1. Sorting Larger Numbers (10 pts)

Suppose we have n integers in the range $0 + n^4 - 1$, and we want to sort them 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 analysis that was presented in the textbook or lecture notes.

It is given that the value that we wish to obtain is $O(n)$ time. Based on the algorithm provided in CLRS for Counting-Sort, it is understood that the run-time analysis is $O(k+n)$, as k is the given range of the input parameter, n. We know that we have a given n number of integers, and the range of integers in this problem is from $0 + n^4 - 1$. By substituting these values in for the correct parameters of Counting-Sort, it is given that the final run-tie will be $O(n^4 + n)$, where n^4 is the range, and n is again, the numerical value of integers. This verifies that Counting-Sort is indeed not an optimal solution for this problem.

b. (3 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. This is not the same as in the previous problem. Then identify the value of d, and invoke the analysis that was presented in the textbook or lecture notes.

It is given that the run-time analysis of Radix-Sort will take $\theta(d(n + k))$ time.
Since the call to Counting-Sort is identified as "k", we know that k will be the range of potential values that each digit can be. If it is given that the stable Counting-Sort runs in a $\theta(n + k)$ time, the analysis can be invoked that $n^4 - 1$ is the range of values that will be taken in as k. When substituting into the original equation, we want to prove that this will run in $O(n)$, the optimal time.

$$O(log_k n)(n + k) = O(nlog_k n)$$
$$O(nlog_k n) \neq O(n)$$

$\therefore$ It is proven that this unmodified Radix-Sort is not an option.

c. (5 pts) CLRS states that "we have some flexibility in how to break each key into digits", and prove a relevant Lemma 8.4. Use this as a hint, describe a modified Radix-Sort that would sort this data in $O(n)$ time and use Lemma 8.4 to show that this is the correct runtime.

From CLRS, pg. 199:

Lemma 8.4
Given n b-bit numbers and any positive integer $r \leqslant b$, RADIX-SORT correctly sorts these numbers in $\theta(\frac{b}{r})(n + 2^r)$ time if the stable sort it uses to take $\theta(n + k)$ time for inputs in the range 0 to k.

We know that from our original subproblem, the maximum amount of the range of values is given as $n^4 - 1$. Therefore, we know that from our original analysis:

$$d = 4logn = b$$

By using Lemma 8.4, if we know that $r = logn$ and $b = 4logn$, we can substitute these values into the given Lemma as: $logn \leqslant 4logn$. Radix-Sort originally takes the time of $\theta(d(n+k))$, so by substituting these values into the Lemma, we know that $d = \lceil \frac{b}{r} \rceil = \lceil \frac{4logn}{logn} \rceil = 4$. If k is given as the range of which we must traverse the input, we know that the range is from $0to2^t - 1 = 0to2^{logn} = k$. We also know that n is uniformly still the number of inputs, so by substituting the information, we know that the total run time is $\theta(\frac{4logn}{logn})(n + 2^{logn}) = \theta(4n)$. By dropping all constants and simplifying this, we get $\theta(n)$.

2. Red-Black Tree and (2,4)-Tree Deletion (20 pts)

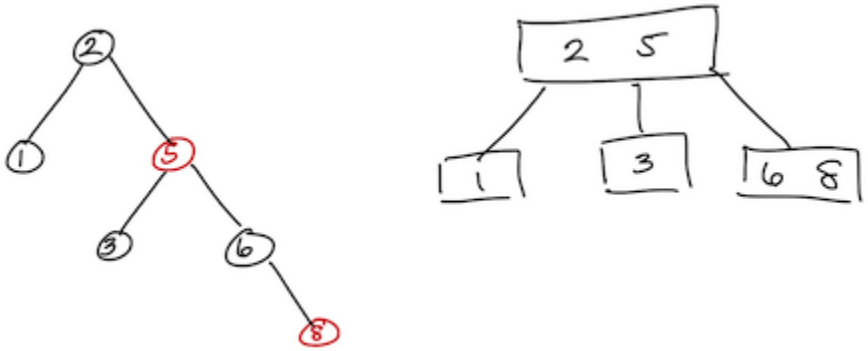
Preliminary Comments
In this problem we delve deeply into the CLRS code for tree deletion. The lecture notes were based on Goodrich Tamassia's textbook, because they show the correspondence of RBTs to 2-4 trees, which makes the former easier to understand as balanced trees. The CLRS version differs somewhat.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

The cases for insertion are similar between GT and CLRS, but the terminology differs (e.g., what the letters w, x, y, and z refer to). The cases for deletion differ: GT 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?): GT and CLRS may be describing the same situation with mirror image graphs.

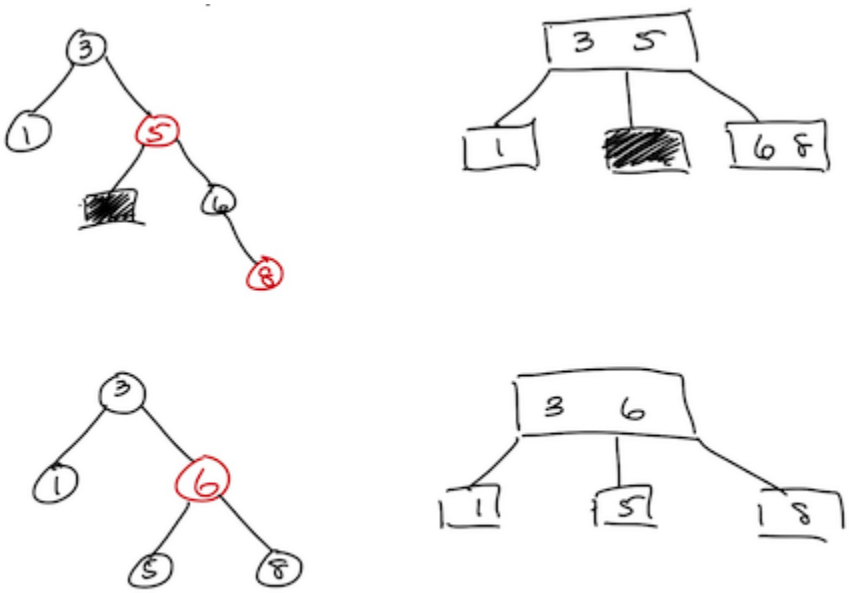
The top level methods in CLRS for RB-INSERT (p. 315) and RB-DELETE (p. 324) essentially do binary search tree (BST) insertion and deletion, and then call "FIXUP" methods to fix the red-black properties. Thus they are very similar to the BST methods TREE-INSERT (p. 294) and TREE-DELETE (p. 298). The real 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

You may want to use the Google Drawing template provided in the document Problem-Set-06-Red-Black-Tree-Deletion-Start. Then convert the results into images to insert here.
(a) RBT as a 2-4 Tree (2 pts) Draw the 2-4 tree that corresponds to the RBT shown below (replace the ? box)



(b) Deletion (6 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. As above, you'll have the rbt on the left and the 2-4 tree on the right.
If a double black node occurs (node x in CLRS), clearly identify which node it is.
For each state change, identify both GT 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.

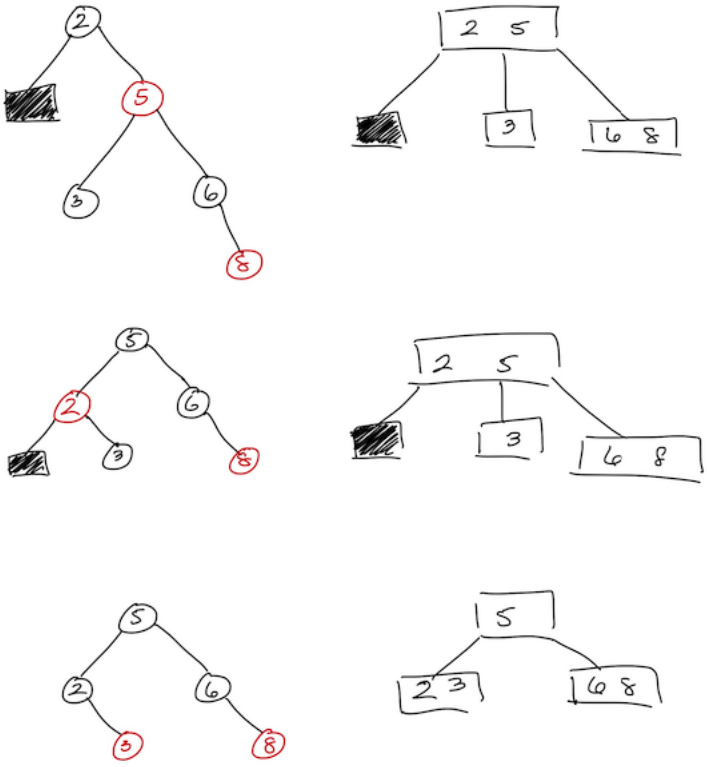


First Image:
When deleting the key 2 from the original red-black tree, to preserve the integrity of the structure, 3 becomes the root of the tree, and an empty node is categorized as the left subtree

of 5. This results in an underflow.

Second Image:
To further preserve the integrity of the red-black tree, and to preserve the height of it as such, 5 is inserted into this empty node, and 6 becomes its parent. Since 6 is a red node, we know that both of its children must be black. We know that because a sibling is black, and has a right child that is red, this structure must be corrected and rotated.

(c) More Deletion (12 pts): Delete key **1** from the **initial** red/black tree shown above in: (a) (NOT the tree that results from (b)). Show all steps as specified above.



First Image:
When deleting the key 1 from the original red-black tree (as given in part a), this results in an empty node occurring in the left sub-tree of 2. This results in an underflow within the tree.

Second Image:
Since there is no child in the left sub-tree of two, this must be restructured to maintain the integrity of the properties of the tree. Since the sibling of this node is red, this means that it is rotated to the left.

Third Image:
Since the sibling of the node is black, and the children are black, the final structuring of this red-black tree must be recolored. It is given that each new node that is being created will be red, so the final nodes of the tree will be red. The final structure maintains the properties of the red-black tree, by using successor to replace the keys after deletion.

3. Red-Black Tree Height (5 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.

It is given in our notes that the height of a corresponding red-black tree from a black root is defined as $2^{bh(x)} - 1$. If there is a red-black tree with a black height of k, this can be substituted in this expression as a red-black tree with a height of $2^k - 1$. Given that this red-black tree is otherwise complete, and satisfies the conditionals of being a red-black tree, the height "k" within this tree is defined when this tree is complete. Therefore, the tree of a given height of 2k, the number of internal nodes can be expressed as $2^{2k} - 1$.

When expressing the height of the corresponding 2-4 tree, because it is given that the height of the red-black tree is denoted as k, we know that the height of the corresponding 2-4 tree must be the same as a red-black tree, since the black node of its root characterizes its height.

(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 should use Lemma 13.1 to make this easier.)
The following Lemma, taken from pg. 309 of CLRS is as follows:

Lemma 13.1
A red-black tree with n internal nodes has height at most $2lg(n + 1)$.

As this lemma continues, in the proof continued on pg. 209 of CLRS, this states that the subtree of any node x will be denoted as $2^{bh(x)} - 1$ internal number of nodes. Since we are given that the height of this red-black tree is denoted as k, the height of the corresponding tree can be substituted in this expression. This can be represented by the expression $2^k - 1$.

The characterization of a red-black tree of height k is otherwise explained as having "black height of bh(x) or bh(x) - 1. " Since the root of the red-black tree must be black, according the the RB-TREE properties, we know that the height of the corresponding 2-4 tree must be the same as a red-black tree, since the black node of its root characterizes its height.

4. Red Nodes in Red-Black Tree (5 pts)

Consider a red-black tree formed by insert n nodes with RB-Insert. Prove that if $n > 1$, the tree has at least one red note. *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.)

This problem will be solved given a variety of cases.

Case 1: Empty Tree Insert where $n > 1$
If $n > 1$, and the tree is initially empty, when a node is created, it is the root. By properties of this tree structure, any root in an RB tree starts off as black, and any modification to this structure, such as INSERTION will allow for one red node to be created in the left subtree of root. By the Depth Property, this tree is still valid, meaning there is one red node in this tree.

Case 2: Red/Black Nodes in Tree where $n > 1$
If a given tree has an assortment of red and black nodes in the tree, it is given that it already has a red node. However, in the case a new node is added, restructuring of the tree may need to occur to ensure it complies with the properties of a RB-TREE. From restructuring, due to some of the properties of RB trees, both recoloring and restructuring may need to happen. This ensures that there is at least one red node in this given tree.

Case 3: Black Nodes in a Tree where $n > 1$
To ensure the integrity of an RB-Tree, we know that it is not possible to have an entire tree constructed of simply black nodes. Therefore, there can be no case where there are no red nodes.

Case 4: Black Nodes in a Tree Insertion where $n > 1$
If a tree has been constructed given it has an n number of black nodes, then calling RB-INSERT will automatically insert a new RED node into the tree. If the tree itself does not maintain the properties of the RB-Tree(color property, root property, external property, internal property, and depth property), then it must be rearranged, restructured, and recolored to ensure it maintains this structure. Once the coloring and structure is preserved, the new tree is created, which has either an assortment of red and black nodes, or a structure that satisfies the give given RB-TREE properties. Calling the RB-INSERT again will again cycle through the restructuring process, so there will be at least one red node in the tree. If the tree itself only has black nodes, as the insertion of nodes continues, if a second node is inserted, its child must be red, so this proves that there must be at least one red node in the tree.

5. O(n) Sort Of Variable Length Integers (10 pts)

Suppose we have a way of representing positive integers with variable numbers of digits, for example “3”, “61” and “317” may be included. Assume that there are no leading 0s, for example, “317” not “00317”. You are given an array of positive integers under this representation where the total number of digits over all the integers in the array is n. Show how to sort the array in O(n) time. (4 points for strategy and 6 points for analysis.)

Note that it is possible for one integer to have O(n) digits and all the others to be small. Therefore unmodified radix sort won't work: d = O(n) passes are required (and it is not viable to pad the integers with leading 0's to make them all the same length).

A potential algorithm that will sort a variable number of digits requires the algorithms presented in both bucket and radix sort. A combination of these instructions should present an runtime of O(n).
Given the first element of these "variable" number of digits must have an integer that is potentially small, by categorizing these digits into a variety of "buckets", we can otherwise sort these digits based on their numerical values and where they should be present in a newly sorted list.
The second part of implementing this algorithm includes using radix sort to obtain the value of these integers from the buckets that sorted them . Since radix sort works by taking the "least significant" digit, arranging these digits from the "bucket" that has the least significant data in a newly sorted array will be an optimal solution.
Analyzing the run-time of this, we know that bucket sort runs as $O(n)$, and Radix Sort runs as $O(d(n+k))$. An analysis of the final run-time mathematically is as follows:

$$O(d(n+n)) = O(d(2n))$$

By dropping constants, the final run-time is implemented as:

$$O(n)$$