

## 2-3 Trees

---

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, November 15, 2023

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

[ggchappell@alaska.edu](mailto:ggchappell@alaska.edu)

© 2005–2023 Glenn G. Chappell

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

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.

# Review

## Introduction to Tables

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

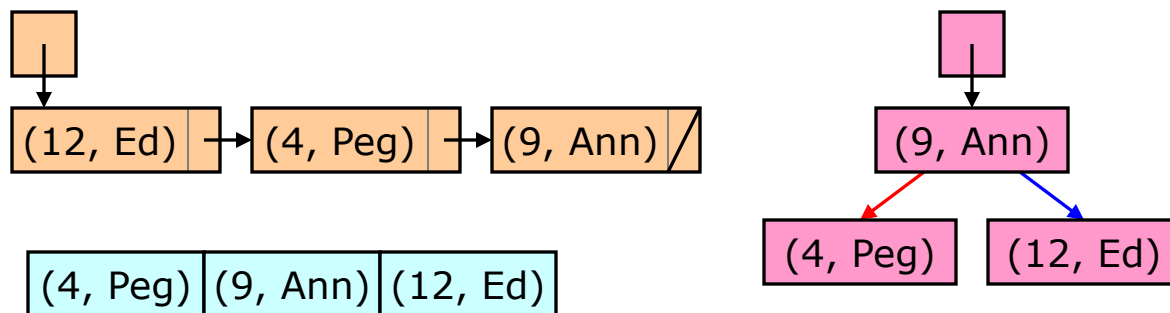
Three single-item operations: retrieve, insert, delete by key.

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

Table

Key	Value
12	Ed
4	Peg
9	Ann

Inefficient Implementations



Three ideas for improving efficiency:

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

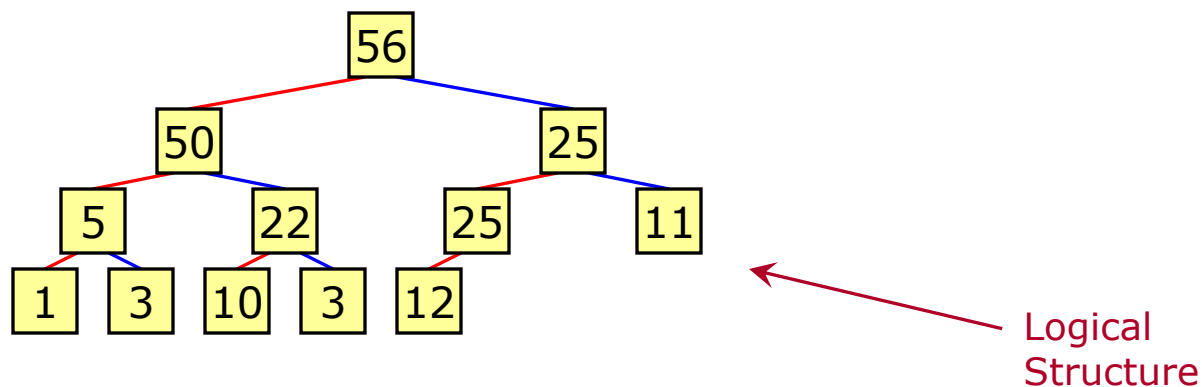
# Unit Overview

## Tables & Priority Queues

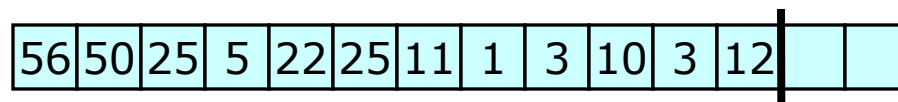
### Topics

- ✓ ■ Introduction to Tables ← Several lousy implementations
  - ✓ ■ Priority Queues
  - ✓ ■ Binary Heap Algorithms
  - ✓ ■ Heaps & Priority Queues in the C++ STL
  - 2-3 Trees
  - Other self-balancing search trees
  - Hash Tables
  - Prefix Trees ← A special-purpose implementation: “the Radix Sort of Table implementations”
  - Tables in the C++ STL & Elsewhere
- Idea #1: Restricted Table
- Idea #2: Keep a tree balanced
- Idea #3: Magic functions

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.



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.

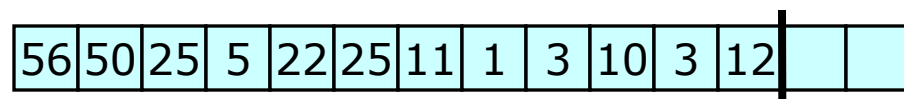
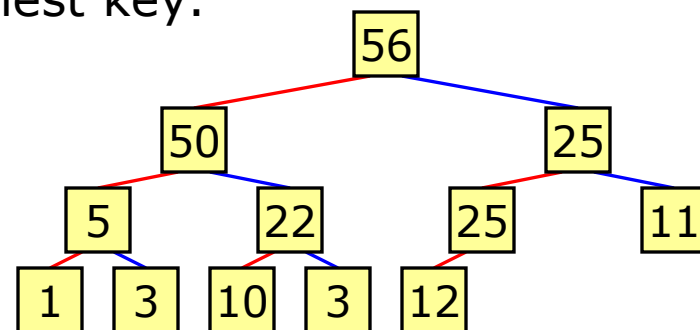
- insert**

- 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!



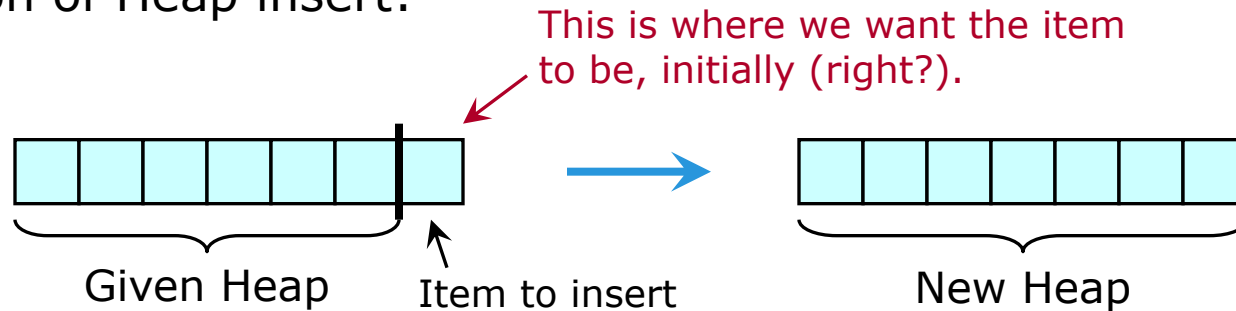


## Review

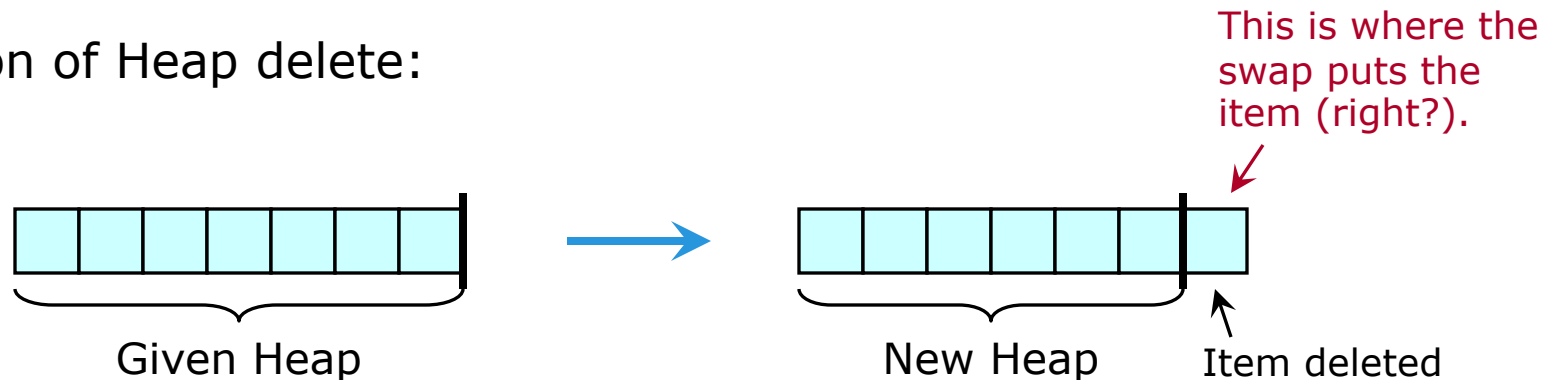
### Binary Heap Algorithms [3/6]

Heap insert and delete are usually given a random-access range. The item to insert or delete is the last item; the rest is a Heap.

- Action of Heap insert:



- Action of Heap delete:



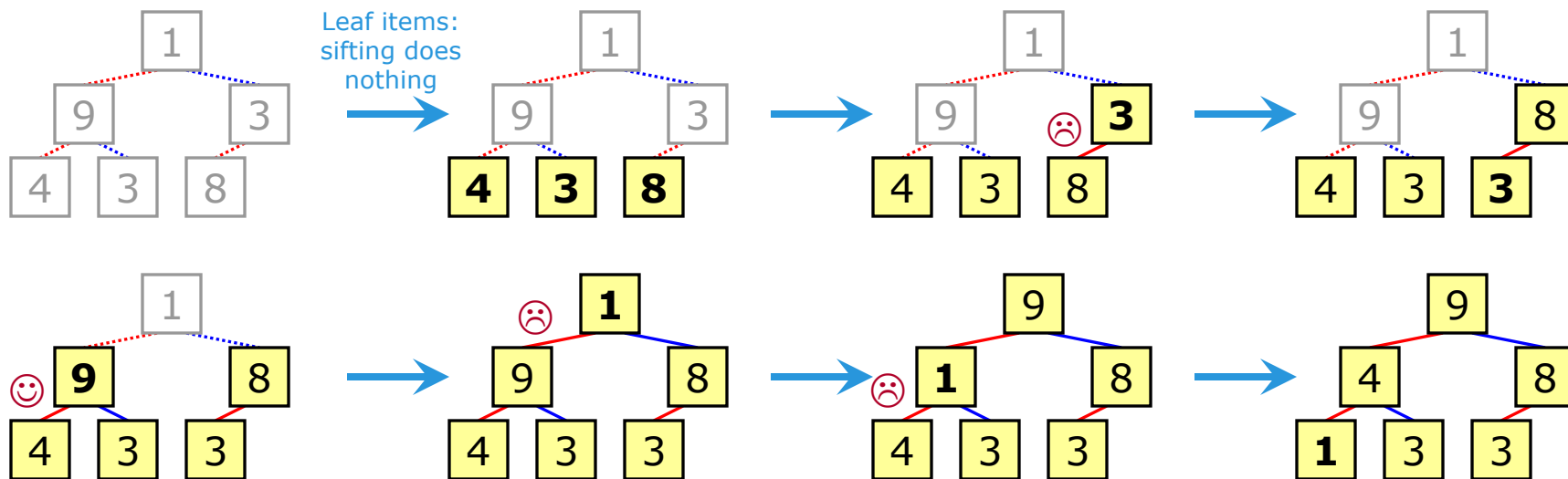
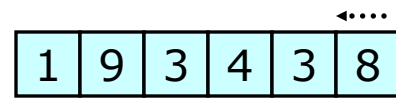
Note that Heap algorithms can do *all* modifications using *swap*. This usually allows for both speed and (exception) safety.

## Review

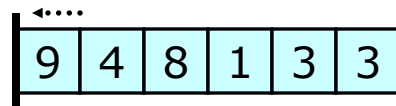
## Binary Heap Algorithms [4/6]

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  $\Theta(n \log n)$ .

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



This Make-Heap algorithm is *linear time*!



**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  $\Theta(n)$ . Then we do  $n$  *Heap delete* operations, each of which is  $\Theta(\log n)$ . Total:  $\Theta(n \log n)$ .

*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`.*

# Review

## Binary Heap Algorithms [6/6]

---

### Efficiency 😊

- Heap Sort is  $\Theta(n \log n)$ .

### Requirements on Data 😞

- Heap Sort requires random-access data.

### Space Usage 😊

- Heap Sort is in-place.

### Stability 😞

- Heap Sort is not stable.

### Performance on Nearly Sorted Data 😊

- Heap Sort is not significantly faster or slower for nearly sorted data.

We have seen these together before (Iterative Merge Sort on a Linked List), but never for an array.

### Notes

- Heap Sort is significantly slower than both Merge Sort and Introsort.
- Heap Sort can be stopped early, with useful results.
- Recall that Heap Sort is the usual fallback algorithm in Introsort.

## Review

### Heaps & PQs in the C++ STL — `std::priority_queue` [1/3]

Note the  
name of the  
header!



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.

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

*container* defaults to `std::vector`.

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

The comparison used by `priority_queue` defaults to `operator<`.

```
priority_queue<Foo> pq1; // Use operator<
```

We can specify a custom comparison.

```
priority_queue<Foo, std::vector<Foo>,  
std::greater<Foo>> pq2; // Use operator>
```

The *type* of the comparison →

We give the third template argument, so we must also give the second.

## Review

### Heaps & PQs in the C++ STL — `std::priority_queue` [3/3]

To pass our own comparison function, we specify:

- Its type, as the third template argument.
- The comparison itself, as a constructor argument.

```
auto comp = [](const Foo & a, const Foo & b)
{ return a.bar() < b.bar(); };
```

} Definition of our comparison

```
std::priority_queue<Foo, std::vector<Foo>,
    decltype(comp)> pq3(comp);
```

*See pq.cpp for  
example code.*

*We will see a more  
practical example  
near the end of  
the semester.*

↑  
The type of our  
comparison

↑  
Our comparison

# 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]

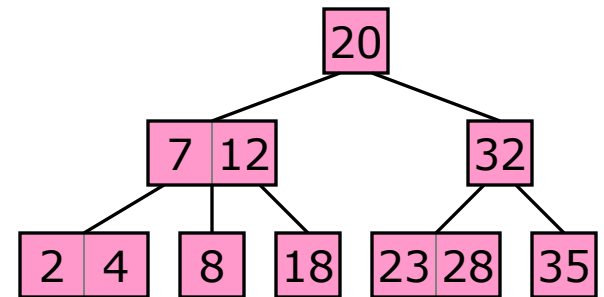
Now we look at the second of the three ideas: keeping a Binary Search Tree strongly balanced.

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.



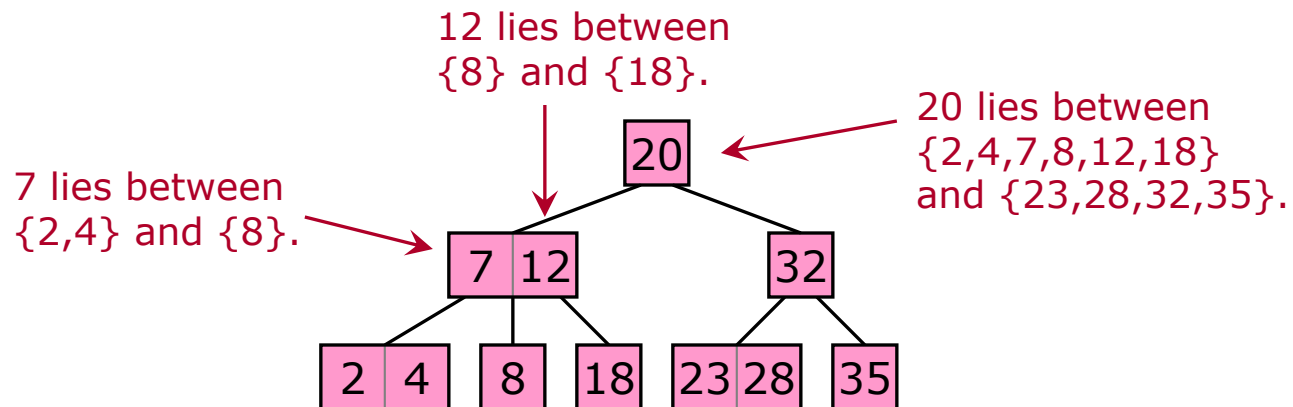
## 2-3 Trees

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

Small height is 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.

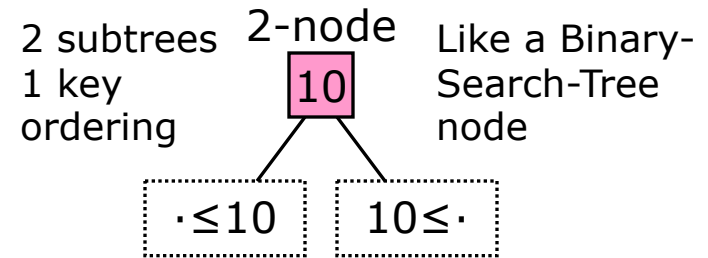


## 2-3 Trees

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

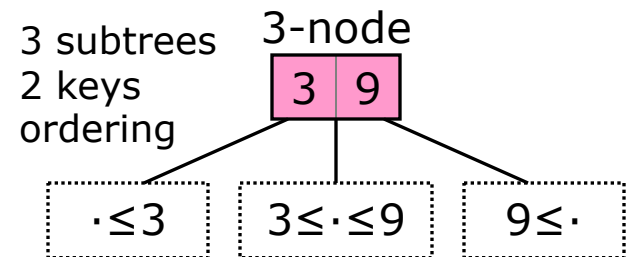
A Binary-Search-Tree style node is a **2-node**.

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

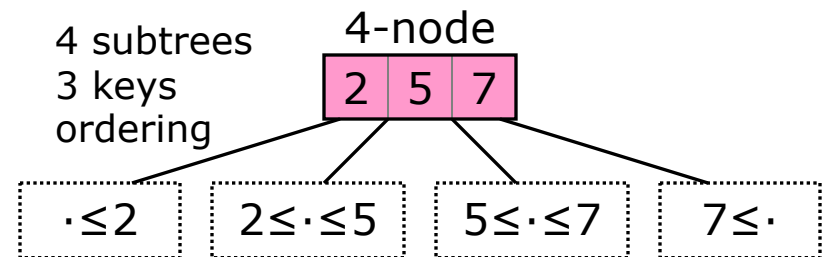


In a *2-3 Tree* we also allow a node to be a **3-node**.

- 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**.

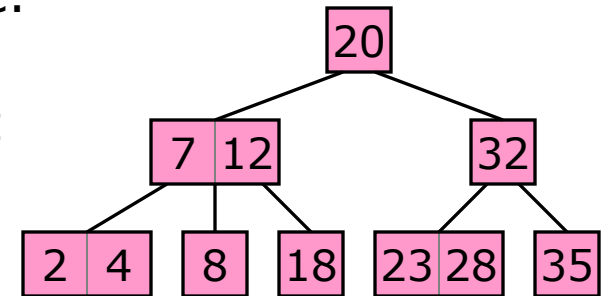


## 2-3 Trees

### Definition [1/3]

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



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.

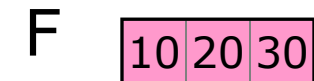
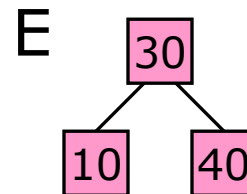
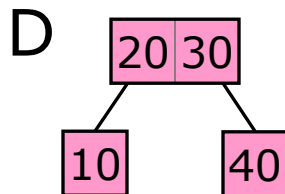
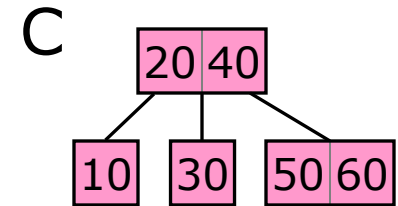
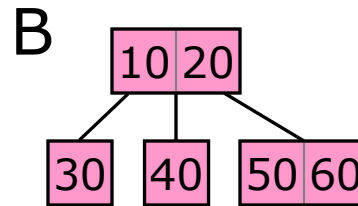
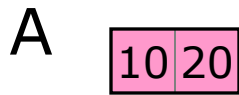
We cover algorithms for 2-3 Trees in detail. For other kinds of trees, we might say something along the lines of, “It works just like a 2-3 Tree, except ...”

## 2-3 Trees

### Definition [2/3] (Try It!)

---

Which of the following are 2-3 Trees?



*Answers on next slide.*

## 2-3 Trees

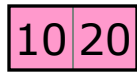
### Definition [3/3] (Try It!)

Which of the following are 2-3 Trees?

### Answers

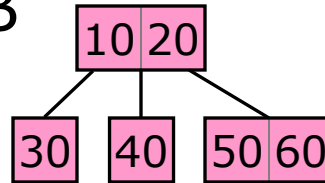
???

A



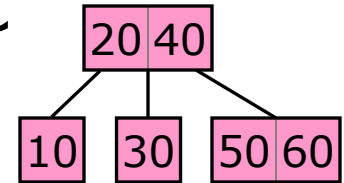
???

B



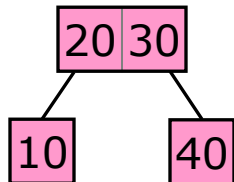
???

C



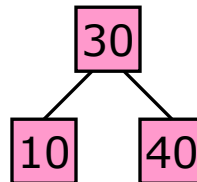
???

D



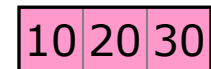
???

E



???

F

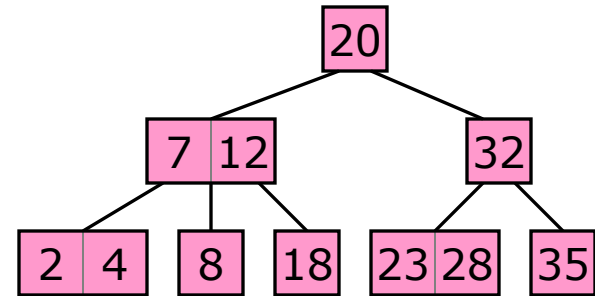


## 2-3 Trees

### 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.



## 2-3 Trees

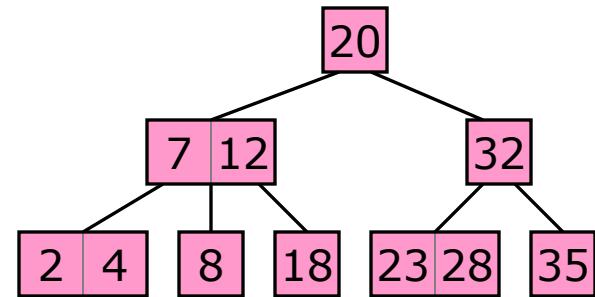
### Single-Item Operations

---

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— $\Theta(\log n)$ —algorithms that maintain the properties of the tree. That is why we like 2-3 Trees.

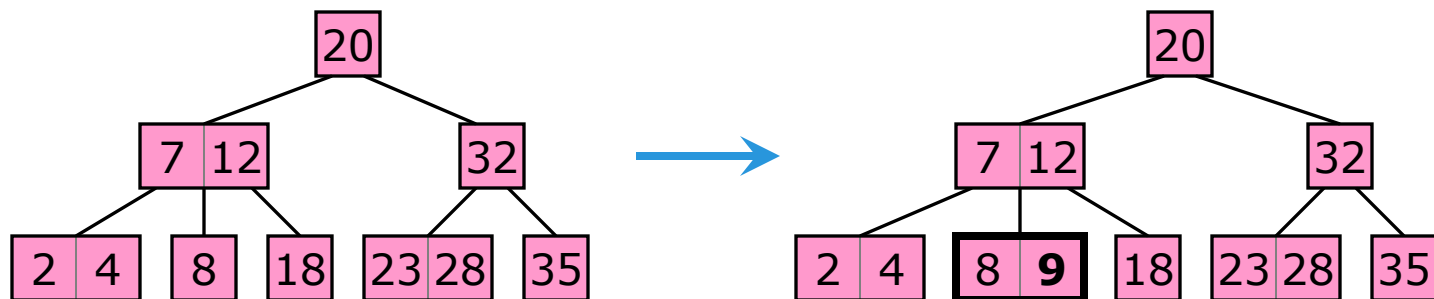
## 2-3 Trees

### 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.

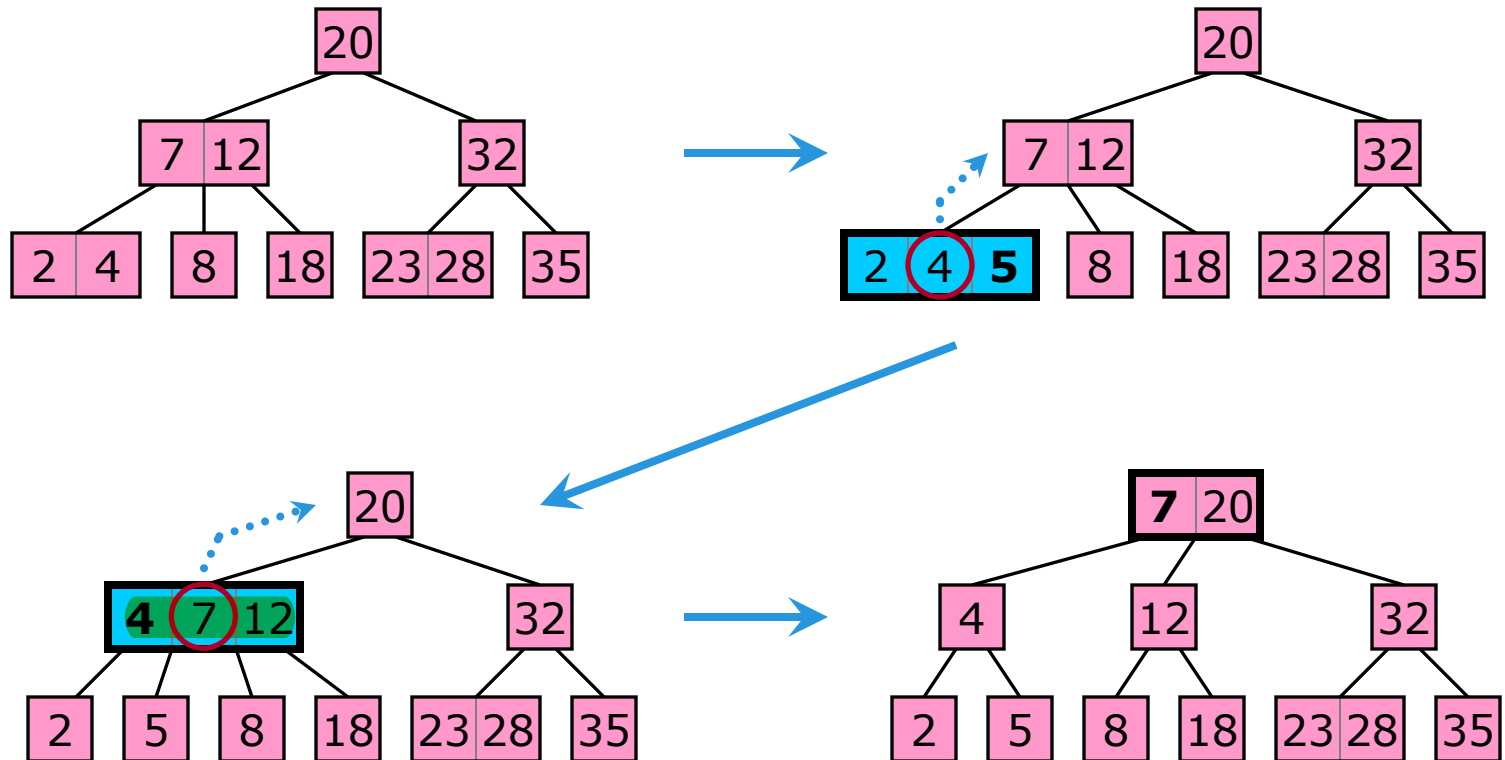


## 2-3 Trees

### Insert Algorithm [2/6]

Example 2. Insert 5.

Over-full nodes  
are **blue**.

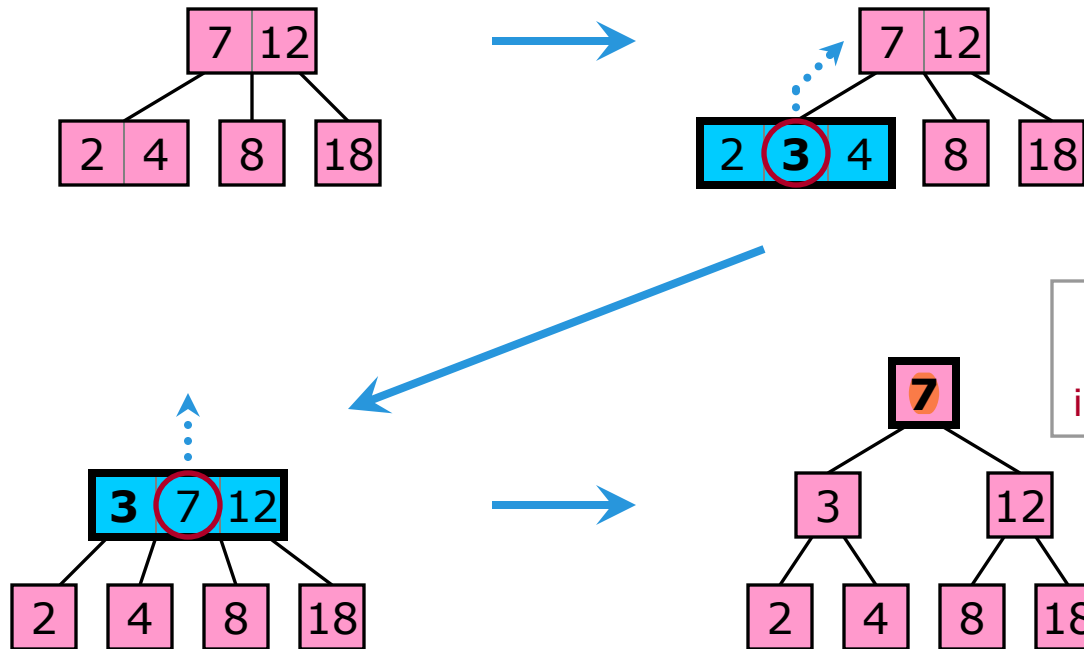


## 2-3 Trees

### Insert Algorithm [3/6]

Example 3. Insert 3.

Over-full nodes are **blue**.





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

## 2-3 Trees

### Insert Algorithm [4/6]

---

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

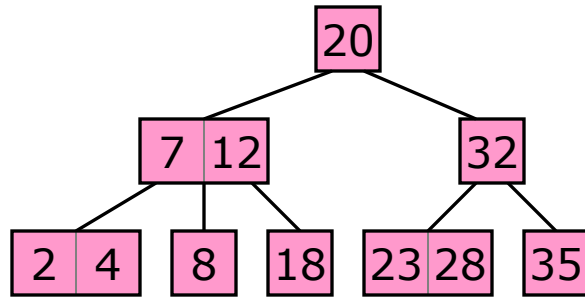
- Search: find the leaf that the new item goes in.
  - In the process of finding this leaf, you may determine that the given key is already in the tree. If so, then act accordingly.
- Add the item to the proper node. 
- If the node is over-full, then split it (dragging subtrees along, if necessary), 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.



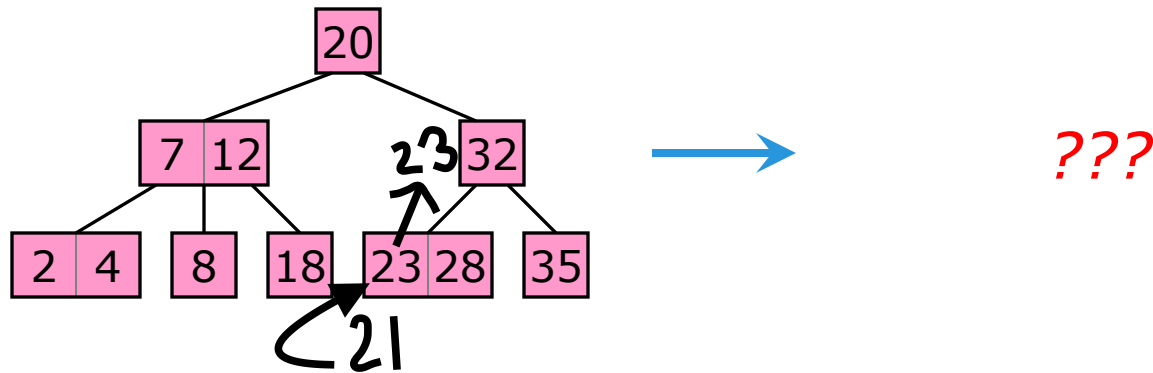
*Answer on next slide.*

## 2-3 Trees

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

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

**Answer**




## 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 search, 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.

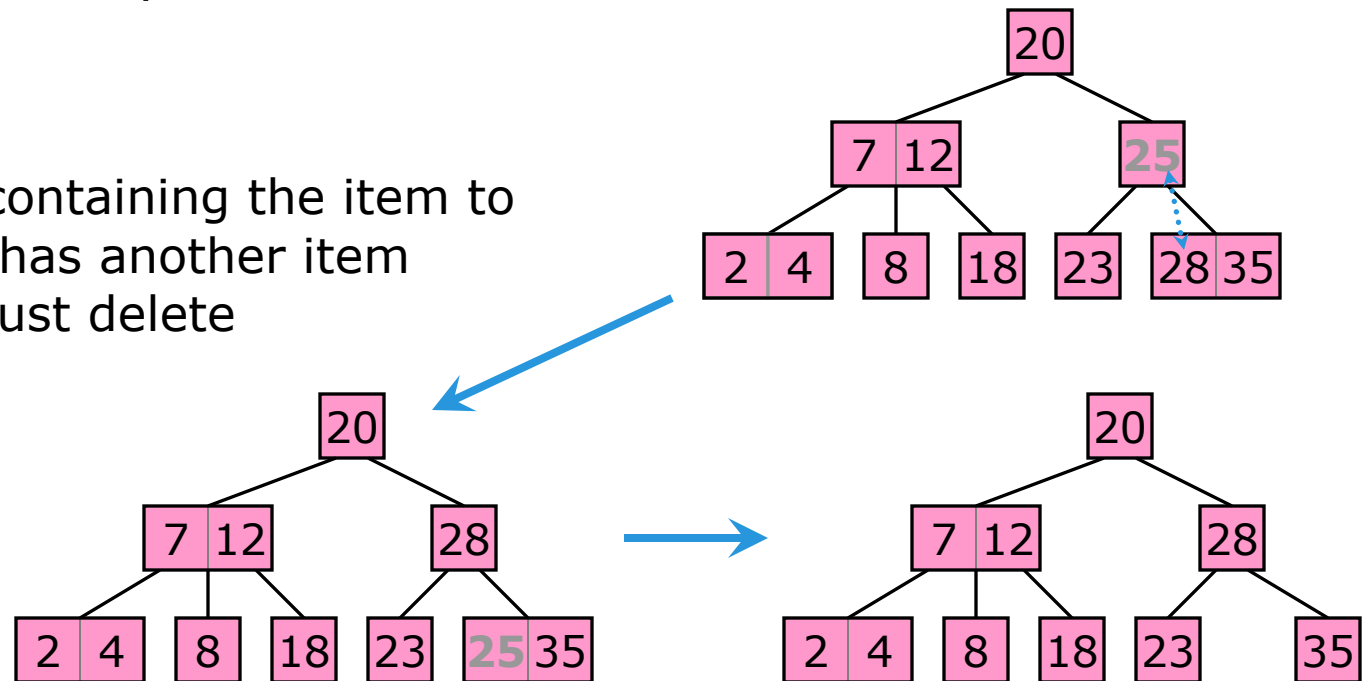
- It must have a successor, which must be a leaf. (Why?)

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.

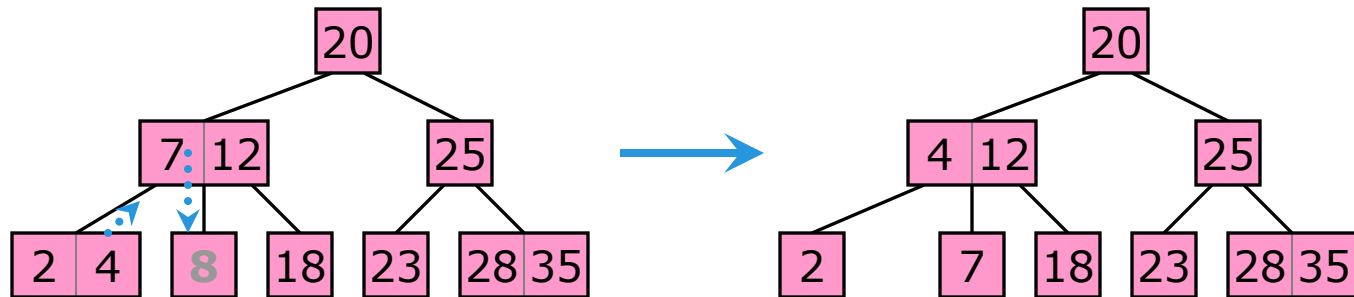
Example 1. Delete 25.



#### Semi-Easy Case

- If the item to be deleted is in a node that contains no other item—and if, next to this node, there is a sibling that contains 2 items, we can perform a **rotation** using the parent:
  - Bring an item up from the nearby sibling.
  - Bring the parent down.
  - Drag along subtrees as appropriate.

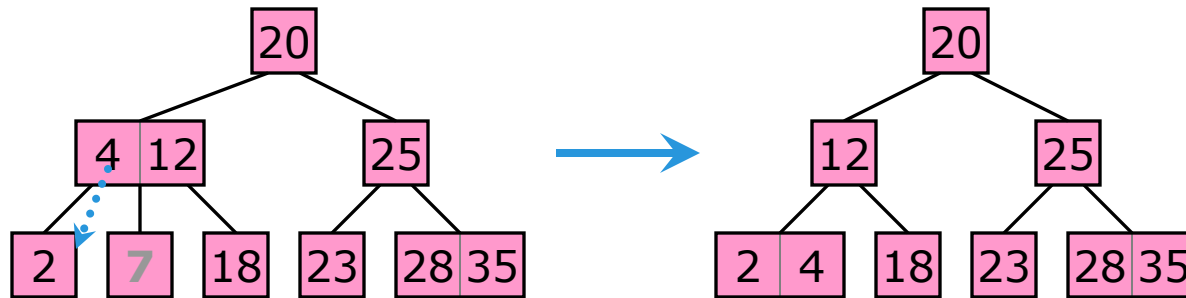
Example 2. Delete 8.



#### 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 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.



Above, “delete” 4 from the tree consisting of the top two levels. 4’s node has another item in it, so this is *easy case*; simply get rid of 4 in the parent (one level down, it goes in the node with 2).

## 2-3 Trees

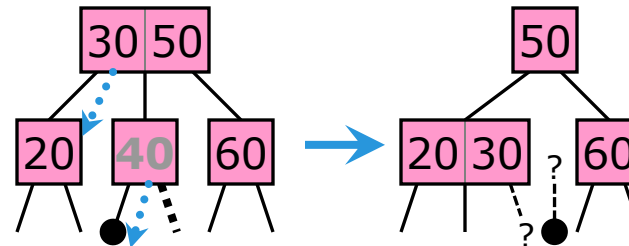
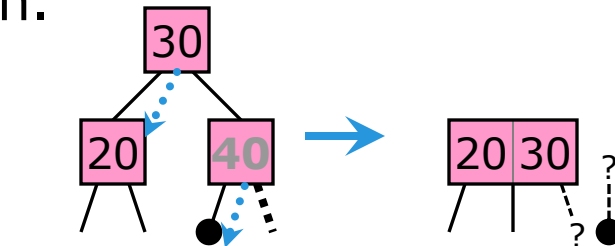
### Delete Algorithm [5/10]

In a recursive “delete”, where do *orphaned subtrees* go?

In each example, we “delete” 40. One of its subtrees is going away, and it is moving down.

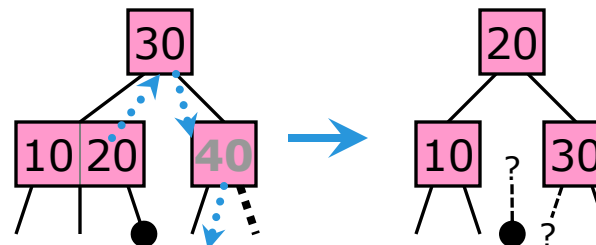
Two *hard case* examples.

- Q. What happens to other subtree of 40?
- A. ???



A *semi-easy case* example.

- 30 comes down to replace 40. 20 goes up.
- Q. What to do with the right subtree of 20?
- A. ???




There is always exactly one spot for an orphaned subtree. Put it there.

## 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

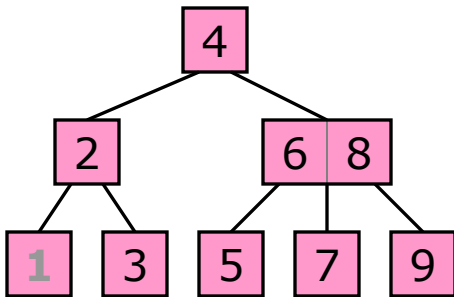
### Delete Algorithm [7/10]

---

A few more examples.

Example 4. Delete 1.

- 1 is *which case???*

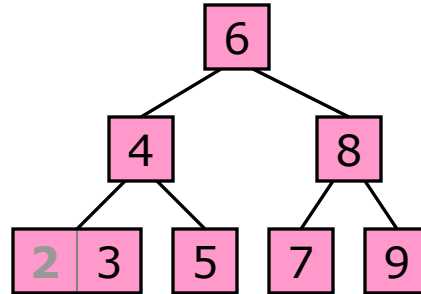


## 2-3 Trees

### Delete Algorithm [8/10]

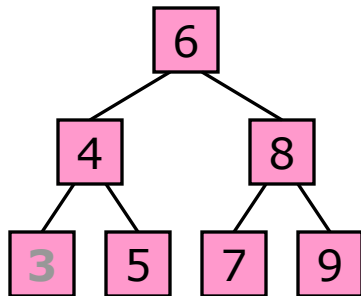
Example 5. Delete 2.

- 2 is *which case???*



Example 6. Delete 3.

- 3 is *which case???*

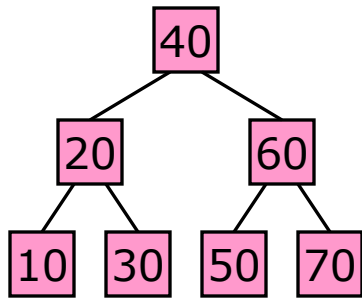


## 2-3 Trees

### Delete Algorithm [9/10] (Try It!)

---

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



*Answer on next slide.*

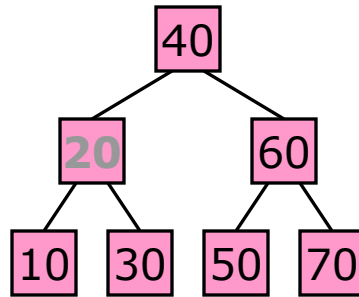
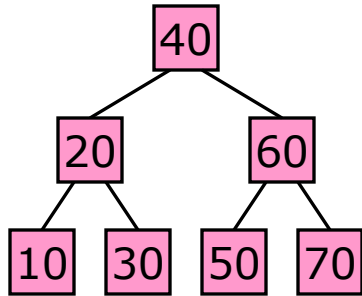


## 2-3 Trees

### Delete Algorithm [10/10] (Try It!)

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

**Answer**



???

## 2-3 Trees

### Efficiency

---

**Fact.** A 2-3 Tree with  $n$  keys must have height less than  $\log_2(n + 1)$ . So the maximum height, given  $n$ , is  $\Theta(???)$ .  
And each single-item operation follows a single root-leaf path.

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

- Traverse  
???
- Retrieve  
???
- Insert  
???
- Delete  
???

## 2-3 Trees In Practice

---

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

A. Pretty much never.

When we consider how to implement a Table, a 2-3 Tree is a good choice. But it is never quite the *best* choice. In every situation, there is some other implementation that is at least a bit better.

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 ...

## 2-3 Trees

### TO BE CONTINUED ...

---

*2-3 Trees* will be continued next time.