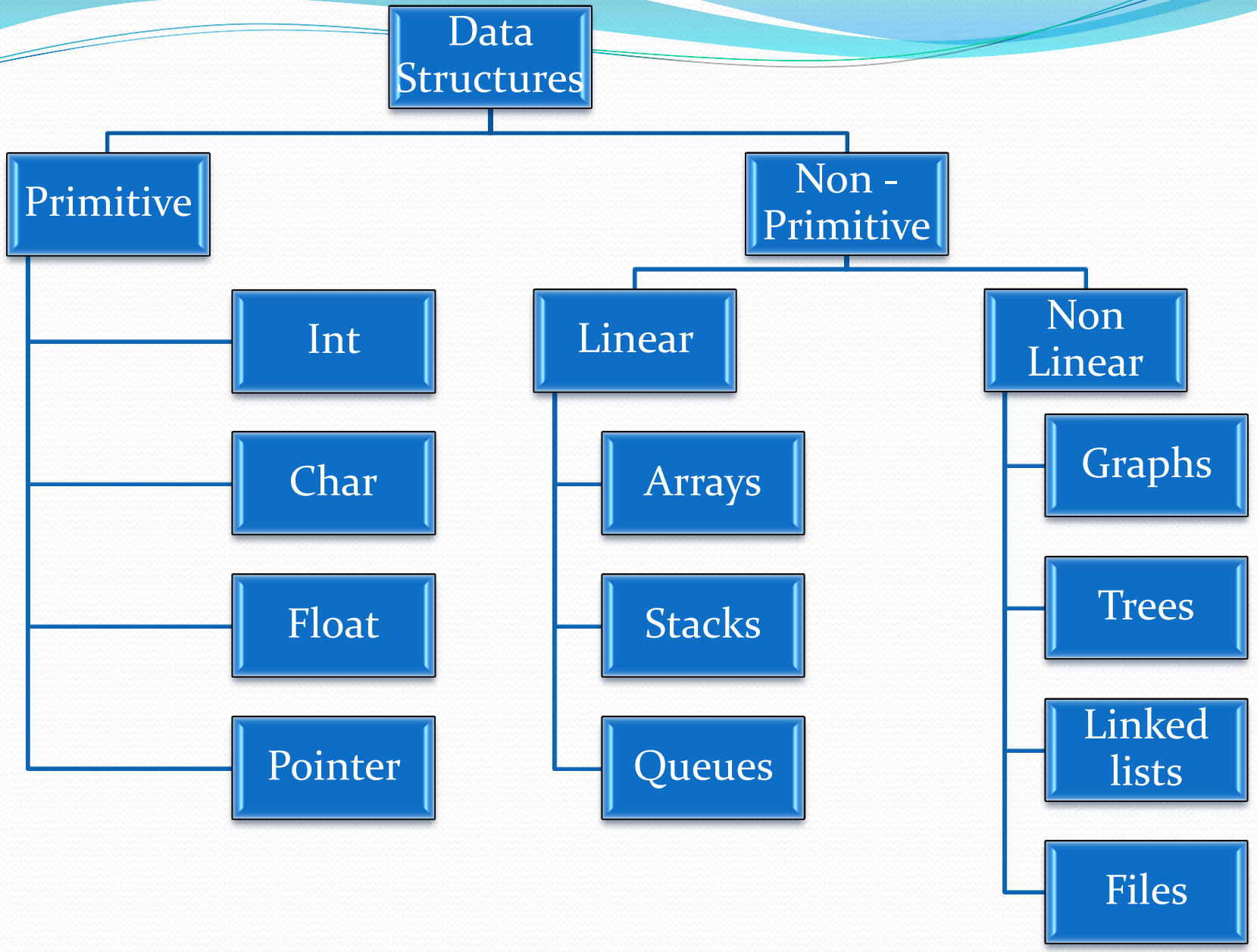
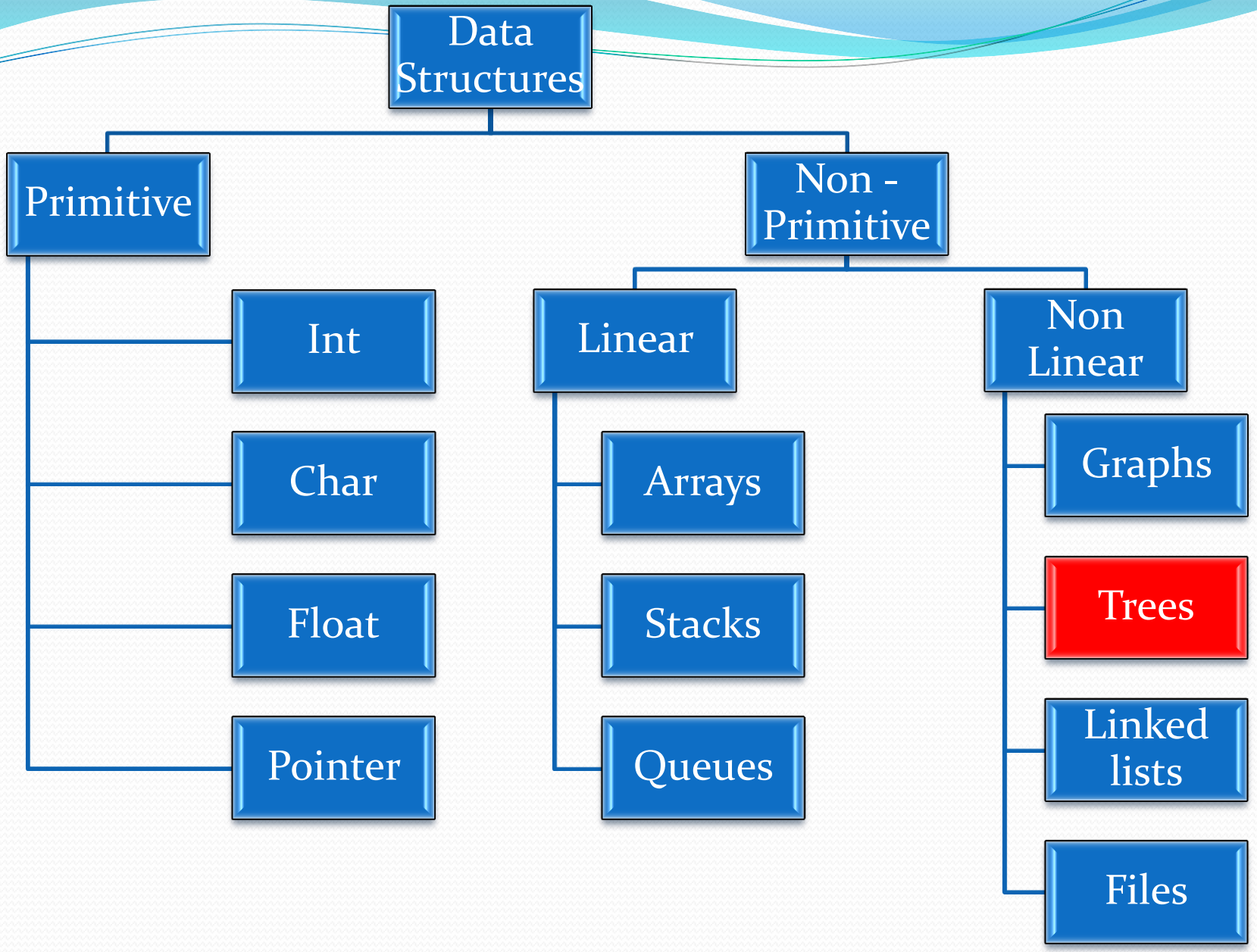

Data Structures
Primitive
Non - Primitive
Int
Char
Float
Pointer
Linear
Non Linear
Arrays
Stacks
Queues
Graphs
Trees
Linked lists
Files

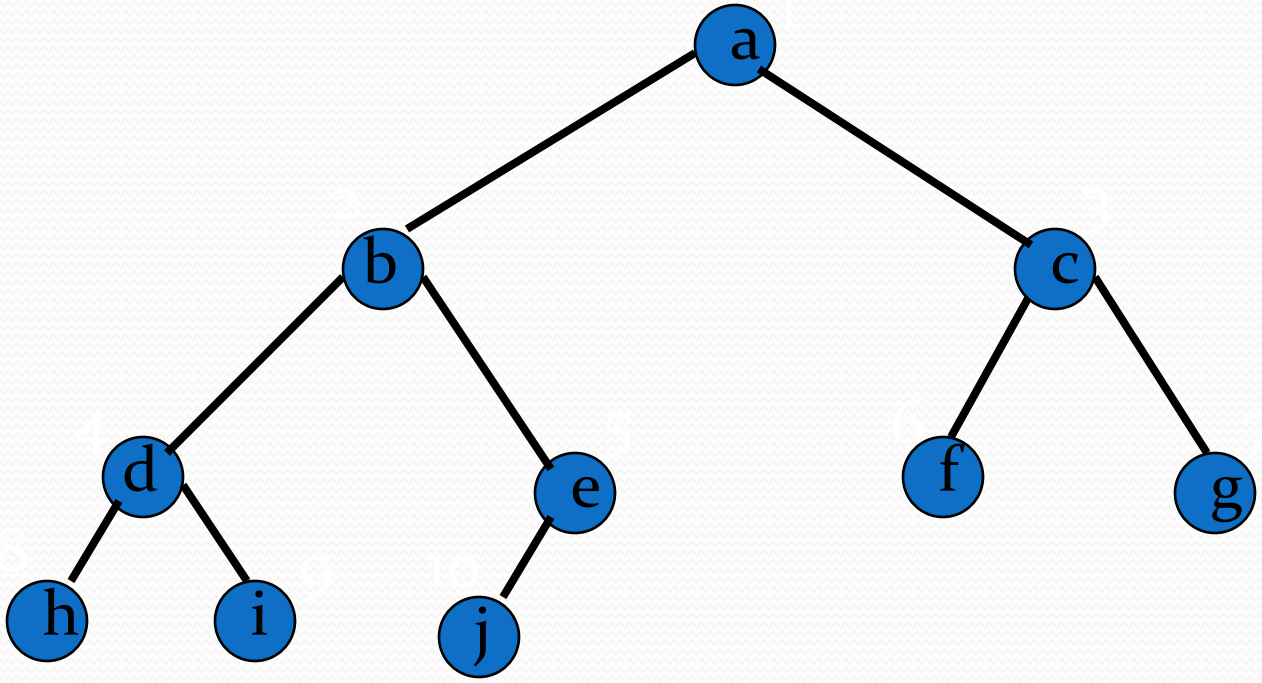

Data Structures
Primitive
Non - Primitive
Int
Char
Float
Pointer
Linear
Non Linear
Arrays
Stacks
Queues
Graphs
Trees
Linked lists
Files

Binary Tree Representation

- Array representation.
- Linked representation.

Array Representation

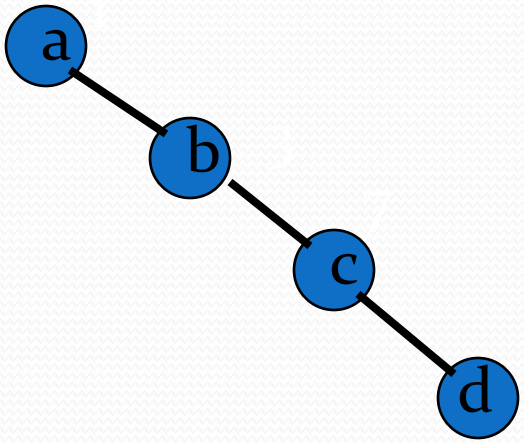
- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in tree[i].



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.**

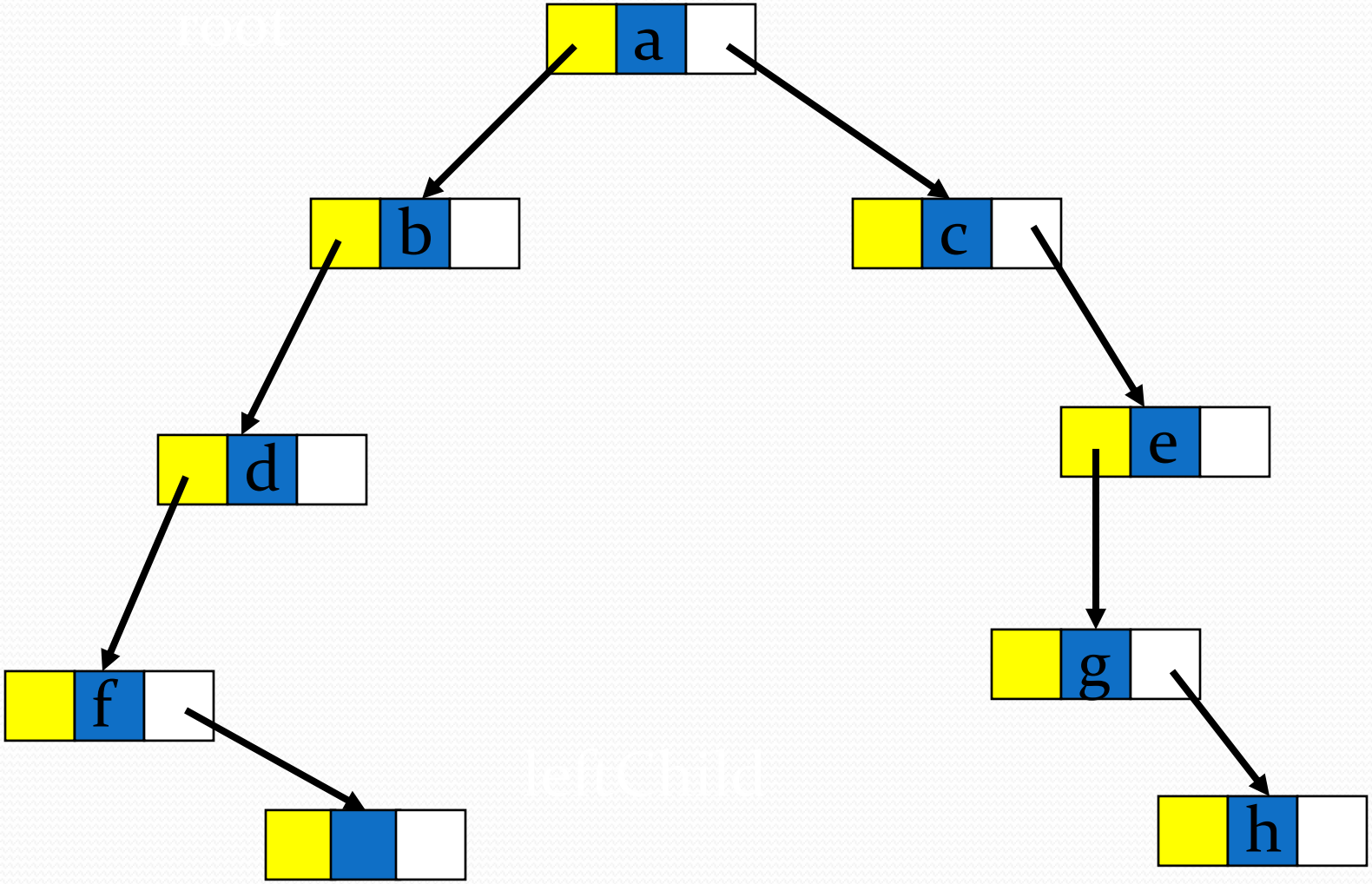
Linked Representation

- 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

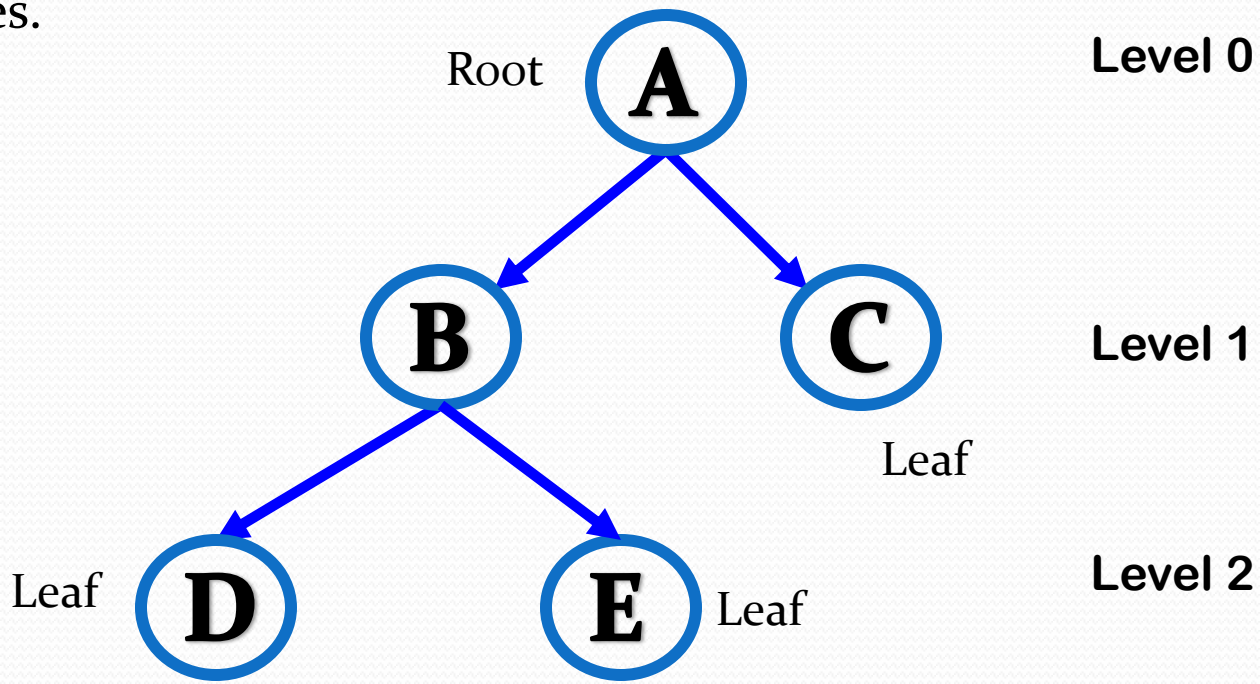
```
  typedef struct node
{
     int data;
     struct node *lc,*rc;
};
```

Linked Representation Example

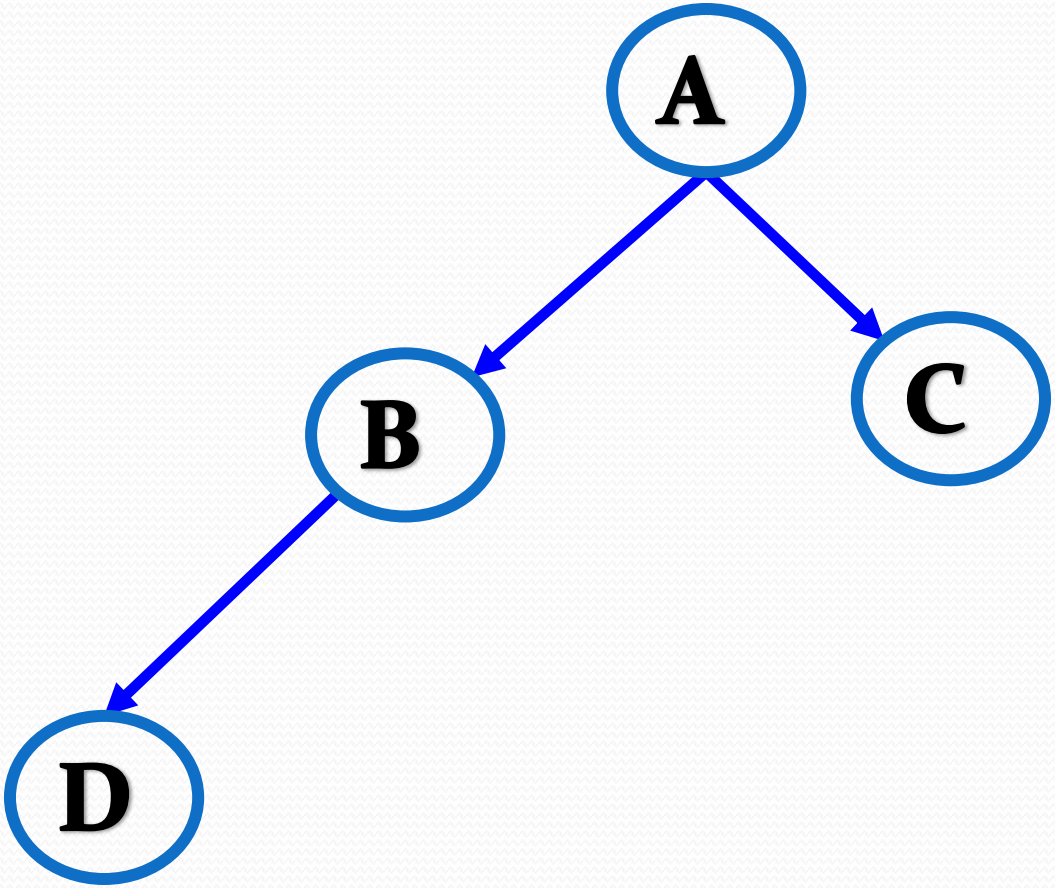


TREE

- 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.



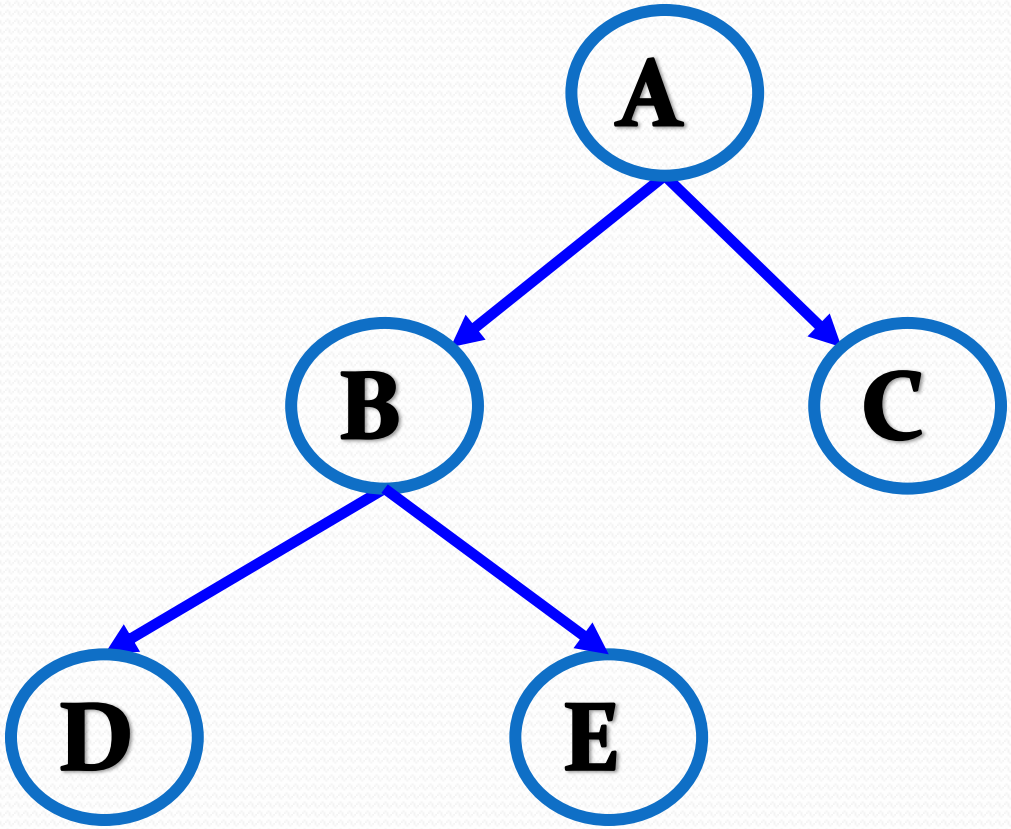
Binary Tree



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.

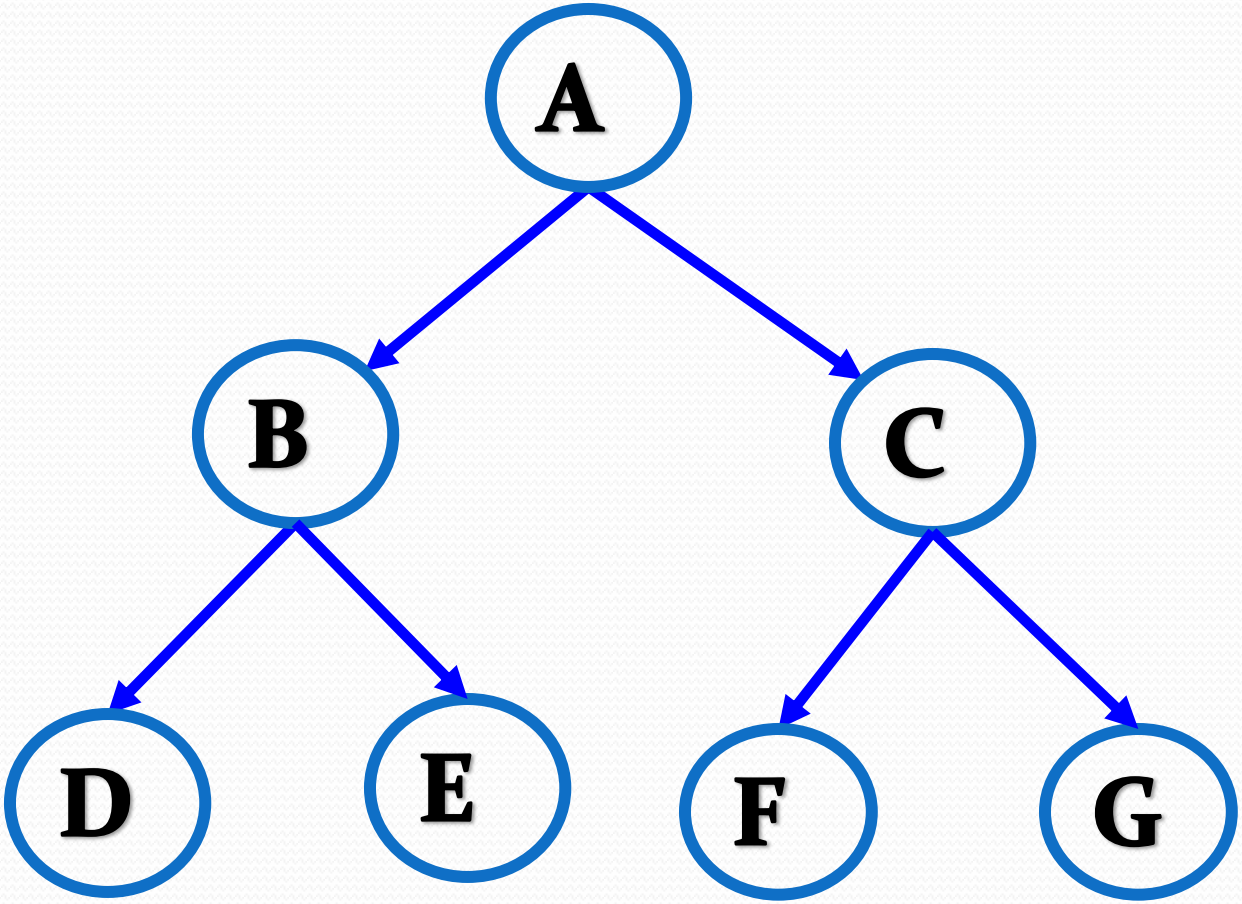
Strict Binary Tree



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

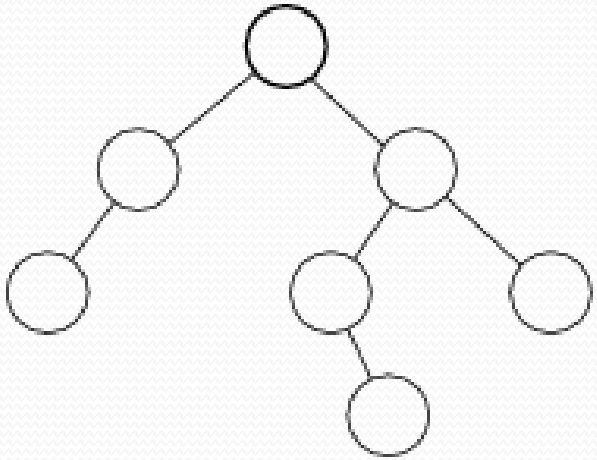


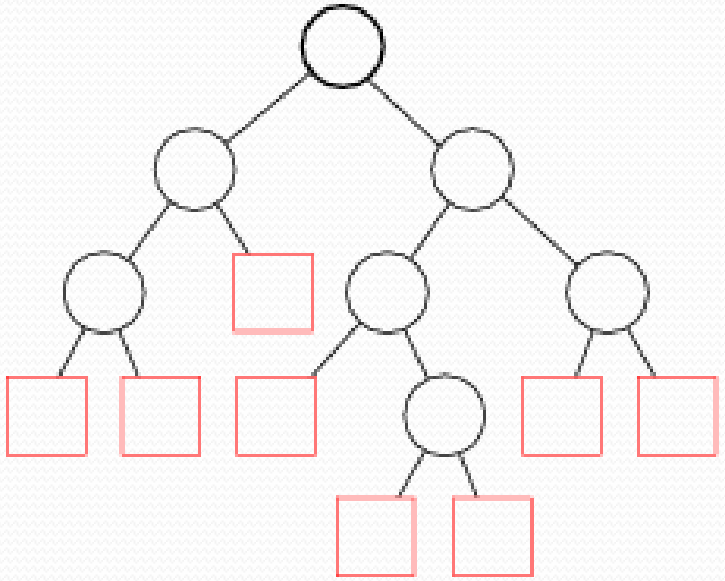
Complete Binary Tree

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.





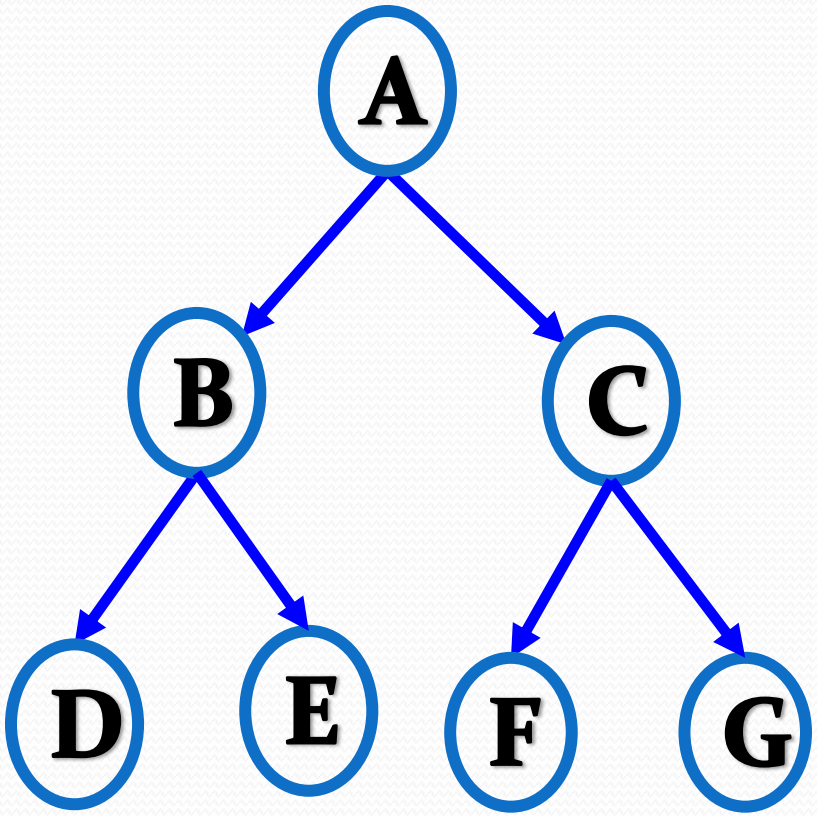
Extended Binary Tree

- 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.”

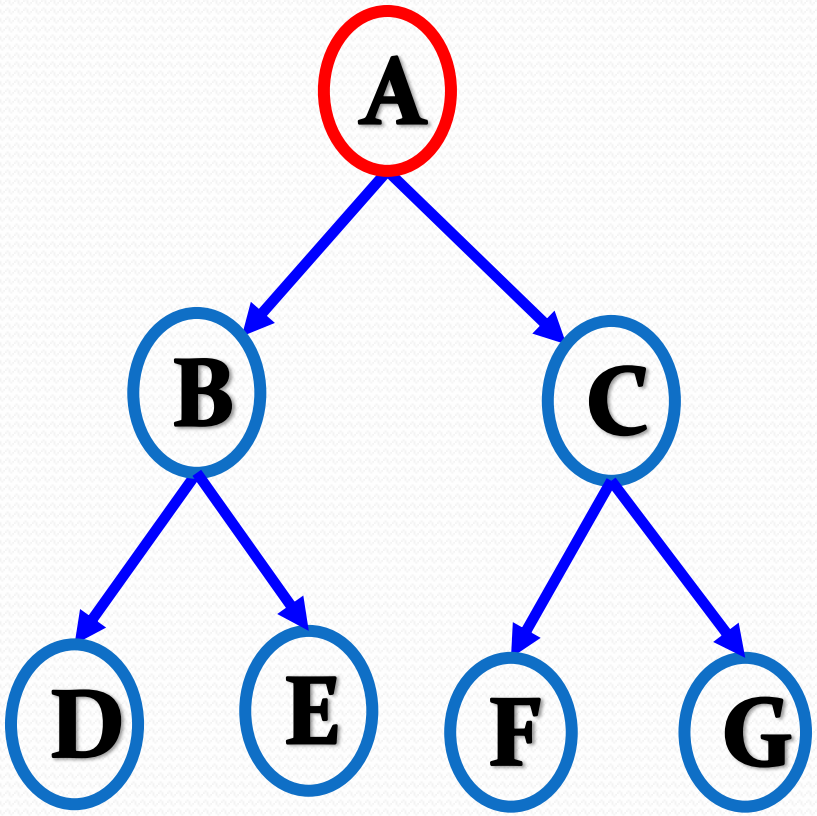
- The nodes from the original **tree** are then called as internal nodes, while the “special nodes” are called as external nodes.

Tree Traversal : Pre order



- Pre order (N L R)

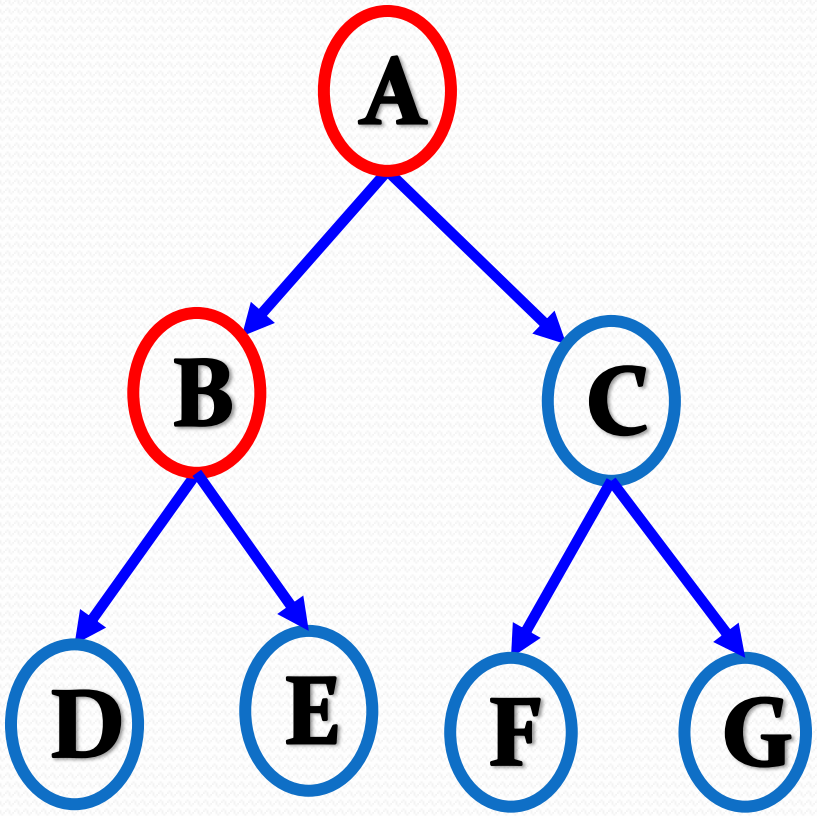
Tree Traversal : Pre order



- Pre order (N L R)

  A

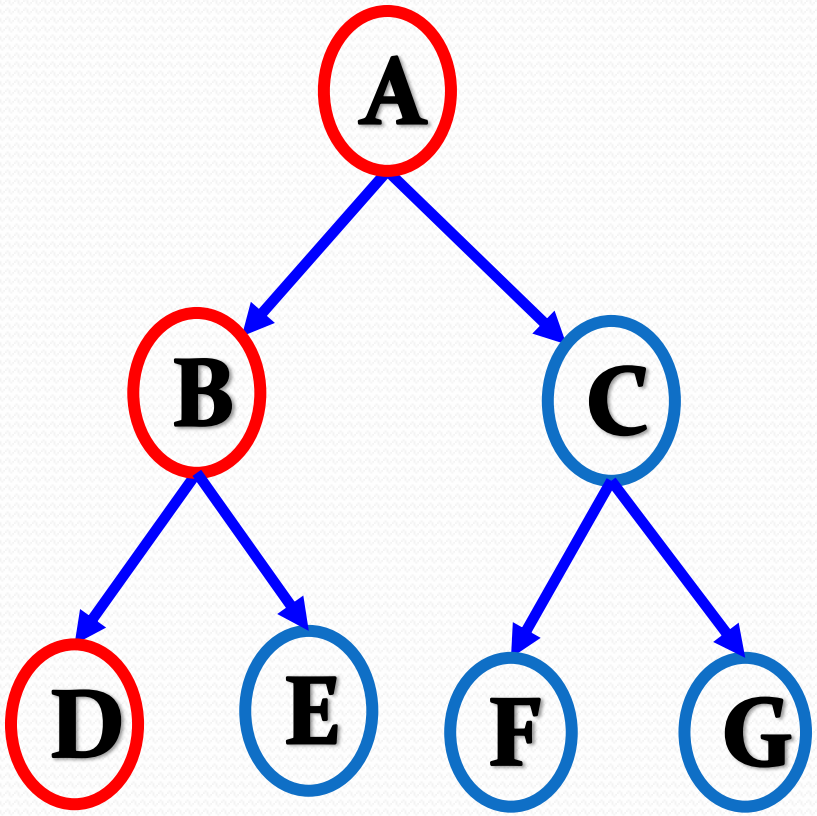
Tree Traversal : Pre order



- Pre order (N L R)

  A, B

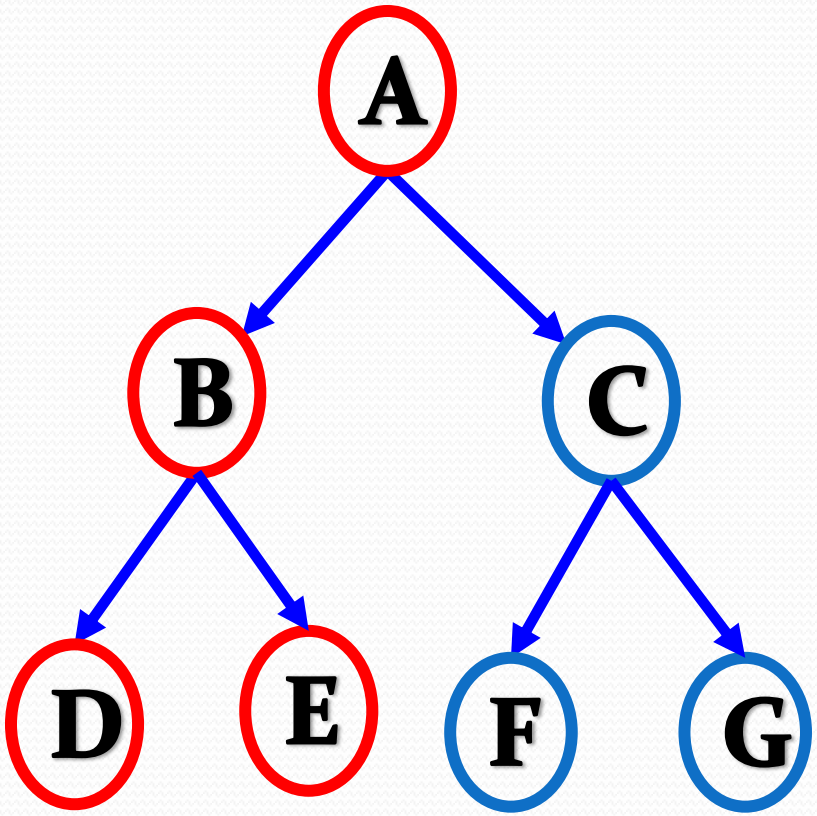
Tree Traversal : Pre order



- Pre order (N L R)

  A, B, D

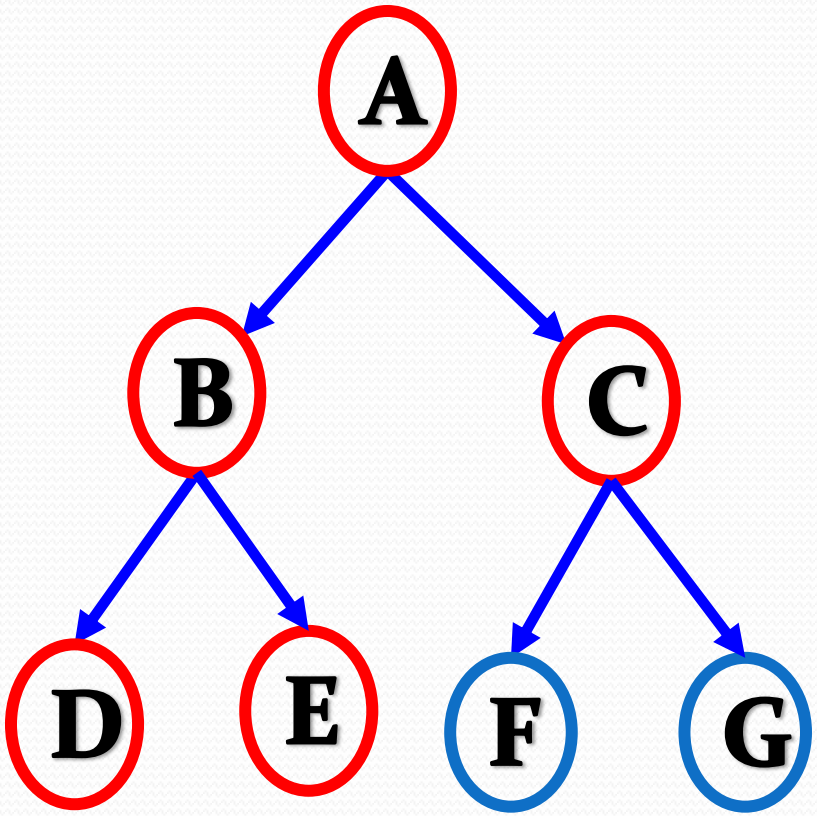
Tree Traversal : Pre order



- Pre order (N L R)

  A, B, D, E

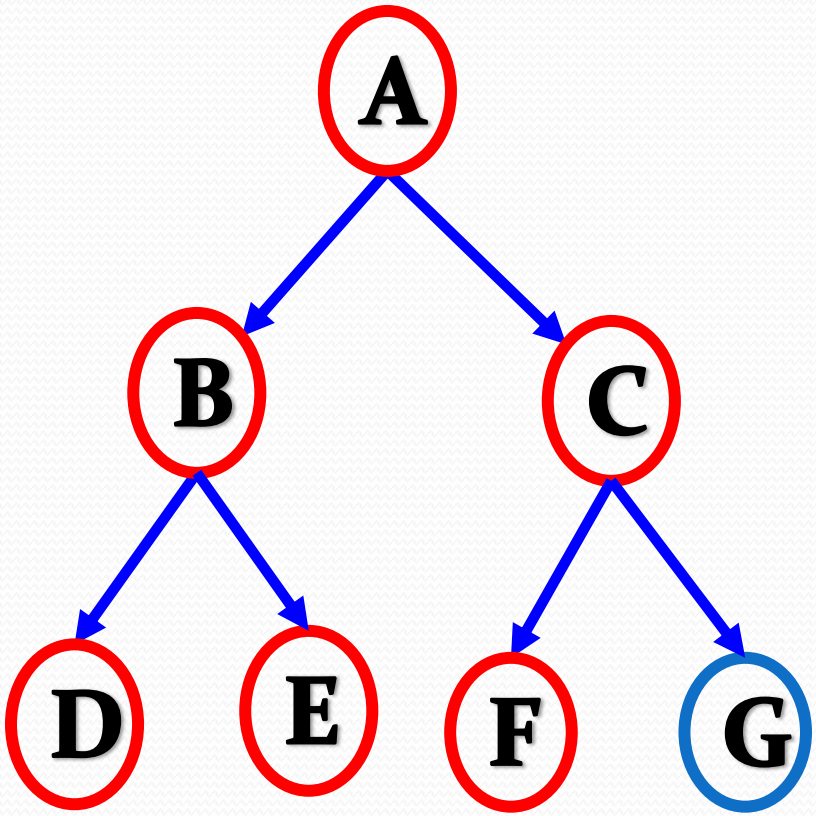
Tree Traversal : Pre order



- Pre order (N L R)

  A, B, D, E, C

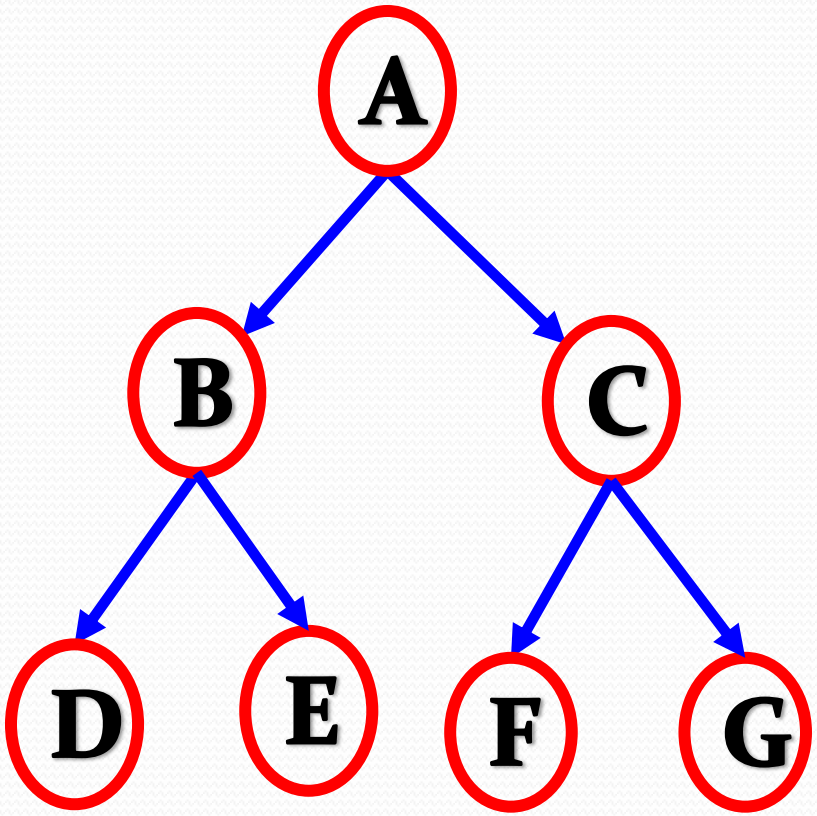
Tree Traversal : Pre order



- Pre order (N L R)

  A, B, D, E, C, F

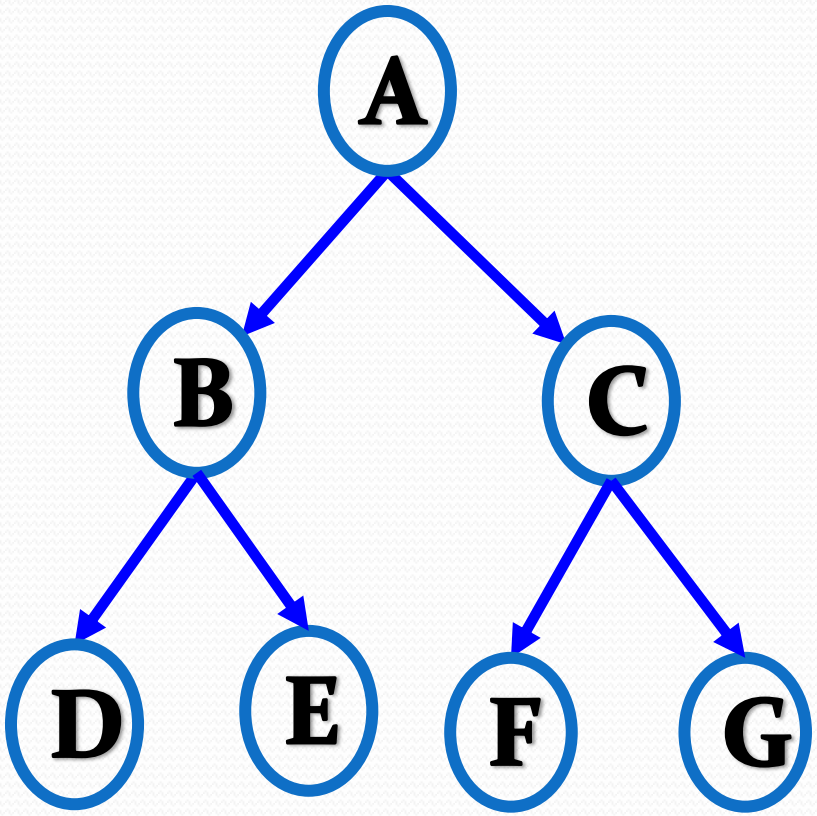
Tree Traversal : Pre order



- Pre order (N L R)

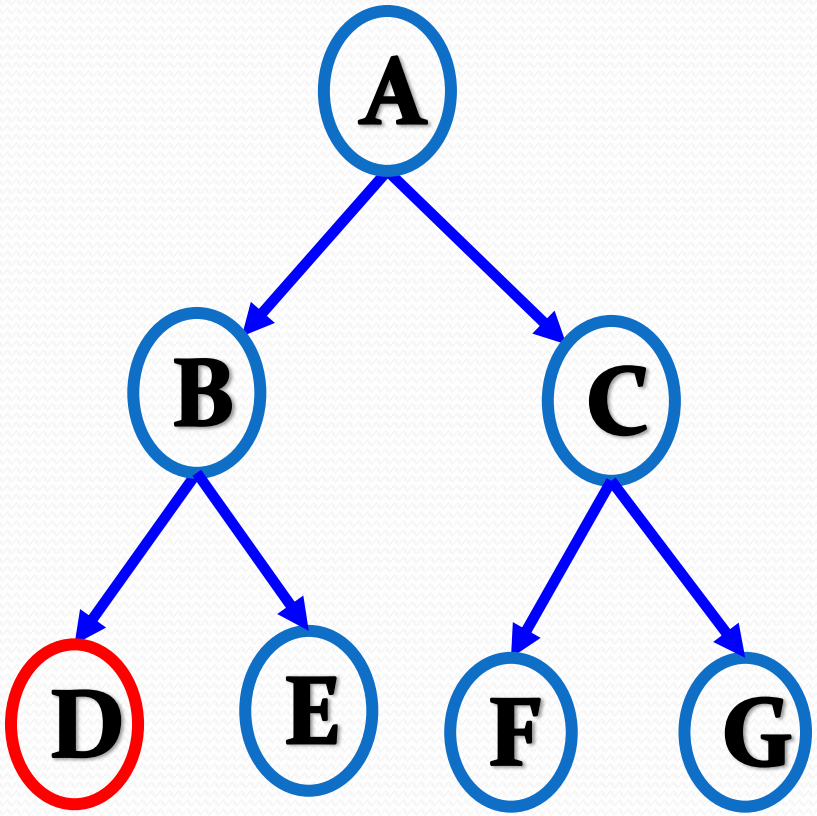
  A, B, D, E, C, F, G

Tree Traversal : In order



- In order (L N R)

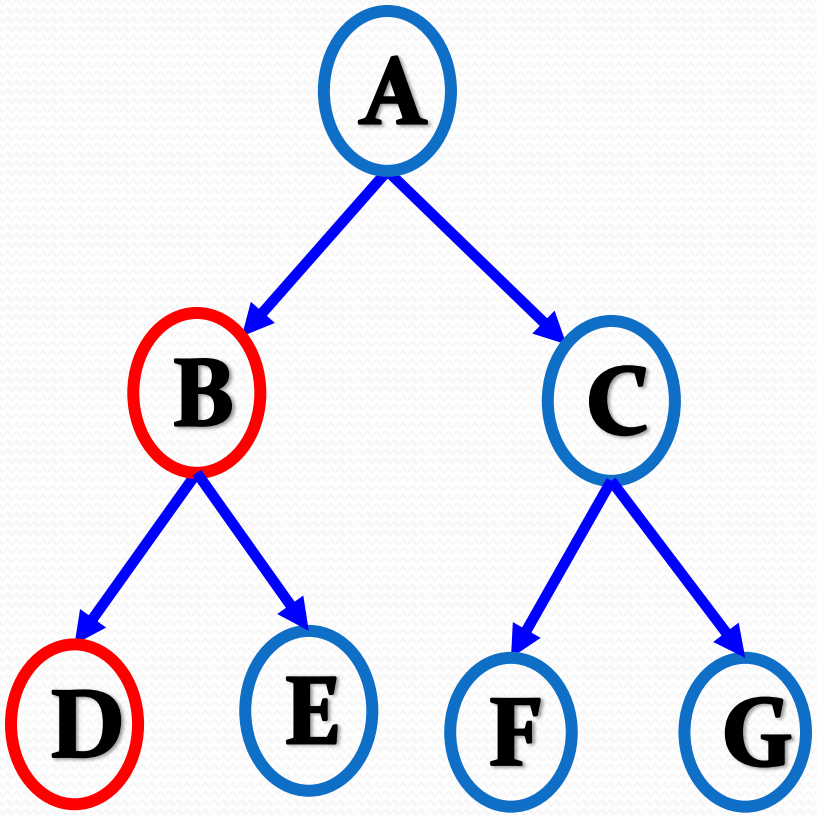
Tree Traversal : In order



- In order (L N R)

  D

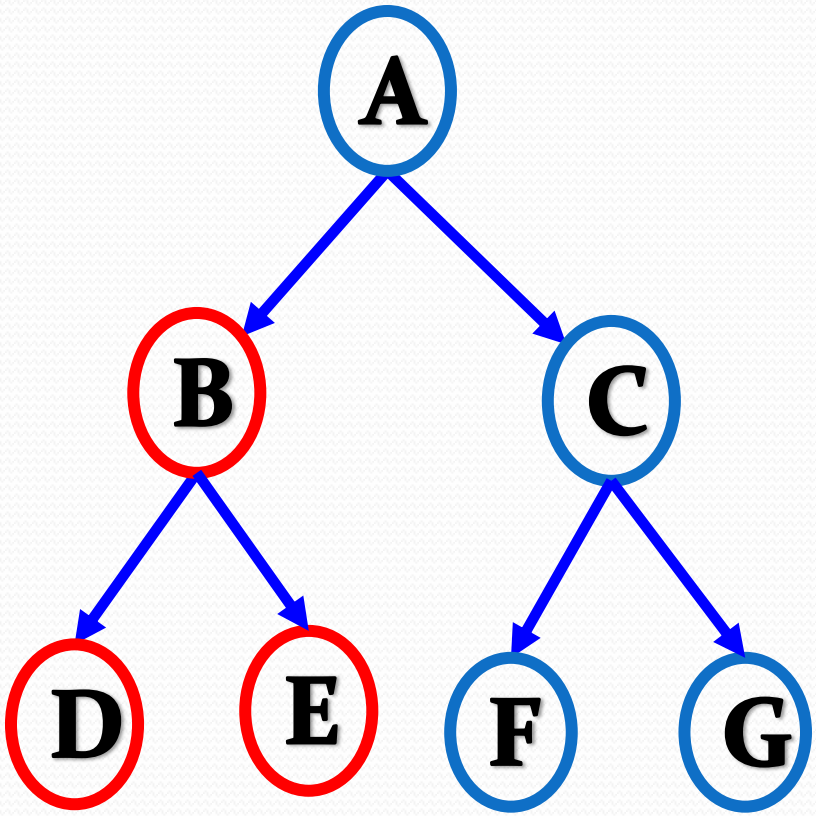
Tree Traversal : In order



- In order (L N R)

  D, B

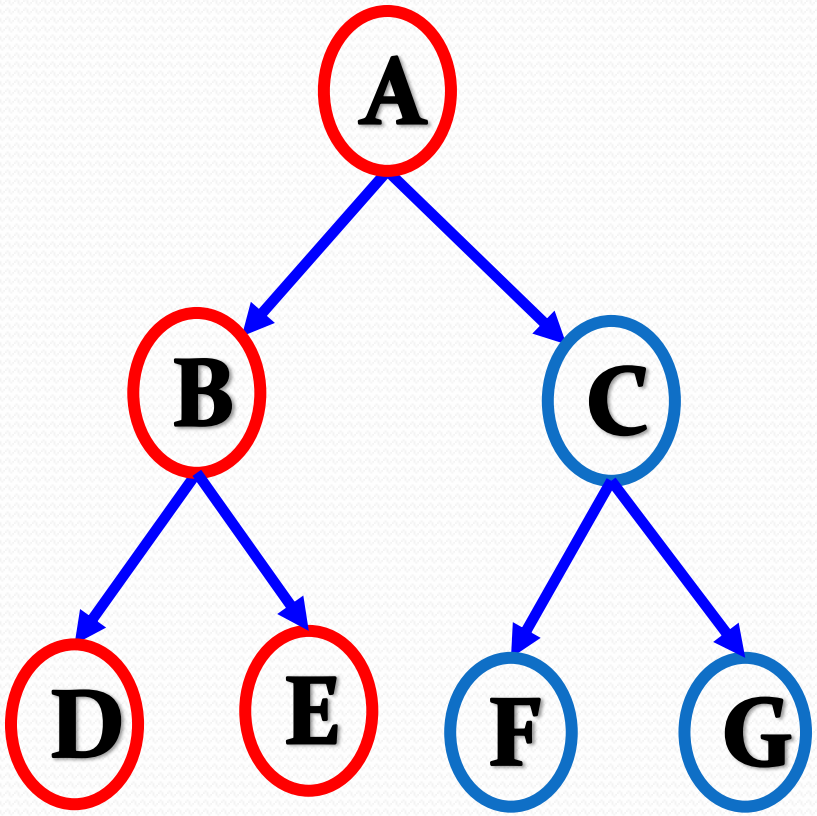
Tree Traversal : In order



- In order (L N R)

  D, B, E

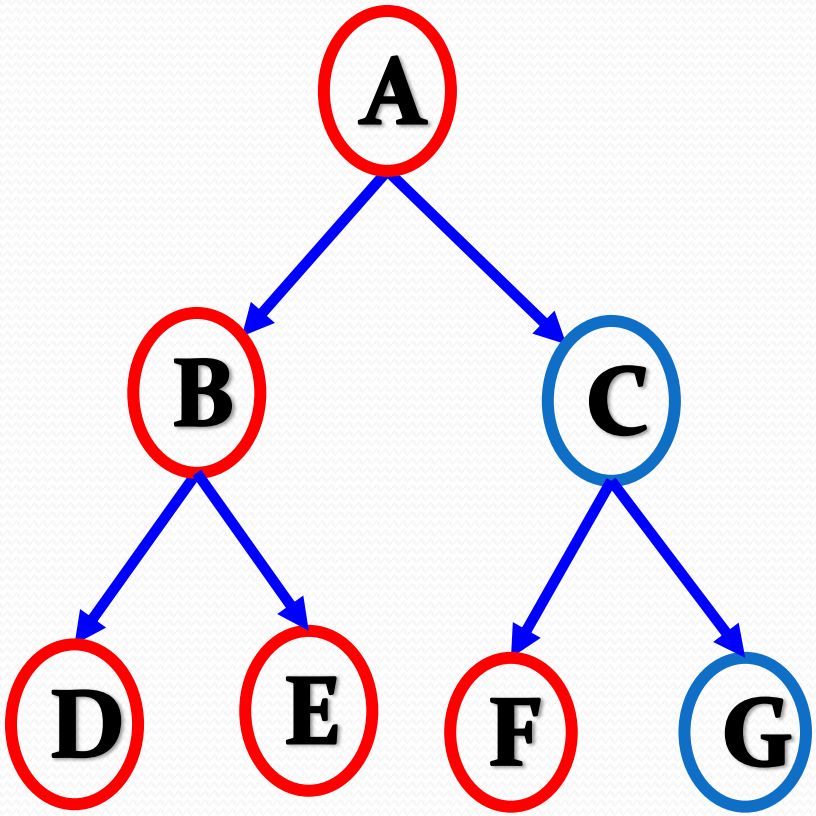
Tree Traversal : In order



- In order (L N R)

  D, B, E, A

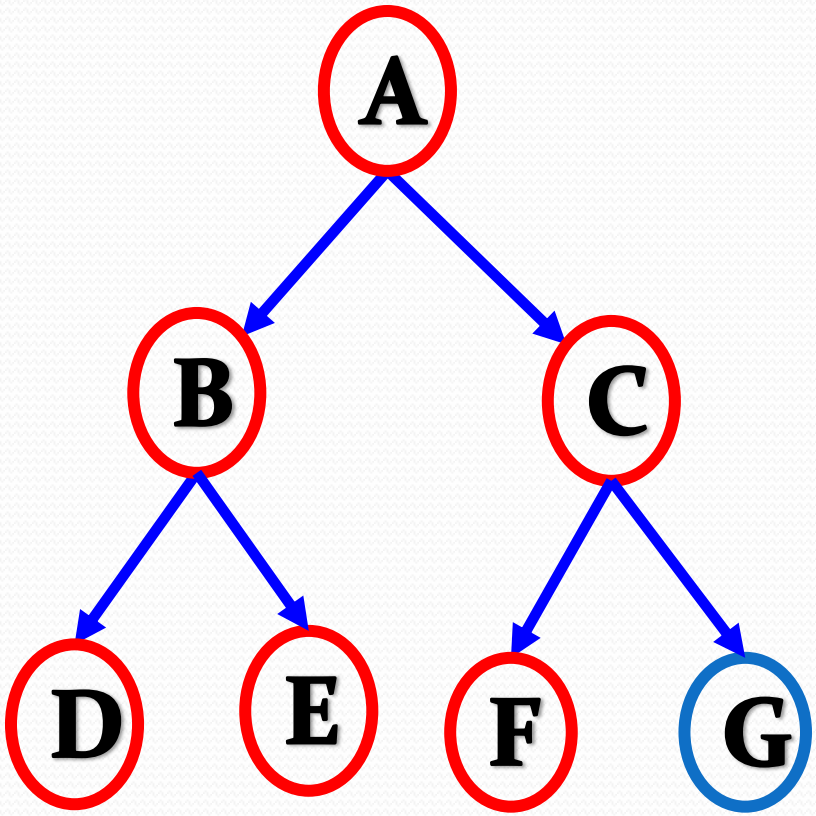
Tree Traversal : In order



- In order (L N R)

  D, B, E, A, F

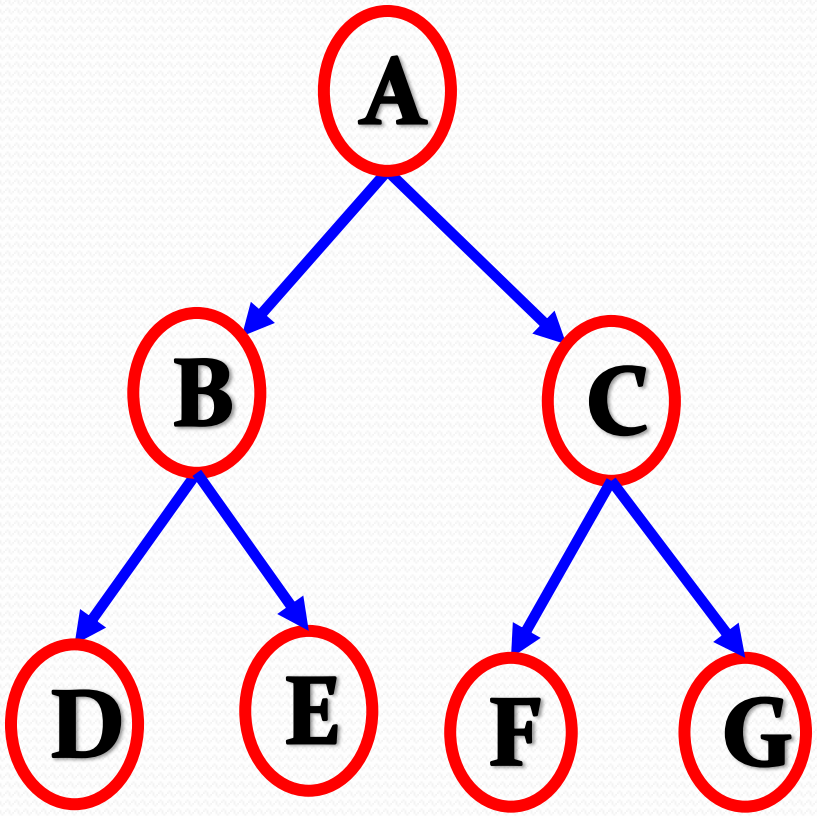
Tree Traversal : In order



- In order (L N R)

  D, B, E, A, F, C

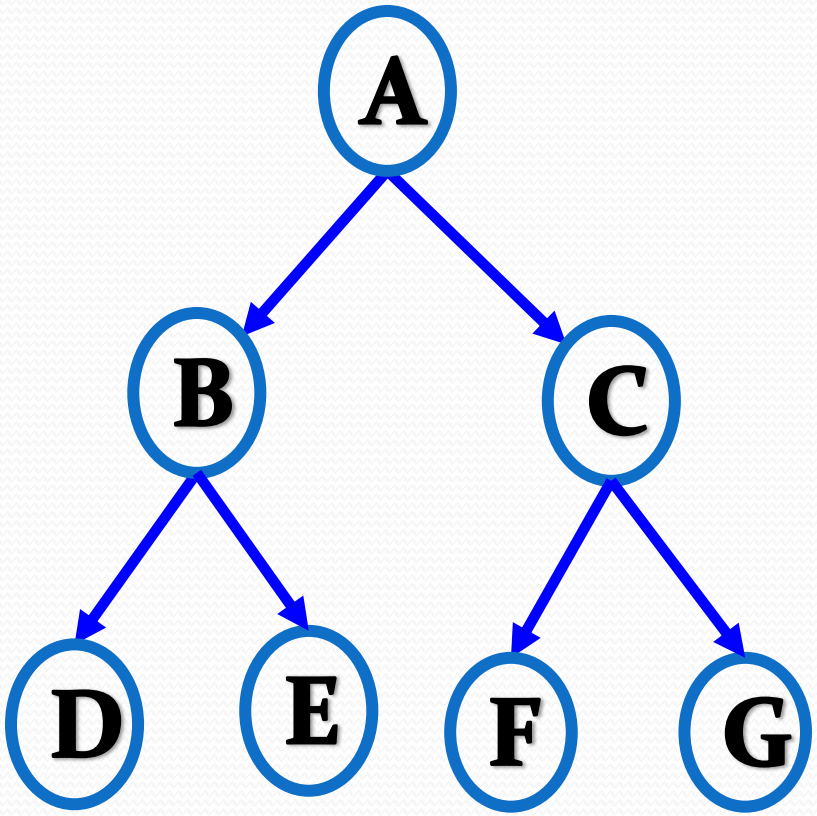
Tree Traversal : In order



- In order (L N R)

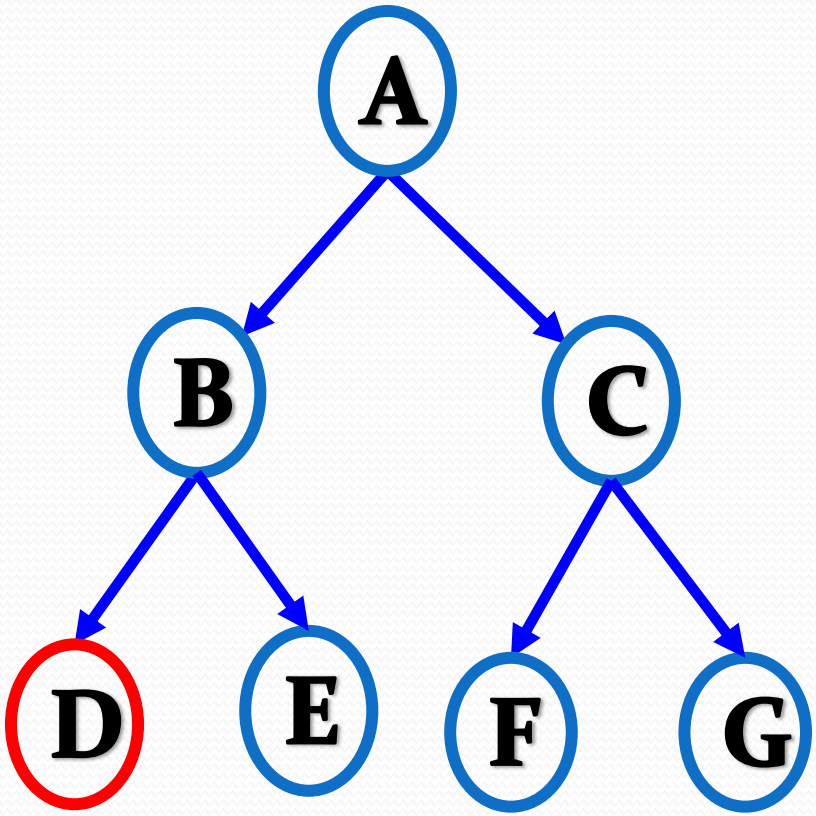
  D, B, E, A, F, C, G

Tree Traversal : Post order



- Post order (L R N)

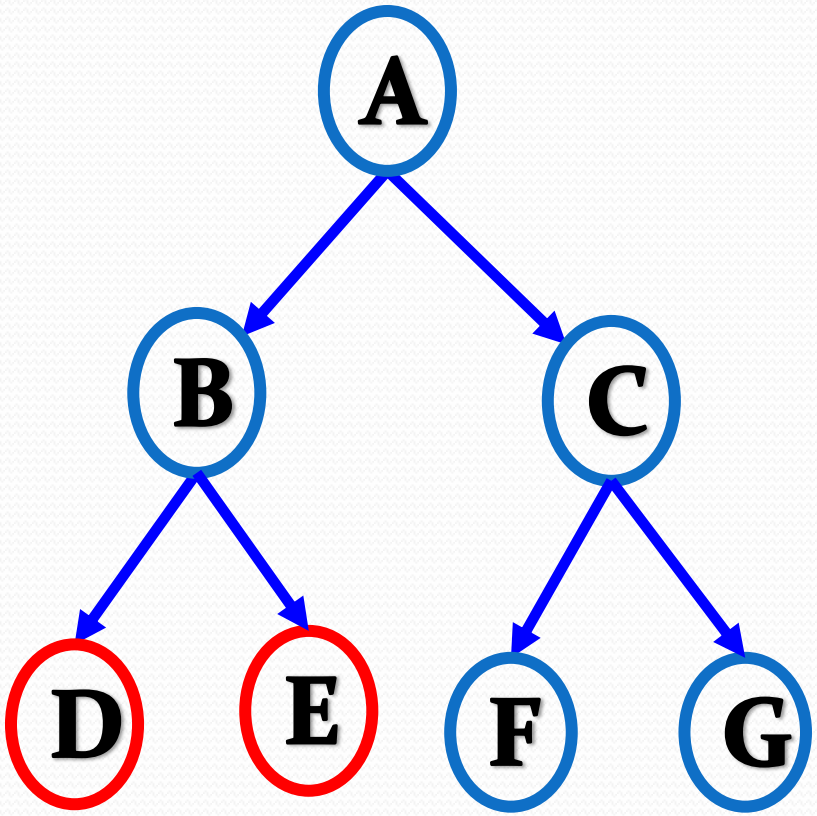
Tree Traversal : Post order



- Post order (L R N)

  D

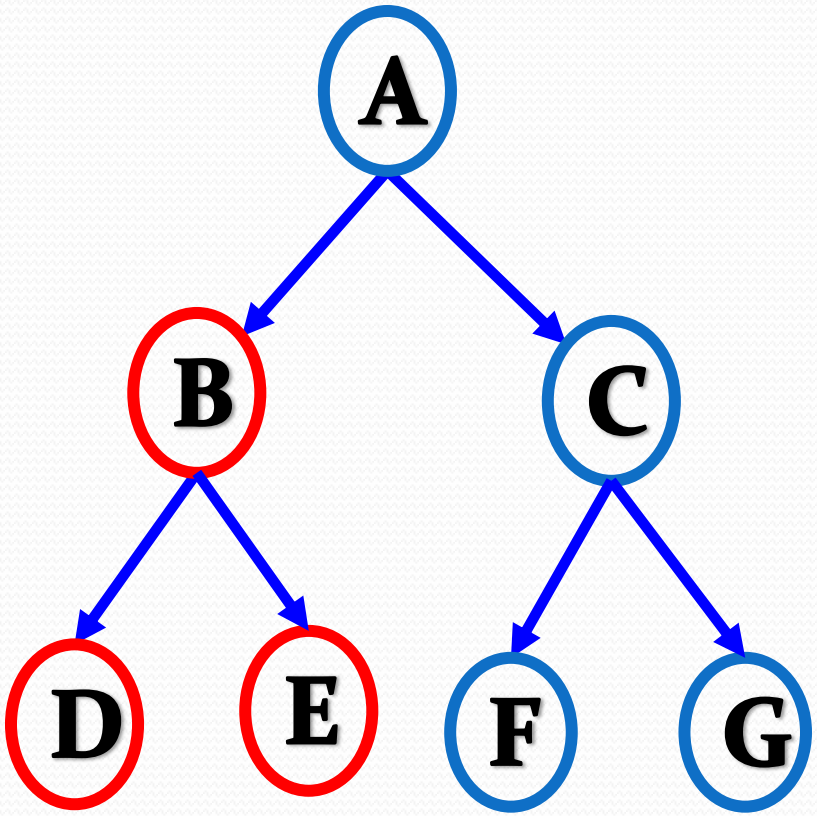
Tree Traversal : Post order



- Post order (L R N)

  D, E

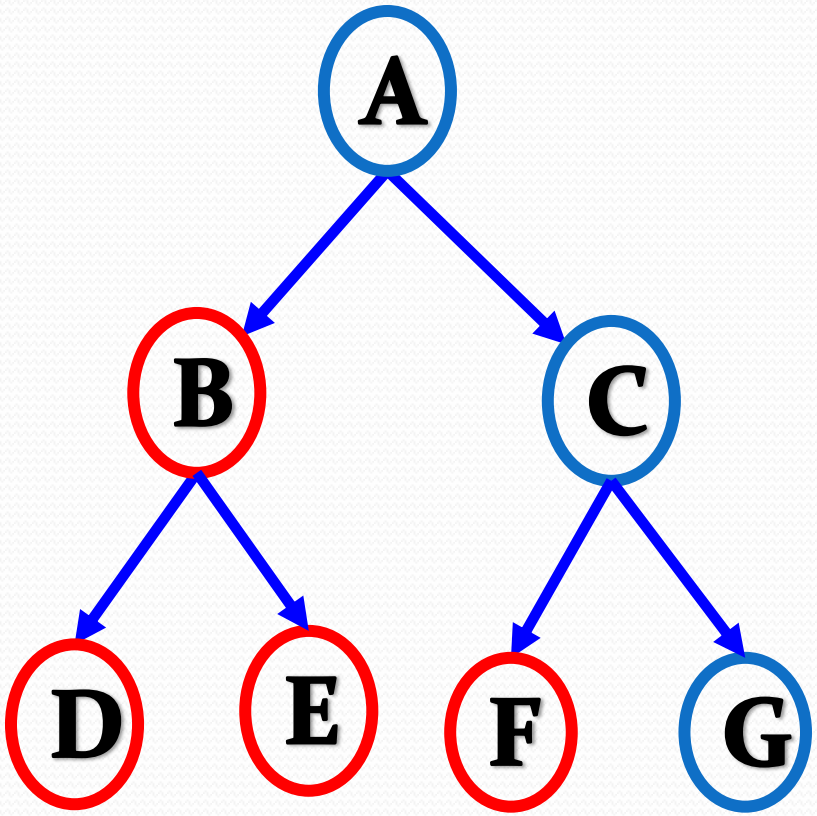
Tree Traversal : Post order



- Post order (L R N)

  D, E, B

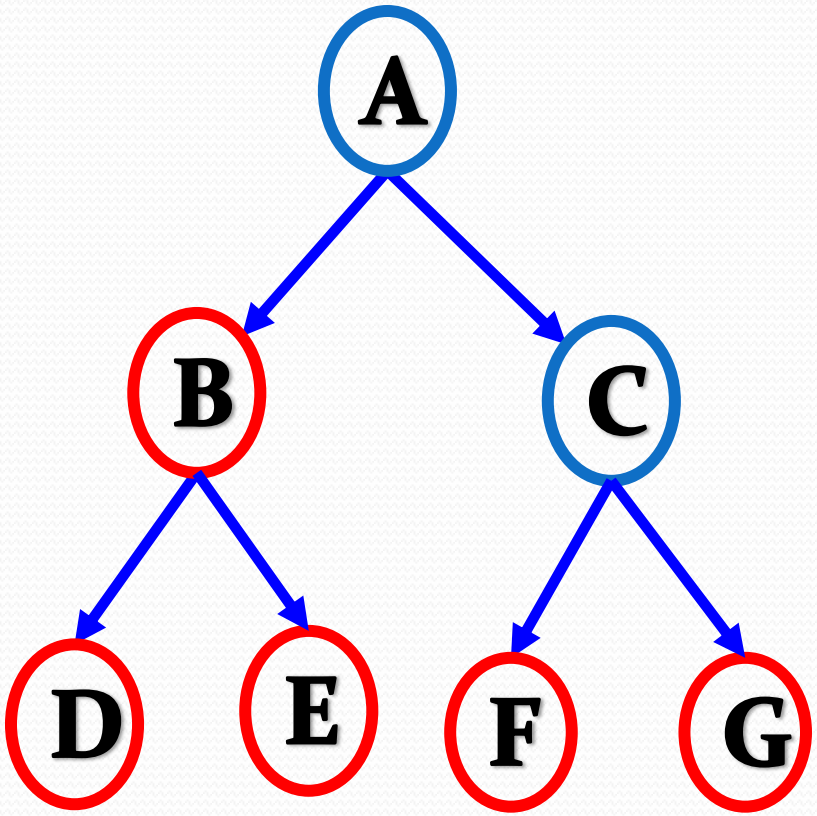
Tree Traversal : Post order



- Post order (L R N)

  D, E, B, F

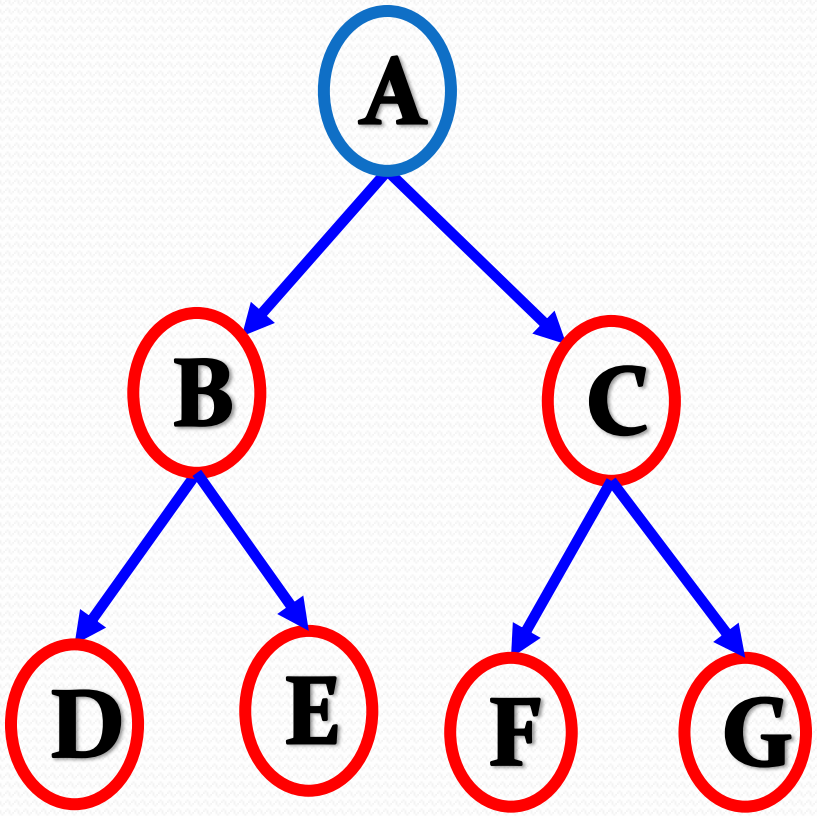
Tree Traversal : Post order



- Post order (L R N)

  D, E, B, F, G

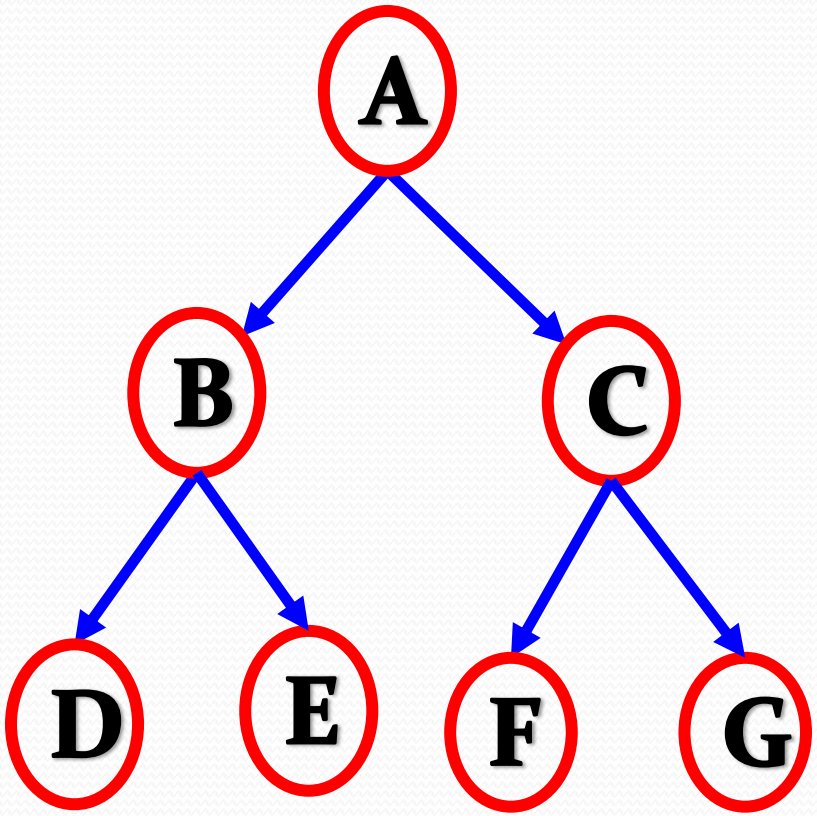
Tree Traversal : Post order



- Post order (L R N)

  D, E, B, F, G, C

Tree Traversal : Post order



- Post order (L R N)

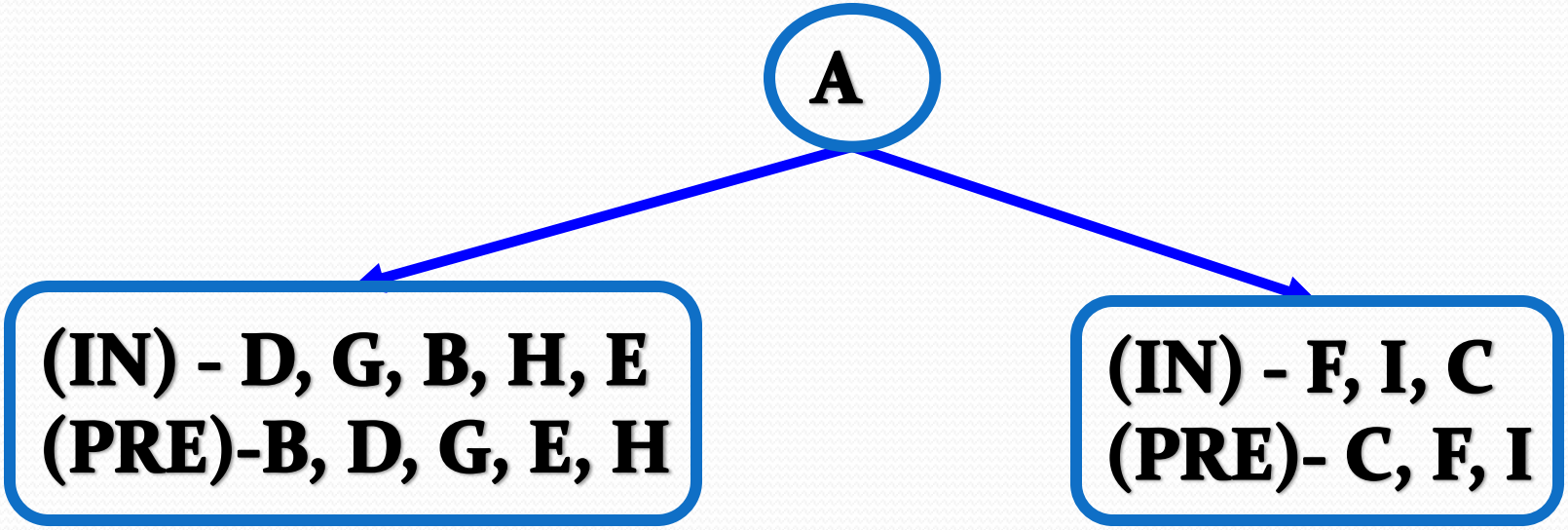
  D, E, B, F, G, C, A

Constructing Binary Tree

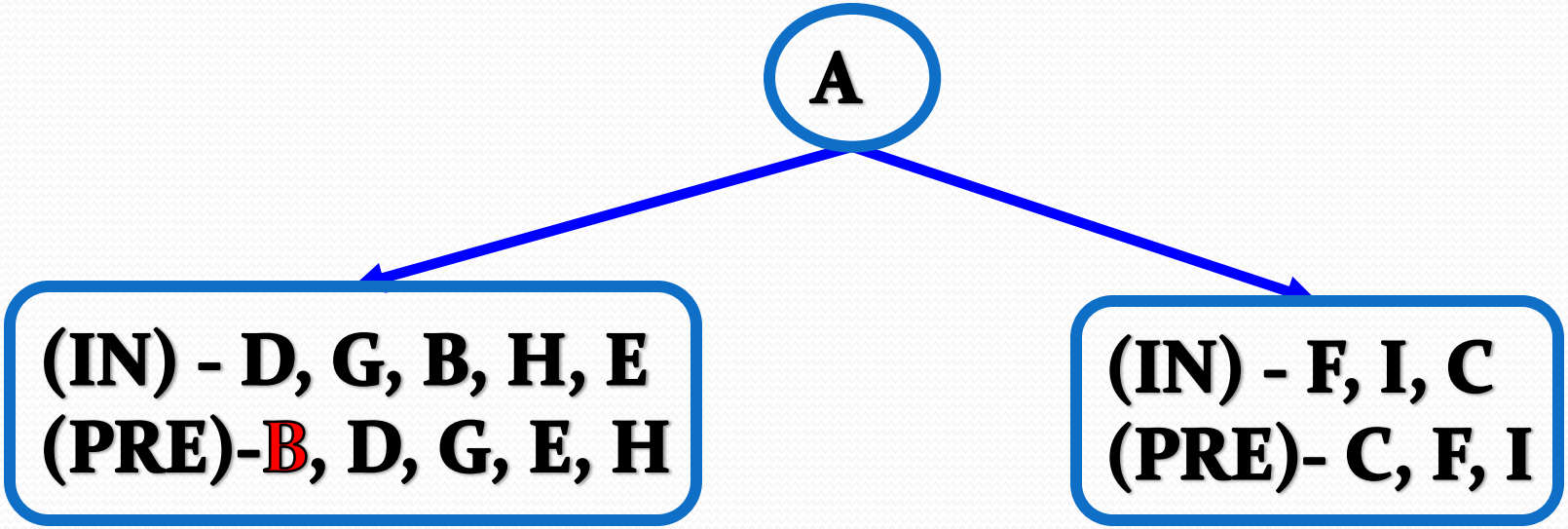
- In order – D, G, B, H, E, A, F, I, C
- Pre order – A, B, D, G, E, H, C, F, I

- 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

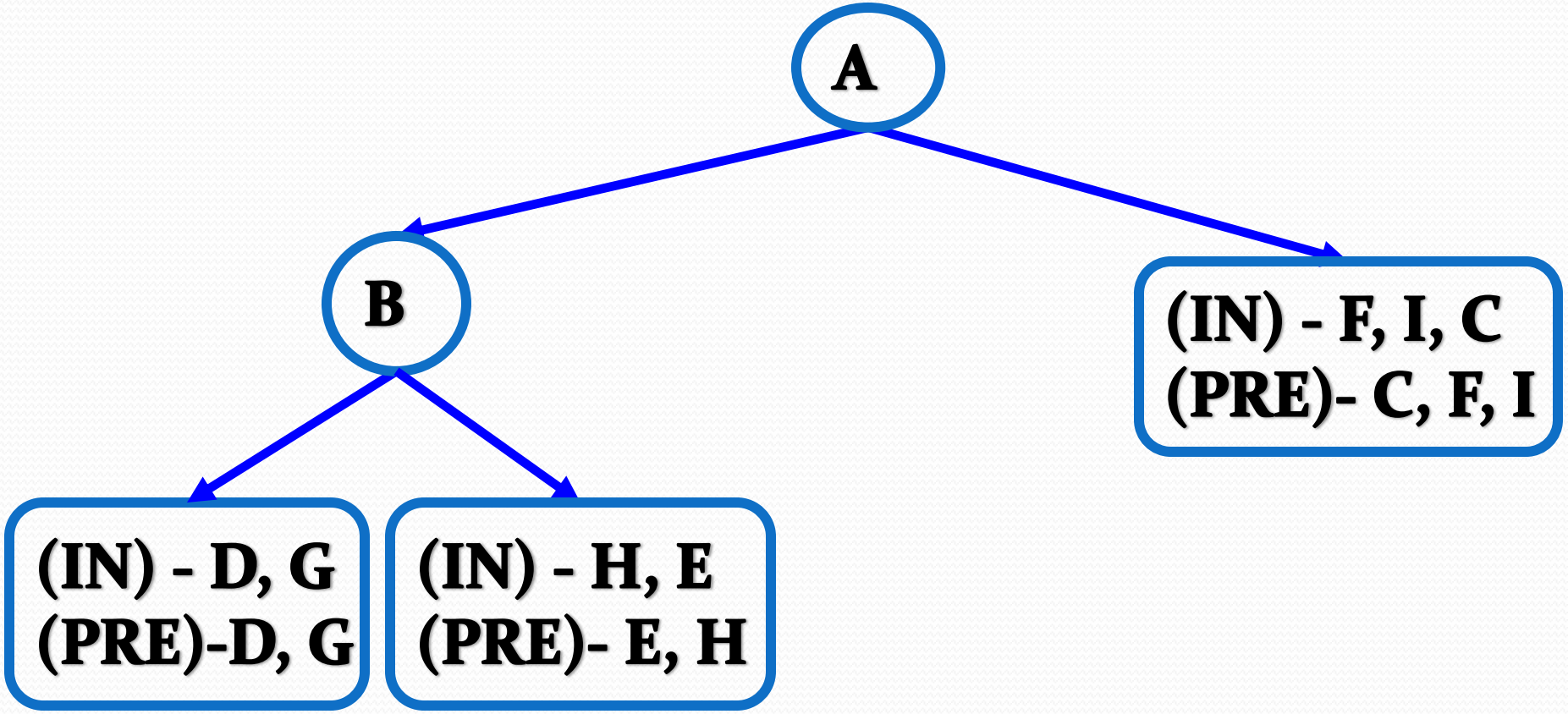
Constructing Binary Tree



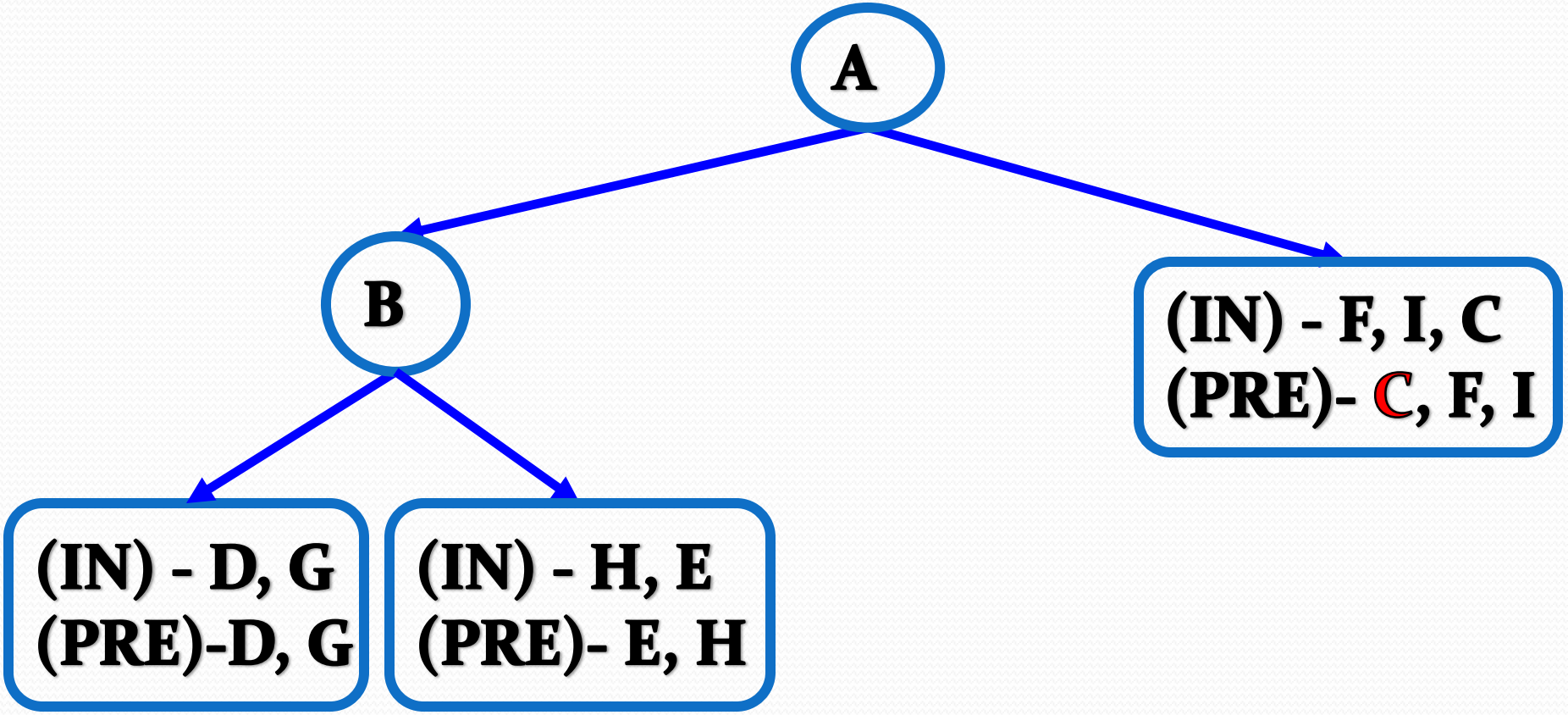
Constructing Binary Tree



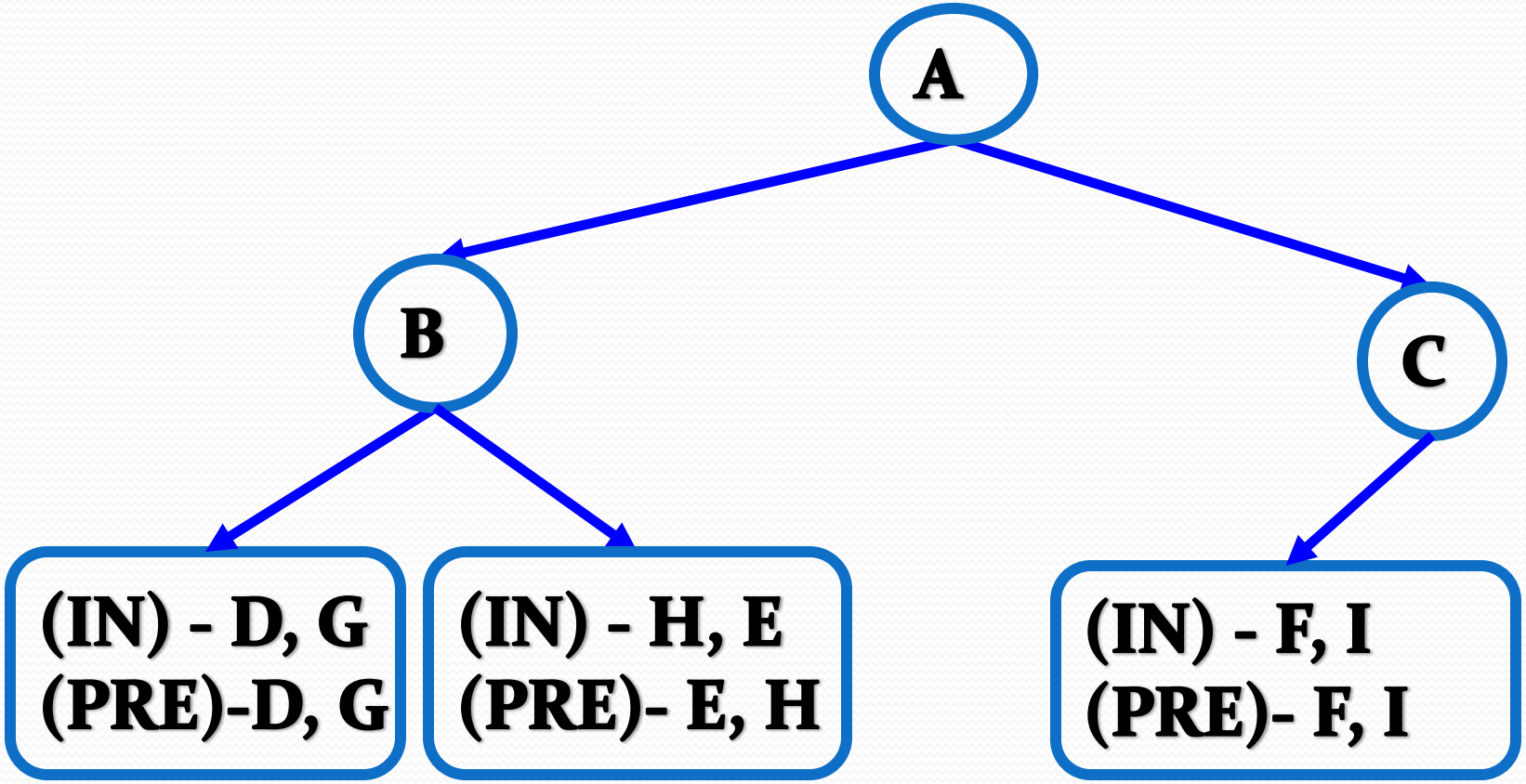
Constructing Binary Tree



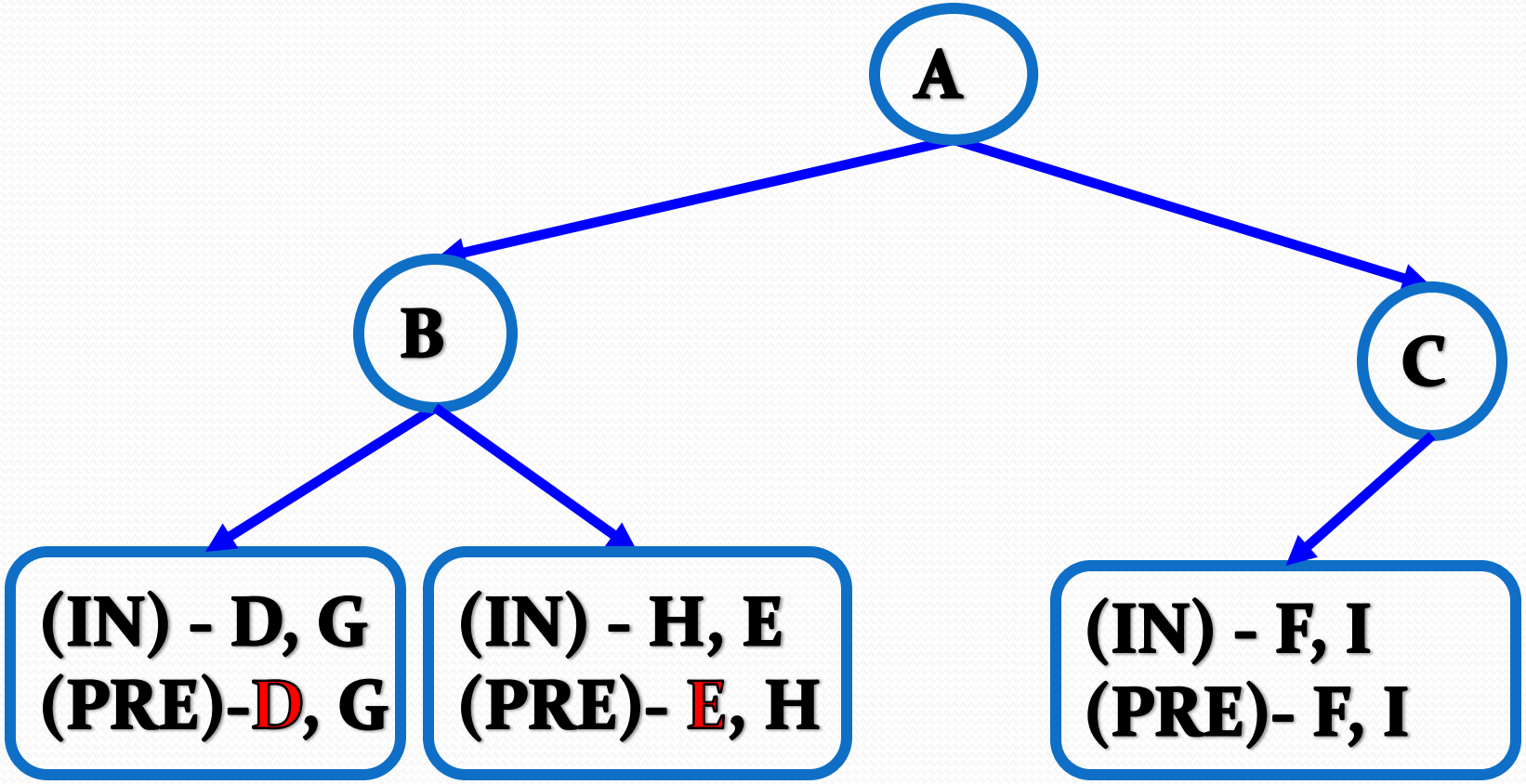
Constructing Binary Tree



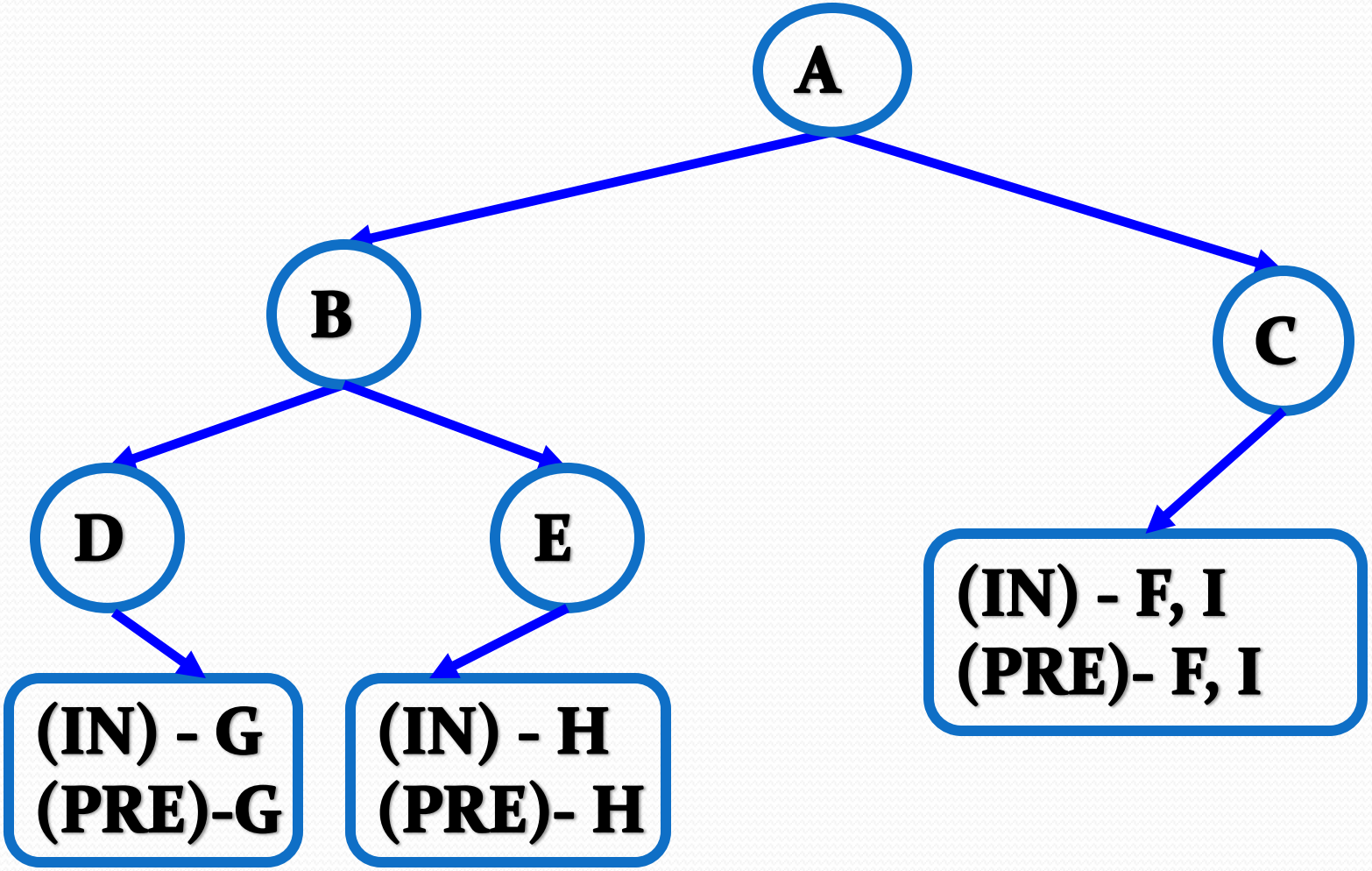
Constructing Binary Tree



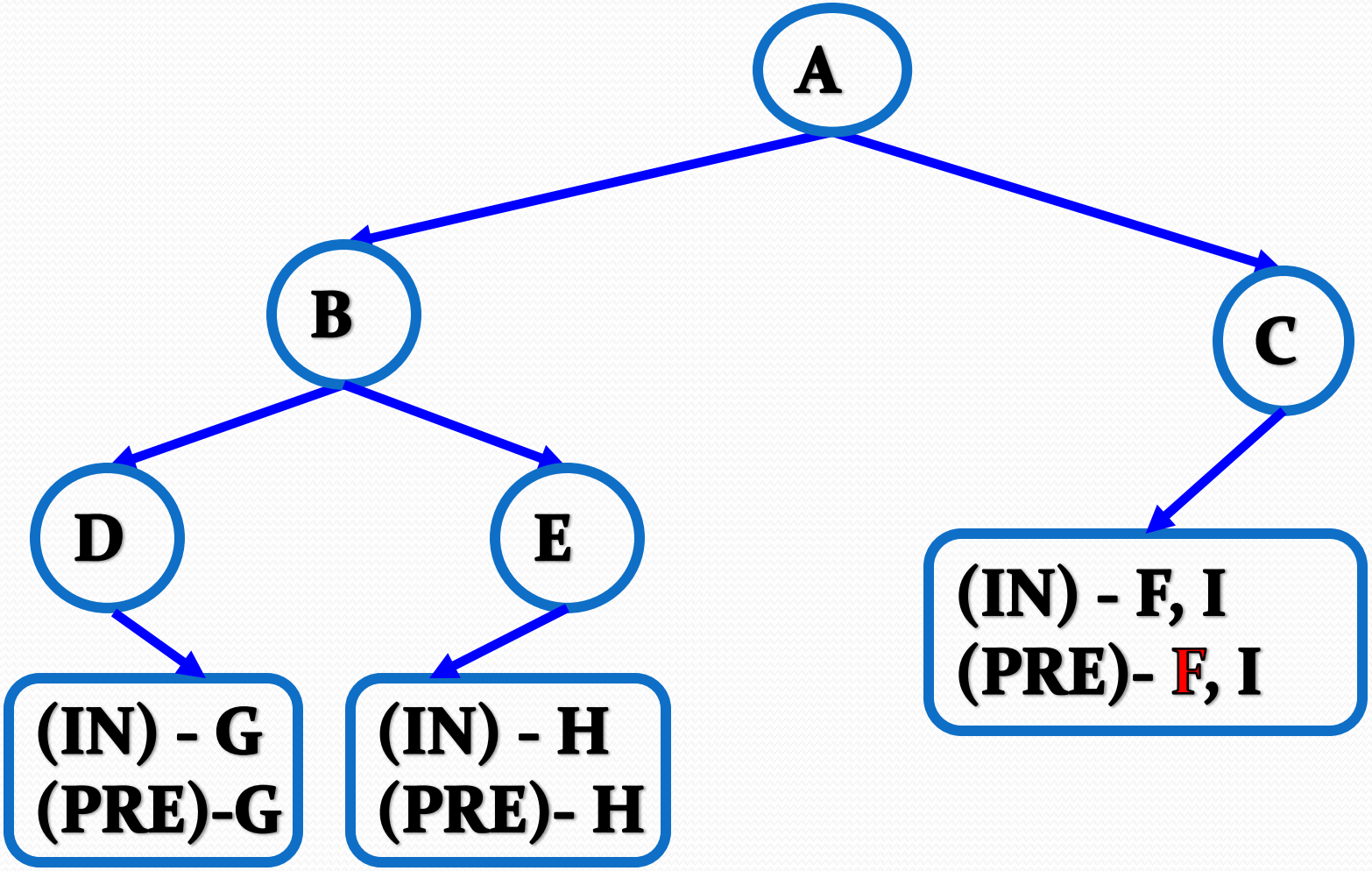
Constructing Binary Tree



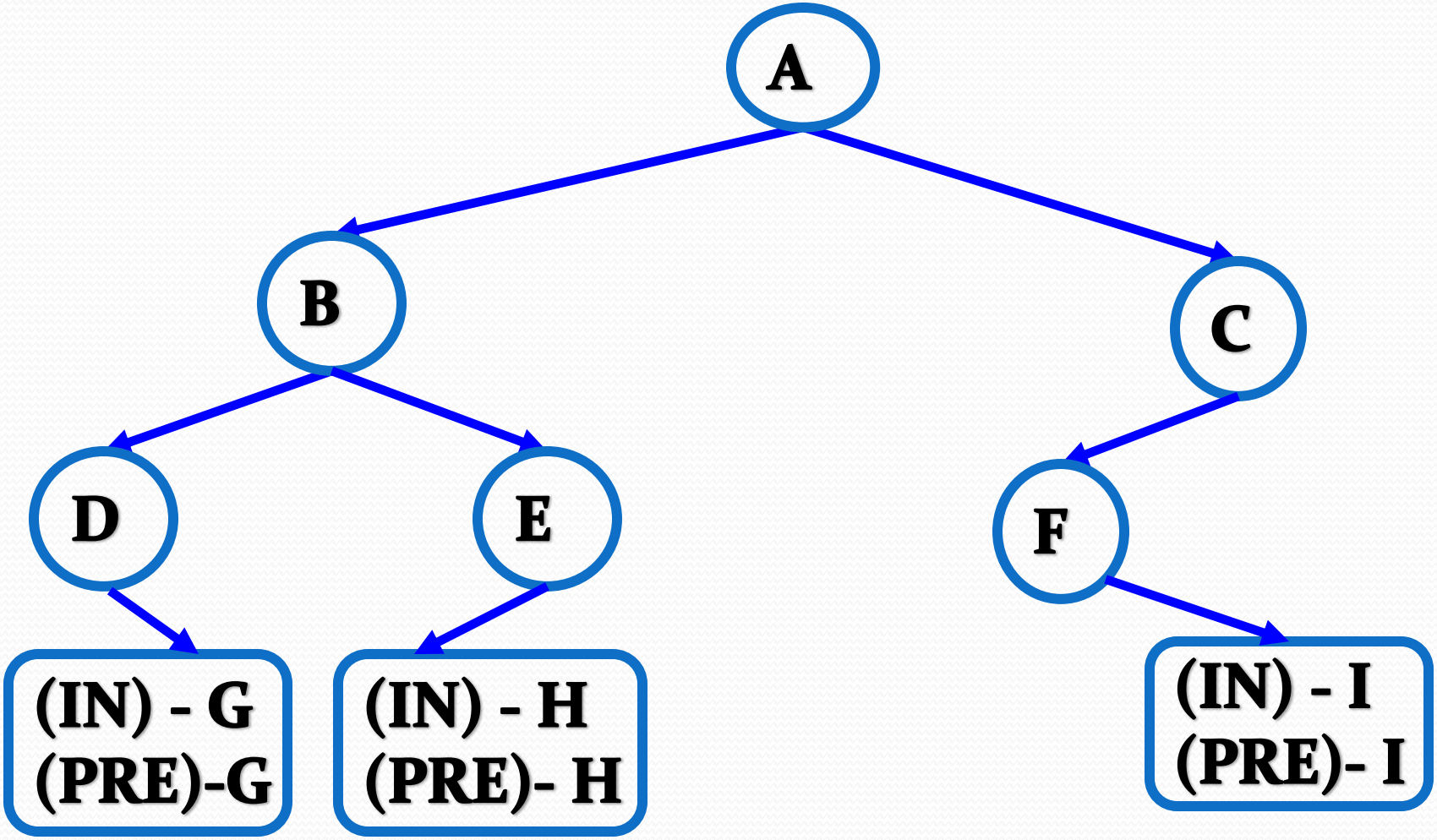
Constructing Binary Tree



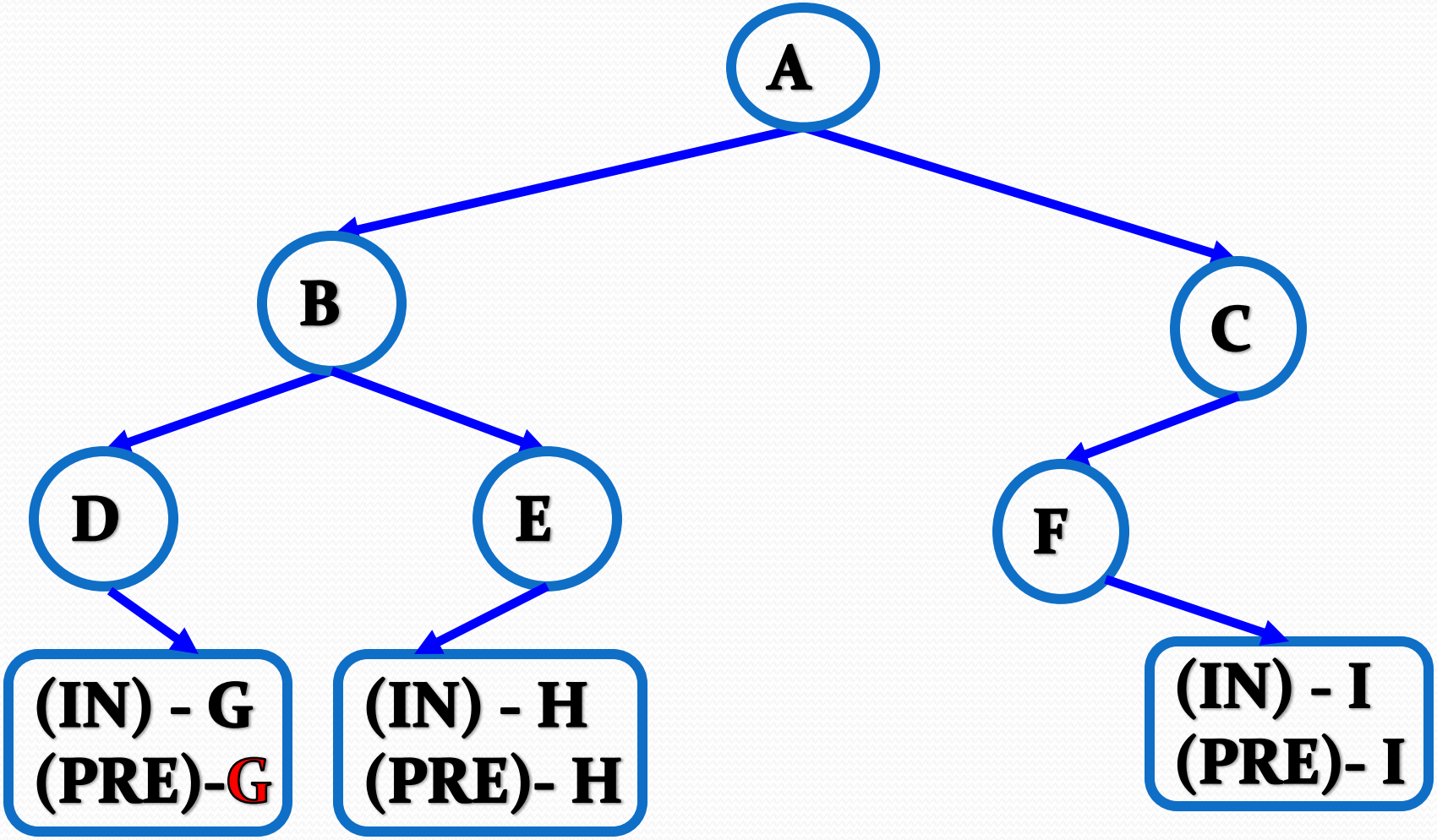
Constructing Binary Tree



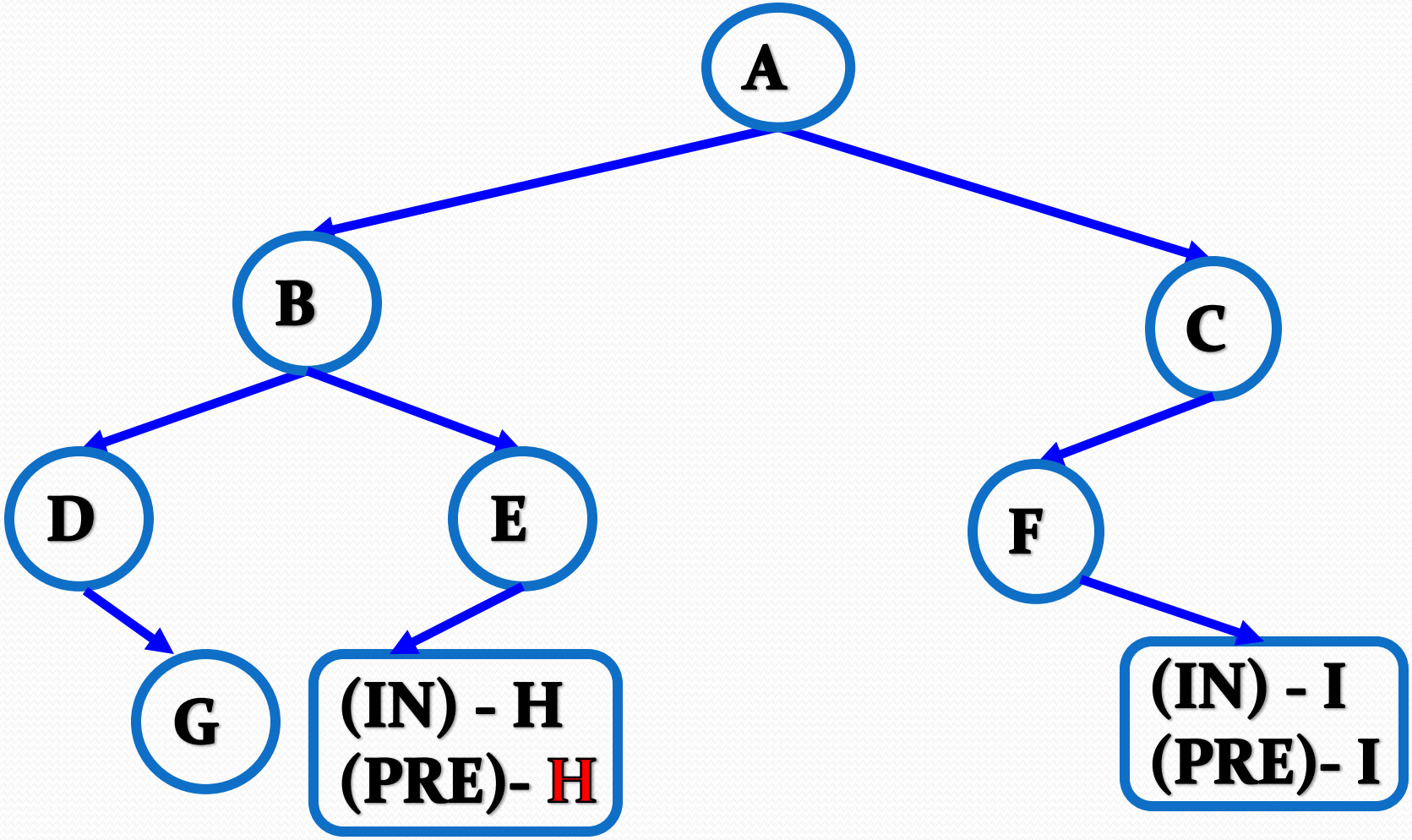
Constructing Binary Tree



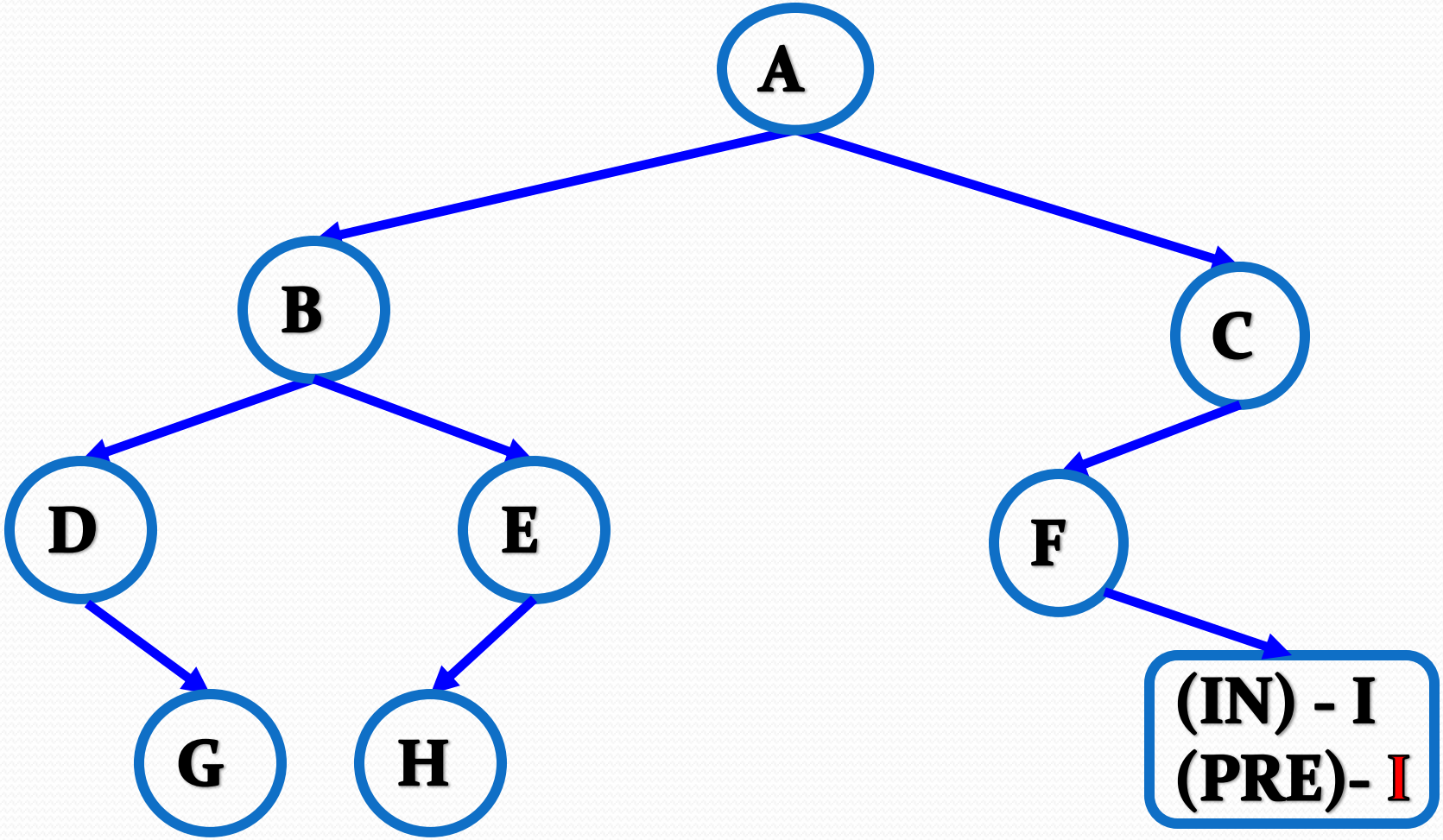
Constructing Binary Tree



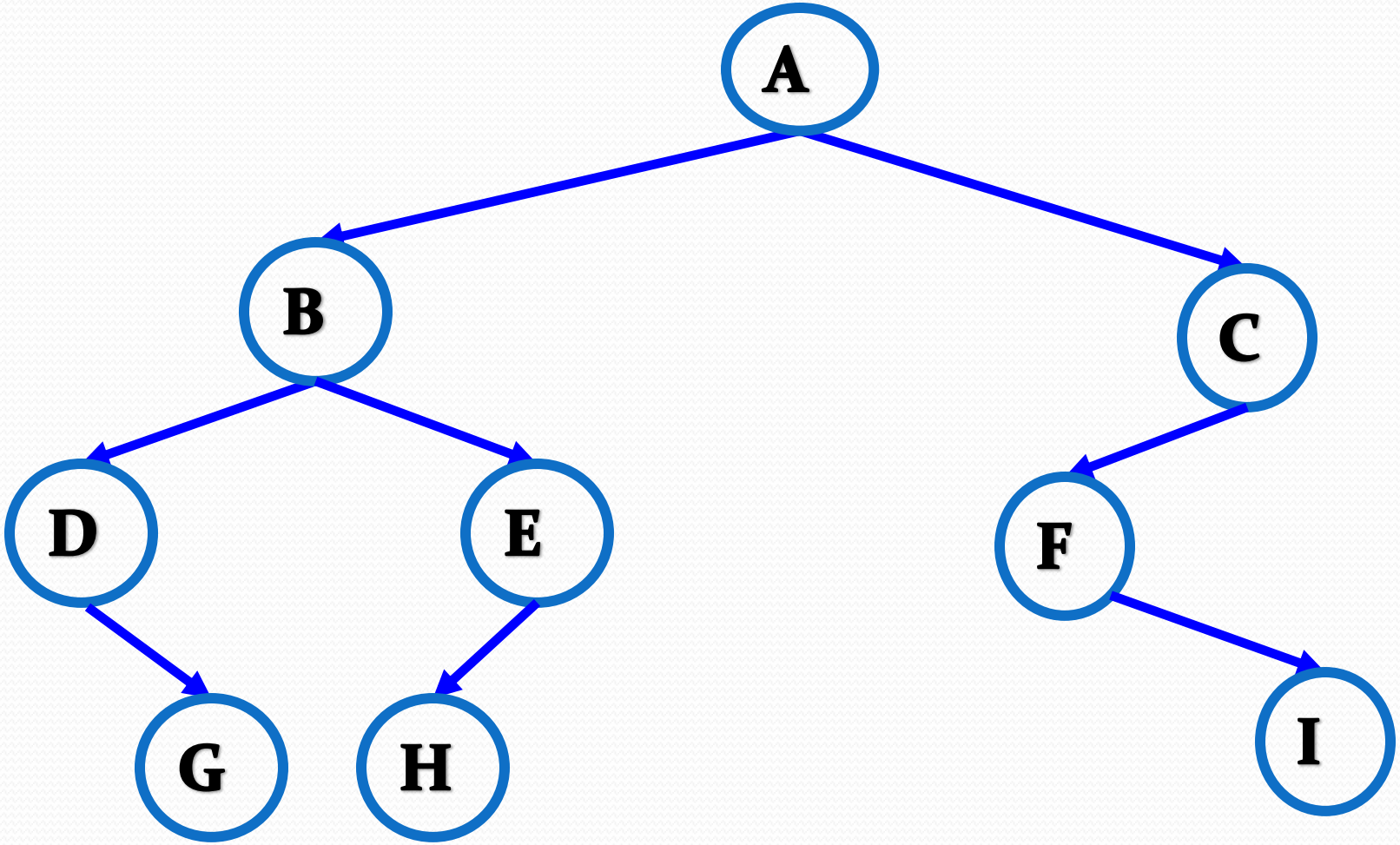
Constructing Binary Tree



Constructing Binary Tree



Constructing Binary Tree



Binary Search Tree

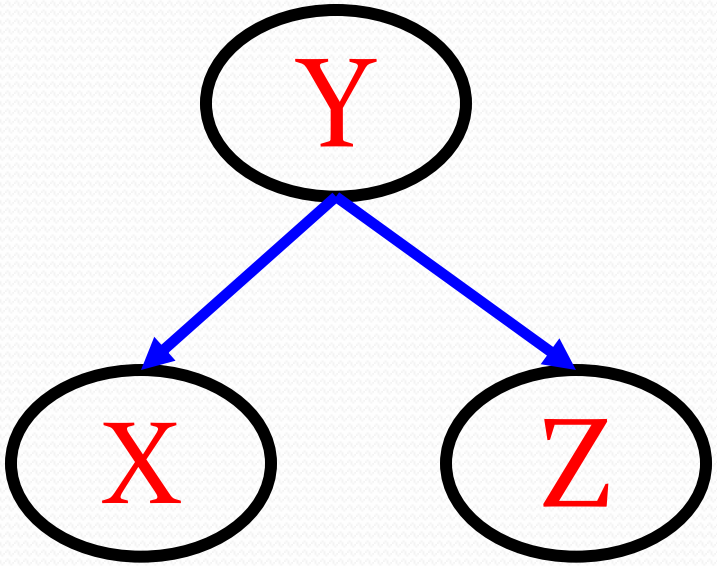
Binary Search Tree

The value at any node,

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

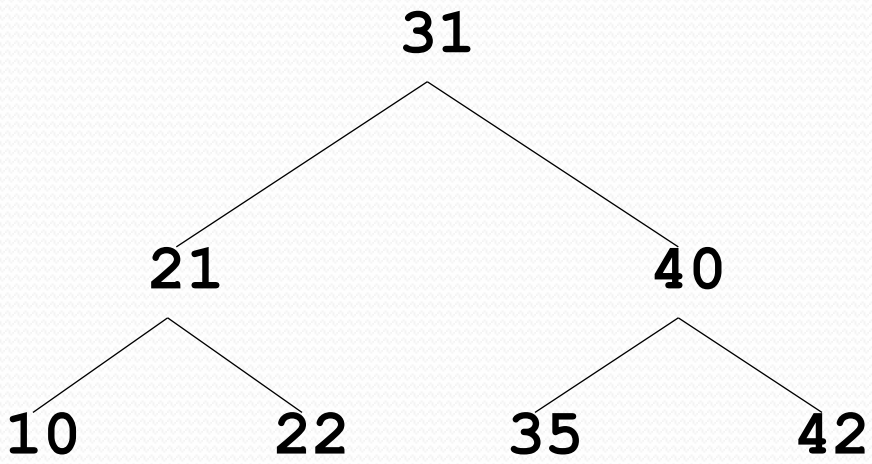
– Example

- Y > X
- Y < Z



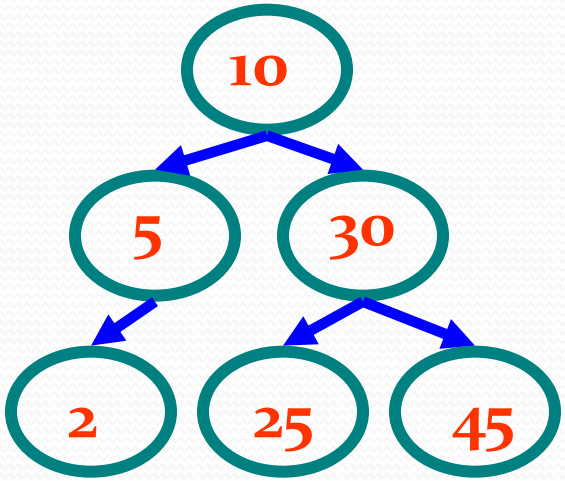
Binary Search Tree

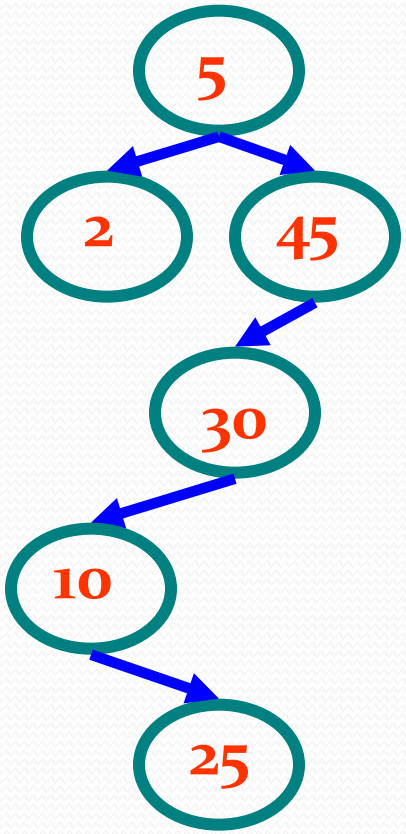
- Values in left sub tree less than parent
- Values in right sub tree greater than parent
- Fast searches in a Binary Search tree, maximum of log n comparisons



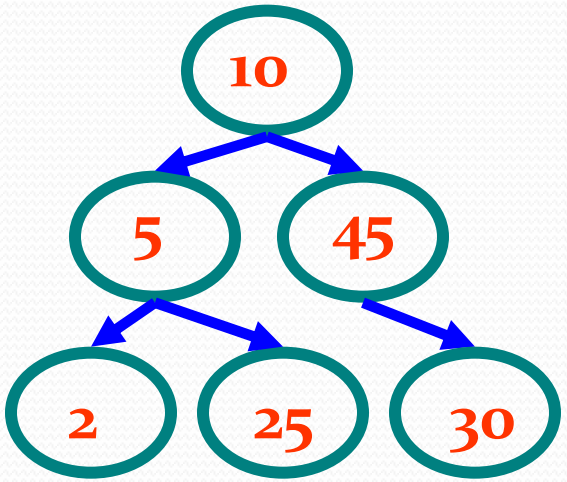
Binary Search Trees

- Examples





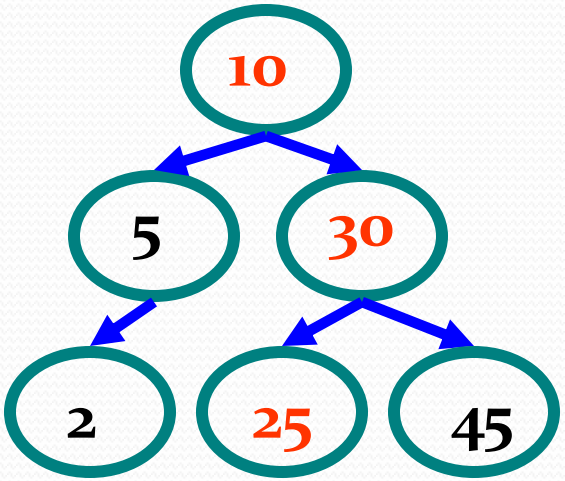
Binary search trees



Not a binary search tree

Example Binary Searches

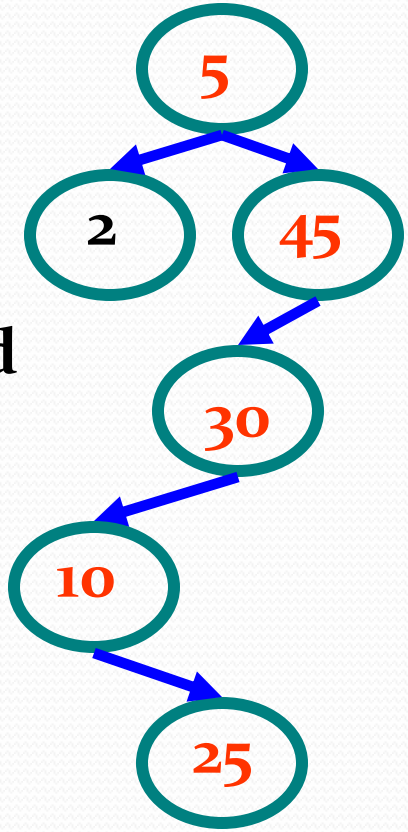
- search (root, 25)



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

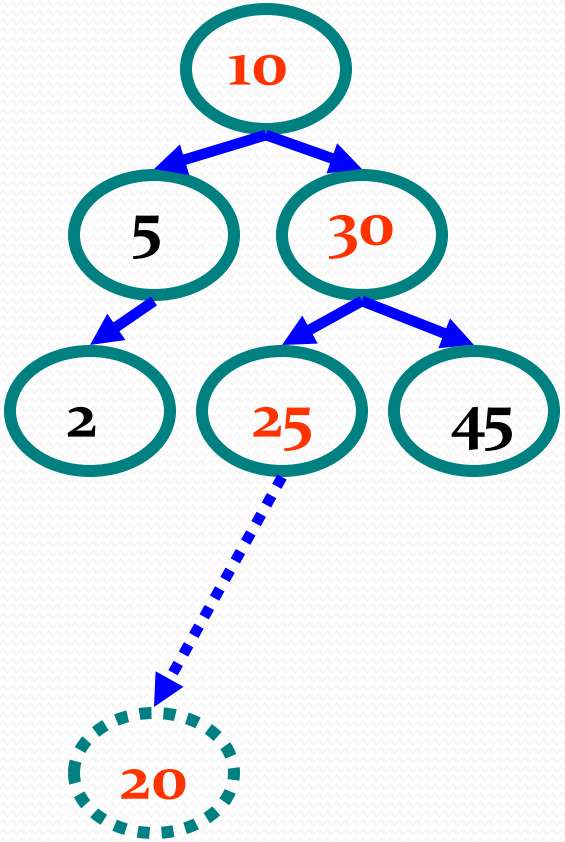
```
Search(node *n, info)
{
    if (n == NULL)                  // Not found
        return( n );
    else if (n->data == item)       // Found it
        return( n );
    else if (n->data > item)        // In left subtree
        return search( n->left, item );
    else                            //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

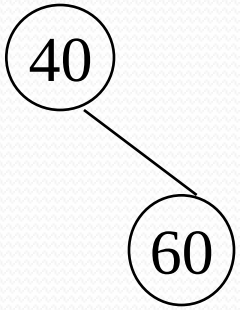
Insert 20 on left

Insertion in a Binary Tree

40, 60, 50, 33, 55, 11

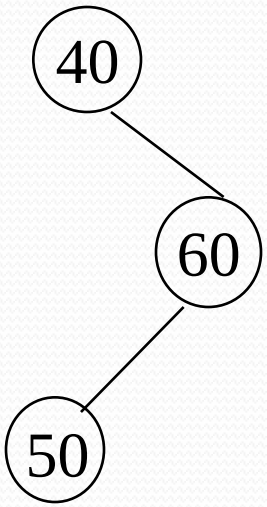
Insertion in a Binary Tree

40, 60, 50, 33, 55, 11



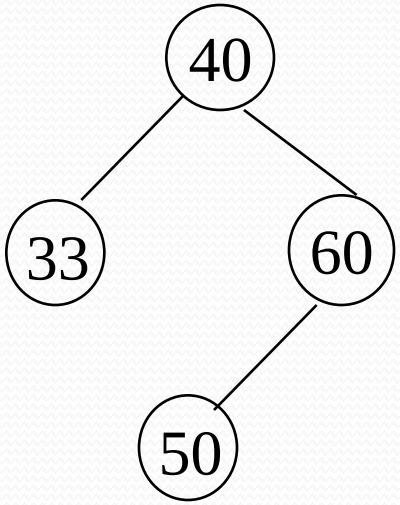
Insertion in a Binary Tree

40, 60, 50, 33, 55, 11



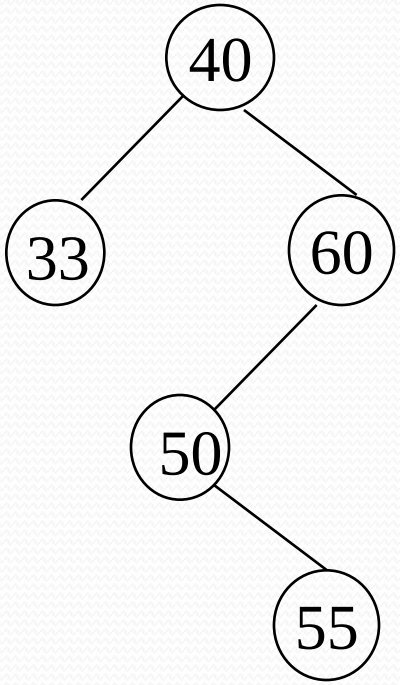
Insertion in a Binary Tree

40, 60, 50, 33, 55, 11



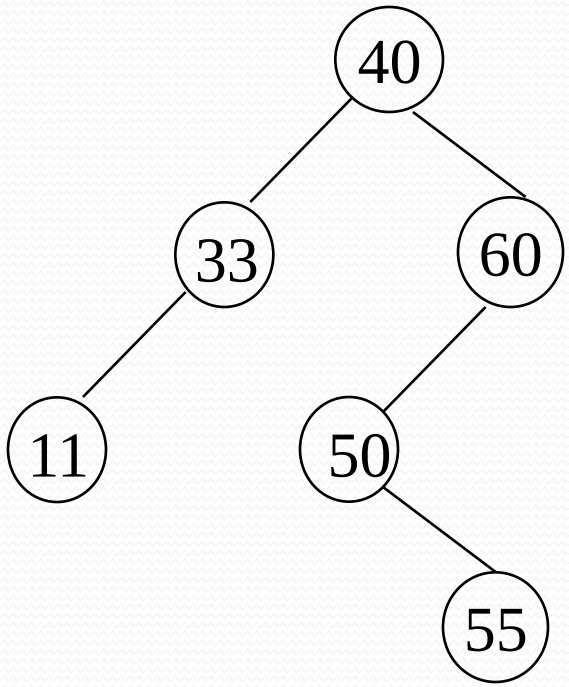
Insertion in a Binary Tree

40, 60, 50, 33, 55, 11



Insertion in a Binary Tree

40, 60, 50, 33, 55, 11



Deletion in Binary Tree

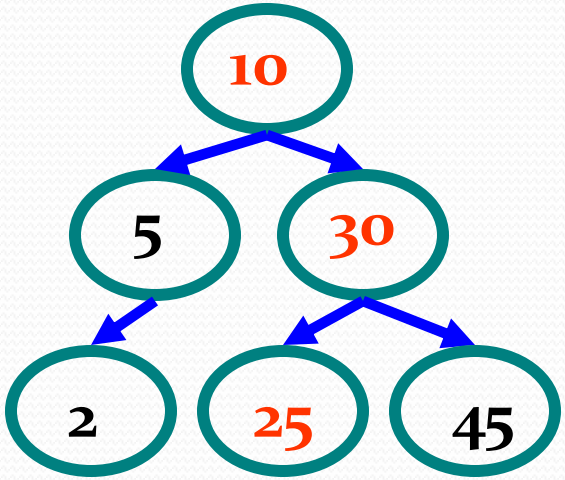
- **Algorithm**
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else //we must delete internal node
     a) Replace with largest value Y on left subtree
        OR smallest value Z on right subtree
     b) Delete replacement value (Y or Z) from subtree

Note :-

- Deletions may unbalance tree

Example Deletion (Leaf)

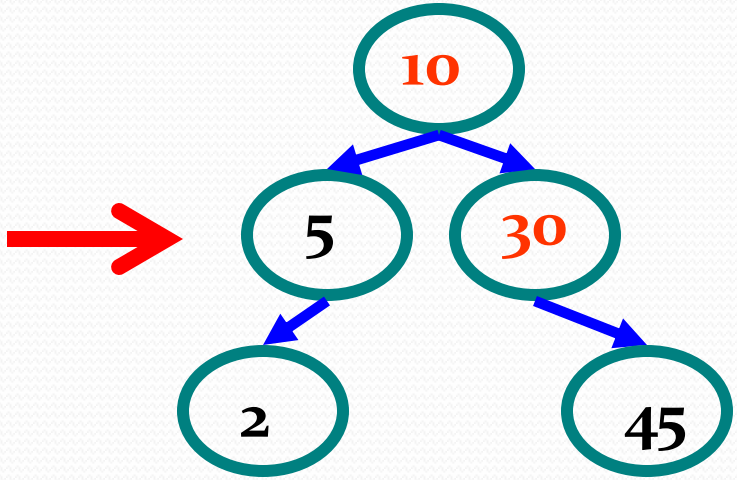
- Delete (25)



10 < 25, right

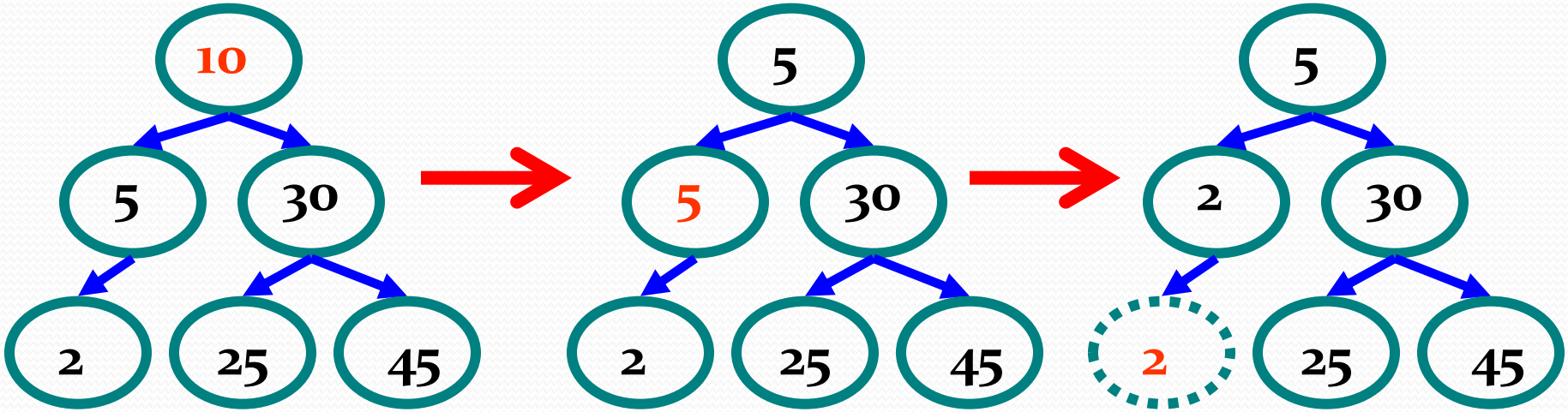
30 > 25, left

25 = 25, delete



Example Deletion (Internal Node)

- Delete (10)



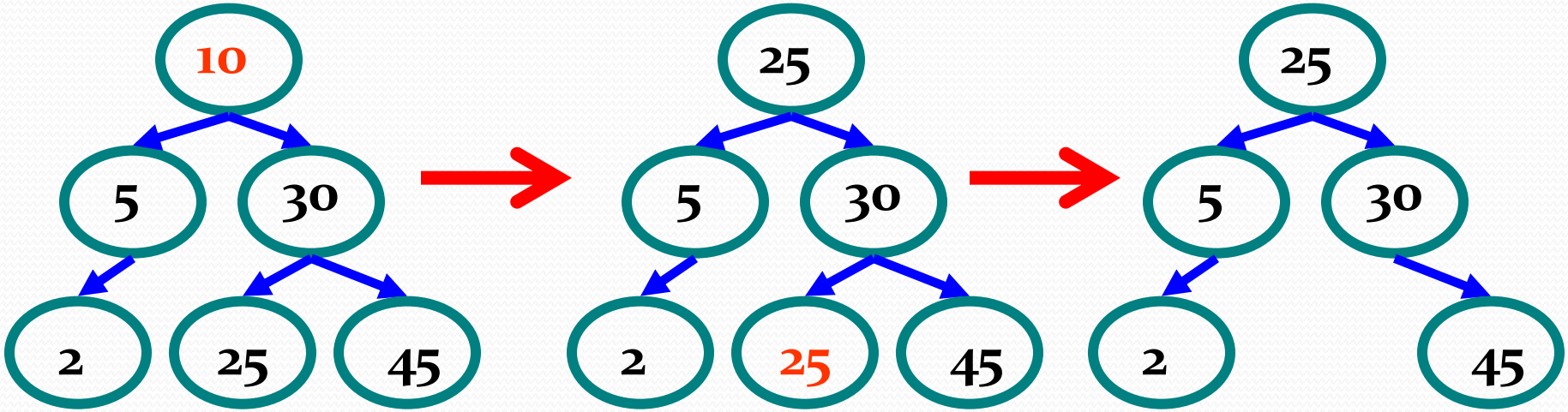
Replacing 10 with largest value in left subtree

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=

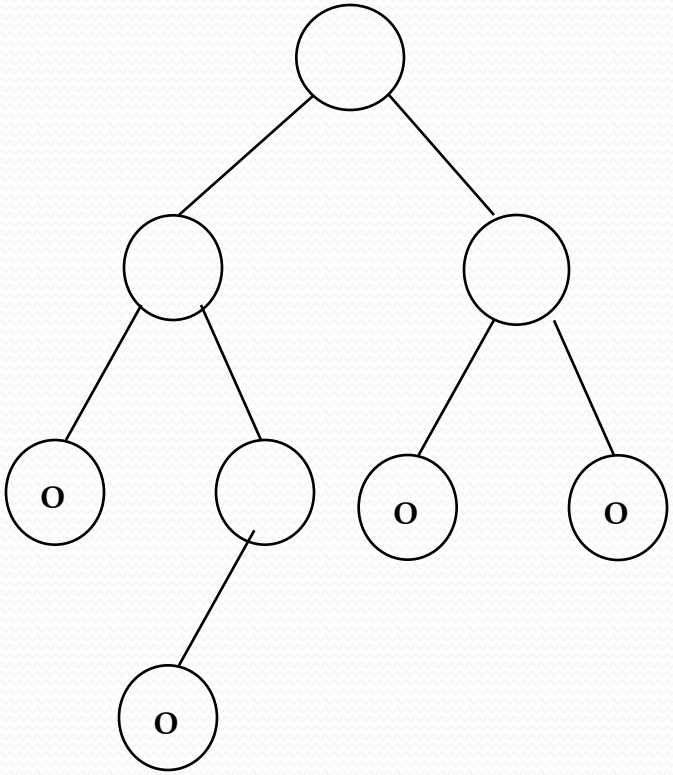
  height(left sub tree) - height(right sub tree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right sub tree can differ by no more than 1
  - Store current heights in each node

AVL Tree

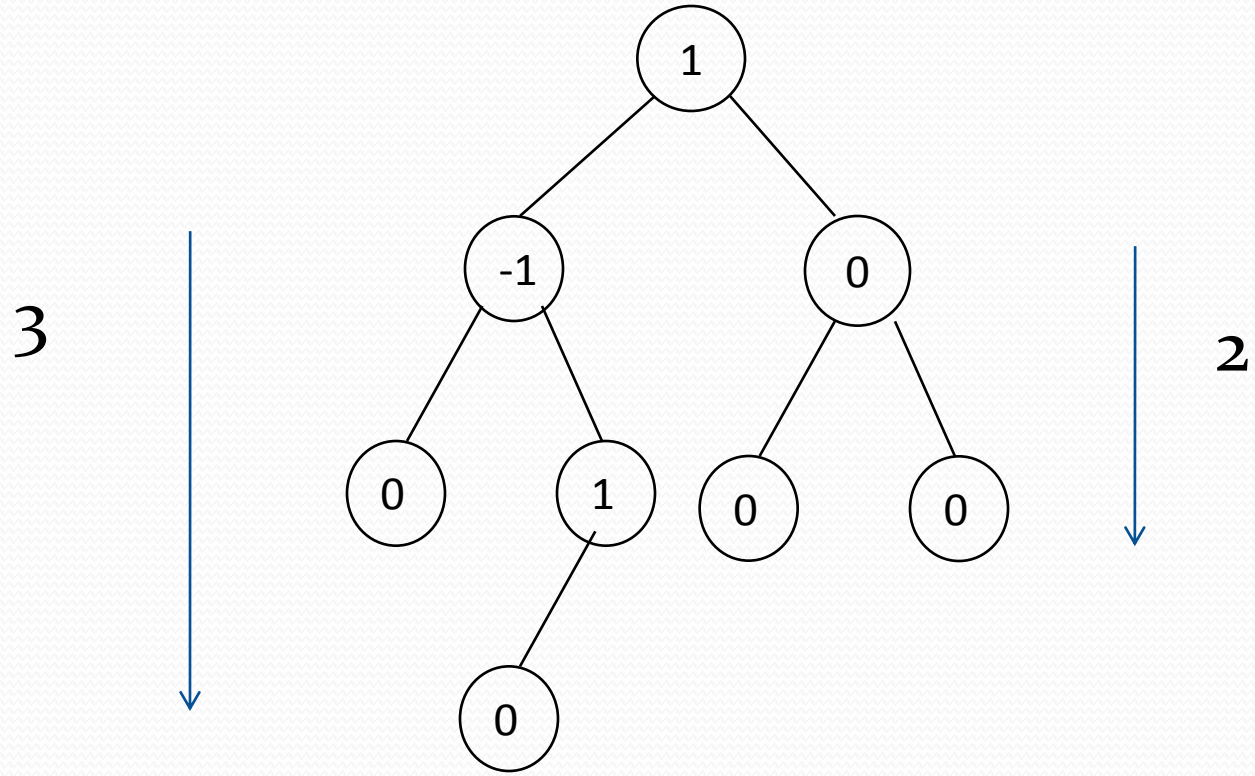
- A binary tree in which the difference of height of the right and left subtree of any node is less than or equal to 1 is known as AVL Tree.
- Height of left subtree – height of right subtree

  can be either -1,0,1

AVL Tree

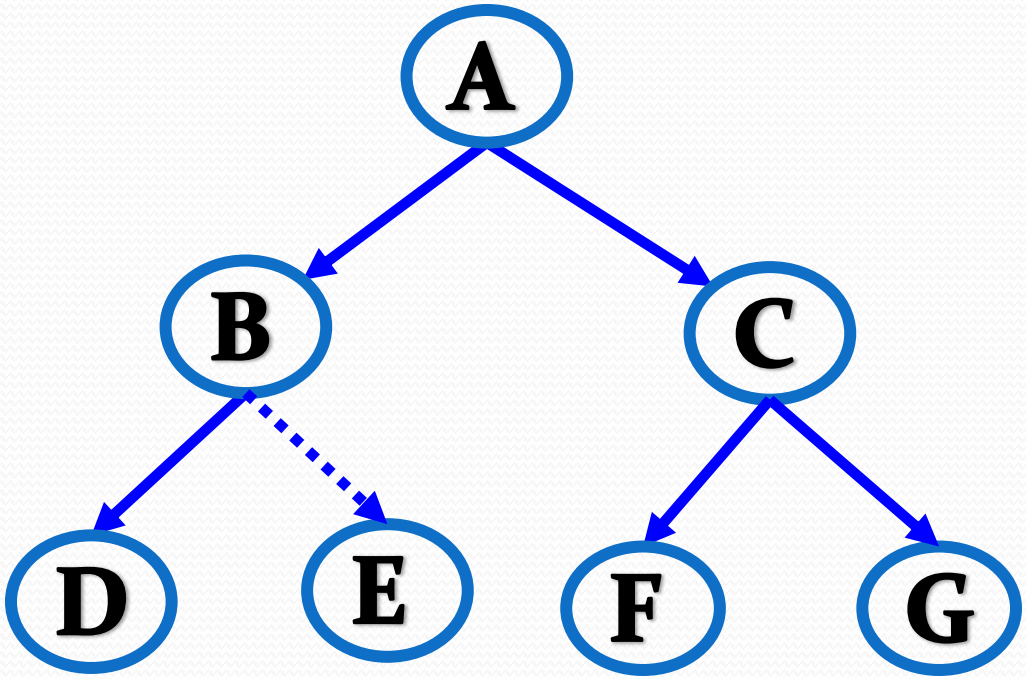


AVL Tree



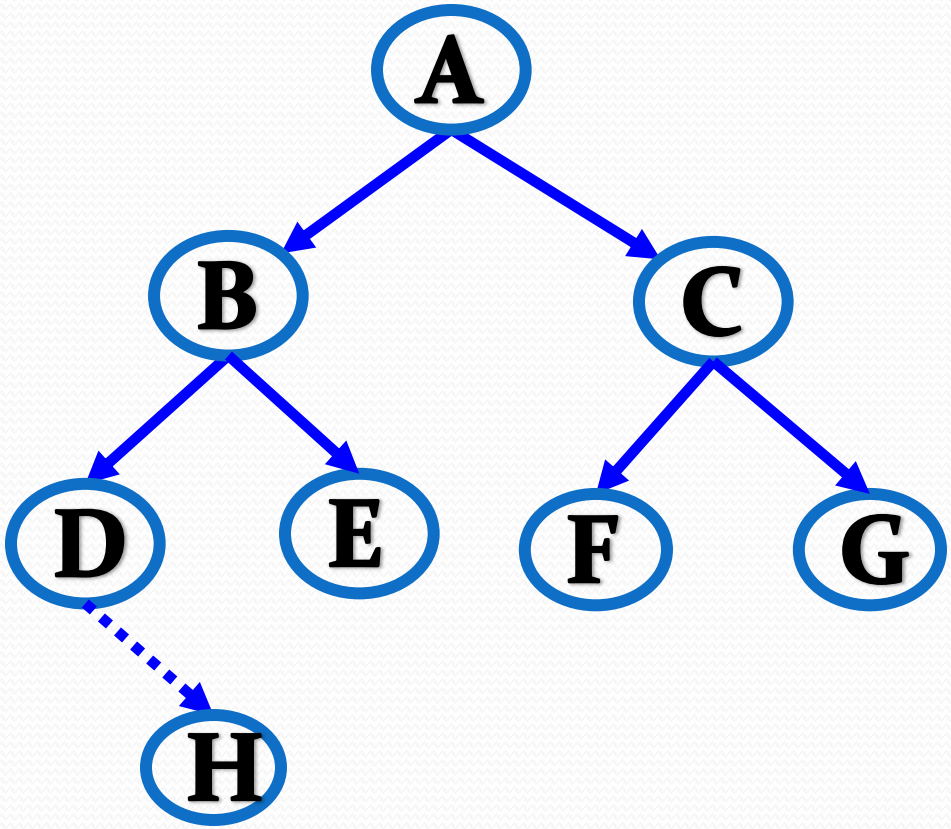
Balanced as LST-RST=1

Insertion in AVL Tree



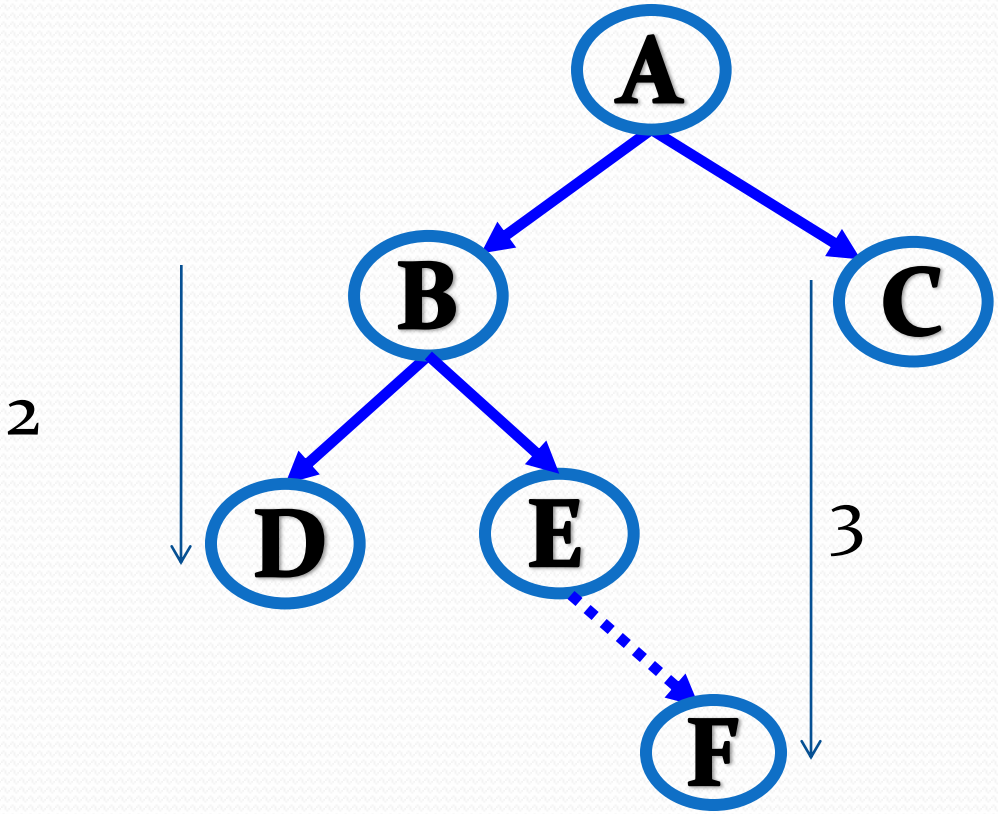
Case 1: the node was either left heavy or right heavy and has become balanced

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

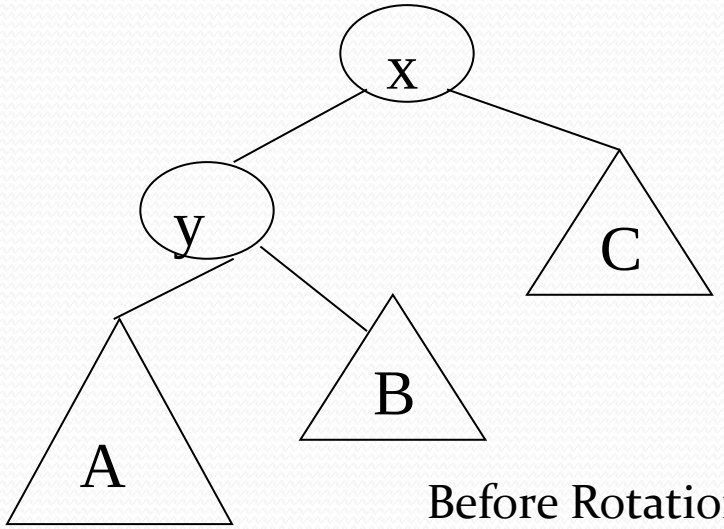
Rebalancing

- 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.

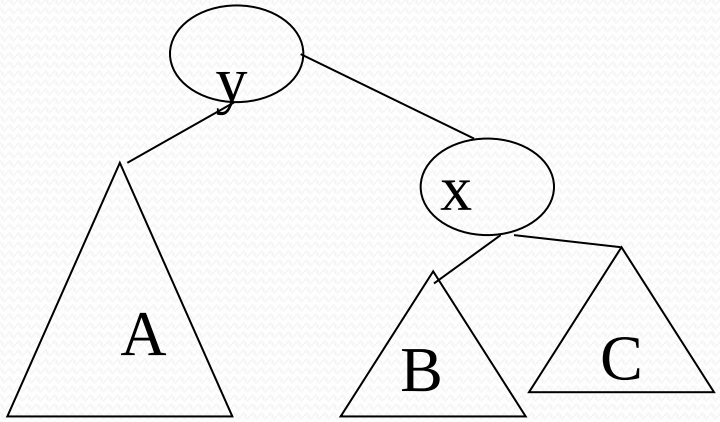
Rotations

- single rotations

e.g. Single Rotation



Before Rotation



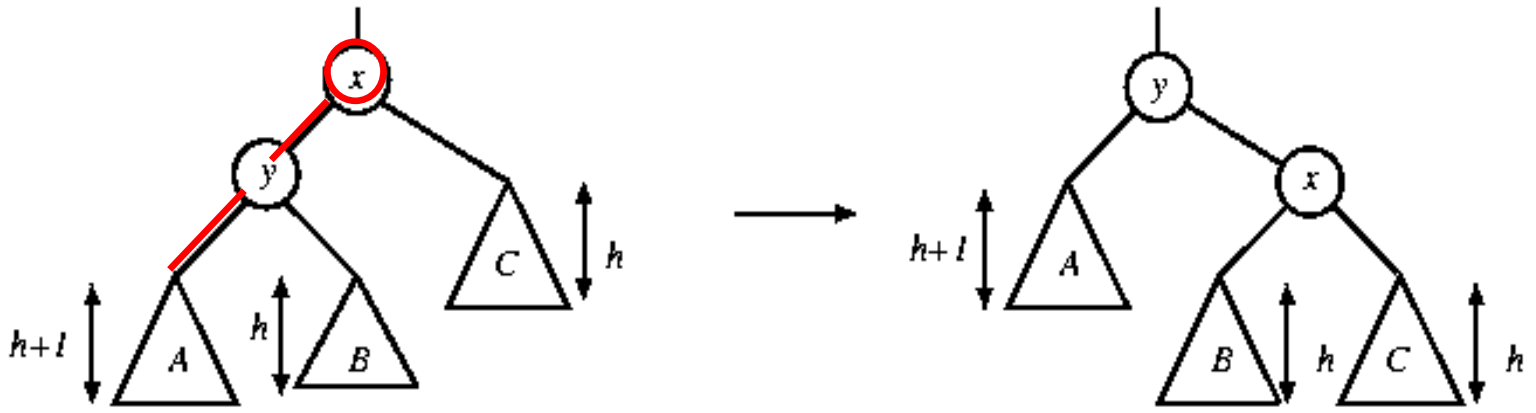
After Rotation

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

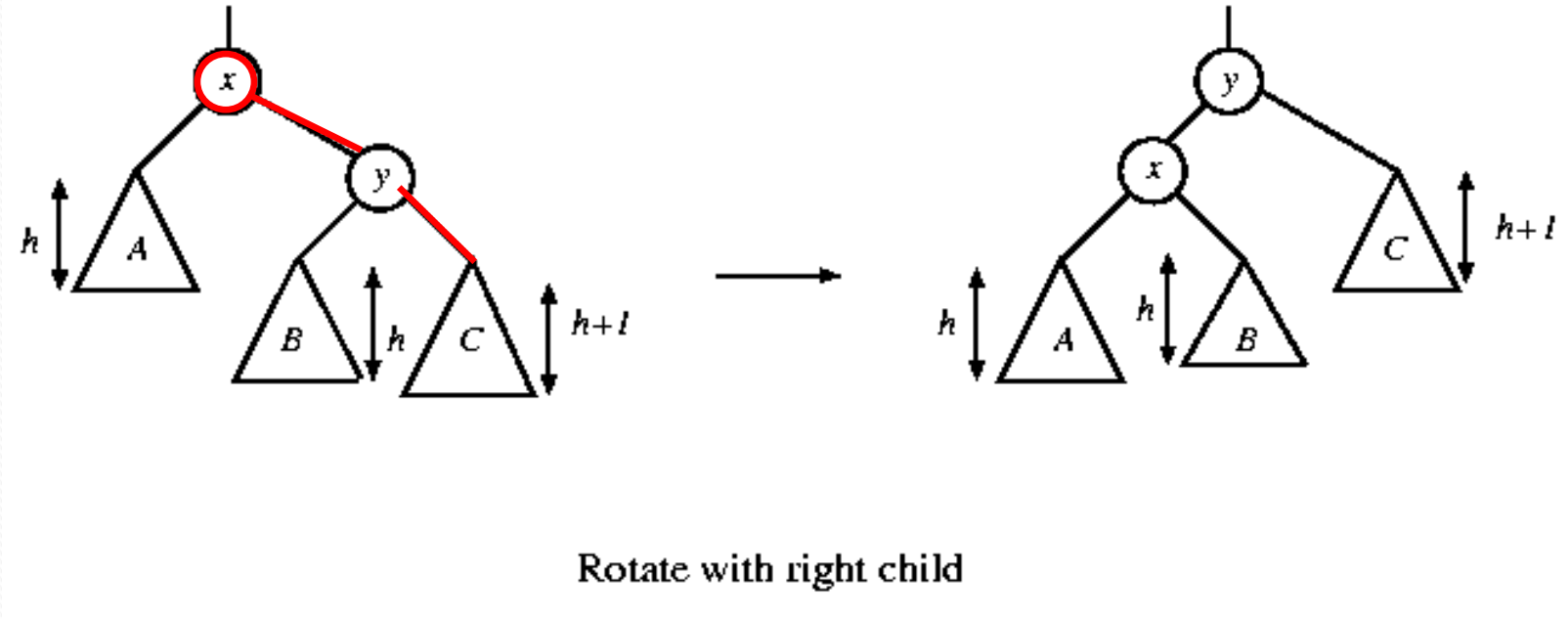
The new item is inserted in the subtree A.
The AVL-property is violated at x
height of left(x) is h+2
height of right(x) is h.



Rotate with left child

Single Rotation

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

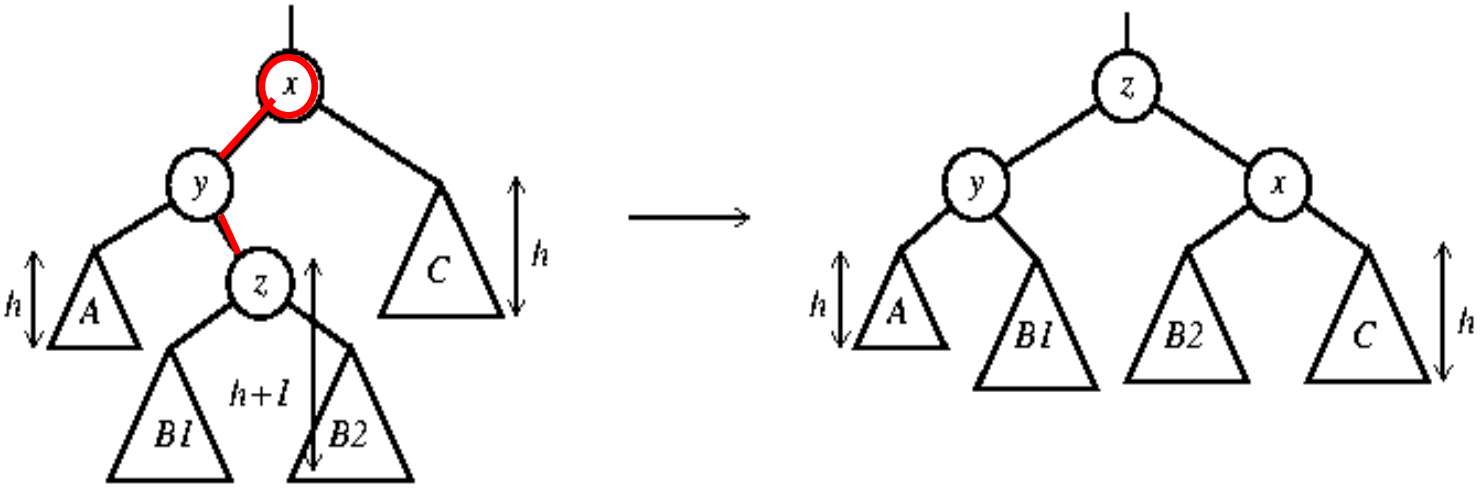


Rotate with right child

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

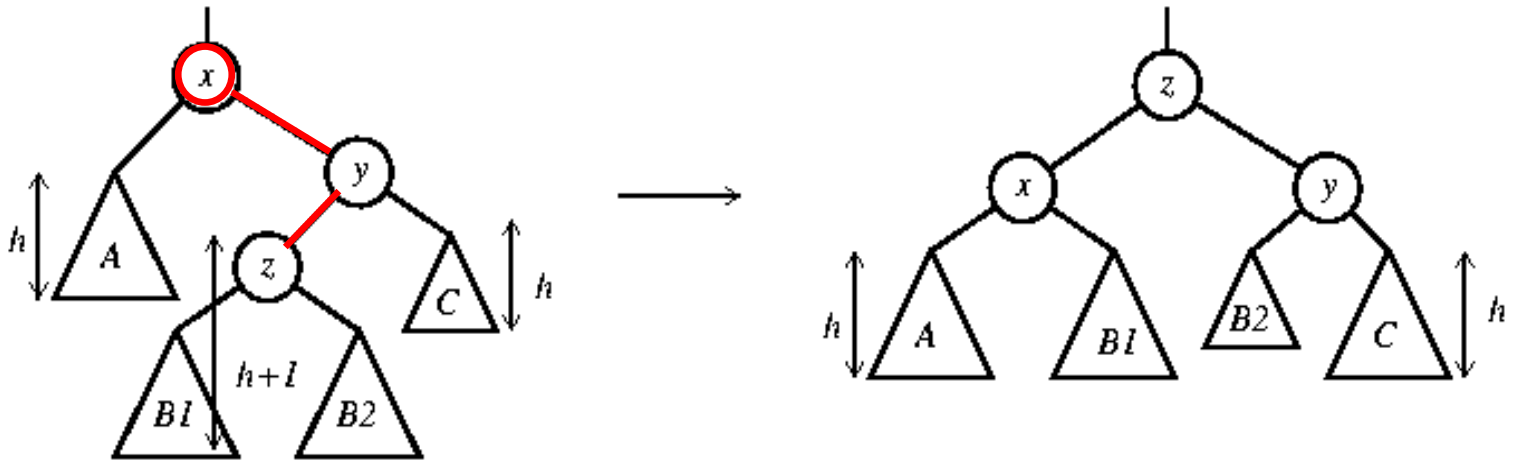


Double rotate with left child

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.

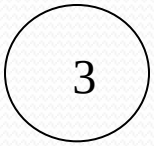


Double rotate with right child

also called right-left rotate

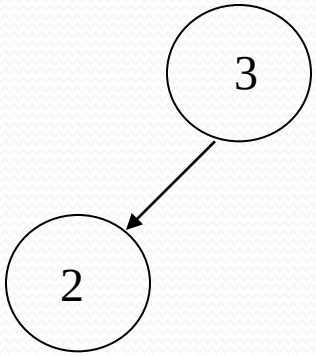
Example

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



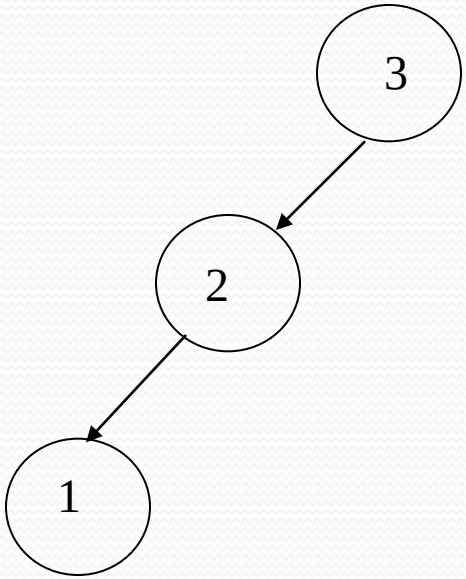
Example

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



Example

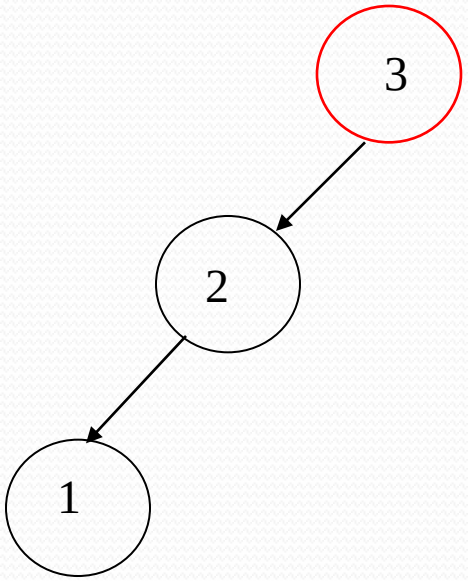
Insert 3,2,1,4,5,6,7



Example

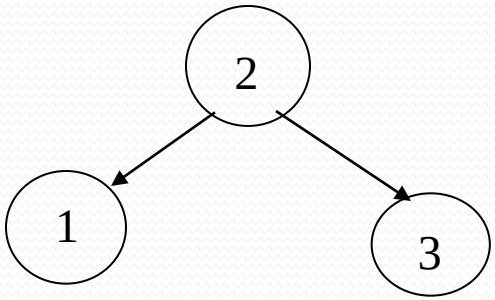
Insert 3,2,1,4,5,6,7

Single rotation



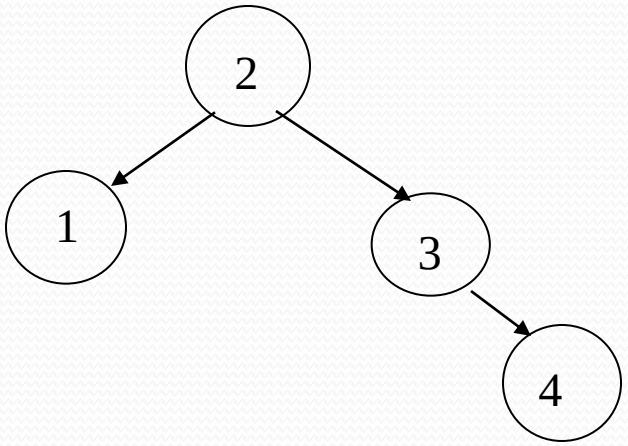
Example

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



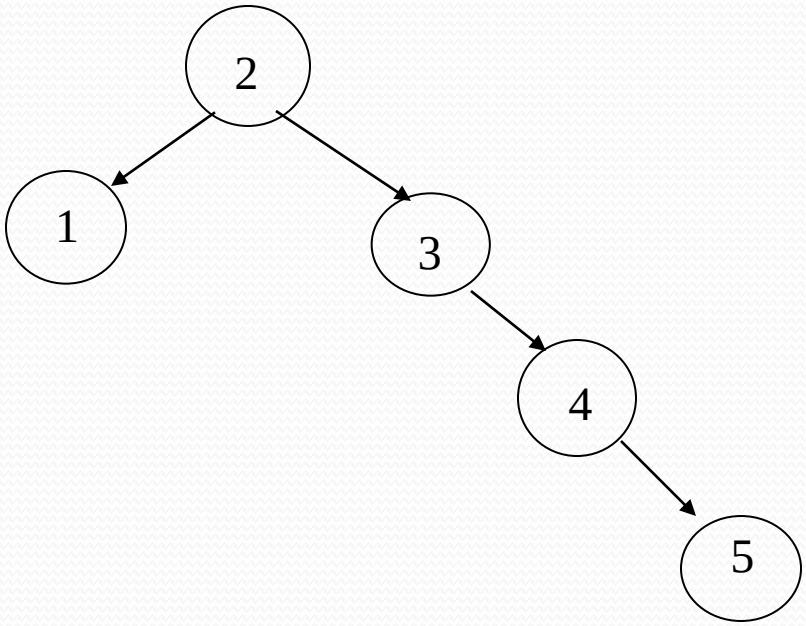
Example

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



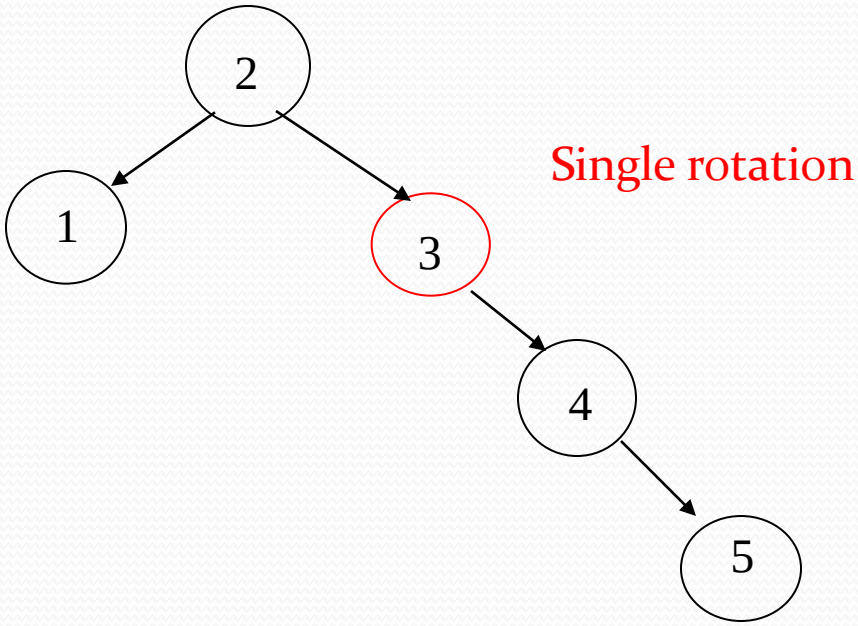
Example

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



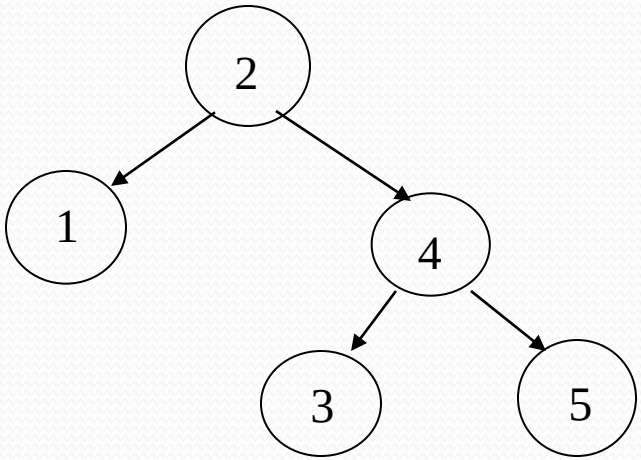
Example

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



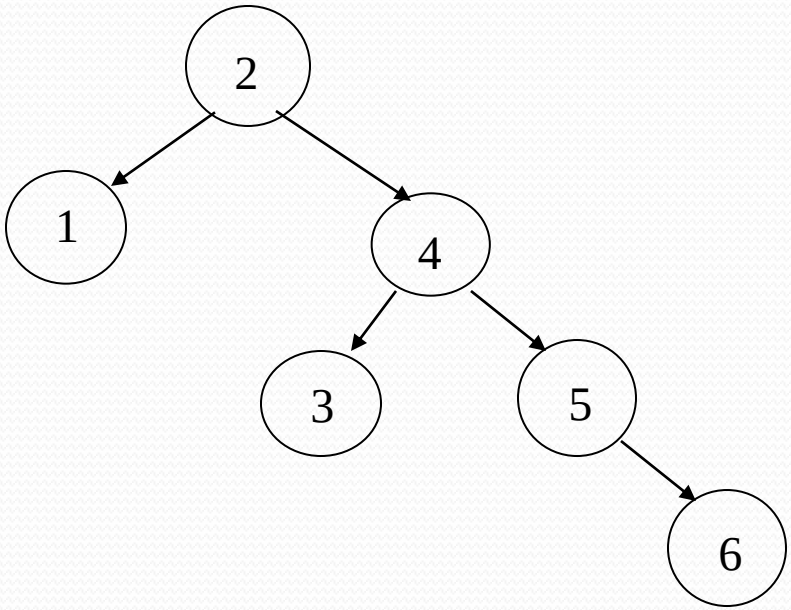
Example

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



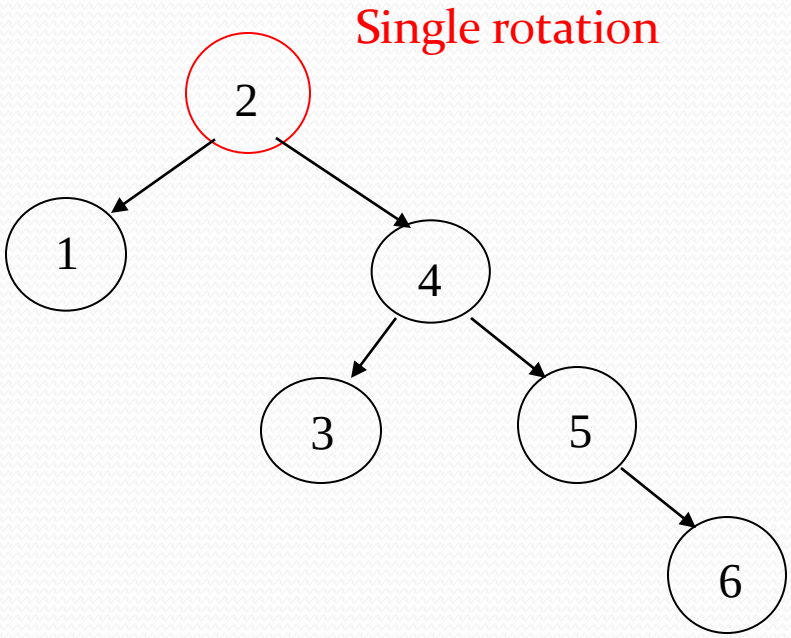
Example

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



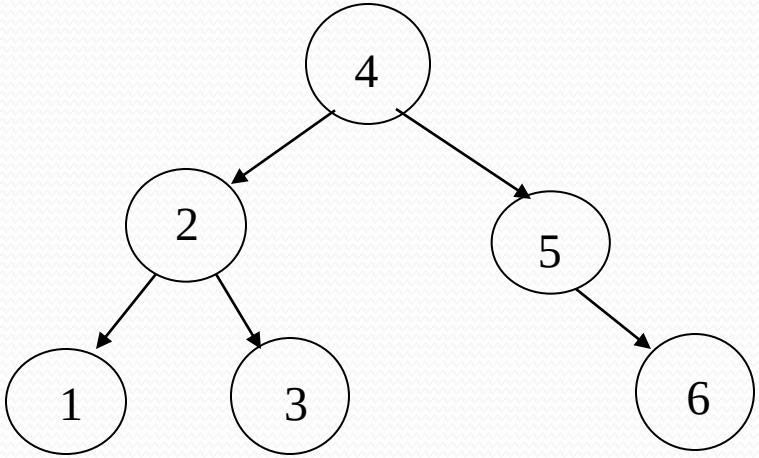
Example

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



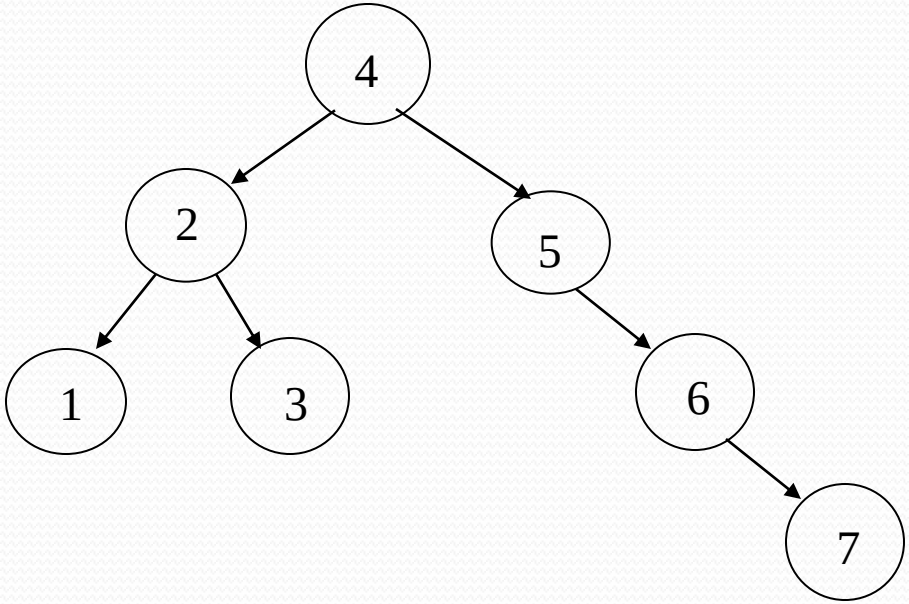
Example

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



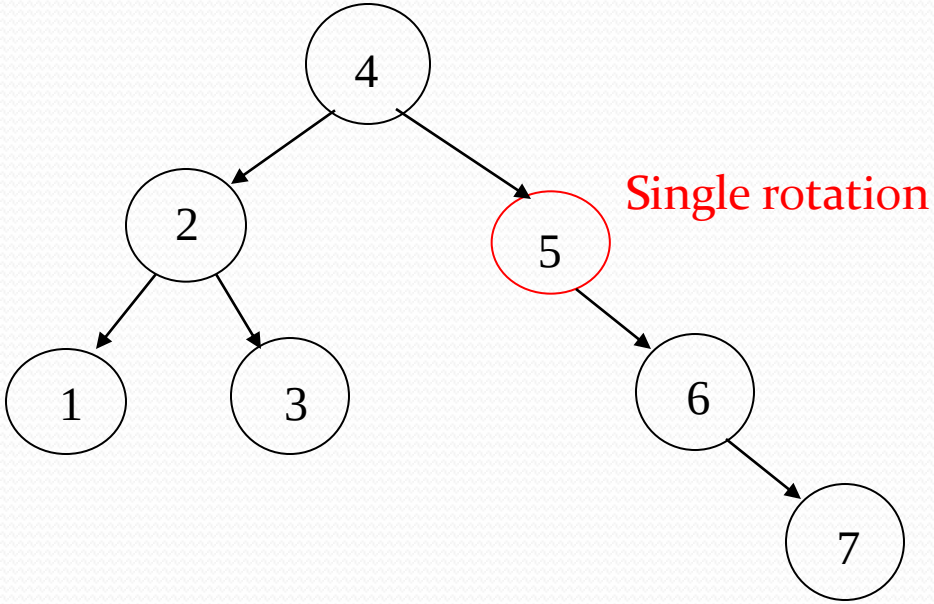
Example

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



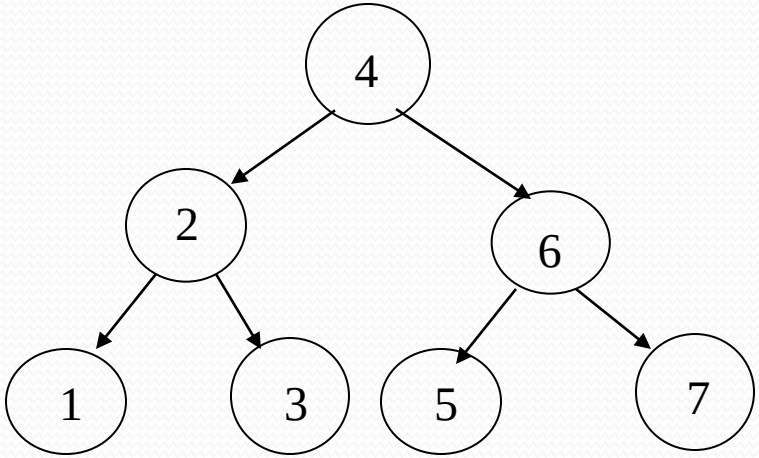
Example

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



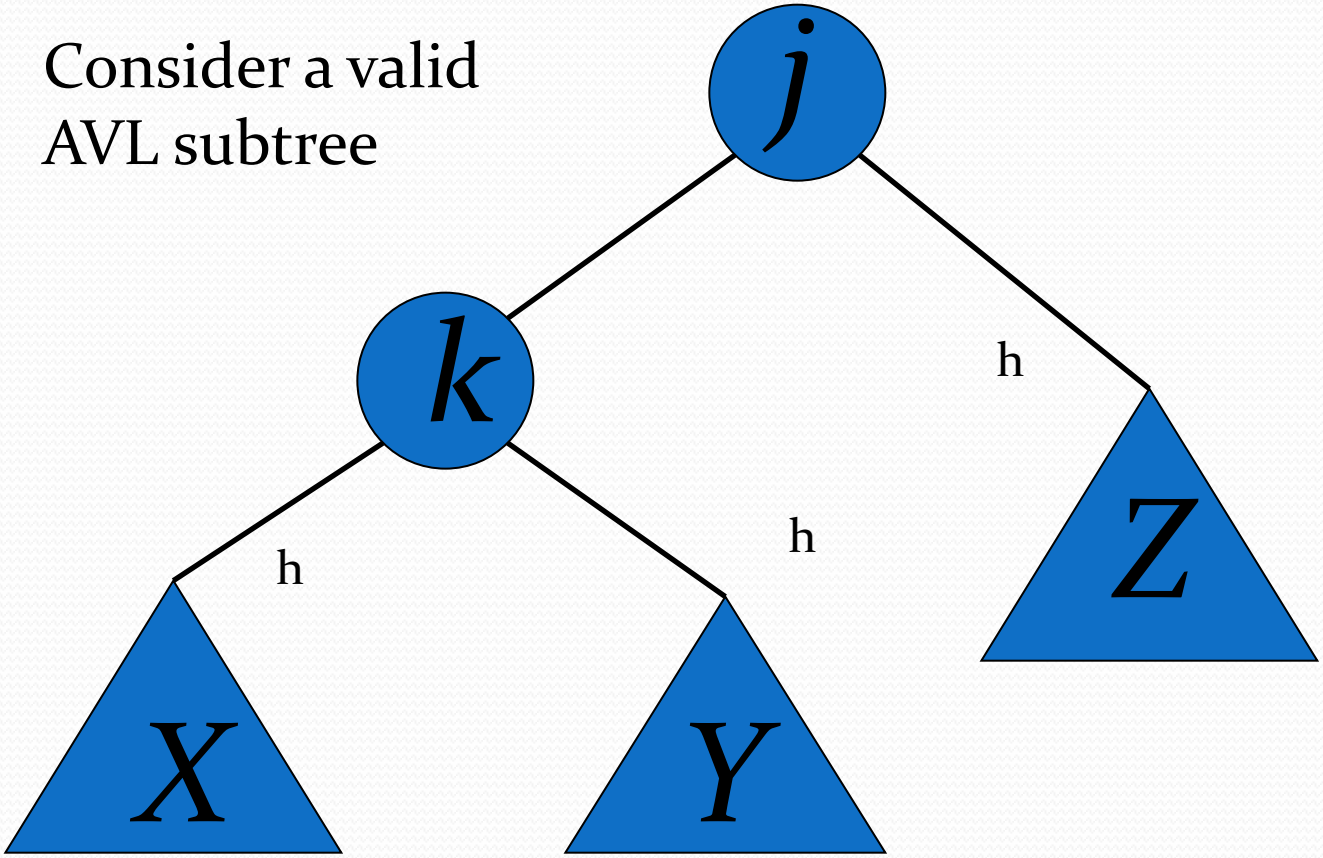
Example

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

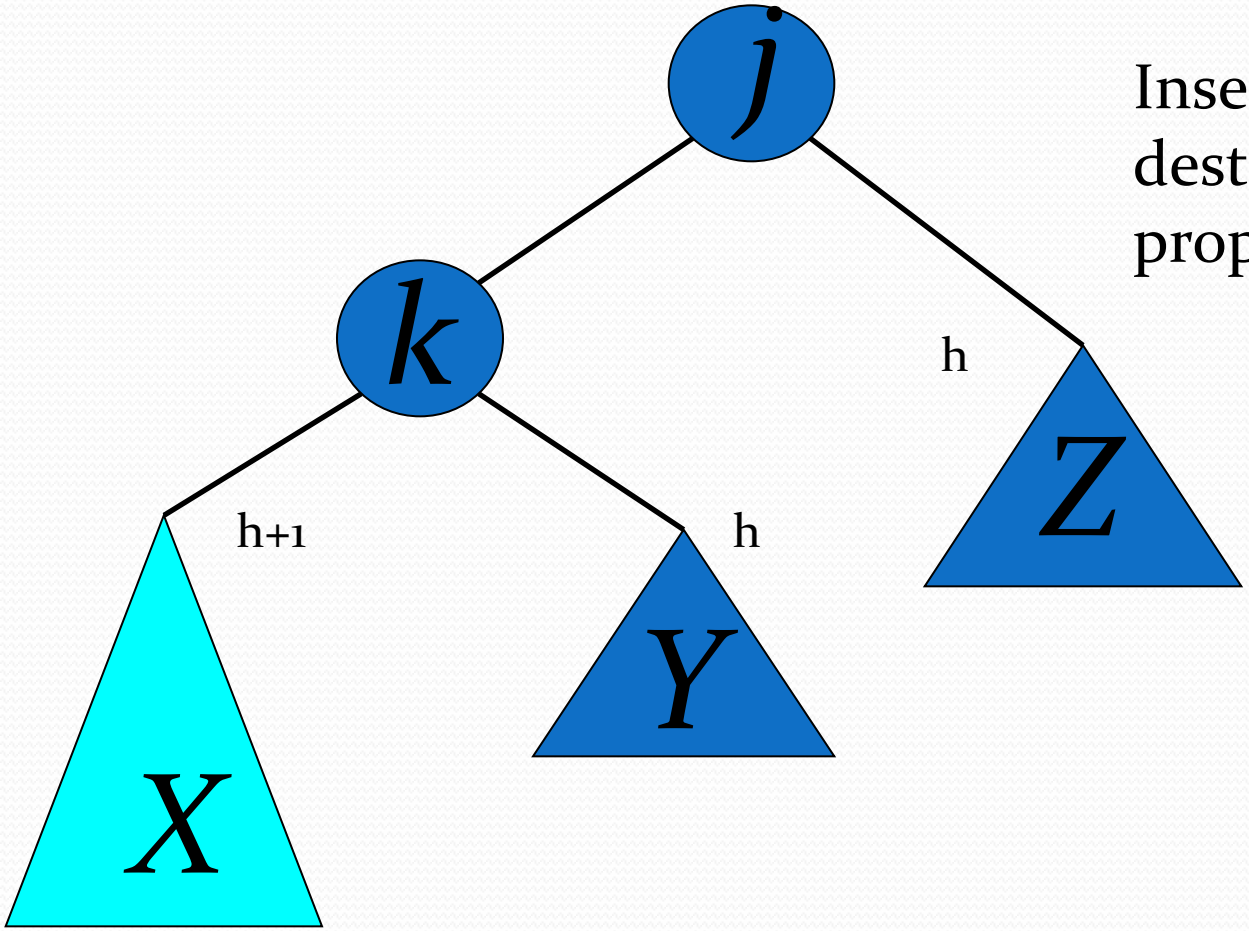


AVL Insertion: Outside Case

Consider a valid AVL subtree

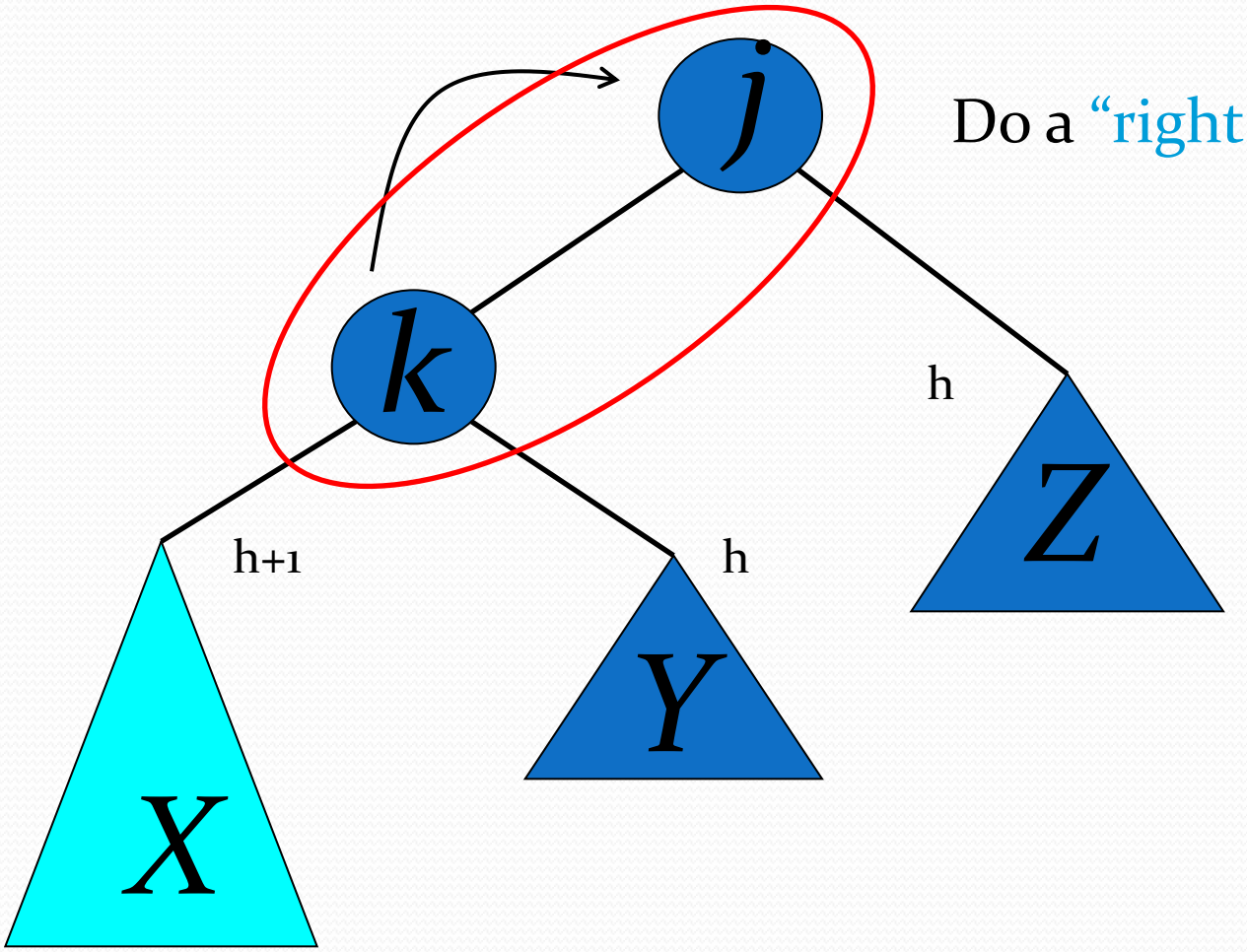


AVL Insertion: Outside Case



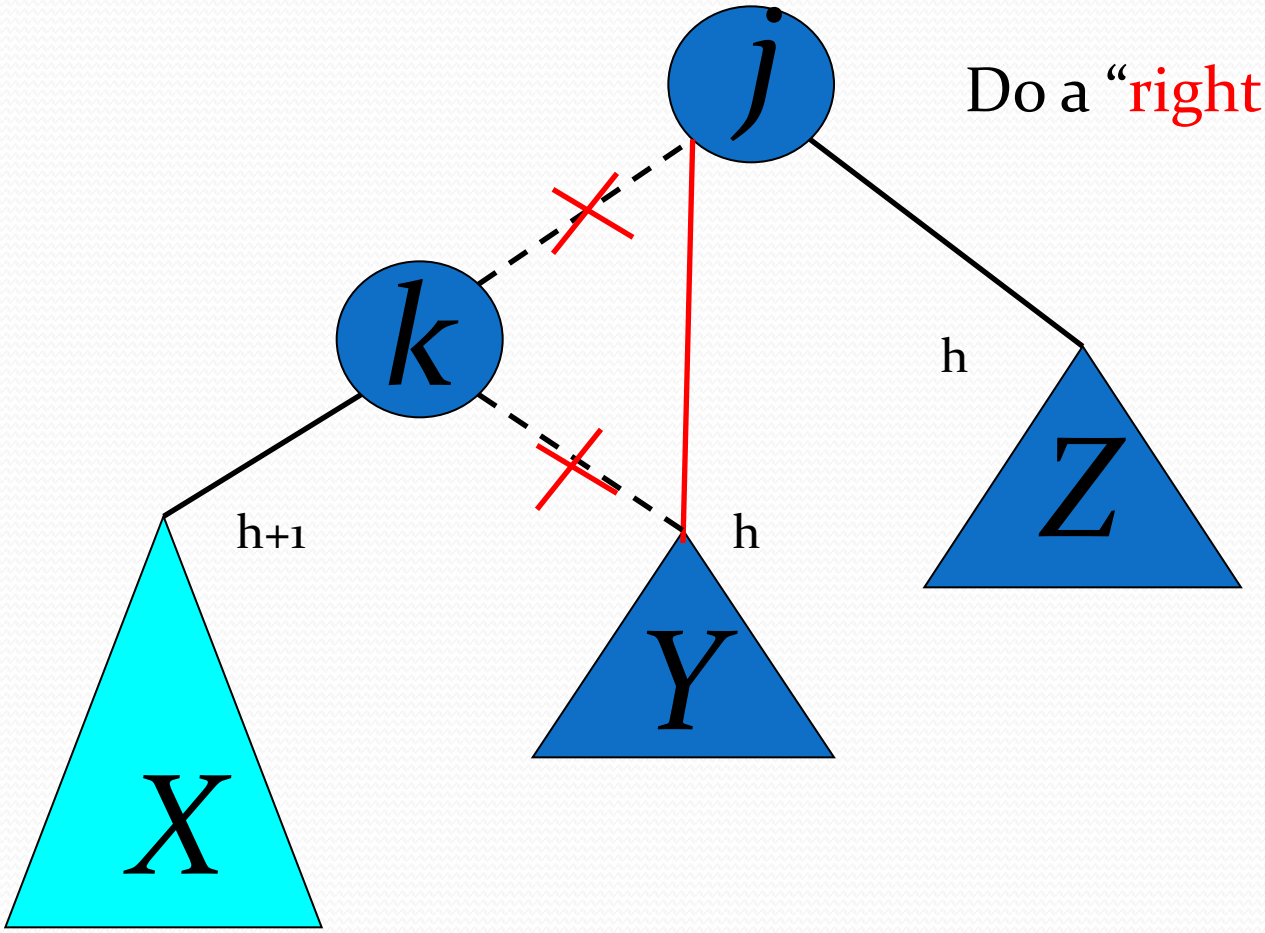
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case



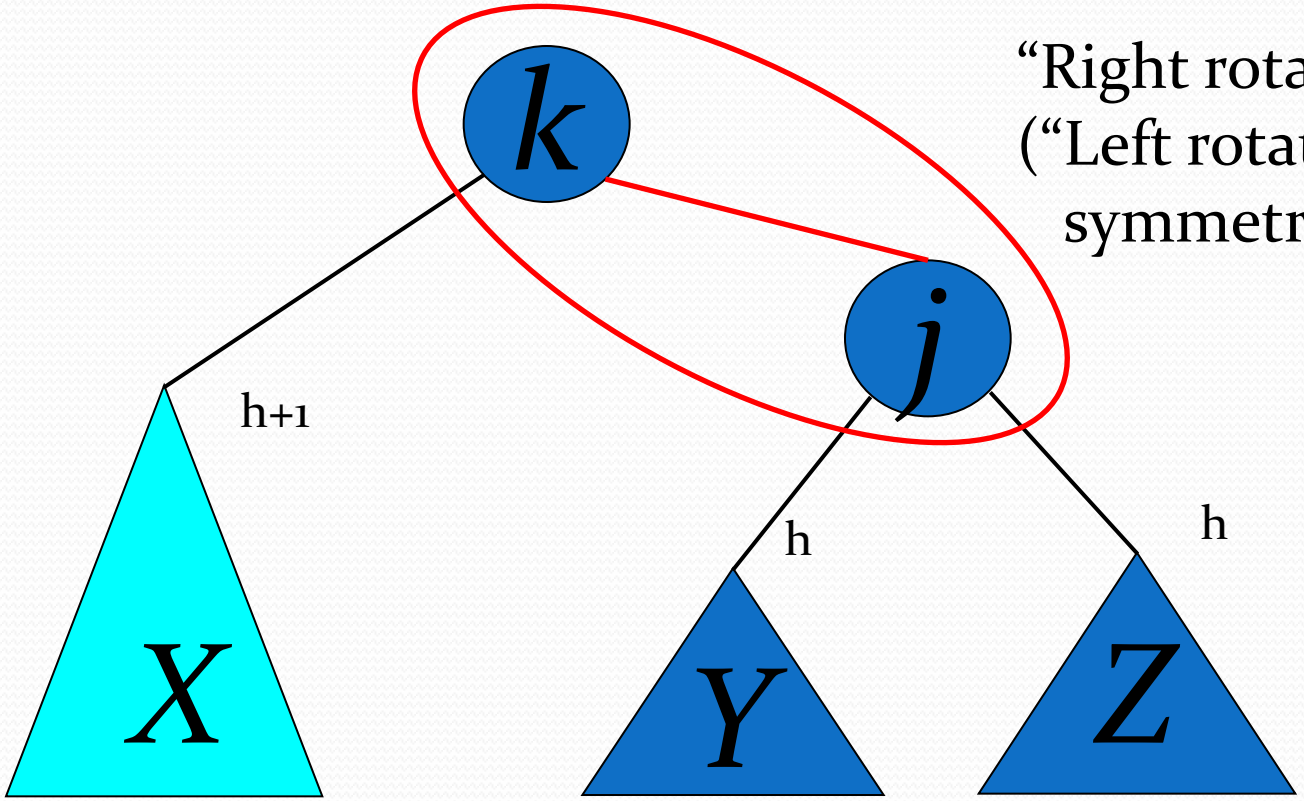
Do a "right rotation"

Single right rotation



Do a "right rotation"

Outside Case Completed

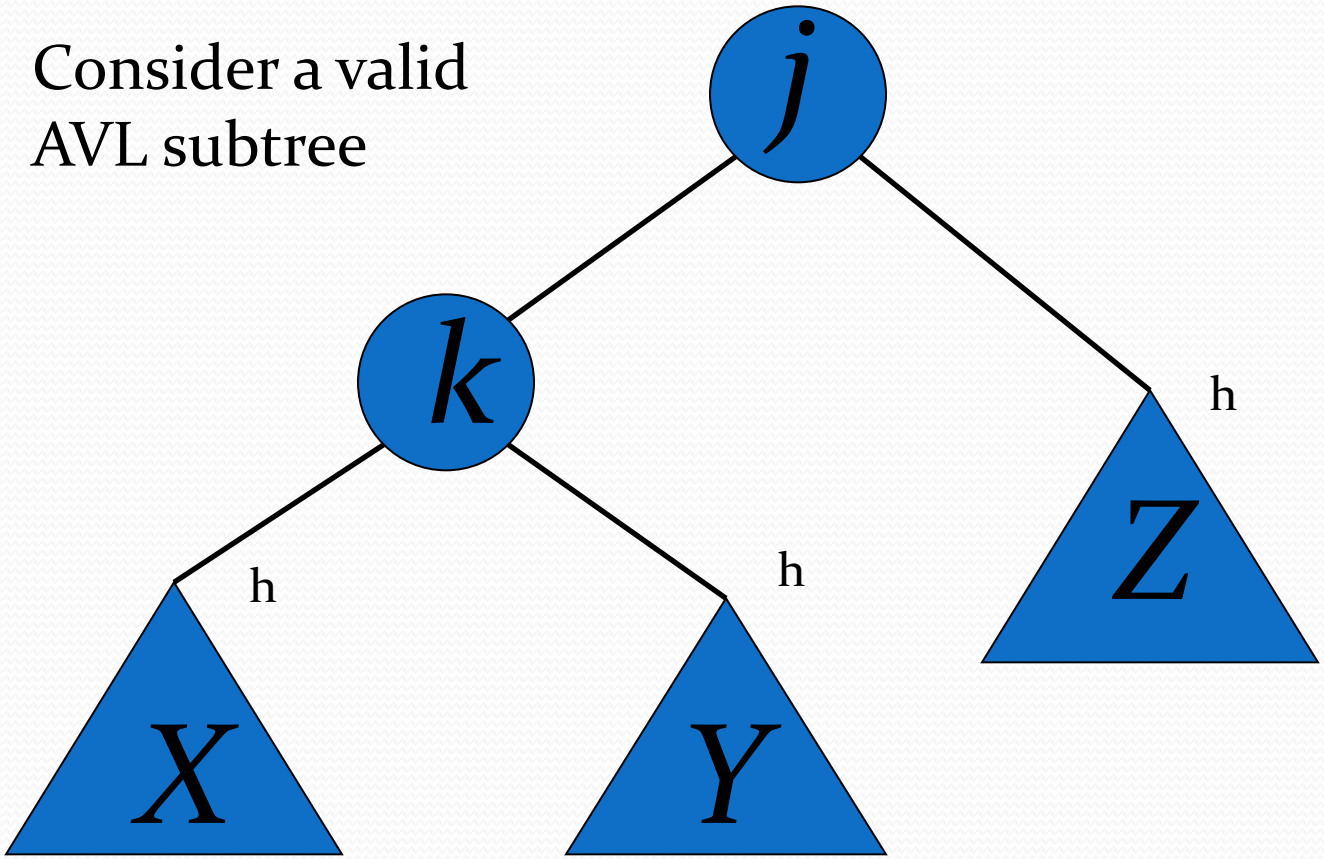


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

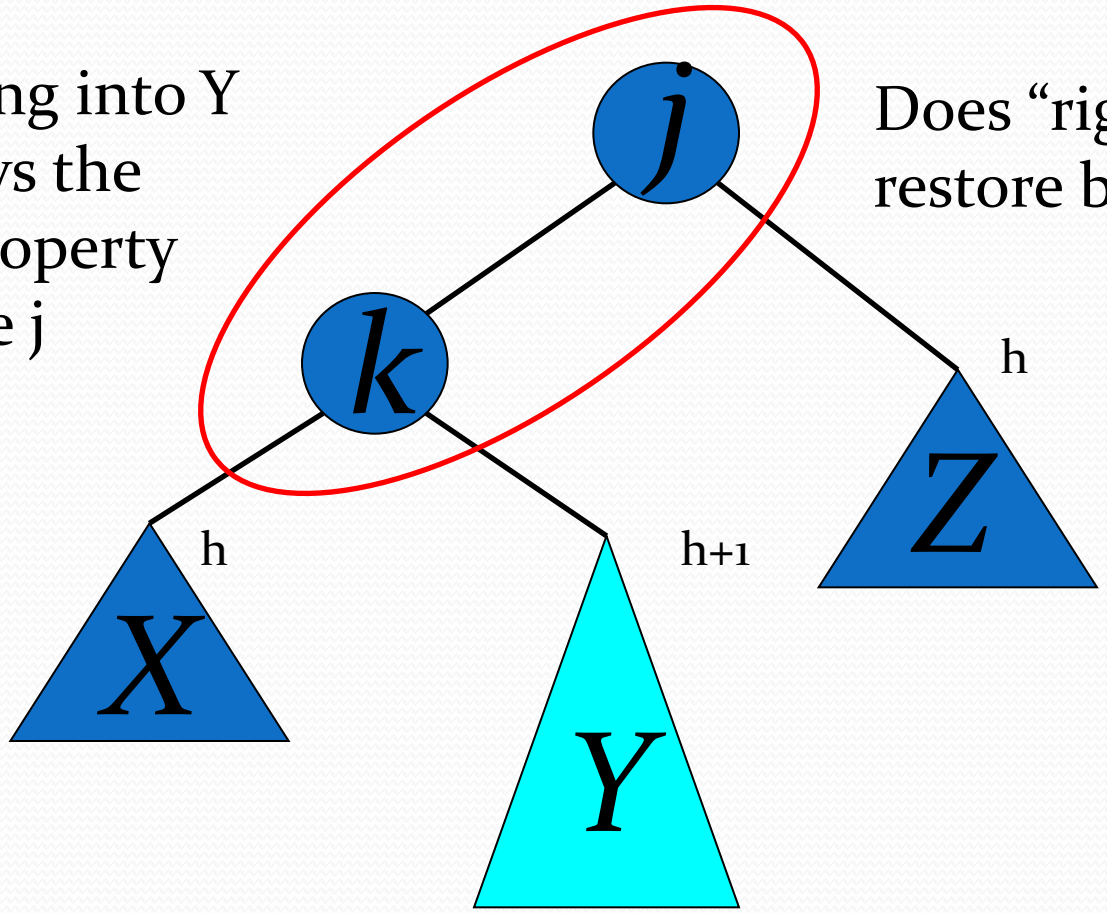
AVL Insertion: Inside Case

Consider a valid AVL subtree



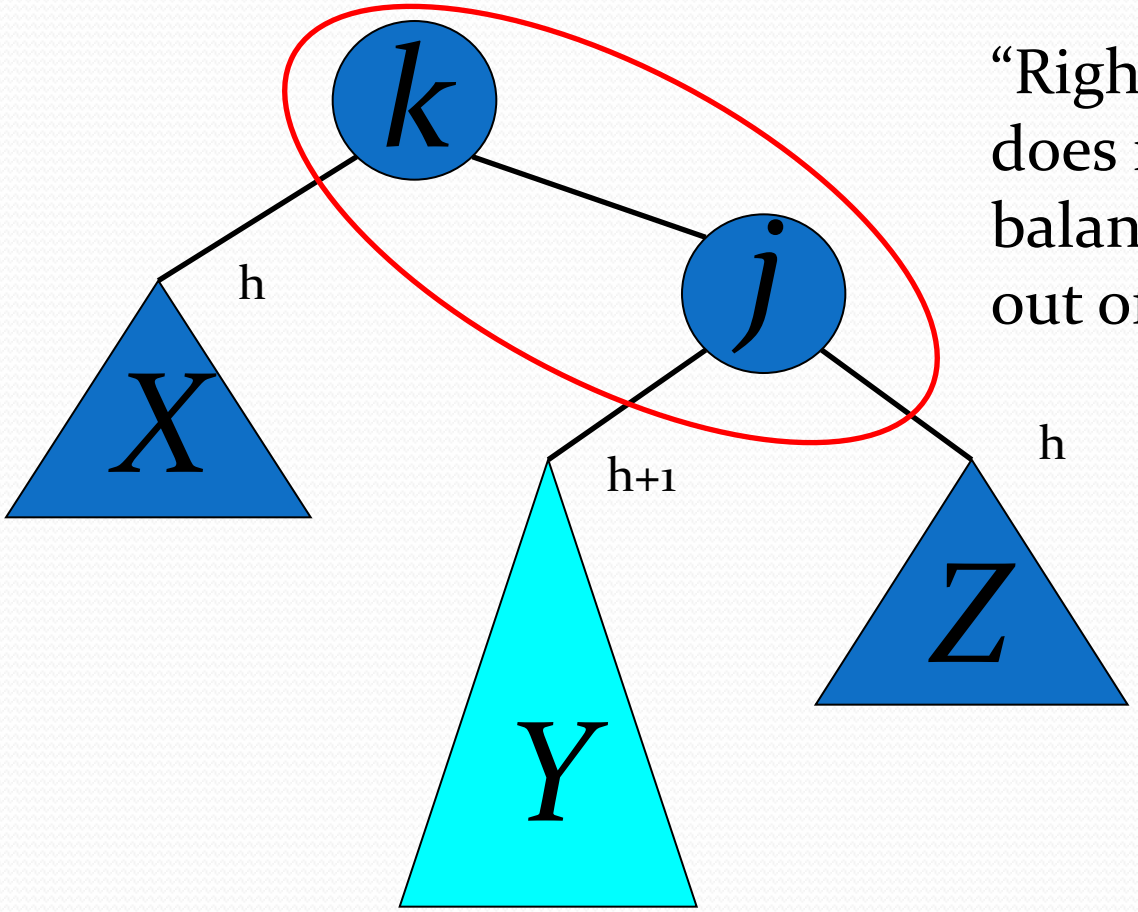
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j



Does “right rotation” restore balance?

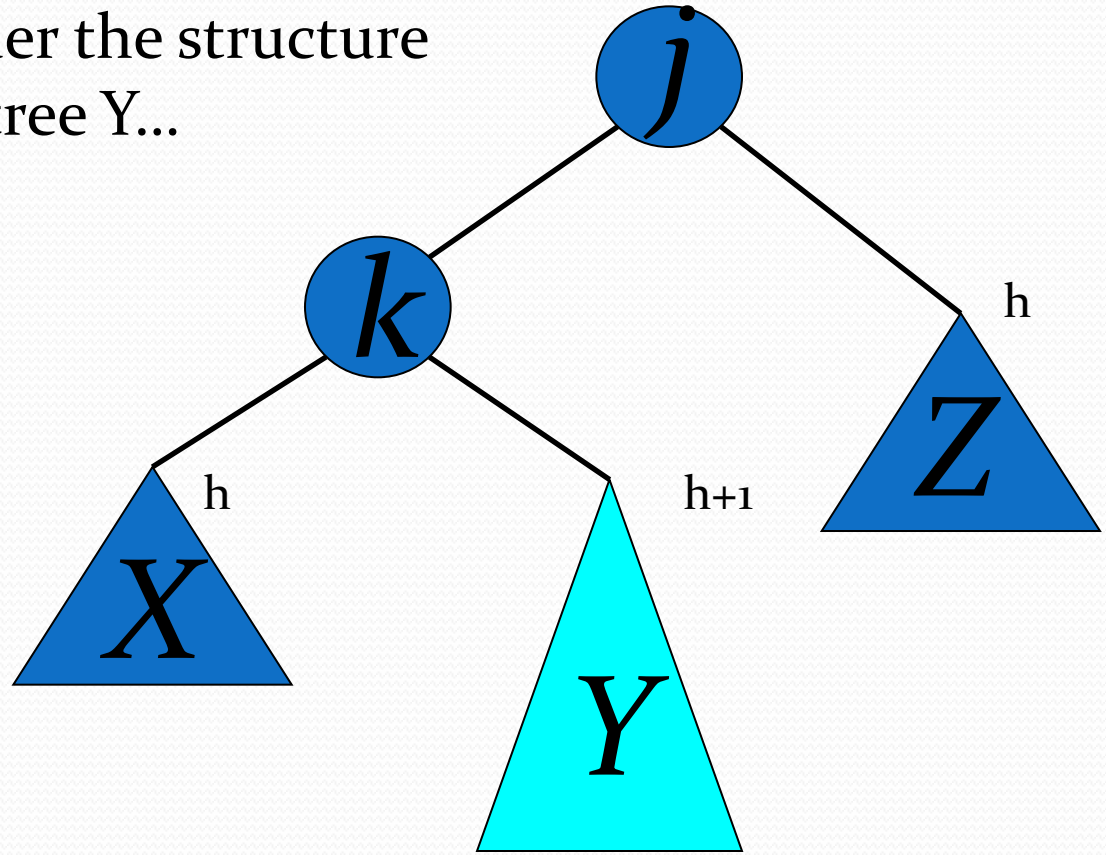
AVL Insertion: Inside Case



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

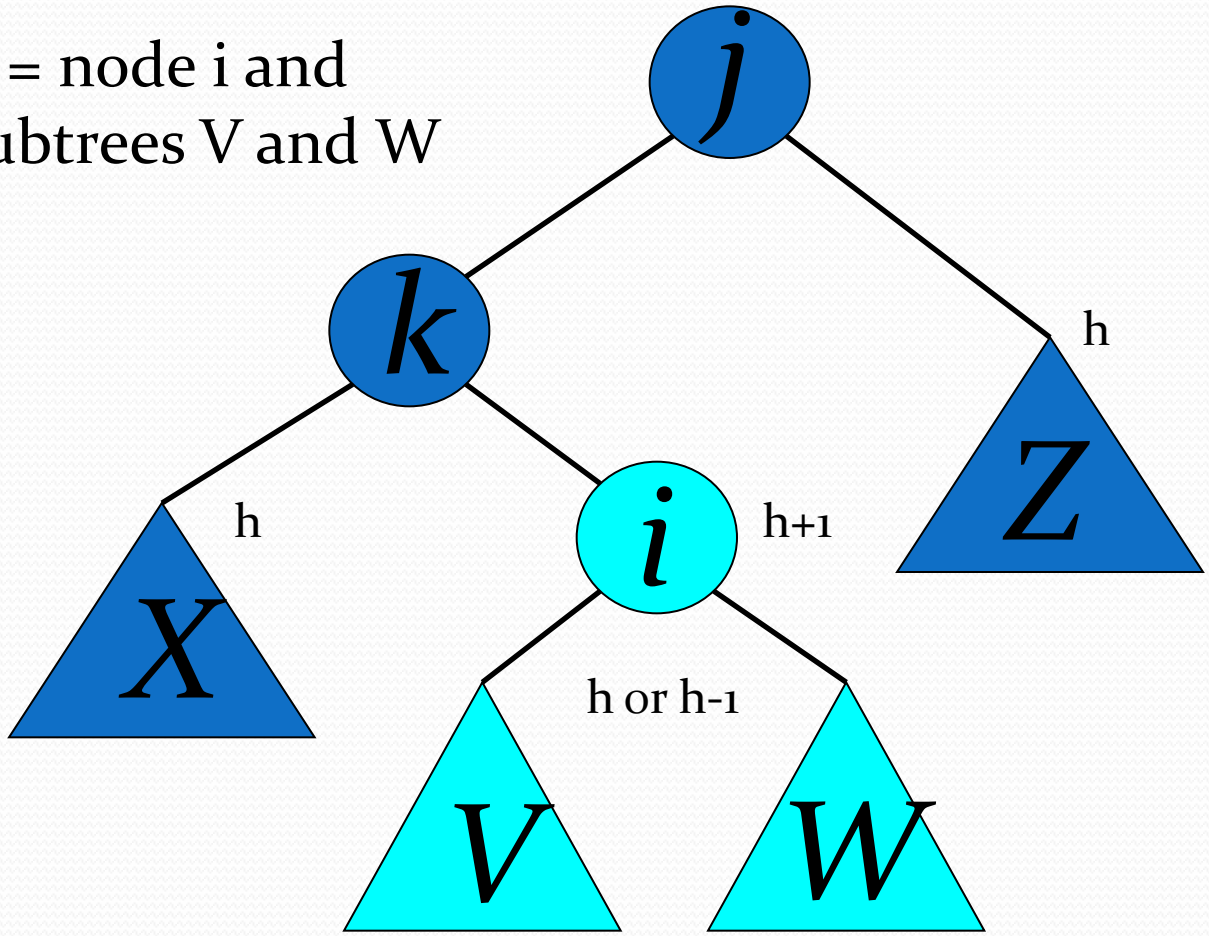
AVL Insertion: Inside Case

Consider the structure of subtree Y…

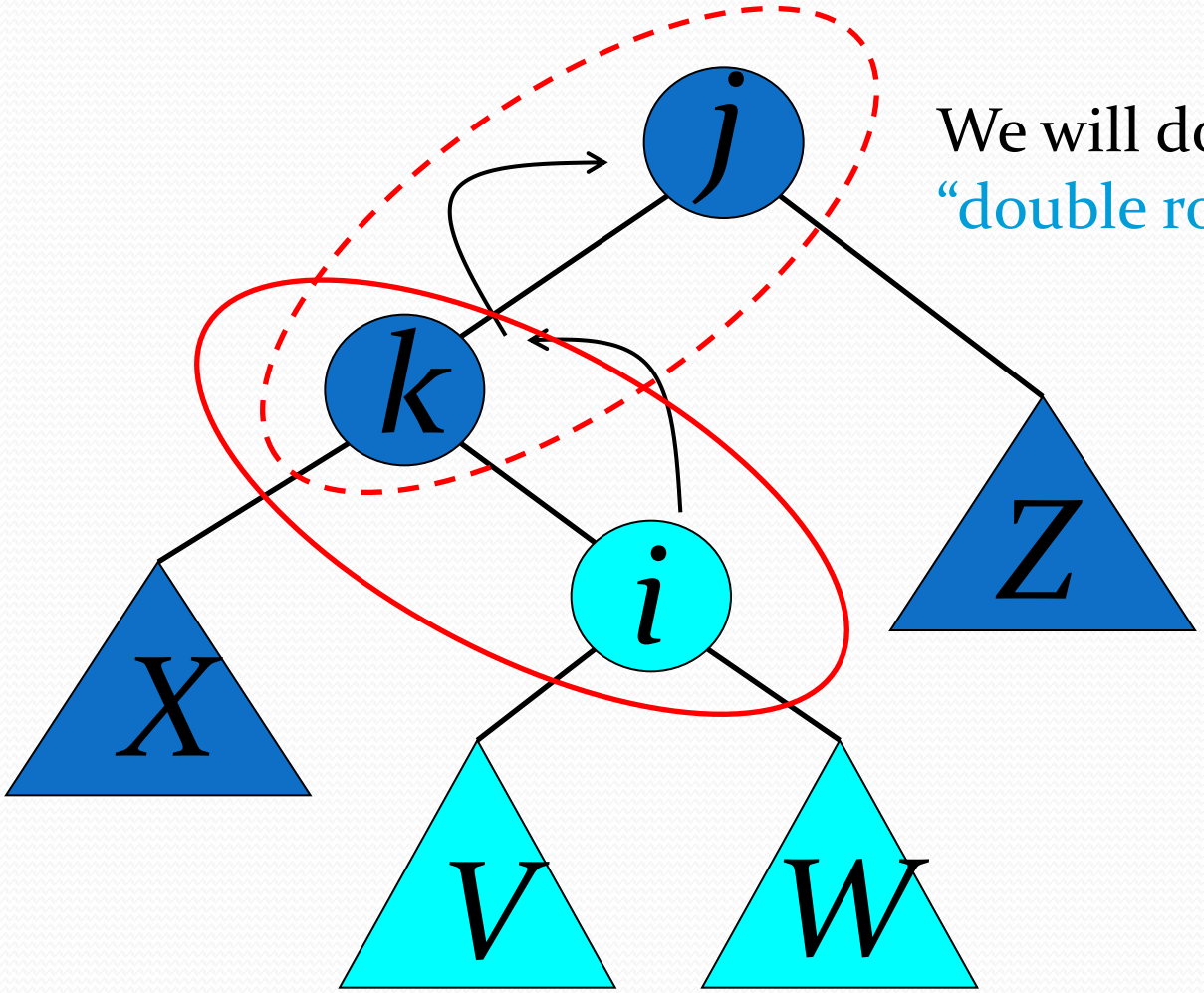


AVL Insertion: Inside Case

Y = node i and
subtrees V and W

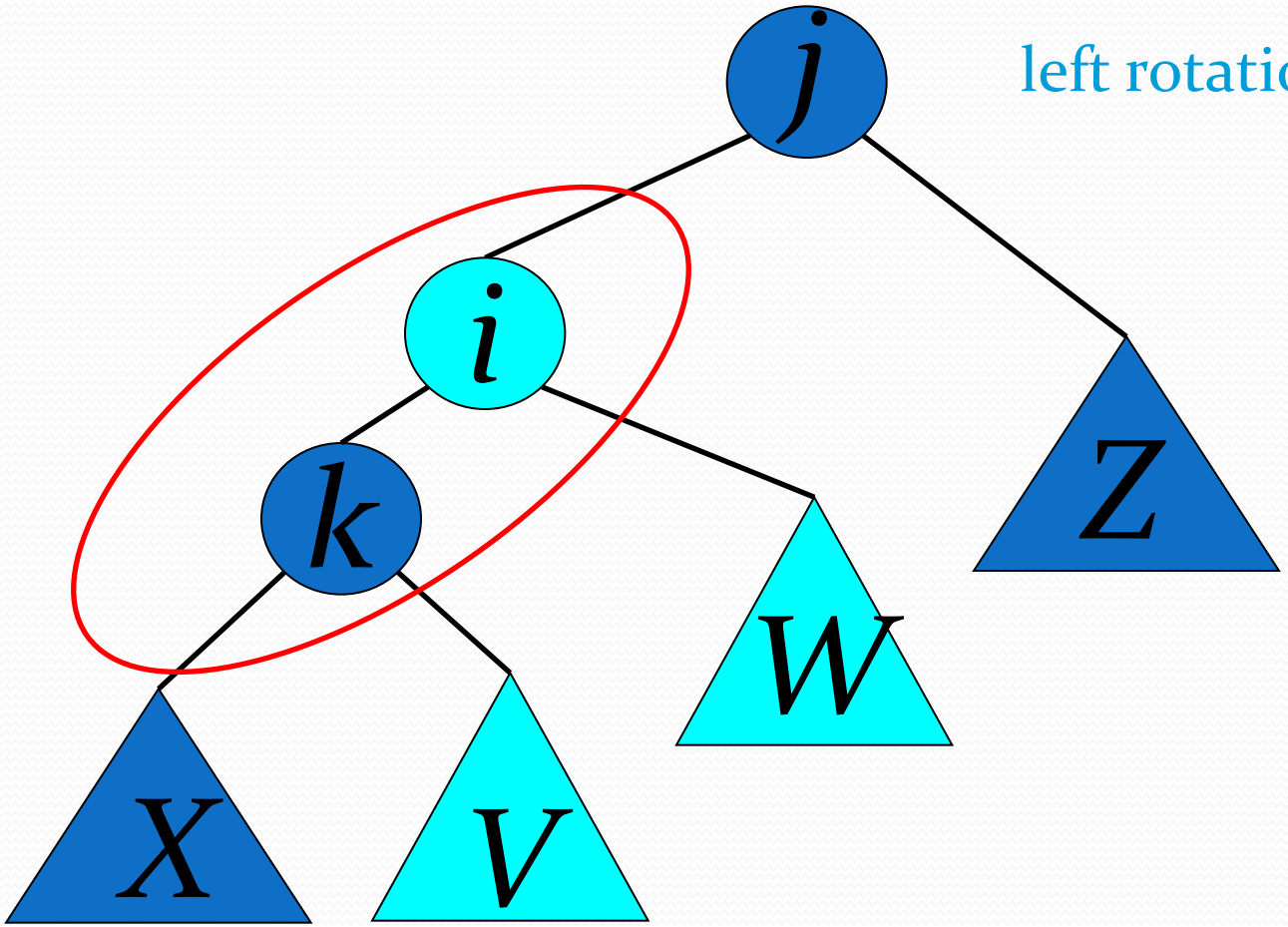


AVL Insertion: Inside Case



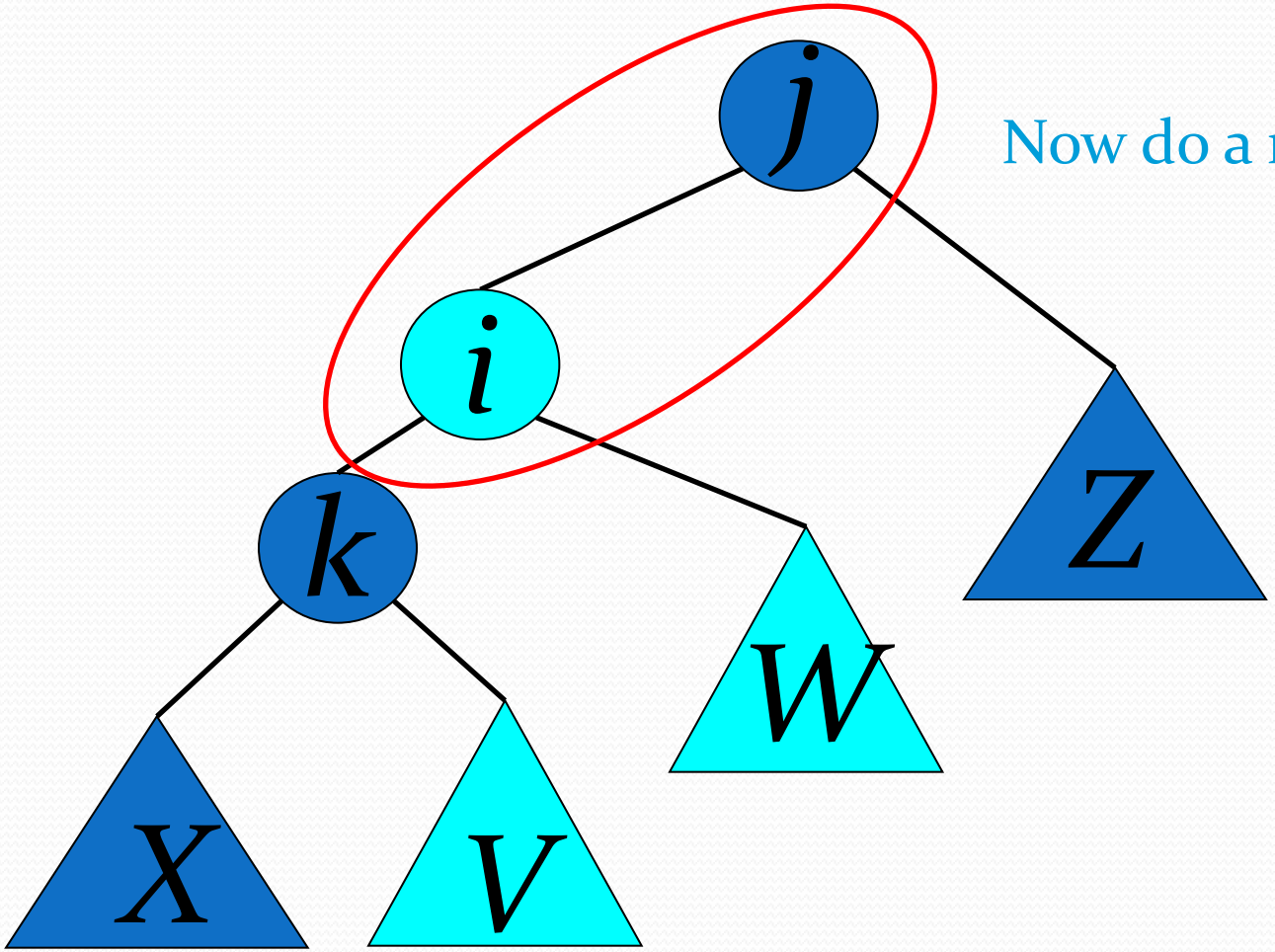
We will do a left-right "double rotation" . . .

Double rotation : first rotation



left rotation complete

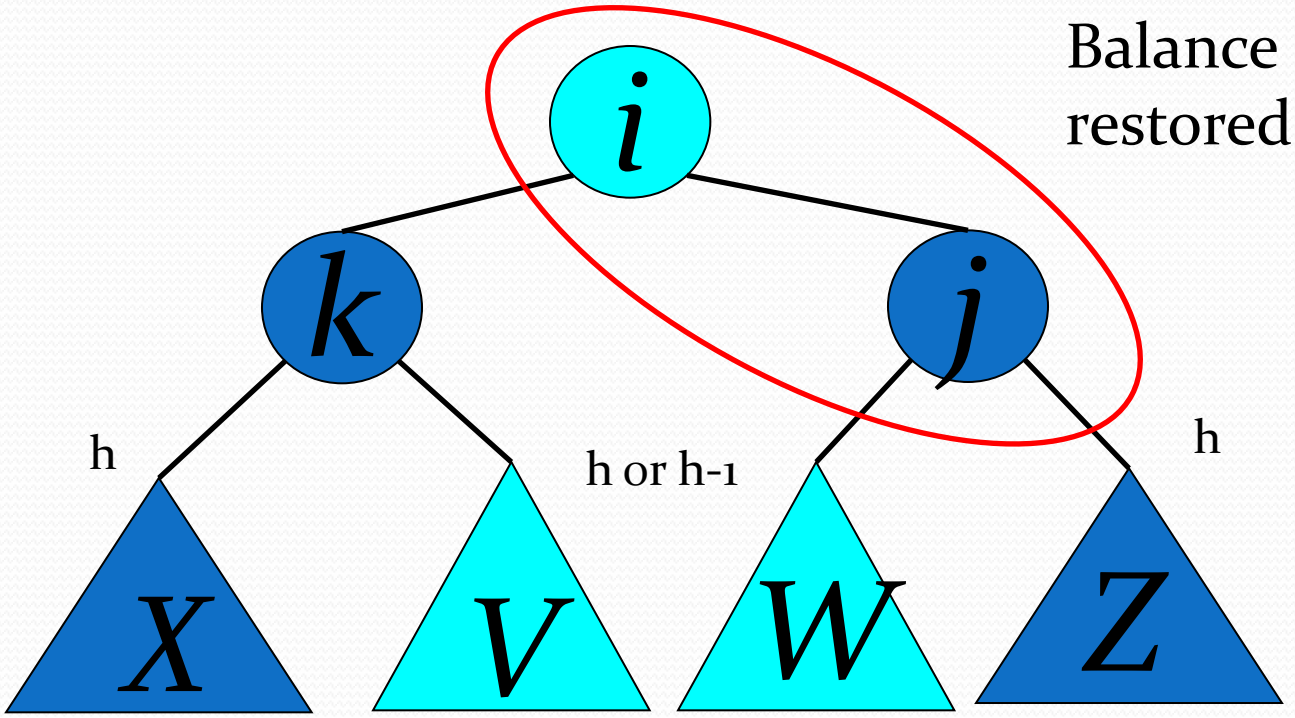
Double rotation : second rotation



Now do a right rotation

Double rotation : second rotation

right rotation complete



Balance has been restored

B-Tree

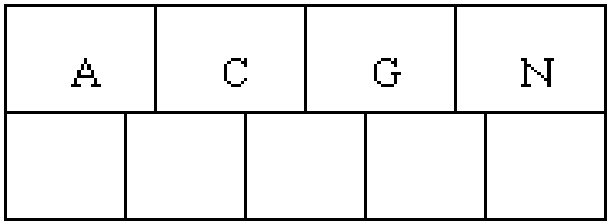
B-tree is a fairly well-balanced tree.

- 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.

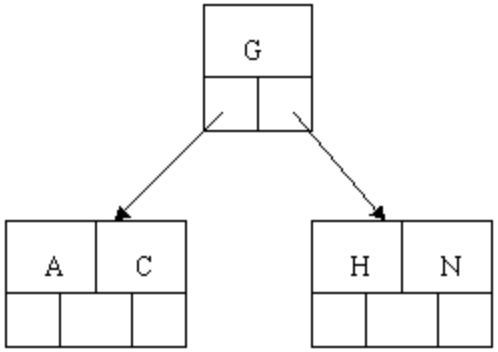
Let's work our way through an example similar to that given by Kruse. Insert the following letters into what is originally an empty B-tree of **order 5**:

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

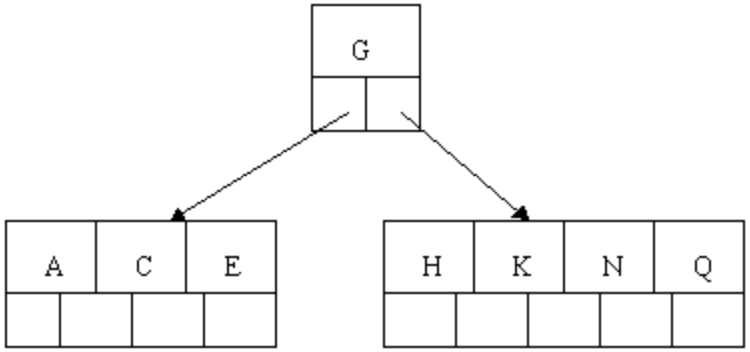
Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



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.

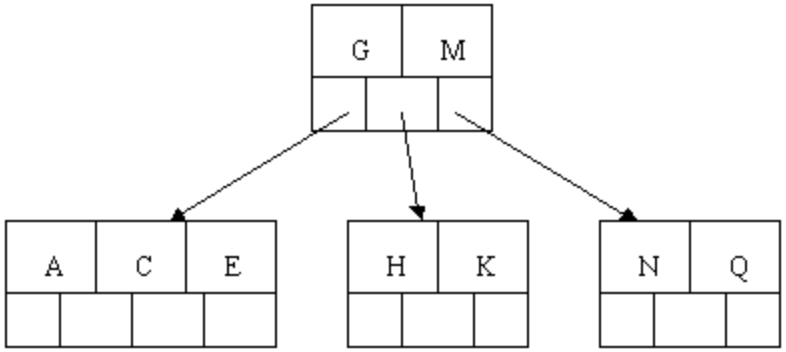


Inserting E, K, and Q proceeds without requiring any splits:

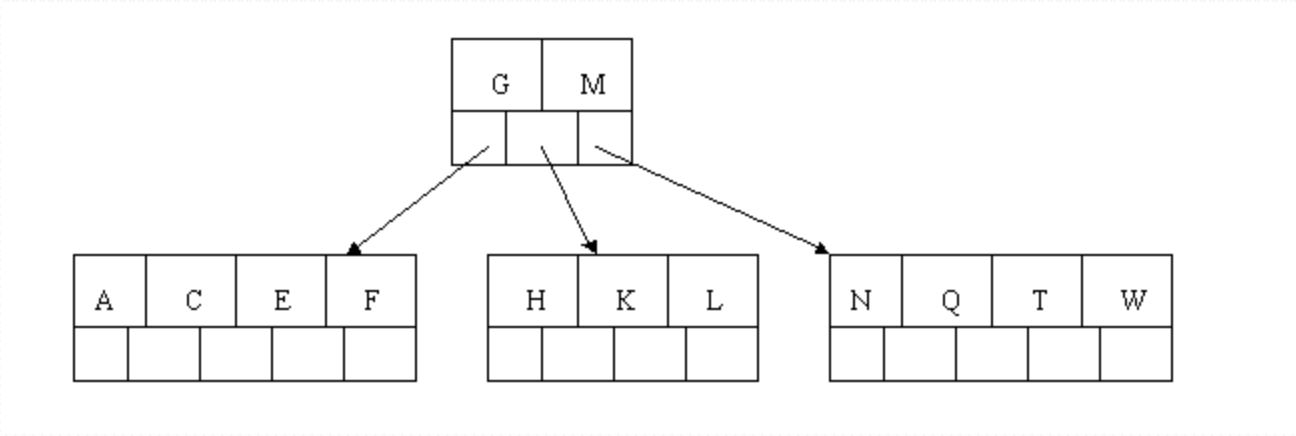


H E K Q M F W L T Z D P R X Y S

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

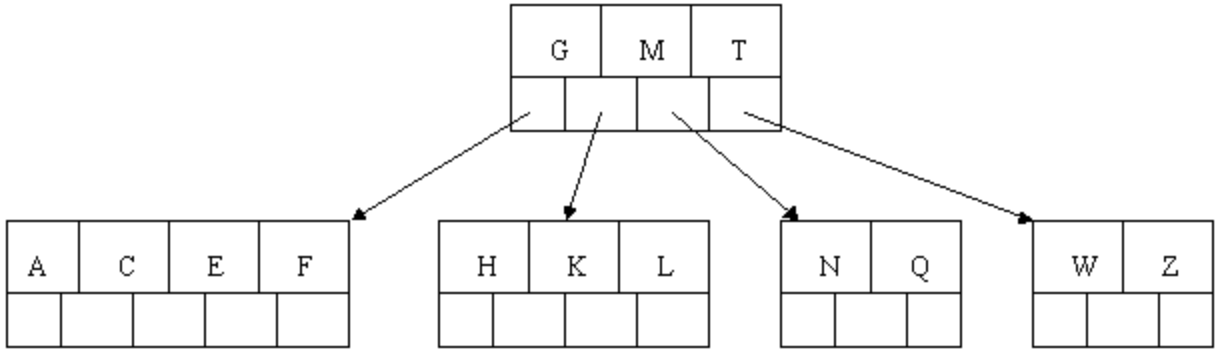


The letters F, W, L, and T are then added without needing any split.

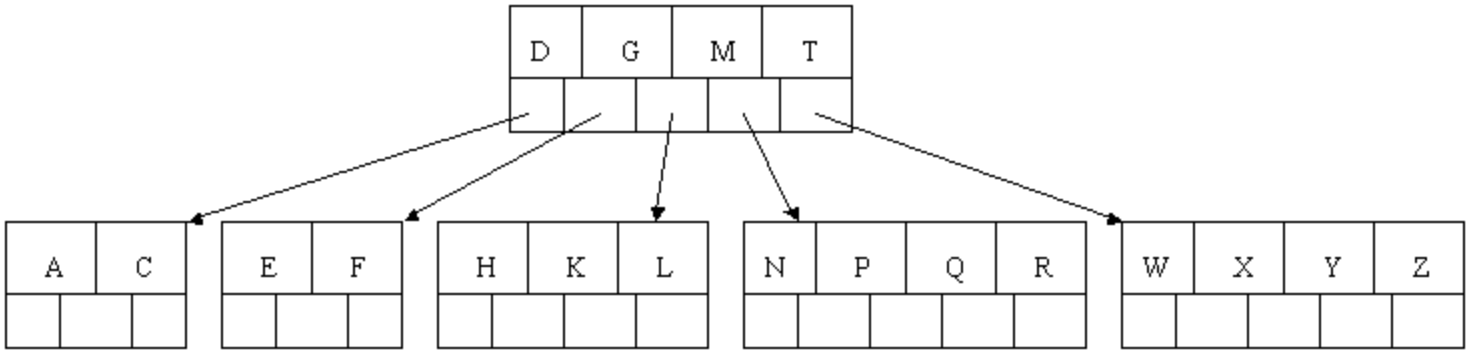


M F W L T Z D P R X Y S

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.

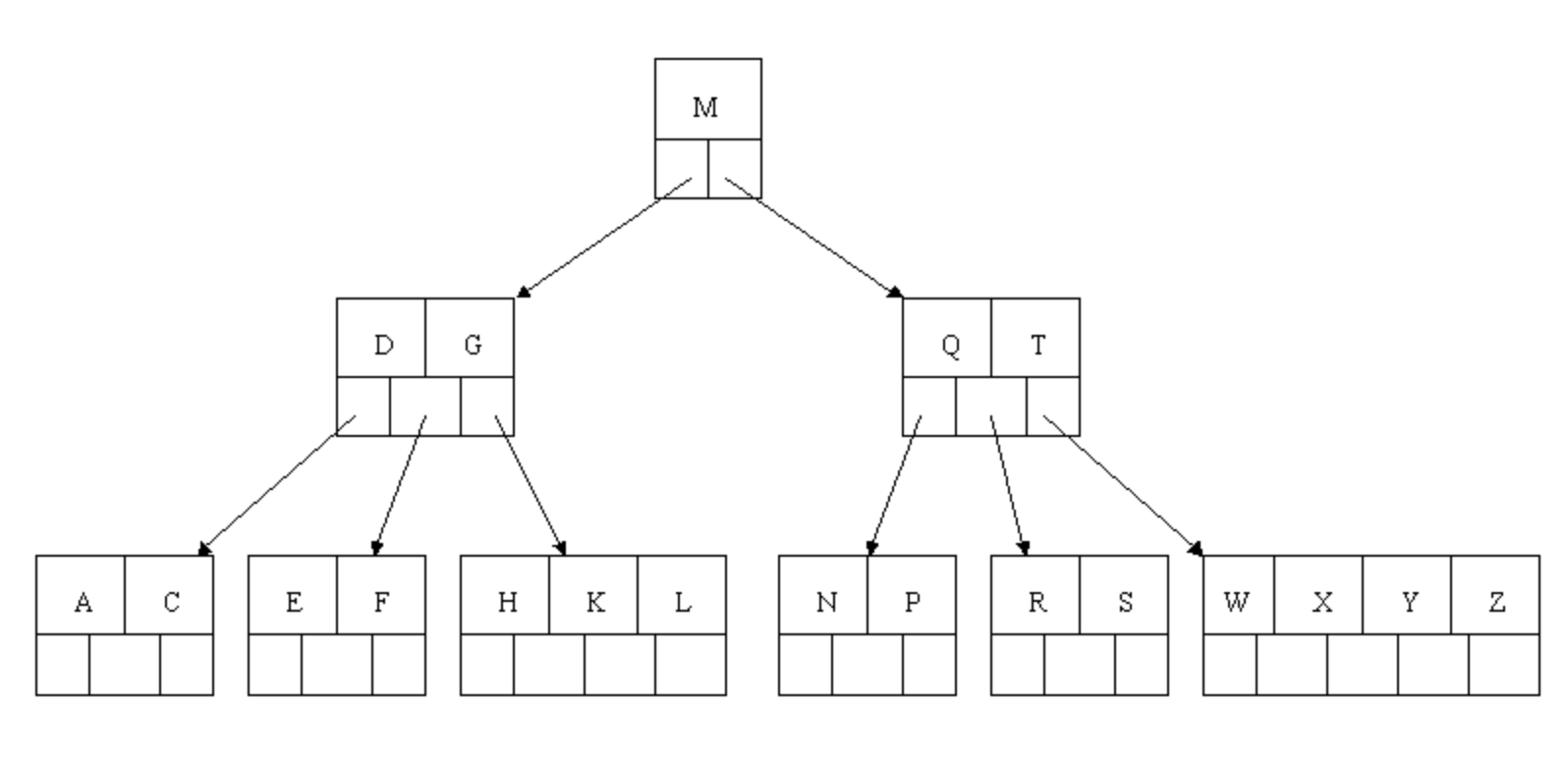


The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



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

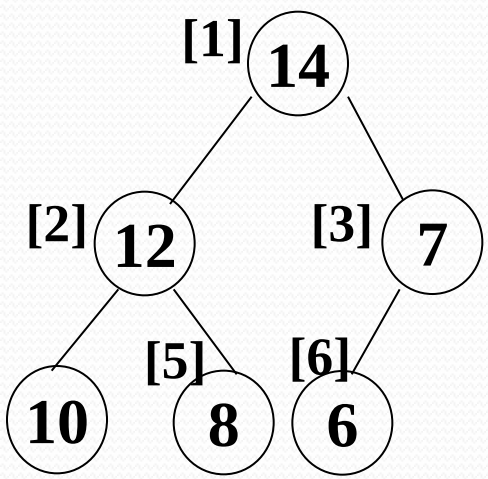
A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children. A *max heap* is a complete binary tree that is also a max tree.

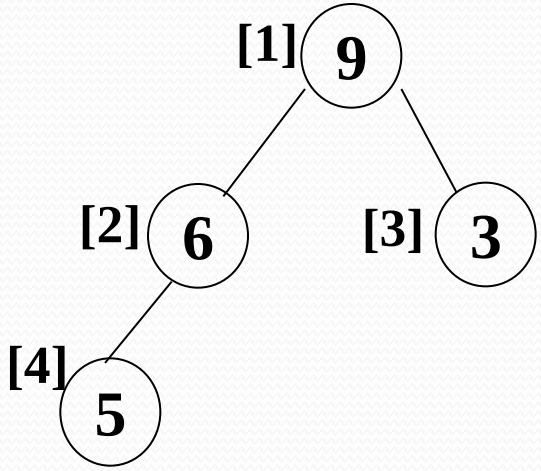
A *min tree* is a tree in which the key value in each node is no larger than the key values in its children. A *min heap* is a complete binary tree that is also a min tree.

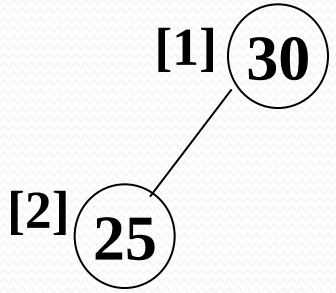
Operations on heaps:

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

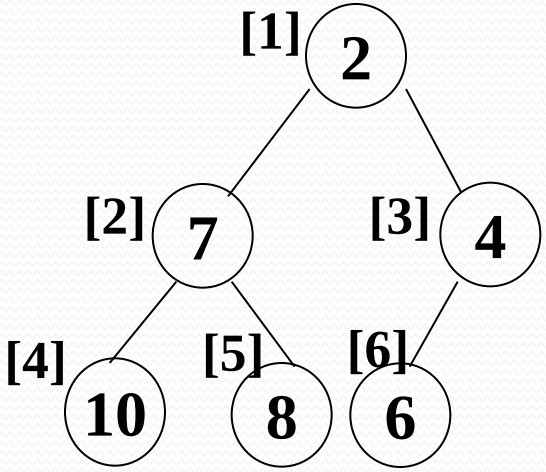
Max Heap

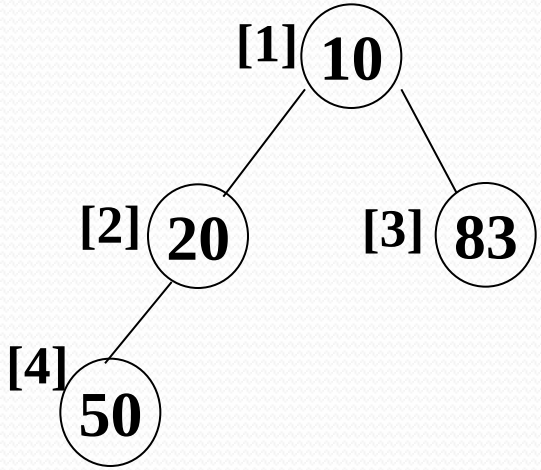


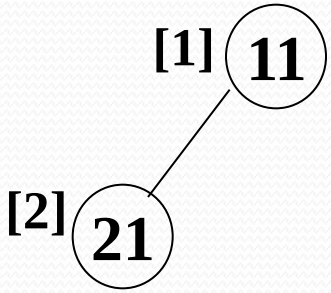




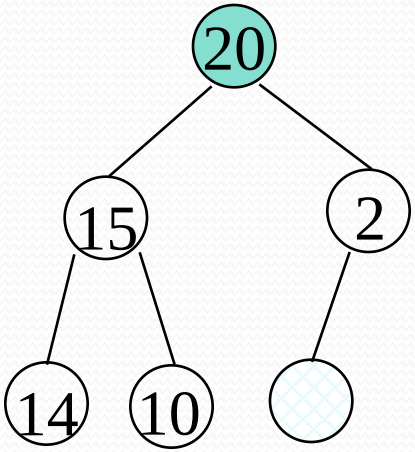
Min Heap



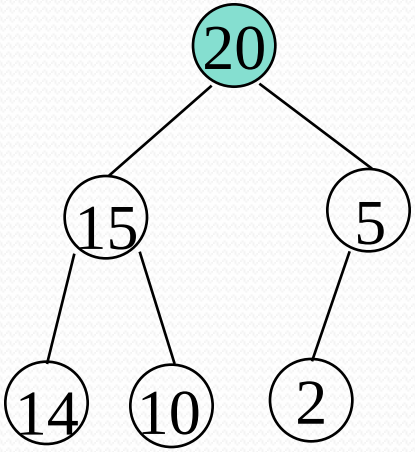




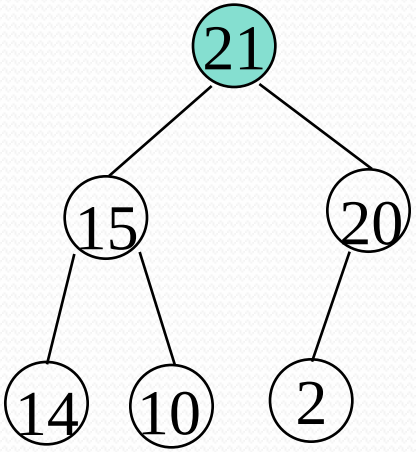
Example of Insertion to Max Heap



initial location of new node



insert 5 into heap



insert 21 into heap

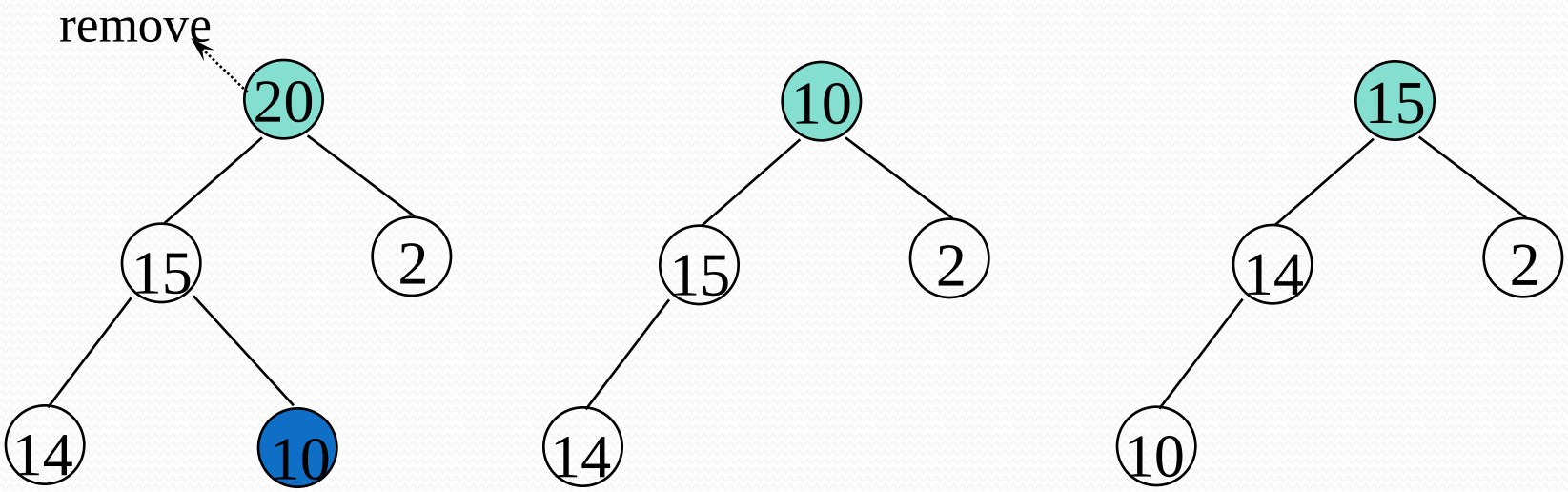
Insertion into a Max Heap

```
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;
}
```

Thank you