

- The objective of this data structure is to maintain a balanced binary search tree (BBST). When there are  $n$  keys stored in the AVL<sup>1</sup> tree, its height is guaranteed to be  $O(\lg n)$ . This helps in accomplishing search, insert, delete, split, and join operations in  $O(\lg n)$  time in the worst-case.
- The AVL tree is a BST on the keys stored in it. The balance condition states that for every node  $v$  in the tree,  $-1 \leq (h(v.right) - h(v.left)) \leq 1$ . At every node  $v$ , the AVL tree stores the balance information  $h(v.right) - h(v.left)$  in a two-bit field  $v.b$ . It is immediate that  $\forall_v v.b \in \{-1, 0, 1\}$  whenever the given BST is an AVL tree.

Observation: Given  $v.b$  for every node  $v$  of a tree  $T$ , the height of  $T$  can be determined in  $O(\text{height}(T))$  time.

- For an AVL tree  $T$  of height  $h$ , let  $n(h)$  be the minimum number of nodes that  $T$  can have.

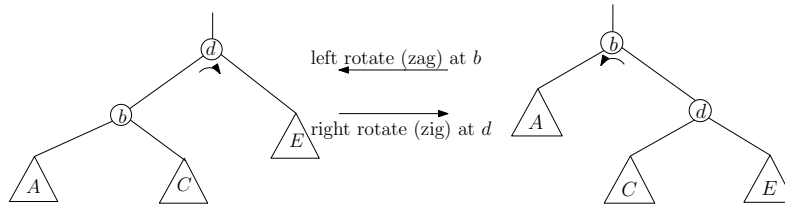
It is obvious,  $n(0) = 1, n(1) = 2$ . Since the  $T$  has height  $h$ , either left or right subtree must have height  $h - 1$ . To minimize the number of nodes in  $T$ , without violating the balance condition, the height of the other subtree can be no smaller than  $h - 2$ . Hence,

$$\begin{aligned}
 n(h) &= n(h-1) + n(h-2) + 1 \text{ for } h \geq 2 \\
 \Rightarrow n(h) &> n(h-1) + n(h-2) \\
 \Rightarrow n(h) &> 2n(h-2) > 2^2 n(h-4) > \dots > 2^i n(h-2i) \\
 \Rightarrow n(h) &> 2^i n(h-2i).
 \end{aligned}$$

Since  $h - 2i \geq 2, i \leq \frac{h}{2} - 1$ . Substituting  $i = \frac{h}{2} - 1$  in the above,  $h$  is  $O(\lg n)$ .

The insert, delete, split, and join algorithms ensure at no node balance condition is violated. Since no additional explicit algorithm is needed to balance the tree, AVL tree is a *self-balancing* binary search tree.

- Let  $T'$  be the tree resultant of right rotating at a node  $d$  of a BST  $T$ . Refer the below figure. Since the inorder traversals of  $T$  and  $T'$  yield the same ordering of keys, and since the relation of  $b, d$ , keys in subtrees  $A, C$ , and  $E$  are correct with respect to keys in  $T - T_{d.parent}$ ,  $T'$  is a BST. The same holds good after any left rotate as well.



Further, a left or a right rotation takes  $O(1)$  time in the worst-case.

Observation: With a right rotate at  $d$ , the  $\text{depth}(d)$  increases by one and  $\text{depth}(d.left)$  decreases by one. With a left rotate at  $b$ , the  $\text{depth}(b)$  increases by one and  $\text{depth}(b.right)$  decreases by one. However, the heights of  $d$  and  $b$  post a rotation depend on the heights of  $A, C$ , and  $E$ .

<sup>1</sup>named after its inventors Adelson-Velskii and Landis

- Algorithm for inserting node  $x$  into AVL tree  $T$ :

First, node  $x$  is inserted into BST using the insertion algorithm for BSTs. Let  $P$  be the simple path from  $x$  to the root. While walking along  $P$  from  $x$ , at every node  $v$ , this algorithm updates  $v.b$ . Further, for the first node  $d$  along  $P$  at which the AVL balance condition is violated, if any, and its predecessor  $b$  along  $P$ ,

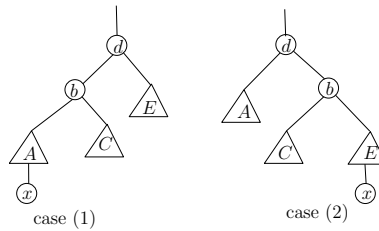
if  $x$  is in a subtree which does not lie between  $d$  and  $b$

then //cases in which a single rotation suffices

if  $b$  is the left child of  $d$  //  $x$  is inserted into  $T_{b, left}$

then right-rotate( $T, d$ ) ————— (1)

else left-rotate( $T, d$ ) ————— (2)

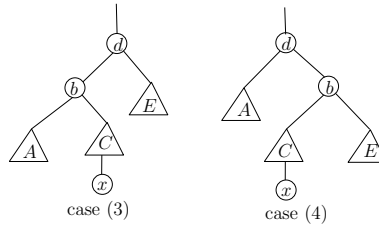


else //cases in which two rotations are needed

if  $b$  is the left child of  $d$

then left-rotate( $T, b$ ) + rightRotate( $T, d$ ) ————— (3)

else right-rotate( $T, b$ ) + leftRotate( $T, d$ ) ————— (4)

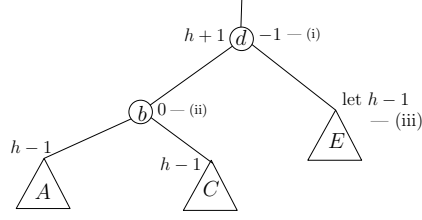


Observation: While walking along  $P$  from  $x$ , for any node  $v$  that occurs on this path, modified  $v.b$  can be determined in constant time in the worst-case.

Below, we present the correctness of (1) and (3). The analysis of cases (2) and (4) is symmetric.

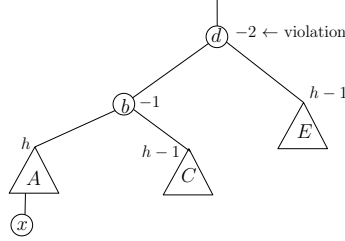
- — correctness of (1) —

\* just before inserting  $x$  into  $A$  (while the insertion of  $x$  causing a balance violation at  $d$ ):



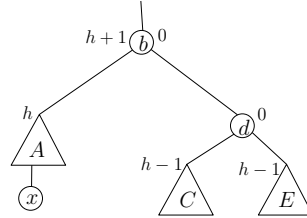
- since the insertion of  $x$  in  $T_b$  causing a violation at  $d$ ,  $d.b$  cannot be equal to 0 or  $+1$
- since the first violation occurs at  $d$  after inserting  $x$  into  $A$ ,  $b.b \neq -1$ ; and, if  $b.b = 1$ , violation does not occur at  $d$  after insertion
- let  $h - 1$  be the height of  $E$ ; then, since  $d.b = -1$ ,  $h(b) = h$ ; since  $b.b = 0$ ,  $h(A) = h(C) = h - 1$ ; further,  $h(d) = h + 1$

\* just after inserting  $x$  into  $A$ :



- since  $h(A \cup \{x\}) = h$ ,  $h(d) = h + 2$ ;  $b.b = -1$ ;  $d.b = -2$

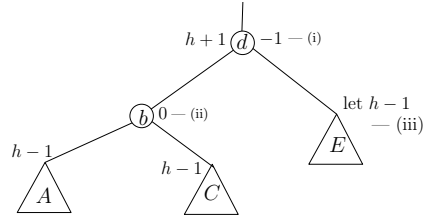
\* just after right rotate at  $d$ :



- $d.b = h(E) - h(C) = (h - 1) - (h - 1) = 0$ ;  $b.b = h(d) - h(A \cup \{x\}) = h - h = 0$ ;

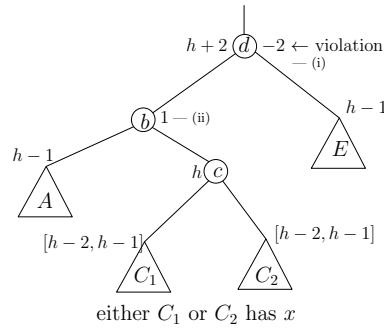
• — correctness of (3) —

\* just before inserting  $x$  into  $C$  (while the insertion of  $x$  causing a balance violation at  $d$ ):



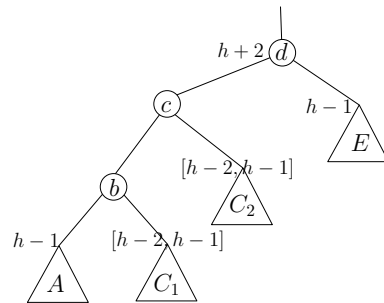
- again, since the insertion of  $x$  in  $T_b$  causing a violation at  $d$ ,  $d.b$  cannot be equal to 0 or  $+1$
- since the balance violation occurs first at  $d$  along  $x$  to  $d$  path after inserting  $x$  into  $C$ ,  $b.b \neq 1$ ; and, if  $b.b = -1$ , violation does not occur at  $d$  after insertion
- let  $h-1$  be the height of  $E$ ; then, since  $d.b = -1$ ,  $h(b) = h$ ; since  $b.b = 0$ ,  $h(A) = h(C) = h-1$ ; hence,  $h(d) = h+1$

\* just after inserting  $x$  into  $C$ :



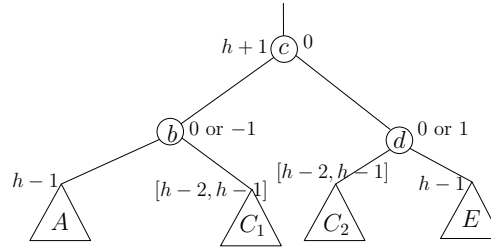
- since  $h(C \cup \{x\}) = h$ ,  $b.b = 1$ ; since  $h(b) = h+1$  and since  $h(E) = h-1$ ,  $d.b = -2$ ; further,  $h(d) = h+2$
  - since  $h(c) = h(C \cup \{x\}) = h$ ,  $h-2 \leq h(C_1) \leq h-1$ : if  $h(C_1) < h-2$ , the first violation along  $x$  to  $d$  path does not occur at  $d$ ; if  $h(C_1) > h-1$ ,  $h(c) > h$
- analogously, since  $h(C \cup \{x\}) = h$ ,  $h-2 \leq h(C_2) \leq h-1$

\* just after left rotate at  $b$ :



- height of  $T_d$  continues to be  $h + 2$ ; balance violation at  $d$  is not yet got fixed

\* just after right rotate at  $d$ :



- since  $\max(h(C_1), h(C_2)) = h - 1$ ,  $h(A) = h - 1$ , and  $h(E) = h - 1$ ,  $h(c) = h + 1$ ;

- since  $h(b) = h$  and  $h(d) = h$ ,  $c.b = 0$ ;

- since  $h - 2 \leq h(C_1) \leq h - 1$ ,  $b.b = -1$  or  $0$

and, since  $h - 2 \leq h(C_2) \leq h - 1$ ,  $d.b = +1$  or  $0$

- Fixing imbalance at  $d$  ensures  $v.b \in \{-1, 0, 1\}$  at every node  $v$  of  $T$ :  $h(b)$  after insertion is made equal to  $h(d)$  before insertion in single rotate case;  $h(c)$  after insertion is made equal to  $h(d)$  before insertion in double rotate case.

The insertion involves traversing the simple path from  $x$  to  $d$  and one/two rotations. Since the height of an AVL tree is  $O(\lg n)$  and since it takes  $O(1)$  time for any one rotate, the time complexity of inserting a node into a AVL tree is  $O(\lg n)$ . Further, since every node obeys the balance condition in the updated tree  $T'$ , the height of  $T'$  is  $O(\lg n + 1)$ , where  $n + 1$  is the number of nodes in  $T'$ .

- Algorithm to delete a node  $z$  from the AVL tree  $T$ :

After deleting node  $z$  from  $T$ , let  $P$  be the path from the position where  $x$  was located to the root. (The significance of  $x$  was detailed in the algorithm for deleting a node from the BST.) For every node  $d$  along  $P$  at which the AVL balance condition is violated, if any:

if  $d.b < -1$

if  $d.left.b \leq 0$

then right-rotate( $d$ ) ————— (5)

else left-rotate( $d.left$ ) + right-rotate( $d$ ) ————— (6)

if  $d.b > 1$

if  $d.right.b \geq 0$

then left-rotate( $d$ ) ————— (7)

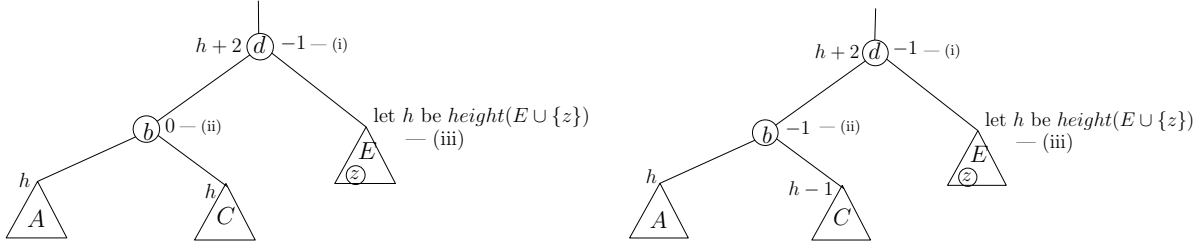
else right-rotate( $d.right$ ) + left-rotate( $d$ ) ————— (8)

Essentially, (5) and (6) handle node getting removed from  $T_{d.right}$ ; (7) and (8) handle node getting removed from  $T_{d.left}$ . Below, we present the correctness of (5) and (6). The analysis of cases (7) and (8) is symmetric.

Observation: Analogous to the case of insertion, while walking along  $P$  from  $x$ , for any node  $v$  that occurs on this path, modified  $v.b$  can be determined in constant time in the worst-case.

• — correctness of (5) —

\* just before deleting  $z$  from  $E \cup \{z\}$  (while that deletion causing a balance violation at  $d$ ):



- since deleting  $z$  from  $E \cup \{z\}$  causing a violation at  $d$ ,  $d.b \neq 1$  and  $d.b \neq 0$

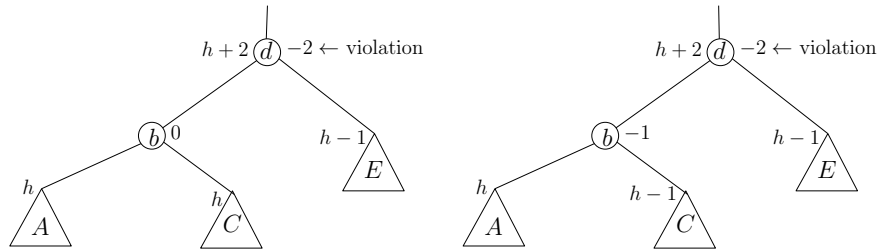
- (5) occurs only if  $b.b = -1$  or  $0$  ((6) handles the case of  $b.b = 1$ )

- let  $h(E \cup \{z\}) = h$ ; then, since  $d.b = -1$ ,  $h(b) = h + 1$

- if  $b.b = 0$ ,  $h(A) = h(C) = h$  — (I)

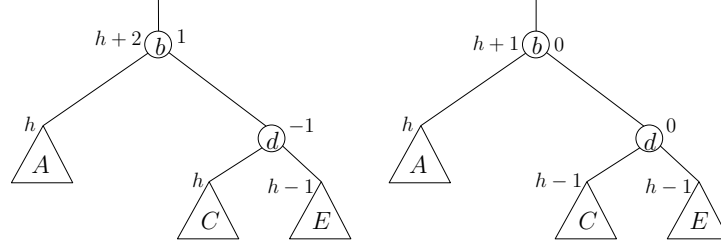
if  $b.b$  is  $-1$ ,  $h(C) = h - 1$  and  $h(A) = h$  — (II)

\* just after deleting  $z$  from  $E \cup \{z\}$ :



- since there is a violation at  $d$  upon deleting  $z$  from  $E \cup \{z\}$ ,  $d.b = -2$ ; since  $d.b$  is getting changed due to deletion,  $h(E) = h - 1$

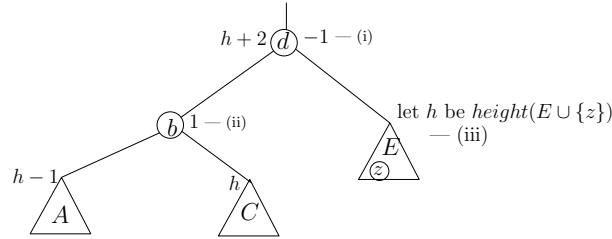
\* just after right rotate at  $d$ :



- in case (I),  $h(A) = h(C) = h$  and  $h(E) = h - 1$ ; hence,  $h(d) = h + 1$ ,  $h(b) = h + 2$ ,  $d.b = -1$ ,  $b.b = 1$
- in case (II),  $h(A) = h$ ,  $h(C) = h - 1$ , and  $h(E) = h - 1$ ; hence,  $h(d) = h$ ,  $h(b) = h + 1$ ,  $d.b = 0$ ,  $b.b = 0$

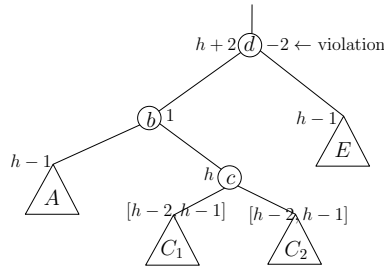
• — correctness of (6) —

\* just before deleting  $z$  from  $E \cup \{z\}$  (while that deletion causing a balance violation at  $d$ ):



- again, since deleting  $z$  from  $E \cup \{z\}$  causing a violation at  $d$ ,  $d.b \neq 1$  and  $d.b \neq 0$
- let,  $h(E \cup \{z\}) = h$ ; since  $d.b = -1$ ,  $h(b) = h + 1$ ; hence,  $h(d) = h + 2$
- $b.b = 1$  (since the case of  $b.b$  equal to either 0 or  $-1$  was handled in (5))
- since  $b.b = 1$ ,  $h(C) = h$  and  $h(A) = h - 1$

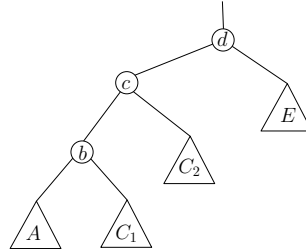
\* just after deleting  $z$  from  $E \cup \{z\}$ :



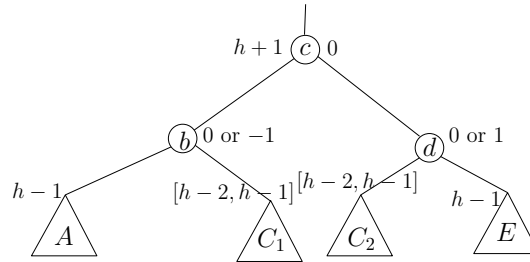
- since  $d.b$  was  $-1$ , the violation occurs at  $d$  only if  $d.b$  changes to  $-2$
- since there is a violation at  $d$ , the  $h(E)$  must be changing from  $h$ ; hence,  $h(E) = h - 1$

- since  $h(C) = h$  and since there is no violation at  $c$ , various possible values for  $(h(C_1), h(C_2))$  tuple are  $(h-1, h-1), (h-1, h-2), (h-2, h-1)$

\* after left rotate at  $b$ :



\* after right rotate at  $d$ :



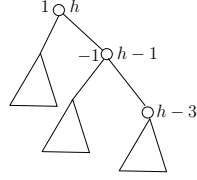
- $h(c) = \max(h(A), h(C_1), h(C_2), h(E)) + 2 = \max(h-1, \max(h(C_1), h(C_2)), h-1) + 2 = h+1$
- since  $h(E) = h-1$  and  $h-2 \leq h(C_2) \leq h-1$ ,  $d.b$  is 0 or +1; analogously,  $b.b$  is 0 or -1
- Since the  $h(d)$  before deletion is  $h+2$  and  $h(c)$  after two rotations is  $h+1$ , further ancestors need to be checked for the balancing condition.

Since the  $h(d)$  before deletion could differ from  $h(b)$  post rotation(s) in some of the cases of deletions, unlike in insert, further ancestors need to be checked for possible balance condition violation.

- The deletion algorithm takes  $O(\lg n)$  time: the work involves, in the worst-case, at every node  $v$  on the simple path from  $x$  to the root, at most a couple of rotations and updating the balance information at a constant number of nodes in the vicinity of  $v$ .
- The following observation is useful in joining AVL trees:

For any AVL tree  $T$  of height  $h$ , there always exists a node on the right spine of  $T$  whose height is either  $i$  or  $i+1$ , for any  $0 \leq i \leq h$ . However, there is no guarantee that the right spine of  $T$  to have a node of height  $i$  or a node of height  $i+1$ . (Refer to below figure.) Specifically, the last node on the right spine of  $T$  is guaranteed to have height either 0 or 1.

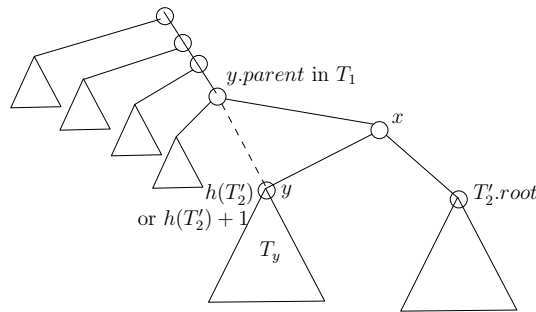




no node on the right spine has height  $h - 2$

\* Given two AVL trees  $T_1$  and  $T_2$  with every key in  $T_1$  less than or equal to every key in  $T_2$ , the following algorithm joins (merges)  $T_1$  and  $T_2$ :

- (a) using the balance information stored at nodes, determine  $height(T_1)$  and  $height(T_2) \leftarrow$  takes  $O(height(T_1) + height(T_2))$  time  
w.l.o.g., suppose  $height(T_1) \geq height(T_2)$ ; the other case is analogous
- (b) find a node  $x$  in  $T_2$  such that  $x.key$  is the minimum among keys stored in  $T_2 \leftarrow$  takes  $O(height(T_2))$  time
  - using the algorithm detailed above, delete  $x$  from AVL tree  $T_2 \leftarrow$  takes  $O(height(T_2))$  time
  - let  $T'_2$  be  $T_2$  sans node  $x$
- (c) using the balance information stored at nodes along the right spine and  $height(T_1)$ , starting from  $root(T_1)$ , walk along the right spine of  $T_1$  to find a node  $y$  whose height equals to either  $height(T'_2)$  or  $height(T'_2) + 1 \leftarrow$  takes  $O(1 + height(T_1) - height(T'_2))$  time
- (d) hang  $T_y$  as the left child of  $x$  and  $T'_2$  as the right child of  $x$ ; make  $y.parent$  in  $T_1$  as the parent of  $x \leftarrow$  takes  $O(1)$  time
- (e) while walking along the simple path from  $x$  to root of the resultant tree, at every node  $v$ , if the balance condition is violated at  $v$ , fix with one/two left/right rotations as in insert algorithm for AVL trees<sup>2</sup>  $\leftarrow$  takes  $O(1 + height(T_1) - height(T'_2))$  time
  - let this tree be  $T$



\* Correctness: Since keys in  $T_y$  are  $\leq x.key$ , since keys in  $T'_2$  are  $> x.key$ , and since  $x.key > y.parent.key$ , the  $T$  is a BST. The violation of balance condition due to deletion of  $x$  from  $T_2$  is taken care of by the delete algorithm for AVL trees. The violations are fixed while traversing along the simple path from  $x$  to root. Hence, the resultant tree is an AVL tree.

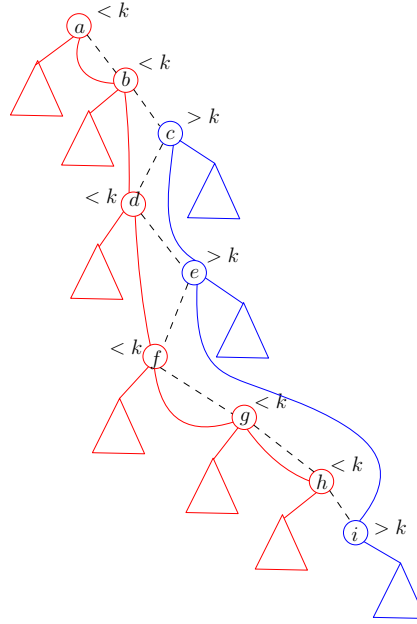
<sup>2</sup>like in insert algorithm, it suffices to fix the first violation along this path

\* This algorithm takes  $O(\text{height}(T_1) + \text{height}(T_2))$  time to join two AVL trees.

- Corollary: Given AVL trees  $T_1$  and  $T_2$  with their respective heights, and a key  $k$  such that every key in  $T_1$  is less than or equal to  $k$  and every key in  $T_2$  is greater than  $k$ , the algorithm to join  $T_1, T_2$  and  $k$  into an AVL tree takes  $O(1 + |\text{height}(T_1) - \text{height}(T_2)|)$  time.

Proof: Since step (a) and step (b) of the algorithm listed above are avoided.

- Given an AVL tree  $T$  with  $n$  keys and a key  $k$ , split  $T$  into two AVL trees  $T', T''$  such that (i)  $T'$  has every key of  $T$  less than or equal to  $k$  and  $T''$  has every key of  $T$  strictly greater than  $k$ , and (ii) the sum of the number of nodes in  $T'$  and the number of nodes in  $T''$  is equal to the number of nodes in  $T$ .



dashed line path is the search path for  $k$  in  $T$ ;  $T'$  is in red and  $T''$  is in blue

\* For easier understanding, the algorithm below is adapted to the above example:

(a) Using the balance information stored at nodes, compute the height of  $T \leftarrow$  takes  $O(\lg n)$  time

(b)  $T_1 = AVLJoin(T_{a.left}, h(T_{a.left}), a, T_{b.left}, h(T_{b.left}))$

$T_2 = AVLJoin(T_1, h(T_1), b, T_{d.left}, h(T_{d.left}))$

$T_3 = AVLJoin(T_2, h(T_2), d, T_{f.left}, h(T_{f.left}))$

$T_4 = AVLJoin(T_3, h(T_3), f, T_{g.left}, h(T_{g.left}))$

$T_5 = AVLJoin(T_4, h(T_4), g, T_{h.left}, h(T_{h.left}))$

$AVLInsert(T_5, h)$

$\leftarrow$  This algorithm takes  $O(\lg n)$  time, due to the following: heights of subtrees  $T_1, T_2, T_3, \dots$  can be determined as part of  $AVLJoin$ , heights of  $T_{a.left}, T_{d.left}, \dots$  are determined when roots of those

subtrees are encountered while exploring the search path for  $k$ , above corollary, and telescoping of terms involved<sup>3</sup> in summing time complexities of joins.

(c)  $T_1 = AVLJoin(T_{e.right}, h(T_{e.right}), c, T_{c.right}, h(T_{c.right}))$   
 $T_2 = AVLJoin(T_{i.right}, h(T_{i.right}), e, T_1, h(T_1))$   
 $AVLInsert(T_2, i)$

← This part of the algorithm also takes  $O(\lg n)$  time.

\* The correctness of this AVLSplit algorithm is immediate given the correctness of BST split algorithm. An algorithm for the latter was detailed and a proof of correctness for the same was provided in an earlier lecture.

---

<sup>3</sup>since the height of  $T_1$  is at most  $\max(h(T_{a.left}), h(T_{b.left})) + 2$ , since the height of  $T_2$  is at most  $\max(h(T_1), h(T_{d.left})) + 2$ , etc.