Heap Algorithms
Heaps & Priority Queues in the C++ STL
2-3 Trees

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, April 15, 2013

Chris Hartman

Department of Computer Science

University of Alaska Fairbanks

cmhartman@alaska.edu

Based on material by Glenn G. Chappell

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

Idea #1: Restricted Table

- 2-3 Trees
- Other balanced search trees

Idea #2: Keep a Tree Balanced

- Hash Tables

Idea #3: “Magic Functions”

- Prefix Trees
- Tables in various languages

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.

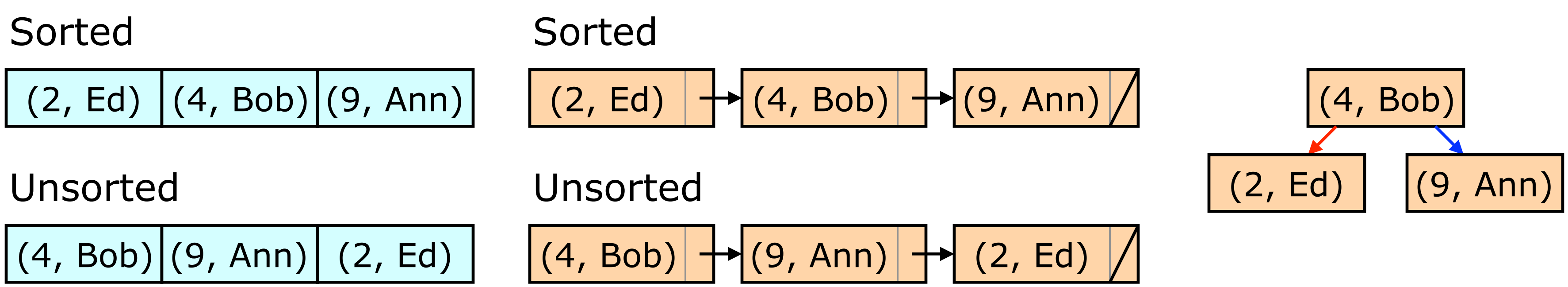
Table

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”

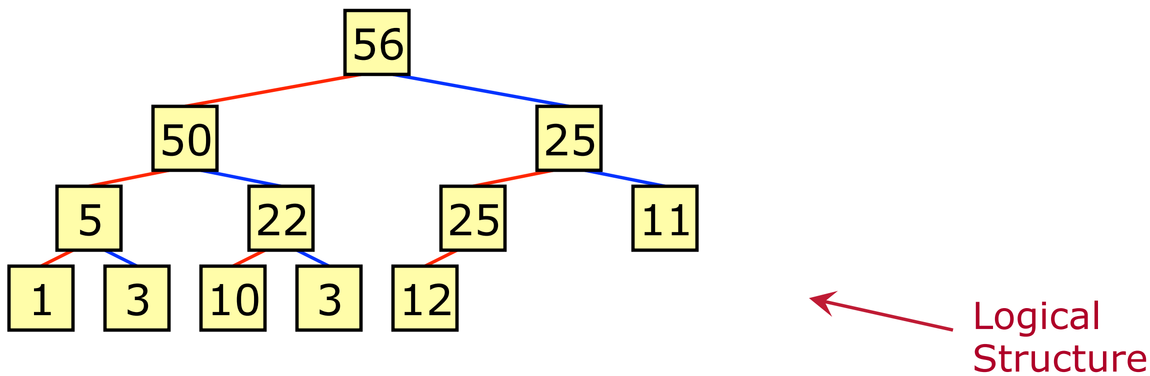
- Use an unsorted array of key-data pairs. Allow array items to be marked as “empty”.
- Have a “magic function” that tells the index of an item.
- Retrieve/insert/delete in constant time? (Actually no, but this is still a worthwhile idea.)

We will look at what results from these ideas:

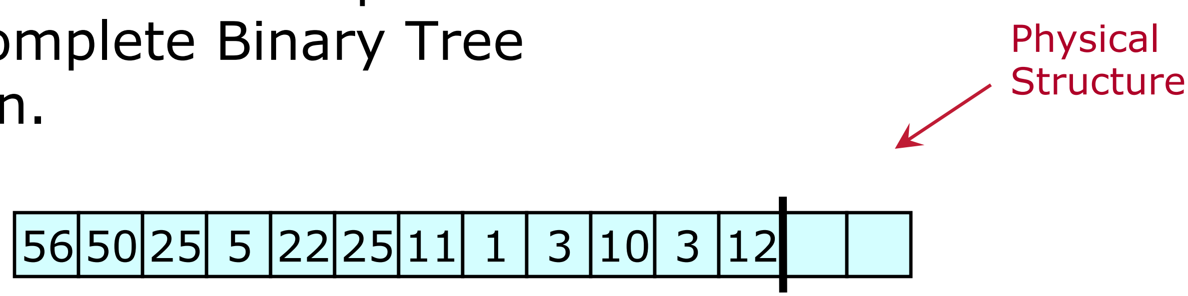
- From idea #1: Priority Queues
- From idea #2: Balanced search trees (2-3 Trees, Red-Black Trees, B-Trees, etc.)
- From idea #3: Hash Tables

Review
Binary Heap Algorithms [1/5]

A **Binary Heap** (usually just **Heap**) is a complete Binary Tree in which *no* node has a data item that is less than the data item in either of its children.



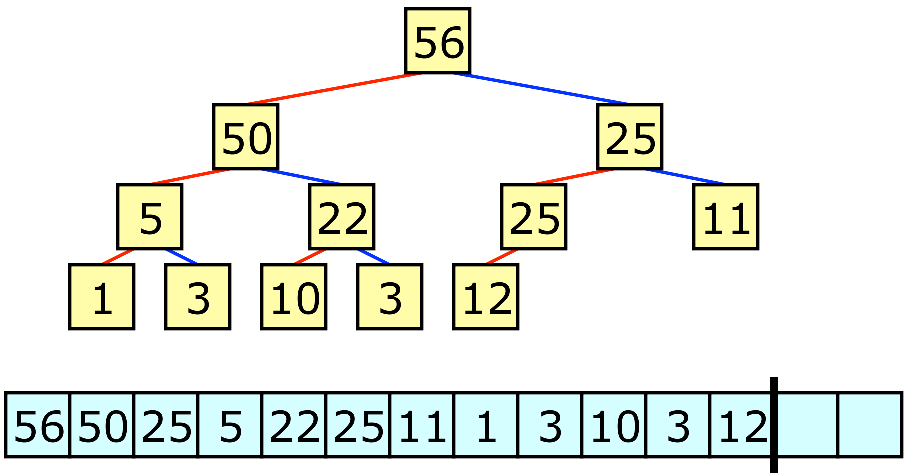
In practice, we often use "Heap" to refer to the array-based complete Binary Tree implementation.



Review
Binary Heap Algorithms [2/5]

We can use a Heap to implement a **Priority Queue**.

- Like a Table, but retrieve/delete only highest key.
  - Retrieve is called "getFront".
- Key is called "priority".
- Insert *any* key-data pair.



Algorithms for the Three Primary Operations

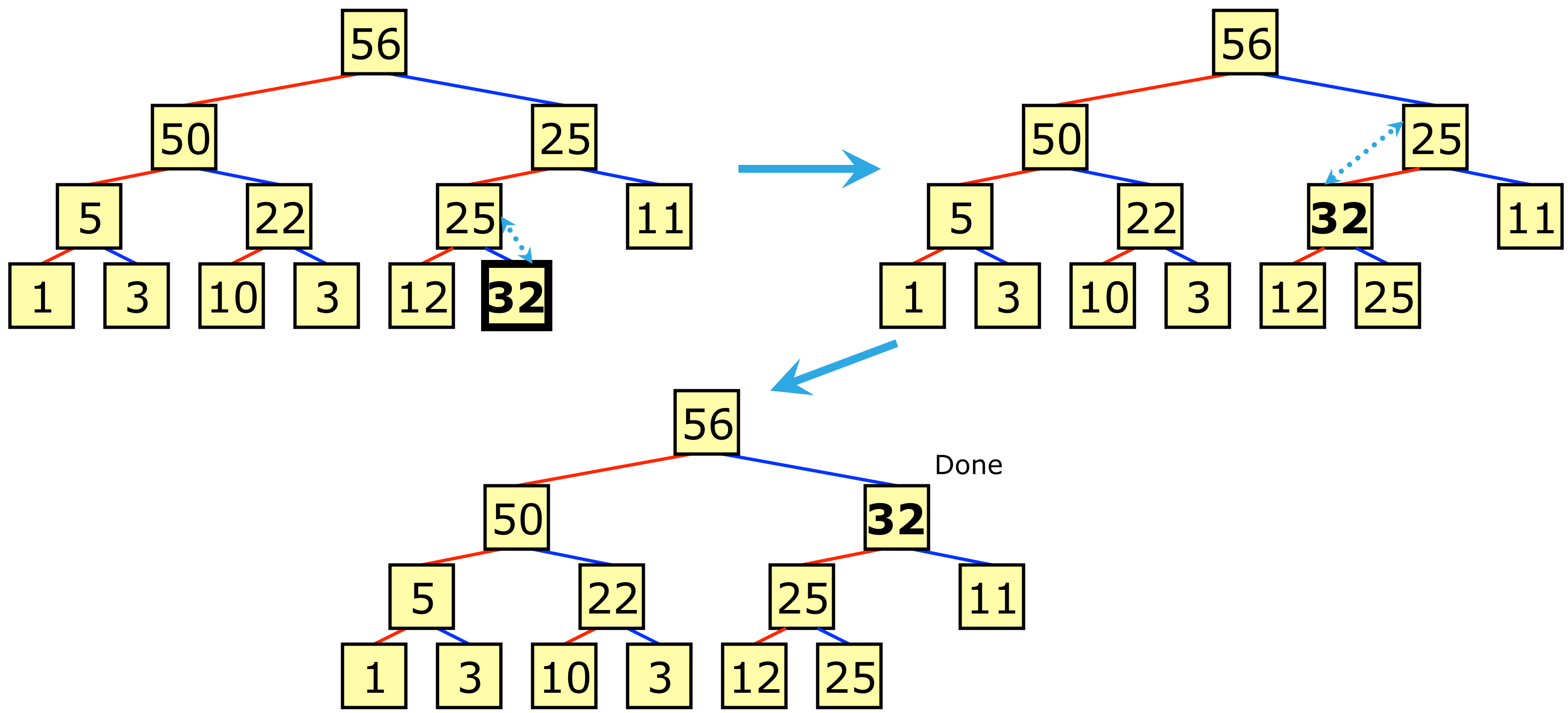
- GetFront
  - Get the root node.
  - Constant time.
- Insert
  - Add new node to end of Heap, "trickle up".
  - Logarithmic time if no reallocate-and-copy required.
    - Linear time if it may be required. Note: Heaps often do *not* manage their own memory, in which case the reallocation will not be part of the Heap operation.
- Delete
  - Swap first & last items, reduce size of Heap, "trickle down" root.
  - Logarithmic time.

Faster than linear time!

Review
Binary Heap Algorithms [3/5]

To insert into a Heap, add new node at the end. Then "trickle up".

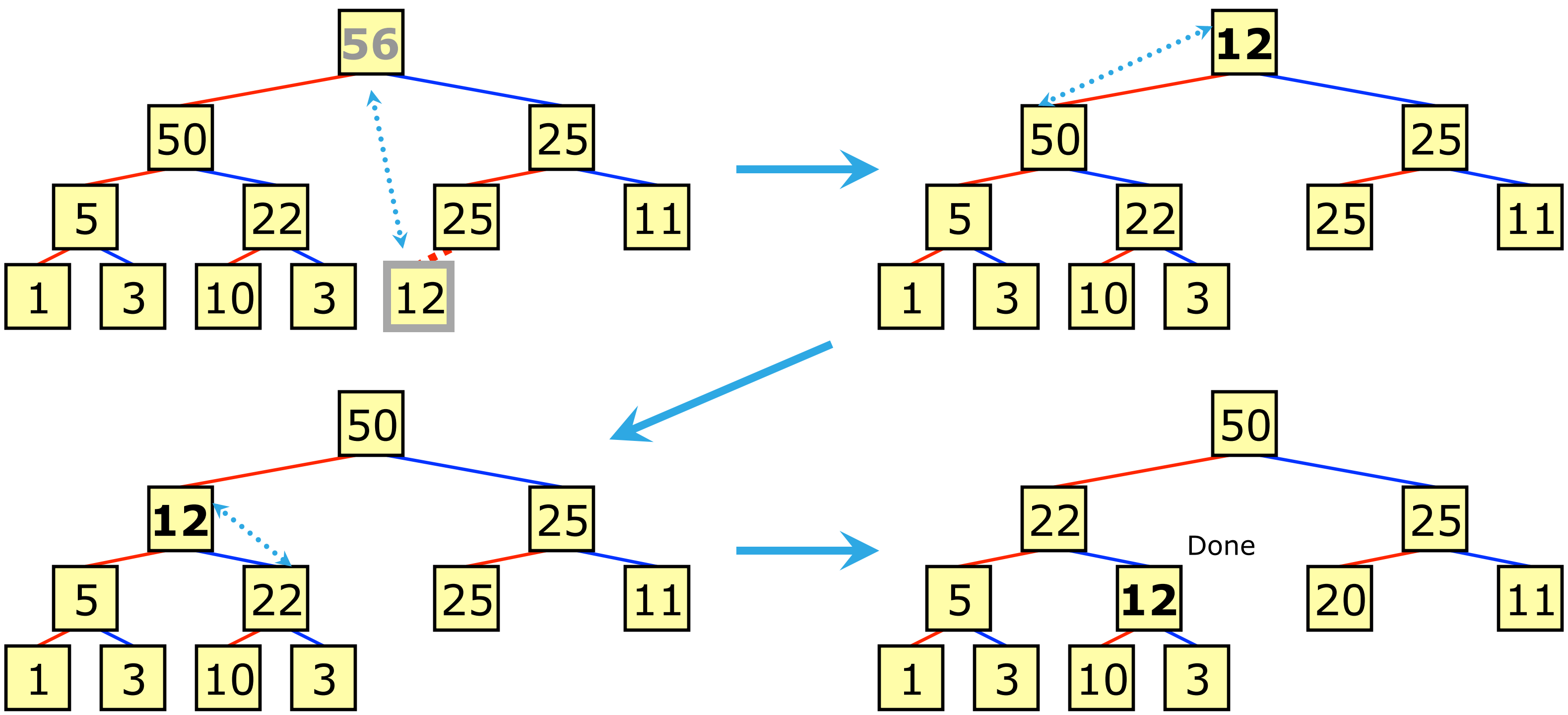
- If new value is greater than its parent, then swap them. Repeat at new position.



Review
Binary Heap Algorithms [4/5]

To delete the root item from a Heap, swap root and last items, and reduce size of Heap by one. Then "trickle down" the new root item.

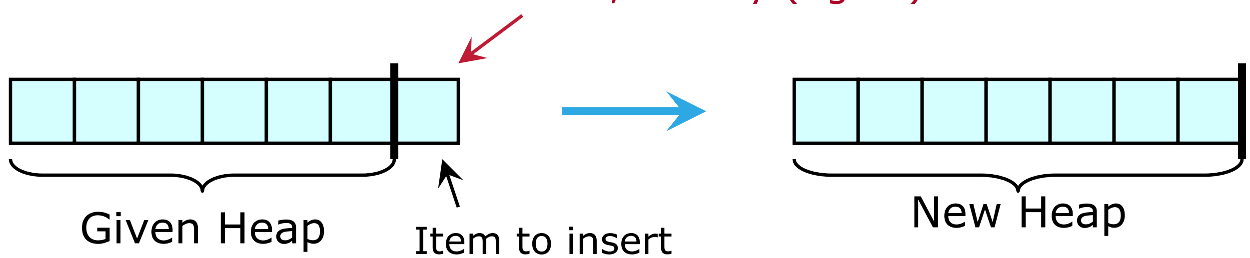
- If the root is ≥ all its children, stop.
- Otherwise, swap the root item with its **largest** child and recursively fix the proper subtree.



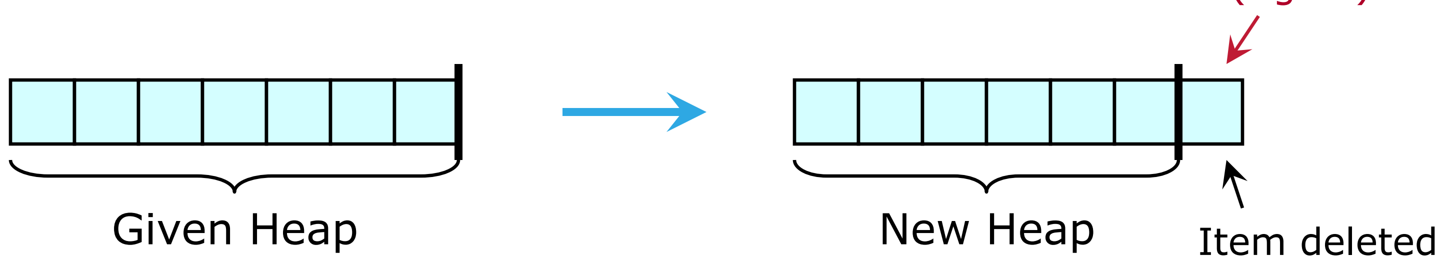
Review
Binary Heap Algorithms [5/5]

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.

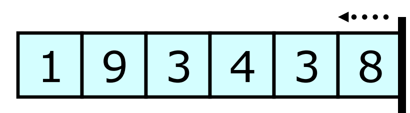
Binary Heap Algorithms
An Efficient "Make" Operation

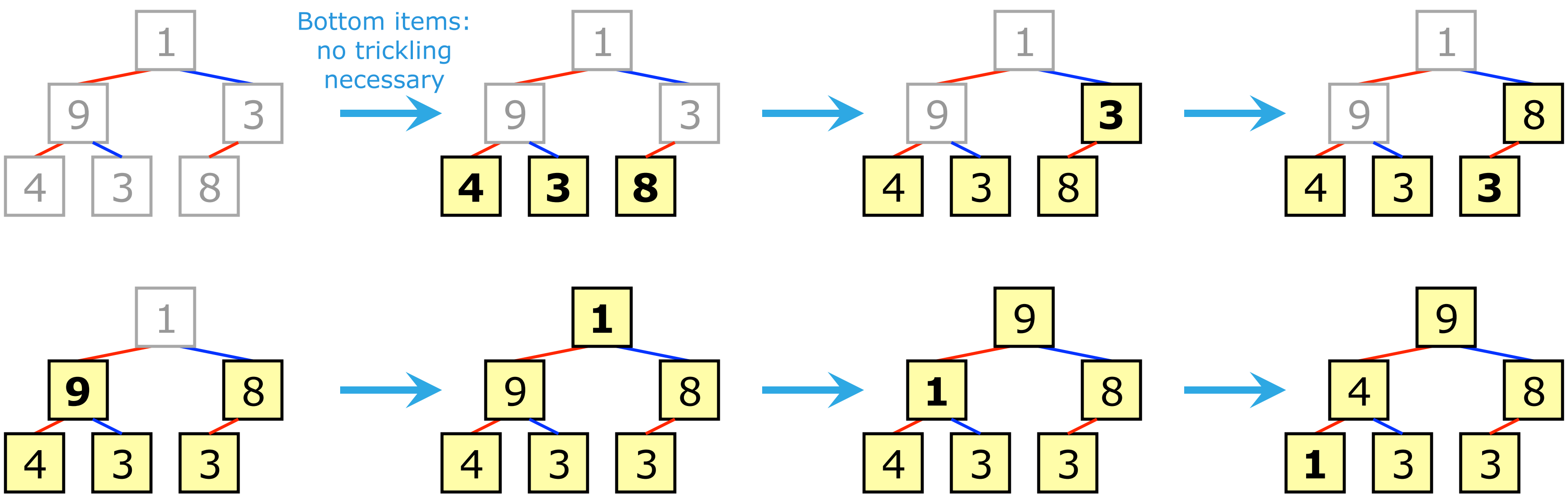
To turn a random-access range (array?) into a Heap, we *could* do n−1 Heap inserts.

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

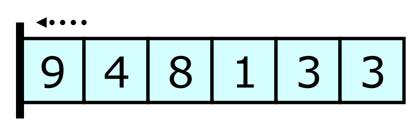
However, we can make a Heap **faster** than this.

- 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!



Binary Heap Algorithms
Heap Sort — Introduction

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 — except that the algorithm is in-place, using no separate data structure.
- 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: $O(n \log n)$.

Binary Heap Algorithms
Heap Sort — Properties

Heap Sort can be done in-place.

- We can create a Heap in a given array.
- As each item is removed from the Heap, put it in the array element that is removed from the Heap.
  - Starting the delete by swapping root and last items does this.
- Results
  - Ascending order, if we used a Maxheap.
  - Only constant additional memory required.
  - Reallocation is avoided.

Heap Sort uses less additional space than Introsort or array Merge Sort.

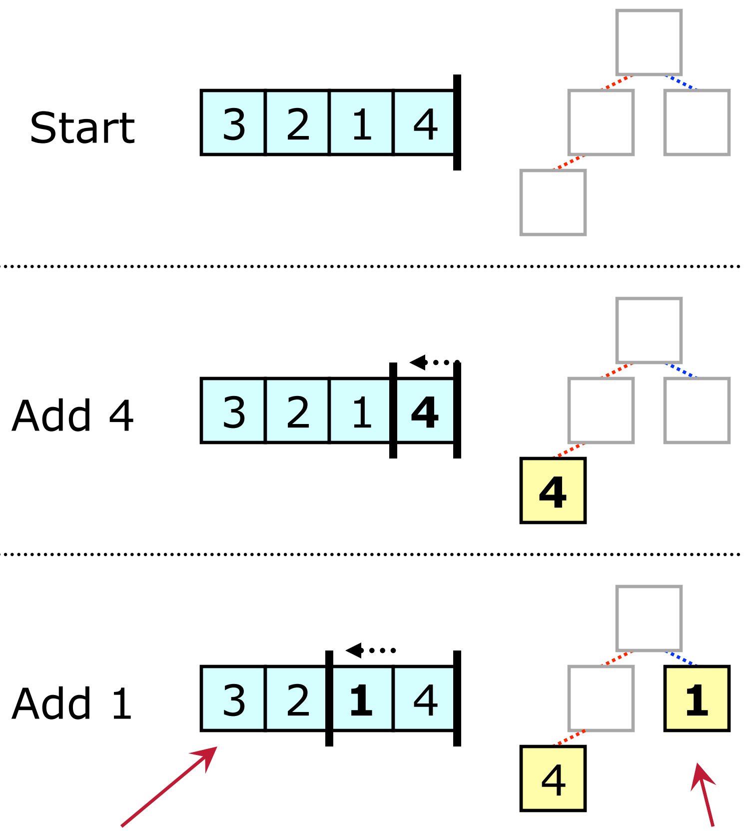
- Heap Sort: $O(1)$.
- Introsort: $O(\log n)$.
- Merge Sort on an array: $O(n)$.

Heap Sort also can easily be generalized.

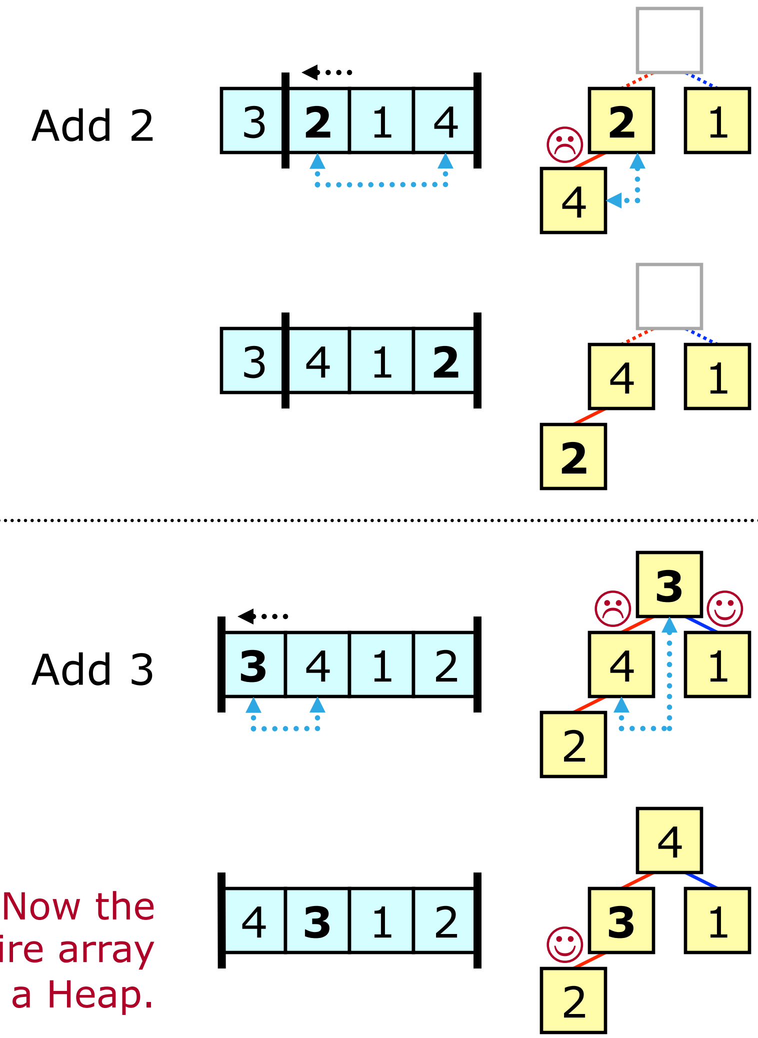
- Doing Heap inserts in the middle of the sort.
- Stopping before the sort is completed.

Binary Heap Algorithms
Heap Sort — Illustration [1/2]

Below: Heap make operation. Next slide: Heap deletion phase.



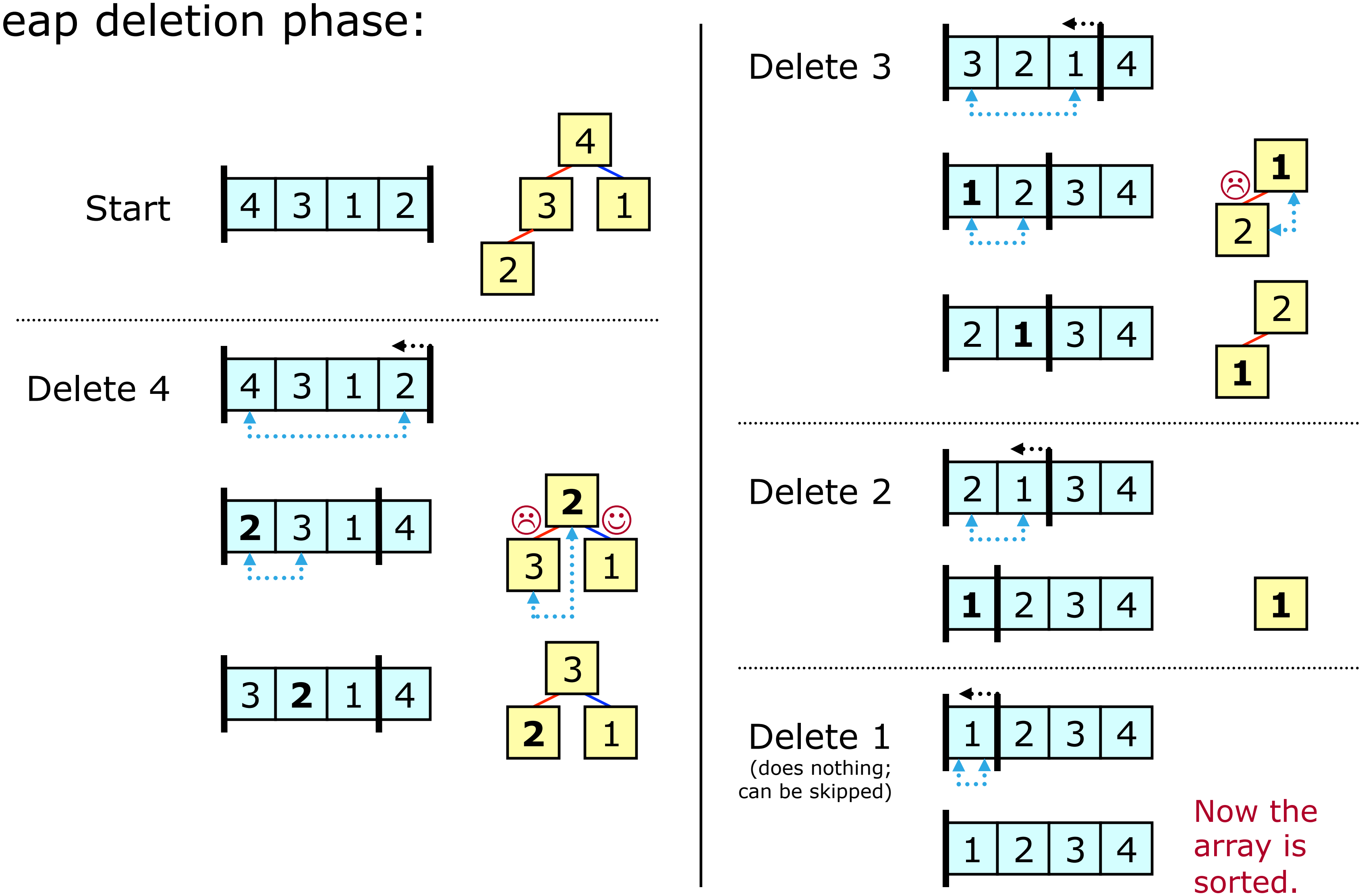
Note: This is what happens in memory. This is just a picture of the logical structure.



Now the entire array is a Heap.

Binary Heap Algorithms
Heap Sort — Illustration [2/2]

Heap deletion phase:



Binary Heap Algorithms
Heap Sort — Analysis

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

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 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 depth of the Quicksort recursion exceeds the maximum allowed.

Binary Heap Algorithms
Thoughts

In practice, a Heap is not so much a data structure as it is an ordinary random-access sequence with a particular ordering property.

Associated with Heaps are a collection of algorithms that allow us to efficiently create Priority Queues and do comparison sorting.

- These **algorithms** are the things to remember.
- Thus the subject heading.

Heaps & Priority Queues in the C++ STL
Heap Algorithms

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 & Priority Queues in the C++ STL
`std::priority_queue` — Introduction

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

- Once again, STL documentation calls **`std::priority_queue`** a "container adapter", not a "container".

As with **`std::stack`** and **`std::queue`**, **`std::priority_queue`** is a wrapper around a container that you choose.

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

- "**`T`**" is the value type.
- "*container*<**`T`**>" can be any standard-conforming **random-access** sequence container with value type **`T`**.
- In particular "*container*" can be **`std::vector`**, **`std::deque`**, or **`std::basic_string`**.
  - But not **`std::list`**.

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

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

Heaps & Priority Queues in the C++ STL
`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 & Priority Queues in the C++ STL
`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, which defaults to **`operator<`**.
- 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 **`int`**s 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>()>
```

Recall:
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

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:
    - **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]

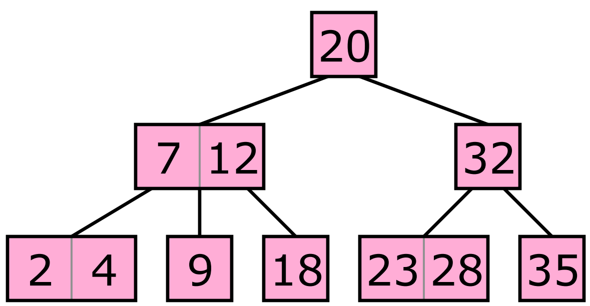
Obviously (?) a Binary Search Tree is a useful idea. The problem is keeping it balanced.

- 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]

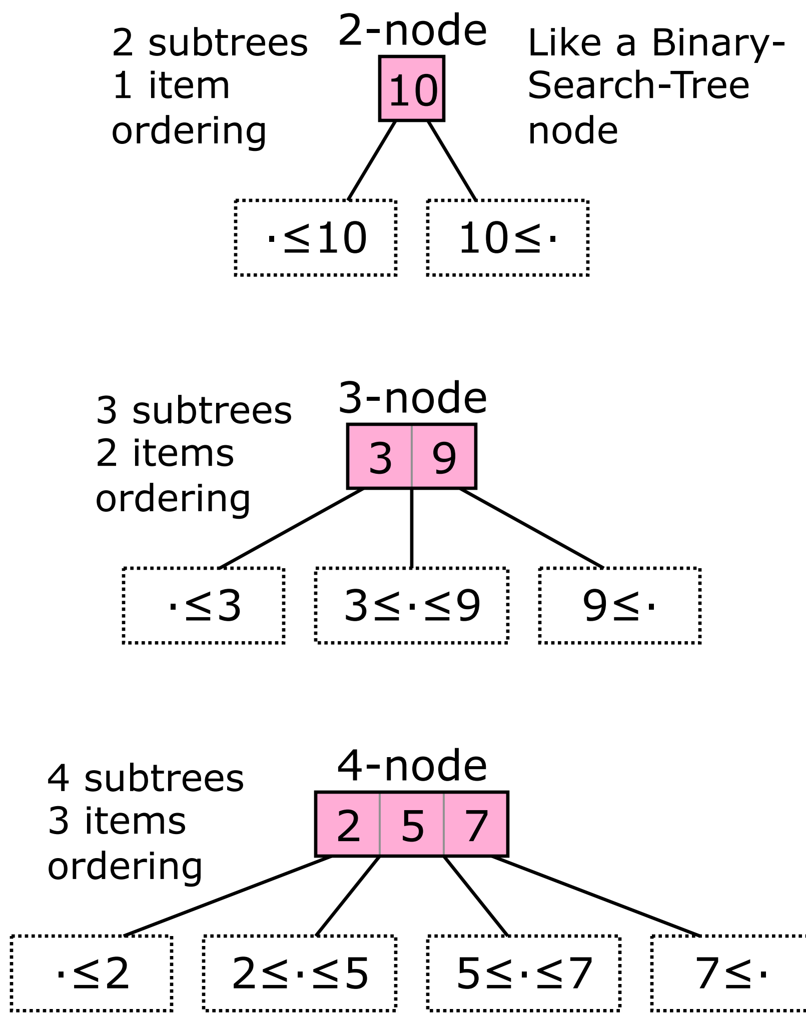
A Binary-Search-Tree style node is a **2-node**.

- 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.

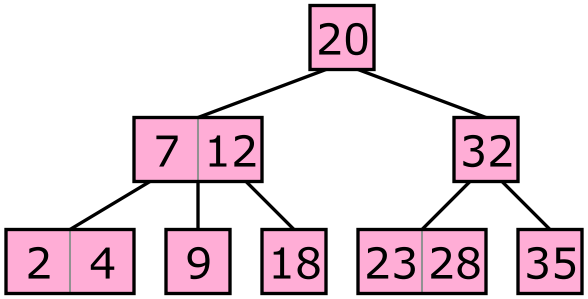
Later, we will look at "2-3-4 trees", which can also have **4-nodes**.



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.

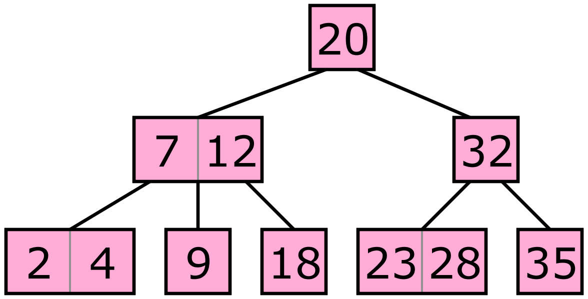
- All nodes contain either 1 or 2 data items.
  - If 2 data items, then the first is ≤ the second.
- All leaves are at the same level.
- All non-leaves are either *2-nodes* or *3-nodes*.
  - They must have the associated order properties.



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?

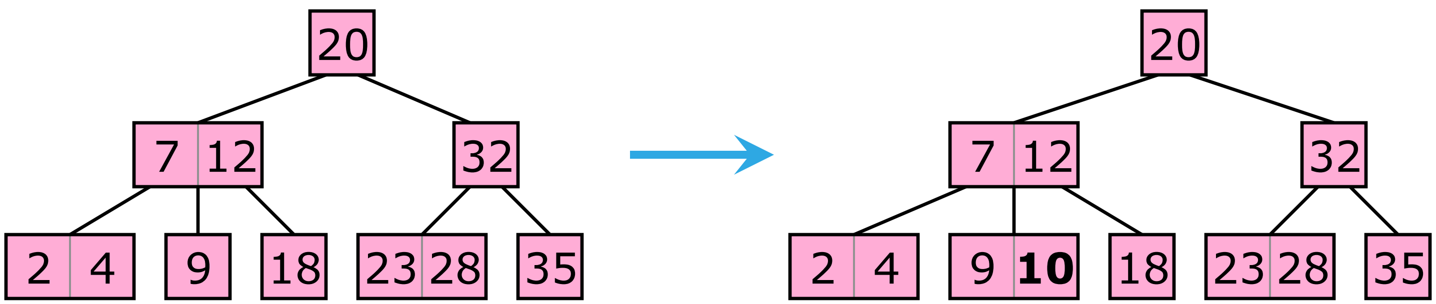
- These are the tough problems.
- It turns out that both have efficient [$O(\log n)$] algorithms, which is why we like 2-3 Trees.

2-3 Trees
Operations — Insert [1/4]

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.

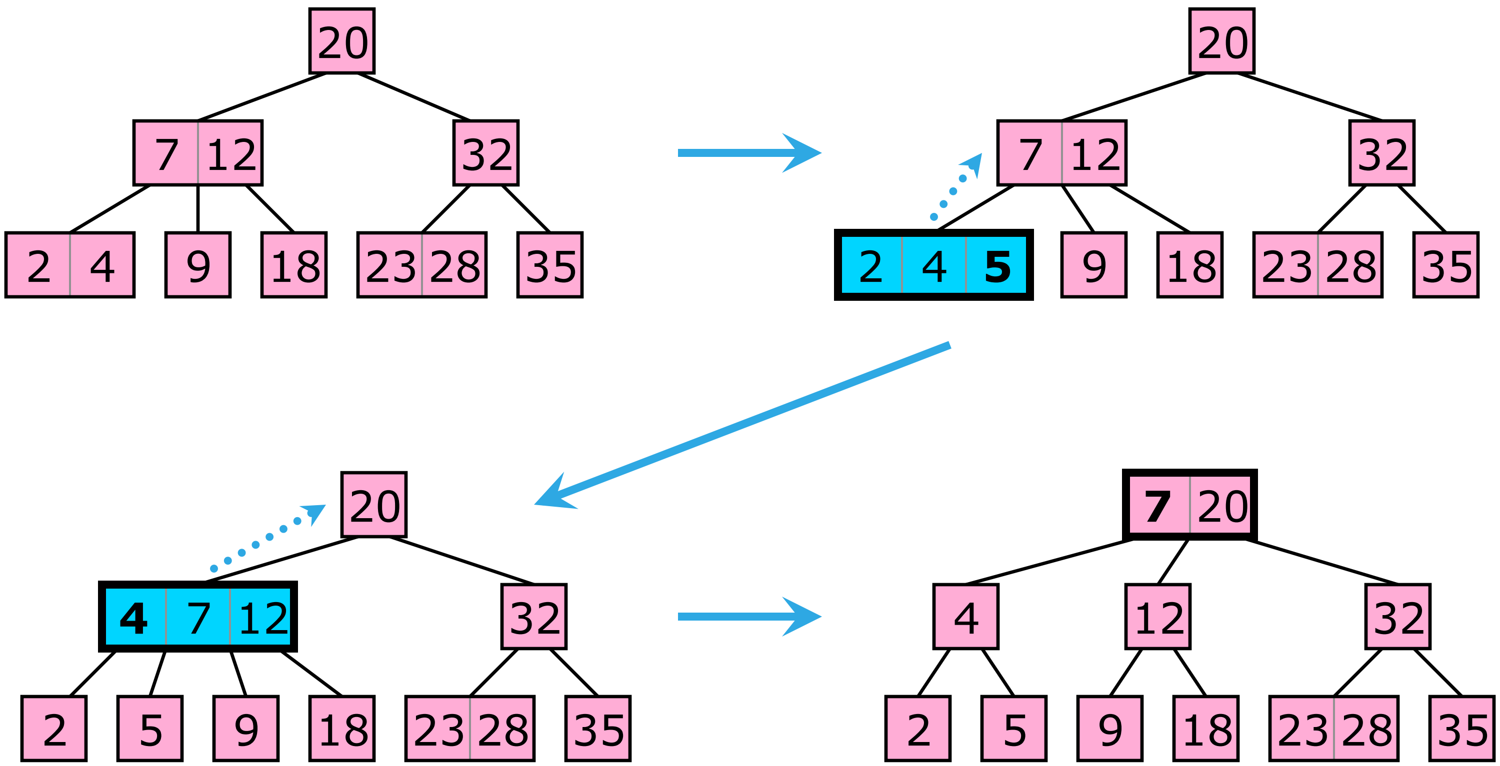
Example 1: Insert 10.



2-3 Trees
Operations — Insert [2/4]

Example 2: Insert 5.

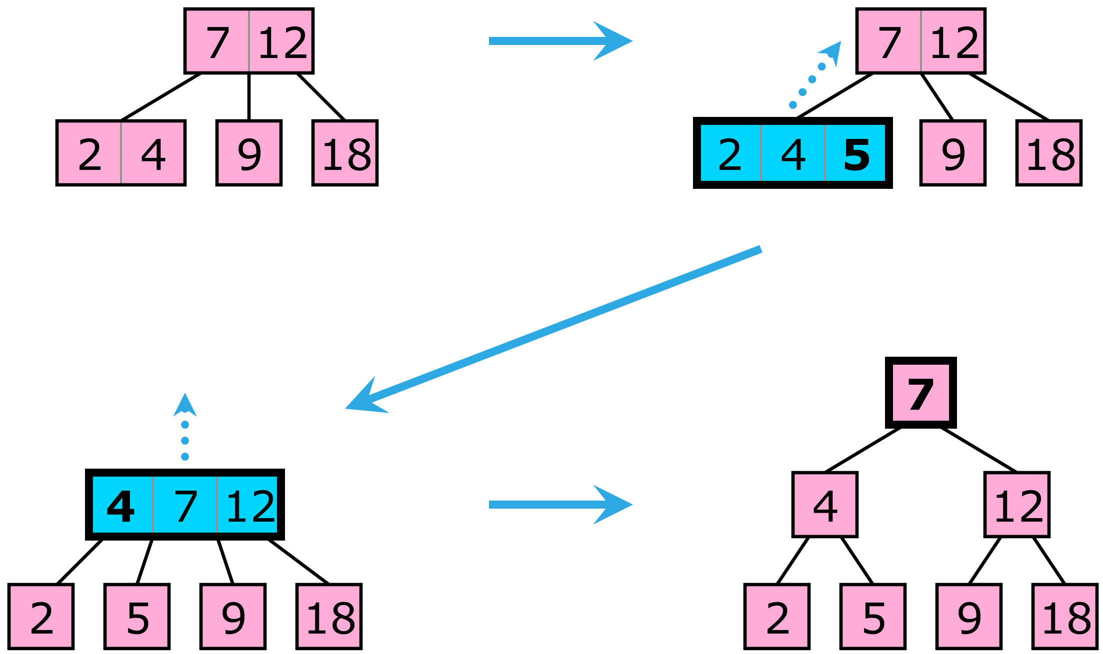
- Over-full nodes are blue.



2-3 Trees
Operations — Insert [3/4]

Example 3: Insert 5.

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



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.