Lec16-splay-trees [Danny Sleator, Tarjan, 1985]

Thursday, October 3, 2024 12:17 PM

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.

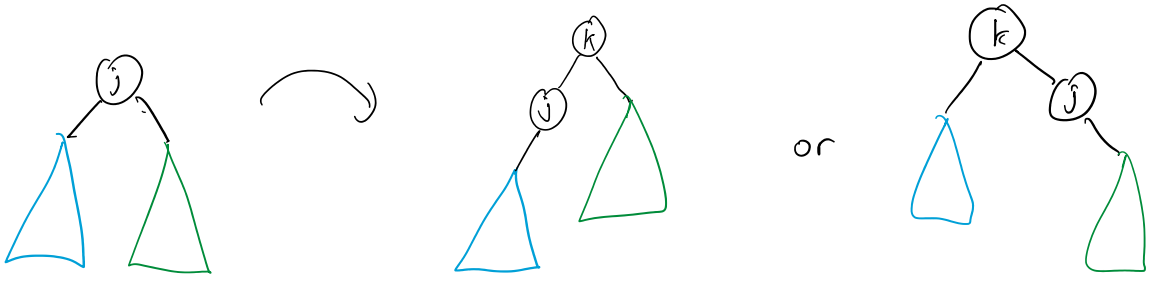
Idea: Modify BST

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

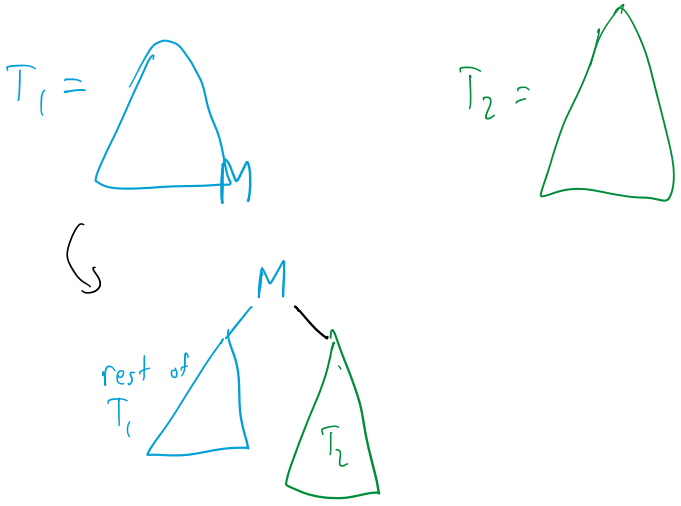
Splay(T,k): If $k \in T$, move k to root with specific transformations of tree.
Otherwise, move either the in-order successor or predecessor of k to the root.

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

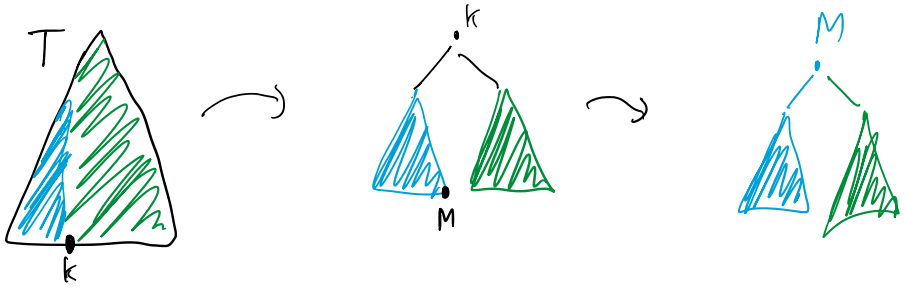
insert(T,k): splay(T,k). If j=root(T)=k, return Duplicate
Else if j<k, set k.left=j, k.right=j.right, j.right=NULL
Else if j>k, set k.right=j, k.left=j.left, j.left=NULL
} Make k the root



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

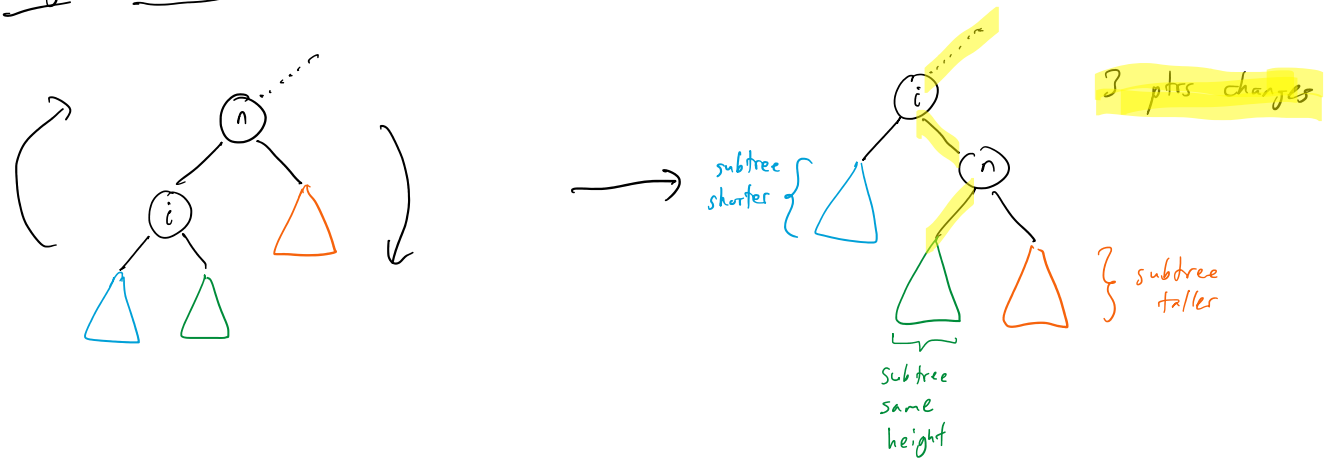


delete (T, k) = splay (T, k). If root r contains k, concat(Left(r), Right(r))

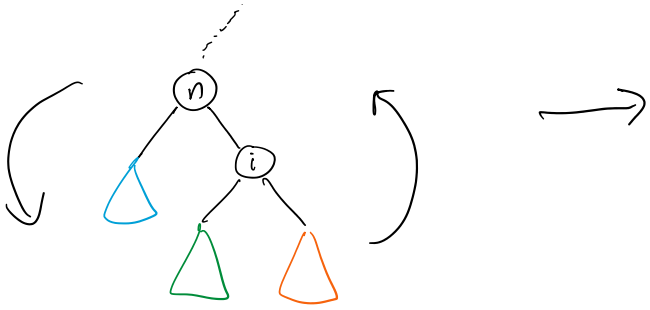


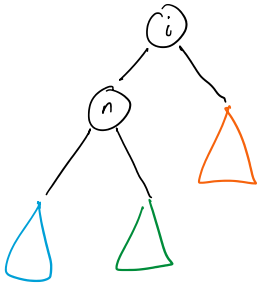
Implementing the splay operation:

Right rotation: (clockwise) rotation around n



Left rotation: (counterclockwise) rotation around n





can help balance (or unbalance) tree

Note: left & right rotations are inverses

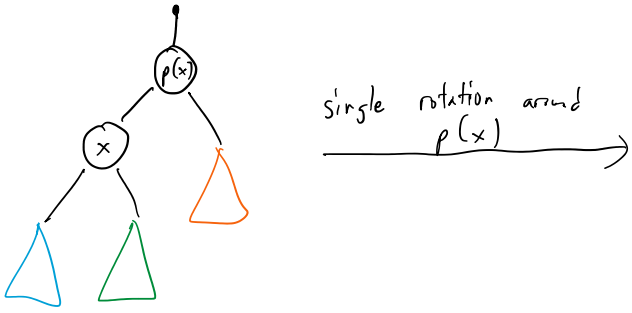
Splay (T, k): find k, walk back up to root, rotating to make k closer

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

Want to move x up.

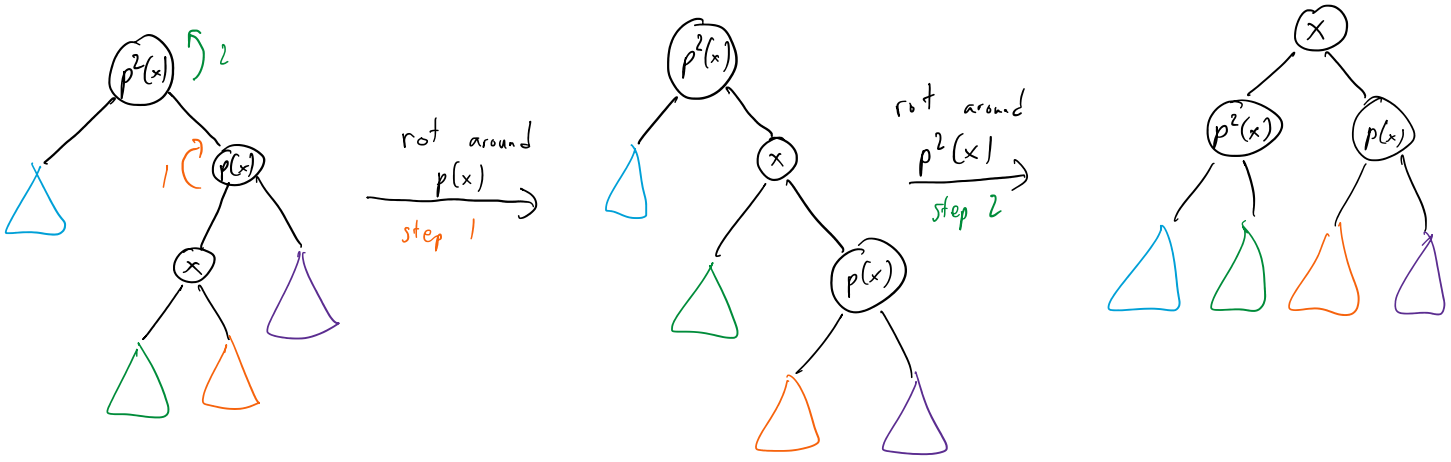
Case 1: x has no grandparent

zig



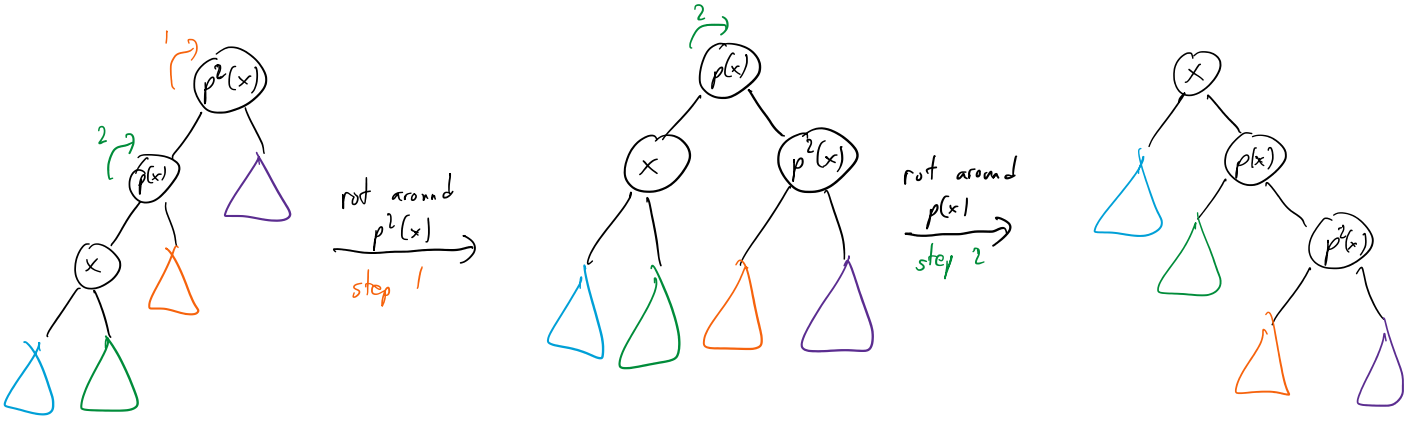
Case 2: x is the left child of parent p(x), which is the right child of parent $p^2(x)$.

zig zag



Case 3: x is left child of parent $p(x)$

zigzig which is left child of grandparent $p^2(x)$



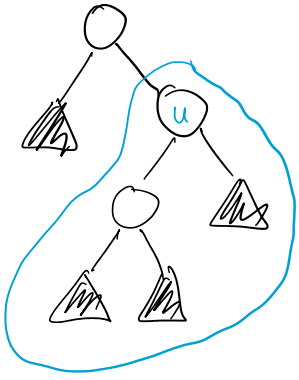
Splays move a node to the root using the rules above to decide what rotations to use.

- 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

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

$r(u)$

Money invariant: we will always keep $\text{rank}(u)$ dollars at every node

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

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 <u>work</u> if we spend dollars already in tree freed up by the splay.
- Starting with empty tree, after m splays, we'll have spent

$$\leq m \cdot (3\lfloor \log n \rfloor + 1) \text{ dollars}$$

to pay for work of rots & the \$ invariant

} $O(m \log n)$ for m splays $\Rightarrow O(\log n)$ amortized complexity for splay.

Notation: $\text{rank}^p(x)$ is $\text{rank}^{(r^p(x))}$ ($r(x)$) 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:

$$\begin{aligned} & 3(r'(x) - r(x)) \\ & + 3(r^2(x) - r'(x)) \\ & + 3(r^3(x) - r^2(x)) \\ & \vdots \\ & + 3(r^k(x) - r^{k-1}(x)) + 1 \end{aligned}$$

(telescoping sum)

} zigzigs + zigzags

} zig

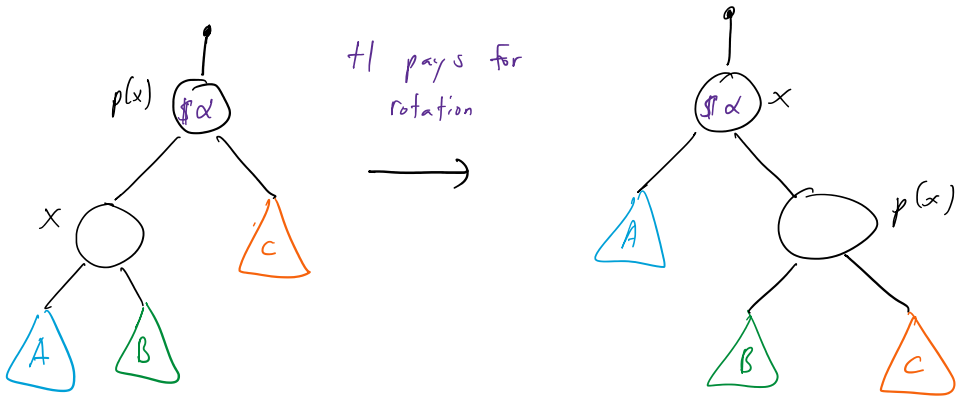
$$= 3\left(r^k(x) - r(x)\right) + 1$$

$$\leq 3\, r^k(x) + 1$$

$$\leq 3\lfloor \log n \rfloor + 1$$

} at root, $r^k(x) = \lfloor \log n \rfloor$

Lemma 1: $\text{cost}(\text{zig}) \leq 3\left(r'(x) - r(x)\right) + 1$



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

total # nodes unchanged

Extra $ to keep invariant:

$$\underbrace{\left[r'(x) + r'(p(x))\right]}_{\$\alpha} - \underbrace{\left[r(x) + r(p(x))\right]}_{\$\alpha}$$

$ needed on x + $p(x)$

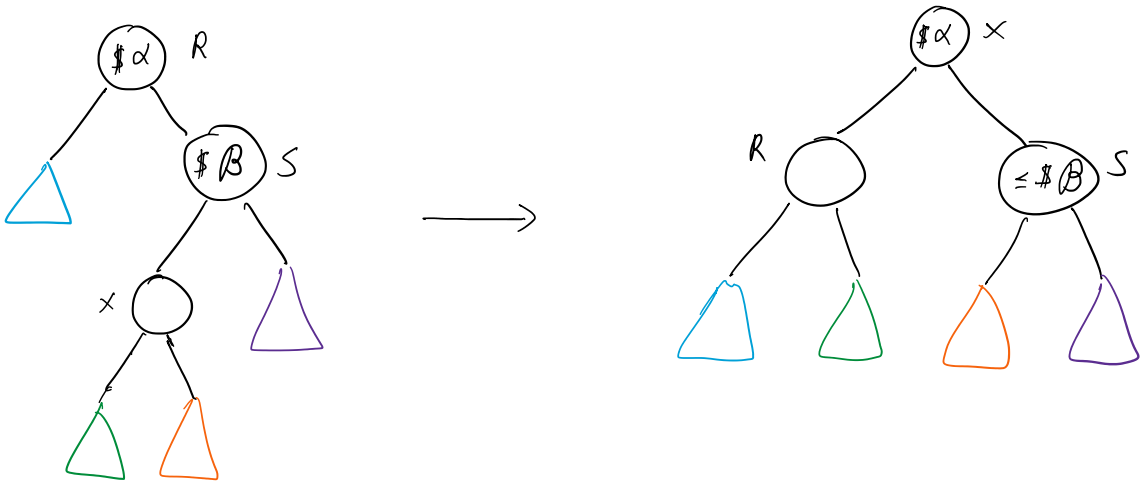
$ already on x or $p(x)$

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

$$\leq r'(x) - r(x)$$ since x is now ancestor of $p(x)$

$$\leq 3\left[r'(x) - r(x)\right]$$

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



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

$\leq r'(x) - r(x)$, so $2(r'(x) - r(x))$ left over in budget.

Also need to pay $ for rotations.

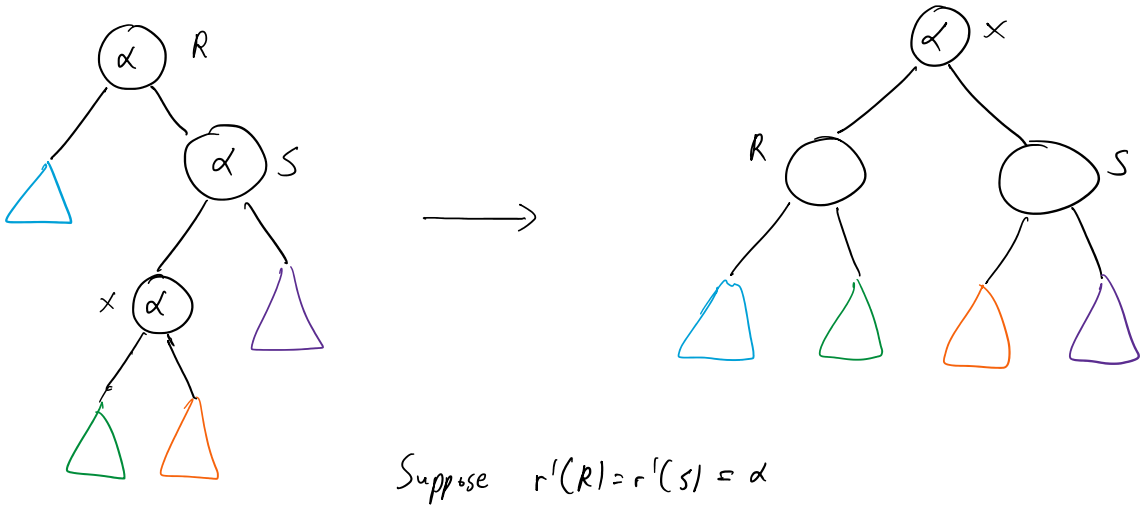
Case 1: If $r'(x) - r(x) \geq 1$, then $2(r'(x) - r(x)) > 1$, so our budget is good.

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

(because parents always as rich as children)



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

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

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

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

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

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

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

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

\$ needed — \$ already present

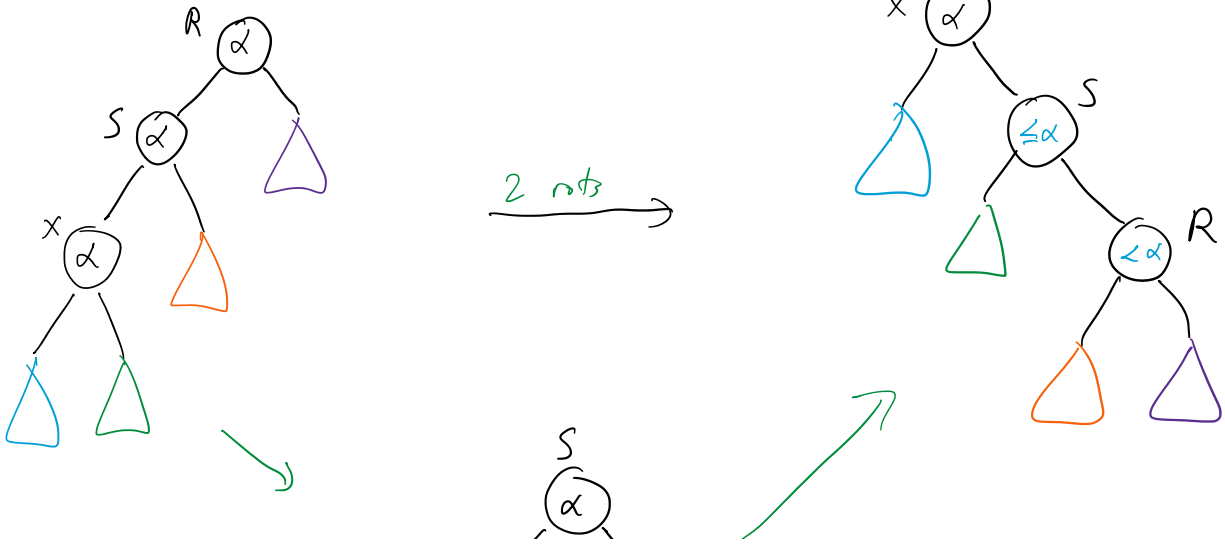
Need additional $[r'(S) + r'(R)] - [r(x) + r(S)]$

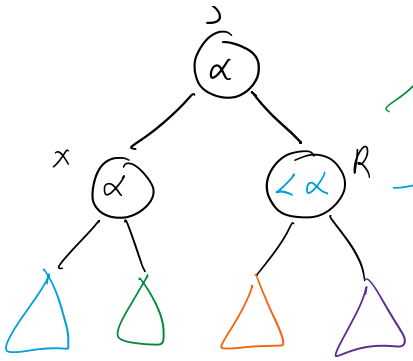
$\leq r'(x) + r'(x) - r(x) - r(x)$

$= 2[r'(x) - r(x)]$ $\Rightarrow$ $r'(x) - r(x)$ left in budget

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

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



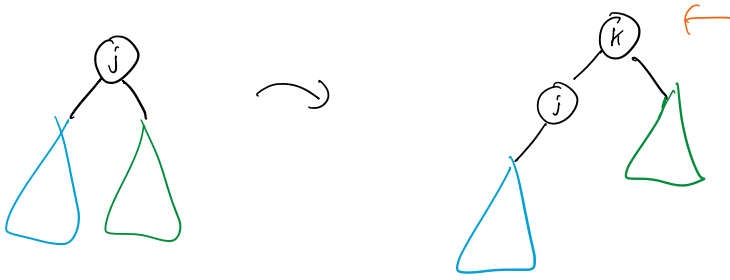


same reasoning as in zig-zag case, as otherwise parent would have rank $\alpha+1$.

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

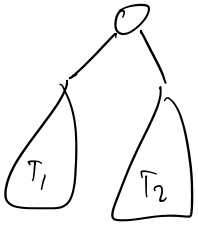
Additional cost of insert & concat = \$⌊log n⌋

Insert:



need to put \$⌊log n⌋ on k.

Concat:



root gets ≤ n new descendants from T_2, so may need ≤ ⌊log n⌋ 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.