# Priority Queues

CSE 332 21AU
Lecture 3

# Formally Big-O

We wanted to find an upper bound on our algorithm's running time, but

- We don't want to care about constant factors.
- We only care about what happens as $n$ gets large.

The formal, mathematical definition is Big-O.

**Big-O**

$f(n)$ is $O(g(n))$ if there exist positive constants $c, n_0$ such that for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

We also say that $g(n)$ "dominates" $f(n)$.

# Writing Proofs

Claim: For every odd integer $y$, there exists an even integer $x$, such that $x > y$.

Proof:

Let $y$ be an arbitrary odd integer. By definition, $y = 2z + 1$ for some integer $z$.

Consider $\mathrm{x} = 2(z + 1)$.

$x$ is even (since it can be written as $2$ times some integer) and

$x = 2(z + 1) = 2z + 2 > 2z + 1 = y$. So $x$ meets both of the required properties. □

# Writing Proofs

Where did that $x = 2(z + 1)$ come from?

You probably came up with that even integer first, before you started writing the proof.

That was some "scratch work" – the insight isn't explained in the final proof

- You just say "Consider"

Don't try to skip the scratch work when drafting your big-O proofs.

- But it won't appear in your final version.

# Using the Definition

Let's show: $10n^2 + 15n$ is $O(n^2)$

# Using the Definition

Let's show: $10n^2 + 15n$ is $O(n^2)$

Recreation of whiteboard:

Scratch work:
$10n^2 \leq 10n^2$
$15n \leq 15n^2$ for $n \geq 1$
$10n^2 + 15n \leq 25n^2$ for $n \geq 1$

Proof:
Take $c = 25$ and $n_0 = 1$.
The inequality $10n^2 \leq 10n^2$ is always true. The inequality $15n \leq 15n^2$ is true for $n \geq 1$, as the right hand side is a factor of $n$ more than the right hand side.
As long as both inequalities are true we can add them, thus
$10n^2 + 15n \leq 25n^2$ holds as long as $n \geq 1$.
This is exactly the inequality we needed to show.

# A new ADT: Priority Queues

# A New ADT

Our previous worklists (stacks, queues, etc.) all choose the next element based on the order they were inserted.

That's not always a good idea.

Emergency rooms aren't first-come-first-served.

Sometimes our objects come with a **priority**, that tells us what we need to do next.

An ADT that can handle a line with priorities is a **priority queue.**

# Priority Queue ADT

**Min Priority Queue ADT**

**state**

Set of comparable values
- Ordered based on "priority"

**behavior**

**insert(value)** – add a new element to the collection

**removeMin()** – returns the element with the smallest priority, removes it from the collection

**peekMin()** – find, but do not remove the element with the smallest priority

Uses:

- Operating System
- Well-designed printers
- Huffman Codes (in 143)
- Sorting (in Project 2)
- Graph algorithms

# Implementing Priority Queues: Take I

Maybe we already know how to implement a priority queue.
How long would insert and removeMin take with these data structures?

| | Insert | removeMin |
|---|---|---|
| Unsorted Array | | |
| Unsorted Linked List | | |
| Sorted Linked List | | |
| Sorted Circular Array | | |
| Binary Search Tree | | |

For Array implementations, assume that the array is not yet full.
Other than this assumption, do **worst case** analysis.

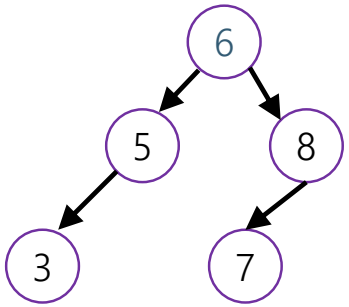
# Review: Binary Search Trees

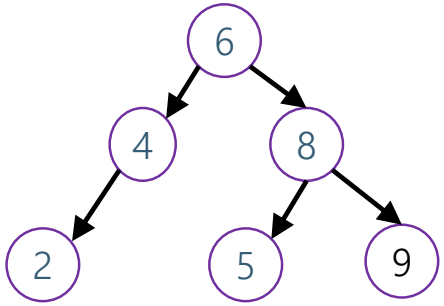
A BST is:

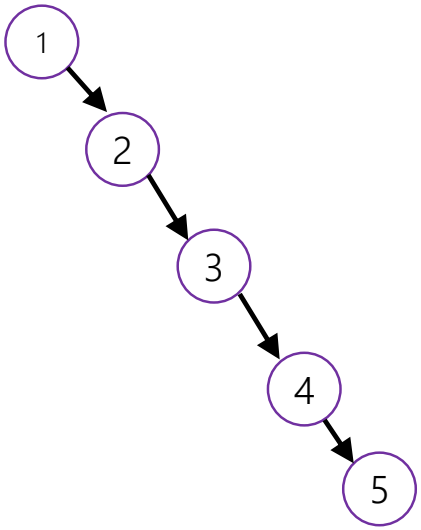
1. A binary tree

2. For each node, everything in its left subtree is smaller than it and everything in its right subtree is larger than it.

# Are These BSTs?



# Implementing Priority Queues: Take I

Maybe we already know how to implement a priority queue.
How long would `insert` and `removeMin` take with these data structures?

| | Insert | removeMin |
|---|---|---|
| Unsorted Array | $\Theta(1)$ | $\Theta(n)$ |
| Unsorted Linked List | $\Theta(1)$ | $\Theta(n)$ |
| Sorted Linked List | $\Theta(n)$ | $\Theta(1)$ |
| Sorted Circular Array | $\Theta(n)$ | $\Theta(1)$ |
| Binary Search Tree | | |

For Array implementations, assume that the array is not yet full.
Other than this assumption, do **worst case** analysis.

# Implementing Priority Queues: Take I

Maybe we already know how to implement a priority queue.
How long would `insert` and `removeMin` take with these data structures?

| | Insert | removeMin |
|---|---|---|
| Unsorted Array | $\Theta(1)$ | $\Theta(n)$ |
| Unsorted Linked List | $\Theta(1)$ | $\Theta(n)$ |
| Sorted Linked List | $\Theta(n)$ | $\Theta(1)$ |
| Sorted Circular Array | $\Theta(n)$ | $\Theta(1)$ |
| Binary Search Tree | $\Theta(n)$ | $\Theta(n)$ |

For Array implementations, assume that the array is not yet full.
Other than this assumption, do **worst case** analysis.

# Implementing Priority Queues: Take II

BSTs have really bad behavior in the **worst** case, but is it actually a common problem?

Fact: On average, the height of a BST is $O(\log n)$
(for some suitable definition of "average")

Can we somehow get that behavior in the **worst case** for priority queues?

# BST Properties

A BST is:

1. A binary tree

2. For each node, everything in its left subtree is smaller than it and everything in its right subtree is larger than it.

Point 2 is what causes the really bad behavior in the worst case.

We probably don't want exactly that requirement for implementing a priority queue.

Maybe we can explicitly enforce that we don't get a degenerate tree.

# Binary Heaps

A Binary Heap is

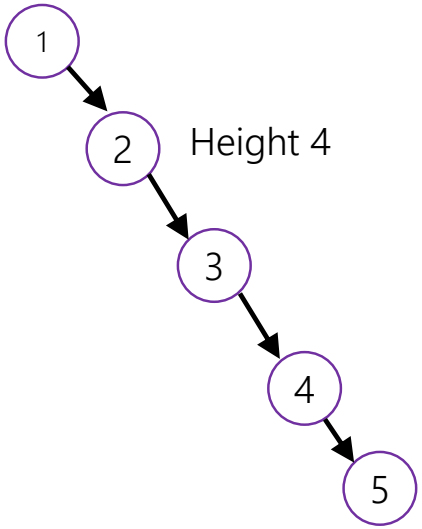
1. A Binary Tree

2. Every node is less than or equal to all of its children
   - In particular, the smallest element must be the root!

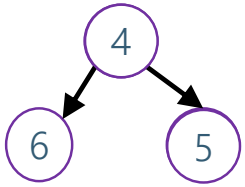
3. The tree is **complete**
   - Every level of the tree is completely filled, except possibly the last row, which is filled from left to right.
   - No degenerate trees!

# Tree Words

Height – the number of edges on the longest path from the root to a leaf.




Height 4

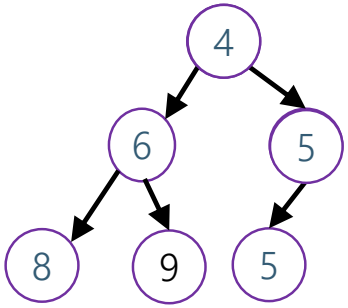



Height 1

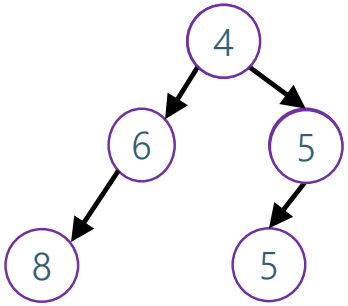
# Tree Words

Complete – every row is completely filled, except possibly the last row, which is filled from left to right.

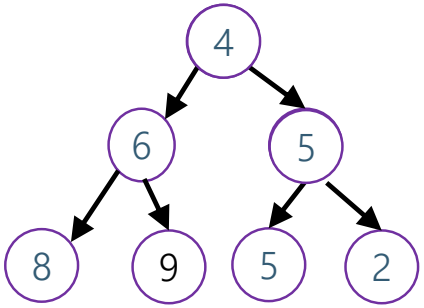
Perfect – every row is completely filled




Complete, but not perfect




Neither




Both Perfect and Complete

# Heights of Perfect Trees

How many nodes are there in level $i$ of a perfect binary tree?

# Heights of Perfect Trees

How many nodes are there in level $i$ of a perfect binary tree?

On the whiteboard we derived that the number of nodes on level $i$ of a binary tree was $2^i$.

Thus the total number of nodes in a perfect binary tree of height $h$ is

$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$.

So if we have $n$ nodes in a perfect tree, we can use the formula

$n = 2^{h+1} - 1$ to conclude that $h = O(\log n)$,so

A perfect tree with $n$ nodes has height $O(\log n)$.

A similar argument can show the same statement for complete trees.

# Binary Heaps

A Binary Heap is

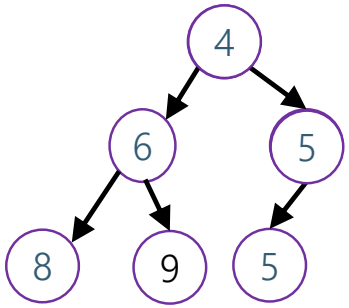
1. A Binary Tree

2. Every node is less than or equal to all of its children
   - In particular, the smallest element must be the root!

3. The tree is **complete**
   - Every level of the tree is completely filled, except possibly the last row, which is filled from left to right.
   - No degenerate trees!

# Implementing Heaps

Let's start with removeMin.



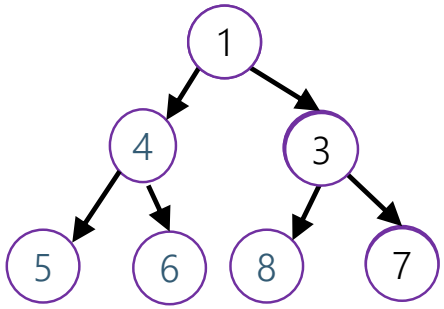

Idea: take the bottom right-most node and use it to plug the hole

Shape is correct now

But that value might be to big. We need to "percolate it down"

# Implementing Heaps

Insertion




What is the shape going to be after the insertion?

Again, plug the hole first.

Might violate the heap property. Percolate it up

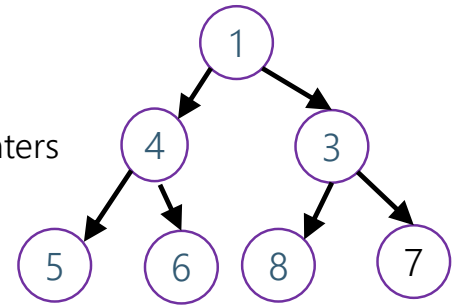
# An Optimization

Pointers are annoying.

They're also slow.

Our tree is so nicely shaped, we don't need pointers

We can use an array instead.




| | 1 | 4 | 3 | 5 | 6 | 8 | 7 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# An Optimization

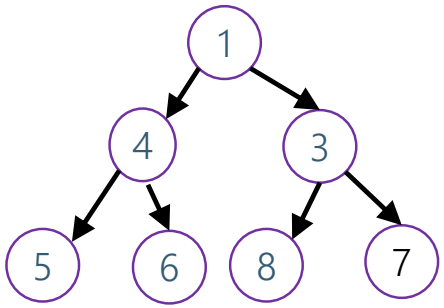
If I'm at index $i$, what is the index of:

My left child, right child and parent?

My left child: $2i$

My right child: $2i + 1$

My parent: $\left\lfloor \frac{i}{2} \right\rfloor$




| | 1 | 4 | 3 | 5 | 6 | 8 | 7 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |