# Data Structures and Algorithms for Engineers

## Module 6: Trees

## Lecture 3: Height Balanced Trees: AVL Trees

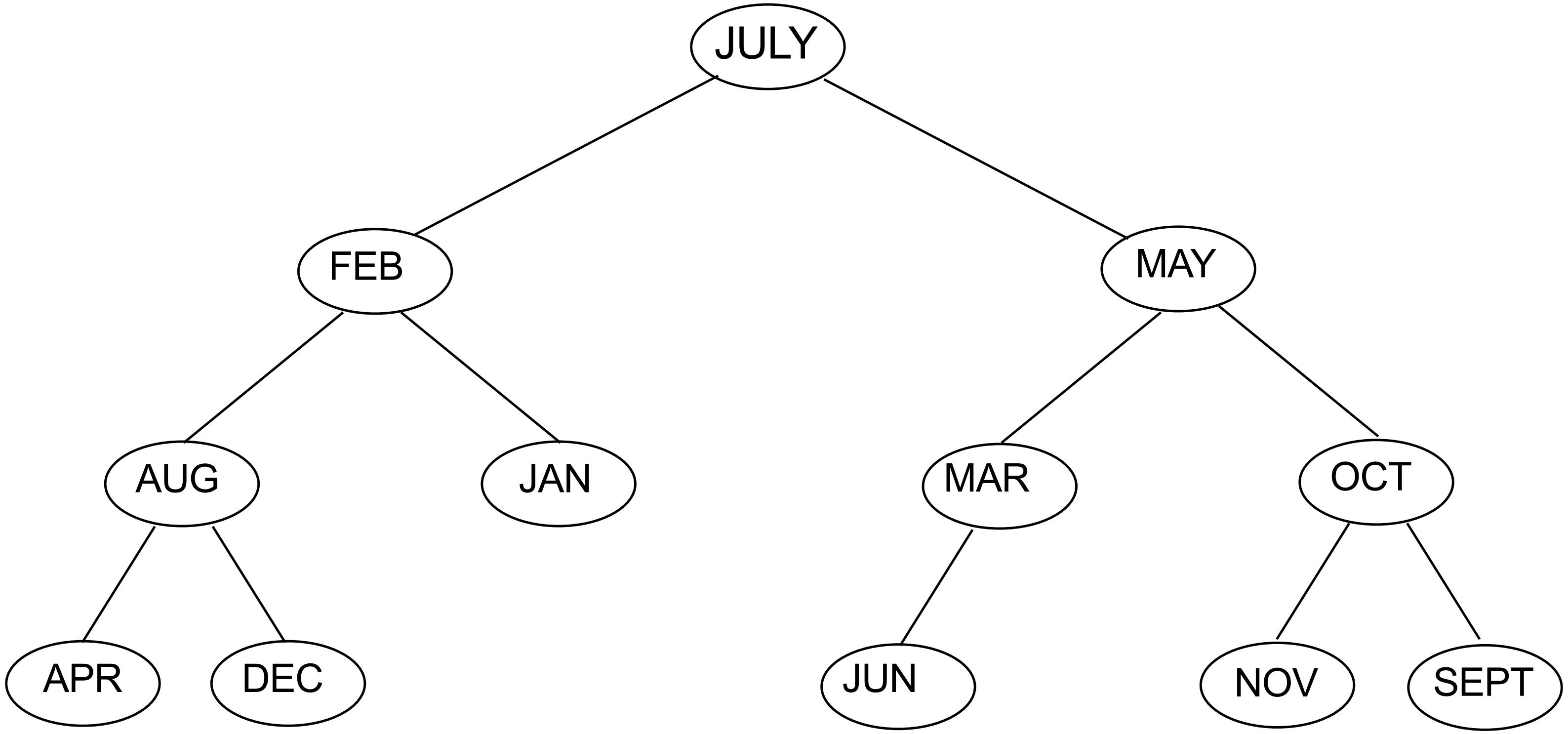
David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

# AVL Trees

- We know from our study of Binary Search Trees (BST) that the average search and insertion time is $O(\log n)$
  - If there are $n$ nodes in the binary tree it will take, on average, $log_2 n$ comparisons/probes to find a particular node (or find out that it isn't there)
- However, this is only true if the tree is 'balanced'
  - Such as occurs when the elements are inserted in random order

# AVL Trees

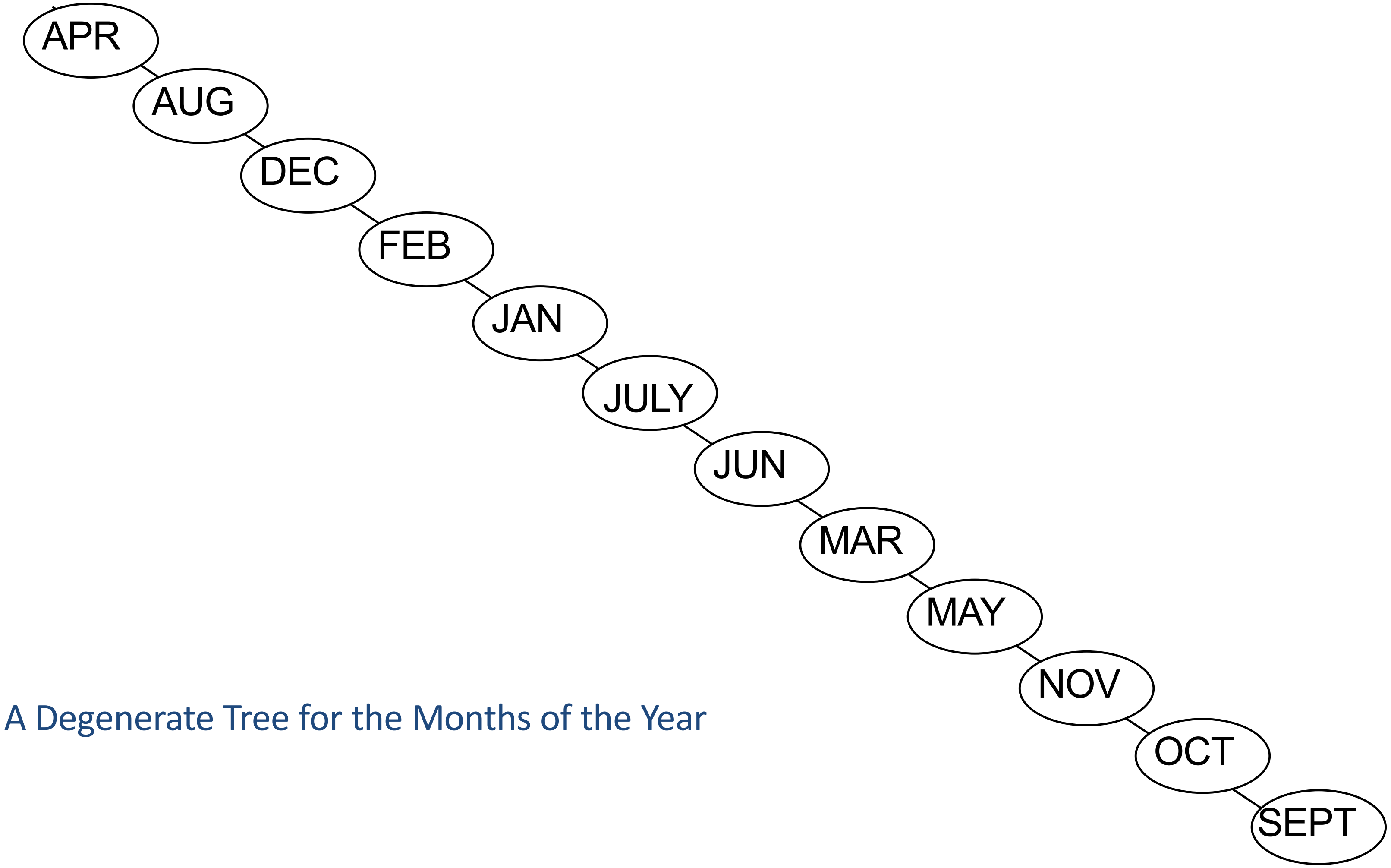



A Balanced Tree for the Months of the Year

# AVL Trees

- However, if the elements are inserted in lexicographic order (i.e., in sorted order) then the tree degenerates into a skinny tree

# AVL Trees




A Degenerate Tree for the Months of the Year

# AVL Trees

- If we are dealing with a dynamic tree …

- nodes are being inserted and deleted over time

  - For example, directory of files
  - For example, index of university students

- we may need to restructure - balance - the tree so that we keep it
  - Fat
  - Full
  - Complete

# AVL Trees

- Adelson-Velskii and Landis in 1962 introduced a binary tree structure that is balanced with respect to the heights of its subtrees

- Insertions (and deletions) are made such that the tree
  - starts off
  - and remains

- Height-Balanced

# AVL Trees

- Definition of AVL Tree

- An empty tree is height-balanced

- If T is a non-empty binary tree with left and right sub-trees $T_1$ and $T_2$, then T is height-balanced iff (if and only if)

  - $T_1$ and $T_2$ are height-balanced, and

  - $|height(T_1) - height(T_2)| \leq$

# AVL Trees

- So, every sub-tree in a height-balanced tree is also height-balanced

# Recall: Binary Tree Terminology

- The height of T is defined recursively as

  $0$ if T is empty and

  $1 + \max(\text{height}(T_1), \text{height}(T_2))$ otherwise,
  where $T_1$ and $T_2$ are the subtrees of the root

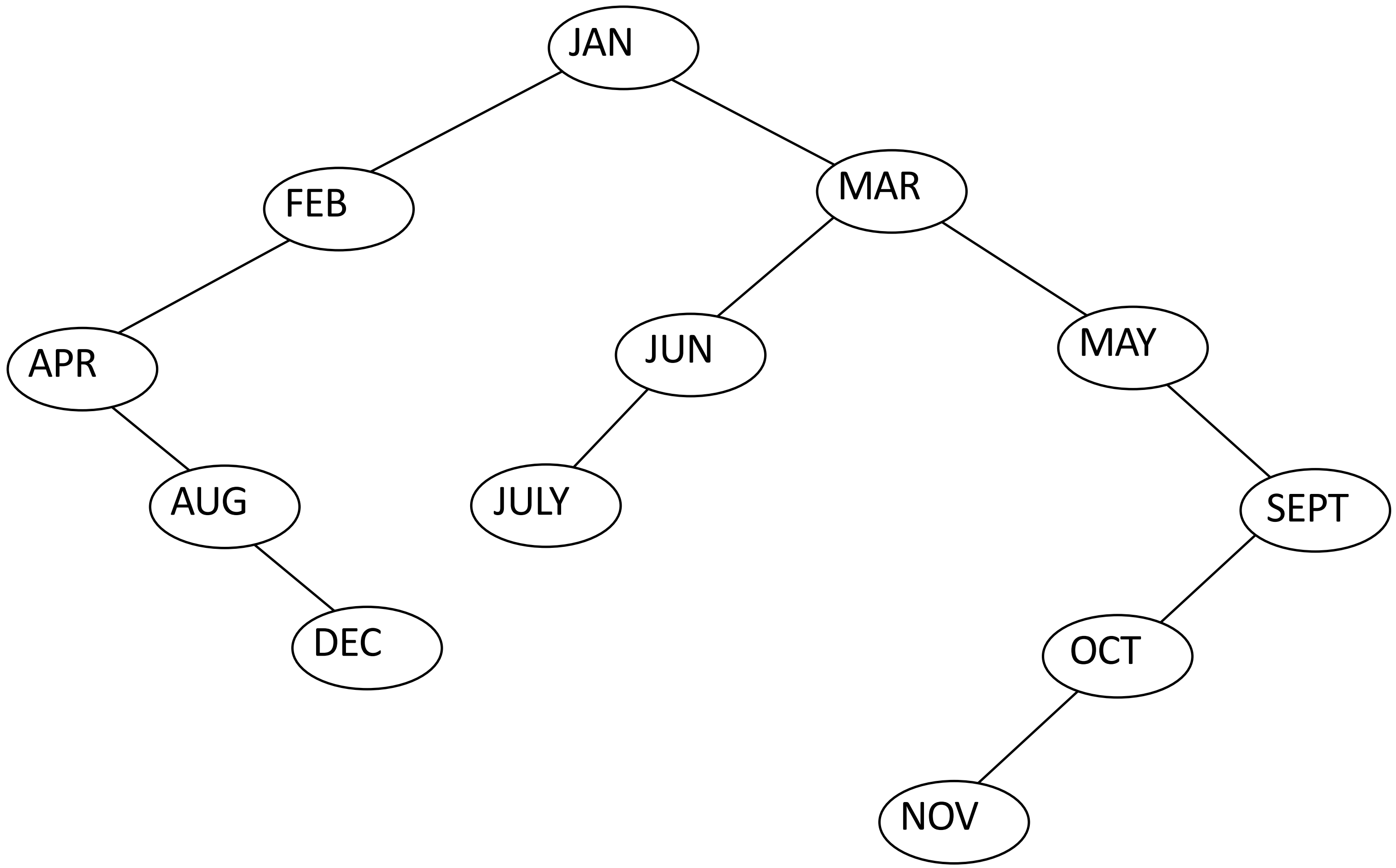
- The height of a tree is the length of a longest chain of descendents

# Recall: Binary Tree Terminology

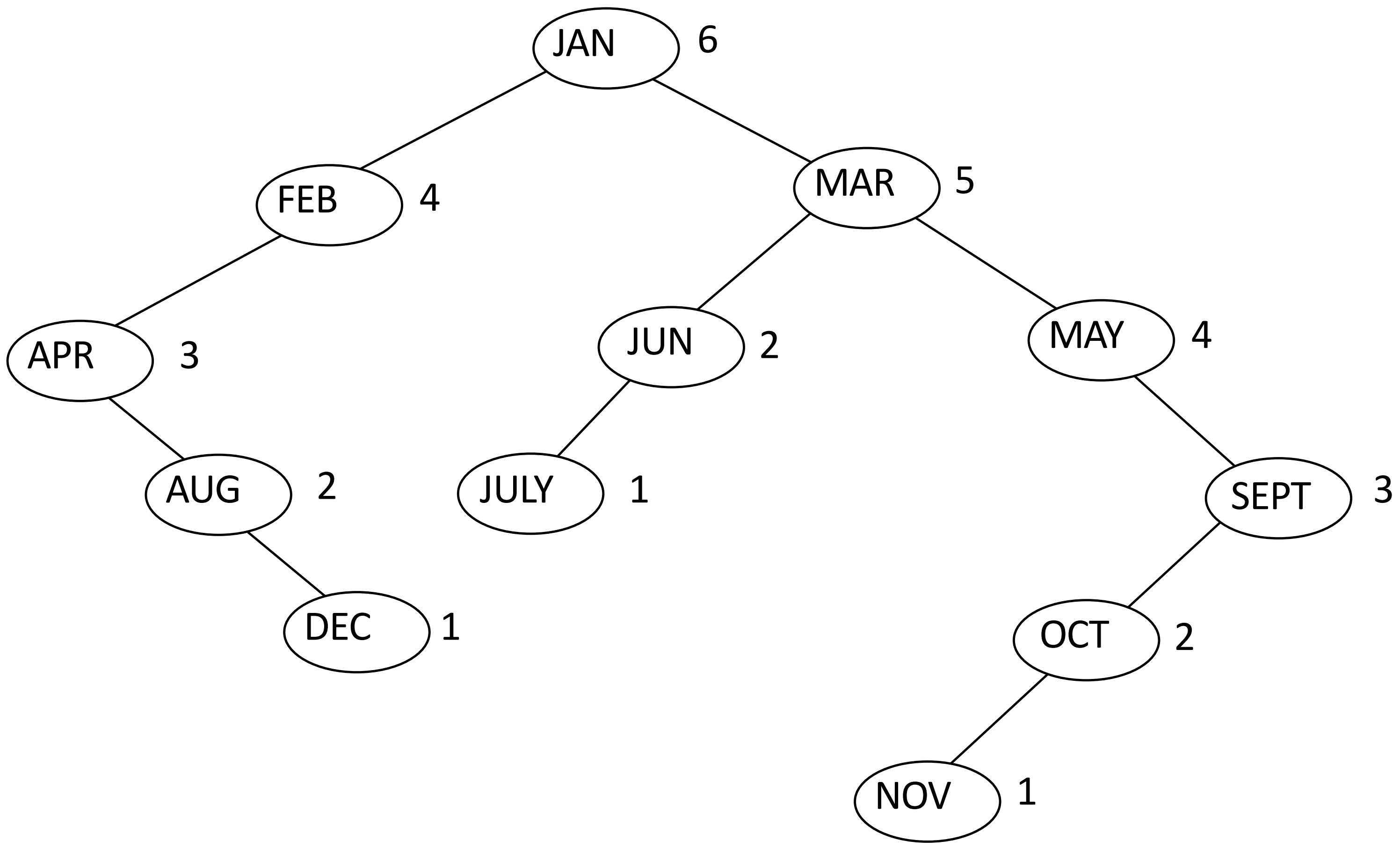
- Height Numbering
  - Number all external nodes $0$
  - Number each internal node to be one more than the maximum of the numbers of its children
  - Then the number of the root is the height of T

- The height of a node u in T is the height of the subtree rooted at u

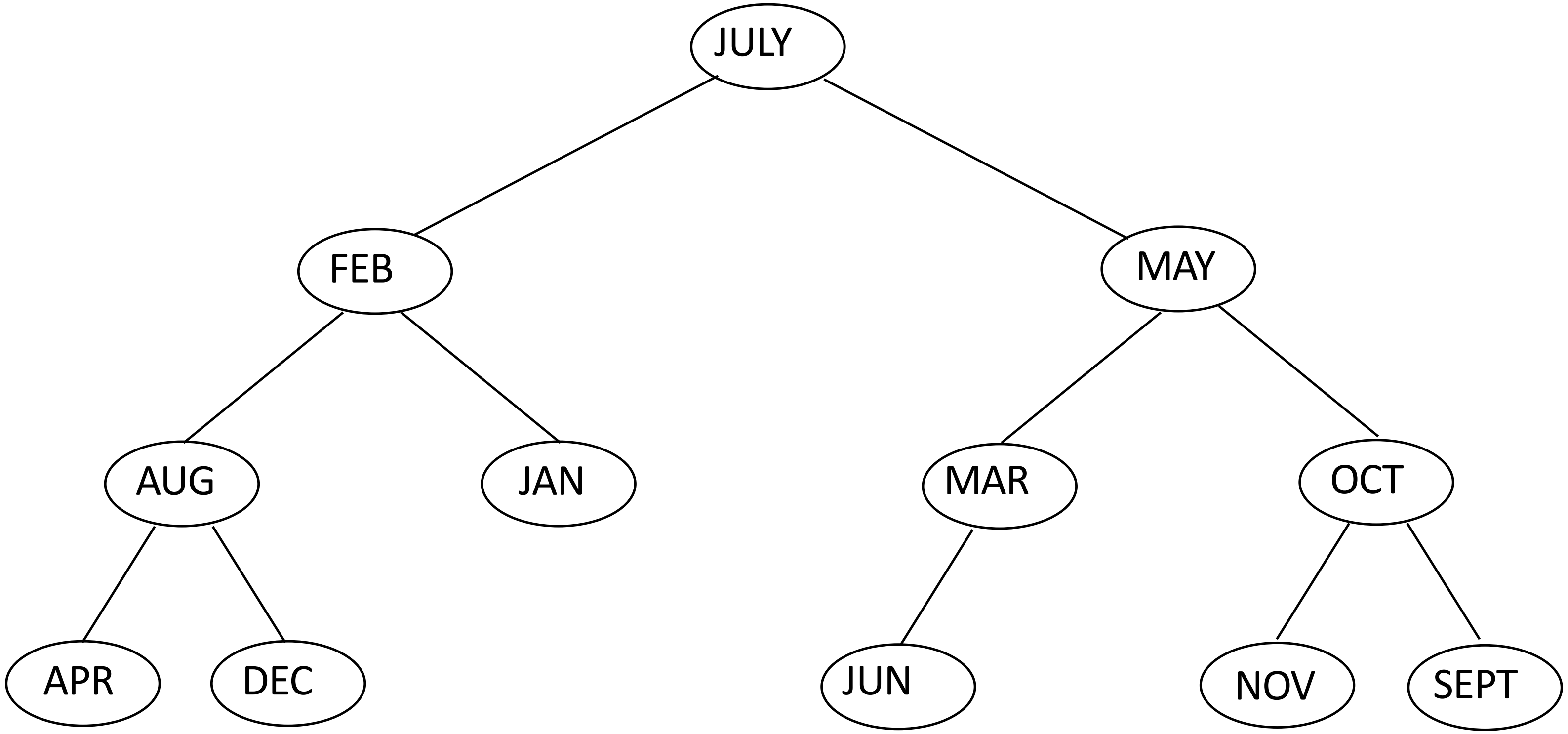
# AVL Trees



# AVL Trees

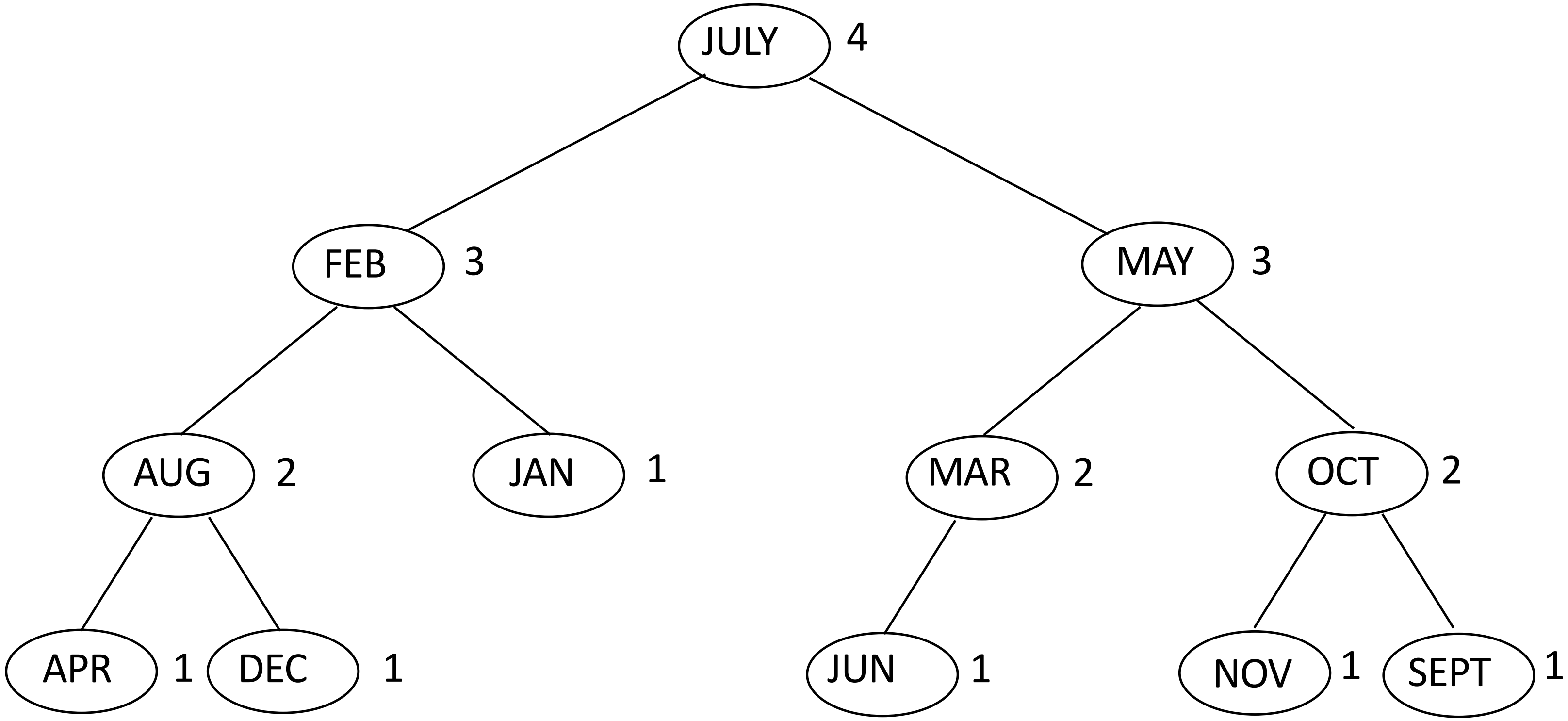

JAN
6
FEB
4
MAR
5
APR
3
JUN
2
MAY
4
AUG
2
JULY
1
SEPT
3
DEC
1
OCT
2
NOV
1

# AVL Trees




A Balanced Tree for the Months of the Year

# AVL Trees




A Balanced Tree for the Months of the Year

# AVL Trees

- Let's construct a height-balanced tree

- Order of insertions:

  March, May, November, August, April, January, December, July, February, June, October, September

- Before we do, we need a definition of a balance factor

# AVL Trees

- Balance Factor BF(T) of a node T in a binary tree is defined to be

  height($T_1$) - height($T_2$)

  where $T_1$ and $T_2$ are the left and right subtrees of T

- For any node T in an AVL tree

  BF(T) = -1, 0, +1

# AVL Trees

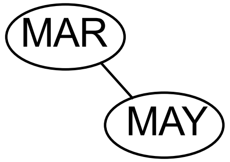
- All re-balancing operations are carried out with respect to the closest ancestor of the new node having balance factor +2 or -2

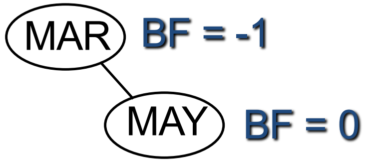
- There are 4 types of re-balancing operations (called rotations)

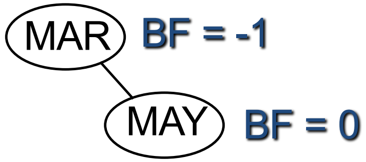
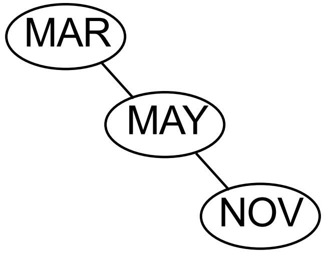
  - RR
  - LL (symmetric with RR)
  - RL
  - LR (symmetric with RL)

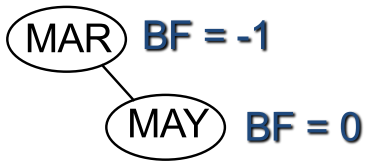
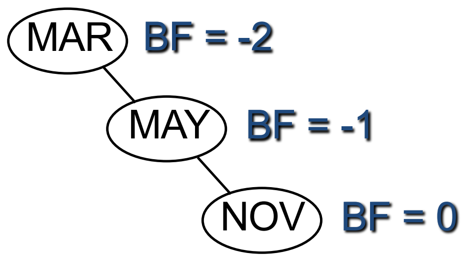
| New Identifier | After Insertion | After Rebalancing |
|---|---|---|
| MARCH |  | |

| New Identifier | After Insertion | After Rebalancing |
| --- | --- | --- |
| MARCH |  | NO REBALANCING NEEDED |

| New Identifier | After Insertion | After Rebalancing |
| --- | --- | --- |
| MARCHÂ |  | NO REBALANCING NEEDED |
| MAY |  | |

| New Identifier | After Insertion | After Rebalancing |
| --- | --- | --- |
| MARCH |   | NO REBALANCING NEEDED |
| MAY |   | NO REBALANCING NEEDED |

| New Identifier | After Insertion | After Rebalancing |
| --- | --- | --- |
| MARCH |   | NO REBALANCING NEEDED |
| MAY |   | NO REBALANCING NEEDED |
| NOVEMBER |   | |

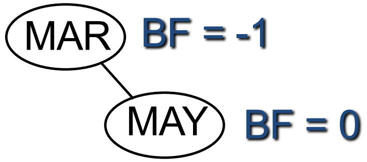
| New Identifier | After Insertion | After Rebalancing |
| --- | --- | --- |
| MARCH |   | NO REBALANCING NEEDED |
| MAY |   | NO REBALANCING NEEDED |
| NOVEMBER |   | |

New Identifier | After Insertion | After Rebalancing

MARCH

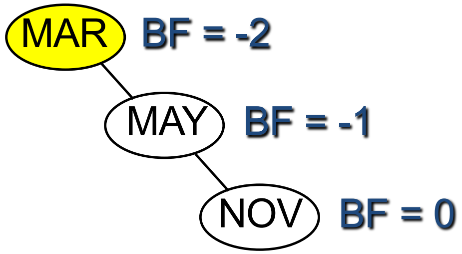

NO REBALANCING NEEDED

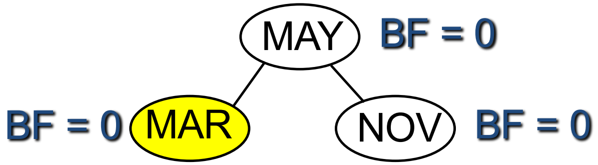
MAY




NO REBALANCING NEEDED

NOVEMBER



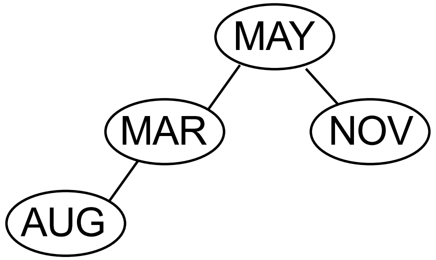

RR rebalancing

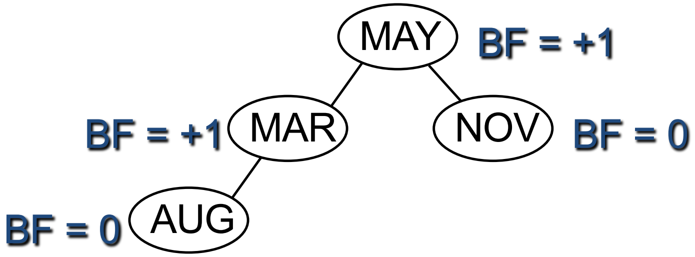
New Identifier

After Insertion

After Rebalancing

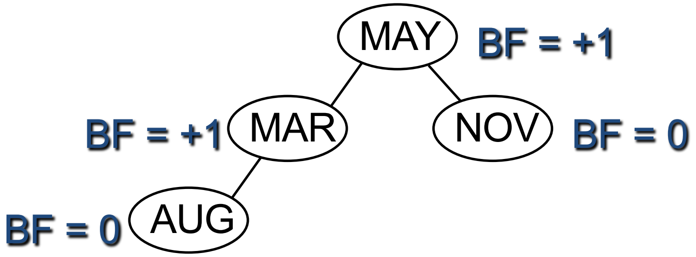
AUGUST



| New Identifier | After Insertion | After Rebalancing |
|---|---|---|
| AUGUST |  | NO REBALANCING NEEDED |

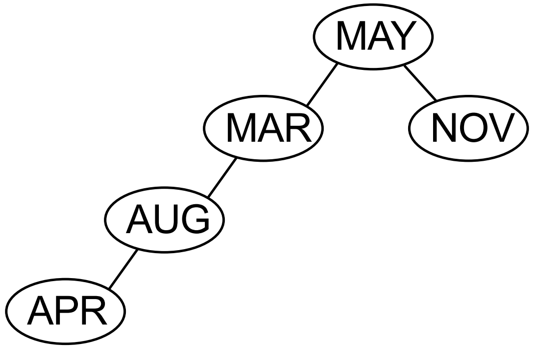
| New Identifier | After Insertion | After Rebalancing |
| --- | --- | --- |

AUGUST



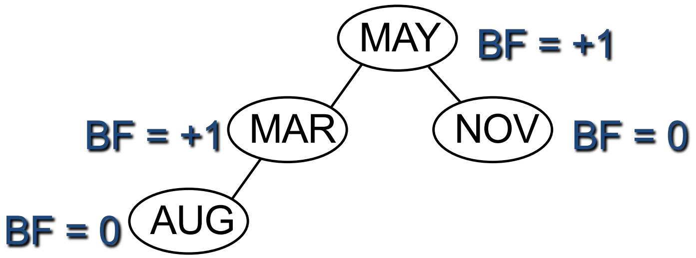

NO REBALANCING NEEDED

APRIL



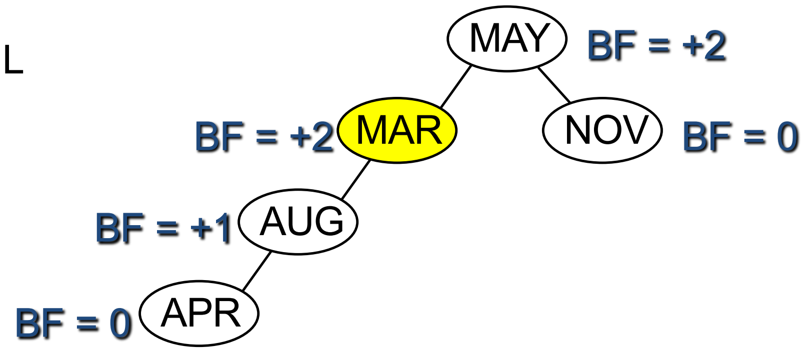
New Identifier | After Insertion | After Rebalancing

AUGUST



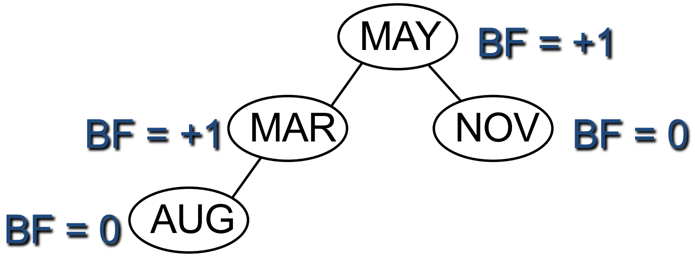

NO REBALANCING NEEDED

APRIL



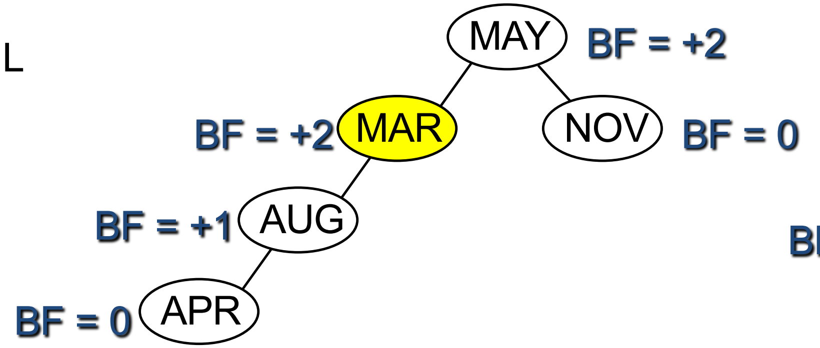
New Identifier | After Insertion | After Rebalancing

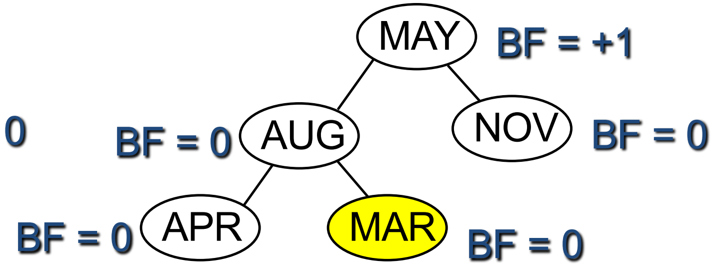
AUGUST




NO REBALANCING NEEDED

APRIL



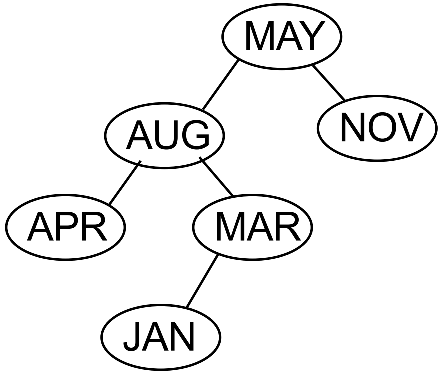

LL rebalancing

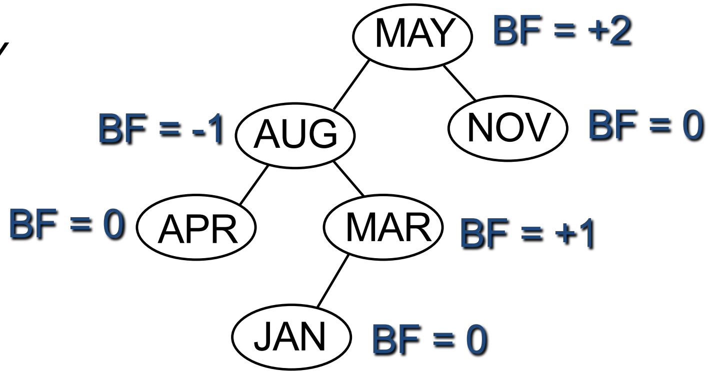
New Identifier

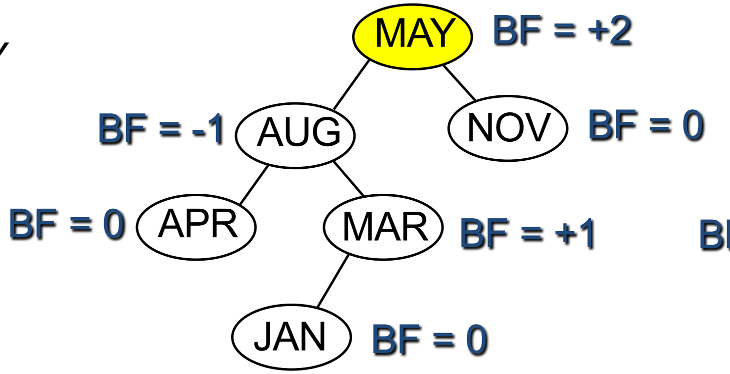
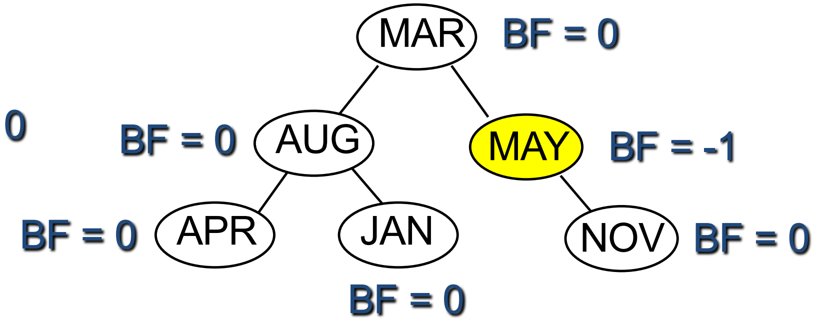
After Insertion

After Rebalancing

JANUARY



| New Identifier | After Insertion | After Rebalancing |
|---|---|---|
| JANUARY |  | |

| New Identifier | After Insertion | After Rebalancing |
| --- | --- | --- |
| JANUARY |  |  |





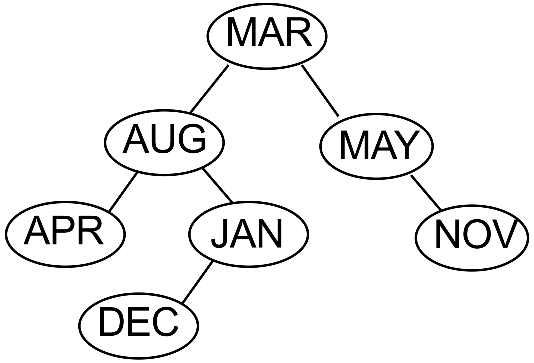
LR rebalancing

New
Identifier

After
Insertion

After
Rebalancing

DECEMBER

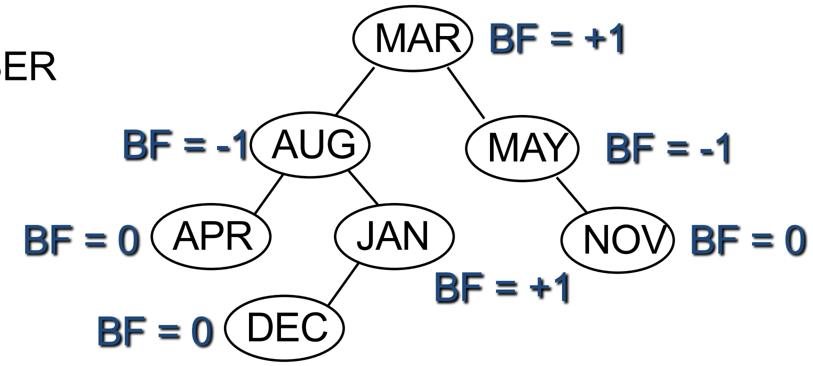


New Identifier

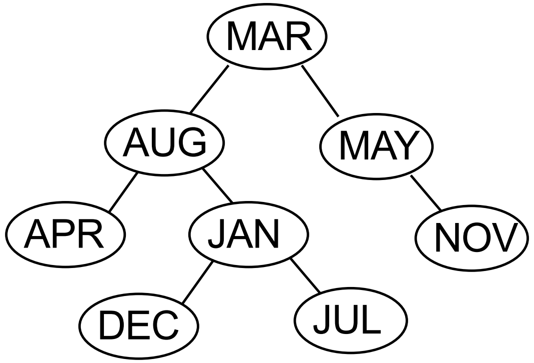
After Insertion

After Rebalancing

DECEMBER

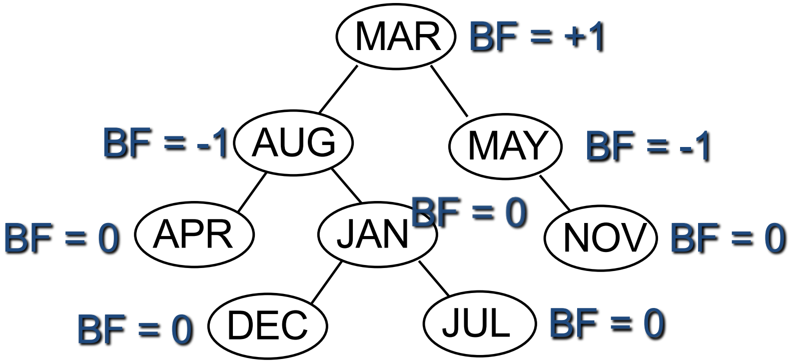



NO REBALANCING NEEDED

| New Identifier | After Insertion | After Rebalancing |
| --- | --- | --- |
| JULY |  | |

New Identifier | After Insertion | After Rebalancing

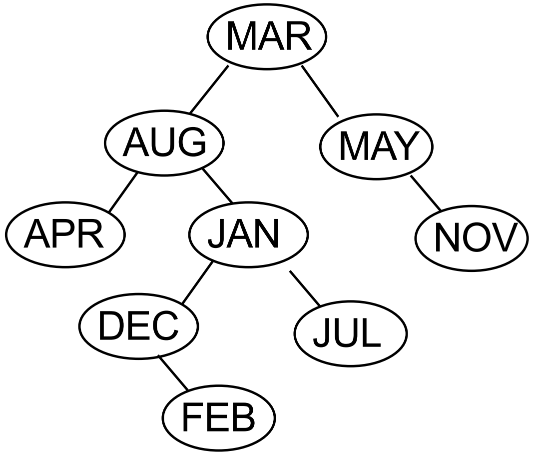
JULY




NO REBALANCING NEEDED

New Identifier | After Insertion | After Rebalancing

FEBRUARY

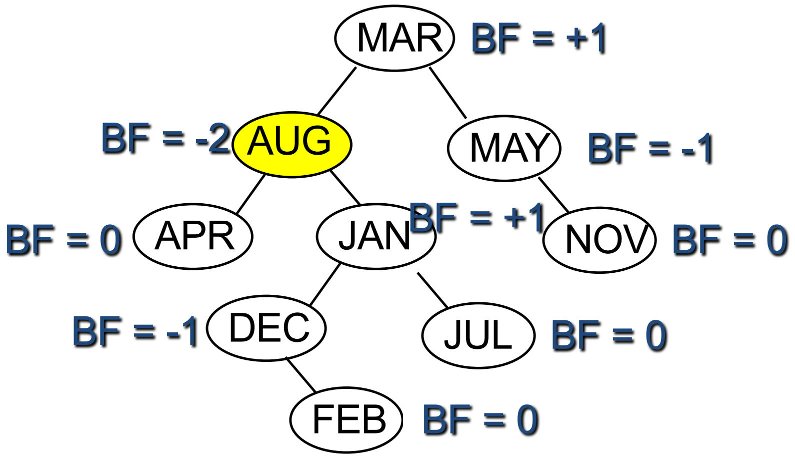


New Identifier

After Insertion

After Rebalancing

FEBRUARY

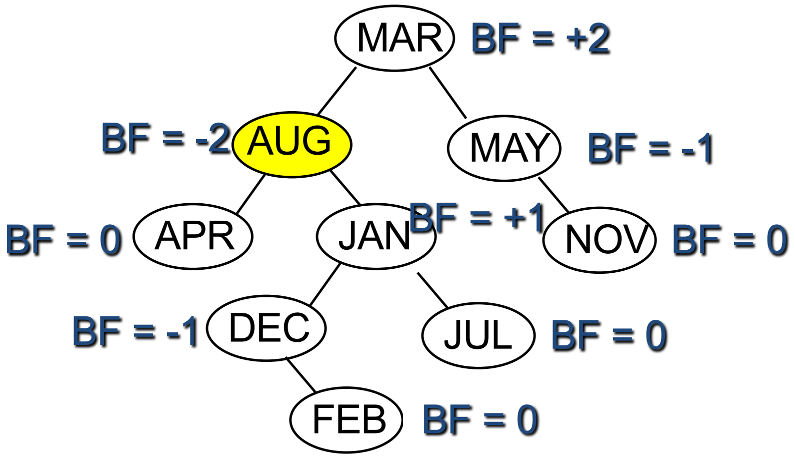


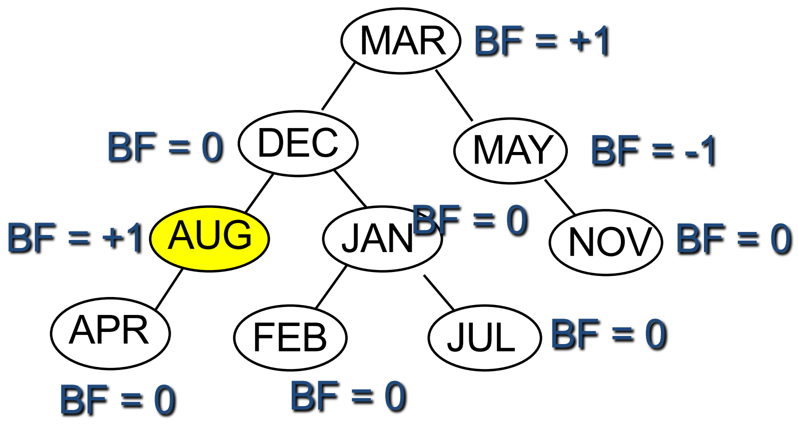
New Identifier

After Insertion

After Rebalancing

FEBRUARY



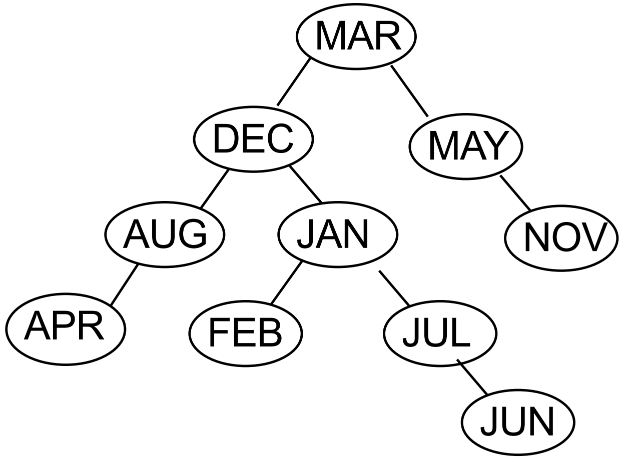

RL rebalancing

New Identifier

After Insertion

After Rebalancing

JUNE

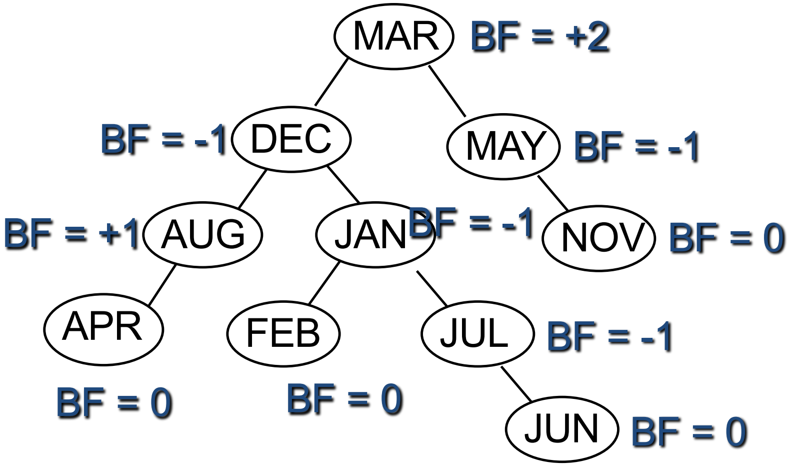


New Identifier

After Insertion

After Rebalancing

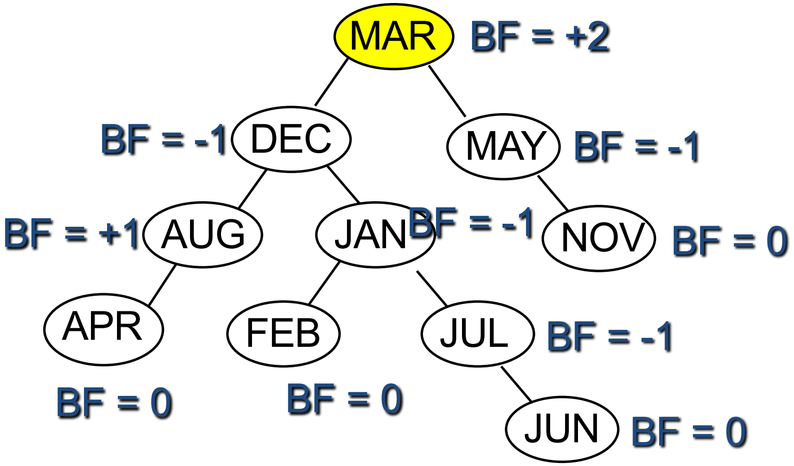
JUNE



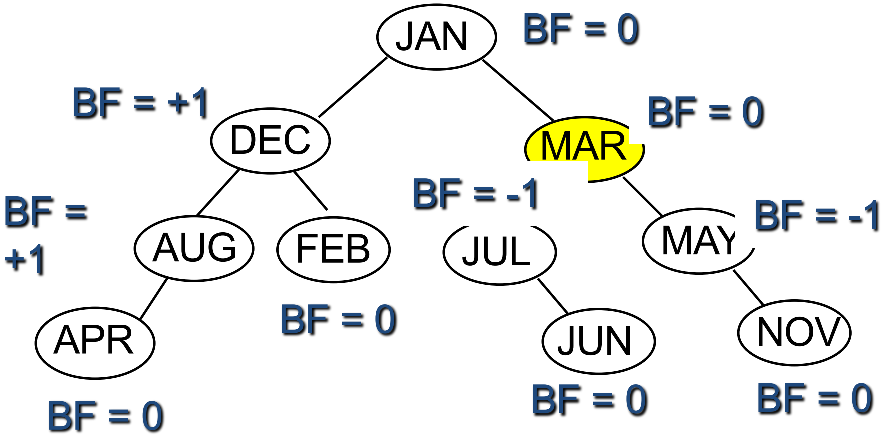
New Identifier

JUNE

After Insertion



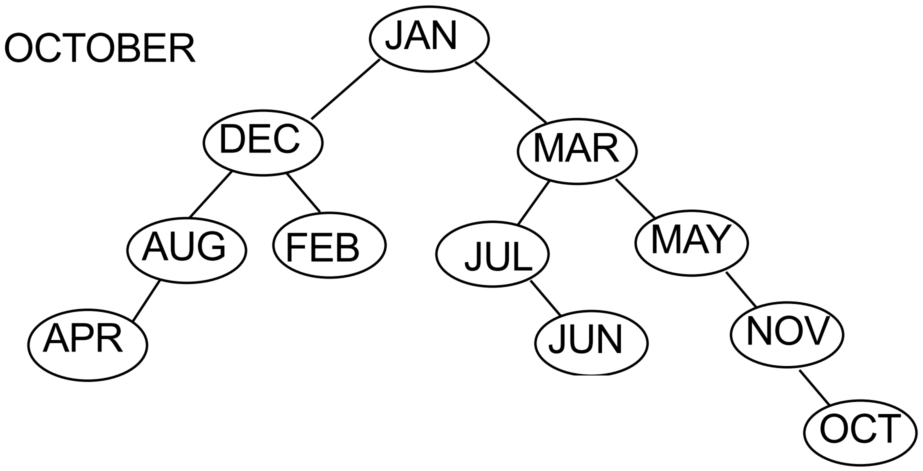

After Rebalancing

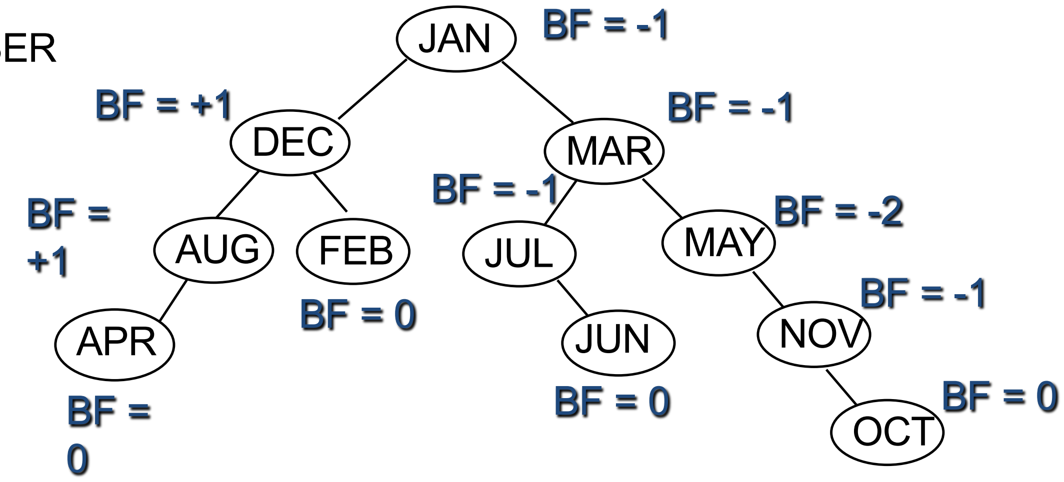



LR rebalancing

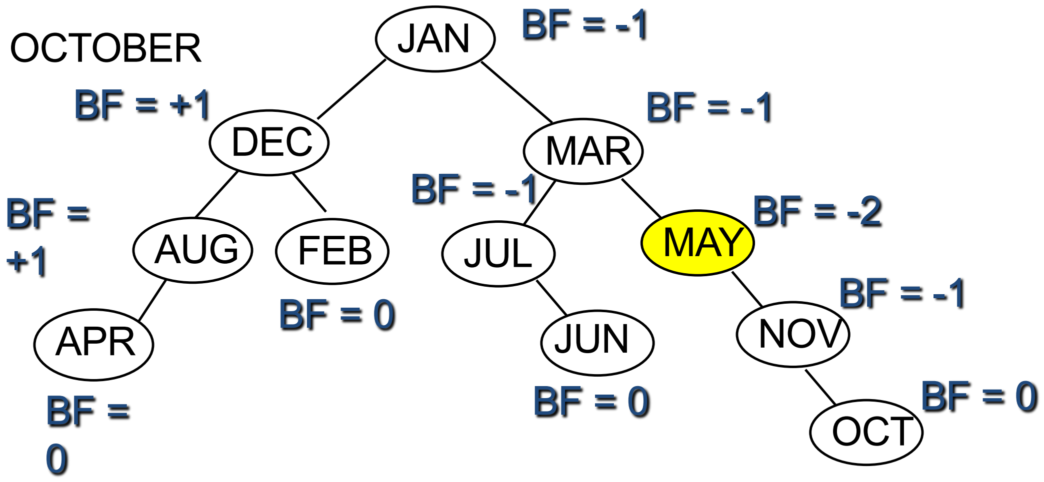
| New Identifier | After Insertion | After Rebalancing |
| --- | --- | --- |

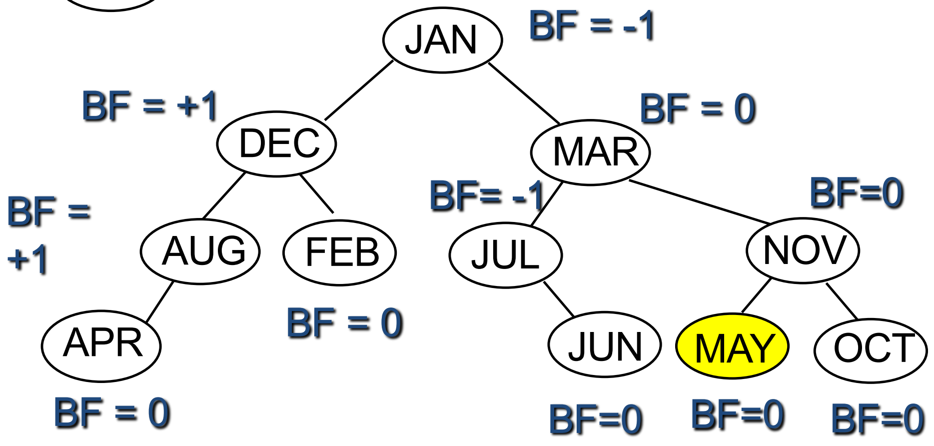
OCTOBER



| New Identifier | After Insertion | After Rebalancing |
| --- | --- | --- |
| OCTOBER |  | |

| New Identifier | After Insertion | After Rebalancing |
| --- | --- | --- |
| OCTOBER | | RR rebalancing |



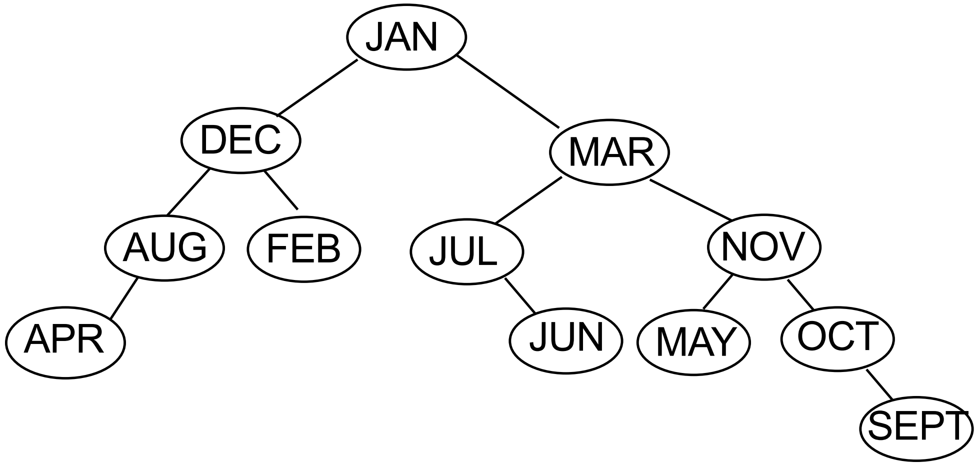



New Identifier

After Insertion

After Rebalancing

SEPTEMBER

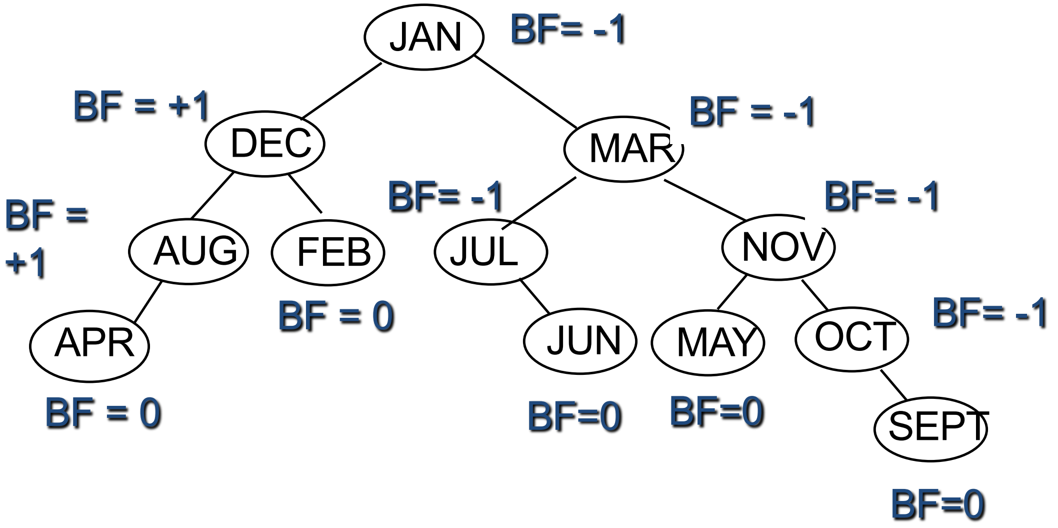


New Identifier

After Insertion

After Rebalancing

SEPTEMBER




NO REBALANCING NEEDED

# AVL Trees

- Let's refer to the node inserted as Y
- Let's refer to the nearest ancestor having balance factor +2 or -2 as A

# AVL Trees

- LL: Y is inserted in the Left subtree of the Left subtree of A
  - LL: the path from A to Y
  - Left subtree then Left subtree
- LR: Y is inserted in the Right subtree of the Left subtree of A
  - LR: the path from A to Y
  - Left subtree then Right subtree

# AVL Trees

- RR: Y is inserted in the
  Right subtree of the Right subtree of A

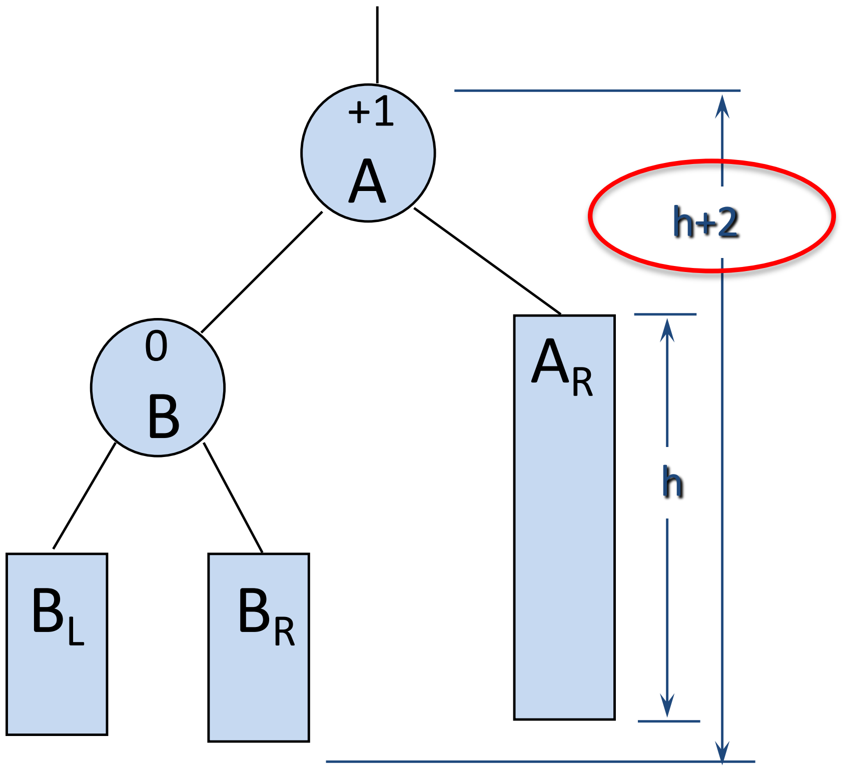
  - RR: the path from A to Y
  - Right subtree then Right subtree

- RL: Y is inserted in the
  Left subtree of the Right subtree of A

  - RL: the path from A to Y
  - Right subtree then Left subtree

# AVL Trees

Balanced Subtree

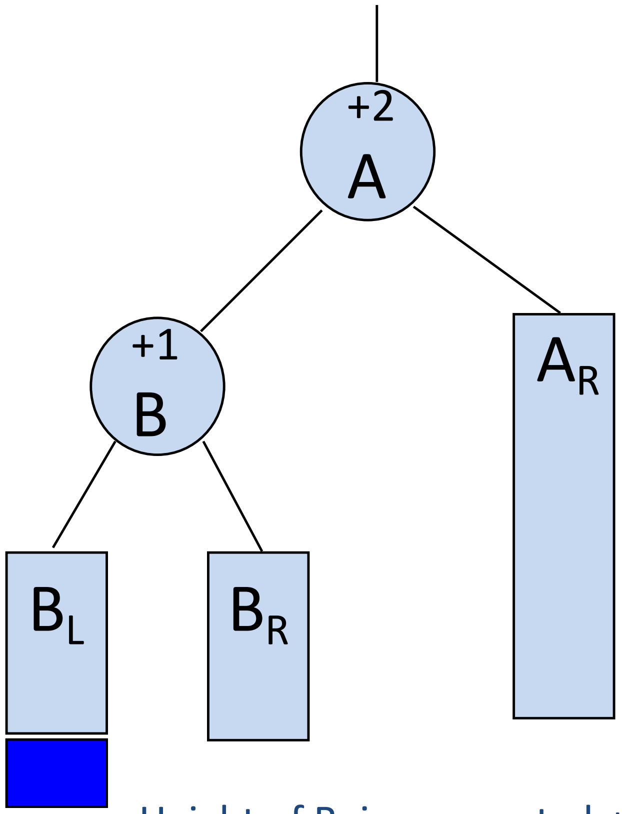



Why?

Hint:
balance factor
of A is +1

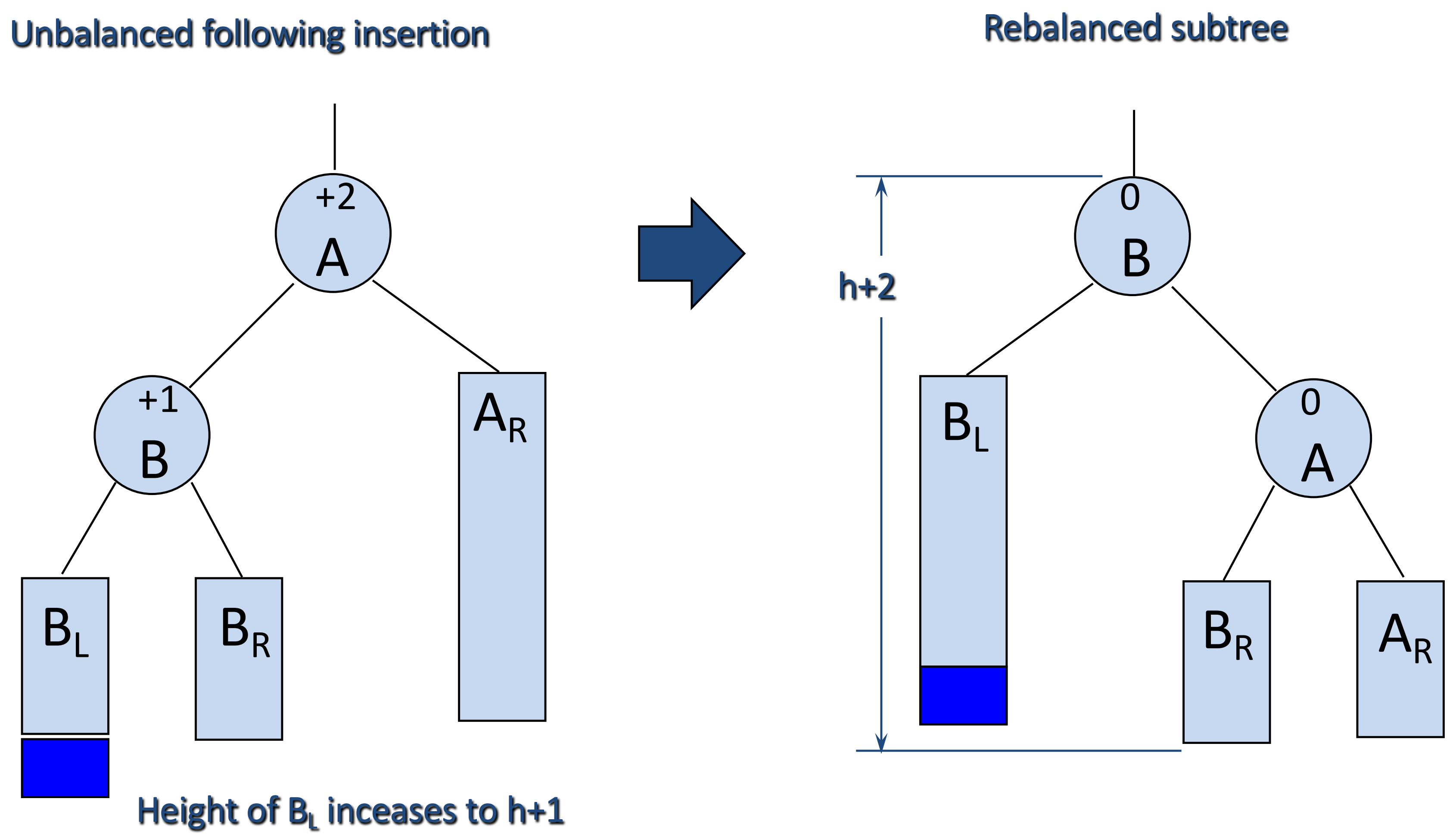
# AVL Trees

Unbalanced following insertion



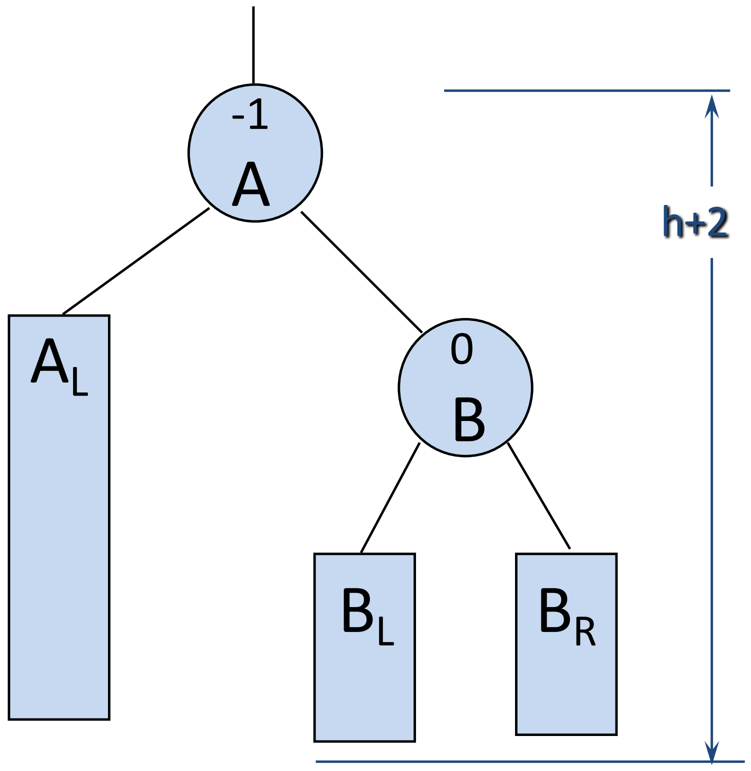

Height of $B_L$ increases to h+1

# AVL Trees - LL rotation



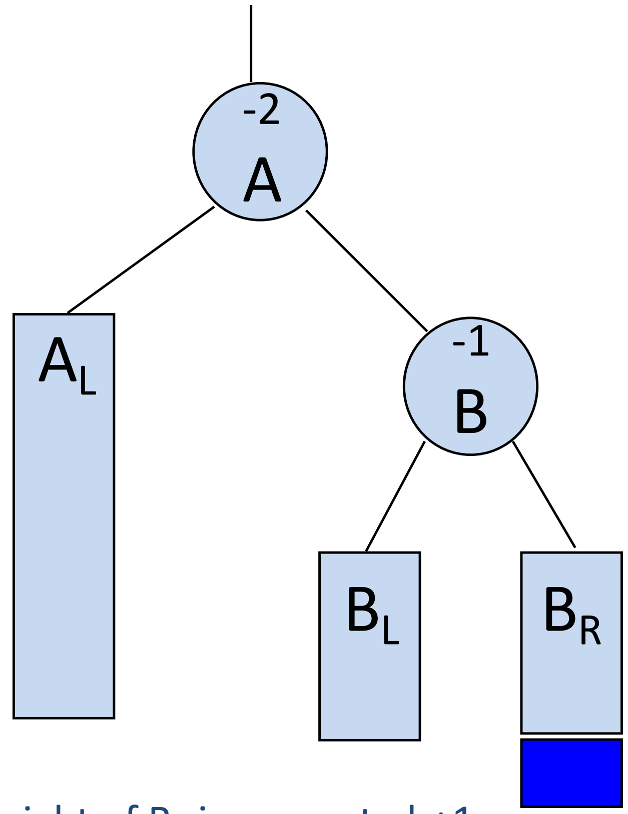
# AVL Trees

Balanced Subtree



# AVL Trees

Unbalanced following insertion

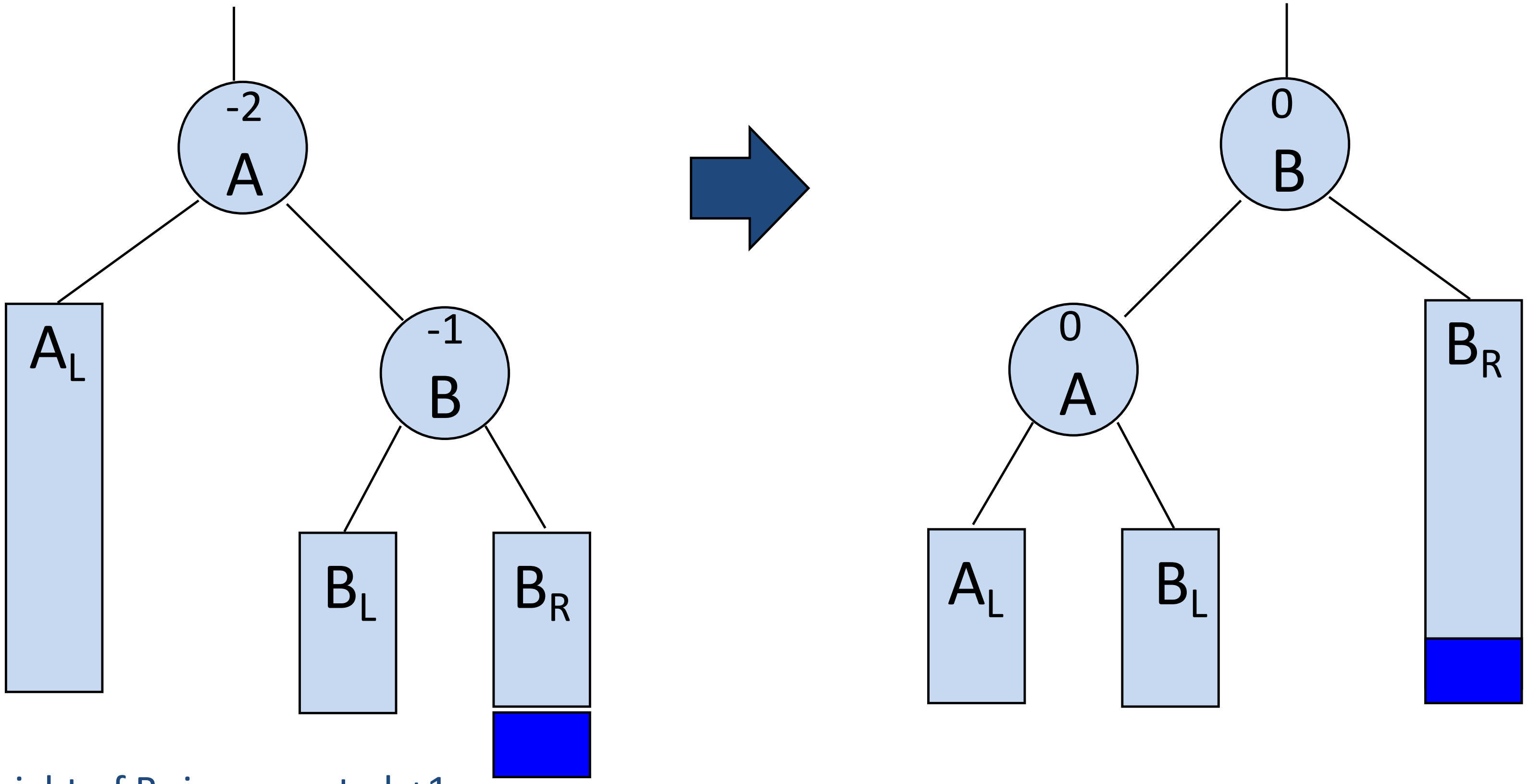



Height of $B_R$ inceases to h+1

# AVL Trees - RR Rotation

Unbalanced following insertion

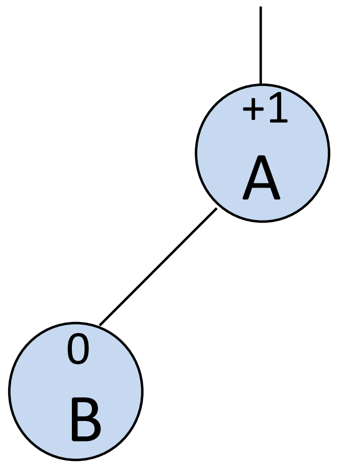
Rebalanced subtree




Height of $B_R$ inceases to h+1

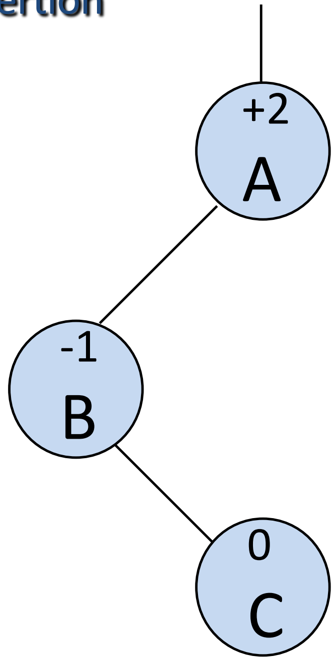
# AVL Trees

Balanced Subtree

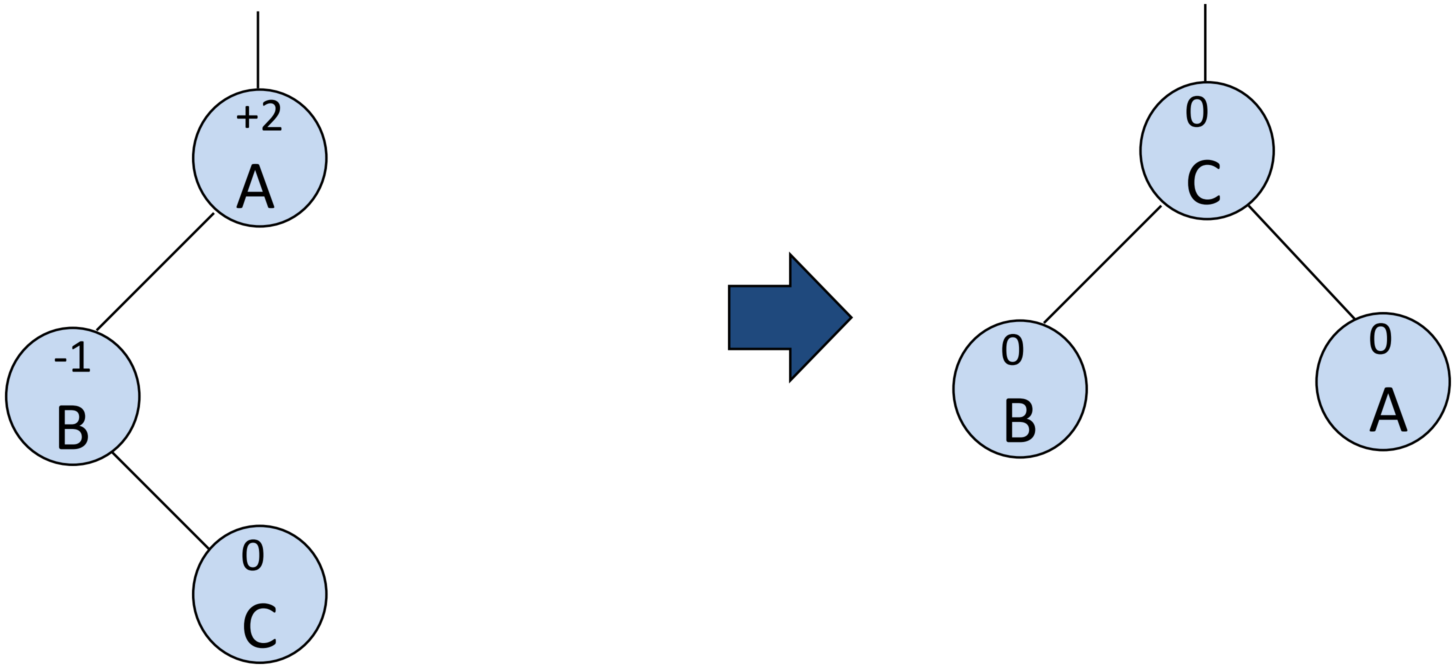


# AVL Trees

Unbalanced following insertion

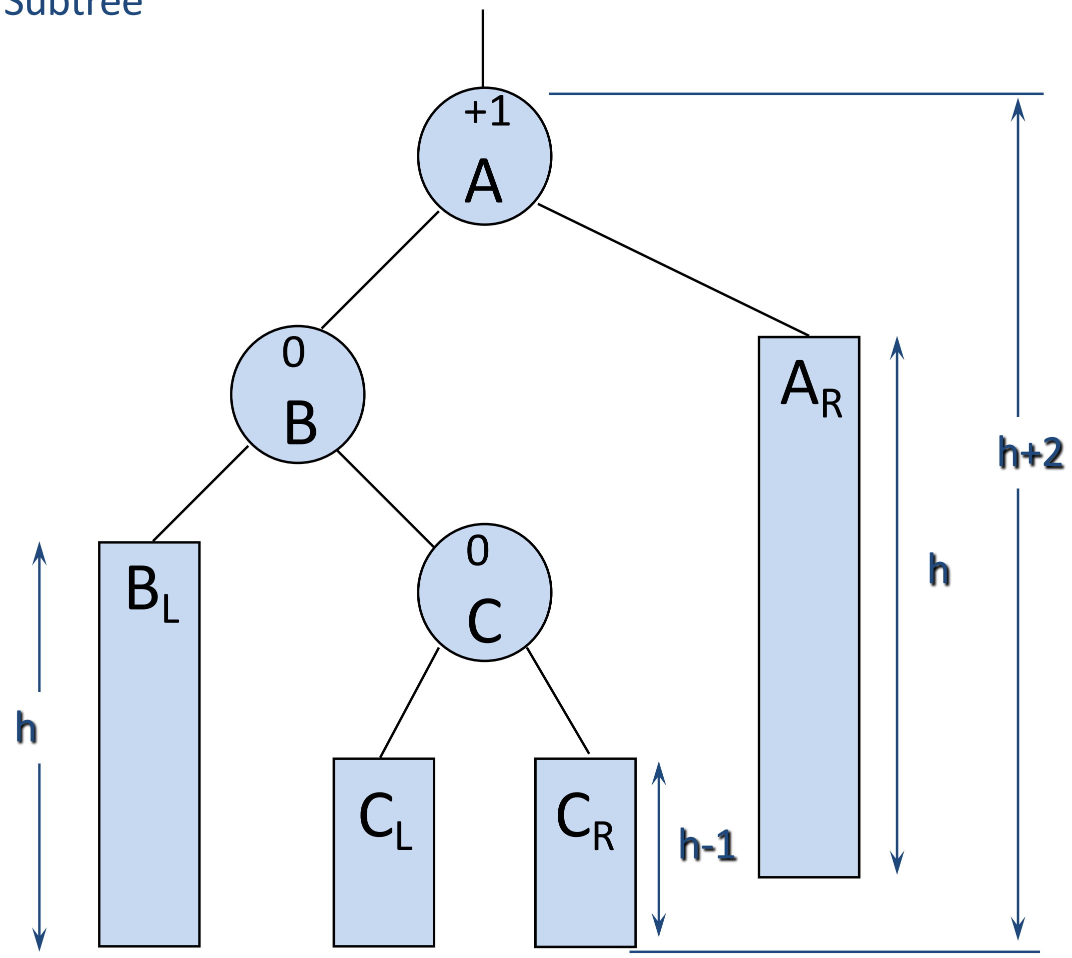


# AVL Trees - LR rotation (a)



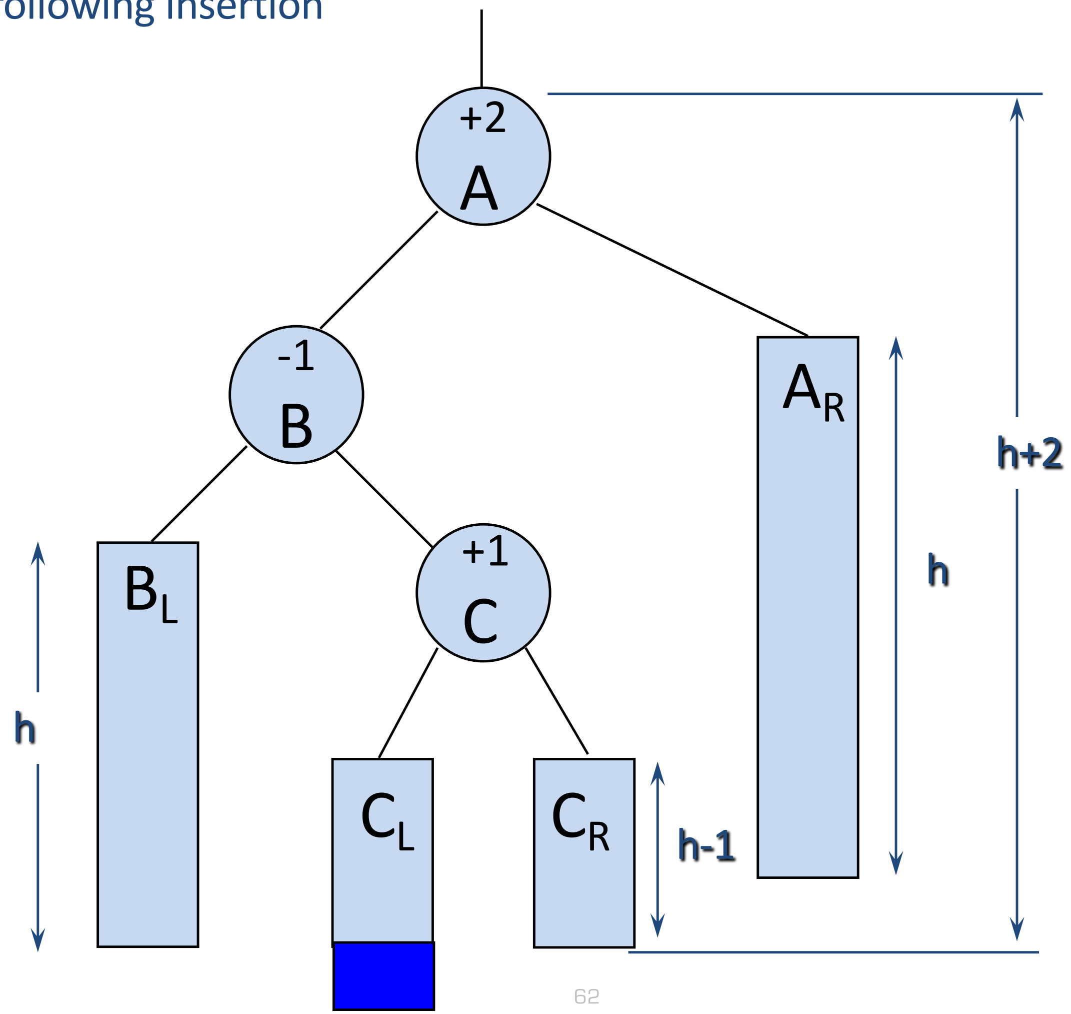
# AVL Trees

Balanced Subtree

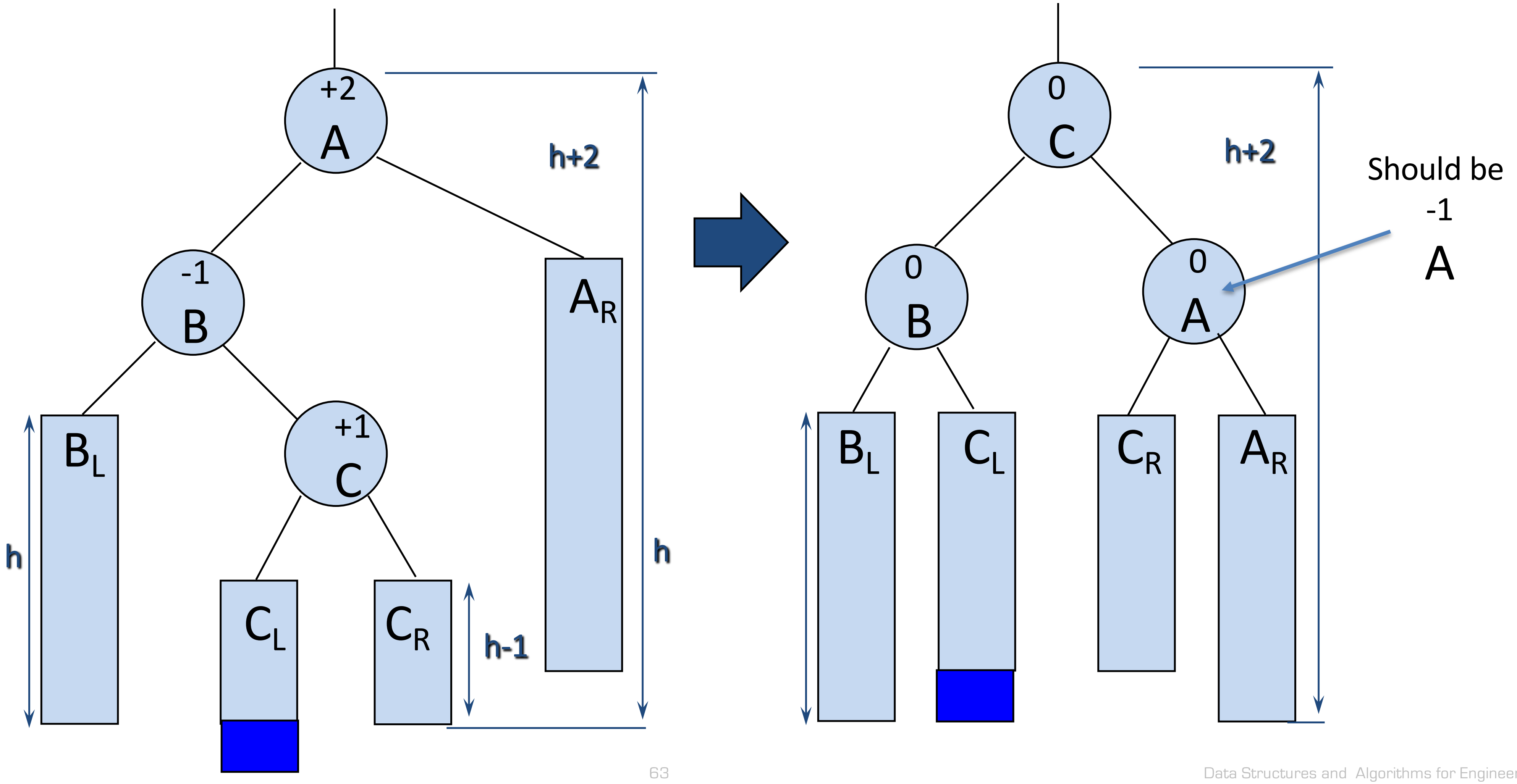


# AVL Trees

Unbalanced following insertion

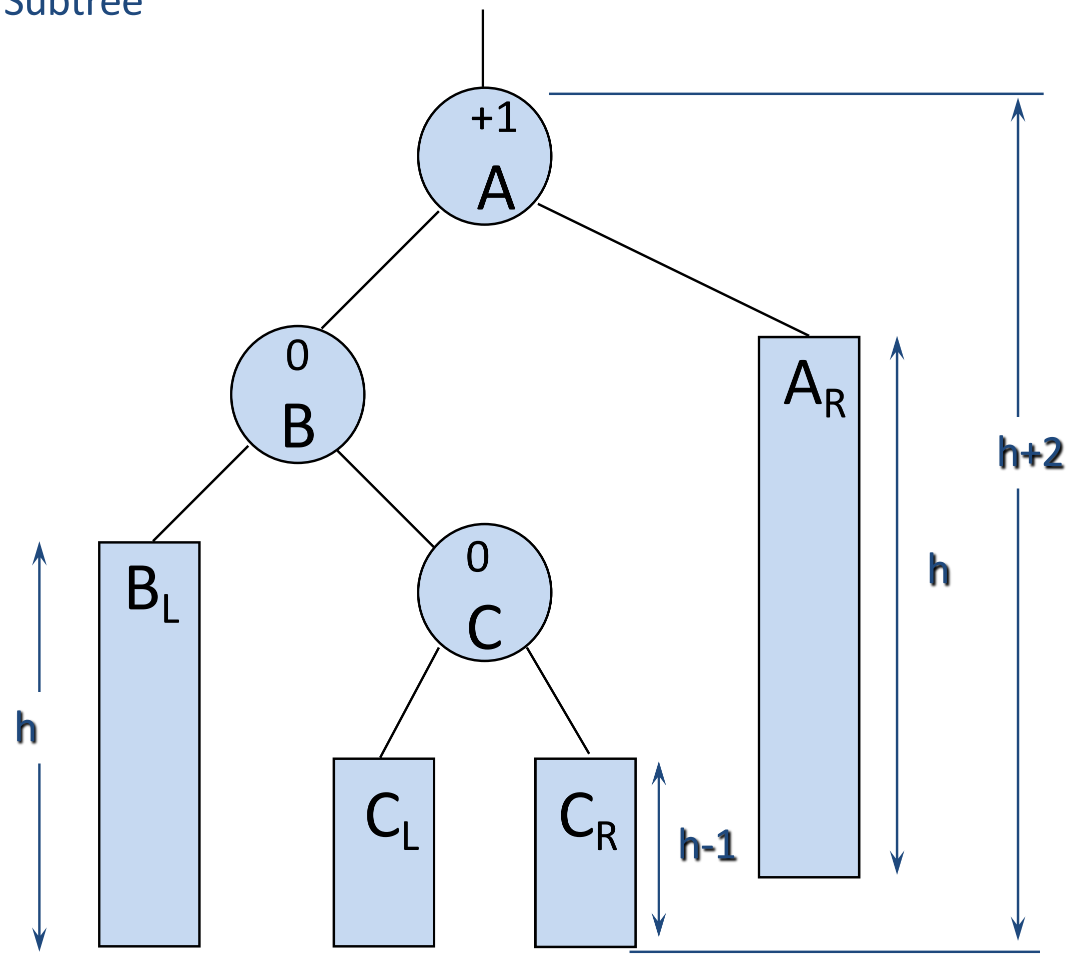


# AVL Trees - LR rotation (b)



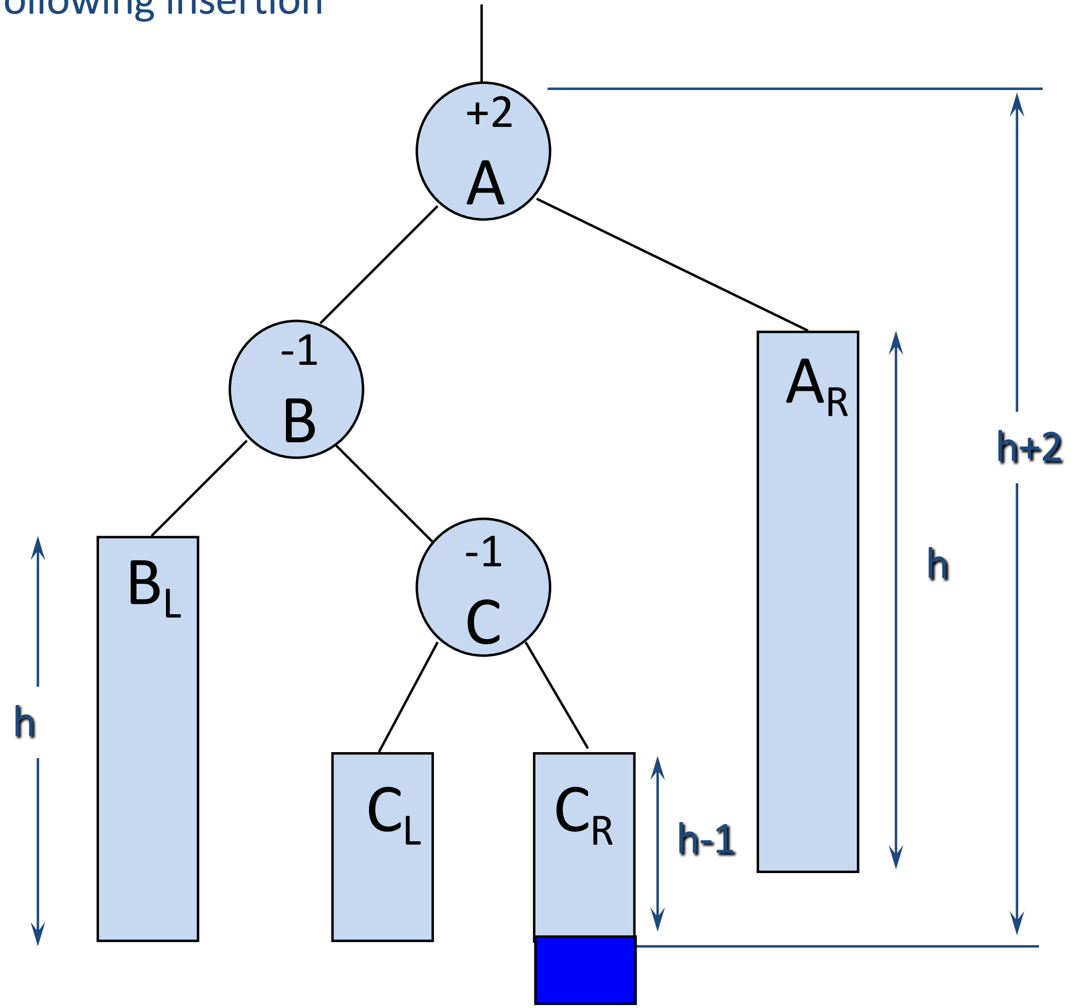
# AVL Trees

Balanced Subtree

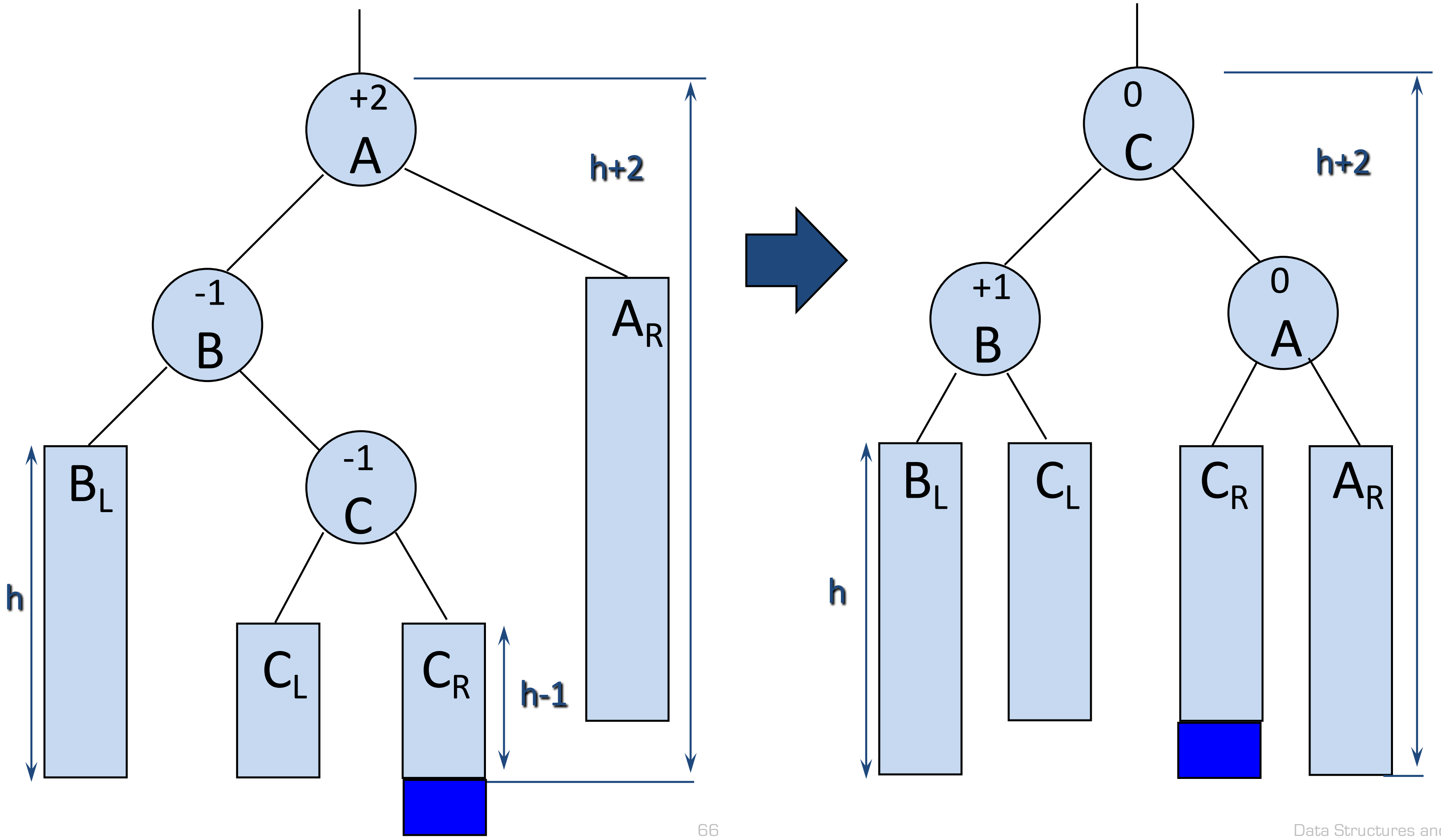


# AVL Trees

Unbalanced following insertion

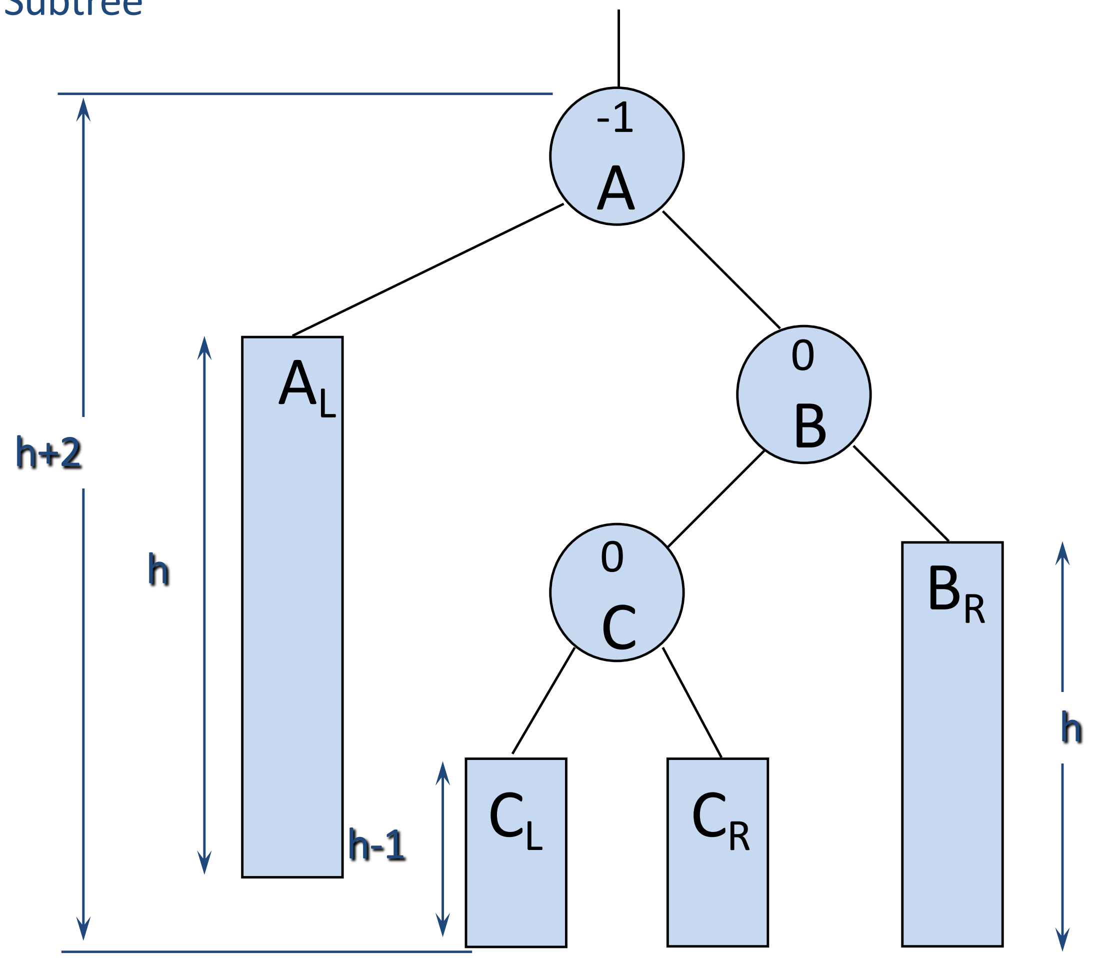


# AVL Trees - LR rotation (c)



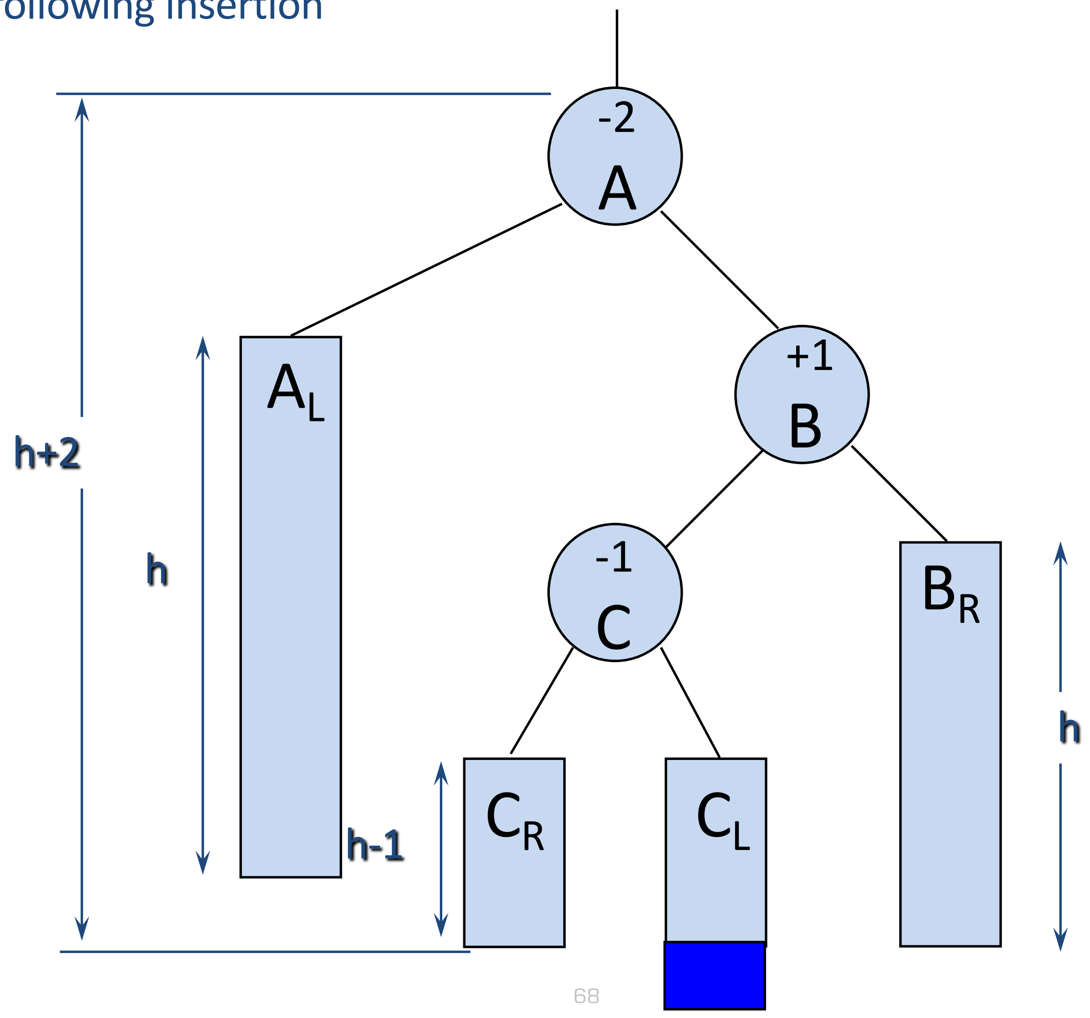
# AVL Trees

Balanced Subtree

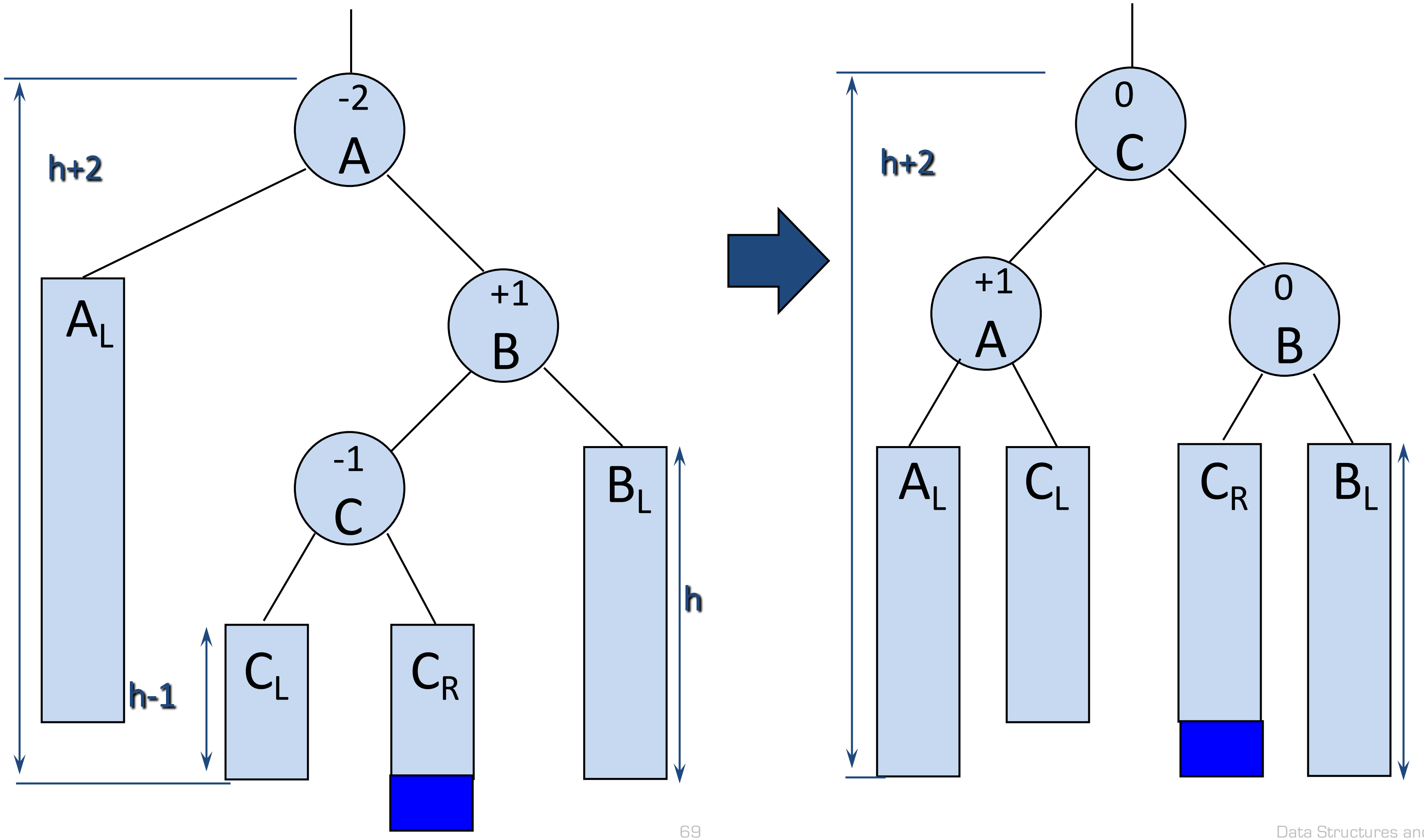


# AVL Trees

Unbalanced following insertion



# AVL Trees - RL rotation



# AVL Trees

- To carry out this rebalancing we need to locate A, i.e., to window A
  - A is the nearest ancestor to Y whose balance factor becomes +2 or -2 following insertion
  - Equally, A is the nearest ancestor to Y whose balance factor was +1 or -1 before insertion
- We also need to locate F, the parent of A … (why?)

# AVL Trees

- Note in passing that, since A is the nearest ancestor to Y whose balance factor <u>was</u> +1 or -1 <u>before</u> insertion, the balance factor of all other nodes on the path from A to Y must be 0

- When we re-balance the tree, the balance factors change (see diagrams above)

  – But changes only occur in sub-tree which is being rebalanced

# AVL Trees

- The balance factors also change  following an insertion which requires no rebalancing
- BF(A) is +1 or -1 before insertion
- Insertion causes height of one of A's sub-trees to increase by 1
- Thus, BF(A) must be 0 after insertion (since, in this case, it's not +2 or -2)

# Pseudo-code Implementation of AVL_Insert()

```
PROCEDURE AVL_insert(e:elementtype; w:windowtype;
                 T: BINTREE);

(* We assume that variables of element type have two *)
(* data fields: the information field and a balance  *)
(* factor                                            *)
(* Assume also existence of two ADT functions to     *)
(* examine these fields:                             *)
(*                      Examine_BF(w, T)             *)
(*                      Examine_data(w, T)           *)
(* and one to modify the balance factor field        *)
(*                      Replace_BF(bf, w, T)         *)
var newnode: linktype;
begin
```

# Pseudo-code Implementation of AVL_Insert()

```
IF isEmpty(tree) /* special case */
   THEN
      insert(e, w, tree);  /*insert with window */
      replace_BF(0, w, tree)
   ELSE
      /* Phase 1: locate insertion point        *)
      /* A keeps track of most recent node with *)
      (* balance factor +1 or -1               *)
      A := w;
      WHILE ((NOT IsExternal(w, T)) AND
              (NOT (e.data = Examine_Data(w, T))) DO
         IF Examine_BF(w, T) <> 0 (* non-zero BF *)
            THEN
               A := w;
         ENDIF;
```

# Pseudo-code Implementation of AVL_Insert()

```
   IF (e.data < Examine_Data(w, T) )
      THEN
         Child(0, w, T)
      ELSE IF (e.data > Examine_Data(w, T) )
         Child(1, w, T)
      ENDIF
   ENDIF
ENDWHILE
(* If not found, then embark on Phase 2: *)
(* insert & rebalance                    *)
IF IsExternal(w, T)
   THEN
      Insert(e, w, T);  (*insert as before *)
      Replace_BF(0, w, T)
ENDIF
```

# Pseudo-code Implementation of AVL_Insert()

```
(* adjust balance factors of nodes on path    *)
(* from A to parent of newly-inserted node    *)
(* By definition, they will have had BF=0     *)
(* and so must now change to +1 or -1         *)
(* Let d = this change,                       *)
(* d = +1 ... insertion in A's left subtree   *)
(* d = -1 ... insertion in A's right subtree  *)

IF (e.data < Examine_Data(A, T) )
   THEN
      v:= A;
      Child(0, v, T)
      B:= v;
      d := +1
   ELSE
```

# Pseudo-code Implementation of AVL_Insert()

```
   ELSE
      v:= A; Child(1, v, T)
      B:= v;
      d := -1
ENDIF
WHILE ((NOT IsEqual(w, v))) DO
   IF (e.data < Examine_Data(v, T) )
      THEN
         ReplaceBF(+1, v, T);
         Child(0, v, T) (* height of Left ^ *)
      ELSE
         ReplaceBF(-1, v, T);
         Child(1, v, T) (* height of Right ^ *)
   ENDIF
ENDWHILE
```

# Pseudo-code Implementation of AVL_Insert()

```
(* check to see if tree is unbalanced *)

IF (ExamineBF(A, T) = 0 )
   THEN
      ReplaceBF(d, A, T) (* still balanced *)
   ELSE
      IF ((ExamineBF(A, T) + d) = 0)
         THEN
            ReplaceBF(0, A, T)(*still balanced*)
         ELSE

            (* Tree is unbalanced      *)
            (* determine rotation type *)
```

# Pseudo-code Implementation of AVL_Insert()

```
(* Tree is unbalanced ... determine rotation type *)

IF d = +1
   THEN  (* left imbalance *)
      IF ExamineBF(B) = +1
         THEN (* LL Rotation *)
            (* replace left subtree of A *)
            (* with right subtree of B   *)
            temp := B; Child(1, temp, T);
            ReplaceChild(0, A, T,  temp);

            (* replace right subtree of B with A *)
            ReplaceChild(1, B, T, A);
            ReplaceBF(0, A, T);
            ReplaceBF(0, B, T);
```

# Pseudo-code Implementation of AVL_Insert()

```
ELSE (* LR Rotation *)
   C := B; Child(1, C, T);
   C_L := C; Child(0, C_L, T);
   C_R := C; Child(1, C_R, T);
   ReplaceChild(1, B, T, C_L);
   ReplaceChild(0, A, T, C_R);
   ReplaceChild(0, C, T, B);
   ReplaceChild(1, C, T, A);
```

# Pseudo-code Implementation of AVL_Insert()

```
IF ExamineBF(C,T) = +1 (* LR(b) *)
   THEN
      ReplaceBF(-1, A, T);
      ReplaceBF(0, B, T);
   ELSE
      IF ExamineBF(C,T) = -1 (* LR(c) *)
         THEN
            ReplaceBF(+1, B, T);
            ReplaceBF(0, A, T);
         ELSE                 (* LR(a) *)
            ReplaceBF(0, A, T);
            ReplaceBF(0, B, T);
      ENDIF
ENDIF
```

# Pseudo-code Implementation of AVL_Insert()

```
            (* B is new root *)
            ReplaceBF(0, C, T);
            B := C
        ENDIF (* LR rotation *)
    ELSE (* right imbalance *)

        (* this is symmetric to left imbalance *)
        (* and is left as an exercise!          *)

ENDIF (* d = +1 *)
```

# Pseudo-code Implementation of AVL_Insert()

```
                    (* the subtree with root B has been *)
                    (* rebalanced and it now replaces   *)
                    (* A as the root of the originally  *)
                    (* unbalanced tree                  *)

                    ReplaceTree(A, T, B)
                    (* Replace subtree A with B in T     *)
                    (* Note: this is a trivial operation *)
                    (* since we are using a complex      *)
                    (* window variable                   *)
                 ENDIF
           ENDIF
     ENDIF
END (* AVL_Insert() *)
```