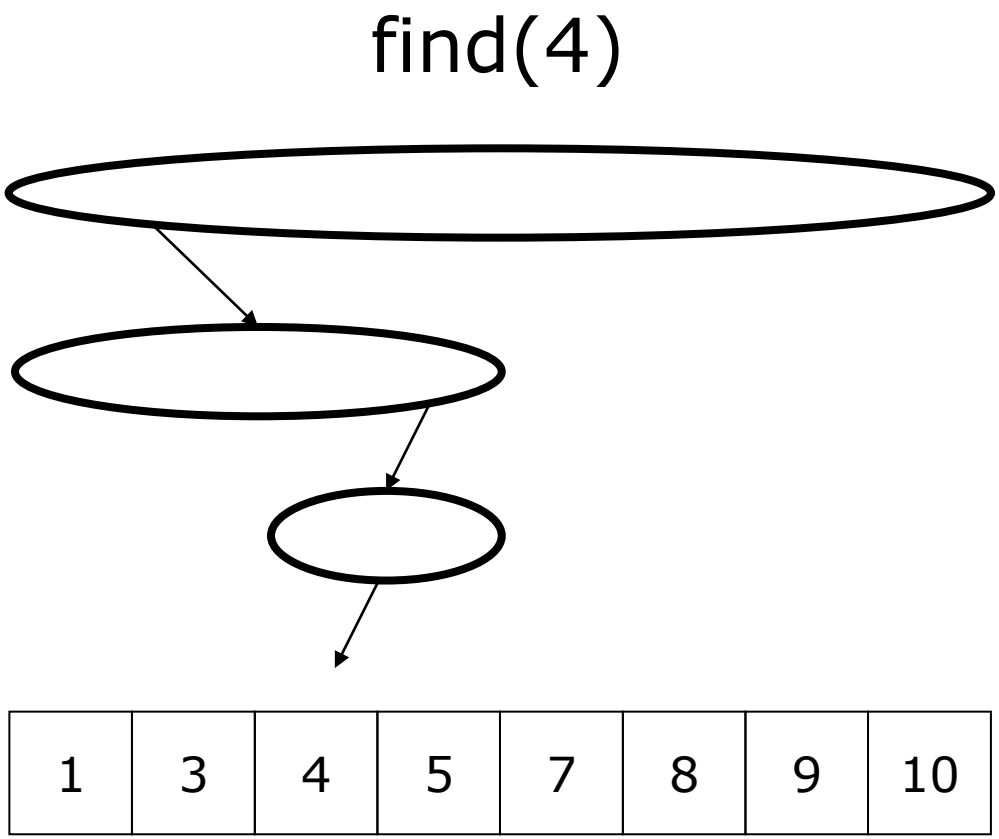
Tree Review

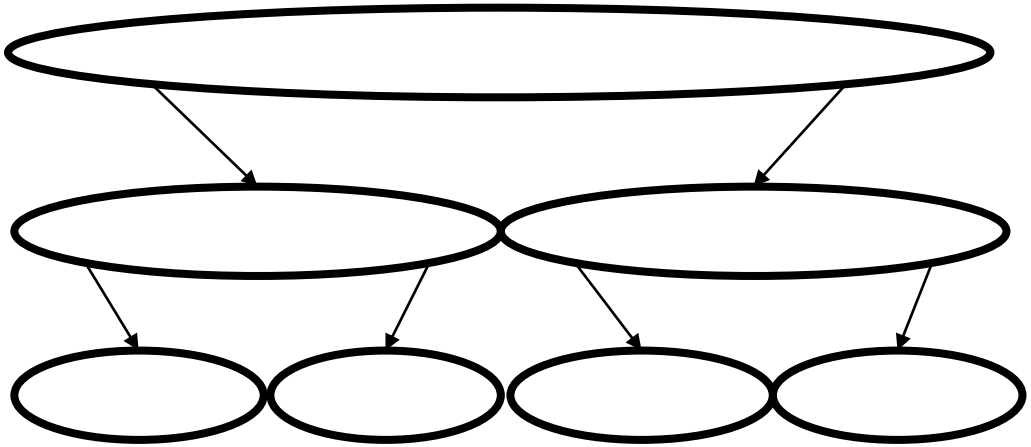
Why Trees?

- Trees offer speed-ups because of their branching factors
  - Binary Search Trees are structured forms of binary search



Binary search tree

- Our goal is the performance of binary search in a tree representation



1	3	4	5	7	8	9	10

Binary Search Tree

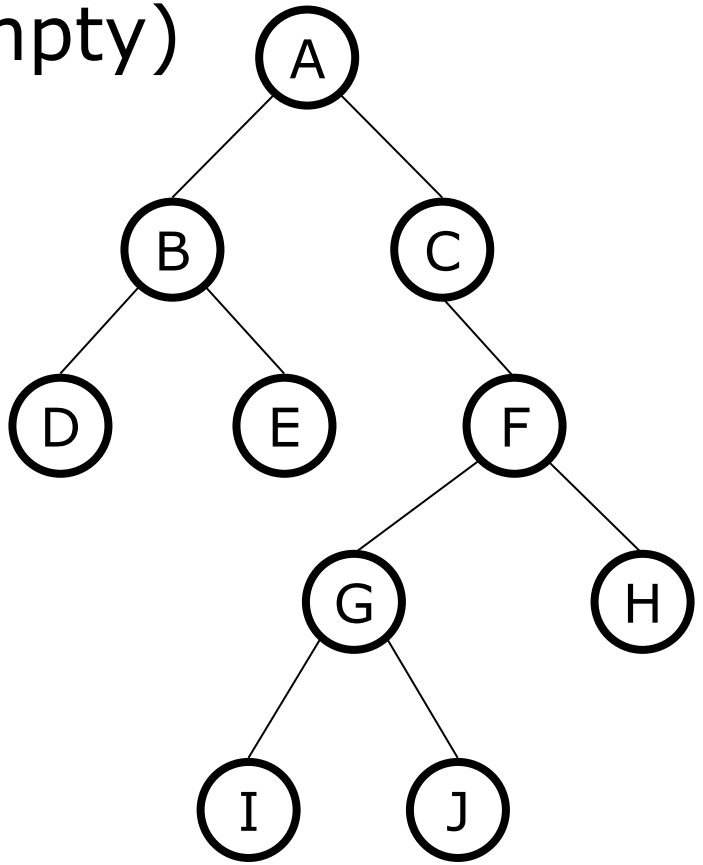
- Even a basic BST is fairly good

	Insert	Find	Delete
Worse-Case	O(n)	O(n)	O(n)
Average-Case	O(log n)	O(log n)	O(log n)

Binary Trees

A non-empty binary tree consists of a

- a root (with data)
- a left subtree (may be empty)
- a right subtree (may be empty)



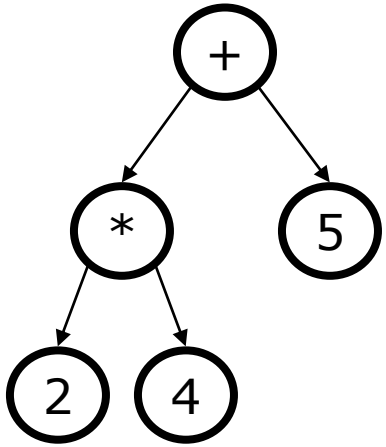
Representation:

Data	
left pointer	right pointer

- For a dictionary, data will include a key and a value

Tree Traversals

A traversal is a recursively defined order for visiting all the nodes of a binary tree



Pre-Order: root, left subtree, right subtree

+ * 2 4 5

In-Order: left subtree, root, right subtree

2 * 4 + 5

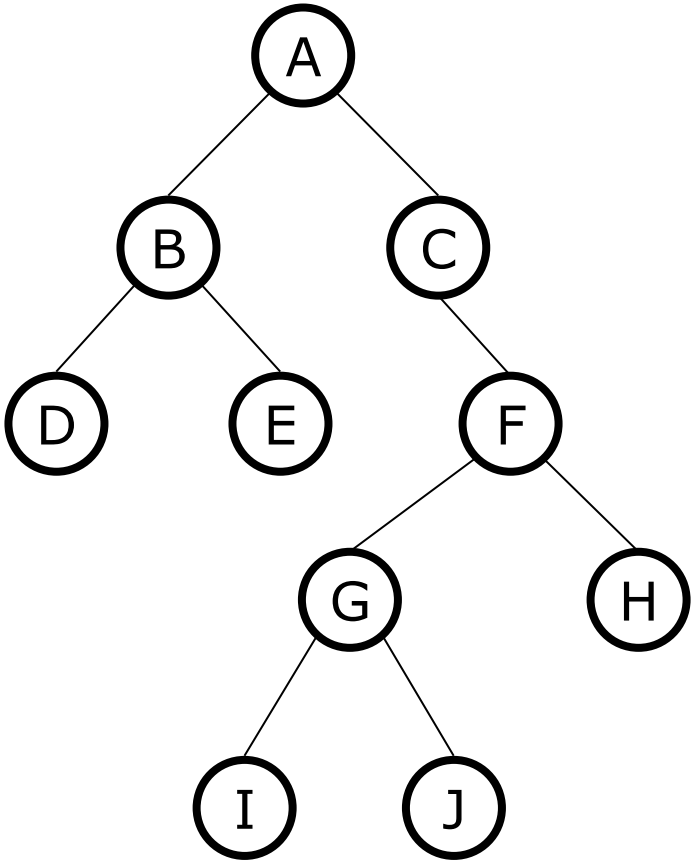
Post-Order: left subtree, right subtree, root

2 4 * 5 +

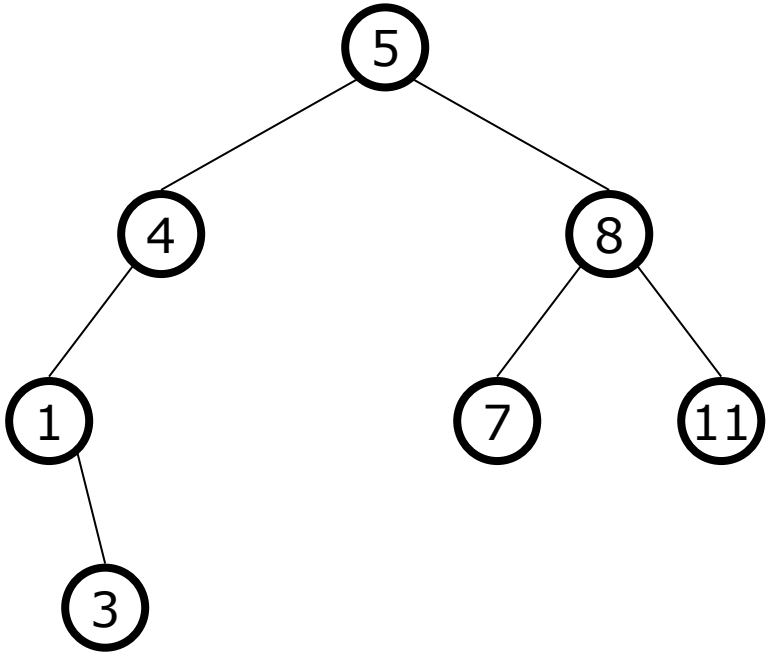
Binary Search Trees

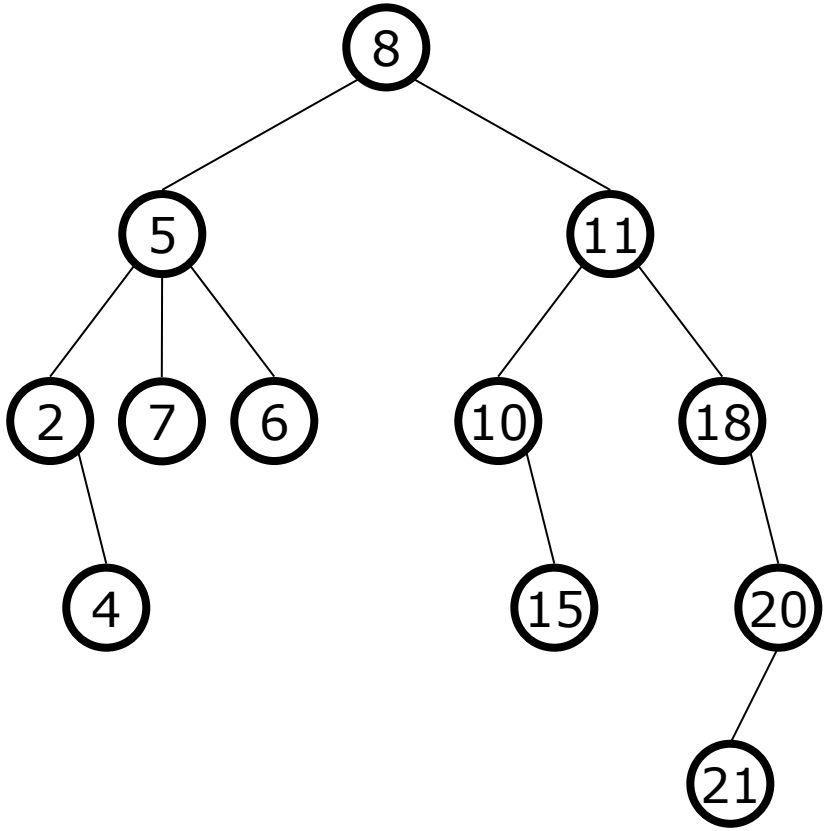
BSTs are binary trees with the following added criteria:

- Each node has a key for comparing nodes
- Keys in left subtree are smaller than node's key
- Keys in right subtree are larger than node's key

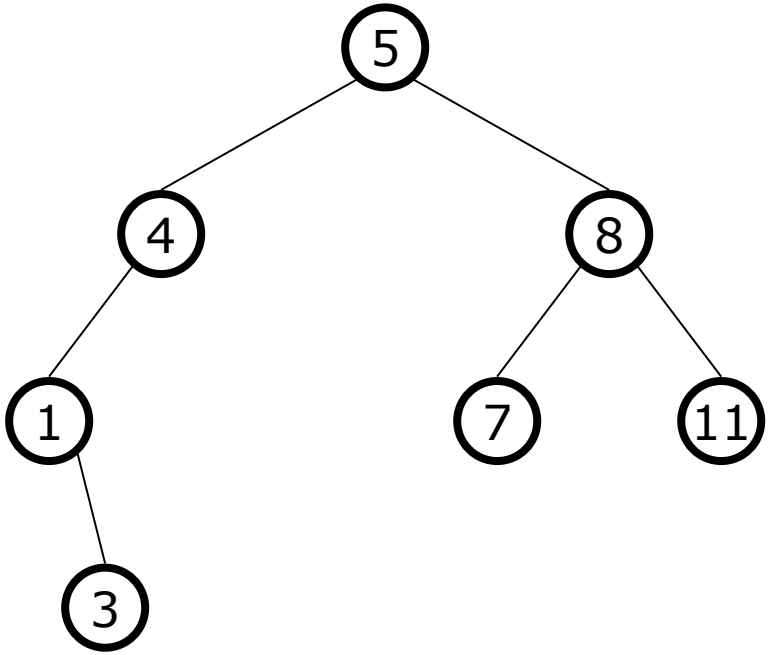


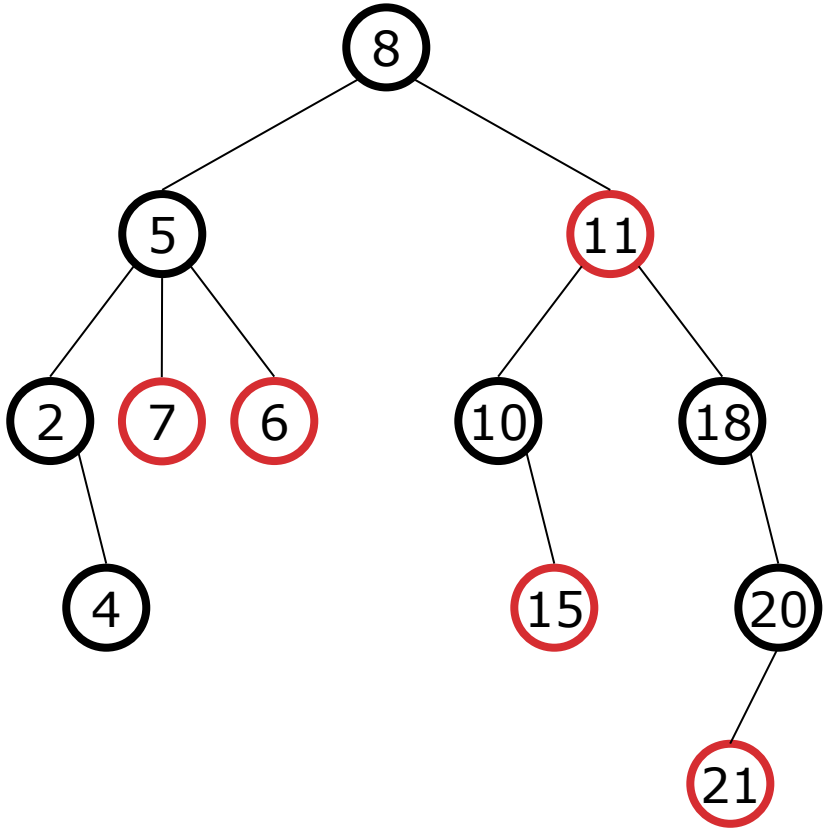
Are these BSTs?





Are these BSTs?





Calculating Height

What is the height of a BST with root r?

```
int treeHeight(Node root) {
  if(root == null)
    return -1;

  return 1 + max(treeHeight(root.left),
                 treeHeight(root.right));
}
```

Running time for tree with *n* nodes:

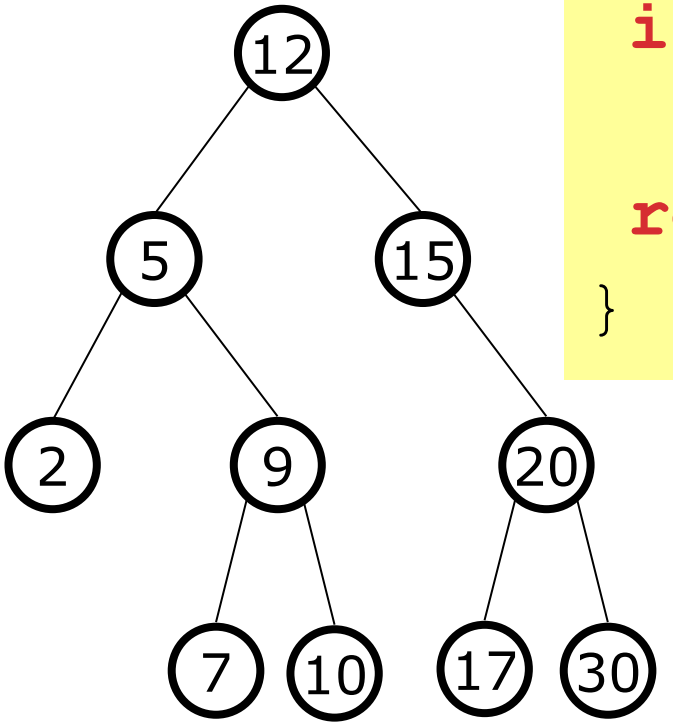
O(*n*) – single pass over tree

How would you do this without recursion?

Stack of pending nodes, or use two queues

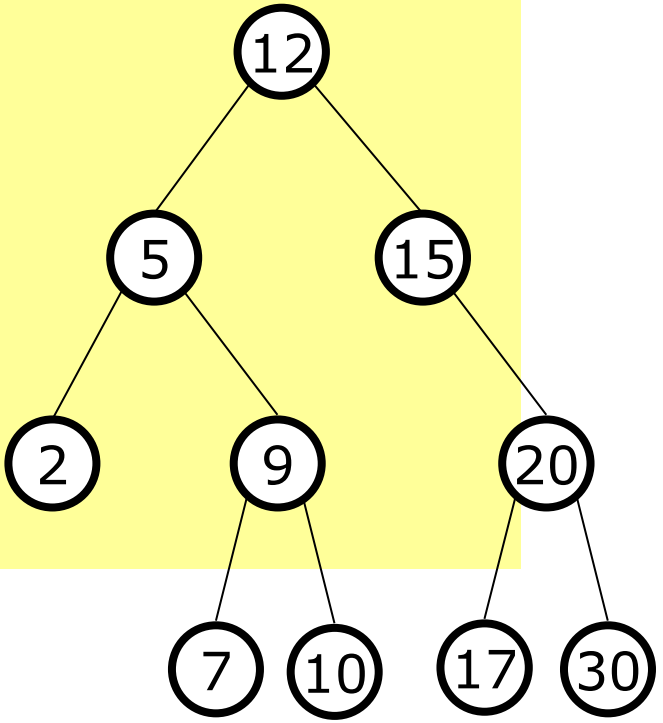
Find in BST, Recursive

```
Data find(Key key, Node root){
 if(root == null)
   return null;
 if(key < root.key)
   return find(key, root.left);
 if(key > root.key)
   return find(key, root.right);
 return root.data;
}
```



Find in BST, Iterative

```
Data find(Key key, Node root){
 while(root != null && root.key != key) {
  if(key < root.key)
     root = root.left;
  else(key > root.key)
     root = root.right;
 }
 if(root == null)
    return null;
 return root.data;
}
```



Performance of Find

We have already said it is worst-case O(n)

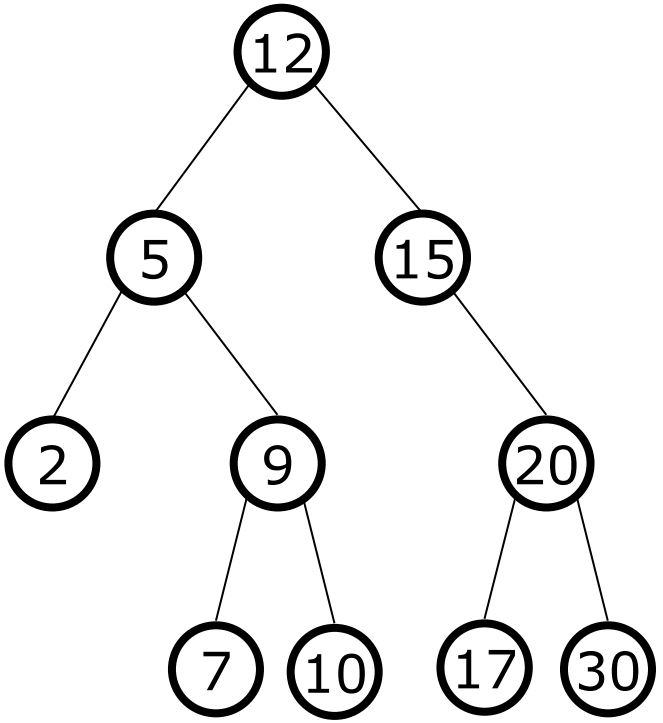
Average case is O(log n)

But if want to be exact, the time to find node x is actually Θ(depth of x in tree)

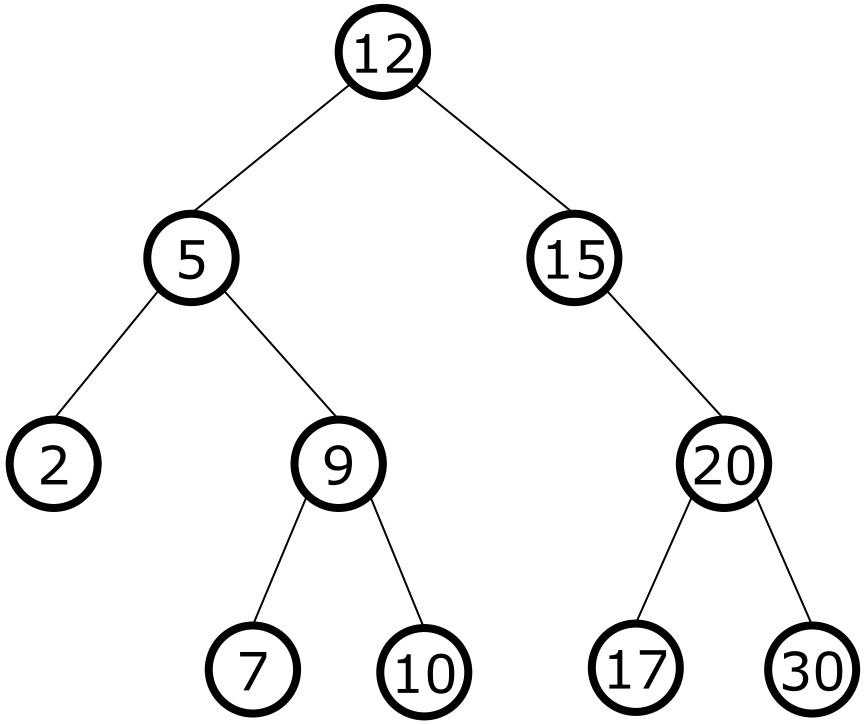
- If we can bound the depth of nodes, we automatically bound the time for find()

Other "Finding" Operations

- Find minimum node
- Find maximum node
- Find predecessor of a non-leaf
- Find successor of a non-leaf
- Find predecessor of a leaf
- Find successor of a leaf

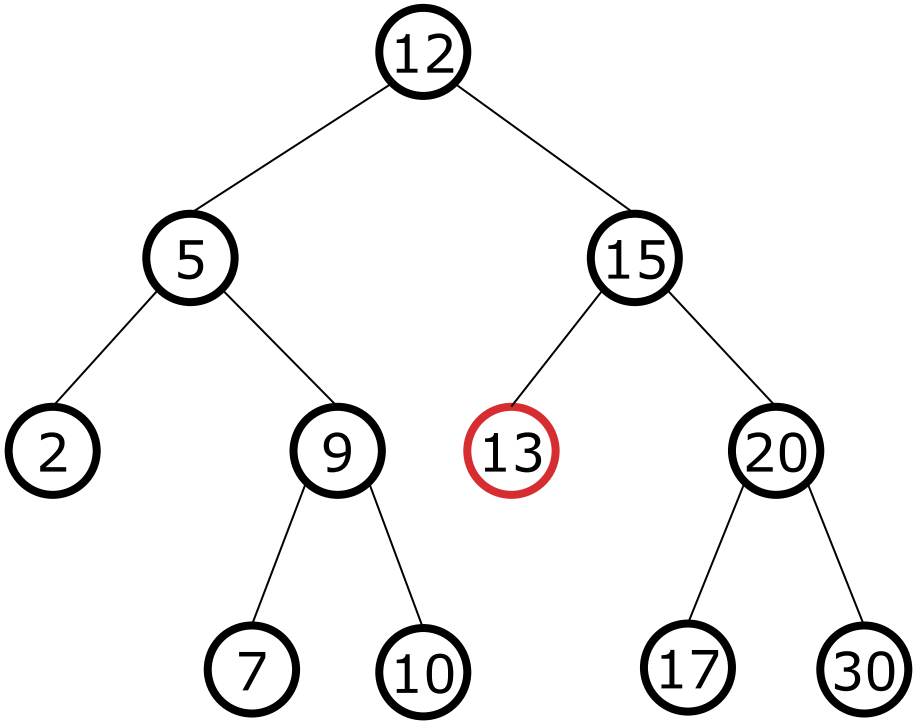


Insert in BST



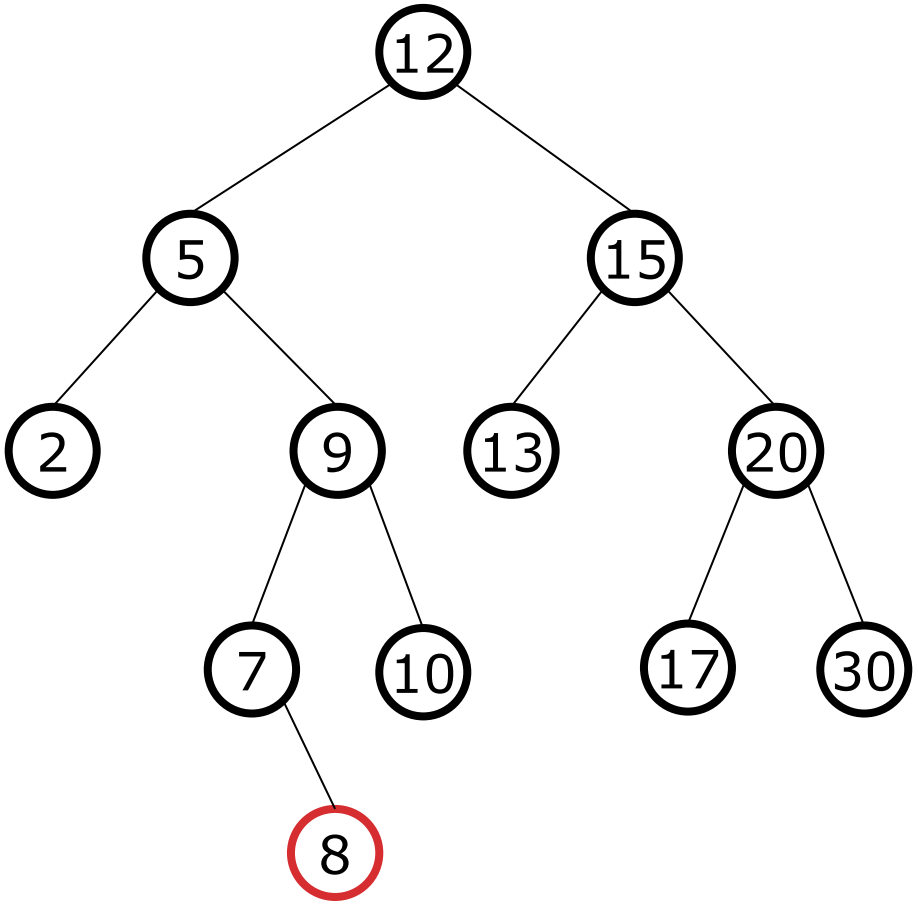

```
insert(13)
insert(8)
insert(31)
```

Insert in BST



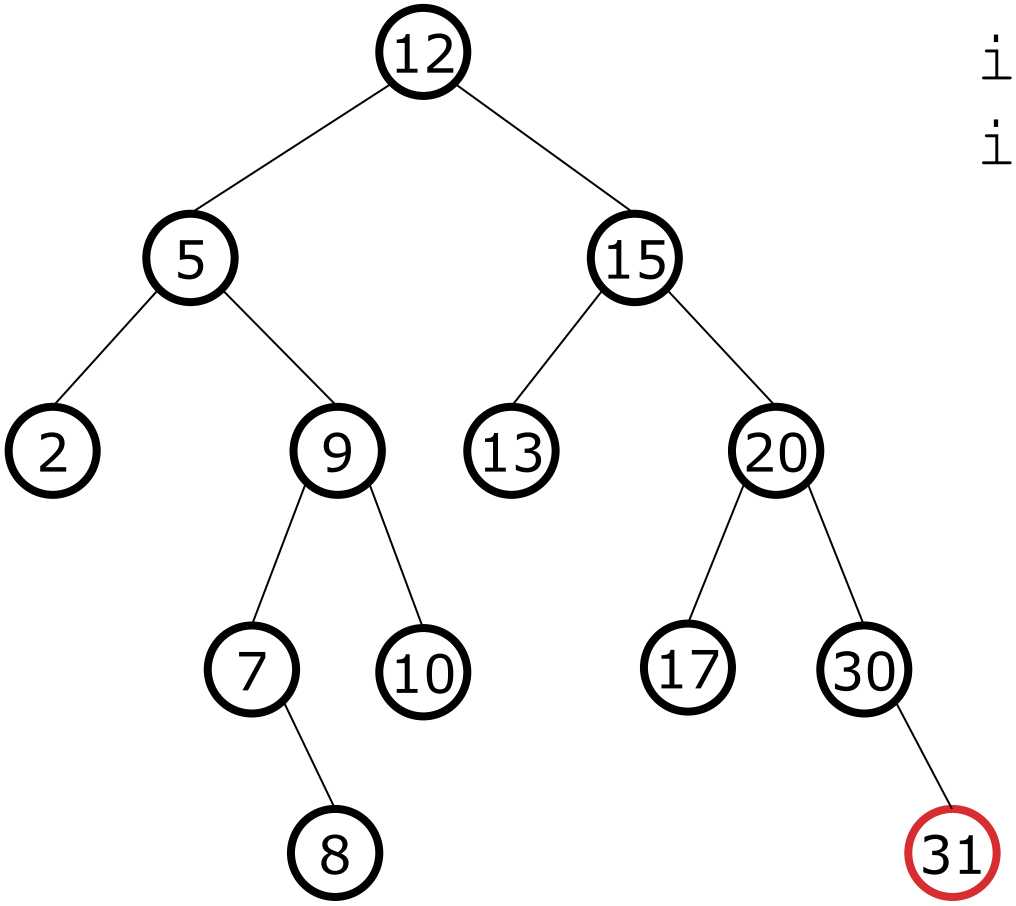

```
insert(13)
insert(8)
insert(31)
```

Insert in BST



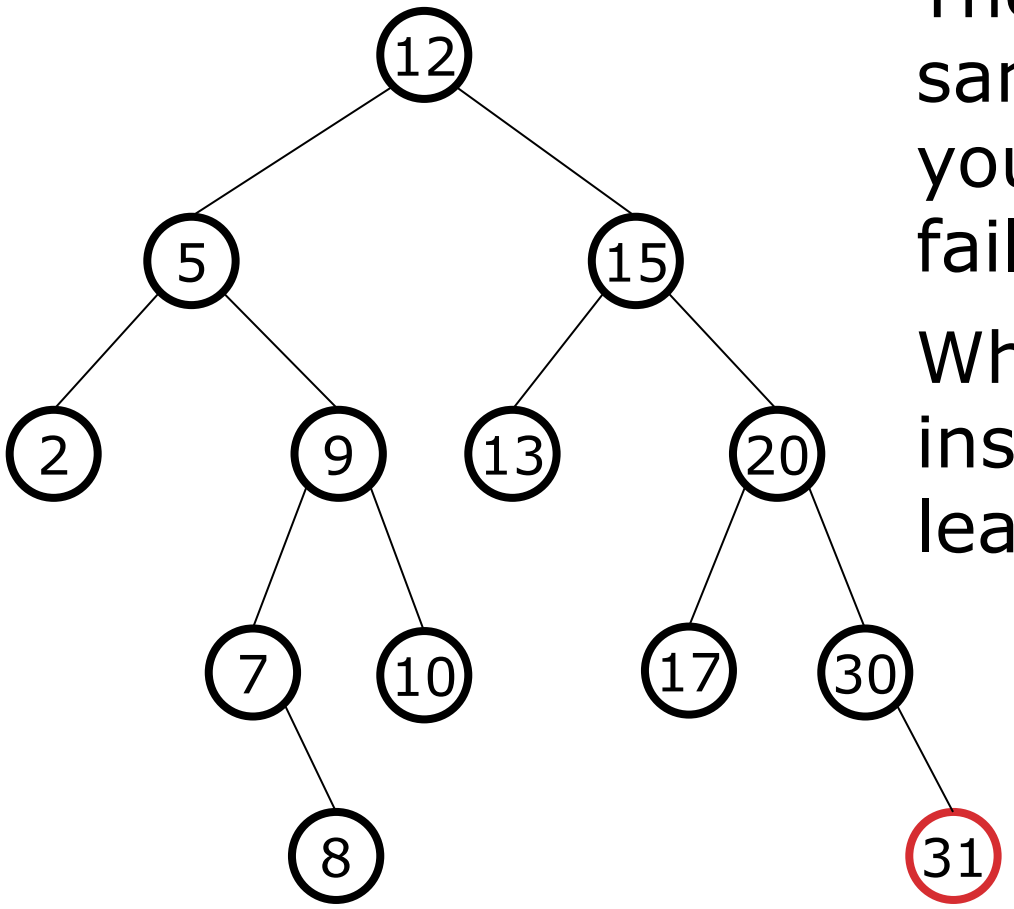

```
insert(13)
insert(8)
insert(31)
```

Insert in BST




```
insert(13)
insert(8)
insert(31)
```

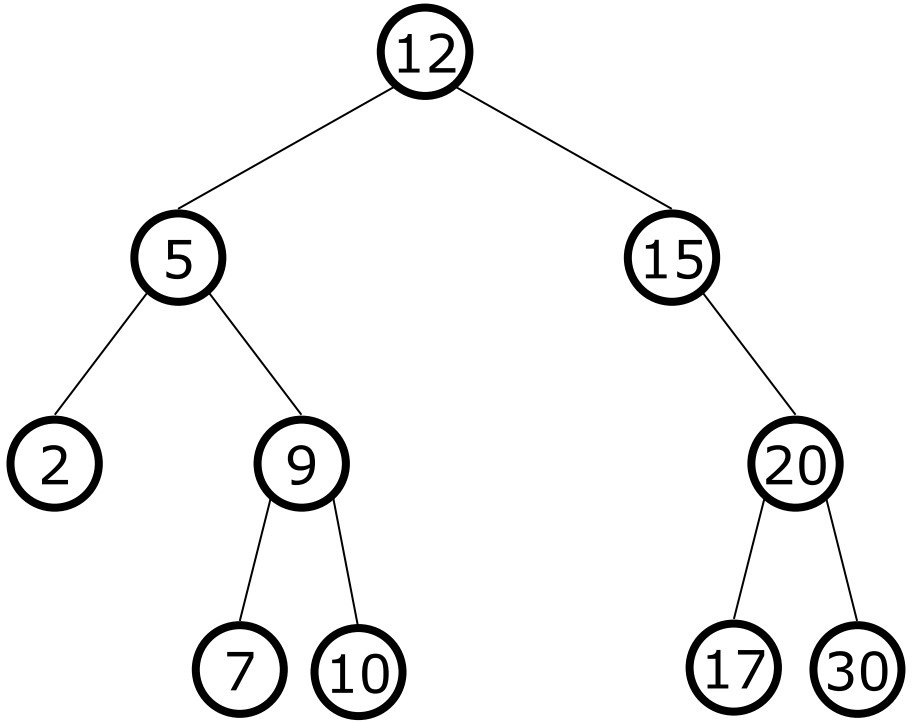
Insert in BST



The code for insert is the same as with find except you add a node when you fail to find it.

What makes it easy is that inserts only happen at the leaves.

Deletion in BST



Why might deletion be harder than insertion?

Deletion

Removing an item disrupts the tree structure

Basic idea:

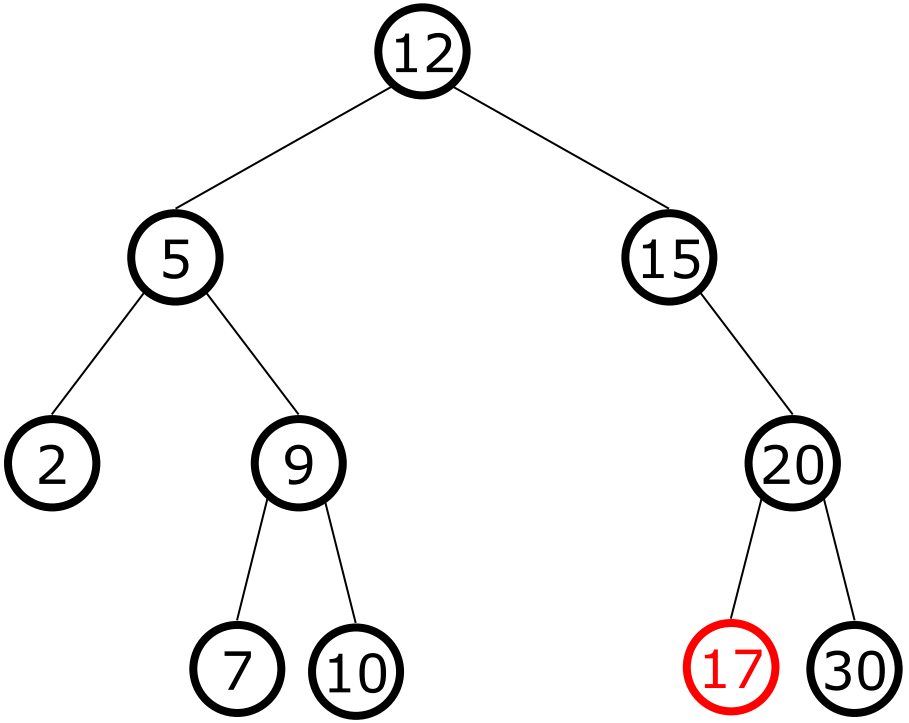
- find the node to be removed,
- Remove it
- Fix the tree so that it is still a BST

Three cases:

- node has no children (leaf)
- node has one child
- node has two children

Deletion – The Leaf Case

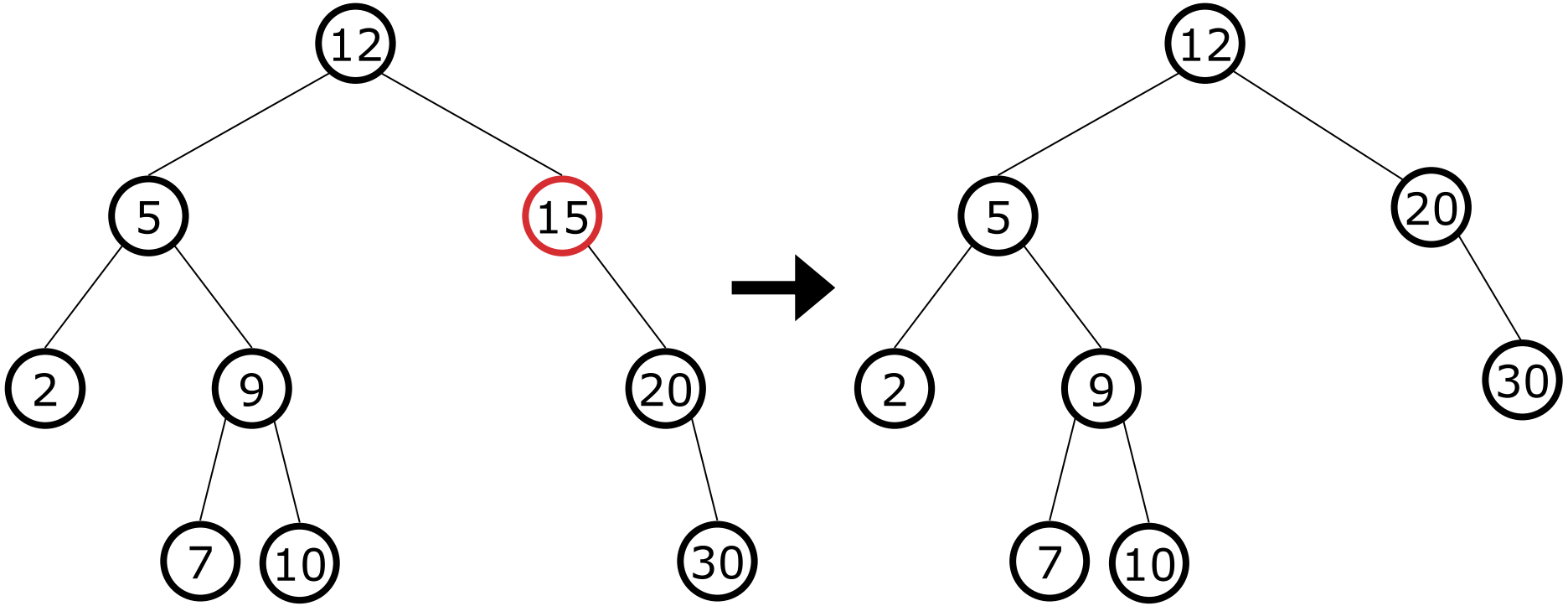
This is by far the easiest case… you just cut off the node and correct its parent



`delete(17)`

Deletion – The One Child Case

If there is only one child, we just pull up the child to take its parents place

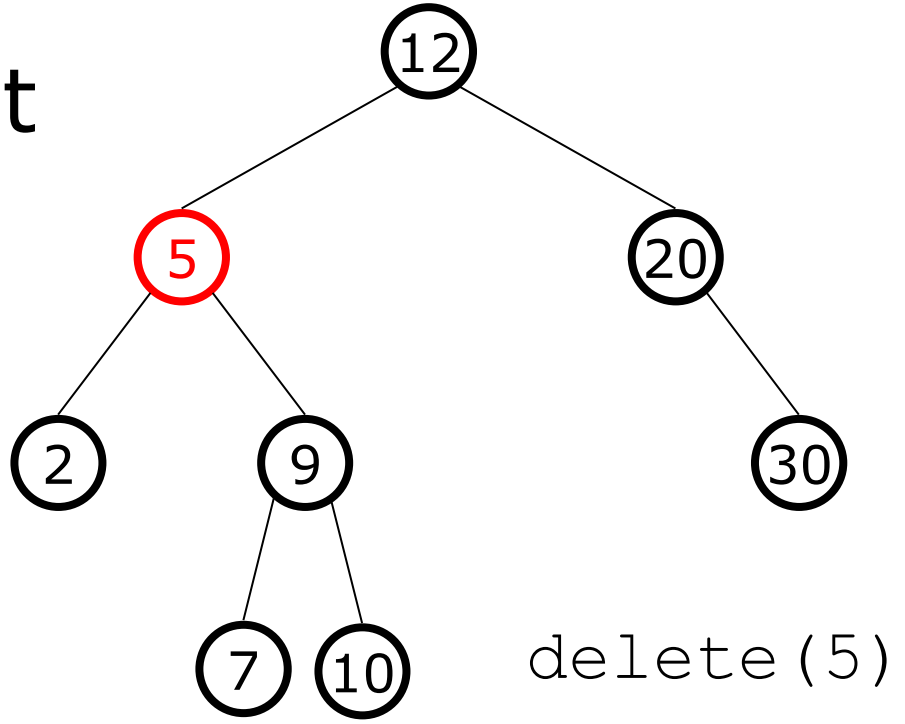


`delete(15)`

Deletion – The Two Child Case

Deleting a node with two children is the most difficult case. We need to replace the deleted node with another node.

What node is the best to replace 5 with?



Deletion – The Two Child Case

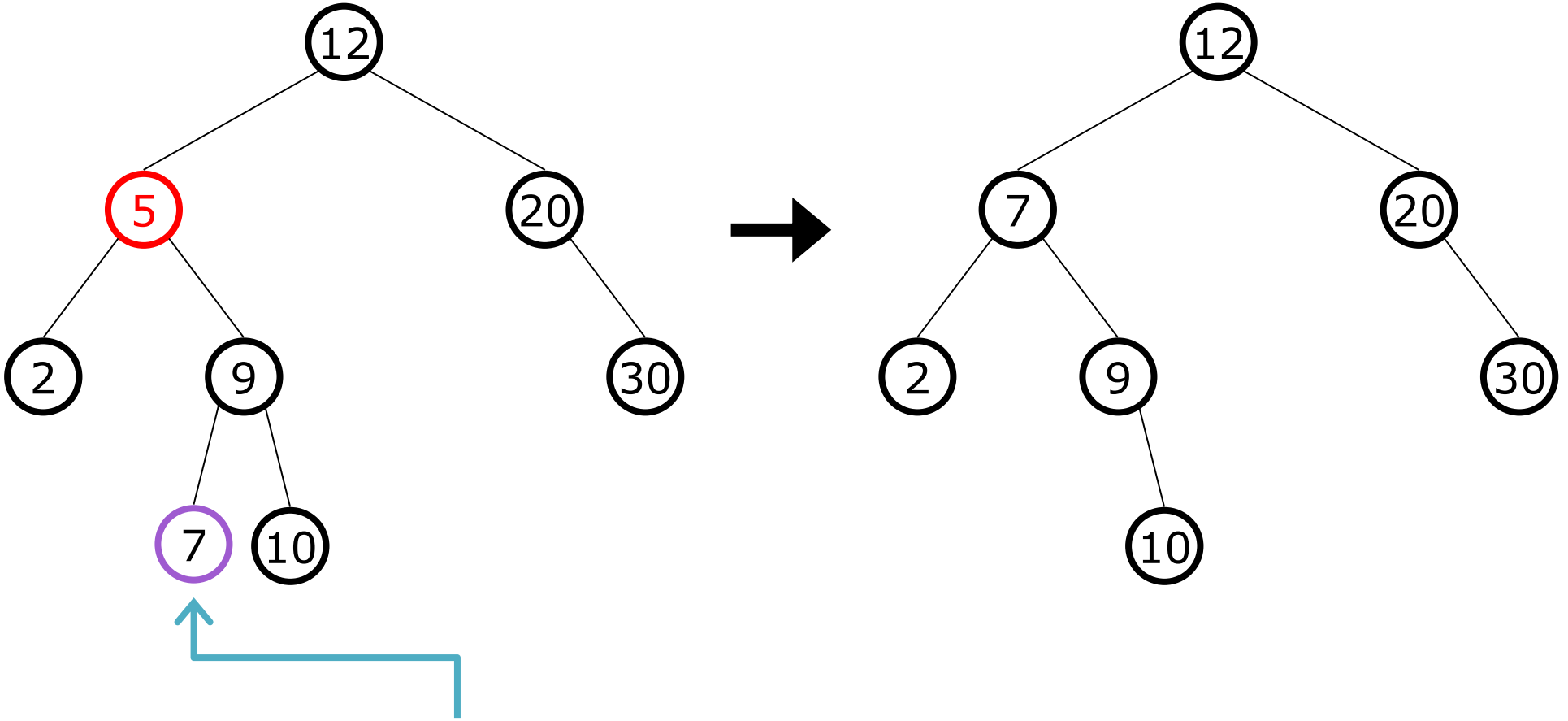
Idea: Replace the deleted node with a value guaranteed to be between the node's two child subtrees

Options are

- successor from right subtree: findMin(node.right)
- predecessor from left subtree: findMax(node.left)
- These are the easy cases of predecessor/successor

Either option is fine as both are guaranteed to exist in this case

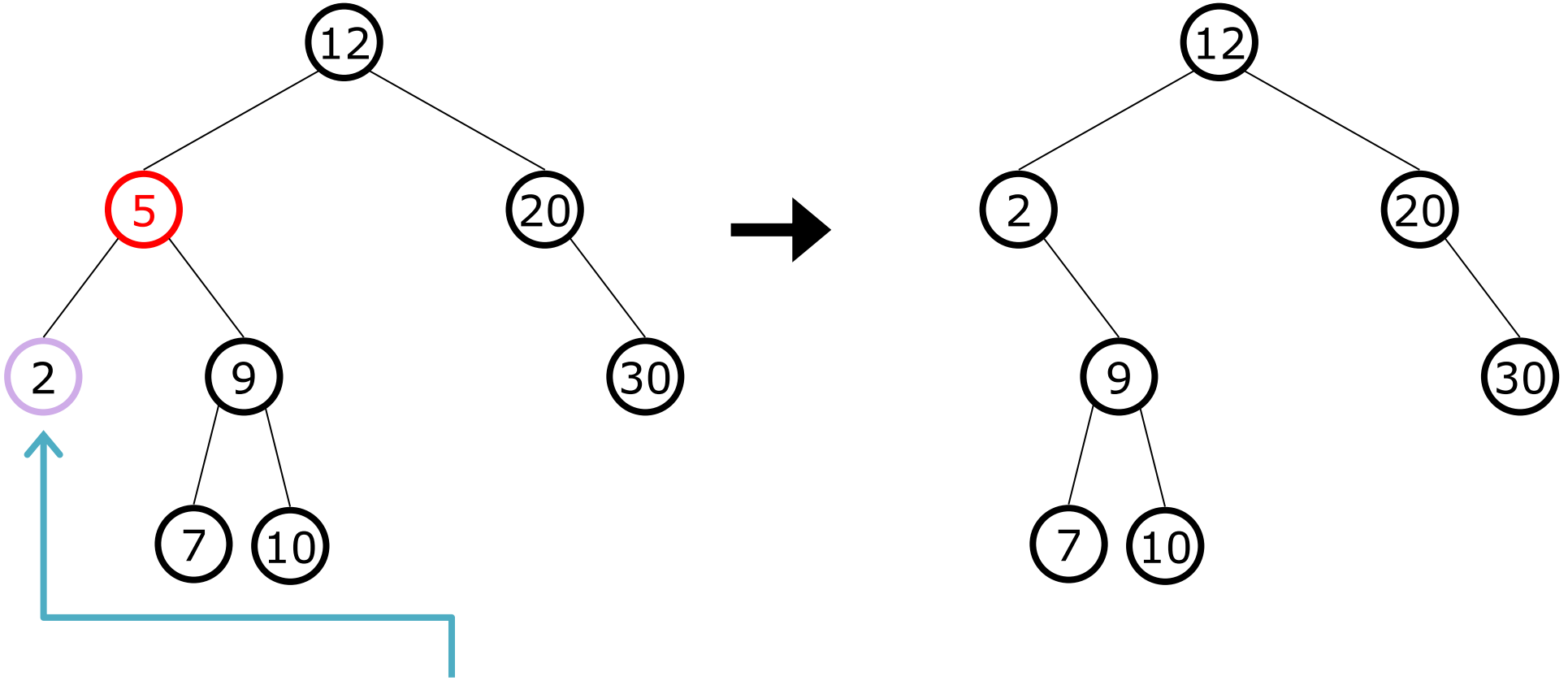
Delete Using Successor



findMin(right sub tree) → 7

```
delete(5)
```

Delete Using Predecessor



findMax(left sub tree) → 2

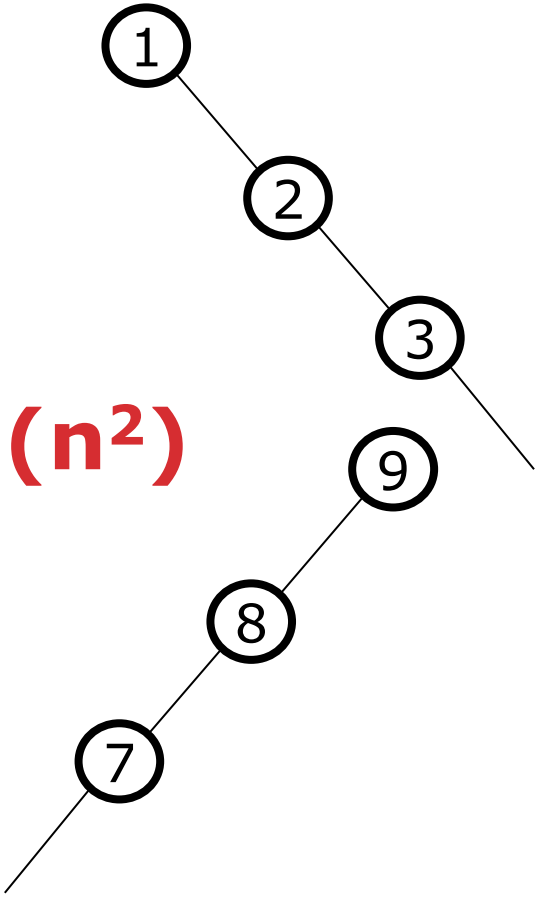
`delete(5)`

BuildTree for BST

We had buildHeap, so let's consider buildTree

Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty tree

- If inserted in given order, what is the tree?
- What big-O runtime for this kind of sorted input?
- Is inserting in the reverse order any better?



$O(n^2)$

BuildTree for BST (take 2)

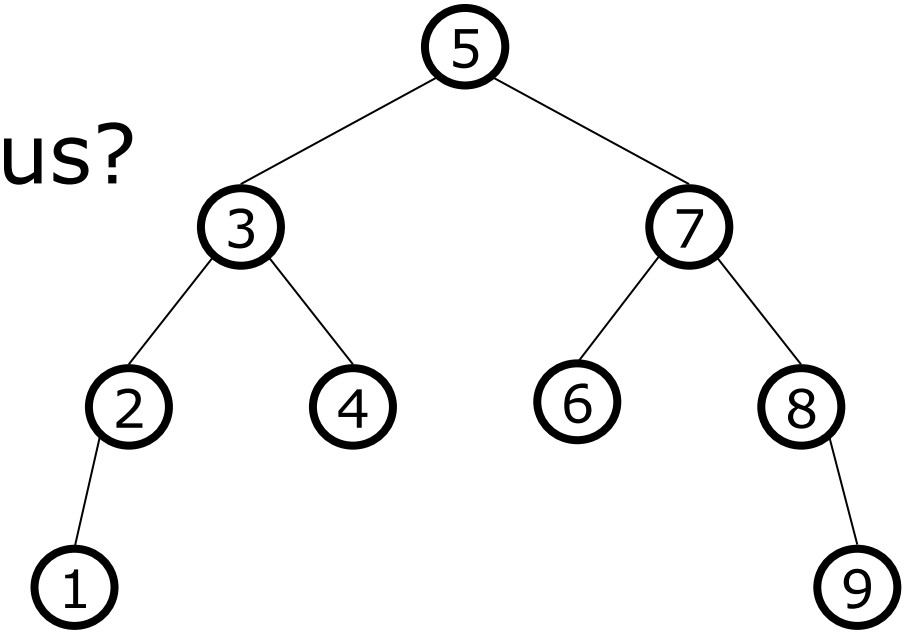
What if we rearrange the keys?

- median first, then left median, right median, etc. → 5, 3, 7, 2, 1, 4, 8, 6, 9

What tree does that give us?

What big-O runtime?

O(n log n)



Give up on BuildTree

The median trick will guarantee a O(n log n) build time, but it is not worth the effort.

Why?

- Subsequent inserts and deletes will eventually transform the carefully balanced tree into the dreaded list
- Then everything will have the O(n) performance of a linked list

Achieving a Balanced BST (part 1)

For a BST with n nodes inserted in arbitrary order

- Average height is O(log n) – see text
- Worst case height is O(n)
- Simple cases, such as pre-sorted, lead to worst-case scenario
- Inserts and removes can and will destroy the balance

Achieving a Balanced BST (part 2)

Shallower trees give better performance

- This happens when the tree's height is O(log n) ← like a perfect or complete tree

Solution: Require a **Balance Condition** that

1. ensures depth is always $O(\log n)$
2. is easy to maintain

Doing so will take some careful data structure implementation… Monday's topic

Priority Queue

Terminology

- Algorithm
  - A high-level, language-independent description of a step-by-step process
- Abstract Data Type (ADT)
  - Mathematical description of a “thing” with a set of operations
- Data structure
  - A specific family of algorithms for implementing an ADT
- Implementation of a data structure
  - A specific implementation in a specific language on a specific machine (both matter!)

Example

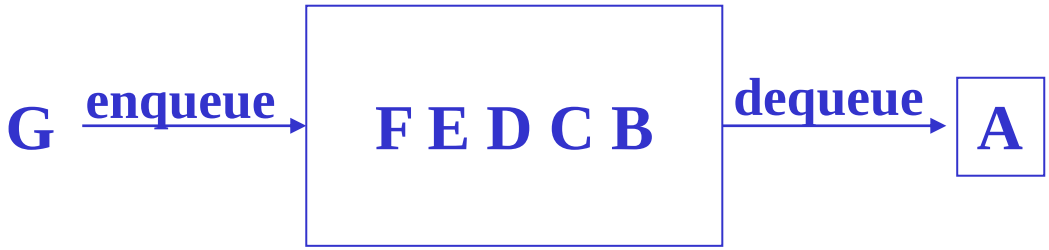
- The Stack ADT supports operations:
  - isEmpty: have there been the same number of pops as pushes
  - push: takes an item
  - pop: raises an error if isEmpty, else returns most-recently pushed item not yet returned by a pop
  - … (possibly more operations)
- A Stack data structure could use a linked list or an array or something else, and associated algorithms for the operations
- One implementation is in the library line in Java (java.util.Stack) or C++ (<stack.h>)

Stack

- It arises all the time in programming
  - Recursive function calls
  - Balancing symbols (parentheses)
  - Evaluating postfix notation: 3 4 + 5 *
  - Clever: Infix ((3+4) * 5) to postfix conversion
- We can code up a reusable library
- We can communicate in high-level terms
  - "Use a stack and push numbers, popping for operators…" rather than, "create a linked list and add a node when…"

Queue

- Operations
  - create
  - destroy
  - enqueue
  - dequeue
  - is_empty
- Just like a stack except:
  - Stack: LIFO (last-in-first-out)
  - Queue: FIFO (first-in-first-out)
- Just as useful and ubiquitous



Queue

- Scenario

What is the difference between waiting for service at a pharmacy versus an Emergency Room?

- Pharmacies usually follow the rule First Come, First Served

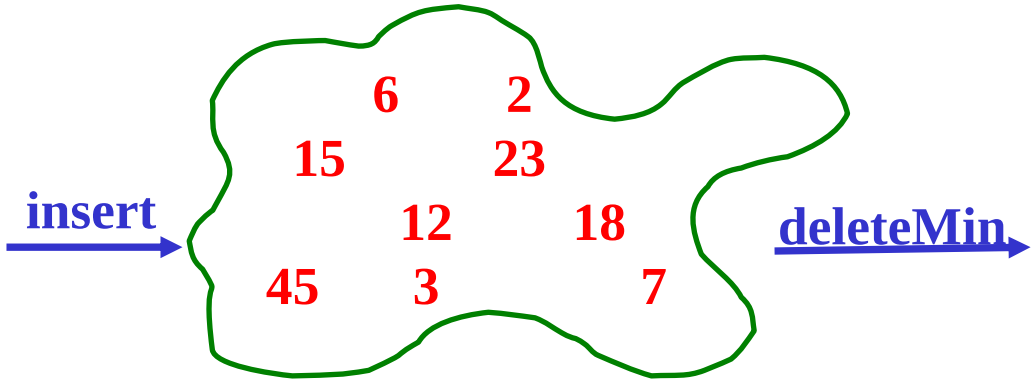
Queue

- Emergency Rooms assign priorities based on each individual's need

Priority Queue

New ADT: Priority Queue

- Each item has a "priority"
  - The next/best item has the lowest priority
  - So "priority 1" should come before "priority 4"
  - Could also do maximum priority if so desired
- Operations:
  - insert
  - deleteMin



- deleteMin returns/deletes the item with the lowest priority
  - Any ties are resolved arbitrarily
  - Fancier Priority Queues may use a FIFO approach for ties

Priority Queue

- Example

insert *a* with priority *5*

insert *b* with priority *3*

insert *c* with priority *4*

w = **deleteMin**

x = **deleteMin**

insert *d* with priority *2*

insert *e* with priority *6*

y = **deleteMin**

z = **deleteMin**

after execution:

w = *b*

x = *c*

y = *d*

z = *a*

Information

a	**5**

To simplify our examples, we will just use the priority values from now on

Priority Queue

- Applications
- Priority Queues are a major and common ADT
  - Forward network packets by urgency
  - Execute work tasks in order of priority
  - “critical” before “interactive” before “compute-intensive” tasks
  - Allocating idle tasks in cloud environments
- A fairly efficient sorting algorithm
  - Insert all items into the Priority Queue
  - Keep calling deleteMin until empty

From ADT to Data Structure

- How will we implement our Priority Queue?

	Insert	Find	Remove	Delete
Unsorted Array	O(1)	O(n)	O(1)	O(n)
Unsorted Circular Array	O(1)	O(n)	O(1)	O(n)
Unsorted Linked List	O(1)	O(n)	O(1)	O(n)
Sorted Array	O(n)	O(log n)	O(n/2)	O(n/2)
Sorted Circular Array	O(n/2)	O(log n)	O(n/2)	O(n/2)
Sorted Linked List	O(n)	O(n)	O(1)	O(n)
Binary Search Tree	O(n)	O(n)	O(n)	O(n)
Balanced BST	O(log n)	O(log n)	O(log n)	O(log n)

- We need to add one more analysis to the above: finding the min value

From ADT to Data Structure

- Finding the Minimum Value

	FindMin
Unsorted Array	O(n)
Unsorted Circular Array	O(n)
Unsorted Linked List	O(n)
Sorted Array	O(1)
Sorted Circular Array	O(1)
Sorted Linked List	O(1)
Binary Search Tree	O(n)
Balanced BST	O(log n)

From ADT to Data Structure

- Best Choice for the Priority Queue?

	Insert	FindMin+Remove
Unsorted Array	O(1)	O(n)+O(1)
Unsorted Circular Array	O(1)	O(n)+O(1)
Unsorted Linked List	O(1)	O(n)+O(1)
Sorted Array	O(n)	O(1)+O(n)
Sorted Circular Array	O(n/2)	O(1)+O(n/2)
Sorted Linked List	O(n)	O(1)+O(1)
Binary Search Tree	O(n)	O(n)+O(n)
Balanced BST	O(log n)	O(log n)+O(log n)

From ADT to Data Structure

We generally have to pay linear time for either insert or deleteMin

- Made worse by the fact that:
  # inserts ≈ # deleteMins

Balanced trees seem to be the best solution with O(log n) time for both

- But balanced trees are complex structures
- Do we really need all of that complexity?

Heap


MOHAMMED VI
POLYTECHNIC
UNIVERSITY
UM6P-CS
SCHOOL OF
COMPUTER SCIENCE

Heap

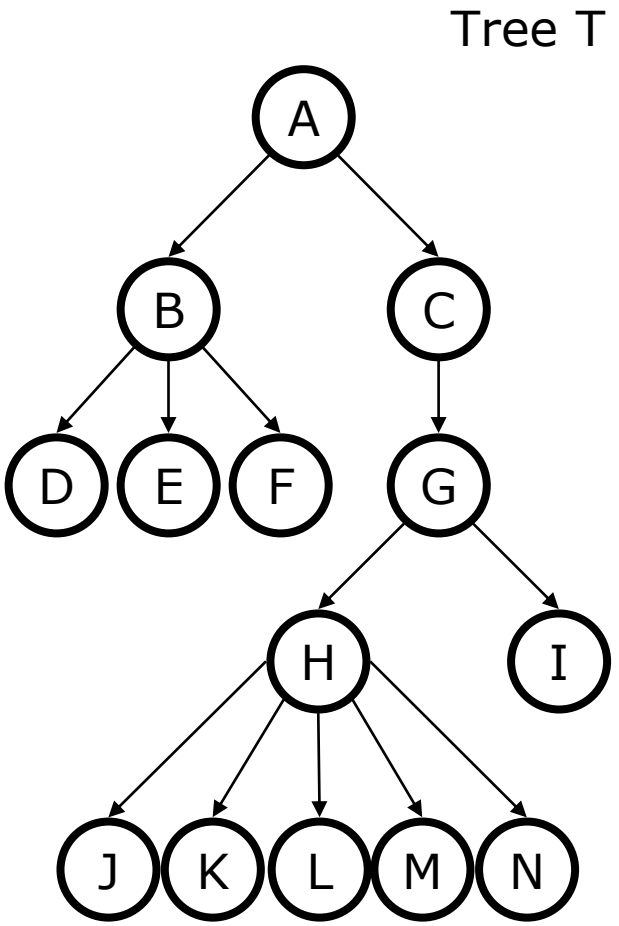
- Key idea: Only pay for functionality needed
  - Do better than scanning unsorted items
  - But do not need to maintain a full sort

- The Heap:
  - O(log n) insert and O(log n) deleteMin
  - If items arrive in random order, then the average-case of insert is O(1)
  - Very good constant factors

Trees

- Terminology

root(**T**):	A
leaves(**T**):	D-F, I, J-N
children(B):	D, E, F
parent(H):	G
siblings(E):	D, F
ancestors(F):	B, A
descendents(G):	H, I, J-N
subtree(G):	G and its children



Trees

- Terminology

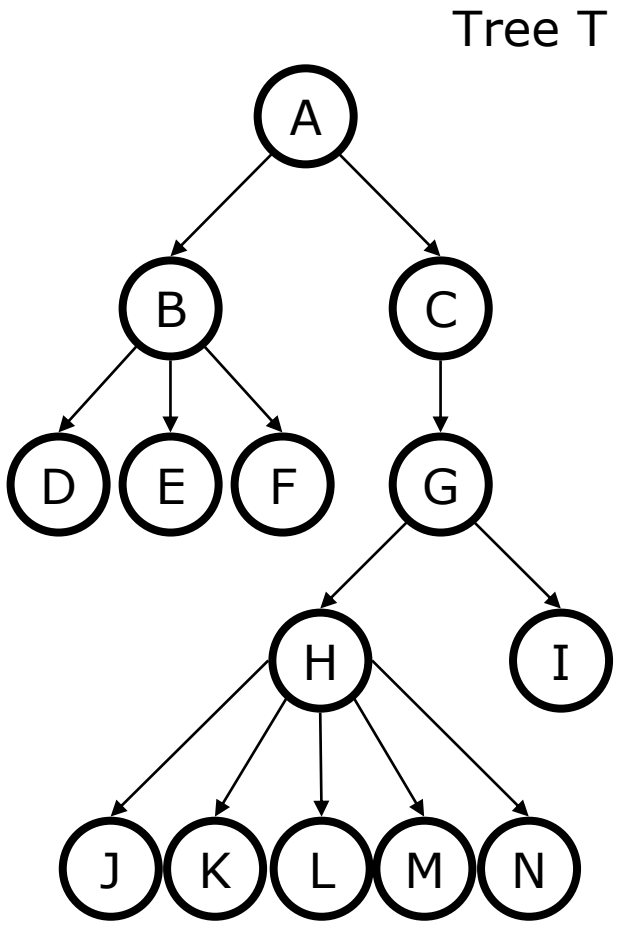
depth(B): *1*

height(G): *2*

height(**T**): *4*

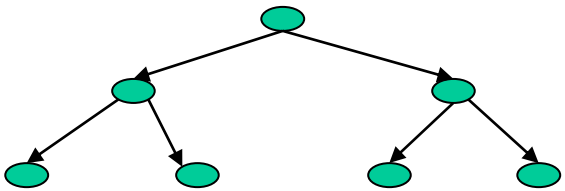
degree(B): *3*

branching factor(**T**): *0-5*

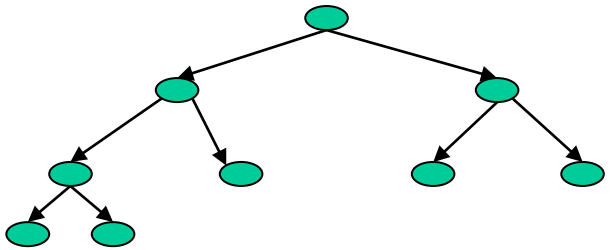


Trees

- Types
- Binary tree: Every node has ≤2 children
- n-ary tree: Every node as ≤n children
- Perfect tree: Every row is completely full
- Complete tree: All rows except the bottom are completely full, and it is filled from left to right



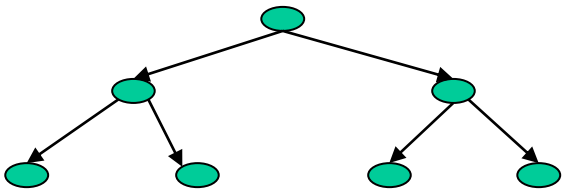
Perfect Tree



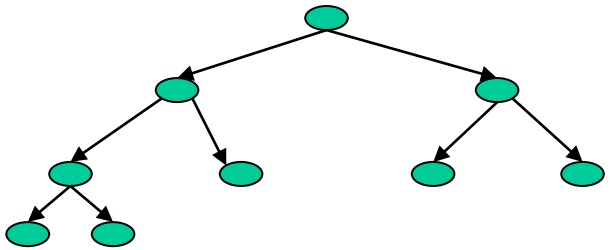
Complete Tree

Trees

- Types
- A height-balanced binary tree is a binary tree in which the height of the left subtree and right subtree of any node does not differ by more than 1 and both the left and right subtree are also height-balanced.



Perfect Tree



Complete Tree

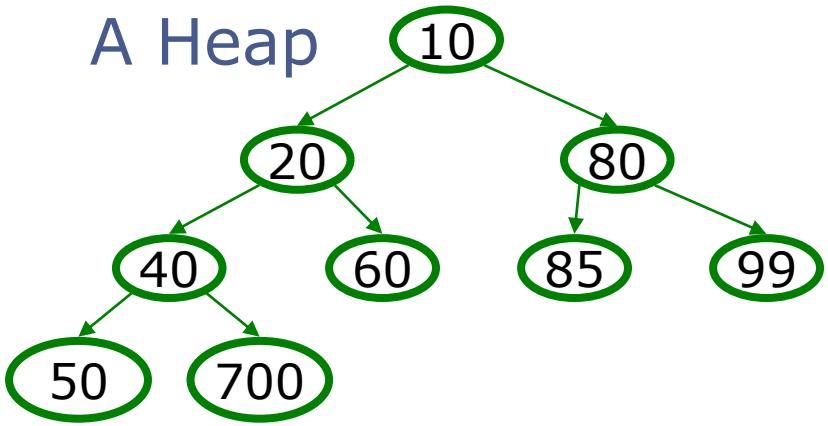
Properties of a Binary Min-Heap

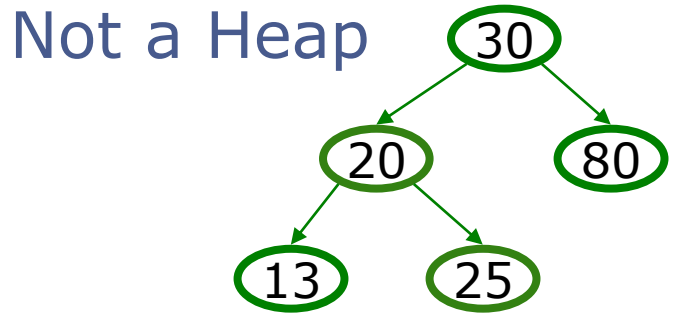
- More commonly known as a binary heap or simply a heap
  - Structure Property: A complete [binary] tree
  - Heap Property: The priority of every non-root node is greater than the priority of its parent

  - How is this different from a binary search tree?

Properties of a Binary Min-Heap

- More commonly known as a binary heap or simply a heap
  - Structure Property: A complete [binary] tree
  - Heap Property: The priority of every non-root node is greater than the priority of its parent





Properties of a Binary Min-Heap

findMin:

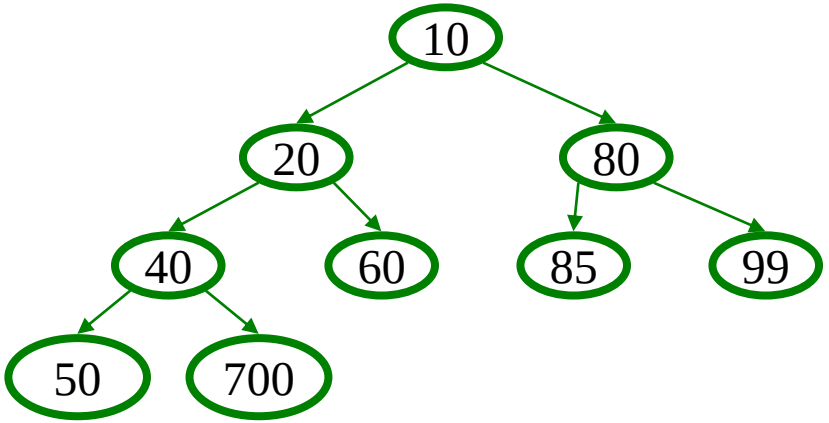
- return root.data

deleteMin:

- Move last node up to root
- Violates heap property, so *Percolate Down* to restore

insert:

- Add node after last position
- Violate heap property, so *Percolate Up* to restore



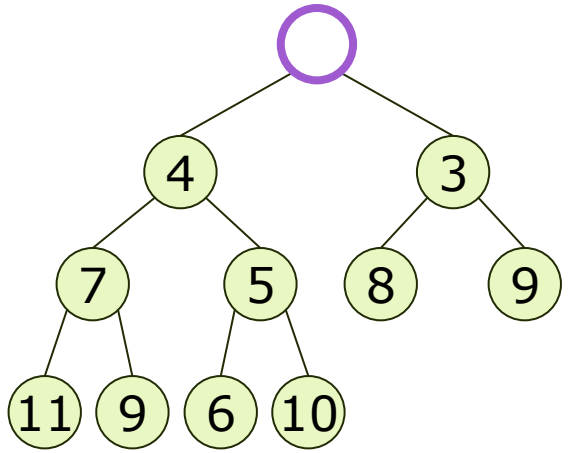
The general idea:

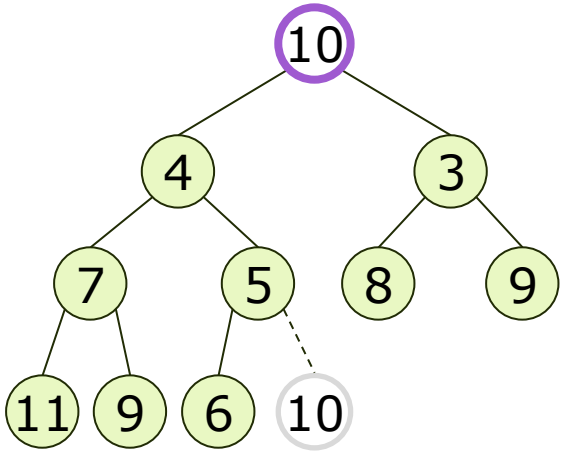
- Preserve the complete tree structure property
- This likely breaks the heap property
- So percolate to restore the heap property

DeleteMin Implementation

1. Delete value at root node (and store it for later return)
2. There is now a "hole" at the root. We must "fill" the hole with another value, must have a tree with one less node, and it must still be a complete tree
3. The "last" node is the is obvious choice, but now the heap property is violated
4. We percolate down to fix the heap

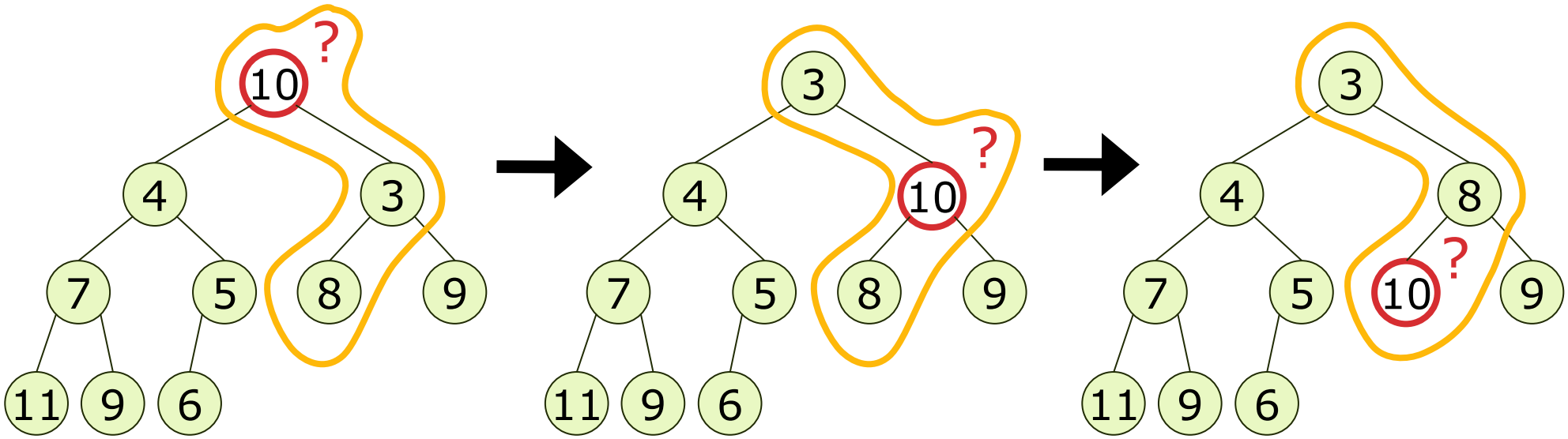
```
While greater than either child
    Swap with the smaller child
```





Percolate Down

```
While greater than either child
   Swap with the smaller child
```



What is the runtime?

O(log n)

Why does this work?

Both children are heaps

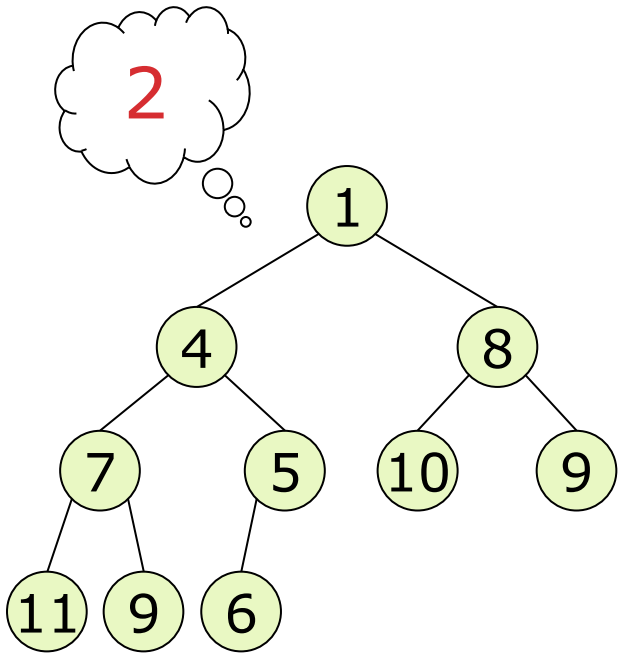
Insert Implementation

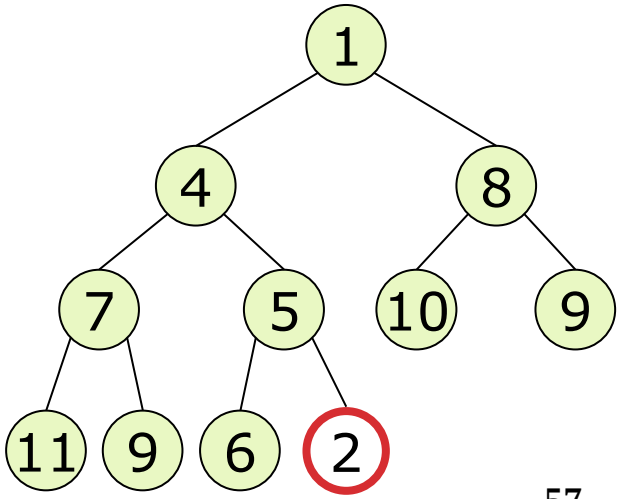
1. To maintain structure property, there is only one node we can insert into that keeps the tree complete.

   We put our new data there.

2. We then percolate up to fix the heap property:

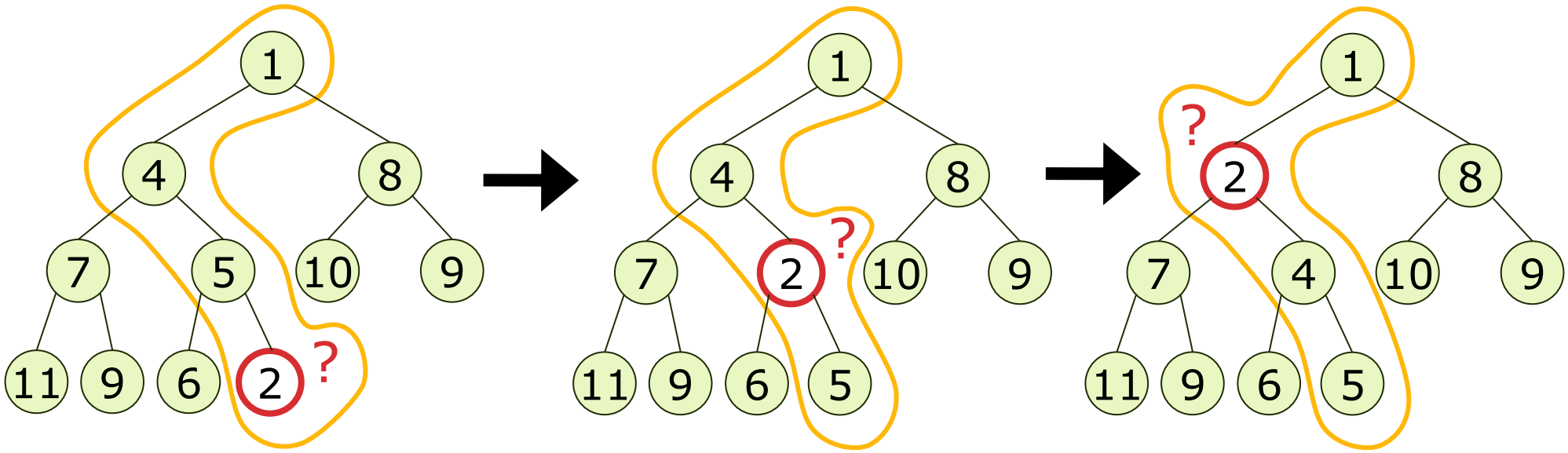
```
While less than parent
   Swap with parent
```





Percolate Up

```
While less than the parent
   Swap with parent
```



What is the runtime?

O(log n)

Why does this work?

Both children are heaps

Achieving Average-Case O(1) insert

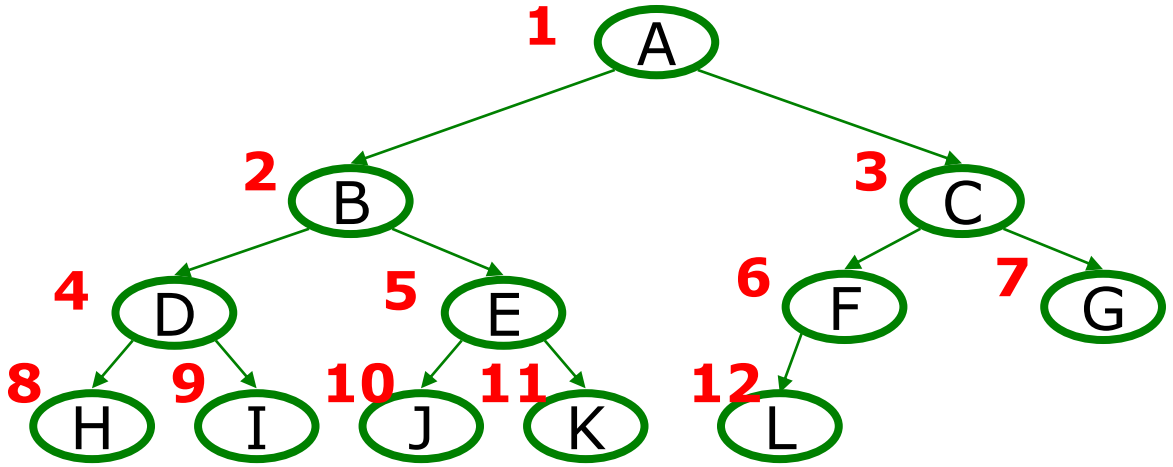
- Clearly, insert and deleteMin are worst-case O(log n)
  - Can we do better ?
- Insert only requires finding that one special spot
  - Walking the tree requires O(log n) steps
- We should only pay for the functionality we need
  - Why have we insisted the tree be complete?
- All complete trees of size n contain the same edges
  - So why are we even representing the edges?

Here comes the really clever bit about implementing heaps!!!

Array Representation of a Binary Heap

From node i:

- left child: 2i
- right child: 2i+1
- parent: i / 2



	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

- We skip index 0 to make the math simpler
- Actually, it can be a good place to store the current size of the heap

Pseudocode: insert

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
void insert(int val) {
  if(size == arr.length-1)
    resize();
  size = size + 1;
  i = percolateUp(size, val);
  arr[i] = val;
}
```

```
int percolateUp(int hole, int val) {
  while(hole > 1 && val < arr[hole/2])
    arr[hole] = arr[hole/2];
    hole = hole / 2;
  }
  return hole;
}
```

Pseudocode: deleteMin

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
int deleteMin() {
  if(isEmpty()) throw… // error
  ans = arr[1];
  hole = percolateDown(1,arr[size]);
  arr[hole] = arr[size];
  size--;
  return ans;
}
```

Pseudocode: deleteMin

```
int percolateDown(int hole, int val) {
  while(2 * hole <= size) {
    left  = 2 * hole;
    right = left + 1;
    if(arr[left] < arr[right] || right > size)
      target = left;
    else
      target = right;

    if(arr[target] < val) {
      arr[hole] = arr[target];
      hole = target;
    }
    else
      break;
  }
  return hole;
}
```

Note that *percolateDown* is more complicated than *percolateUp* because a node might not have two children

Advantages of Array Implementation

- Minimal amount of wasted space:
  - Only index 0 and any unused space on the right
  - No "holes" due to complete tree property
  - No wasted space representing tree edges
- Fast lookups:
  - Benefit of array lookup speed
  - Multiplying and dividing by 2 is extremely fast (can be done through bit shifting)
  - Last used position is easily found by using the Priority Queue's size for the index

Disadvantages of Array Implementation

- May be too clever:
  - Will you understand it at 3am three months from now?

- What if the array gets too full or too empty?
  - Array will have to be resized
  - Stretchy arrays give us O(1) amortized performance

Advantages outweigh disadvantages: This is how heaps are implemented.

So why O(1) average-case insert?

- Yes, insert's worst case is O(log n)
- The trick is that it all depends on the order the items are inserted
- Experimental studies of randomly ordered inputs shows the following:
- Average 2.607 comparisons per insert
  (# of percolation passes)
  - An element usually moves up 1.607 levels
  - deleteMin is average O(log n)
- Moving a leaf to the root usually requires re-percolating that value back to the bottom

Algorithm Analysis

Data structures

- [Often highly non-obvious] ways to organize information to enable efficient computation over that information
- A data structure supports certain operations, each with a:
  - Meaning: what does the operation do/return
  - Performance: how efficient is the operation
- Examples:
  - List with operations insert and delete
  - Stack with operations push and pop

Trade-offs

- A data structure strives to provide many useful, efficient operations
- But there are unavoidable trade-offs:
  - Time performance vs. space usage
  - Getting one operation to be more efficient makes others less efficient
  - Generality vs. simplicity vs. performance
- That is why there are many data structures and educated CS engineers internalize their main trade-offs and techniques
  - And recognize logarithmic < linear < quadratic < exponential