2-3 Trees continued
Other Balanced Search Trees

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, November 23, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

Review
Where Are We? — The Big Problem

Our problem for much of the rest of the semester:

- Store: a collection of data items, all of the same type.
- Operations:
  - Access items [one item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- All this needs to be efficient in both time and space.

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

Generic containers: those in which client code can specify the type of data stored.

Review
Binary Search Trees — Efficiency

	B.S.T. (balanced & average case)	Sorted Array	B.S.T. (worst case)
Retrieve	Logarithmic	Logarithmic	Linear
Insert	Logarithmic	Linear	Linear
Delete	Logarithmic	Linear	Linear

Binary Search Trees have poor worst-case performance.

But they have very good performance:

- On average.
- If balanced.
  - But we do not know an efficient way to make them *stay* balanced.

Can we efficiently keep a Binary Search Tree balanced?

- We will look at this question again later.

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
- (part) 2-3 Trees
- Other balanced search trees
- Hash Tables
- Prefix Trees
- Tables in various languages

Idea #1: Restricted Table

Idea #2: Keep a Tree Balanced

Idea #3: “Magic Functions”

Review
Introduction to Tables [1/2]

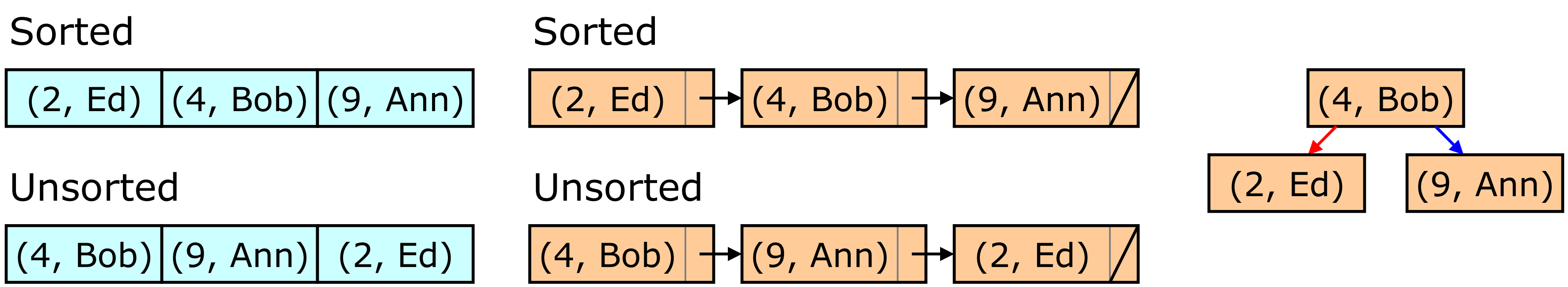
What are possible Table implementations?

- A Sequence holding key-data pairs.
  - Sorted or unsorted.
  - Array-based or Linked-List-based.
- A Binary Search Tree holding key-data pairs.
  - Implemented using a pointer-based Binary Tree.

Table

Key	**Data**
4	Bob
9	Ann
2	Ed

Array Implementations | Linked List Implementations | Binary Search Tree Implementation



Review
Introduction to Tables [2/2]

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	Balanced (how?) Binary Search Tree
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	Logarithmic
Insert	Linear	Constant???	Linear	Constant	Linear	Logarithmic
Delete	Linear	Linear	Linear	Linear	Linear	Logarithmic

Idea #1: Restricted Table
- Perhaps we can do better if we do not implement a Table in its full generality.

Idea #2: Keep a Tree Balanced
- Balanced Binary Search Trees look good, but how to keep them balanced efficiently?

Idea #3: “Magic Functions”
- Use an unsorted array of key-data pairs. Allow array items to be marked as “empty”.
- Have a “magic function” that tells the index of an item.
- Retrieve/insert/delete in constant time? (Actually no, but this is still a worthwhile idea.)

We will look at what results from these ideas:
- From idea #1: Priority Queues
- From idea #2: Balanced search trees (2-3 Trees, Red-Black Trees, B-Trees, etc.)
- From idea #3: Hash Tables

Overview of Advanced Table Implementations

We will cover the following advanced Table implementations.

- Balanced Search Trees
  - Binary Search Trees are hard to keep balanced, so to make things easier we allow more than 2 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 a balanced Binary Tree again:
    - **AVL Tree**
- Alternatively, forget about trees entirely:
  - **Hash Tables**
- Finally, "the Radix Sort of Table implementations":
  - **Prefix Tree**

Review
2-3 Trees — Introduction & Definition [1/2]

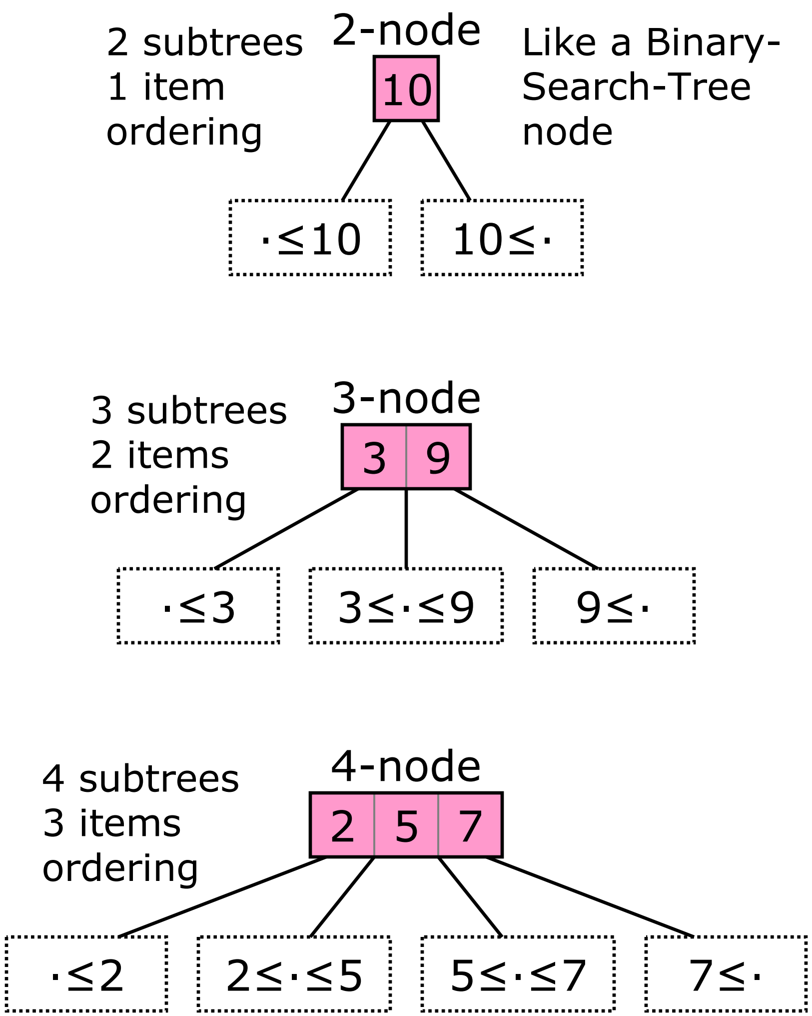
A Binary-Search-Tree style node is a **2-node**.

- This is a node with 2 subtrees and 1 data 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 data items.
- Each of the 2 data items has a value that lies between the values in the corresponding pair of consecutive subtrees.

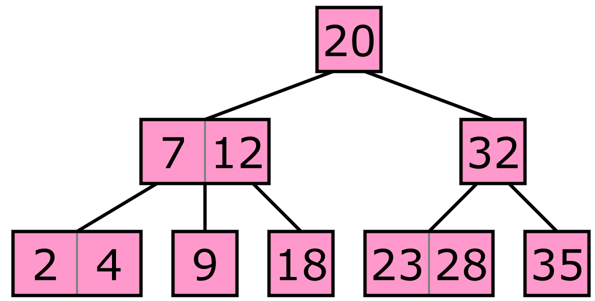
Later, we will look at "2-3-4 trees", which can also have **4-nodes**.



2-3 Trees — Introduction & Definition [2/2]

A **2-3 Search Tree** (generally we just say **2-3 Tree**) is a tree with the following properties.

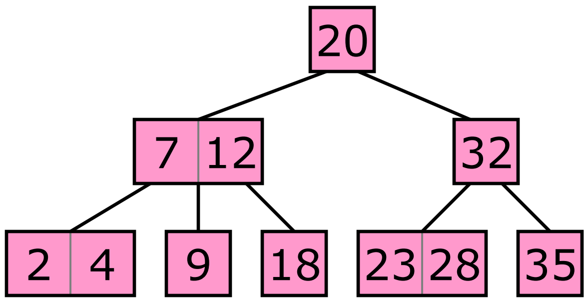
- All nodes contain either 1 or 2 data items.
  - If 2 data items, then the first is ≤ the second.
- All leaves are at the same level.
- All non-leaves are either *2-nodes* or *3-nodes*.
  - They must have the associated order properties.



Review
2-3 Trees — Operations: Traverse & Retrieve

How do we **traverse** a 2-3 Tree?

- We generalize the procedure for doing an **inorder traversal** of a Binary Search Tree.
  - For each leaf, go through the items in it.
  - For each non-leaf 2-node:
    - Traverse subtree 1.
    - Do item.
    - Traverse subtree 2.
  - For each non-leaf 3-node:
    - Traverse subtree 1.
    - Do item 1.
    - Traverse subtree 2.
    - Do item 2.
    - Traverse subtree 3.



- This procedure lists all the items in sorted order.

How do we **retrieve** by key in a 2-3 Tree?

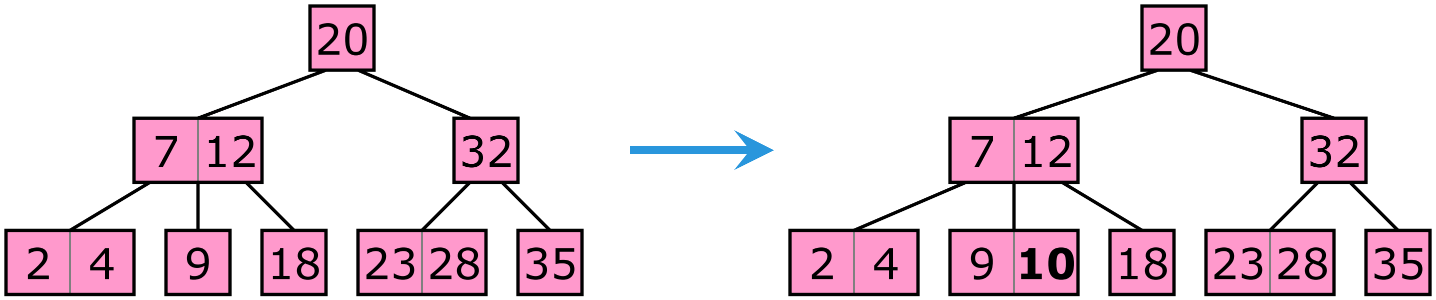
- Start at the root and proceed downward, making comparisons, just as in a Binary Search Tree.
- 3-nodes make the procedure *slightly* more complex.

2-3 Trees — Operations: Insert [1/4]

Ideas in the 2-3 Tree **insert** algorithm:

- Start by adding the item to the appropriate leaf.
- Allow nodes to expand when legal.
- If a node gets too big (3 items), split the subtree rooted at that node and propagate the **middle** item upward.
- If we end up splitting the entire tree, then we create a new root node, and all the leaves advance one level simultaneously.

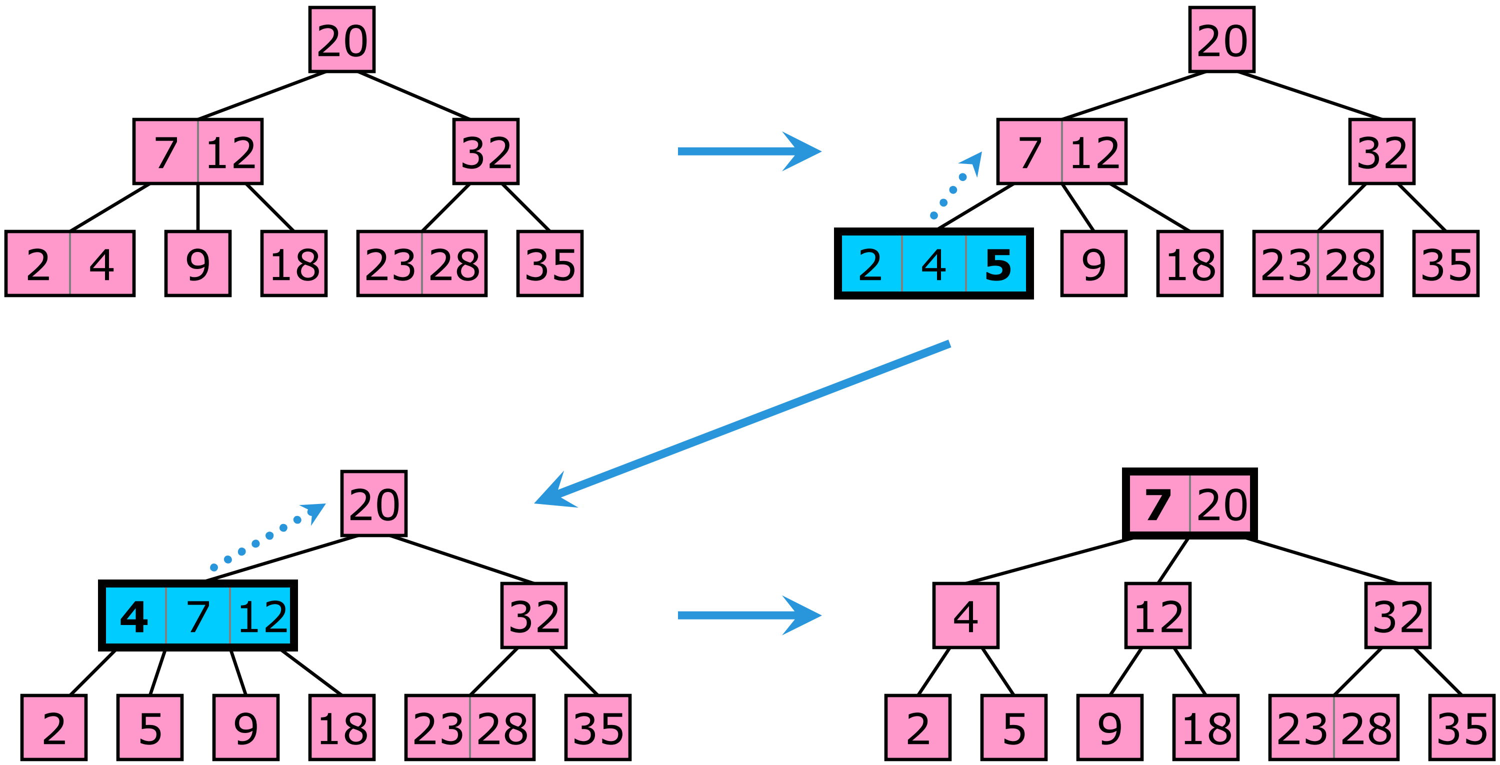
Example 1: Insert 10.



2-3 Trees — Operations: Insert [2/4]

Example 2: Insert 5.

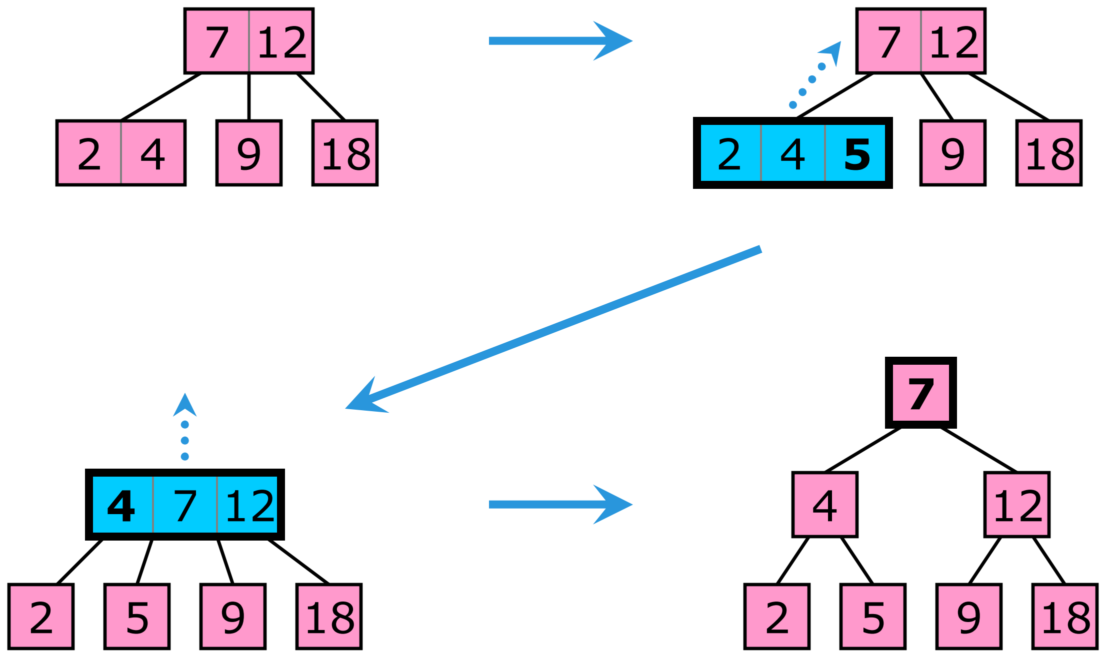
- Over-full nodes are blue.



Review
2-3 Trees — Operations: Insert [3/4]

Example 3: Insert 5.

- Here we see how a 2-3 Tree increases in height.



2-3 Trees — Operations: Insert [4/4]

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

- Find the leaf the new item goes in.
  - Note: In the process of finding this leaf, you may determine that the given key is already in the tree. If you do, act accordingly.
- Add the item to the proper node.
- If the node is overfull, then split it (dragging subtrees along, if necessary), and move the middle item up:
  - If there is no parent, then make a new root. Done.
  - Otherwise, add the moved-up item to the parent node. To add the item to the parent, do a recursive call to the insertion procedure.

Review
2-3 Trees — Operations: Delete [1/8]

Deleting from a 2-3 Tree is similar to inserting.

- We will use the recursive-thinking idea to avoid describing every detail of the process.
- We try to delete from a leaf. If it does not work, rearrange.
- If that does not work, bring an item from the parent down. This is deleting from the parent. Recurse (or reduce the height and we are done).
- As with inserting, we start at a leaf and work our way up.

2-3 Trees — Operations: Delete [2/8]

Observation

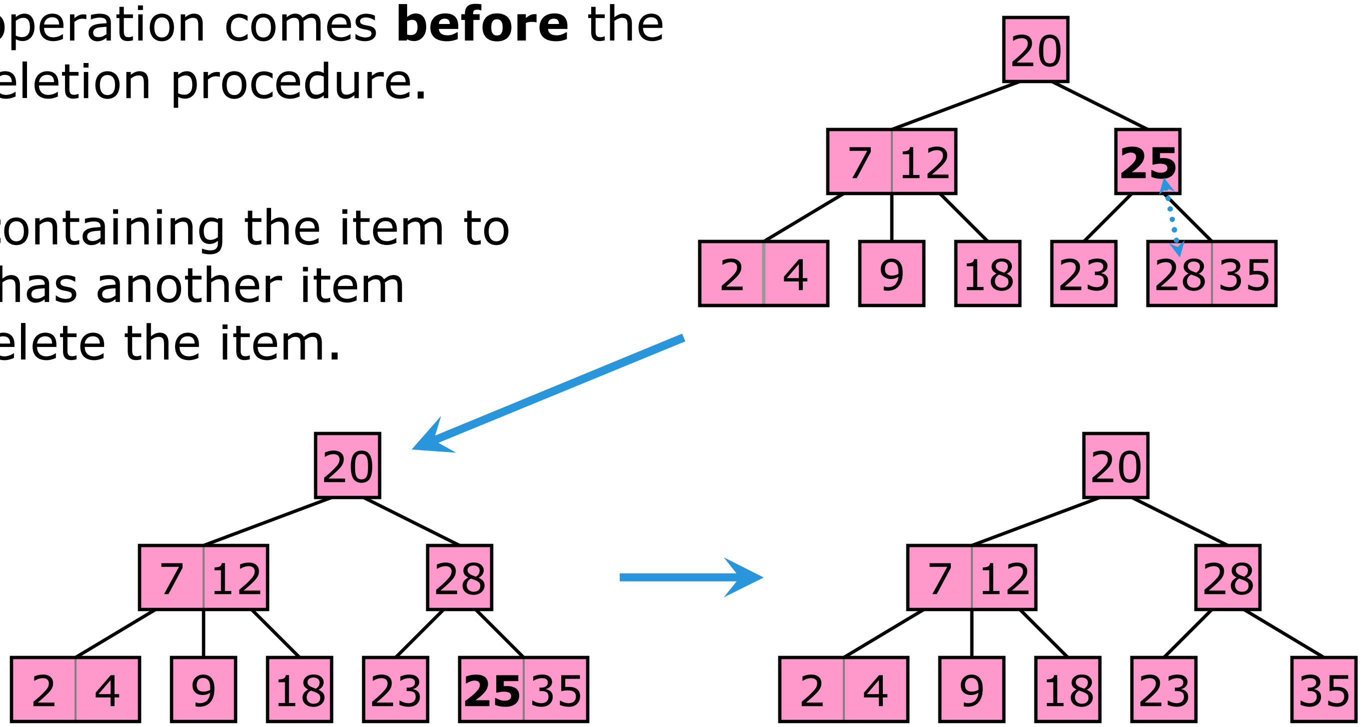
- We can always start our deletion at a leaf.
- If the item to be deleted is not in a leaf, swap it with its "inorder" successor.
  - It must have one. (Why?)
- This swap operation comes **before** the recursive deletion procedure.

Easy Case

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

Example

- Delete 25.

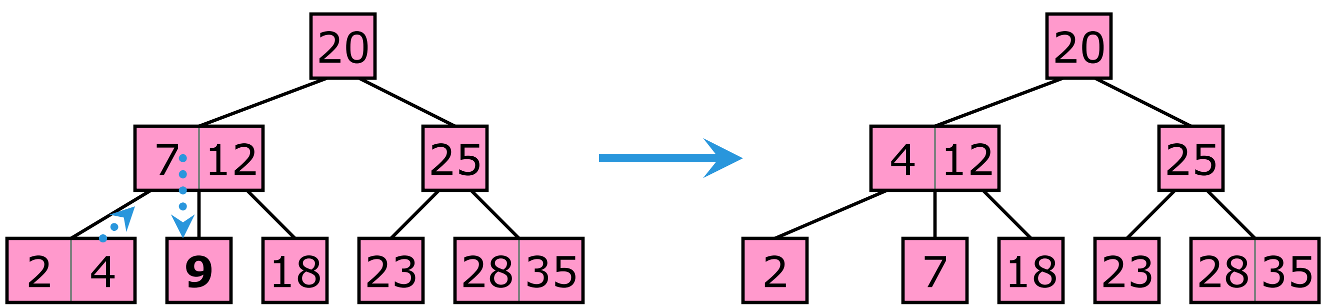


2-3 Trees — Operations: Delete [3/8]

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 rearrange using the parent.

Example: Delete 9.

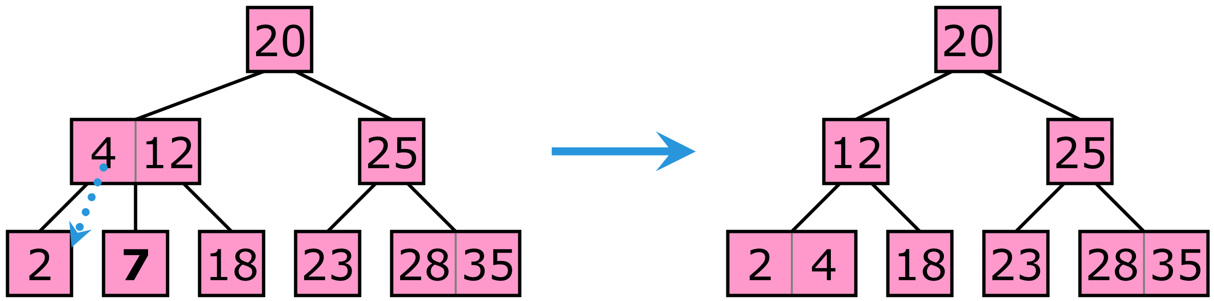


Review
2-3 Trees — Operations: Delete [4/8]

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 must bring down an item from the parent and place it in a nearby sibling node.
- We need to join nodes/subtrees to make the invariants work.

Example: Delete 7.



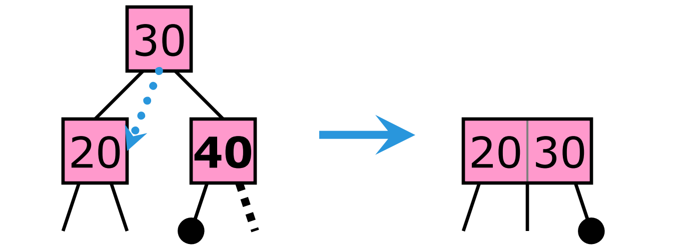
In the above example, recursively "delete" 4 from the tree consisting of the first two levels. Since 4's node has another item in it, this is the easy case; we simply get rid of 4 (and then put it in the node containing 2).

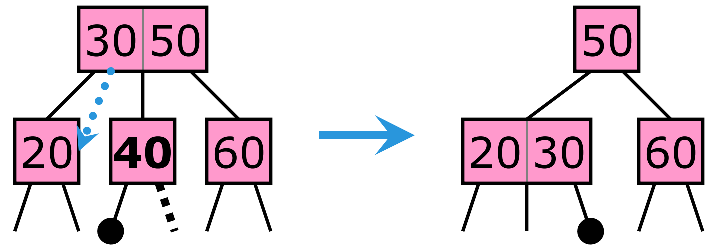
Review
2-3 Trees — Operations: Delete [5/8]

If we do a recursive delete above the leaf level, where do “orphaned” subtrees go?

Consider two Hard Case examples.

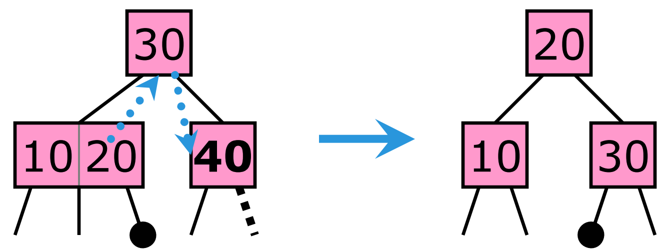
- We delete 40. Why? Because one of its subtrees is going away. What do we do with the other subtree?
- Answer: Make it a subtree of the item we bring down.





Consider a Semi-Easy Case example.

- Again, we delete 40. One of its subtrees is going away. 30 is coming down to replace it. 20 is going up. What do we do with the right-subtree of 20?
- Answer: Make it the left subtree of 30.



Idea: There is always exactly one spot available for an orphaned subtree. Put it in that spot.

2-3 Trees — Operations: Delete [6/8]

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

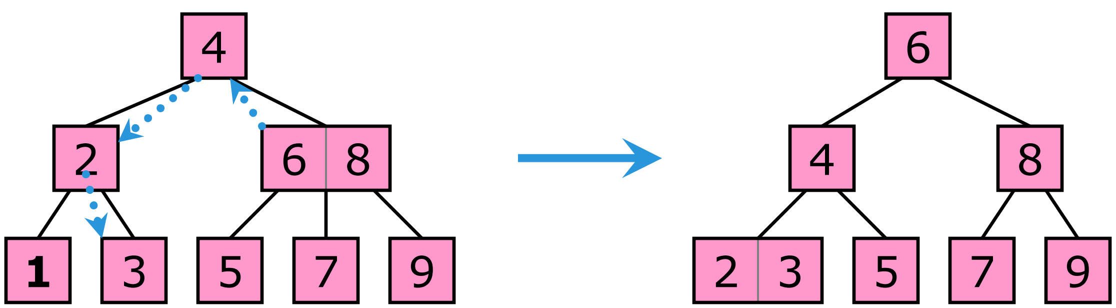
- Find the node holding the given key.
  - Note: In the process of this search, you may determine that the given key is not in the tree. If you do, act accordingly.
- If the above node is not a leaf, then swap its item with its successor in the traversal ordering. Continue with the deletion procedure: 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. Eliminate item from parent using a recursive call to the deletion procedure (dragging subtrees along).

2-3 Trees — Operations: Delete [7/8]

A few more examples.

Example: Delete 1.

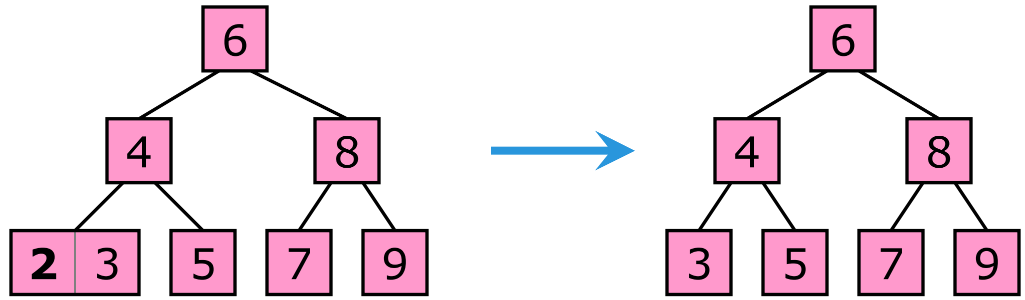
- 1 is "Hard Case", so we bring down the parent (recursively "delete" 2) and join it with 3 in a single node.
- 2 is "Semi-Easy Case", so rotate (6 to 4 to 2).
- The 5 is orphaned. We make it the right child of 4.



2-3 Trees — Operations: Delete [8/8]

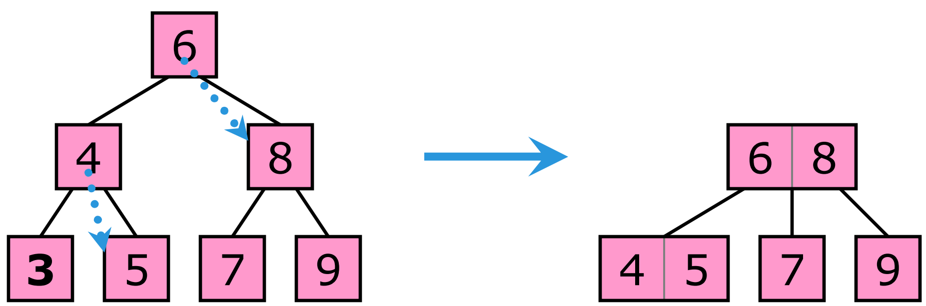
Example: Delete 2.

- This is “Easy Case”.



Example: Delete 3.

- This 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. We reduce the height of the tree.



2-3 Trees continued
Efficiency

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

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

This is the **first** time we have seen a delete-by-key that handles *any* given 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.

However, there are better bases.

- Not necessarily a **lot** better, but better.

Other Balanced Search Trees Better Than a 2-3 Tree?

Again, the Table operations retrieve, insert, and delete are all $O(\log n)$ for a 2-3 Tree implementation.

We do not know any structure in which all operations are $O(\log n)$ [worst case], and at least one is faster.

- Of course, we **can** make some operations better & some worse. For example, for an unsorted Linked List implementation, Table insert is $O(1)$, while Table retrieve & delete are $O(n)$.

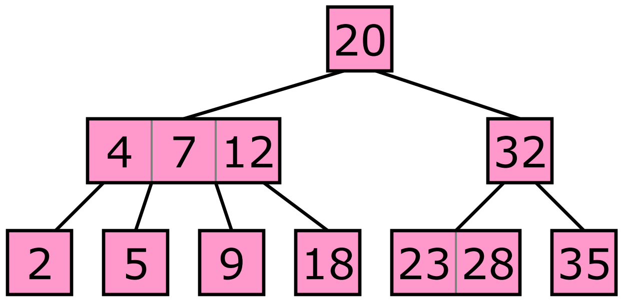
However, we can make everything a **little** faster than a 2-3 Tree, although still $O(\log n)$.

We do this with other kinds of balanced search trees.

- These are all similar to a 2-3 Tree.
- Thus, we will not look at them in great detail.
- See the text for details.

Other Balanced Search Trees
2-3-4 Trees

In a **2-3-4 Tree**, we also allow 4-nodes.



The insert and delete algorithms are not terribly different from those of a 2-3 Tree.

- They are a little more complex.
- And they tend to be a little faster.

Why not also allow 5-nodes (a "2-3-4-5 Tree")?

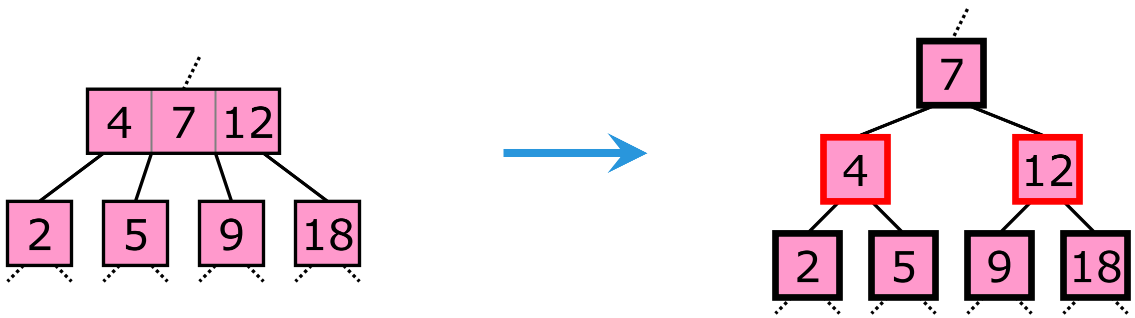
- Because the algorithms tend to be a little slower.

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

It turns out that 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 representation we will discuss is called a **Red-Black Tree**.

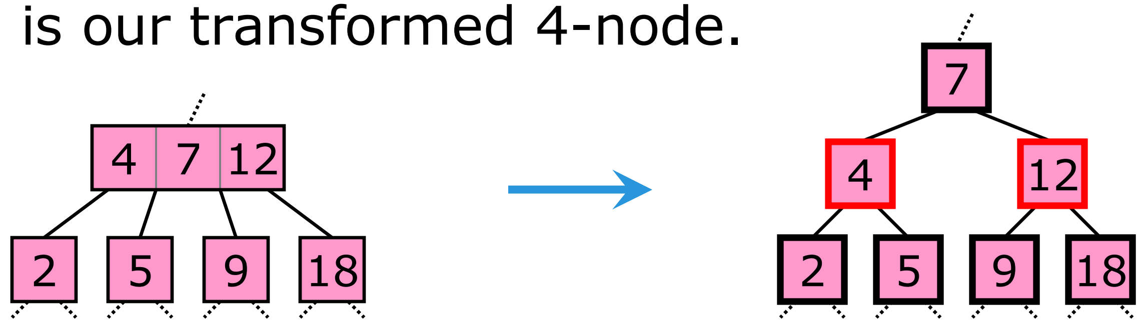
Consider the 4-node below. We can represent this part of the 2-3-4 Tree using only 2-nodes if we add two new nodes (shown in red).



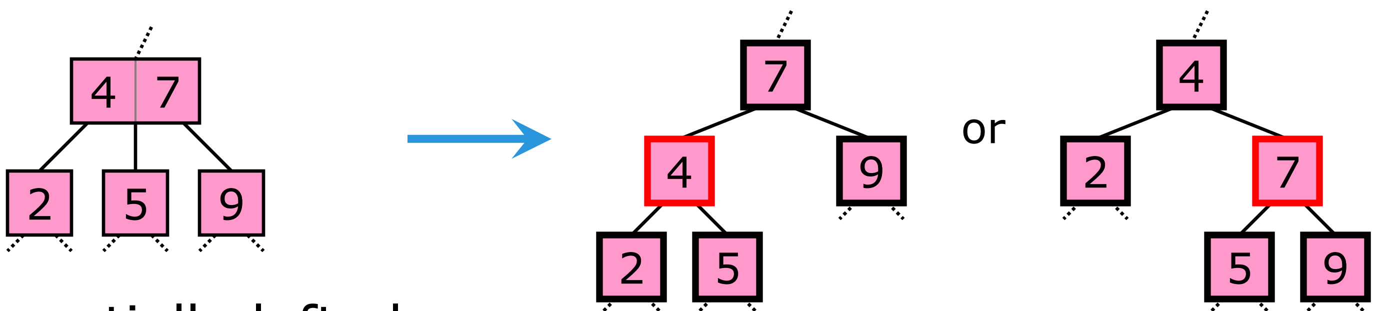
Note that the ordering property of the 2-3-4 Tree translates into the ordering property of a Binary Search Tree.

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

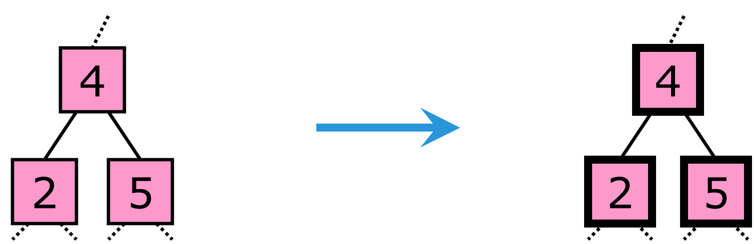
Here again is our transformed 4-node.



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



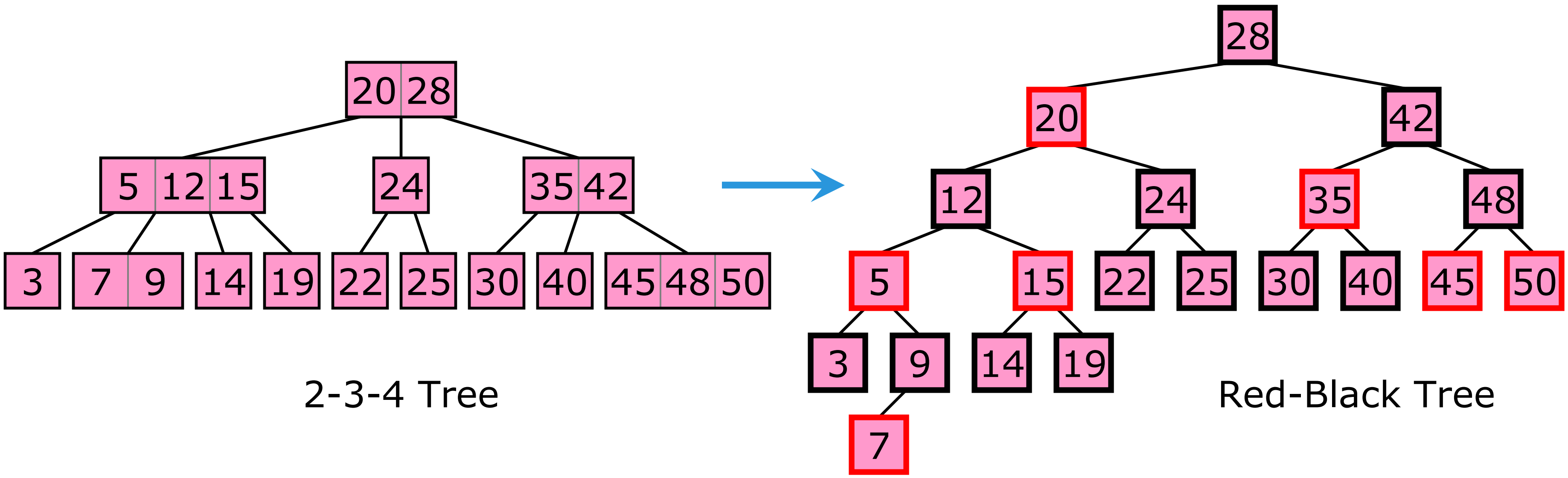
2-nodes are essentially left alone.



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

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

- A R.B.T. is a Binary Search Tree in which each node is "**red**" or "**black**".
- Think of **black** nodes as representing 2-3-4 Tree nodes.
- Think of **red** nodes as being the extra ones required to make a Binary Tree out of the 2-3-4 Tree.



2-3-4 Tree

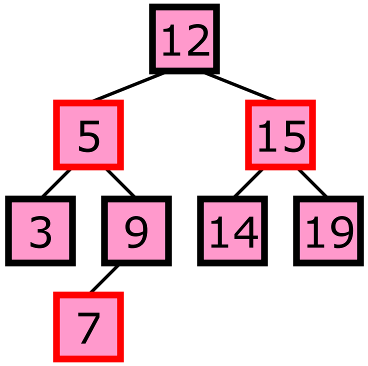
Red-Black Tree

- It is no longer true that every leaf is at the same level. However, given a node, every path from it down to a leaf goes through the same number of **black** nodes.

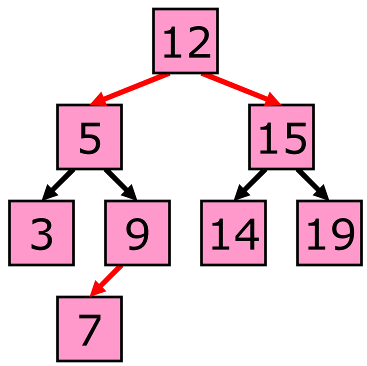
Other Balanced Search Trees
Red-Black Trees — Variations

Implementations of Red-Black Trees vary.

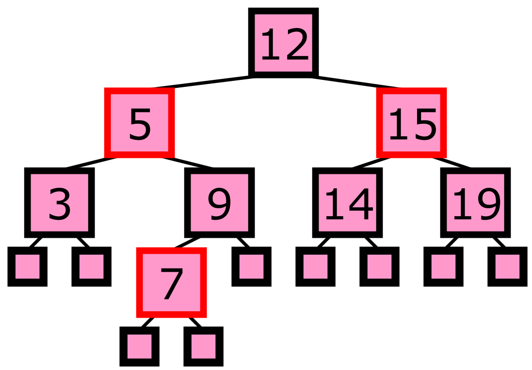
- I have presented them as having red and black **nodes**.



- The text talks about red and black **pointers**.
  - Note that the root is always black, so it does not matter whether the root's color is stored somewhere.



- Some (most?) versions add "**null nodes**" at the bottom.
  - Null nodes are black and have no data.
  - All leaves are null nodes, and all null nodes are leaves.



Other Balanced Search Trees
Red-Black Trees — Usage

How do we use Red-Black Trees?

- Retrieve and traverse are exactly the same as for Binary Search Trees. Just ignore the color.
- Insert and delete algorithms are complicated (and will not be covered). They are based on *rotations*, which we will see when we cover AVL Trees (shortly).

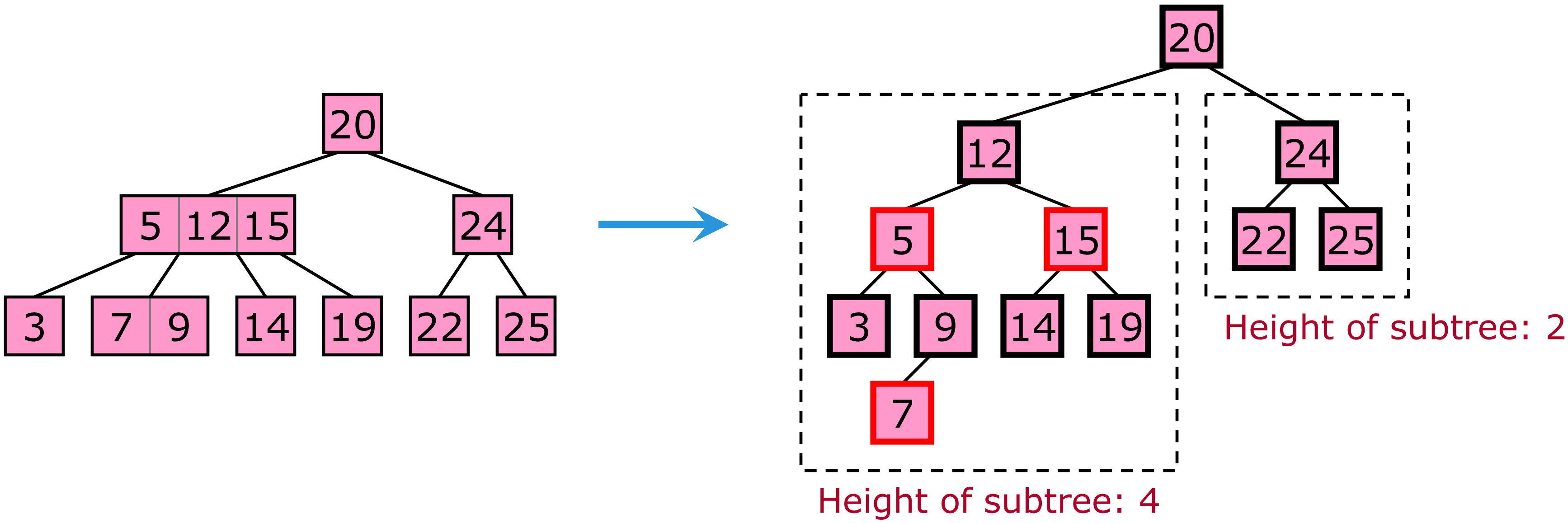
Why do we use Red-Black Trees?

- 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 $O(\log n)$ insert, delete, and retrieve.

Red-Black Trees are the most common basis for implementations of C++ STL Tables (**`std::set`**, **`std::map`**, etc.).

Other Balanced Search Trees
Red-Black Trees — Notes [1/2]

A Red-Black Tree is not necessarily a balanced Binary Tree, as we defined "balanced" earlier.



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

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

Other Balanced Search Trees
Red-Black Trees — Notes [2/2]

In practice, we *never* do the 2-3-4 Tree to Red-Black Tree conversion. Rather, we implement only a Red-Black Tree.

- The conversion was illustrated here in order to explain where Red-Black Trees come from and how they work.

If you need an efficient balanced search tree for in-memory data, use a Red-Black Tree.

- The insert & delete algorithms get rather complex. Look up the details.

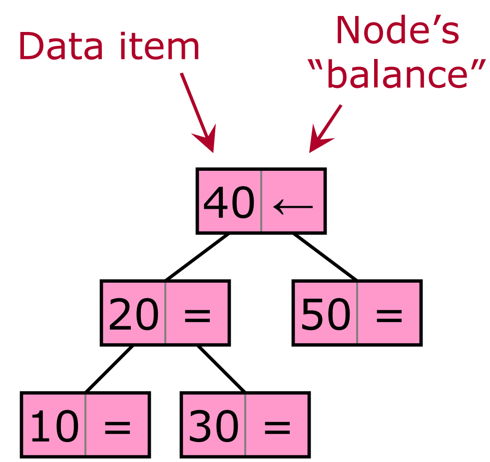
Other Balanced Search Trees
AVL Trees — Definition

The first kind of self-balancing search tree was the “AVL Tree”.

- AVL trees are named after the authors of a 1962 paper describing them: Georgy Maximovich **A**delson-**V**elsky and Yevgeniy Mikhailovich **L**andis.
- These days, AVL Trees are mostly a historical curiosity.

An **AVL Tree** is a balanced (in our original, strict sense) Binary Search Tree in which each node has an extra piece of data: its “balance”: left high [←], right high [→], or even [=].

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

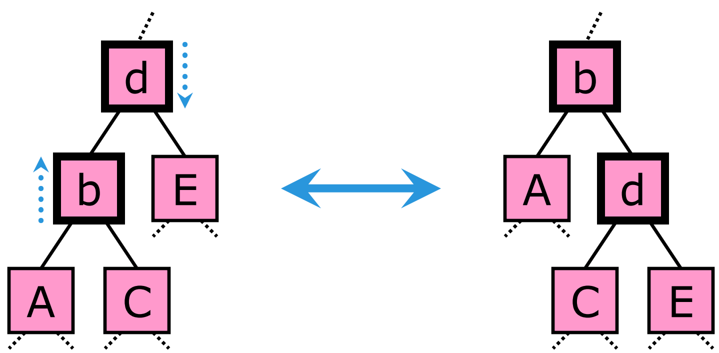


A-V & L discovered logarithmic-time algorithms to do insert and delete while maintaining the balanced property.

Other Balanced Search Trees
AVL Trees — Rotation

We will not cover all of the details of the AVL Tree algorithms.

- We note that they rest on an operation known as **rotation**.
- Rotation is pictured below. For nodes labeled A, C, E, the subtrees of which they are the roots are moved along with them.
- Note that we have seen something (roughly) like this before, in the "semi-easy case" of 2-3 Tree deletion.

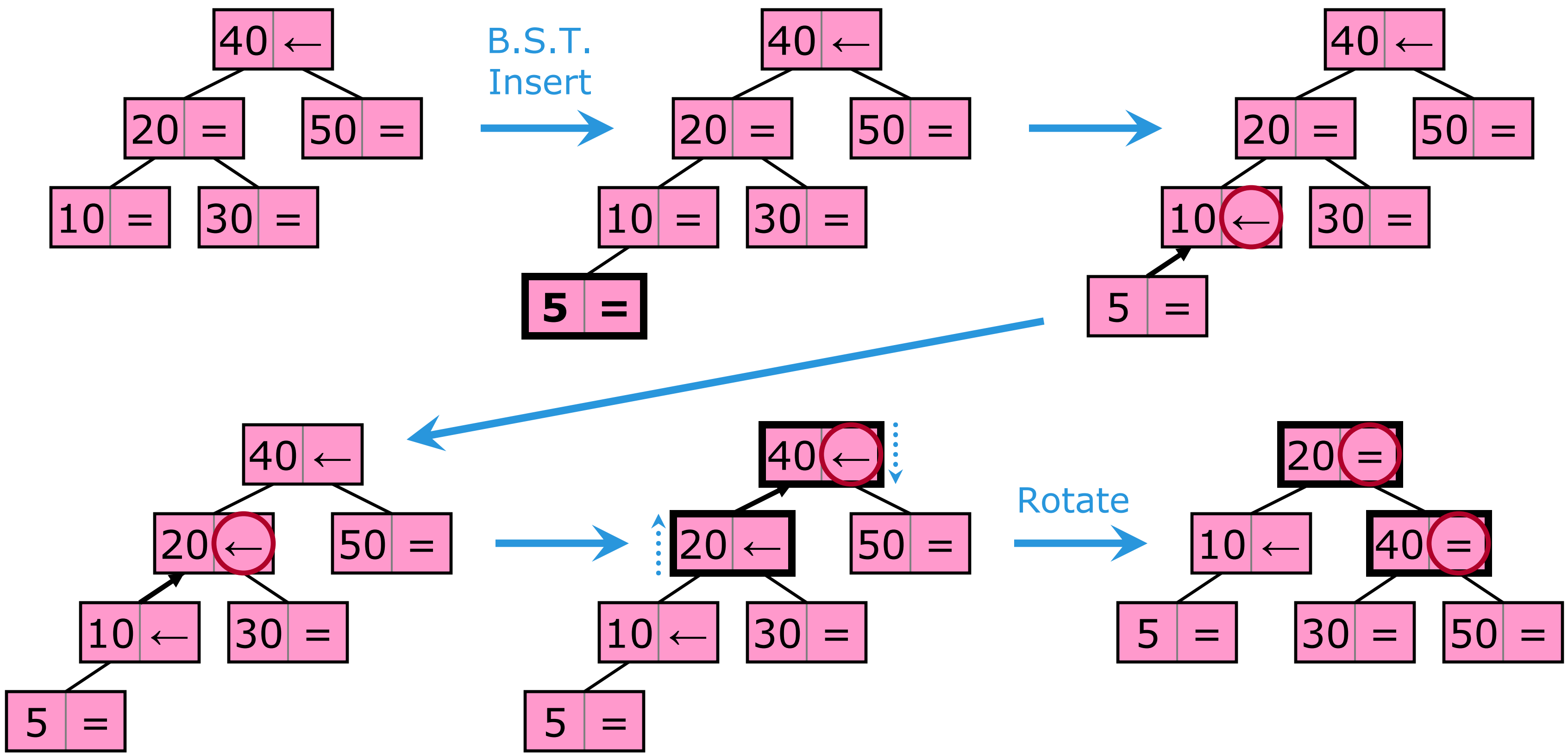


When we allow rotations, we can insert or delete using at most $O(\log n)$ operations, while maintaining the balanced property.

- Thus, insert and delete (and, by the balanced property, retrieve) are
$O(\log n)$ operations for an AVL Tree.

Quick example of AVL Tree insert: Do Binary Search Tree insert, then proceed up to the root, adjusting "balances" and, if needed, rotating.

- Below we illustrate Insert 5.



Other Balanced Search Trees
Wrap-Up

All balanced search trees offer an implementation of the Table ADT in which the insert, delete, and retrieve operations are $O(\log n)$.

Generally, the Red-Black Tree is agreed to have best **overall** performance.

- It is the one that tends to be used to implement things like **`std::map`**.
- The word "overall" is important. For example, an AVL Tree has a faster retrieve operation than a Red-Black Tree.
  - But a sorted array has an even faster retrieve; no one uses AVL Trees.

Implementation details may be changed due to various trade-off's.

- Space vs. time, etc.