CSE 326
Lecture 10

Spring Quarter 1997

AVL Tree Deletion

- Iterative Approach

  1. Do normal tree deletion all the way to the physical deletion, storing the deletion path on a stack.

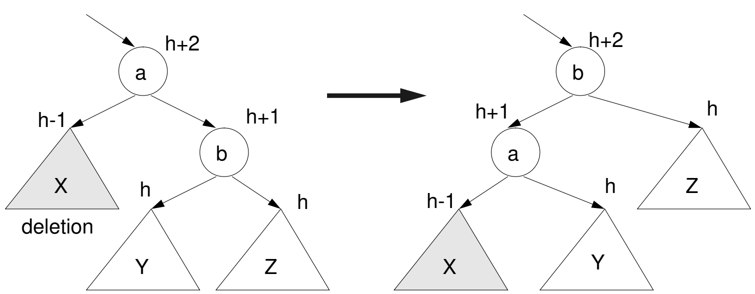
  2. Popping the stack update the height fields looking for an imbalance. If imbalance is found do a rotation. The direction of the rotation will be opposite the direction of deletion. Continue updating height fields if need. Note: more than one rotation may be needed.

- Recursive Approach

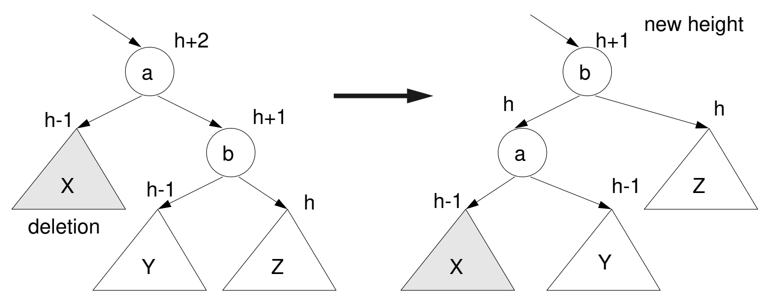
  1. Recursively Delete all the way to the node that is physically deleted.

  2. On return, check for an imbalance. If imbalance do a rotation and update the height fields in all cases.

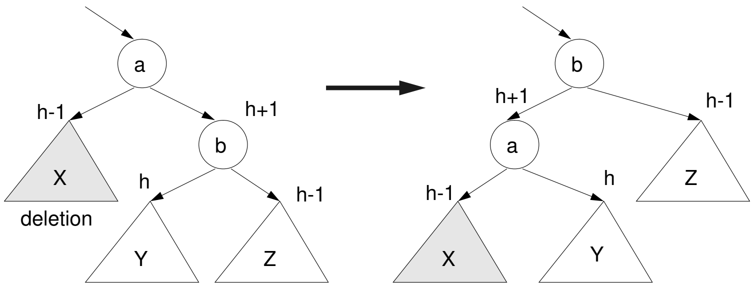
Single Rotation with no Propagation in Deletion



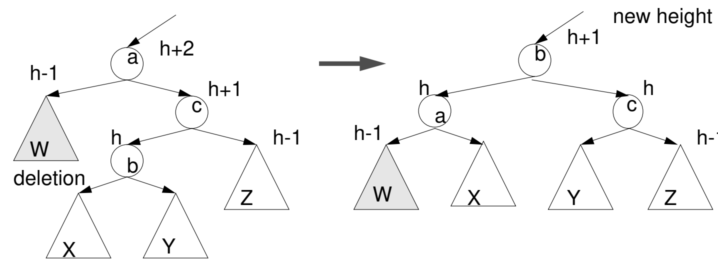
Single Rotation with Propogation in Deletion



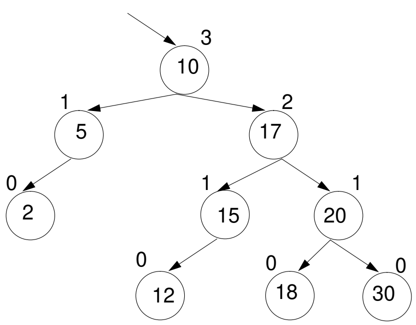
Why Single Rotation is not Enough in Deletion



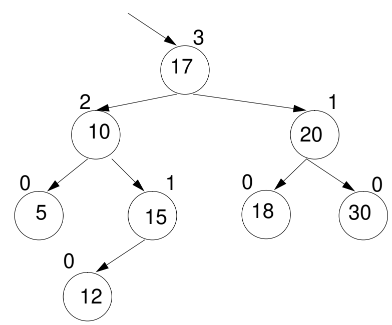
Double Rotation in Deletion



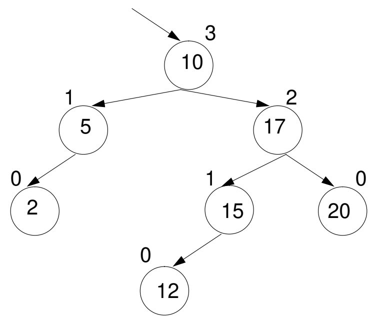
Delete 2



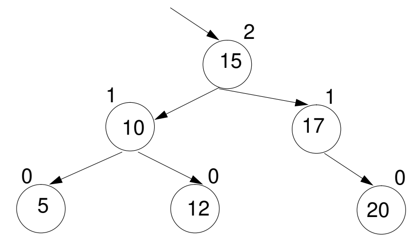
Result of Deleting 2



Delete 2 Again



Result of Deleting 2



Batch Deletion in AVL Trees

- Recall that we simply mark nodes as deleted.
- There are two steps

  1. Do an inorder traversal putting all the unmarked nodes in an array.

  2. Do a divide and conquer construction of a balanced tree, which will have the AVL tree property.
- Suppose we have m deletions in a tree of size n.

  Ordinary AVL deletion time is O(m log n + n)

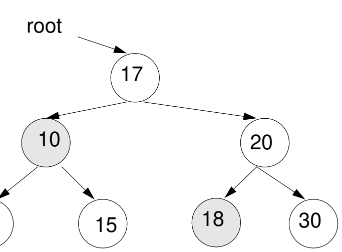
  Batch deletion is O(n) + the time to mark deleted nodes.

Listing the Unmarked Items

- List(n,i) puts the items in the tree rooted at n in order in the array A starting at index i.
- List(n, ref i) {

```
if not(n=null) do {
   List(n.left, i);
   if not(n.mark) then {A[i] := n.data; i := i+1};
   List(n.right,i) }}
```

- Call List(root,1)



Creating the AVL Tree

- Assume A[1..N] is a sorted array. The procedure Build(i,j) will return a balanced tree whose nodes have data A[i],A[i+1],...,A[j].
- Build(i,j) {

```
if j < i then return(null)
else {
  k := (i+j)/2;
  return(New_Node(A[k],Build(i,k-1),Build(k+1,j)))  }}
```

- Call Build(1,N) to create the entire tree.
- Fill in the height fields correctly with a postorder traversal.
- Running time is O(n) which is better than inserting items individually, which would be O(n log n).

Example of Creating An AVL Tree

- 5 , 8, 10, 15, 17, 20, 30, 35, 40

Time Complexity of Building a Balanced Tree

$$T(0) \le b$$

$$T(n) \le T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + c$$

- We can prove by induction that T(n) ≤ (b+c)n + b.

$$T(0) \le b \;=\; (b+c)\,0 + b$$

$$T(n) \le (b+c)\left\lceil \frac{n-1}{2} \right\rceil + (b+c)\left\lfloor \frac{n-1}{2} \right\rfloor + c \;=\; (b+c)\,(n-1) + c \;=\; (b+c)\,n + b$$

Summary of AVL Trees

- The worst cast height of an AVL tree is approximately 1.44 log n
- Complexity of Operations

  Insert - O(log n)
  Delete - O(log n)
  Find - O(log n)
- Features

  - At most one rotation needed to rebalance in insertion
  - Multiple rotations may be needed to rebalance on deletion
  - Compatible with lazy deletion.
  - height field can be implemented with 2 bits for -1,0,1
- Alternatives to AVL (height balanced) trees

  - weight balanced trees - not nearly as nice as height balanced trees
  - red-black trees - a constant number of rotations needed to rebalance on both insertion and deletion
  - splay trees - self adjusting binary trees with several good properties but does not have worst case O(log n) per operation.