Chapter Fourteen

Intro. To Dictionaries

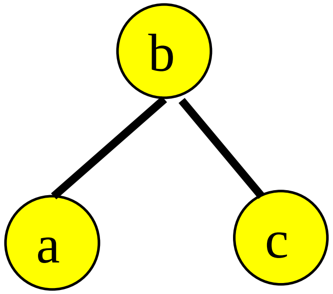
Static Dictionaries

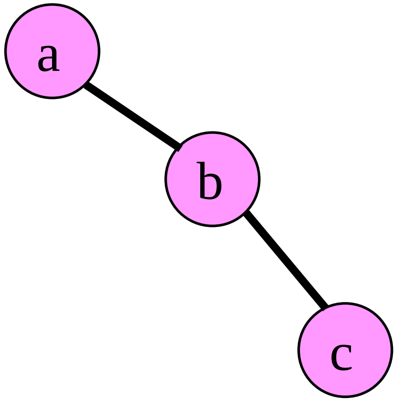
- Collection of items.
- Each item is a pair.
  - (key, element)
  - Pairs have different keys.
- Operations are:
  - initialize/create
  - get (search)
- Each item/key/element has an estimated access frequency (or probability).
- Consider binary search tree only.

Example

Key	Probability
a	0.8
b	0.1
c	0.1

a < b < c





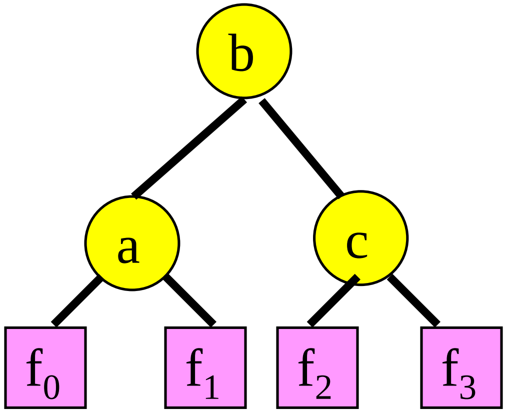
Cost = 0.8 * 2 + 0.1 * 1 + 0.1 * 2

= 1.9

Cost = 0.8 * 1 + 0.1 * 2 + 0.1 * 3

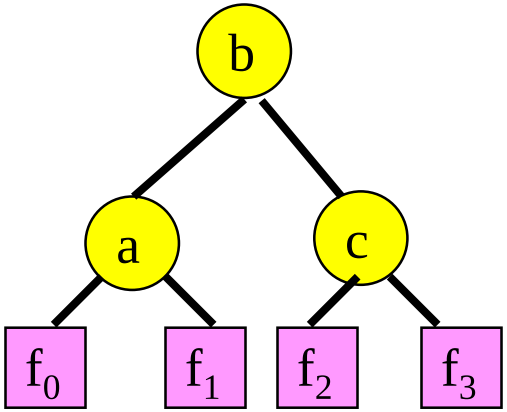
= 1.2

Search Types



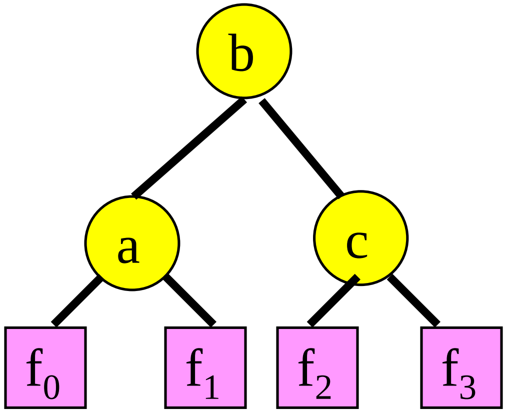
- Successful.
  - Search for a key that is in the dictionary.
  - Terminates at an internal node.
- Unsuccessful.
  - Search for a key that is not in the dictionary.
  - Terminates at an external/failure node.

Internal And External Nodes



- A binary tree with n internal nodes has $n + 1$ external nodes.
- Let s_1, s_2, …, s_n be the internal nodes, in inorder.
- $key(s_1) < key(s_2) < \ldots < key(s_n)$.
- Let $key(s_0) = –infinity$ and $key(s_{n+1}) = infinity$.
- Let f_0, f_1, …, f_n be the external nodes, in inorder.
- f_i is reached iff $key(s_i) < search\ key < key(s_{i+1})$.

Cost Of Binary Search Tree



- Let p_i = probability for key(s_i).
- Let q_i = probability for key(s_i) < search key < key(s_{i+1}).
- Sum of ps and qs = 1.
- Cost of tree = $\Sigma_{0 <= i <= n} q_i (\text{level}(f_i) - 1)$

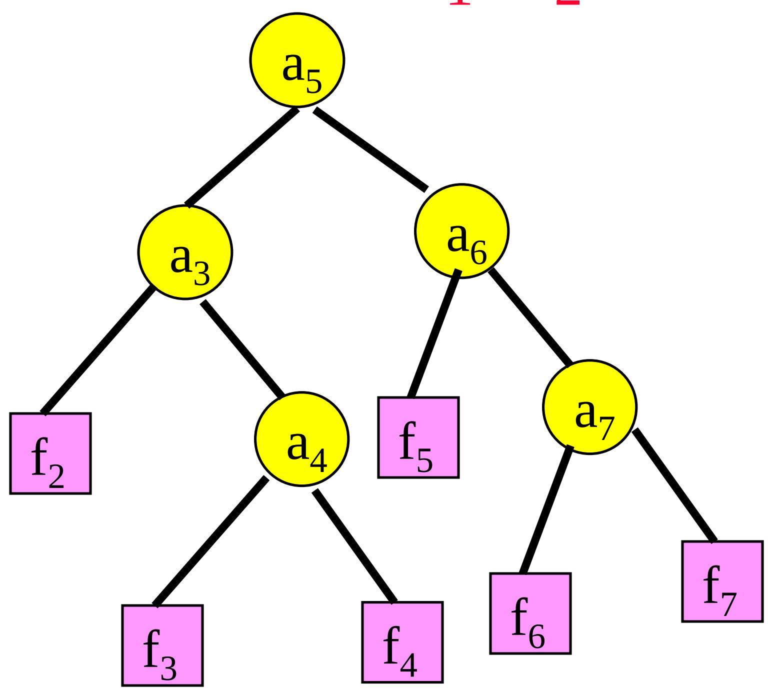
  $+ \Sigma_{1 <= i <= n} p_i * \text{level}(s_i)$
- Cost = weighted path length.

Brute Force Algorithm

- Generate all binary search trees with n internal nodes.
- Compute the weighted path length of each.
- Determine tree with minimum weighted path length.
- Number of trees to examine is $O(4^n/n^{1.5})$.
- Brute force approach is impractical for large n.

Dynamic Programming

- Keys are $a_1 < a_2 < \ldots < a_n$.
- Let $T_{i\,j}$ = least cost tree for $a_{i+1}, a_{i+2}, \ldots, a_j$.
- T_{0n} = least cost tree for $a_1, a_2, \ldots, a_n$.



$T_{2,7}$

- $T_{i\,j}$ includes $p_{i+1}, p_{i+2}, \ldots, p_j$ and $q_i, q_{i+1}, \ldots, q_j$.

Terminology

- $T_{i\,j}$ = least cost tree for $a_{i+1}, a_{i+2}, \ldots, a_j$.
- $c_{i\,j}$ = cost of $T_{i\,j}$

  $= \Sigma_{i <= u <= j}\, q_u\,(\text{level}(f_u) - 1)$

  $+ \Sigma_{i < u <= j}\, p_u \;*\, \text{level}(s_u)$.
- $r_{i\,j}$ = root of $T_{i\,j}$.
- $w_{i\,j}$ = weight of $T_{i\,j}$

  = sum of ps and qs in $T_{i\,j}$

  $= p_{i+1} + p_{i+2} + \ldots + p_j + q_i + q_{i+1} + \ldots + q_j$

$i = j$

- $T_{i\,j}$ includes p_{i+1}, p_{i+2}, …, p_j and q_i, q_{i+1}, …, q_j.
- $T_{i\,i}$ includes q_i only.

f_i T_{ii}

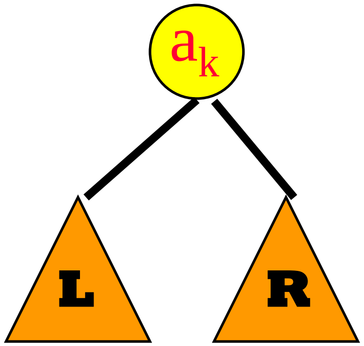
- $c_{i\,i}$ = cost of $T_{i\,i}$ = 0.
- $r_{i\,i}$ = root of $T_{i\,i}$ = 0.
- $w_{i\,i}$ = weight of $T_{i\,i}$

  = sum of ps and qs in $T_{i\,i}$

  = q_i

$i < j$

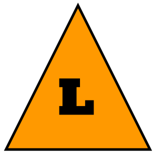
- $T_{i\,j}$ = least cost tree for $a_{i+1}, a_{i+2}, \ldots, a_j$.
- $T_{i\,j}$ includes $p_{i+1}, p_{i+2}, \ldots, p_j$ and $q_i, q_{i+1}, \ldots, q_j$.
- Let a_k, $i < k <= j$, be in the root of $T_{i\,j}$.



$T_{i\,j}$

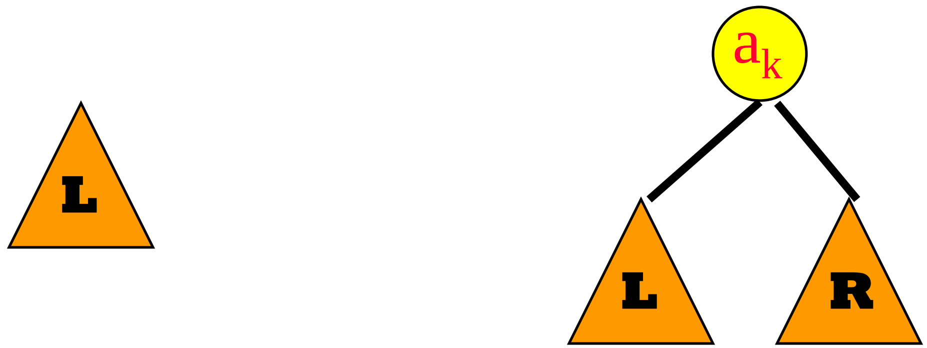
- **L** includes $p_{i+1}, p_{i+2}, \ldots, p_{k-1}$ and $q_i, q_{i+1}, \ldots, q_{k-1}$.
- **R** includes $p_{k+1}, p_{i+2}, \ldots, p_j$ and $q_k, q_{k+1}, \ldots, q_j$.

cost(L)



- L includes p_{i+1}, p_{i+2}, …, p_{k-1} and q_i, q_{i+1}, …, q_{k-1}.
- cost(L) = weighted path length of L when viewed as a stand alone binary search tree.

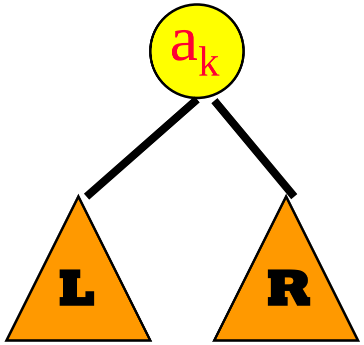
Contribution To c_{ij}



$T_{i\,j}$

- $c_{i\,j} = \Sigma_{i <= u <= j}\, q_u\, (\text{level}(f_u) - 1)$
  $+ \Sigma_{i < u <= j}\, p_u * \text{level}(s_u)$.
- When **L** is viewed as a subtree of $T_{i\,j}$, the level of each node is 1 more than when **L** is viewed as a stand alone tree.
- So, contribution of **L** to c_{ij} is $\text{cost}(\mathbf{L}) + w_{i\,k-1}$.

c_{ij}

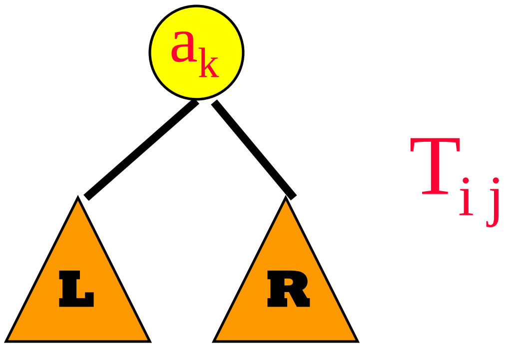


$T_{i\,j}$

- Contribution of **L** to c_{ij} is $\mathrm{cost}(\mathbf{L}) + w_{i\,k-1}$.
- Contribution of **R** to c_{ij} is $\mathrm{cost}(\mathbf{R}) + w_{kj}$.
- $c_{ij} = \mathrm{cost}(\mathbf{L}) + w_{i\,k-1} + \mathrm{cost}(\mathbf{R}) + w_{kj} + p_k$

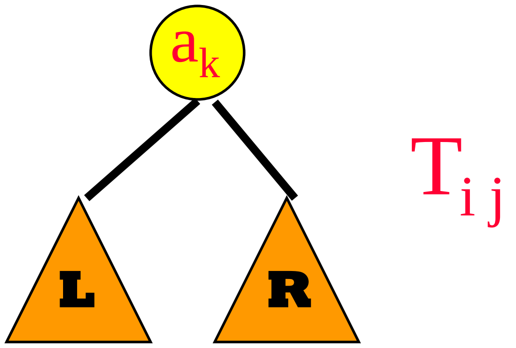
  $= \mathrm{cost}(\mathbf{L}) + \mathrm{cost}(\mathbf{R}) + w_{ij}$

c_{ij}



- $c_{ij} = \text{cost}(\mathbf{L}) + \text{cost}(\mathbf{R}) + w_{ij}$
- $\text{cost}(\mathbf{L}) = c_{ik-1}$
- $\text{cost}(\mathbf{R}) = c_{kj}$
- $c_{ij} = c_{ik-1} + c_{kj} + w_{ij}$
- Don't know k.
- $c_{ij} = \min_{i < k <= j}\{c_{ik-1} + c_{kj}\} + w_{ij}$

c_{ij}

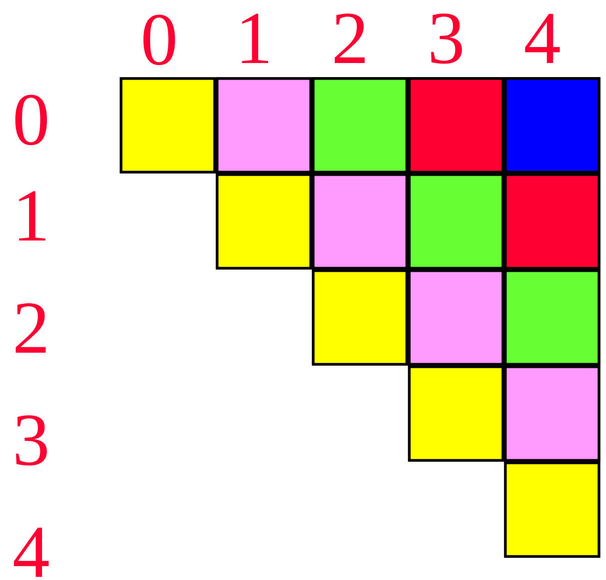


- $c_{ij} = \min_{i < k <= j}\{c_{ik-1} + c_{kj}\} + w_{ij}$
- $r_{ij} = k$ that minimizes right side.

Computation Of c_{0n} And r_{0n}

- Start with $c_{i\,i} = 0$, $r_{i\,i} = 0$, $w_{i\,i} = q_i$, $0 <= i <= n$ (zero-key trees).
- Use $c_{ij} = \min_{i < k <= j}\{c_{ik-1} + c_{kj}\} + w_{ij}$ to compute c_{ii+1}, $r_{i\,i+1}$, $0 <= i <= n - 1$ (one-key trees).
- Now use the equation to compute c_{ii+2}, $r_{i\,i+2}$, $0 <= i <= n - 2$ (two-key trees).
- Now use the equation to compute c_{ii+3}, $r_{i\,i+3}$, $0 <= i <= n - 3$ (three-key trees).
- Continue until c_{0n} and r_{0n}(n-key tree) have been computed.

Computation Of c_{0n} And r_{0n}



c_{ij}, r_{ij}, $i <= j$

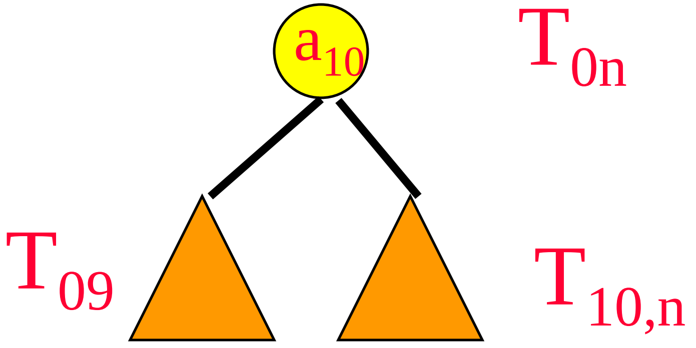
Complexity

- $c_{ij} = \min_{i < k <= j}\{c_{ik-1} + c_{kj}\} + w_{ij}$
- $O(n)$ time to compute one c_{ij}.
- $O(n^2)$ c_{ij}s to compute.
- Total time is $O(n^3)$.
- May be reduced to $O(n^2)$ by using

$$c_{ij} = \min_{ri,j-1 < k <= ri+1,j}\{c_{ik-1} + c_{kj}\} + w_{ij}$$

Construct T_{0n}

- Root is r_{0n}.
- Suppose that $r_{0n} = 10$.



- Construct T_{09} and $T_{10,n}$ recursively.
- Time is $O(n)$.

Dynamic Dictionaries

- Primary Operations:
  - Get(key) => search
  - Insert(key, element) => insert
  - Delete(key) => delete
- Additional operations:
  - Ascend()
  - Get(index)
  - Delete(index)

Complexity Of Dictionary Operations
Get(), Insert() and Delete()

Data Structure	Worst Case	Expected
Hash Table	O(n)	O(1)
Binary Search Tree	O(n)	O(log n)
Balanced Binary Search Tree	O(log n)	O(log n)

n is number of elements in dictionary

Complexity Of Other Operations

Ascend(), Get(index), Delete(index)

Data Structure	Ascend	Get and Delete
Hash Table	O(D + n log n)	O(D + n log n)
Indexed BST	O(n)	O(n)
Indexed Balanced BST	O(n)	O(log n)

D is number of buckets

AVL Tree

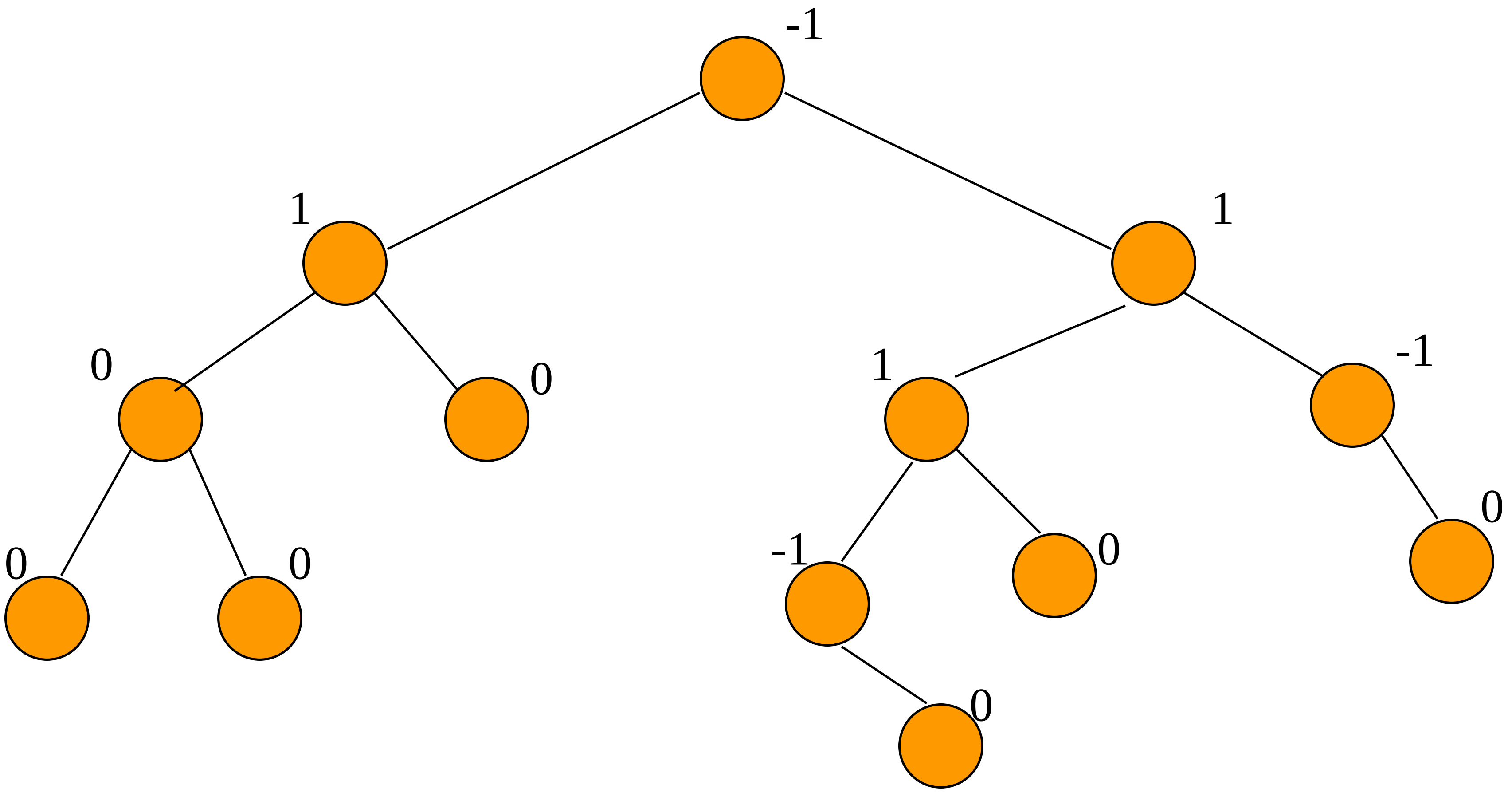
- binary tree
- for every node x, define its balance factor

  balance factor of x = height of left subtree of x

  – height of right subtree of x

- balance factor of every node x is – 1, 0, or 1

Balance Factors



This is an AVL tree.

Height Of An AVL Tree

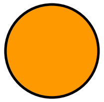
The height of an AVL tree that has n nodes is at most $1.44 \log_2 (n+2)$.

The height of every n node binary tree is at least $\log_2 (n+1)$.

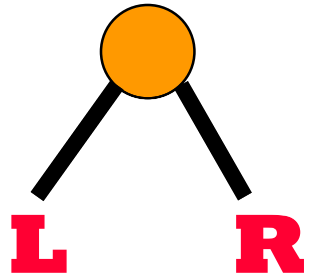
$$\log_2 (n+1) <= \text{height} <= 1.44 \log_2 (n+2)$$

Proof Of Upper Bound On Height

- Let N_h = min # of nodes in an AVL tree whose height is h.
- $N_0 = 0$.
- $N_1 = 1$.



N_h, $h > 1$

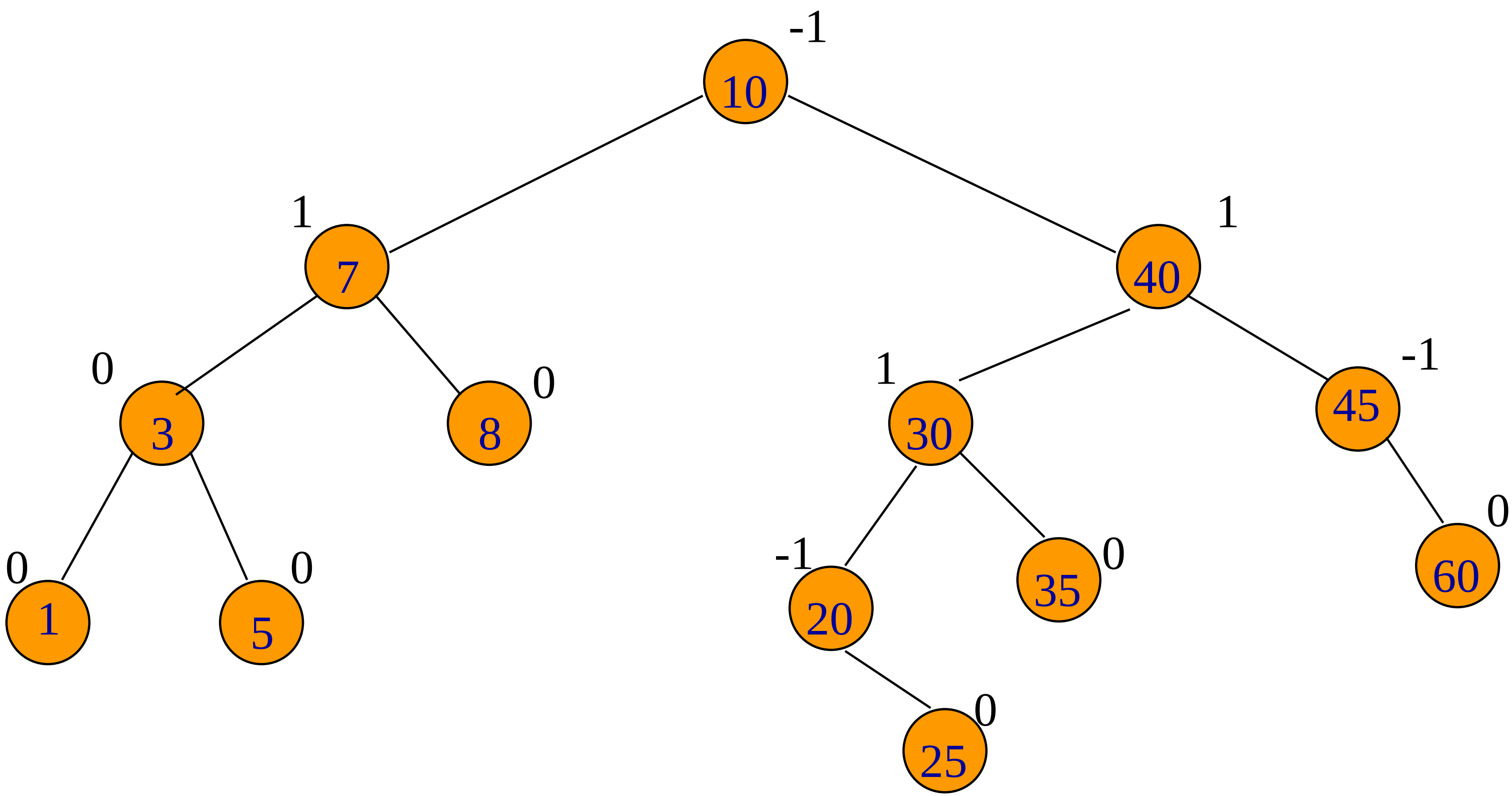


- Both **L** and **R** are AVL trees.
- The height of one is h-1.
- The height of the other is h-2.
- The subtree whose height is h-1 has N_{h-1} nodes.
- The subtree whose height is h-2 has N_{h-2} nodes.
- So, $N_h = N_{h-1} + N_{h-2} + 1$.

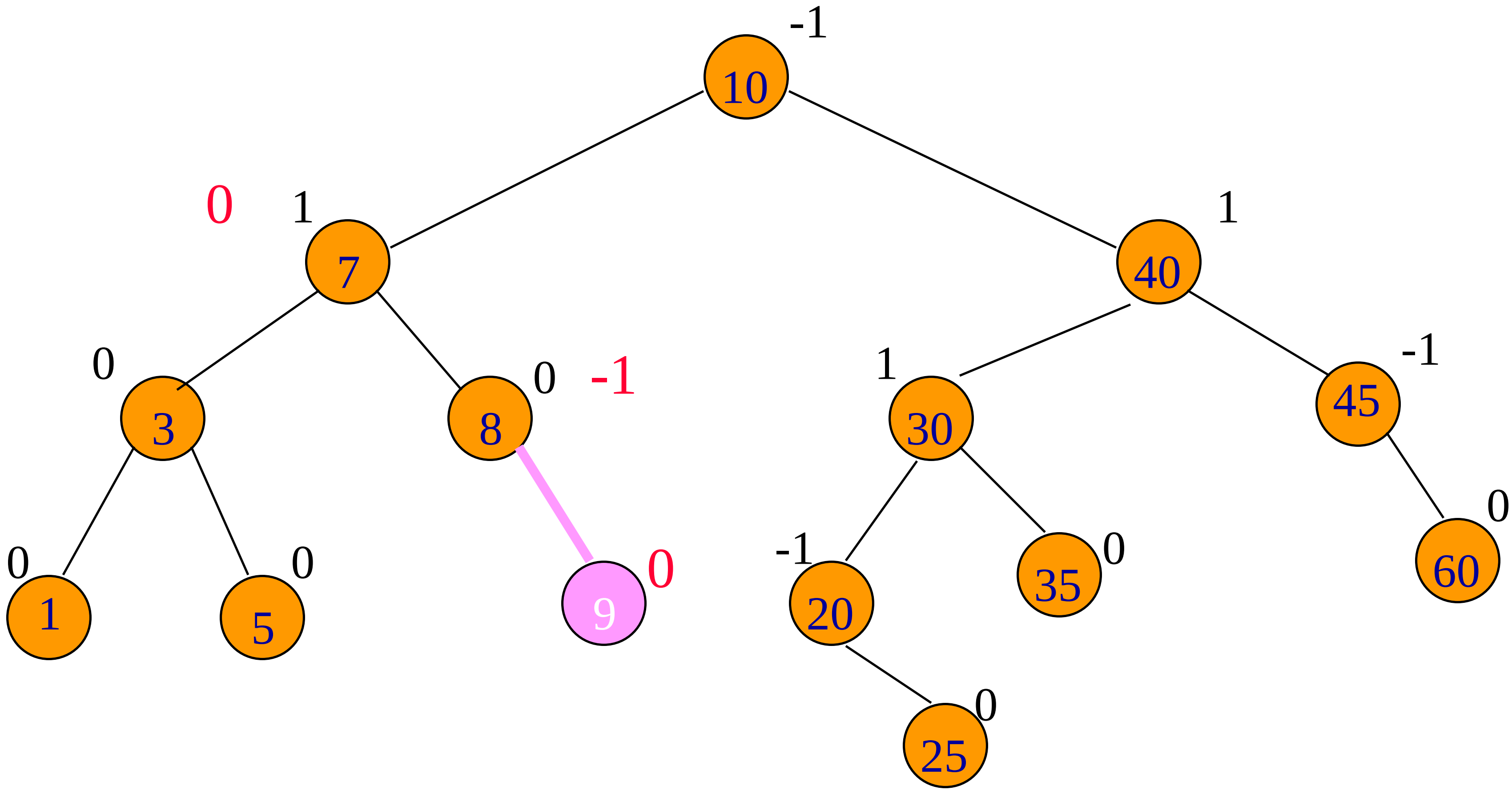
Fibonacci Numbers

- $F_0 = 0, F_1 = 1.$
- $F_i = F_{i-1} + F_{i-2}, i > 1.$
- $N_0 = 0, N_1 = 1.$
- $N_h = N_{h-1} + N_{h-2} + 1, i > 1.$
- $N_h = F_{h+2} - 1.$
- $\underline{F_i \sim \phi^i/\text{sqrt}(5)}.$
- $\phi = (1 + \text{sqrt}(5))/2.$

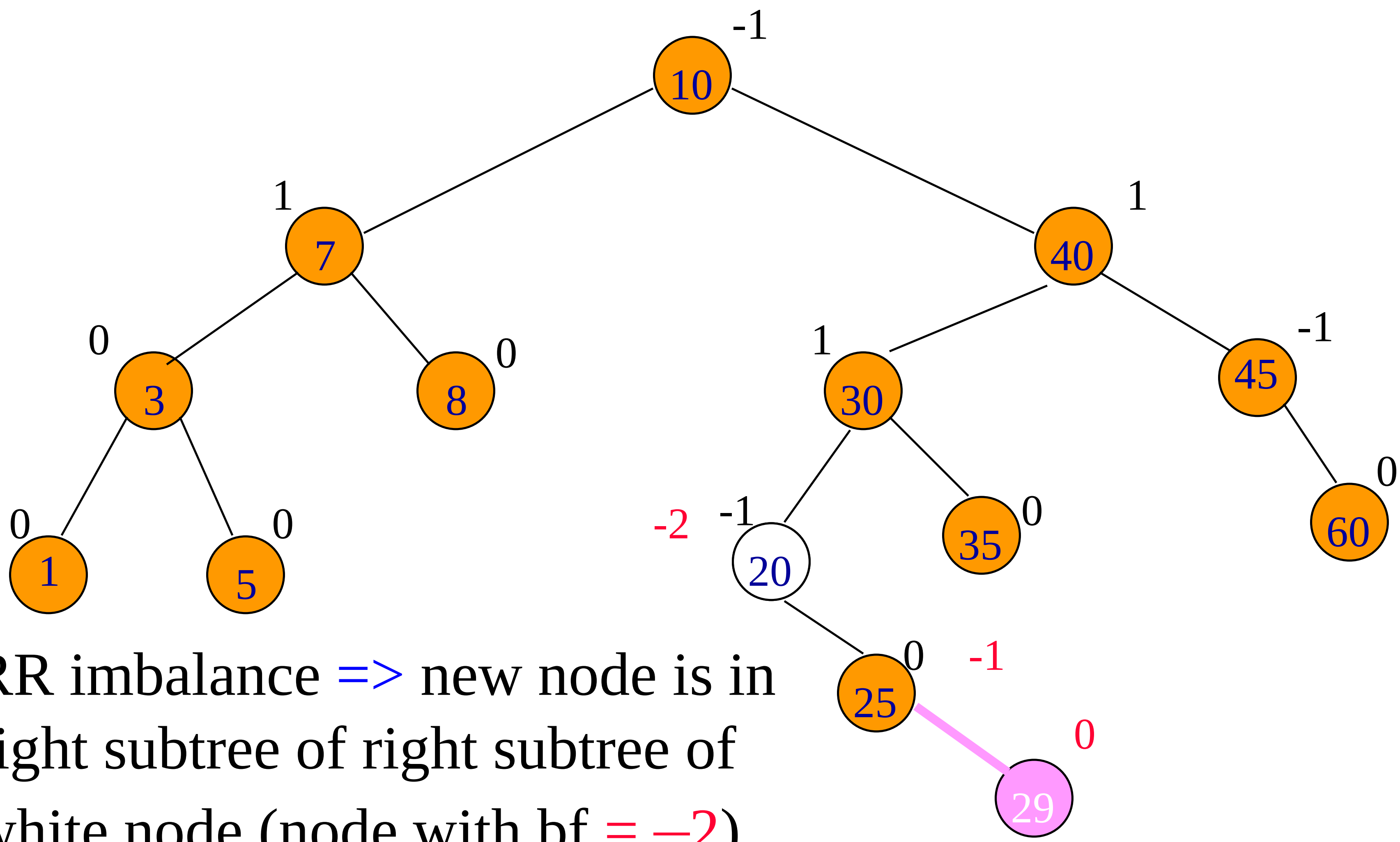
AVL Search Tree



Insert(9)

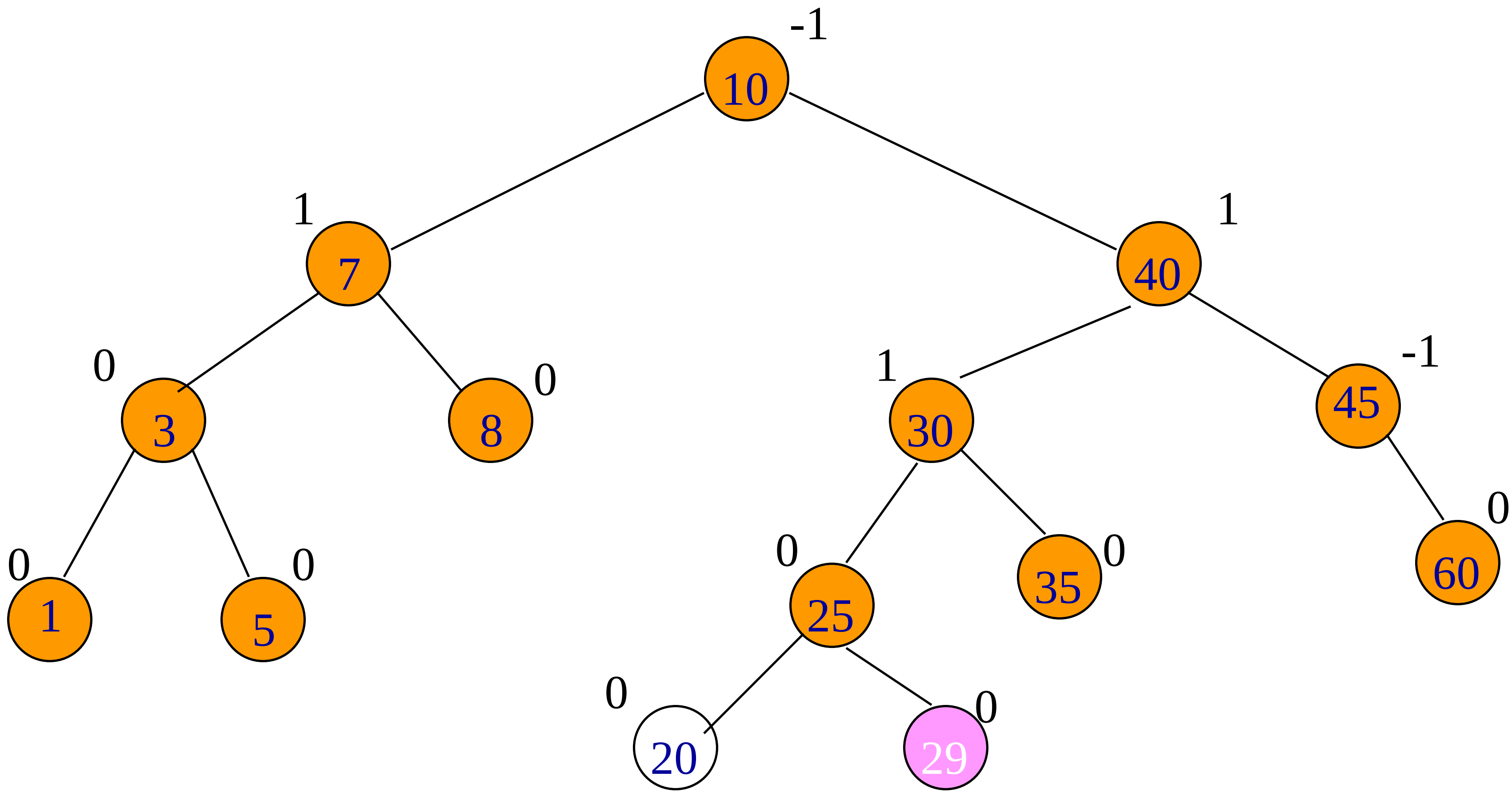


Insert(29)



RR imbalance => new node is in right subtree of right subtree of white node (node with bf = –2)

Insert(29)



RR rotation.

Insert

- Following insert, retrace path towards root and adjust balance factors as needed.
- Stop when you reach a node whose balance factor becomes 0, 2, or –2, or when you reach the root.
- The new tree is not an AVL tree only if you reach a node whose balance factor is either 2 or –2.
- In this case, we say the tree has become unbalanced.

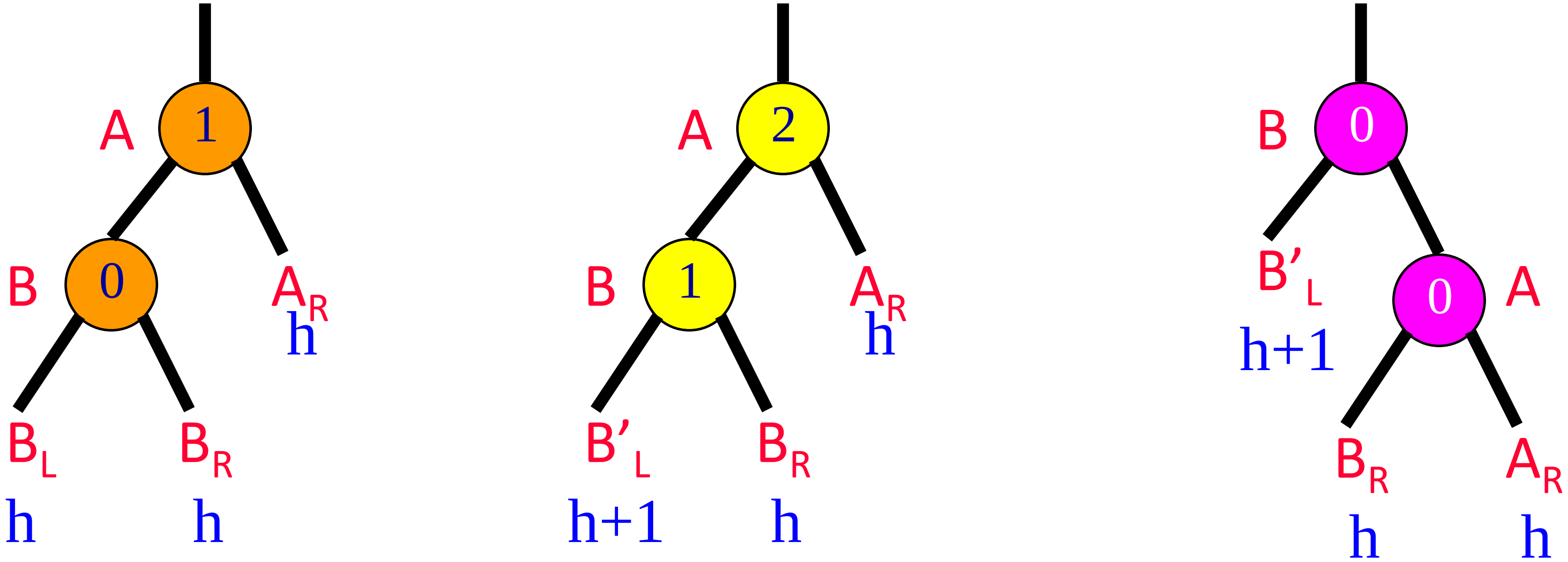
A-Node

- Let A be the nearest ancestor of the newly inserted node whose balance factor becomes +2 or –2 following the insert.
- Balance factor of nodes between new node and A is 0 before insertion.

Imbalance Types

- RR … newly inserted node is in the right subtree of the right subtree of A.
- LL … left subtree of left subtree of A.
- RL… left subtree of right subtree of A.
- LR… right subtree of left subtree of A.

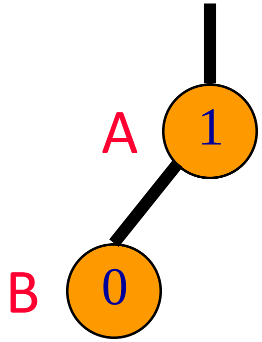
LL Rotation



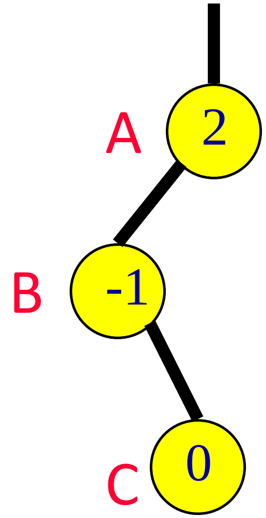
Before insertion. After insertion. After rotation.

- Subtree height is unchanged.
- No further adjustments to be done.

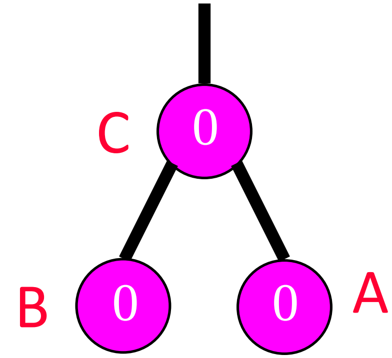
LR Rotation (case 1)



Before insertion.



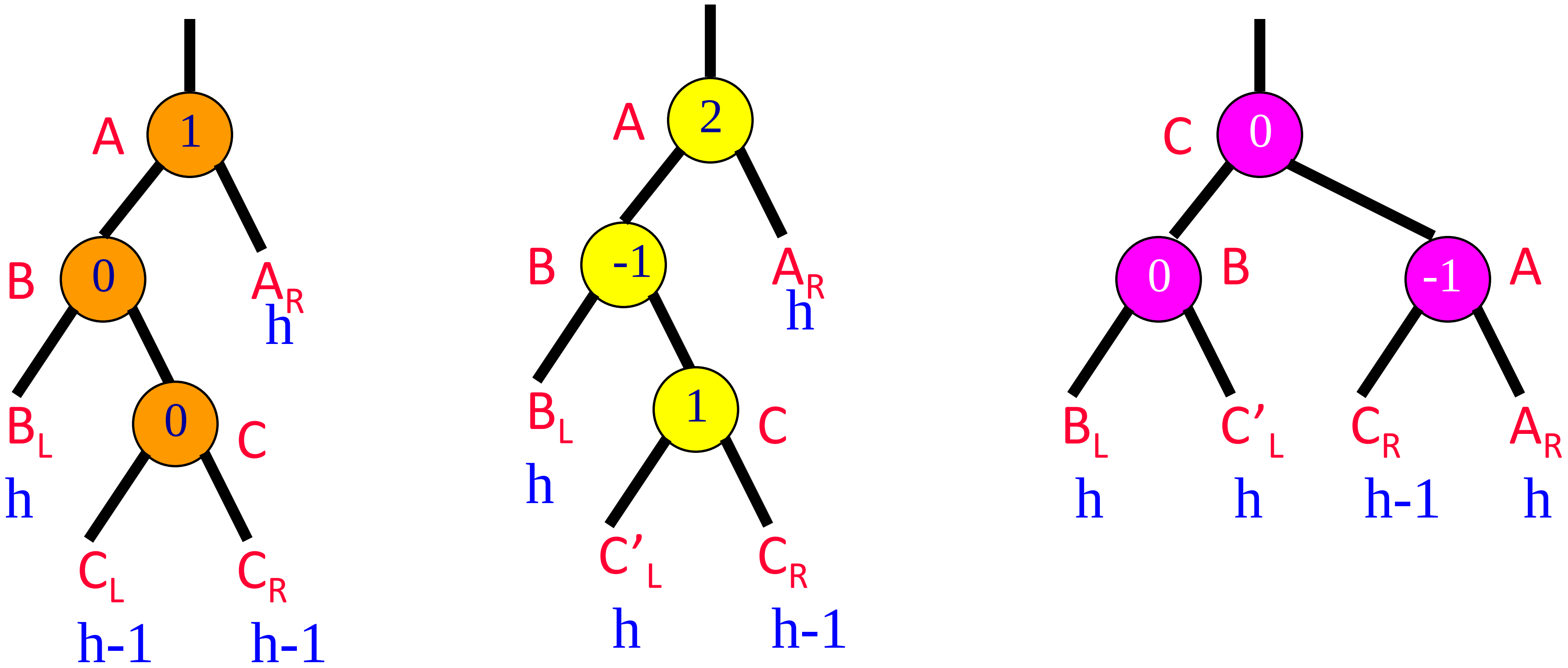
After insertion.



After rotation.

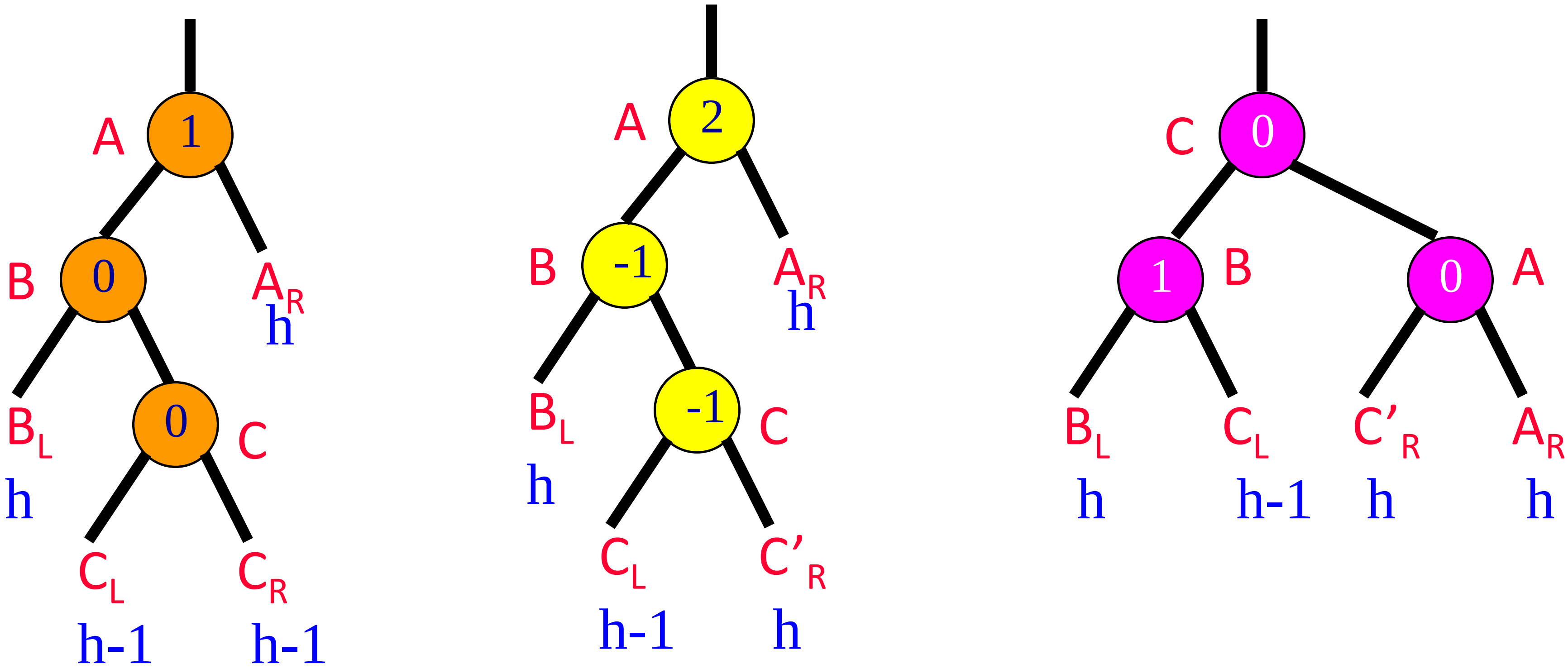
- Subtree height is unchanged.
- No further adjustments to be done.

LR Rotation (case 2)



- Subtree height is unchanged.
- No further adjustments to be done.

LR Rotation (case 3)

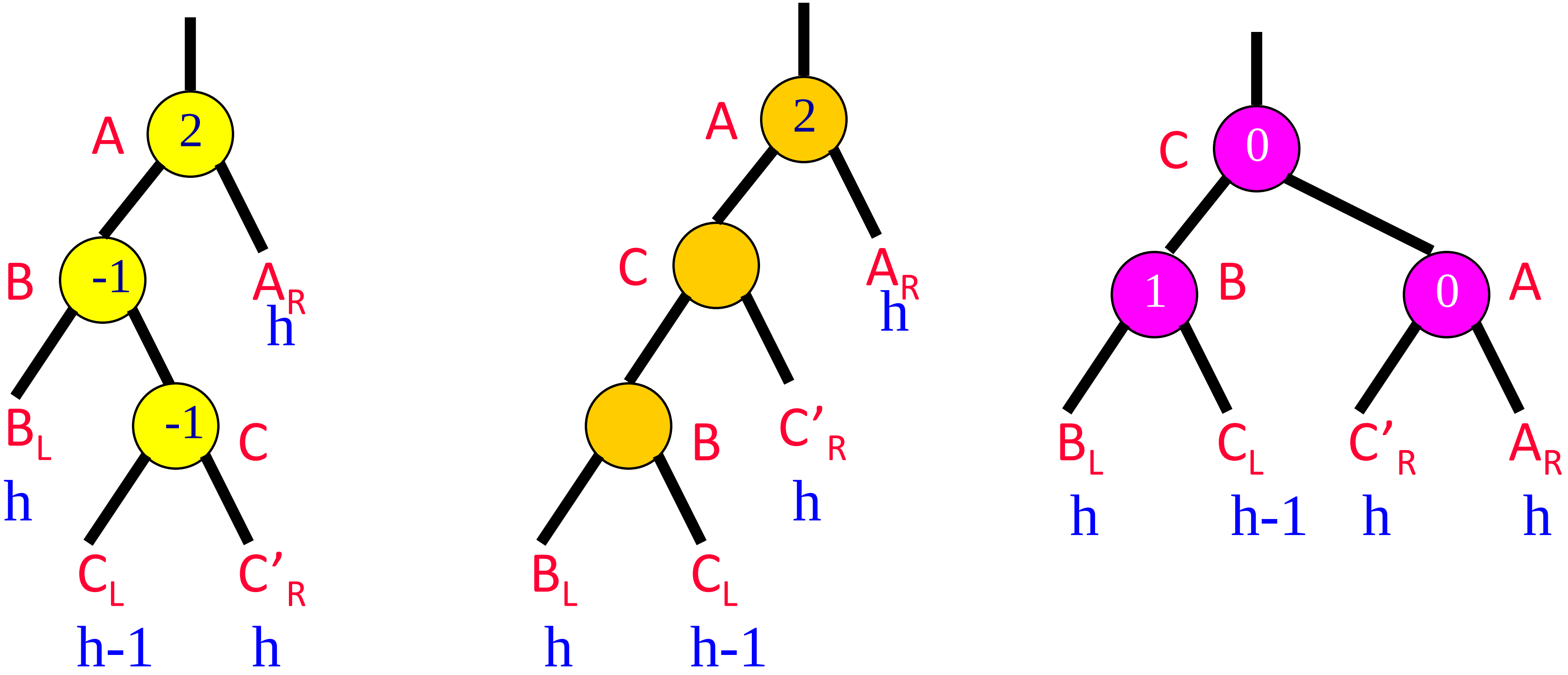


- Subtree height is unchanged.
- No further adjustments to be done.

Single & Double Rotations

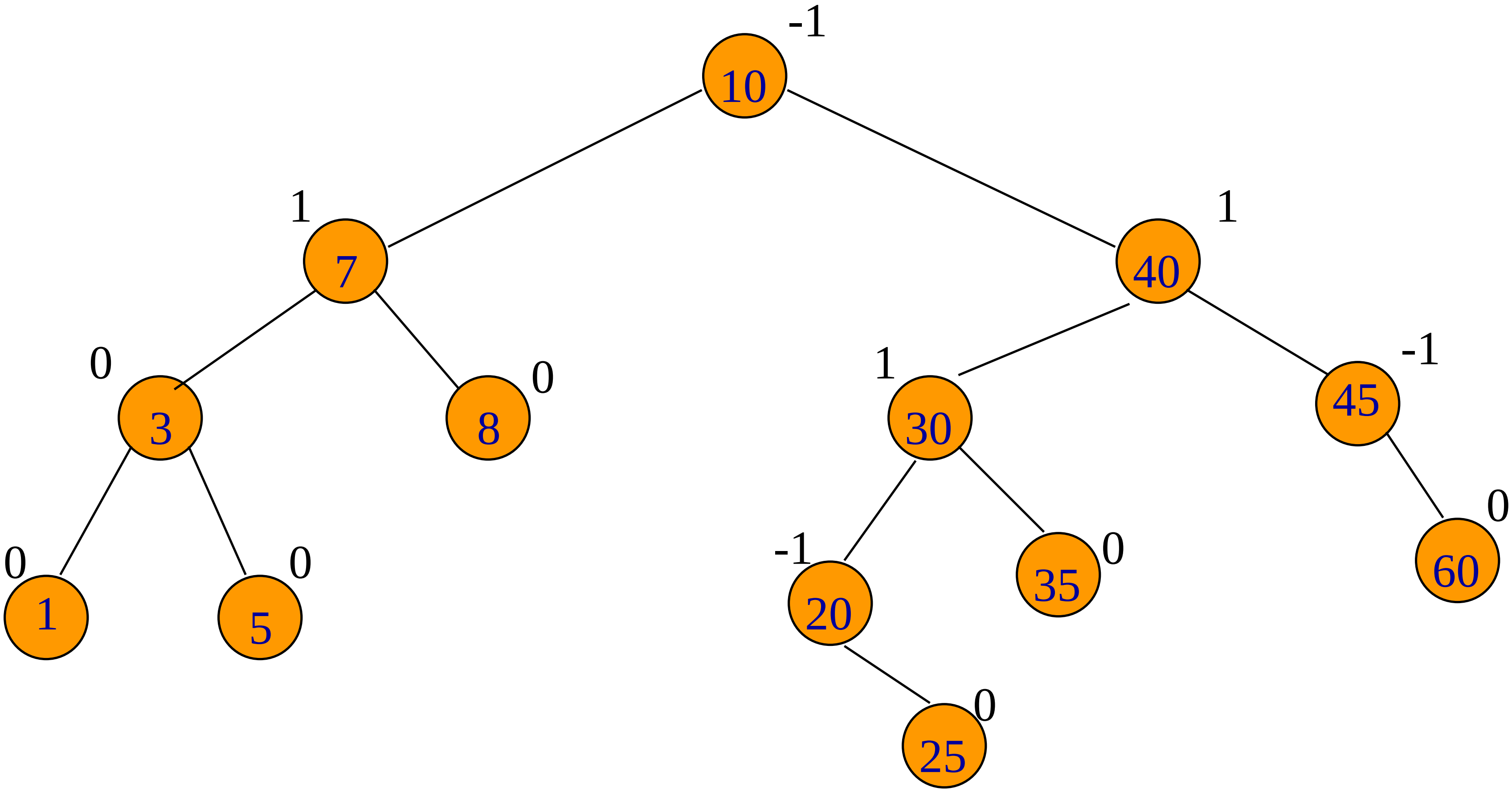
- Single
  - LL and RR
- Double
  - LR and RL
  - LR is RR followed by LL
  - RL is LL followed by RR

LR Is RR + LL



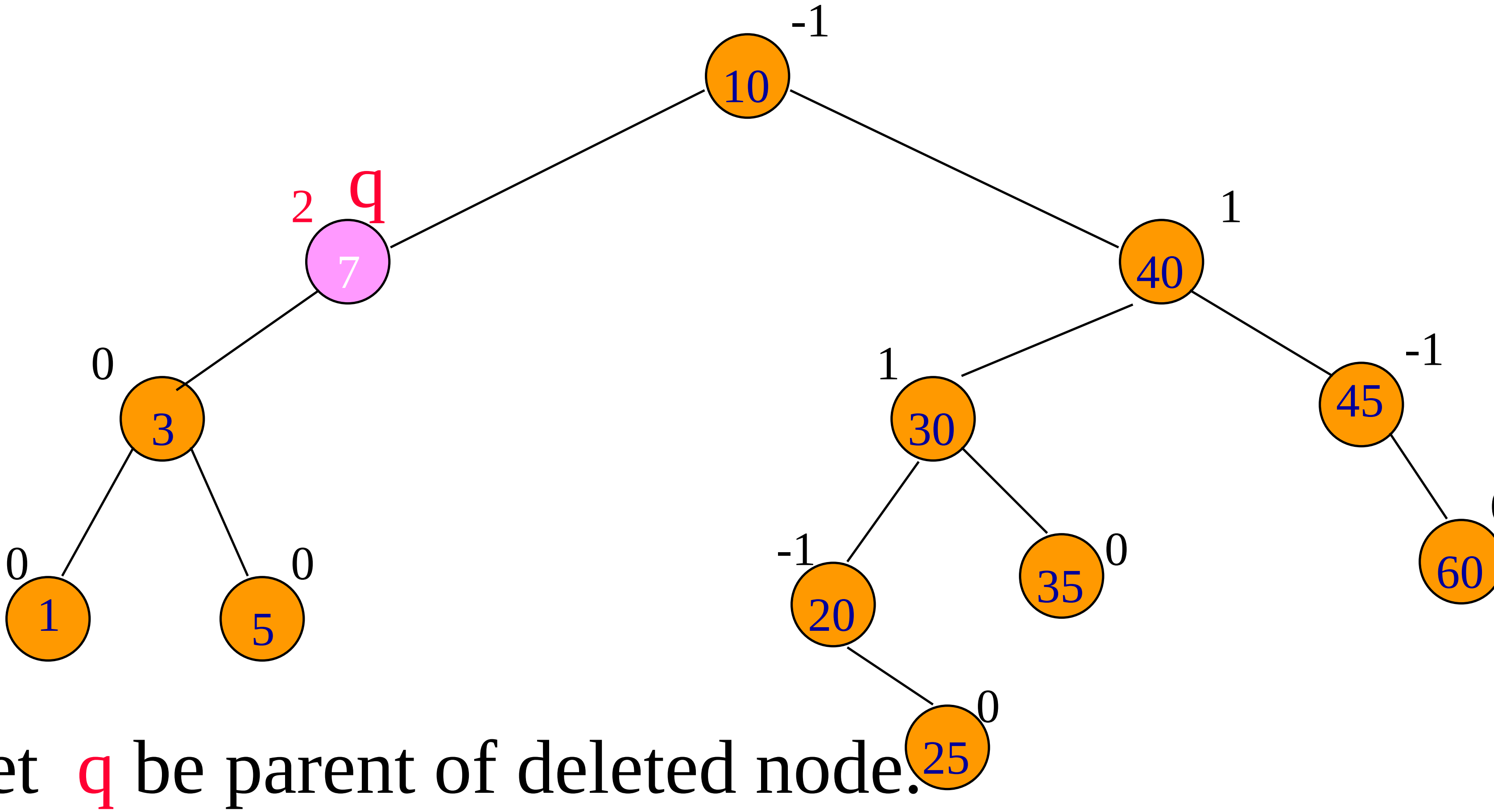
After insertion. After RR rotation. After LL rotation.

Delete An Element



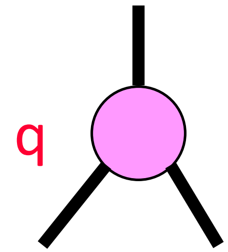
Delete 8.

Delete An Element



- Let q be parent of deleted node.
- Retrace path from q towards root.

New Balance Factor Of q

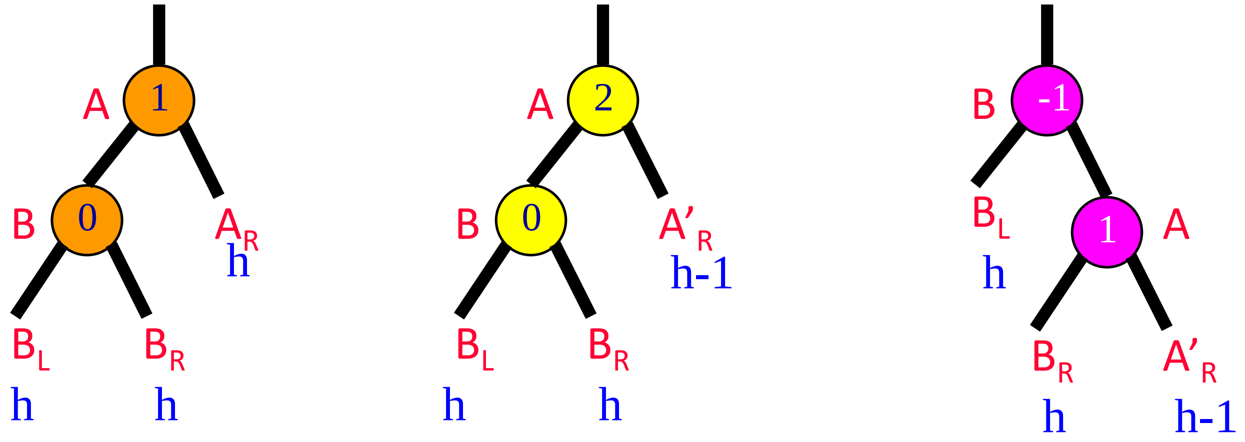


- Deletion from left subtree of q => bf--.
- Deletion from right subtree of q => bf++.
- New balance factor = 1 or –1 => no change in height of subtree rooted at q.
- New balance factor = 0 => height of subtree rooted at q has decreased by 1.
- New balance factor = 2 or –2 => tree is unbalanced at q.

Imbalance Classification

- Let A be the nearest ancestor of the deleted node whose balance factor has become 2 or –2 following a deletion.
- Deletion from left subtree of A => type L.
- Deletion from right subtree of A => type R.
- Type R => new bf(A) = 2.
- So, old bf(A) = 1.
- So, A has a left child B.
  - bf(B) = 0 => R0.
  - bf(B) = 1 => R1.
  - bf(B) = –1 => R-1.

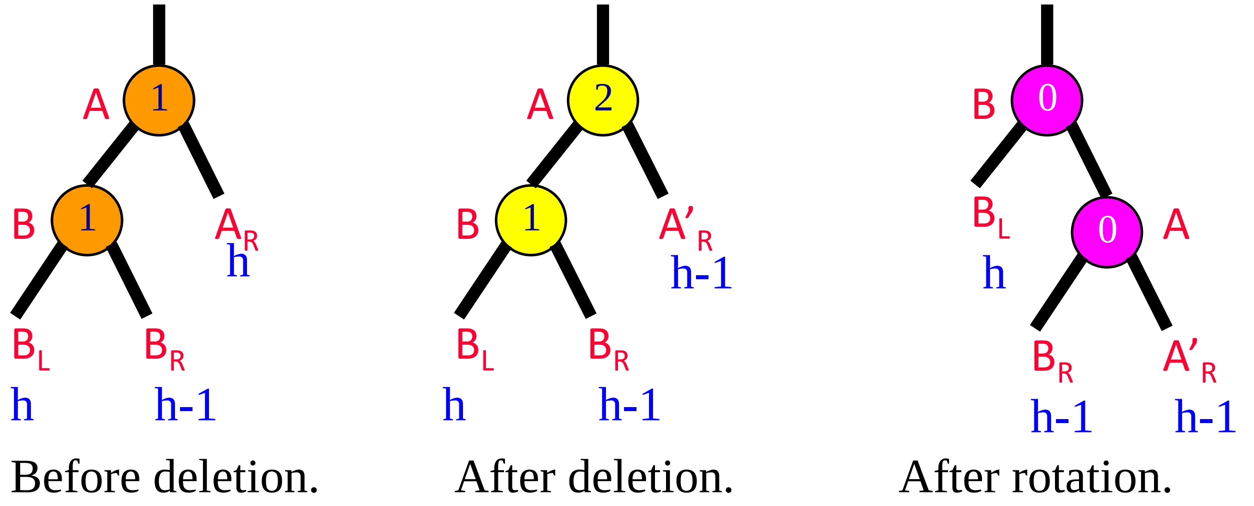
R0 Rotation below



Before deletion. After deletion. After rotation.

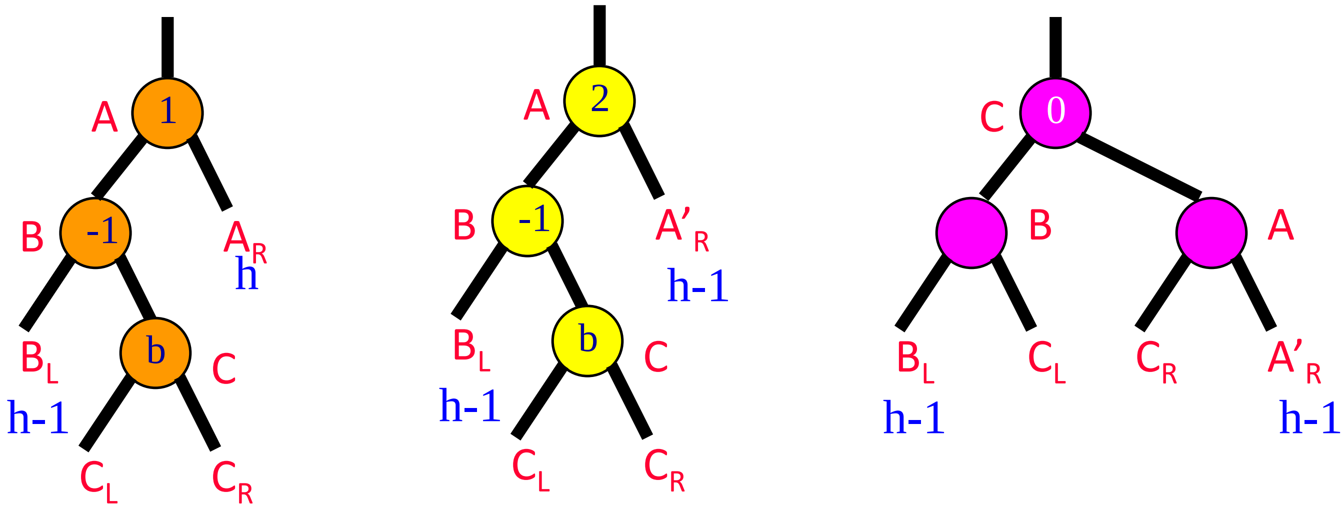
- Subtree height is unchanged.
- No further adjustments to be done.
- Similar to LL rotation.

R1 Rotation



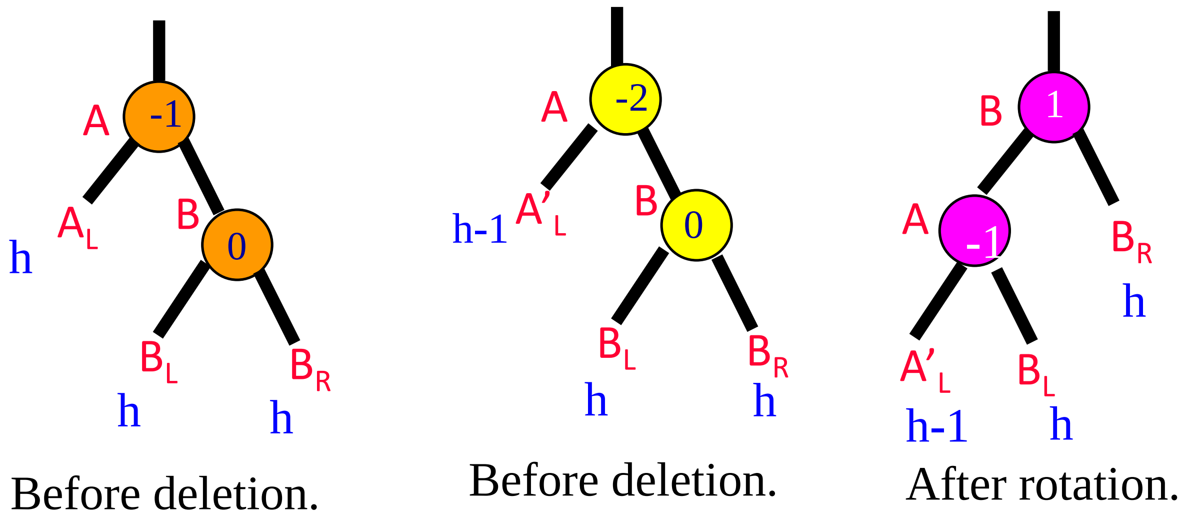
- Subtree height is reduced by 1.
- Must continue on path to root.
- Similar to LL and R0 rotations.

R-1 Rotation



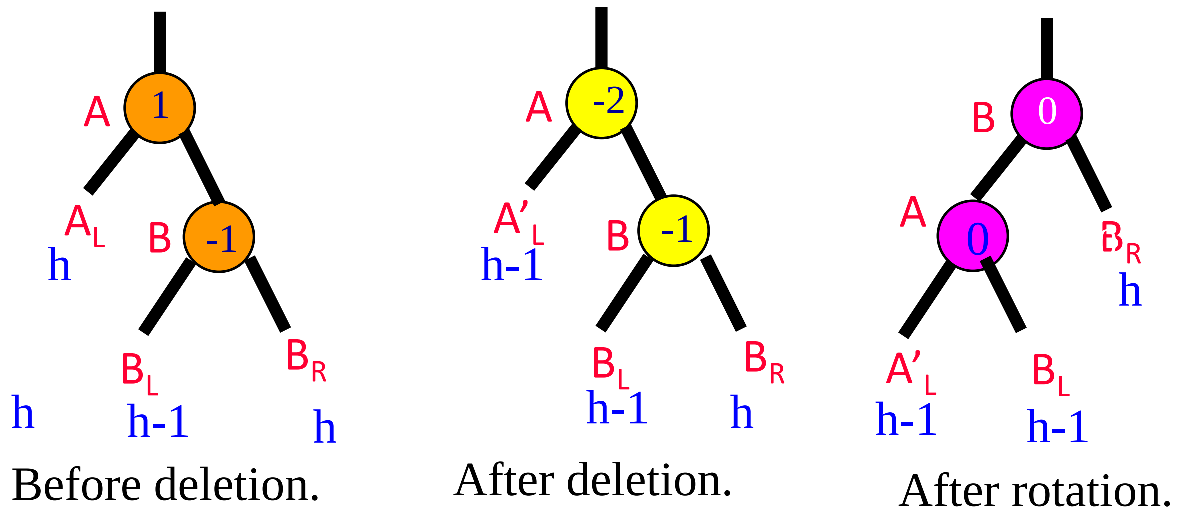
- New balance factor of A and B depends on b.
- b=0 =>$H(C_L)=H(C_R)$, b=1 =>$H(C_L)=H(C_R)+1$, b=-1 =>$H(C_L)=H(C_R)-1$
  =>$H(A'_R)=H(C_L)=H(C_R)=H(B_R)$, b=1=> $H(A'_R)=H(C_R)+1$&$H(B_L)= H(C_L)$, b=-1=> $H(A'_R)=H(C_R)$=&$H(B_L)= H(C_L)+1$
- Subtree height is reduced by 1.
- Must continue on path to root.
- Similar to LR.

Similarly, L0, L1, & L-1 Rotations
L0 Rotation below



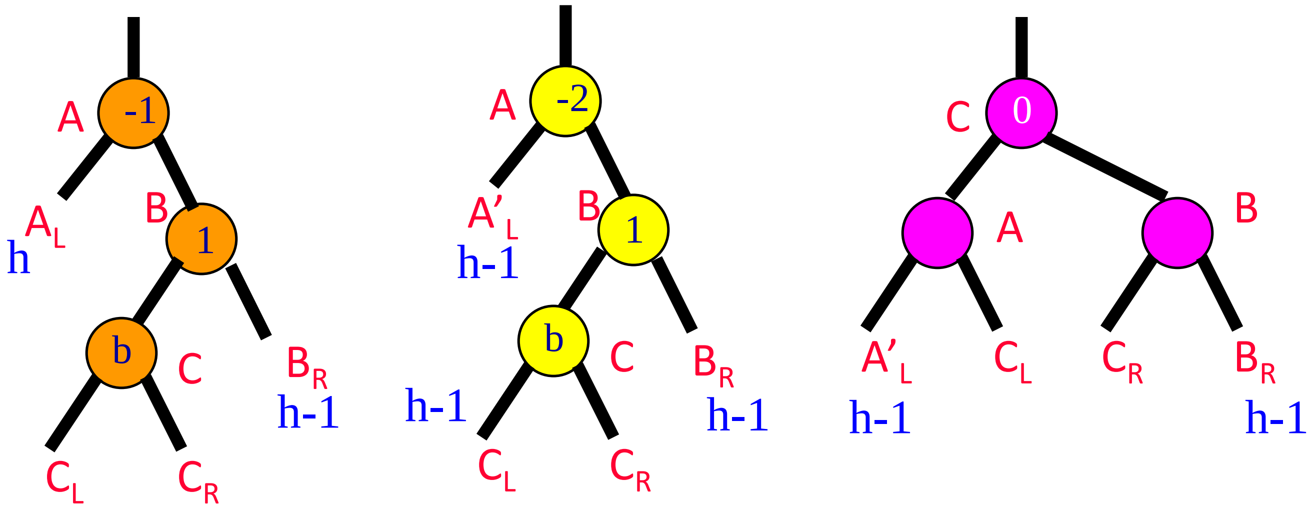
- Subtree height is unchanged.
- No further adjustments to be done.
- Similar to RR rotation.

L1 Rotation



- Subtree height is reduced by 1.
- Must continue on path to root.
- Similar to RR and L0 rotations.

L-1 Rotation



- New balance factor of A and B depends on b.
- b=0 =>$H(C_L)=H(C_R)$, b=1 =>$H(C_L)=H(C_R)+1$, b=-1 =>$H(C_L)=H(C_R)-1$
  =>$H(A'_L)=H(C_L)=H(C_R)=H(B_R)$, b=1=> $H(A'_L)=H(C_L)$=&$H(B_R)= H(C_R)+1$, b=-1=> $H(A'_L)=H(C_L)+1$=&$H(B_R)= H(C_R)$
- Subtree height is reduced by 1.
- Must continue on path to root.
- Similar to RL.

Number Of Rebalancing Rotations

- At most 1 for an insert.
- O(log n) for a delete.

Rotation Frequency

- Insert random numbers.
  - No rotation … 53.4% (approx).
  - LL/RR … 23.3% (approx).
  - LR/RL … 23.2% (approx).