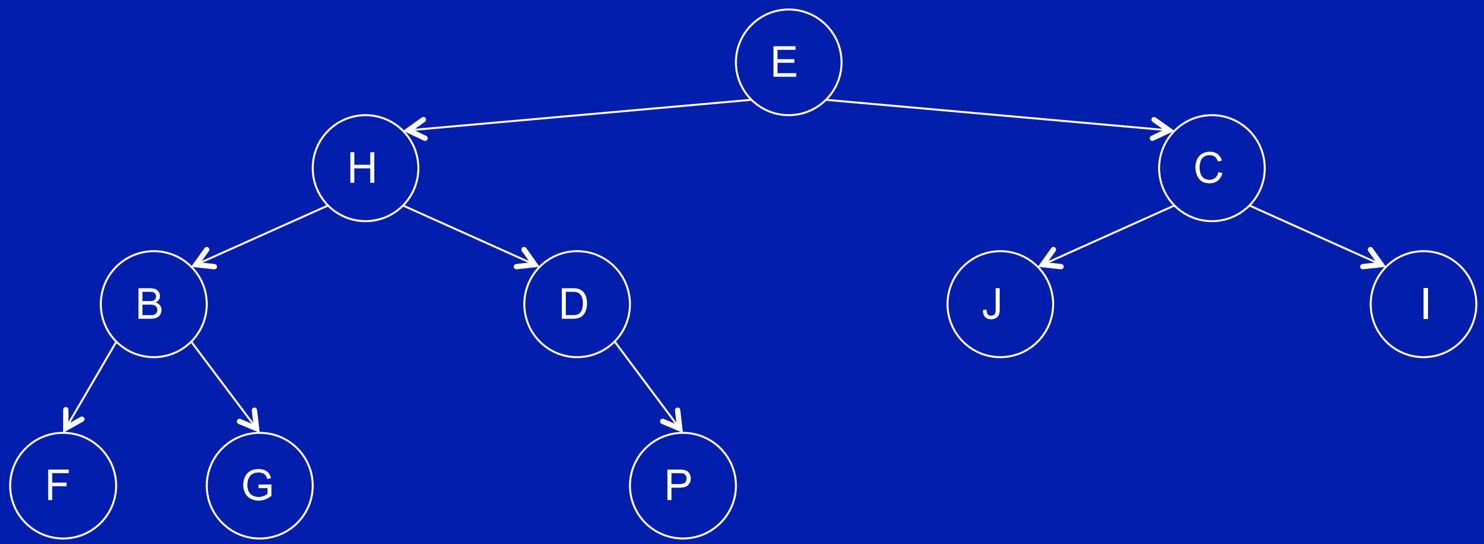
Full? No
Complete? Yes

Tree terminology



Full?
Complete?

Tree terminology



Full? No
Complete? No

Attributes of a “good” BST

BSTs make it possible to manage lots of data while providing fast search, fast insertion, and fast deletion. If the BST has “good” shape, these operations have O(log n) time complexity.

But the height of the tree has to be small relative to the number of nodes (= n) in the tree. Operations are speedy as long as the tree is “shallow”. (e.g., complete binary search tree, or one which isn’t necessarily complete but the “fringe” is only at the bottom level)

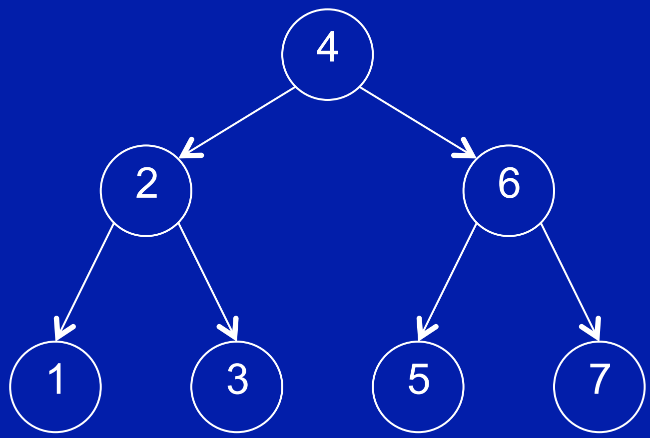
Attributes of a “good” BST

However, insertions and deletions can make some branches long and others short. The tree is no longer “shallow” and operations require more time to finish.

This is a problem.

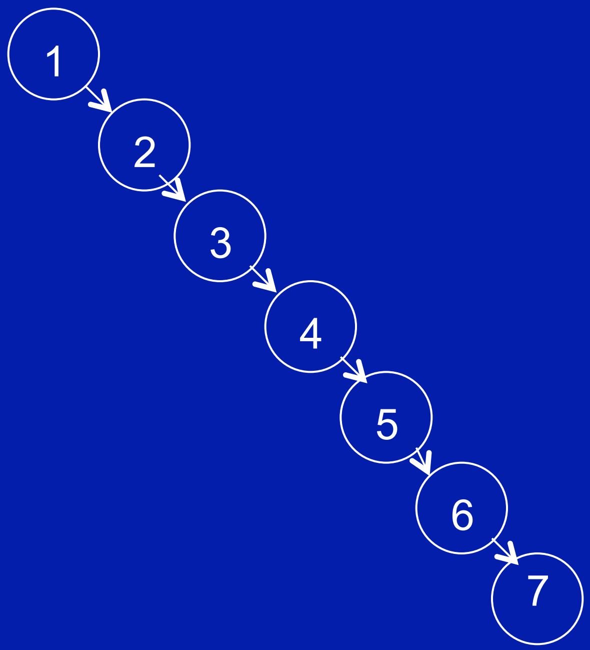
Attributes of a “good” BST

This is good...



Attributes of a "good" BST

...but this is bad.



Is there a solution?

Of course there is.

What’s needed is a way to manage the shape/height of the tree as you add things to the tree or take things away from the tree.

What’s needed is balance.

AVL trees

In 1962, Russian mathematicians Georgy Adelson-Velsky and Evegeny Landis published an algorithm for automatically maintaining the overall balance in a binary search tree. BSTs using this approach are called AVL trees.

AV

L

AVL trees

Their algorithm maintains a tree's balance by tracking the difference in height of each subtree. That balance is the height difference between the two subtrees.

As items are inserted in (or deleted from) the tree, the balance of each subtree is updated from the insertion (or deletion) point up to the root. If the balance ever goes outside the nominal range of -1 to 1, rotation is applied to bring the balance back to the nominal range.

AVL trees

Their algorithm maintains a tree's balance by tracking the difference in height of each subtree. That balance is the height difference between the two subtrees.

As items are inserted in (or deleted from) the tree, the balance of each subtree is updated from the insertion (or deletion) point up to the root. If the balance ever goes outside the nominal range of -1 to 1, rotation is applied to bring the balance back to the nominal range.

That gives us three ideas we need to understand if we're going to work with AVL trees:

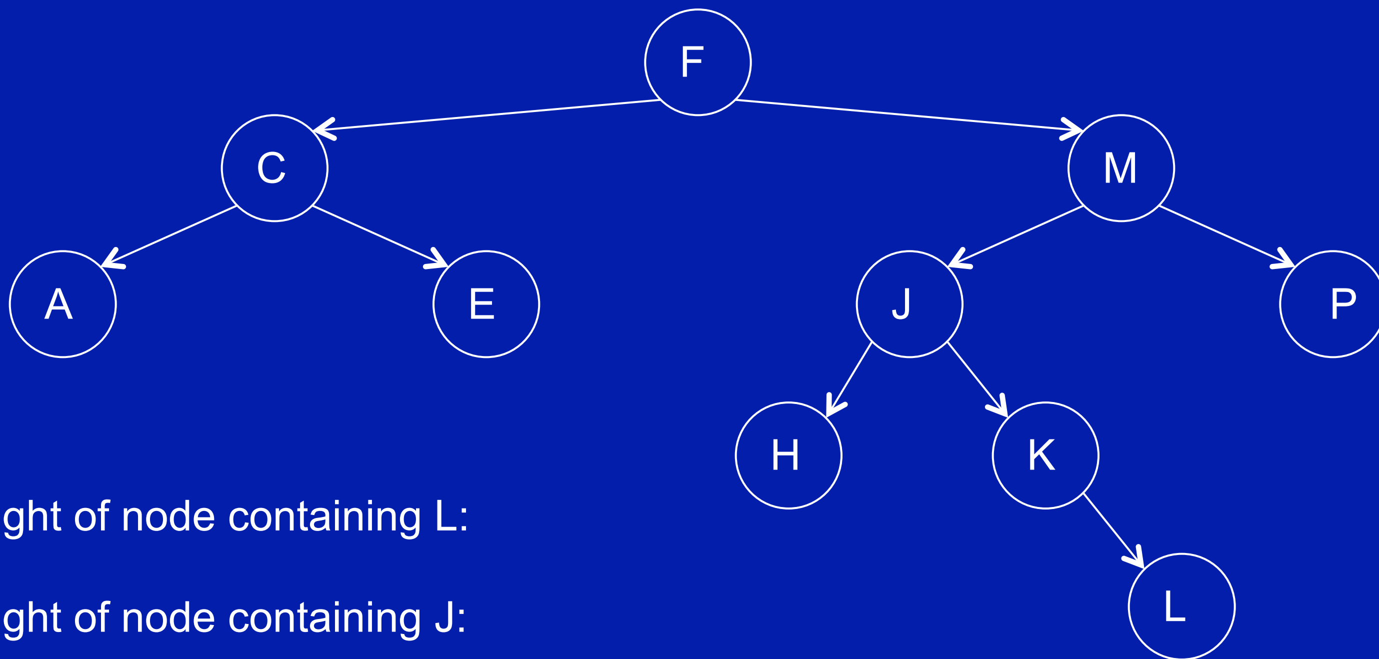
height balance rotation

Height review

The height of a node N is the length of the longest path from N to a leaf node (a node with no child). The height of a leaf node is 0.

The height of a tree is the height of its root node. The height of the empty tree is -1. The height of a tree with only a single node is 0.

Height examples



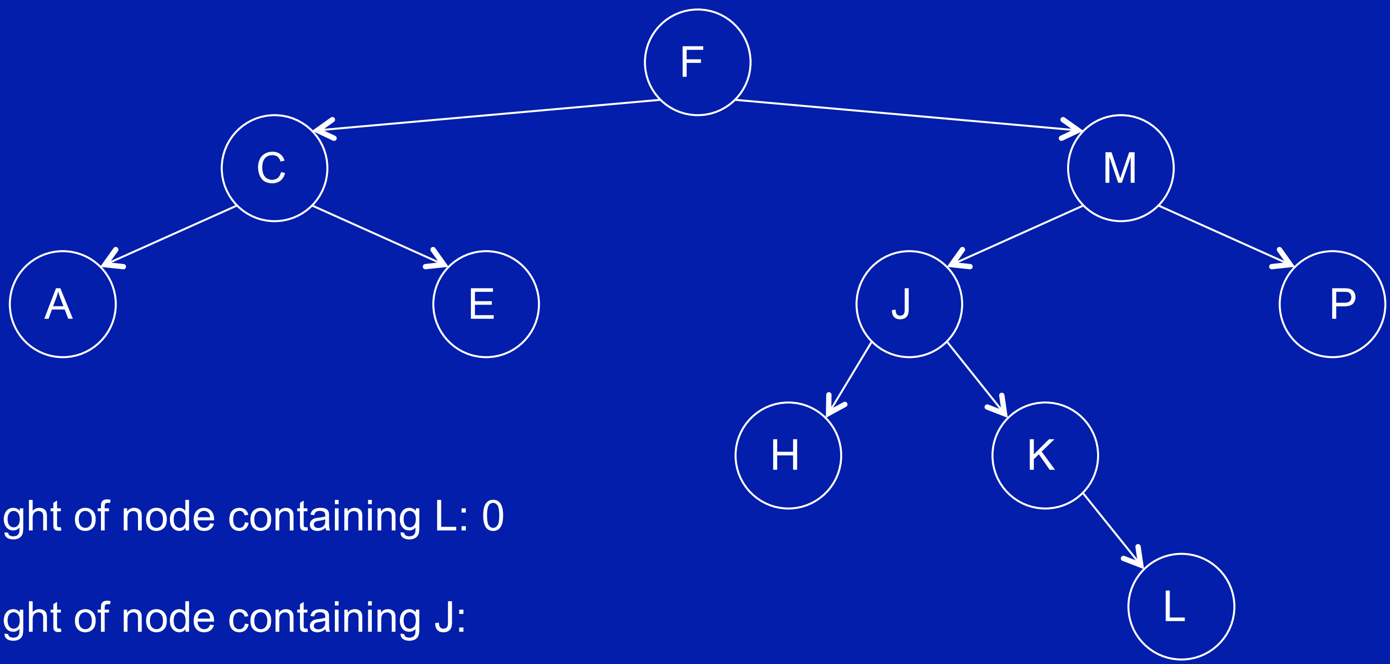
Height of node containing L:

Height of node containing J:

Height of the entire tree:

Height of the left subtree of the node containing E:

Height examples



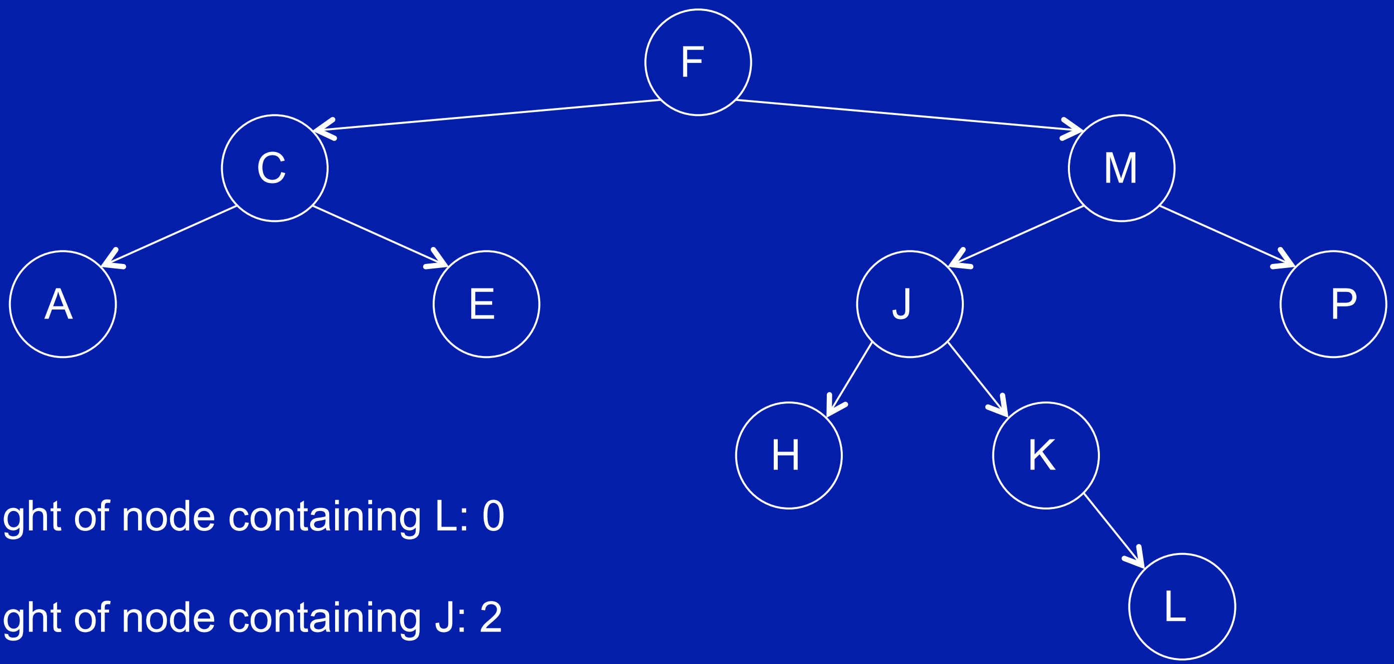
Height of node containing L: 0

Height of node containing J:

Height of the entire tree:

Height of the left subtree of the node containing E:

Height examples



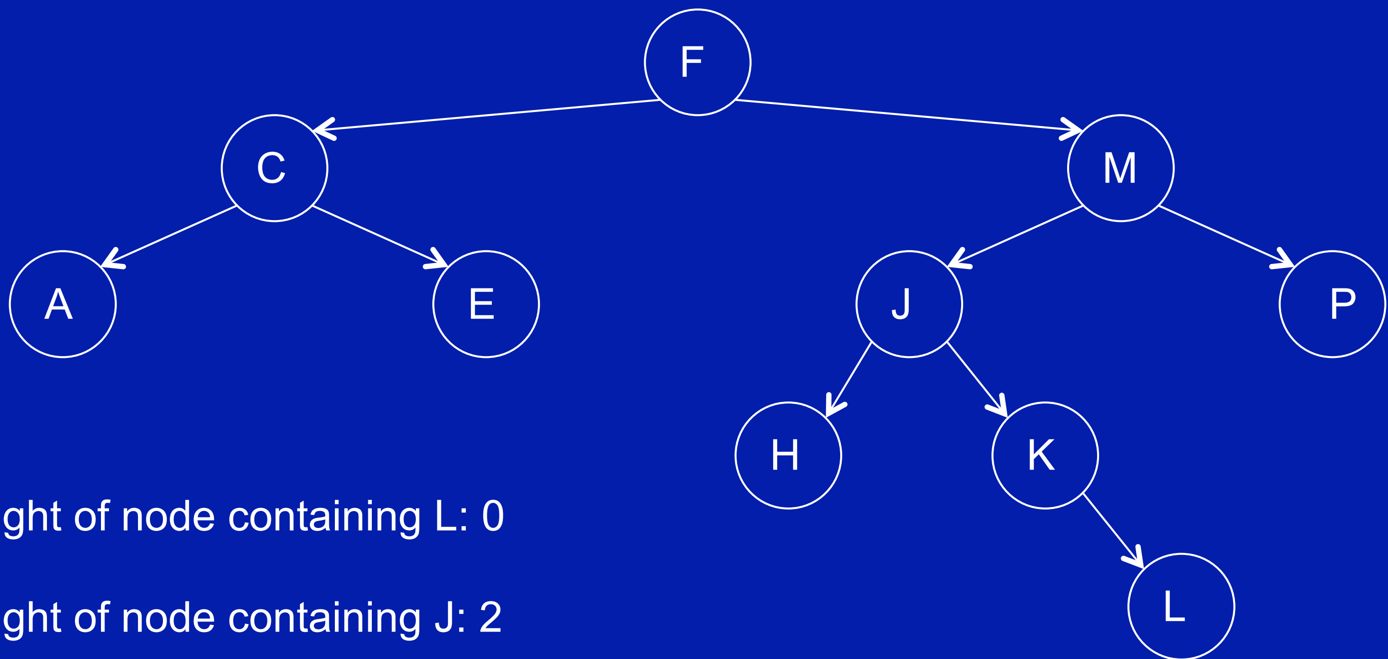
Height of node containing L: 0

Height of node containing J: 2

Height of the entire tree:

Height of the left subtree of the node containing E:

Height examples



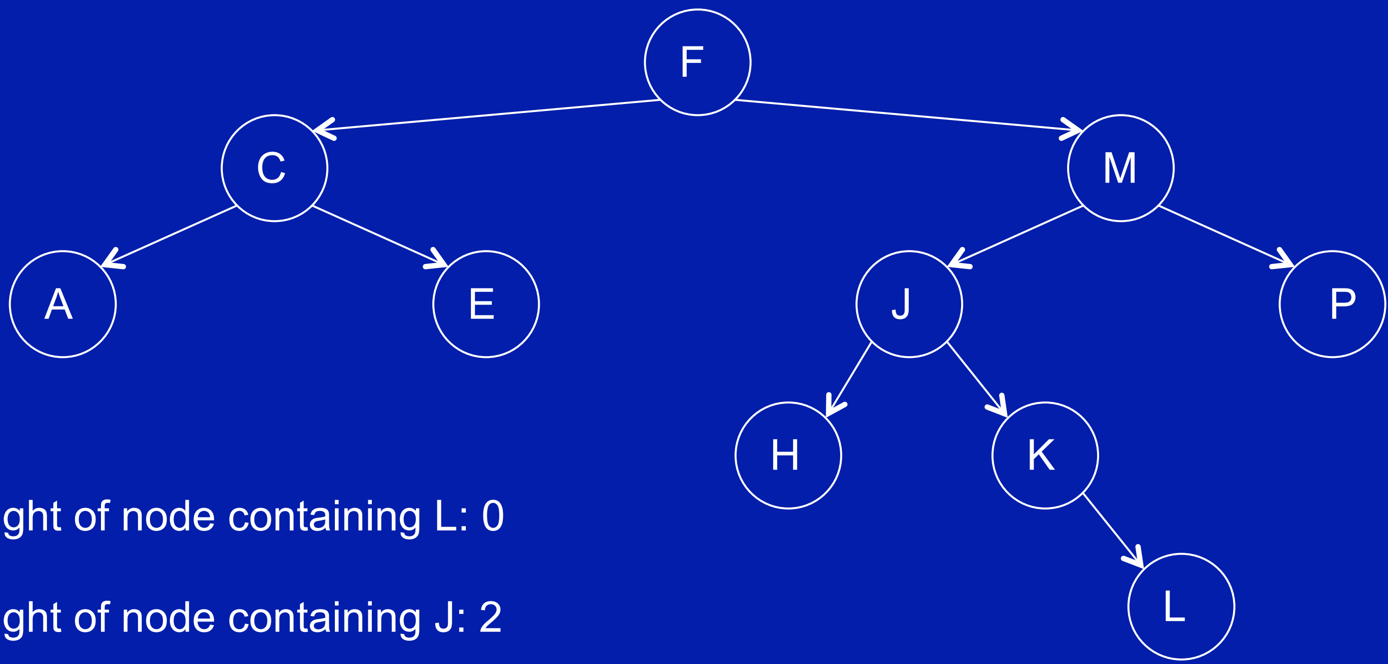
Height of node containing L: 0

Height of node containing J: 2

Height of the entire tree: 4

Height of the left subtree of the node containing E:

Height examples



Height of node containing L: 0

Height of node containing J: 2

Height of the entire tree: 4

Height of the left subtree of the node containing E: -1

Balance

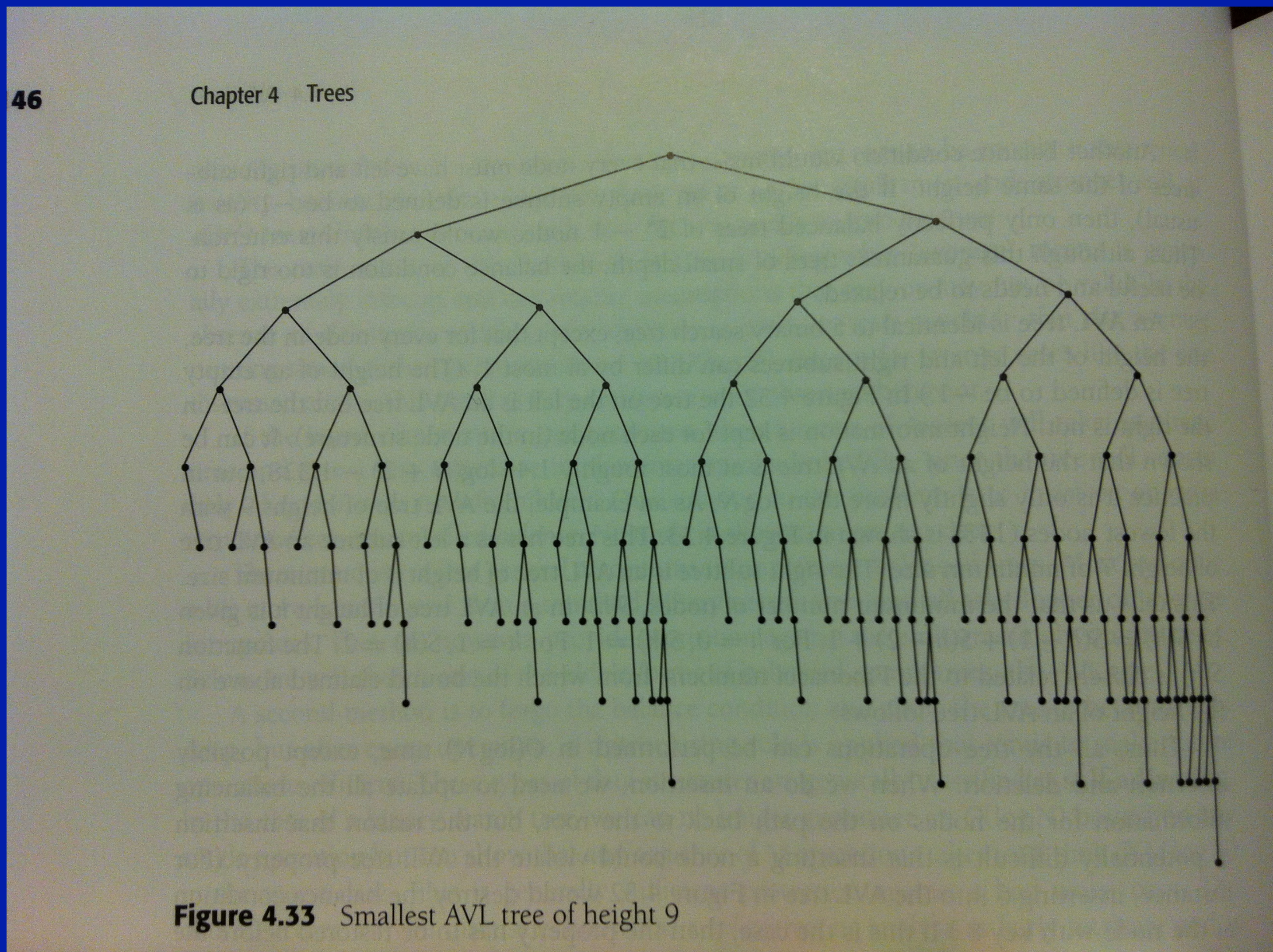
Balance for a tree at its root = height(right subtree) – height(left subtree)*

If that difference is zero everywhere, the tree is perfectly balanced.

If that difference is small everywhere, then the tree is balanced enough, even though it may not look like it…

* Balance could also be height(left subtree) – height(right subtree).
  It works either way. Change the arithmetic accordingly.

Balanced enough



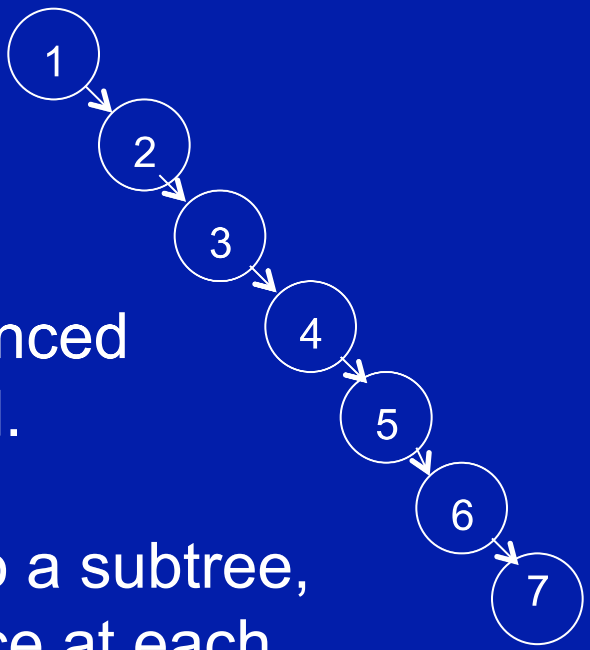
Balanced enough

If the balance is between -1 and 1 everywhere in the tree, then the maximum height of the tree is approximately 1.44 log n.* So we don’t need a perfectly balanced BST to maintain O(log n) time complexity for search.

* Let’s accept that a proof exists. Or maybe it could be a question on your final exam.

Self-balancing search trees

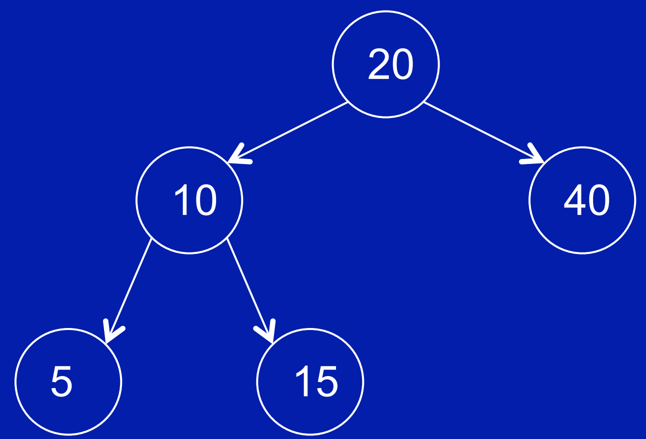
As we've seen, we can come up with a severely unbalanced BST when building the tree.



The easiest way to keep a tree balanced is never to let it become unbalanced.

So when a new node is inserted into a subtree, the AVL algorithm checks the balance at each parent node up the insertion path.

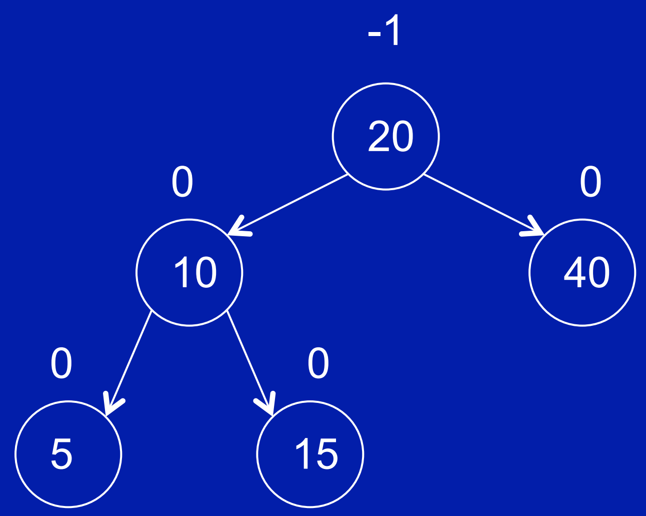
Balance



As we work with our binary search tree, we want to maintain balance to preserve our fast search times. As the balance changes from good to not so good, we need to change the relative height of the left and right subtrees that are imbalanced while preserving BST properties.

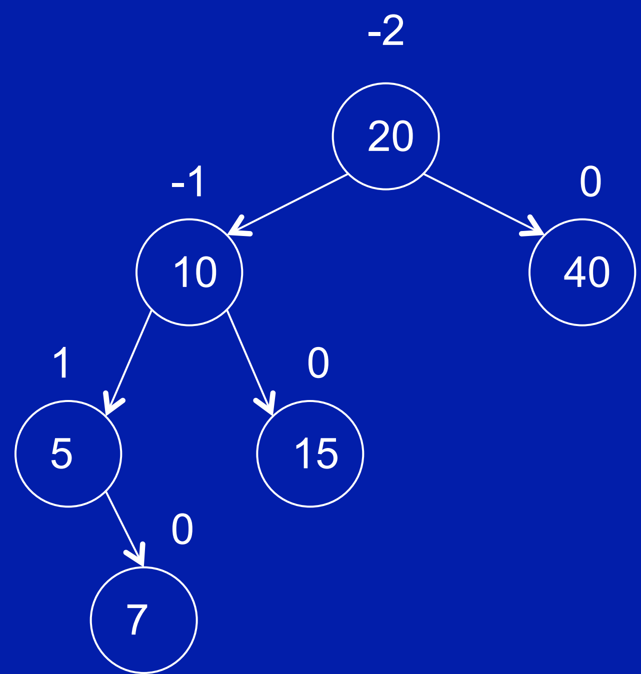
How's the balance for this tree?

Balance



We've computed the balance at the root of each subtree. These numbers are good. What happens when we add the value 7 to the tree?

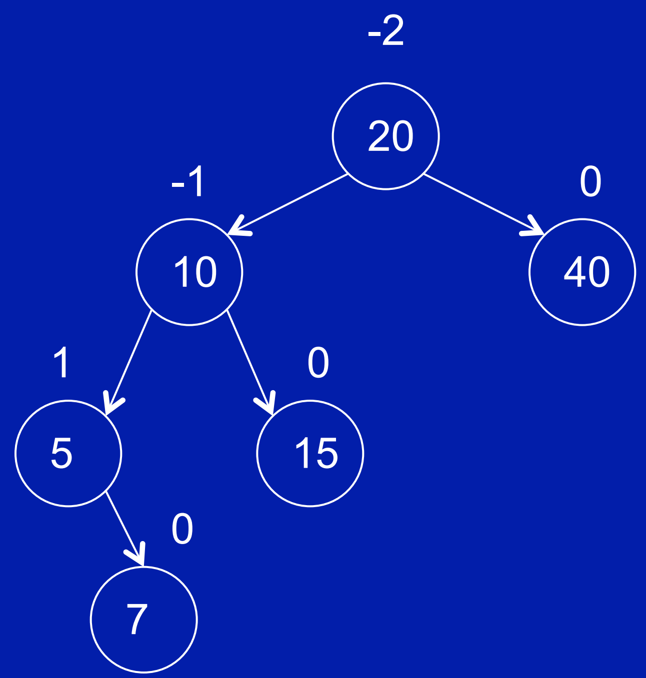
Balance



We've computed the balance at the root of each subtree. These numbers are good. What happens when we add the value 7 to the tree?

Now the balance at the root is unacceptable.

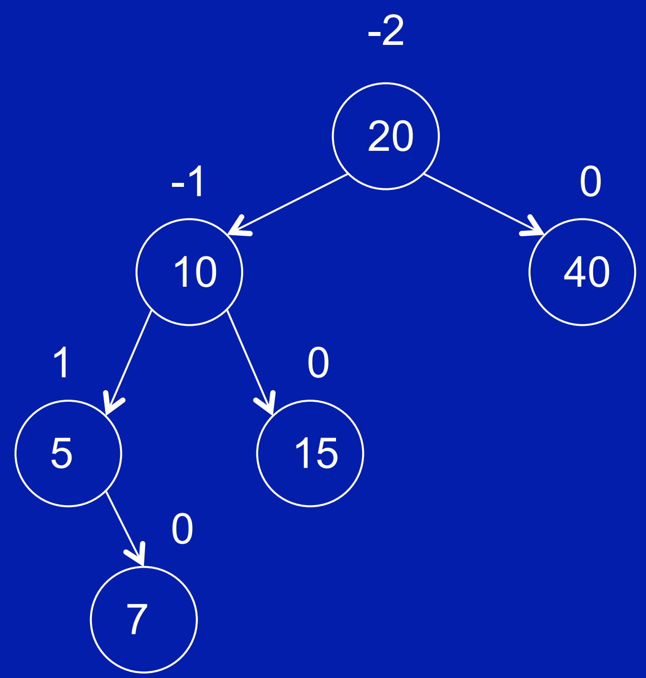
Balance



We've computed the balance at the root of each subtree. These numbers are good. What happens when we add the value 7 to the tree?

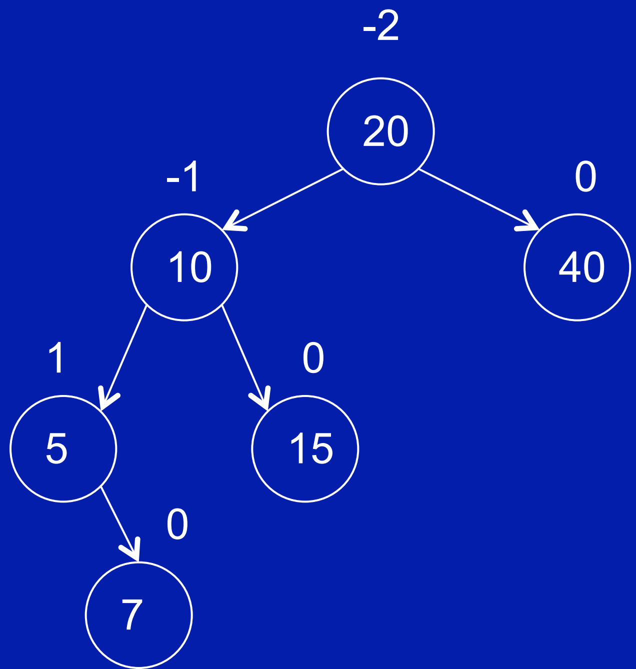
Now the balance at the root is unacceptable. The fact that the balance is negative tells us that the tree is heavy to the left, and that we should rotate the tree to the right to correct the imbalance.

Rotation



Do you have some intuition about what should happen?

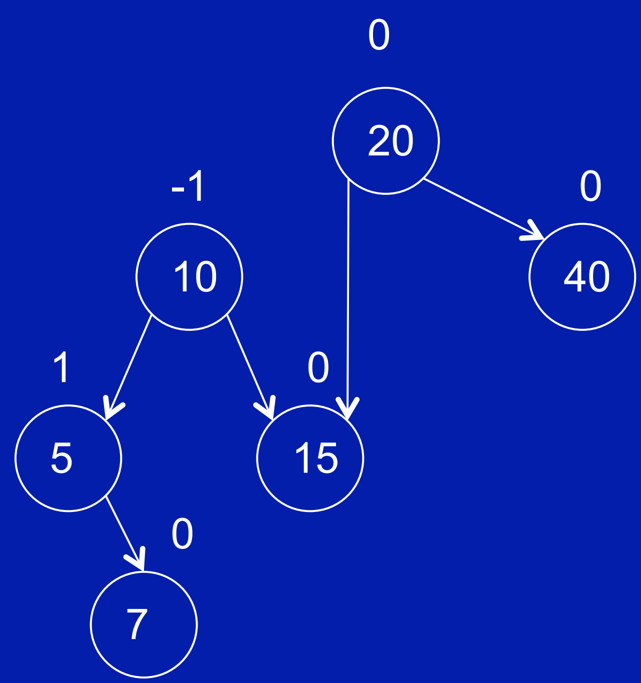
Rotation



Do you have some intuition about what should happen?

We want to raise that left subtree up, so we'll manipulate pointers accordingly…

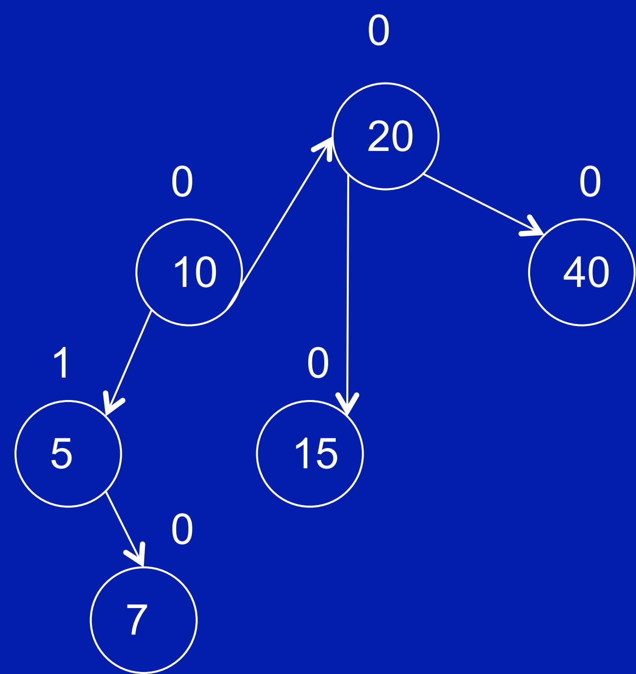
Rotation



Do you have some intuition about what should happen?

We want to raise that left subtree up, so we'll manipulate pointers accordingly…

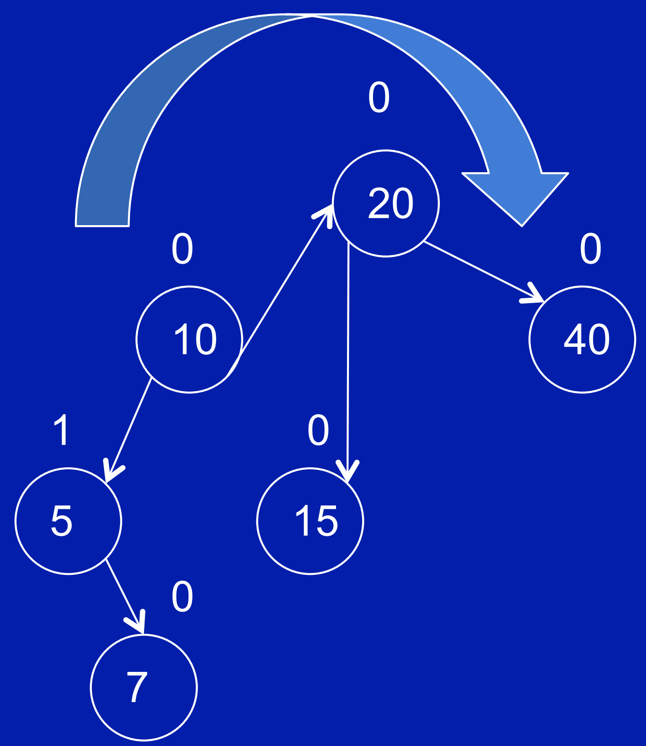
Rotation



Do you have some intuition about what should happen?

We want to raise that left subtree up, so we'll manipulate pointers accordingly…

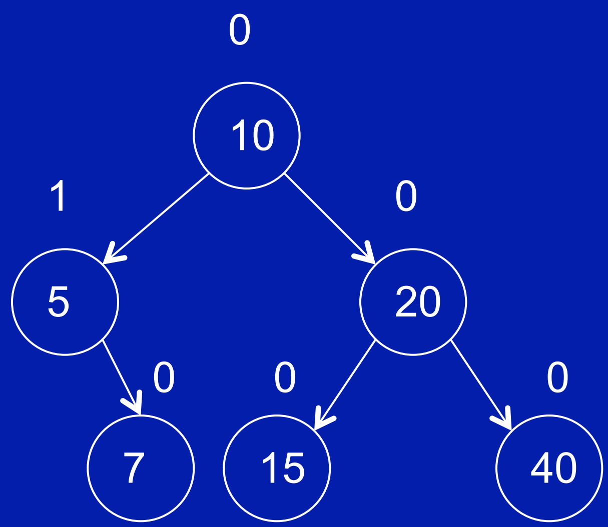
Rotation



Do you have some intuition about what should happen?

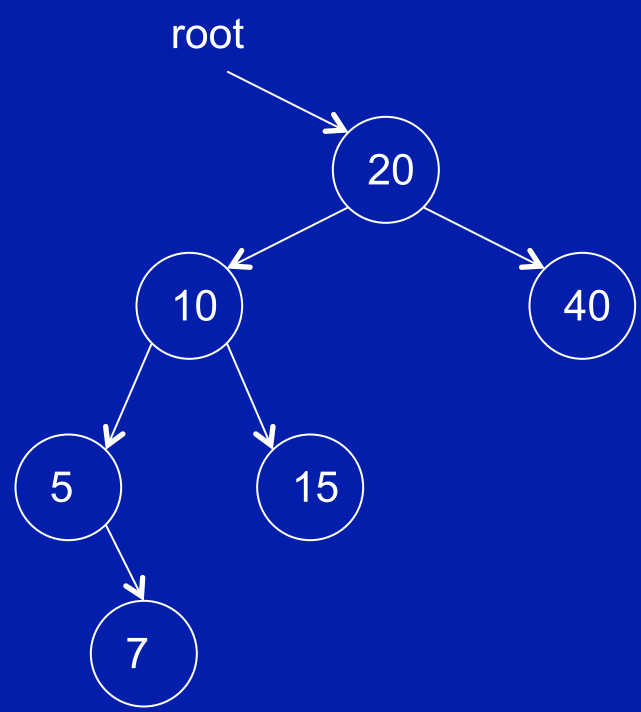
We want to raise that left subtree up, so we'll manipulate pointers accordingly…

Rotation



And now the tree is back in balance.

Rotation

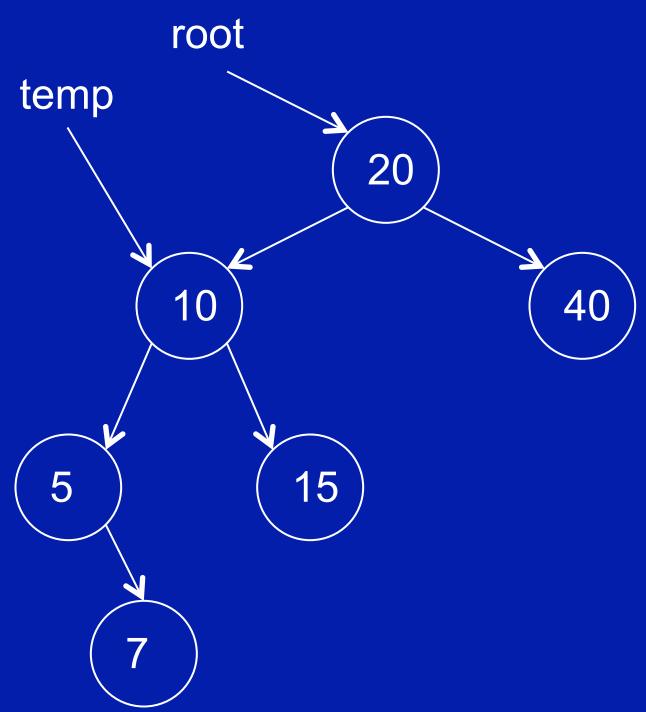


It's not magic. It's just code.

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

Rotation

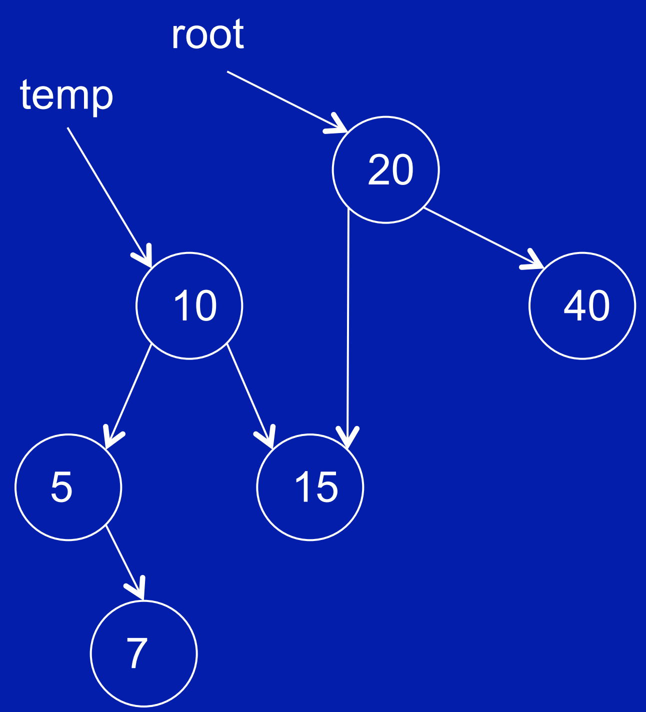


It's not magic. It's just code.

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

Rotation

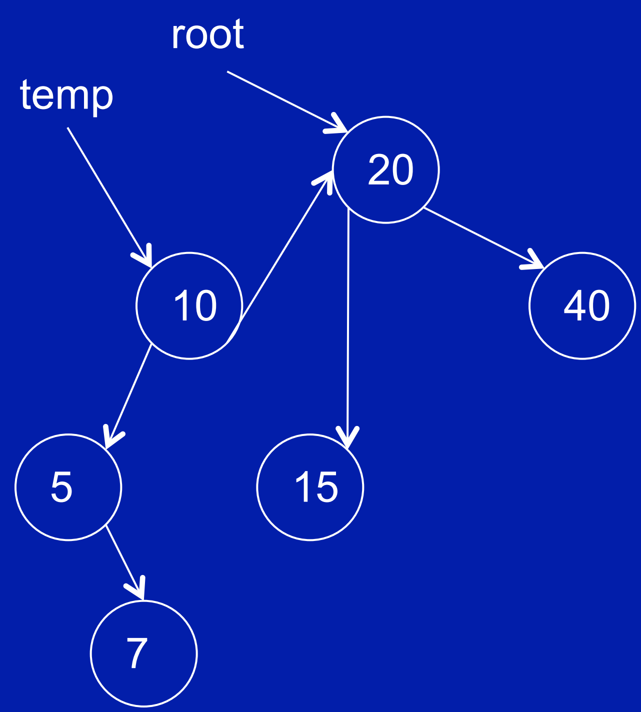


It’s not magic. It’s just code.

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

Rotation

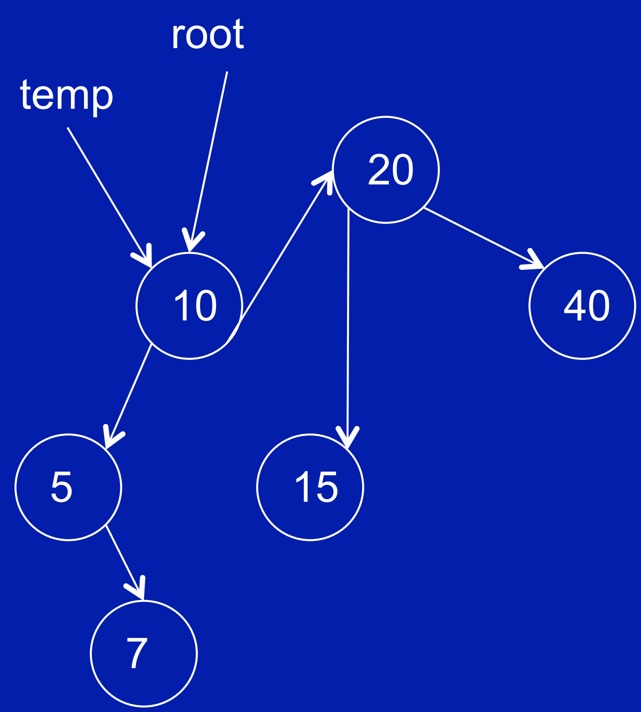


It's not magic. It's just code.

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

Rotation

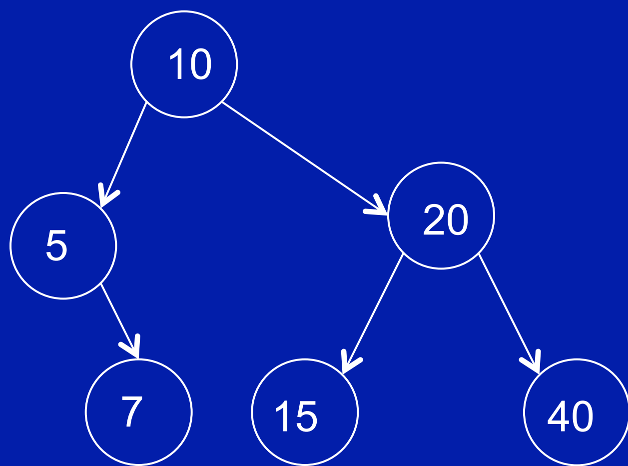


It's not magic. It's just code.

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

Rotation

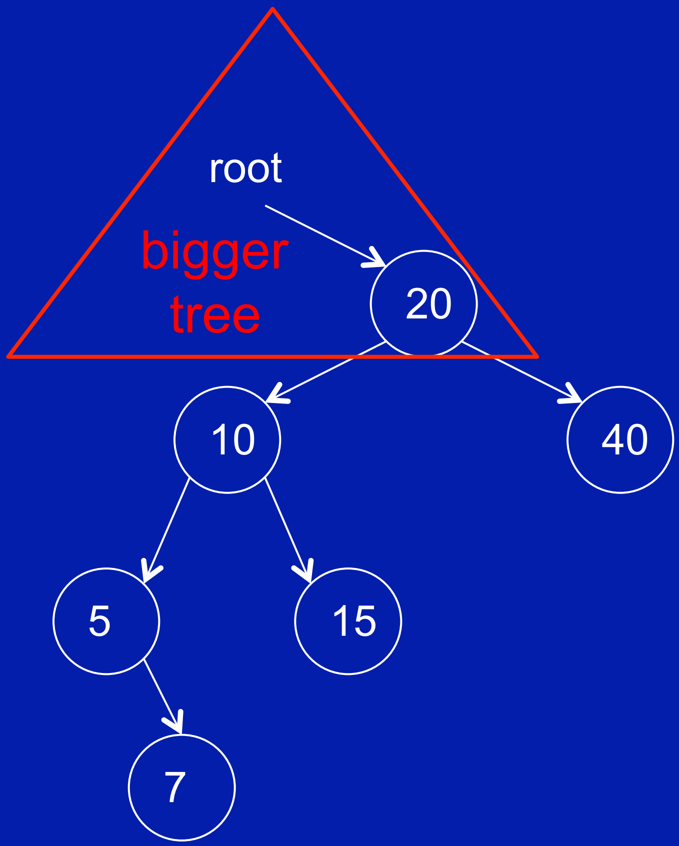


It's not magic. It's just code.

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

Rotation



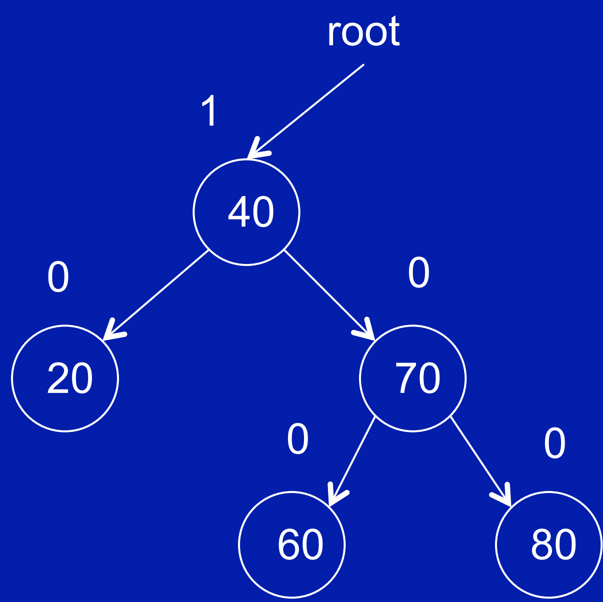
It's not magic. It's just code.

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

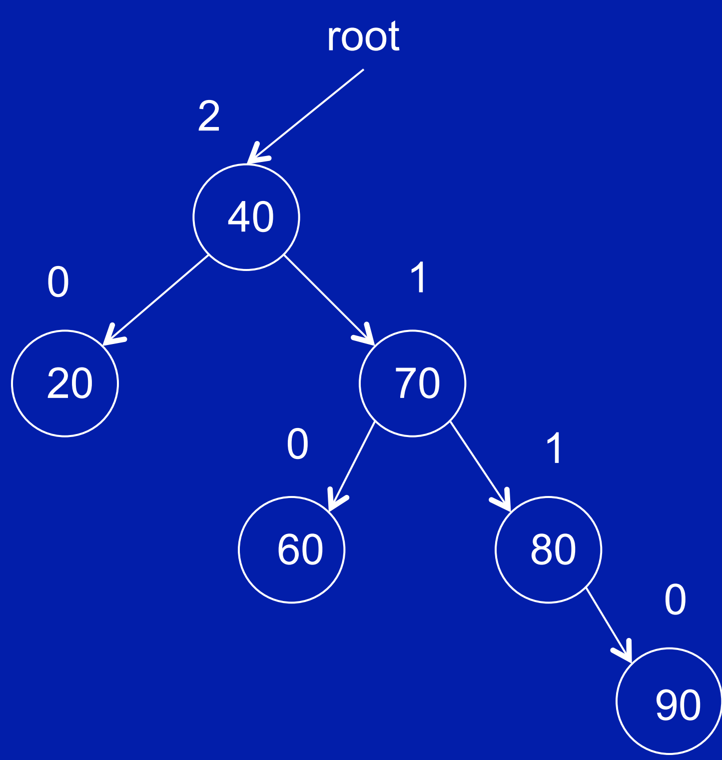
Rotation



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

Rotation

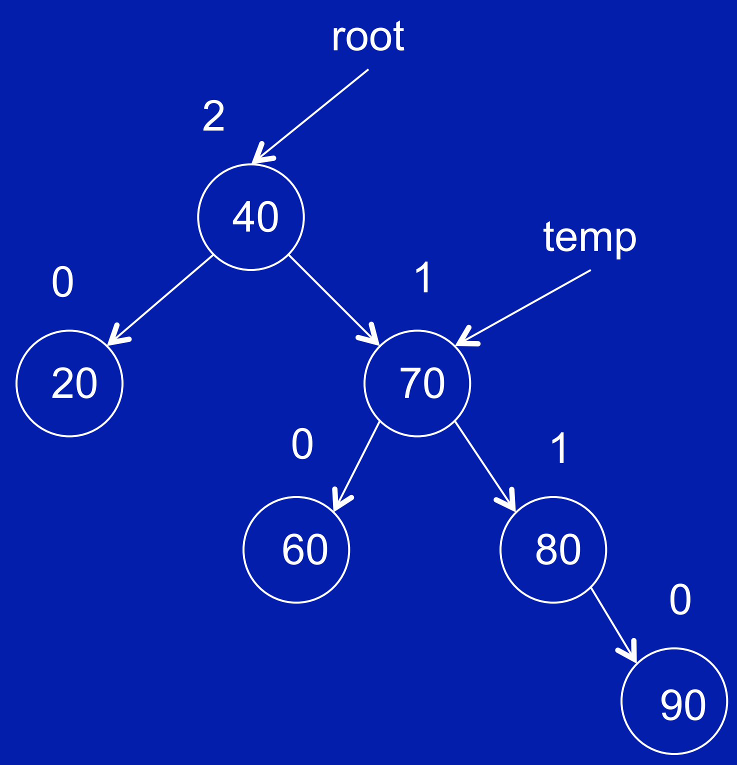


Rotate left is just the mirror image of rotate right….

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

Rotation

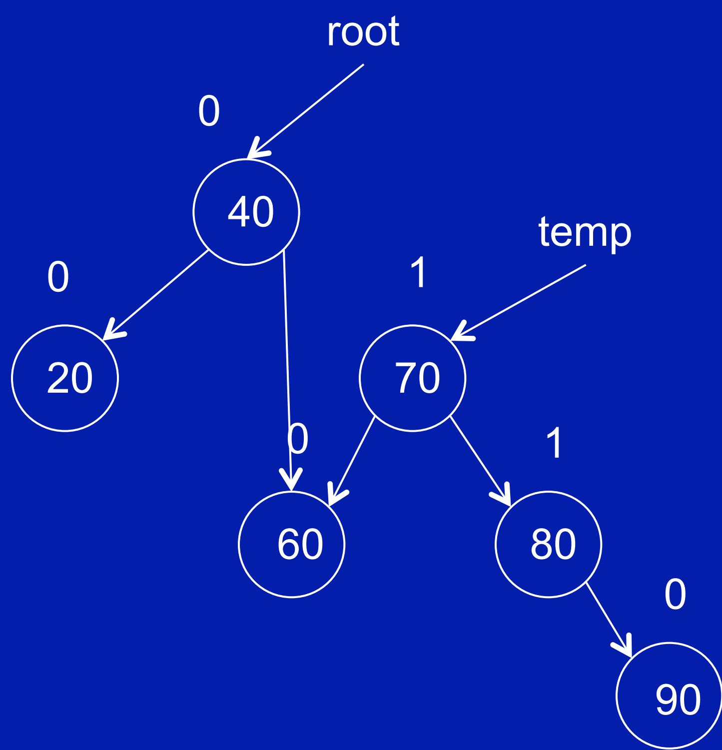


Rotate left is just the mirror image of rotate right….

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

Rotation

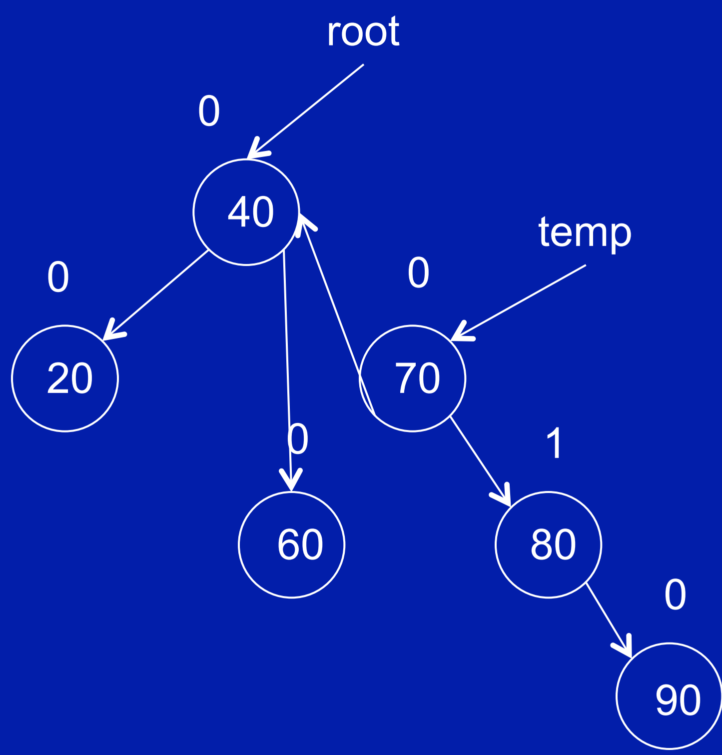


Rotate left is just the mirror image of rotate right….

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

Rotation

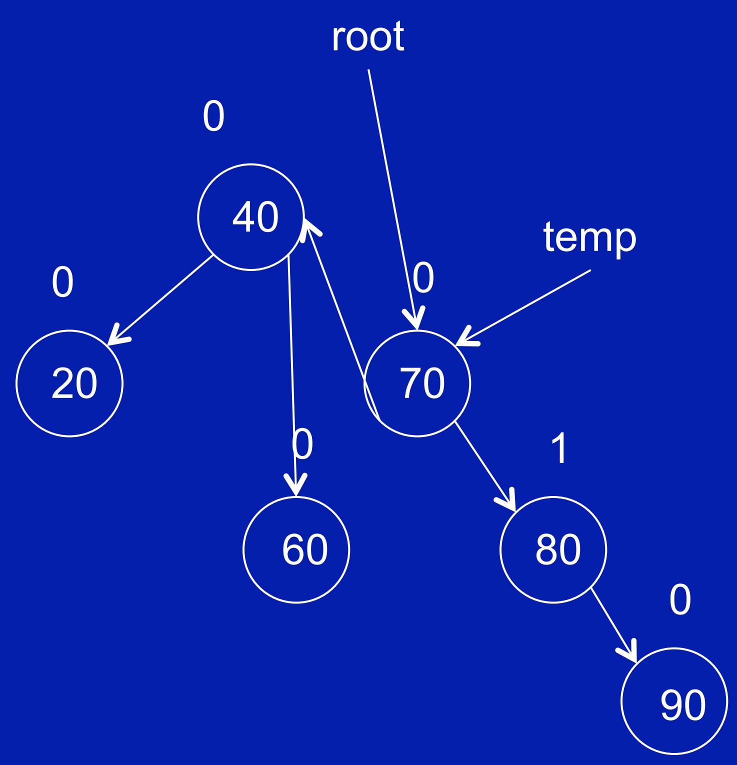


Rotate left is just the mirror image of rotate right….

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

Rotation

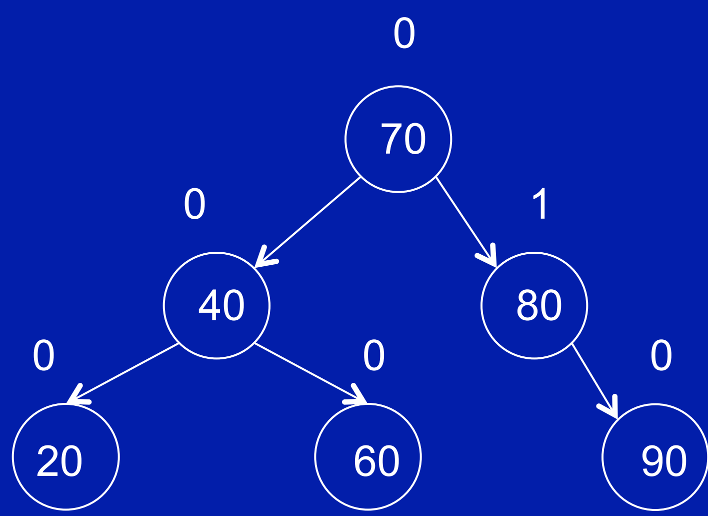


Rotate left is just the mirror image of rotate right….

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

Rotation



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

The AVL algorithm

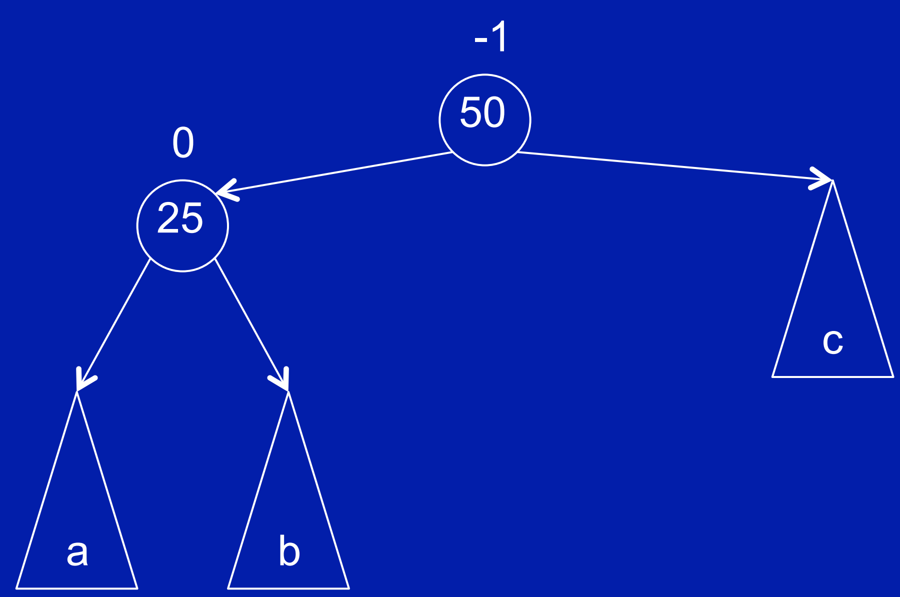
An AVL tree is a balanced binary search tree in which each node has a balance value that is equal to the difference between the heights of its right and left subtrees: $h_R - h_L$.

A node in the tree is balanced if it has a balance value of 0. A node is left-heavy if it has a balance of -1. A node is right-heavy if it has a balance of +1.

Rebalancing is done when a node along the insertion or deletion path has a balance of -2 or +2.

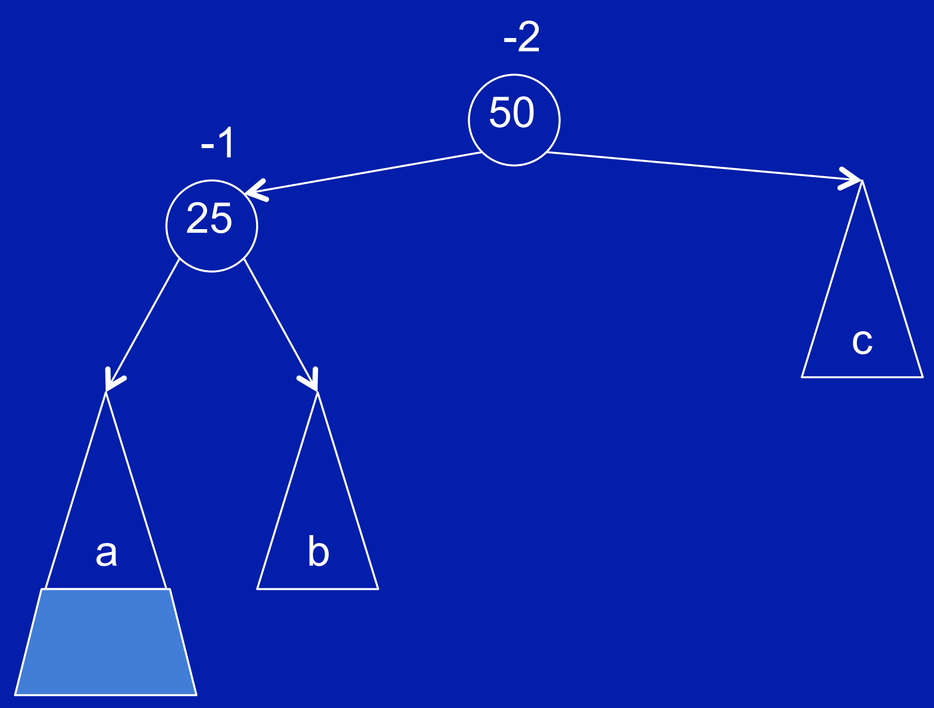
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Left tree (parent and child nodes are both left-heavy, parent balance is -2, child balance is -1). Fix by rotating right around the parent.



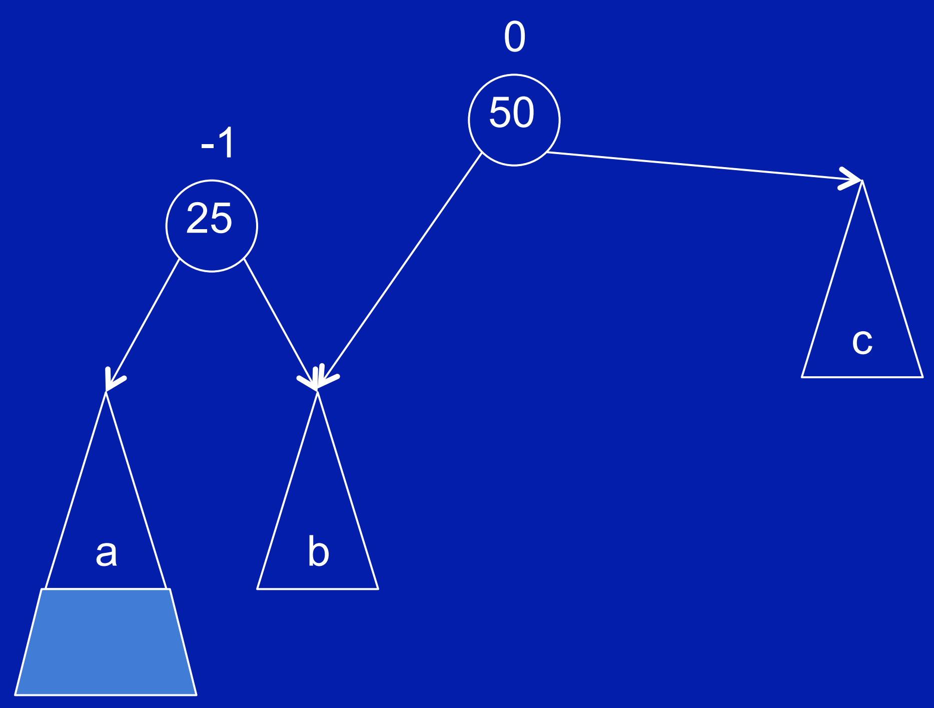
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Left tree (parent and child nodes are both left-heavy, parent balance is -2, child balance is -1). Fix by rotating right around the parent.



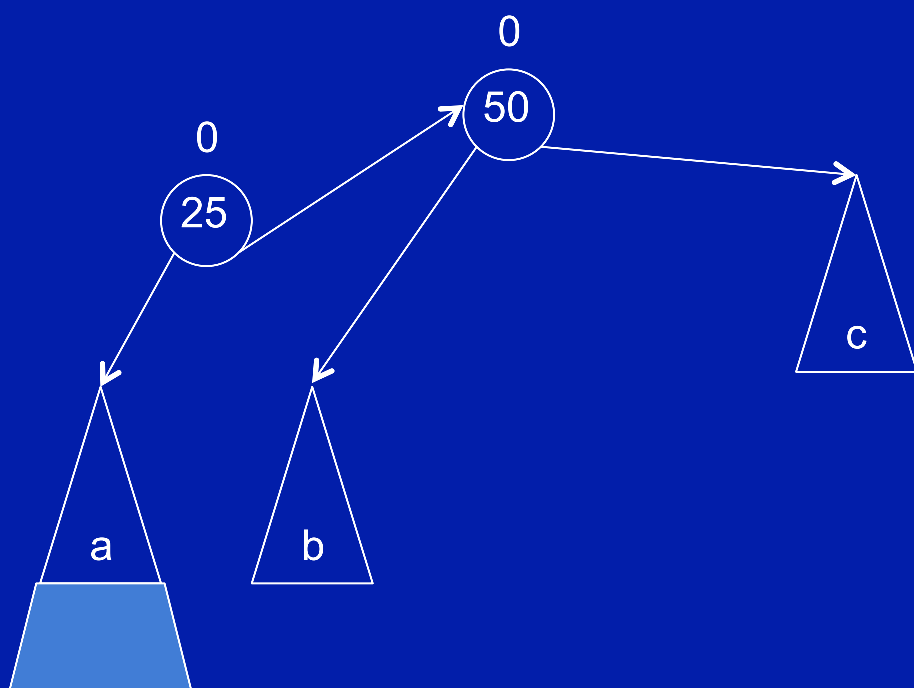
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Left tree (parent and child nodes are both left-heavy, parent balance is -2, child balance is -1). Fix by rotating right around the parent.



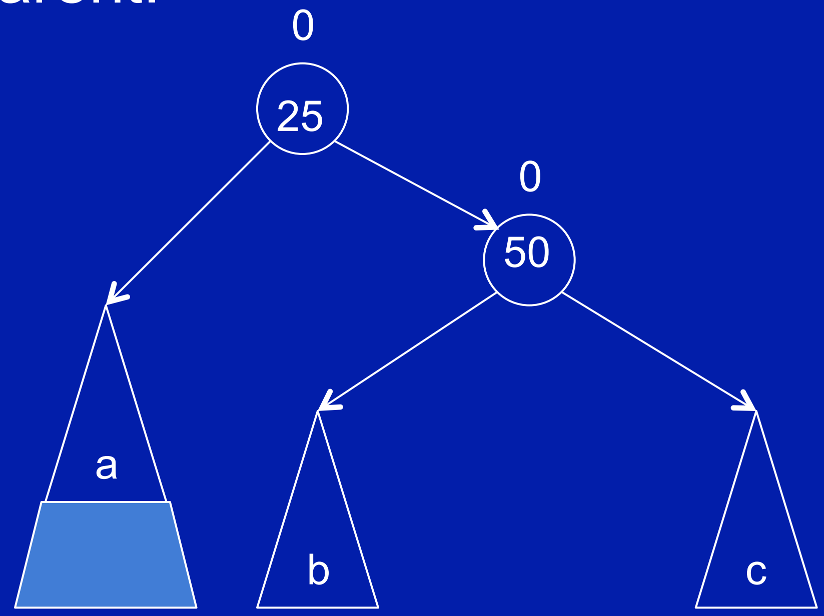
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Left tree (parent and child nodes are both left-heavy, parent balance is -2, child balance is -1). Fix by rotating right around the parent.



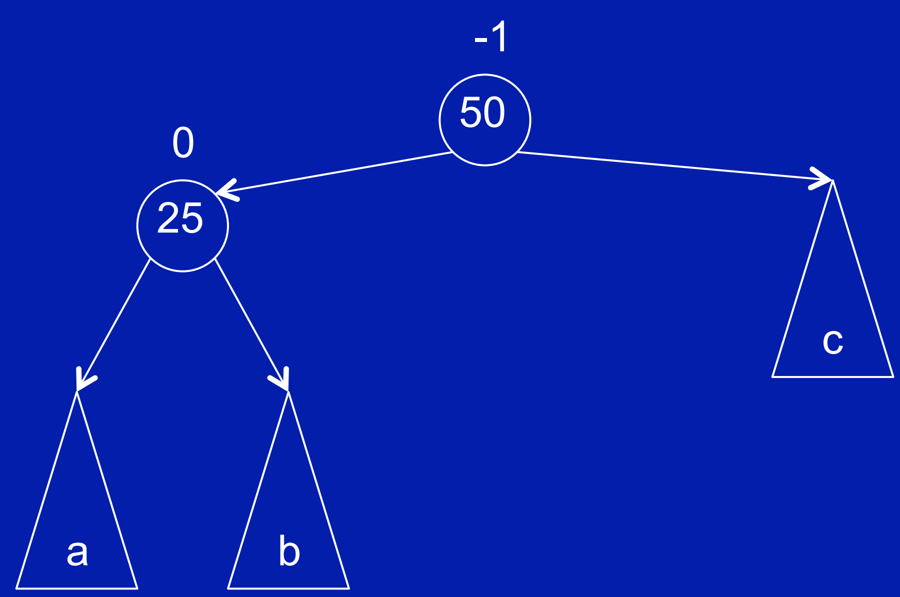
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Left tree (parent and child nodes are both left-heavy, parent balance is -2, child balance is -1). Fix by rotating right around the parent.



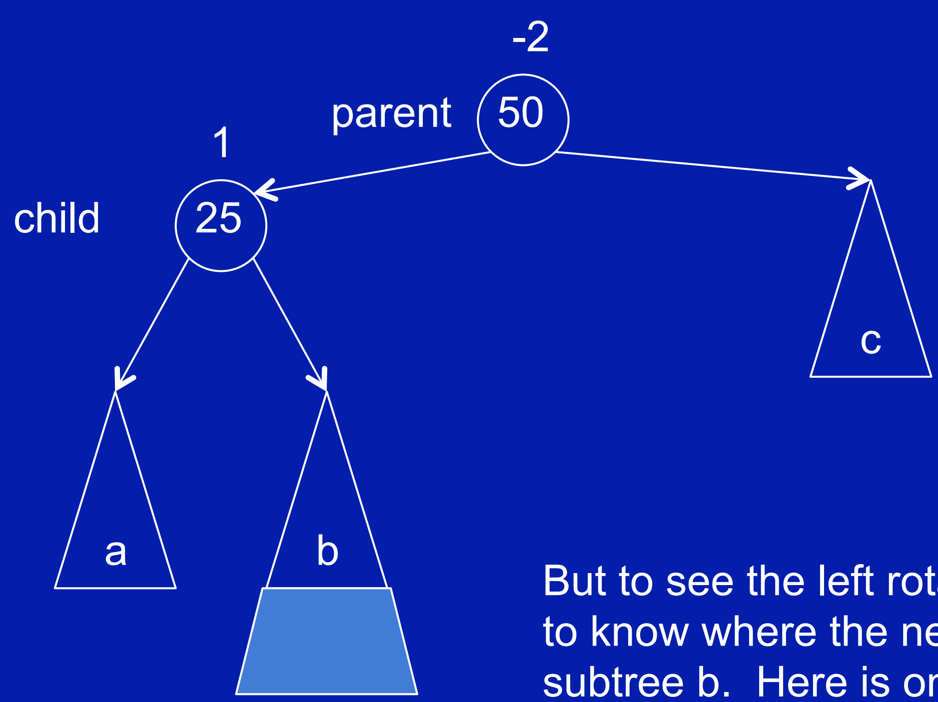
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



AVL trees

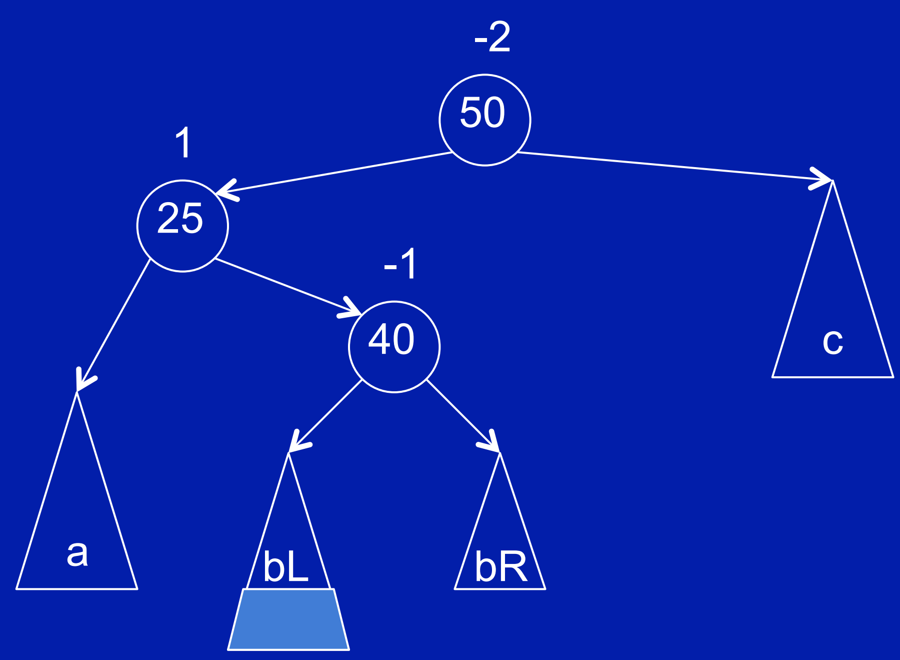
The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



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

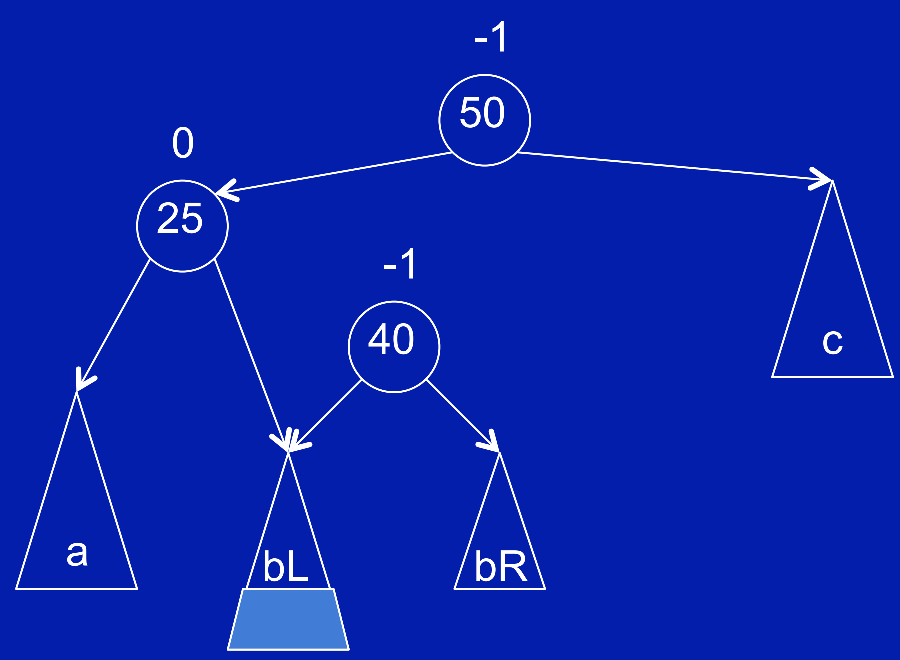
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



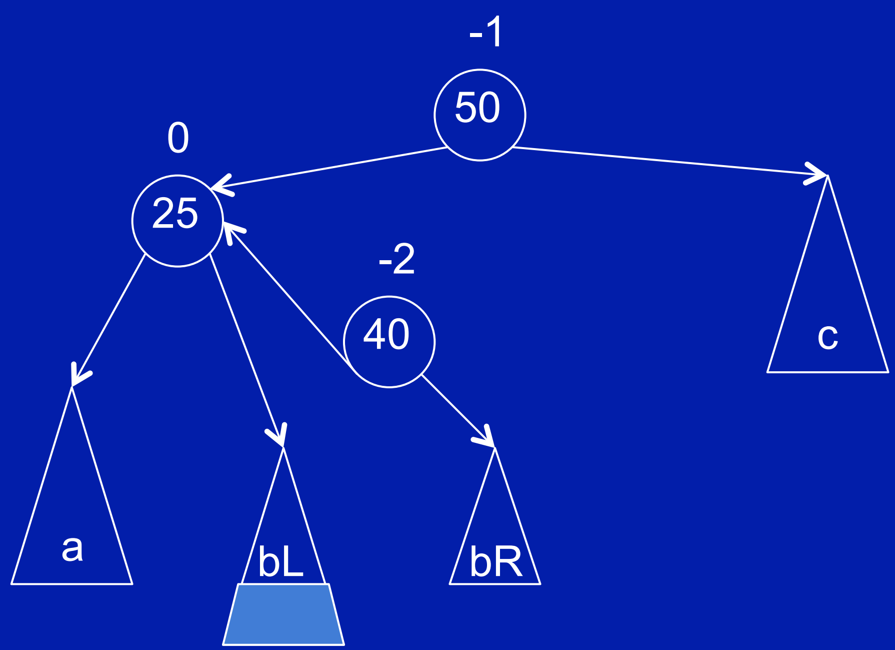
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



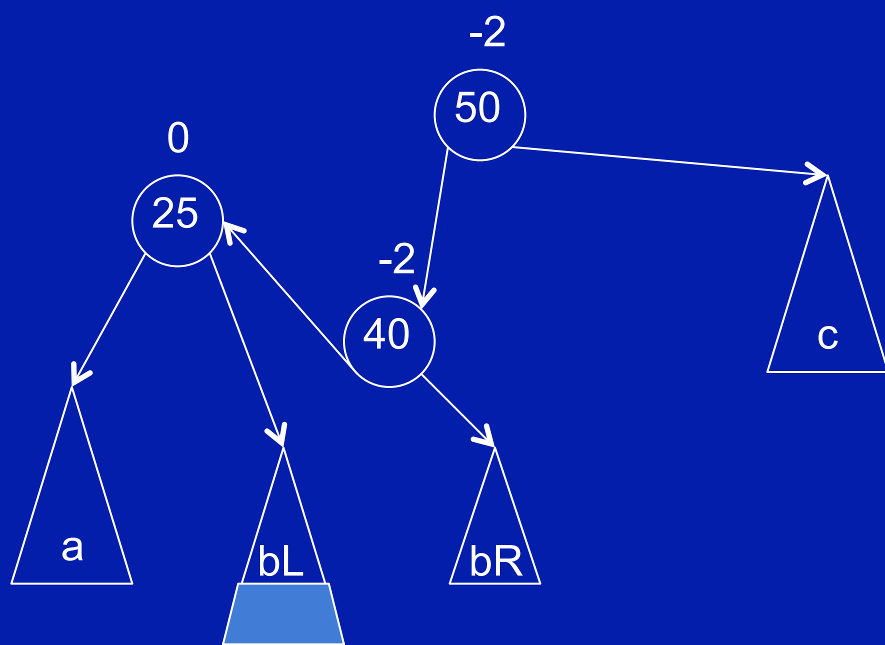
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



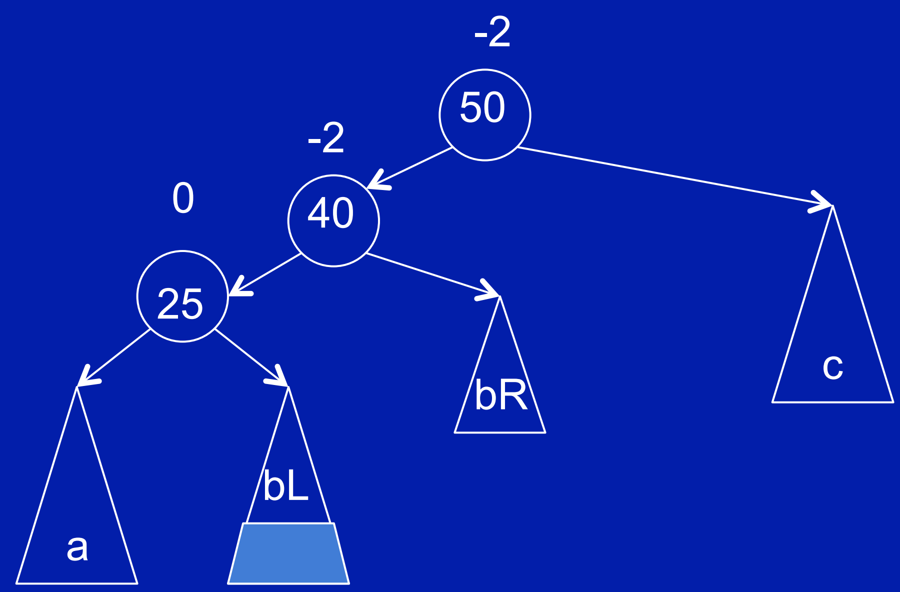
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



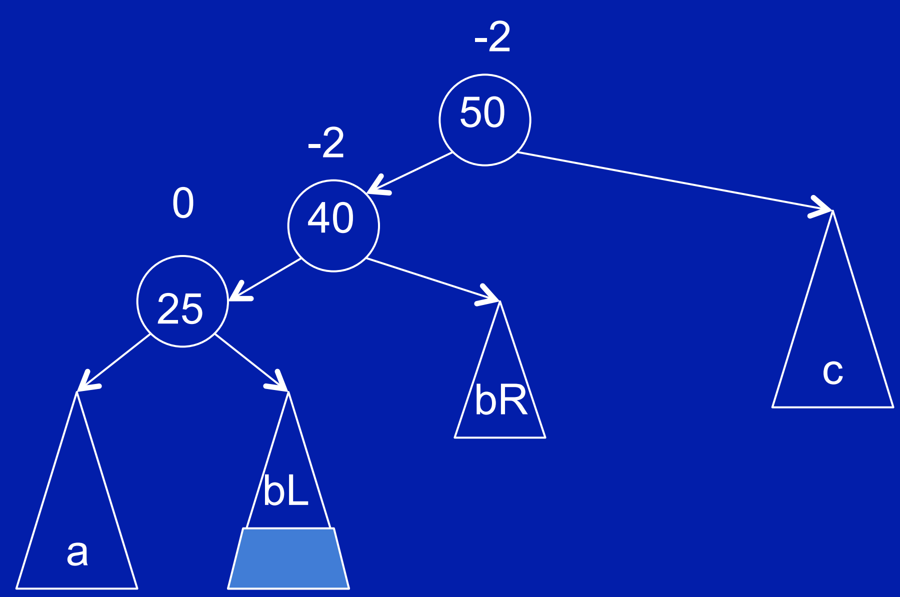
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



AVL trees

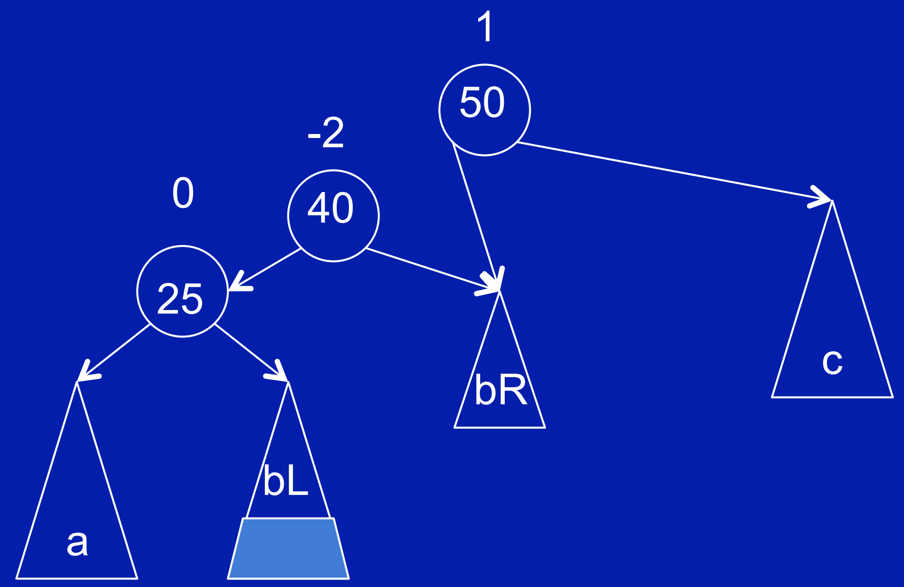
The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



Now right rotation around parent

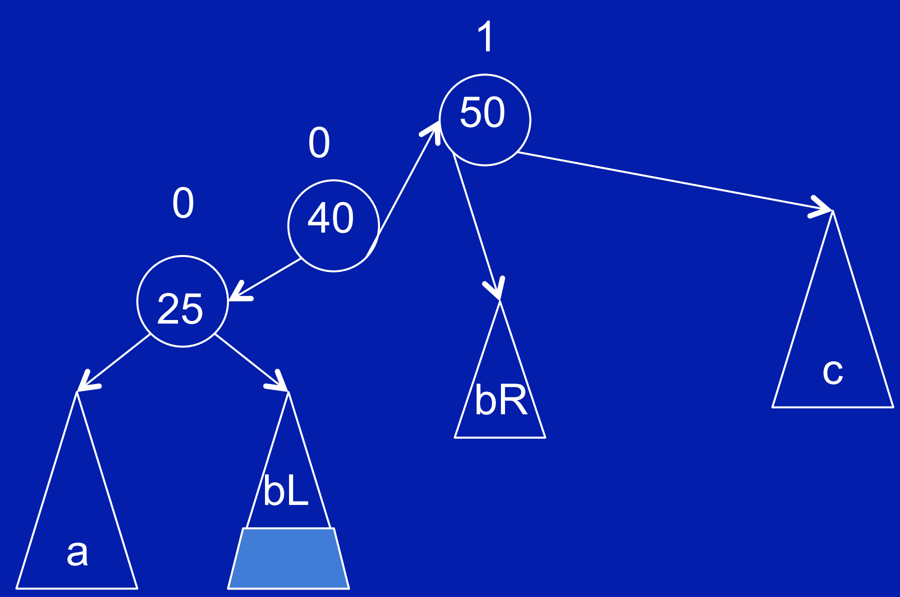
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



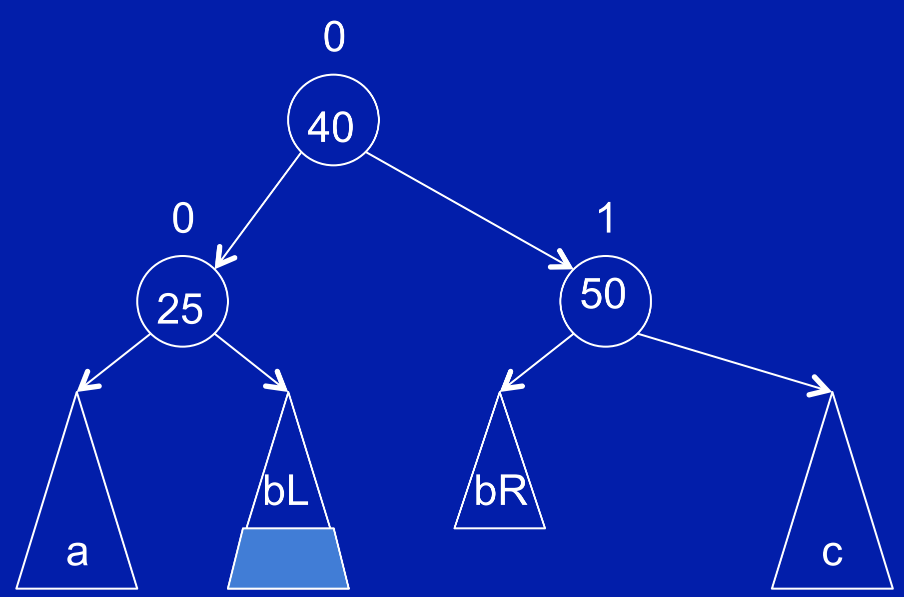
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



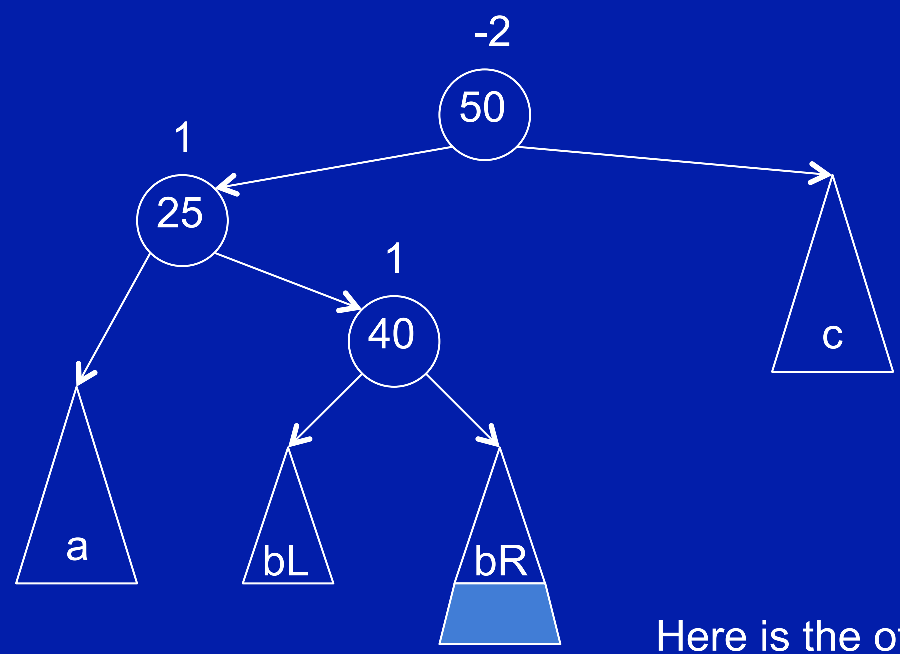
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



AVL trees

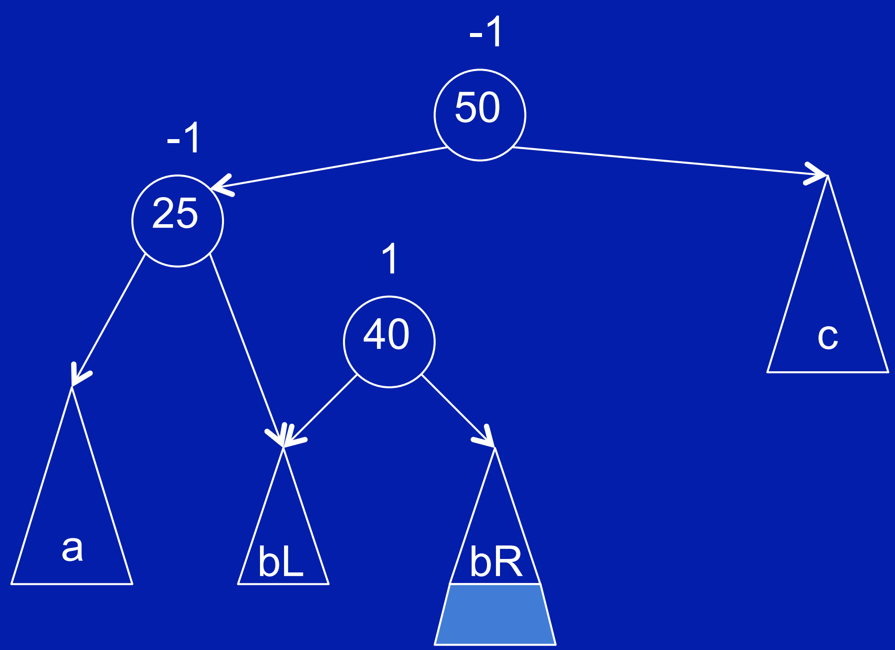
The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



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

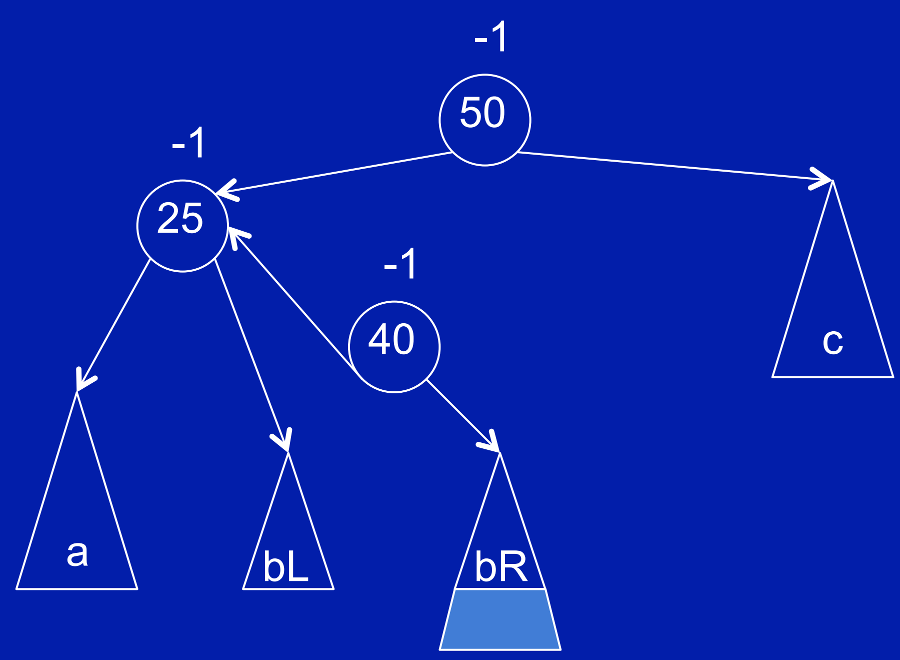
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



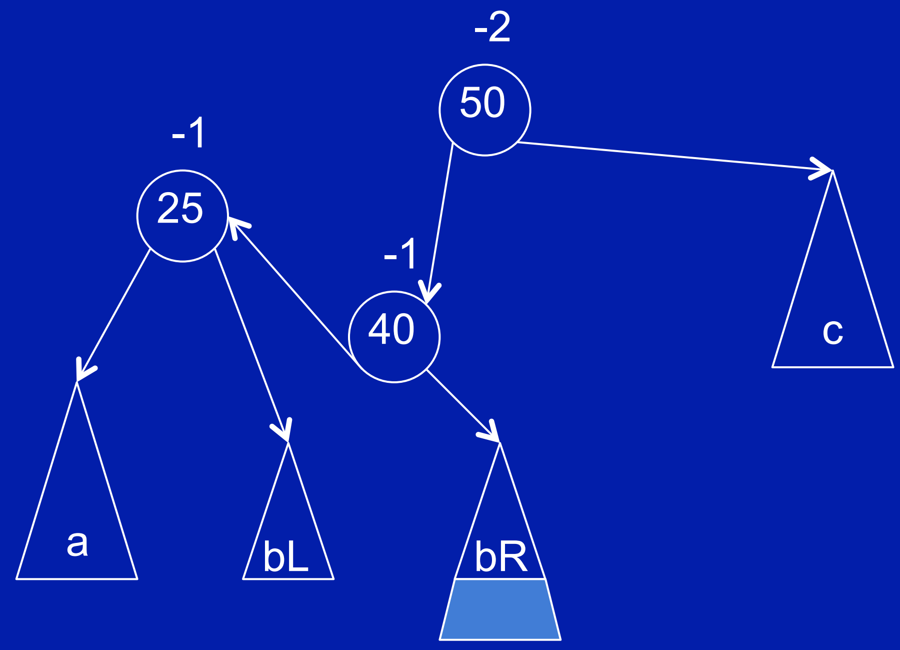
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



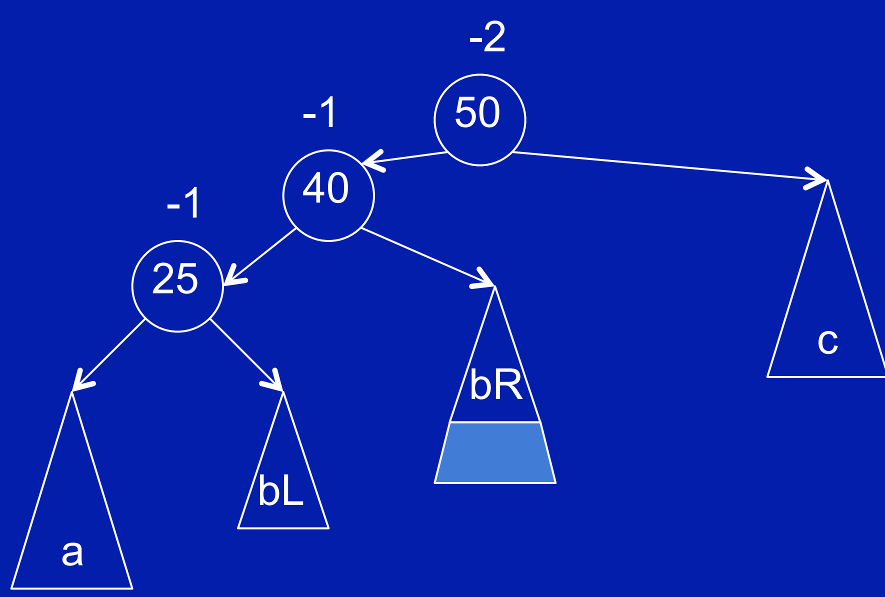
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



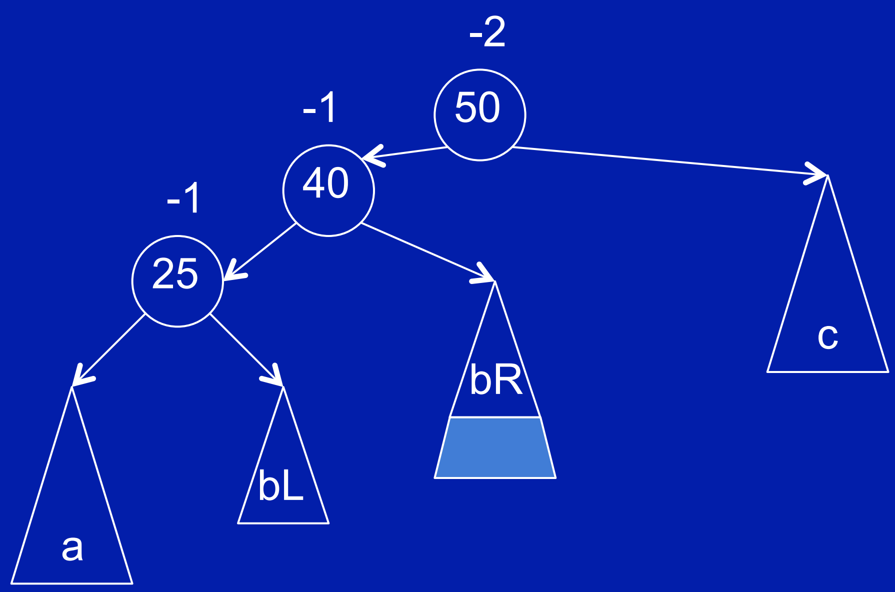
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



AVL trees

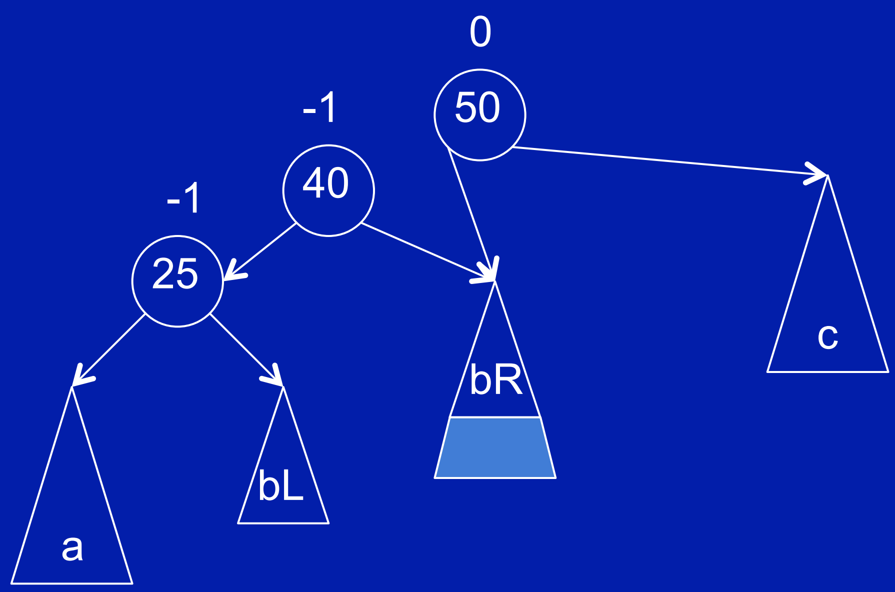
The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



Now right rotation around parent

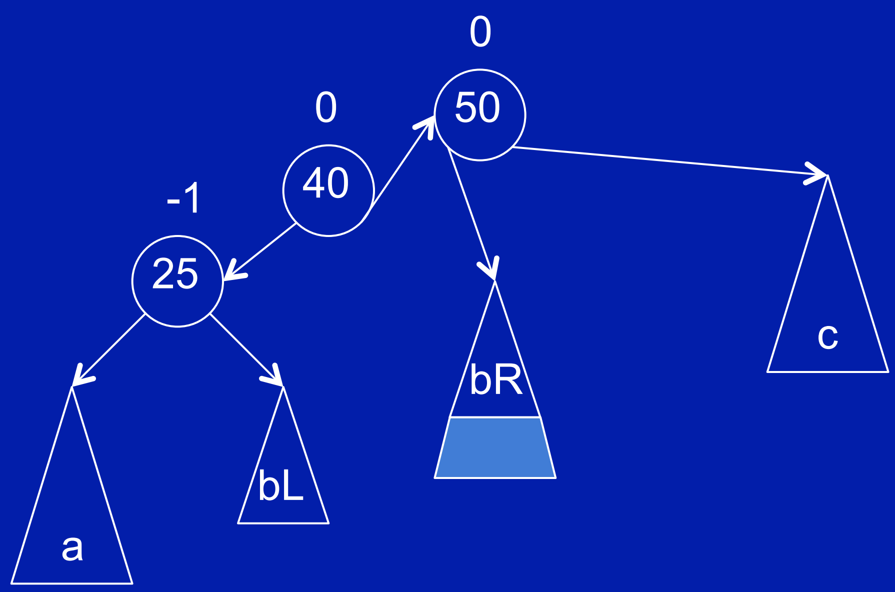
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



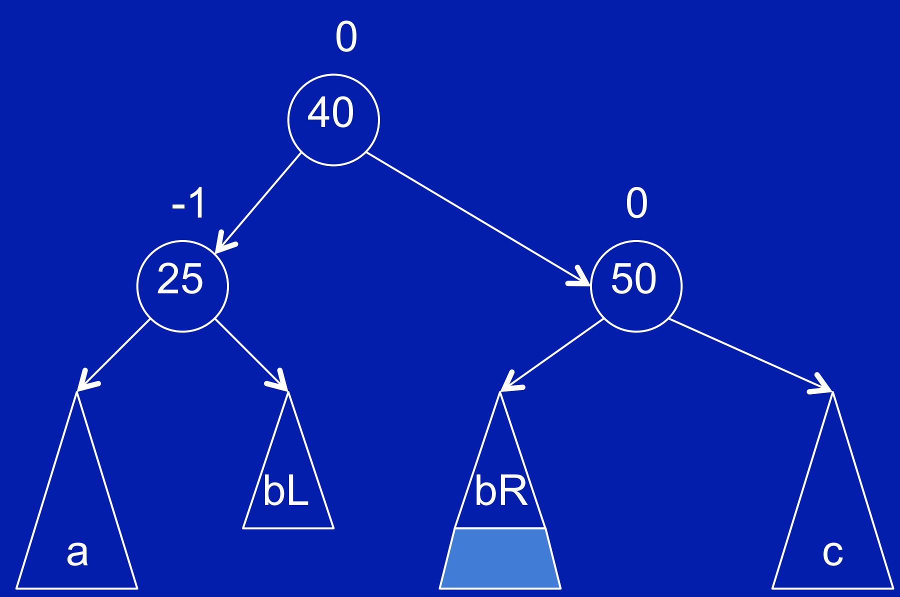
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



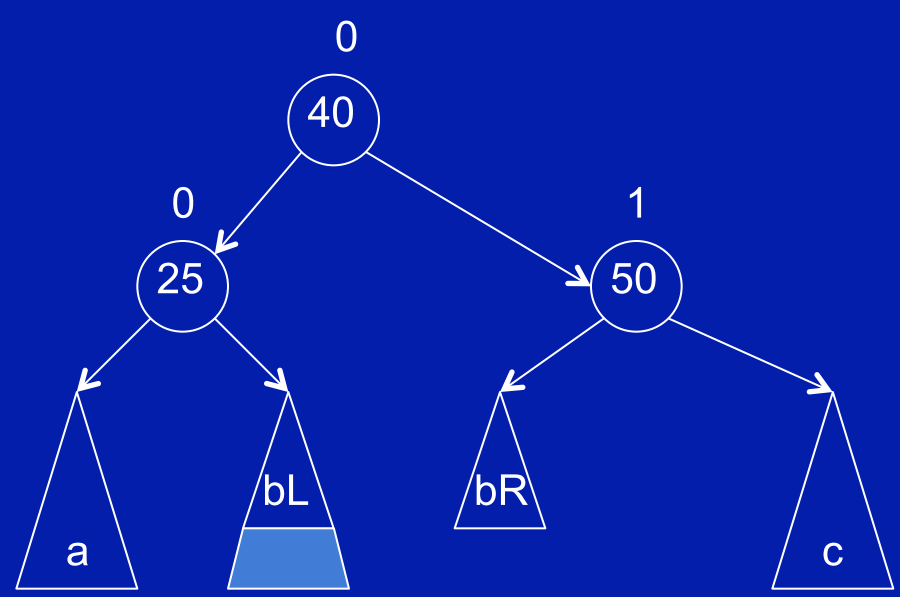
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:
The Left-Right tree (parent balance is -2, child balance is +1).
Fix by rotating left around the child then right around the parent.



AVL trees

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



AVL trees

Here are the other two kinds of unbalanced trees:

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

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

Right-Right is the mirror image of Left-Left. You've seen that.
Right-Left is the mirror image of Left-Right.
You should work out the details on your own.

What about deletion from AVL tree?

Data structures and algorithms textbooks tend to devote more space to insertion than deletion. Why? The principles of self-balancing apply to both insertion and deletion: you change the tree, you recompute balances, you perform the necessary rotations.

The big difference between insertion and deletion is this:

- Rebalancing after insertion requires at most two rotations.
- Rebalancing after deletion may require more rotations, depending on where the deleted node was and where its replacement came from.

In general, the number of rotations needed for rebalancing after deletion is O(log n).

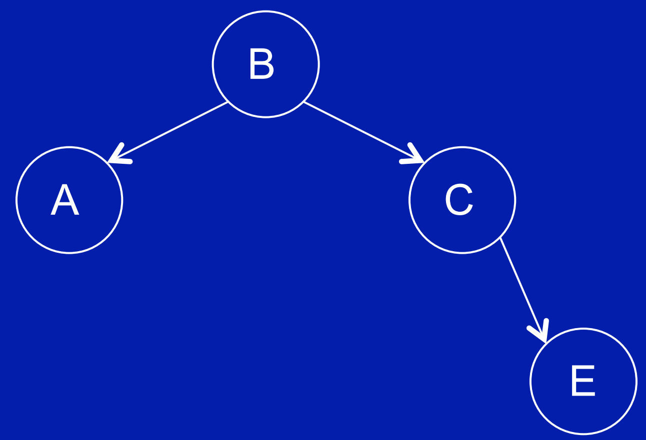
How to study this stuff!

Work through the examples by hand with paper and pencil. Make sure you can apply the AVL algorithm to new problems.

Even then, after you've done a few examples of rebalancing AVL trees, you may get the sense that you are now able to "eyeball" these things and re-balance without walking through the algorithm.

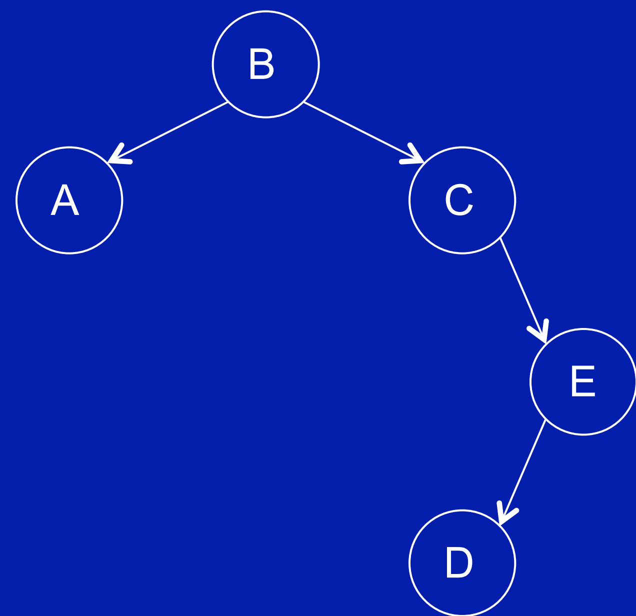
You may be right, but your final exam is not the place to test this newly-acquired skill for the first time…

Final exam from years ago



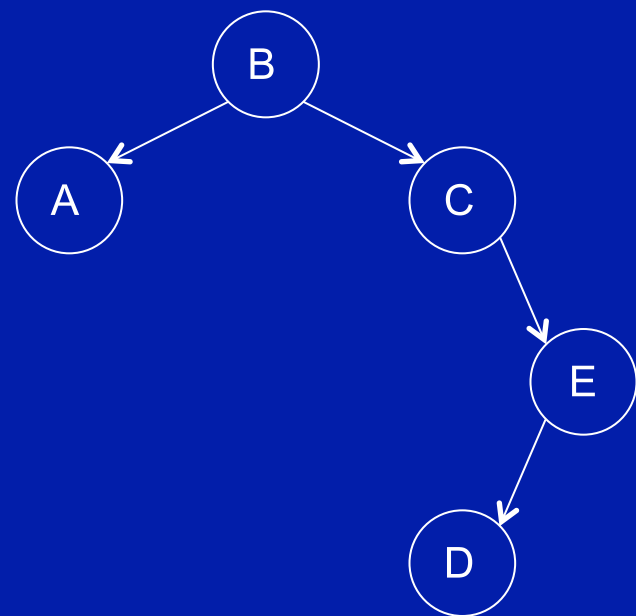
The AVL tree above contains search keys which are letters of the alphabet and which are compared in alphabetical order.

Final exam from years ago



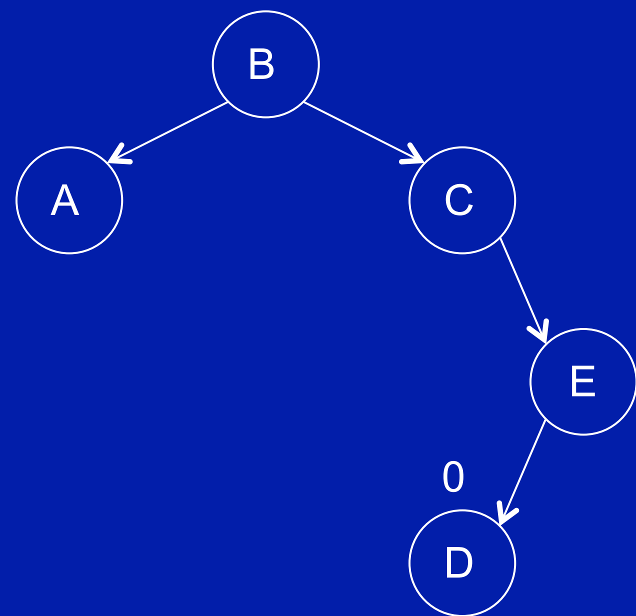
The AVL tree above contains search keys which are letters of the alphabet and which are compared in alphabetical order. When the search key D is inserted in this tree, it loses the AVL property. Draw the tree that results from inserting D and applying the proper rotation to restore the AVL property.

Final exam from years ago



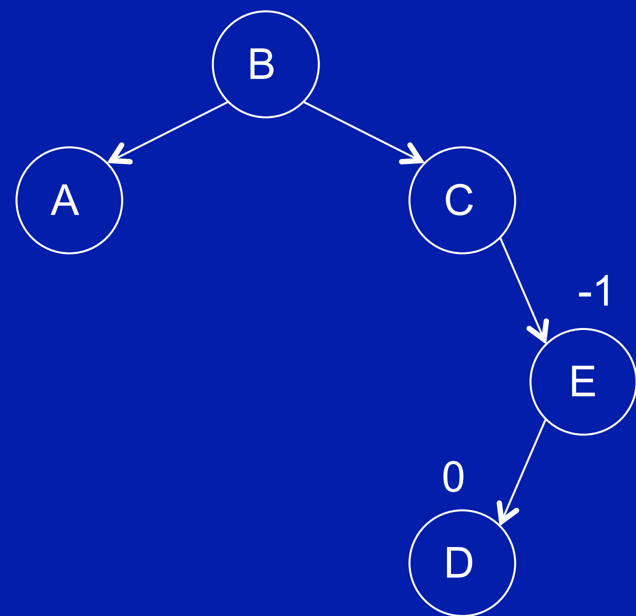
The AVL tree above contains search keys which are letters of the alphabet and which are compared in alphabetical order. When the search key D is inserted in this tree, it loses the AVL property. Draw the tree that results from inserting D and applying the proper rotations to restore the AVL property.

Final exam from years ago



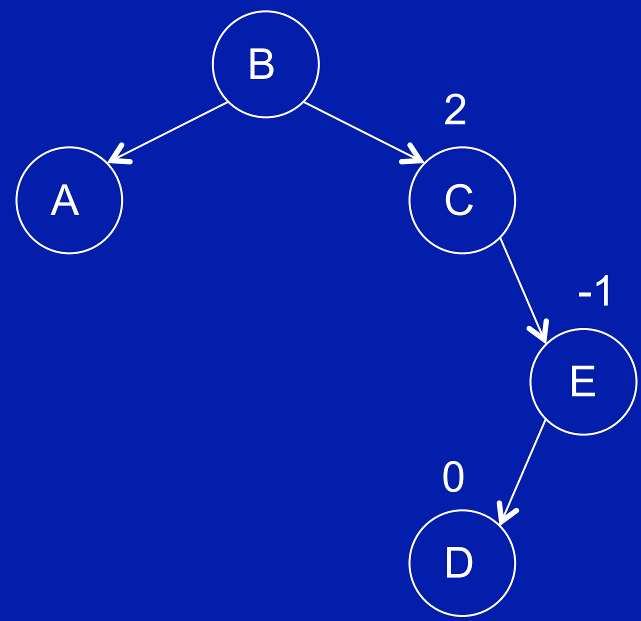
The AVL tree above contains search keys which are letters of the alphabet and which are compared in alphabetical order. When the search key D is inserted in this tree, it loses the AVL property. Draw the tree that results from inserting D and applying the proper rotations to restore the AVL property.

Final exam from years ago



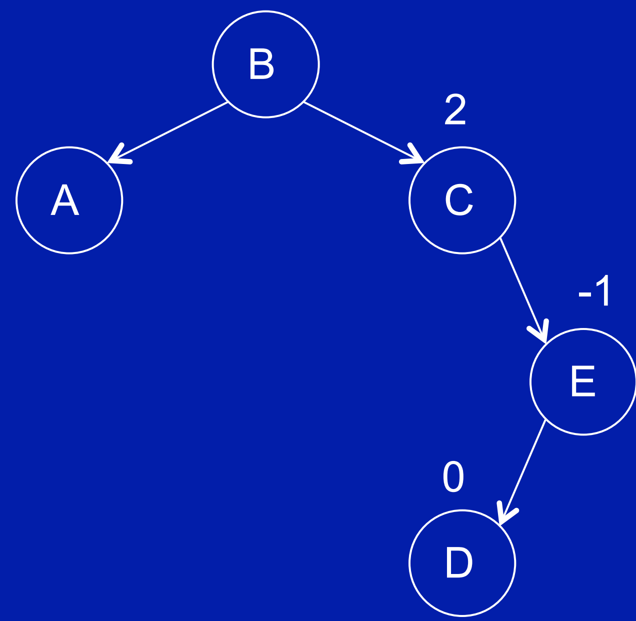
The AVL tree above contains search keys which are letters of the alphabet and which are compared in alphabetical order. When the search key D is inserted in this tree, it loses the AVL property. Draw the tree that results from inserting D and applying the proper rotations to restore the AVL property.

Final exam from years ago



The AVL tree above contains search keys which are letters of the alphabet and which are compared in alphabetical order. When the search key D is inserted in this tree, it loses the AVL property. Draw the tree that results from inserting D and applying the proper rotations to restore the AVL property.

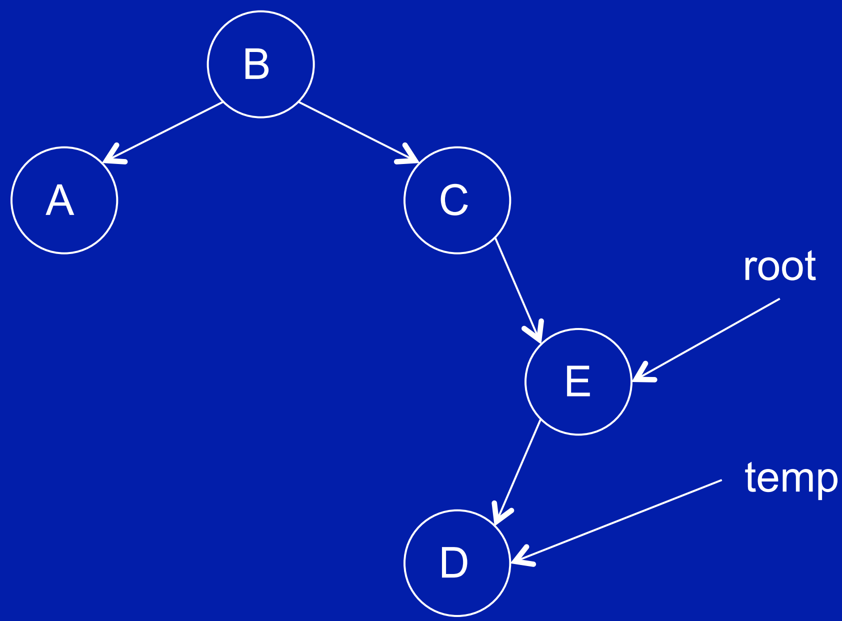
Final exam from years ago



The 2 at C and -1 at E says this is a right-left tree. First we rotate right around the child (E). The child is the root of the first subtree to be rotated.

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

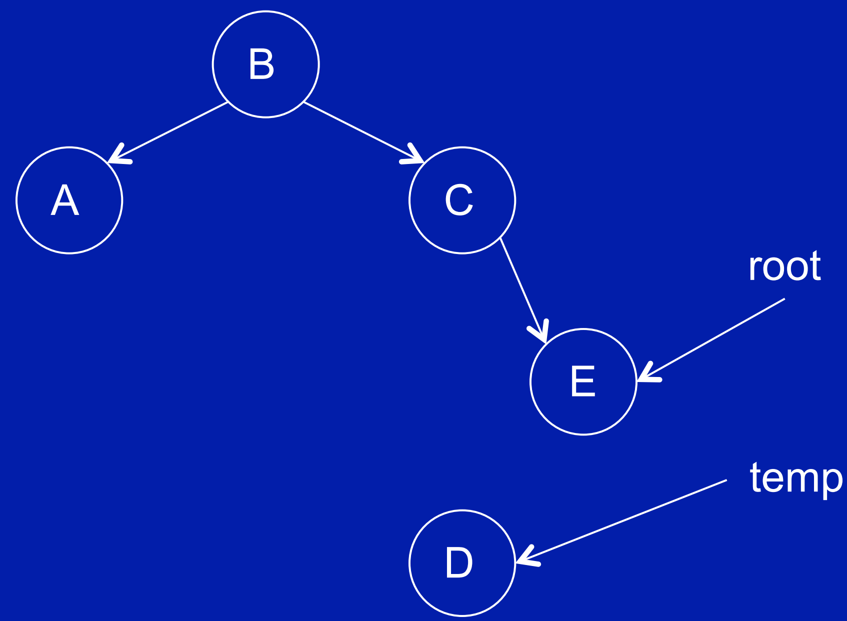
Final exam from years ago



The 2 at C and -1 at E says this is a right-left tree. First we rotate right around the child (E). The child is the root of the first subtree to be rotated.

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

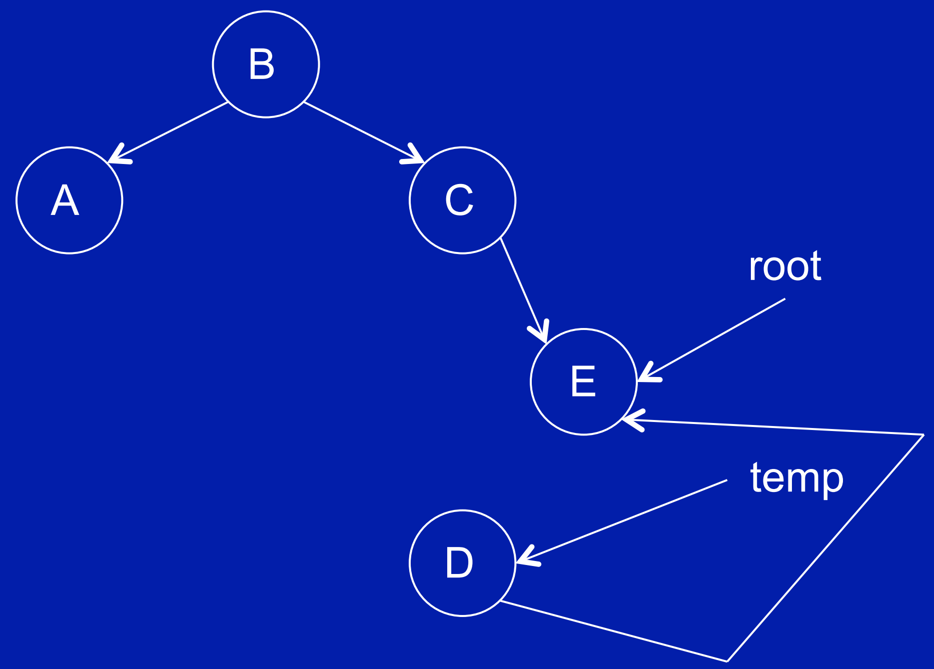
Final exam from years ago



The 2 at C and -1 at E says this is a right-left tree. First we rotate right around the child (E). The child is the root of the first subtree to be rotated.

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

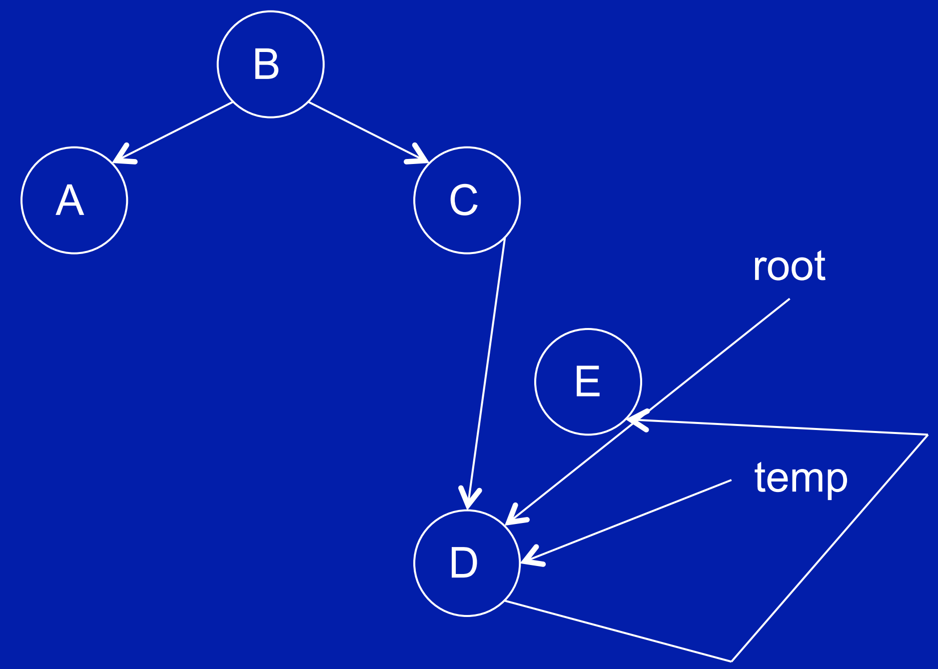
Final exam from years ago



The 2 at C and -1 at E says this is a right-left tree. First we rotate right around the child (E). The child is the root of the first subtree to be rotated.

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

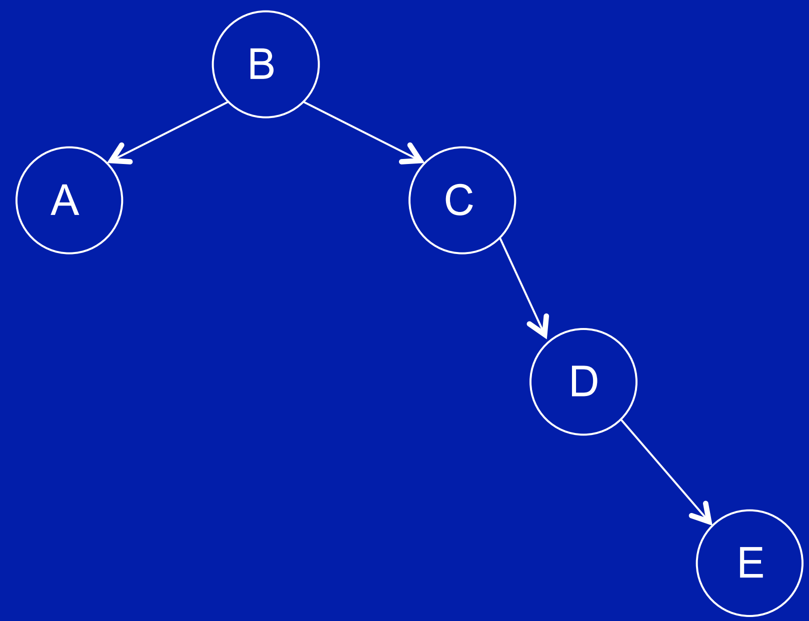
Final exam from years ago



The 2 at C and -1 at E says this is a right-left tree. First we rotate right around the child (E). The child is the root of the first subtree to be rotated.

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

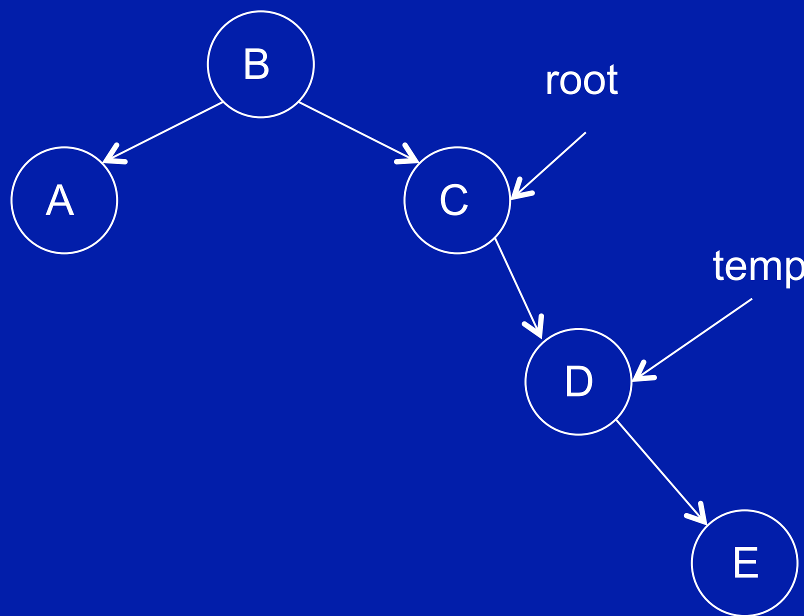
Final exam from years ago



So far, so good. But we're not done yet. We now do a left rotation around the parent (C). The parent is the root of the second subtree to be rotated.

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

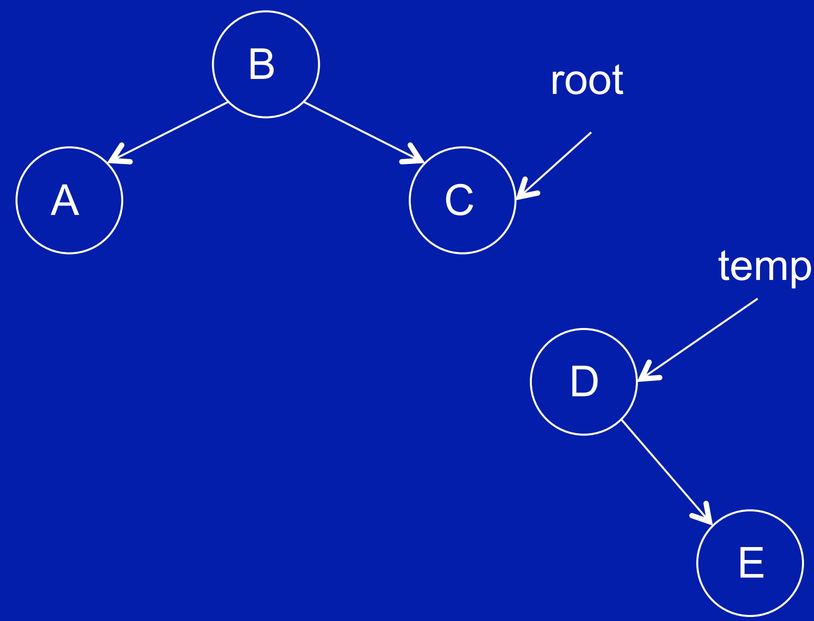
Final exam from years ago



So far, so good. But we're not done yet. We now do a left rotation around the parent (C). The parent is the root of the second subtree to be rotated.

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

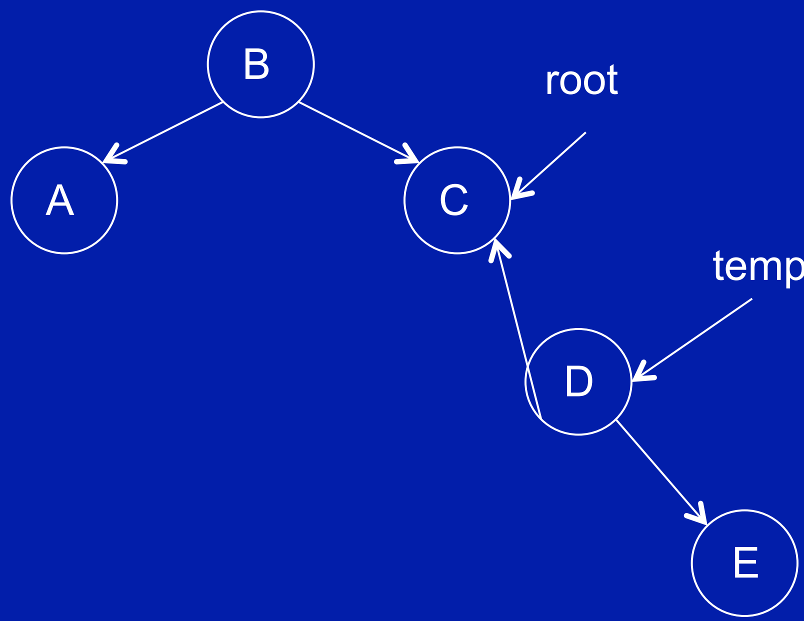
Final exam from years ago



So far, so good. But we're not done yet. We now do a left rotation around the parent (C). The parent is the root of the second subtree to be rotated.

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

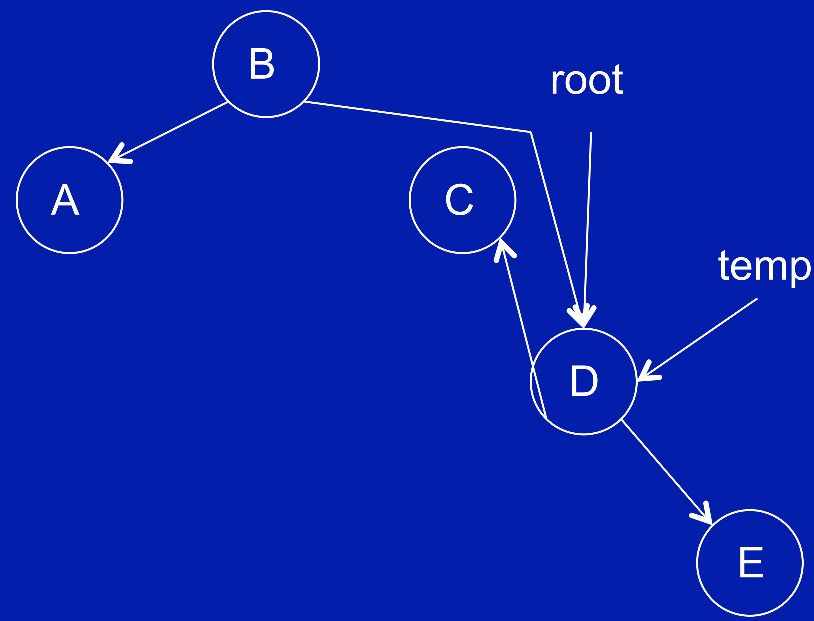
Final exam from years ago



So far, so good. But we're not done yet. We now do a left rotation around the parent (C). The parent is the root of the second subtree to be rotated.

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

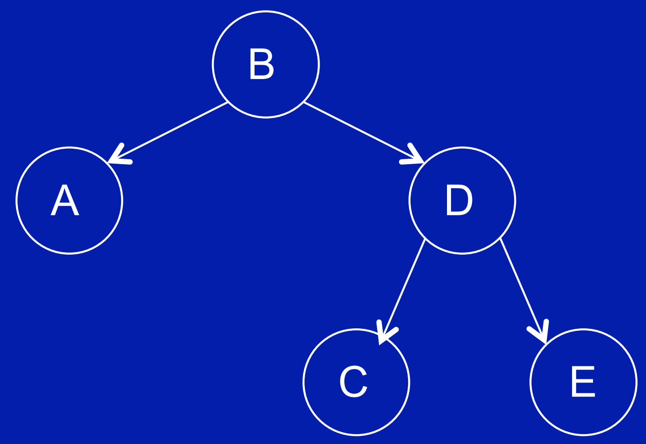
Final exam from years ago



So far, so good. But we're not done yet. We now do a left rotation around the parent (C). The parent is the root of the second subtree to be rotated.

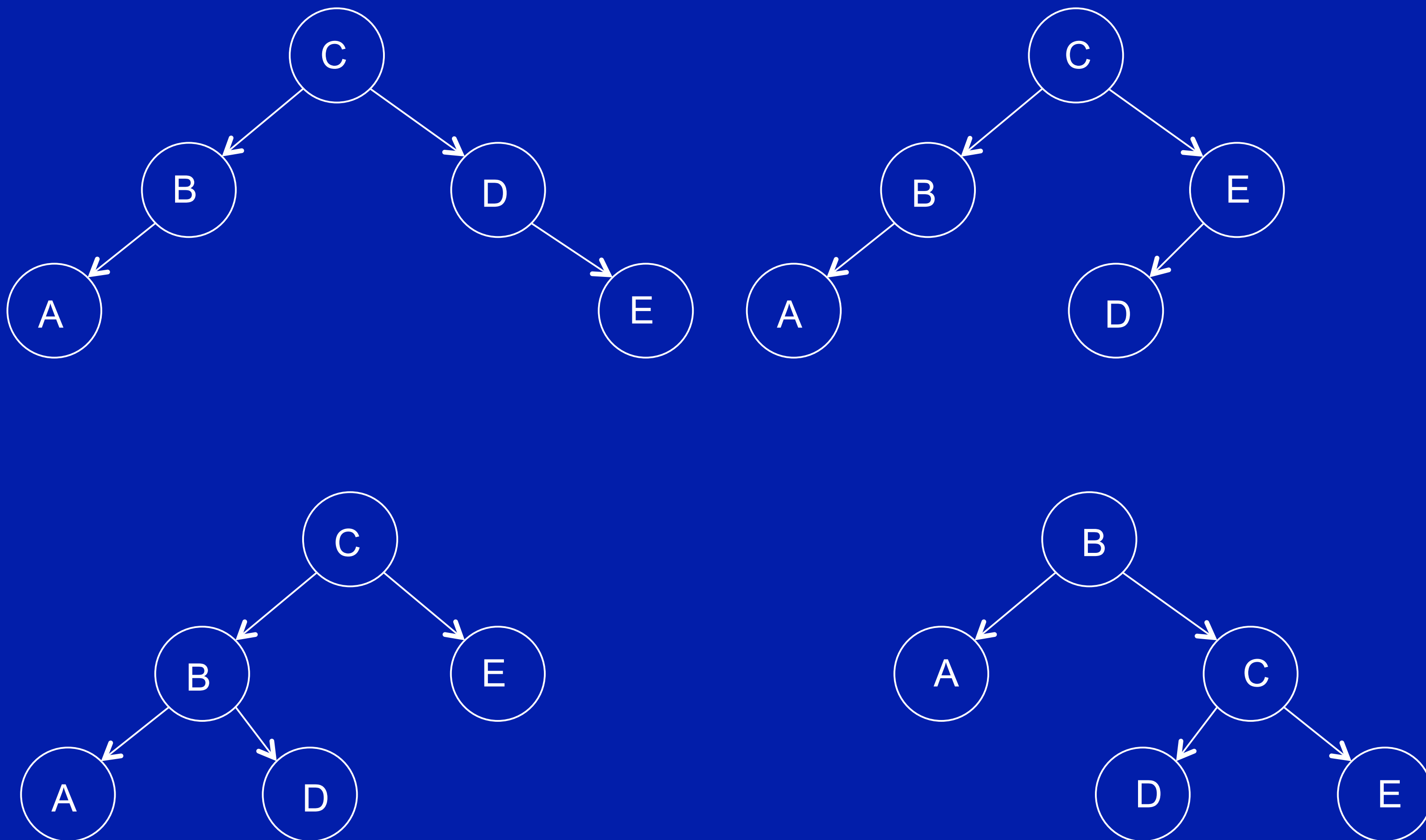
1. Remember the value of root->right (temp = root->right)
2. Set root->right to value of temp->left
3. Set temp->left to root
4. Set root to temp

Final exam from years ago



Here's the final result. This seems fairly obvious, despite what we went through to get here. Yet students who clearly didn't apply the AVL algorithm correctly, if at all, came up with these answers among others:

Some answers from years ago



A practice problem

What if we insert the values 1, 2, 3, 4, 5, 6, 7 into an empty AVL tree in that order?

Do we get the tree at the right?

What shape will we get?