2-3 Trees

CMSC 420

Problems with AVL Trees

Have to store height, compare subtree heights for balance

Rotations are expensive, especially RL and LR

Can we do better, while maintaining $\mathcal{O}(\log n)$ search?

Yes! 2-3 Trees, B-Trees, RB-Trees

We'll start with 2-3 Trees, which will lead into the others

Properties of 2-3 Trees

Perfectly Balanced

- All nodes have either 0 children or the maximum they support
- All leaf nodes are at the same *depth*
- That is, $\forall n \in T, B(n) = 0$ (for a suitable definition of B)

How Do We Achieve Perfect Balance?

This isn't possible for a BST, unless the number of nodes is $2^a - 1$

So what do we do?

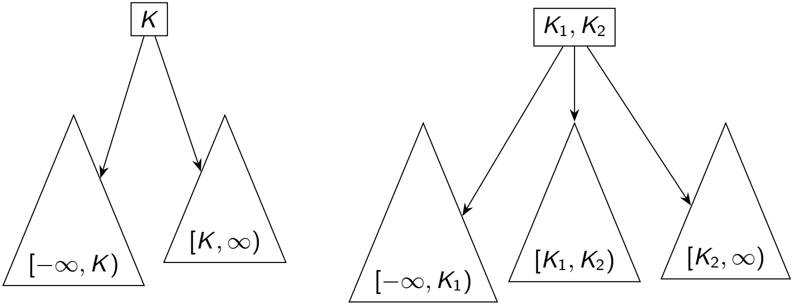
Define 2-nodes and 3-nodes

- 2-nodes have 2 children
- 3-nodes have 3 children

Node Types

A 2-node is a BST node:

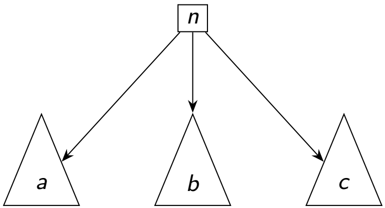
A 3-node has 2 keys:

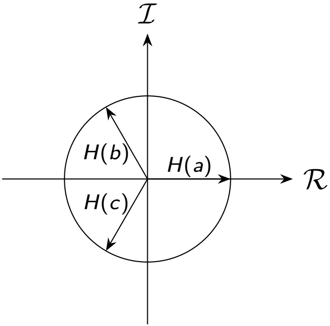


How do we define balance for a 3-node?

A Working Definition of Balance

You are only responsible for knowing the last line

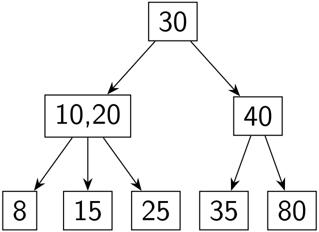




$$B(n) = H(a) + \frac{(-1+i\sqrt{3})}{2}H(b) + \frac{(-1-i\sqrt{3})}{2}H(c)$$

For our purposes, the invariant implies $H(a) = H(b) = H(c)$

An Example of a 2-3 Tree



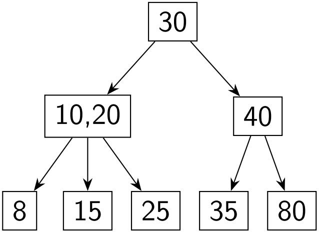
10,20 is a 3-node

The rest are 2-nodes

Search in a 2-3 Tree

This works (almost) identically to a BST

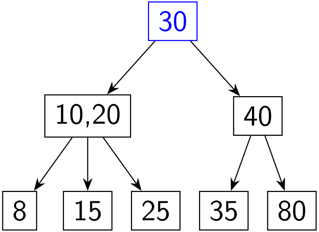
Let's say we're searching for 15



Search in a 2-3 Tree

This works (almost) identically to a BST

Let's say we're searching for 15

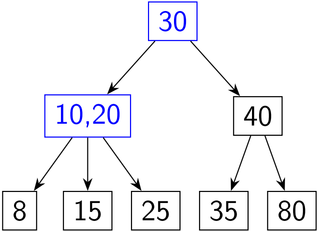


Start at the root

Search in a 2-3 Tree

This works (almost) identically to a BST

Let's say we're searching for 15



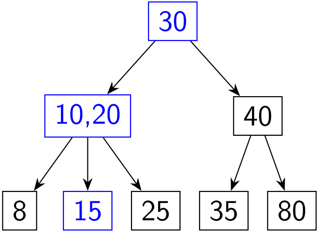
Start at the root

$15 < 30$

Search in a 2-3 Tree

This works (almost) identically to a BST

Let's say we're searching for 15



Start at the root

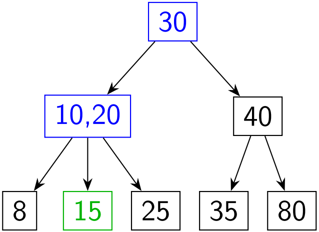
$15 < 30$

$10 \leq 15 < 20$

Search in a 2-3 Tree

This works (almost) identically to a BST

Let's say we're searching for 15



Start at the root

$15 < 30$

$10 \leq 15 < 20$

$15 = 15$

Insertion into a 2-3 Tree

Start with an empty tree

Insertion into a 2-3 Tree

Start with an empty tree

Insert an element
$\Rightarrow$ We have a 2-node

$\boxed{3}$

Insertion into a 2-3 Tree

Start with an empty tree

Insert an element
⇒ We have a 2-node

Insert another element
⇒ It expands to a 3-node

3,20

Insertion into a 2-3 Tree

Start with an empty tree

Insert an element
$\Rightarrow$ We have a 2-node

Insert another element
$\Rightarrow$ It expands to a 3-node

Insert a third element
$\Rightarrow$ A 4-node isn't allowed!

$\boxed{3,12,20}$

Insertion into a 2-3 Tree

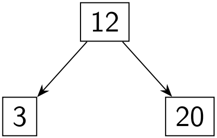
Start with an empty tree

Insert an element
⇒ We have a 2-node

Insert another element
⇒ It expands to a 3-node

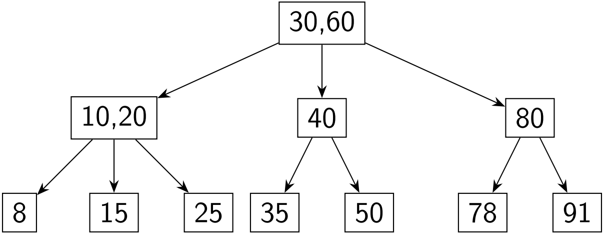
Insert a third element
⇒ A 4-node isn't allowed!

Split the node,
the middle element becomes a 2-node



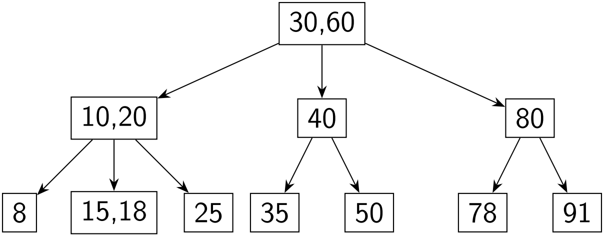
How Does a 2-3 Tree Grow?

Let's suppose it were a fancy BST:



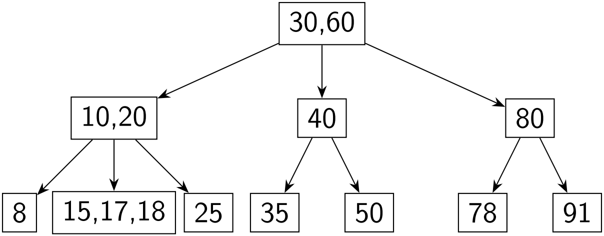
How Does a 2-3 Tree Grow?

Let's suppose it were a fancy BST:



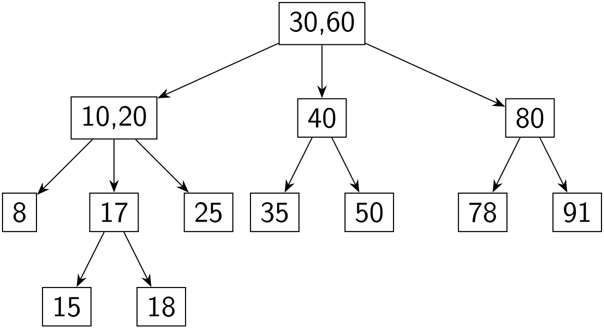
How Does a 2-3 Tree Grow?

Let's suppose it were a fancy BST:



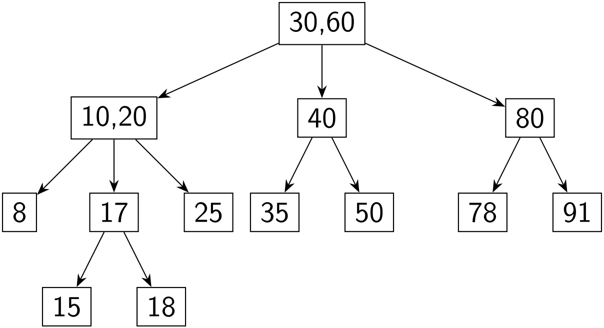
How Does a 2-3 Tree Grow?

Let's suppose it were a fancy BST:



How Does a 2-3 Tree Grow?

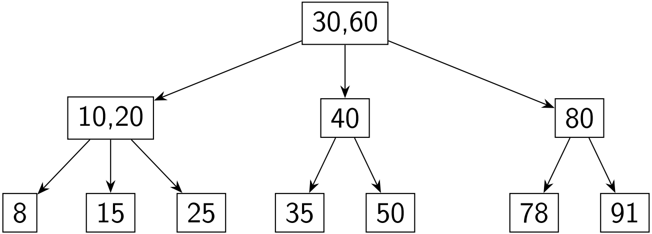
Let's suppose it were a fancy BST:



This violates our invariant!

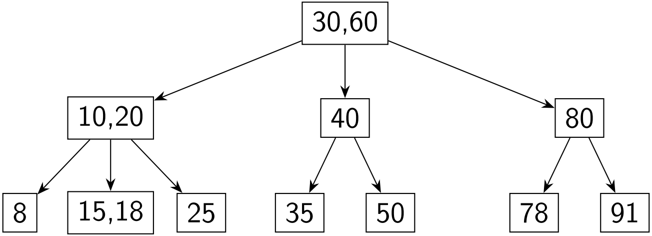
A Key Distinction

BSTs grow *downwards*, but 2-3 trees grow *upwards*!



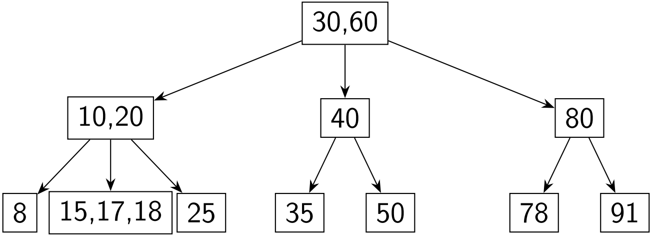
A Key Distinction

BSTs grow *downwards*, but 2-3 trees grow *upwards*!



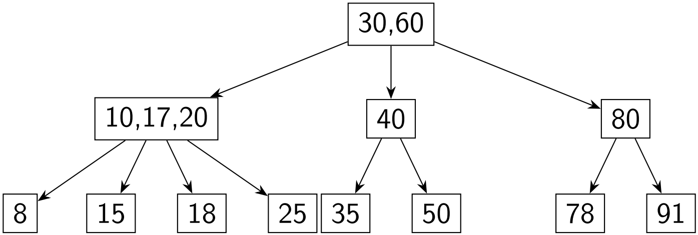
A Key Distinction

BSTs grow *downwards*, but 2-3 trees grow *upwards*!



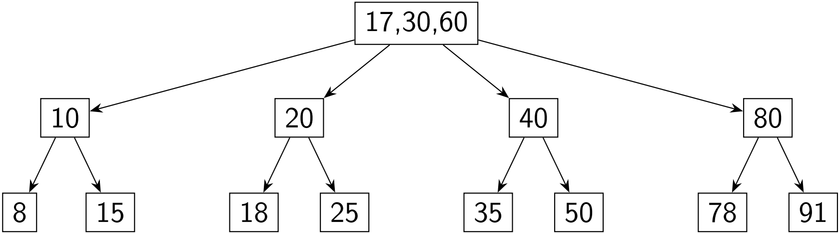
A Key Distinction

BSTs grow *downwards*, but 2-3 trees grow *upwards*!



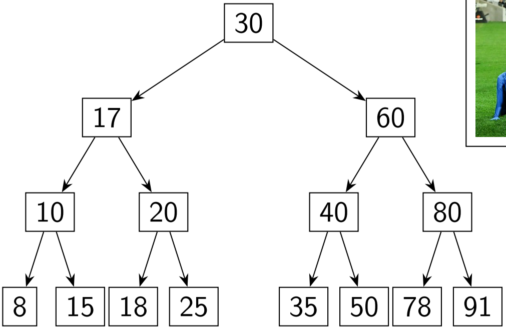
A Key Distinction

BSTs grow *downwards*, but 2-3 trees grow *upwards*!



A Key Distinction

BSTs grow *downwards*, but 2-3 trees grow *upwards*!



Insertion Abstracted

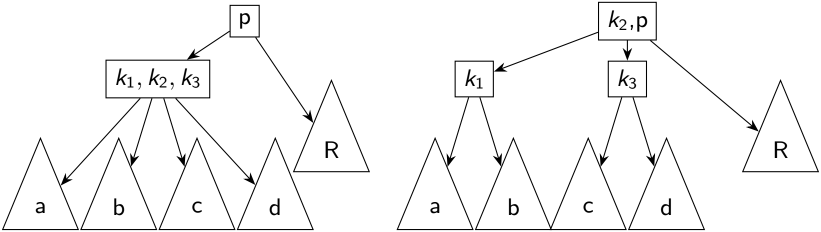
1. Search to find the appropriate leaf for this element
2. Is it a 2-node?

   yes Add the new element here, making it a 3-node, and terminate

   no Continue to the next step
3. Temporarily create a 4-node with three keys: $k_1 < k_2 < k_3$
4. Is this the root?

   yes Create a new root 2-node with k_2 and children k_1 (with children a and b) and k_3 (with children c and d); terminate

   no Create k_1 and k_3 as above; add k_2 to p; go to step 2



Keeping Trees Shorter

The previous technique works

Trees stay perfectly balanced

Tends to make more 2-nodes, which means taller trees
$\Rightarrow$ More steps to reach a leaf

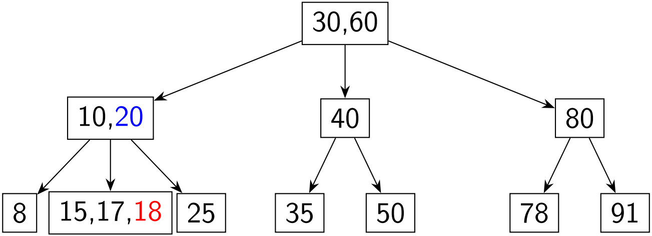
More 2-nodes $\Rightarrow \mathcal{O}(\log_2 n)$
More 3-nodes $\Rightarrow \mathcal{O}(\log_3 n)$

Note: Technically $\mathcal{O}(\log_2 n) = \mathcal{O}(\log_3 n)$

How do we avoid creating 2-nodes?

Key Rotation

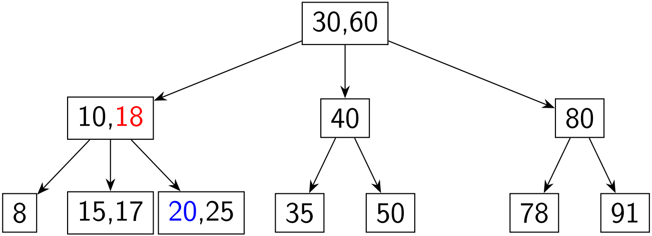
Core idea:
We might have siblings who are 2-nodes, and can expand



Key Rotation

Core idea:

We might have siblings who are 2-nodes, and can expand

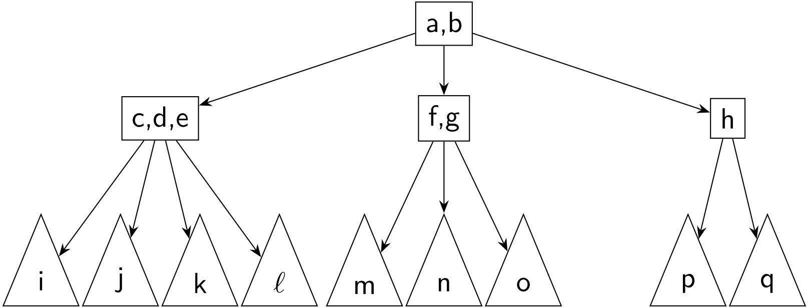


Key Rotation Abstracted

First, assign each node an *age*, increasing with keys
$\Rightarrow$ 15,17 is older than 8 and younger than 20,25

Prefer to rotate towards older siblings, starting with closest in age
$\Rightarrow$ Try younger siblings (closest first) if no available older ones

If we have a 2-node sibling:

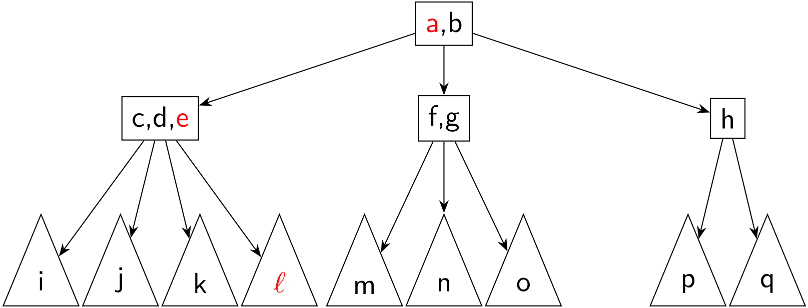


Key Rotation Abstracted

First, assign each node an *age*, increasing with keys
$\Rightarrow$ 15,17 is older than 8 and younger than 20,25

Prefer to rotate towards older siblings, starting with closest in age
$\Rightarrow$ Try younger siblings (closest first) if no available older ones

If we have a 2-node sibling:

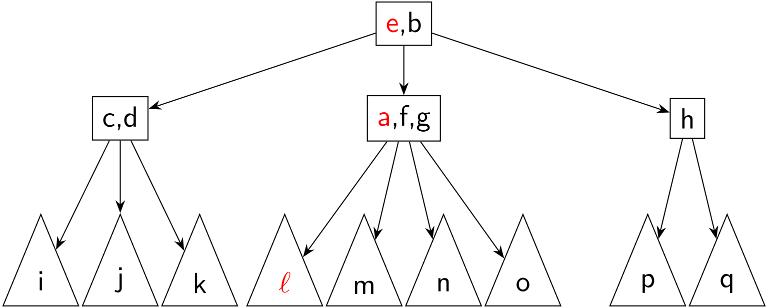


Key Rotation Abstracted

First, assign each node an *age*, increasing with keys
$\Rightarrow$ 15,17 is older than 8 and younger than 20,25

Prefer to rotate towards older siblings, starting with closest in age
$\Rightarrow$ Try younger siblings (closest first) if no available older ones

If we have a 2-node sibling:

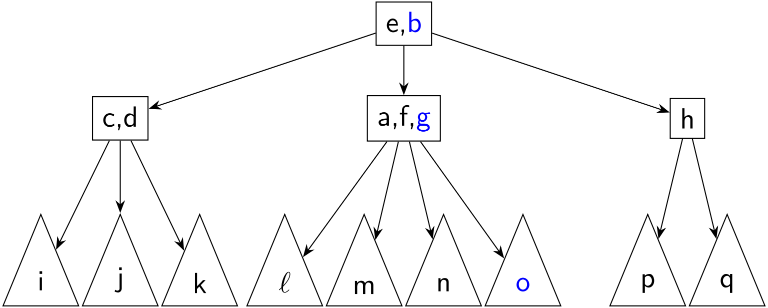


Key Rotation Abstracted

First, assign each node an *age*, increasing with keys
$\Rightarrow$ 15,17 is older than 8 and younger than 20,25

Prefer to rotate towards older siblings, starting with closest in age
$\Rightarrow$ Try younger siblings (closest first) if no available older ones

If we have a 2-node sibling:

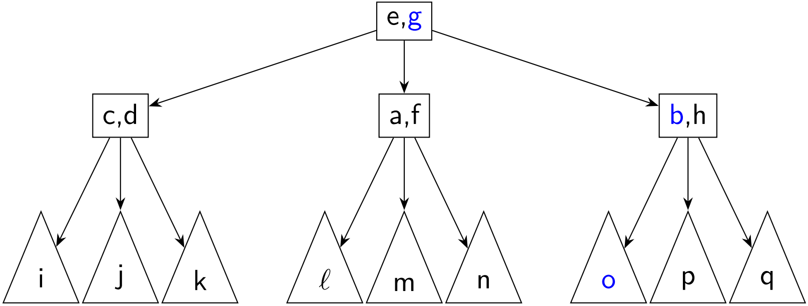


Key Rotation Abstracted

First, assign each node an *age*, increasing with keys
⇒ 15,17 is older than 8 and younger than 20,25

Prefer to rotate towards older siblings, starting with closest in age
⇒ Try younger siblings (closest first) if no available older ones

If we have a 2-node sibling:



Deletion from a 2-3 Tree

We'll start with leaves

Inner node deletions will become leaf deletions

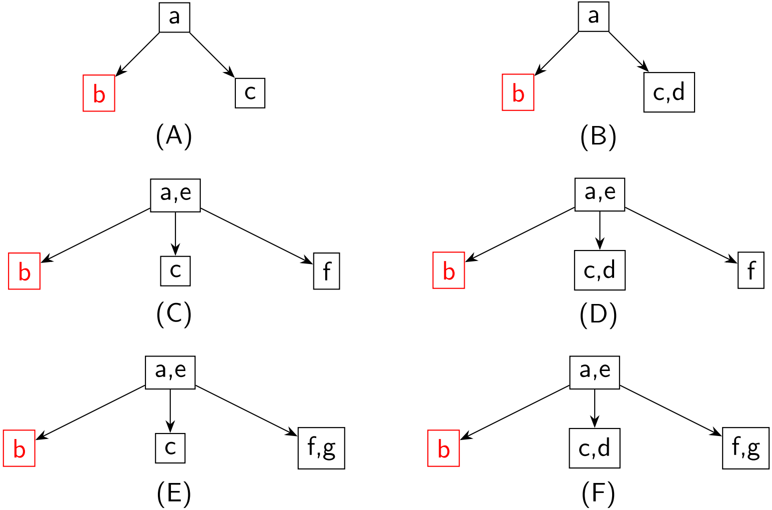
Deleting a key from a leaf 3-node is easy

$$\boxed{k_1, k_2} \Longrightarrow \boxed{k_1}$$

Deleting from a 2-node will be more complicated

Deletion from a 2-3 Tree

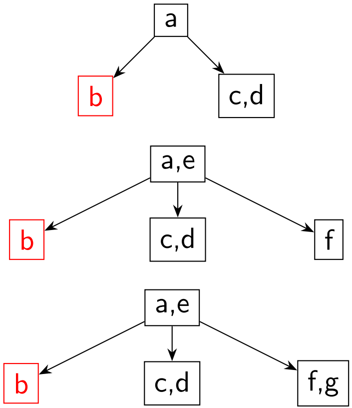
It's only interesting if this isn't the only element in the tree, so



Deletion from a 2-3 Tree

Cases B, D, and F

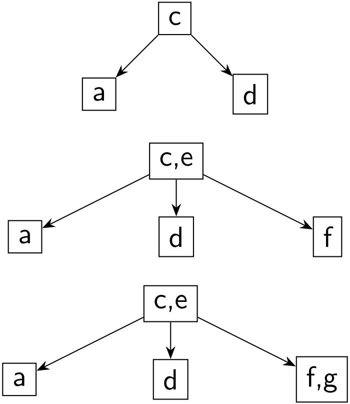
We can rotate keys to the left



Deletion from a 2-3 Tree

Cases B, D, and F

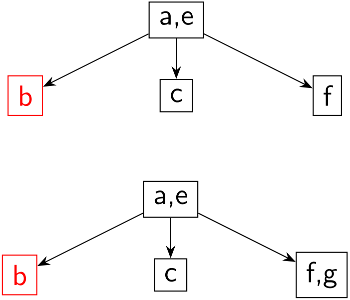
We can rotate keys to the left



Deletion from a 2-3 Tree

Cases C and E

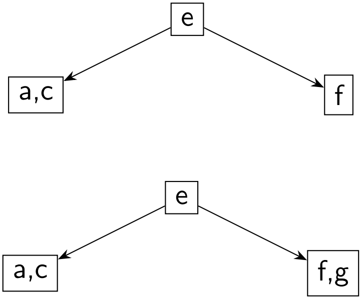
We can merge from the parent



Deletion from a 2-3 Tree

Cases C and E

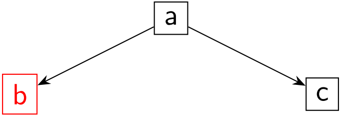
We can merge from the parent



Deletion from a 2-3 Tree

Case A

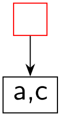
There's nothing we can rotate in this case



Deletion from a 2-3 Tree

Case A

There's nothing we can rotate in this case

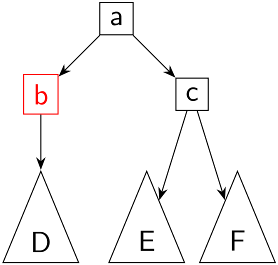


Empty node now has to be deleted, propagating upwards!

Propagating Deleted Nodes

We only need to consider three cases, corresponding to cases A, B, and C

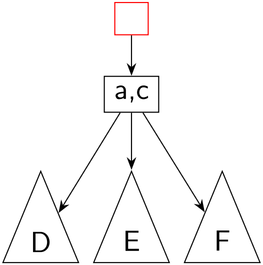
Case A:



Propagating Deleted Nodes

We only need to consider three cases, corresponding to cases A, B, and C

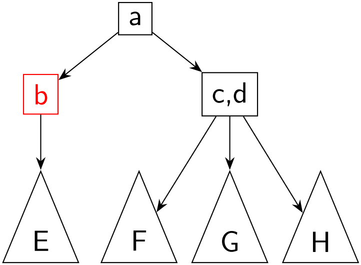
Case A:



Propagating Deleted Nodes

We only need to consider three cases, corresponding to cases A, B, and C

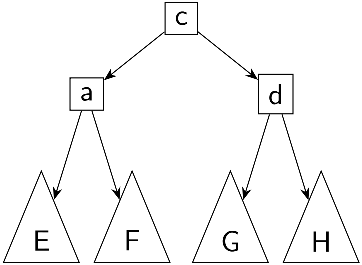
Case B:



Propagating Deleted Nodes

We only need to consider three cases, corresponding to cases A, B, and C

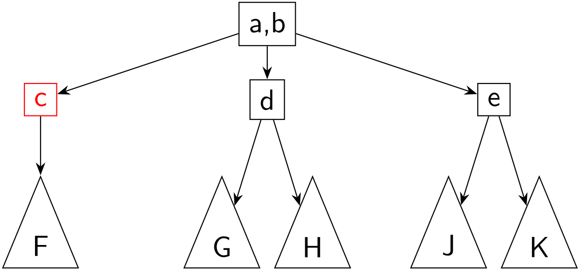
Case B:



Propagating Deleted Nodes

We only need to consider three cases, corresponding to cases A, B, and C

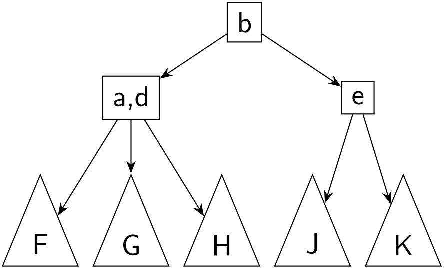
Case C:



Propagating Deleted Nodes

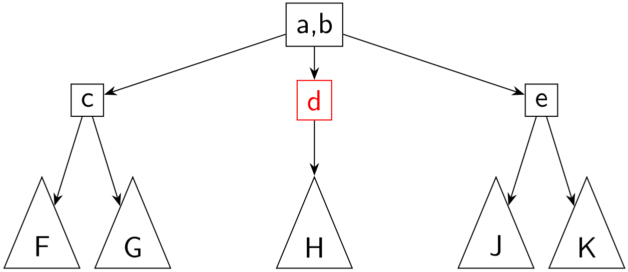
We only need to consider three cases, corresponding to cases A, B, and C

Case C:



Propagating Deleted Nodes

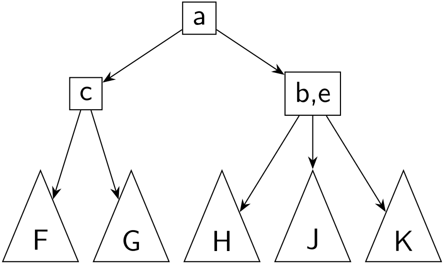
There's one additional wrinkle:



Do we merge *a* and *c* or *b* and *e*?

Propagating Deleted Nodes

There's one additional wrinkle:



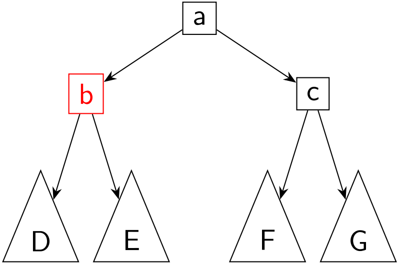
Do we merge *a* and *c* or *b* and *e*?

Always merge **right** when given the option!

$\Rightarrow$ This will be important when we cover B-Trees

Deleting Interior Keys

We haven't looked at any deletions like:



Why not?

In-Order Successors!

Like with BSTs, 2-3 Trees replace a removed interior item with its in-order successor

If the successor is still in an interior node, we continue with its successor

This will ultimately result in reaching a leaf node

Deletion is Expensive

Deleting a single item can cause a cascade of deletions

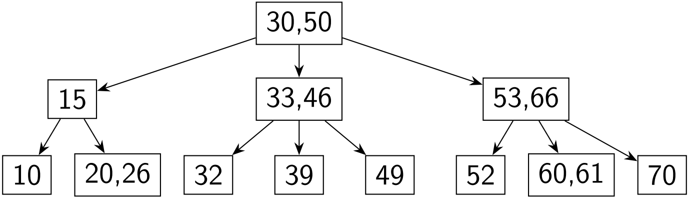
This might go all the way back to the root!

We search a *lot* more than we delete items, so efficient search is more valuable than efficient deletion

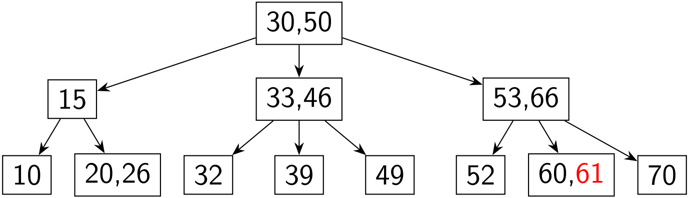
Many implementations use *Mark-and-Sweep*, both for 2-3 trees and AVL trees

- Delete less often
- May be able to combine deletions for efficiency gains

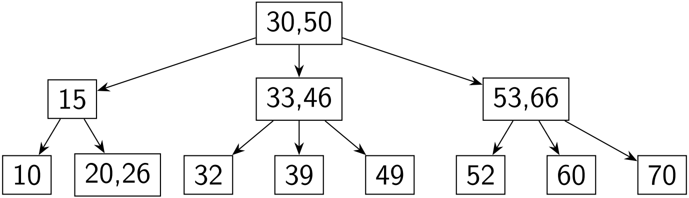
Deletion Example



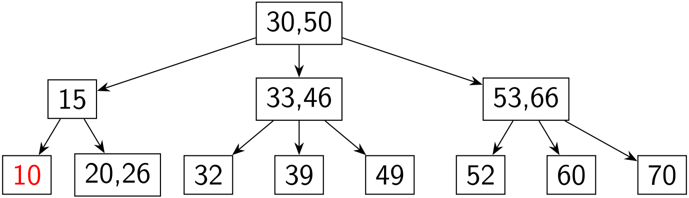
Deletion Example



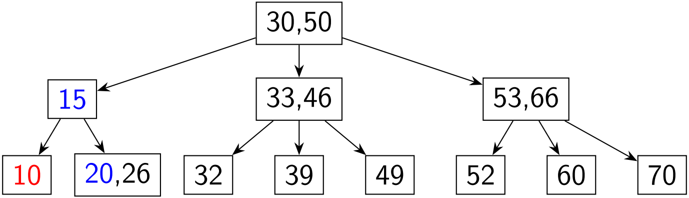
Deletion Example



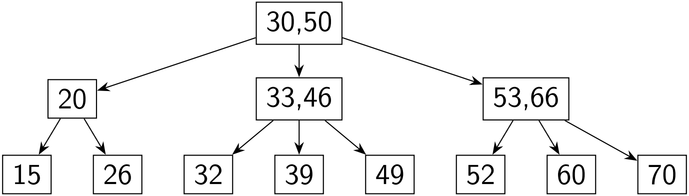
Deletion Example



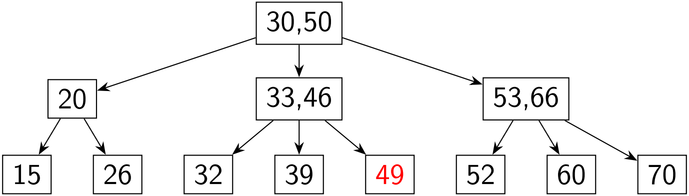
Deletion Example



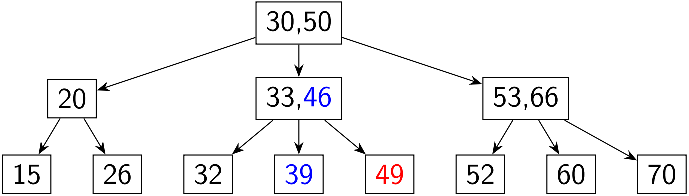
Deletion Example



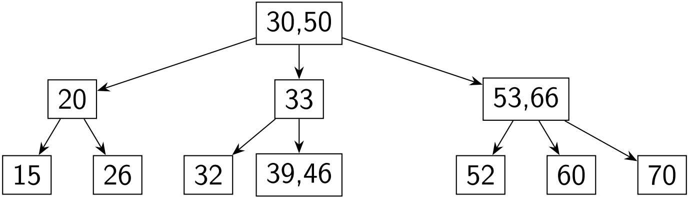
Deletion Example



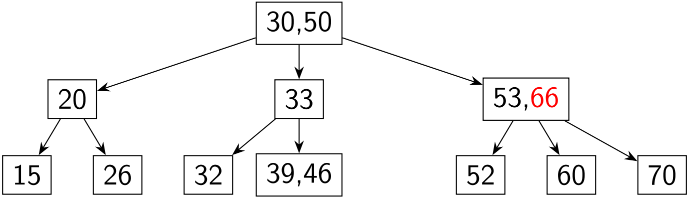
Deletion Example



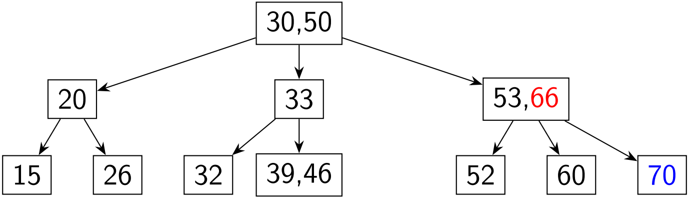
Deletion Example



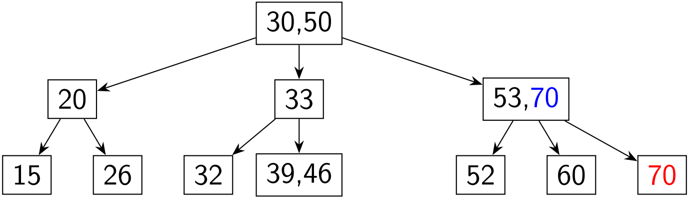
Deletion Example



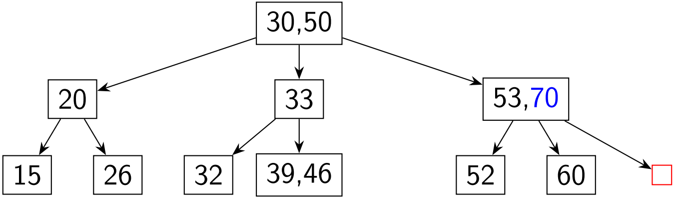
Deletion Example



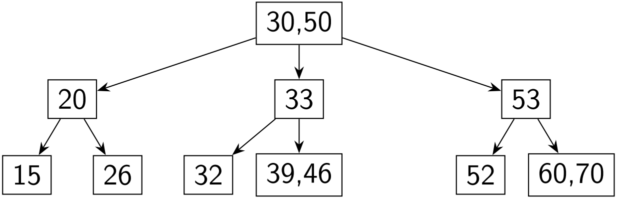
Deletion Example



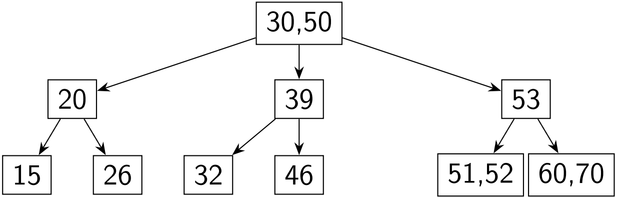
Deletion Example



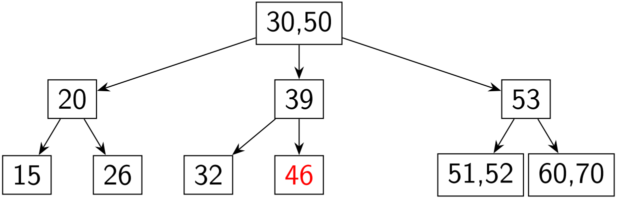
Deletion Example



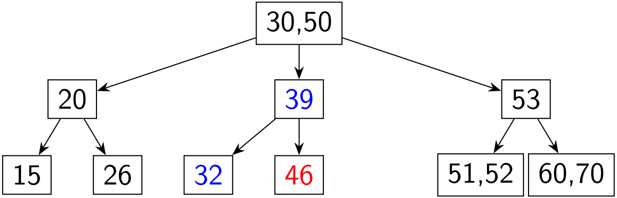
Deletion Example



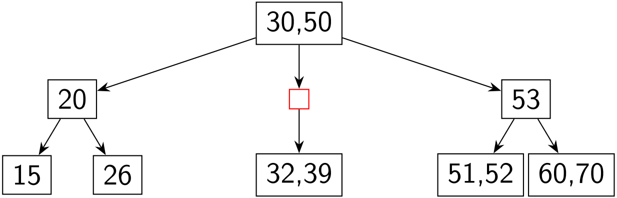
Deletion Example



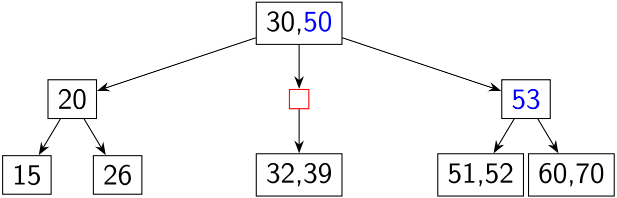
Deletion Example



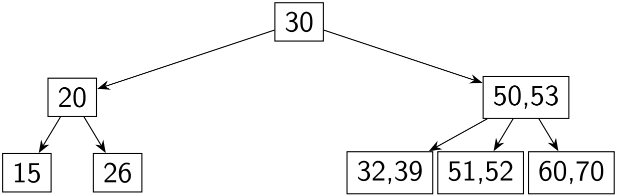
Deletion Example



Deletion Example



Deletion Example



Implementing 2-3 Trees as Binary Trees

We can implement a 2-3 tree as a binary tree!

This is called a Red-Black Binary Search Tree

- Also known as an RBBST
- Also known as an RB-Tree

This will be our next topic