COS 423 Spring 2006

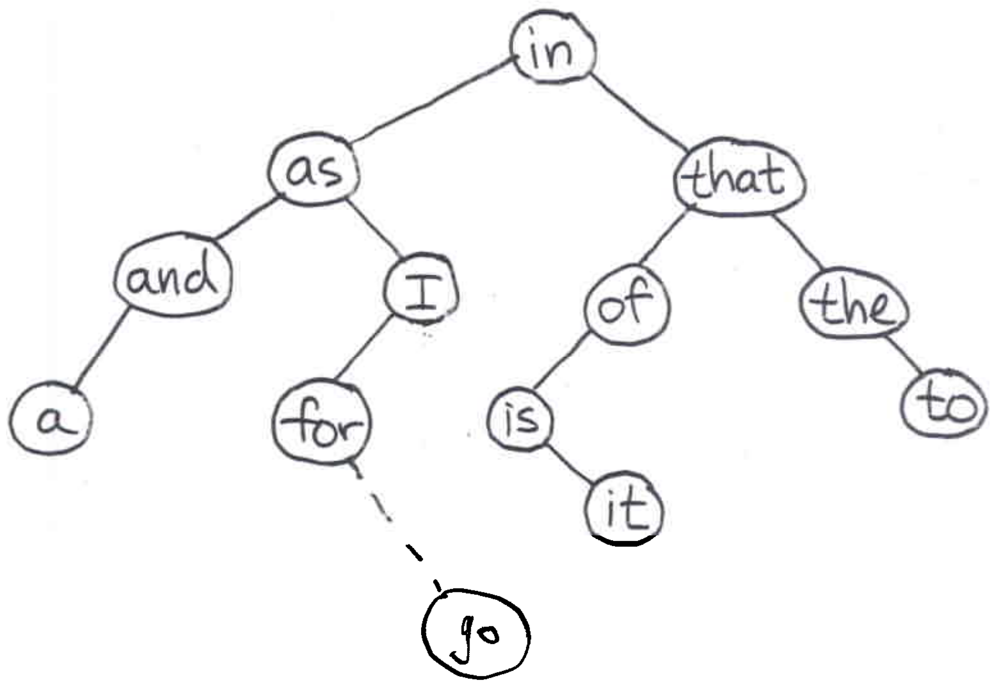
Binary Search Trees

Binary Search Trees

Binary Tree: A rooted tree, each node having a left and a right child, either or both missing.

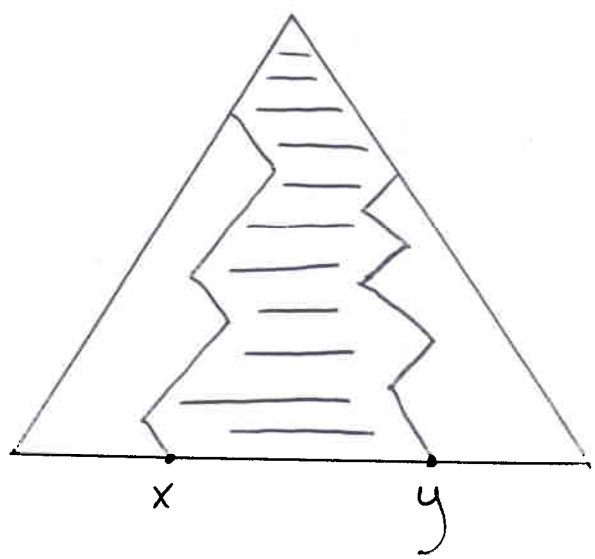
Binary Search Tree: Each node contains an item. Items are totally ordered and arranged in the tree in symmetric order: all items in left subtree are less, all items in right subtree are greater.

Binary search trees support access, insert, delete in O(depth) time.

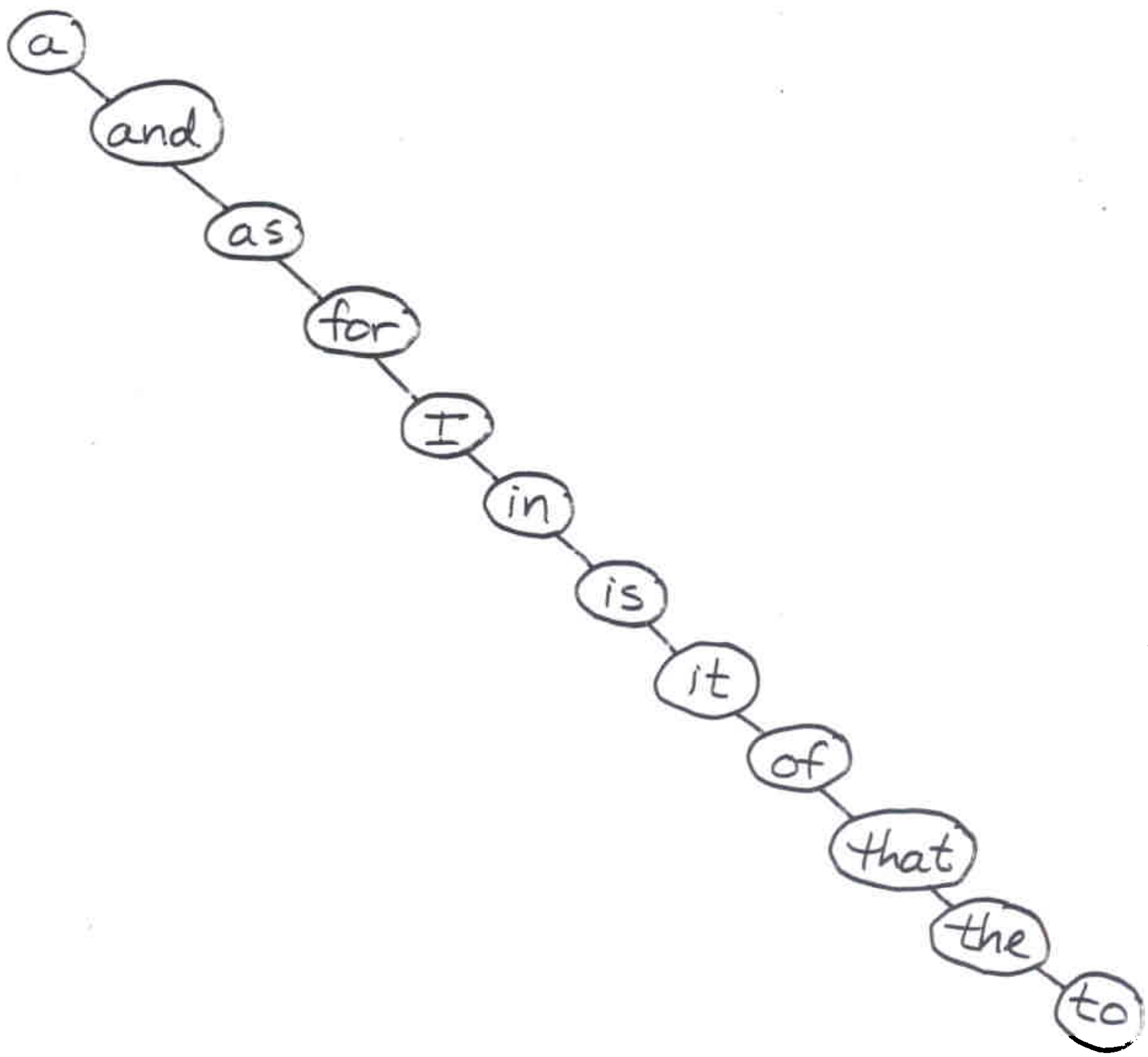

in
as
that
and
I
of
the
a
for
is
to
it
go

Search Trees Can Be Used For Range Queries

Report all entries between x and y:



Another binary search tree



How do we keep depth small?

Classical answer: Maintain a (local) balance condition.

Two properties:

(i) Implies $O(\log n)$ depth of an n-node tree.

(ii) Easily restorable after an update: $O(\log n)$ time by rebalancing along access path.

Since ~ 1962 many kinds of such

balanced search trees

have been discovered.

Classes of Balanced Trees

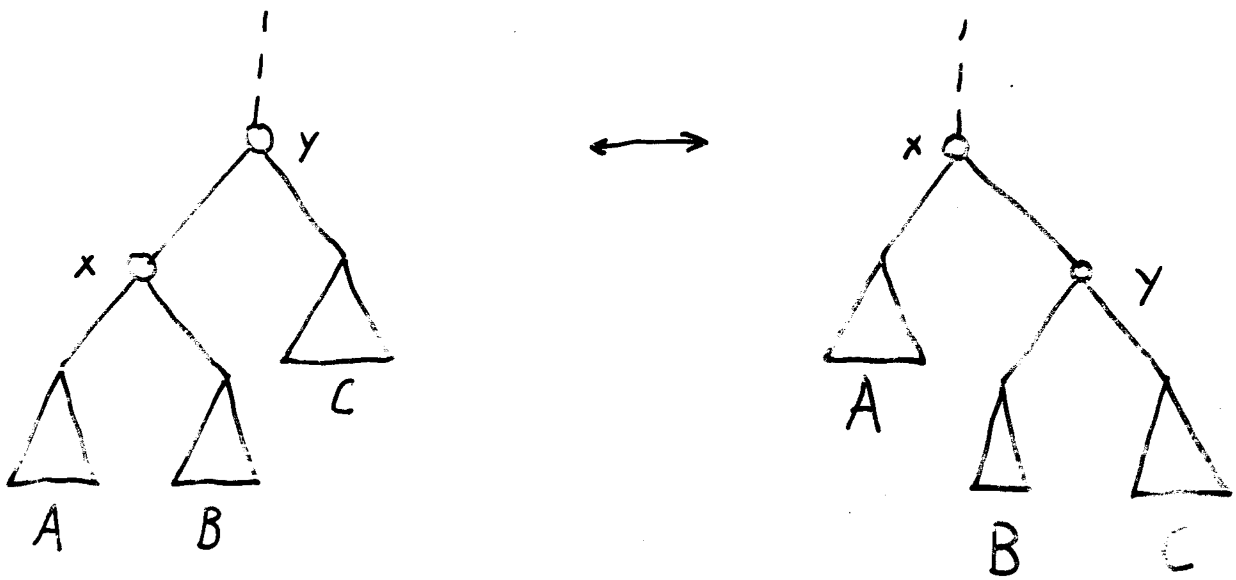
1. Height-balanced (AVL) trees
2. Weight-balanced (BB(α)) trees
3. 2,3 trees
4. B-trees
5. Brother trees — not binary
6. 2,4 trees
7. Symmetric binary B-trees — equivalent
8. Red-black trees
9. Half-balanced trees

etcetera

All achieve $O(\log n)$

access/insert/delete time

A Rotation



Changes depths of some nodes

Takes $O(1)$ time (3 pointer changes)

Preserves symmetric order

Red-Black Trees

1. Each node is either red or black.
2. The root and all missing nodes are black.
3. There are no two red nodes in a row.
4. All paths from the root to a missing node have the same number of black nodes.

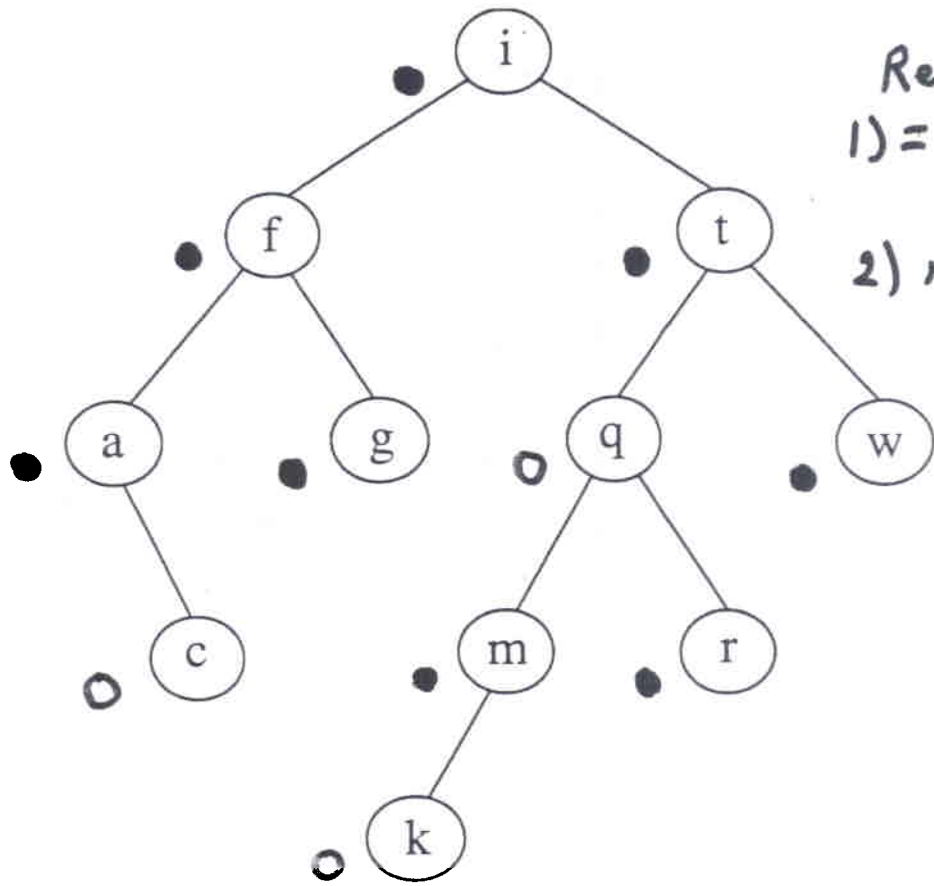
Equivalent to:

2,4 trees

Symmetric binary B-trees

Half-balanced trees

A Binary Search Tree

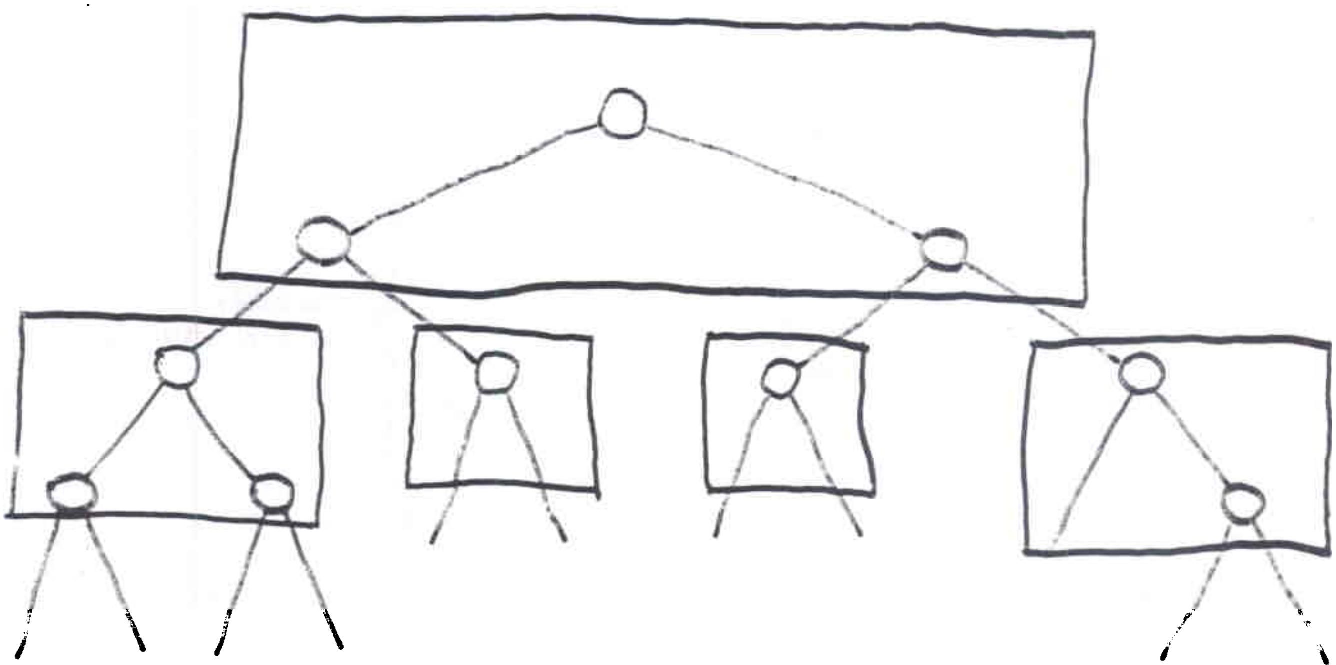


Red / Black:
1) = #s blacks on paths,
2) red nodes have black parents

Items in internal nodes, in symmetric order:
items in left subtree smaller,
items in right subtree larger.

Allows binary search for items
search time = 1+ depth.

A Red-Black Tree



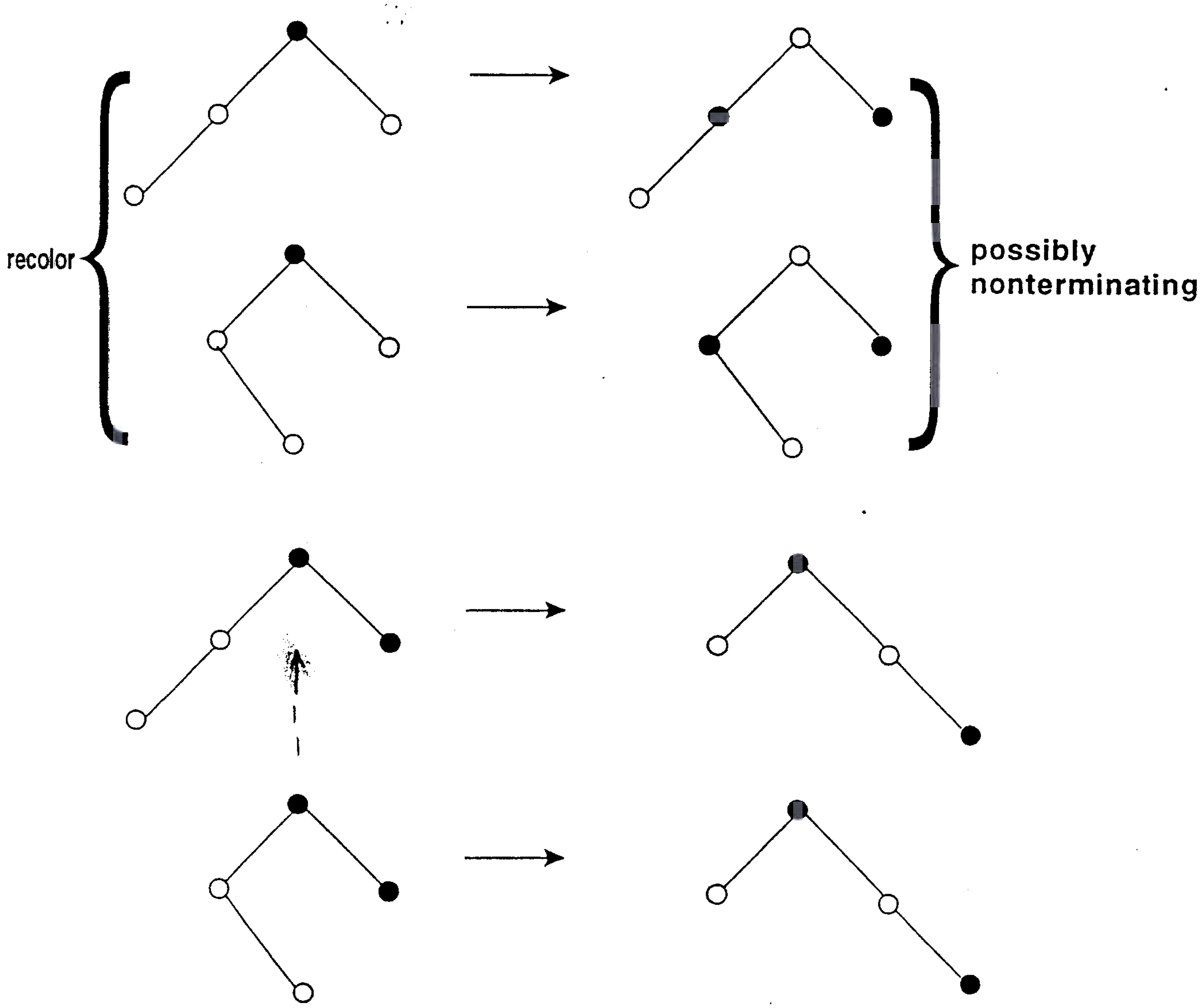
A 2,4 Tree

Red-black tree updates

● black

○ red

Insert ○ root ⟶ ●

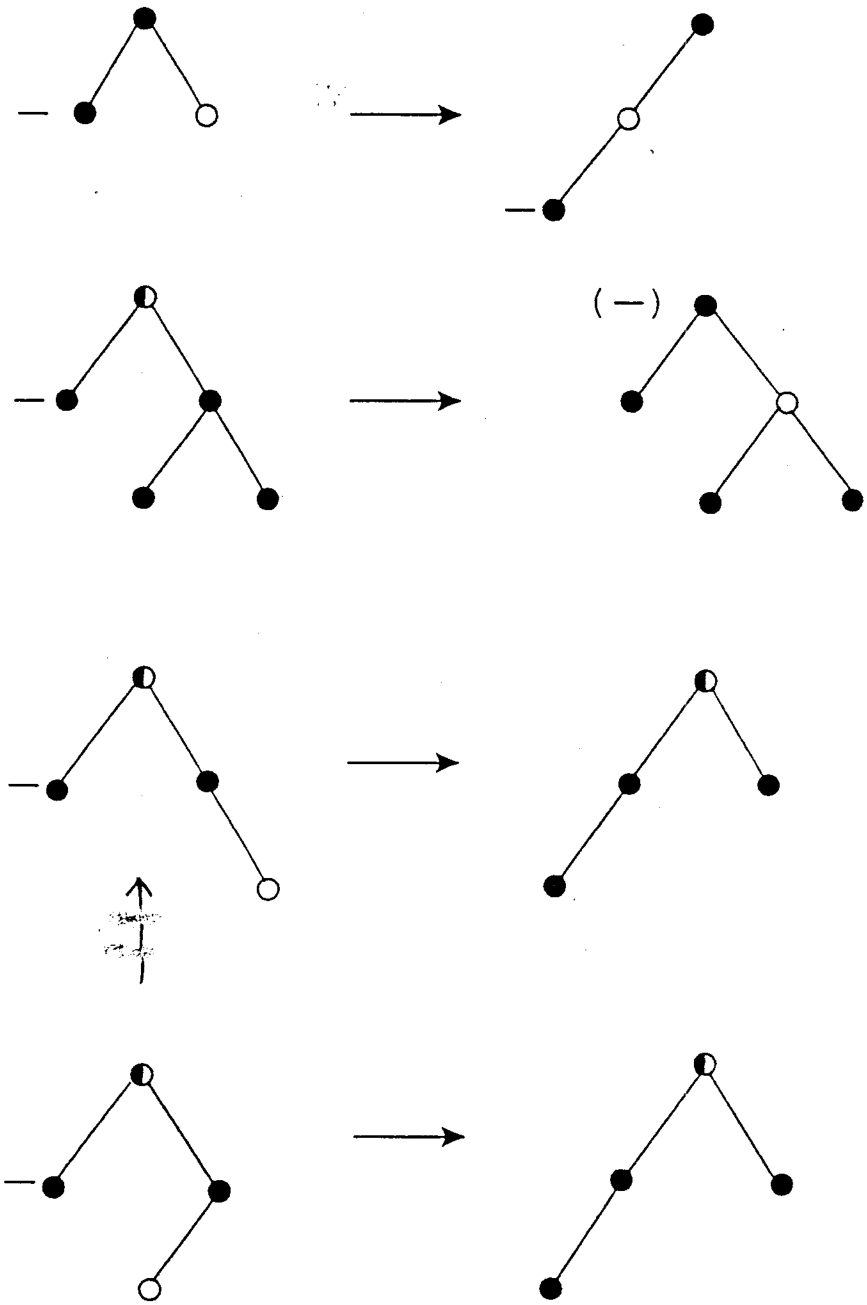


Delete — short node (all paths down lack one black node)

◐ red or black node (color preserved)

—● root ⟶ ●

—○ ⟶ ●

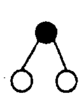
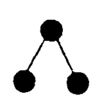


creates a terminating case

Nonterminating if orginal root is black

O(log n) recolorings; 0, 1, 2, or 3 rotations

O(1) amortized recoloring time for insert/delete:

Φ = 2 for , 1 for 

How long to process a sequence of searches?

If access frequencies are known in advance and initial tree is arbitrary but fixed, an optimum binary search tree (Knuth-style) minimizes the total search time.

What if access frequencies are not known in advance?

What if tree is allowed to change during the sequence?

Total time for a sequence of accesses

= total search time

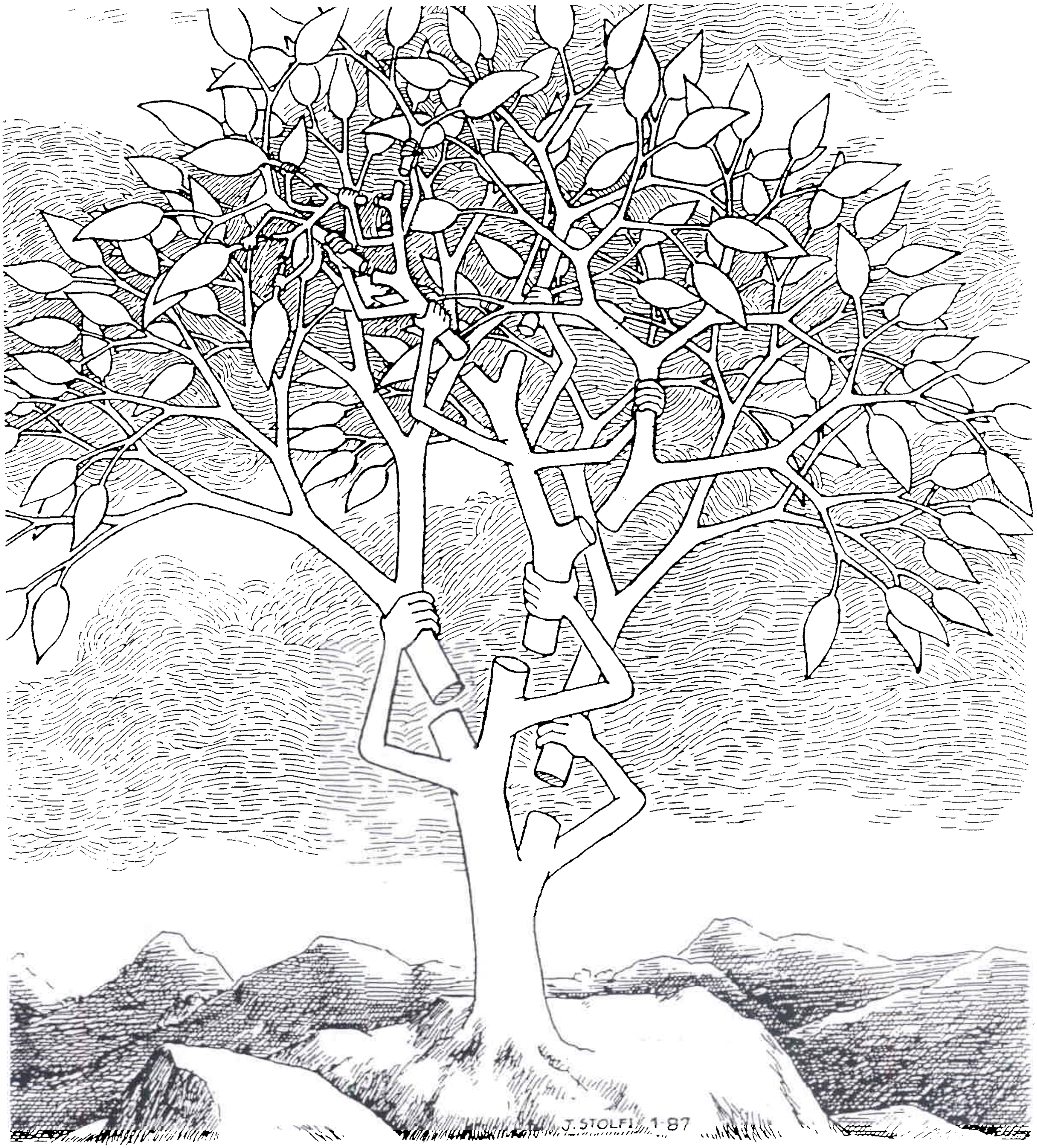
(sum of 1+ depth of accessed item, when accessed)

+ total number of rotations

(between searches arbitrary rotations can be done)

Goal: Compare the minimum-cost off-line strategy with (simple) on-line strategies.

Can an on-line strategy
(no future knowledge)
achieve a performance within a constant factor of that of the optimum off-line strategy
(access requests known in advance)?

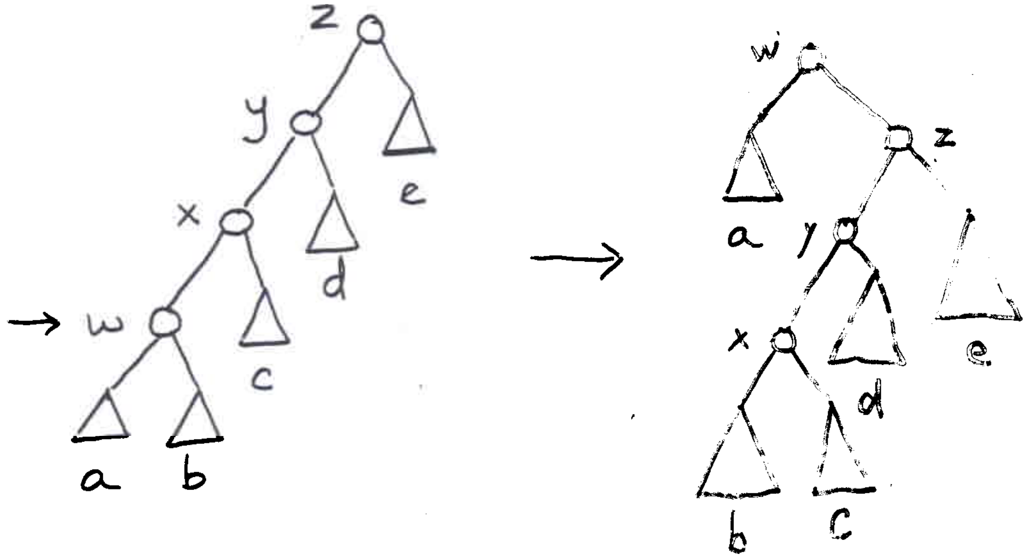


A Self-Adjusting Search Tree

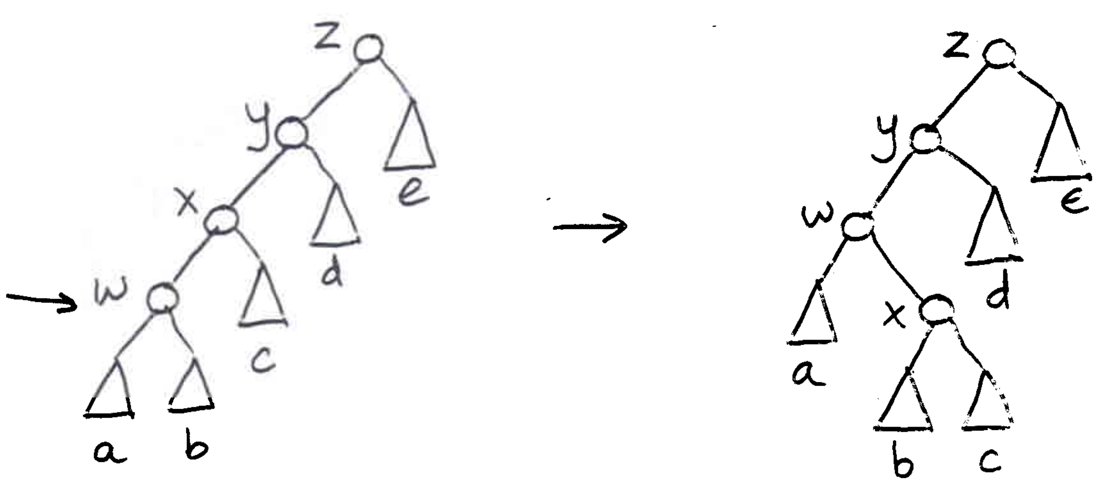
Previous Self-Adjusting Heuristics

(Allen and Munro, Bitner)

1. Move to root: do single rotations all along access path.



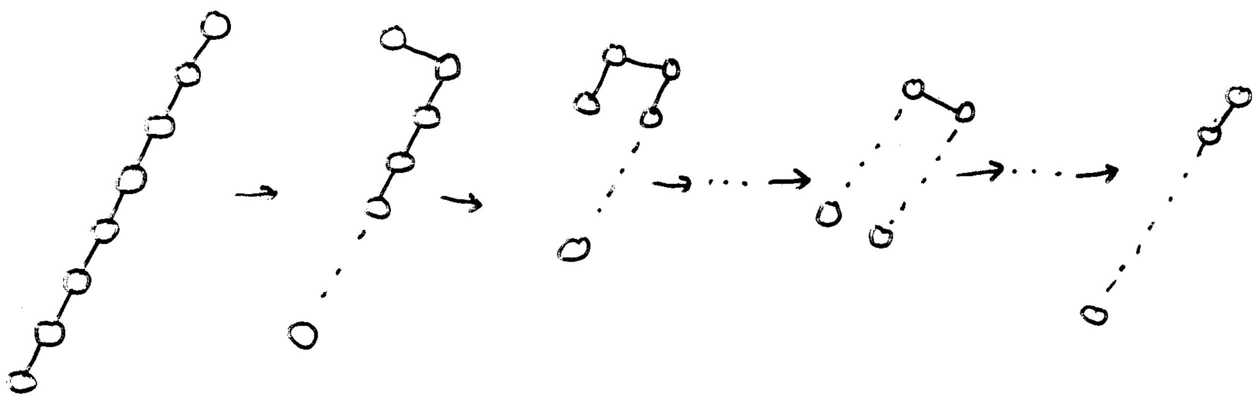
2. Single exchange: do one rotation at parent of accessed node.



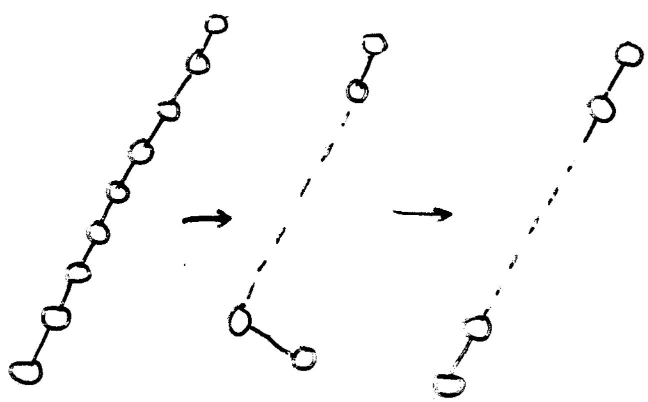
Both are o(n) per operation, even amortized.

Bad Examples

MTR



SE



Splaying: Sleator and Tarjan (1985)

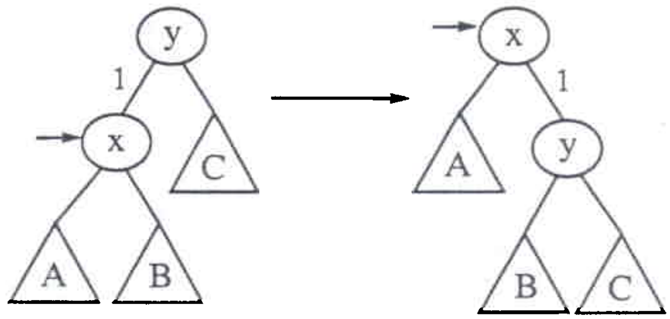
Rotate each edge along an access path.

Perform rotations in pairs, roughly bottom-up.

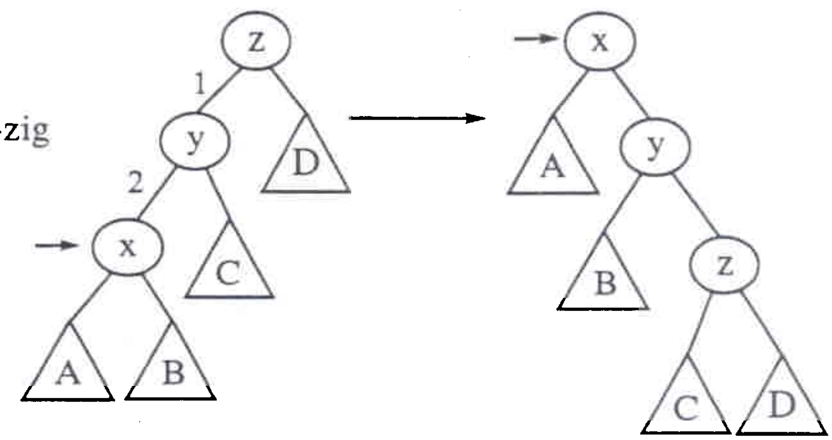
Access path is (roughly) halved, other nodes can move down, but only by a few steps.

Cases of Splaying

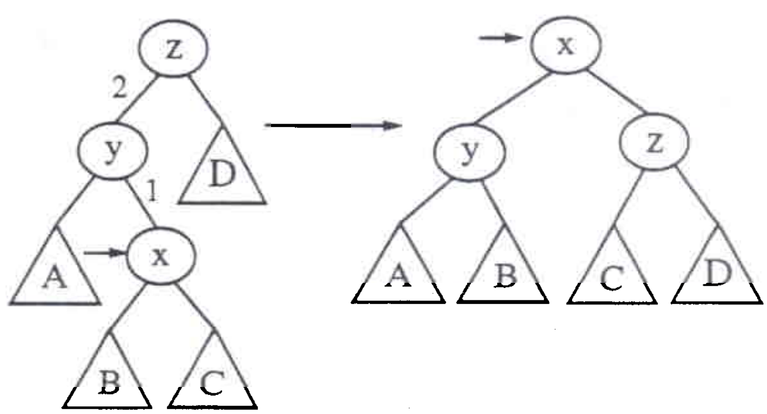
zig



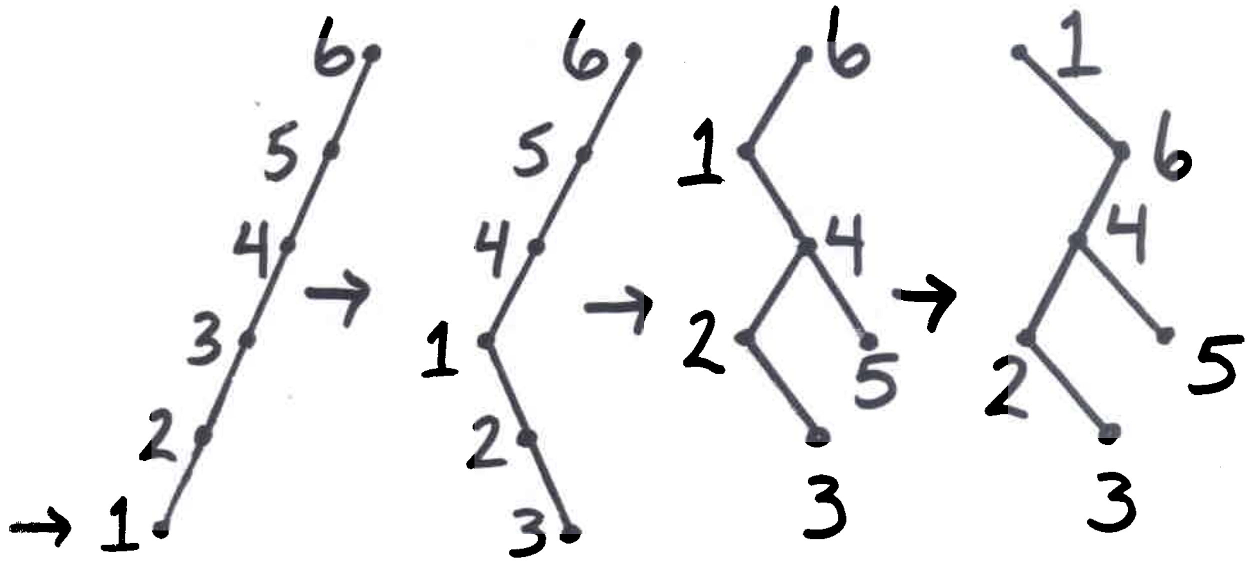
zig-zig

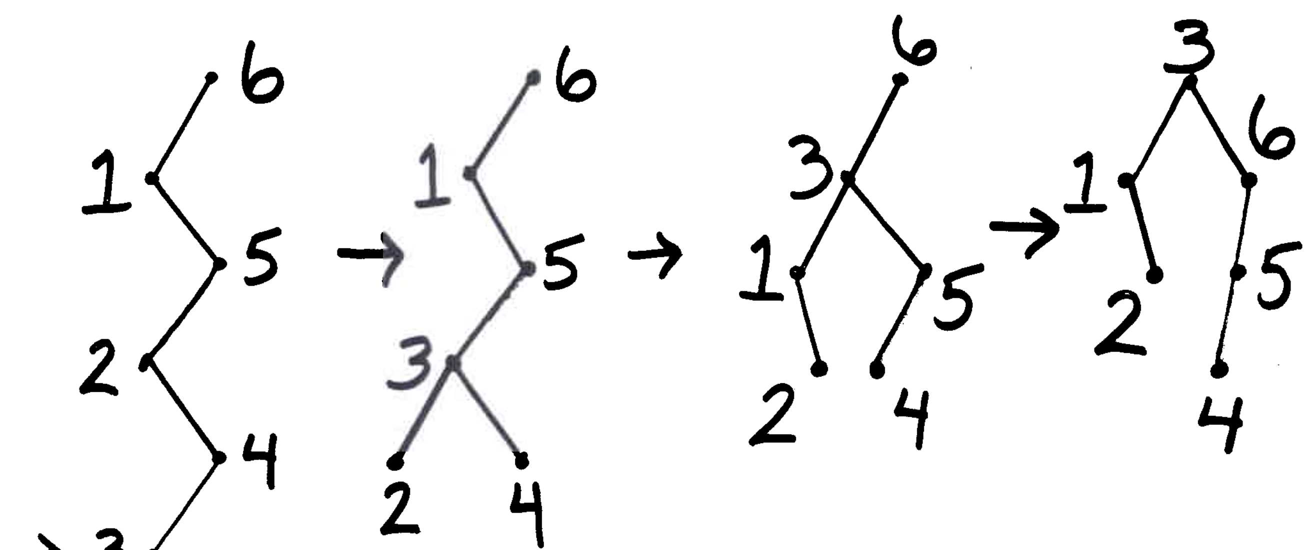


zig-zag

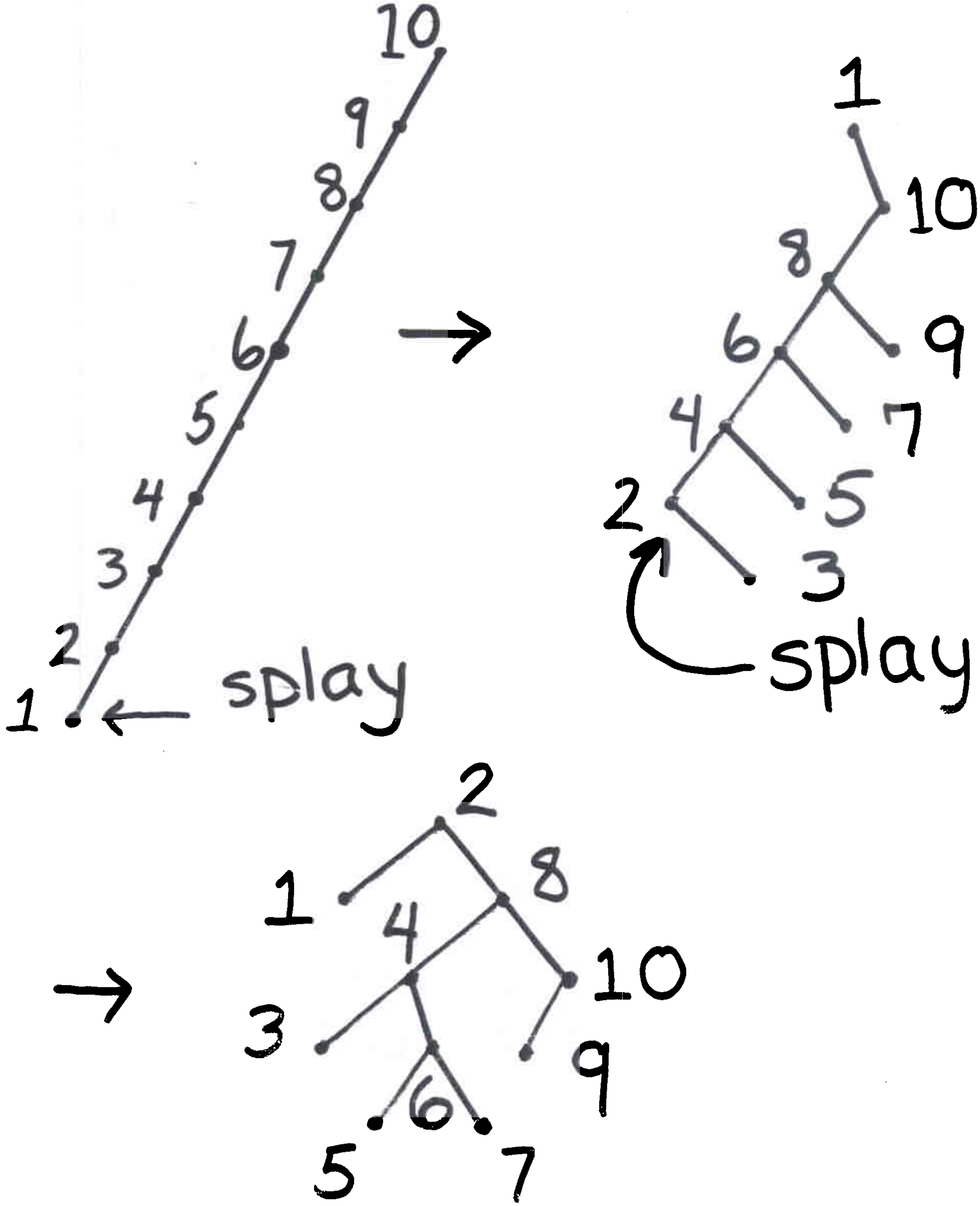


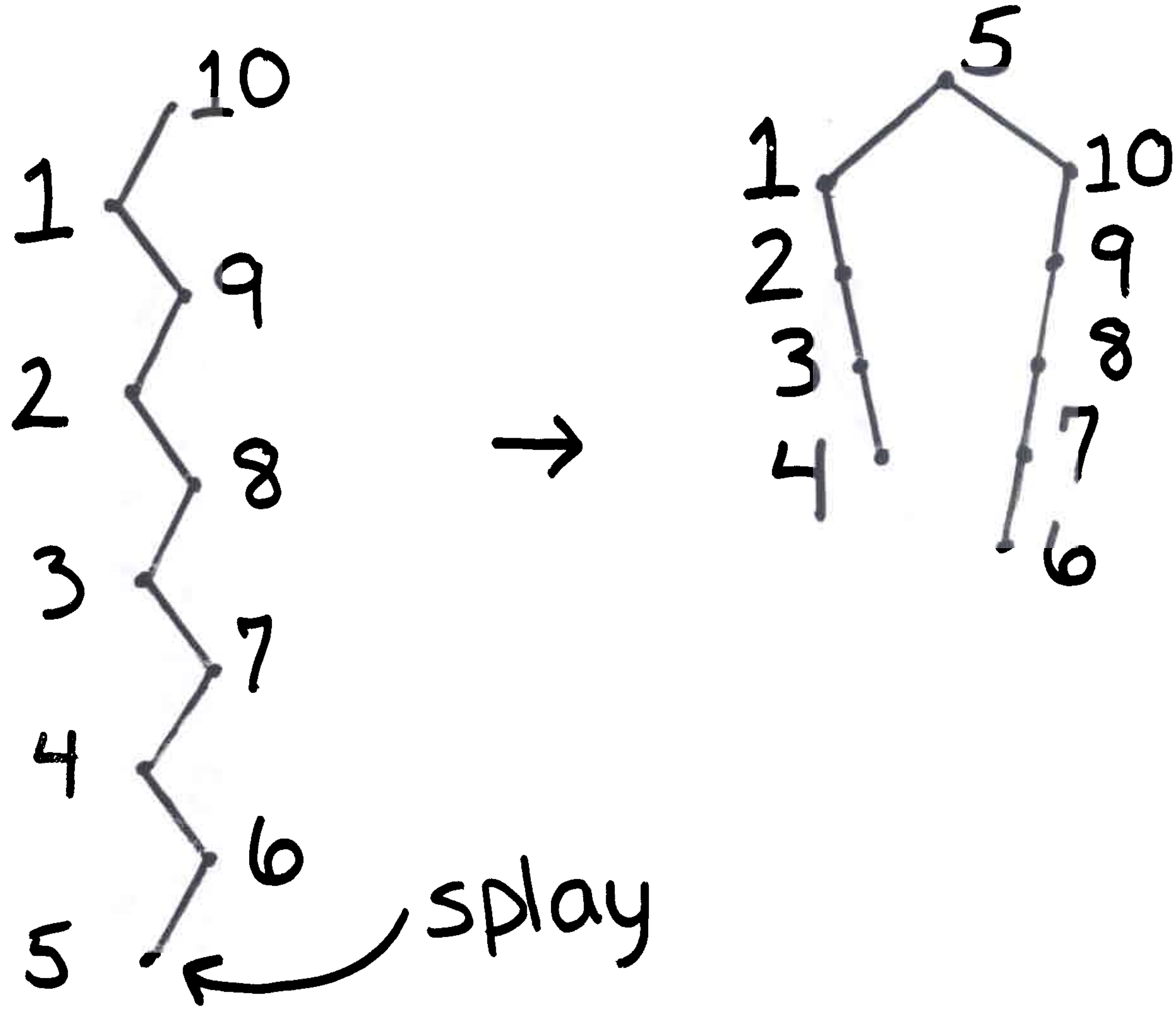
Step by Step Examples





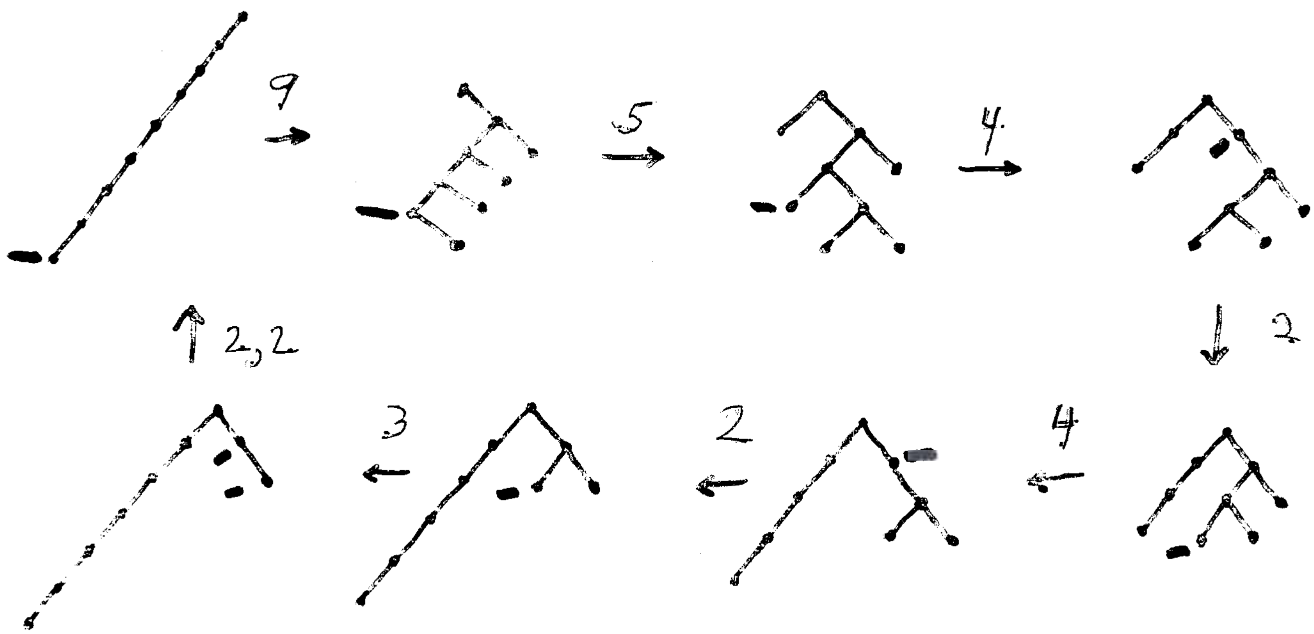
EXAMPLES



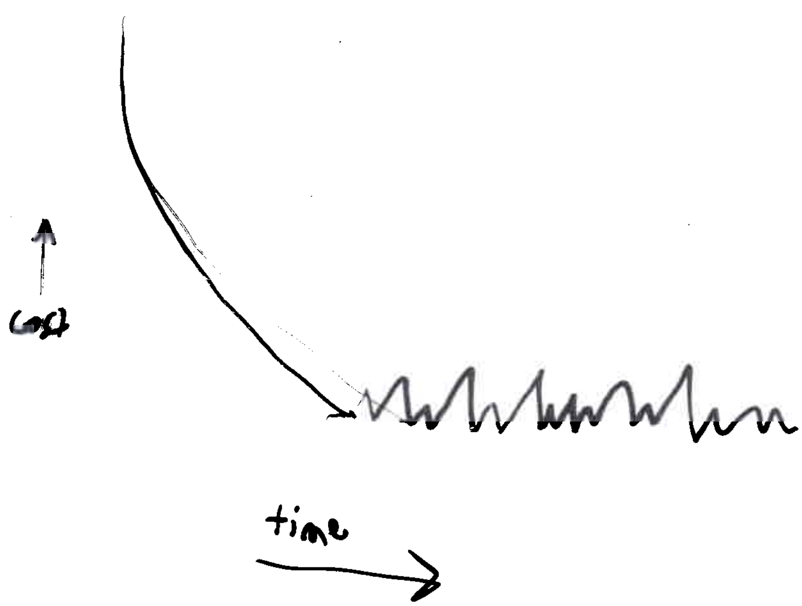


Accessed node moves to root, distance of the other nodes from the root essentially halves.

Splaying in Sequential Order



average = $3\frac{2}{3}$



What is Known

Let m be the number of accesses,
n the number of nodes.
Assume $m \geq n$.

Total time for m accesses $= O(m \log n)$:
matches bound for balanced trees.

Total time for any access sequence is within a constant factor of that for an optimum *static* tree.

Total time for n accesses, one per item, in symmetric order, is $O(n)$.

Access Lemma

For any assignment of positive weights to items, the amortized time to access item i is at most

$$3 \log (W/w_i) + 1$$

where W = total weight and the cost of an access is the depth of the accessed node.

Note. The item weights are parameters of the analysis, not of the algorithm.

Potential: define the total weight of a node to be the sum of the individual weights of its descendants, including itself.

The potential of a tree is the sum of the (base-two) logarithms of the weights of its nodes.

$$\Phi = \sum_{i=1}^{n} \log_2 (tw_i)$$

Potential: define the total weight of of a node to be the sum of the individual weights of its descendants, including itself.

$$\Phi = \sum_{i=1}^{n} tw(node\ i)$$

The potential of a tree is the sum of the (base-two) logarithms of the total weights of its nodes.

$$\Phi = \sum_{i=1}^{n} \log_2(tw_i)$$

Let $tw(x)$ = sum of weights of all items in subtree of x

rank of $x = r(x) = \log_2 tw(x)$

We shall show:

amortized time of a splay step at x is

$$\le 3(\underset{\text{after}}{r'(x)} - \underset{\text{before}}{r(x)}) \quad (+1 \text{ if zig})$$

Then total amortized time of splay is

$$\le 3(r_{final}(x) - r_{initial}(x)) + 1$$

$$\le 3(\log W - \log w_i) + 1$$

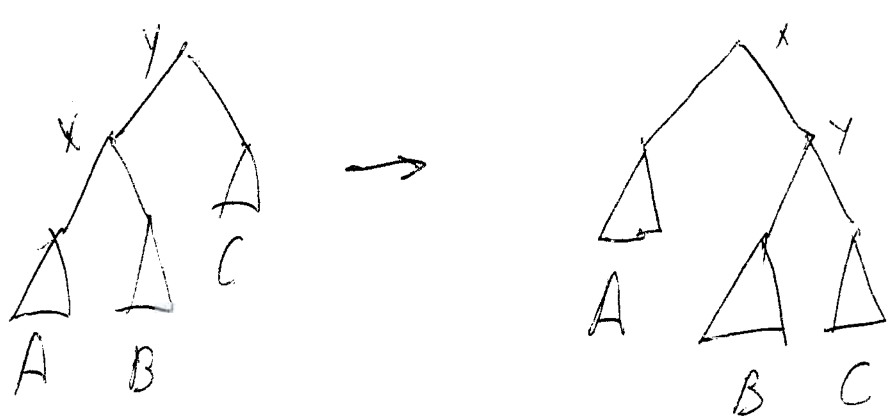
$$\le 3 \log (W/w_i) + 1$$

zig

Am. time =

$$1 + r'(y) - r(x)$$

$$\leq 1 + \left(r'(x) - r(x)\right)$$



zig-zag

Am time $= 1 + r'(y) + r'(z) - r(x) - r(y)$

~~#1 2 4~~

$$\overset{?}{\leq} 2(r'(x) - r(x))$$

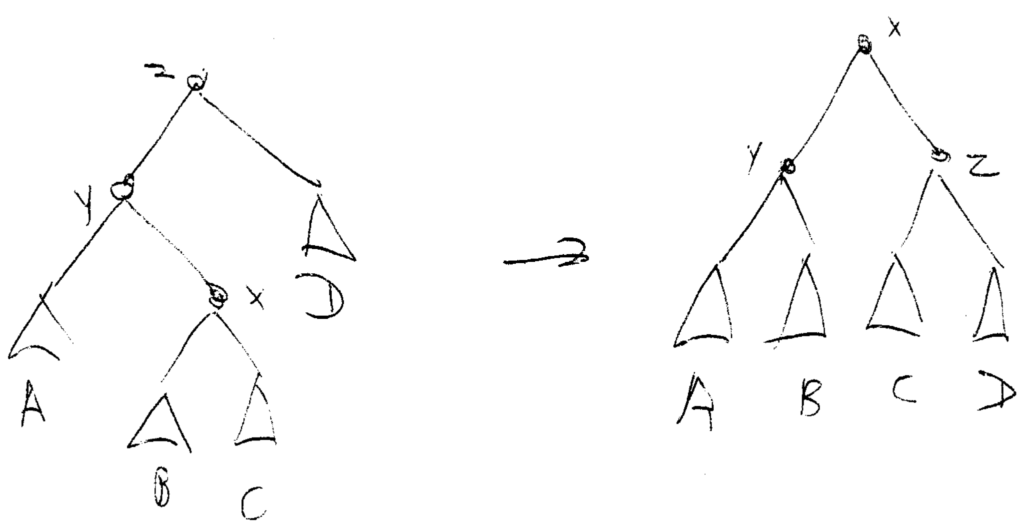
That is, $1 \overset{?}{\leq} (r'(x) - r'(y)) + (r'(x) - r'(z))$

since $r(y) \geq r(x)$

But ~~this holds~~

$1 \leq r'(x) - r'(y)$ if $tw(y) \leq tw(z)$;

$1 \leq r'(x) - r'(z)$ if $tw(z) \leq tw(y)$.



Analysis of Case 2 (zig-zig) Step

Amortized time of step

$= 1 + r'(y) + r'(z) - r(x) - r(y)$

$\leq 1 + r'(x) + r'(z) - 2r(x)$ since $r'(x) \geq r'(y)$, $r(y) \geq r(x)$

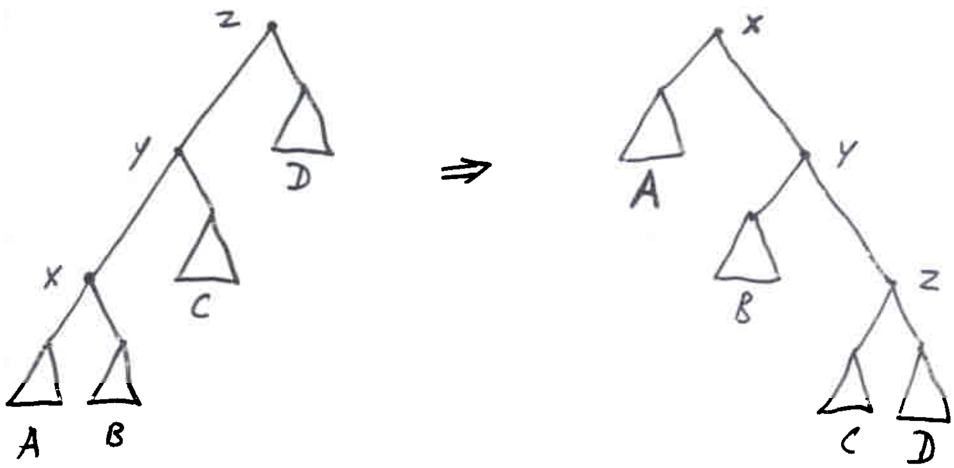
$\leq 3(r'(x) - r(x))$ iff

$$2r'(x) - r(x) - r'(z) \geq 1.$$

But $r'(x) \geq \max\{r(x), r'(z)\}$. Also $tw(x) + tw'(z) \leq tw'(x)$

Thus $\min\{tw(x), tw'(z)\} \leq tw'(x)/2$. I.e. $r'(x) \geq \min\{r(x), r'(z)\} + 1$.

$r(x) = \log tw(x)$



Access lemma holds for variants of splaying, including top-down and move half-way to root methods. For the latter, the constant factor is 2.

Corollaries

Balance Theorem
The total time for m accesses in an n-node tree is $O((m+n)\log(n+2))$.

Static Optimality Theorem
If every item is accessed at least once, the total access time is $O(m + \sum_{i=1}^{n} q_i \log(m/q_i))$, where q_i is the access frequency of item i.

Extension of argument shows that self-adjusting trees are as efficient (to within a constant factor) as <u>optimum</u> trees, over a sequence of operations.

Static Finger Theorem

The total access time is

$$O\left(n \log n + \sum_{j=1}^{m} \log(d(i_j, f) + 2)\right),$$

where f is any fixed item, i_j is the item accessed during the j^{th} access, and $d(i, i')$ is the (symmetric-order) distance between items i and i'.

"Working Set" Theorem

The total access time is

$$O\left(n \log n + \sum_{j=1}^{m} \log(t(i,j)+2)\right),$$

where $t(i,j)$ is the number of different items accessed before access j since the last access of item i.

$x \quad y \quad z \quad w \quad y \quad a \quad z \ \ldots \ x$

Thm. Total time to access all items once, in symmetric order, using splaying

$= O(n)$.

(any initial tree)

Conjecture

Dynamic Optimality

For any access sequence, splaying minimizes the total access time to within a constant factor among dynamic binary search tree algorithms, assuming unit cost per rotation and access cost equal to depth.

(Initial tree is given or $+O(n)$ term)