Solutions: ICS 311, Spring 2016, Problem Set 06, Topics 10B & 11

#1. Peer Credit Assignment

1 Point Extra Credit for replying

You should have named your group partners and allocated 6 points total across them for class 2/29 and 3/02 (section 1). If you did not, please email the TA with your points assignment.

#2 Red-Black Tree and (2,4)-Tree Deletion (12 pts)

Preliminary Comments

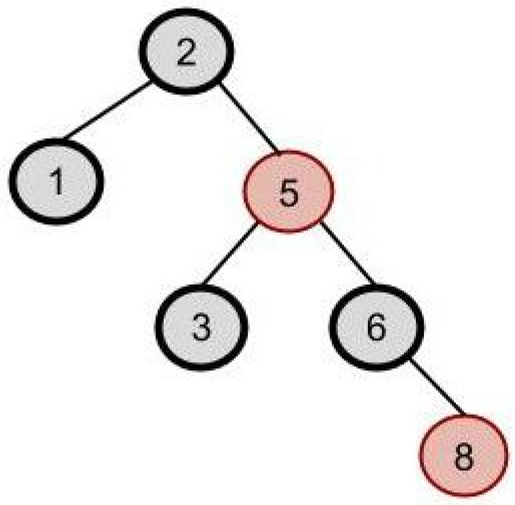
In this problem we delve deeply into the CLRS code for tree deletion. The lecture notes were based on Goodrich & Tamassia's textbook, because they show the correspondence of RBTs to 2-4 trees, which makes the former easier to understand as balanced trees. The CLRS version differs somewhat. You will need to read the CLRS text to answer this question.

The cases for insertion are similar between G&T and CLRS, but the terminology differs (e.g., what the letters w, x, y, and z refer to). The cases for deletion differ: G&T have 3 while CLRS have 4! Be careful because there are mirror images of every situation (e.g., is the double black node a left child or a right child?): G&T and CLRS may be describing the same situation with mirror image graphs.

The top level methods in CLRS for RB-INSERT (p. 315) and RB-DELETE (p. 324) essentially do binary search tree (BST) insertion and deletion, and then call "FIXUP" methods to fix the red-black properties. Thus they are very similar to the BST methods TREE-INSERT (p. 294) and TREE-DELETE (p. 298). The real work specific to RBTs is in these fixup methods, so we will focus on them in these questions, but you should also study the top level methods to understand them as BST methods.

Problems

(a) RBT as 2-4 Tree (2 pts) Draw the 2-4 tree that corresponds to the RBT shown below.



(b) Deletion (4 pts): Delete key **2** from the red/black tree shown above, and show the deletion in the (2,4) representation.

- Show every state of the RBT tree, including after the BST-style deletion and after each case applied by RB-Delete-Fixup. Clearly identify the colors of the nodes.
- Also show the state of the 2-4 tree for each of these RBT states.
- If a double black node occurs (node x in CLRS), clearly identify which node it is.
- For each state change, identify **both** G&T case(s) from the web notes and the CLRS case(s) from the textbook that are applied in **each** of your steps.
- Your final diagram should show the RBT after RB-Delete-Fixup and the (2,4) tree representation that results.

(c) More Deletion (6 pts): Delete key **1** from the **initial** red/black tree shown above (NOT the tree that results from (b)), showing all steps as specified above.

Solutions: See Google Drawing AT END OF THIS PDF FILE

#3 Red-Black Tree Height (6 pts)

(a) (4 pts) What is the largest possible number of internal nodes (those with keys) in a red-black tree with black height k? What is the height of the corresponding 2-4 tree? Prove your claims.

(b) (2 pts) What is the smallest possible number of internal nodes (those with keys) in a red-black tree with black height k? What is the height of the corresponding 2-4 tree? Prove your claims (you may use Lemma 13.1).

Solution:
(a) (4 pts) Maximum number of internal nodes is achieved when RB-Tree is full, with layers of red and layers of black nodes alternating. Note, this is possible under the insertion rules. Thus, the maximum number of internal nodes is 2^{2k}-1. Since black nodes of a RB-Tree are the only ones that define the structure of the 2-4 tree, the height of the corresponding 2-4 tree is *k*-1 (height is defined as one less than the number of levels, and NIL leaf nodes are not part of the 2-4 trees).

(b) (2 pts) The number of internal nodes is at least 2^k - 1. The inductive proof is the proof of Lemma 13.1 in CLRS. The height of the corresponding 2-4 tree is also *k*-1.

#4 Red Nodes in Red-Black Tree (2 pts)

Consider a red-black tree formed by inserting *n* nodes with RB-Insert. Prove that if *n* > 1, the tree has at least one red node. *Hint:* Nodes are red when inserted (line 16 of RB-Insert). Show that if *n* > 1 one node must be red after RB-Insert-Fixup is complete.

(It is not sufficient to say that the second node inserted will be red. You must show that some node remains red under all possible insertion sequences and the transformations that result.)

Solution: The first node inserted will be black (property 2, ensured by line 16 of RB-Insert-Fixup). The second node inserted will initially be red (line 16 of RB-Insert). A node can change from red to black only by virtue of being the root, or in clauses of RB-Insert-Fixup. A root may become red only if (1) the inserted node is the only one in the tree or (2) in case 1 of CLRS where the inserted node's grandparent is the root node and it gets colored red (line 7). In case of (1), n = 1, and is not applicable for us. In case of (2) the inserted node is left to be red, i.e. at least one node in the RB-Tree is red. In other two cases of CLRS, RB-Insert-Fixup leaves at least one node red (line 13), right after it colors the parent of *z* black. Since the parent of *z* is colored black, RB-Insert-Fixup will exit the **while** loop after this, leaving at least one node red.

#5 Sorting larger numbers (10 pts)

Suppose we want to sort *n* integers in the range 0 to n^4-1 in *O(n)* time.

a. (2 pts) Show that Counting-Sort is not an option by analyzing the runtime of Counting-Sort on this data. *Hint:* Identify the value of *k* and invoke the existing analysis.

b. (4 pts) Show that unmodified Radix-Sort is not an option by analyzing the runtime of Counting-Sort on this data. *Hint:* Identify the value of *k* for each call to Counting-Sort. Then identify the value of *d,* and invoke the existing analysis.

c. (6 pts) CLRS section 8.3 states that "we have some flexibility in how to break each key into digits" and gives a relevant Lemma. Using this as a hint, describe a modified Radix-Sort that would sort this data in O(*n*) time and do the analysis to show that this is the correct runtime.

Solution:
We cannot apply any of the O(n) time sorting algorithms directly because all of them make assumptions about the input, as illustrated by the first two questions:

a. (2 pts) Counting-Sort is O(k+n), where k is the maximum value of an integer and n is the number of integers in the input. In this problem, $k = n^4$-1, so the runtime is O(n^4 + n -1) = O(n^4).

b. (3 pts) Radix-Sort calls the stable sort (such as Counting-Sort) on one digit at a time. This avoids the problem in (a), as k=9 in base 10 or k=1 for base 2. However, it takes d = lg(n^4) = 4lg n digits to represent a number up to n^4-1. (With base 10, the constant differs but it is still O(lg n) digits.) Therefore, the runtime of such radix sort would be O(d (n+k)) = O(n log n).

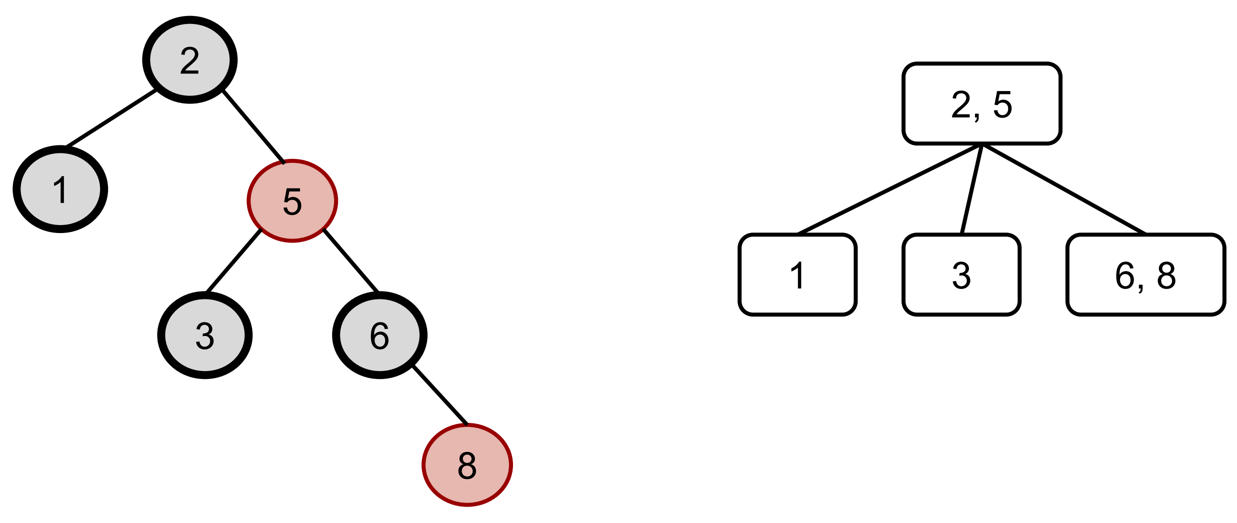
c. (5 pts) Lemma 8.4 in the book states:

Lemma 8.4 Given n b-bit numbers and any positive integer $r \leq b$, Radix sort correctly sorts
these numbers in Θ((b/r) (n + 2^r)) time if the stable sort it uses takes Θ(n + k)
time for inputs in the range 0 to k.

Our numbers consist of 4 log n bits, i.e. b = 4 log n. If we let r = log n, then each number is in the range 0 to k = n-1 and counting sort is stable and takes Θ(n + n) = Θ(n) time. Then Radix sort correctly sorts these numbers in Θ((4 log n / log n) (n + $2^{\log n}$)) = Θ(4(2n)) = Θ(n) time.

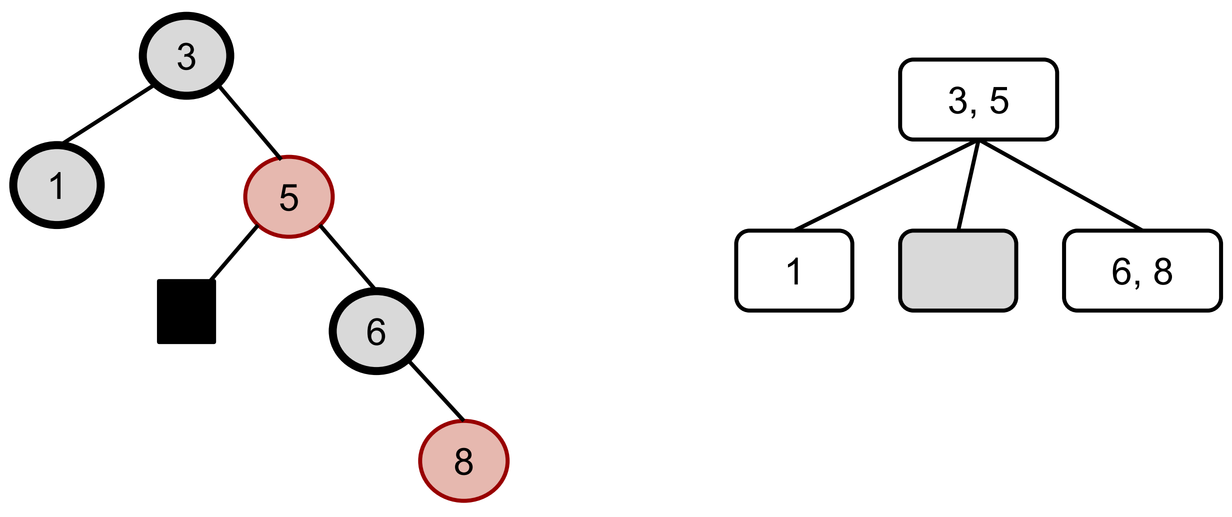
In other words, we treat each number as a 4-digit number, where each digit is presented base-*n*, i.e. each digit is in radix *n*. Another way of seeing this is as follows. Each number in the range [0… n^4-1] requires 4 log n bits to be represented in binary. We view each group of log *n* bits (out of 4 groups) comprising a single number. Thus, each such radix-*n* digit (group of log n binary digits) takes value between 0 and *n*-1. Then we can apply radix sort, and sort each of 4 radix-*n* digit using counting sort. The runtime is O(d*n) = O(4*n) = O(n).

(a) RBT as 2-4 Tree

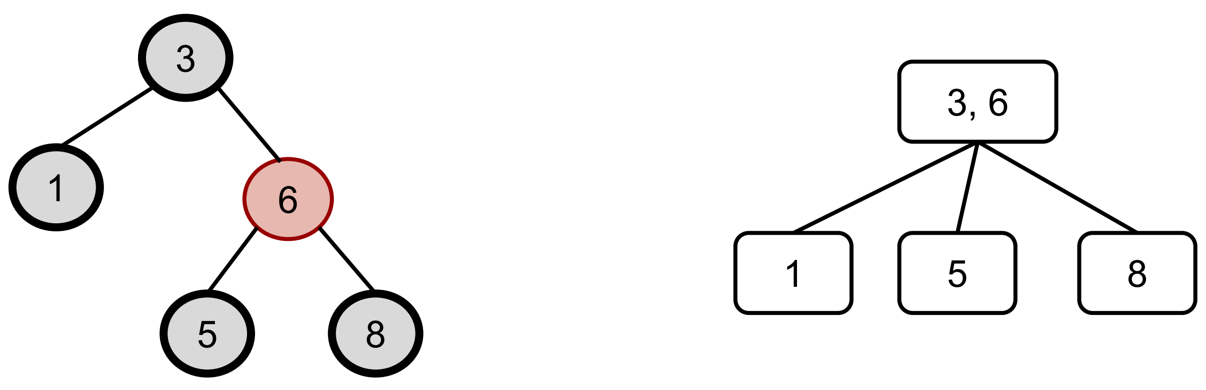


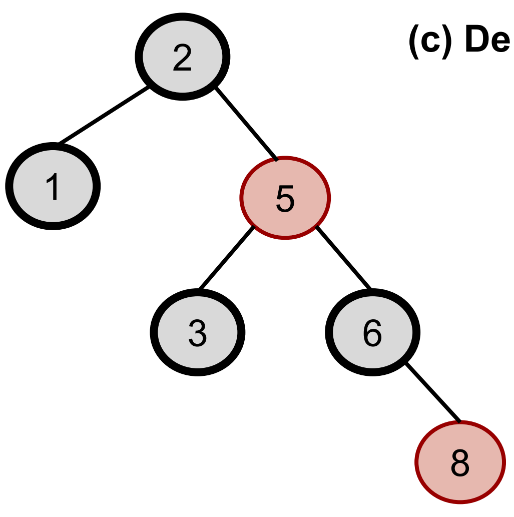
(b) Delete 2

BST deletion: replace 2 with successor 3, which was black, so we get underflow (double black).

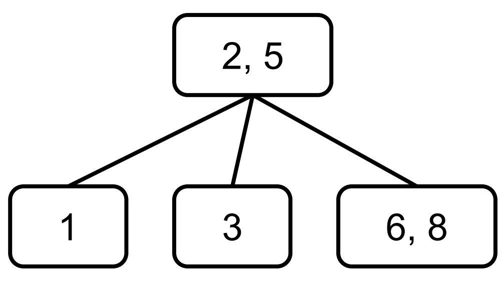


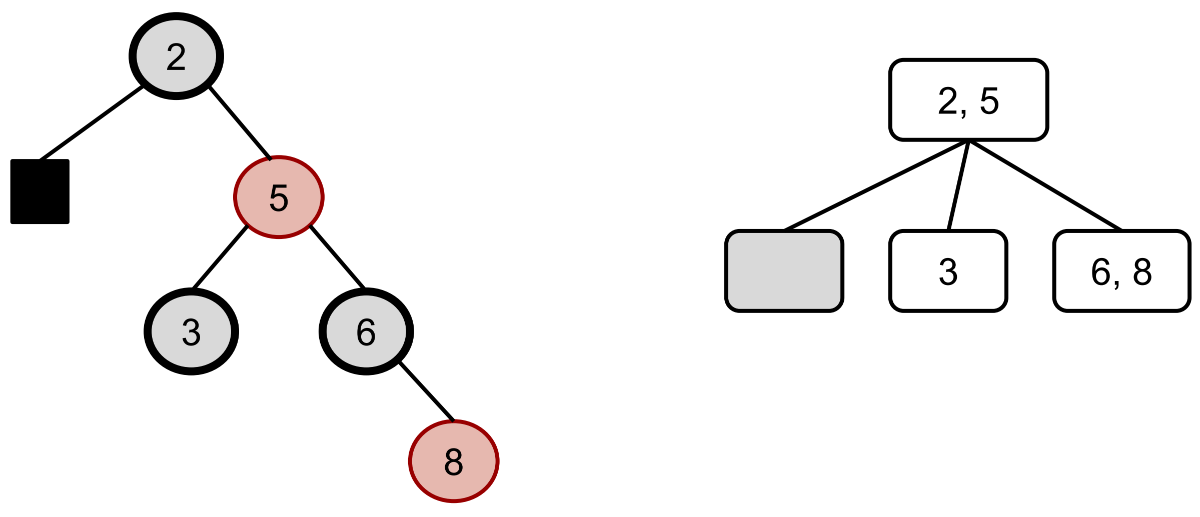
G&T Case 1: y is black and has red child: Restructure (2,4 transfer)
CLRS case 4: x's sibling is black and has red right child: Same result



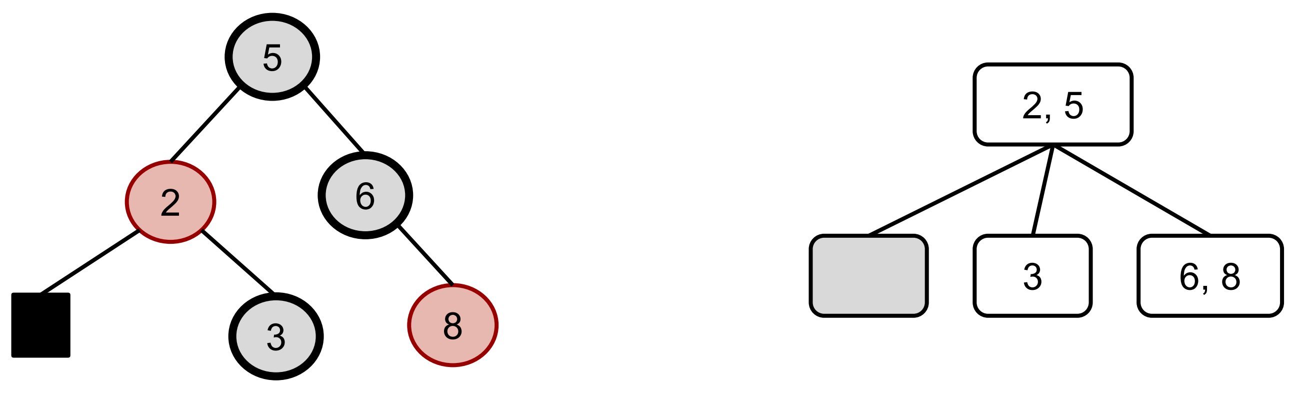


(c) Delete 1

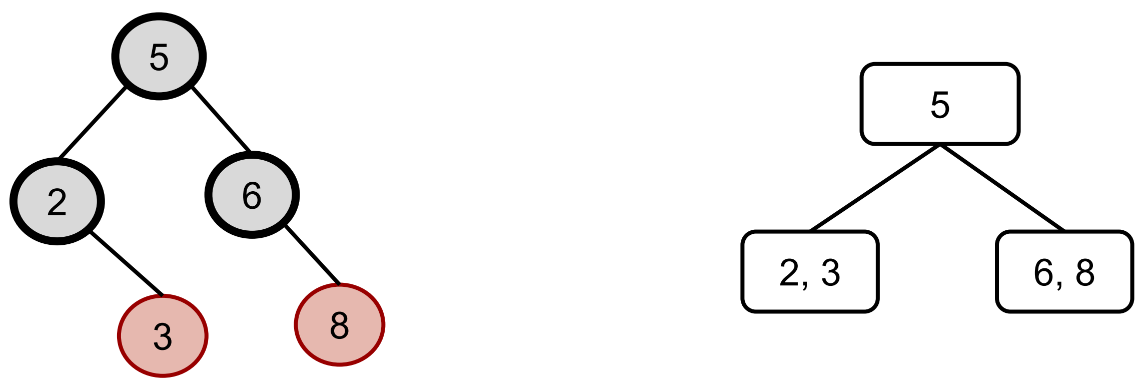




G&T Case 3: y is red, adjustment so another case applies
CLRS: Case 1: x's sibling is red. Change colors and left rotation. Case 2, 3, or 4 applies.



G&T Case 2: sibling black and children both black: recolor (2-4 fusion)
CLRS Case 2: x's sibling is black with black children: recolor



5
2
6
3
8
5
2, 3
6, 8