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 automatically trigger a rebalance.
- Several strategies, including **red-black** trees and **AVL** trees

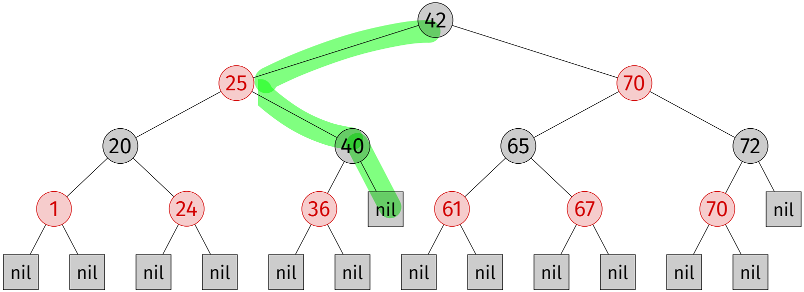
Red-Black Trees

- 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.

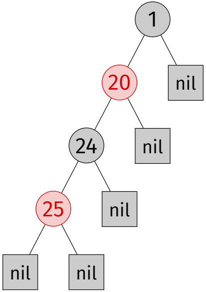
Example



42
25
70
20
40
65
72
1
24
36
nil
61
67
70
nil
nil
nil
nil
nil
nil
nil
nil
nil
nil
nil
nil
nil

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[1]

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

[1] 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 $h = \Theta(\log n)$.

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.[2]

Pugh, 1990

[2] 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.

Summary

- As a data scientist, you should know that self-balancing BSTs exist, guaranteeing $\Theta(\log n)$ worst-case time for all operations.
- But you should **not** implement them yourself.

DSC 190

Data Structures & Algorithms

Lecture 5 | Part 3

Randomization to the Rescue

Implementing BSTs

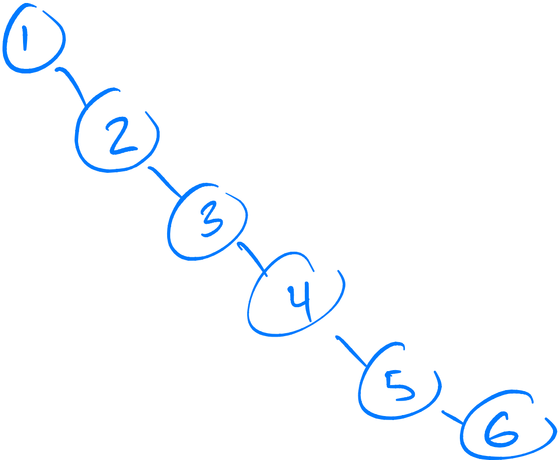
- Red-black trees are **complicated** to implement.
  - Use someone else's implementation.
- But sometimes an off-the-shelf implementation doesn't solve your problem.
  - Example: BSTs for order statistics.
- How do we implement a self-balancing BST **simply** and **efficiently**?

Order Matters

- The structure of a BST depends on insertion order.

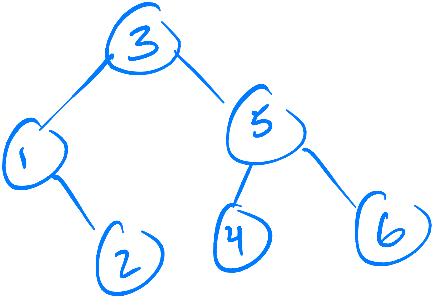
Example

- 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

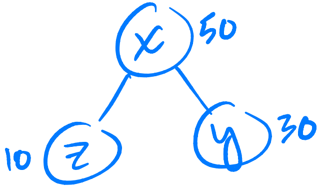
Randomization

- 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

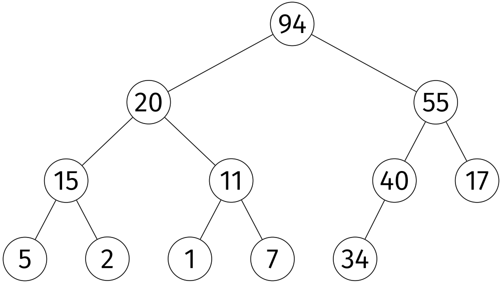
- A **max heap** is a **binary tree** where:
  - each node has a priority.
  - if *y* is a child of node *x*, then

$$\texttt{y.priority} \leq \texttt{x.priority}$$



Example

- This is a max heap:

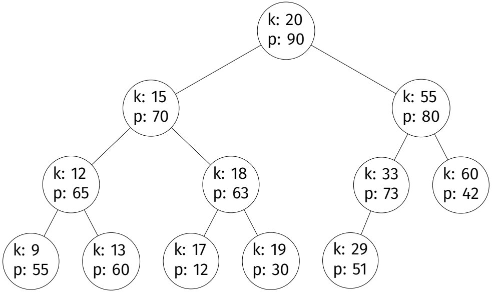


Treaps

- 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.

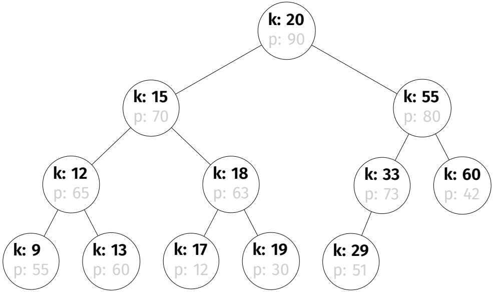
Example

- This is a treap:



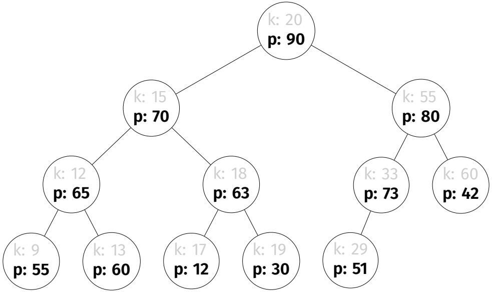
Example

- 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.

(k, p)

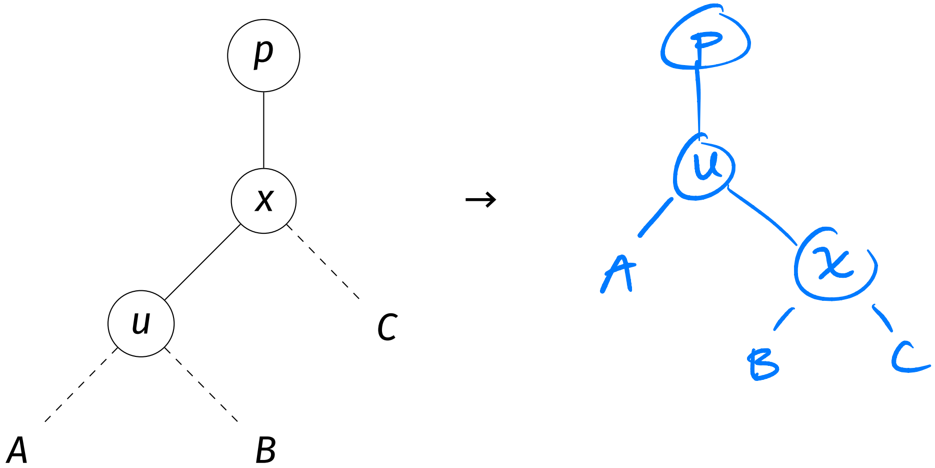
Insertion

1. 1) Using the key, find place to insert node as if in a BST.

2. 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.

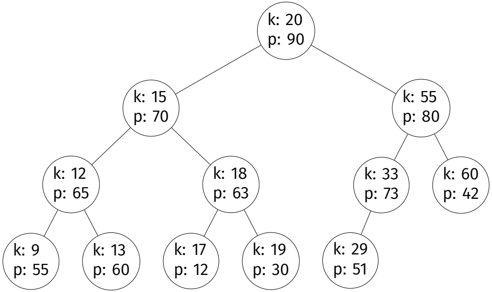
- Rotate preserves BST, repeat until heap property satisfied.

(Right) Rotation



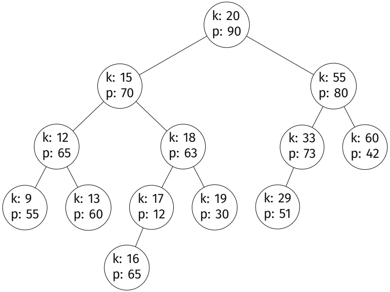
Example: Insertion

- Insert key: 16, priority: 65.



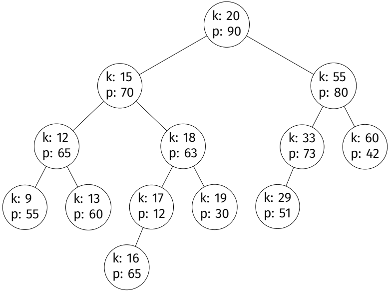
Example: Insertion

- Insert key: 16, priority: 65.



Example: Insertion

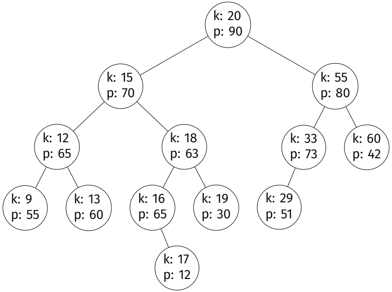
- Insert key: 16, priority: 65.



Observe: This *is* a BST, not a heap. Rotate to fix.

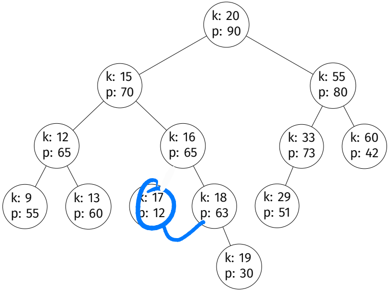
Example: Insertion

- Right rotate 16.



Example: Insertion

- Right rotate 16 again.



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

Treaps and Order

BSTs and Order

- There are many possible BSTs representing the same set of keys.
- The **order** in which keys are inserted has a large effect on the structure of the resulting BST.
- What about for treaps?

Claim

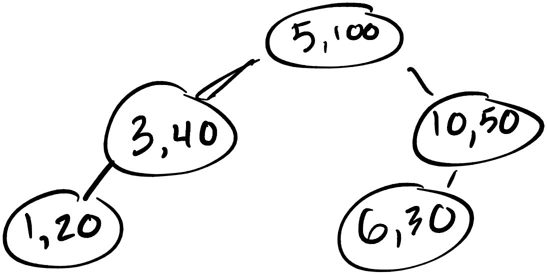
Given any set of (key, priority) pairs, if all keys and priorities are unique, then the treap is **unique**.

Claim

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

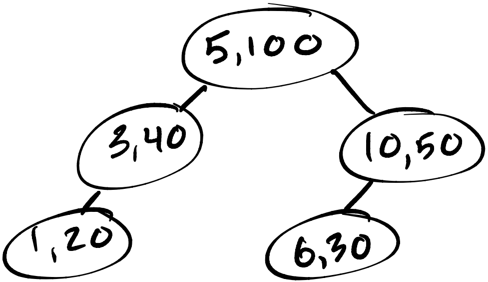
Example

- 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 Sketch

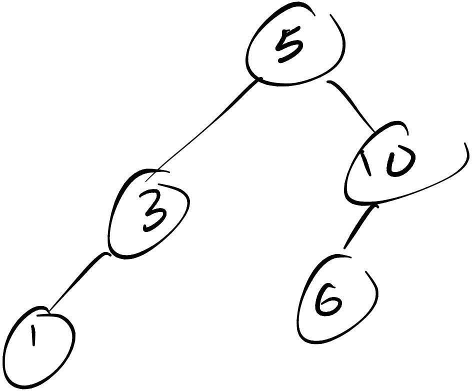
- Node w/ highest priority must be the root.
- Root's left (right) child must have highest priority among nodes with key < (>) root key.
- Apply recursively.

Which BST?

- Given a set of unique (key, priority) pairs, there are many BSTs for the keys.
  - Each corresponding to a different insertion order.
- Only one of these BSTs is **also** a heap for the priorities.
- What insertion order corresponds to this BST?

Example

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



Claim

The BST obtained by building a treap is the same BST you'd get by inserting nodes in decreasing order of priority.

Main Idea

The structure of the treap is determined not by insertion order, but by the **priorities**.

DSC 190

Data Structures & Algorithms

Lecture 5 | Part 6

Randomized Binary Search Trees

Recall

- We saw before that inserting keys in random order results in a balanced tree, on average.
- But we often can’t control the order in which we see keys.
- Also saw that order doesn’t matter for treaps; priorities do.

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.

Randomized Binary Search Tree

- This is called a **randomized binary search tree**[3].
- Introduced by Cecilia Rodriguez Aragon, Raimund Seidel in 1989; later, Conrado Martínez and Salvador Roura in 1997.

[3] Sometimes people call these treaps

Main Idea

By generating priorities randomly, we “simulate” inserting keys in random order, without actually having to see the keys in random order.

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.
- Instead, the **tree's structure** is random.

Example

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

1, 50.7
2, 20.3
3, 71.2

Time Complexities

- For randomized BSTs, **expected** times:

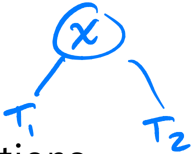
query	$\Theta(\log n)$
minimum/maximum	$\Theta(\log n)$
next largest/smallest	$\Theta(\log n)$
insertion	$\Theta(\log n)$

- 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!

Bulk Operations



- Treaps also allow for very fast set operations.

- **Example:** Given a treap T and a "splitting value" x, split into two treaps T_1 and T_2 such that:
  - T_1 contains all keys $< x$;
  - T_2 contains all keys $\geq x$.

- **Idea:** Insert x into T with a very high priority.

- The time needed is only $\Theta(\log n)$, not $\Theta(n)$!

Priority Hacks

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

Idea: “Learning” Treaps

- Idea: Frequently-queried items should be near top of tree.

- When an item is queried, update its priority:

  new priority = *max*(old priority, random number)

Demo

- A demo notebooks is posted on the course website.

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 **kth order statistic** is the kth smallest number in the collection.

Example

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

- 1st order statistic: -77
- 2nd order statistic: -12
- 4th order statistic: 99

Exercise

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

Special Cases

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

---

[4] 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.

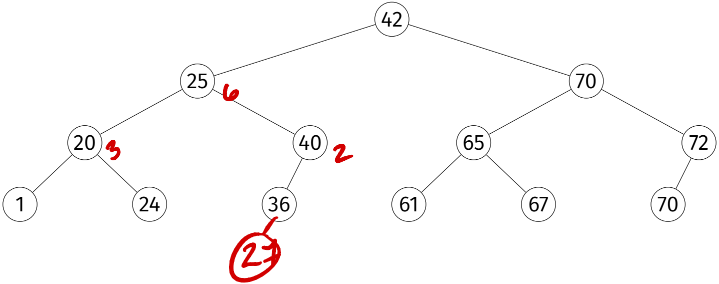
Exercise

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	$\Theta(\log n)$[5]	$\Theta(1)$
extract minimum/maximum	$\Theta(\log n)$	$\Theta(\log n)$
insertion	$\Theta(\log n)$	$\Theta(\log(n))$

[5] 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**