Data Structures

Lec-10 Disjoint Set & Suffix Array

Objective of Balanced BST

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

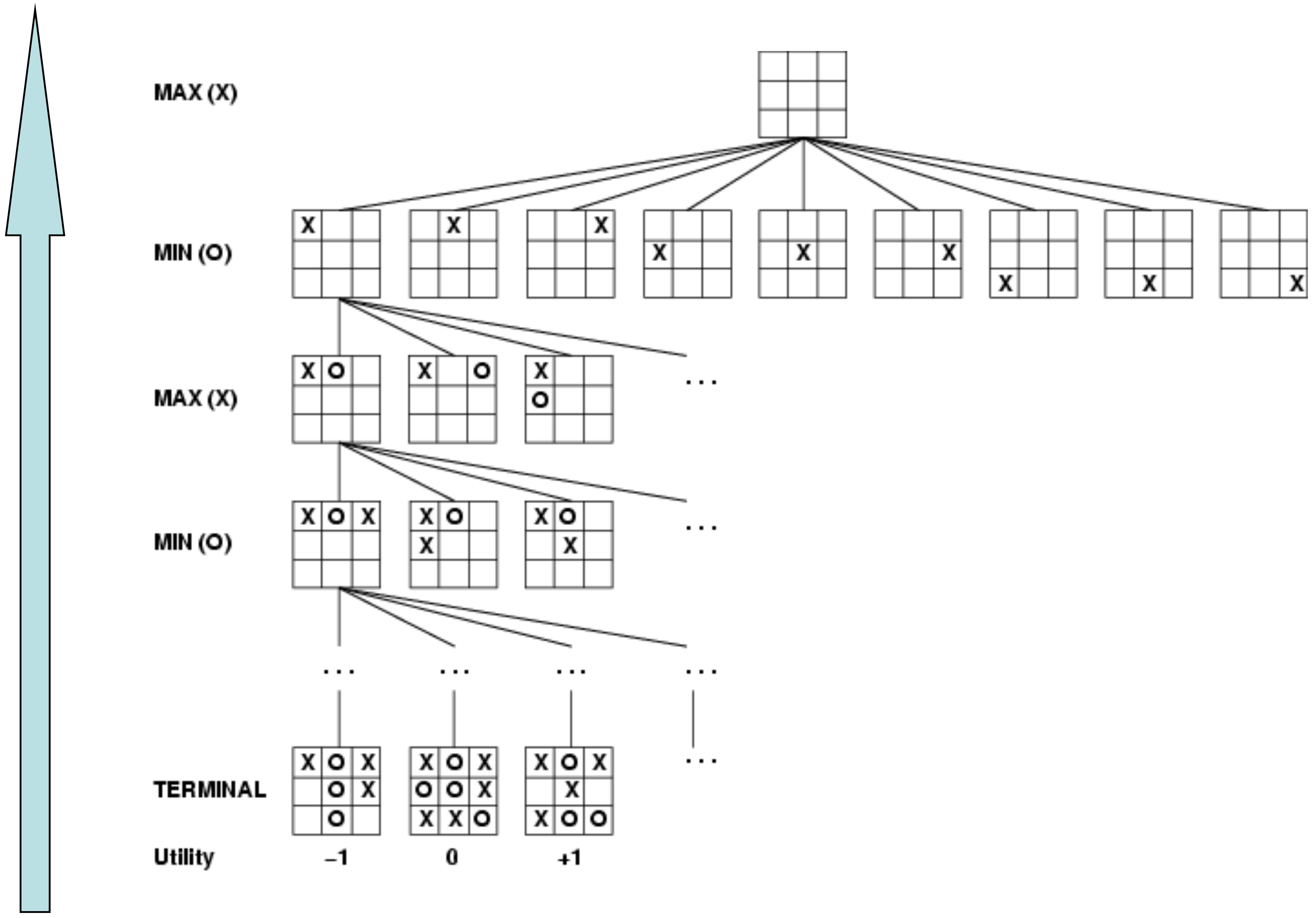
Review: Game

- 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

Review: Game Tree

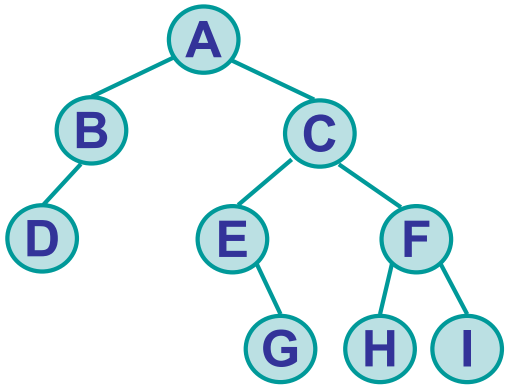
- Properties of the game tree:
  - Root of the tree is the initial game status
  - Every node in the tree is a possible game status
  - Every edge in the tree is a possible move by the players
  - Any path from root to leaf represents a possible game
- How to make decisions using the game tree?
  - Each 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

Review: Minimax (How to search)



Review: DFS (Depth First Search)

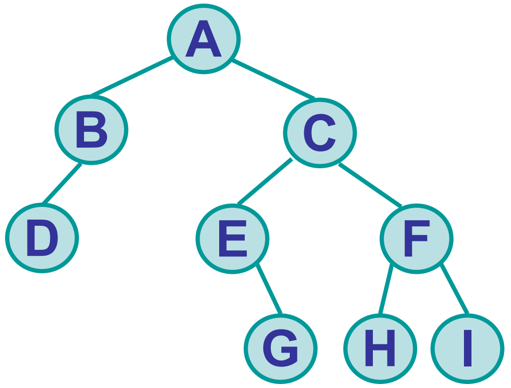
- DFS: Go as deep as you can
- In tree, DFS=
  - Preorder Traversal
  - Solve maze by always turning right
- Example (start from A):



DFS order: ABDCEGFHI

Review: BFS (Breadth First Search)

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

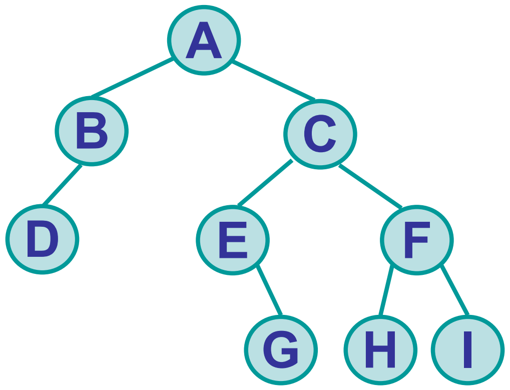


BFS order: ABCDEFGHI

- How to output nodes by increasing depth?
  - Queue-based approach

Review: BFS (Breadth First Search)

- BFS traversal using queue (FIFO)



```
void BFS_queue(TreeNode root)
{
        if (root == NULL)
            return;
        Queue<TreeNode> treeQueue;
         treeQueue.enqueue(root);
         while (treeQueue.empty() == false)
         {
            TreeNode currNode = treeQueue.dequeue();
            process(currNode->data);
            if (currNode->left != NULL)
                treeQueue.enqueue(currNode->left);
            if (currNode->right != NULL)
                treeQueue.enqueue(currNode->right);
          }
}
```

Review: The Minimax Algorithm

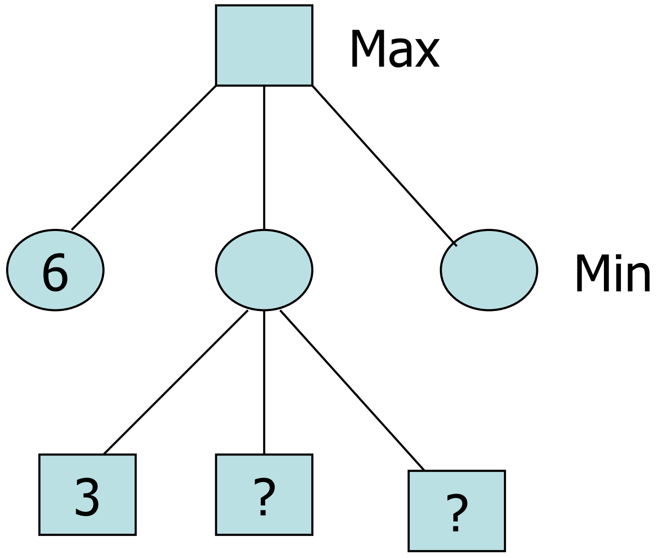
- Designed to find the optimal strategy or just best first move for MAX

- **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

Review: α-β pruning



Do we need to know '?'

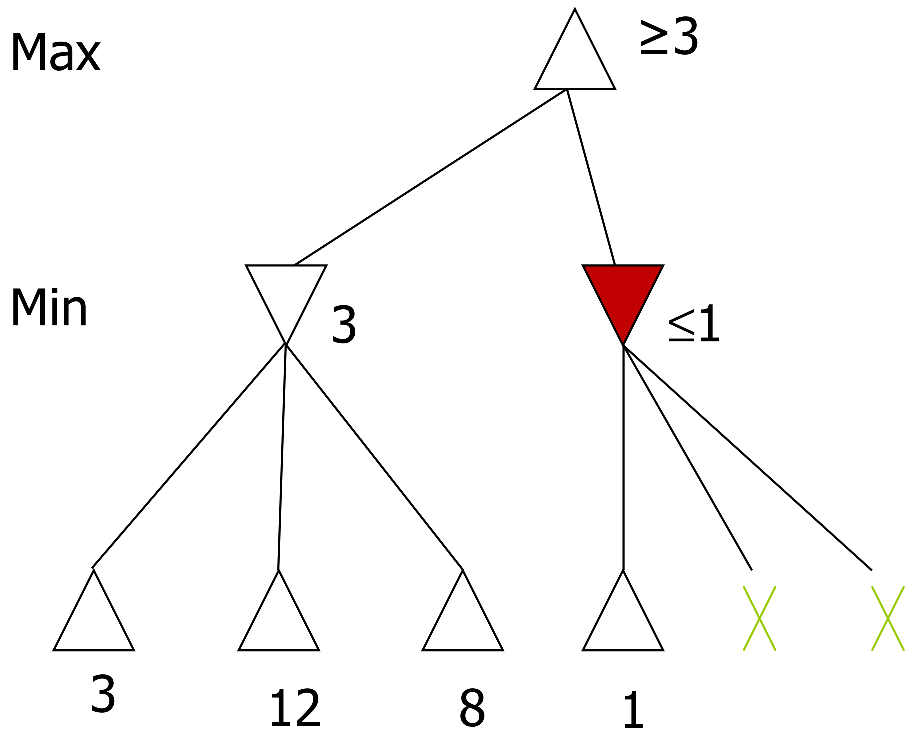
Review: Alpha-Beta Pruning
- Exploiting the Fact of an Adversary

- **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.

Review: 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.

Review: α-β pruning example



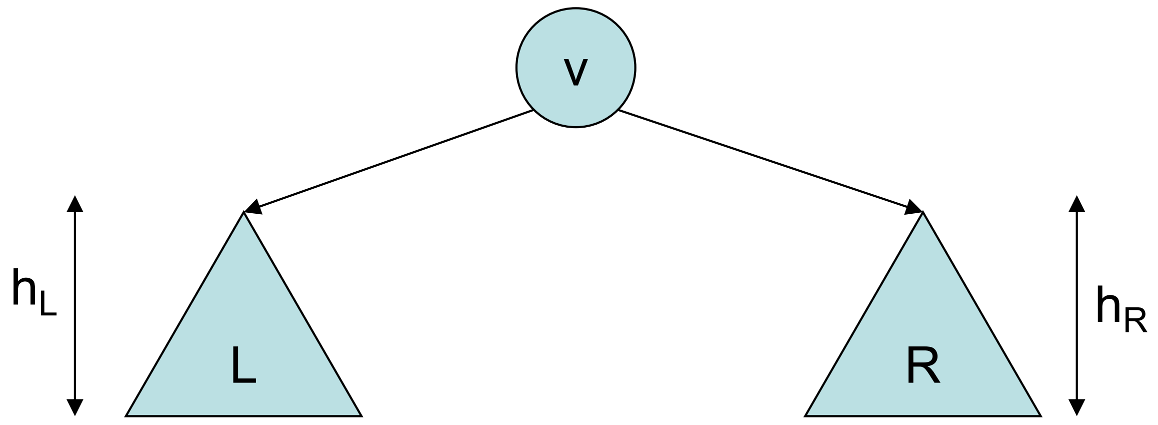
Review: AVL Trees

- 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*!

Review: AVL Trees

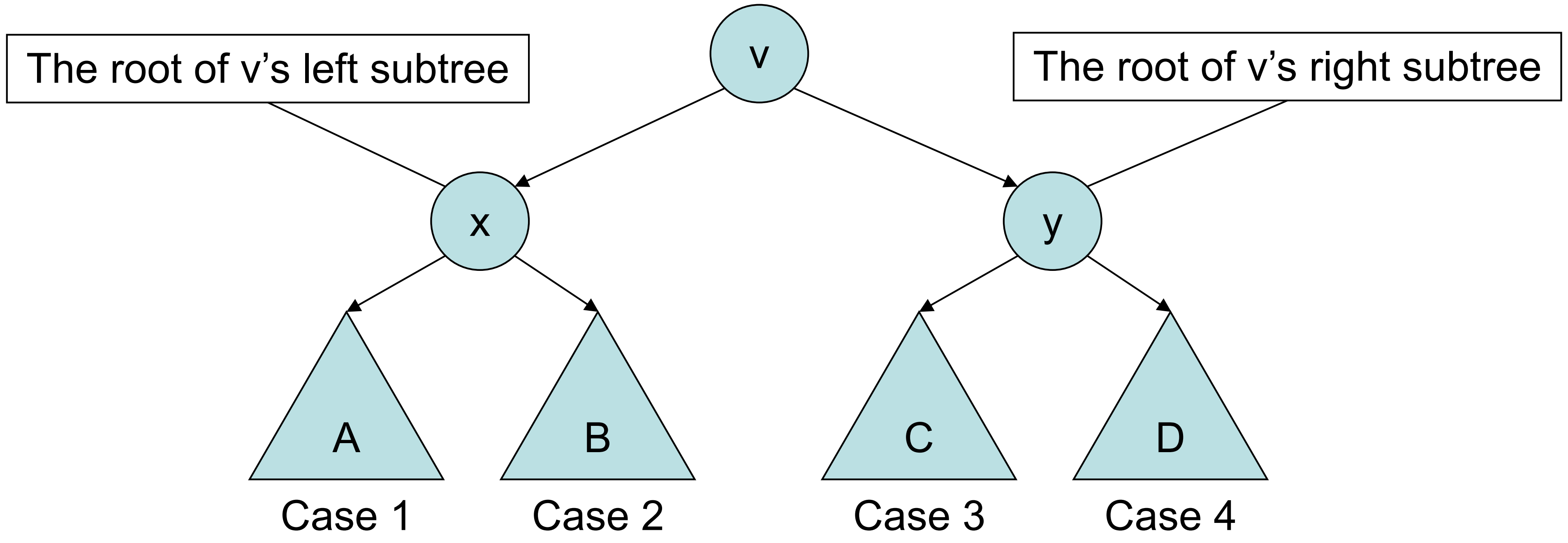
- Height of a tree = number of edges on a longest root-to-leaf path
- Note: height of the tree below is

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



Review: Insertion

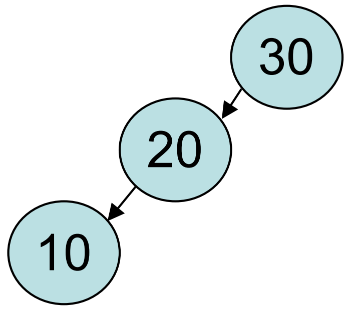
- Consider insert(u): only nodes along the path from root to the point of insertion may be unbalanced
- Suppose node *v* unbalanced, 4 cases:
  - Cases 1 & 4 are mirror image symmetries with respect to v
  - Cases 2 & 3 are mirror image symmetries with respect to v



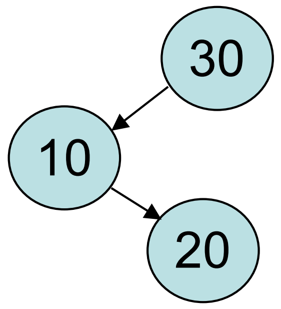
Review: Insertion Example

Insert Inputs: {10, 20, 30}

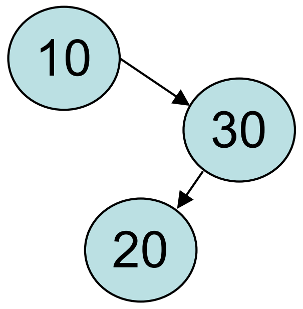
Insert Inputs with order: 30, 20, 10



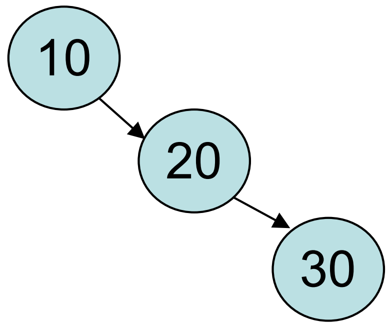
Insert Inputs with order: 30, 10, 20



Insert Inputs with order: 10, 30, 20

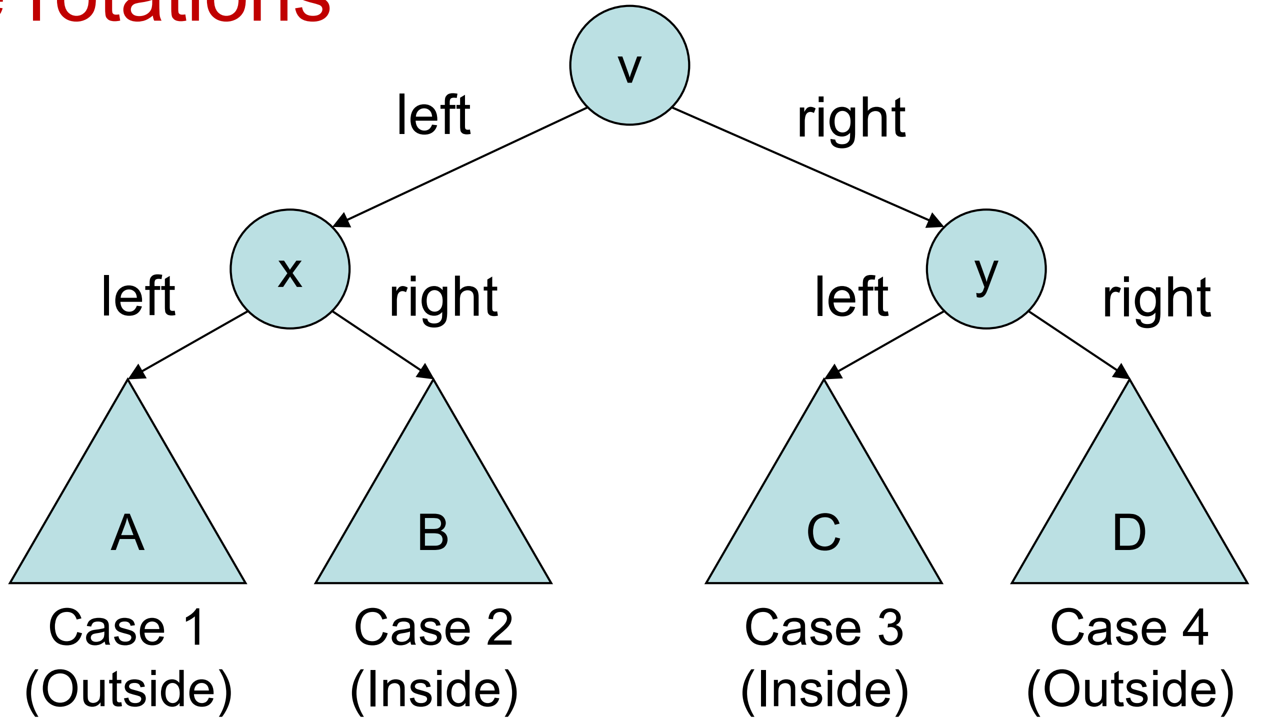


Insert Inputs with order: 10, 20, 30



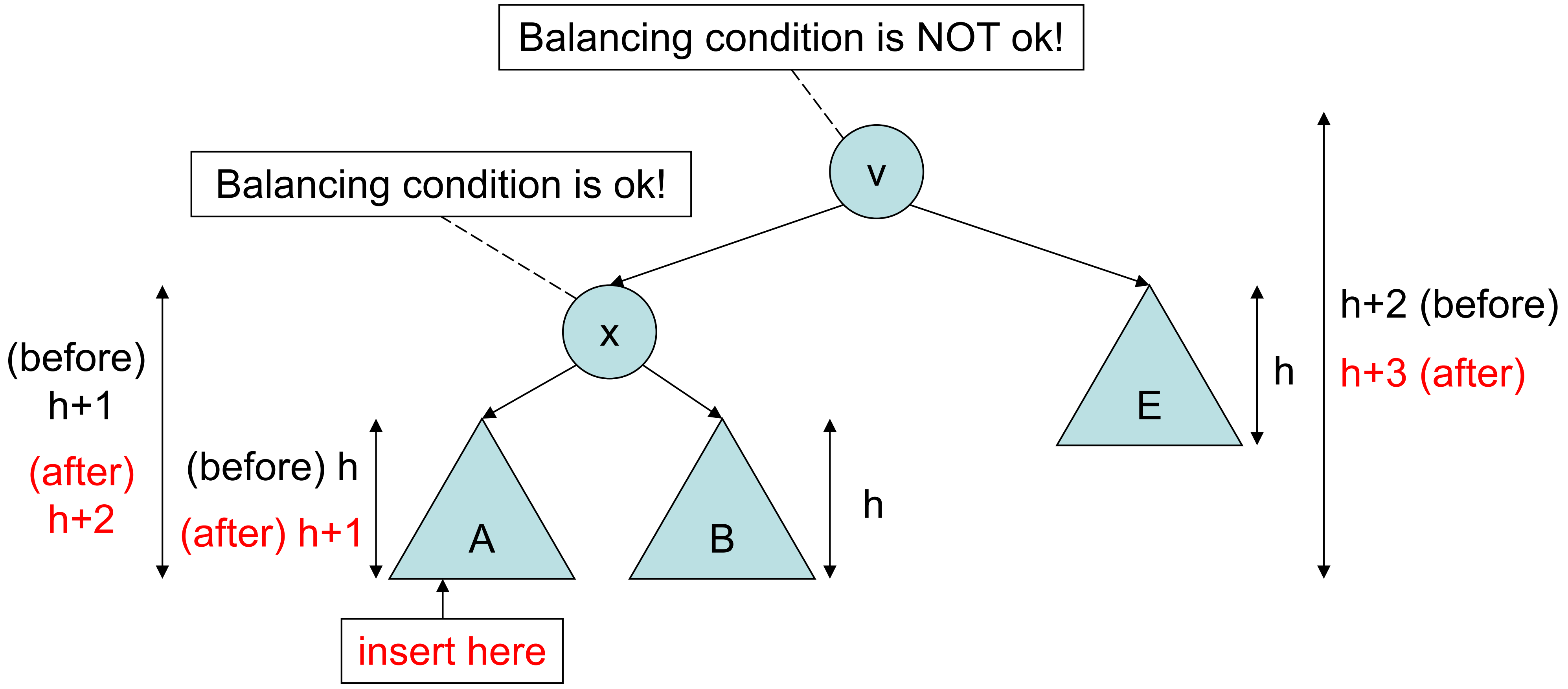
Review: Insertion

- 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



Review: Insertion – Case 1

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



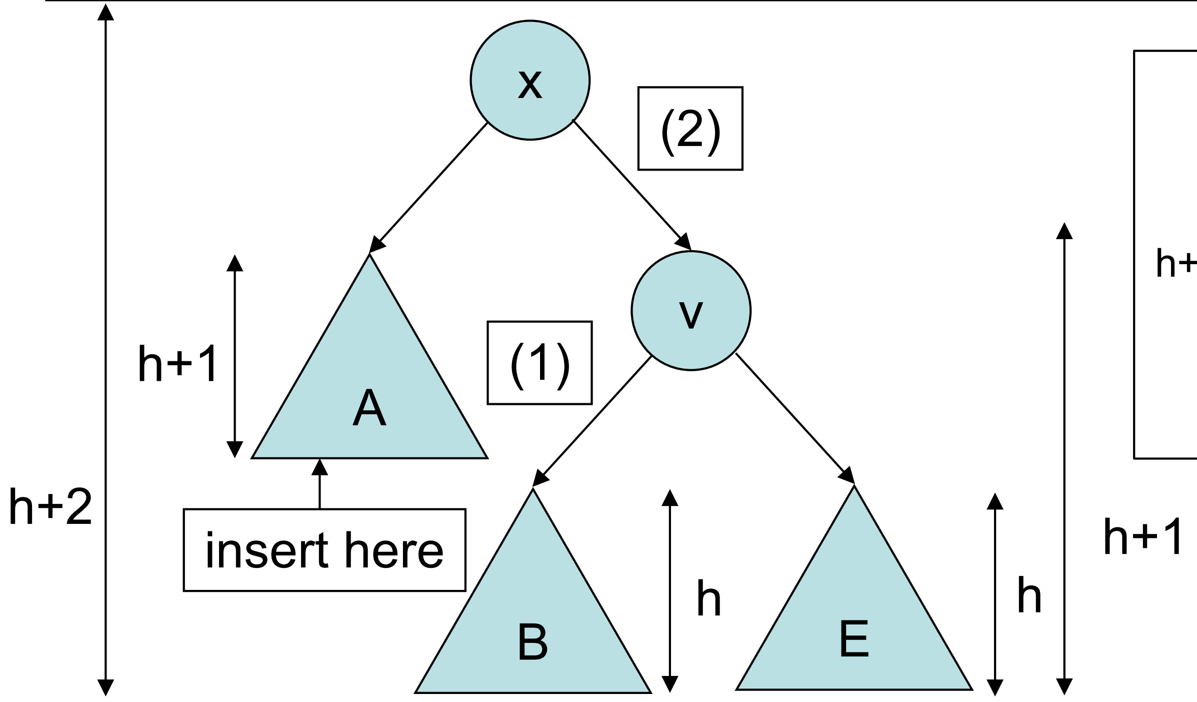
Review: Insertion – Case 1

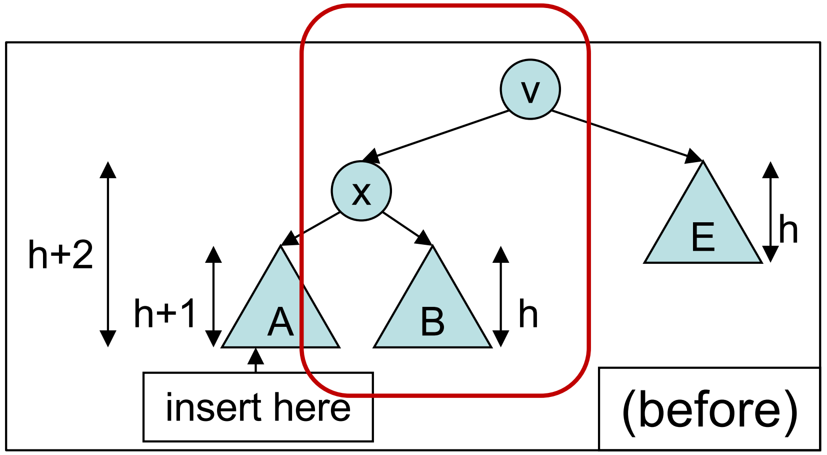
```
void rotateL(Node*& v) // v & v->lson rotate
{
  Node* x=v->lson;
  v->lson=x->rson;
  x->rson=v;
  v->height=max( h(v->lson), h(v->rson) )+1;
  x->height=max( h(x->lson), v->height )+1;
  v=x;
}
```

(1) v becomes x's right child

(2) x's original right become v's left

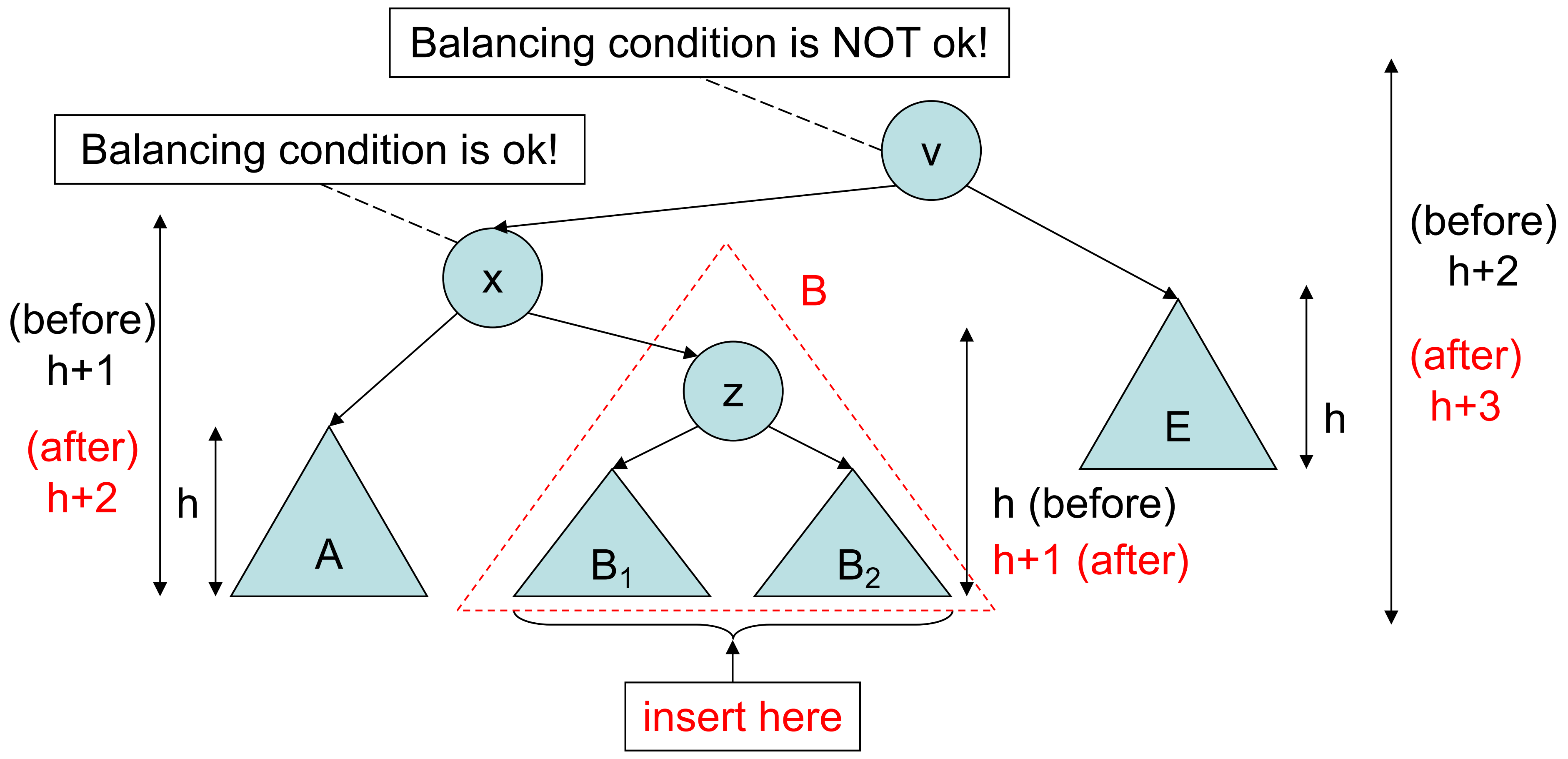
Make x as the root





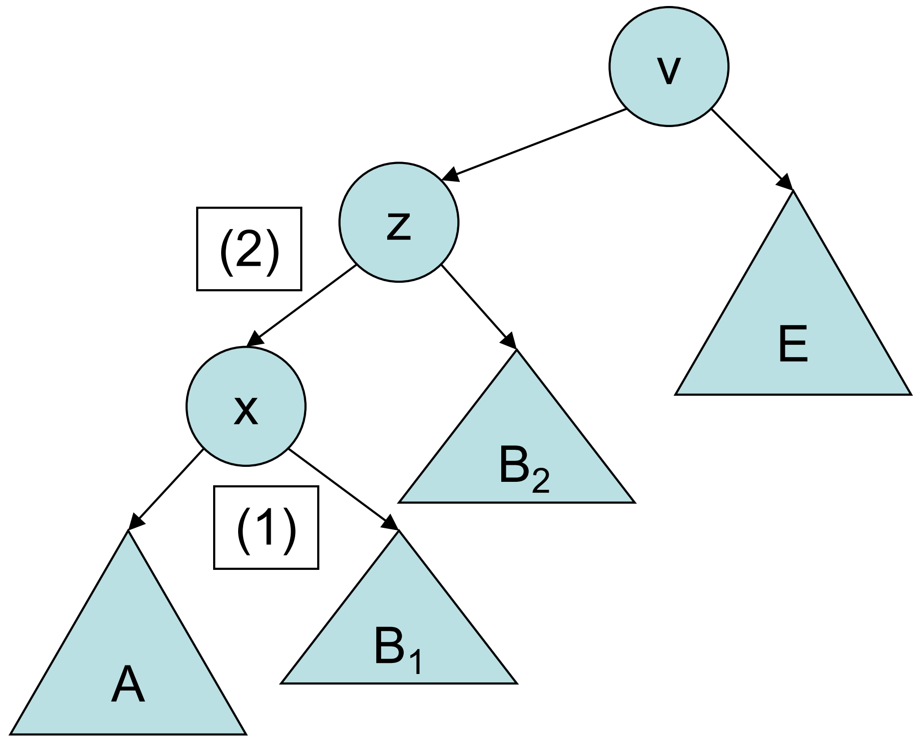
Review: Insertion – Case 2

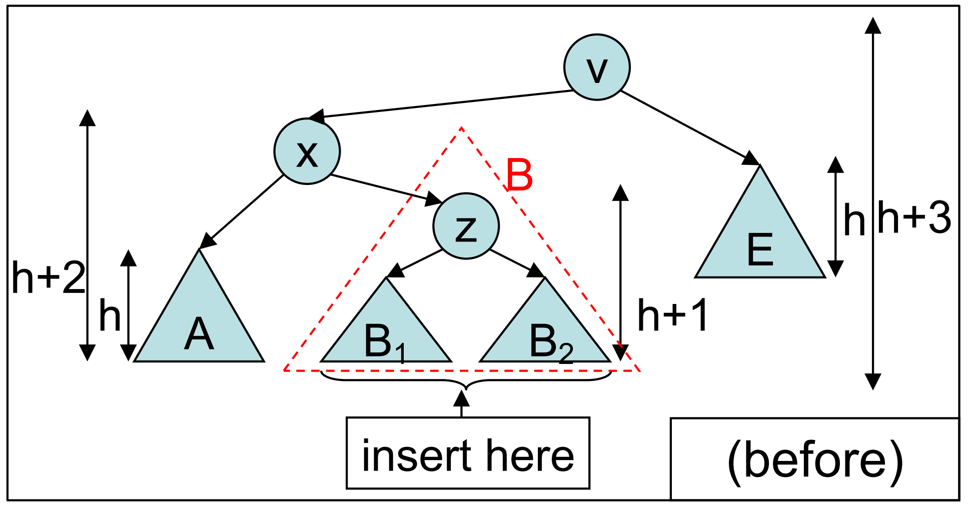
- Before insertion, height of subtree B = h and height of subtree E = h
- After insertion, height of subtree B = h+1, either B1 or B2 has height h, the other h-1



Review: Insertion – Case 2

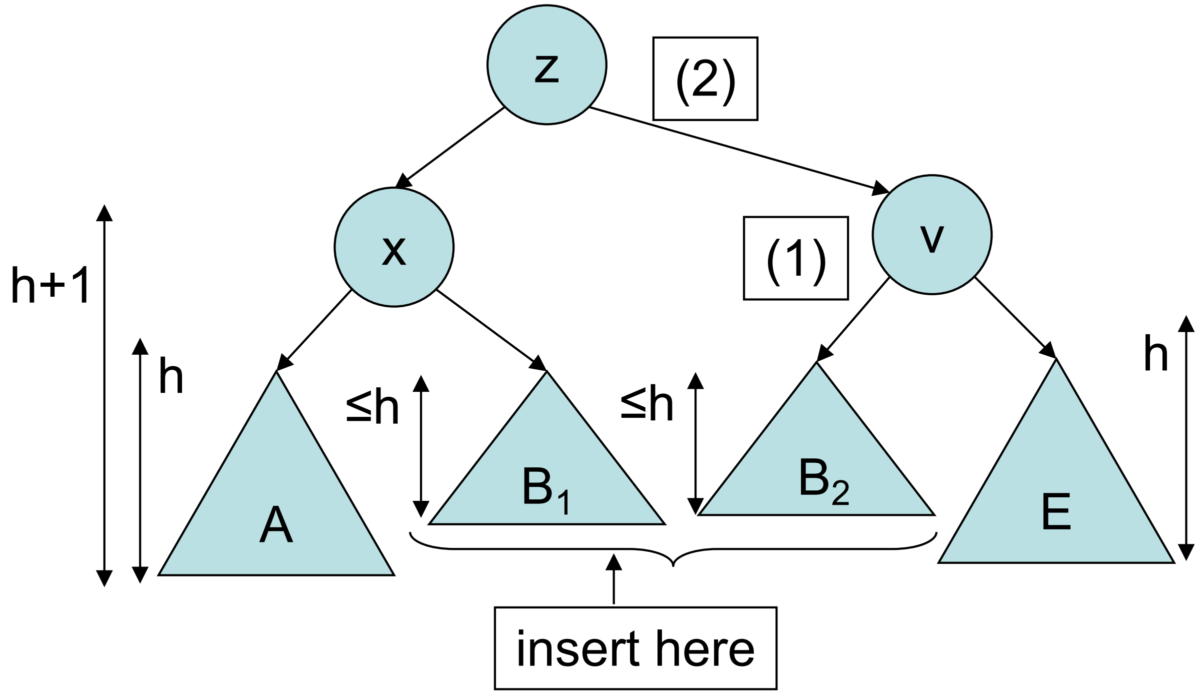
```
void rotateR(Node*& x)  //s & s->rson rotate
{
  Node* z=x->rson;
  x->rson=z->lson;   <- (1)
  z->lson=x;   <- (2)
  x->height=max( h(x->lson), h(x->rson) )+1;
  z->height=max( h(z->rson), x->height )+1;
  x=z;   <- Make z as the root
}
```

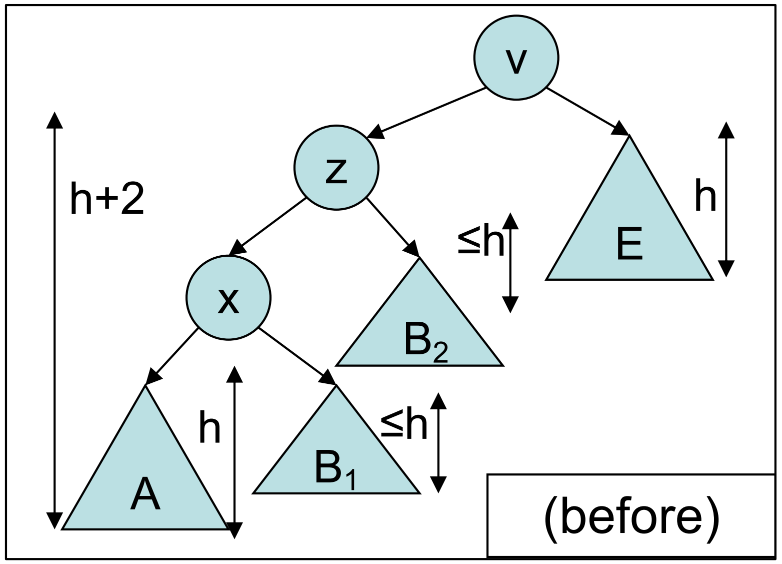




Review: Insertion – Case 2

```
void rotateL(Node*& v) // v & v->lson rotate
{
  Node* z=v->lson;
  v->lson=z->rson;      ← (1)
  z->rson=v;            ← (2)
  v->height=max( h(v->lson), h(v->rson) )+1;
  z->height=max( h(z->lson), v->height )+1;
  v=z;                  ← Make z as the root
}
```

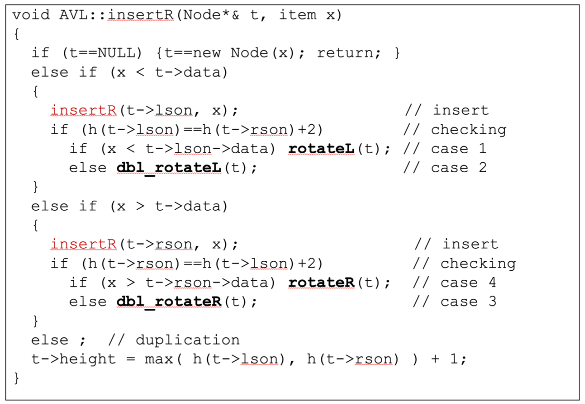




Review: Overall Scheme for insert(*x*)

- 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

Review: 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)

$$\left.\begin{array}{r} T(k)=T(k-1)+c \\ T(k-1)=T(k-2)+c \\ T(k-2)=T(k-3)+c \\ \dots \\ T(1)=T(0)+c \\ \hline T(k)=T(0)+kc \end{array}\right\} k \text{ equations}$$

- Worst case time complexity of insert(x)

  = worst case time complexity of insertR(root, x)

  = O(h)

  where h = height of the whole tree

Review: Theorem

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

  $\geq \alpha^h - 1$ nodes (i.e., $n \geq \alpha^h - 1$)
  - where $\alpha = (1+\text{sqrt}(5))/2 \approx 1.618$ (Golden ratio)
  - Note: $\alpha^2 = \alpha+1$

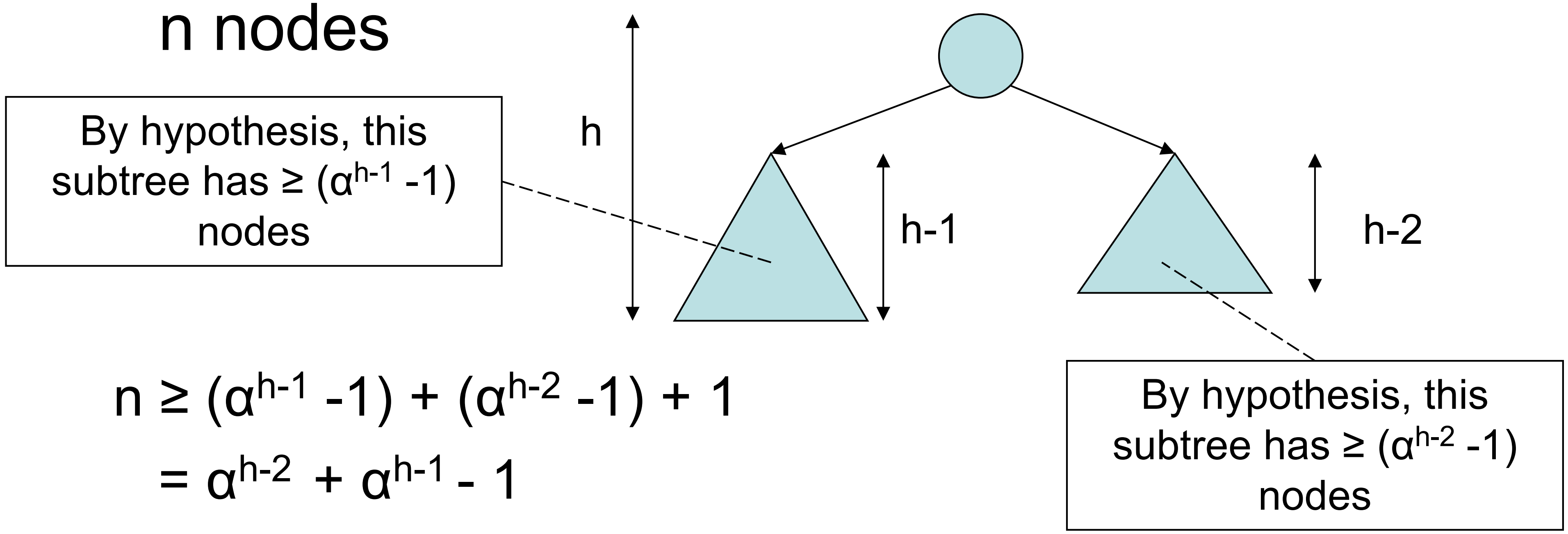
- Proof: (by induction on h)

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

Review: Induction Step

– Assume every AVL-tree of height *(h-1)* has $\geq \alpha^{h-1} - 1$ nodes

– Consider an arbitrary AVL tree of height h with n nodes


By hypothesis, this subtree has $\geq (\alpha^{h-1} - 1)$ nodes
h
h-1
h-2
By hypothesis, this subtree has $\geq (\alpha^{h-2} - 1)$ nodes

$$
\begin{aligned}
n &\geq (\alpha^{h-1} - 1) + (\alpha^{h-2} - 1) + 1 \\
&= \alpha^{h-2} + \alpha^{h-1} - 1 \\
&= \alpha^{h-2} (\alpha + 1) - 1 \\
&= \alpha^{h-2} (\alpha^2) - 1 = \alpha^h - 1 \text{ (proved)}
\end{aligned}
$$

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

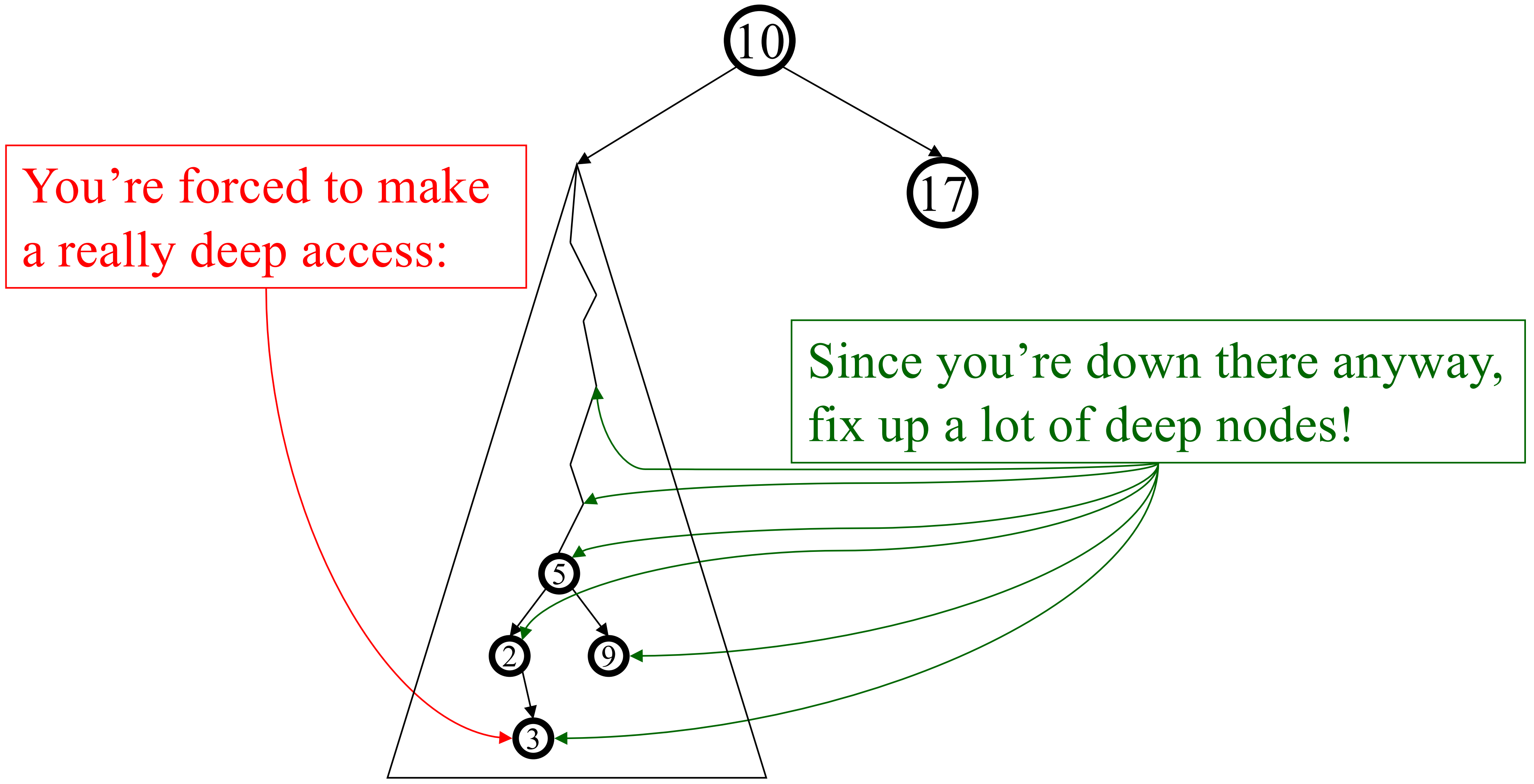
Incomplete!

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

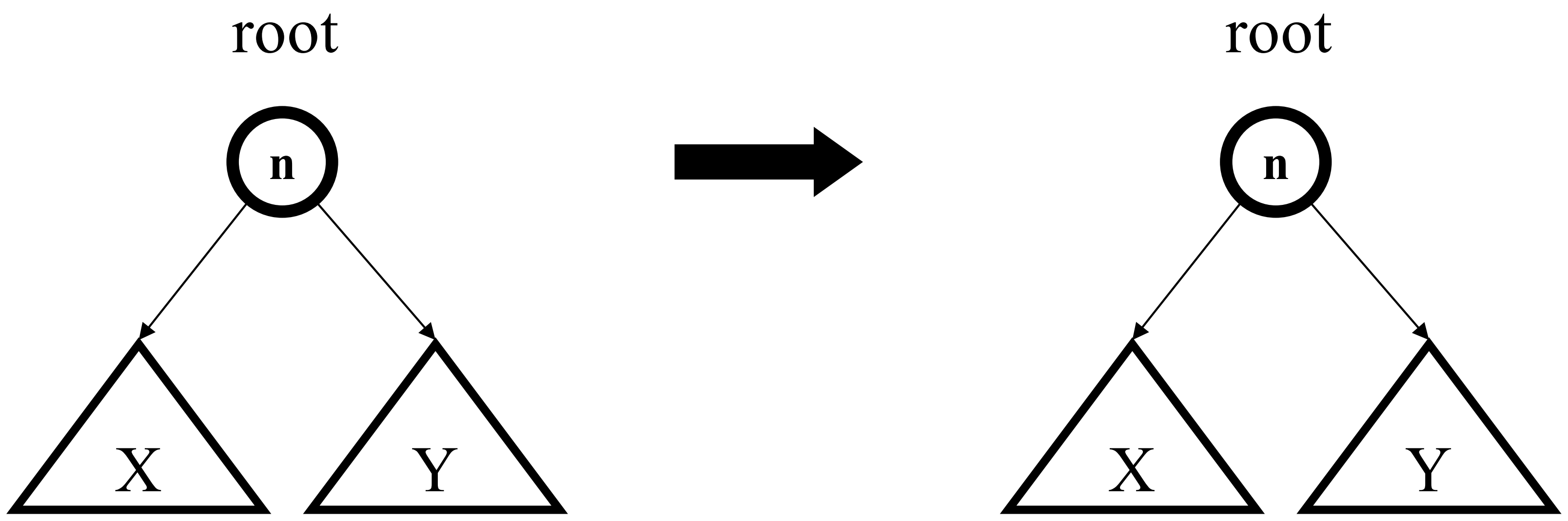

10
17
You're forced to make a really deep access:
Since you're down there anyway, fix up a lot of deep nodes!
5
2
9
3

Splaying Cases

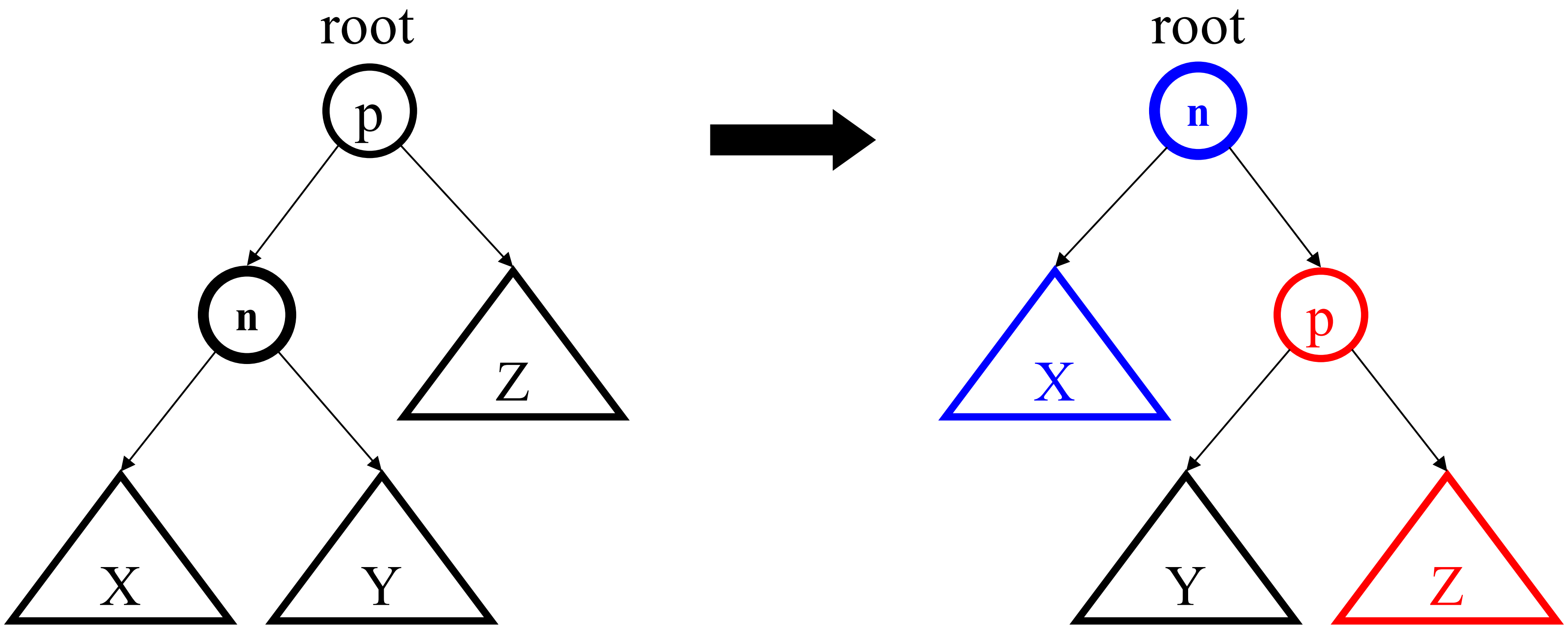
Node being accessed (n) is:

- Root
- Child of root
- Has both parent (p) and grandparent (g)
  - Zig-z**i**g pattern: g → p → n is left-left or right-right
  - Zig-z**a**g pattern: g → p → n is left-right or right-left

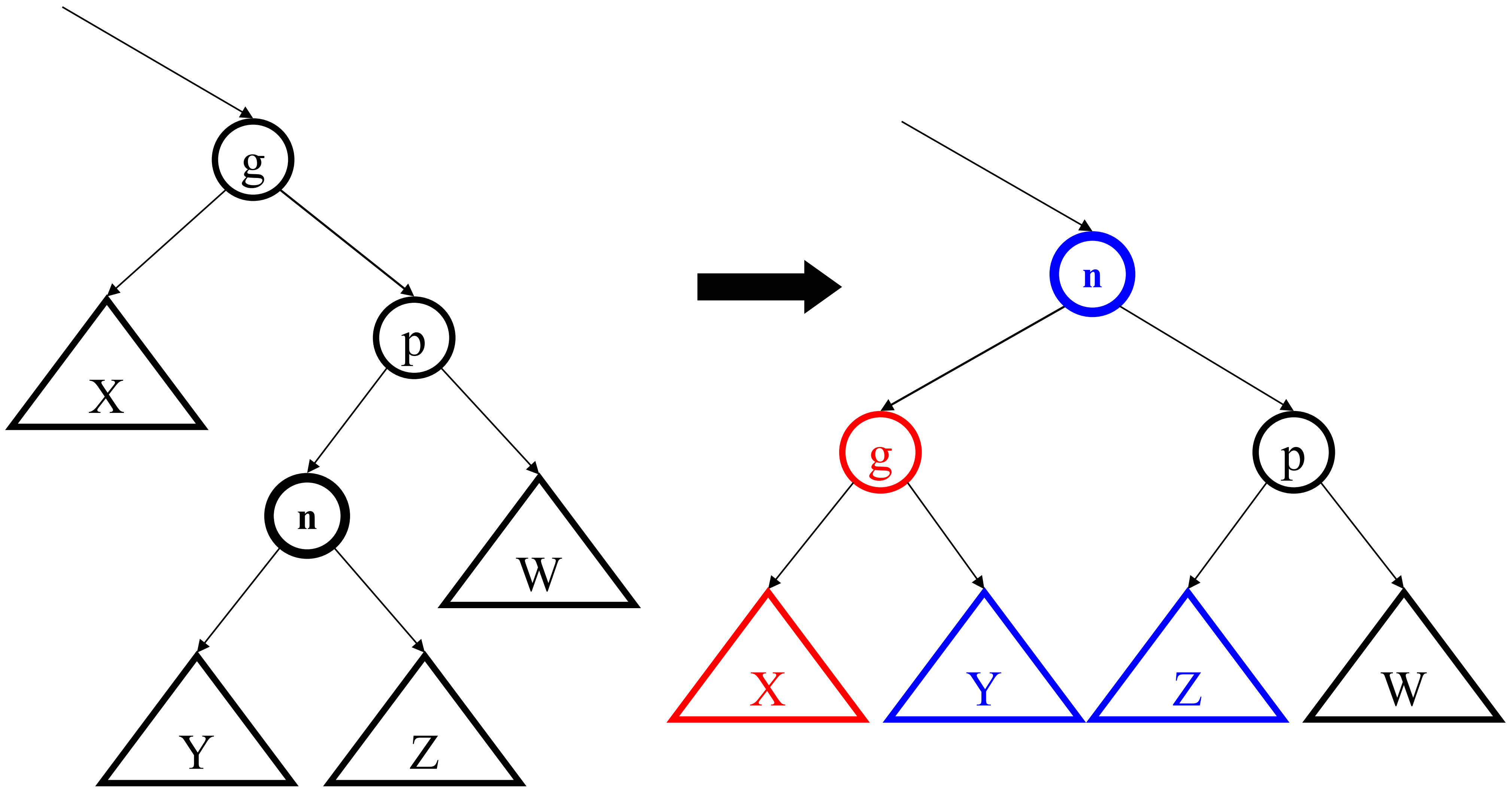
Access root:
Do nothing (that was easy!)



Access child of root:
Zig (AVL single rotation)

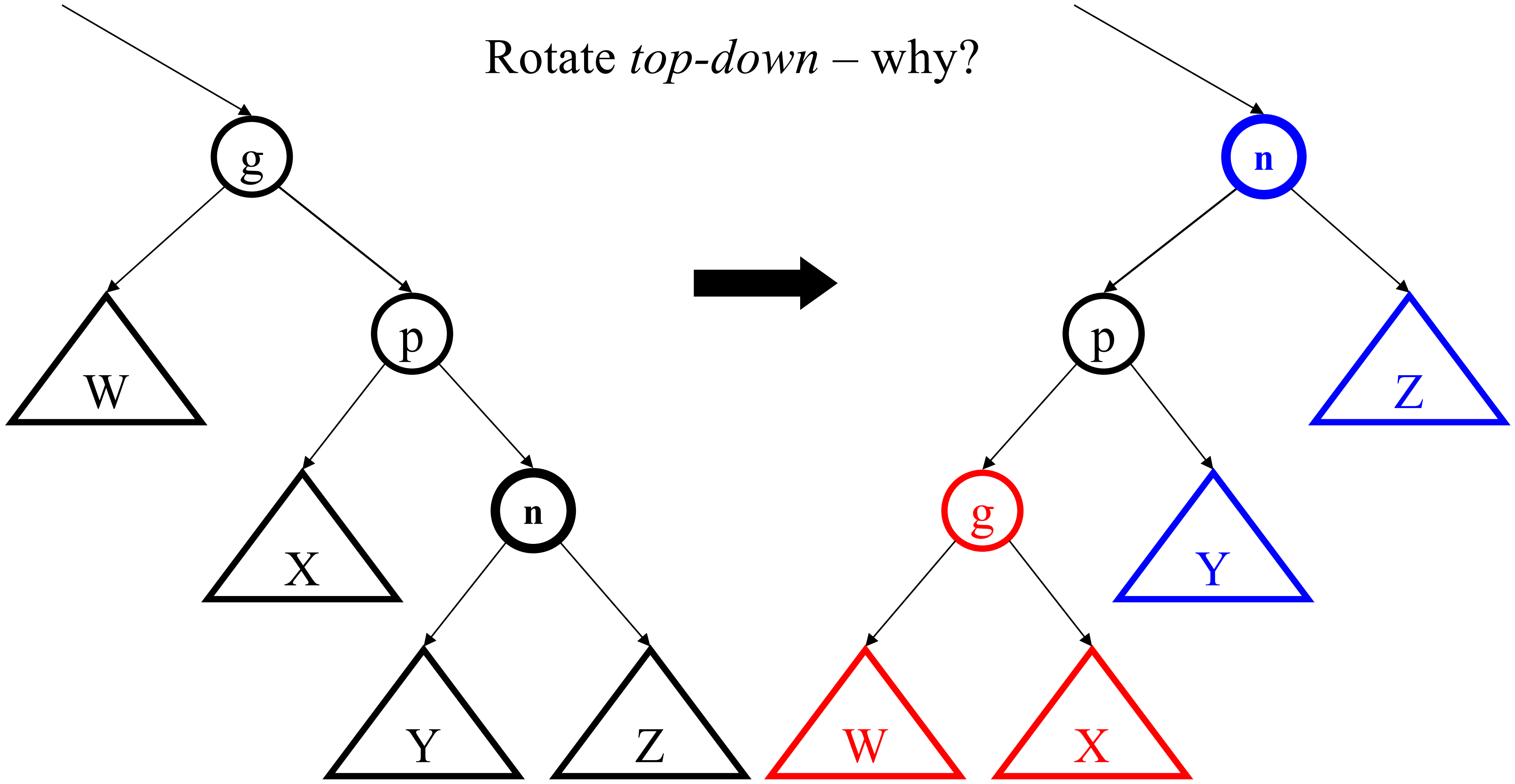


Access (LR, RL) grandchild: Zig-Zag (AVL double rotation)



Access (LL, RR) grandchild: Zig-Zig

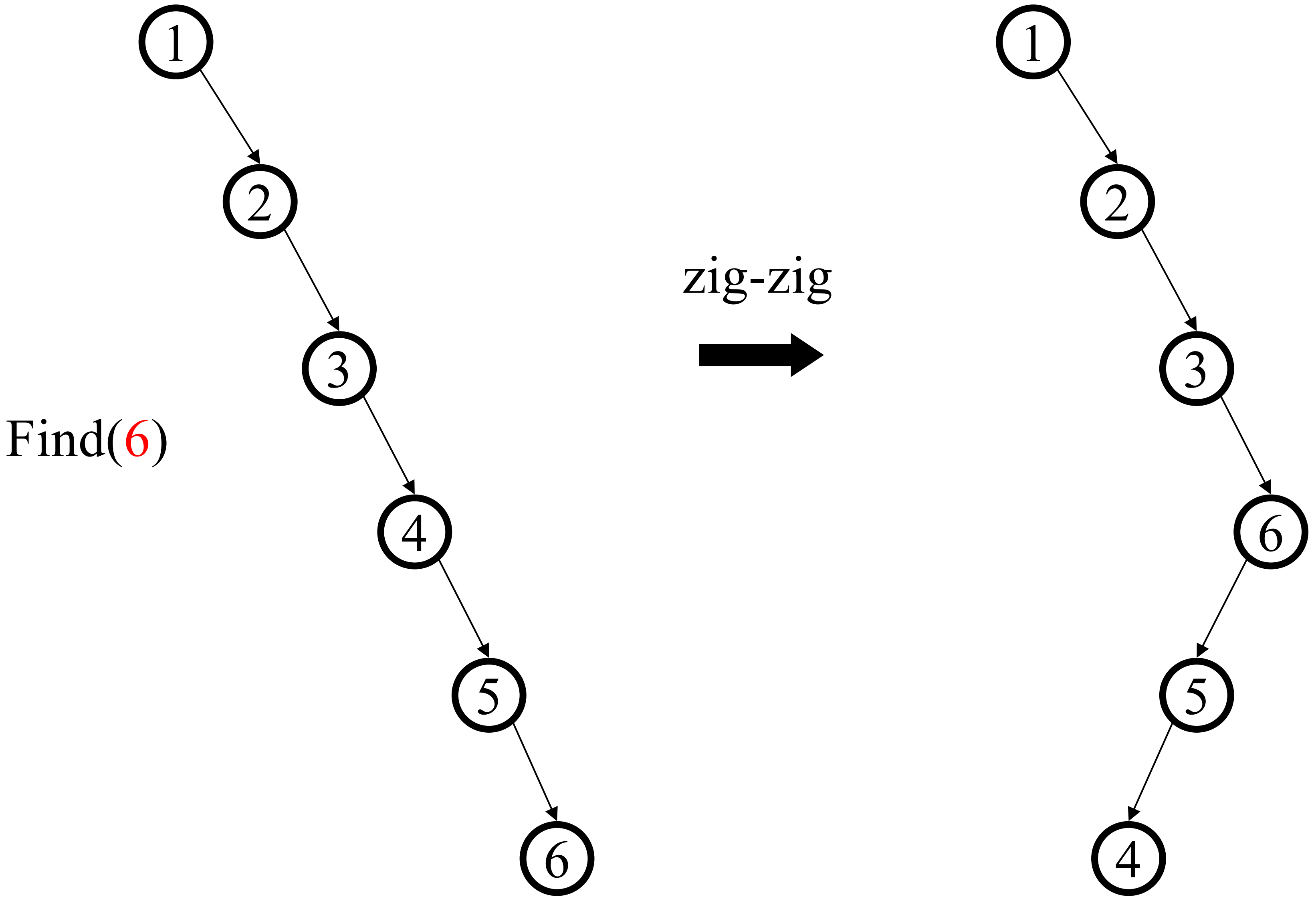
Rotate *top-down* – why?



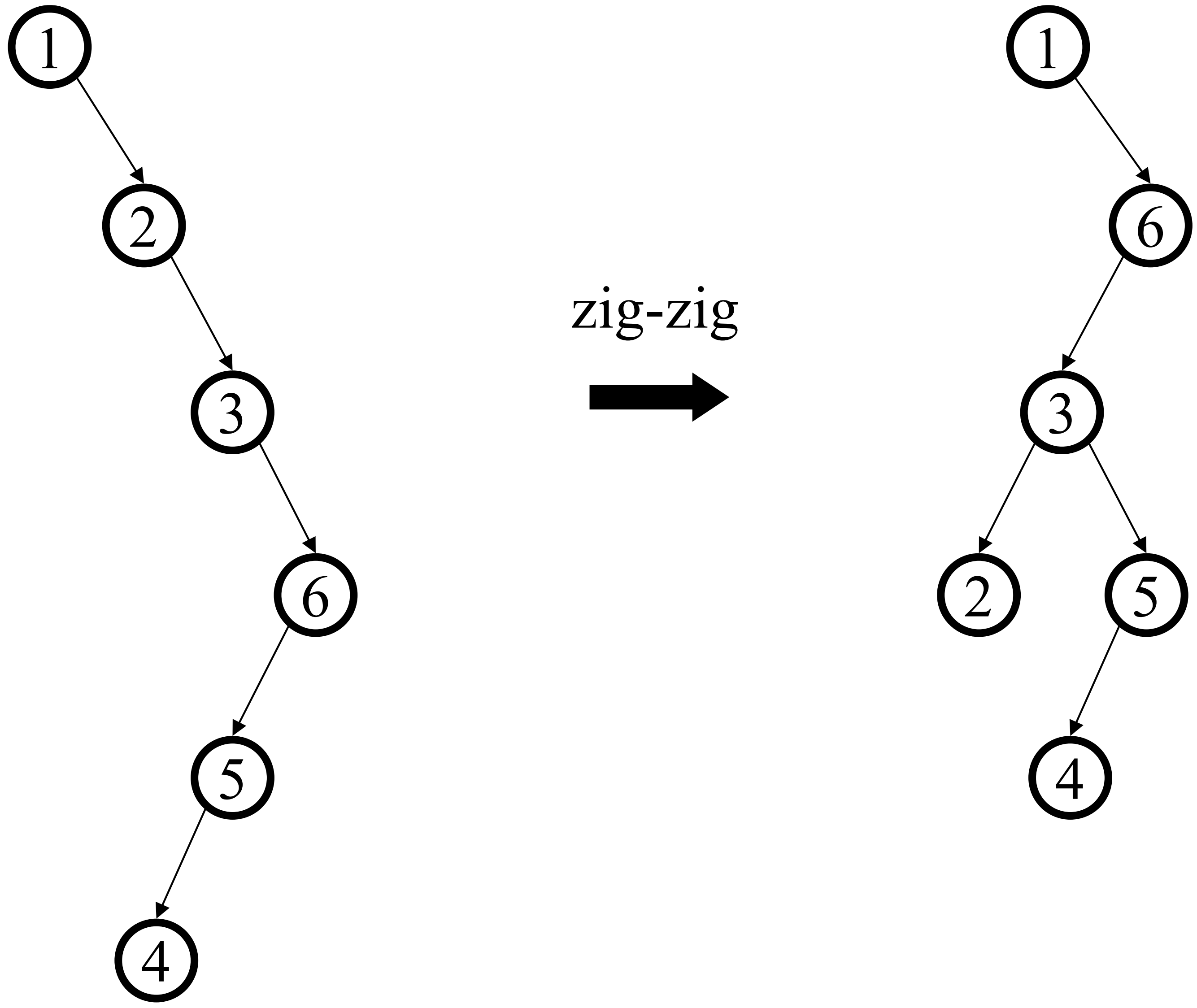
Splay Operations: Find

- Find the node in normal BST manner
- Splay the node to the root

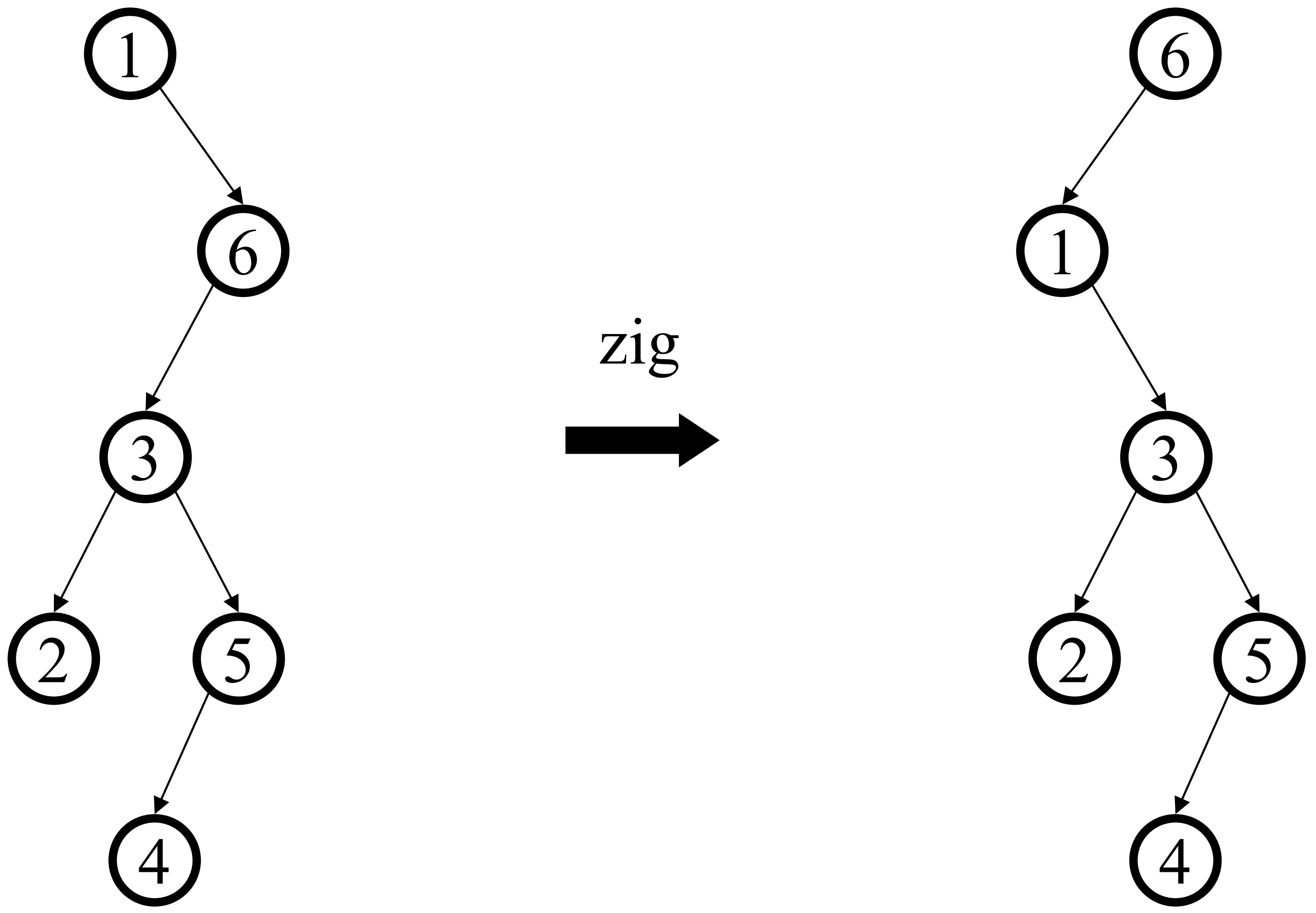
Splaying Example: Find(6)



… still splaying …

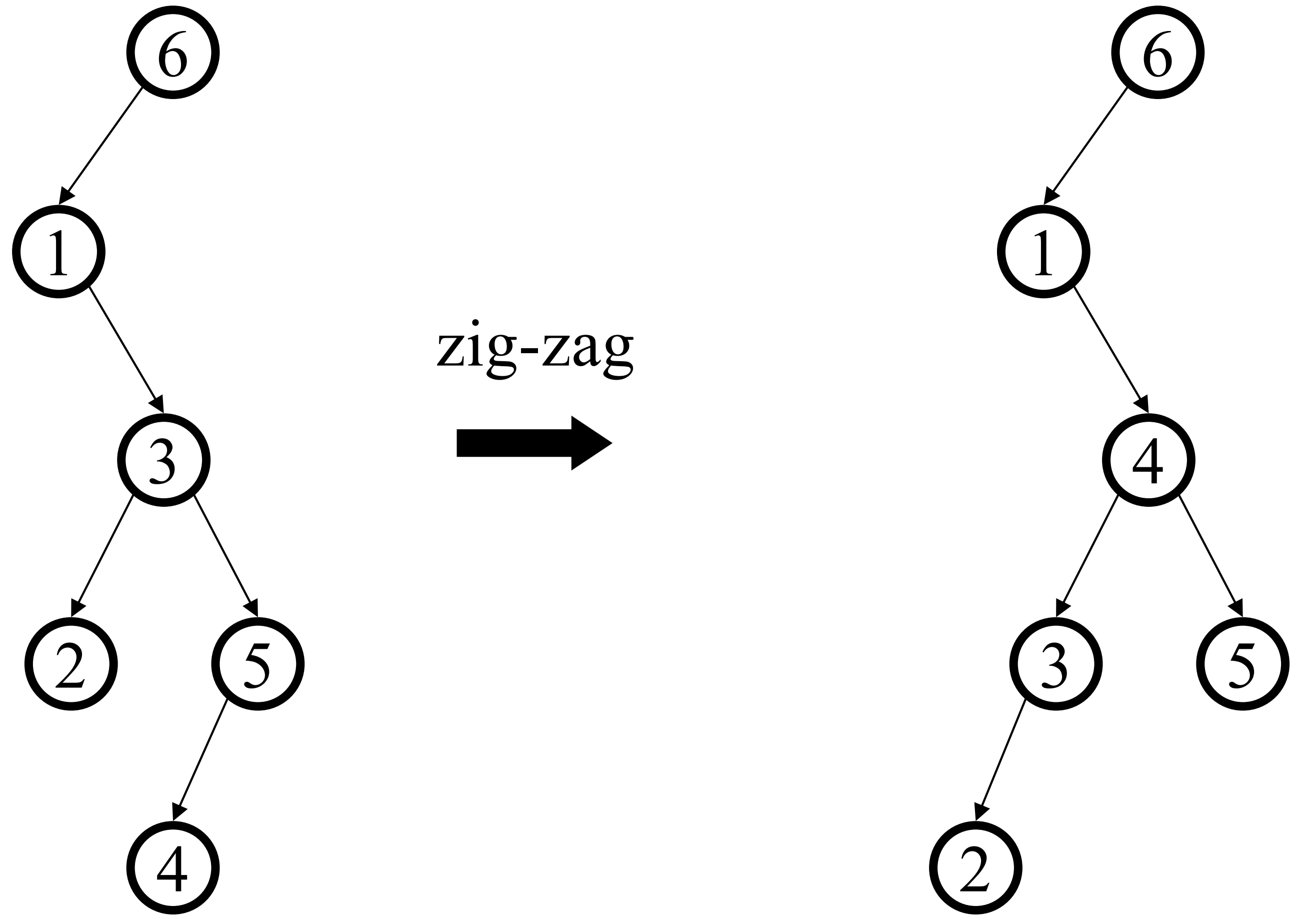


… 6 splayed out!

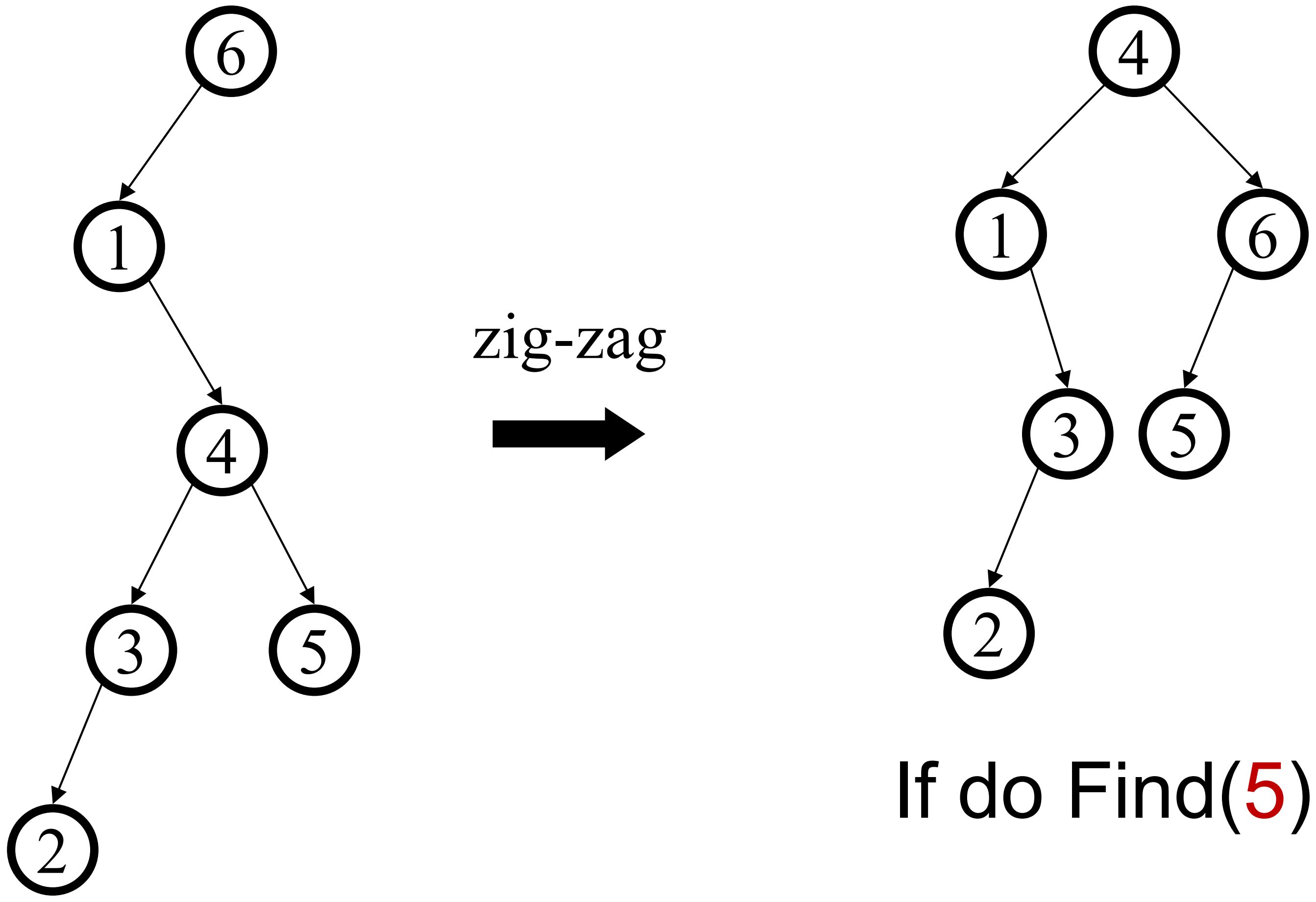


Splay it Again! Find (4)

Find(4)



… 4 splayed out!



If do Find(5)

Why Splaying Helps

- If a node *n* on the access path is at depth *d* before the splay, it's at about depth *d/2* after the splay
  - Exceptions are the root, the child of the root, and the node splayed
- Overall, nodes which are below nodes on the access path tend to move closer to the root
- Splaying gets amortized O(log n) performance. (Maybe not now, but soon, and for the rest of the operations.)

Splay Operations: Insert

- Ideas?
- Can we just do BST insert?

Splay Tree: Splitting

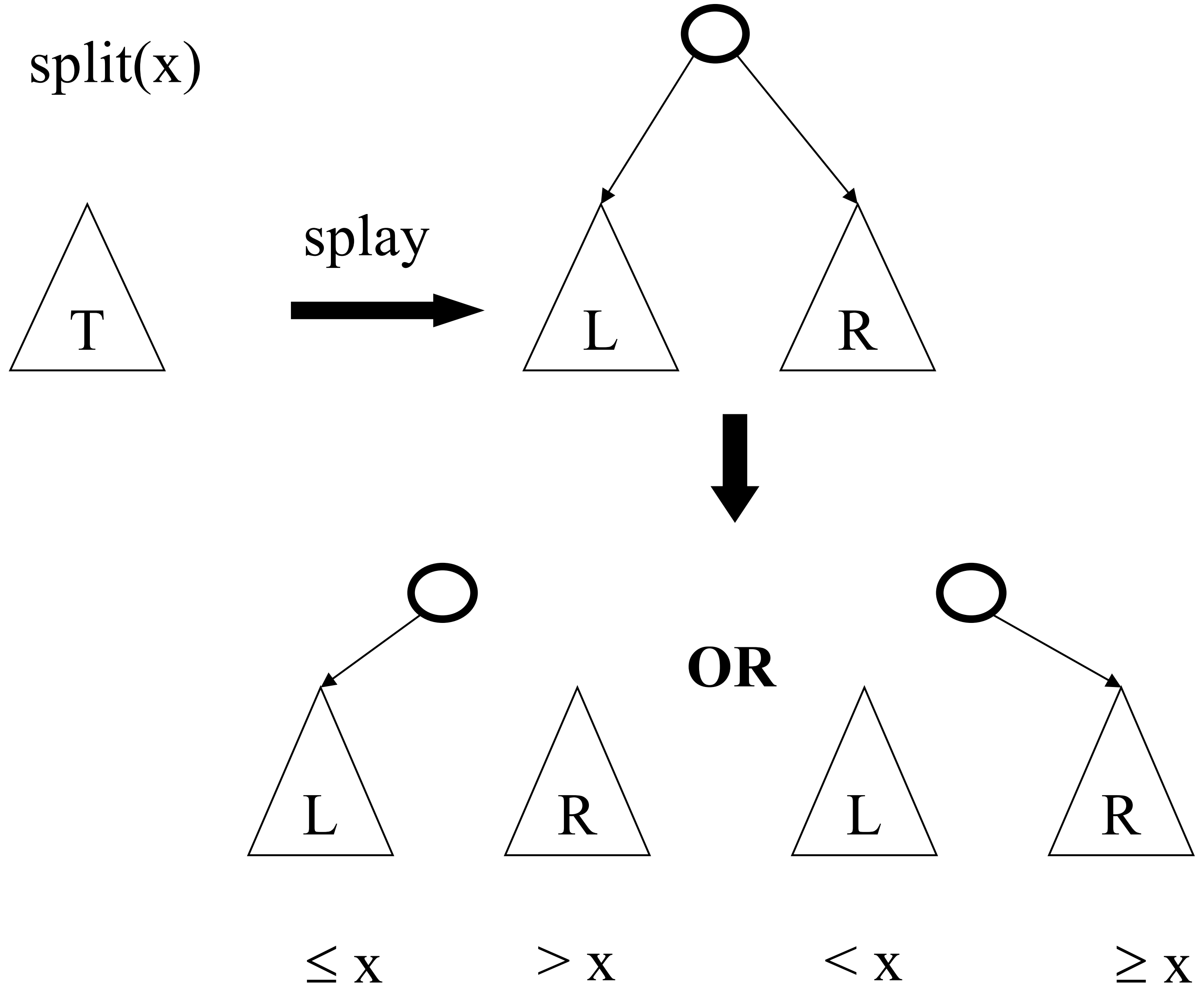
- Split(T, x) creates two BSTs L and R:
  - all elements of T are in either L or R ($T = L \cup R$)
  - all elements in L are $\leq$ x
  - all elements in R are $\geq$ x
  - L and R share no elements ($L \cap R = \varnothing$)

Splitting in Splay Trees

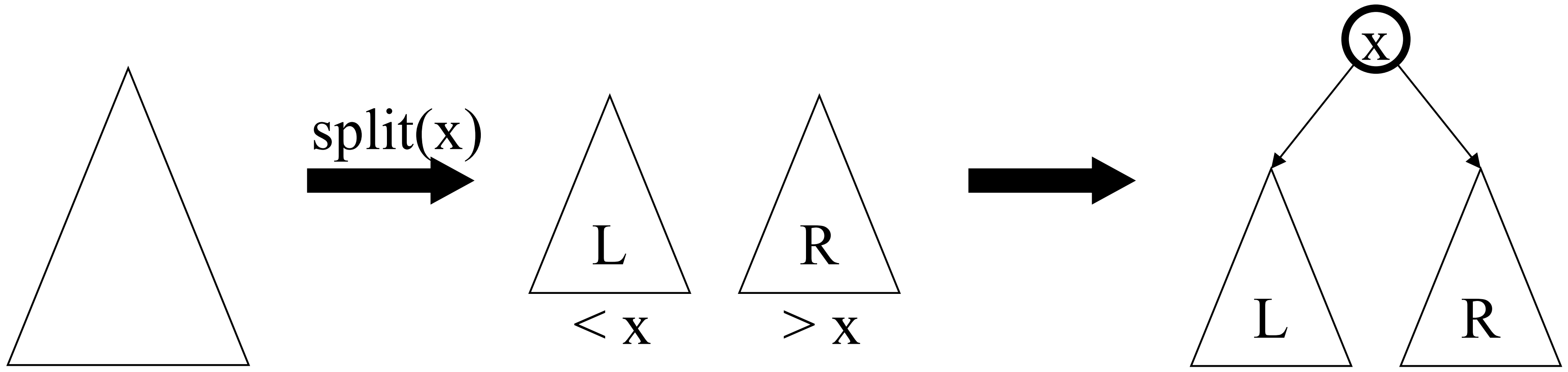
How can we split?

- We have the splay operation.
- We can find x or the parent of where x should be.
- We can splay it to the root.
- Now, what’s true about the left subtree of the root?
- And the right?

Splitting in Splay Trees

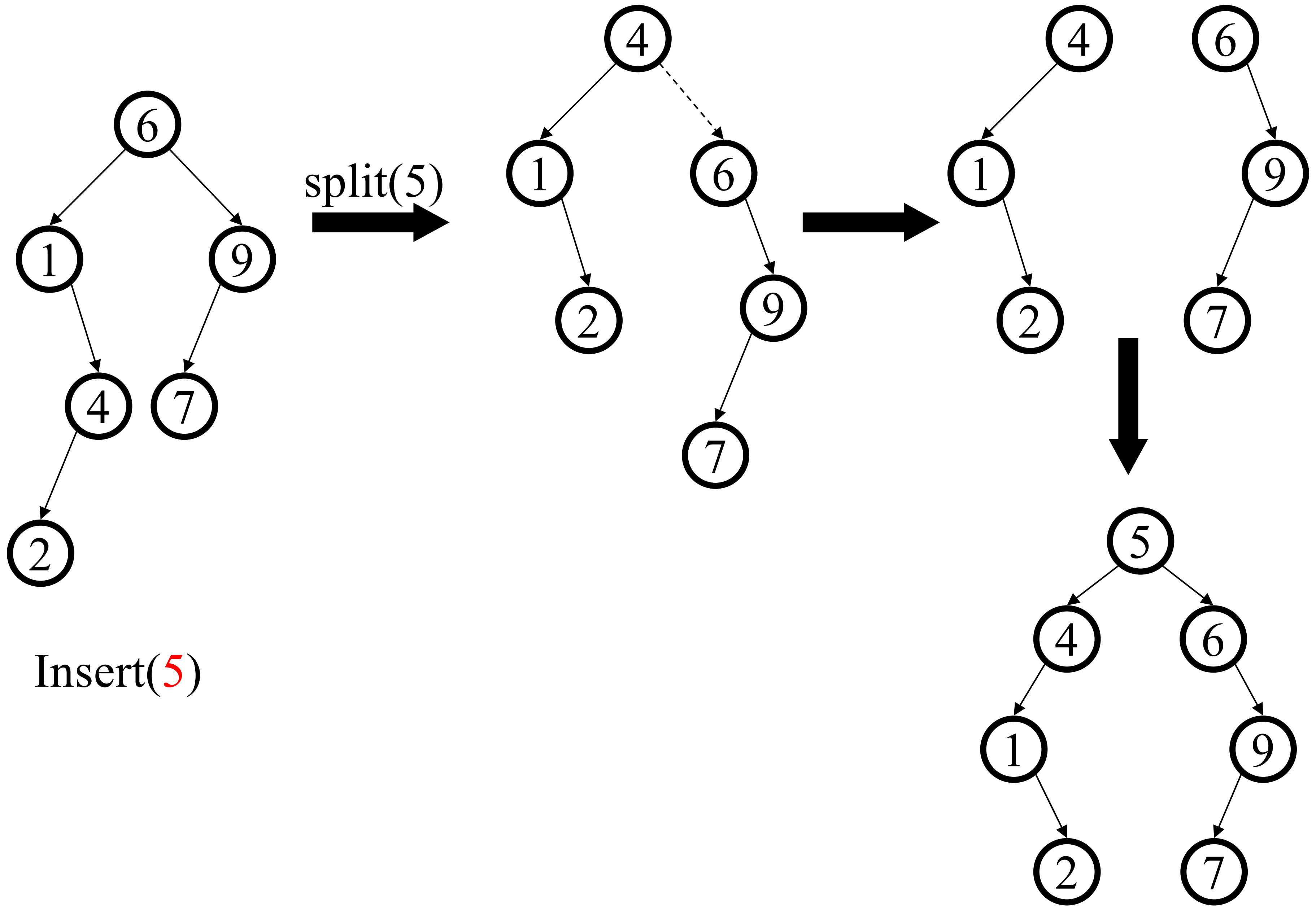


Splay Operations: Insert



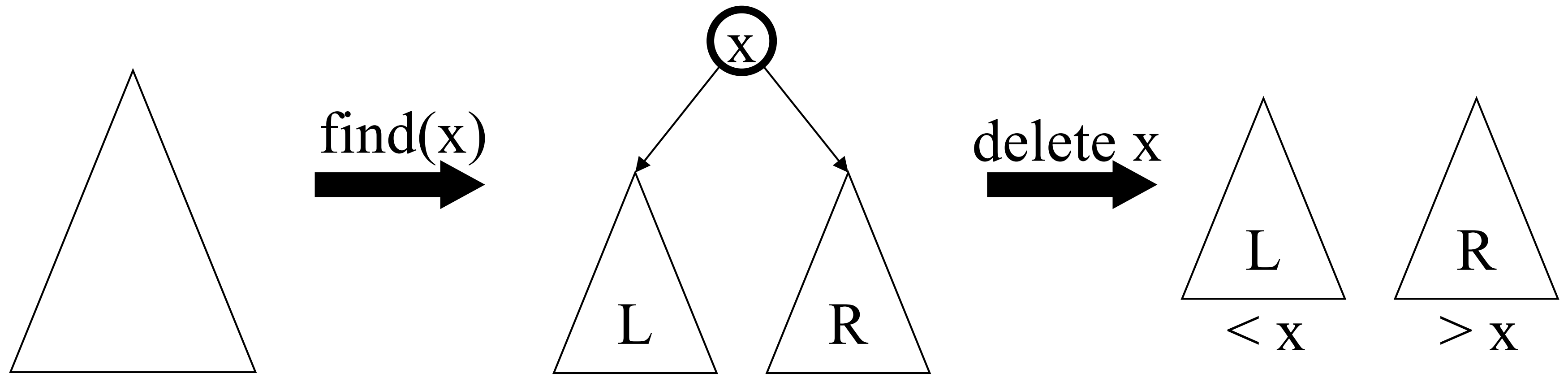
```
void insert(Node *& root, Object x)
{
  Node * left, * right;
  split(root, left, right, x);
  root = new Node(x, left, right);
}
```

Insert Example



Insert(5)

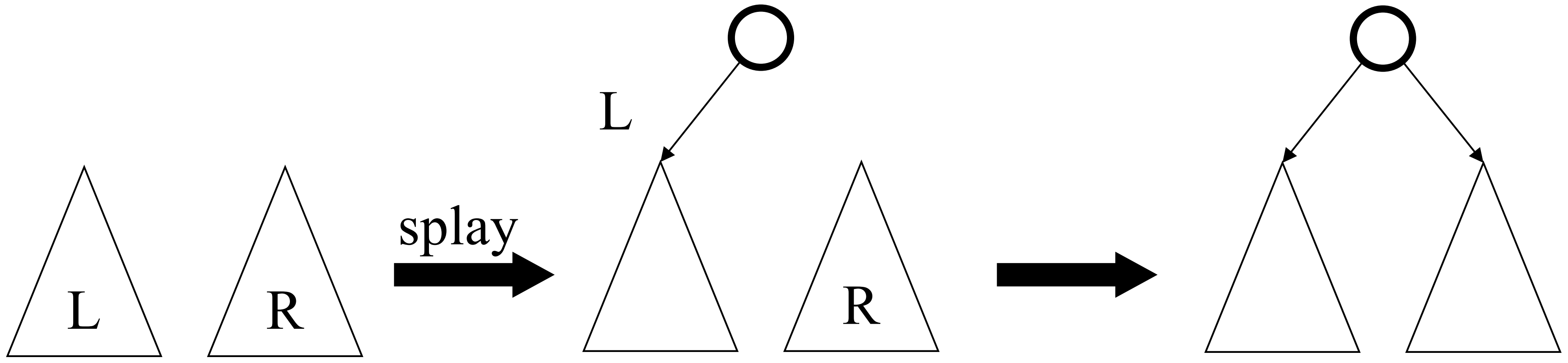
Splay Operations: Delete



Now what?

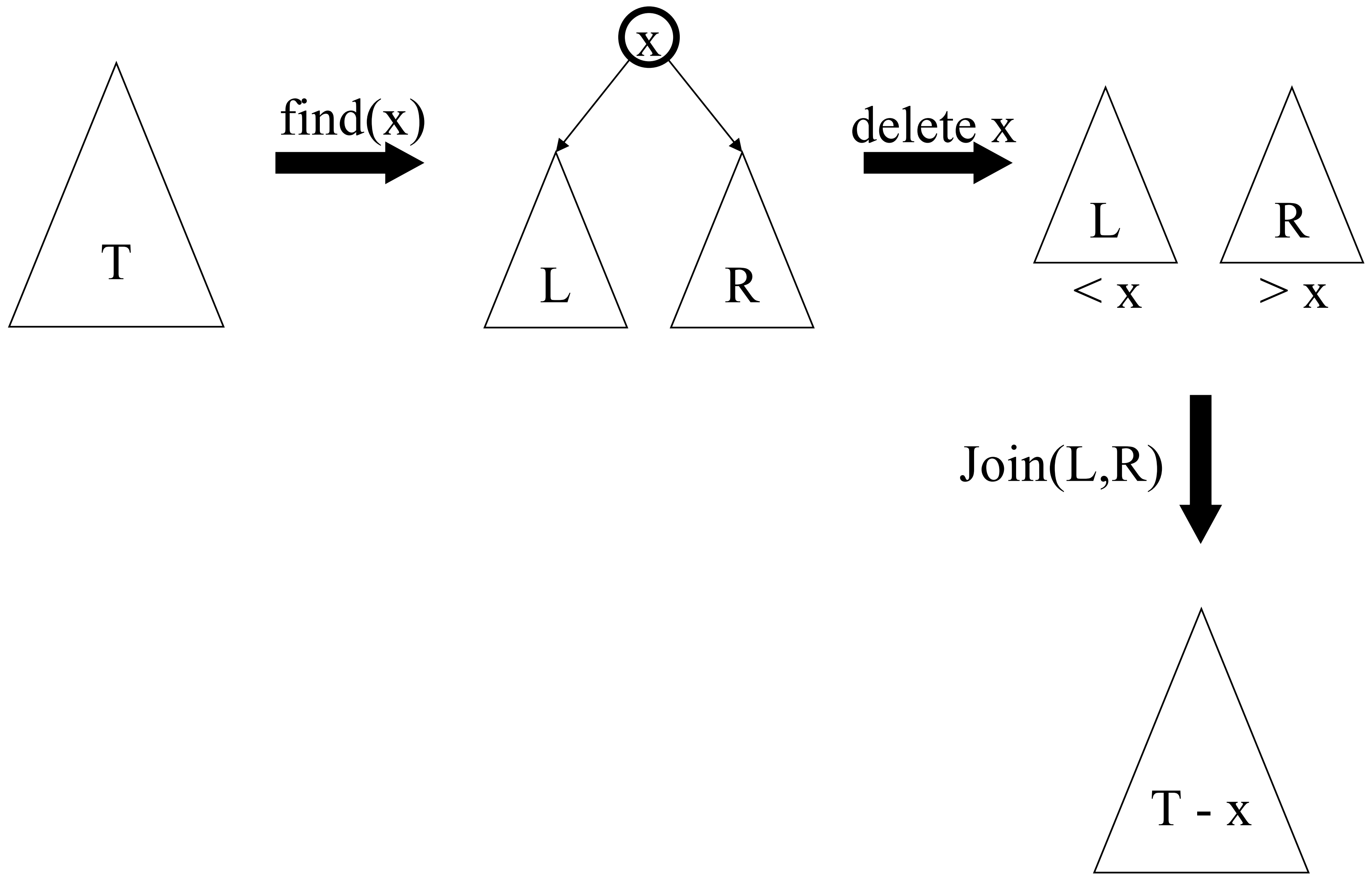
Splay tree: Join

Join(L, R): given two trees such that L < R, merge them

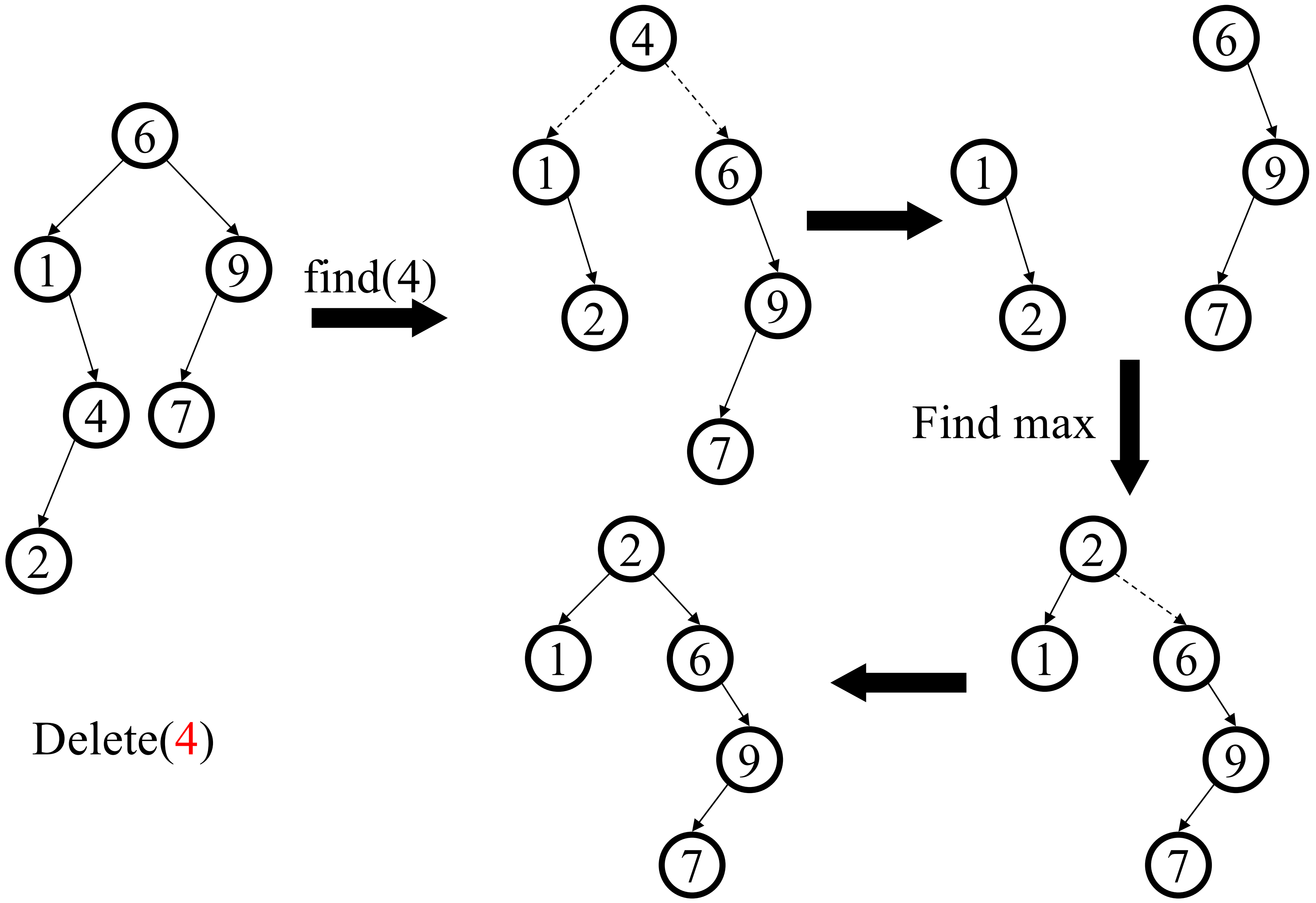


Splay on the maximum element in L, then attach R

Delete Completed



Delete Example



Delete(4)

Splay Tree Summary

- Can be shown that any *M* consecutive operations starting from an empty tree take at most O(M log(N))

  → All splay tree operations run in amortized O(log N) time

- O(N) operations can occur, but splaying makes them infrequent

- Implements most-recently used (MRU) logic
  - Splay tree structure is self-tuning

Splay Tree Summary

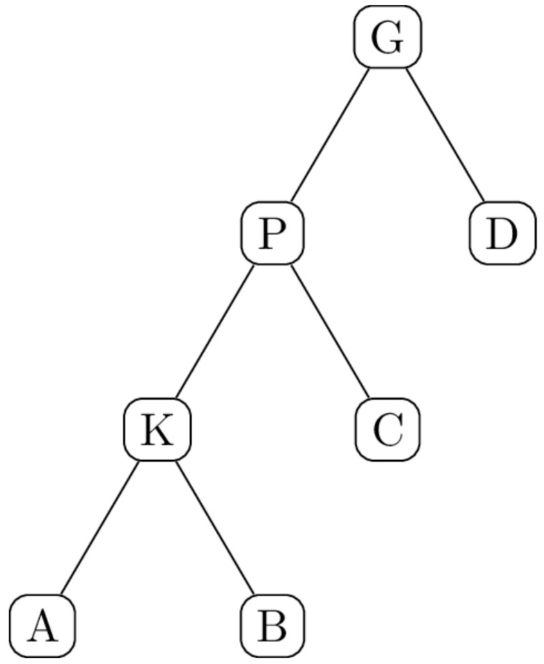
- Splaying can be done top-down; better because:
  - only one pass
  - no recursion or parent pointers necessary

- There are alternatives to split/insert and join/delete

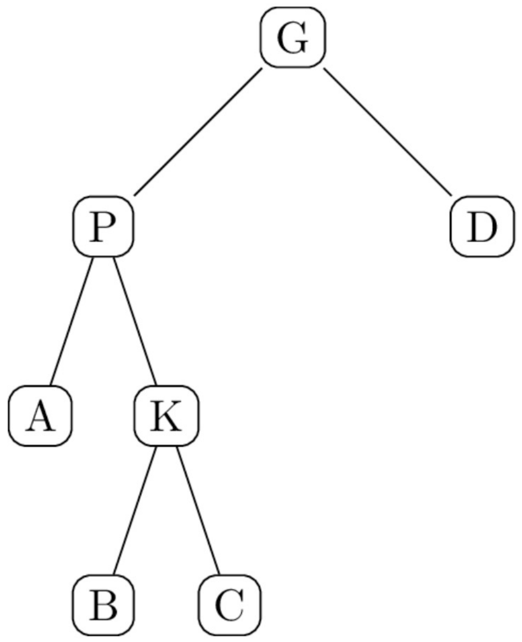
- Splay trees are *very* effective search trees
  - relatively simple: no extra fields required
  - excellent **locality** properties:
    - frequently accessed keys are cheap to find (near top of tree)
    - infrequently accessed keys stay out of the way (near bottom of tree)

Splay Tree Exercise 1

Show the resulting splay trees after performing Find(K).



(a)



(b)

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

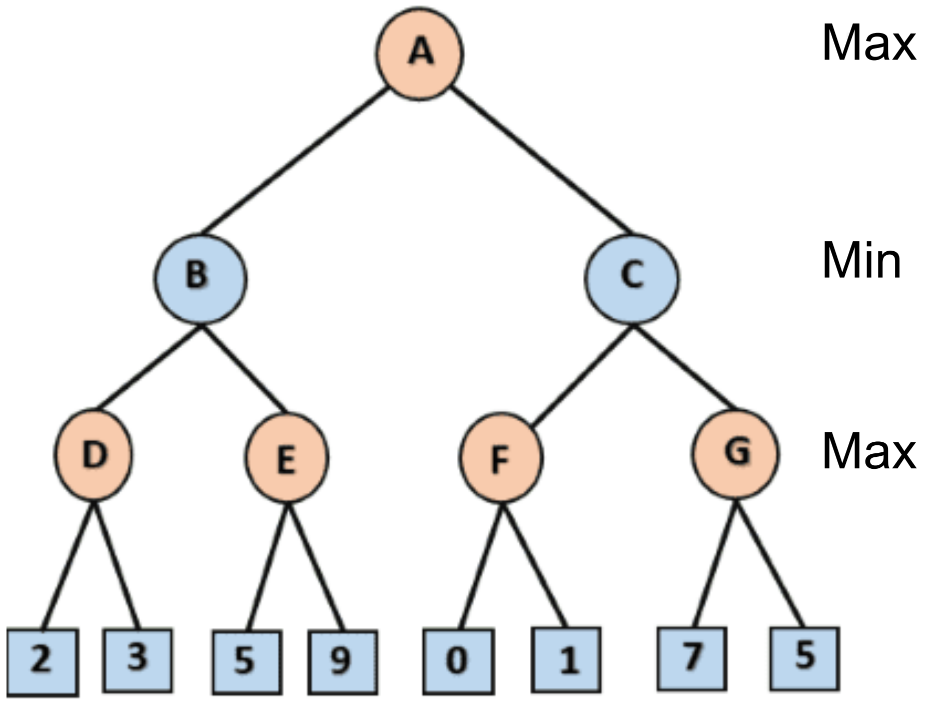
D:1; C:1,2; B:1,2,3; A:1,2,3,4

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

Exercise 1

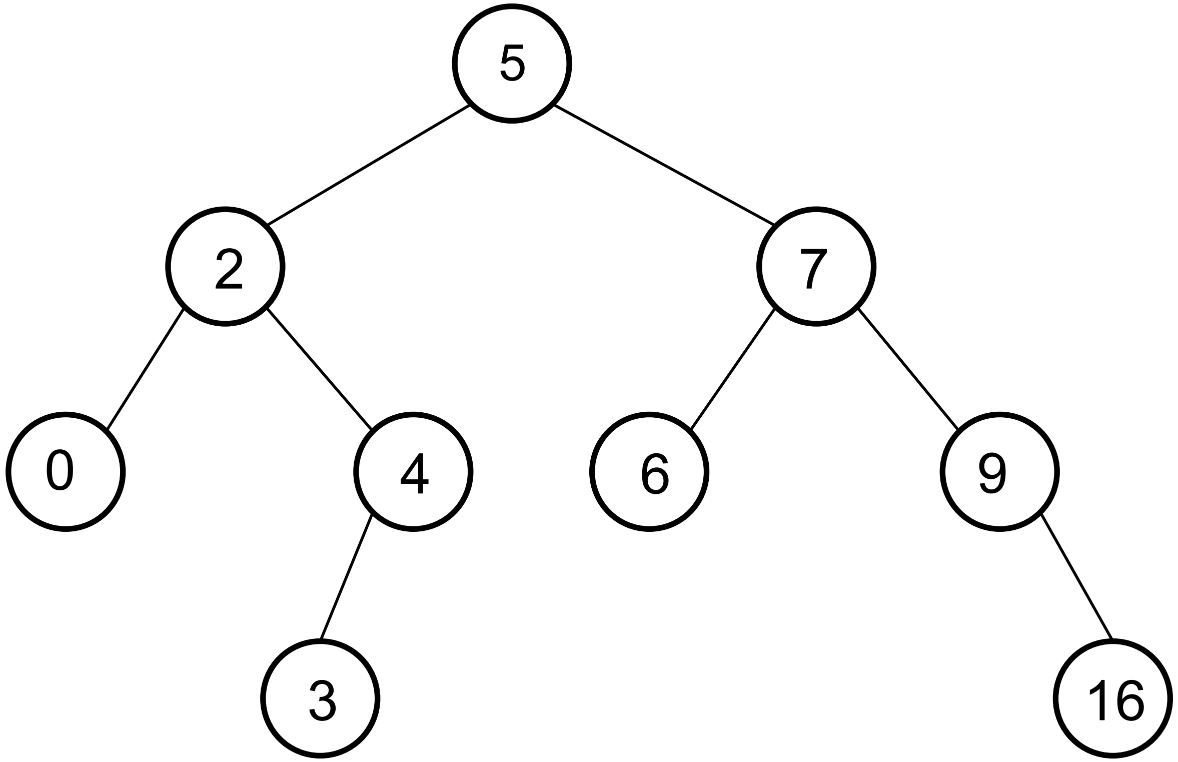


Two player are playing a game where player 1 takes the first move and the value specified for a certain leaf is the amount of dollars player 1 can get if the game process follows the path from root to that leaf.

Given the game tree and providing ***minimax*** algorithm with ***Alpha-Beta Pruning***, describe the best first move for player 1 and list all each leaf nodes that will be pruned.

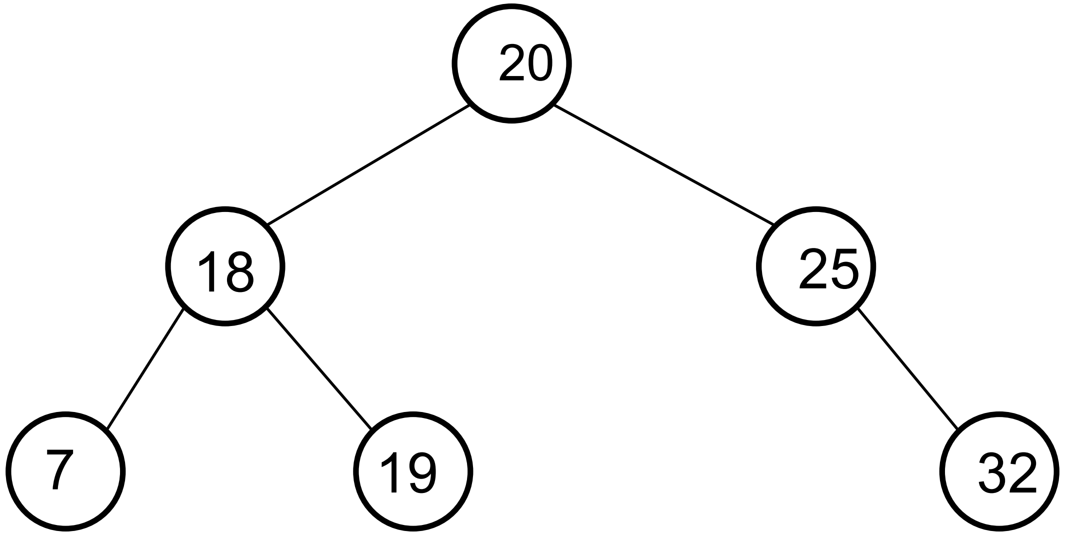
Exercise 2

Showing each step of inserting 15 to the following AVL tree.



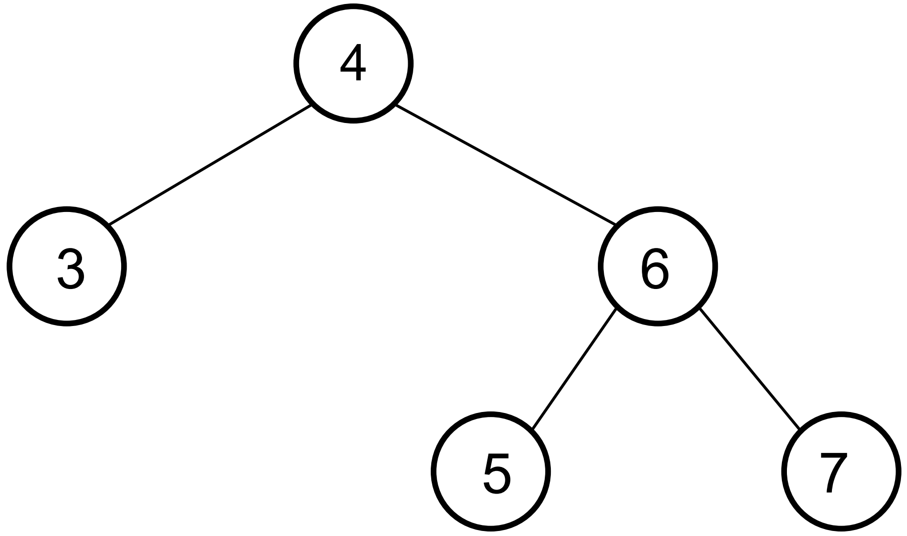
Exercise 3

Showing the final AVL tree after inserting the following numbers:
{18, 20, 7, 25, 32, 19}

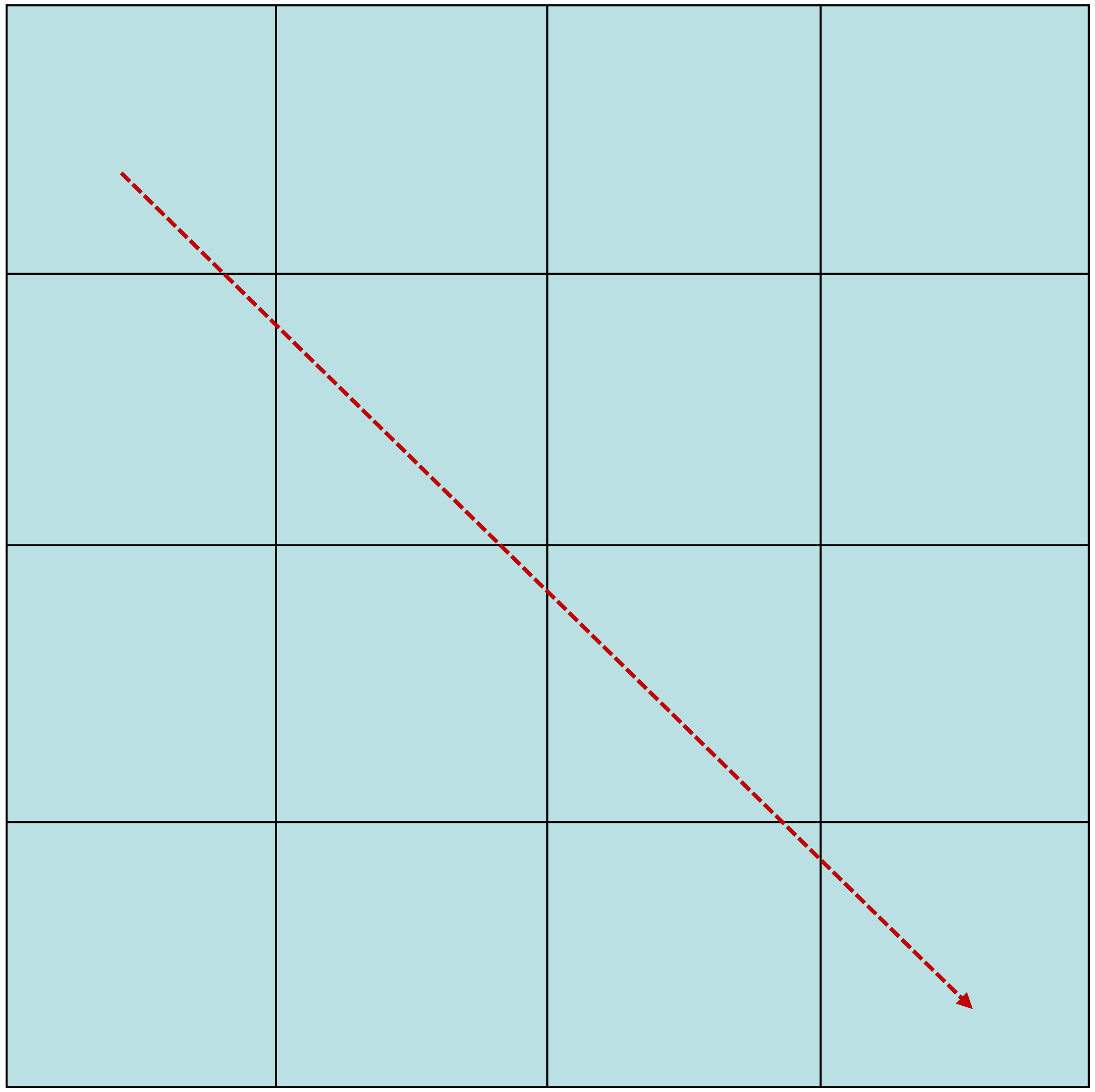


Exercise 4

Showing each step of find(5) in the following splay tree.



Generate a Maze



Objective - Disjoint Set

- Equivalence relations
- Equivalence classes
- Disjoint set Abstract Data Type
- Array Implementation
- Application——Maze generation

Equivalence Relations

- A relation R is defined on **a set S** if for every pair of elements (*a*,*b*), *a,b* in S, *a* R *b* is true or false
- If *a* R *b* is true, we say that a is related to b
- R is an equivalence relation if
  - (Reflexive) *a* R *a* is true for all element *a* in set S
  - (Symmetric) *a* R *b* → *b* R *a*, *b* R *a*→ *a* R *b*
  - (Transitive) *a* R *b* and *b* R *c* implies *a* R *c*
- If R is an equivalence relation, then R divide S into several disjoint components, elements in the same component is equivalent to each other
  - The relation “same country” divide the world map into different countries

Equivalence Relations

- Given an equivalence relation ~, the natural problem is to decide,

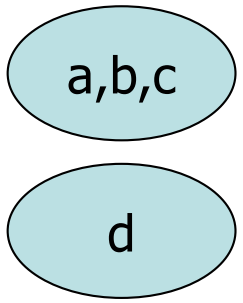
  for any a and b, whether a ~ b.

	a1	a2	a3	a4	a5
a1	1	1			1
a2	1	1		1	
a3			1	1	
a4		1	1	1	
a5	1				1

a1 ~ a2
a3 ~ a4
a5 ~ a1
a4 ~ a2

Equivalence Classes

- An equivalence class of an element a is the subset of all the elements that are equivalent to a
- Equivalence classes are disjoint
- Example:
  - Elements:a,b,c,d
  - if a~b,b~c
  - There are two equivalence classes
  - Elements can be represented like:



Exercise :

- Elements: a,b,c,d,e,f,g,h,i,j,k
- if a~b, b~c, b~d, e~f, g~h,i~e,j~k,k~c
- How many equivalence classes?
- Draw these equivalence classes

- Hint: use transitive rule to identify all the elements that are equivalent to ‘a’

Disjoint Set ADT

- Value:
  - A set of items that belong to some data type ITEM_TYPE
  - Each item is associated with an equivalence class name

- Operations on DS
  - **Find**(ITEM_TYPE a)
    - ❑ return the equivalence class name for a
    - ❑ If Find(a)==Find(b), then a and b are in the same equivalence class
  - **Union**(ITEM_TYPE a, ITEM_TYPE b)
    - ❑ Combine a's equivalence class with b's equivalence class (make them the same name)
    - ❑ Precondition (Suppose): Find(a)==Find(c), Find(b)==Find(d)
    - ❑ Postcondition: a,b,c,d are in the same equivalence class (their equivalence class name are the same.)

Array Implementation

- Array s[]
- s[i] stores the equivalence class name for i

0	1	2	3	4	5	6	7	8	9

- find(i): return s[i] worst case O(1)
- Union(i,j): change all the value s[j] into s[i] in the array: O(N)
- Example:
  - Union(1,2)
  - Union(3,1)
  - Find(1): return __

For Find(), the return value is not important.
The important thing is whether Find(i) and Find(j) are the same

0	1	2	3	4	5	6	7	8	9
0	1	1	3	4	5	6	7	8	9
0	3	3	3	4	5	6	7	8	9

Array Implementation

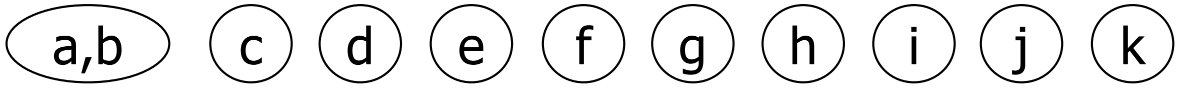
```
class DisjointSet
{
        public:
                int Find(int )
                void Union(int,int)
        private:
                int name[MAX_SIZE]
}
```

```
int DisjointSet::Find(int i)
{
        return name[i];
}



void DisjointSet::Union(int i, int j)
{
        if Find(i)!=Find(j)
        {
                int temp=name[j];

                for (k=0;k<MAX_SIZE;k++)
                {
                        if(name[k]==temp)
                                name[k]=name[i];
                }
        }
}
```

Exercise

Elements: a,b,c,d,e,f,g,h,i,j,k

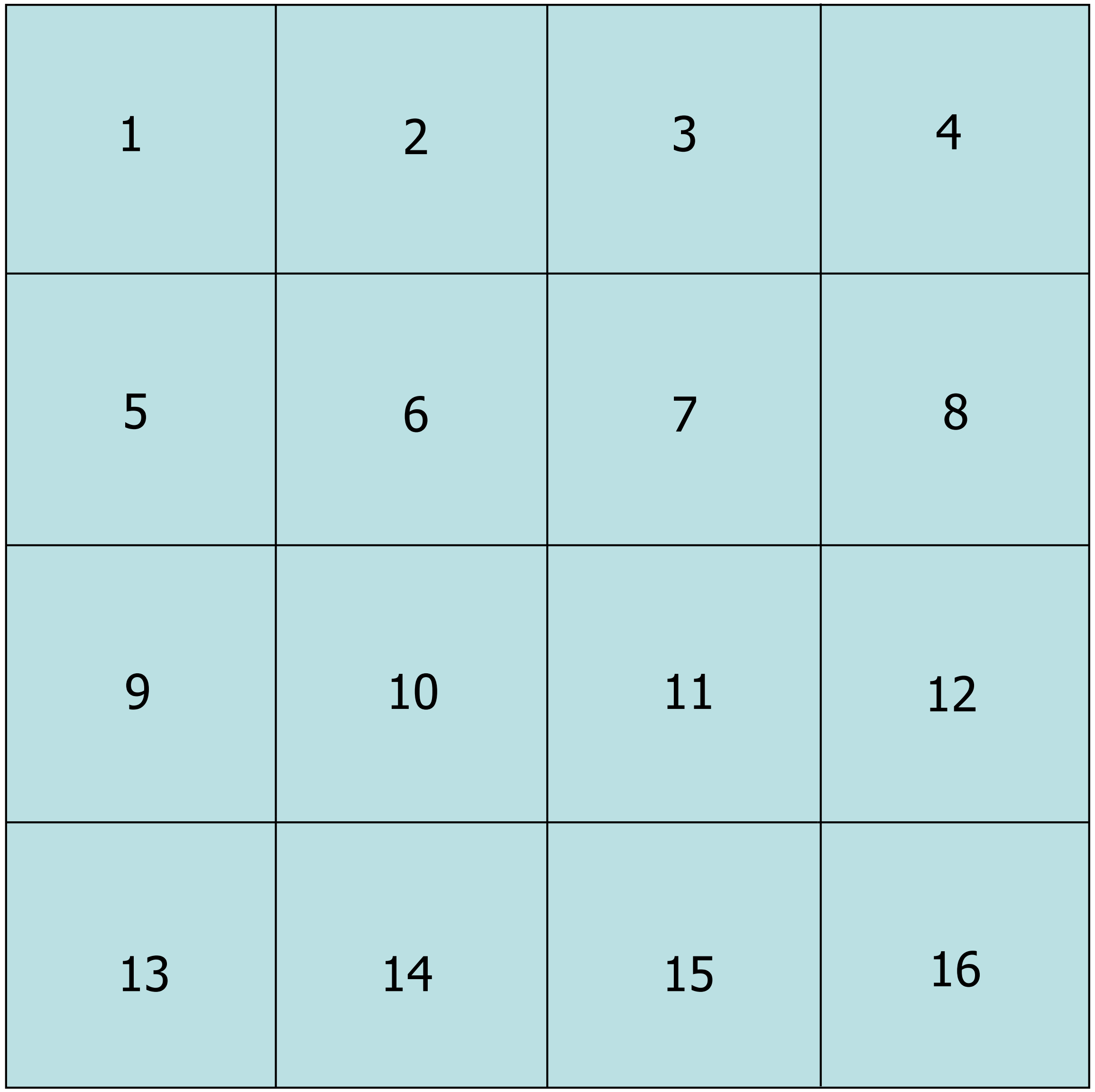
- if a~b, b~c, b~d, e~f, g~h, i~e, j~k, k~c
- Method:
  - Initially, all 11 elements are distinct
  - After adding a~b:
  - After adding b~c:
  - After adding b~d:
  - After adding e~f:
  - After adding g~h:
  - After adding i~e:
  - After adding j~k:
  - After adding k~c:



Maze Generation

```
While the entrance cell is not connected to the exit cell
    Randomly pick a wall (two adjacent cells i and j);
    If (Find(i)!=Find(j))
        Union(i,j);        //Break the wall
```

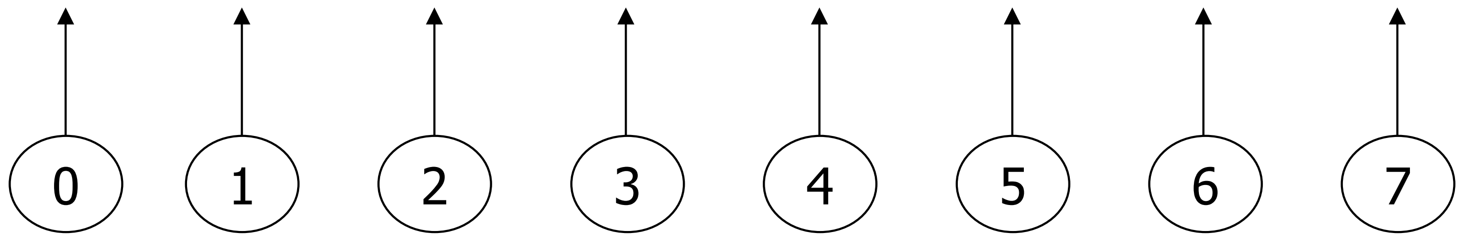
Generating a Maze



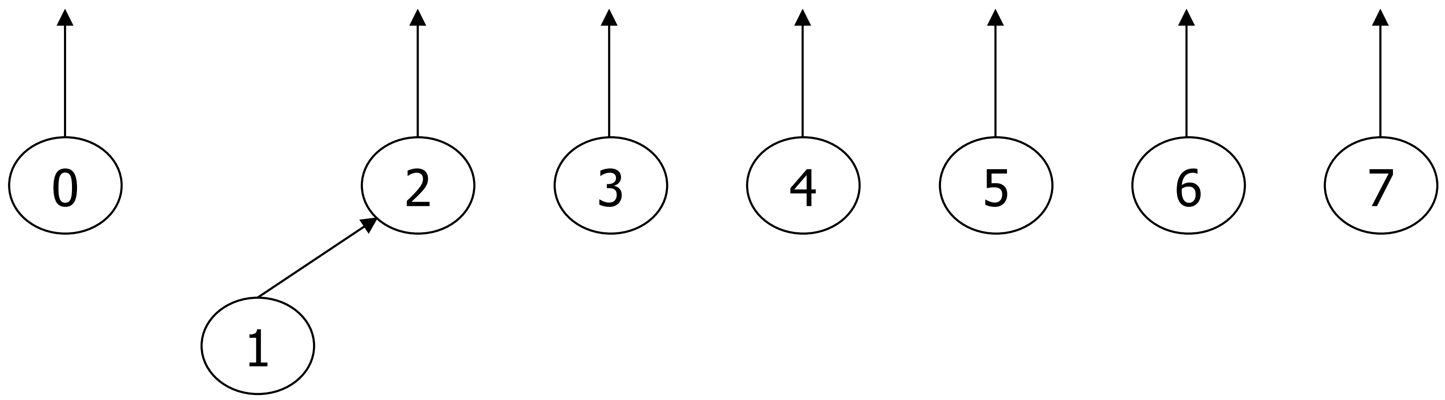
- Union(1,2)
- Union(4,8)
- Union(6,7)
- Union(14,15)
- Union(5,9)
- Union(3,7)
- Union(2,6)
- Union(8,12)
- Union(3,4)
- Union(11,12)
- Union(11,15)
- Union(15,16)

Disjoint Set (Tree Implementation)

- Initially, all the nodes are independent

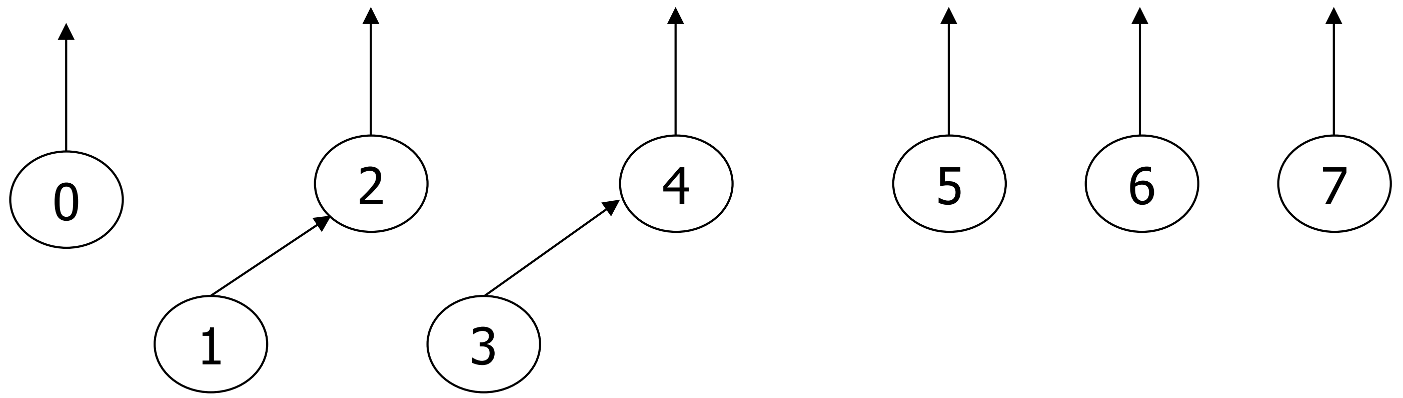


- After Union(1,2)

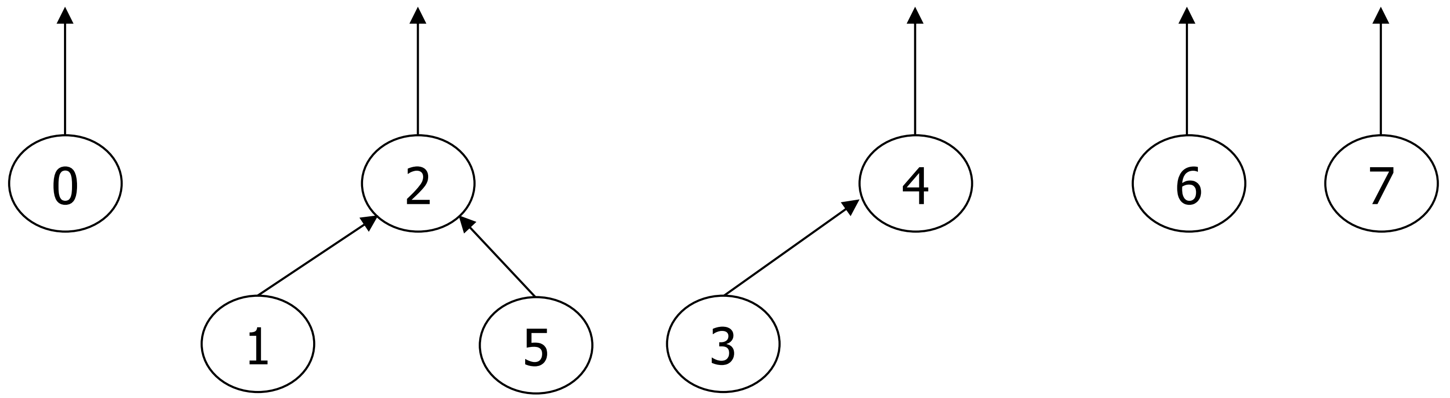


Disjoint Set (Tree Implementation)

- After Union(3,4)

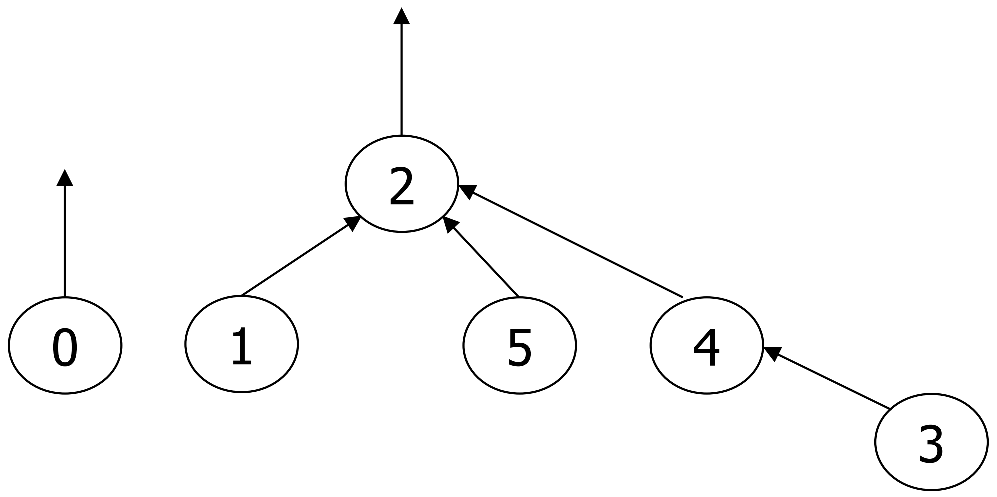


- After Union(1,5)

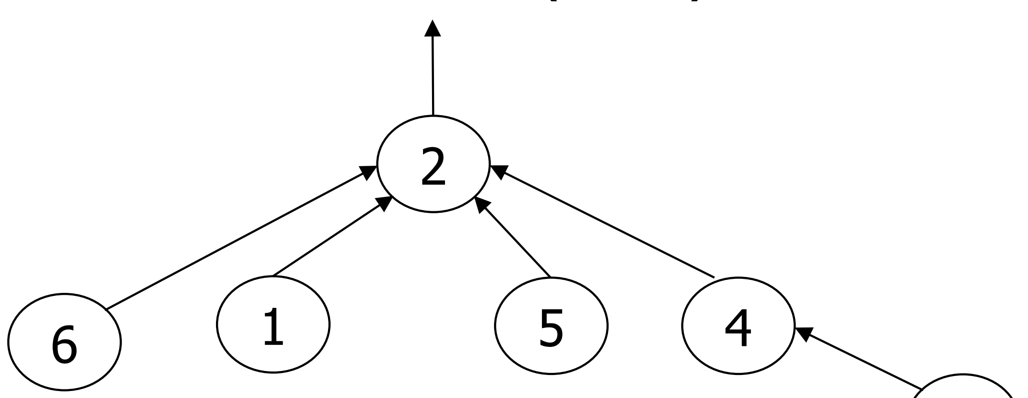


Disjoint Set (Tree Implementation)

- After Union(1,3)

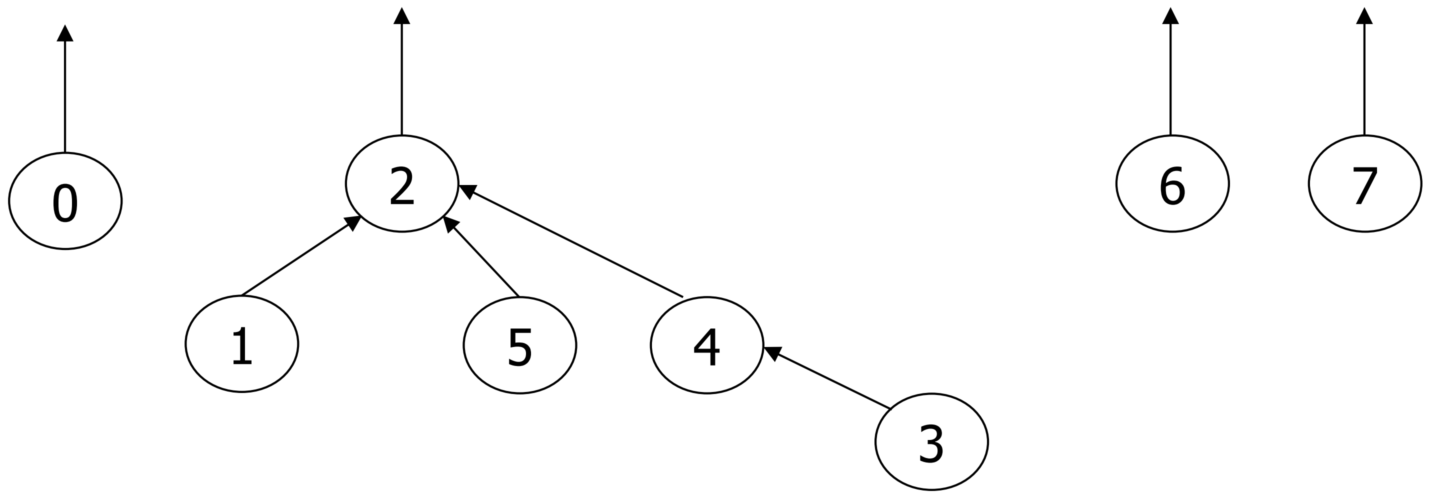


- After Union(4,6)



Can we use array?

- Can we use array to represent link?



-1	2	-1	4	2	2	-1	-1
0	1	2	3	4	5	6	7

Disjoint Set (Tree Implementation)

- Rule for Find(*i*)
  - Follow upward links of node *i* until we can go no further by following links

- Rule for Union(*i*,*j*)

  Find(i): Find the root r(*i*) of the tree which *i* belongs to

  Find(j): Find the root r(*j*) of the tree which *j* belongs to

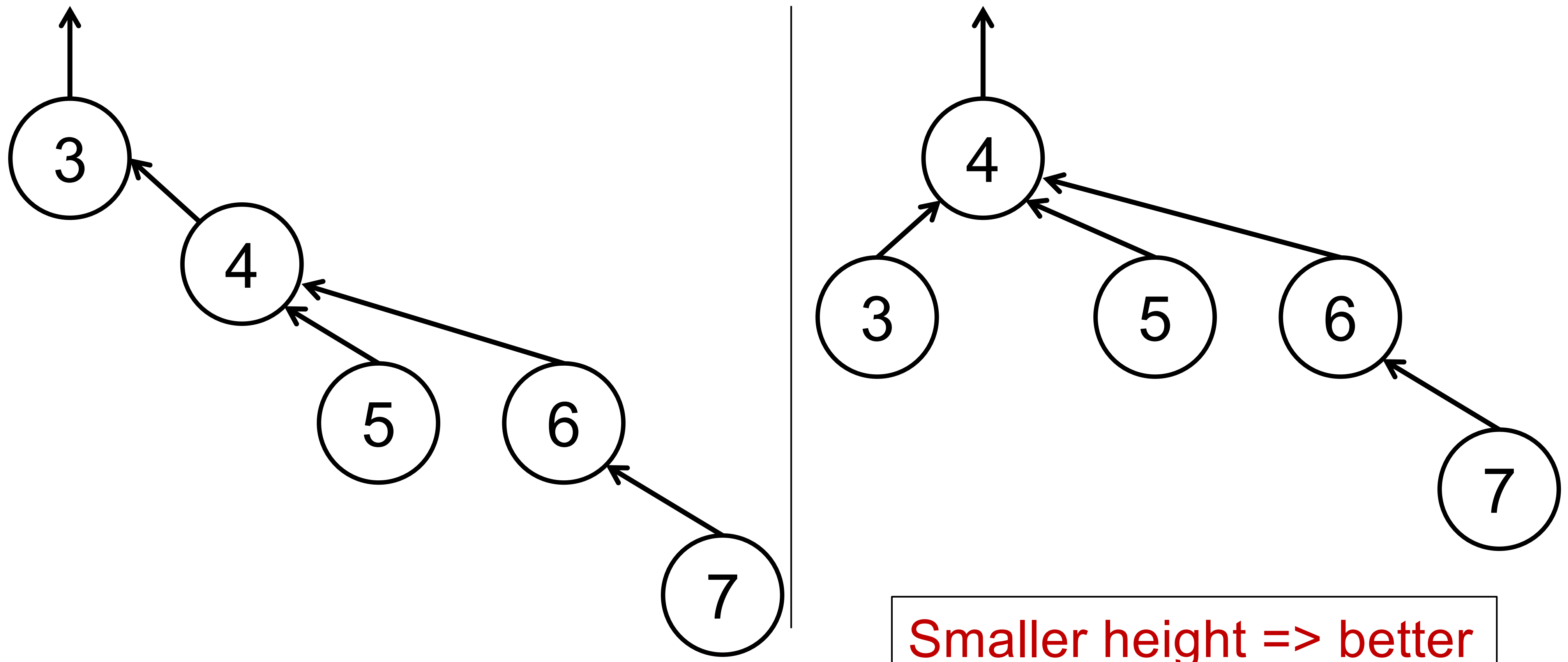
  r(*i*)->next=r(*j*)

  Connecting rule:

  - smaller tree to larger tree
  - shallower tree to deeper tree

Smart Union

- Which result is better for Find(7) after Union(3,4)?



Smaller height => better

Union-by-Height

- Guarantee all the trees have depth at most ***O(log N)***, where N is the total number of elements
- Running time for a find operation is ***O(log N)***
- The height of a tree increases one only when two equally deep trees are joined.

Union-by-Height: Exercise

- Show the results of the following sequence of instructions (using an array):
  - Union(0, 1); Union(2, 3); Union(4, 5); Union(1, 3); Union(5, 6); Union(5, 7); Union(5, 8); Union(3, 5).

We perform them with Union-by-Height, and when the heights are the same, we connect the first tree to the second tree.

1	**3**	**3**	**-1**	**5**	**3**	**5**	**5**	**5**	**-1**

Theorem 1

- **Theorem 1:** Union-by-height guarantees that all the trees have depth at most O(log N), where N is the total number of elements.

- **Lemma 1.** For any root x, size(x) ≥ $2^{\text{height}(x)}$, where size(x) is the number of nodes of x’s tree, including x.

Proof of Lemma 1 (1/3)

- Proof (by induction)
  - **Base case:** At beginning, all heights are 0 and each tree has size 1.
  - **Inductive step:** Assume true just before Union(x,y). Let size'(x) and height'(x) (or size'(y) and height'(y)) be the size and length of the merged tree after union, respectively.

Proof of Lemma 1 (2/3)

- **Case 1. height(x) < height(y)**
  Then, size'(y) = size(x) + size(y)

$$\geq 2^{\text{height}(x)} + 2^{\text{height}(y)}$$
$$\geq 2^{\text{height}(y)}$$
$$= 2^{\text{height}'(y)}$$

- **Case 2. height(x) = height(y)**
  Then, size'(y) = size(x) + size(y)

$$\geq 2^{\text{height}(x)} + 2^{\text{height}(y)}$$
$$\geq 2(2^{\text{height}(y)})$$
$$\geq 2^{\text{height}(y)+1}$$
$$= 2^{\text{height}'(y)}$$

Proof of Lemma 1 (3/3)

- **Case 3. height(x) > height(y)**
  Then, $\text{size}'(x) = \text{size}(x) + \text{size}(y)$

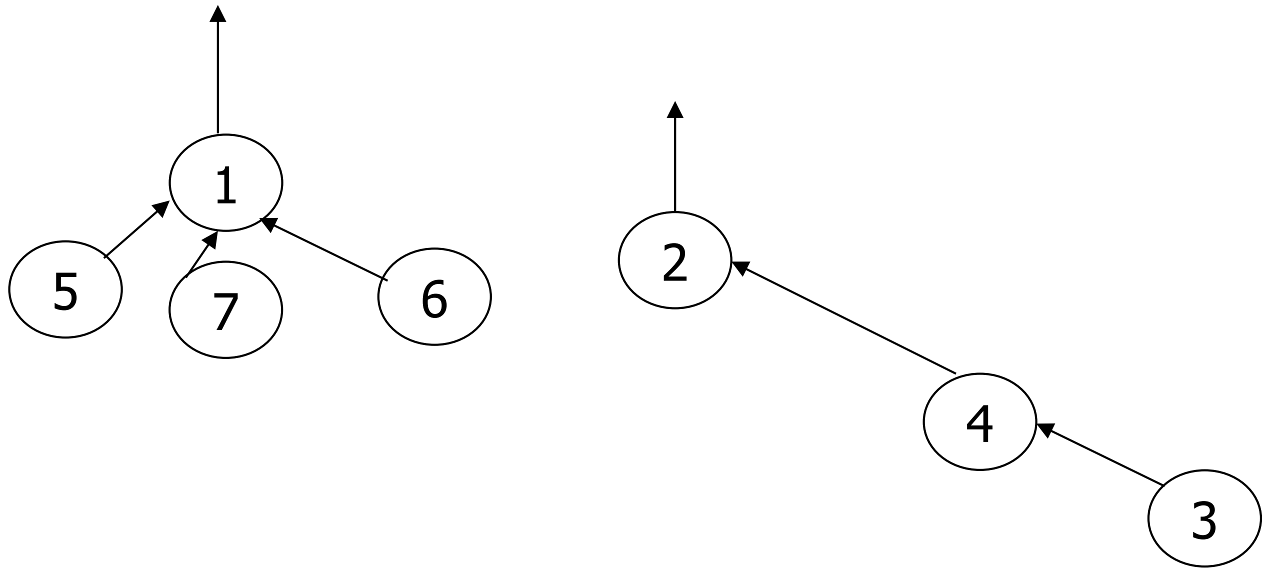
$$\begin{aligned} &\geq 2^{\text{height}(x)} + 2^{\text{height}(y)} \\ &\geq 2^{\text{height}(x)} \\ &= 2^{\text{height}'(x)} \end{aligned}$$

Proof of Theorem 1

- By Lemma 1, $\text{size}(x) \geq 2^{\text{height}(x)}$.
- Let $n=\text{size}(x)$ and $h=\text{height}(x)$.
- Every node has size $n \geq 2^h$
  $\Rightarrow \log n \geq h$
  $\Rightarrow h = O(\log n)$

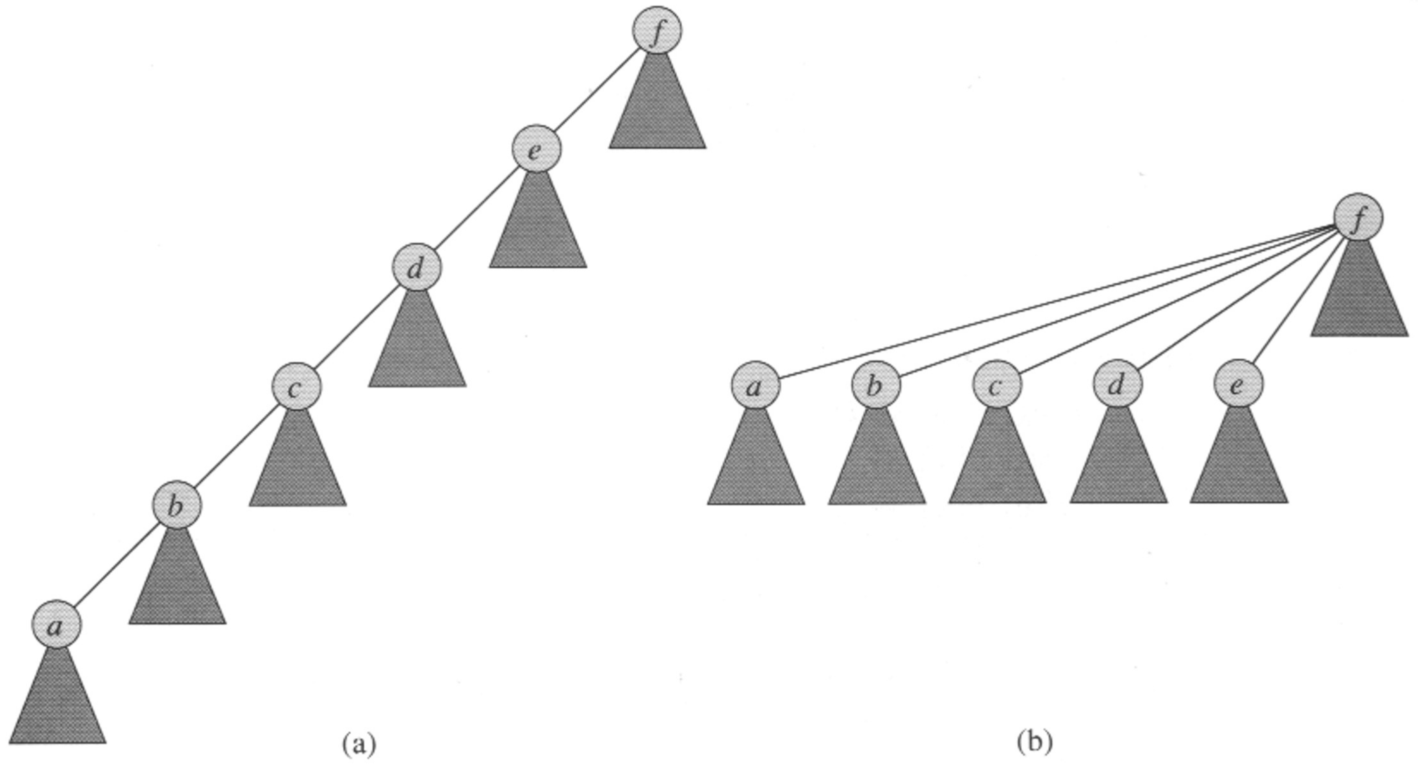
Union by Size

- Smaller size tree link to bigger size tree
- Is it good?



- Can also guarantee h=O(logn)
- How to prove?

How to implement : Enhancement



Path Compression

```
int Find (int element)   {
      if (A[element] < 0)
            return element;
      else
            return A[element] = Find(A[element]);
}
```

Path Compression!

Whenever Find() is performed, all items along the path update its parent to the topmost root

Next query (Find()) will be faster…

Union-by-Height with path compression

- Show the results of the following sequence of instructions (using an array):
  - Union(0, 1); Union(2, 3); Union(4, 5); Union(1, 3); Union(5, 6); Union(5, 7); Union(5, 8); Union(3, 5).

  We perform them with Union-by-Height, and when the heights are the same, we connect the first tree to the second tree.

3	**3**	**3**	**-2**	**3**	**3**	**3**	**3**	**3**	**-1**

Combination

- Union by Height together with Path Compression has an amortized running time O(a(n)) where a(n) is the inverse Ackermann function
- Ackermann function

  A(m,n) =
    - n+1 (if m=0)
    - A(m-1,1) (if m>0 and n=0)
    - A(m-1, A(m,n-1)) (if m>0 and n>0)

Inverse Ackermann Function

- Single parameter one (roughly)
  - A(i)=2^{A(i-1)}
  - a(n)=min{i>=1, A(i)>log n}
- Examples:
  - if A(1)=1, then A(2)=2^1=2
  - A(3)=2^2=4, A(4)=2^4=16, A(5)=2^16=65536
  - A(6)=2^65536
- Hence, usually a(n) is very small
  - if n=2^65535, a(n)=?
- We also use log*(n) to represent this a(n)

Learning Objectives

1. Understand the concept of Disjoint Set
2. Able to analyze time complexities for operations on Disjoint Set
3. Understand the best implementation of Disjoint Set
4. Able to use Disjoint Set to solve problems

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

Objective - Suffix Array

- Suffixes
- Exact Pattern Matching
- Suffix Tree and Suffix Array
- Sorting Suffixes

What is suffixes?

Given a string (or text)

$T = t_1\ t_2\ t_3\ \ldots\ t_{n-1}\ t_n$

then it has **n** suffixes,

they are

Suffix(T,i) = S[i]

$= t_i\ t_{i+1}\ t_{i+2}\ \ldots\ t_n$

for

$1 \leq i \leq n$

```
T    = innovation
S[1]= innovation
S[2]=  nnovation
S[3]=   novation
S[4]=    ovation
S[5]=     vation
S[6]=      ation
S[7]=       tion
S[8]=        ion
S[9]=         on
S[10]=         n
```

Why suffixes?

- Prefix of a string $\mathtt{T = t_1\ t_2\ t_3\ \ldots\ t_{n-1}\ t_n}$
  - $\mathtt{Prefix(T,i) = t_1\ t_2\ t_3\ \ldots\ t_{i-1}\ t_i}$
- Tricky (keep in mind please)
  - Any substring (or pattern) of `T`, must be a prefix of some suffix from `T`!
  - Example `T = mississippi,`
    `P =       ssip`

    then `P = Prefix( Suffix(T, 6), 4 )`

Exact Pattern Matching

- How do you find the occurrence of a pattern **`P`** in a text **`T`**?
  - Test for each **`i`** whether **`P`** is a prefix of **`Suffix(T,i)`**
- Naïve implementation: **O(`PT`)** time, too slow!
- Knuth-Morris-Pratt (1977, SIAM J. Comput.)
  - Key Point: ignore testing impossible suffixes
  - **O(`P`)** preprocessing **`P`**
    - Calculate **`Next(k)`** : which suffix should try next when the first **`k`** chars of **`P`** are matched in the current suffix?
  - **O(`P + T`)** searching for any text **`T`**
  - Will be covered in CS 4335