Dictionaries I

Data Structures and Parallelism

Logistics

New project is coming, which means new (or old) partnerships

- Email me if you *DON'T* want to use the same partnership as P1
- I'm planning to post the spec early
- Repos will be made on Friday night

Midterm is Friday next week, will cover material up through coming Monday (Separate Chaining Hash Tables)

- Full topic list will be posted soon

Outline

Two new (old?) ADTs

- Dictionaries
- Sets

Review BSTs

Intro AVL trees

Our Next ADT

Dictionary ADT

state

Set of (key, value) pairs

behavior

insert(key, value) – inserts (key, value) pair. If key was already in dictionary, overwrites the previous value.

find(key) – returns the stored value associated with key.

delete(key) – removes the key and its value from the dictionary.

Real world intuition:
keys: words
values: definitions

Dictionaries are often called "maps"

Our Next ADT

Set ADT
state Set of elements **behavior** **insert(element)** – inserts element into the set. **find(element)** – returns true if element is in the set, false otherwise. **delete(key)** – removes the key and its value from the dictionary.

Usually implemented as a dictionary with values “true” or “false”

Later in the course we’ll want more complicated set operations like **union(set1, set2)**

Uses of Dictionaries

Dictionaries show up all the time.

There are too many applications to really list all of them:

- Phonebooks
- Indexes
- Databases
- Operating System memory management
- The internet (DNS)
- …

Any time you want to organize information for easy retrieval.

We're going to design three *completely different* implementations of Dictionaries – they have that many different uses.

Simple Dictionary Implementations

	Insert	Find	Delete
Unsorted Linked List			
Unsorted Array			
Sorted Linked List			
Sorted Array			

What are the worst case running times for each operation if you have n (key, value) pairs.

Assume the arrays do not need to be resized.

Think about what happens if a repeat key is inserted!

Simple Dictionary Implementations

	Insert	Find	Delete
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$

What are the worst case running times for each operation if you have n (key, value) pairs.
Assume the arrays do not need to be resized.
Think about what happens if a repeat key is inserted!

Aside: Lazy Deletion

Lazy Deletion: A general way to make delete() more efficient.

Don't remove the entry from the structure, just "mark" it as deleted.

Benefits:

- Much simpler to implement
- More efficient (no need to shift values on every single delete)

Drawbacks:

- Extra space:
  - For the flag
  - More drastically, data structure grows with all insertions, not with the current number of items.
- Sometimes makes other operations more complicated.

Simple Dictionary Implementations

	Insert	Find	Delete
Unsorted Linked List	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$
Unsorted Array	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$
Sorted Linked List	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$
Sorted Array	$\Theta(m)$	$\Theta(\log m)$	$\Theta(\log m)$

We can do slightly better with lazy deletion, let m be the total number of elements ever inserted (even if later lazily deleted)

Think about what happens if a repeat key is inserted!

A Better Implementation

What about BSTs?

Keys will have to be comparable…

	Insert	Find	Delete
Average			
Worst			

A Better Implementation

What about BSTs?

Keys will have to be comparable...

	Insert	Find	Delete
Average	$\Theta(\log n)$	$\Theta(\log n)$	
Worst	$\Theta(n)$	$\Theta(n)$	

Let's talk about how to implement delete.

Deletion from BSTs

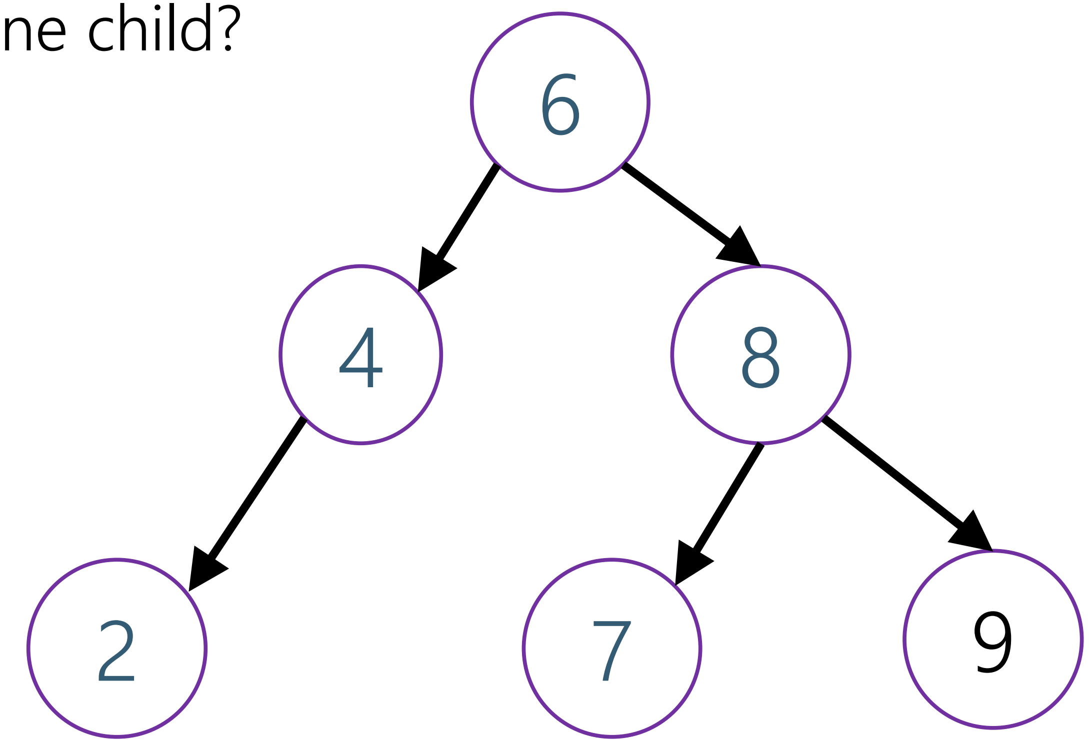
Deleting will have three steps:

- Finding the element to delete
- Removing the element
- Restoring the BST property

Deletion – Easy Cases

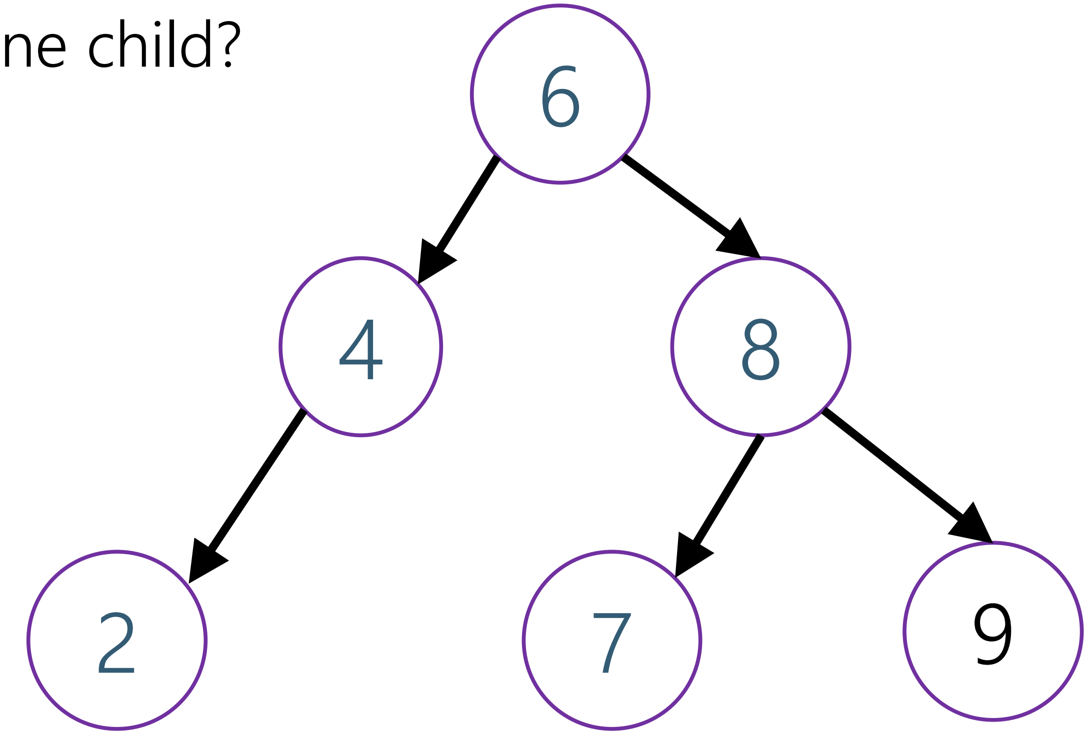
What if the elements to delete is:

- A leaf?
- Has exactly one child?



Deletion – Easy Cases

What if the elements to delete is:
- A leaf?
- Has exactly one child?

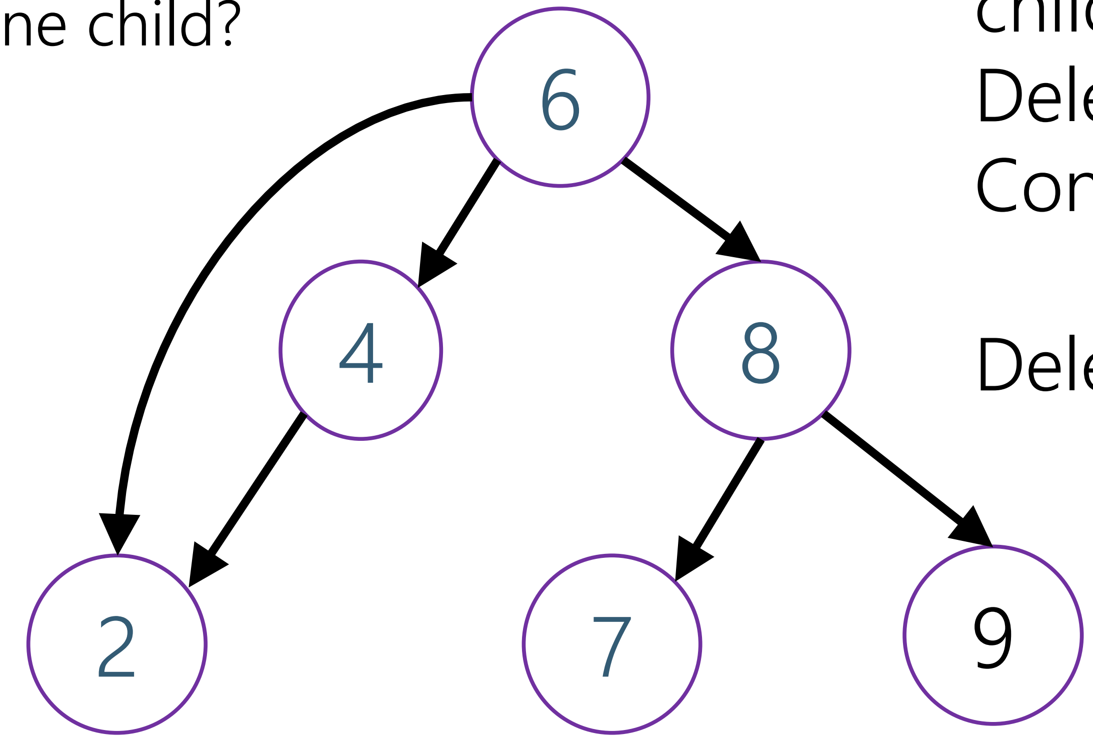


Deleting a leaf:
Just get rid of it.

Delete(7)

Deletion – Easy Cases

What if the elements to delete is:
- A leaf?
- Has exactly one child?

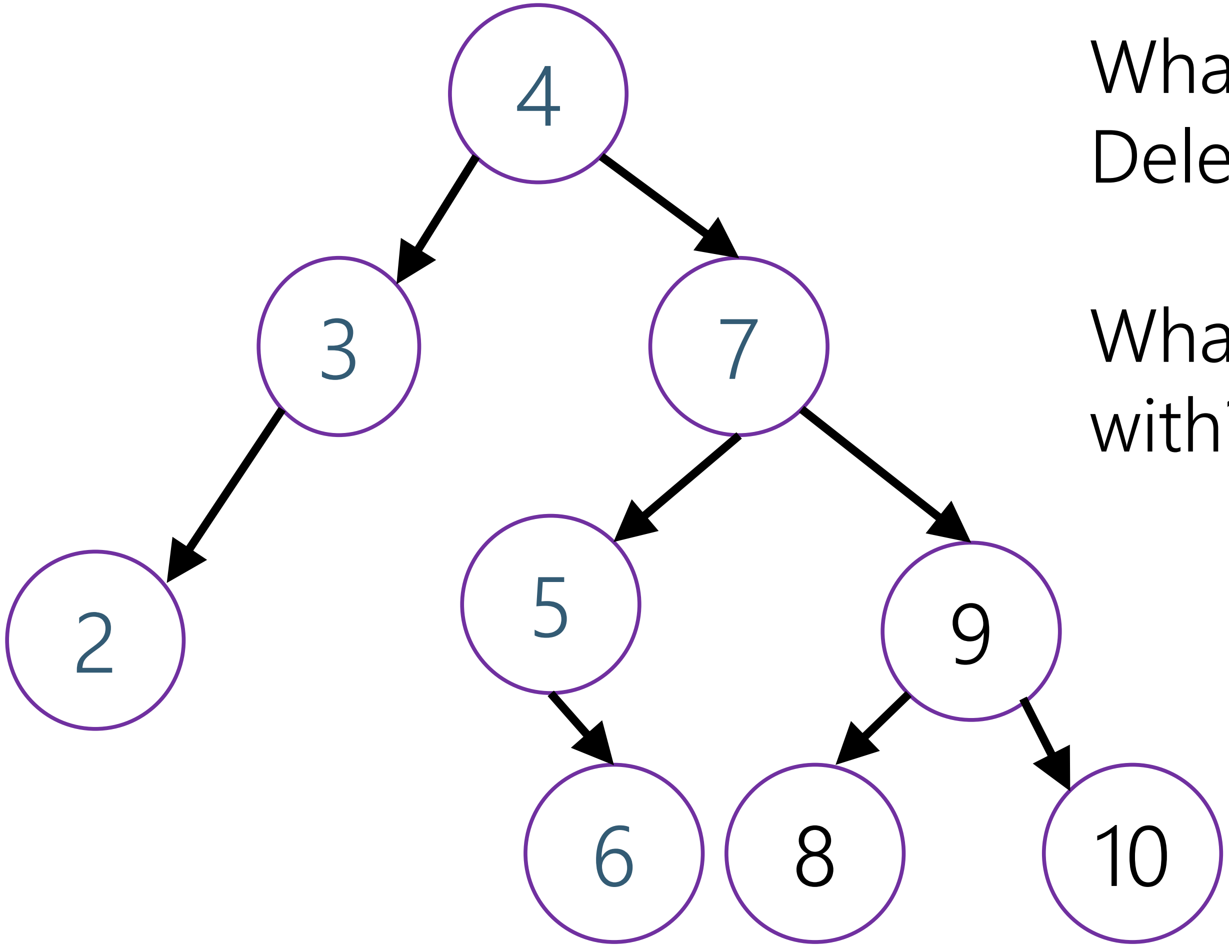


Deleting a node with one child:
Delete the node
Connect its parent and child

Delete(4)

Deletion – The Hard Case

What happens if the node to delete has two children?



What if we try Delete(7)?

What can we replace it with?

6 or 8
The biggest thing in left subtree or smallest thing in right subtree.

A Better Implementation

What about BSTs?

Keys will have to be comparable.

	Insert	Find	Delete
Average	$\Theta(\log n)$	$\Theta(\log n)$	
Worst	$\Theta(n)$	$\Theta(n)$	

A Better Implementation

What about BSTs?

Keys will have to be comparable.

	Insert	Find	Delete
Average	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

We're in the same position we were in for heaps
BSTs are great on average,
but we need to avoid the worst case.

Avoiding the Worst Case

Take I:

Let's require the tree to be complete.

It worked for heaps!

What goes wrong:

When we insert, we'll break the completeness property.

Insertions always add a new leaf, but you can't control where.

Can we fix it?

Not easily :/

Avoiding the Worst Case

Take II:

Here are some other requirements you might try. Could they work? If not what can go wrong?

Root Balanced: The root must have the same number of nodes in its left and right subtrees

Recursively Balanced: Every node must have the same number of nodes in its left and right subtrees.

Root Height Balanced: The left and right subtrees of the root must have the same height.

Avoiding the Worst Case

Take III:

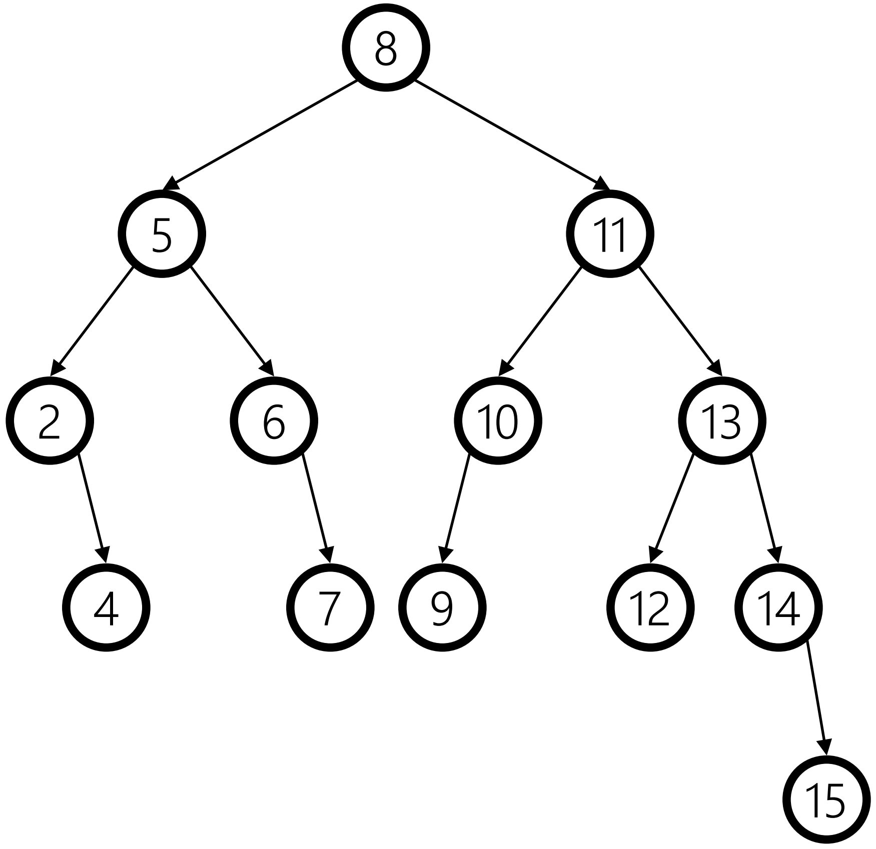
The AVL condition

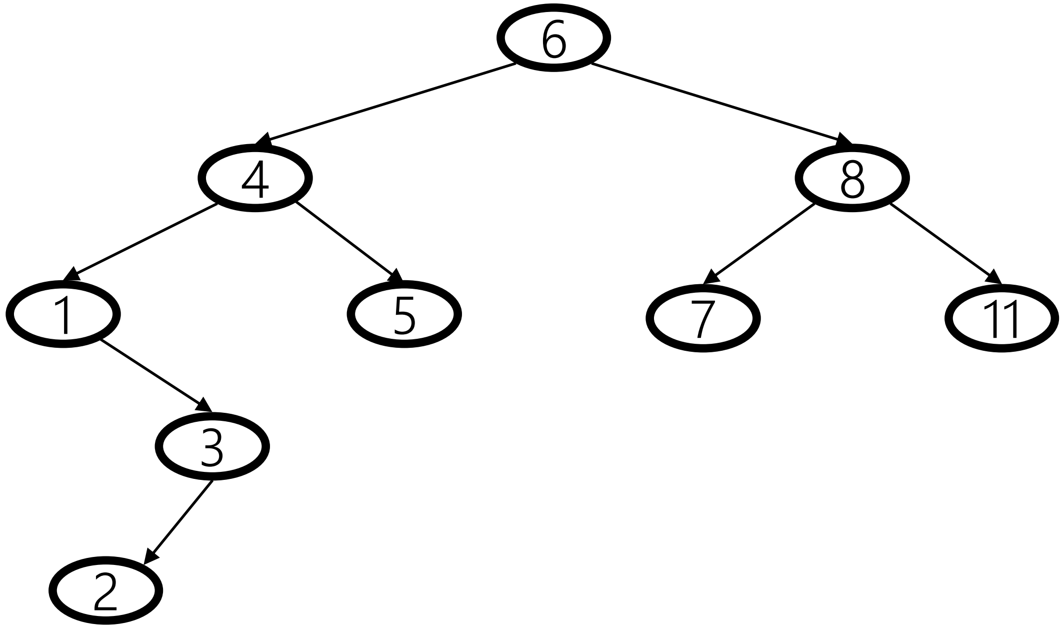
AVL condition: For every node, the height of its left subtree and right subtree differ by at most 1.

This actually works. To convince you it works, we have to check:

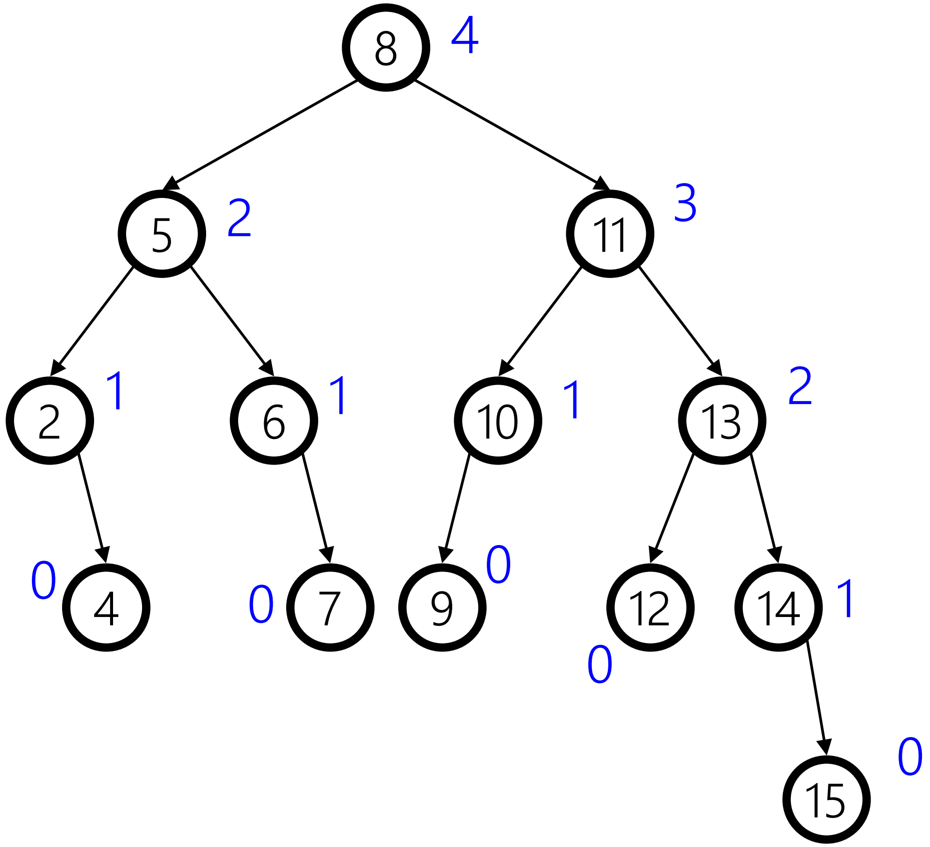
1. Such a tree must have height $O(\log n)$.
2. We must be able to maintain this property when inserting/deleting

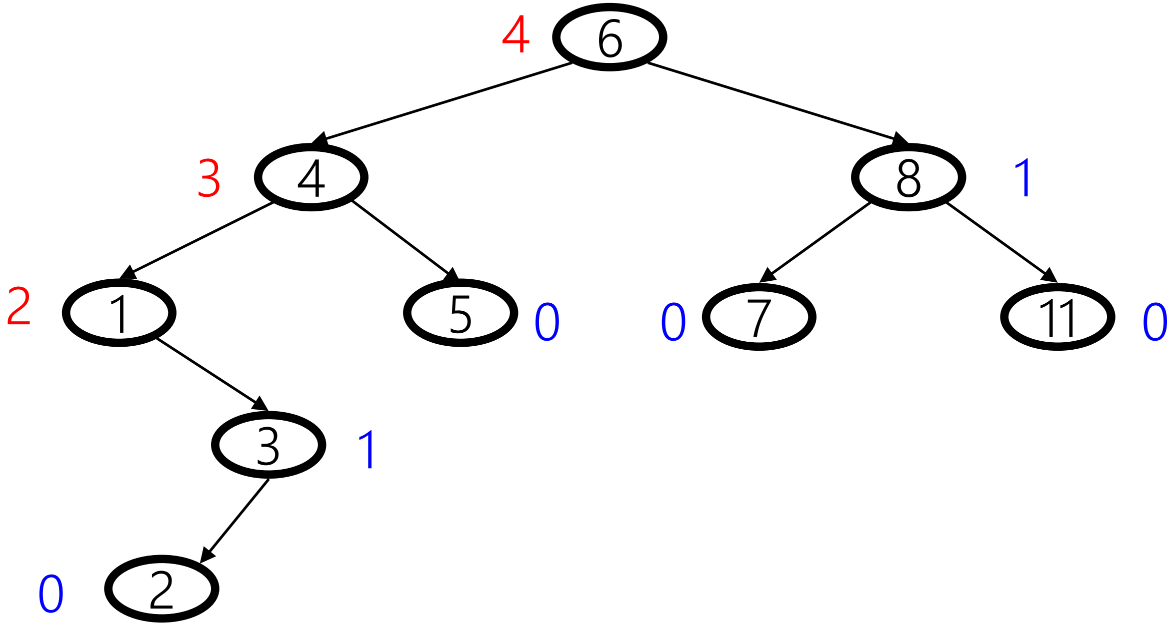
Are these valid AVL Trees?





Are these valid AVL Trees?





Bounding the Height

Suppose you have a tree of height h, meeting the AVL condition.

AVL condition: For every node, the height of its left subtree and right subtree differ by at most 1.

What is the minimum number of nodes in the tree?

If $h = 0$, then 1 node
If $h = 1$, then 2 nodes.
If $h = 2$, then 4 nodes

In general?

Bounding the Height

In general, let $N()$ be the minimum number of nodes in a tree of height h, meeting the AVL requirement.

$$N(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ N(h-1) + N(h-2) + 1 & \text{otherwise} \end{cases}$$

AVL Height Proof

AVL Height Proof

AVL Height Proof

$N(h) = N(h-1) + N(h-2) + 1$

$N(h) > N(h-1)$ thus $N(h-1) > N(h-2)$

$N(h) > 2N(h-2)$ (this means N more than doubles when h increases by 2)

$N(h) > 2^i N(h-2i)$ (for i > 0, now we find an i that satisfies a base case)

$i = \lceil h/2 \rceil - 1$ works

h is even: $h - 2i = h - (h-2) = 2$ (not a base case, but we know N(2) is 4)

h is odd: $h - 2i = h - \left(2\left\lceil \frac{h}{2} \right\rceil - 2\right) = h - \left((h+1) - 2\right) = 1$

$N(h) > 2^i N(h-2i) = 2^{\left\lceil \frac{h}{2} \right\rceil - 1} N(h-2i) \geq 2^{\left\lceil \frac{h}{2} \right\rceil - 1} N(1) = 2^{\left\lceil \frac{h}{2} \right\rceil} \geq 2^{\frac{h}{2}}$

So $N(h) > 2^{h/2}$

Now solve for h

$\log\left(N(h)\right) > \left(\frac{h}{2}\right)$ or $h < 2\log\left(N(h)\right)$

We defined N as the minimum number of nodes allowed in a tree of height h, so h is $O(\log(n))$!