CSE 373: Data Structures and Algorithms

Lecture 11: Finish AVL Trees; Priority Queues; Binary Heaps

Instructor: Lilian de Greef
Quarter: Summer 2017

Today

- Announcements
- Finish AVL Trees
- Priority Queues
- Binary Heaps

Announcements

- Changes to Office Hours: from now on…
  - **No** more Wednesday morning & **shorter** Wednesday afternoon office hours
  - **New** Thursday office hours!
  - Kyle's Wednesday hour is now 1:30-**2:30**pm
  - Ben's office hours is now Thursday 1:**00**-3:**00**pm

- AVL Tree Height
  - In section, computed that minimum # nodes in AVL tree of a certain height is
    S(h) = 1 + S(h-1) + S(h-2)   where h = height of tree
  - Posted a proof next to these lecture slides online for your perusal

Announcements

- Midterm
  - Next Friday! (at usual class time & location)
  - Everything we cover in class until exam date is fair game (minus clearly-marked "bonus material"). That includes next week's material!
  - Today's hw3 due date designed to give you time to study.

- Course Feedback
  - Heads up: official UW-mediated course feedback session for part of Wednesday
  - Also want to better understand an anonymous concern on course pacing → Poll

Back to AVL Trees

Finishing up last couple cases for insert, then wrapping up BSTs

The AVL Tree Data Structure

An **AVL tree** is a *self-balancing* binary search tree.

Structural properties

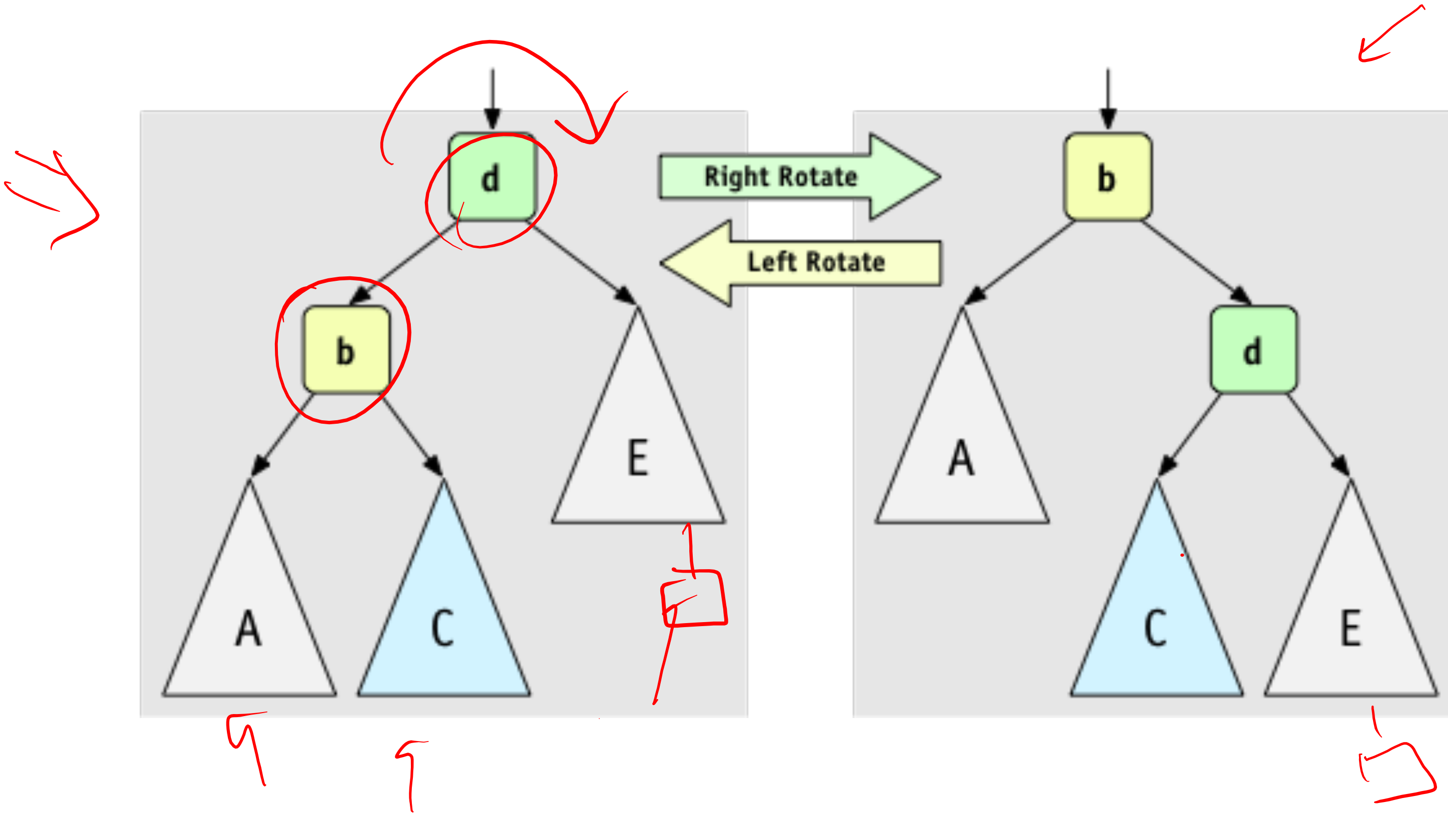
1. Binary tree property (same as BST)
2. Order property (same as for BST)

3. Balance condition:
   balance of every node is between -1 and 1

   where **balance**(*node*) = height(*node*.left) – height(*node*.right)

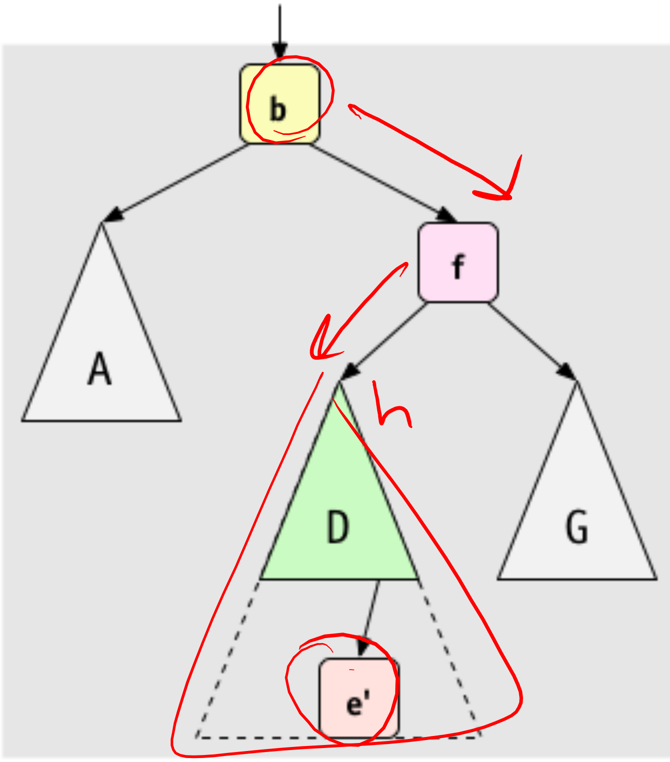
Result: **Worst-case** depth is O(log n)

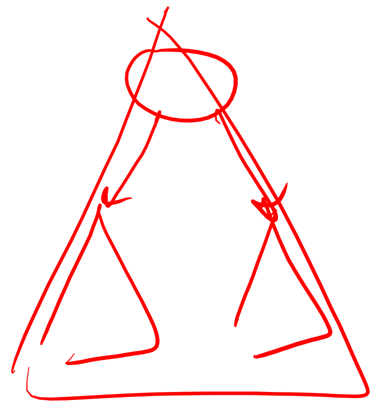
Single Rotations



(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

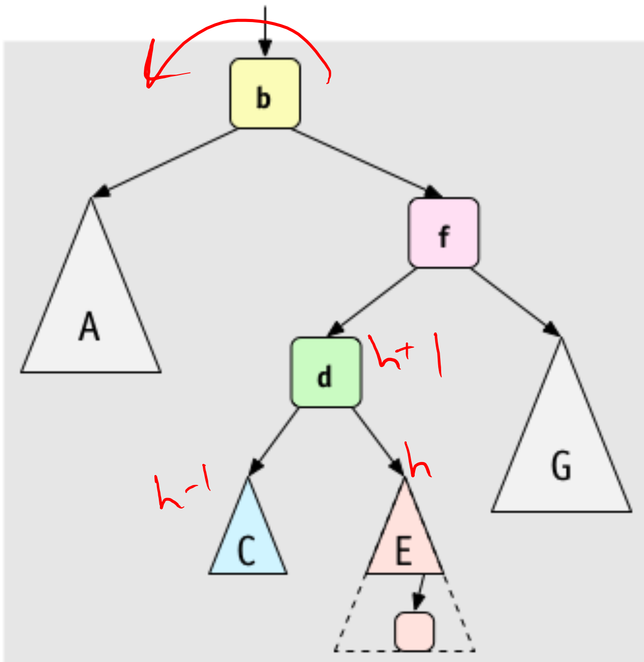
Case #3: Right-Left Case





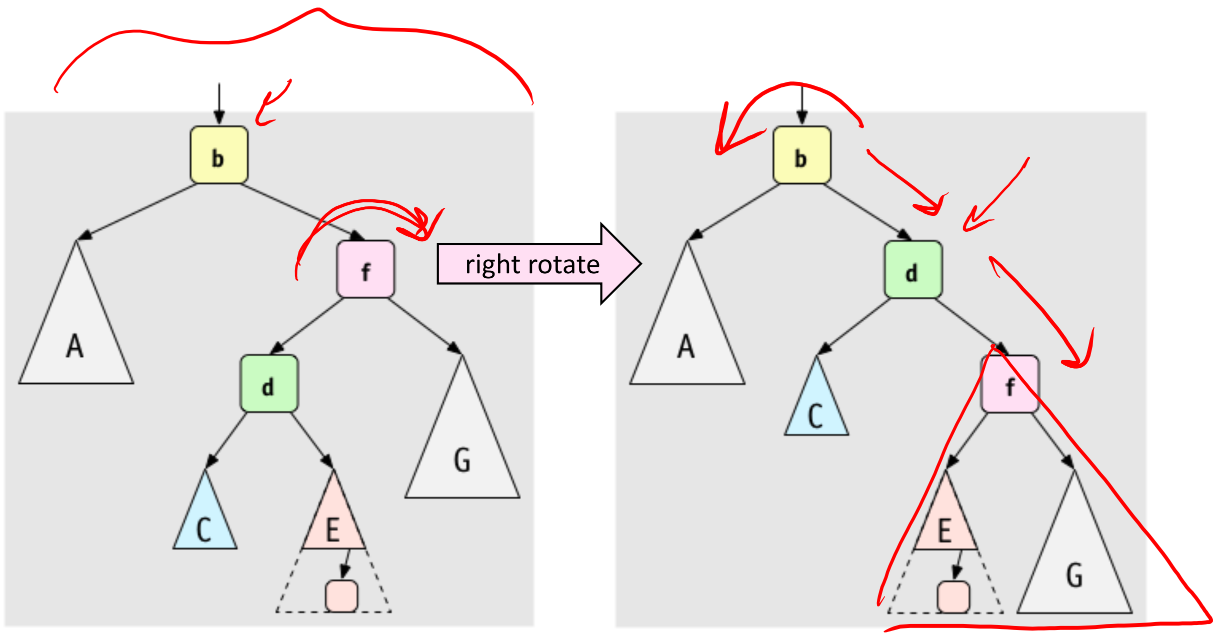
(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

A Better Look at Case #3:



(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

Case #3: Right-Left Case (after one rotation)

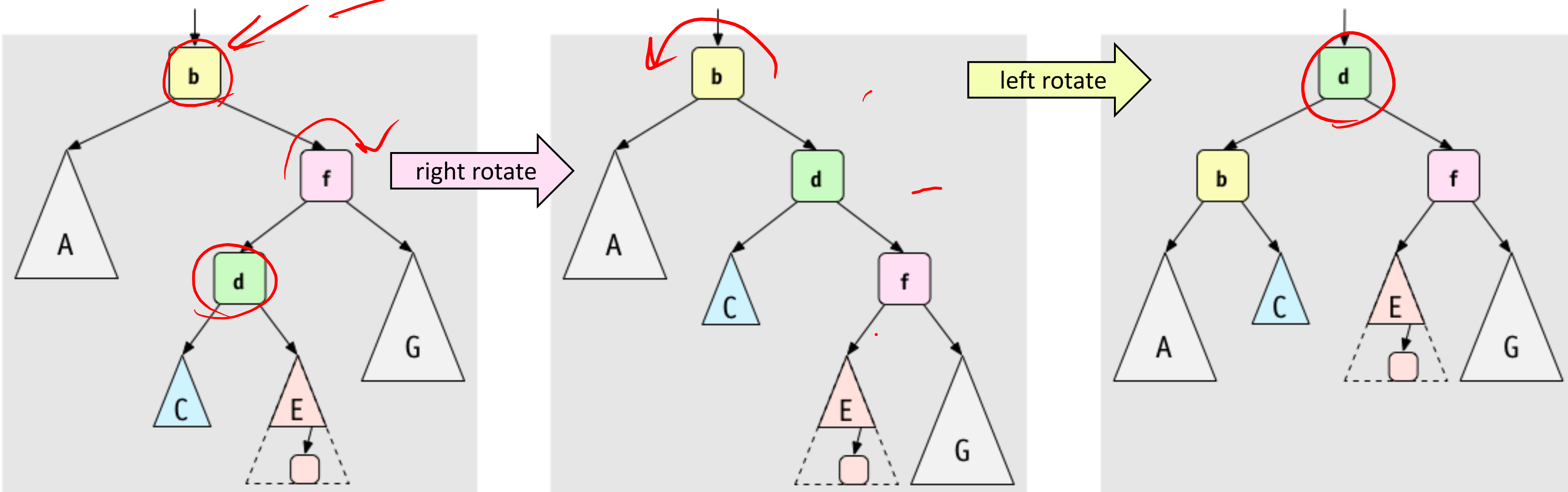


(Almost) Right-Right case!

(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

Case #3: Right-Left Case (after two rotations)

imbalanced!

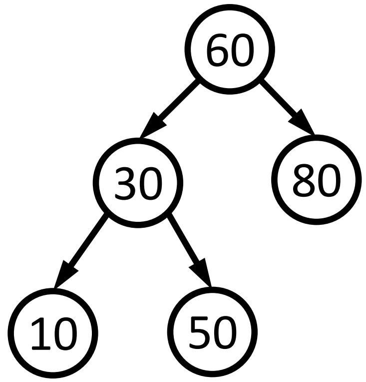


A way to remember it:
Move d to grandparent's position. Put everything else in their only legal positions for a BST.

(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

Practice time! Example of Case #4

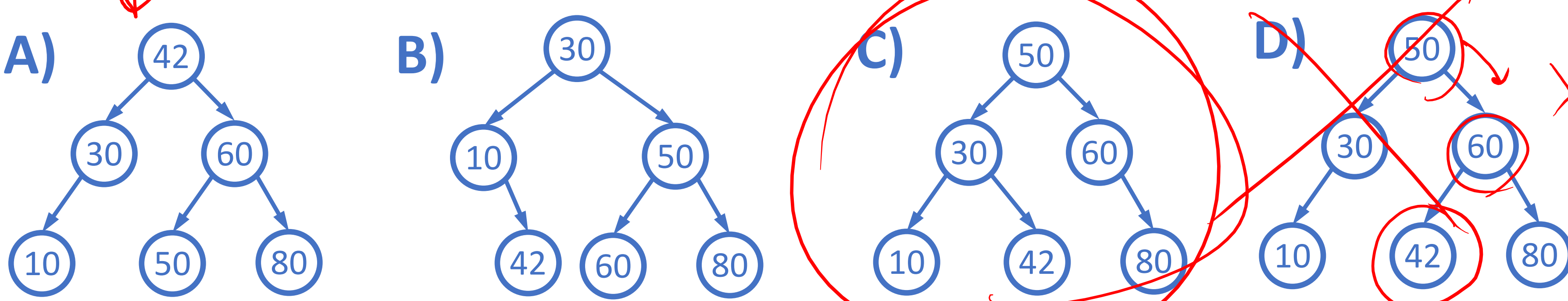
Starting with this AVL tree:



Which of the following is the updated AVL tree after inserting 42?

is an AVL Tree!

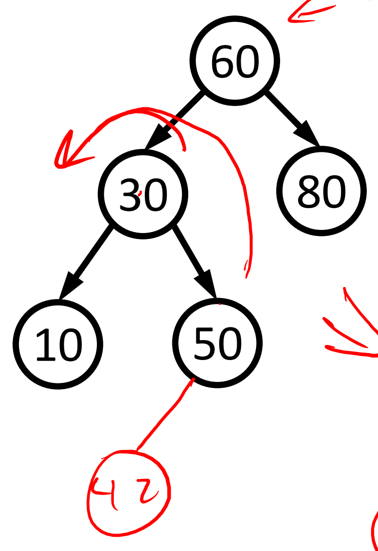
Is an AVL tree!



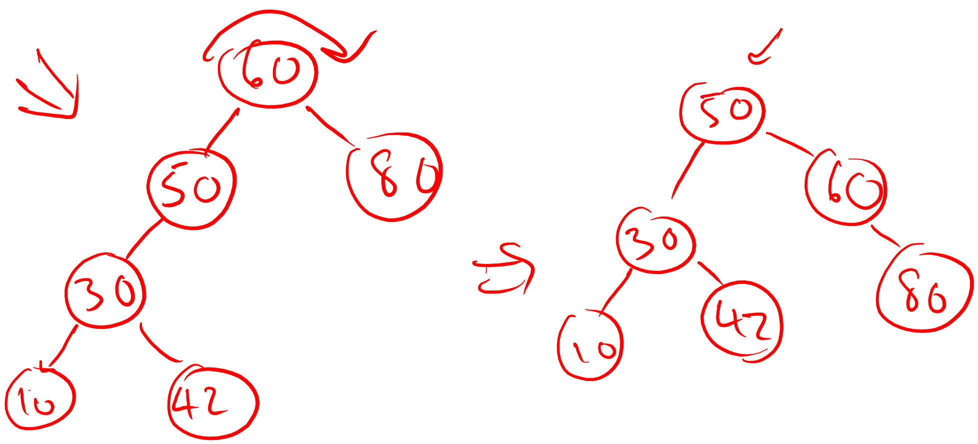
Practice time! Example of Case #4

Starting with this AVL tree:

imbalanced!



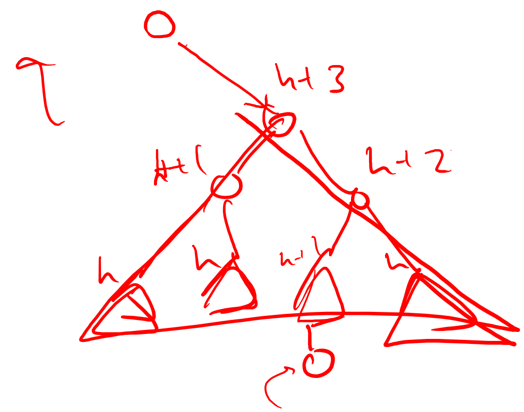
Which of the following is the updated AVL tree after inserting 42?



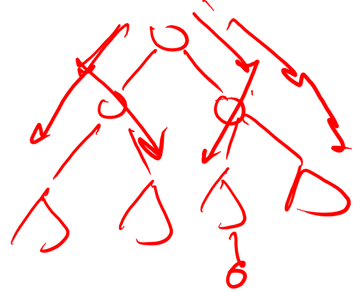
What's the name of this case? Left-Right

What rotations did we do? Left rotate (30)
Right rotate (60)

Insert, summarized



- Insert as in our generic BST
- Check back up path for imbalance, which will be 1 of 4 cases:
  - Node's left-left grandchild is too tall
  - Node's left-right grandchild is too tall
  - Node's right-left grandchild is too tall
  - Node's right-right grandchild is too tall



- Only *one case* occurs because *tree was balanced before insert*
- After the appropriate single or double rotation, the smallest-unbalanced subtree has the same height as before the insertion
  - So all ancestors are now balanced

AVL Tree Efficiency

- Worst-case complexity of `find`: $O(\log n)$
- Worst-case complexity of `insert`: $O(\log n)$
  - $O(\log n)$ to find where to insert
  - $O(1)$ to do rotation
- Worst-case complexity of `buildTree`: $O(n \log n)$

Takes some more rotation action to handle `delete`...
(not covered in this course)

Pros and Cons of AVL Trees

Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are *always* balanced
2. Height balancing adds no more than a constant factor to the speed of `insert` and `delete`

Arguments against AVL trees:

1. Difficult to program & debug [but done once in a library!]
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. If *amortized* logarithmic time is enough, use splay trees (also in the text, not covered in this class)

Lots of cool Self-Balancing BSTs out there!

Popular self-balancing BSTs include:

- AVL tree
- Splay tree
- 2-3 tree
- AA tree
- Red-black tree
- Scapegoat tree
- Treap

(Not covered in this class, but several are in the textbook and all of them are online!)

(From https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree#Implementations)

Wrapping Up: Hash Table vs BST Dictionary

Hash Table advantages:

Average $\Theta(1)$ find, insert, delete

BST advantages:

- Can get keys in sorted order without much extra effort
- Same for ordering statistics, finding closest elements, range queries
- Easier to implement if don't have hash function (which are hard!).
- Can guarantee O(log n) *worst-case* time, whereas hash tables are O(1) *average* time.

Priority Queue ADT

Like a Queue, but with priorities for each element.

An Introductory Example…

Gill Bates, the CEO of the software company Millisoft,
built a robot secretary to manage her hundreds of emails.
During the day, Bates only wants to look at a few emails every now and then so she can stay focused.
The robot secretary sends her the most important emails each time.
To do so, he assigns each email a *priority* when he gets them
and only sends Bates the *highest priority* emails upon request.

All of your computer servers are on fire!
Priority: 1

Here's a reminder for our meeting in 2 months.
Priority: 42

Priority Queue ADT

A **priority queue** holds *compare-able data*

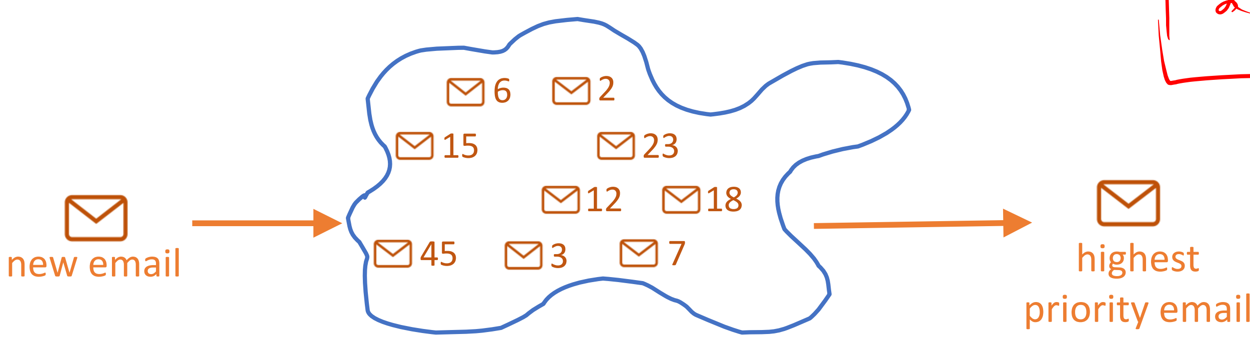
In our introductory example:

Item:

priority: # assigned

data: email

- Like dictionaries, we need to *compare items*
  - Given *x* and *y*, is *x* less than, equal to, or greater than *y*
  - Meaning of the ordering can depend on your data



- The data typically has two fields: priority and data
  - We'll now use integers for examples, but can use other types /objects for priorities too!

Priority Queue ADT

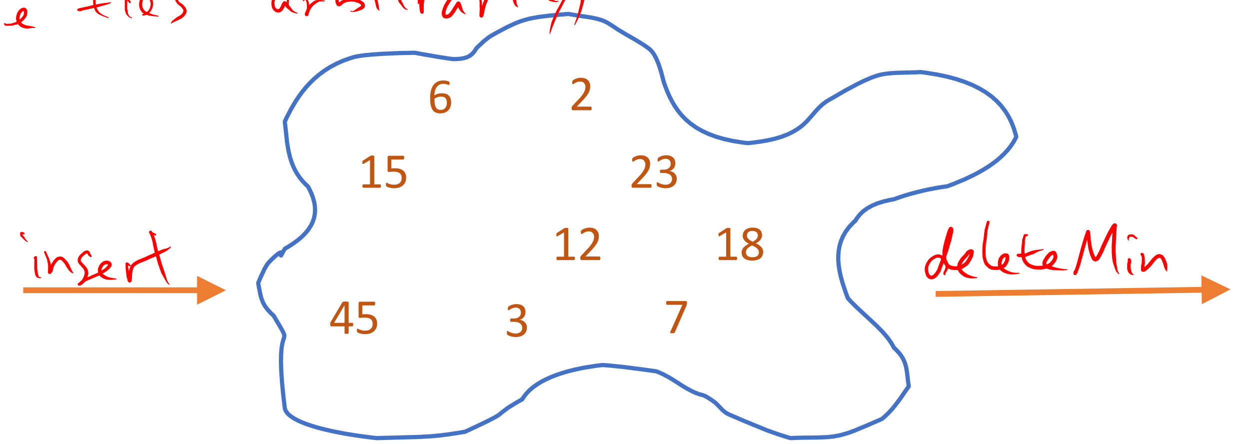
Meaning:

- A **priority queue** holds *compare-able data*
- Key property: deleteMin returns and deletes the item with the highest priority.

(can resolve ties arbitrarily)

Operations:

- deleteMin
- insert
- contains
- isEmpty



Priority Queue: Example

`insert` x_1 with priority *5*

`insert` x_2 with priority *3*

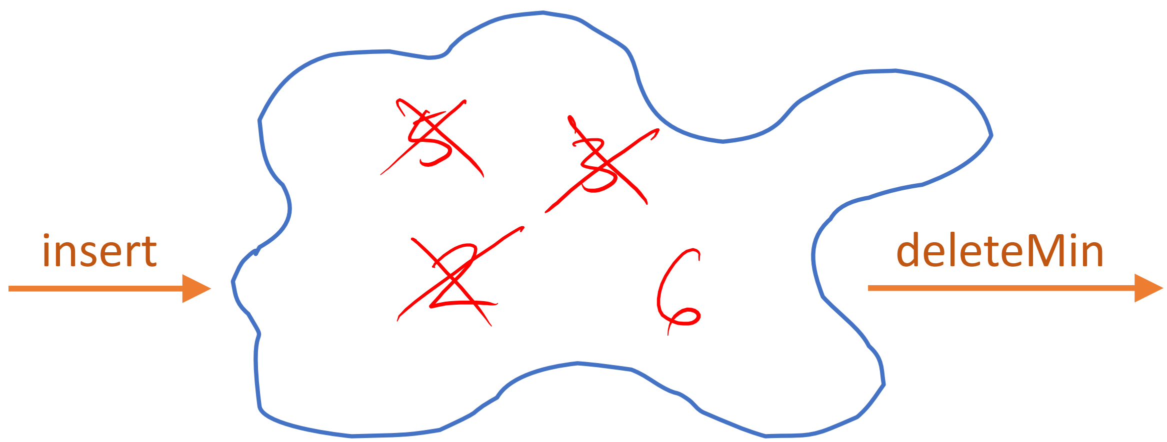
a `= deleteMin`

`insert` x_3 with priority *2*

`insert` x_4 with priority *6*

c `= deleteMin`

d `= deleteMin`



a = 3

c = 2

d = 5

Analogy: `insert` is like `enqueue`, `deleteMin` is like `dequeue`

But the whole point is to use priorities instead of FIFO