Lecture 7: Analyzing Recursive Code

CSE 373: Data Structures and Algorithms

Administrivia

Project 1 Part 1 due Wednesday

Exercise 1 due Friday.

Where Are We?

Analyzing Code:

So far:

- Writing a Code Model
- Simplifying to O, Ω, Θ
- Formally proving O, Ω, Θ
- Worst case vs. Best case

This week

- Recursive Code
- Dictionaries!

Binary Search

```
int binarySearch(int[] arr, int toFind, int lo, int hi){
      if( hi < lo )
            return -1;
      if(hi == lo)
            if(arr[hi] == toFind)
                  return hi;
            return -1;
      int mid = (lo+hi) / 2;
      if(arr[mid] == toFind)
            return mid;
      else if(arr[mid] < toFind)
            return binarySearch(arr, toFind, mid+1, hi);
      else
            return binarySearch(arr, toFind, lo, mid-1);
}
```

Binary search runtime

For an array of size N, it eliminates ½ until 1 element remains.

N, N/2, N/4, N/8, ..., 4, 2, 1

- How many divisions does it take?

Think of it from the other direction:

- How many times do I have to multiply by 2 to reach N?

  1, 2, 4, 8, ..., N/4, N/2, N
- Call this number of multiplications "x".

$2^x = N$

$x = \log_2 N$

Binary search is in the **logarithmic** complexity class.

Moving Forward

The analysis is correct! But it's a little ad hoc.

It works great for binary search, but we're going to deal with more complicated recursive code in this course.

We need more powerful tools.

Model

Let's start by just getting a model. Let $T(n)$ be our model for the worst-case running time of binary search.

```
int binarySearch(int[] arr, int toFind, int lo, int hi){
      if( hi < lo )
            return -1;
      if(hi == lo)
            if(arr[hi] == toFind)
                  return hi;
            return -1;
      int mid = (lo+hi) / 2;
      if(arr[mid] == toFind)
            return mid;
      else if(arr[mid] < toFind)
            return binarySearch(arr, toFind, mid+1, hi);
      else
            return binarySearch(arr, toFind, lo, mid-1);
}
```

$$T(n) = \begin{cases} 2 & \text{if } n = 0 \\ 4 & \text{if } n = 1 \\ 5 + T\left(\frac{n}{2}\right) & \text{othwerwise} \end{cases}$$

Recurrence

Our code is recursive.

It makes sense that our model will be recursive too!

A recursive definition of a function is called a **recurrence.**

It's a lot like recursive code:

- At least one base case and at least one recursive case.
- The cases of your recurrence usually correspond exactly to the cases of the code.
- Input size should be getting smaller.

Write a recurrence

```
int recursiveFunction(int n){
      if(n < 3)
            return 3;
      for(int int i=0; i < n; i++)
            System.out.println(i);
      int val1 = recursiveFunction(n/3);
      int val2 = recursvieFunction(n/3);
      return val1 * val2;
}
```

pollEV.com/cse373su19

Write a recurrence for the running time of `recursiveFunction`

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T\left(\frac{n}{3}\right) + n & \text{otherwise} \end{cases}$$

Recurrence to Big-O

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T\left(\frac{n}{3}\right) + n & \text{otherwise} \end{cases}$$

Alright what's the big-O?

There's another similarity between recursive functions and recursive code.

It's still really hard to tell what the big-O is just by looking at it.

I can’t tell what the big-O is. What do we do?

It’s ok.

Mathematicians and computer scientists have been hard at work.

And they’ve written books.

And the answer to our problem is in one of them.

Master Theorem

Given a recurrence of the following form, where a, b, c, and d are constants:

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

Master Theorem

Given a recurrence of the following form, where a, b, c, and d are constants:

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

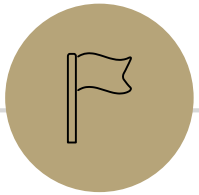
If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T\left(\frac{n}{3}\right) + n & \text{otherwise} \end{cases}$$

$\log_3 2 < 1$
We're in case 1
$T(n) \in \Theta(n)$



Binary Search Trees

Binary Search Trees

We have one more algorithm analysis topic…

But first a little bit about binary search trees.

Review: Trees!

A **tree** is a collection of nodes

- Each node has at most 1 parent and 0 or more children

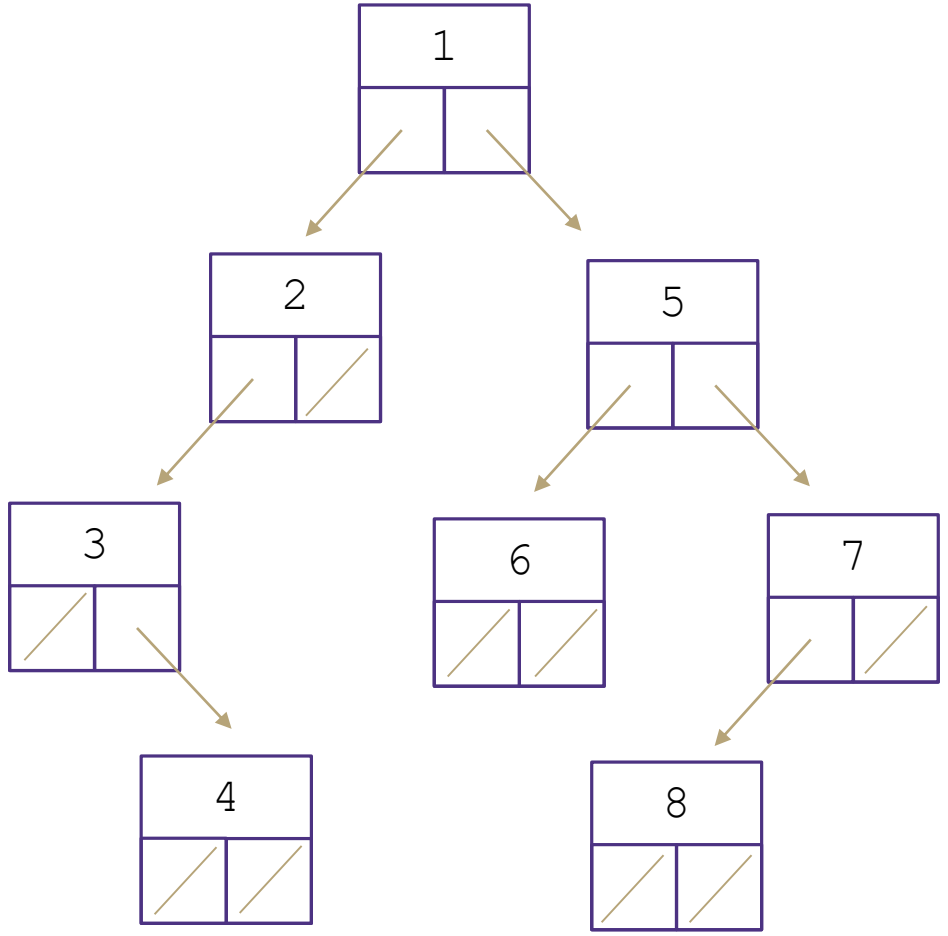
Root node: the only node with no parent, "top" of the tree

Leaf node: a node with no children

Edge: a pointer from one node to another

Subtree: a node and all it descendants

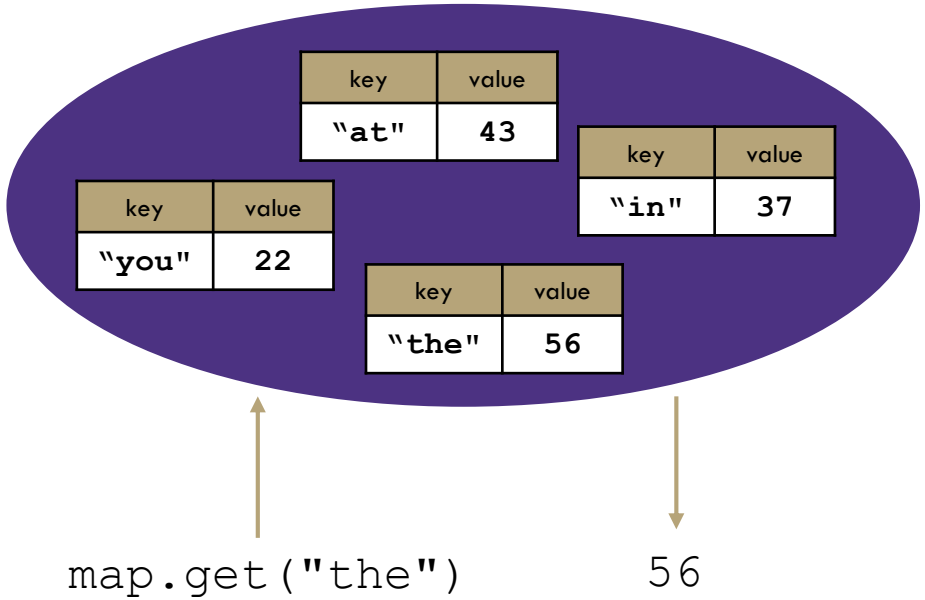
Height: the number of edges contained in the longest path from root node to some leaf node



Review: Maps

map: Holds a set of distinct *keys* and a collection of *values*, where each key is associated with one value.

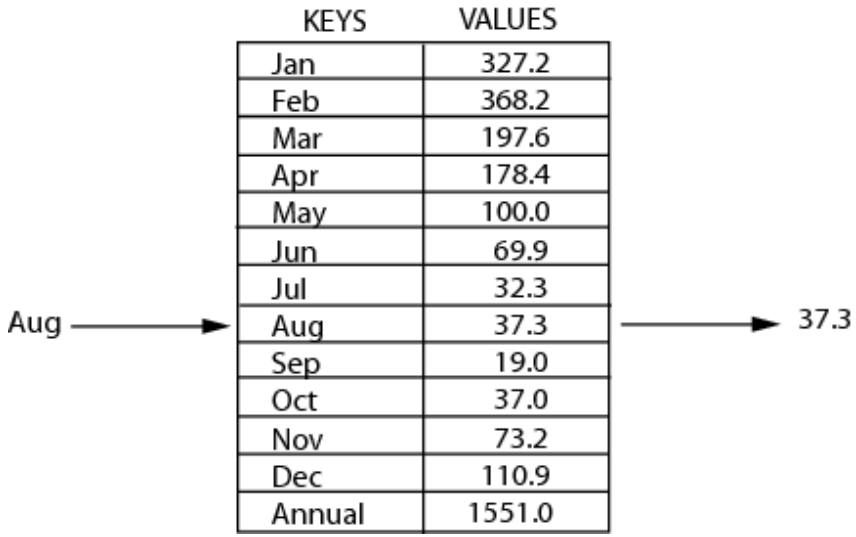
- a.k.a. "dictionary"



Dictionary ADT
state Set of items & keys Count of items
behavior put(key, item) add item to collection indexed with key get(key) return item associated with key containsKey(key) return if key already in use remove(key) remove item and associated key size() return count of items

supported operations:

- **put**(*key*, *value*): Adds a given item into collection with associated key,
  - **if the map previously had a mapping for the given key, old value is replaced**
- **get**(*key*): Retrieves the value mapped to the key
- **containsKey**(key): returns true if key is already associated with value in map, false otherwise
- **remove**(*key*): Removes the given key and its mapped value

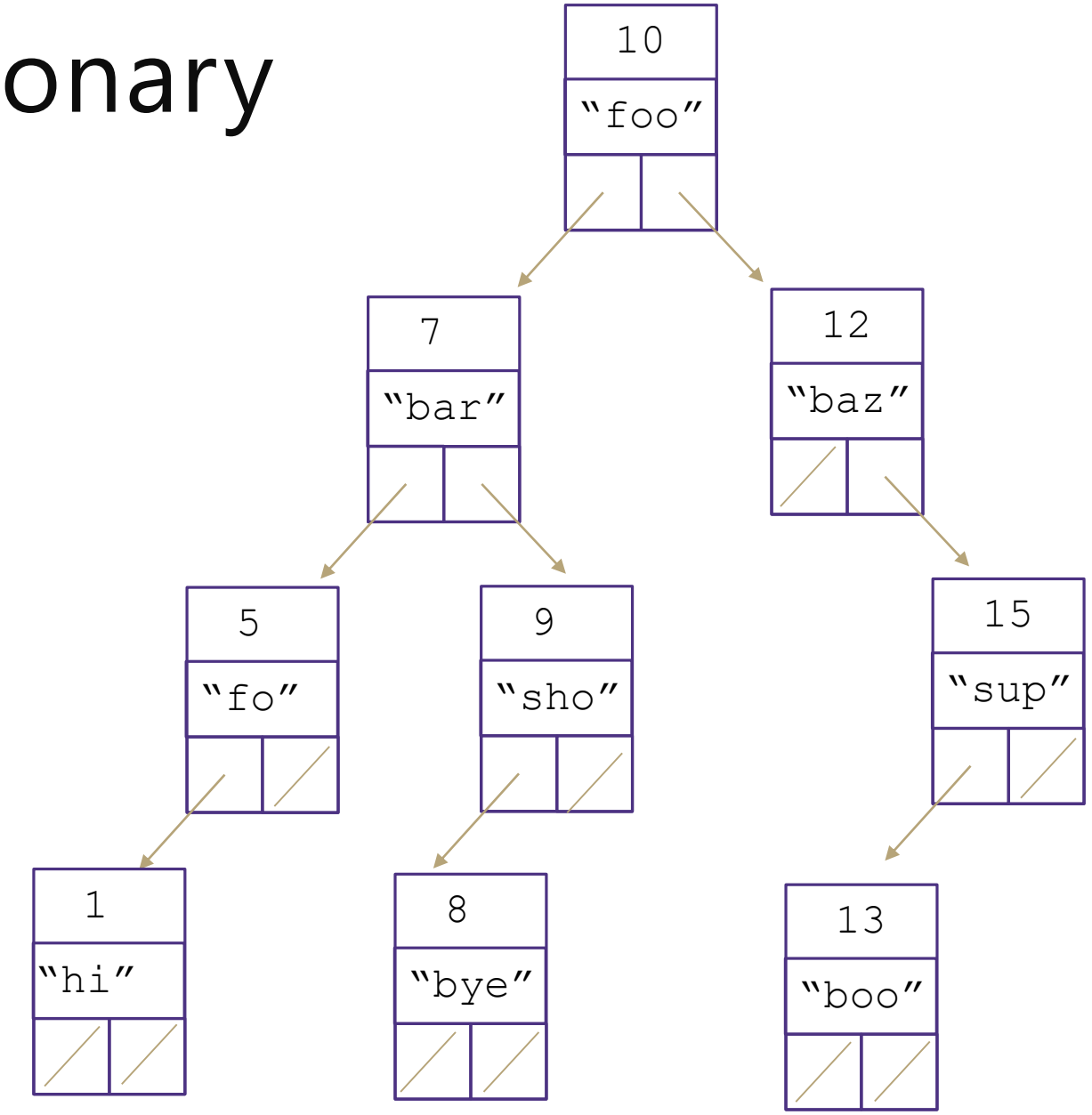


Implement a Dictionary

Binary Search Trees allow us to:

- quickly find what we're looking for
- add and remove values easily

Let's use them to implement dictionaries!



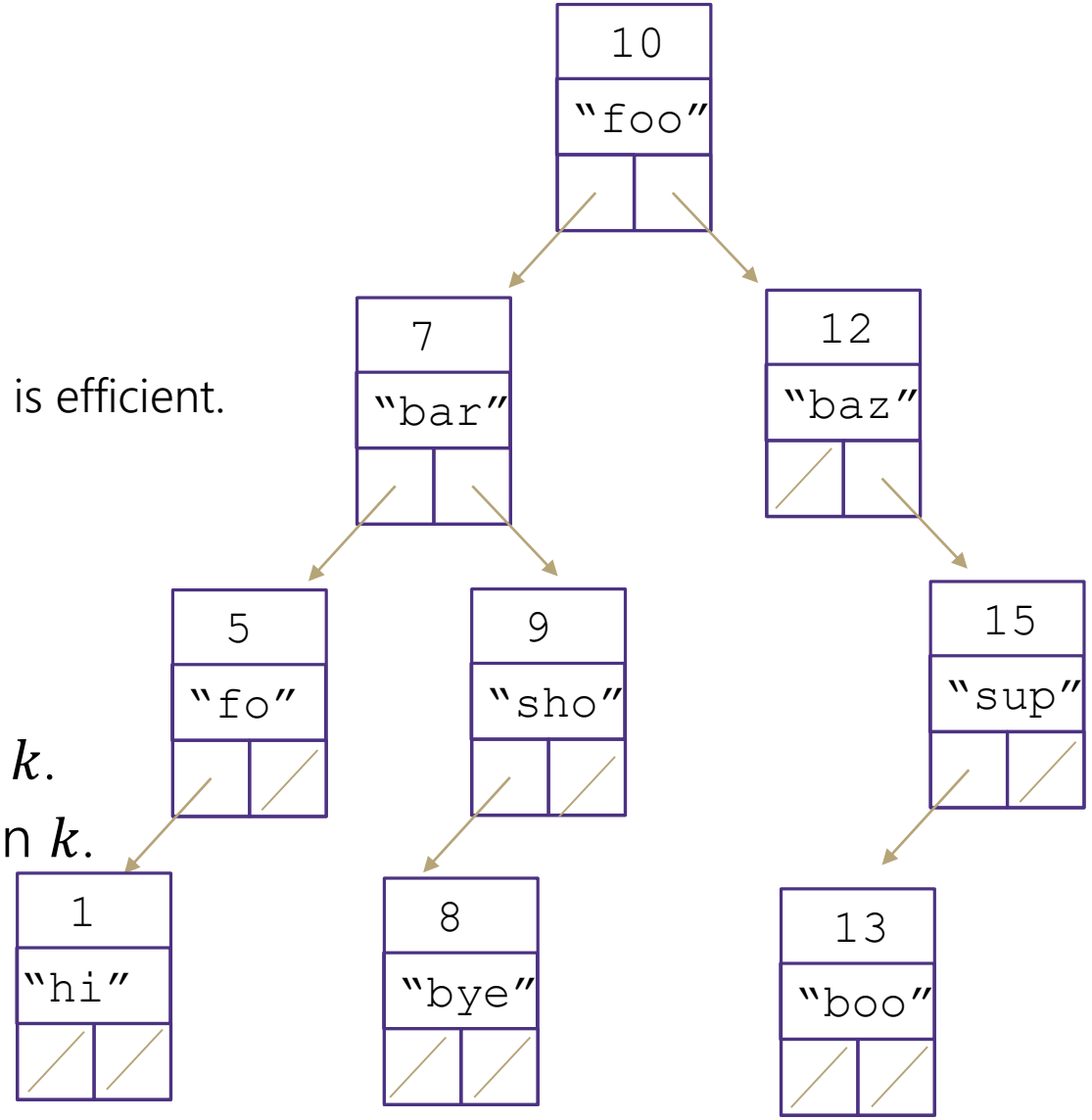
Binary Search Tree

Invariants

- Things that are always true.
- The way you make sure your data structure works and is efficient.

Binary Search Tree invariants:

- For every node with key k:
  - The left subtree has only keys smaller than k.
  - The right subtree has only keys greater than k.



BST Invariants

Why write down invariants?

They help us write methods?

Binary Search Tree invariants:

For every node with key k:

The left subtree has only keys smaller than k.

The right subtree has only keys greater than k.

How does `get(key)` work?

- Is the current node the one we're looking for?
  - Return it's value
  - Is the current node null?
    - It's not in there
- Is the current node's key too small?
  - Recurse on the right subtree
- Is the current node's key too big?
  - Recurse on the left subtree

How does `put(key, value)` work?

Let's just put it anywhere?

No! Remember the invariants!

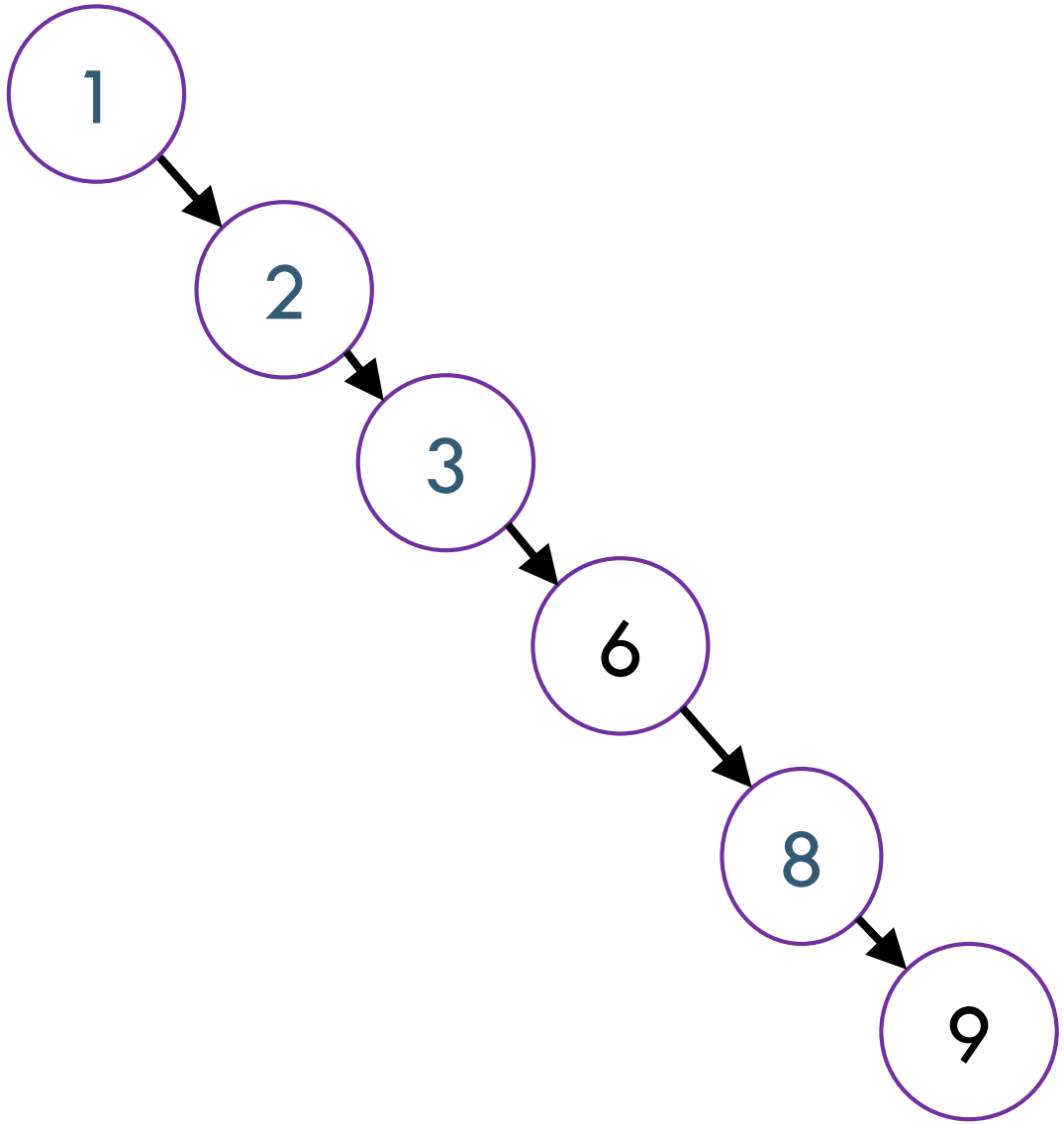
Also remember key might already be in the dictionary.

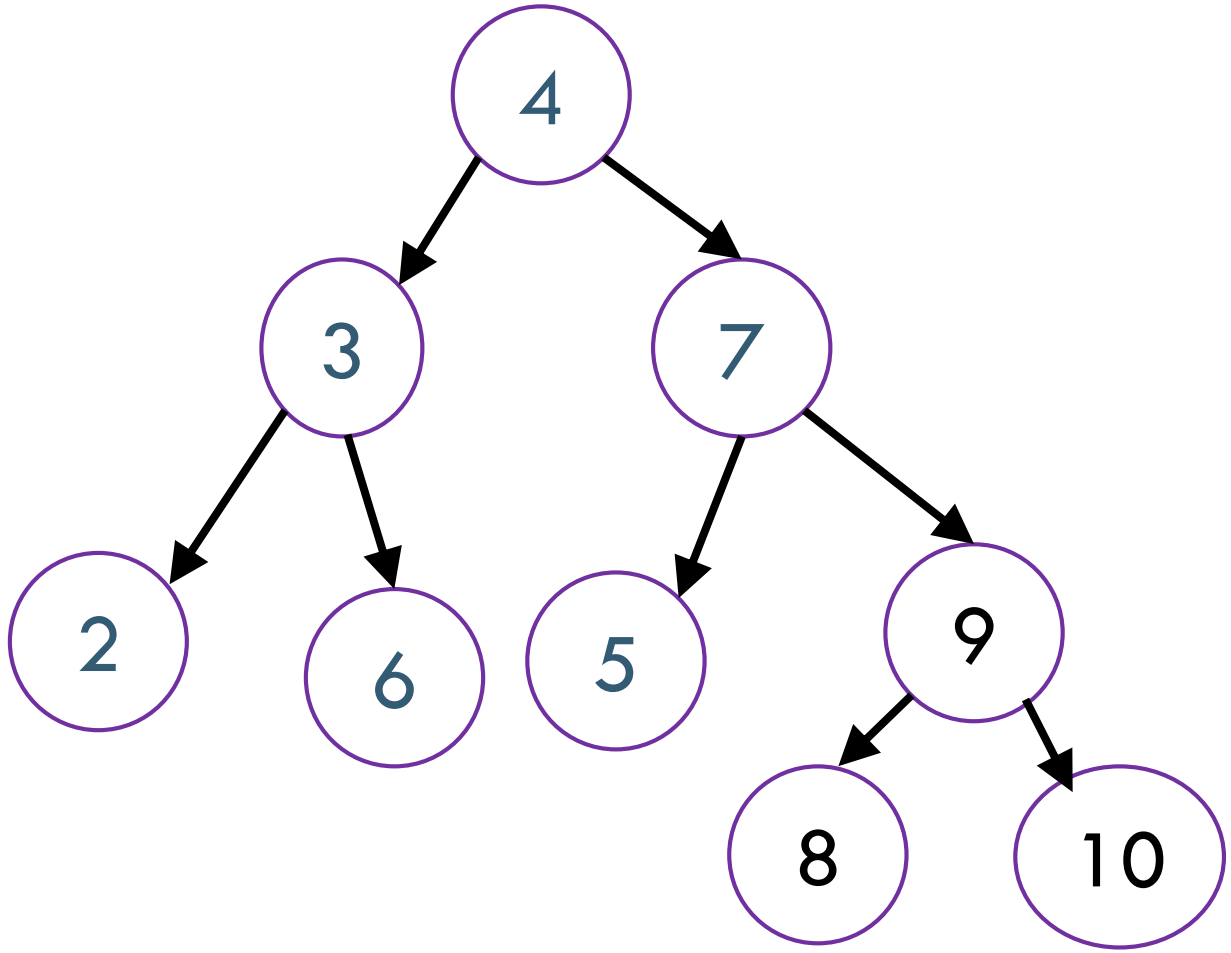
`find` first

If key is in there, overwrite the value

Otherwise, wherever we ended up is where the new node should go.

Are These Binary Search Trees?

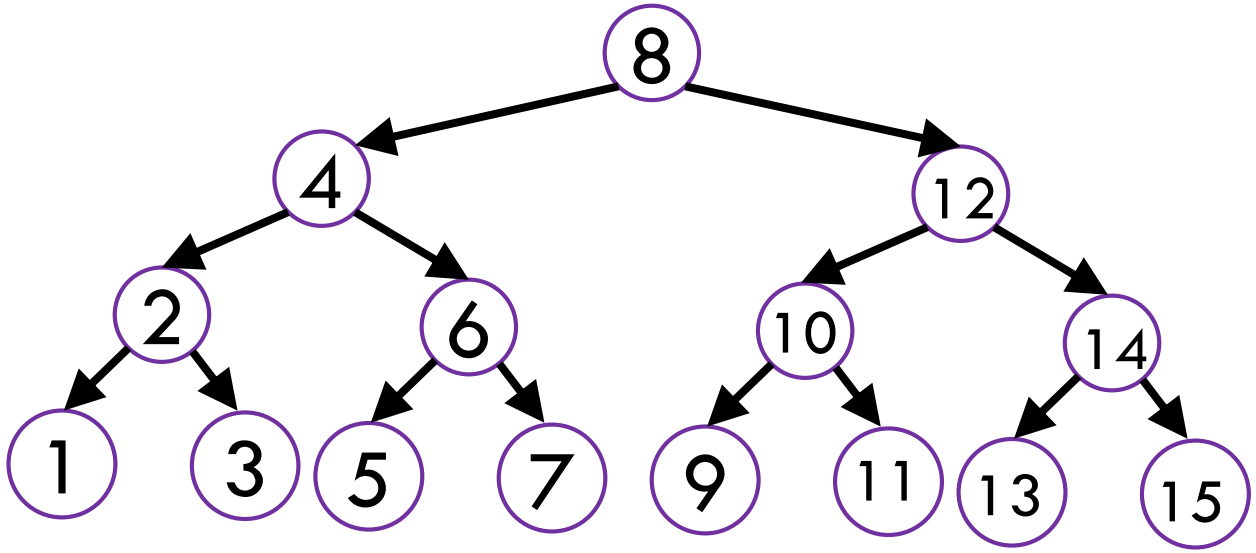




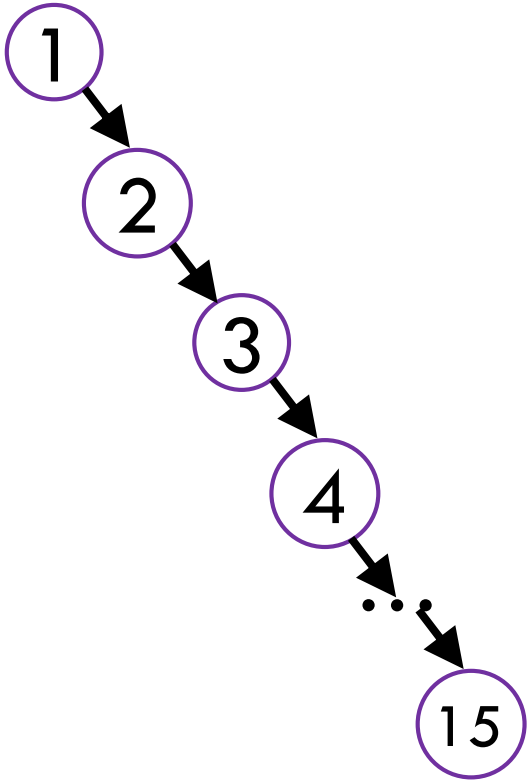
BSTs as dictionaries

Let's figure out the worst case of `get()` for two different states our BST could be in.

Perfectly balanced – for every node, its descendants are split evenly between left and right subtrees.



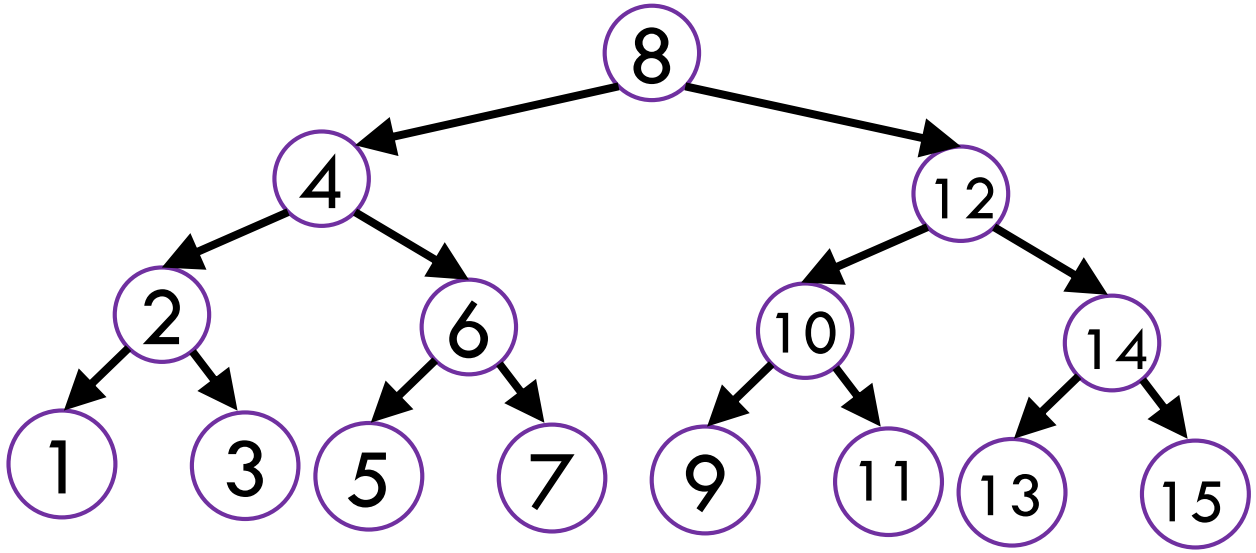
Degenerate – for every node, all of its descendants are in the right subtree.



BSTs as dictionaries

Let's figure out the worst case of `get()` for two different states our BST could be in.

Perfectly balanced – for every node, its descendants are split evenly between left and right subtrees.



`get()` is a recursive method!

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + 1 \text{ if } n > 1 \\ 3 \qquad \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

$\log_2 1 = 0 = n^0$ Case 2

$\Theta(n^0 \log n) = \Theta(\log n)$.

BSTs as dictionaries

Let's figure out the worst case of `get()` for two different states our BST could be in.

`get()` is a recursive method!

$$T(n) = \begin{cases} T(n-1) + 1 \text{ if } n > 1 \\ 3 \qquad\qquad \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

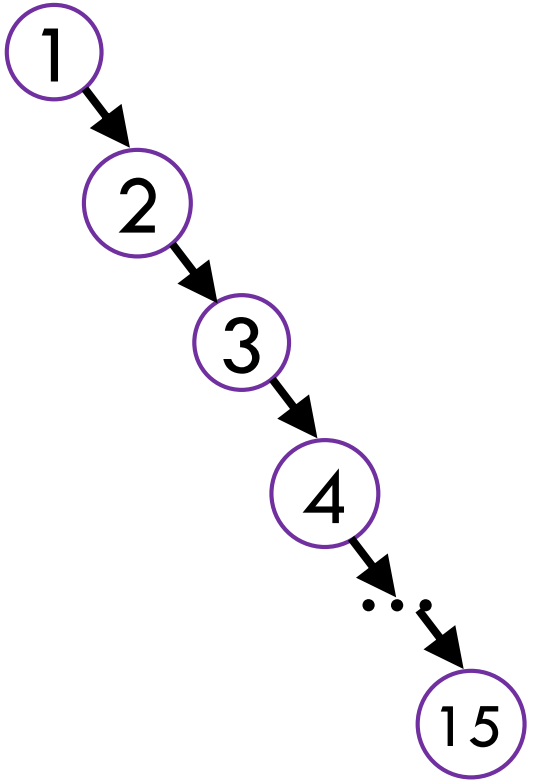
If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

Master Theorem doesn't apply!

Degenerate – for every node, all of its descendants are in the right subtree.



I can’t tell what the big-O is. What do we do?

It’s ok.

Mathematicians and computer scientists have been hard at work.

And they’ve written books.

And I’ve checked them all

This meme is outdated/unfunny.

This meme is funny.

Don’t Panic

The books don’t have a nice theorem;

They do have methods for figuring out the big-O.

Unrolling

$$T(n) = \begin{cases} T(n-1) + 1 \text{ if } n > 1 \\ 3 \qquad\qquad \text{otherwise} \end{cases}$$

Idea: keep plugging the definition of $T()$ into itself.
Until you find the pattern and can hit the base case.

Unrolling

$$T(n) = \begin{cases} T(n-1) + 1 \text{ if } n > 1 \\ 3 \qquad\qquad \text{otherwise} \end{cases}$$

$T(n) = T(n-1) + 1$

$[T(n-1-1)+1] + 1 = T(n-2) + 1 + 1$

$[T(n-2-1)+1] + 1 + 1 = T(n-3) + 1 + 1 + 1$

$[T(n-3-1)+1] + 1 + 1 + 1 = T(n-4) + 1 + 1 + 1 + 1$

$T(n-i) + i$ for any i.

The thing we don't understand is $T()$. We can get rid of it by hitting the base case.

Set i so that $n - i = 1$. → $i = n - 1$

$T\big(n-(n-1)\big) + (n-1)$

$T(1) + (n-1) = 3 + (n-1) = n + 2$

$$T(n) = n + 2$$

We did it!

For BSTs:

If we're in the case where everything is balanced, we have a much better dictionary.

But if have that degenerate BST, we're no better off than with an array or linked list.

For analyzing code:

We didn't just get the big-Θ, we actually got an exact expression too!

Let's try another one!

More Practice

Write a recurrence to describe the running time of this function, then find the big-Θ for the running time.

```
public int dumbFindMax(int[] arr, int hi){
      if(hi == 0)
            return arr[0];
      int maxInd = 0;
      for(int i=0; i<hi; i++){
            if(arr[i] > arr[maxInd])
                  maxInd=i;
      }
      return Math.max(arr[maxInd], dumbFindMax(arr, hi-1));
}
```

$$T(n) = \begin{cases} T(n-1) + n & \text{if } n \geq 2 \\ 1 & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} T(n-1) + n & \text{if } n \geq 2 \\ 1 & \text{otherwise} \end{cases}$$

$T(n-1) + n$

$T(n-1-1) + (n-1) + n = \mathrm{T(n-2)} + (\mathrm{n-1}) + \mathrm{n}$

$T(n-3) + (n-2) + (n-1) + n$

$T(n-4) + (n-3) + (n-2) + (n-1) + n$

$$T(n-i) + \sum_{j=0}^{i-1} n - j$$

Plug in i so $n - i$ is 1

$$T(n-(n-1)) + \sum_{j=0}^{n-1-1} n - j =$$

$$T(n) = \begin{cases} T(n-1) + n & \text{if } n \geq 2 \\ 1 & \text{otherwise} \end{cases}$$

$$1 + \sum_{j=0}^{n-2} n - j = 1 + \sum_{j=0}^{n-2} n - \sum_{j=0}^{n-2} j$$

$$= 1 + n(n-1) - \sum_{j=0}^{n-2} j$$

$$= 1 + n(n-1) - \frac{(n-1)(n-2)}{2}$$

$$= n^2 - n - \frac{n^2}{2} + \frac{3n}{2} - 1$$

$$\in \Theta(n^2)$$

$$T(n) = \begin{cases} 3T\left(\frac{n}{4}\right) + n^2 \text{ if } n > 1 \\ 4 \qquad\qquad \text{otherwise} \end{cases}$$

We can unroll to get the answer here, but it's really easy to make a small algebra mistake.

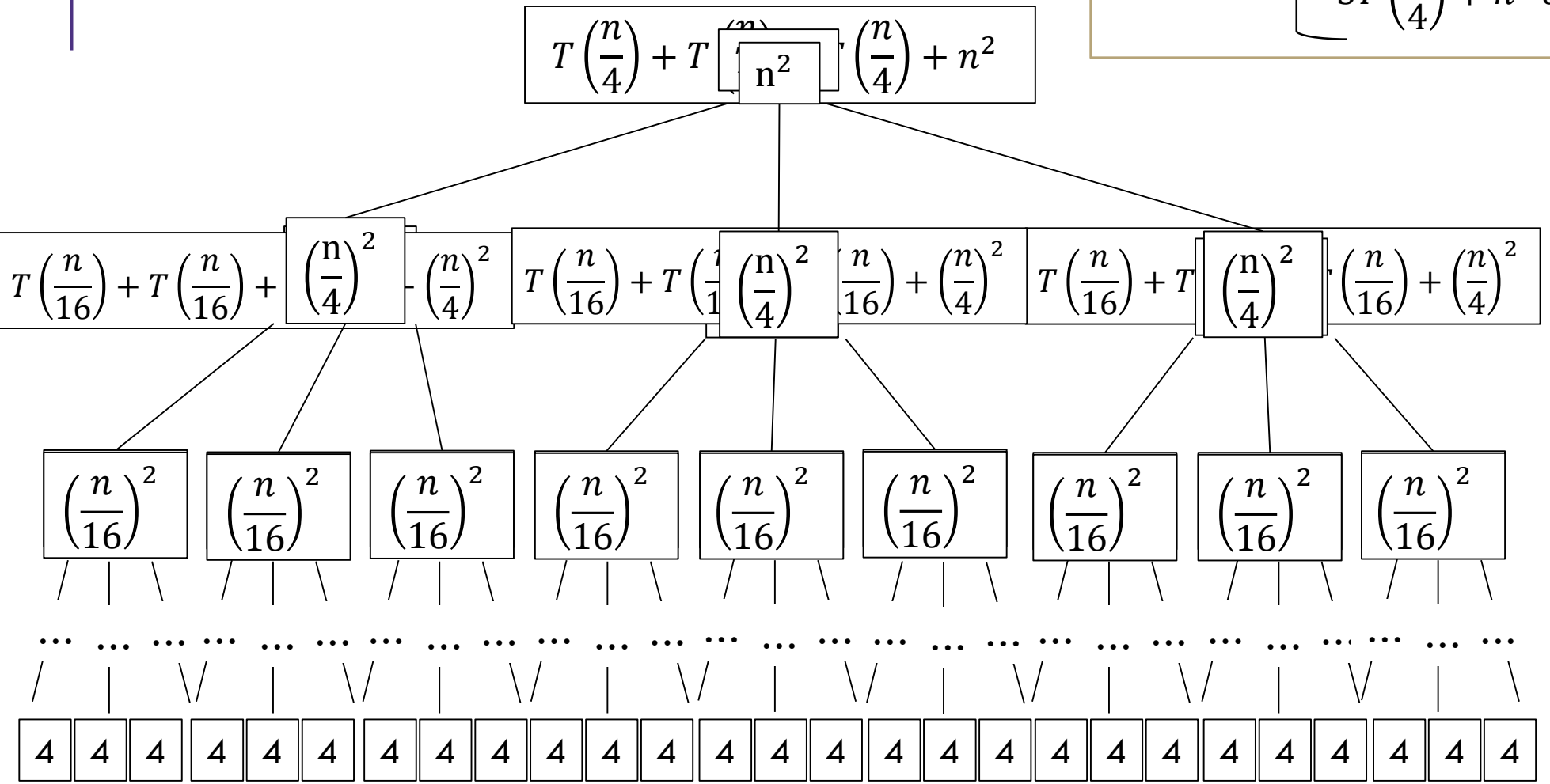
If that happens we might not be able to find the pattern

- Or worse find the wrong pattern.

There's a way to organize our algebra so it's easier to find the pattern.

Tree Method Practice

$$T(n) = \begin{cases} 4 \text{ when } n \leq 1 \\ 3T\left(\frac{n}{4}\right) + n^2 \text{ otherwise} \end{cases}$$



Answer the following questions:

1. What is the size of the input on level i?
2. What is the work done by each node on the i^{th} recursive level
3. What is the number of nodes at level i?
4. What is the total work done at the i^th recursive level?
5. What value of i does the last level occur?
6. What is the total work across the base case level?

EXAMPLE PROVIDED BY CS 161 – JESSICA SU
HTTPS://WEB.STANFORD.EDU/CLASS/ARCHIVE/CS/CS161/CS161.1168/LECTURE3.PDF

Tree Method Practice

$$T(n) = \begin{cases} 4 \text{ when } n \leq 1 \\ 3T\left(\frac{n}{4}\right) + cn^2 \text{ otherwise} \end{cases}$$

5 Minutes

1. What is the size of the input on level i? $\left(\frac{n}{4}\right)^i$
2. What is the work done by each node on the i^{th} recursive level? $\left(\frac{n}{4^i}\right)^2$
3. What is the number of nodes at level i? 3^i
4. What is the total work done at the i^th recursive level? $3^i\left[\left(\frac{n}{4}\right)^i\right]^2 = \left(\frac{3}{16}\right)^i n^2$
5. What value of i does the last level occur?

   $\frac{n}{4^i} = 1 \rightarrow n = 4^i \rightarrow i = \log_4 n$
6. What is the total work across the base case level? $3^{\log_4 n} \cdot 4$

power of a log

$x^{\log_b y} = y^{\log_b x}$

$4 \cdot n^{\log_4 3}$

Level (i)	Number of Nodes	Work per Node	Work per Level
0	1	n^2	n^2
1	3	$\left(\frac{n}{4}\right)^2$	$\frac{3}{4^2}n^2$
2	9	$\left(\frac{n}{4^2}\right)^2$	$\frac{3^2}{4^4}n^2$
base	$3^{\log_4 n}$	4	$12^{\log_4 n}$

Combining it all together…

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i n^2 + 4n^{\log_4 3}$$

Tree Method Practice

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i n^2 + 4n^{\log_4 3}$$

factoring out a constant

$$\sum_{i=a}^{b} cf(i) = c\sum_{i=a}^{b} f(i)$$

$$T(n) = n^2 \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i + 4n^{\log_4 3}$$

finite geometric series

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

Closed form:

$$T(n) = n^2 \left(\frac{\left(\frac{3}{16}\right)^{\log_4 n} - 1}{\frac{3}{16} - 1} \right) + 4n^{\log_4 3}$$

So what's the big-Θ…

$$T(n) = n^2 \left(-\frac{16}{13}\right)\left(\frac{3}{16}\right)^{\log_4 n} + \left(\frac{16}{13}\right)n^2 + 4n^{\log_4 3}$$

$$T(n) = n^2 \left(-\frac{16}{13}\right)(n)^{\log_4 \frac{3}{16}} + \left(\frac{16}{13}\right)n^2 + 4n^{\log_4 3}$$

$$T(n) \in O(n^2)$$