2-3 Trees continued
Other Self-Balancing Search Trees

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, November 13, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu
© 2005–2020 Glenn G. Chappell
Some material contributed by Chris Hartman

Review

Review
Where Are We? — The Big Challenge

Our problem for most of the rest of the semester:

- Store: A collection of data items, all of the same type.
- Operations:
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- Time & space efficiency are desirable.

A solution to this problem is a **container**.

In a **generic container**, client code can specify the value type.

Unit Overview
Tables & Priority Queues

Major Topics

- ✓ Introduction to Tables ← Lots of lousy implementations
- ✓ Priority Queues
- ✓ Binary Heap Algorithms
- ✓ Heaps & Priority Queues in the C++ STL

} Idea #1: Restricted Table

- (part) 2-3 Trees
- Other self-balancing search trees

} Idea #2: Keep a tree balanced

- Hash Tables

} Idea #3: Magic functions

- Prefix Trees ← A special-purpose implementation: "the Radix Sort of Table implementations"
- Tables in the C++ STL & Elsewhere

Review
Introduction to Tables

A **Table** allows for arbitrary key-based look-up.

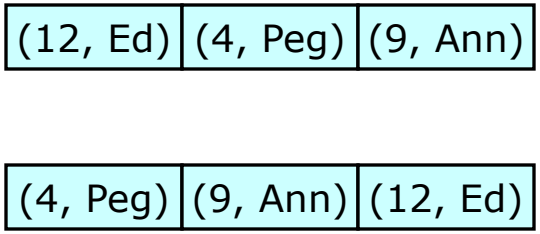
Three single-item operations: retrieve, insert, delete by key.

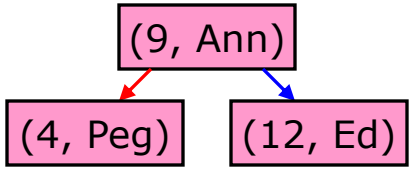
A Table implementation typically holds **key-value pairs**.

Table

Key	Value
12	Ed
4	Peg
9	Ann

Inefficient Implementations





Three ideas for efficient implementations:

1. Restricted Table → Priority Queues
2. Keep a tree balanced → Self-balancing search trees
3. Magic functions → Hash Tables

Overview of Advanced Table Implementations

We cover the following advanced Table implementations.

- Self-balancing search trees
  - To make things easier, allow more children (?):
    - (part) **2-3 Tree**
      - Up to 3 children
    - **2-3-4 Tree**
      - Up to 4 children
    - **Red-Black Tree**
      - Binary Tree representation of a 2-3-4 Tree
  - Or back up and try for a strongly balanced Binary Search Tree again:
    - **AVL Tree**
- Alternatively, forget about trees entirely:
  - **Hash Table**
- Finally, "the Radix Sort of Table implementations":
  - **Prefix Tree**

Idea #2:
Keep a tree balanced

Later, we will cover other self-balancing search trees: B-Trees, B+ Trees.

Idea #3:
Magic functions

Review
2-3 Trees [1/5]

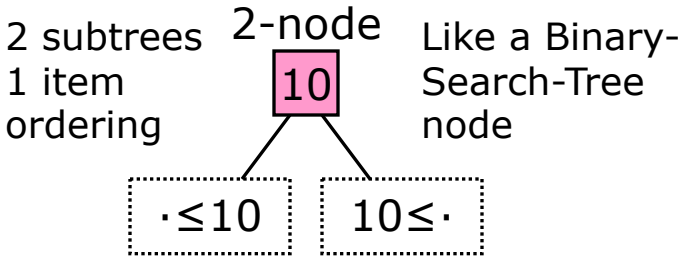
A Binary-Search-Tree style node is a **2-node**.

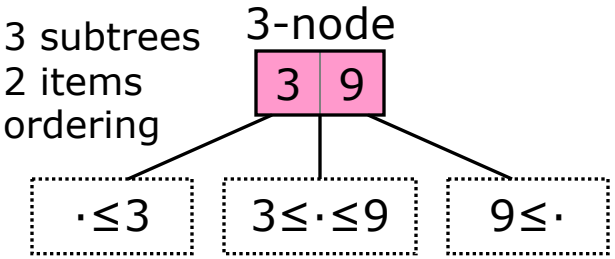
- This is a node with 2 subtrees and 1 item.
- The item's value lies between the values in the two subtrees.

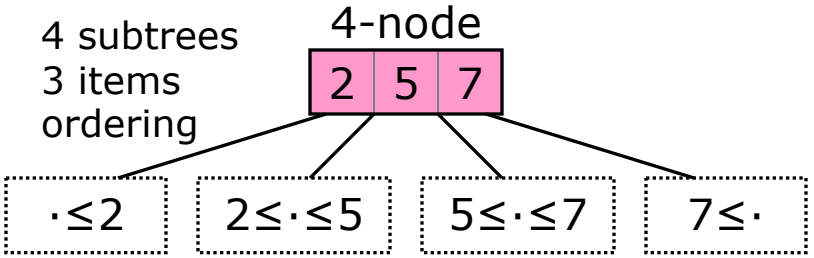
In a *2-3 Tree* we also allow a node to be a **3-node**.

- This is a node with 3 subtrees and 2 items.
- Each of the 2 items has a value that lies between the values in the corresponding pair of consecutive subtrees.

Later, we will look at *2-3-4 Trees*, which can also have **4-nodes**.



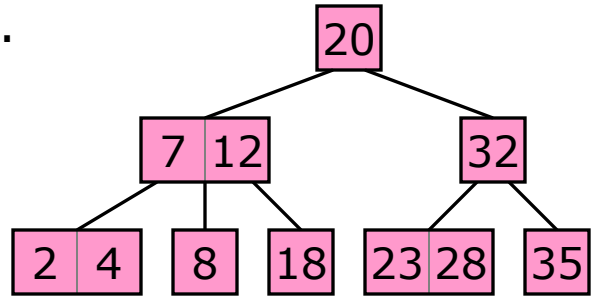




Review
2-3 Trees [2/5]

A **2-3 Search Tree** (generally just **2-3 Tree**) is a tree with the following properties [John Hopcroft 1970].

- Each node is either a 2-node or a 3-node. The associated order properties hold.
- Each node either has its full complement of children, or else is a leaf.
- All leaves lie in the same level.



To **traverse** in a 2-3 Tree:

- Use the appropriate generalization of inorder traversal.
- Items are visited in sorted order.

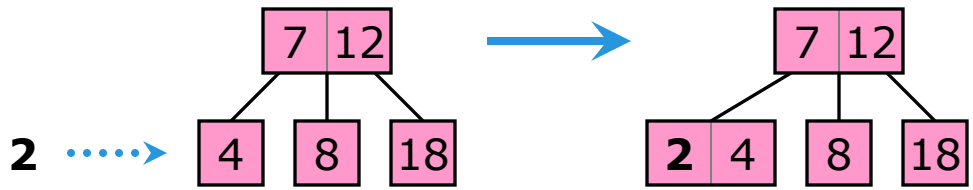
To **retrieve** by key in a 2-3 Tree:

- Begin at the root and go down, using the order properties, until either the key is found, or it is clearly not in the tree.

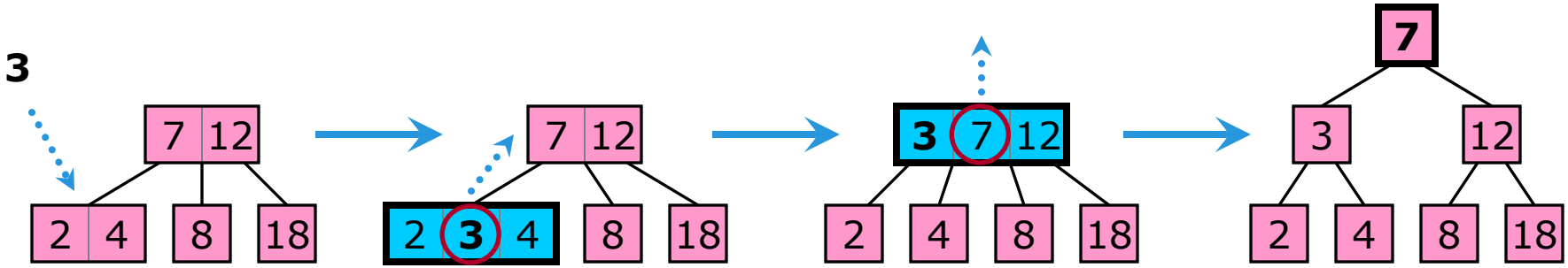
Review
2-3 Trees [3/5]

To **insert** in a 2-3 Tree:

- Find the leaf that the new item should go in.
- If it fits, then simply put it in.

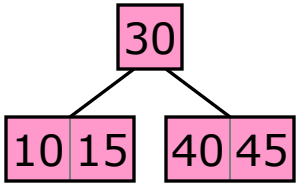


- Otherwise, there is an **over-full** node (shown in **blue**). Split it, and move the **middle** item up. Either recursively insert this item in the parent, or else create a new root, if there is no parent.



Review
2-3 Trees [4/5] (Try It!)

Do **insert 20** in the following 2-3 Tree. Draw the resulting tree.

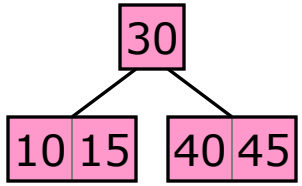


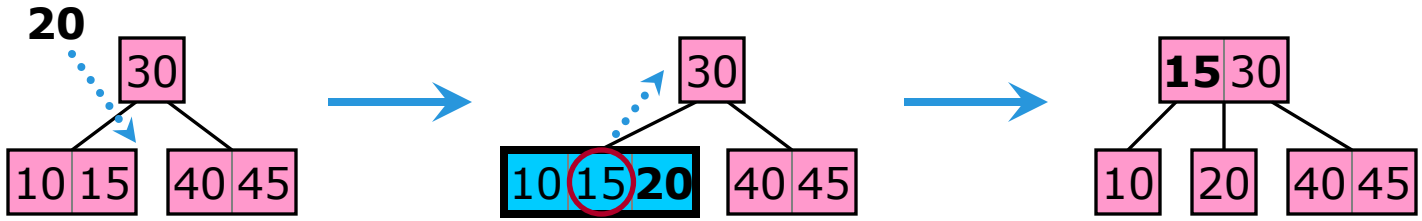
Answer on next slide.

Review
2-3 Trees [5/5] (Try It!)

Do **insert 20** in the following 2-3 Tree. Draw the resulting tree.

Answer





2-3 Trees

continued

2-3 Trees
Delete Algorithm [1/10]

Deleting from a 2-3 Tree is similar to inserting, but a bit more complicated.

- As with inserting, we start at a leaf and work our way up.
  - Simply delete from a leaf, if possible.
  - If that does not work, then do a *rotation*, if possible.
  - If neither works, then bring an item from the parent down. This is like "deleting" from the parent. Recursively apply the delete procedure to the parent, dragging subtrees along as appropriate. If we end up "deleting" the root, then the height of the tree goes down by one.

I call these three **easy case**, **semi-easy case**, and **hard case**, respectively.

2-3 Trees
Delete Algorithm [2/10]

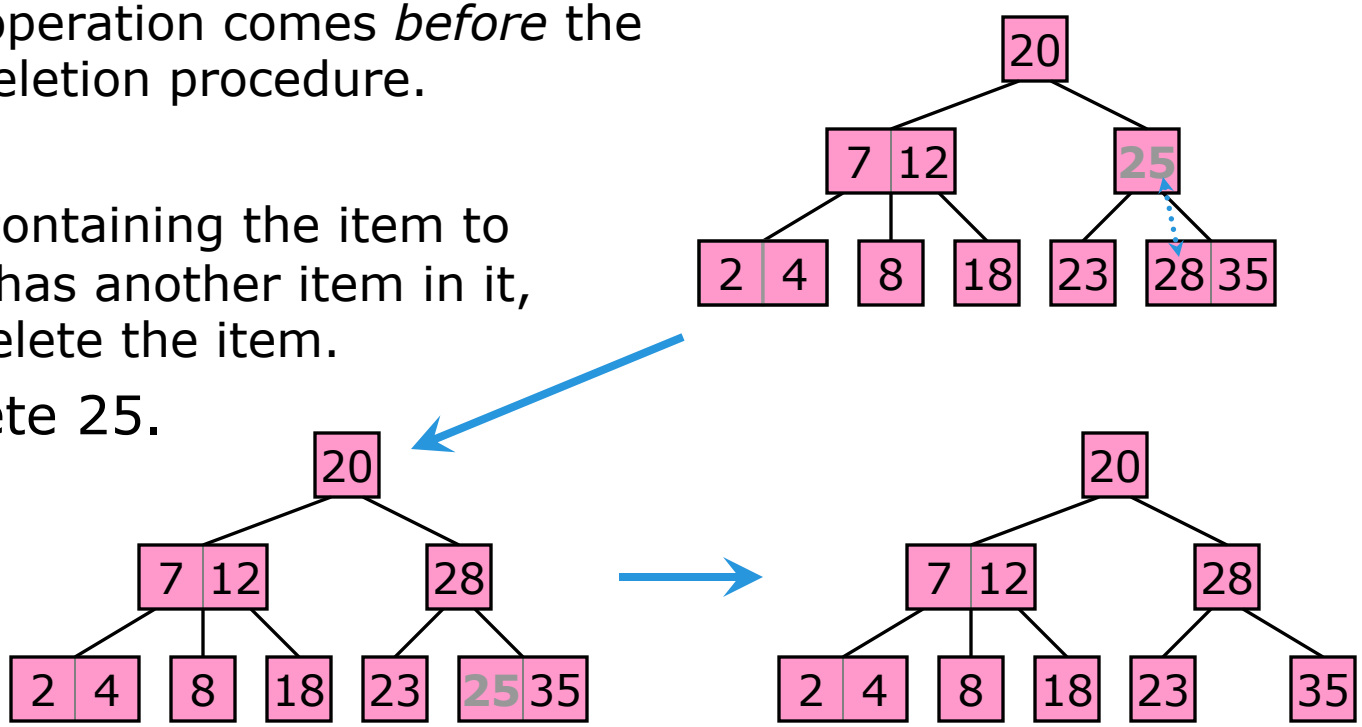
Observation

- We can always start our deletion at a leaf.
- If the item to be deleted is not in a leaf, then swap it with its successor in the sorted traversal order.
  - It must have a successor. (Why?)
- This swap operation comes *before* the recursive deletion procedure.

Easy Case

- If the leaf containing the item to be deleted has another item in it, then just delete the item.

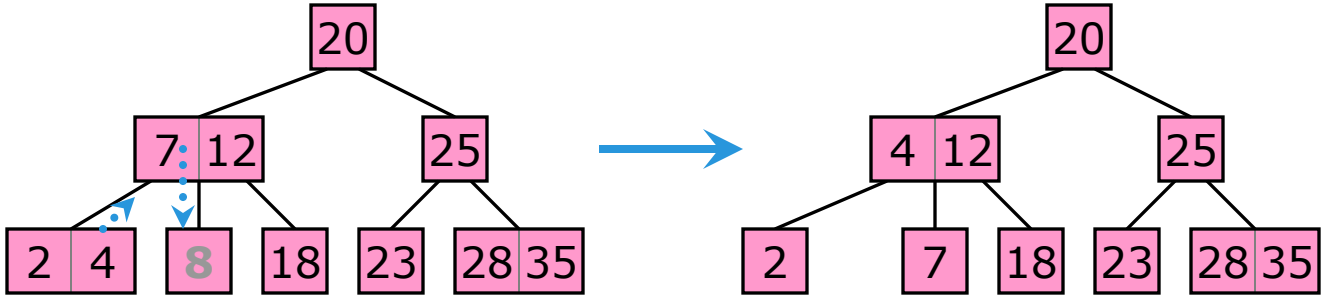
Example 1. Delete 25.



Semi-Easy Case

- Suppose the item to be deleted is in a node that contains no other item.
- If, next to this node, there is a sibling that contains 2 items, we can perform a **rotation** using the parent.

Example 2. Delete 8.

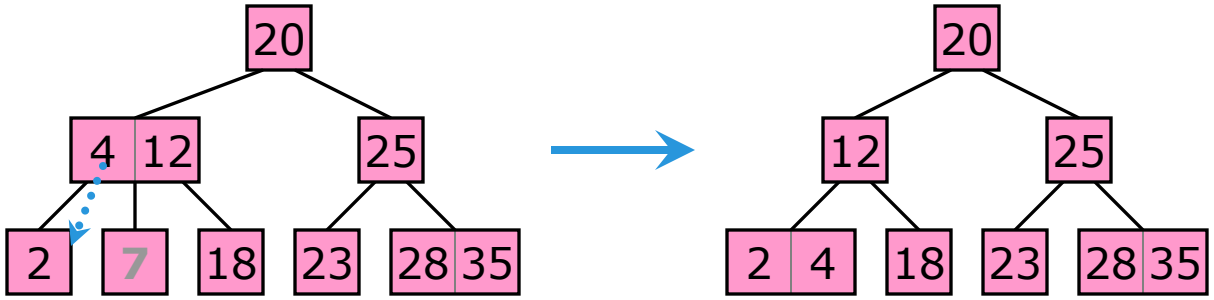


2-3 Trees
Delete Algorithm [4/10]

Hard Case

- If the item to be deleted is in a node with no other item, and there are no nearby 2-item siblings, then we bring down an item from the parent and place it in a nearby sibling node.
- Bringing down an item requires recursively applying the delete procedure on the next level up, dragging subtrees along as needed.

Example 3. Delete 7.



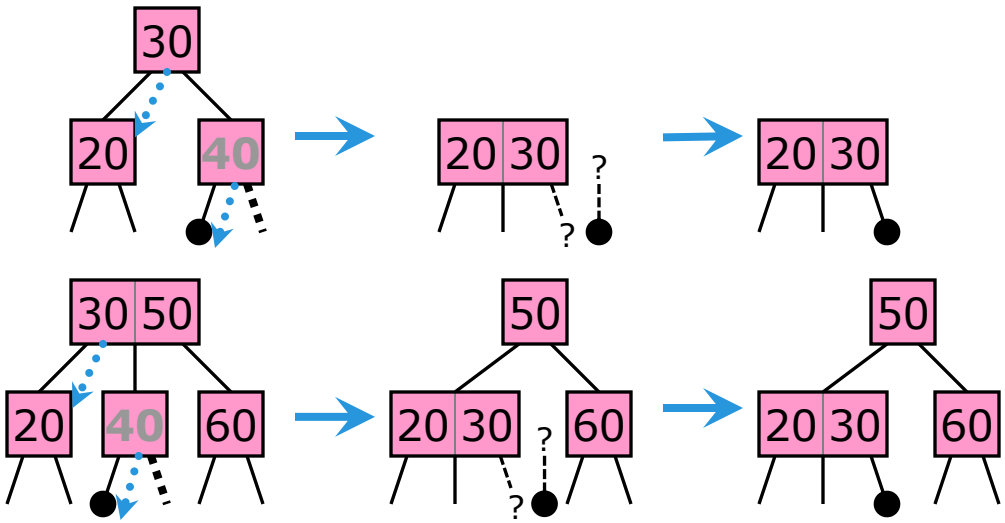
In the above example, recursively "delete" 4 from the tree consisting of the top two levels. Since 4's node has another item in it, this is *easy case*; we simply get rid of 4 (in the parent; one level down, it goes in the node containing 2).

2-3 Trees
Delete Algorithm [5/10]

When we "delete" above the leaf level, where do *orphaned subtrees* go?

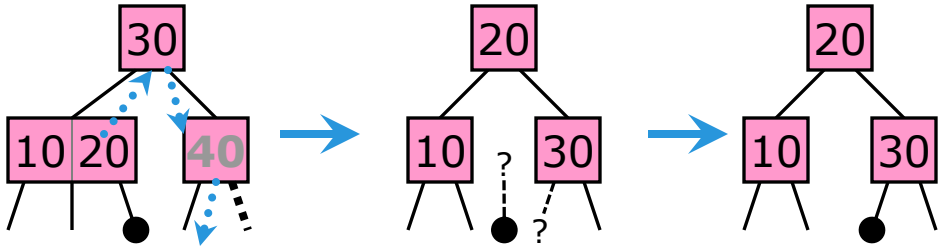
Two *hard case* examples.

- "Delete" 40. One of its subtrees is going away, and it is moving down. What happens to other subtree?
- Answer. Make it a subtree of the item we bring down.



A *semi-easy case* example.

- "Delete" 40. One of its subtrees is going away, and it is moving down. 30 is coming down to replace it. 20 is going up. What to do with the right subtree of 20?
- Answer. Make it the left subtree of 30.



There is always exactly one spot available for an orphaned subtree.

2-3 Trees
Delete Algorithm [6/10]

2-3 Tree **Delete** Algorithm (outline)

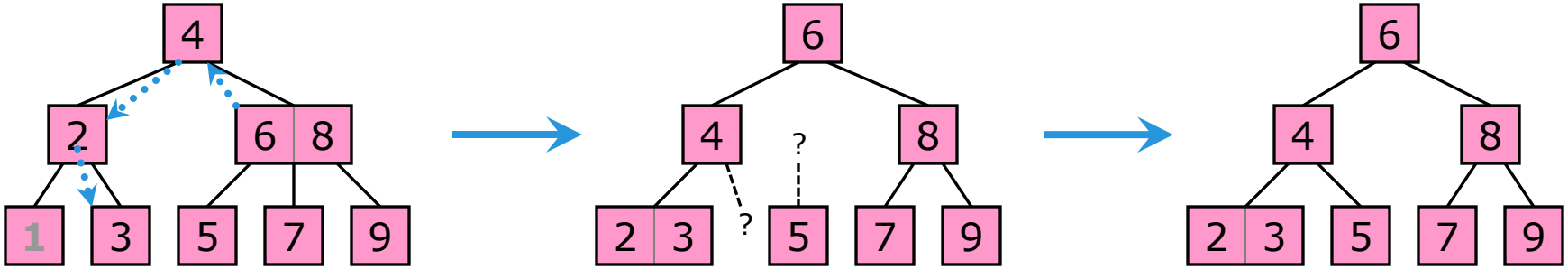
- Find the node holding the given key.
  - If it turns out that the given key is not in the tree, then act accordingly.
- If the above node is not a leaf, then swap its item with its successor in the traversal ordering. Continue with the deletion procedure: delete the given key from its new (leaf) node.
- 3 Cases
  - **Easy Case** (item shares a node with another item). Delete item. Done.
  - **Semi-Easy Case** (otherwise: item has a consecutive sibling holding 2 items). Do rotation: sibling item up, parent down, to replace the item to be deleted. Done.
  - **Hard Case** (otherwise). Eliminate the node holding the item, and move item from the parent down, adding it to consecutive sibling node. If the parent is the root, then reduce the height of the tree. Otherwise, eliminate the item from the parent by recursively applying the deletion procedure—dragging subtrees along.

2-3 Trees
Delete Algorithm [7/10]

A few more examples.

Example 4. Delete 1.

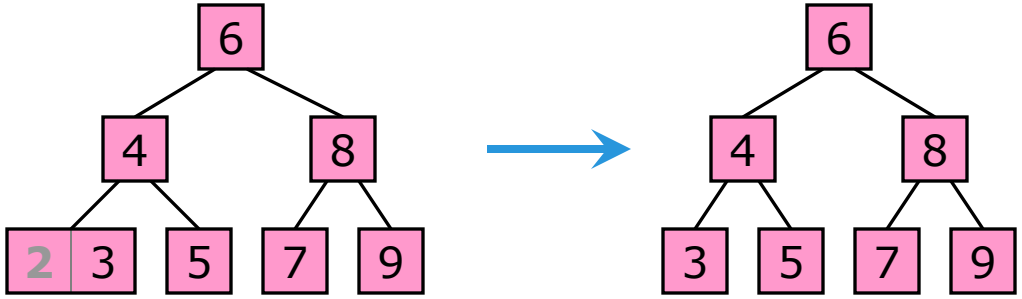
- 1 is *hard case*, so we bring down the parent (recursively "delete" 2) and join it with 3 in a single node.
- 2 is *semi-easy case*, so rotate (6 to 4 to 2).
- The 5 node is orphaned. Make it the right child of the 4 node.



2-3 Trees
Delete Algorithm [8/10]

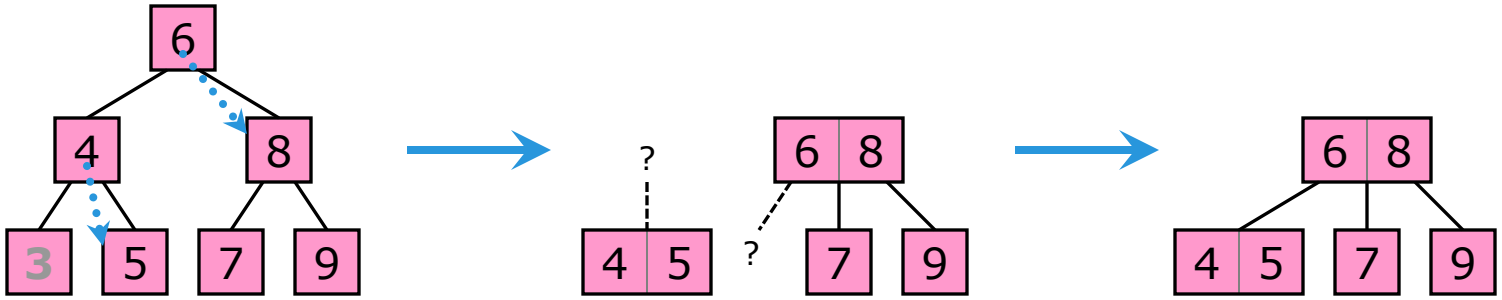
Example 5. Delete 2.

- 2 is *easy case*.



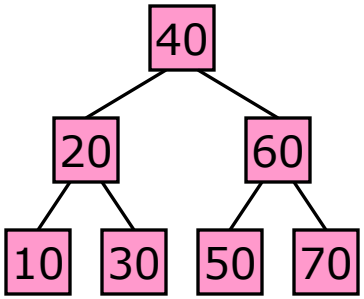
Example 6. Delete 3.

- 3 is *hard case*. We need to bring down 4 and join it with 5.
- 4 is *hard case*. We need to bring down 6 and join it with 8.
- 6 is the root. Reduce the height of the tree.
- The 4-5 node is orphaned. Make it the left child of the new root.



2-3 Trees
Delete Algorithm [9/10] (Try It!)

Do **delete 20** in the following 2-3 Tree. Draw the resulting tree.

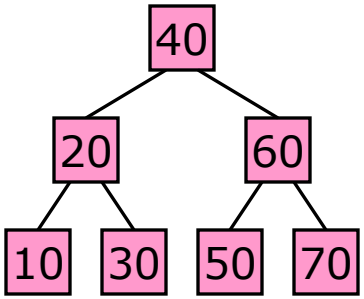


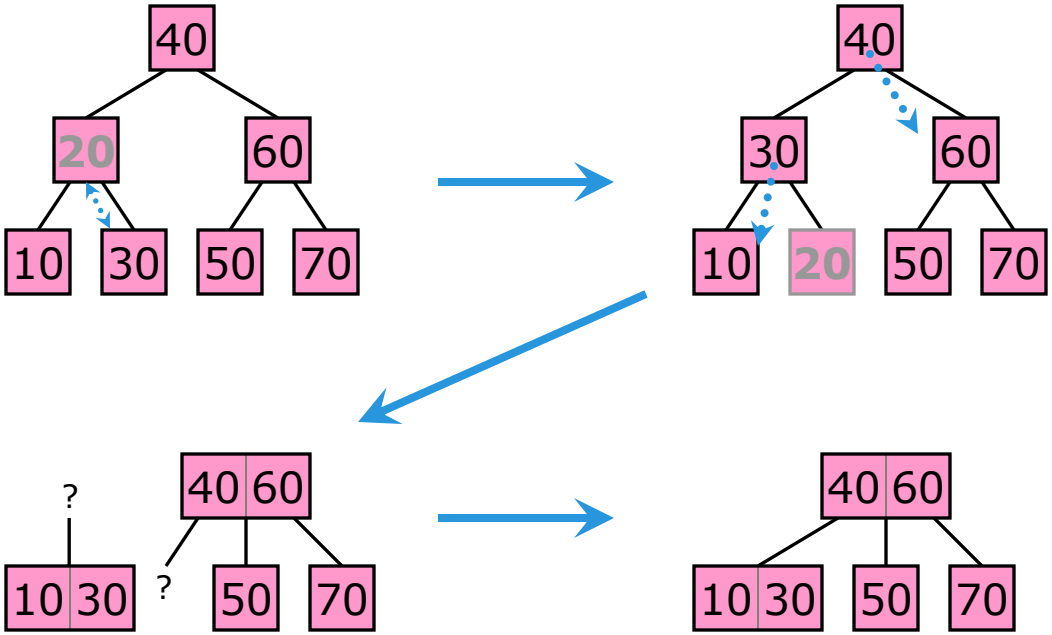
Answer on next slide.

2-3 Trees
Delete Algorithm [10/10] (Try It!)

Do **delete 20** in the following 2-3 Tree. Draw the resulting tree.

Answer





2-3 Trees
Efficiency

What is the order of the following operations for a 2-3 Tree?

- Traverse
  - $\Theta(n)$ [as usual].
- Retrieve
  - $\Theta(\log n)$.
  - The number of steps is roughly proportional to the height of the tree.
- Insert
  - $\Theta(\log n)$.
  - Comments as for Retrieve.
- Delete
  - $\Theta(\log n)$.
  - Comments as for Retrieve.

This is the first time we have seen a delete-by-key that handles an arbitrary key and is faster than linear time.

This is what we have been looking for.

A 2-3 Tree is a good basis for an implementation of a Table.

2-3 Trees
In Practice

Q. When are 2-3 Trees used in practice?

A. Pretty much never.

The reason for this is that, when we are considering how to implement a Table, a 2-3 Tree is never quite the best solution. There is always something else that is a little better. (A 2-3 Tree is a *good* solution, but never the *best* solution.)

Q. Why, then, are we studying 2-3 Trees in such detail?

A. Some of the self-balancing search trees that are actually used in Table implementations are based on 2-3 Trees, but have added complexity. Studying 2-3 Trees helps us understand them.

We will *not* study these other self-balancing search trees in detail. Rather, we will note that they are just like 2-3 Trees, except …

Other Balanced Search Trees

Other Self-Balancing Search Trees Overview

A 2-3 Tree is a kind of **self-balancing search tree**. There are many other kinds.

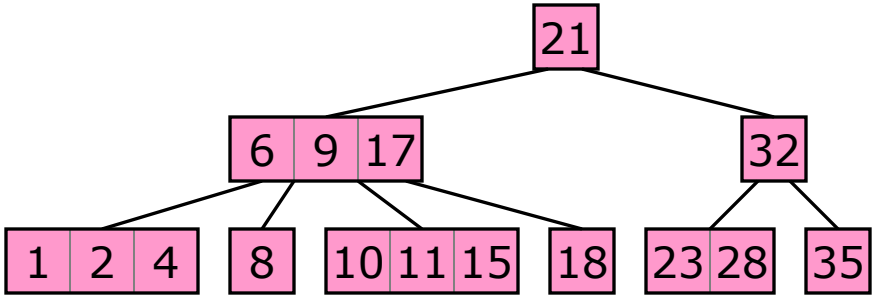
These are mostly *not* strongly balanced Binary Search Trees, but most of them do place limits on height in a way that allows for retrieve, insert, and delete to be logarithmic-time operations.

We look briefly at the following self-balancing search trees, all of which have $\Theta(\log n)$ retrieve, insert, and delete.

- 2-3-4 Trees
  - Very much like 2-3 Trees, but allowing 4-nodes.
- Red-Black Trees
  - Binary Search Tree representation of a 2-3-4 Tree. Each node also holds a Boolean value.
- AVL Trees
  - Actual strongly balanced Binary Search Trees, but with a bit of extra data in each node, telling which subtree has greater height.

Other Self-Balancing Search Trees
2-3-4 Trees

Suppose we generalize a 2-3 Tree slightly, by allowing 4-nodes. The result is a **2-3-4 Search Tree** (generally just **2-3-4 Tree**) [Rudolf Bayer 1972].



In all other ways, the definition of a 2-3-4 Tree is exactly like that of a 2-3 Tree.

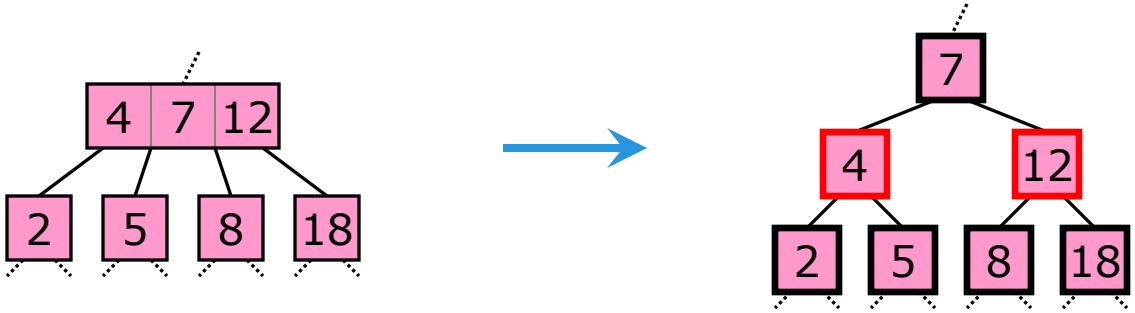
Insert & delete algorithms for 2-3-4 Trees are very similar to those for 2-3 Trees.

They tend to be just a little faster.

Other Self-Balancing Search Trees
Red-Black Trees — Idea [1/3]

We can increase the efficiency of 2-3-4 Tree operations by representing the tree using a Binary Search Tree plus a little more information. The result is a **Red-Black Tree** [Leonidas J. Guibas & Robert Sedgewick 1978].

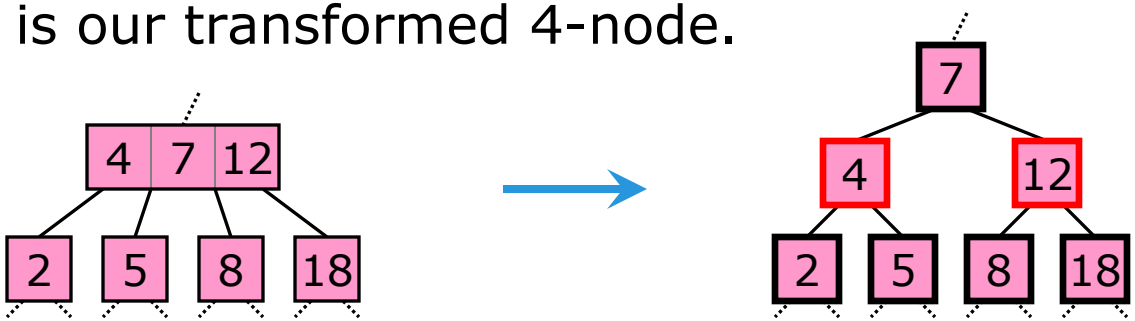
Consider the 4-node on the left, below. We can represent this part of the 2-3-4 Tree using only 2-nodes if we add two new nodes (shown in **red**).



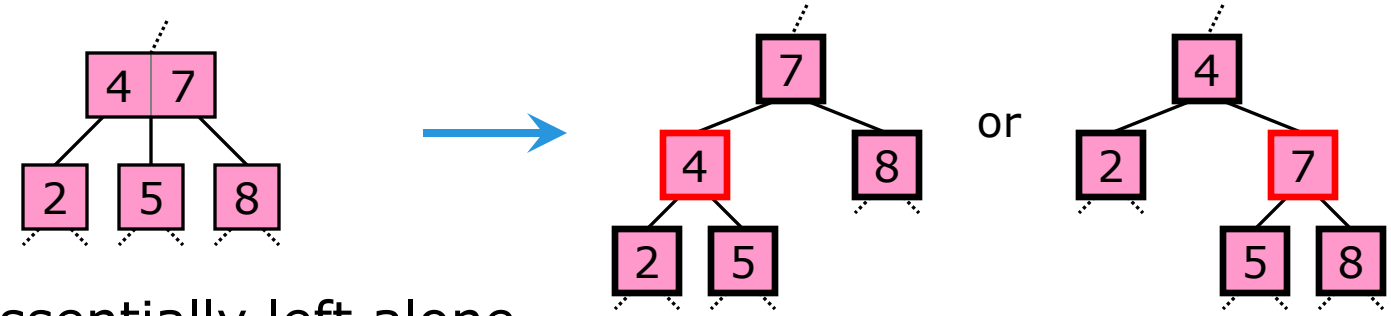
The ordering property of the 2-3-4 Tree translates into the ordering property of a Binary Search Tree.

Other Self-Balancing Search Trees
Red-Black Trees — Idea [2/3]

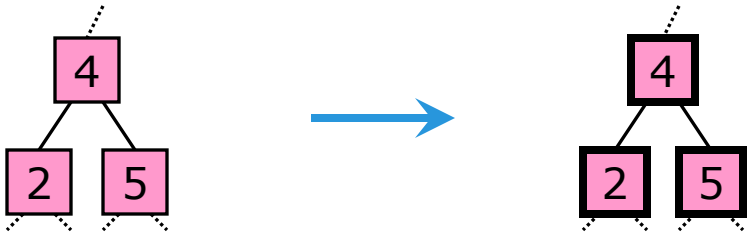
Here again is our transformed 4-node.



We can also apply this process to a 3-node—in two different ways.



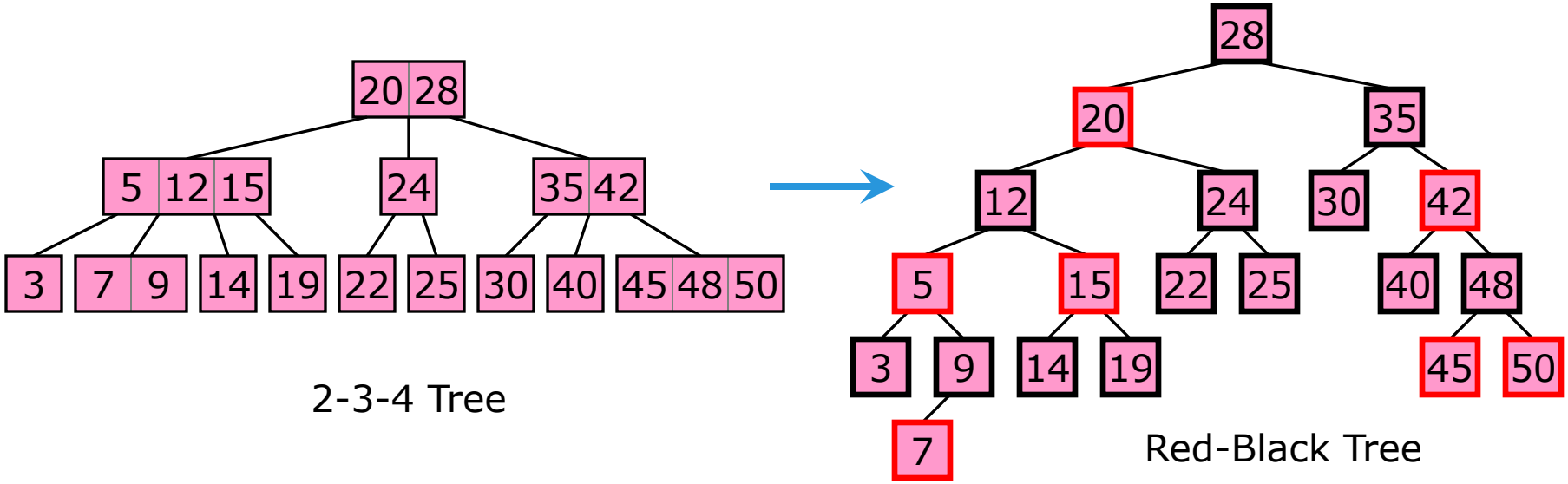
2-nodes are essentially left alone.



Other Self-Balancing Search Trees
Red-Black Trees — Idea [3/3]

A **Red-Black Tree** is a Binary-Tree representation of a 2-3-4 Tree.

- A RBT is a Binary Search Tree in which each node is **red** or **black**.
- Think of **black** nodes as representing 2-3-4 Tree nodes.
- **Red** nodes are extras needed since each node only holds one item.

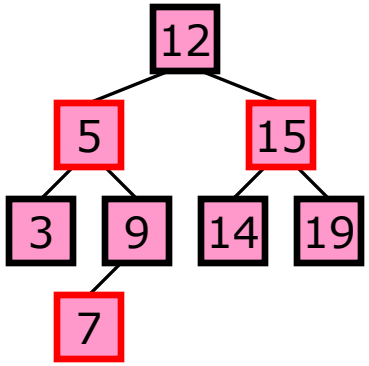


The leaves are no longer all at the same level. However, given a node, every path from it down to a leaf goes through the same number of **black** nodes. Also, no red node has a red child.

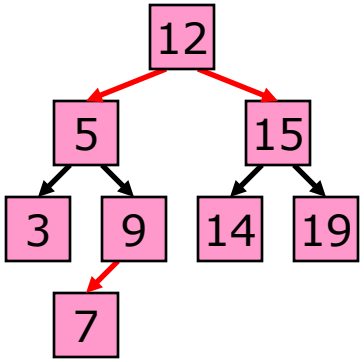
Other Self-Balancing Search Trees
Red-Black Trees — Implementations

Implementations of Red-Black Trees vary a bit.

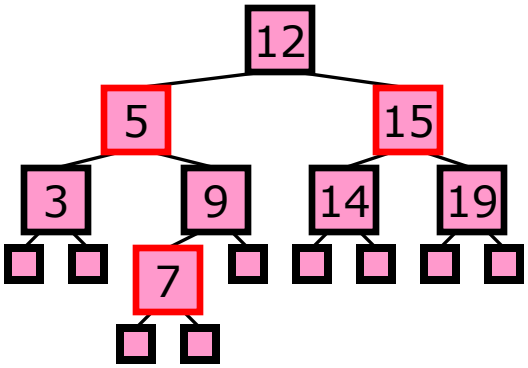
- I have presented red and black **nodes**.



- A node's color might be stored with the pointer to it in the parent node: red and black **pointers**.
  - The root is always black, so it does not matter whether the root's color is stored somewhere.



- Some implementations add **null nodes**.
  - Null nodes are black and have no data.
  - All leaves are null nodes, and all null nodes are leaves.
  - This may allow for slightly faster insert & delete algorithms. It is an example of a **time-space trade-off**.



Other Self-Balancing Search Trees
Red-Black Trees — Usage

Red-Black Trees are a good choice for in-memory Tables, when worst-case performance is important.

- Red-Black Trees, or variations, are the usual implementation for C++ STL sorted Tables: `std::map`, `std::set`, etc.

How do we use Red-Black Trees?

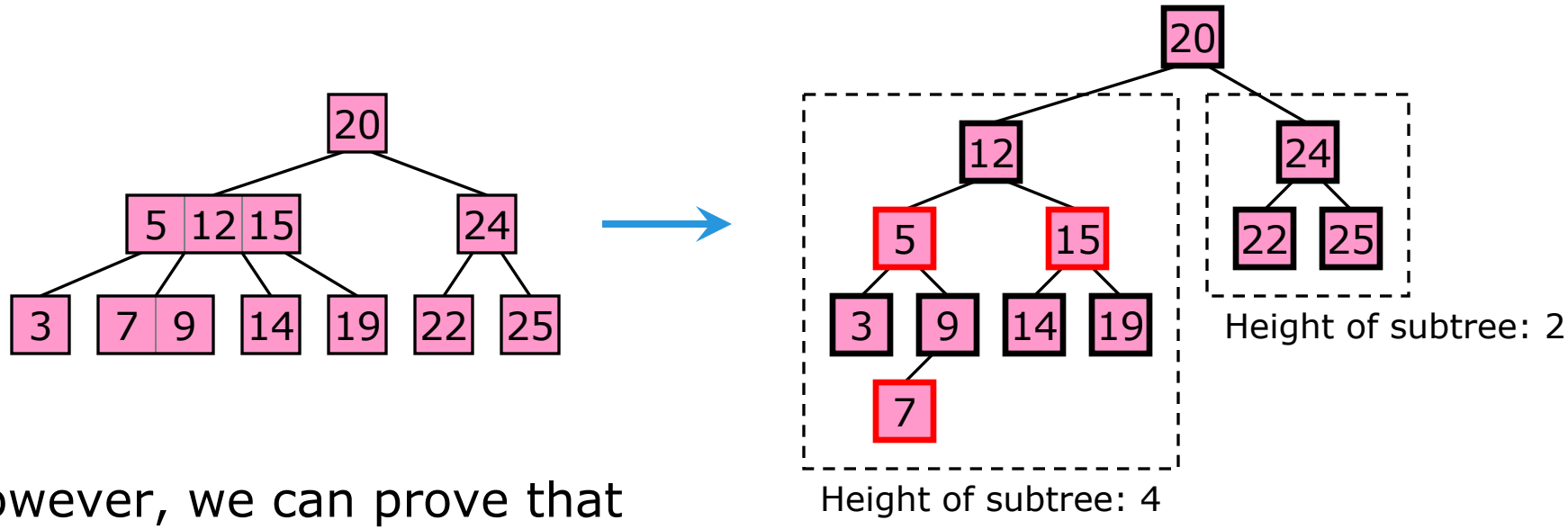
- Traverse & retrieve are exactly as for Binary Search Trees.
- Insert & delete are complicated, and will not be covered. They involve **rotations**.
- The 2-3-4 Tree → Red-Black Tree conversion shows where Red-Black Trees come from; but in practice, we *never* do the conversion.

Why do we use Red-Black Trees?

- Because they tend to be just a little more efficient than 2-3-4 Trees, which are just a little more efficient than 2-3 Trees.
- All three have $\Theta(\log n)$ insert, delete, and retrieve.

Other Self-Balancing Search Trees
Red-Black Trees — Efficiency

A Red-Black Tree is a Binary Search Tree (with extra data in each node), but it is generally *not* strongly balanced.

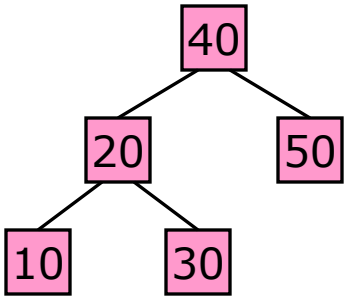


However, we can prove that
a Red-Black Tree with n nodes
cannot have height more than $2 \log_2(n + 1)$.

Thus, the height is $\Theta(\log n)$, which makes the retrieve, insert, and delete operations $\Theta(\log n)$ time.

Other Self-Balancing Search Trees
AVL Trees — Preliminaries

Recall the question we began with: are there fast insert & delete algorithms for a strongly balanced Binary Search Tree that maintain the strongly balanced property?



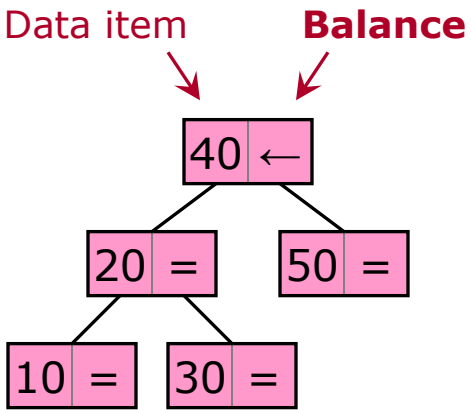
Let's answer: no, such algorithms are impossible—without adding information to the tree.

But we have seen that this fact is not really important. There are structures that are just as good as a strongly balanced Binary Search Trees, and for which algorithms maintaining the structure do exist: 2-3 Trees, 2-3-4 Trees, Red-Black Trees, and others that we have not covered.

However, it turns out that we *can* efficiently maintain the strongly balanced property of a Binary Search Tree if we add a small amount of information to each node. The result is an *AVL Tree.*

Other Self-Balancing Search Trees
AVL Trees — Definition

Recall: a Binary Tree is **strongly balanced**, if, for each node in the tree, its two subtrees have heights differing by at most 1.



An **AVL Tree** is a strongly balanced Binary Search Tree in which each node has an extra piece of information: its **balance**: left high [←], right high [→], or even [=].

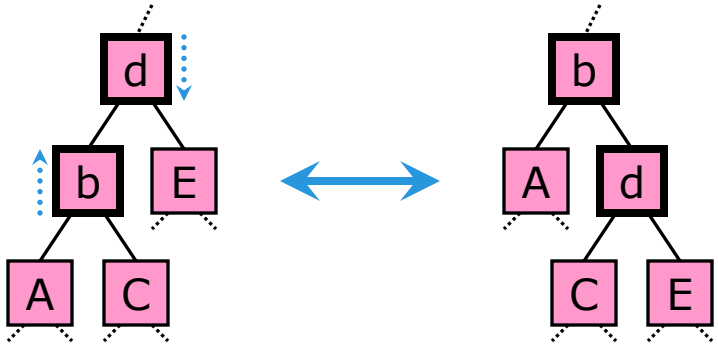
Storing the balance in each node allows for logarithmic-time insert & delete algorithms that maintain the required properties.

AVL Trees were the first kind of self-balancing search trees to be developed [Georgy M. Adelson-Velsky & Yevgeniy M. Landis 1962].

Other Self-Balancing Search Trees
AVL Trees — Rotation

We will not cover all the details of the AVL Tree algorithms. We note that they are based on the **rotation** operation.

- Rotation is pictured below. For nodes labeled A, C, E, the subtrees of which they are the roots are moved along with them.
- We have seen this before, in the "semi-easy case" of the 2-3 Tree delete algorithm.



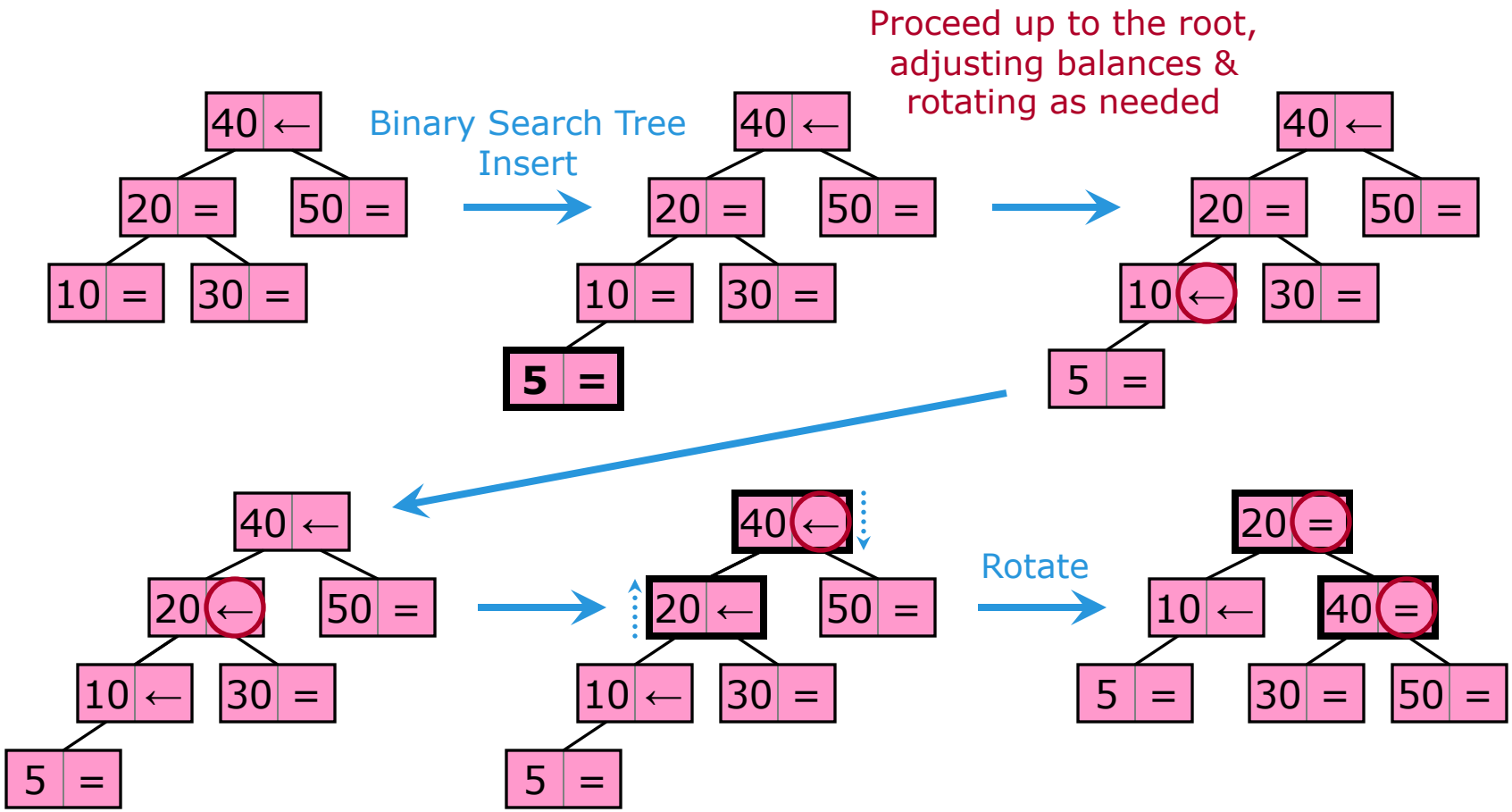
Using rotations, as above, along with a fancier version (the *double rotation*), we can design Θ(log *n*) insert and delete algorithms that maintain the strongly balanced property.

Note that, by the strongly balanced property, retrieve is also Θ(log *n*) for an AVL Tree.

Other Self-Balancing Search Trees
AVL Trees — Example

Example of AVL Tree insert: Do Binary Search Tree insert, then proceed up to the root, adjusting balances & rotating as needed.

- Below we illustrate Insert 5.



Other Self-Balancing Search Trees Wrap-Up [1/2]

All of these self-balancing search trees give Table implementations in which retrieve, insert, and delete by key are $\Theta(\log n)$.

Generally, the **Red-Black Tree** is agreed to have the best *overall* performance, for in-memory datasets with many insert & delete operations, when worst-case performance is important.

A Red-Black Tree, or some variation, is the usual implementation for `std::map`.

Above, the word "overall" is important. For example, an AVL Tree has a faster retrieve operation than a Red-Black Tree, since it tends to have smaller height. (But a sorted array has an even faster retrieve, using Binary Search.)

Other Self-Balancing Search Trees Wrap-Up [2/2]

This completes our coverage of self-balancing search trees—for the moment.

Later, when we look at **external data**, we will cover another kind of self-balancing search tree: **B-Trees**.

- B-Trees are similar to 2-3 Trees and 2-3-4 Trees, but nodes can be larger. For example, a B-Tree node might contain 50 keys.
- Using large nodes can greatly increase efficiency when a Table is stored on a **block-access device**—like a disk.
- We will also look at a B-Tree variant called a **B+ Tree**.