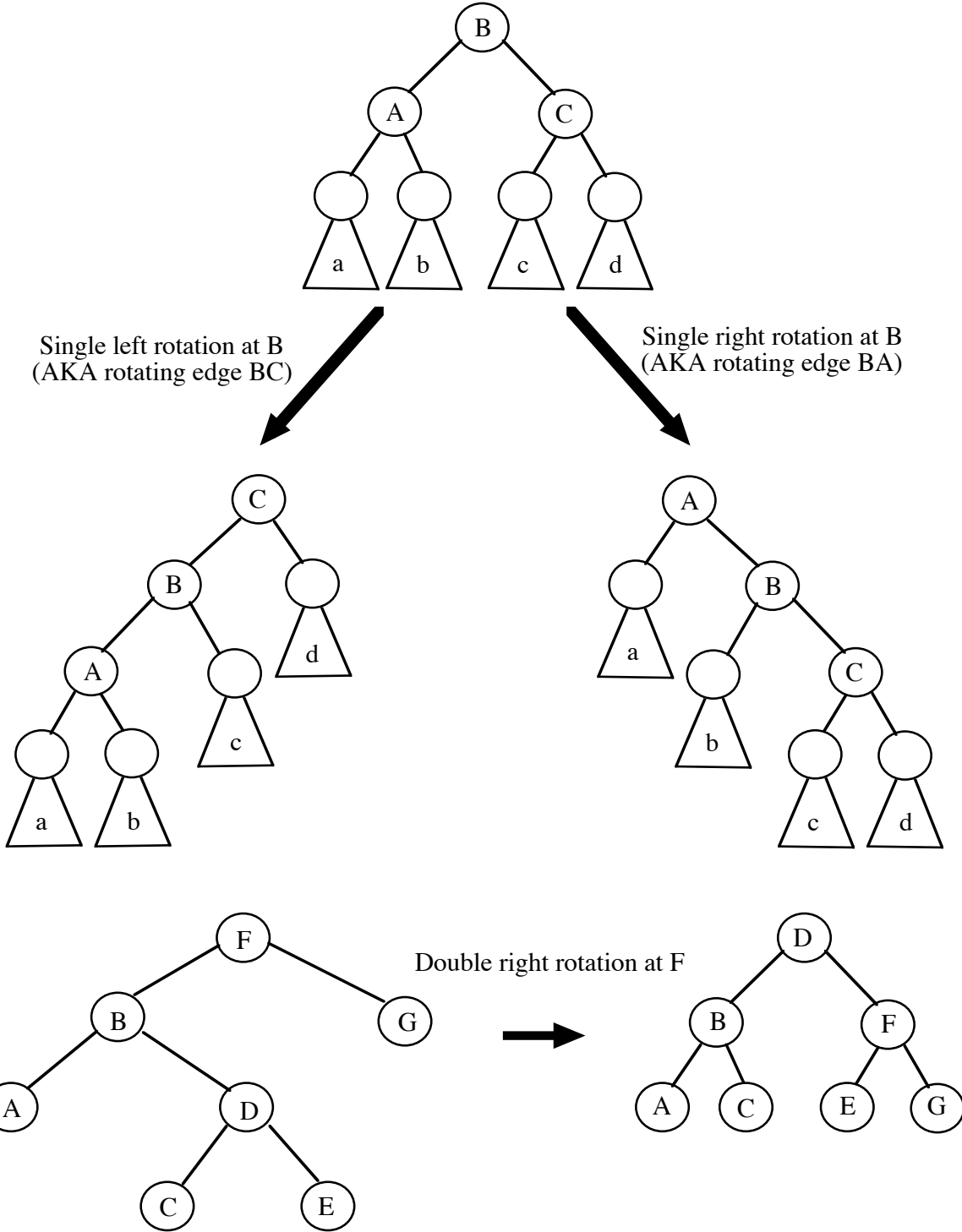


CSE 5311 Notes 2: Binary Search Trees

(Last updated 1/30/17 10:44 AM)

ROTATIONS



What two single rotations are equivalent?

(BOTTOM-UP) RED-BLACK TREES

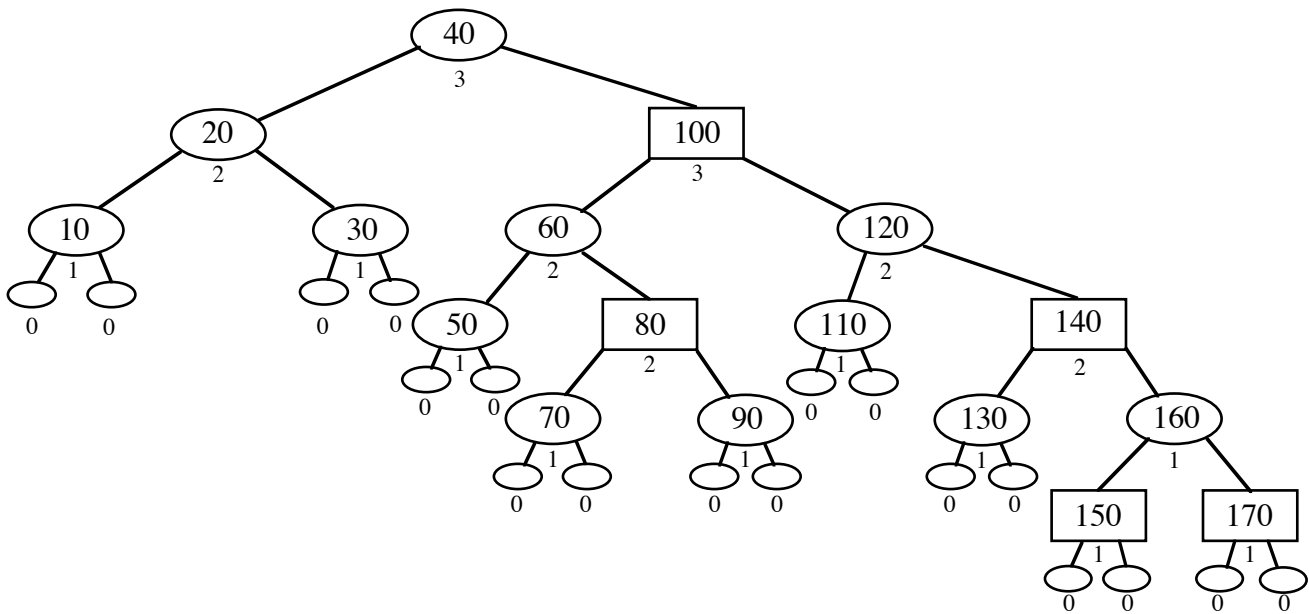
A *red-black tree* is a binary search tree whose height is logarithmic in the number of keys stored.

1. Every node is colored red or black. (Colors are only examined during insertion and deletion)
2. Every “leaf” (the sentinel) is colored black.
3. Both children of a red node are black.
4. Every simple path from a child of node X to a leaf has the same number of black nodes.

This number is known as the *black-height* of X ($bh(X)$).

Example:

□ = red ○ = black



Observations:

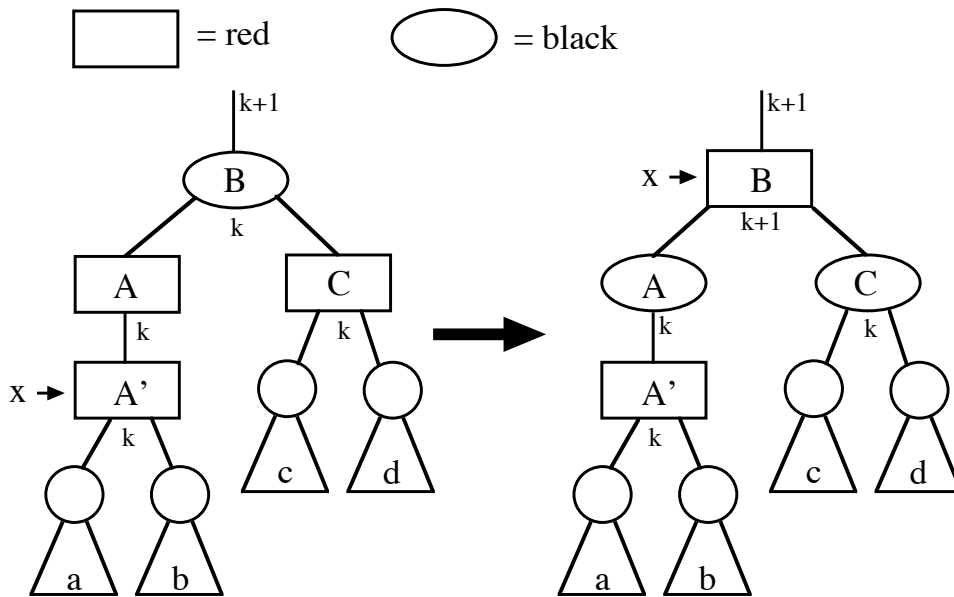
1. A red-black tree with n internal nodes (“keys”) has height at most $2 \lg(n+1)$.
2. If a node X is not a leaf and its sibling is a leaf, then X must be red.
3. There may be many ways to color a binary search tree to make it a red-black tree.
4. If the root is colored red, then it may be switched to black without violating structural properties.

INSERTION

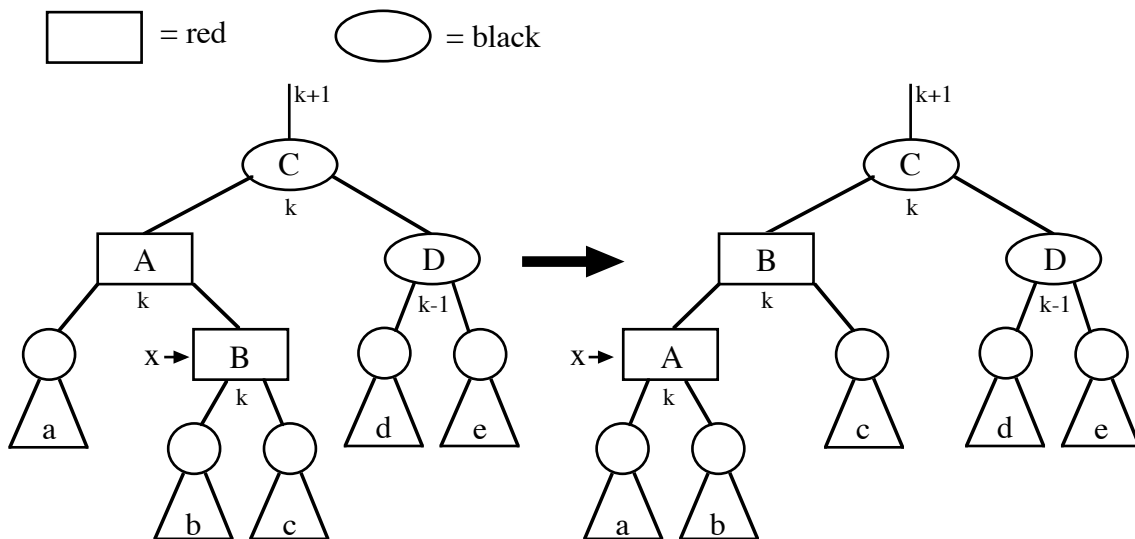
1. Start with unbalanced insert of a “data leaf” (both children are the sentinel).
2. Color of new node is _____.
3. May violate structural property 3. Leads to three cases, along with symmetric versions.

The x pointer points at a red node whose parent might also be red.

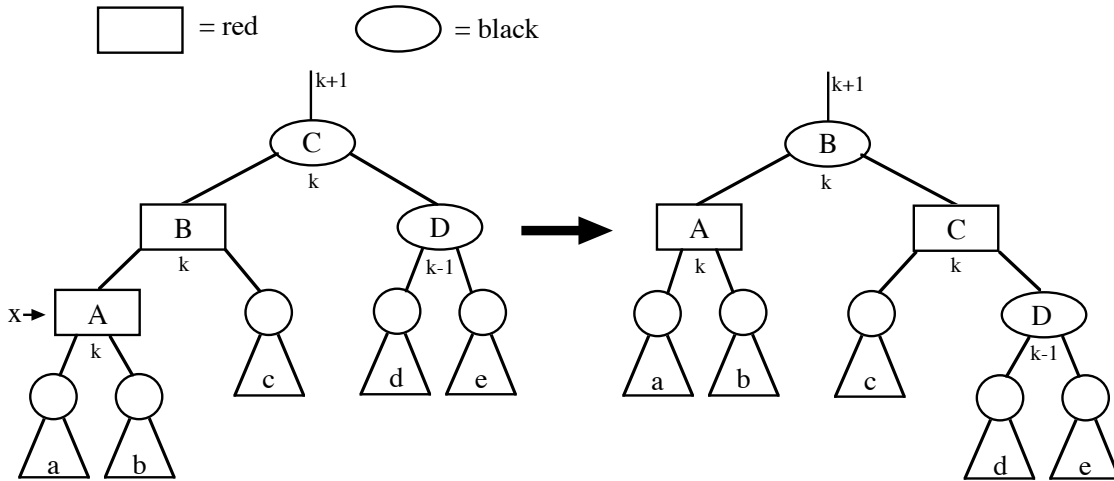
Case 1:



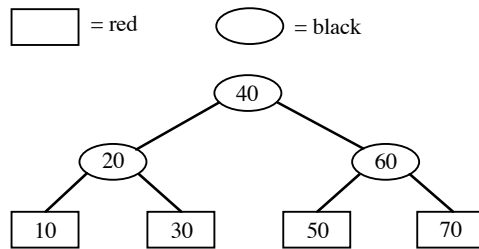
Case 2:



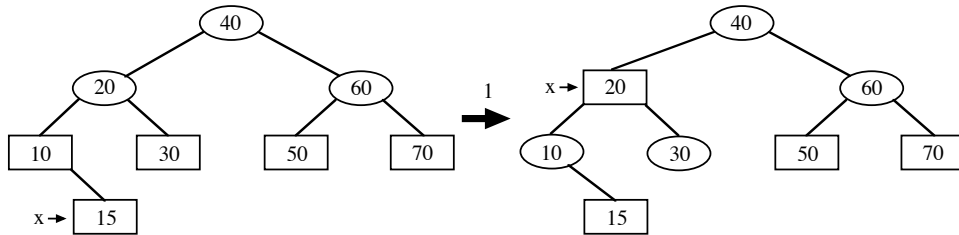
Case 3:



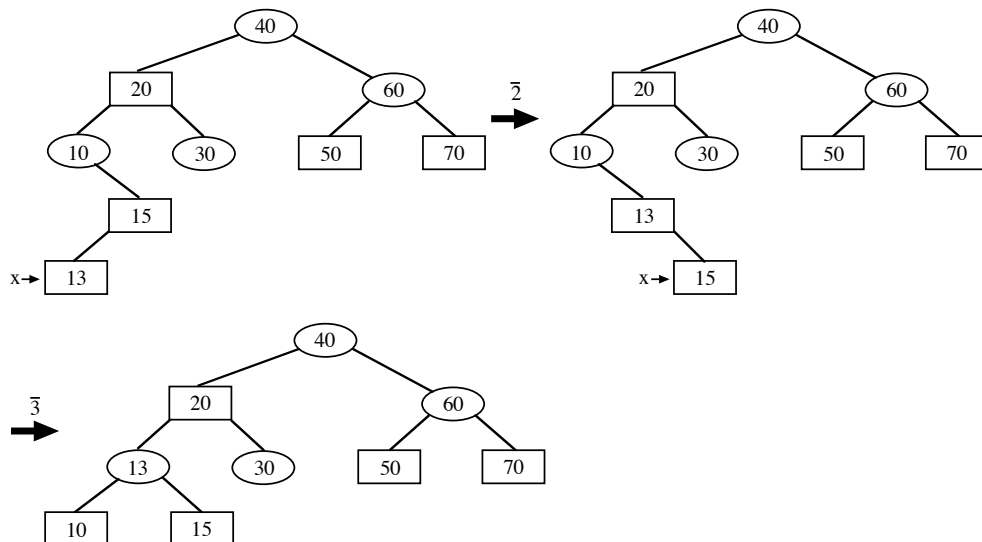
Example:



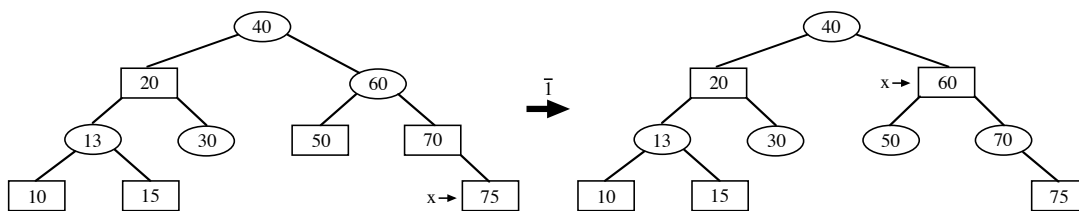
Insert 15



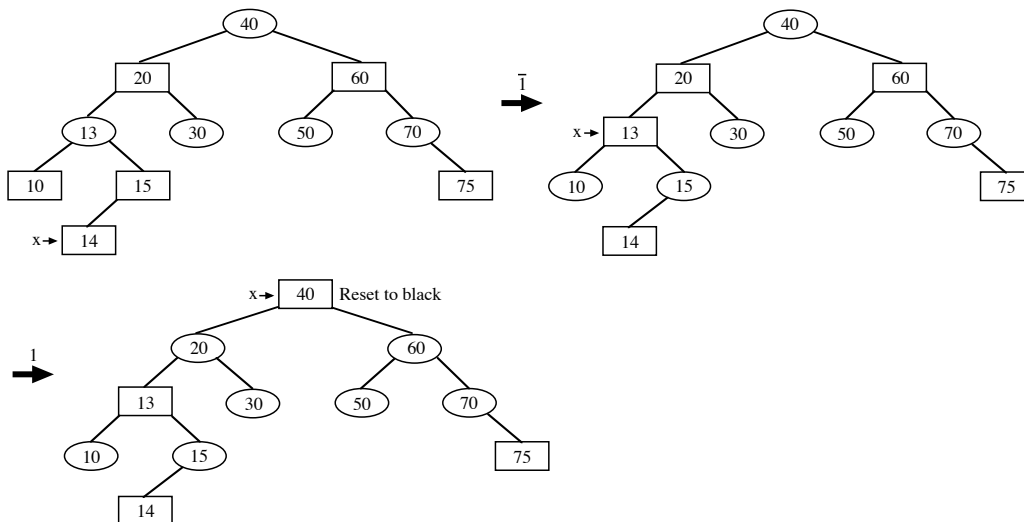
Insert 13



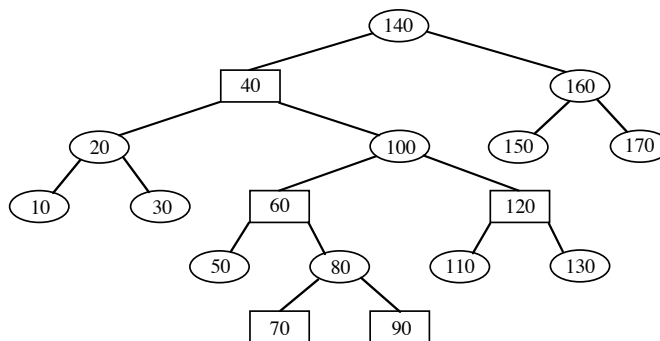
Insert 75



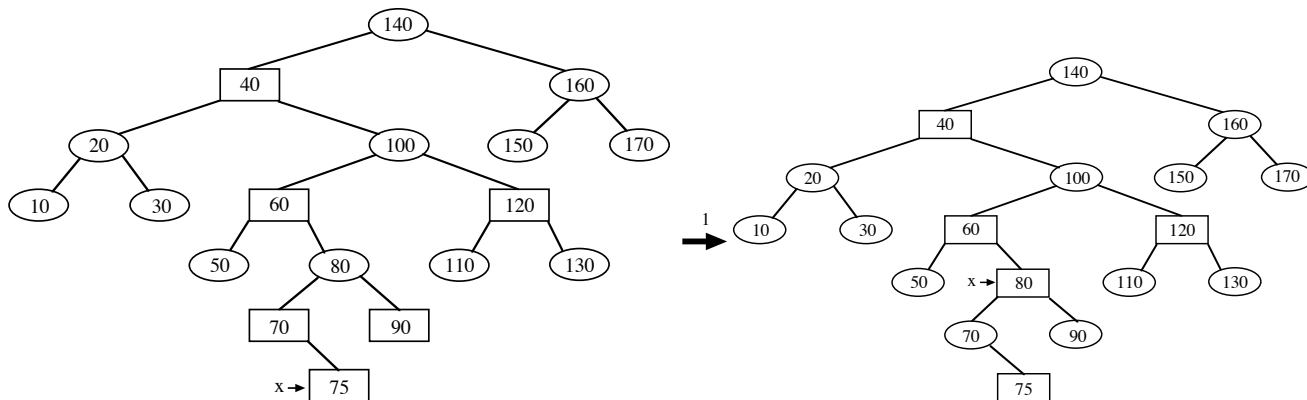
Insert 14

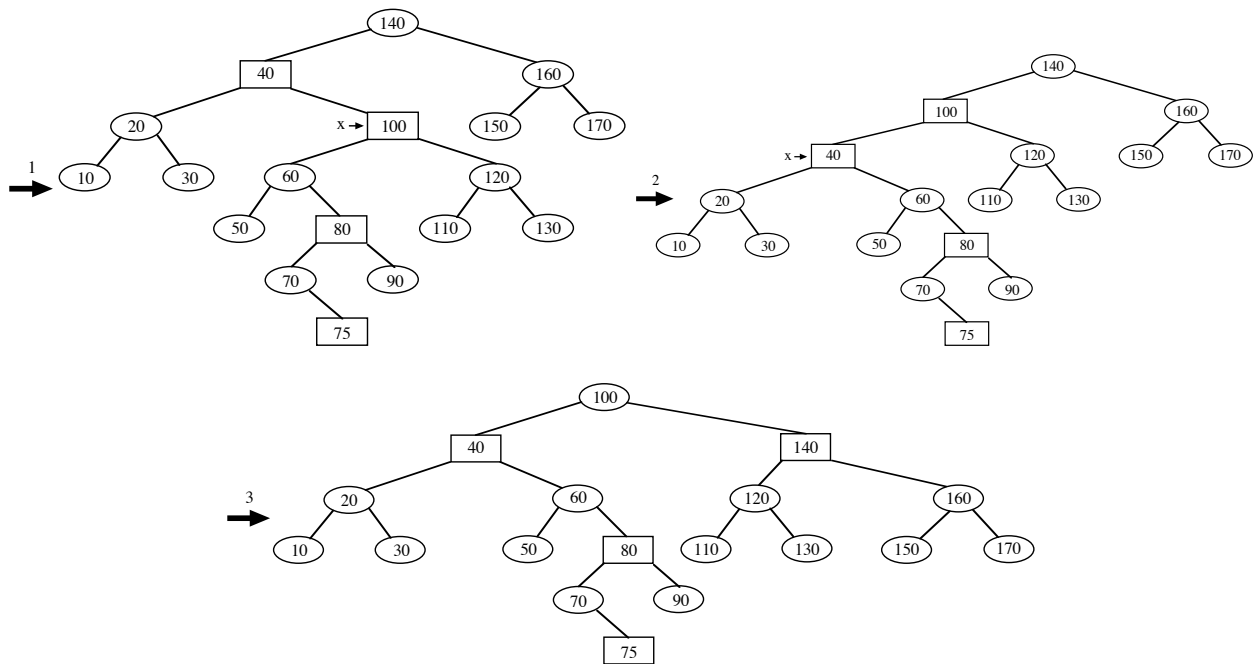


Example:



Insert 75



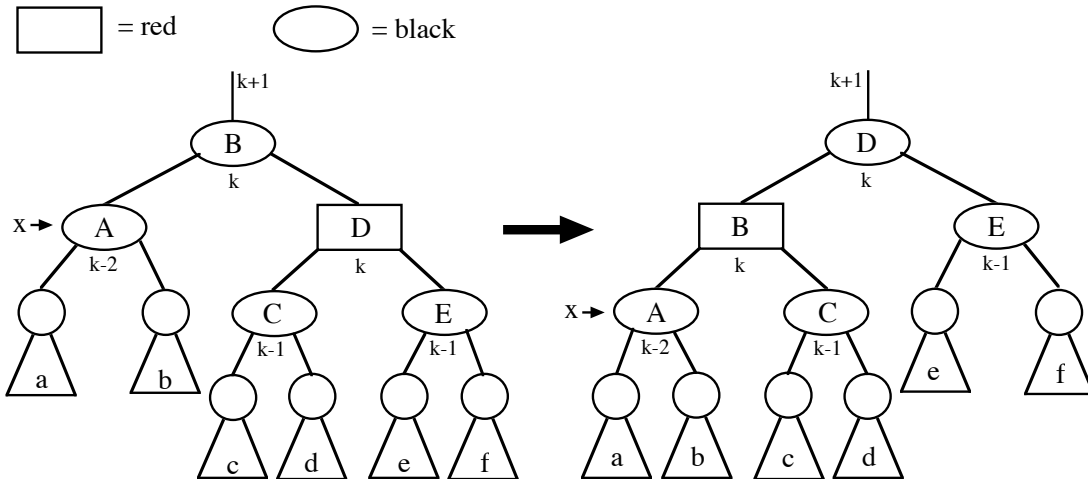


Deletion

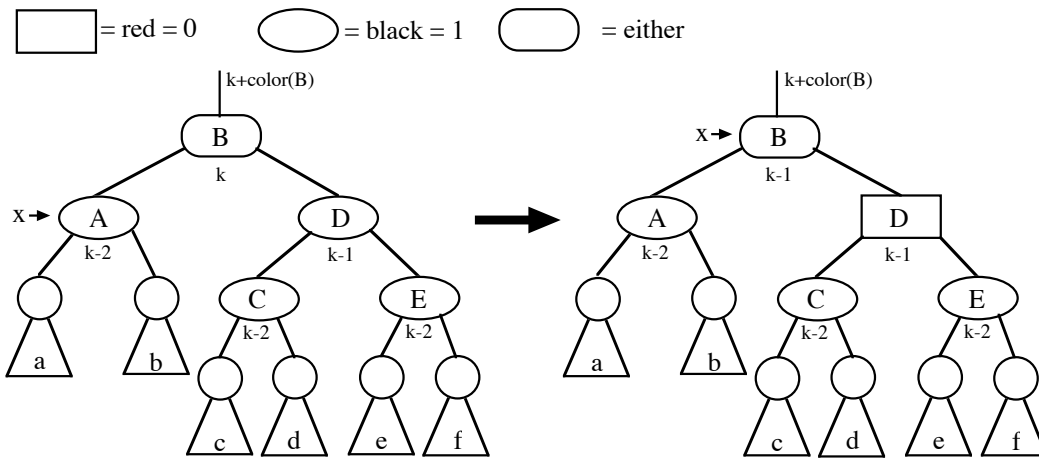
Start with one of the unbalanced deletion cases:

1. Deleted node is a “data leaf”.
 - a. Splice around to sentinel.
 - b. Color of deleted node?
 - Red \Rightarrow Done
 - Black \Rightarrow Set “double black” pointer at sentinel.
Determine which of four rebalancing cases applies.
2. Deleted node is parent of one “data leaf”.
 - a. Splice around to “data leaf”
 - b. Color of deleted node?
 - Red \Rightarrow Not possible
 - Black \Rightarrow “data leaf” must be red. Change its color to black.
3. Node with key-to-delete is parent of two “data nodes”.
 - a. “Steal” key and data from successor (but not the color).
 - b. “Delete” successor using the appropriate one of the previous two cases.

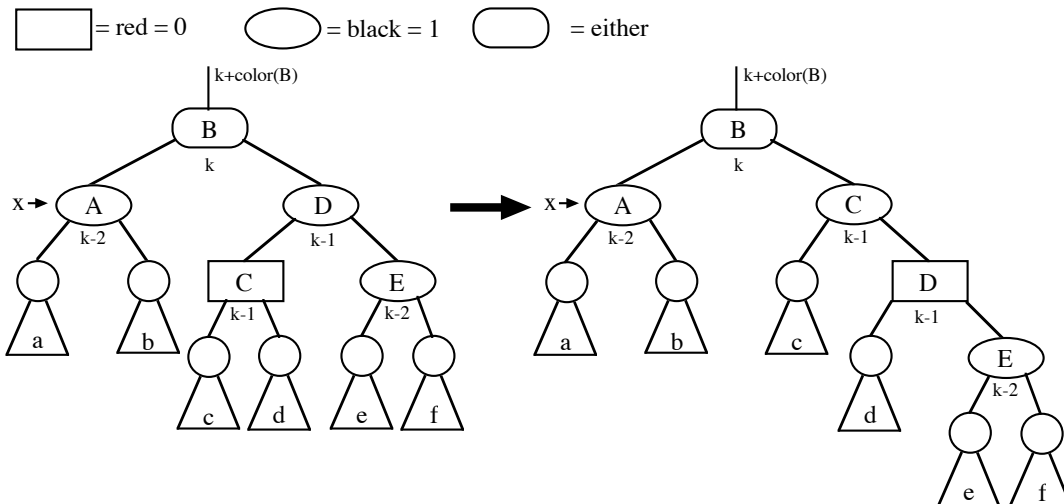
Case 1:



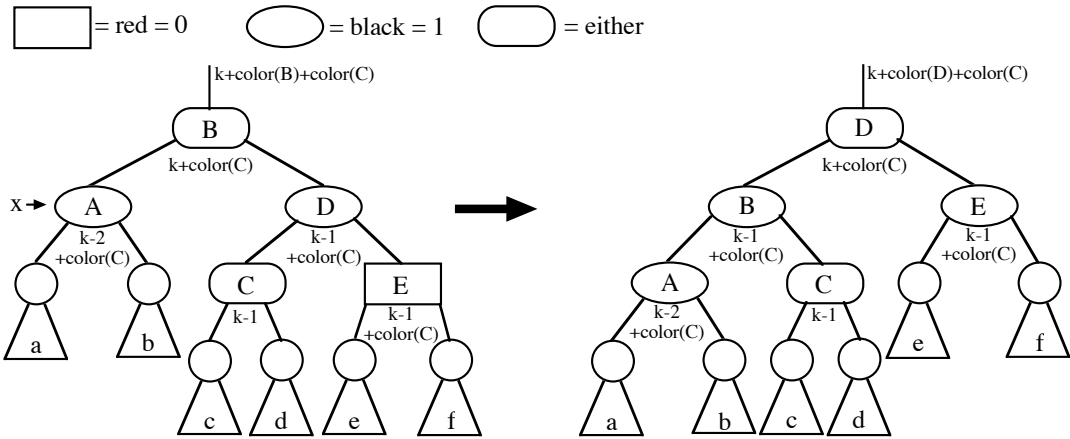
Case 2:



Case 3:

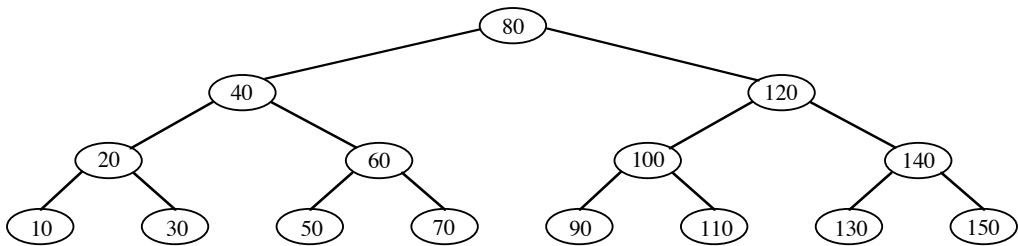


Case 4:

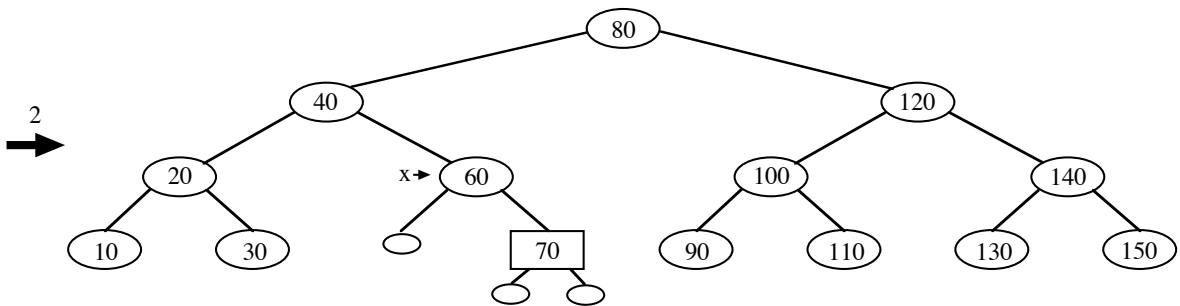
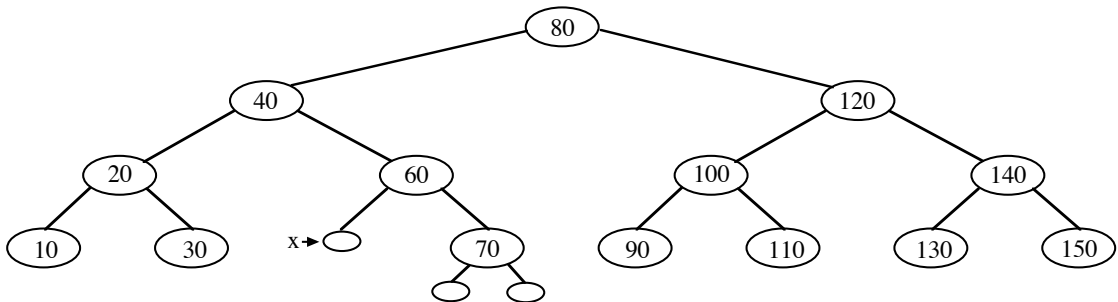


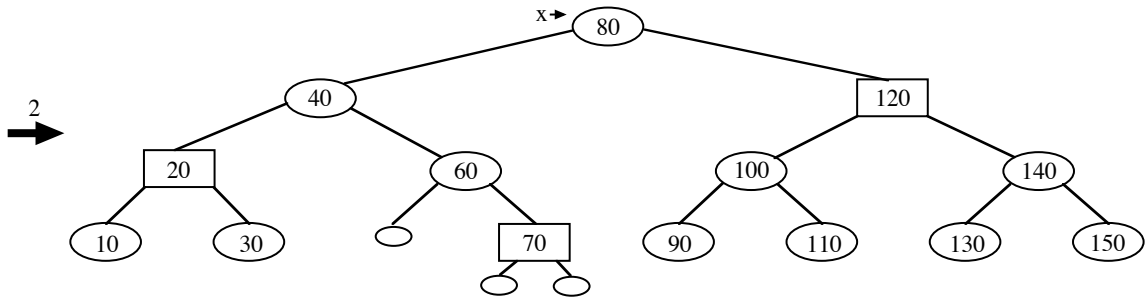
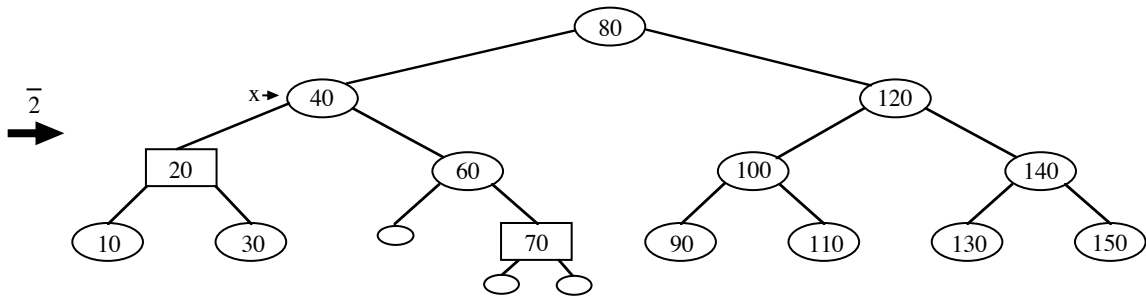
(At most three rotations occur while processing the deletion of one key)

Example:



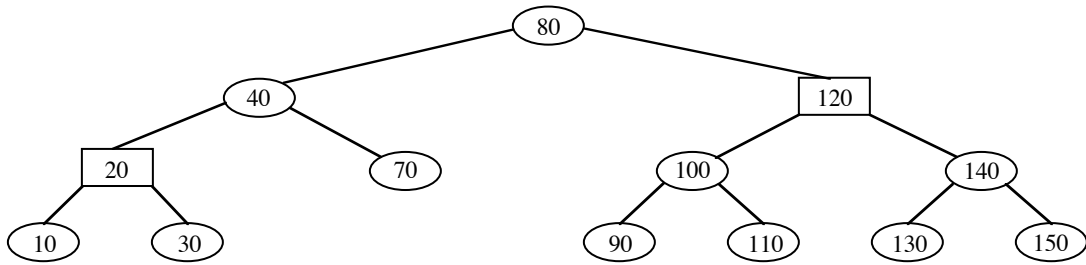
Delete 50



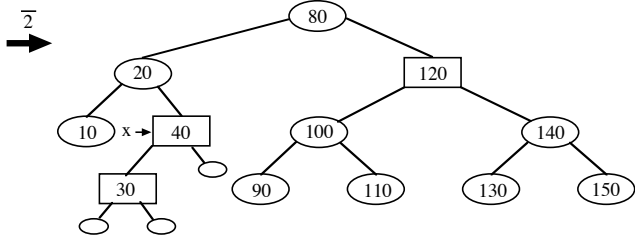
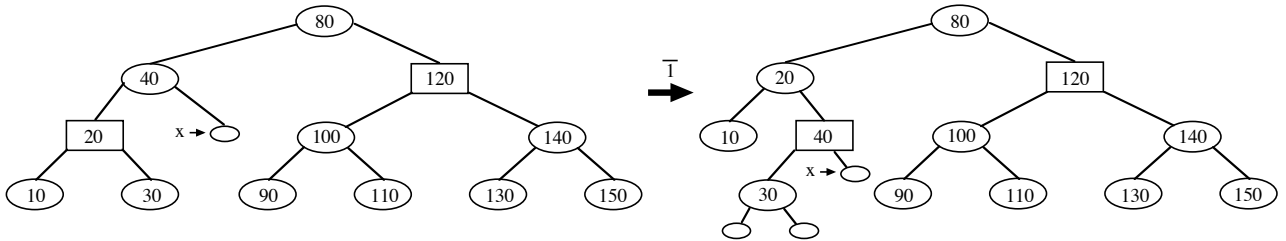


If x reaches the root, then done. Only place in tree where this happens.

Delete 60

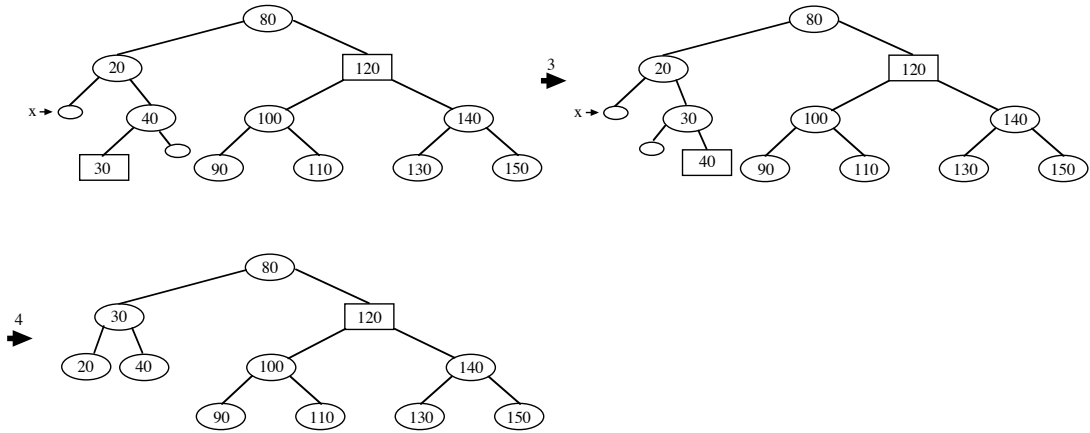


Delete 70

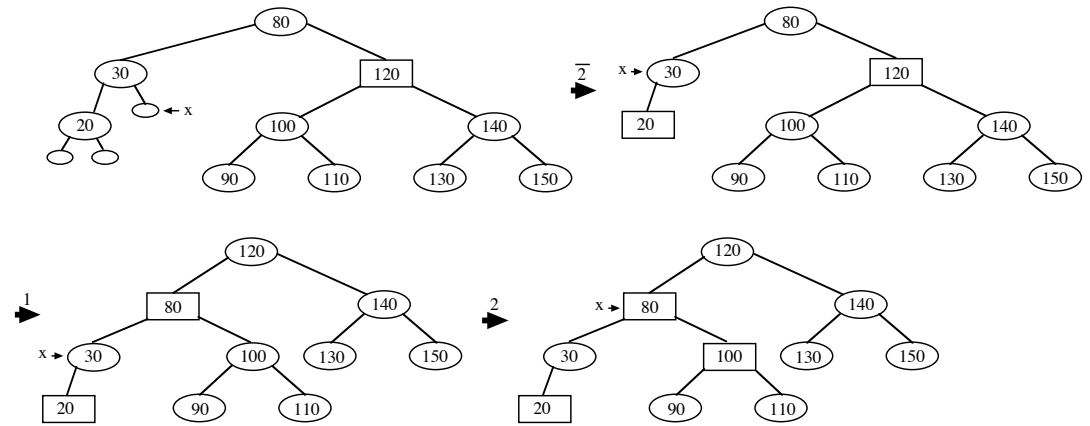


If x reaches a red node, then change color to black and done.

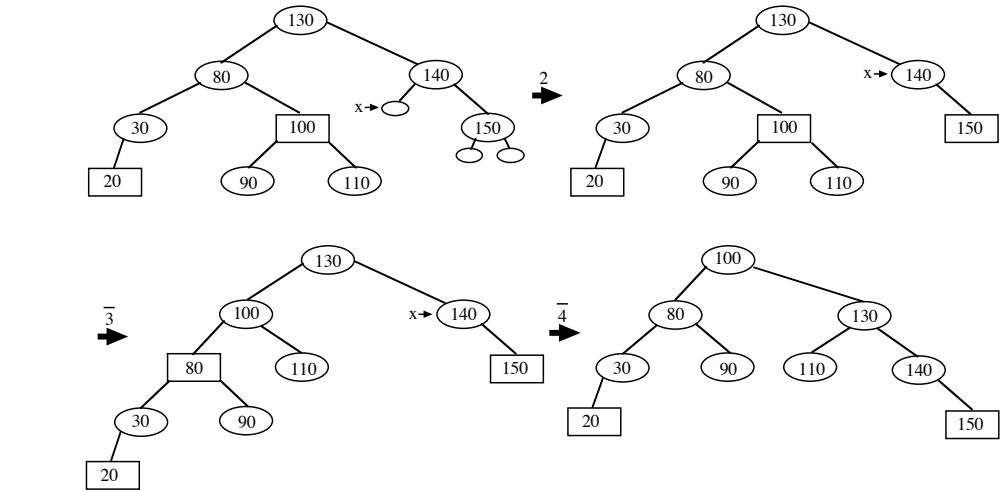
Delete 10



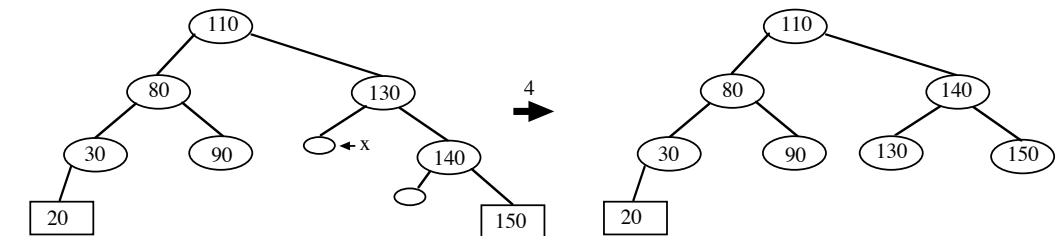
Delete 40



Delete 120



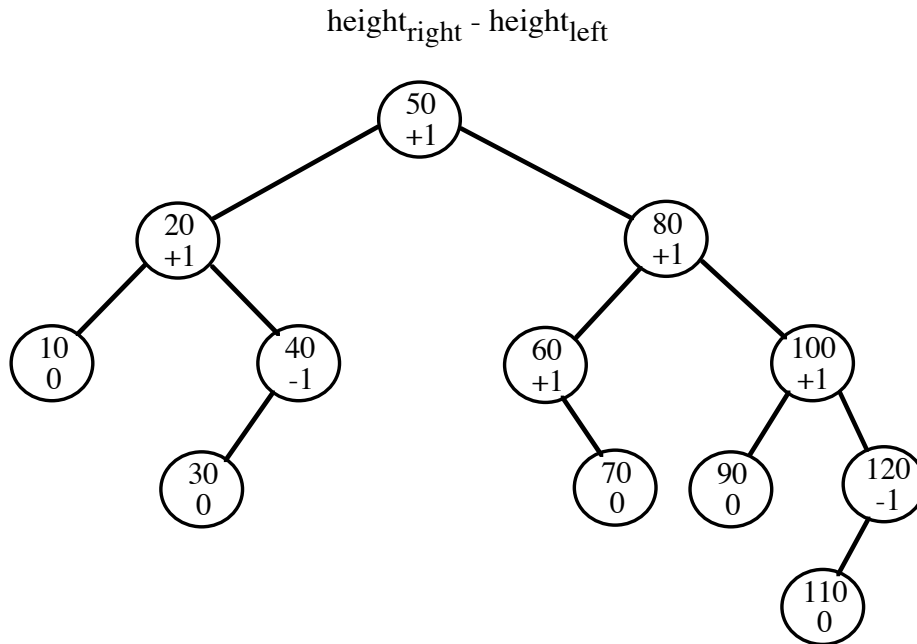
Delete 100



AVL TREES

An *AVL tree* is a binary search tree whose height is logarithmic in the number of keys stored.

- Each node stores the difference of the heights (known as the *balance factor*) of the right and left subtrees rooted by the children:



- A balance factor must be +1, 0, -1 (leans right, “balanced”, leans left).

- An insertion is implemented by:
 - Attaching a leaf

- Rippling changes to balance factor:
 - Right child ripple

Parent.Bal = 0 \Rightarrow +1 and ripple to parent

Parent.Bal = -1 \Rightarrow 0 to complete insertion

Parent.Bal = +1 \Rightarrow +2 and ROTATION to complete insertion

- Left child ripple

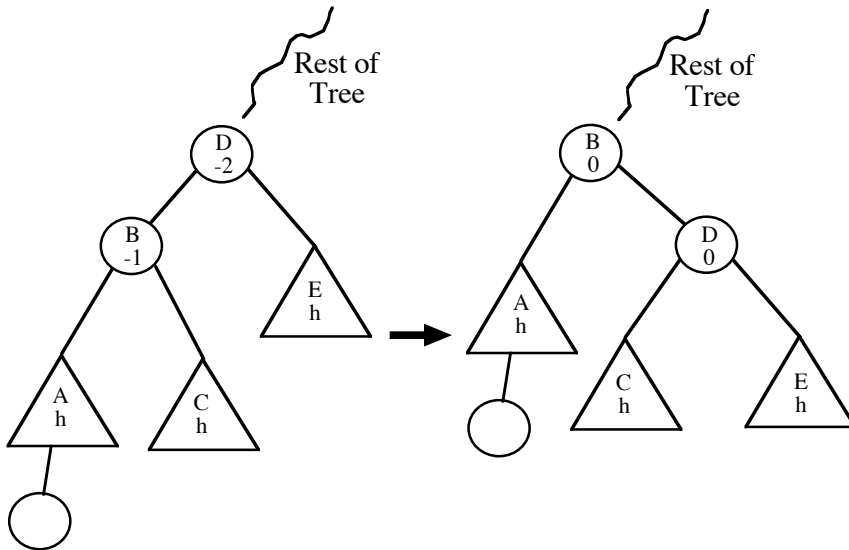
Parent.Bal = 0 \Rightarrow -1 and ripple to parent

Parent.Bal = +1 \Rightarrow 0 to complete insertion

Parent.Bal = -1 \Rightarrow -2 and ROTATION to complete insertion

4. Rotations

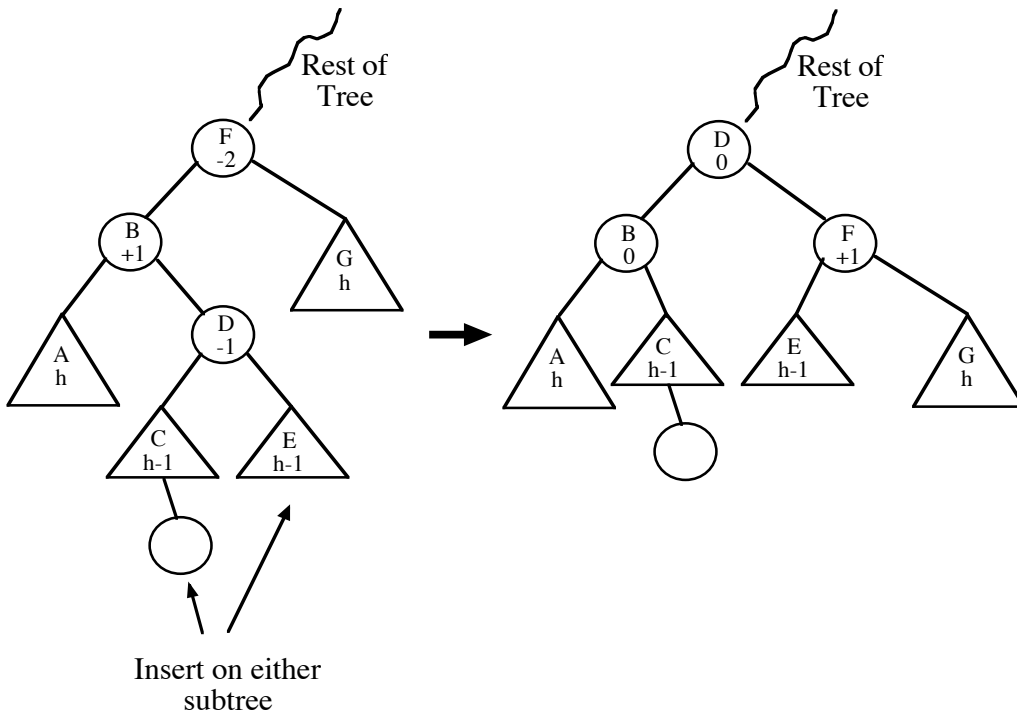
a. Single (LL) - right rotation at D



Restores height of subtree to pre-insertion number of levels

RR case is symmetric

b. Double (LR)



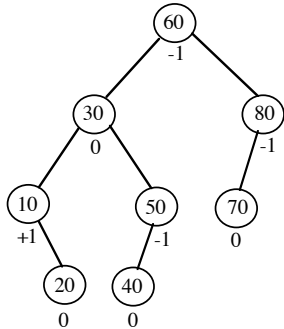
Restores height of subtree to pre-insertion number of levels

RL case is symmetric

Deletion -

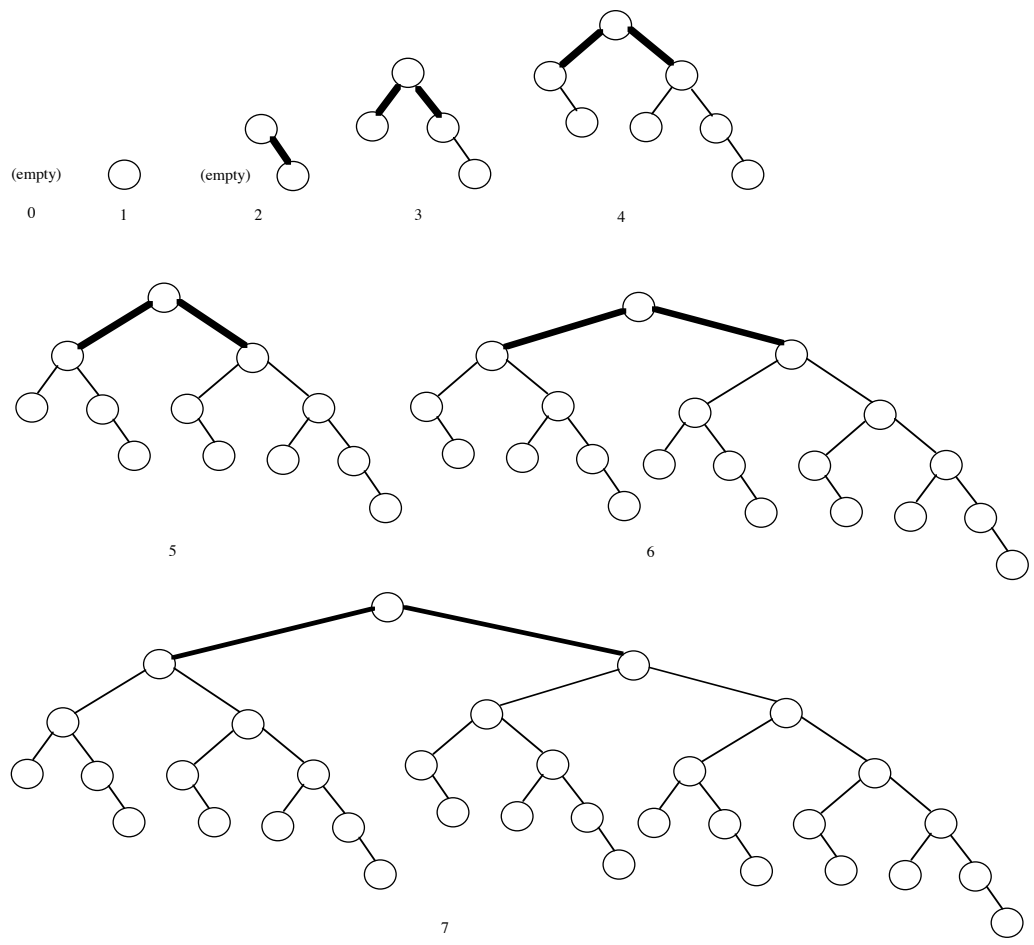
Still have RR, RL, LL, and LR, but two additional (symmetric) cases arise.

Suppose 70 is deleted from this tree. Either LL or LR may be applied.



Fibonacci Trees - special case of AVL trees exhibiting two worst-case behaviors -

1. Maximally skewed. (max height is roughly $\log_{1.618} n = 1.44 \lg n$, expected height is $\lg n + .25$)
2. $\theta(\log n)$ rotations for a single deletion.



TREAPS (CLRS, p. 333)

Hybrid of BST and min-heap ideas

Gives code that is clearer than RB or AVL (but comparable to skip lists)

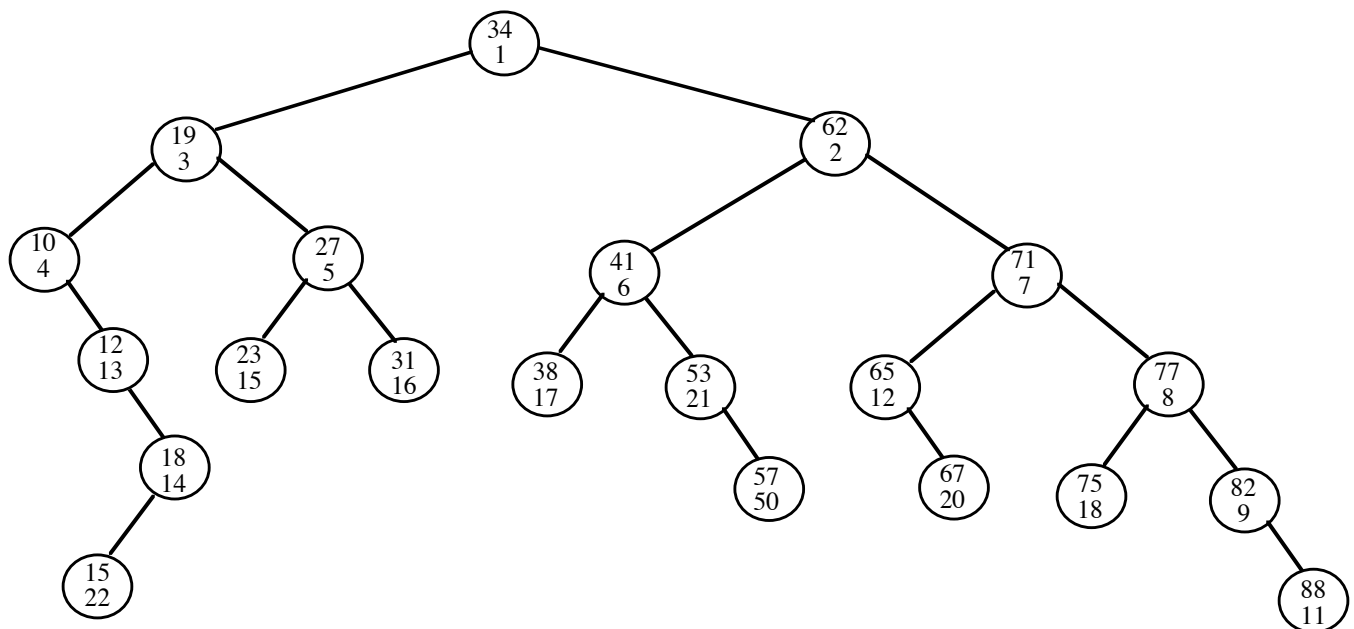
Expected height of tree is logarithmic ($2.5 \lg n$)

Keys are used as in BST

Tree also has min-heap property based on each node having a priority:

Randomized priority - generated when a new key is inserted

Virtual priority - computed (when needed) using a function similar to a hash function



Asides: the first published such hybrid were the *cartesian trees* of J. Vuillemin, “A Unifying Look at Data Structures”, *C. ACM* 23 (4), April 1980, 229-239. A more complete explanation appears in E.M. McCreight, “Priority Search Trees”, *SIAM J. Computing* 14 (2), May 1985, 257-276 and chapter 10 of M. de Berg et.al. These are also used in the elegant implementation in M.A. Babenko and T.A. Starikovskaya, “Computing Longest Common Substrings” in E.A. Hirsch, *Computer Science - Theory and Applications*, LNCS 5010, 2008, 64-75.

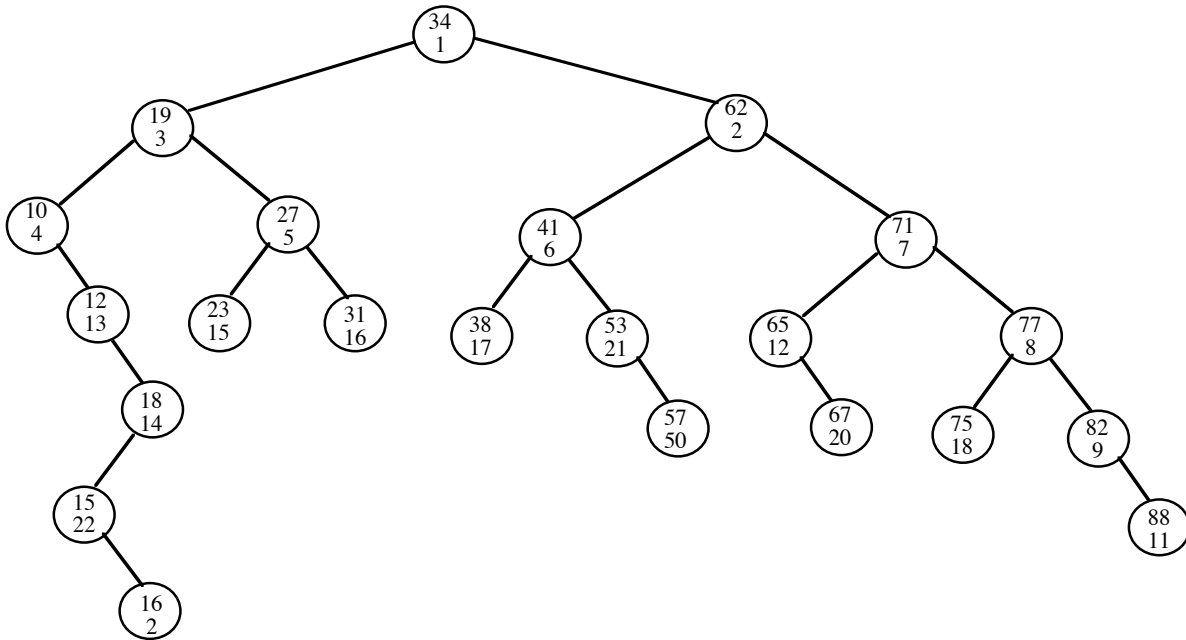
Insertion

Insert as leaf

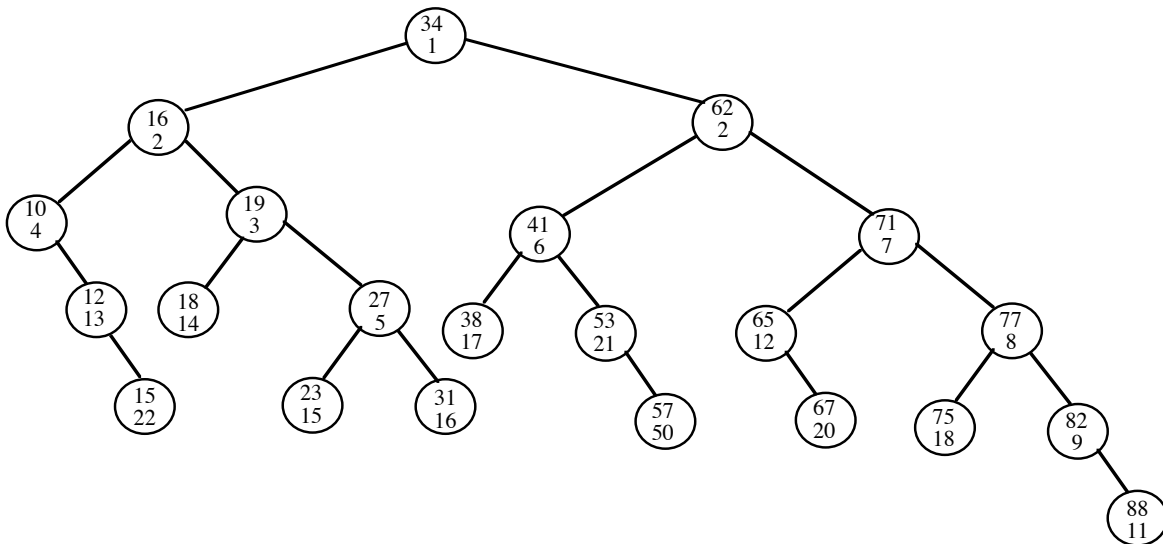
Generate random priority (large range to minimize duplicates)

Single rotations to fix min-heap property

Example: Insert 16 with a priority of 2



After rotations:



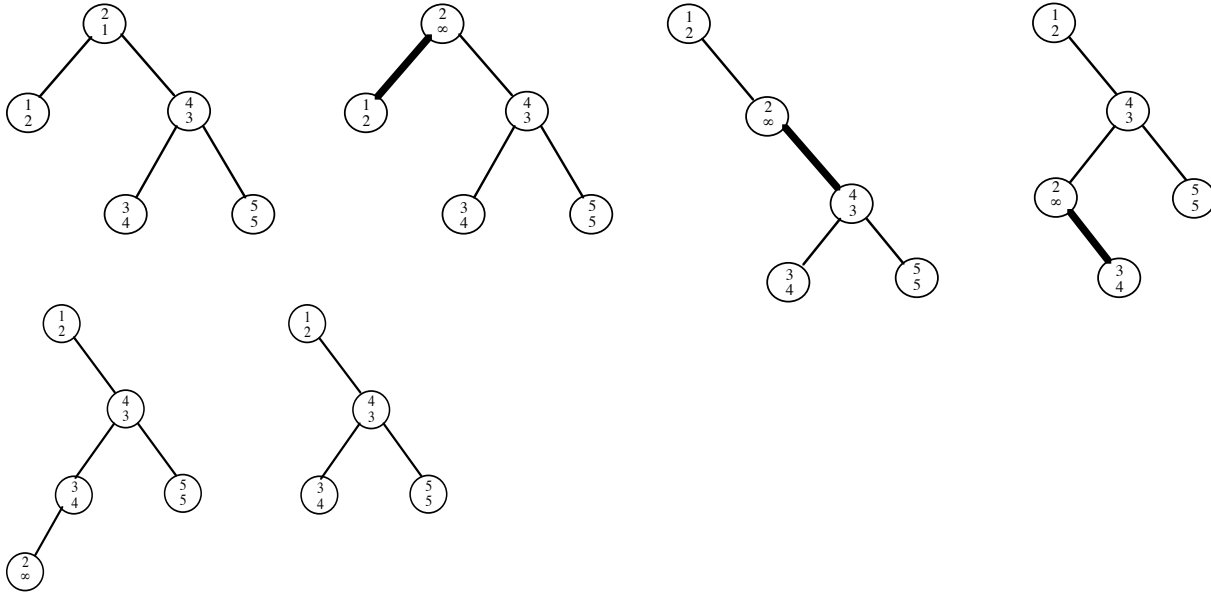
Deletion

Find node and change priority to ∞

Rotate to bring up child with lower priority. Continue until min-heap property holds.

Remove leaf.

Delete key 2:



AUGMENTING DATA STRUCTURES

Read CLRS, section 14.1 on using RB tree with *ranking* information for order statistics.

Retrieving an element with a given rank

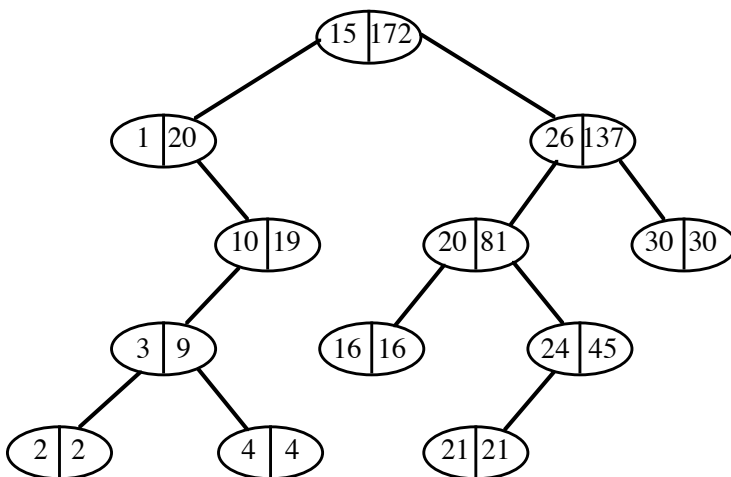
Determine the rank of an element

Problem: Maintain summary information to support an aggregate operation on the k smallest (or largest) keys in $O(\log n)$ time.

Example: Prefix Sum

Given a key, determine the sum of all keys \leq given key (prefix sum).

Solution: Store sum of all keys in a subtree at the root of the subtree.



Key	Prefix Sum
1	1
2	3
3	6
4	10
10	20
15	35
16	51
20	71
21	92
24	116
26	142
30	172

To compute prefix sum for a key:

Initialize sum to 0

Search for key, modifying total as search progresses:

Search goes left - leave total alone

Search goes right or key has been found - add present node's key and left child's sum to total

Key is 24: $(15 + 20) + (20 + 16) + (24 + 21) = 116$

Key is 10: $(1 + 0) + (10 + 9) = 20$

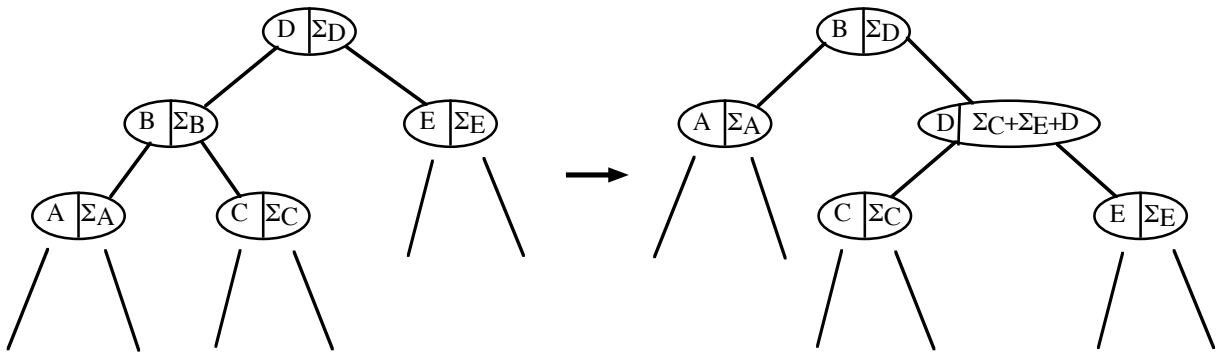
Key is 16: $(15 + 20) + (16 + 0) = 51$

Variation: Determine the smallest key that has a prefix sum \geq a specified value.

Updates to tree:

Non-structural (attach/remove node) - modify node and every ancestor

Single rotation (for prefix sum)



(Similar for double rotation)

General case - see CLRS 14.2, especially “Theorem” 14.1

Interval trees (CLRS 14.3) - a more significant application

Set of (closed) intervals $[low, high]$ - low is the key, but duplicates are allowed

Each subtree root contains the *max* value appearing in any interval in that subtree

Aggregate operation to support - find any interval that overlaps a given interval $[low', high']$

Modify BST search . . .

```

if ptr == nil
    no interval in tree overlaps [low', high']

if high' ≥ ptr->low and ptr->high ≥ low'
    return ptr as an answer

if ptr->left != nil and ptr->left->max ≥ low'
    ptr := ptr->left
else
    ptr := ptr->right

```

Updates to tree - similar to prefix sum, but replace additions with maximums

OPTIMAL BINARY SEARCH TREES

What is the optimal way to organize a *static* list for searching?

1. By decreasing access probability - optimal static/fixed ordering.
2. Key order - if misses will be “frequent”, to avoid searching entire list.

Other Considerations:

1. Access probabilities may change (or may be unknown).
2. Set of keys may change.

These lead to proposals (later in this set of notes) for (online) data structures whose adaptive behavior is asymptotically close (analyzed in Notes 3) to that of an optimal (offline) strategy.

Online - must process each request before the next request is revealed.

Offline - given the entire sequence of requests before any processing. (“knows the future”)

What is the *optimal* way to organize a *static* tree for searching?

An *optimal (static) binary search tree* is significantly more complicated to construct than an optimal list.

1. Assume access probabilities are known:

keys are $K_1 < K_2 < \dots < K_n$

p_i = probability of request for K_i

q_i = probability of request with $K_j < \text{request} < K_{j+1}$

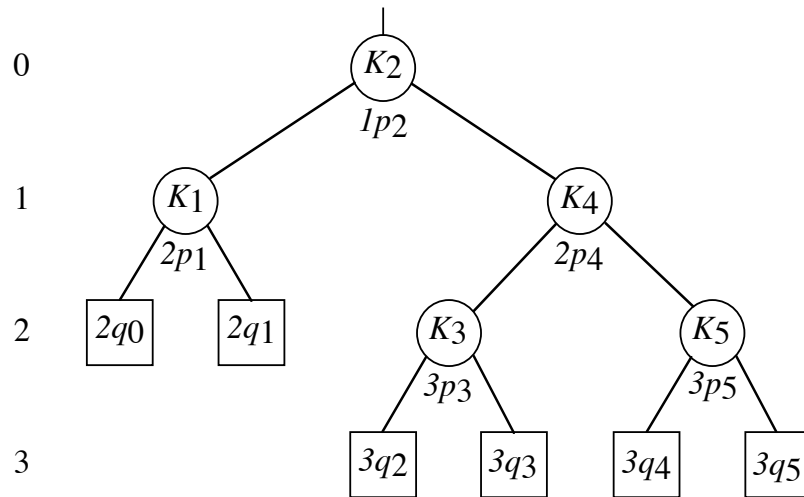
q_0 = probability of request $< K_1$

q_n = probability of request $> K_n$

2. Assume that levels are numbered with root at level 0. Minimize the expected number of comparisons to complete a search:

$$\sum_{j=1}^n p_j (\text{KeyLevel}(j) + 1) + \sum_{j=0}^n q_j \text{MissLevel}(j)$$

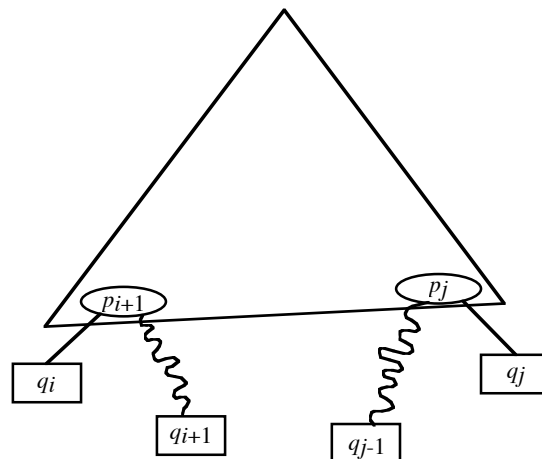
3. Example tree:



4. Solution is by dynamic programming:

Principle of optimality - solution is not optimal unless the subtrees are optimal.

Base case - empty tree, costs nothing to search.



$c(i, j)$ – cost of subtree with keys K_{i+1}, \dots, K_j

$c(i, j)$ always includes exactly p_{i+1}, \dots, p_j and q_i, \dots, q_j

$c(i, i) = 0$ – Base case, no keys, just misses for q_i (request between K_i and K_{i+1})

Recurrence for finding optimal subtree:

$$c(i,j) = w(i,j) + \min_{i < k \leq j} (c(i,k-1) + c(k,j))$$

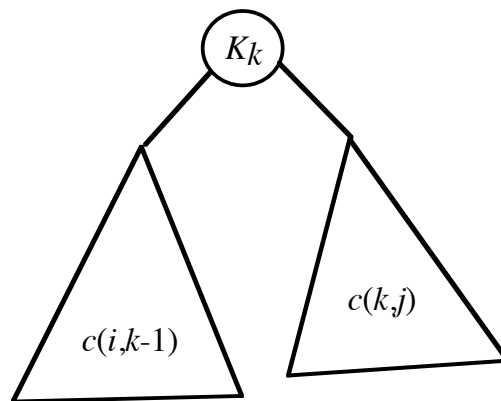
tries every possible root (“ k ”) for the subtree with keys K_{i+1}, \dots, K_j

$w(i,j) = p_{i+1} + \dots + p_j + q_i + \dots + q_j$ accounts for adding another probe for all keys in subtree :

Left: $p_{i+1} + \dots + p_{k-1} + q_i + \dots + q_{k-1}$

Right: $p_{k+1} + \dots + p_j + q_k + \dots + q_j$

Root: p_k



5. Implementation: A k -family is all cases for $c(i, i+k)$. k -families are computed in ascending order from 1 to n . Suppose $n = 5$:

0	1	2	3	4	5
$c(0,0)$	$c(0,1)$	$c(0,2)$	$c(0,3)$	$c(0,4)$	$c(0,5)$
$c(1,1)$	$c(1,2)$	$c(1,3)$	$c(1,4)$	$c(1,5)$	
$c(2,2)$	$c(2,3)$	$c(2,4)$	$c(2,5)$		
$c(3,3)$	$c(3,4)$	$c(3,5)$			
$c(4,4)$	$c(4,5)$				
$c(5,5)$					

Complexity: $O(n^2)$ space is obvious. $O(n^3)$ time from:

$$\sum_{k=1}^n k(n+1-k)$$

where k is the number of roots for each $c(i, i+k)$ and $n+1-k$ is the number of $c(i, i+k)$ cases in family k .

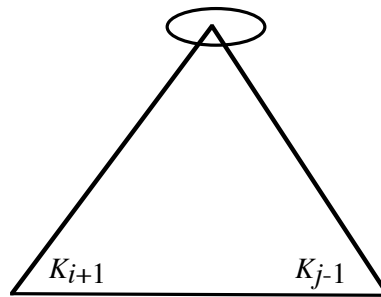
6. Traceback - besides having the minimum value for each $c(i, j)$, it is necessary to save the subscript for the optimal root for $c(i, j)$ as $r[i][j]$.

This also leads to Knuth's improvement:

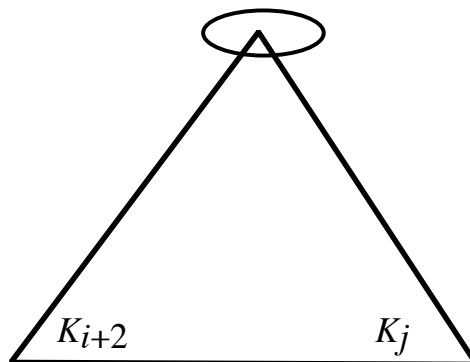
Theorem: The root for the optimal tree $c(i, j)$ must have a key with subscript no less than the key subscript for the root of the optimal tree for $c(i, j - 1)$ and no greater than the key subscript for the root of optimal tree $c(i + 1, j)$. (These roots are computed in the preceding family.)

Proof:

1. Consider adding p_j and q_j to tree for $c(i, j - 1)$. Optimal tree for $c(i, j)$ must keep the same key at the root or use one further to the right.



2. Consider adding p_{i+1} and q_i to tree for $c(i + 1, j)$. Optimal tree for $c(i, j)$ must keep the same key at the root or use one further to the left.



7. Analysis of Knuth's improvement.

Each $c(i, j)$ case for k -family will vary in the number of roots to try, but overall time is reduced to $O(n^2)$ by using a telescoping sum:

$$\begin{aligned}
\sum_{k=2}^n \sum_{i=0}^{n-k} (r[i+1][i+k] - r[i][i+k-1] + 1) &= \sum_{k=2}^n \left(\begin{array}{c} r[1][k] - r[0][k-1] + 1 \\ + \\ r[2][1+k] - r[1][k] + 1 \\ + \\ r[3][2+k] - r[2][1+k] + 1 \\ + \cdots + \\ r[n-k+1][n] - r[n-k][n-1] + 1 \end{array} \right) \\
&= \sum_{k=2}^n (r[n-k+1][n] - r[0][k-1] + n - k + 1) \\
&\leq \sum_{k=2}^n (n - 0 + n - k + 1) = \sum_{k=2}^n (2n - k + 1) = O(n^2)
\end{aligned}$$

```

n=7;
q[0]=0.06;
p[1]=0.04;
q[1]=0.06;
p[2]=0.06;
q[2]=0.06;
p[3]=0.08;
q[3]=0.06;
p[4]=0.02;
q[4]=0.05;
p[5]=0.10;
q[5]=0.05;
p[6]=0.12;
q[6]=0.05;
p[7]=0.14;
q[7]=0.05;
for (i=1;i<=n;i++)
  key[i]=i;

```

```

w[0][0]=0.060000
w[0][1]=0.160000
w[0][2]=0.280000
w[0][3]=0.420000
w[0][4]=0.490000
w[0][5]=0.640000
w[0][6]=0.810000
w[0][7]=1.000000
w[1][1]=0.060000
w[1][2]=0.180000
w[1][3]=0.320000
w[1][4]=0.390000
w[1][5]=0.540000
w[1][6]=0.710000
w[1][7]=0.900000
w[2][2]=0.060000
w[2][3]=0.200000
w[2][4]=0.270000
w[2][5]=0.420000
w[2][6]=0.590000
w[2][7]=0.780000
w[3][3]=0.060000
w[3][4]=0.130000
w[3][5]=0.280000
w[3][6]=0.450000
w[3][7]=0.640000
w[4][4]=0.050000
w[4][5]=0.200000
w[4][6]=0.370000
w[4][7]=0.560000
w[5][5]=0.050000
w[5][6]=0.220000
w[5][7]=0.410000
w[6][6]=0.050000
w[6][7]=0.240000
w[7][7]=0.050000

```

Counts - root trick 44 without root
trick 77

Average probe length is 2.680000
trees in parenthesized prefix

```

c(0,0) cost 0.000000
c(1,1) cost 0.000000
c(2,2) cost 0.000000
c(3,3) cost 0.000000
c(4,4) cost 0.000000
c(5,5) cost 0.000000
c(6,6) cost 0.000000
c(7,7) cost 0.000000
c(0,1) cost 0.160000 1
c(1,2) cost 0.180000 2
c(2,3) cost 0.200000 3
c(3,4) cost 0.130000 4
c(4,5) cost 0.200000 5

```

```

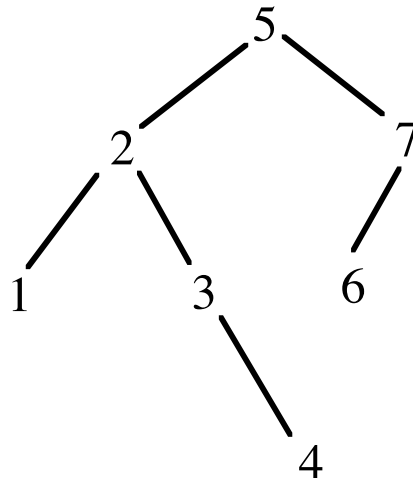
c(5,6) cost 0.220000 6
c(6,7) cost 0.240000 7
c(0,2) cost 0.440000 2(1,)
c(1,3) cost 0.500000 3(2,)
c(2,4) cost 0.400000 3(,4)
c(3,5) cost 0.410000 5(4,)
c(4,6) cost 0.570000 6(5,)
c(5,7) cost 0.630000 7(6,)
c(0,3) cost 0.780000 2(1,3)
c(1,4) cost 0.700000 3(2,4)
c(2,5) cost 0.820000 4(3,5)
c(3,6) cost 0.800000 5(4,6)
c(4,7) cost 1.000000 6(5,7)
c(0,4) cost 1.050000 2(1,3(,4))
c(1,5) cost 1.130000 3(2,5(4,))
c(2,6) cost 1.210000 5(3(,4),6)
c(3,7) cost 1.290000 6(5(4,),7)
c(0,5) cost 1.490000 3(2(1,),5(4,))
c(1,6) cost 1.630000 5(3(2,4),6)
c(2,7) cost 1.810000 5(3(,4),7(6,))
c(0,6) cost 2.050000 3(2(1,),5(4,6))
c(1,7) cost 2.230000 5(3(2,4),7(6,))
c(0,7) cost 2.680000 5(2(1,3(,4)),7(6,))

```

$$3: c(0,2) + c(3,7) + w[0][7] \\ 0.44 \quad 1.29 \quad 1.0 \quad = 2.73$$

$$4: c(0,3) + c(4,7) + w[0][7] \\ 0.78 \quad 1.0 \quad 1.0 \quad = 2.78$$

$$5: c(0,4) + c(5,7) + w[0][7] \\ 1.05 \quad 0.63 \quad 1.0 \quad = 2.68$$



CONCEPTS OF SELF-ORGANIZING LINEAR SEARCH

Have list adapt to give better performance.

Advantages:

Simple to code.

Convenient for situations with relatively small # of elements to avoid more elaborate mechanism.

Useful for some user interfaces.

Access Distributions for Probabilistic Analysis:

Uniform - Theoretically convenient

80-20 (or 90-10) Rule

Zipf - n items, $P_i = \frac{1}{iH_n}$, $H_n = \sum_{k=1}^n \frac{1}{k}$

Since distribution may be unknown or changing, we are dealing with

Locality (temporary heavy accesses)

vs.

Convergence (obtaining optimal ordering)

Implementation Approaches

Move-to-front (good locality)

Transpose (Slow to converge. Alternating request anomaly.)

Count - Number of accesses is stored in each record (or use CLRS problem 5-1 to reduce bits)

Sort records in decreasing count order

Move-ahead- k : more aggressive than transpose

Probabilistic analysis may be pursued by Markov (state-transition) approaches or simulation

SPLAY TREES

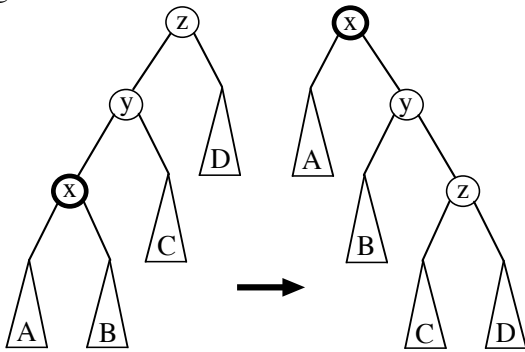
Self-adjusting counterpart to AVL and red-black trees

Advantages - 1) no balance bits, 2) some help with locality of reference, 3) amortized complexity is same as AVL and red-black trees

Disadvantage - worst-case for operation is $O(n)$

Algorithms are based on rotations to *splay* the last node processed (x) to root position.

Zig-Zig:

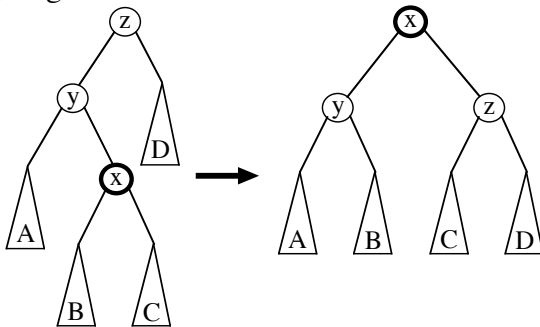


1. Single right rotation at z .

2. Single right rotation at y .

(+ symmetric case)

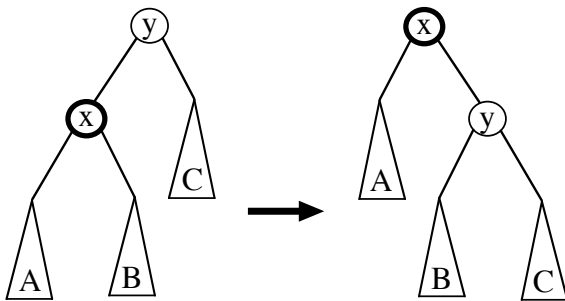
Zig-Zag:



Double right rotation at z .

(+ symmetric case)

Zig: Applies ONLY at the root



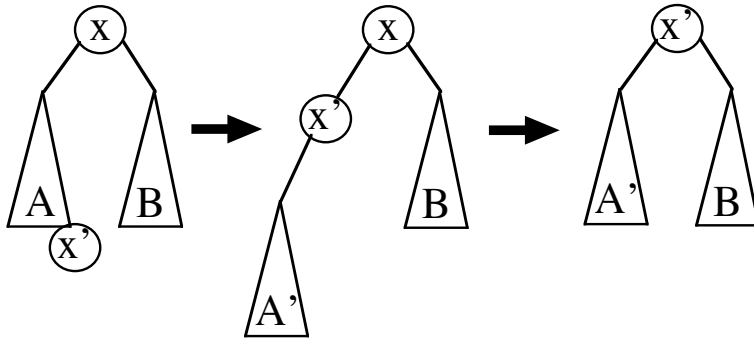
Single right rotation at y .

(+ symmetric case)

Insertion: Attach new leaf and then splay to root.

Deletion:

1. Access node x to delete, including splay to root.



2. Access predecessor x' in left subtree A and splay to root of left subtree.
3. Take right subtree of x and make it the right subtree of x' .