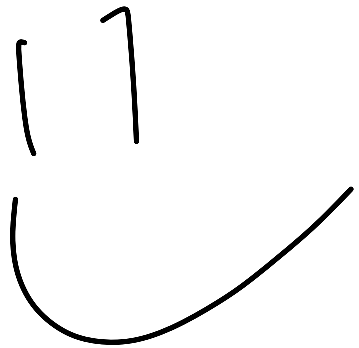
Algorithms and Data Structures for Data Science

Binary Search Tree 2

CS 277
Brad Solomon

March 6, 2024



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Reminder: mp_automata due Friday

93% credit late day extension through Saturday

Additional extensions by request

Reminder: Spring Break next week

Lab on Friday will still happen, will be due after spring break

No office hours during spring break

Exam 2: 3/19 - 3/21

Yes its right after spring break. Sorry!

Covered material described on website

One coding question — likely similar to mp_automata

Practice exam (hopefully) later this week

Learning Objectives

Finish implementation of BST ADT → Remove( )

Introduce the Huffman Tree

Practice recursion in the context of trees

Binary Search Tree

```python
class bstNode:
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right
```

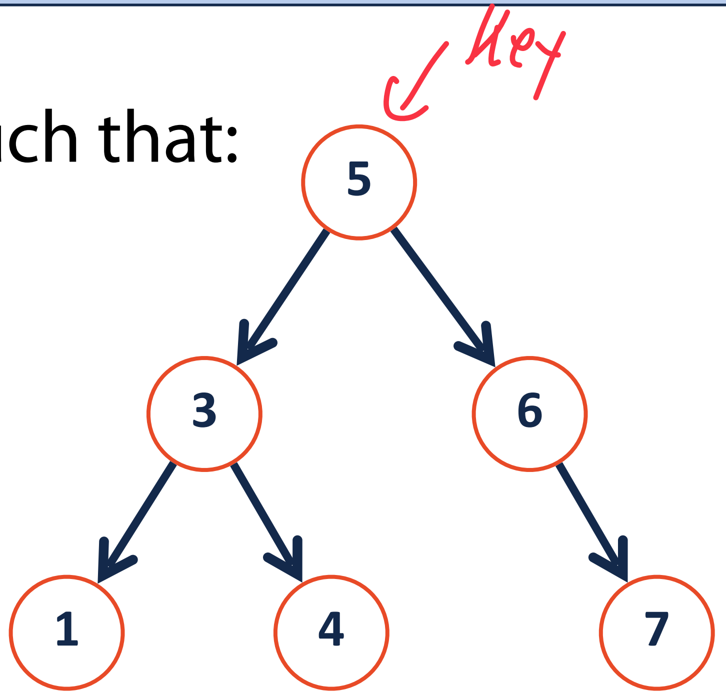
A **BST** is a binary tree $T = treeNode(val, T_L, T_r)$ such that:

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

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

Left is smaller

Right is larger

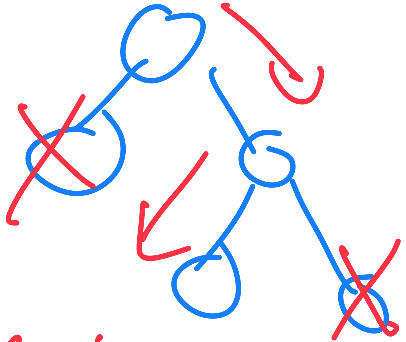


Key	5	3	6	7	1	4
Value	A	B	C	D	E	F

Binary **Search** Tree ADT — **what changed?**

Constructor: Build a new (empty) tree

Insert: Find the correct insert location based on BST structure



Remove: Find the node being removed and… ???

X implies don't have to look!

Traverse: Visit every node in tree (all objects)

Search is better b/c structure

Search: Find a specific node in the tree using the 'key' value

BST Remove

remove(40)

0 child case

1) Find node to be removed

2) Set parent.child = None

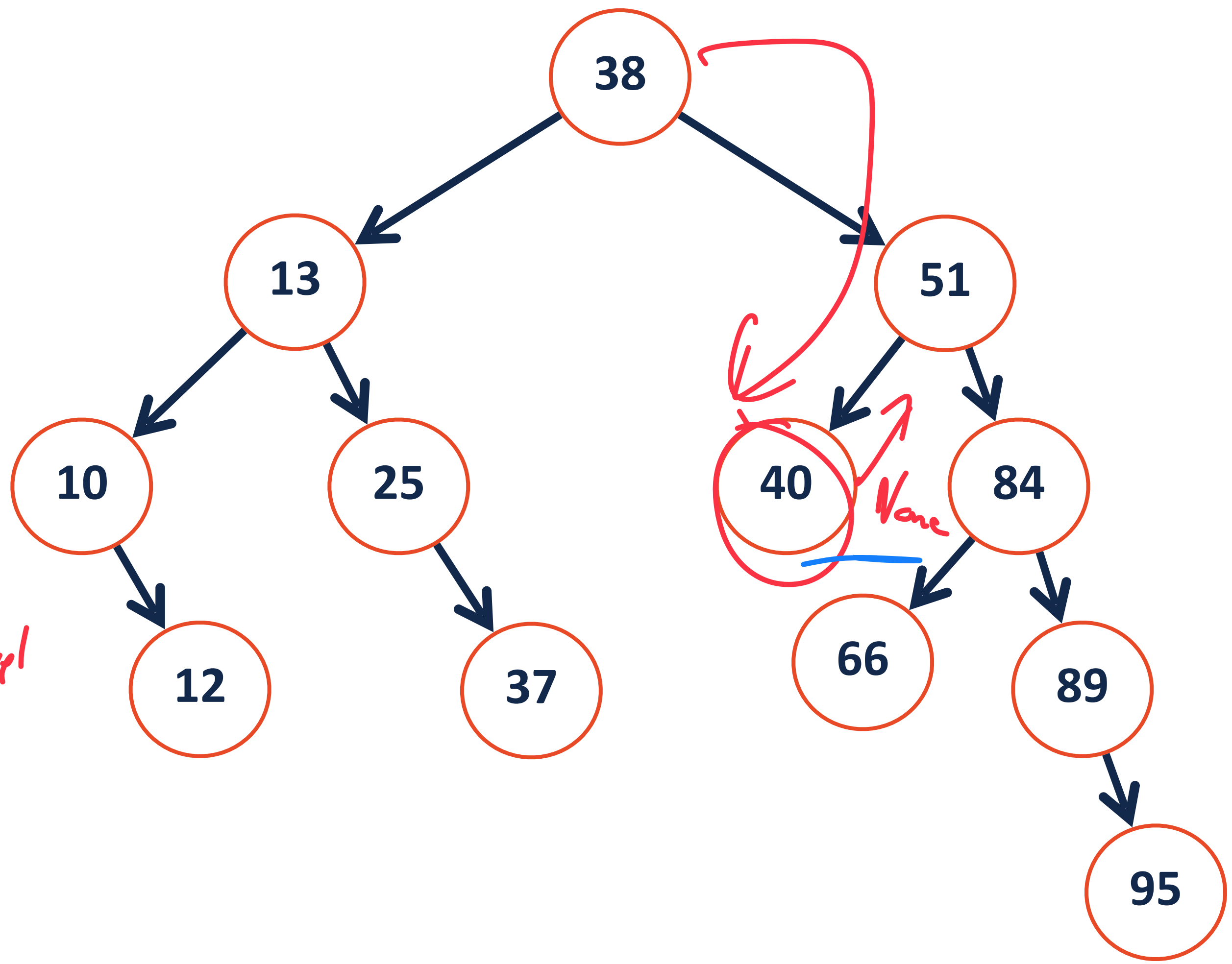
remove @ 38
→ remove @ 51
→ remove @ 40

return None

In our recursion Parent is one level up

node.left = (remove@ 40)

node@51.left = None



BST Remove

remove(25)

1 child case

Linked List!

Set parent.child = p.child.child

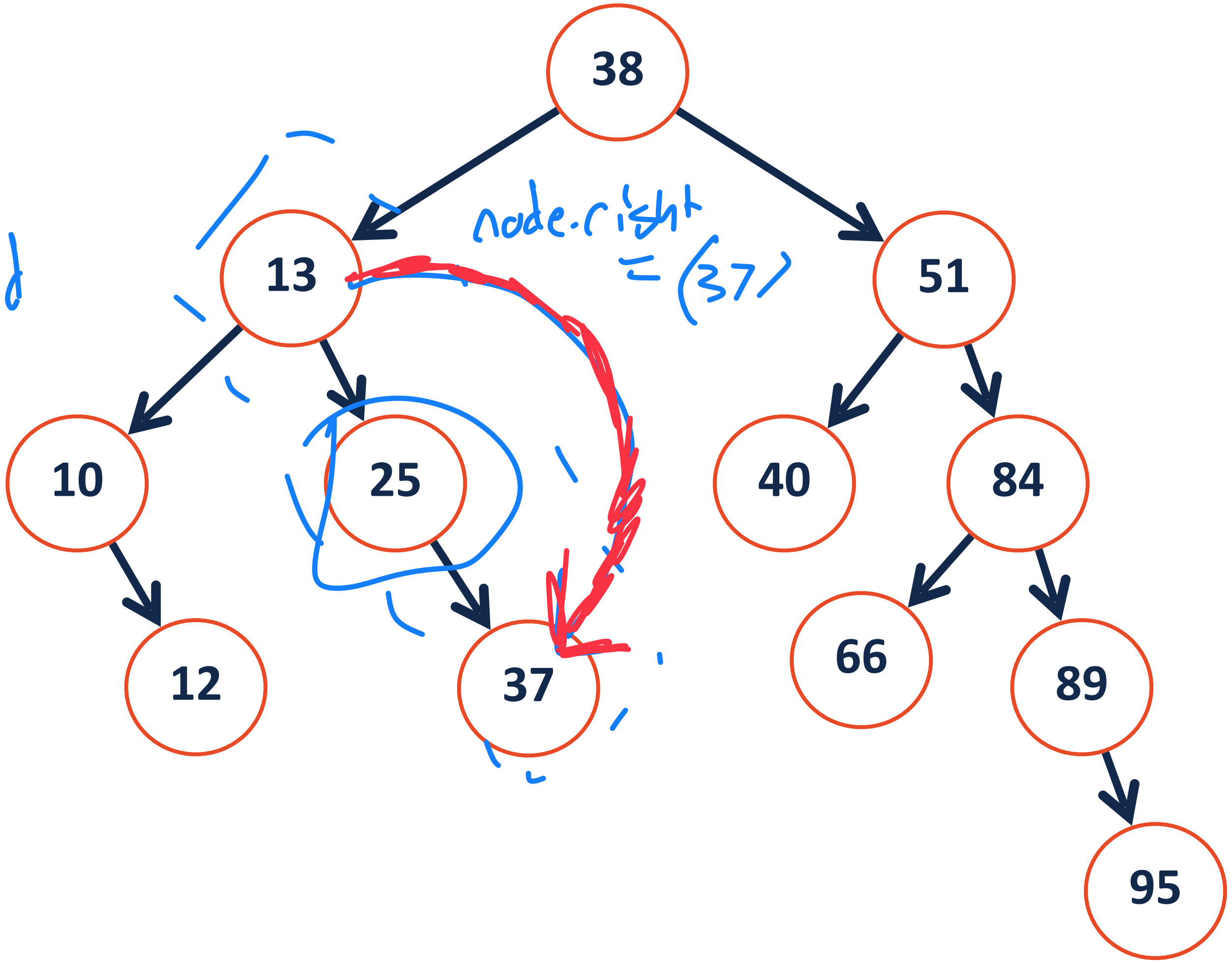
remove@38

↳ remove@13

↳ remove@25

return (25).right

node.right = remove@25



BST Remove

remove(13)

2 child case

1) Find Node being removed

2) Swap w/ IOP or IOS

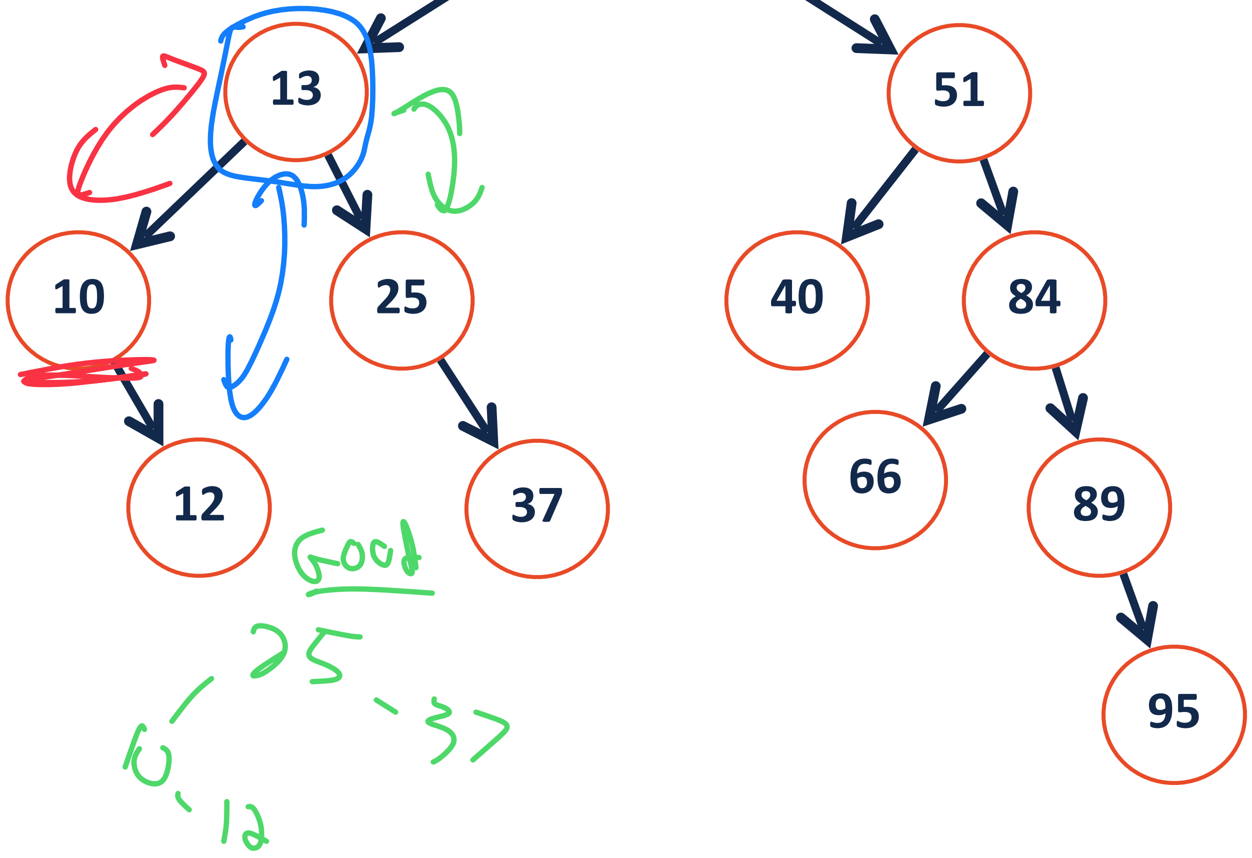
↳ Find IOP/IOS

↳ swap key, value

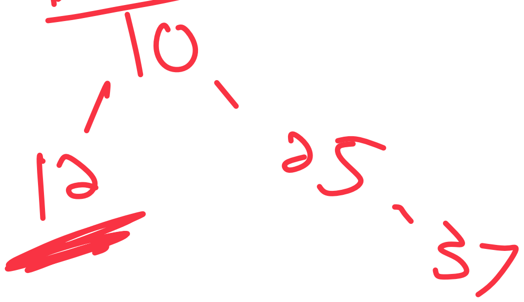
IOT: 10 12 ~~13~~ 25 37 ....

In-order predecessor

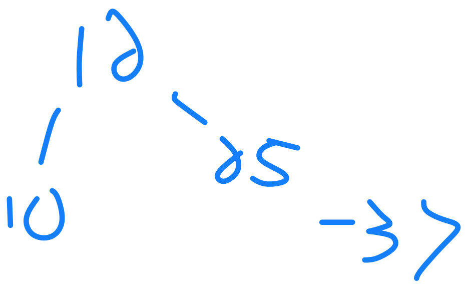
In-order successor



Bad



Good



BST Remove

Remove(5)

```
def remove(self, key):
    self.root = self.remove_helper(self.root, key)

def remove_helper(self, node, key):
```

1) Find Node

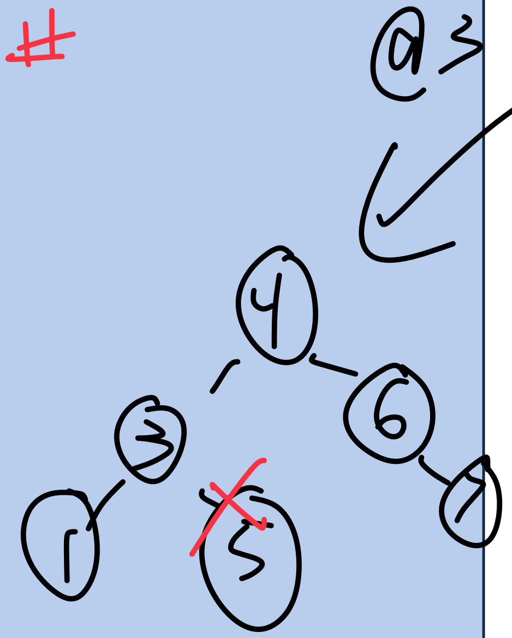
2) Find IOP / IOS — the next largest #

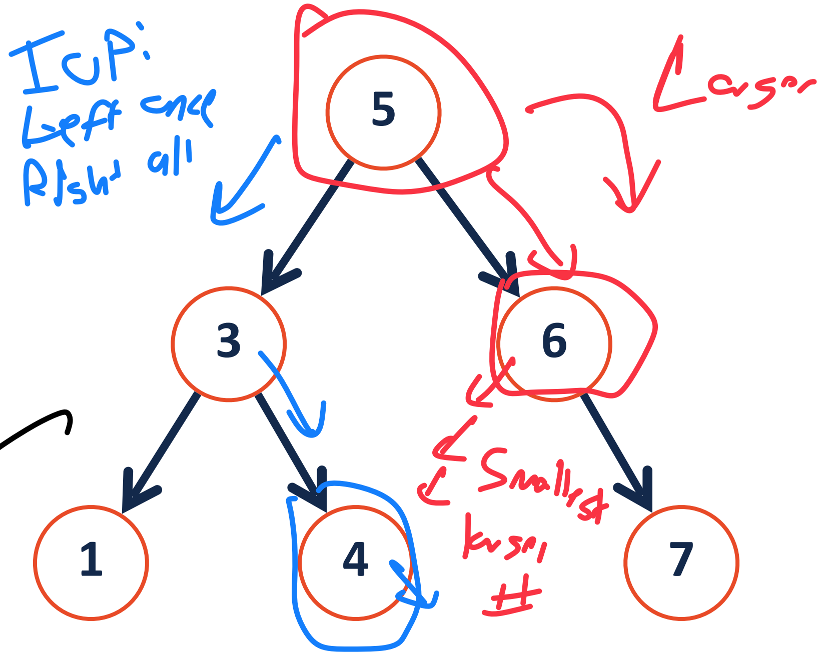
3) Swap Values

4) Remove Node

↳ 0 or 1 child case

guaranteed





IOS:

↳ Recurse right once

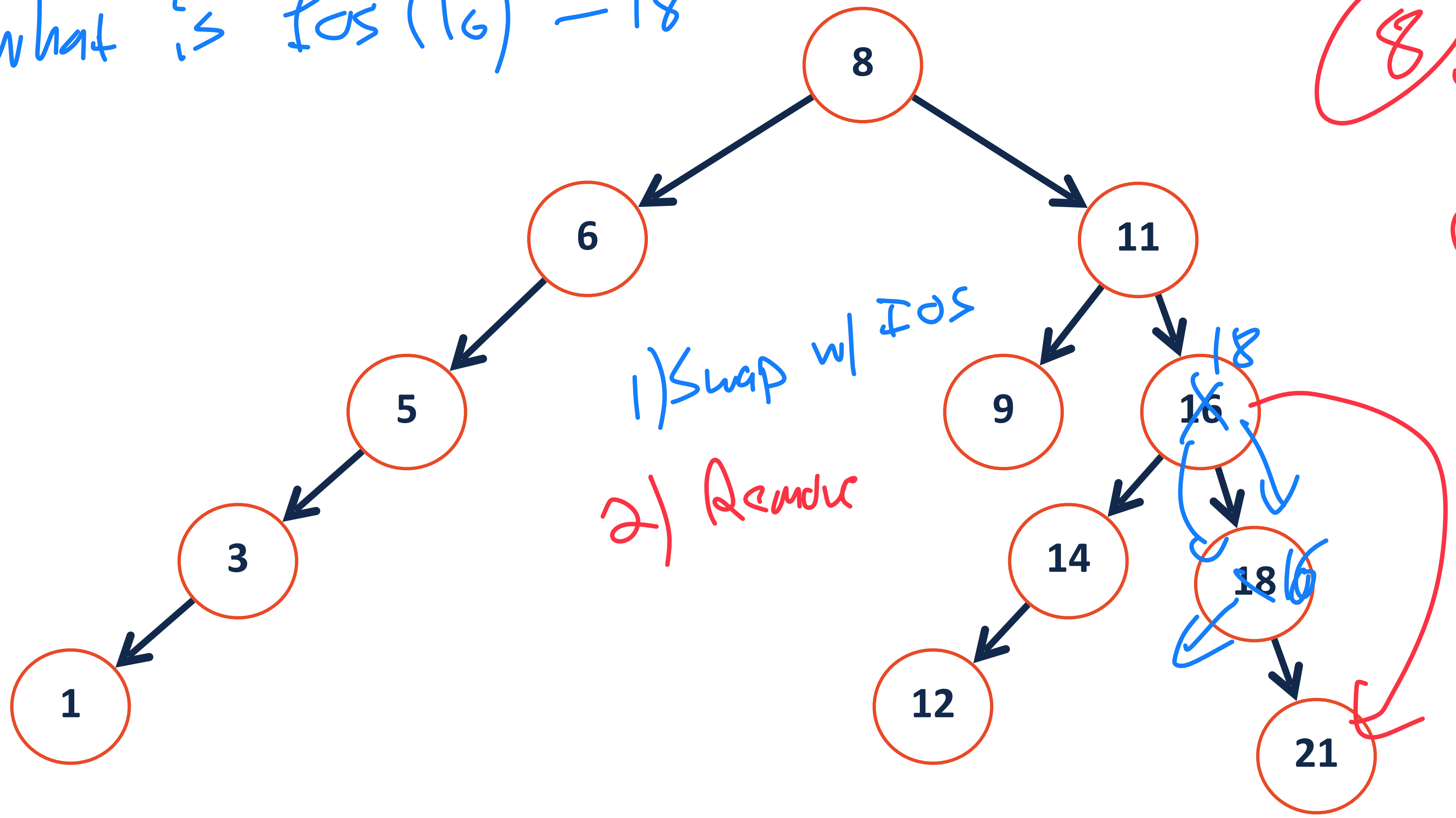
↳ Recurse left until None

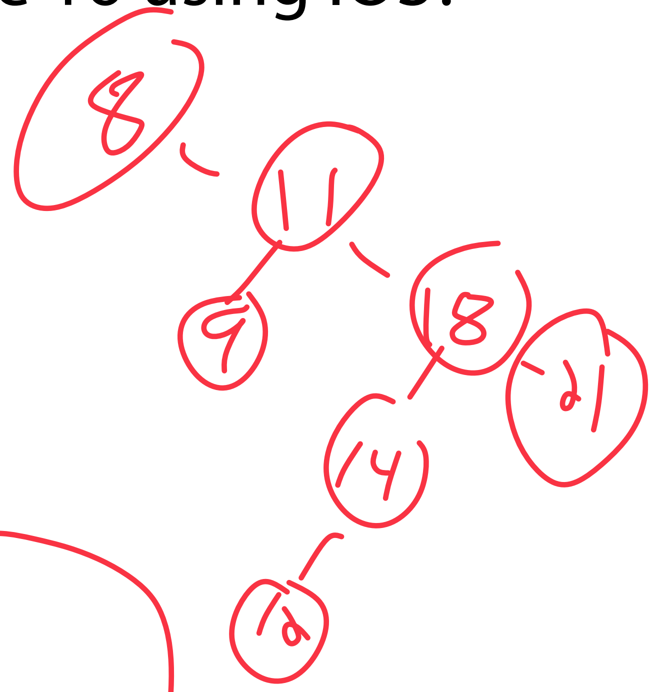
↳ the last non-None value is the IOS

BST Remove

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

1) What is IOS(16) — 18





BST Analysis – Running Time

Operation	BST Worst Case
find	$O(n)$
insert	$O(n)$
~~delete~~ remove	$O(n)$
traverse	$O(n)$

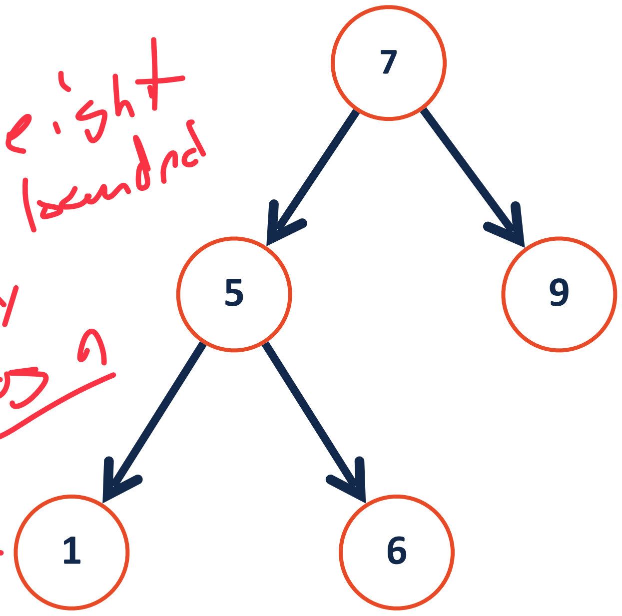
$O(\text{height})$

height = n

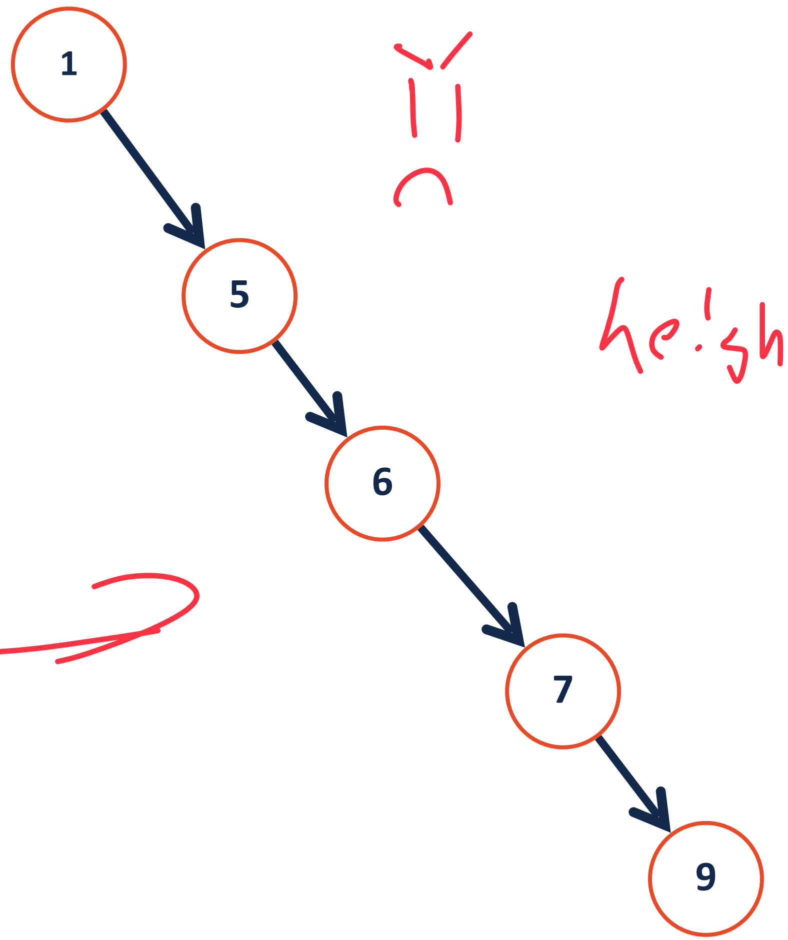
My worst case tree

Limiting the height of a tree

height bounded by log n



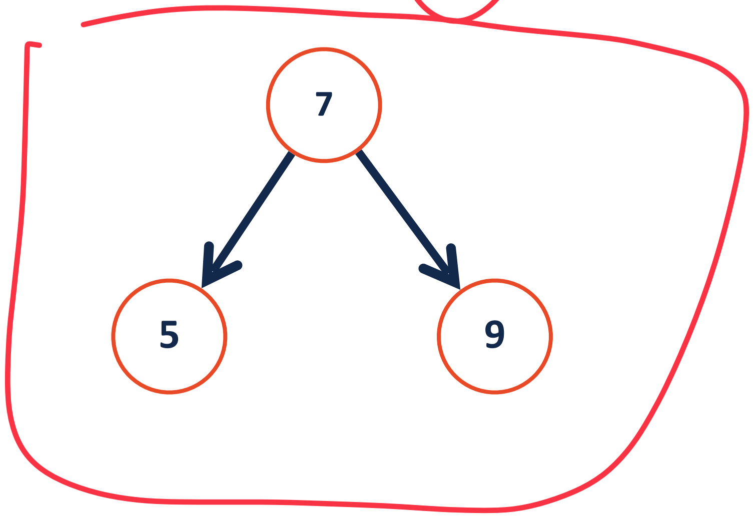
can be log n



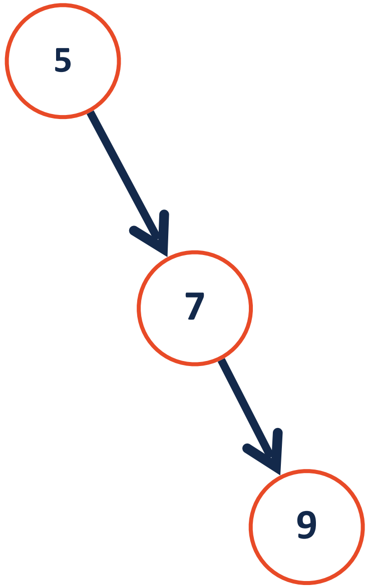
height is n

Height-Balanced Tree

What tree is better? A



B

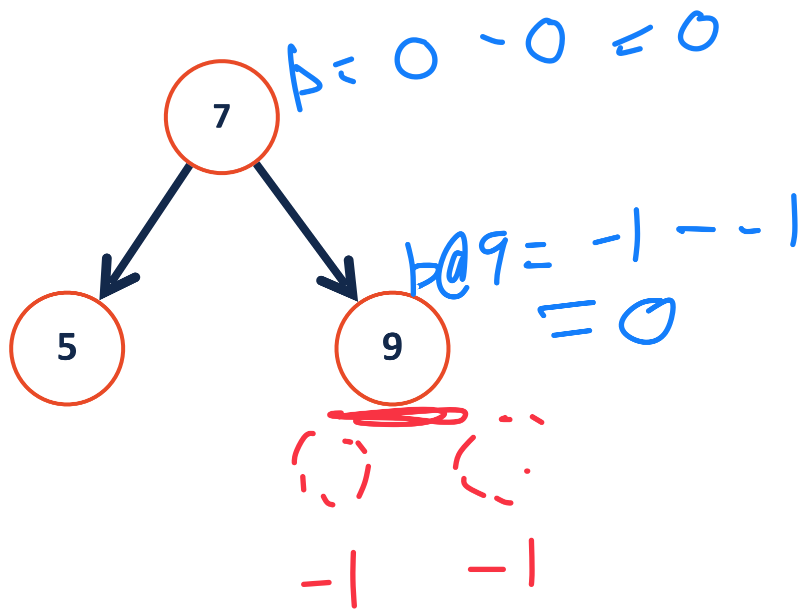


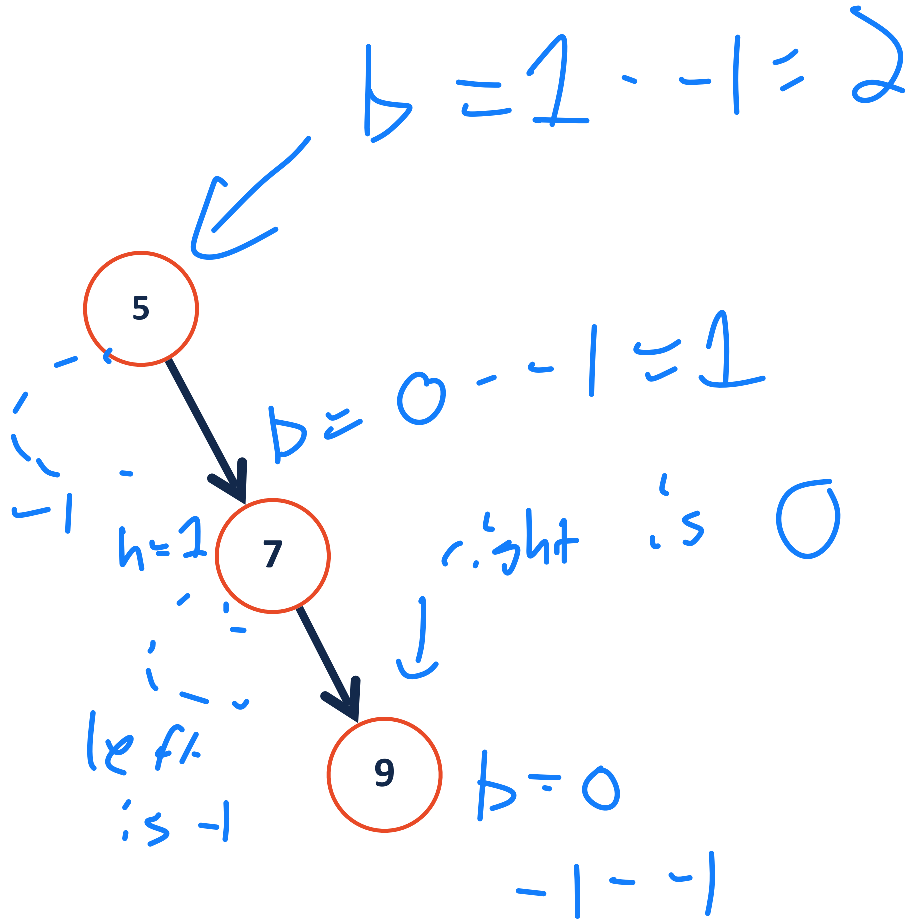
How would you describe this mathematically?

Height-Balanced Tree

x
root is leaf
height = 0

What tree is better?





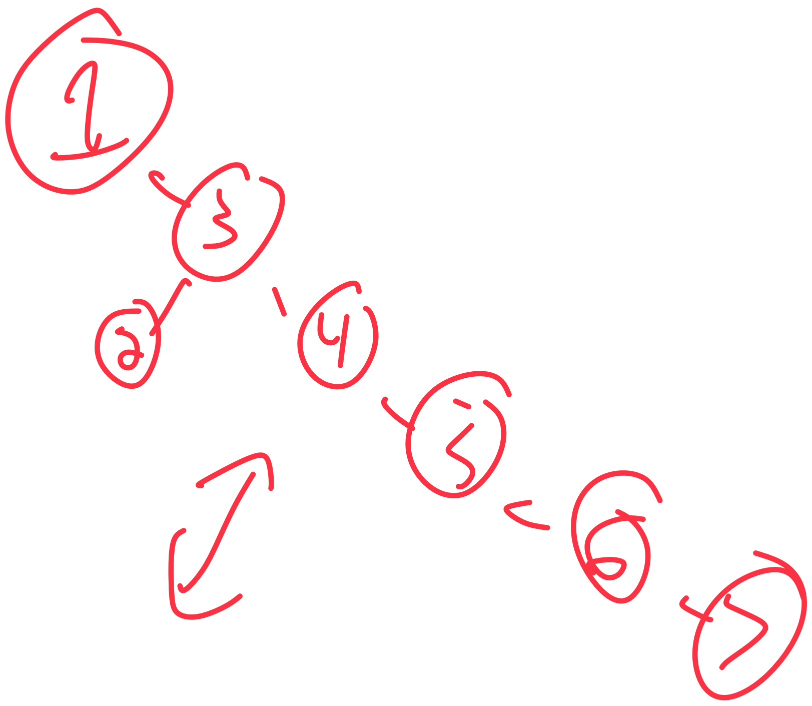
Height balance: $b = height(T_R) - height(T_L)$

A tree is "balanced" if: all nodes have $b < 2$

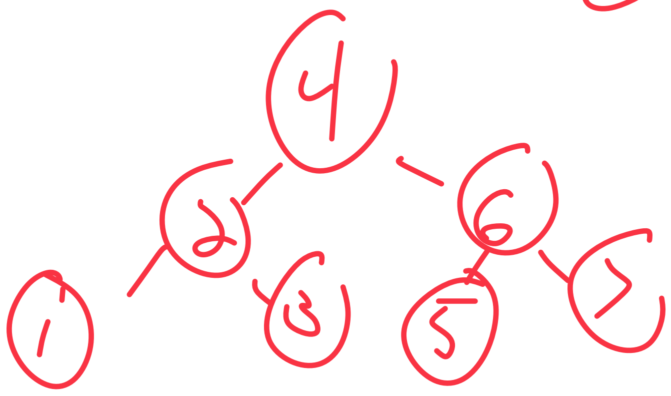
Option A: Correcting bad insert order

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]

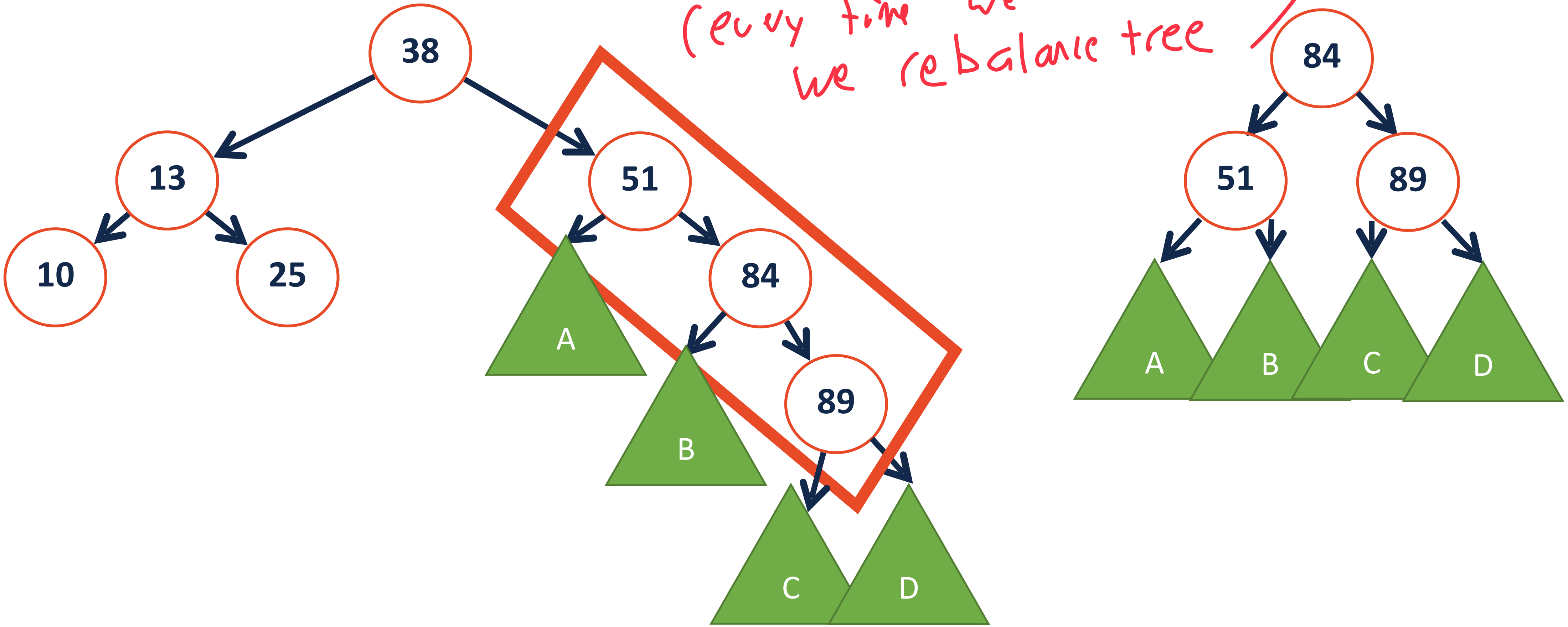


AVL-Tree: A self-balancing binary search tree

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

↳ insert & remove

(every time we make change we rebalance tree)



We will return to this topic… after spring break!

Optimal Storage Costs

Achieving an optimal storage cost for a dataset is often important

Let's use strings as an accessible example!

What is the minimum bits needed to encode the message:

Char	Binary
f	000
e	001
d	010
m	100
r	011
o	101
' '	110

`'feed me more food'`

$2^3 = 8$

7 characters = 3 bits

111 - 8th char

Optimal Storage Costs

Using three bits per character, we have 51 bits total. But can we do better?

`'feed me more food'`

If we think about our input as a sorted list of frequencies, yes!

r : 1 | d : 2 | f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

count # of each character

More bits

low freq

f3

Few bit cost

high freq

4

Using binary trees for string encoding

Lets define a tree with the following:

The keys are individual characters

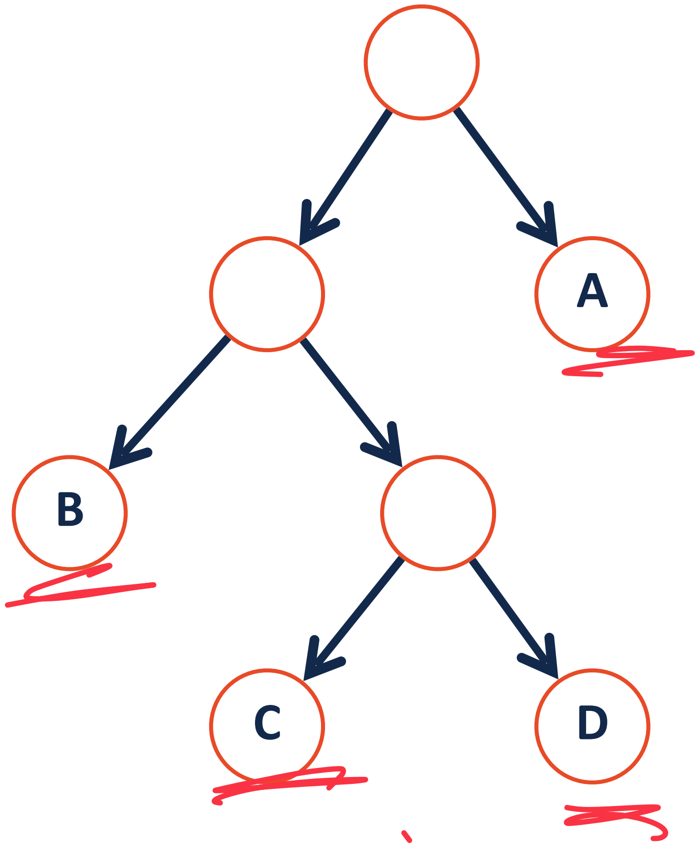
The values are the frequencies of those characters

```
class treeNode:
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right
```

Key	A	B	C	D
Value	7	5	2	4

← characters

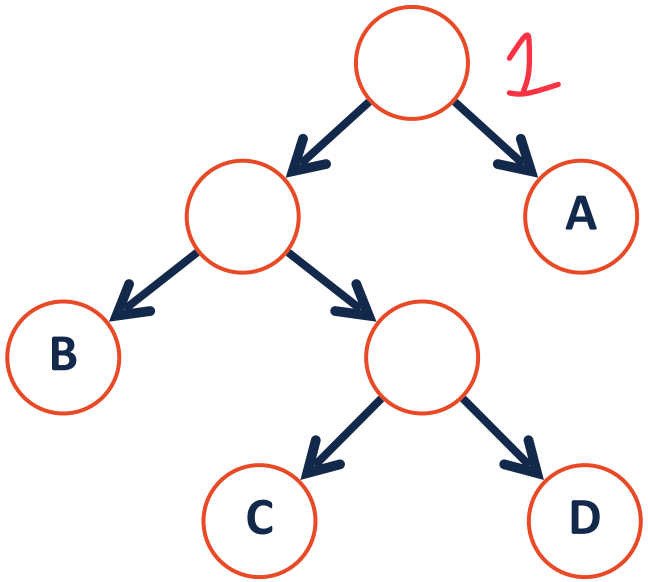
← frequency

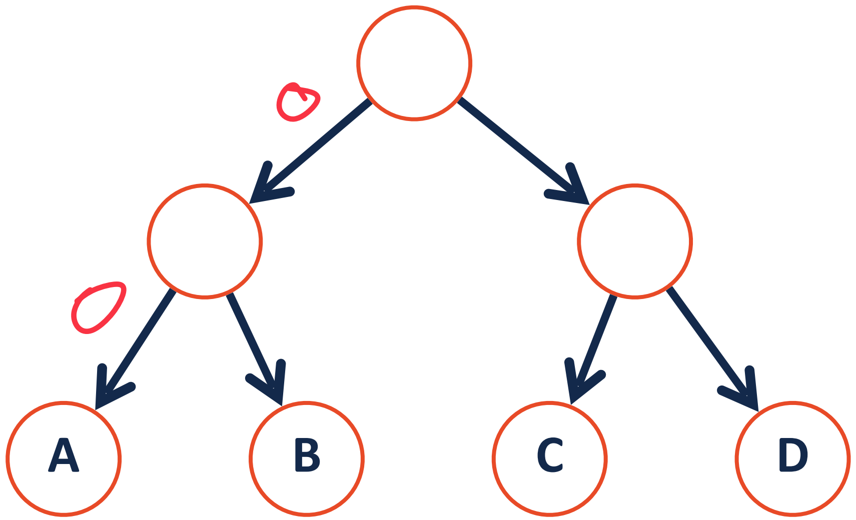


leaves are single characters

Binary Tree encoding

Given the following two trees, how might we define an encoding?



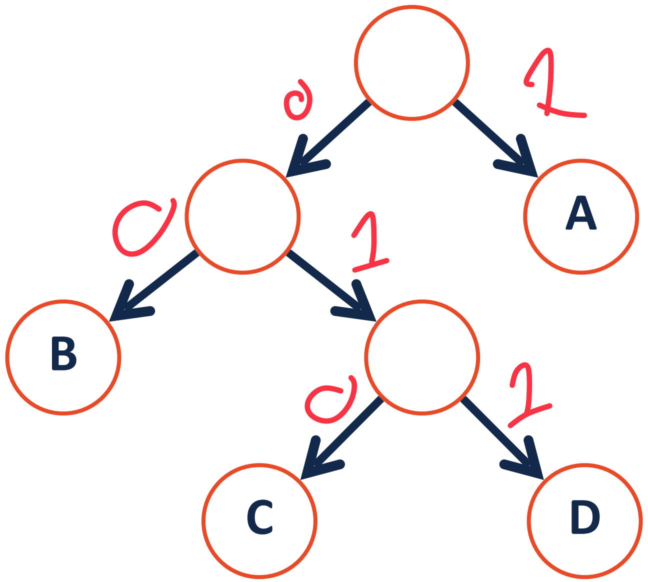


A = 1

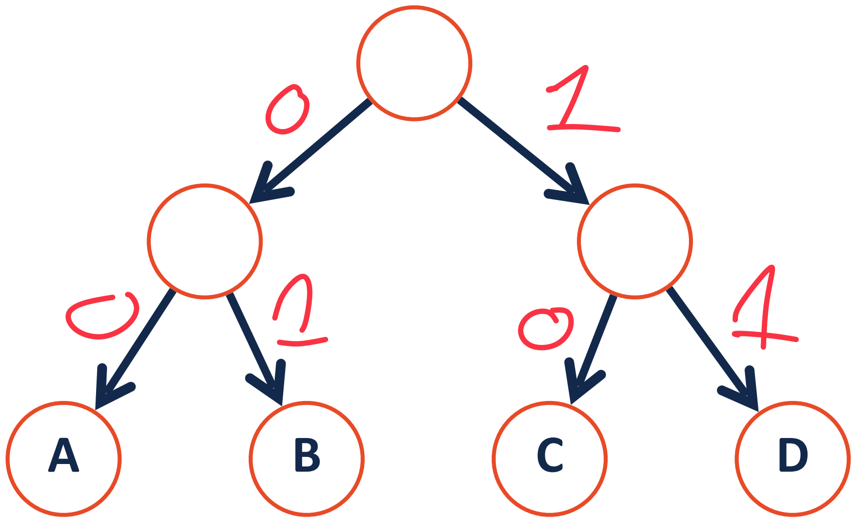
A = 00

Binary Tree encoding

How did we produce this encoding?



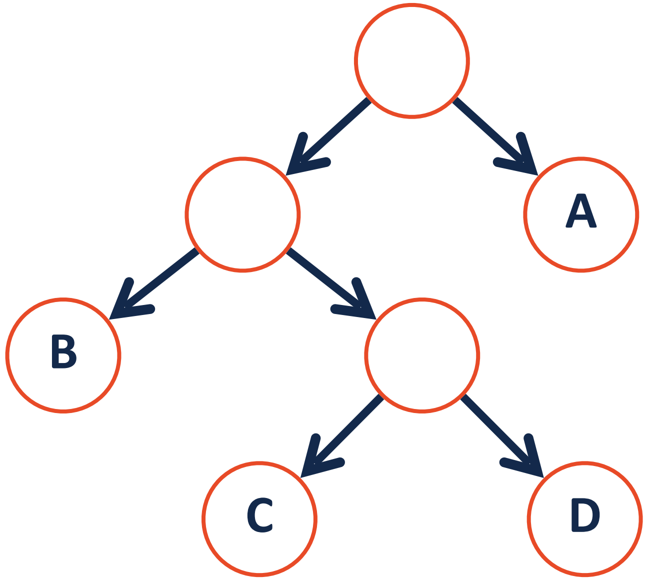
Char	Binary
A	1
B	00
C	010
D	011



Char	Binary
A	00
B	01
C	10
D	11

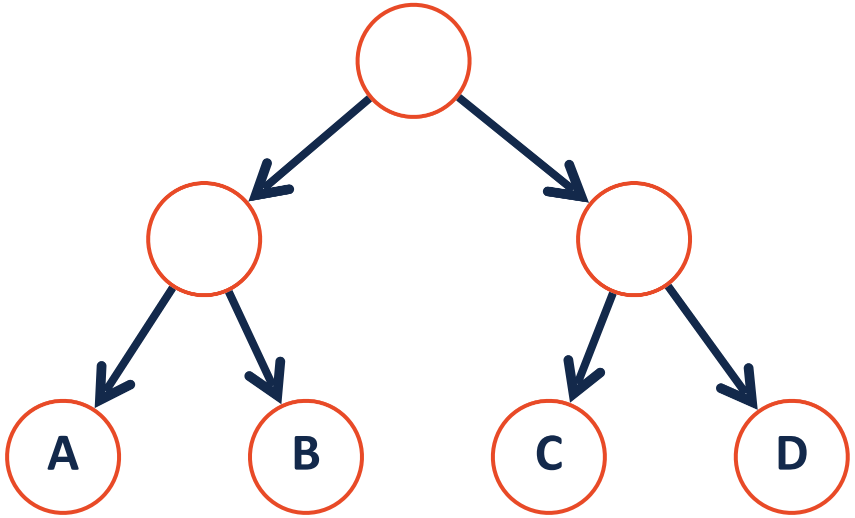
Binary Tree encoding

The **path** from root to leaf defines our encoding, but which tree is best?



Going left = 0

Going right = 1



Char	Binary
A	1
B	00
C	010
D	011

???

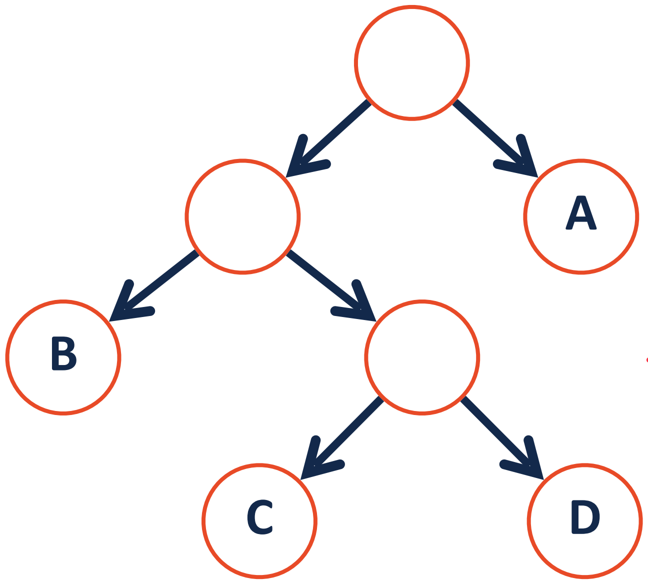
the ~~best~~ tree
correct

Char	Binary
A	00
B	01
C	10
D	11

Binary Tree encoding

If my frequencies are {A : 7 | B : 5 | C : 2 | D : 4 }, which tree was better?

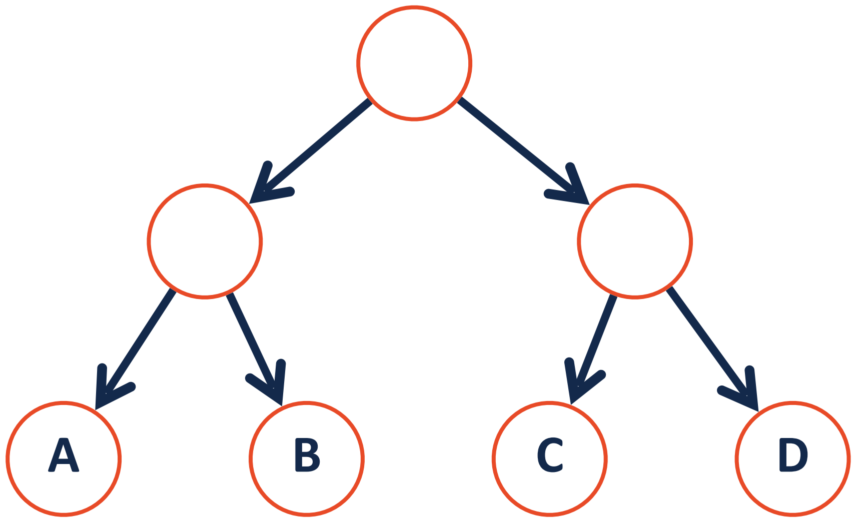
Build a tree based on frequencies



35

36

~ "Better"



Char	Binary
A	1
B	00
C	010
D	011

7*1
5*2
2*3
4*3

Char	Binary
A	00
B	01
C	10
D	11

2*(7 + 5 + 2 + 4)

Building the Huffman Tree

The **Huffman Tree** is the tree with the optimal total path length for a given set of characters and their frequencies.

Step 1: Calculate the frequency of every character in text and order by increasing frequency. Store in a queue (a sorted list).

`Input: 'feed me more food'`

r : 1 | d : 2 | f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

only remove smallest

only add larger

Building the Huffman Tree

Step 2: Build a tree from the bottom up. Start by taking the two least frequent characters and merge them (create a parent node). Store the merged characters in a new queue.

getSmallest()
↳ To be a queue
Removes & returns

Input:

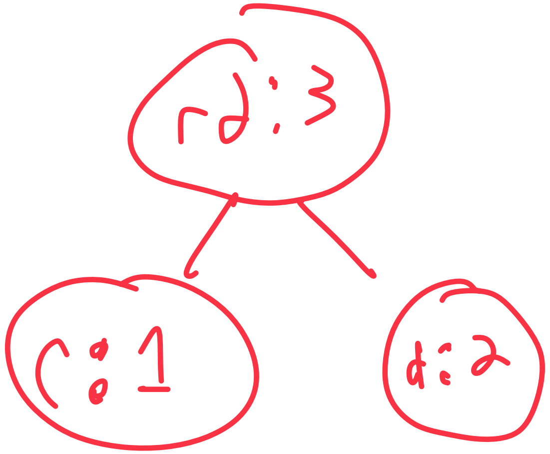
~~r : 1~~ | ~~d : 2~~ | f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

Merge:

1) concatenate strings/characters

c + d = "cd"

2) sum the frequencies

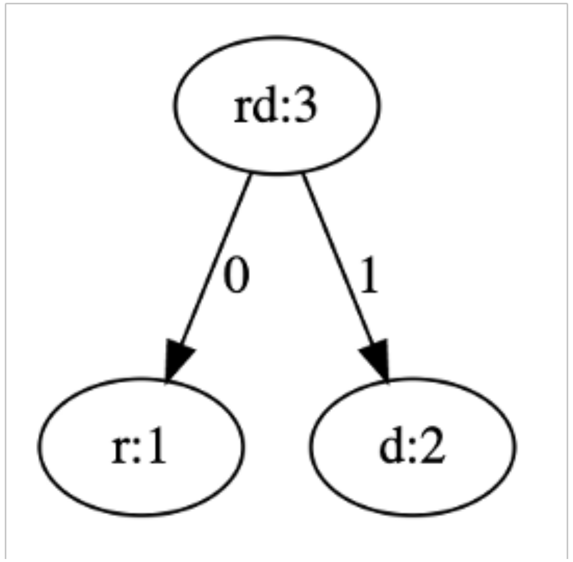


Building the Huffman Tree

Step 2: Build a tree from the bottom up. Start by taking the two least frequent characters and merge them (create a parent node). Store the merged characters in a new queue.

Input:

r : 1 | d : 2 | f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4



Output:

Single: f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3

Building the Huffman Tree

Step 3: Repeatedly merge the minimum two items from either list. Be sure to **remove and return** the minimum item as seen below:

Queue is easy to maintain!

Input:

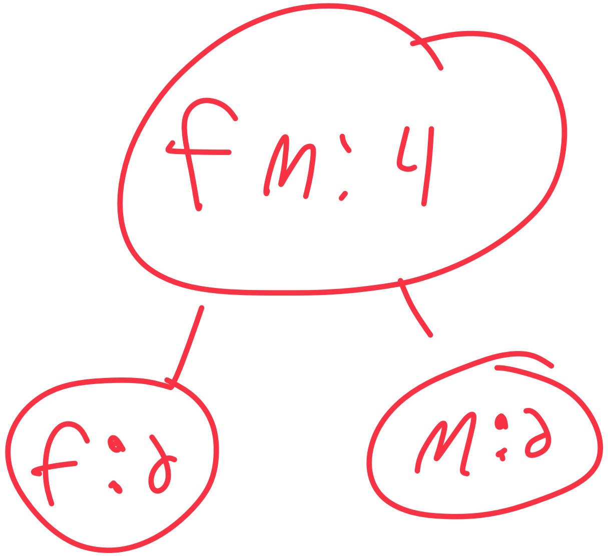
Single: f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3 , fm : 4

Merge always add to end

1st 2nd smallest

3rd 4th smallest



Building the Huffman Tree

Step 3: Repeatedly merge the minimum two items from either list. Be sure to **remove and return** the minimum item as seen below:

Input:

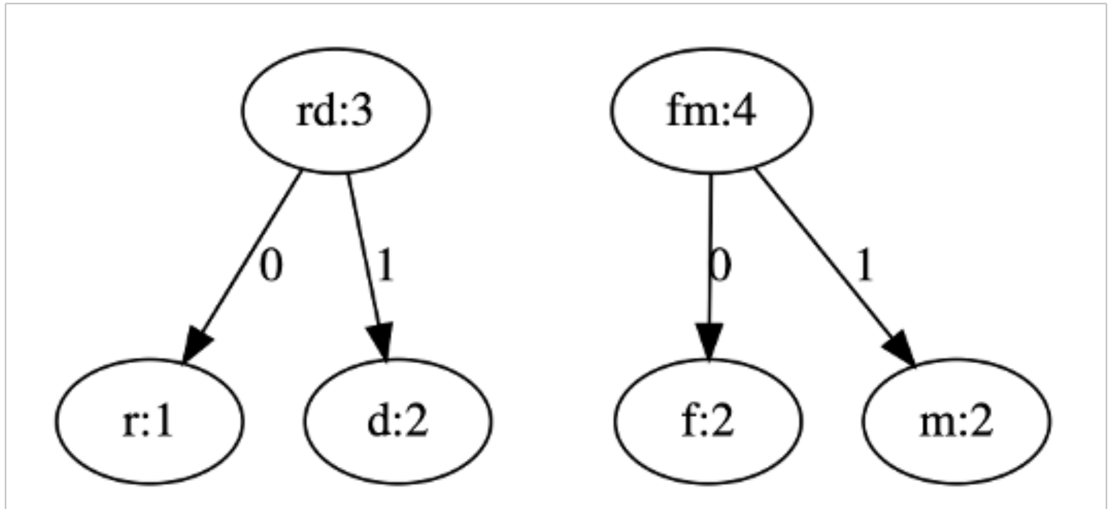
Single: f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3

Output:

Single: o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3 | fm : 4



Building the Huffman Tree

Step 3: Repeatedly merge the minimum two items. Note that **by inserting in the back** the merged items will always remain sorted!

Input:

Single: o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3 | fm : 4

Building the Huffman Tree

Step 3: Repeatedly merge the minimum two items. Note that **by inserting in the back** the merged items will always remain sorted!

Input:

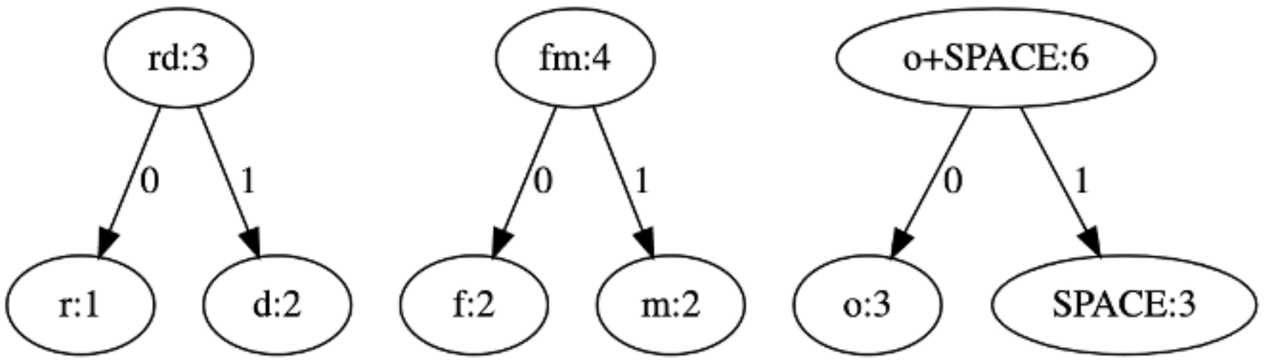
Single: o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3 | fm : 4

Output:

Single: e : 4

Merged: rd : 3 | fm : 4 | o'SPACE' : 6



Building the Huffman Tree

Step 3: Once the 'single' character list has been exhausted, we can easily merge the rest of our list by taking the front two values in merged.

Input:

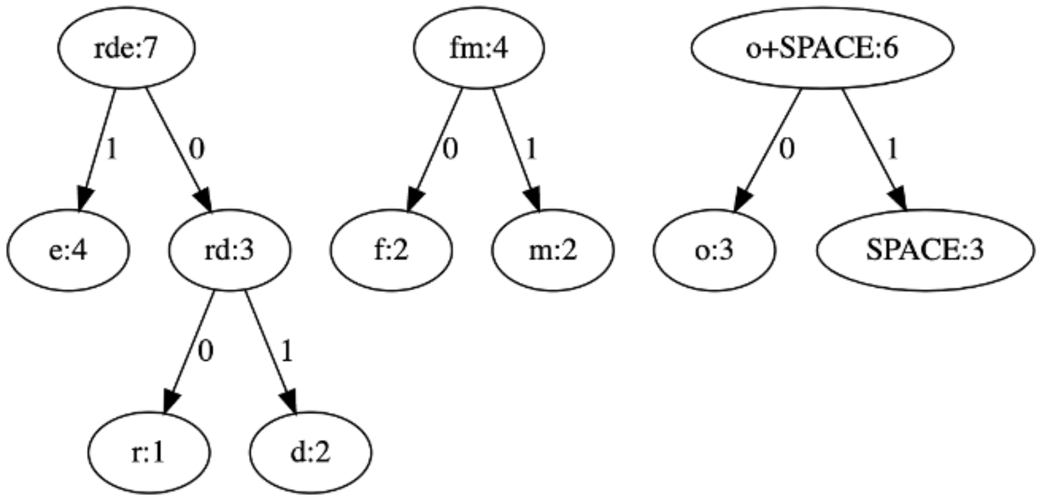
Single: e : 4

Merged: rd : 3 | fm : 4 | o'SPACE' : 6

Output:

Single:

Merged: fm : 4 | o'SPACE' : 6 | rde : 7



Building the Huffman Tree

Step 3: Once the 'single' character list has been exhausted, we can easily merge the rest of our list by taking the front two values in merged.

Input:

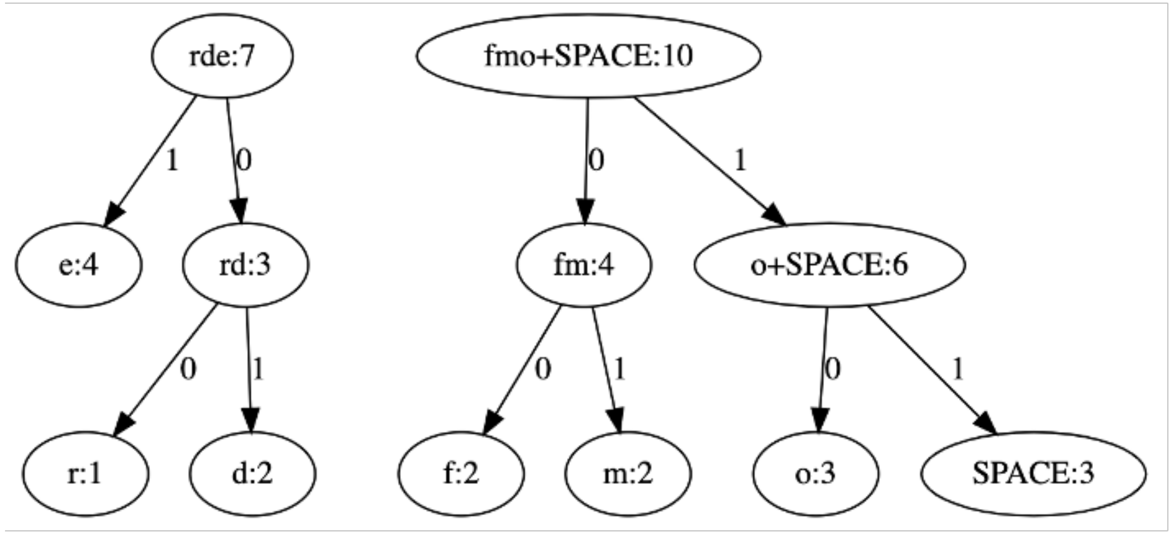
Single:

Merged: fm : 4 | o'SPACE' : 6 | rde : 7

Output:

Single:

Merged: rde : 7 | fmo'SPACE' : 10



Building the Huffman Tree

Step 4: Stop when there is only a single item in either queue. This is our complete binary tree!

Input:

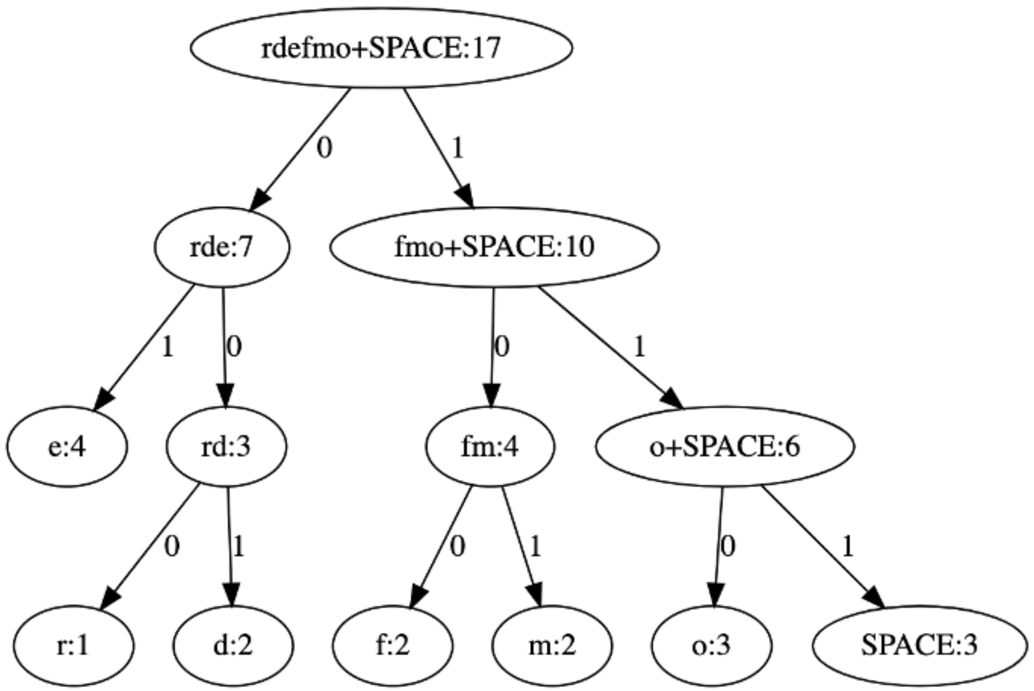
Single:

Merged: rde : 7 | fmo'SPACE' : 10

Output:

Single:

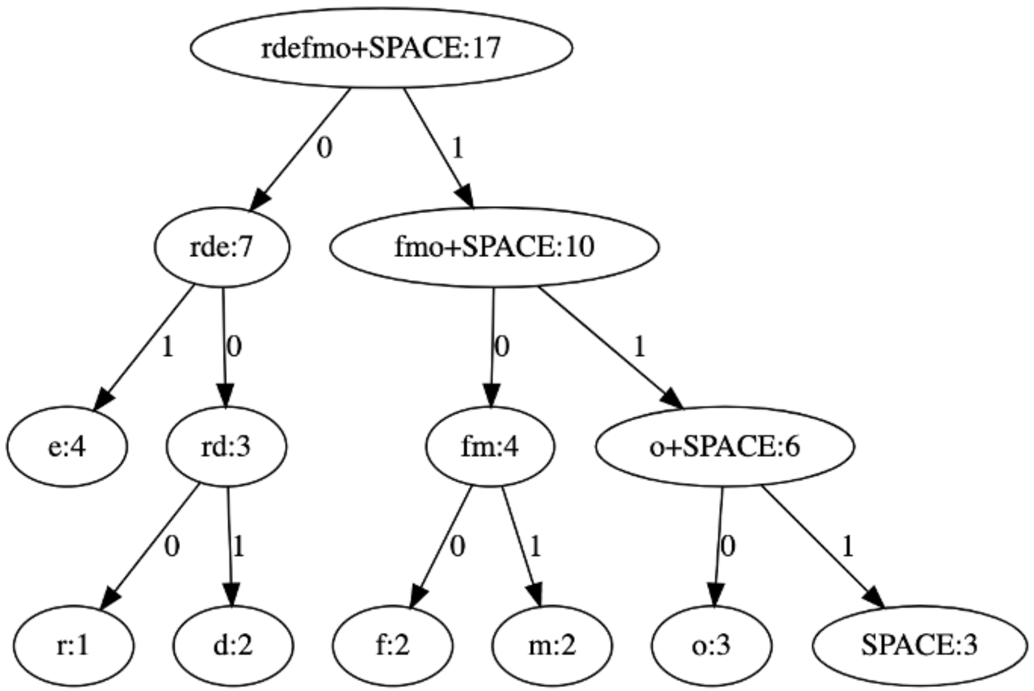
Merged: rdefmo'SPACE' : 17



Encoding using the Huffman Tree

The path through the tree defines each individual character's encoding!

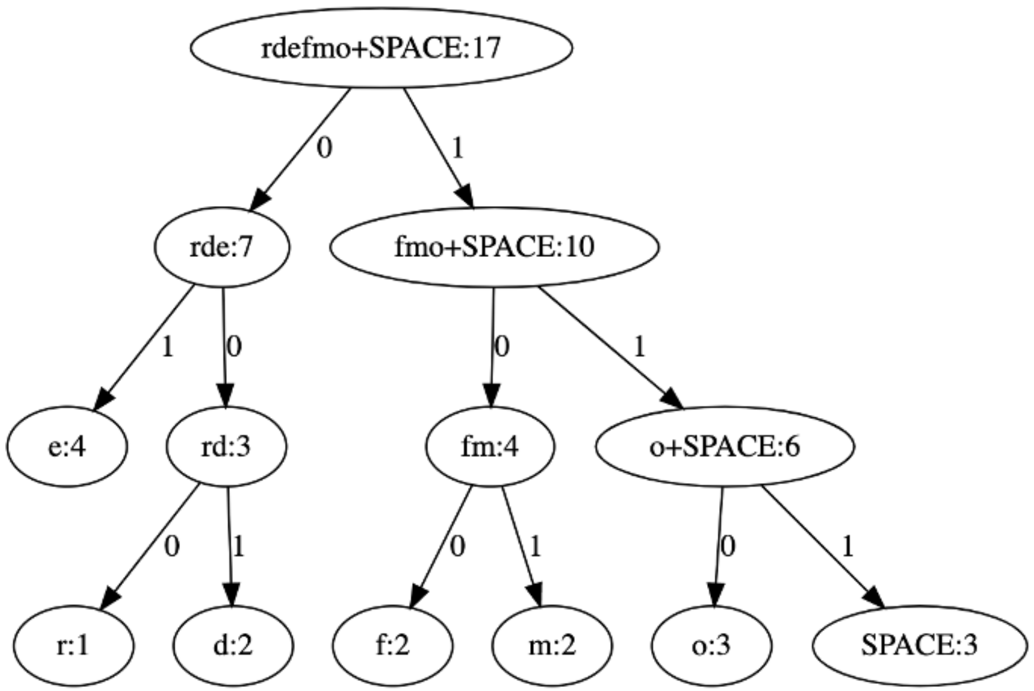
Char	Binary
f	
e	
d	
m	
r	
o	
' '	



Encoding using the Huffman Tree

The path through the tree defines each individual character's encoding!

Char	Binary
f	100
e	01
d	001
m	101
r	000
o	110
' '	111



Decoding using the Huffman Tree

We can decode by walking through the tree using 0s and 1s as instructions!

Input: `100010100111110101`

Output: