

# DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 1

**Today's Lecture**

# Last Time

- ▶ Time needed for BST operations is proportional to height.
- ▶ If tree is balanced,  $h = \Theta(\log n)$
- ▶ If tree is unbalanced,  $h = O(n)$

# Today

- ▶ How do we ensure that tree is balanced?
- ▶ Approach 1: Complicated rules, red-black trees.
- ▶ Approach 2: Randomization
- ▶ We'll introduce **treaps**.

# DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 2

## Red-Black Trees

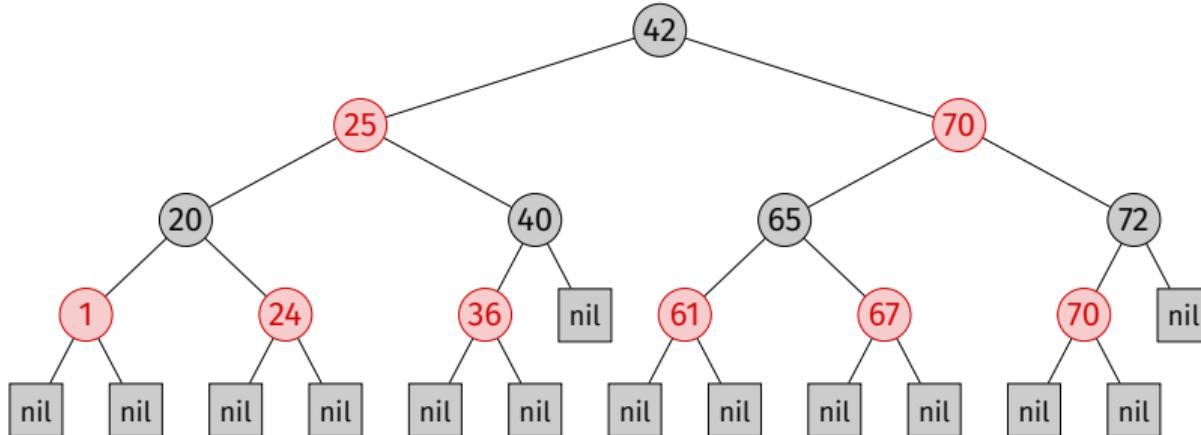
# Self-Balancing BSTs

- ▶ We wish to ensure that the tree does not become unbalanced.
- ▶ Idea: If tree becoming unbalanced, it will balance itself.
- ▶ Several strategies, including **red-black** trees and **AVL** trees

# Red-Black Trees

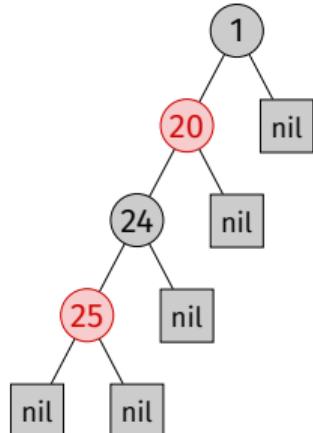
- ▶ A **red-black** tree is a BST whose nodes are colored **red** and **black**.
- ▶ Leaf nodes are “nil”.
- ▶ Must satisfy four additional properties:
  1. The root node is **black**.
  2. Every leaf node is **black**.
  3. If a node is **red**, both child nodes are **black**.
  4. For any node, all paths from the node to a leaf contain the same number of **black** nodes.

# Example



# Example

- ▶ This **not** a red-black tree.
  - ▶ Violates last property



## Claim

If a red-black tree has  $n$  internal (non-nil) nodes, then the height is at most  $2 \log(n + 1)$ .

# Proof Intuition<sup>1</sup>

- ▶ All paths from root to a leaf are about the same length ( $\approx h$ ).
- ▶ Therefore, the tree is close to balanced.
- ▶ So height is proportional to  $\log n$

---

<sup>1</sup>Formal proof proceeds by induction.

# Non-Modifying Operations

- ▶ As a result, the non-modifying operations take  $\Theta(\log n)$  time in red-black trees.
  - ▶ query
  - ▶ minimum/maximum
  - ▶ next smallest/largest
- ▶ Proof: these take  $\Theta(h)$  time in any BST, and in a red-black tree  $h = O(\log n)$ .

# Insertion and Deletion

- ▶ Standard BST `.insert` and `.delete` methods preserve BST, but **not** red-black properties.
- ▶ Insertion/deletion in a red-black tree is considerably more **complicated**.
- ▶ But both take  $\Theta(\log n)$  time.

*Implementing balanced trees is an exacting task and as a result balanced tree algorithms are rarely implemented except as part of a programming assignment in a data structures class<sup>2</sup>.*

Pugh, 1990

---

<sup>2</sup>For computer science majors.

# Summary

- ▶ For red-black trees, worst cases:

query	$\Theta(\log n)$
minimum/maximum	$\Theta(\log n)$
next largest/smallest	$\Theta(\log n)$
insertion	$\Theta(\log n)$

- ▶ But they are **tricky** to implement.

# DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 3

## Randomization to the Rescue

# **Order Matters**

- ▶ The structure of a BST depends on insertion order.

# Example

- ▶ Insert 1,2,3,4,5,6 into BST, in that order.

# Example

- ▶ Insert 3, 5, 1, 2, 4, 6 into BST, in that order.

## Claim

The expected height of a BST built by inserting the keys in random order is  $\Theta(\log n)$ .

# Idea

- ▶ To build a BST, take all  $n$  keys, shuffle them randomly, then insert.
- ▶ No need for Red-Black Trees, right?

# Problem

- ▶ Usually don't have all the keys right now.
- ▶ This is a **dynamic set**, after all.
- ▶ The keys come to us in a stream, can't specify order.

# Goal

- ▶ Design a data structure that **simulates** random insertion order without actually changing the order.

# DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 4

## Treaps

# Randomization

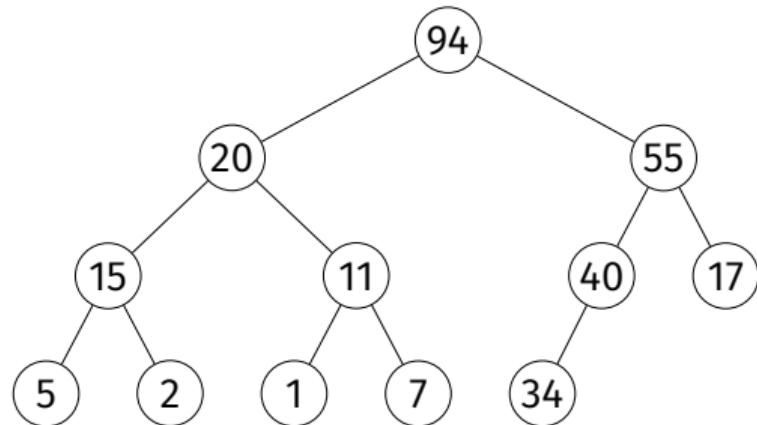
- ▶ If insertions are in a random order, expected depth of a BST is  $\Theta(\log n)$ .
- ▶ But in **online** operation, we cannot randomize insertion order.
- ▶ Now: an elegant data structure simulating random insertion order in online operation.

# First: Recall Heaps

- ▶ A **max heap** is a **binary tree** where:
  - ▶ each node has a priority.
  - ▶ if  $y$  is a child of node  $x$ , then
$$y.\text{priority} \leq x.\text{priority}$$

# Example

- ▶ This is a max heap:

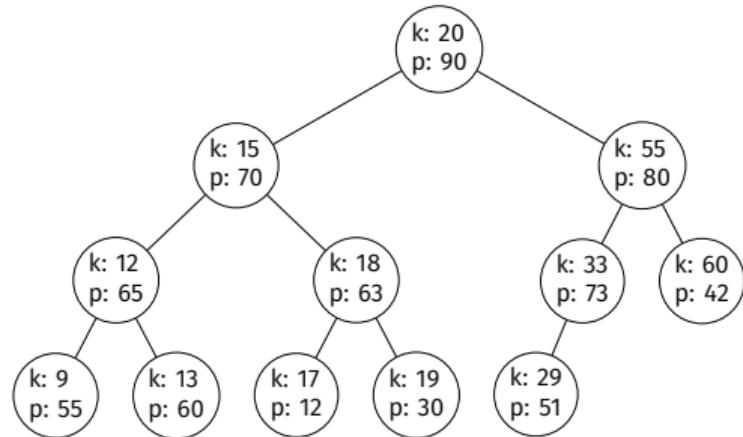


# Treaps

- ▶ A **treap** is a binary tree in which each node has both a **key** and a **priority**.
- ▶ It is a **max heap** w.r.t. its priorities.
- ▶ It is a **binary search tree** w.r.t. its keys.

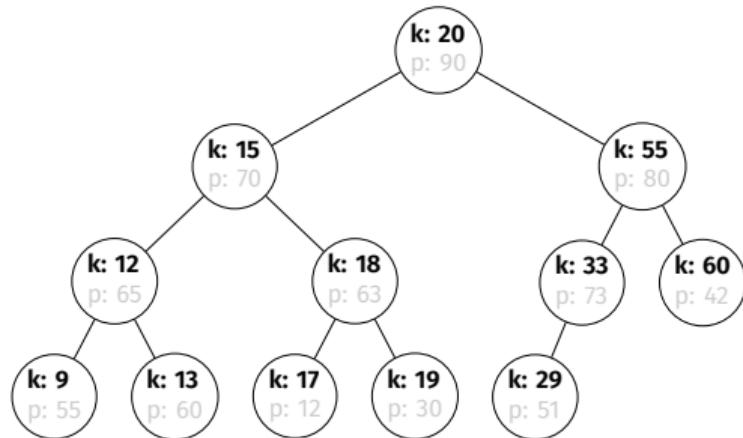
# Example

- ▶ This is a treap:



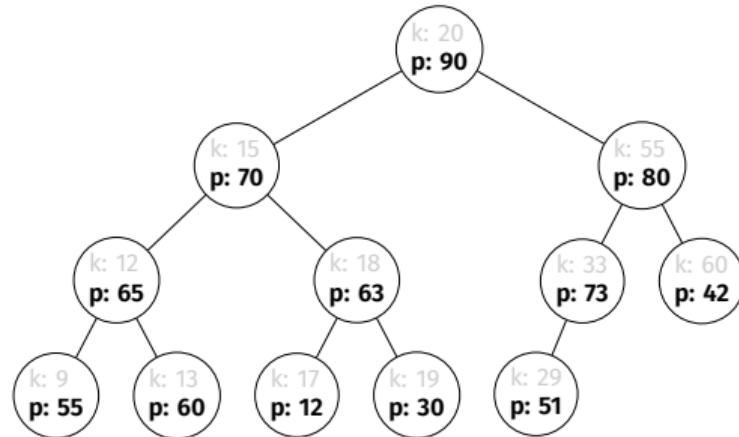
# Example

- ▶ This is a treap:



# Example

- ▶ This is a treap:



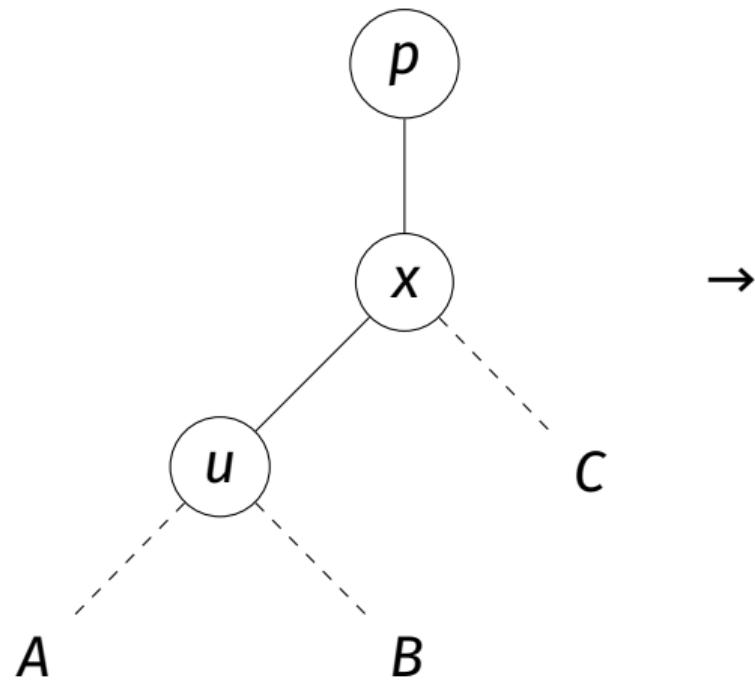
# BST Operations

- ▶ Because a treap is a BST, querying, finding max/min by key is done the same.
- ▶ Insertion and deletion require care to preserve **heap** property.

# Insertion

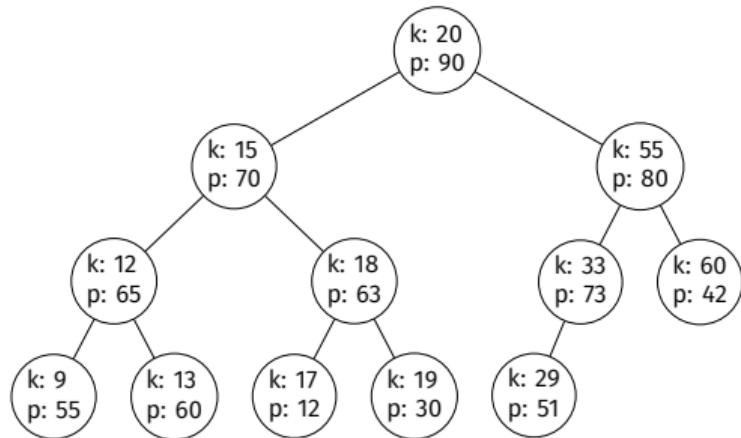
- ▶ Find place to insert node as usual.
- ▶ While priority of new node is  $>$  than parent's:
  - ▶ Left rotate new node if it is the right child.
  - ▶ Right rotate new node if it is the left child.
- ▶ Rotate preserves BST, repeat until heap property satisfied.

# (Right) Rotation



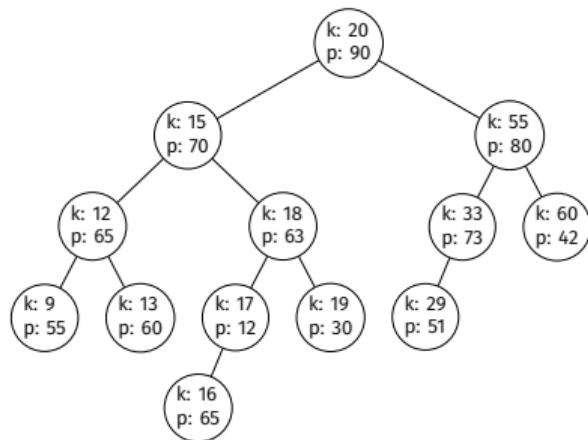
# Example: Insertion

- ▶ Insert key: 16, priority: 65.



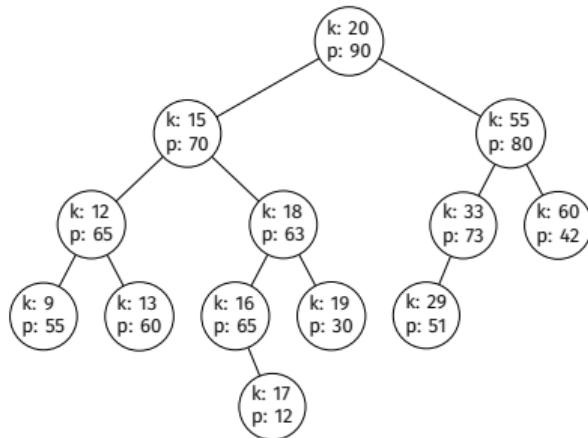
# Example: Insertion

- ▶ Insert key: 16, priority: 65.



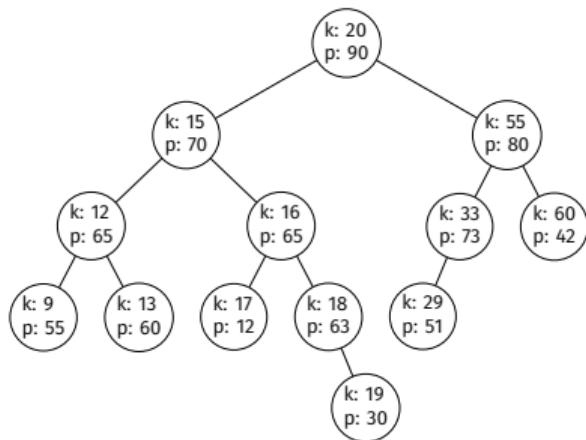
# Example: Insertion

- ▶ Insert key: 16, priority: 65.



# Example: Insertion

- ▶ Insert key: 16, priority: 65.



# **Deletion**

- ▶ While node is not a leaf:
  - ▶ Rotate it with child of highest priority.
- ▶ Once it is a leaf, delete it.

**DSC 190**

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 5

**Treap Properties**

# **Good Question**

- ▶ Is it always possible to build a treap?

## Claim

Given any set of (key, priority) pairs, inserting them one-by-one into a treap always results in a valid treap (no matter the insertion order).

# Proof Idea

- ▶ Start with a treap (possibly empty).
- ▶ Inserting new (key, priority) preserves treap:
  - ▶ **BST**: rotation preserves BST property
  - ▶ **heap**: initially violated, but rotation repeated until it is satisfied

## Claim

Given any set of (key, priority) pairs, if both keys and priorities are unique, then the treap is **unique**.

## Claim

**Corollary:** Given any set of (key, priority) pairs, if both keys and priorities are unique, inserting them one-by-one into a treap results in the same treap, no matter the insertion order.

# Example

- ▶ Insert  $(3, 40)$ ,  $(1, 20)$ ,  $(10, 50)$ ,  $(6, 30)$ ,  $(5, 100)$ , in that order

# Example

- ▶ Insert  $(5, 100)$ ,  $(10, 50)$ ,  $(3, 40)$ ,  $(6, 30)$ ,  $(1, 20)$ , in that order

# Proof Idea

- ▶ Root node must be node w/ highest priority.
- ▶ Root's left (right) child must have highest priority among nodes with key  $<$  ( $>$ ) root key.
- ▶ Apply recursively.

## Claim

Given any set of (key, priority) pairs, if both keys and priorities are unique, then inserting them one-by-one into a treap (in any order) results in the **same** BST one would obtain by inserting into a BST in decreasing order of priority.

**DSC 190**

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 6

**Randomized Binary Search Trees**

## Claim

Given any set of keys, if they are inserted into a BST in random order, the result is (almost surely) balanced. The expected height is  $\Theta(\log n)$ .

## Claim

Given any set of (key, priority) pairs, if both keys and priorities are unique, then inserting them one-by-one into a treap (in any order) results in the **same** BST one would obtain by inserting into a BST in decreasing order of priority.

# The Idea

- ▶ When inserting a node into a treap, generate priority **randomly**.
- ▶ The resulting treap will be the same tree as a BST built with nodes randomly ordered according to these priorities.
- ▶ It will almost surely be balanced.
- ▶ This is called a **randomized binary search tree**<sup>3</sup>.

---

<sup>3</sup>Sometimes people call these treaps

# Warning

- ▶ Randomness does not mean that the result of, for example, a query has some probability of being incorrect.
- ▶ BST operations on treaps are always, 100% correct.
- ▶ The structure is random.

# Example

- ▶ Insert 1, 2, 3, 4, 5, 6 into a treap, generating priorities randomly.

# Time Complexities

- ▶ For randomized BSTs, expected times:

query	$\Theta(\log n)$
minimum/maximum	$\Theta(\log n)$
next largest/smallest	$\Theta(\log n)$
insertion	$\Theta(\log n)$
- ▶ Worst case times are  $\Theta(n)$ , but very rare

# Comparison to Red-Black Trees

- ▶ When compared to red-black trees, randomized BSTs are:
  - ▶ same in terms of expected time;
  - ▶ perhaps slightly slower in practice;
  - ▶ **much** easier to implement/modify.
- ▶ Good trade-off for a data scientist!

# Priority Hacks

- ▶ Several interesting strategies for generating a new node's priority, beyond simply generating a random number.

# Idea #1: Hashing

- ▶ Instead of randomly generating a number, hash the key to get priority.
- ▶ Works, provided hash function looks random.
- ▶ **Careful!** In python, `hash(300) == 300`

## Idea #2: “Learning”

- ▶ Idea: Frequently-queried items should be near top of tree.
- ▶ When an item is queried, update its priority:  
 $\text{new priority} = \max(\text{old priority}, \text{random number})$

# DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 7

## Order Statistic Trees

# Modifying BSTs

- ▶ More than most other data structures, BSTs must be modified to solve unique problems.
- ▶ Red-black trees are a pain to modify.
- ▶ Treaps/randomized BSTs are easy!

# Order Statistics

- ▶ Given  $n$  numbers, the  **$k$ th order statistic** is the  $k$ th smallest number in the collection.

# Example

[99, 42, -77, -12, 101]

- ▶ 1st order statistic:
- ▶ 2nd order statistic:
- ▶ 4th order statistic:

## Exercise

Some special cases of order statistics go by different names. Can you think of some?

# Special Cases

- ▶ **Minimum:** 1st order statistic.
- ▶ **Maximum:**  $n$ th order statistic.
- ▶ **Median:**  $[n/2]$ th order statistic<sup>4</sup>.
- ▶  **$p$ th Percentile:**  $\lceil \frac{p}{100} \cdot n \rceil$ th order statistic.

---

<sup>4</sup>What if  $n$  is even?

# Computing Order Statistics

- ▶ Quickselect finds any order statistic in linear expected time.
- ▶ This is efficient for a static set.
- ▶ Inefficient if set is dynamic.

# Goal

- ▶ Create a dynamic set data structure that supports fast computation of **any** order statistic.

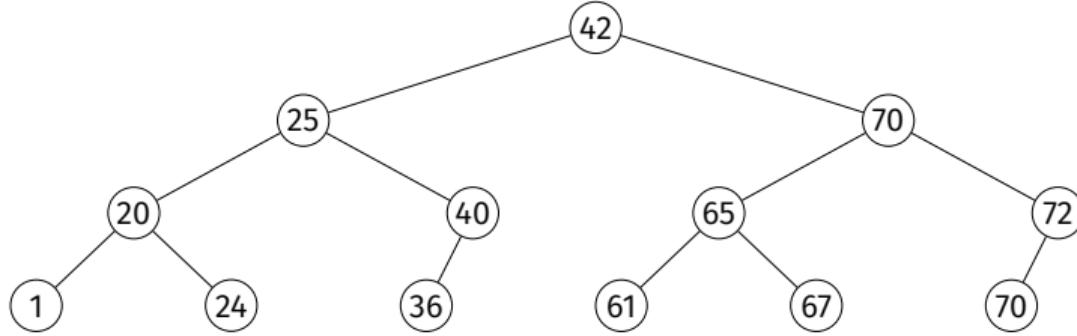
## Exercise

Does the “two heaps” trick from before work?

# BST Solution

- ▶ For each node, keep attribute `.size`, containing # of nodes in subtree rooted at current node

# Example: Insert/Delete



# Challenge

- ▶ `.number_lt` changes when nodes are inserted/deleted
- ▶ We must **modify** the code for insertion/deletion
- ▶ A pain with R-B tree; easy with treap!

**DSC 190**

DATA STRUCTURES & ALGORITHMS

Lecture 5 | Part 8

**BSTs vs. Heaps**

# **BSTs vs. Heaps**

- ▶ Seemingly similar.
- ▶ Both are binary trees.
- ▶ Similar time complexities.

# Summary

	Balanced BST	Binary Heap
get minimum/maximum	$\Theta(\log n)$ <sup>5</sup>	$\Theta(1)$
extract minimum/maximum	$\Theta(\log n)$	$\Theta(\log n)$
insertion	$\Theta(\log n)$	$\Theta(\log(n))$

---

<sup>5</sup>Can actually be optimized to  $\Theta(1)$

# Comparison

## BSTs

- ▶ No cache locality
- ▶ Memory for pointers
- ▶ Maintains sorted order
- ▶ **Used for order statistics, queries**

## Heaps

- ▶ Cache locality
- ▶ Use less memory
- ▶ Costly to query
- ▶ **Used for max/min**