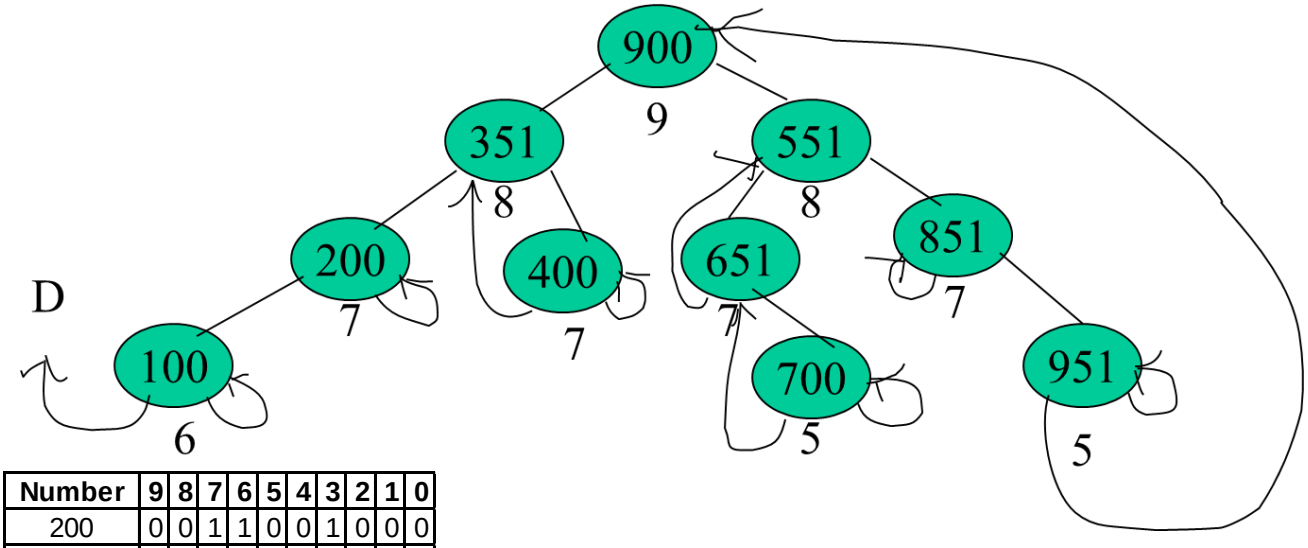
Algorithm for Deletions from a Patricia Tree

1) **First, check to see if your tree is empty. If it is you cannot delete any values so return your error message.**
2) **Otherwise, your tree is not empty. Search in the existing tree for the Item you are deleting, keeping track of the bit tests as you search.**
3) **Once the most recent bit test is equal to or higher numerically than the previous bit test, stop your search.**
4) **Compare the Old (by Old we mean pre-existing in the tree) Item you finished your search with to the Item you are deleting in ALL bits. If they agree in all bits then you have found your item.**
5) **We will need the following definition as well: We will call a node a 'true child' of its parent if the node is pointed to by one of the parent's pointers (as a left or right child) AND if the bit test in the child is strictly less than the bit test in the parent. We will use the example tree below to point out some of the cases for deletion:**

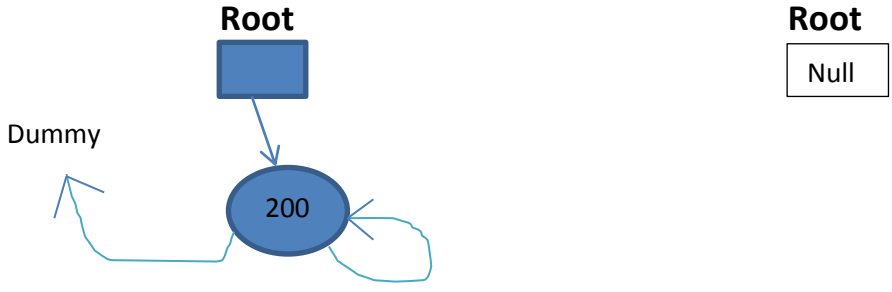


Number	9	8	7	6	5	4	3	2	1	0
200	0	0	1	1	0	0	1	0	0	0
351	0	1	0	1	0	1	1	1	1	1
900	1	1	1	0	0	0	0	1	0	0
551	1	0	0	0	1	0	0	1	1	1
400	0	1	1	0	0	1	0	0	0	0
651	1	0	1	0	0	0	1	0	1	1
700	1	0	1	0	1	1	1	1	0	0
851	1	1	0	1	0	1	0	0	1	1
100	0	0	0	1	1	0	0	1	0	0
951	1	1	1	0	1	1	0	1	1	1

Cases:

a) **(tree with one node, the root node case) if the node did not have a true parent, then the node that we found was the root node. After deleting this node the root should indicate the tree is empty.**

Before: **After:**



b) **If the node p to be deleted has zero true children: This implies the node points to itself in one of its pointers and points "up above" to a node higher in the tree in its other pointer. Point the parent of p to this "upward node" that p pointed to by setting the pointer that previously pointed to p to point to this upward node. Do not change any bit tests.**

Note: for the Patricia tree on the previous page, deletion of 100, 400, 700 and 951 are examples of this case.

c) **If the node to be deleted has a true child:**

i) **Case: The node p to be deleted is its own search predecessor. Here we remove it by changing the pointer in p's parent to point to the deleted node's branch that doesn't point to itself. This essentially "pulls up" the single true child. Do not change the bit test on the child we are pulling up.**

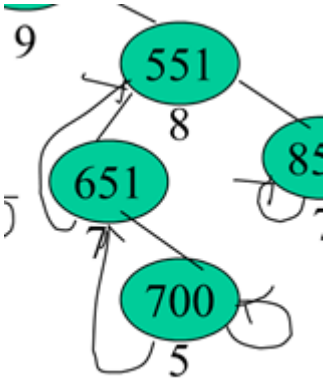
Note: for the Patricia tree on the previous page, deletion of 200 and 851 are examples of this case.

ii) **Case: The node to be deleted is not its own search predecessor. Replace the value in the node with that of its search predecessor, do not change the bit test in this node. Now physically delete the search predecessor. Physically deleting the search predecessor can result in –**

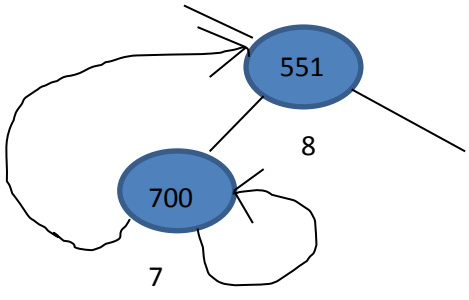
(1) **A deletion of a node with zero true children (see case b above and illustration below).**

Note: for the Patricia tree on the previous page, deletion of 351, 900 and 651 are examples of this case and deletion of 651 is illustrated below. Notice that we have copied the value of the search predecessor, 700, into the node previously containing 651. We do not change the bit test of this node. Now we physically delete the search predecessor and this is a case b deletion.

Before:



After:

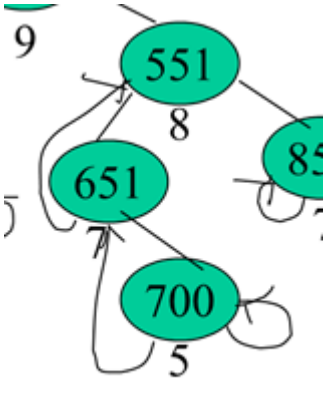


Number	9	8	7	6	5	4	3	2	1	0
200	0	0	1	1	0	0	1	0	0	0
351	0	1	0	1	0	1	1	1	1	1
900	1	1	1	0	0	0	0	1	0	0
551	1	0	0	0	1	0	0	1	1	1
400	0	1	1	0	0	1	0	0	0	0
651	1	0	1	0	0	0	1	0	1	1
700	1	0	1	0	1	1	1	1	0	0
851	1	1	0	1	0	1	0	0	1	1
100	0	0	0	1	1	0	0	1	0	0
951	1	1	1	0	1	1	0	1	1	1

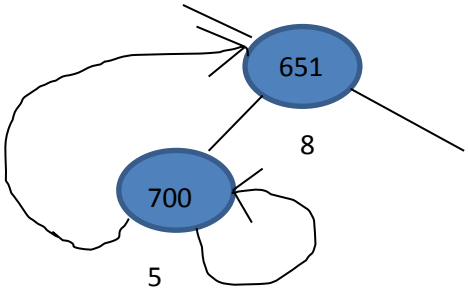
(2) **A deletion of a node with exactly one true child (why exactly one?). If you are in this case, pull up the one true child of the predecessor and set the pointer of this child that is not pointing to a true child to point up to the original location you deleted from. That is make this pointer point to where you copied the first predecessor you found in the deletion process.**

Note: for the Patricia tree at the beginning of this discussion, deletion of 551 is an example of this case and is illustrated below. First, we copy the value of the search predecessor, 651, up into the location previously occupied by 551, the value to be deleted. Now we physically delete the search predecessor, "old" 651. Since 651 has one true child, 700, we pull up that child to be the left child of the "new" 651 (in the after picture). Since this child is the search predecessor of 651, we set the appropriate pointer (in this case the left pointer) on the child we pull up to point "up" to its search predecessor, 651.

Before:



After:



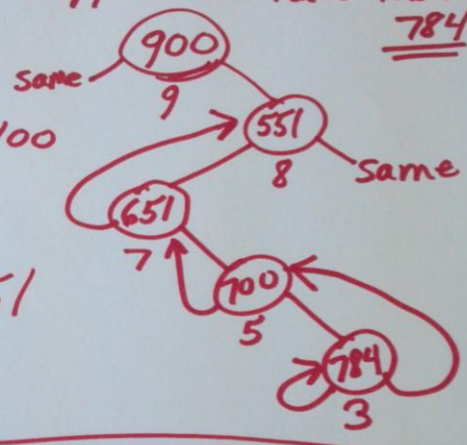
Number	9	8	7	6	5	4	3	2	1	0
200	0	0	1	1	0	0	1	0	0	0
351	0	1	0	1	0	1	1	1	1	1
900	1	1	1	0	0	0	0	1	0	0
551	1	0	0	0	1	0	0	1	1	1
400	0	1	1	0	0	1	0	0	0	0
651	1	0	1	0	0	0	1	0	1	1
700	1	0	1	0	1	1	1	1	0	0
851	1	1	0	1	0	1	0	0	1	1
100	0	0	0	1	1	0	0	1	0	0
951	1	1	1	0	1	1	0	1	1	1

See one more e.g. on next page!

Another example of this last case: Suppose we have inserted 784

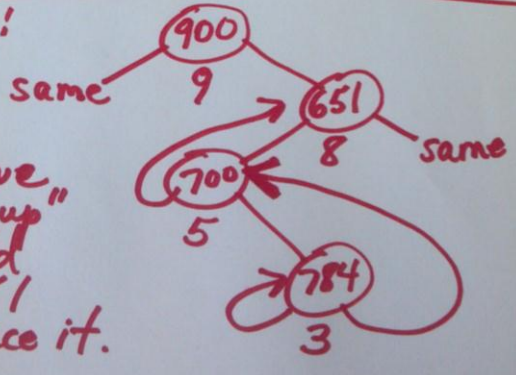
Before!

where -
784 = 1010101100



NOW DELETE 551

After!



we have "pulled up" the child of 651 to replace it.