Splay Trees

CMSC 420: Lecture 8

AVL Trees

- **Nice Features:**
  - Worst case O(log n) performance guarantee
  - Fairly simple to implement

- **Problem though:**
  - Have to maintain extra balance factor storage at each node.

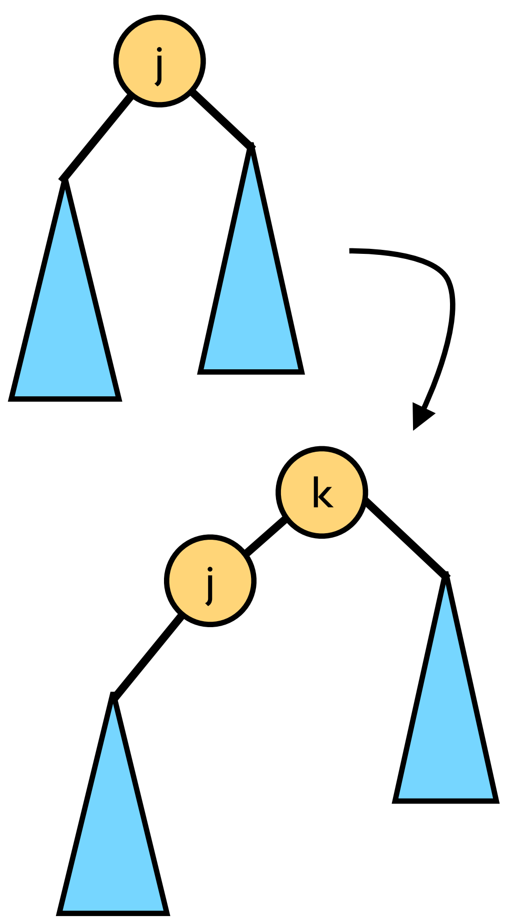
- **Splay trees (Sleator & Tarjan, 1985)**
  - remove extra storage requirement,
  - even simpler to implement,
  - heuristically move frequently accessed items up in tree
  - amortized O(log n) performance
  - worst case single operation is $\Omega(n)$

Splay Trees

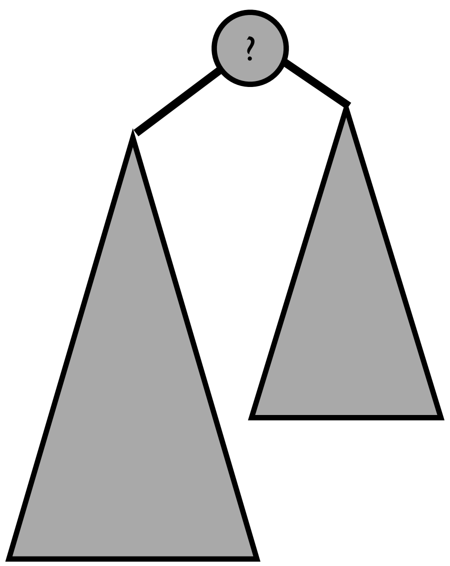
splay(T, *k*): if $k \in$ T, then move *k* to the root. Otherwise, move either the inorder successor or predecessor of *k* to the root.

Without knowing how *splay* is implemented, we can implement our usual operations as follows:

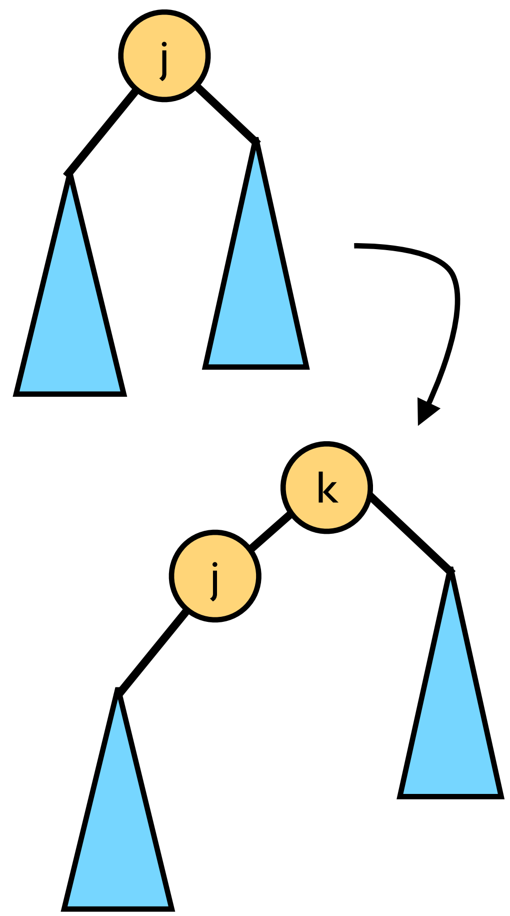
- *find*(T, k): *splay*(T, k). If *root*(T) = *k*, return *k*, otherwise return **not found**.
- *insert*(T, k): *splay*(T, k). If *root*(T) = *k*, return **duplicate!**; otherwise, make *k* the root and add children as in figure.
- *concat*(T_1, T_2): Assumes all keys in T_1 are < all keys in T_2. *Splay*(T_1, ∞). Now root T_1 contains the largest item, 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*)).



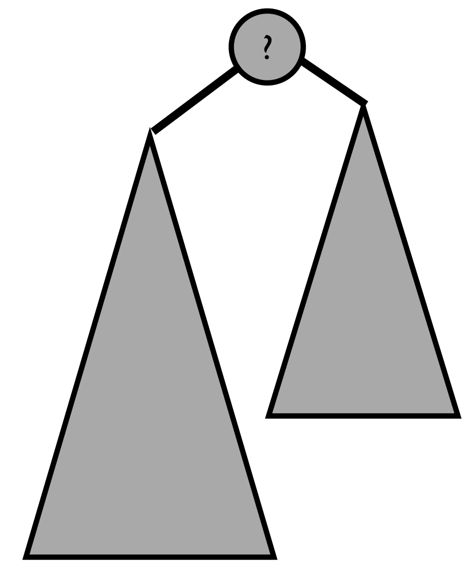
Dictionary Operations, in pictures



find(T, *k*): splay & check root

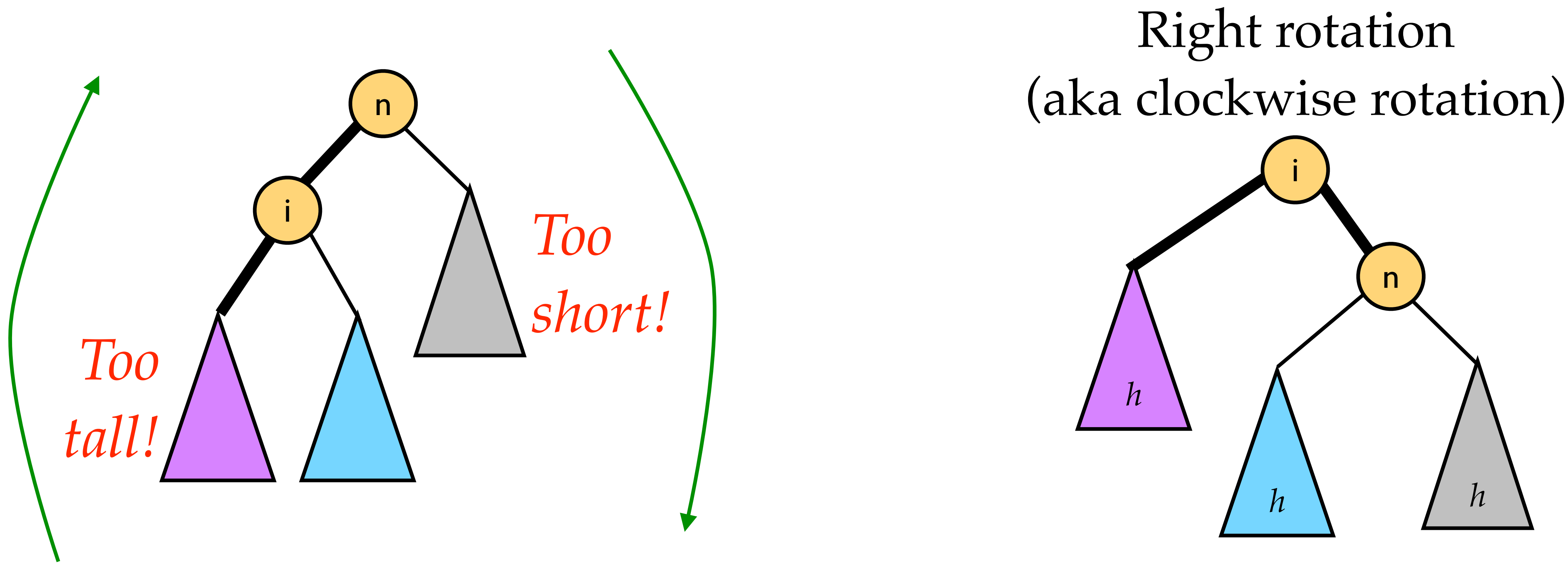


insert(T, k): splay and insert just below root

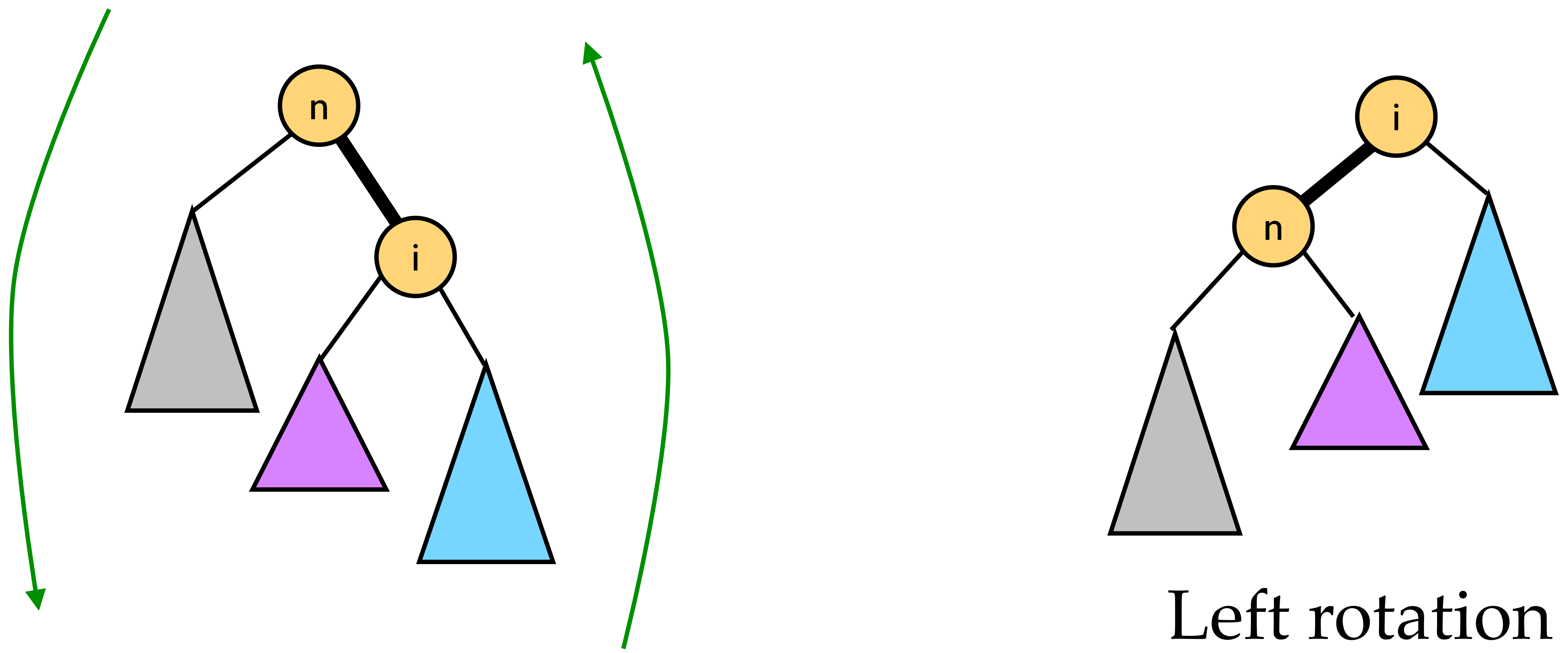


delete(T, *k*): splay & concat left & right subtrees

Right rotation (at n)



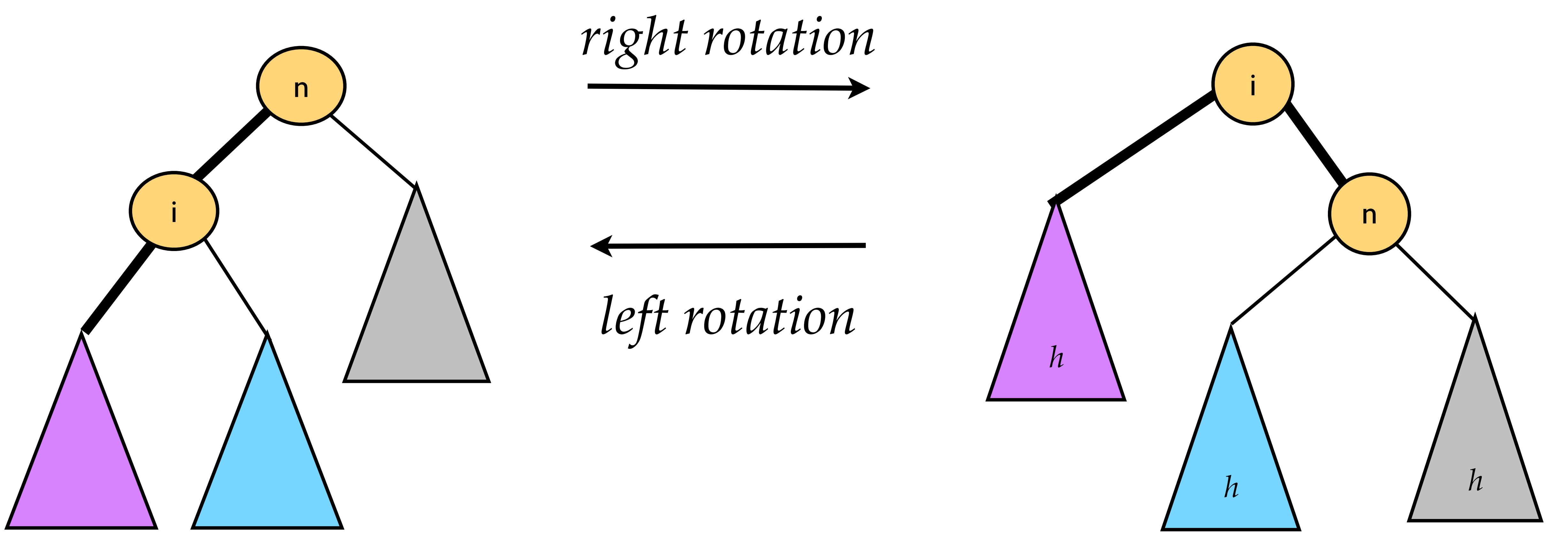
Left Rotation (at n)



Left rotation
(aka counterclockwise rotation)

Only a constant # of pointers need to be updated for a rotation: O(1) time

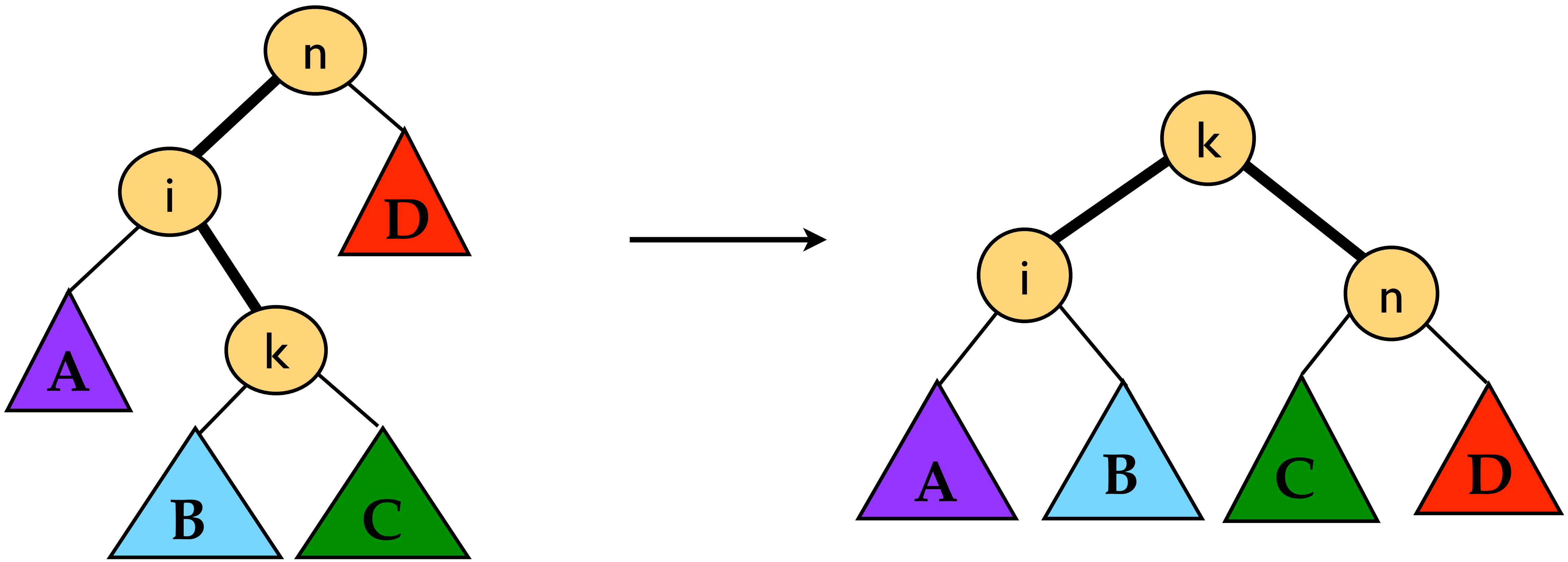
Right & Left Rotations are Inverses



i moves toward the root

n moves toward the root

Double Rotation



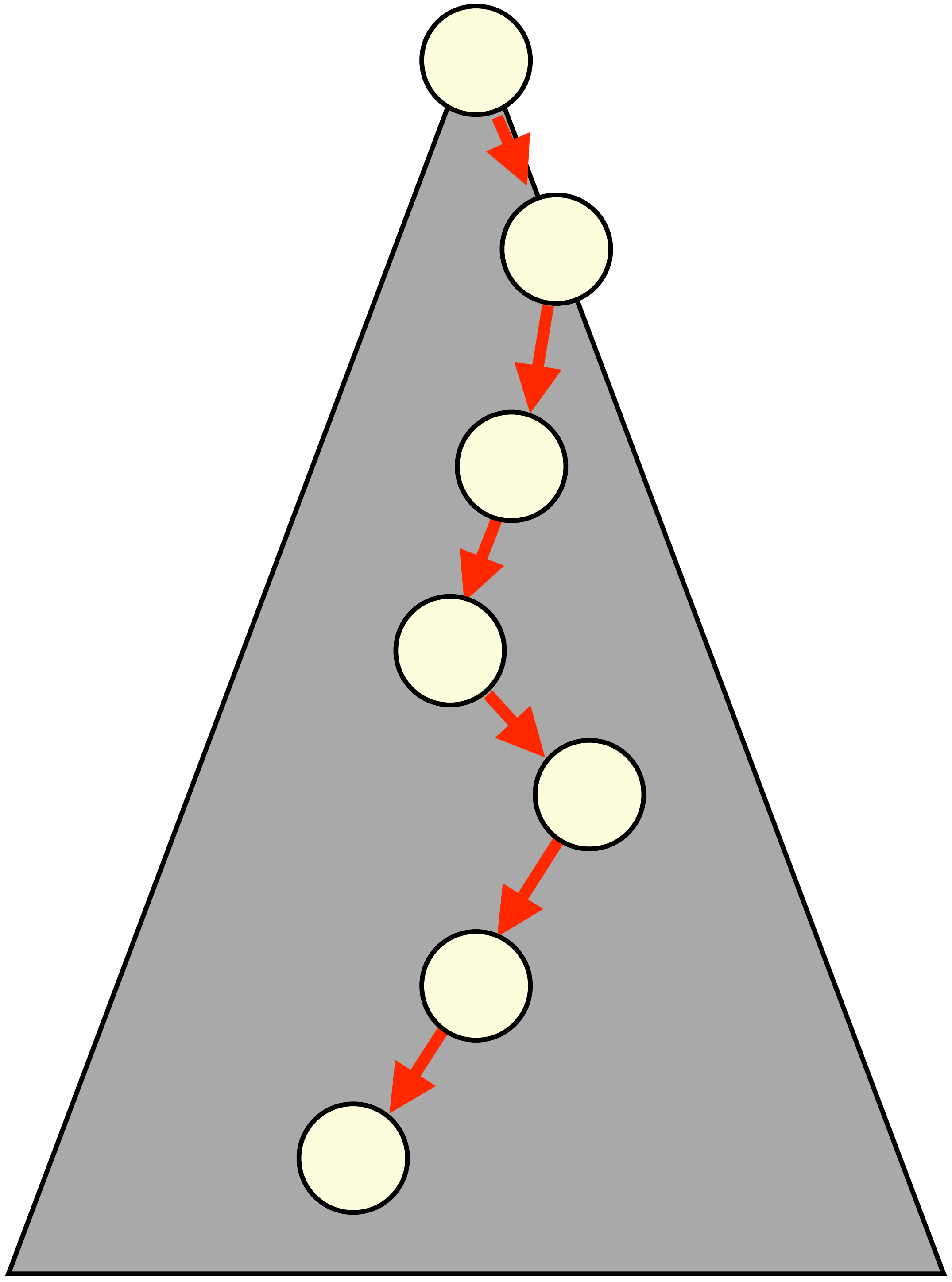
k moves toward the root

Remembering Search Paths

1. **Stack:** as you walk down tree, push nodes onto stack

2. **Parent pointers:** always store parent(*u*) at every node *u*

3. **Link inversion:** as you follow link *u* -> *v*, reverse it to *u* <- *v*.



Splay Operation

- *Splay*(T, *k*): find *k*, walk back up root. Let x be the current node.
- Cases:

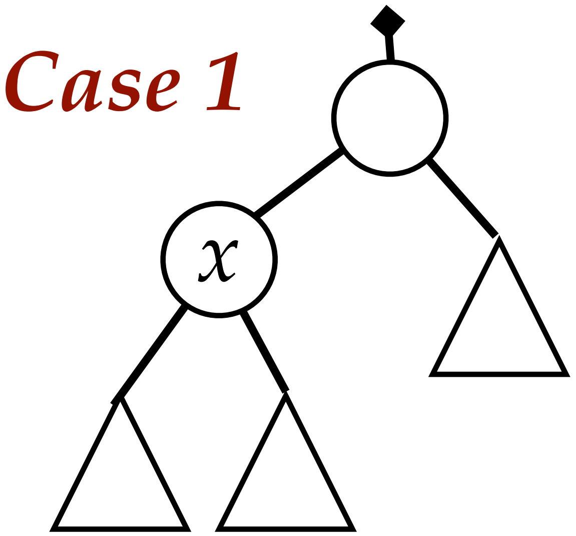
  1. x has no grandparent

  2. x is left child of parent(x), which is the right child of parent^2(x).

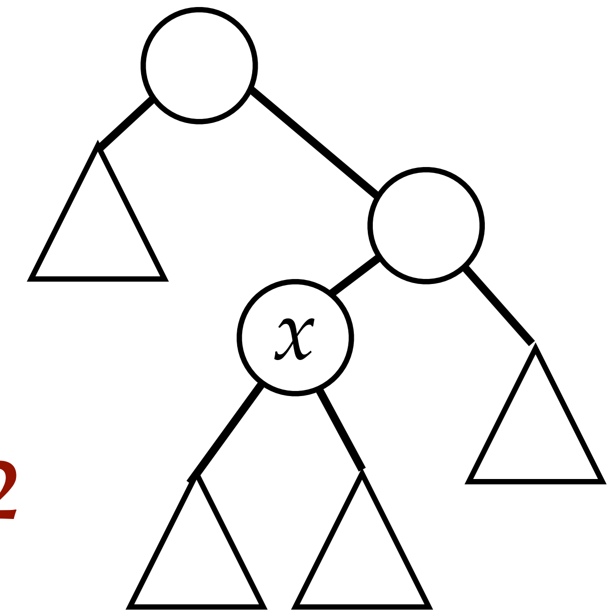
  3. x is left child of parent(x), which is the left child of parent(parent(x)) = parent^2(x)

Rotations with goal: move x toward the root

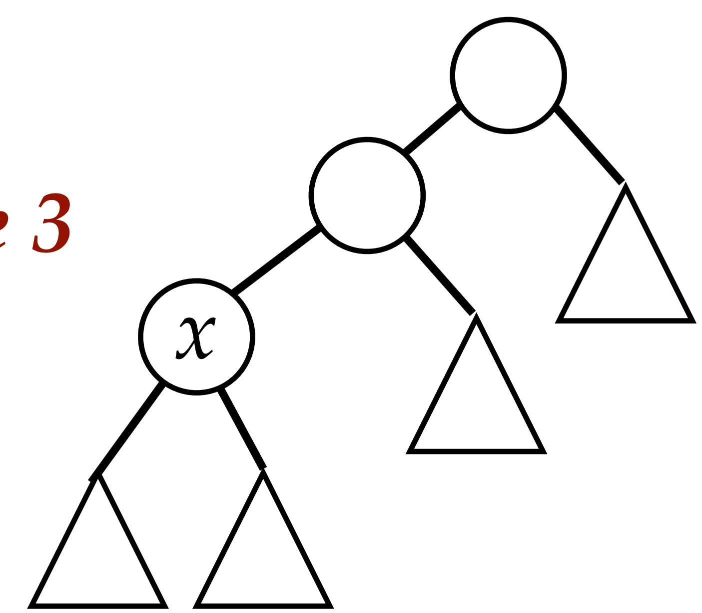
Case 1



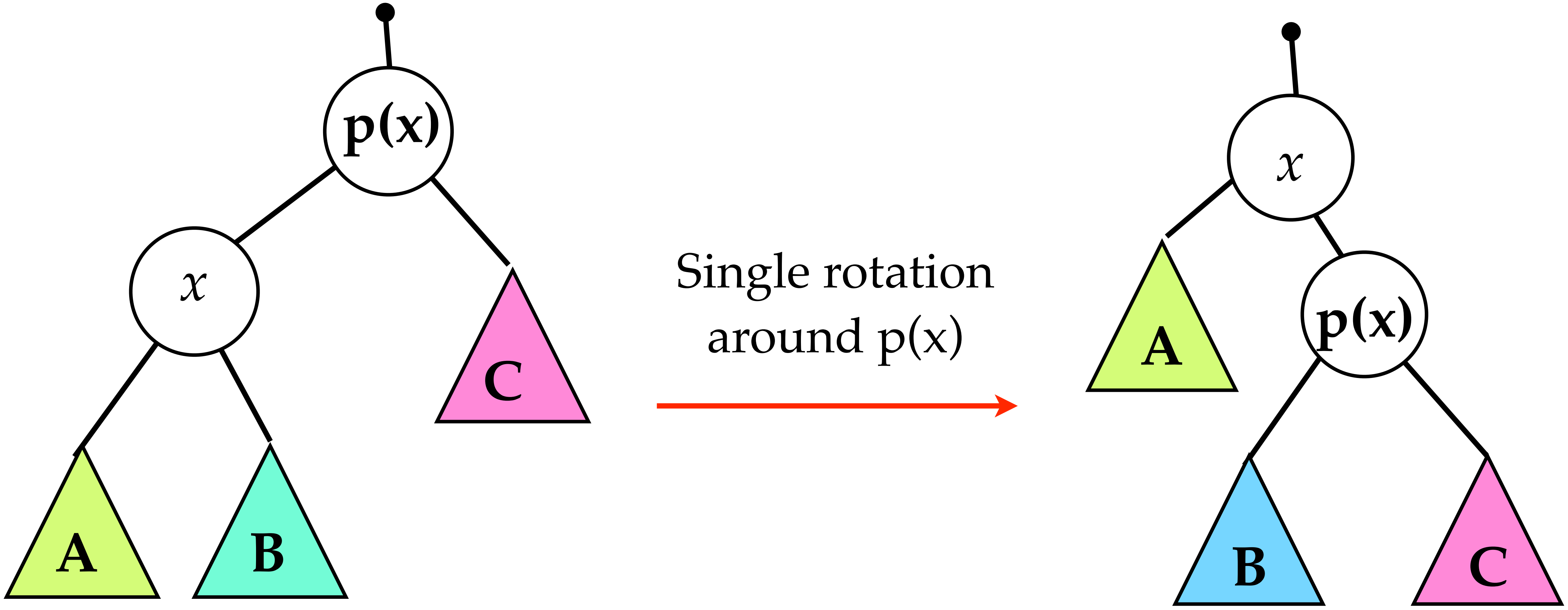
Case 2



Case 3

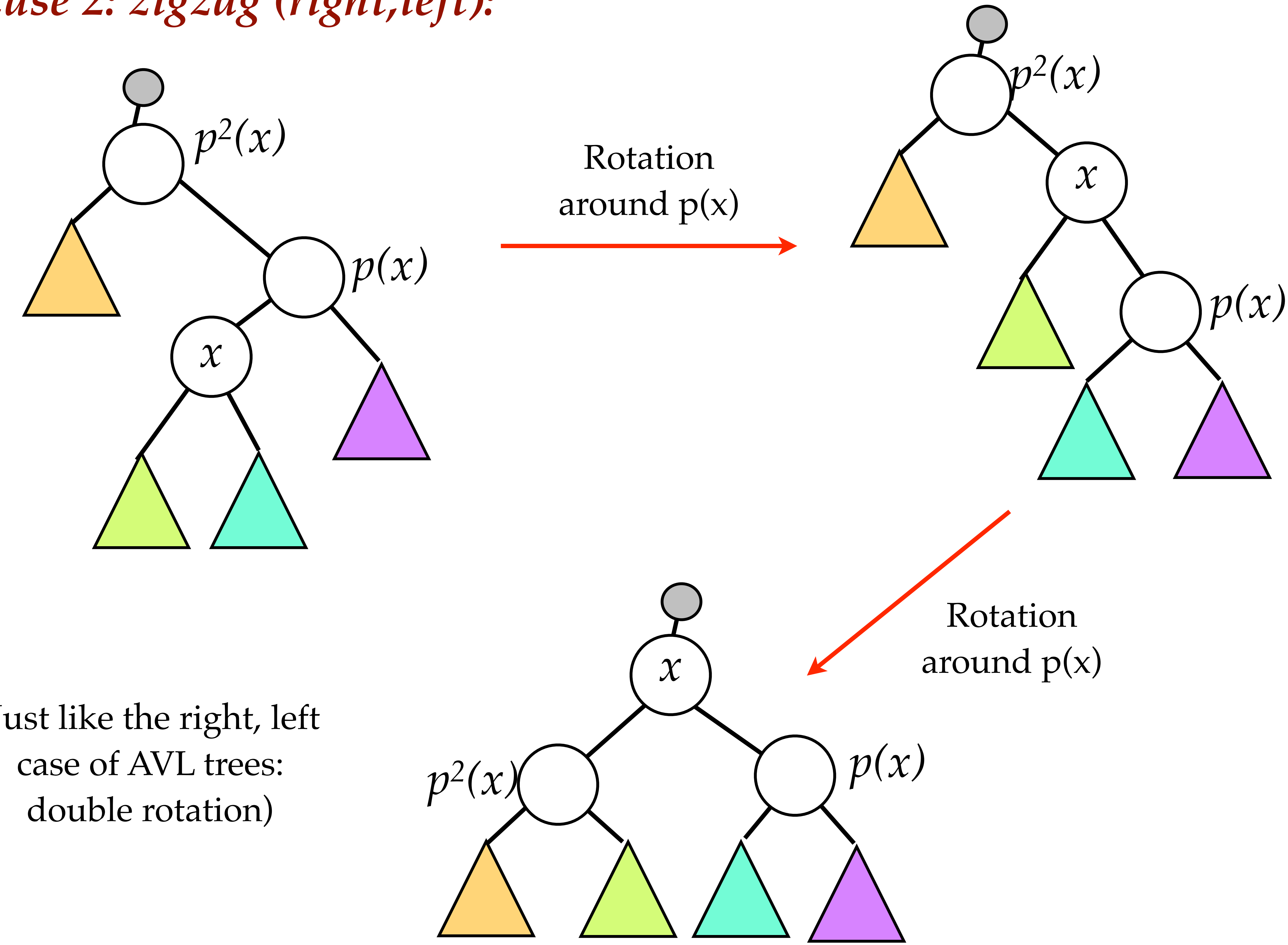


Case 1: no grandparent:



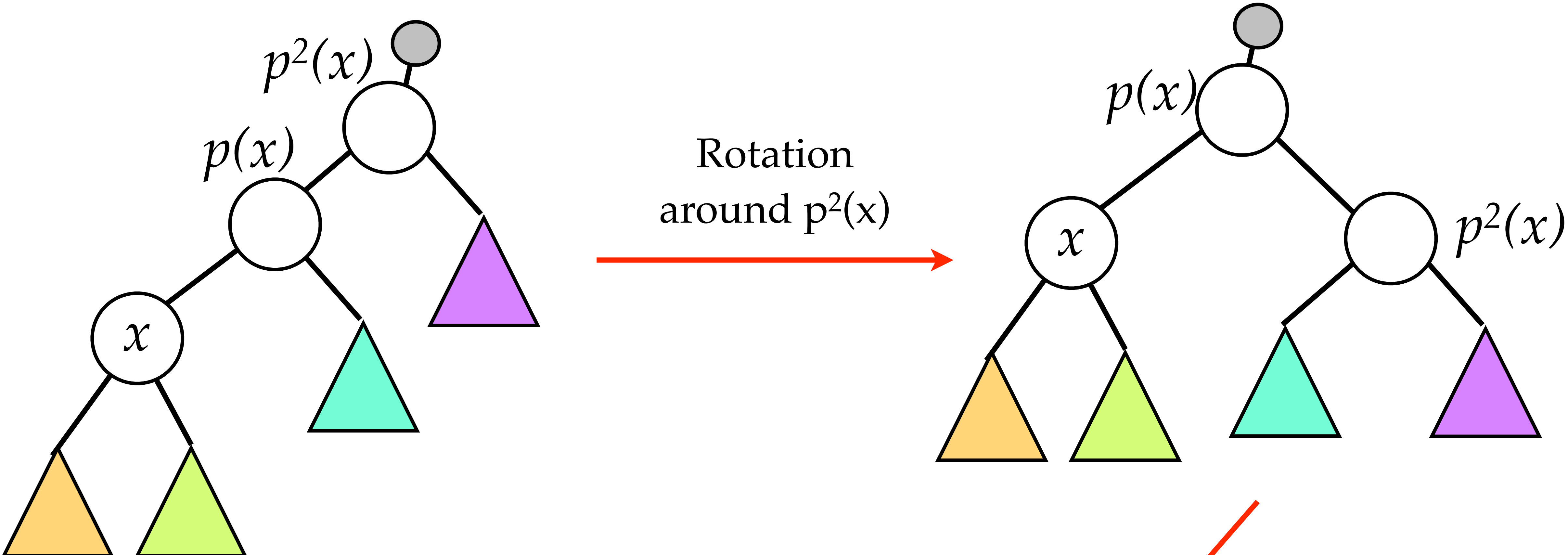
(Just like the single
rotation case of AVL trees)

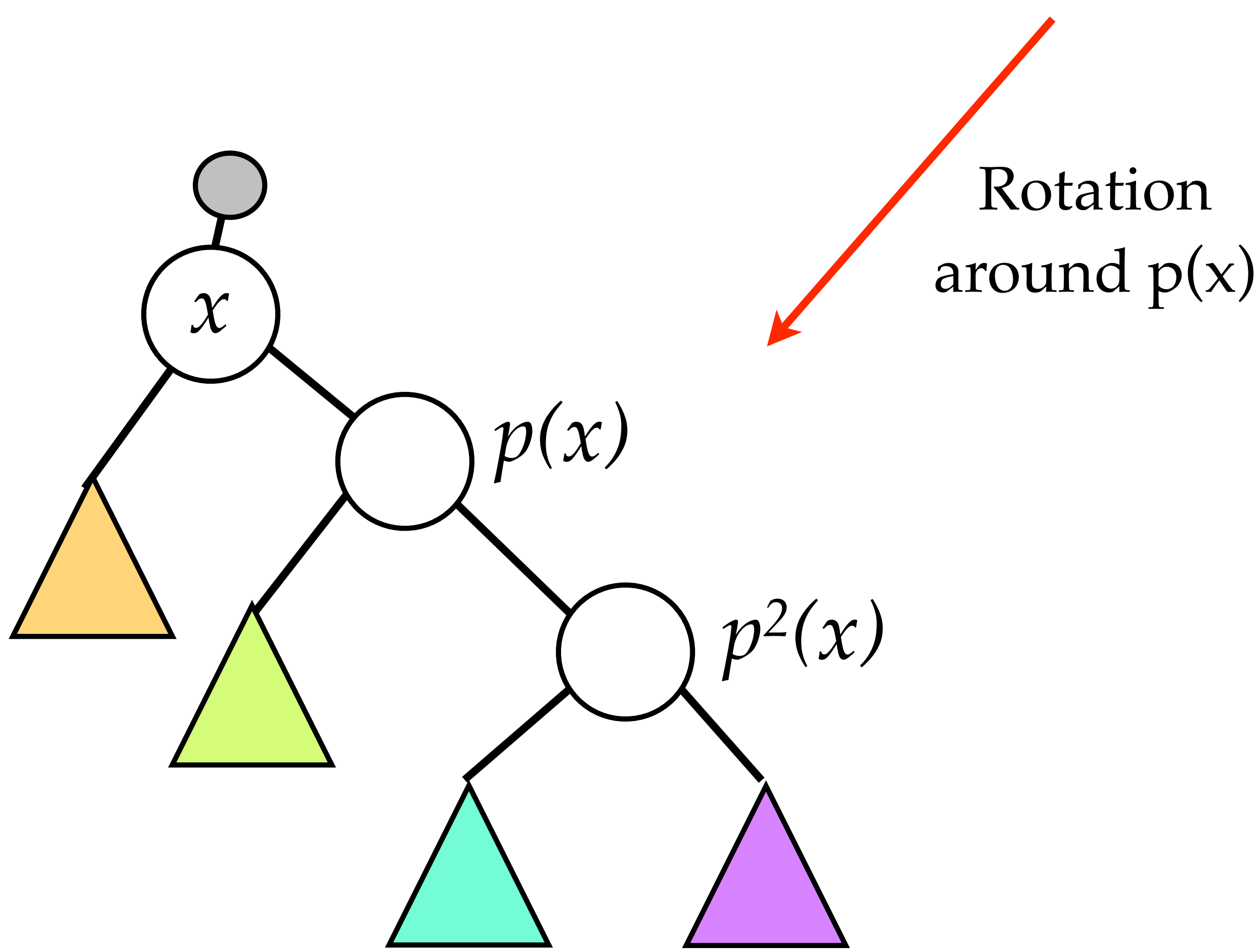
Case 2: zigzag (right,left):



(Just like the right, left case of AVL trees: double rotation)

Case 3: zigzig (left, left):



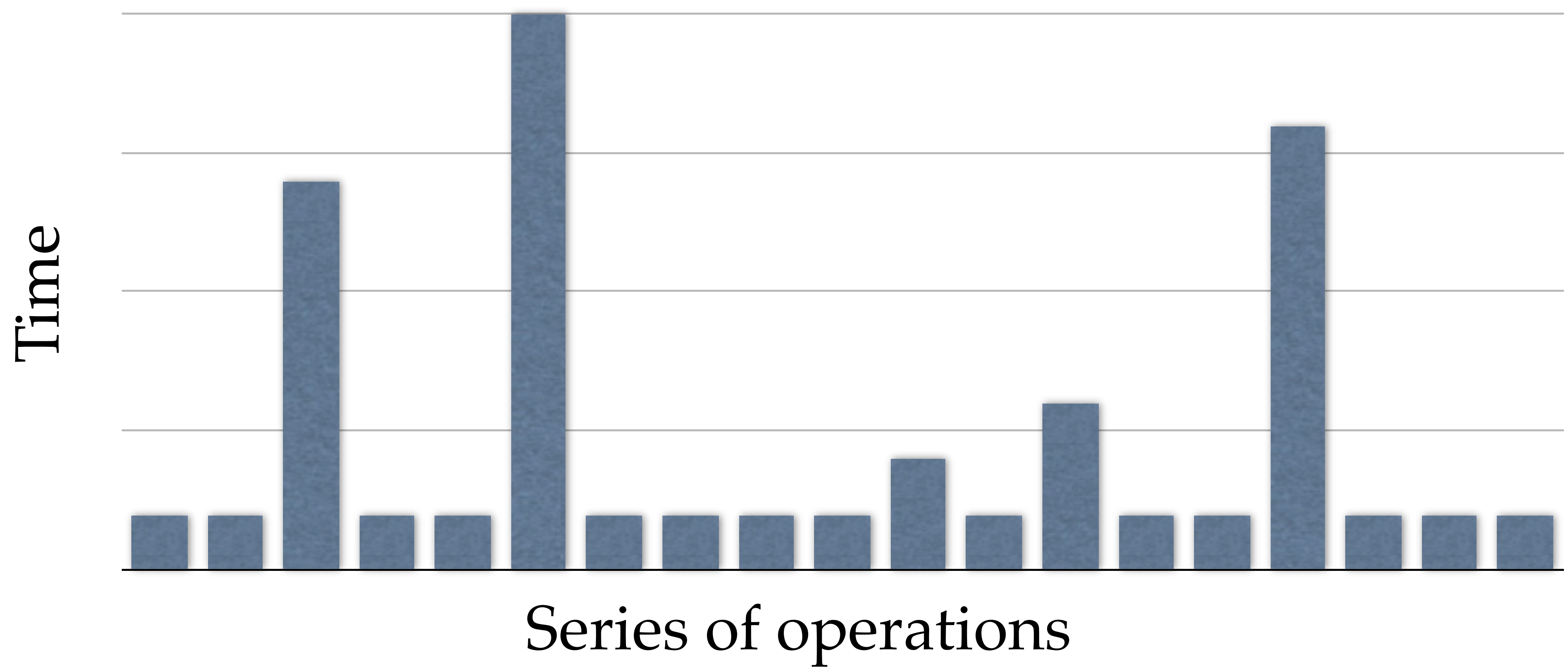


This one is different than with AVL trees: AVL would do only one rotation

Splay Notes

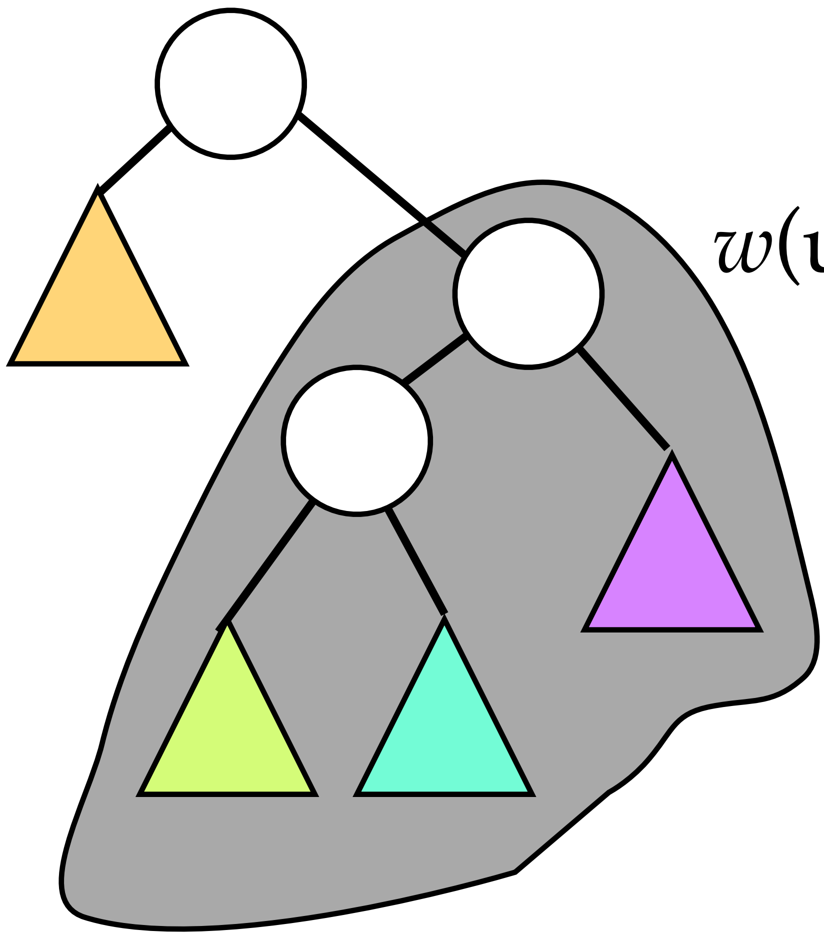
- Might make tree less balanced
- Might make tree taller
- So, how can they be good?

Amortized Analysis – Concept



- Some operations will be costly, some will be cheap
- Total area of *m* bars bounded by some function f(*m*,*n*).
  - m = number of operations, n = number of elements
- E.g. if area = O(m log n), each operation takes O(log n) amortized time

Node Ranks & Money Invariant



w(u) := weight of u

:= # nodes in the subtree rooted at u

$rank(u) := [\log w(u)]$

[x] means floor(x)

Money Invariant: we will always keep $rank(u)$ dollars stored at every node.

Each rotation/double rotation costs $1.

O(1) amount of work

Also have to spend $ to maintain invariant.

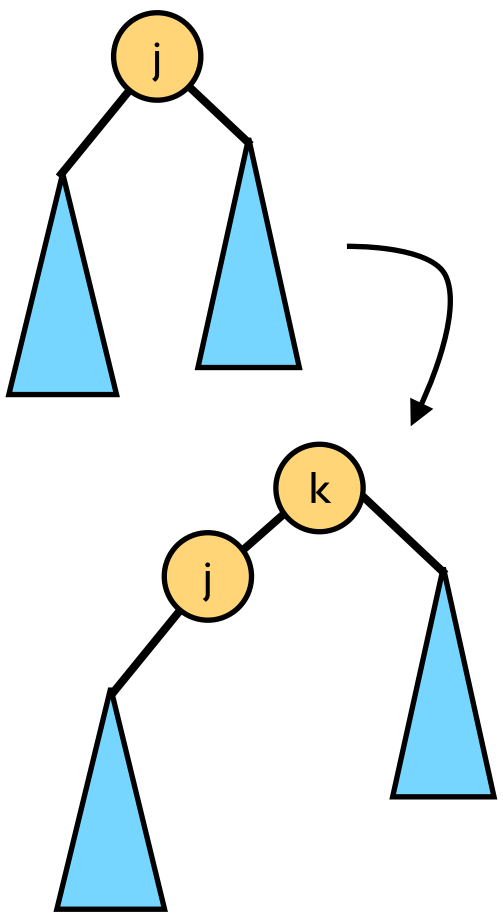
Idea:

Thm. It costs 3[log n] + 1 dollars to splay, keeping the money invariant

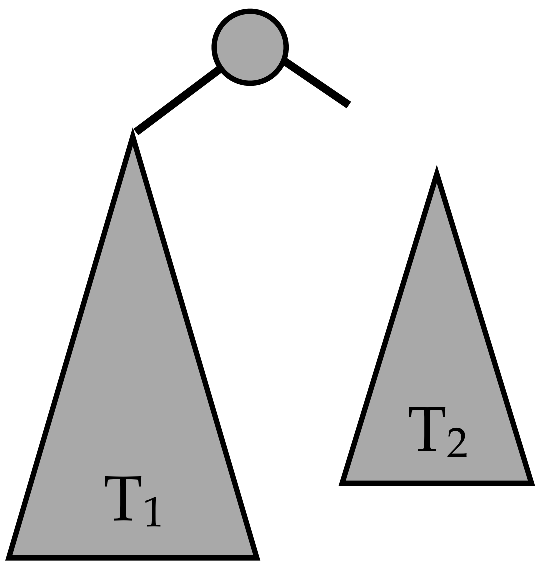
- So, for every splay, we're going to spend O(log n) new dollars.
- If we start with an empty tree, after m splay operations, we'll have spent m(3[log n] +1) dollars.
- The dollars pay for both:
  - the money invariant
  - cost of all the rotations (time)
- So, total time for m splay operations is O(m log n).

Additional Cost of Insert & Concat

- Cost of *insert* & *concat* more than the cost of a splay because may have to add \$s to root to maintain invariant:



insert(T, k): *k* has *n* descendants, so need to put [log n] \$ on k



concat(T_1, T_2): root gets at most *n* new descendants from T_2, so need to put [log n] dollars on root.

Thm. It costs 3[log n] + 1 dollars to splay, keeping the money invariant.

Suppose a splay rotation at x costs the following:

case 1: $3(rank^1(x) - rank(x)) + 1$
case 2: $3(rank^1(x) - rank(x))$
case 3: $3(rank^1(x) - rank(x))$

Then cost of a whole splay =

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

Telescoping sum

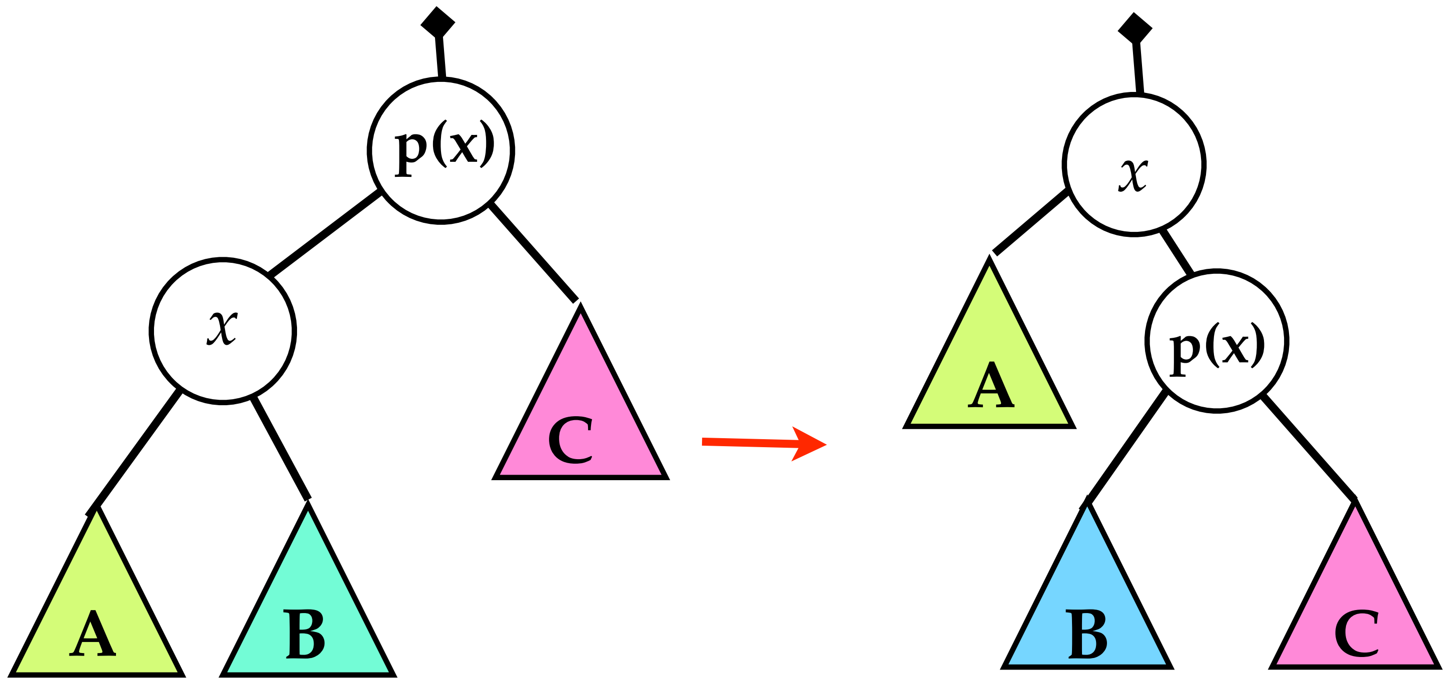
Then cost of a whole splay

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

$\leq 3(rank^k(x)) + 1$ *$rank^k(x)$ = rank of the original root*

$\leq 3[\log n] + 1$

case 1: $3(rank^1(x) - rank(x)) + 1$



+1 pays for the rotation

$rank^1(x) = rank(p(x))$

Extra \$ to keep the invariant is:

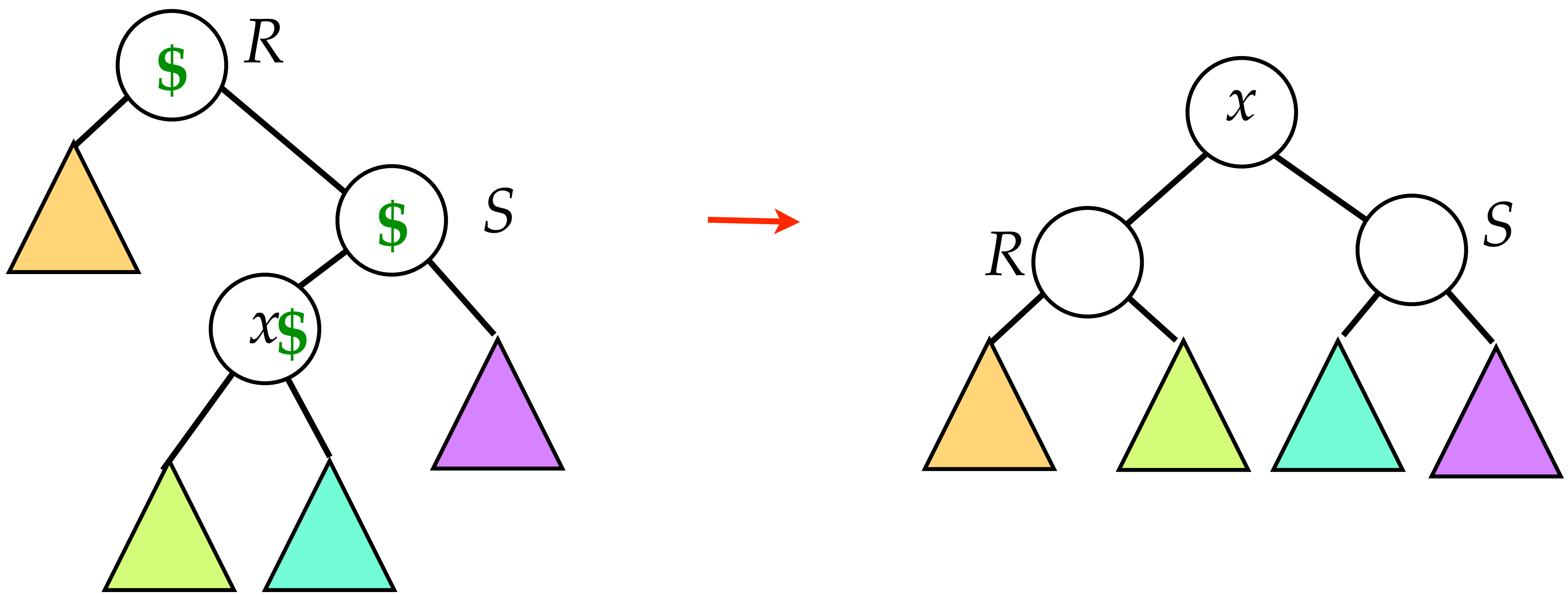
$rank^1(x) + rank^1(p(x)) - (rank(x) + rank(p(x)))$

\$ needed for x and p(x) — \$ already on x and p(x)

$= rank^1(p(x)) - rank(x)$

$\leq rank^1(x) - rank(x)$

case 2: $3(rank^1(x) - rank(x))$



$ needed to add: $rank^1(R) - rank(x)$

$\leq rank^1(x) - rank(x)$

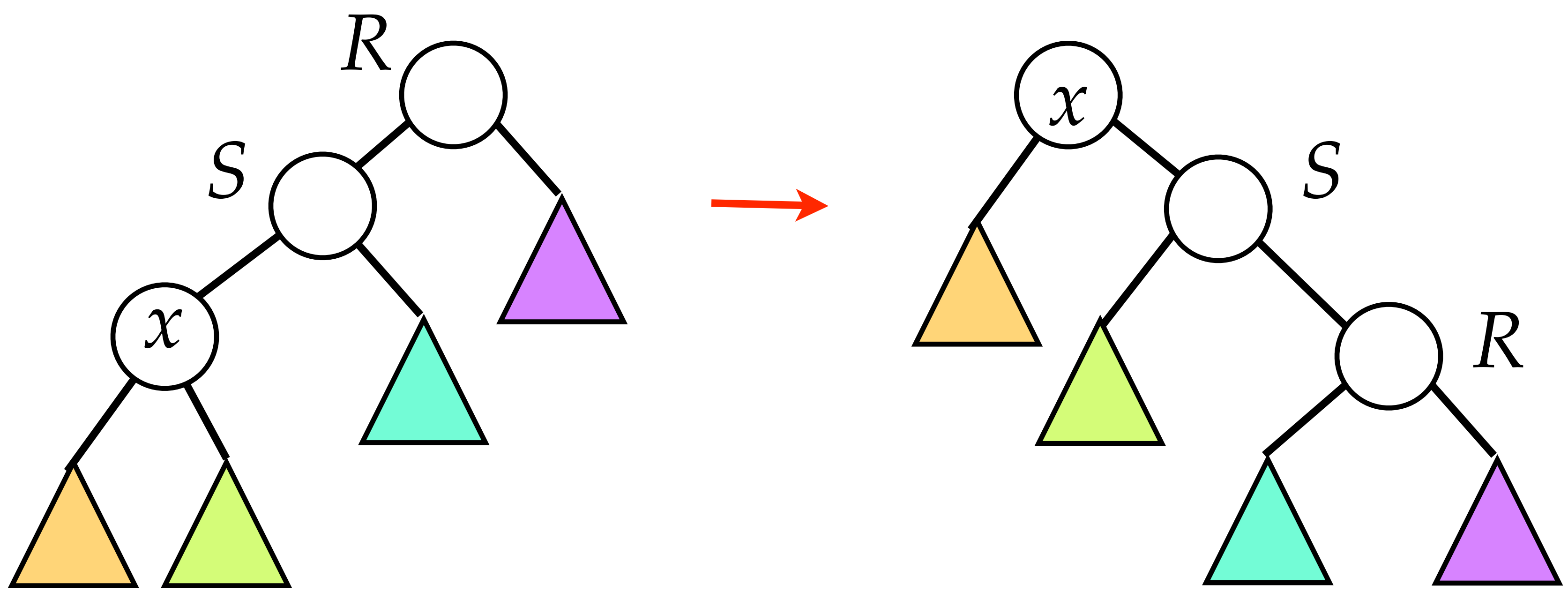
If $rank^1(x) - rank(x) > 0$, then we have at least $1 to pay for the rotations.

Otherwise $r^1(x) = r(x) = r(R) = r(S)$

Also, $r^1(R) < r^1(x)$ or $r^1(S) < r^1(x)$

So, $r^1(R) < r(x)$ or $r^1(S) < r(S)$

case 3: $3(rank^1(x) - rank(x))$



$ needed to add:

$r^1(x) + r^1(S) + r^1(R) - (r(x) + r(S) + r(R))$

$ needed for moved nodes $ already on moved nodes

$r^1(S) + r^1(R) - (r(x) + r(S))$ $r^1(x) = r(R)$

$\leq 2(r^1(x) - r(x))$ $r^1(R) \leq r^1(S) \leq r^1(x)$

$r(x) \leq r(S)$