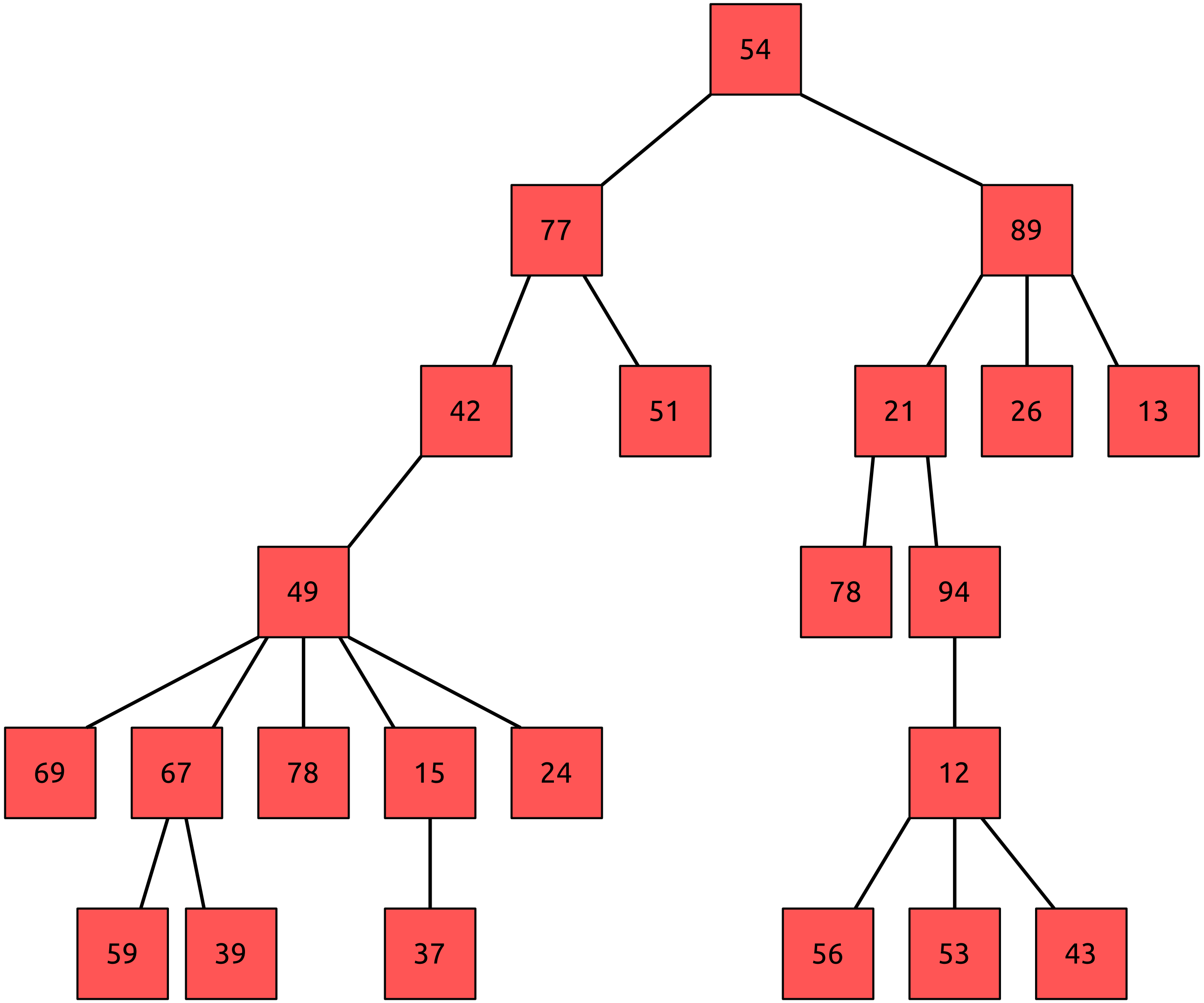
ECS 32B - Trees: Intro

Aaron Kaloti

UC Davis - Summer Session #2 2020

Tree

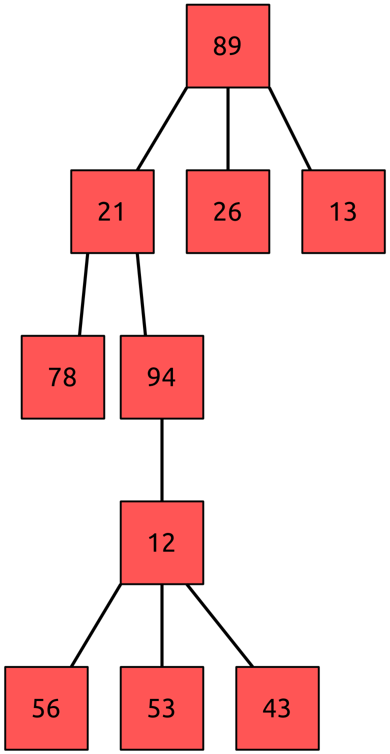
Example



Yes, trees are upside-down in the world of computer science. It makes more sense if you don't think about it.

Tree

Terminology



- Tree consists of **nodes/vertices** and **edges**.
  - Edges have implicit downward direction, e.g. edge from 89 to 13.
- **root**: node with no incoming edges, i.e. 89.
- **children** (of node c): the nodes to which c has an (implicit, downward) edge.
  - e.g. 21, 26, and 13 are children of 89.
  - e.g. 12 is a child of 94.
  - non-e.g. 78 is *not* a child of 13.
  - non-e.g. 53 is *not* a child of 94.
- **parent** (of node c): the node from which there is an edge to c.
  - e.g. 89 is the parent of 21, 26, and 13; 94 is a parent of 12.
  - non-e.g. 13 is *not* a parent of 78; non-e.g. 94 is *not* a parent of 53.
- **leaf**: has no children, e.g. 78, 43.
- **siblings**: nodes that have the same parent, e.g. 78 and 94.

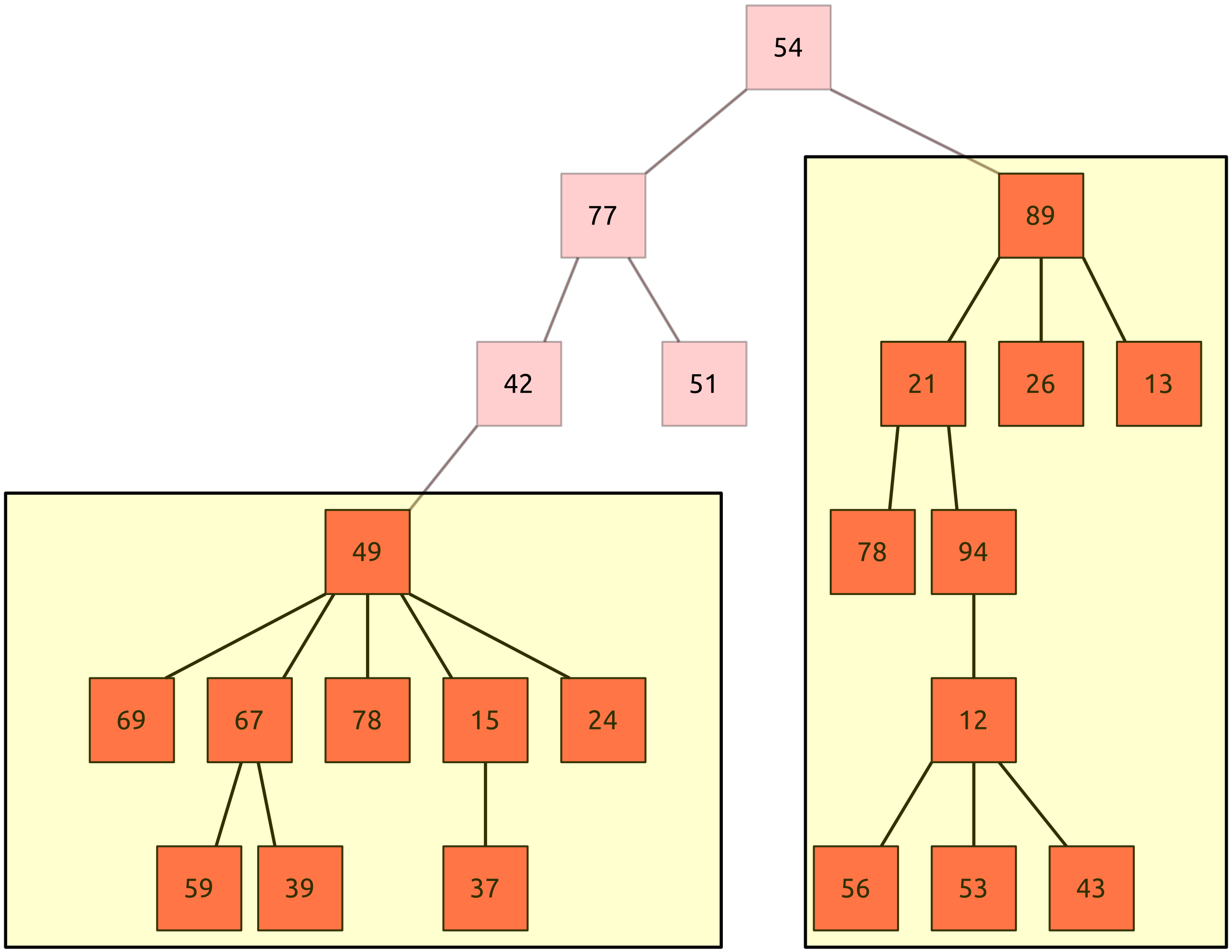
- **ancestor** (of c): a node b encountered on the path from the root to c. **proper ancestor** doesn't include itself.
  - e.g. 21 is an ancestor of 53.
  - e.g. 53 is an ancestor of itself but *not* a proper ancestor.
  - non-e.g. 26 is *not* an ancestor of 94.
- **descendant** (of b): a node c such that b is an ancestor of c. **proper descendant** doesn't include itself.
  - e.g. 53 is a descendant of 21.

Tree

Terminology

- **subtree**: parent and its descendants.

Examples



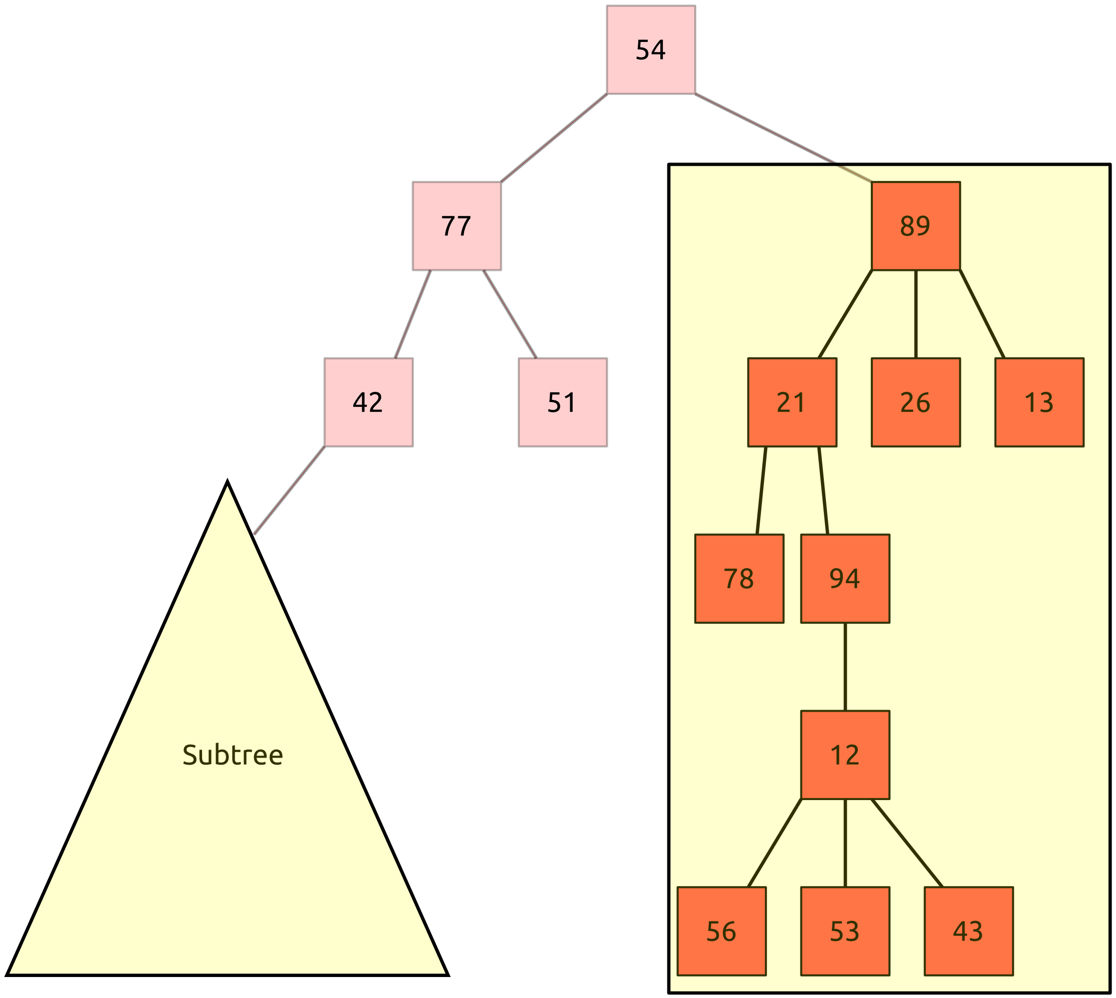
Note that the two pointed out subtrees are of course not the only subtrees.

Tree

Terminology

- **subtree**: parent and its descendants.

Example of Alternative View

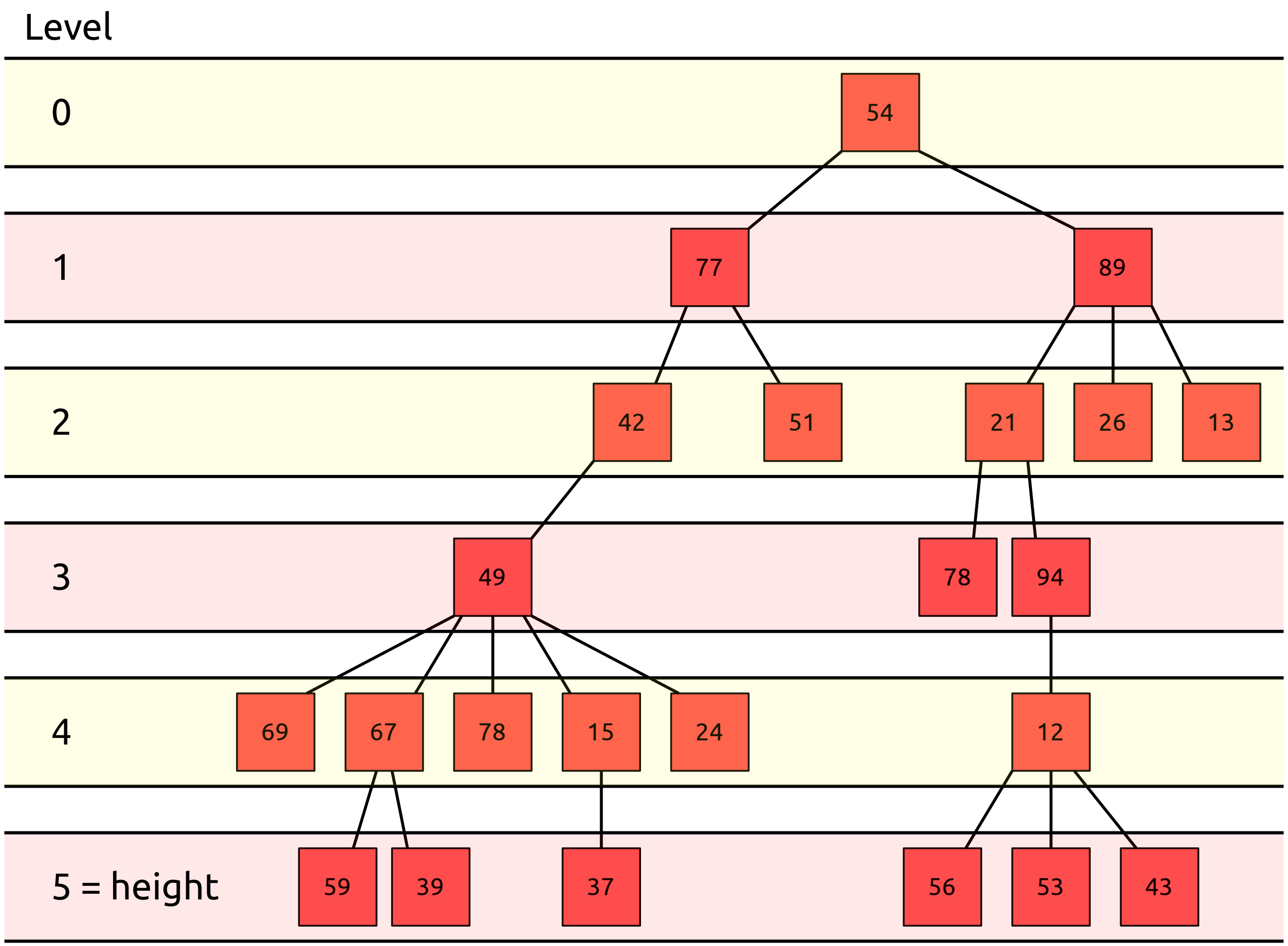


Tree

Terminology

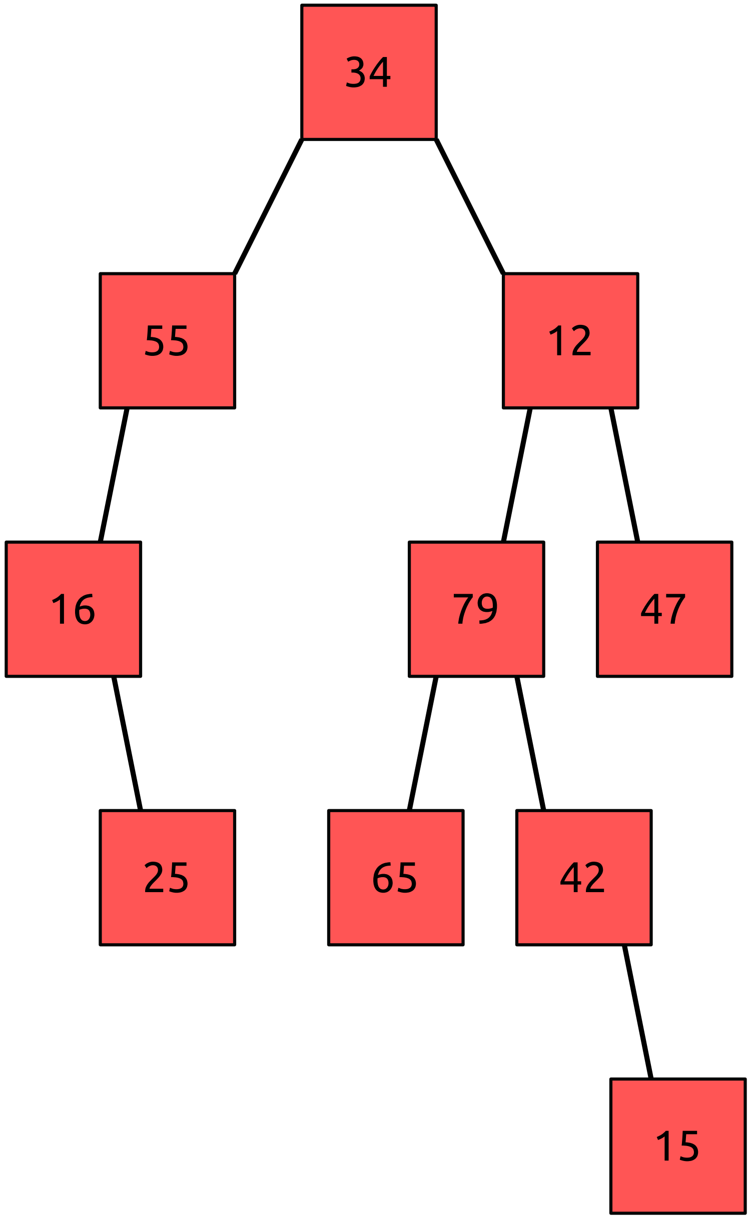
- **level/depth** (of a node): number of edges on path from root to that node.
- **height** (of a tree): maximum level of any node in the tree.

Example



Binary Tree

- **binary tree**: each node has at most two children.

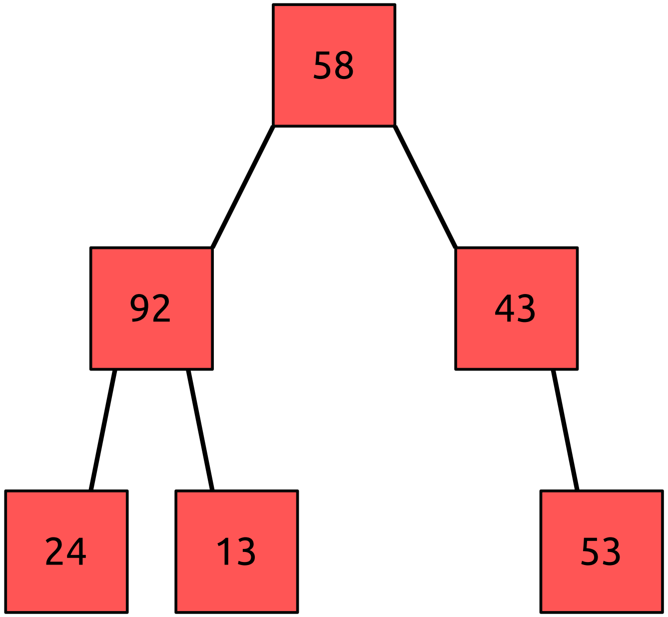


Binary Tree

Implementation - List of Lists of Lists of Lists of ... (Recursive)

- Represent tree with a list of three elements:
    1. Root.
    2. Left subtree.
    3. Right subtree.

Example #1

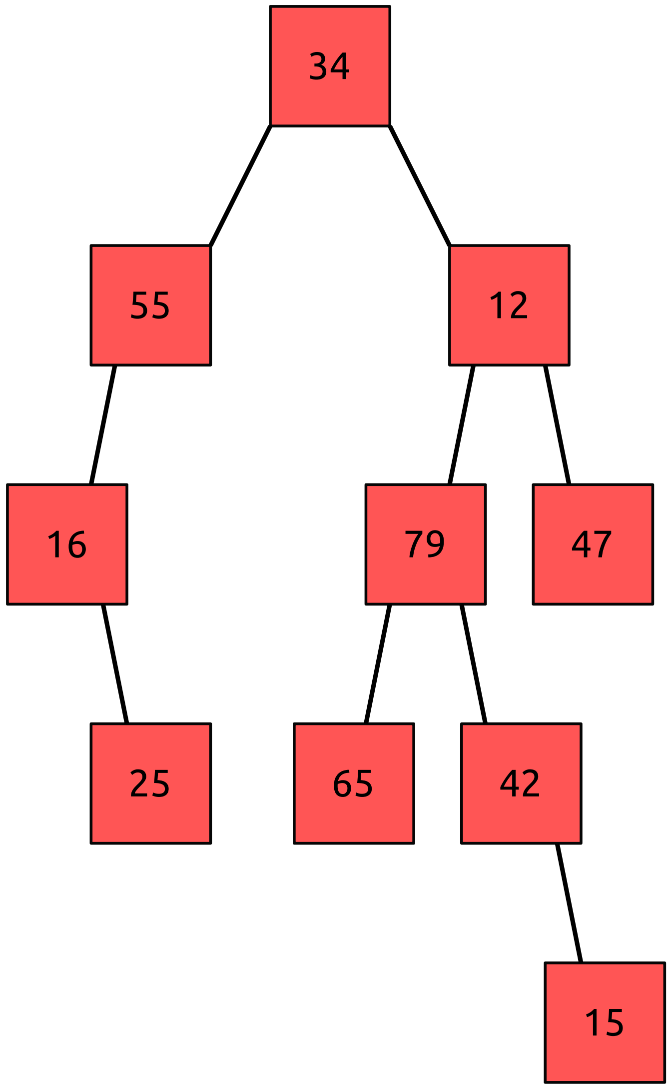



```
[58,
  [92,  # left subtree
    [24, [], []],
    [13, [], []]
  ],
  [43,  # right subtree
    [],
    [53, [], []]
  ]
]
```

Binary Tree

Implementation - List of Lists of Lists of Lists of ... (Recursive)

Example #2




```
[34,
  [55,  # left subtree
    [16,
      [],
      [25, [], []]
    ],
    []
  ],
  [12,  # right subtree
    [79,
      [65, [], []],
      [42,
        [],
        [15, [], []]
      ]
    ],
    [47, [], []]
  ]
]
```

Binary Tree

Implementation - List of Lists of Lists of Lists of ... (Recursive)

Other Operations

- You should read the rest of section 7.4 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.

Binary Tree

Implementation - Nodes and References (Recursive *and* Object-Oriented)

- Each node has a reference to its left/right subtrees.

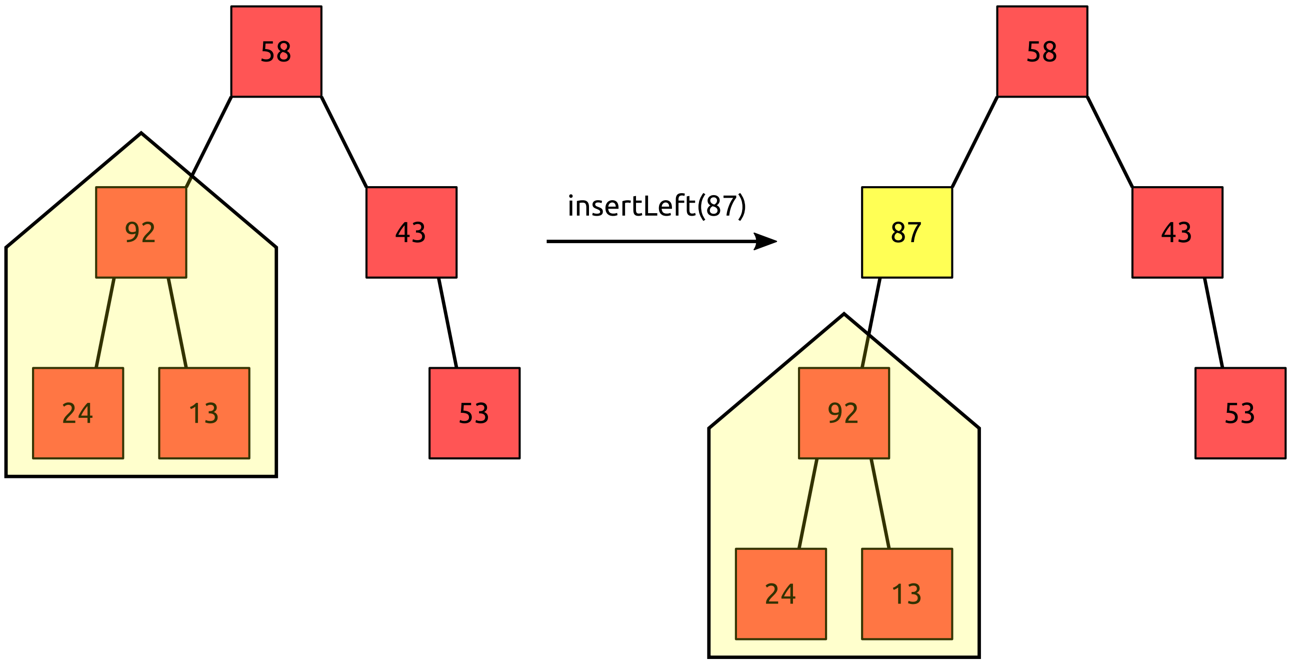
```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
```

Binary Tree

Implementation - Nodes and References (Recursive *and* Object-Oriented)

`insertLeft(newNode)`

```
def insertLeft(self,newNode):
    if self.leftChild == None:
        self.leftChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.leftChild = self.leftChild
        self.leftChild = t
```



Binary Tree

Implementation - Nodes and References (Recursive *and* Object-Oriented)

Entire Implementation

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self,newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self,newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t

    ...
```

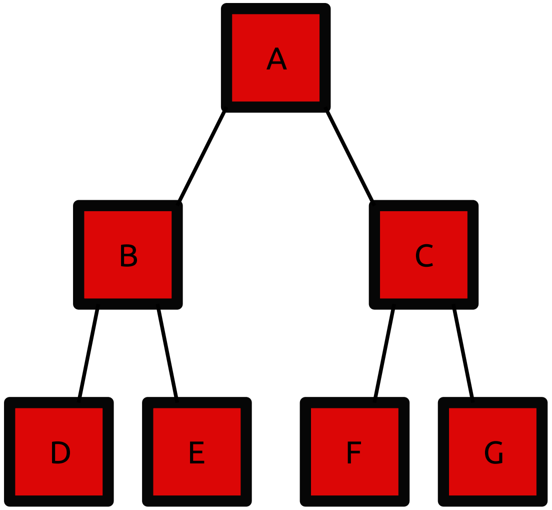
```
    ...
    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self,obj):
        self.key = obj

    def getRootVal(self):
        return self.key
```

Binary Tree Traversals

- Common patters for traversing all nodes of tree.
- When applicable, can process right subtree before left, if consistent about it, e.g. preorder can be PRL.

Preorder Traversal

- PLR: parent, left, right.
- In the e.g.: A, B, D, E, C, F, G.



Inorder Traversal

- LPR: left, parent, right.
- In the e.g.: D, B, E, A, F, C, G.

Postorder Traversal

- LRP: left, right, parent.
- In the e.g.: D, E, B, F, G, C, A.

Binary Tree Traversals

Implementation[1]

Preorder Traversal (PLR)

```
def preorder(tree):
    if tree:
        print(tree.getRootVal())
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())
```

- Same tree class:

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None

    ...
```

Inorder Traversal (LPR)

```
def inorder(tree):
    if tree != None:
        inorder(tree.getLeftChild())
        print(tree.getRootVal())
        inorder(tree.getRightChild())
```

Postorder Traversal (LRP)

```
def postorder(tree):
    if tree != None:
        postorder(tree.getLeftChild())
        postorder(tree.getRightChild())
        print(tree.getRootVal())
```

1. These impementations are all from section 7.7 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum. This reference also provides versions of the below functions that are methods of the `BinaryTree` class.

Binary Tree Traversals

Runtime Analysis

- A tree is a specific kind of graph (we'll get to graphs eventually...). Graph algorithms usually scale based on two input variables, n and m.
  - $n \Rightarrow$ number of nodes; $m \Rightarrow$ number of edges.
- Doubling n means doubling number of nodes to traverse.
- With trees, $n = m + 1$.
- The tree traversals all take $O(n)$ time.

Binary Search Tree

Definition

- **binary search tree**: binary tree in which nodes in left subtree are less than root; nodes in right subtree are greater.

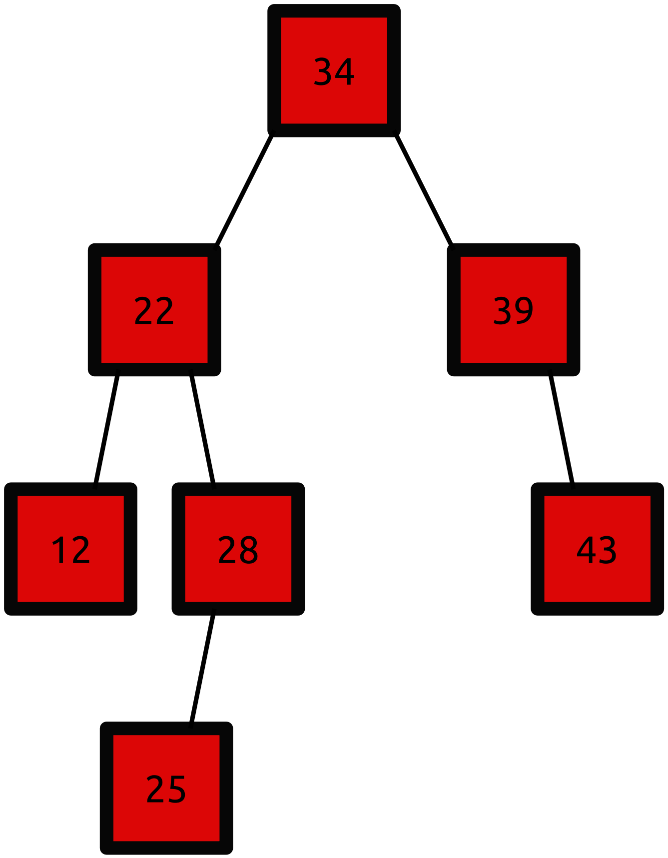
Interview Tip

- If an interview problem asks about a "binary tree", don't assume they mean "binary search tree"; might lead to wrong assumptions in your approach.

Note on Essential Operations

- Once we study self-balancing binary search trees, we will primarily only care about the following generic operations:
  - Find.
  - Insert.
  - Delete.

Example

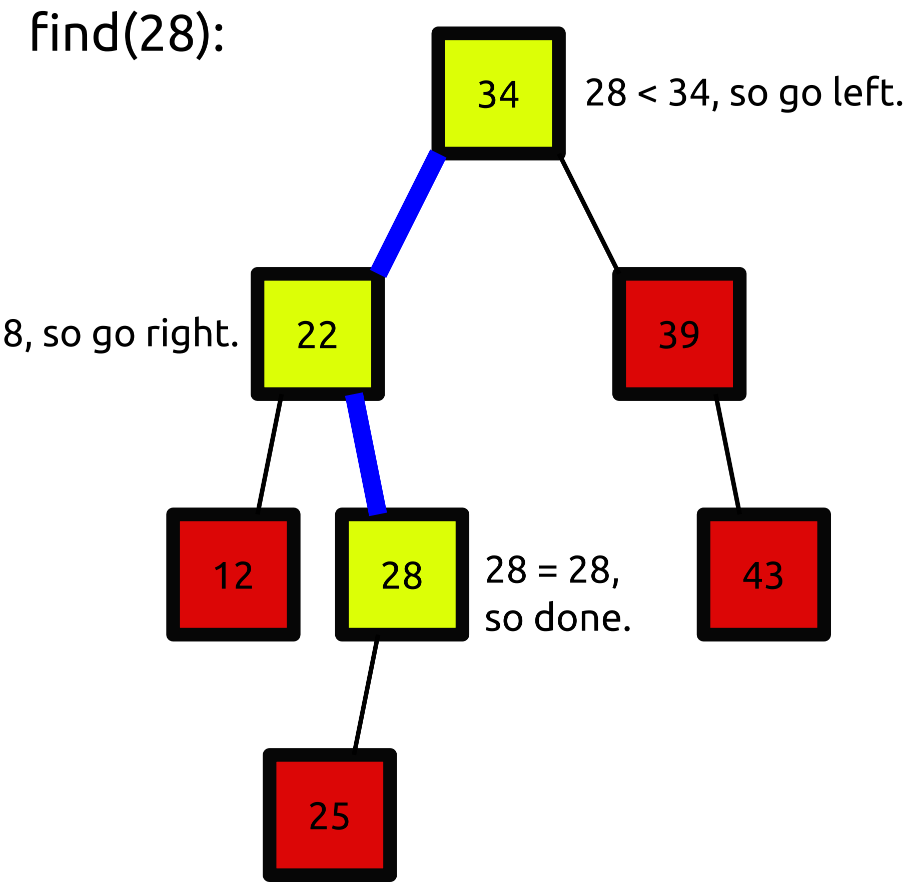


Binary Search Tree

Find

- A binary search tree is like a binary search as a data structure.

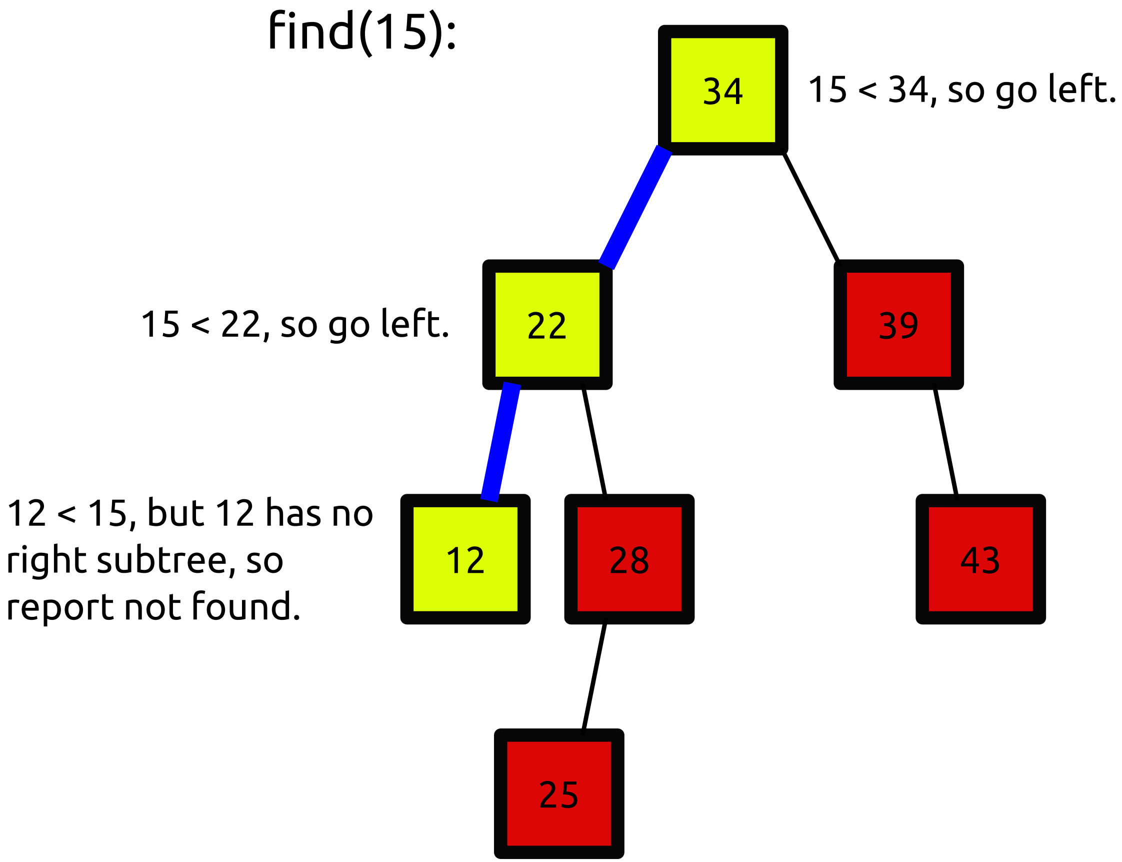
Example



Binary Search Tree

Find

Example: Not Found

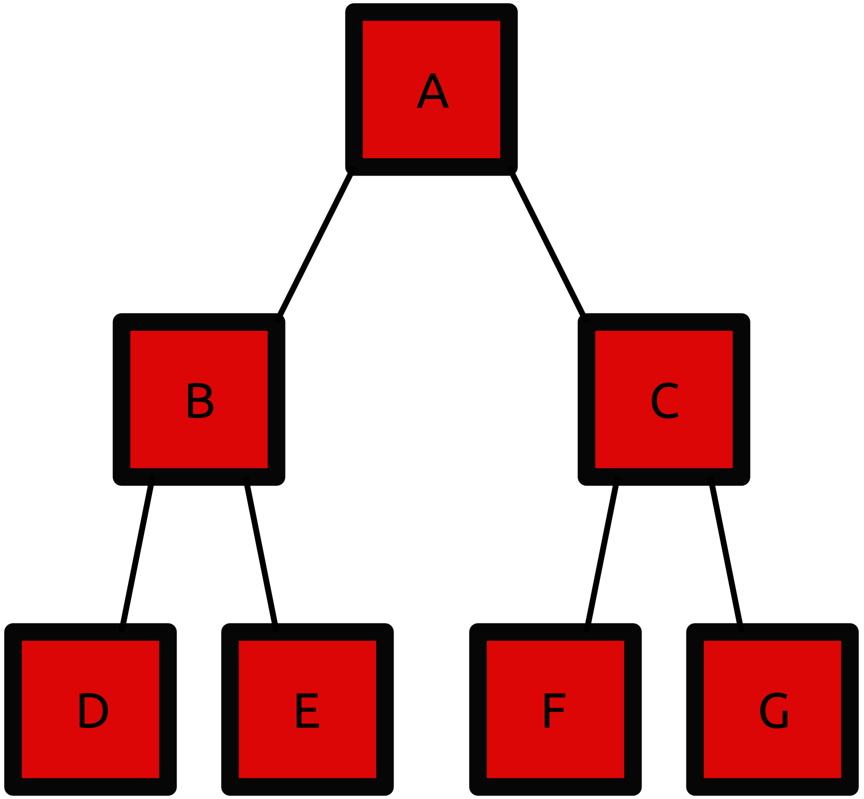


Binary Search Tree

Find

Worst-Case Runtime Analysis

- Is a tree shaped this the worst-case?



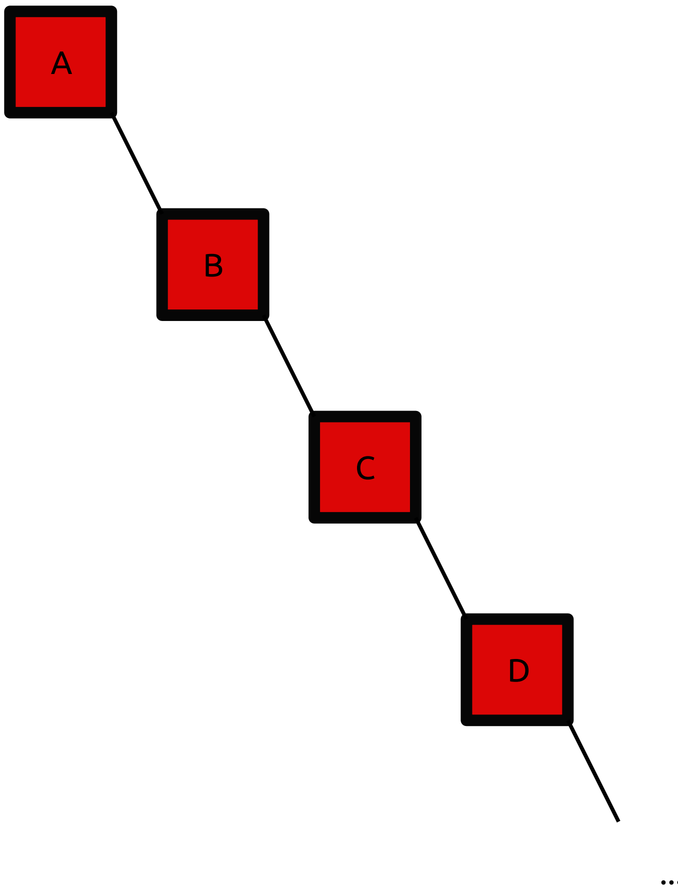
- Would mean worst-case find takes $\Theta(\lg n)$ time.

Binary Search Tree

Find

Worst-Case Runtime Analysis

- No, a tree shaped like this is...

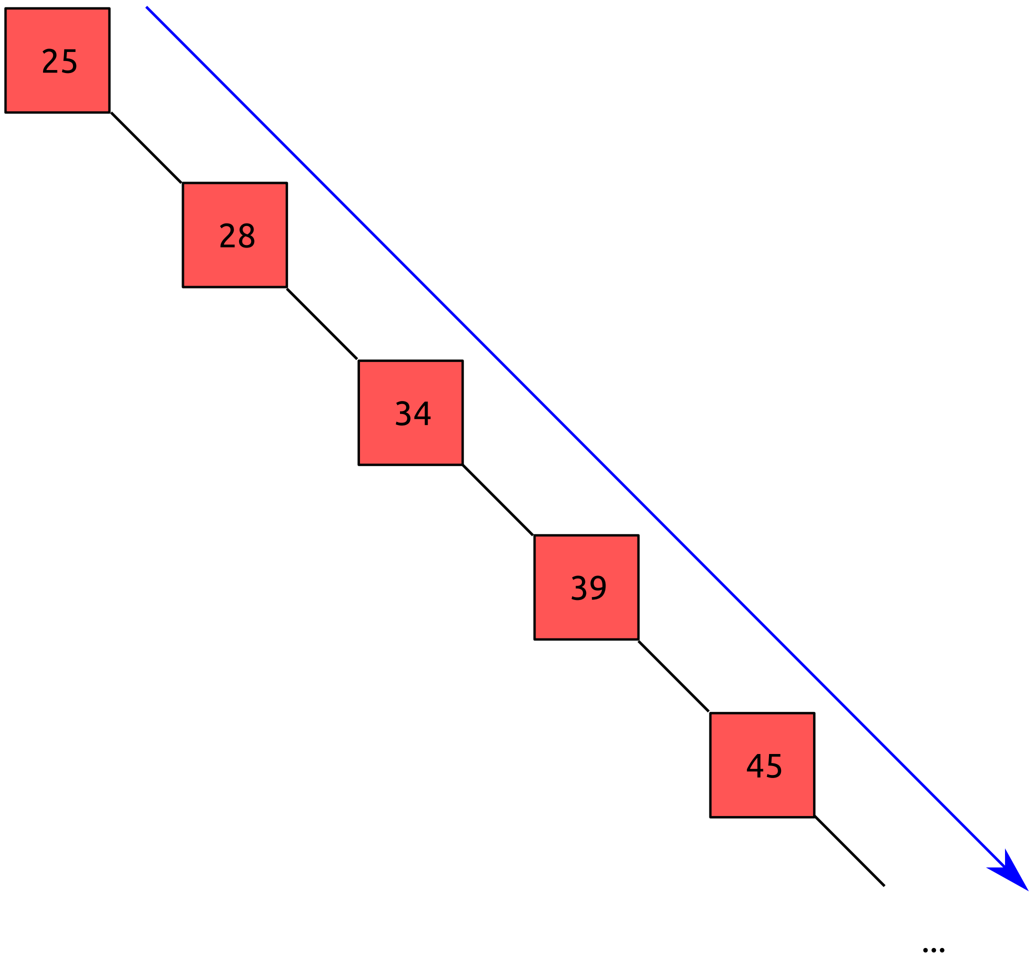


- **Worst-case find** takes $\Theta(n)$ time.
- How could this happen?

Binary Search Tree

Find

Worst-Case Runtime Analysis: Example



- Find element greater than greatest element in tree.

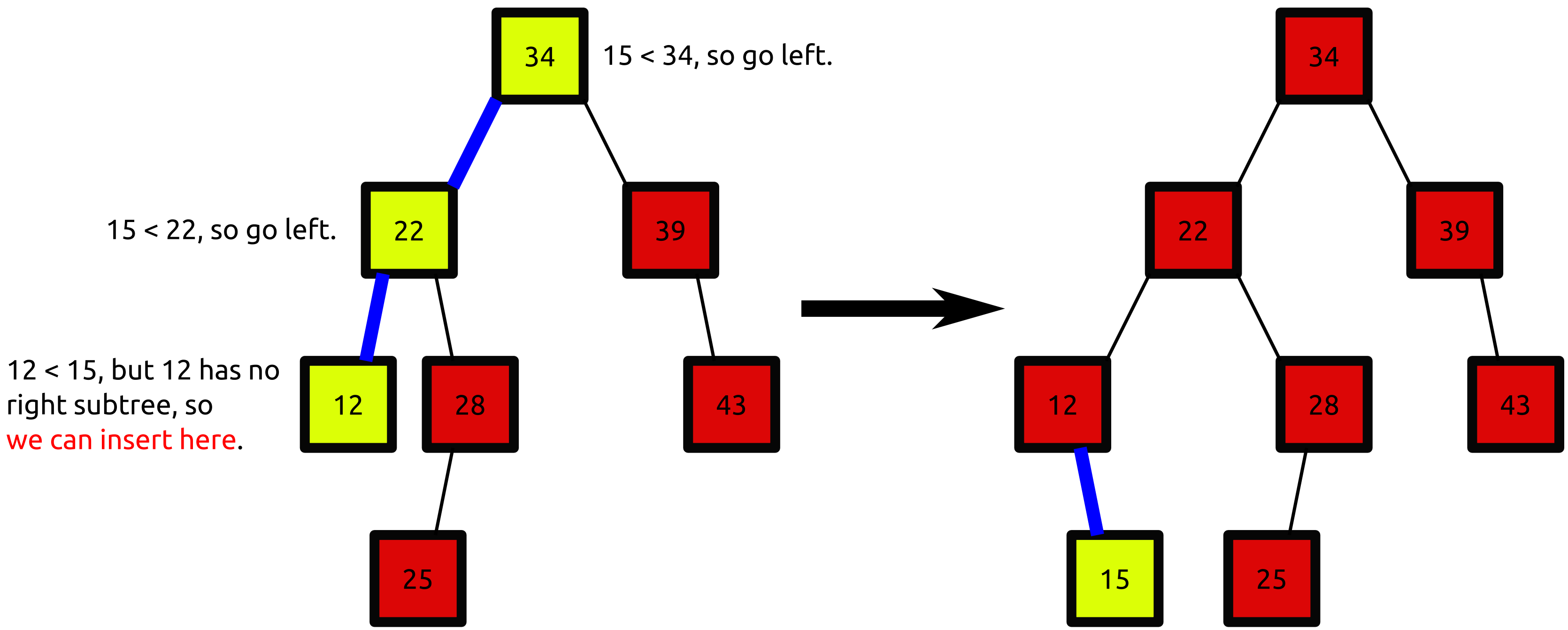
Binary Search Tree

Insert

- We'll assume no duplicate elements
- Steps:
  1. Perform a find to determine location for new element.
  2. Create new node/link.

Example #1

insert(15):

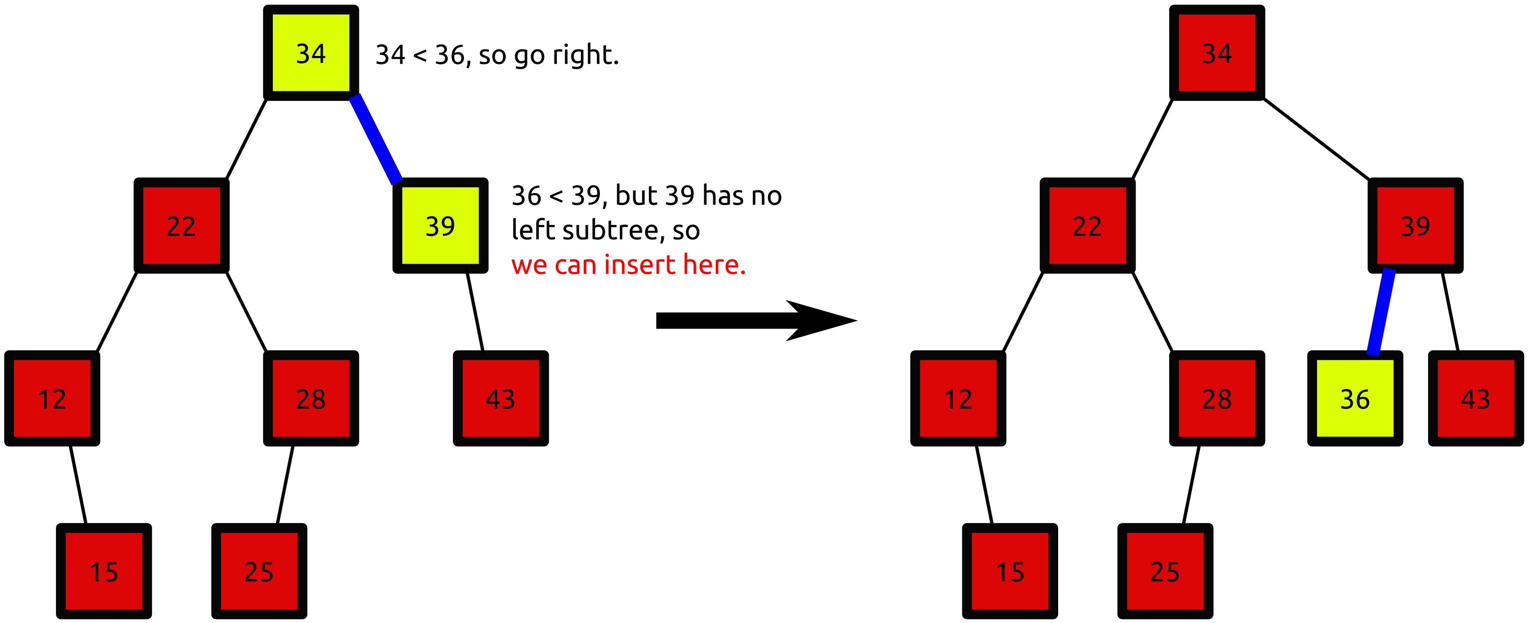


Binary Search Tree

Insert

Example #2

insert(36):

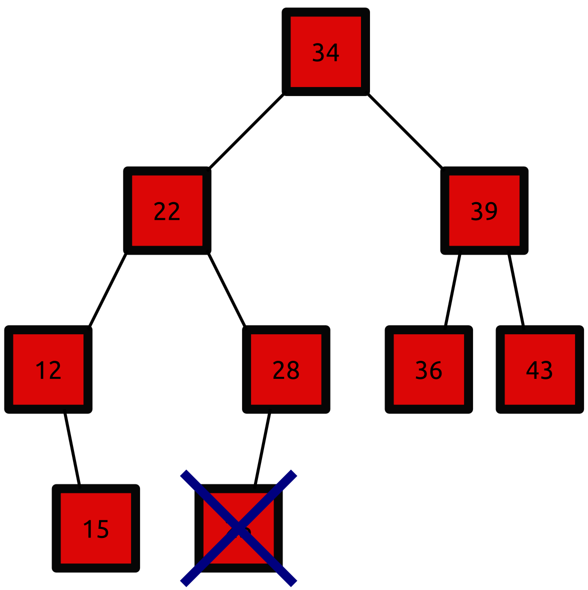


Binary Search Tree

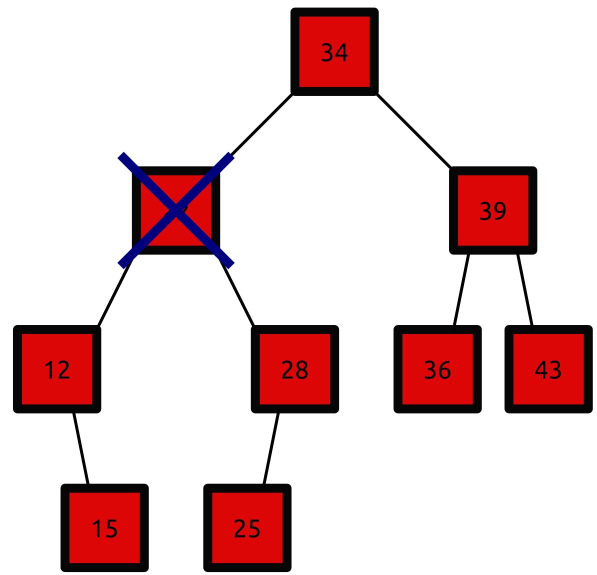
Delete

- Steps:
  1. Find the target element.
  2. Remove it.
     - If target is a leaf, we're done.
     - Otherwise, must do a bit more work.

delete(25): easy



delete(22): leaves a blank spot...



- *Solution*: replace deleted element with a proper descendant. Need to pick element adjacent to deleted one in the sorted order of the values.

Binary Search Tree

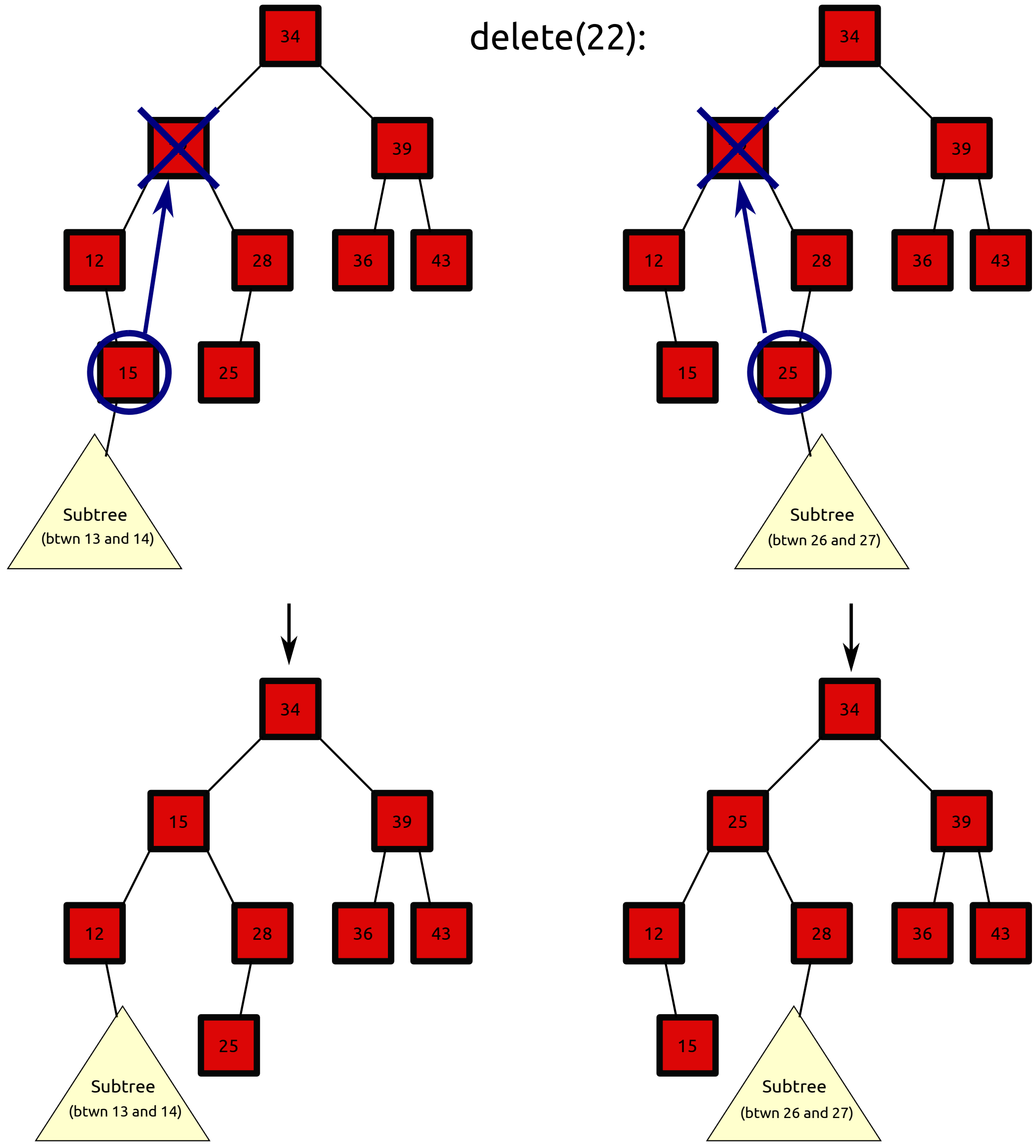
Delete

- Two choices for replacement:
    1. Max of left subtree.
        - That max's left subtree is moved up to max's old spot.
    2. Min of right subtree.
        - That min's right subtree is moved up to min's old spot.

Binary Search Tree

Delete

Examples

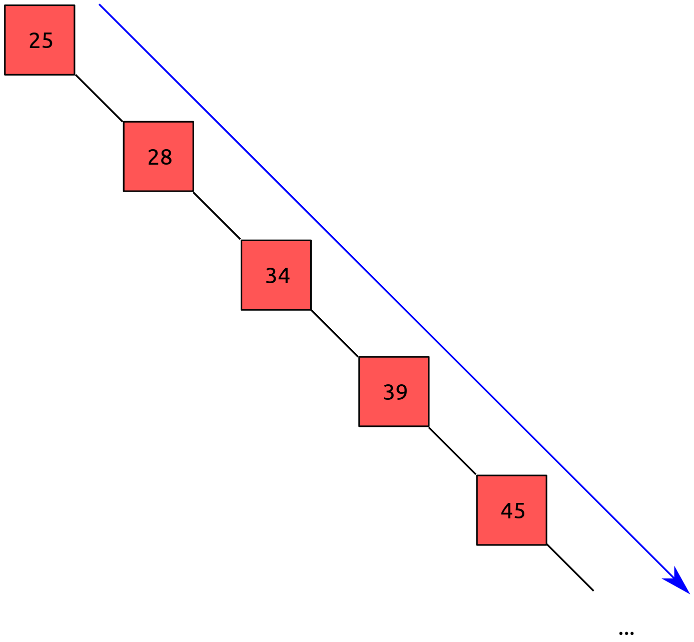


Binary Search Tree

Recap: Worst-Case Runtimes

Operation	**Worst-Case Time Complexity**
Find	$\Theta(n)$
Insert	$\Theta(n)$
Delete	$\Theta(n)$

- Need to find some way to fix this...



References / Further Reading

- **Primary reference**: Chapter 7 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.
- Chapter 4 of *Data Structures and Algorithm Analysis in C*++ by Mark Allen Weiss (Fourth Edition).