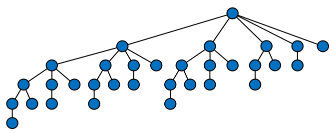
Lecture 9. Splay Trees, Sweep Line Algorithms

CpSc 8400: Algorithms and Data Structures

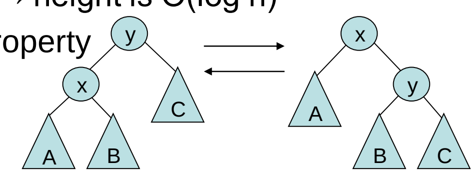
Brian C. Dean



School of Computing
Clemson University
Spring, 2016

Recall: "Worst-Case" Balancing Mechanisms (Height Always O(log n))

- AVL trees
  - Augment nodes with subtree heights
  - Height balanced property → height is O(log n)
  - Maintain height balance after insert or delete with a few carefully-chosen rotations.
- Red-black trees
  - Augment nodes with a color: red or black
  - "Red black" property→ height is O(log n)
  - Maintain red-black property after insert or delete with rotations and re-colorings



Today: Splay Trees

- Surprisingly simple way to achieve the same performance as a balanced BST, even though strict balance may not always hold.
- Any sequence of k splay tree operations, starting with an empty tree, takes O(k log n) time, so each operation takes O(log n) <u>amortized</u> time.

3

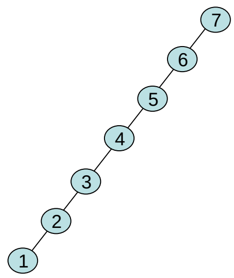
Rotating Elements to the Root

- What if every time an element is accessed, we rotate it one step closer to the root.
- In general, this seems like it should move frequently-accessed elements closer to the root, making subsequent accesses to these elements faster.
- Is there any situation where this wouldn't help much?

4

Rotating Elements to the Root

- What if every time an element is accessed, we rotate it one step closer to the root.



- Bad example: access 1, 2, 1, 2, 1, 2, 1, 2, …

5

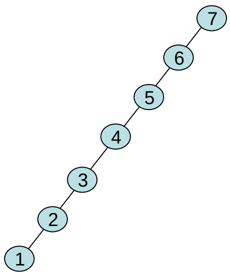
Rotating Elements to the Root

- New idea: what if every time an element is accessed, we rotate it all the way to the root.

6

Rotating Elements to the Root

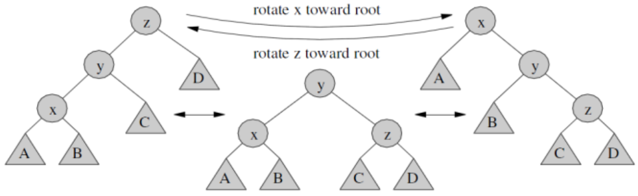
- New idea: what if every time an element is accessed, we rotate it all the way to the root.
- Unfortunately, there are still situations where this doesn't quite give us the performance we want.



- Example, what if we access 1, 2, 3, 4, 5, 6, 7, …?

Rotate to Root : An Improvement?

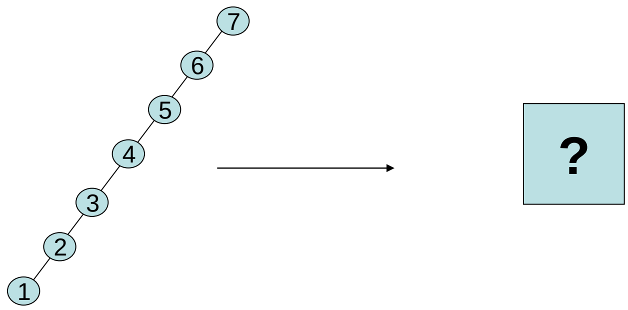
- **Idea #3:** When an element is accessed, "splay" it to the root as follows:
  - If element is one step below root, rotate up to the root.
  - If element is "in-line" with parent and grandparent, rotate parent first, then element.



  - Otherwise, rotate element up two steps as before, using single rotations

Splaying : Example

- Consider the path example that was bad for single rotations:

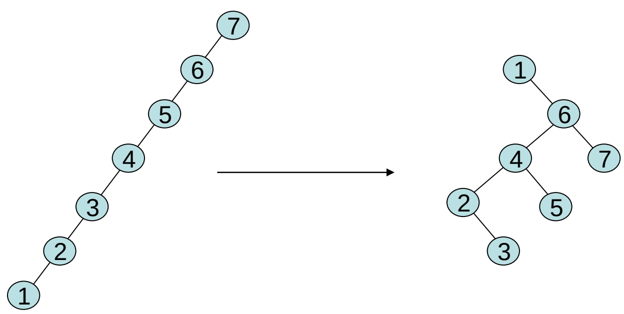


- What happens when we access element 1, and then splay it to the root?

9

Double Rotations : Example

- Consider the path example that was bad for single rotations:



- We essentially halve the length of the path, thereby making the tree more balanced!

10

Splay Trees

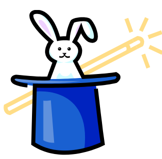
- **Splay(e)** : Move e to root using double rotations.
- A splay tree is a BST in which we splay an element every time it is accessed:
  - *find*(e) : Find as e usual, then *splay*(e).
  - *insert*(e) : Insert as e usual, then *splay*(e).
  - *delete*(e) : Discuss in a moment…
- A splay tree is called a **self-adjusting** tree, since it continually modifies its structure according to simple local update rules *that do not depend on any augmented information stored within the tree*.

11

Splay Trees : Performance

- **Remarkable property:** all operations on a splay tree run in O(log n) amortized time!

  (i.e., any sequence of k operations, starting with an empty tree, takes O(k log n) time).

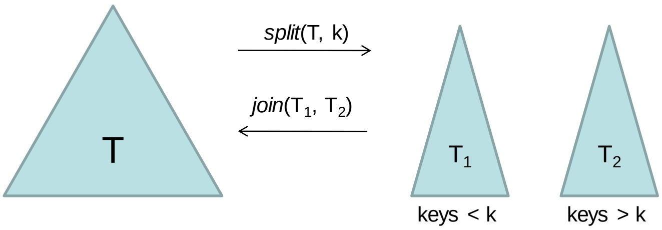


- So a splay tree magically stays balanced (in an amortized sense), even though it maintains no augmented information to help it do so!

12

Split and Join

- Splay trees easily support the extended BST operations **split** and **join**:
  - *split*(T, k) : Split the BST T into two BSTs, one containing key ≤ k and the other keys > k.
  - *join*(T_1, T_2) : Take two BSTs T_1 and T_2, the keys in T_1 all being less than the keys in T_2, and join them into a single BST.

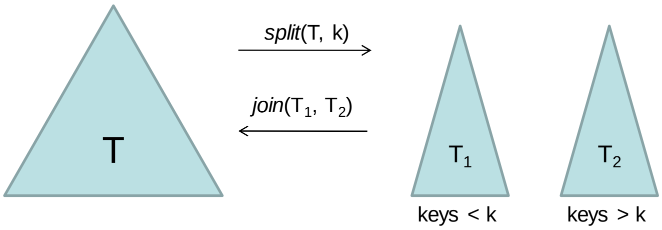


13

Split and Join on a Splay Tree

- On a splay tree:
  - *split*(T, k) : Find the element e of key k. Then splay e to root and remove its right subtree.

    (if k doesn't exist, use pred(k) instead)
  - *join*(T_1, T_2) : Splay the maximum element in T_1 to the root. Then attach T_2 as its right subtree.



14

Insert and Delete Using Split and Join

- If we can split and join easily, then we can also insert and delete easily:
  - *insert*(T, e) *:* split T on e's key into T_1 and T_2. Then make e the root, with left subtree T_1 and right subtree T_2.
  - *delete*(e) : replace e with the *join* of its left and right subtrees.
- On a splay tree, this is how we implement delete (insert is usually done by inserting as in a normal BST then splaying to root).

15

Split, Join, and Dynamic Sequences

- Remember how a BST can encode a dynamic sequence so that *insert*, *delete*, and rank-based access all take O(log n) time.
- Now suppose we want the ability to cut a sequence into two shorter sequences, and to link two sequences together end-to-end.
- The split and join BST operations do precisely this, so on a splay tree, we can easily perform these operations in O(log n) amortized time.
- Application: cut/paste.

16

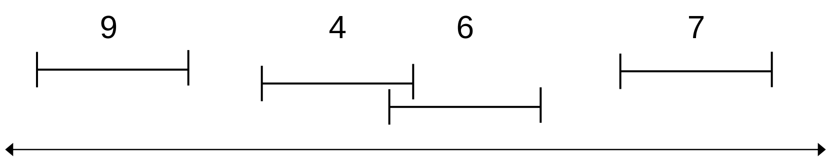
Sorting as a Preprocessing Step

- Many problems get easier after sorting the input first as a preprocessing step.
- Today's lecture: "Sort and scan" or "sweep line" algorithms.

- These are particularly common in computational geometry. And they are a good way to exercise your data structure skills…

Warm-Up

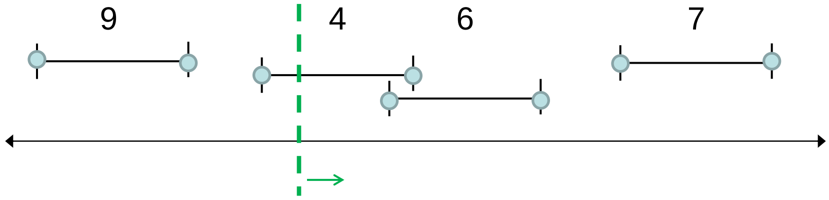
- You are told N intervals on the number line, each with an associated value.



- Find a point of maximum overlap (i.e., maximizing the sum of interval values overlapping at that point).

A “Sweep Line” Approach

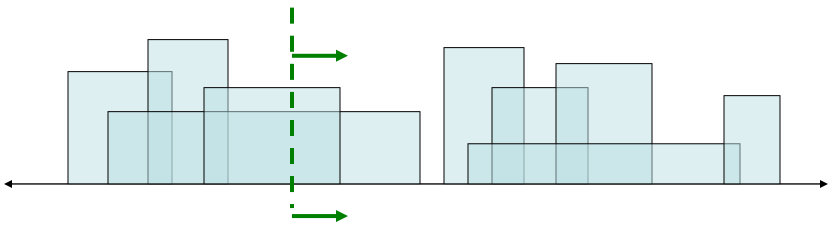
- You are told N intervals on the number line, each with an associated value.



- Find a point of maximum overlap (i.e., maximizing the sum of interval values overlapping at that point).

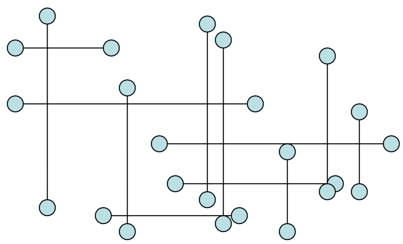
Example: The Skyline Problem

- Give N rectangular buildings sharing a common base, find the area of the “skyline” they form.

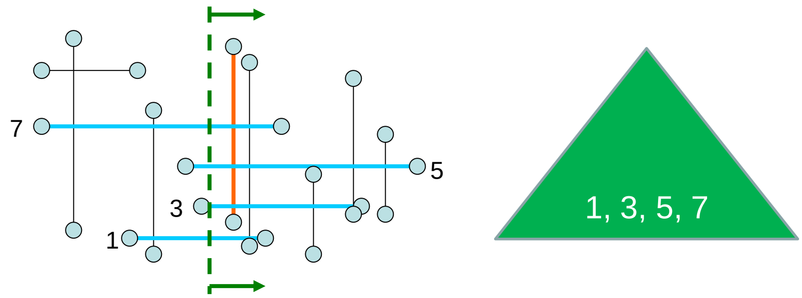


- Sweep line + binary heap (or balanced BST).
- N inserts, N deletes, 2N find-max: O(N log N)

Counting Intersection Points Among Axially-Aligned Segments

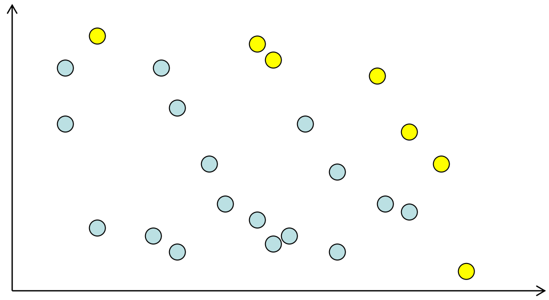


Counting Intersection Points Among Axially-Aligned Segments

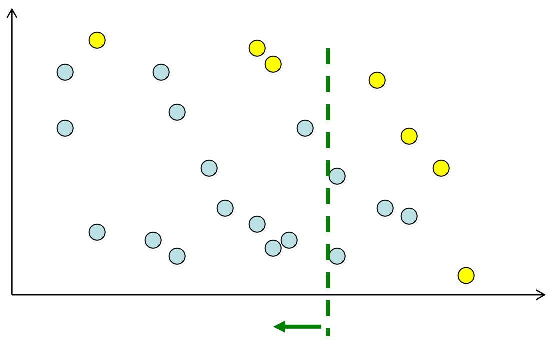


- Sweep line on x + balanced BST of “active” horizontal segments, keyed on y.
- ≤ 2N inserts, ≤ 2N deletes, ≤ N range counting queries: O(N log N)

Non-Dominated Points

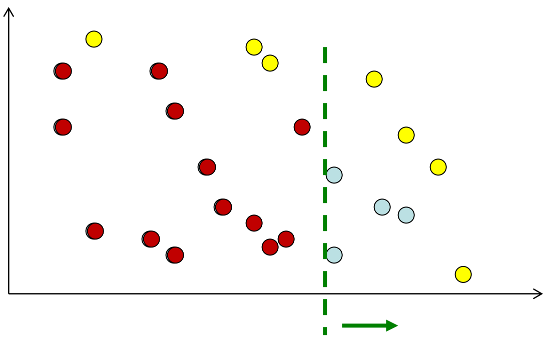


Non-Dominated Points



- Reverse sweep line on x + running max on y.
- O(N log N)

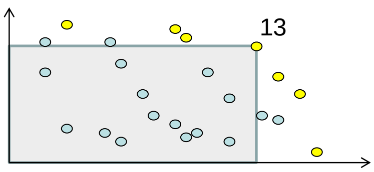
Non-Dominated Points



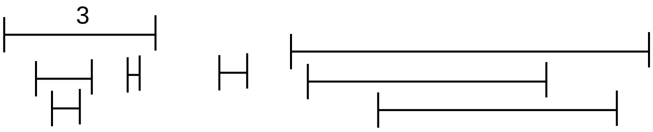
- Forward sweep line on x + heap or BST on y.
- N inserts, ≤ N deletes: O(N log N)

Three Interesting Problems

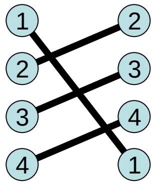
1. Given a set of points in 2D, compute the "domination count" for each point.



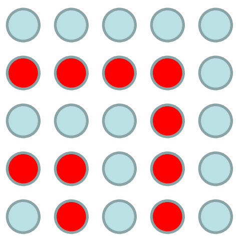
2. Given N intervals on the number line, compute for each one the number of intervals contained within it.

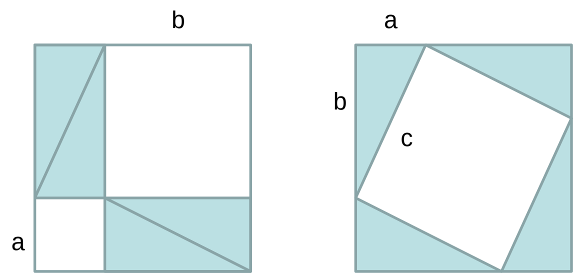


3. Count the number of crossings to the right…



Visualizing Problems Graphically





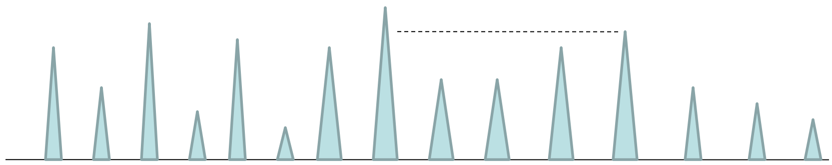
sum of first n odd numbers = n^2

$a^2 + b^2 = c^2$

“Parameter Space”

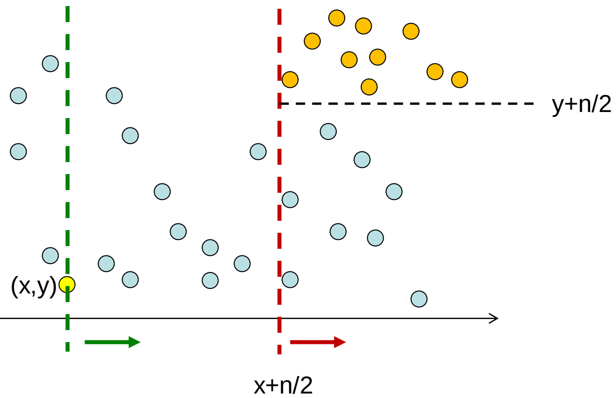
- Suppose a robot arm has 4 movable joints, each with one degree of freedom.
- You want to move the arm from its current state to a new “goal” state (say, with the end of the arm at a certain position).
- Certain joint configurations are not valid (i.e., the arm can’t fold back and cross through itself!)
- Moving the arm becomes an easier problem, conceptually, if we view it as finding a path through a 4d “parameter space” in which obstacles correspond to invalid configurations.

Example: Longest Line of Sight



- Given the heights of N individuals standing in a line.
- **Goal:** find the length of the longest line of sight (difference in indices between two people between whom everyone else is strictly shorter).
- How might you solve this problem using sweep lines?

Example: Counting Distant Pairs of Points…



Example: Closest Pair of Points in 2D