



哈爾濱工業大學(深圳)

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

数据结构

Data Structures

Chapter 9 Tree II (Advanced Trees)

Prof. Yitian Shao

School of Computer Science and Technology

Tree II (Advanced Trees)

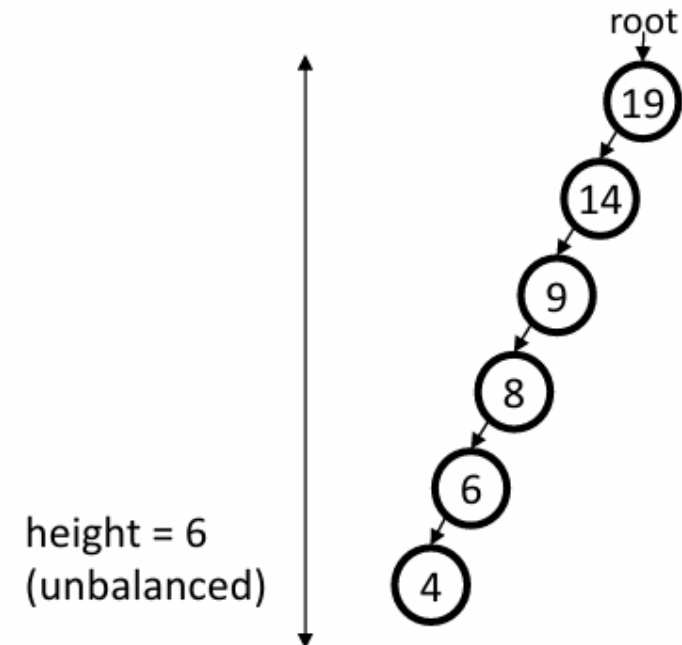
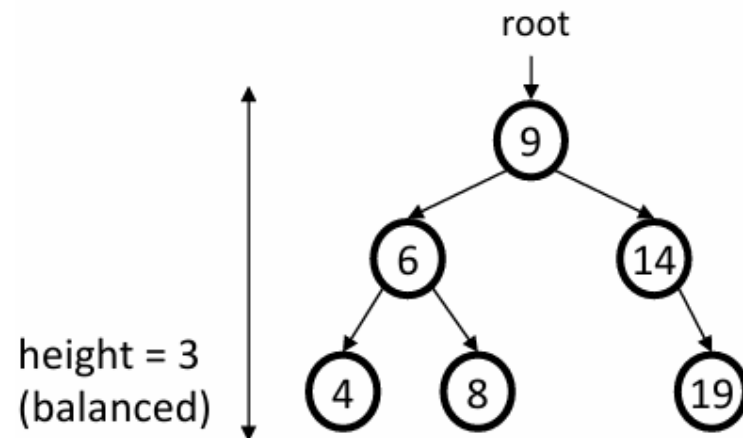
Course Overview

- Balanced Trees
- AVL Trees
- Heap and Priority Queue
- Forest and Generic Trees
- Convert Generic Tree to Binary Tree
- Other Binary Trees

The course content is developed partially based on Stanford CS106B. Copyright (C) Stanford Computer Science and Tyler Conklin, licensed under Creative Commons Attribution 2.5 License.

Trees and Balance

- **Balanced tree:** One where for every node R, the height of R's subtrees differ by at most 1, and R's subtrees are also balanced.
- Runtime of add / remove / contains are closely related to height.
- Balanced tree's height is roughly $\log_2 N$.
- Unbalanced is closer to N



BST Balance Question

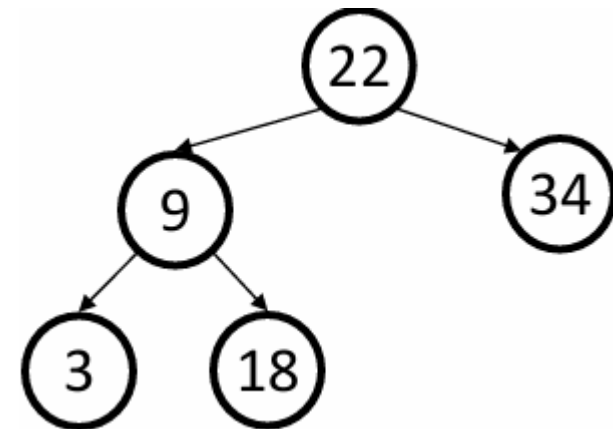
- Adding the following nodes to an empty BST in the following order produces the tree shown below: 22, 9, 34, 18, 3.
- What is an order in which we could have added the nodes to produce an unbalanced tree?

A. 18, 9, 34, 3, 22

B. 9, 18, 3, 34, 22

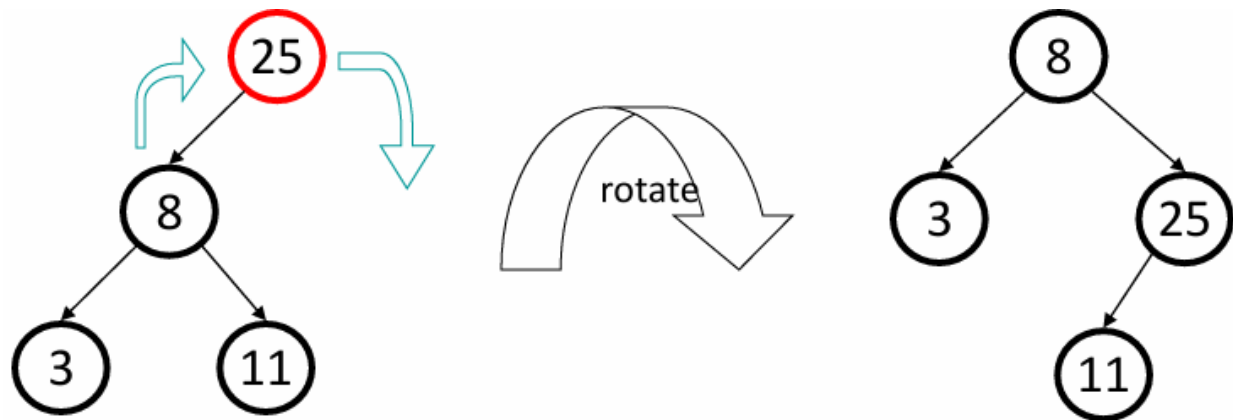
C. 9, 22, 3, 18, 34

D. none of the above



AVL Tree (Adelson-Velsky Landis)

- The **AVL tree** is named after its two Soviet inventors, Georgy **Adelson-Velsky** and Evgenii **Landis**, published in 1962 via a paper.
- AVL tree is a binary search tree that uses modified add and remove operations to **stay balanced as its elements change**.
- Basic idea: When nodes are added/removed, repair tree shape until balance is restored.
 - Rebalancing is $O(1)$
 - Overall tree maintains an $O(\log_2 N)$ height



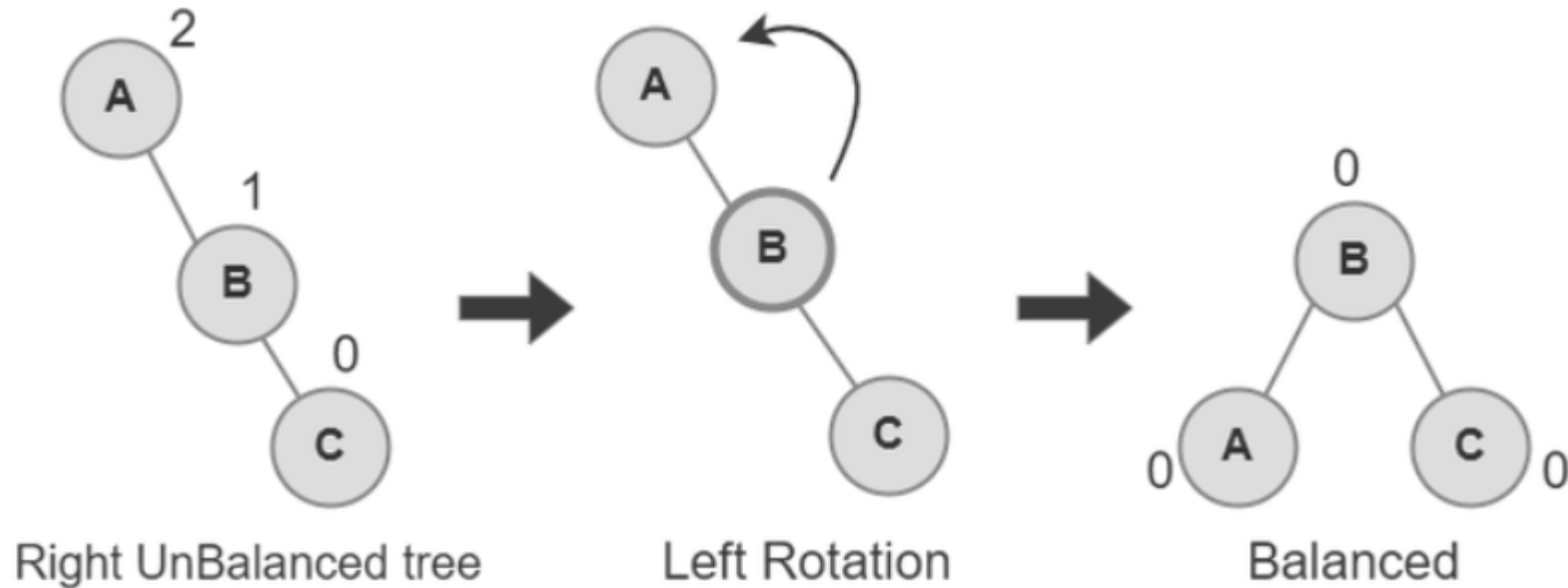
AVL Tree

- AVL tree self-balancing: the difference between heights of left and right subtrees for any node cannot be more than one.
- Advantage of AVL Tree: the time complexities of **search, insert, delete, max, min** become $O(\log_2 N)$. **(Why?)**

AVL Tree Rotation

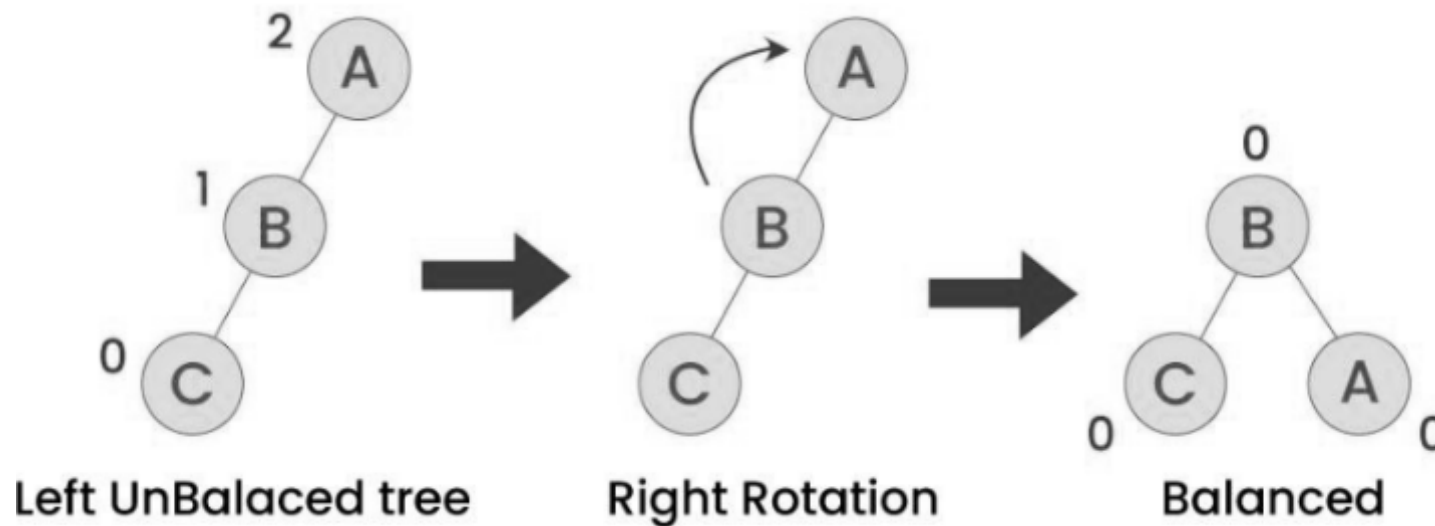
- **Left Rotation**

- add a node into the **right** subtree of the **right** subtree
- if the tree gets out of balance, we do **a single left rotation**



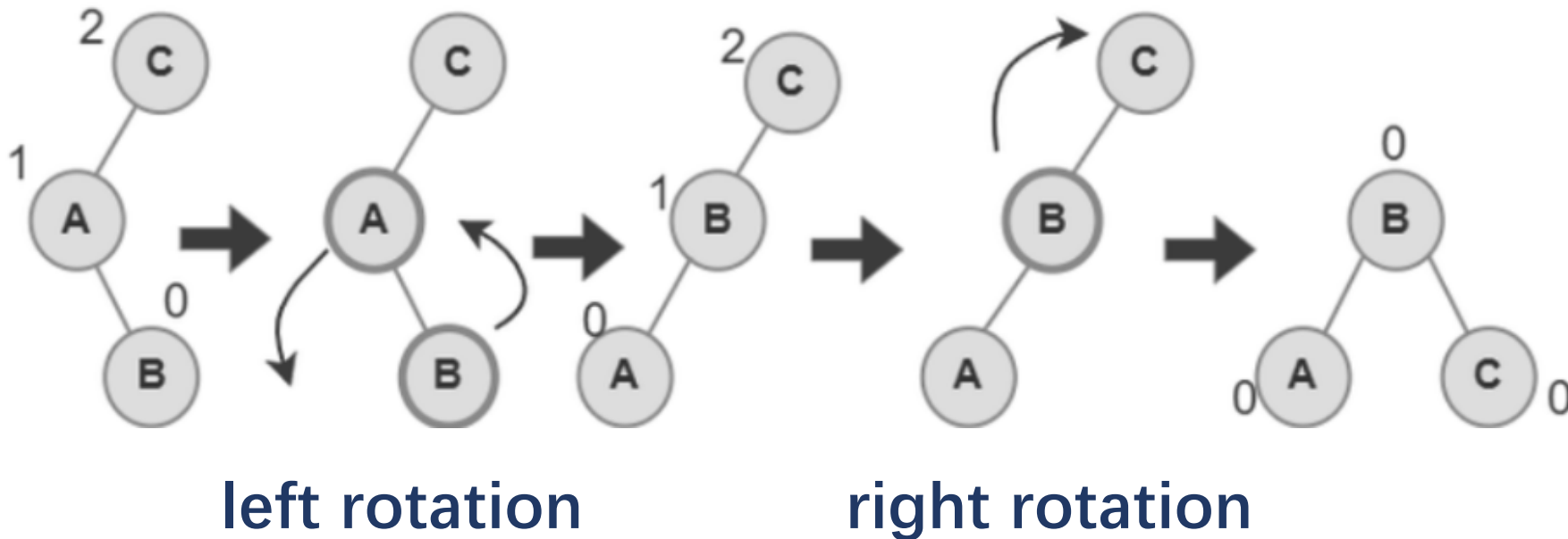
AVL Tree Rotation

- **Right Rotation:**
 - add a node into the **left** subtree of the **left** subtree
 - if the tree gets out of balance, we do **a single right rotation**



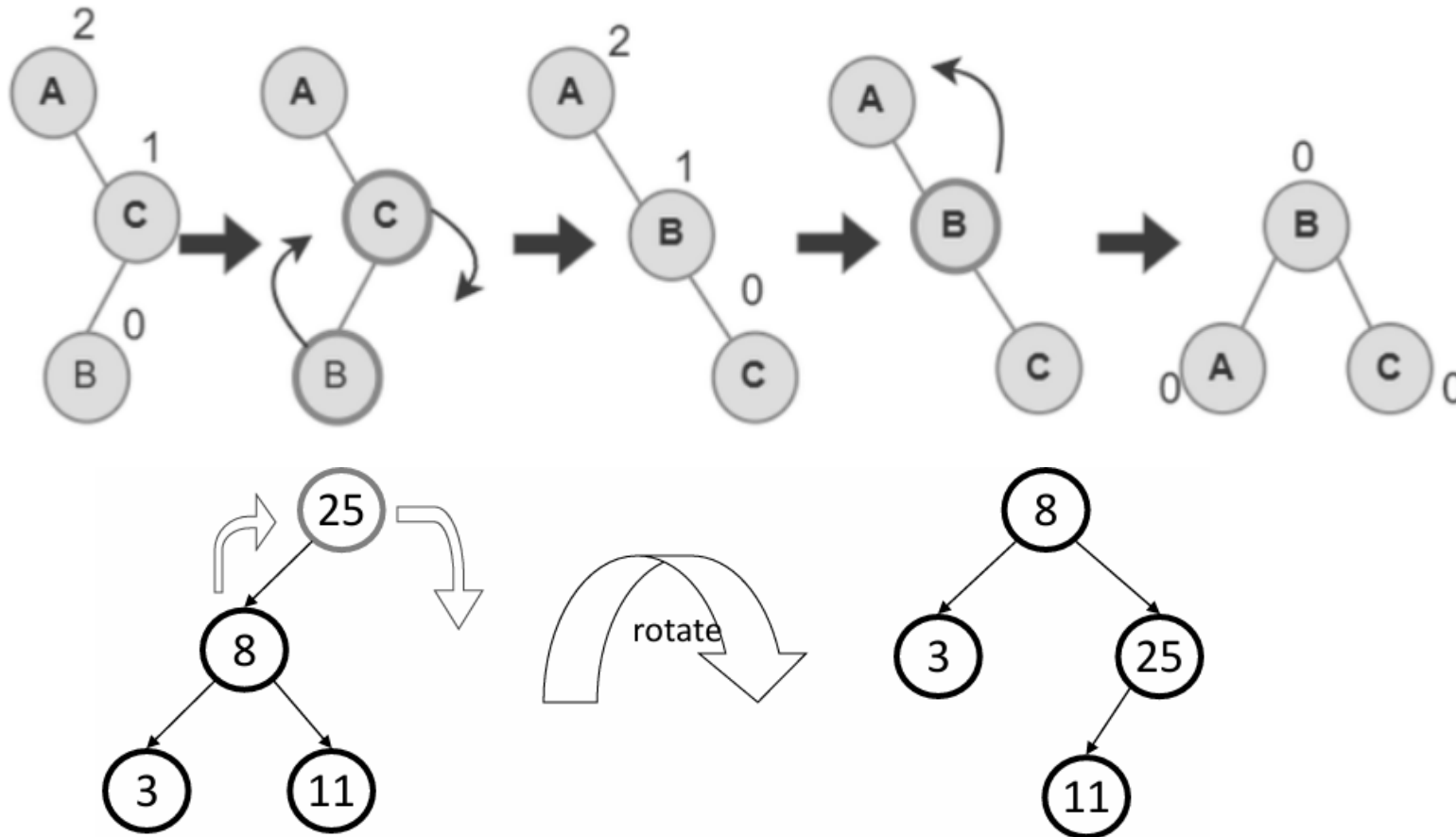
AVL Tree Rotation

- **Left-Right Rotation:**
 - first left rotation, then right rotation



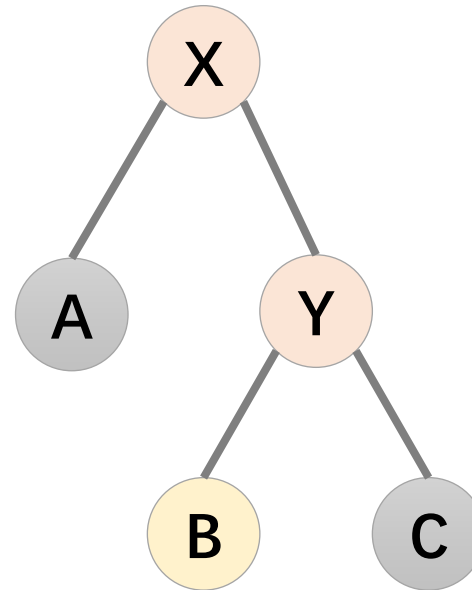
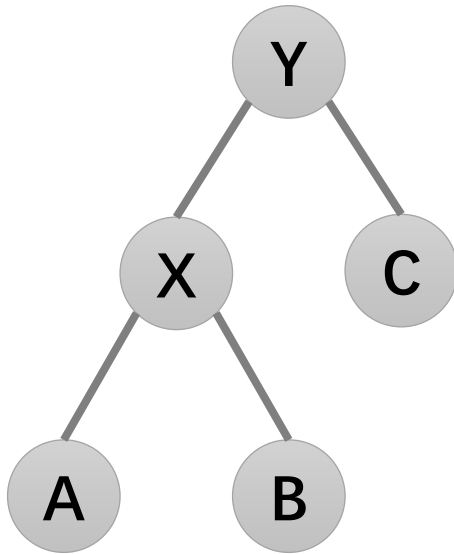
AVL Tree Rotation - Exercise

- What operation is this?



AVL Tree Rotation - Exercise

- Perform right rotation on Node Y



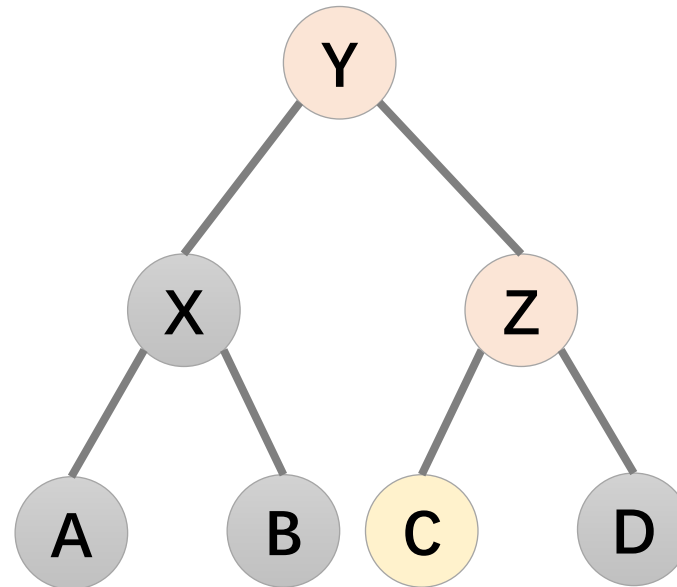
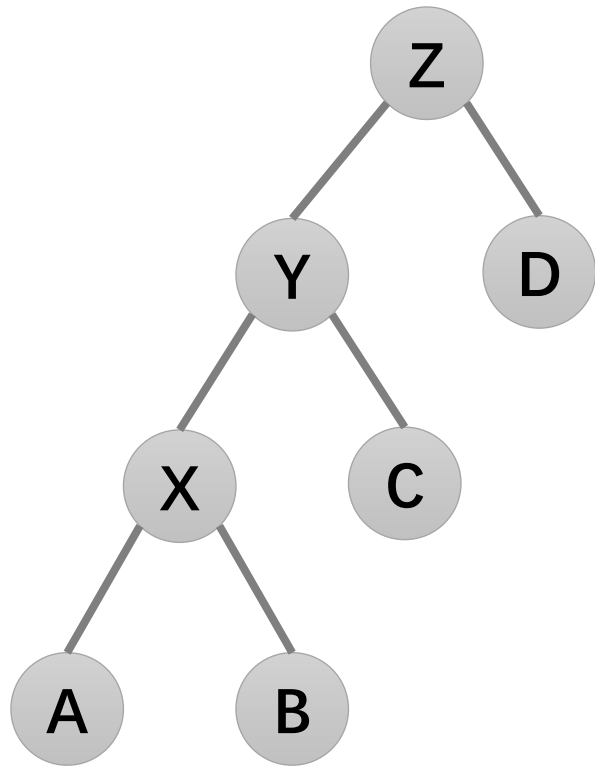
Node *x = y->left;
Node *B = x->right;

x->right = y;
y->left = B;

What operation can restore the original tree?

AVL Tree Rotation - Exercise

- Balance the following tree



Non-BST Binary Trees

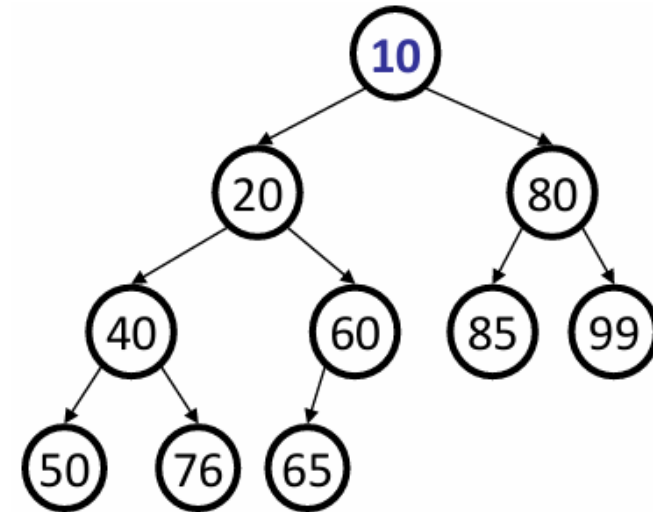
- What if you want to find the k-smallest elements in an unsorted Vector?
 - Find the top 10 students in a class?
- What if you wanted to constantly insert and remove in sorted order?
 - Model a hospital emergency room where individuals are seen in order of their urgency
 - Priority Queue
- What's a good choice?

Non-BST Binary Trees

- Idea: if we use a Vector, it takes a long time to insert or remove in sorted order (or search the Vector for the smallest element)
- If we use a binary search tree, it's fast to insert and remove ($O(\log_2 N)$) but it's slow to find the minimum/maximum element ($O(\log_2 N)$)
- Idea: use a tree, but **store the minimum/maximum element as the root**
 - Trees have $O(\log_2 N)$ insertion/deletion
 - Looking at the root is $O(1)$

Heaps

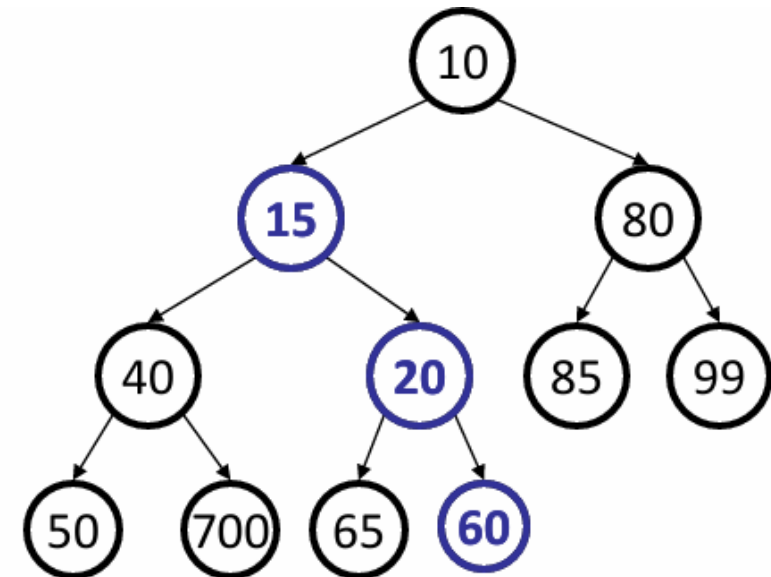
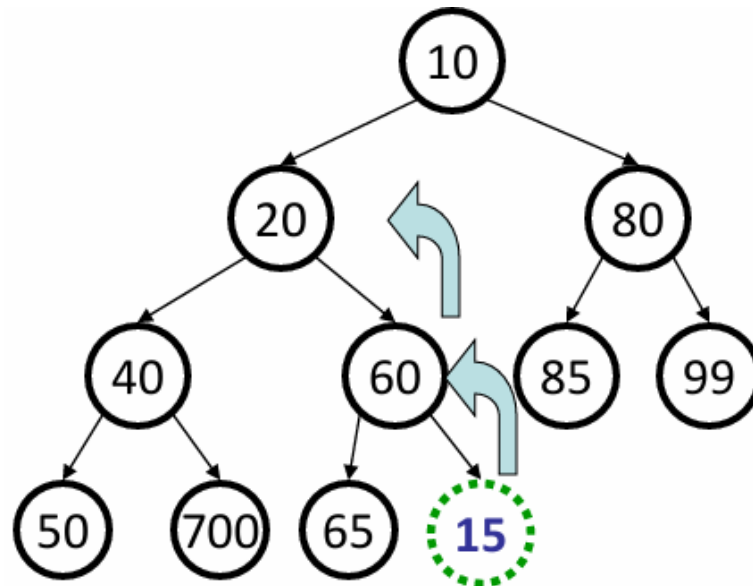
- **Heap:** A **complete binary tree** with vertical ordering:
 - **Min-heap:** all children must be \geq parent's value
 - **Max-heap:** all children must be \leq parent's value
- **Complete tree:** all levels are full of children except perhaps the bottom level, in which all existing nodes are maximally to the left.
(Heaps are always balanced)



a min-heap

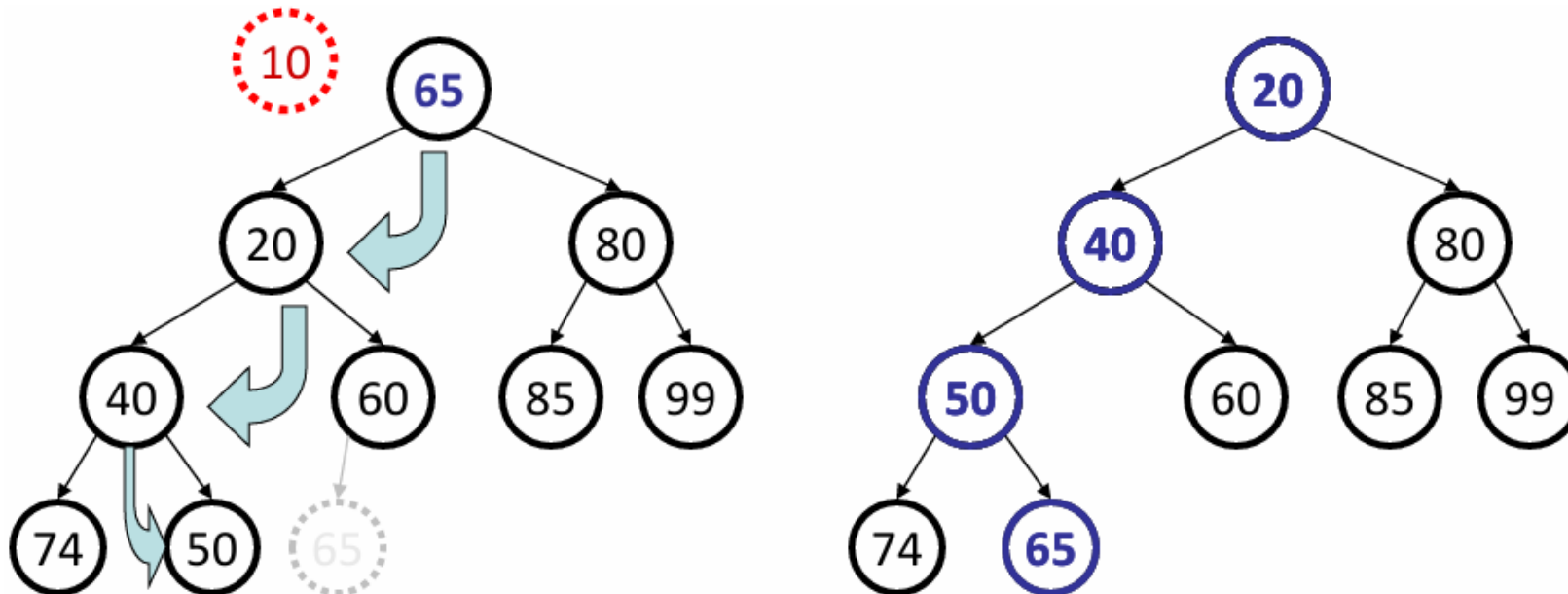
Heap enqueue

- When **adding** to a heap, **the value is first placed at bottom-right**.
- To restore heap ordering, the newly added element is shifted ("bubbled") up the tree until it reaches its proper place (reach the root, or the parent is **smaller** [min-heap]).
- Example: enqueue 15 at bottom-right; bubble up until in order.



Heap dequeue

- Remove the root, and replace it with the **furthest-right** bottom node
- To restore heap order, the improper root is shifted ("bubbled") down the tree by **swapping with its smaller child** [**min-heap**].
- Dequeue min of 10; swap up bottom-right leaf of 65; bubble down.



Heap – Priority Queue

priority_queue

- Elements can be inserted in the priority queue using **push()** method.
- After insertion, priority queue reorganize itself in such a way that the highest priority element is always at the top.

```
#include <queue>
using namespace std;

int main() {
    priority_queue<int> pq;
    pq.push(9);
    pq.push(8);
    pq.push(6);

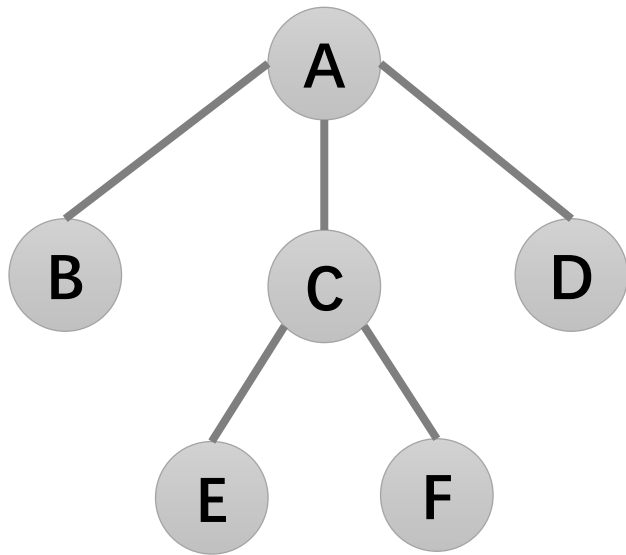
    cout << pq.top();
    return 0;
}
```

9 is at the top

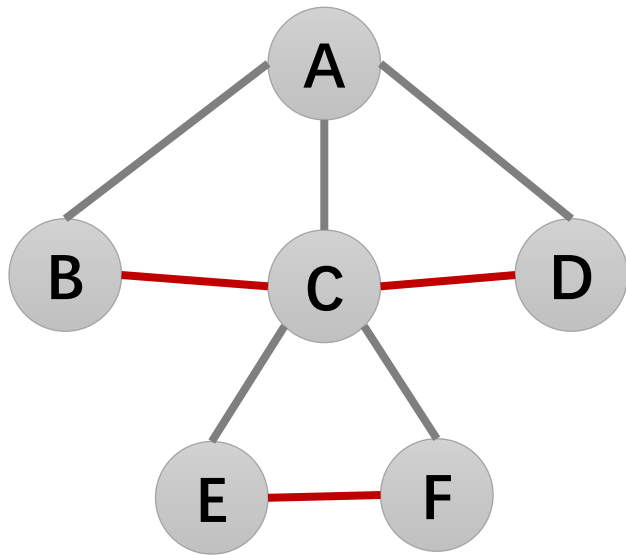
Forests and (Generic) Trees

- In a binary tree, no node has more than two children. This characteristic makes it obvious how to represent and manipulate such trees, and it relates naturally to “divide-and-conquer” algorithms (divide a problem into two subproblems).
- However, sometimes we consider general trees.
- Definition
 - A **tree** is a node (called the root) connected to a set of disjoint **trees**.
 - A **forest** is a set of disjoint **trees**.
- The subtrees of a node are its children; a root node has no parents.

Convert a (Generic) Tree to a Binary Tree

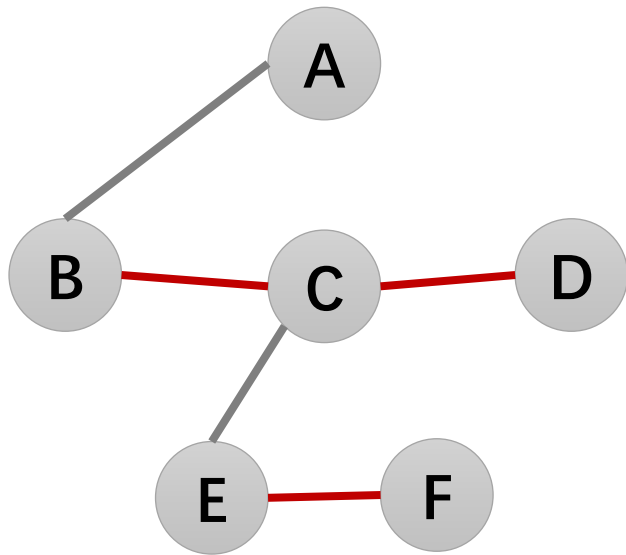


Convert a (Generic) Tree to a Binary Tree



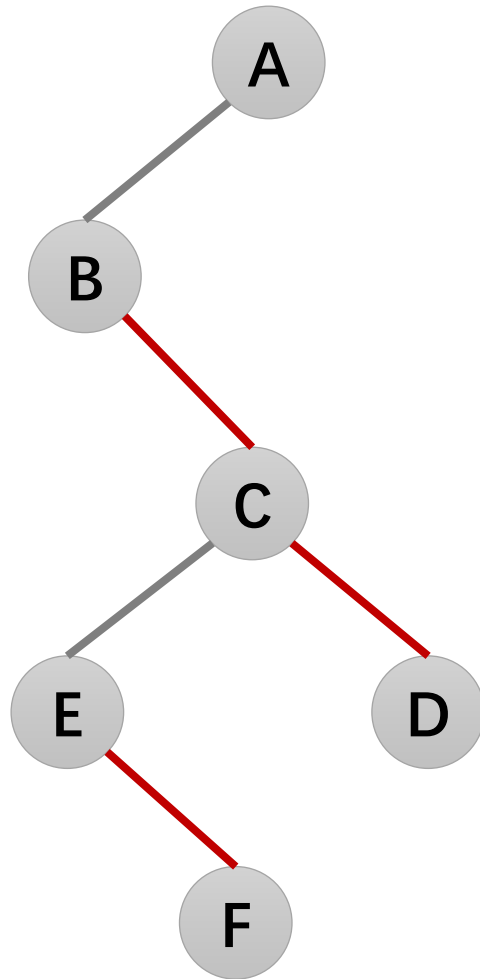
Connect siblings (same level, same parent)

Convert a (Generic) Tree to a Binary Tree



**Disconnect parent and children
except the leftmost one**

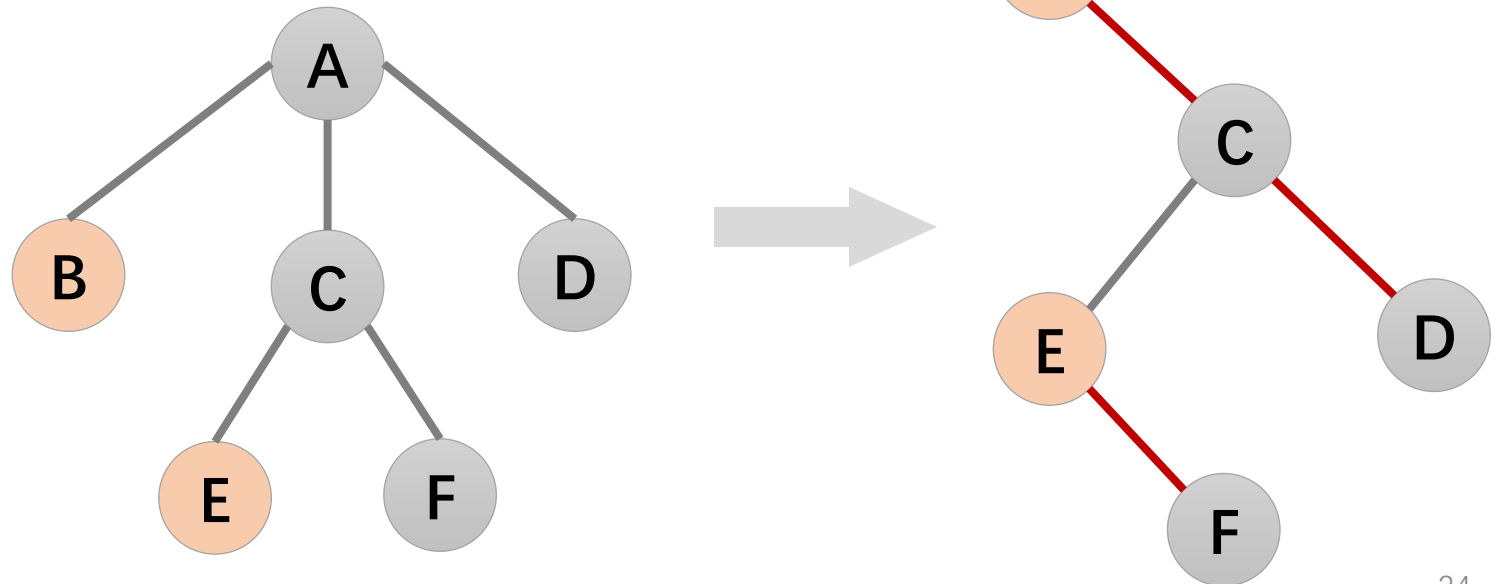
Convert a (Generic) Tree to a Binary Tree



Siblings turn into parents and children

Convert a (Generic) Tree to a Binary Tree

- The root of the **Binary Tree** is the root of the Generic Tree.
- The **left child** of a node in the Generic Tree is the **left child** of that node in the Binary Tree.
- The right sibling of any node in the Generic Tree is the right child of that node in the Binary Tree.



Huffman Trees and Coding

- Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.
- Compress the data in which there are **frequently occurring characters**.

B C A A D D D C C A C A C A C

If each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of $8 * 15 = 120$ bits are required to send this string.

Huffman Coding can compress the string to a smaller size!

Huffman Trees and Coding

- Compress the data in which there are **frequently occurring characters**.

B	C	A	A	D	D	D	C	C	A	C	A	C	A	C
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.

Calculate the **frequency** of each character in the string.

A	B	C	D
5	1	6	3

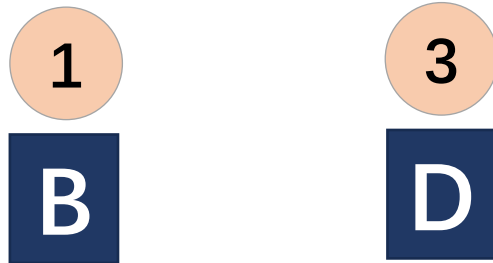
Huffman Trees and Coding

- Sort the characters in **increasing order of the frequency**. These are stored in a priority queue Q.

B	D	A	C
1	3	5	6

Huffman Trees and Coding

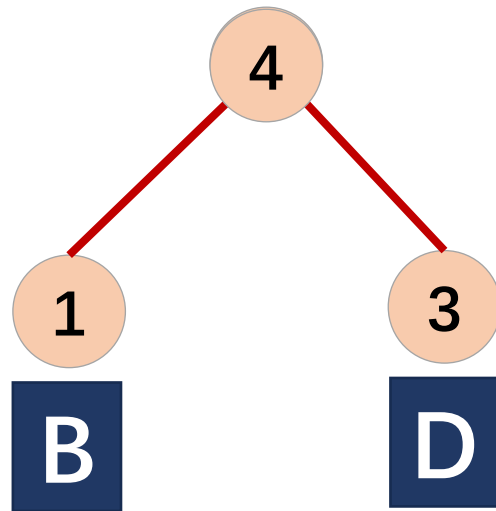
- Make each unique character as a leaf node.



		A	C
		5	6

Huffman Trees and Coding

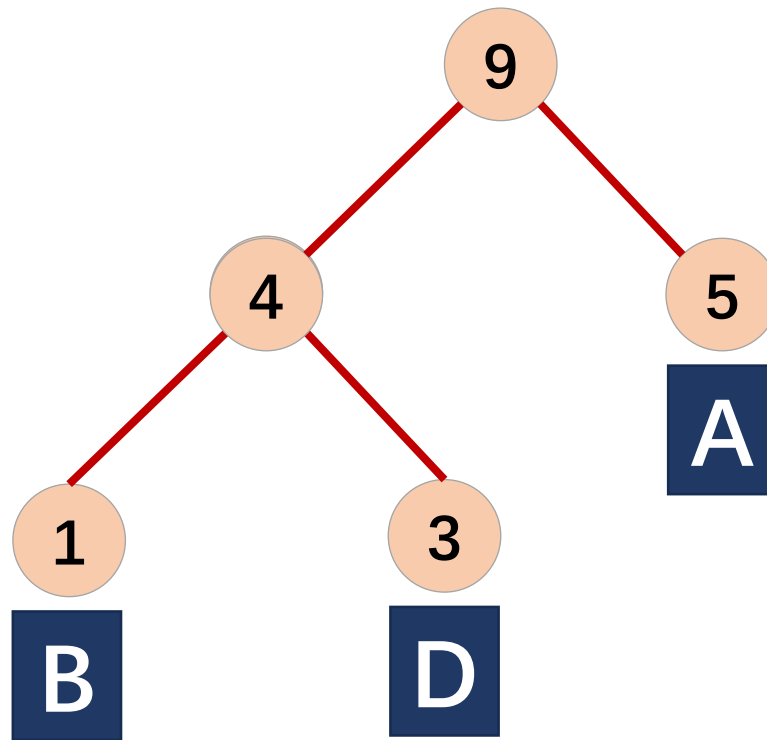
- Create an empty node E. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the E as the sum of the above two minimum frequencies.
- Remove these two minimum frequencies from Q and add the sum into the list of frequencies; Insert node E into the tree.



		A	C
	4	5	6

Huffman Trees and Coding

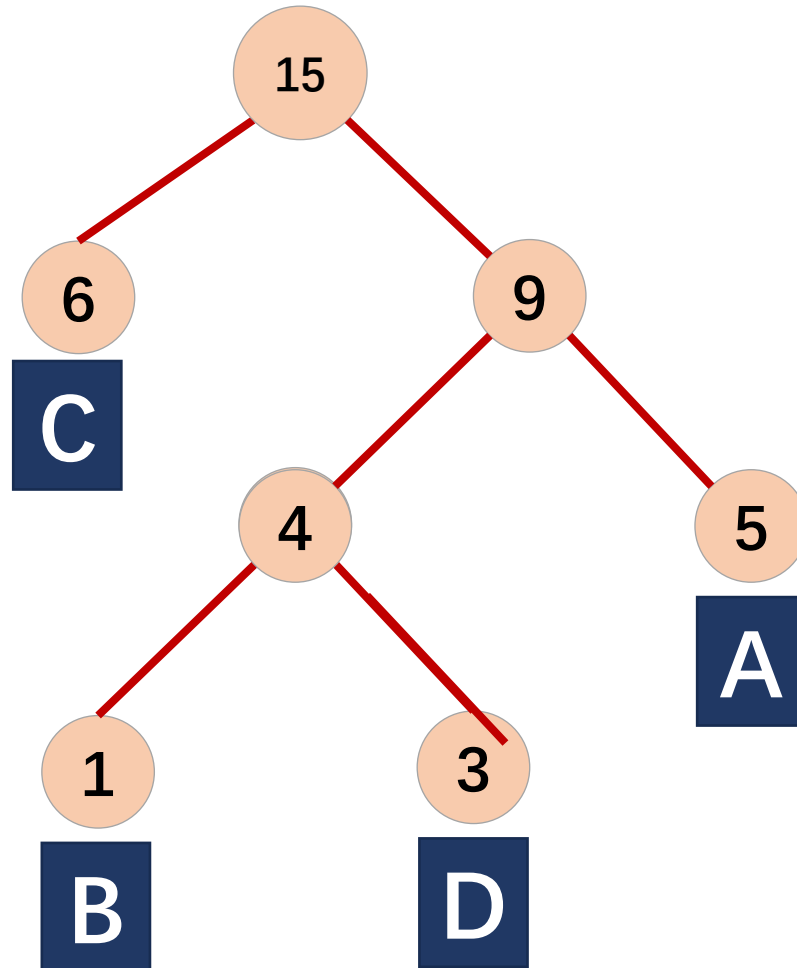
- Repeat steps for all remaining characters



			C
		9	6

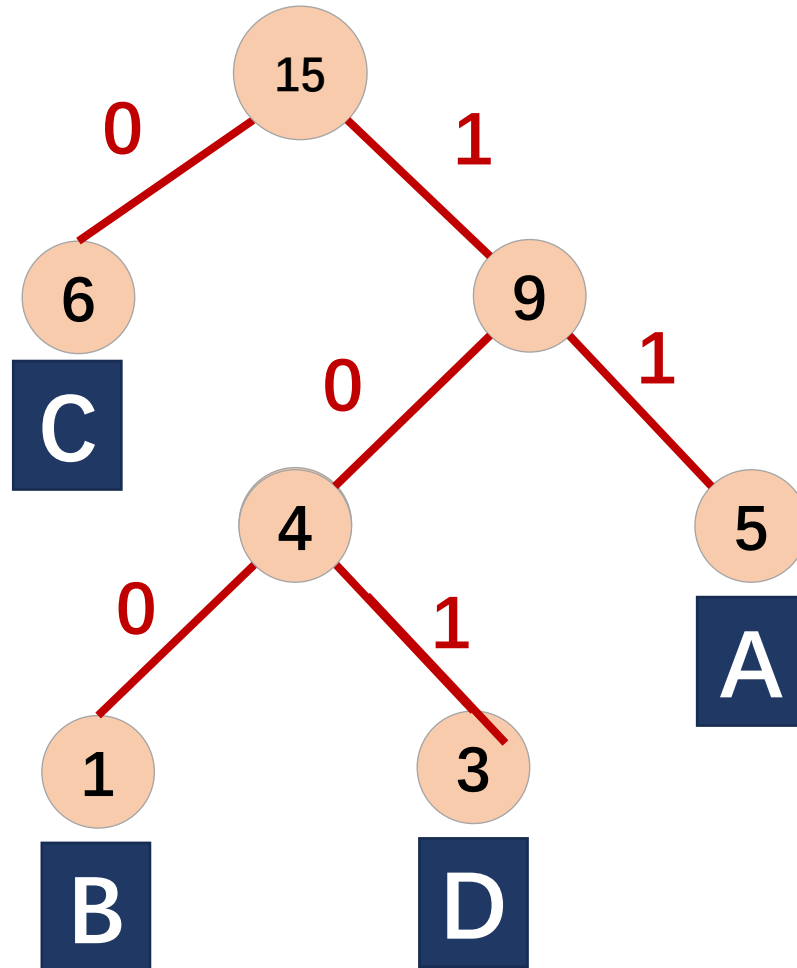
Huffman Trees and Coding

- Repeat steps for all remaining characters



Huffman Trees and Coding

- For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



A	B	C	D
11	100	0	101

Huffman Coding

Huffman Trees and Coding

- For each non-leaf node, assign 0 to the left edge and 1 to the right edge.

Frequency	5	1	6	3	
Character	A	B	C	D	= 4*8 bits
Huffman Coding	11	100	0	101	
Coding Size	5*2	1*3	6*1	3*3	= 28 bits

Huffman Trees and Coding

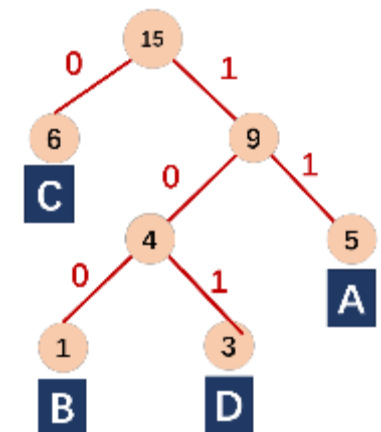
B C A A D D D C C A C A C A C



A	B	C	D
11	100	0	101

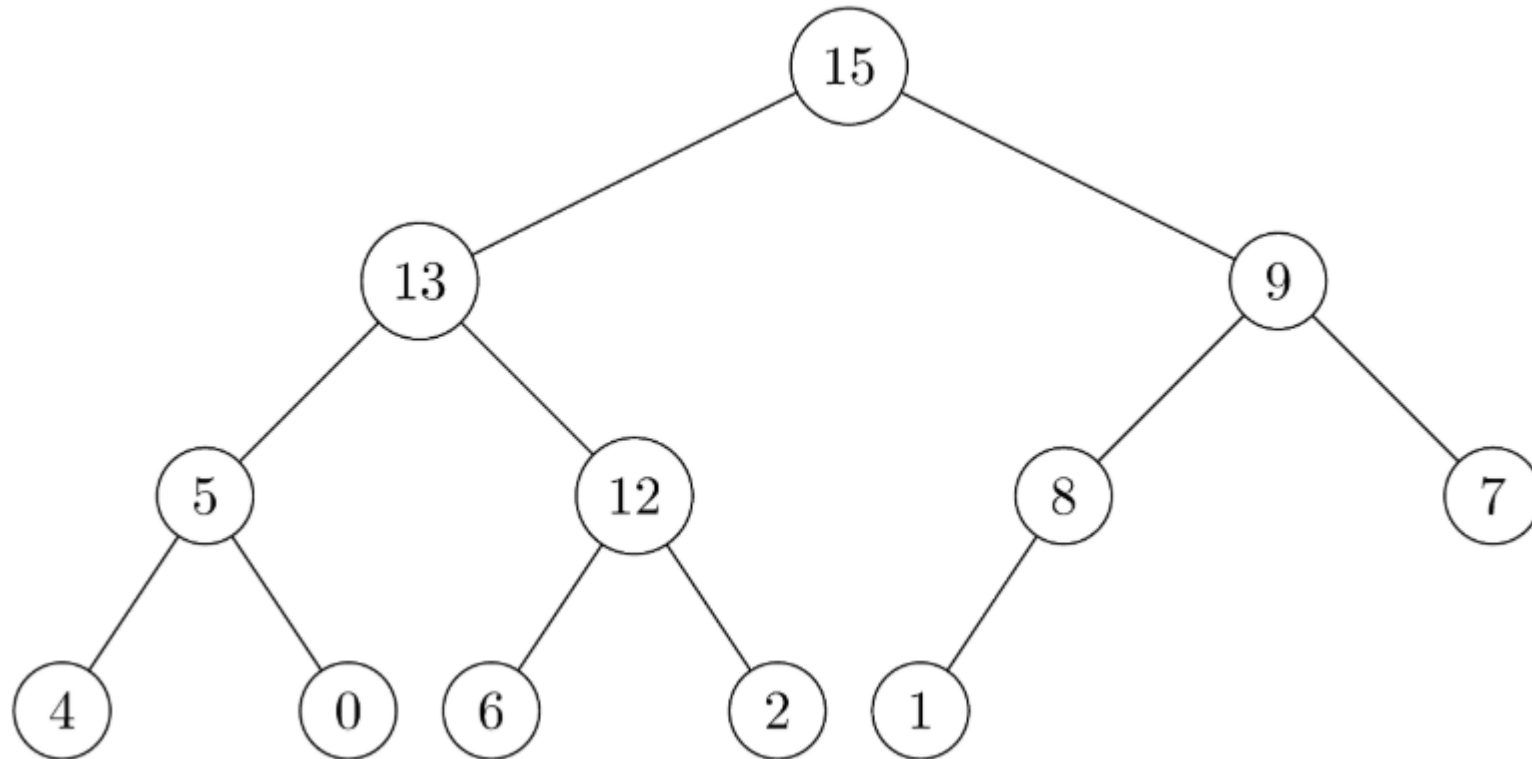
1000111110110110100110110110

(For **decoding the code**, we can take the code and traverse through the tree to find the character)



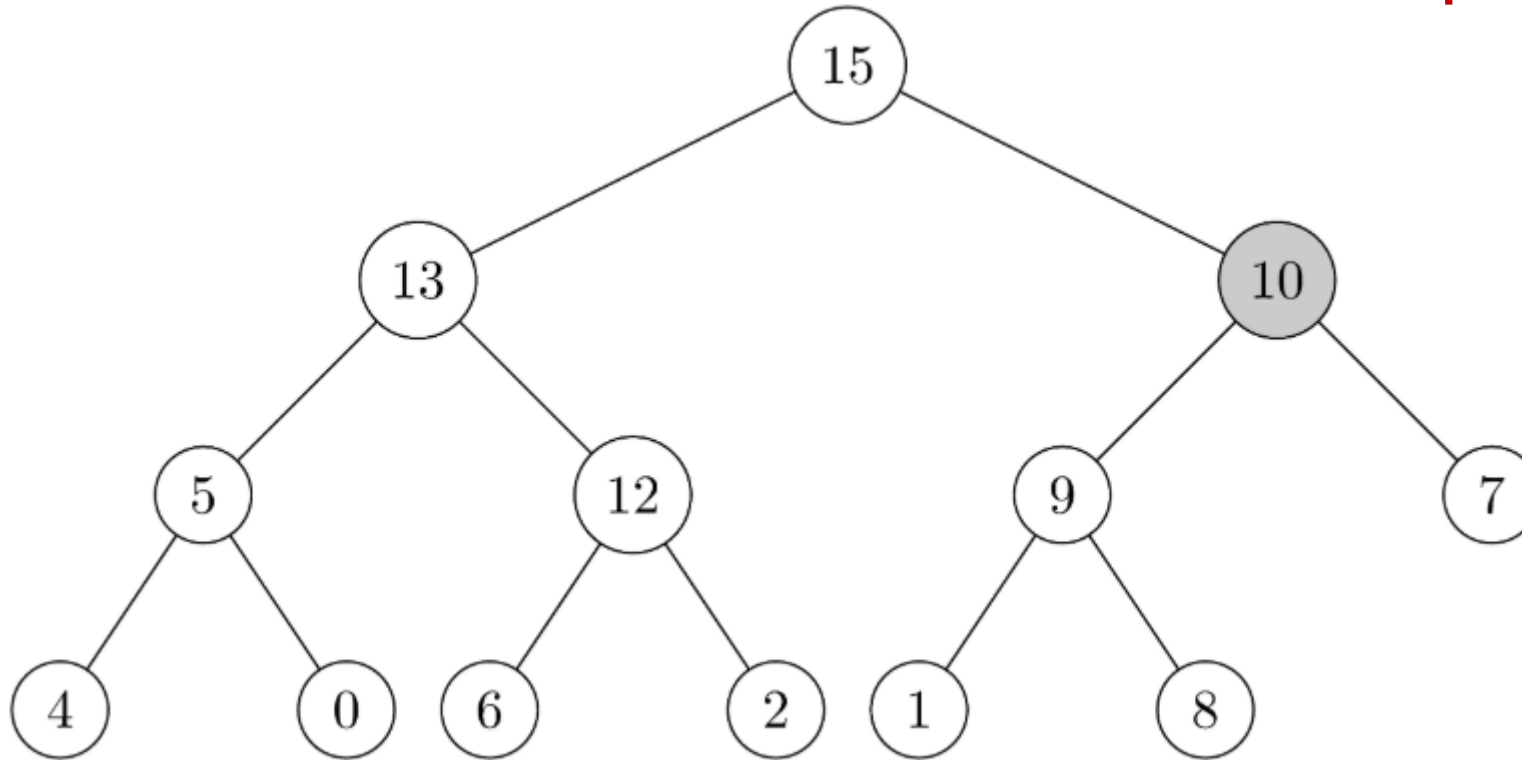
In-Class Exercise: Max Heap

- Given a max heap, enqueue a node with value of 10



In-Class Exercise: Max Heap

Then, how about **dequeue** 15?



In-Class Exercise: C++ Max Heap

- Given an array of CPU tasks, each labeled with a letter from A to Z, and a number n.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z								

- Each CPU interval can be idle or allow the completion of one task.
- Tasks can be completed in any order, but there's a constraint: there has to be **a gap of at least n intervals between two tasks with the same label.**
- Return the **minimum** number of CPU intervals required to complete all tasks.

For example: Input: tasks = ["A","A","A","B","B","B"], n = 2

Answer (Output): 8

count	1	2	3	4	5	6	7	8
task	A	B		A	B		A	B

Explanation: A possible sequence is: A -> B -> idle -> A -> B -> idle -> A -> B.

After completing task A, you must wait two intervals before doing A again. The same applies to task B. In the 3rd interval, neither A nor B can be done, so you idle. By the 4th interval, you can do A again as 2 intervals have passed.


In-Class Exercise: C++ Max Heap

For another example: Input: tasks = ["A","C","A","B","D","B"], n = 1

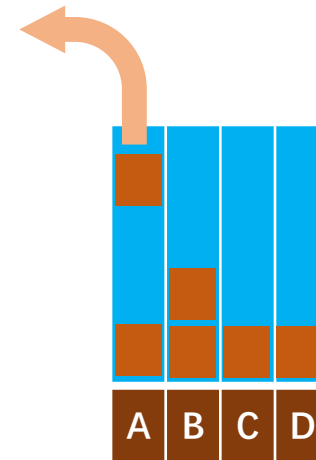
Answer (Output): 6

A possible sequence is:

count	1	2	3	4	5	6
task	A	B	C	D	A	B


cycle = 2

Task cooldown time count n = 1



In-Class Exercise: C++ Max Heap

// Given an array of CPU tasks, each labeled with a letter from A to Z, and a number n.

```
int leastInterval(vector<char>& tasks, int n) {
```

```
    // Implement this function
```

```
}
```

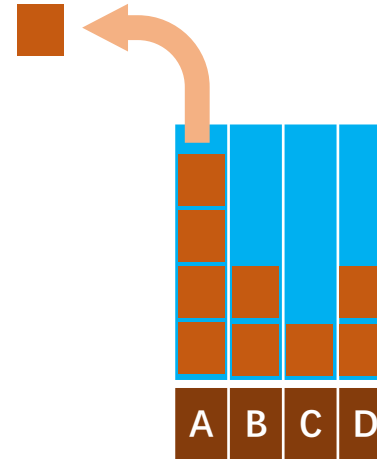
Another example:

count	1	2	3	4	5	6
task	A	B	C	D	A	B



cycle = 4

Task cooldown time count n = 3



In-Class Exercise: C++ Max Heap

```
int leastInterval(vector<char>& tasks, int n) {  
    // Building frequency map  
    int freq[26] = {0};  
    for (char ch : tasks) {  
        freq[ch - 'A']++;  
    }  
  
    // Max heap to store frequencies  
    priority_queue<int> pq;  
    for (int i = 0; i < 26; i++) {  
        if (freq[i] > 0) {  
            pq.push(freq[i]);  
        }  
    }  
}
```

//(Continue on the next slide...)

In-Class Exercise: C++ Max Heap

```
int time = 0;
while (!pq.empty()) { // Process tasks until the heap is empty
    int cycle = n + 1;
    vector<int> store; // Temporarily Store updated task numbers
    int taskCount = 0;
    while (cycle && !pq.empty()) { // Execute tasks in each cycle
        if (pq.top() > 1) {
            store.push_back(pq.top() - 1); // Temporarily Store updated task numbers
        }
        pq.pop();
        taskCount++;
        cycle--;
    }
    for (int x : store) { // Restore updated frequencies to the heap
        pq.push(x);
    }
    time += (pq.empty() ? taskCount : n + 1); // Add time for the completed cycle
}
return time;
};
```

Exercise Analysis

1008. Construct Binary Search Tree from Preorder Traversal

Medium

Topics

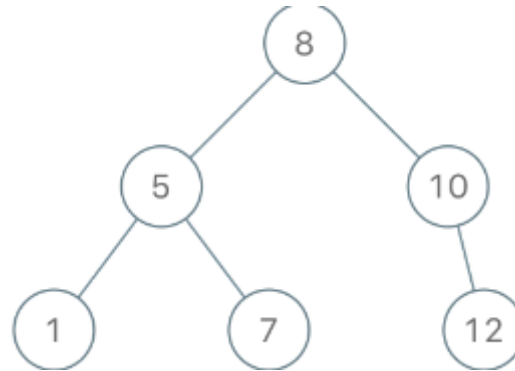
Companies

Given an array of integers `preorder`, which represents the **preorder traversal** of a BST (i.e., **binary search tree**), construct the tree and return *its root*.

It is **guaranteed** that there is always possible to find a binary search tree with the given requirements for the given test cases.

A **binary search tree** is a binary tree where for every node, any descendant of `Node.left` has a value **strictly less than** `Node.val`, and any descendant of `Node.right` has a value **strictly greater than** `Node.val`.

A **preorder traversal** of a binary tree displays the value of the node first, then traverses `Node.left`, then traverses `Node.right`.



Input: `preorder = [8,5,1,7,10,12]`

Output: `[8,5,10,1,7,null,12]`

Exercise Analysis

```
class Solution {
public:
    TreeNode* bstFromPreorder(vector<int>& preorder) {
        int i = 0;
        return build(preorder, i, INT_MAX);
    }

    TreeNode* build(vector<int>& preorder, int& i, int bound){
        if(i == preorder.size() || preorder[i] > bound) return nullptr;
            (if array ends)           (if i-th node is bigger than bound)

        TreeNode* root=new TreeNode(preorder[i++]); // Create a node then increase i

        root->left = build(preorder, i, root->val); // Any left must be smaller than root

        root->right = build(preorder, i, bound); // Any right must be smaller than parent

        return root;
    }
};
```

Exercise 9.1

- Complete [LeetCode 617](#)

617. Merge Two Binary Trees

Easy Topics Companies

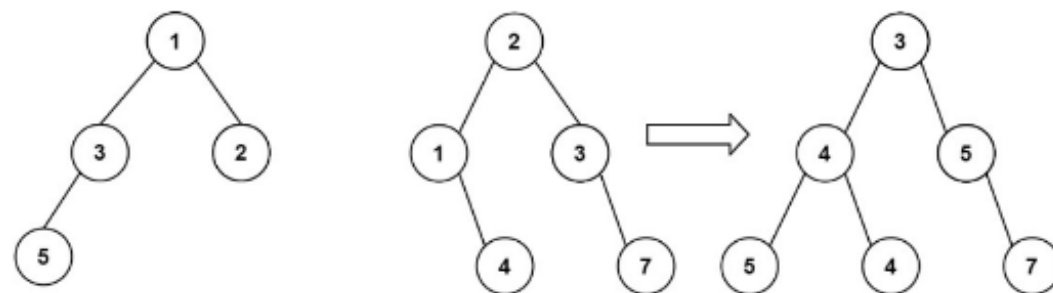
You are given two binary trees `root1` and `root2`.

Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.

Return the merged tree.

Note: The merging process must start from the root nodes of both trees.

Example 1:



Input: `root1 = [1,3,2,5]`, `root2 = [2,1,3,null,4,null,7]`

Output: `[3,4,5,5,4,null,7]`

Exercise 9.2

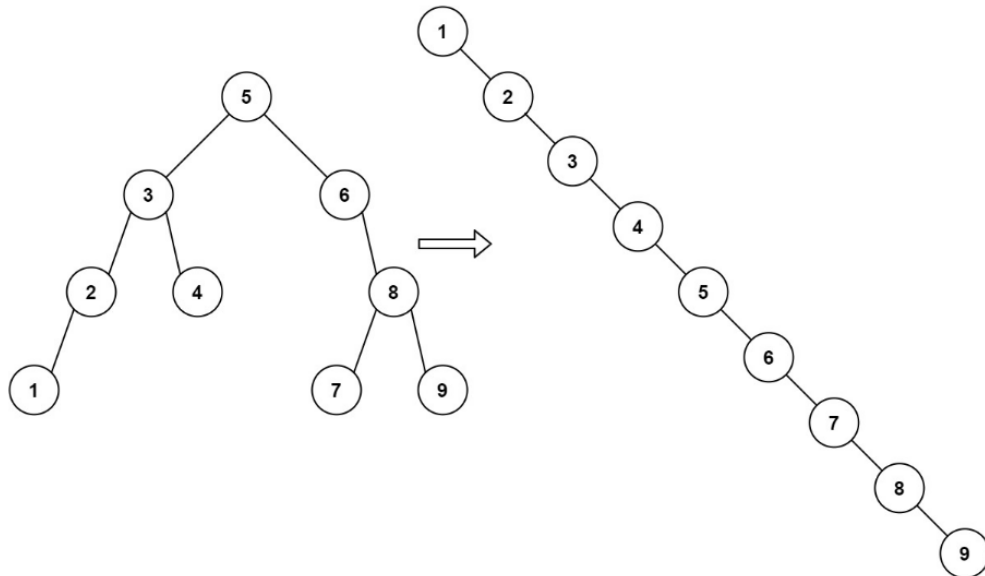
- Complete [LeetCode 897](#)

897. Increasing Order Search Tree

Easy Topics Companies

Given the `root` of a binary search tree, rearrange the tree in **in-order** so that the leftmost node in the tree is now the root of the tree, and every node has no left child and only one right child.

Example 1:



Input: `root = [5,3,6,2,4,null,8,1,null,null,null,7,9]`

Output: `[1,null,2,null,3,null,4,null,5,null,6,null,7,null,8,null,9]`

Exercise 9.3

- Complete [LeetCode 1845](#)

1845. Seat Reservation Manager

Medium

Topics

Companies

Hint

Design a system that manages the reservation state of n seats that are numbered from 1 to n .

Implement the `SeatManager` class:

- `SeatManager(int n)` Initializes a `SeatManager` object that will manage n seats numbered from 1 to n . All seats are initially available.
- `int reserve()` Fetches the **smallest-numbered** unreserved seat, reserves it, and returns its number.
- `void unreserve(int seatNumber)` Unreserves the seat with the given `seatNumber`.

`priority_queue<int, vector<int>, greater<int>> minHeap;` [// How you can create a min heap in C++](#)

Exercise 9.4

- Complete [LeetCode 1046](#) (Use Priority Queue)

1046. Last Stone Weight

Easy Topics Companies Hint

You are given an array of integers `stones` where `stones[i]` is the weight of the i^{th} stone.

We are playing a game with the stones. On each turn, we choose the **heaviest two stones** and smash them together. Suppose the heaviest two stones have weights x and y with $x \leq y$. The result of this smash is:

- If $x == y$, both stones are destroyed, and
- If $x \neq y$, the stone of weight x is destroyed, and the stone of weight y has new weight $y - x$.

At the end of the game, there is **at most one** stone left.

Return the weight of the last remaining stone. If there are no stones left, return 0.

Example 1:

Input: `stones = [2,7,4,1,8,1]`

Output: 1

Explanation:

We combine 7 and 8 to get 1 so the array converts to `[2,4,1,1,1]` then,
we combine 2 and 4 to get 2 so the array converts to `[2,1,1,1]` then,
we combine 2 and 1 to get 1 so the array converts to `[1,1,1]` then,
we combine 1 and 1 to get 0 so the array converts to `[1]` then that's the value of the last stone.

Exercise 9.5

- Complete [LeetCode 110](#)

110. Balanced Binary Tree

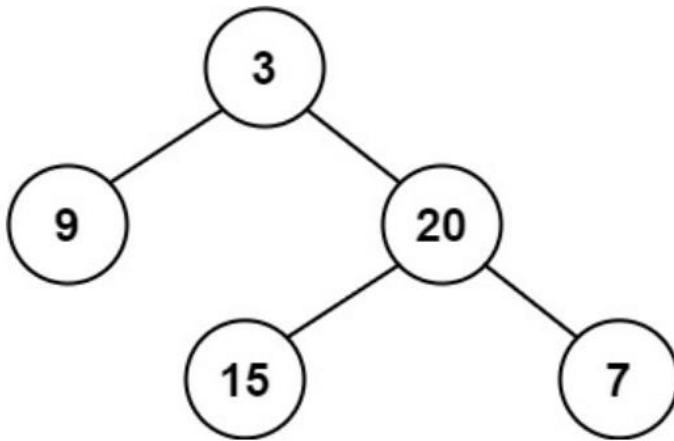
Easy

Topics

Companies

Given a binary tree, determine if it is **height-balanced**.

Example 1:

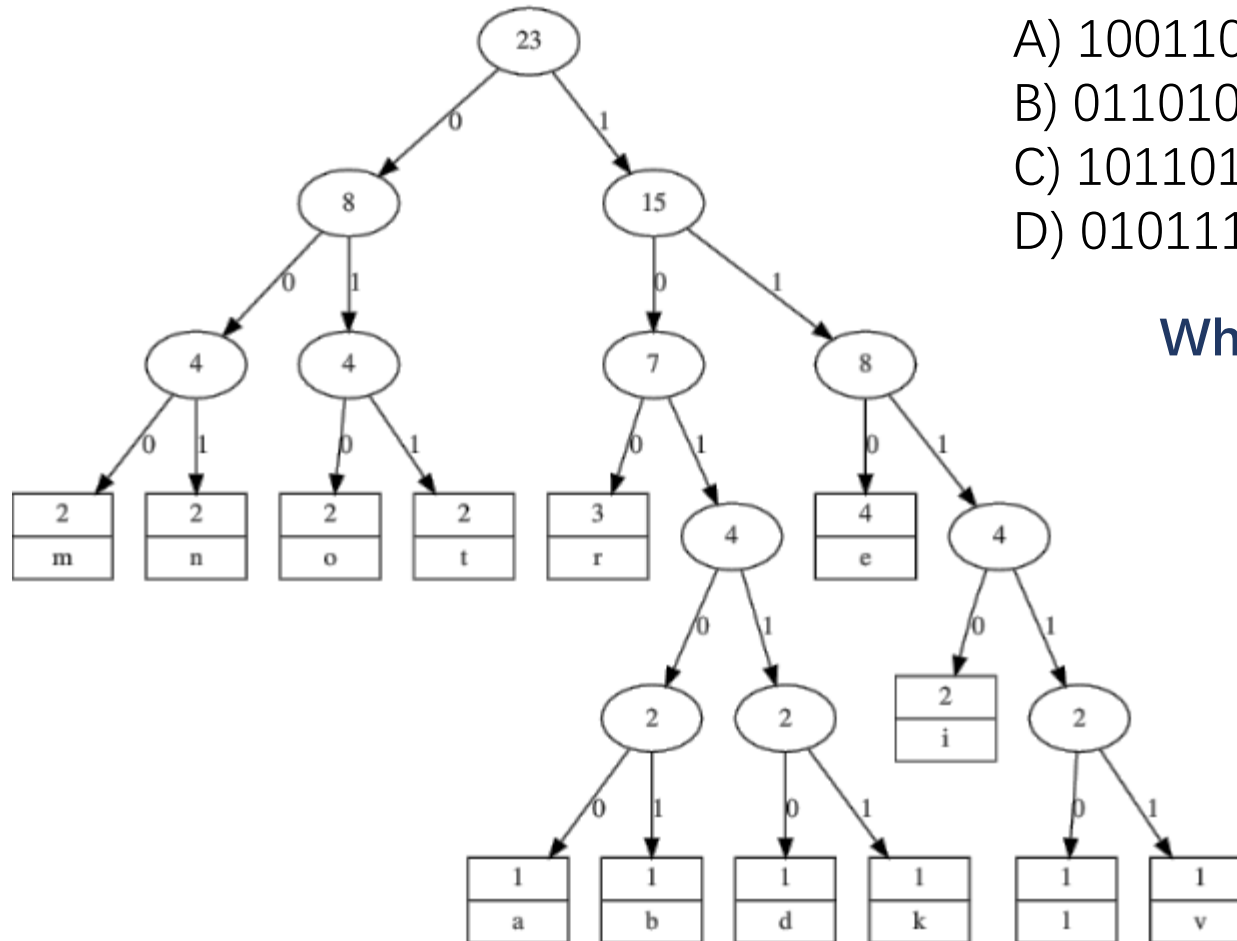


Input: root = [3,9,20,null,null,15,7]

Output: true

Exercise 9.6

- Use the following Huffman tree to decode the four binary sequences below:



A) 10011000011000010101110100

B) 011010

C) 10110100111000110111

D) 0101111110100111100111110001110

What is the message?

Exercise 9.7

• You May Try LeetCode 1382

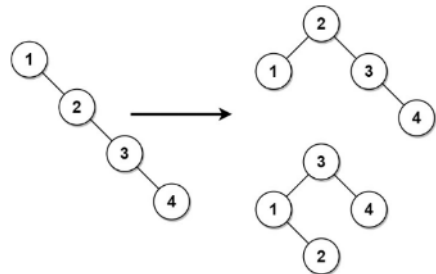
1382. Balance a Binary Search Tree

Medium Topics Companies Hint

Given the `root` of a binary search tree, return a **balanced** binary search tree with the same node values. If there is more than one answer, return **any** of them.

A binary search tree is **balanced** if the depth of the two subtrees of every node never differs by more than 1.

Example 1:



Input: root = [1,null,2,null,3,null,4,null,null]

Output: [2,1,3,null,null,4,null]

Explanation: This is not the only correct answer, [3,1,4,null,2] is also correct.

Easy Solution: Inorder Traversal + Recursive Construction

Hard Solution: In-Place Balancing

- **Step 1:** Transforms the BST into a right-skewed tree (resembling a linked list). This is achieved through a series of right rotations. The process involves traversing the tree and performing a right rotation whenever a node with a left child is encountered, continuing until the entire tree is right-skewed.
- **Step 2:** Determine the total number of nodes N , by traversing the right-skewed tree and counting each node.
- **Step 3:** Convert the right-skewed tree into a balanced BST through a series of left rotations.
- **Step 4:** Calculate $M = \text{pow}(2, \text{floor}(\log_2(N + 1))) - 1$;
- **Step 5:** Perform $(N-M)$ left rotations to partially balance the tree. **This ensures that the remaining nodes will form a complete binary tree.**
- **Step 6:** Enter a loop where m is halved repeatedly. For each iteration, perform left rotations to balance the next level of the tree. This process continues until fully transformed into a balanced BST.