Algorithms and Data Structures for Data Science

Binary Search Trees 2

CS 277

March 27, 2023

Brad Solomon

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Exam 2 Observations

NOTE: Some students haven't taken exam 2 yet!

Average: 76% **Std Dev:** 14%

Exam grades are fairly normalized around mean

Remember: You get one free retake with the final!

Come to office hours to go over exam

Learning Objectives

Review binary search trees (find)

Conceptualize and implement BST insert and remove

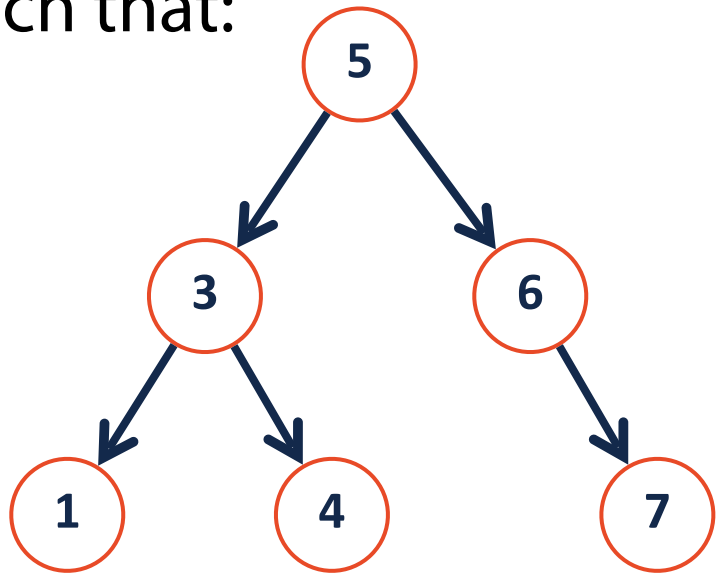
Discuss applications of BSTs

Binary Search Tree

A **BST** is a binary tree $T = treeNode(val, T_L, T_r)$ such that:

$$\forall n \in T_L,\ n.val < T.val$$

$$\forall n \in T_R,\ n.val > T.val$$

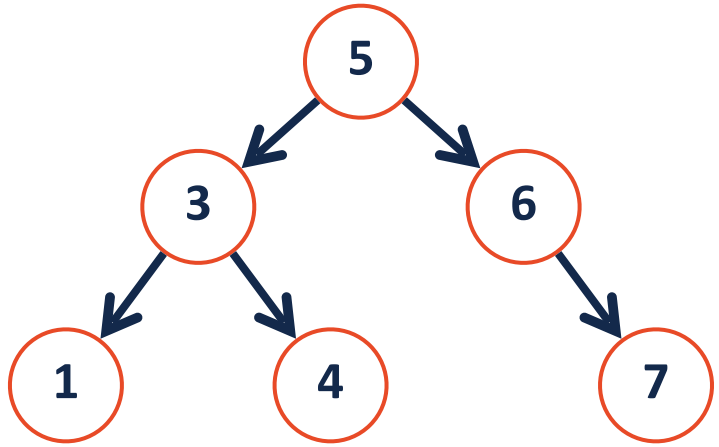



```
1  class bstNode:
2      def __init__(self, key, val, left=None, right=None):
3          self.key = key
4          self.val = val
5          self.left = left
6          self.right = right
```

Key	5	3	6	7	1	4
Value	A	B	C	D	E	F

BST Find

Base Case:

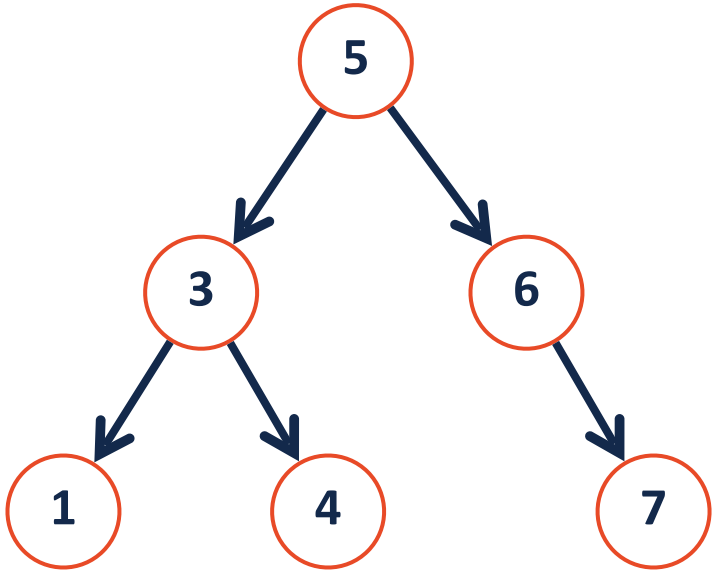


Recursive Step:

Combining:

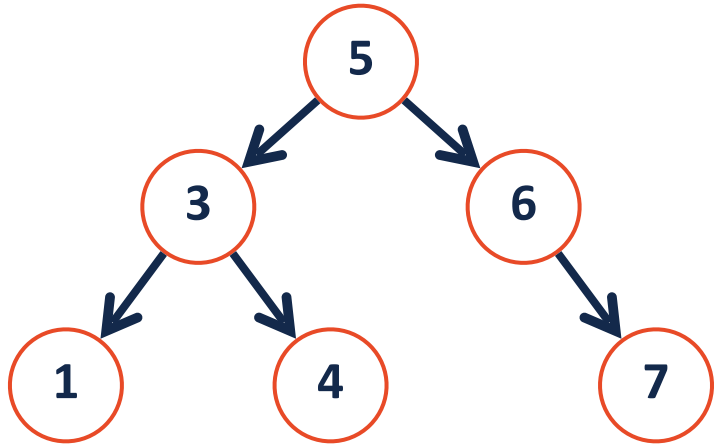
BST Find

```
1  def find(root, key):
2      return find_helper(root, key).val
3
4  def find_helper(node,key):
5      # Base Case (Tree is empty)
6      if not node:
7          return None
8
9      # Found match
10     if node.key == key:
11         return node
12
13     # Recurse to either left or right
14     if node.key > key:
15         return find_helper(node.left, key)
16
17     if node.key < key:
18         return find_helper(node.right, key)
19
20
21
22
23
```



BST Insert

Base Case:

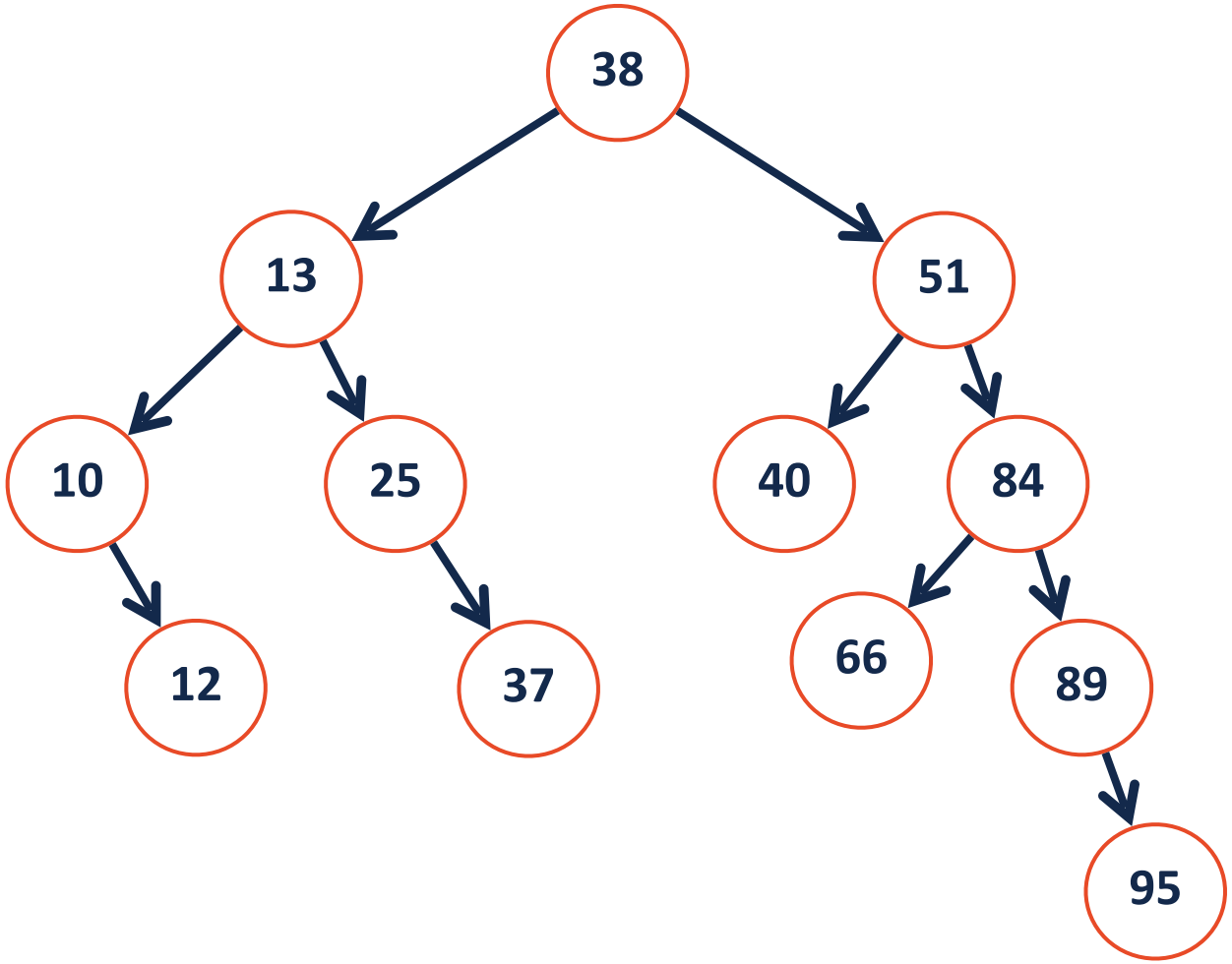


Recursive Step:

Combining:

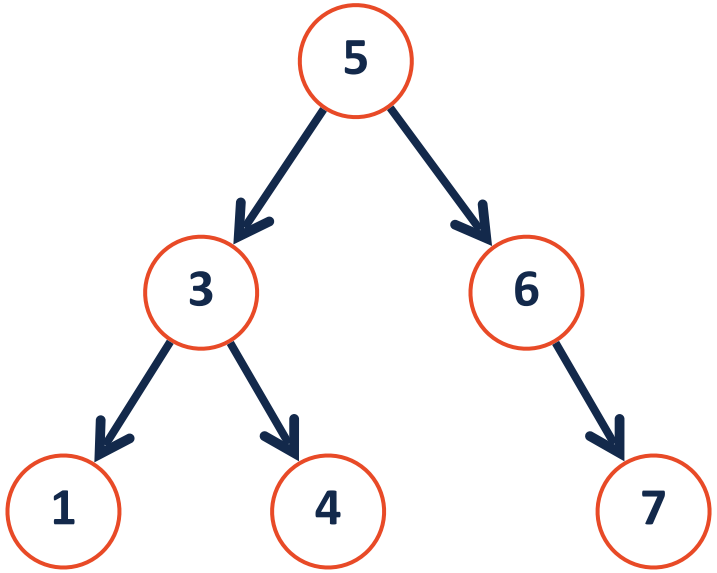
BST Insert

`insert(33)`



BST Insert

```
1  def insert(root, key, value):
2      if root == None:
3          root = bstNode(key,value)
4      else:
5          insert_helper(root, key, value)
6      return root
7
8  def insert_helper(node, key, value):
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

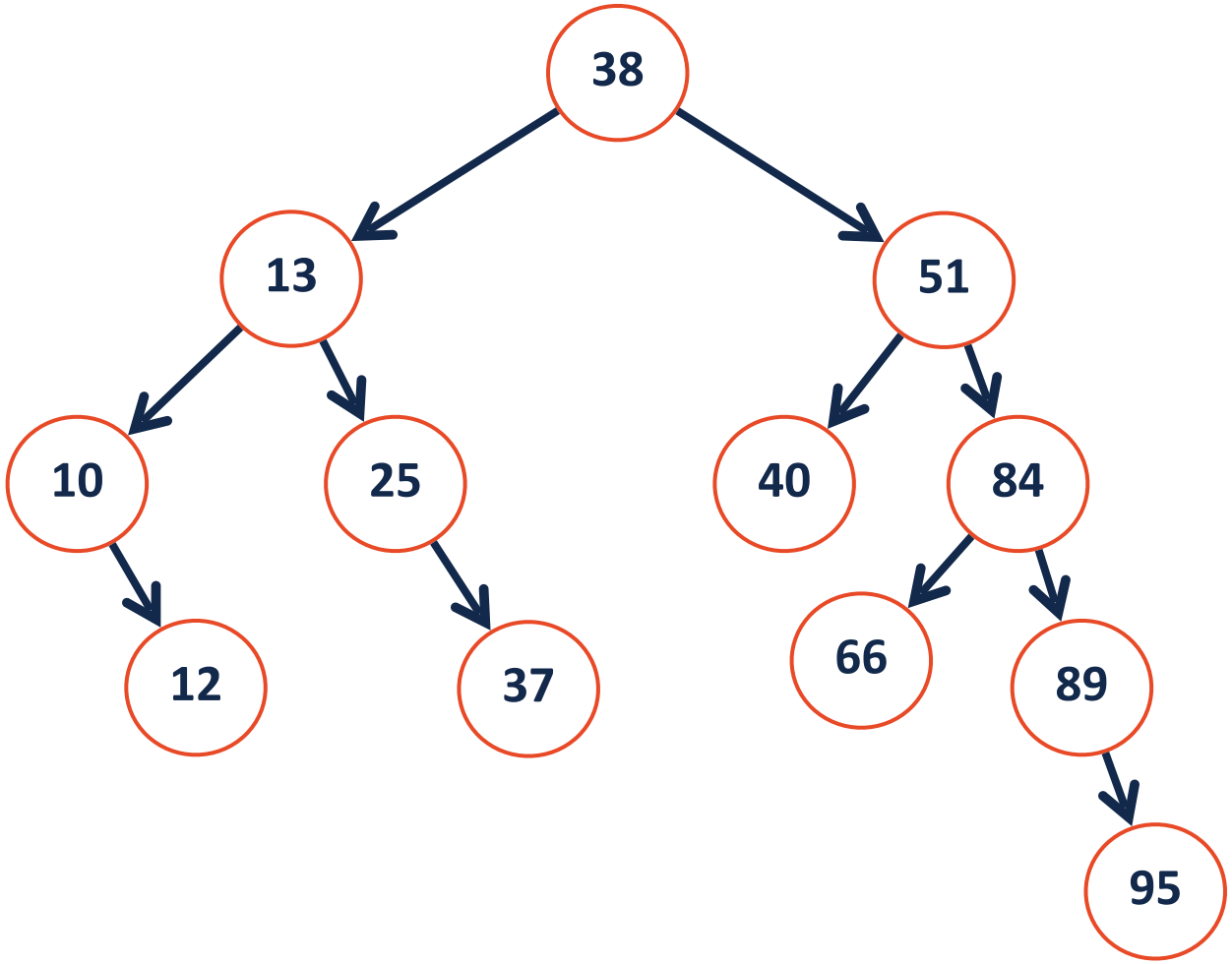


BST Insert

What tree would be formed by inserting the following sequence of integers: `[3, 7, 2, 1, 4, 8, 0]`?

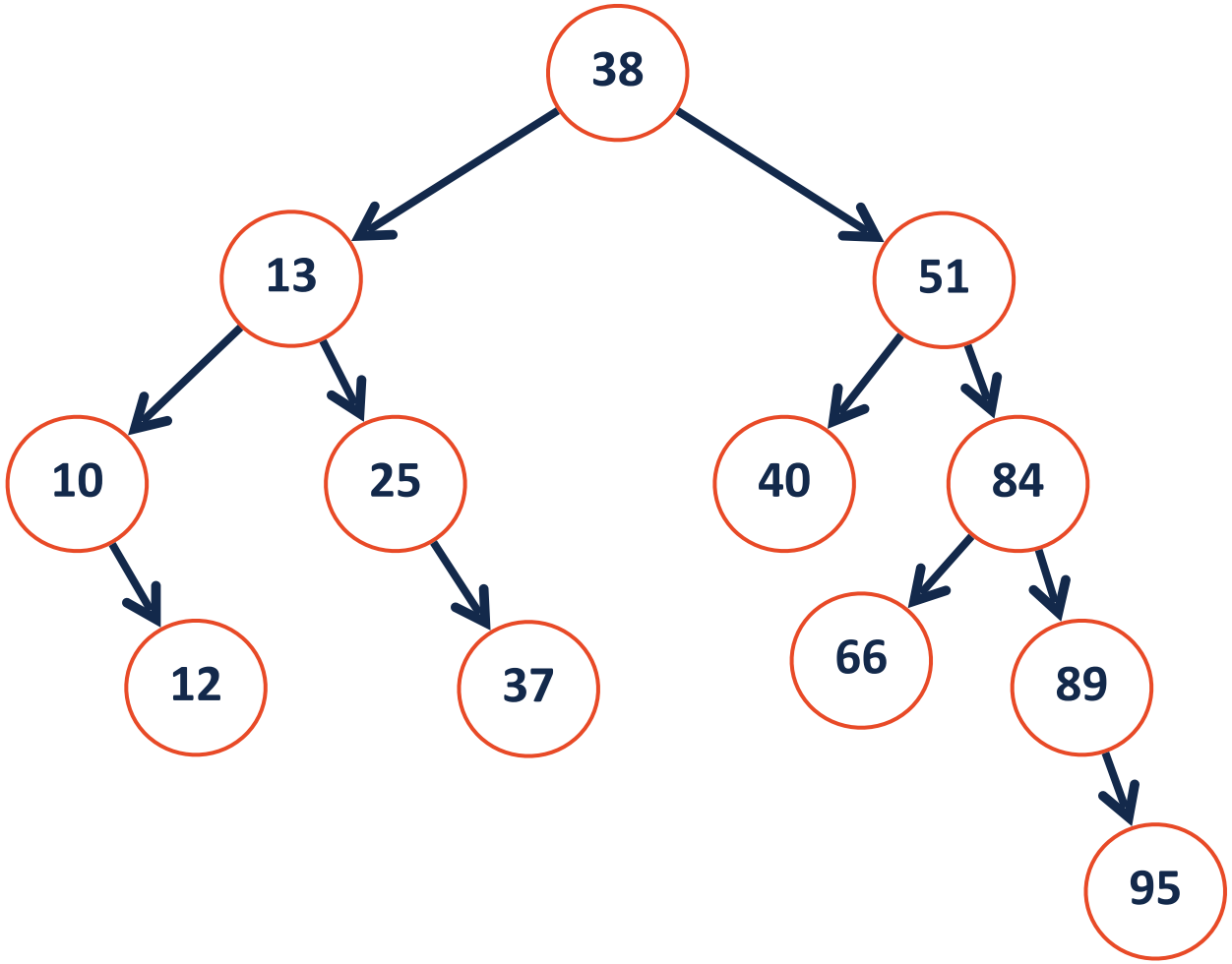
BST Remove

`remove(40)`



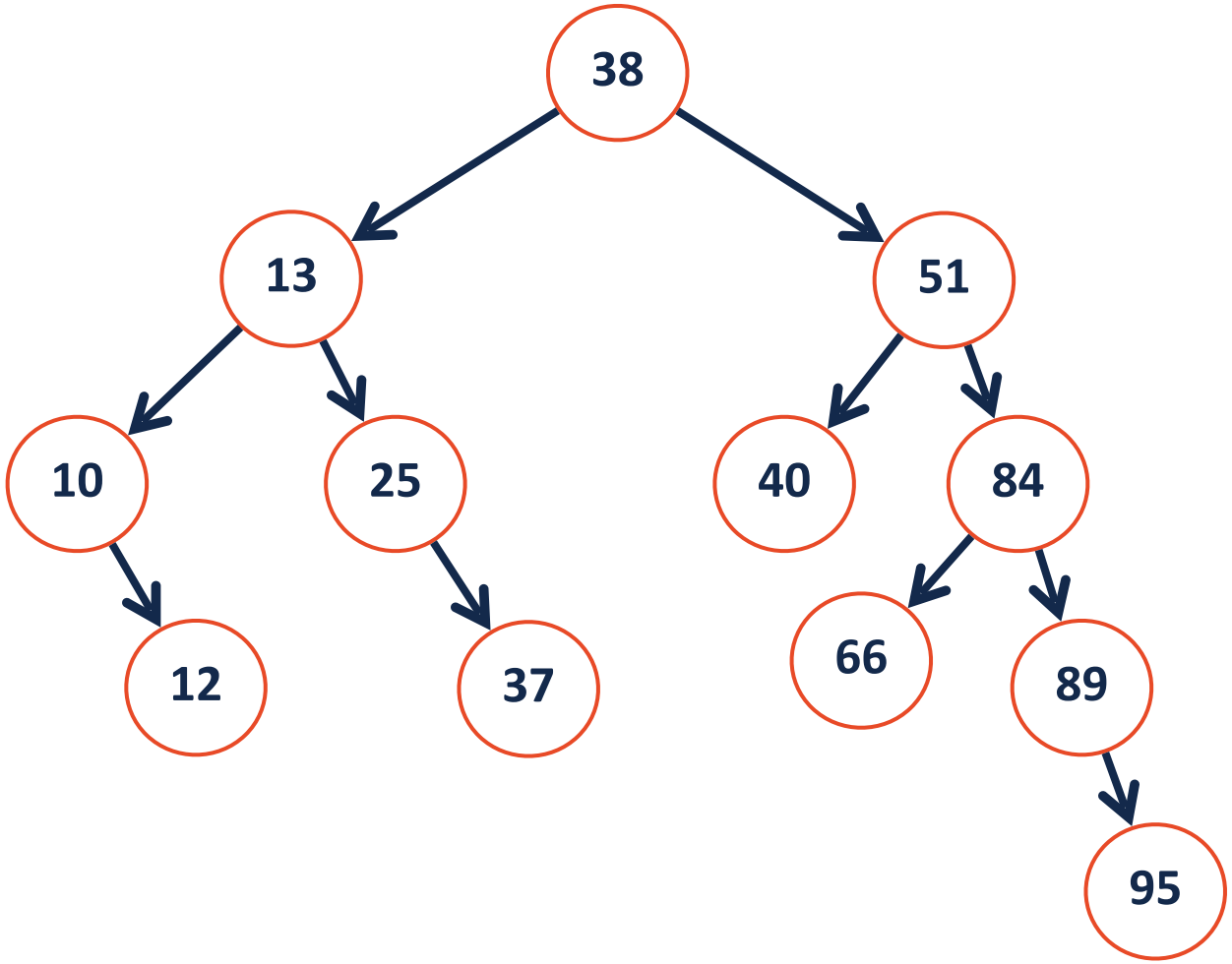
BST Remove

`remove(25)`



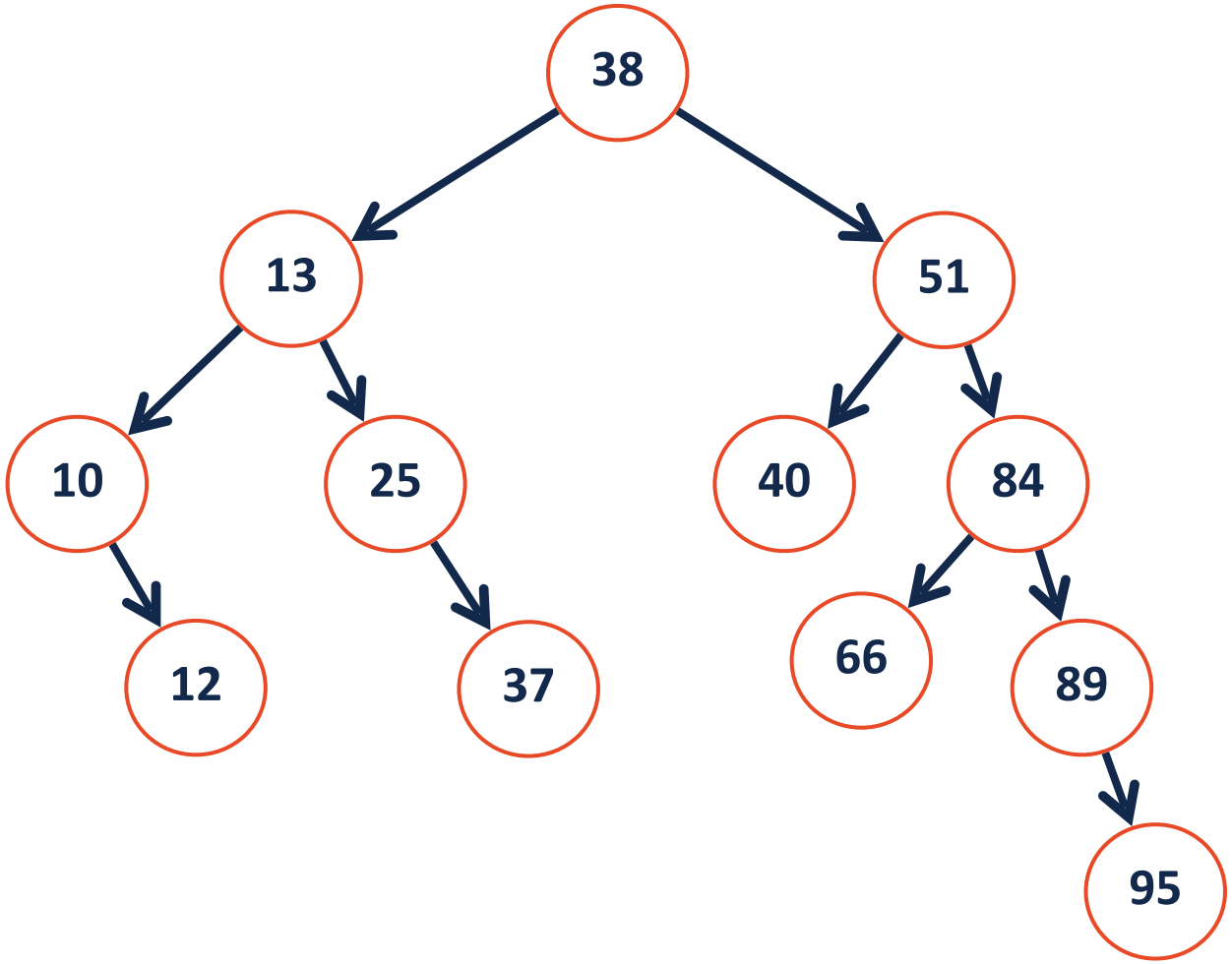
BST Remove

`remove(13)`



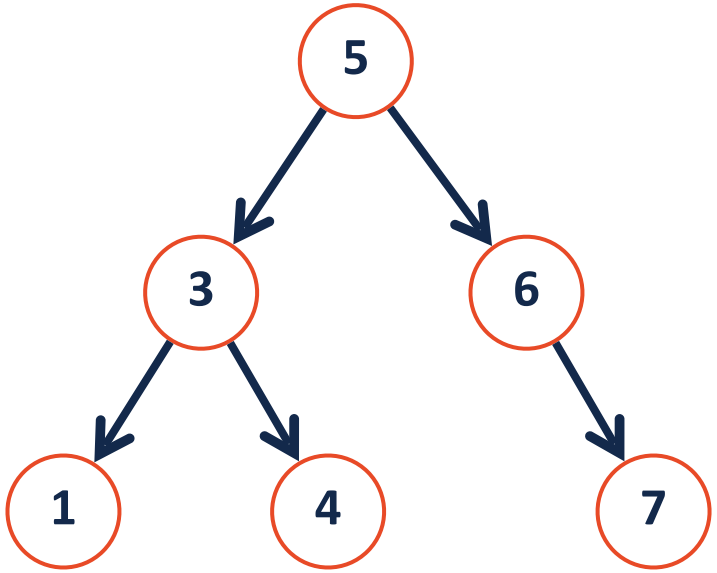
BST Remove

`remove(51)`



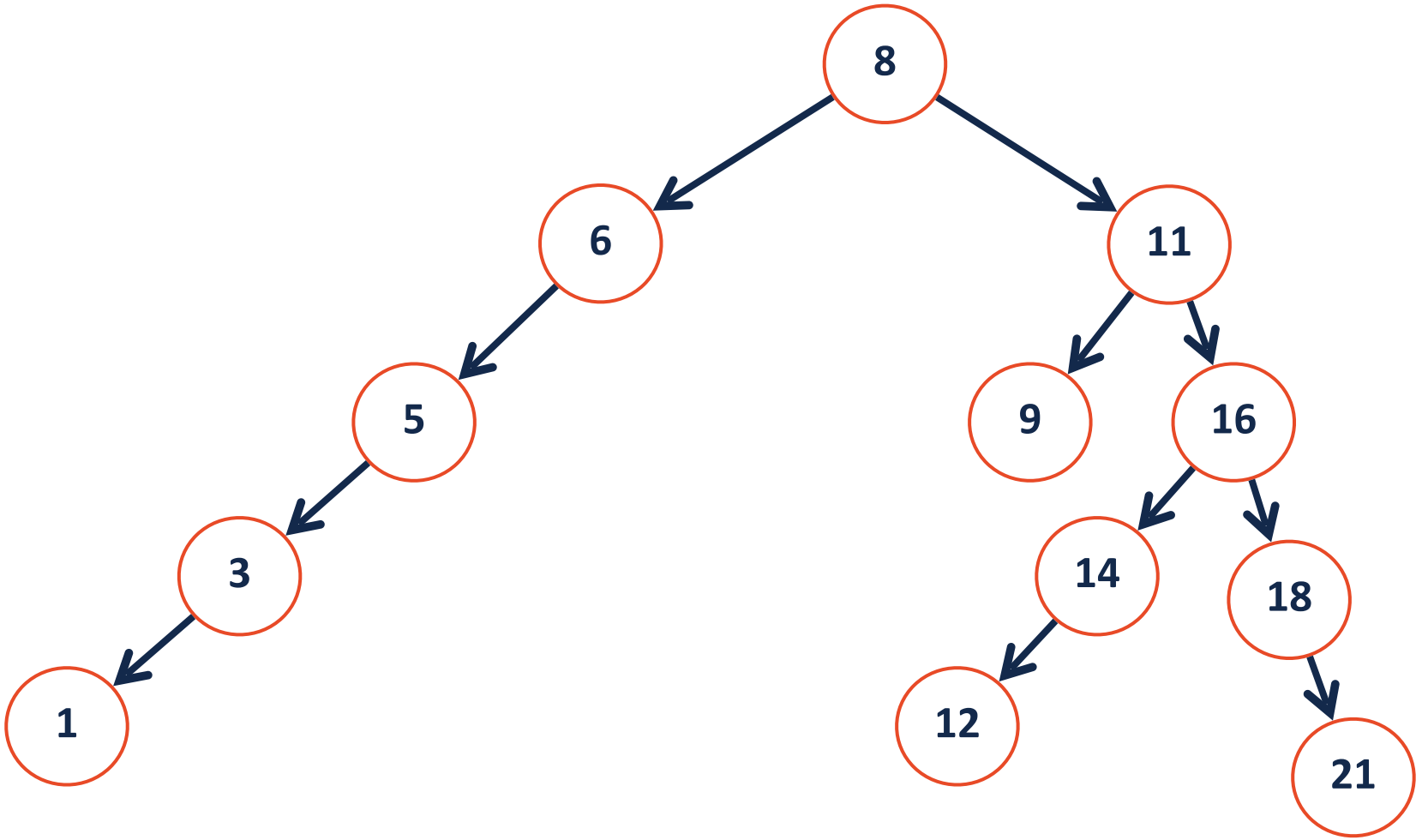
BST Remove

```
1  def remove(root, key):
2      root = remove_helper(root, key)
3      return root
4
5  def remove_helper(node, key):
6
7
8
9
10
11
12
13
14
15
16
17
18
19 def findIOP(node):
20     pass
21
22 def findIOS(node):
23     pass
```



BST Remove

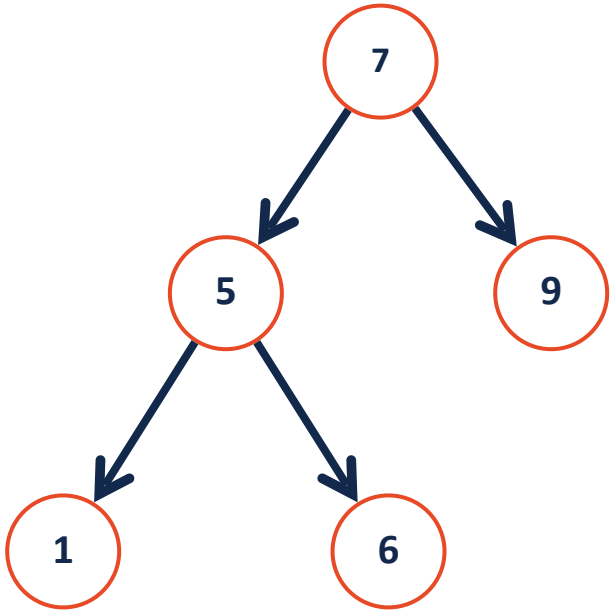
What will the tree structure look like if we remove node 16 using IOS?

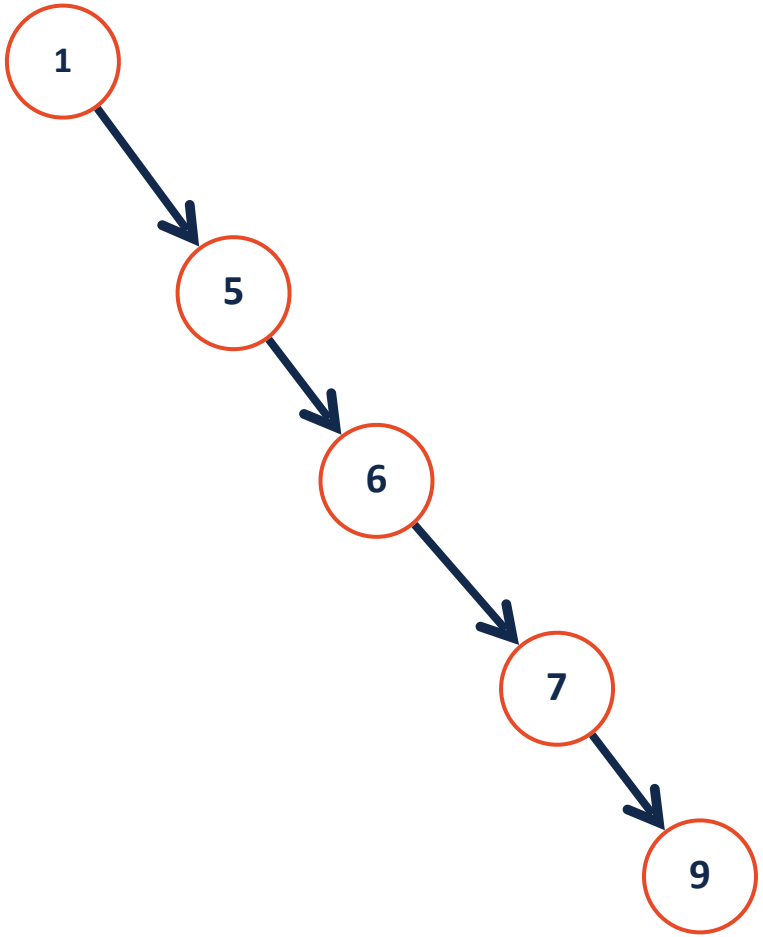


BST Analysis – Running Time

Operation	BST Worst Case
find	
insert	
delete	
traverse	

Limiting the height of a tree





Option A: Correcting bad insert order

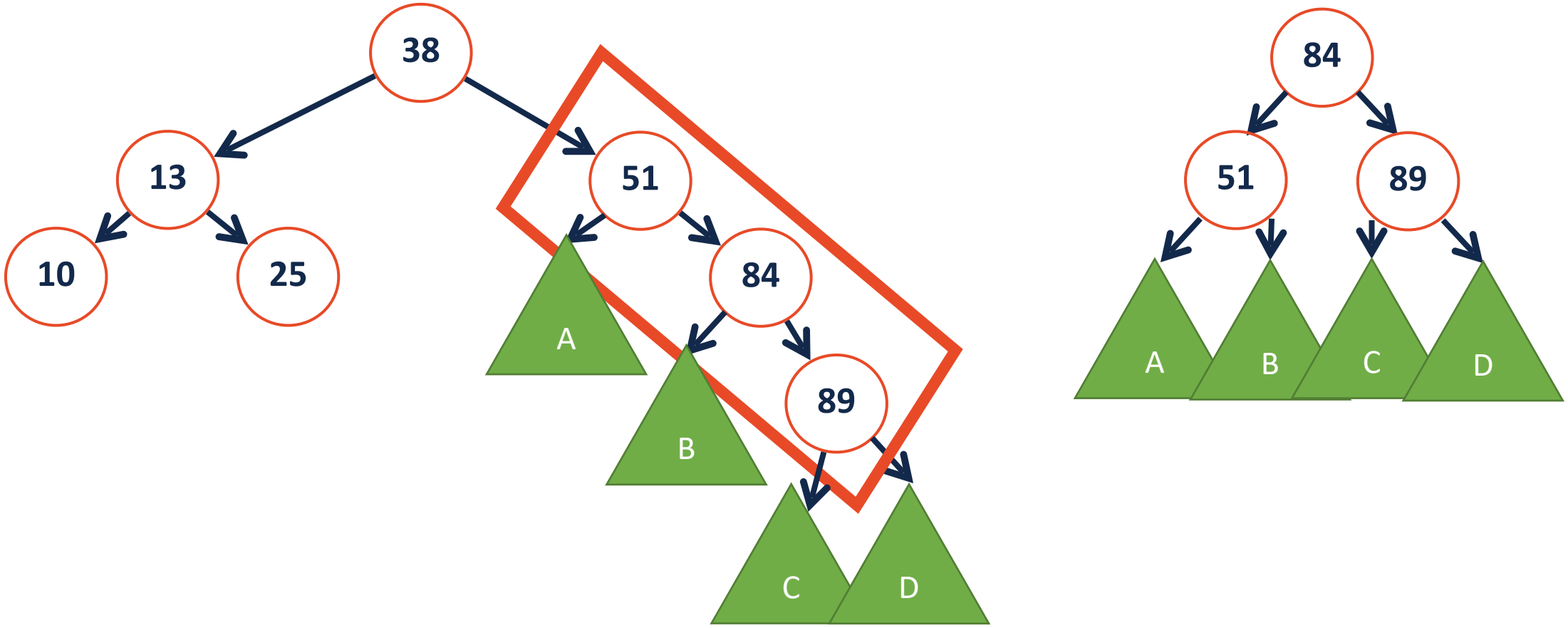
The height of a BST depends on the order in which the data was inserted

Insert Order: [1, 3, 2, 4, 5, 6, 7]

Insert Order: [4, 2, 3, 6, 7, 1, 5]

AVL-Tree: A self-balancing binary search tree

Rather than fixing an insertion order, just correct the tree as needed!



When would we use a tree?

Pretend for a moment that we always have an optimal BST.

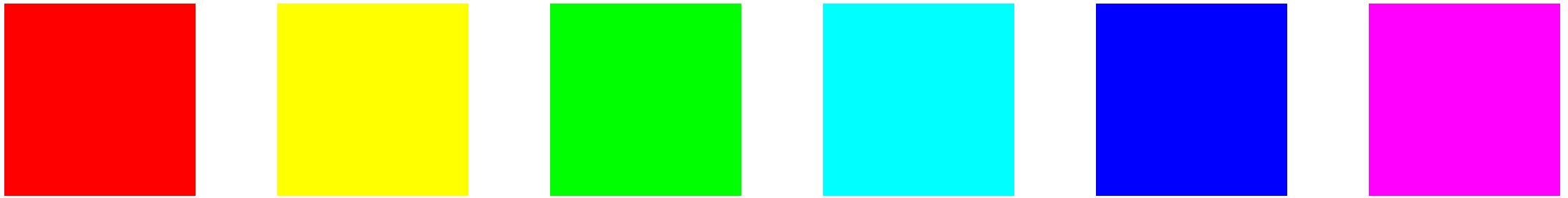
What is the running time of **find**?

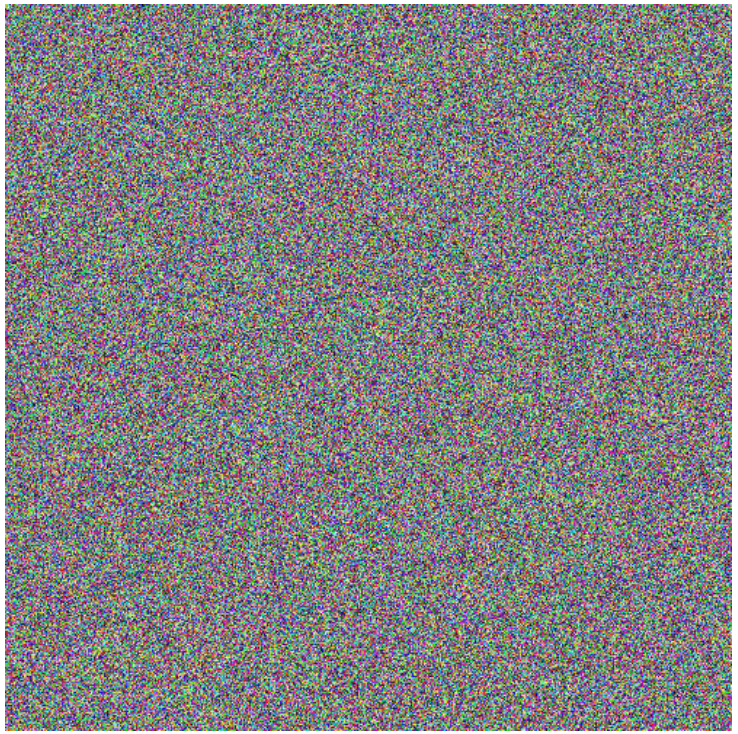
What is the running time of **insert**?

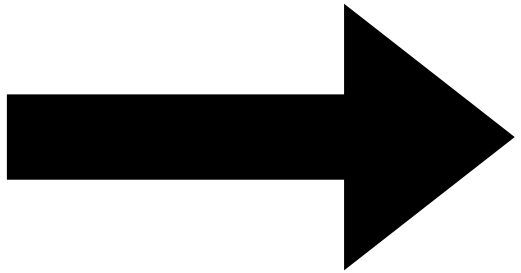
What is the running time of **remove**?

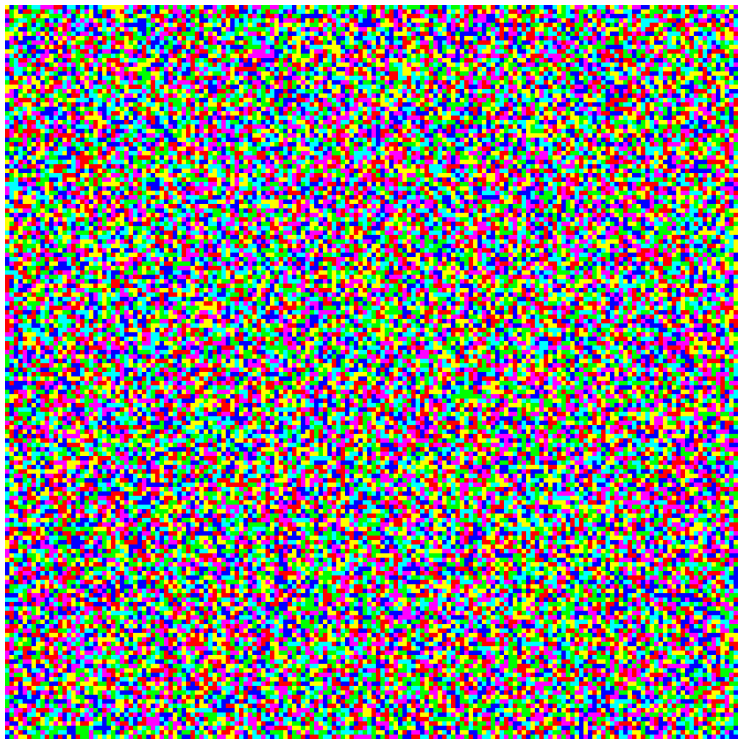
Is there a data structure with a *better* running time for all of these?

Real World Use Case: Nearest neighbor search









Real World Use Case: Nearest neighbor search

(255, 0, 0)

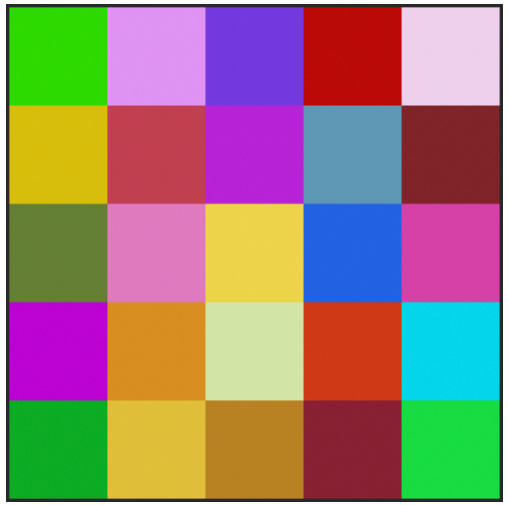
(255, 255, 0)

(0, 255, 0)

(0, 255, 255)

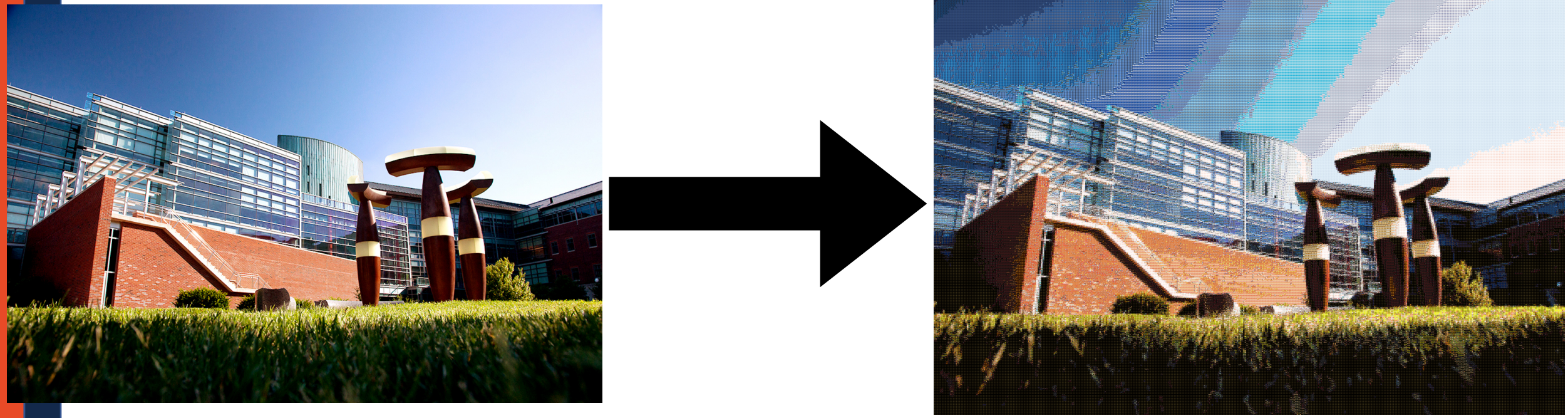
(0,0,255)

(255,0,255)

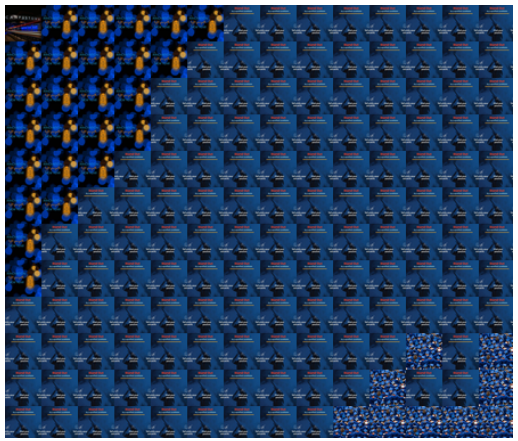


```
[[ 45 218   0],[223 147 243],[116  57 223],[187   9   9],[238 208 236]]

[[216 190  15],[193  64  80],[184  35 215],[ 95 152 180],[128  36  41]]

[[101 128  53],[224 122 191],[237 212  74],[ 35  98 227],[214  66 167]]

[[188   3 211],[217 142  33],[210 229 167],[208  57  22],[  3 213 235]]

[[ 11 172  37],[225 191  57],[184 130  34],[136  33  51],[ 26 220  67]]
```

Real World Use Case: Nearest neighbor search



Real World Use Case: Nearest neighbor search



Real World Use Case: Nearest neighbor search

Given an input image, how can we find the closest match from a collection collection of other images?