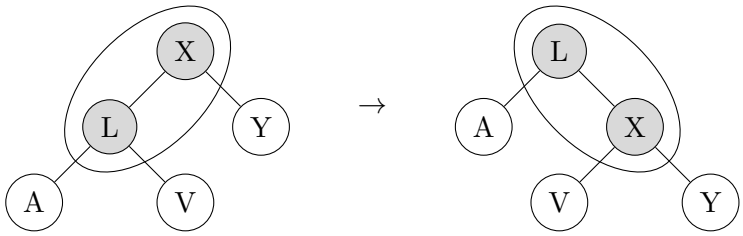
COMP20007 Design of Algorithms

Week 11 Workshop Solutions

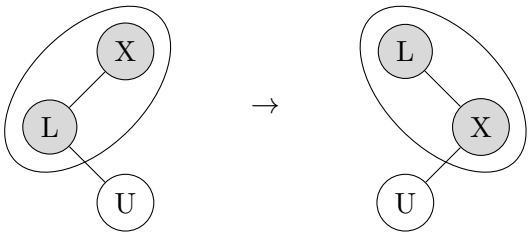
Tutorial

1. Rotations

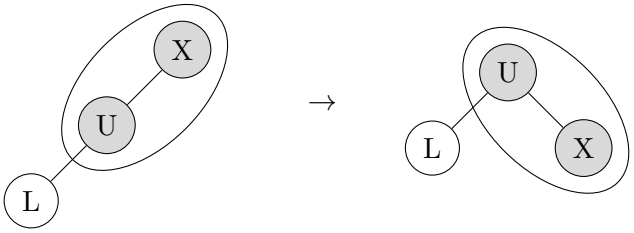
(a) doesn't improve overall balance:



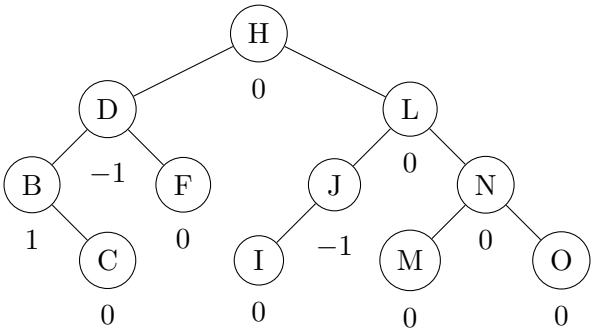
(b) doesn't improve overall balance:



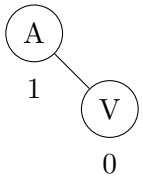
(c) *does* improve overall balance:



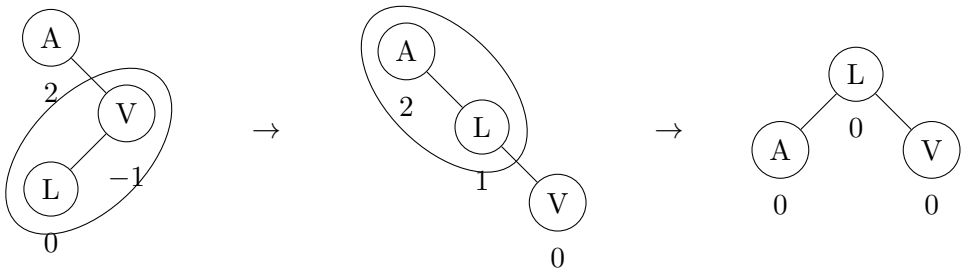
2. Balance factor Balance factor listed below each node. Calculated by subtracting the height of the node's left subtree from the height of its right subtree.



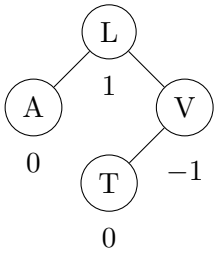
3. AVL Tree Insertion Insertion of A and V is fine, all balanced (balance factor below node):



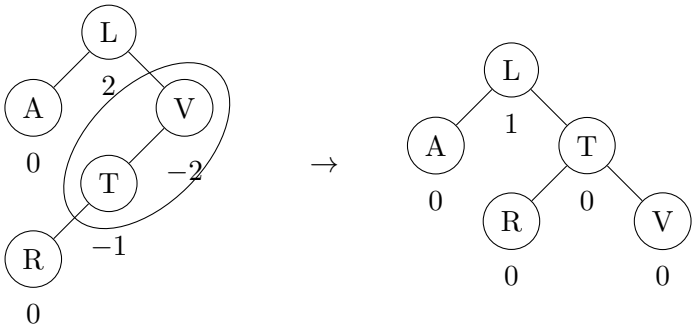
Inserting L causes an imbalance at A, and it's a zig-zag case so we need two rotations to fix it:



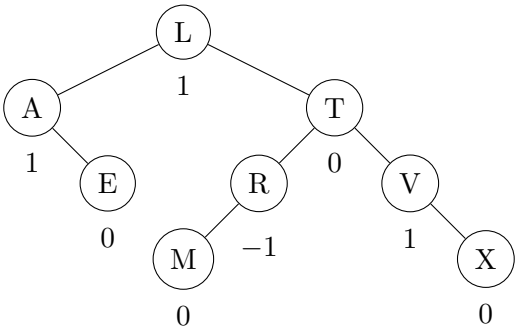
Then, inserting T is fine:



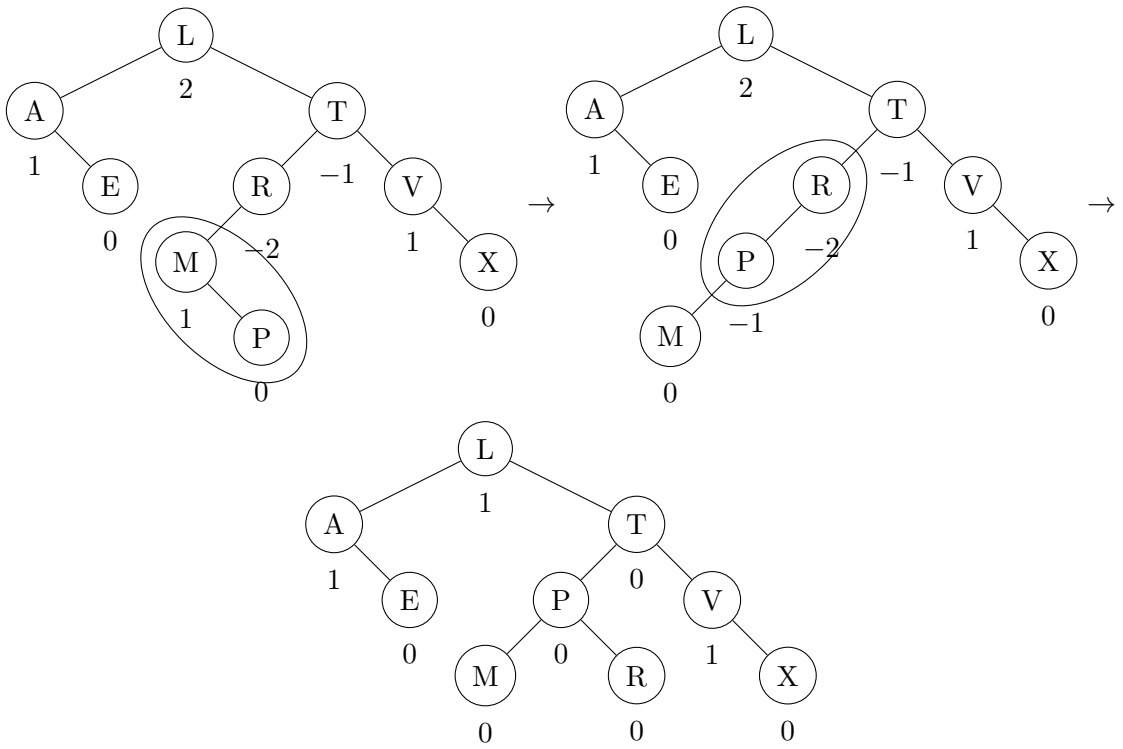
But inserting R gets us into trouble! It causes an imbalance at V. This time it's a zag-zag case, so we only need one rotation:



Next, insertion of E, X, and M cause no imbalances:



The final insertion, P, causes a zag-zig problem at R, once again fixed with two rotations:

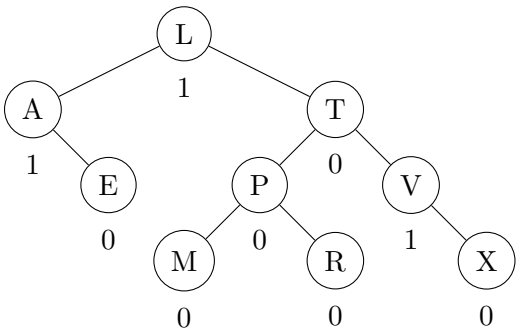


Notes:

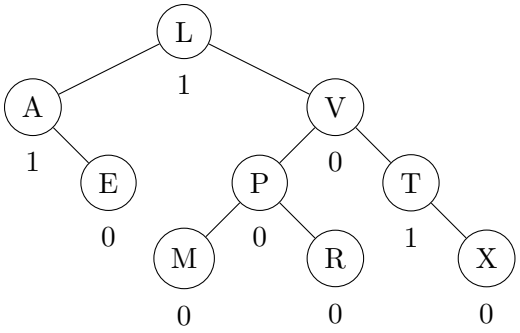
- The terms 'zig' and 'zag' are used to describe imbalances. A 'zig-zag' case, for example, refers to a node with an imbalance because of its right ('zig') subtree, and the imbalance in that subtree is coming from the left side ('zag').
- To determine which type of imbalance we are dealing with, we can look at the sign of the balance factor. If it's a +2, that means the problem is in the right child. If the right child has a -1 (different sign), then the problem is with it's left child, and it's a zig-zag case. Two rotations are needed. If the right child has a +1 (same sign), then the problem is with its right child, and it's a zig-zig case. One rotation is needed. Likewise, if the sign at the unbalanced node is -2, then the problem is with the left child. If the left child has a balance factor of -1 (same sign), then the problem is with its left child, i.e. it's a zag-zag case. One rotation is needed. If the left child has a balance factor of +1 (different sign), then the problem is with its right child, i.e. it's a zag-zig case. Two rotations are needed.
- Always deal with an imbalance at the deepest available point. For example, when inserting P, both R and L became unbalanced, but we dealt with the problem at R. This automatically fixed the problem at L. In reality, we only calculate balance factors on our way back up the tree from the insertion, so we can simply deal with the first imbalance (+2 or -2 balance factor) we encounter. All balance factors have been shown at all stages in the above diagrams, but in reality only the heights would be stored with each node — balance factors would only be calculated on the way back up the tree after an insertion, and only for the nodes on this direct path.

4. AVL Tree Deletion

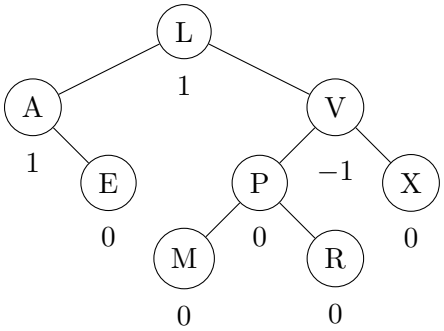
(a) In the deletion of T from the AVL tree we trigger Case 3, since we have two children we have the option of replacing T with the in-order successor (V) or the in-order predecessor (R). For the parts in this task we will choose to use the in-order successor.



→

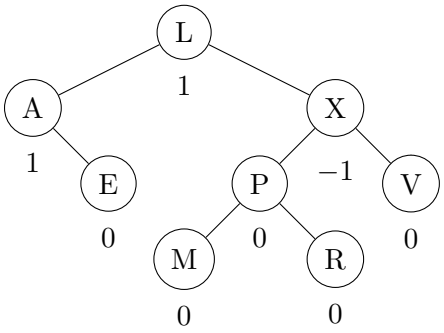


After swapping T with its in-order successor, we are left with a Case 2 situation. So T is finally removed by replacing it with its child.

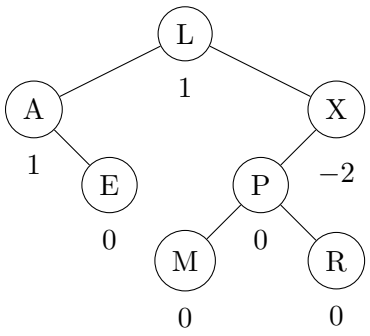


The AVL condition still holds for all parents of the deleted node so the deletion is finished.

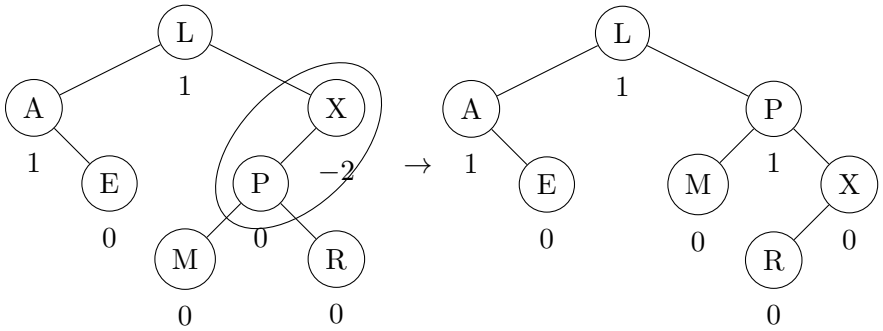
(b) In the deletion of V, we run into a Case 3 situation initially, so we again replace it with its in-order successor.



V is now in a Case 1 situation. So is simply deleted.

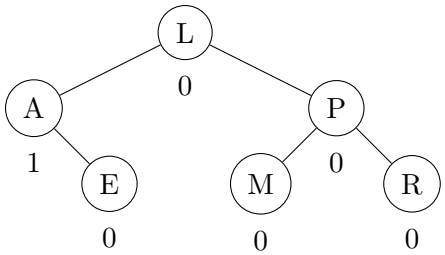


Unlike the previous deletion, we now have an imbalance to the left, so we fix it with a right rotation.

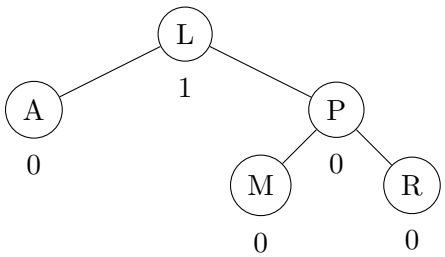


The tree is now in AVL condition.

(c) In deleting X we see it only has a single child, so falls into a Case 2 scenario and we can simply replace X with its child.

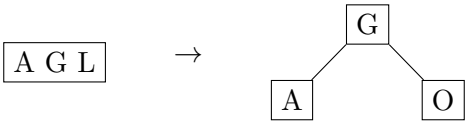


(d) To delete E, since it has no children, it falls into a Case 1 scenario and we can simply delete it.

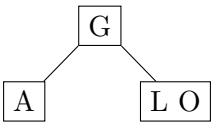


6. 2-3 Tree Insertion Inserting the first two elements results in a leaf node:

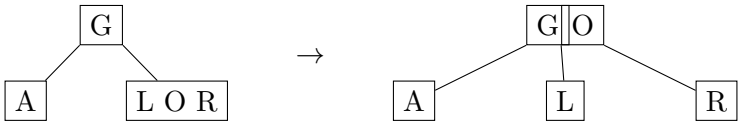
With the insertion of G, our leaf ends up with 3 elements, so it must be split up – we promote the middle element like so:



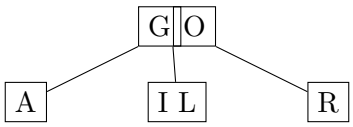
Inserting O adds to the right most leaf node:



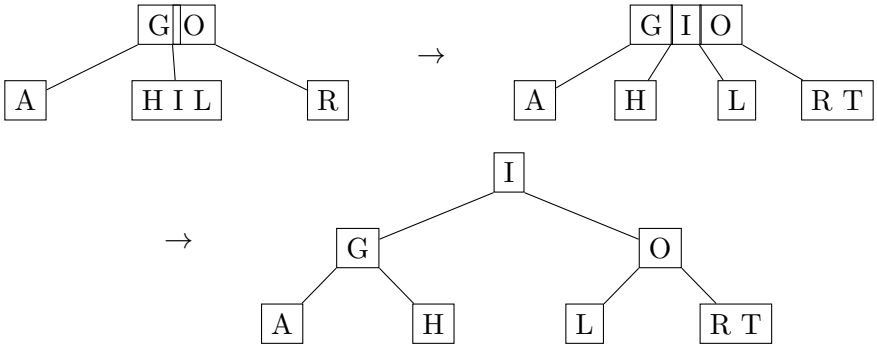
When we add R we get a node with 3 elements and must promote the middle one.



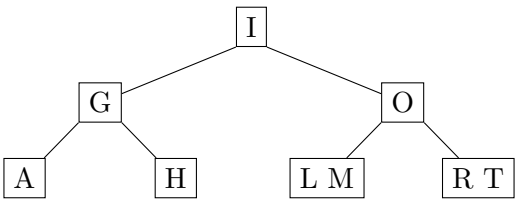
Adding I just adds to a leaf node.



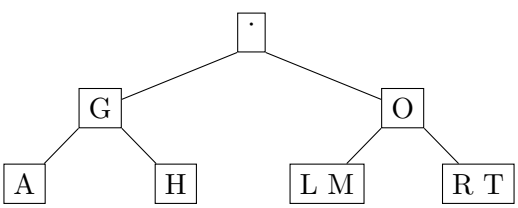
Adding H leaves a node with 3 elements, we will fix this my promoting I twice:



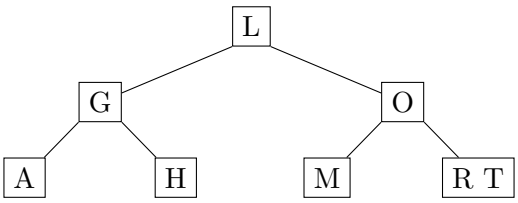
Finally, we can insert M into a leaf node without having to do any promotions, so the final 2-3 tree looks like:



5. 2-3 Tree Deletion Performing the first deletion (I), we first remove the root value:

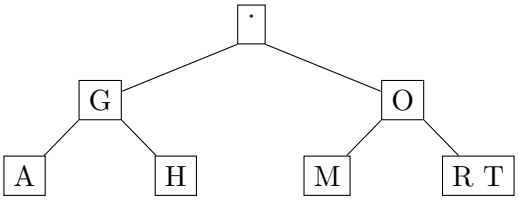


We then replace it with its in-order successor, L.

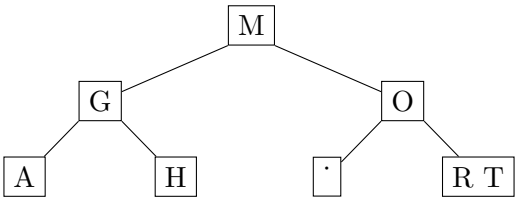


Since the node containing the successor contained two values, no more work needs to be done and the deletion of I is complete.

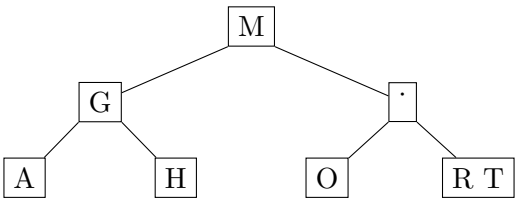
For the second deletion, we remove L.



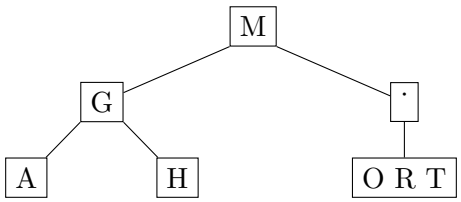
Like the first deletion, we replace it with the in-order successor, M.



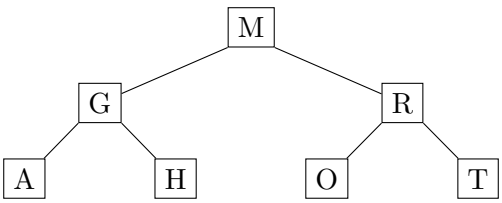
Unlike the first deletion, this creates a node with no items. To resolve this, we swap the hole with its parent, O.



As the hole is only allowed to have one child, we merge the two children into one.

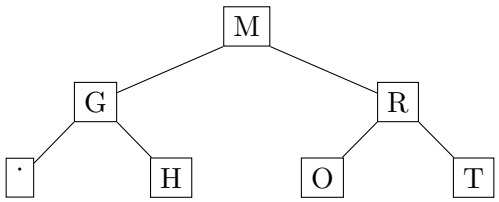


As the merged child has three items, we split it, moving the median up.

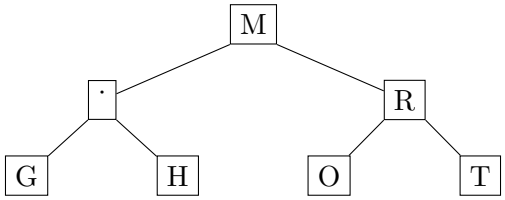


Since the item containing the hole now has more than one item, we can remove the hole and conclude the removal of L.

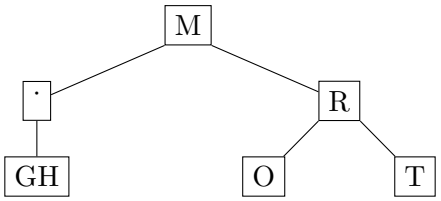
For the final removal (A), we again replace it with a hole.



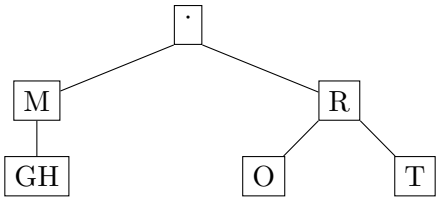
We swap the hole with its parent as usual.



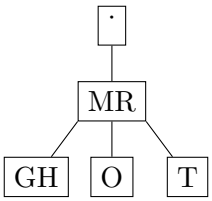
And again merge the two children to give the hole a single child.



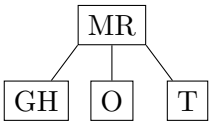
We again swap the hole with its parent.



Once again, the hole must have a single child, so we merge the two children.



As the hole is at the root of the tree, we can set the root to the single child of the hole and conclude the deletion of A.



7. Mergesort Time Complexity Recurrence relation:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

where $T(n)$ is the runtime of mergesort sorting n elements. The first $T(\frac{n}{2})$ is the time it takes to sort the left half of the input using mergesort. The other $T(\frac{n}{2})$ is the time it takes to sort the right half. $\Theta(n)$ is a bound on the time it takes to merge the two halves together.

Recall that the Master Theorem states that if we have a recurrence relation $T(n)$ such that

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d),$$
$$T(1) = c,$$

then,

$$T(n) \in \begin{cases} \Theta\left(n^d\right) & \text{if } a < b^d \\ \Theta\left(n^d \log n\right) & \text{if } a = b^d \\ \Theta\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}.$$

We can recognise that the mergesort recurrence relation fits the form required by the Master Theorem, with constants $a = 2$, $b = 2$, and $d = 1$.

$$b^d = 2 = a$$

so, by the master theorem, $T(n) \in \Theta(n \log n)$.