

# Heaps and Priority Queues in Practice

## 2-3 Trees

---

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, April 25, 2008

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

[CHAPPELLG@member.ams.org](mailto:CHAPPELLG@member.ams.org)

© 2005–2008 Glenn G. Chappell

# Unit Overview

## Tables & Priority Queues

---

### Major Topics

- ✓ • Introduction to Tables
- ✓ • Priority Queues
- ✓ • Heap algorithms
  - Heaps and Priority Queues in practice
  - 2-3 Trees
  - Other balanced search trees
  - Hash Tables
  - Prefix Trees
  - Tables in practice

# Review

## Introduction to Tables

---

	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

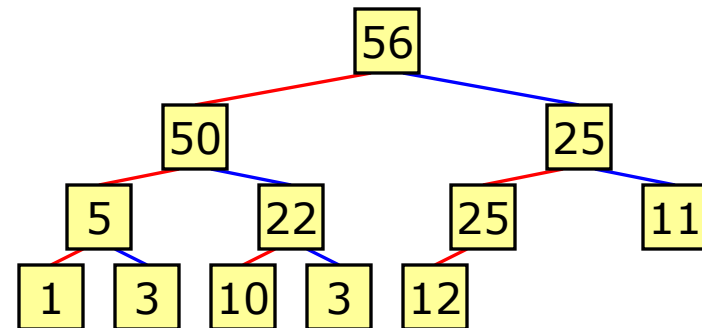
# Review

## Heap Algorithms [1/6]

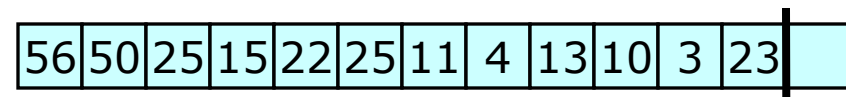
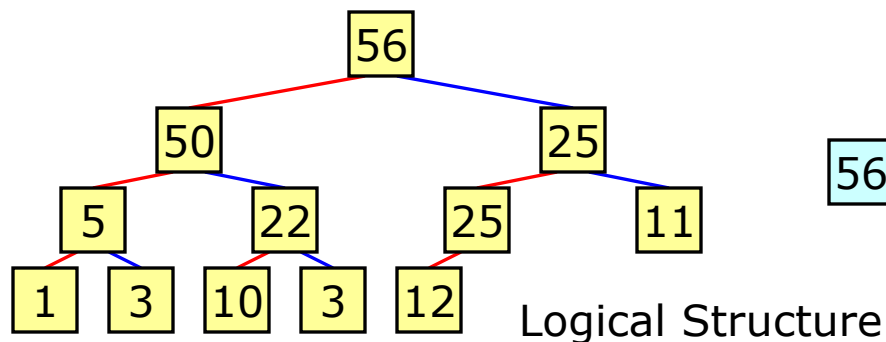
---

We define a **Heap** to be a complete Binary Tree that

- Is empty,
- Or else
  - The root's key (priority) is  $\geq$  than the key of each of the root's children, if any, and
  - Each of the root's subtrees is a Heap.



The usual implementation of a Heap uses an array-based complete Binary Tree.



Physical Structure

# Review

## Heap Algorithms [2/6]

---

We discussed how to perform the Heap operations.

- To do getFront:
  - Look at item 0 (the root item in the Heap).
- To do delete:
  - Swap the first and last items.
  - Reduce the size of the Heap by 1.
  - “Trickle down” the root item.
- To do insert:
  - Add a new item with the value to be inserted.
  - “Trickle up” the new item.

### Efficiency

- GetFront is constant time.
- Delete is logarithmic time.
- Insert is logarithmic time if no reallocation is required.
  - And, due the way we use Heaps, typically it is never required.
  - It might be required (making the order linear time) if we are using a Heap to implement an encapsulated Priority Queue.

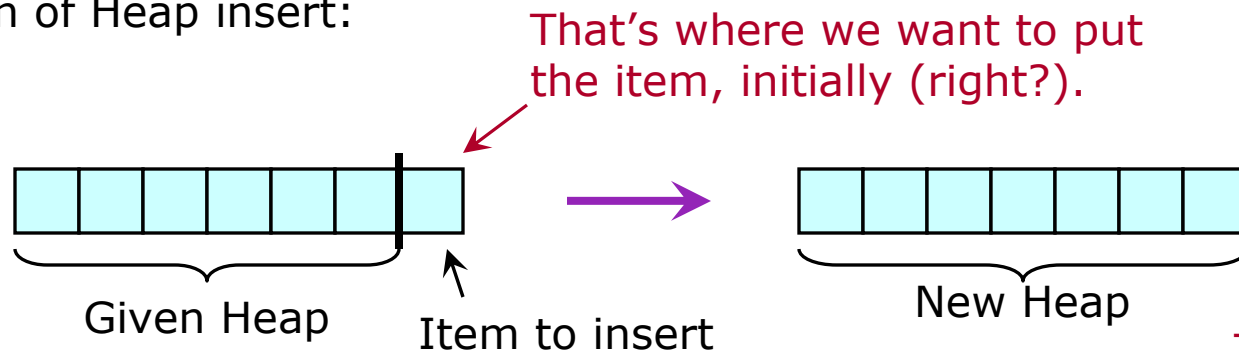
# Review

## Heap Algorithms [3/6]

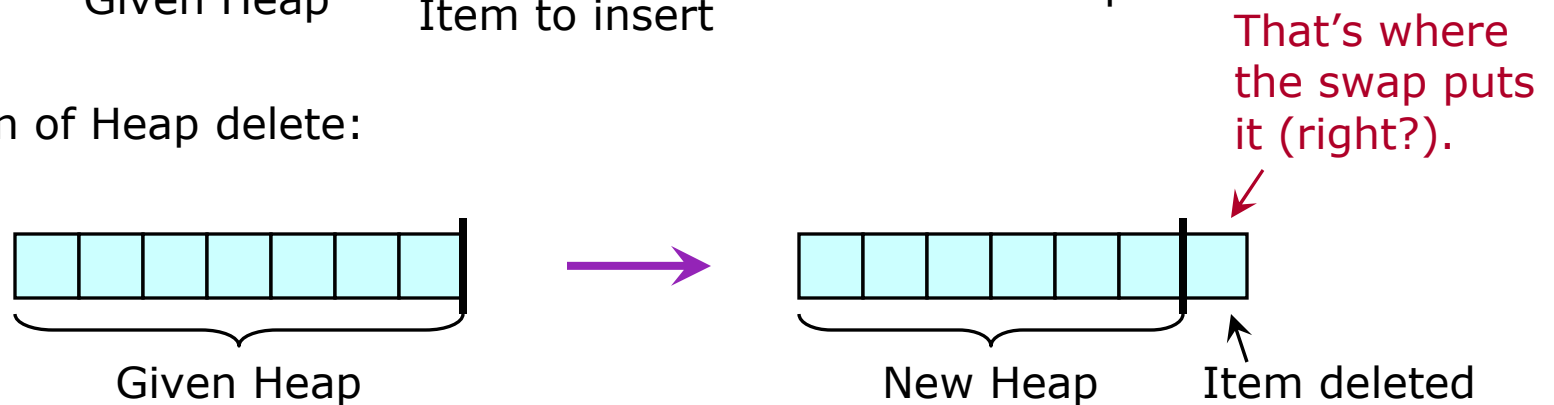
---

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

- Action of Heap insert:



- Action of Heap delete:



Note that Heap algorithms can do **all** their work using **swap**.

- This usually allows for both speed and safety.

# Review

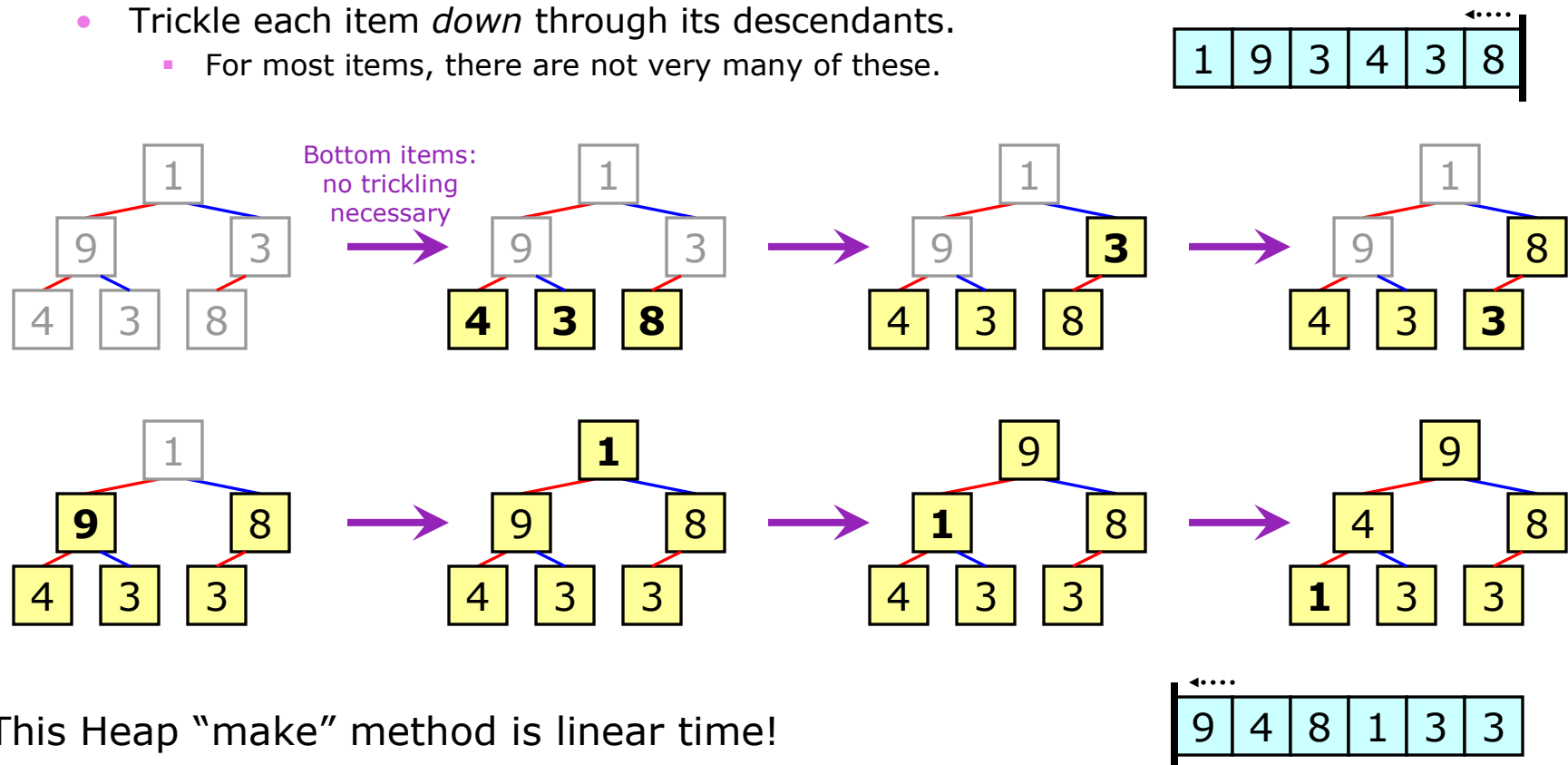
## Heap Algorithms [4/6]

To turn a random-access range (array?) into a Heap, we *could* do  $n-1$  Heap inserts.

- Each insert operation is  $O(\log n)$ , and so making a Heap in this way is  $O(n \log n)$ .

However, we can make a Heap **faster** than this.

- Place each item into a partially-made Heap, in **backwards order**.
- Trickle each item *down* through its descendants.
  - For most items, there are not very many of these.



This Heap "make" method is linear time!

## Review

### Heap Algorithms [5/6]

---

Our last sorting algorithm is **Heap Sort**.

- This is a sort that uses Heap algorithms.
- We can think of it as using a Priority Queue, where the priority of an item is its value.
- Procedure: Make a Heap, then delete all items, using the delete procedure that places the deleted item in the top spot.
- We do a **make** operation, which is  $O(n)$ , and  $n$  getFront/delete operations, each of which is  $O(\log n)$ .
- Total:  $O(n \log n)$ .

# Review

## Heap Algorithms [6/6]

---

### Efficiency 😊

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

### Requirements on Data 😞

- Heap Sort requires random-access data.

### Space Usage 😊

- Heap Sort uses only constant additional storage.
- In particular, it can be done 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.

### Notes

- Heap Sort can be generalized to handle sequences that are modified (in certain ways) in the middle of sorting.
- Recall that Heap Sort is used by Introsort, when the recursion depth of Quicksort exceeds the maximum allowed.

# Heaps and Priority Queues in Practice

## Heap Algorithms in the C++ STL

---

The C++ STL includes several Heap algorithms.

- These operate on ranges specified by pairs of random-access iterators.
  - **Any** random-access range can be a Heap: array, vector, deque, part of these, etc.
- An STL Heap is a Maxheap with an optional client-specified comparison.
- Heap algorithms are used by STL Priority Queues (`std::priority_queue`).

Example: `std::push_heap` (in `<algorithm>`) inserts into an existing Heap.

- Called as `std::push_heap(first, last)`.
- Assumes `[first, last)` is nonempty, and `[first, last-1)` is already a Heap.
- Inserts `*(last-1)` into the Heap.

Similarly:

- `std::pop_heap`
  - Heap delete operation. Puts the deleted element in `*(last-1)`.
- `std::make_heap`
  - Make a range into a Heap.
- `std::sort_heap`
  - Is given a Heap. Does a bunch of `pop_heap` calls.
  - Calling `make_heap` and then `sort_heap` does Heap Sort.
- `std::is_heap`
  - Tests whether a range is a Heap.

# Heaps and Priority Queues in Practice

## `std::priority_queue` — Introduction

---

The STL has a Priority Queue: `std::priority_queue`, in `<queue>`.

- STL documentation does not call `std::priority_queue` a “container”, but rather a “container adapter”.
- This is because `std::priority_queue` is explicitly a wrapper around some other container.

You get to pick what that container is.

- You say “`std::priority_queue<T, container<T> >`”.
  - “`T`” is the value type.
  - “`container`” can be `std::vector` or `std::deque`.
  - “`container<T>`” can be any standard-conforming **random-access** sequence container.
- `container` defaults to `std::vector`.
  - You can say just “`std::priority_queue<T>`” to get “`std::priority_queue<T, std::vector<T> >`”.

# Heaps and Priority Queues in Practice

## `std::priority_queue` — Members

---

The member function names used by `std::priority_queue` are the same as those used by `std::stack`.

- Not those used by `std::queue`.
- Thus, `std::priority_queue` has “top”, not “front”.

Given a variable `pq` of type `std::priority_queue<T>`, you can do:

- `pq.top()`
- `pq.push(item)`
  - “*item*” is some value of type `T`.
- `pq.pop()`
- `pq.empty()`
- `pq.size()`

# Heaps and Priority Queues in Practice

## `std::priority_queue` — Comparison

---

How do we specify an item's priority?

- We really don't need to know an item's priority; we only need to know, given two items, which has the **higher** priority.
- Thus, we use a comparison.
- A third, optional template parameter is a "comparison object":

```
std::priority_queue<T, std::vector<T>, compare>
```

- Comparison objects work the same as those passed to STL sorting algorithms (`std::sort`, etc.) and STL Heap algorithms.
- So, for example, a priority queue of `ints` whose highest priority items are those with the lowest value, would have the following type:

```
std::priority_queue<int, std::vector<int>, std::greater<int>()>
```

# Advanced Table Implementations

## Overview

---

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:
    - **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**

## 2-3 Trees

### Introduction & Definition [1/3]

---

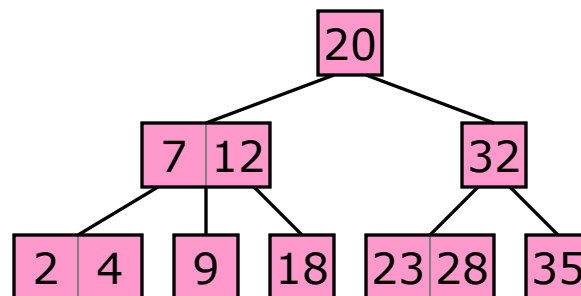
Obviously (?) a Binary Search Tree is a useful idea. The problem is keeping it balanced.

- Or at least keeping the height small.

It turns out that small height is much easier to maintain if we allow a node to have more than 2 children.

But if we do this, how do we maintain the “search tree” concept?

- We generalize the idea of an inorder traversal.
- For each pair of consecutive subtrees, a node has one data item lying between the values in these subtrees.



# 2-3 Trees

## Introduction & Definition [2/3]

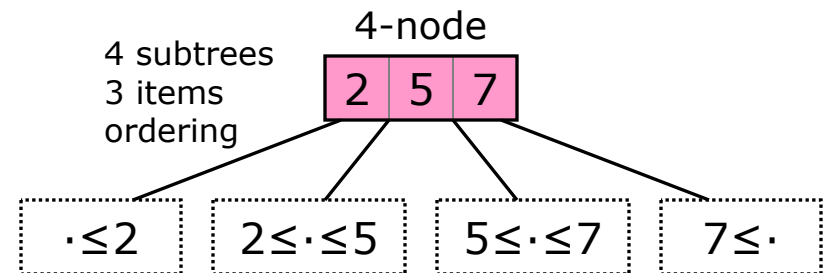
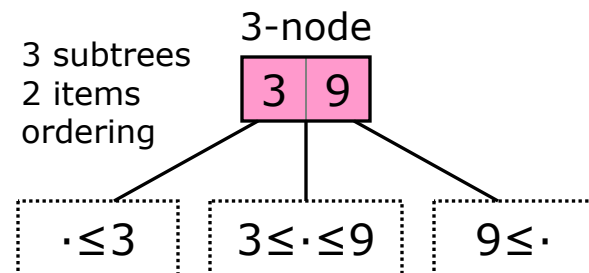
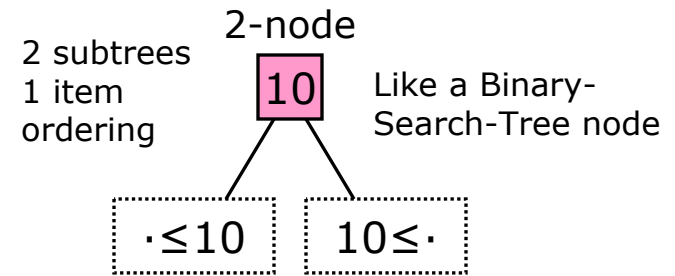
A Binary-Search-Tree style node is a **2-node**.

- 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**.



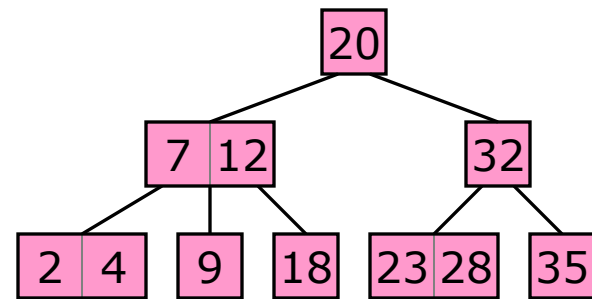
# 2-3 Trees

## Introduction & Definition [3/3]

---

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  $\leq$  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.



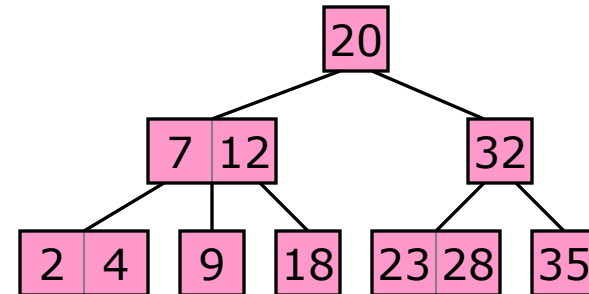
# 2-3 Trees

## 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 & Delete

---

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

- These are the tough problems.
- It turns out that both have efficient [ $O(\log n)$ ] algorithms, which is why we like 2-3 Trees.

## 2-3 Trees

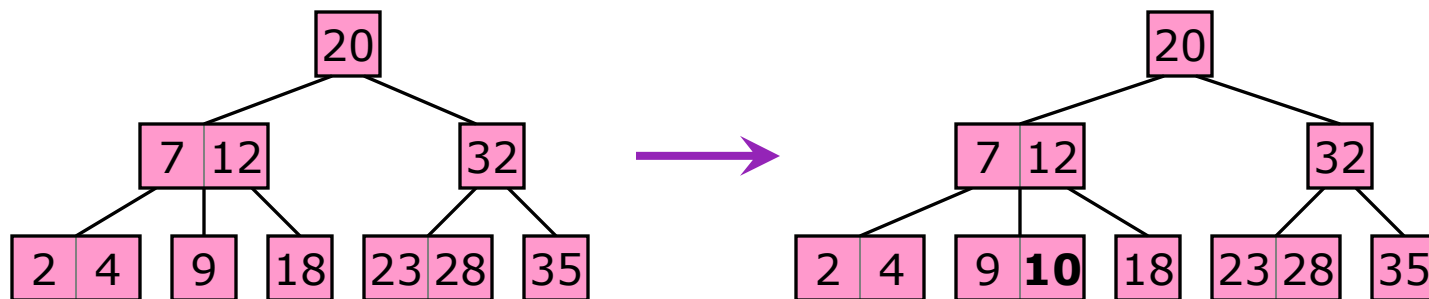
### Operations — Insert [1/4]

---

Basic ideas behind the 2-3 Tree **insert** algorithm:

- 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.



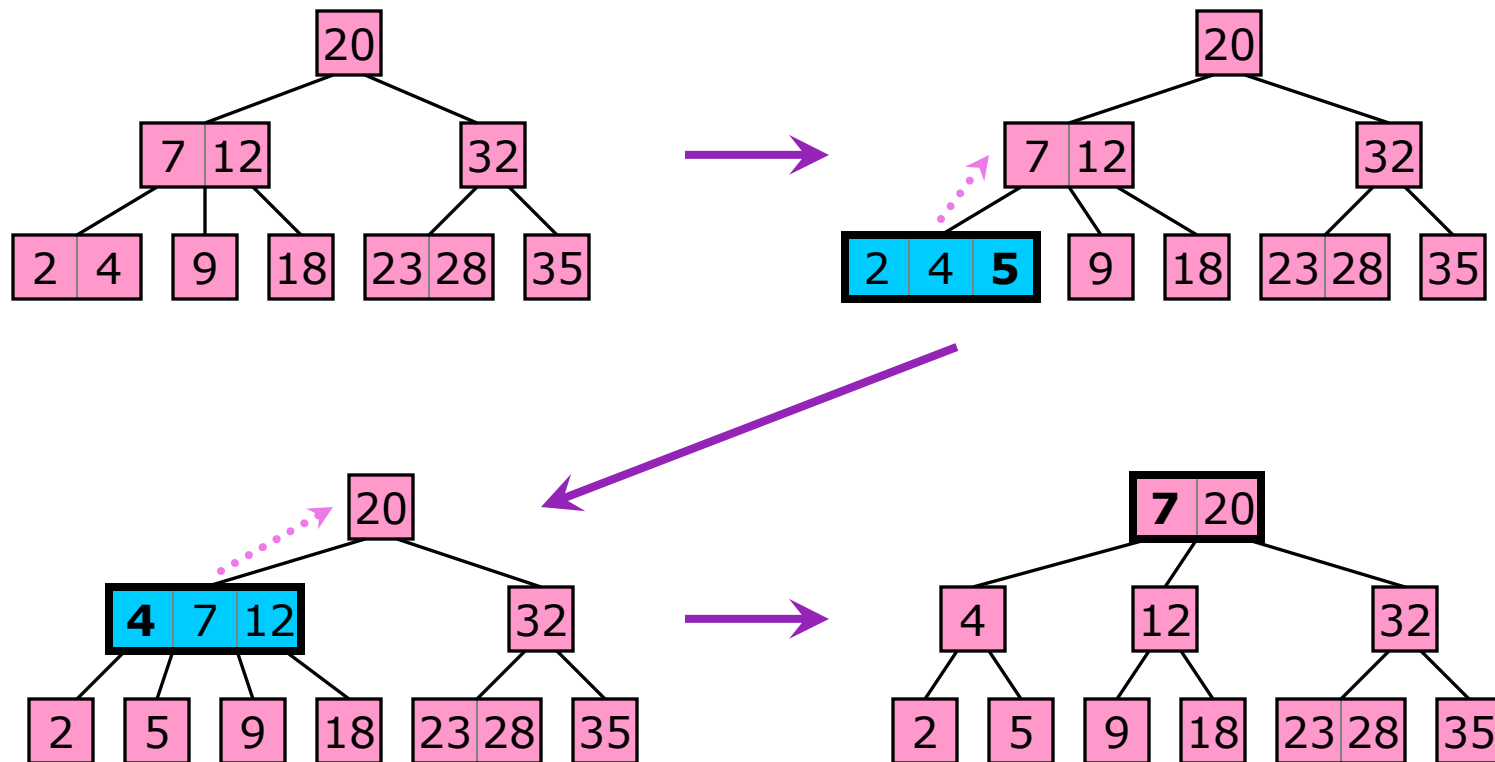
# 2-3 Trees

## Operations — Insert [2/4]

---

### Example 2: Insert 5.

- Over-full nodes are blue.



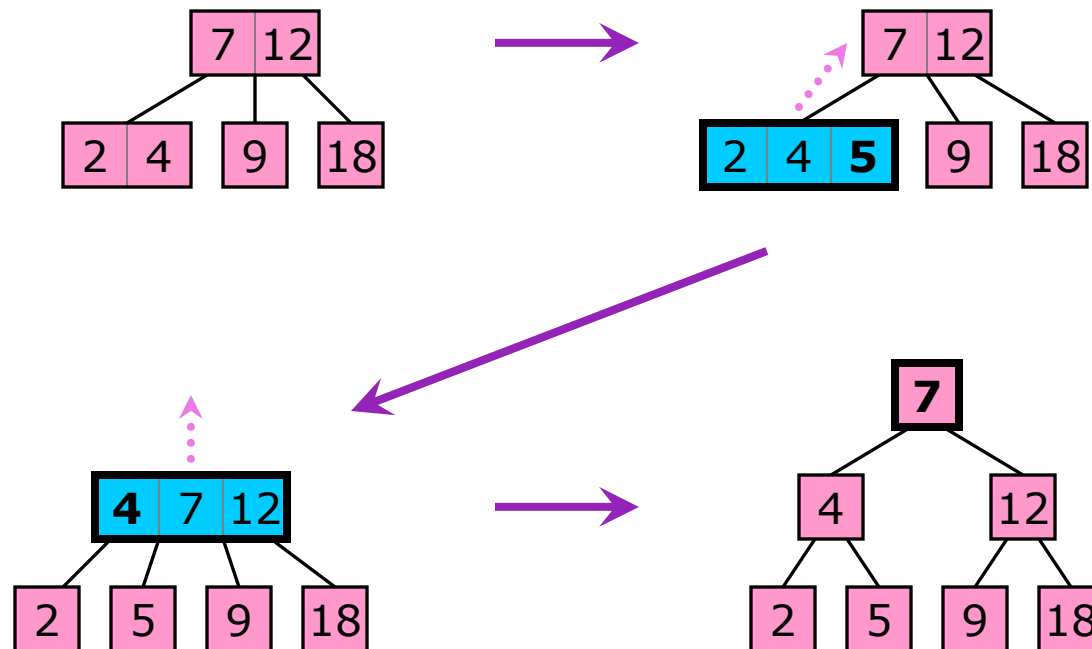
# 2-3 Trees

## 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]

---

In the middle of a 2-3 Tree insertion, overfull nodes are always leaves or 4-nodes.

- (Of course 4-nodes are illegal.)
- This is why we can always split at the middle item.

2-3 Tree insertion can be thought of recursively.

- Insert into a node.
  - This node will be a leaf the first time.
- If the node is overfull, split and move the middle item up.
  - We split the subtree rooted at the node. If the node is a leaf, this is easy. If the node is not a leaf, this is more work, but not hard to understand.
- Moving an item up is inserting into the parent.
  - So we recurse.
  - Or we make increase the height (by making a new root), and we are done.

## 2-3 Trees

### Operations — Delete [1/6]

---

**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 [2/6]

---

### Observation

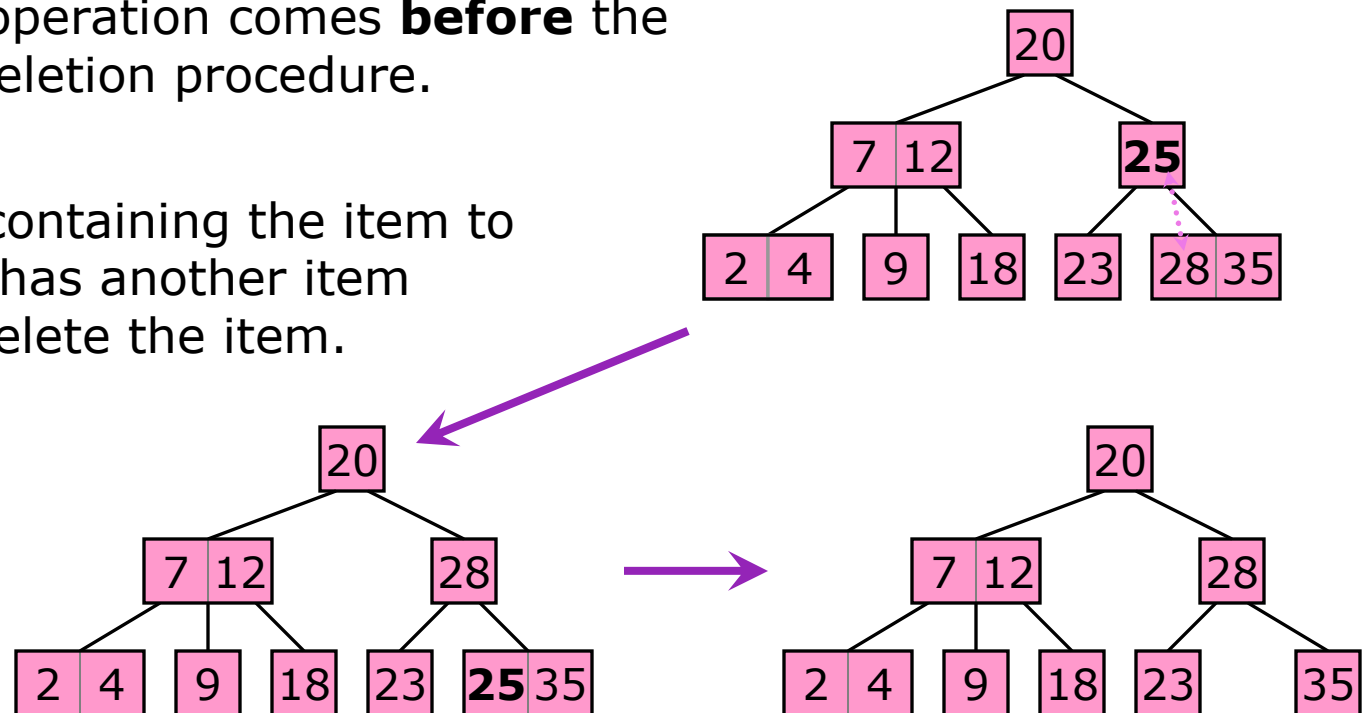
- We can always start our deletion at a leaf.
- If the item to be deleted is not in a leaf, swap it with its “inorder” successor.
  - It must have one. (Why?)
- This swap operation comes **before** the recursive deletion procedure.

### 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

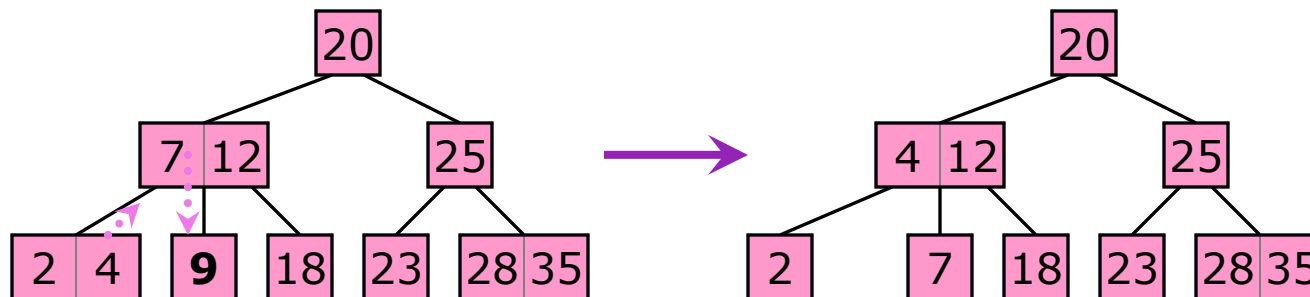
## Operations — Delete [3/6]

---

### 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.



## 2-3 Trees

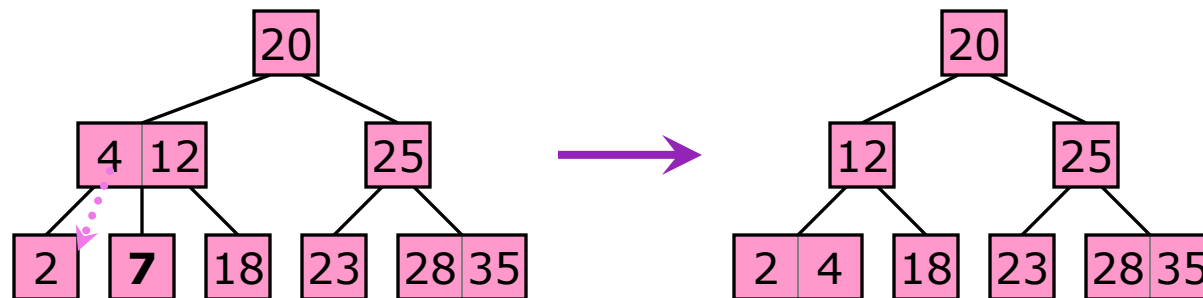
### Operations — Delete [4/6]

---

#### 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 combine 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).

# 2-3 Trees

## Operations — Delete [5/6]

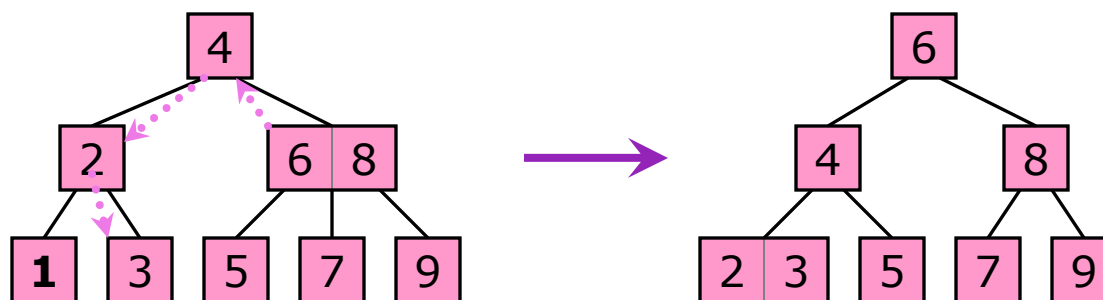
---

### Various Examples

- Here are a few more ...

### Example: Delete 1.

- 1 is “hard case”, so we bring down the parent (recursively “delete” 2) and combine it with 3 in a single node.
- 2 is “semi-easy case”, so rotate (6 to 4 to 2).
- The 5 comes with the 6. 5 goes to the left of 6, which means to the right of 4.



## 2-3 Trees

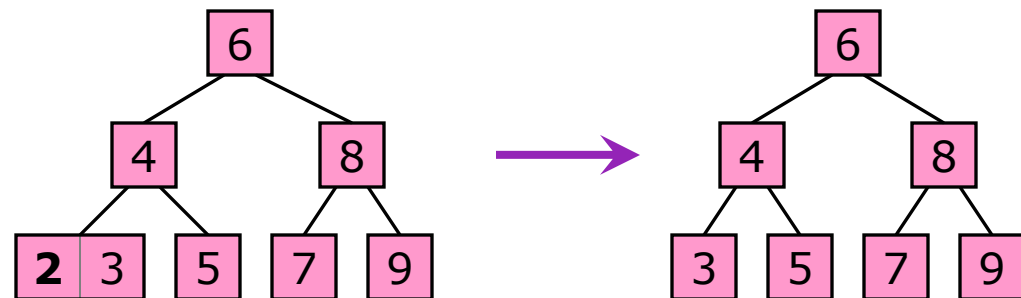
### Operations — Delete [6/6]

---

Various Examples, cont'd

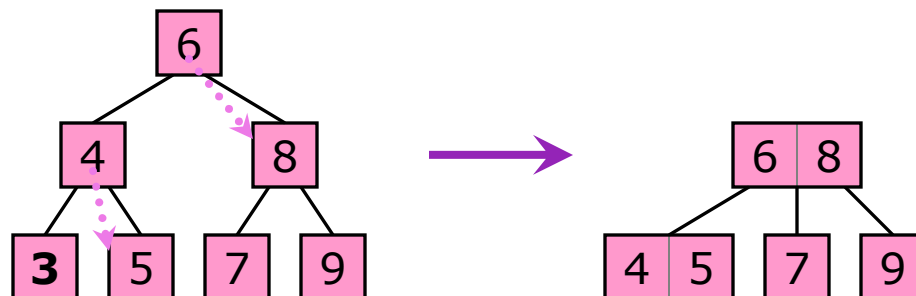
Example: Delete 2.

- This is "easy case".



Example: Delete 3.

- This is "hard case". We need to bring down 4 and combine it with 5.
- 4 is "hard case". We need to bring down 6 and combine it with 8.
- 6 is the root. We reduce the height of the tree.



## 2-3 Trees

### 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.