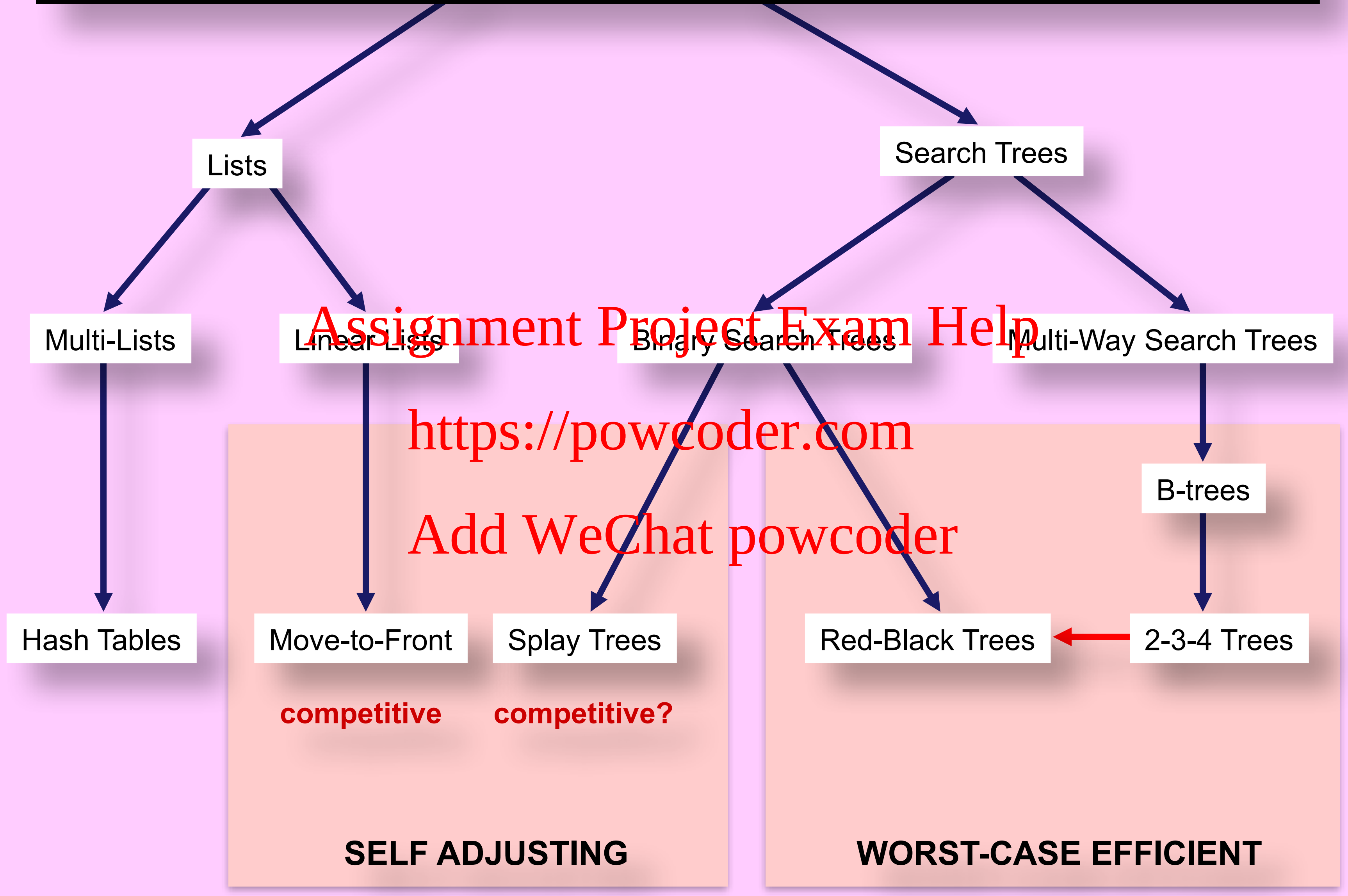
EECS 4101/5101

Prof. Andy Mirzaian

Splay Tree: Self Adjusting BST

DICTIONARIES



References:

- Lecture Note 3
- AAW animation

Self-Adjusting Binary Search trees

- Red-Black trees guarantee that each dictionary operation (Search, Insert, Delete) takes O(log n) time in the worst-case. They achieve this by enforcing an explicit “balance” constraint on the tree.

- Can we endow self-adjusting feature to the standard BST by the use of appropriate rotations to restructure the tree and improve the efficiency of future operations in order to keep the amortized cost per operation low? Is there a simple self-adjusting data structure that guarantees O(log n) amortized time per dictionary operation? YES, Splay trees.

- As with Red-Black trees, Splay trees give the following guarantee:

  Any sequence of m dictionary operations on an initially empty dictionary takes in total O(m log n) time, where n is the max size of the dictionary during this sequence.

  However, a single dictionary operation on Splay trees in the worst-case may cost up to Θ(n) time.

- Splay Trees maintain a BST in an arbitrary state, with no “balance” information/constraint. However, they perform the simple, uniformly defined, splay operation after each dictionary operation. A splay operation on BSTs is analogous to Move-to-Front on linear lists.

Generic Splay

- **GenericSplay(x):** Consider the links on the search path of node x in BST T. This operation performs one rotation per link on this path in some appropriate order to move node x up to the root of T.
- The number of rotations (at a cost of O(1) per rotation) done by GenericSplay(x) is $\text{depth}_T(x)$, i.e., depth of x in T before the splay. Access cost of node x is $O(1+ \text{depth}_T(x))$.

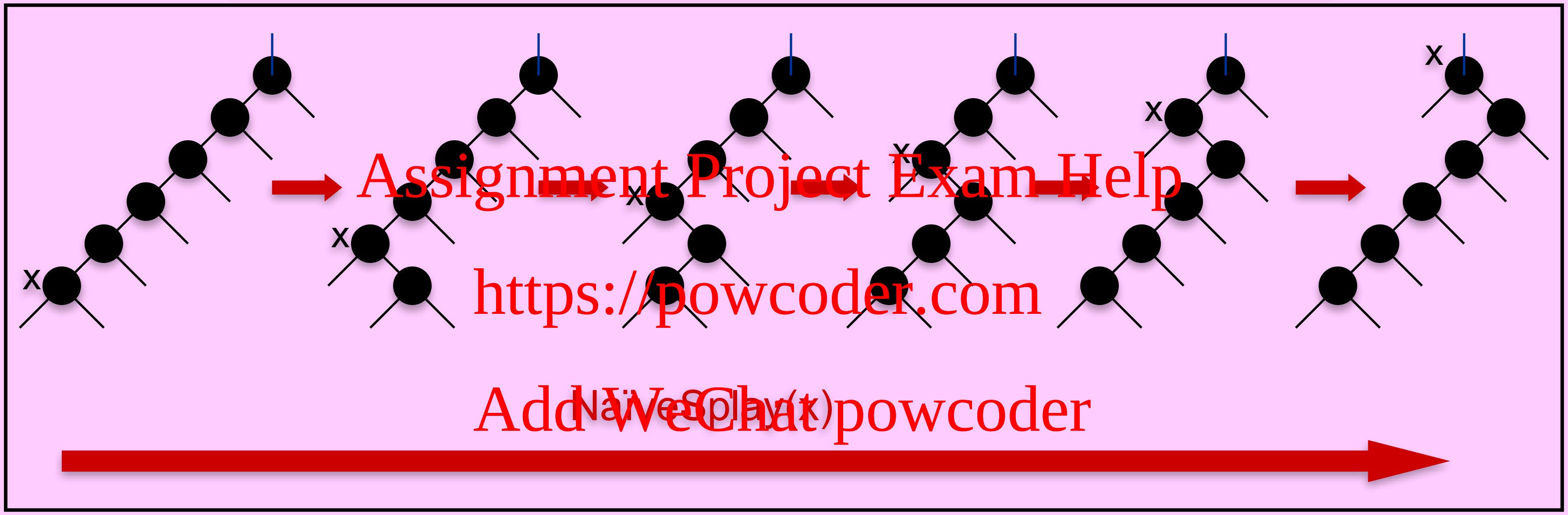
- **Generic Splay tree:** This is an arbitrary BST such that after each dictionary operation we perform GenericSplay(x), where x is the deepest node accessed by the dictionary operation that is still in the tree. The cost of the dictionary operation, including the splay, is O(1+depth(x)).

Which node is x?
  - Successful search: x is the node just accessed.
  - Unsuccessful search: x is parent of the external node just accessed.
  - Insert: x is the node now holding the insertion key.
  - Unsuccessful Delete: same as unsuccessful search.
  - Successful Delete: x is parent of the spliced-out node.

- Any sequence of m dictionary operations takes O(m+R) time, where R is the total number of rotations done by the generic splay operations in the sequence. So, we need to find an upper bound on R.

Naïve Splay

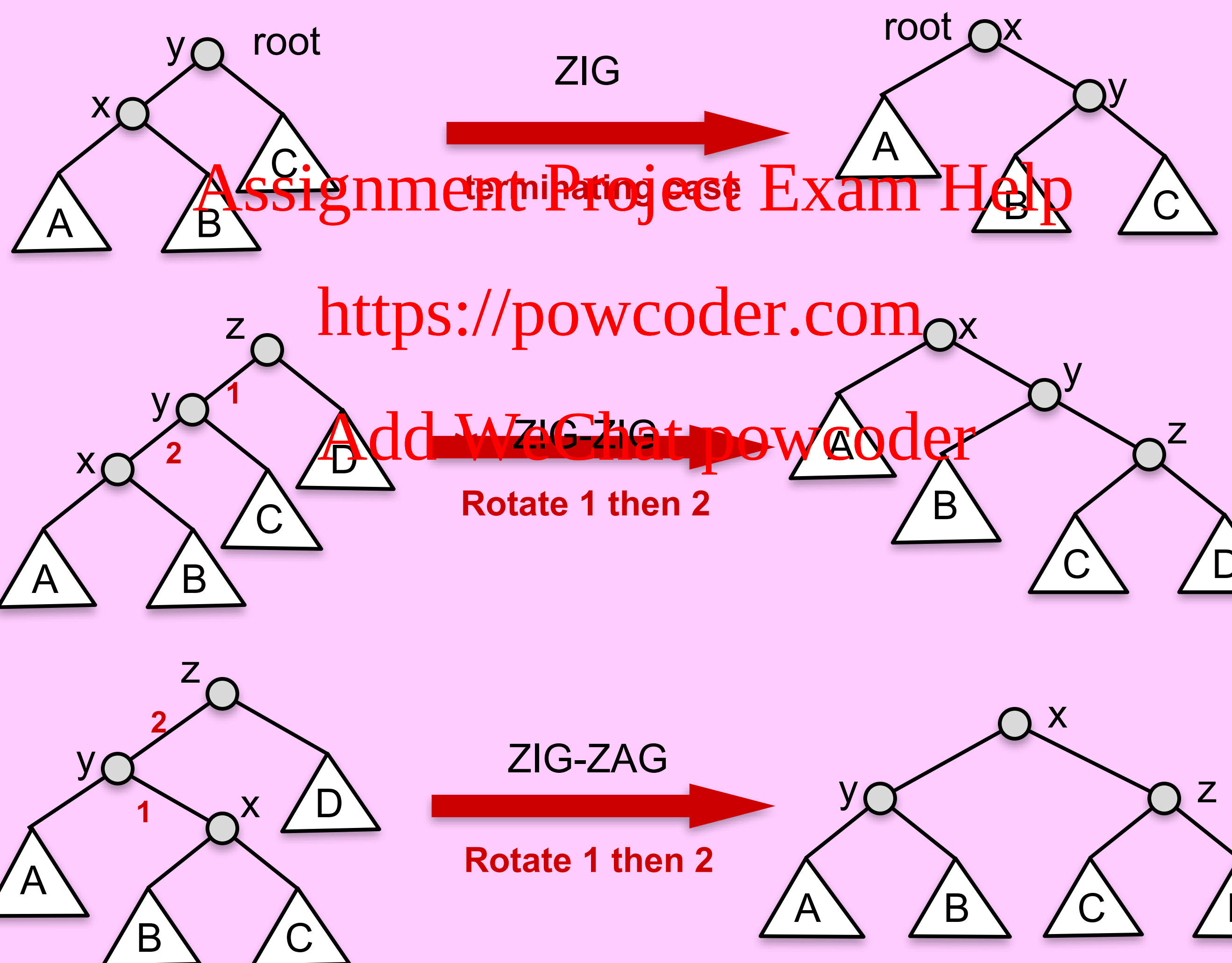
```
NaïveSplay(x):   A series of single rotations up the tree:
    while p[x] ≠ nil do
        Rotate(p[x],x)
```



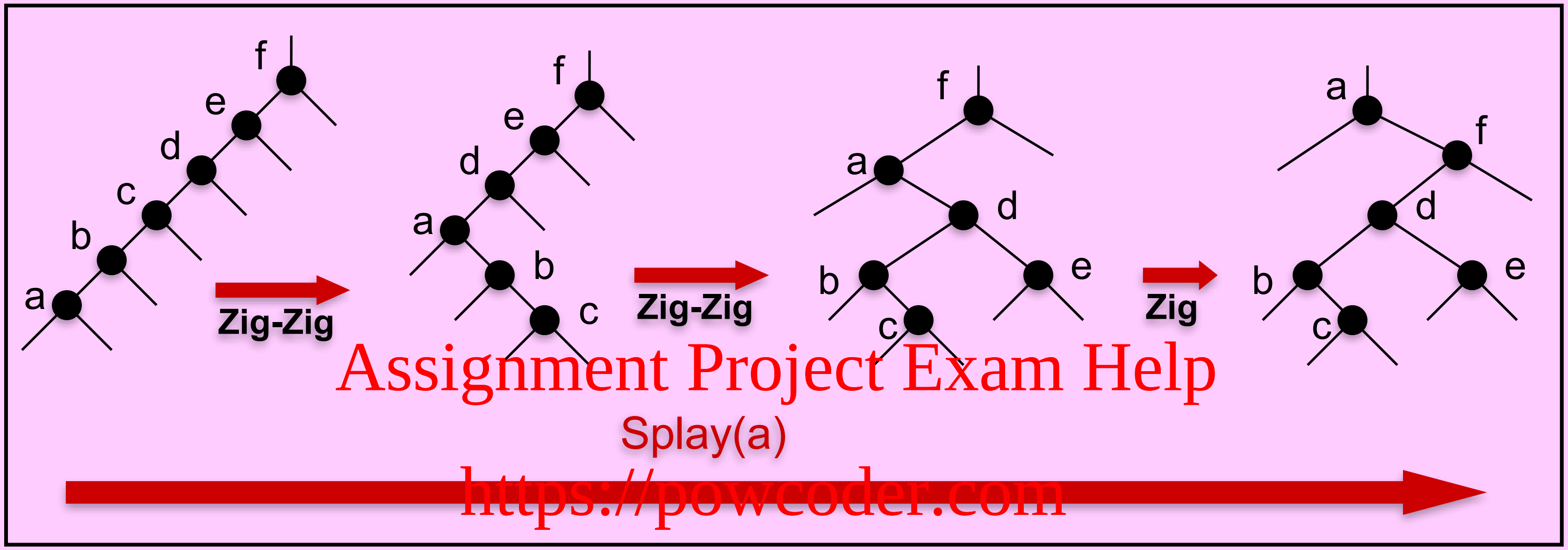
Exercise: Show that there is an adversarial sequence of n dictionary operations on an initially empty BST on which NaïveSplay takes $\Omega(n^2)$ time total, i.e., amortized $\Omega(n)$ time per dictionary operation.

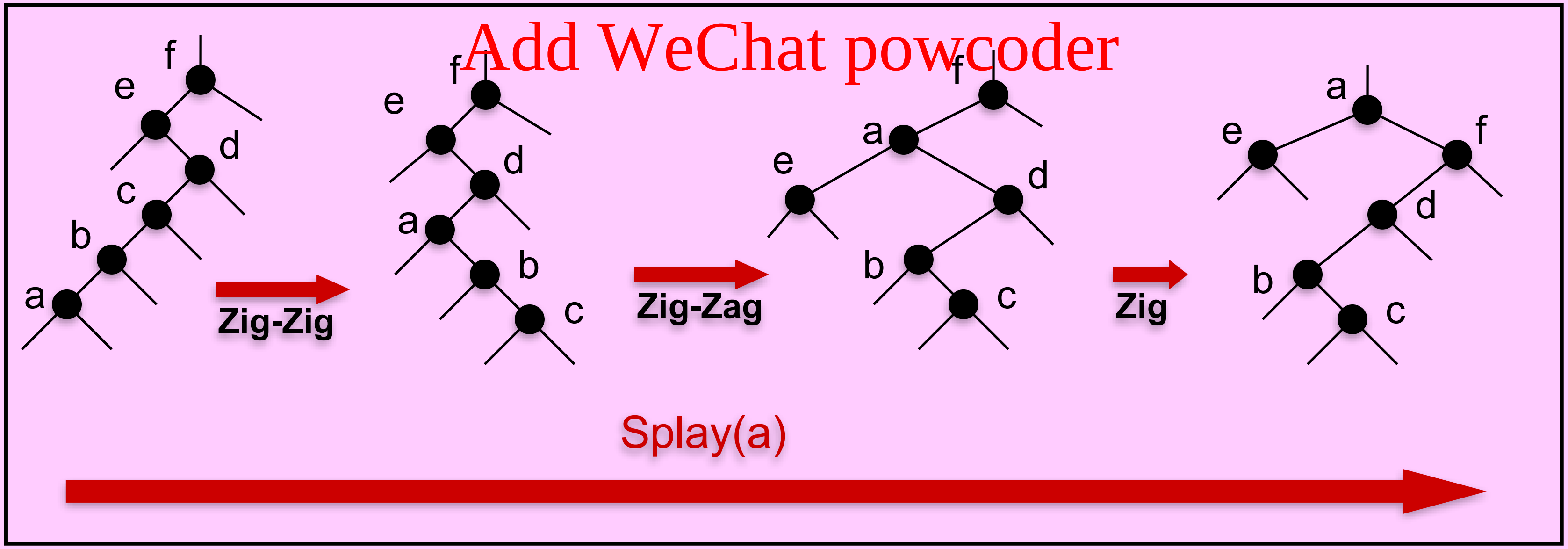
Splay(x)

Perform a series of the following splay steps, whichever applies, until x becomes the root of the tree. (Left-Right symmetric cases not shown.)



Splay Examples





Amortized Analysis

- The aggregate time on any sequence of m dictionary operations on an initially empty BST is

O(m+R)

R = total # rotations done by Splay operations in the sequence.

- We shall upper bound R by amortized analysis.

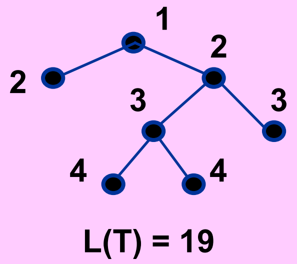
- We analyze amortized cost of Splay with unit cost per rotation.
- What is a "good" candidate for the potential function?

Potential Function (part 1/2)

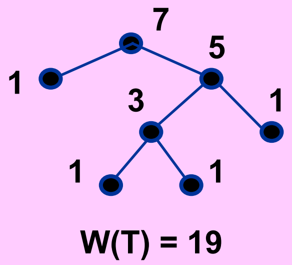
- $1+\text{depth}_T(x)$ = # ancestors of x (including x) in T
  = access cost of x in T.

- $w_T(x)$ = weight of x in T
  = # descendants of x (including x) in T
  = # nodes in the subtree rooted at x.

- Candidate 1:
  Internal path length of T: $L(T) = \Sigma_{x\in T}(1+\text{depth}_T(x))$.



- Candidate 2:
  Weight of T: $W(T) = \Sigma_{x\in T}\, w_T(x)$.



- **FACT:** (1) $\forall T\ L(T) = W(T)$. Why?

  (2) As potentials, L(T) and W(T) are regular
  (i.e., always 0, $L(\varnothing) = W(\varnothing) = 0$).

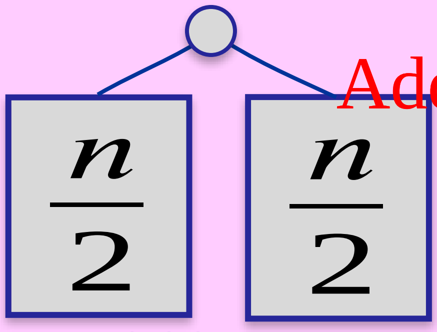
  (3) $\Theta(n \log n) \leq L(T) = W(T) \leq \Theta(n^2)$, where $n=|T|$.

- L(T) & W(T) can become too large. They won't work!
  Dampen W(T) by a sublinear function, e.g., log.

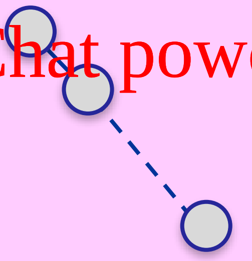
Potential Function (part 2/2)

- Rank of x in T: $r_T(x) = \log w_T(x)$.
- **Potential:** $\Phi(T) = \Sigma_{x \in T}\, r_T(x)$.
- **FACT:** $\Phi(T)$ is also regular, and

  $\Theta(n) \le \Phi(T) \le \Theta(n \log n)$.



$\Phi(n) = 2\,\Phi(n/2) + \log n = \Theta(n)$



$\Phi(n) = \Phi(n\text{-}1) + \log n = \Theta(n \log n)$

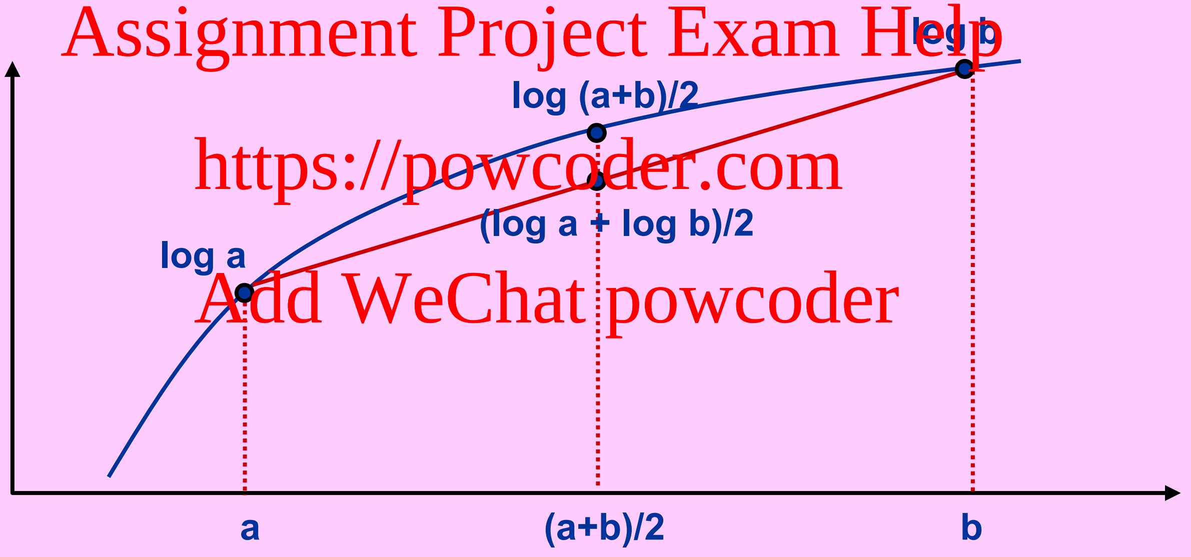
The LOG LEMMA

LEMMA 0: For any numbers a, b, c 1

a + b ≤ c ⇒ log a + log b ≤ 2 log c – 2.

Proof 1: c^2 $(a+b)^2 = 4ab + (a-b)^2$ 4ab. So, c^2 4ab. Now take log.

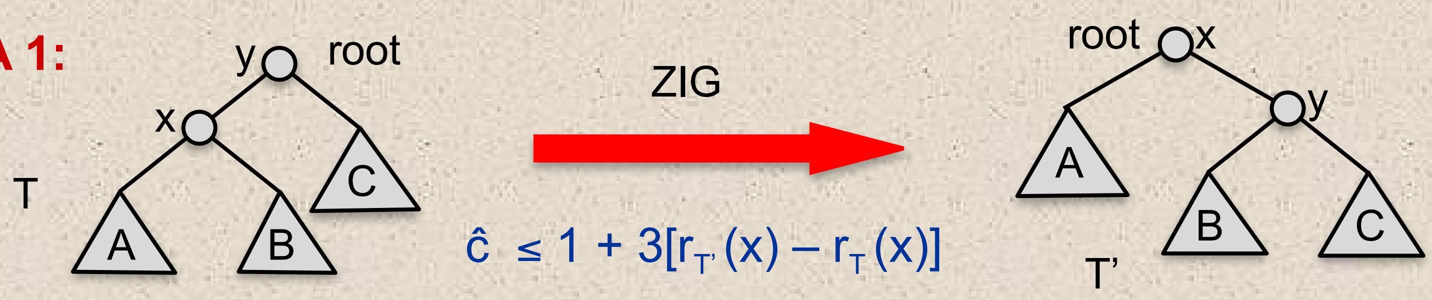
Proof 2: Logarithm is a concave function:



This makes the potential function work!

ZIG LEMMA

LEMMA 1:



$\hat{c} \le 1 + 3[r_{T'}(x) - r_T(x)]$

Proof:

$$\hat{c} = c + \Delta\Phi$$

$$= 1 + [r'(x) + r'(y)] - [r(x) + r(y)]$$

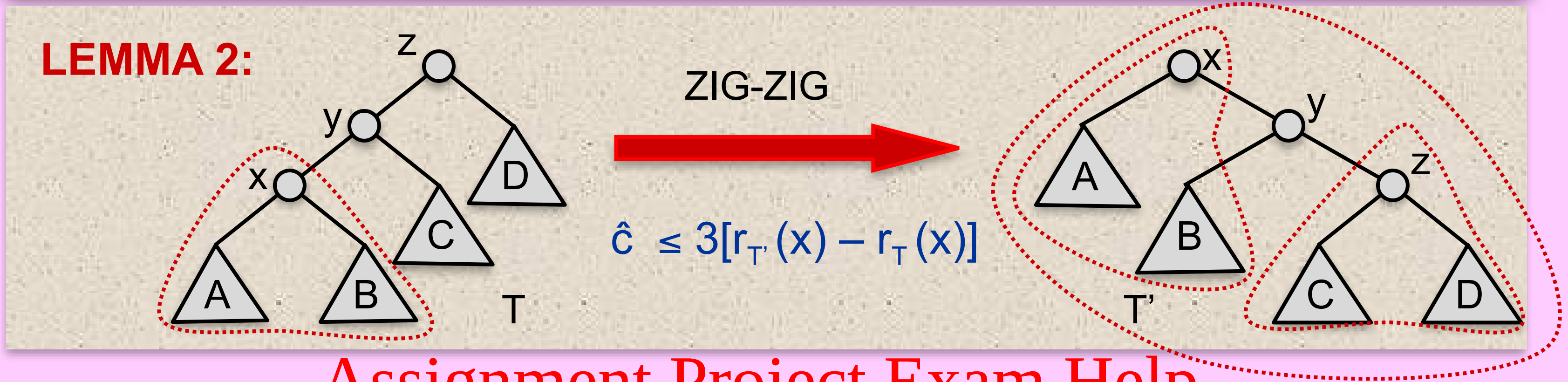
$$= 1 + r'(y) - r(x) \qquad \text{since} \quad r'(x) = r(y)$$

$$\le 1 + r'(x) - r(x) \qquad \text{since} \quad r'(y) \le r'(x)$$

$$\le 1 + 3[r'(x) - r(x)]. \qquad \text{since} \quad r(x) \le r'(x)$$

ZIG-ZIG LEMMA

LEMMA 2:



$\hat{c} \le 3[r_{T'}(x) - r_T(x)]$

Proof:

$\hat{c} = c + \Delta\Phi$

$= 2 + [r'(x) + r'(y) + r'(z)] - [r(x) + r(y) + r(z)]$

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

$\le 3\left(r'(x) - r(x)\right).$ since $w(x) + w'(z) \le w'(x)$

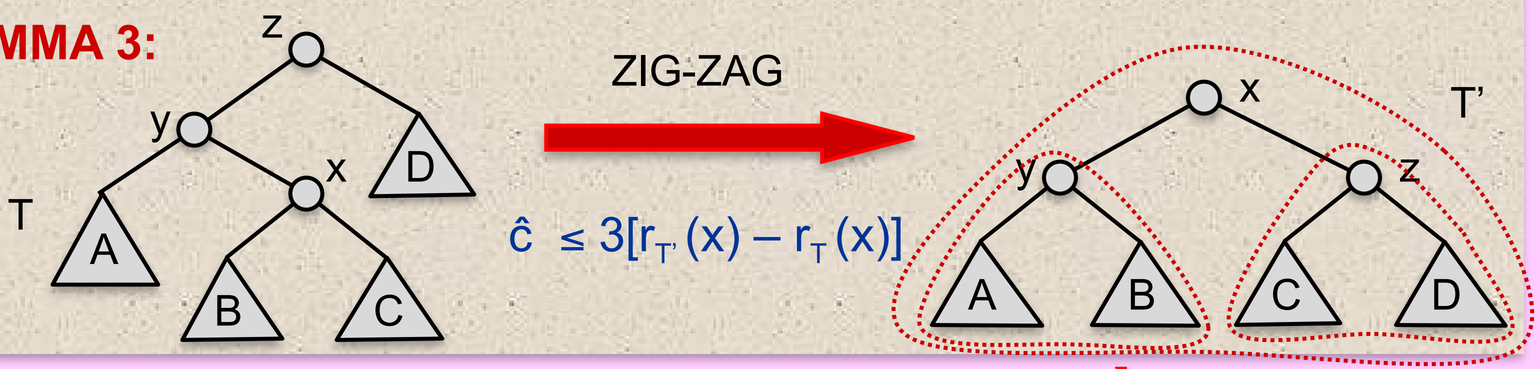
$\Rightarrow r(x) + r'(z) \le 2r'(x) - 2$

by the LOG Lemma

$\Rightarrow 2 + r'(z) \le 2r'(x) - r(x)$

ZIG-ZAG LEMMA

LEMMA 3:



$\hat{c} \le 3[r_{T'}(x) - r_T(x)]$

Proof:

$\hat{c} = c + \Delta\Phi$

$= 2 + [r'(x) + r'(y) + r'(z)] - [r(x) + r(y) + r(z)]$

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

$\le 2\left(r'(x) - r(x)\right)$ since $w'(y) + w'(z) \le w'(x)$

$\Rightarrow r'(y) + r'(z) \le 2r'(x) - 2$

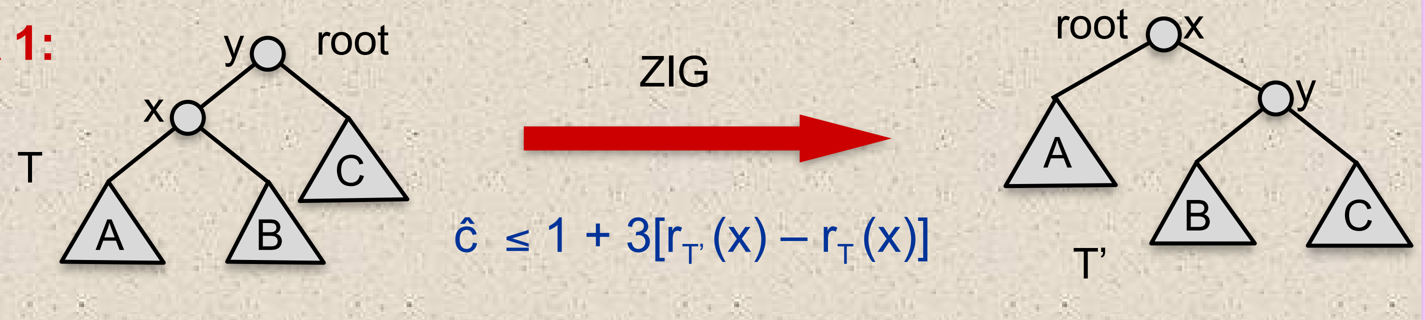
by the LOG Lemma

$\le 3\left(r'(x) - r(x)\right).$

$\Rightarrow 2 + r'(y) + r'(z) \le 2r'(x)$

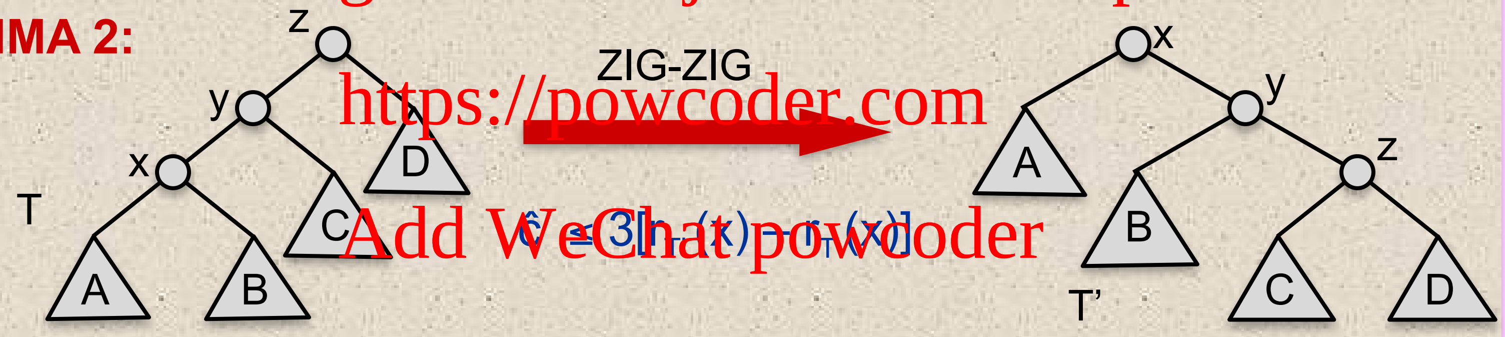
Lemmas on each Splay Step

LEMMA 1:



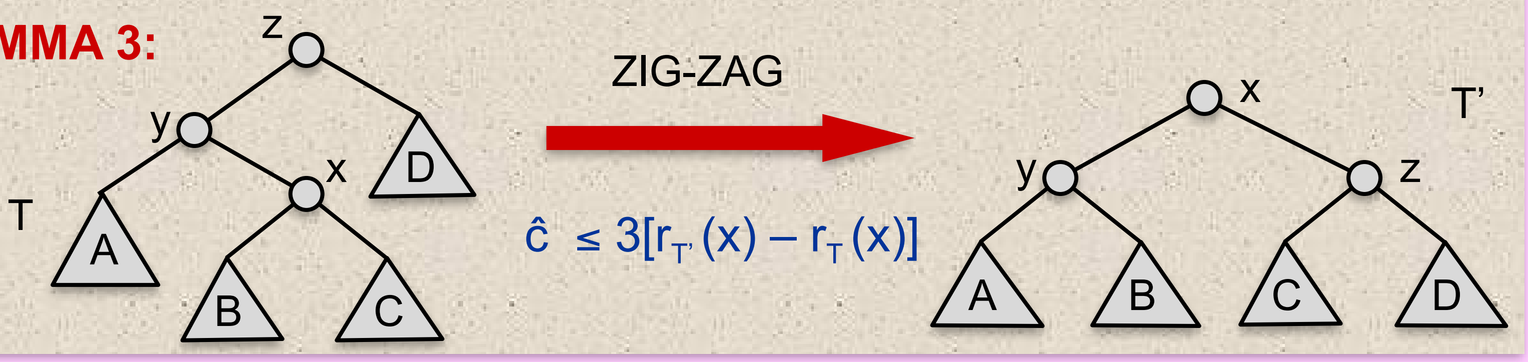
$\hat{c} \leq 1 + 3[r_{T'}(x) - r_T(x)]$

LEMMA 2:



$\hat{c} \leq 3[r_{T'}(x) - r_T(x)]$

LEMMA 3:



$\hat{c} \leq 3[r_{T'}(x) - r_T(x)]$

Amortized cost of Splay

THEOREM 1: Amortized number of rotations by a Splay operation on an n-node BST is at most **3 log n + 1.**

Proof: Splay(x) operation is a sequence of splay steps:

$$\begin{tikzcd}
T_{before} = T_0 \arrow[r] & \cdots \arrow[r] & T_{i-1} \arrow[r, "c_i", "\hat{c}_i"'] & T_i \arrow[r] & \cdots \arrow[r] & T_k = T_{after} \\
\Phi_{before} = \Phi_0 & & \Phi_{i-1} & \Phi_i & & \Phi_k = \Phi_{after}
\end{tikzcd}$$

$$\hat{c} = c + \Phi_{after} - \Phi_{before}$$

$$= \sum_{i=1}^{k} (c_i + \Phi_i - \Phi_{i-1})$$

$$= \sum_{i=1}^{k} \hat{c}_i$$

$$\le \sum_{i=1}^{k-1} 3(r_i(x) - r_{i-1}(x)) + (1 + 3(r_k(x) - r_{k-1}(x)))$$

only last splay step can be a ZIG

$$= 1 + 3(r_{after}(x) - r_{before}(x))$$

$$\le 1 + 3\log n \qquad \text{since} \quad r_{after}(x) = \log n, \ r_{before}(x) \quad 0$$

Amortized cost of dictionary op's

LEMMA 4:
Let T be any n-node BST.
(a) Increase in potential due to insertion of a new leaf into T, excluding the cost of splaying, is at most log n.
(b) Splicing-out any node from T cannot increase the potential.

Proof: Exercise 1.

THEOREM 2:
Aggregate number of rotations by any sequence of m dictionary operations on an initially empty Splay Tree of max size ≤ n is R ≤ m(4 log n + 1).

Hence, aggregate execution time for such a sequence is at most O(m+R) = O(m log n).

Proof: Amortized number of rotations by a dictionary operation:

Search:	at most	1 + 3 log n	(by Theorem 1) .
Insert:	at most	1 + 4 log n	(by Theorem 1 & Lemma 4a) .
Delete:	at most	1 + 3 log n	(by Theorem 1 & Lemma 4b) .

Competitiveness?

Dynamic Optimality Conjecture

- Move-to-Front is competitive on linear lists with exchanges.
- Is SPLAY competitive on binary search trees with rotations?
- Sleator, Tarjan, "Self-adjusting binary search trees,"
  J. ACM 32(3), pp: 652-686, 1985

originally proposed Splay Trees and made the following bold conjecture:

Dynamic Optimality Conjecture:

s = any sequence of dictionary operations starting with any n-node BST.

OPT = optimum off-line algorithm on s.

OPT carries out each operation by starting from the root and accessing nodes on the corresponding search path, and possibly some additional nodes, at a cost of one per accessed node, and that between each operation performs an arbitrary number of rotations on any accessed node in the tree, at a cost of one per rotation.

Then, $\mathbf{C_{SPLAY}(s) = O(n+C_{OPT}(s))}$.

Special Cases

- Dynamic Optimality Conjecture is a very strong claim and is still open.
- Some special cases have been confirmed.
- The following paper gives a survey, offers alternatives to splaying, and develops a new method to aid finding competitive online BST algorithms:

  Demaine, Harmon, Iacono, Kane, Pătraşcu, "The geometry of binary search trees," SODA, pp: 496-505, 2009.

  [To see this paper, click here.]

- Some known special cases:
  - ➢ Sequential Access [Tar85]: Access each node of a given n-node BST in inorder (smallest, then 2nd smallest, …, then largest) always starting an access from the root. This takes Θ(n) time on Splay trees!
    Red-Black trees are not competitive and take Θ(n log n) time.
  - ➢ Static Optimality [ST85]: Access only operations on an n-node BST. Static OPT (no rotations allowed) [Knuth's dynamic programming algorithm]. (See AAW animation on Optimal Static BST, my EECS 3101 LS7 or LN9.) Red-Black trees are not competitive.
  - ➢ Dynamic Finger Search Optimality [CMSS00].
  - ➢ Working-set bound [ST85].
  - ➢ Entropy bound [ST85].

Bibliography:

[ST85] D.D. Sleator, R.E. Tarjan, "Self-adjusting binary search trees," J. ACM 32(3):652-686, 1985. [Original paper on Splay Trees.]

[Tar85] R.E. Tarjan, "Sequential access on splay trees takes linear time," Combinatorica 5(4), pp: 367-378, 1985.

[DHIKP09] E.D. Demaine, D. Harmon, J. Iacono, D. Kane, M. Pătraşcu, "The geometry of binary search trees," 20th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp: 496-505, 2009.

[WDS06] C.C. Wang, J. Derryberry, D.D. Sleator, "O(lg lg n)-competitive dynamic binary search trees," 17th SODA, pp: 374-383, 2006.

[Goo00] M.T. Goodrich, "Competitive tree-structured dictionaries," 11th SODA, pp:494-495, 2000.

[CMSS00] R. Cole, B. Mishra, J. Schmidt, A. Siegel, "On the dynamic finger conjecture for splay trees. Parts I and II," SIAM J. on Computing 30(1), pp: 1-85, 2000.

[Wei06] M.A. Weiss, "Data Structures and Algorithms in C++," 3rd edition, Addison Wesley, 2006.

[GT02] M.T. Goodrich, R. Tamassia, "Algorithm Design: Foundations, Analysis, and Internet Examples," John Wiley & Sons, Inc., 2002.

Exercises

1. Prove Lemma 4.

2. Consider the sequence of $O(n)$ dictionary operations performed by procedure **Test** below.

```
procedure Test(n)
        T ← ∅        (* T is initialized to an empty dictionary *)
        for i ← 1..n  do  Insert (i,T)
        for i ← 1..n  do  Search (i,T)
end
```

Show the following time complexities hold for procedure **Test**:
(a) $\Theta(n \log n)$ time (i.e., at least $\Omega(n \log n)$ and at most $O(n \log n)$ time) if T is a Red-Black tree.
(b) At least $\Omega(n^2)$ time if T is a NaïveSplay tree.
(c) At most $O(n \log n)$ time if T is a Splay tree.
(d) [Hard: Extra Credit and Optional] Only $\Theta(n)$ time if T is a Splay tree.

3. For any positive integers m and n, $n \le m$, let $\Sigma(m,n)$ be the set of all possible sequences s of m dictionary operations applied to an initially empty dictionary, where n is the maximum number of keys in the dictionary at any time during s. We have already seen that executing any sequence $s \in \Sigma(m,n)$ takes at most $O(m \log n)$ time on both Red-Black trees and Splay trees. On the other hand, Splay trees tend to adjust to the pattern of usage. Prove for all m and n as above, there is a sequence $s \in \Sigma(m,n)$ whose execution takes at least $\Omega(m \log n)$ time on Red-Black trees, but at most $O(m)$ time on Splay trees.

4. Consider inserting the n keys $\langle 1,2,\ldots,n \rangle$, in that order, into an initially empty search tree T.
(a) What is the tight asymptotic running time of this insertion sequence if T is a Red-Black tree?
    Describe the structure of the final Red-Black tree as accurately as you can.
(b) Answer the same questions as in (a) if T is a Splay tree.
(c) Consider a worst permutation of the n keys $\langle 1,2,\ldots,n \rangle$, so that if these n keys are inserted into an initially
    empty Splay tree, in the order given by that permutation, the running time is asymptotically maximized.
    Characterize one such worst permutation for Splay trees. What would be the tight asymptotic running time for the insertion sequence according to that permutation?

5. **Split and Join on Splay trees:** These are cut and paste operations on dictionaries. The Split operation takes as input a dictionary (a set of keys) A and a key value K (not necessarily in A), and splits A into two disjoint dictionaries $B = \{ x \in A \mid key[x] \leq K \}$ and $C = \{ x \in A \mid key[x] > K \}$. (Dictionary A is destroyed as a result of this operation.) The Join operation is essentially the reverse; it takes two input dictionaries A and B such that every key in A < every key in B, and replaces them with their union dictionary $C = A \cup B$. (A and B are destroyed as a result of this operation.) Design efficient Split and Join on Splay trees and do their amortized analysis.
   [Note: Split and Join, as defined here, is the same we gave on BST's, 2-3-4 trees, and Red-Black trees.]

6. **Top-Down Splay:** Describe how to do top-down splay and analyze its amortized cost.

7. **[Goodrich-Tamassia, C-3.23, C-3.24, p. 215]** Let T be a Splay Tree consisting of the n sorted keys $\langle x_1 , x_2, \ldots , x_n \rangle$. Consider an arbitrary sequence s of m successful searches on items in T. Let f_i be the access frequency of x_i in s. Assume each item is searched at least once (i.e., f_i 1 for each i=1..n).
   (a) Show that the total execution time for sequence s on the Splay tree starting with T is
   $$O(m + \sum_{i=1}^{n} f_i \log \tfrac{m}{f_i}).$$
   [Hint: Modify the potential function by redefining the weight of a node as the sum of access frequencies of its descendants including itself.]
   (b) Design and analyze an algorithm that constructs an off-line static (i.e., fixed) Binary Search Tree T' consisting of the same items $\langle x_1 , x_2, \ldots , x_n \rangle$, such that depth of item x_i in T' is $O(\log (m/f_i))$.
   Your algorithm should take at most O(n log n) time, and for an extra credit, it should take O(n) time.
   [Hint: Let x_i be the root, where i is the smallest index such that $f_1 + f_2 + \ldots + f_i$ m/2. Recurse on that idea over the subtrees.]
   (c) How do parts (a) and (b) above compare the execution times over the access sequence s on the on-line self-adjusting Splay tree T versus the off-line static binary search tree T'?
   [Hint: You may also consult [CLRS §15.5], AAW, and my EECS3101 Lecture Note 9 and Slide 7 on Knuth's dynamic programming algorithm that constructs the optimal static binary search tree.]

8. **[Goodrich-Tamassia, p. 676] Energy-Balanced Binary Search Tree (EB-BST):**
Red-Black trees use rotations to explicitly maintain a balanced BST to ensure efficiency in the dictionary operations. Splay trees are a self-adjusting alternative. On the other hand, an Energy-Balanced BST explicitly stores a potential energy parameter at each node in the tree. As dictionary operations are performed, the potential energies of tree nodes are increased or decreased. Whenever the potential energy of a node reaches a threshold level, we rebuild the subtree rooted at that node.

More specifically, an EB-BST is a binary search tree T in which each node x maintains w[x] (weight of x, i.e., the number of nodes in the subtree rooted at x, including x. Assume w[nil]=0.) More importantly, each node x of T also maintains a potential energy parameter p[x].

Search, insert and delete operations are done as in standard (unbalanced) binary search trees, with one small modification. Every time we perform an insert or delete which traverses a search path from the root of T to a node v in T, we increment p[x] by 1 for each node x on that search path. If there is no node on this path such that p[x] w[x]/2, then we are done. Otherwise, let x be the highest node in T (i.e., closest to the root) such that p[x] w[x]/2. We rebuild the subtree rooted at x as a completely balanced binary search tree, and we zero out the potential fields of each node in this subtree (including x).
(Note: a binary tree is called completely balanced, if for each node, the sizes of the left and right subtrees of that node differ by at most one.)

(a) Show the rebuilding can be done in time linear in the size of the subtree that is being rebuilt. That is, design and analyze an algorithm that converts an arbitrary given BST into an equivalent completely balanced BST in time linear in the size of that tree.
(b) Prove that if x and y are any two sibling nodes in an EB-BST, then w[y] ≤ 3w[x] +2.
[Hint: observe the potential energy of their parent node.]
(c) Using part (b), prove that the maximum height of any n-node EB-BST is O(log n).
(d) Prove that the amortized time of any dictionary operation on any n-node EB-BST is O(log n).

END