Set and Map ADTs: B-Trees

CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Aaron Johnston	Ethan Knutson	Nathan Lipiarski
Amanda Park	Farrell Fileas	Sam Long
Anish Velagapudi	Howard Xiao	Yifan Bai
Brian Chan	Jade Watkins	Yuma Tou
Elena Spasova	Lea Quan	

Announcements

- Asymptotic Analysis: handout coming soon
- Workshops
  - Student-centered study groups; bring your questions!
  - Friday 11:30-12:30 @ CSE 203
- Extra Drop-in Time
  - Saturday morning: 10:30am-12pm @ Odegaard 117E
- “tl;dr” slides are your per-topic learning objectives

Questions from Reading Quiz

- Why is the reading quiz borkened 💔? We will regrade!
- Why is BST height in $O(N^2)$? Best structured tree: $h \in \Theta(\log N)$
  Worst structured tree: $h \in \Theta(N)$
- Why is BST height NOT in $\Theta(N)$? $\therefore$ No Θ-bound for the overall case!
- How do you calculate average depth? Why do we care about depth? How does it relate to height?

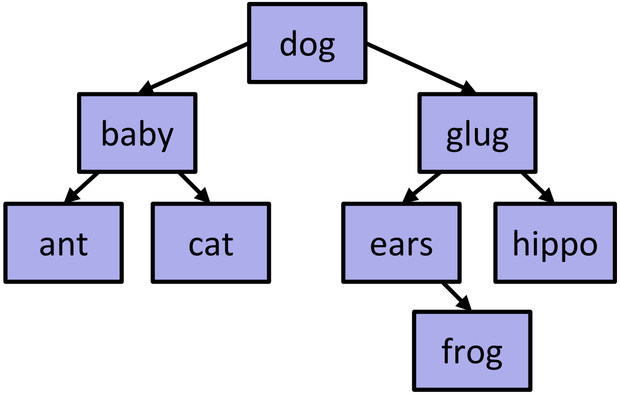
Node attributes	Tree attributes
depth	avg depth
	height

Lecture Outline

- **BST Remove (cont.)**
- BST Tree Height
- 2-3 Trees
- B-Trees

Binary Search Trees: Remove

- 3 cases based on the number of children
  1. Key has no children
  2. Key has one child
  3. Key has two children

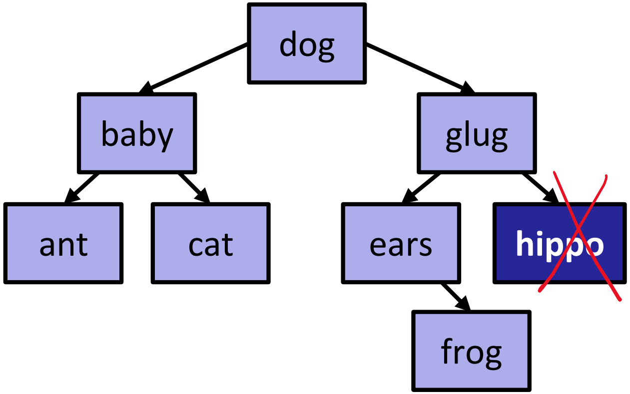


- In each case, we must maintain the **Binary Search Tree Invariant**!

BST Remove: Case #1: Leaf

- Remove the node with the value **hippo**

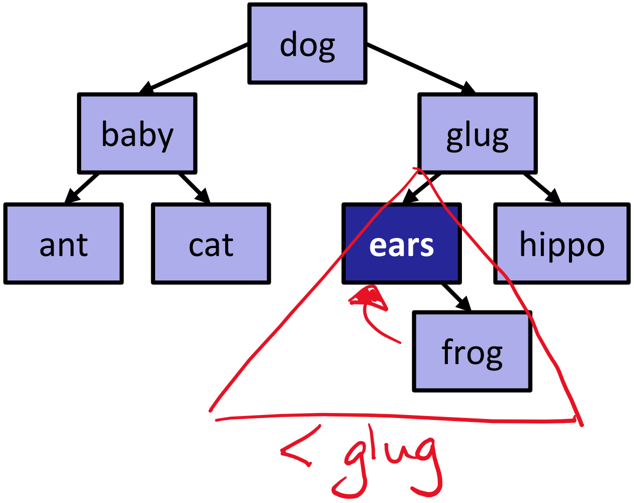
```
BSTNode remove(BSTNode n) {








}
```



BST Remove: Case #2: One Child

- Remove the node with the value **ears**
  - What does the BST invariant say about the descendant's values?

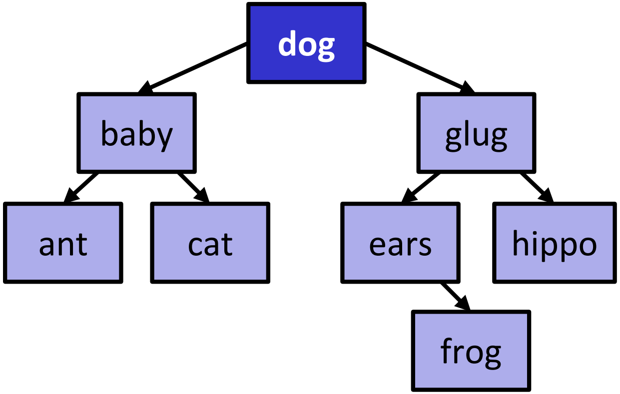
```
BSTNode remove(BSTNode n) {







}
```



BST Remove: Case #3: Two Children

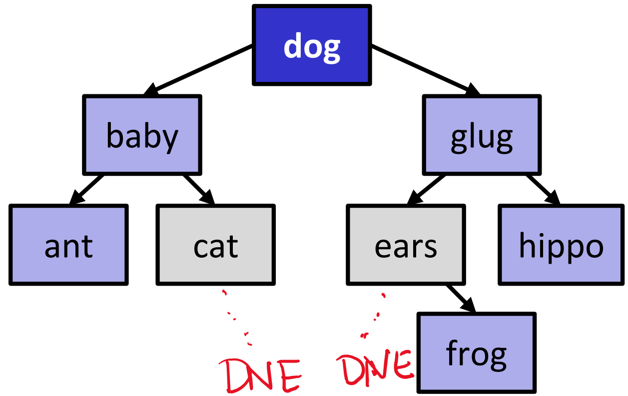
- Remove the node with the value **dog**

- The replacement node:
  - Must be ≻ than all keys in left subtree
  - Must be ≺ than all keys in right subtree



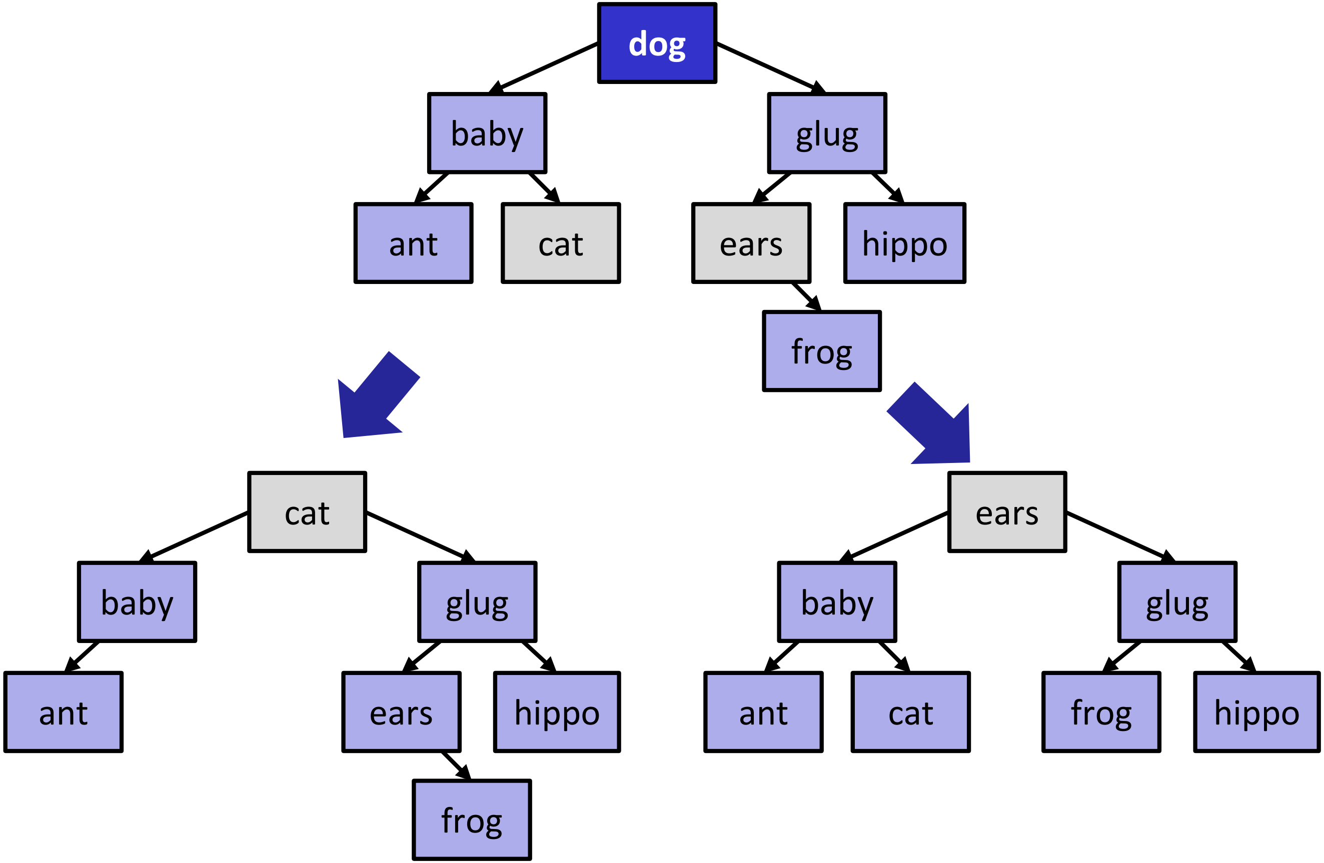
BST Remove: Case #3: Two Children

- Remove the node with the value **dog**
- The replacement node:
  - Must be ≻ than all keys in left subtree: **predecessor** (**cat**)
  - Must be ≺ than all keys in right subtree: **successor** (**ears**)
- The predecessor or successor have either 0 or 1 children



a, b, c, d, e, f, g, h

BST Remove: Case #3: Two Children

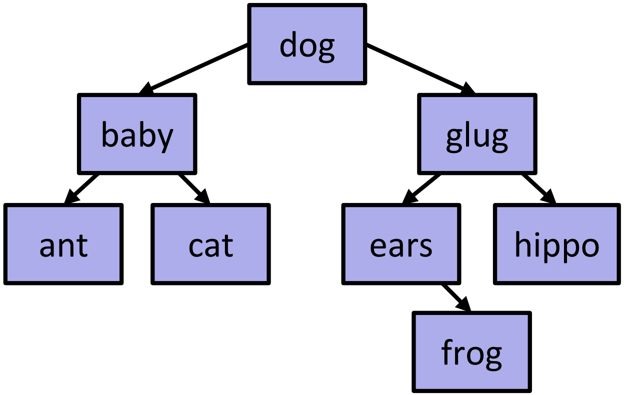


Aside: Finding the largest (or smallest) node

- The predecessor is the largest node in the left subtree
- The successor is the smallest node in the right subtree

- How do you find the largest (and smallest) node in a tree?
  - Remember that subtrees are trees too

```
BSTNode largest(BSTNode n) {
  while (n.right != null) {
    n = n.right;
  }
  return n;
}
```



tl;dr

- Binary Search Trees implement both the Set and Map ADTs
- Binary Search Trees are *recursively defined*
- Binary Search Trees can be an efficient Map/Set ADT

	LinkedList Map, Worst Case	BST Map, Worst Case
Find	Θ(N)	Θ(h)
Add	Θ(N)	Θ(h)
Remove	Θ(N)	Θ(h)

What is the relationship between N & h? 🤔

Lecture Outline

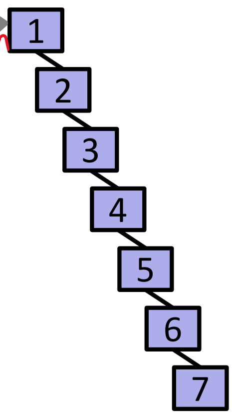
- BST Remove (cont.)
- **BST Tree Height**
- 2-3 Trees
- B-Trees

Binary Search Tree: Height

- Suppose we want to build a BST out of {1, 2, 3, 4, 5, 6, 7}

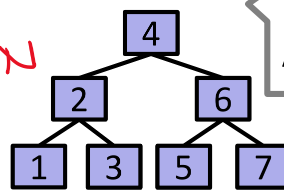
- Give a sequence of add operations that result in:
  - a **spindly** tree ("worst case") : sorted order
  - a **bushy** tree ("best case") : ? any non-sorted order?

Height: 6
Average Depth: 3



N

log N



Height: 2
Average Depth: 1.43

Randomization: Mathematical Analysis

- Binary search tree height is in O(N)
  - Worst case height: Θ(N)
  - Best case height: Θ(log N)
  - Θ(log N) via randomized insertion
    - Randomized insertion with randomized deletion is still Θ(log N) height
- BSTs are frequently concerned with best- and worst-case *tree structure*

Average Depth of a Randomized BST

If N distinct keys are inserted in random order, the expected average depth is

~ 2 ln N = Θ(log N).

Total Height of a Randomized BST

If N distinct keys are inserted in random order, the expected height is

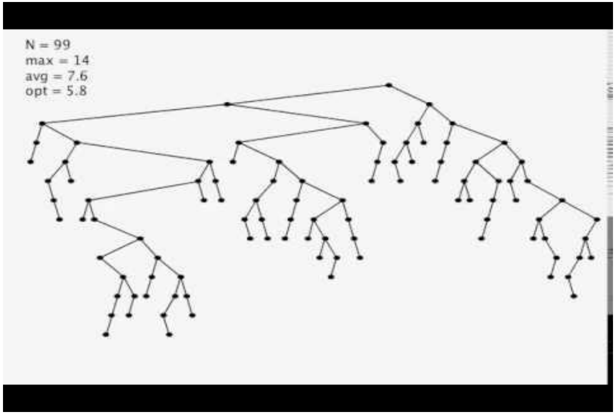
~ 4.311 ln N = Θ(log N).

The Height of a Randomized Binary Search Tree (Reed/STOC 2000)

What About "Real World" BSTs?

- These examples are contrived! What about real-world workloads?

- An approximation of the real-world: inserting random numbers



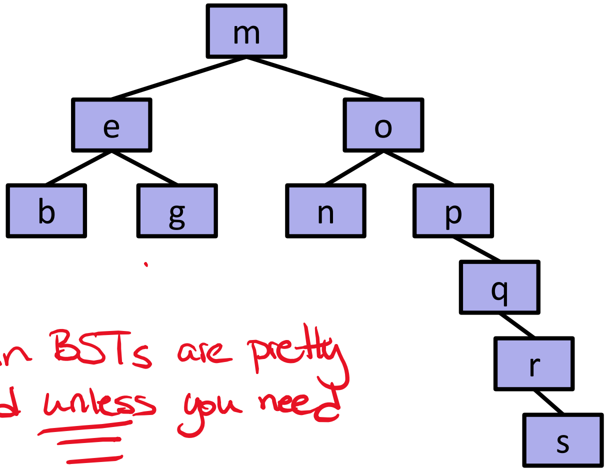
Random Insertion into a BST (Kevin Wayne/Princeton)
https://www.youtube.com/watch?v=5dGkblzqdmc

Random trees have Θ(log N) average depth and height

Random trees are bushy, not spindly

Randomization is Pretty Good!

- BSTs have great runtime if we insert keys randomly
  - Θ(log N) per insertion



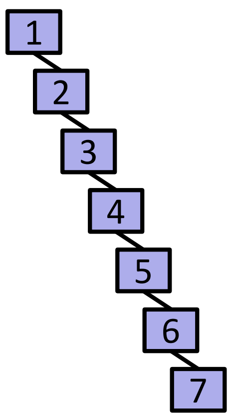
- But:
  - We can't always insert our keys in a random order. Why?
  - What if we need *guaranteed* Θ(log N) runtime?

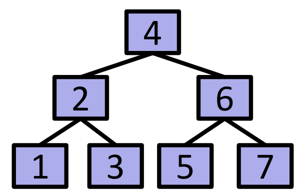
plain BSTs are pretty good unless you need a guarantee against sorted input

Bounding the Height (ie, protecting against sorted input)

- Recall that a Binary Search Tree's invariant is:
  - The left subtree only contains values <k
  - The right subtree only contains values >k

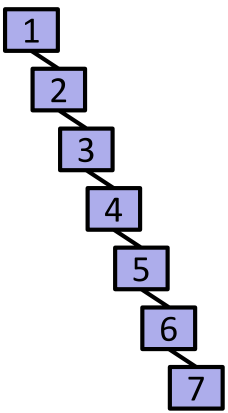
- What invariants could we add, to bound the height to log N?

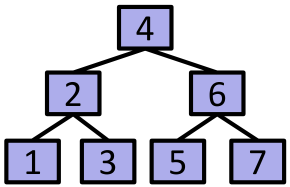




Bounding the Height: Example Invariant

- **Hypothesis:** Every node has either 0 or 2 children
- **Analysis**: What is the worst-case height for this tree?



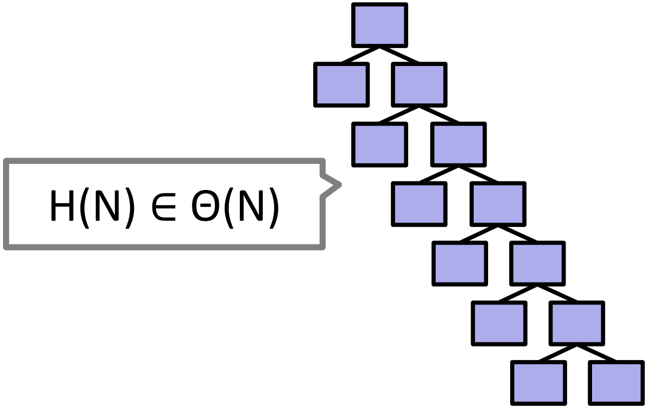


Poll Everywhere

pollev.com/uwcse373

What is the worst-case height of a BST where every node must have either 0 or 2 children?

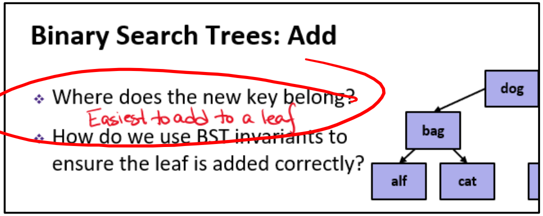
A. $\Theta(1)$
B. $\Theta(\log_2 N)$
C. $\Theta(N)$
D. $\Theta(N \log_2 N)$
E. $\Theta(N^2)$



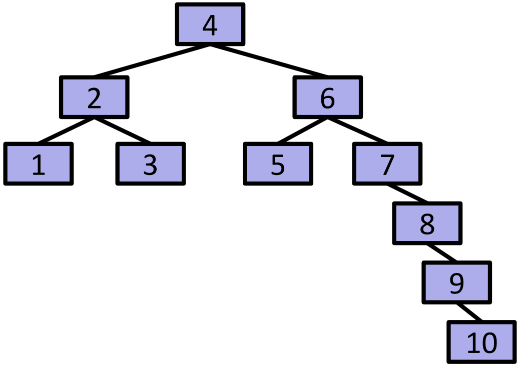
How do you add a node to this tree?

Adding Nodes Creates Worst-case Height Trees

- Unbalanced growth leads to worst-case height trees



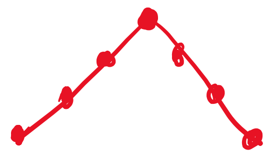
- When does adding a new node affect the height of a tree?
  - Can you explain in terms of the subtrees (ie, recursively)?



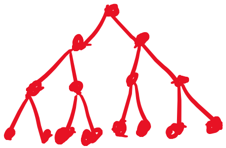
Your Turn: Generate Some Invariants

- Generate an invariant that might balance your tree
  - Is it strong enough to roughly-balance the tree?
  - Is it flexible enough to be maintainable?

· Root balanced: not strong enough



· Recursively balanced: strong, but "complete" trees are unmaintainable



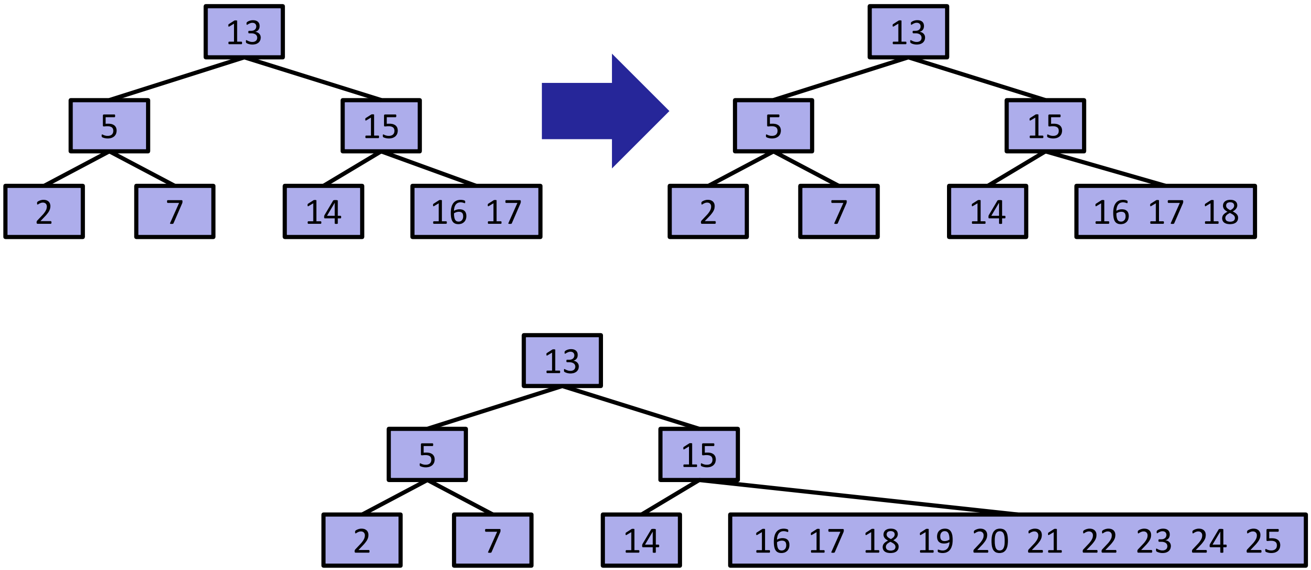
Lecture Outline

- BST Remove (cont.)
- BST Tree Height
- **2-3 Trees**
- B-Trees

Bounding the Height: Overstuff the leaves

Results in a non-binary search tree!

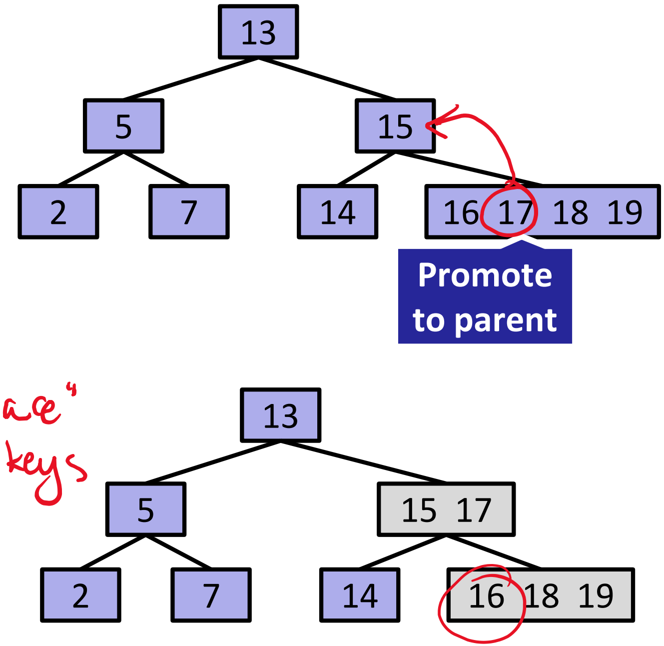
- If we never add new leaves, the tree can never get unbalanced
  - **Instead:** Overstuff existing leaves to avoid adding new leaves



Overstuffed Leaves: Promote the Keys

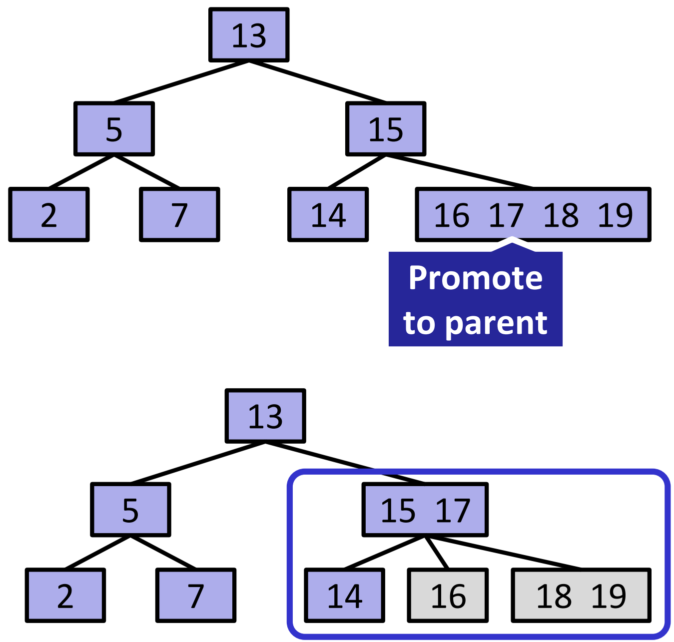
- Set a limit L on number of keys
  - e.g. L=3
- If any node has more than L keys, give ("promote") a key to the parent
  - e.g. the left-middle key
  - Why not the leftmost or rightmost?

want "space" between keys



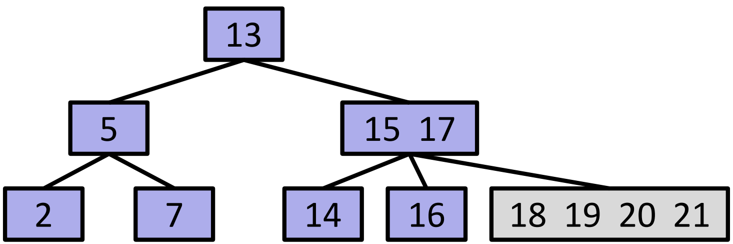
Promoting Keys Splits the Leaf Node

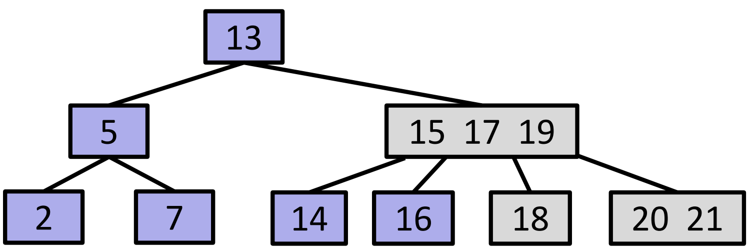
- Set a limit L on number of keys
  - e.g. L=3
- If any node has more than L keys, give (“promote”) a key to the parent
  - e.g. the left-middle key
  - Why not the leftmost or rightmost?
  - **Promoting a key splits the old overstuffed node into two new parts: left and right.**



Practice: Adding More Keys

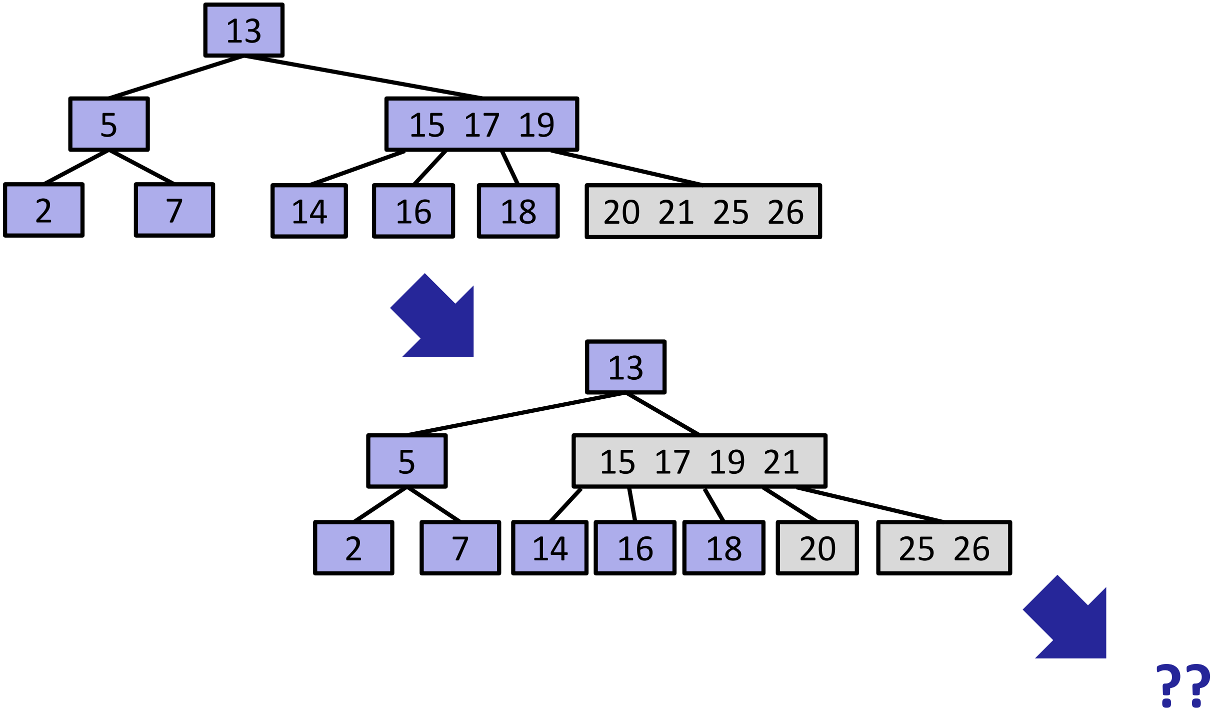
- Suppose we add the keys 20 and 21.
- If our cap is at most L=3 keys per node, draw the post-split tree.





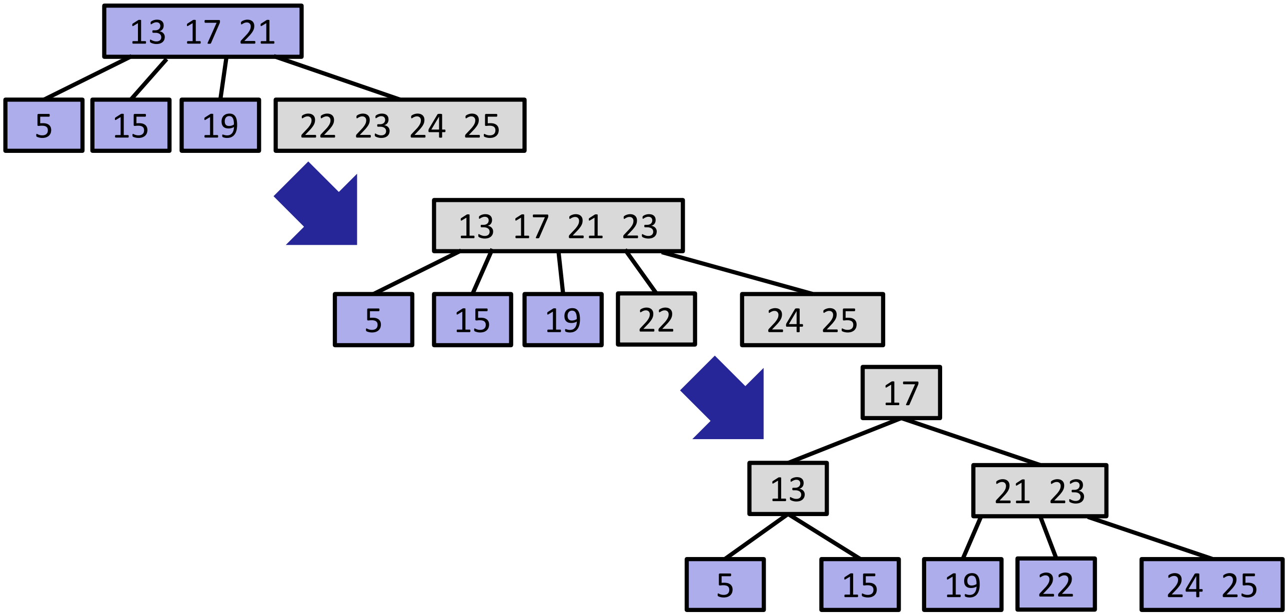
Promoting Keys Can Cascade Into Ancestors

- Add 25 and 26



Overstuffing the Root Node

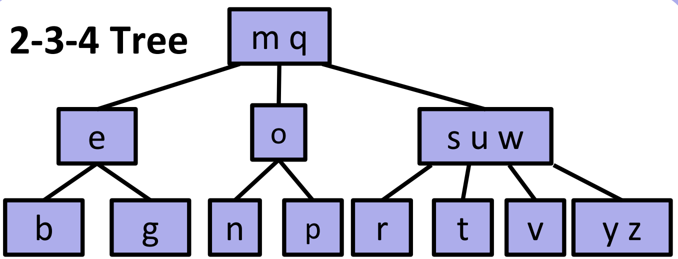
- If promotions can cascade up the tree, we may eventually need to split the root.
- Splitting the root is the only time a tree grows in height!



2-3, 2-3-4, and B-Trees

- We chose limit L=3 keys in each node. Formally, this is called a **2-3-4 Tree**: each non-leaf node can have 2, 3, or 4 children

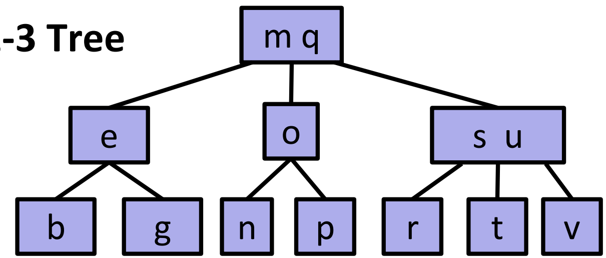
2-3-4 Tree



Max 3 keys and 4 non-null children per node.

- **2-3 Tree**. Choose L=2 keys. Each non-leaf node can have 2 or 3 children

2-3 Tree

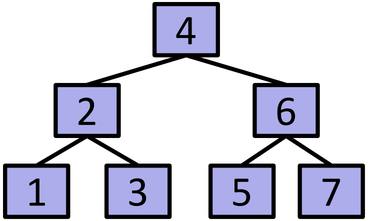


Max 2 keys and 3 non-null children per node.

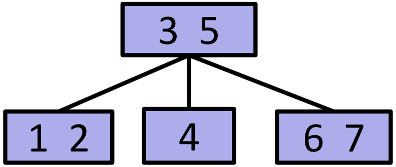
- **B-Trees** are the generalization of this idea for any choice of L

2-3 Tree Practice

- Give an insertion order for the keys {1, 2, 3, 4, 5, 6, 7} that results in:
  - a **max-height** 2-3 Tree
  - a **min-height** 2-3 Tree



1, 2, 3, 4, 5, 6, 7



2, 3, 4, 5, 6, 1, 7

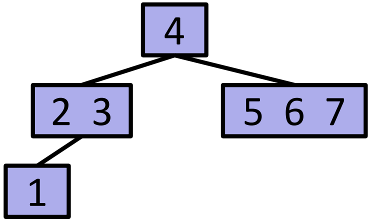
Demo: https://www.cs.usfca.edu/~galles/visualization/BTree.html

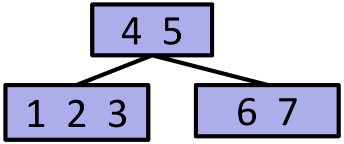
Lecture Outline

- BST Remove (cont.)
- BST Tree Height
- 2-3 Trees
- **B-Trees**

B-Tree Invariants

- B-Tree's invariants guarantee "bushy" trees (ie, $H(N) \in \Theta(\log_2 N)$)
  1. All leaves must be the same depth from the root
     - Achieved because the tree's height only grows from the root
  2. A non-leaf node with k keys must have exactly k + 1 non-null children
     - Achieved because we *remove two keys* from an overstuffed child: one is promoted to the parent and the other becomes the new child of the newly-promoted parent key
  3. A non-leaf non-root node must have at least ceil(L/2) children
     - (A non-leaf root node must have >=2 children)
- Why are these invalid B-Trees?

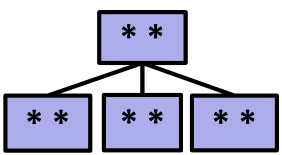


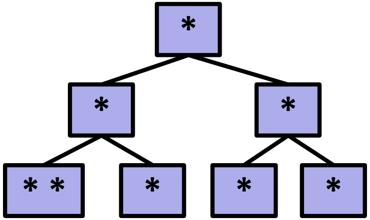


B-Tree Invariants Bound Its Height

- Smallest possible height ("shortest tree") is when all nodes have L keys
  - $H(N) \sim \log_{L+1} N \in \Theta(\log N)$
- Largest possible height ("tallest tree") is when all non-leaf nodes have just 1 key
  - $H(N) \sim \log_2 N \in \Theta(\log N)$

N=8, L=2, H(N) = 1



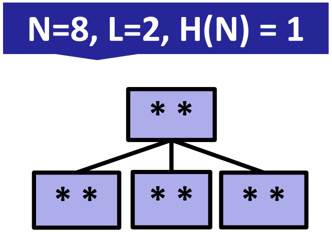


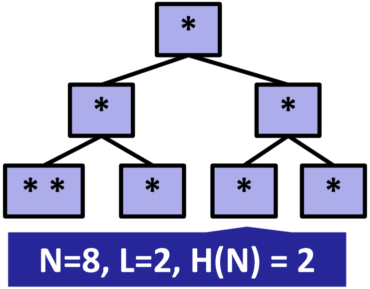
N=8, L=2, H(N) = 2

Search Runtime

- *Shortest-case number of nodes to inspect*: $\log_{L+1} N$
- *Shortest-case number of keys to inspect per node*: L
- *Runtime*: $L \log_{L+1} N \in \Theta(\log N)$

- *Tallest-case number of nodes to inspect*: $\log_2 N + 1$
- *Tallest-case number of keys to inspect per node*: 1
- *Runtime*: $\log_2 N + 1 \in \Theta(\log N)$

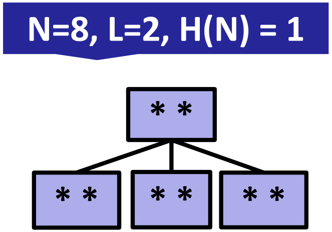


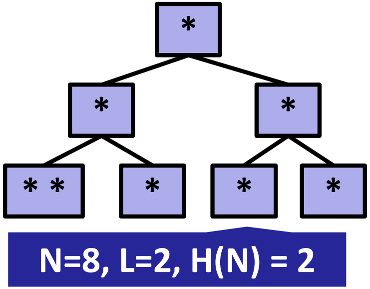


Insertion Runtime

- *Shortest-case number of nodes to inspect*: $\log_{L+1} N$
- *Shortest-case number of keys to inspect per node*: L
- *Shortest-case number of splits*: $\log_{L+1} N$
- *Runtime*: $2L \log_{L+1} N \in \Theta(\log N)$

- *Tallest-case number of nodes to inspect*: $\log_2 N + 1$
- *Tallest-case number of keys to inspect per node*: 1
- *Tallest-case number of splits*: $\log_2 N + 1$
- *Runtime*: $2\log_2 N + 2 \in \Theta(\log N)$





tl;dr

- **Search Trees** have great runtimes most of the time
  - But they struggle with sorted (or mostly-sorted) input
  - Must bound the height if we need runtime guarantees

- **Plain BSTs**: simple to reason about/implement. A good starting point

- **B-Trees** are a *Search Tree variant* that binds the height to Θ(log N) by only allowing the tree to grow from its root
  - A good choice for a Map and/or Set implementation

	LinkedList Map, Worst Case	BST Map, Worst Case	B-Tree Map, Worst Case
Find	Θ(N)	h = Θ(N)	Θ(log N)
Add	Θ(N)	h = Θ(N)	Θ(log N)
Remove	Θ(N)	h = Θ(N)	Θ(log N)