eecs 281 DATA STRUCTURES AND ALGORITHMS

Lecture 12: AA Trees
Treaps

AA-Trees

The implementation and number of rotation cases in Red-Black Trees is complex

AA-trees:

- fewer rotation cases so easier to code, especially deletions (eliminates about half of the rotation cases)
- named after its inventor Arne Andersson (1993), an optimization over original definition of Binary B-trees (BB-trees) by Bayer (1971)

AA-trees still have $O(\log n)$ searches in the worst-case, although they are slightly less efficient empirically

Demo: http://www.cis.ksu.edu/~howell/viewer/viewer.html

[McCollam]

AA-Tree Ordering Properties

An AA-Tree is a binary search tree with all the ordering properties of a red-black tree:

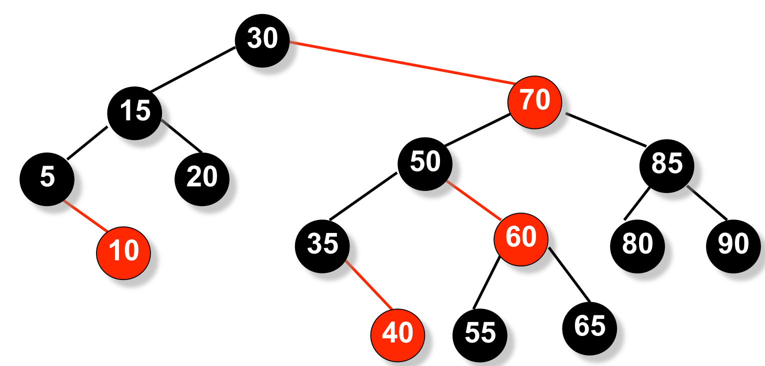
1. Every node is colored either red or black
2. The root is black
3. External nodes are black
4. If a node is red, its children must be black
5. All paths from any node to a descendent leaf must contain the same number of black nodes (black-height, not including the node itself)

PLUS

6. Left children may not be red

[McCollam]

An AA-Tree Example



No left red children!

Half of red-black tree rotation cases eliminated!

(Which M-way trees are AA-trees equivalent to?)

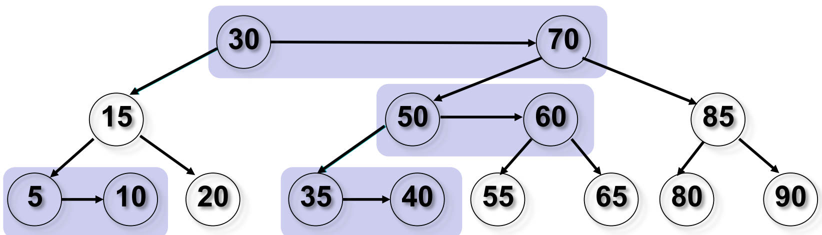
[McCollam]

Representation of Balancing Info

The level of a node (instead of color) is used as balancing info

- "red" nodes are simply nodes that located at the same level as their parents

For the tree on the previous slide:

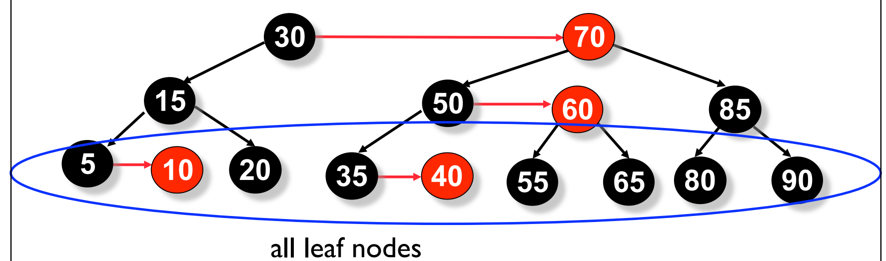


[McCollam]

Redefinition of "Leaf"

Both the terms leaf and level are redefined:

A leaf in an AA-tree is a node with no black internal-node as children

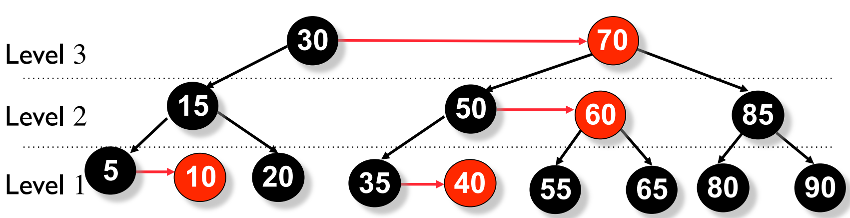


[McCollam]

Redefinition of "Level"

The level of a node in an AA-tree is:

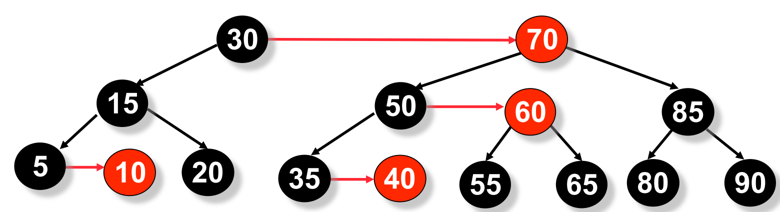
- leaf nodes are at level 1
- red nodes are at the level of their parent
- black nodes are at one less than the level of their parent
  - as in red-black trees, a black node corresponds to a level change in the corresponding 2-3 tree



[McCollam]

Implications of Ordering Properties

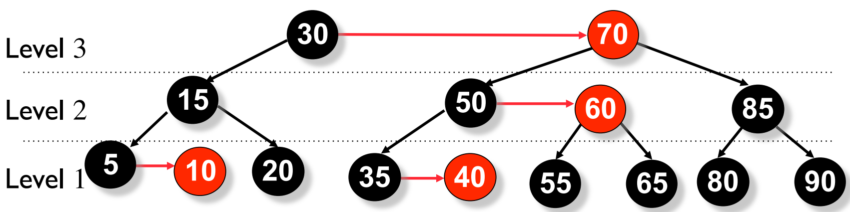
1. Horizontal links are right links
   - because only right children may be red
2. There may not be double horizontal links
   - because there cannot be double red nodes



[McCollam]

Implications of Ordering Properties

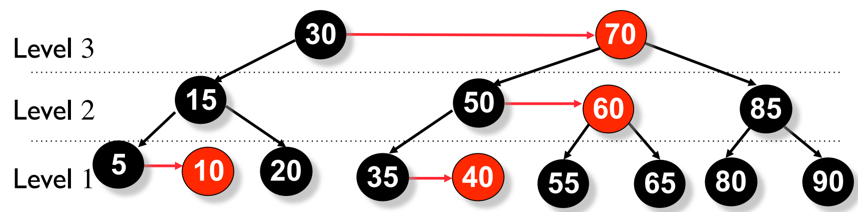
3. Nodes at level 2 or higher must have two children
4. If a node does not have a right horizontal link, its two children are at the same level



[McCollam]

Implications of Ordering Properties

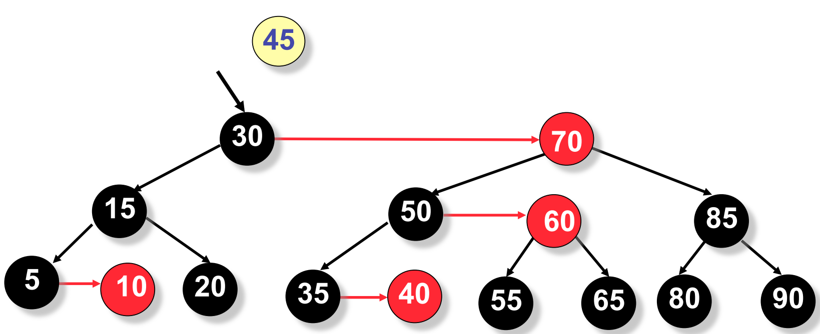
5. Any simple path from a black node to a leaf contains one black node on each level



[McCollam]

Example: Insert 45

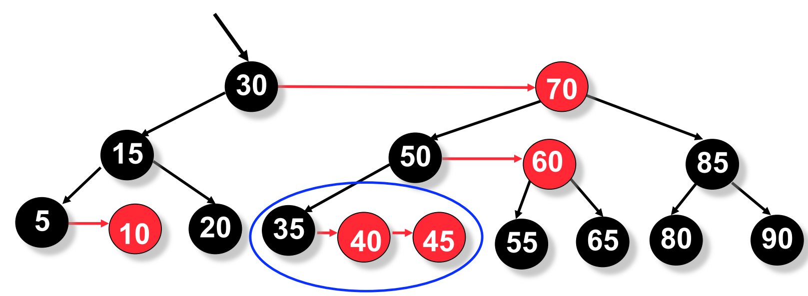
First, insert as for simple binary search tree
Newly inserted node is red



[McCollam]

Example: Insert 45

After insert to right of 40:
Problem: double right horizontal links starting at 35, need to split



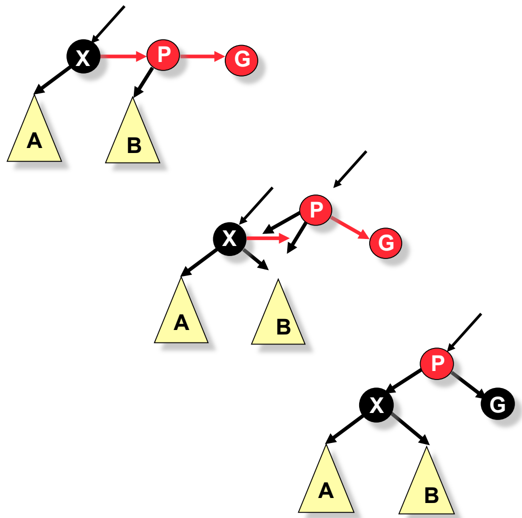
[McCollam]

Split: Removing Double Reds

Problem: With G inserted, there are two reds in a row

Split is a simple left rotation between X and P

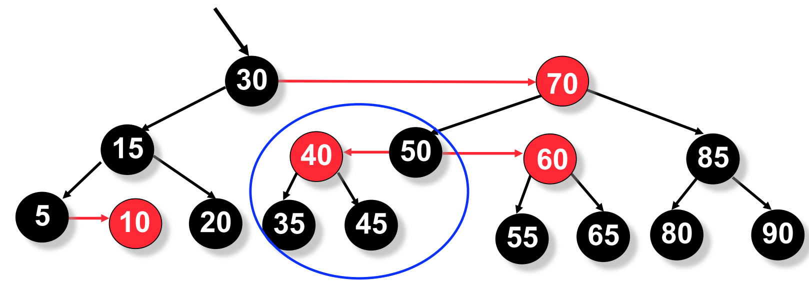
P's level increases in the AA-tree



[McCollam]

Example: Insert 45

After split at 35:
Problem: left horizontal link at 50 is introduced, need to skew



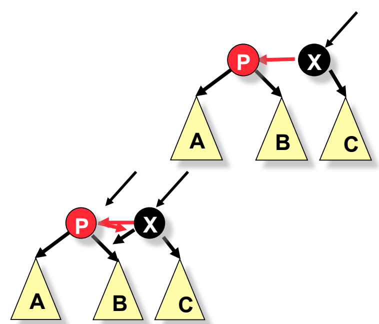
[McCollam]

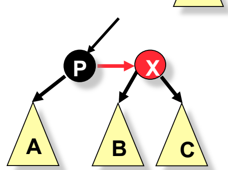
Skew: Removing Left Horizontal Link

Problem: left horizontal link in AA-tree

Skew is a simple right rotation between X and P

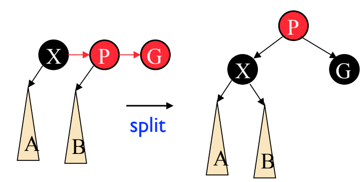
P remains at the same level as X



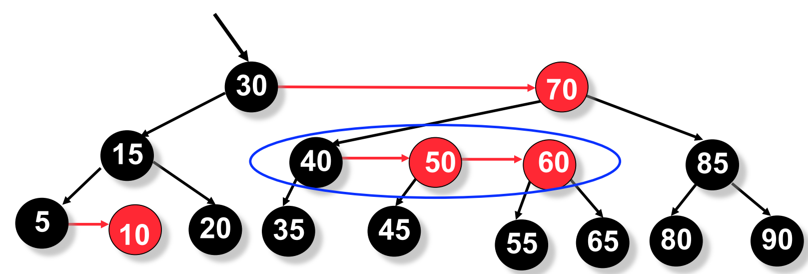


[McCollam]

Example: Insert 45

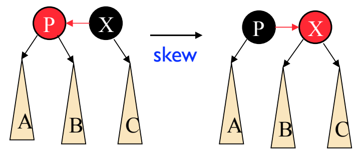


After skew at 50:
Problem: double right horizontal links starting at 40, need to split



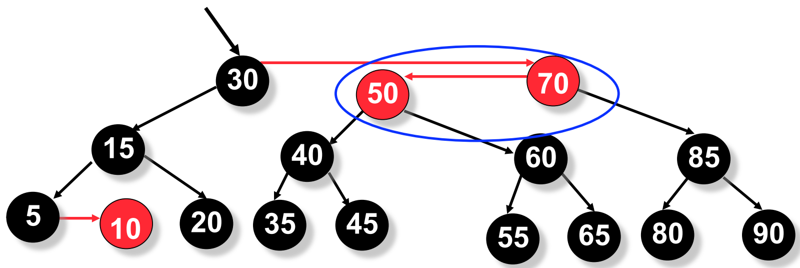
[McCollam]

Example: Insert 45



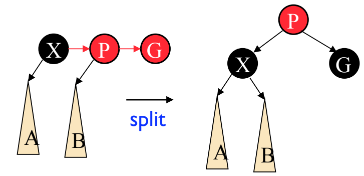
After split at 40:

Problem: left horizontal link at 70 introduced (50 is now on same level as 70), need to skew



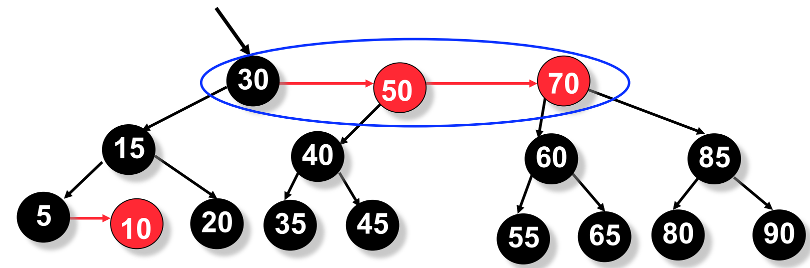
[McCollam]

Example: Insert 45



After skew at 70:

Problem: double right horizontal links starting at 30, need to split

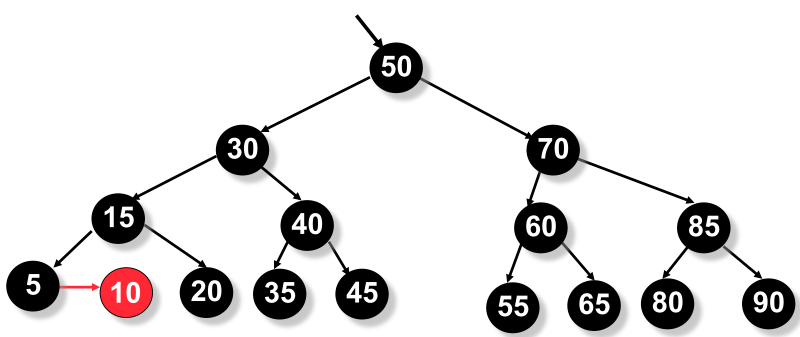


[McCollam]

Example: Insert 45

After split at 30:

Insertion is complete (finally!)



[McCollam]

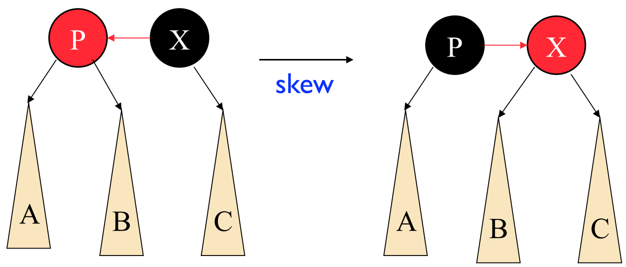
AATree::Insert()

```
void AATree::
insert(Link &root, Node &add) {
  if (root == NULL)  // have found where to insert y
    root = add;
  else if (add→key < root→key) // <= if duplicate ok
    insert(root→left, add);
  else if (add→key > root→key)
    insert(root→right, add);
  // else handle duplicate if not ok

  skew(root);    // do skew and split at each level
  split(root);
}
```

[McCollam]

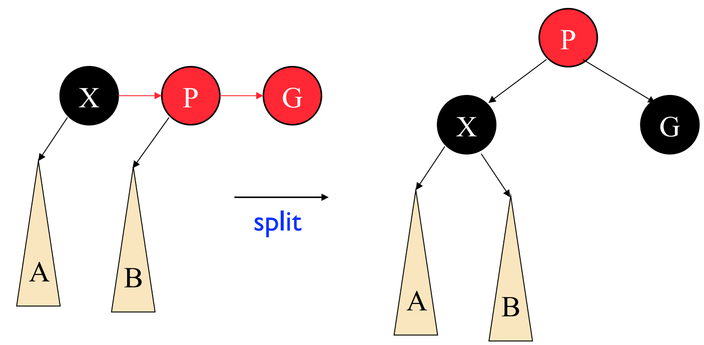
Skew: Remove Left Horizontal Link




```
void AATree::skew(Link &root) { // root = X
  if (root→left→level == root→level)
    rotate_right(root);
}
```

[Lee,Andersson]

Split: Remove Double Reds




```
void AATree::split(Link &root) { // root = X
  if (root→right→right→level == root→level)
    rotate_left(root);
}
```

[Lee,Andersson]

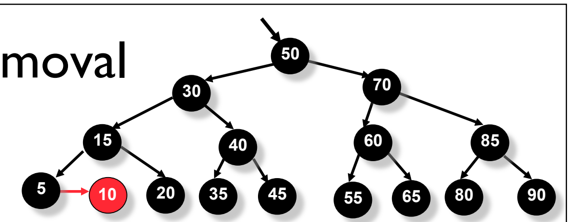
More on Skew and Split

Skew may cause double reds
- first we apply skew, then we do split if necessary

After a split, the middle node increases a level, which may create a problem for the original parent
- parent may need to skew and split

[Lee]

AA-Tree Removal



Rules:

1. if node to be deleted is a red leaf, e.g., 10, remove leaf, done
2. if it is parent to a single internal node, e.g., 5, it must be black; replace with its child (must be red) and recolor child black
3. if it has two internal-node children, swap node to be deleted with its in-order successor
   - if in-order successor is red (must be a leaf), remove leaf, done
   - if in-order successor is a single child parent, apply second rule

In both cases the resulting tree is a legit AA-tree (we haven't changed the number of black nodes in paths)

3. if in-order successor is a black leaf, or if the node to be deleted itself is a black leaf, things get complicated . . .

Black Leaf Removal

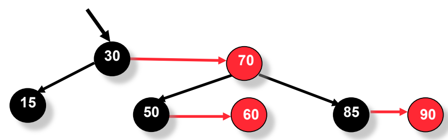
Follow the path from the removed node to the root

At each node p with 2 internal-node children do:

- if either of p's children is two levels below p
  - decrease the level of p by one
  - if p's right child was a red node, decrease its level also
- skew(p); skew($p\rightarrow$right); skew($p\rightarrow$right$\rightarrow$right);
- split(p); split($p\rightarrow$right);

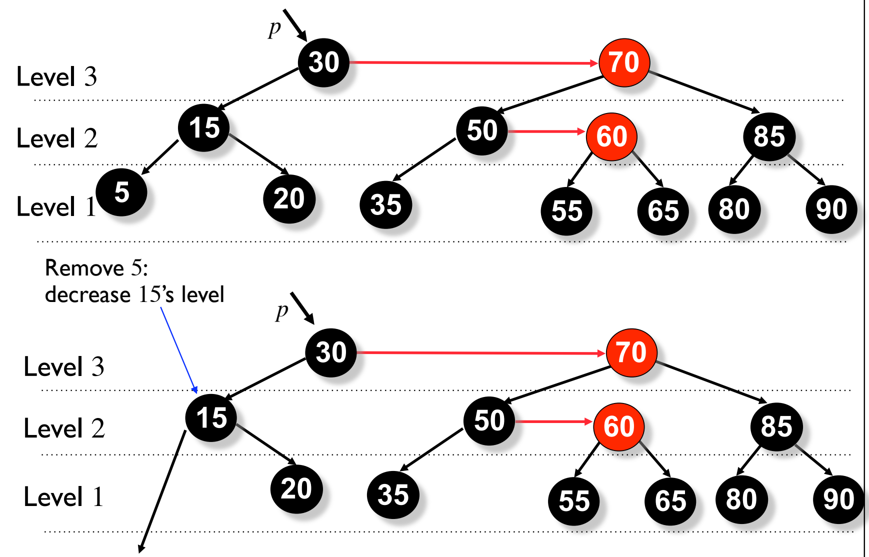
In the worst case, deleting one leaf node, e.g., 15, could cause six nodes to all be at one level, connected by horizontal right links

- but the worst case can be resolved by 3 calls to `skew()`, followed by 2 calls to `split()`!



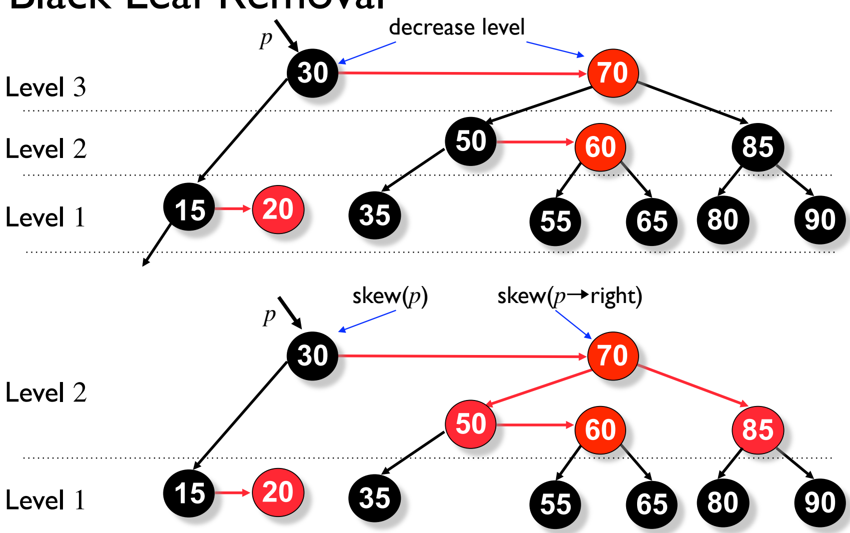
[Andersson,McCollam]

Black Leaf Removal



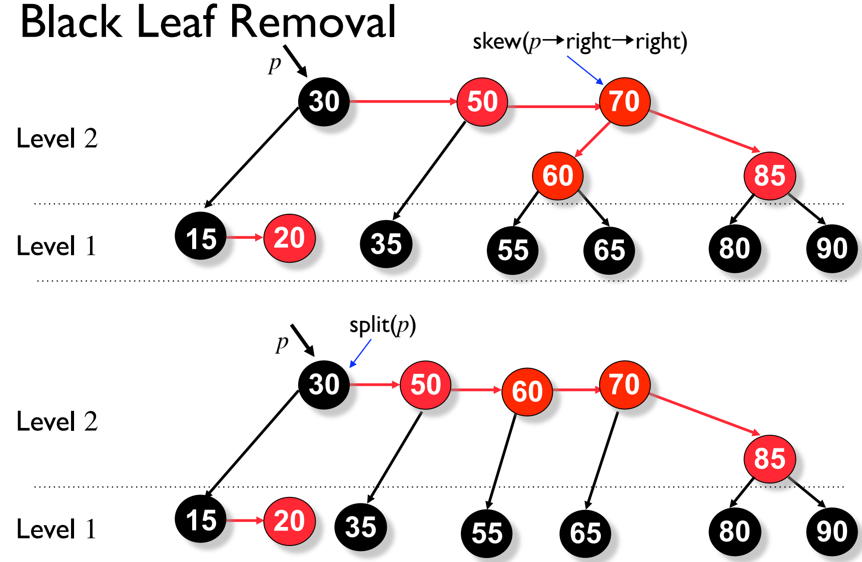
[Andersson,McCollam]

Black Leaf Removal



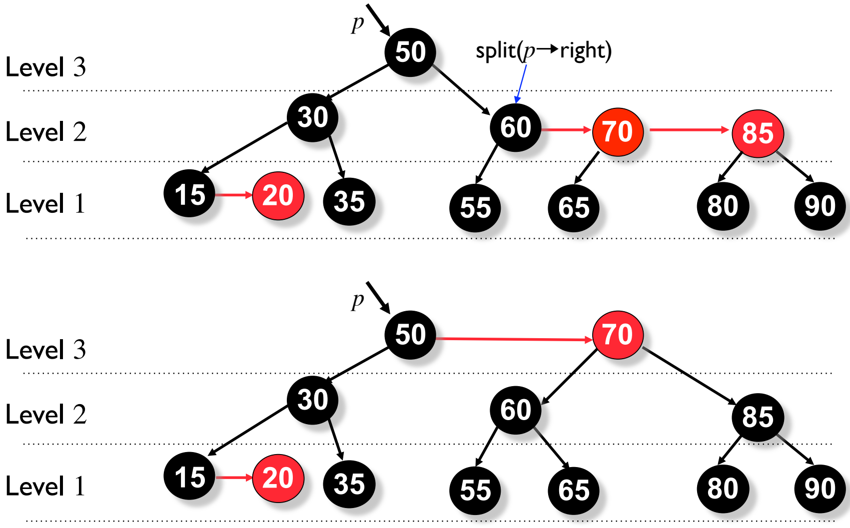
[Andersson,McCollam]

Black Leaf Removal



[Andersson,McCollam]

Black Leaf Removal



[Andersson,McCollam]

AA-Tree Implementation

```
procedure Skew (var t: Tree);
var temp: Tree;
begin
  if t↑.left↑.level = t↑.level then
  begin { rotate right }
    temp := t;
    t := t↑.left;
    temp↑.left := t↑.right;
    t↑.right := temp;
  end;
end;

procedure Split (var t: Tree);
var temp: Tree;
begin
  if t↑.right↑.right↑.level = t↑.level then
  begin { rotate left }
    temp := t;
    t := t↑.right;
    temp↑.right := t↑.left;
    t↑.left := temp;
    t↑.level := t↑.level + 1;
  end;
end;

procedure Insert (var x: data;
    var t: Tree; var ok: boolean);
begin
  if t = bottom then begin
    new (t);
    t↑.key := x;
    t↑.left := bottom;
    t↑.right := bottom;
    t↑.level := 1;
    ok := true;
  end else begin
    if x < t↑.key then
      Insert (x, t↑.left, ok)
    else if x > t↑.key then
      Insert (x, t↑.right, ok)
    else ok := false;
    Skew (t);
    Split (t);
  end;
end;

procedure Delete (var x: data;
    var t: Tree; var ok: boolean);
begin
  ok := false;
  if t <> bottom then begin

{ 1: Search down the tree and }
{ set pointers last and deleted. }
    last := t;
    if x < t↑.key then
      Delete (x, t↑.left, ok)
    else begin
      deleted := t;
      Delete (x, t↑.right, ok);
    end;

{ 2: At the bottom of the tree we }
{ remove the element (if it is present). }
    if (t = last) and (deleted <> bottom)
    and (x = deleted↑.key) then
    begin
        deleted↑.key := t↑.key;
        deleted := bottom;
        t := t↑.right;
        dispose (last);
        ok := true;
    end

{ 3: On the way back, we rebalance. }
    else if (t↑.left↑.level < t↑.level-1)
    or (t↑.right↑.level < t↑.level-1) then
    begin
      t↑.level := t↑.level -1;
      if t↑.right↑.level > t↑.level then
        t↑.right↑.level := t↑.level;
      Skew (t);
      Skew (t↑.right);
      Skew (t↑.right↑.right);
      Split (t);
      Split (t↑.right);
    end;
  end;
end;
```

[Andersson]

Balanced BST Summary

AVL Trees: maintain balance factor by rotations

2-3 Trees: maintain perfect trees with variable node sizes using rotations

2-3-4 Trees: simpler operations than 2-3 trees due to pre-splitting and pre-merging nodes, wasteful in memory usage

Red-black Trees: binary representation of 2-3-4 trees, no wasted node space but complicated rules and lots of cases

AA-Trees: simpler operations than red-black trees, binary representation of 2-3 trees

Randomized Search Trees

Motivations:

- when items are inserted in order into a BST, worst-case performance becomes $O(n)$
- balanced search trees either waste space or requires complicated (empirically expensive) operations or both
- randomly permuting items to be inserted would ensure good performance of BST with high probability, but randomly permuting input is not always possible/practical, instead . . .

Randomized search trees balance the trees probabilistically instead of maintaining balance deterministically

Treaps

A treap is a binary tree that:

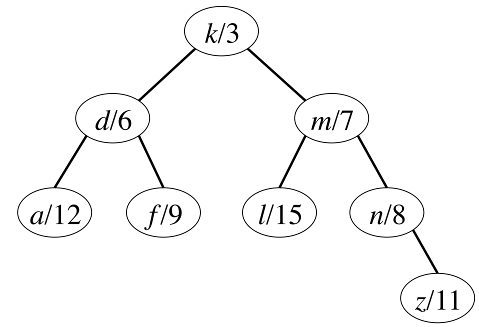
- has a key associated with each of its internal node:
  - the key in any node is > the keys in all nodes in its left subtree and < the keys in all nodes in its right subtree
  - i.e., internal nodes are arranged in in-order with respect to their keys
- and simultaneously has a priority associated with each of its internal node:
  - the priority of a parent is higher than those of its descendants
  - i.e., internal nodes are arranged in heap-order with respect to their priorities

A treap is a BST with heap-ordered priorities (but it is not a heap as it is not required to be a complete binary tree)

Example of a Treap

assuming min-heap
ordering of the priorities:

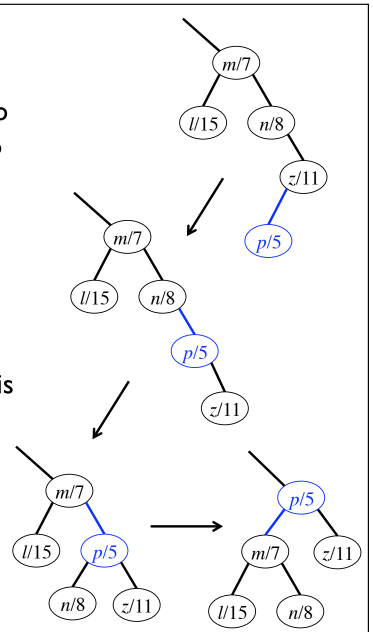
T:



Treaps: Insert

1. a new item to be inserted into a treap is given a random, unique priority (no duplicates)
2. the new item is then inserted into a treap as a leaf node, just like it would be under a standard BST
3. if its priority violates the heap-order property of the treap, the new node is rotated up until it is in the correct heap-order priority, using one or more single left- or right-rotation

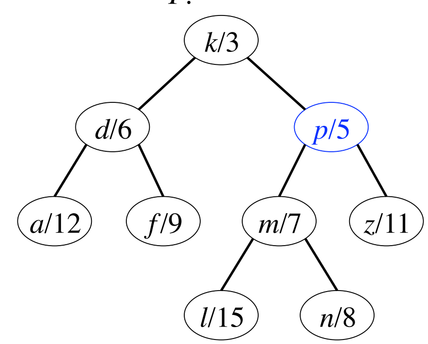
Example: insert $(p/5)$ into the example treap



Example Treap with $p/5$ Inserted

assuming min-heap
ordering of the priorities:

T:

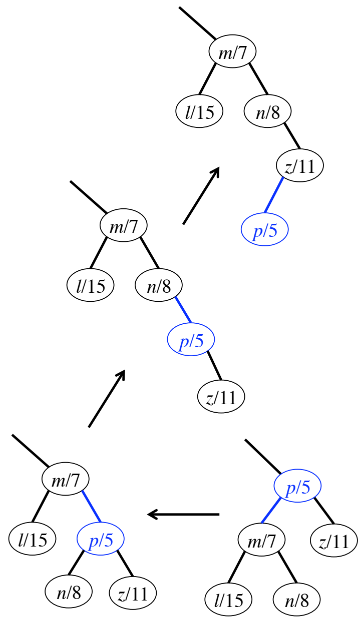


Treaps: Delete

Exact reverse of insert:

1. Rotate the node to be deleted such that its child with larger priority becomes the new parent
2. continue rotating until the node to be deleted is a leaf node
3. delete the leaf node

Example: delete $(p/5)$ from the example treap



Treaps: Search

Standard BST search

If it is desirable to keep frequently accessed items near the root, e.g., when the treap is used to maintain a cache, whenever an item is accessed, assign the item a new random number that gives it a higher priority and, if necessary, rotate its node up to maintain heap-order

If it is desirable for the treap of a set of keys to be unique, use one-way hash function on keys to generate priorities

Runtime Complexity

Various metrics to measure the complexity of an algorithm:

- asymptotic worst-case bound
- average-case bound
- amortized bound
- probabilistic expected-case bound

Treaps Running Time

The expected depth of any node is $O(\log n) \Rightarrow$ the expected running time of search, insert, delete (and tree split and join) are all $O(\log n)$

The expected number of rotations per insertion or deletion is less than $2 \Rightarrow$ fast implementation

Proof: relies on probabilistic analysis that is beyond the scope of this course . . .

Calls to random number generator usually incur non-trivial cost

Treap Exercise

Insert F, E, D, C, B, A with random priorities

- assuming min-heap ordering of the priorities