

Binary Search Tree Operations (cont.)

TreeSort

Introduction to Tables

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, April 12, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell
© 2005–2009 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.

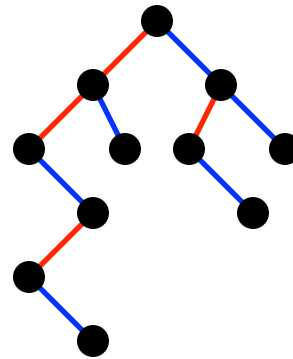
A solution to this problem is a **container**.

Generic containers: those in which client code can specify the type of data stored.

Binary Trees — What a Binary Tree Is

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**.
- 
- ```
graph TD; A(()) --- B(()); A --- C(()); B --- D(()); B --- E(()); C --- F(()); C --- G(())
```



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

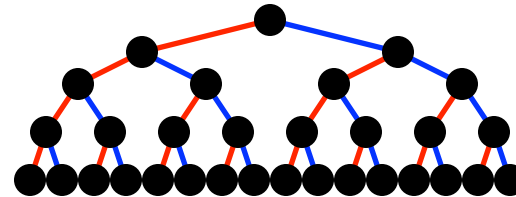


## Review

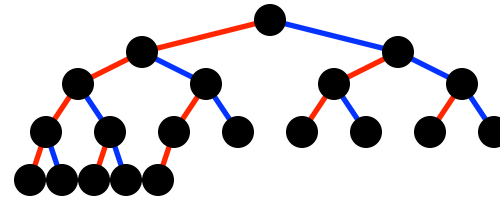
### Binary Trees — Three Special Kinds

---

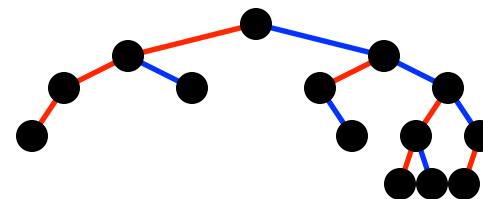
A typical **full** Binary Tree:



A typical **complete** Binary Tree:



A typical **balanced** Binary Tree:



A full Binary Tree is complete; a complete Binary Tree is balanced.

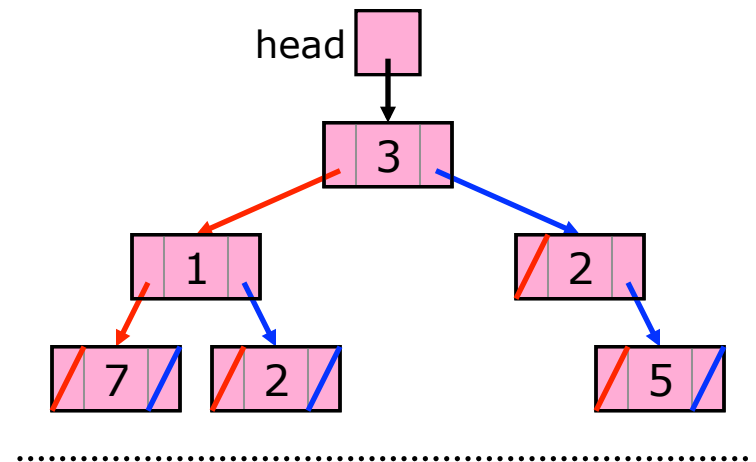
Full → Complete → Balanced

## Review

### Binary Trees — Implementation

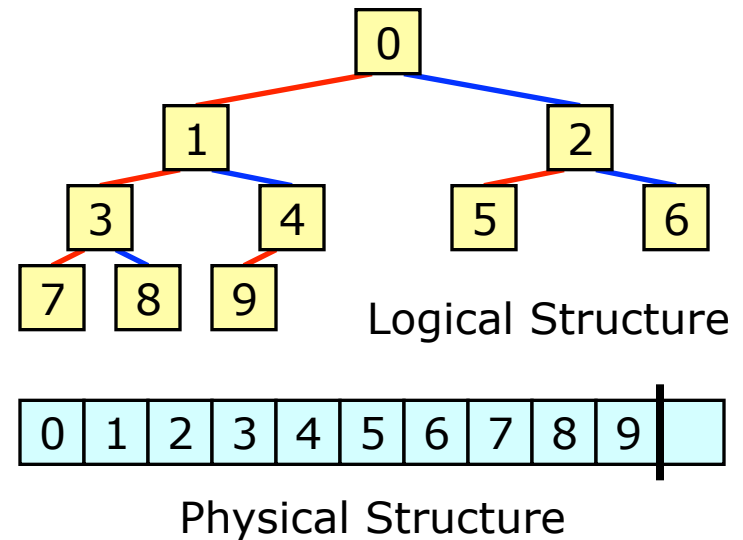
A common way to implement a Binary Tree is to use separately allocated nodes referred to by pointers.

- This is very similar to the way we implemented a Linked List.
- Each node has a data item and two child pointers: left & right.
- A pointer is null if there is no child.



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

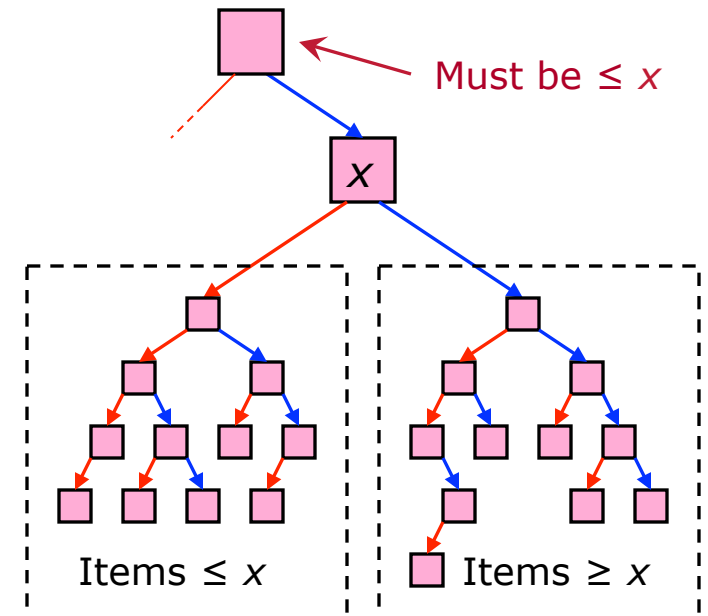
- This is a nice implementation, but it is only useful in situations in which the tree *stays complete*.



## Review

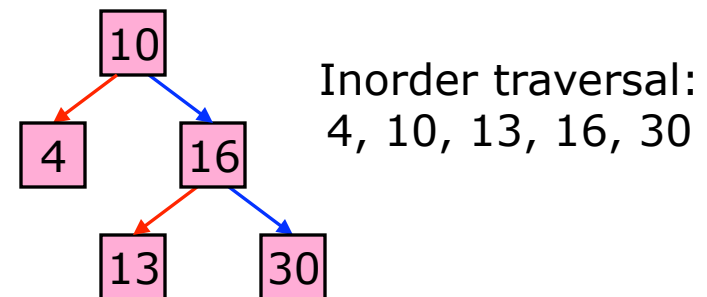
### Binary Search Trees — What a Binary Search Tree Is

- A **Binary Search Tree** is a Binary Tree in which, for each node:
- Descendants holding data less than the node's data are in its left subtree.
  - Descendants holding data greater than the node's data are in its right subtree.



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

This is another value-oriented ADT (while ADT Binary Tree is position-oriented).



# Review: Binary Search Trees

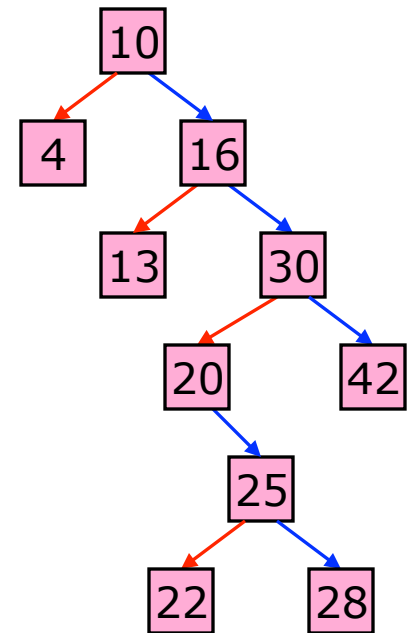
## Operations

---

### ADT Binary Search Tree

- Data
  - A collection of nodes, each with a data item.
- Operations
  - **create** empty B.S.T.
  - **destroy**.
  - **isEmpty**.
  - **insert** (given item [which includes a key]).
  - **delete** (given key).
  - **retrieve** (given key, returns item).
  - The three traversals: **preorderTraverse**, **inorderTraverse**, **postorderTraverse**.

Here  
they are  
again



## Binary Search Trees

### Review: Operations — Retrieve

---

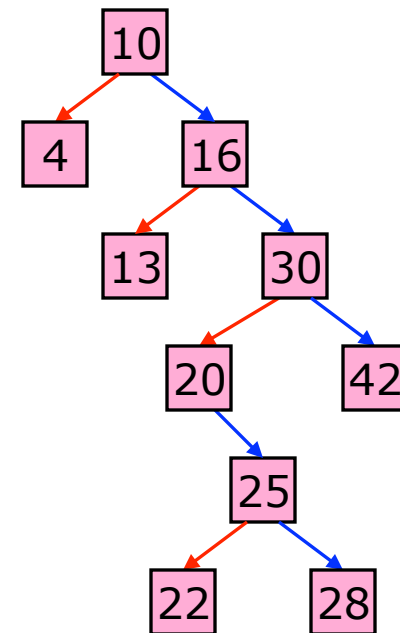
To **retrieve** a value from a Binary Search Tree, we begin at the root and repeatedly follow left or right child pointers, depending on how the search key compares to the key in each node.

For example, search for the key 20 in the tree shown:

- $20 > 10 \rightarrow$  right.
- $20 > 16 \rightarrow$  right.
- $20 < 30 \rightarrow$  left.
- $20 = 20 \rightarrow$  FOUND.

When searching for a key that is not in the tree, we stop when we find where the key “should” be. Search for the key 18 in the same tree:

- $18 > 10 \rightarrow$  right.
- $18 > 16 \rightarrow$  right.
- $18 < 30 \rightarrow$  left.
- $18 < 20 \rightarrow$  left.
- No left child  $\rightarrow$  NOT FOUND.





## Binary Search Trees

### Review: Operations — Insert

---

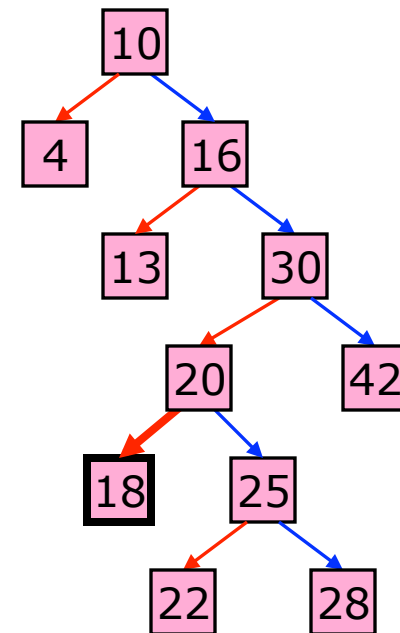
To **insert** a value with a given key, we find where the key “should” go (**retrieve** operation), and then we put the value there.

- For example, we just found where 18 should go.

If our key turns out to be **in the tree already**, then our action depends on the specification of the Binary Search Tree.

- We may **add a new value** having the same key.
  - Result: multiple equivalent keys.
- Or we may **replace** the value with the given key.
- Or we may leave the tree **unchanged**.
  - If the last option is taken, we may wish to signal an **error condition**.

Note: Not just for Binary Search Trees!  
These are always the options, when we insert a duplicate key into a data set.



## Binary Search Trees

### Operations — Delete [1/3]

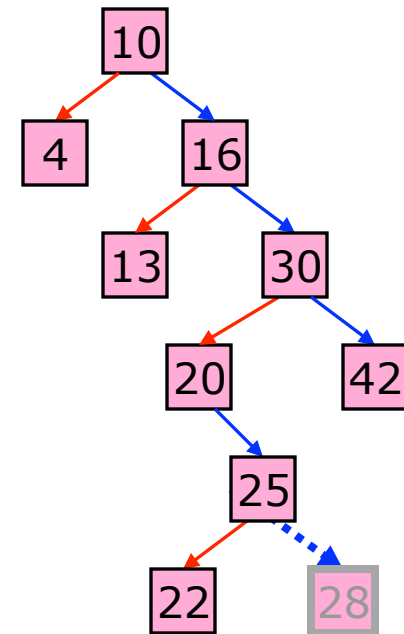
---

To **delete** an item from a Binary Search Tree, given its key:

- Find the node (**retrieve** operation).
- Then, three cases:
  - The node to be deleted has no children (it is a leaf).
  - The node has 1 child.
  - The node has 2 children.

**No-child** (leaf) case:

- Delete the node, using the appropriate **Binary Tree** operation.



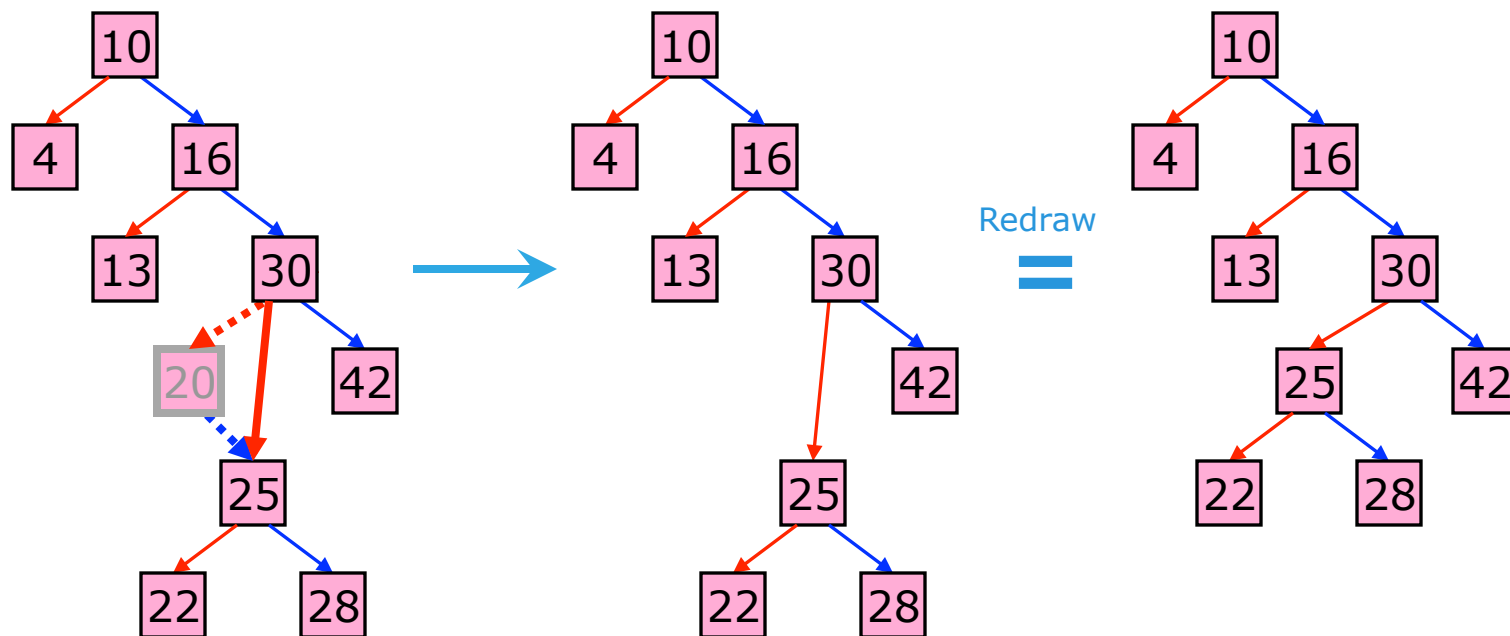
# Binary Search Trees

## Operations — Delete [2/3]

---

### One-child case:

- Replace the subtree rooted at the node with the subtree rooted at its child.
  - This is a constant-time operation, once the node is found.



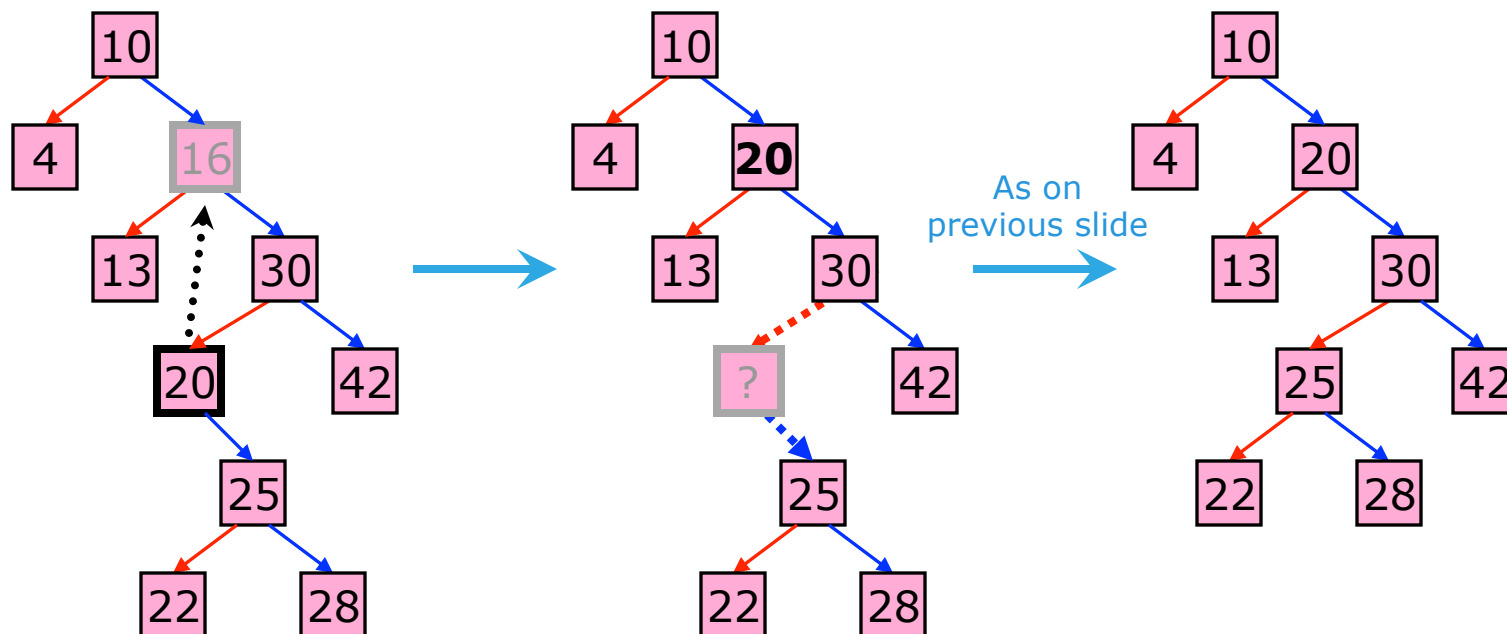
# Binary Search Trees

## Operations — Delete [3/3]

---

### Two-child case:

- Replace the data in the node with the data in its **inorder successor** (copy or swap).
- Delete the inorder successor.
  - This has at most one child.



## Binary Search Trees

### Operations — Summary & Thoughts

---

Algorithms for the three primary B.S.T. operations:

- Retrieve
  - Start at root. Go down, left or right as appropriate, until either the given key or an empty spot is found.
- Insert
  - Retrieve, then ...
  - Put the value in the spot where it should be.
- Delete
  - Retrieve, then ...
  - Check number of children 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.

All three operations, in the worst case, require a number of steps proportional to the **height of the tree**.

The height of a tree is small (and all three operations are fast) if the tree is **balanced**.

## Binary Search Trees

### Efficiency — Height of a Balanced Binary Tree [1/3]

---

B.S.T. insert, delete, and retrieve follow links down from the root.

- The number of steps required is usually something like the height of the tree.
- In the worst case, the height of a tree is its size. But what about when a tree is balanced?
- So: Given the **size** of a balanced Binary Tree, how **large** can its **height** be?

In order to answer this, we look at the “reverse” question: Given the **height** of a balanced Binary Tree, how **small** can its **size** be? That is, what is the minimum size of a balanced Binary Tree with height  $h$ ?

Answer (apparently):  $F_{h+2} - 1$ , for  $h = 0, 1, 2, \dots$

- $F_k$  is Fibonacci number  $k$ .  $F_0 = 0$ ,  $F_1 = 1$ ,  $F_2 = 1$ ,  $F_3 = 2$ , etc.
- It is not too hard to prove this, using mathematical induction.

## Binary Search Trees

### Efficiency — Height of a Balanced Binary Tree [3/3]

---

Back to the original question: Given the size of a balanced Binary Tree, how large can its height be?

- We know that, if we have a balanced Binary Tree with height  $h$  and size  $n$ , then  $n \geq F_{h+2} - 1$ .
- Fact: Let  $\varphi = \frac{1+\sqrt{5}}{2}$ . Then  $F_k \approx \frac{\varphi^k}{\sqrt{5}}$ . (Remember `fibonacci5.cpp`?)
- Thus, roughly:  $n \geq \frac{\varphi^{h+2}}{\sqrt{5}}$ .
- Solving for  $h$ , we obtain, roughly:  $h \leq \log_{\varphi}(\sqrt{5}n) - 2$ .
- We conclude that, for a balanced Binary Tree,  **$h$  is  $O(\log n)$** .

Even better, the height of a Binary Search Tree is, with high probability,  $O(\log n)$  for **random data**.

- We will not verify this statement.

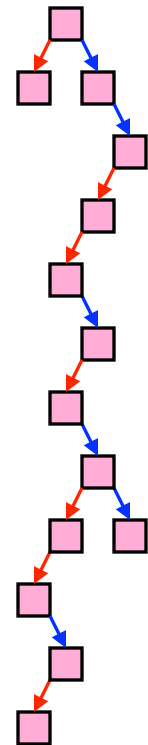
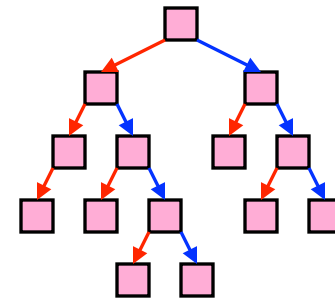
# Binary Search Trees

## Efficiency — Order of Operations

---

How efficient are the B.S.T. operations (using the standard implementation)?

- isEmpty
  - Constant time.
- Retrieve
  - Linear time.
  - Worst case is roughly the height.
    - If balanced: logarithmic time. But it might not **stay** balanced.
    - Logarithmic time on average for random data.
  - Retrieve does not modify the tree. It may be worth the time to create a balanced tree beforehand.
- Insert
  - Linear time.
  - See second point under “Retrieve”.
- Delete
  - Linear time.
  - See second point under “Retrieve”.
- Pre-, in-, postorder traversal
  - Linear time.





## Binary Search Trees

### Efficiency — The Problem

---

A B.S.T. has a nice interface for sets and key-based look-up.

Further, a B.S.T. has good average performance; **retrieve**, **insert**, and **delete** are logarithmic time for typical data.

But in the worst case, a B.S.T. is worse than a sorted array.

- It is also worse in memory usage.

|          | B.S.T.<br>(balanced &<br>average case) | Sorted Array | B.S.T.<br>(worst case) |
|----------|----------------------------------------|--------------|------------------------|
| Retrieve | Logarithmic                            | Logarithmic  | Linear                 |
| Insert   | Logarithmic                            | Linear       | Linear                 |
| Delete   | Logarithmic                            | Linear       | Linear                 |

Can we efficiently *keep* a Binary Search Tree balanced, while allowing for insert & delete operations?

- We will look at this question again later.

## Treesort

### Introduction

---

For most sorted containers, there is an associated sorting algorithm.

- Insert all items into the container, and then iterate through it.
- For a sorted array, this algorithm is pretty nearly Insertion Sort.
  - It would be a non-in-place version of Insertion Sort.
- For a Binary Search Tree, the algorithm is called **Treesort**.
  - Note: We must allow equivalent items in our B.S.T.

Treesort is not a very good algorithm, but it is worth looking at.

What is the order of Treesort?

- $O(n^2)$ .
  - There are  $n$  insert operations, each of which is  $O(n)$ , plus a single  $O(n)$  traversal.
- However, it is *usually* pretty fast:  $O(n \log n)$  on average.
  - Because B.S.T. Insert is  $O(\log n)$  for average data.

Have we seen Treesort before?

- Kind of. It is basically **Quicksort** in disguise.
- The main practical difference is that Treesort requires a large auxiliary data structure.

# Treesort Analysis

---

## Efficiency ☹️

- Treesort is  $O(n^2)$ .
- Treesort has an acceptable average-case time:  $O(n \log n)$ . 😊

## Requirements on Data 😊?\*

- Treesort does not require random-access data.
- It works with Linked Lists.

## Space Usage ☹️

- Treesort requires  $O(n)$  additional space for the tree.
  - And this space holds data items.

## Stability 😊

- Treesort is stable.
  - Even though Quicksort is not. Do you see why?

## Performance on Nearly Sorted Data ☹️

- Treesort is **slow** on nearly sorted data:  $O(n^2)$ .
  - Just like unoptimized Quicksort.

\*This is not much of an advantage for an algorithm that is inefficient in both time and space. (Suppose it did require random-access data. To fix this, we could simply start by copying to an array.)

## Unit Overview

### Tables & Priority Queues

---

Next we begin a unit on ADTs Table and Priority Queue and their implementations (in particular, Binary Heaps and the associated algorithms, balanced search trees, and Hash Tables).

#### Major Topics

- Introduction to Tables
- Priority Queues
- Binary Heap algorithms
- Heaps & Priority Queues in the C++ STL
- 2-3 Trees
- Other balanced search trees
- Hash Tables
- Prefix Trees
- Tables in various languages

This will be the last *big* unit in the class. After this, if time permits, we will look briefly at:

- External methods
- Graph algorithms

## Binary Search Trees — Efficiency

---

|          | B.S.T.<br>(balanced &<br>average case) | Sorted Array | B.S.T.<br>(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

---

Next we begin a unit on ADTs Table and Priority Queue and their implementations (in particular, Binary Heaps and the associated algorithms, balanced search trees, and Hash Tables).

#### Major Topics

- Introduction to Tables
- Priority Queues
- Binary Heap algorithms
- Heaps & Priority Queues in the C++ STL
- 2-3 Trees
- Other balanced search trees
- Hash Tables
- Prefix Trees
- Tables in various languages

This will be the last *big* unit in the class. After this, if time permits we will look briefly at:

- External methods
- Graph algorithms

# Introduction to Tables

## Types of ADTs

---

### **Position-Oriented ADTs**

- Get an item based on where it is stored.
- Organize data according to where the client wants it.

#### Examples

- Sequence
- Stack
- Queue
- Binary Tree

### **Value-Oriented ADTs**

- Get an item based on its value.
  - Or part of the value: key-based look-up.
- Organize data for greatest efficiency.

#### Examples

- SortedSequence
- Binary Search Tree

Since the client code does not need to know how the data are organized, but only needs efficiency, maybe we can do better here?

## Introduction to Tables Databases

---

A value-oriented ADT can be thought of as an interface to a general database.

Consider the four data-manipulation commands in the Structured Query Language (SQL):

- **Select**
  - Retrieve a record. Key-based look-up.
- **Update**
  - Change a record.
  - A redundant operation, since we can always delete and then insert. Alternatively, have Select return a reference (or iterator or whatever).
- **Insert**
  - Given a record, insert it.
- **Delete**
  - Given a key, delete the associated record.

These are essentially the operations in a value-oriented ADT.

- We want an implementation that makes them efficient.



## Introduction to Tables

### Operations — Possibilities

---

A general value-oriented ADT is called “Table”.

What operations should Table have?

- The Usual
  - **create, destroy, copy.**
  - **isEmpty.**
  - **size** (maybe).
- Data Manipulation
  - **retrieve** (like SQL Select).
  - *Maybe* **set** (like SQL Update).
    - We generally handle this either by having retrieve return a value in modifiable form, or by simply using delete, then insert.
  - **insert.**
  - **delete.**
- Access All Data
  - **traverse.**

## Introduction to Tables

### Operations — Issues

---

Allow multiple items with **equivalent keys**?

- It depends on the requirements of the client.

Require **traverse** to list items in sorted order?

- There are sorted & unsorted implementations. Requiring a sorted traverse would make the latter inefficient.

Allow modification of data while it is in the Table?

- If we have key-data pairs, then modifying the **data** part is no problem.
- Modifying the **key** is tricky, since the item is generally located according to its key. Changing the key means we have to move the item.

Have a separate interface in which the key is the entire value?

- Maybe. Call it “Set”.

Conclusion

- There is no single, best interface to a Table. But they are all very similar.
- Therefore, we will be a little vague about *exactly* what a Table is.

## Introduction to Tables Applications

---

What do we use a Table for?

- To hold data accessed by key fields. For example:
  - Customers accessed by phone number.
  - Students accessed by student ID number.
  - Any other kind of data with an ID code.
- To hold “Set” data.
  - Data in which the only question we ask is whether a key lies in the data set.
- To hold “arrays” whose indices are not nonnegative integers.
  - `arr2["hello"] = 3;`
- To hold array-like data sets that are **sparse**.
  - `arr[6] = 1; arr[1000000000] = 2;`

# Introduction to Tables

## Possible Implementations [1/3]

---

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

Sorted

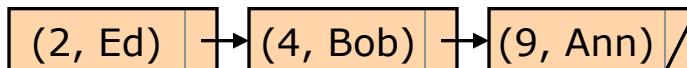
|         |          |          |
|---------|----------|----------|
| (2, Ed) | (4, Bob) | (9, Ann) |
|---------|----------|----------|

Unsorted

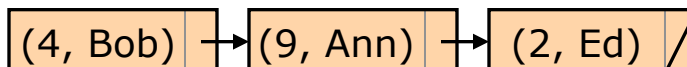
|          |          |         |
|----------|----------|---------|
| (4, Bob) | (9, Ann) | (2, Ed) |
|----------|----------|---------|

Linked List  
Implementations

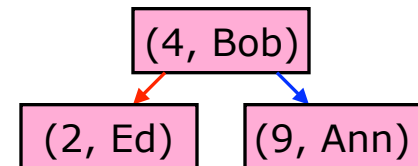
Sorted



Unsorted



Binary Search Tree  
Implementation



## Introduction to Tables

### Possible Implementations [2/3]

---

Find the order of each operation, for the following Table implementations.

- Allow multiple equivalent keys, where it matters. (Otherwise, when inserting, we must always do a retrieve operation first.)

|          | Sorted Array | Unsorted Array | Sorted Linked List | Unsorted Linked List | Binary Search Tree | Balanced*** Binary Search Tree |
|----------|--------------|----------------|--------------------|----------------------|--------------------|--------------------------------|
| Retrieve |              |                |                    |                      |                    |                                |
| Insert   |              |                |                    |                      |                    |                                |
| Delete   |              |                |                    |                      |                    |                                |

\*\*\*We do not—yet—know any way to guarantee that the tree will *stay* balanced, unless we can restrict ourselves to read-only operations (no insert, delete).

## Introduction to Tables

### Possible Implementations [2/3]

---

Find the order of each operation, for the following Table implementations.

- Allow multiple equivalent keys, where it matters. (Otherwise, when inserting, we must always do a retrieve operation first.)

|          | Sorted Array | Unsorted Array                    | Sorted Linked List | Unsorted Linked List | Binary Search Tree | Balanced*** Binary Search Tree |
|----------|--------------|-----------------------------------|--------------------|----------------------|--------------------|--------------------------------|
| Retrieve | Logarithmic  | Linear                            | Linear             | Linear               | Linear             | Logarithmic                    |
| Insert   | Linear       | Amortized constant (or constant)* | Linear**           | Constant             | Linear             | Logarithmic                    |
| Delete   | Linear       | Linear                            | Linear**           | Linear**             | Linear             | Logarithmic                    |

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

\*\*We must find the location first (retrieve operation); that requires linear time.

\*\*\*We do not—yet—know any way to guarantee that the tree will *stay* balanced, unless we can restrict ourselves to read-only operations (no insert, delete).

## Introduction to Tables

### Possible Implementations [3/3]

---

Tables can be implemented in many ways.

- Different implementations are appropriate in different circumstances.

In special situations, the (amortized) constant-time insertion for an unsorted array and the logarithmic-time retrieve for a sorted array can be combined!

- Insert all data into an unsorted array, sort the array, then use Binary Search to retrieve data.
- This is a good way to handle Table data with **separate filling & searching phases** (and few or no deletes).
- Note: We will be talking about some complicated Table implementations. But *sometimes* a simple solution is the best.

## Introduction to Tables

### Better Ideas? [1/3]

---

|          | 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.
- Maybe do not allow retrieve & delete on *all* keys.

In practice: Priority Queue



## Introduction to Tables

### Better Ideas? [2/3]

---

|          | 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 #2: Keep a Tree Balanced

- Figure out how to keep a Binary Search Tree balanced, without compromising efficiency.
- Maybe loosen the “binary” requirement.
- Maybe loosen the “balanced” requirement, too.

### In practice: Balanced search trees

- 2-3 Tree & 2-3-4 Tree
- **Red-Black Tree**
- AVL Tree
- **B-Tree & B+-Tree**

## Introduction to Tables

### Better Ideas? [3/3]

---

|          | 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 #3: “Magic Functions”

- Consider the simplest structure possible: unsorted array.
  - Arrays have fast look-up by index. So ...
  - ... a “magic function” that gives us an key’s index might make things very fast.
  - Ah, but what about deleting? Idea: Allow **empty** items, so that we do not need to move things down when we delete.
  - It looks like we might be able to insert, delete, and retrieve in **constant time**.
- But can we find a magic function?

In practice: Hash Tables

## Priority Queues

### What a Priority Queue Is — Introduction

---

Our next ADT is **Priority Queue**.

- It has almost the same operations as Queue.
- The difference is that each data item has a key, called its **priority**.  
The item retrieved/deleted is the item with the highest priority.

Conceptually, a Priority Queue is another way to do “standing in line”.

- However, items are not handled in the order they were inserted, but rather in order of priority.

Thus, a Priority Queue is not a Queue!

## Priority Queues

### What a Priority Queue Is — Restricted Table

---

We have discussed turning a Sequence into a Queue.

- In a Sequence, we retrieve/delete at **any given position**.
- In a Queue, we can retrieve/delete only the element at the **highest position**.

We can similarly turn a (sorted) Table into a Priority Queue.

- In a Table, we retrieve/delete the item with **any given key**.
- In a Priority Queue, we can retrieve/delete only the element with the **highest key** (priority).

Thus, a Priority Queue is a restricted-access (sorted) Table, just as a Queue is a restricted-access Sequence.

# 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 priority.
- Operations
  - The Usual
    - **create**, **destroy**, **copy**.
    - **isEmpty**.
  - Operations Specific to Priority Queues
    - **insert**. Given an item (which includes a priority).
    - **getFront**. Gets item with highest priority.
    - **delete**. Removes item with highest priority.

## Priority Queues Applications

---

A PQ is useful when we have items to process and some are more important than others.

By cleverly choosing priorities, we can produce “hybrid” structures.

- If an item’s priority includes a time stamp, then we can simulate a mixed Queue/PQ.
  - Items with the same priority can be dealt with in FIFO order.
- Or a mixed Stack/PQ, if you like.
  - What is this good for? I couldn’t say.

PQs can be used to do sorting.

- Insert all items, then retrieve/delete all items. The resulting sequence is sorted by priority.
- We can also use a PQ to handle special cases, in which the data to be sorted are modified during sorting.
- Note: Once again, a sorted container gives us a sorting algorithm. However, as with Insertion Sort, instead of using a separate container to sort with, we prefer to use an in-place version of the algorithm. We will call this “Heap Sort”.

## 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. ☹
- In particular, they all have linear-time delete.

The most interesting thing about PQs is the way they are *usually* implemented, using a structure called a (Binary) Heap.

- We discuss this next ...

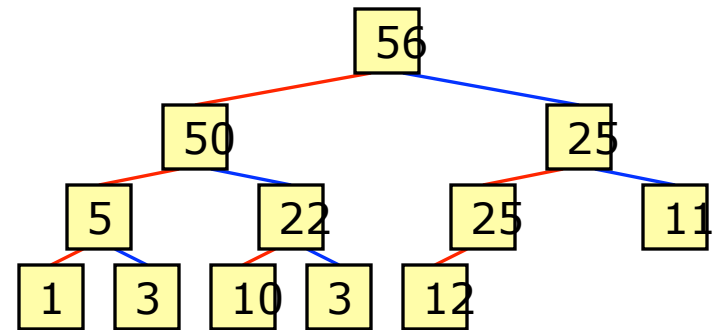
## Binary Heap Algorithms

### What is a Binary Heap? — Definition

---

We define a **Binary Heap** (usually just **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 Binary Heap.



### Notes

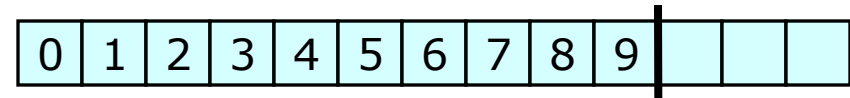
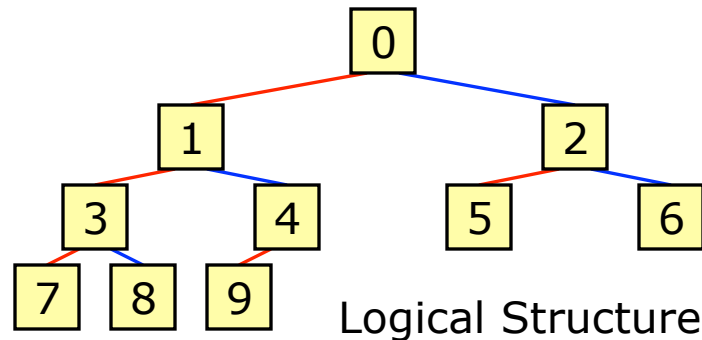
- This is a **Maxheap**. If we reverse the order, so that the root's key is  $\leq$  than the keys of its children, we get a **Minheap**.
- Some texts present Heap as an ADT with essentially the same operations as a Priority Queue. I am not doing this.
- As we will see, a Binary Heap is a good basis for an implementation of a Priority Queue.



## Refresher: Complete Binary Trees

We discussed an array implementation for a **complete** Binary Tree:

- Put the nodes in an array, in the order in which they would be added to a complete Binary Tree.
- No pointers/arrows/indices are required.
- We store *only* the **array** of data items and the **number** of nodes.



Physical Structure

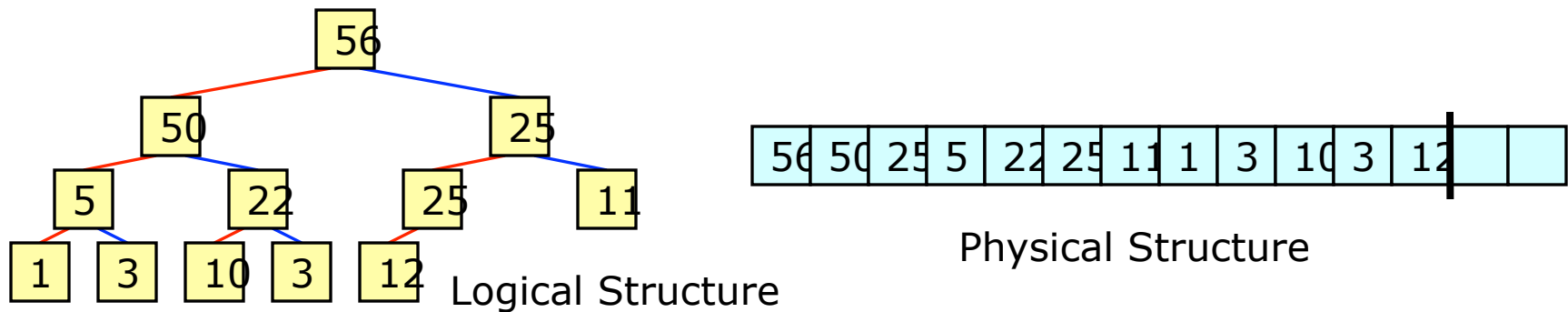
- Put the root, if any, at index 0.
- The left child of node  $k$  is at index  $2k + 1$ . It exists if  $2k + 1 < \text{size}$ .
- The right child is similar, but at  $2k + 2$ .
- The parent of node  $k$  is at index  $(k - 1)/2$  [int division]. It exists if  $k > 0$ .

## Binary Heap Algorithms

### What is a Binary Heap? — Implementation

---

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



### Notes

- There are no required order relationships between siblings.
- None of the standard traversals gives any sensible ordering.
- **In practice, we usually use “Heap” to mean a Binary Heap using this array representation.**
- In order to base a Priority Queue on a Heap, we need to know how to implement the operations.
  - getFront is easy (right?). Next we look at delete & insert.

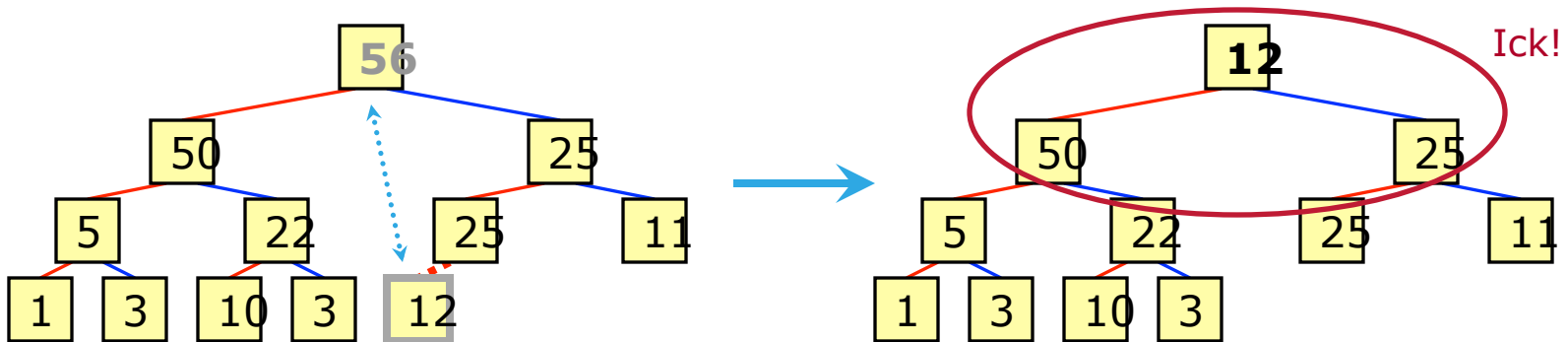
## Binary Heap Algorithms

### Operations — Delete [1/2]

In a Priority Queue, we can delete the highest-priority item.

In a Maxheap, this corresponds to the root. 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, and so the **last node** must go away.
- But the **last item** is not going away.
- Solution: Move the last node's item to the root; delete the last node.
  - We do this by swapping the items (which has other advantages, as we will see).



- But now we have another problem: This is no longer a Heap.
  - How do we fix it?

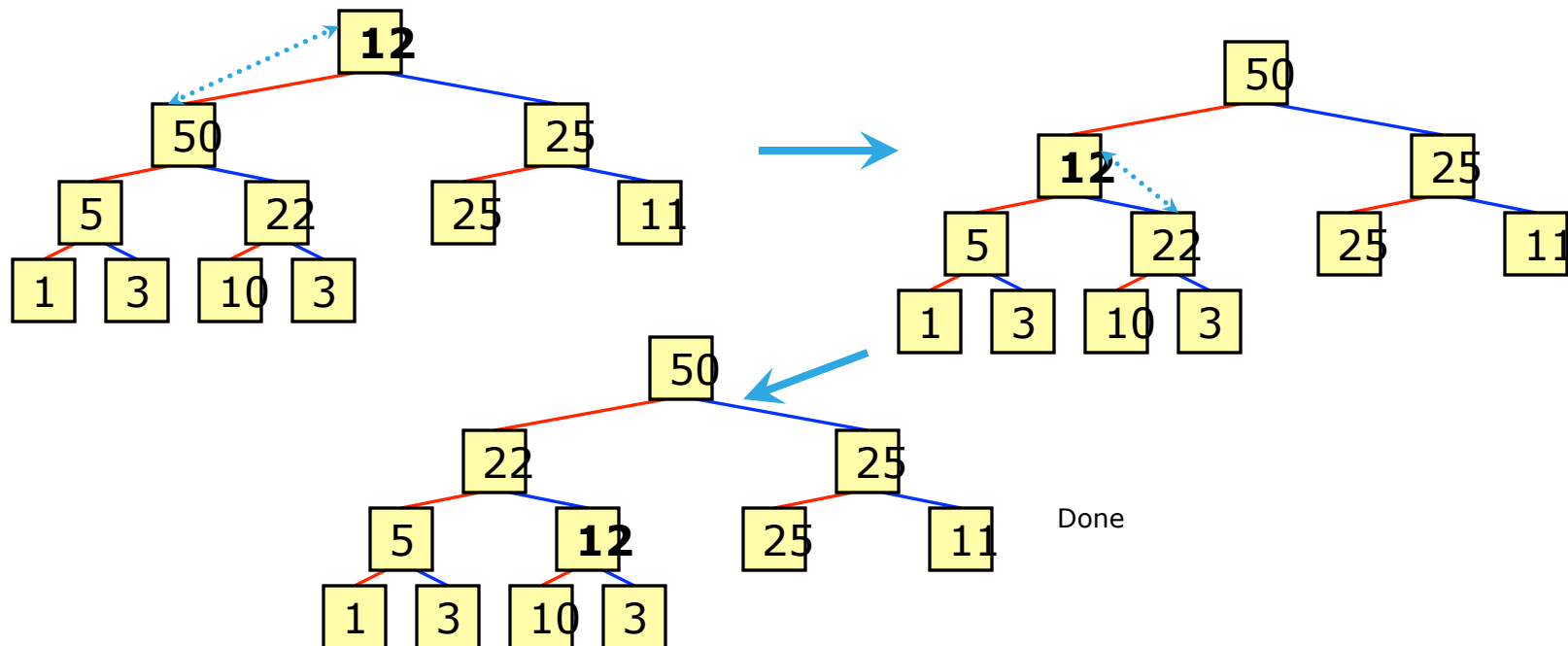
## Binary Heap Algorithms

### Operations — Delete [2/2]

After swapping the root with the last leaf and removing the node, we no longer have the heap property. The new root is not bigger than its children.

Solution: “Trickle down” (or “sift down”)

- Not bigger than both children? Swap with child (which one?) and repeat.
  - Swap with the larger child, so it is bigger than both of its children after the swap.



## Binary Heap Algorithms

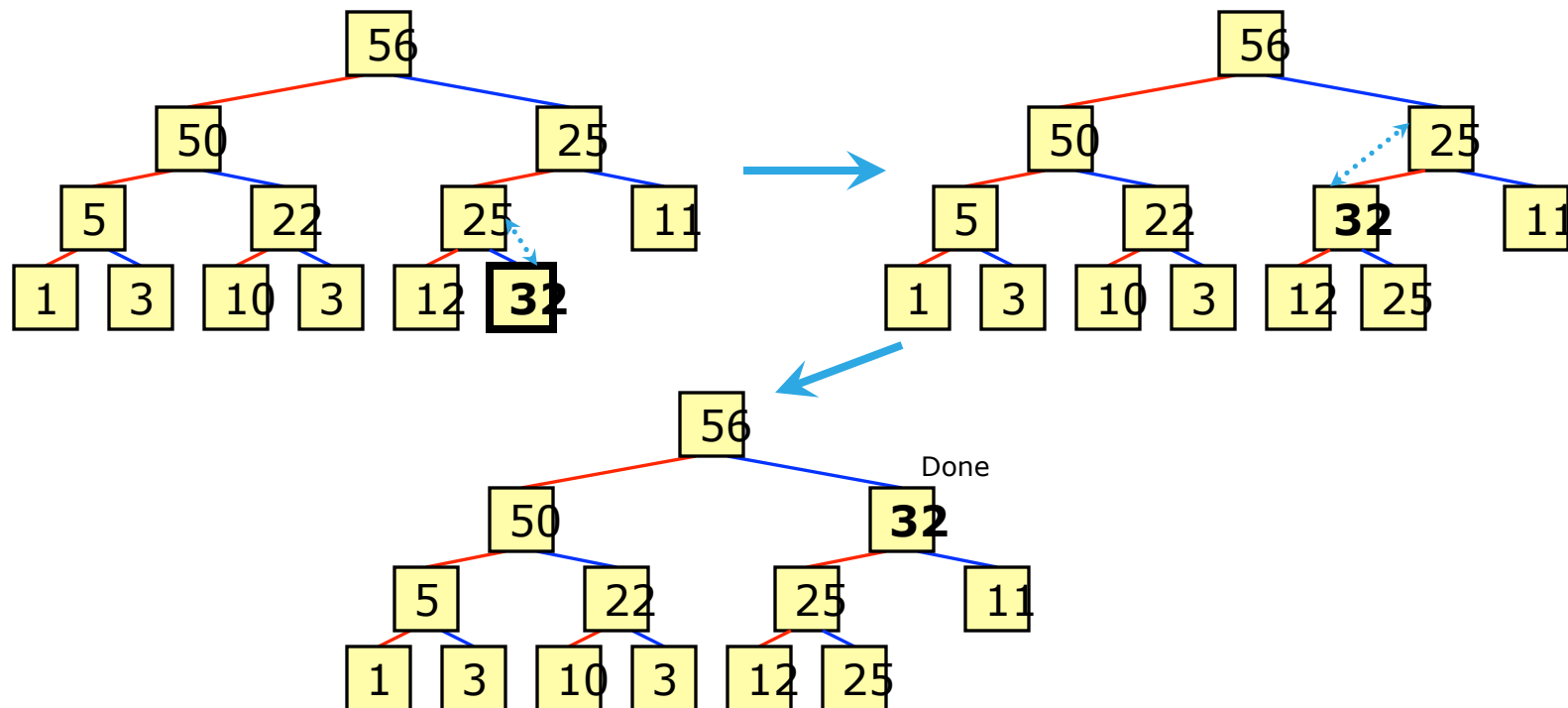
### Operations — Insert

To insert into a Heap, add the new item in a new node at the end.

- But if we put our new value in this node, we may not have a Heap.

Solution: “Trickle up” (or “sift up”).

- New value greater than parent? Swap them. Repeat.

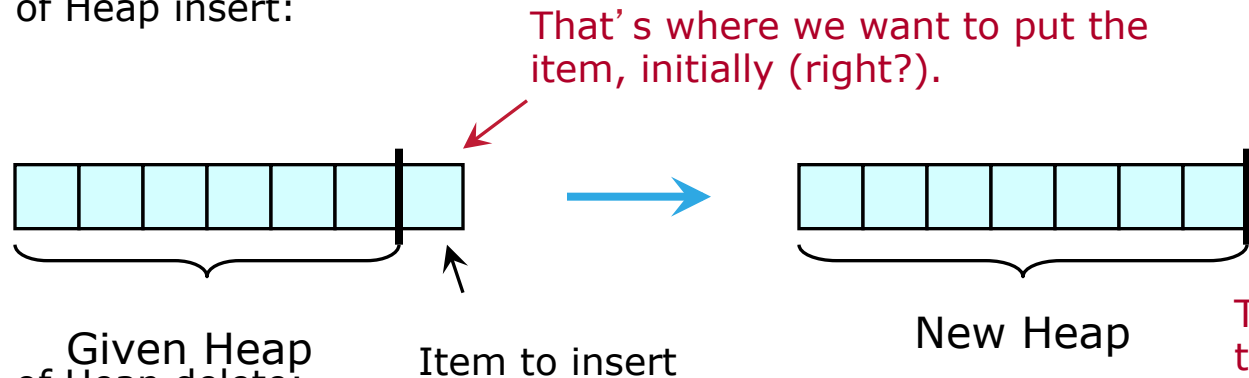


## Binary Heap Algorithms

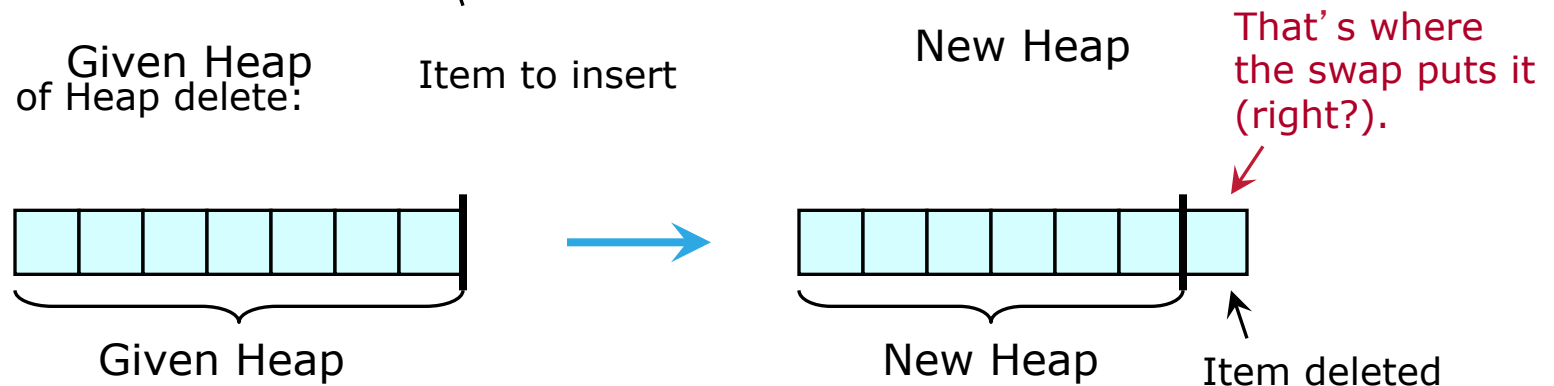
### Operations — Using an Array

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

### Efficiency

---

What is the order of the three main Priority Queue operations, if we use a Binary Heap implementation based on a complete Binary Tree stored in an array?

- **getFront**
  - Constant time.
- **insert**
  - Logarithmic time.
    - Assuming no reallocation, that is, assuming the array is large enough to hold the new item. As on the previous slide, the way that Heaps are **used** often guarantees that this is the case. (Linear time if possible reallocation.)
  - The number of operations is roughly the height of the tree. Since the tree is balanced, the height is  $O(\log n)$ .
- **delete**
  - Logarithmic time.
  - No reallocation, of course. Other comments as for insert.

Better than linear time!

We have not seen this before,  
for a *value-oriented* delete.

We conclude that a Heap is an excellent basis for an implementation of a Priority Queue.

# Binary Heap Algorithms

## Write It!

---

### TO DO

- Examine 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.

*Done. See `heapalgs.cpp`,  
on the web page.*

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



# Binary Heap Algorithms

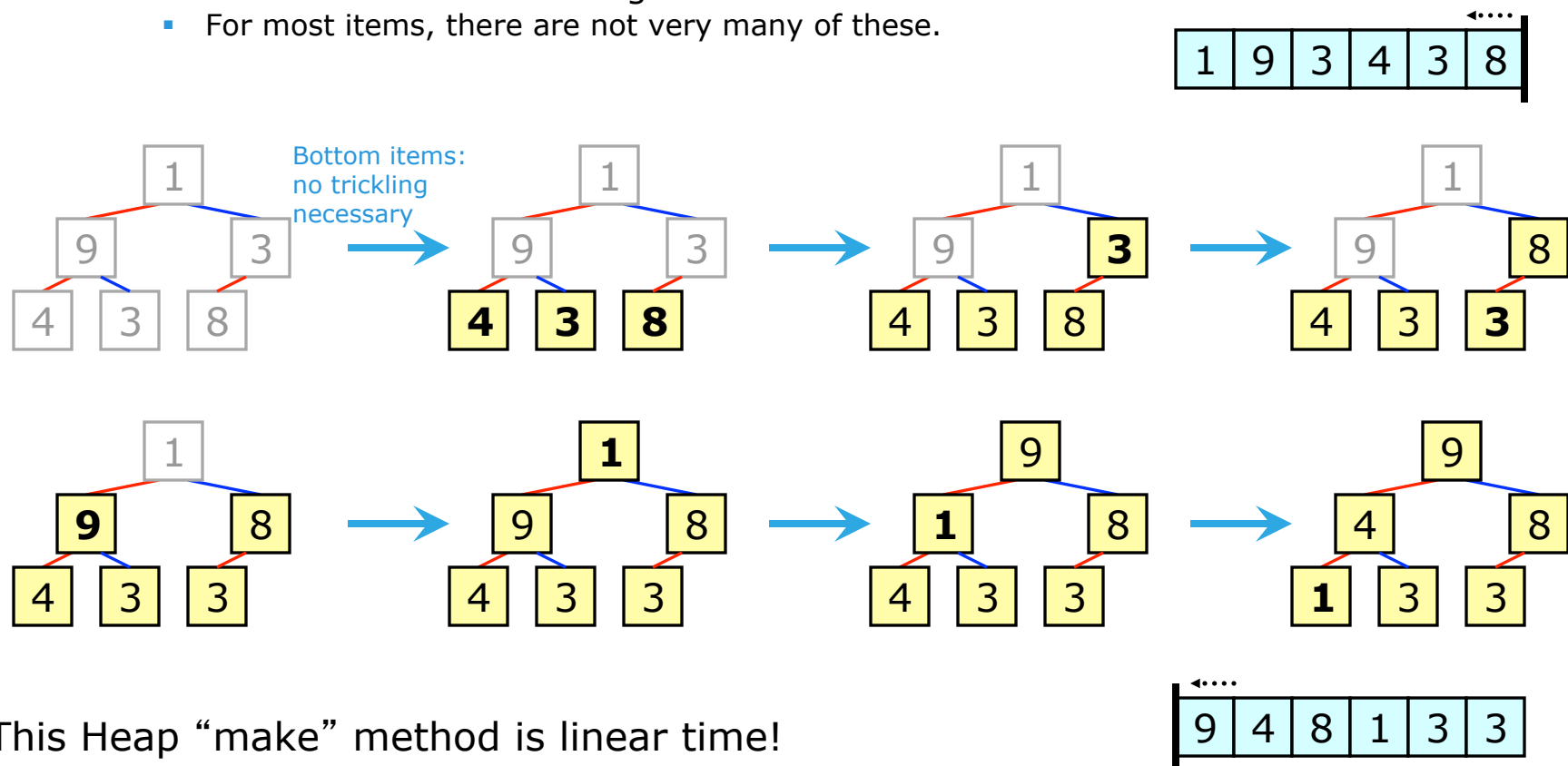
## An Efficient “Make” Operation

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:  $O(n)$ .

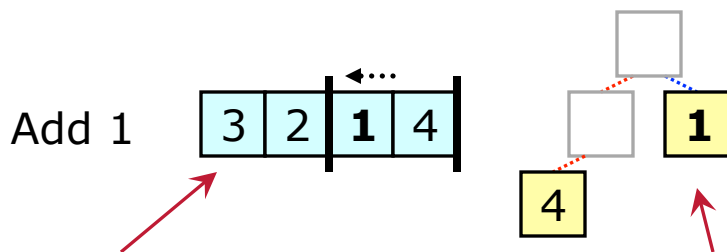
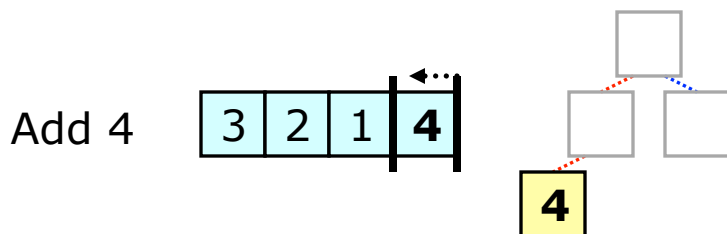
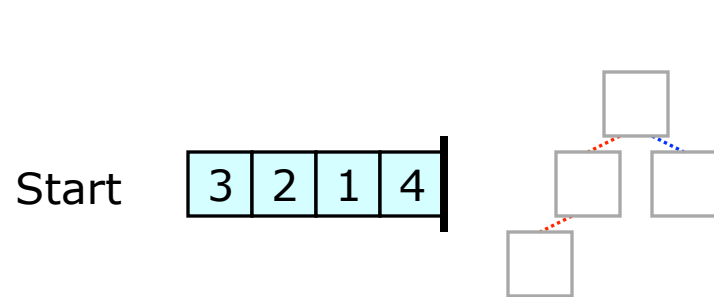
Heap Sort also can easily be generalized.

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

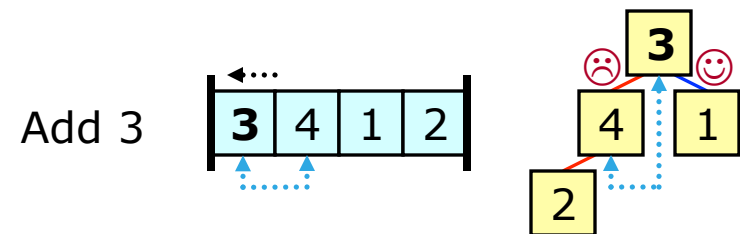
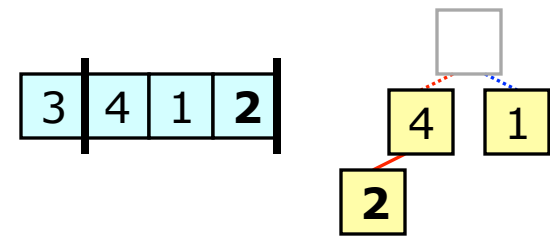
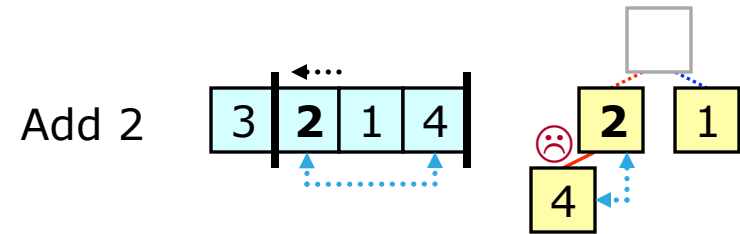
# Binary Heap Algorithms

## Heap Sort — Illustration [1/2]

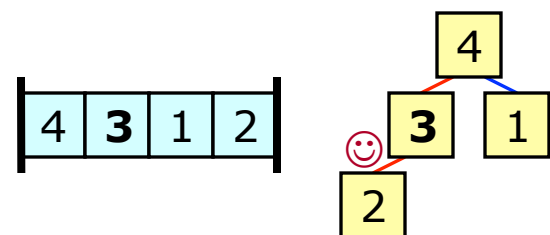
Below: Heap make operation. Next slide: Heap deletion phase.



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



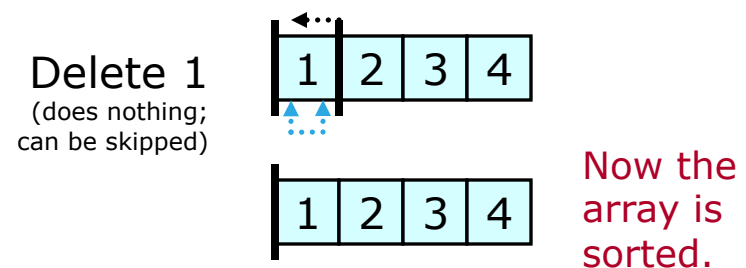
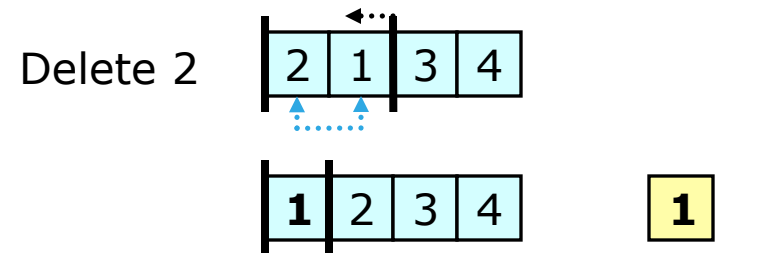
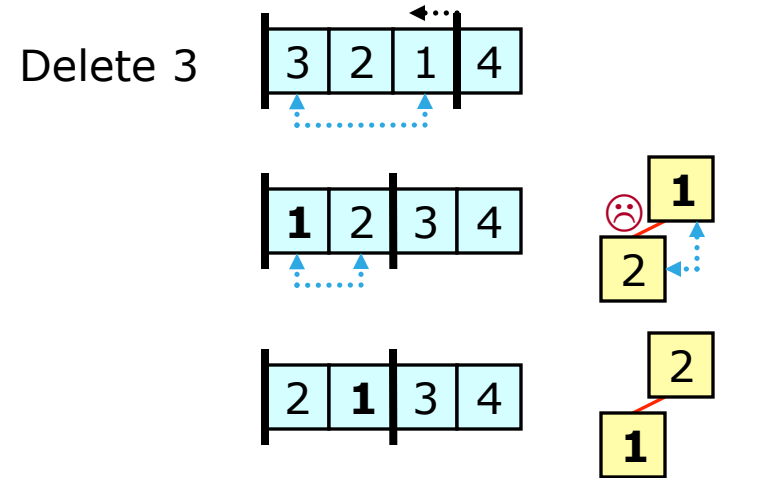
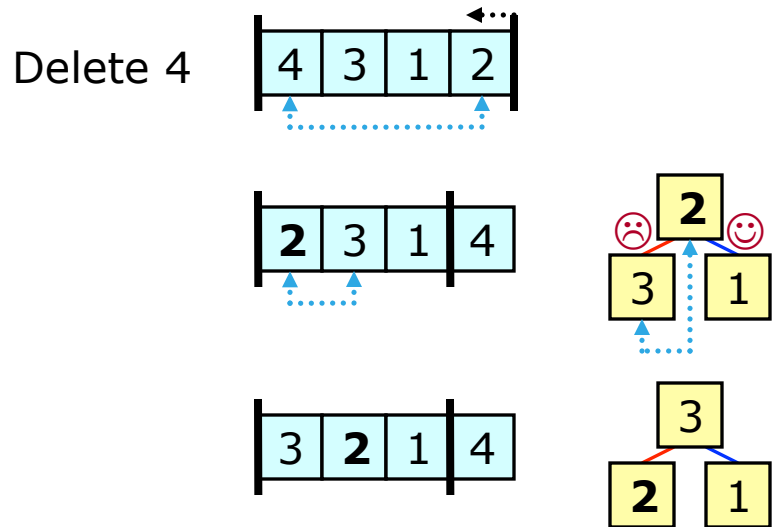
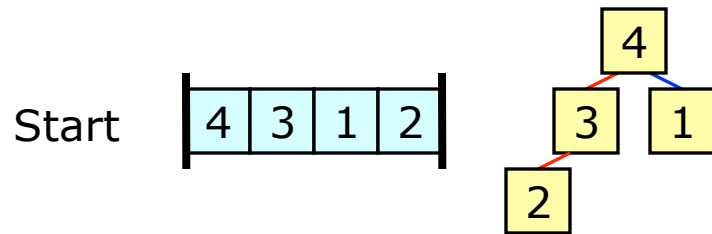
Now the entire array is a Heap.



# 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.