Splay Trees

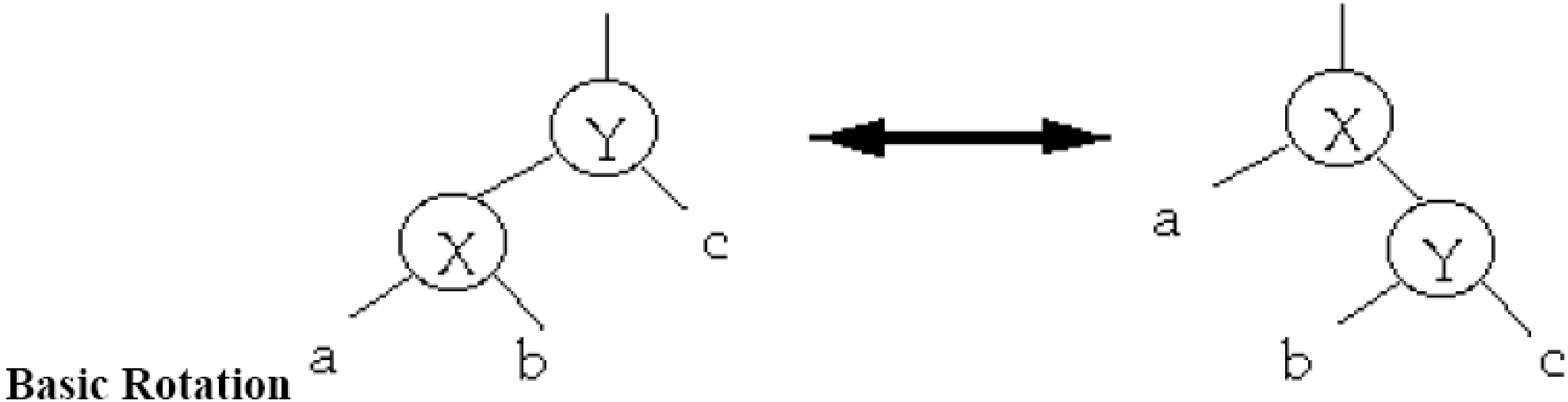
Georgia Kaouri

Advanced Data Structures

Need for Splay Trees

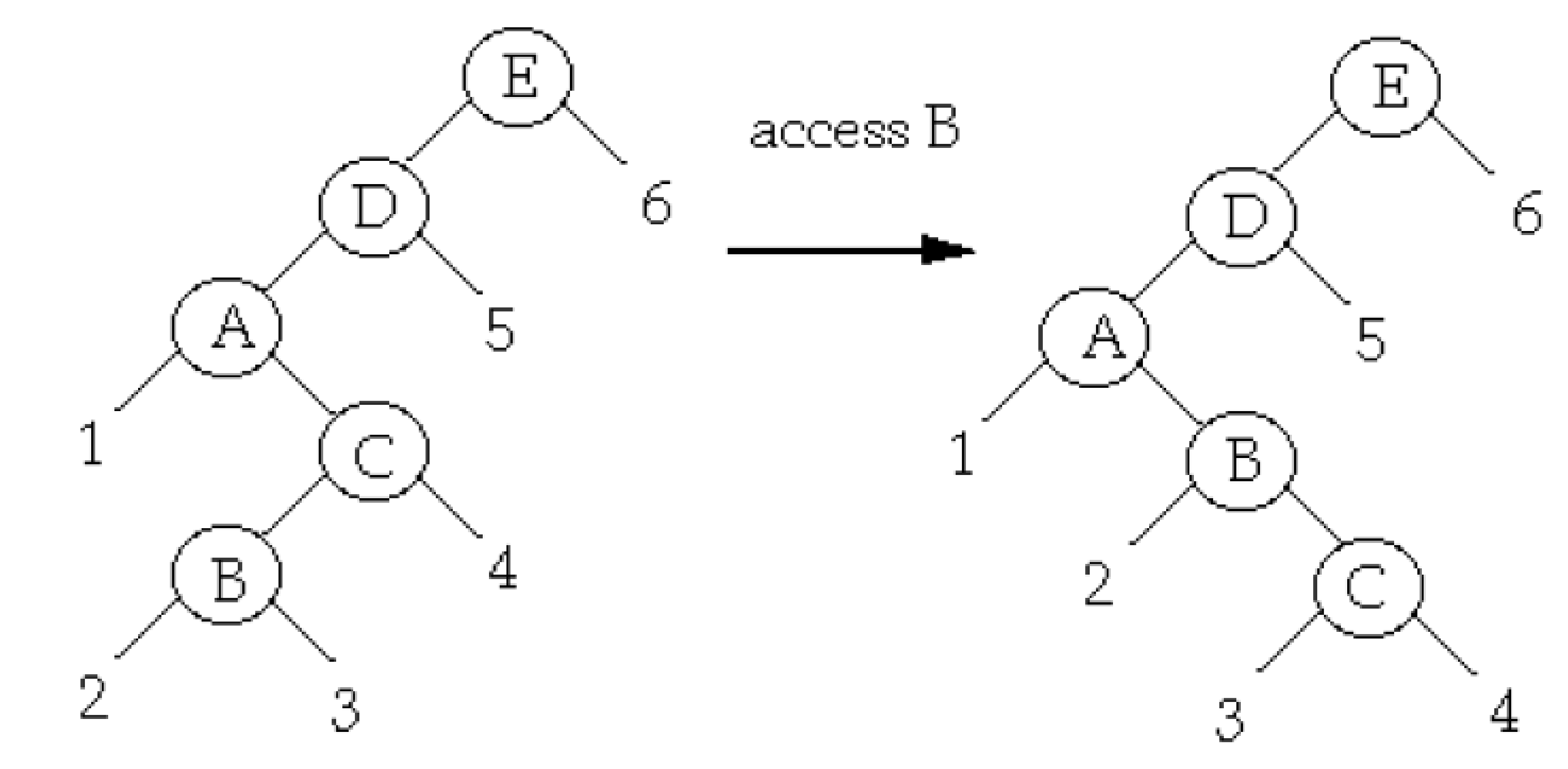
- We want to carry out a sequence of access operations on a BST.
- To minimize the total access time, accessed items should be near the root.
- Allen and Munro proposed two heuristics based on *single rotation* and *move to root* actions.

Single Rotation

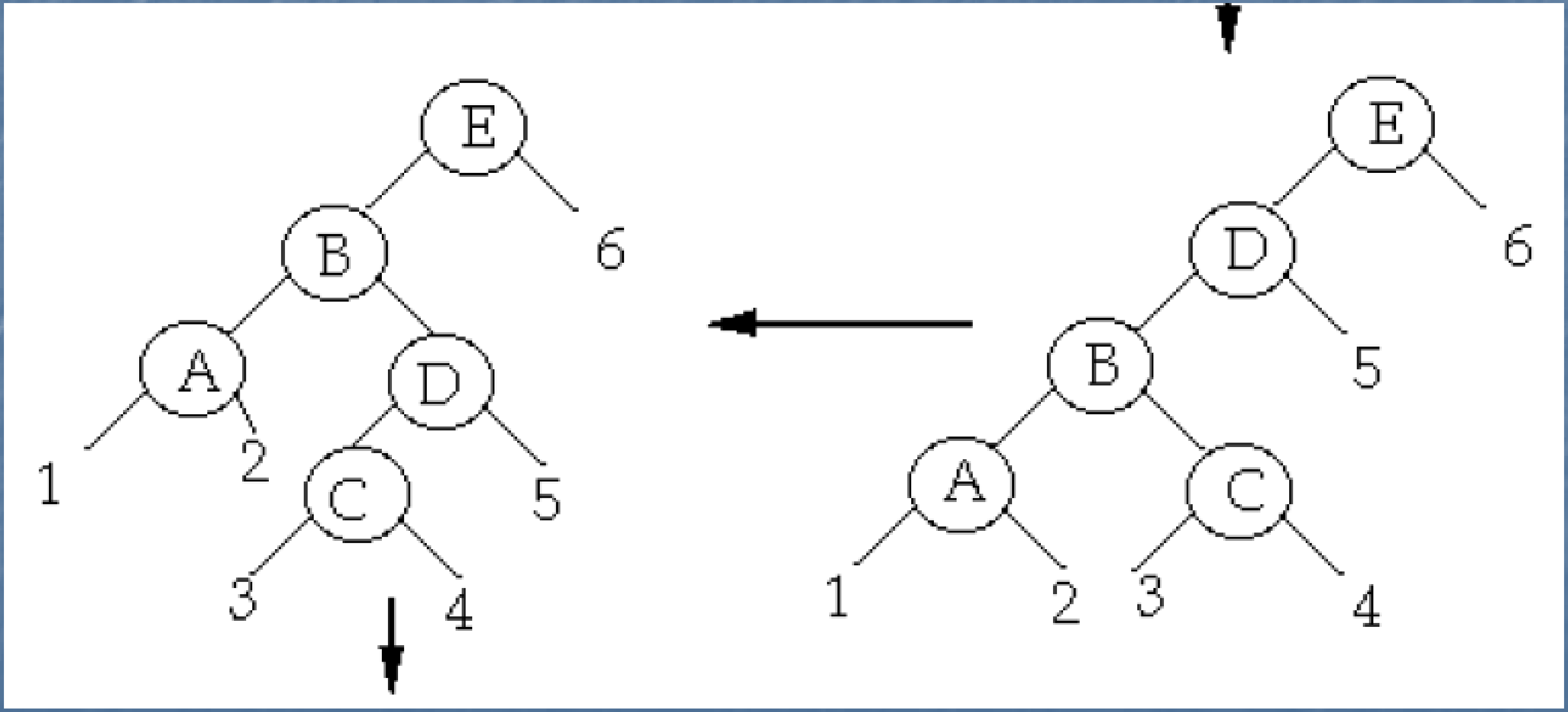


Simple Exchange (Attempt 1)

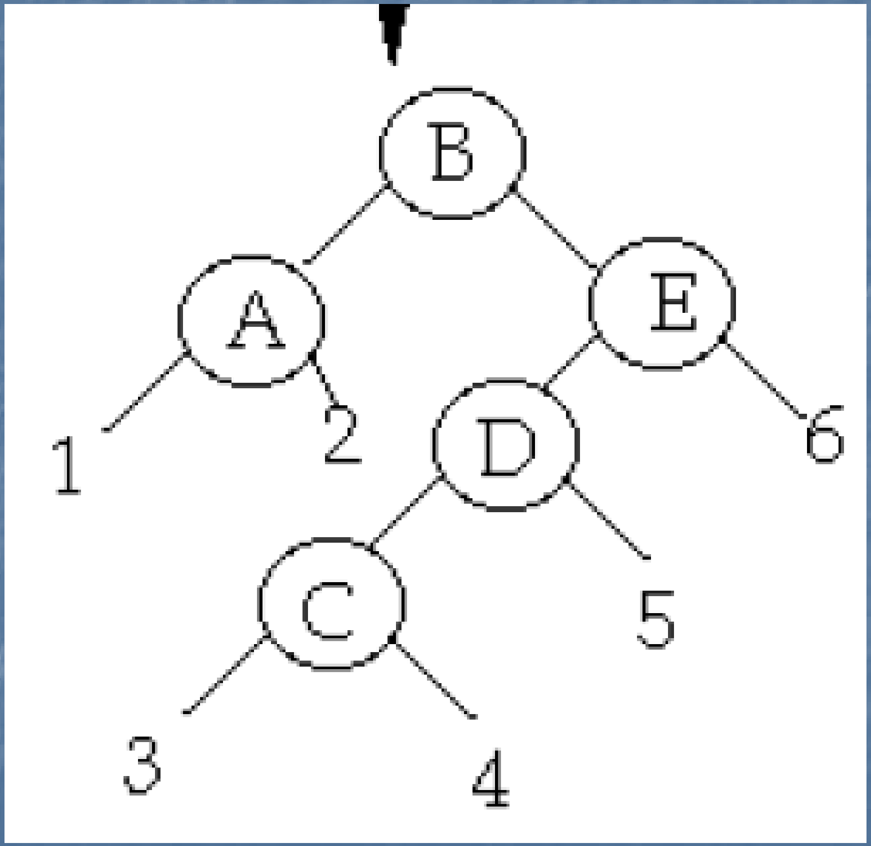
Move to root (example)



Move to root (example)



Move to root (example)



Need for Splay Trees

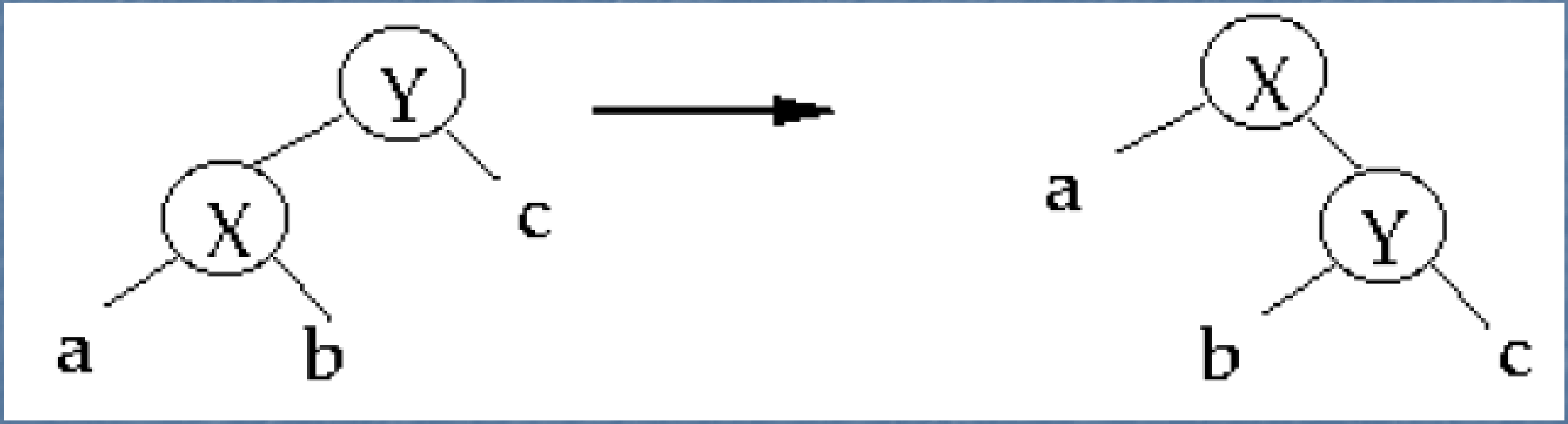
- For each one, the time per access is O(n).
- Move to root has an asymptotic average time per access within a constant factor of minimum, supposing access probabilities of the items are fixed and the accesses are independent.
- Not good enough...

Splaying

- To splay a tree at node x (with parent p(x), grandparent q(x)), we repeat the splaying step until x is the root.
  - zig
  - zig-zig
  - zig-zag

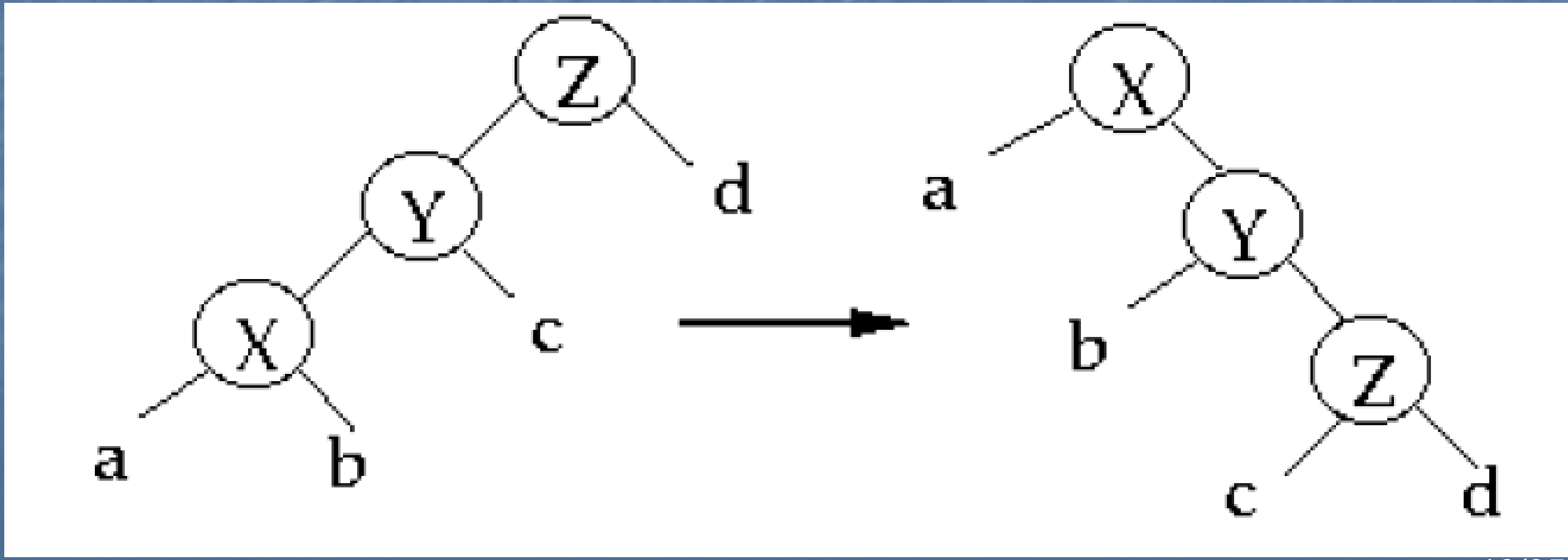
Zig

- If p(x) is the root, rotate the edge joining x with p(x).



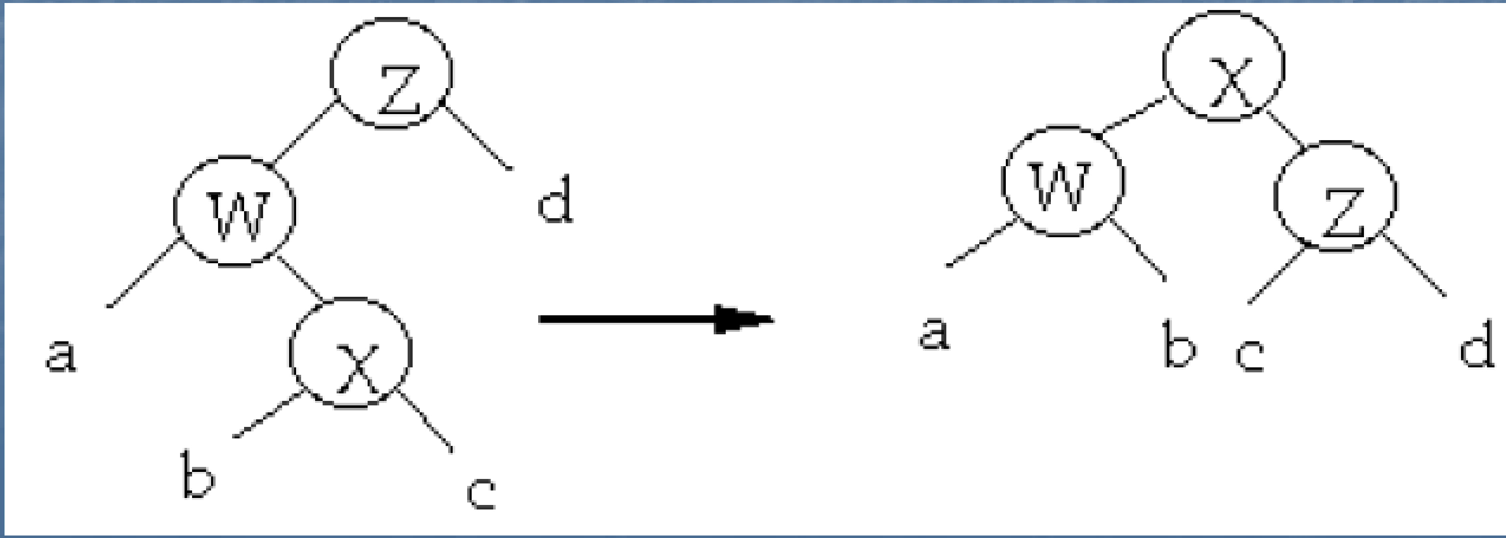
Zig-zig

- If p(x) is not the root and both x and p(x) are left or right children, we first rotate the edge joining p(x) with q(x) and then the edge joining x and p(x).



Zig-zag

- If x is a left and p(x) is a right child (or vice-versa), rotate the edge joining x with p(x) and then rotate the edge joining x with “new” p(x) (old q(x)).



Notes

- Each step has a mirror image variant that covers all the cases.
- Only the zig-zig step distinguishes splaying from rotation to the root.

Analysis of the performance of splaying

- Each node x has an arbitrary chosen positive **weight** w(x).
- Assigning different weights leads to a bound on the cost of a sequence of accesses (better bound when frequent elements have high weight).
- **size** of node x s(x)=Sum(over y in the subtree rooted at x) of w(y)
- **rank** of node x r(x)=logs(x)
- **potential** of a tree is the sum of the ranks of all its nodes

Ranks

- **Rank Rule:** Suppose two siblings have the same rank r. Then the parent has rank at least r+1.
- When a node has rank r, its size is at least 2^r. So the two siblings have total size at least $2^{(r+1)}$, so the rank of their parent has rank at least r+1.
- When a node x and its parent have the same rank r, the sibling of x must have rank <r.

About the potential

- When performing a rotation between nodes x and y, only the ranks of nodes x and y are affected.
- If y was the root of the tree before a rotation, then $r(y)=r'(x)$.
- If for each node x in T $w(x)=1$, then the potential of a balanced tree is $O(n)$ and of a long chain is $O(n\log n)$.

Amortized complexity using potential

- a: amortized time of an operation
- t: actual time of an operation (is equal to the number of rotations)
- Φ: potential before an operation
- Φ’: potential after an operation

$$a = t + \Phi' - \Phi$$

- For a sequence of m operations:

$$\sum_{j=1}^{m} t_j = \sum_{j=1}^{m} (a_j + \Phi_{j-1} - \Phi_j) = \sum_{j=1}^{m} a_j + \Phi_0 - \Phi_m$$

Access Lemma

- The amortized time to splay a tree with root t at a node x is at most

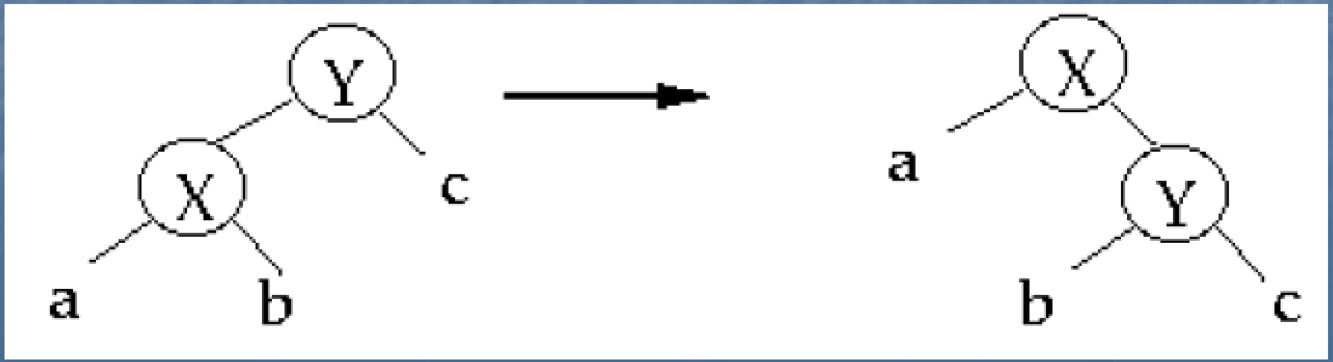
$$3(r(t)-r(x))+1=O(\log(s(t)/(s(x)))).$$

- Proof: When there are no rotations, the bound is obvious.
- Suppose at least one rotation.
- Let s, s', r, r' be the size and rank functions before and after the splaying step.
- Let y be the parent of x and z the parent of y before the step (if it exists).
- w(x)=1 for all of the nodes of the tree

Proof (cont.)

- **Zig**: One rotation takes place, so the amortized time of the step is:

$1+r'(x)+r'(y)-r(x)-r(y)$	only x,y change ranks
$\le 1+r'(x)-r(x)$	$r(y) \ge r'(y)$
$\le 1+3(r'(x)-r(x))$	$r'(x) \ge r(x)$



Proof (cont.)

- **Zig-Zig**: Two rotations are done, so:

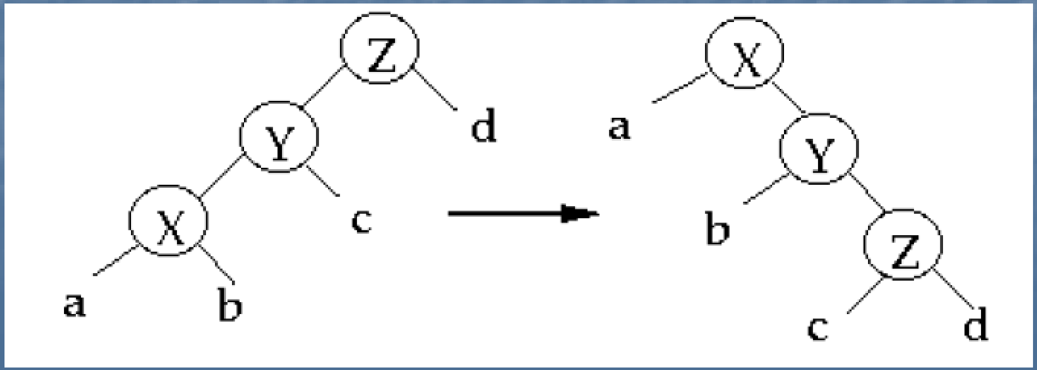
$2+r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z)$

$= 2+r'(y)+r'(z)-r(x)-r(y)$ $\quad r'(x)=r(z)$

$\le 2+r'(x)+r'(z)-2r(x)$ $\quad r'(x) \ge r'(y)$ and $r(y) \ge r(x)$

Claim: $2+r'(x)+r'(z)-2r(x) \le 3(r'(x)-r(x))$

$\leftrightarrow 2r'(x)-r(x)-r'(z) \ge 2$ follows from the convexity of the log and $s(x)+s'(z) \le s'(x)$.

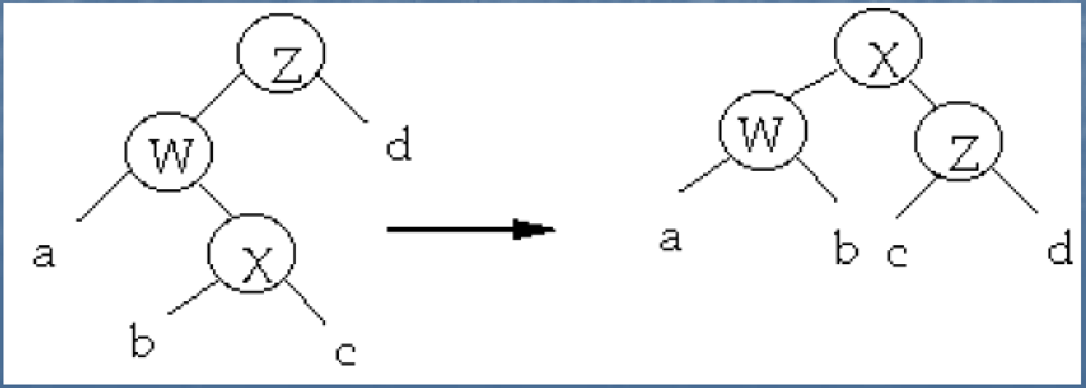


Proof (cont.)

- Zig-Zag: Also 2 rotations:

$2+r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z)$
$\leq 2+r'(y)+r'(z)-2r(x)$ $\quad r'(x)=r(z)$ and $r(x)\leq r(y)$

Claim: $2+r'(y)+r'(z)-2r(x)\leq 2(r'(x)-r(x)) \leftrightarrow$
$2r'(x)-r'(y)-r'(z)\geq 2$
As in the zig-zig case and also $s'(y)+s'(z) \leq s'(x)$.



Proof (cont.)

- Summing the amortized times estimates for all the splaying steps, and since the zig step can only occur once, the lemma follows.

- Note that the zig-zig step is the most expensive of the three.

BALANCE THEOREM: The total access time is O(m+(n+m)logn)

- Proof: For item i (1≤i≤n) assign weights w(i)=1/n. Then the total weight is 1 and the rank of the root is 0. By Access Lemma the amortized cost of an access is bounded by 3logn+1 and summing over all accesses gives O(m+mlogn).
- For a node i, the rank log(1/n)≤ r(i)≤0.
- The net decrease in the potential is at most nlogn (since the net decrease in potential over a sequence of steps is at most $\sum_{i=1}^{n} \log(W/w(i))$, where $W = \sum_{i=1}^{n} w(i)$, because the size of node i is at most W and at least w(i)).

STATIC OPTIMALITY THEOREM: The total cost of a sequence of m accesses is equal to $O\left(m + \sum_{i=1}^{n} q(i) log\left(\frac{m}{q(i)}\right)\right)$

- Proof (sketch):

  Assign weights to item i to be equal to q(i)/m.
- q(i)>0 is the access frequency of item i, i.e. the total number of times item i is accessed.
- Note that m=Sum over i of q(i).

STATIC FINGER THEOREM: If f is any fixed item, the total access time is
$$O(n \log n + m + \sum_{j=1}^{m} \log(|i_j - f| + 1))$$

Proof (sketch):

- Assign weight to item i equal to $1/(|i - f| + 1)^2$
- $W \leq 2\Sigma(1/k^2) = O(1)$
- Amortized time of the jth access: $O(\log(|i_j - f| + 1))$
- Net potential drop over the sequence: O(nlogn) (since the weight of any item is at least $1/n^2$).

WORKING SET THEOREM:

Let t(j) be the number of accesses of different items that occurred between access j and the previous access of the same item. Then the total access time is $O(n\log n + m + \Sigma^{m} \log(t(j)+1))$.

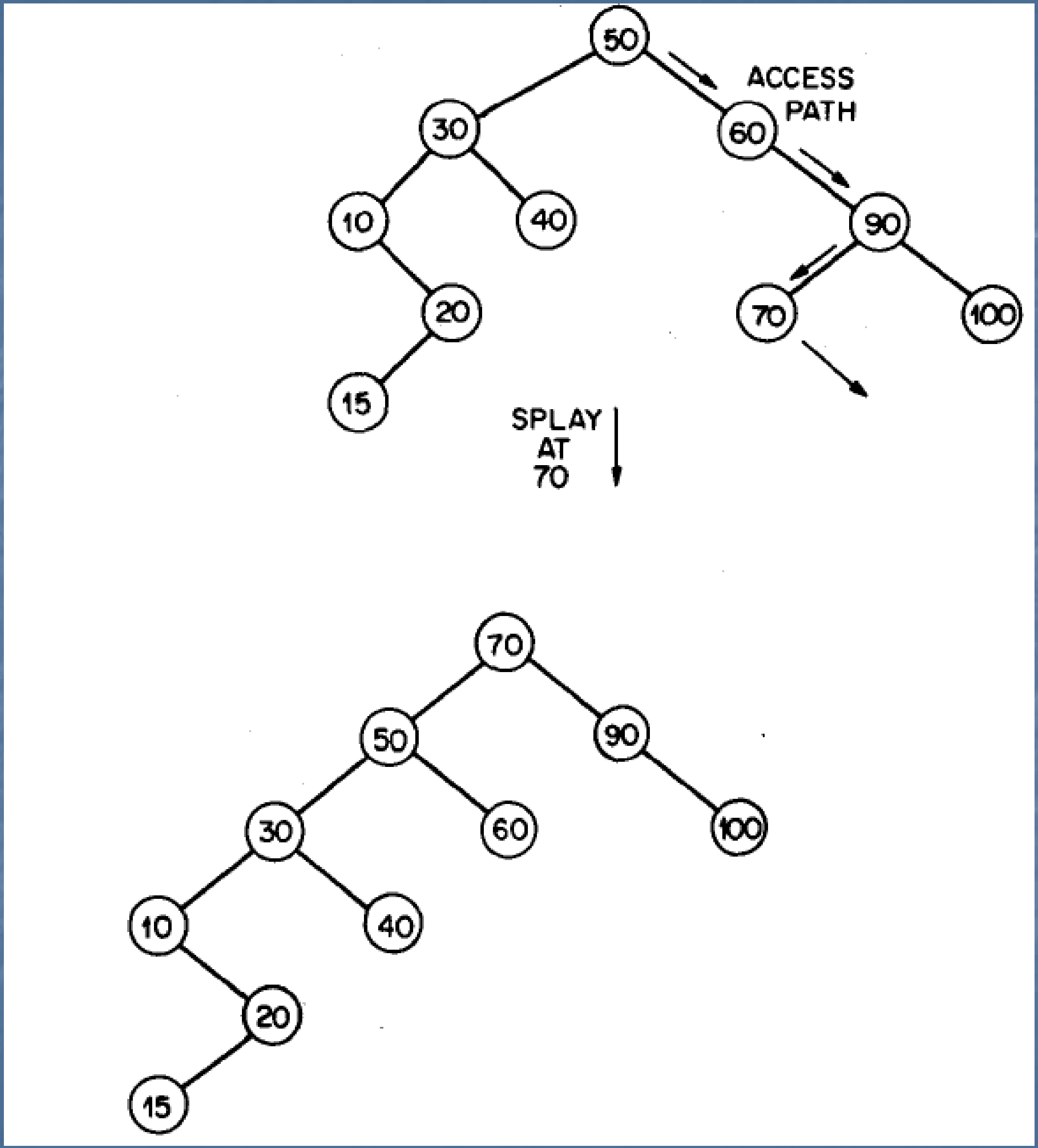
- The theorem states that if accesses concentrate on a smaller set of elements, the cost is the logarithm of this set and not of n.

DYNAMIC OPTIMALITY CONJECTURE:

Consider any sequence of successful accesses on an n-node search tree. Let A be any algorithm that carries out each access by traversing the path from the root to the node containing the accessed item, at a cost of one plus the depth of the node containing the item and that between accesses performs an arbitrary number of rotations anywhere in the tree at a cost of one per rotation. Then the total time to perform all accesses by splaying is no more than O(n) plus a constant times the time required by algorithm A.

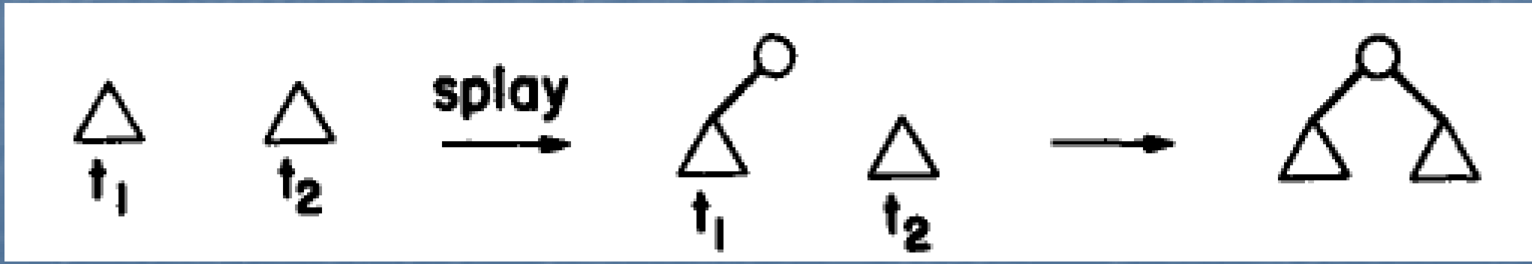
access (i, t): If i is in the tree t, return a pointer to its location, otherwise return a pointer to the null node.

- We search from the root to node i. If we find node x containing i, we splay at x and return a pointer to x, else we will find a null node (indicating i is not in the tree), we split to the last nonnull node reached and we return a pointer to null.


50
ACCESS
PATH
30
60
10
40
90
20
70
100
15
SPLAY
AT
70
70
50
90
30
60
100
10
40
20
15

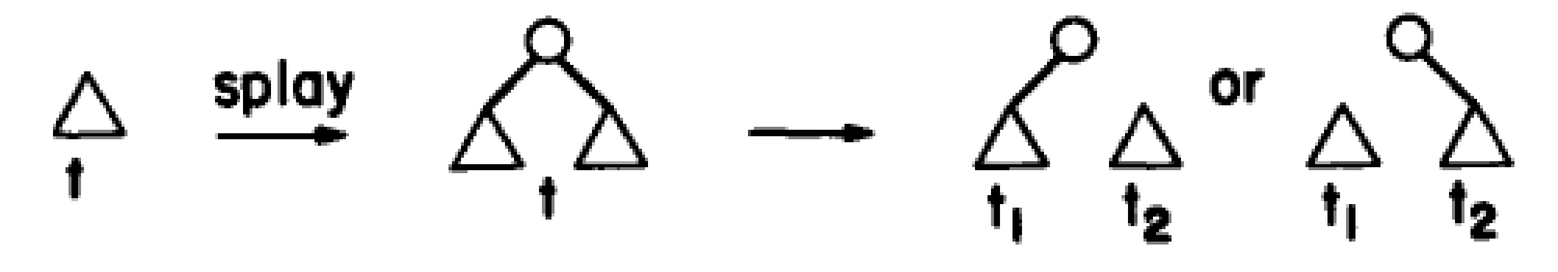
join (t_1, t_2): combine trees t_1 and t_2 into a single tree containing all items from both trees and return the resulting tree, assuming that all items in t_1 are less than those in t_2 and destroys both t_1 and t_2

- We access the largest element i in t_1 and splay at i. We make t_2 the right subtree of i and return the resulting tree.



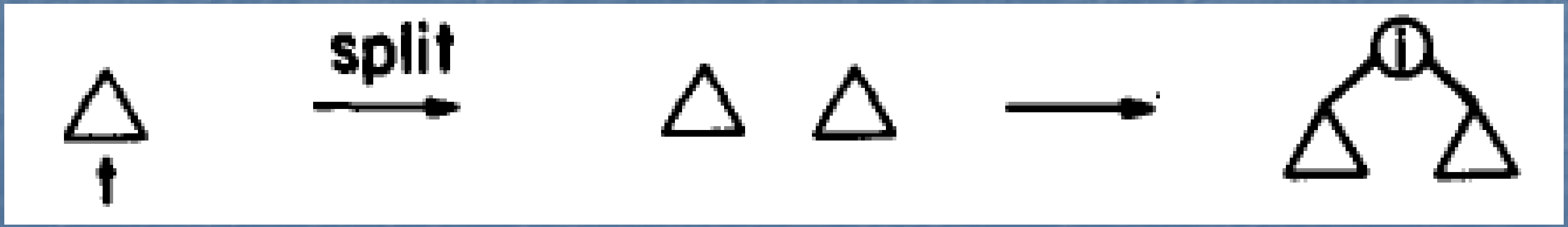
split (i, t): construct two trees t_1 and t_2, t_1 contains all items in t less than or equal to i and t_2 contains all items in t greater than i and destroy t

- We perform access(i, t) and return the two trees formed by breaking either the left link or the right link from the new root of t, depending on wheather the root contains an item greater than i or not greater than i.



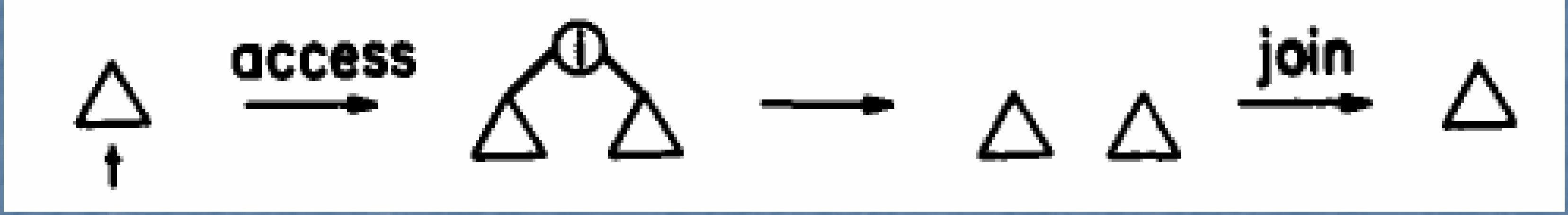
insert(i, t): insert item i to tree t, assuming t is not there already

- We perform split(i, t) and then replace t by a tree consisting of a new root node containing i, whose left and right subtrees are the trees t_1 and t_2 returned by the split.

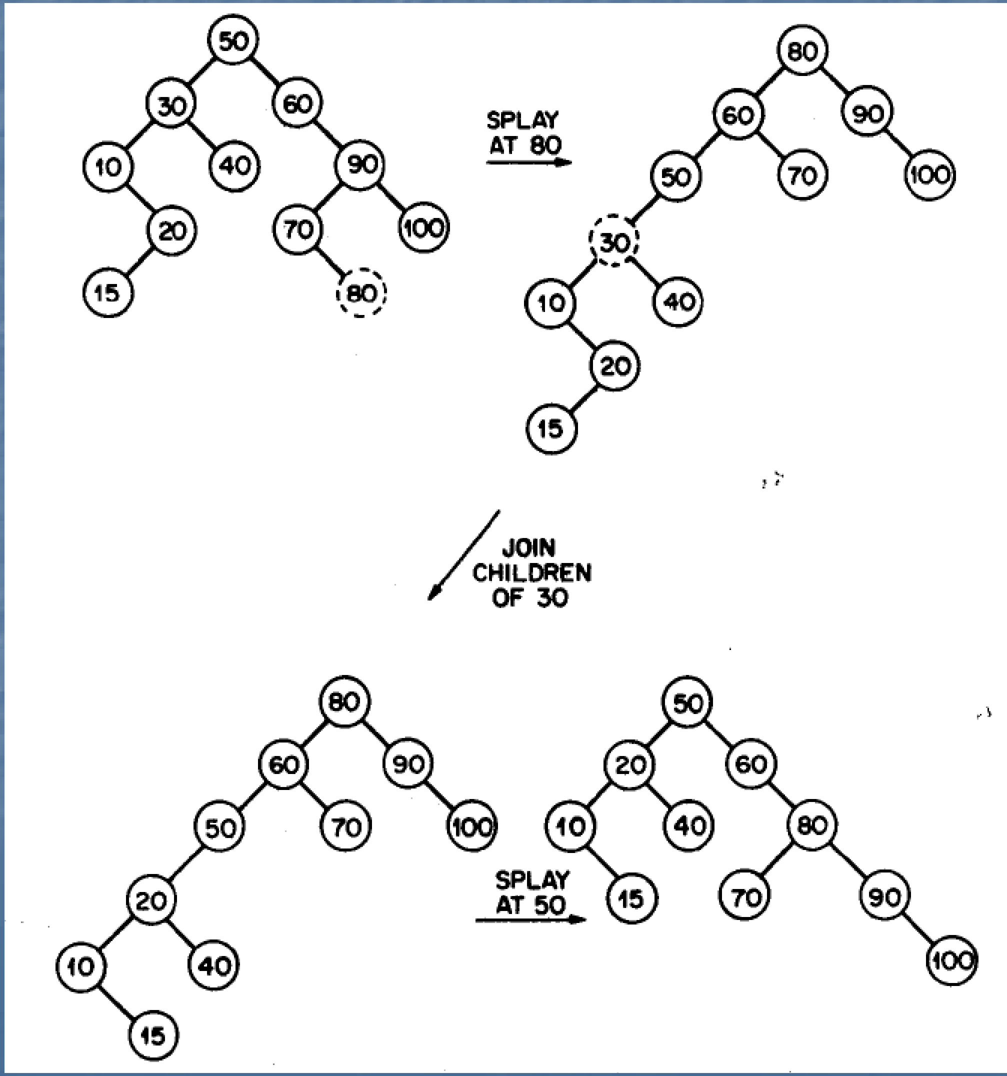


delete(i, t): delete item i from tree t, assuming it is in the tree

- We perform access(i, t) and then replace t by the join of its left and right subtrees.



Alternative definitions of insert and delete



UPDATE LEMMA about the amortized times of the previous operations

$$\textit{access}(i, t): \begin{cases} 3\log\left(\frac{W}{w(i)}\right) + 1 & \text{if } i \text{ is in } t; \\ 3\log\left(\frac{W}{\min\{w(i-), w(i+)\}}\right) + 1 & \text{if } i \text{ is not in } t. \end{cases}$$

$$\textit{join}(t_1, t_2): \quad 3\log\left(\frac{W}{w(i)}\right) + O(1), \quad \text{where } i \text{ is the last item in } t_1.$$

$$\textit{split}(i, t): \begin{cases} 3\log\left(\frac{W}{w(i)}\right) + O(1) & \text{if } i \text{ is in } t; \\ 3\log\left(\frac{W}{\min\{w(i-), w(i+)\}}\right) + O(1) & \text{if } i \text{ is not in } t. \end{cases}$$

$$\textit{insert}(i, t): \quad 3\log\left(\frac{W - w(i)}{\min\{w(i-), w(i+)\}}\right) + \log\left(\frac{W}{w(i)}\right) + O(1).$$

$$\textit{delete}(i, t): \quad 3\log\left(\frac{W}{w(i)}\right) + 3\log\left(\frac{W - w(i)}{w(i-)}\right) + O(1).$$

The end