Priority Queues
Binary Heap Algorithms

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, November 8, 2024

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

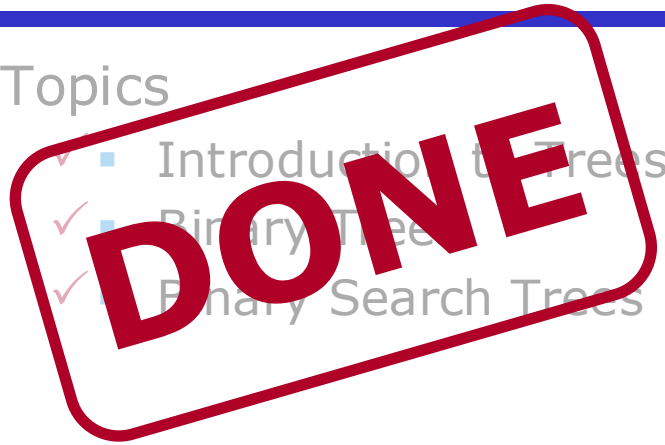
ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
The Basics of Trees

Topics

- ✓ Introduction to Trees
- ✓ Binary Trees
- ✓ Binary Search Trees



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.
- Things we need to be able to do:
  - 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.

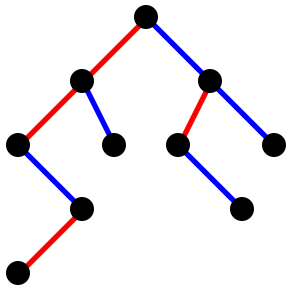
A solution to this problem is a **container**.

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

Review
Binary Trees — Definitions [1/2]

A **Binary Tree** consists of a set *T* of nodes so that either:

- *T* is **empty** (no nodes), or
- *T* consists of a node *r*, the **root**, and two subtrees of *r*, each of which is a Binary Tree:
  - the **left subtree**, and
  - the **right subtree**.

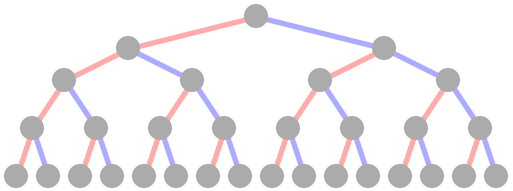


We make a strong distinction between left and right subtrees. Sometimes we use them for very different things.

Review
Binary Trees — Definitions [2/2]

Full Binary Tree

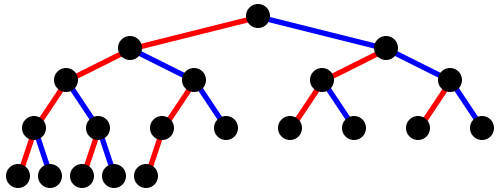
- Leaves all lie in the same level.
  All other nodes have two children each.



Complete Binary Tree

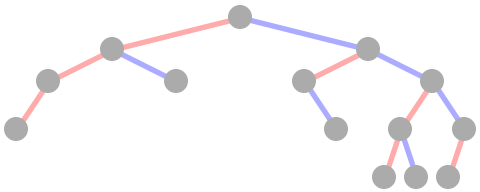
Particularly important today!

- All levels above the bottom are full.
  Bottom level is filled left-to-right.
- Importance. Nodes are added in a fixed order. Has a useful array representation.



Strongly Balanced Binary Tree

- For each node, the left and right subtrees have heights that differ by at most 1. (An empty Binary Tree has height -1.)
- Importance. Height of entire tree is small. This can allow for fast operations.



Every full Binary Tree is complete.

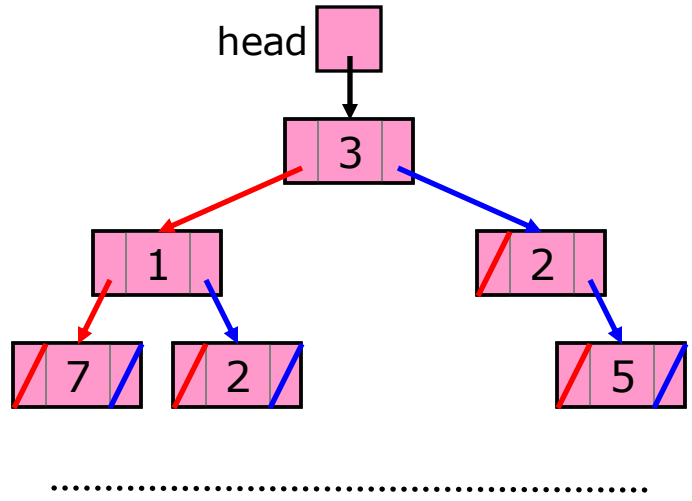
Every complete Binary Tree is strongly balanced.

All three of these concepts can be useful notions of "bushy".

Binary Trees — Implementation

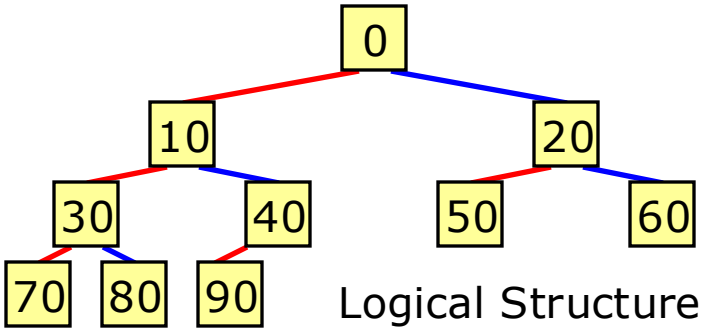
A common way to implement a Binary Tree is to use separately allocated nodes referred to by pointers.

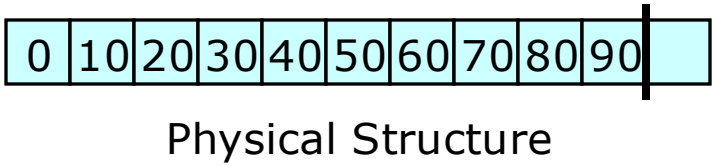
- Each node has a data item and two child pointers: left & right.
- A pointer is null if there is no child.
- There *might* also be a pointer to the parent—if that would be helpful.



A *complete* Binary Tree can be implemented by simply putting the items in an array and keeping track of the size of the tree.

This implementation is very efficient (time & space), but it is only useful when the tree will stay complete.



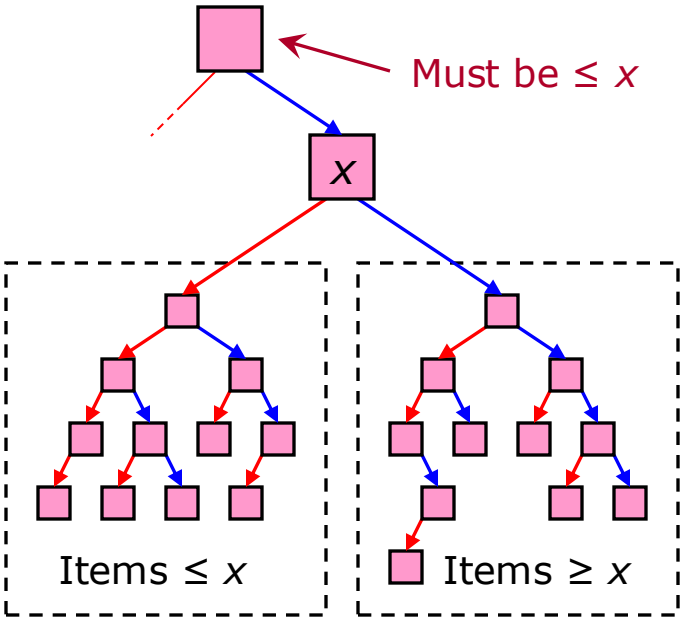


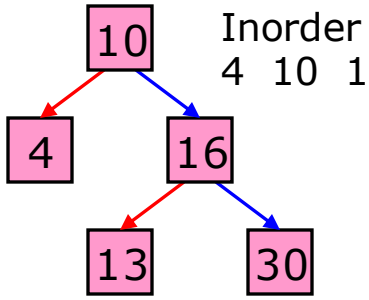
Binary Search Trees — What a Binary Search Tree Is

A **Binary Search Tree** is a Binary Tree in which each node contains a single data item, which includes a **key**, and:

- Descendants holding keys less than the node's are in its left subtree.
- Descendants holding keys greater than the node's are in its right subtree.

In other words, an inorder traversal gives keys in sorted order.





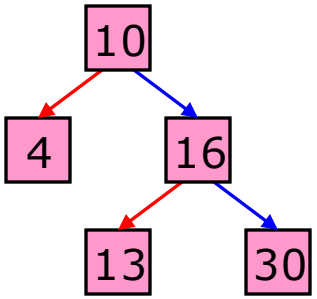
Inorder traversal:
4 10 13 16 30

Binary Search Trees — Operations

Algorithms for the BST operations:

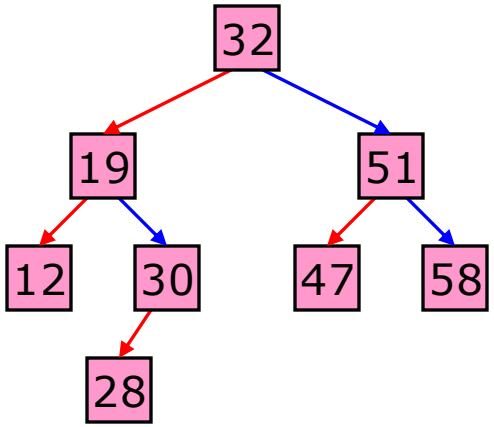
- Traverse
  - Recursively traverse left subtree of root.
  - Visit the root.
  - Recursively traverse right subtree of root.

  } Inorder Traversal
- Retrieve
  - **Search**. Start at the root. Go down, left or right as appropriate, until either the given key or an empty spot is found.
- Insert
  - *Search*, then …
  - Put the value in the spot where it should go.
- Delete
  - *Search*, then …
  - Check the number of children the node has:
    - 0. Delete node.
    - 1. Replace subtree rooted at node with subtree rooted at child.
    - 2. Copy data from (or swap data with) inorder successor. Proceed as above.



Binary Search Trees — Operations: Try It! [1/2]

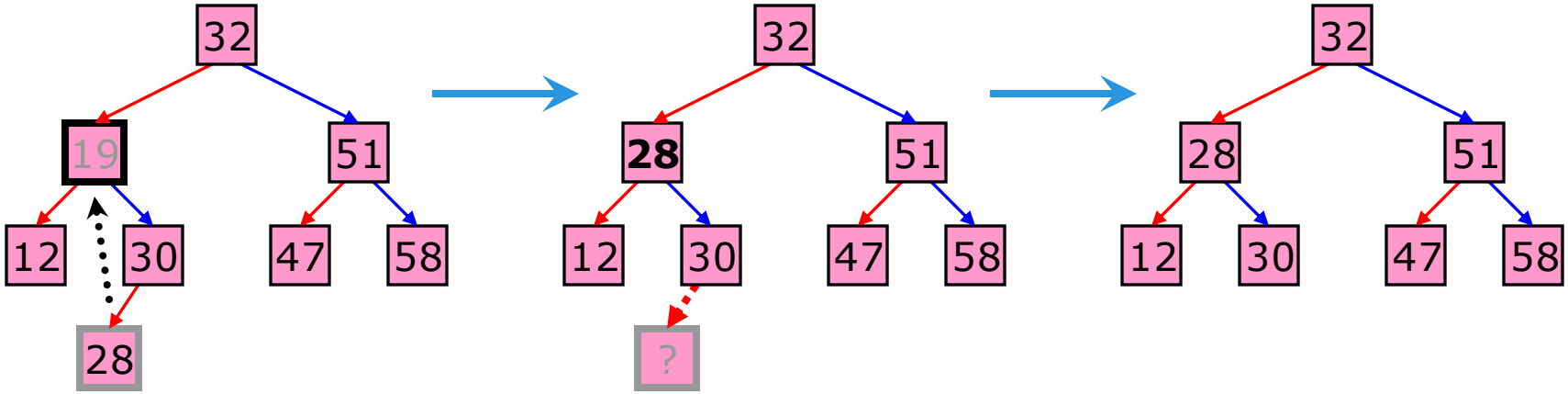
Do delete key 19 on the Binary Search Tree shown. Draw the resulting tree.



Answer on next slide.

Binary Search Trees — Operations: Try It! [2/2]

Do delete key 19 on the Binary Search Tree shown. Draw the resulting tree.



Procedure

- The 19 node is 2-children case.
- Find the inorder successor: the 28 node.
- Copy/swap 28 to the 19 node.
- Delete the old 28 node. This is 0-child case.
- So just remove the old 28 node.

Review
Binary Search Trees — Efficiency

The efficiency of BST operations depends on the tree's height.

A strongly balanced Binary Search Tree has logarithmic height, but the insert & delete operations do not keep the height small.

	BST: strongly balanced OR average case	**Sorted Array**	**BST: worst case**
Retrieve	Logarithmic	Logarithmic	Linear
Insert	Logarithmic	Linear	Linear
Delete	Logarithmic	Linear	Linear

Because we do not (yet?) know how to keep a tree strongly balanced, this is not (yet?) practical.

So Binary Search Trees have poor worst-case performance.

But they have good performance:

- On average, for random data.
- If strongly balanced. But if we allow insert/delete operations, then we need an efficient way to make a tree *stay* strongly balanced.

Can we efficiently *keep* a Binary Search Tree strongly balanced?

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
Introduction to Tables [1/4]

Our ultimate value-oriented ADT is **Table**.

Three primary single-item operations:

- **Retrieve** (by key).
- **Insert** (item—commonly a key-value pair).
- **Delete** (by key).

Table

Key	Value
12	Ed
4	Peg
9	Ann

What do we use a Table for?

- Data accessed by a **key** field.
  - For example, any kind of data that we look up using an ID.
- **Set** data.
  - Each item has only a key, with no associated value.
  - Fundamentally, the only question is which keys lie in the dataset.
- Array-like datasets whose indices are not nonnegative integers.
  - `arr2["hello"] = 3;`
- Array-like datasets that are **sparse** (or not sparse).
  - `arr[6] = 1; arr[1000000000] = 2;`

Review
Introduction to Tables [2/4]

What are possible Table implementations?

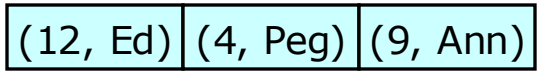
- A Sequence holding key-value pairs.
  - Array-based or Linked-List-based.
  - Sorted or unsorted.
- A Binary Search Tree holding key-value pairs.
  - Implemented using a pointer-based Binary Tree.

Table

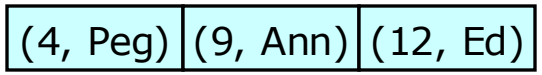
Key	Value
12	Ed
4	Peg
9	Ann

Array Implementations

Unsorted

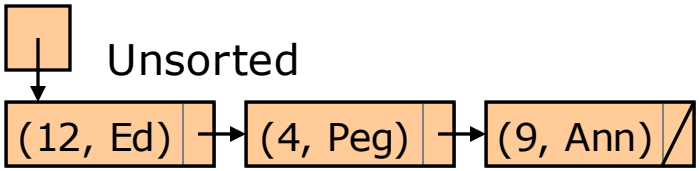


Sorted by Key

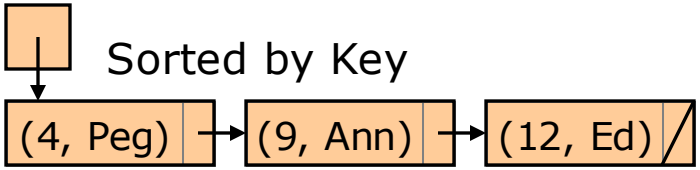


Linked List Implementations

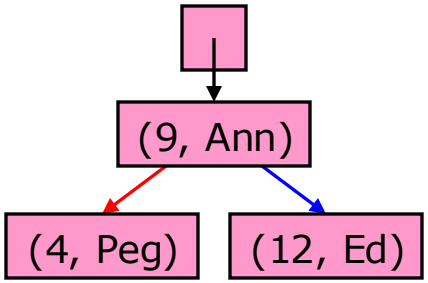
Unsorted



Sorted by Key



Binary Search Tree Implementation



How efficient are these Table implementations?

Review
Introduction to Tables [3/4]

Q. How efficient are these Table implementations?

A. Not very efficient at all.

In particular, the ones we know how to do all have a linear-time *delete* operation.

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	*Strongly Balanced* BST?*
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	*Logarithmic*
Insert	Linear	Linear/ amortized constant**	Linear	Constant	Linear	*Logarithmic*
Delete	Linear	Linear	Linear	Linear	Linear	*Logarithmic*

Above, we allow multiple equivalent keys.

*We do not (yet?) know how to ensure that the tree will *stay* strongly balanced, unless we restrict ourselves to read-only operations (no insert, delete).

**Constant time if we have pre-allocated enough storage.

Introduction to Tables [4/4]

All the Table implementations we have thought of are inefficient. What can we do about this? Here are three ideas.

Idea #1: Restricted Tables

- Only allow retrieve/delete on the greatest key.
- In practice: Priority Queues

Idea #2: Keep a tree balanced

- In practice: Self-balancing search trees (2-3 Trees, etc.)

Idea #3: Magic functions

- Use an unsorted array. Each item can be a key-value pair or empty.
- A magic function tells the index where a given key is stored.
- Retrieve/insert/delete in constant time? No, but still a useful idea.
- In practice: Hash Tables

Unit Overview
Tables & Priority Queues

Major 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

Priority Queues

What a Priority Queue Is — Introduction

Now we look at the first of our three ideas: restricting the Table operations, in order to gain efficiency.

Our restricted-Table ADT is called **Priority Queue**.

- This has almost the same operations as Queue.
- The difference is that the item that is retrieved or deleted is the one with the greatest key.
- So items are not removed in the order they were inserted, but rather in order of priority: greatest key to least key.

Despite the name, a Priority Queue is *not* a Queue!

Priority Queues
What a Priority Queue Is — ADT

Priority Queue has the following data and operations.

Data

- A collection of items, each of which has a key.

Operations

- **getFront**. Look at item with greatest key.
- **insert**. Add a given item.
- **delete**. Remove item with greatest key.
- And the usual:
  - **create**, **destroy**, **copy**.
  - **isEmpty**.
  - **size**.

Three single-item operations, yet again.

So we can insert anything, but there is only one key that we can retrieve (getFront) or delete. Thus, Priority Queue is a restricted Table, just as Stack & Queue are restricted Sequences.

Priority Queues
Applications

A Priority Queue is useful when we have items to process, and some are *better* (lower cost?) or *more urgent* than others.

Near the end of the semester, we will look at an application of a Priority Queue, as part of a method to find a *minimum spanning tree* in a graph. This involves prioritizing lower-cost items.

A Priority Queue can be used to sort: insert all items, then getFront/delete all items. The items are removed from greatest key to least key.

This leads to a fast sort called *Heap Sort* (recall: the usual fall-back algorithm in Introsort).

A Priority Queue can also do variations on sorting.

- To find the *k* greatest items in a list: insert them all into a Priority Queue, then do getFront/delete *k* times.
- A Priority Queue can hold that a dataset is modified during sorting.

Priority Queues
Implementation

We can implement a Priority Queue using any of the methods we have discussed for implementing a Table.

And they are all still just as dissatisfying.

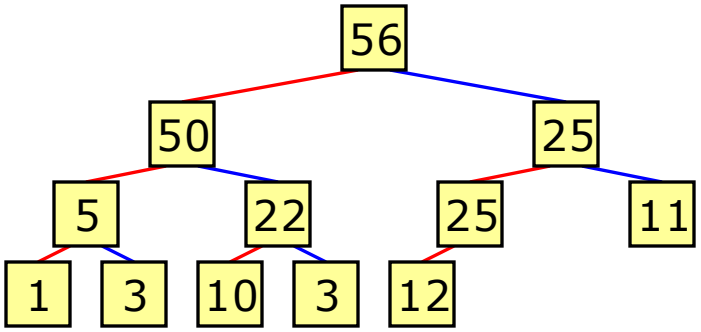
The most interesting thing about Priority Queues is their most common implementation: a structure called a *Binary Heap*. We discuss this next.

Binary Heap Algorithms

Binary Heap Algorithms
What a Binary Heap Is — Definition

A **Binary Heap** (or just **Heap**) is a complete Binary Tree in which

- each node contains a single data item, which includes a **key**, and
- each node's key is ≥ the keys in its children, if any.



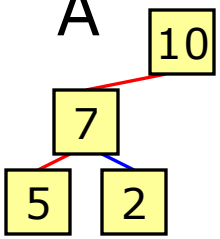
Notes

- There are no required order relationships between siblings.
- The above is a **Maxheap**. If we reverse the order, so that a node's key is ≤ the keys in its children, then we get a **Minheap**, which works just the same in all other ways.
- You may see another meaning of "heap": the area of memory used for dynamic allocation. This usage is unrelated to that above.

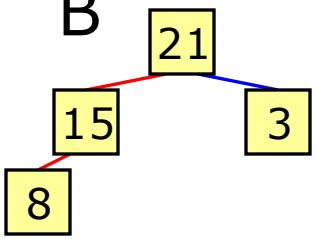
Binary Heap Algorithms
What a Binary Heap Is — Try It! [1/2]

Which of the following are Binary Heaps?

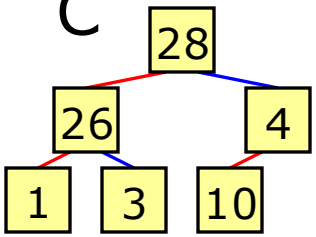
A



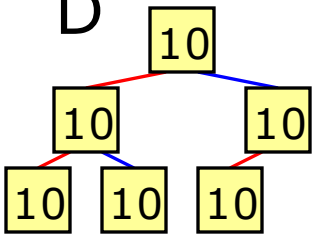
B



C



D

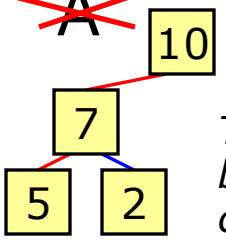


Answers on next slide.

Which of the following are Binary Heaps?

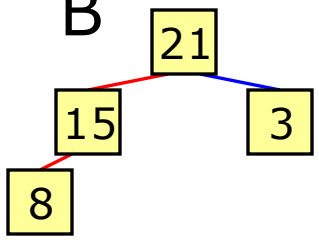
Answers

~~A~~

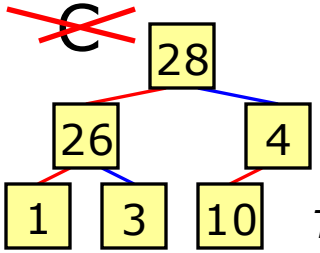


This is a Binary Tree, but it is not a complete Binary Tree.

✓ B

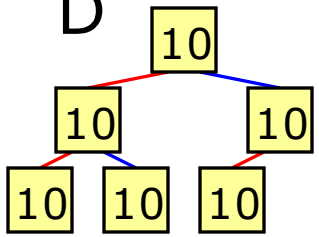


~~C~~



This is a complete Binary Tree, but it does not have the Heap order property, since 4 < 10.

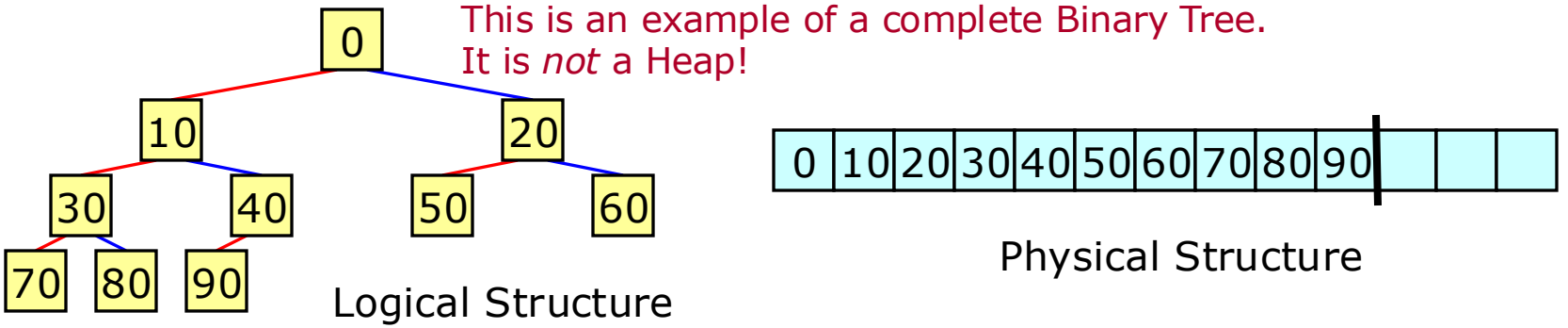
✓ D



Binary Heap Algorithms
What a Binary Heap Is — Refresher: Complete Binary Trees

Recall the array implementation of a **complete Binary Tree**:

- Put the nodes in an array, in the order in which they would be added to a complete Binary Tree.
- Store *only* the **array** of data items and the **number** of nodes.

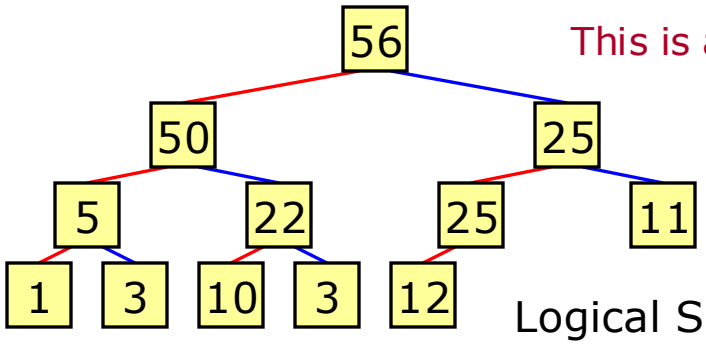


No stored pointers are required.

We can navigate around the tree (find the root, find children, find the parent, determine whether each of these exists) by doing arithmetic with array indices.

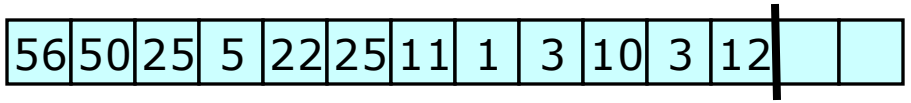
Binary Heap Algorithms
What a Binary Heap Is — Implementation

The standard implementation of a Binary Heap uses this array-based complete Binary Tree.



This is a Heap.

Logical Structure



Physical Structure

In practice, we use "Heap" to mean a Binary Heap implemented using this array representation.

In order to base a Priority Queue on a Heap, we need to know how to perform the Priority Queue operations with a Heap.

Operation *getFront* is easy: look at the root item.

Next we consider *insert* & *delete*.

Binary Heap Algorithms
Primary Operations — Insert

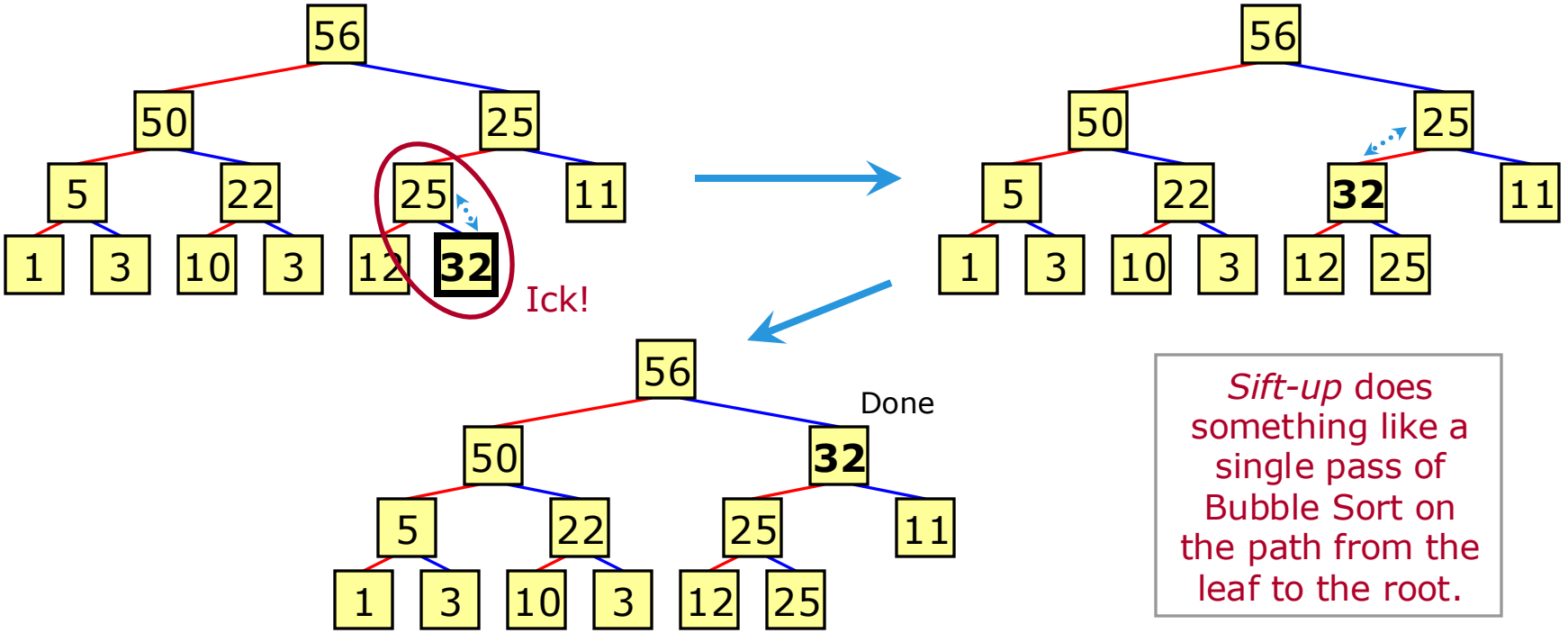
In a Priority Queue, we can insert any value. (How about 32.)

In a complete Binary Tree, we can only add a new node at the end.

So add the the new item in a new node at the end.

But now we may not have a Heap.

Solution: **sift-up**. New value greater than parent? Swap & repeat.



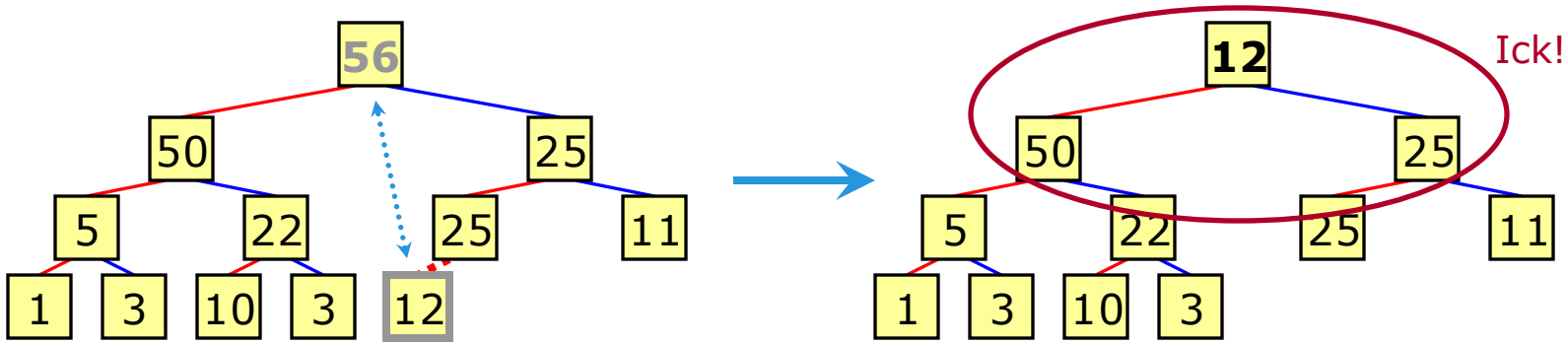
Sift-up does something like a single pass of Bubble Sort on the path from the leaf to the root.

Binary Heap Algorithms
Primary Operations — Delete [1/2]

In a Priority Queue, we can delete the item with the greatest key.

In a Maxheap, this is the root item. How do we delete the **root item**, while maintaining the Heap properties?

- We cannot delete the **root node**—unless it is the only node.
- The Heap will have one less item, so the **last node** must go away.
- But the **last item** is not going away.
- So, put the last node's item in the root node; delete the last node.
  - Do this by *swapping* items—which has other advantages, as we will see.

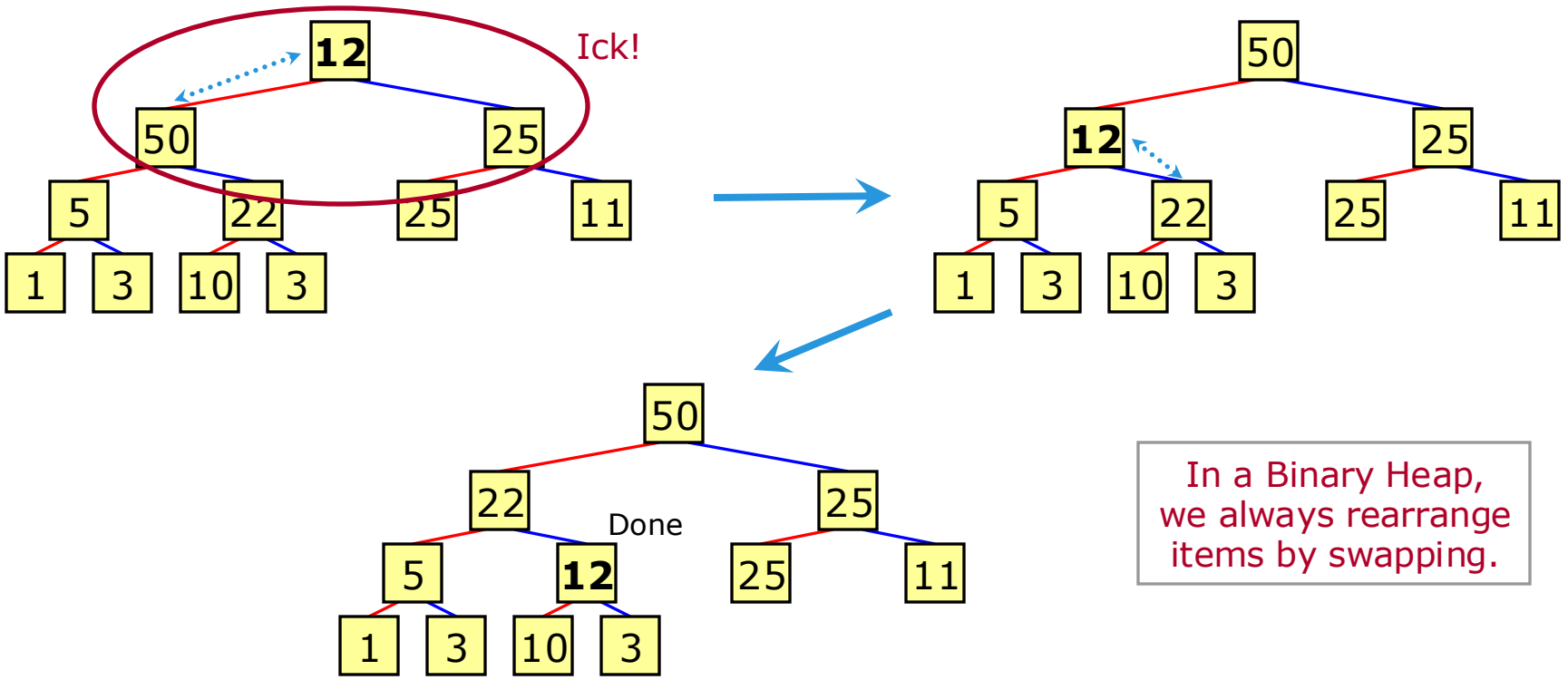


We have a problem again: this is no longer a Heap. How to fix it?

Binary Heap Algorithms
Primary Operations — Delete [2/2]

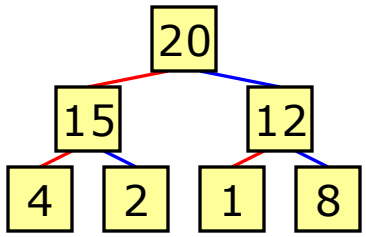
After swapping items in root & last node, and then removing the last node, we may no longer have a Heap.

Solution: **sift-down** the new root item. Smaller than a child? Swap with *larger* child & repeat.



Binary Heap Algorithms
Primary Operations — Try It! [1/2]

Do Heap delete on the Binary Heap shown below. Draw the resulting Binary Heap, as a tree.

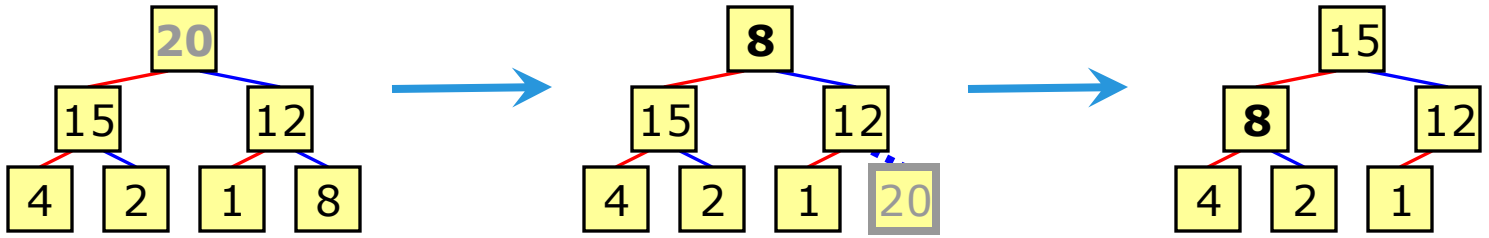


Answer on next slide.

Binary Heap Algorithms
Primary Operations — Try It! [2/2]

Do Heap delete on the Binary Heap shown below. Draw the resulting Binary Heap, as a tree.

Answer



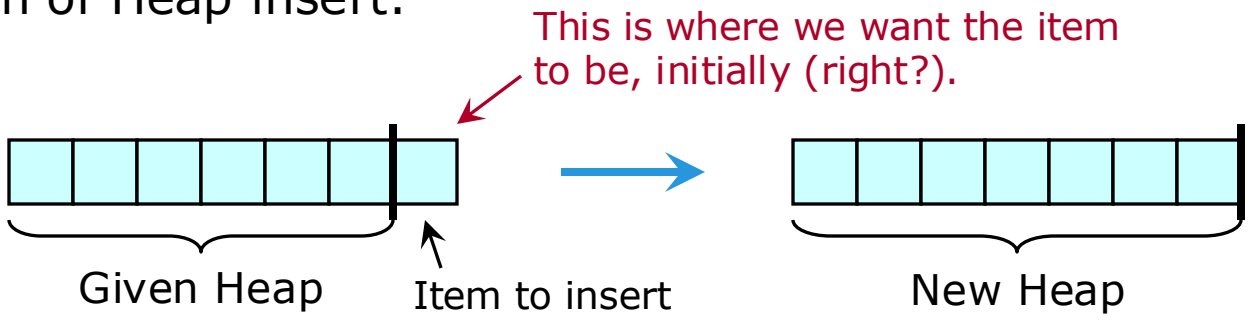
Procedure

- We are deleting the root item (20).
- Swap the root item with the item in the last node (8), and then delete the last node.
- Sift-down the new root item (8).

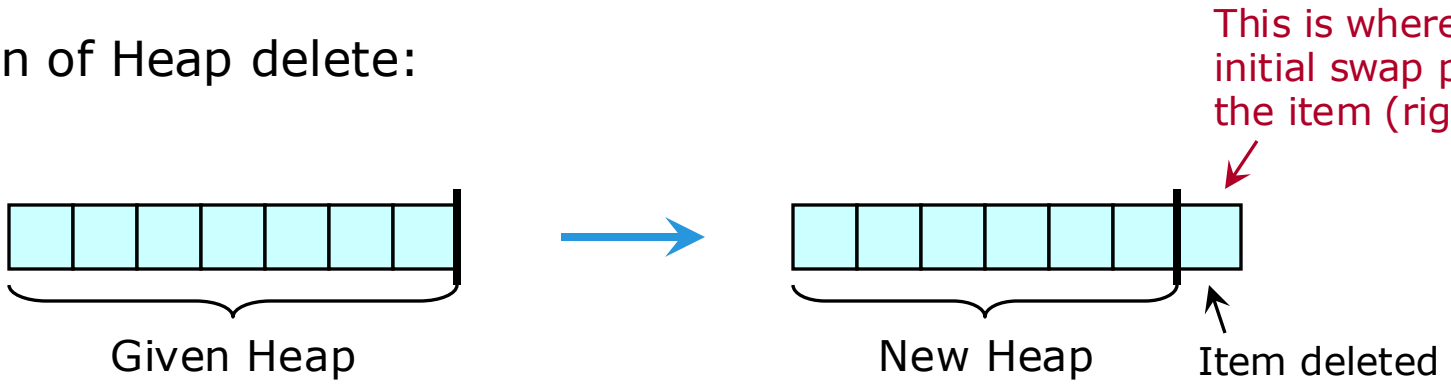
Binary Heap Algorithms
Primary Operations — Using an Array

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.

Binary Heap Algorithms
Primary Operations — Efficiency

What is the order of the Priority Queue operations, if we use a Binary Heap in the array representation?

- **getFront**
  - Constant time.
- **insert**
  - Linear time in general, due to possible reallocate-and-copy.
  - Logarithmic time, if we can be sure that no reallocation is required.
    - That is, assuming the array is large enough to hold the new item. The way that Heaps are often used guarantees that this is the case.
    - The number of operations is roughly the height of the tree. Since the tree is strongly balanced, the height is $O(\log n)$.
- **delete**
  - Logarithmic time.
  - There is no reallocation. See the comment on height under *insert*.

Better than linear time!

We have not seen this before, for a delete by key.

So a Heap is a good basis for implementing a Priority Queue.

Binary Heap Algorithms
Primary Operations — CODE

TO DO

- Write the Heap insert algorithm.
  - Prototype is shown below.
  - The item to be inserted is the final item in the given range.
  - All other items should form a Heap already.
- Write other Heap algorithms as time permits.

```
// Requirements on types:
//     RAIter is a random-access iterator type.
template<typename RAIter>
void heapInsert(RAIter first, RAIter last);
```

Done. See `heap_algs.hpp`*.*
The other Heap algorithms have also been written.

See `heap_algs_main.cpp` *for a program that uses this header.*

Binary Heap Algorithms
TO BE CONTINUED …

Binary Heap Algorithms will be continued next time.