

This is a full 1:25 lecture.

## 1 Splay Trees

Sleator and Tarjan, “Self Adjusting Binary Search Trees” JACM 32(3) 1985

They claim “planning ahead.” But matches previous idea of being lazy, letting potential build up, using it to pay for expensive operation.

Background.

- binary search trees
- bad worst case behavior, so balance
- Lots of variants. All mark. most rotate.
- idea of self adjusting
- Show splay tree is as good as any of them!
- access frequencies
- Static finger theorem.
- working set theorem
- Dynamic finger conjecture.

Lazy

- past balancing schemes require maintenance of balance info at all times, are aggressive
- idea: be lazy, only rebalance when must
- use work of searches to pay for work of rebalancing
- Two separate but related intuitions: path shortening and rebalancing

Path shortening:

- instead, when I take too long, fix it
- i.e., when I go down a long path, make it shorter!
- Intuition: shortening of path releases potential that pays for long traversal (and for work done to shorten after)
- Note inserting item makes longer—source of potential for later shortening
- But wait, shortening this path makes others longer
- So how do I know this shortening is helpful overall?

### Rotation-rebalancing

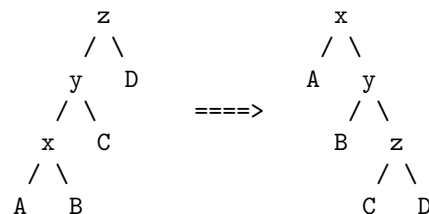
- When my search descends to smaller child, happy
- because can only happen  $O(\log n)$  times before done
- So, “fat children are bad”
- because fat children have high potential
- Do away with fat children?
- potential pays for cost
- Simple idea: single-rotate to root (cheap next time!). Doesn’t work—might pull up a leaf but leave whole fat subtree behind to be descended again and again.
- more sophisticated rotations work. Idea: halve depth of **all** nodes on search path, which draws up whole fat subtree.
- Note: don’t actually destroy fat children—just promote them
- unfortunate: analysis is black magic. no idea how discovered.

### Splaying

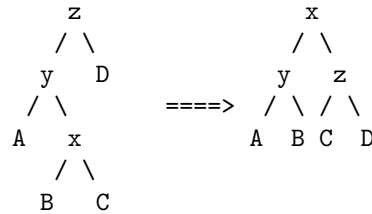
- Moves  $x$  to root
- via a sequence of rotations
- that also shorten original root- $x$  path
- single rotations don’t work—leave original path unchanged
- instead, use double rotations (draw)
- splaying a path halves its length

Draw pictures of rotations.

- zig-zig



- zig-zag



Search algorithm:

- Walk down tree to find item
- Splay item to root

## 1.1 Access Theorem

Search only. Later show insert, delete.

Analysis: different choices of weights. Note *analysis only*, don't affect implementation.

Potential function:

- **weight**  $w_x$  on each node  $x$
- **size**  $s(x)$  is total weight of subtree nodes, i.e. “number of nodes”
- **rank**  $r(x) = \lg s(x)$  (base 2 log) “best height of subtree at  $x$ ”
- **potential**  $\Phi = \sum r(x)$
- Intuition:
  - Having fat subtree deep down is bad—makes us search deep
  - Should be reflected by large potential
  - And it is: because that fat subtree contributes to all ranks above it
  - and we should make that potential vanish to pay for the search
  - ie, bring up the fat subtree

Access lemma: Amortized time to splay node  $x$  given root  $t$  is at most  $3(r(t) - r(x)) + 1 = O(\log(s(t)/s(x)))$ .

- Intuition:  $r(t) - r(x)$  is “ideal” depth of  $x$ .
- and if  $x$  is ideally high in tree (fat) then amortized search time should be small
- (just like normal BST)

- (but we will see power of changing notion of “fat subtree”)

Consequence

- assign weight 1 to each node
- total weight is  $n$
- so rank of root is  $\log n$
- rank of every node is at least 1
- so Access lemma says time to (find and) splay node to root is  $O(\log n)$ .

Proof: Analyze one step:

- new sizes/ranks  $s', r'$ .
- Show amortized cost is  $3(r'(x) - r(x))$  except +1 for last single rot.
- telescope sum for overall result (since final  $r'(x) = r(t)$ ).

Do zig-zig (hardest). zig-zag in paper.

- old  $y$  parent of  $x$  and  $z$  parent of  $y$ .
- Show amortized cost  $3(r'(x) - r(x))$ .
- Only those nodes change ranks
- $x$  children  $A, B$ ,  $y$  has  $C$ .  $z$  has  $D$
- Real cost 2. Potential change  $r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$

Intuition

- $x$  rank increases,  $z$  decreases,  $y$  “unchanged” (not really)
- If  $x$  rank increases a lot, then  $r'(x) - r(x)$  very big
- So  $3(r'(x) - r(x)) \gg (r'(x) - r(x)) + 2$ ,
- i.e. tripling potential “absorbs” real cost into potential
- trouble if  $x$  rank unchanged (can’t decrease)
- But that means  $x$  was fat and should be higher
- and that is what happens—data structure “improves”
- reflected in fact that ranks of  $y, z$  decrease a lot (since they no longer contain  $x$ )
- that decrease in potential “cancels” real rotation cost

Math:

- Simplify:

$$\begin{aligned}
2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
&= 2 + r'(y) + r'(z) - r(x) - r(y) \quad \text{since } r'(x) = r(z) \\
&\leq 2 + r'(x) + r'(z) - r(x) - r(x) \\
&= 2 + (r'(z) - r(x)) + (r'(x) - r(x))
\end{aligned}$$

- last term is desired
- So just need to show  $2 + r'(z) - r(x) \leq 2(r'(x) - r(x))$
- i.e., that  $r'(z) + r(x) - 2r'(x) \leq -2$
- Now note  $r'(z) - r'(x) + r(x) - r'(x) = \lg s'(z)/s'(x) + \lg s(x)/s'(x)$
- And note  $s'(z) + s(x) \leq s'(x)$  since are separate subtrees
- So, eqn is  $\leq \log a + \log(1 - a)$  for some  $0 < a < 1$
- $= \log a(1 - a)$
- which is maximized by maximizing  $a(1 - a)$  at  $a = 1/2$
- which yields  $-2$  (using  $\log_2$ )

Usage:

- One more tricky problem with potential function. Have to account for initial potential
  - (remember: real cost equals amortized cost *minus* change in potential.
  - So must *add* overall decrease on potential to amortized cost in order to bound real cost.
- $m$  accesses on  $n$  nodes
- item  $i$  weight  $w_i$ ,  $\sum w_i = W$ .
- initial potential at most  $n \log W$
- final potential at least  $\sum \log w_i$
- max change at most  $\sum \log W/w_i$ 
  - i.e. add once to amortized cost of splaying item  $i$  amount  $O(\log W/w_i)$ .
  - (note potential change equals cost of splaying each item once)

## 1.2 Applications

Balance theorem: total access for  $m$  ops is  $O((m + n) \log n)$  (as good as any balanced tree)

- weight  $1/n$  to each node.
- potential drop  $n \log n$
- amortized cost of search:  $1 + 3 \log n$

Static Optimality: (as good as any fixed tree)

- item  $i$  accessed  $p_i m$  times
- lower bound for static access:  $m \sum p_i \log 1/p_i$  (entropy)
- item weight  $p_i$
- $W = 1$
- access time for item  $i$  at most  $3(1 - \log p_i) + 1 = O(1 + \log 1/p_i)$
- potential drop  $O(\sum \log 1/p_i)$ .

Static finger theorem:

- $w_i = 1/(1 + |i - f|)^2$
- $\sum w_i \leq 2 \sum 1/k^2 = O(1)$
- access time  $O(\log |i - f|)$
- potential drop  $O(n \log n)$

Working set theorem:

- At access  $j$  to item  $i_j$ , let  $t_j$  be number of distinct items since that item was last accessed. Then time  $O(n \log n + \sum \log t_j)$ .

Unified theorem: cost is sum of best possible choices from previous theorem.

## 1.3 Updates

Update operations: insert, delete, search (might not be there)

- define split, join
- set  $w_i = 1$  so splay is  $O(\log n)$ .
- to split, splay and separate—splay  $O(\log n)$ , potential drops
- to join, access largest item and merge—splay  $O(\log n)$ , root potential only up by  $O(\log n)$

- splits and joints have amortized cost  $O(\log n)$
- insert/delete via split/join
- important to splay on unsuccessful search

#### Remarks

- Top down splaying.
- can choose to splay only when path is “long” (real cost too large so need to amortize). Drawback: must know weights.
- can choose to stop splaying after a while. good for random access frequencies.
- Open: dynamic optimality.
- Open: dynamic finger
- tarjan: sequential splay is  $O(n)$
- Tango-trees:  $O(\log \log n)$  competitive
- conjectured tight upper/lower bounds in Demaine SODA 09