LEC 12

CSE 123

Binary Search Trees

BEFORE WE START

Talk to your neighbors:

Debrief Quiz 3. How do you feel like it went in comparison to Quiz 2?

Music: 123 24su Lecture Tunes

Instructor: Joe Spaniac

TAs: Andras, Daniel, Eric, Nicole, Sahej, Trien, Zach

Questions during Class?

Raise hand or send here

sli.do #cse123

Lecture Outline

- **Announcements**
- Binary Trees
  - Constructor
- Binary Search Trees (BSTs)
  - Definition
  - Why?
  - Runtime

Announcements

- Quiz 3 Completed! 😮💨
  - Congrats! Expect grades back around next Thursday(hopefully)
  - Last quiz of the quarter – all that's left is the final exam (last Friday of the quarter during our typical lecture timeslot)
- Creative Project 3 due tonight @ 11:59pm
  - Submit *something* so we can give you feedback!
- P2 / R4 feedback out after lecture today
- Resubmission Period 5 closes this Friday (8/2) @ 11:59pm
  - Available assignments: **C2**, P2
  - Last opportunity to resubmit C2

Lecture Outline

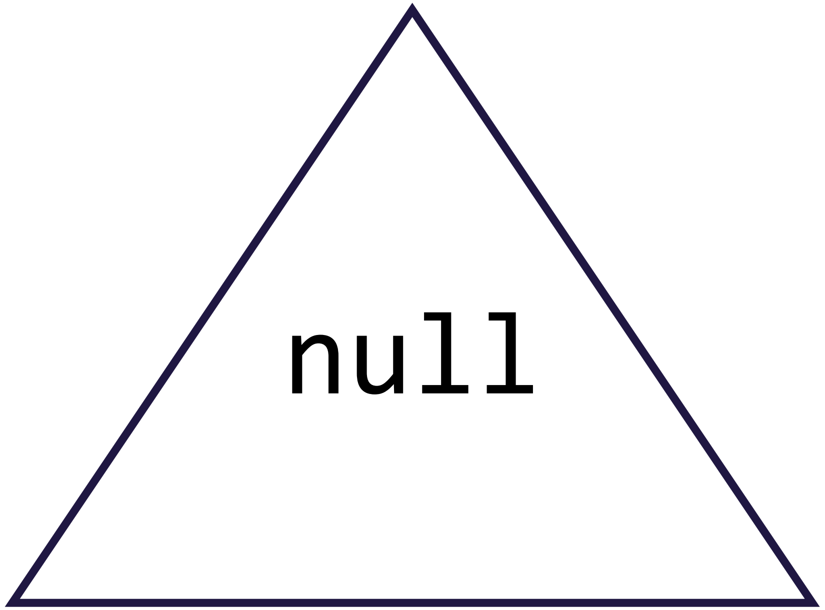
- Announcements
- **Binary Trees**
  - Constructor
- Binary Search Trees (BSTs)
  - Definition
  - Why?
  - Runtime

Lecture Outline

- Announcements
- Binary Trees
  - Constructor
- **Binary Search Trees (BSTs)**
  - Definition
  - Why?
  - Runtime

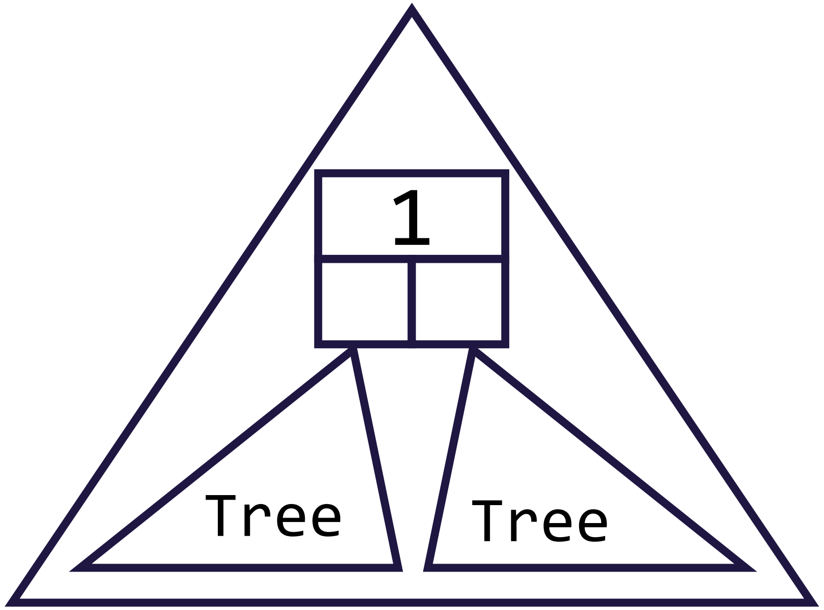
Binary Trees [Review]

- We'll say that any Binary Tree falls into one of the following categories:



Empty tree

`root == null`



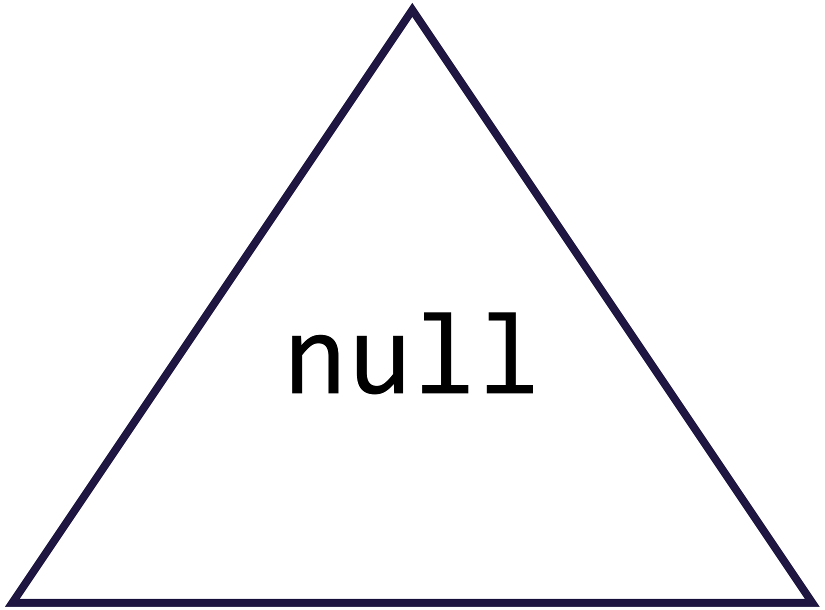
Node w/ two subtrees

`root != null`
`root.left / root.right = Tree`

This is a recursive definition! A tree is either empty or a node with two more trees!

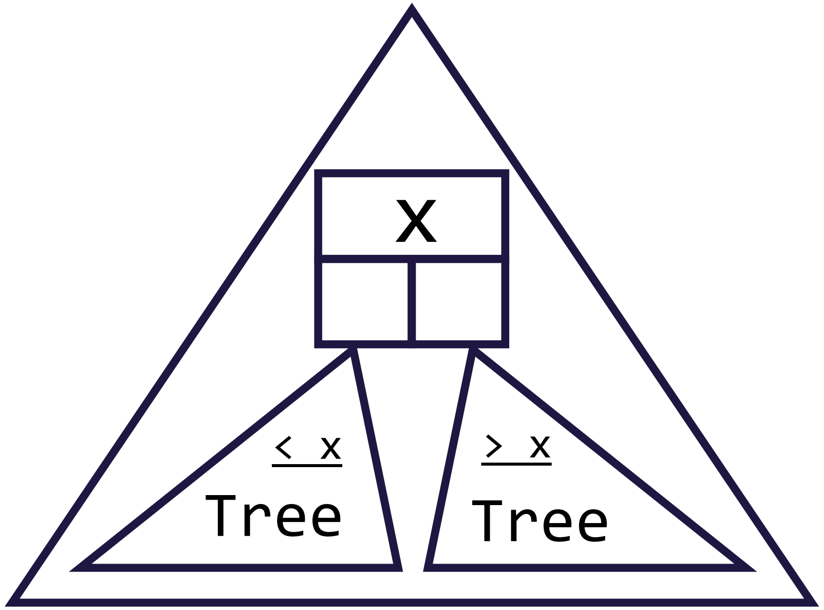
Binary Search Trees (BSTs)

- We'll say that any Binary Search Tree falls into the following categories:



Empty tree

`root == null`



Node w/ two subtrees

```
root != null
root.left / root.right = Tree
max(root.left) < x && min(root.right) > x
```

Note that not all Binary Trees are Binary Search Trees

Why BSTs?

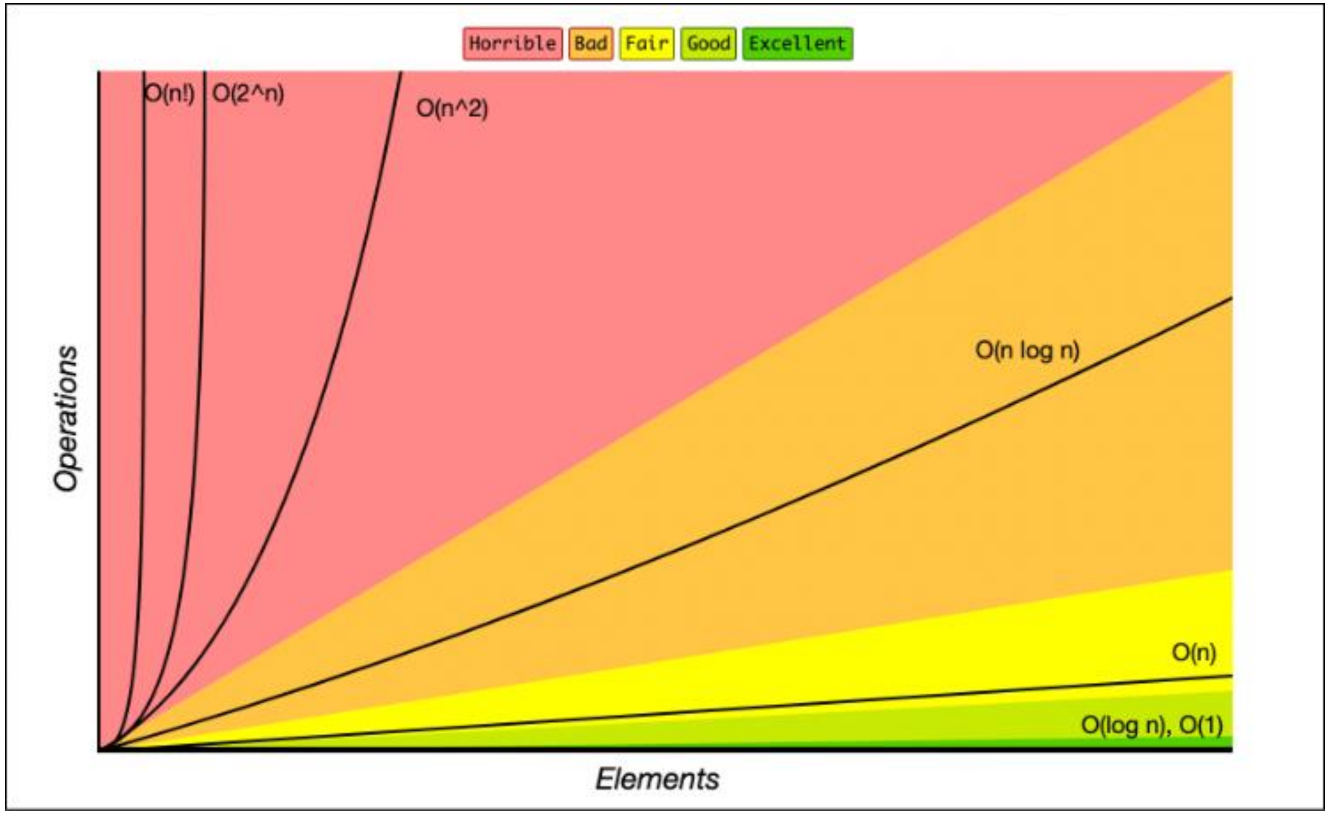
- Our `IntTree` implementation to `contains(int value)`

```
private boolean contains(int value, IntTreeNode root) {
    if (root == null) {
        return false;
    } else {
        return root.data == value ||
               contains(value, root.left) ||
               contains(value, root.right);
    }
}
```

- Which direction(s) do we travel if `root.data != value`?
  - Both left and right
- In a Binary Search Tree, should we check both sides?
  - Remember, additional constraint: `max(root.left) < root.data &&`
    `min(root.right) > root.data`

BSTs & Runtime

- Contains operation on a balanced BST runs in `O(log(n))`
  - Leverages removing half of the values at each step
  - *New runtime class unlocked!*



BSTs & Runtime

- Contains operation on a balanced BST runs in `O(log(N))`
  - Leverages removing half of the values at each step
  - *New runtime class unlocked!*

- Comparison between data structures:

Operation	ArrayIntList	LinkedIntList	IntSearchTree
contains(x)	O(N)	O(N)	O(log(N))

- Let's verify that this is true!

BSTs & Runtime

- Contains operation on a balanced BST runs in `O(log(N))`
  - Leverages removing half of the values at each step
  - *New runtime class unlocked!*

- Comparison between data structures:

Operation	ArrayIntList	LinkedIntList	IntSearchTree
contains(x)	O(N)	O(N)	***O(N)***

- Let's verify that this is true!

O(Log(N)) runtime is only guaranteed for **BALANCED** *BSTs. Since our tree isn't balanced, we see O(N) runtime!*

BSTs In Java

- Self-balancing BST implementations (AVL / Red-black) exist
  - AVL better at contains, Red-black better at adding / removing
- Both the `TreeMap` / `TreeSet` implementations use self-balancing BSTs
  - Determines said ordering via the `Comparable` interface / `compareTo` method
  - Printing out shows natural ordering – preorder traversal
- Complete table comparing data structures:

Operation	ArrayList	LinkedList	TreeSet
contains(x)	O(N)	O(N)	O(log(N))
add(x)	O(1*)	O(1)	O(log(N)*)
remove(x)	O(N)	O(N)	O(log(N)*)

**It's slightly more complicated but we'll leave that for a higher level course*