CS230 Data Structures
Prof. Lyn Turbak
Wellesley College

Handout # 28
Sunday, December 8, 2002

2-3 Trees

Balanced Search Trees

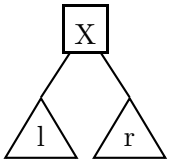
Many data structures use binary search trees or generalizations thereof. Operations on such search trees are often proportional to the height of the tree. To guarantee that such operations are efficient, it is necessary to ensure that the height of the tree is logarithmic in the number of nodes. This is usually achieved by some sort of *balancing* mechanism that guarantees that subtrees of a node never differ too much in their heights.

There are many forms of balanced search tree. Here we will study a particularly elegant form of balanced search tree known as a *2-3 tree*. There are many other forms of balanced search trees, some of which you will encounter in CS231. These include red-black trees, AVL trees, 2-3-4 trees, and B-trees.

2-3 Trees

A 2-3 tree has three different kinds of nodes:

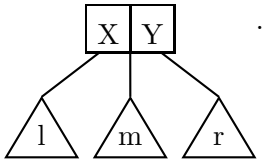
1. A leaf, written as ●.
2. A 2-node, written as



   X is called the **value** of the 2-node; l is its **left subtree**; and r is its **right subtree**. Every 2-node must satisfy the following invariants:

   (a) Every value v appearing in subtree l must be $\leq X$.
   (b) Every value v appearing in subtree r must be $\geq X$.
   (c) The length of the path from the root of the 2-node to *every* leaf in its subtrees must be the same.

3. A 3-node, written as



.

X is called the **left value** of the 3-node; Y is called the **right value** of the 3-node; l is its **left subtree**; m is its **middle subtree**; and r is its **right subtree**.

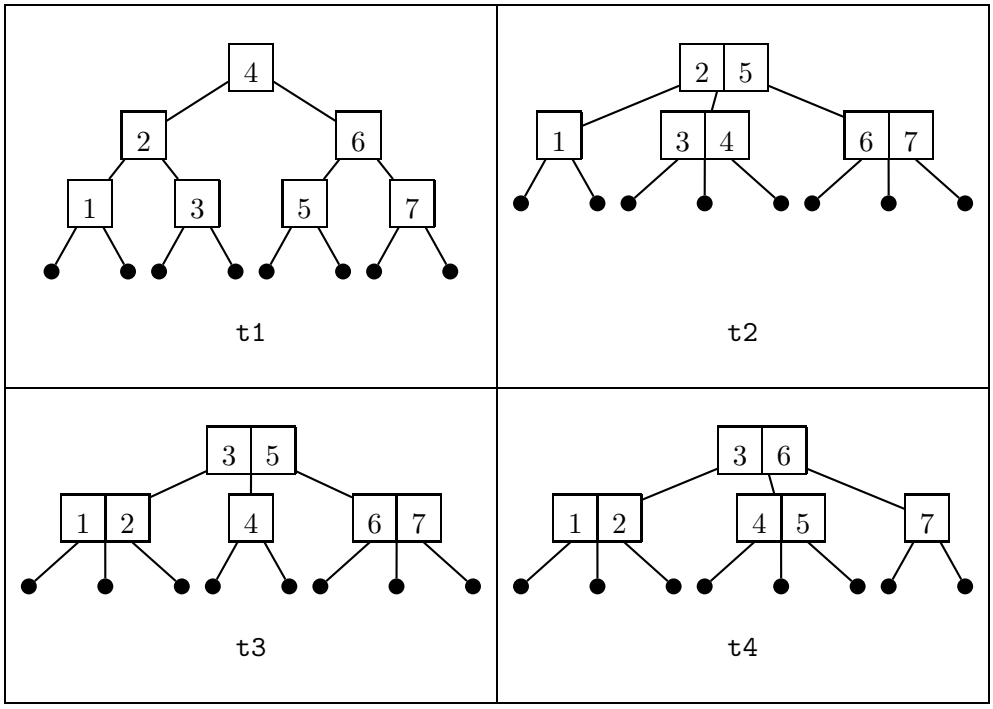
Every 3-node must satisfy the following invariants:

1. Every value v appearing in subtree l must be $\leq X$.
2. Every value v appearing in subtree m must be $\geq X$ and $\leq Y$.
3. Every value v appearing in subtree r must be $\geq Y$.
4. The length of the path from the root of the 3-node to *every* leaf in its subtrees must be the same.

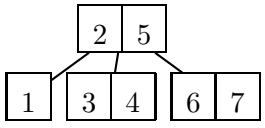
The path-length invariant on 2-nodes and 3-nodes means that 2-3 trees are necessarily balanced; the height of a 2-3 tree with n nodes cannot exceed $\log_2(n+1)$. Together, the tree balance and the ordered nature of the nodes means that testing membership in, inserting an element into, and deleting an element from a 2-3 tree takes logarithmic time.

2-3 Tree Examples

Given a collection of three or more values, there are several 2-3 trees containing those values. For instance, below are all four distinct 2-3 trees containing first 7 positive integers.



We shall use the term *terminal node* to refer to a node that has leaves as its subtrees. To save space, we often will not explicitly show the leaves that are the childred of a terminal node. For instance, here is another depiction of the tree `t2` above without the explicit leaves:

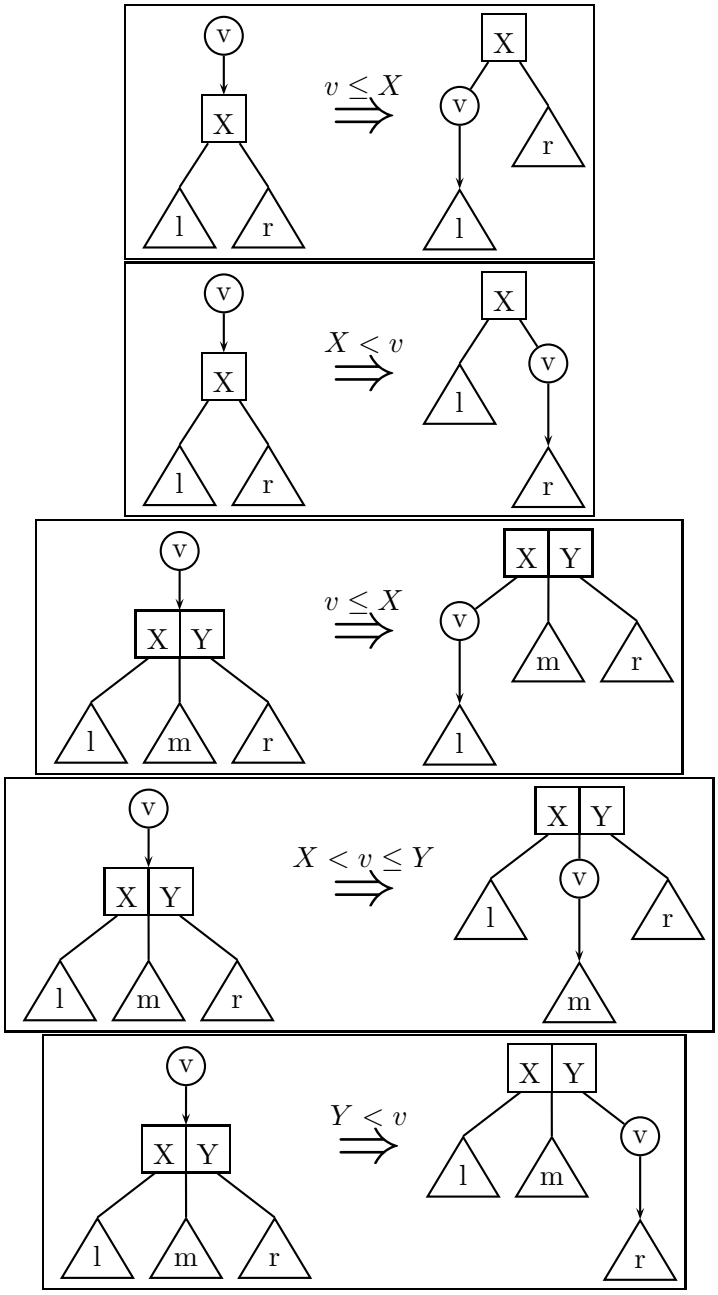


2-3 Tree Insertion: Downward Phase

When inserting an element v into a 2-3 tree, care is required to maintain the invariants of 2-nodes and 3-nodes. As shown in the rules below, the order invariants are maintained much as in a binary search tree by comparing v to the node values encountered when descending the tree and moving in a direction that satisfies the order invariants.

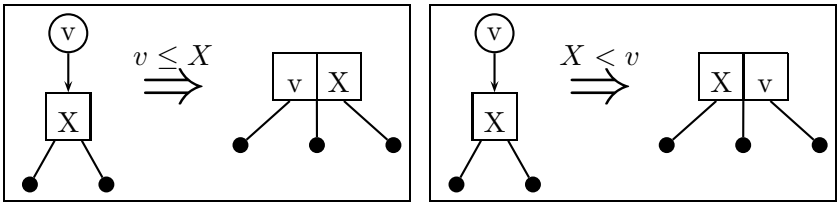
In the following rules, the result of inserting an element v into a 2-3 tree is depicted as a circled v with an arrow pointing down toward the tree in which it is to be inserted. X and Y are variables that stand for any elements, while triangles labeled l, m, and r stand for whole subtrees. In the case of insertion, such trees must be non-empty.

The rules state that elements equal to a node value are always inserted to the left of the node. This is completely arbitrary, they could be inserted to the right as well.



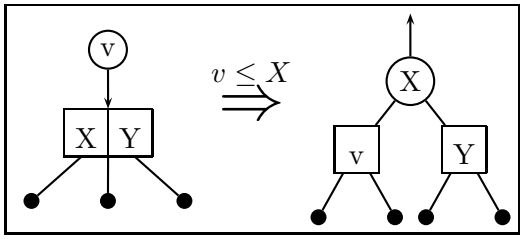
2-3 Tree Insertion: Terminal Cases

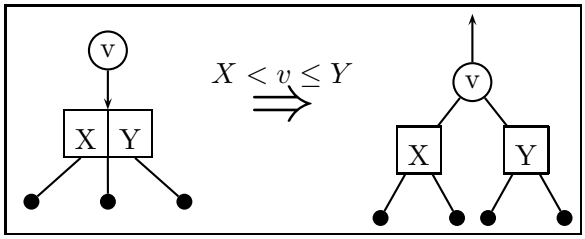
If v reaches a terminal 2-node with value X, the balance invariant can be maintained by absorbing the inserted value v in a newly constructed 3-node with values v and X, as shown in the following cases:

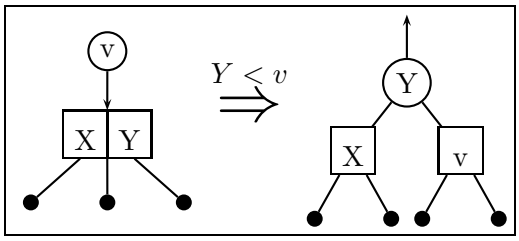


However, if If v reaches a terminal 3-node with values X and Y, the value v cannot be absorbed. But it is possible to split the three values v, X, and Y into two terminal 2-nodes and a third value that is "kicked upstairs" because there is no room "downstairs":

In the following rules, the value "kicked upstairs" is drawn in a circle with an arrow pointing up from it and with two subtrees that result from the split. We will consider the height of such a configuration to be the height of the subtrees – i.e., the height does not include the circled element. It is an invariant that the heights of the two subtrees of a kicked-up node will be identical.

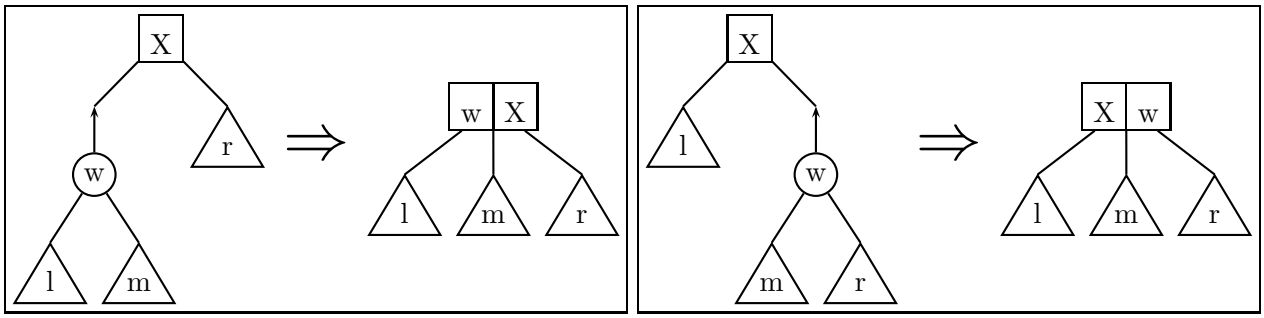




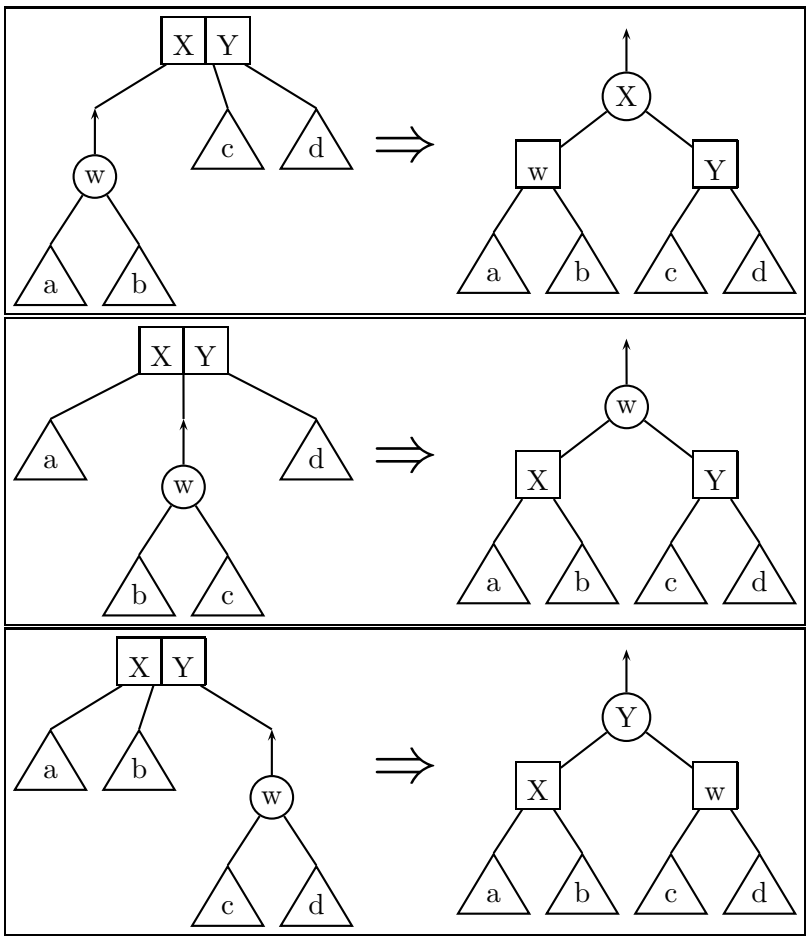


2-3 Tree Insertion: Upward Phase

If there is a 2-node upstairs, the value kicked upward (call it w) can be absorbed by the 2-node:



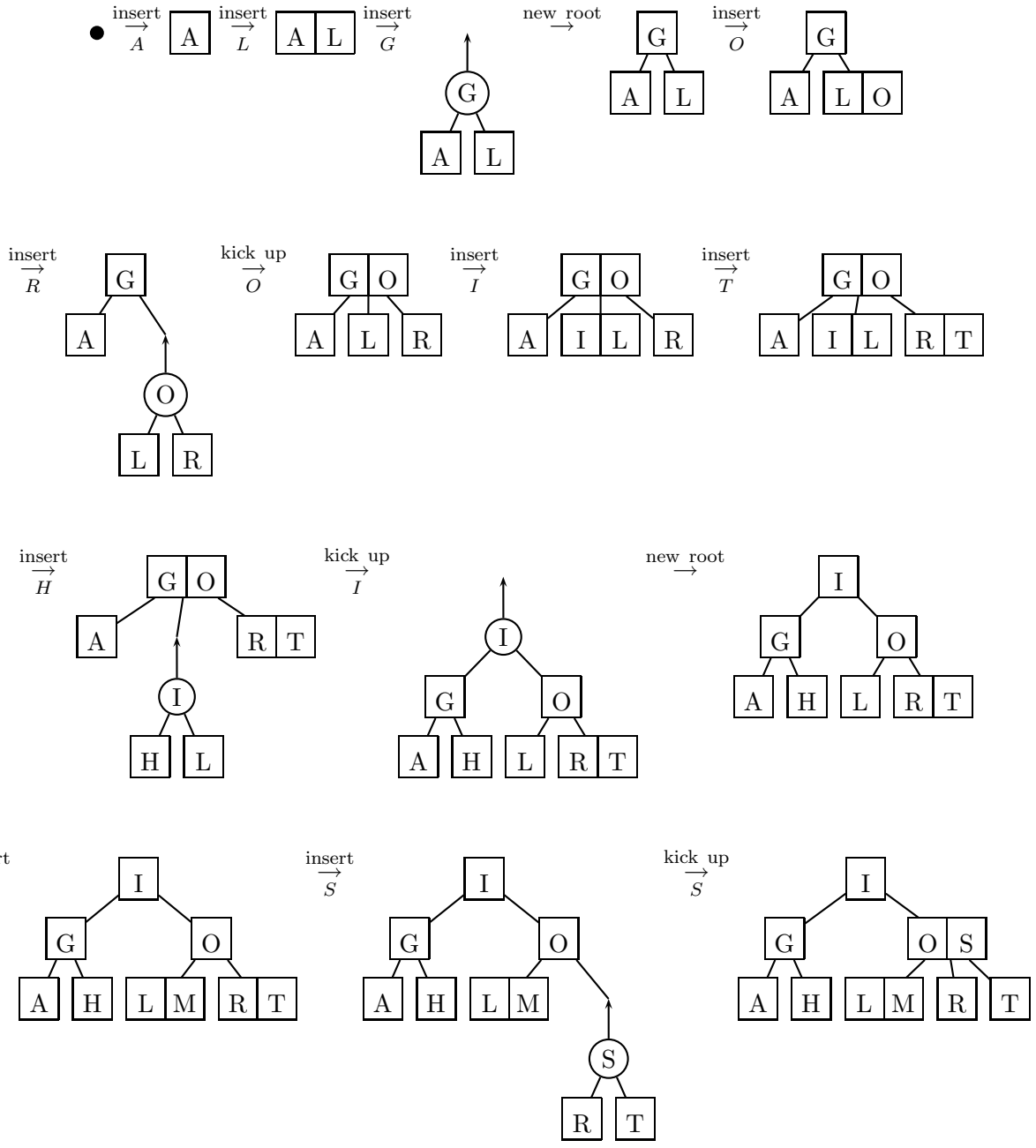
If there is a 3-node upstairs, it cannot simply be absorbed, and another split takes place in which yet another value is kicked upstairs. This process continues until either the kicked-up value is absorbed, or the root of the tree is reached. In the latter case, the kicked-up value becomes the value of a brand new a new 2-node that increases the height of the tree by one. This is the only way that the height of a 2-3 tree can increase.



You should convince yourself that path lengths and element order are unchanged by the downward or upward phases of the insertion algorithm. This means that the tree result from insertion is a valid 2-3 tree.

2-3 Tree Insertion Example

Here we show the letter-by-letter insertion of the letters `A L G O R I T H M S` into an initially empty 2-3 tree.



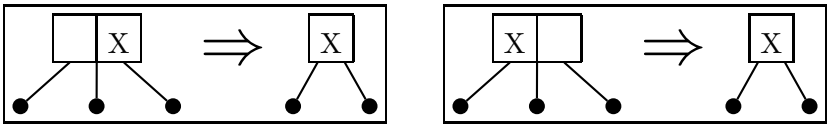
2-3 Tree Deletion: Downward Phase

The downward phase for deleting an element from a 2-3 tree is the same as the downward phase for inserting an element *except* for the case when the element to be deleted is equal to the value in a 2-node or a 3-node. In this case, if the value is not part of a terminal node, the value is replaced by its in-order predecessor or in-order successor, just as in binary search tree deletion. This leaves a *hole* in a terminal node.

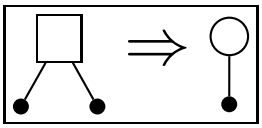
The goal of the rest of the deletion algorithm is to remove the hole without violating the other invariants of the 2-3 tree.

2-3 Tree Deletion: Terminal Cases

Handling the removal of a hole from a terminal 3-node is easy: just turn it into a 2-node!



To deal with a hole in a terminal 2-node, we consider it to be a special *hole node* that has a *single* subtree.



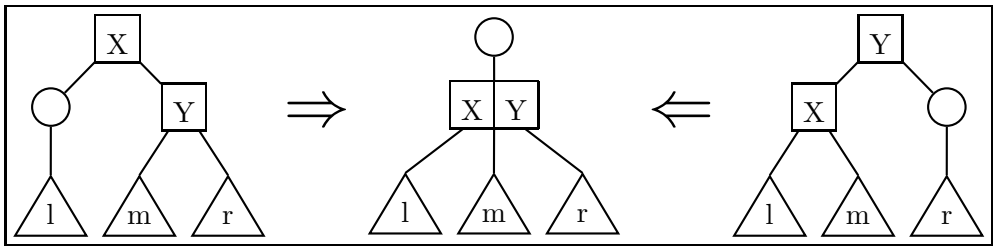
For the purposes of calculating heights, such a hole node *does* contribute to the height of the tree. This decision allows the the path-length invariant to be preserved in trees with holes.

2-3 Tree Deletion: Upward Phase

The goal of the upward phase of 2-3 tree deletion is to propagate the hole up the tree until it can be eliminated. It is eliminated either (1) by being "absorbed" into the tree (as in the cases 2, 3, and 4 below) or (2) by being propagated all the way to the root of the 2-3 tree by repeated applications of the case 1. If a hole node propagates all the way to the top of a tree, it is simply removed, decreasing the height of the 2-3 tree by one. This is the only way that the height of a 2-3 node can decrease.

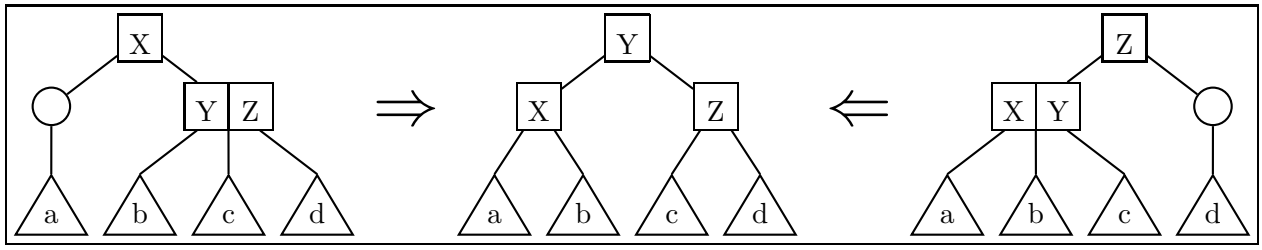
There are four cases for hole propagation/removal, which are detailed below. In the following rules, triangles stand for subtrees, which may possibly by empty. You should convince yourself that each rule preserves both the element-order and path-length invariants.

1. *The hole has a 2-node as a parent and a 2-node as a sibling.*



In this case, the heights of the subtrees l, m, and r are the same.

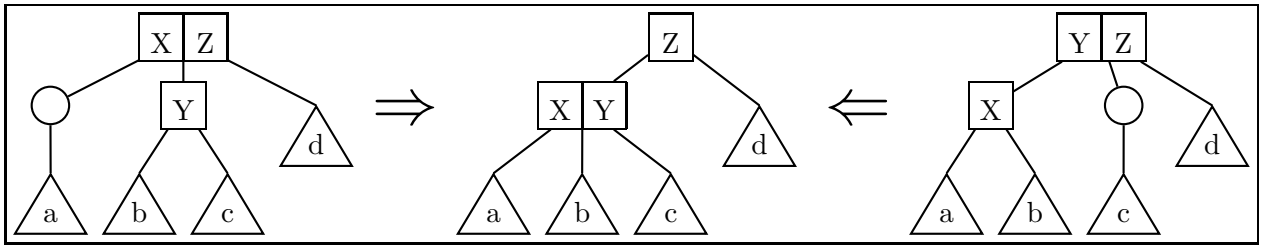
2. *The hole has a 2-node as a parent and a 3-node as a sibling.*



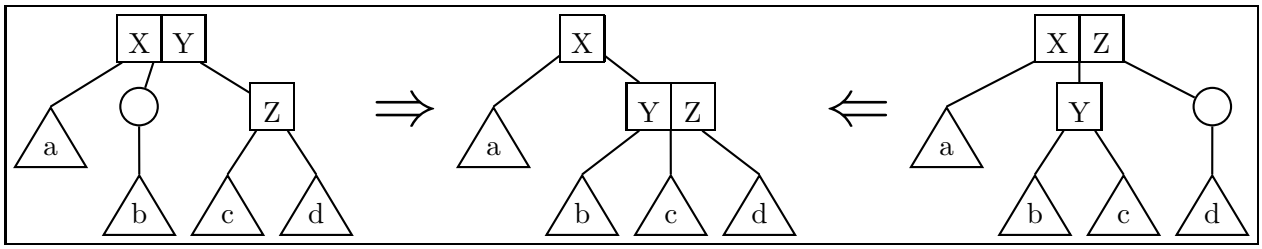
In this case, the heights of the subtrees a, b, c, and d are the same.

3. *The hole has a 3-node as a parent and a 2-node as a sibling.* There are two subcases:

   (a) The first subcase involves subtrees a, b, and c whose heights are one less than that of subtree d.

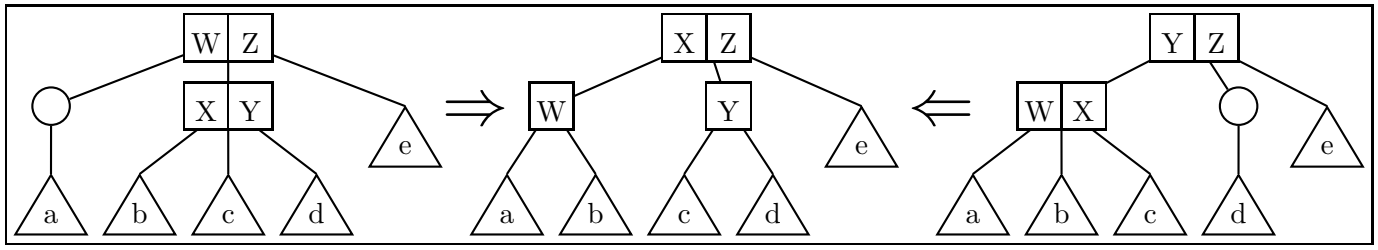


   (b) The second subcase involves subtrees b, c, and d whose heights are one less than that of subtree a. When the hole is in the middle, there may be ambiguity in terms of whether to apply the right-hand rule of the first subcase or the left-hand rule of the second subcase. Either application is OK.

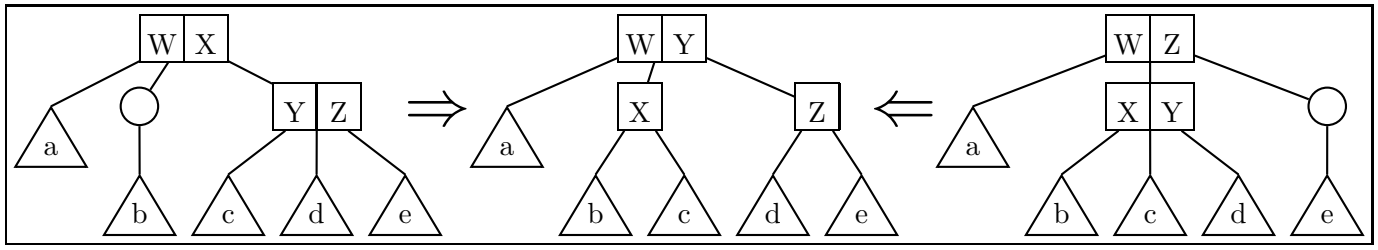


4. *The hole has a 3-node as a parent and a 3-node as a sibling.* Again there are two subcases.

   (a) The first subcase involves subtrees a, b, c, and d, whose heights are one less than that of subtree e.

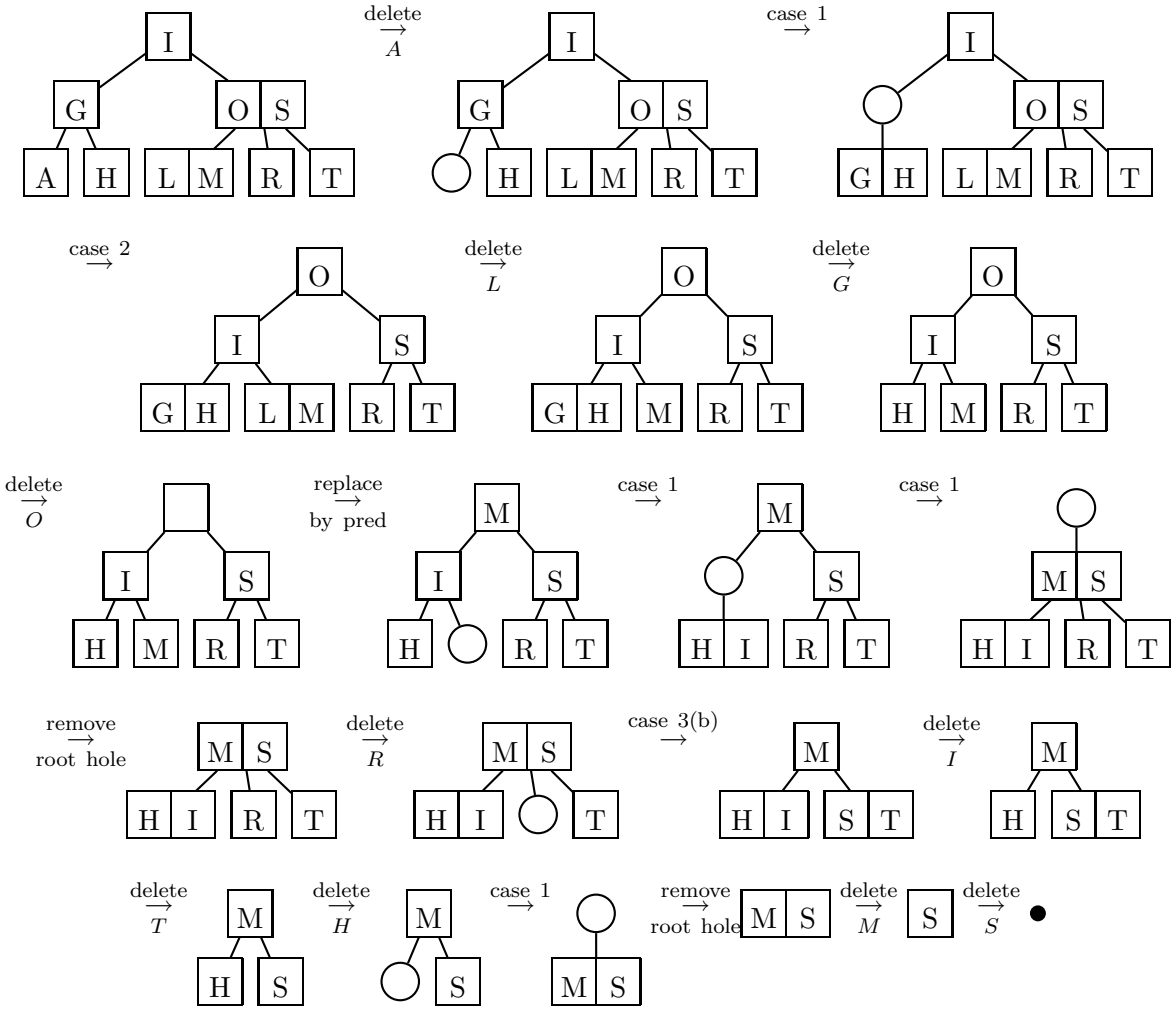


   (b) The second subcase involves subtrees b, c, d, and e, whose heights are one less than that of subtree a. When the hole is in the middle, there may be ambiguity in terms of whether to apply the right-hand rule of the first subcase or the left-hand rule of the second subcase. Either application is OK.

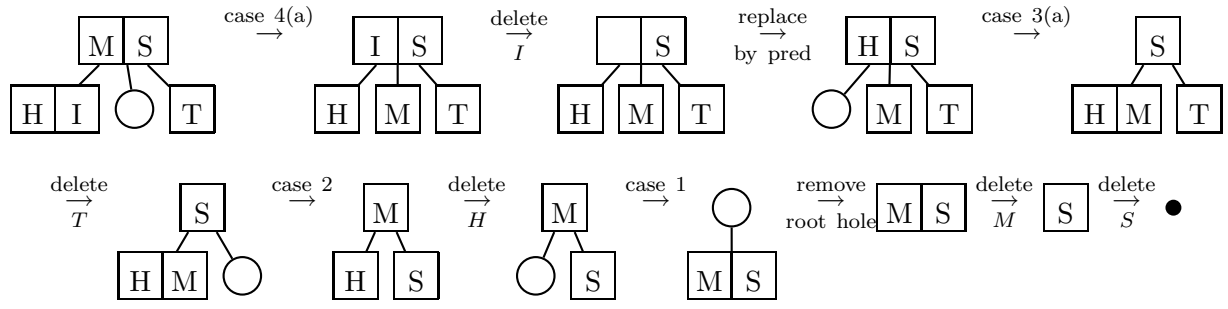


2-3 Tree Deletion Example

Here we show the letter-by-letter deletion of the letters `A L G O R I T H M S` from the final 2-3 tree of the insertion example:



At the point where case 3(b) was applied, it is also possible to apply case 4(a) instead. This leads to the following alternative deletion sequence:



There are some other choices in the above sequences. In each of the spots where a deleted value in a non-terminal node was replaced by its in-order predecessor, it could have been replaced by its in-order successor.