

## CSE3358 Problem Set 8 Solution

### Problem 1: Practice Insertions (10 points)

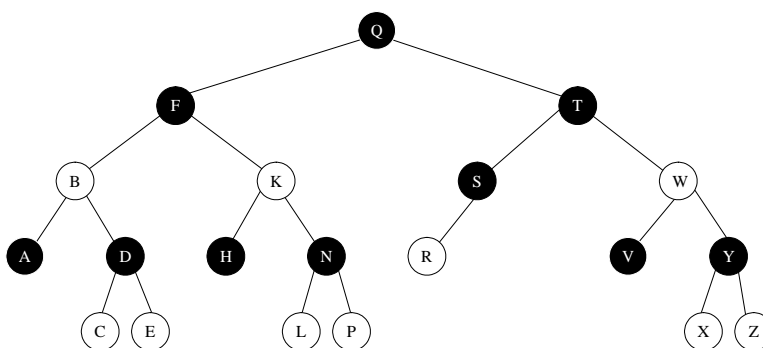
Show the results of inserting the following keys

F, S, Q, K, C, L, H, T, V, W, R, N, P, A, B, X, Y, D, Z, E

in the order shown into an originally empty

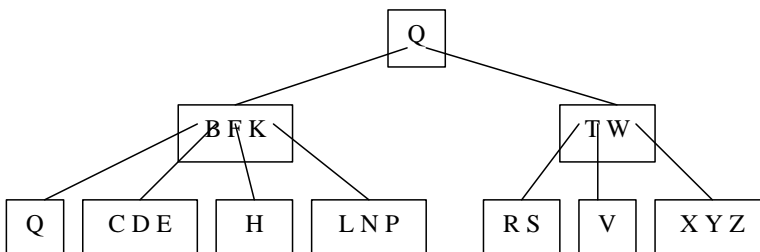
(a) (5 points) Red-Black Tree

ANSWER:



(b) (5 points) B-tree with  $t = 2$

ANSWER:



### Problem 2: AVL tree (30 points)

In 1962, Adelson-Velskii and Landis invented the AVL tree. An AVL tree is a binary search tree that satisfies the following additional property:

AVL property: For any node  $x$ ,  $|h[left[x]] - h[right[x]]| \leq 1$  where  $h[x]$  denotes the height of  $x$ , and we define  $h[NIL] = -1$ .

In other words, the heights of the left and right subtrees (children) of a node differ by at most one.

(a) (5 points) Let  $S(h)$  be the minimum number of nodes in an AVL tree of height  $h$ . Show that  $S(0) = 1$  and  $S(1) = 2$  and that for  $h \geq 2$ ,  $S(h) = S(h - 1) + S(h - 2) + 1$ .

**ANSWER:** We require one node (a root) to make a tree of height 0; therefore,  $S(0) = 1$ . We require at least 2 nodes to make a tree of height 1; therefore,  $S(1) = 2$ . Now consider an AVL tree with height  $h \geq 2$ . This tree must have

- a root
- a left AVL subtree  $L$
- a right AVL subtree  $R$

At least one of  $L$  and  $R$  must have height  $h - 1$  for our tree to have height  $h$ . Without loss of generality, say  $L$  has height  $h - 1$ . Since  $L$  is an AVL tree, it has at least  $S(h - 1)$  nodes. The height of  $R$  could be either  $h - 1$  or  $h - 2$  but not less, since our tree is an AVL tree. Therefore,  $R$  must have at least  $S(h - 2)$  nodes. As a result, our AVL tree with height  $h$  must have at least  $S(h - 1) + S(h - 2) + 1$  nodes and hence  $S(h) = S(h - 1) + S(h - 2) + 1$  for  $h \geq 2$ .

(b) (5 points) Using induction, show that  $S(h) \geq \phi^h$ , where  $\phi = \frac{1+\sqrt{5}}{2}$ .

**ANSWER:**

Base case:  $S(0) = 1 \geq \phi^0 = 1$  and  $S(1) = 2 \geq \phi^1 = 1.618$ .

Inductive step: Assume this hold for all heights  $\leq h$ . Let's show it holds for height  $h + 1$ .  $S(h + 1) = S(h) + S(h - 1) + 1 < S(h) + S(h - 1) \leq \phi^h + \phi^{h-1} = \phi^{h+1}(\frac{1}{\phi} + \frac{1}{\phi^2}) = \phi^{h+1}.1 = \phi^{h+1}$ .

(c) (5 points) Conclude that the height  $h$  of an AVL tree on  $n$  nodes satisfies  $h = \Theta(\log n)$ .

**ANSWER:**  $n \geq S(h) \geq \phi^h$ . Therefore,  $h \leq \log_{\phi} n$ , i.e.  $h = O(\log n)$ . Since  $h = \Omega(\log n)$  for any binary tree, we conclude that  $h = \Theta(\log n)$ .

(d) (15 points) Given that every node  $x$  stores its height  $h[x]$ , an AVL tree can be maintained after an insertion operation. Here's how: After inserting a new node  $x$  as a leaf, we set  $h[x] = 0$  and we go up the tree updating the height of every node on the path from  $x$  to the root by incrementing its height by 1. When we encounter the first node  $y$  for which  $|h[\text{left}[y]] - h[\text{right}[y]]| = 2$  (if any), we fix the AVL property for  $y$  by performing rotations. Depending on the position of  $x$  relative to  $y$ , different rotation(s) must be performed. We have two cases to consider (the other cases are symmetric)

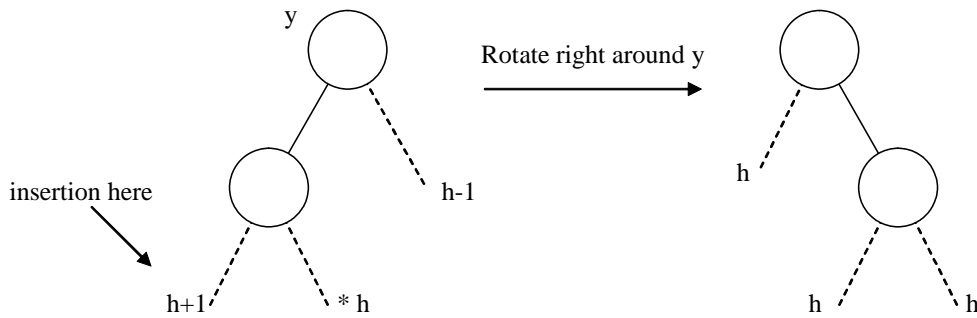
- case 1:  $x$  is in the left subtree of  $\text{left}[y]$
- case 2:  $x$  is in the right subtree of  $\text{left}[y]$

The above two cases suggest that, since node  $y$  violates the AVL property after insertion, the left subtree of  $y$  is higher than the right subtree of  $y$  (the insertion was done in the left subtree of  $y$ ), and the height difference is 2 (more than 1). For each of the two above cases, describe the rotation(s) required to re-establish the AVL property for node  $y$ . Argue that after re-establishing the AVL property for node  $y$ , no more height updates or fixes will be needed, i.e. the resulting tree is AVL.

**ANSWER:** Before insertion, the left subtree of  $y$  had height  $h$  and the right subtree of  $y$  had height  $h - 1$ . After insertion, the left subtree of  $y$  has height  $h + 1$  resulting in a difference of 2 in the heights for both subtrees.

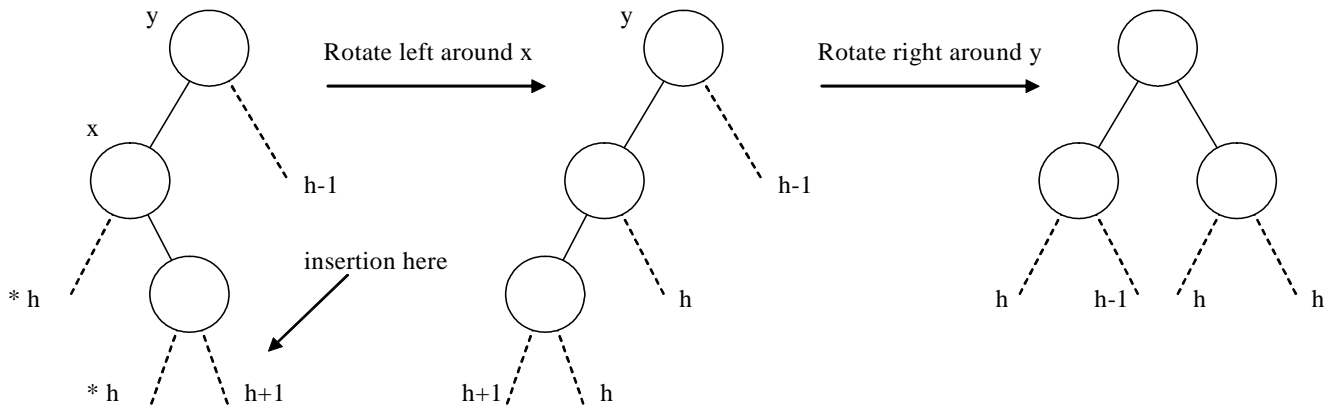
In the following figures, the numbers represent the maximum depth of a node from node  $y$ .

Case 1



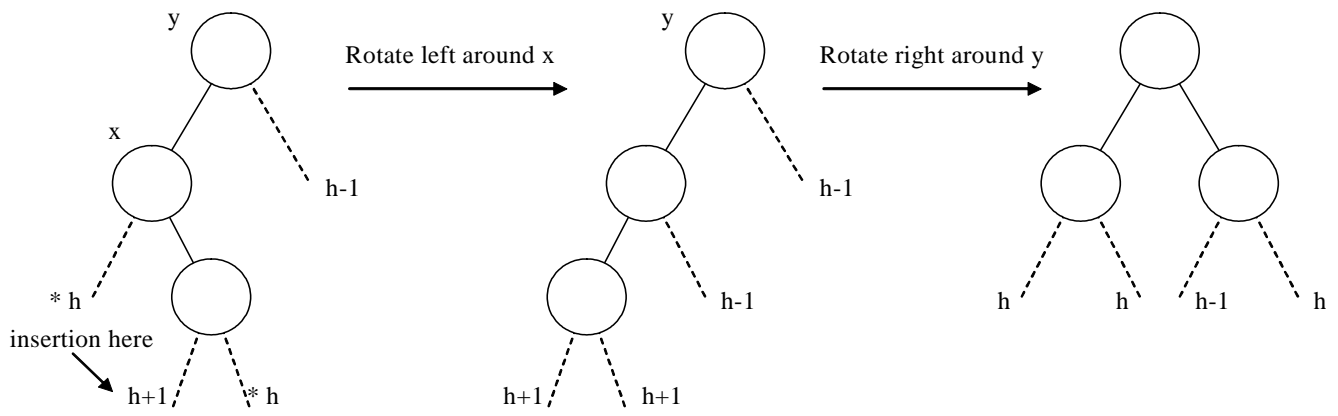
\* this must be  $h$  because  $y$  is the first node to violate the property and the tree was AVL before the insertion

Case 2



\* these must be  $h$  because  $y$  is the first node to violate the property and the tree was AVL before the insertion

OR



\* these must be  $h$  because  $y$  is the first node to violate the property and the tree was AVL before the insertion

Since in all cases, the maximum depth of a node from node  $y$  goes back to  $h$ , the height of  $y$  does not change and no further height updates or rotations are needed.

**Problem 3: The dishonest successor (30 points)**

As you know, the homework policy states that dishonest students are given red tags. Therefore, every student has a grade and a red-tag boolean flag to identifying whether the student is dishonest or not.

We wish to support all dynamic set operations on students such as INSERT, DELETE, SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR in addition to the following operation:

Given a student  $x$  with grade  $k$ , find the dishonest student  $y$  (if any) with the lowest grade  $k'$  such that  $k' > k$  in  $O(\log n)$  time, where  $n$  is the number of students. In other words, given a student  $x$ , we wish to find the dishonest successor of  $x$  in  $O(\log n)$  time.

Explain how you can augment a red-black tree to support this operation. In doing so, describe the 4 steps discussed in class:

(a) (0 points) Choose the underlying data structure (red-black tree in this case) and determine what constitute the keys.

**ANSWER:** They keys will be the grades.

(b) (10 points) Identify the additional information that will augment the red-black tree

**ANSWER:** In addition to the key  $key[x]$ , the flag  $dishonest[x]$  and the color  $color[x]$ , every node  $x$  will have a field  $n[x]$  that represents the number of dishonest students in  $x$ 's subtree (i.e. including  $x$  itself if  $dishonest[x] = 1$ ).

(c) (10 points) Argue that the additional information can be maintained easily by insertion and deletion and describe how it is maintained by rotations

**ANSWER:** Since  $n[x]$  can be completely obtained from  $left[x]$  and  $right[x]$ , it is possible to update and maintain  $n[x]$  efficiently upon insertions, deletions, and during rotations.

$$n[x] = n[left[x]] + n[right[x]] + dishonest[x]$$

$$n[NIL] = 0$$

(d) (10 points) Provide a pseudocode for the operation DISHONEST-SUCCESSOR( $x$ )

DISHONEST-SUCCESSOR( $x$ )

```

if  $n[right[x]] \neq 0$ 
  then  $y \leftarrow right[x]$ 
    while  $n[left[y]] \neq 0$  or ( $dishonest[y] = 0$  and  $n[right[y]] \neq 0$ )
      do if  $n[left[y]] \neq 0$ 
        then  $y \leftarrow left[y]$ 
      else  $y \leftarrow right[y]$ 
    return  $y$ 
 $y \leftarrow$  first ancestor of  $x$  such that  $x$  in left subtree of  $y$ 
if  $y \neq NIL$ 
  then if  $dishonest[y]$ 
    then return  $y$ 
    else return DISHONEST-SUCCESSOR( $y$ )
  else return  $NIL$ 

```

The pseudocode above searches for a dishonest student with smallest grade in the right subtree of  $x$ . If one exists, this is returned as the dishonest successor. If one does not exist, then we examine the first ancestor  $y$  of  $x$  such that  $x$  lies in the left subtree of  $y$ . If  $y$  is dishonest,  $y$  is returned as the dishonest successor; otherwise, we recursively look for the dishonest successor of  $y$ .

The running time of the algorithm is  $O(\log n)$  since it goes down the tree at most once and up the tree at most once, and the height of an red-black tree is  $O(\log n)$ .

**Problem 4: Some B-tree operations (30 points)**

Given a B-tree  $T$  with minimum degree  $t$ , describe an algorithm that:

(a) (10 points) Finds  $(x, i)$  in  $O(\log_t n)$  time such that  $key_i[x]$  is the MINIMUM key in  $T$ .

**ANSWER:** The minimum key is the first key of the left-most leaf in the B-tree. Therefore, this can be obtained by returning  $key_1[x]$ , where  $x$  is obtained by repeatedly following the pointer  $c_1$  starting from the root of the tree. This takes  $O(h) = O(\log_t n)$  time.

(b) (10 points) Given  $(x, i)$ , finds  $(y, j)$  such that  $key_j[y]$  is the SUCCESSOR of  $key_i[x]$  in  $T$  in time:

$O(\log_t n)$  if  $x$  is a non-leaf

$O(\log_2 n)$  if  $x$  is a leaf

**ANSWER:** If  $x$  is not a leaf, the successor of  $key_i[x]$ , is the minimum key in the tree rooted at  $c_{i+1}[x]$ . Therefore, this can be obtained in  $O(\log_t n)$  time as described in (a).

If  $x$  is a leaf, then the successor of  $key_i[x]$  is  $key_{i+1}[x]$ , and hence can be obtained in  $O(1)$  time. If  $i = 2t - 1$  (hence  $key_{i+1}[x]$  does not exist), then let  $(y, j)$  be such that  $c_j[y] = x$  ( $y$  is the parent of  $x$ ). The successor of  $key_i[x]$  can be obtained by recursively treating  $y$  as a leaf and finding the successor of  $key_{j-1}[y]$ . This takes  $O(h) = O(\log_t n)$  time if  $(y, j)$  is known at every node. However,  $(y, j)$  is not known and needs to be constructed by first making a downward pass to  $x$  and pushing  $(y, j)$  onto a stack on the way down. Then the  $(y, j)$  information can be obtained by repeatedly popping the stack. The downward path takes  $O(\log_2 n)$  time as described in part (c) below, yielding a total of  $O(\log_2 n)$  time.

(c) (10 points) Given  $k$ , finds  $(x, i)$  in  $O(\log_2 n)$  time such that  $key_i[x] = k$  in  $T$ .

**ANSWER:** The search algorithm described in class (and found in the book) makes a downward pass to node  $x$  by spending  $O(t)$  time at every node  $y$  on the path to locate the pointer to the appropriate branch, by finding two keys  $k_1$  and  $k_2$  in  $y$  such that  $k_1 \leq k \leq k_2$ . Therefore, the total running time of the search is  $O(th) = O(t \log_t n) = O(\frac{t}{\log_2 t} \log_2 n)$ . The  $O(t)$  time can be reduce to  $O(\log t)$  by performing binary search on the node  $y$  to find the interval  $[k_1, k_2]$ . This modification will result is an  $O((\log_2 t)h) = O(\log_2 t \log_t n) = O(\log_2 n)$  time.

**Problem 5: Overlapping rectangles (20 points)**

VLSI databases commonly represent an integrated circuit as a list of rectangles. Assume that each rectangle is rectilinearly oriented (sides parallel to the  $x$ - and  $y$ -axis), so that a representation of a rectangle consists of its minimum and maximum  $x$ - and  $y$ -coordinates. Give an  $O(n \log n)$  time algorithm to decide whether or not a set of rectangles so represented contains two rectangles that overlap. Your algorithm need not report all intersecting pairs, but it must report that an overlap

exists if one rectangle entirely covers another, even if the boundary lines do not intersect. (*Hint*: Move a “sweep” line across the set of rectangles and maintain their heights in an interval tree).

**ANSWER:** Move a sweep line from left to right, while maintaining the set of rectangles currently intersected by the line in an interval tree. The interval tree will organize all rectangles whose  $x$  interval includes the current position of the sweep line, and will be based on the  $y$  intervals of the rectangles, so that any overlapping  $y$  intervals in the interval tree correspond to overlapping rectangles.

To emulate a sweep line, sort the rectangles by their  $x$ -coordinates. Each rectangle must appear twice in the sorted list, once for its left  $x$ -coordinate and once for its right  $x$ -coordinate.

Now scan the sorted list (from lowest to highest in  $x$ -coordinate).

Entering a rectangle:

When an  $x$ -coordinate of a left edge is found, check whether the rectangle’s  $y$ -coordinate interval overlaps an interval in the tree, and insert the rectangle (keyed on its  $y$ -coordinate interval) into the tree.

Exiting a rectangle:

When a  $x$ -coordinate of a right edge is found, delete the rectangle from the interval tree.

The interval tree always contains the set of “open” rectangles intersected by the sweep line. If an overlap is ever found in the interval tree, there are overlapping rectangles.

The total time is  $O(n \log n)$  to sort the rectangles (e.g. mergesort, heapsort) and  $O(n \log n)$  for interval-tree operations (insert, delete, check for overlap).