The Australian National University Semester 2, 2024
School of Computing Tutorial 8

Convenor(s): Yun Kuen Cheung (Marco) & Ahad N. Zehmakan

Concrete version: Saturday 21st September, 2024, 20:28 (Compilation date and time)

Algorithms (COMP3600/6466)

Problems marked with ($*$) are challenge exercises. They will not be discussed in tutorials, and solutions will not be released. Once confident in your solution for a challenge exercise, you are welcome to discuss it with your tutor during the consultation period in the tutorial session, or schedule a time with Marco to present it.

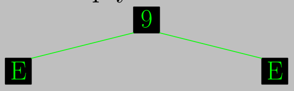
Exercise 1 *Red-Black Trees*

1. Start from an empty red-black tree, perform the following operations consecutively, and draw the updated red-black tree after each operation.
   (a) insert a new key 9;
   (b) insert a new key 25;
   (c) insert a new key 36;
   (d) insert a new key 49;
   (e) insert a new key 16;
   (f) insert a new key 12;
   (g) delete the node with key 12;
   (h) delete the node with key 49;
   (i) delete the node with key 36.

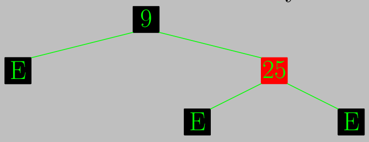
   Observe that the tree after (e) and the tree after (g) are different, although the operations in between them are insertion and deletion of the same key.

Solution

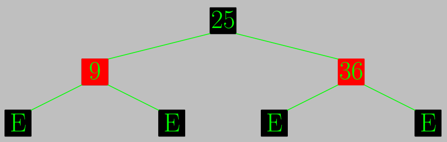
(a) Originally the red-black tree is empty. After inserting the key 9, it is in the root.



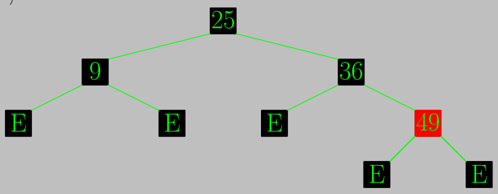
(b) This falls into Case $\mathcal{I}1$, which can be done easily.



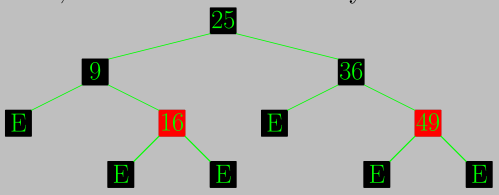
(c) This falls into Case $\mathcal{I}2$a. The resultant tree is



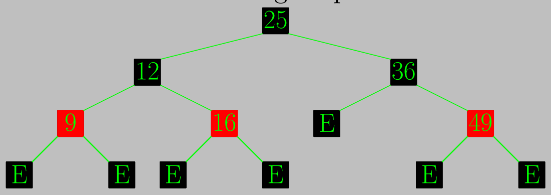
(d) This falls into Case $\mathcal{I}2$b. After following Step 1 of this case, the node 25 is recolored red, but since it is the root, it is then recolored black. The resultant tree is



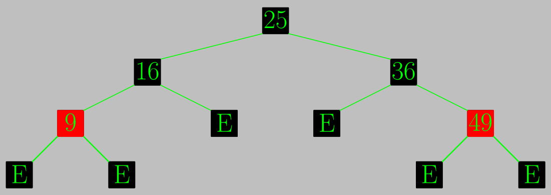
(e) This falls into Case $\mathcal{I}1$, which can be done easily.



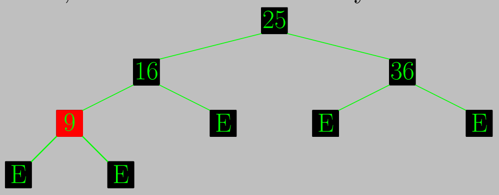
(f) This falls into Case $\mathcal{I}3$a. After following Steps 1 and 2 of this case, the resultant tree is



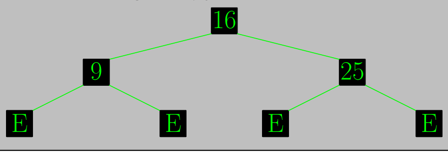
(g) This falls into Case $\mathcal{D}5$a. The resultant tree is



(h) This falls into Case $\mathcal{D}1$, which can be done easily.



(i) This falls into Case $\mathcal{D}2$ of red-black tree deletion. The node with key 36 is deleted, and is replaced by a double-black empty node. We need to repair it. Note that the sibling of the double-black empty node is node 16 which is black, and node 16's left child is node 9 which is red. So the situation falls into Case $\mathcal{R}1$ of red-black tree repairing. The resultant tree is



2. If a red-black tree has no red node, how does the tree look like?

Solution

Due to the same-black-height property of the red-black tree, all empty nodes (which are leaves) are on the same level. Since each internal node has exactly two children, so the tree must be a perfect binary tree.

3. Red-black trees are said to be less restrictive than height-balanced BST. We will see why in this question.

   (a) In a height-balanced BST, the height of the right subtree of the root is 100. What are the possible values of the height of the left subtree of the root?

   (b) In a red-black tree, the height of the right subtree of the root is 100. What are the minimum and maximum possible heights of the left subtree of the root?

Solution

(a) 99, 100 or 101

(b) Since the height of the right subtree of the root is 100, the longest path from the root to a leaf in the right subtree is 101. Along this path, there are 102 nodes, with the first node (the root) and the last node (a leaf) both colored black. Due to the no-red-parent-child condition, in the remaining 100 nodes, at most 50 of them are red. Thus, in this path, there are at least 52 black nodes, and hence the black height of the whole red-black tree is at least 52. Due to the same-black-height condition, any path from the root to a leaf in the left subtree of the root must have at least 52 nodes, and hence the height of the left subtree is at least **50**.

On the other hand, if the longest path from the root to a leaf in the right subtree contains only black nodes, then the black height of the red-black tree is 102. Due to the same-black-height condition, in a path from the root to a leaf in the left subtree of the root, there are 102 black nodes, and two of them are the root and the leaf. It is easy to see that there can be as many as 101 red nodes in this path. Thus, the total number of nodes in the path is 203, and hence the height of the left subtree can be at most **201**.

4. Suppose a red-black tree has n nodes, including the empty nodes.
   (a) Show that there are at most $\frac{n-1}{3}$ red nodes.
   (b) Consider a perfect binary tree of height h, where h is an even integer. The root is in level 0, its two children are in level 1, and so on.
      i. How many nodes are there in the tree?
      ii. Suppose the nodes in odd-number levels are all red, and the nodes in even-number levels are all black. How many red nodes are there in the tree?
      iii. Use (b)(i) and (b)(ii) to explain why the upper bound in (a) is tight.

Solution

(a) Due to the properties of red-black tree, each red node must have two children, and both children are black. Group each red node with its two black children. Clearly, no two groups share a common node. Also, the root does not belong to any of such groups since it is black and it has no parent. Thus, the number of red nodes is at most $\frac{n-1}{3}$.

(b) i. $1 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$.

ii. $2^1 + 2^3 + 2^5 + \ldots + 2^{h-1} = 2 \times (1 + 2^2 + 2^4 + \ldots + 2^{h-2}) = 2 \times \frac{2^h - 1}{2^2 - 1} = \frac{2^{h+1} - 2}{3}$.

iii. The coloring in (b)(ii) implies that a perfect binary tree of even height h can be converted to a red-black tree. The number of red nodes is $\frac{2^{h+1}-2}{3} = \frac{(2^{h+1}-1)-1}{3}$, which matches exactly with the upper bound in (a).

5. Present an algorithm whose input is a BST in which each node is colored red or black, and the algorithm determines whether the BST is a valid red-black tree. Your algorithm should run in $\mathcal{O}(n)$ time. (Hint: There are six conditions of red-black tree. You may design one algorithm for each condition, and then combine them.)

Solution

Among the six conditions, some of them are easy to check. Checking the root is black is easy and takes $\mathcal{O}(1)$ time. Checking conditions 2–4 and 6 (see lecture slides for what these conditions are) can be done by traversing each node and check the conditions directly. By employing standard BST traversal algorithm like in-order traversal, these can be done in $\mathcal{O}(n)$ time.

To check the same-black-height condition, we can employ a divide-and-conquer style algorithm. The idea is given a node, compute the black heights of its left and right subtrees, and check if these two black heights are equal. The black height of a leaf (which is empty node) is 1. In the Python implementation below, we use -1 to indicate that some node's two subtrees have different black heights.

```
def check_same_black_height(node):
    if node is None:
        return 1
    bh_left = check_same_black_height(node.left)
    bh_right = check_same_black_height(node.right)
    if bh_left == −1 or bh_right == −1:
        return -1
    if bh_left != bh_right:
        return -1
    if node.color == "black":
        return bh_left + 1
    else:
        return bh_left
```

By following the structure of the induction proof about BST traversal running time, it is not hard to see that the above algorithm for checking same-black-height condition takes $\mathcal{O}(n)$ time.

6. (*) Given a height-balanced BST, is it always possible to color the nodes in BST, and add empty nodes as leaves appropriately, so that the tree becomes a valid red-black tree? If you think so, give a rigorous proof to explain why. Otherwise, give a counter-example.

7. (*) Question 4 above discusses the tight bound on the number of red nodes in a perfect red-black tree. In this question, we will see how to compute the tight bound on the number of red nodes in a red-black tree with exactly n nodes.

   First, we define an *almost-red-black tree* to be a tree with node coloring that satisfies all properties of red-black tree, except that we allow the root to be either red or black (while for red-black tree the root must be black).

   Let $P(n,b)$ denote the maximum possible number of red nodes in an almost red-black tree which has n nodes, black height b and a red root. If there does not exist such an almost red-black tree, we set $P(n,b) = -\infty$. Analogously, let $Q(n,b)$ denote the maximum possible number of red nodes in an almost red-black tree which has n nodes, black height b and a black root. If there does not exist such an almost red-black tree, we set $Q(n,b) = -\infty$. Let $R(n,b) = \max\{P(n,b), Q(n,b)\}$

   (a) Compute the values of $P(n,b)$ and $Q(n,b)$ when $n = 1,2$ and $b \geq 0$.
   (b) Compute the values of $P(n,0)$ and $Q(n,0)$ when $n \geq 3$.
   (c) Explain why $P(n,b) = Q(n,b) = -\infty$ when $b > 2\log(n+1)+1$.
   (d) For $n \geq 3$ and $b \geq 1$, explain why

   $$P(n,b) = 1 + \max_{1\leq m\leq \lfloor\frac{n-1}{2}\rfloor} \{Q(m,b) + Q(n-1-m,b)\} \ ,$$

   and

   $$Q(n,b) = \max_{1\leq m\leq \lfloor\frac{n-1}{2}\rfloor} \{R(m,b-1) + R(n-1-m,b-1)\} \ .$$

   (e) Write a program in your favorite programming language, to compute the maximum possible number of red nodes in a red-black tree with 2025 nodes.

Exercise 2 *Heaps*

8. Suppose a binary heap is implemented using an array, as what we described during lecture. The keys in the array are:

   $$5, 7, 20, 11, 8, 24, 35, 21, 30, 10, 12, 26, 100, 66 \ .$$

   (a) What is the key of the parent of the node with key 30?
   (b) What are the keys of the children of the node with key 20?
   (c) Perform the following operations with the above binary heap consecutively, and write down the updated array after each operation.
      i. insert a new key 19;
      ii. extract-min;
      iii. extract-min;
      iv. insert a new key 4.

Solution

(a) Key 30's index in the array is 8, so its parent has index $\lfloor \frac{8-1}{2} \rfloor = 3$, so the answer is 11.

(b) Key 20's index in the array is 2, so its children have indices $2 \times 2 + 1 = 5$ and $2 \times 2 + 2 = 6$. So the answers are 24 and 35.

(c)
   i. 5, 7, 19, 11, 8, 24, 20, 21, 30, 10, 12, 26, 100, 66, 35
   ii. 7, 8, 19, 11, 10, 24, 20, 21, 30, 35, 12, 26, 100, 66
   iii. 8, 10, 19, 11, 12, 24, 20, 21, 30, 35, 66, 26, 100
   iv. 4, 10, 8, 11, 12, 24, 19, 21, 30, 35, 66, 26, 100, 20

9. Design a data structure that supports find-min and find-max operations in $\mathcal{O}(1)$ time, and it supports insertion, extract-min and extract-max operations in $\mathcal{O}(\log n)$ time. (Hint: In the lecture, we discuss min-heap, but a max-heap can be analogously defined. If we naively maintain one min-heap and one max-heap simultaneously, find-min and find-max operations can be done in $\mathcal{O}(1)$, while insertion can be done in $\mathcal{O}(\log n)$ time. The main challenge is: when extract-min is called, what should you do to the max-heap?)

Solution

The hint already discusses how find-min, find-max and insertion can be done in the desired runtime. Now we focus on how to perform extract-min (extract-max will be symmetric, so we won't discuss in detail).

The main reason that if we were to naively maintain one min-heap and one max-heap simultaneously is when extract-min is called, we would need long time (potentially $\Omega(n)$ time) to find out where the smallest key locates in the max-heap. To avoid this issue, when we implement the min-heap and max-heap as two arrays simultaneously, we do not store the keys in the arrays separately, but instead, the two cells in the two arrays with the same key are now pointing to a single object with three member variables: the key, its index in the min-heap, and its index in the max-heap. When extract-min is called, we use the min-heap to immediately identify the object with the smallest key, and then we can find out its location in the max-heap using its member variable instantly.

To perform extract-min in the min-heap, we follow the algorithm we discuss in the lecture. Denote the object with the smallest key by v. To delete v, we first find out its location in the max-heap as described in the last paragraph. Then we move the last object in the max-heap array (denote this object by u) to the location. This is like inserting u at the location of v. Then we call increase-key (the analogous operation of decrease-key in min-heap) on u. Since it is like inserting u, the runtime is the same as insertion, which is $\mathcal{O}(\log n)$.