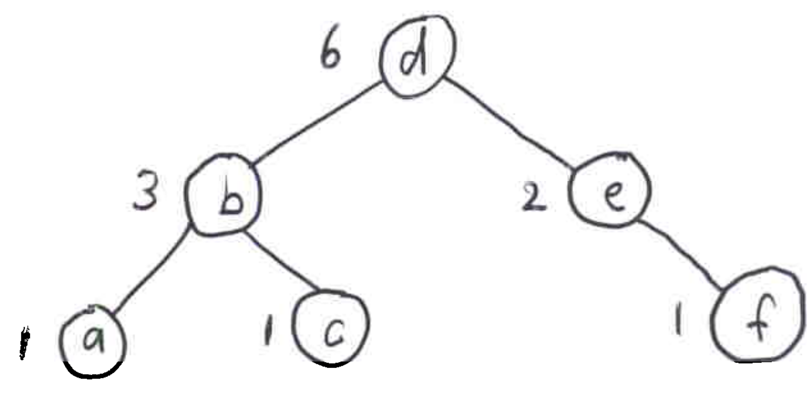
Doubly Ordered Trees

Idea: Use both symmetric order and heap order (on different values)

1. Dynamic order statistics (CLRS 302)

access k^{th} in a list

Method: store subtree size in each node



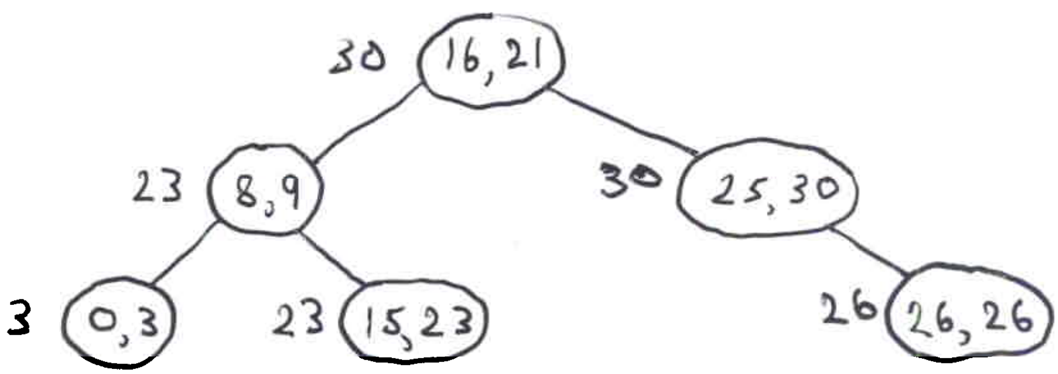
$$size(x) = size(left(x)) + size(right(x)) + 1$$

$O(1)$ per rotation

2. "Interval trees" (CLRS 311)

store intervals [x, y]

symmetric order on x
store max y-value in subtree



can do intersection, containment queries

But (at least) one other kind of "interval tree" exists: CLRS dfn <u>not</u> standard

Segment trees are a related structure

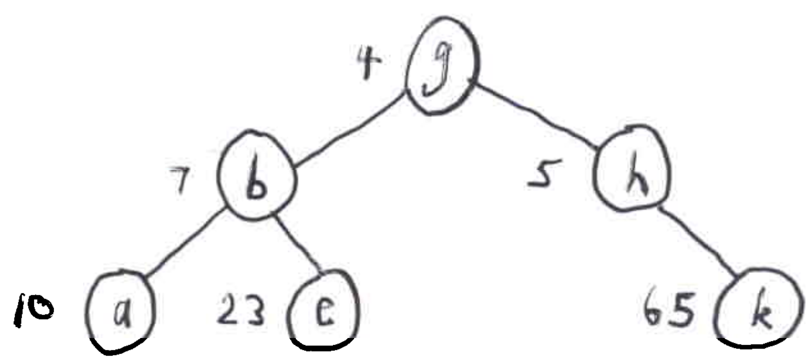
3. Treaps: randomized search trees (CLRS 296)

Each newly inserted item gets a random priority

Maintain symmetric order by value, heap order by priority:

after insert rotate up along access path to restore heap order

The tree always looks like a tree generated by random insertions



Big drawback: high-precision priorities

4. Priority Search Trees (McCreight, Section 3)

Store pairs $[x, y]$

Given x_0, x_1, y_1, list all pairs $[x, y]$
with $x_0 \le x \le x_1$ and $y \le y_1$

1 ½-D searching

Time to list k pairs is $O(k + \log n)$

"Interval trees" give $O(k \log n)$

Another approach: make a treap
with y-values as priorities
(Vuillemin: pagoda)

But not balanced: pairs with $x = y$

McCreight: store (up to) to pairs per node:

one (p) with min y-value, the other (q) with splitting x-value.

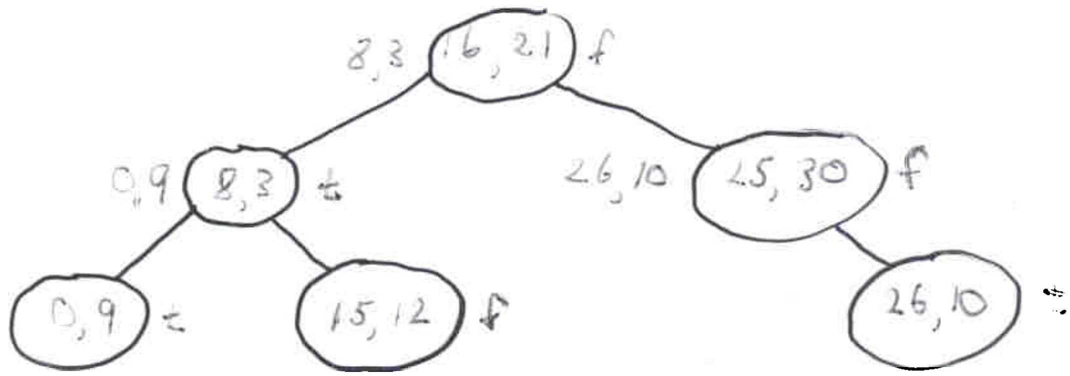
Tree is symmetrically ordered on x-values of q's, min-heap-ordered on y-values of p's.

Each pair appears exactly once as a q, and may appear once as a p, in a proper ancestor.

$t.p$ is a pair with min y that is q in a proper descendant of t and not p for any proper ancestor of t.

t.validP is false iff there is no pair $t.p$ (false ⇒ false at all descendants)

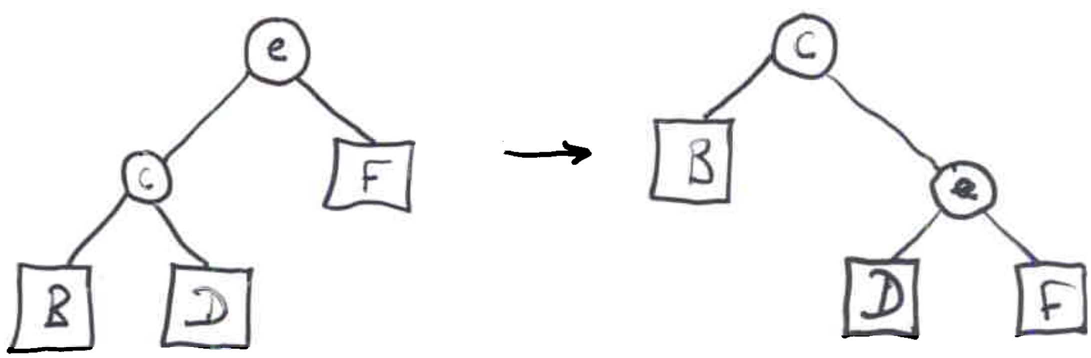
t.duplQ is true iff some ancestor a of t has a.validP = true and $a.p = t.q$



list(t): { report t.p if in range;
report t.q if in range and not t.dupIQ;
if $t.p.y \le y_1$ (or not t.validP and $t.q.y \le y_1$)
then { if $x_0 \le t.q.x$ then list(left(t));
if $x_1 \ge t.q.x$ then list(right(t)) }}

Proof of $O(k + \log n)$ bound: descent both left and right lists a pair; any non-extreme descent list a pair unless terminal.

Rotation



q's okay   p's?

dispose(e); dispose(c);

rotate

extract(e); extract(c)

dispose(t): push t.p down into left or right subtree as appropriate, bumping down lower p's, until some p reaches its q

extract(t): use min available among left(t).p, left(t).q, right(t).p, right(t).q; recur on left(t) or right(t) if necessary

Each recurs down a single tree path

$\Rightarrow O(\lg n)$ time

extract (t):
use min available among :
t.left.p, t.left.q, t.right.p, t.right.q;
recurse on t.left or t.right
as necessary

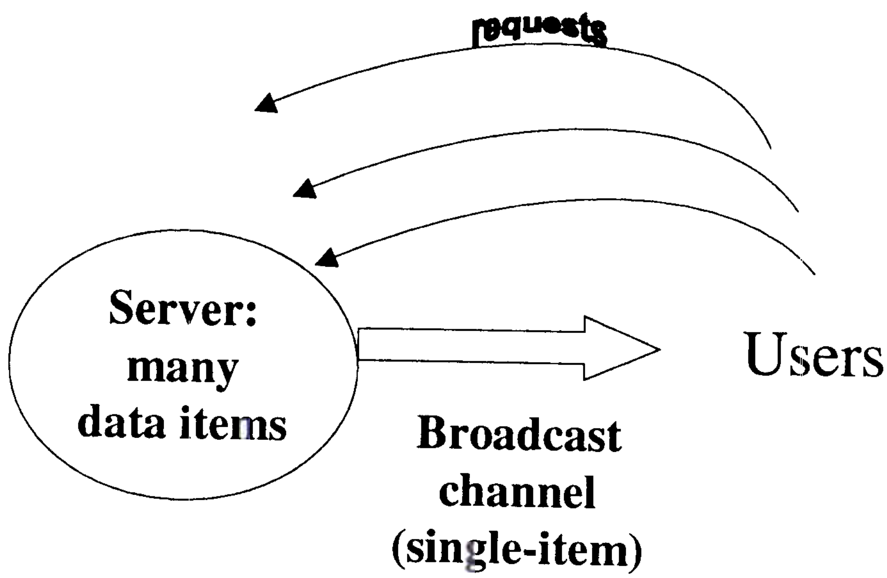
~~dispose (t): if t.valid.p then~~
~~{if t.p.x ≤ t.q.x then~~
~~(dispose into left subtree)~~
~~if t.p = t.left.q then~~
~~t.left.duplQ = false~~
~~else{dispose (t.left);~~
~~t.left.p = t.p;~~
~~t.left.validP = true}~~
~~else (dispose into right subtree);~~
~~t.validP = false}~~

dispose(t): if t.p.x < t.q.x then
if t.p ≠ left(t).q then
{dispose(left(t)); left(t).p = t.p}
else(symmetric on right)

dispose (t): recursively push t.p down into left or right subtree as appropriate, bumping down other p's, until a bumped p meets its q

Broadcast Scheduling

(Lecture by Mike Franklin ⟶ research papers)



One server, many possible items to send.

One broadcast channel.

Users submit requests for items.

Goal: Satisfy users as well as possible, making decisions on-line.

Abstractions:

All items have the same broadcast time.

Minimize the sum of waiting times?

Scheduling Policies (heuristics)

Greedy = Longest Wait first (LWF):

Send item with largest sum of waiting times.

(vs. number of requests or longest single waiting time)

R x W: Max # requests x longest waiting time

Approximations to R x W

Results of Franklin and others:

LWF schedules well "in practice" (in simulations)

but too expensive (linear-time)

This claim used to justify approximations to

R x W, still linear-time but with a smaller

(parameterized) constant.

Questions (for an algorithm guy or gal)

LWF does well compared to what?

⇨ Try a competitive analysis

Can we improve the cost of LWF?

⇨ What data structure?

Parametic Heap

A collection of **items**, each with an associated **key.**

key (i) = $a_i\, x + b_i$ $a_{i,}$, b_i reals, x a real-valued parameter

a_i = slope, b_i = constant

Operations:

make an empty heap h.

insert item i with key $a_i\, x + b_i$ into heap h.

find an item i in heap h of minimum key for $x = x_0$.

delete item i from heap h.

Kinetic Heap

A parametric heap such that successive x-values of find mins are non-decreasing.

(Think of x as time.)

x_c = largest x so far (current time)

Additional operation:

decrease the key of an item i, replacing it by a key that is no larger for all $x \geq$ (next) x_c

Broadcast Scheduling via a Kinetic Heap

Max-heap (replace find min by find max,

decrease key by increase key =

change sign of all keys)

Can implement LWF or R x W or any similar policy:

Broadcast decision is find max plus delete

Request is insert (if first) or increase key (if not)

Only find max need be real-time, other ops

can proceed concurrently with broadcasting

Slopes are integers that count requests

What is known about parametric and kinetic heaps?

A **key** is a **line** ⇨computational geometry

Equivalent problems:

maintain the lower envelope of a collection of lines in 2 D

$\Big\downarrow$ projective duality

maintain the convex hull of a set of points in 2D under insertion and deletion

"kinetic" restriction = "sweep line" query constraint

(Seminal) Results

Overmars and van Leeuwen (1981)

> **Dynamic convex hulls and lower envelopes**
>
> **in $O(\log n)$ time per query,**
>
> **$O(\log^2 n)$ time per update, worst-case**

Basch, Guibas, and Hershberger (1997)

> **"Kinetic" data structure paradigm**

(Much other work: improvements, restrictions, etc.)

Simple Kinetic Heap

A balanced binary tree, with items in leaves in left-right order by key slope.

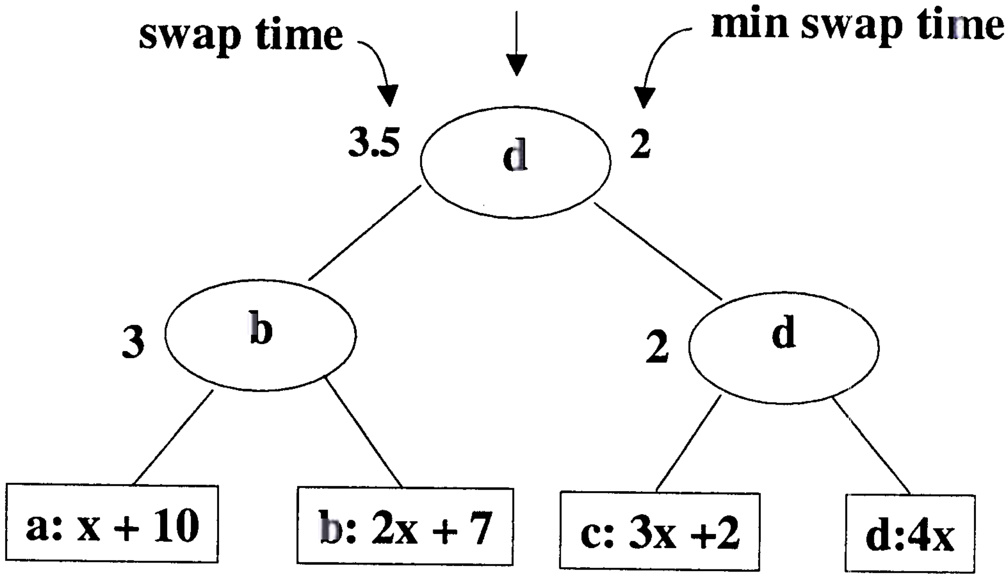
The tree is a tournament on items by current key.

The tree also contains swap times (times when winning keys change) and is a tournament on swap times.

$O(1)$ (worst-case) find min,
$O(\log^2 n)$ amortized insert/delete
Φ = # right child winners $\cdot \log n$

Combines seminal ideas with our own

Is it practical?



$x_c = 0$

A Simple Kinetic Heap