eecs 281 Data Structures and Algorithms

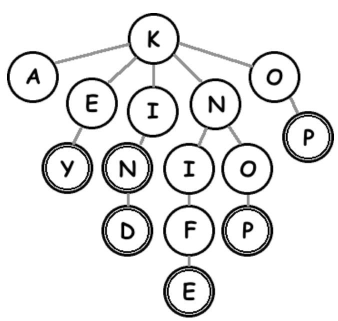
Midterm Review: Week of Oct 17, 2011

Summary

- 93 Survey Responses
- 5 most-requested topics:
  - Tries
  - AA Trees
  - Multi-Way Trees
  - Red-Black Trees
  - AVL Trees

Tries

- Consider implementing by a Trie
  - Nodes don't contain the character just for demonstration
  - Runtime of search ?
  - Memory of Trie ?



Tries

- Given a dictionary with words {"A", "to", "tea", "ted", "ten", "i", "in", and "inn"}
- Consider implementing by a BST
  - Runtime of search ?
  - Memory of BST ?

Tries—Insert

```
void insert(String input, int len, tree t){
  /*base case—inserting last character of input*/
  if(len == 1){
    if(!(t->children.contains(input.substr(0, 1)))){ //first char
      t->children.add(input);
    }
    return;
  }

  /*recursive call—adding node if needed and
   *calling insert with input as everything but the
   *first letter of the input
   *e.g. old input = dogs, new input = ogs
   */
  if(!(t->children.contains(input.substr(0, 1)))){
    t->children.add(input.substr(0, 1));
  }
  insert(input.substr(1, len – 1), len – 1, t->children.get(input.substr(0, 1));
}
```

AA Trees

Definition:

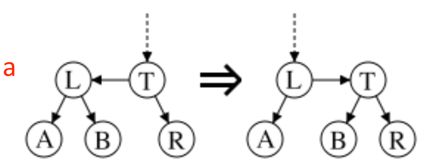
1) Every node is red or black
2) Root is black
3) External nodes are black
4) If a node is red, children must be black
5) Black height must be constant (number of black nodes from a node to a leaf node is the same regardless of path)
6) Left child cannot be red

AA Trees: Two Cases that Need Correcting

- So, you're inserting or a node in the tree as you normally would based upon the ordering condition (e.g. R-> less than, L-> greater than)
- When you do this either you violate the integrity of the tree or you don't
- If you DON'T violate the integrity, you're done
- When you DO violate the integrity, you've made one of two cases:

AA Tree Integrity Violation 1: Left Horizontal Link

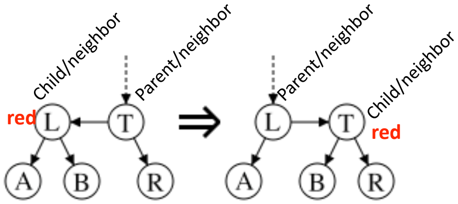
i.e. a node has a red left child/ neighbor



Node "L" is red

This violates the integrity of an AA Tree

How to fix a Red Left Child?

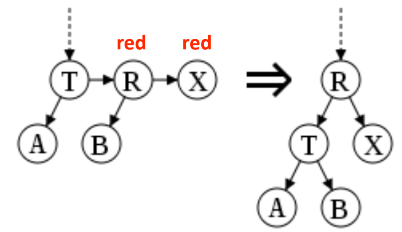


With an operation called "skew":

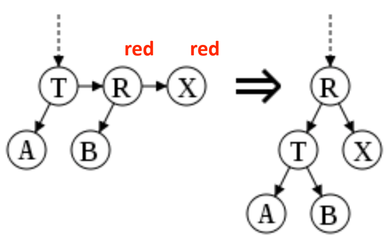
```
skew(…){
  -set red left child/neighbor (L) as parent/neighbor to old
   parent/neighbor (T) (swap direction of pointer)
  -set old parent/neighbor (T)'s old parent (dotted line) to
   new parent/neighbor (L) as its child
  -set new parent/neighbor (L)'s right subtree (B) as new
   child/neighbor's (T)'s left child
}
```

AA Tree Integrity Violation 2: Double Right Horizontal Link

- (i.e. red node has a right red node)



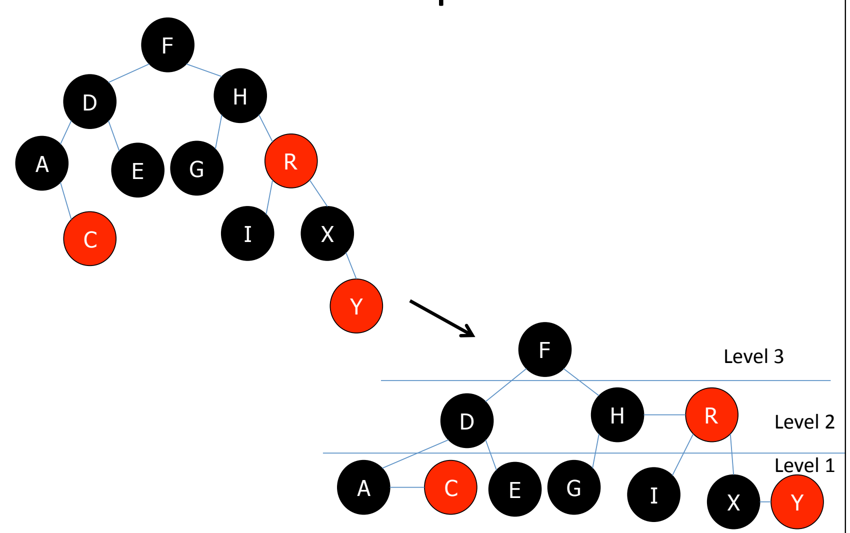
How to fix 2 Consecutive (Right) Reds?



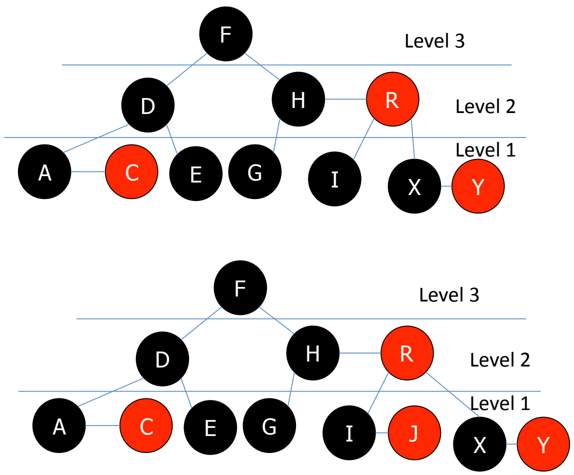
With an operation called "split":

```
split(…){
  -elevate middle red node (R) to a parent with it's    left/
neighbor (T) and right/neighbor (X) as children
  -set old middle node's (R) old children (B) as grandchildren
   (make B a child of T)
 }
```

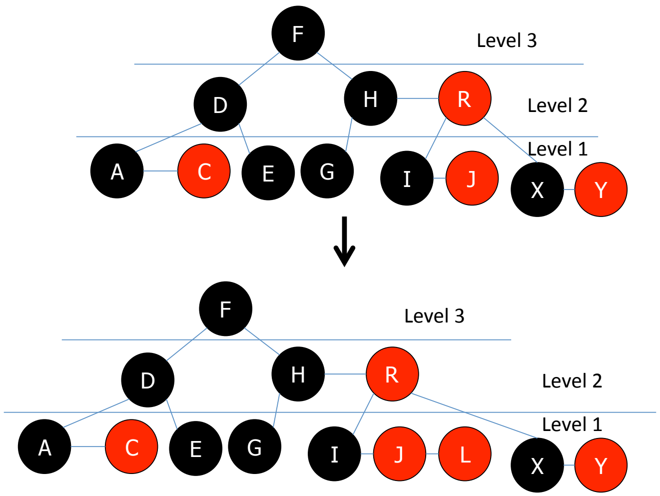
Example



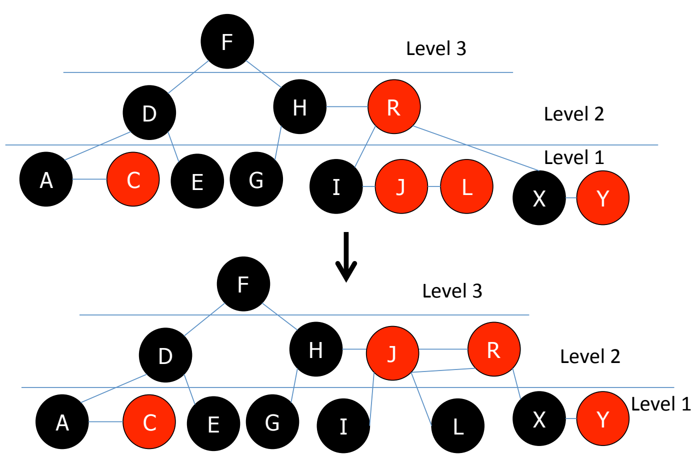
Insert J



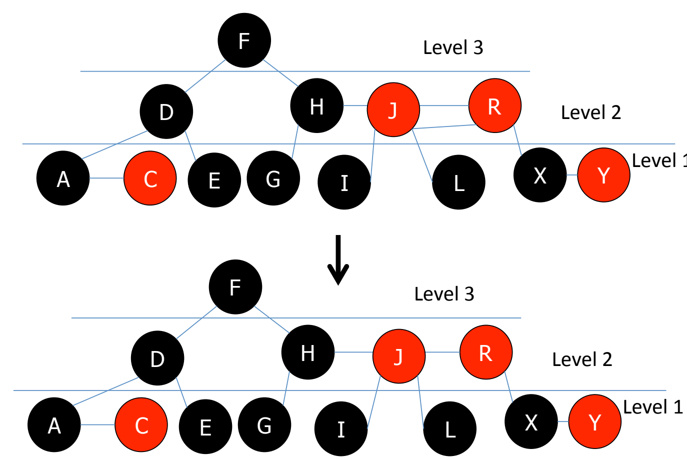
Insert L



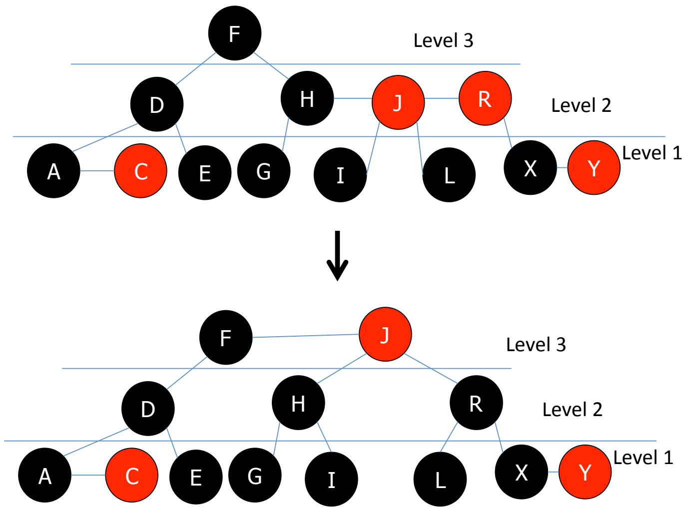
Insert L



Insert L



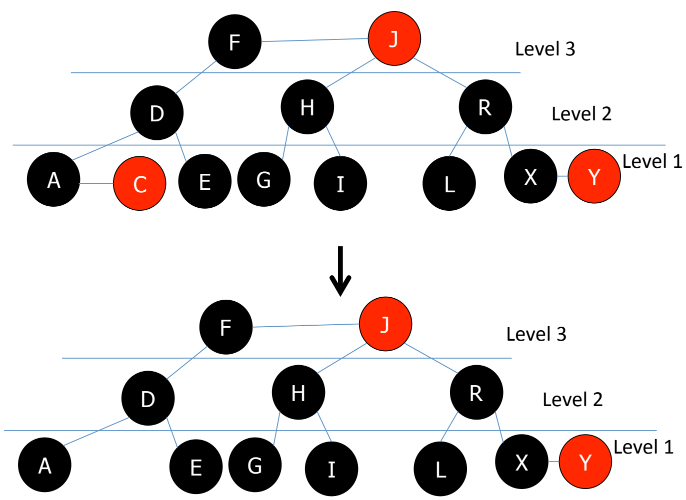
Insert L



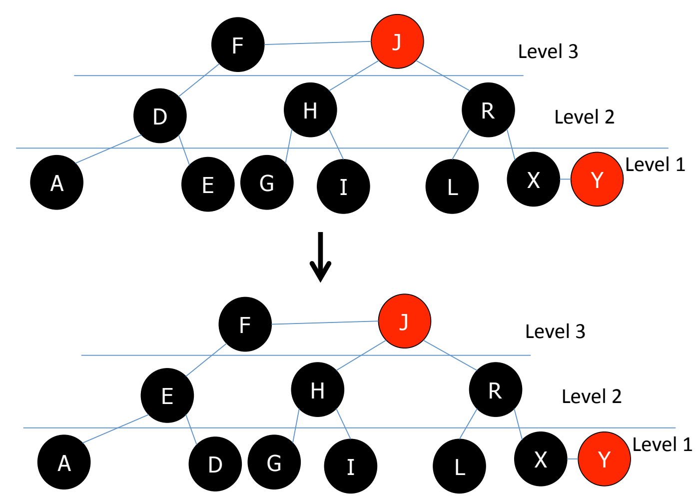
AA Trees—Deleting a Node: A little Trickier

- Decrease the level (if needed)
  - Find the node to delete; delete it.
  - the deleted node's parent is *not* on level 1 and now it only has one child. So, swap the deleted node's parent with it's remaining child and make the new child red—this reduces the overall tree to have one fewer levels
- Call skew on (what was the root—since now there's multiple nodes at the top level)
- Call split on the new root that was created by skew

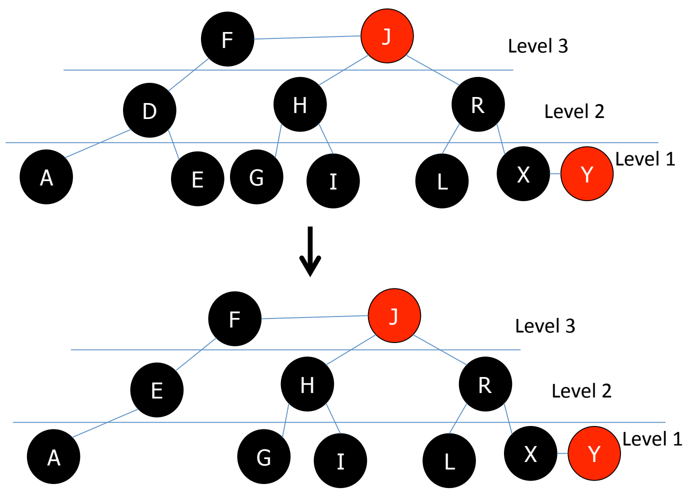
Delete C



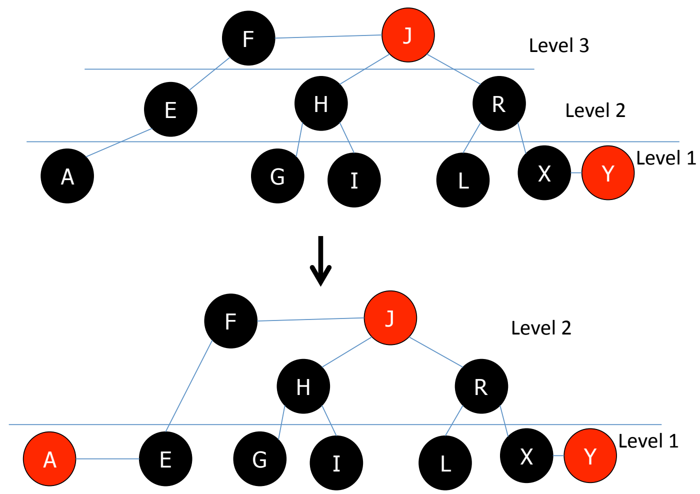
Delete D



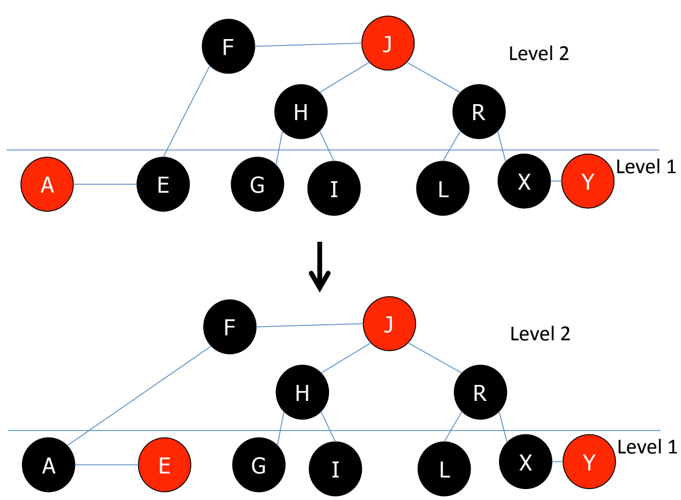
Delete D



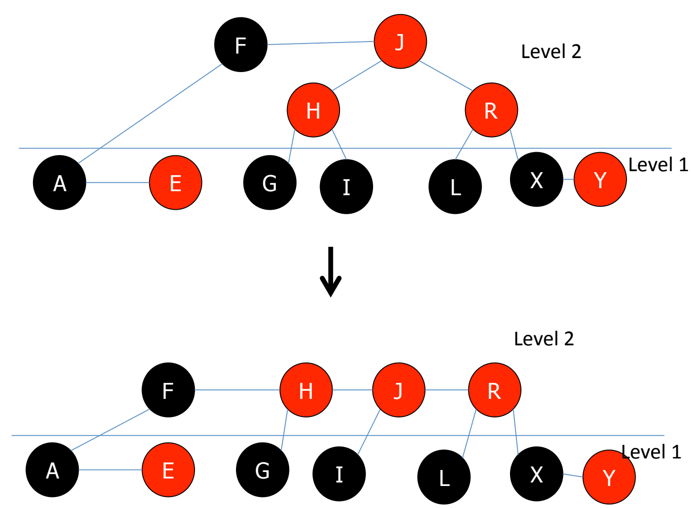
Delete D



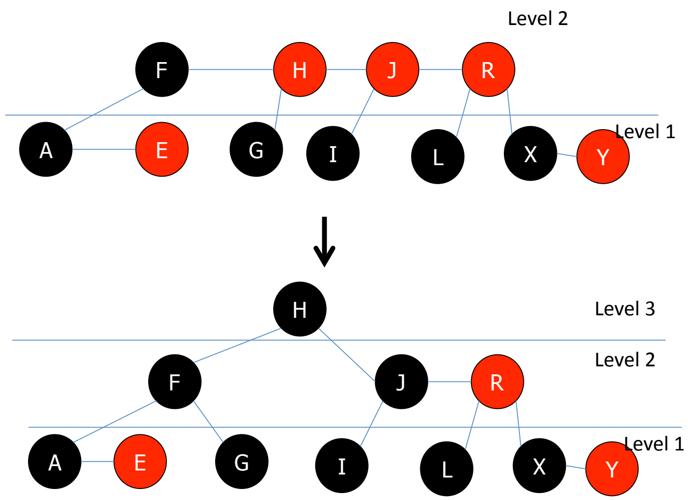
Delete D



Delete D



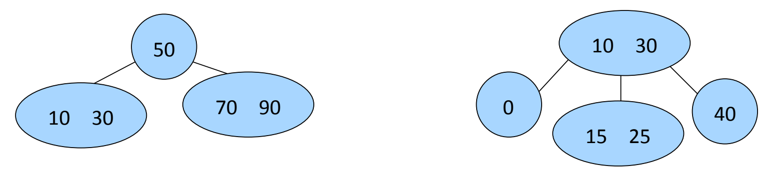
Delete D



2-3 Trees

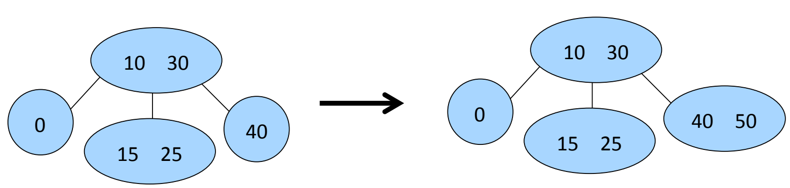
- Two types of nodes:

2 Node 3 Node



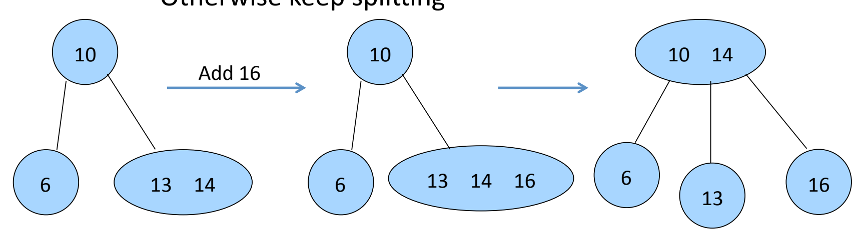
Insertion

- Possible Cases:
  - Inserting into 2-node leaf
    - Just insert number



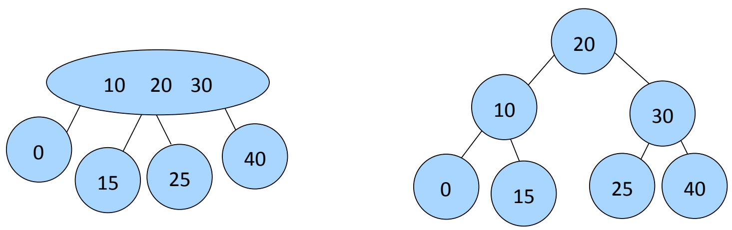
Insertion

- Possible Cases:
  - Inserting into 3-node leaf
    - Must split the 3-node
    - Promote middle
    - If promotion changes 2-node to a 3-node you're done
    - Otherwise keep splitting

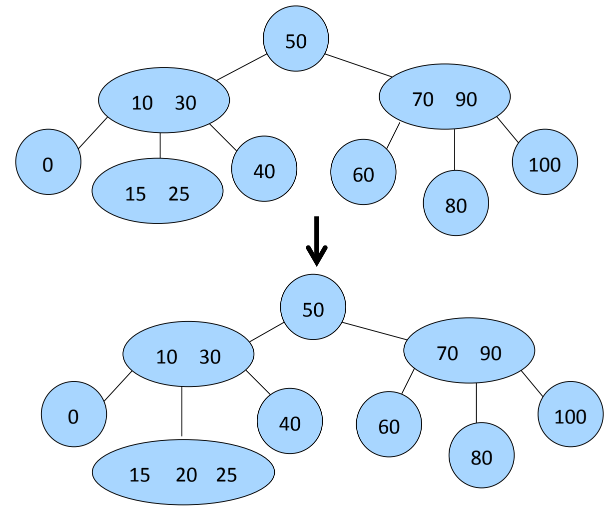


Insertion

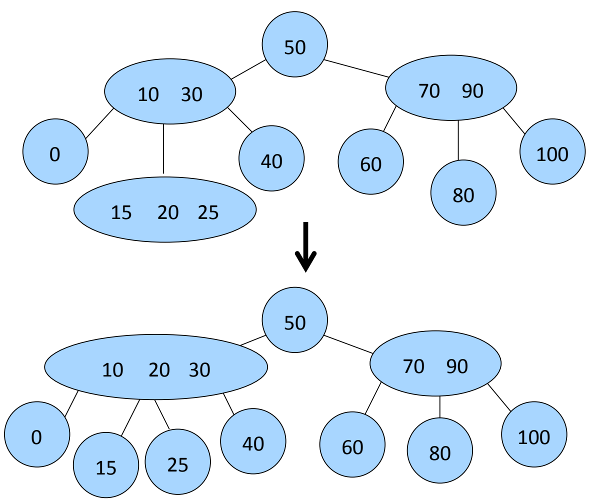
- Splitting with children
  - Left sibling adopts left-most children
  - Right sibling adopts right-most children



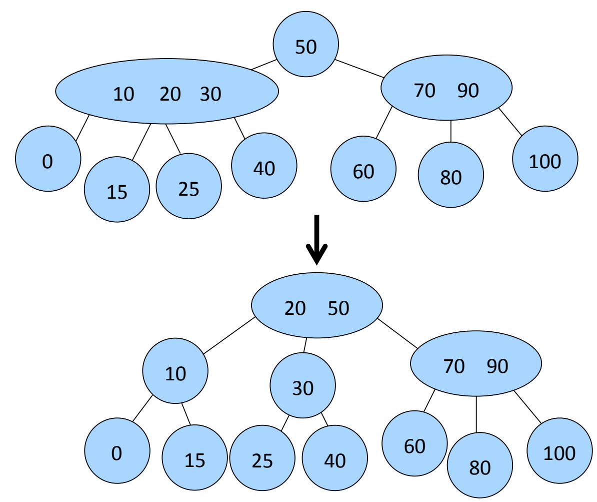
Insert 20



Insert 20



Insert 20

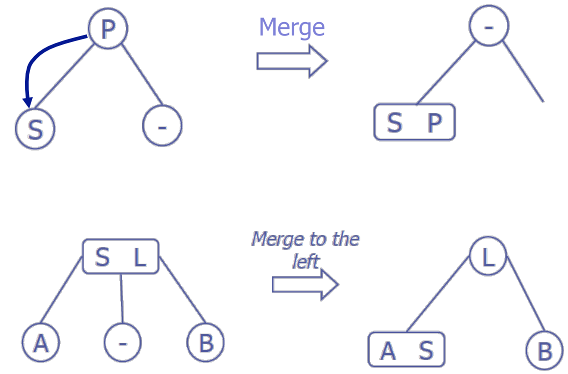


Deletion

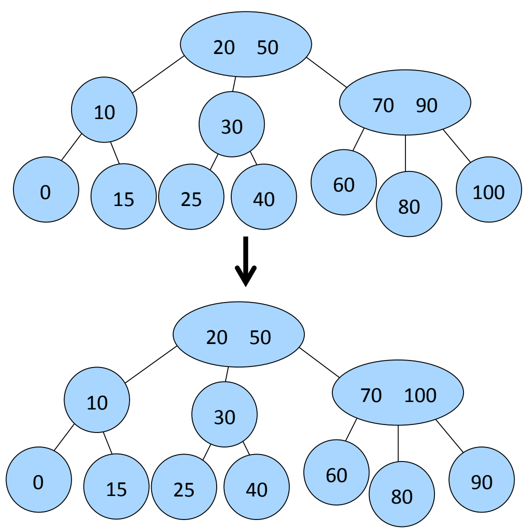
1) If node to delete is not a leaf, swap with in order successor. Now you are deleting from a leaf
2) If you're deleting from a 3-node, delete and you're done
3) If you're deleting from a 2-node, need to either merge or rotate

Deletion - Merge

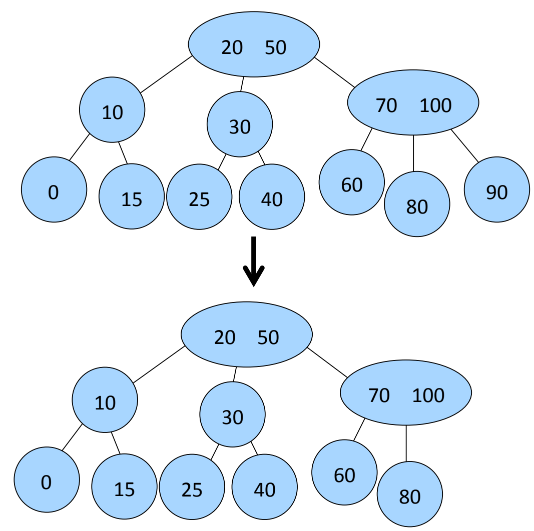
- Merge if sibling is a 2-node



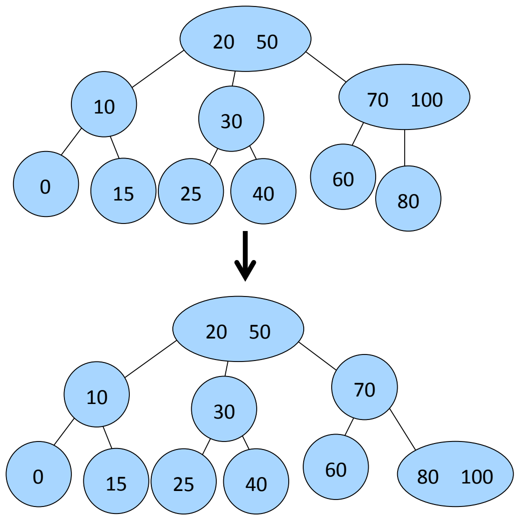
Delete 90



Delete 90

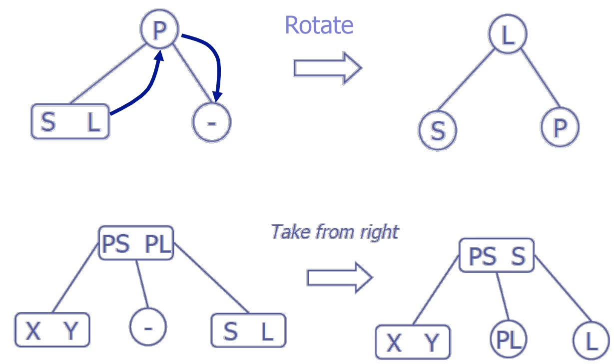


Delete 90

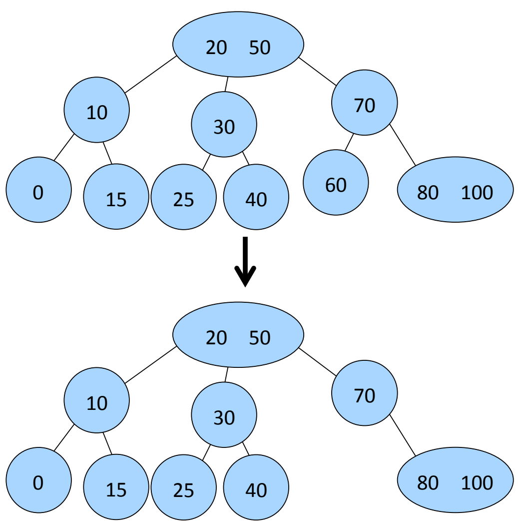


Deletion - Rotate

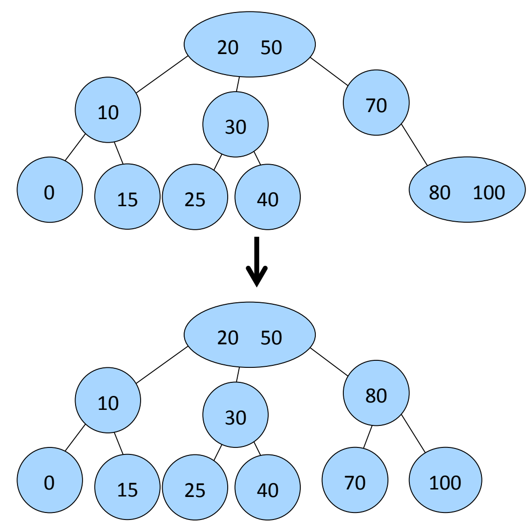
- Rotate if sibling is a 3-node



Delete 60



Delete 60



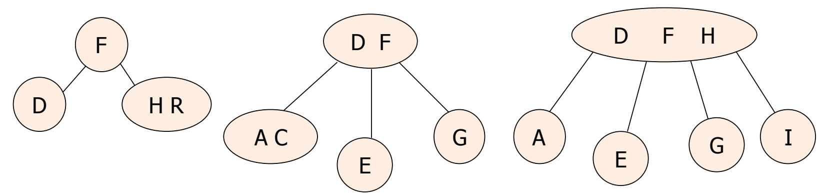
Deletion

- What if one sibling is a 3-node and one is a 2-node
  - Take your pick you can either merge with the 2-node or rotate with the 3-node
  - Whatever is easiest for you

2-3-4 Trees

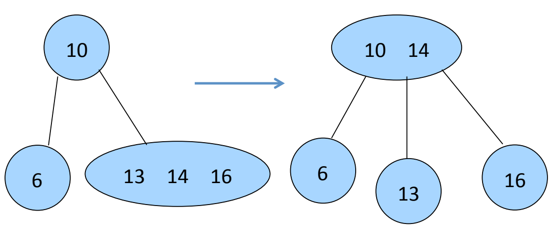
- 3 types of nodes

2 node 3 node 4 node

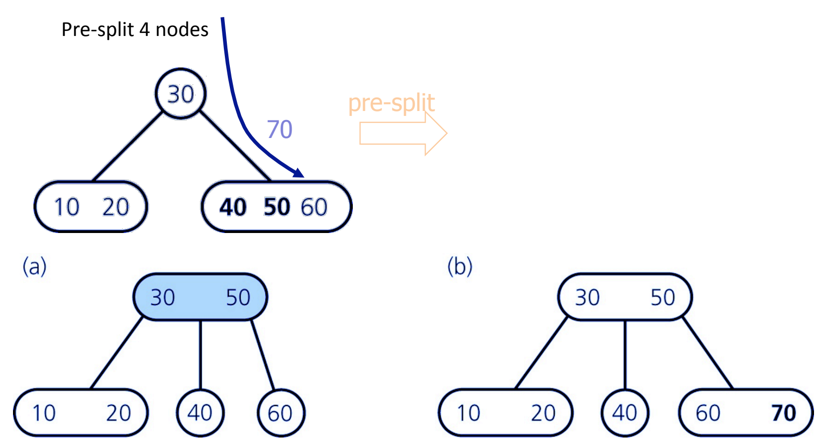


Insertion

- Traverse through the tree and insert just like a binary tree
- Every 4-node you pass: pre-split
- Pre-split is same as splitting a 4-node in a 2-3 tree



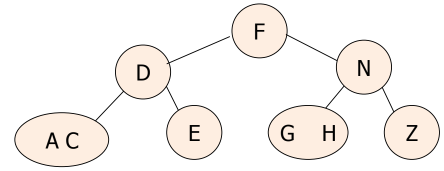
Insertion



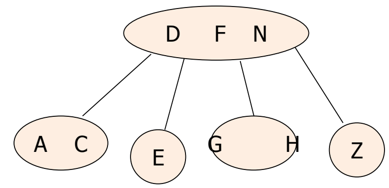
Deletion

- Preemptively turn 2-nodes in 3 and 4-nodes
  - This way deletion can be done in one pass
  - **Rotate** if sibling is not a 2-node
  - **Merge** if sibling is a 2-node

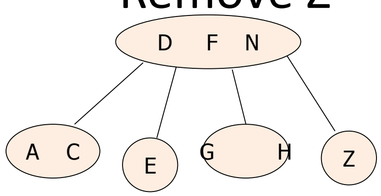
Remove Z



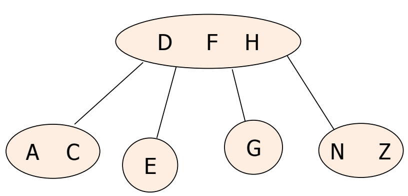
N is a 2 node. We must fix it.
Sibling is a 2 node so Merge



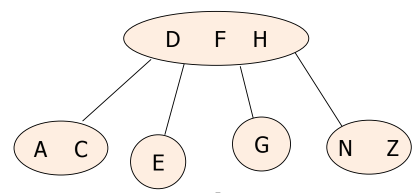
Remove Z

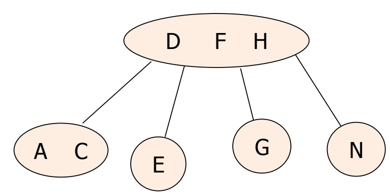


Z is a 2 node. We must fix it.
Sibling is a 3 node so Rotate



Remove Z





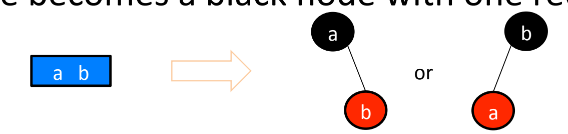
Red-Black Trees

- Converts 2-4 trees into binary trees
- Red-Black Trees are BSTs where every node is colored red or black

Converting from 2-4 to red-black

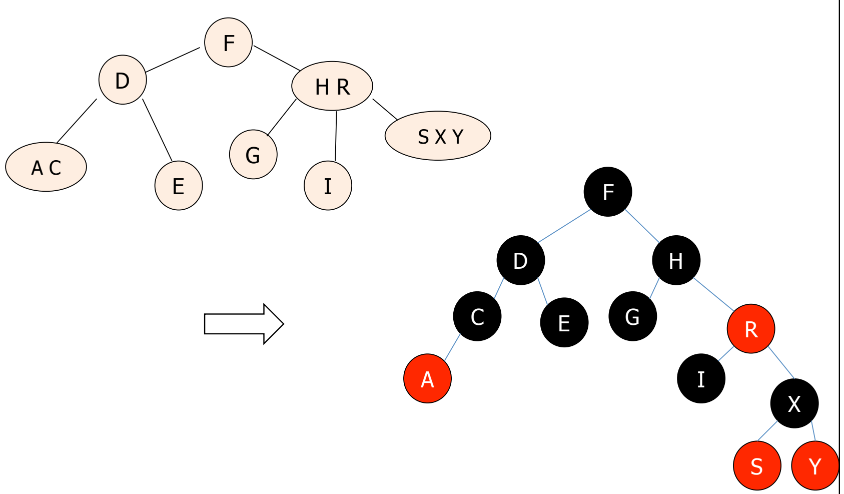
- 2 Node becomes a black node

- 3 Node becomes a black node with one red child



- 4 Node becomes a black node with 2 red children

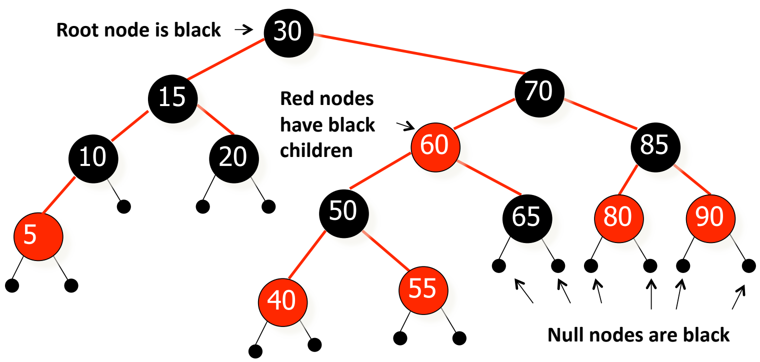
Converting from 2-4 to red-black



Red-Black Properties

- Every node is either red or black
- The root is black
- External Nodes (nulls) are black
- If a node is red, both children are black
- Every path from a node to a null has the same number of black nodes (the black height)

Example Red-Black Tree



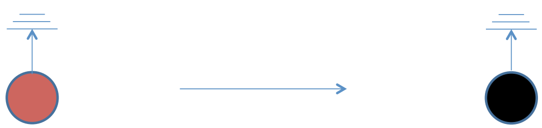
- Every node is either red or black
- Each path from root to null have the same number of black nodes.

Red-Black Tree Insertion

- Insert like a regular binary search tree.
- Inserted node is always red.
- Then fix the tree using 4 steps (really just 3).

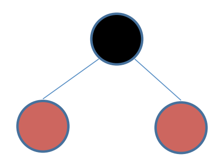
Red-Black Trees Insertion

- Step 1
  - If the root is red, color it black.



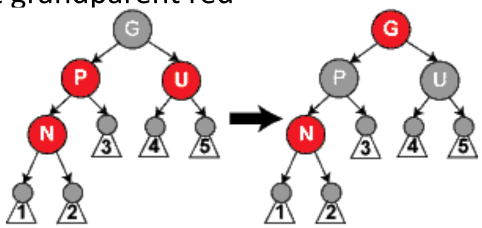
Red-Black Trees Insertion

- Step 2 (not really a step)
  - If the parent is black, then you are done.



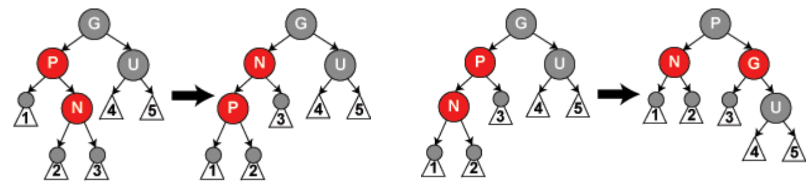
Red-Black Trees Insertion

- Step 3
  - If parent is red, and uncle is red:
  - Paint parent and uncle black
  - Paint grandparent red



Red-Black Trees Insertion

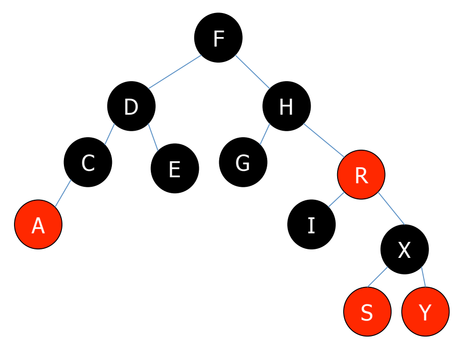
- Step 4
  - If parent is red, and uncle is black:
  - Rotate on parent (if necessary)
  - Rotate on grandparent, paint gp red, parent black



Removal

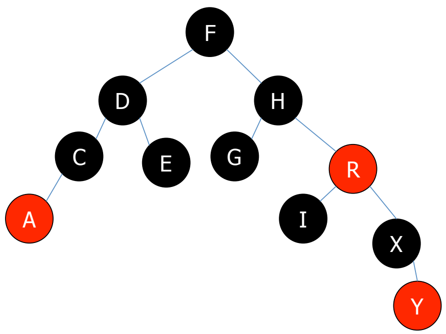
- If removing red leaf, just remove and you're done
- If it is a single child parent, must be black. Delete, and recolor it's child (which must be red) black.
- If the node has two children, swap node with in order successor
  - If in-order successor is red, remove it and you're done
  - If in-order is a single child parent, apply previous rule

Example



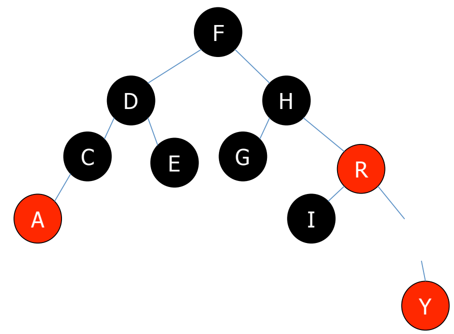
- Remove S

Example



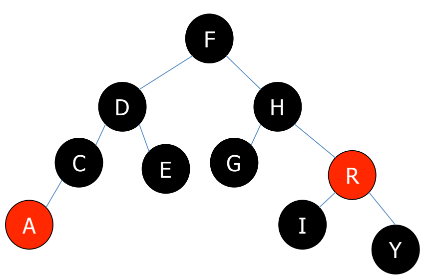
- Remove X

Example



- Remove X
- Delete it

Example

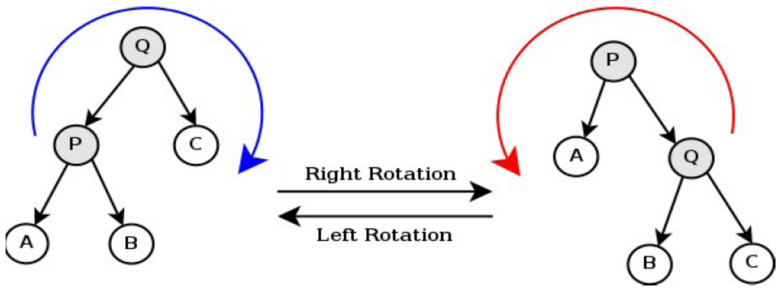


- Remove X
- Delete it
- Recolor child black

AVL Trees

- Balanced Trees
- Have a height constraint. For each node, the difference in height of the left and right subtree must be -1, 0, or 1

Rotations



AVL Rotation: The Breakdown

Note: “P” is the root

Left Rotation:

```
-Let Q be P's right child.
-Set Q to be the new root.
-Set P's right child to be Q's left child.
-Set Q's left child to be P.
```

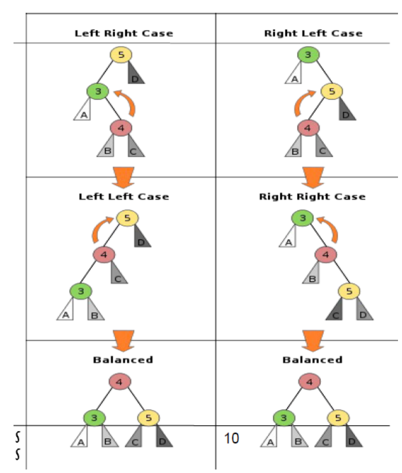
Right Rotation:

```
-Let P be Q's left child.
-Set P to be the new root.
-Set Q's left child to be P's right child.
-Set P's right child to be Q.
```

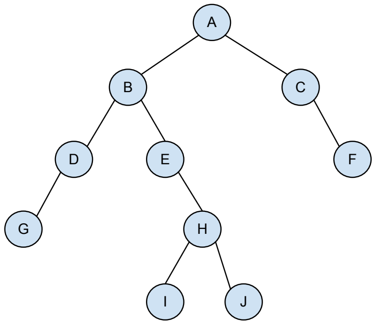
AVL Rotation: THE CHEATSHEET

Case (what this case is called)	When to Use It	What to do
Right-Right	-Right subtree outweighs left subtree -Balance factor of root’s right is -1	1) Left rotation on root
Right-Left	-Right subtree outweighs left subtree -Balance factor of root’s right is +1	1) Right rotation on right 2) Left rotation on root
Left-Left	-Left subtree outweighs the right subtree -Balance factor Of root’s left is +1	1) Right rotation on root
Left-Right	-Left subtree outweighs the right subtree -Balance factor of root’s left is -1	1) Left rotation on left 2) Right rotation on root

Rotations

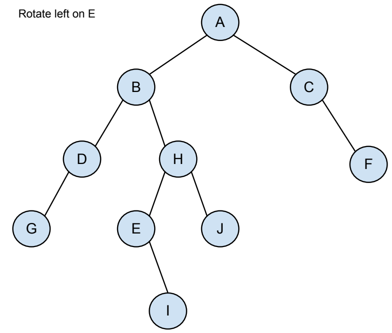


Practice AVL Tree



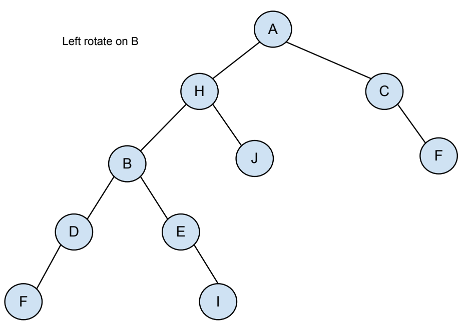
Practice AVL Tree

Rotate left on E



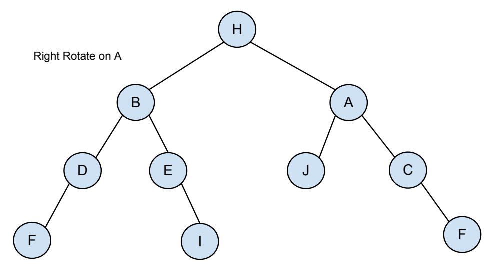
Practice AVL Tree

Left rotate on B



Practice AVL Tree

Right Rotate on A



Practice AVL Tree

- Inserting and deleting are the same as in binary search trees.
- Use rotation to fix balance issues