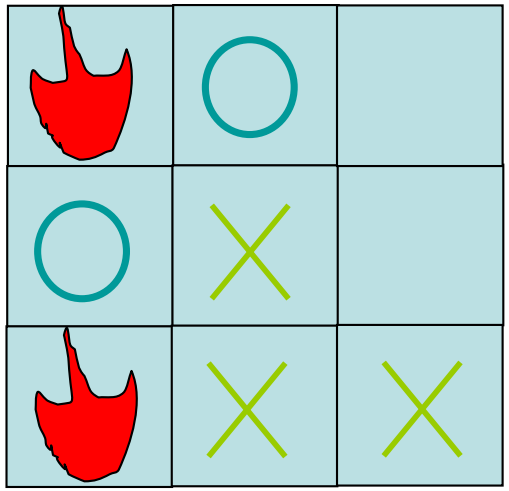
Data Structures
Lec-9 Balanced Binary Search Trees

Objective

- Game Tree
  - Minimax
  - BFS/DFS
  - α-β pruning
- AVL Tree
  - Definition
  - Rotations
- Splay tree
  - Operations

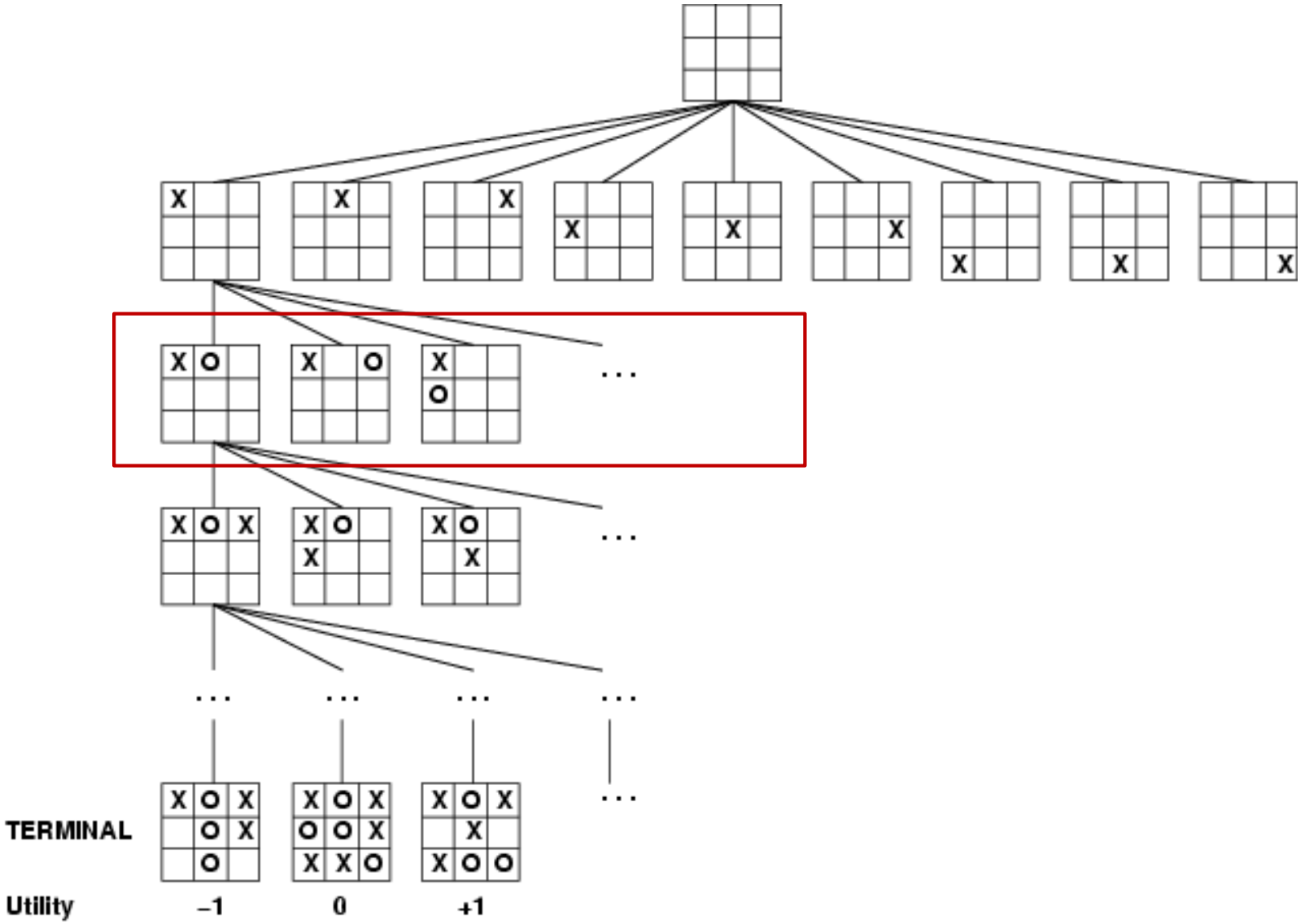
Game Tree —— Motivation

- How do you choose strategies when playing turn-based games?
- How should you design AI so that it can compete with humans in turn-based games?



Tic Tac Toe

Game Tree



Game

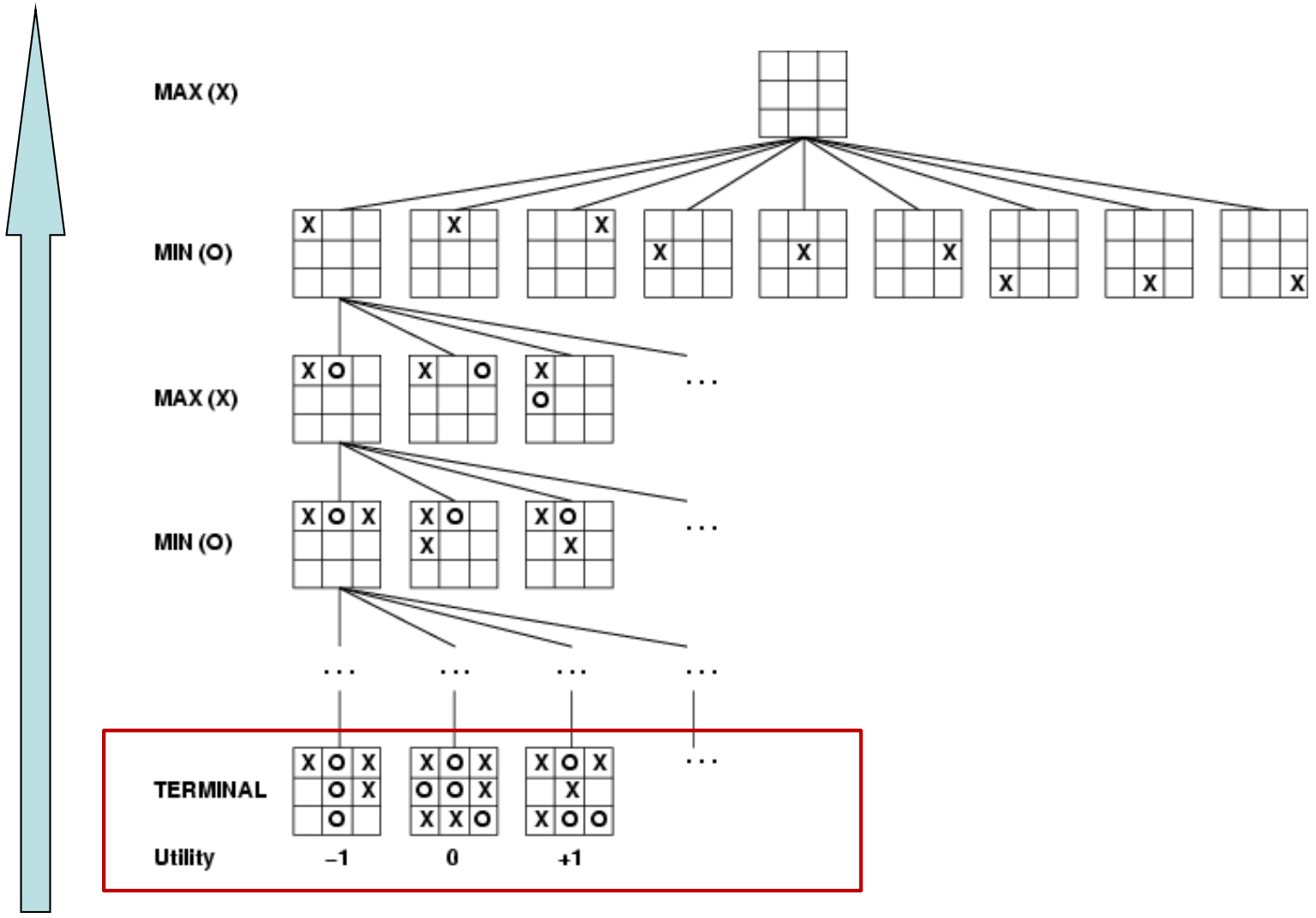
- Adversarial search problems
- Consider games with the following two properties:
  - **Two players** - we do not deal with coalitions, etc.
  - **Zero sum** - one player's win is the other's loss; there are no cooperative victories

- Examples: tic tac toe, chess, checkers, and go

Game Tree

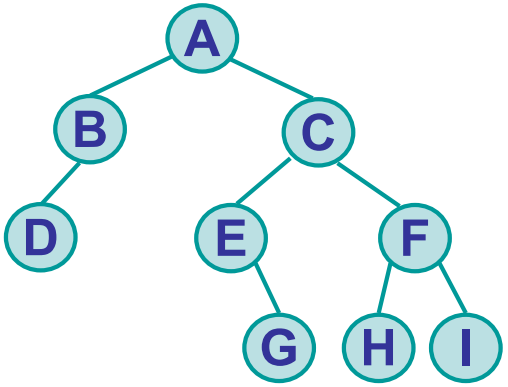
- Properties of the game tree:
  - Root of the tree is the initial game status
  - Every node in the tree is a possible game status
  - Every edge in the tree is a possible move by the players
  - Any path from root to leaf represents a possible game
- How to make decisions using the game tree?
  - Each leaf is associated with a value (advantages to player A), bottom up evaluation for internal nodes
  - Player A always go to the child with maximum value
  - Player B always go to the child with minimum value

Minimax (How to search)



DFS (Depth First Search)

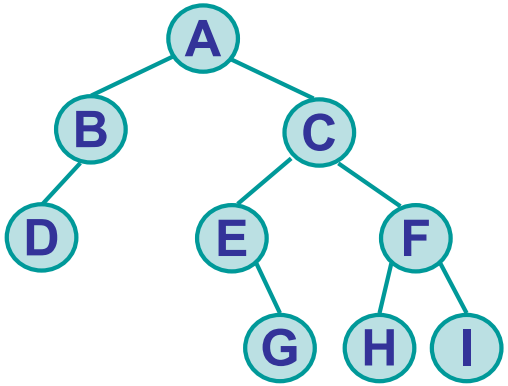
- DFS: Go as deep as you can
- In tree, DFS=
  - Preorder Traversal
  - Child nodes before Siblings
- Example (start from A):



DFS order: ABDCEGFHI

BFS (Breadth First Search)

- BFS: Go as broad as you can
- Example:

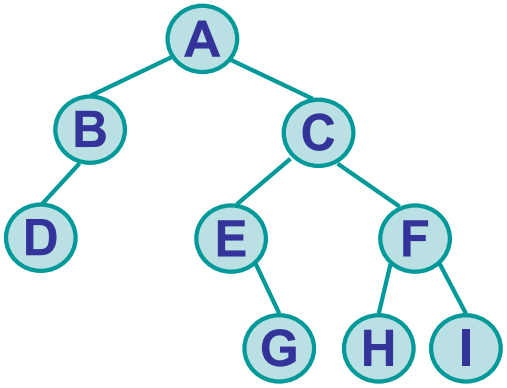


BFS order: ABCDEFGHI

- How to output nodes by increasing depth?

BFS (Breadth First Search)

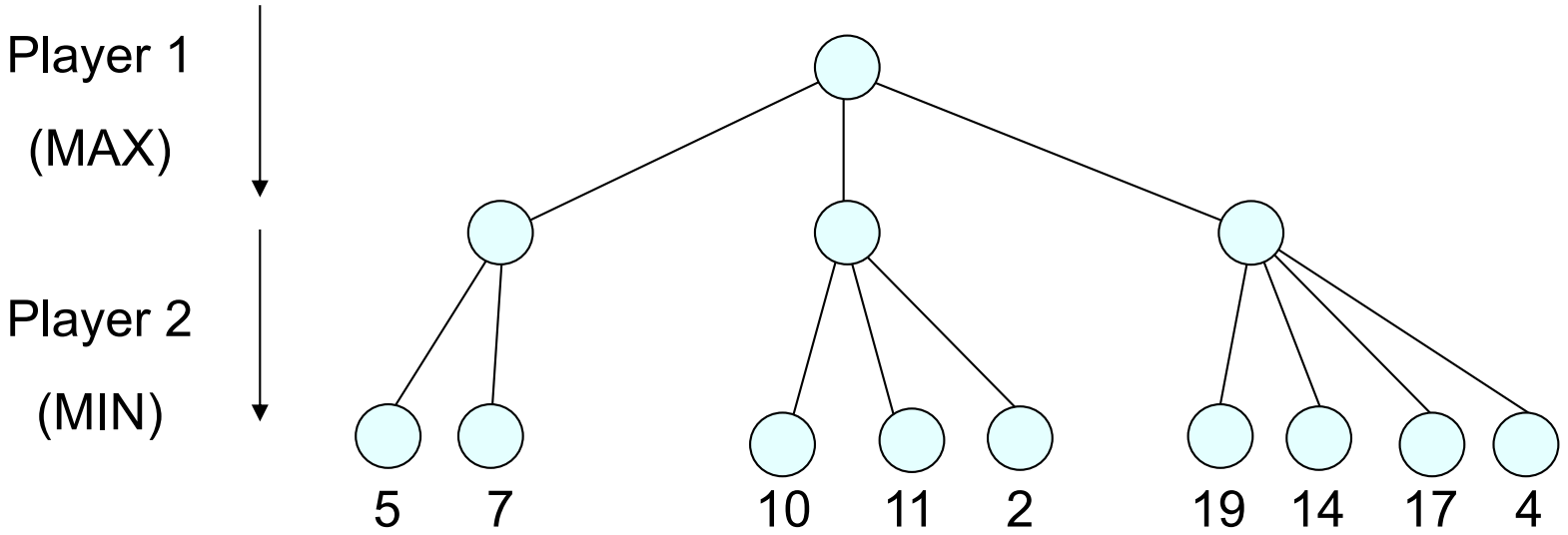
- BFS traversal using queue (FIFO)




```
L1  void BFS_queue(TreeNode root)
L2  {
L3        if (root == NULL)
L4           return;
L5        Queue<TreeNode> treeQueue;
L6         treeQueue.enqueue(root);
L7         while (treeQueue.empty() == false)
L8         {
L9           TreeNode currNode = treeQueue.dequeue();
L10          process(currNode->data);
L11          if (currNode->left != NULL)
L12             treeQueue.enqueue(currNode->left);
L13          if (currNode->right != NULL)
L14             treeQueue.enqueue(currNode->right);
L15        }
L16 }
```

A Simpler Example

- Value in a node means the advantage to player 1 if the game is in that status



- Which branch does player 1 take?

The Minimax Algorithm

- Designed to find the optimal strategy or just best first move for MAX

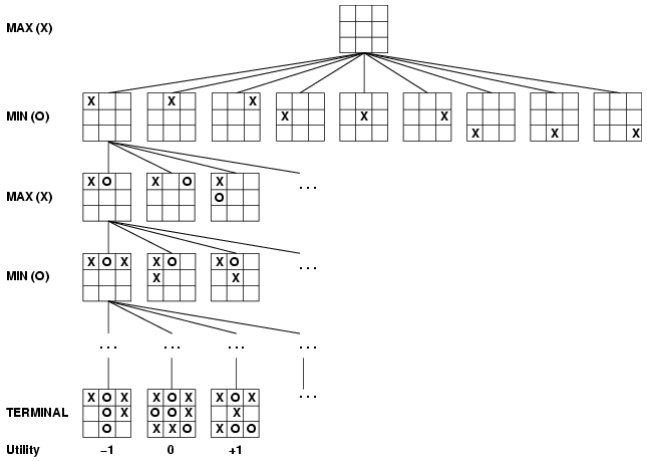
- **Brute-force**
  1. Generate the whole game tree to leaves
  2. Apply utility (payoff) function to leaves
  3. Back-up values from leaves toward the root:
     - a Max node computes the max of its child values
     - a Min node computes the min of its child values
  4. When value reaches the root: choose max value and the corresponding move.

- **Minimax**

  Search the game-tree in a **DFS** manner to find the value of the root

Properties of minimax

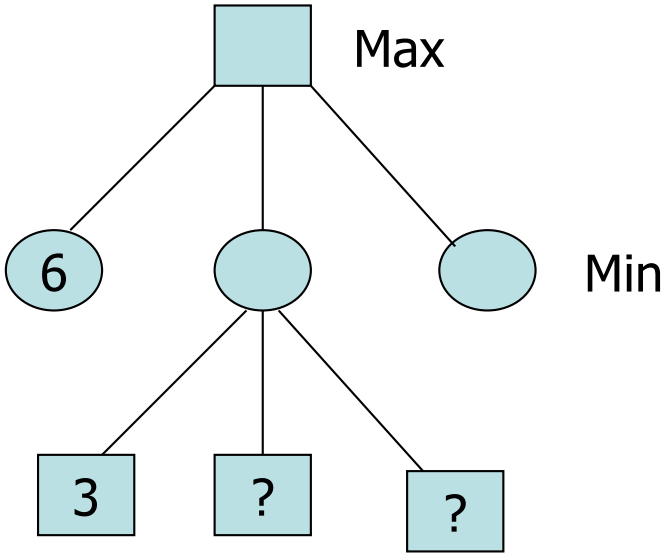
- Suppose at each step, a player has b choices
- Suppose m steps are needed in the worst case before the game finishes
- What's the running time for generating the best move?
  - Number of depth 1 nodes:
  - Number of depth 2 nodes:
  - …
  - Number of depth m nodes:
  - Total nodes searched:
- Time Complexity? Space Complexity?
- How to reduce the **searching time**?
  - Cut useless branches
  - Reduce searching depth



Game Tree Size

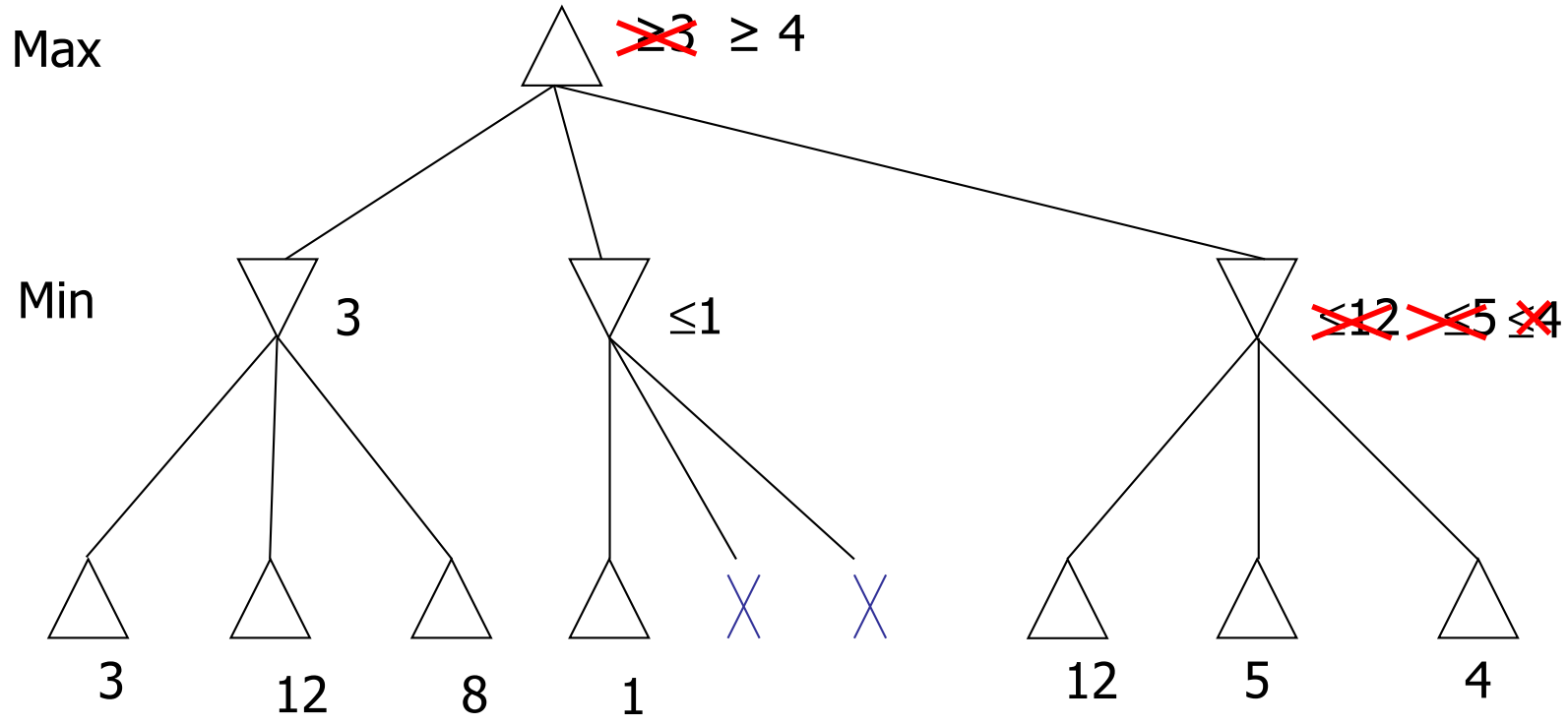
- Tic-Tac-Toe
  - b ≈ 5 legal actions per state on average, total of 9 plies in game.
  - 5^9 = 1,953,125
- Chess
  - b ≈ 35 (approximate average branching factor)
  - m ≈ 100 (depth of game tree for “typical” game)
  - $35^{100} \approx 10^{154}$!!
- It is usually impossible to develop the whole search tree.

α-β pruning



Do we need to know '?'

α-β pruning example



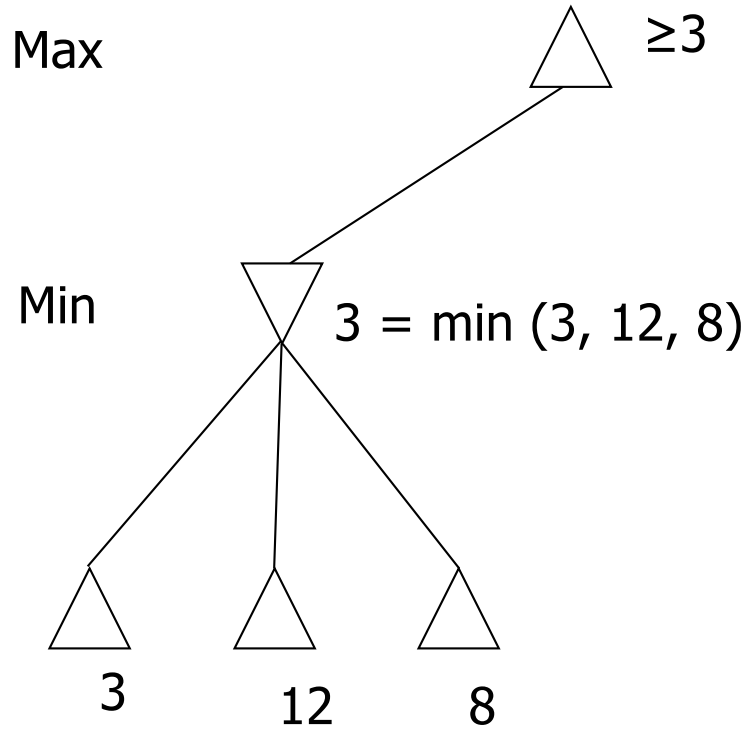
Alpha-Beta Pruning
- Exploiting the Fact of an Adversary

- **Bad** = not better than we already know we can achieve elsewhere
- **If a position is provably bad**
  - It is NO USE expending search time to find out exactly how bad, if you have a better alternative
- **If the adversary can force a bad position**
  - It is NO USE expending search time to find out the good positions that the adversary won't let you achieve anyway
- **Contrast normal search**
  - ANY node might be a winner.
  - ALL nodes must be considered.

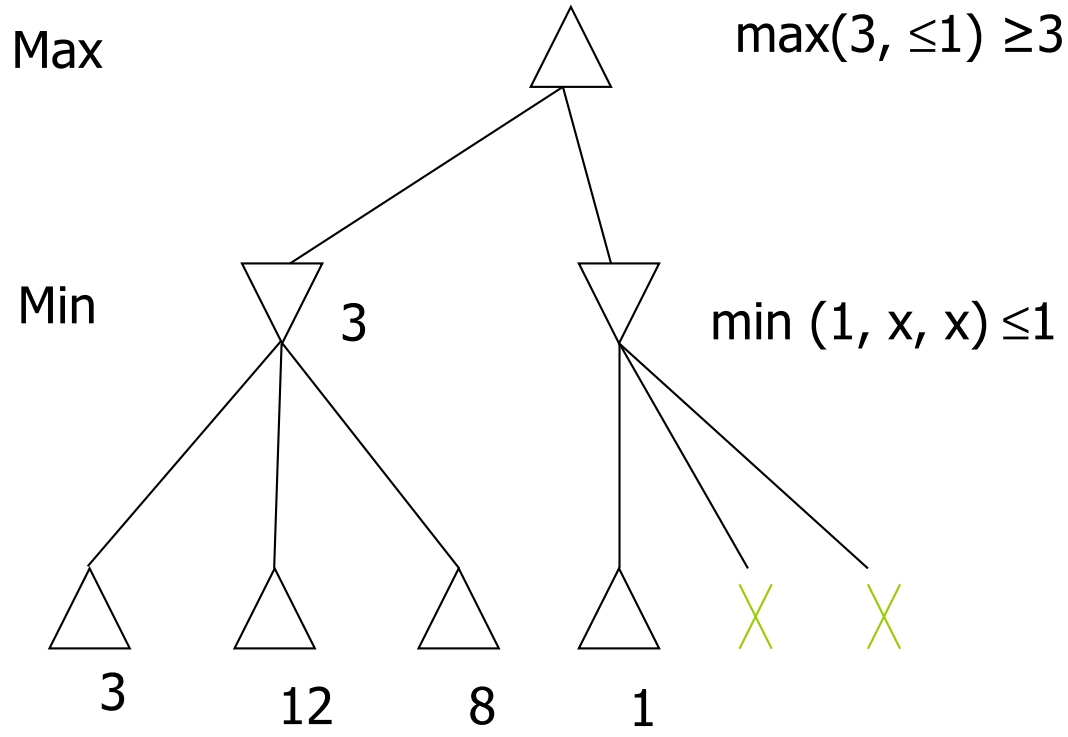
Alpha Beta Procedure

- **Idea**
  - Do **depth first search** to generate partial game tree,
  - Give evaluation function to leaves,
  - Compute bound on internal nodes.
- **Update α-β bounds**
  - **α** value for max node means that max real value is at least **α**.
  - **β** for min node means that min can guarantee a value no more than **β**
- **Prune whenever α ≥ β**
  - Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.
  - Prune below a Min node whose beta value becomes less than or equal to the alpha value of its ancestors.

α-β pruning example



α-β pruning example



The α-β Pruning Algorithm

function MINIMAX-DECISION($state$) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT($state$, a))

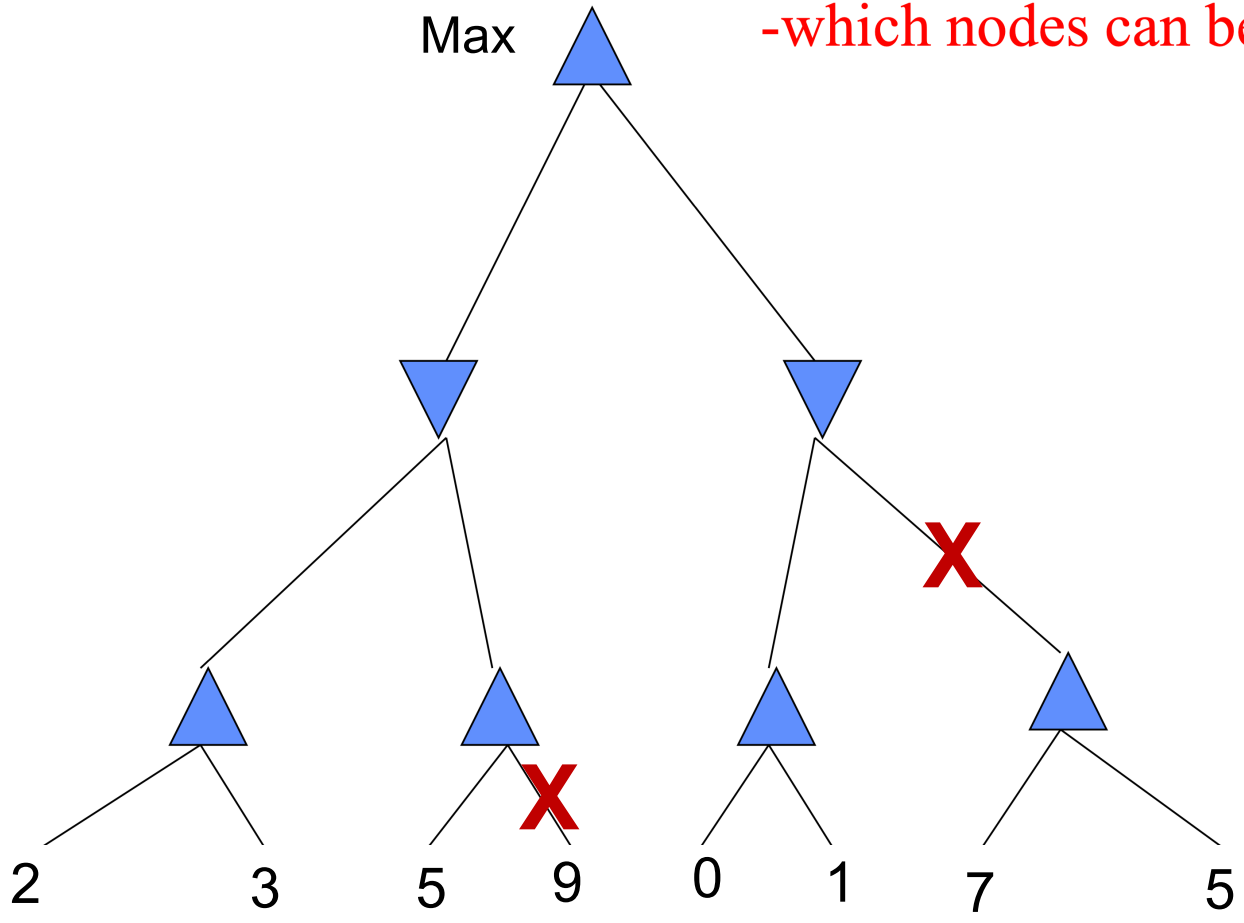
---

function MAX-VALUE($state$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

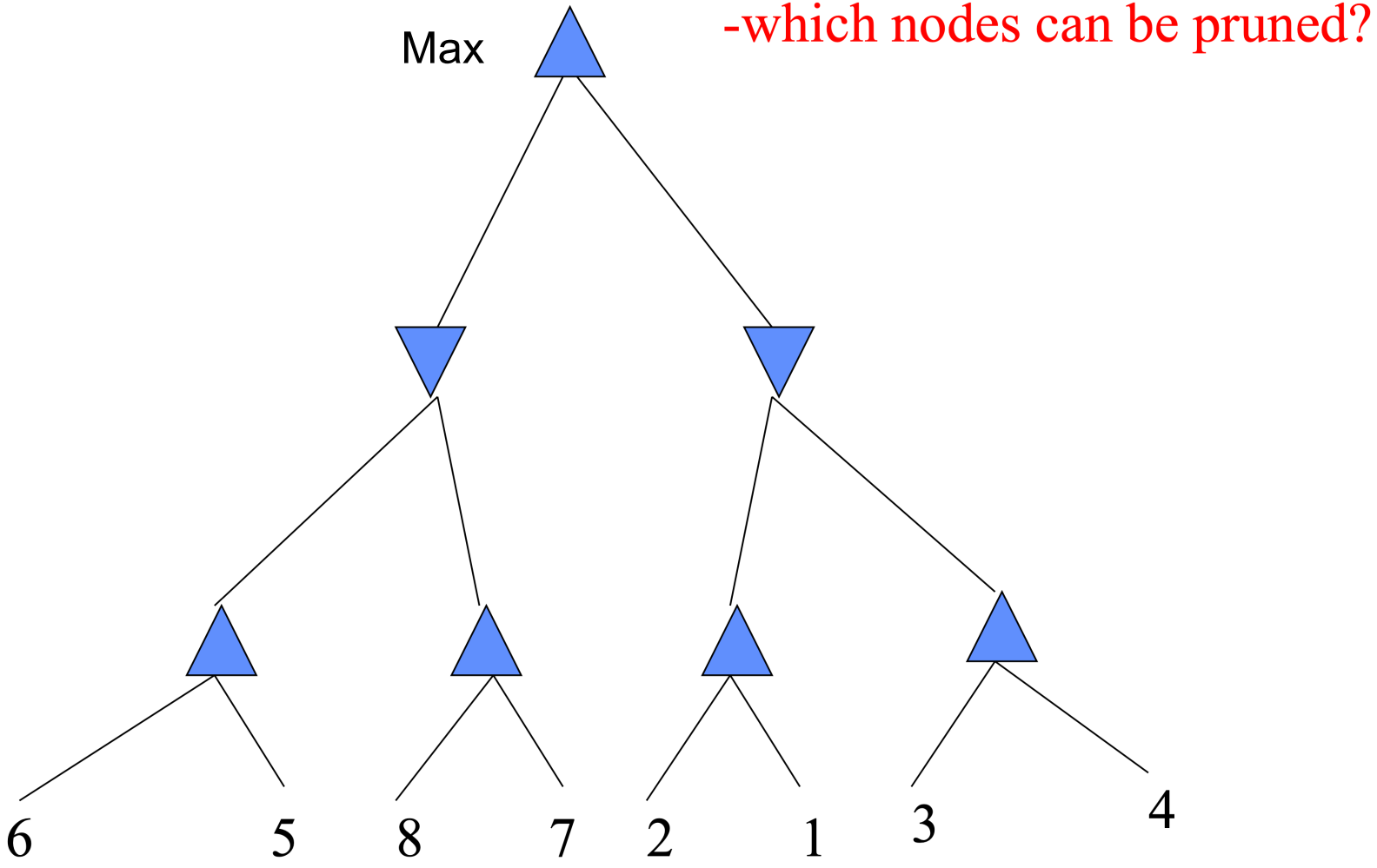
---

function MIN-VALUE($state$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow \infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

α-β Pruning Exercise 1



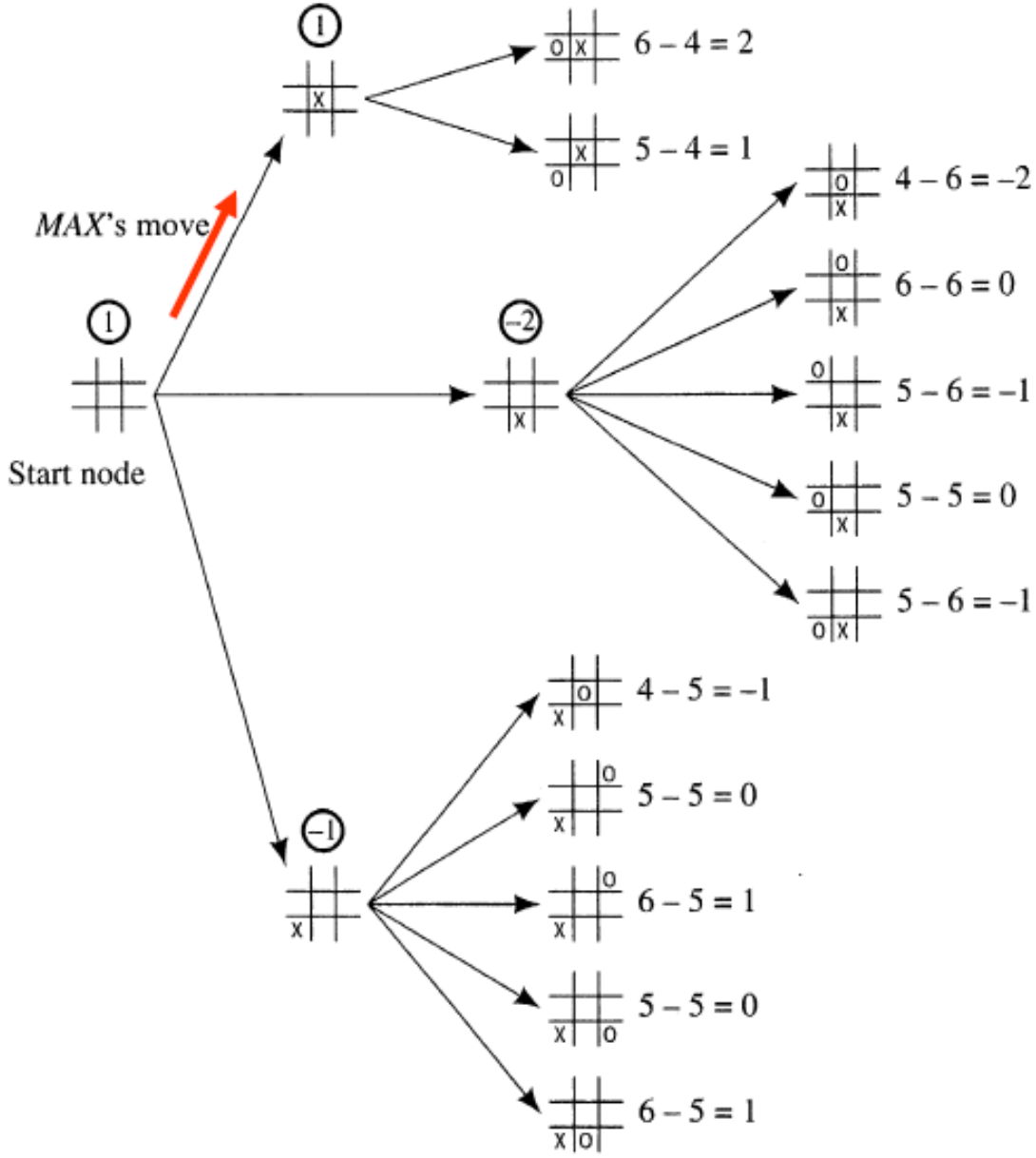
α-β Pruning Exercise 2



The α-β Pruning Effectiveness

- Worst-Case
  - Branches are ordered so that no pruning takes place. In this case alpha-beta gives no improvement over exhaustive search
- Best-Case
  - Each player's best move is the left-most alternative (i.e., evaluated first)
  - In practice, performance is closer to best rather than worst-case
- In practice it is often b(2m/3)

Reducing searching depth



- For games like chess, it is hardly possible for you to search to a finishing state
- Evaluation function for the game status
- In tic tac toe, we can use
  - the number of possible winning lines for X - the number of possible winning lines for O

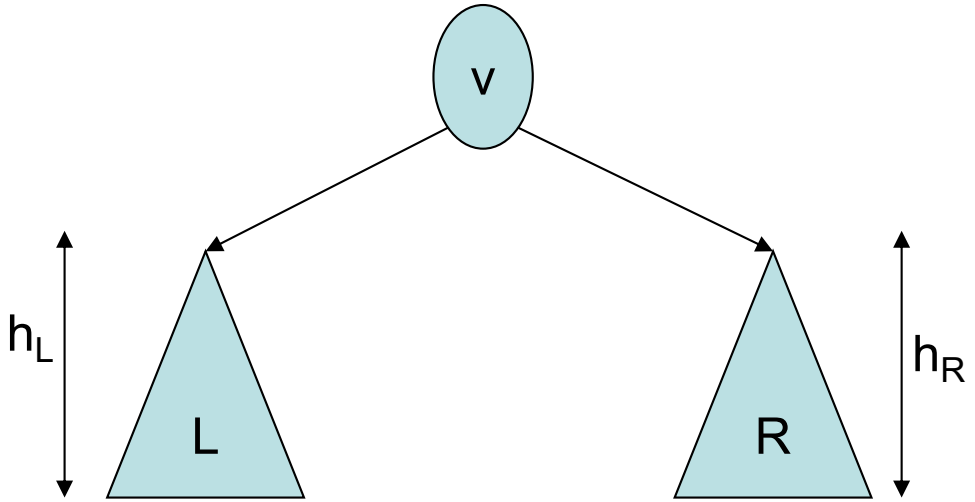
AVL Trees

- Oldest & most well-known balanced binary search tree (BST)
- Balancing Condition: For each node *v*, the difference between the height of its left subtree and the height of its right subtree ≤ 1
- Having this balancing condition will keep the tree height to be O(logn). This implies fast operation time
- How to maintain the balancing condition after insertions & deletions? By *rotations*!

AVL Trees

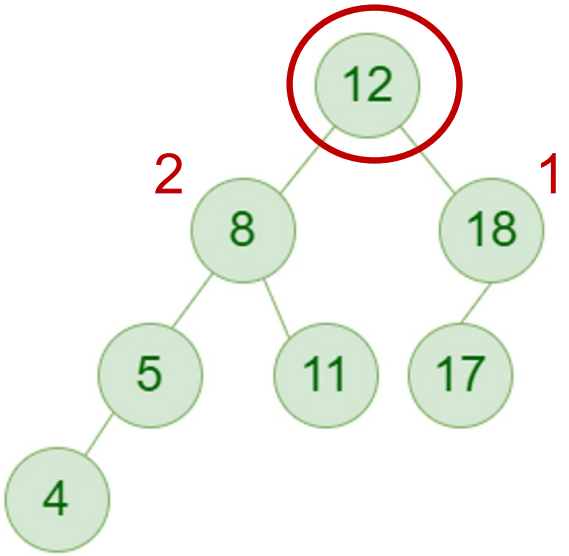
- Height of a tree = number of edges on a longest root-to-leaf path
- Note: height of the tree below is

$$= \max(h_L, h_R) + 1$$



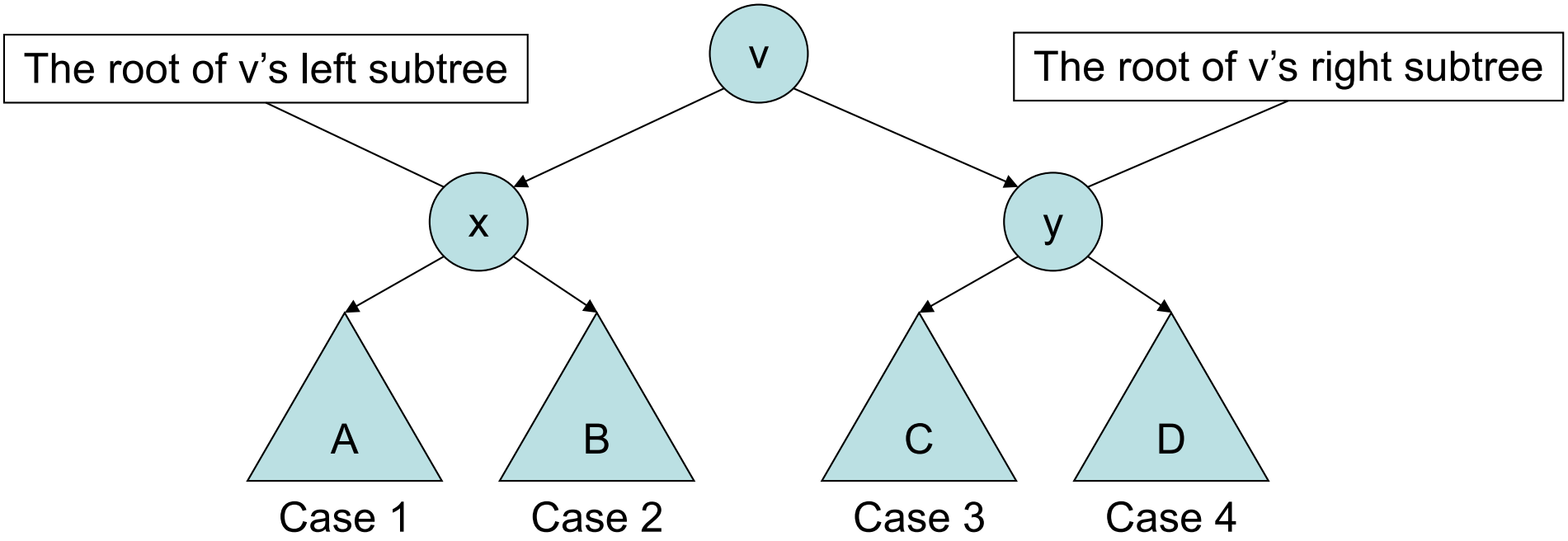
AVL Trees Example

- For each node *v*, the difference between the height of its left subtree and the height of its right subtree ≤ 1



Insertion

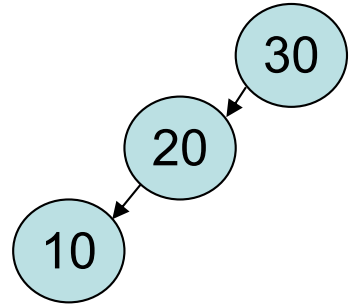
- Consider insert(u): only nodes along the path from root to the point of insertion may be unbalanced
- Suppose node *v* unbalanced, 4 cases:
  - Cases 1 & 4 are mirror image symmetries with respect to v
  - Cases 2 & 3 are mirror image symmetries with respect to v



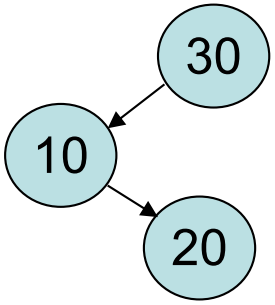
Insertion Example

Insert Inputs: {10, 20, 30}

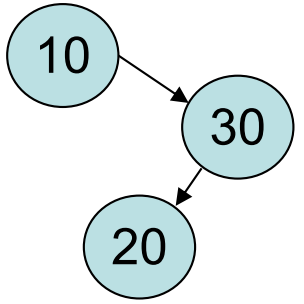
Insert Inputs with order: 30, 20, 10



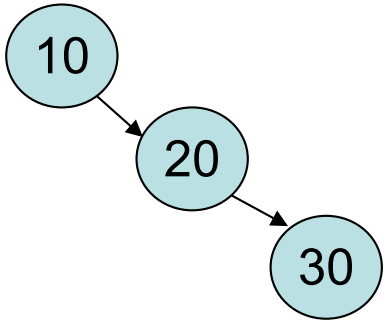
Insert Inputs with order: 30, 10, 20



Insert Inputs with order: 10, 30, 20

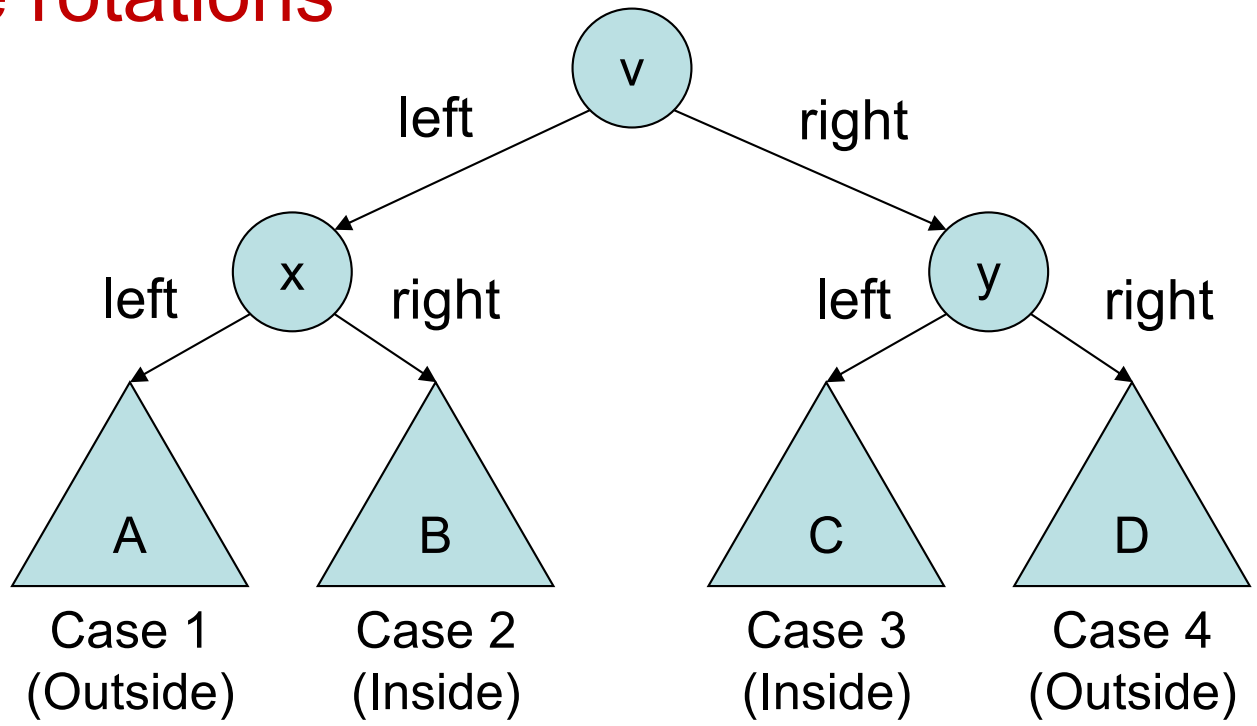


Insert Inputs with order: 10, 20, 30



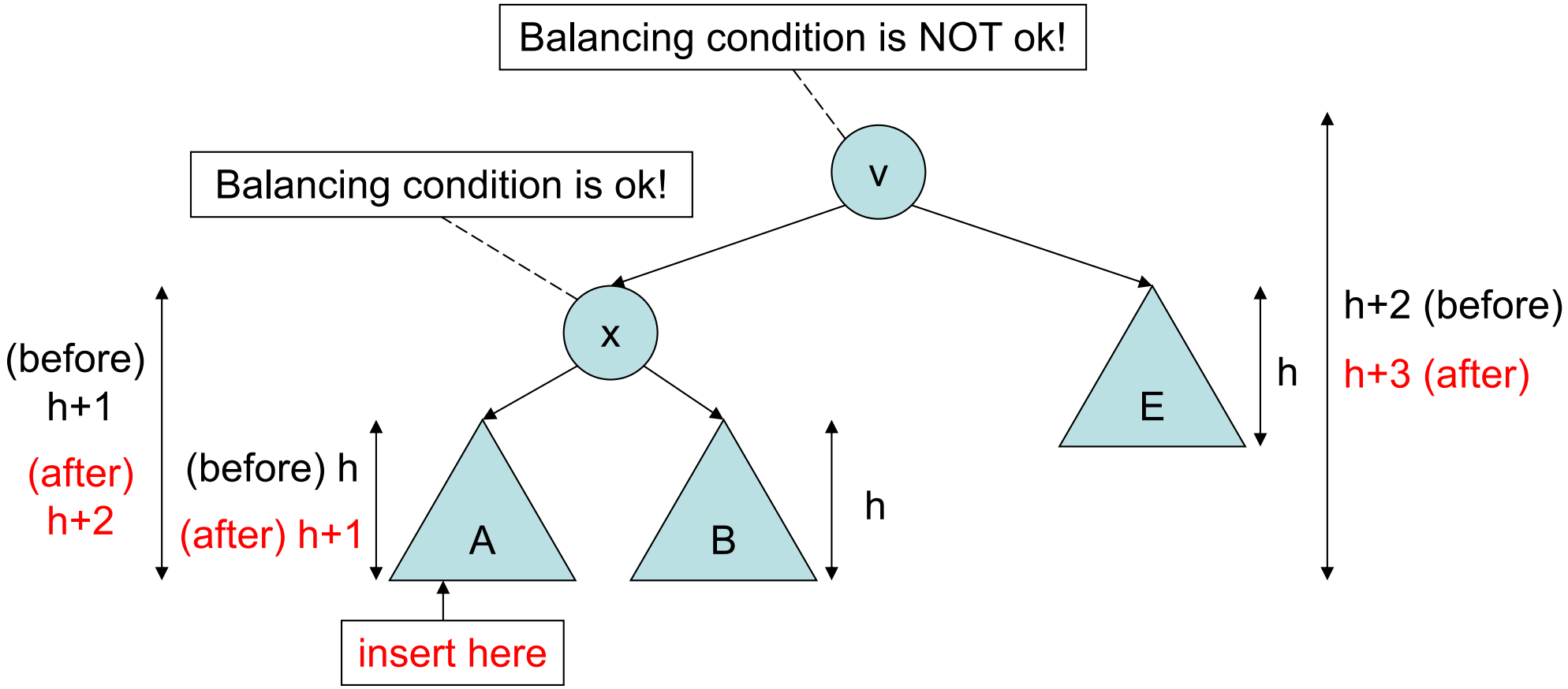
Insertion

- In Case 1 (or Case 4), the insertion occurs on the "outside" (i.e., left-left or right-right); it is fixed by a single rotation
- In Case 2 (or Case 3), the insertion occurs on the "inside" (i.e., left-right or right-left); it is handled by double rotations



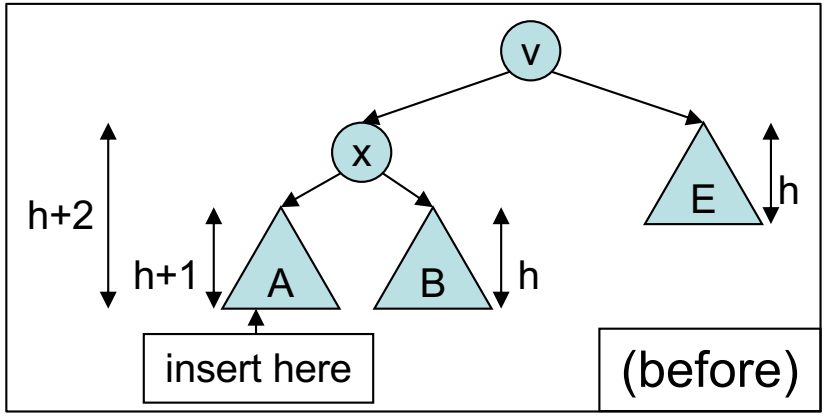
Insertion – Case 1

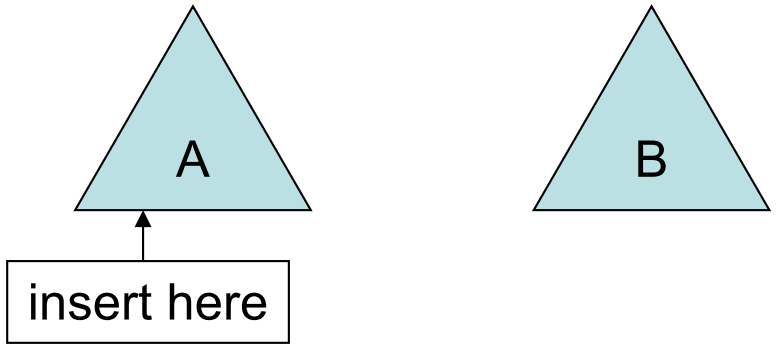
- Before insertion, height of subtree A = *h* and height of subtree E = *h*
- After insertion, height of subtree A = *h*+1

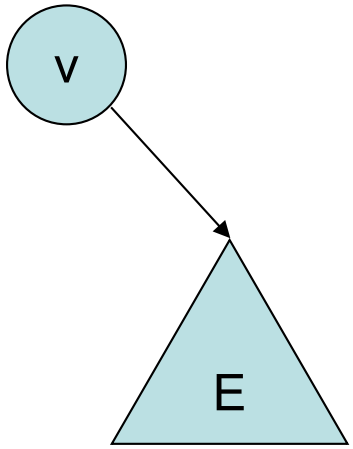


Insertion – Case 1

- Solution: single rotation, i.e., make x as root

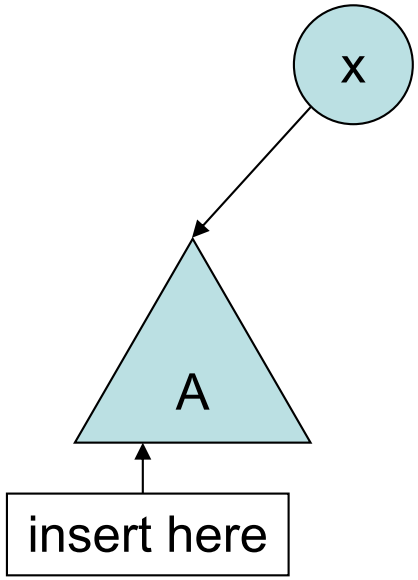


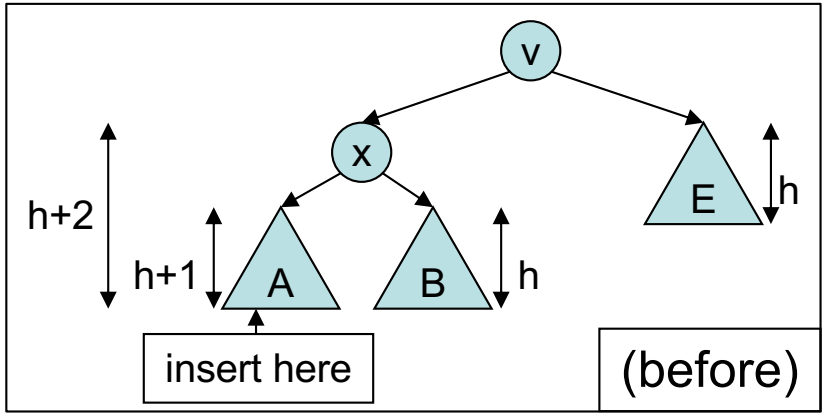


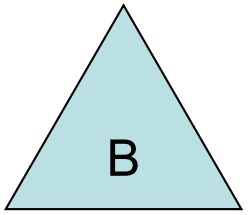


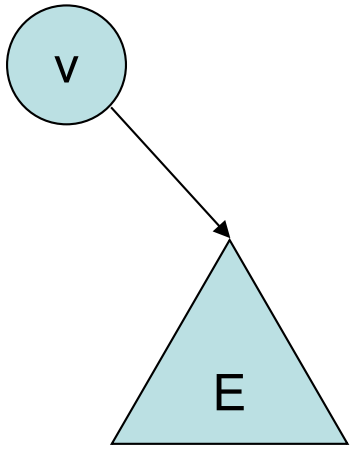
Insertion – Case 1

- Only A can be x's left subtree



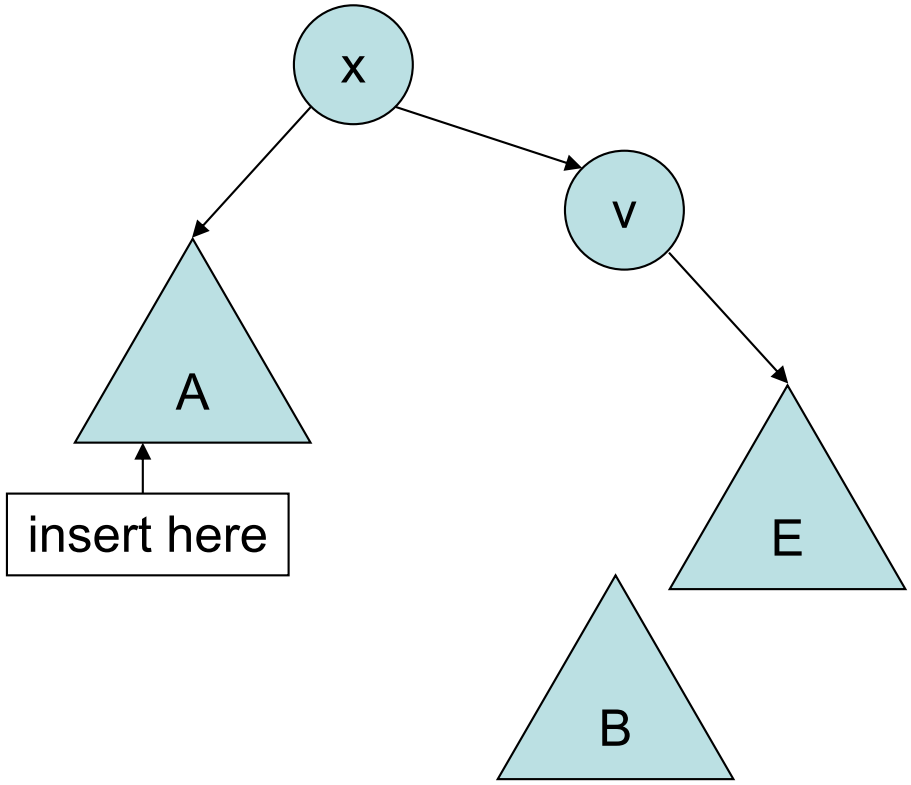


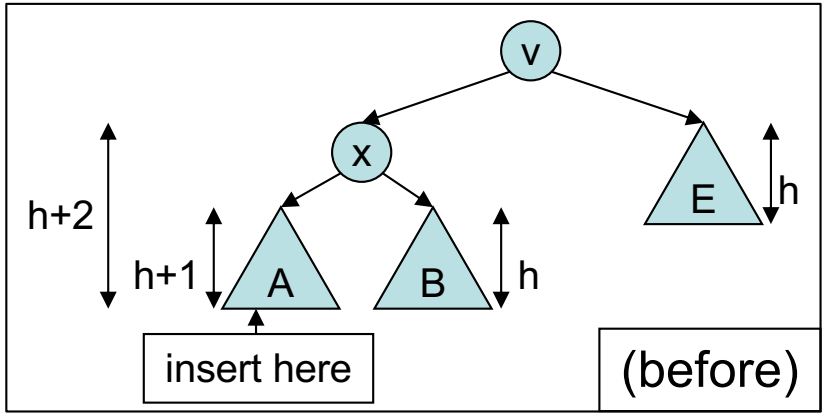




Insertion – Case 1

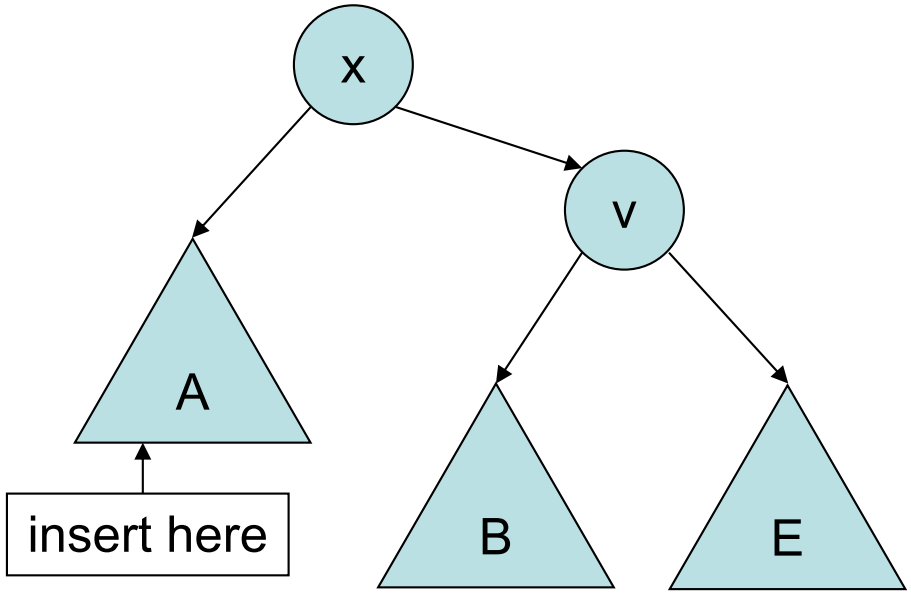
- v must be x's right subtree

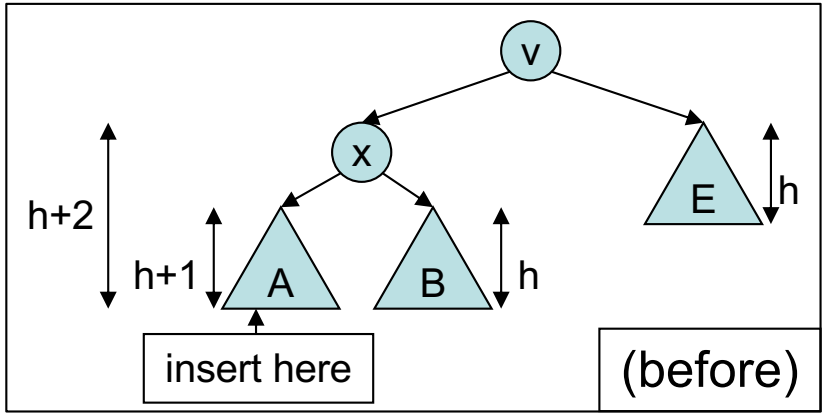




Insertion – Case 1

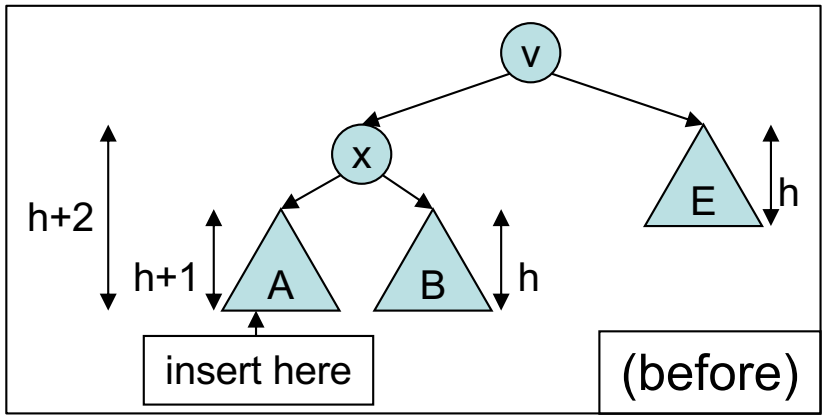
- B must be v's left subtree

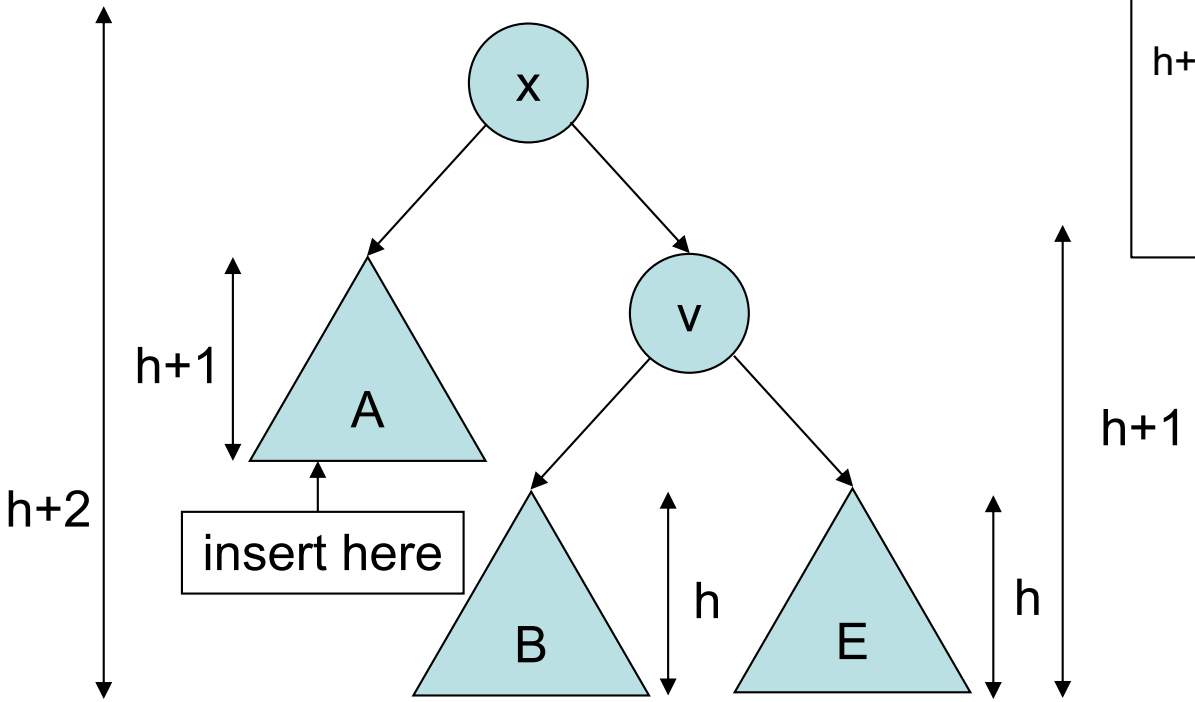




Insertion – Case 1

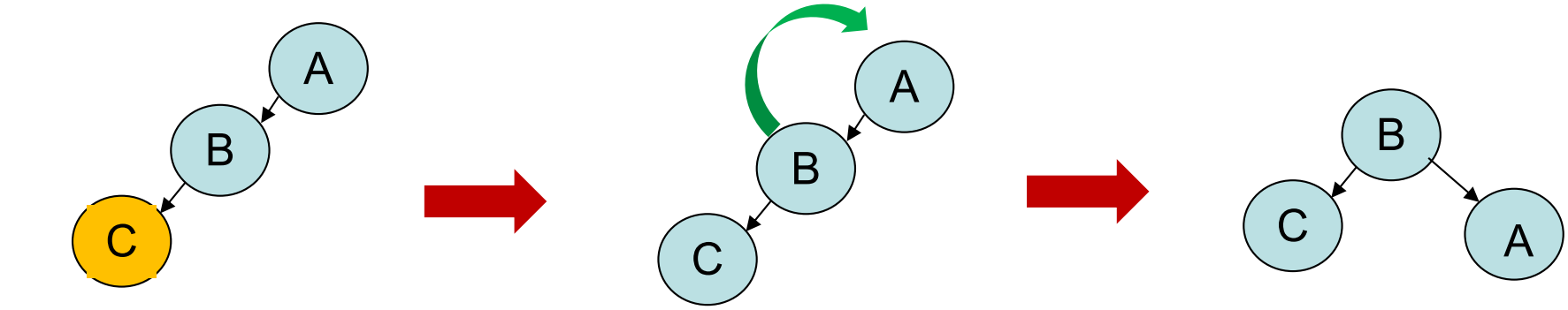
- After rotation: height of x = h+2
  (= height of subtree rooted at v before insertion)
- The left and right subtrees of x have the same height





Insertion – Case 1 RotateL

- If a node is added to **the left subtree of the left subtree**, the AVL tree may get out of balance, we do a single rotation (called Rotate Left Subtree or right Rotation).

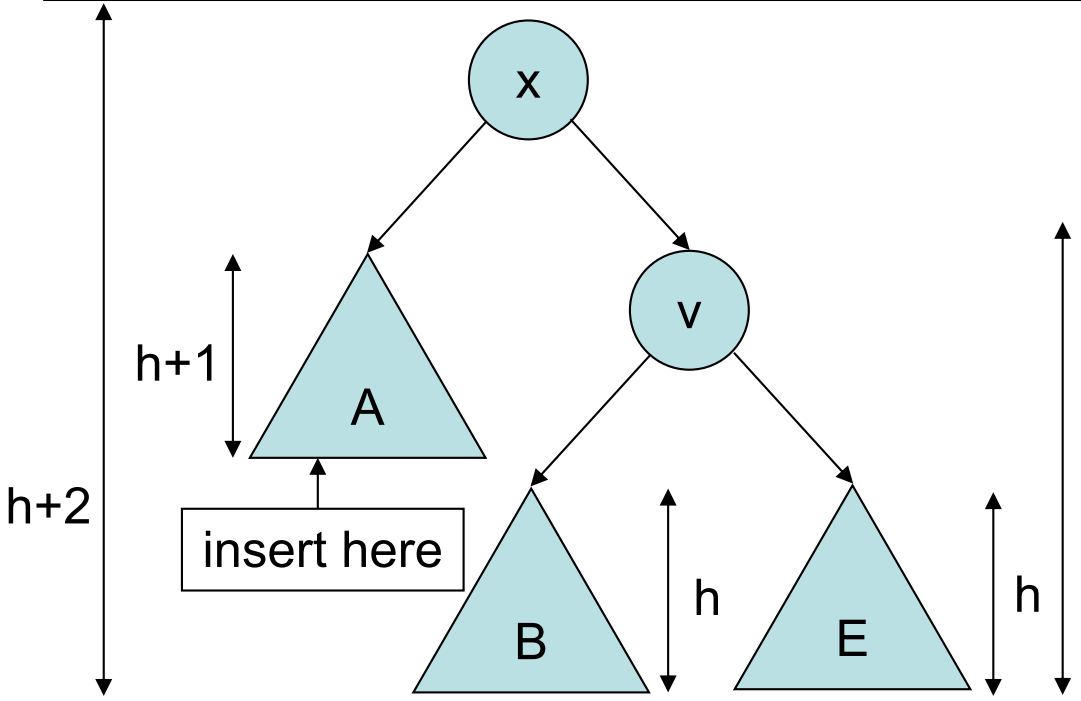


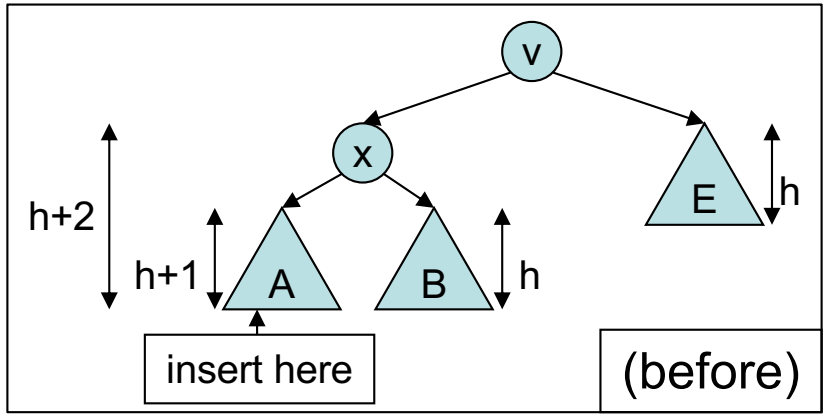
Left Unbalanced Tree Right Rotation Balanced

Insertion – Case 1

```
void rotateL(Node*& v) // v & v->lson rotate
{
  Node* x=v->lson;
  v->lson=x->rson;
  x->rson=v;
  v->height=max( h(v->lson), h(v->rson) )+1;
  x->height=max( h(x->lson), v->height )+1;
  v=t;
}
```

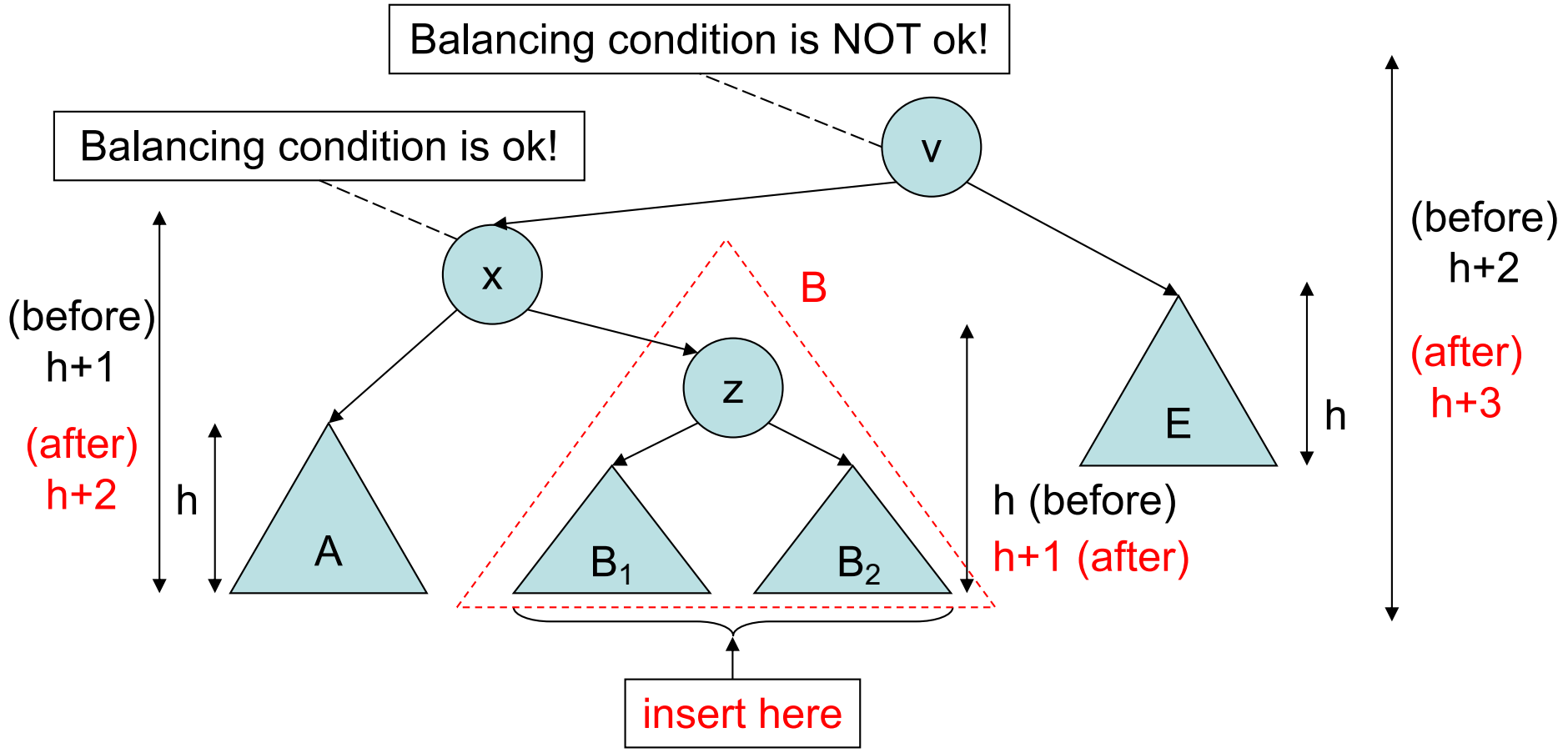
Make x as the root





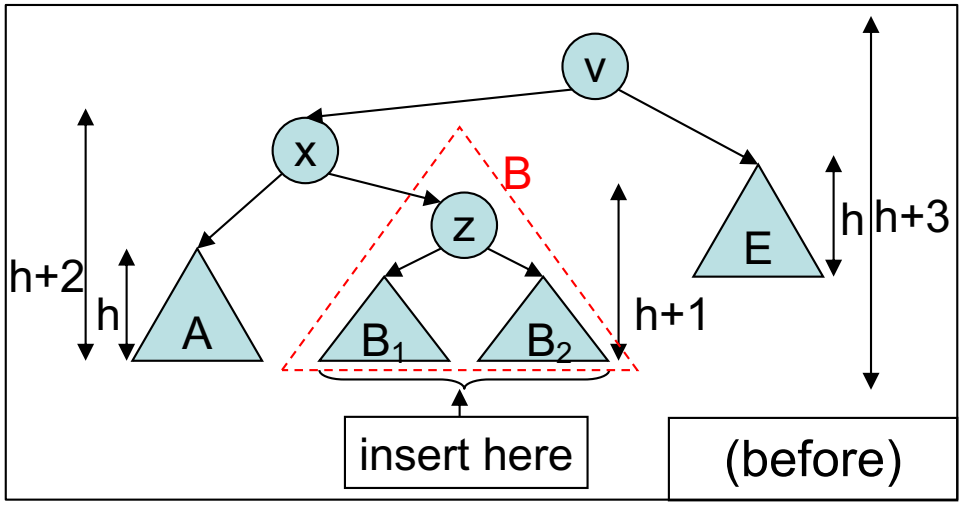
Insertion – Case 2

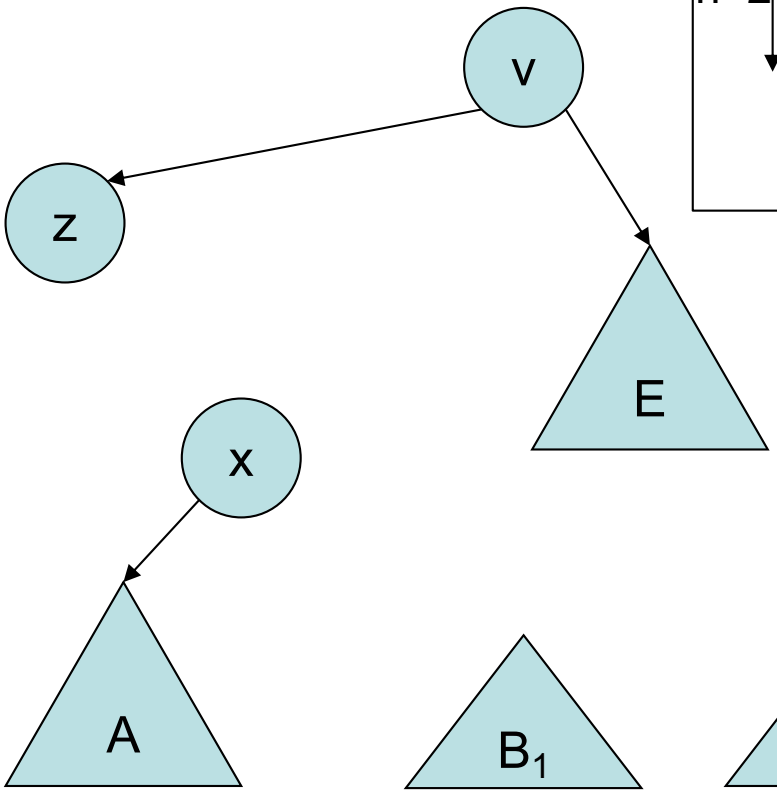
- Before insertion, height of subtree B = h and height of subtree E = h
- After insertion, height of subtree B = h+1, either B1 or B2 has height h, the other h-1



Insertion – Case 2

- Solution: double rotation, i.e., make z as root
- 1st rotation: make z as the root of its parent's subtree, i.e., its parent is x

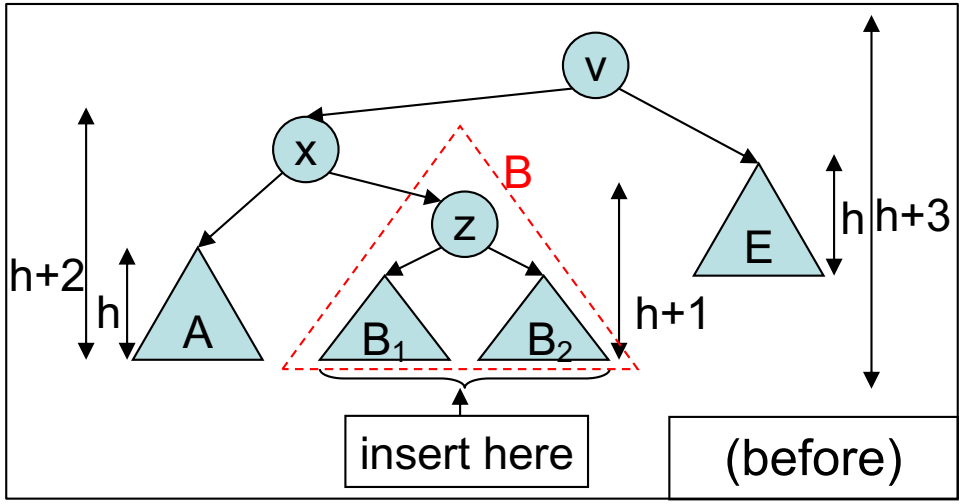


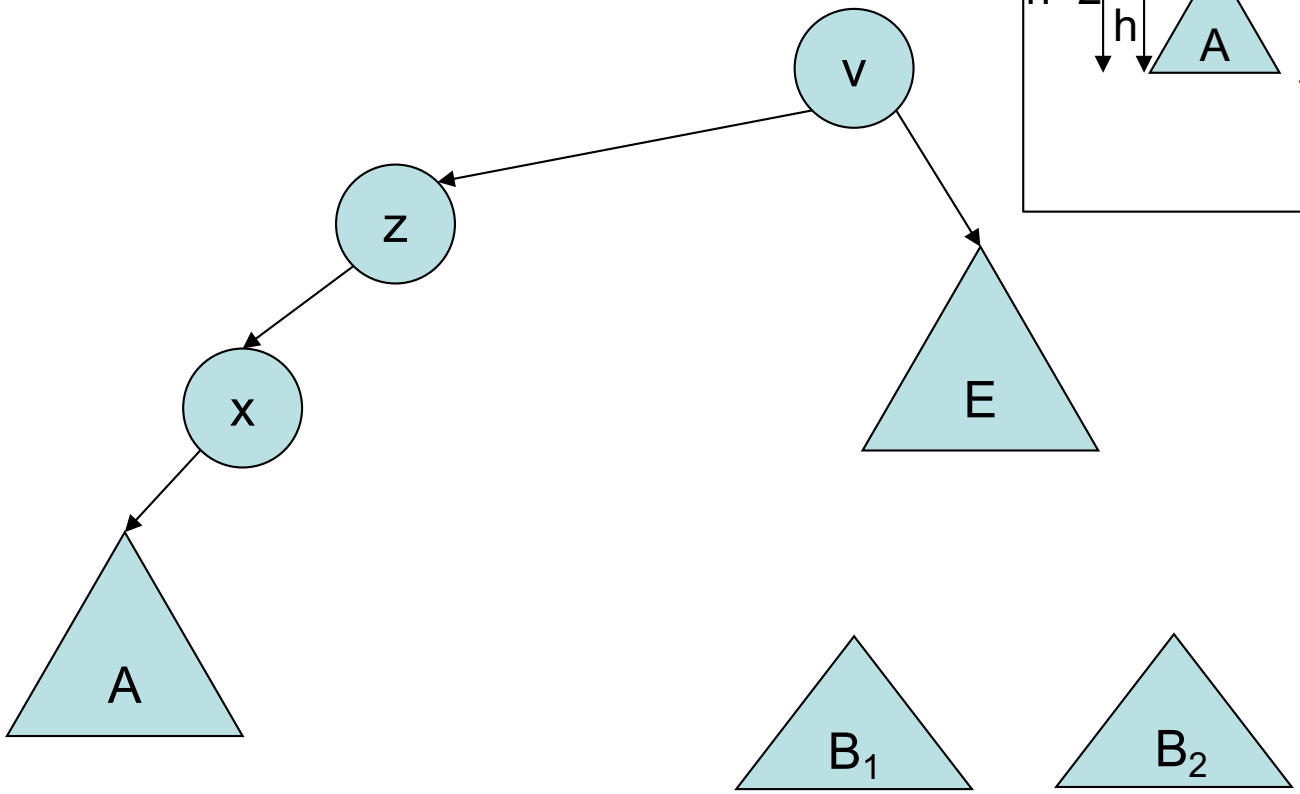



```
void dbl_rotateL(Node*& s)
{
    rotateR(s->lson);
    rotateL(s);
}
```

Insertion – Case 2

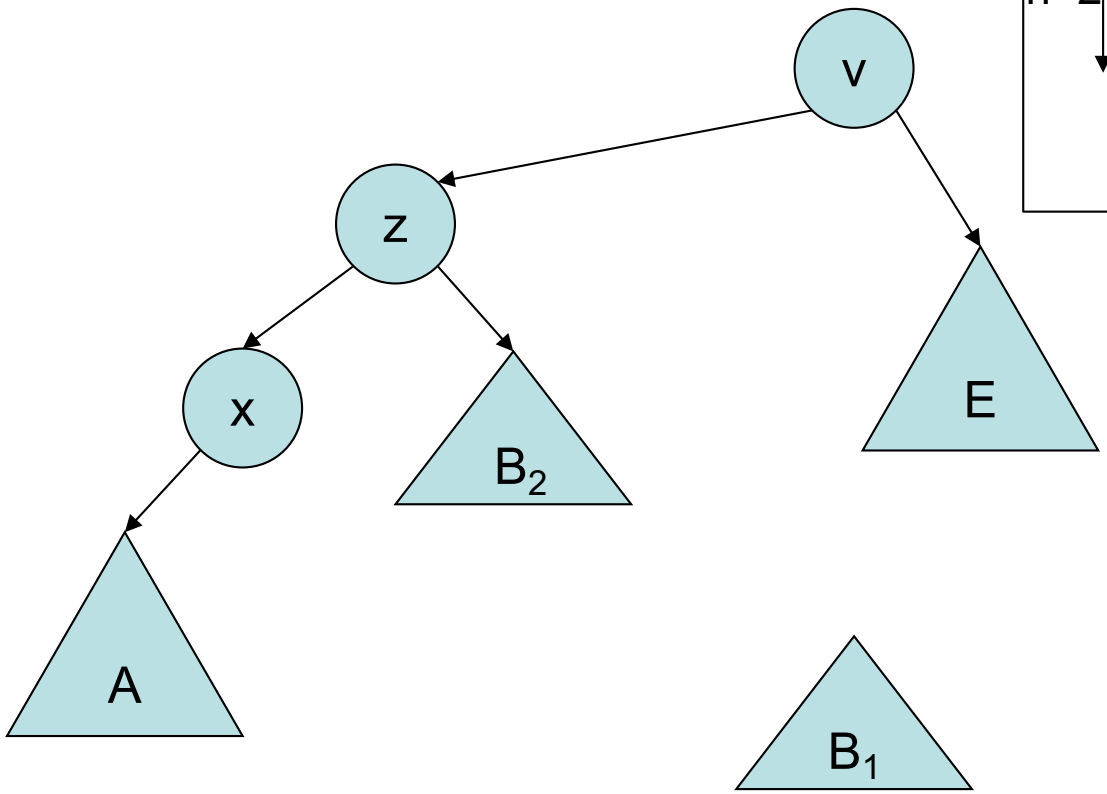
- 1st rotation: x must be z's left son

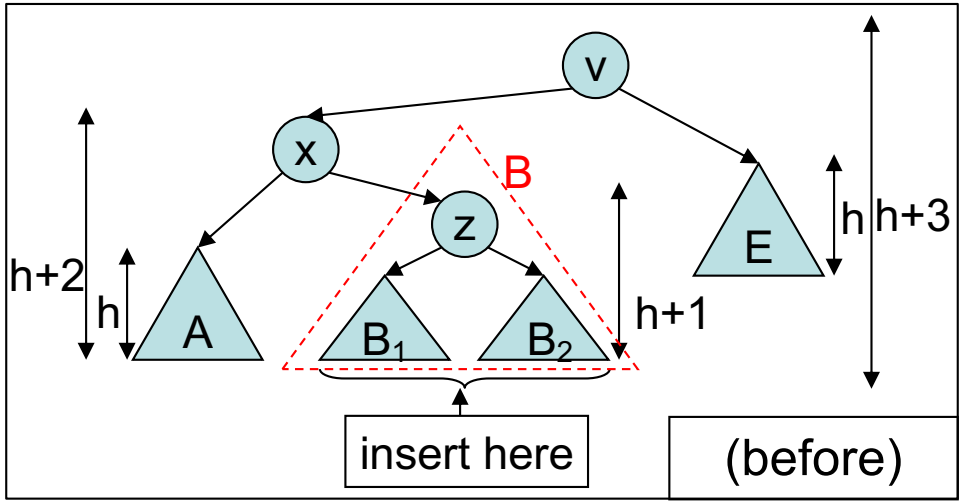




Insertion – Case 2

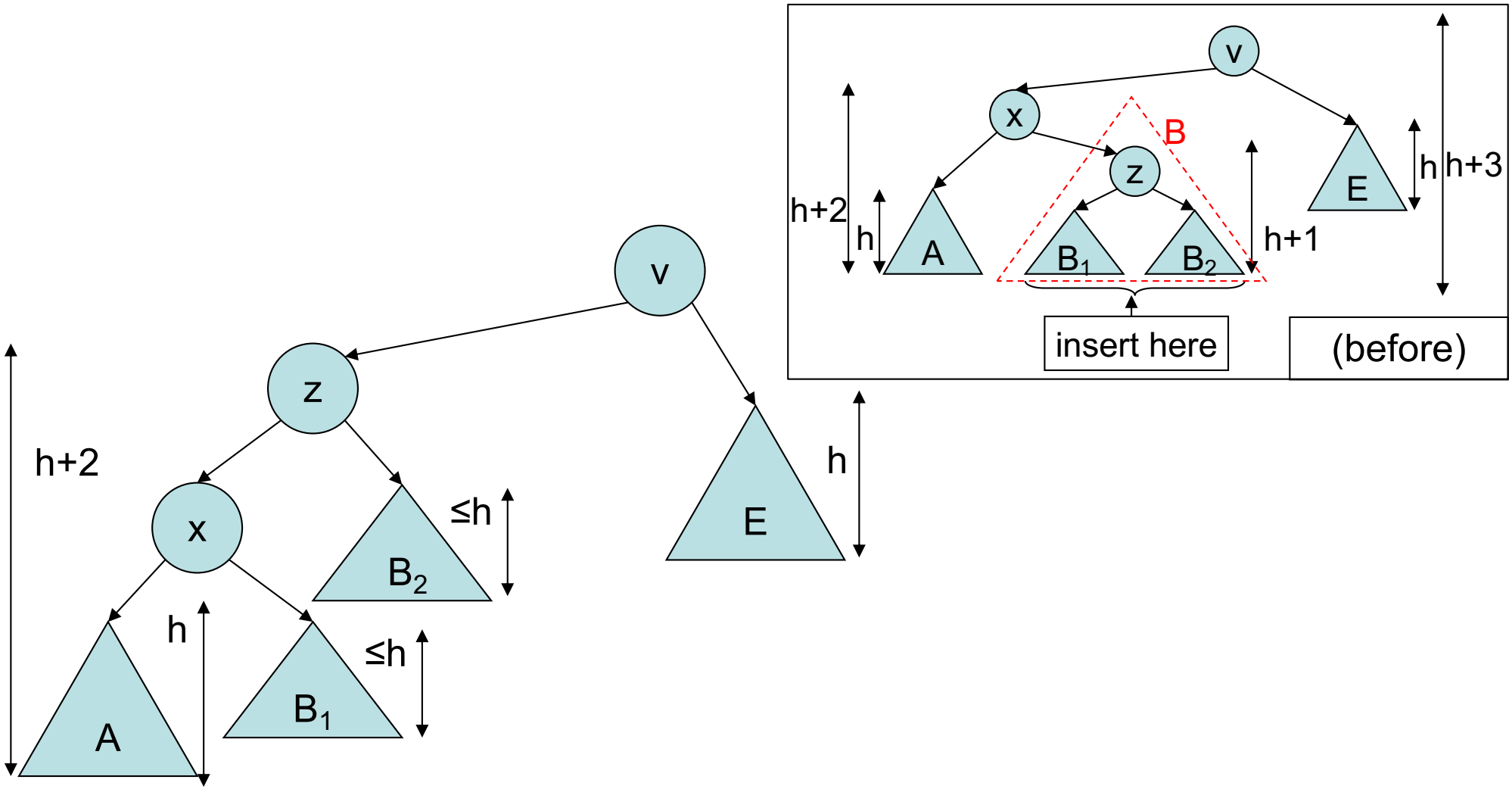
- 1st rotation: B_2 must be z's right subtree





Insertion – Case 2

- 1st rotation: B_1 must be x's right subtree



Insertion – Case 2

```
void rotateR(Node*& s)  //s & s->rson rotate
{
  Node* t=s->rson;
  s->rson=t->lson;
  t->lson=s;
  s->height=max( h(s->lson), h(s->rson) )+1;
  t->height=max( h(t->rson), s->height )+1;
  s=t;
}
```

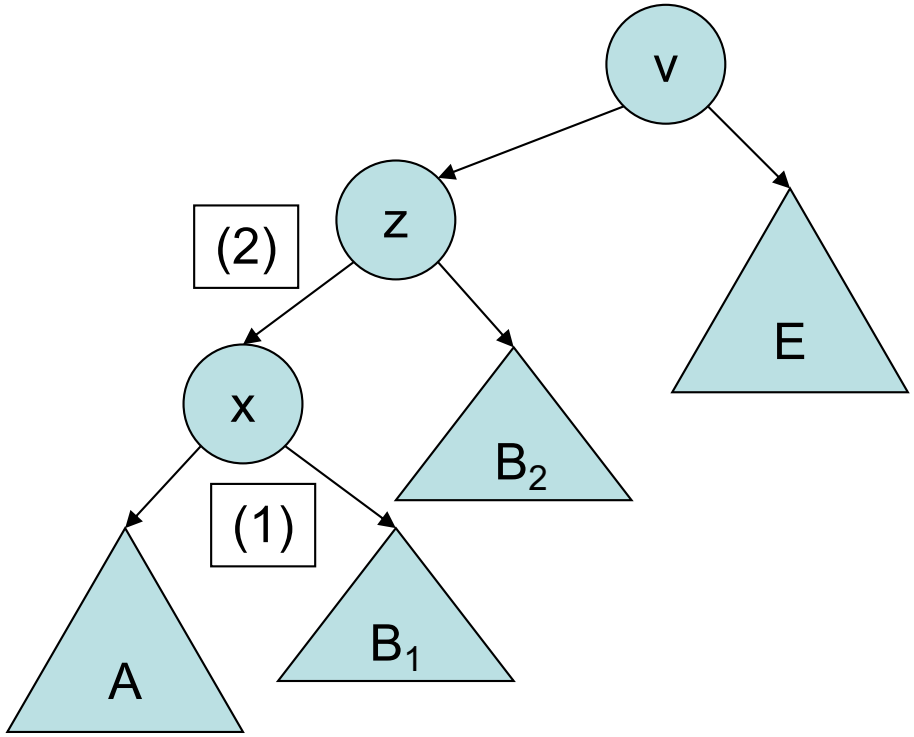
z

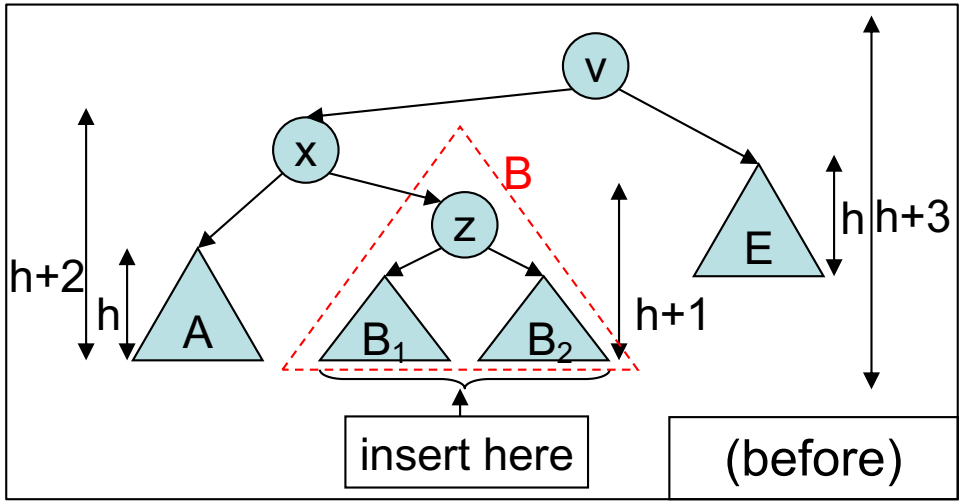
x

(1)

(2)

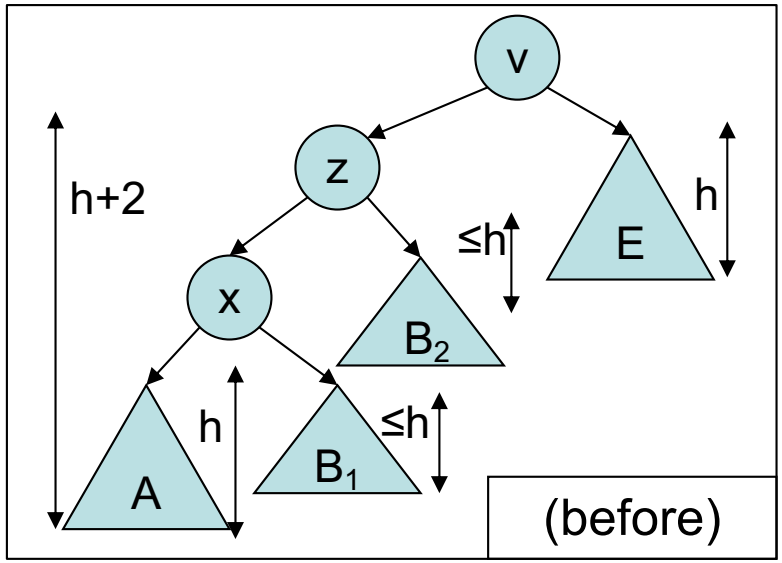
Make z as the root

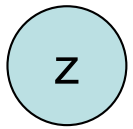




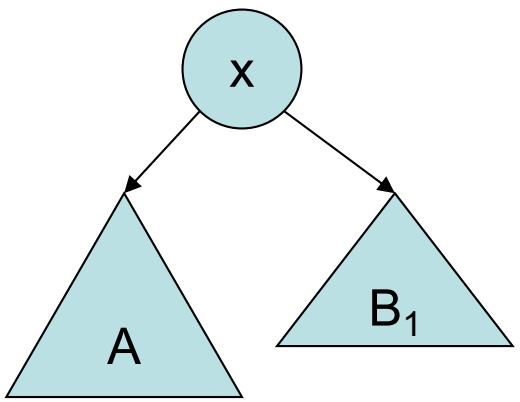
Insertion – Case 2

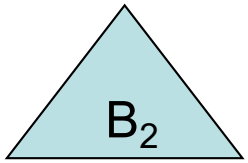
- 2nd rotation: make z as the root

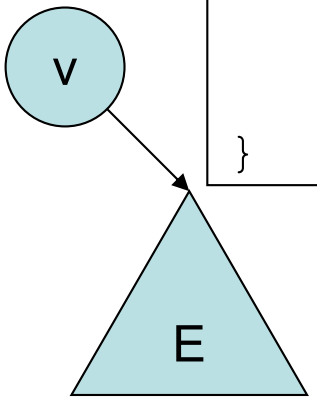





```
void dbl_rotateL(Node*& s)
{
    rotateR(s->lson);
    rotateL(s);
}
```

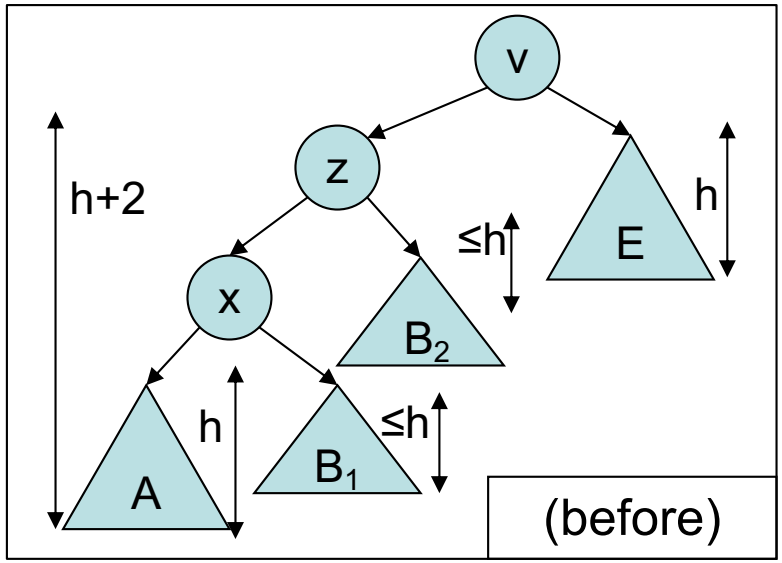


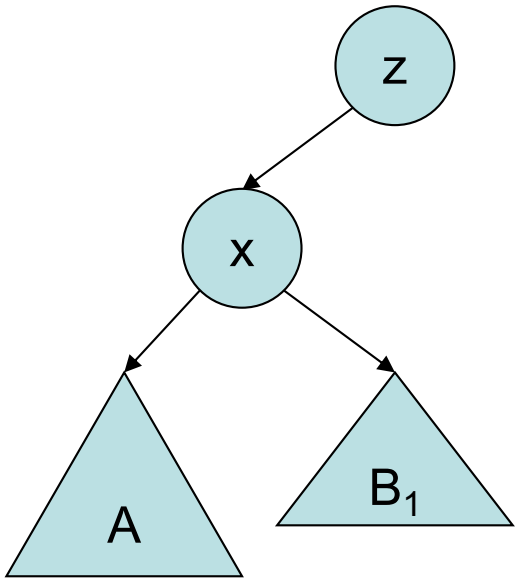


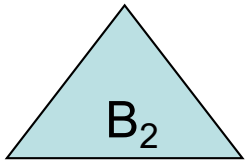


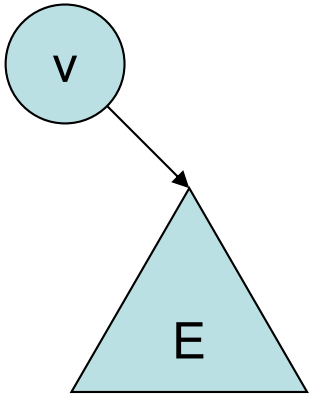
Insertion – Case 2

- 2nd rotation: x must be z's left subtree



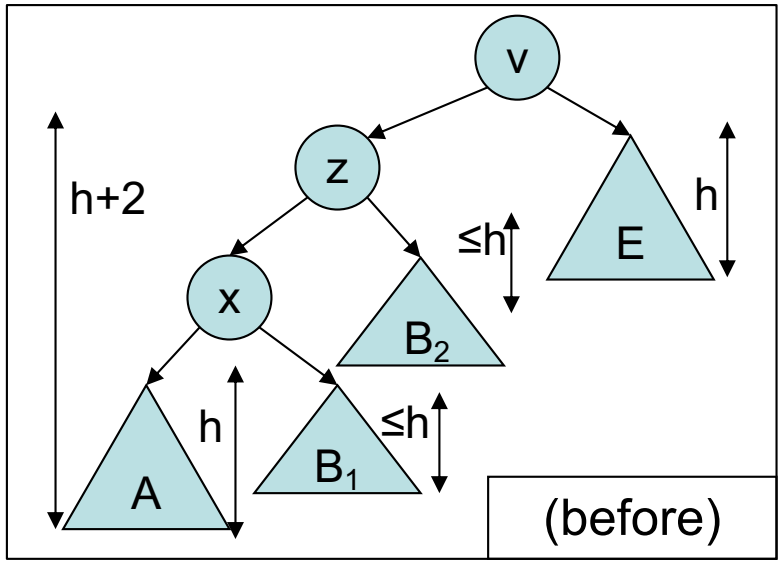


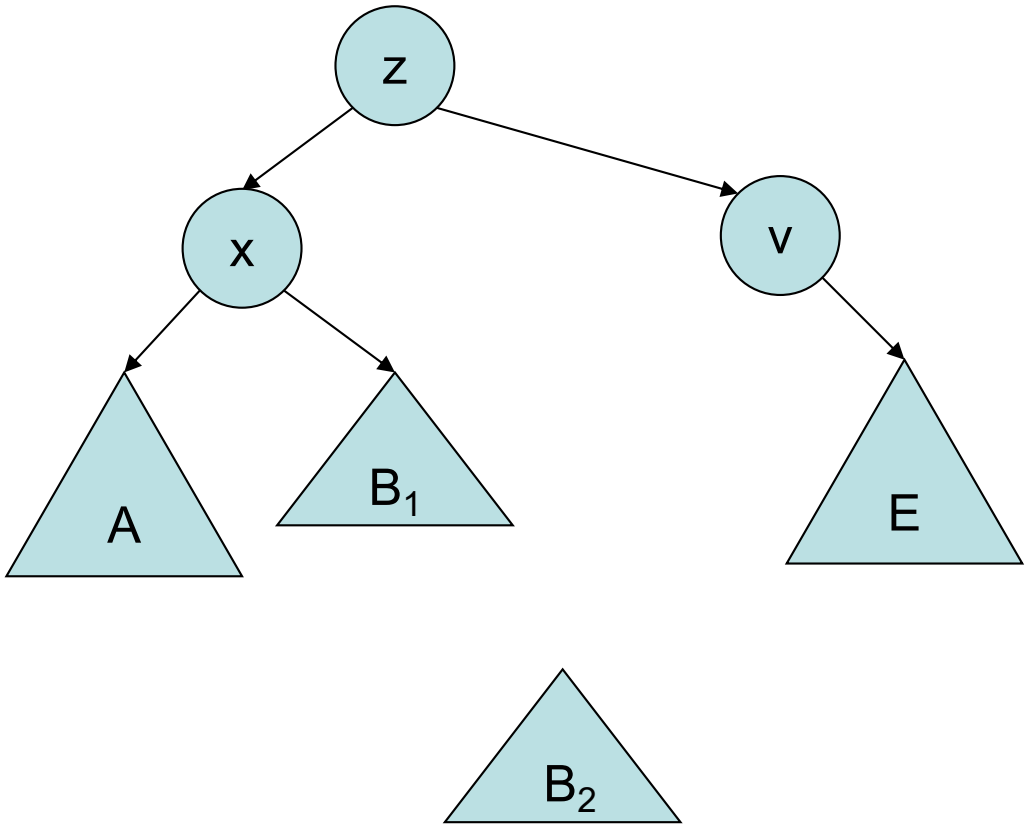




Insertion – Case 2

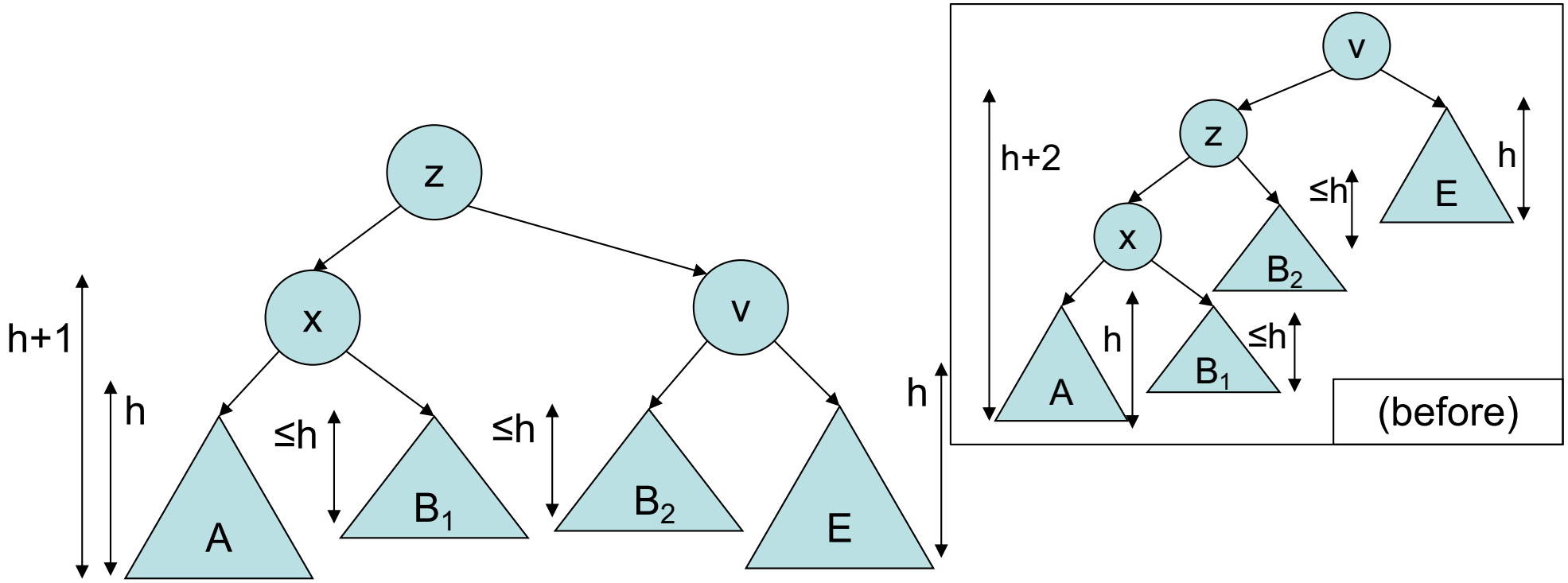
- 2nd rotation: v must be z's right subtree





Insertion – Case 2

- 2nd rotation: B_2 must be v's left subtree



- After rotation: height of z = h+2 (= height of subtree rooted at v before insertion)
- The left and right subtrees of z have the same height

Insertion – Case 2

```
void rotateL(Node*& s) // s & s->lson rotate
{
  Node* t=s->lson;
  s->lson=t->rson;
  t->rson=s;
  s->height=max( h(s->lson), h(s->rson) )+1;
  t->height=max( h(t->lson), s->height )+1;
  s=t;
}
```

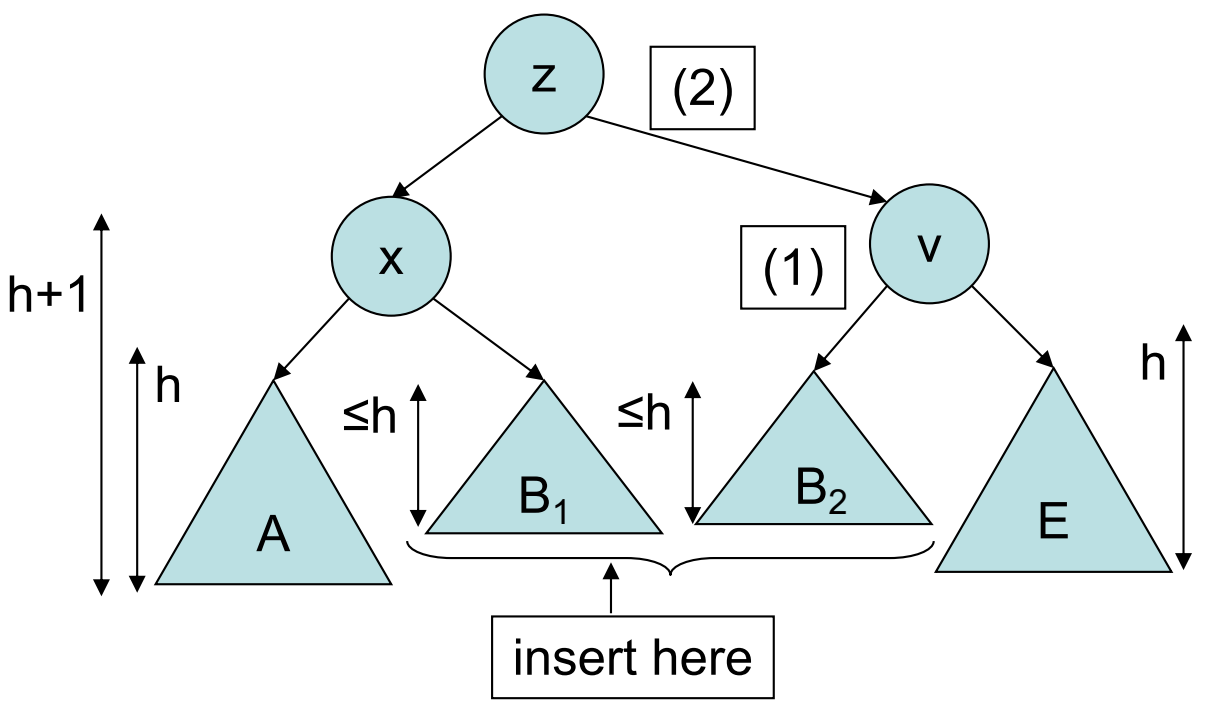
v

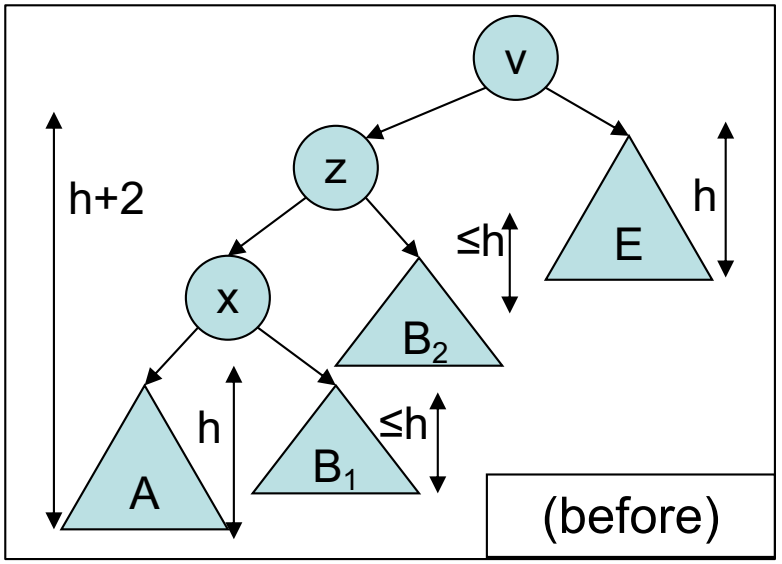
z

(1)

(2)

Make z as the root





Overall Scheme for insert(*x*)

- Search for *x* in the tree; insert a new leaf for *x* (as in previous BST)
- If parent of *x* not balanced, perform single or double rotation as appropriate
  - How do we know the height of a subtree?
  - Have a “height” attribute in each node
- Set *x* = parent of *x* and repeat the above step until *x* = root

```
struct Node
{
   Node(item x):data(x), height(1),
                lson(NULL), rson(NULL){}
   item data;
   int height;  // height of subtree rooted at this
                // node
   Node* lson;
   Node* rson;
};

int h(Node* t){
   return t==NULL ? 0 : t->height;
}

int max(int x, int y){
   return x>=y ? x : y;
}
```

```
void AVL::insertR(Node*& t, item x)
{
  if (t==NULL) {t==new Node(x); return; }
  else if (x < t->data)
  {
    insertR(t->lson, x);                    // insert
    if (h(t->lson)==h(t->rson)+2)           // checking
      if (x < t->lson->data) rotateL(t);    // case 1
      else dbl_rotateL(t);                  // case 2
  }
  else if (x > t->data)
  {
    insertR(t->rson, x);                     // insert
    if (h(t->rson)==h(t->lson)+2)            // checking
      if (x > t->rson->data) rotateR(t);     // case 4
      else dbl_rotateR(t);                   // case 3
  }
  else ;  // duplication
  t->height = max( h(t->lson), h(t->rson) ) + 1;
}
```

```
void rotateL(Node*& s) // s & s->lson rotate (Case 1)
{
  Node* t=s->lson;
  s->lson=t->rson; t->rson=s;
  s->height=max( h(s->lson), h(s->rson) )+1;
  t->height=max( h(t->lson), s->height )+1;
  s=t;
}

void rotateR(Node*& s)  //s & s->rson rotate (Case 4)
{
  Node* t=s->rson;
  s->rson=t->lson; t->lson=s;
  s->height=max( h(s->lson), h(s->rson) )+1;
  t->height=max( h(t->rson), s->height )+1;
  s=t;
}
```

```
void dbl_rotateL(Node*& s) // (Case 2)
{
   rotateR(s->lson);
   rotateL(s);
}

void dbl_rotateR(Node*& s) // (Case 3)
{
   rotateL(s->rson);
   rotateR(s);
}
```

Complexity

```
void AVL::insertR(Node*& t, item x)
{
  if (t==NULL) {t==new Node(x); return; }
  else if (x < t->data)
  {
    insertR(t->lson, x);                  // insert
    if (h(t->lson)==h(t->rson)+2)         // checking
      if (x < t->lson->data) rotateL(t); // case 1
      else dbl_rotateL(t);                // case 2
  }
  else if (x > t->data)
  {
    insertR(t->rson, x);                   // insert
    if (h(t->rson)==h(t->lson)+2)          // checking
      if (x > t->rson->data) rotateR(t);  // case 4
      else dbl_rotateR(t);                 // case 3
  }
  else ;  // duplication
  t->height = max( h(t->lson), h(t->rson) ) + 1;
}
```

- Worst case time complexity:
  - Local work requires constant time *c*
  - At most 1 recursive call with tree height *k-1* where *k* = height of subtree pointed to by *t*
  - So, T(k) = T(k-1) + c
    = T(k-2) + 2c
    …
    = T(0) + kc
    = O(k)

(Another approach)

$$
\begin{aligned}
T(k)&=T(k-1)+c\\
T(k-1)&=T(k-2)+c\\
T(k-2)&=T(k-3)+c\\
&\cdots\\
T(1)&=T(0)+c\\
\hline
T(k)&=T(0)+kc
\end{aligned}
$$

k equations

- Worst case time complexity of insert(x)
  = worst case time complexity of insertR(root, x)
  = O(h)

  where h = height of the whole tree

Theorem

- What is h (as a function of n)?
- Theorem: Every AVL-tree of height *h* has

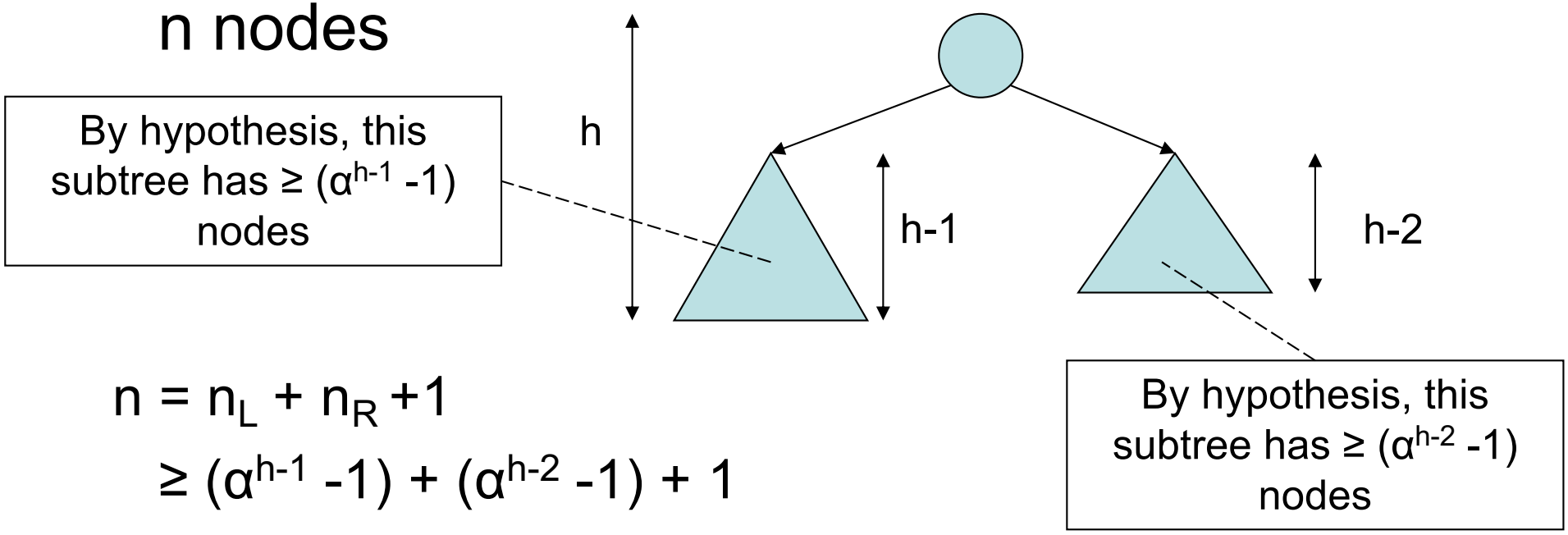
  $\geq \alpha^h - 1$ nodes

  - where $\alpha = (1+\text{sqrt}(5))/2 \approx 1.618$ (Golden ratio)
  - Note: $\alpha^2 = \alpha + 1$
- Proof: (by induction on h)

  Base Case (h=1)
  - An AVL tree of height 1 has 1 node
  - and $n=1 \geq \alpha^h - 1$

Induction Step

- Assume every AVL-tree of height *k* has
  $k \geq \alpha^k - 1$ nodes for all k<h for some h>1
- Consider an arbitrary AVL tree of height h with n nodes



$n = n_L + n_R + 1$

$\geq (\alpha^{h-1} - 1) + (\alpha^{h-2} - 1) + 1$

$= \alpha^{h-2} + \alpha^{h-1} - 1$

$= \alpha^{h-2} (\alpha + 1) - 1$

$= \alpha^{h-2} (\alpha^2) - 1 = \alpha^h - 1$ (proved)

So, $\alpha^h \leq n+1$, i.e., $h \leq \log_\alpha(n+1) = O(\log n)$

Deletion

- Do normal deletion for binary search tree
- Rebalancing (need to check all the ancestors of the affected node)
- Rather complicated

AVL drawback

- extra storage/complexity for height fields
  - ❑ require additional storage to maintain the balance factor
- ugly delete code

AVL Trees Exercise 1

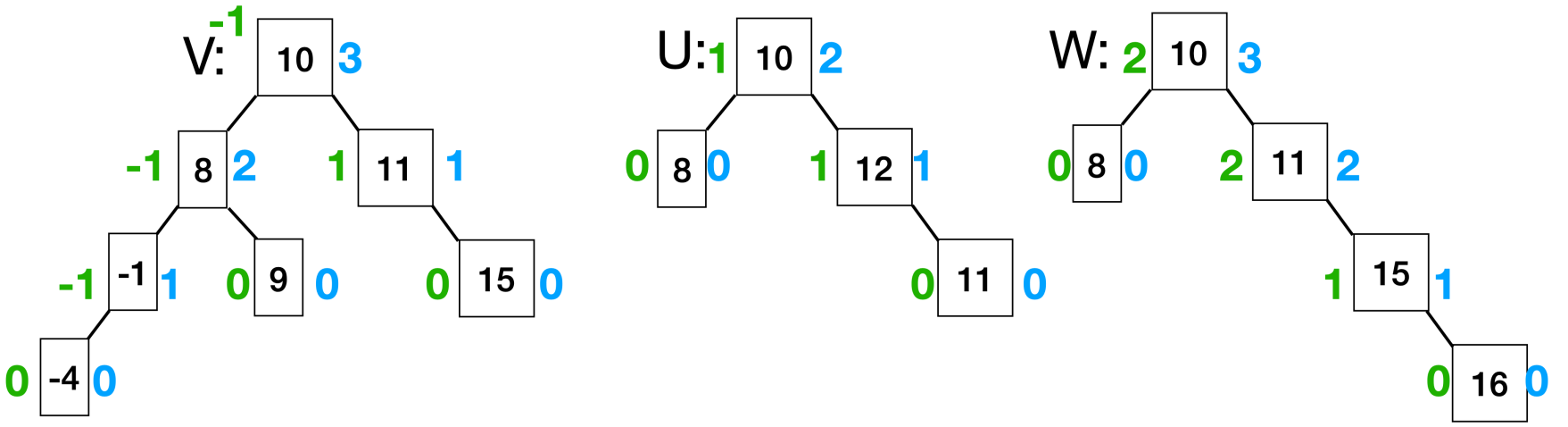
Which of these is/are not AVL trees?

a) V

b) W

c) V and W

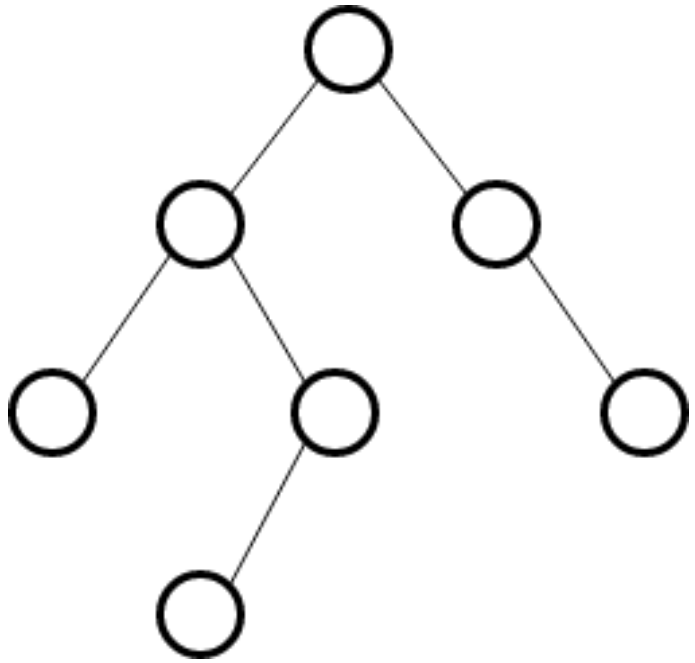
d) U and W



AVL Trees Exercise 2

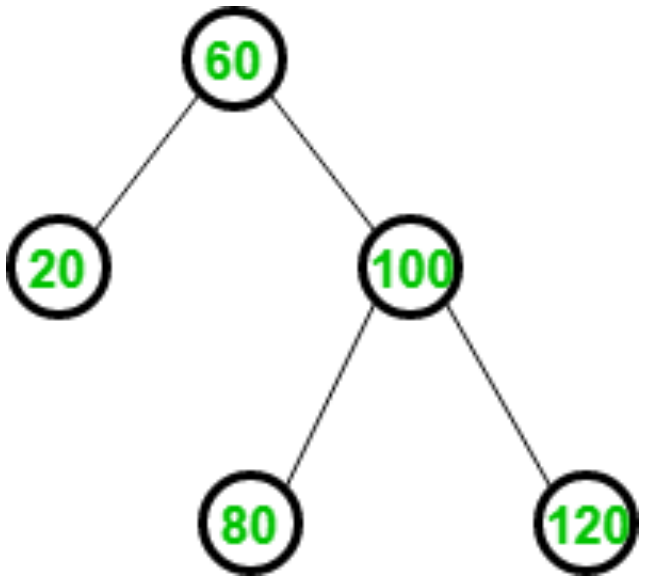
What is the maximum height of any AVL-tree with 7 nodes?
Assume that the height of a tree with a single node is 0.
And giving the corresponding example with the max height.

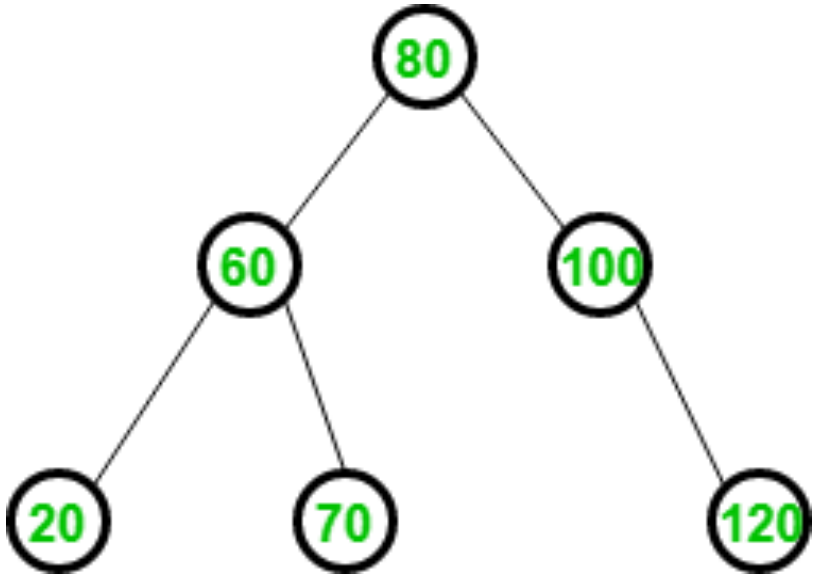
a) 2
b) 3
c) 4
d) 5



AVL Trees Exercise 3

What is updated AVL tree after insertion of 70?





AVL Trees Exercise 4

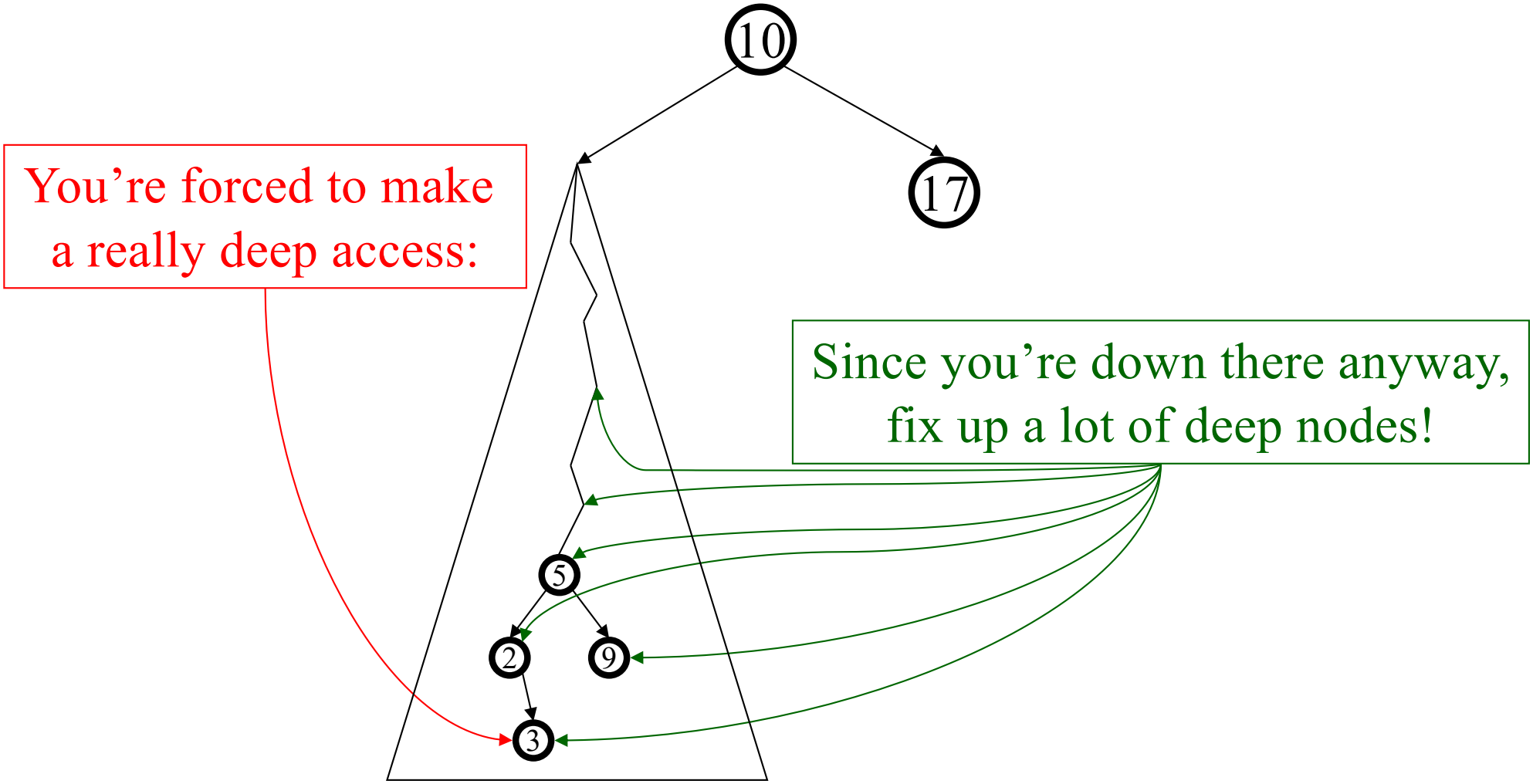
Show the AVL tree that results after each of the integer keys **9, 27, 50, 15, 2, 21, and 36** are inserted, in that order, into an initially empty AVL tree.

Splay Trees

- blind adjusting version of AVL trees
- doesn't care about differing heights of subtrees
- amortized time for all operations is O(log n)
- worst case time is O(n)
- insert/find always rotates node to the root!

* Credits: https://courses.cs.washington.edu/courses/cse326/01au/lectures/SplayTrees.ppt

Splay Tree Idea

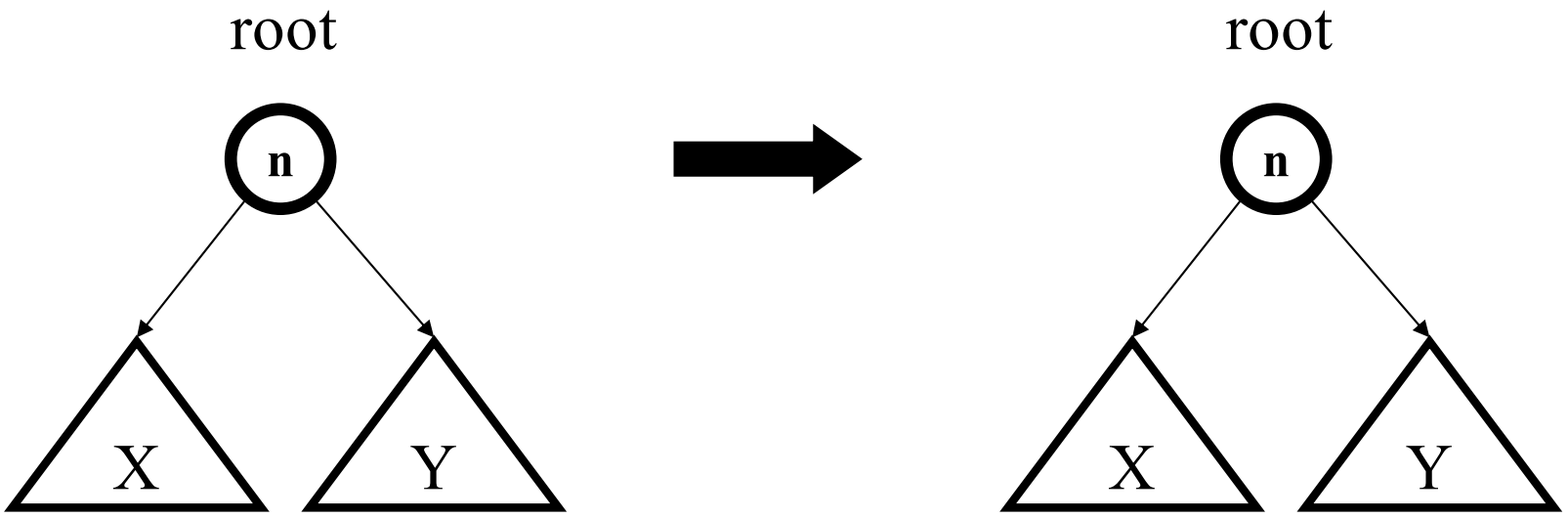


Splaying Cases

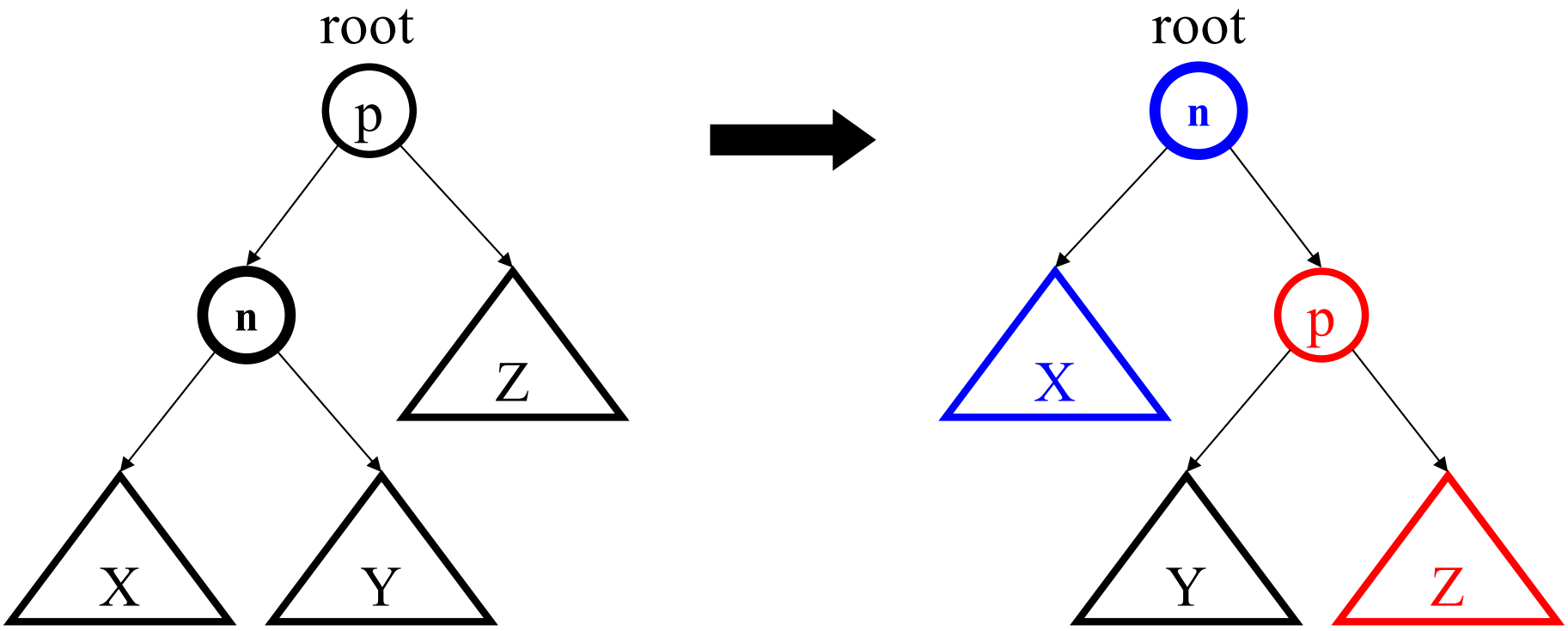
Node being accessed (n) is:

- Root
- Child of root
- Has both parent (p) and grandparent (g)
  - Zig-zig pattern: g → p → n is left-left or right-right
  - Zig-zag pattern: g → p → n is left-right or right-left

Access root:
Do nothing (that was easy!)



Access child of root:
Zig (AVL single rotation)



Learning Objectives 1

1. Explain the concepts of Game Trees and Heaps
2. Able to decide the best move on a game tree
3. Able to insert into and delete from a Min-heap or Max-heap manually
4. Able to do α-β pruning manually

D:1; C:1,2; B:1,2,3; A:1,2,3,4

Learning Objectives 2

1. Know the definition of AVL trees
2. Able to do rotations on AVL trees
3. Able to compare AVL trees and splay trees
4. Able to prove the complexity of operations in Balanced Binary Search Trees

D:1; C:1,2; B:1,2,3; A:1,2,3,4