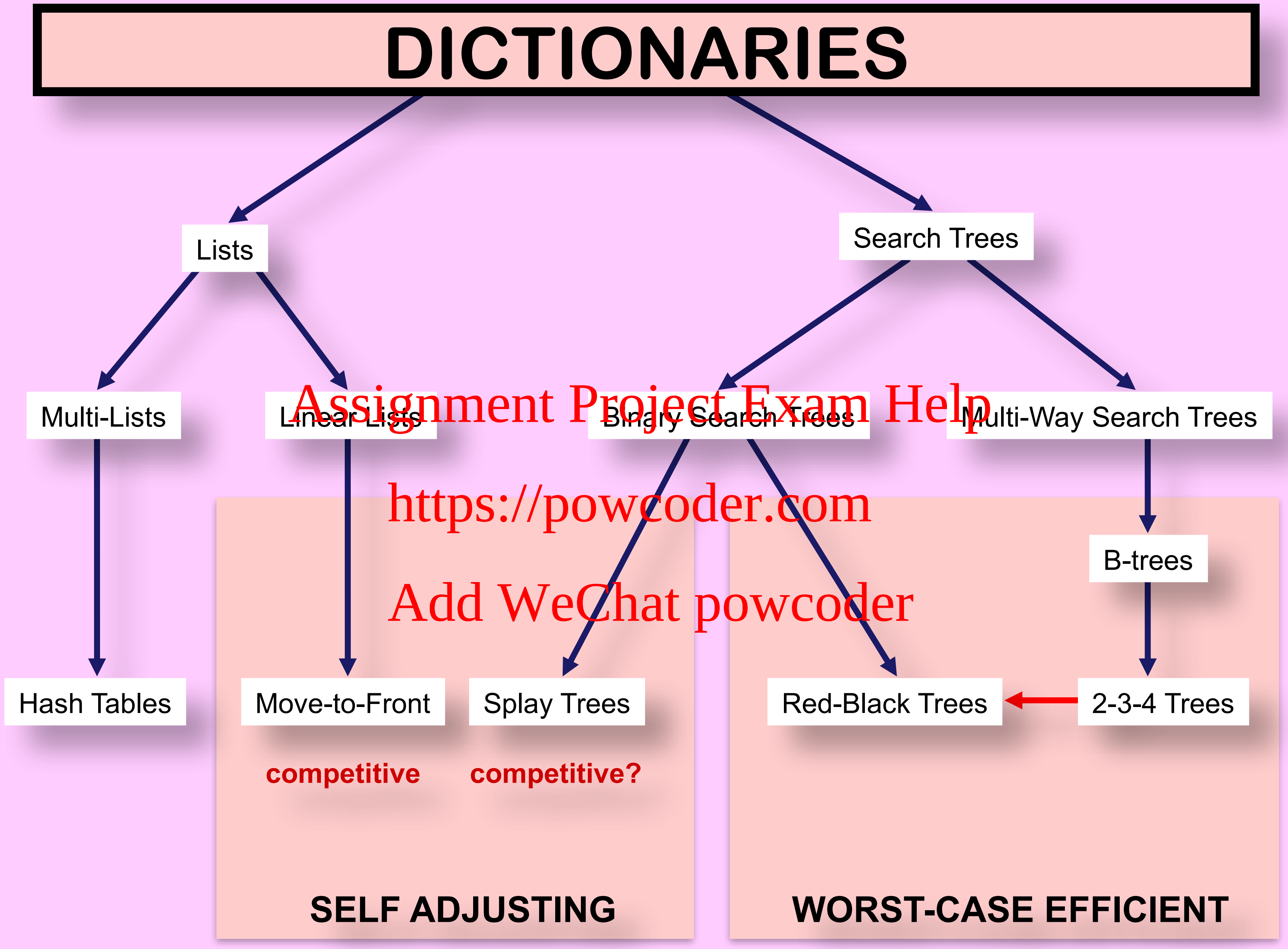
# EECS 4101/5101


Prof. Andy Mirzaian



# Red-Black Tree

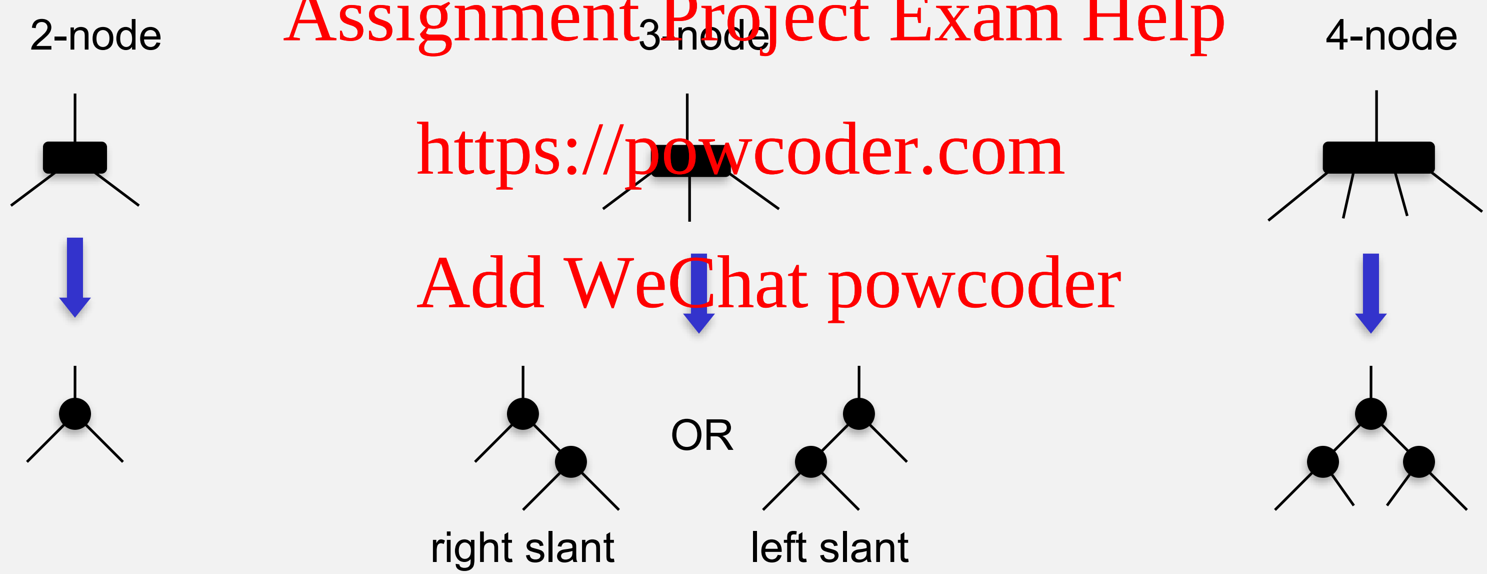

DICTIONARIES
Lists
Search Trees
Multi-Lists
Linear Lists
Binary Search Trees
Multi-Way Search Trees
B-trees
Hash Tables
Move-to-Front
Splay Trees
Red-Black Trees
2-3-4 Trees
competitive
competitive?
SELF ADJUSTING
WORST-CASE EFFICIENT

# References:

- [CLRS] chapter 13
- AAW animation

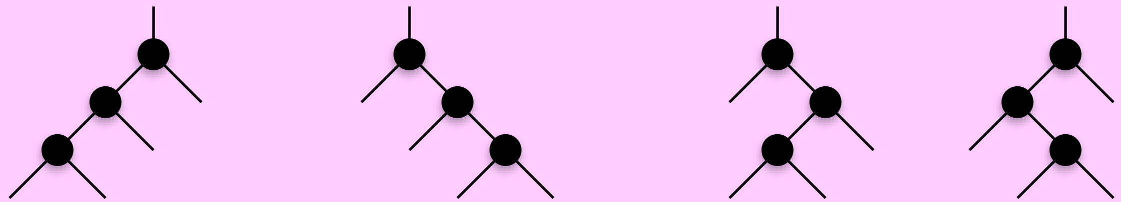
# Binary Search trees from 2-3-4 trees

- 2-3-4 trees are perfectly balanced ( height: ½ $\log(n+1) \leq h \leq \log(n+1)$ ) search trees that use 2-nodes, 3-nodes, and 4-nodes.
- We can transform a 2-3-4 tree to an $O(\log n)$ height BST by replacing each 3-node and 4-node by a small-clustered BST with 2 or 3 binary nodes.
- Dilemma: How do we distinguish "node clusters"?





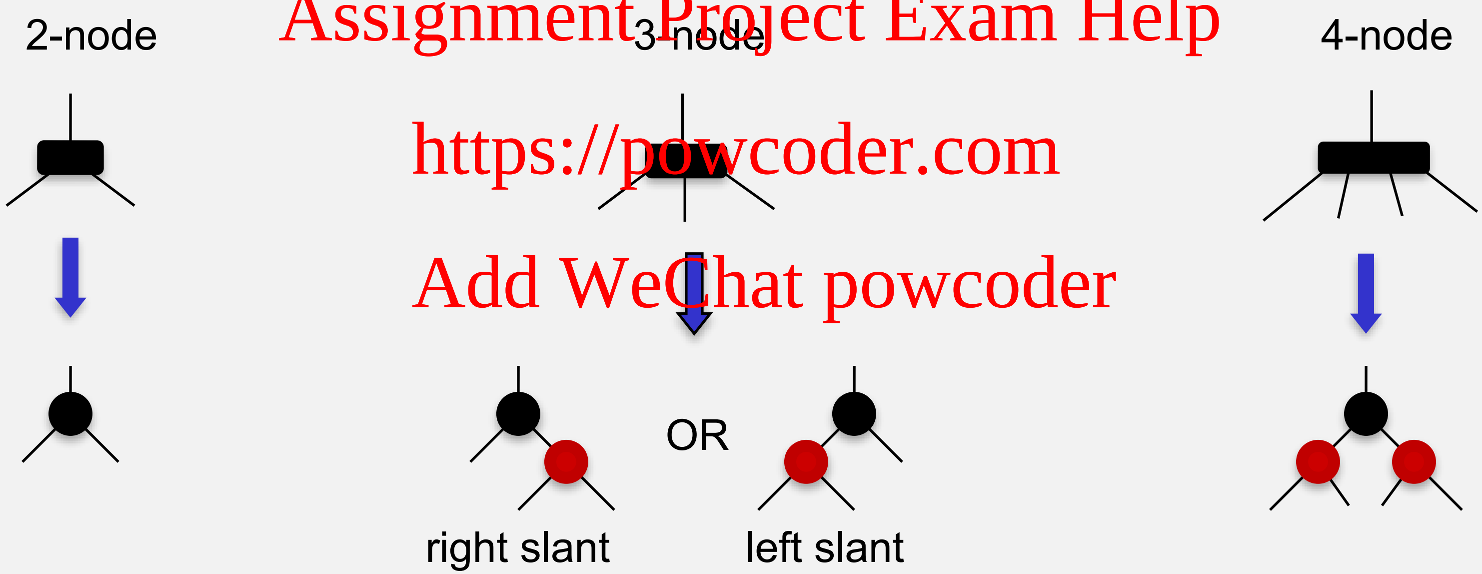

Disallowed 4-node clusters:



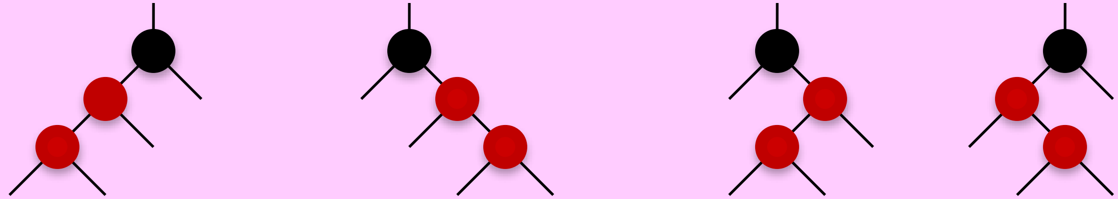
# Red-Black trees from 2-3-4 trees

- Although the resulting BST has height O(log n), it loses the “node cluster” information and renders the 2-3-4 tree algorithms obsolete. We use a “node cluster” colour-coding scheme to resolve this.
- Convention: each cluster top level is black (including external nodes),
  lower levels within each cluster are red.
  So, any red node is clustered within its parent cluster.





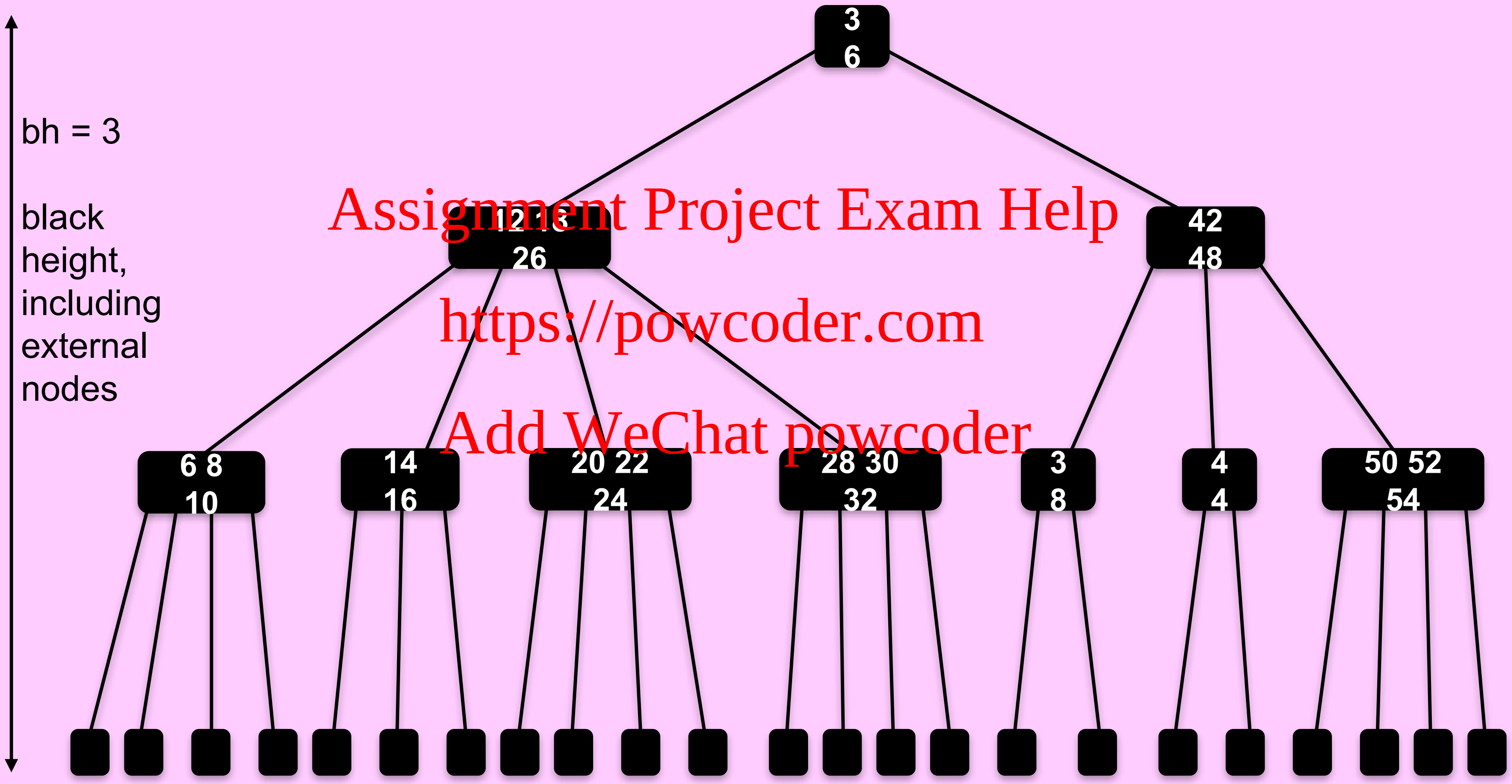

Disallowed 4-node clusters:



# **Example:** 2-3-4 tree

bh = 3

black height, including external nodes






$$\frac{1}{2}\log(n+1) \le bh \le \log(n+1).$$

# Transformed Red-Black tree

bh = 3

black height, including external nodes




$$\text{height } h \le 2\,bh - 1 \le 2\log(n+1) - 1.$$

# **Definition:** Red-Black tree

**DEFINITION:** T is a Red-Black tree if it satisfies the following:

1. T is a Binary Search Tree with a red/black colour bit per node.

2. Every red node has a black parent. ( ⇒ root is black.)

3. By convention we assume all external nodes are black.

4. All external nodes have the same number (namely, bh) of black proper ancestors.

# Implementing Operations

Access operations:

SEARCH
MINIMUM
MAXIMUM
PREDECESSOR
SUCCESSOR

Use the BST algorithms without change.
(Simply ignore the node colours.)

Worst-case time: O(h) = O(log n).



Update Operations:

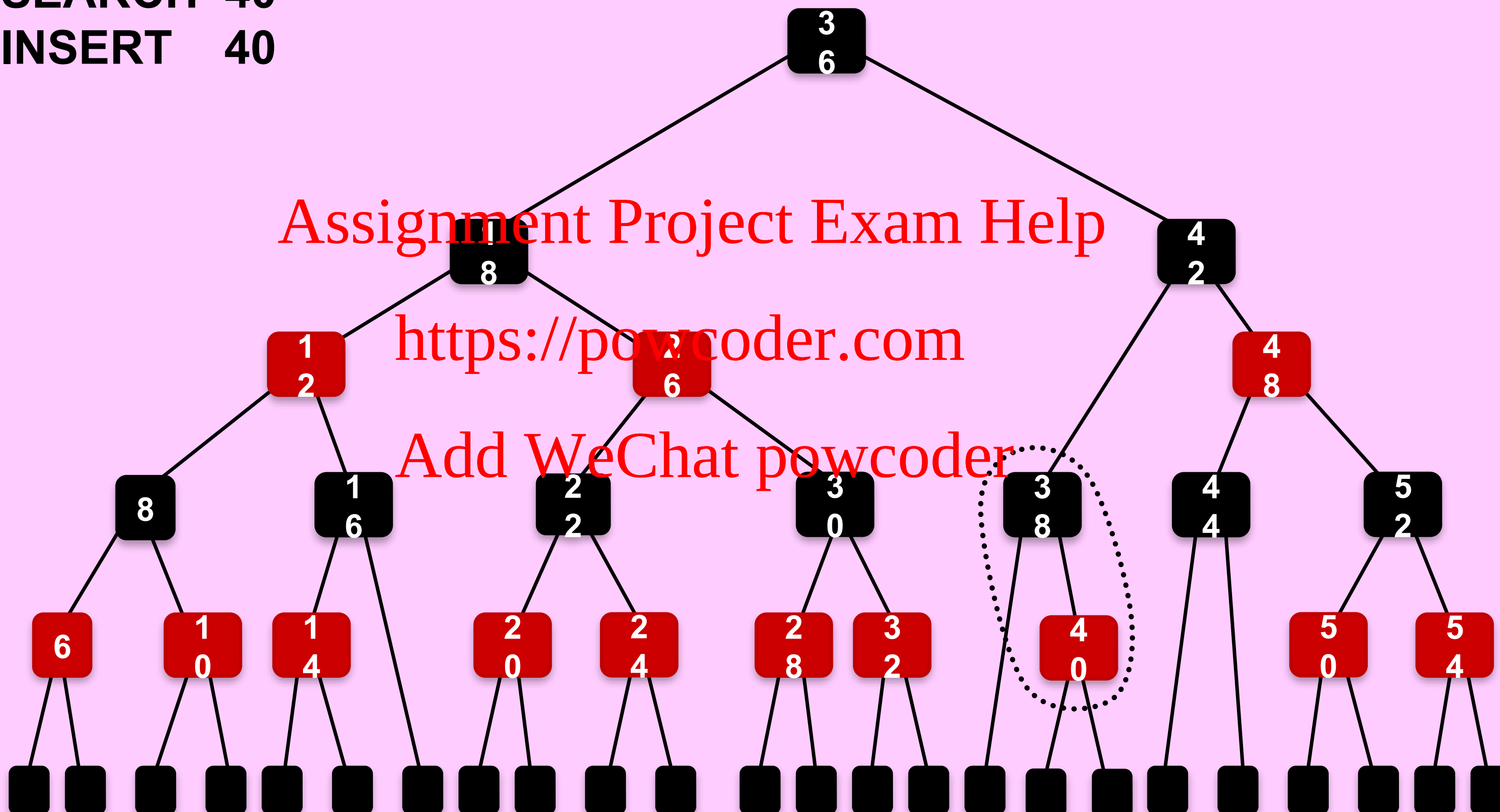
INSERT
DELETE

Simulate the 2-3-4 tree algorithms
as shown next.

Worst-case time: O(h) = O(log n).

# Example operations

**SEARCH 40**
**INSERT 40**

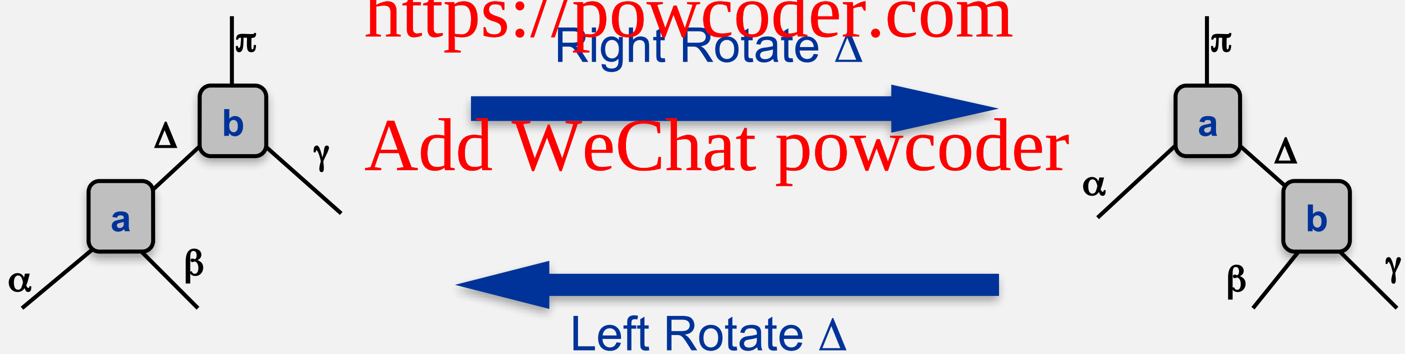


# Local Restructuring

INSERT and DELETE need the following local operations that take O(1) time each:

- Node Colour Switch: red ↔ black
- Link Rotation






**FACT:** Rotation preserves INORDER sequence of the tree nodes and changes slant of link Δ.

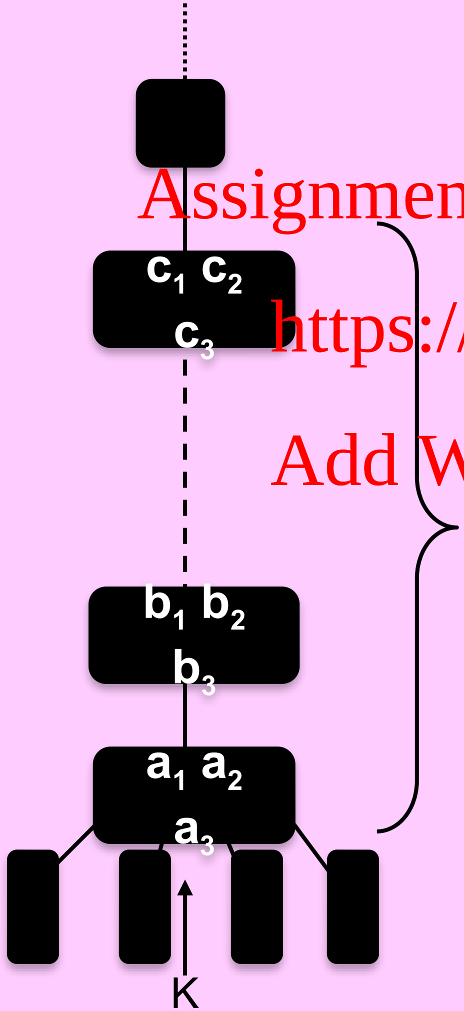
# Bottom-Up INSERT part 0:4

## Bottom-Up INSERT (K,T)

Simulate 2-3-4 tree bottom up insertion.








When inserting K at the bottom of the tree, repeatedly split 4-nodes upwards until you either split the root, or reach a 2-node or a 3-node.



Simulate this on the Red-Black tree.

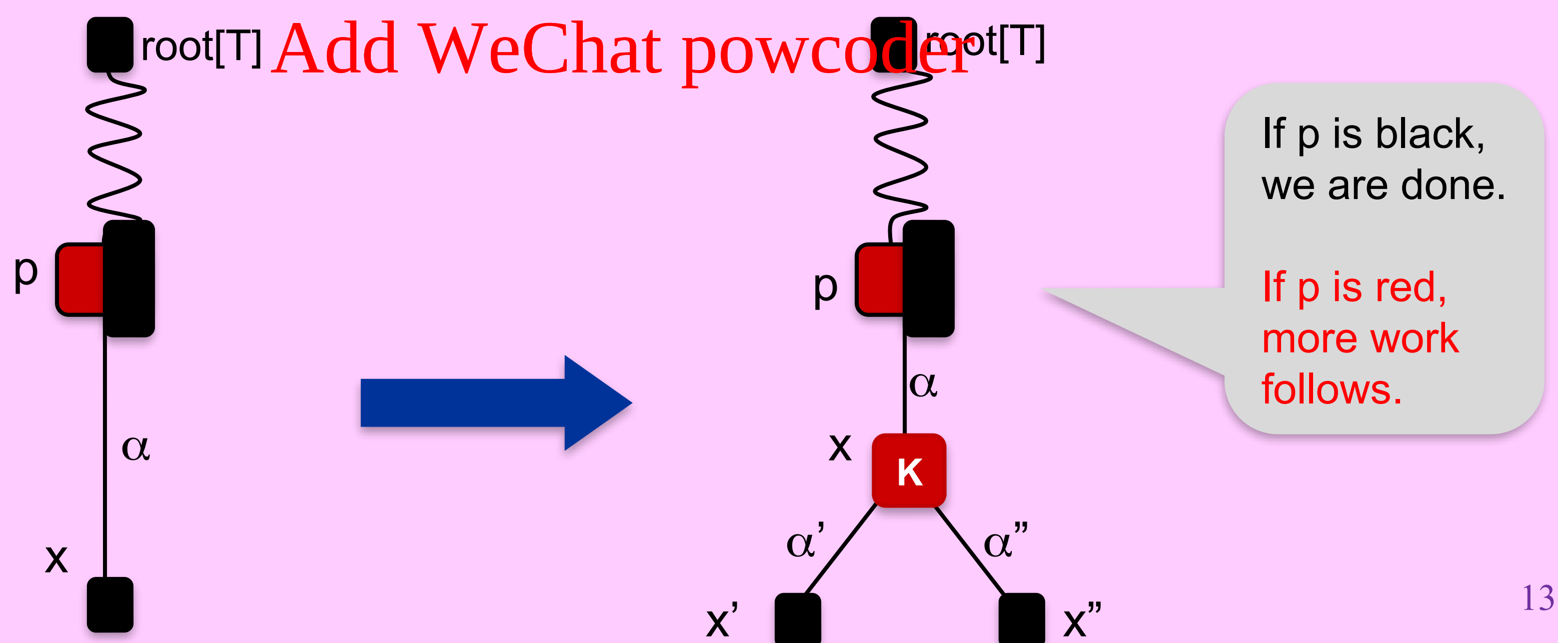
# Bottom-Up INSERT part 1:4

## Bottom-Up INSERT (K,T)

Step 1: Follow the search path of K down Red-Black tree T.
If K is found, return. Otherwise, we reach a black external node x. Convert x into a red leaf and store K in it.

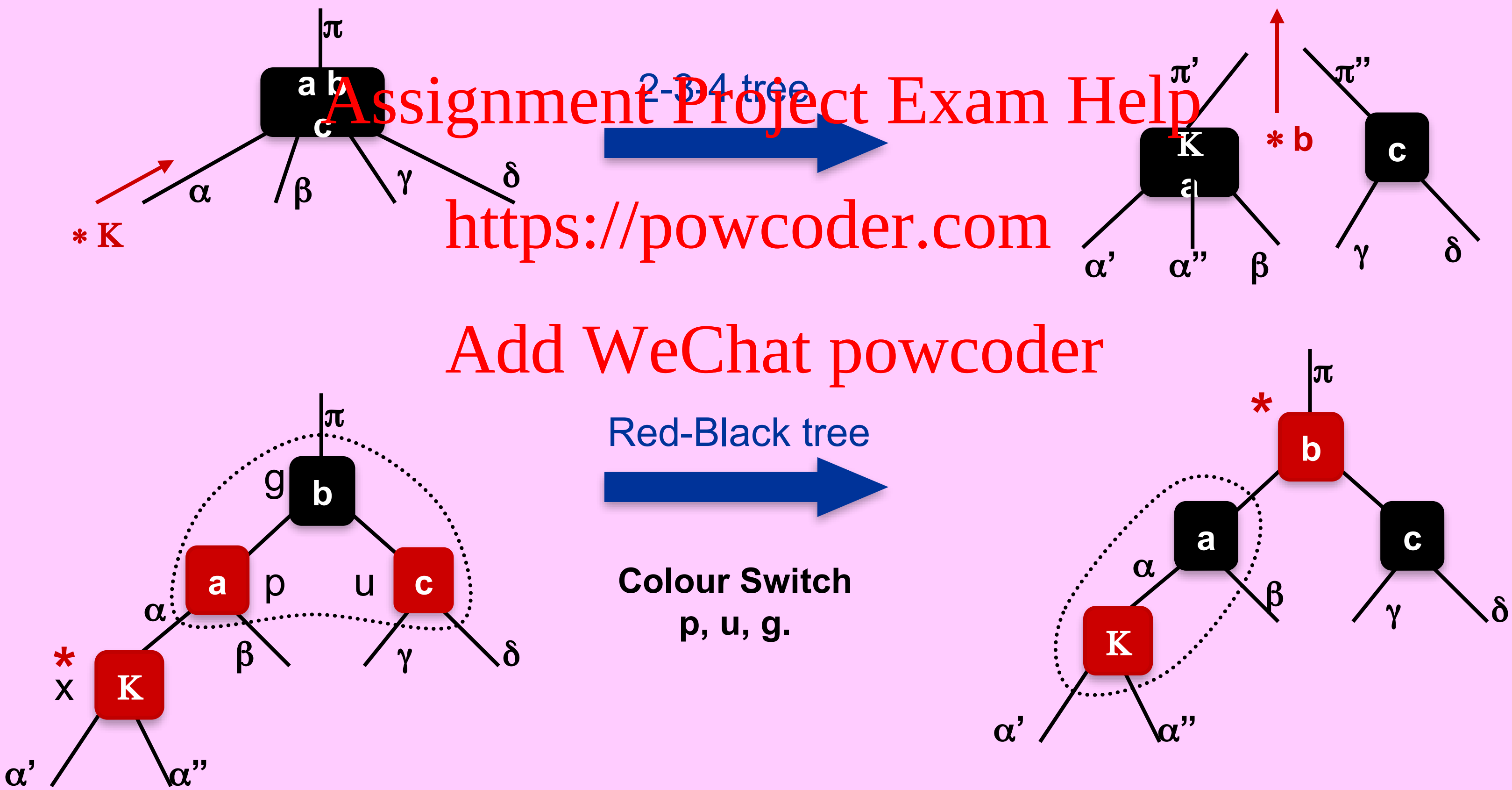
- At the end of the algorithm we will re-colour the root black even if it might have temporarily become red.
- K is red means x splits into x' & x", & K is inserted into the parent cluster.
- Now T satisfies all 4 properties of RB-trees except property 2: red x might have a red parent p. We need to climb up the tree to fix up.





# Bottom-Up INSERT part 2a:4

Step 2: While parent is part of a “4-node cluster”, keep splitting it and promote its middle key up. (May repeat many times.)



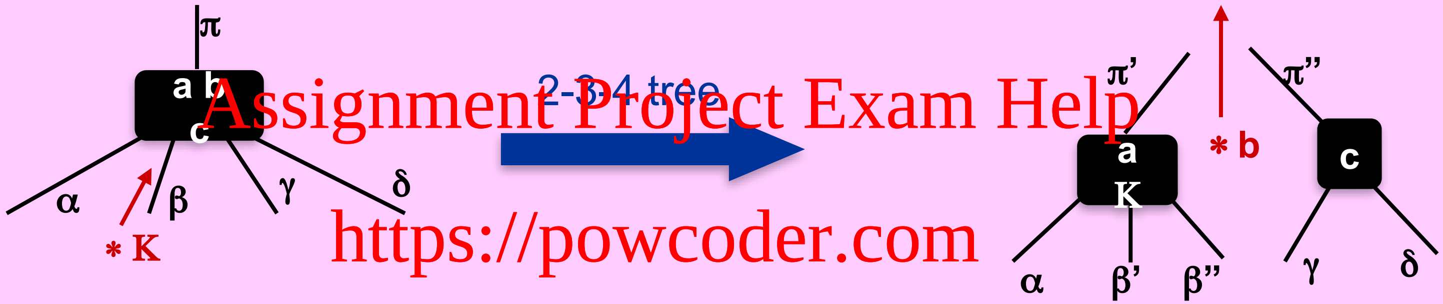


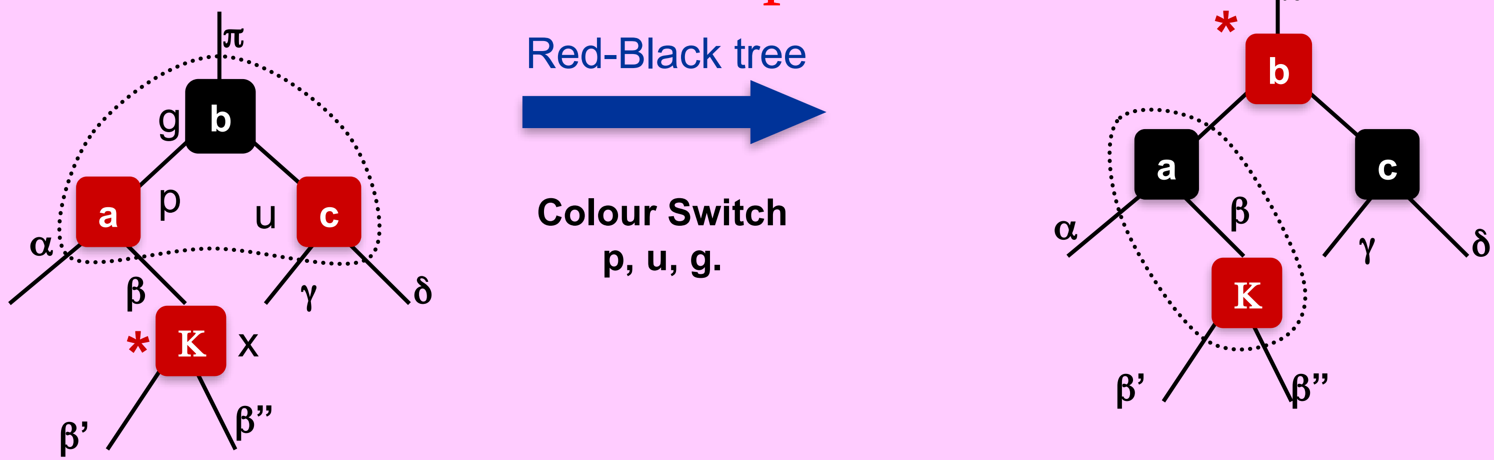

[ K up from δ is symmetric ]

# Bottom-Up INSERT part 2b:4

Step 2: While parent is part of a "4-node cluster", keep splitting it and promote its middle key up. (May repeat many times.)



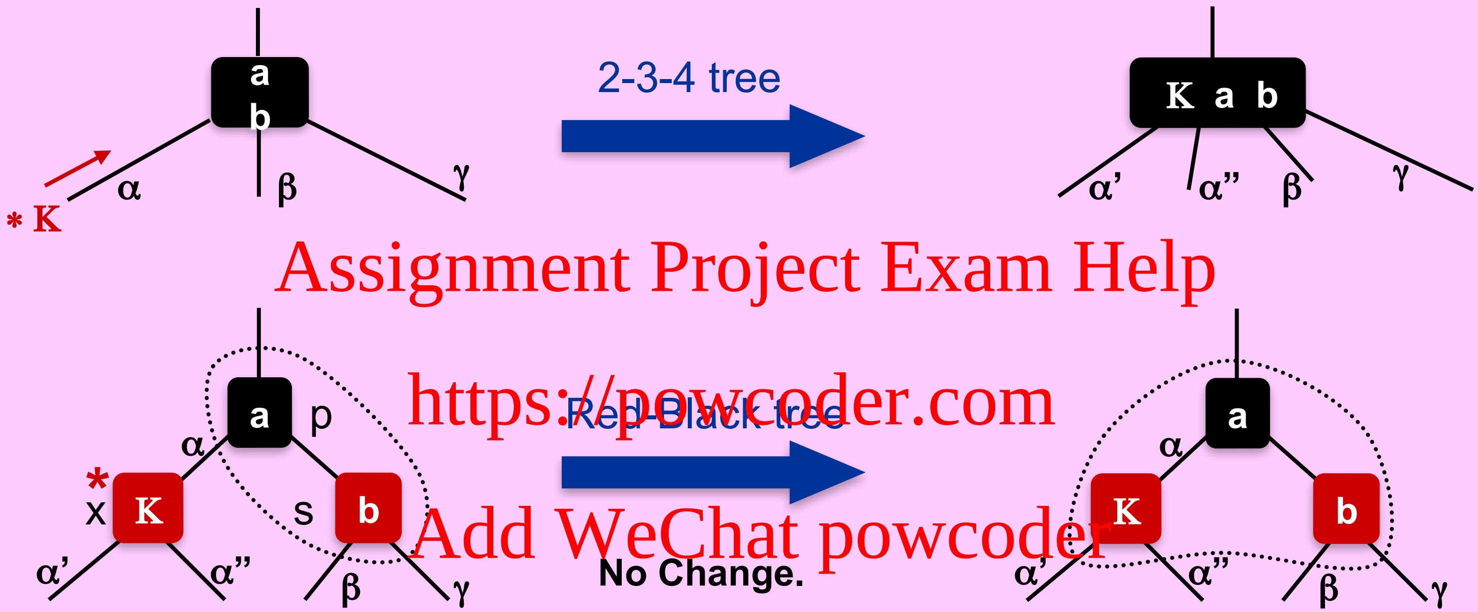



[ K up from γ is symmetric ]

# Bottom-Up INSERT part 3a:4

Step 3a: Have reached a 3-node cluster.





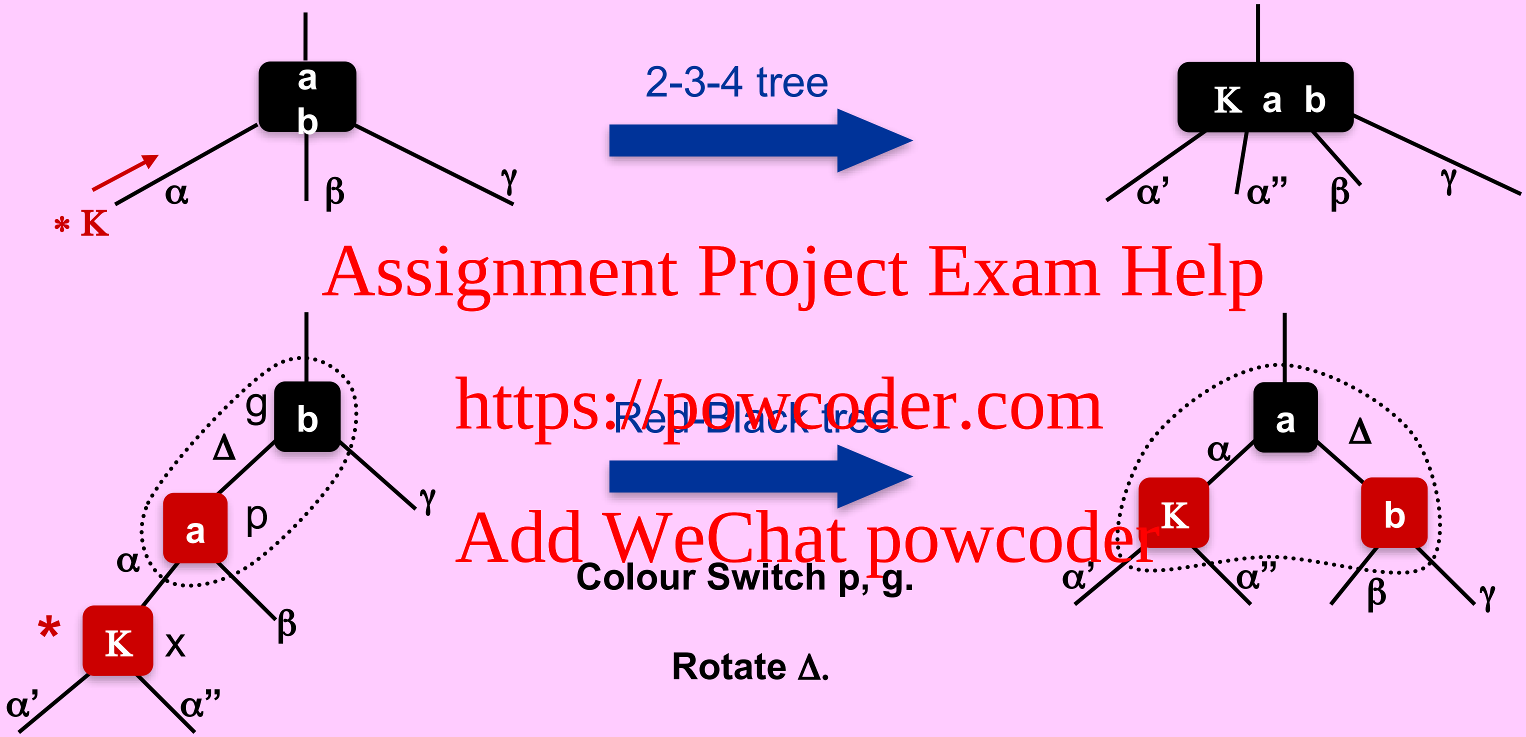

Opposite Slant

**[ K up from γ is symmetric ]**

TERMINAL CASE

# Bottom-Up INSERT part 3b:4

Step 3b: Have reached a 3-node cluster.






Same Slant
Zig-Zig

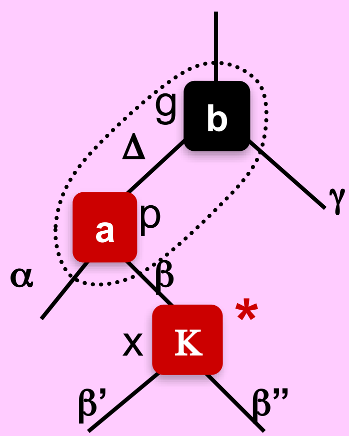
[ K up from γ is symmetric ]

TERMINAL CASE

# Bottom-Up INSERT part 3c:4

Step 3c: Have reached a 3-node cluster.

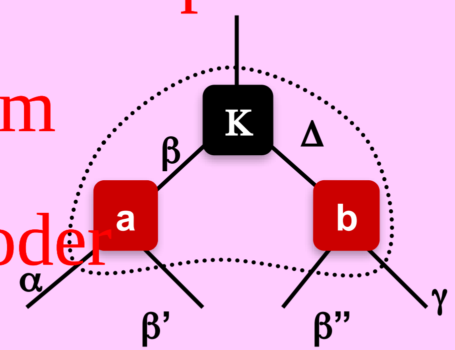






**Colour Switch x, g.**
**DoubleRotate(x,p,g):**
**i.e., Rotate β**
**then Rotate Δ.**

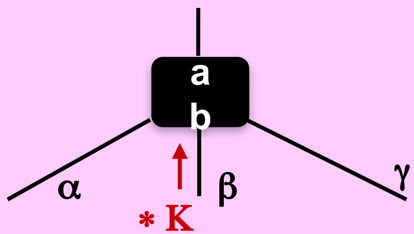



LR Zig-Zag

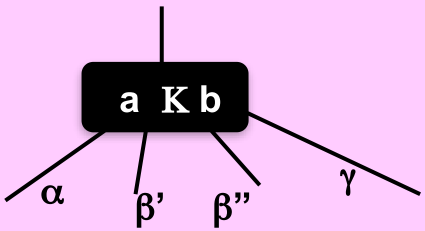
TERMINAL CASE

# Bottom-Up INSERT part 3d:4

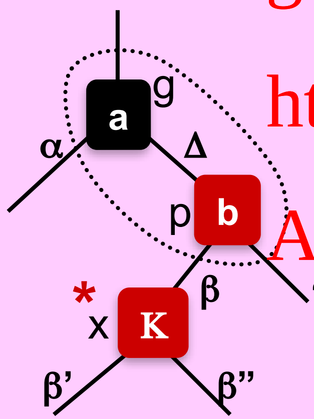
Step 3d: Have reached a 3-node cluster.




2-3-4 tree



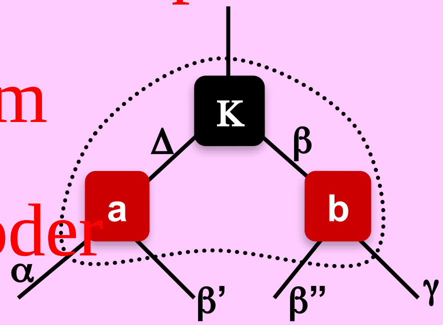



Red-Black tree



**Colour Switch x, g.**
**DoubleRotate(x,p,g):**
**i.e., Rotate β**
**then Rotate Δ.**

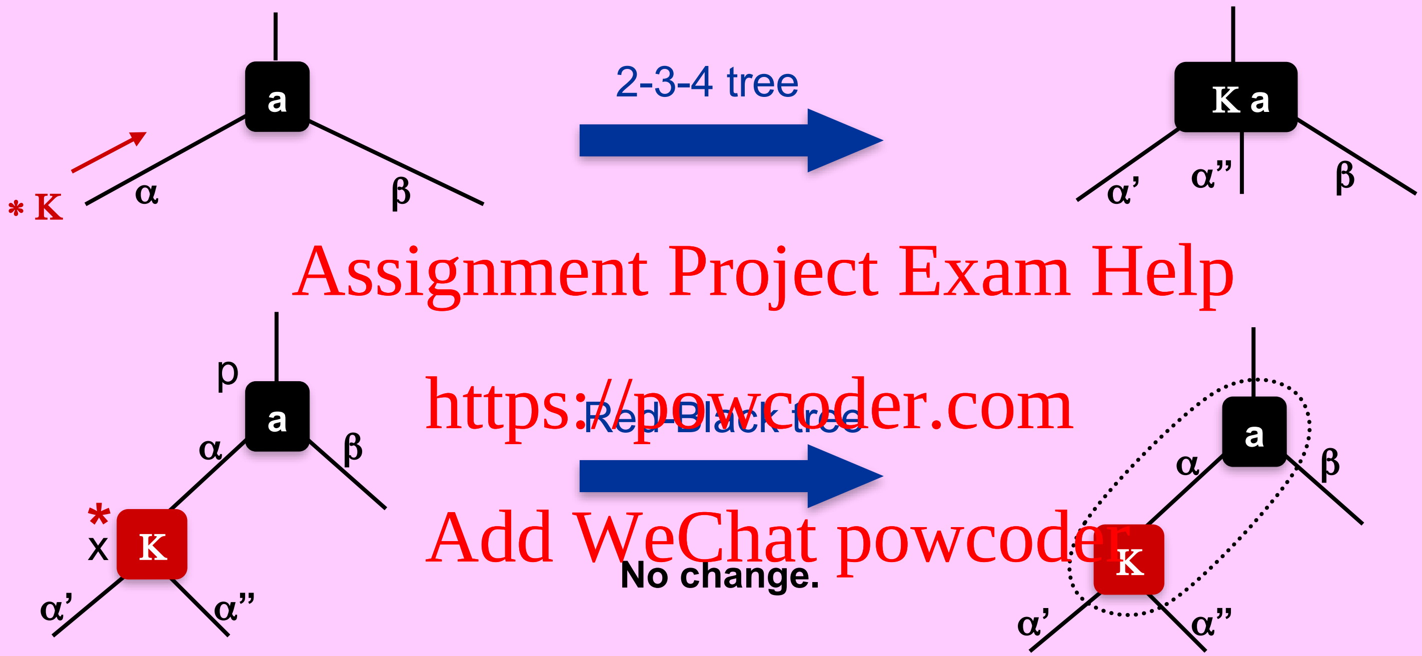



RL Zig-Zag

TERMINAL CASE

# Bottom-Up INSERT part 3e:4

Step 3e: Have reached a 2-node cluster.






[ K up from β is symmetric ]

TERMINAL CASE

# Bottom-Up INSERT part 4:4

Step 4: Have reached nil (parent of root).




**This last step always resets root colour to black.**
**Black-height increases by one if root was temporarily red.**

END

# Bottom-Up INSERT summary

**Bottom-Up INSERT (K,T)**

Step 1: Follow the search path of K down Red-Black tree T.
**If** K is found **then return**
Otherwise, we have reached a black external node x.
Convert x into a red leaf and store K in it.



Step 2: **while** (p, u)=red (* x=red, parent in 4-node cluster *)
**do** SwitchColour(p,u,g); x ← g **end-while**



Step 3: **if** p=red & u=black (u possibly external) (* x=red *)
**then case:** (* 3-node turns into unbalanced 4-node *)



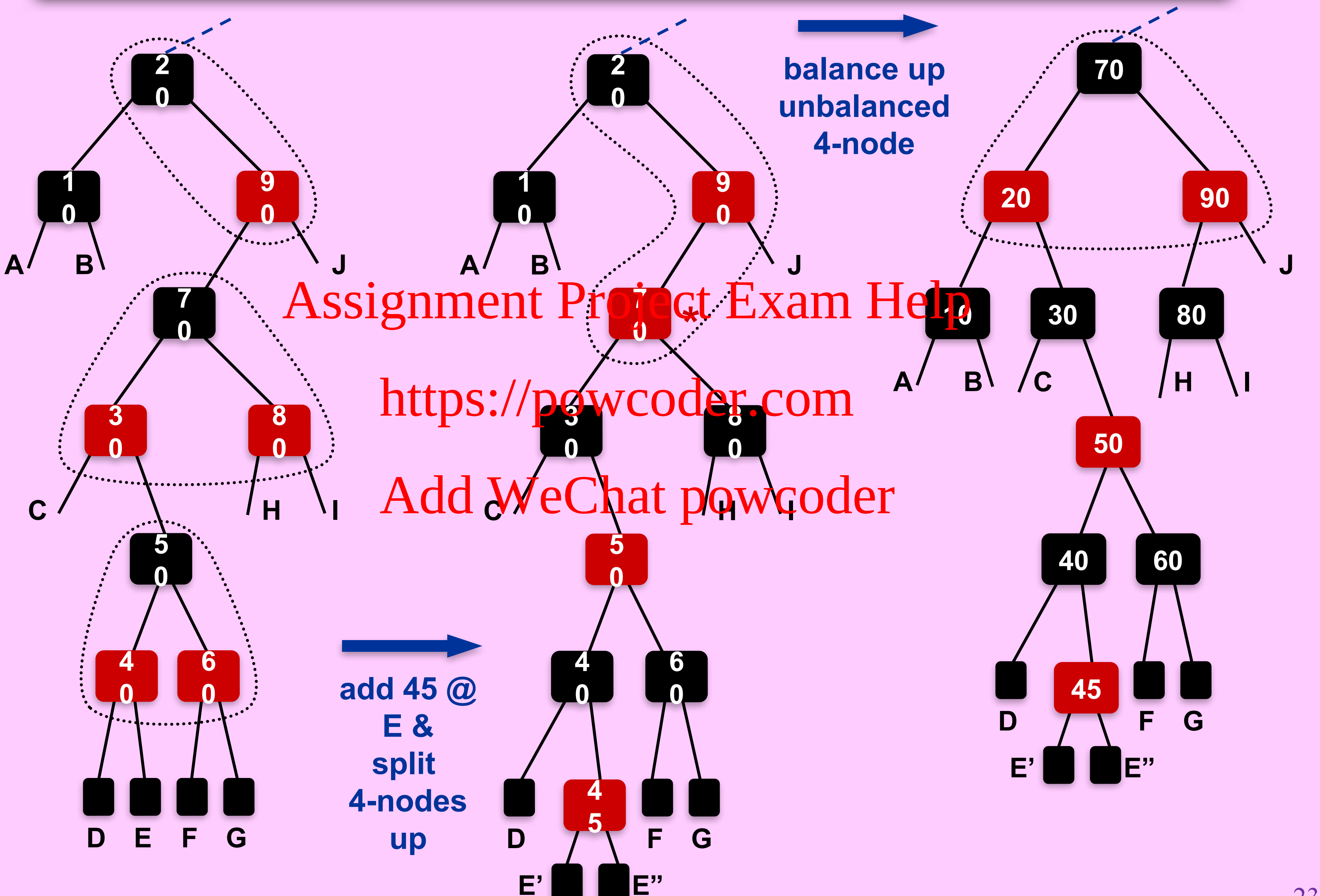
[Zig-Zig (x,p,g)]: SwitchColour(p,g); Rotate(p,g)
[Zig-Zag(x,p,g)]: SwitchColour(x,g); DoubleRotate(x,p,g)
**end-case**

Step 4: **if** root[T] = nil **then** root[T] ← x
colour[root[T]] ← black

**end**

**x = current node, p = parent of x, u = uncle of x, g = grand parent of x.**

# Example: Bottom-Up Insert 45



# INSERT & DELETE

- Top-Down Insert (exercise).
- Bottom-Up Delete (see [CLRS]).
- Top-Down Delete:

  While going down the search path, maintain LI:
  [LI: current node x is either root[T] or x is red or x has a red child.]
  With LI, "splice-out" can finish the task in O(1) time.



FACT 1: Bottom-Up Insert makes at most 2 rotations.



This occurs when after the "4-node splitting loop" a 3-node turns into a zig-zag 4-node. It needs 2 rotations to turn it into a balanced 4-node cluster.



FACT 2: Bottom-Up Delete makes at most 3 rotations.

FACT 3: Top-Down Insert & Delete may make up to $\Theta(\log n)$ rotations. Explain why.

This makes them unsuitable for persistent data structures.

# Bibliography:


- Red-Black Trees: [R. Bayer, E.M. McCreight, "Symmetric binary B-trees: Data Structure and maintenance algorithms," Acta Informatica, 1:290-306, 1972.]
- Colour coded Red-Black Trees: [L.J. Guibas, R. Sedgewick, "A dichromatic framework for balanced trees," in Proceedings of the $19^{th}$ Annual Symposium on Foundations of Computer Science (FOCS), pp: 8-21, IEEE Computer Society, 1978.]
- AA-trees: [A. Andersson, "Balanced search trees made simple," in Proceedings $3^{rd}$ Workshop on Algorithms and Data Structures (WADS), in Lecture Notes in Computer Science LNCS709, pp: 60-71, Springer-Verlag, 1993.]
- Scapegoat trees: [I. Galperin, R.L. Rivest, "Scapegoat trees," in Proc. $4^{th}$ ACM-SIAM Symp. on Discrete Algorithms (SODA), pp: 165-174, 1993.]
- Treaps: [R. Seidel, C.R. Aragon, "Randomized search trees," Algorithmica, 16:464-497, 1996.]
- AVL trees: [G.M. Adel'son-Vel'skiĭ, E.M. Landis, "An algorithm for the organization of information," Soviet Mathematics Doklady, 3:1259-1263, 1962.]
- Weight balanced trees: [J. Nievergelt, E.M. Reingold, "Binary search trees of bounded balance," SIAM J. of Computing, 2(1):33-43, 1973.]
- K-neighbor trees: [H.A. Mauer, Th. Ottmann, H.-W. Six, "Implementing dictionaries using binary trees of very small height," Information Processing Letters, 5(1):11-14, 1976.]

# Exercises

1. **[CLRS, Exercise 13.2-4, p. 314] Rotation Sequence:**
Show that any BST holding a set of n keys can be transformed into any other BST holding the same set of keys using O(n) rotations. [Hint: first show that at most n-1 right rotations suffice to transform the tree into a right-going chain.]

2. **[CLRS, Exercise 13.2-5, p. 314] Right Rotation Sequence:**
We say that a BST $T_1$ can be ***right-converted*** to BST $T_2$ if it is possible to obtain $T_2$ from $T_1$ via a series of right rotations. Give an example of two trees $T_1$ and $T_2$, holding the same set of keys, such that $T_1$ cannot be right-converted to $T_2$. Then show that if a tree $T_1$ can be right-converted to $T_2$, it can be done so using $O(n^2)$ right rotations, where n is the number of keys in $T_1$.



3. **Top-Down Insert & Delete:** Design and analyze O(log n) time top-down Delete and Insert on red-black trees. [Hint: simulate top-down 2-3-4 tree procedures & use the hints given in these slides.]



4. **[CLRS, Exercise 13.4-7, p. 330]** Suppose we use bottom-up insert and delete on red-black tree T. Suppose that a key K is inserted into T and then immediately deleted from it. Is the resulting tree always the same as the initial tree? Justify your answer.



5. **Split and Join on Red-Black 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 and analyze efficient Split and Join on red-black trees. [Note: Definition of Split and Join here is the same we gave on BST's and 2-3-4 trees, and slightly different than the one in Problem 13-2, pages 332-333 of [CLRS].]

6. **Red-Black tree Insertion sequence:** Bottom-Up Insert integers 1..n one at a time in increasing order into an initially empty red-black tree in O(n) time total. [Hint: This is related to exercise 4 in the Slide on B-trees. Keep a pointer to the largest key node.]

**7.** **RB-Balance:** We say a binary tree T is **RB-balanced** if it satisfies the following property:

***RB-balance:*** *for every node x in tree T, the length of a longest path from x to any of its descendant external nodes is at most twice the length of a shortest path from x to any of its descendant external nodes.*

(a) Show that every red-black tree is an RB-balanced Binary Search Tree.

(b) Now we want to prove that the converse also holds. Let T be an arbitrary RB-balanced BST with n nodes. We want to show (algorithmically) that it is always possible to colour each node of T red or black to make it a red-black tree.



[Note that we make no structural change to T other than colouring its nodes.]
Design and analyze an O(n) time algorithm to do the node colouring to turn T into a red-black tree. [You may assume there is space for a colour bit in each node of T.]



(c) Carefully prove the correctness of your algorithm in part (b).



**8.** **BST to RB-tree Conversion:** Given an n-node arbitrary BST, design and analyze an O(n) time algorithm to construct an equivalent red-black tree (i.e., one that contains the same set of keys).

**9.** **Range-Search Reporting in RB-tree:** Let T be a given red-black tree. We are also given a pair of key values a and b, $a < b$ (not necessarily in T). We want to report every item x in T such that $a \le key[x] \le b$. Design an algorithm that solves this problem and takes O(R+log n) time in the worst case, where n is the number of items in T and R is the number of reported items (i.e., the output size).

10. **Restricted Red-Black Tree:** We define a Restricted Red-Black Tree (RRB-tree) to be a standard Red-Black Tree with the added structural constraint that every red node must be the right child of its parent. So, every left child is black. (In comparison with 2-3-4 trees, this indicates that we have no 4-node clusters, and every 3-node cluster is right slanted.) We want to show that it is possible to maintain such a structure while performing dictionary operations efficiently. Let T be an arbitrary n-node RRB-tree.

(a) Obtain tight lower and upper bounds on height of T as a function of n.

(b) Show how to perform the dictionary insert operation on T efficiently. Make sure you consider all possible cases in the algorithm. What is its worst-case running time as a function of n?

(c) Consider a leaf node x in T. (Note x is not an external node.) What are possible structures of the subtree rooted at the sibling of x?

(d) Using your answer to part (c), show how to perform the dictionary delete operation on T efficiently. Make sure you consider all possible cases in the algorithm. What is its worst-case running time as a function of n?

# END