Lec14-splay-trees

Thursday, October 3, 2024 12:17 PM

[Danny Sleator, Tarjan, 1985]

On Monday, we finished up SkipLists, a probabilistic dictionary / BST replacement that is always probably balanced.

Today, we will introduce a data structure that always remains balanced in an amortized sense.

Idea: Modify BST

- move frequently accessed items up in tree
- Simple to implement, no extra storage.
- Amortized $O(\log n)$ performance
- Worst-case $\Omega(n)$ performance

Splay(T,k): If $k \in T$, move k to root with specific transformations of tree.
Otherwise, move either the in-order successor or predecessor of k to the root.

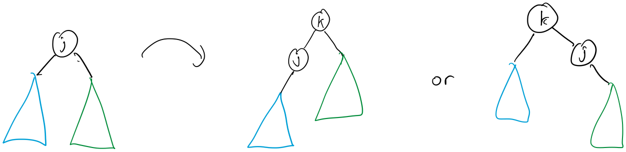
find(T,k): splay(T,k). If root$(T) = k$, return k. Else return NOT FOUND

insert(T,k): splay(T,k). If $j = \text{root}(T) = k$, return Duplicate

Else if $j < k$, set $k.\text{left} = j$, $k.\text{right} = j.\text{right}$, $j.\text{right} = \text{NULL}$

Else if $j > k$, set $k.\text{right} = j$, $k.\text{left} = j.\text{left}$, $j.\text{left} = \text{NULL}$

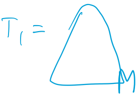
} Make k the root

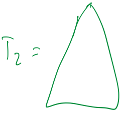


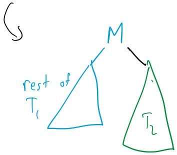
Concat(T_1, T_2): Assume all keys in T_1 < All keys in T_2

Splay(T_1, ∞). Then root$(T_1) = \max(T_1)$ and has no right child.

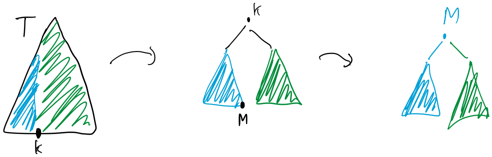
Make T_2 right child of T_1





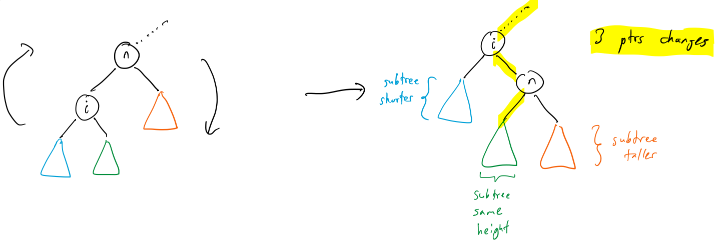


delete(T,k) = splay(T,k). If root r contains k, concat(Left(r), Right(r))

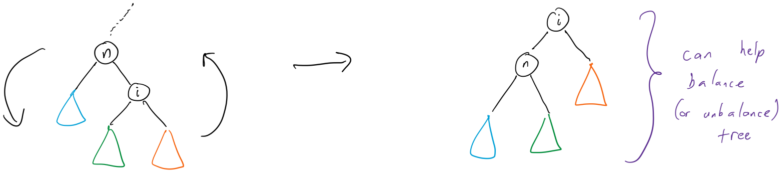


Implementing the splay operation:

Right rotation: (clockwise) rotation around n



Left rotation: (counterclockwise) rotation around n



Note: left & right rotations are inverses

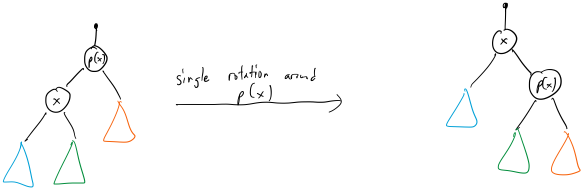
Splay (T,k): find k, walk back up to root, rotating to make k closer

Notice: each rotation moves many nodes closer to root, so we have some choice in who to rotate around.

Want to move x up.

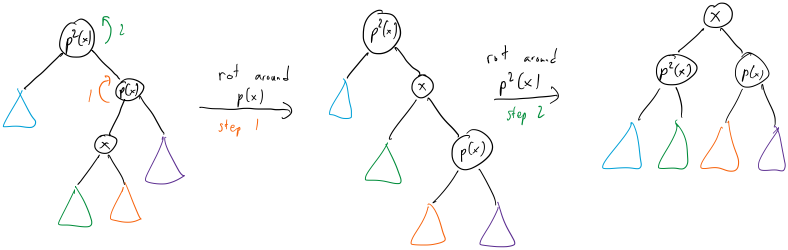
Case 1: x has no grandparent

zig



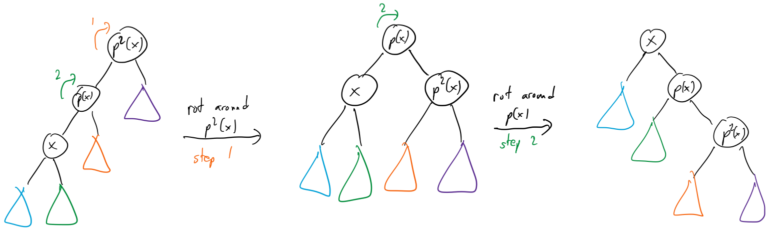
Case 2: x is the left child of parent p(x), which is the right child of parent $p^2(x)$.

zig zag



Case 3: x is left child of parent p(x) which is left child of grandparent $p^2(x)$

zigzig



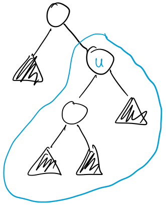
Splays move a node to the root using the rules above to decide what rotations to use.

- Might make tree taller, or less balanced.

  Why is it good?

Ans: Amortized analysis.

Main idea: we use cheap operations to build up money (potential energy) to help pay for expensive ops, so any time we encounter an expensive op, it has already been paid for.



$w(u)$ = weight of u

= # nodes in subtree rooted at u

rank(u) = $\lfloor \log w(u) \rfloor$ ← floor

r(u)

Money invariant: we will always keep rank(u) dollars at every node

Each rotation / double rotation costs \$1 O(1) work

plus money needed to maintain invariant

Aside: we don't actually separately keep track of money in algorithm implementation, just here for the analysis.

Idea: Thm It costs at most $3\lfloor \log n \rfloor + 1$ new dollars to splay + keep the money invariant.

- Spend $O(\log n)$ new dollars, but might do more work if we spend dollars already in tree freed up by the splay.

- Starting with empty tree, after m splays, we'll have spent

$$\underbrace{\le m\cdot\left(3\lfloor \log n\rfloor + 1\right) \text{ dollars}}_{\substack{\text{to pay for work of rots} \\ \text{+ the \$ invariant}}} \quad \Bigg\} \; O(m \log n) \text{ for } m \text{ splays} \Rightarrow O(\log n) \text{ amortized complexity for splay.}$$

Notation: $\text{rank}^p(x)$ $(r^p(x))$ is $\text{rank}(x)$ $(r(x))$ after p rots/double rots during a single splay.

Lemma 1: zig at x costs $3(r'(x) - r(x)) + 1$

Lemma 2: zigzag at x costs $3(r'(x) - r(x))$

Lemma 3: zigzig at x costs $3(r'(x) - r(x))$

} to be proven later

Aside, $r(x) = r^0(x)$

Then cost of splay is:

(telescoping sum)

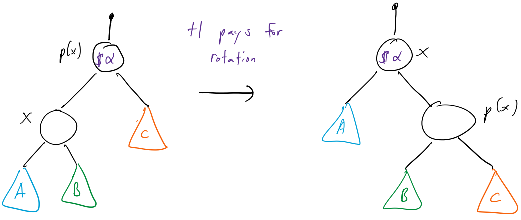
$$\left.\begin{aligned} &3(r'(x) - r(x)) \\ &+ 3(r^2(x) - r'(x)) \\ &+ 3(r^3(x) - r^2(x)) \\ &\quad\vdots \end{aligned}\right\} \text{zigzigs + zigzags}$$

$$+ 3(r^k(x) - r^{k-1}(x)) + 1 \quad \} \text{ zig}$$

$$= 3(r^k(x) - r(x)) + 1$$

$$\left.\begin{aligned} &\le 3 r^k(x) + 1 \\ &\le 3\lfloor \log n \rfloor + 1 \end{aligned}\right\} \text{at root, } r^k(x) = \lfloor \log n \rfloor$$

Lemma 1: $\mathrm{Cost}(\mathrm{zig}) \le 3(r'(x) - r(x)) + 1$



+1 pays for rotation

$$\underbrace{r'(x) = r(p(x))}_{\text{total \# nodes unchanged}}$$

Extra \$ to keep invariant:

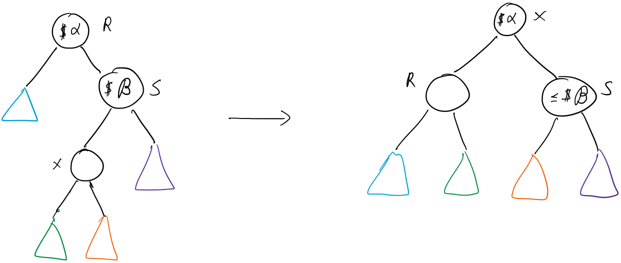
$$\underbrace{\Big[\overset{\$\alpha}{\cancel{r'(x)}} + r'(p(x))\Big]}_{\substack{\$\text{ needed on } x \\ + p(x)}} - \underbrace{\Big[r(x) + \overset{\$\alpha}{\cancel{r(p(x))}}\Big]}_{\substack{\$\text{ already on } x \\ \text{or } p(x)}}$$

$= r'(p(x)) - r(x)$

$\le r'(x) - r(x)$ Since x is now ancestor of $p(x)$

$\le 3\,[r'(x) - r(x)]$ ∎

Lemma 2: $\text{cost}(\text{zigzag}) \le 3(r'(x) - r(x))$ ← budget



Need additional $r'(R) - r(x)$ dollars

$\le r'(x) - r(x)$, So $2(r'(x) - r(x))$ left over in budget.

Also need to pay \$ for rotations.

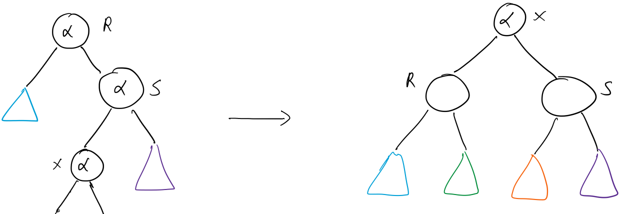
Case 1: If $\underbrace{r'(x) - r(x) \ge 1}_{}$, then $2(r'(x) - r(x)) > 1$, so our budget is good.

Roughly speaking if weight of x doubles. (not exactly because of log-floor function)

Case 2: If $r'(x) - r(x) = 0$, then $r'(x) = r(x)$

Also, $r'(x) = \alpha = r(R) \implies r(x) = r(S) = r(R) = \alpha$.

(because parents always as rich as children)



Suppose $r'(R) = r'(S) = \alpha$

Then $2^{\alpha} \le w'(R) \le 2^{\alpha+1}$

$2^{\alpha} \le w'(S) \le 2^{\alpha+1}$

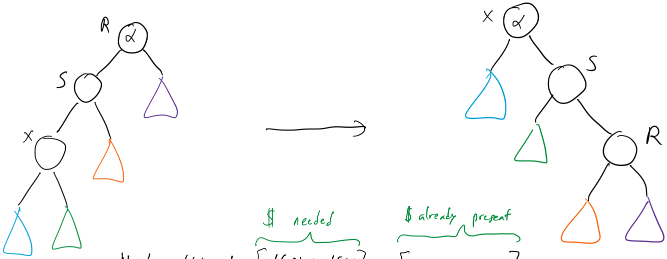
But $w'(x) \ge w'(R) + w'(S) \ge 2 \cdot 2^{\alpha} = 2^{\alpha+1}$

$\Rightarrow r'(x) \ge \alpha + 1$, a contradiction

$\Rightarrow r'(R) + r'(S) < 2\alpha$.

So, we have extra \$1 to spare to pay for rotation.

Lemma 3: $\text{cost}(\text{zigzag}) \le 3(r'(x) - r(x))$



$$\text{Need additional } \overbrace{[r'(S) + r'(R)]}^{\$\text{ needed}} - \overbrace{[r(x) + r(S)]}^{\$\text{ already present}}$$

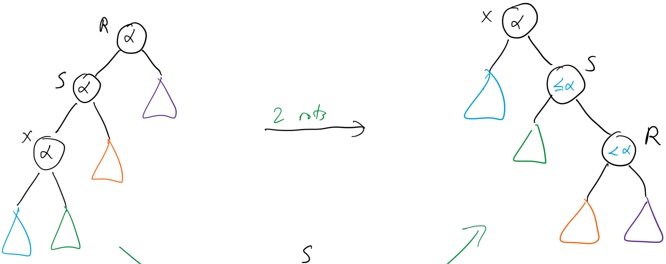
$$\le r'(x) + r'(x) - r(x) - r(x)$$

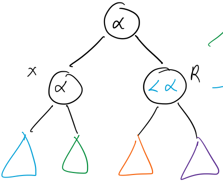
$$= 2[r'(x) - r(x)]$$

$\Rightarrow r'(x) - r(x)$ left in budget

Case 1: $r'(x) - r(x) \ne 0$. Then can pay for rotation out of budget.

Case 2: $r(x) = r'(x) = \alpha$



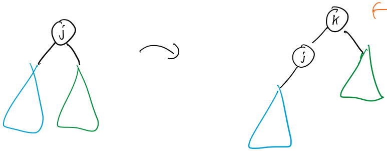


same reasoning as in zigzag case, as otherwise parent would have rank $\alpha+1$.

$\Rightarrow$ before we had $\$3\alpha$ on nodes, no $<\$3\alpha$, so we can use extra $\$1$ to pay for rotations.

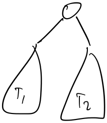
Additional cost of insert & concat = $\$\lfloor \log n \rfloor$

Insert:



need to put $\$\lfloor \log n \rfloor$ on k.

Concat:



root gets $\leq n$ new descendents from T_2, so may need $\leq \lfloor \log n \rfloor$ new dollars

Important: dollars are not actually stored in data structure. Think of like potential energy.

In some other balanced BST, like red/black trees, we explicitly store additional data to maintain invariant.