Binary Search Trees (continued)

Recap
Deleting internal nodes from a BST
Examples
Complexity, and an issue…

Recap

A Binary Search Tree is meant to be efficient for searching and manipulating

A BST is a Binary Tree such that:

- all left descendants of a node have values less than the node's value
- all right descendants of a node have values greater than the node's value

To search a BST from a node, looking for x:

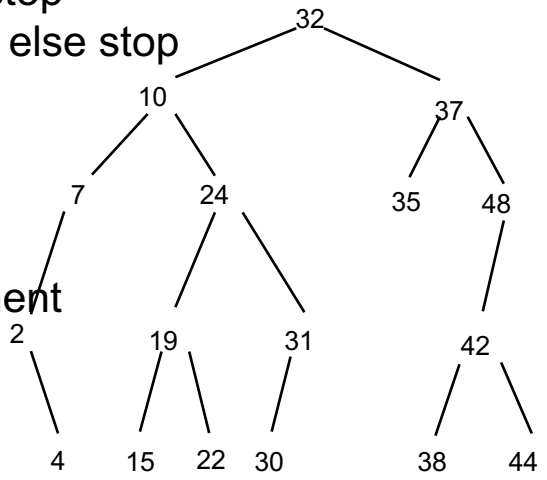
if x < node.element, search from left childnode if it exists; else stop
else if x > node.element, search from right childnode if it exists; else stop
else if x == node.element, found it

To add a new element x to a BST:

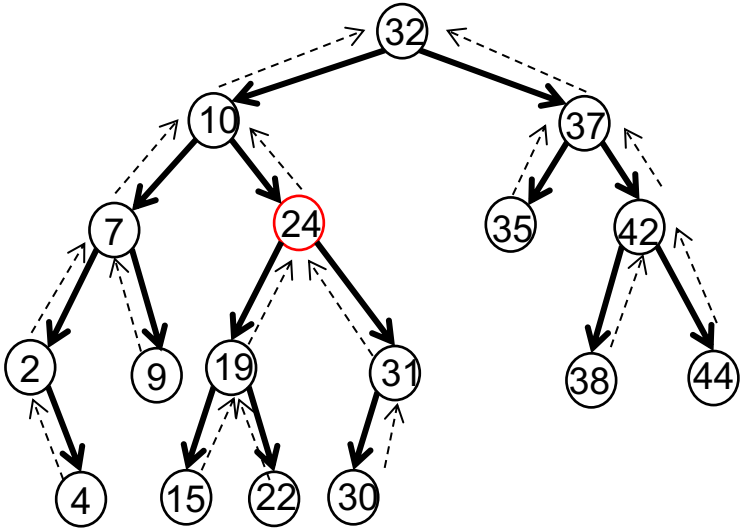
find the place where x should be
if it is not there, add a new node in that place, with x as its element

To remove an element x, find its node N, and do:

if N is a leaf, remove it (and clean up links)
else if N is a semileaf, move its child node up into position (and clean up)
else ....?



Removing an internal node from a BST



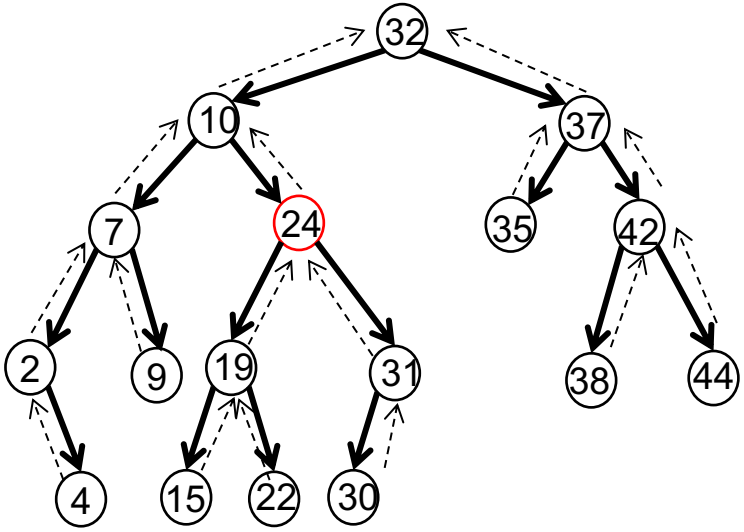
Example: remove 24

We are going to have to do some work - deleting, or moving one child up, might cut off one or more existing children, or break the order

We will have to pick one of the items below our node, and then rearrange the subtree to maintain the order

Which choice requires least work?
Can we avoid making the tree deeper?

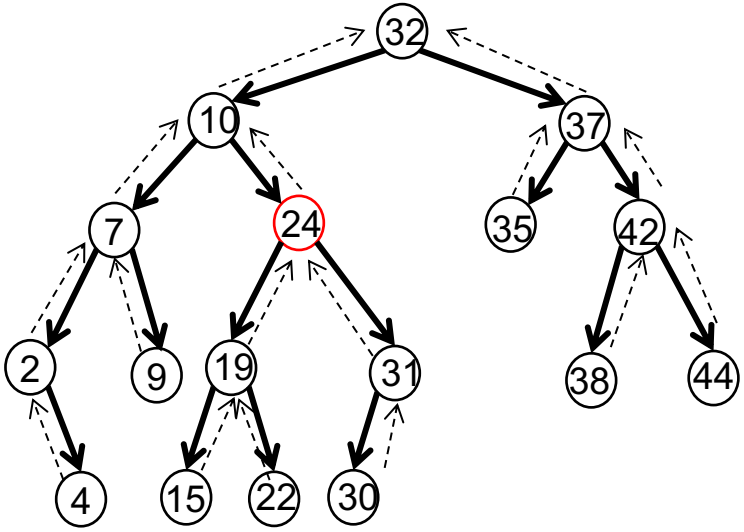
Removing an internal node from a BST (ii)



Example: remove 24

1. We will always pick the *biggest* element less than our current node
2. We will move that *element* straight into our current node
3. Then remove the node we took the element from
4. Return the original element

Finding biggest element less than current



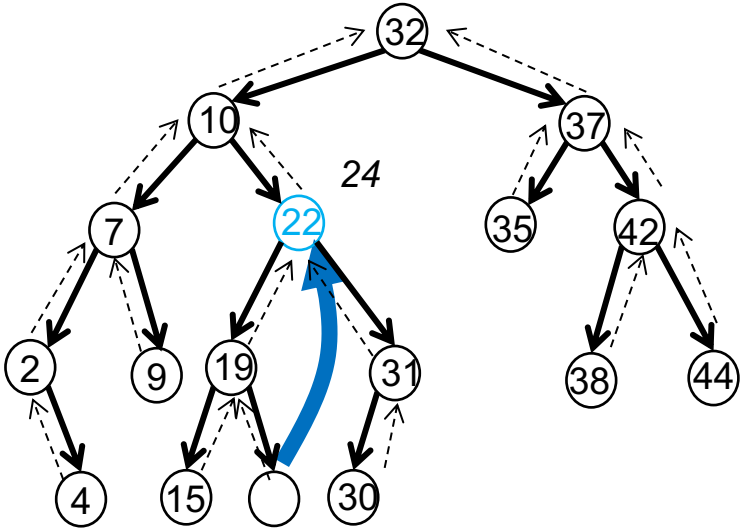
Exercise: how do we find the node with the biggest element less than a given node's element?

Solution:

- move to the left child
- then keep taking the right child until we find a null

- the discovered node does not have a right child ...

Removing an internal node from a BST (iii)

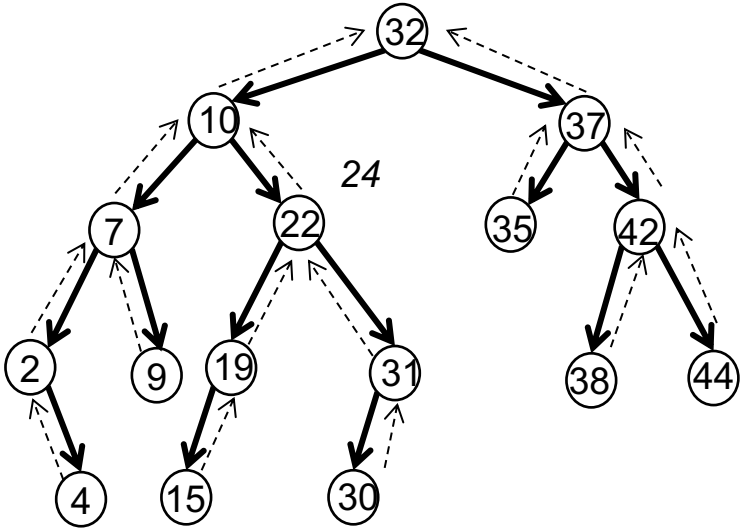


Example: remove 24

1. We will always pick the *biggest* element less than our current node
2. We will move that *element* straight into our current node
3. Then remove the node we took the element from
4. Return the original element

Moving the biggest element maintains the order:
- every right descendant of the removed value must be bigger
- every other left descendant of the removed value must be smaller

Removing an internal node from a BST (iv)



Example: remove 24

1. We will always pick the *biggest* element less than our current node
2. We will move that element straight into our current node
3. Then remove the node we took the element from
4. Return the original element

The node we moved the element from does **not** have any right child, so it is a semileaf or leaf – and we know how to remove these.

Could also have chosen to replace a removed internal node with the smallest element that is bigger than it.

Recap (again)

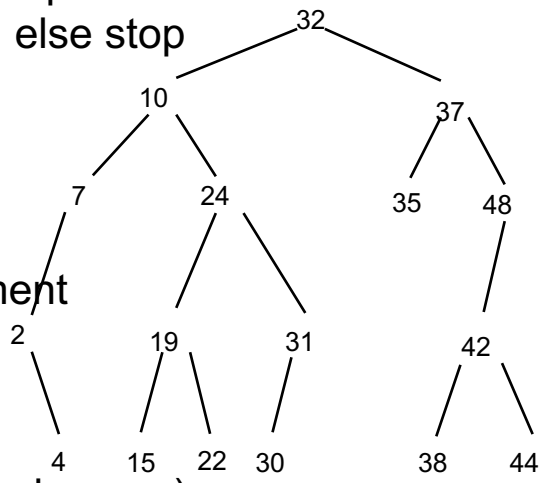
A Binary Search Tree is meant to be efficient for searching and manipulating

A BST is a Binary Tree such that:
- all left descendants of a node have values less than the node's value
- all right descendants of a node have values greater than the node's value

To search a BST from a root node, looking for x:
    if x < node.element, search from left childnode if it exists; else stop
    else if x > node.element, search from right childnode if it exists; else stop
    else if x == node.element, found it

To add a new element x to a BST:
    find the place where x should be
    if it is not there, add a new node in that place, with x as its element

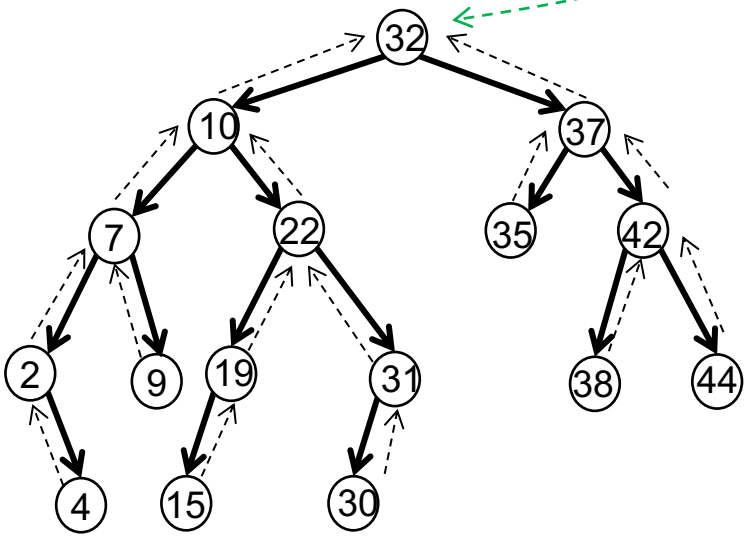
To remove an element x, find its node N, and do:
    if N is a leaf, remove it (and clean up links)
    else if N is a semileaf, move its child node up into position (and clean up)
    else find the node Z with the biggest element in N's left subtree
        move Z's element up into N
        remove node Z (i.e. a recursive call)



Note: this is not clear pseudocode …

Removing an internal node from a BST (iv)

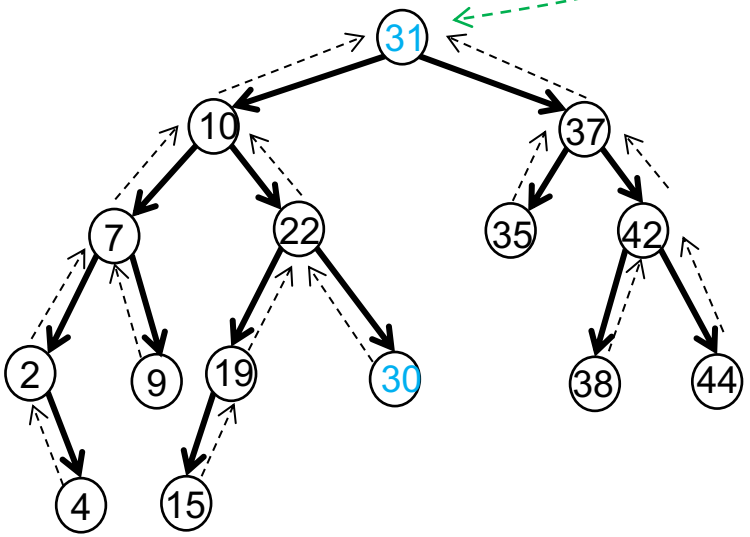
We know the value we want to remove: x
We know the location of this node:



Class Exercise: remove 32

Removing an internal node from a BST (iv)

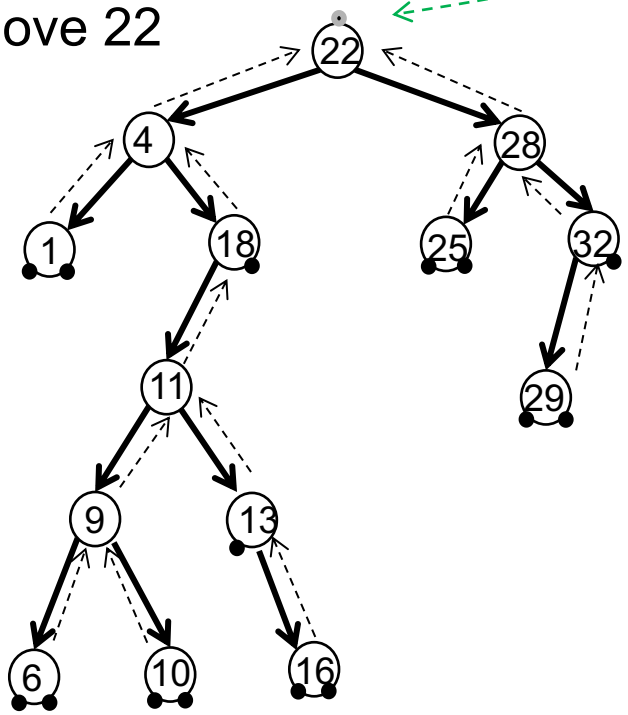
We know the value we want to remove: x
We know the location of this node:



Exercise

We know the value we want to add or remove: x
We know the location of this node:

Add 27
Remove 22



Removing a node from a BST: complexity

To find the node is O(height of tree)

- worst case when it is a leaf on the longest branch

To remove a node is O(height of tree)

- find the biggest item less than it, which is O(height of node)
- constant number of operations to replace

Note:

Implementing node removal in a BST is tricky.

Maintaining the parent references makes it easier to keep track of where you are in the tree, and of which nodes need to be updated.

But there are now more references to update, and more special cases to handle.

You will get this wrong when you first implement it, so careful and exhaustive testing is very important.

So – are we finished?

Complexity of maintaining and searching a ‘sorted’ collection of *n* items:

	array-based (Python) list	linked list	binary search tree of height h
searching	O(log n)	O(n)	O(h)
adding	O(n)	O(n)	O(h)
removing	O(n)	O(n)	O(h)

Is minimizing the work the right thing to do?

Our methods for adding and removing nodes in Binary Search Trees focused on doing as little work as possible (while maintaining the BST properties).

But if this increases the height of the tree, this might be storing up trouble, and creating more work to do when searching the tree (and so also for adding and removing)

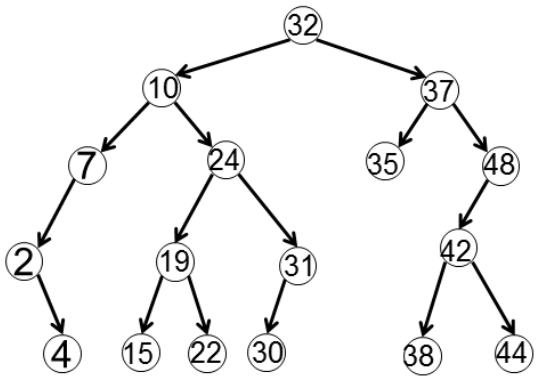
From Lecture 11 ...

Searching a binary search tree

Goal: if the element we are searching for is in the tree, return the node; else return None

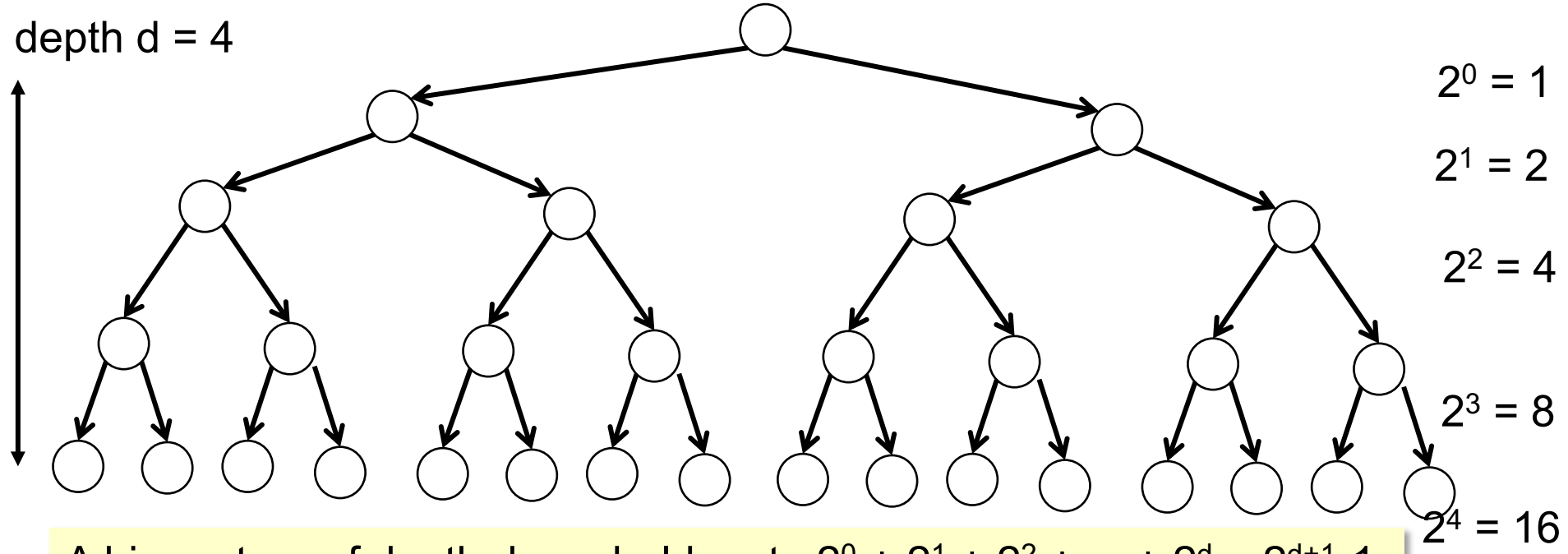
- all left descendants of a node have values less than the node's value
- all right descendants of a node have values greater than the node's value

```
search(node, item):
  if node == None
    return None
  if node's element > item
     return search(leftchild, item)
  else if node's element < item
    return search(rightchild, item)
  else return node
```



If h is the height of the root,
then this is O(h)

Complete Binary Trees



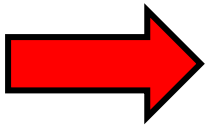
A binary tree of depth d can hold up to $2^0 + 2^1 + 2^2 + ... + 2^d = 2^{d+1}-1$
Or n items can be stored in a binary tree of depth = $\lfloor \log n \rfloor$

The *promise* of Binary Search Trees

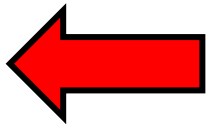
If h is the height of the root,
- search in a BST is O(h)
- adding items is O(h)
- removing items is O(h)

Height of the root is 'depth' of tree.
n items can be stored in a binary tree of depth = $\lfloor \log n \rfloor$

Binary search trees support search and updates with complexity O(log n)



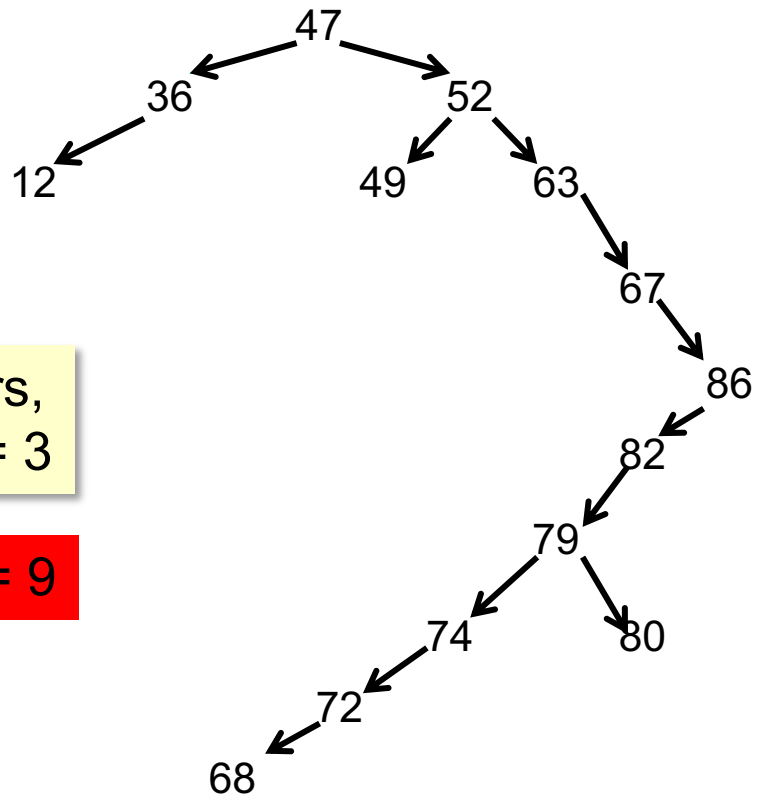
if we can ensure that the tree is close to being *complete*



Is there any indication that Binary Search Trees are close to complete?

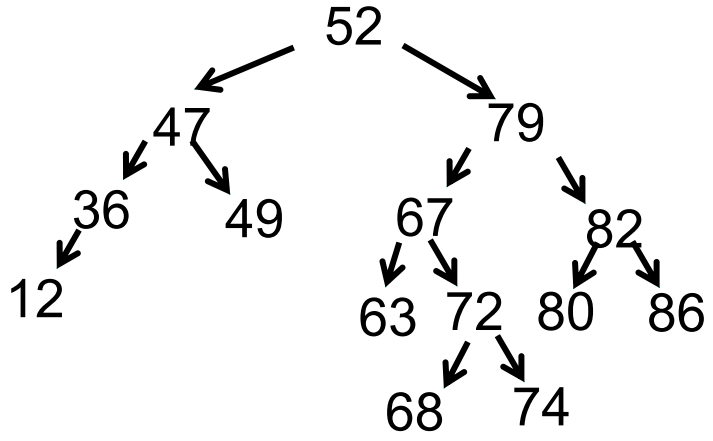
How complete are BSTs?

Add elements to a BST in the order: 47,52,36,63,49,67,86,82,79,80,12,74,72,68



14 numbers, so $\lfloor \log n \rfloor = 3$

but depth = 9



this BST has the same content, but its height is only 4

Next Lecture

re-balancing Binary Search Trees