CSI 403
Design and Analysis of Algorithms

Lecture 12 – Introduction to Height-Balanced Trees (AVL)

J Marques de Carvalho

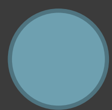
Review: Trees

- Trees (in general)
  - Root
  - Internal Nodes
  - Leaves
- Binary Trees
  - Zero, one, or two children per node
- Nodes consist of (at a minimum):
  - *Some* data
  - Left and Right child pointers
  - The child pointers are NULL if the corresponding child nodes do not exist

Full Trees

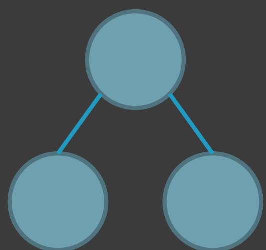
- When we first started talking about binary trees, we said that a _full_ (or _complete_) binary tree had no NULL pointers on internal nodes (only leaves have NULL pointers)

Full Trees and Tree Height

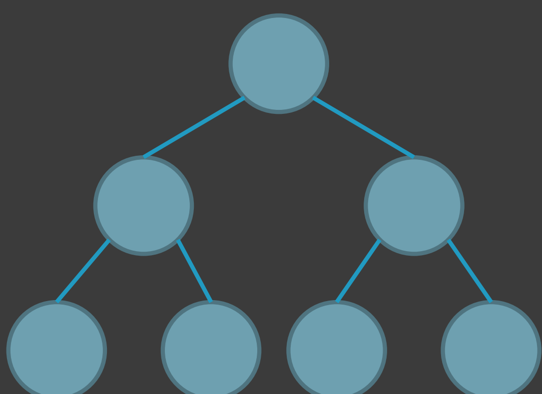


1 node: h = 1

Completely full trees will have $2^h - 1$ nodes.



3 nodes: h = 2



7 nodes: h = 3

15 nodes: h = 4

Full Trees and Tree Height

- In general, a full tree with h levels will have

  $2^{(h-1)} \leq n \leq 2^{(h)} - 1$ nodes

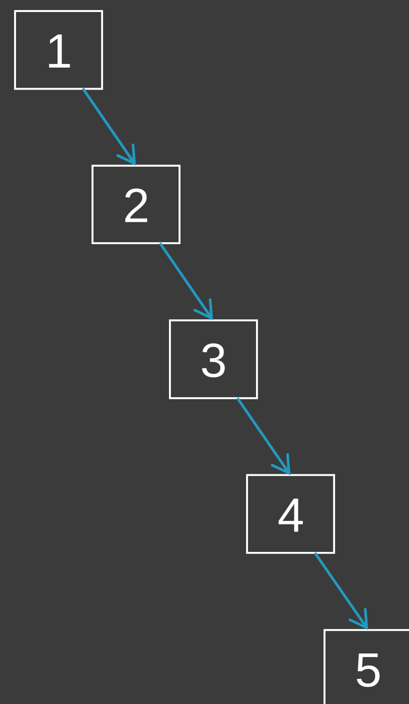
h	n_{min}	n_{Max}
1	1	1
2	2	3
3	4	7
4	8	15
5	16	31
6	32	63
7	64	127
8	128	255

Full trees are, by definition, always balanced.

Adding Items To A Binary Tree

- The resulting tree depends on the order in which the items were added!
- If we insert 1, 2, 3, 4, and 5, *in that order*, then:
  - 1 becomes the root,
  - 2 is the right child of 1,
  - 3 is the right child of 2,
  - 4 is the right child of 3, and
  - 5 is the right child of 4:

Result When Items Inserted In Order



Tree degenerates into a simple linked list – it's IMBALANCED!

Searching and inserting are now $O(n)$, rather than $O(\lg n)$

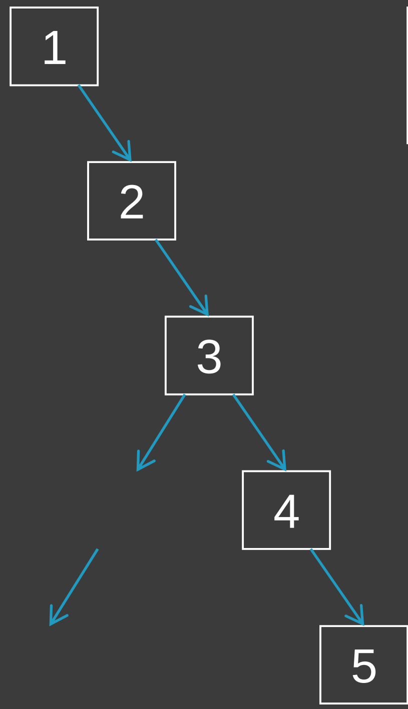
We've lost our "each step discards (roughly) half of the tree" advantage

In a tree with a million nodes, a degenerate tree will require checking (on average) 500,000 nodes; a balanced tree would require checking about 10 nodes

Balanced Trees

- We can't always control the order in which data is inserted in the tree
- The order the data is inserted determines the degree to which the tree will remain balanced
- What we need is a way to accept insertions in any order, and still maintain a balanced tree.
- Inserting a new node will create a new leaf, and may alter the tree's height → imbalance
- If inserting a new node causes the tree to become imbalanced, we need to be able to <u>*re*</u>-balance it

Rebalancing The Tree



Suppose we could “re-balance” the tree from this…

... to *this*

Balanced Trees

- There are over 100 types of balanced search trees
- Among the more often used types are AVL trees, B-Trees, and Red-Black Trees
- In a perfectly balanced tree containing N nodes, the height $h \leq (lg\ N)$
- AVL Trees don't guarantee *perfect* balance, but they DO guarantee $h \leq (\sim 1.4404\ \ lg\ N)$
- Red-Black Trees (later) guarantee $h \leq (2\ \ lg\ N)$

AVL Trees

- AVL Trees (1962)
  - Developed by Adelson-Velski and Landis
  - Every insertion (and deletion) can result in imbalance.
  - The Approach:
    - Go ahead and make the insertion
    - If an imbalance occurs, detect it and *re-balance*!
  - Re-balance operations are called “rotations”

AVL Tree Node Structure

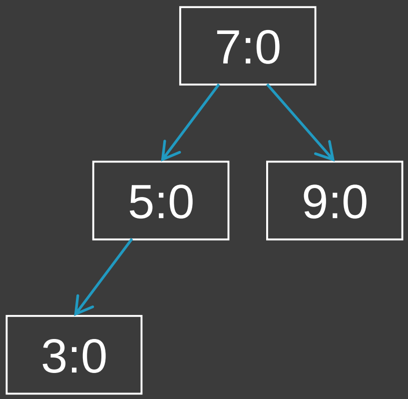
- Data
- Left and right Child Pointers
- Balance Factor (-1, 0, or +1)
  - Balance Factor (BF) at any node is the height of the node’s LEFT sub-tree minus the height of the node’s RIGHT sub-tree.
  - When insertion (or deletion) pushes a BF from -1 to -2 or from +1 to +2, we re-balance the tree
  - Tree is not maintained with perfect balance; however, it does stay “nearly balanced” or "balanced enough"

Consider This Tree

- Each node shows Data:BF
- BF at a leaf is always zero
- Height of 7’s left sub-tree = 1
- Height of 7’s right sub-tree = 1
- BF of 7 = (1 – 1) = 0

Let’s Insert A Node

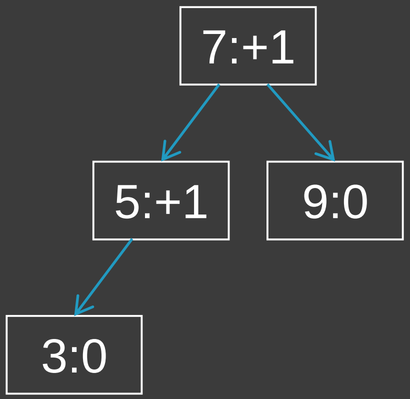
- Insert 3



- Update BFs changed by insertion

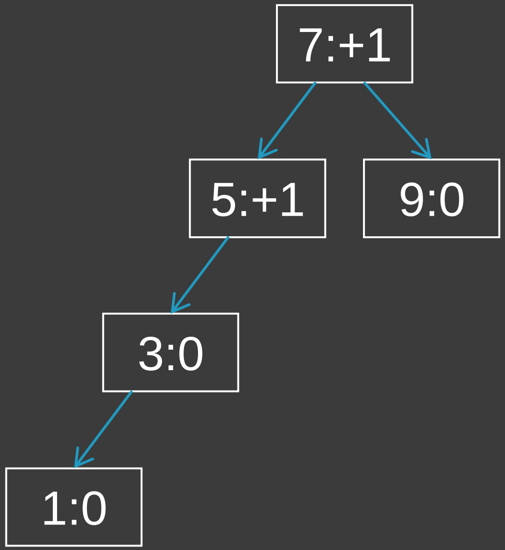
- The height of 7's left subtree (2) – the height of its right subtree (1) = +1
- The height of 5's left subtree (1) – the height of its right subtree (0) = +1
- The balance factors at 7 and 5 now become +1
- Nodes 3 and 9 are leaves, and by definition, always have a BF of 0

Balance Factors Adjusted



Let's Insert Again

- Insert 1



- Update BFs changed by insertion

- What will be the balance factors at each node after this insertion?

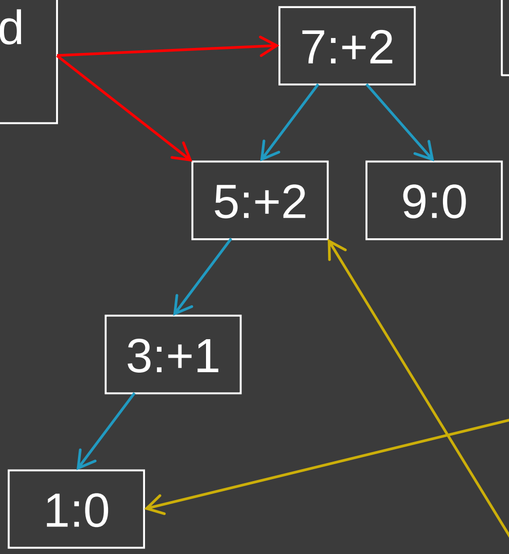
BFs Updated

An imbalance occurs when an insertion pushes a balance factor from +1 to +2 or from -1 to -2

Rotations *must* do two things:

1. Reduce the sub-tree's height by 1
2. Preserve the BST Property (same in-order traversal order)

Imbalanced Nodes!

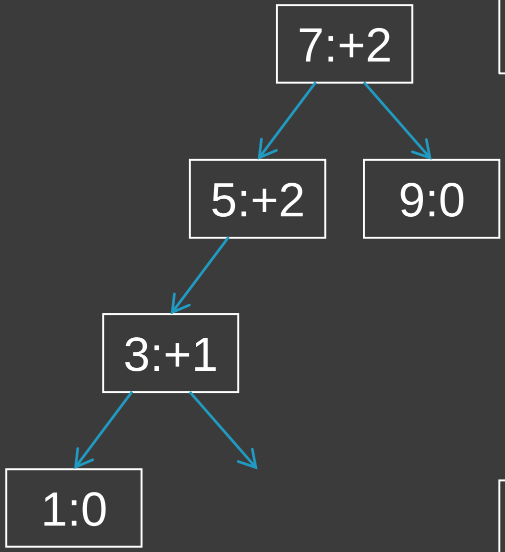


Rotations are nothing more than child pointer changes!

Relative to the node we just inserted, rotations occur below the most recent ancestor with a BF of +2 or -2

When an imbalance occurs, we solve the imbalance by applying a *rotation.*

The Rotation Itself

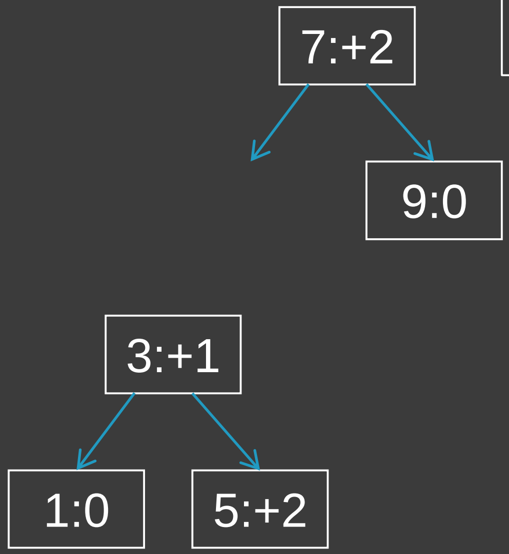


Rotations *must* do two things:

1. Reduce the sub-tree's height by 1
2. Preserve the BST Property
   (same in-order traversal order)

Rearrange the nodes (change child pointers) to reduce the sub-tree's height

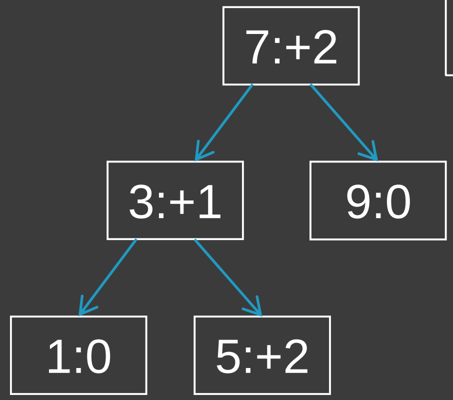
The Rotation Itself



Rotations *must* do two things:

1. Reduce the sub-tree's height by 1
2. Preserve the BST Property
   (same in-order traversal order)

The Rotation Itself



Rotations _must_ do two things:

1. Reduce the sub-tree's height by 1
2. Preserve the BST Property
   (same in-order traversal order)

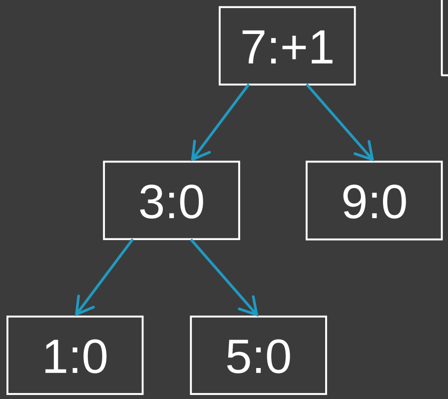
Finally, adjust BFs

What are the new (post-rotation) balance factors?

The Completed Rotation

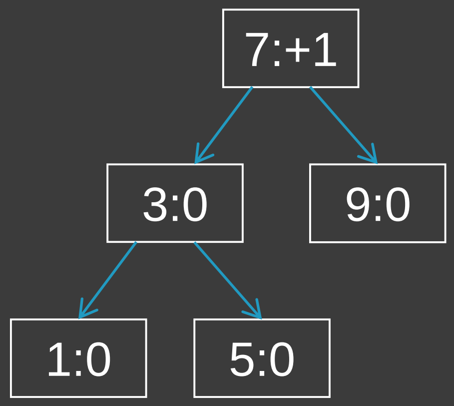
Rotations *must* do two things:

1. Reduce the sub-tree's height by 1
2. Preserve the BST Property
   (same in-order traversal order)



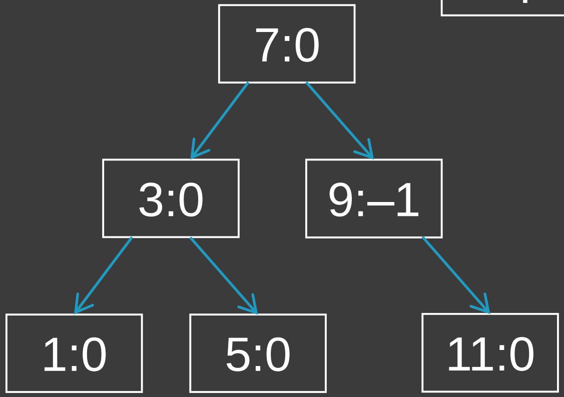
Keep Inserting

Insert 11 and adjust BFs



Keep Inserting

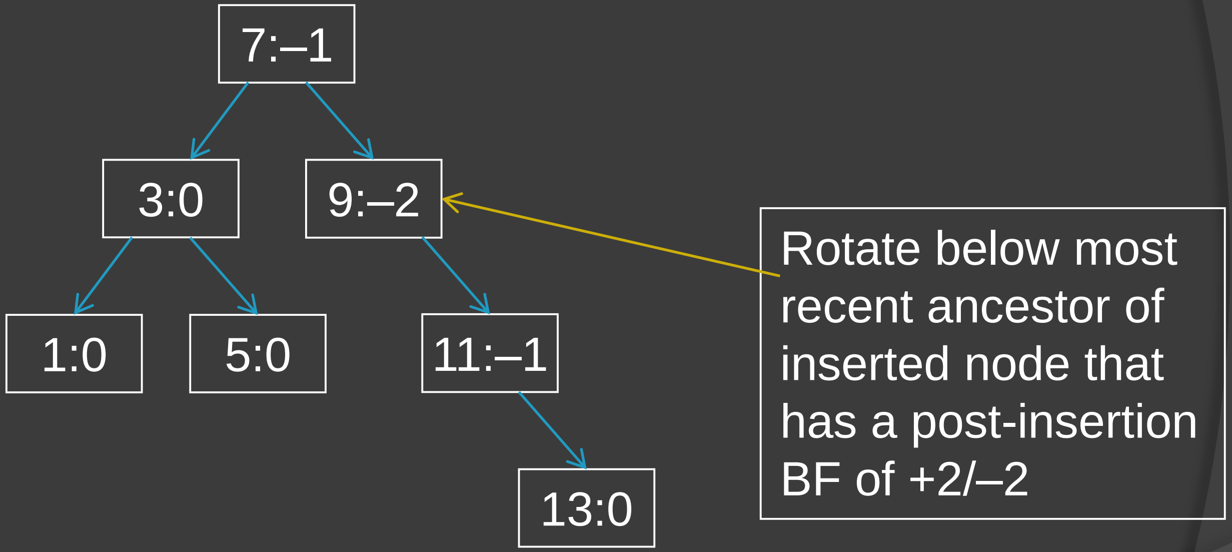
We could insert 10 *without* needing to re-balance, but inserting 13 *would* require a rotation.



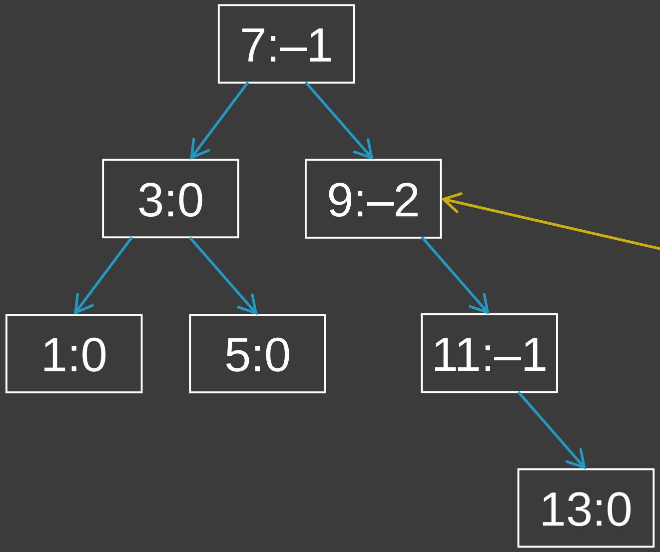
Insert 13 and adjust BFs

After we insert 13, what will all of the BF's be?

Keep Inserting

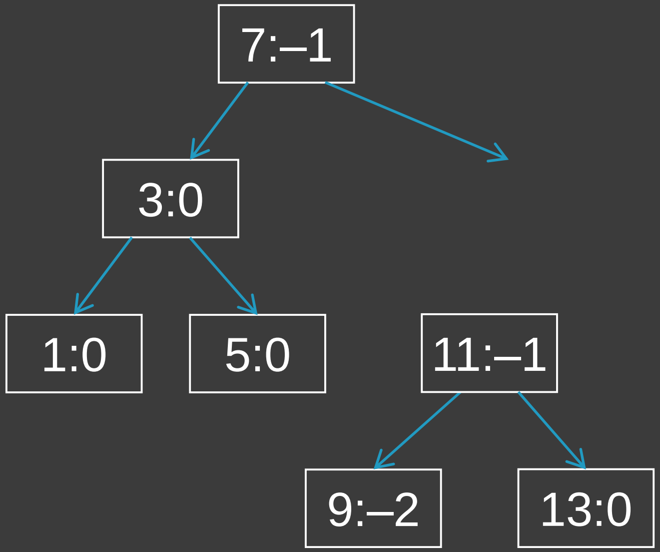


Keep Inserting



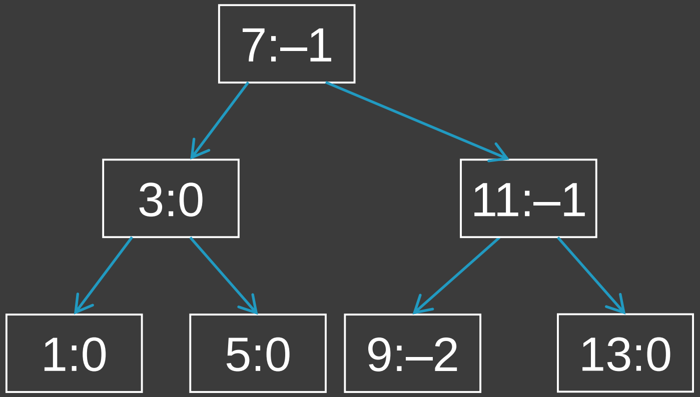
Rotate below most recent ancestor of inserted node that has a post-insertion BF of +2/–2

Keep Inserting



Re-attach rotated sub-tree

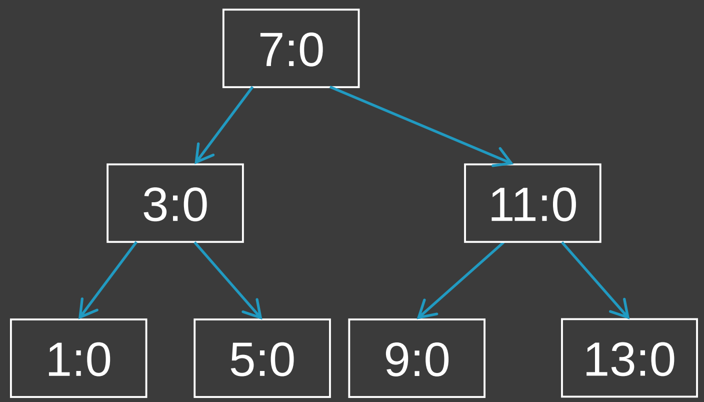
Keep Inserting



Re-attach rotated sub-tree

Adjust Balance Factors

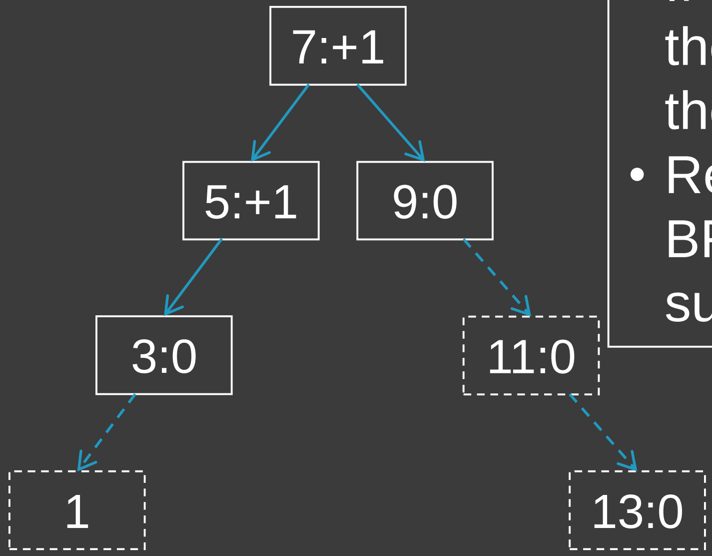
Keep Inserting



More On Insertions & Rotations

- When we insert an item, and some node's BF goes to <u>+2</u>, we have two possibilities. The insertion was made in:
  - the <u>*left*</u> subtree of this node's <u>left</u> subtree (LL)
  - the <u>*right*</u> subtree of this node's <u>left</u> subtree (LR)
- When we insert an item, and some node's BF goes to <u>-2</u>, we have two possibilities. The insertion was made in:
  - the <u>*left*</u> subtree of this node's <u>right</u> subtree (RL)
  - the <u>*right*</u> subtree of this node's <u>right</u> subtree (RR)

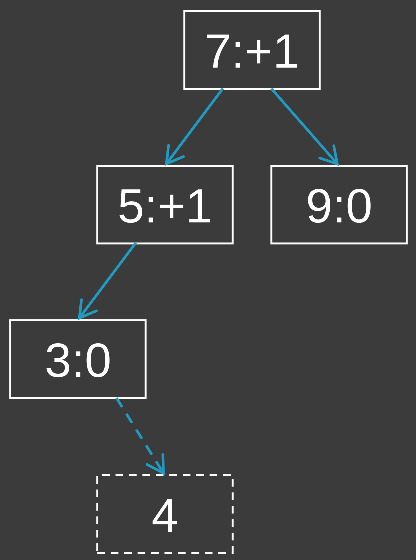
Previous Example Revisited



- If we insert 1 or 2 (a left child of 3), then the BF at 3 goes to +1, and the BF at 5 goes to +2
- Relative to 5 (the node with the +2 BF), this is an insertion in the left subtree of the left subtree (LL)

- LL and RR are symmetrical
- If we insert 11 and then insert 13, that would be RR (the BFs at 9 and 11 would go to -2 and -1, respectively).

Previous Example Revisited



- If we insert 4 (the *right* child of 3), then the BF at 3 goes to -1, and the BF at 5 goes to +2
- Relative to 5 (the node with the new +2 BF), this is an insertion in the right subtree of the left subtree (LR [go *L*eft, then go *R*ight])

Previous Example Revisited

- If we insert 10 (the *left* child of 11), then the BF at 11 goes to +1, and the BF at 9 goes to -2
- Relative to 9 (the node with the new -2 BF), this is an insertion in the left subtree of the right subtree (RL [go *R*ight, then go *L*eft)

Rotations

- These four rotations (LL, RR, LR, and RL) are all we need.
- The rotations we've done on previous slides have all been either LL or RR – the easy ones
- LR and RL are more complicated – each has three sub-cases to check for

A New Example

- Let's insert the names of the months (JAN through DEC) into an AVL tree (all comparisons are done alphabetically). We will see all four rotations in this exercise.
- For the sake of this exercise, we assume the names arrive in this order:

  MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP.
- In each case, we will show the tree AFTER the insertion and BF adjustments

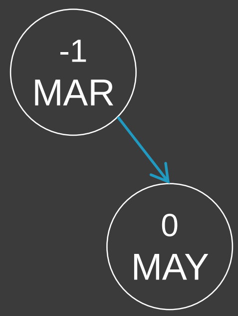
Insert MAR (Into Empty Tree)

~~MAR~~
MAY
NOV
AUG
APR
JAN
DEC
JUL
FEB
JUN
OCT
SEP

No rotation
Needed

Insert MAY

~~MAR~~
~~MAY~~
NOV
AUG
APR
JAN
DEC
JUL
FEB
JUN
OCT
SEP



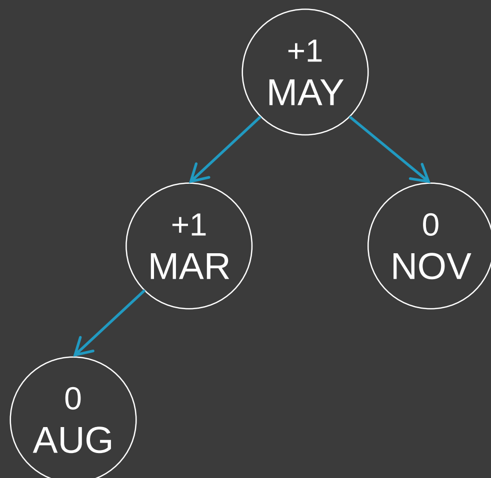
No rotation Needed

Insert NOV

~~MAR~~
~~MAY~~
~~NOV~~
AUG
APR
JAN
DEC
JUL
FEB
JUN
OCT
SEP

Insert AUG

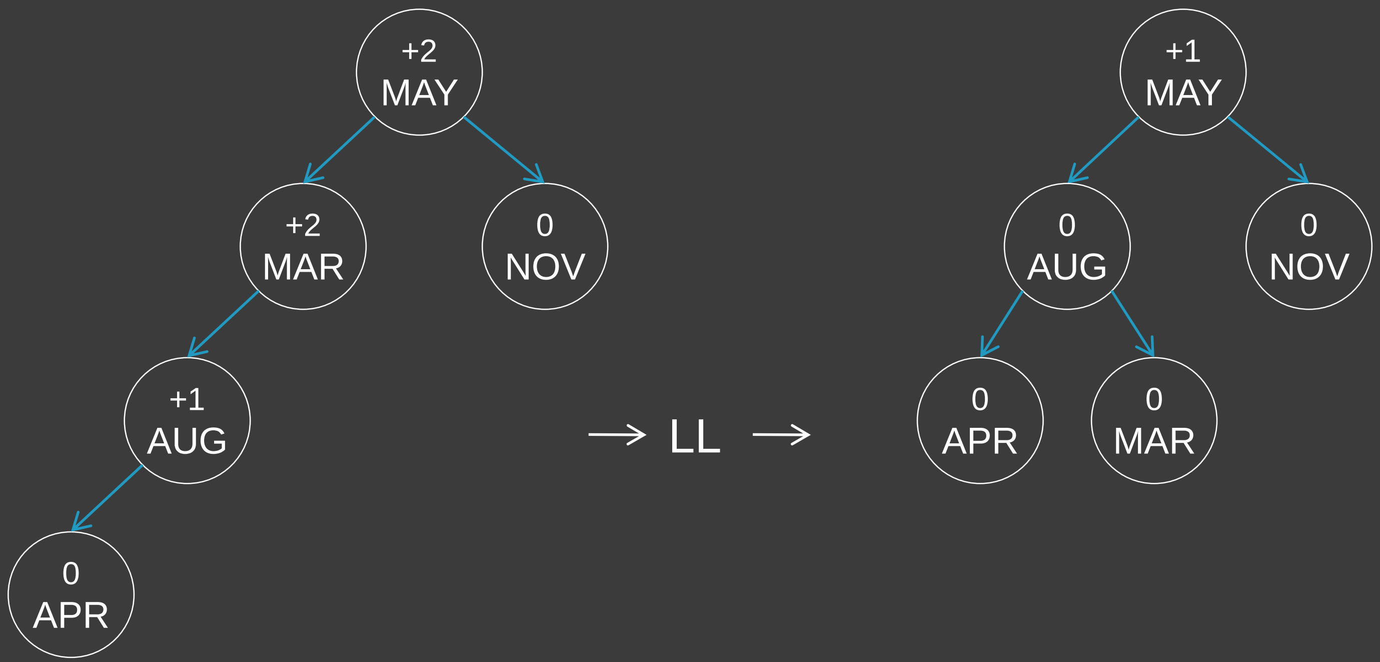
~~MAR~~
~~MAY~~
~~NOV~~
~~AUG~~
APR
JAN
DEC
JUL
FEB
JUN
OCT
SEP



No rotation Needed

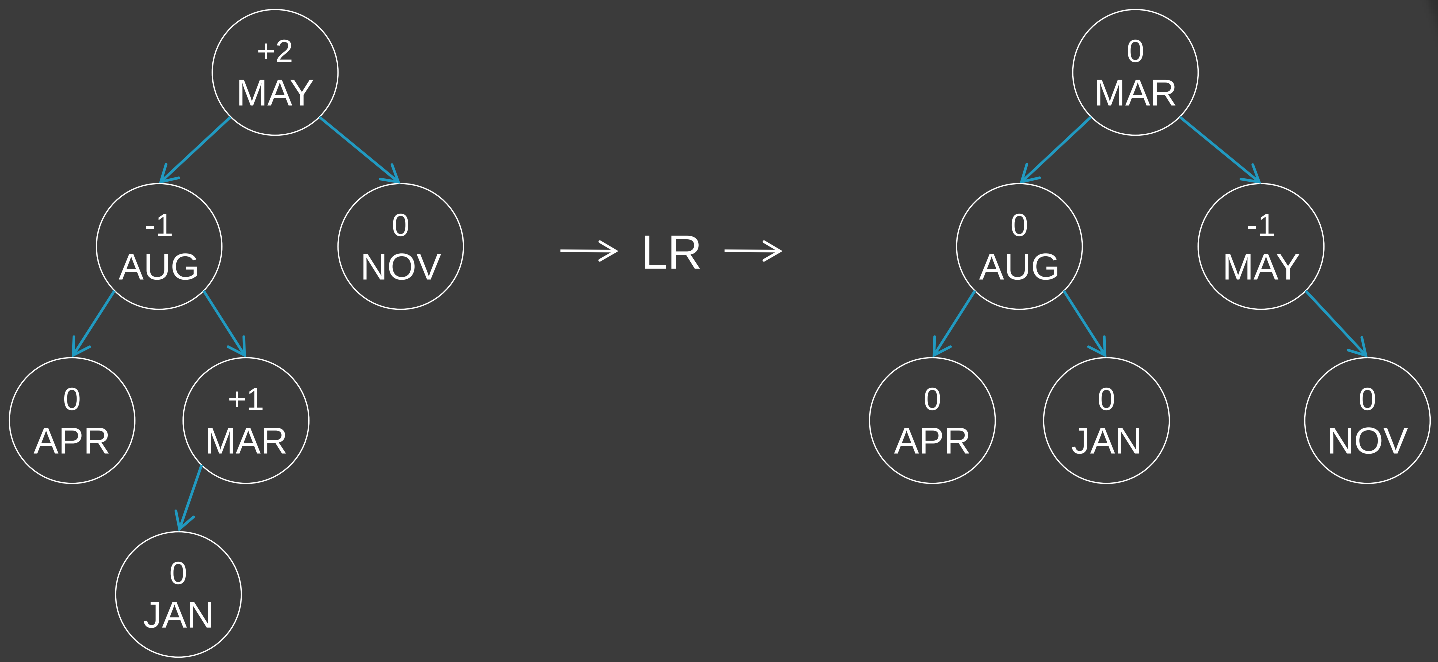
Insert APR

~~MAR~~
~~MAY~~
~~NOV~~
~~AUG~~
~~APR~~
JAN
DEC
JUL
FEB
JUN
OCT
SEP



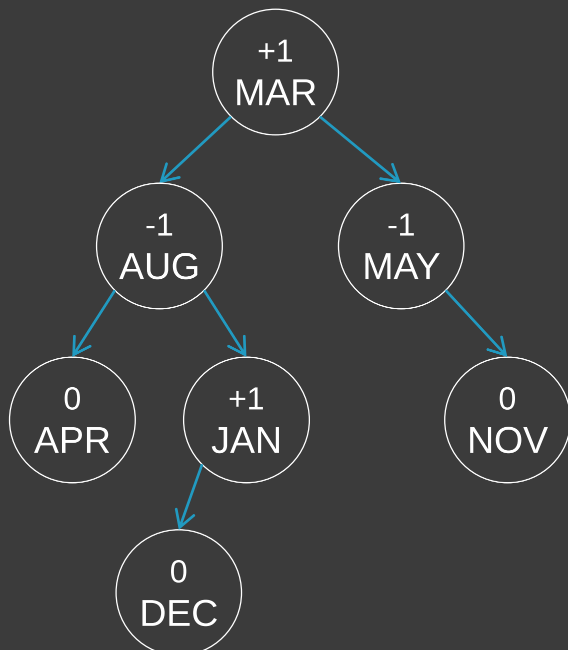
Insert JAN

~~MAR~~
~~MAY~~
~~NOV~~
~~AUG~~
~~APR~~
~~JAN~~
DEC
JUL
FEB
JUN
OCT
SEP



Insert DEC

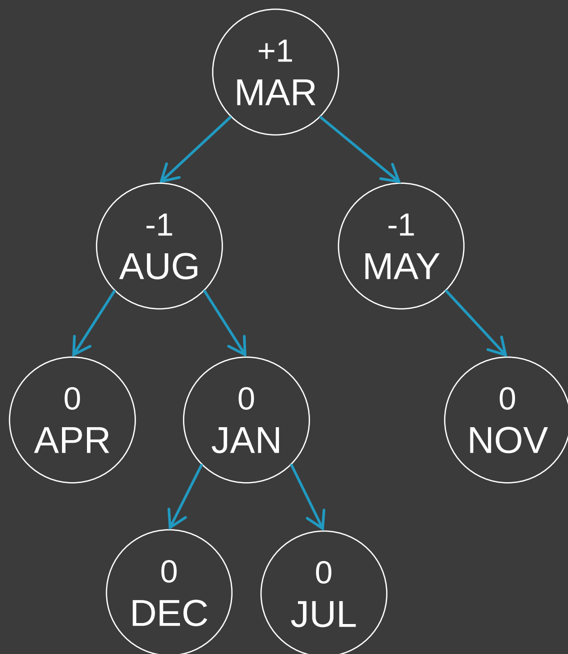
~~MAR~~
~~MAY~~
~~NOV~~
~~AUG~~
~~APR~~
~~JAN~~
~~DEC~~
JUL
FEB
JUN
OCT
SEP



No rotation
Needed

Insert JUL

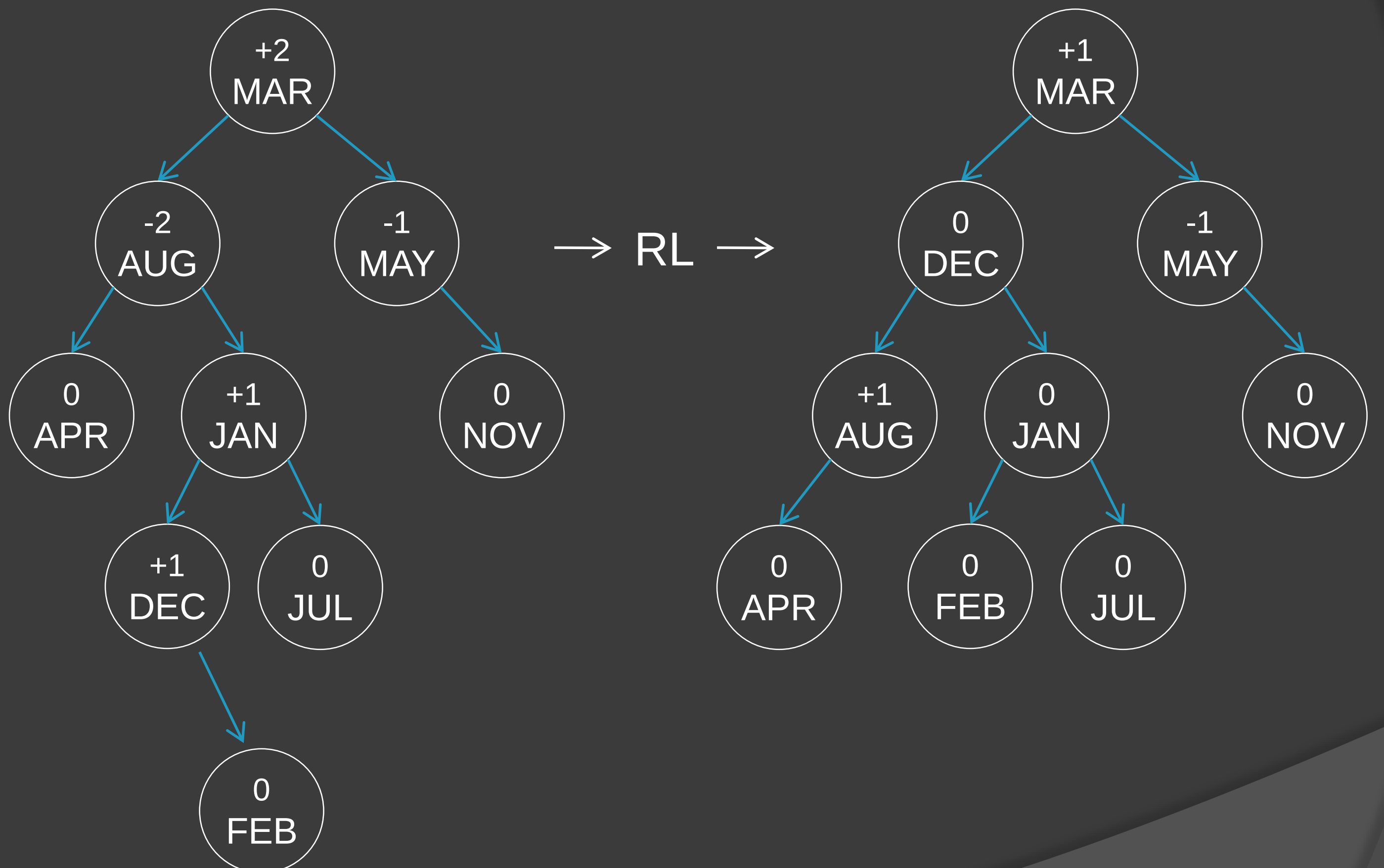
~~MAR~~
~~MAY~~
~~NOV~~
~~AUG~~
~~APR~~
~~JAN~~
~~DEC~~
~~JUL~~
FEB
JUN
OCT
SEP



No rotation Needed

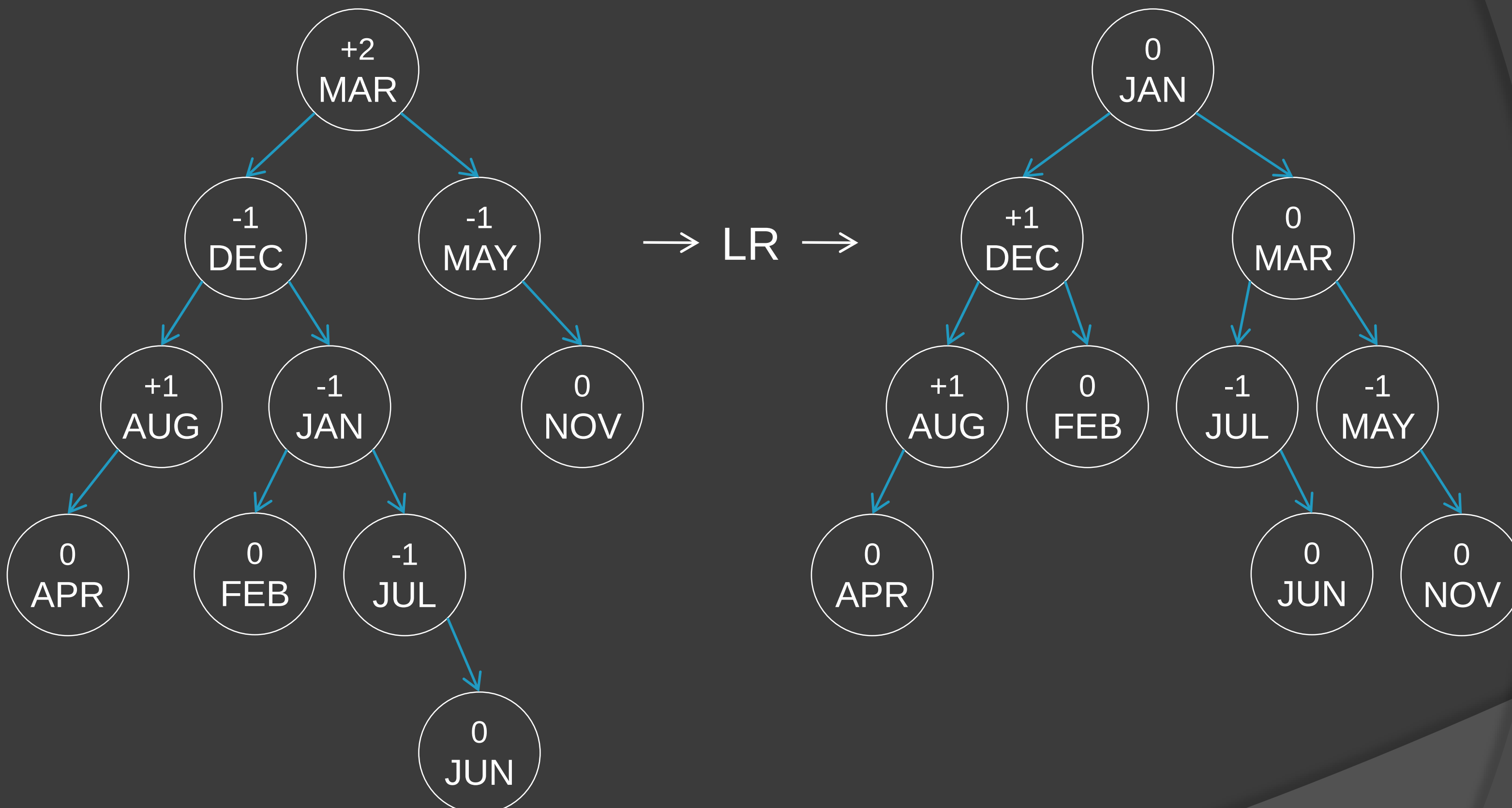
Insert FEB

~~MAR~~
~~MAY~~
~~NOV~~
~~AUG~~
~~APR~~
~~JAN~~
~~DEC~~
~~JUL~~
~~FEB~~
JUN
OCT
SEP



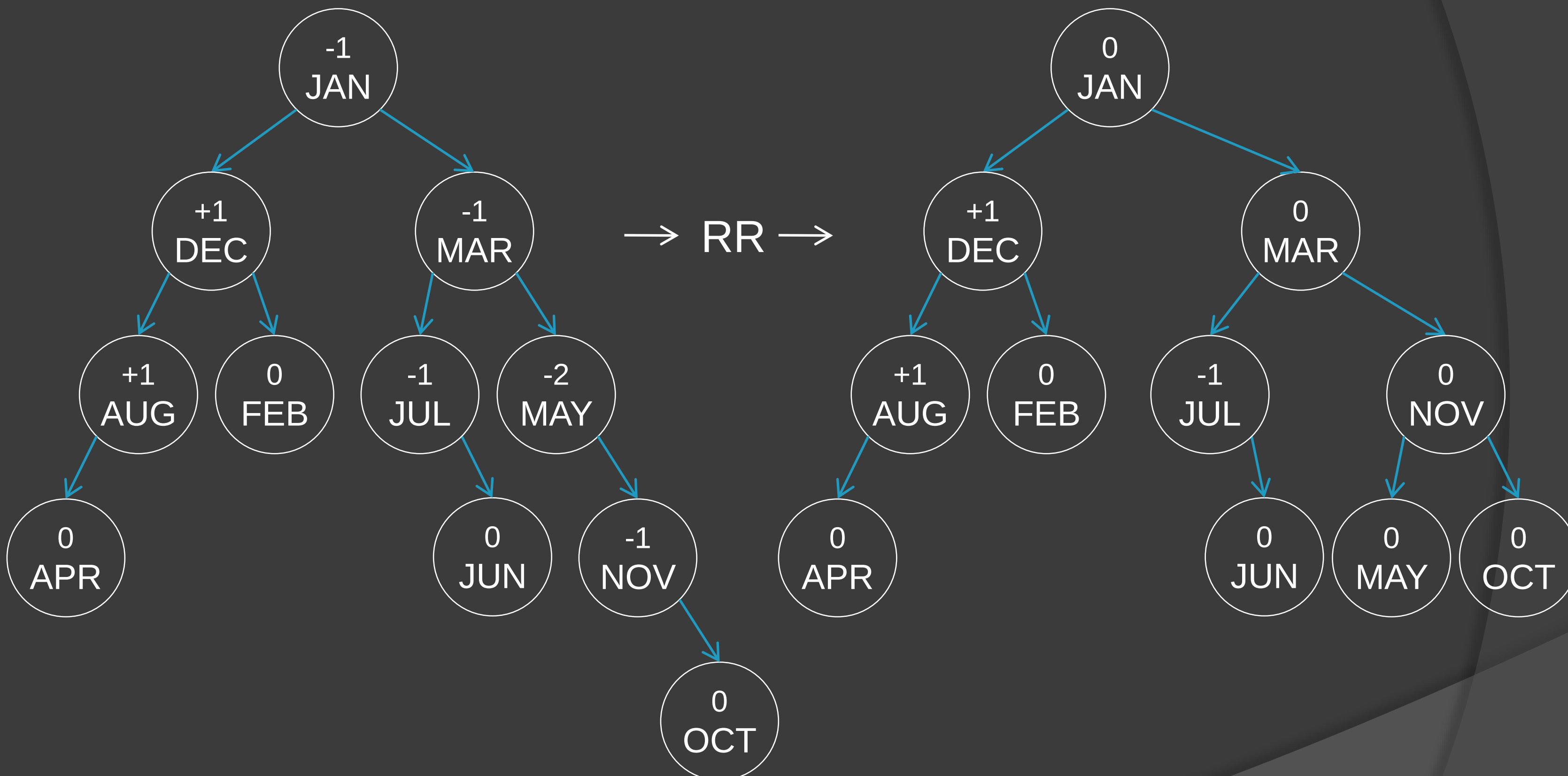
Insert JUN

~~MAR~~
~~MAY~~
~~NOV~~
~~AUG~~
~~APR~~
~~JAN~~
~~DEC~~
~~JUL~~
~~FEB~~
~~JUN~~
OCT
SEP



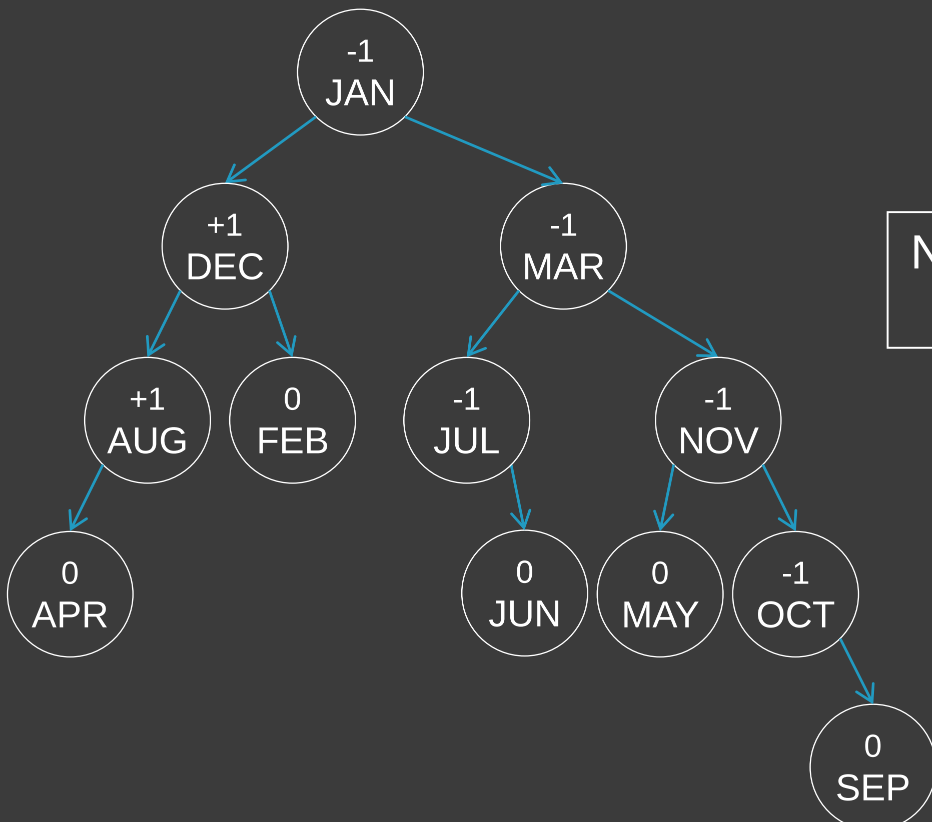
Insert OCT

~~MAR~~
~~MAY~~
~~NOV~~
~~AUG~~
~~APR~~
~~JAN~~
~~DEC~~
~~JUL~~
~~FEB~~
~~JUN~~
~~OCT~~
SEP



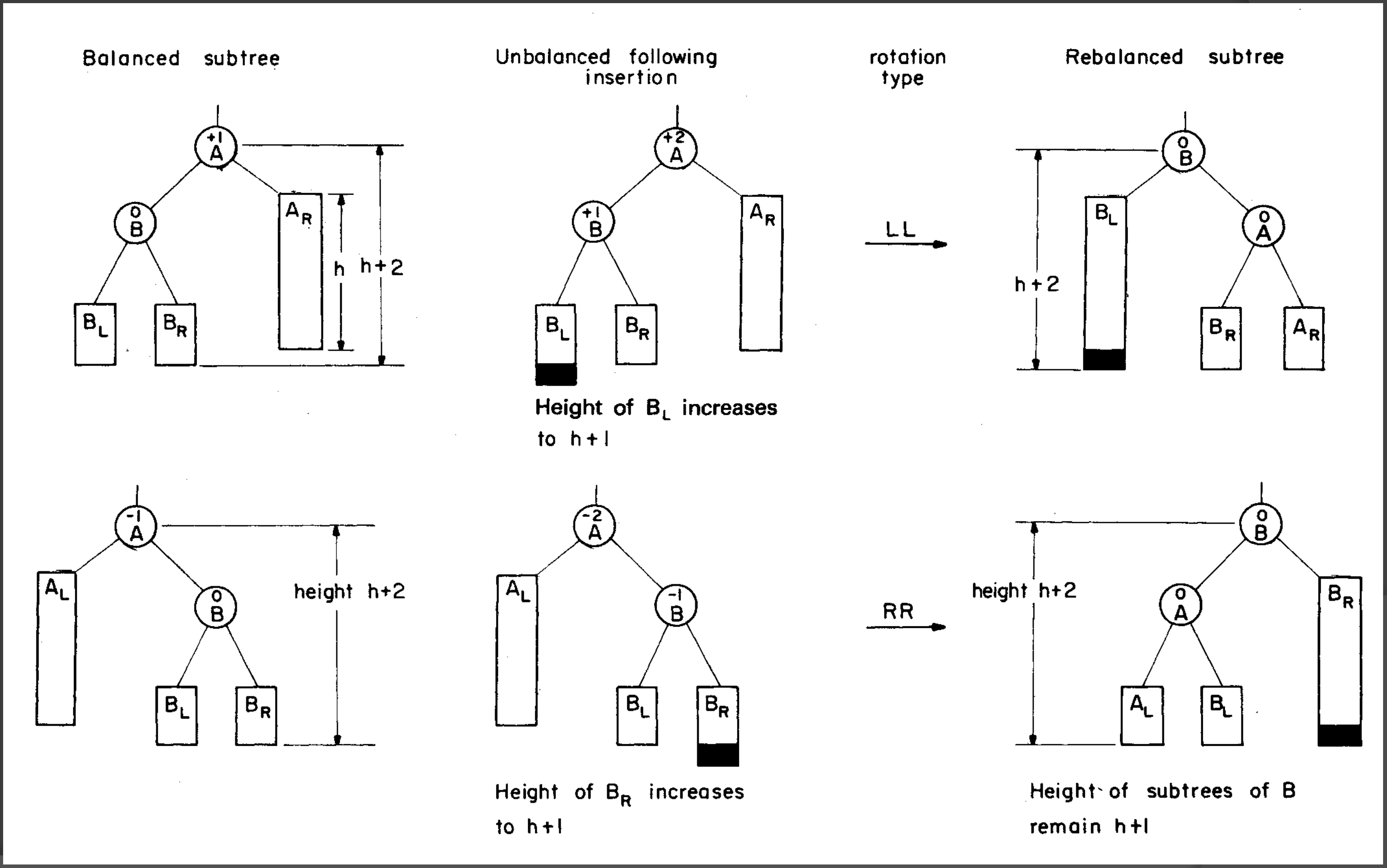
Insert SEP

~~MAR~~
~~MAY~~
~~NOV~~
~~AUG~~
~~APR~~
~~JAN~~
~~DEC~~
~~JUL~~
~~FEB~~
~~JUN~~
~~OCT~~
~~SEP~~

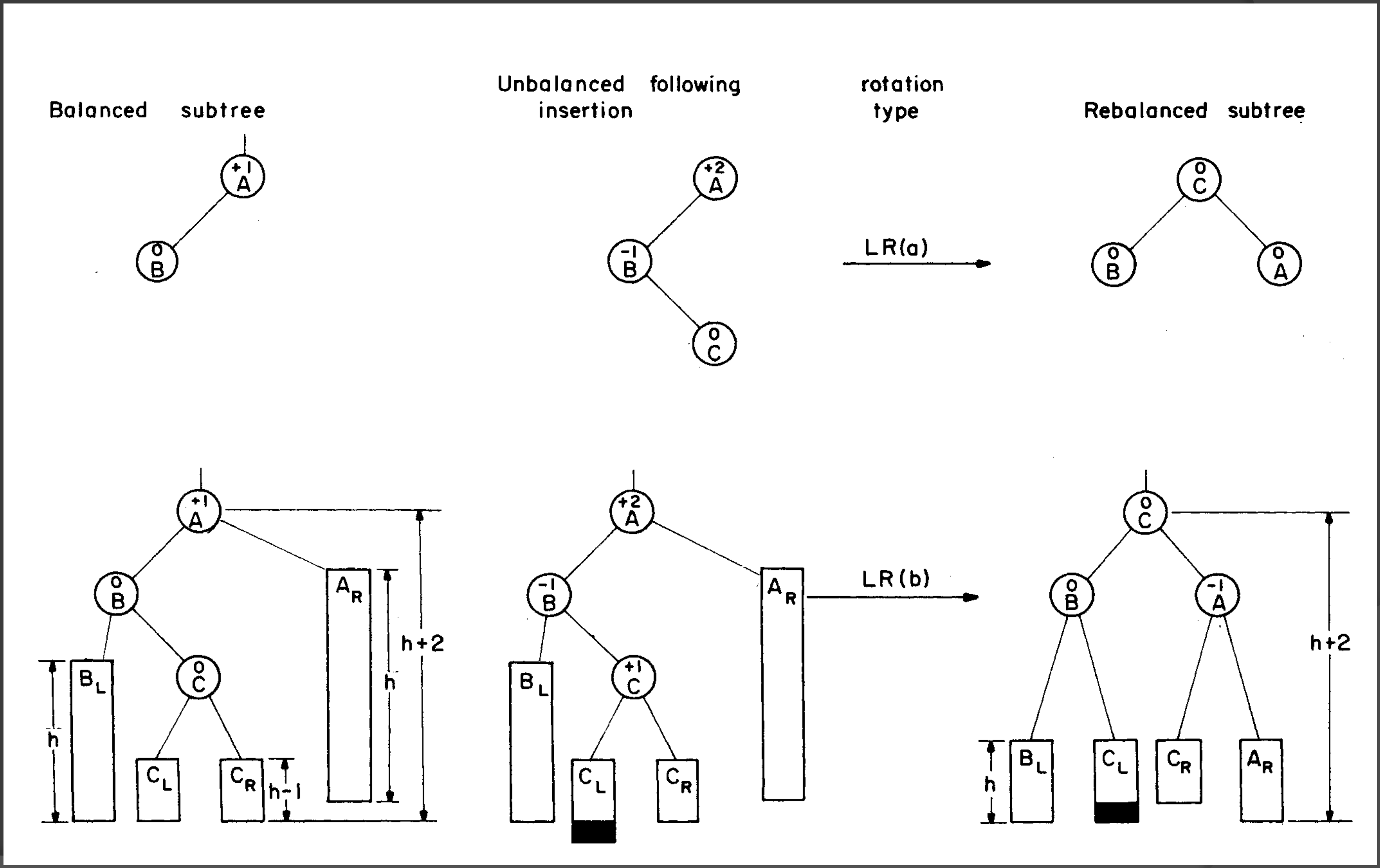


No rotation Needed

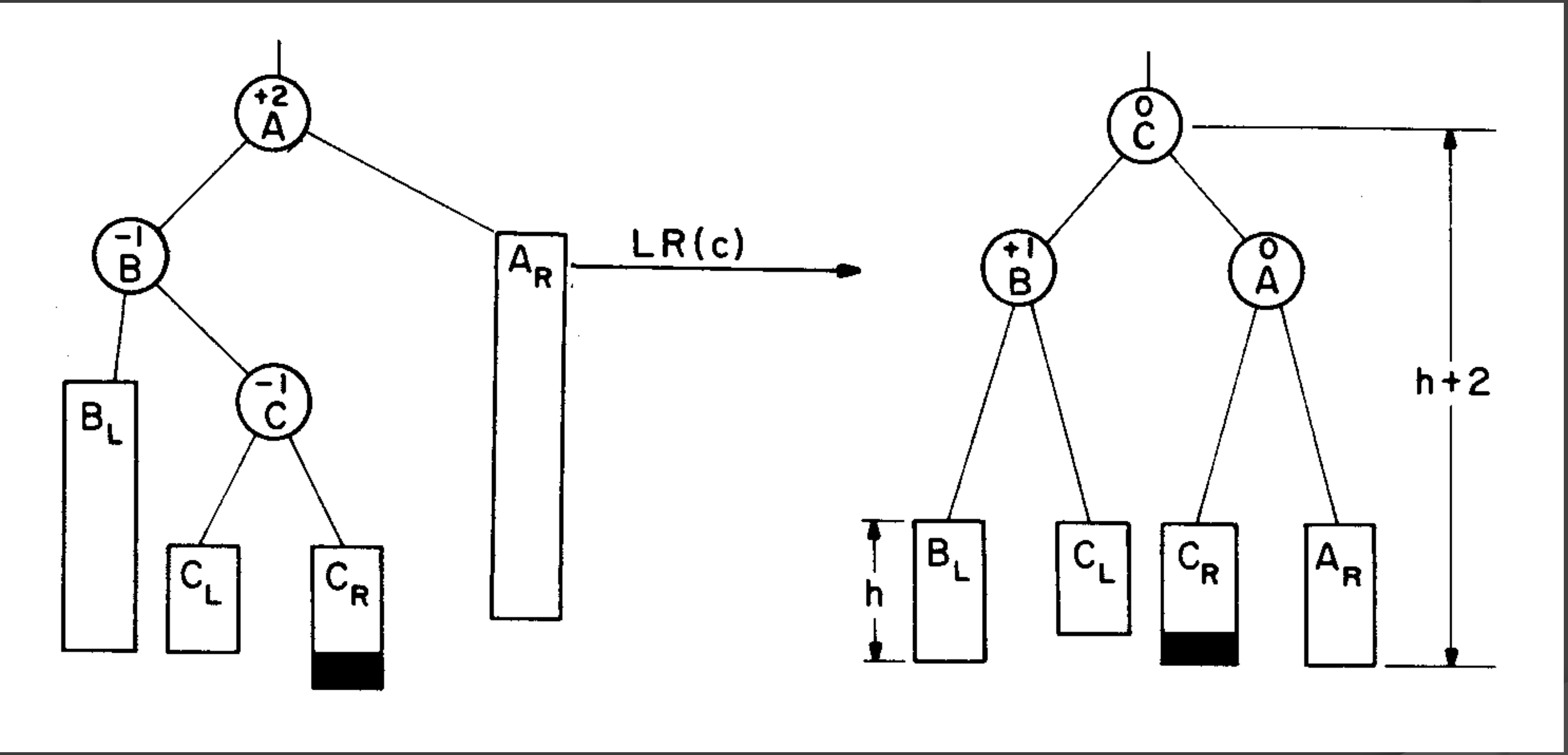
The Rotations, Schematically (1)



The Rotations, Schematically (2)



The Rotations, Schematically (3)



The AVL Insert Process (Pseudocode)

SUPER high-level:

1. If tree is empty, make new root node (BF=0)
2. Otherwise, scan through tree, looking for where what we want to insert belongs. If we find it (already in the tree), exit.
3. Make a new node, and make this new node the appropriate child of its new parent.
4. Adjust Balance Factors above new node
5. If imbalance occurs, rotate to rebalance.

The AVL Insert Process (In C): 1/8

```
node *root=NULL;
void AVL_Insert(data X) {
  node *Y;            // The new node we insert
  node *A, *B, *F;    // see below
  node *C, *CL, *CR // for description
  int  d;             // Used to adjust BFs
  if (root==NULL) // Empty tree? Make root!
  {
    Y = new node;
    Y->data=X;
    Y->LCH =Y->RCH=NULL;
    Y->BF  =0;
    root = Y;
    return; }
```

The AVL Insert Process (In C): 2/8

```
// Locate insertion point for X.
// P scans through the tree
// Q is P's parent (Q lags behind P)
//    New Node Y will be a child of Q
// A is the last parent above Y w/BF=+/-1
// F is A's parent (F lags behind A)
//
F = Q = NULL;
A = P = root;

while (P != NULL) {
  if (P->BF !=0) {A=P; F=Q;}
  if (X == P->data) return;
  if (X < P->data) {Q=P; P=P->LCH;}
             else {Q=P; P=P->RCH;}
}
```

The AVL Insert Process (In C): 3/8

```
// At this point, P is NULL, but Q points
// at the last node where X belongs
// (either as Q's LCH or RCH)
//
Y = new node;
Y->data = X;
Y->LCH  = NULL; // New nodes are always
Y->RCH  = NULL; // inserted as leaves
Y->BF   = 0;    // Leaves always balanced


// Will Y be Q's new left or right child?
if (X < Q->data) Q->LCH = Y;
            else Q->RCH = Y;
```

The AVL Insert Process (In C): 4/8

```
// Adjust BFs from A to Q. Since A was the
// LAST ancestor with a BF of +/- 1, ALL
// nodes BETWEEN A and Q must have a BF of
// 0, and will, therefore, BECOME +/-1.
// If X is inserted in the LEFT subtree
// of A, then d=+1 (d=-1 means we inserted
// X in the RIGHT subtree of A.

if (X > A->data) {P=A->RCH; B=P; d=-1;}
            else {P=A->LCH; B=P; d=+1;}

while (P != Y) {
  if (X>P->data) {P->BF=-1; P=P->RCH;}
            else {P->BF=+1; P=P->LCH;}
}
```

The AVL Insert Process (In C): 5/8

```
// Now we check the BF at A and see if we
// just BALANCED the tree, IMBALANCED the
// tree, or if it is still BALANCED ENOUGH.

if (A->BF==0) {A->BF=d; // Tree is still
               return;} // mostly balanced

if (A->BF+d==0) {A->BF=0; // Insertion put
                 return;} // tree INTO bal

// If we didn't take either of the two
// returns immediately above, then the
// tree is IMBALANCED.  We have to
// determine the required rotation type
```

The AVL Insert Process (In C): 6/8

```
if (d==+1) { // left imbalance.  LL or LR?
  if (B->BF==+1) {           // LL rotation
    // Change the child pointers at A and B
    // To reflect the rotation.  Adjust
    // the balance factors at A and B
    // <<< LEFT FOR YOU TO WRITE >>>
  }
  else {                     // LR Rotation
    // Adjust the child pointers of nodes
    // A, B, and C to reflect the new
    // structure after the rotation
    // <<< LEFT FOR YOU TO WRITE,  >>>
    // <<< BUT HERE'S A HEAD START >>>
    C  = B->RCH; // C is B's right child
    CL = C->LCH; // CL and CR are C's left
    CR = C->RCH; //    and right children
```

The AVL Insert Process (In C): 7/8

```
      switch (C->BF) {
        // Set the new balance factors at
        // A and B based on the BF at C
        // Note: There are 3 cases
        // <<< LEFT FOR YOU TO WRITE >>>
      }

      C->BF=0; B=C;
    } end of else (LR Rotation)
  } // end of "if (d=+1)"
  else { // d=-1.  This is a right imbalance
         // (RR or RL).  THAT code goes here.
         // <<< LEFT FOR YOU TO WRITE >>>
  }
```

The AVL Insert Process (In C): 8/8

```
  // The subtree with root B has been
  // reblanaced, and is the new subtree
  // of F.  The original subtree of F had
  // root A.

  // did we rebalance the root?
  if (F == NULL) {root=B; return;}

  // otherwise, we rebalanced whatever was
  // the child (left or right) of F.
  if (A == F->LCH) {F->LCH=B; return;}
  if (A == F->RCH) {F->RCH=B; return;}
  cout << “We should never be here\n”;

} // End of AVL_Insert
```

Properties of AVL Trees

- Time to insert: $O(h)$
- Time to search: $O(h)$
- Time to find successor / predecessor: $O(h)$
- Time to delete $O(h)$
- Time to find min / max: $O(h)$

- So, what is h?
- AVL guarantees $h \leq \sim 1.44 \lg n$
  - Where n is the number of nodes in the tree.
  - The 1.44 comes from Fibbonaci theory

? Questions ?

? Questions ?