2-3 Trees

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, November 13, 2024

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Tables & Priority Queues

Topics

- ✓ Introduction to Tables
- ✓ Priority Queues
- ✓ Binary Heap Algorithms
- ✓ Heaps & Priority Queues in the C++ STL
- 2-3 Trees
- Other self-balancing search trees
- Hash Tables
- Prefix Trees
- Tables in the C++ STL & Elsewhere

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.

Note the three primary single-item operations: **retrieve**, **insert**, **delete**. We will see these over & over again.

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

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

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

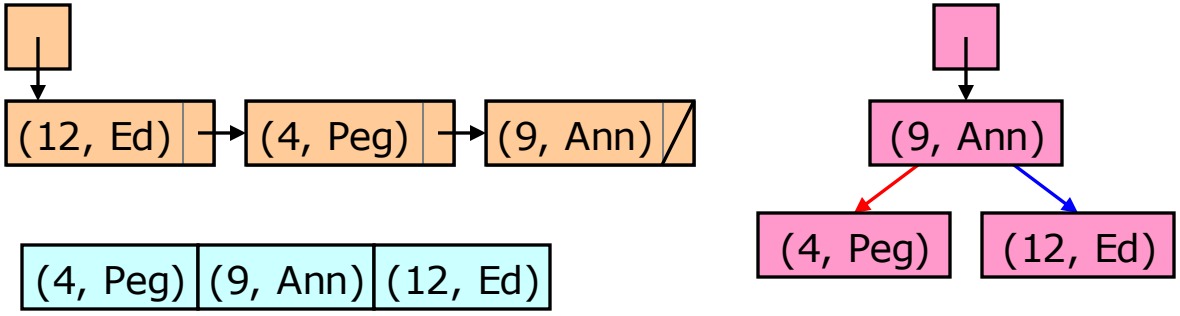
Three single-item operations: retrieve, insert, delete by key.

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

Table

Key	Value
12	Ed
4	Peg
9	Ann

Inefficient Implementations



Three ideas for improving efficiency:

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

Unit Overview
Tables & Priority Queues

Topics

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

  Idea #1: Restricted Table

- 2-3 Trees
- Other self-balancing search trees

  Idea #2: Keep a tree balanced

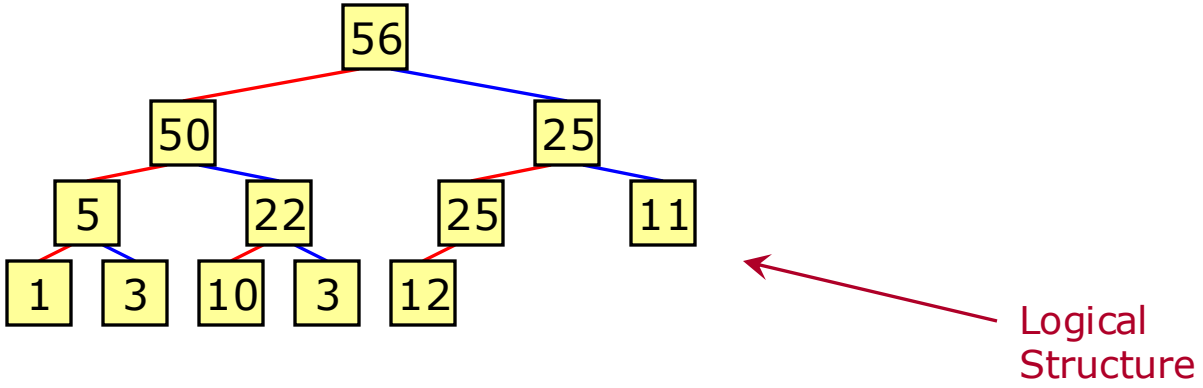
- Hash Tables

  Idea #3: Magic functions

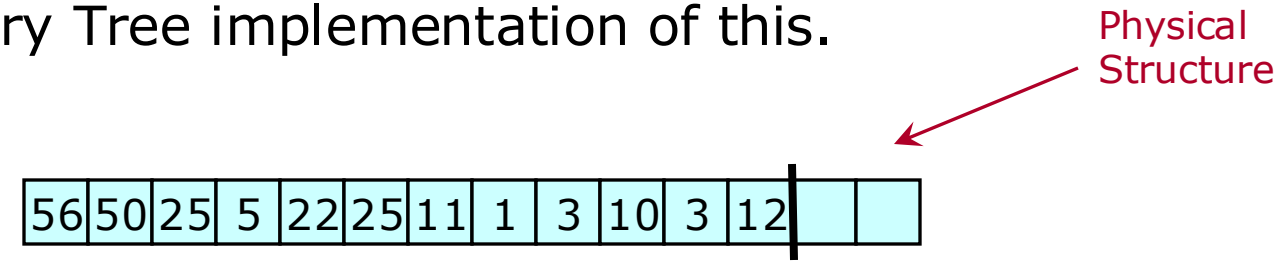
- Prefix Trees ← A special-purpose implementation: "the Radix Sort of Table implementations"
- Tables in the C++ STL & Elsewhere

Binary Heap Algorithms [1/4]

A **Binary Heap** (or just **Heap**) is a complete Binary Tree with one data item—which includes a **key**—in each node, where no node has a key that is less than the key in either of its children.



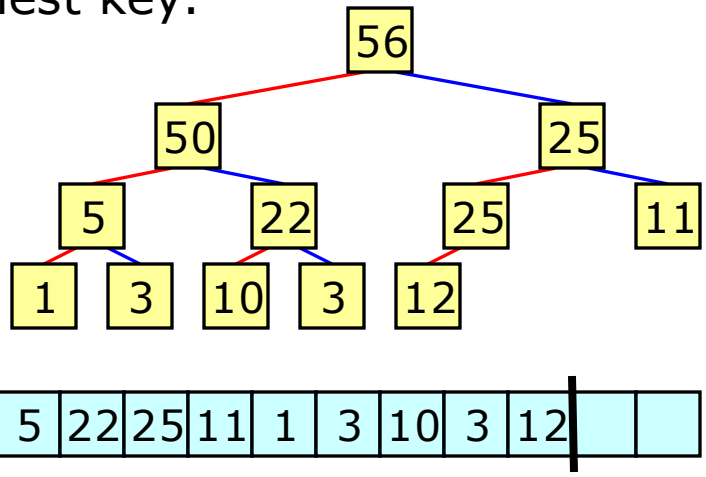
In practice, we use "Heap" to refer to the array-based complete Binary Tree implementation of this.



Review
Binary Heap Algorithms [2/4]

A Heap is a good basis for an implementation of a **Priority Queue**.

- Like Table, but retrieve/delete only highest key.
- Insert any key-value pair.



Algorithms for three primary operations

- **getFront**
  - Get the root node.
  - Constant time.
- **insert**
  - Add new node to end of Heap. Sift-up last item.
  - Logarithmic time if no reallocation required.
  - Linear time otherwise. However, in practice, a Heap often does not manage its own memory, which makes the operation logarithmic time.
- **delete**
  - Swap first & last items. Reduce size of Heap. Sift-down new root item.
  - Logarithmic time.

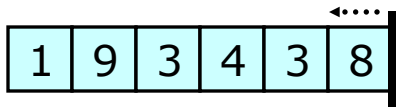
Faster than linear time!

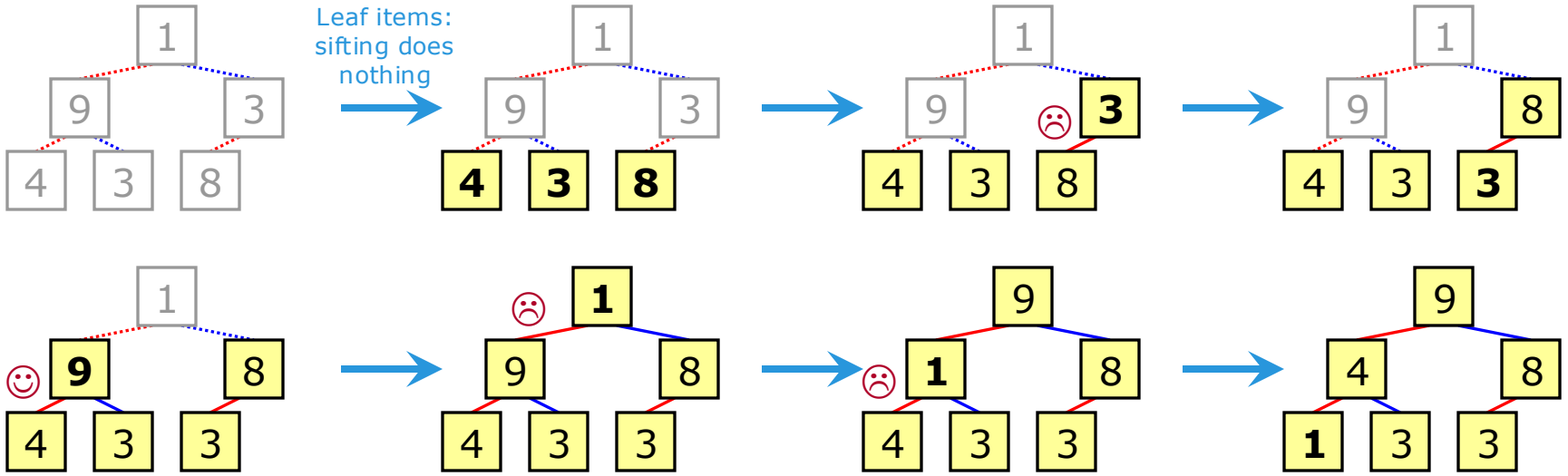
Review
Binary Heap Algorithms [3/4]

To turn a range into a Heap, we could do $n-1$ Heap inserts. Each insert op is $\Theta(\log n)$; making a Heap in this way is $\Theta(n \log n)$.

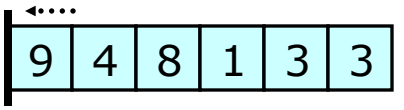
However, there is a faster way.

- Go through the items in reverse order.
- Sift-down each item through its descendants.





This Make-Heap algorithm is *linear time*!



Binary Heap Algorithms [4/4]

Heap Sort is a fast comparison sort that uses Heap algorithms.

- We can think of it as using a Priority Queue, except that the algorithm is in-place, with no separate data structure used.
- Procedure. Make a Heap, then delete all items, using the Heap delete procedure that places the deleted item in the top spot.
- The *Make-Heap* operation is Θ(*n*). Then we do *n Heap delete* operations, each of which is Θ(log *n*). Total: Θ(*n* log *n*).

See `heap_sort.cpp` *for a Heap Sort implementation. This uses the Heap algorithms in* `heap_algs.hpp`*.*

Recall *Introsort*, a Quicksort variant that switches to Heap Sort if the recursion gets too deep. We can write this now.

See `introsort.cpp` *for an Introsort implementation. This also uses the Heap algorithms in* `heap_algs.hpp`*.*

Heaps & PQs in the C++ STL

All the Heap algorithms we wrote have counterparts (with different names) in the STL. Each takes an optional comparison.

The STL has a Priority Queue: `std::priority_queue` (`<queue>`).

This is another *container adapter*: wrapper around a container.

And once again, you get to pick what that container is.

Note the name of the header!

```
std::priority_queue<T, container<T>>
```

container defaults to `std::vector`.

```
std::priority_queue<T>
    // = std::priority_queue<T, std::vector<T>>
```

A comparison can be specified; *see the slides from last time*.

Overview of Advanced Table Implementations

This ends our coverage of Idea #1: restricted Tables.

Next, actual Tables, allowing retrieve & delete for arbitrary keys.

We cover the following advanced Table implementations.

- Self-balancing search trees
  - To make things easier, allow more children (?):
    - **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 cover other self-balancing search trees: B-Trees, B+ Trees.

Idea #3: Magic functions

2-3 Trees

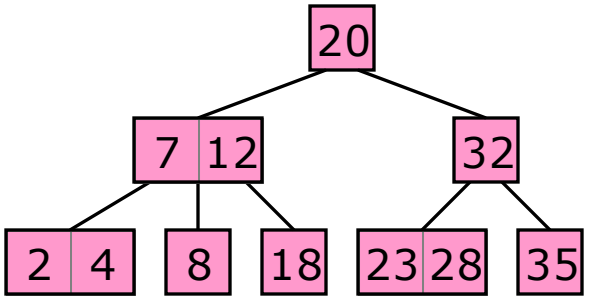
2-3 Trees
Self-Balancing Search Trees [1/3]

Now we look at the second of the three ideas: keeping a Binary Search Tree strongly balanced.

But let's not insist on strongly balanced Binary Search Trees. Rather, we want trees that, like these, have small height, and do not require visiting many nodes to find a given key. We also want fast insert/delete algorithms that *keep* the height small.

These structures are called **self-balancing search trees**.

- There are many kinds. All are similar to Binary Search Trees. They may or may not be strongly balanced. But many are not actually Binary Trees.
- All the self-balancing search trees we will cover have logarithmic-time retrieve, insert, and delete algorithms that maintain the required structure.

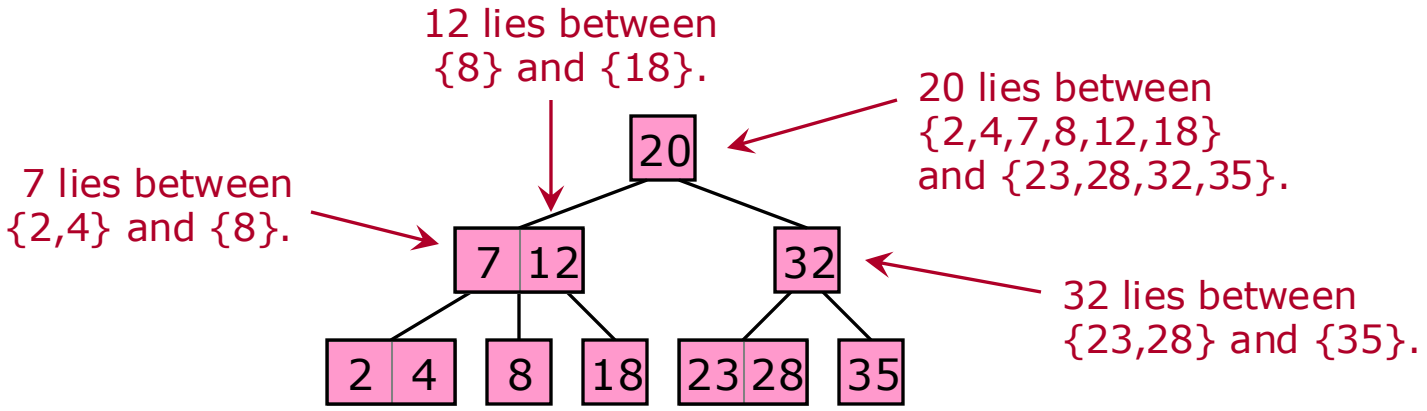


Self-Balancing Search Trees [2/3]

Small height can be easier to maintain if we allow a node to have more than 2 children.

Q. If we do this, how do we maintain the *search tree* idea?

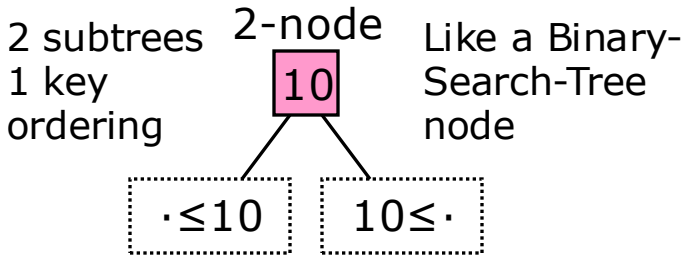
A. We generalize the order property of a Binary Search Tree: For each pair of consecutive subtrees, a node has one key lying between the keys in these subtrees.



2-3 Trees
Self-Balancing Search Trees [3/3]

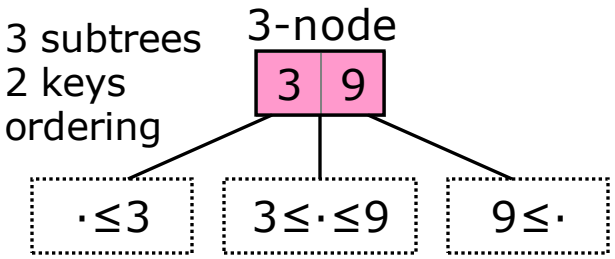
A Binary-Search-Tree style node is a **2-node**.

- A node with 2 subtrees & 1 key.
- The key lies between the keys in the two subtrees.

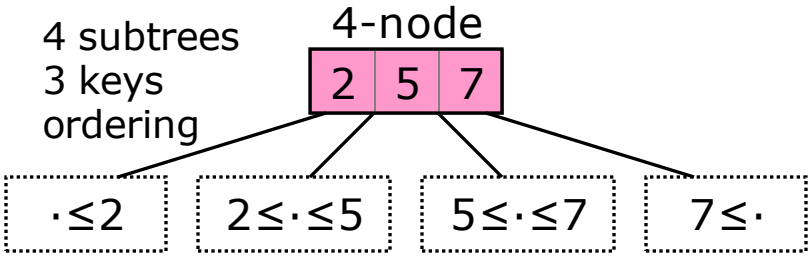


In a *2-3 Tree* we also allow a node to be a **3-node**.

- A node with 3 subtrees & 2 keys.
- Each of the 2 keys lies between the keys in the corresponding pair of consecutive subtrees.



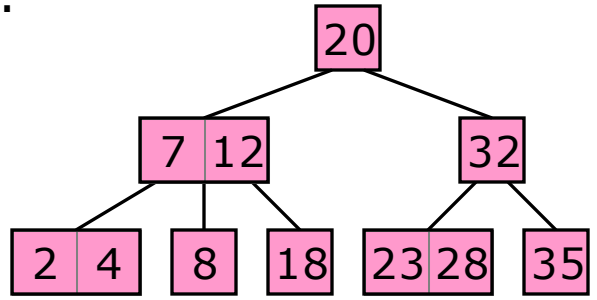
Later, we will look at *2-3-4 Trees*, which can also have **4-nodes**.



2-3 Trees
Definition [1/3]

A **2-3 Search Tree** (usually 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.



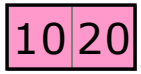
We will look at a number of kinds of self-balancing search trees. Most of these will be closely related to the 2-3 Tree.

We will cover algorithms for 2-3 Trees in detail. For other kinds of self-balancing search trees, we might say something along the lines of, "It works just like a 2-3 Tree, except …"

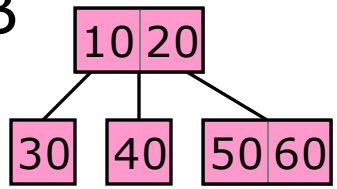
2-3 Trees
Definition [2/3] (Try It!)

Which of the following are 2-3 Trees?

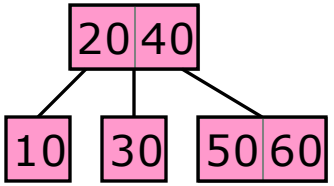
A



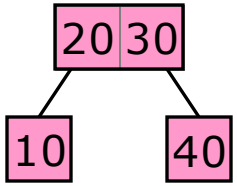
B



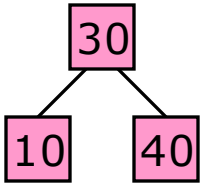
C



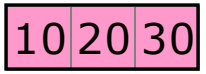
D



E



F



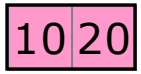
Answers on next slide.

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

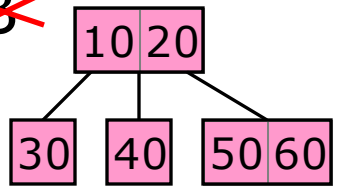
Which of the following are 2-3 Trees?

Answers

✓ A

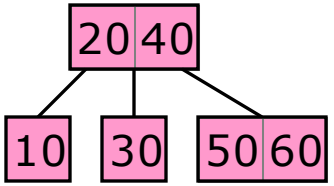


~~B~~

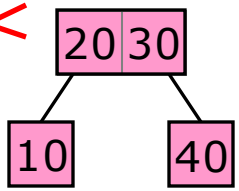


The order property does not hold (30 > 10).

✓ C

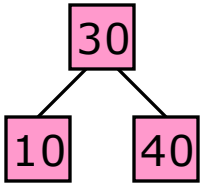


~~D~~

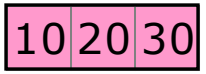


Each node must either have its full complement of children, or else be a leaf.

✓ E



~~F~~

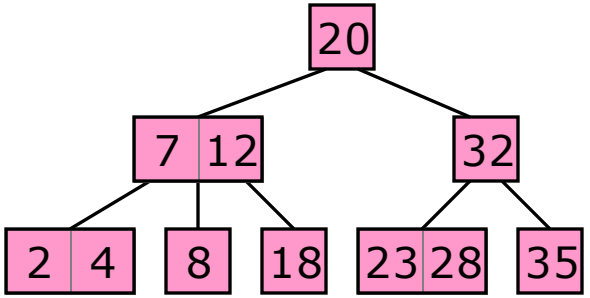


This will turn out to be a valid *2-3-4 Tree*, but it is not a 2-3 Tree. A 2-3 Tree has at most 2 items in each node.

2-3 Trees
Traverse

To **traverse** a 2-3 Tree, generalize the **inorder traversal** of a Binary Search Tree.

- For each leaf, go through the items in it, in order.
- For each non-leaf 2-node:
  - Traverse subtree 1.
  - Visit the item.
  - Traverse subtree 2.
- For each non-leaf 3-node:
  - Traverse subtree 1.
  - Visit item 1.
  - Traverse subtree 2.
  - Visit item 2.
  - Traverse subtree 3.



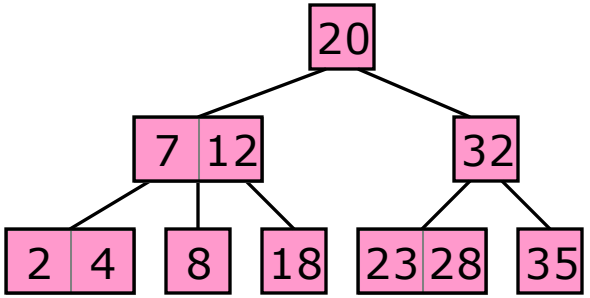
This procedure visits all the items in sorted order. A 2-3 Tree is another *sorted container*.

2-3 Trees
Single-Item Operations

The single-item operations are the usual three: retrieve, insert, and delete. All are done by key.

To **retrieve** by key in a 2-3 Tree, **search**: start at the root and proceed downward, making comparisons, just as when doing a search in a Binary Search Tree.

3-nodes make the procedure just a bit more complicated.



How do we **insert** & **delete** by key in a 2-3 Tree?

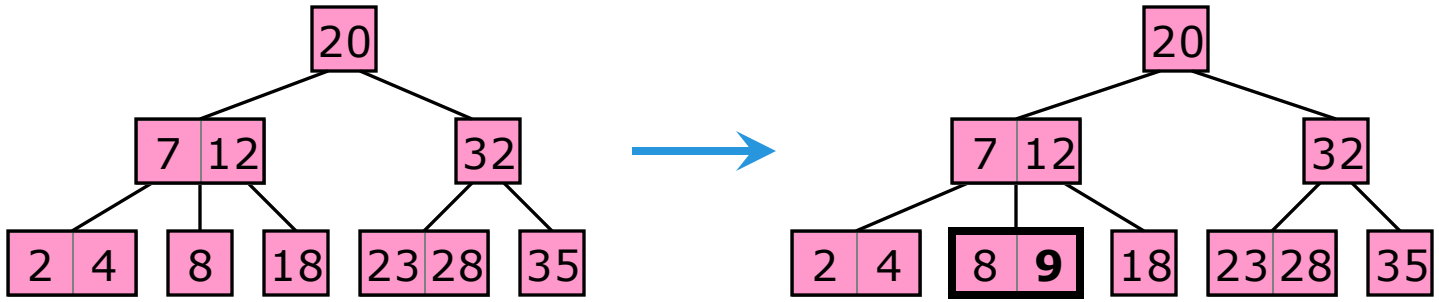
- These are trickier problems.
- It turns out that both have efficient—Θ(log *n*)—algorithms that maintain the properties of the tree. That is why we like 2-3 Trees.

2-3 Trees
Insert Algorithm [1/6]

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

- Search: find the proper leaf. Add the item to that leaf.
- Allow nodes to expand when legal.
- If a node becomes **over-full** (3 items), then split the subtree rooted at that node and propagate the *middle* item upward.
- If we split the entire tree, then create a new root node—which effectively advances all other nodes down one level simultaneously.

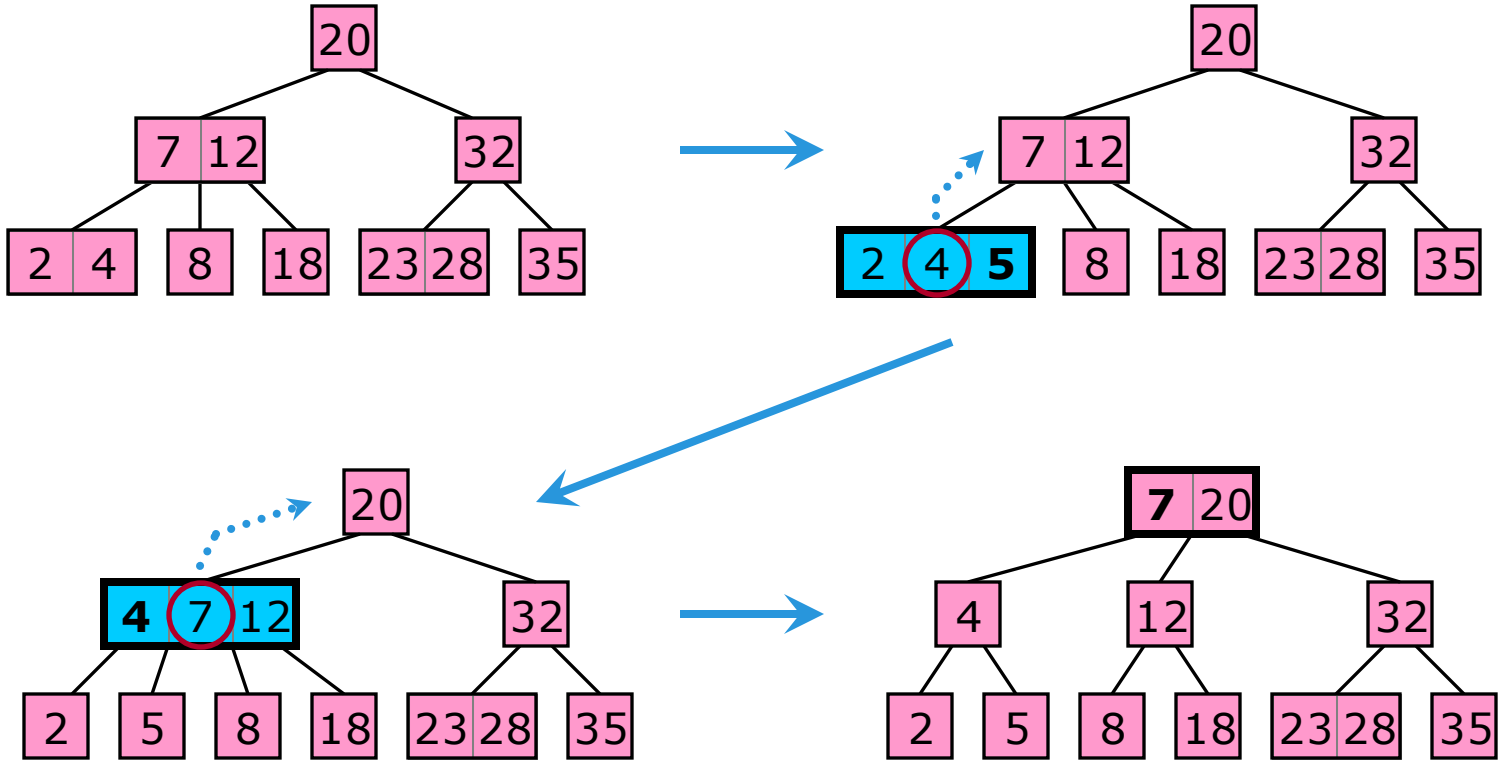
Example 1. Insert 9.



Insert Algorithm [2/6]

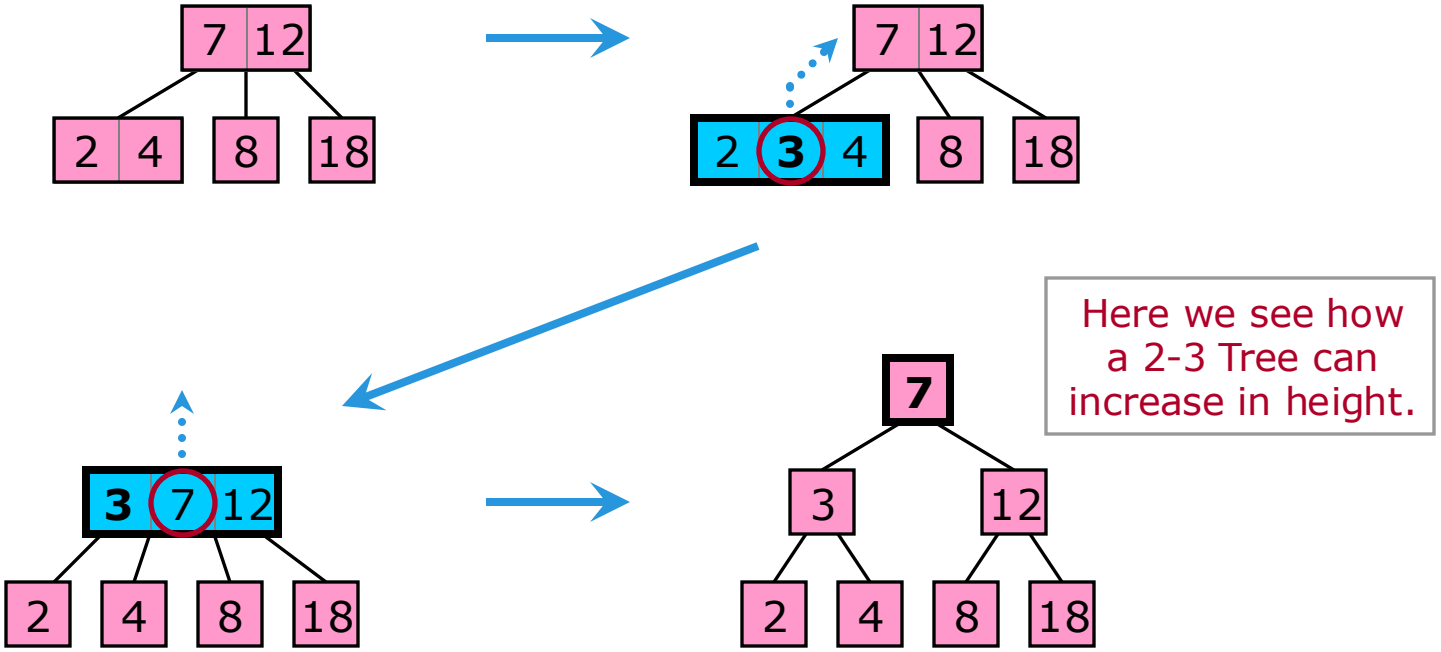
Example 2. Insert 5.

Over-full nodes are **blue**.



Example 3. Insert 3.

Over-full nodes are **blue**.

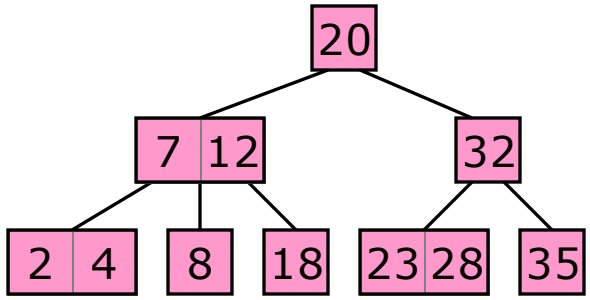


2-3 Tree **Insert** Algorithm (outline)

- Search: find the leaf that the new item goes in.
  - In the process of finding this leaf, you may determine that the given key is already in the tree. If so, then act accordingly.
- Add the item to the proper node.
- If the node is over-full, 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. Recursively apply the insertion procedure one level up.

2-3 Trees
Insert Algorithm [5/6] (Try It!)

Do insert 21 in the following 2-3 Tree. Draw the resulting Tree.

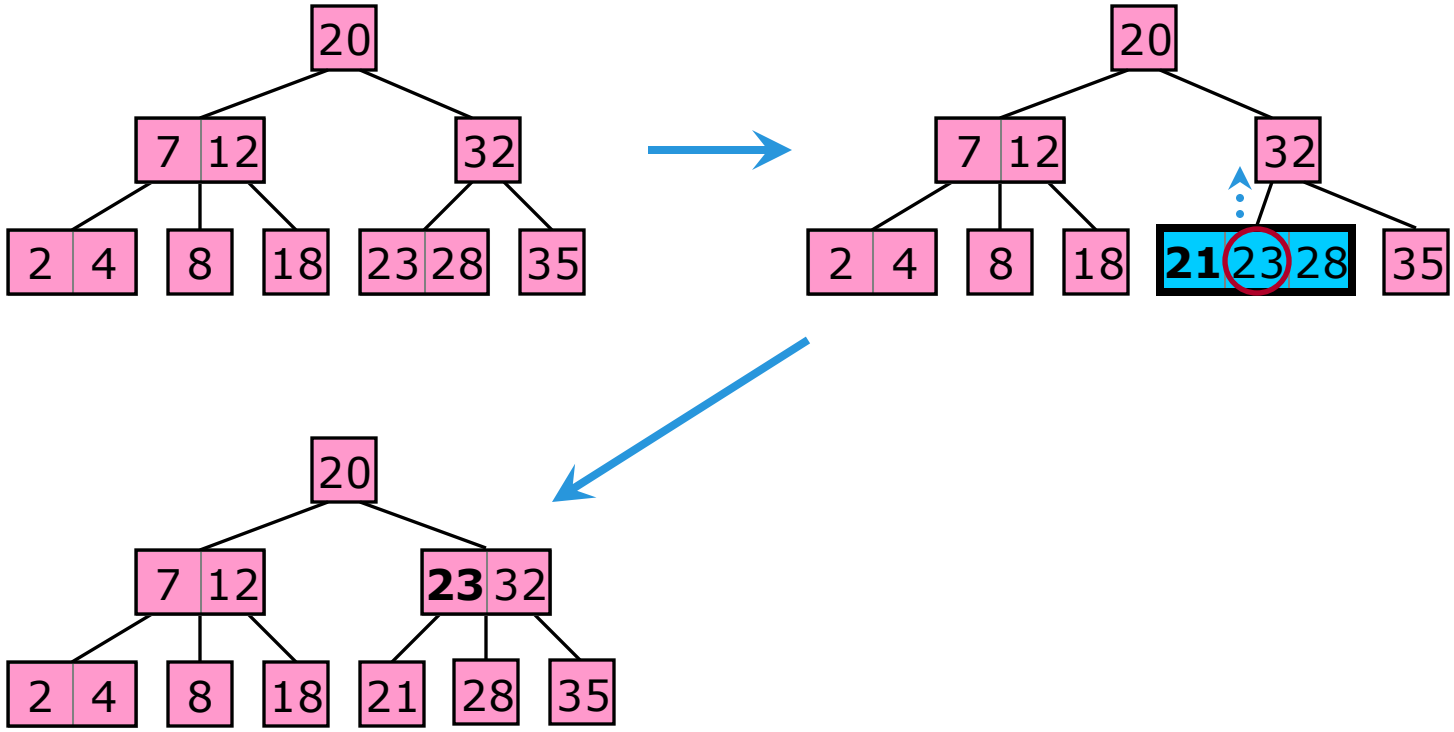


Answer on next slide.

Insert Algorithm [6/6] (Try It!)

Do insert 21 in the following 2-3 Tree. Draw the resulting Tree.

Answer



2-3 Trees
TO BE CONTINUED …

2-3 Trees will be continued next time.