Lec16-splay-trees

[Sleator, Tarjan, 1985]

Sunday, October 1, 2023 4:54 PM

A couple lectures ago, we discussed SkipLists, where we introduced a new kind of average-case analysis.

Today, we will see an example of amortized analysis, where we balance occasional expensive ops with cheap ops in aggregate.

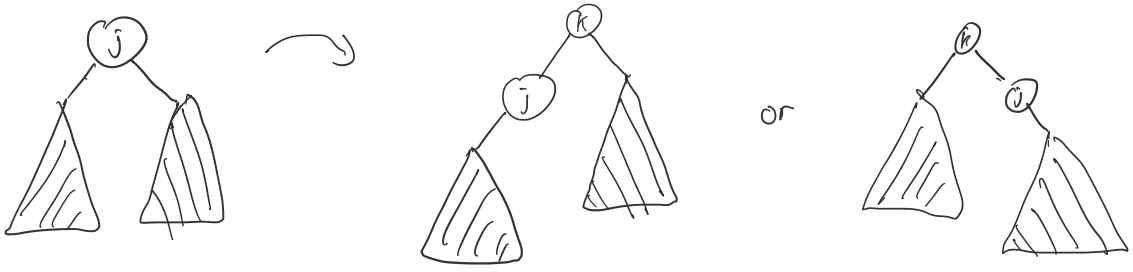
Idea:

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

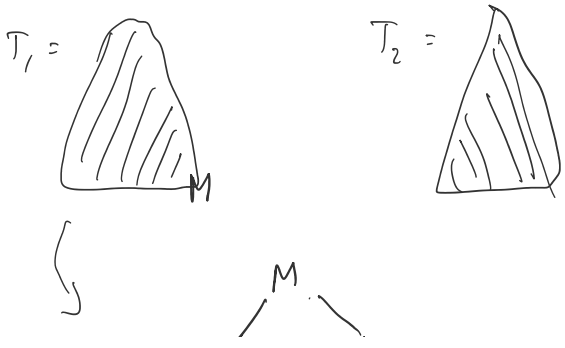
Splay(T,k): if $k \in T$, move k to root with specific transformations of tree,
Otherwise, move either the in-order successor or predecessor of k to the root.

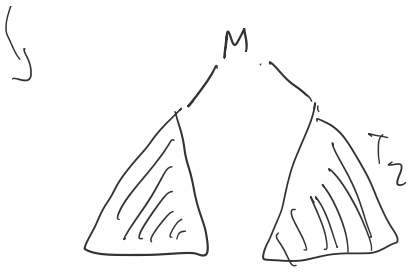
find(T,k): splay(T,k). If root$(T)=k$, return k. Else, return Not Found.

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.

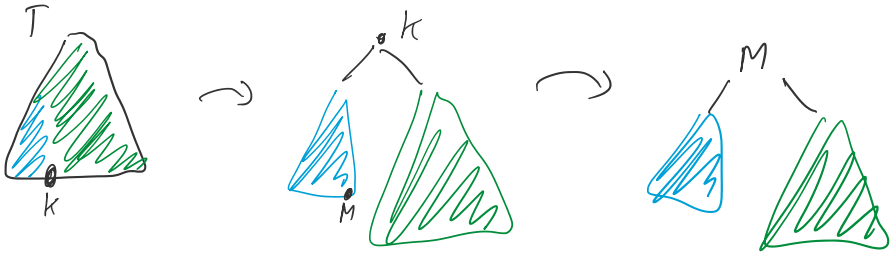


Concat(T_1,T_2): Assume all keys in T_1 < all keys in T_2.
Splay(T_1,∞). Then root$(T_1)=$max(T_1), and has no right child.
Make T_2 right child of T_1.



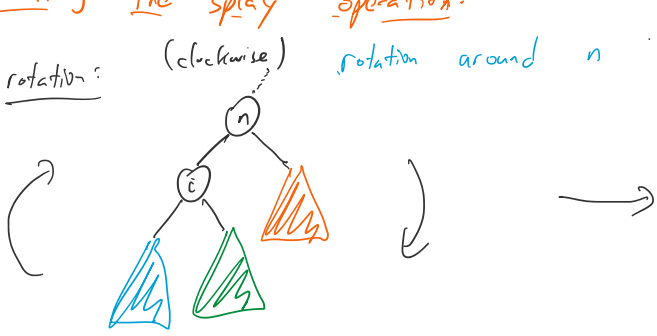


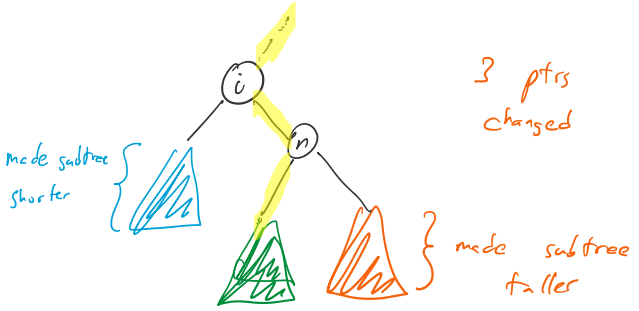
- delete (T, k): splay (T, k). If root r contains k, concat $(\text{Left}(r), \text{Right}(r))$.



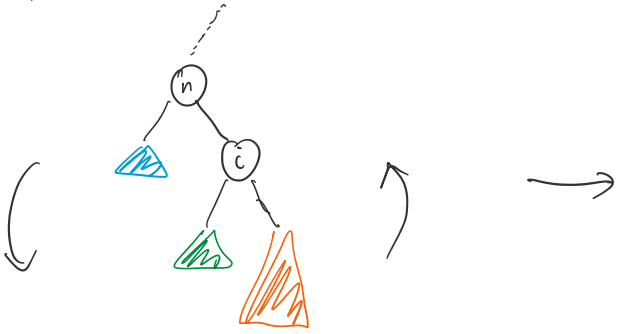
Implementing the splay operation:

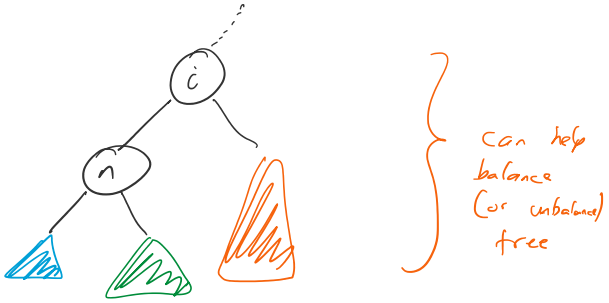
Right rotation: (clockwise) rotation around n





Left rotation: (counter clockwise) rotation around n





Notice: left & right rotations are inverses.

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

Goal 1: move k closer to root

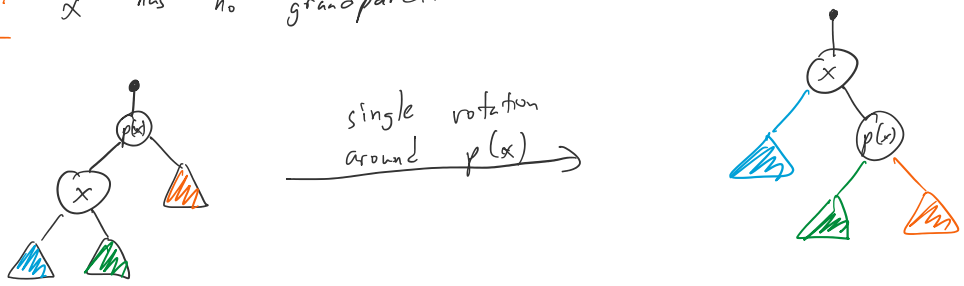
Goal 2: make T "balanced"

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

Want to move node x up.

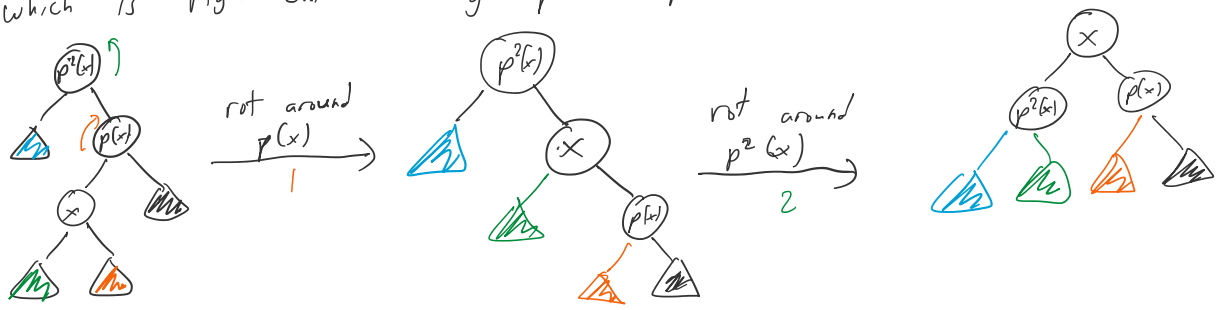
Case 1: x has no grandparent

zig



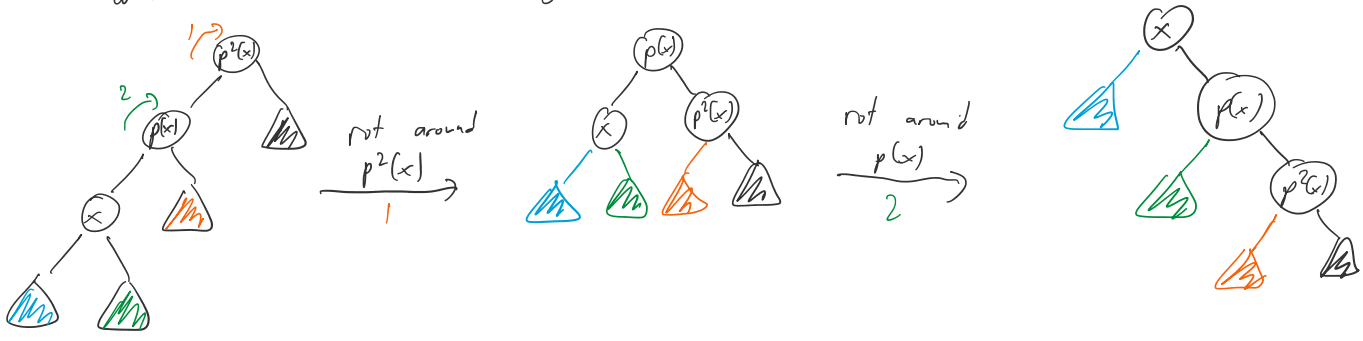
Case 2: x is the left child of parent $p(x)$, which is right child of grandparent $p^2(x)$.

zigzag



Case 3: x is the 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?

Amortized analysis:

Amortized analysis:

- Average cost of operation over a series of operations
  - some ops costly, some cheap

} total area of m bars bounded by some area function $f(m, n)$

num operations, num elements

If $f(m, n) = O(m \log n)$, each operation takes $O(\log n)$ amortized time

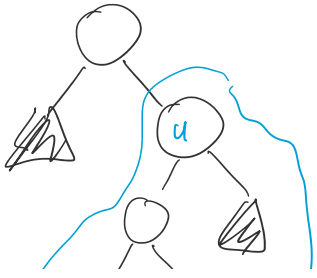
This is different from the expected time analysis of randomized algorithms, since we are averaging over a series of operations instead of over a probability distribution.

- with randomized algs like SkipList, possible for every single search to be bad $O(n)$ instead of $O(\log n)$.
- not possible for amortized $O(\log n)$ time, where one op might be $\Omega(n)$, be on average over many ops you must get $O(\log n)$.

Ex. looking at average number over

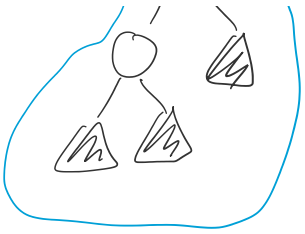
Rolling 10-sided dice (randomized alg) vs. looking at least sig. digit of page number in book (amortized value)

Money idea: 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 offset by cheap ops.



$w(u)$ = weight of u
= # of nodes in subtree rooted at u

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



= # of nodes in u's subtree

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

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 $\leq m(3\lfloor \log n \rfloor + 1)$ dollars.

to pay for both work of rotations & the dollar invariant

$O(m \log n)$ for m splays

Notation: $rank^p(x)$ is rank(x) after p rots/double rots during a single splay

Lemma 1: Zig at x costs $3(rank'(x) - rank(x)) + 1$

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

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

} To be proved later

Then cost of a splay is =

$$3(rank'(x) - rank(x))$$
$$+ 3(rank^2(x) - rank'(x))$$
$$+ 3(rank^3(x) - rank^2(x))$$
$$\vdots$$
$$+ 3(rank^k(x) - rank^{k-1}(x)) + 1$$

(telescoping sum)

} zigzigs + zigzags

} zig

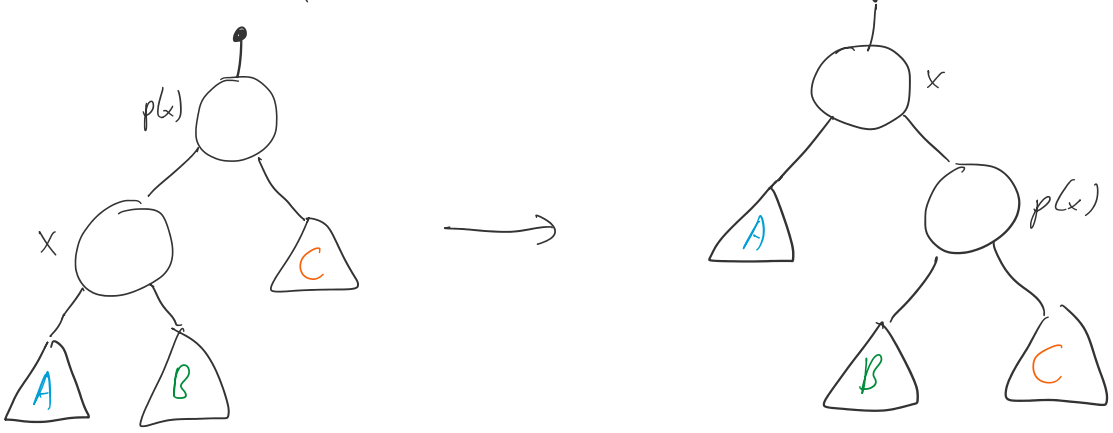
$$= 3\left(rank^{k}(x) - rank(x)\right) + 1$$

$$\leq 3\, rank^{k}(x) + 1$$

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

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

Lemma 1: $cost(zig) \leq 3\left(rank'(x) - rank(x)\right) + 1$



+1 pays for rotation

$rank'(x) = rank(p(x))$

total # nodes is unchanged

Extra $ to keep invariant:

$$\underbrace{rank'(x) + rank'(p(x))}_{\$ \text{ needed on } x + p(x)} - \underbrace{\left[rank(x) + rank(p(x))\right]}_{\$ \text{ already on } x + p(x)}$$

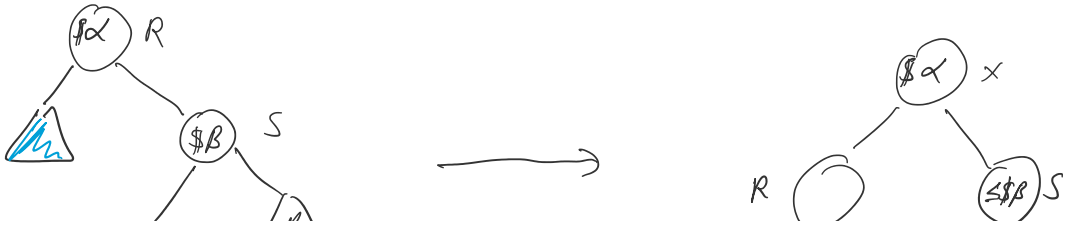
$$= rank'(p(x)) - rank(x)$$

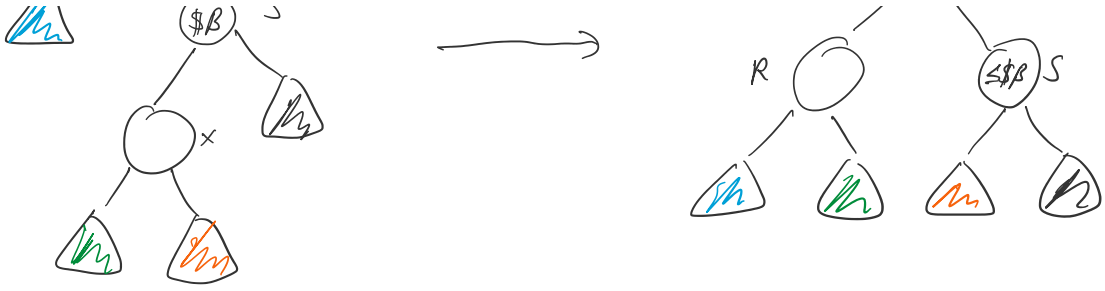
$$\leq rank'(x) - rank(x)$$

since x is now an ancestor of $p(x)$

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

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





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

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

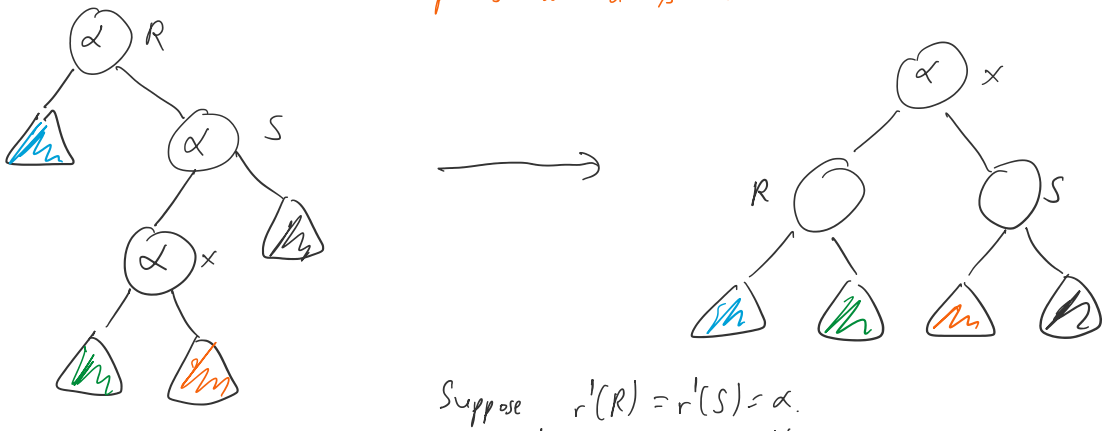
Also need to pay $ for rotations.

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

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

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

because parents are always as rich as children.



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

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

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

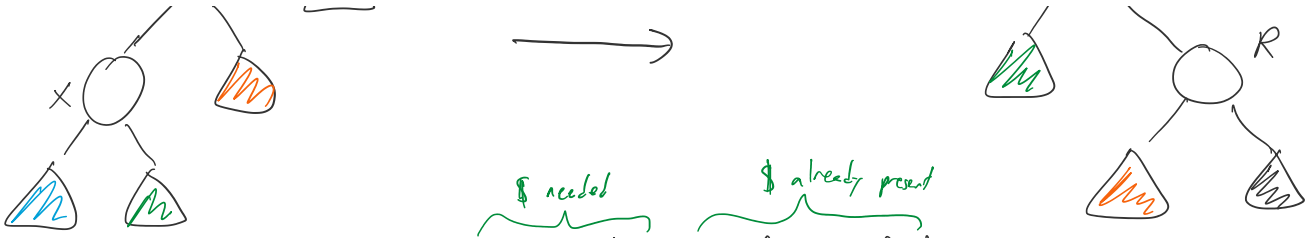
But $w'(\alpha) \geq w'(R) + w'(S) \geq 2 \cdot 2^{\alpha} = 2^{\alpha+1}$

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

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

So, we have $1 to spare from the invariant to pay for the rotation.

Lemma 3: $cost(zigzig) \leq 3(r'(x) - r(x))$



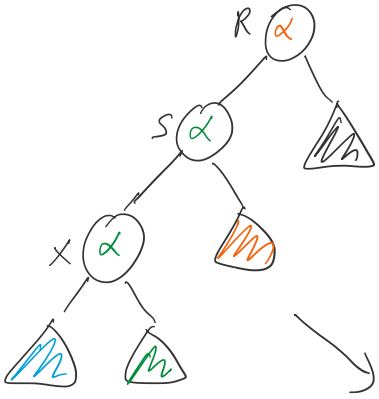
Need additional $\underbrace{r'(S)+r'(R)}_{\$ \text{ needed}} - \underbrace{r(x)-r(S)}_{\$ \text{ already present}}$

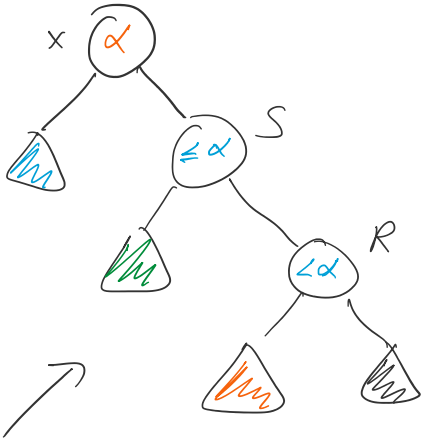
$\leq r'(x)+r'(x)-r(x)-r(x)$

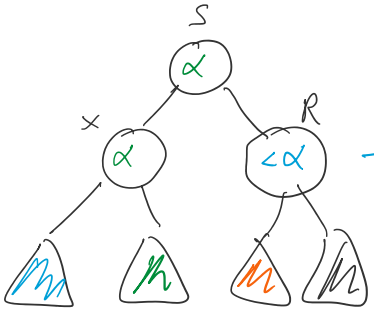
$= 2\left[r'(x)-r(x)\right] \Rightarrow r'(x)-r(x)$ left in budget.

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

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





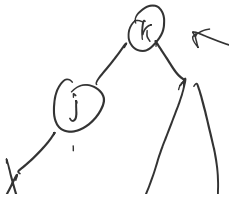


Same reasoning as in zigzag case, as otherwise, the parent would have rank $\alpha+1$.

$\Rightarrow$ Before, we had $\$3\alpha$ on nodes, now $<\$3\alpha$, so we can use extra $\$1$ to pay for rotations.

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

Insert:

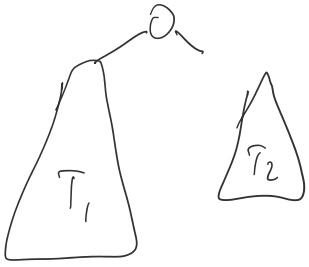


need to put $\$\lfloor \log n \rfloor$ on k

Insert:

$\$\lfloor \log n \rfloor$ on k

Concat:



root gets $\leq n$ new descendants 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 binary trees, like red/black trees,
we explicitly store additional data to maintain invariant.