7/8 Lecture Agenda

- Announcements
- Part 3-3: Heaps and Priority Queues
- 10 minute break!
- Part 3-4: Self-Balancing BSTs

Announcements!

- HW2 due this Sunday (Jul 10)
- Pre-HW3 due next Wednesday (Jul 13)
- HW3 out tonight (best case) or tomorrow (worst case)
- HW1 solutions out tonight; we'll get it graded as soon as we can!

Looking ahead to the midterm

- 85 minutes; two or three multipart problems and some "True/False + justify your answer" questions

- Lecture coverage: Units 1, 2, 3, first lecture of Unit 4. HW / Pre-HW coverage: Units 1, 2, 3

- You are not responsible for small details (e.g., the Gray code from Pre-HW1). Emphasis will be on ideas we've seen and practiced multiple times.

- Weds. lecture (before the Friday exam): Midterm Review!
    - I'll walk through each lecture explaining what to focus on.

7/8 Lecture Agenda

- Announcements
- Part 3-3: Heaps and Priority Queues
- 10 minute break!
- Part 3-4: Self-Balancing BSTs

Heaps and Priority Queues

Divide and Conquer

Sorting & Randomization

Data Structures

Graph Search

Dynamic Programming

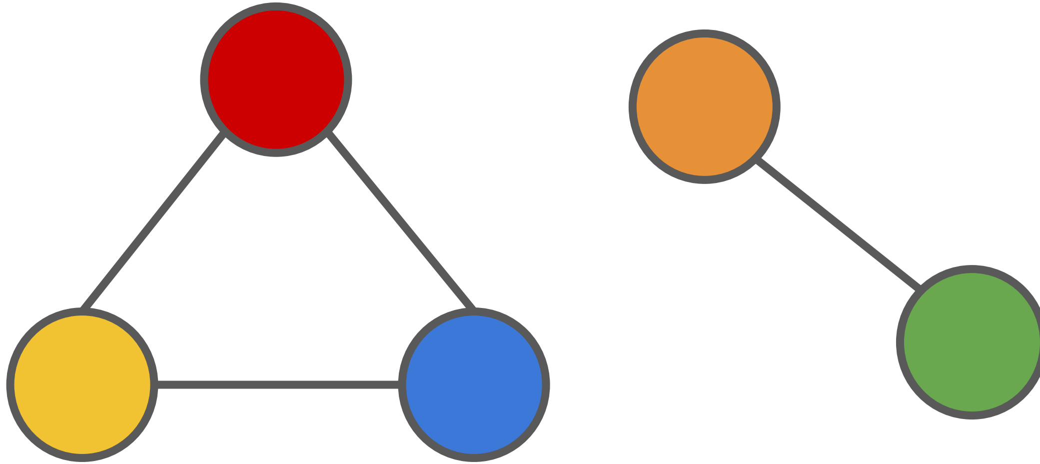
Greed & Flow

Special Topics

First, let's recap trees

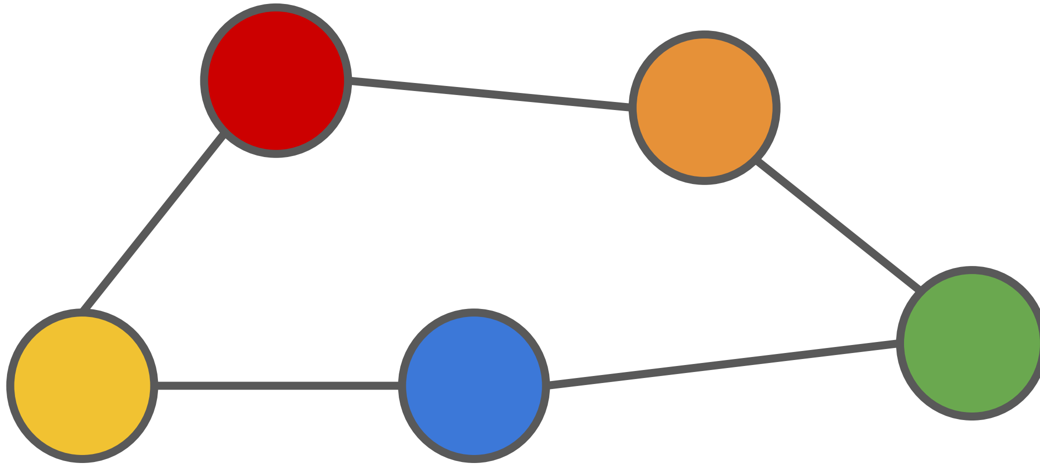
- A tree is a **connected** graph with n nodes and exactly n−1 edges.
    - Node is just another word for vertex. But for some reason it's more common to call them nodes when talking about trees.
- These properties are enough to guarantee an absence of cycles!

Not a tree



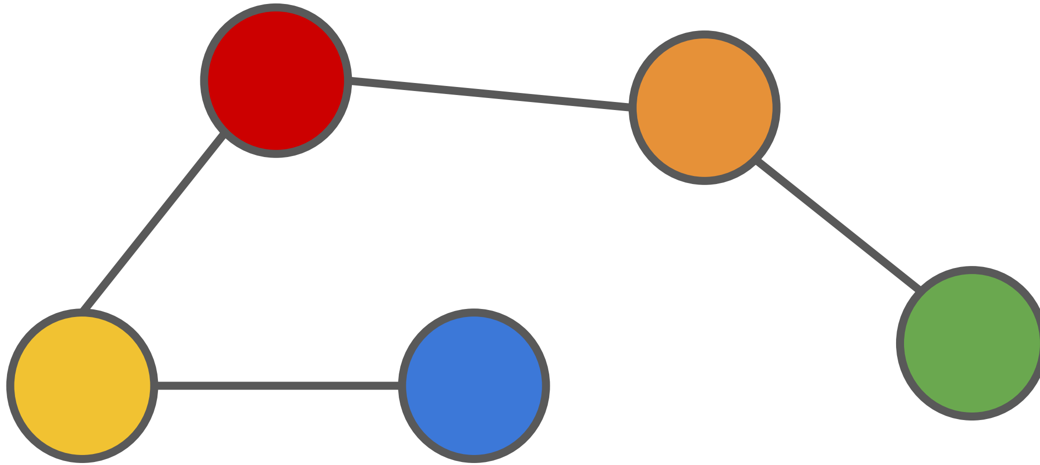
Correct number of edges, but not connected.

Also not a tree



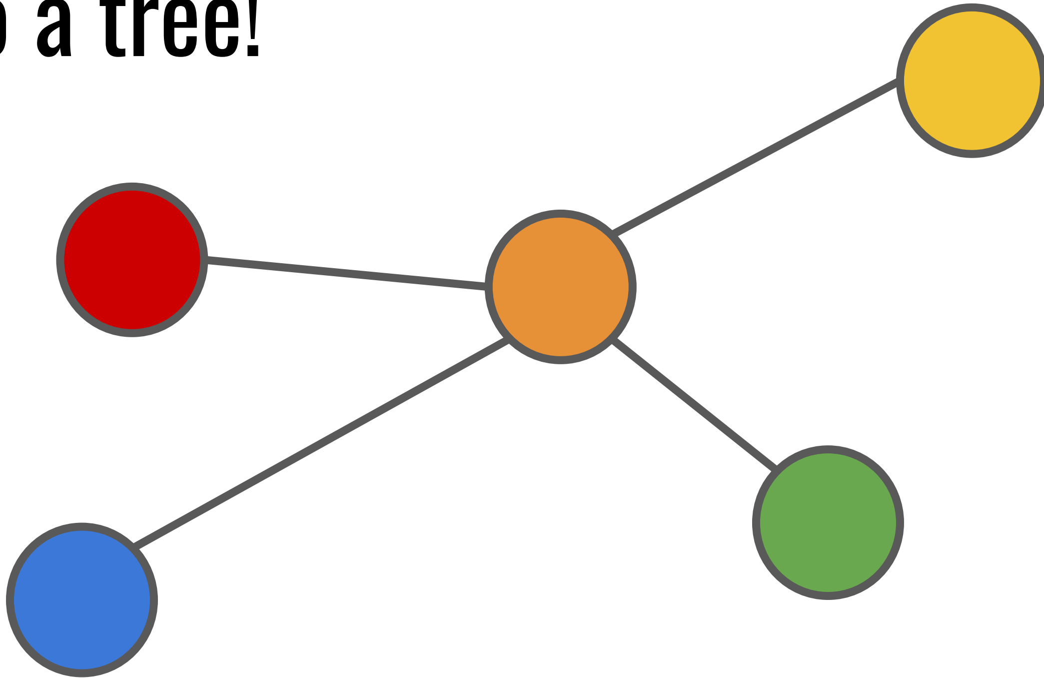
Connected, but too many edges.
(Even one edge too many is guaranteed to create a cycle!)

A tree!



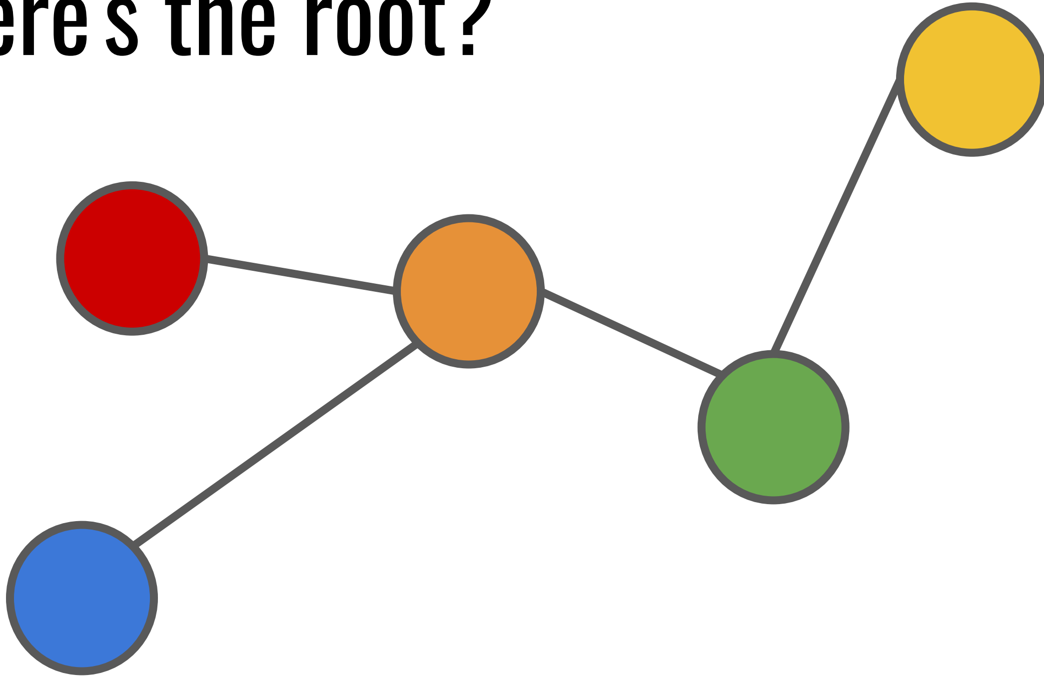
Even though it doesn't branch, a linear graph like this is still a perfectly legit tree.

Also a tree!

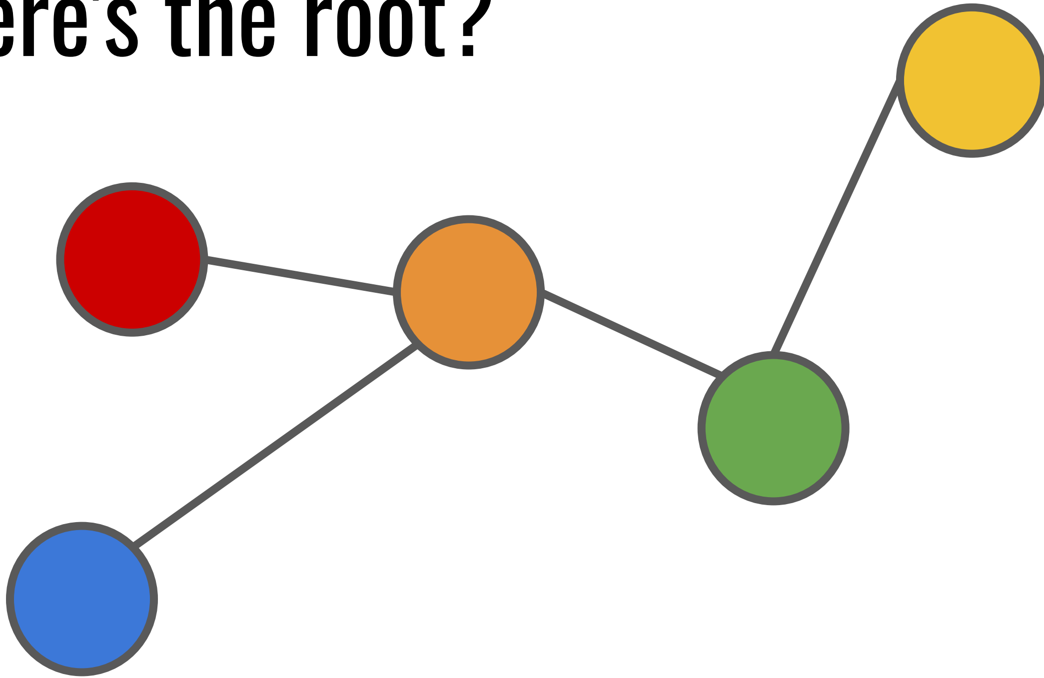


This is called a "star graph". The term isn't important, but this can be a useful worst (or at least extreme) case for some algorithms.

Where's the root?

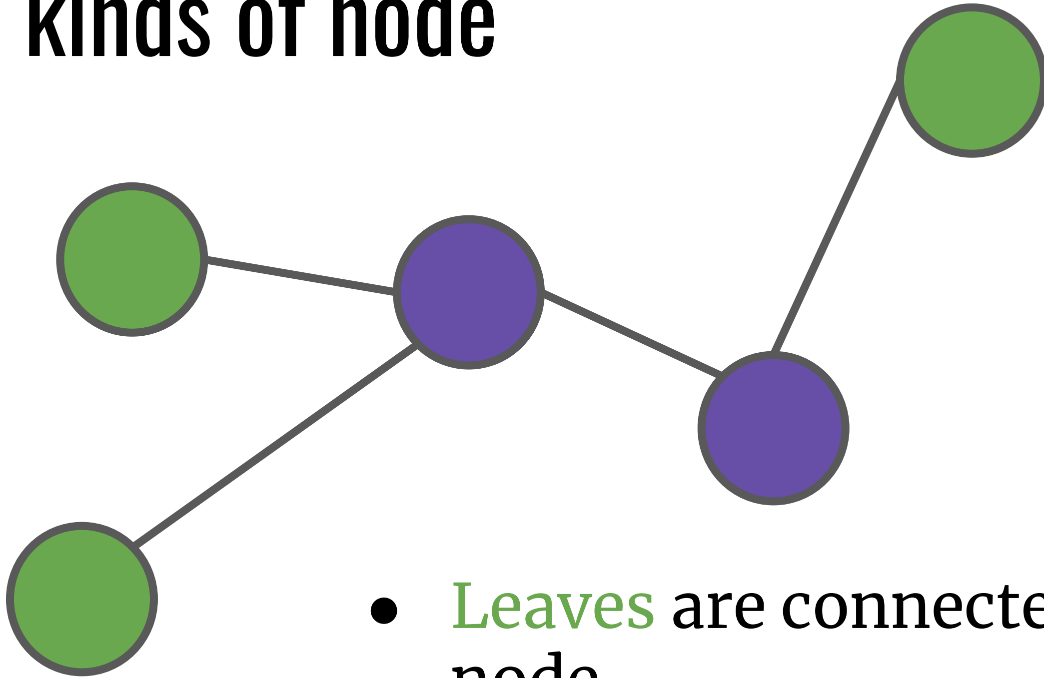


Where's the root?



Technically, any node in a tree can be the root!

Two kinds of node

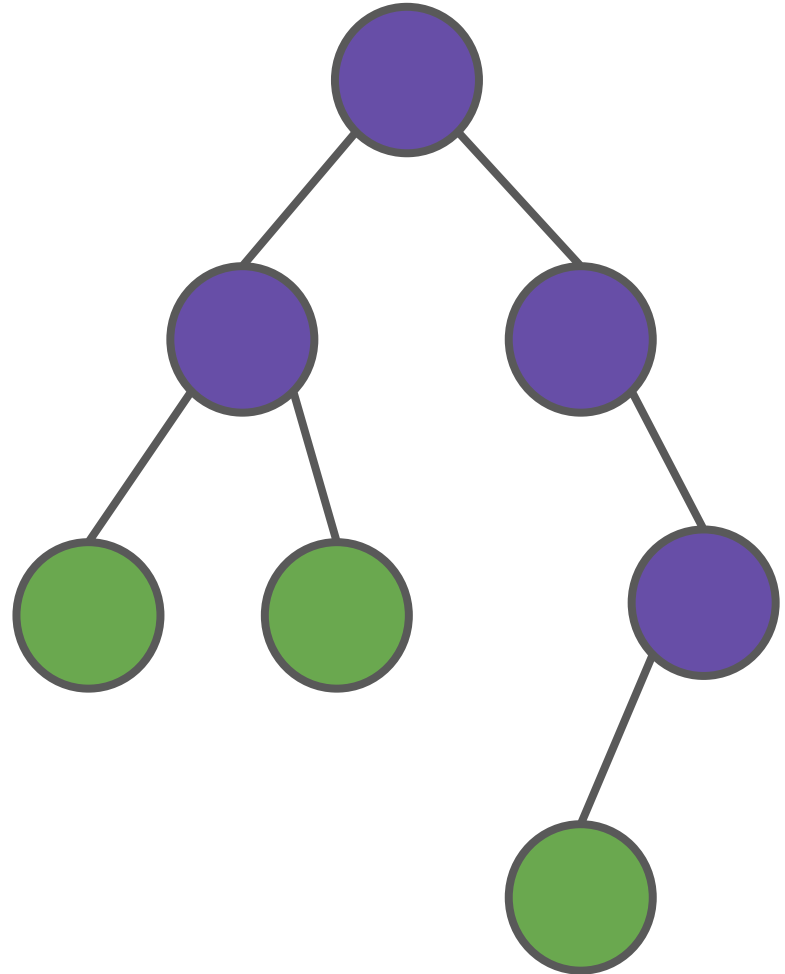


- Leaves are connected to only 1 other node.

- The other nodes are sometimes called "internal nodes".

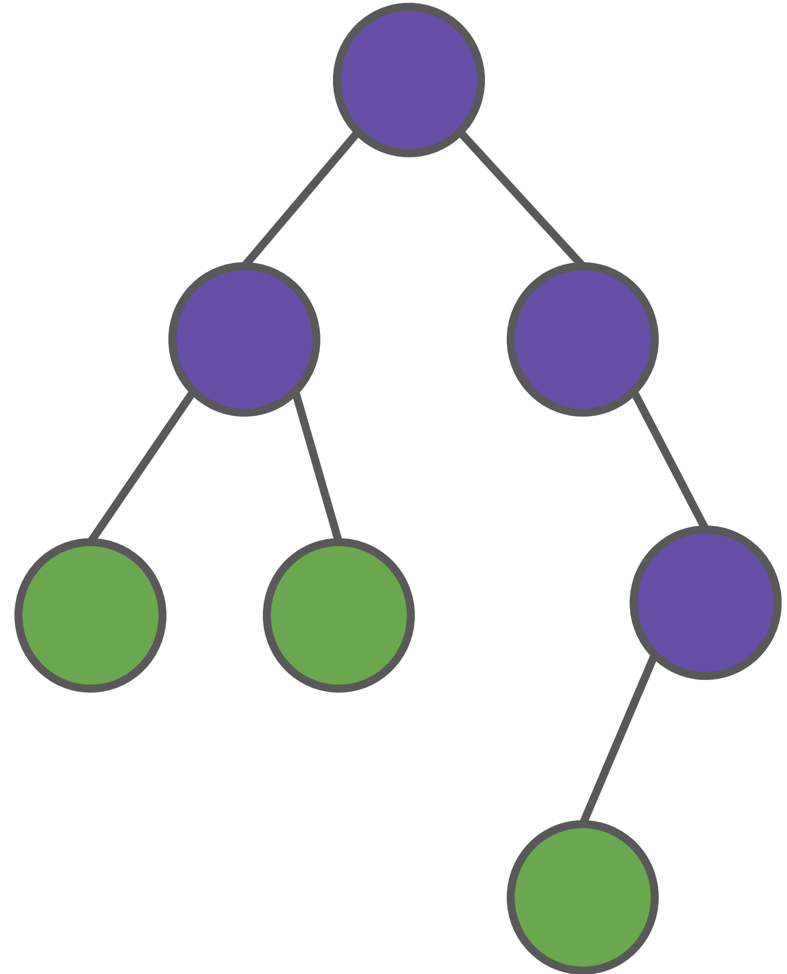
Binary trees

- A tree is **binary** if every node is either a leaf or has *at most* two children.
  - (By talking about children, we imply that we have designated a root.)



Notice that

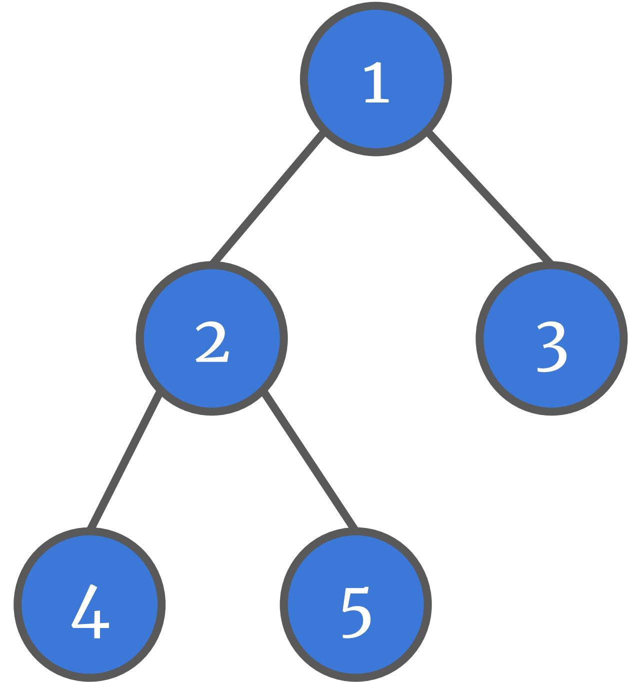
- Binary trees are not necessarily well-balanced
- Internal nodes do not necessarily have 2 children
- There are other possible ways to root this tree



Trees: Why do we care?

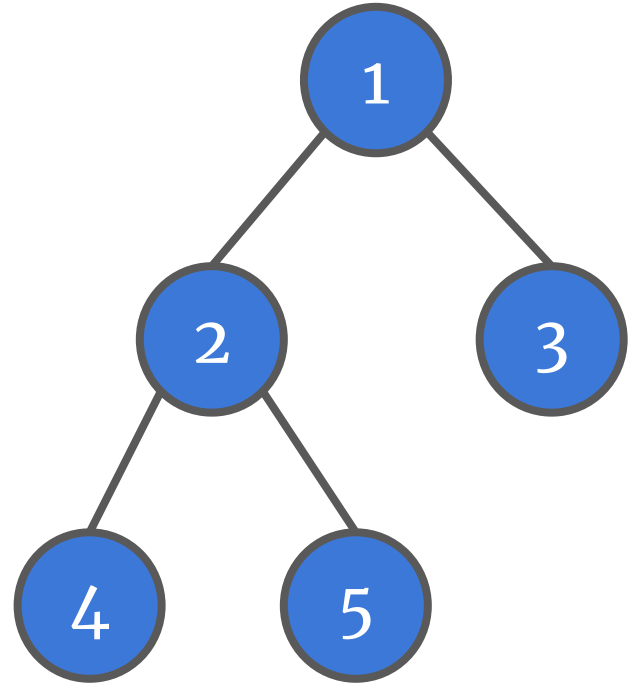
- Computer programs can generally be represented as trees. (CS143, Compilers, is a tree class!)
    - Same with natural language...
- Trees are a good fit for anything with a natural hierarchical relationship (e.g. directories on a computer)

Nodes can have values!

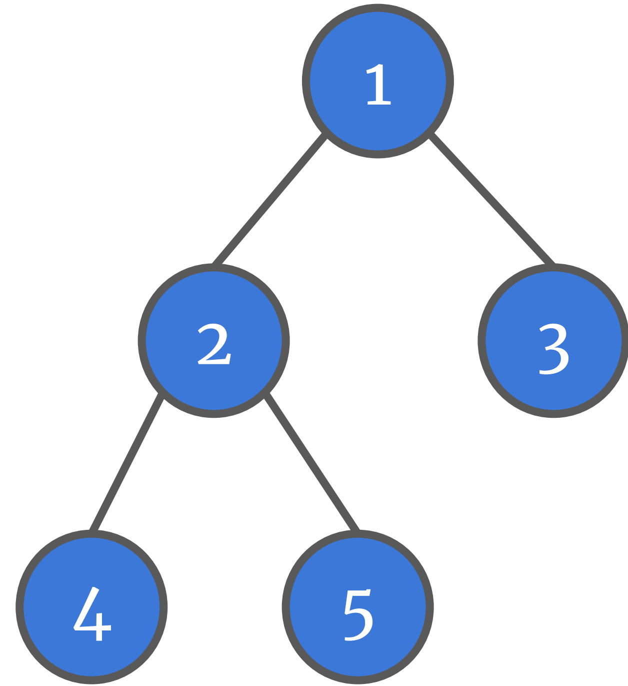


Traversals

- The only one that's really important in CS161 is the **inorder traversal**, which is defined recursively:
  - Do an inorder traversal of the left child, if any.
  - Give your own value.
  - Do an inorder traversal of the right node, if any.

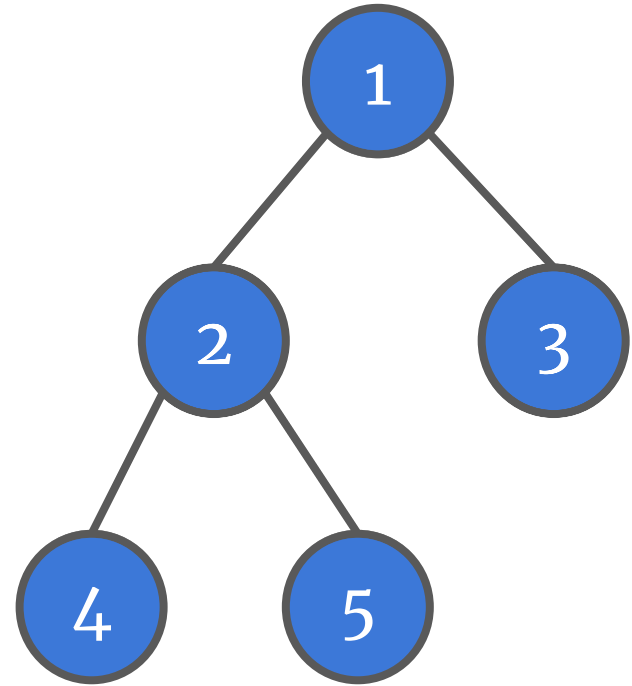


- Start at the root, 1.
- Traverse the left child:
    - Now the root is 2.
    - Traverse the left child:
        - Now the root is 4.
        - No left child.
        - **Print the root, 4.**
        - No right child.
    - **Print the root, 2.**
    - Traverse the right child:
        - Now the root is 5.
        - No left child.
        - **Print the root, 5.**
        - No right child.
- **Print the root, 1.**
- Traverse the right child:
    - Now the root is 3.
    - No left child.
    - **Print the root, 3.**
    - No right child.



Result: 4, 2, 5, 1, 3.

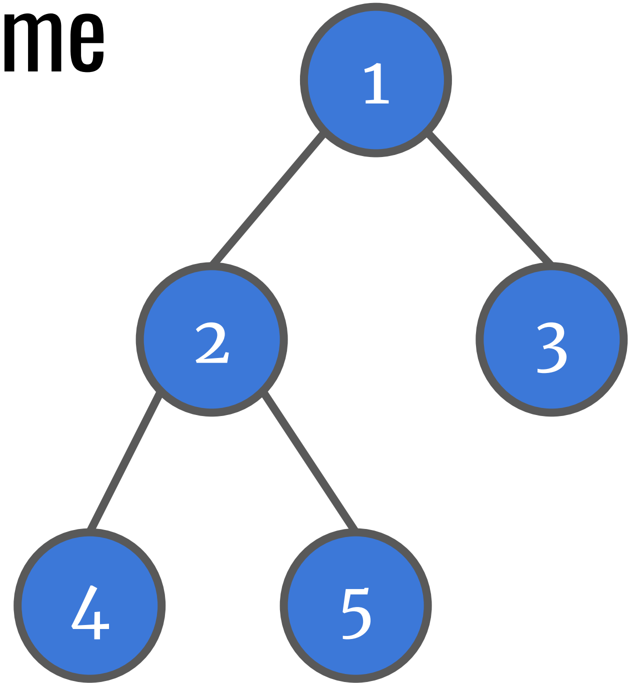
- Start at the root, 1.
- Traverse the left child:
    - Now the root is 2.
    - Traverse the left child:
        - Now the root is 4.
        - No left child.
        - **Print the root, 4.**
        - No right child.
    - **Print the root, 2.**
    - Traverse the right child:
        - Now the root is 5.
        - No left child.
        - **Print the root, 5.**
        - No right child.
- **Print the root, 1.**
- Traverse the right child:
    - Now the root is 3.
    - No left child.
    - **Print the root, 3.**
    - No right child.



Result: 4, 2, 5, 1, 3.
This matches the "left-to-right order".

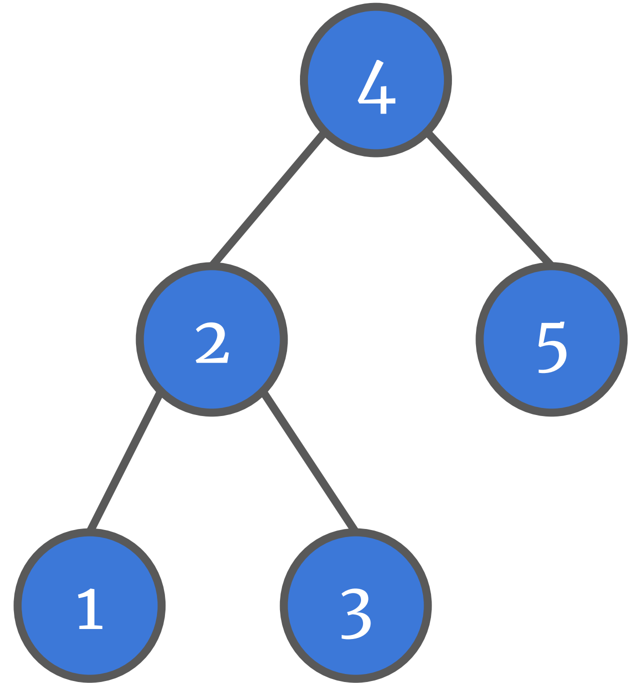
Inorder traversal: running time

- For each node, we visit it, travel to its left child (if any) and back, print the node's value, and travel to its right child (if any) and back.
- This is constant work per node, so: n * O(1) = O(n)
- This argument does not depend on the topology / balance of the tree!



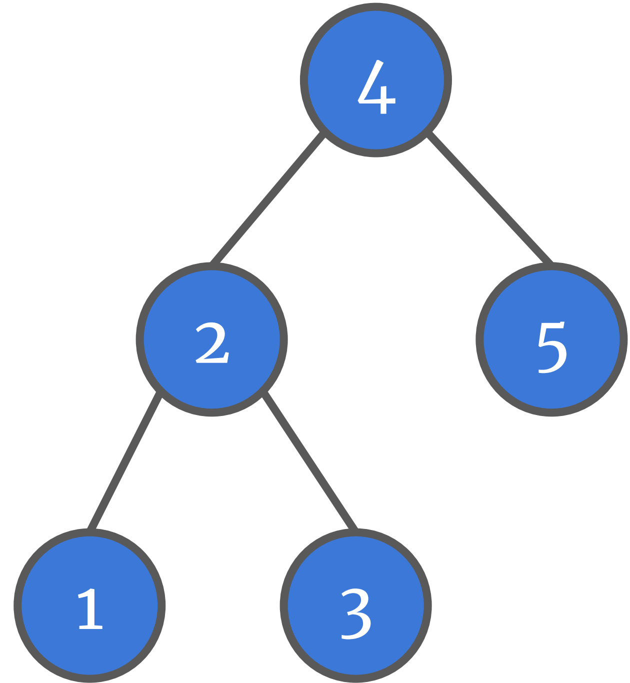
Binary search trees (BSTs)

- These are labeled binary trees such that:
  - the values of any node's **left child** (and all its descendants) are all **no greater** than the node's own value
  - the values of any node's **right child** (and all its descendants) are all **no smaller** than the node's own value



Binary search trees (BSTs)

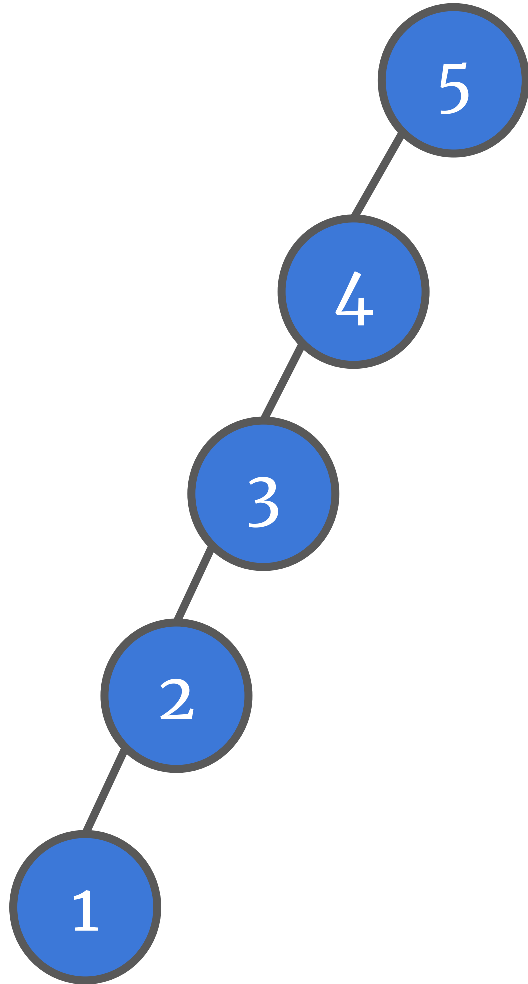
- These are labeled binary trees such that:
  - the values of any node's **left child** (and all its descendants) are all **no greater** than the node's own value
  - the values of any node's **right child** (and all its descendants) are all **no smaller** than the node's own value



An inorder traversal of a BST gives a sorted list of the values!

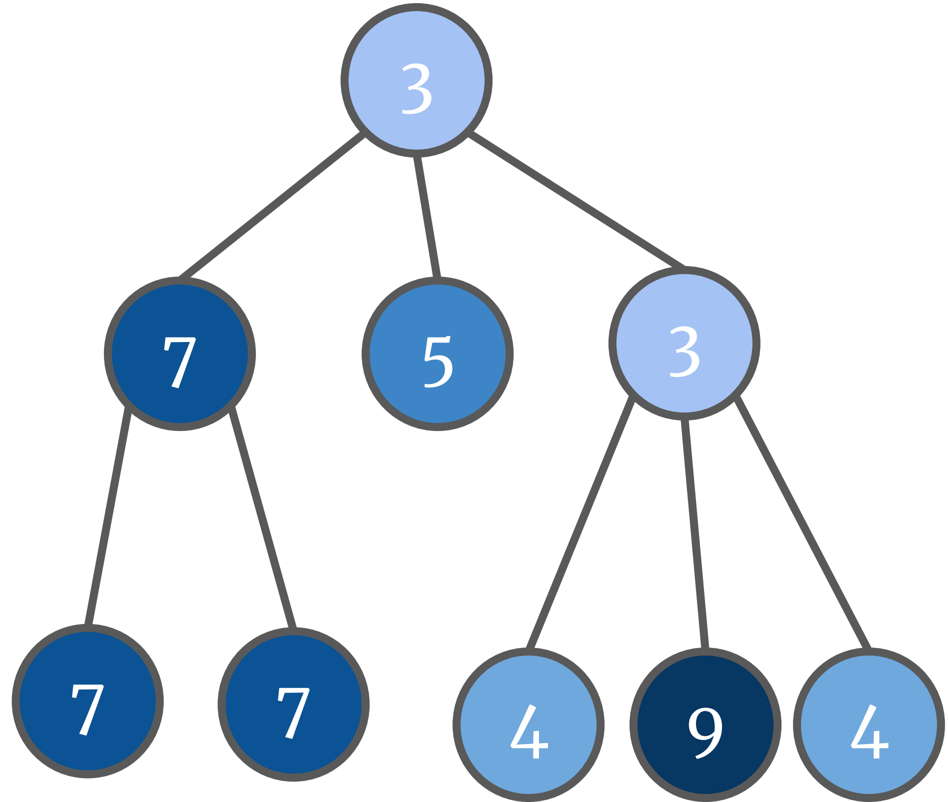
Careful:

- Not every labeled binary tree is a BST.
- BSTs are not necessarily well-balanced.
  - More on this in the second half...



And now: min-heaps!

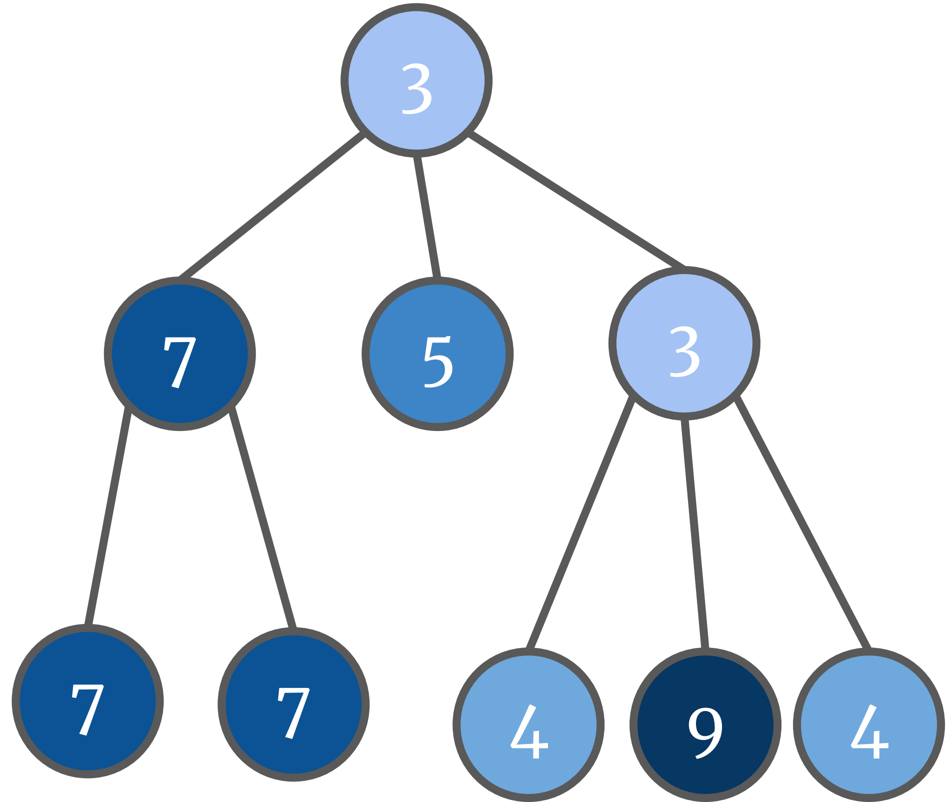
- Min-heaps are special labeled rooted trees with one simple property: Every node's value is no larger than the values of all its children.



And now: min-heaps!

- Min-heaps are special labeled rooted trees with one simple property: Every node's value is no larger than the values of all its children.

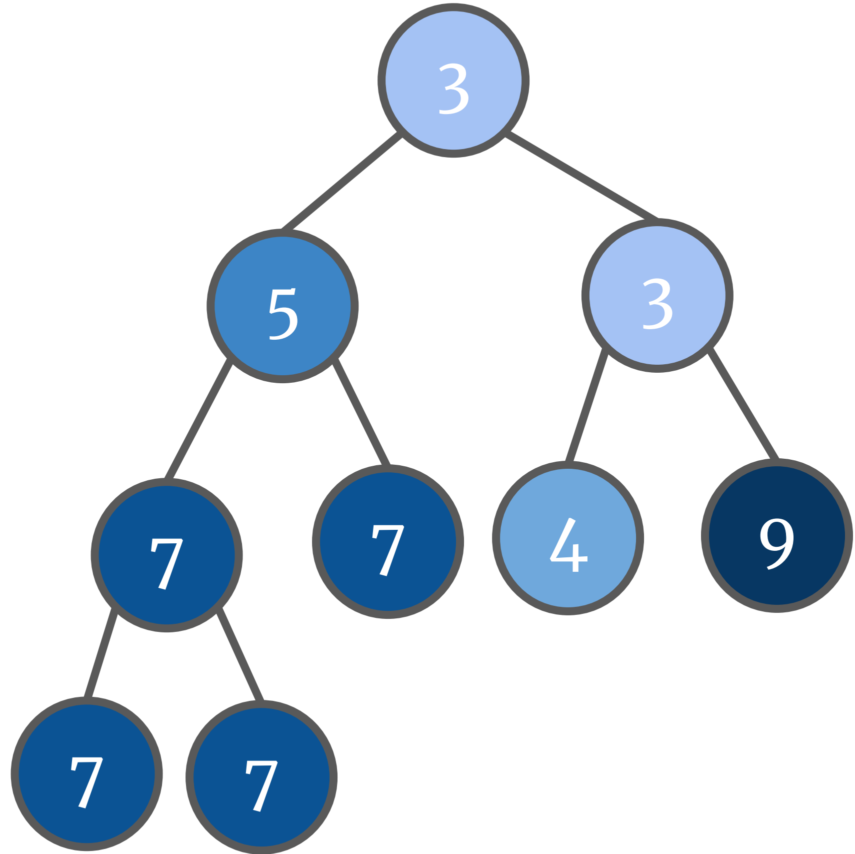
There is no requirement that min-heaps be binary or balanced, but let's enforce that.



Binary min-heaps!

Let's impose some requirements:

- Every level is full except maybe the bottom one...
  - in which case it is full up to a point.



What's the point of a min-heap?

- To easily find the minimum of a data set!

So what? That's easy.

What's the point of a min-heap?

- To easily find the minimum of a data set!

So what? That's easy.

- ...even as stuff gets deleted or inserted!

Why not just maintain a sorted list?

- If you use an **array**:
    - Finding where to insert an element is easy using binary search.
    - Inserting into or deleting from an array can be awful – you may have to shift a lot of values over to make room, and/or resize the array.

Why not just maintain a sorted list?

- If you use a **linked list**:
    - Inserting into or deleting from a (doubly) linked list is easy. Just rewire a few pointers.
    - Finding where to insert a new element can be awful – you can't binary search, only traverse the linked list in order!

Why not just maintain a sorted list?

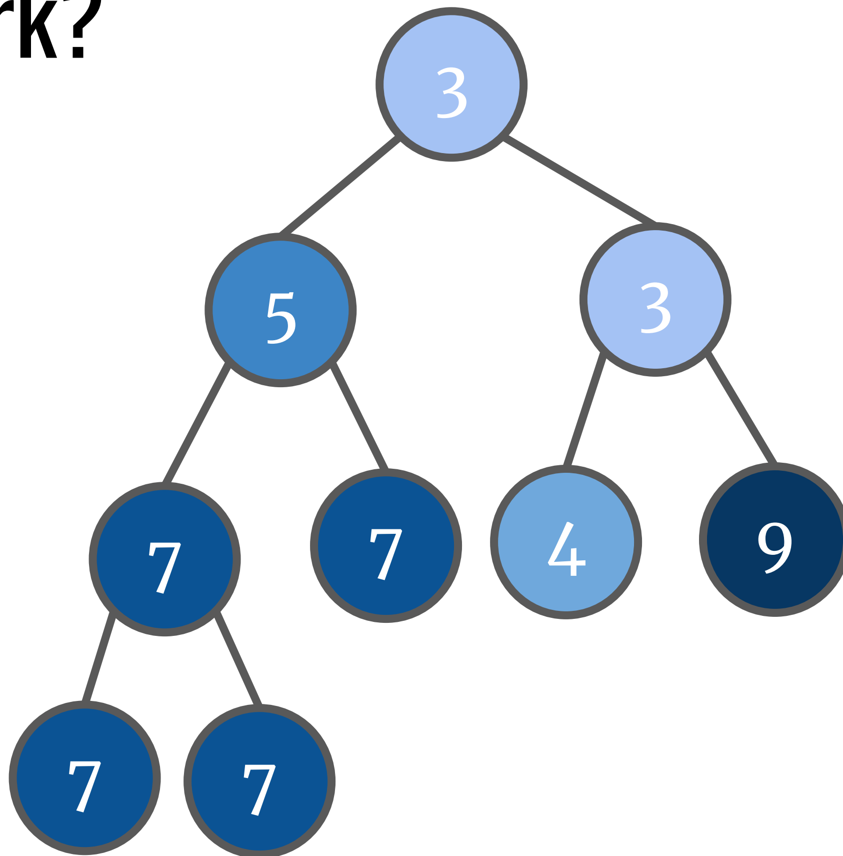
- If you use a **linked list**:
  - Inserting into or deleting from a (doubly) linked list is easy. Just rewire a few pointers.
  - Finding where to insert a new element can be awful – you can't binary search, only traverse the linked list in order!

Can you think of a way to augment a linked list to do this? Look up "skip lists" if you're curious!

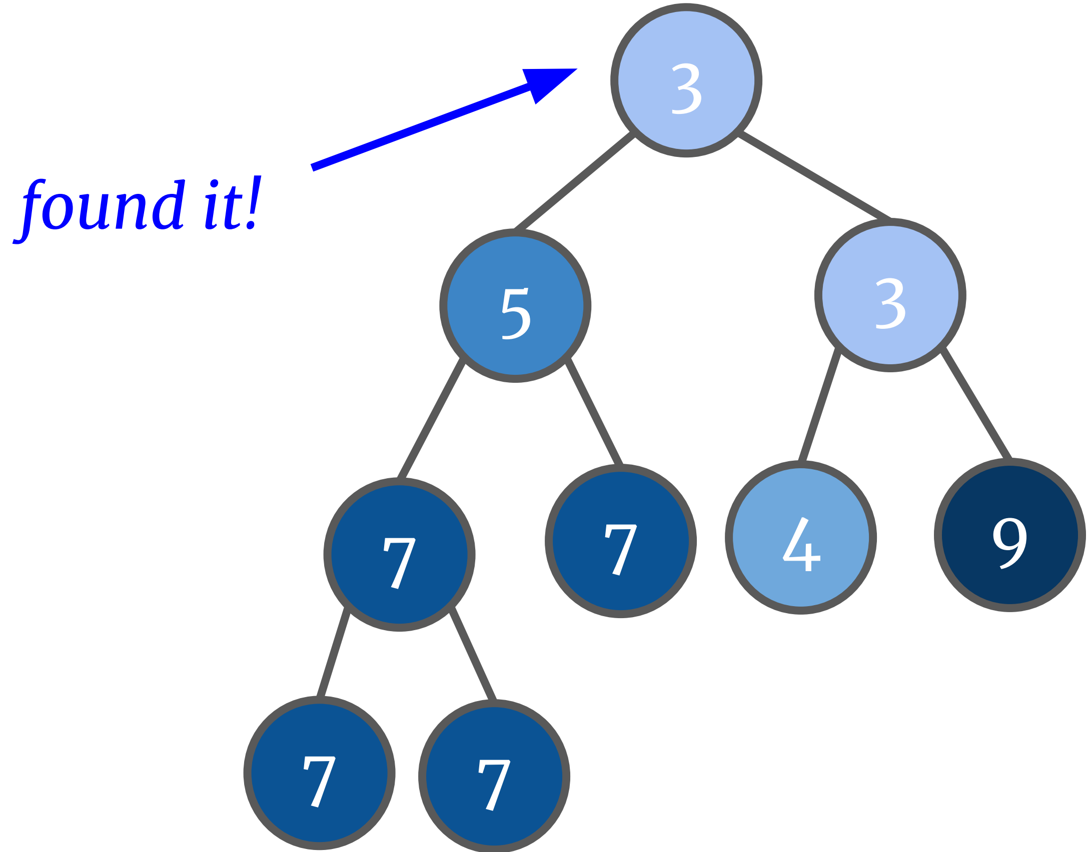
How do min-heaps work?

We need to support three operations:

- Find-Min
- Insertion
- Delete-Min



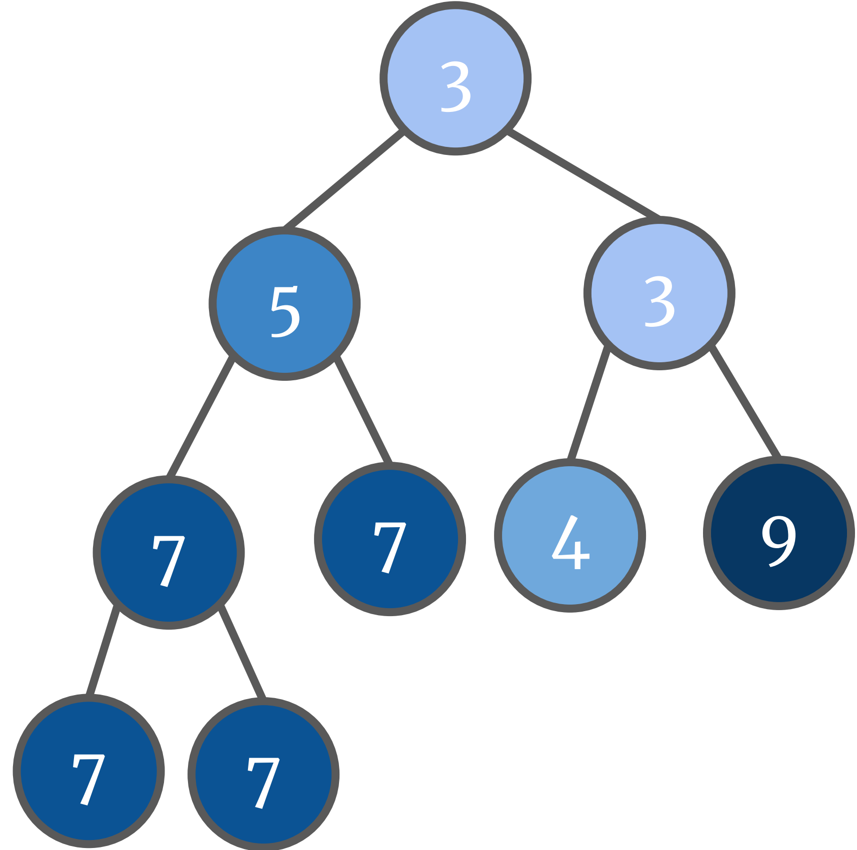
Find-Min is O(1)



Insertion

Suppose we want to insert a new value.

We want to maintain the shape of the tree...

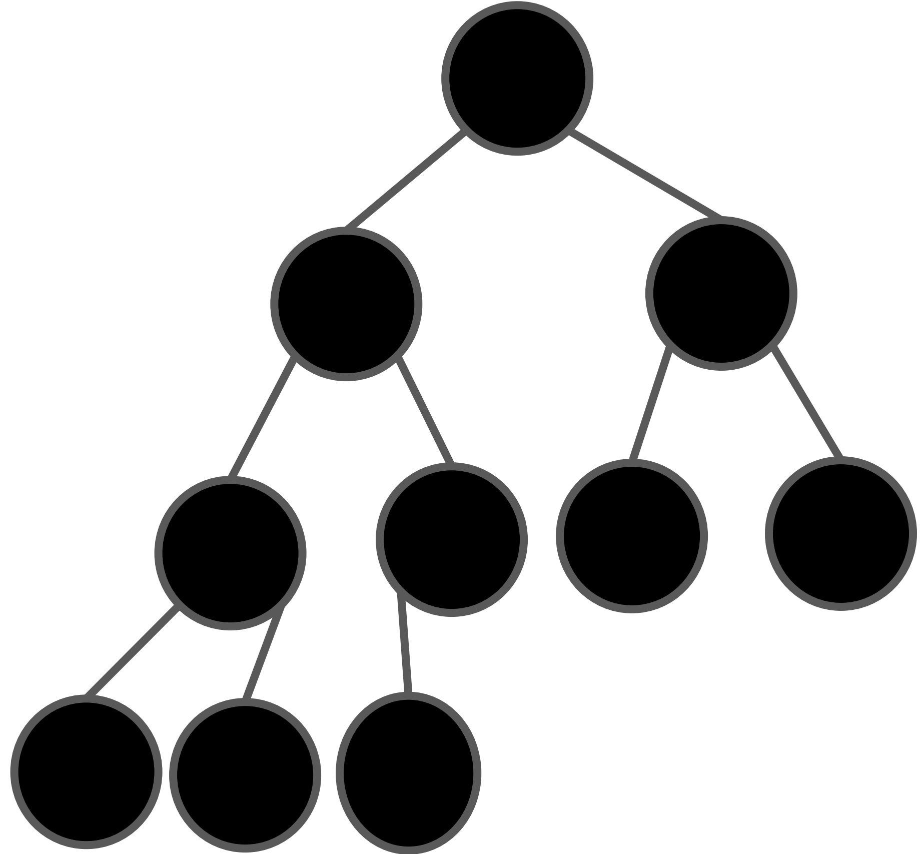


Insertion

Suppose we want to insert a new value.

We want to maintain the shape of the tree...

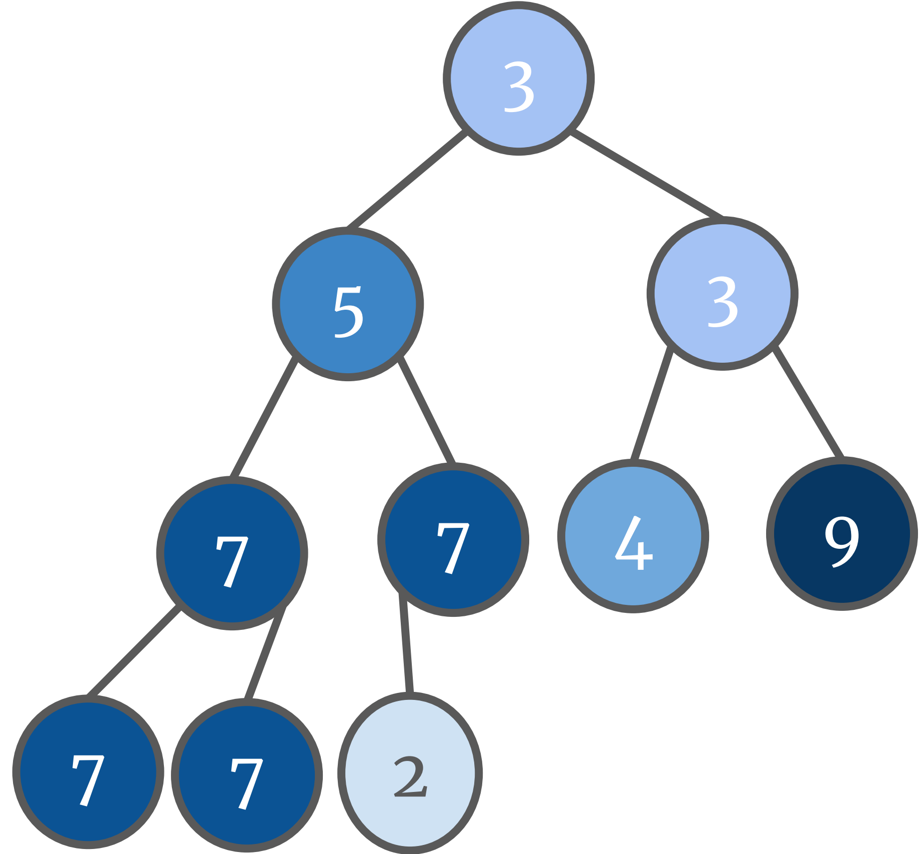
...so the tree should have this shape when we're done.



Insertion

Say our new value is 2.

Let's start by putting it in the next available spot.

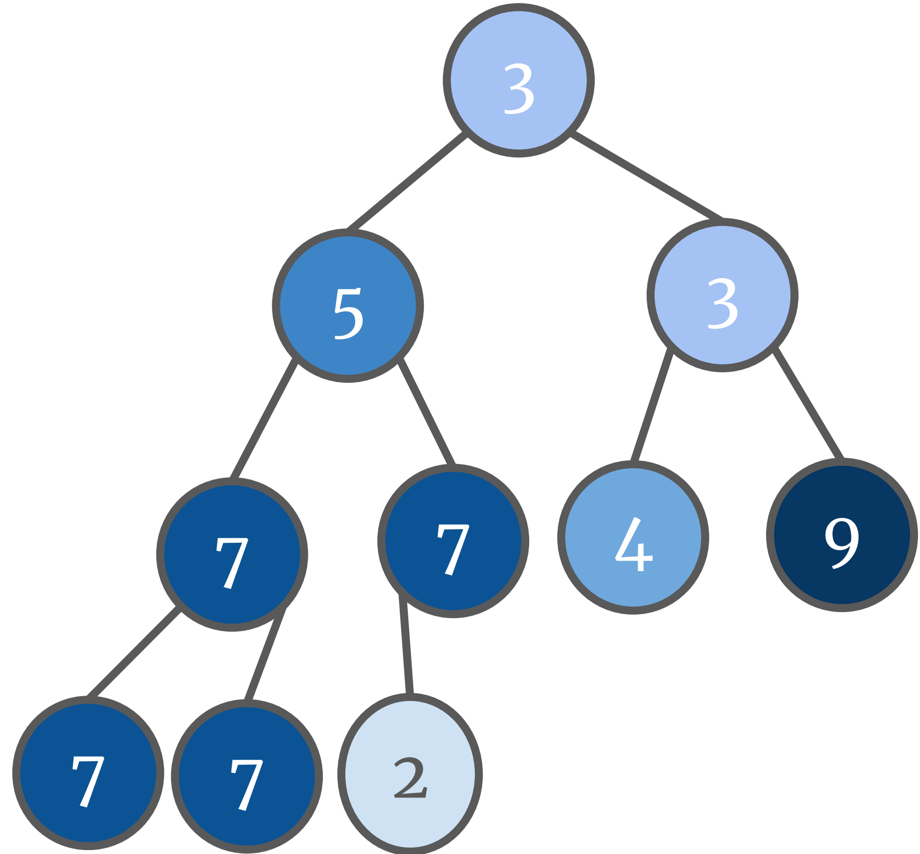


Insertion

Say our new value is 2.

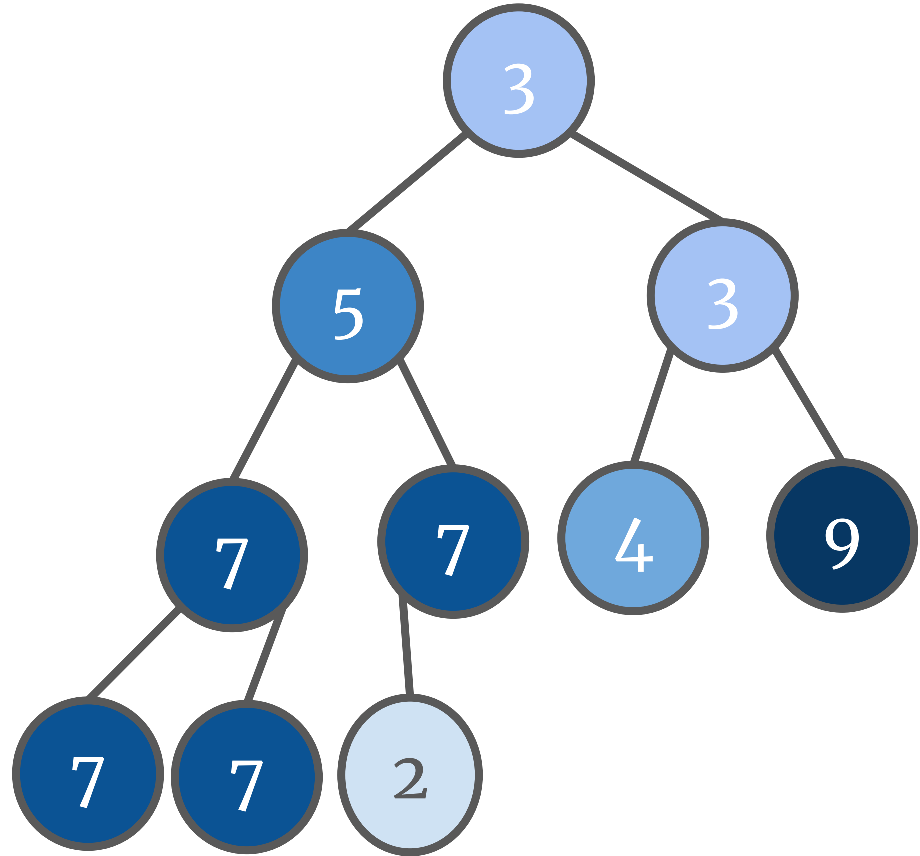
Let's start by putting it in the next available spot.

But there's a problem! The heap property is violated.



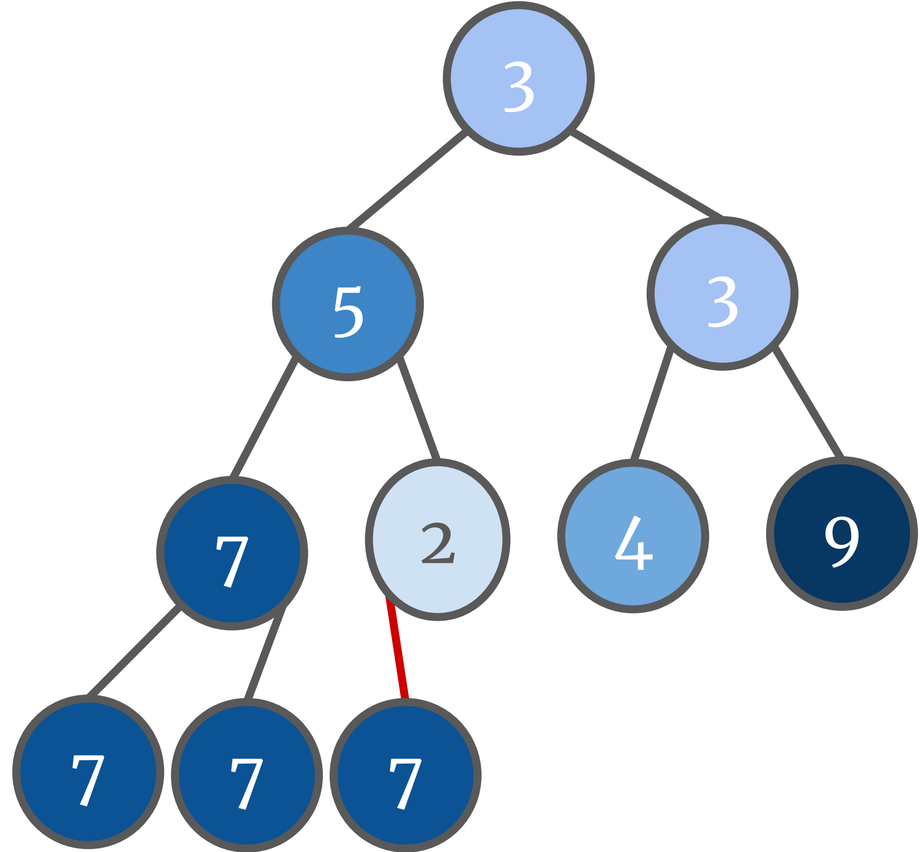
Insertion

A fix: As long as the newly inserted value is smaller than its parent, swap it with its parent.



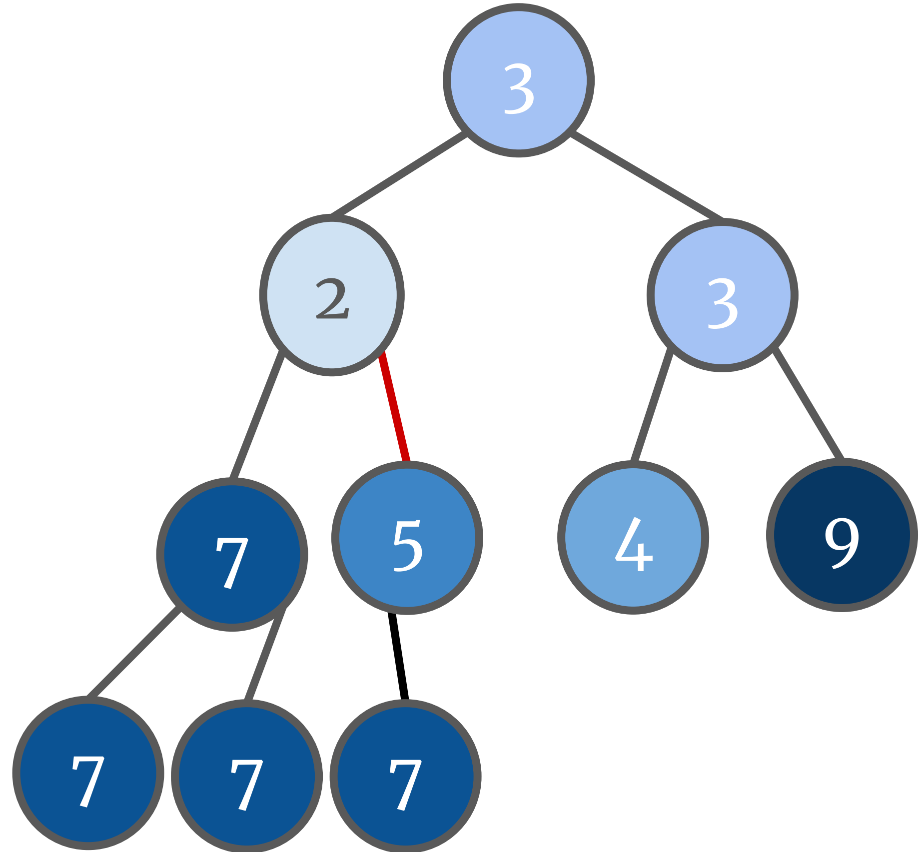
Insertion

A fix: As long as the newly inserted value is smaller than its parent, swap it with its parent.



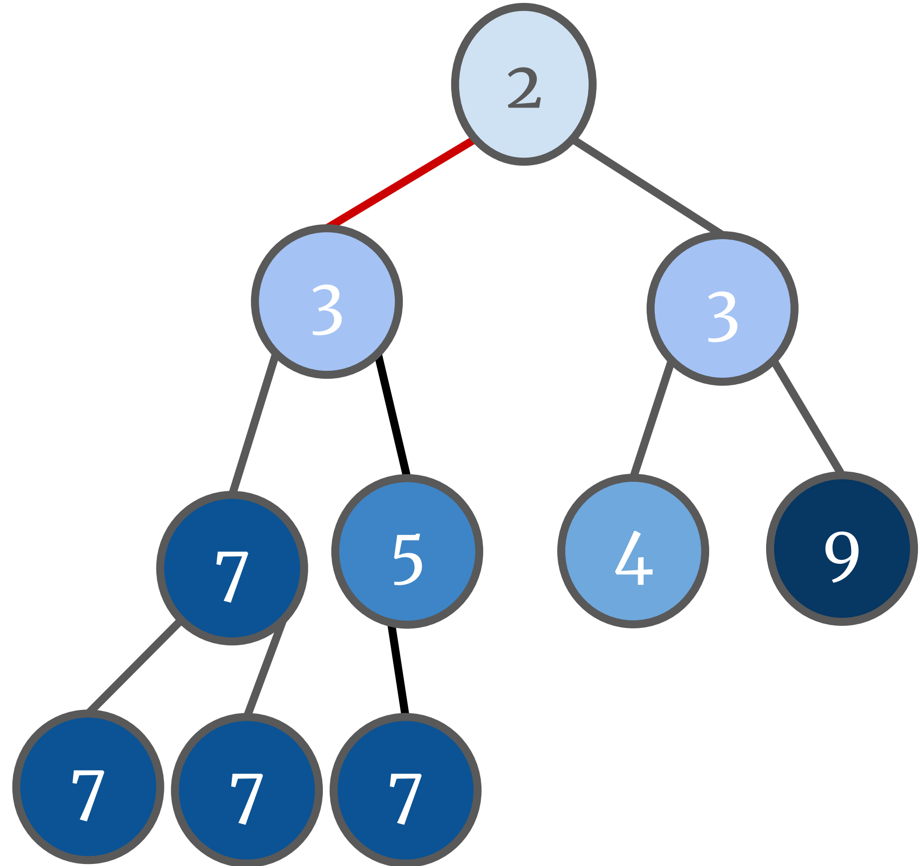
Insertion

A fix: As long as the newly inserted value is smaller than its parent, swap it with its parent.



Insertion

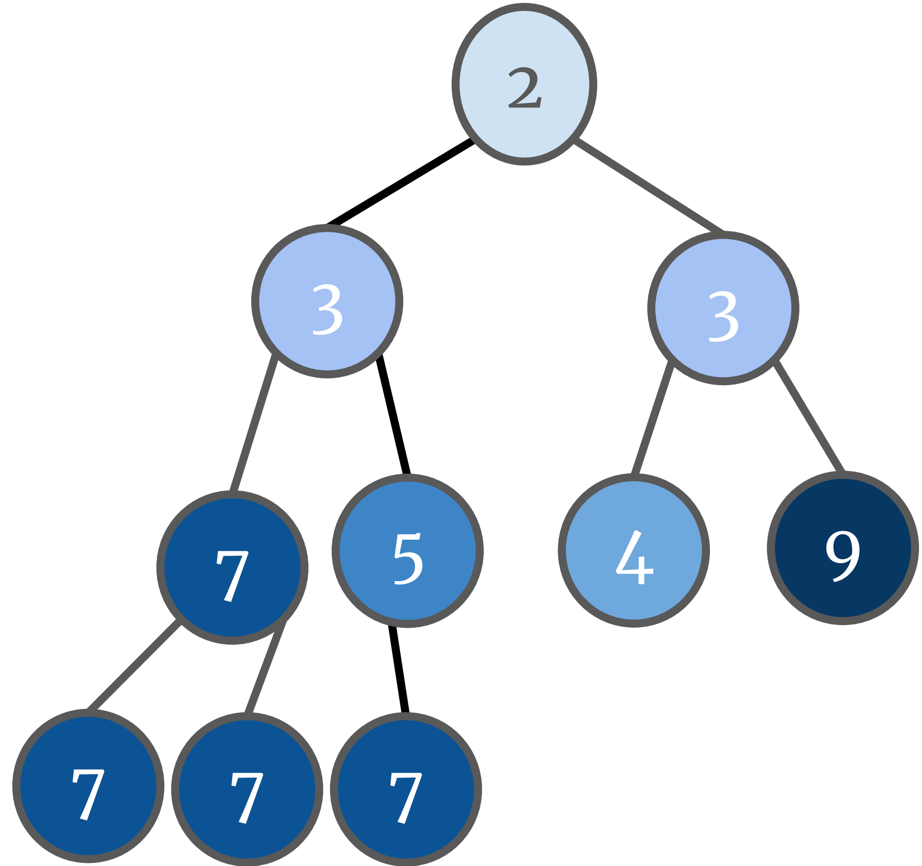
A fix: As long as the newly inserted value is smaller than its parent, swap it with its parent.



Insertion

A fix: As long as the newly inserted value is smaller than its parent, swap it with its parent.

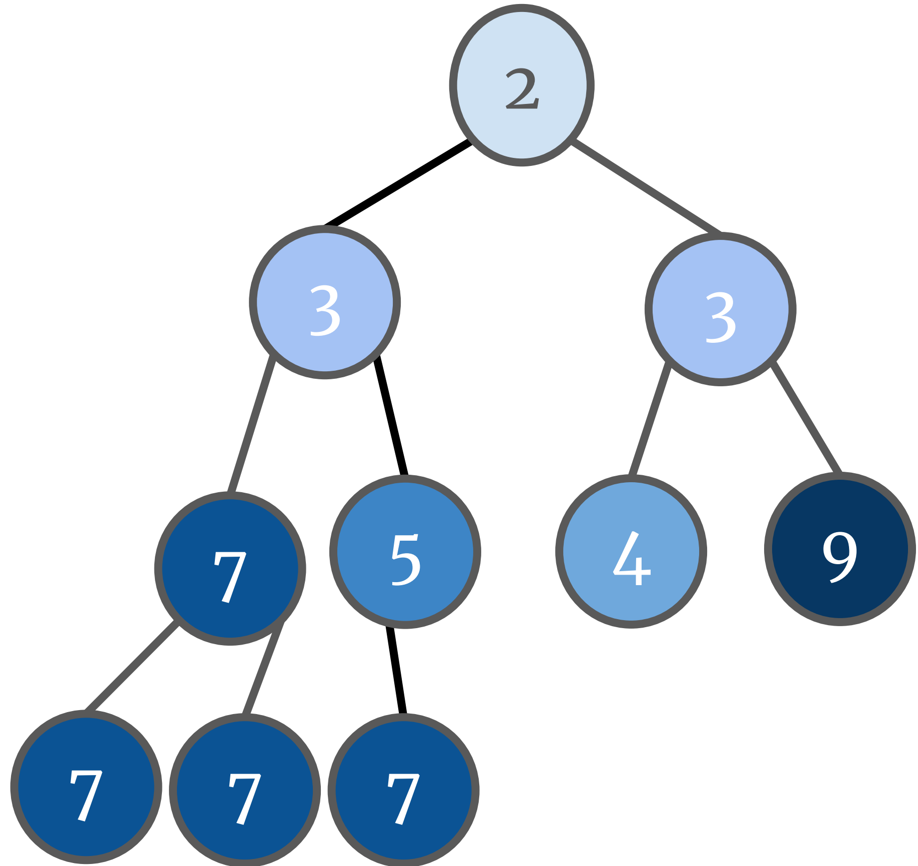
All fixed!



Insertion is O(log n)

Assuming we can find the next position in the heap in O(1) time...

we only need to go from a leaf to the root, swapping nodes. The height is O(log n), so this is O(log n).

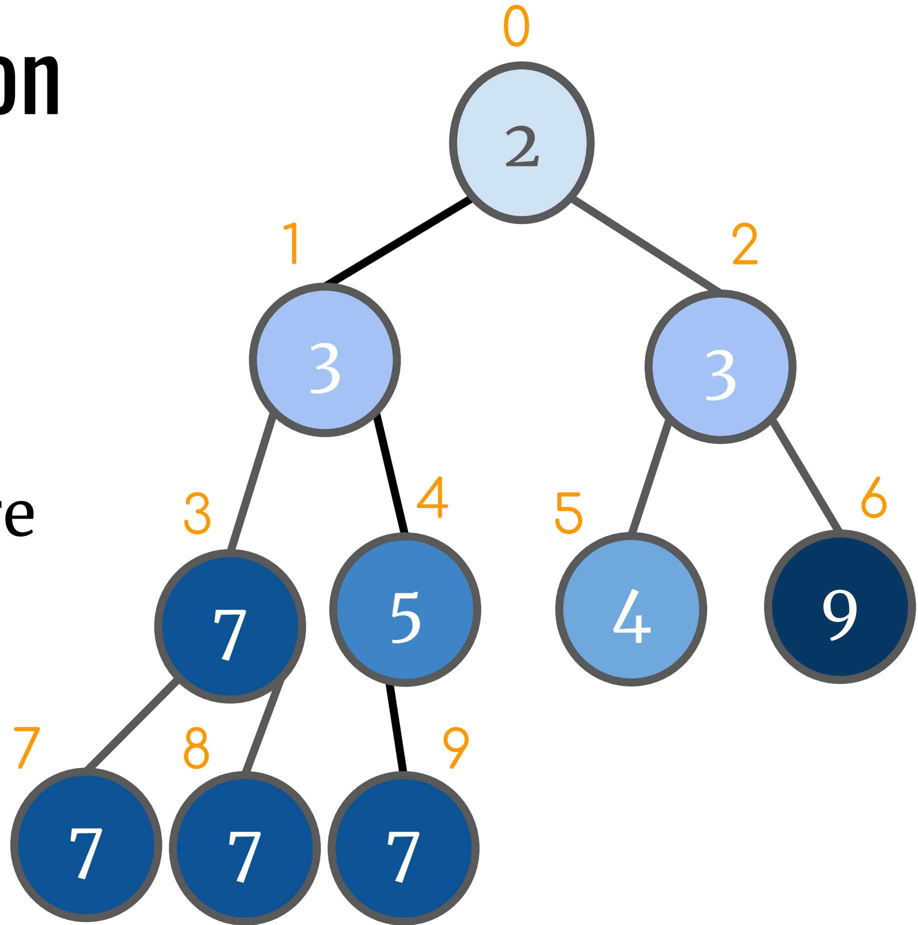


Aside: implementation

This works nicely as an array!

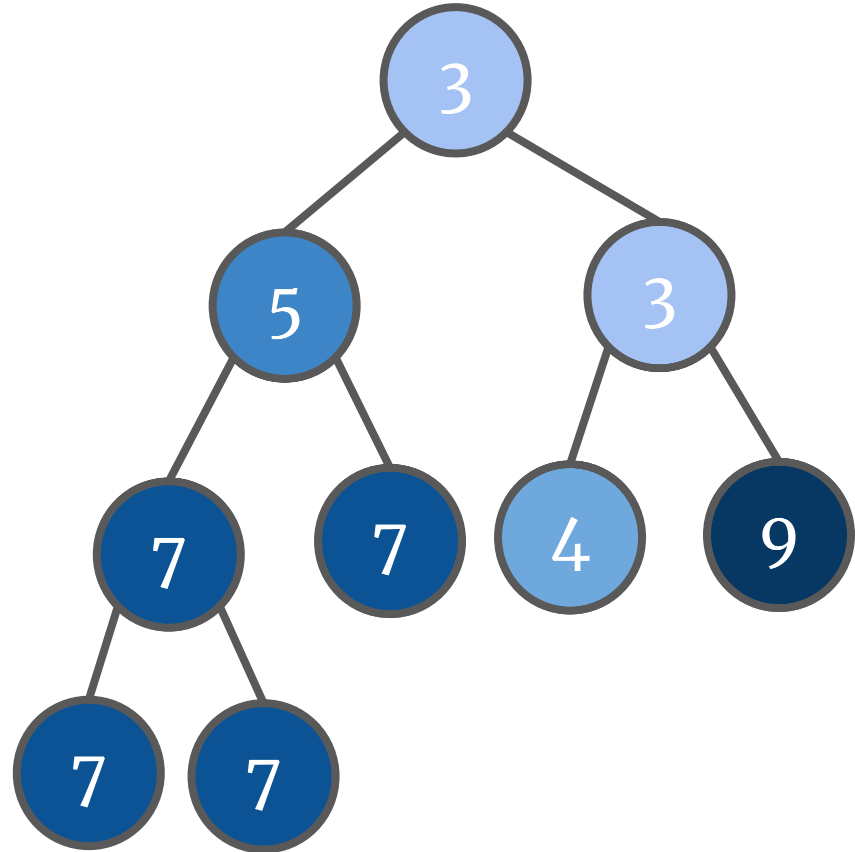
The children of node i are at indices $2i+1$ and $2i+2$.

We keep a pointer to the next open slot.



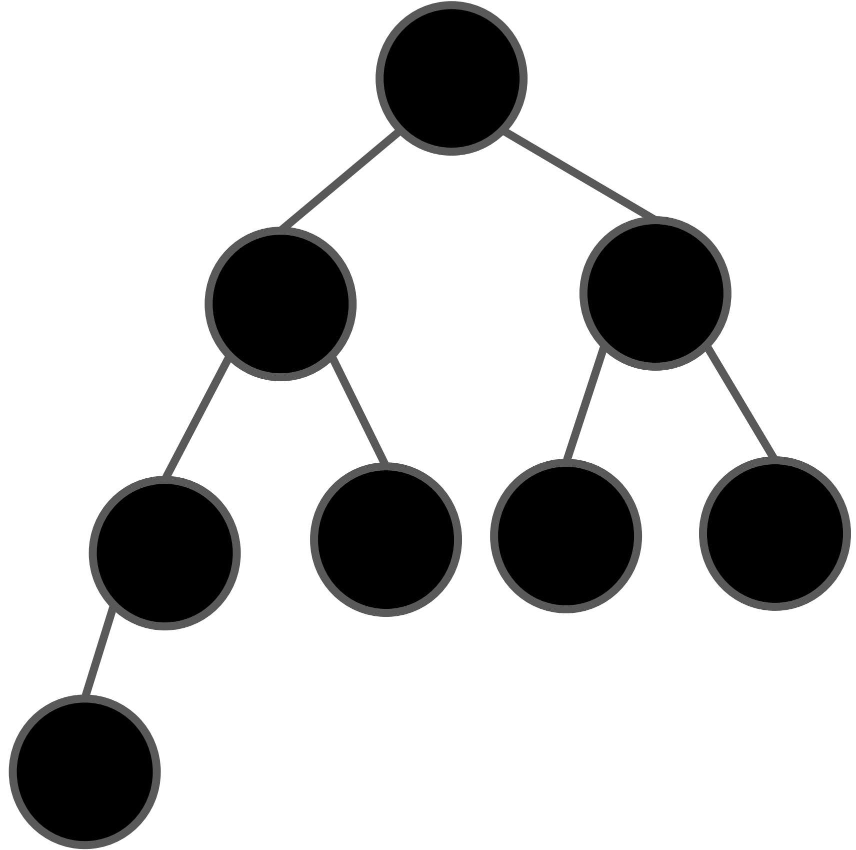
Delete-Min

- This kind of heap best supports deleting the minimum, not an arbitrary element.
  - *(you can, but we won't dwell on it)*



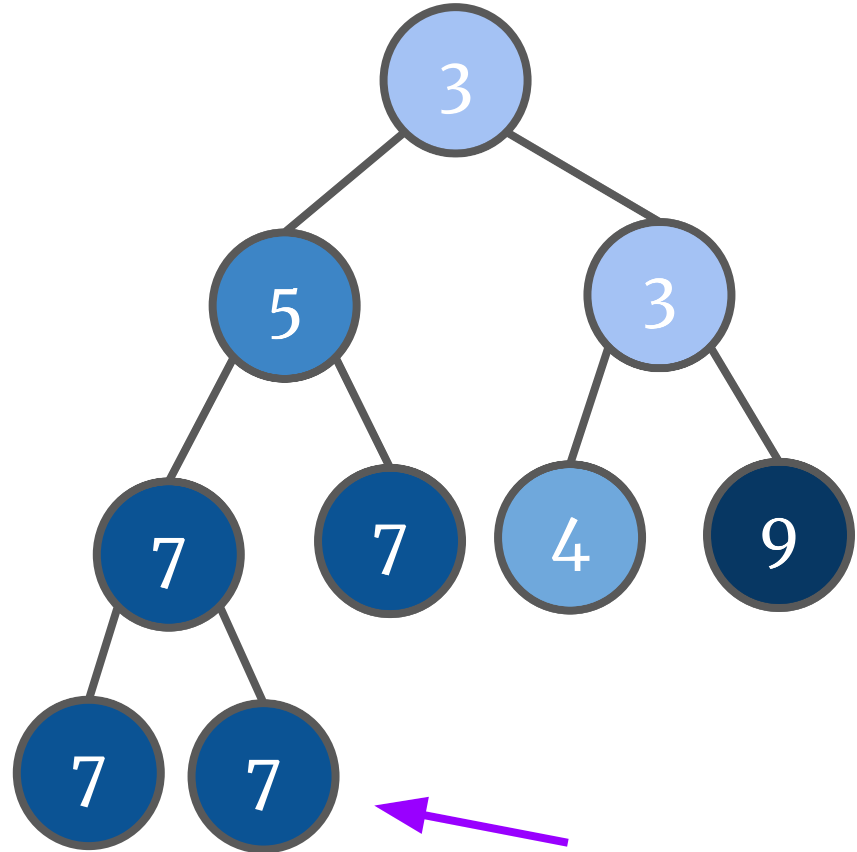
Delete-Min

- Again, we know what kind of shape we want the final result to have.



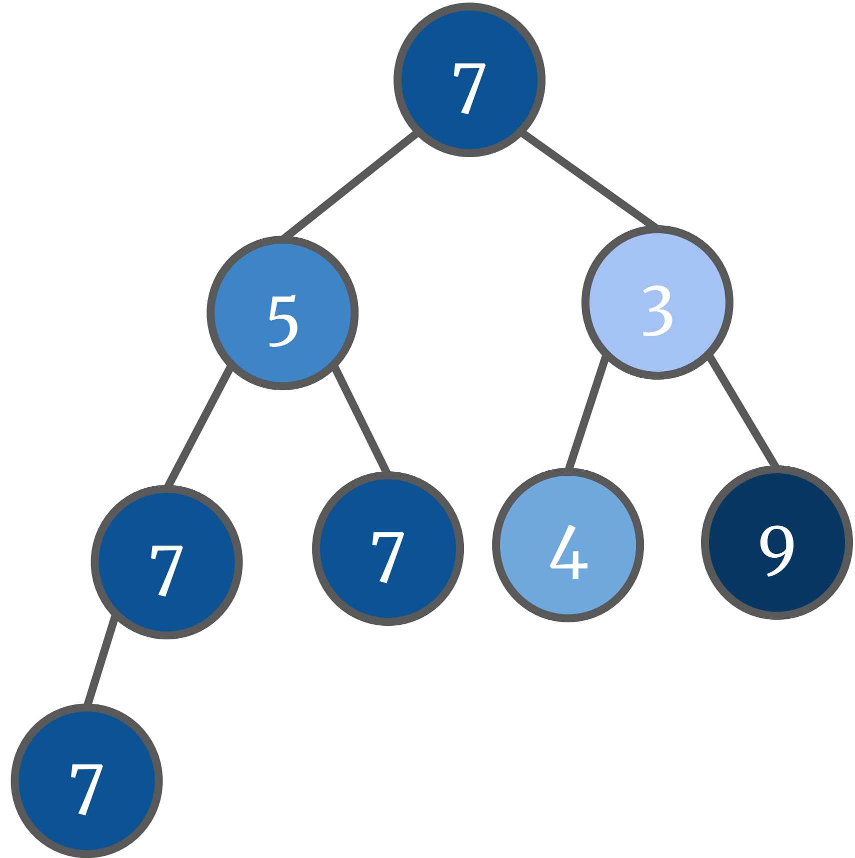
Delete-Min

- Remove the element in the last position and put it in the root.



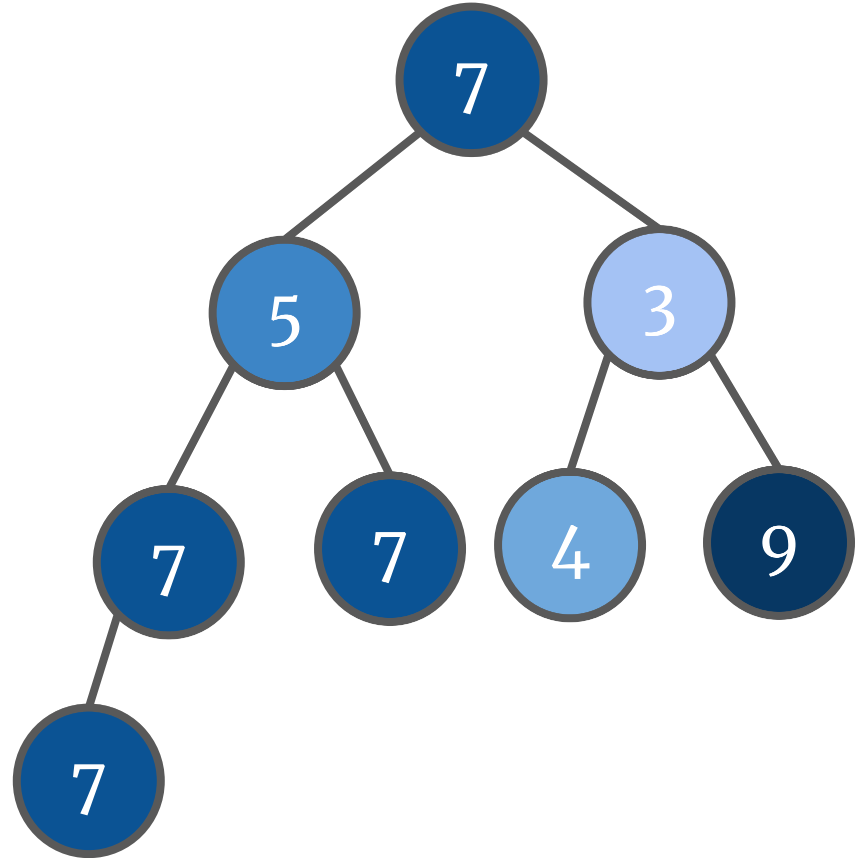
Delete-Min

- Remove the element in the last position and put it in the root.



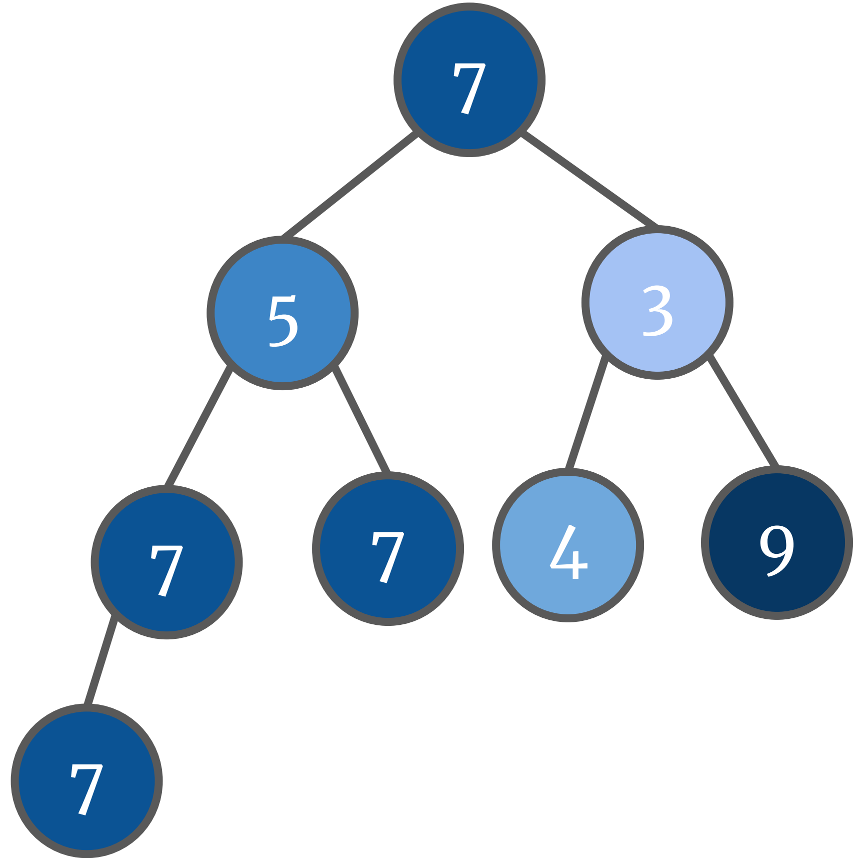
Delete-Min

- Remove the element in the last position and put it in the root.
- *How can we fix the heap property now?*



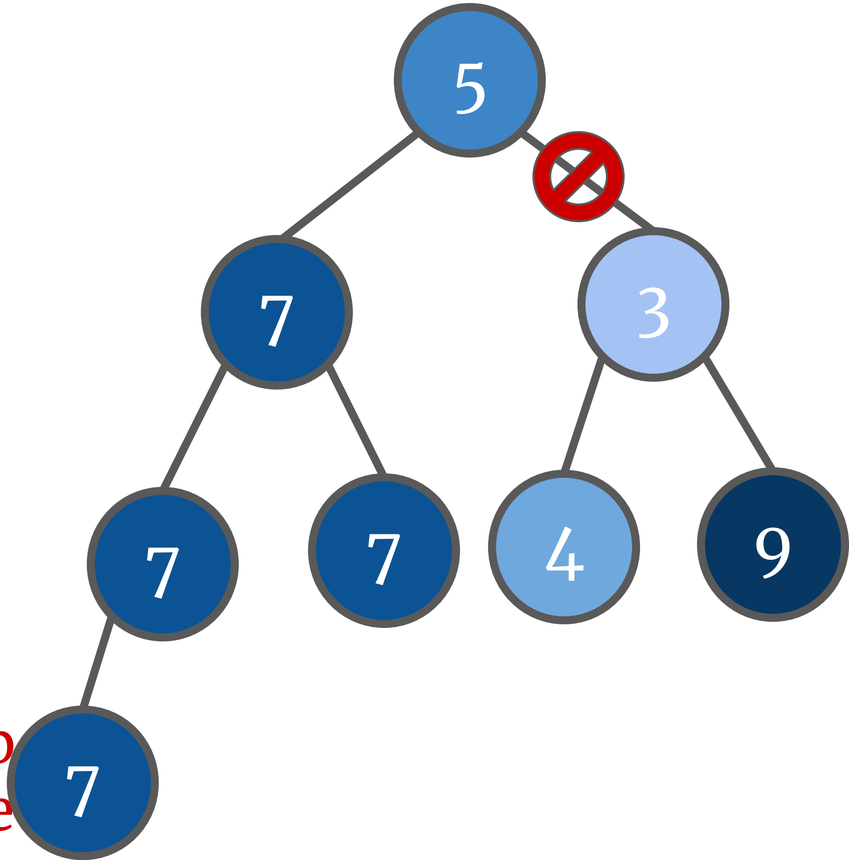
Delete-Min

- Remove the element in the last position and put it in the root.
- We should swap the root with one of the two values below...



Delete-Min

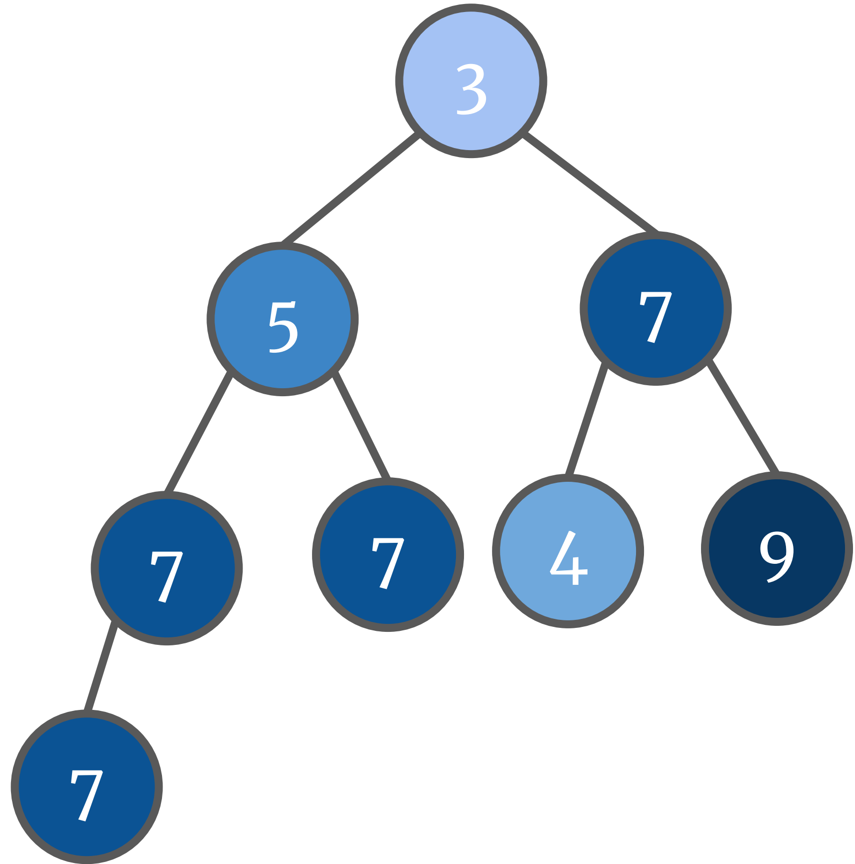
- Remove the element in the last position and put it in the root.
- We should swap the root with one of the two values below...
  - If we chose 5, the heap property would still be broken.



Delete-Min

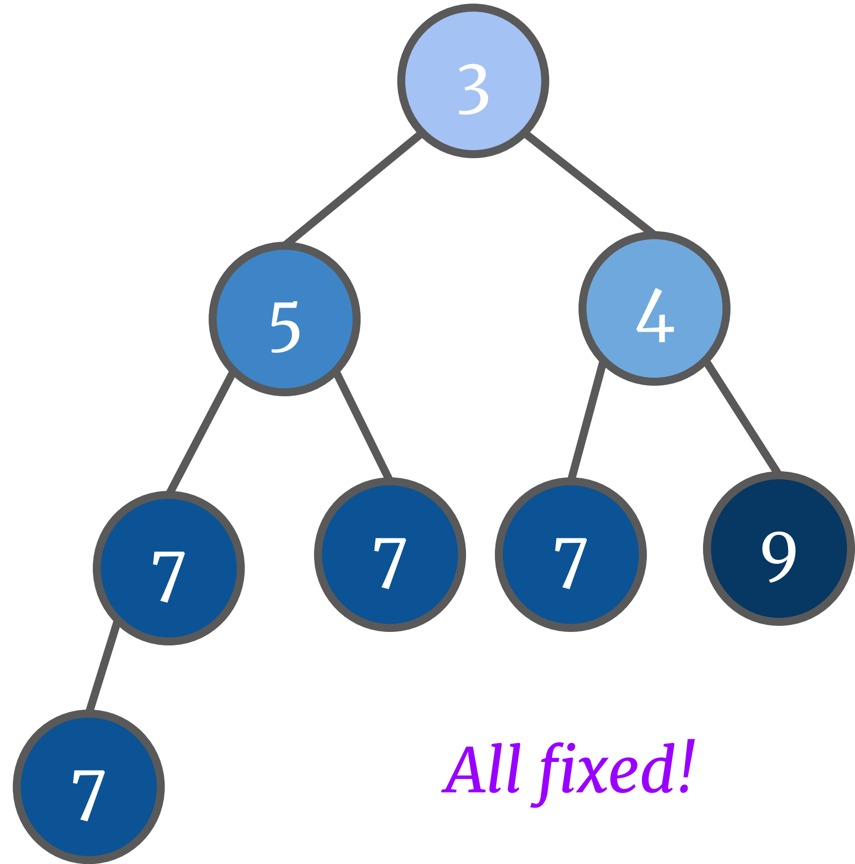
- Remove the element in the last position and put it in the root.

- We should swap the root with one of the two values below...
  - specifically, the smallest (if any).



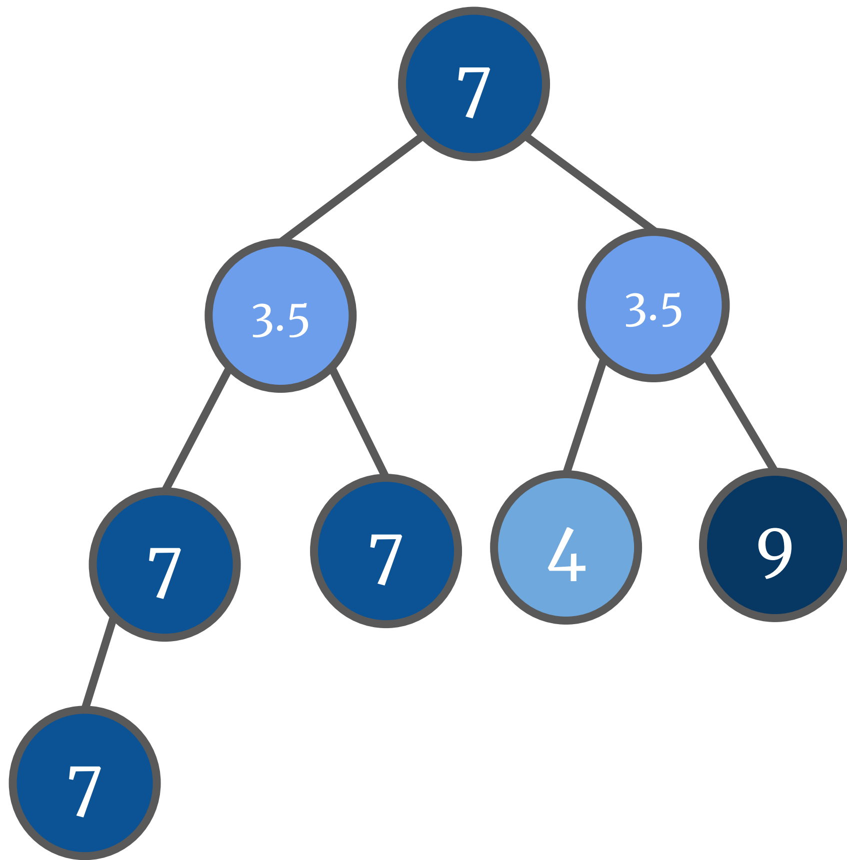
Delete-Min

- Remove the element in the last position and put it in the root.
- We should swap the root with one of the two values below...
  - specifically, the smallest (if any).
  - And so on.



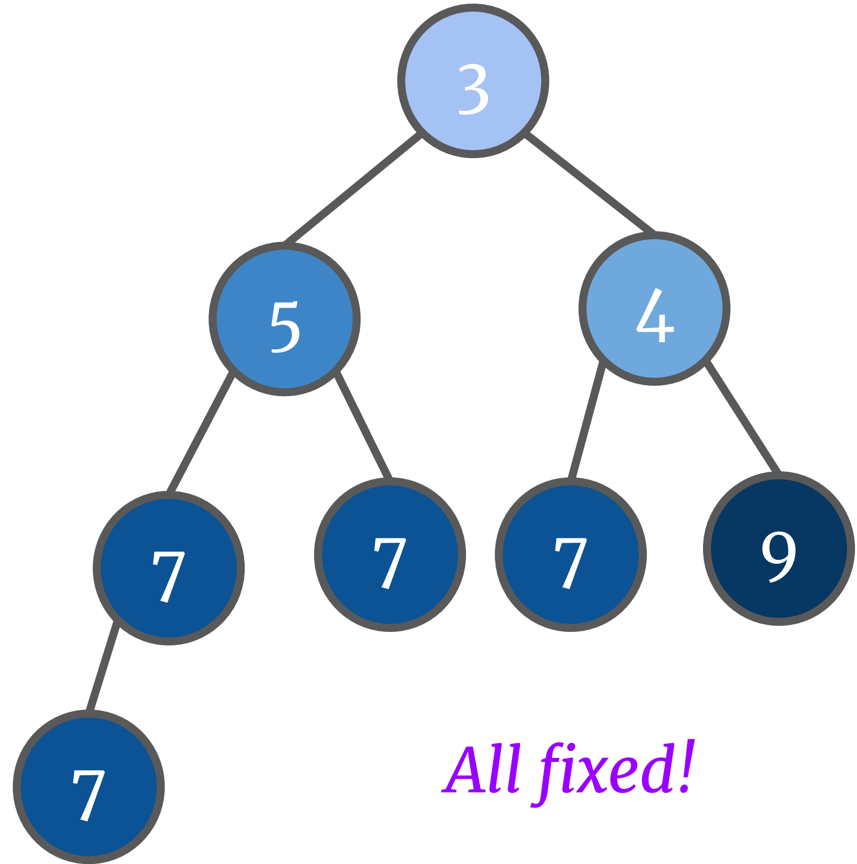
What if there had been a tie?

- Then it doesn't matter which one we pick!
  - There's no way to make a wrong choice.
  - *(Maybe the clump of 7s is not great, but the algorithm can't know this when it decides...)*



Deletion is O(log n)

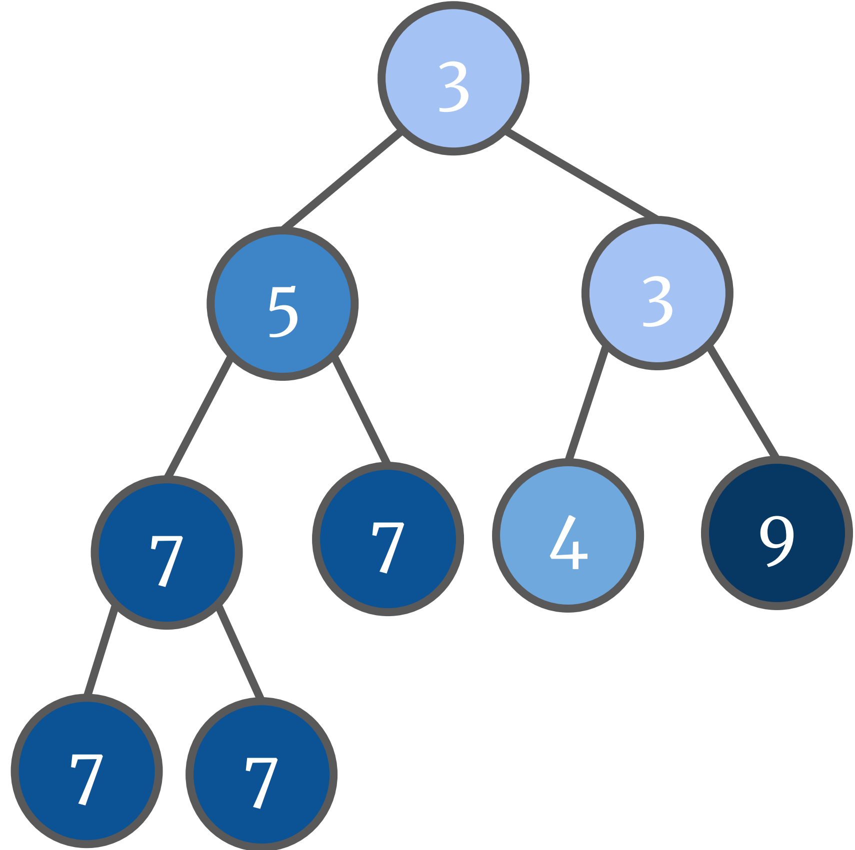
Basically the same argument as for insertion!



Final score

Running times of our three operations:

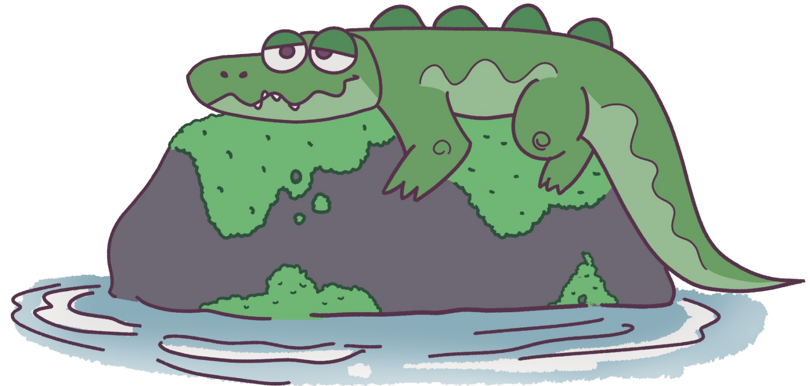
- Find-Min: $O(1)$
- Insertion: $O(\log n)$
- Delete-Min: $O(\log n)$



Unsurprisingly, there are also max heaps

They work the same way!

The default implementation of heaps (e.g., Python's `heapq`) tends to be a min-heap, but one super-lazy way to make it a max-heap is to just use the negative of all the values..



Deletion of non-min elements

- You *can* do the same sort of thing as when deleting the root: replace the deleted element with the "last" element of the heap, then swap either up or down as needed.

- Or, fake deletion: just leave stale stuff in the heap, and use a hash table to keep track of which items are stale. When a new min surfaces, check the hash table to see whether to delete it. Lazy, but may be OK if not too much stale stuff builds up!

Hey, we get a free sort: Heapsort!

- To sort a list of size n:
  - Stuff all the elements into the heap, one by one. — n * O(log n)
  - Delete-min, one by one – they come out in sorted order! — n * O(log n)

O(n log n)

Hey, we get a free sort: Heapsort!

- To sort a list of size n:
  - Stuff all the elements into the heap, one by one. $n * O(\log n)$
  - Delete-min, one by one – they come out in sorted order! $n * O(\log n)$

Notice that the n in question is a loose upper bound here... early insertions / later deletions are cheaper!

$O(n \log n)$

The best part of heaps: Priority queues!

- A *priority queue* is a data structure that efficiently identifies the item with highest priority.

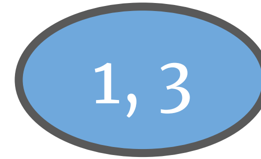
- A heap is a natural way to implement a priority queue.

Homework 1, Problem 6 can be solved this way!

- We did not expect you to use priority queues – we hadn't even gotten to them yet! – but it is possible.
- The idea:
  - Insert all students' (enter, exit) pairs into the heap.
  - When we delete-min, reinsert a pair saying when the student will exit.
  - When a student exits, increment our total number of pairs by the number of students in the room.

(1, 3), (5, 6), (3, 7), (2, 4), (1, 2)

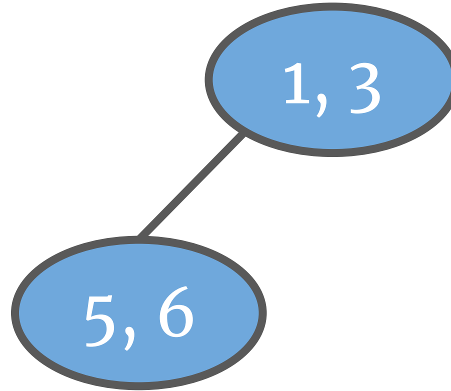
(1, 3), (5, 6), (3, 7), (2, 4), (1, 2)



Insert (1, 3)

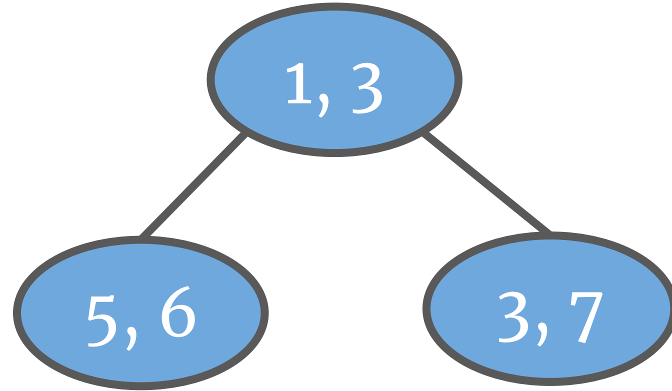
(1, 3), (5, 6), (3, 7), (2, 4), (1, 2)

Insert (5, 6)



(1, 3), (5, 6), (3, 7), (2, 4), (1, 2)

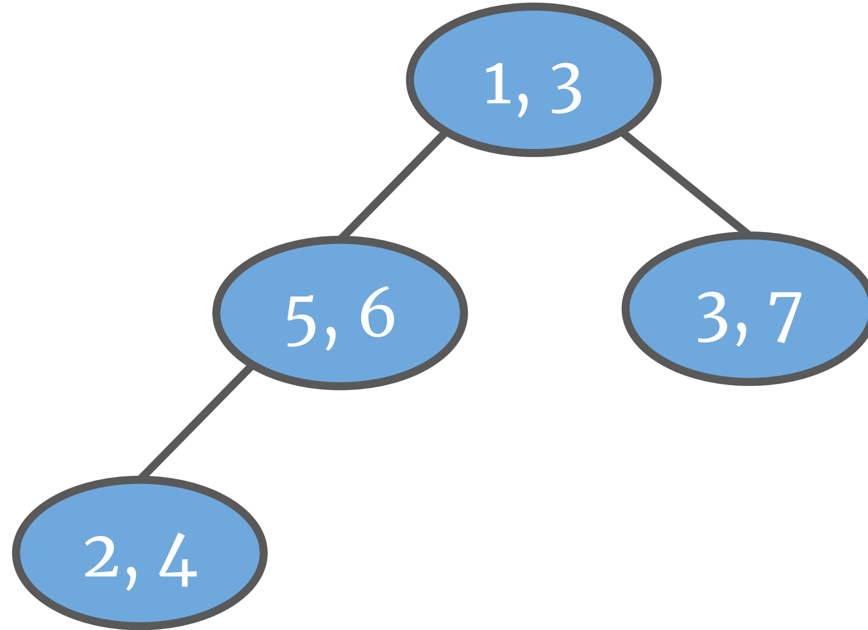
Insert (3, 7)



(1, 3), (5, 6), (3, 7), (2, 4), (1, 2)

Insert (2, 4)

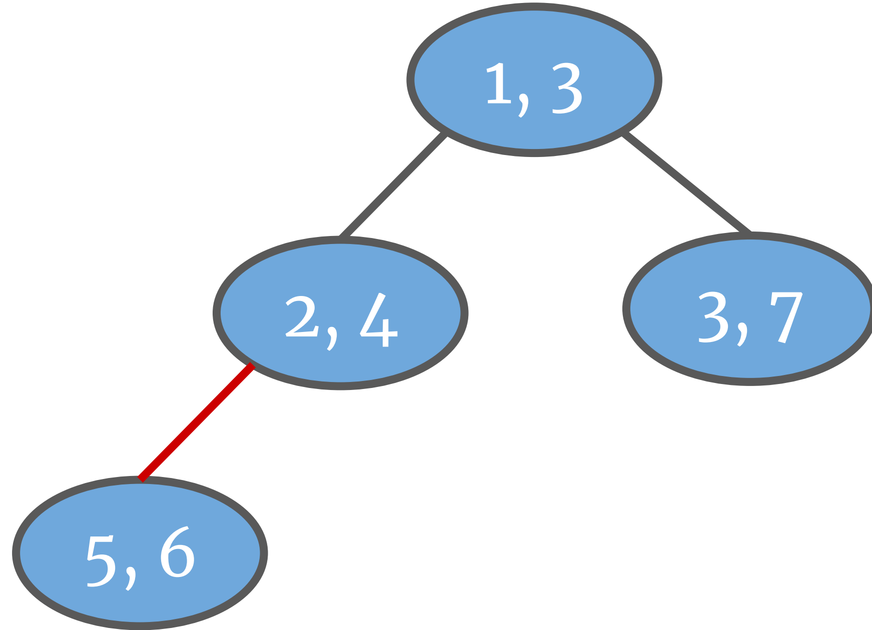
now the heap is sad!



(1, 3), (5, 6), (3, 7), (2, 4), (1, 2)

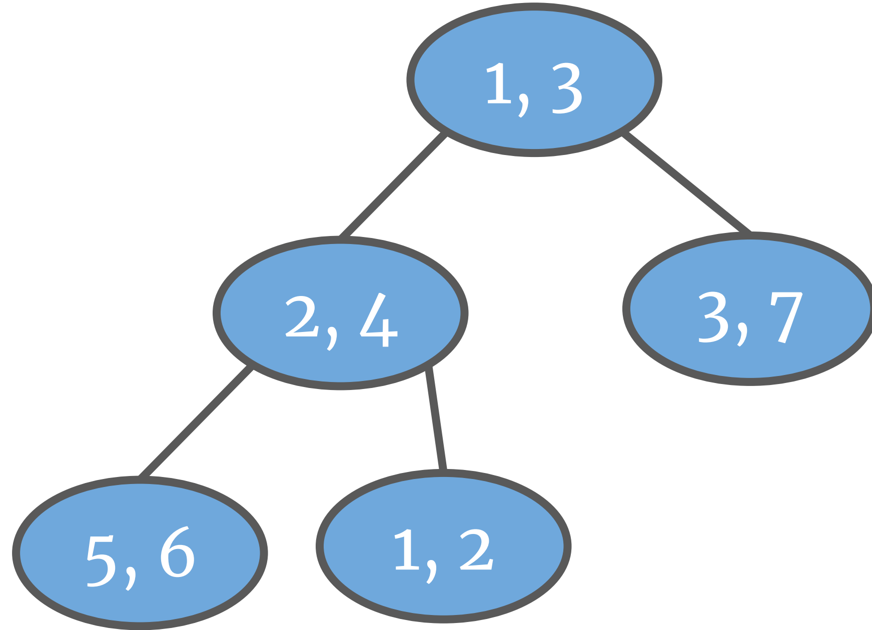
Insert (2, 4)

fixed!



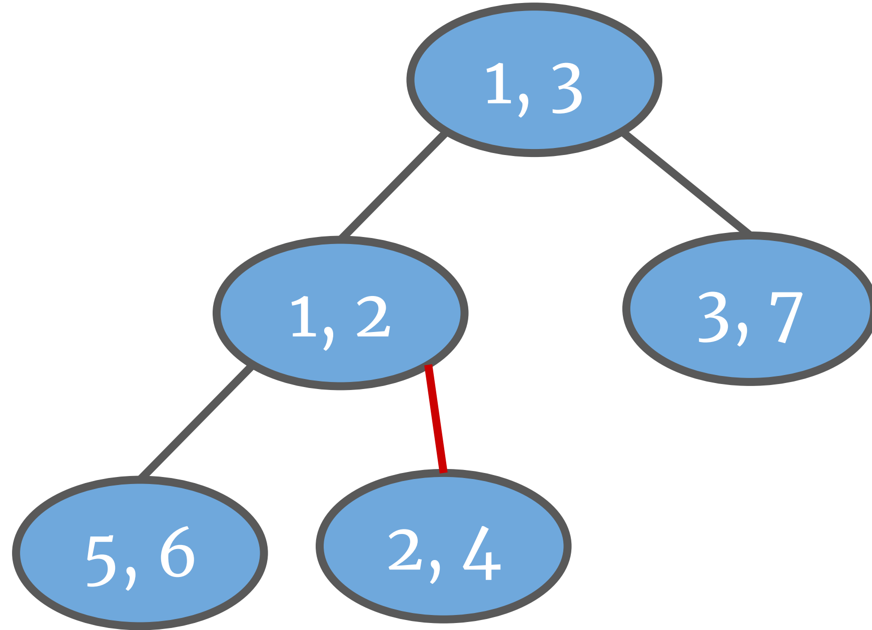
(1, 3), (5, 6), (3, 7), (2, 4), (1, 2)

Insert (1, 2)



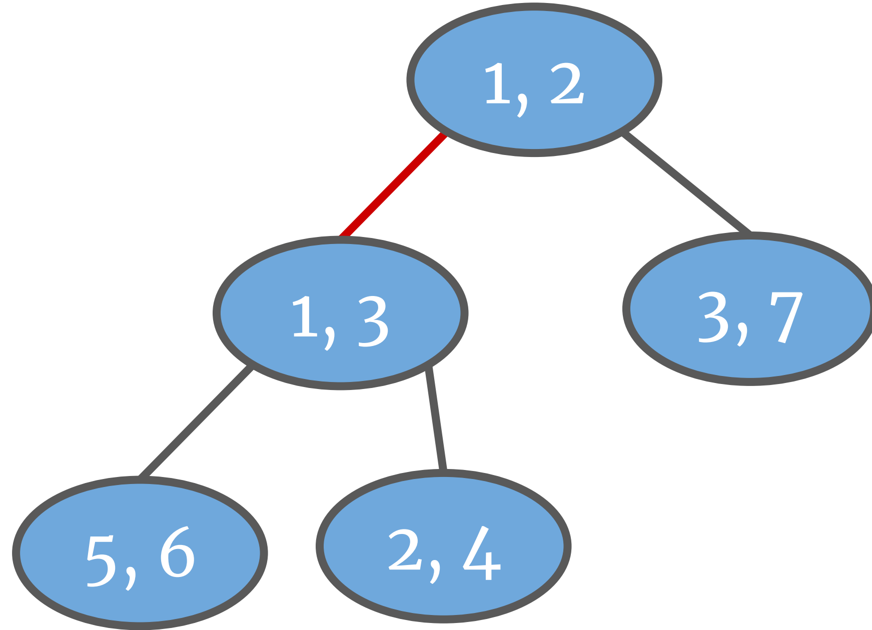
(1, 3), (5, 6), (3, 7), (2, 4), (1, 2)

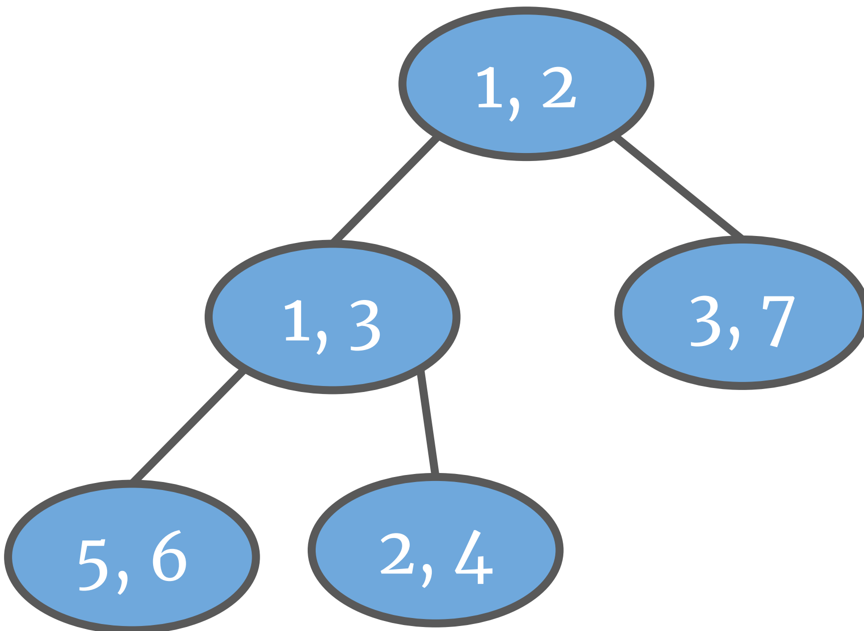
Insert (1, 2)



(1, 3), (5, 6), (3, 7), (2, 4), (1, 2)

Insert (1, 2)



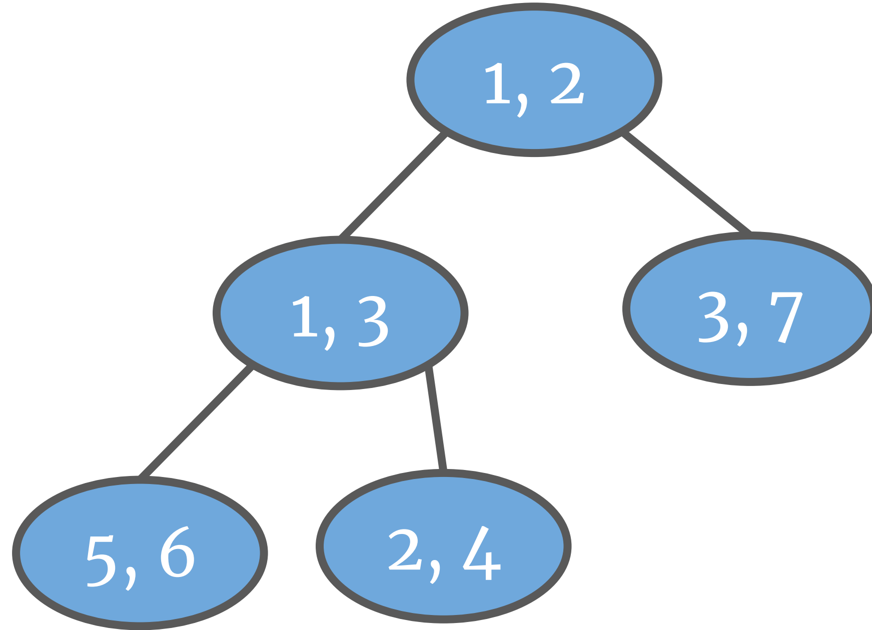


Students in room: 0

Total pairs: 0

Delete (1, 2)

Insert (2, ∞)

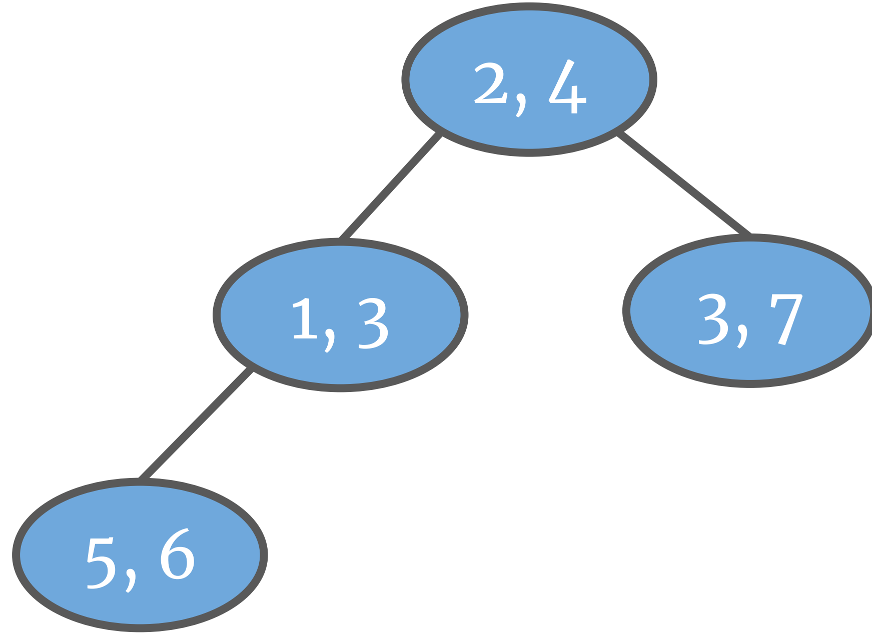


Students in room: 0

Total pairs: 0

Delete (1, 2)

Insert (2, ∞)

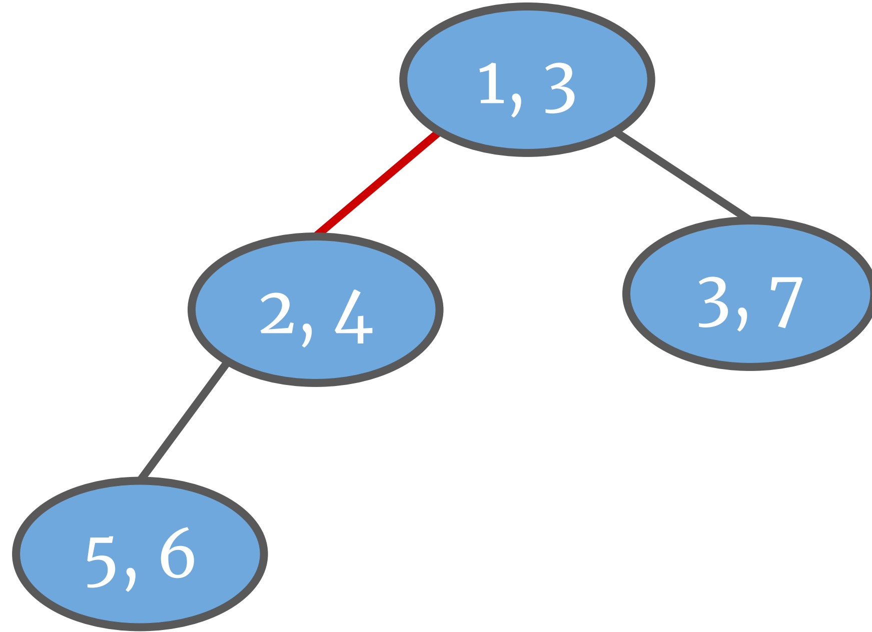


Students in room: 1

Total pairs: 0

Delete (1, 2)

Insert (2, ∞)

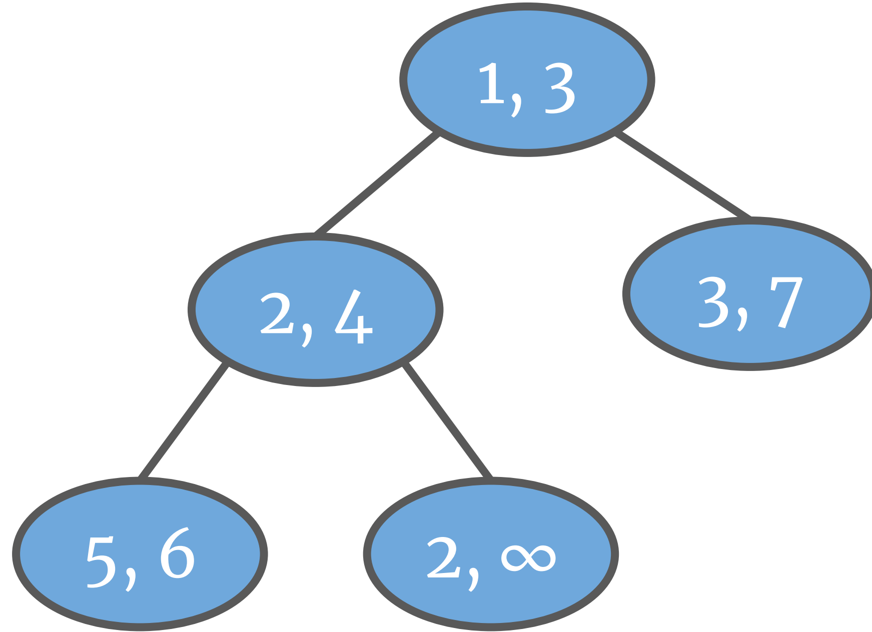


Students in room: 1

Total pairs: 0

Delete (1, 2)

Insert (2, ∞)

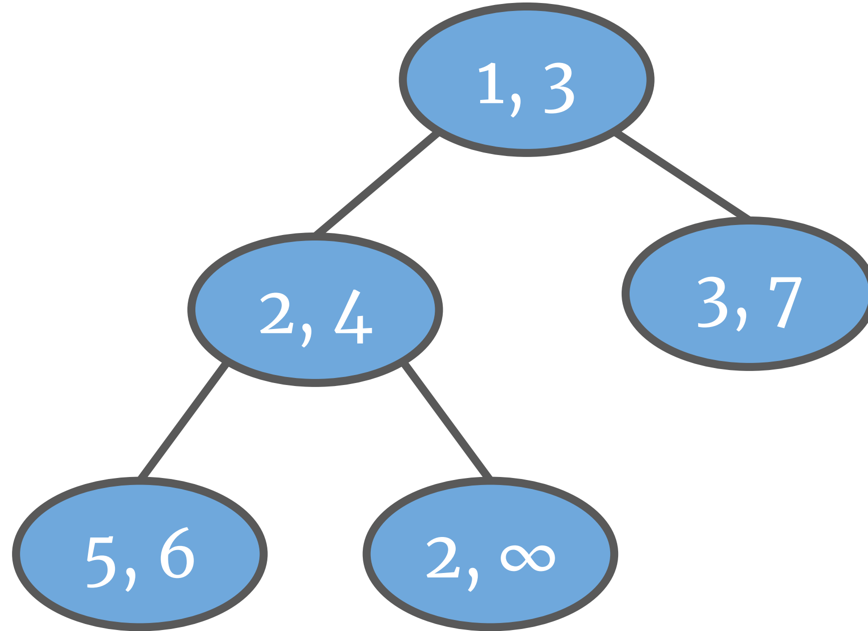


Students in room: 1

Total pairs: 0

Delete (1, 3)

Insert (3, ∞)

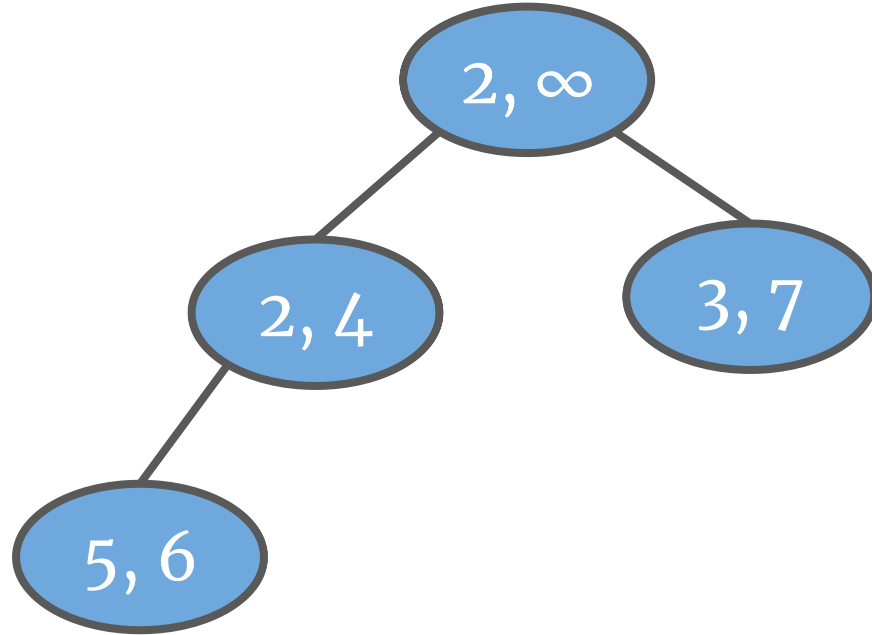


Students in room: 1

Total pairs: 0

Delete (1, 3)

Insert (3, ∞)

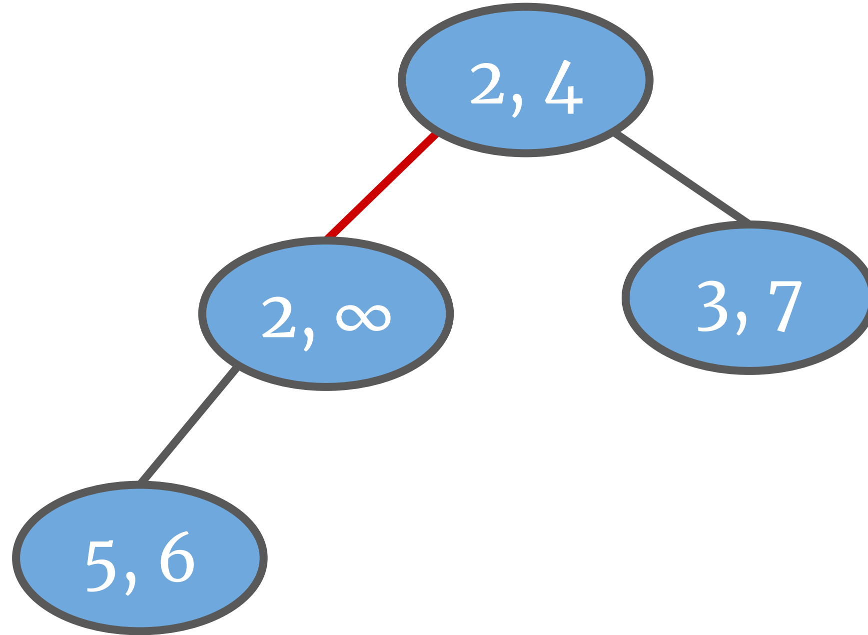


Students in room: 2

Total pairs: 0

Delete (1, 3)

Insert (3, ∞)

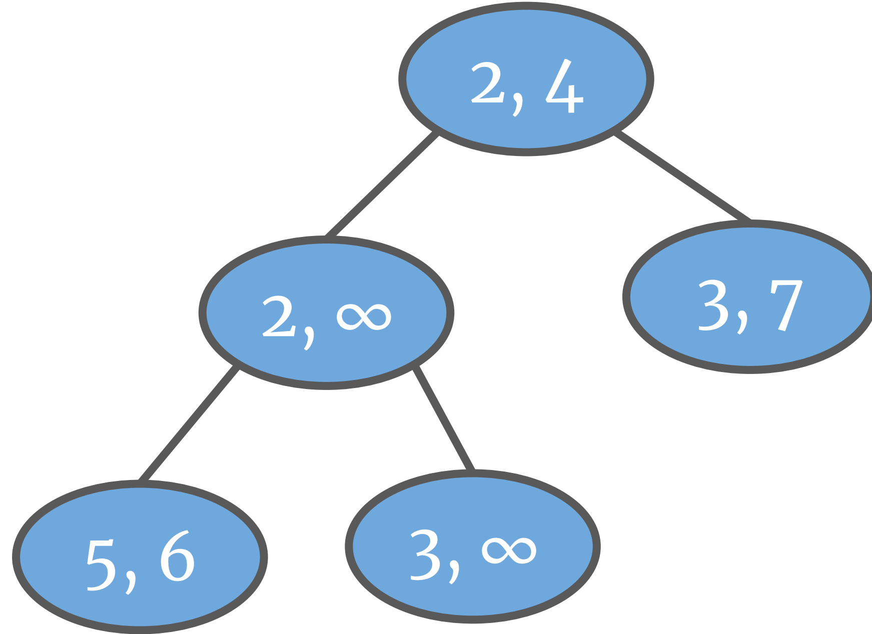


Students in room: 2

Total pairs: 0

Delete (1, 3)

Insert (3, ∞)

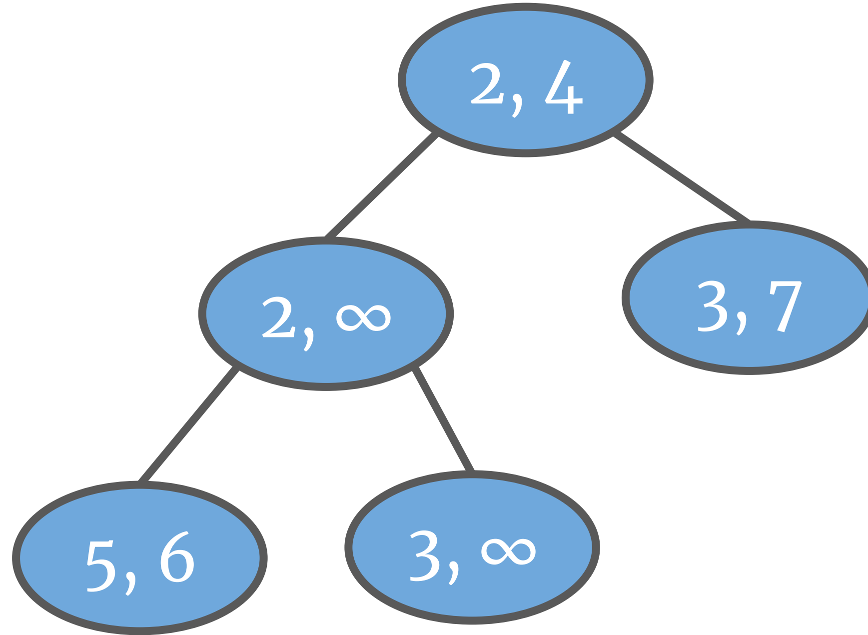


Students in room: 2

Total pairs: 0

Delete (2, 4)

Insert (4, ∞)

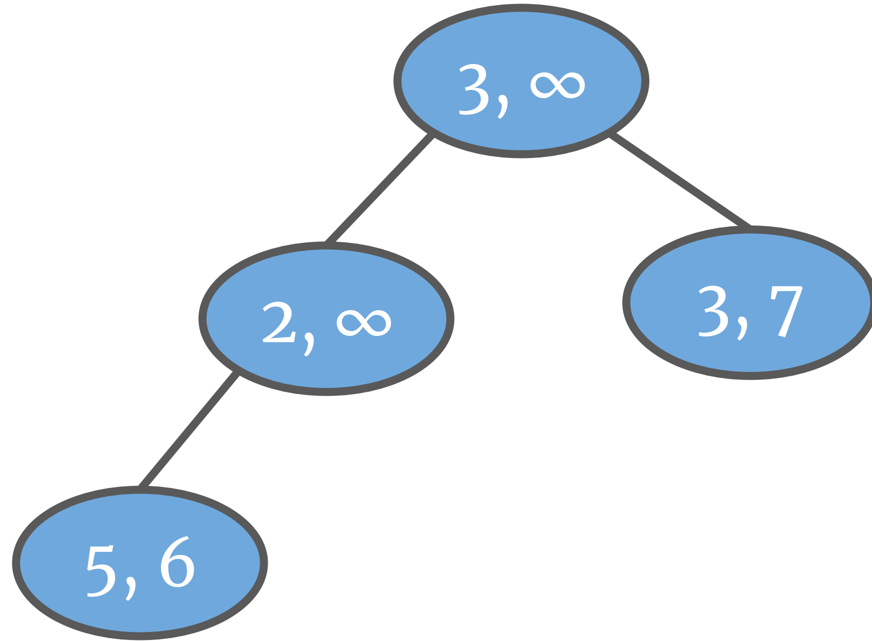


Students in room: 2

Total pairs: 0

Delete (2, 4)

Insert (4, ∞)

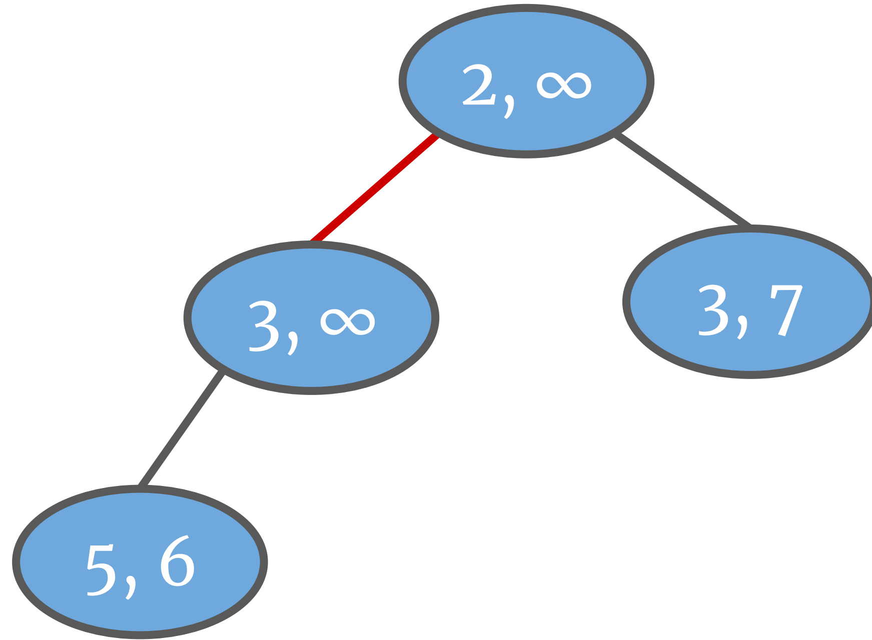


Students in room: 3

Total pairs: 0

Delete (2, 4)

Insert (4, ∞)

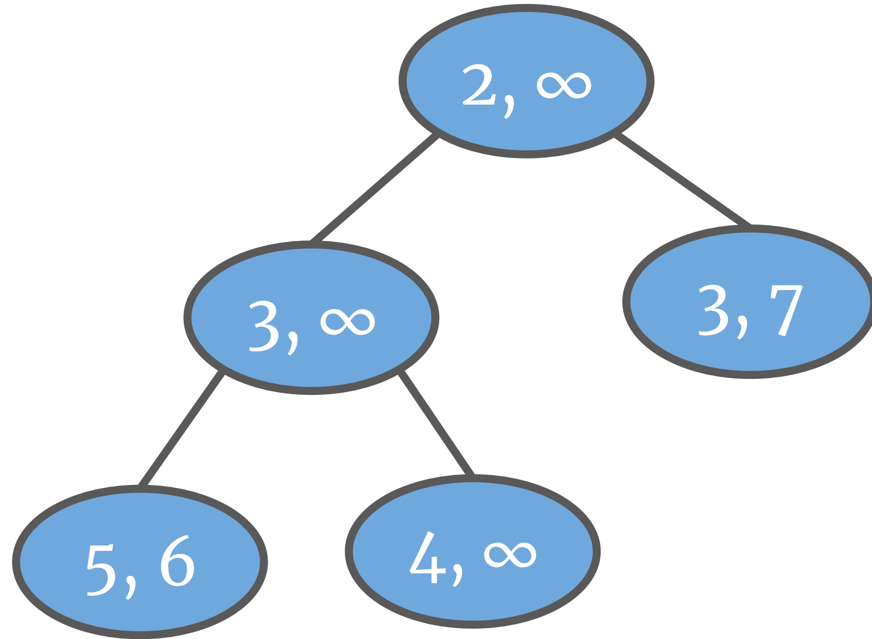


Students in room: 3

Total pairs: 0

Delete (2, 4)

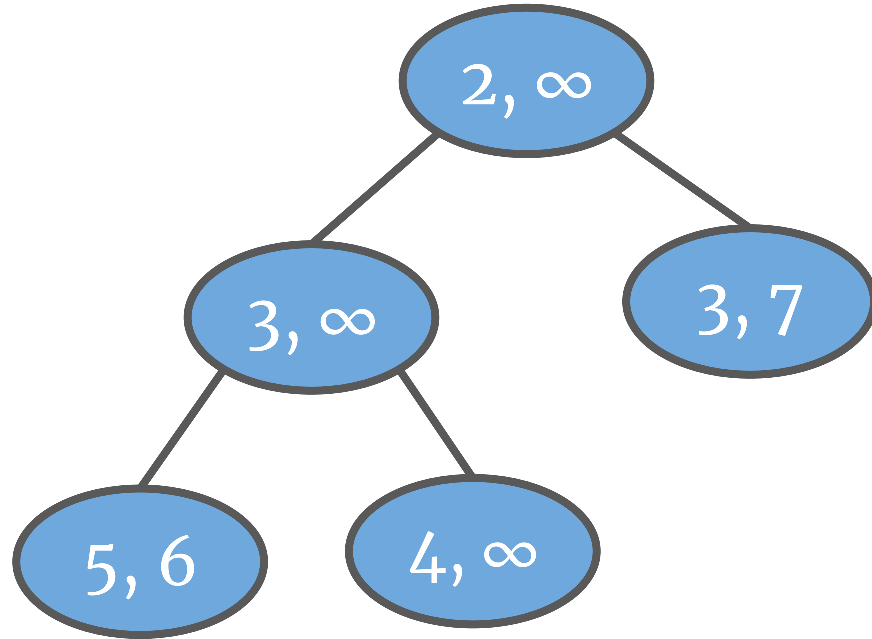
Insert (4, ∞)



Students in room: 3

Total pairs: 0

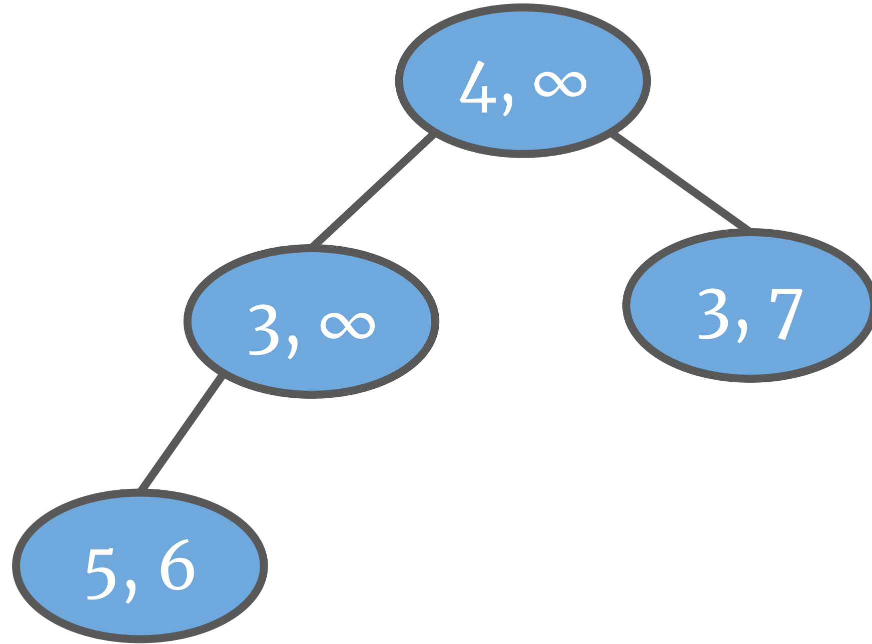
Delete $(2, \infty)$



Students in room: 3

Total pairs: 0

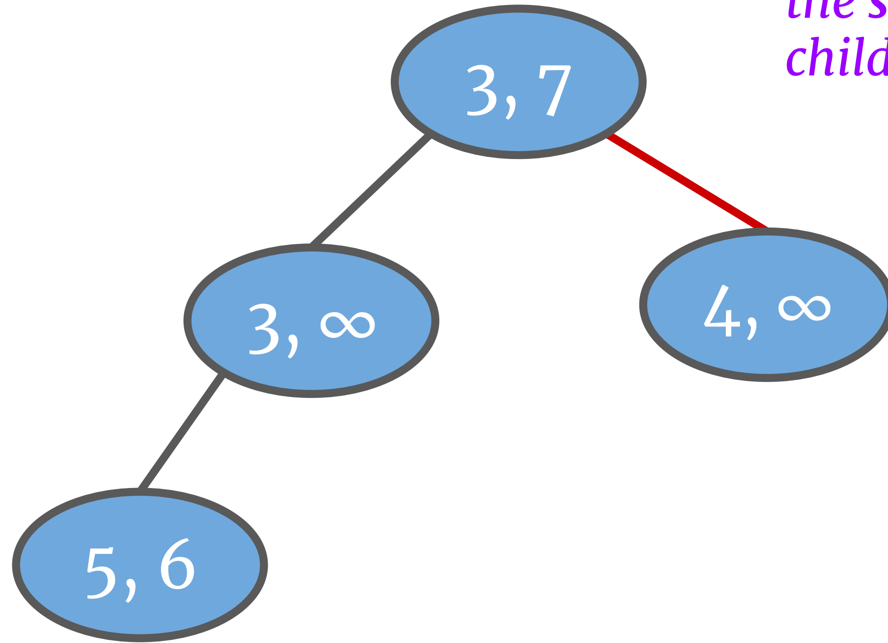
Delete (2, ∞)



Students in room: 2

Total pairs: **2**

Delete (2, ∞)

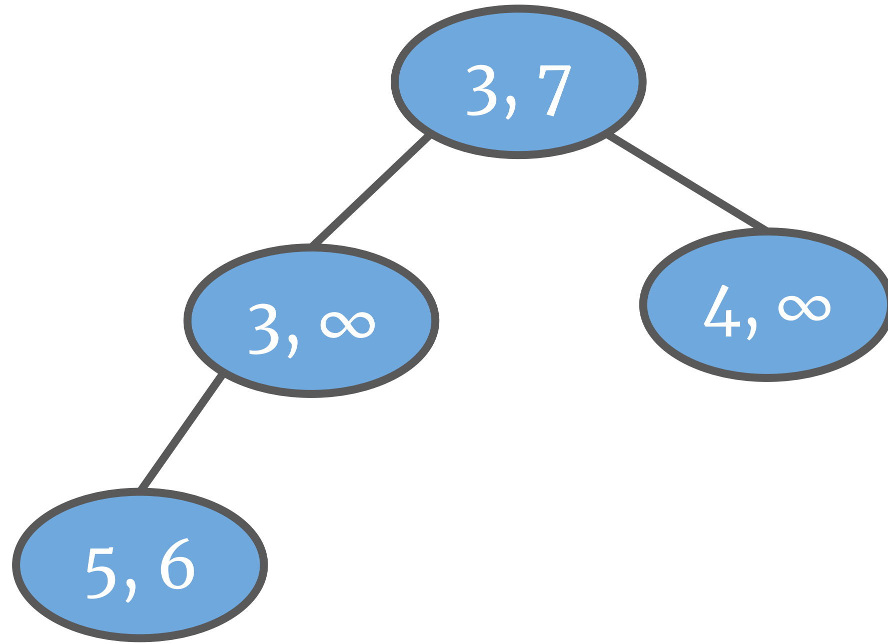


Students in room: 2

Total pairs: **2**

Delete (3, 7)

Insert (7, ∞)

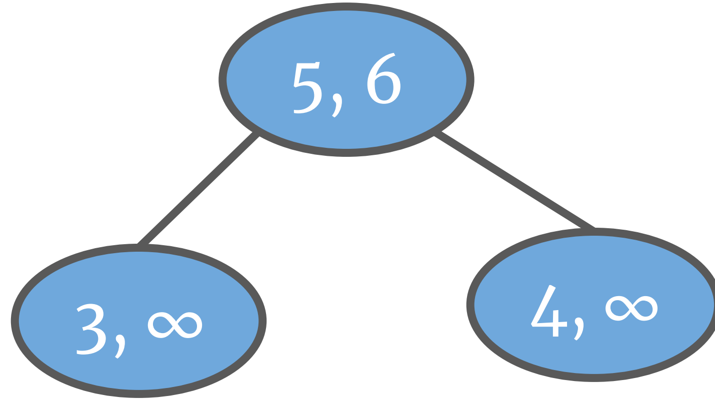


Students in room: 2

Total pairs: **2**

Delete (3, 7)

Insert (7, ∞)

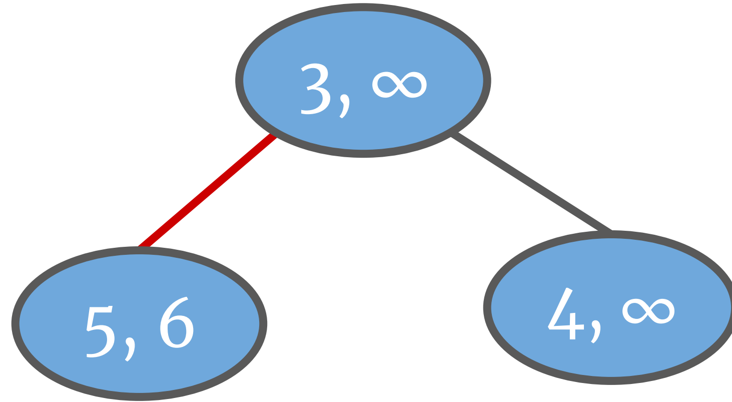


Students in room: 3

Total pairs: **2**

Delete (3, 7)

Insert (7, ∞)

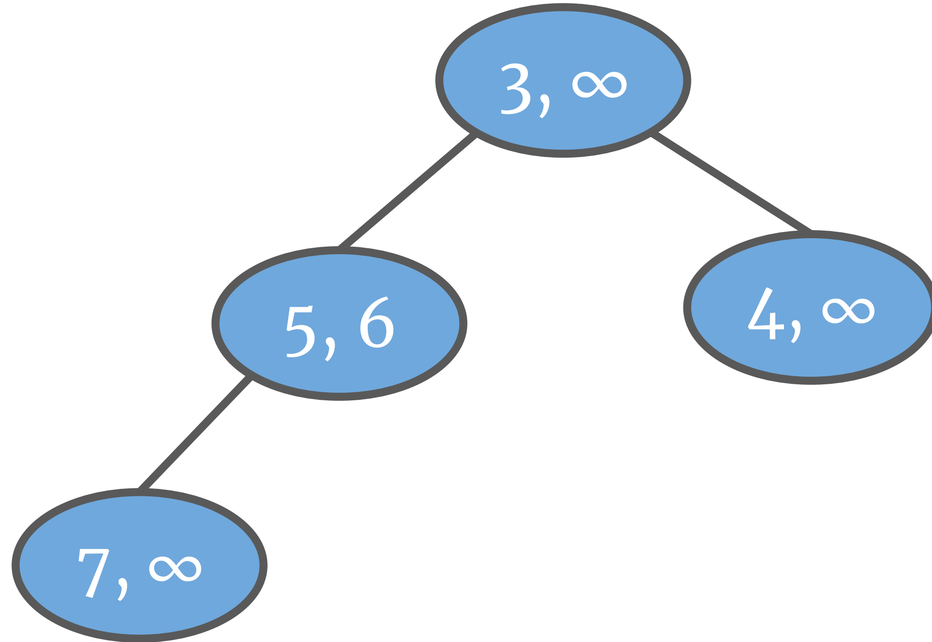


Students in room: 3

Total pairs: **2**

Delete (3, 7)

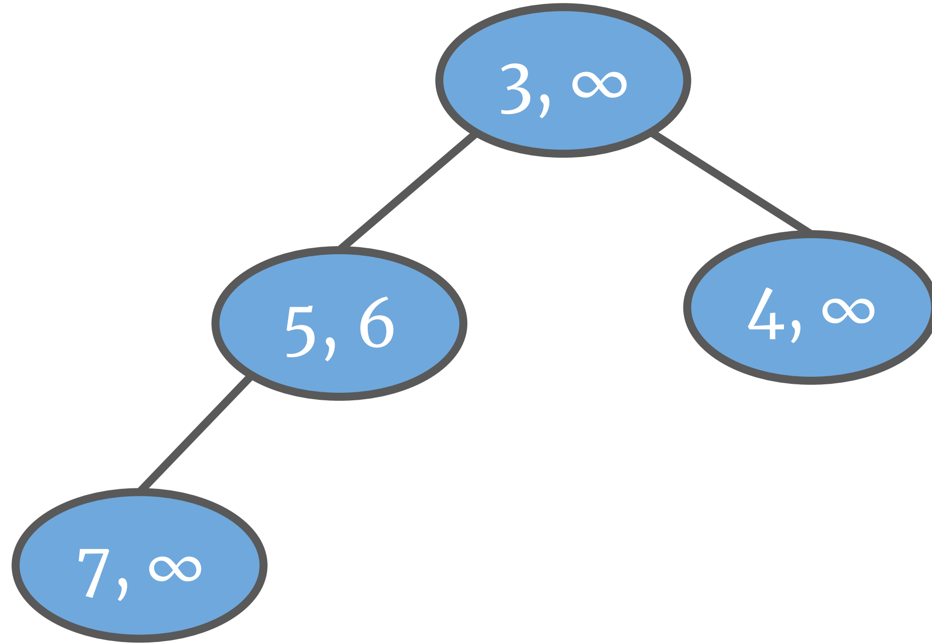
Insert (7, ∞)



Students in room: 3

Total pairs: **2**

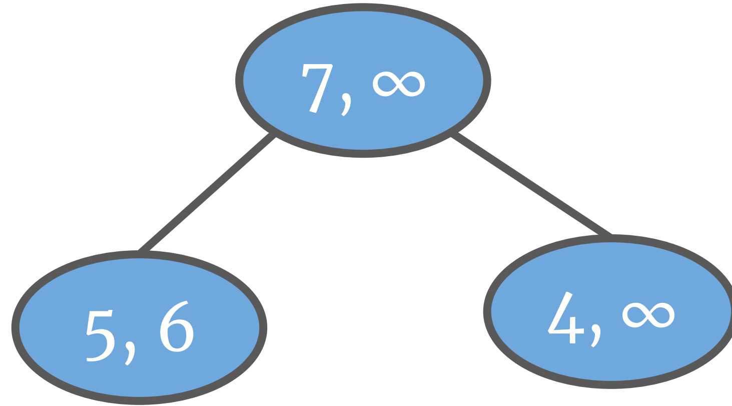
Delete $(3, \infty)$



Students in room: 3

Total pairs: **2**

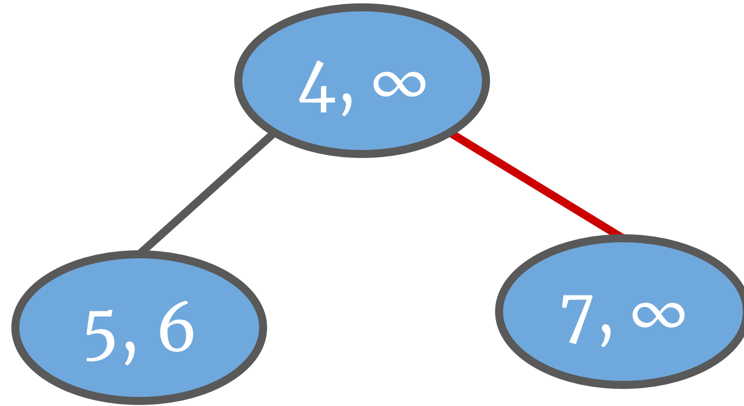
Delete $(3, \infty)$



Students in room: 2

Total pairs: **4**

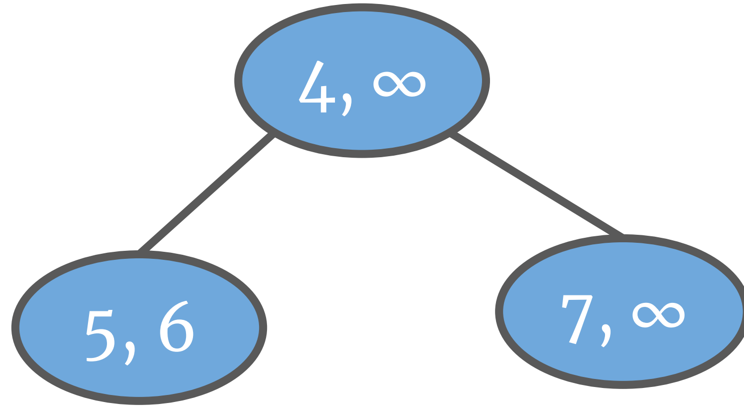
Delete (3, ∞)



Students in room: 2

Total pairs: **4**

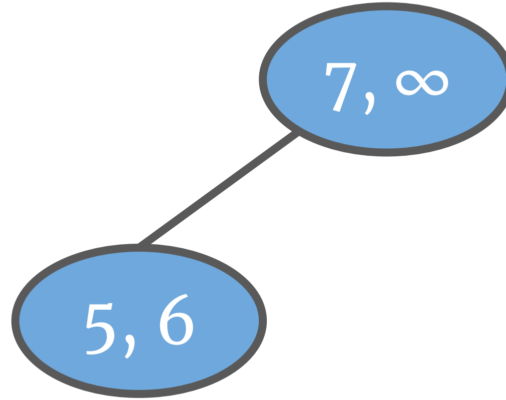
Delete $(4, \infty)$



Students in room: 2

Total pairs: **4**

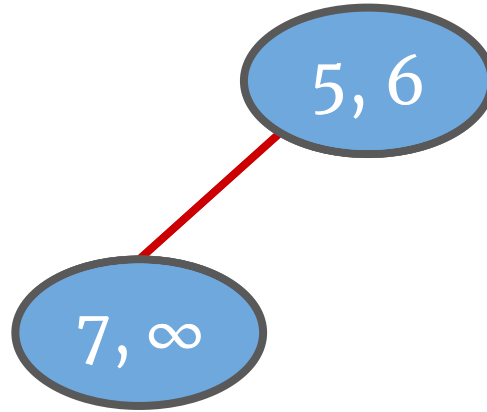
Delete $(4, \infty)$



Students in room: 1

Total pairs: **5**

Delete $(4, \infty)$

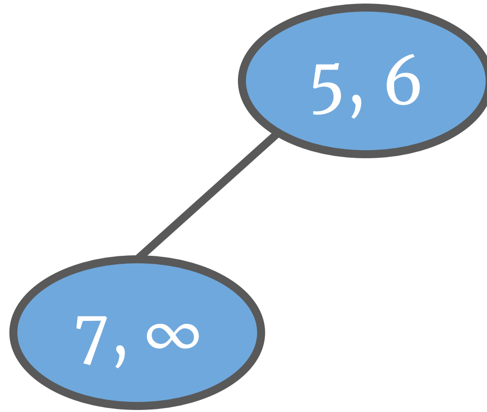


Students in room: 1

Total pairs: **5**

Delete $(5, 6)$

Insert $(6, \infty)$

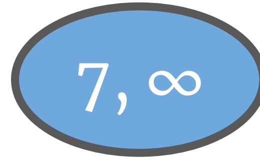


Students in room: 1

Total pairs: **5**

Delete (5, 6)

Insert (6, ∞)

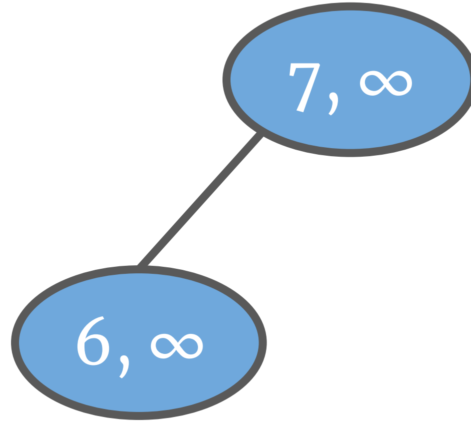


Students in room: 2

Total pairs: **5**

Delete $(5, 6)$

Insert $(6, \infty)$

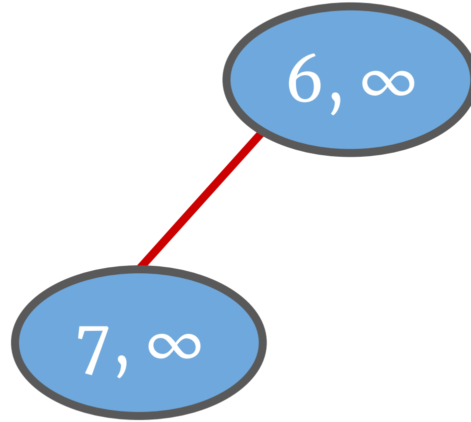


Students in room: 2

Total pairs: **5**

Delete (5, 6)

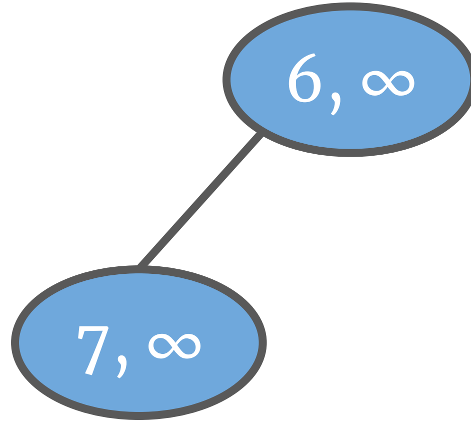
Insert (6, ∞)



Students in room: 2

Total pairs: **5**

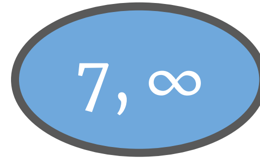
Delete $(6, \infty)$



Students in room: 2

Total pairs: **5**

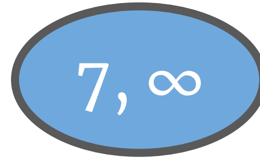
Delete $(6, \infty)$



Students in room: 1

Total pairs: **6**

Delete $(7, \infty)$



Students in room: 1

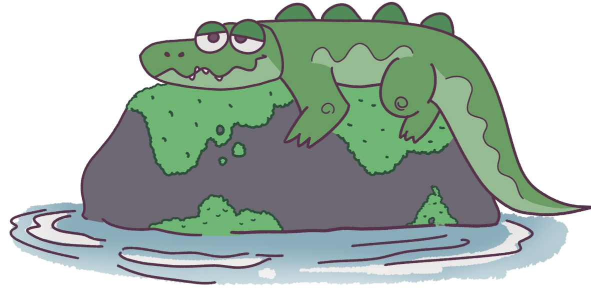
Total pairs: **6**

Delete $(7, \infty)$

Students in room: 0

Total pairs: **6**

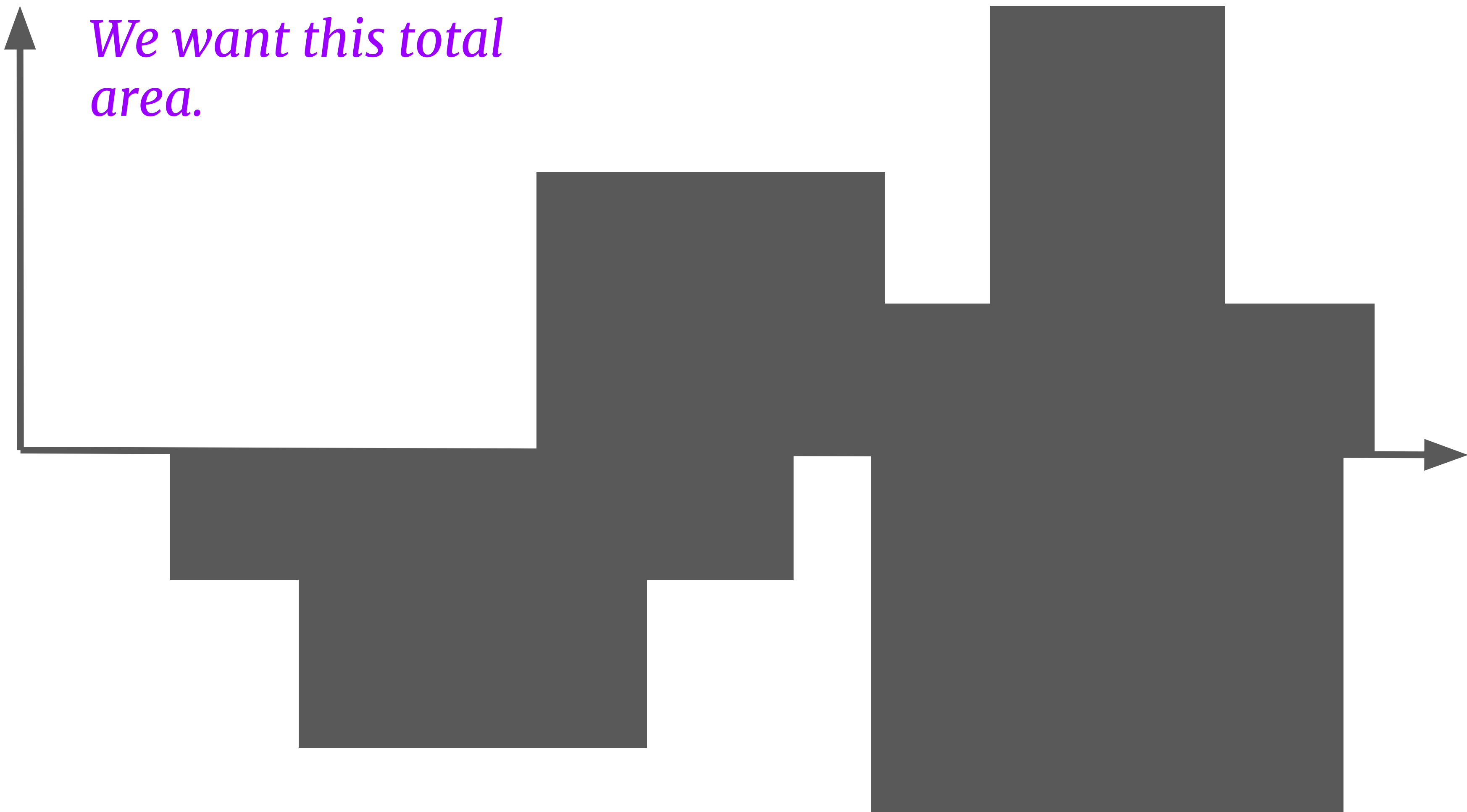
We never explicitly sorted the list!

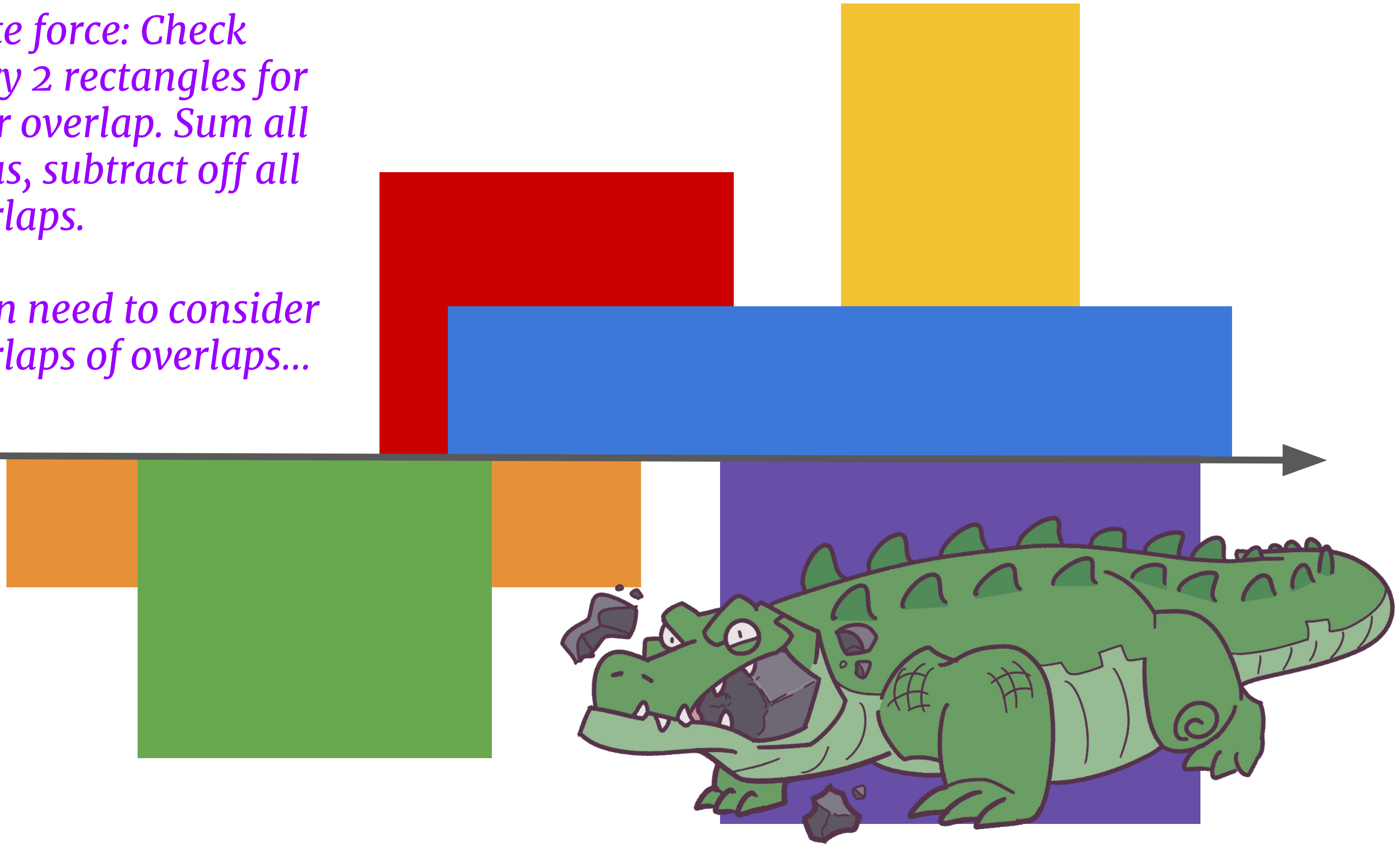


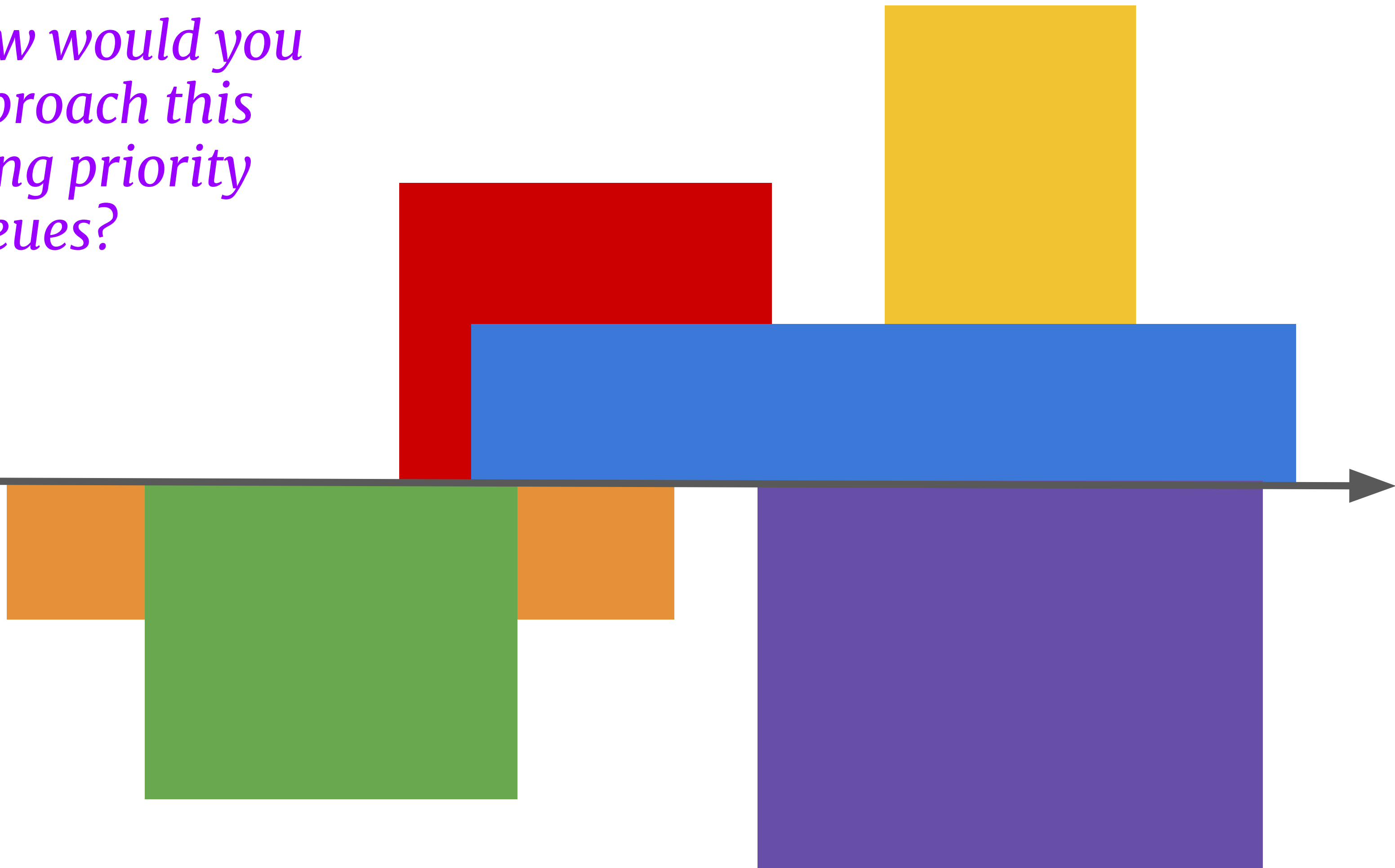
But we got an O(n log n) solution – the same as with MergeSort.

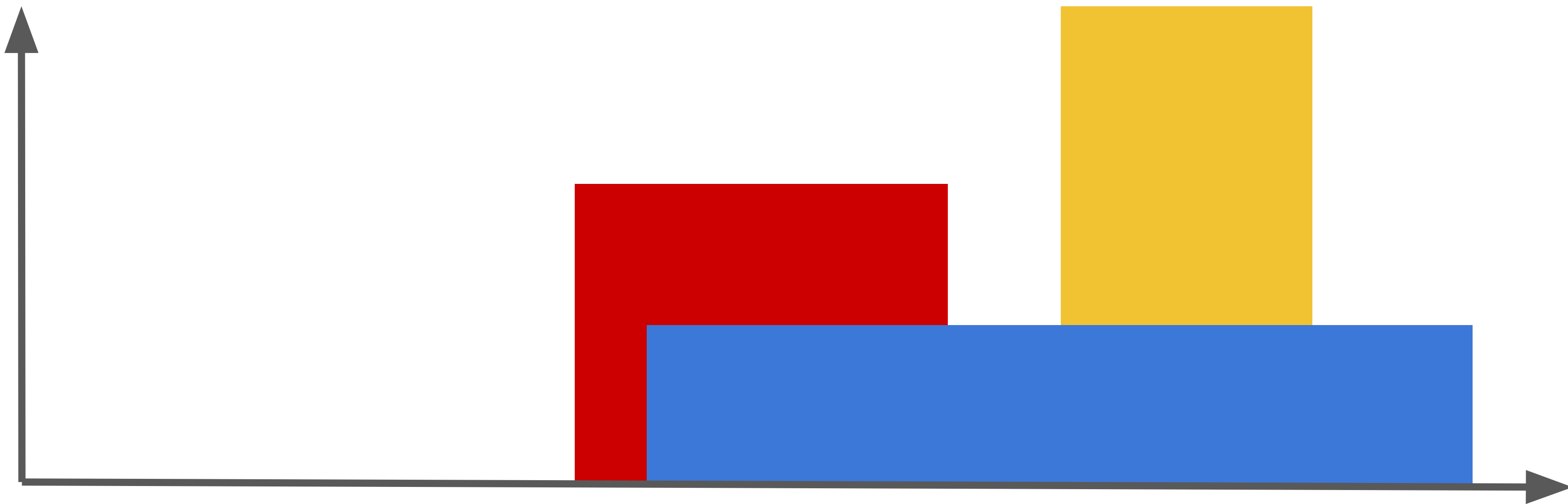
Another classic priority queue scenario

- You have a piece of abstract art made up of rectangles.
- Each rectangle has one of its sides overlapping the x-axis.
- Given the locations and sizes of the rectangles, find the total area of the shape.

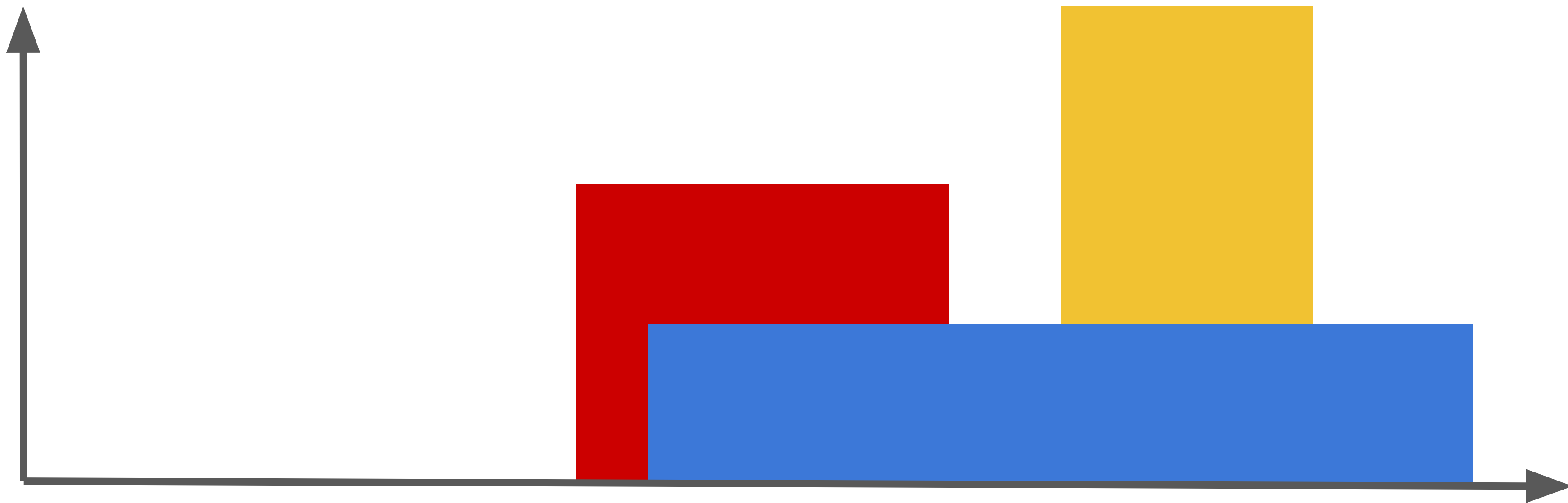

We want this total area.


Brute force: Check every 2 rectangles for their overlap. Sum all areas, subtract off all overlaps.
Then need to consider overlaps of overlaps...


How would you approach this using priority queues?

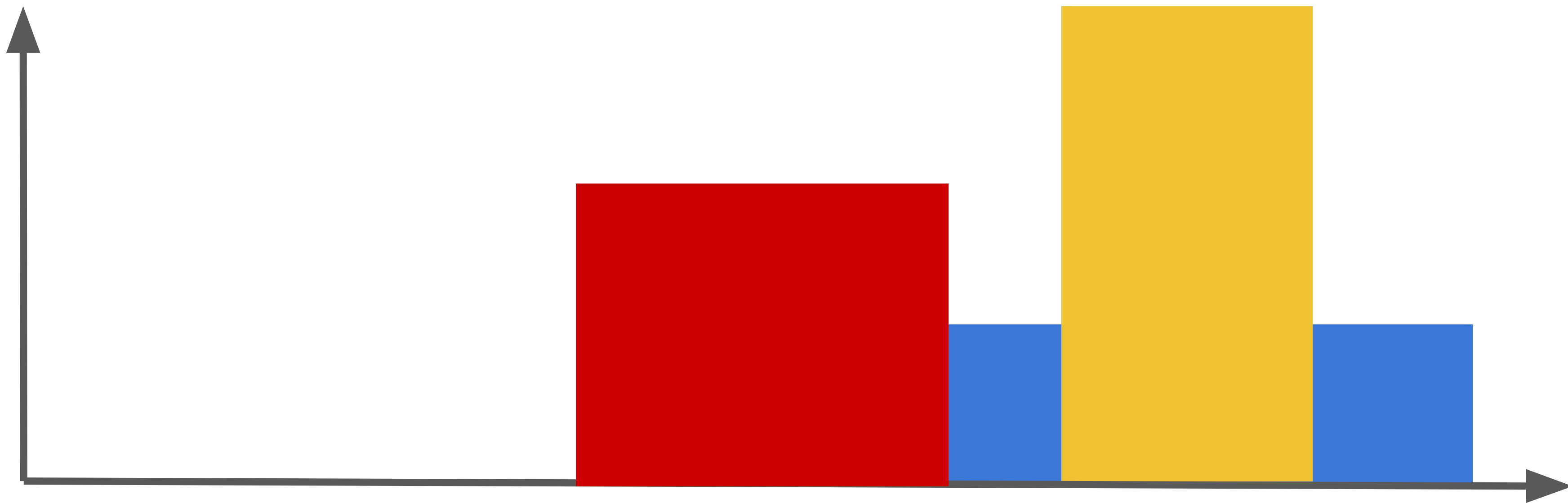


Let's start with just the stuff above the line.



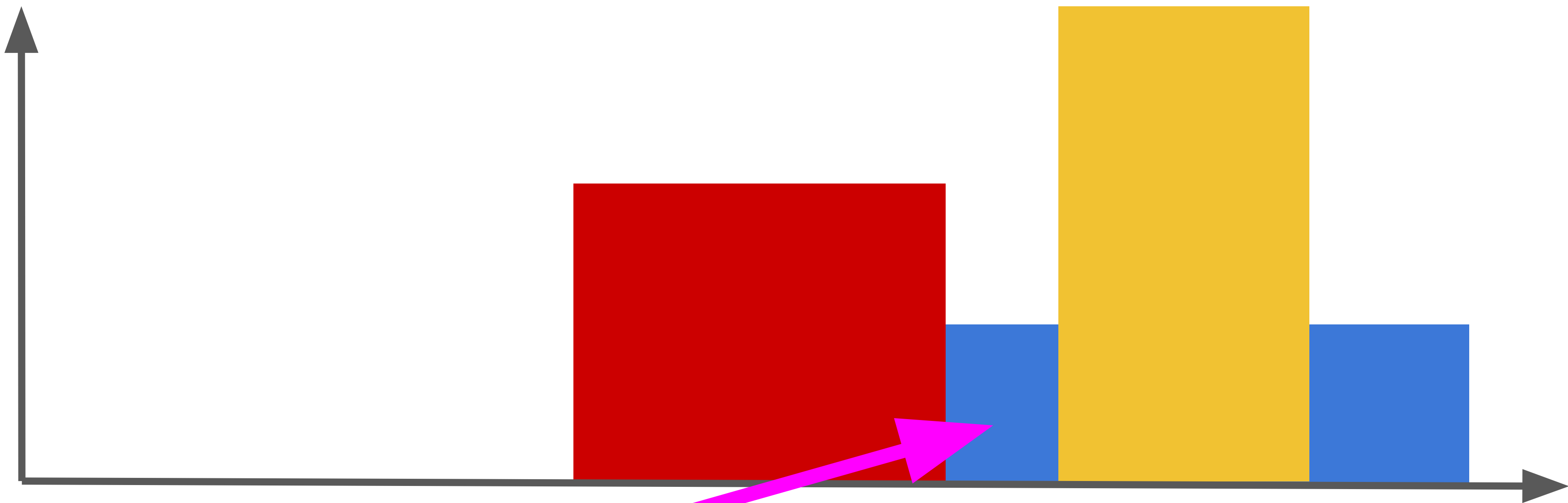
Let's start with just the stuff above the line.

Looking left to right: red is initially most important, then blue, then yellow, then blue

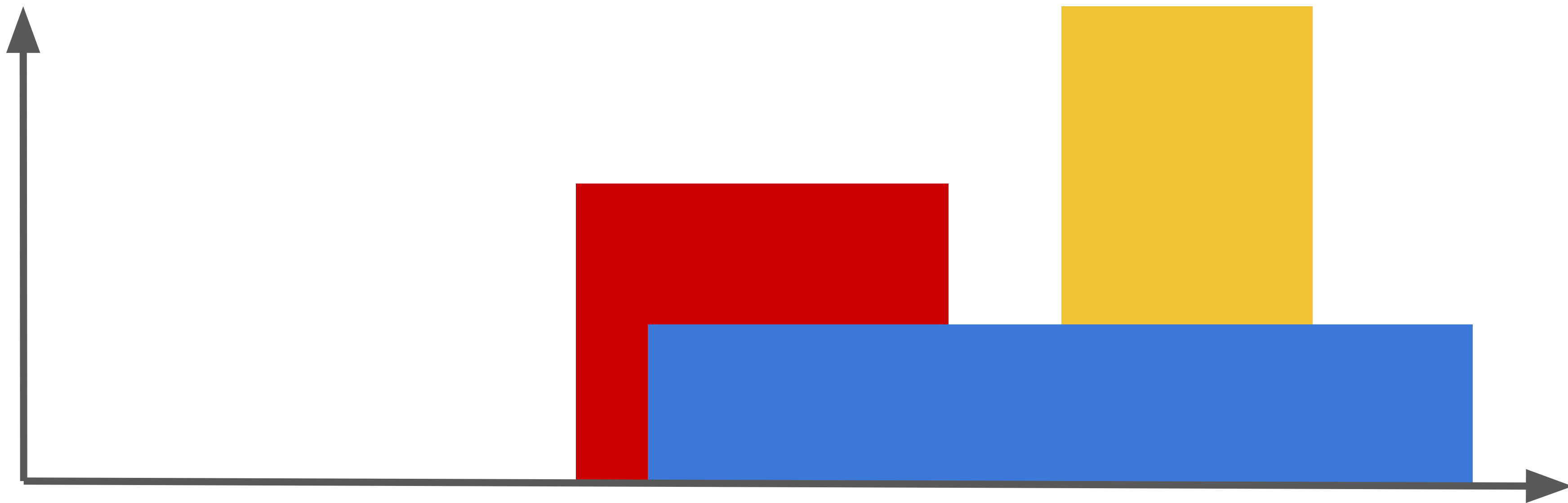


Let's start with just the stuff above the line.

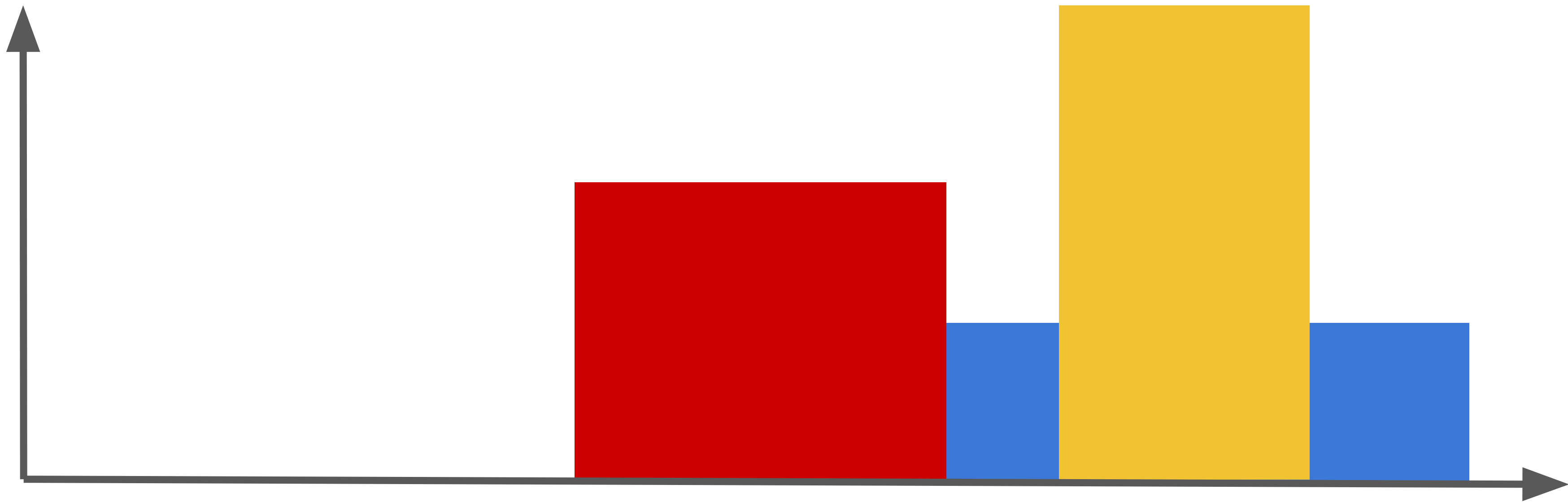
Looking left to right: red is initially most important, then blue, then yellow, then blue



Stuff like this is annoying. We can't forget about the blue rectangle is there since it becomes important again later (twice!)



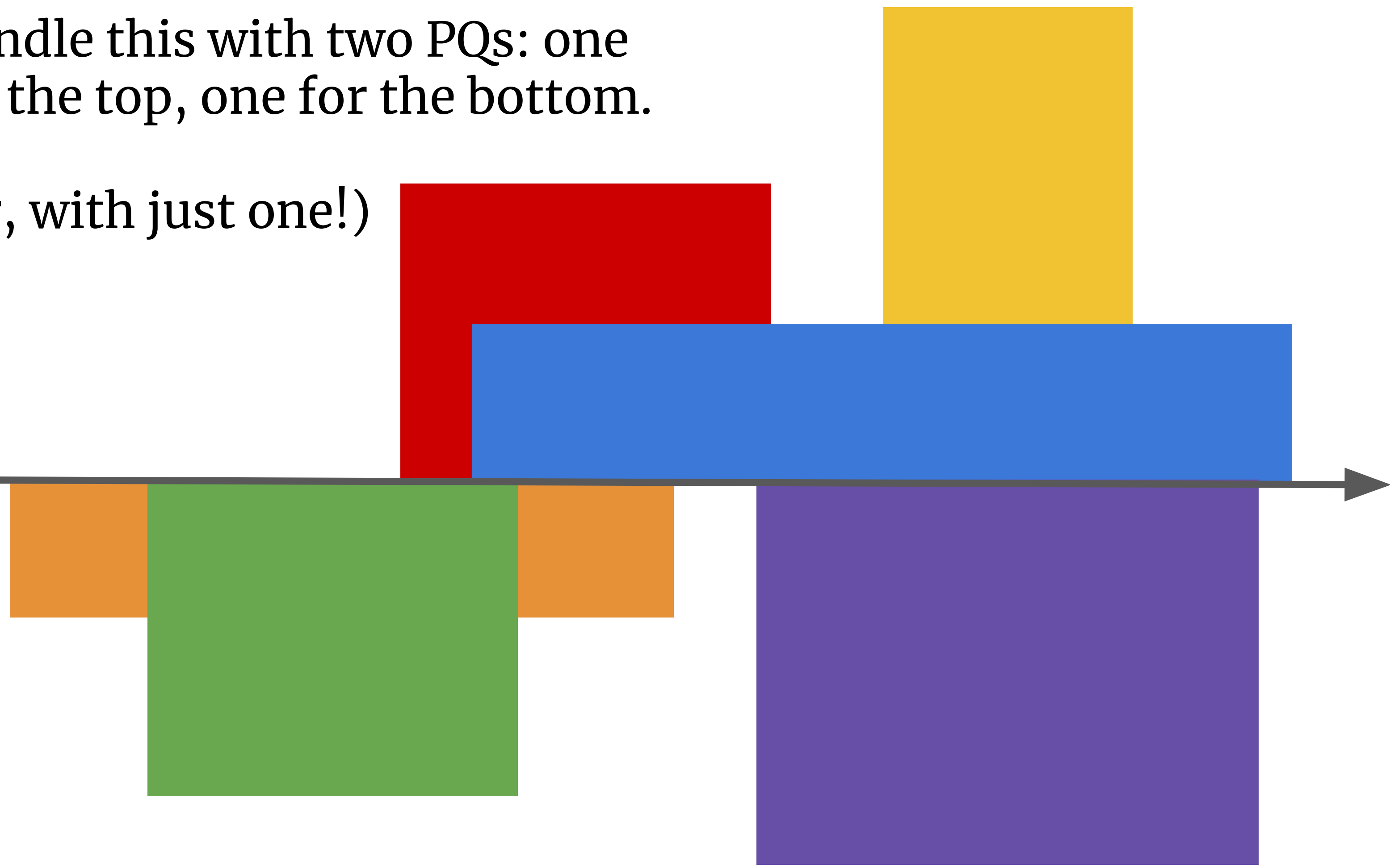
- This is similar to the office hours problem, but with some extra info!
- The PQ keeps track of events: a rectangle starts, or a rectangle ends.



- The PQ keeps track of events: a rectangle starts, or a rectangle ends.
- Maintain separate variables for current max height, and total area so far.
- Upon each event, add area seen since last event.

Handle this with two PQs: one for the top, one for the bottom.

(Or, with just one!)



Priority Queues recap

- This is my favorite data structure!
  - It is common in tech interview questions (and got me a job!)

- There are lots of flavors of these, but you are only responsible for understanding the binary-heap one and its array-based implementation.

Priority Queues recap

- Heaps are good at one thing, and so they are really good at it.

- More specialized priority queues do stuff like: support decreasing the value of an entry efficiently (rather than deleting it and adding it again)

- There's not one single optimal data structure. (*Not even the mighty hash table!*) It really depends on your needs.

7/8 Lecture Agenda

- Announcements
- Part 3-3: Heaps and Priority Queues
- 10 minute break!
- Part 3-4: Self-Balancing BSTs

7/8 Lecture Agenda

- Announcements
- Part 3-3: Heaps and Priority Queues
- 10 minute break!
- Part 3-4: Self-Balancing BSTs

WORLD 3-4

Self-Balancing BSTs

Divide and Conquer

Sorting & Randomization

Data Structures

Graph Search

Dynamic Programming

Greed & Flow

Special Topics

Back to Binary Search Trees (BSTs)

- We need to support at least two operations:
    - **Insert** an element into the tree.
    - **Search** the tree to see if it has an element.

- We might also want to support **deletion**.

But don't hash tables do this?

- We need to support at least two operations:
  - **Insert** an element into the tree.
  - **Search** the tree to see if it has an element.

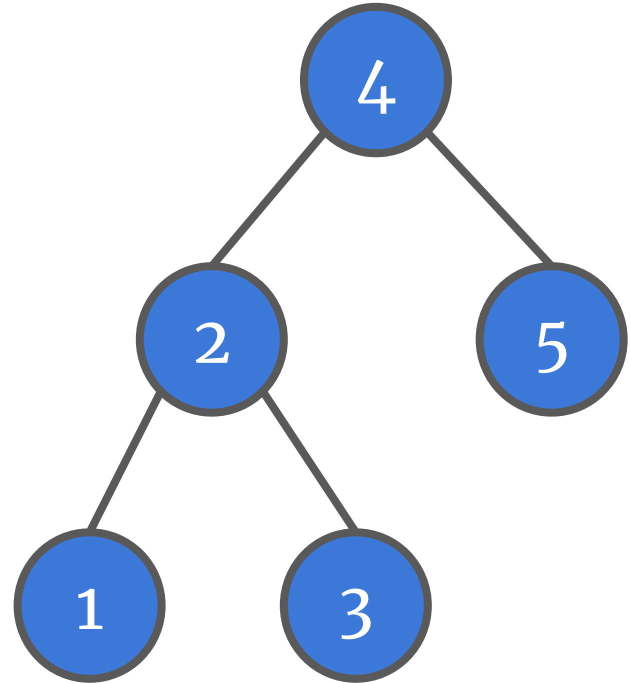
- We might also want to support **deletion**.

Why would we need a BST?

BSTs can do some things hash tables can't!

- Inorder traversal of a BST produces a sorted list.
    - Hash tables do not retain any ordering information, so you can't do this efficiently.

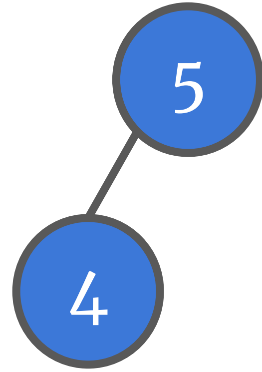
- If you search a BST for something that isn't there, you can at least find which values are closest to it. Hash tables have no idea, though.



A problem: bad insertion patterns can cause BSTs to become imbalanced.

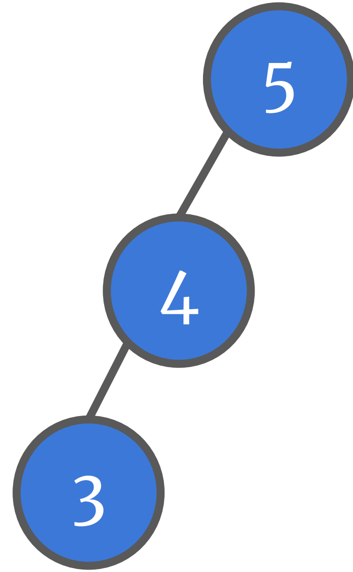
- This is the result of inserting 5, 4, 3, 2, 1.
- Each insertion is O(n)...
- Searching is also O(n)...

A problem: bad insertion patterns can cause BSTs to become imbalanced.



- This is the result of inserting 5, 4, 3, 2, 1.
- Each insertion is O(n)...
- Searching is also O(n)...

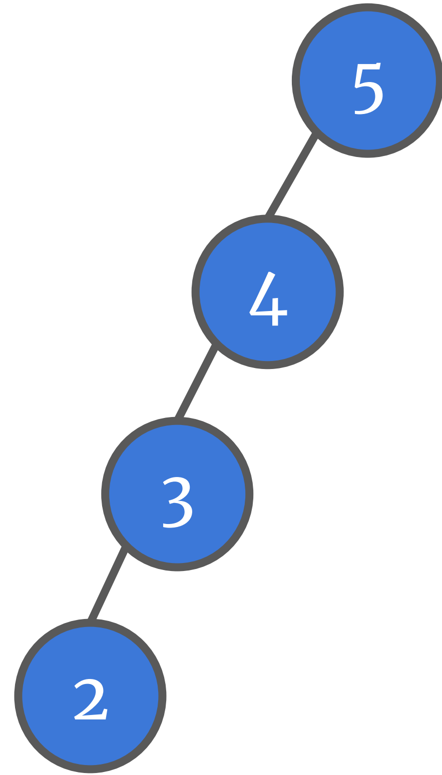
A problem: bad insertion patterns can cause BSTs to become imbalanced.



- This is the result of inserting 5, 4, 3, 2, 1.
- Each insertion is O(n)...
- Searching is also O(n)...

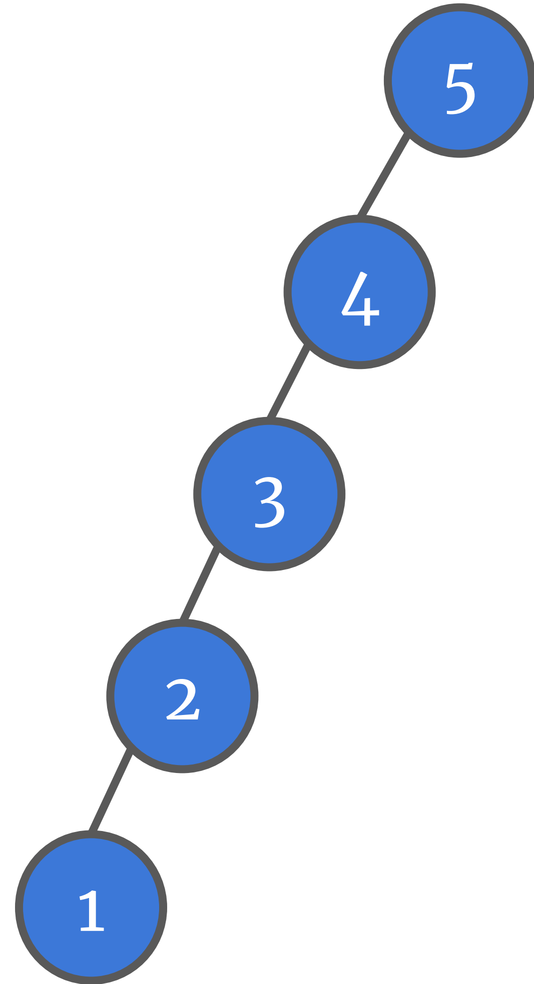
A problem: bad insertion patterns can cause BSTs to become imbalanced.

- This is the result of inserting 5, 4, 3, 2, 1.
- Each insertion is O(n)...
- Searching is also O(n)...

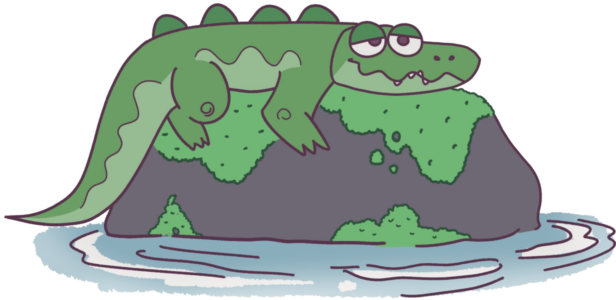


A problem: bad insertion patterns can cause BSTs to become imbalanced.

- This is the result of inserting 5, 4, 3, 2, 1.
- Each insertion is $O(n)$...
- Searching is also $O(n)$...



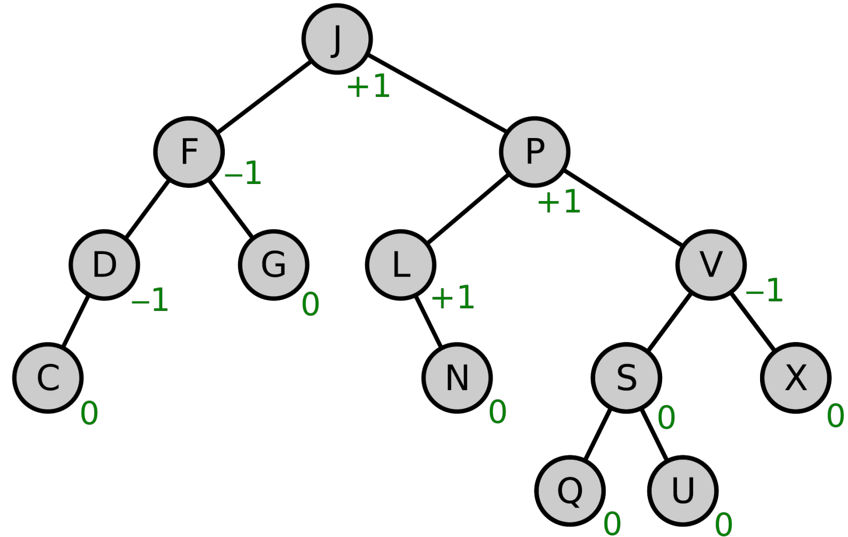
We could go in and rebalance the tree every so often...



How would we know when? And that's a lot of work! Can the tree please just rebalance itself?

Good news, everyone!

- There are lots of self-balancing binary search trees out there!
  - 2-3-(4) trees
  - AVL trees
  - B-trees
  - Red-black trees
  - Splay trees
  - Tango trees
  - Treaps
  - ...and more!



Terrible news, everyone!

- These are all complicated, because **this is a hard problem**.

- Lots of fussy details and casework...

  - 2-3-(4) trees
  - AVL trees
  - B-trees
  - Red-black trees
  - Splay trees
  - Tango trees
  - Treaps
  - ...and more!

Good news, everyone!

- We'll just focus on *one* of these...

  - 2-3-(4) trees
  - AVL trees
  - B-trees
  - Red-black trees
  - Splay trees
  - Tango trees
  - Treaps
  - ...and more!

Terrible news, everyone!

- This is one of *the* fussiest and caseworky of all self-balancing BSTs!
    - 2-3-(4) trees
    - AVL trees
    - B-trees
    - Red-black trees
    - Splay trees
    - Tango trees
    - Treaps
    - ...and more!

Good news, everyone!

- This is one of *the* fussiest and caseworky of all self-balancing BSTs!
    - I want you to understand the high-level idea of why they work, but you are not responsible for the details.

- 2-3-(4) trees
- AVL trees
- B-trees
- Red-black trees
- Splay trees
- Tango trees
- Treaps
- ...and more!

A small part of the Wikipedia article

Notes to the insert diagrams [edit]

before				case →	rota-tion	assig-nment	after				→ next	Δh
P	G	U	*x*				P	G	U	*x*		
—				I3							→	
⚫				I1							→	
🔴	—			I4			⚫				→	
🔴	⚫	🔴		I2		N:=G	?				?	2
🔴	⚫	⚫	i	I5	P↰N	N:=P	🔴	⚫	⚫	o	I6	0
🔴	⚫	⚫	o	I6	P↱G		⚫	🔴	⚫		→	

Insertion: This synopsis shows in its *before* columns, that all possible cases[32] of constellations are covered.

- In the diagrams, **P** is used for **N**'s parent, **G** for the grandparent, and **U** will denote **N**'s uncle.
- The diagrams show the parent node **P** as the left child of its parent **G** even though it is possible for **P** to be on either side. The sample code covers both possibilities by means of the side variable `dir`.
- **N** is the insertion node, but as the operation proceeds also other nodes may become current (see case I2).
- The diagrams show the cases where **P** is red as well, the red-violation.
- The column *x* indicates the change in child direction, i.e. o (for "outer") means that **P** and **N** are both left or both right children, whereas i (for "inner") means that the child direction changes from **P**'s to **N**'s.
- The column group *before* defines the case, whose name is given in the column *case*. Thereby possible values in cells left empty are ignored. So in case I2 the sample code covers both possibilities of child directions of **N**, although the corresponding diagram shows only one.
- The rows in the synopsis are ordered such that the coverage of all possible RB cases is easily comprehensible.
- The column *rotation* indicates whether a rotation contributes to the rebalancing.
- The column *assignment* shows an assignment of **N** before entering a subsequent step. This possibly induces a reassignment of the other nodes **P**, **G**, **U** as well.
- If something has been changed by the case, this is shown in the column group after

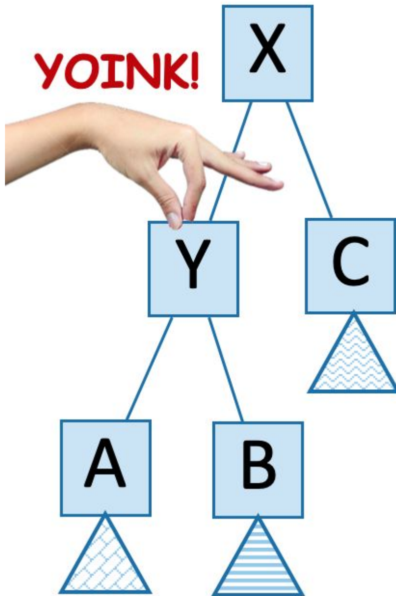
you do not need to understand this! It's just to gawk at

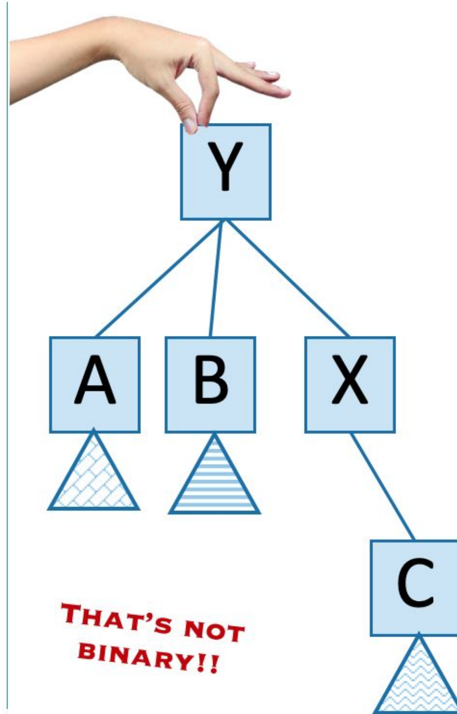
Idea 1: Rotations

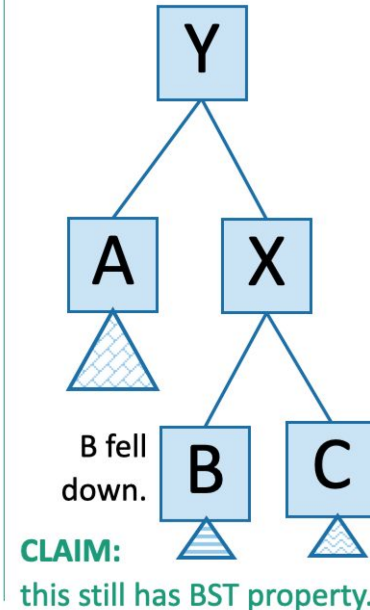
No matter what lives underneath A,B,C, **this takes time O(1).** (Why?)

- Maintain Binary Search Tree (BST) property, while moving stuff around.

Note: A, B, C, X, Y are variable names, not the contents of the nodes.

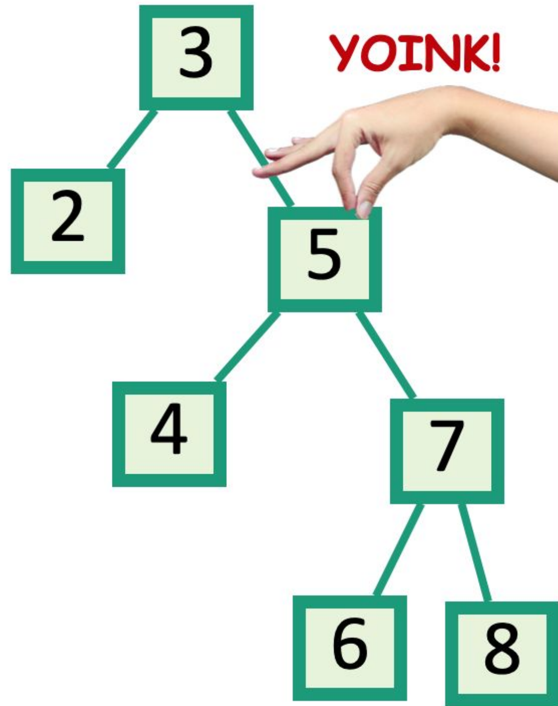


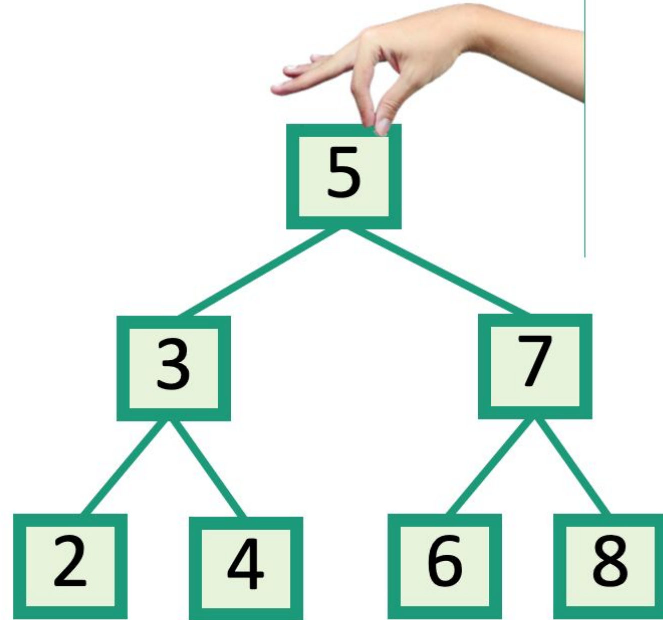




CLAIM:
this still has BST property.

This seems helpful





Idea 2: have some proxy for balance

- Maintaining perfect balance is too hard.
- Instead, come up with some proxy for balance:
  - If the tree satisfies [SOME PROPERTY], then it's pretty balanced.
  - We can maintain [SOME PROPERTY] using rotations.

There are actually several ways to do this, but today we'll see…

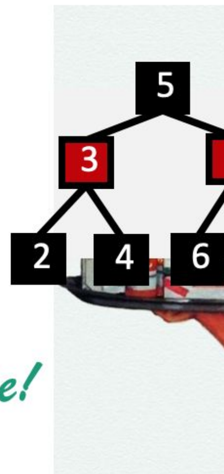
Red-Black Trees

- A Binary Search Tree that balances itself!
- No more time-consuming by-hand balancing!
- Be the envy of your friends and neighbors with the time-saving...

Red-Black tree!

Maintain balance by stipulating that **black nodes** are balanced, and that there *aren't too many* **red nodes**.

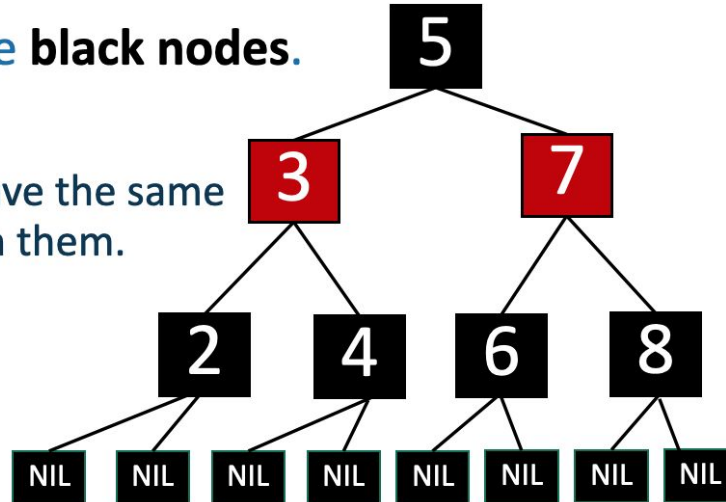
It's just good sense!



Red-Black Trees

obey the following rules (which are a proxy for balance)

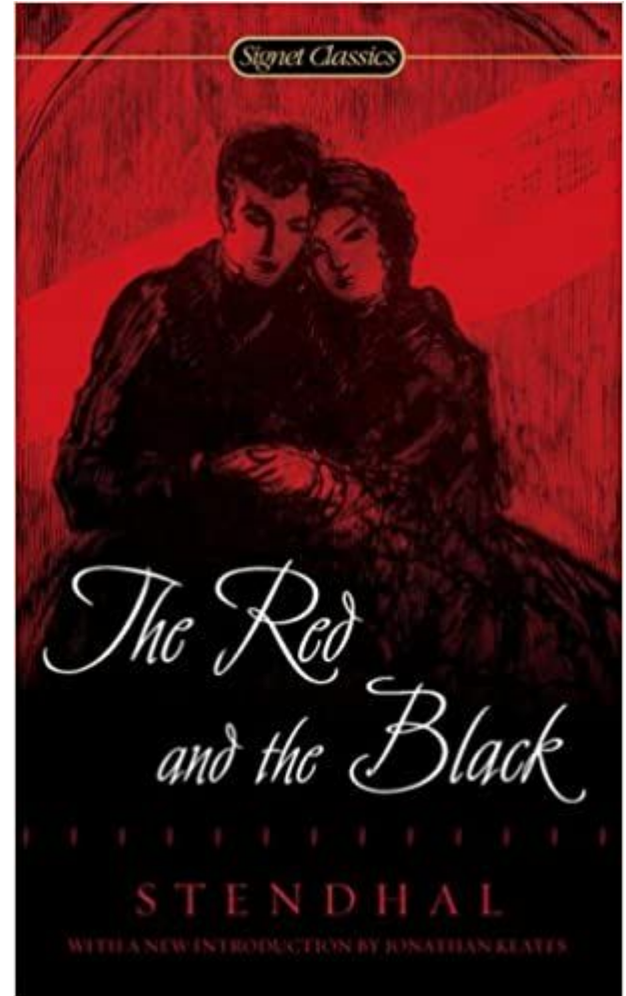
- Every node is colored **red** or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



I'm not going to draw the NIL children in the future, but they are treated as black nodes.

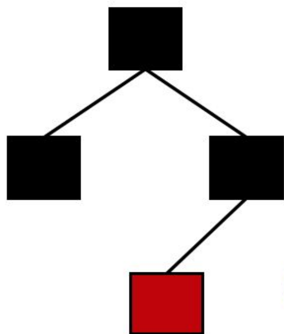
Why red and black?

Maybe something about ink? I have never found a definitive answer

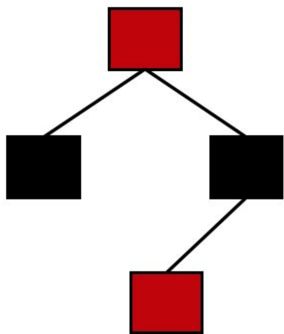


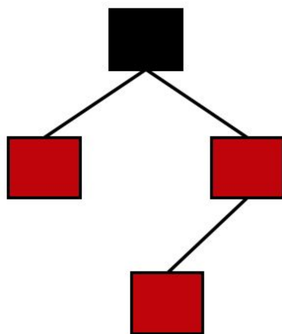
Examples(?)

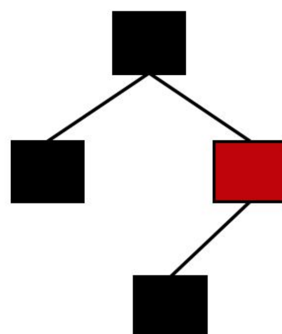
- Every node is colored **red** or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



Which of these are red-black trees? (NIL nodes not drawn)

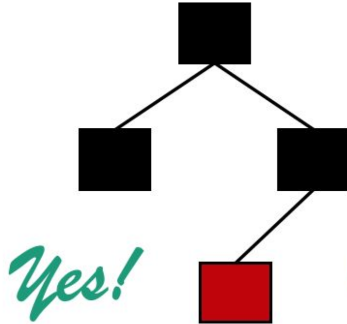




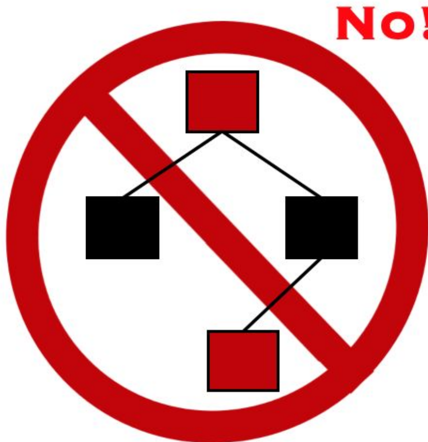


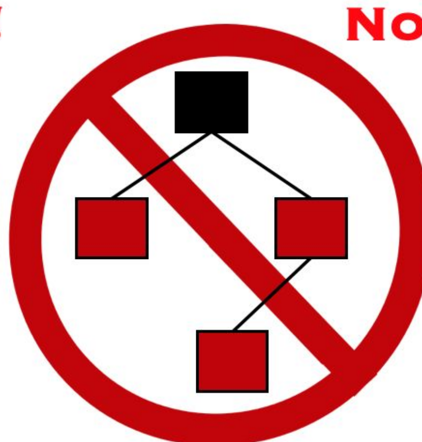
Examples(?)

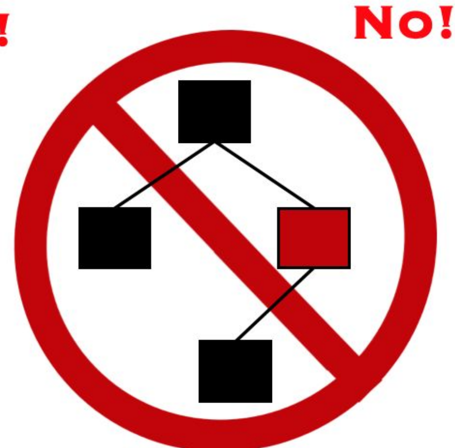
- Every node is colored red or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



Which of these are red-black trees? (NIL nodes not drawn)

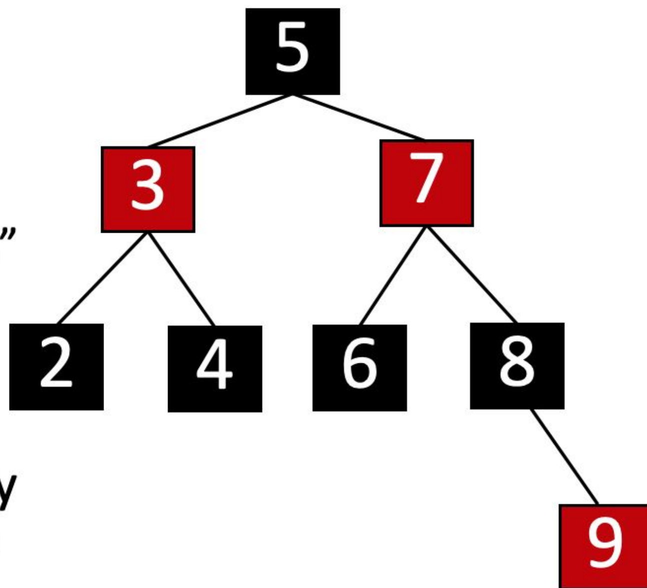






Why these rules???????

- This is pretty balanced.
  - The **black nodes** are balanced
  - The **red nodes** are "spread out" so they don't mess things up too much.



- We can maintain this property as we insert/delete nodes, by using rotations.

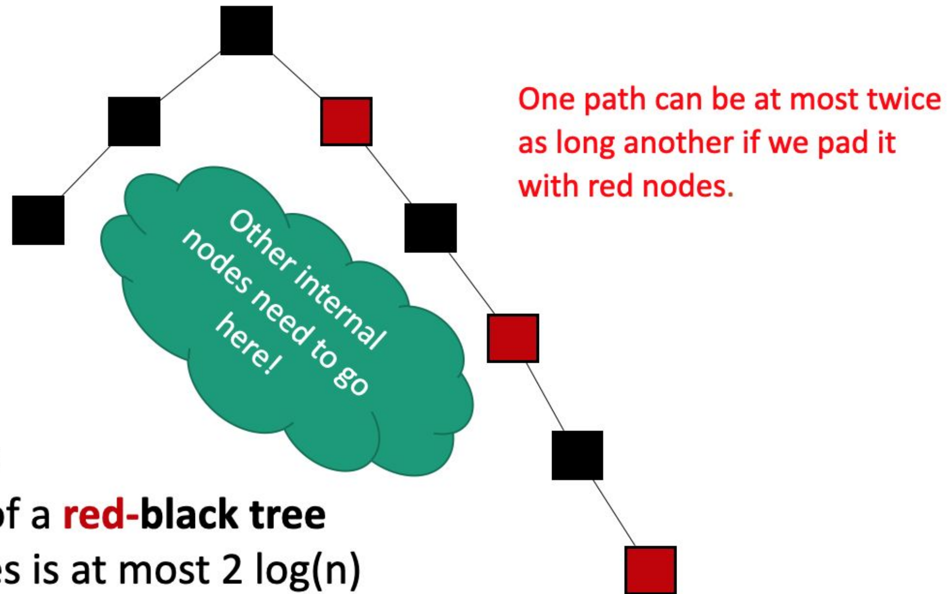
This is the really clever idea!

This **Red-Black** structure is a proxy for balance.

It's just a smidge weaker than perfect balance, but we can actually maintain it!

This is “pretty balanced”

- To see why, intuitively, let’s try to build a Red-Black Tree that’s unbalanced.

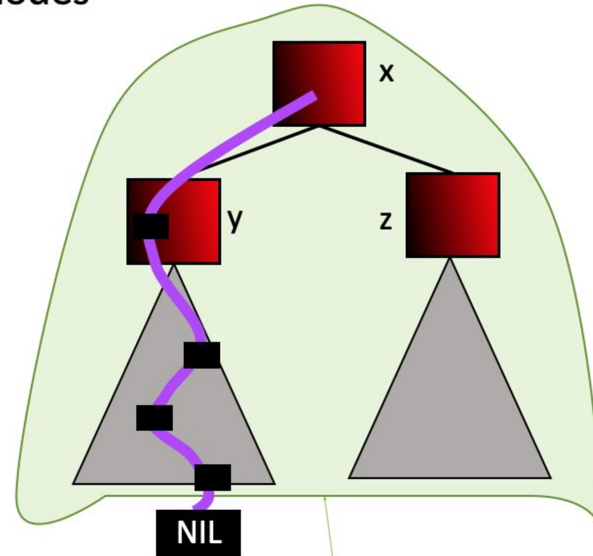


One path can be at most twice as long another if we pad it with red nodes.

Conjecture:
the height of a **red-black tree** with n nodes is at most 2 log(n)

The height of a RB-tree with n non-NIL nodes is at most $2\log(n+1)$

- Define b(x) to be the number of black nodes in any path from x to NIL.
  - (excluding x, including NIL).
- Claim:
  - There are at least $2^{b(x)} - 1$ non-NIL nodes in the subtree underneath x. (Including x).
- Proof by induction – optional exercise!



Claim: at least $2^{b(x)} - 1$ nodes in this WHOLE subtree (of any color).

Then:

$$n \geq 2^{b(root)} - 1$$ using the Claim

$$\geq 2^{\frac{height}{2}} - 1$$ b(root) >= height/2 because of RBTree rules.

Rearranging:

$$n + 1 \geq 2^{\frac{height}{2}} \Rightarrow height \leq 2\log(n+1)$$

This is great!

- SEARCH in an RBTree is immediately O(log(n)), since the depth of an RBTree is O(log(n)).
- What about INSERT/DELETE?
  - Turns out, you can INSERT and DELETE items from an RBTree in time O(log(n)), while *maintaining* the RBTree property.
  - That's why this is a good property!

INSERT/DELETE

- You are **not responsible** for the details of INSERT/ DELETE for RBTrees for this class.
  - You should know what the “proxy for balance” property is and why it ensures approximate balance.
  - You should know **that** this property can be efficiently maintained, but you do not need to know the details of how.
  - See CLRS for details if you’re curious!

Conclusion: The best of both worlds

	Sorted Arrays	Linked Lists	Binary Search Trees*
Search	O(log(n))	O(n)	O(log(n))
Delete	O(n)	O(n)	O(log(n))
Insert	O(n)	O(1)	O(log(n))

A red/black tree simulator!

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Red/Black Tree

Insert Delete Find Print Show Null Leaves

0007 >= 0003. Looking at right subtree

0007

0003

0001

0005

You'll use this on HW3!