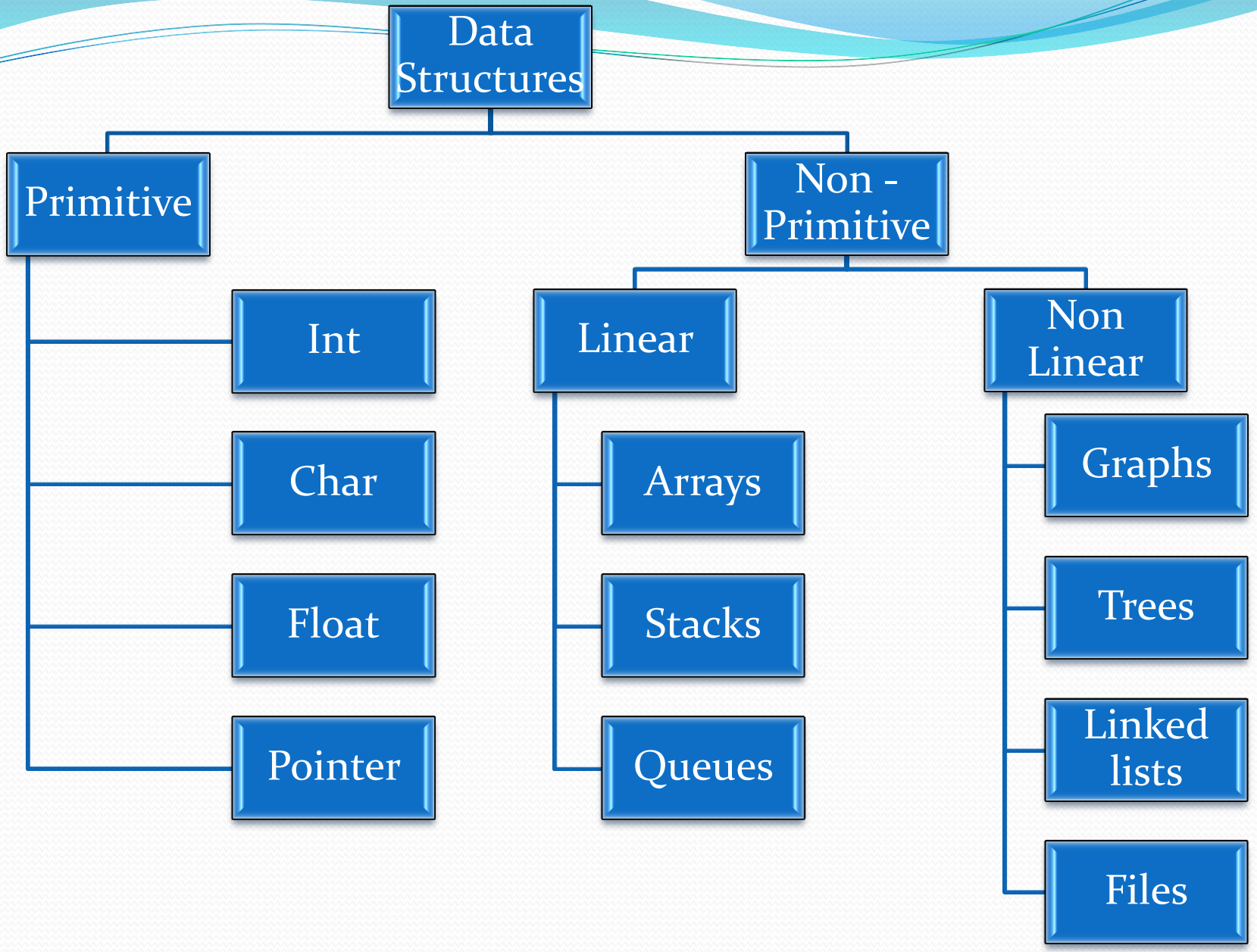
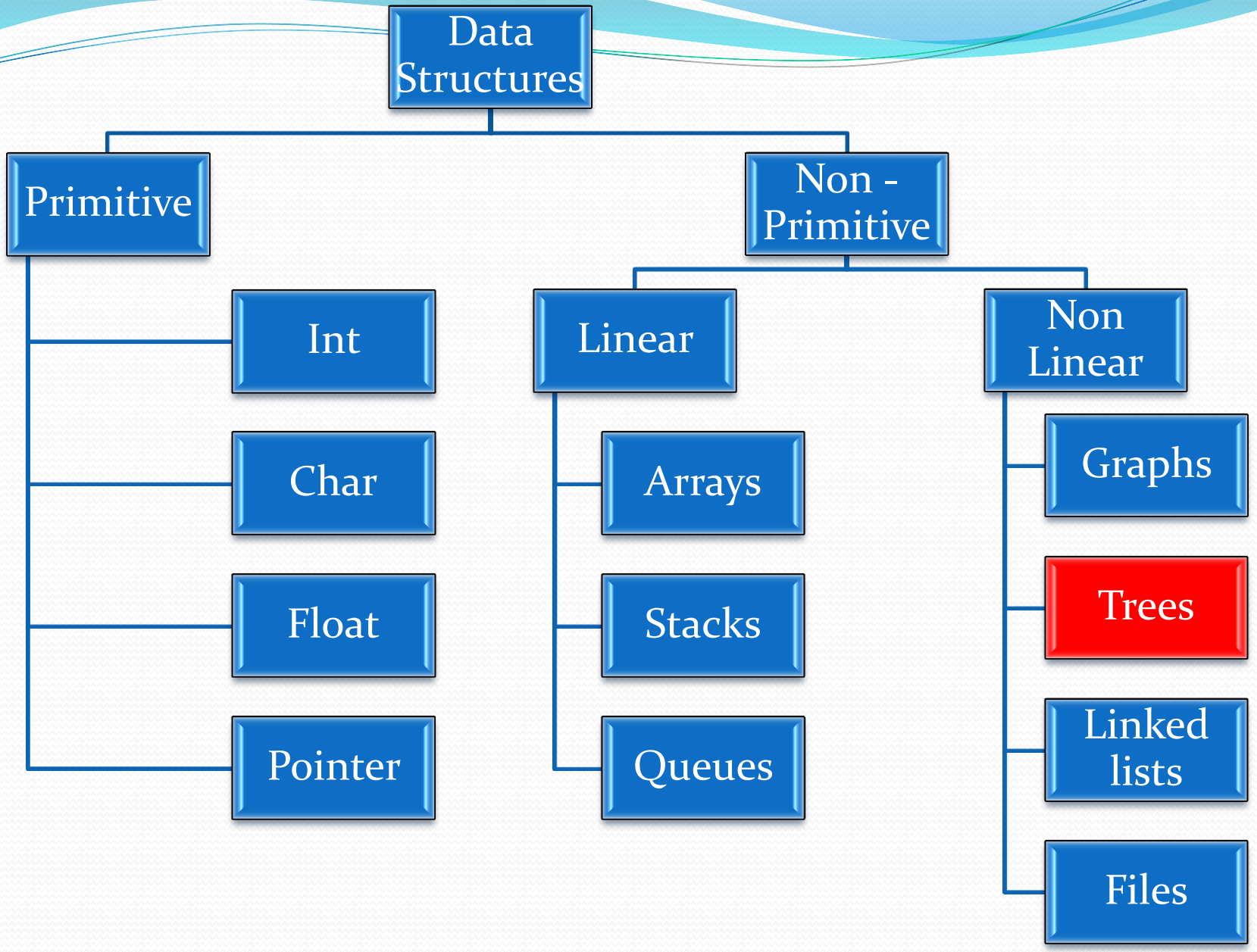

Data Structures
Primitive
Non - Primitive
Int
Char
Float
Pointer
Linear
Non Linear
Arrays
Stacks
Queues
Graphs
Trees
Linked lists
Files

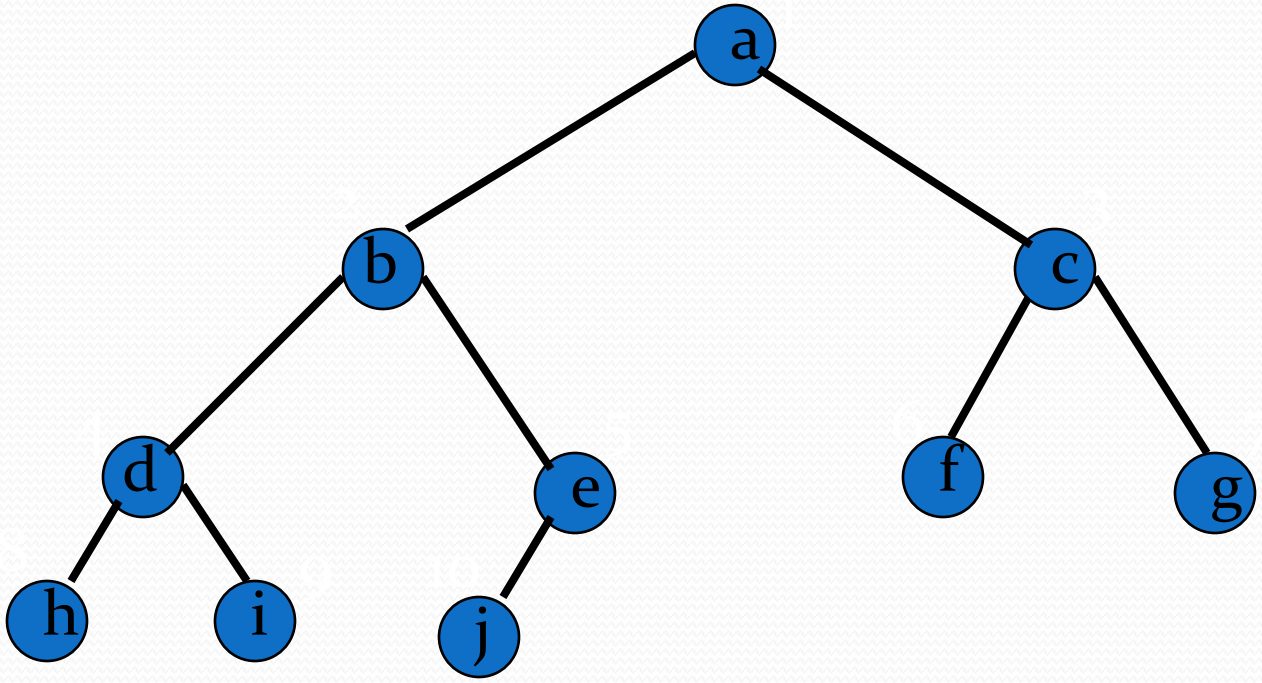

Data Structures
Primitive
Non - Primitive
Int
Char
Float
Pointer
Linear
Non Linear
Arrays
Stacks
Queues
Graphs
Trees
Linked lists
Files

# Binary Tree Representation

- Array representation.
- Linked representation.

# Array Representation

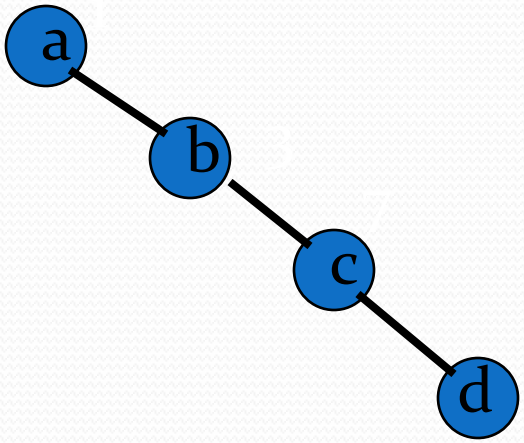
- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in tree[i].




tree[]

| | a | b | c | d | e | f | g | h | i | j |
|---|---|---|---|---|---|---|---|---|---|---|

# Right-Skewed Binary Tree




tree[]

| | a | - | b | - | - | - | c | - | - | - | - | - | - | - | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

- **An n node binary tree needs an array whose length is between n+1 and $2^n$.**

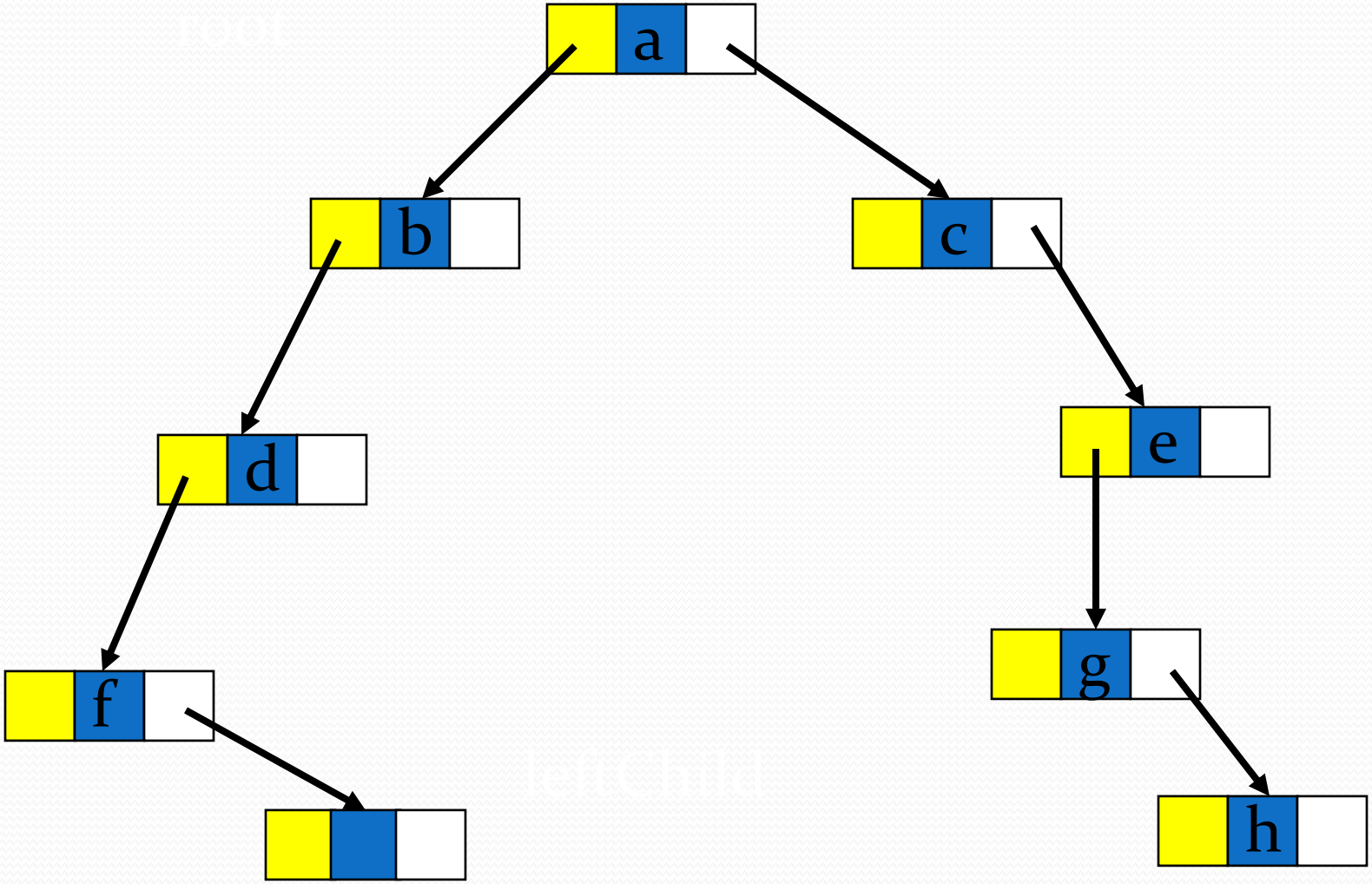
# Linked Representation

- Each binary tree node is represented as an object whose data type is BinaryTreeNode.
- The space required by an n node binary tree is n * (space required by one node).

# Link Representation of Binary Tree

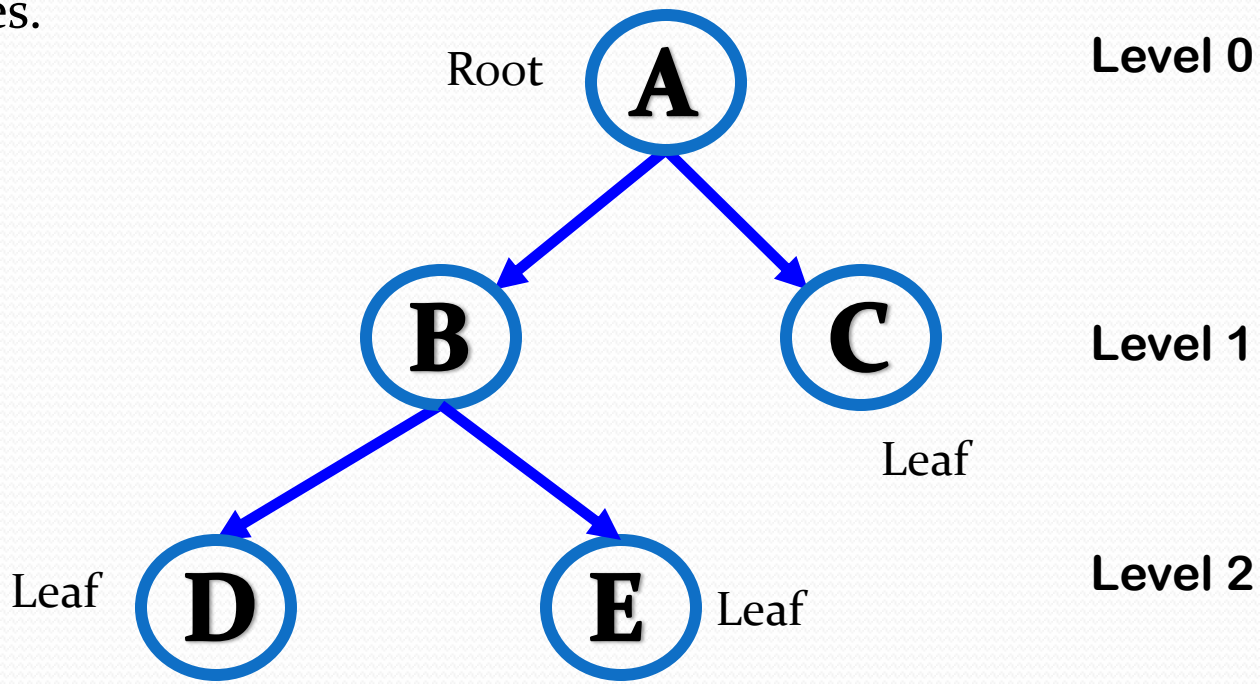
```
  typedef struct node
{
     int data;
     struct node *lc,*rc;
};
```

# Linked Representation Example

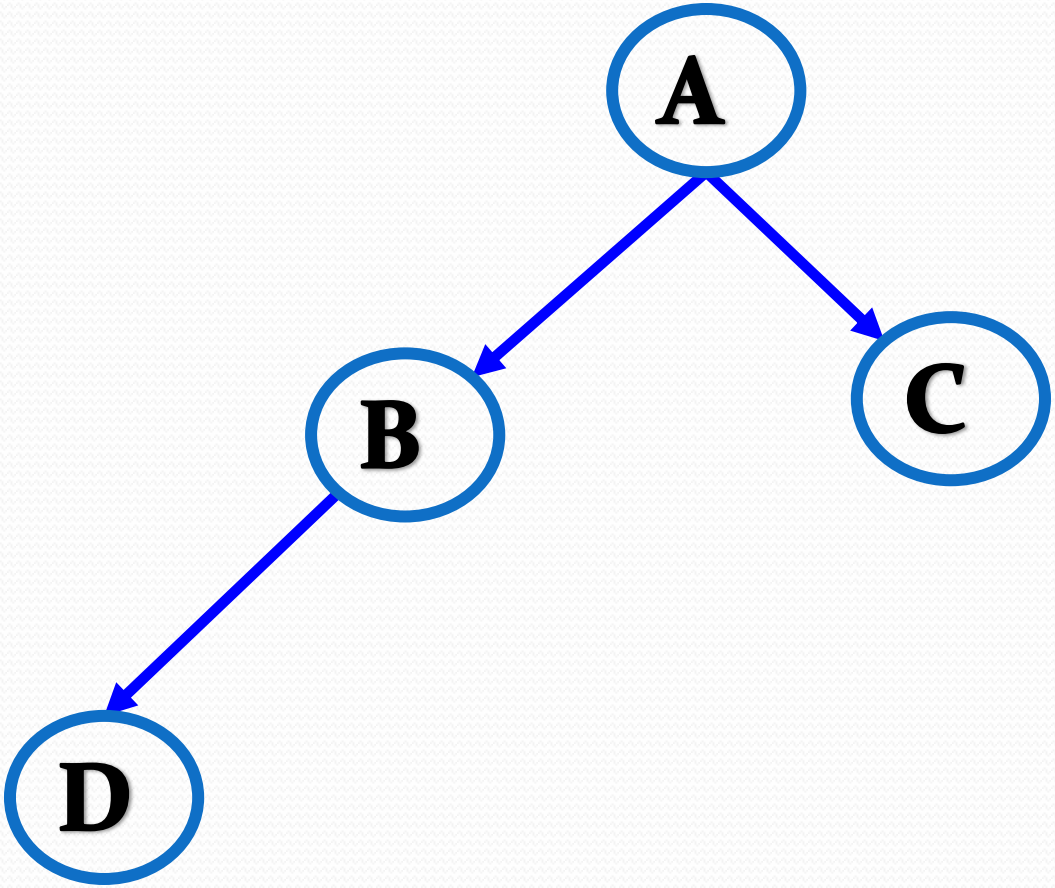


# TREE

- A **tree** is a hierarchical representation of a finite set of one or more data items such that:
  - There is a special node called the root of the tree.
  - The nodes other than the root node form an ordered pair of disjoint subtrees.



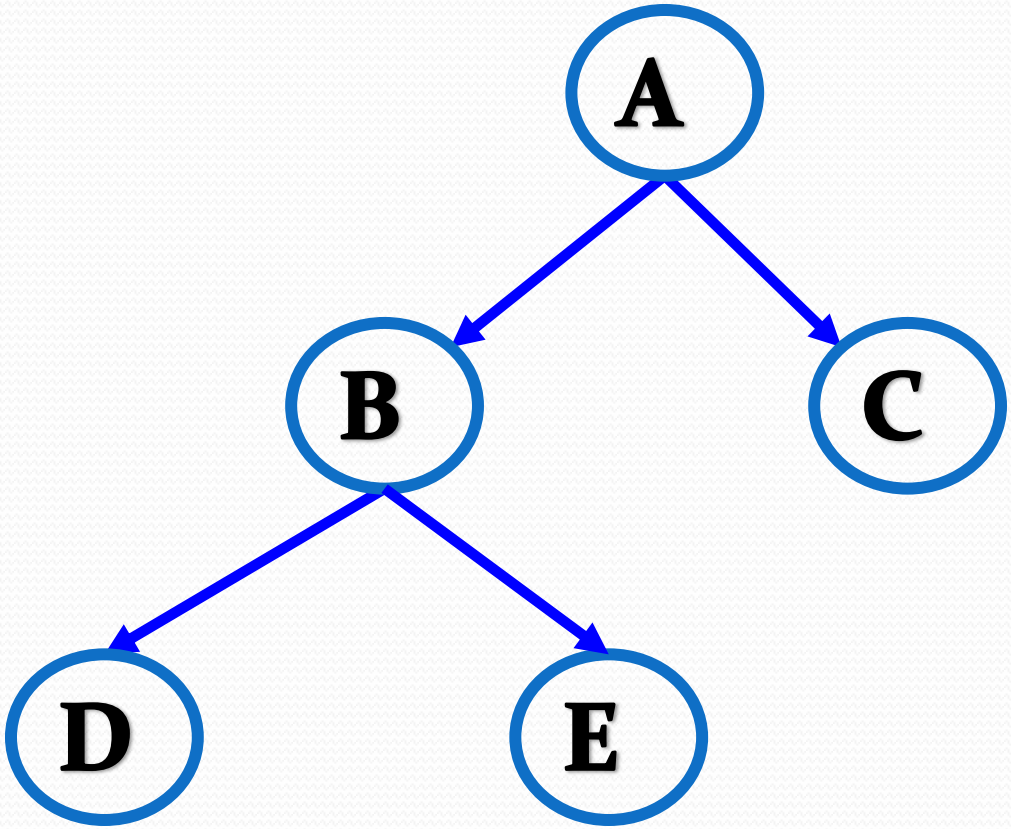
# Binary Tree



# Binary Tree

*Binary Tree is a rooted tree in which root can have maximum two children such that each of them is again a binary tree. That means, there can be 0,1, or 2 children of any node.*

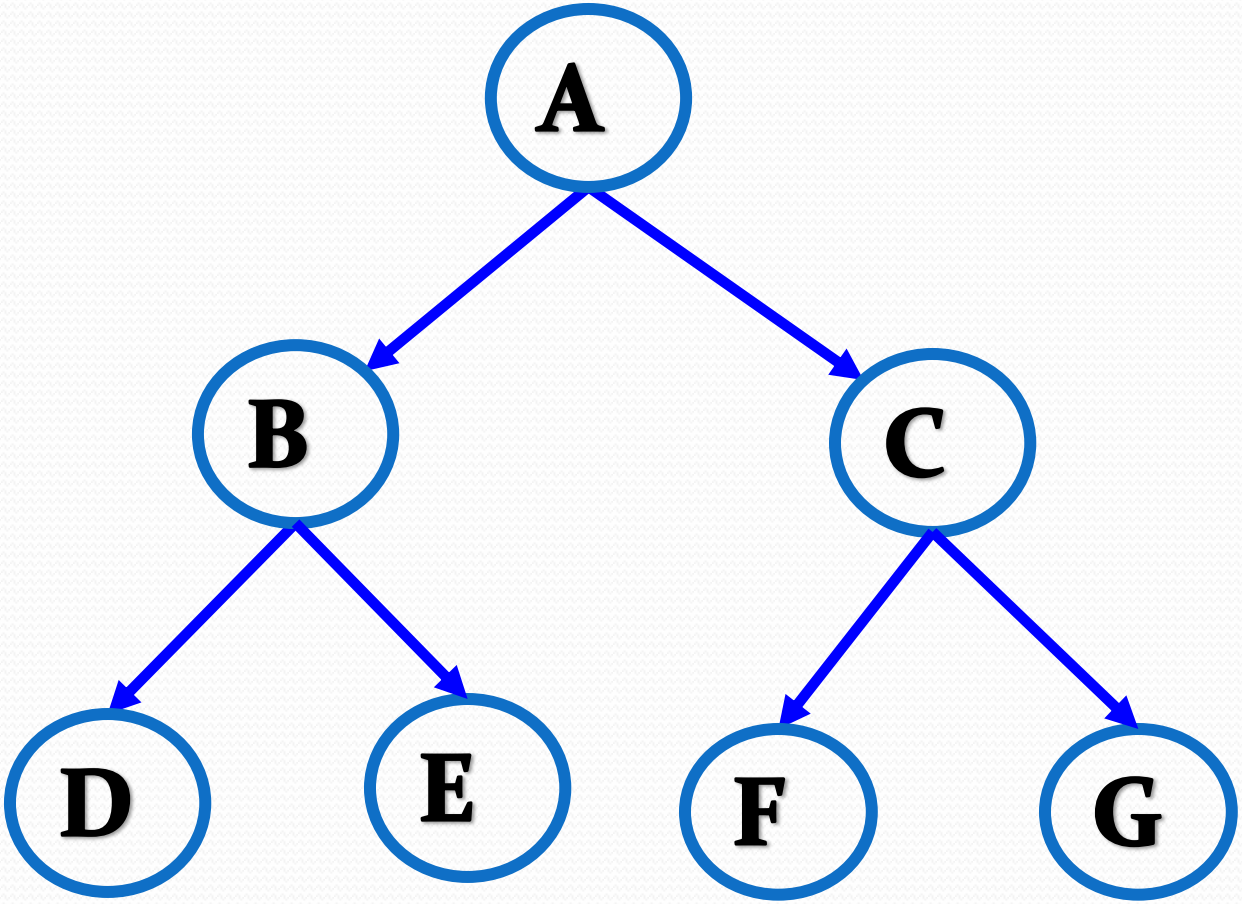
# Strict Binary Tree



# Strict Binary Tree

*Strict Binary Tree is a Binary tree in which root can have exactly two children or no children at all. That means, there can be 0 or 2 children of any node.*

# Complete Binary Tree

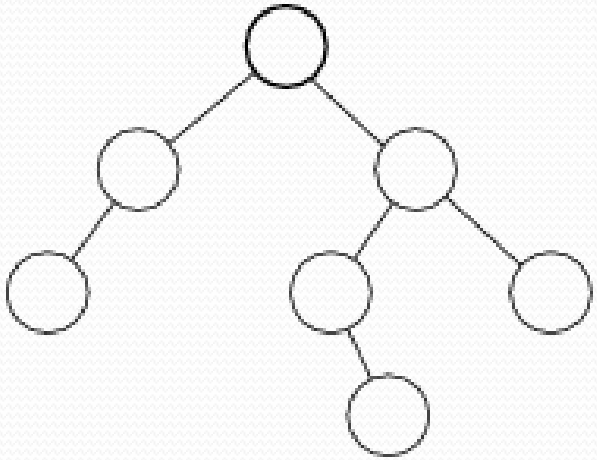


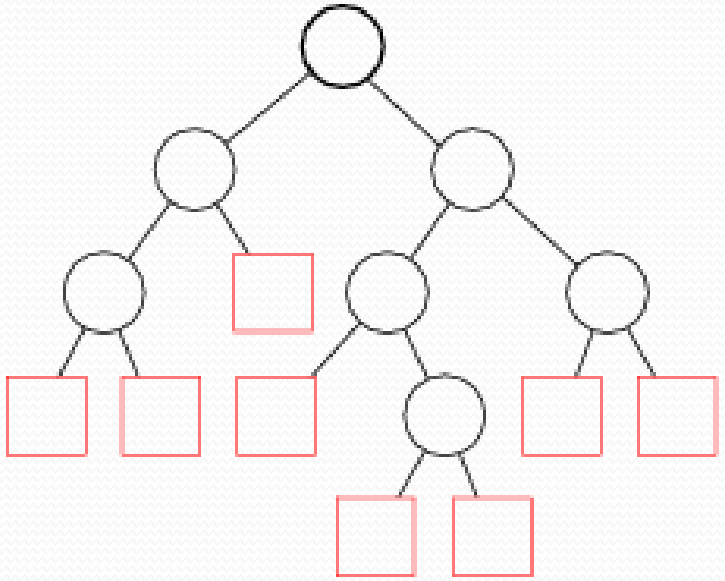
# Complete Binary Tree

*Complete Binary Tree is a Strict Binary tree in which every leaf node is at same level. That means, there are equal number of children in right and left subtree for every node.*

# Extended Binary Tree

- A *binary tree* with special *nodes* replacing every *null* subtree. Every regular node has two *children*, and every special node has no children.





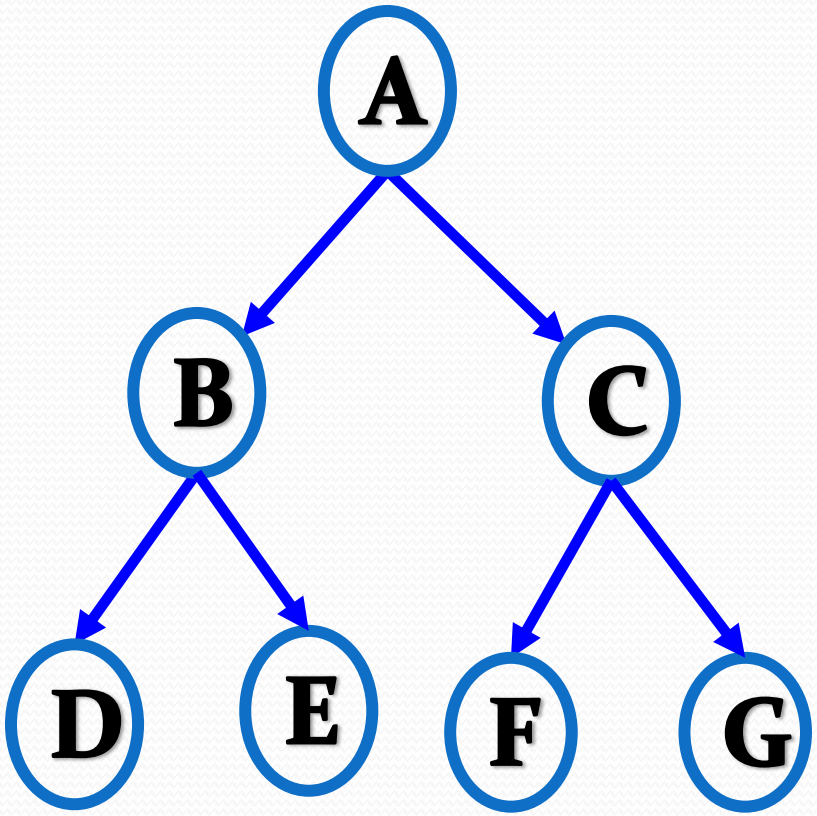
# Extended Binary Tree

- An **extended binary tree** is a transformation of any **binary tree** into a complete **binary tree**.

- This transformation consists of replacing every null subtree of the original **tree** with “special nodes.”

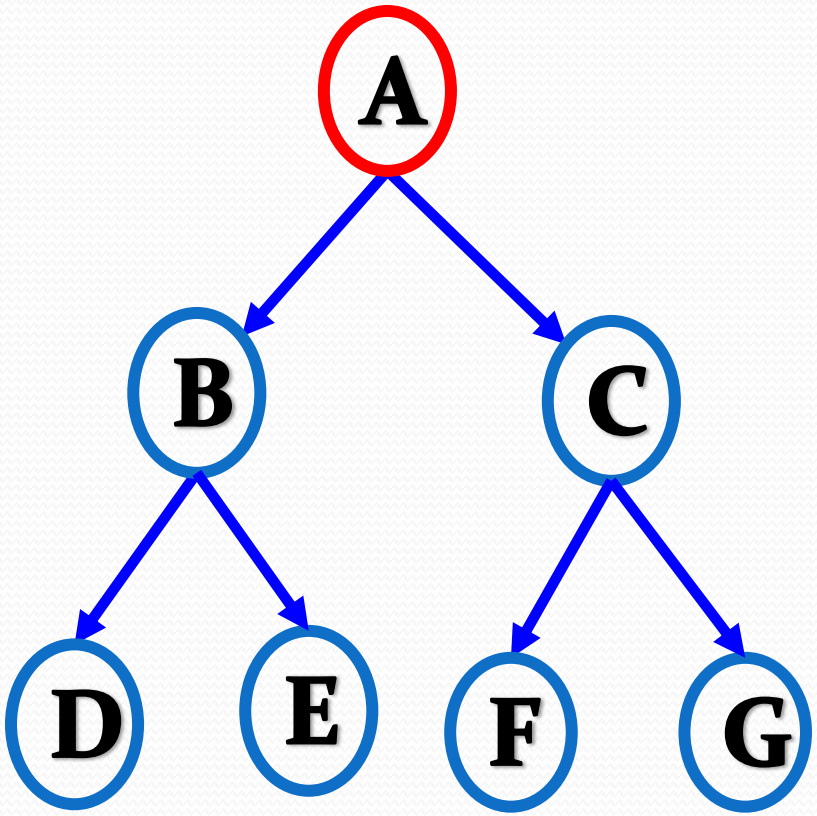
- The nodes from the original **tree** are then called as internal nodes, while the “special nodes” are called as external nodes.

# Tree Traversal : Pre order




- Pre order (N L R)

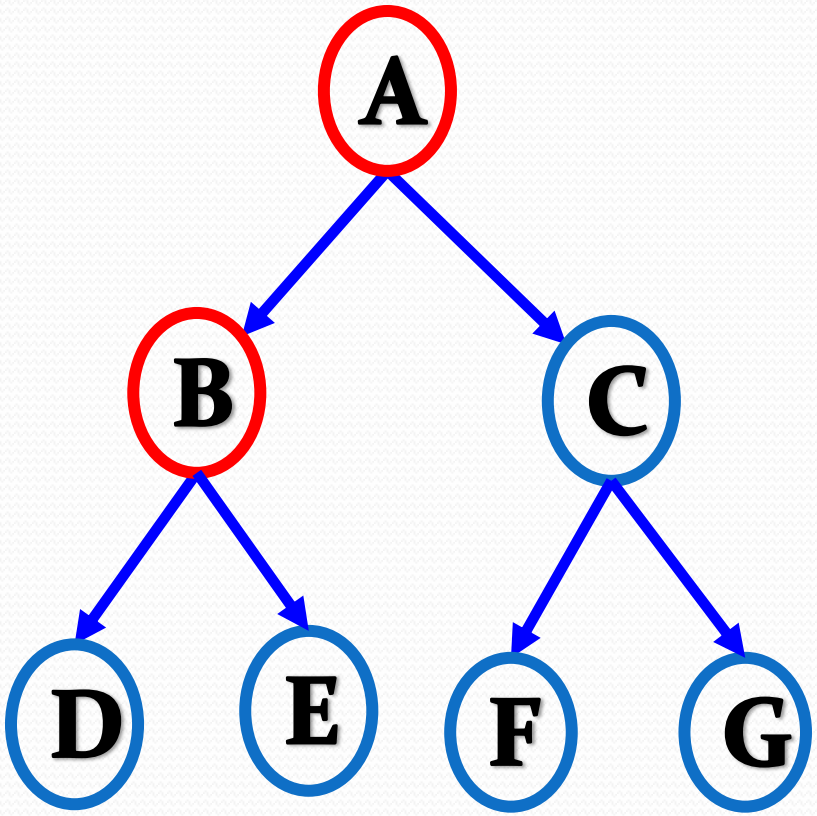
# Tree Traversal : Pre order




- Pre order (N L R)

  A

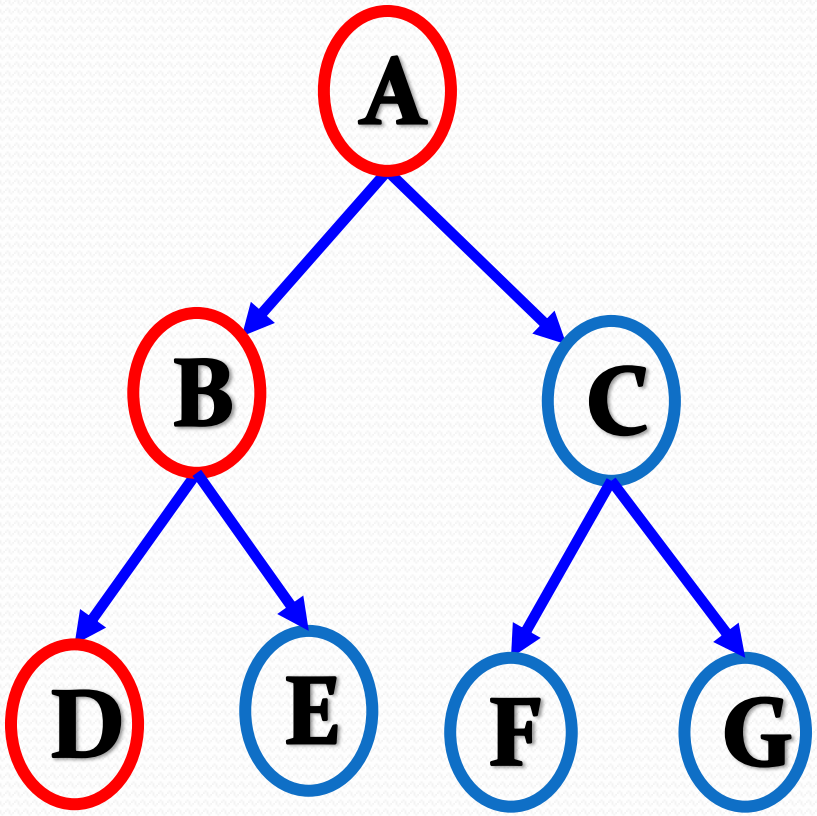
# Tree Traversal : Pre order




- Pre order (N L R)

  A, B

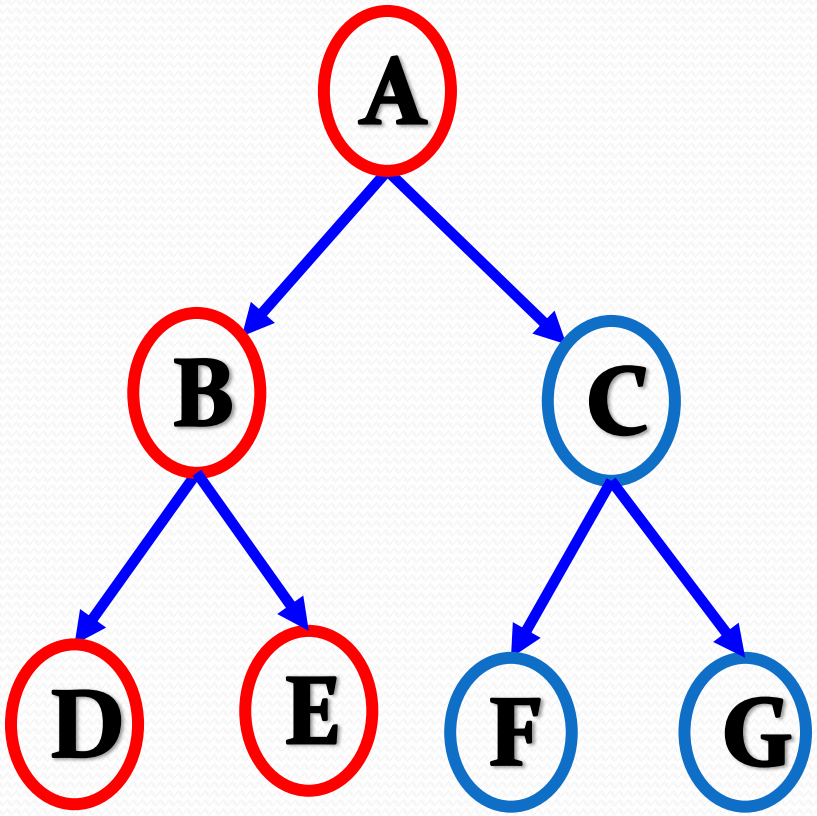
# Tree Traversal : Pre order




- Pre order (N L R)

  A, B, D

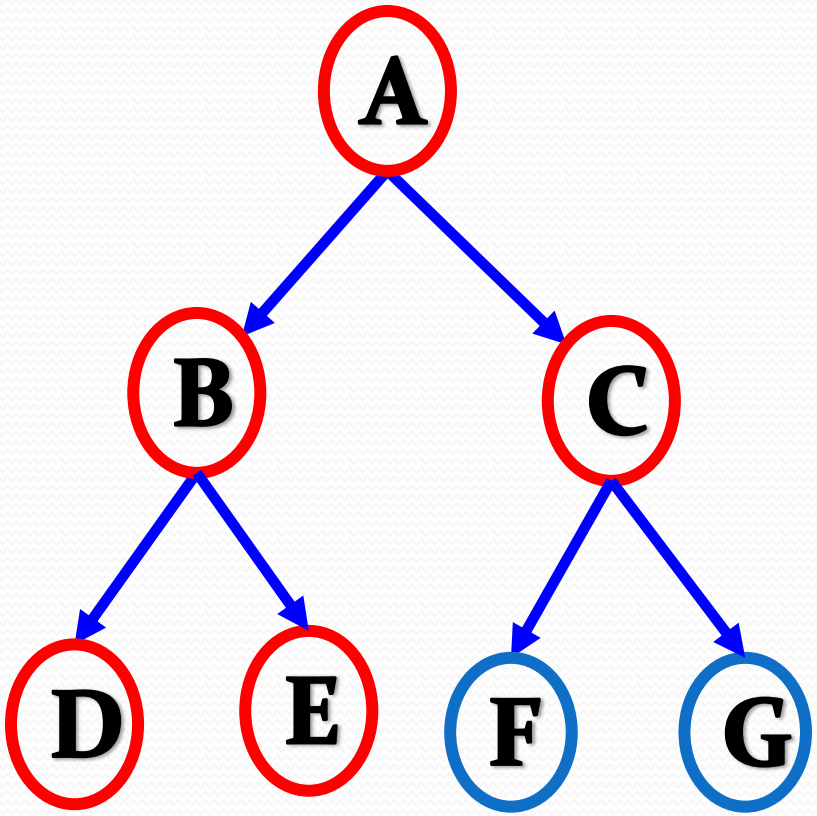
# Tree Traversal : Pre order




- Pre order (N L R)

  A, B, D, E

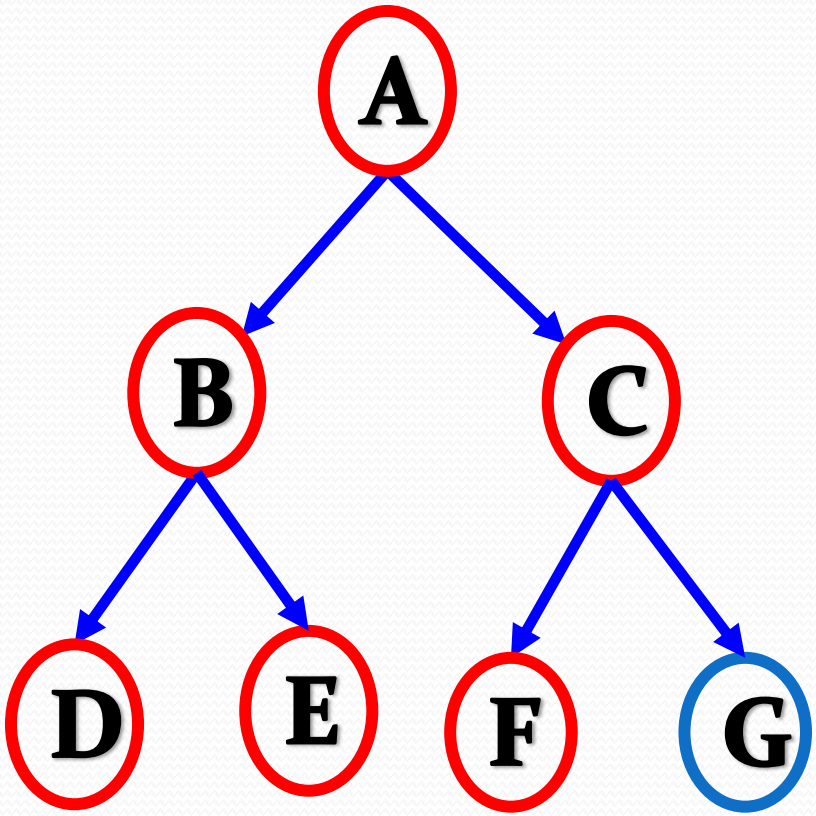
# Tree Traversal : Pre order




- Pre order (N L R)

  A, B, D, E, C

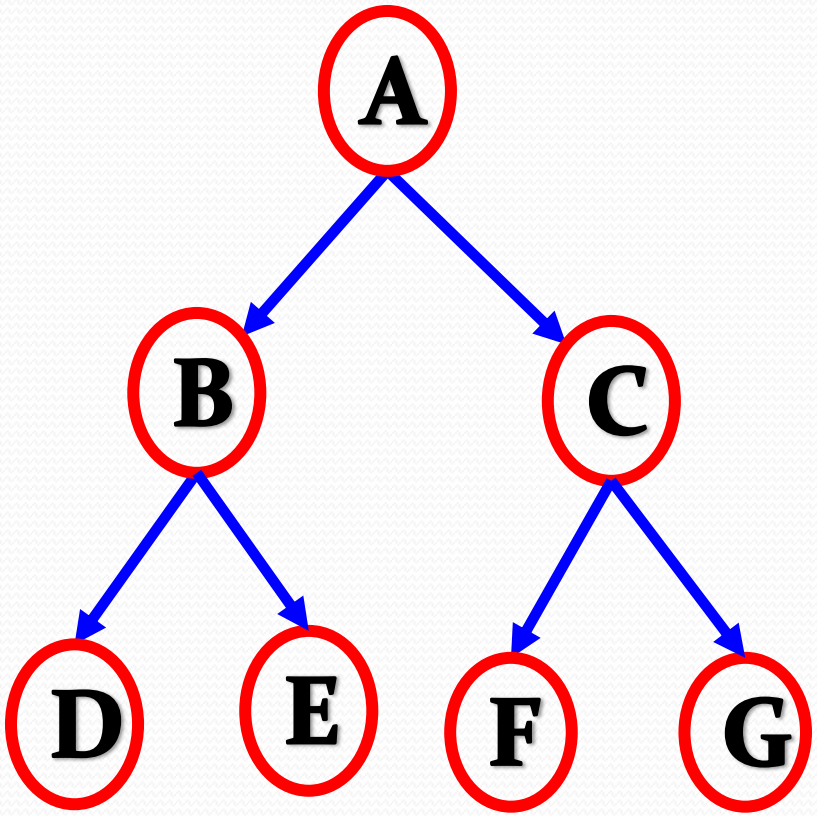
# Tree Traversal : Pre order




- Pre order (N L R)

  A, B, D, E, C, F

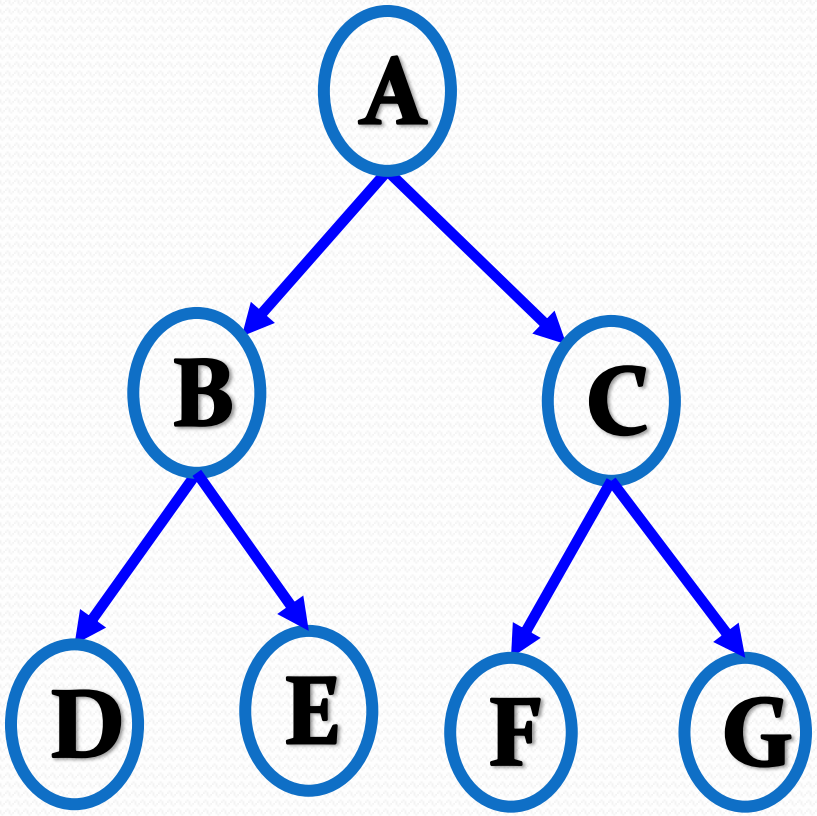
# Tree Traversal : Pre order




- Pre order (N L R)

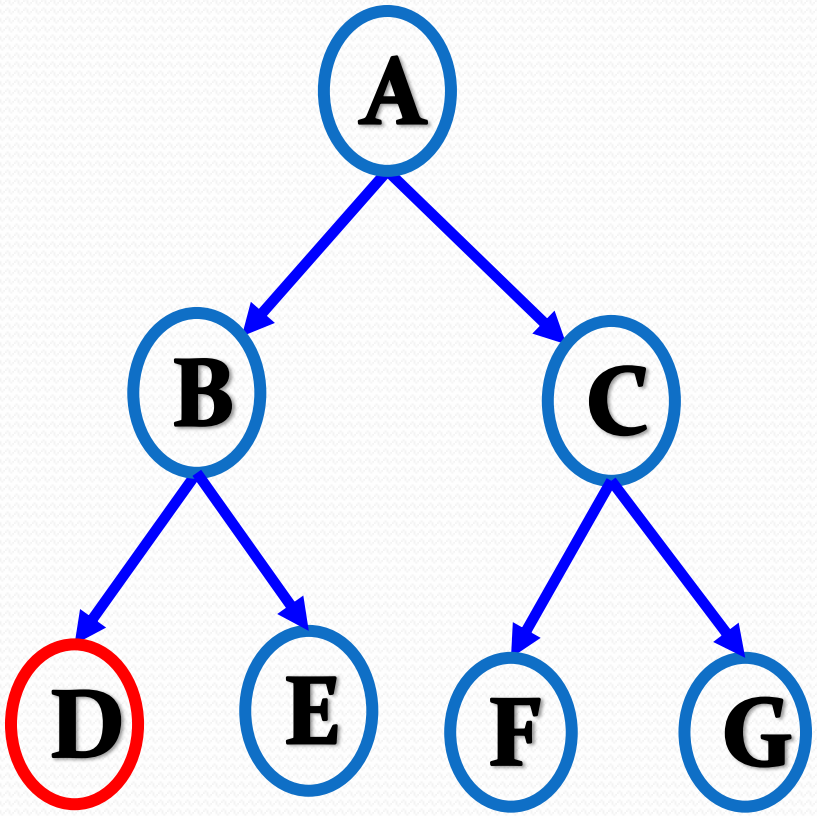
  A, B, D, E, C, F, G

# Tree Traversal : In order




- In order (L N R)

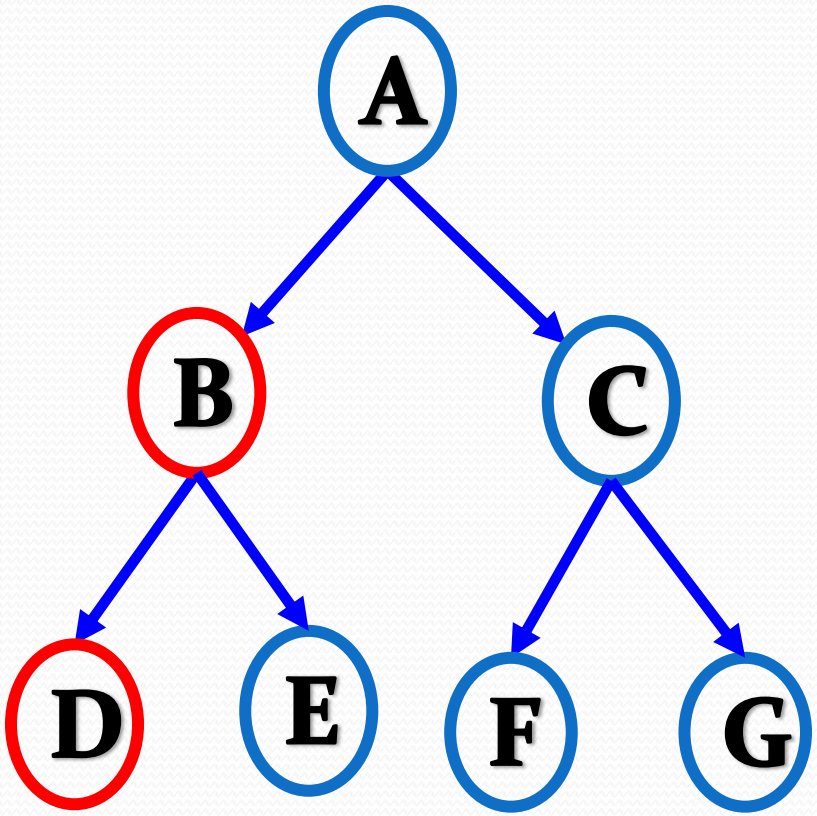
# Tree Traversal : In order




- In order (L N R)

  D

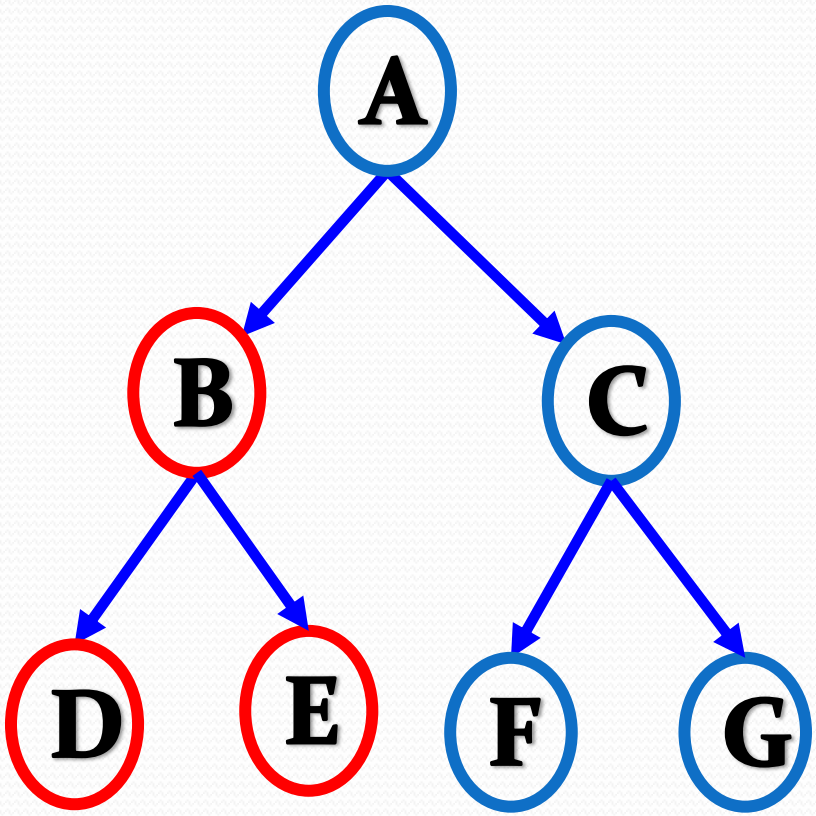
# Tree Traversal : In order




- In order (L N R)

  D, B

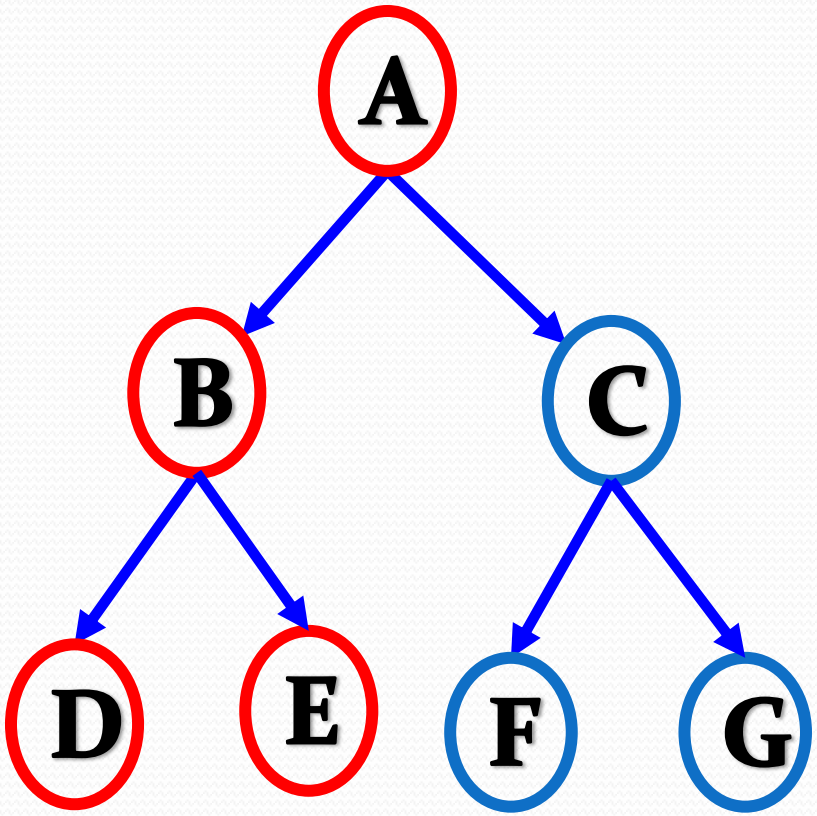
# Tree Traversal : In order




- In order (L N R)

  D, B, E

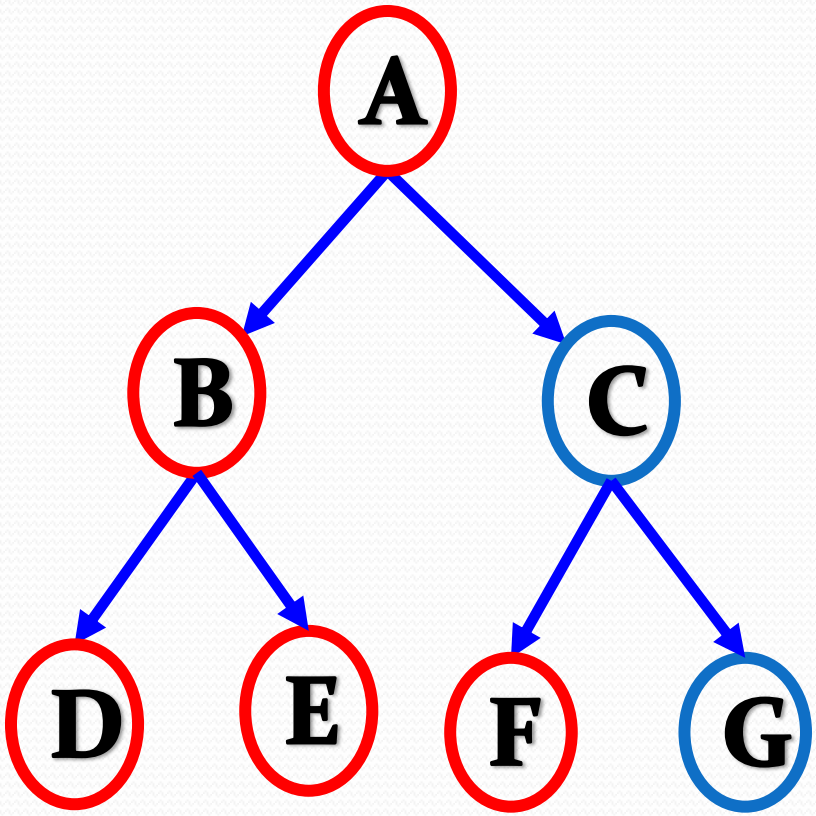
# Tree Traversal : In order




- In order (L N R)

  D, B, E, A

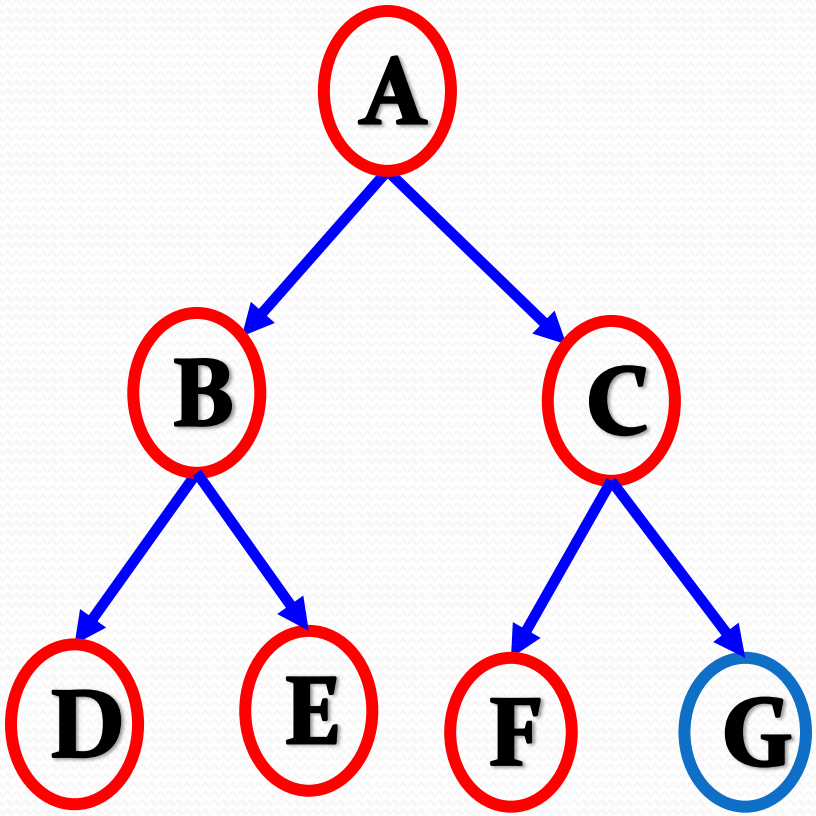
# Tree Traversal : In order




- In order (L N R)

  D, B, E, A, F

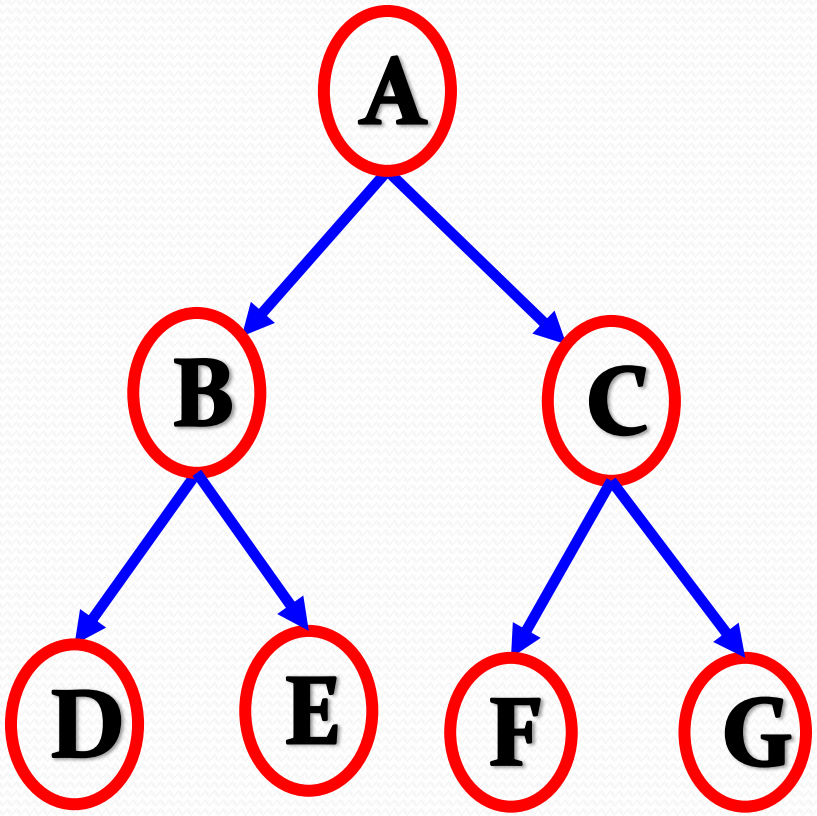
# Tree Traversal : In order




- In order (L N R)

  D, B, E, A, F, C

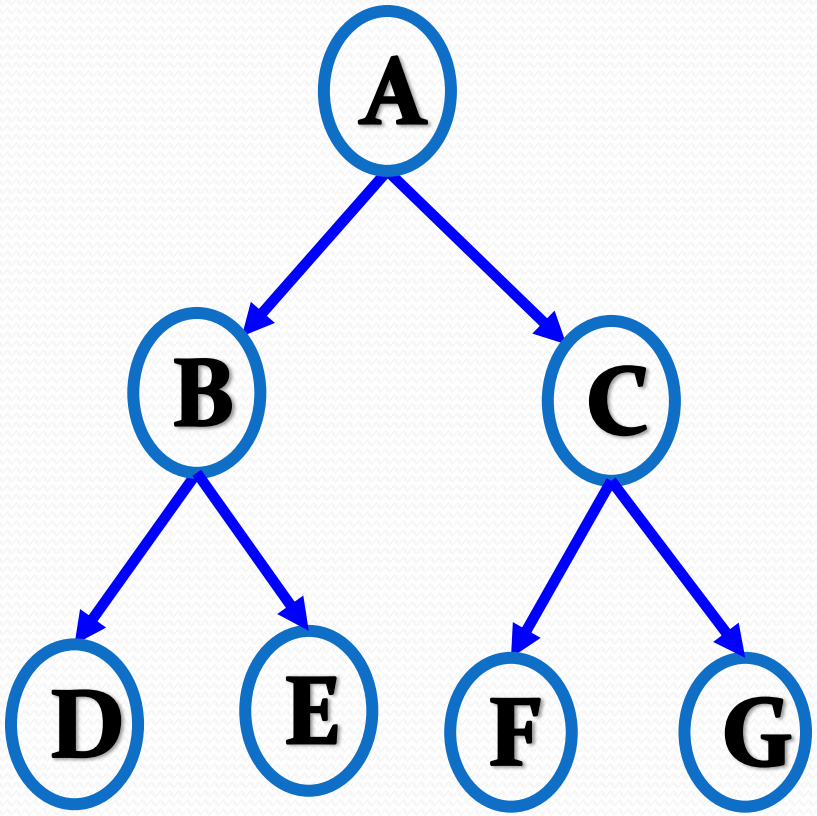
# Tree Traversal : In order




- In order (L N R)

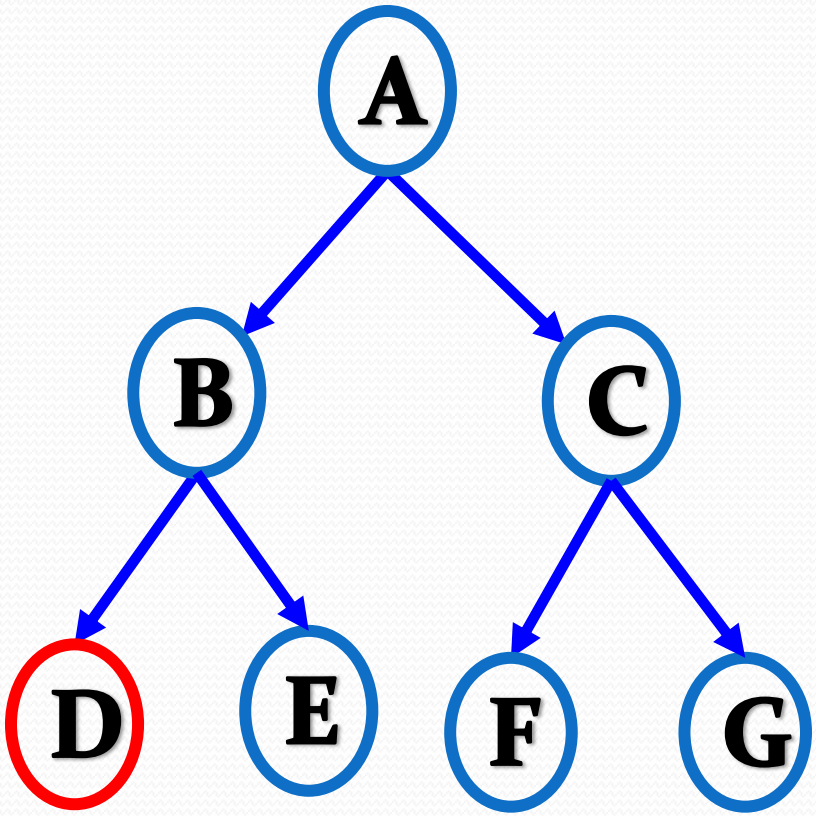
  D, B, E, A, F, C, G

# Tree Traversal : Post order




- Post order (L R N)

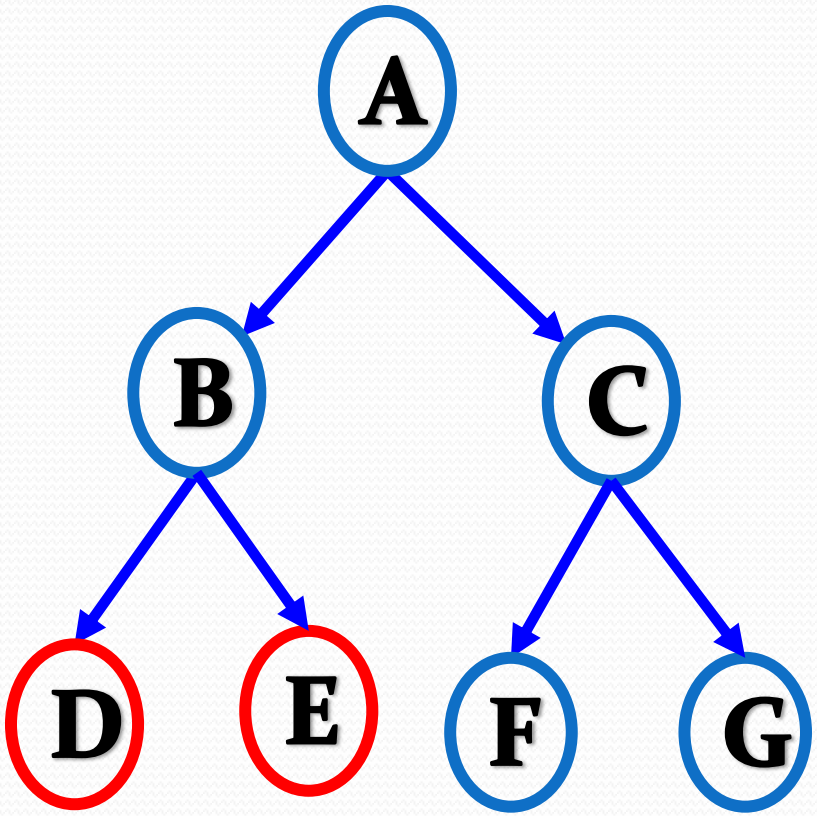
# Tree Traversal : Post order




- Post order (L R N)

  D

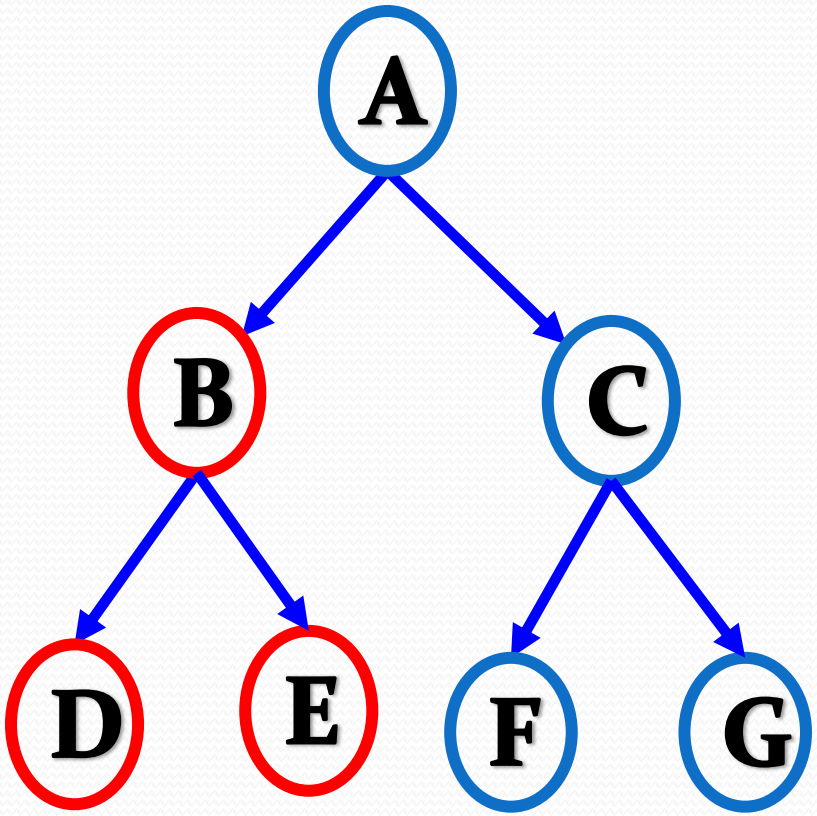
# Tree Traversal : Post order




- Post order (L R N)

  D, E

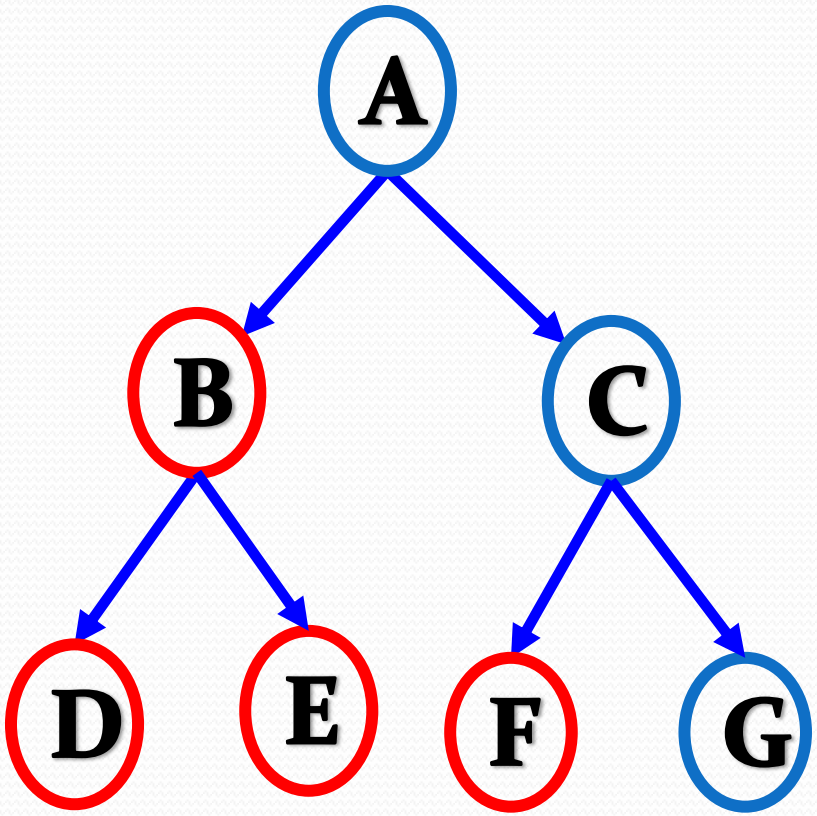
# Tree Traversal : Post order




- Post order (L R N)

  D, E, B

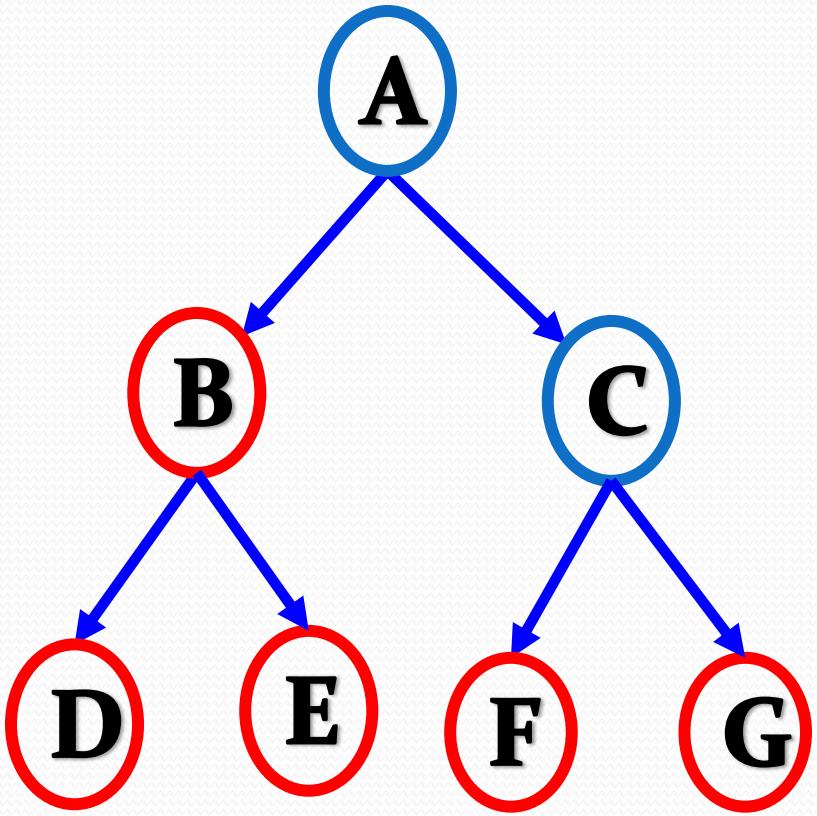
# Tree Traversal : Post order




- Post order (L R N)

  D, E, B, F

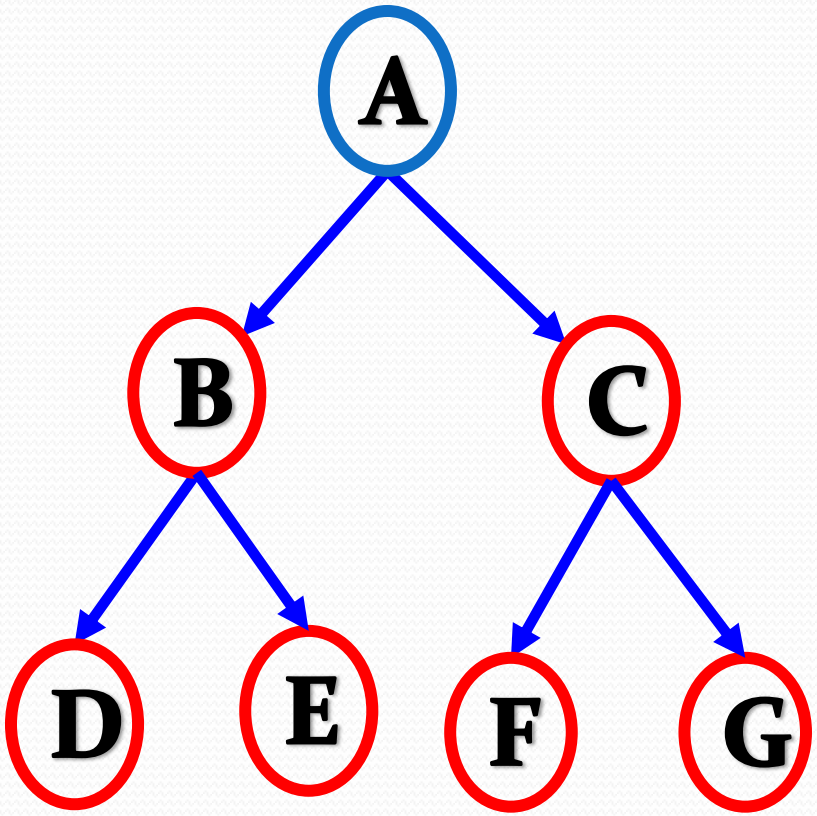
# Tree Traversal : Post order




- Post order (L R N)

  D, E, B, F, G

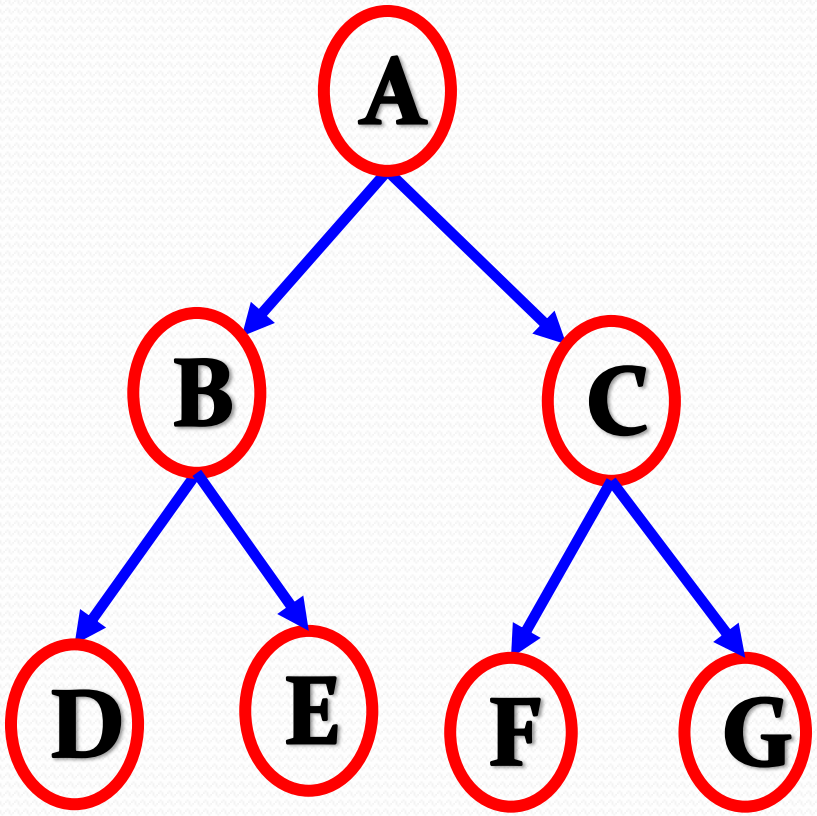
# Tree Traversal : Post order




- Post order (L R N)

  D, E, B, F, G, C

# Tree Traversal : Post order




- Post order (L R N)

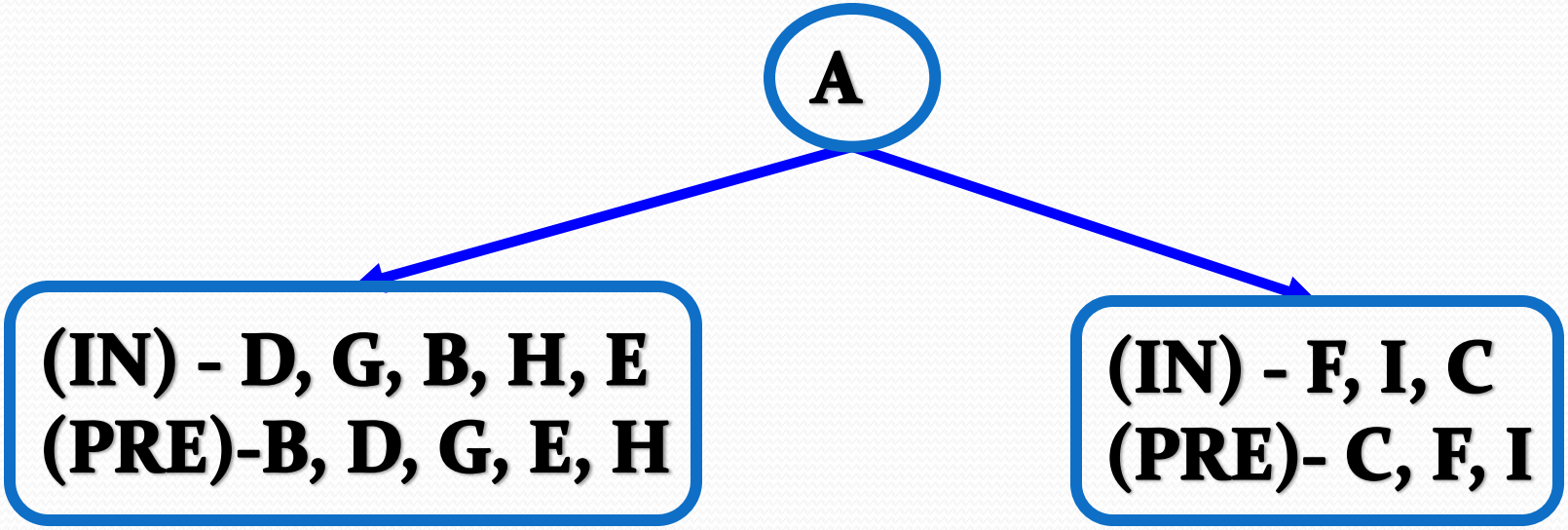
  D, E, B, F, G, C, A

# Constructing Binary Tree

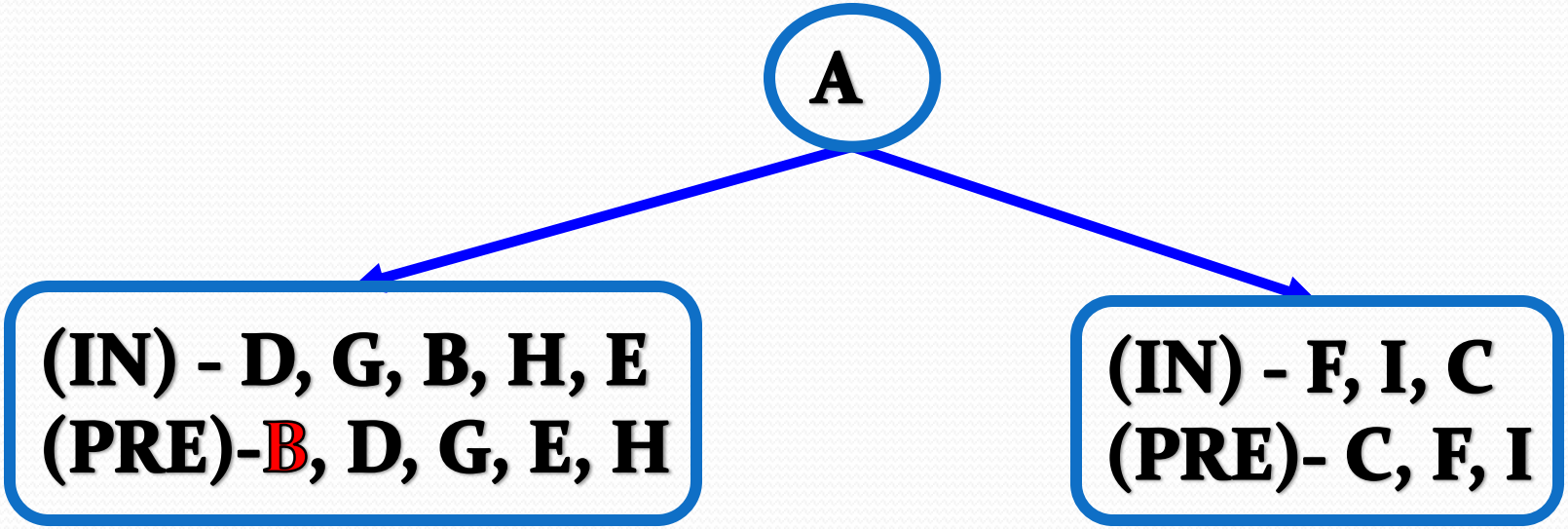
- In order – D, G, B, H, E, A, F, I, C
- Pre order – A, B, D, G, E, H, C, F, I

- Step 1 : finding the root
  - Root Node – A (from pre order)
- Step 2 : Find left and right part of the root
  - Left part - **(IN) - D, G, B, H, E**
    **(PRE)-B, D, G, E, H**
  - Right part - **(IN) - F, I, C**
    **(PRE)- C, F, I**

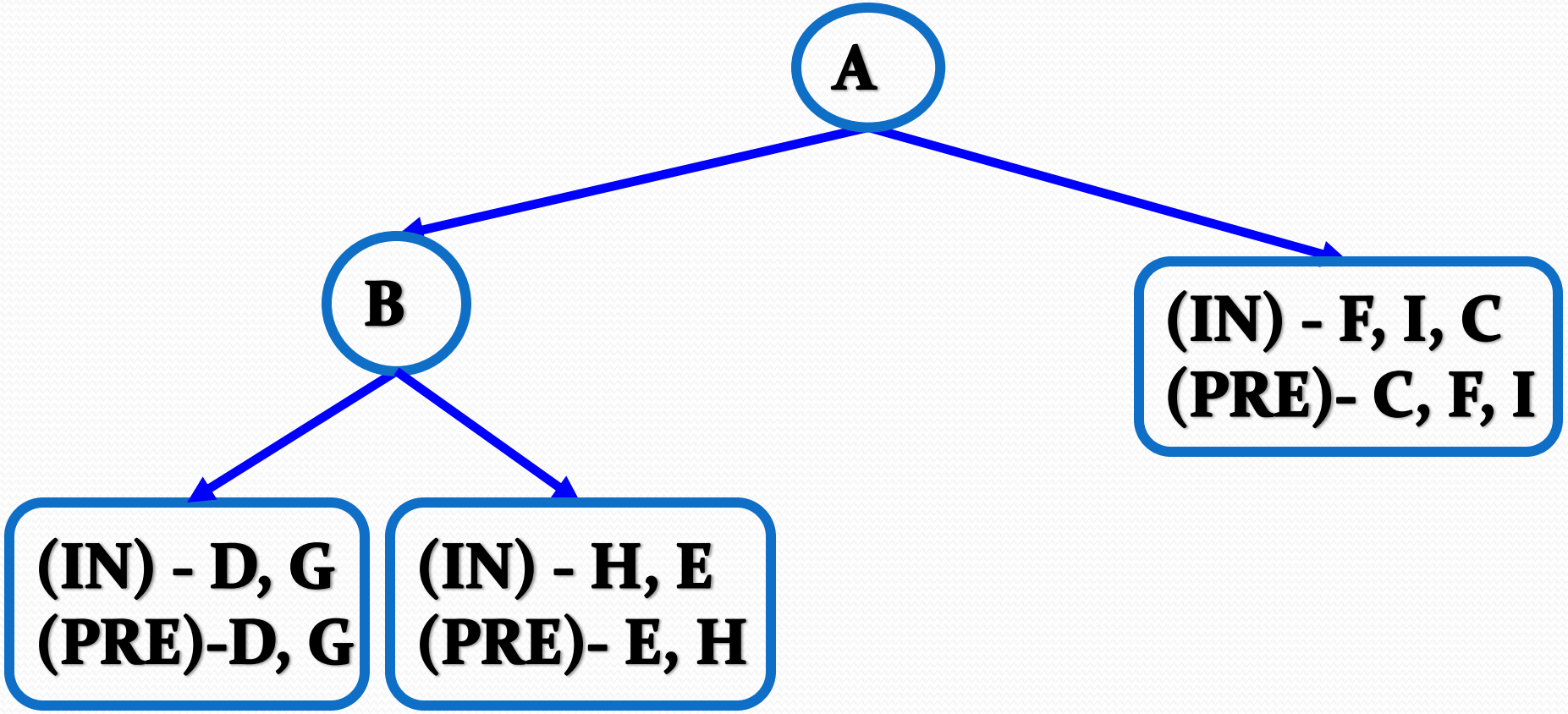
# Constructing Binary Tree



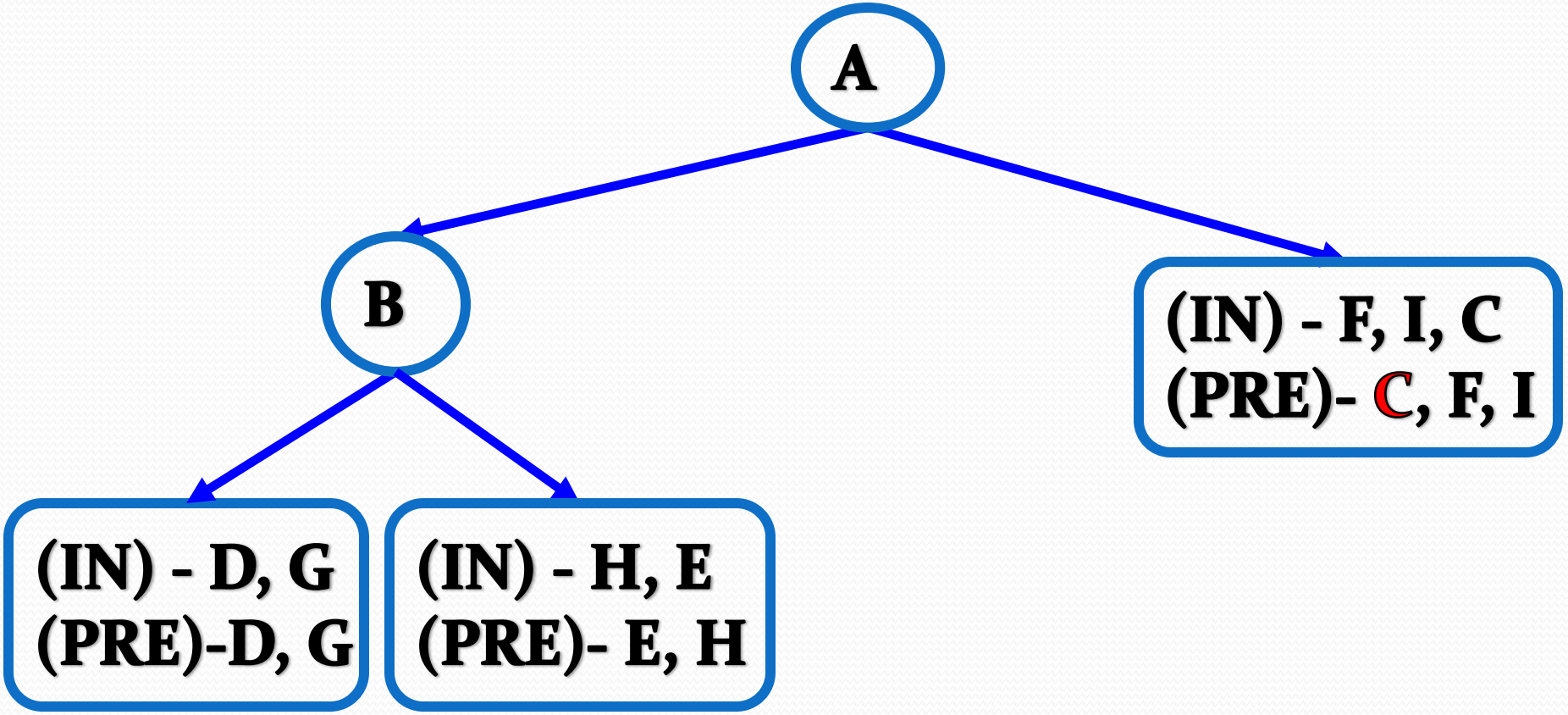
# Constructing Binary Tree



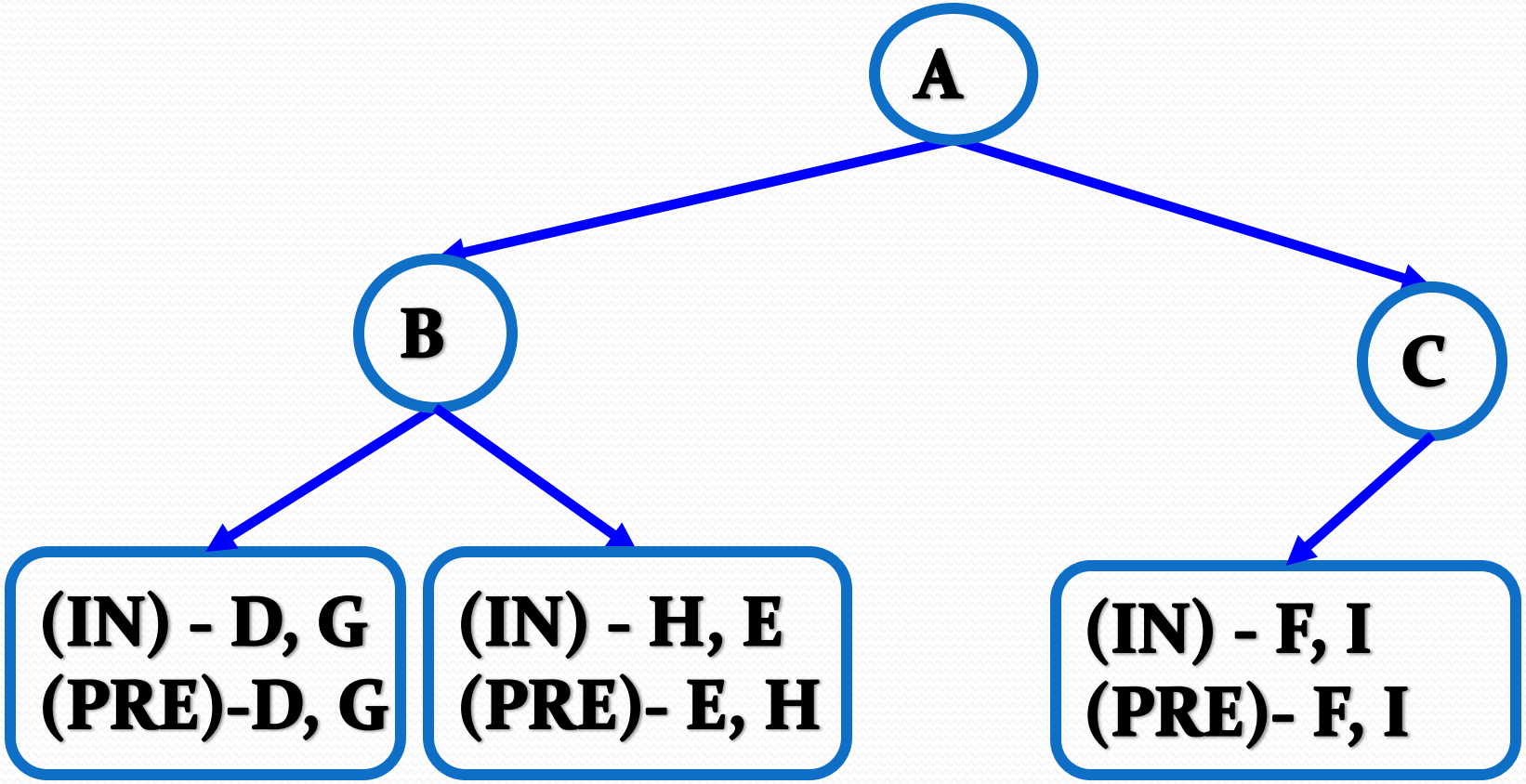
# Constructing Binary Tree



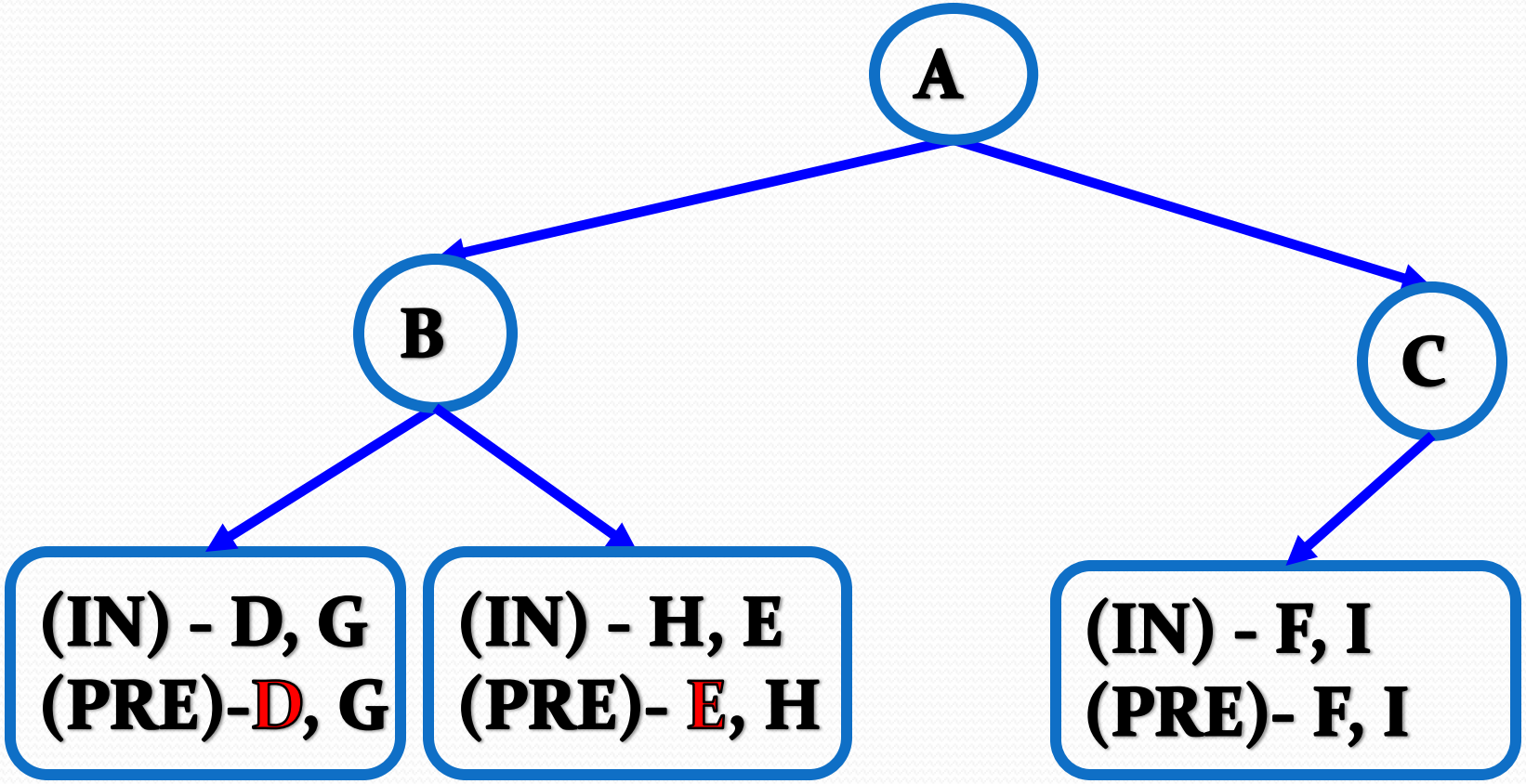
# Constructing Binary Tree



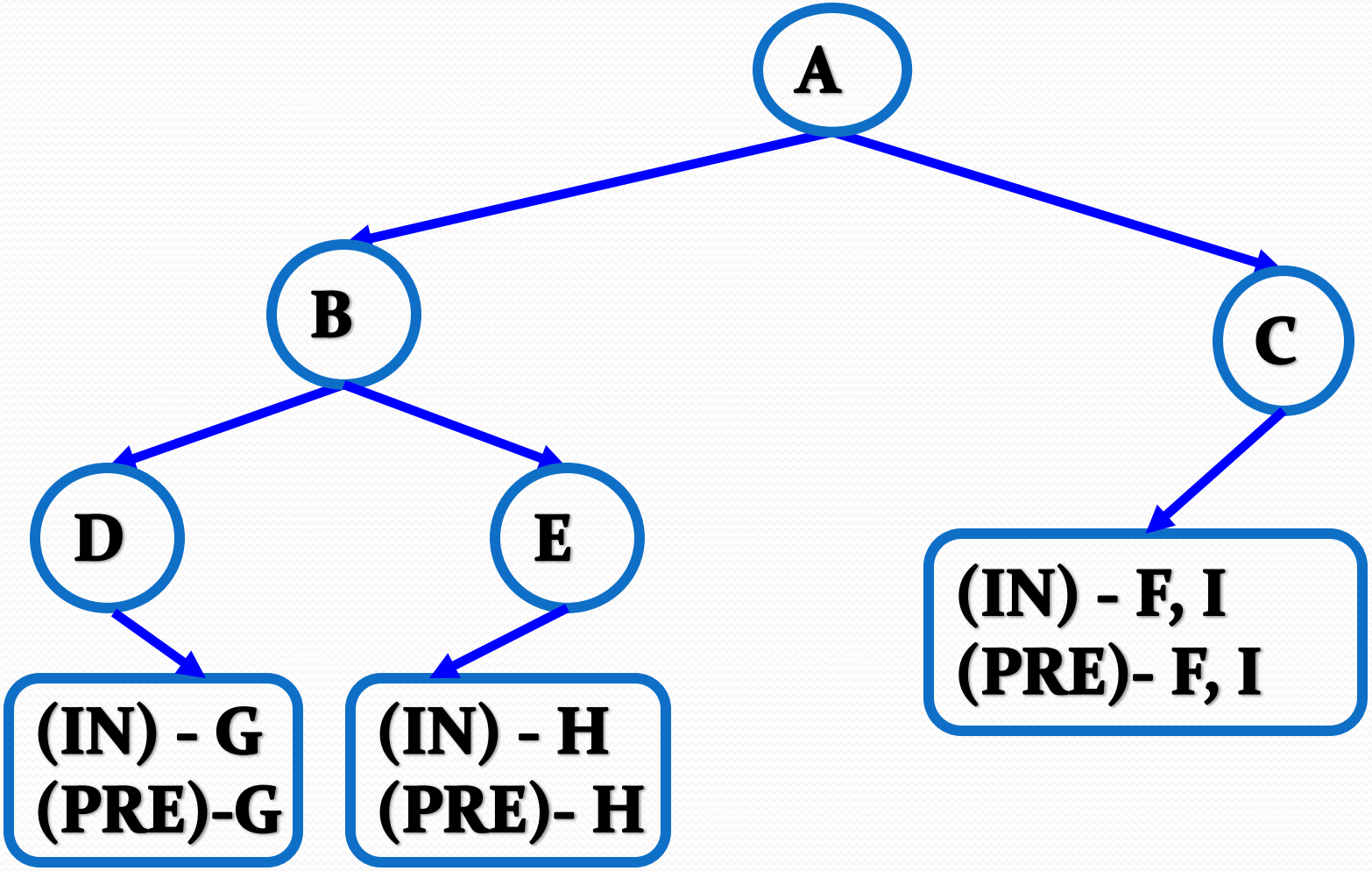
# Constructing Binary Tree



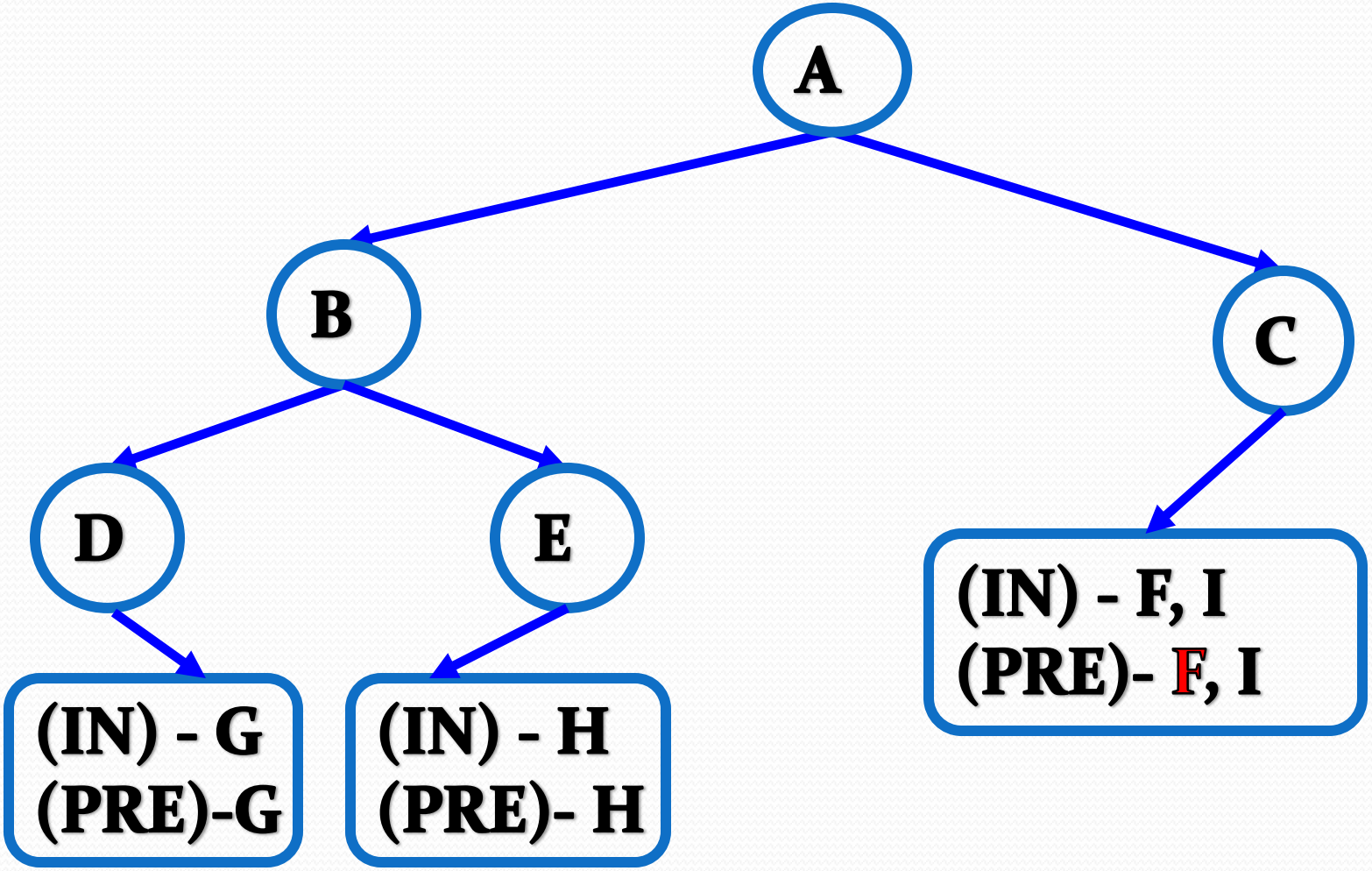
# Constructing Binary Tree



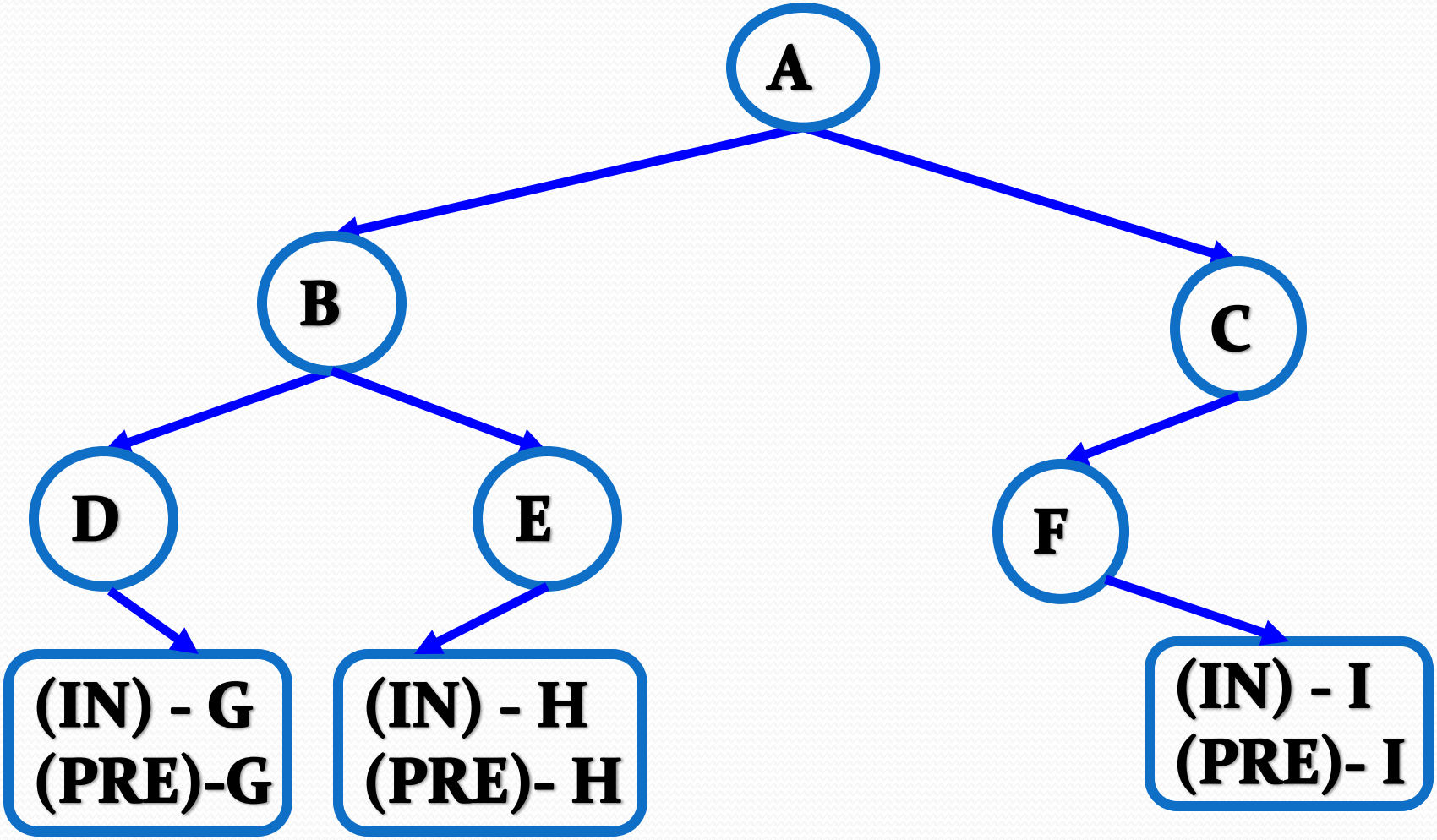
# Constructing Binary Tree



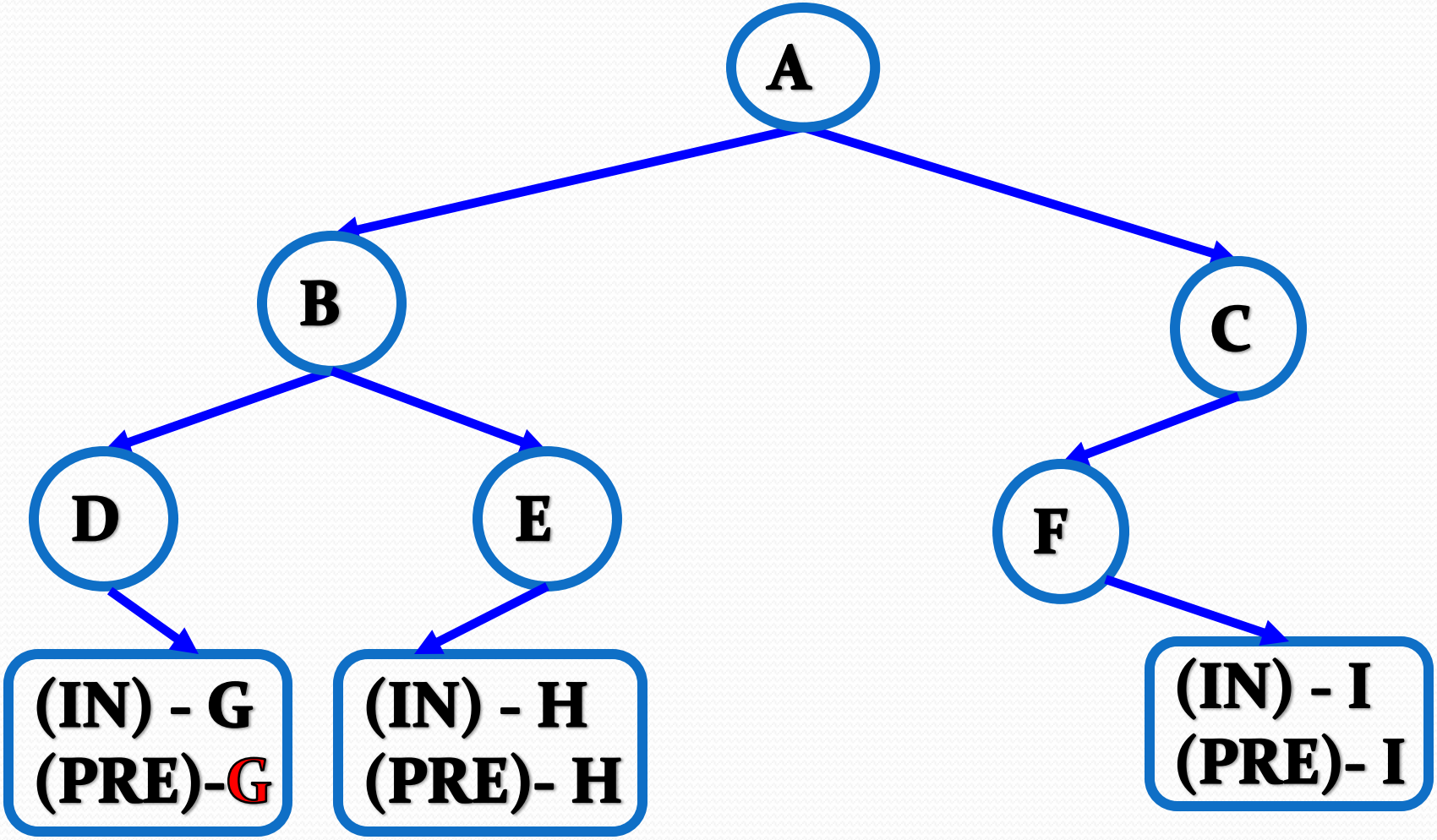
# Constructing Binary Tree



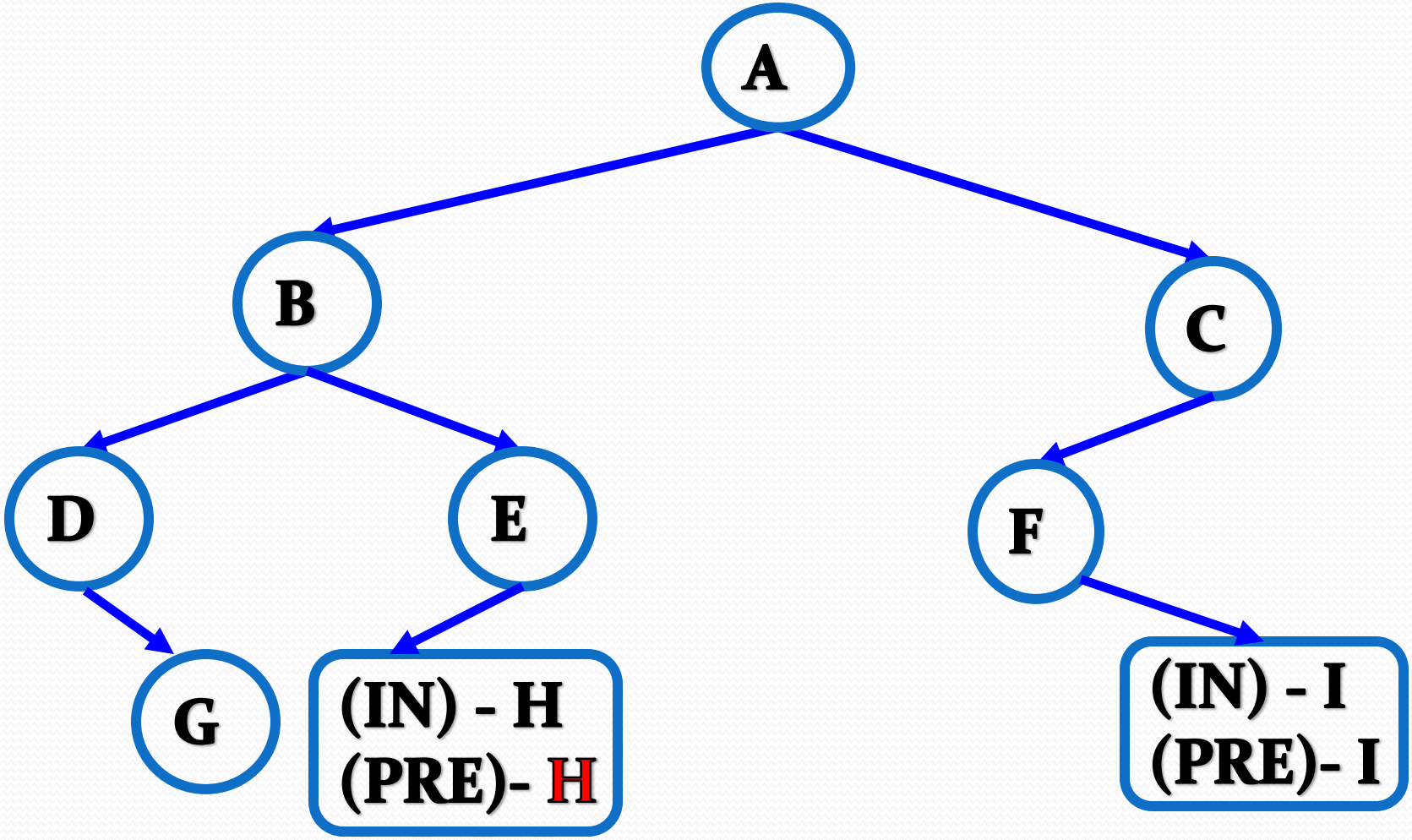
# Constructing Binary Tree



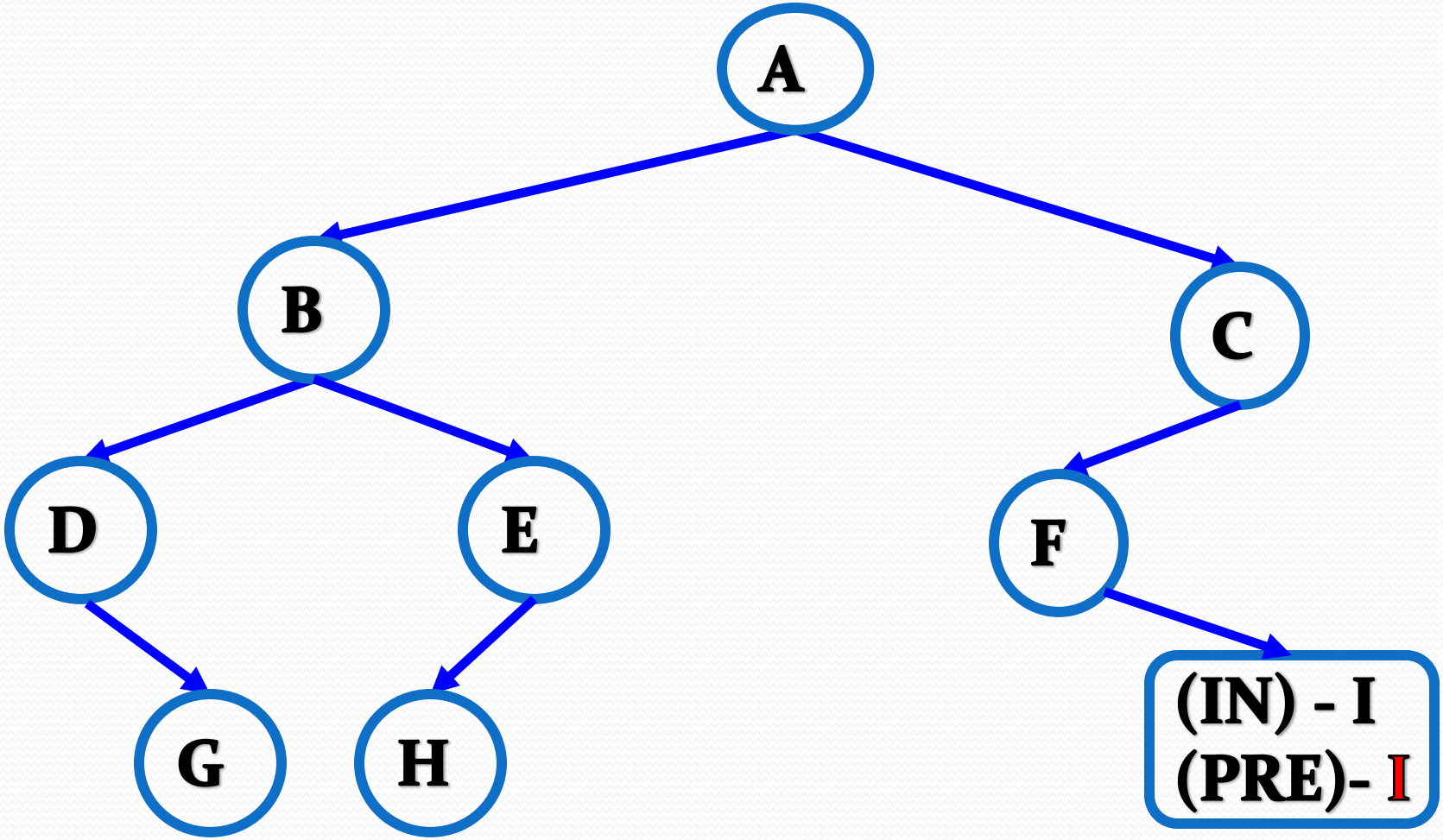
# Constructing Binary Tree



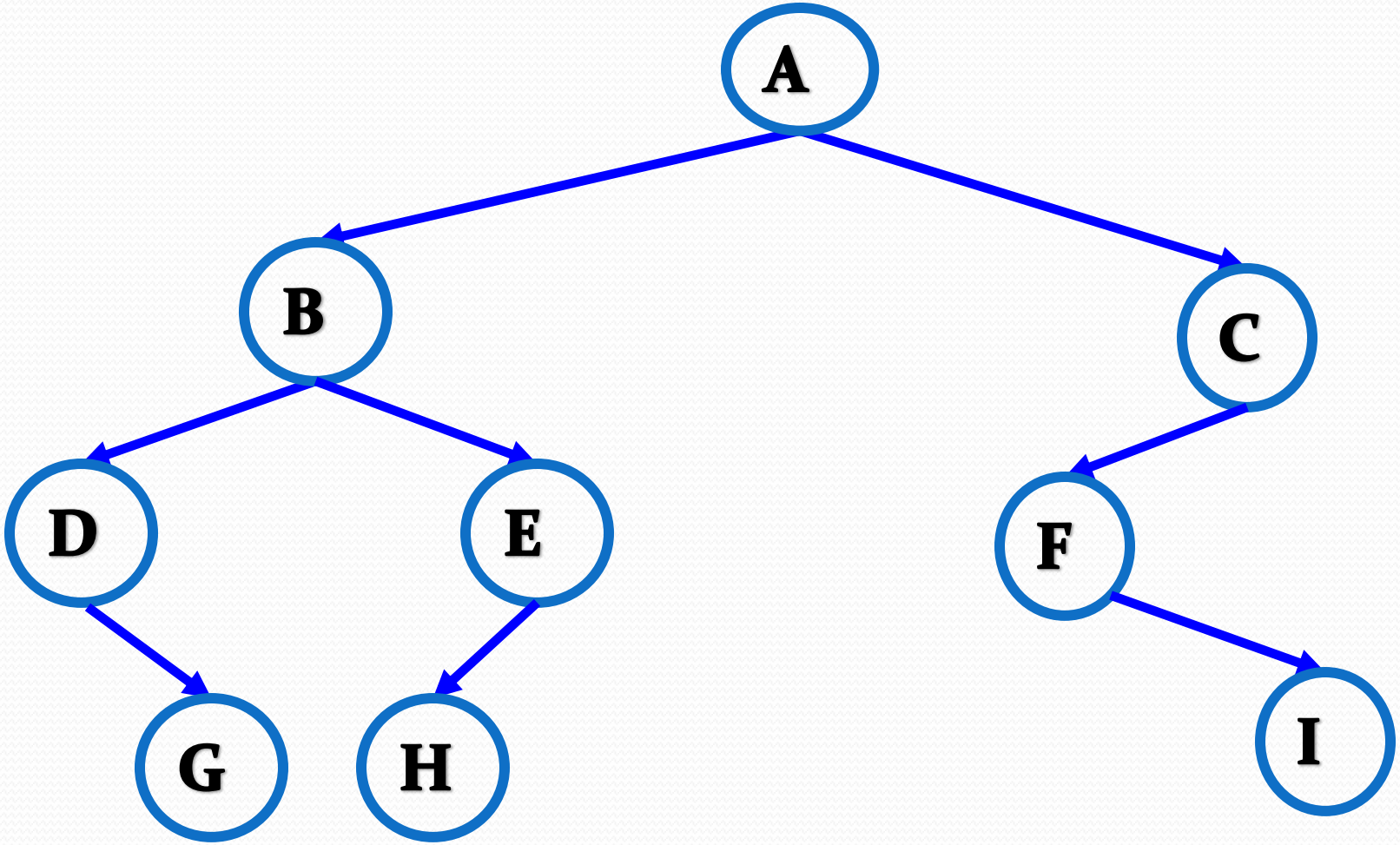
# Constructing Binary Tree



# Constructing Binary Tree



# Constructing Binary Tree



# Binary Search Tree

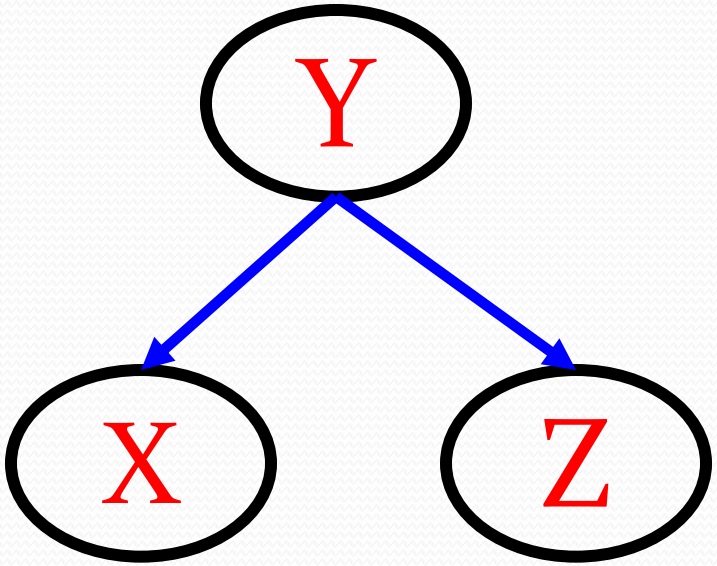
# Binary Search Tree

The value at any node,

- Greater  than every value in left subtree
- Smaller than every value in right subtree

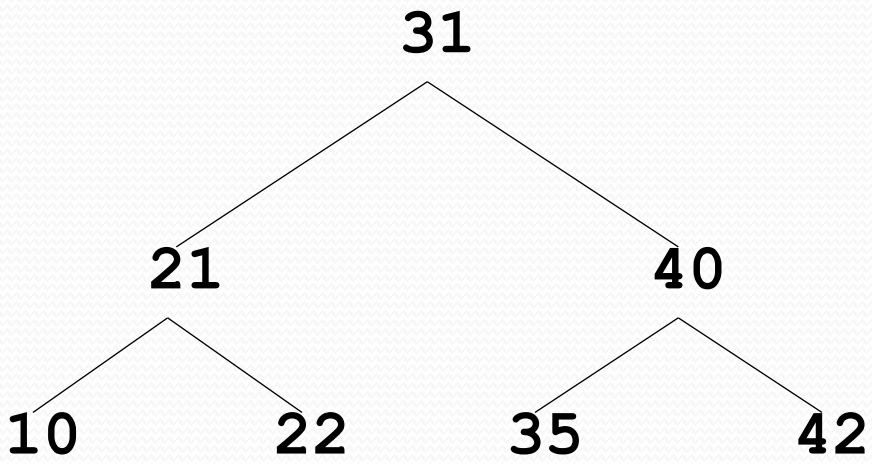
– Example

- Y > X
- Y < Z



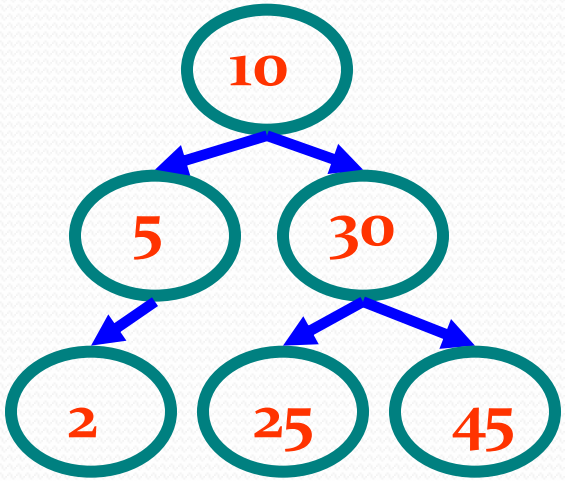
# Binary Search Tree

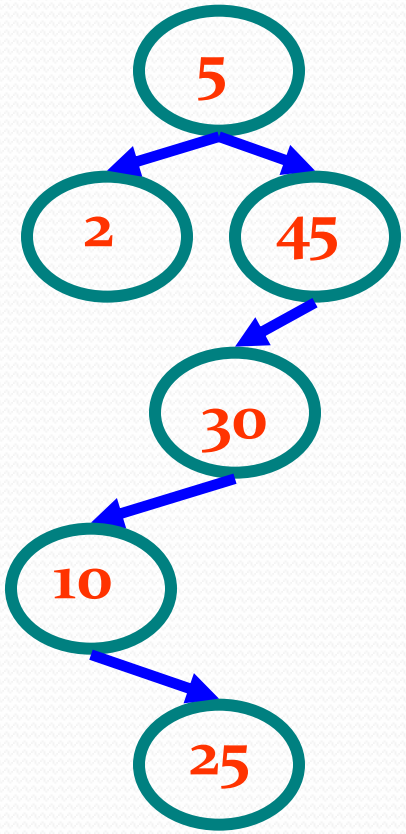
- Values in left sub tree less than parent
- Values in right sub tree greater than parent
- Fast searches in a Binary Search tree, maximum of log n comparisons



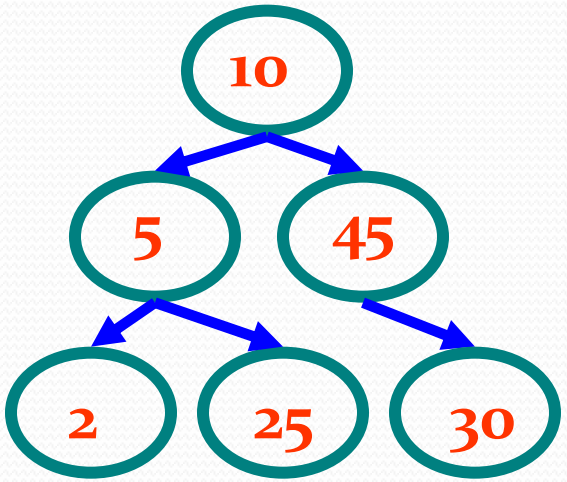
# Binary Search Trees

- Examples



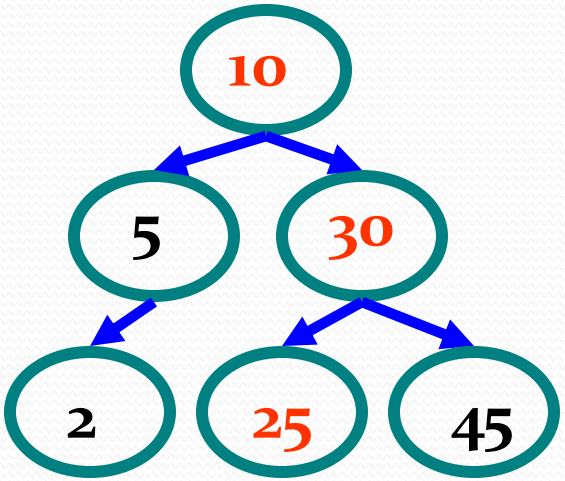

Binary search trees




Not a binary search tree

# Example Binary Searches

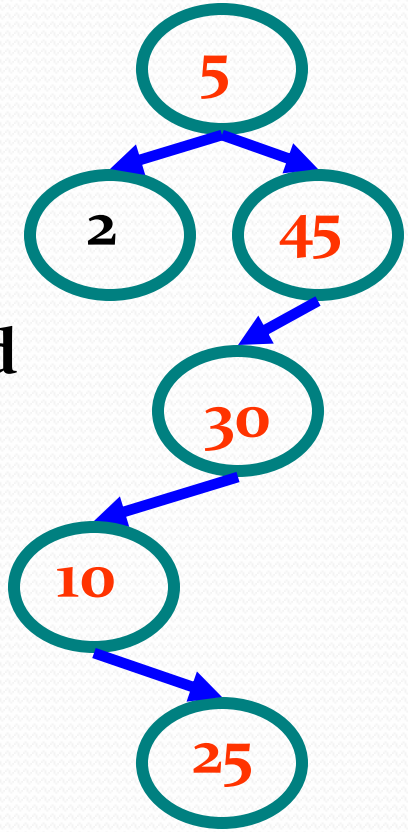
- search (root, 25 )




10 < 25, right

30 > 25, left

25 = 25, found




5 < 25, right

45 > 25, left

30 > 25, left

10 < 25, right

25 = 25, found

# Algorithm for Binary Search Tree

- A) compare ITEM with the root node N of the tree
  - i) if ITEM<N, proceed to the left child of N.
  - ii) if ITEM>N, proceed to the right child of N.
- B) repeat step (A) until one of the following occurs
  - i) we meet a node N such that ITEM=N, i.e. search is successful.
  - ii) we meet an empty sub tree, i.e. the search is unsuccessful.

# Binary Tree Implementation

```
  typedef struct node
{
    int data;
    struct node *lc,*rc;
};
```

# Iterative Search of Binary Search Tree

```
search()
{
      while (n != NULL)
      {
      if (n->data == item)       // Found it
            return n;
          if (n->data > item)              // In left subtree
            n = n->lc;
          else                        // In right subtree
            n = n->rc;
      }
       return null;
}
```

# Recursive Search of Binary Search Tree

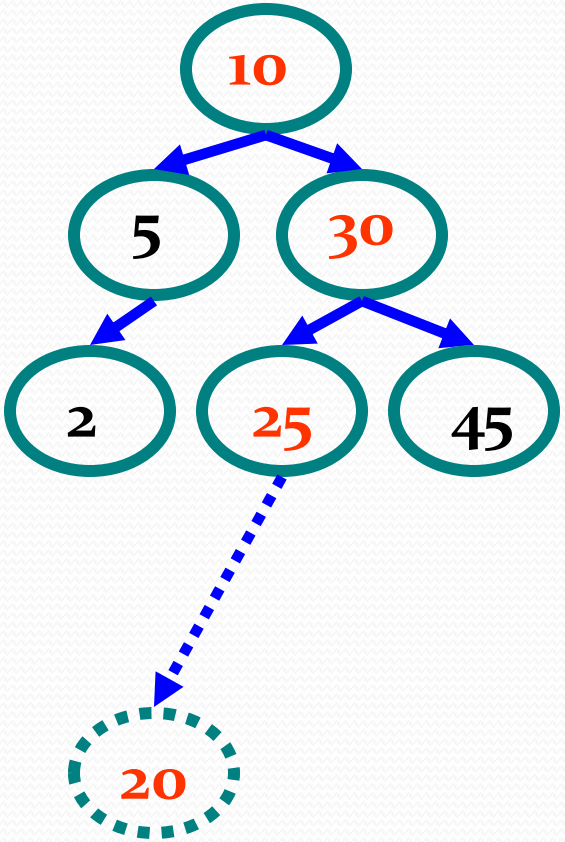
```
Search(node *n, info)
{
    if (n == NULL)                  // Not found
        return( n );
    else if (n->data == item)       // Found it
        return( n );
    else if (n->data > item)        // In left subtree
        return search( n->left, item );
    else                            // In right subtree
        return search( n->right, item );
}
```

# Insertion in a Binary Search Tree

- Algorithm
  1. Perform search for value X
  2. Search will end at node Y (if X not in tree)
  3. If X < Y, insert new leaf X as new left subtree for Y
  4. If X > Y, insert new leaf X as new right subtree for Y

# Insertion in a Binary Search Tree

- Insert ( 20 )




**10 < 20, right**

**30 > 20, left**

**25 > 20, left**

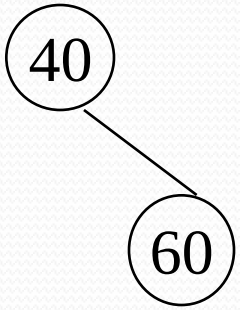
**Insert 20 on left**

# Insertion in a Binary Tree

**40, 60, 50, 33, 55, 11**

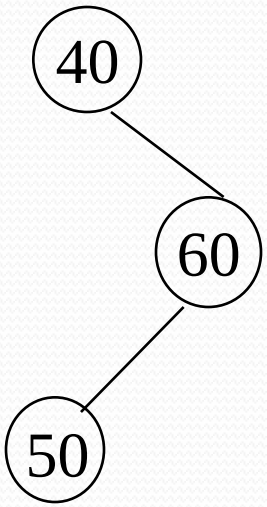
# Insertion in a Binary Tree

**40, 60, 50, 33, 55, 11**



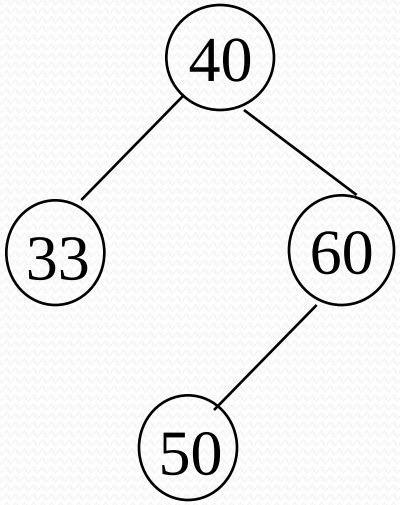
# Insertion in a Binary Tree

**40, 60, 50, 33, 55, 11**



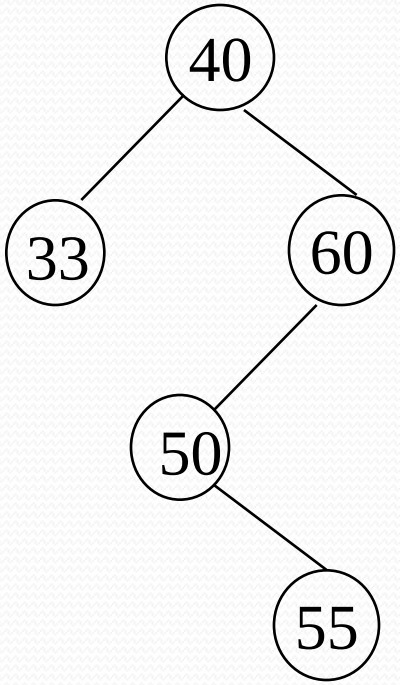
# Insertion in a Binary Tree

**40, 60, 50, 33, 55, 11**



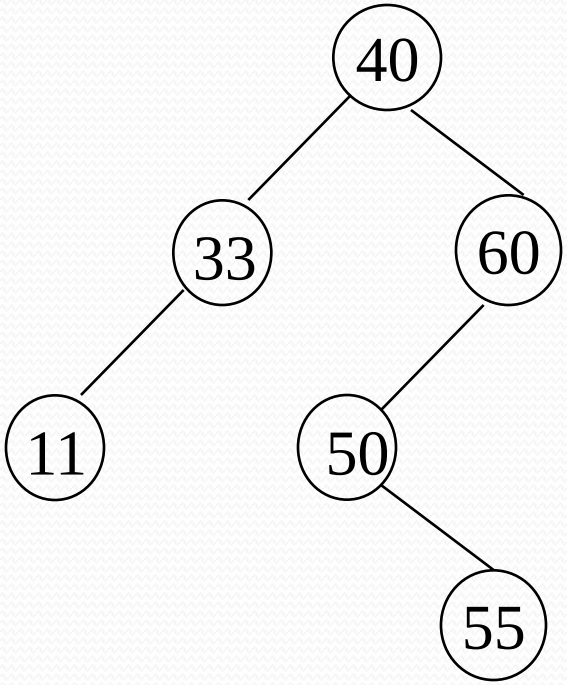
# Insertion in a Binary Tree

**40, 60, 50, 33, 55, 11**



# Insertion in a Binary Tree

**40, 60, 50, 33, 55, 11**



# Deletion in Binary Tree

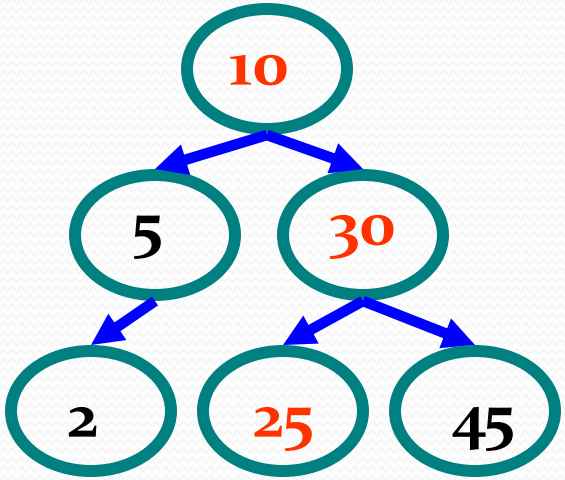
- **Algorithm**
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else //we must delete internal node
     a) Replace with largest value Y on left subtree
        OR smallest value Z on right subtree
     b) Delete replacement value (Y or Z) from subtree

Note :-

- Deletions may unbalance tree

# Example Deletion (Leaf)

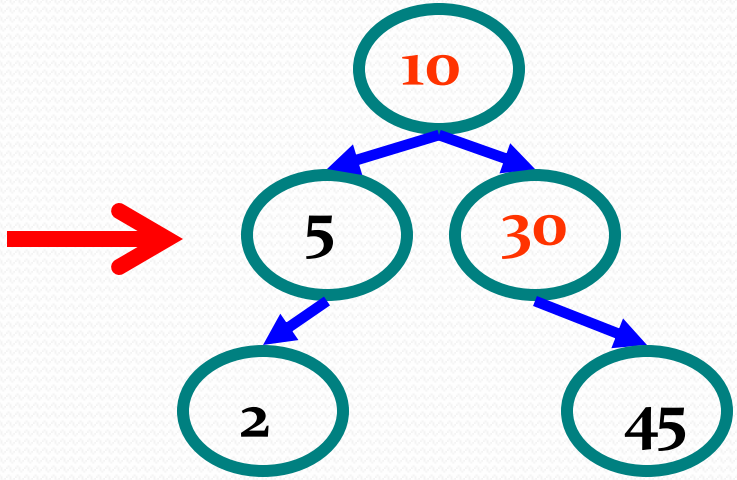
- Delete ( 25 )




10 < 25, right

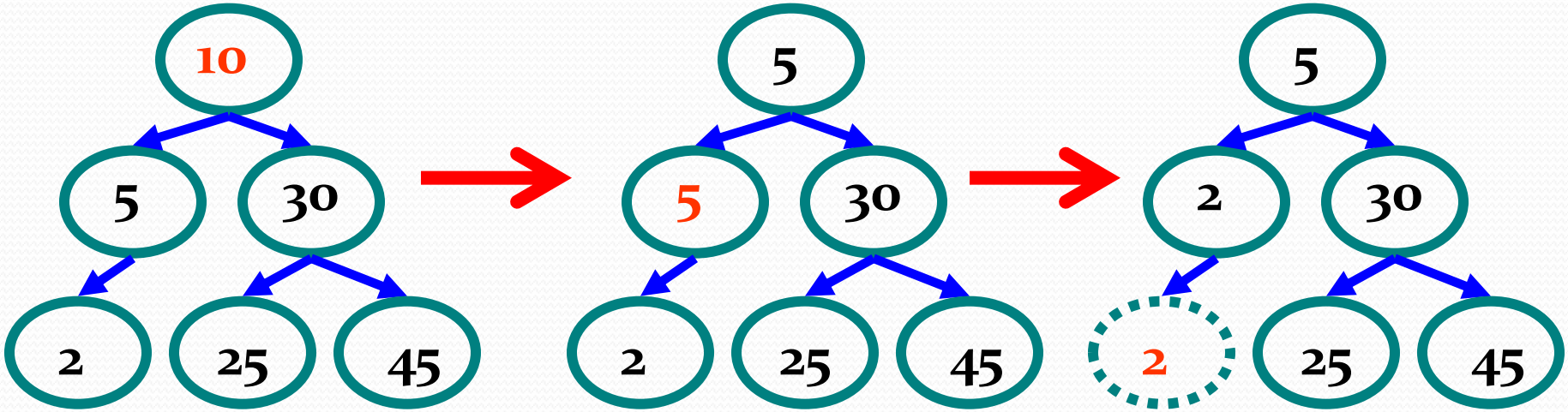
30 > 25, left

25 = 25, delete



# Example Deletion (Internal Node)

- Delete ( 10 )



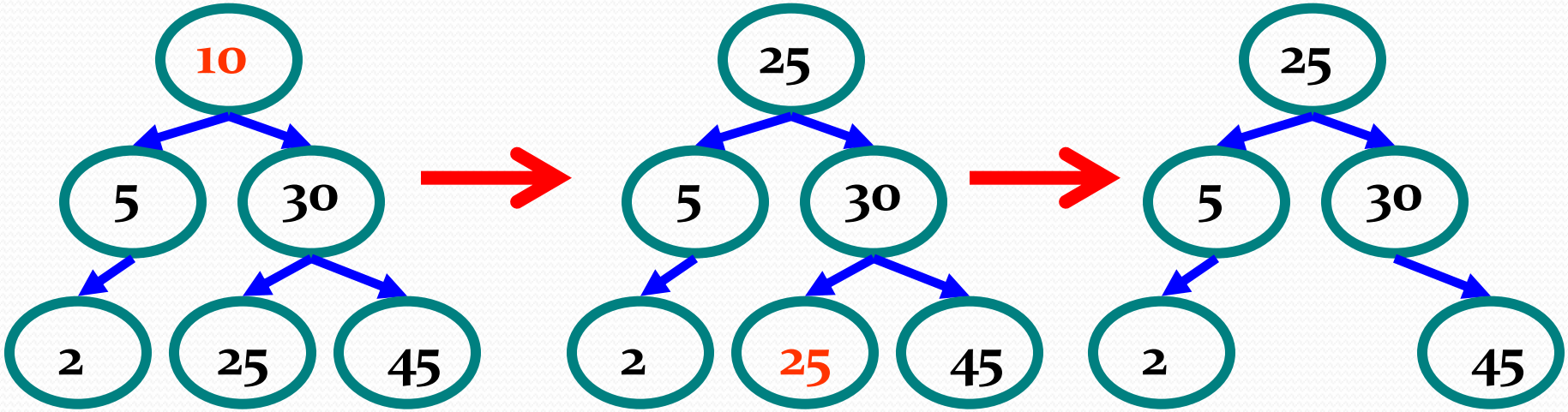

**Replacing 10 with largest value in left subtree**

**Replacing 5 with largest value in left subtree**

**Deleting leaf**

# Example Deletion (Internal Node)

- Delete ( 10 )




**Replacing 10 with smallest value in right subtree**

**Deleting leaf**

**Resulting tree**

# Binary Search Properties

- Time of search
  - Proportional to height of tree
  - Balanced binary tree
    - O( log(n) ) time
  - Degenerate tree
    - O( n ) time
    - Like searching linked list / unsorted array

# AVL Tree

- AVL trees are height-balanced binary search trees
- Balance factor of a node=

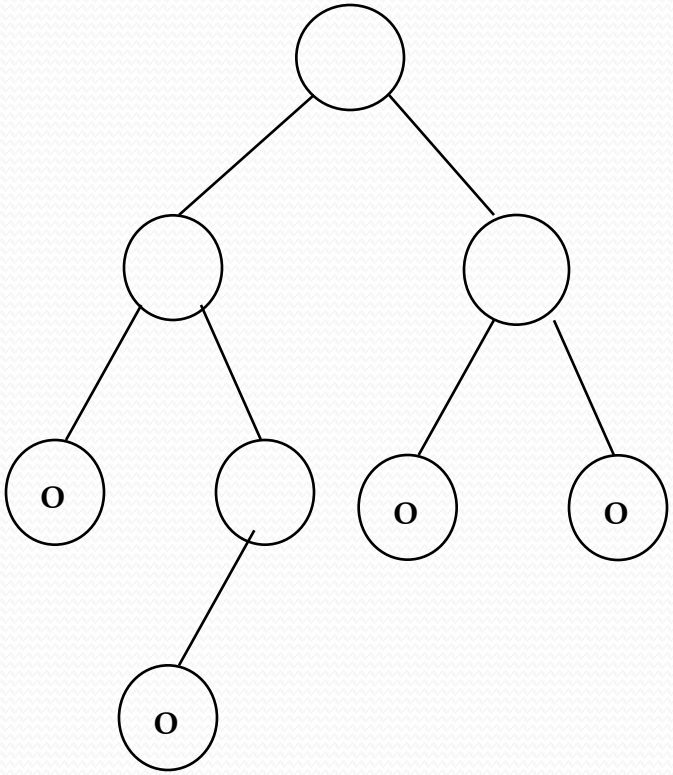
  height(left sub tree) - height(right sub tree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right sub tree can differ by no more than 1
  - Store current heights in each node

# AVL Tree

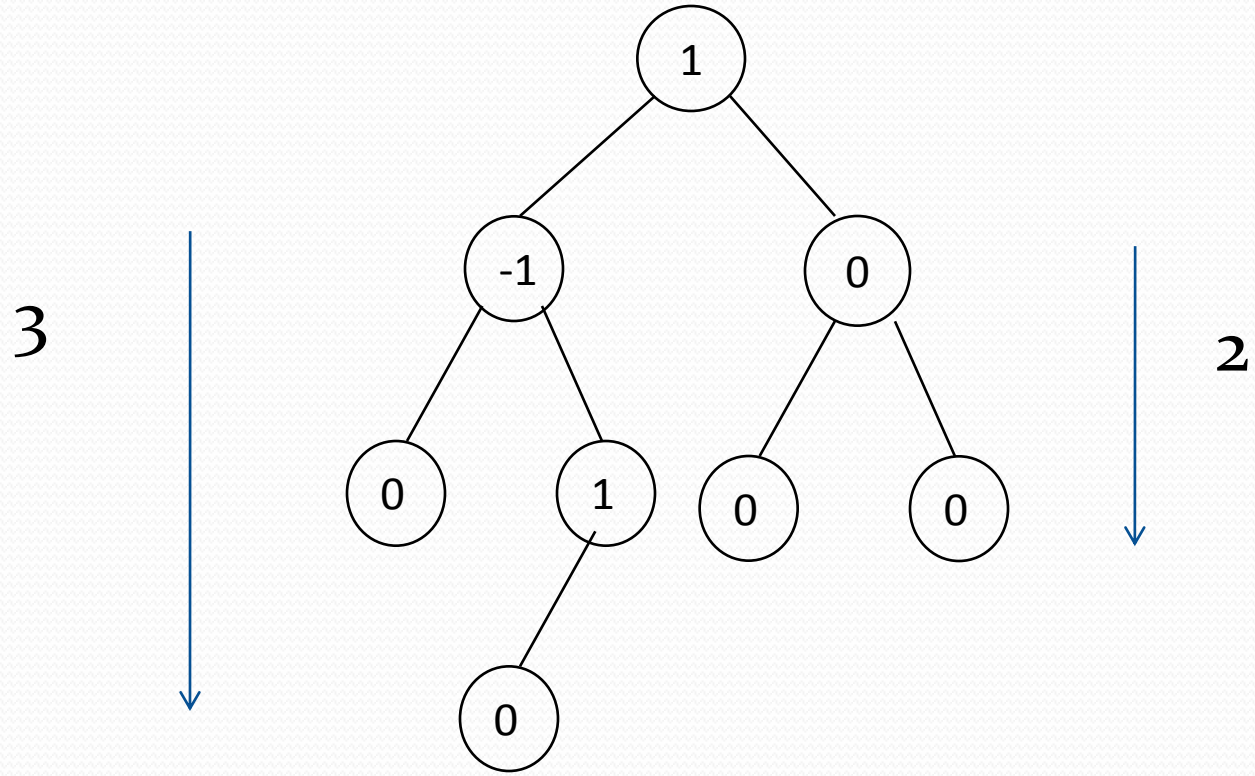
- A binary tree in which the difference of height of the right and left subtree of any node is less than or equal to 1 is known as AVL Tree.
- Height of left subtree – height of right subtree

  can be either -1,0,1

# AVL Tree

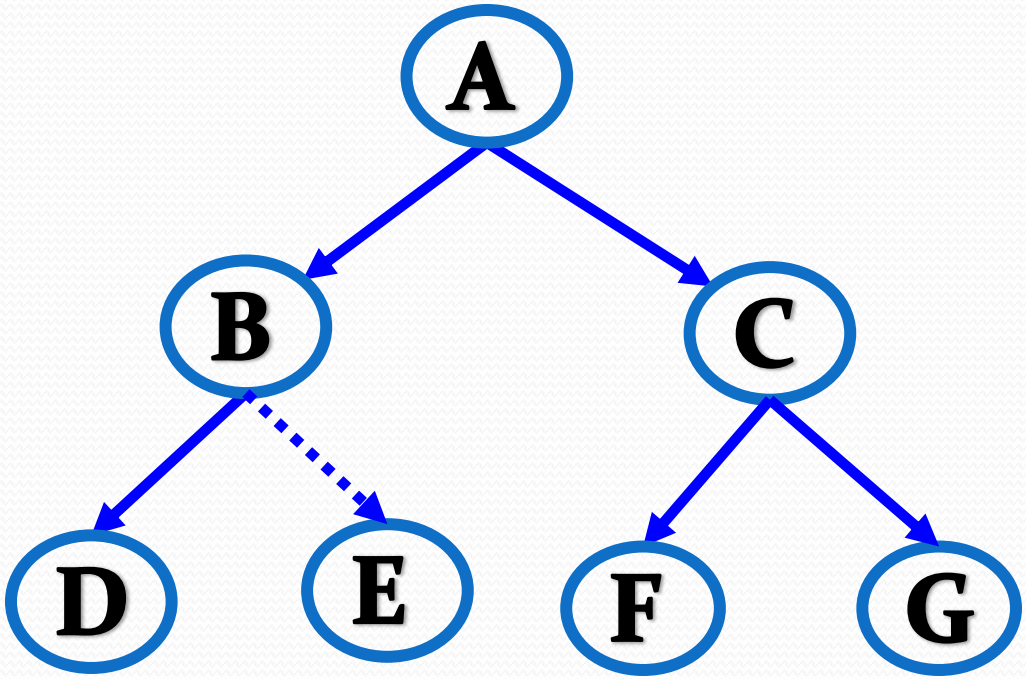


# AVL Tree



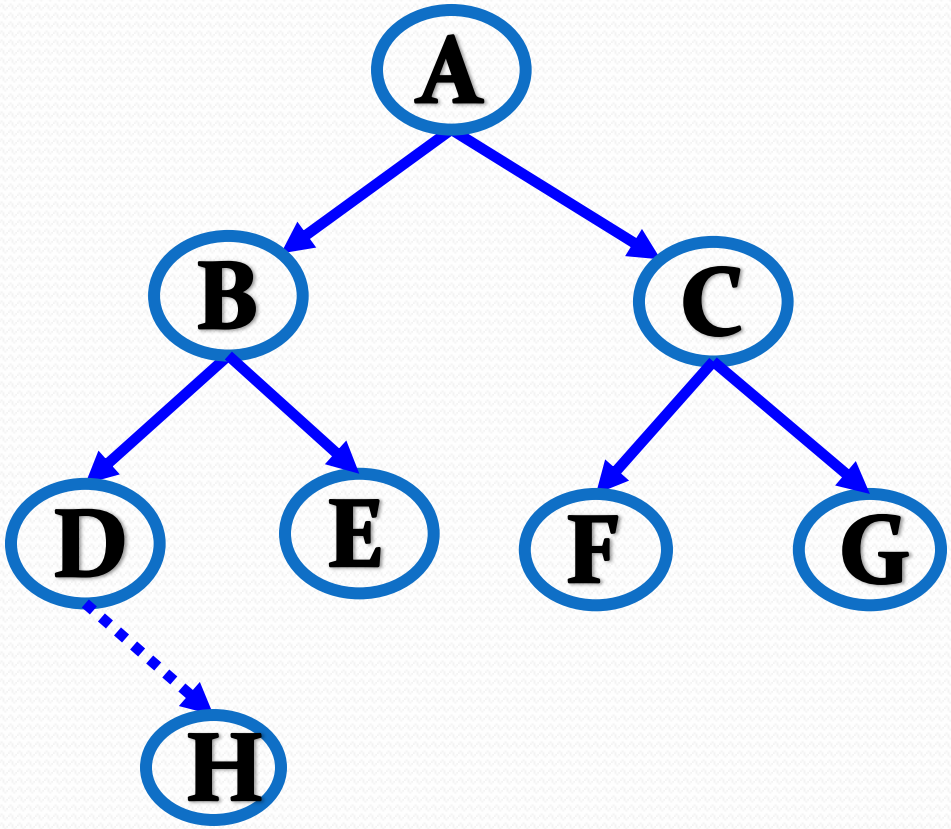

Balanced as LST-RST=1

# Insertion in AVL Tree



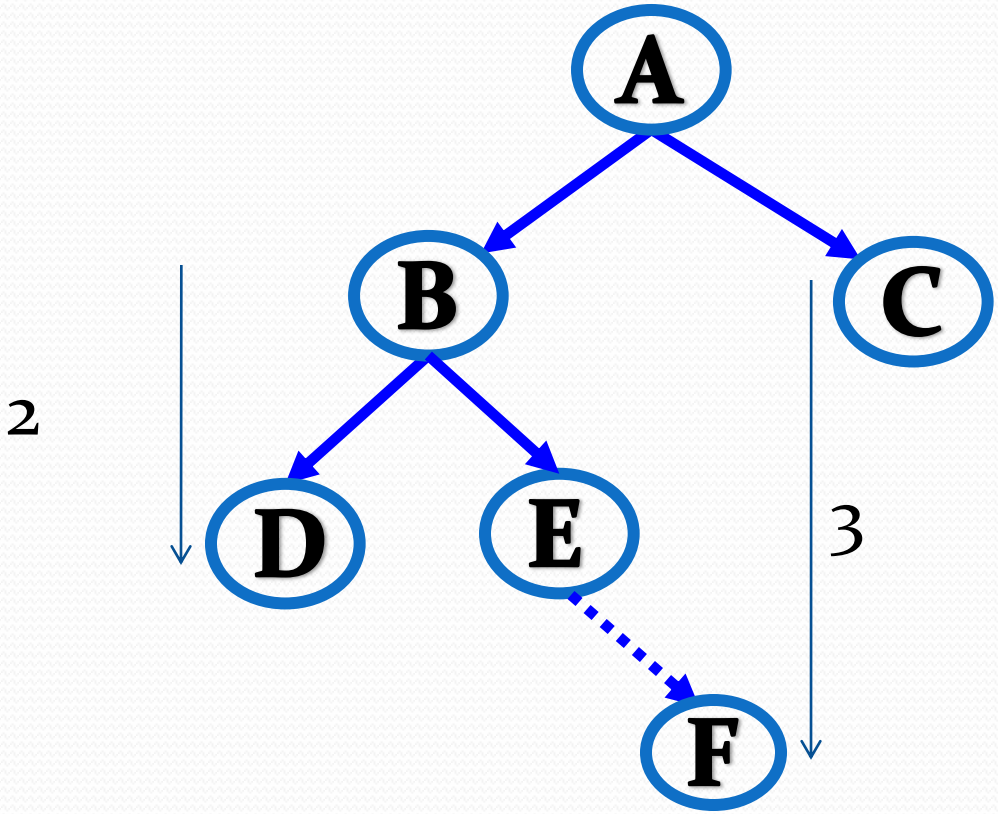

**Case 1:** the node was either left heavy or right heavy and has become balanced

# Insertion in AVL Tree




**Case 2:** the node was balanced and has now become left or right heavy

# Insertion in AVL Tree




**Case 3:** the node was heavy and the new node has been inserted in the heavy sub tree thus creating an unbalanced sub tree

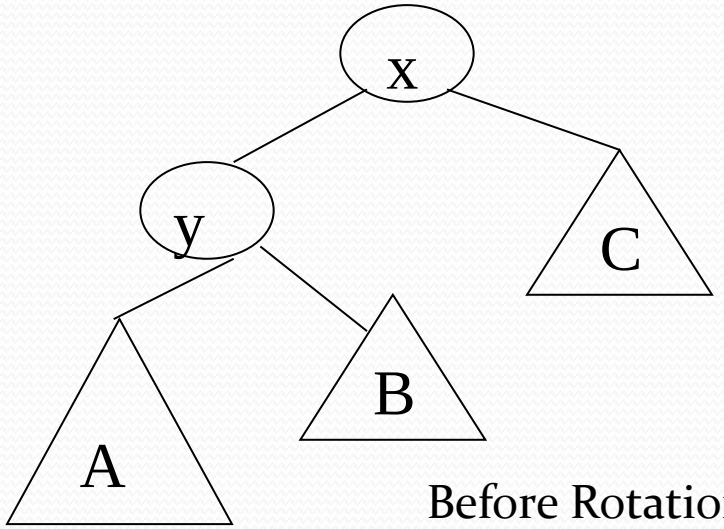
# Rebalancing

- When the tree structure changes (e.g., insertion or deletion), we need to transform the tree to restore the AVL tree property.
- This is done using single rotations or double rotations.

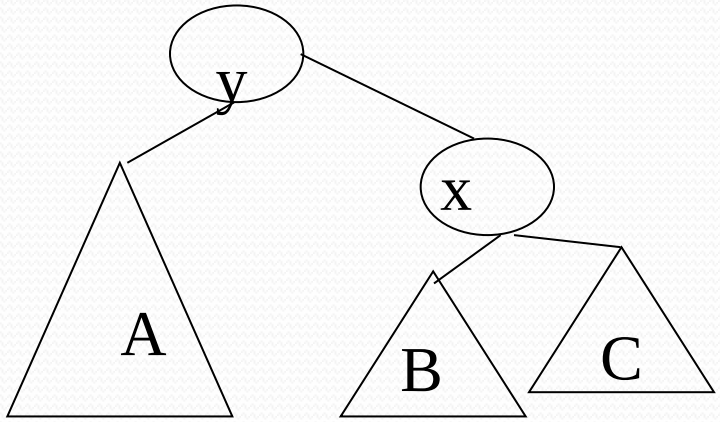
# Rotations

- single rotations

e.g. Single Rotation




Before Rotation



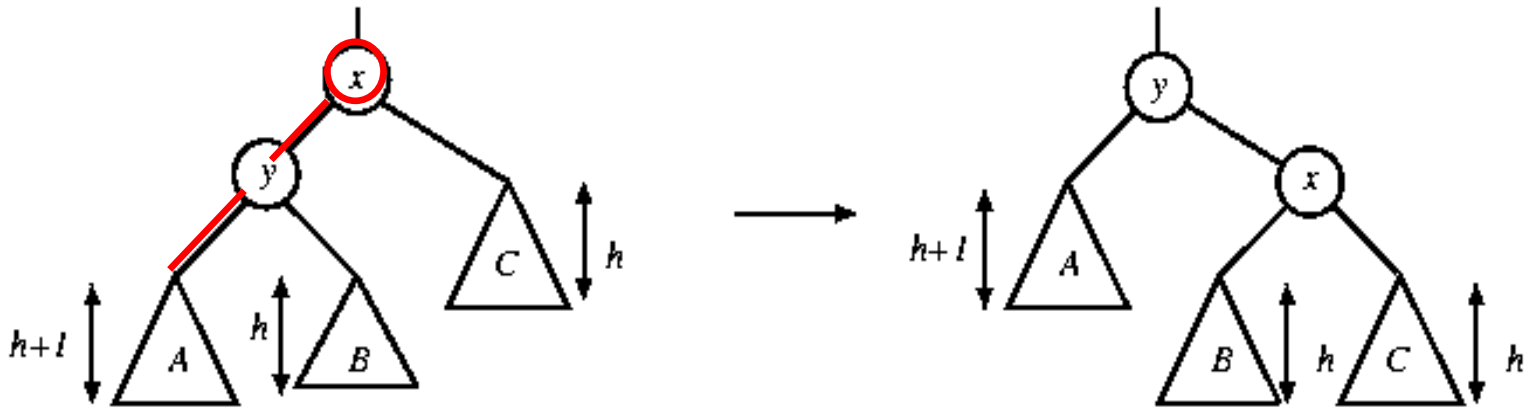

After Rotation

# Rotations

- Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of some subtree by 1
- Thus, if the AVL tree property is violated at a node x, it means that the heights of left(x) and right(x) differ by exactly 2.
- Rotations will be applied to x to restore the AVL tree property.

# Single Rotation

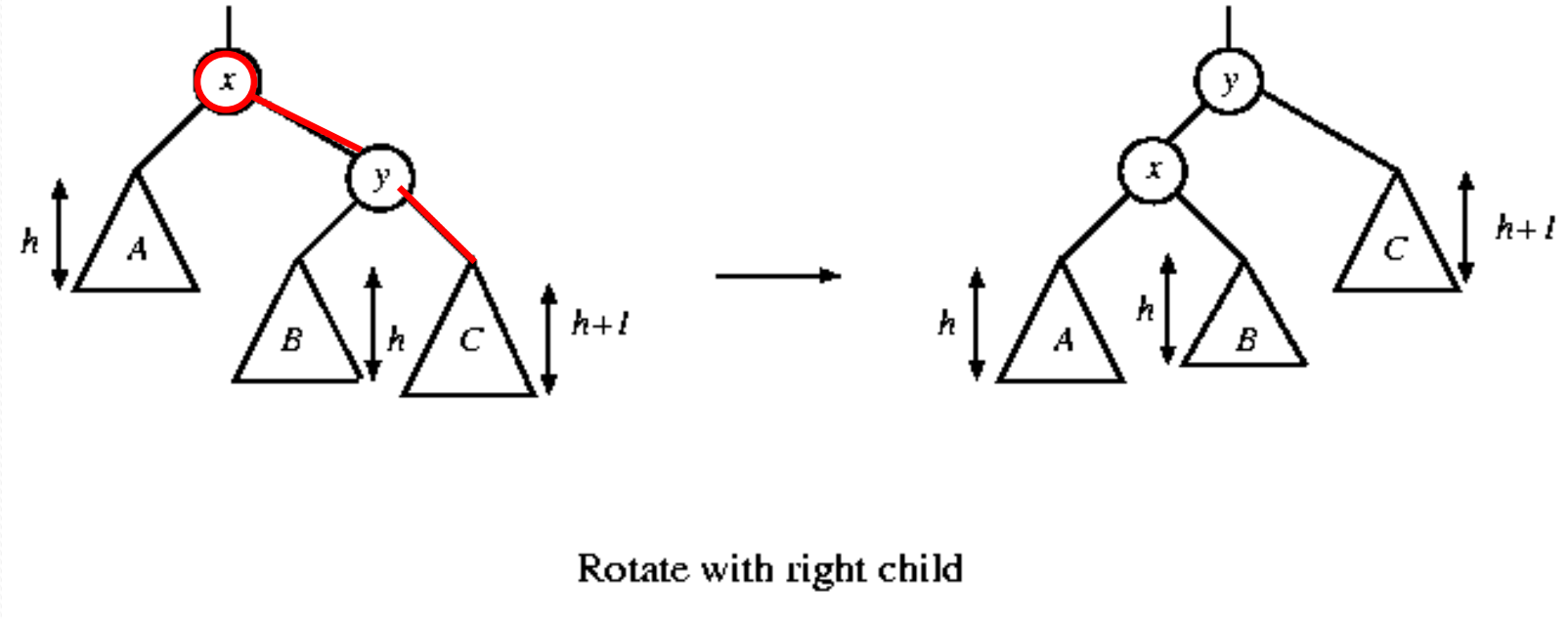
The new item is inserted in the subtree A.
The AVL-property is violated at x
height of left(x) is h+2
height of right(x) is h.




Rotate with left child

# Single Rotation

The new item is inserted in the subtree C.
The AVL-property is violated at x.

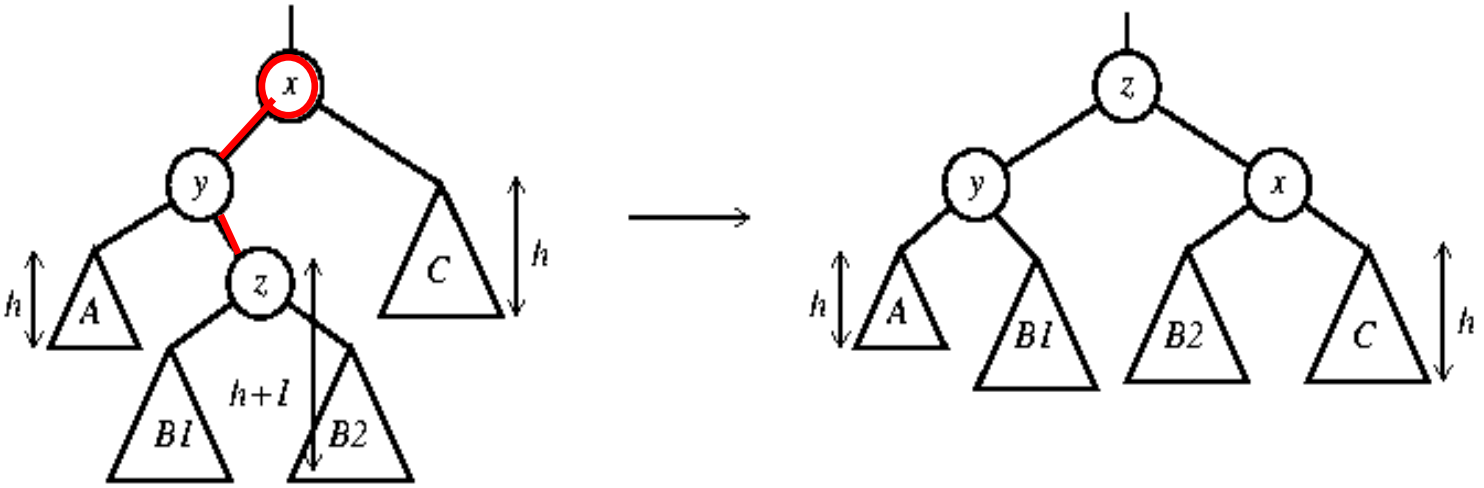



Rotate with right child

Single rotation takes O(1) time.
Insertion takes O(log N) time.

# Double Rotation

The new key is inserted in the subtree B1 or B2.
The AVL-property is violated at x.
x-y-z forms a zig-zag shape

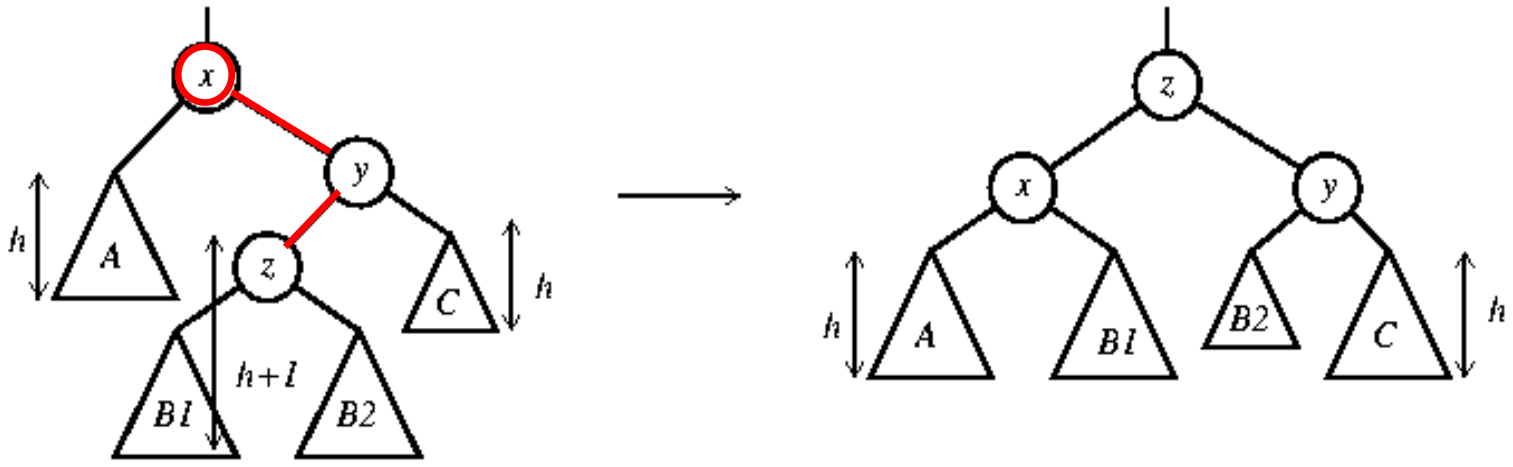



Double rotate with left child

also called left-right rotate

# Double Rotation

The new key is inserted in the subtree B1 or B2.
The AVL-property is violated at x.

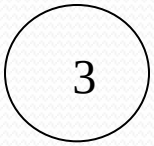



Double rotate with right child

also called right-left rotate

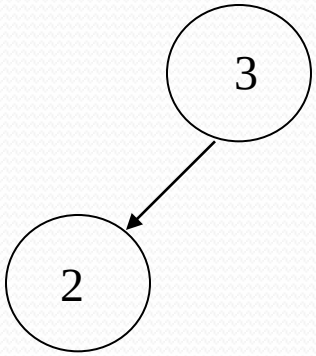
# Example

Insert 3,2,1,4,5,6,7



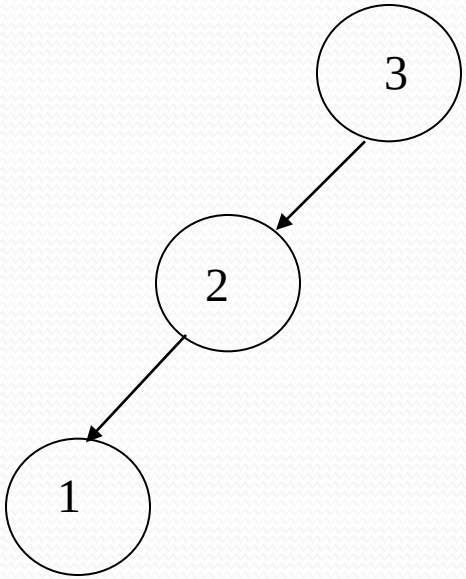
# Example

Insert 3,2,1,4,5,6,7



# Example

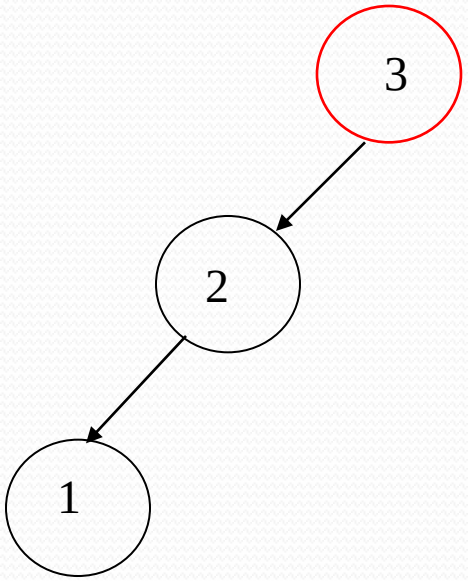
Insert 3,2,1,4,5,6,7



# Example

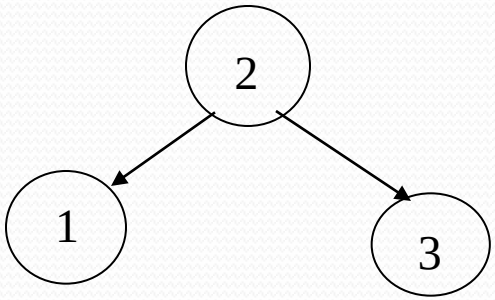
Insert 3,2,1,4,5,6,7

Single rotation



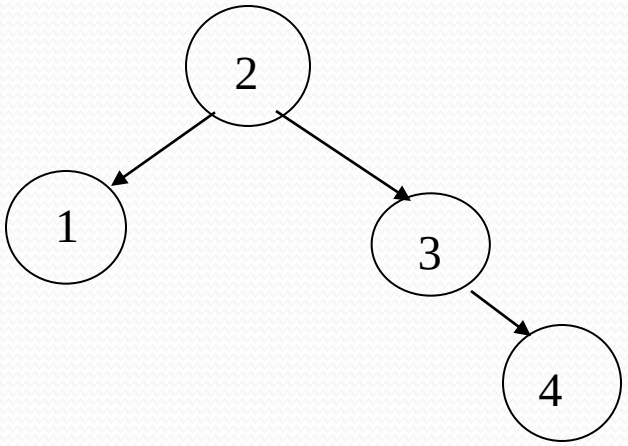
# Example

Insert 3,2,1,4,5,6,7



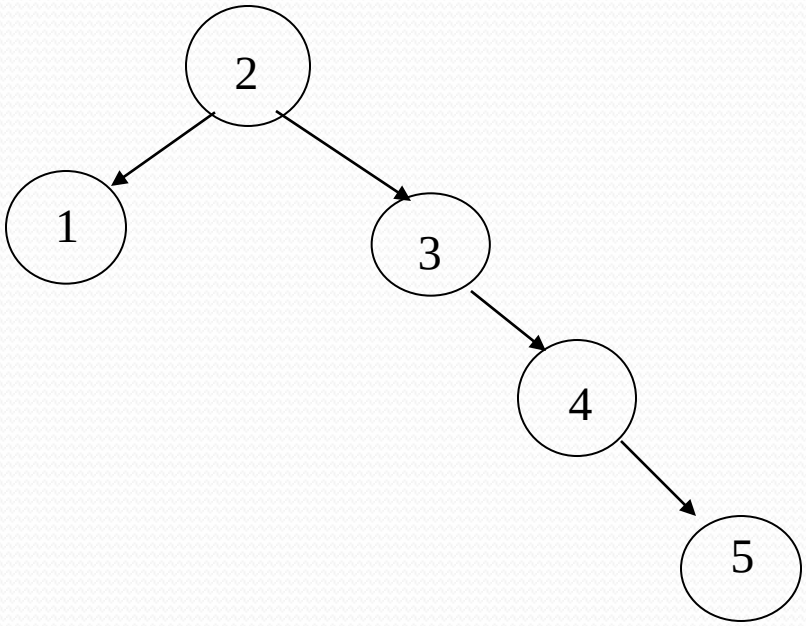
# Example

Insert 3,2,1,4,5,6,7



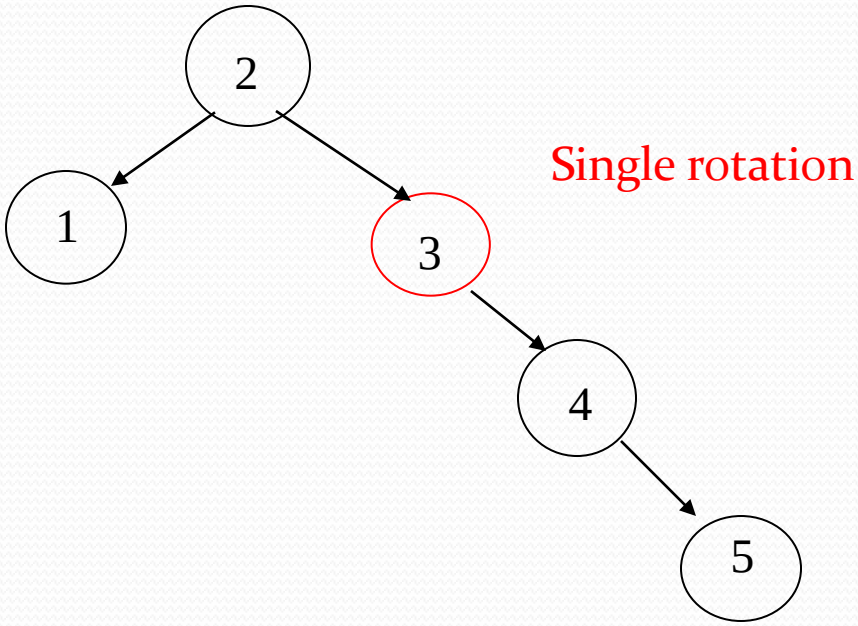
# Example

Insert 3,2,1,4,5,6,7



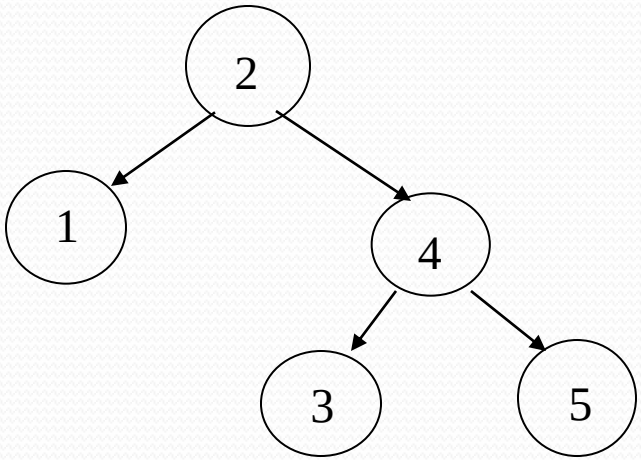
# Example

Insert 3,2,1,4,5,6,7



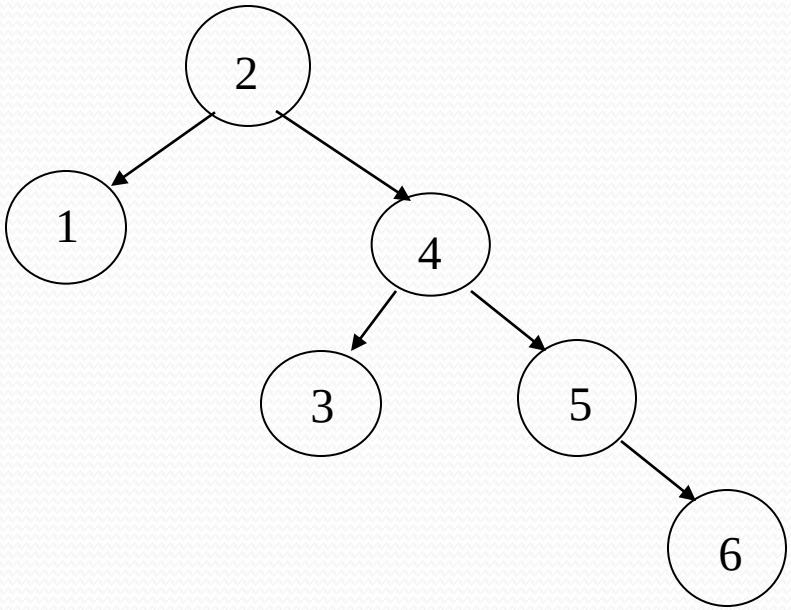
# Example

Insert 3,2,1,4,5,6,7



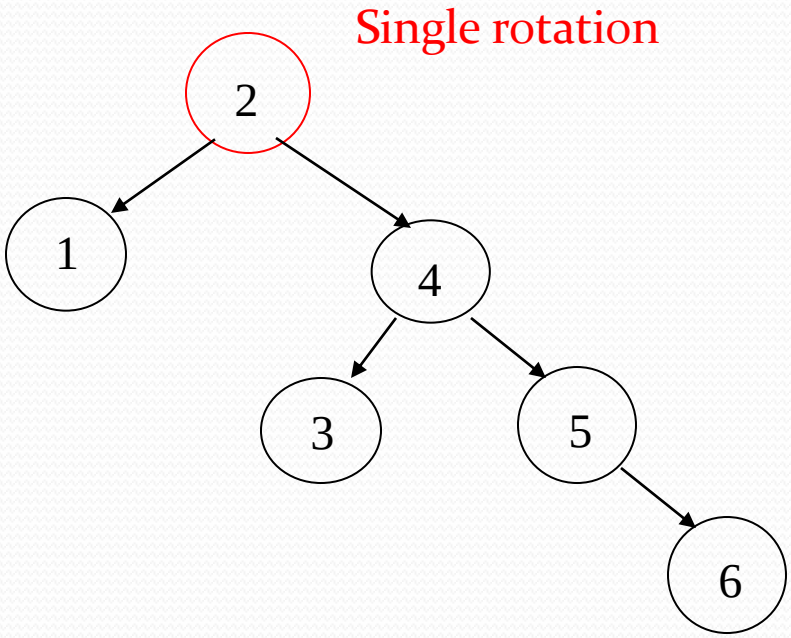
# Example

Insert 3,2,1,4,5,6,7



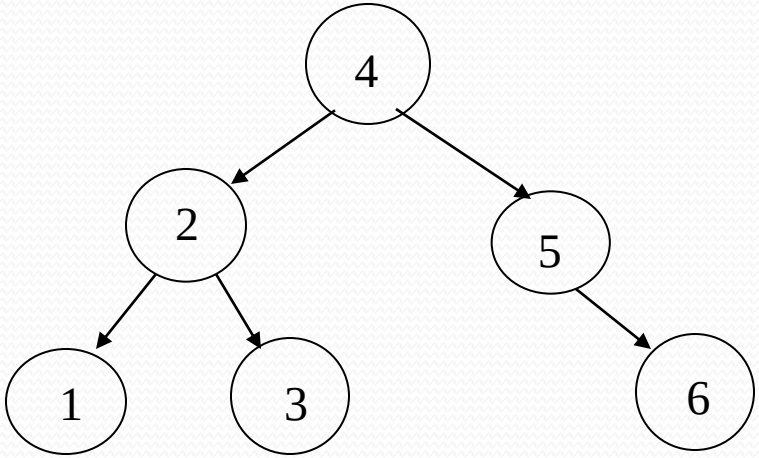
# Example

Insert 3,2,1,4,5,6,7



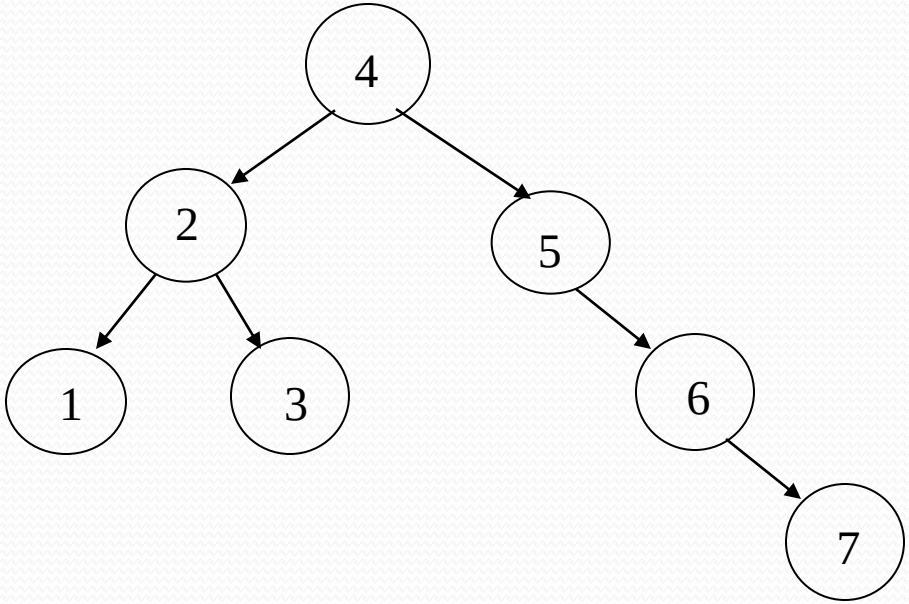
# Example

Insert 3,2,1,4,5,6,7



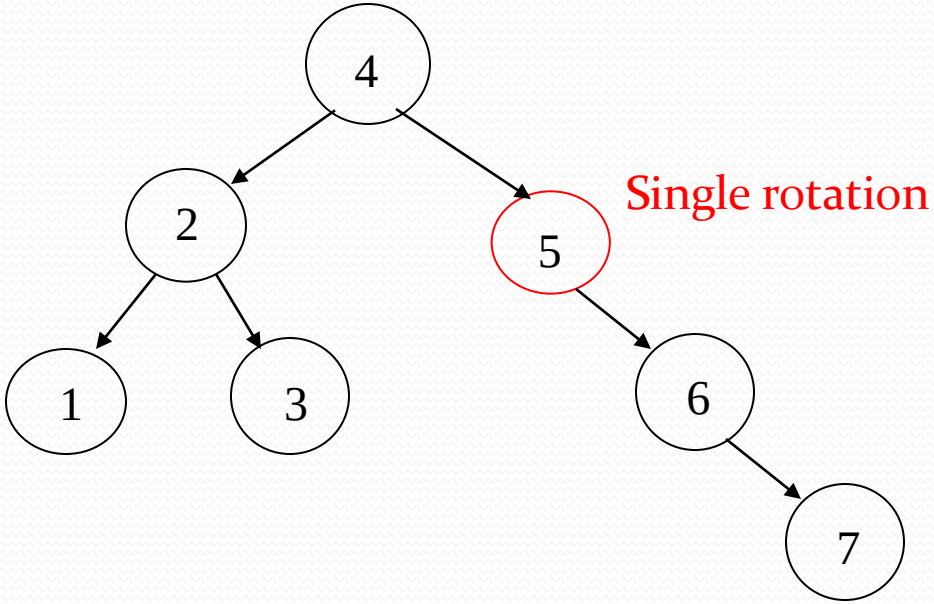
# Example

Insert 3,2,1,4,5,6,7



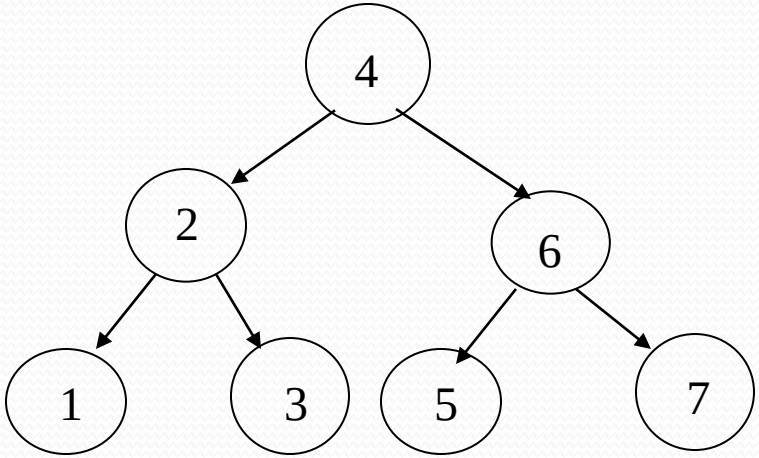
# Example

Insert 3,2,1,4,5,6,7



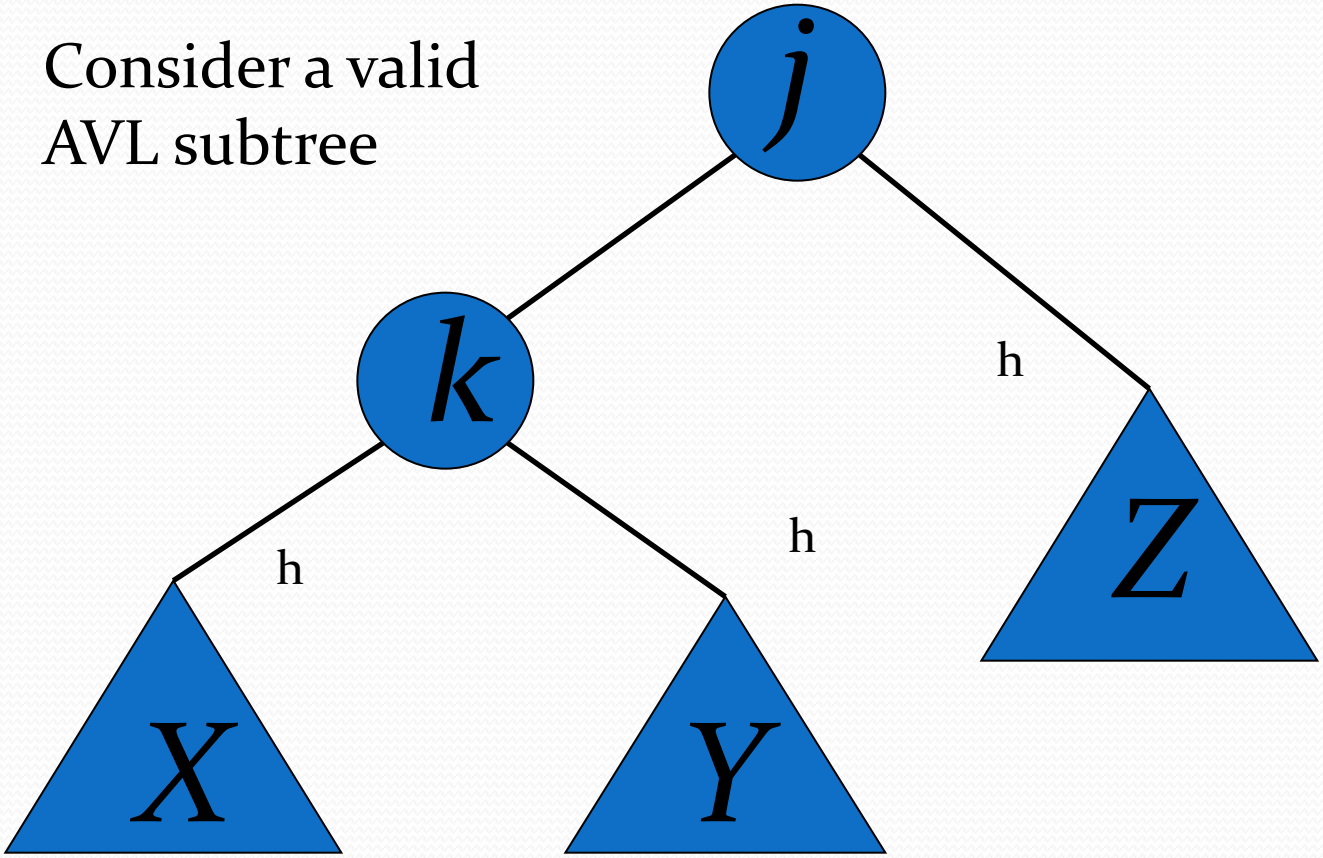
# Example

Insert 3,2,1,4,5,6,7

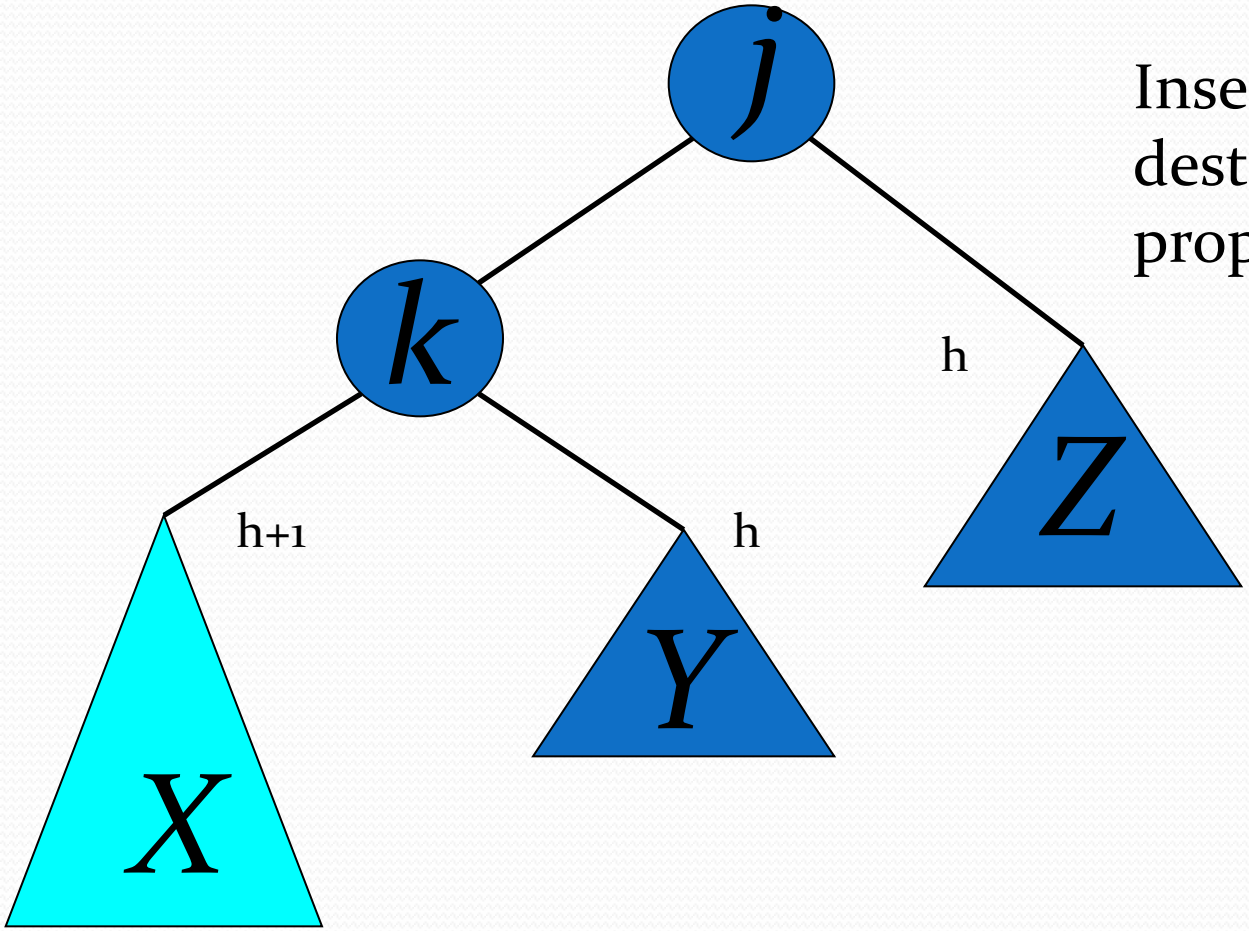


# AVL Insertion: Outside Case

Consider a valid AVL subtree

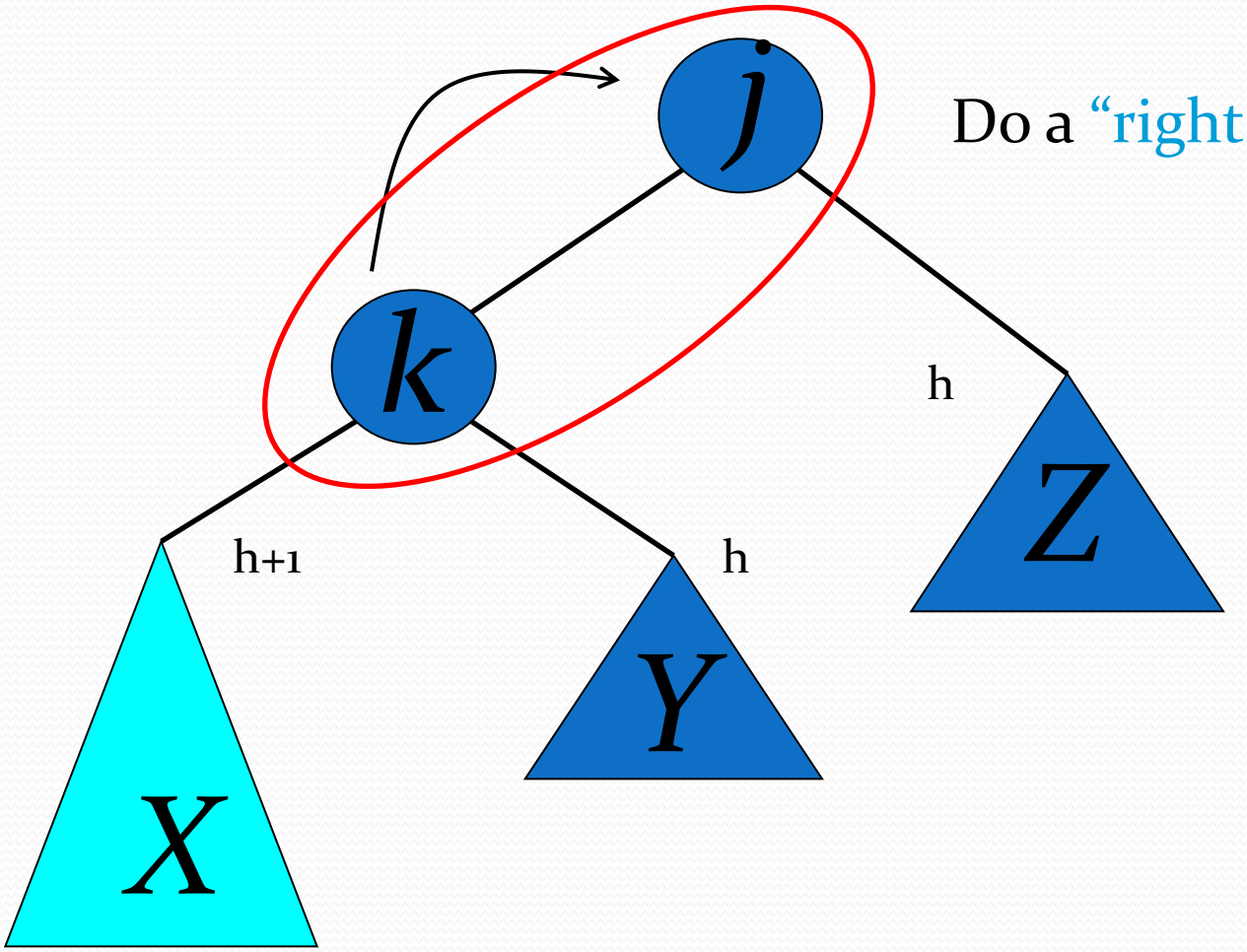


# AVL Insertion: Outside Case



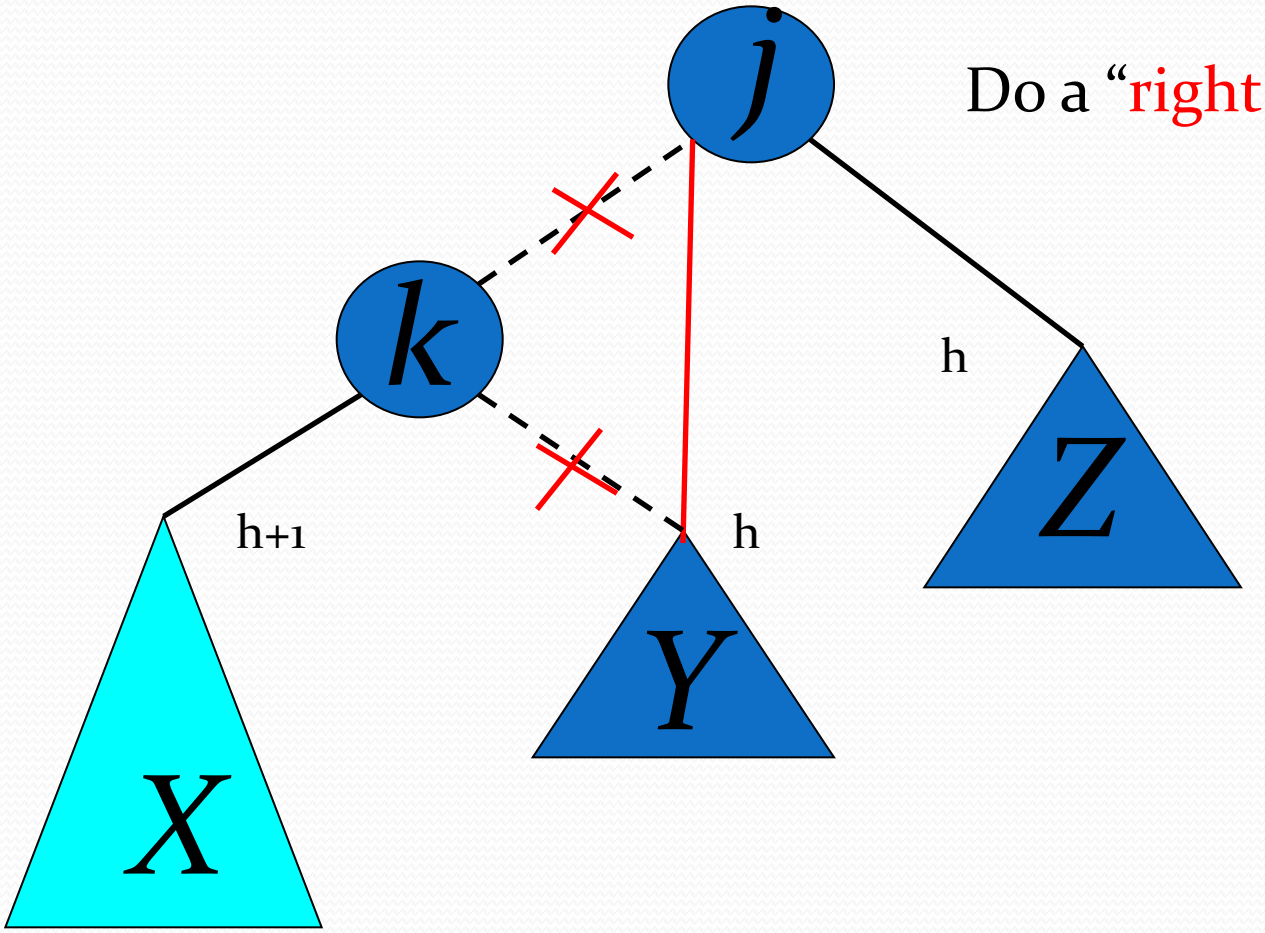

Inserting into X destroys the AVL property at node j

# AVL Insertion: Outside Case



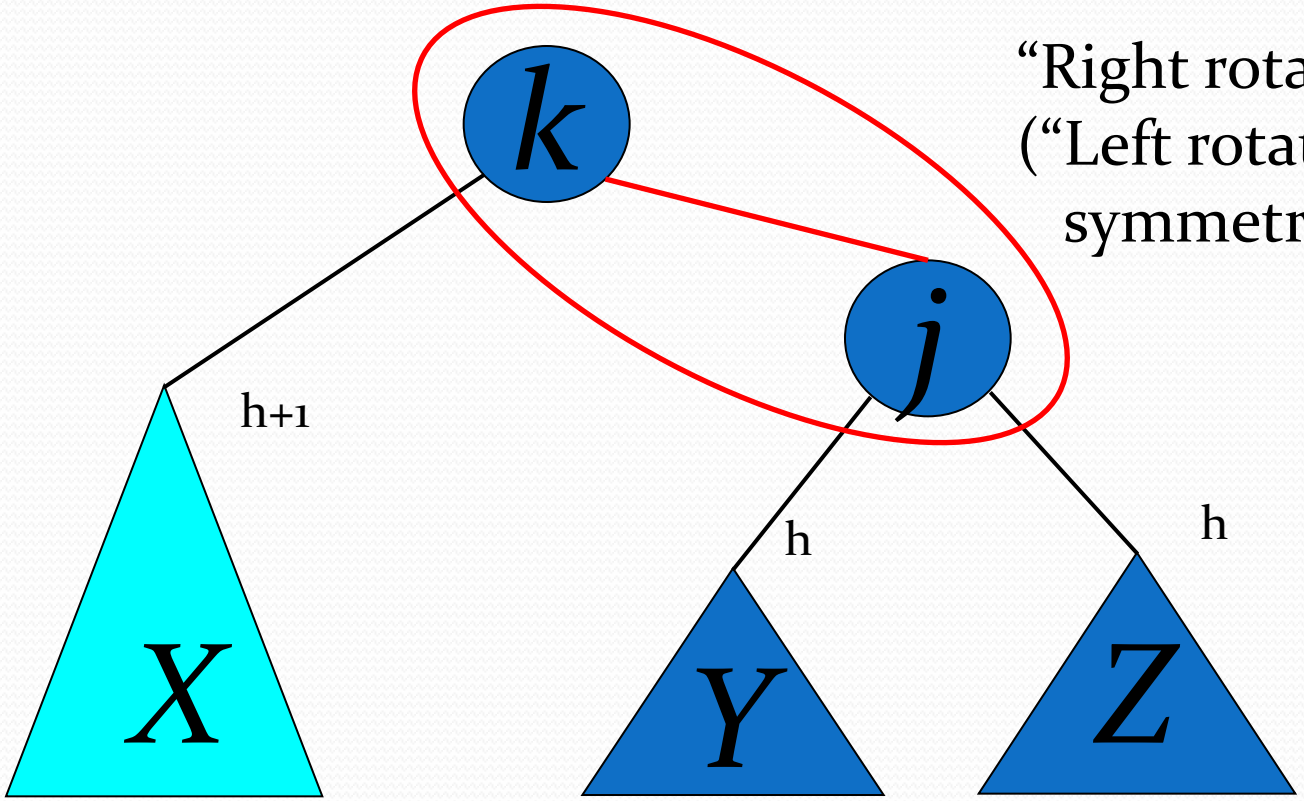

Do a "right rotation"

# Single right rotation




Do a "right rotation"

# Outside Case Completed

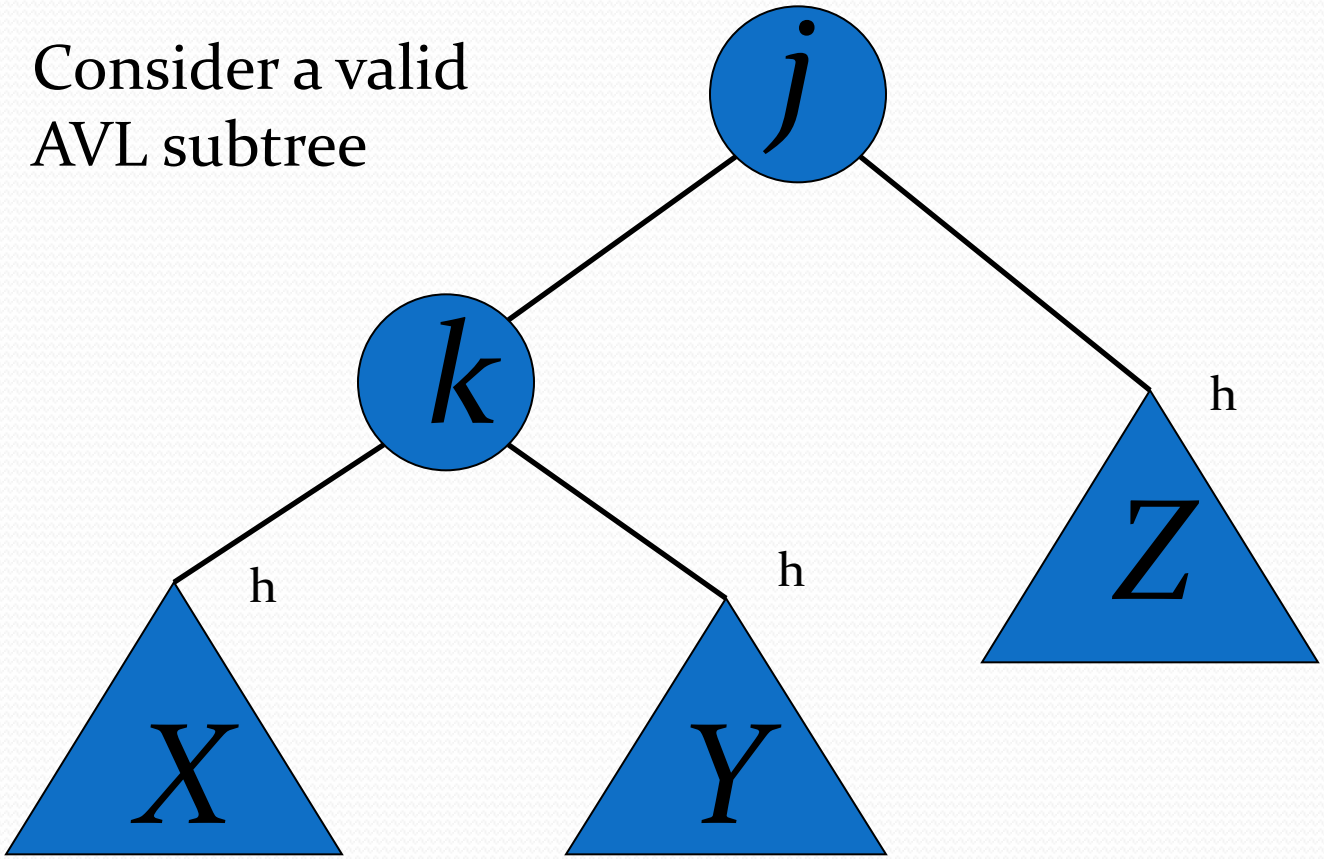



"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

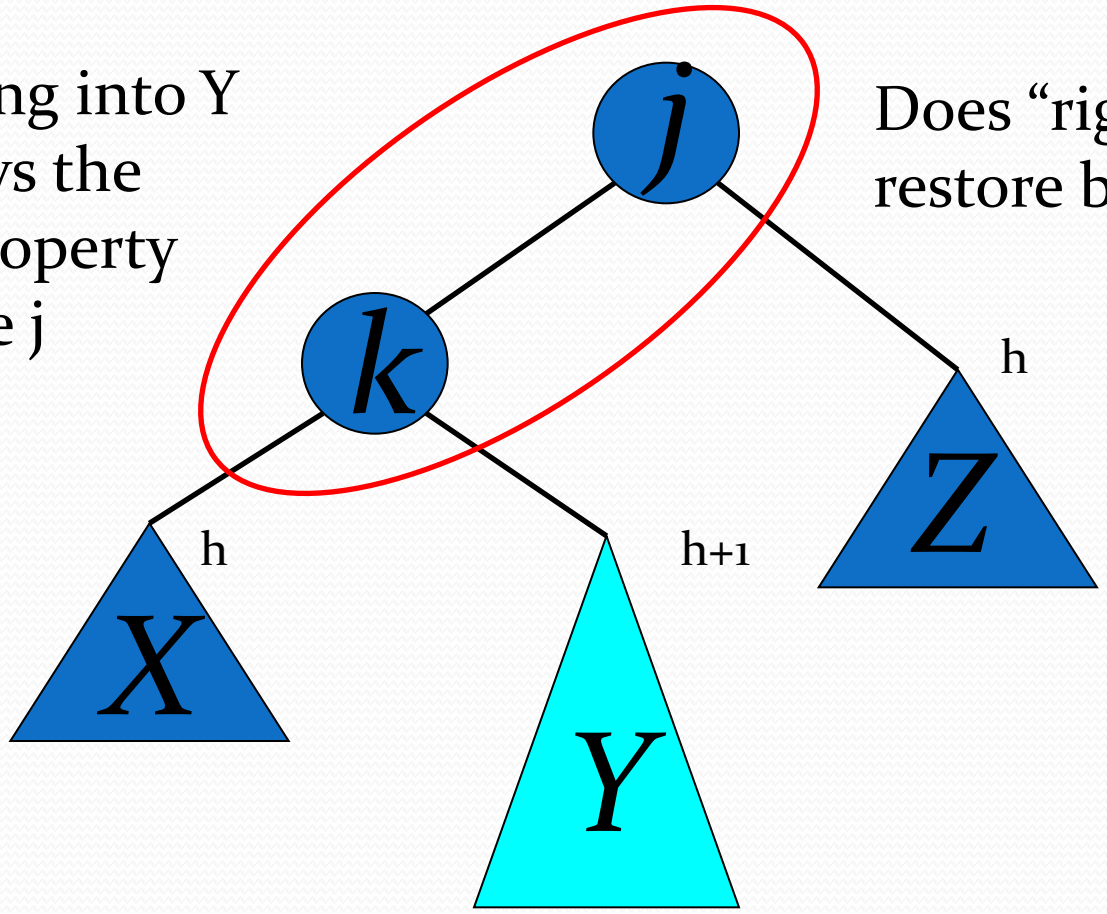
# AVL Insertion: Inside Case

Consider a valid AVL subtree



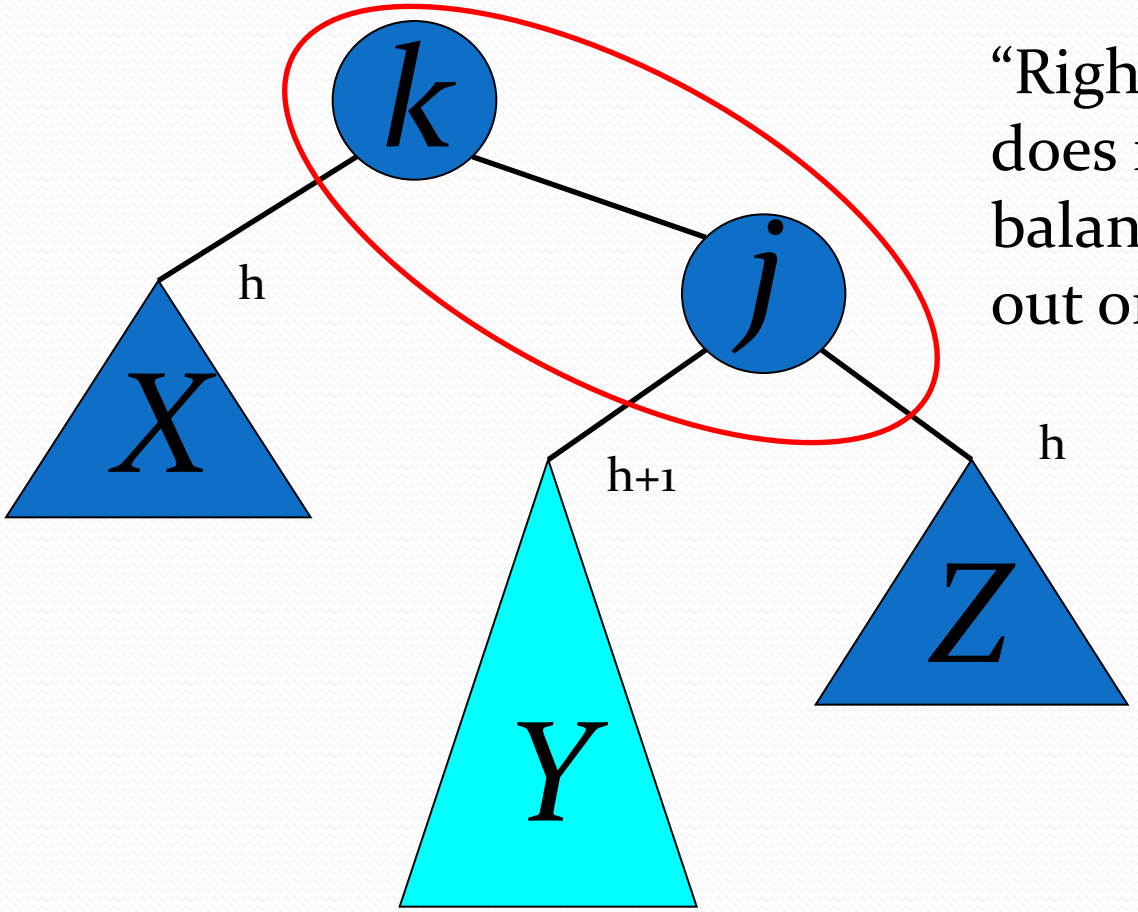
# AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j




Does “right rotation” restore balance?

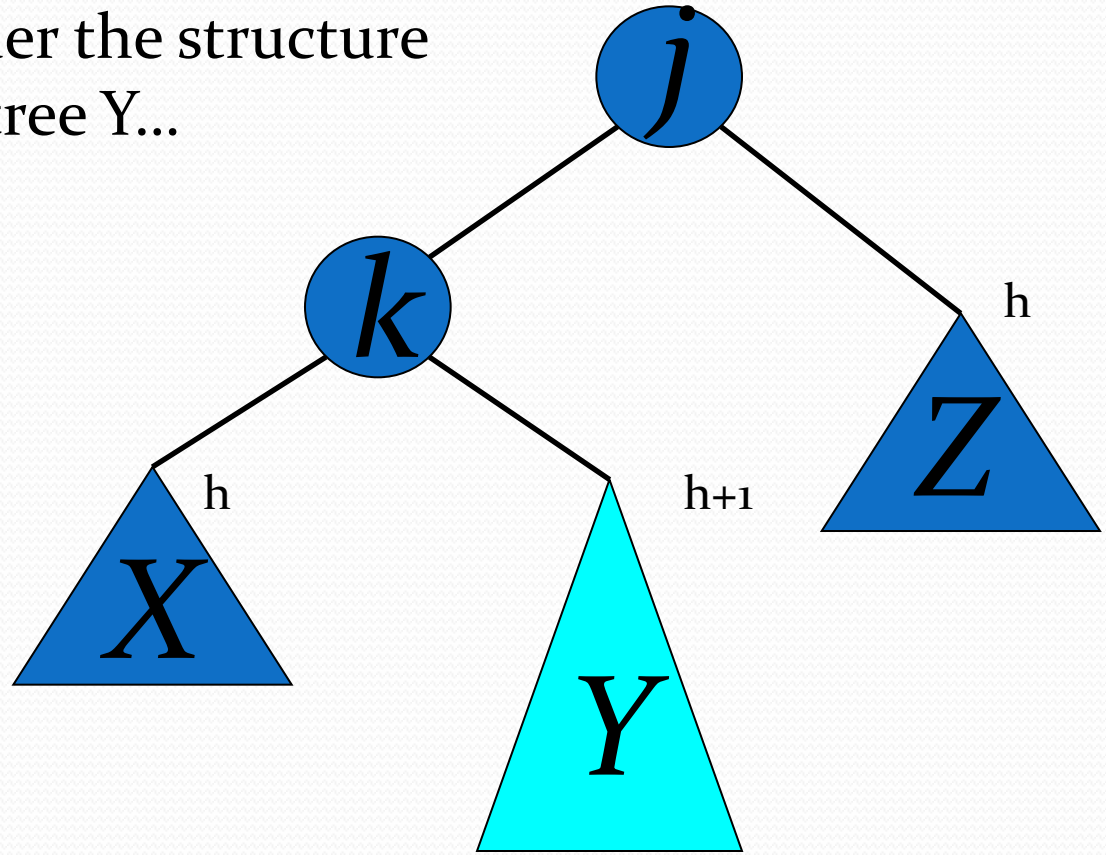
# AVL Insertion: Inside Case




"Right rotation" does not restore balance… now k is out of balance

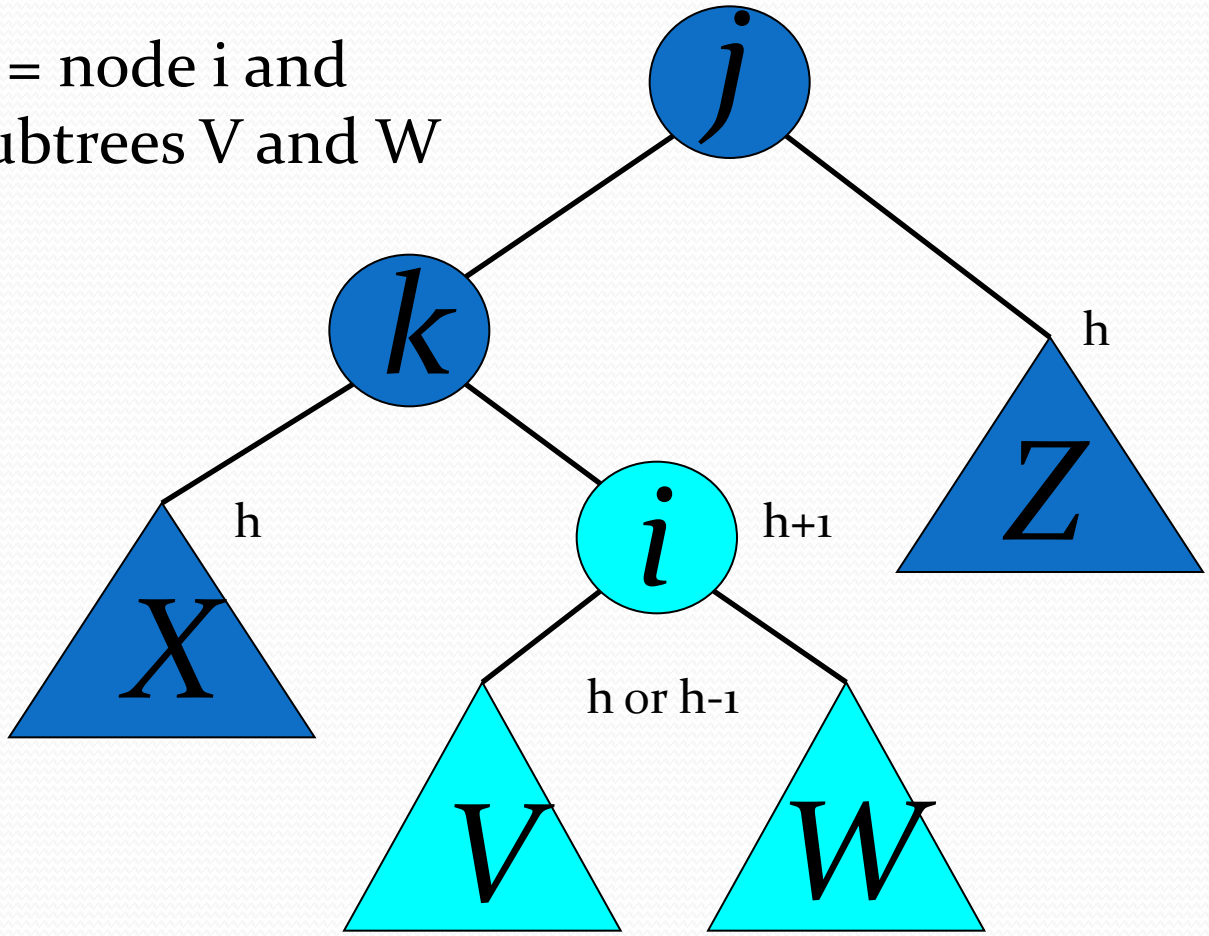
# AVL Insertion: Inside Case

Consider the structure of subtree Y…

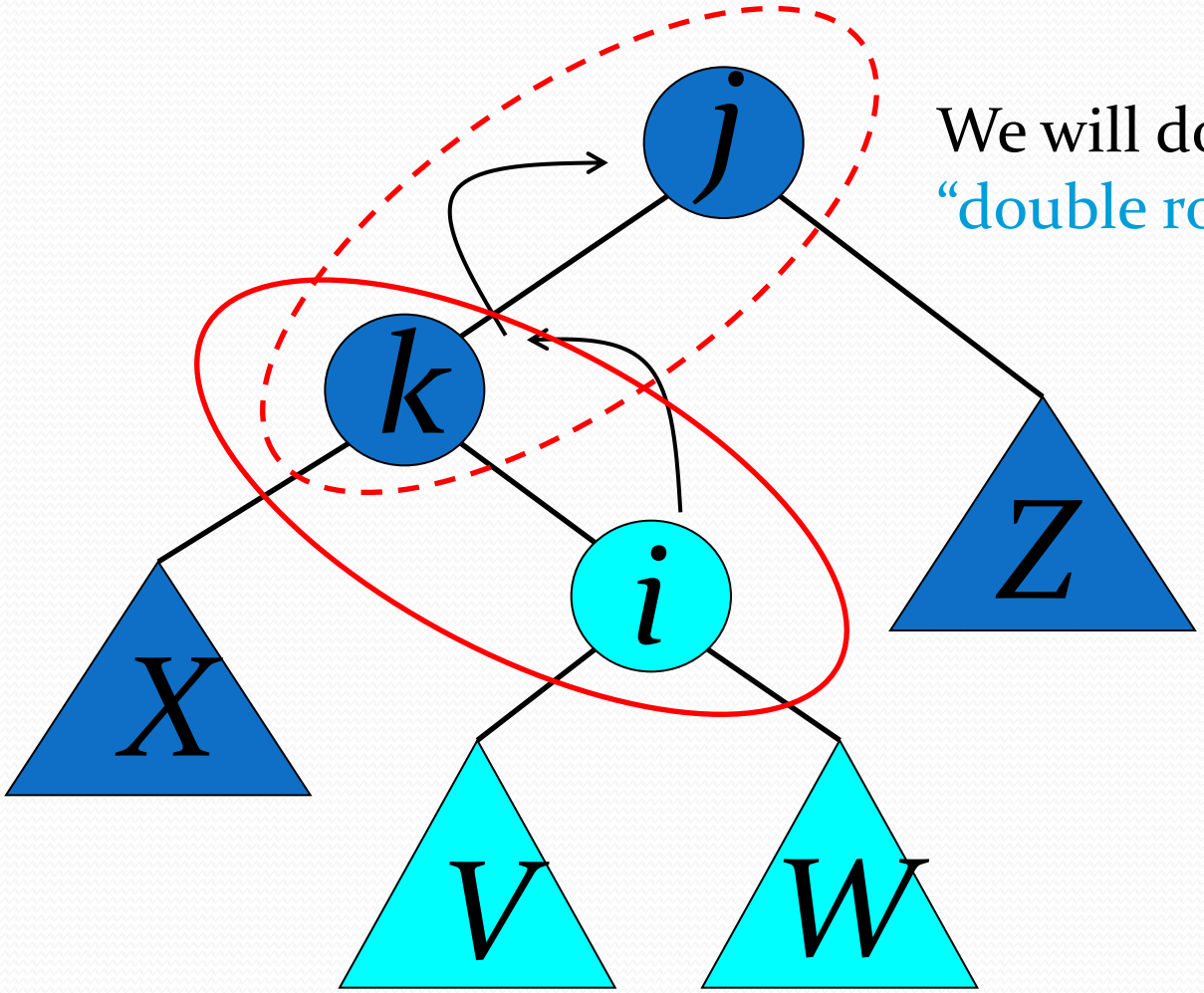


# AVL Insertion: Inside Case

Y = node i and
subtrees V and W

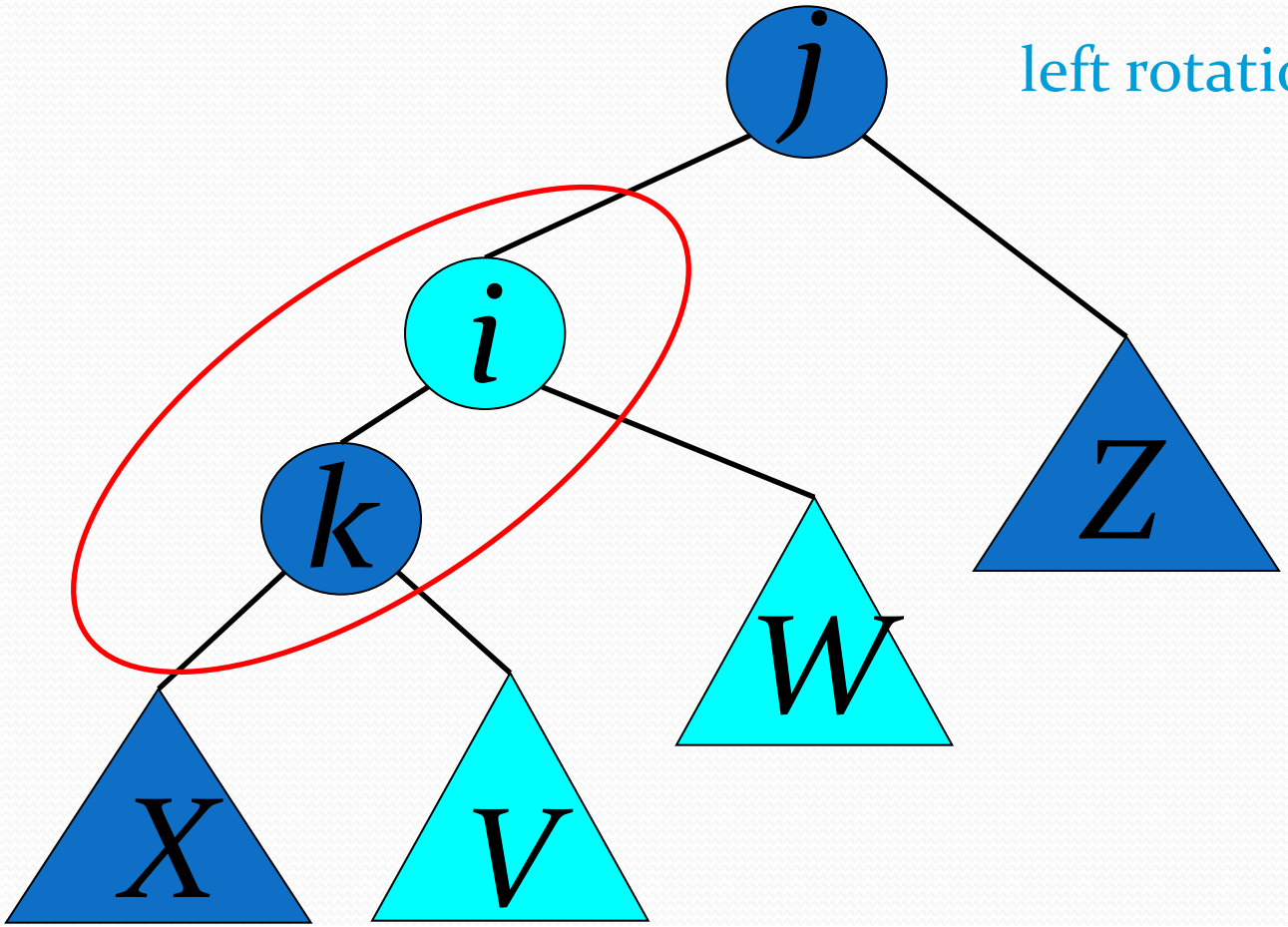


# AVL Insertion: Inside Case



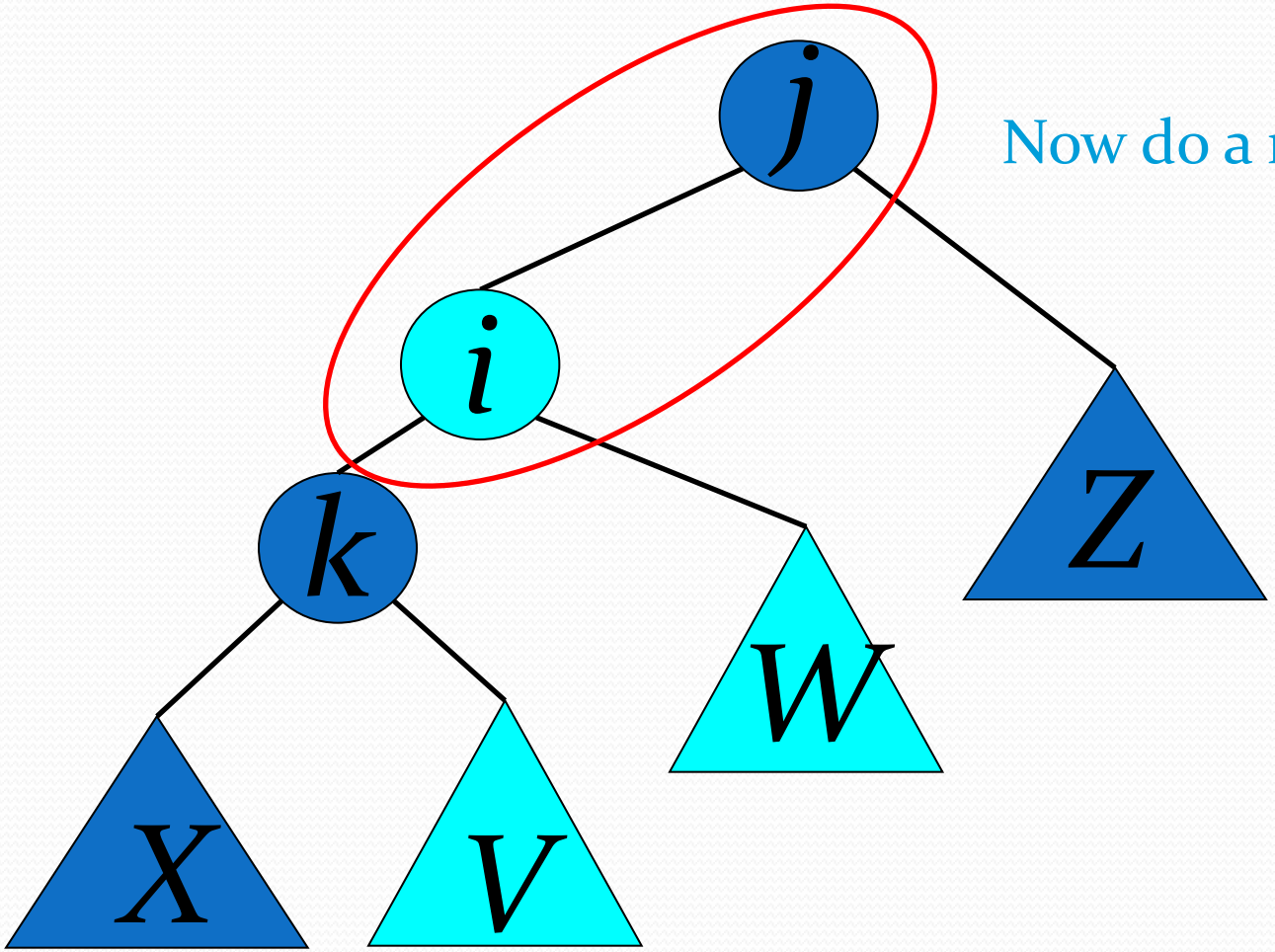

We will do a left-right "double rotation" . . .

# Double rotation : first rotation




left rotation complete

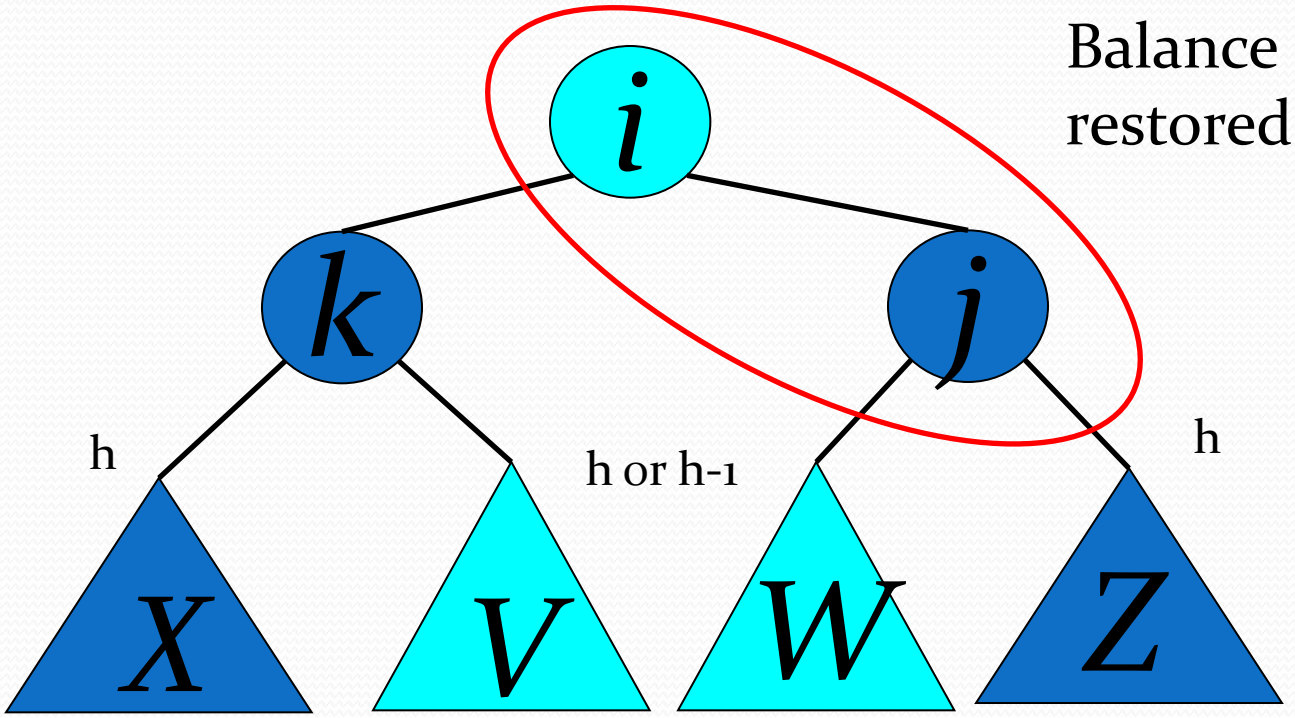
# Double rotation : second rotation




Now do a right rotation

# Double rotation : second rotation

right rotation complete




Balance has been restored

# B-Tree

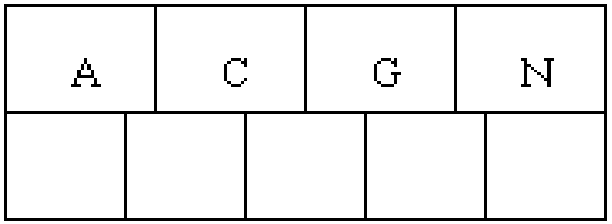
B-tree is a fairly well-balanced tree.

- All leaves are on the bottom level.
- All internal nodes (except perhaps the root node) have at least cell(m / 2) (nonempty) children.
- The root node can have as few as 2 children if it is an internal node, and can obviously have no children if the root node is a leaf (that is, the whole tree consists only of the root node).
- Each leaf node (other than the root node if it is a leaf) must contain at least ceil(m / 2) - 1 keys.

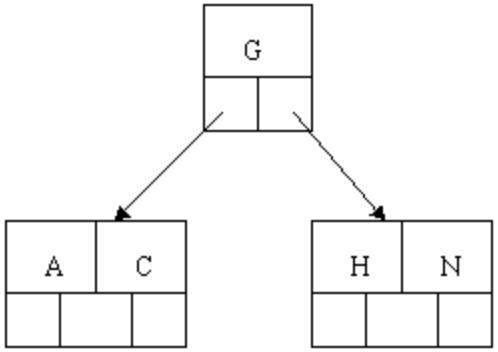
Let's work our way through an example similar to that given by Kruse. Insert the following letters into what is originally an empty B-tree of **order 5**:

**C N G A H E K Q M F W L T Z D P R X Y S**

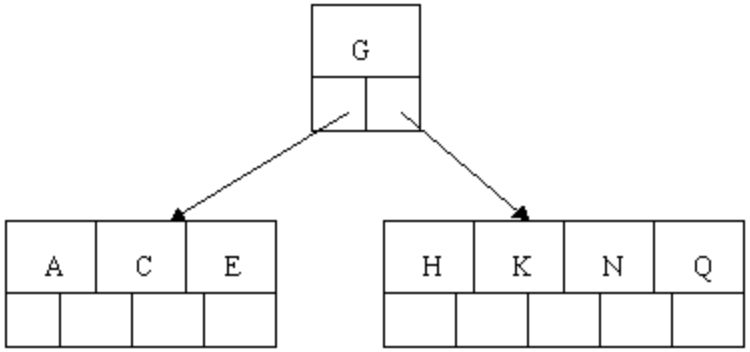
Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node. Note that in practice we just leave the A and C in the current node and place the H and N into a new node to the right of the old one.

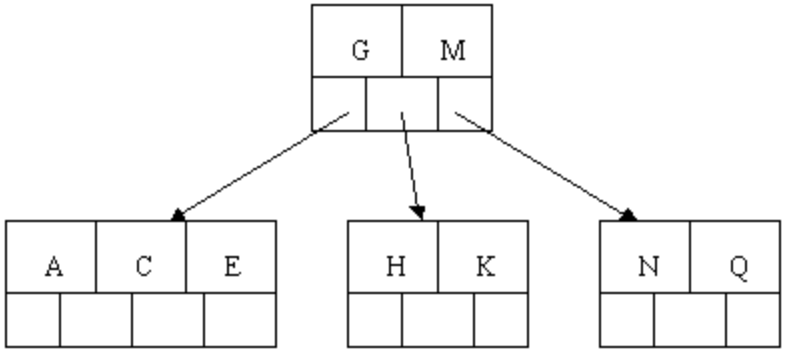



Inserting E, K, and Q proceeds without requiring any splits:

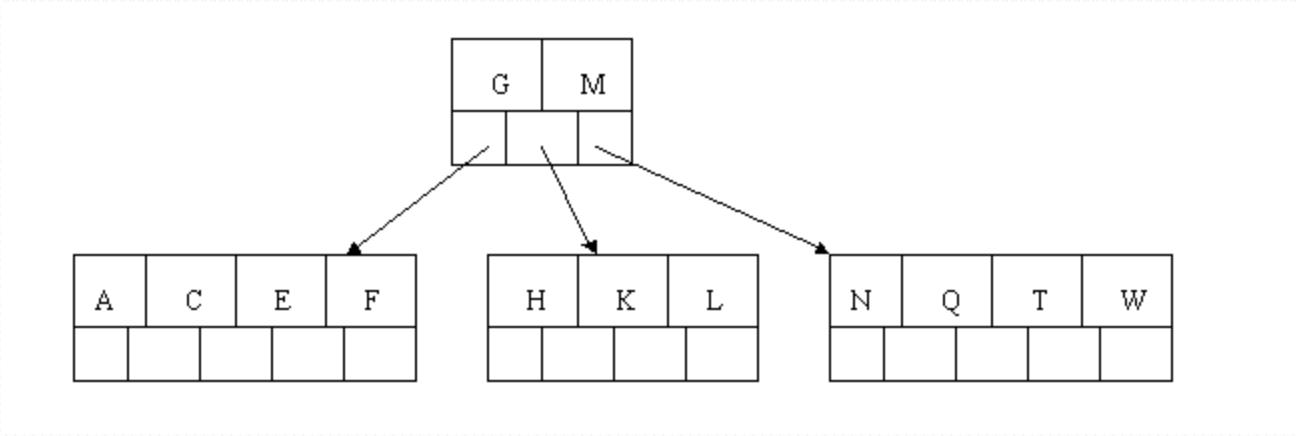



**H E K Q M F W L T Z D P R X Y S**

Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.

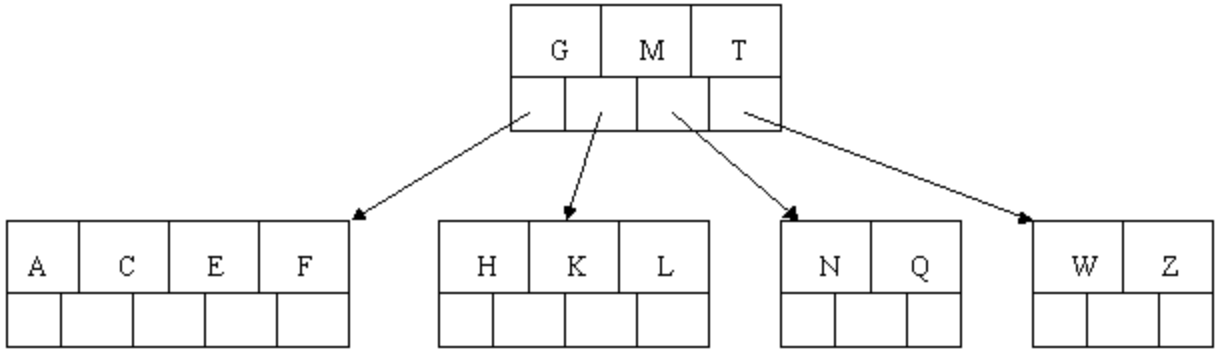



The letters F, W, L, and T are then added without needing any split.

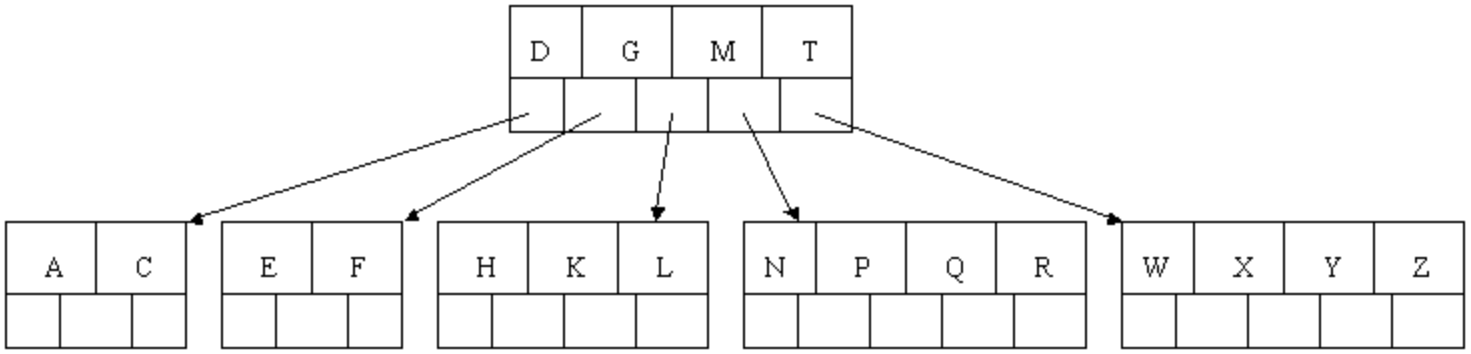



**M F W L T Z D P R X Y S**

When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.

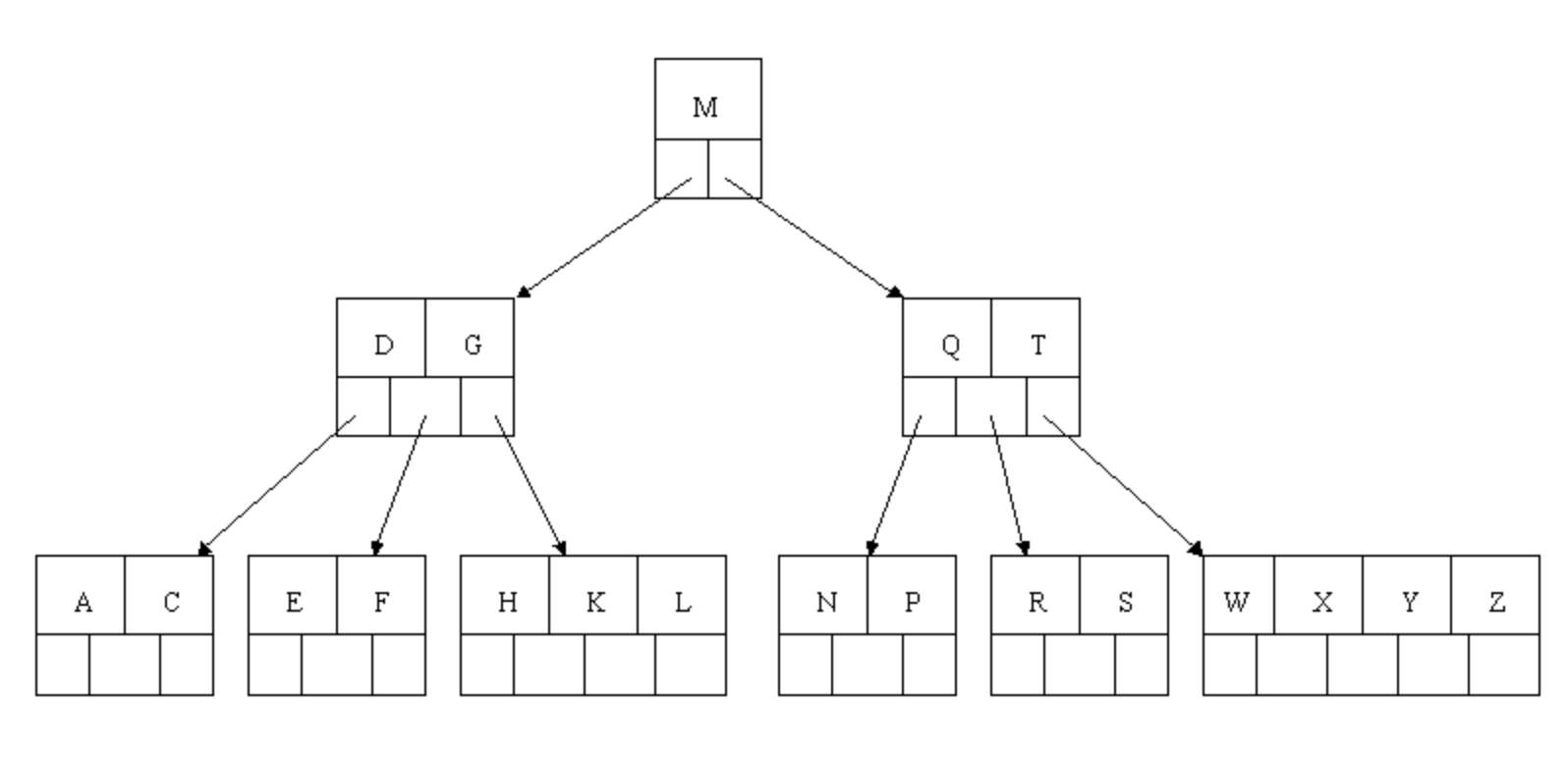



The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:




**Z D P R X Y S**

Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G.



# HEAP

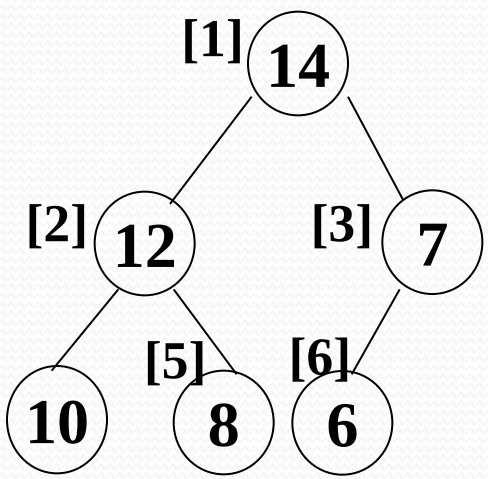
A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children. A *max heap* is a complete binary tree that is also a max tree.

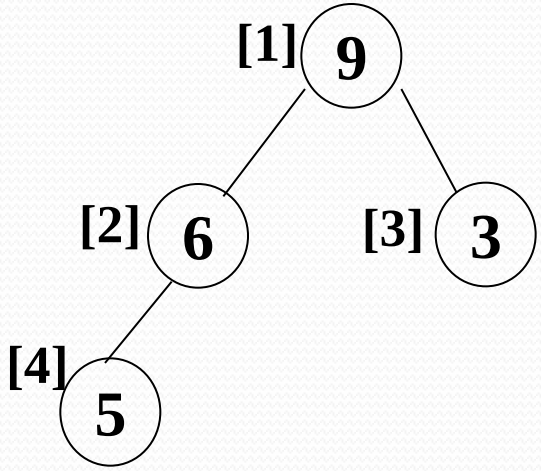
A *min tree* is a tree in which the key value in each node is no larger than the key values in its children. A *min heap* is a complete binary tree that is also a min tree.

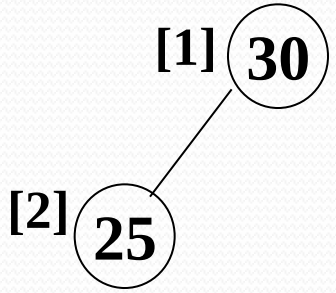
Operations on heaps:

- creation of an empty heap
- insertion of a new element into the heap;
- deletion of the largest element from the heap

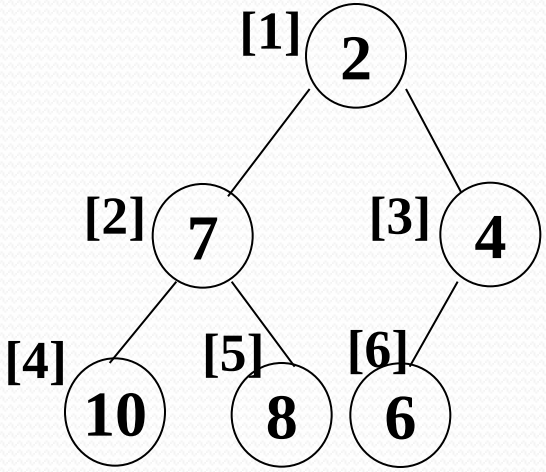
# Max Heap

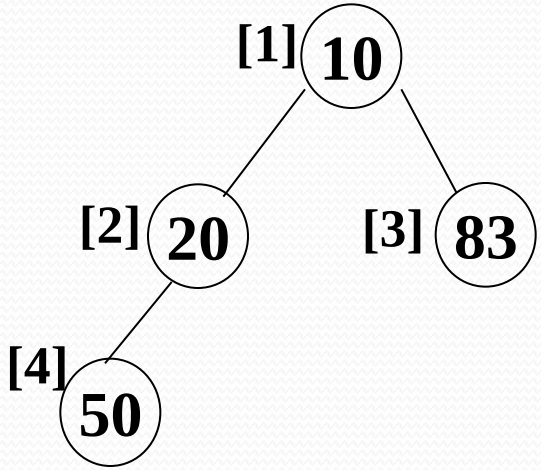


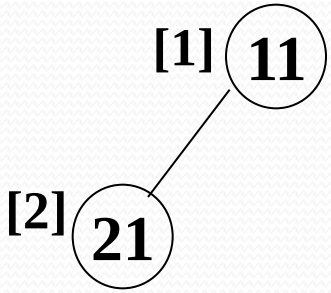



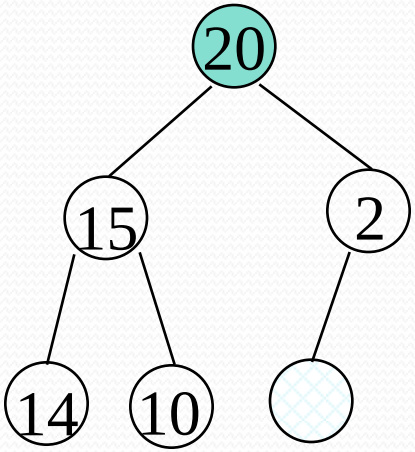

# Min Heap



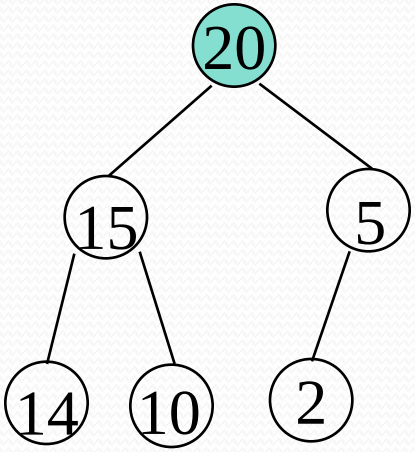



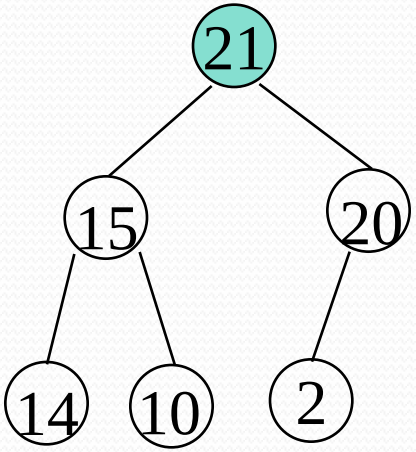
# Example of Insertion to Max Heap




initial location of new node




insert 5 into heap




insert 21 into heap

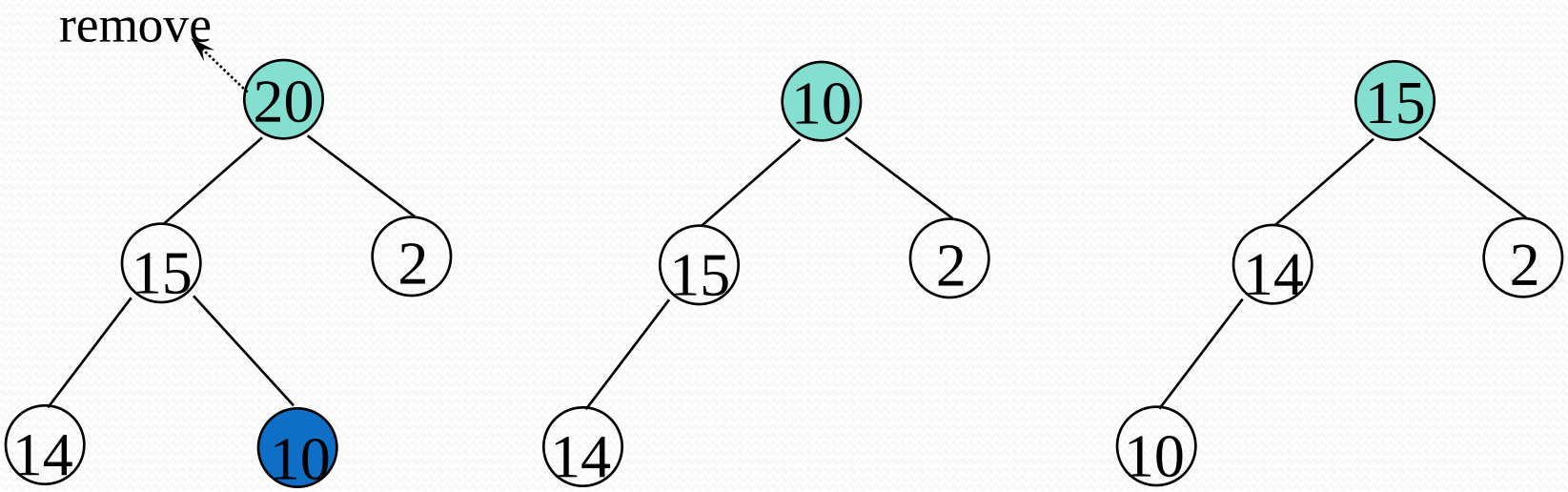
# Insertion into a Max Heap

```
void insert_max_heap(element item, int *n)
{
  int i;
  if (HEAP_FULL(*n)) {
    fprintf(stderr, "the heap is full.\n");
    exit(1);
  }
  i = ++(*n);
  while ((i!=1)&&(item.key>heap[i/2].key)) {
    heap[i] = heap[i/2];
    i /= 2;
  }
  heap[i]= item;
}
```

$2^k-1=n \Longrightarrow k=\lceil \log_2(n+1) \rceil$

$O(\log_2 n)$

# Example of Deletion from Max Heap

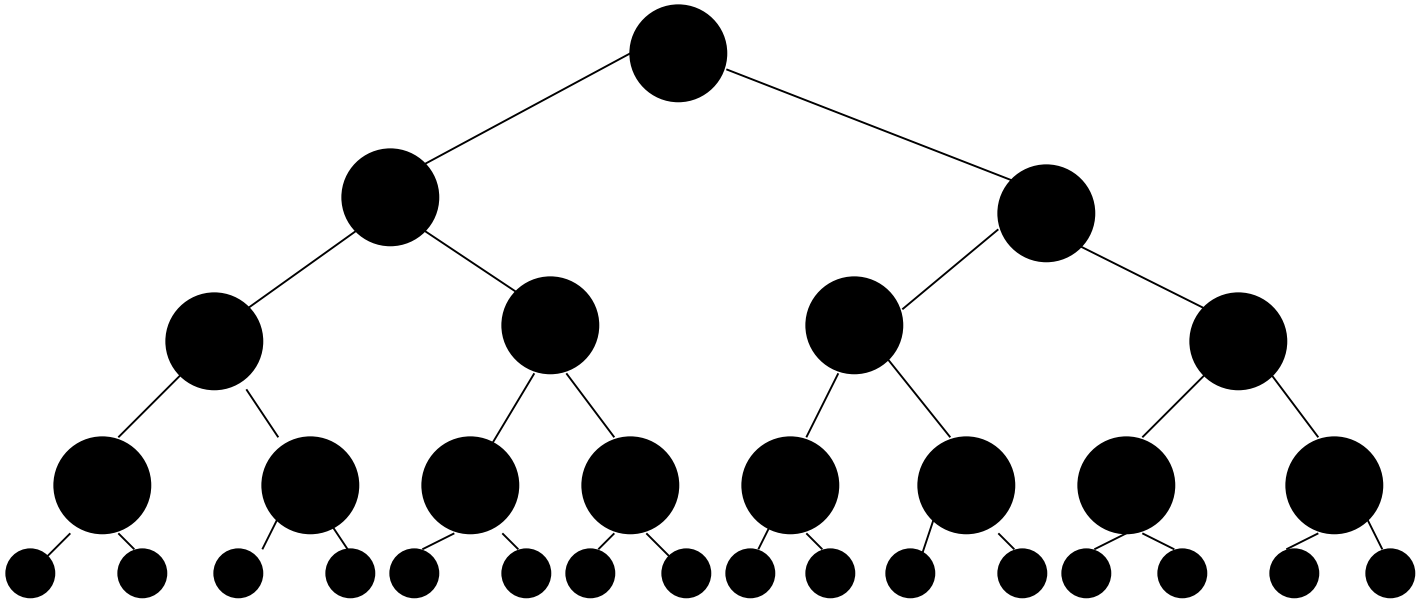


# Deletion from a Max Heap

```
element delete_max_heap(int *n)
{
  int parent, child;
  element item, temp;
  if (HEAP_EMPTY(*n)) {
    fprintf(stderr, "The heap is empty\n");
    exit(1);
  }
  /* save value of the element with the
     highest key */
  item = heap[1];
  /* use last element in heap to adjust heap */
  temp = heap[(*n)--];
  parent = 1;
  child = 2;
```

```
while (child <= *n) {
    /* find the larger child of the current
       parent */
    if ((child < *n)&&
        (heap[child].key<heap[child+1].key))
      child++;
    if (temp.key >= heap[child].key) break;
    /* move to the next lower level */
    heap[parent] = heap[child];
    child *= 2;
  }
  heap[parent] = temp;
  return item;
}
```

# Red-Black Trees

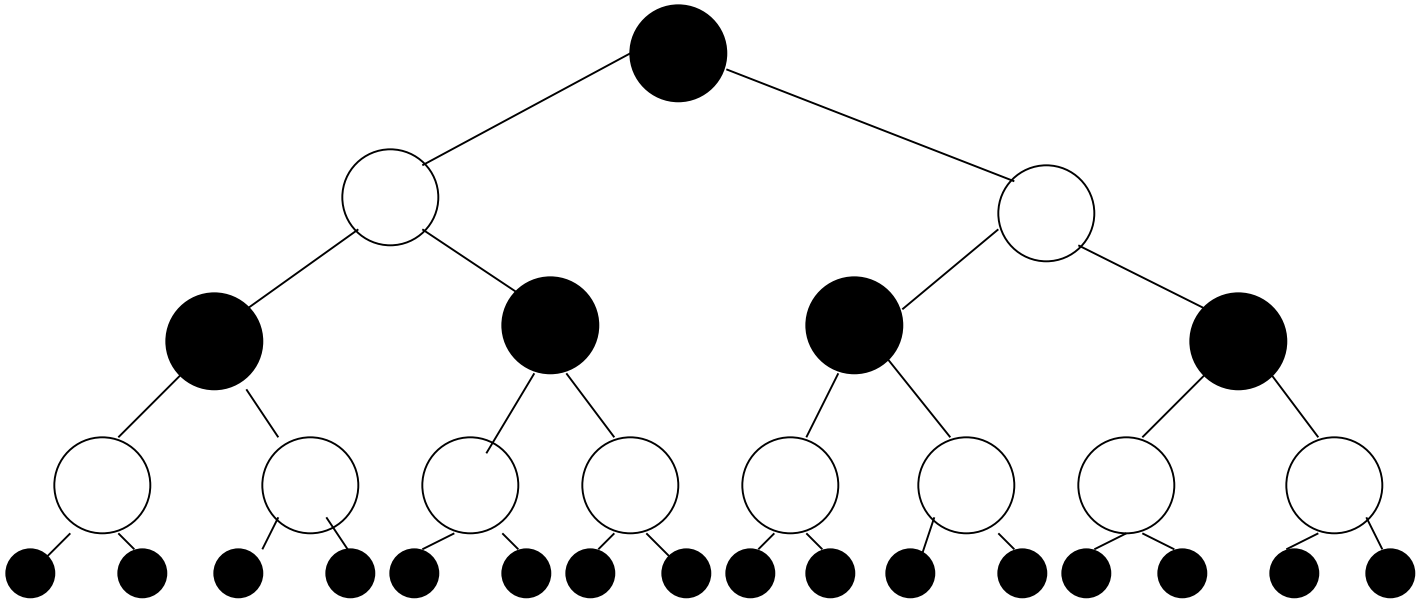


A Red-Black Tree with NULLs shown

Black-Height of the tree = 4

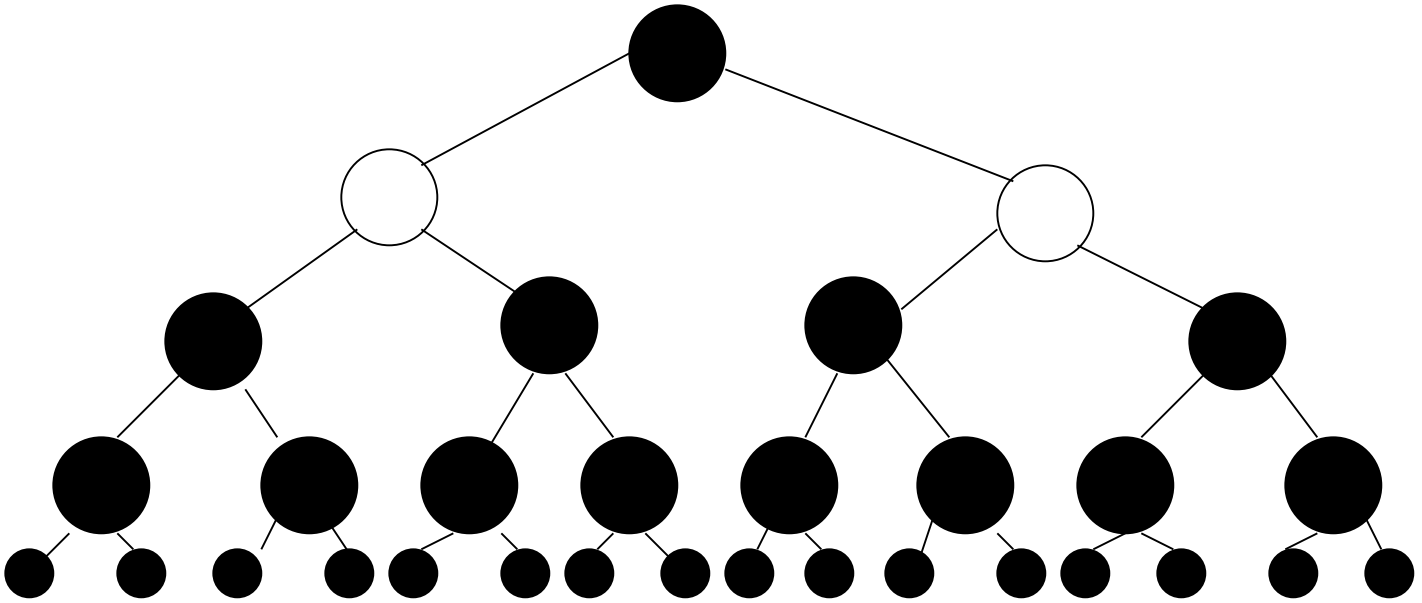
# Red-Black Trees

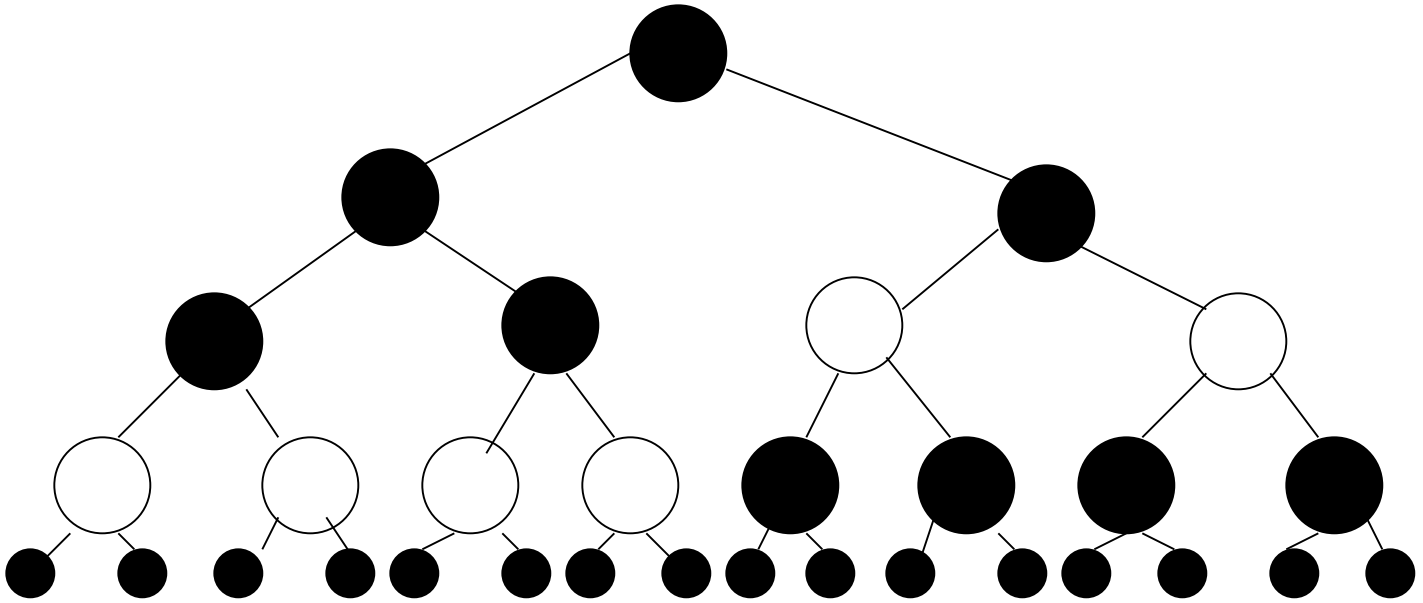
- Definition: A red-black tree is a binary search tree where:
  - Every node is either red or black.
  - Each NULL pointer is considered to be a black node
  - If a node is red, then both of its children are black.
  - Every path from a node to a leaf contains the same number of black nodes.
- Definition: The black-height of a node, n, in a red-black tree is the number of black nodes on any path to a leaf, not counting n.



A valid Red-Black Tree

Black-Height = 2





Theorem 1 – Any red-black tree with root ***x***, has at least $\mathbf{n = 2^{bh(x)} - 1}$ internal nodes, where bh(x) is the black height of node x.

Proof: by induction on height of x.

Theorem 2 – In a red-black tree, at least half the nodes on any path from the root to a leaf must be black.

Proof – If there is a red node on the path, there must be a corresponding black node.

Theorem 3 – In a red-black tree, no path from any node, N, to a leaf is more than twice as long as any other path from N to any other leaf.

Proof: By definition, every path from a node to any leaf contains the same number of black nodes. By Theorem 2, a least ½ the nodes on any such path are black. Therefore, there can no more than twice as many nodes on any path from N to a leaf as on any other path. Therefore the length of every path is no more than twice as long as any other path

Theorem 4 –

A red-black tree with ***n*** internal nodes has height $h <= 2 \lg(n + 1)$.

Proof: Let h be the height of the red-black tree with root x. By Theorem 2,

$$bh(x) >= h/2$$

From Theorem 1, $n >= 2^{bh(x)} - 1$

Therefore $n >= 2^{h/2} - 1$

$$n + 1 >= 2^{h/2}$$

$$\lg(n + 1) >= h/2$$

$$2\lg(n + 1) >= h$$

# Bottom –Up Insertion

- Insert node as usual in BST
- Color the Node RED
- What Red-Black property may be violated?
  - Every node is Red or Black
  - Leaf nodes are Black NULLS
  - If node is Red, both children must be Black
  - Every path from node to descendant leaf must contain the same number of Blacks

# Bottom Up Insertion

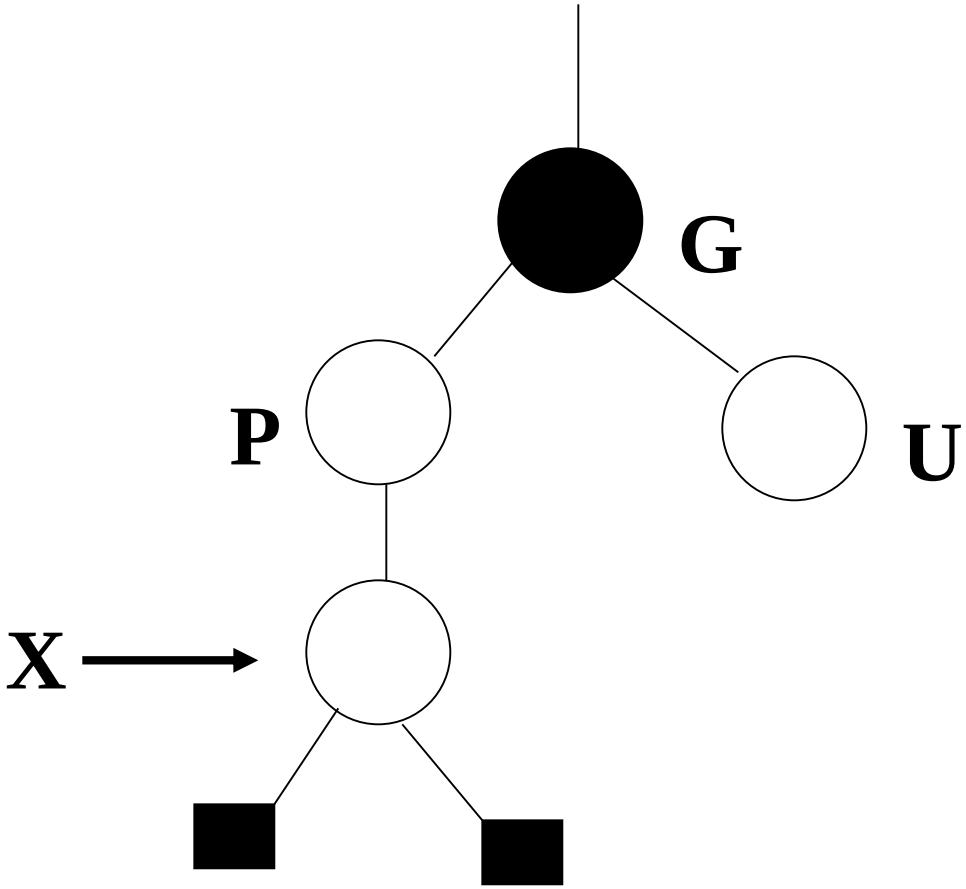
- Insert node; Color it RED; X is pointer to it
- Cases

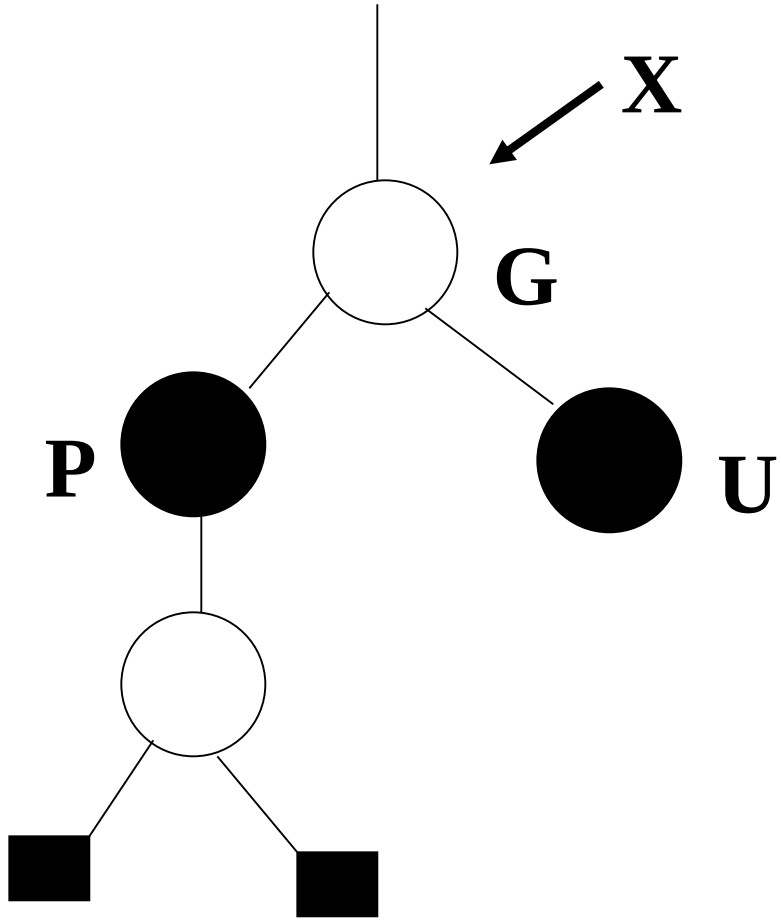
  0: X is the root -- color it black

  1: Both parent and uncle are red -- color parent and uncle black, color grandparent red, point X to grandparent, check new situation

  2 (zig-zag): Parent is red, but uncle is black. X and its parent are opposite type children -- color grandparent red, color X black, rotate left on parent, rotate right on grandparent

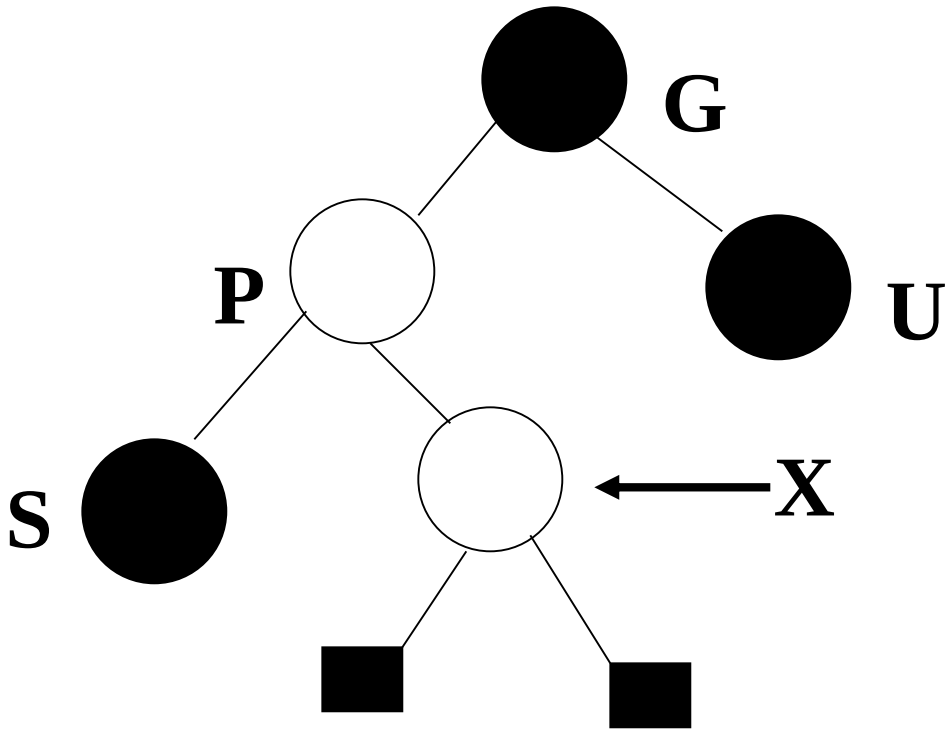
  3 (zig-zig): Parent is red, but uncle is black. X and its parent are both left or both right children -- color parent black, color grandparent red, rotate right on grandparent

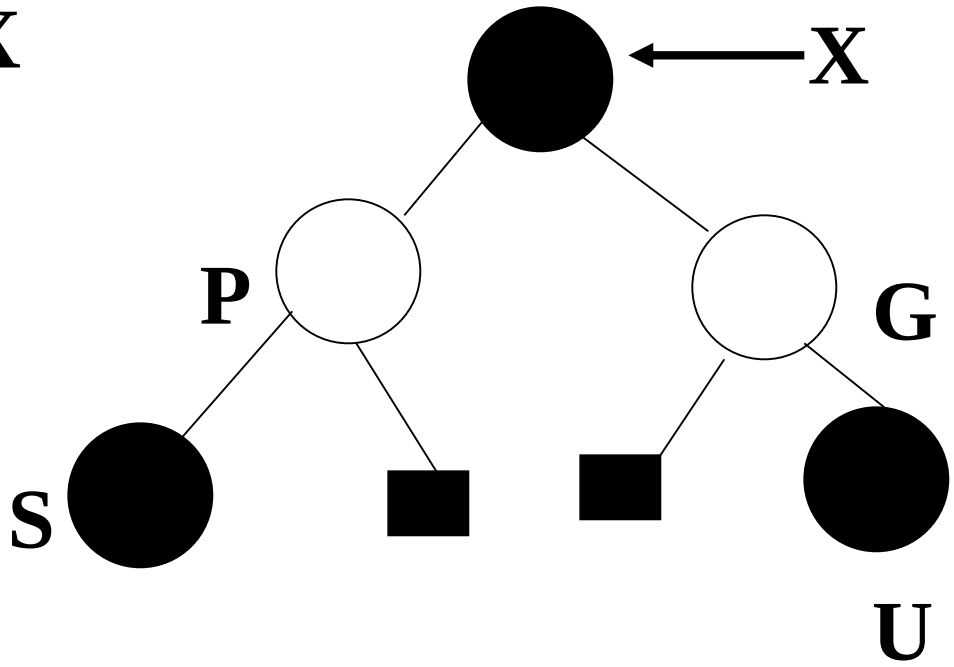



Case 1 – U is Red

Just Recolor and move up

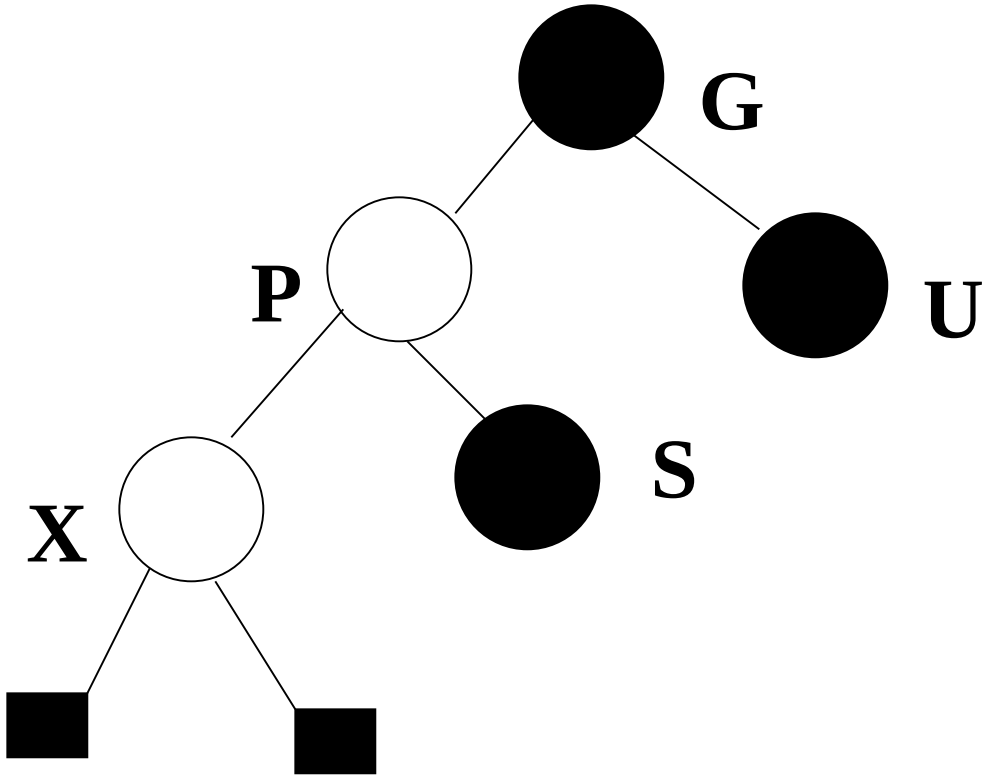



Case 2 – Zig-Zag

Double Rotate
X around P; X around G

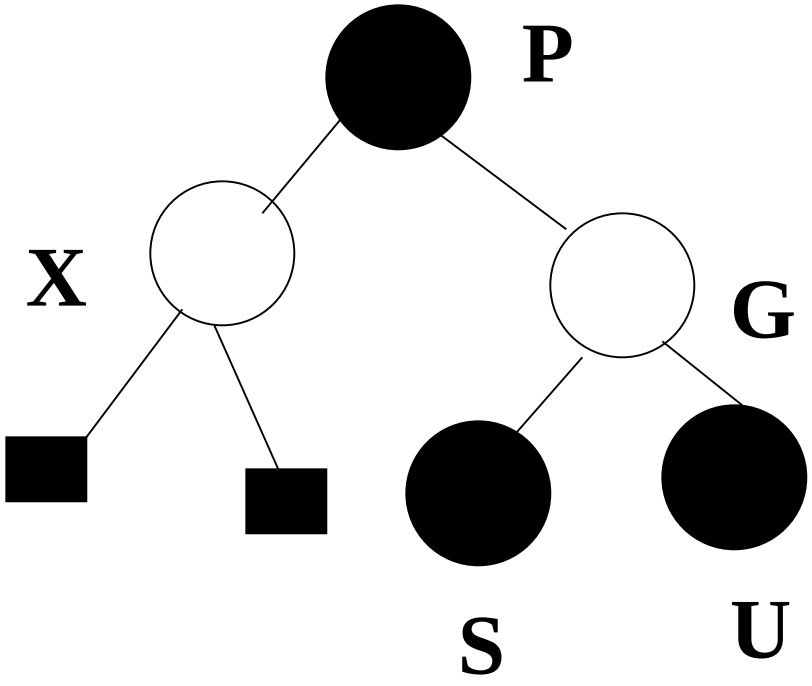
Recolor G and X



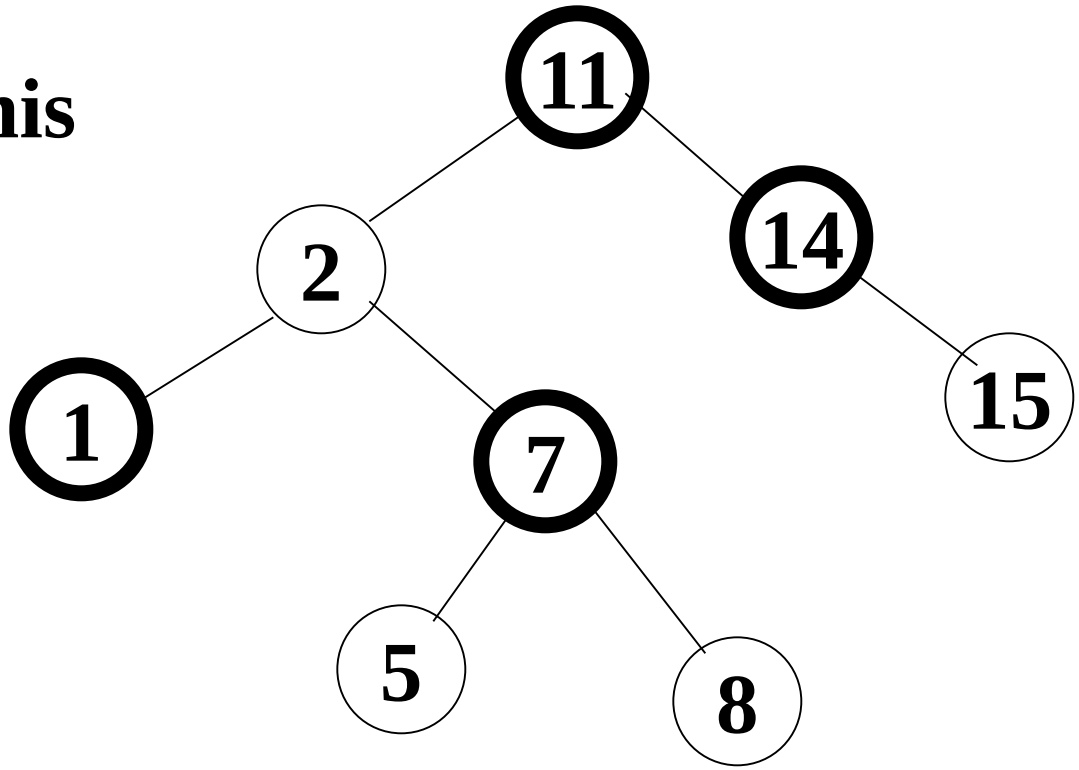

Case 3 – Zig-Zig

Single Rotate P around G

Recolor P and G



**Insert 4 into this R-B Tree**




Black node

Red node

# Insertion Practice

Insert the values 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty Red-Black Tree

# Asymptotic Cost of Insertion

- O(lg n) to descend to insertion point
- O(1) to do insertion
- O(lg n) to ascend and readjust == worst case only for case 1

- Total: O(log n)

# What is Splay Tree?

- A balanced search tree data structure
- NIST Definition: A binary search tree in which operations that access nodes restructure the tree.
- Goodrich: A splay tree is a binary search tree T. The only tool used to maintain balance in T is the splaying step done after every search, insertion, and deletion in T.
- Kingston: A binary tee with splaying.

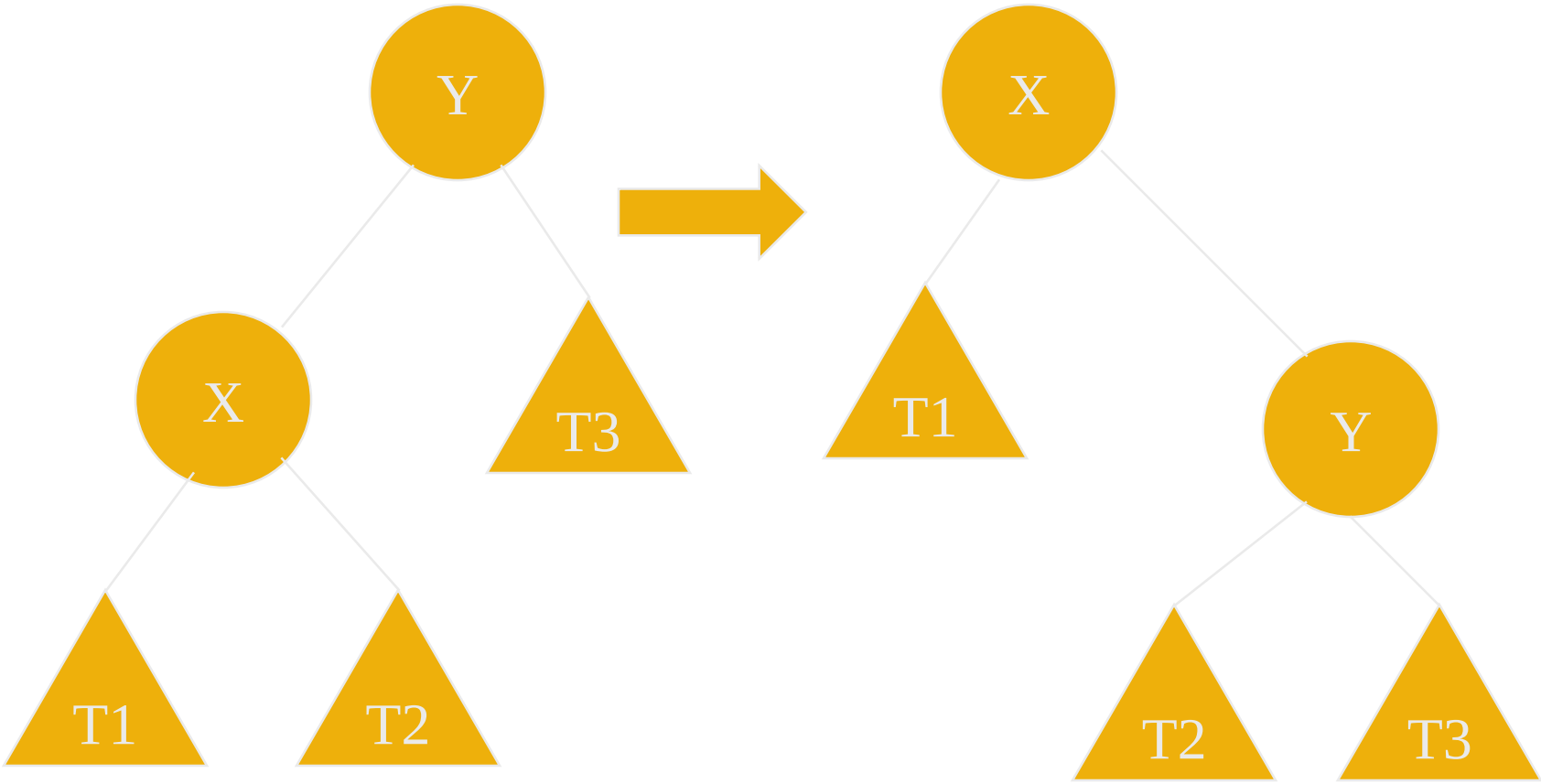
# Splaying

- Left and right rotation
- Move-to-root operation
- Zig-zig
- Zig-zag
- Zig
- Comparing move-to-root and splaying
- Example of splaying a node

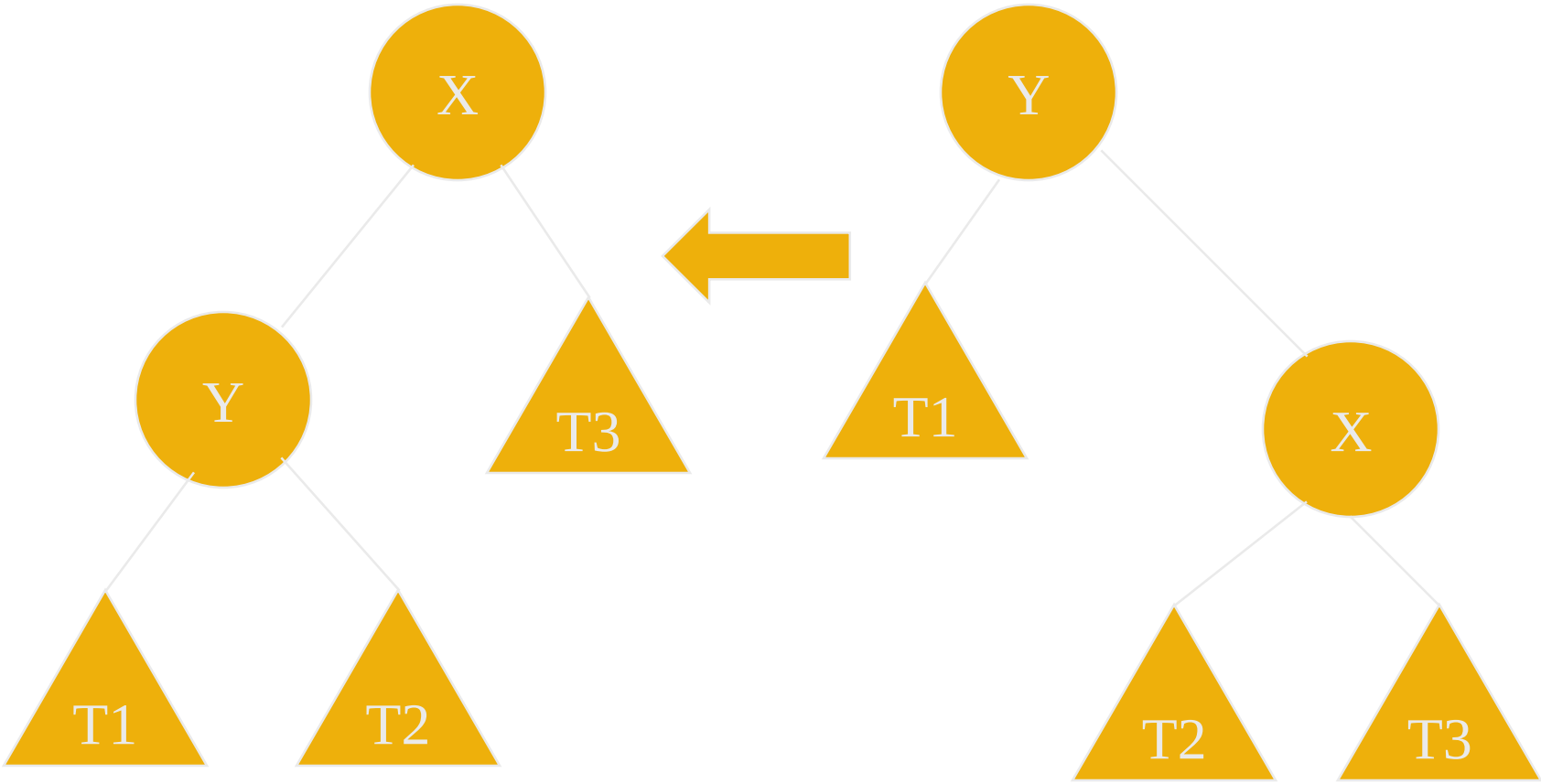
# Left and right rotation

Adel'son-Vel'skii and Landis (1962), Kingston

Left rotation:

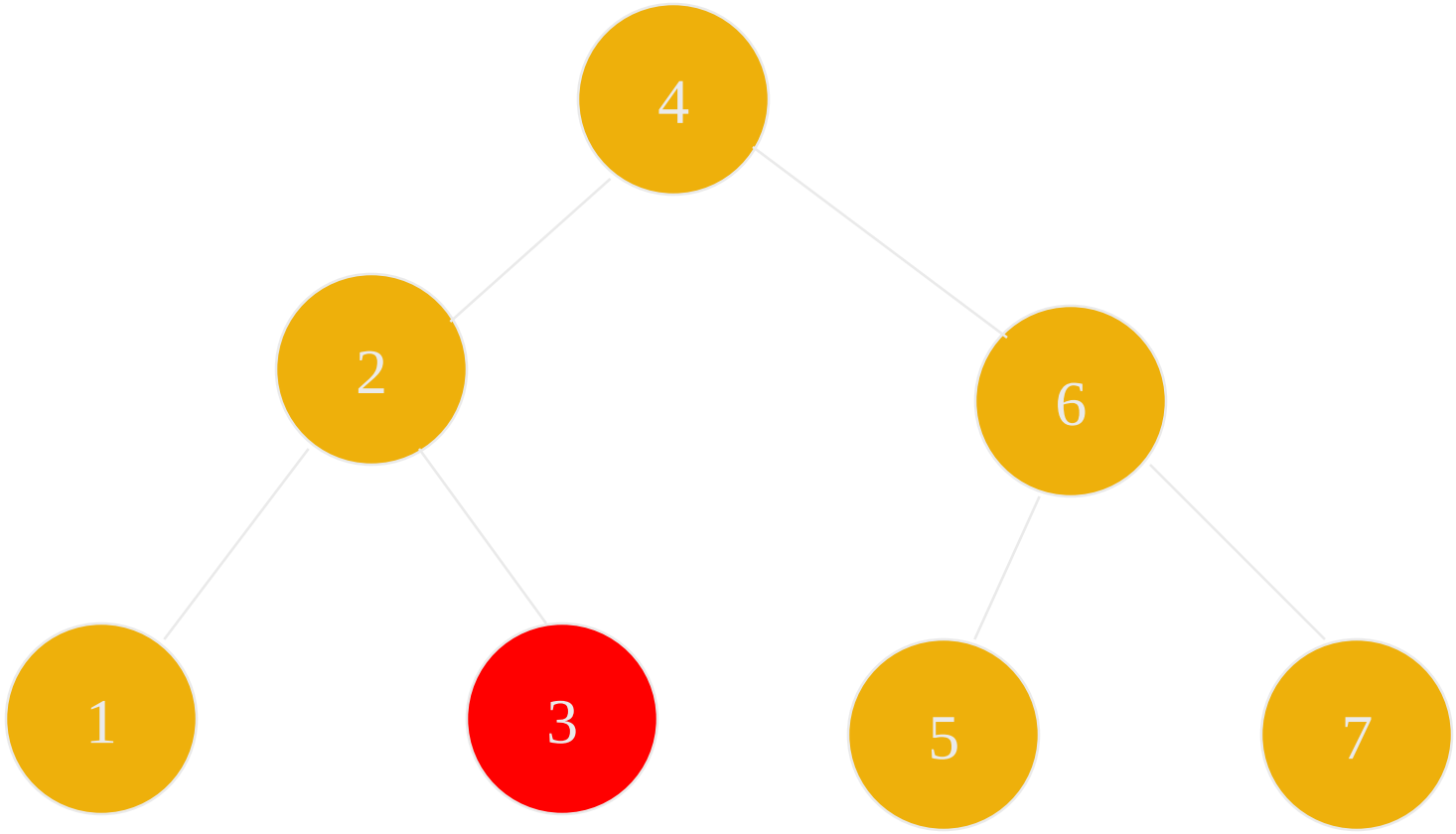


## Right Rotation:

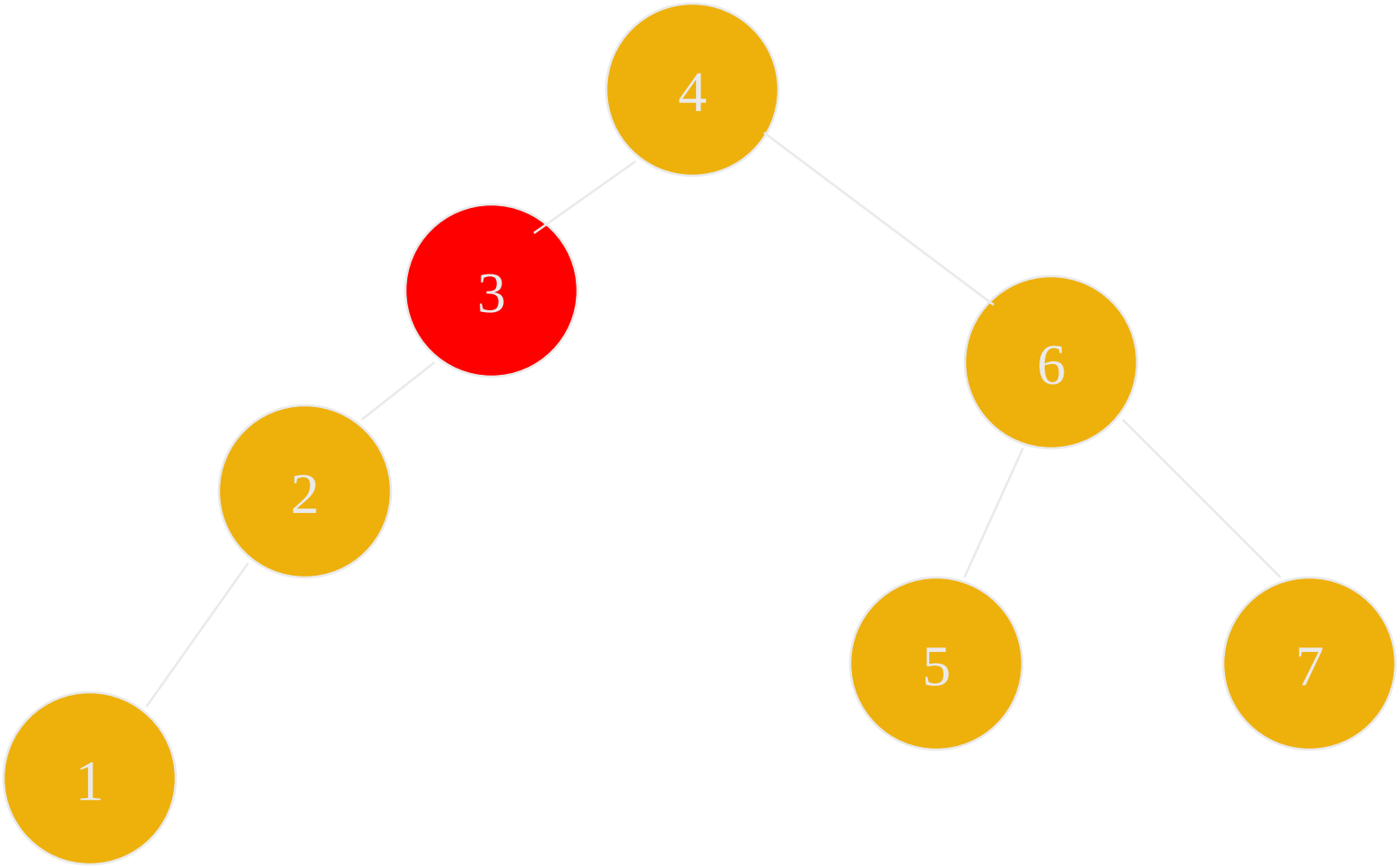


# Move-to-root operation (x=3)

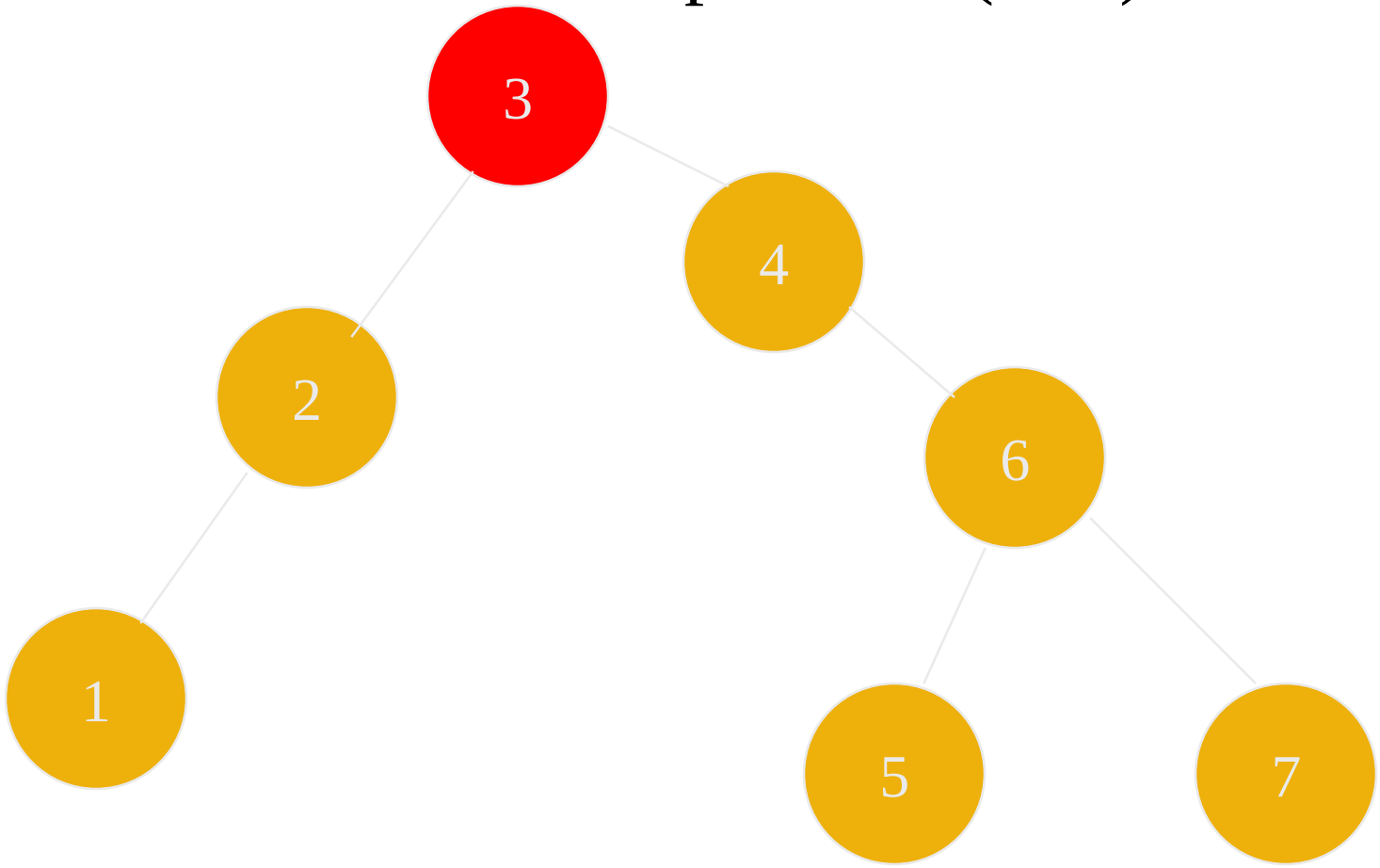
Allen and Munro (1978), Bitner(1979), Kingston



# Move-to-root operation (x=3)



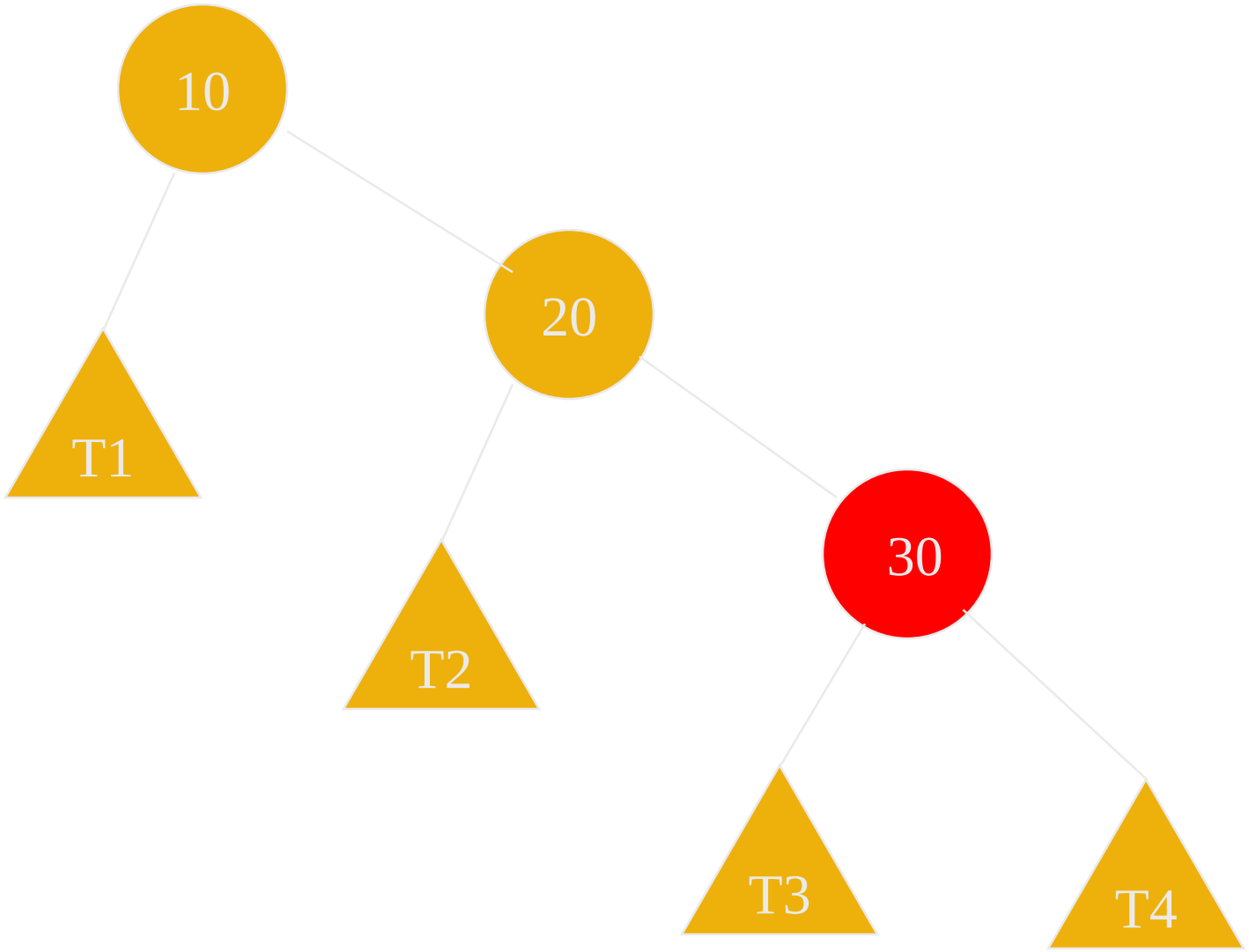
# Move-to-root operation (x=3)



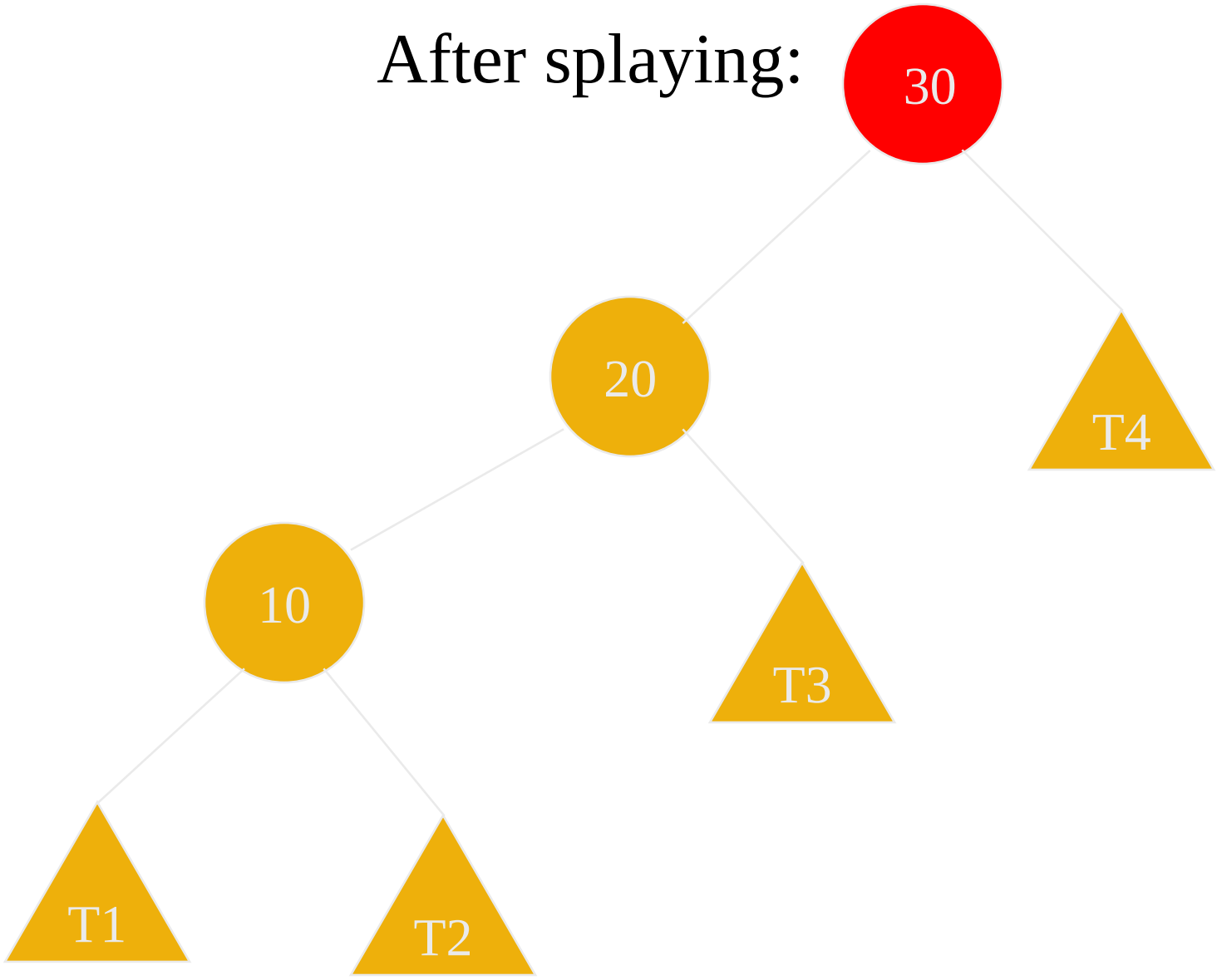
# Zig-zig splaying

- The node x and its parent y are both left children or both right children.
- We replace z by x, making y a child of x and z a child of y, while maintaining the inorder relationships of the nodes in tree T.
- Example: x is 30, y is 20, z is 10

# Before splaying:



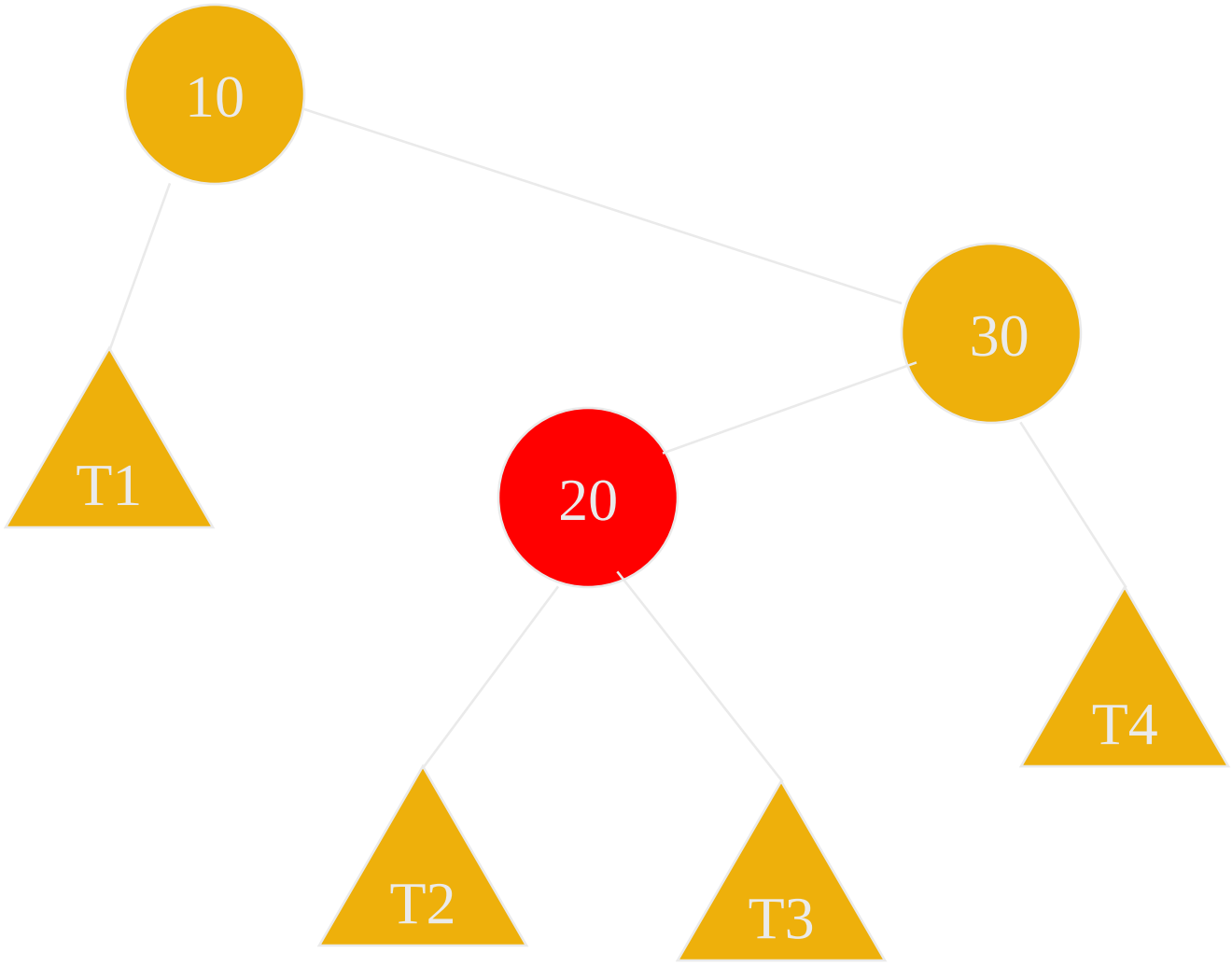
After splaying:



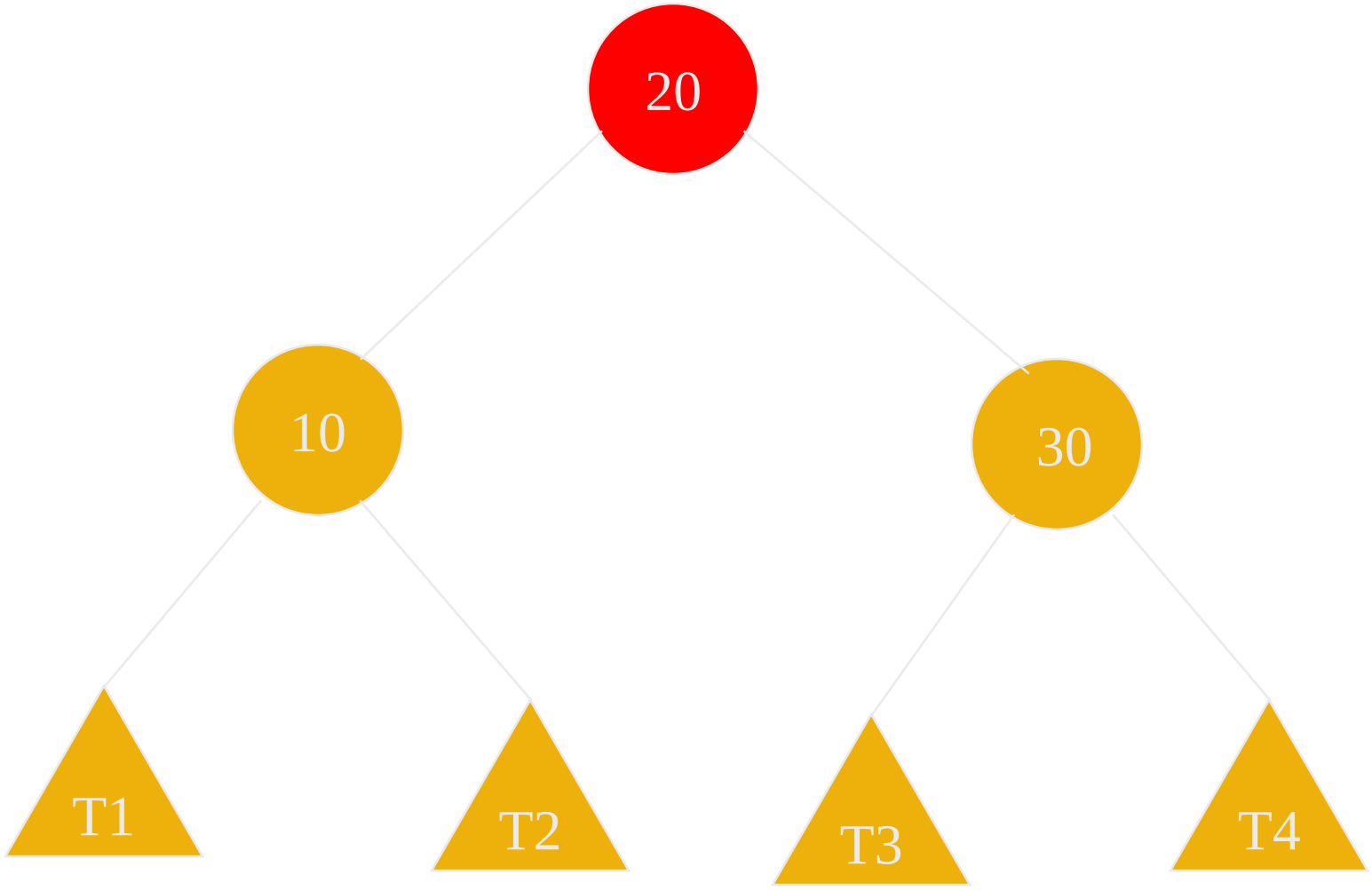
# Zig-zag splaying

- One of x and y is a left child and the other is a right child.
- We replace z by x and make x have nodes y and z as its children, while maintaining the inorder relationships of the nodes in tree T.
- Example: z is 10, y is 30, x is 20

# Before splaying:



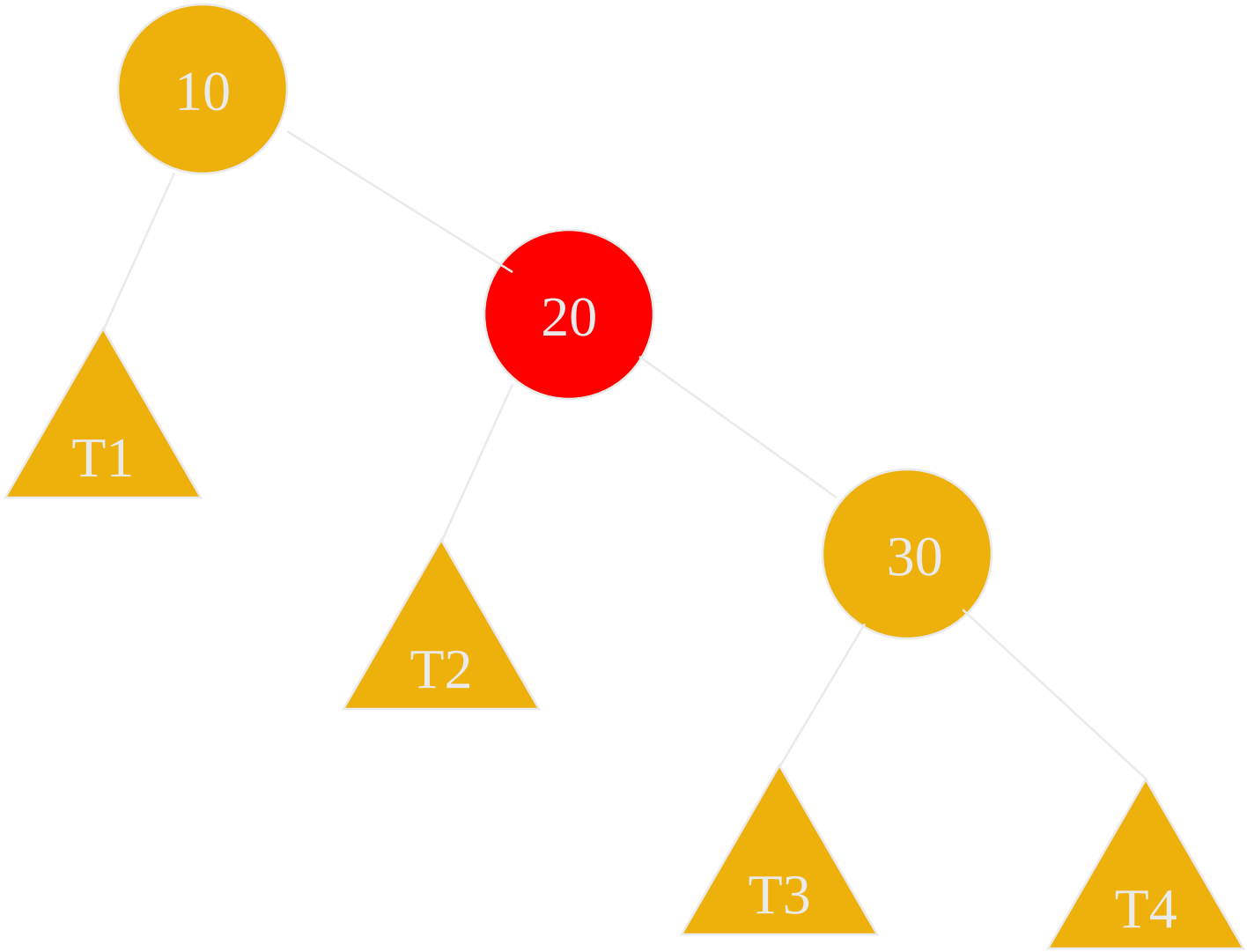
# After splaying:



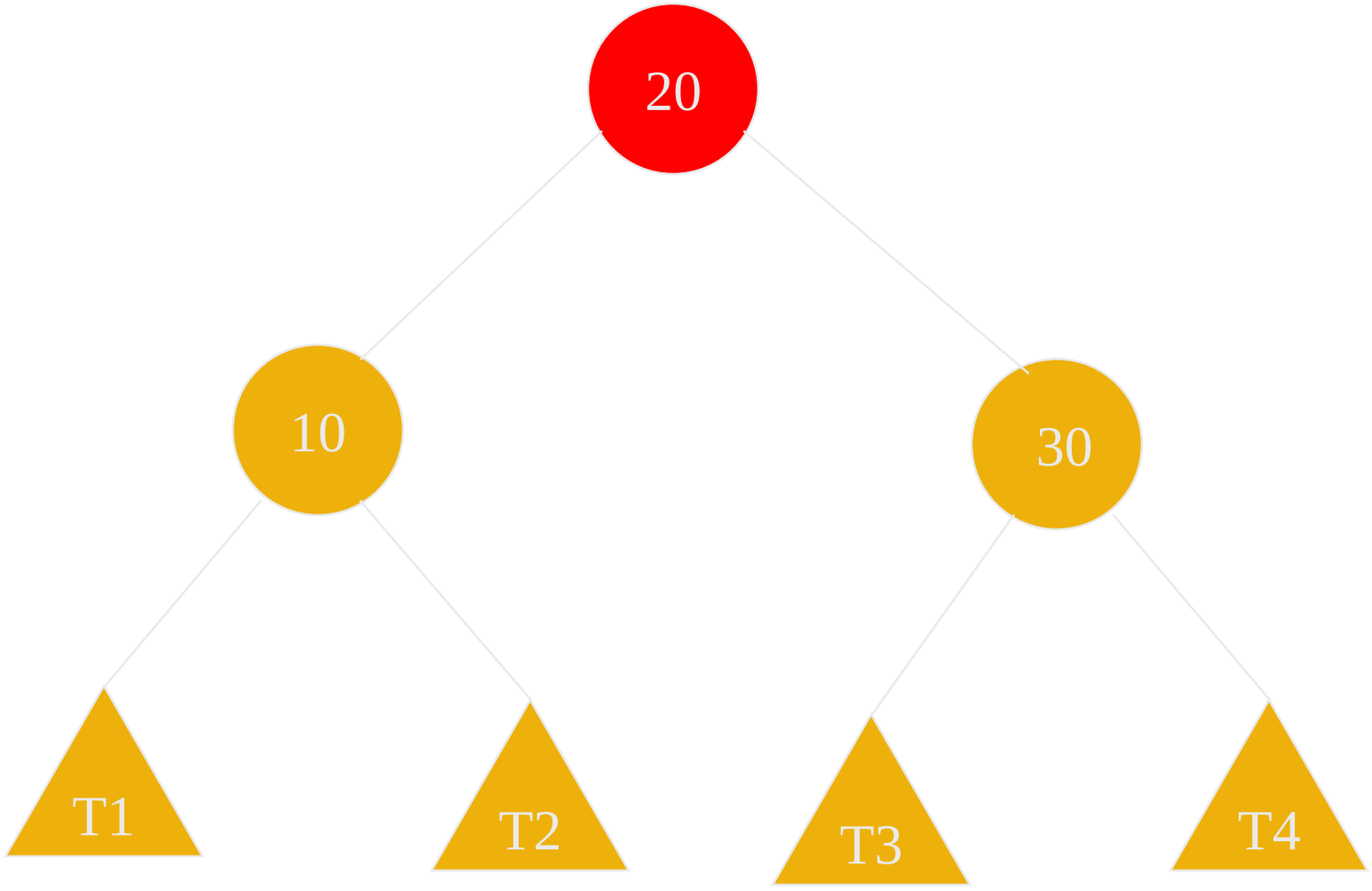
# Zig splaying

- X doesn’t have a grand parent(or the grandparent is not considered)
- We rotate x over y, making x’s children be the node y and one of x’s former children u, so as to maintain the relative inorder relationships of the nodes in tree T.
- Example: x is 20, y is 10, u is 30

# Before splaying:



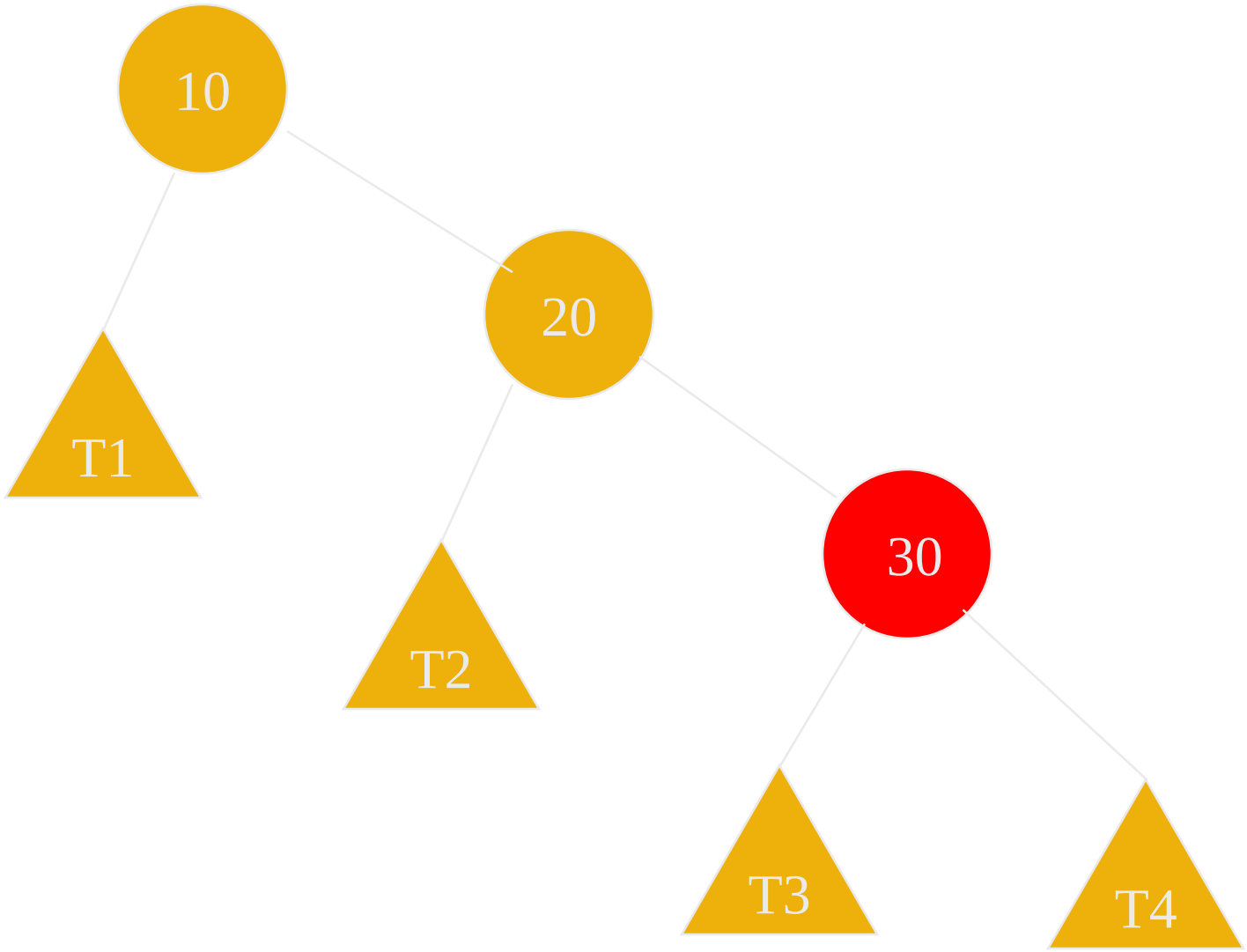
# After splaying:



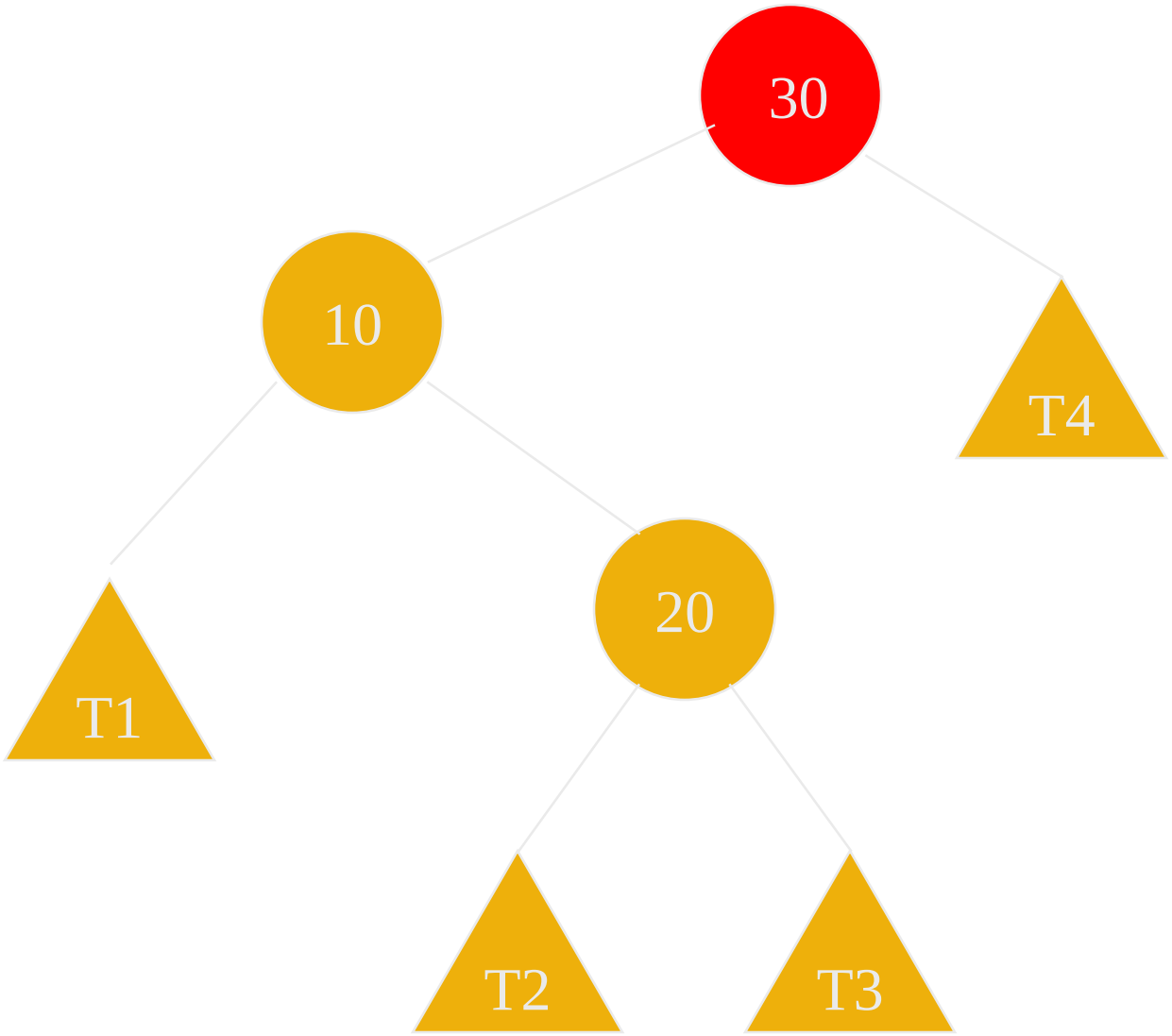
# Move-to-root vs. splaying

- Move-to-root should improve the performance of the BST when there is locality of references in the operation sequence, but it is not ideal. The example we use before, the result is not balanced even the original tree was.
- Splaying also moves the subtrees of node x(which is being splayed) up 1 level, but move-to-root usually leaves one subtree at its original level.
- Example:

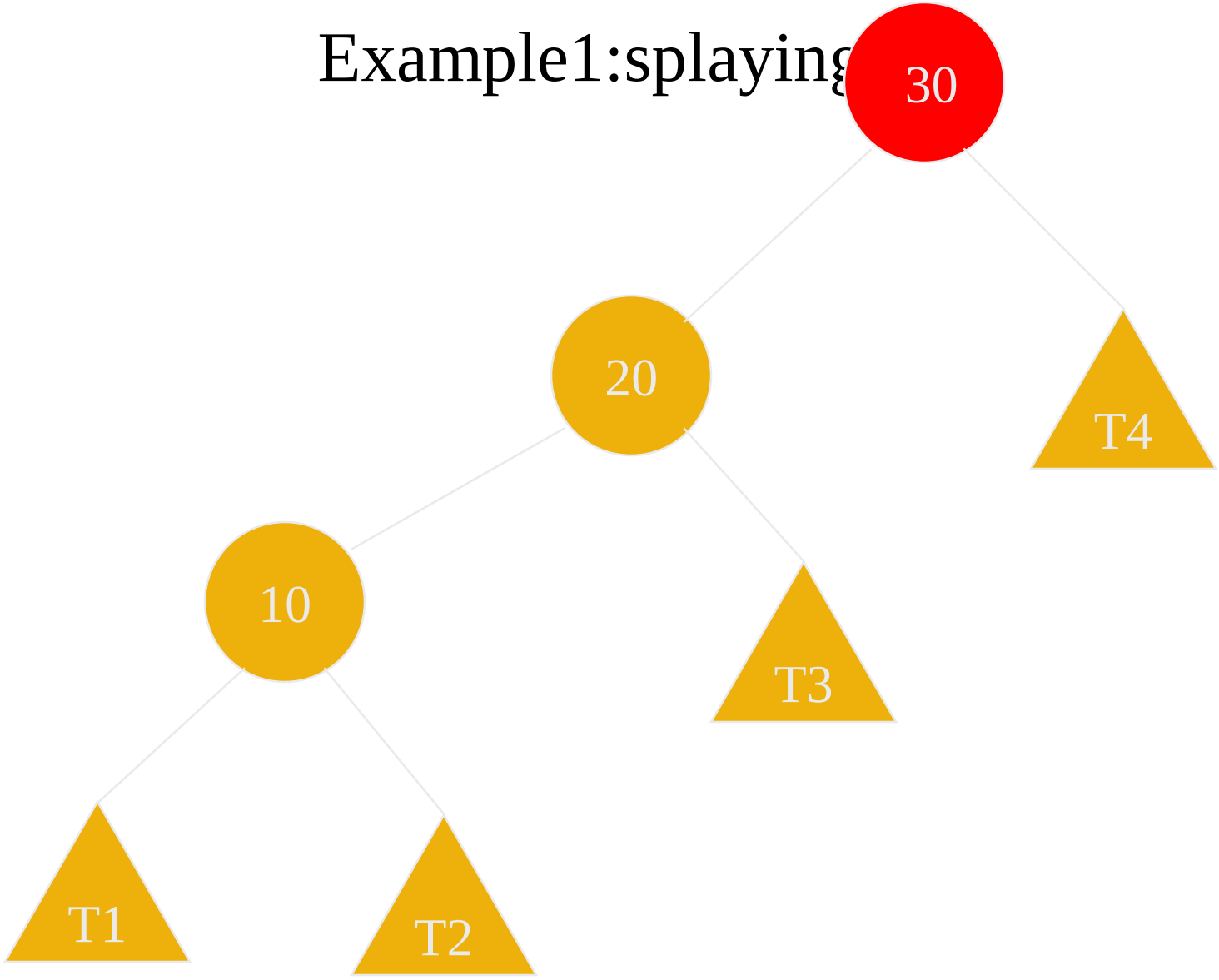
# Example1:original tree



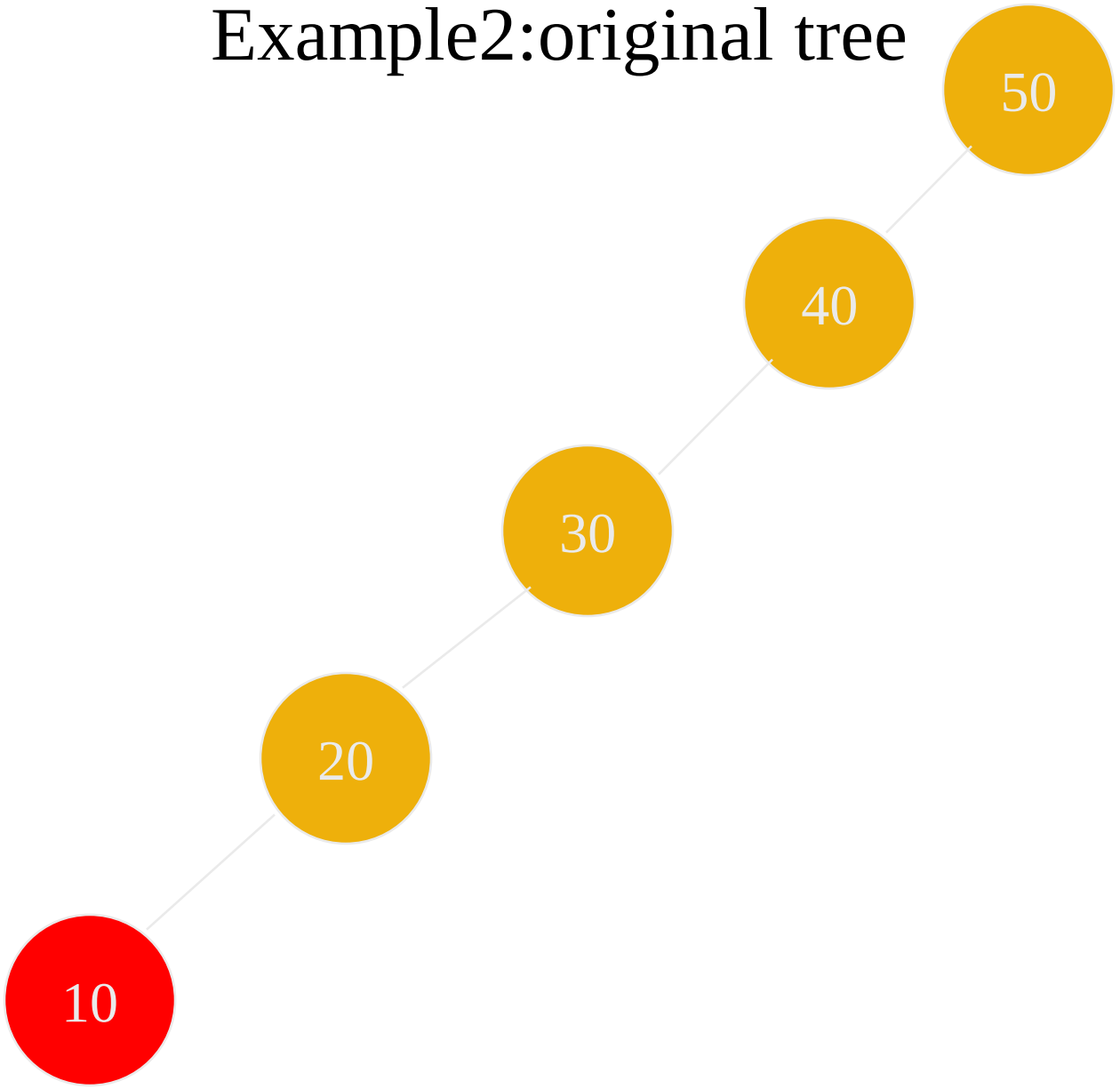
# Example1:move-to-root



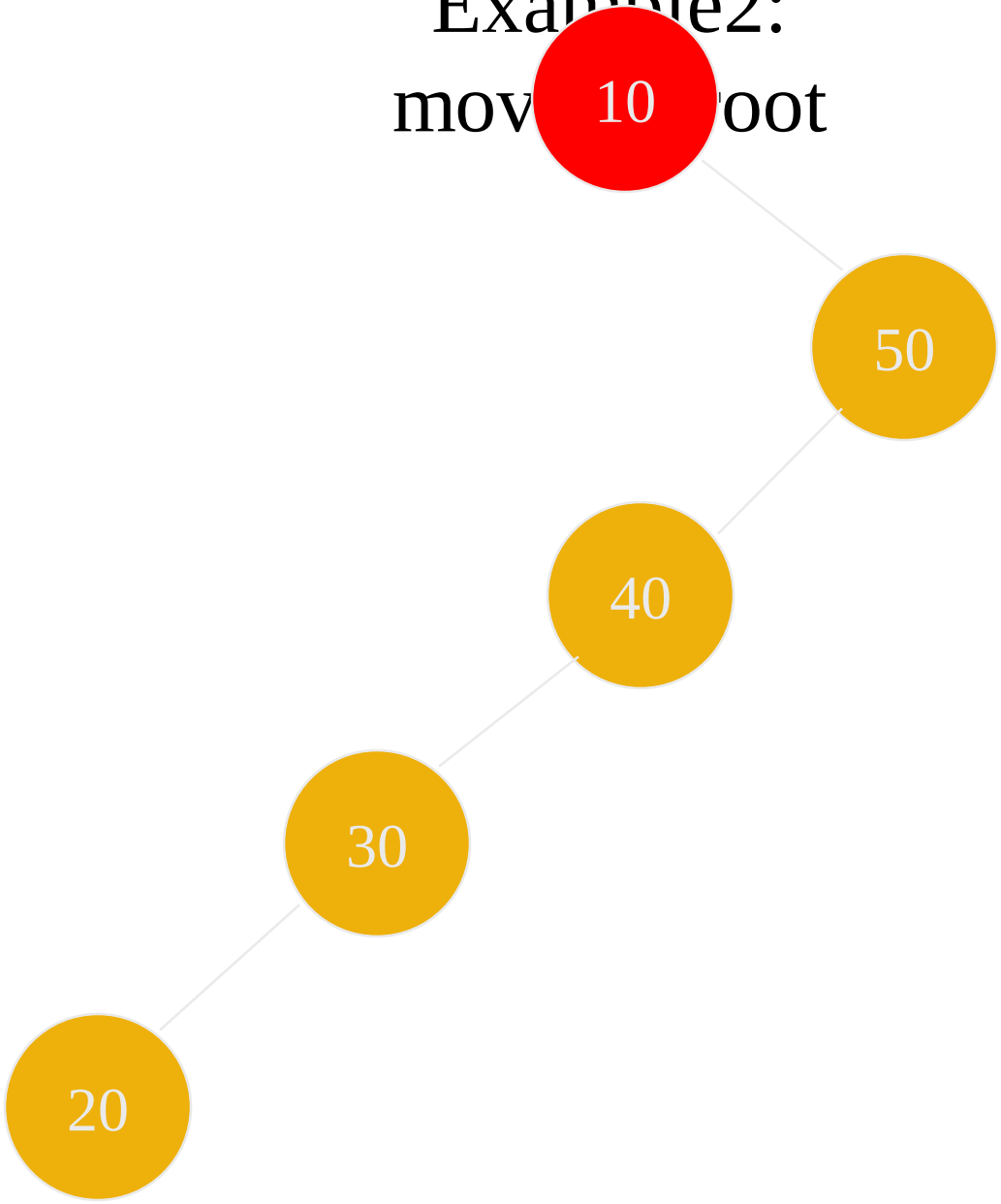
# Example1:splaying



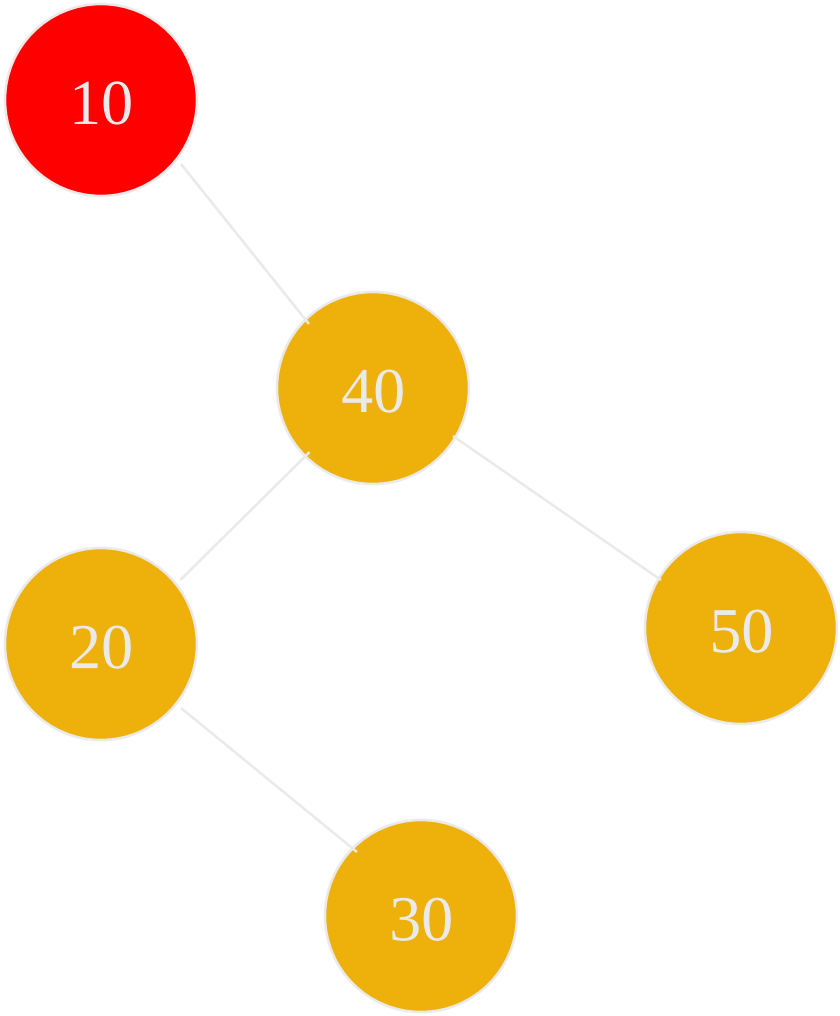
# Example2:original tree



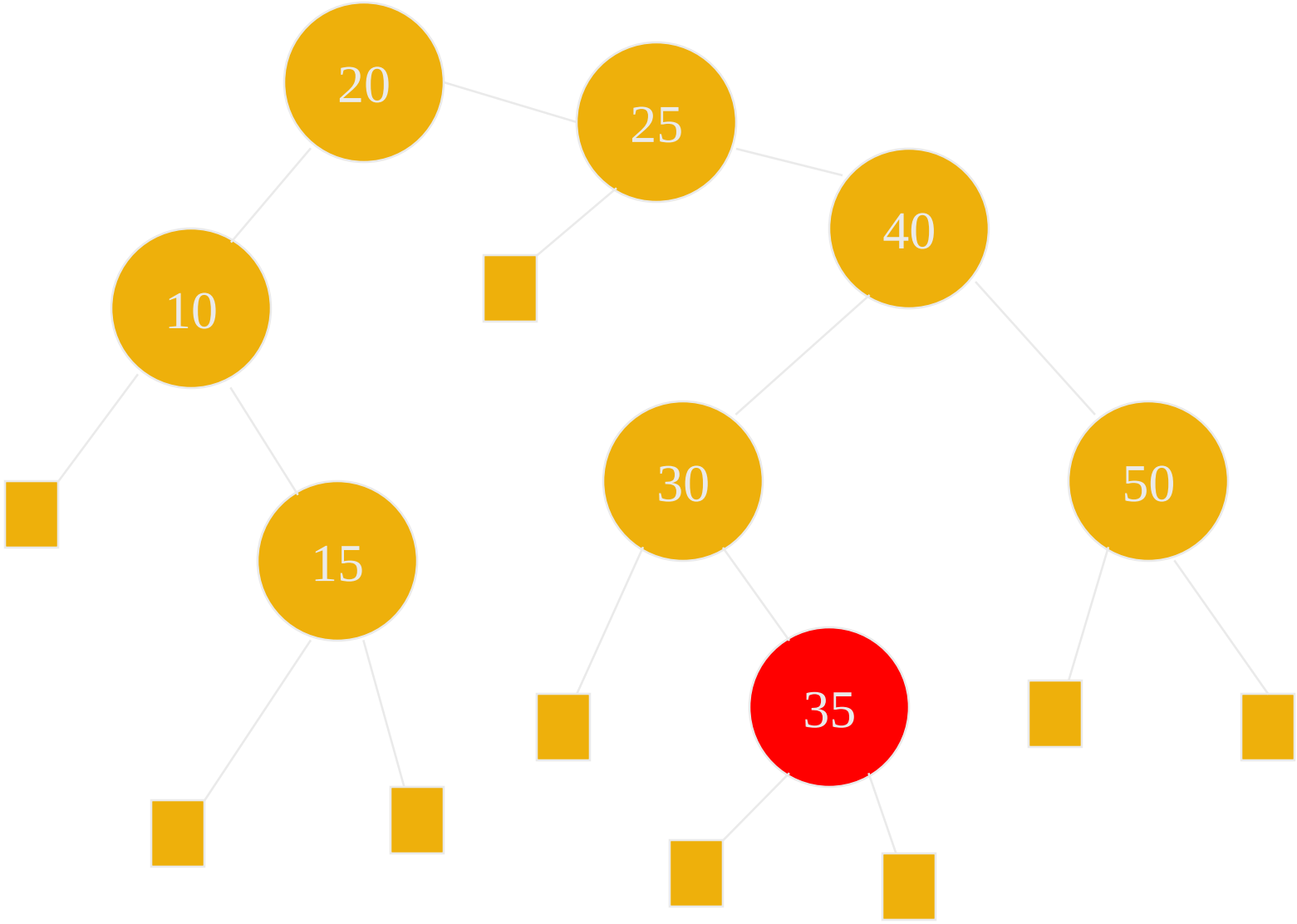
# Example2:
# mov[illegible]oot



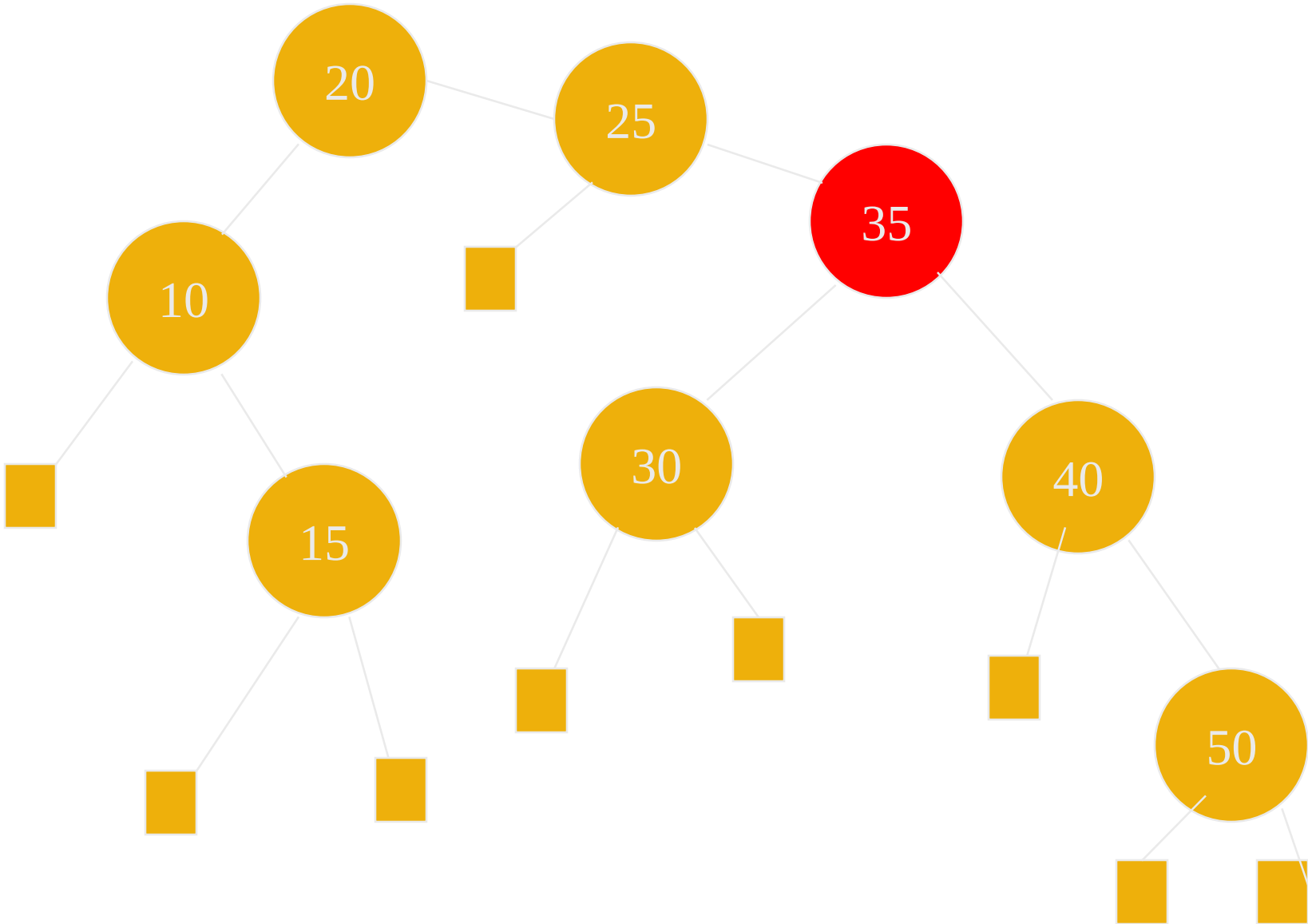
# Example2:splaying



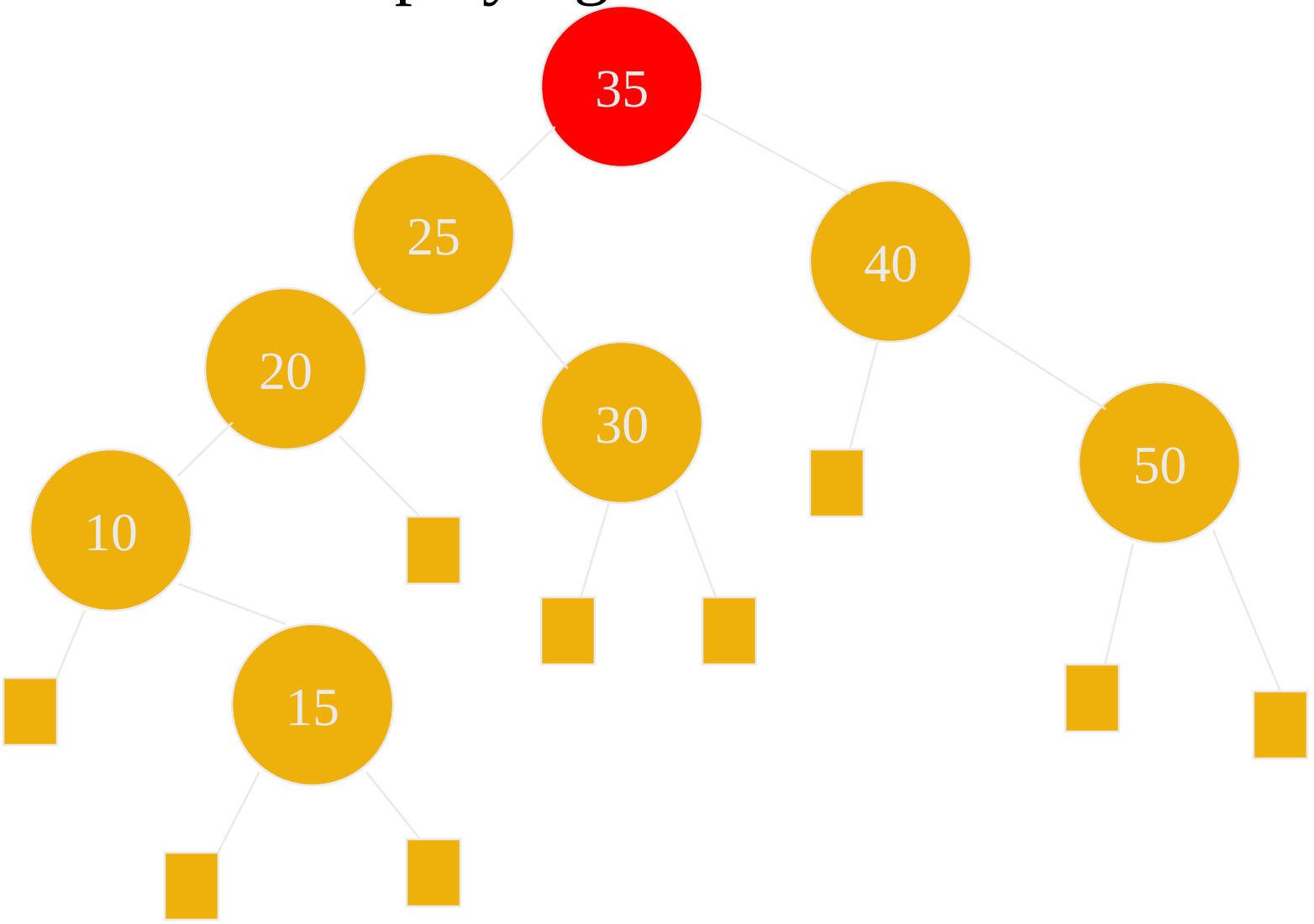
# Splaying a node: original tree



# Splaying a node: cont.



# Splaying a node: cont.



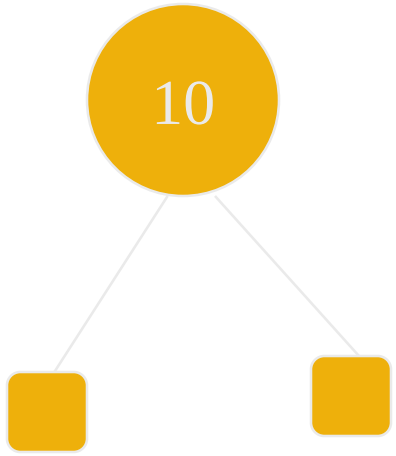
# Rules of splaying: Search

- When search for key I, if I is found at node x, we splay x.
- If not successful, we splay the parent of the external node at which the search terminates unsuccessfully.(null node)
- Example: see above slides, it can be for successfully searched I=35 or unsuccessfully searched say I=38.

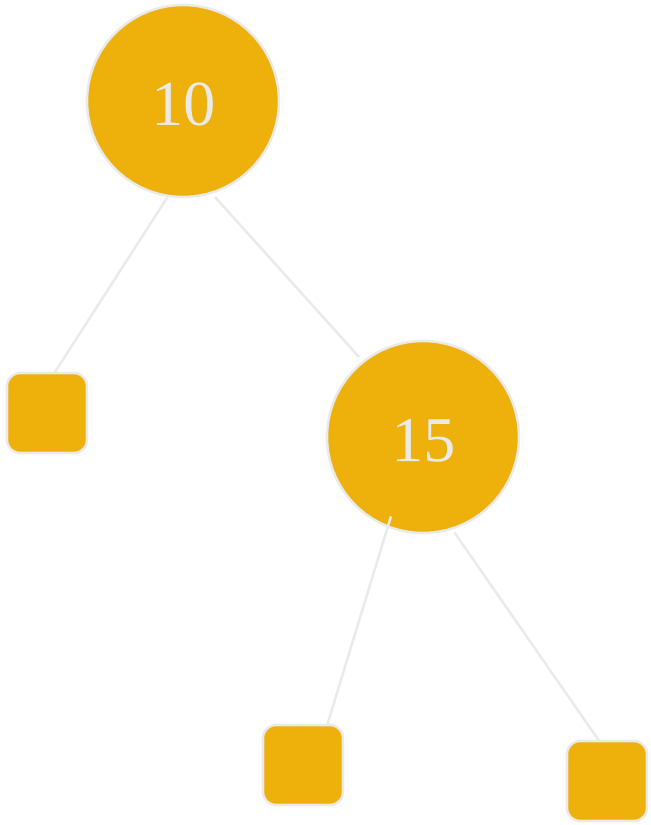
# Rules of splaying: insertion

- When inserting a key I, we splay the newly created internal node where I was inserted.
- Example:

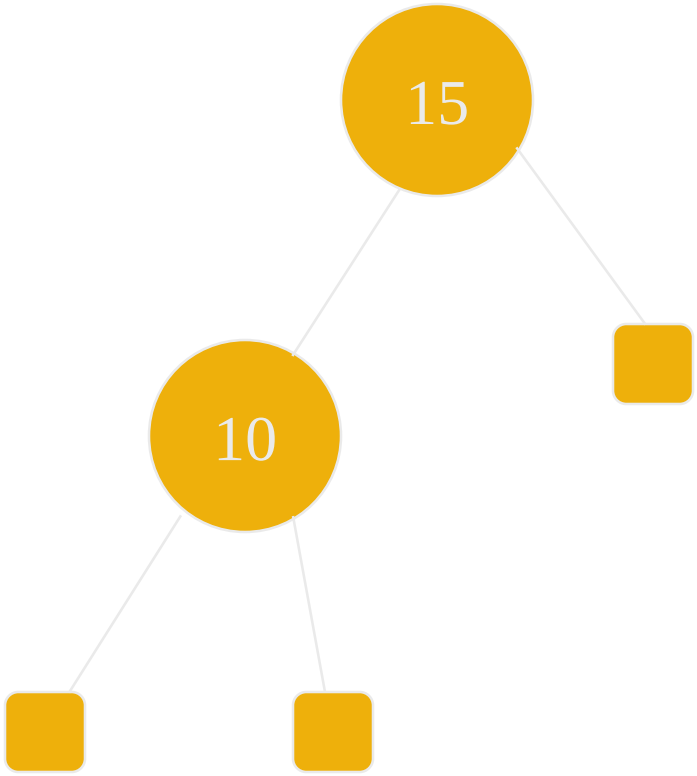
# Original tree



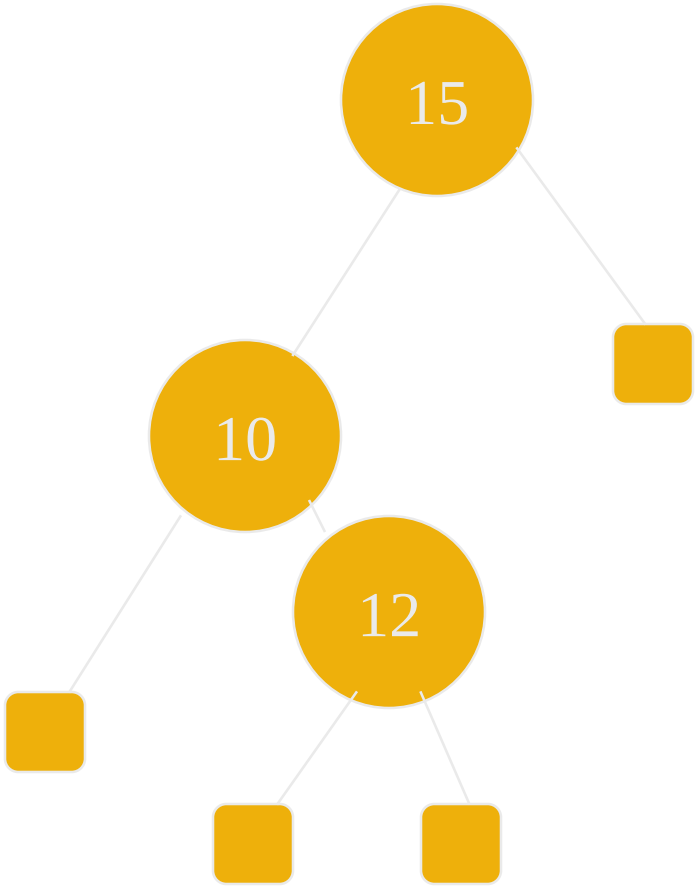
# After insert key 15



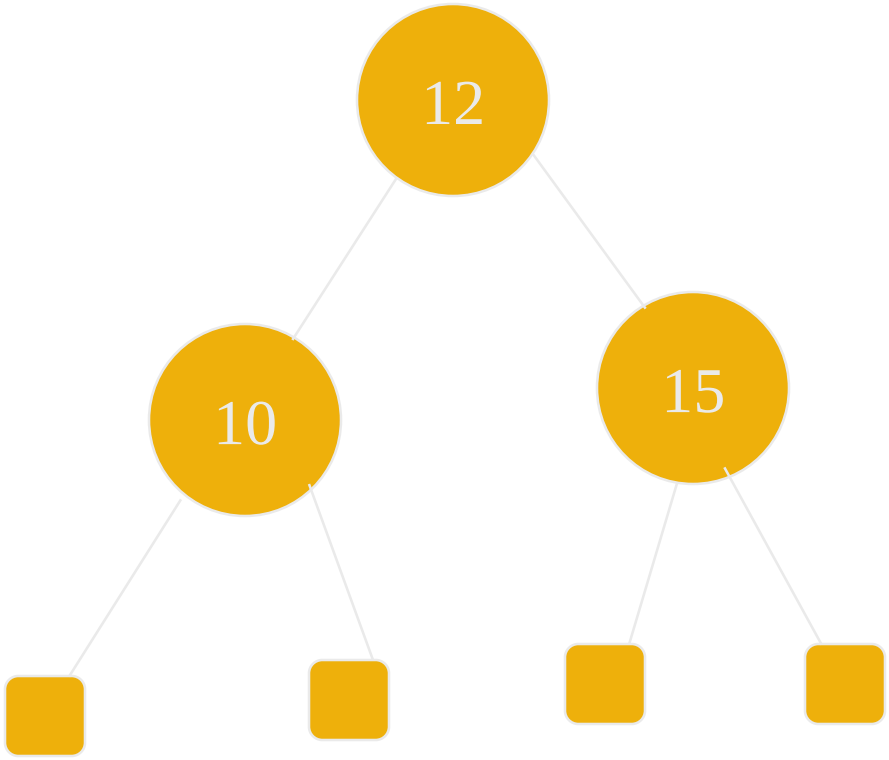
# After splaying



# After insert key 12



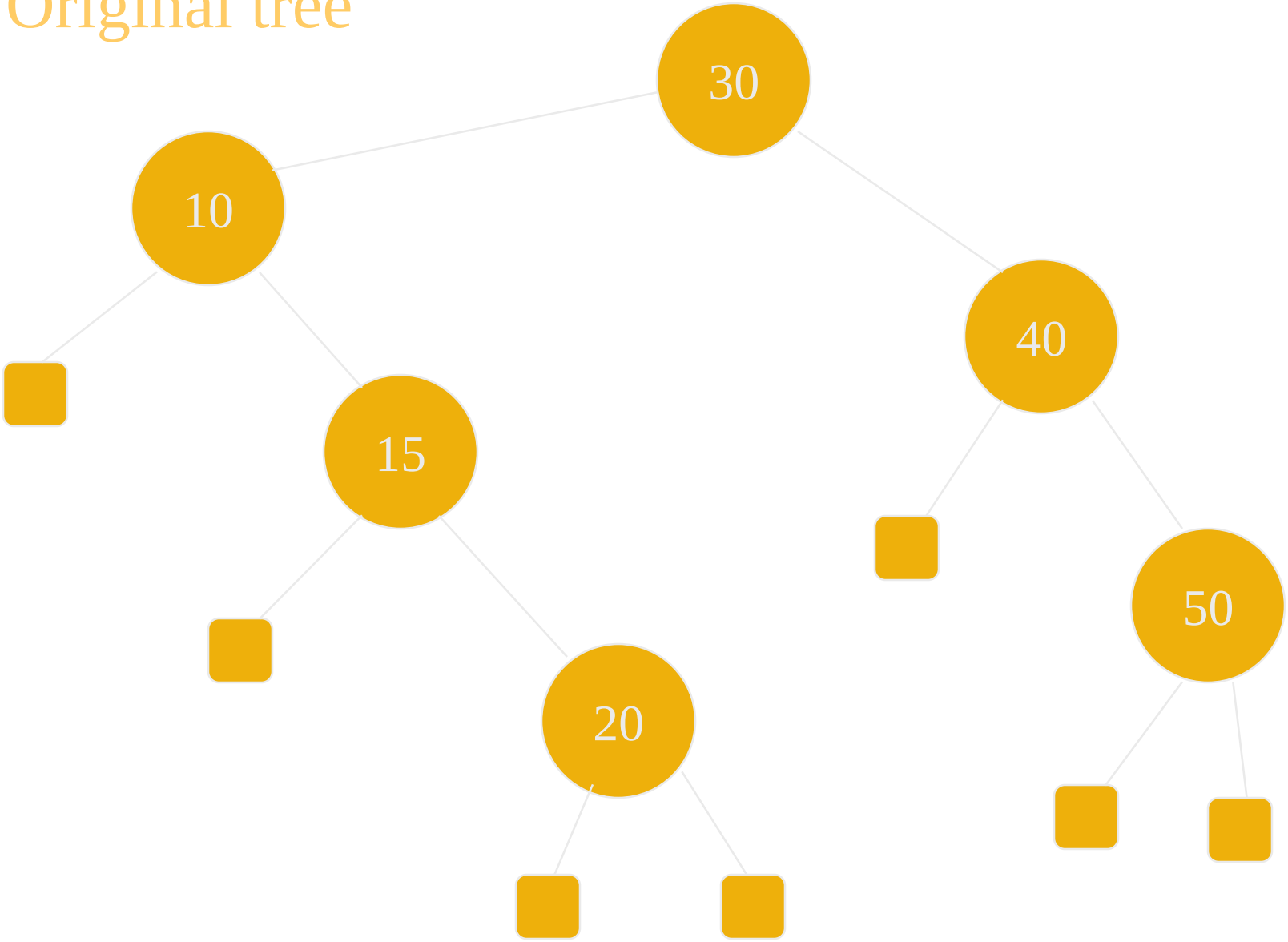
# After splaying



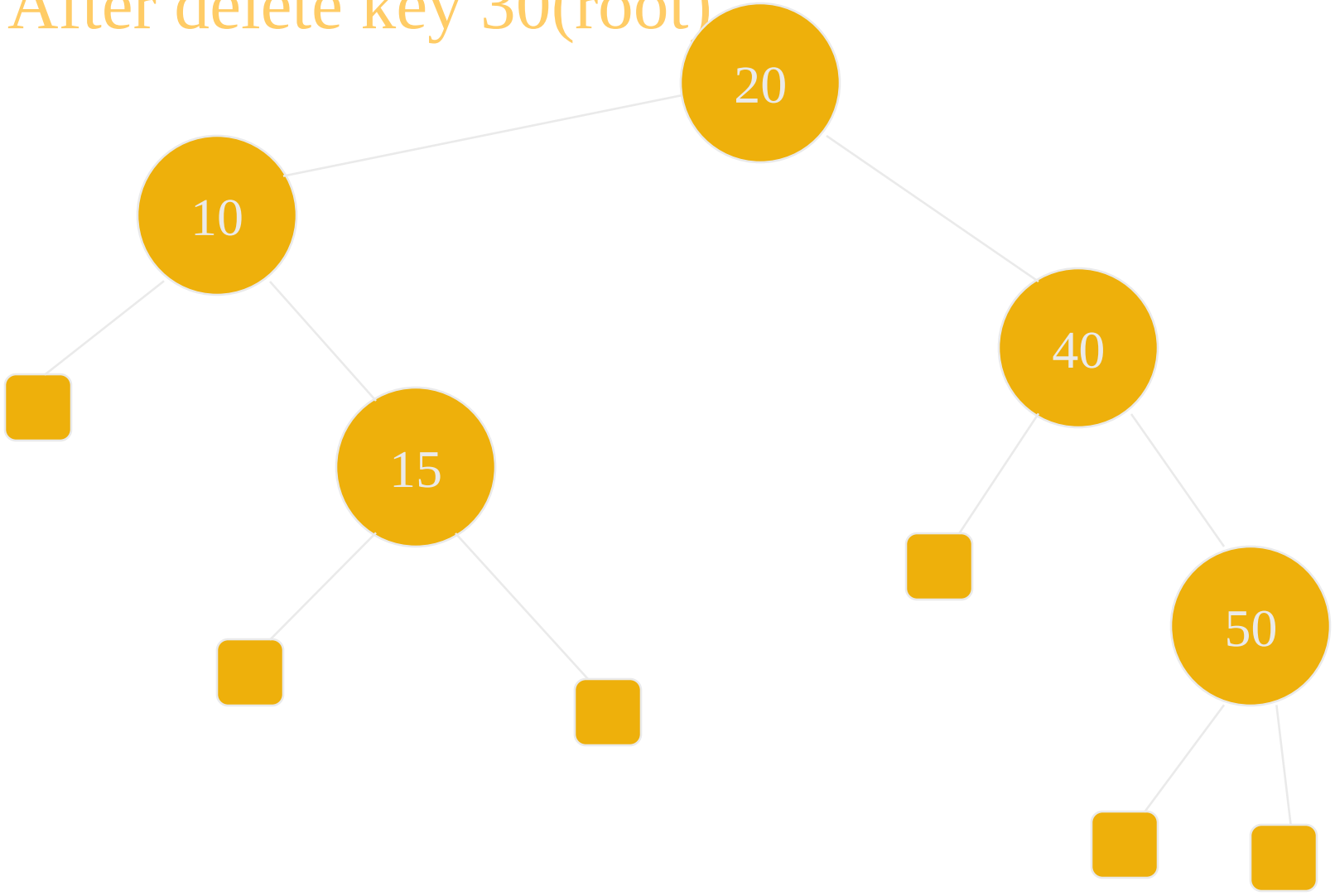
# Rules of splaying: deletion

- When deleting a key I, we splay the parent of the node x that gets removed. x is either the node storing I or one of its descendents(root, etc.).
- If deleting from root, we move the key of the right-most node in the left subtree of root and delete that node and splay the parent. Etc.
- Example:

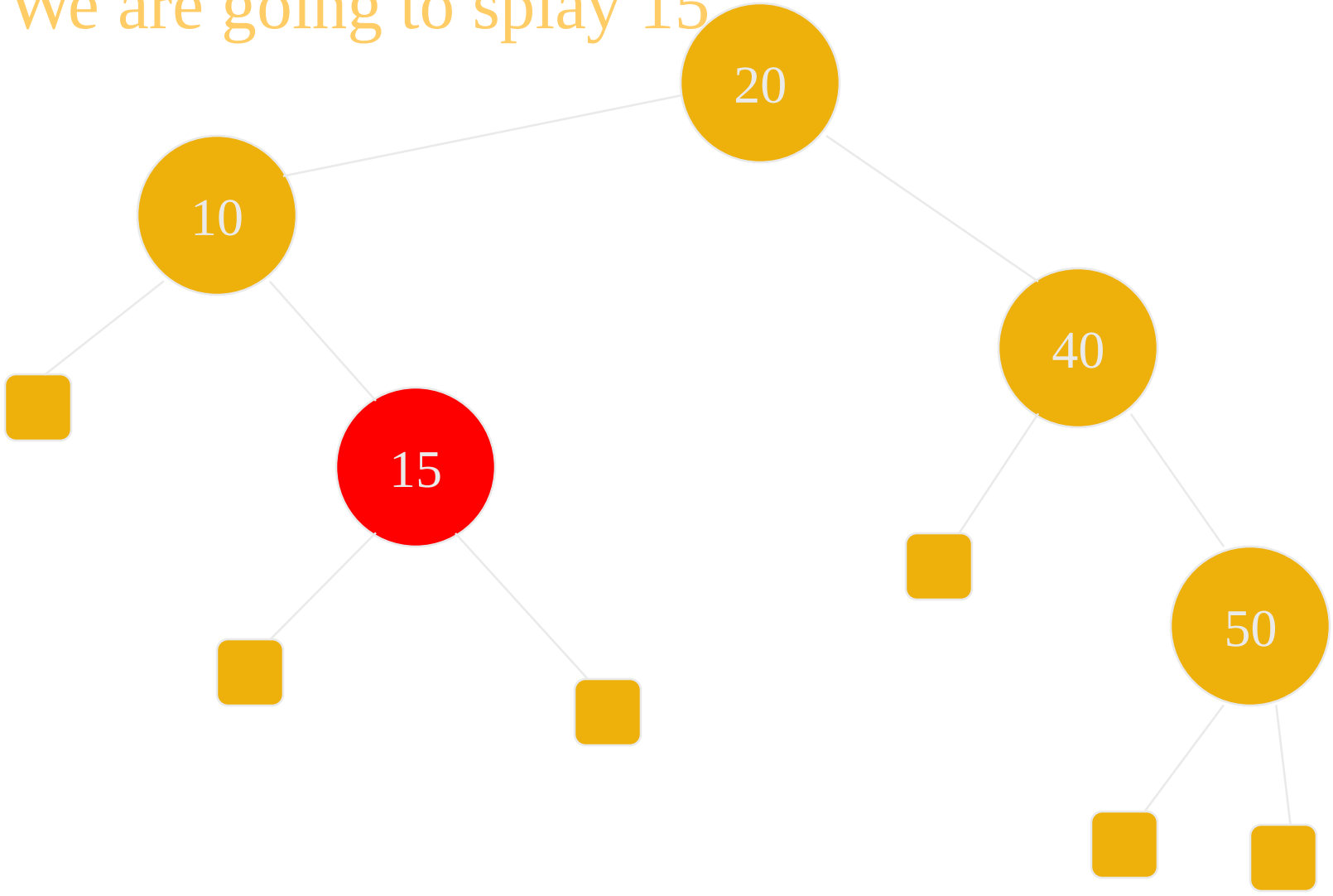
# Original tree

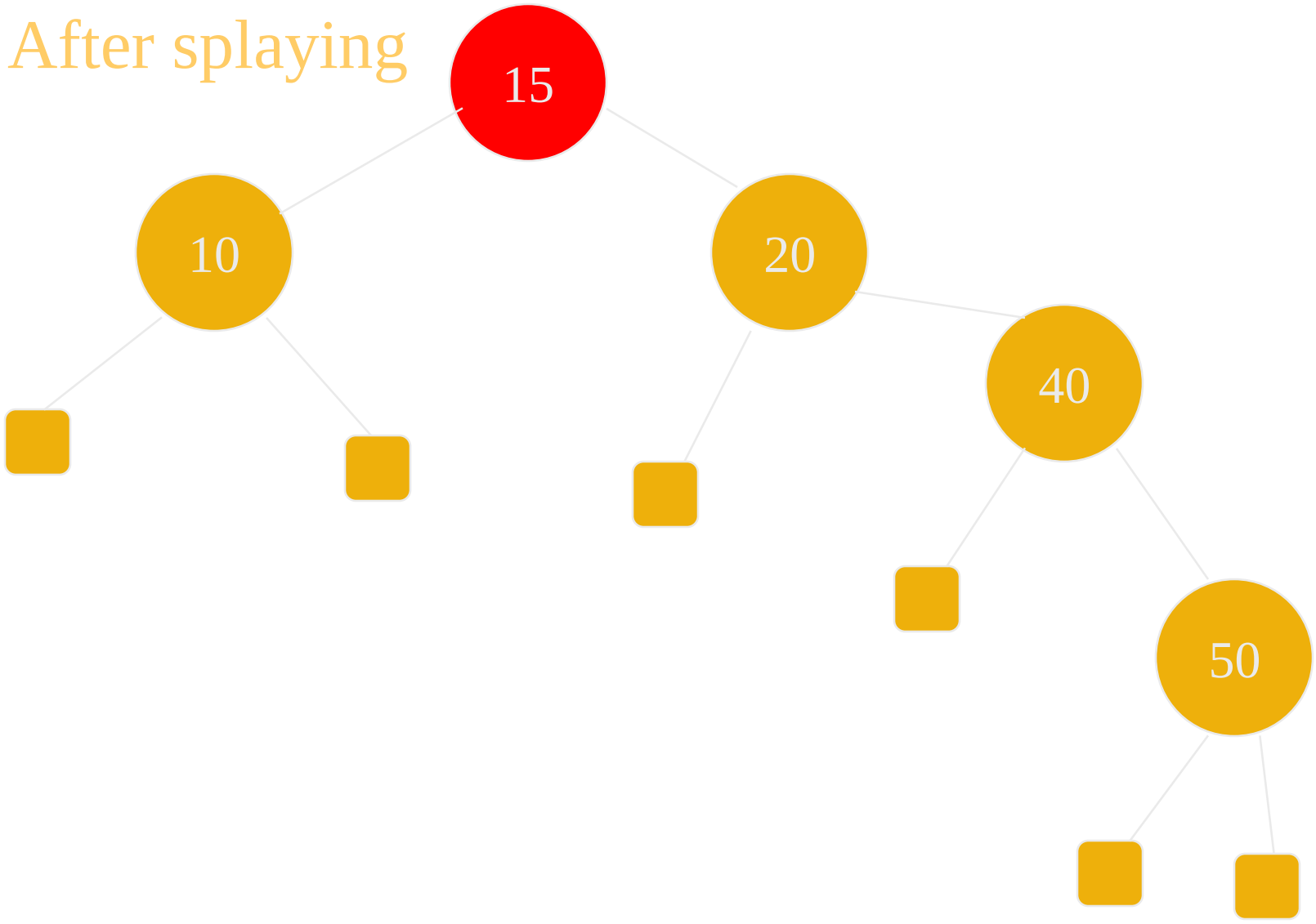


# After delete key 30(root)



We are going to splay 15




After splaying
15
10
20
40
50

# Complexity

- Worst case: O(n), all nodes are on one side of the subtree. (In fact, it is $\Omega(n)$.)
- Amortized Analysis:

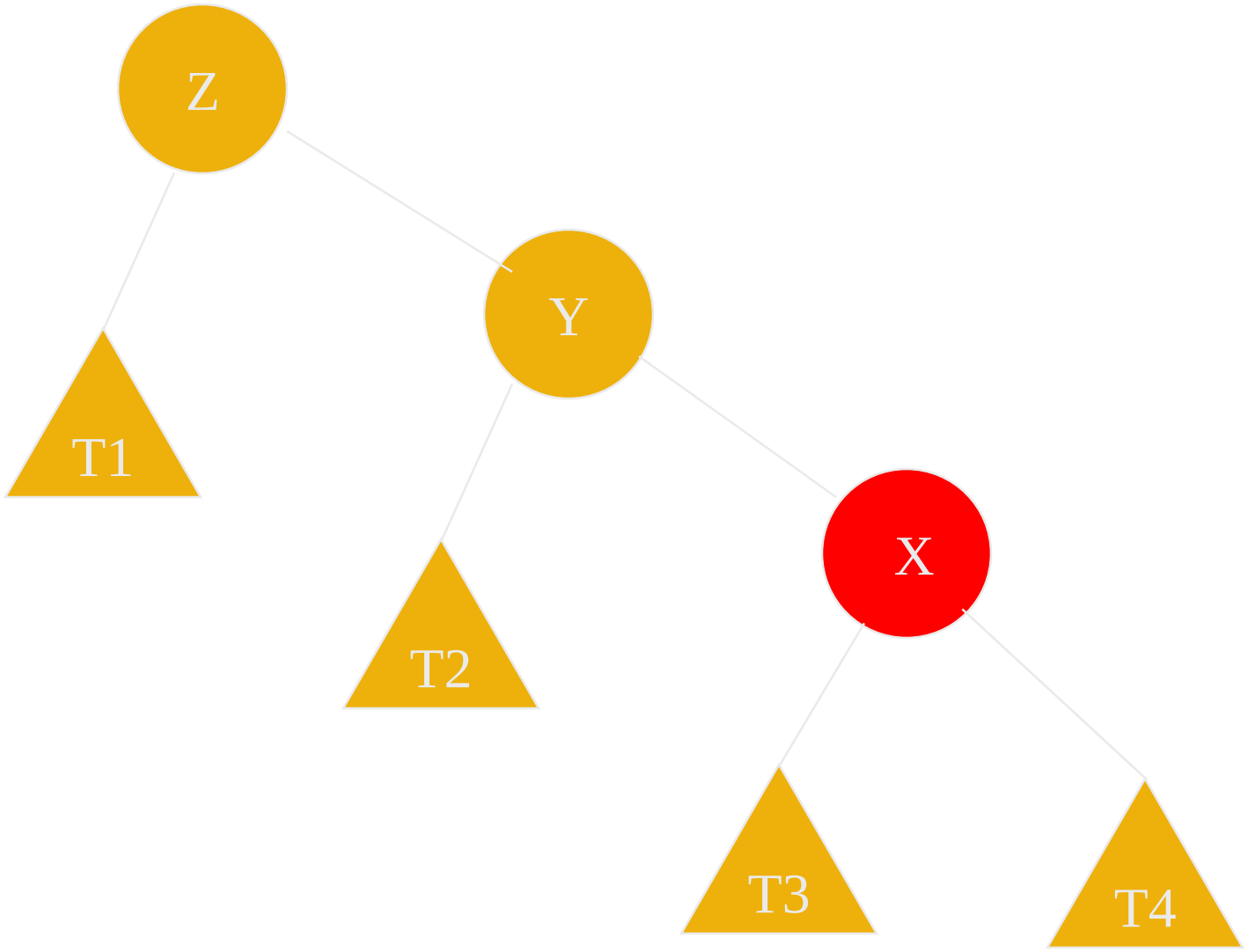
  We will only consider the splaying time, since the time for perform search, insertion or deletion is proportional to the time for the splaying they are associated with.

# Amortized analysis

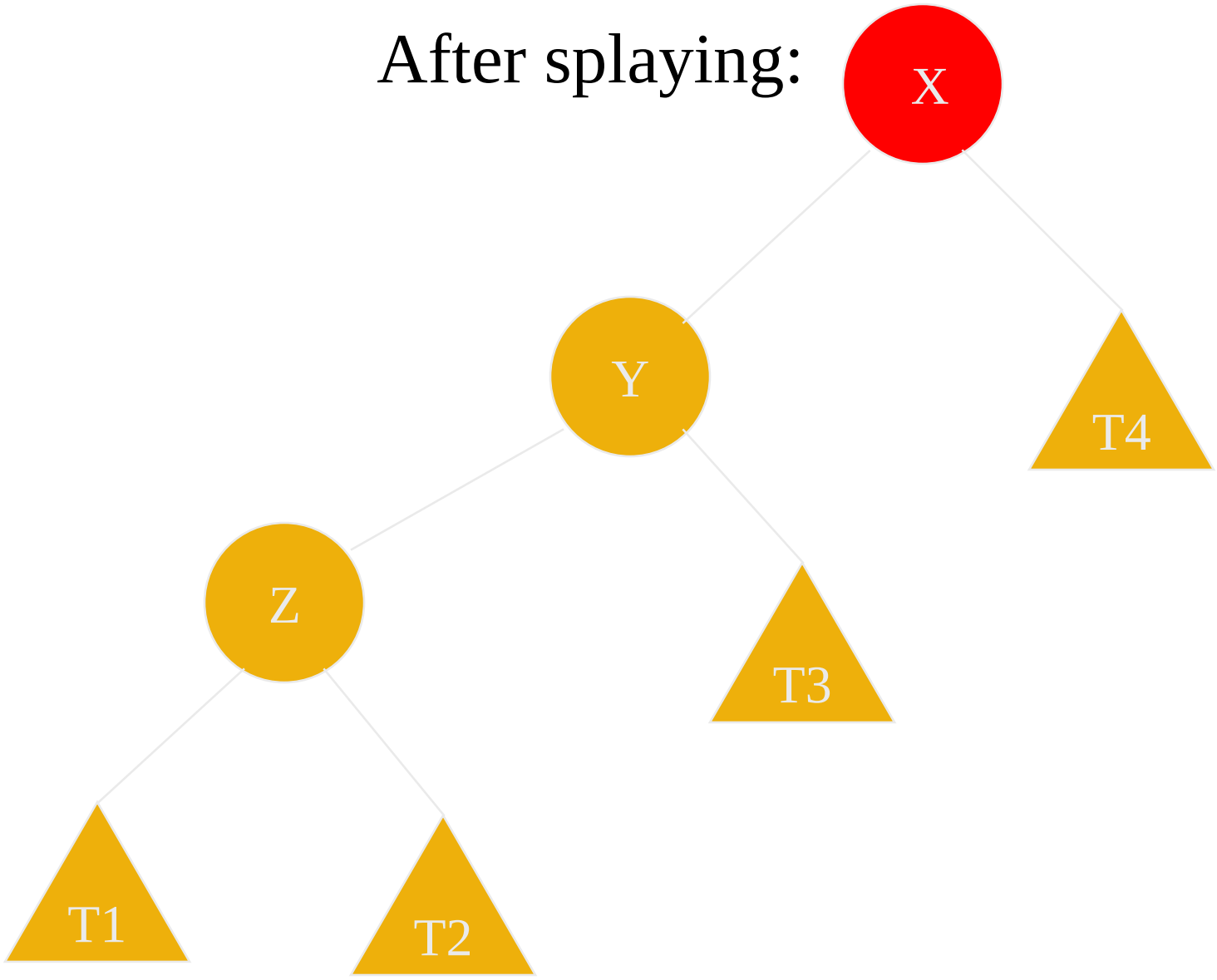
- Let n(v) = the number of nodes in the subtree rooted at v
- Let r(v) = log(n(v)), rank.
- Let r'(v) be the rank of node v after splaying.
- If a>0, b>0, and c>a+b, then

  log a + log b <= 2 log c –2
- *Search the key takes d time, d = depth of the node before splaying.

# Before splaying:



After splaying:



# Cont.

- Zig-zig:

  variation of r(T) caused by a single splaying substep is:

  r’(x)+r’(y)+r’(z)-r(x)-r(y)-r(z)

  =r’(y)+r’(z)-r(x)-r(y)  (r’(x)=r(z))

  <=r’(x)+r’(z)-2r(x)  (r’(y)<=r’(x) and r(y)>=r(x))

Also n(x)+n’(z)<=n’(x)

We have r(x) + r’(z)<=2r’(x)-2  (see previous slide)

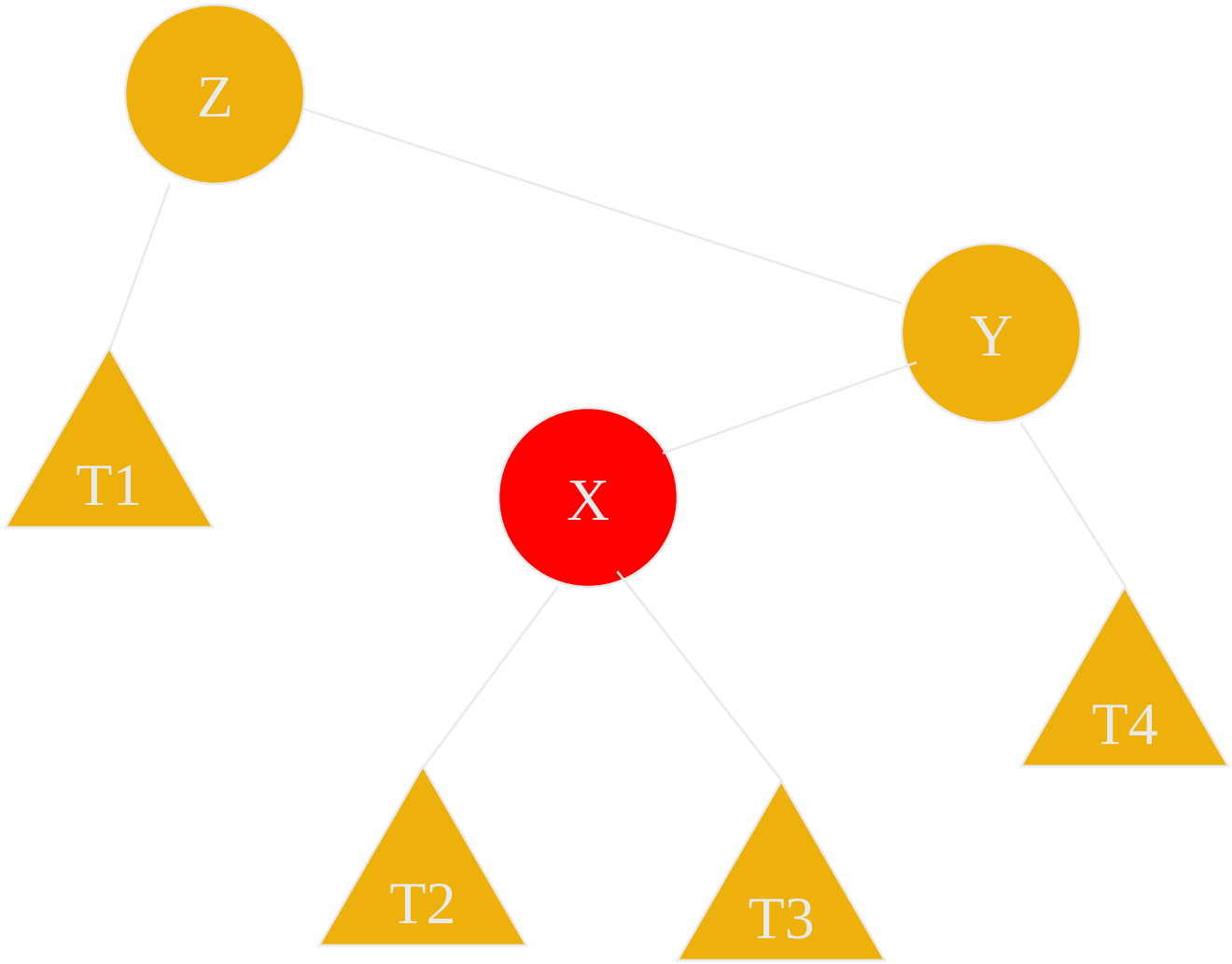
⇒ r’(z)<=2r’(x)-r(x)-2

so we have variation of r(T) by a single splaying step is:

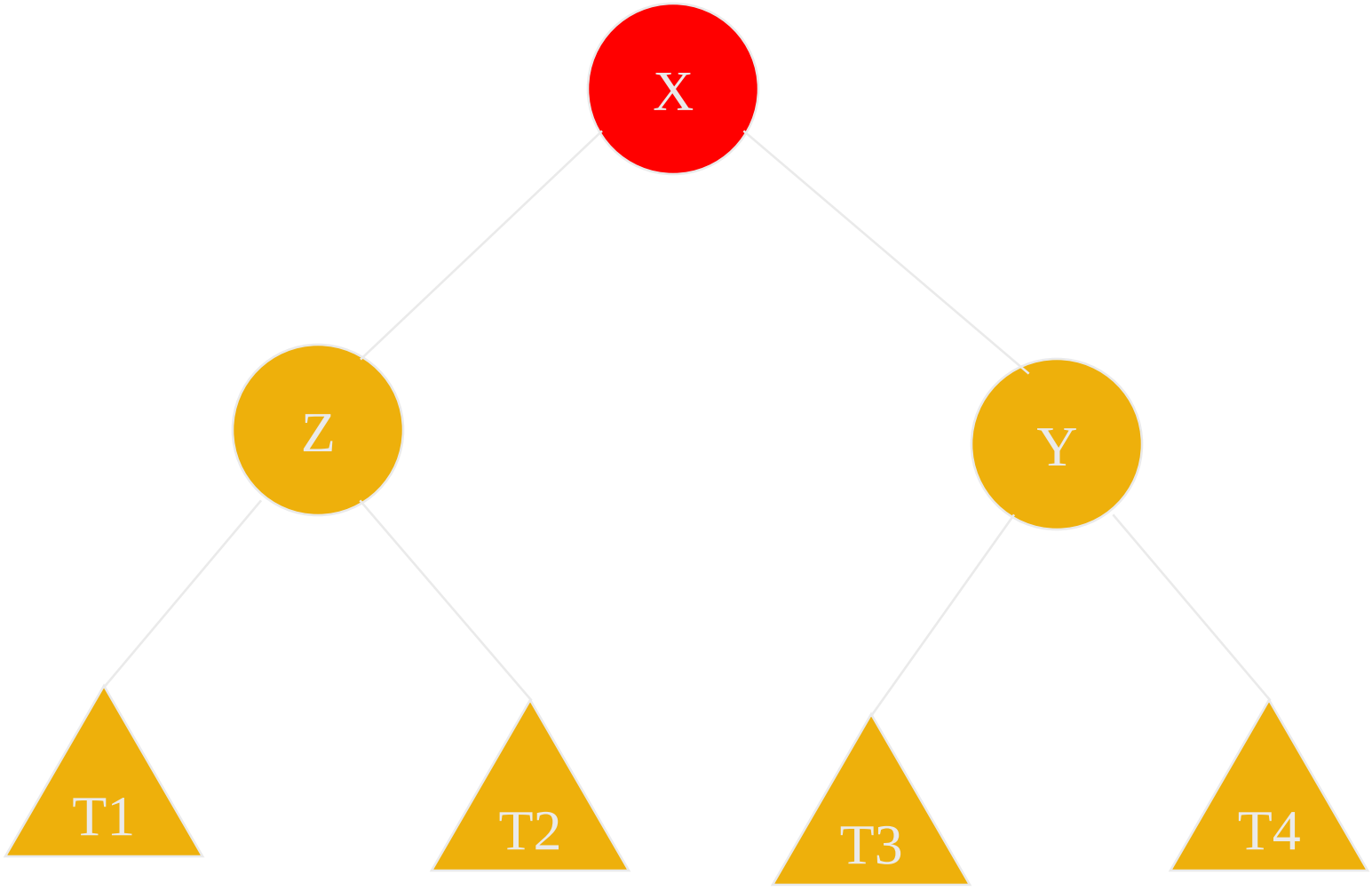
<=3(r’(x)-r(x))-2

Since zig-zig takes 2 rotations, the amortized complexity will be 3(r’(x)-r(x))

# Before splaying:



# After splaying:



# Cont.

- Zig-zag:

  variation of r(T) caused by a single splaying substep is:

  r’(x)+r’(y)+r’(z)-r(x)-r(y)-r(z)

  =r’(y)+r’(z)-r(x)-r(y)  (r’(x)=r(z))

  <=r’(y)+r’(z)-2r(x)  (r(y)>=r(x))

Also n’(y)+n’(z)<=n’(x)

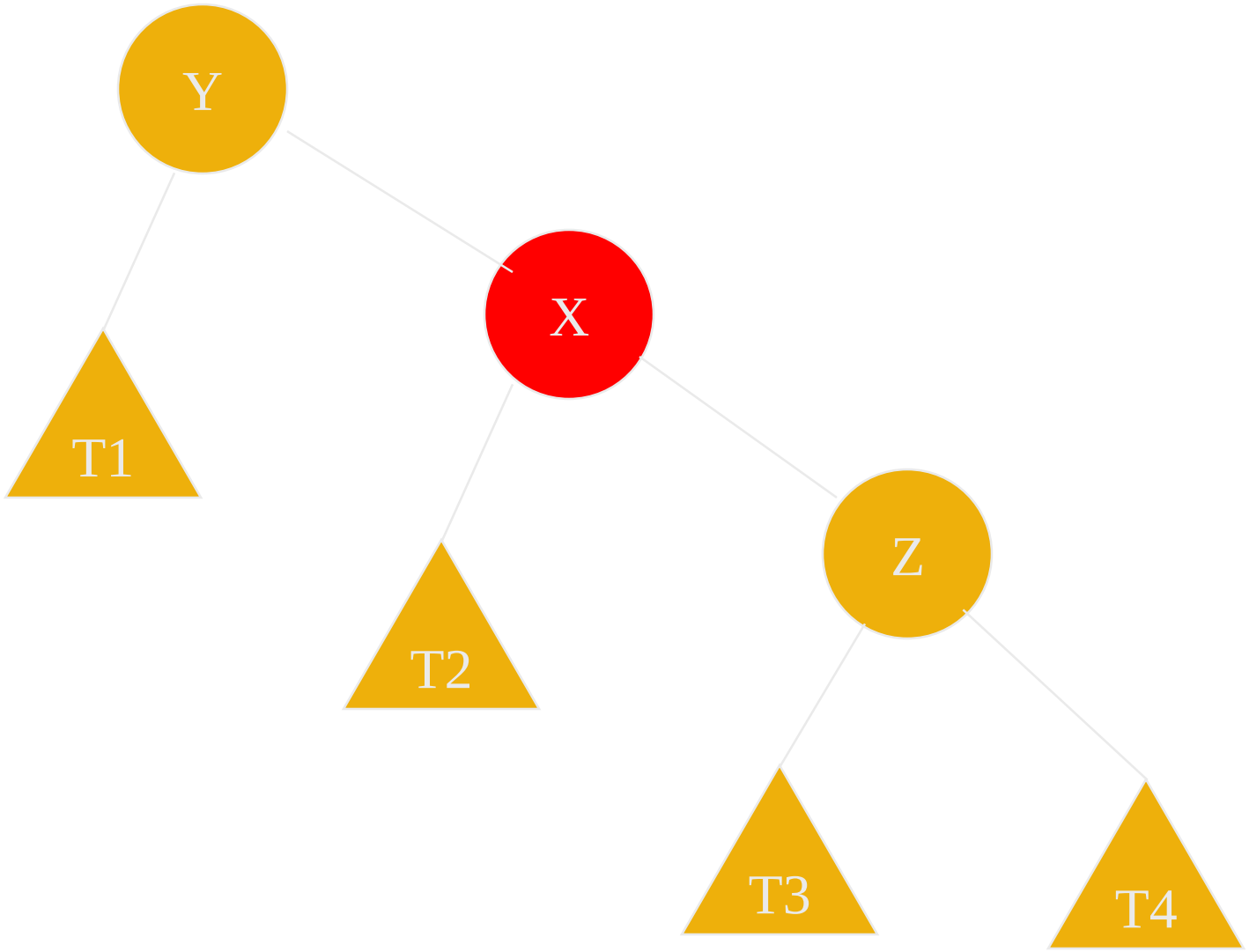
We have r’(y)+r’(z)<=2r’(x)-2

So we have variation of r(T0 by a single splaying substep is:

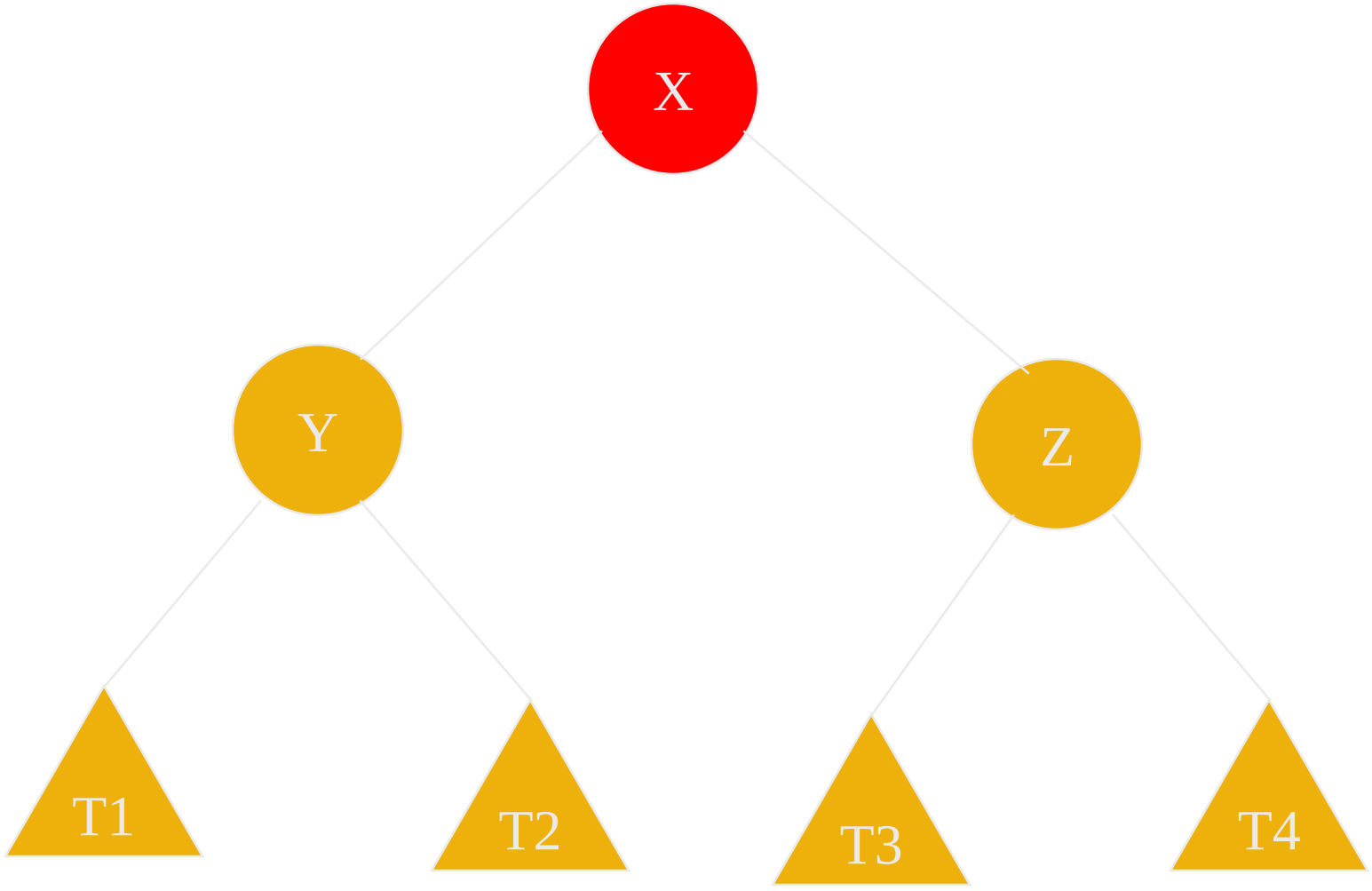
<=2(r’(x)-r(x))-2 <=3(r’(x)-r(x))-2

Since zig-zag takes 2 rotations, the amortized complexity will be 3(r’(x)-r(x))

# Before splaying:



# After splaying:



# Cont.

- Zig:

  variation of r(T) caused by a single splaying substep is:

  $r'(x)+r'(y)-r(x)-r(y)$

  $\leq r'(x)-r(x)$ ($r'(y)\leq r(y)$ and $r'(x)\geq r(x)$)

  $\leq 3(r'(x)-r(x))$

  Since zig only takes 1 rotation, so the amortized complexity will be $3(r'(x)-r(x))+1$

# Cont.

- Splaying node x consists of d/2 splaying substeps, recall d is the depth of x.
- The total amortized complexity will be:

  $<=\Sigma(3(r_i(x)-r_{i-1}(x))) + 1 \quad (1<=i<=d/2)$

  recall: the last step is a zig.

  $=3(r_{d/2}(x)-r_0(x)) +1$

  $<=3(r(t)-r(x))+1$ (r(t): rank of root)

# Cont.

- So from before we have:

  $3(r(t)-r(x))+1$

  $<=3r(t)+1$

  $=3\log n +1$

  thus, splaying takes O(log n).
- So for m operations of search, insertion and deletion, we have O(mlog n)
- This is better than the O(mn) worst-case complexity of BST

# Implementation

- LEDA ?
- Java implementation:

  SplayTree.java and etc from previous files(?)

  Modified printTree() to track level and child identity, add a small testing part.

# Demos

- A Demo written in Perl, it has some small bugs http://bobo.link.cs.cmu.edu/cgi-bin/splay/splay-cgi.pl
- An animation java applet: http://www.cs.technion.ac.il/~itai/ds2/framesplay/splay.html

# Conclusion

- A balanced binary search tree.
- Doesn’t need any extra information to be stored in the node, ie color, level, etc.
- Balanced in an amortized sense.
- Running time is O(mlog n) for m operations
- Can be adapted to the ways in which items are being accessed in a dictionary to achieve faster running times for the frequently accessed items.(O(1), AVL is about O(log n), etc.)

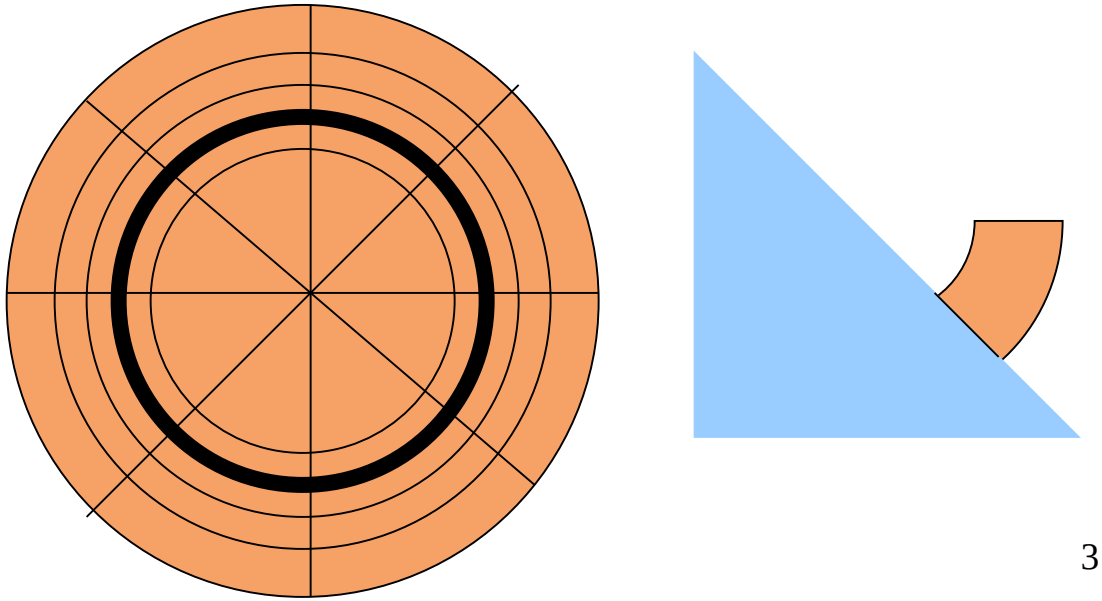
# B-Trees

# B-Trees

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL ) it is assumed that everything is in main memory.

# Concept behind B-Tree

To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses.



# B-Trees : Concept

- Most of the tree operations (search, insert, delete, max, min, etc ) require O(h) disk accesses where h is the height of the tree.

- B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node.

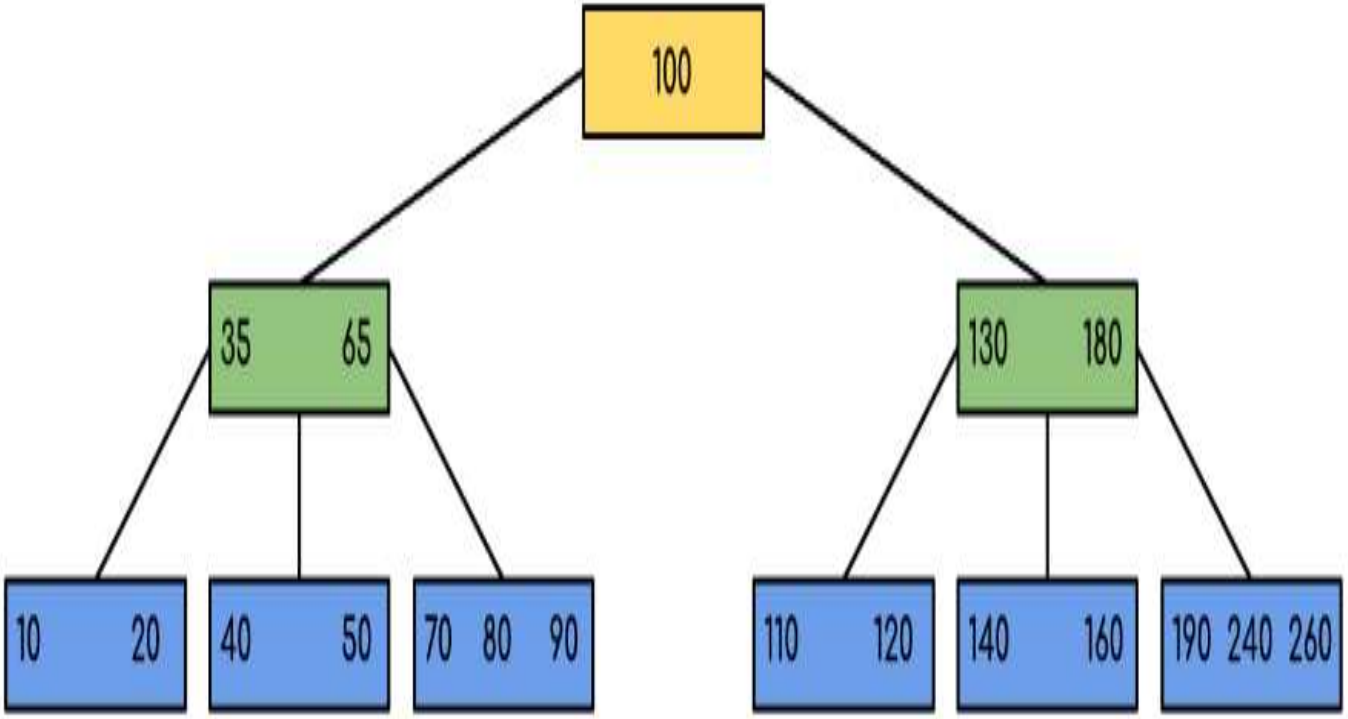
- Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, etc

# Properties of B-Tree

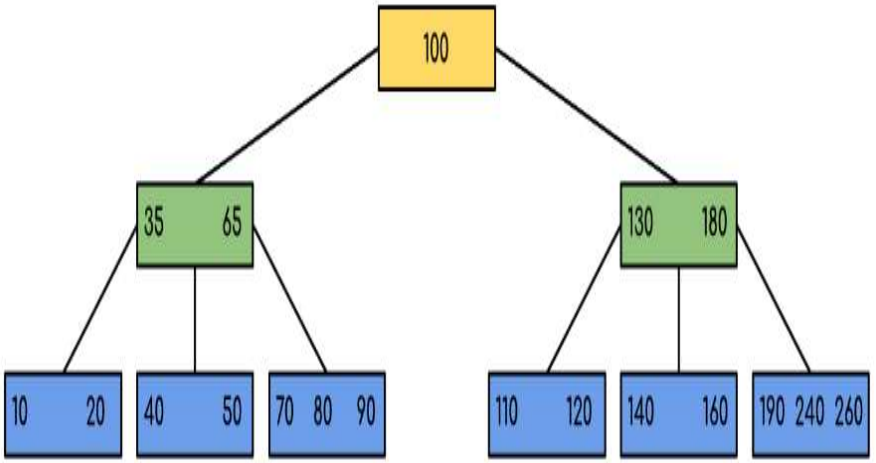
For balancing the tree each node should contain n/2 keys. So the B tree of order n can be defined as:

1.All leaf nodes should be at same level.
2.All leaf nodes can contain maximum n-1 keys.
3.The root has at least two children.
4.The maximum number of children should be n and each node can contain k keys. Where, k<=n-1.
5.Each node has at least n/2 and maximum n nonempty children.
6.Keys in the non-leaf node will divide the left and right sub-tree where the value of left subtree keys will be less and value of right subtree keys will be more than that particular key.

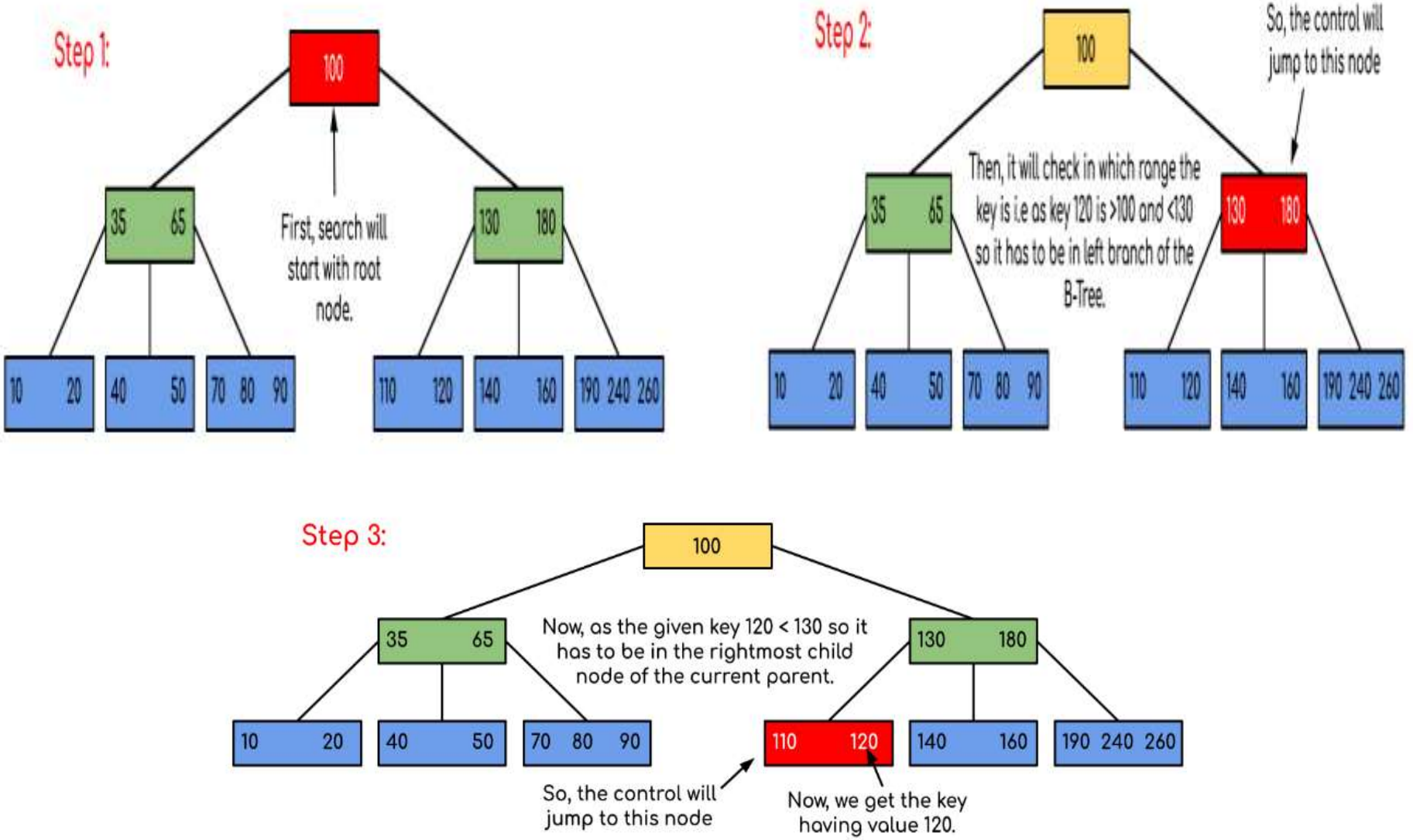
# Example



# Example: Searching 120 in the given B-Tree



# Solution:


Step 1:
100
35 65
First, search will start with root node.
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260
Step 2:
100
So, the control will jump to this node
35 65
Then, it will check in which range the key is i.e as key 120 is >100 and <130 so it has to be in left branch of the B-Tree.
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260
Step 3:
100
35 65
Now, as the given key 120 < 130 so it has to be in the rightmost child node of the current parent.
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260
So, the control will jump to this node
Now, we get the key having value 120.

# B-Tree performance

Let $h$ = height of the B-tree.

get(k): at most $h$ disk accesses. O($h$)

put(k): at most $3h + 1$ disk accesses. O($h$)

remove(k): at most $3h$ disk accesses. O($h$)

$h < \log_d (n + 1)/2 + 1$ where $d = \lceil m/2 \rceil$ (Sahni, p.641).

An important point is that the constant factors are relatively low.

$m$ should be chosen so as to match the maximum node size to the block size on the disk.

Example: $m = 128$, $d = 64$, $n \approx 64^3 = 262144$ , $h = 4$.

# 2-3 Trees

A B-tree of order *m* is a kind of *m*-way search tree.

A B-Tree of order 3 is called a 2-3 Tree.

In a 2-3 tree, each internal node has either 2 or 3 children.

In practical applications, however, B-Trees of large order (e.g., *m* = 128) are more common than low-order B-Trees such as 2-3 trees.

# Single valued

If a node is single-valued then it has two children. Left children will contain values less than parent value, and right children will contain values greater than parent value.

# Double valued

If a node has two values then it will have three children. Left children will contain values lesser than the left parent value, and middle children will contain values greater than the left parent value but less than the right parent value. Right children will contain a value greater than the right parent's value. Since, each node has either two children or three children, that's why it is called 2-3 trees. It is a height-balanced tree, and the reason is all the leaf nodes will be at the same level.

# There are three operations in this tree:

## 1. Search

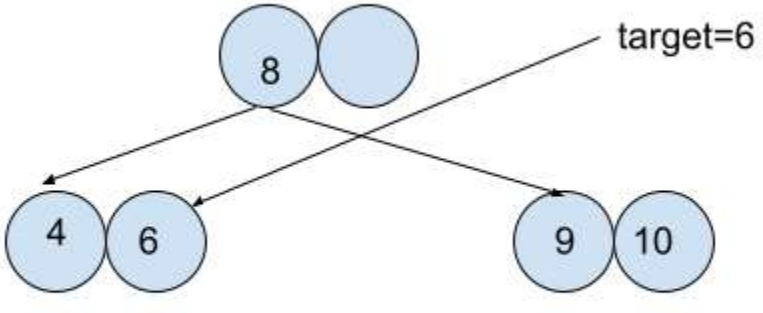
Search is the operation where we are given the root node and target value. If the value is available in the tree, it returns true; else, it will return false.
We can use recursion to search for any element in the tree.
**Case 1:**
If the current node is single-valued and the value is lesser than the node's value, then call the recursive function for the left child. Else, call the recursive function for the right child.
**Case 2:**
If the current node is double valued, and if the value is lesser than the left value, then call the recursive function for the left child. If the target element is greater than the current node's left value and lesser than the current node's right value, then call the recursive function for middle children. Else, call the recursive for the right child.

## Insertion

If we want to insert any element in the tree, then we will find its correct position, and then we will insert it. We can have three cases in the insertion operation:

**Case1:**

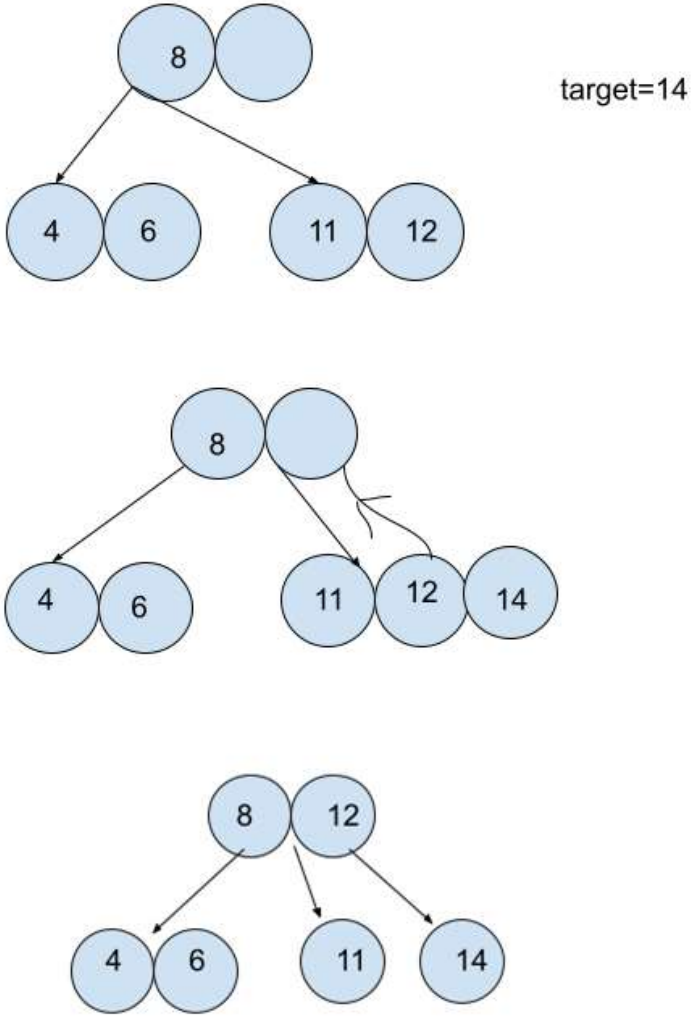
Suppose the node at which we want to put the element contains a single value. In this case, we will simply insert the element.




In the above example, we simply put the target in its correct position.

**Case2:**

If the node at which we want to insert the target element contains the double value and its parent is single-valued, then we will put the element at the node, and the middle value of the node, and the middle value of the node will be shifted to its parent. And the current node will be split into two separate nodes.



**Case 3:**
If the node at which we want to insert is a double-valued node and its parent is also a double-valued node. We will shift the middle element to the parent node and split the current node. Now its parent has three values, so it will also shift its middle element to its parent node and split the parent node into two separate nodes.

## Deletion

A value is removed after being replaced by its in-order successor in order to be deleted. Two nodes must be combined together if a node has less than one data value remaining. After removing a value, a node is merged with another if it becomes empty.