COS 423 Lecture 5
Self-Adjusting Search trees

Balanced trees minimize worst-case access time to within a constant factor, but what if accesses are not uniform?

Access locality:

- Different but fixed access probabilities
- Spatial locality: frequent accesses near certain positions: fixed or moving fingers, e.g. first, last
- Time locality: working set

Ways to exploit locality:

- **Custom-built** data structure:
  - Optimum search tree (given fixed access probabilities)
  - Finger search tree (heterogenous or homogeneous)
  - “Working set” tree?
- **Self-adjusting** data structure

Self-adjusting search tree: during or after each access, modify the tree (to reduce overall cost of accesses and updates).

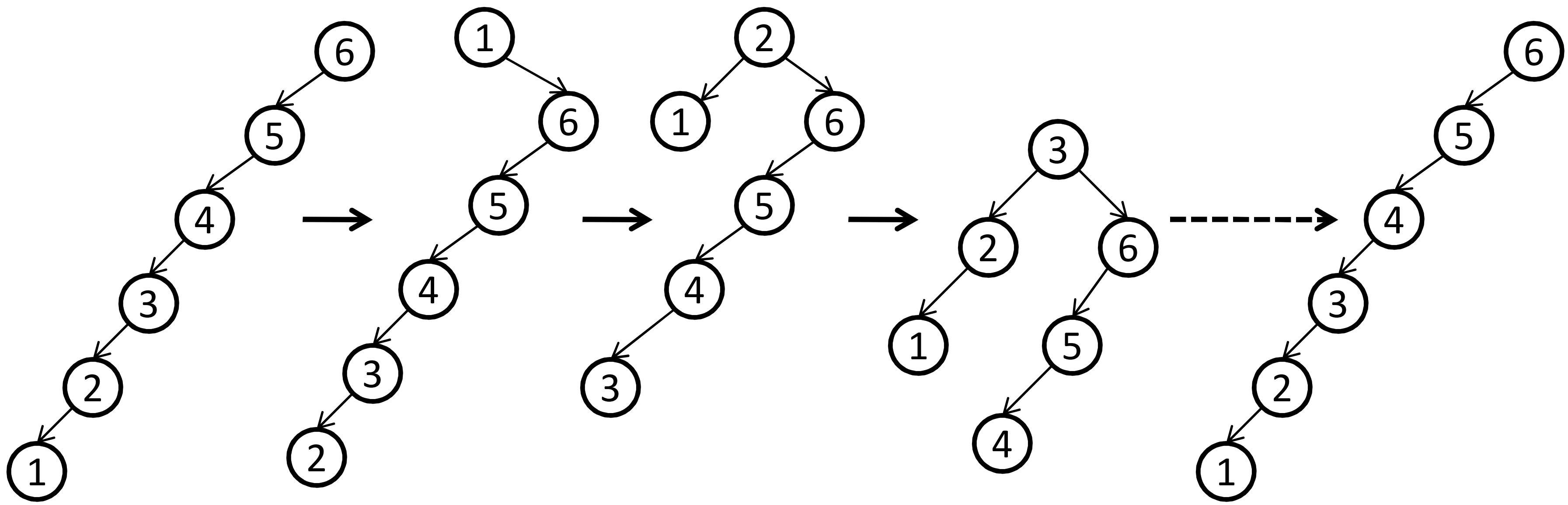
List analogy:

swap: rotation

move to front: move to root

First try: after an access or insert, move the accessed or inserted node to the root by bottom-up rotations along the access path.

Bad example: sequential access



n accesses in sequential order cost ~$n^2/2$,
and self-reproducing!

Splay Trees (Sleator and Tarjan 1983)

Splay: to spread out. *splay*(x) moves x to root via rotations, two at a time. Rotation order is generally bottom-up, but if the current node and its parent are both left or both right children, the top rotation is done first.

splay(x): **while** $p(x) \neq$ null **do**

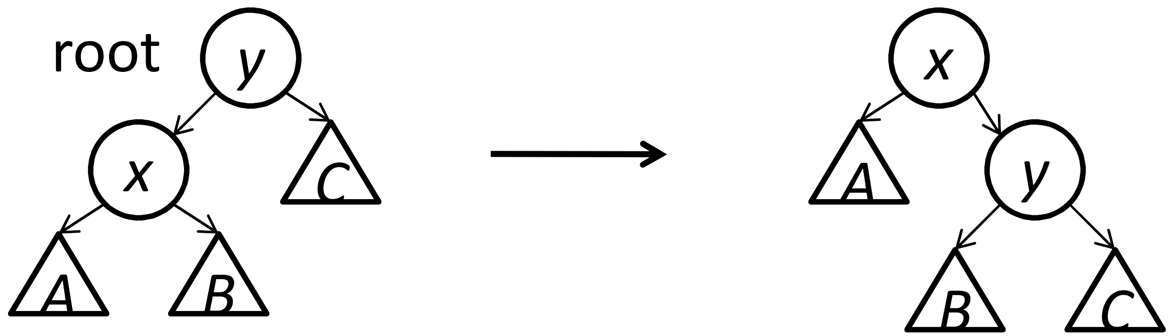
if $p(p(x))$ = null then *rotate*(x) [*zig*]

else if x is *left* **and** $p(x)$ is *right* **or** x is *right* **and**

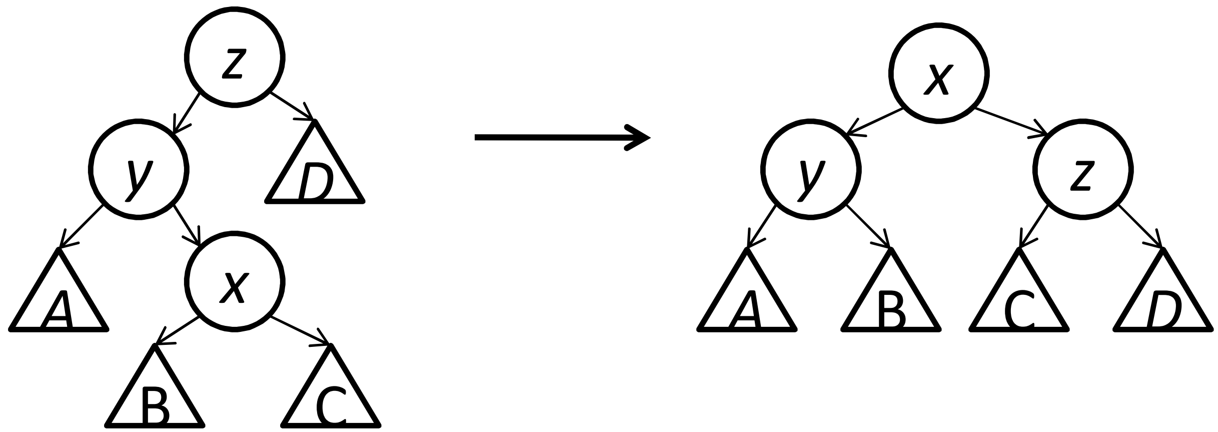
$p(x)$ is *left* then {*rotate*(x), *rotate*(x)} [*zig-zag*]

else {*rotate*($p(x)$, *rotate*(x)} [*zig-zig*]

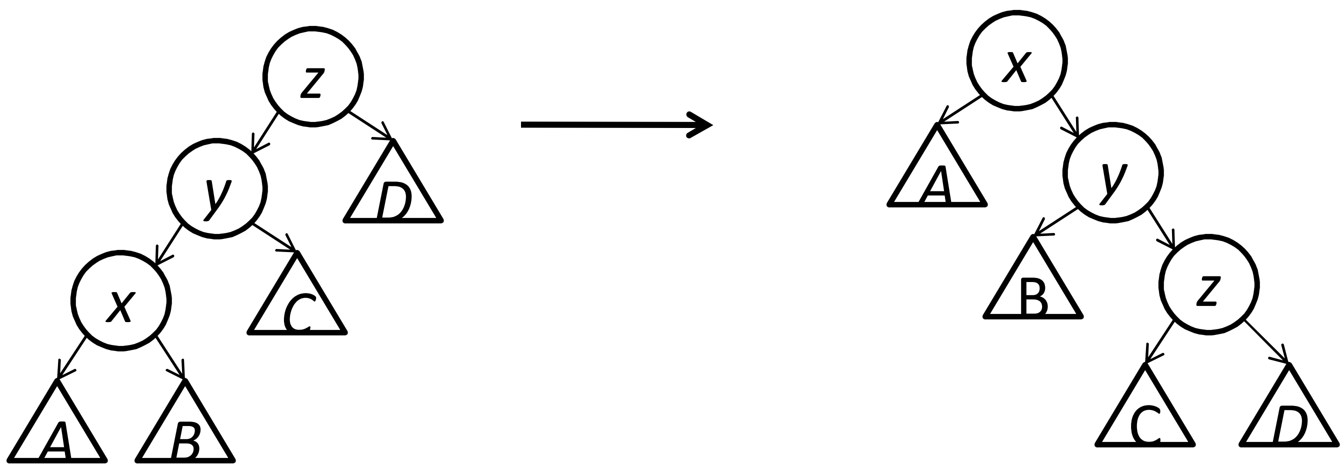
zig



zig-zag



zig-zig



Operations on splay trees

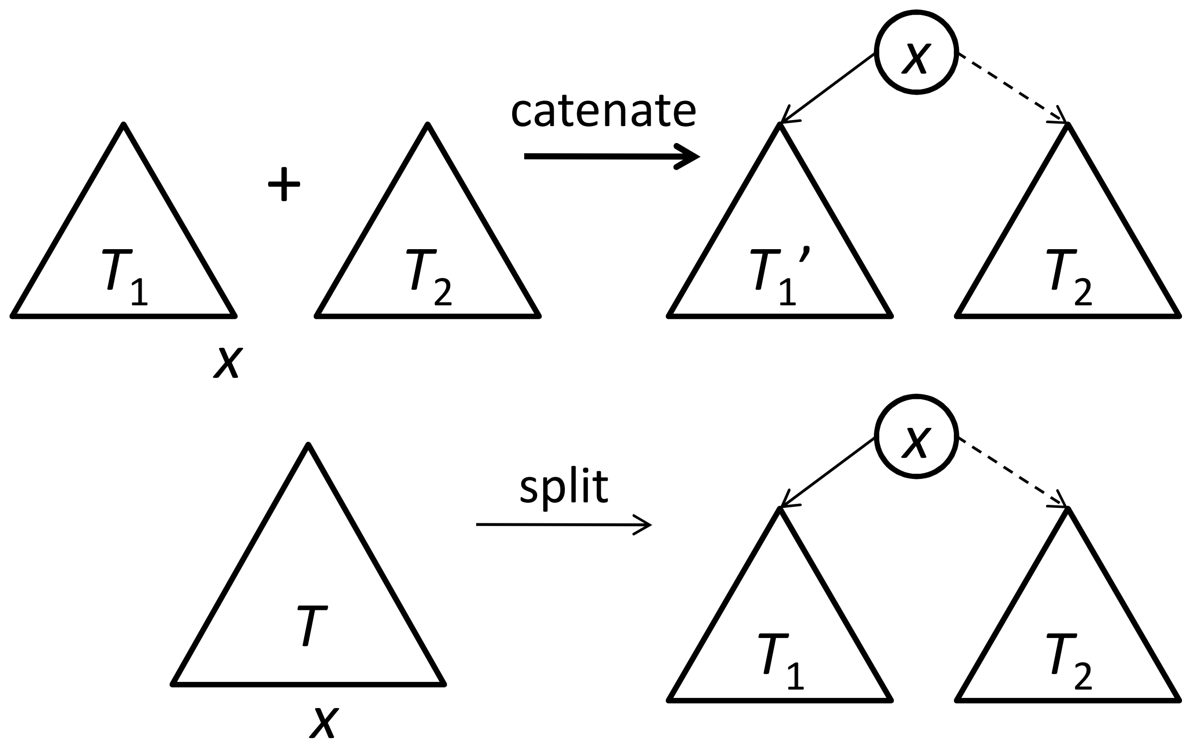
Access x: follow search path to x, then *splay*(x). Moves x to root, takes time O($d(x)$ + 1), including $d(x)$ rotations.

Insert x: follow search path to null, replace by x, *splay*(x).

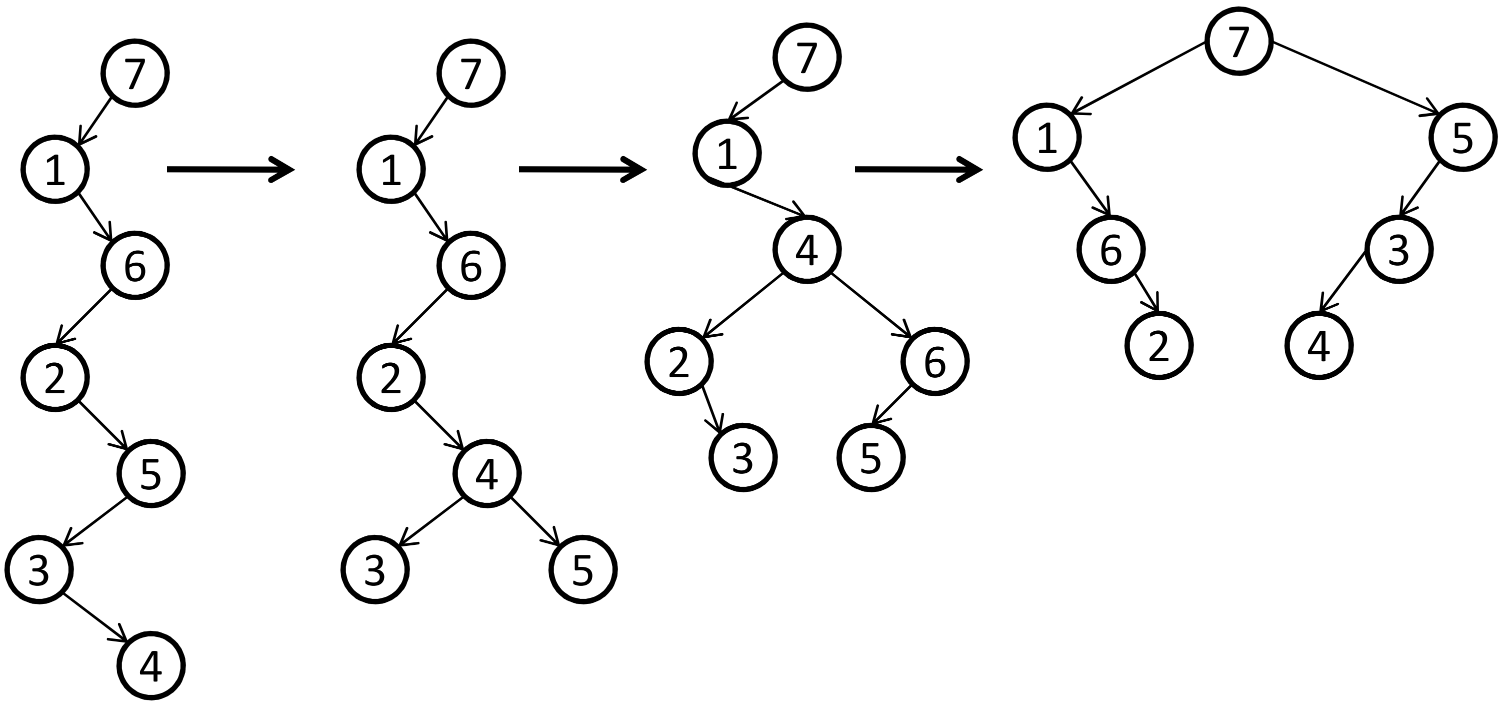
Delete x: follow search path to x, swap with successor if binary, delete x, splay at old parent.

Catenate(T_1, T_2)(all items in T_1 < all items in T_2):
splay at last node x in T_1; *right*(x) ← *root*(T_2).

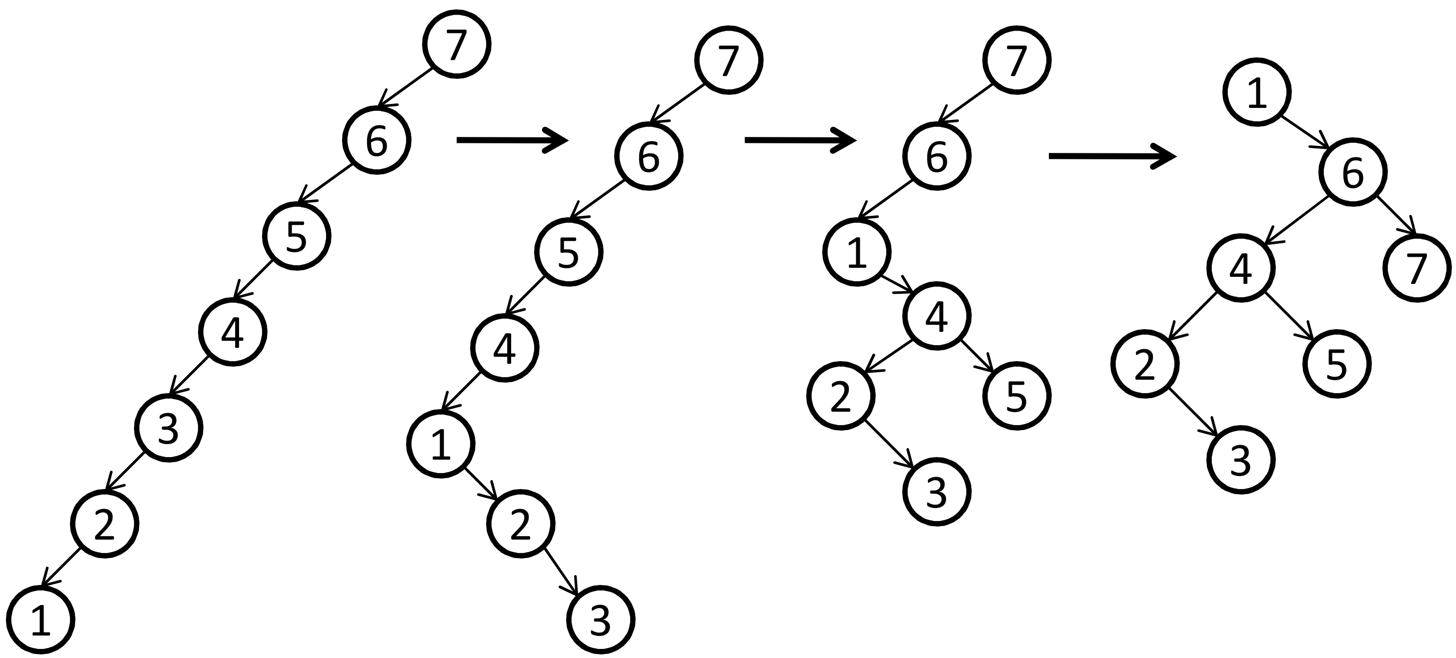
Split(T, x): *splay*(x); detach *right*(x) = root of tree containing all items > x.



Splay: pure zig-zag



Splay: pure zig-zig



Analysis of splaying

Let the *cost* of *splay*(x) be $d(x) + 1$ = #rots + 1.

Assign each item x a positive *weight* $w(x)$. The *total weight* $W(x)$ of x is the sum of the weights of all items in the subtree of x, including x. E.g. $w(x) = 1 \rightarrow W(x) = s(x)$.

$$\Phi(x) = \lg W(x) \qquad\qquad \Phi(T) = \Sigma\Phi(x)$$

If $w(x) = 1$, $0 \leq \Phi(T) \leq n\lg n$.

If $w(x) \geq 1$, $\Phi(T) \geq 0$.

Access Lemma: The amortized cost of *splay*(x) is $\leq 3\Delta\Phi(x) + 2$.

Useful inequality:

$$0 \le (a - b)^2 = a^2 - 2ab + b^2 \rightarrow 2ab \le a^2 + b^2$$

$$\rightarrow 4ab \le a^2 + 2ab + b^2 = (a + b)^2$$

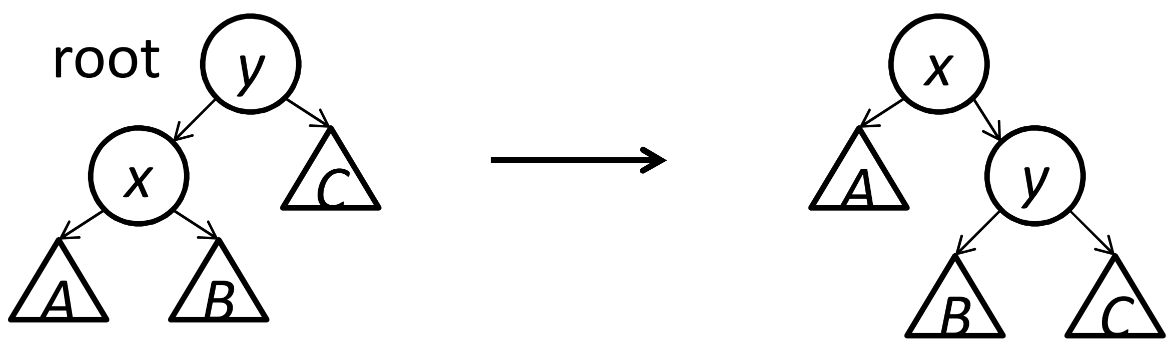
$$\rightarrow \lg a + \lg b \le 2\lg(a + b) - 2 \qquad (*)$$

Proof of access lemma: Case analysis of splay steps.

zig: actual cost = 1

$$\Delta\Phi(T) = \Phi'(x) + \Phi'(y) - \Phi(x) - \Phi(y)$$
$$= \Phi'(y) - \Phi(x) \le \Phi'(x) - \Phi(x)$$
$$= \Delta\Phi(\mathrm{x}) \le 3\Delta\Phi(\mathrm{x})$$

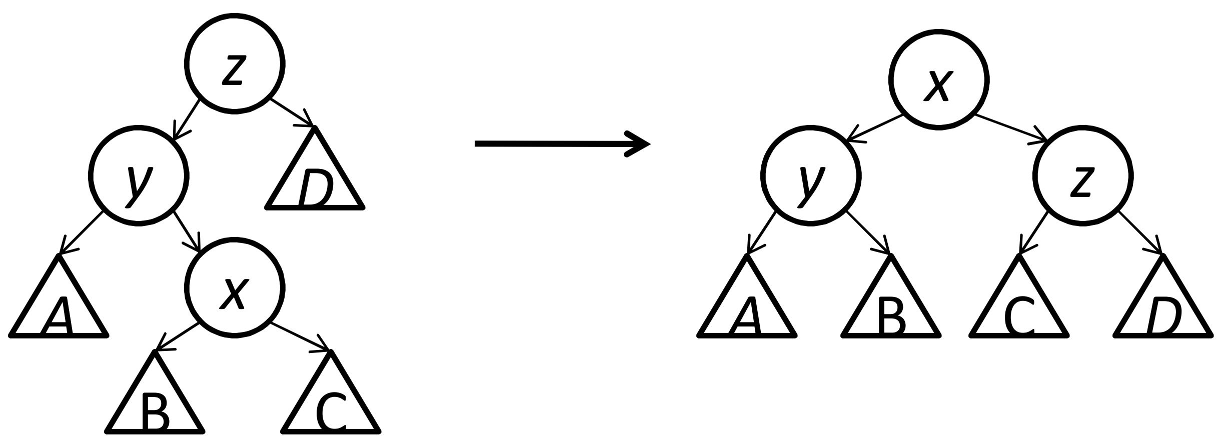
$\rightarrow$ amortized cost $\le 3\Delta\Phi(\mathrm{x}) + 1$



zig-zag: actual cost = 2

$$\begin{aligned}\Delta\Phi(T) &= \Phi'(y) + \Phi'(z) - \Phi(x) - \Phi(y) \\ &\le 2\Phi'(x) - 2 - 2\Phi(x) \text{ by } (*) \\ &\le 2\Delta\Phi(x) - 2\end{aligned}$$

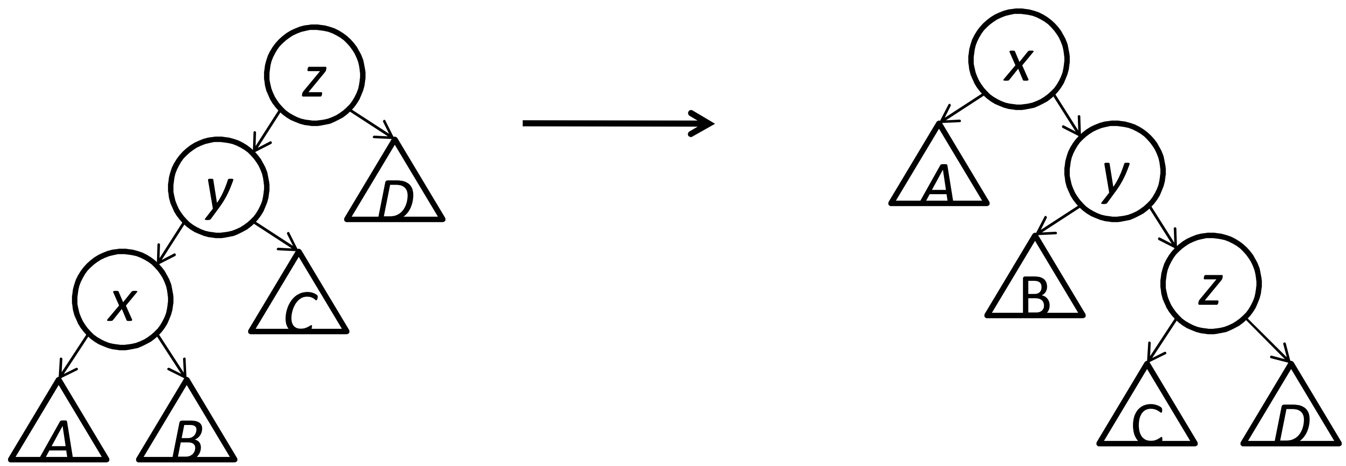
→ amortized cost ≤ $2\Delta\Phi(x) \le 3\Delta\Phi(x)$



zig-zig: actual cost = 2

$$\Delta\Phi(T) = \Phi'(y) + \Phi'(z) - \Phi(x) - \Phi(y)$$
$$= \Phi'(y) + \Phi'(z) + \Phi(x) - 2\Phi(x) - \Phi(y)$$
$$\leq \Phi'(x) + 2\Phi'(x) - 2 - 3\Phi(x) \text{ by } (*)$$
$$= 3\Delta\Phi(x) - 2$$

$\rightarrow$ amortized cost $\leq 3\Delta\Phi(x)$



Summing over all splay steps gives the access lemma: the amortized cost of *splay*(x) is $\leq 3\Delta\Phi(x) + 2$.

Applications of the access lemma

Balance Theorem: Starting from an empty tree, an arbitrary sequence of accesses, insertions, and deletions takes $O(1 + \lg n)$ amortized time per operation.

Proof: Choose $w(x) = 1$. Insertion of leaf x (without splaying) increases $\Phi(T)$ by $O(\lg n)$. Amortized cost of *splay(x)* is $O(1 + \lg n)$. (**You verify**.)

Static optimality theorem: Start from an arbitrary tree and do an arbitrary sequence of m accesses, with each item accessed at least once. Let $f(x)$ = #accesses of x, The amortized time to access x is $O(1 + \lg(m/f(x)))$.

Proof: Choose $w(x) = f(x)$. $\lg f(x) \le \Phi(x) \le \lg m$

→ competitive with static optimum tree for given access frequencies

Working Set Theorem: Start with an arbitrary tree and do an arbitrary sequence of accesses, with each item accessed at least once. The amortized time to access x is $O(1 + \lg k(x))$, where $k(x)$ is the number of distinct items accessed since the last time x was accessed.

Proof: Assign weights 1, 1/4, 1/9, 1/16,…, $1/n^2$ in order by most recent access.

True but proof is long and complicated

Dynamic Finger Theorem (Cole 1990): Start from an empty tree and do an arbitrary sequence of insertions, deletions, and accesses. The amortized time for an operation is $O(1 + \lg t)$, where t is the number of nodes in symmetric order between the last node splayed and the current node splayed, inclusive.

Dynamic optimality conjecture

Splay trees are O(1)-competitive with the optimum off-line binary search tree algorithm.

Best known so far: Several more-complicated binary search trees are O(lglg*n*)-competitive.

Advantages of splay trees:

No balance information required.

Simple operations.

Take advantage of any exploitable pattern in the access sequence.

Disadvantage of splay trees:

Many rotations, even during accesses!