CSCI 335 Software Design and Analysis III

Trees/AVL trees

Chapter 4

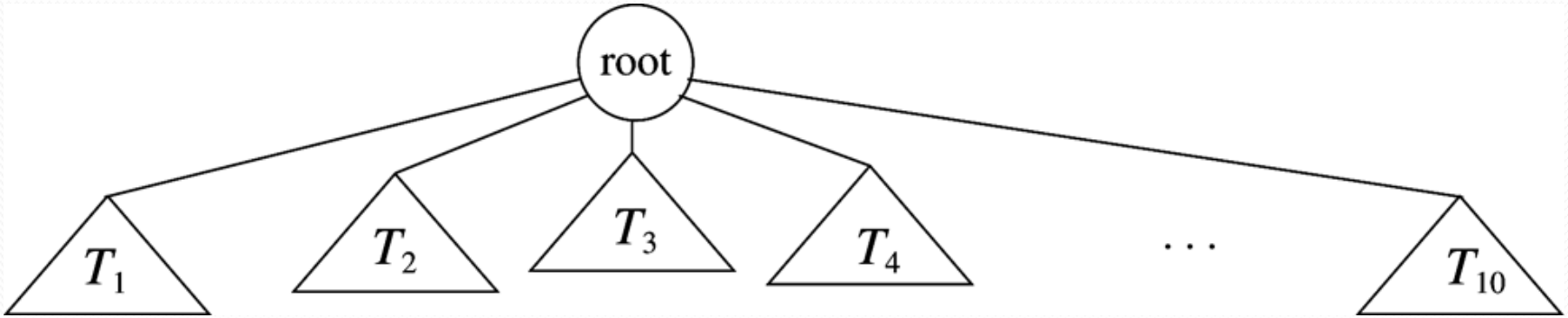
Trees

- Recursive definitions
  - A tree is either empty or consists a root node r and zero or more non-empty subtrees the roots of which are connected by an edge from r.
- Terms: Root, child, parent, leaves, siblings, path from node n_1 to n_k, length of path, grandparent, grandchild, ancestors, and descendants.

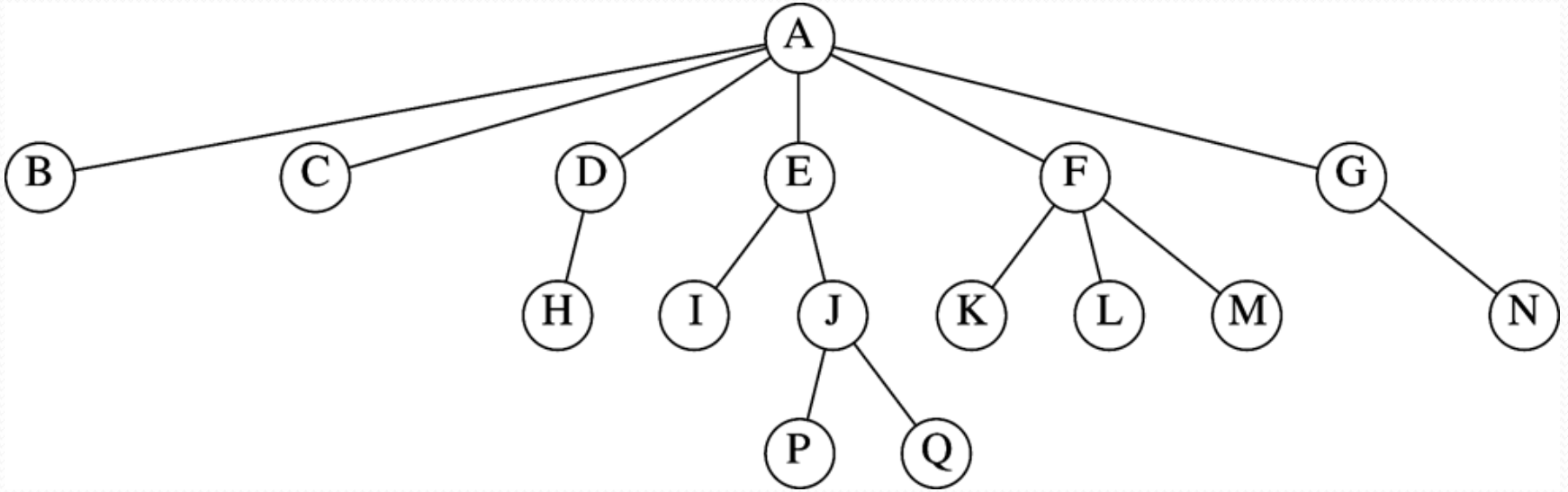
Trees

- For any node n_i
  - The **depth** is the length of the unique path from the root to n_i. The depth of the root is zero.
  - The **height** is the length of the longest path from n_i to a leaf. All leaves are at height zero.
- The height of a tree is the height of the root. The depth of a tree is the depth of the deepest leaf. These two values are the same.

Trees (general)



Tree (example)

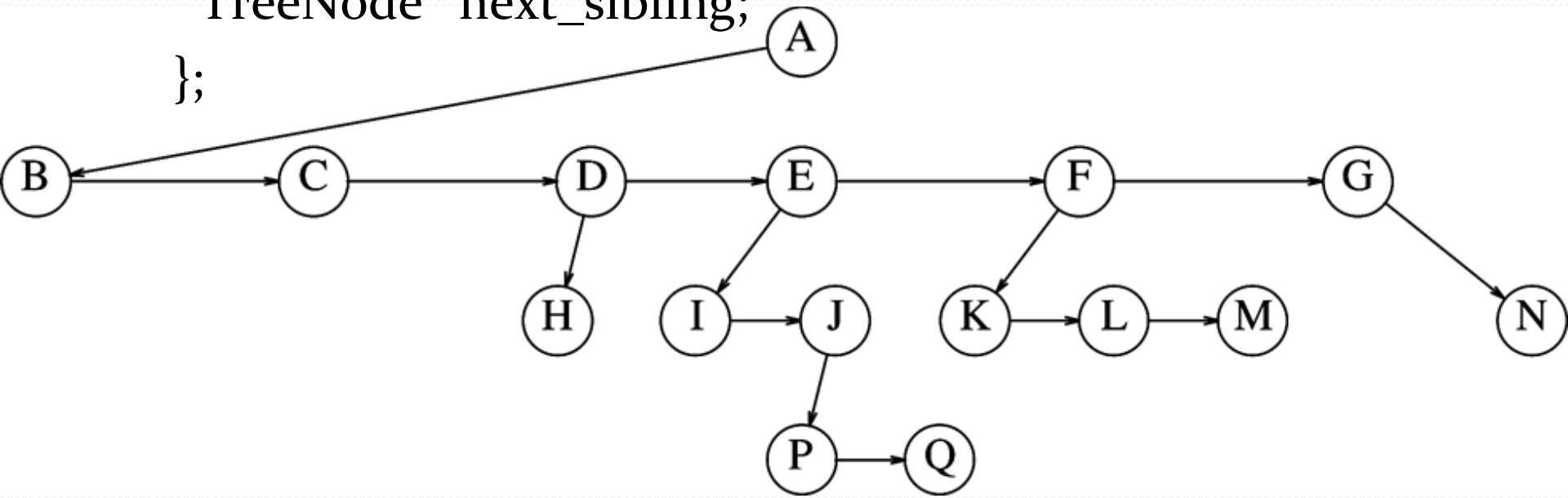


Implementation

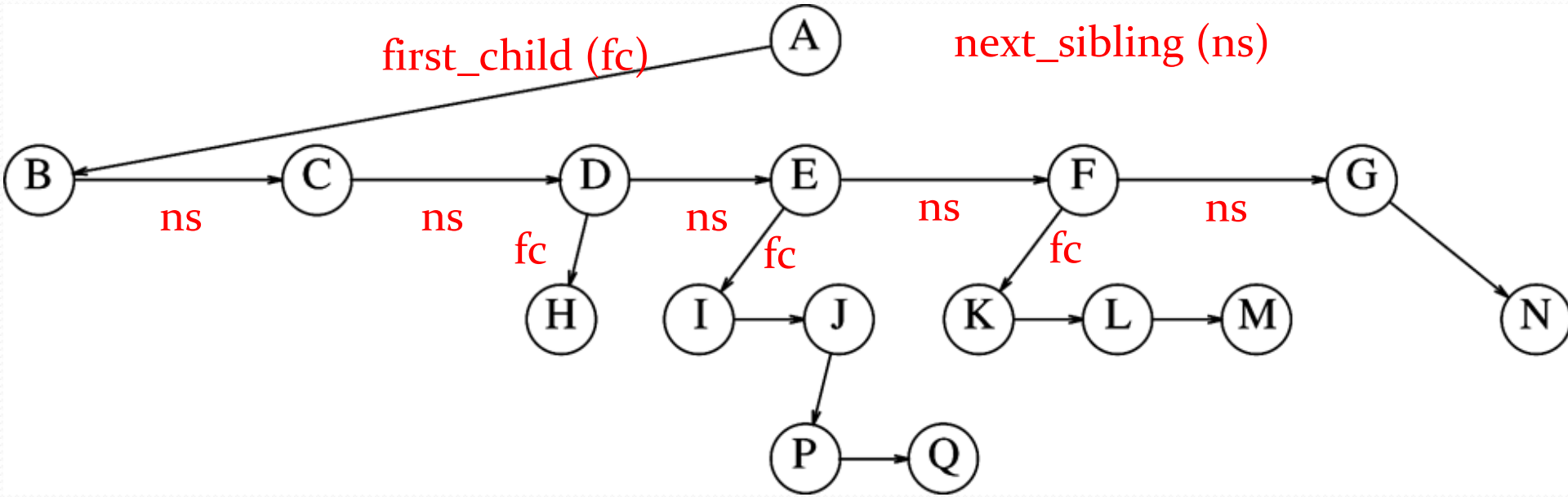
- What if the number of children per parent is not known?

Implementation

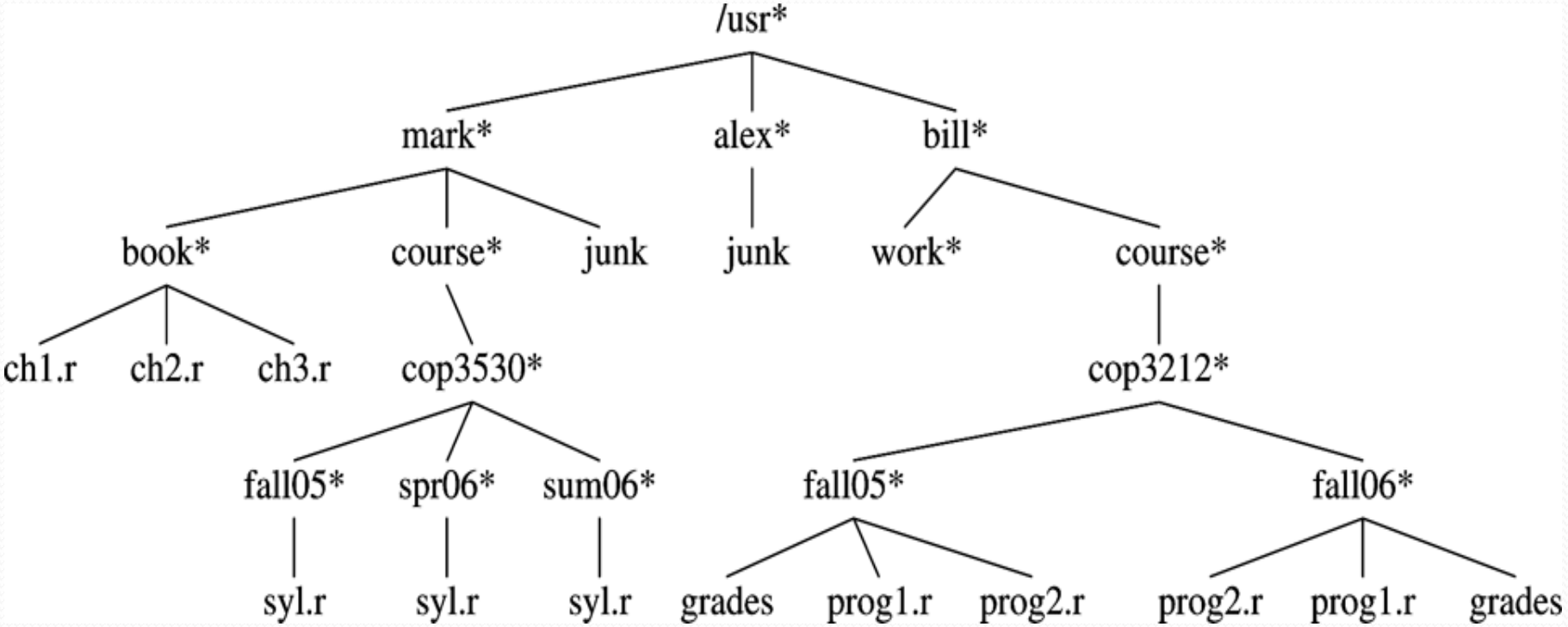
```
template <typename Object>
struct TreeNode {
  Object element;
  TreeNode *first_child;
  TreeNode *next_sibling;
};
```



Implementation



Example (Unix file system) Traversals?



Tree Traversal

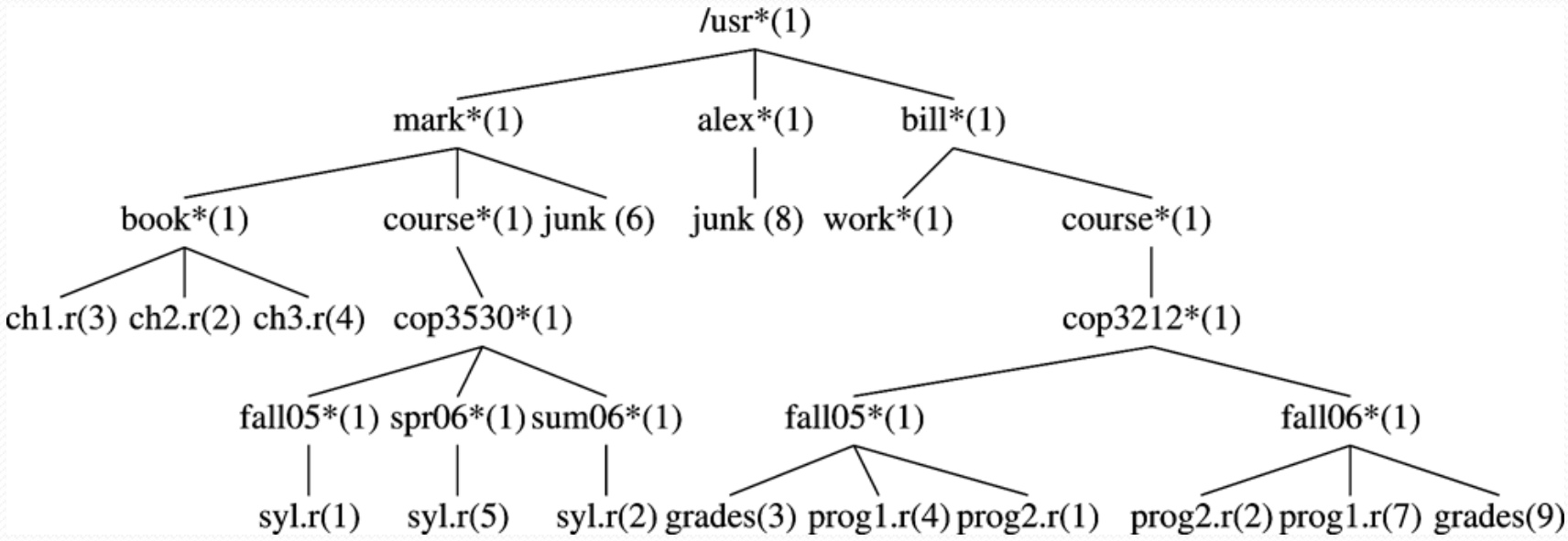
```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall05
                    syl.r
                spr06
                    syl.r
                sum06
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall05
                    grades
                    prog1.r
                    prog2.r
                fall06
                    prog2.r
                    prog1.r
                    grades
```

Tree Traversal

```
    void FileSystem::listAll( int depth = 0 ) const
    {
1       printName( depth );  // Print the name of the object
2       if( isDirectory( ) )
3           for each file c in this directory (for each child)
4               c.listAll( depth + 1 );
    }
```

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall05
                    syl.r
                spr06
                    syl.r
                sum06
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall05
                    grades
                    prog1.r
                    prog2.r
                fall06
                    prog2.r
                    prog1.r
                    grades
```

Calculate Directory Size



Calculate size of directory

```
            ch1.r            3
            ch2.r            2
            ch3.r            4
        book                10
                syl.r        1
            fall05           2
                syl.r        5
            spr06            6
                syl.r        2
            sum06            3
            cop3530         12
        course              13
        junk                 6
    mark                    30
        junk                 8
    alex                     9
        work                 1
                grades       3
                prog1.r      4
                prog2.r      1
            fall05           9
                prog2.r      2
                prog1.r      7
                grades       9
            fall06          19
            cop3212         29
        course              30
    bill                    32
/usr                        72
```

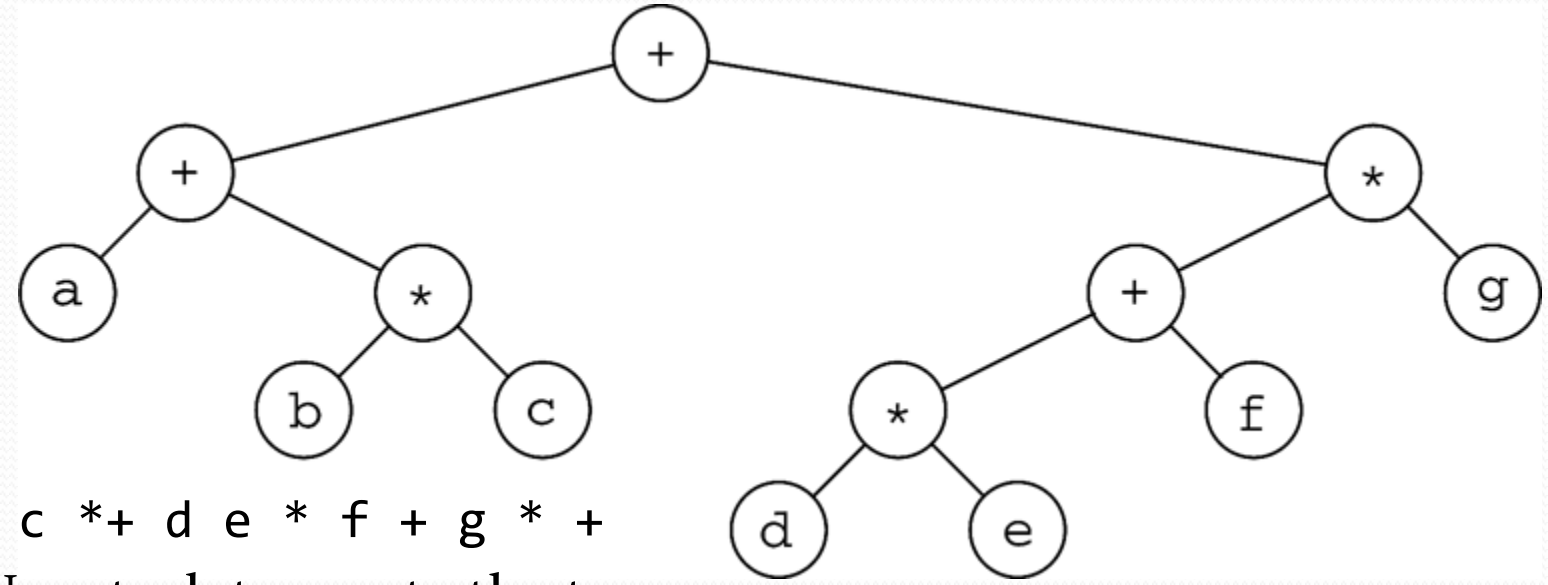
Calculate size of directory

```
int FileSystem::size( ) const
{
    int totalSize = sizeOfThisFile( );

    if( isDirectory( ) )
        for each file c in this directory (for each child)
            totalSize += c.size( );

     return totalSize;
}
```

```
                ch1.r            3
                ch2.r            2
                ch3.r            4
            book                10
                        syl.r    1
                    fall05       2
                        syl.r    5
                    spr06        6
                        syl.r    2
                    sum06        3
                cop3530         12
            course              13
            junk                 6
        mark                    30
            junk                 8
        alex                     9
            work                 1
                        grades   3
                        prog1.r  4
                        prog2.r  1
                    fall05       9
                        prog2.r  2
                        prog1.r  7
                        grades   9
                    fall06      19
                cop3212         29
            course              30
        bill                    32
/usr                            72
```

Expression Trees

- `(a+b*c)+((d*e + f)*g)`
- Algorithm: from postfix expression to expression tree



- `a b c *+ d e * f + g * +`
  - Use stack to create the tree
    - Example `a b + c d e + * *`

Binary search trees

- Implementation header (fig 4.16)
- Public methods for insert, remove, contains (fig 4.17)
- Contains method (fig 4.18)
- Insertion routine (fig 4.23)
- BST using a function object for less (fig 4.19)
- Deletion routine (fig 4.26)

Binary Node

```
//
// Binary Search Tree implementation
// Usage: BinarySearchTree<int> a_tree;
//        a_tree.Insert(10);

template <typename Comparable>
class BinarySearchTree {
 public: // … Big five.
 private:
   struct BinaryNode {
       Comparable element_;
       BinaryNode *left_;
       BinaryNode *right_;
       BinaryNode(const Comparable &the_element, BinaryNode *lt, BinaryNode *rt):
             element_{the_element}, left_{lt}, right_{rt} { }
       BinaryNode(Comparable &&the_element, BinaryNode *lt, BinaryNode *rt):
             element_{std::move(the_element)}, left_{lt}, right_{rt} { }
    };
    BinaryNode *root_;
    // …
};
```

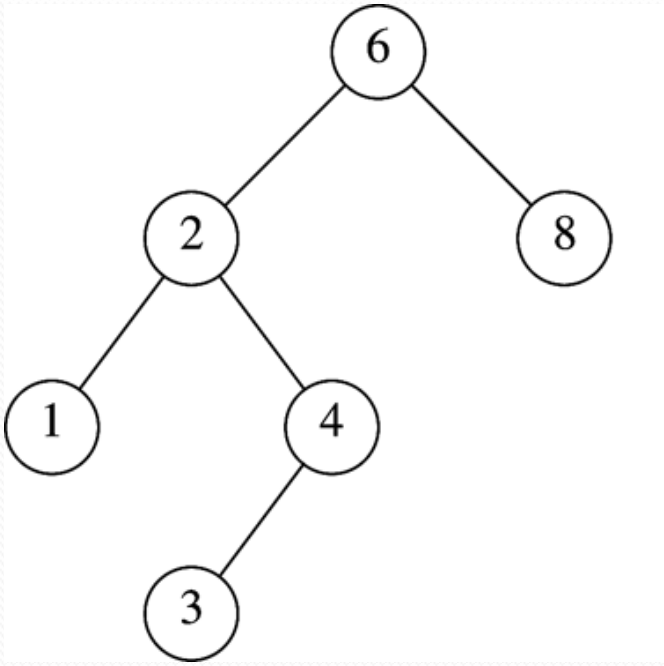
Regular Insert

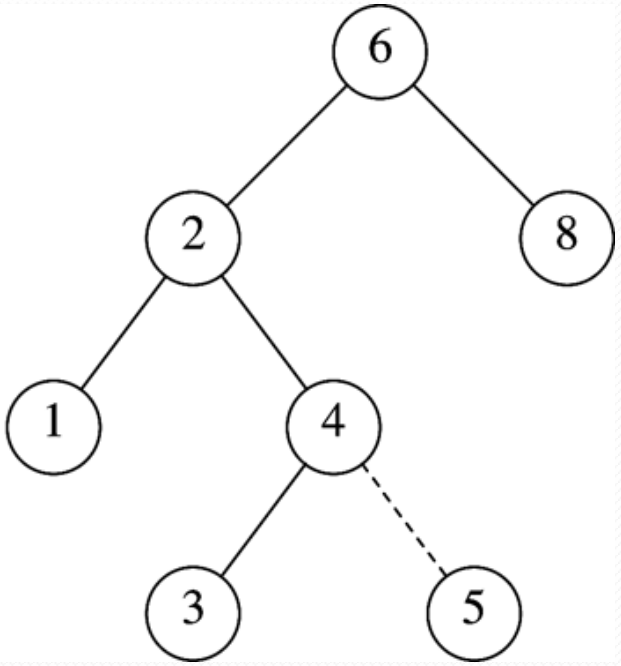
```
// Internal method to insert into a subtree.
// @x is the item to insert.
// @t is the pointer to the node that roots the subtree.
// x  is inserted in the subtree, and t is updated.
// No insertion if x is already in the tree.
void Insert(const Comparable &x, BinaryNode * &t) {
     if (t == nullptr)
         t = new BinaryNode{x, nullptr, nullptr};
     else if (x < t->element_)
         Insert(x, t->left_);
     else if (t->element_ < x)
         Insert(x, t->right_);
     else
                  ;  // Duplicate; do nothing
}
```

Move Insert

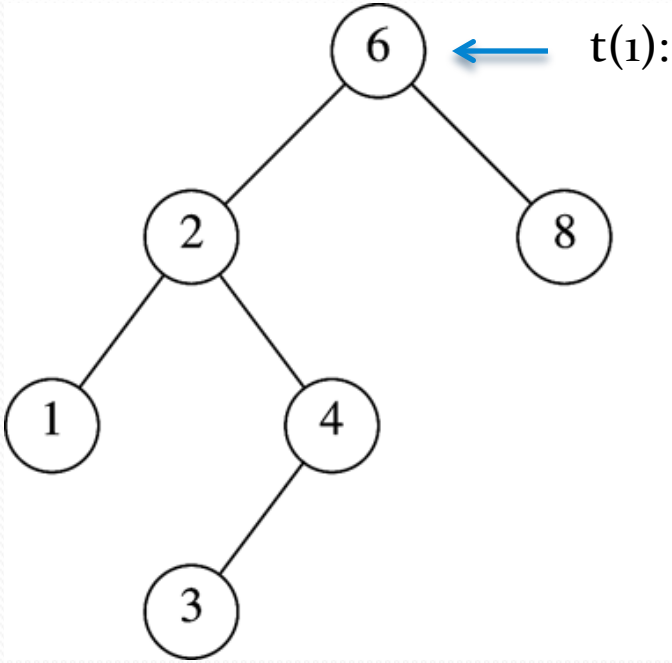
```
// Internal method to insert into a subtree.
// @x is the item to insert.
// @t is the pointer to the node that roots the subtree.
// x  is inserted in the subtree, and t is updated.
// No insertion if x is already in the tree.
void Insert(const Comparable &&x, BinaryNode * &t) {
   if (t == nullptr)
      t = new BinaryNode{std::move(x), nullptr, nullptr};
   else if (x < t->element_)
      Insert(std::move(x), t->left_);
   else if (t->element_ < x)
      Insert(std::move(x), t->right_);
   else
               ;  // Duplicate; do nothing
}
```

Inserting 5

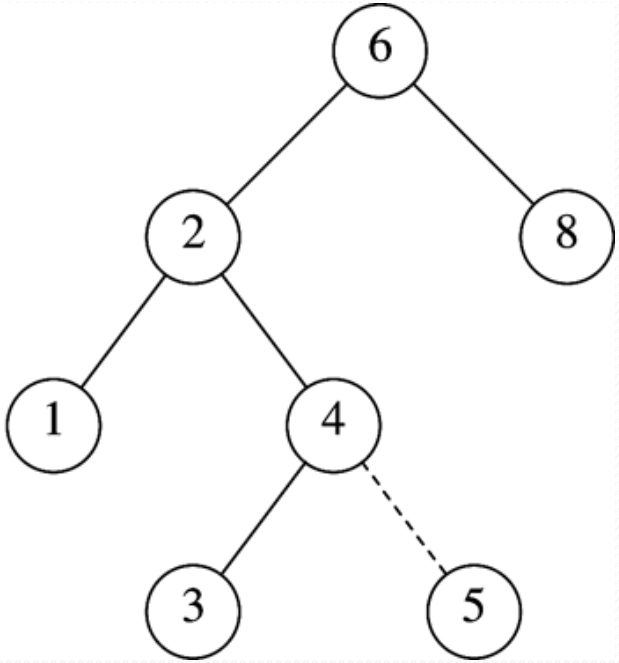




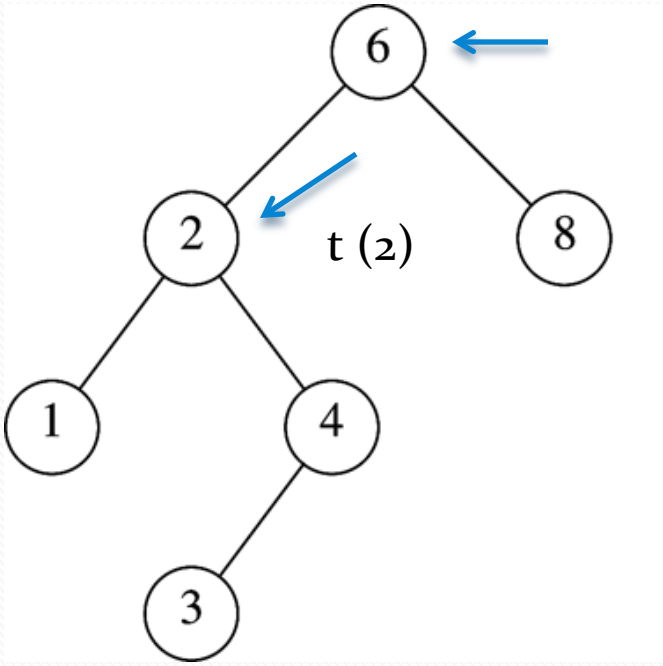
Inserting 5

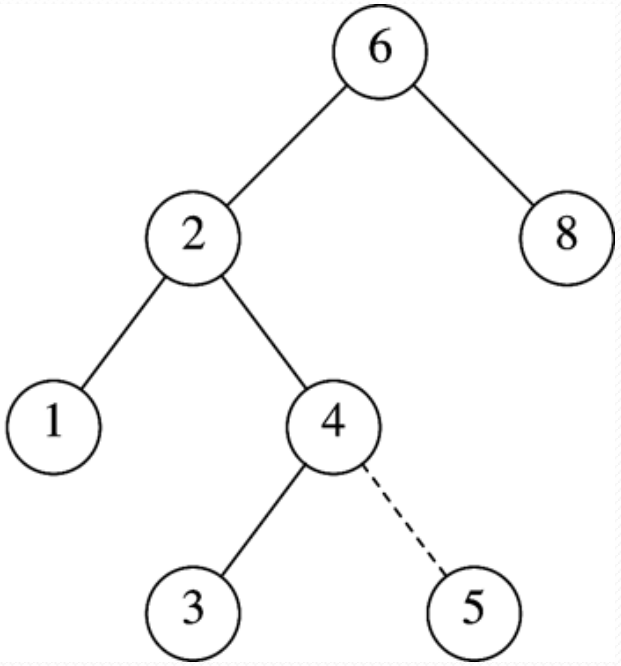


t(1): START

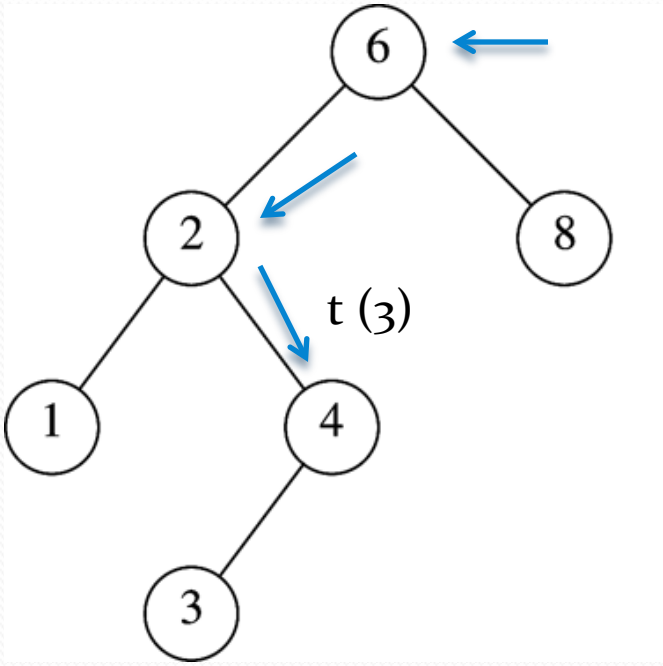


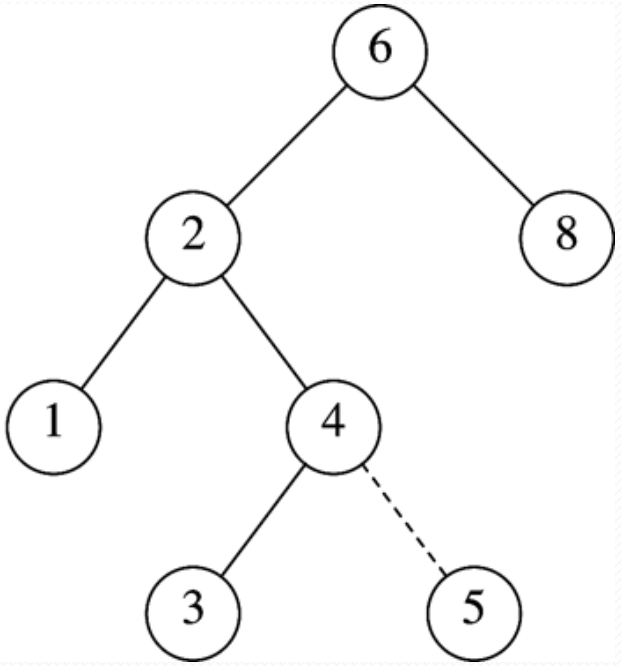
Inserting 5



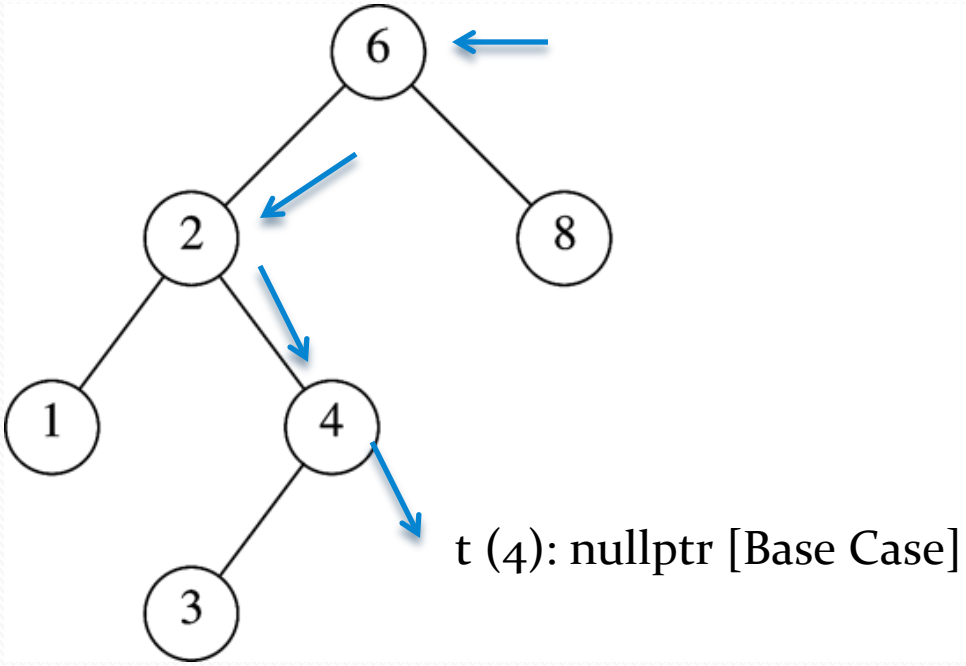


Inserting 5

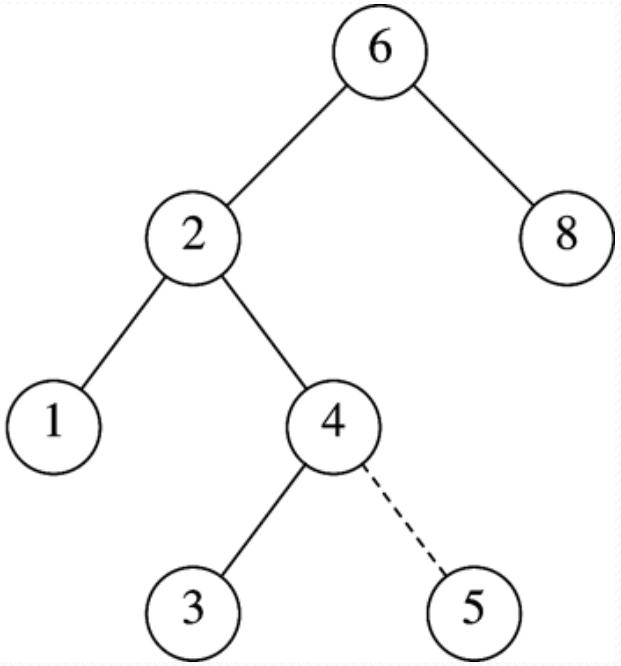




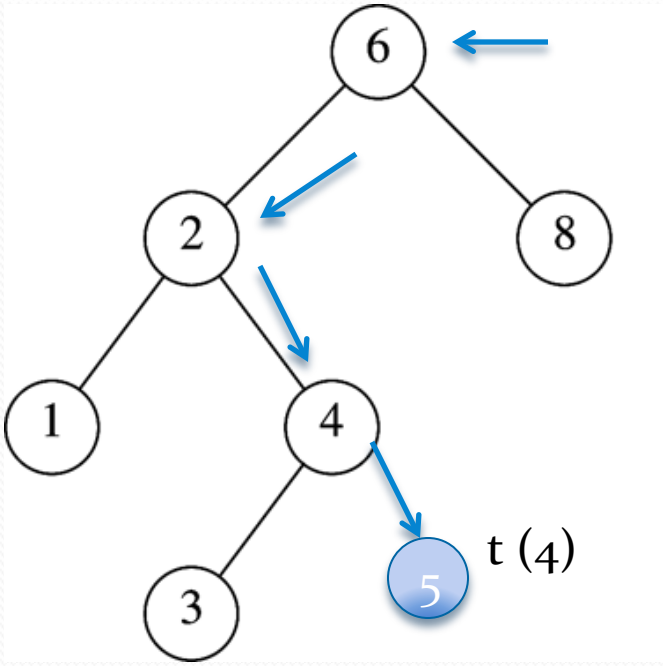
Inserting 5

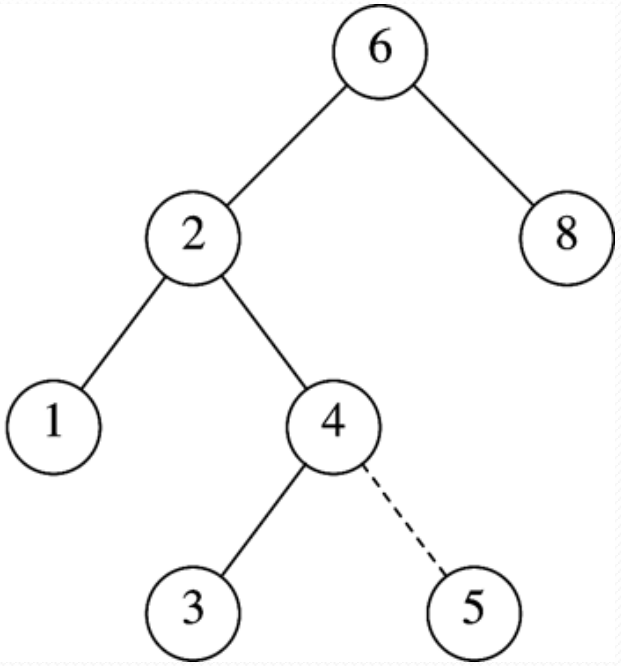


t (4): nullptr [Base Case]

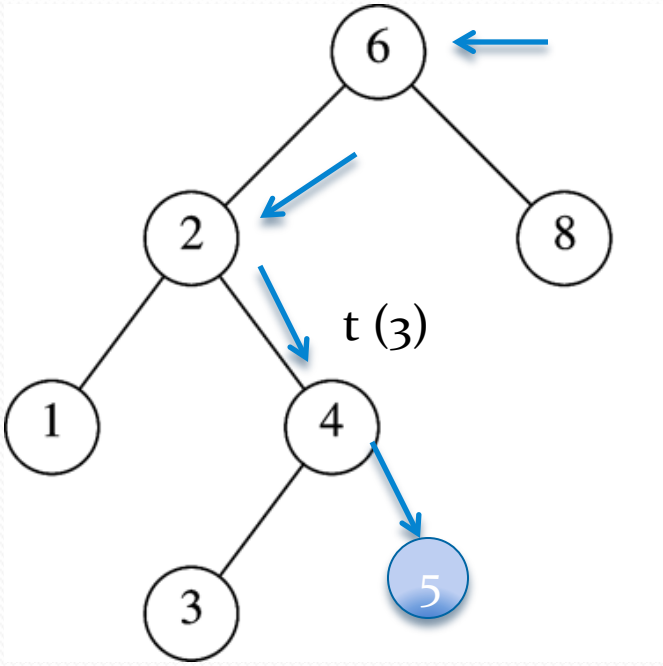


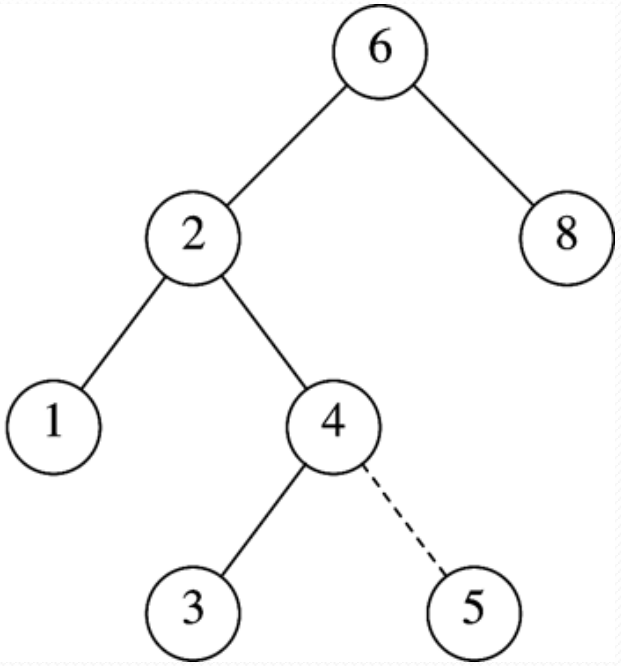
Inserting 5



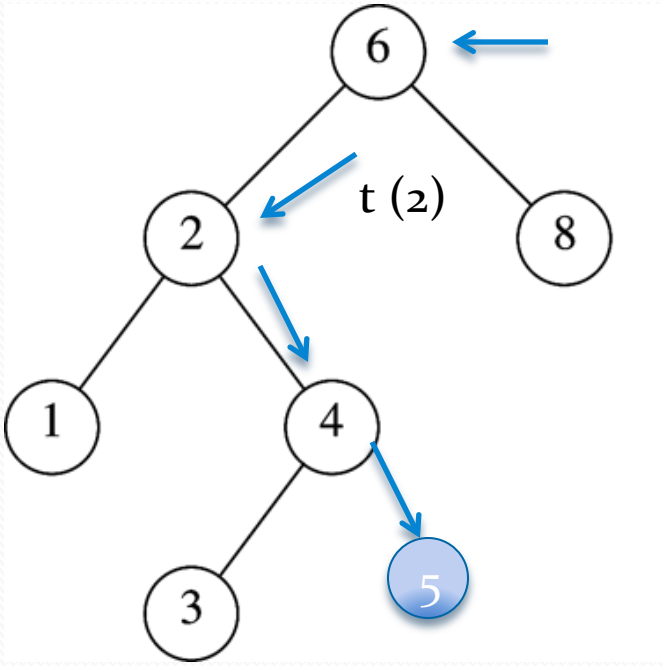


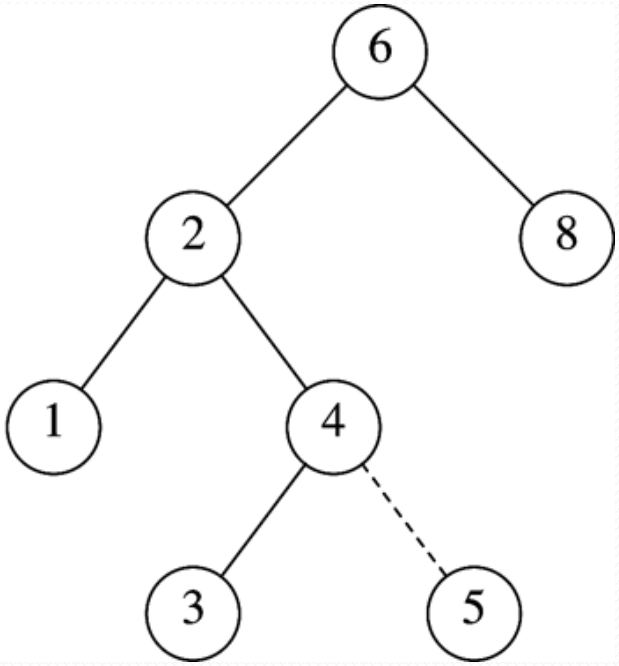
Inserting 5



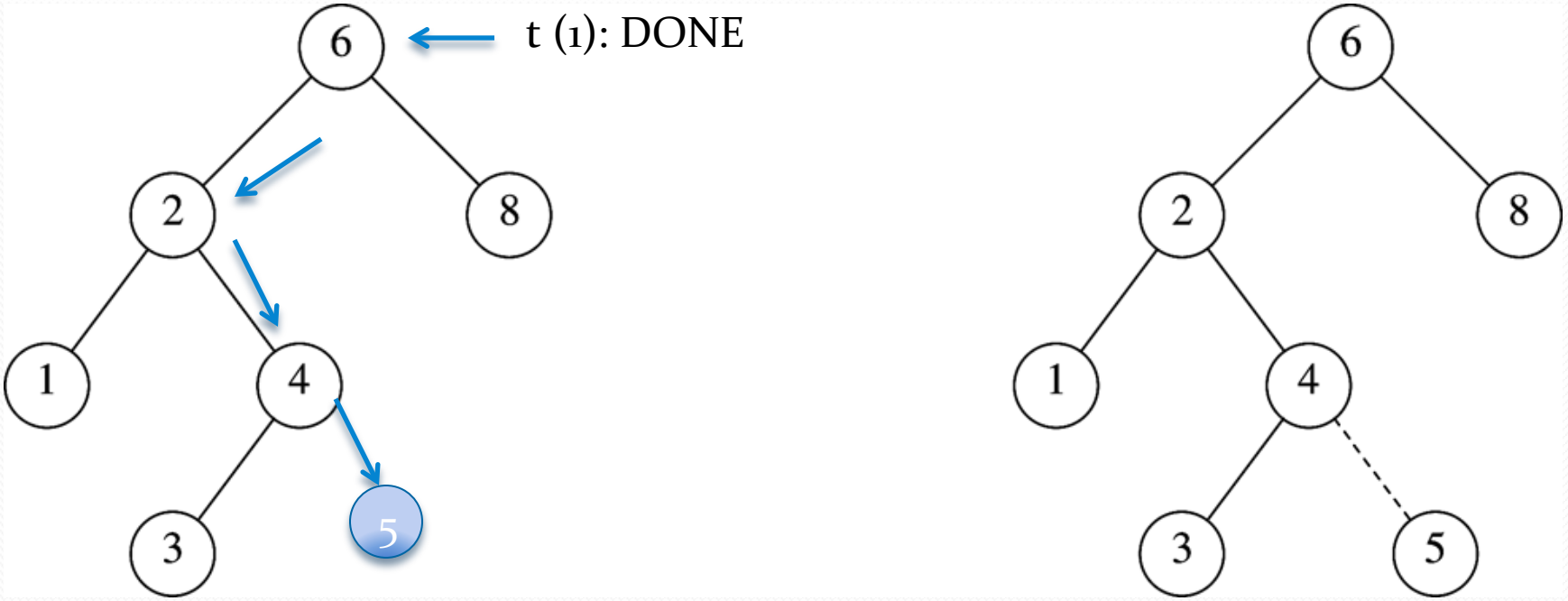


Inserting 5





Inserting 5



Remove

```
// Internal method to remove from a subtree.
// @x is the item to remove.
// @t is the pointer to the node that roots the subtree.
void Remove(const Comparable &x, BinaryNode * &t) {
      if (t == nullptr)
          return;   // Item not found; do nothing
      if (x < t->element_) {
          Remove(x, t->left_);
      } else if (t->element_ < x) {
          Remove(x, t->right_);
      } else if (t->left_ != nullptr && t->right_ != nullptr) {  // 2 children.
          t->element_ = FindMin(t->right_)->element_;
          Remove(t->element_, t->right_);
      } else {// One or no children.
          BinaryNode *old_node = t;
          t = (t->left != nullptr) ? t->left : t->right;
          delete old_node;
      }
}  // End remove
```

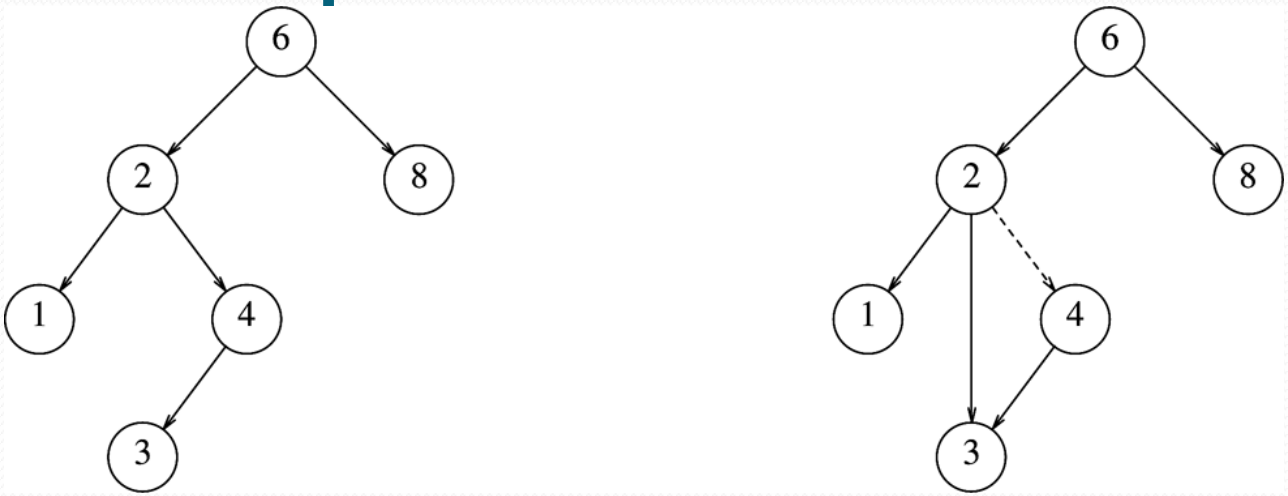
Remove

```
//
// @t: a given node of the tree.
// Find the node of minimum value under t.
// @return the node of minimum value.
//  if t is nullptr returns nullptr.
BinaryNode *FindMin(BinaryNode *t) {

}  // End remove
```

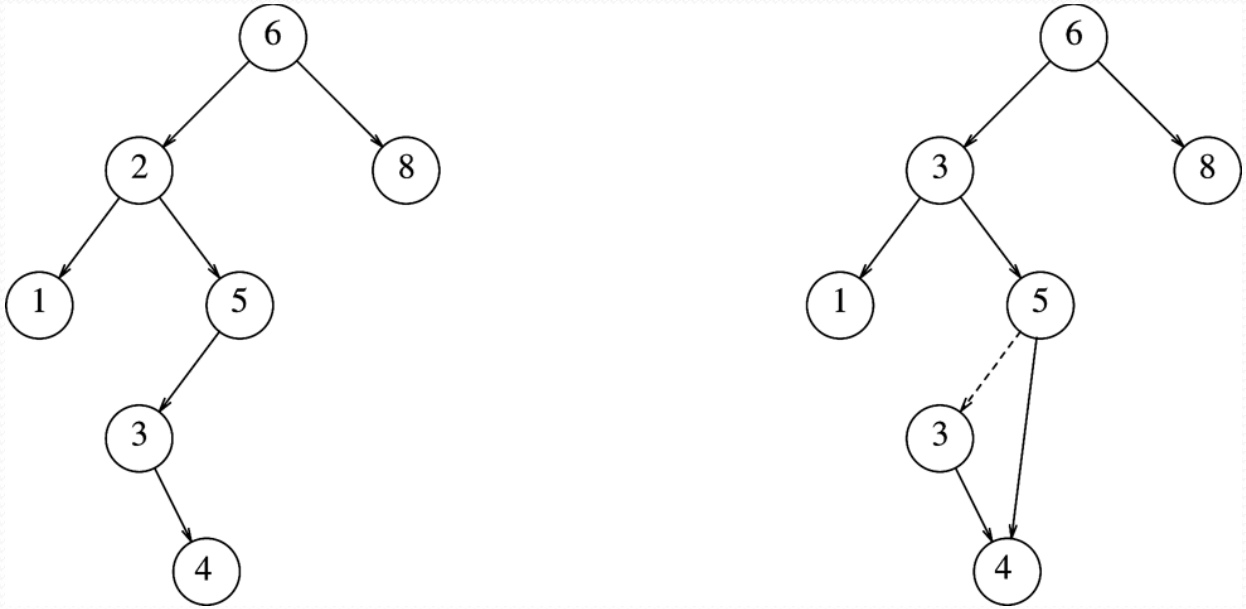
Remove

```
//
// @t: a given node of the tree.
// Find the node of minimum value under t.
// @return the node of minimum value.
//  if t is nullptr returns nullptr.
// Recursive implementation.

BinaryNode *FindMin(BinaryNode *t) const {
  return t == nullptr ? nullptr: (t->left == nullptr ? t: FindMin(t->left));
}  // End remove
```

Examples of Deletion

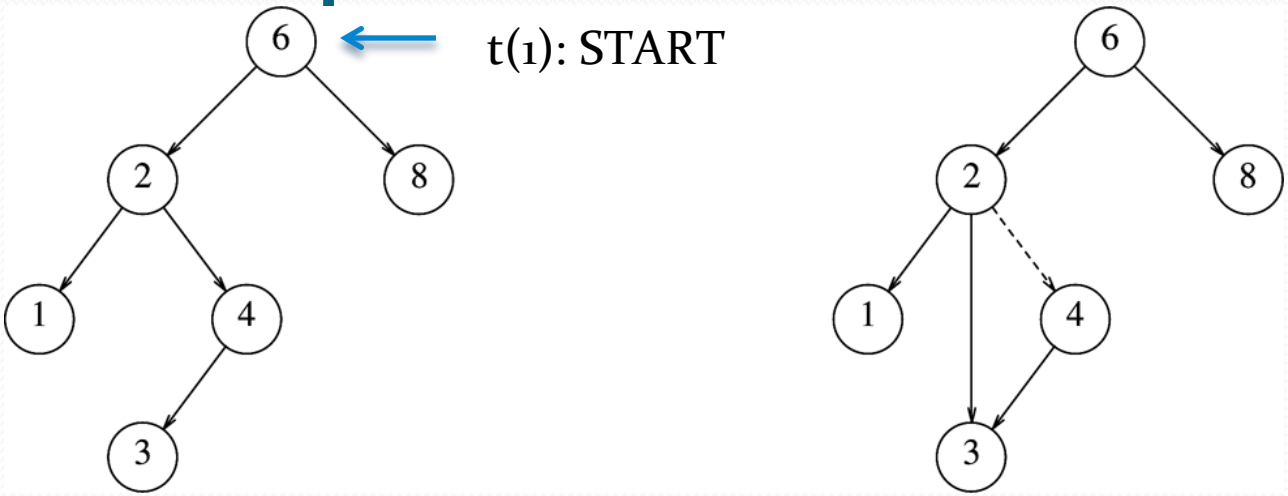


Remove 4



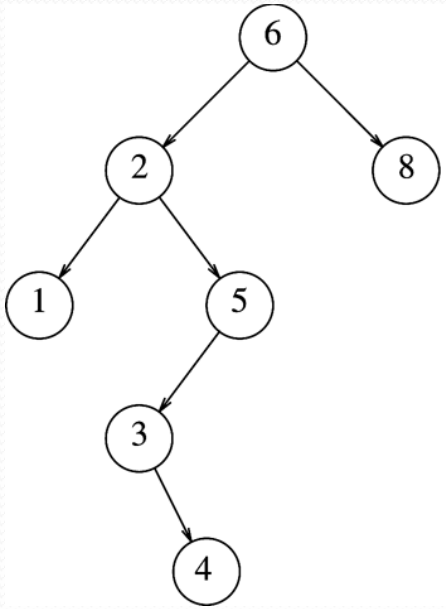
Remove 2

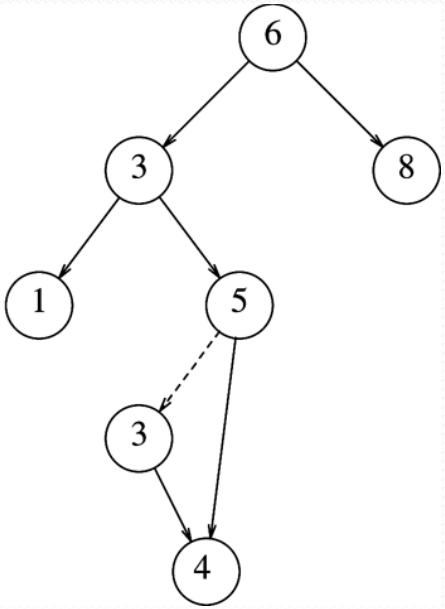
Examples of Deletion



t(1): START

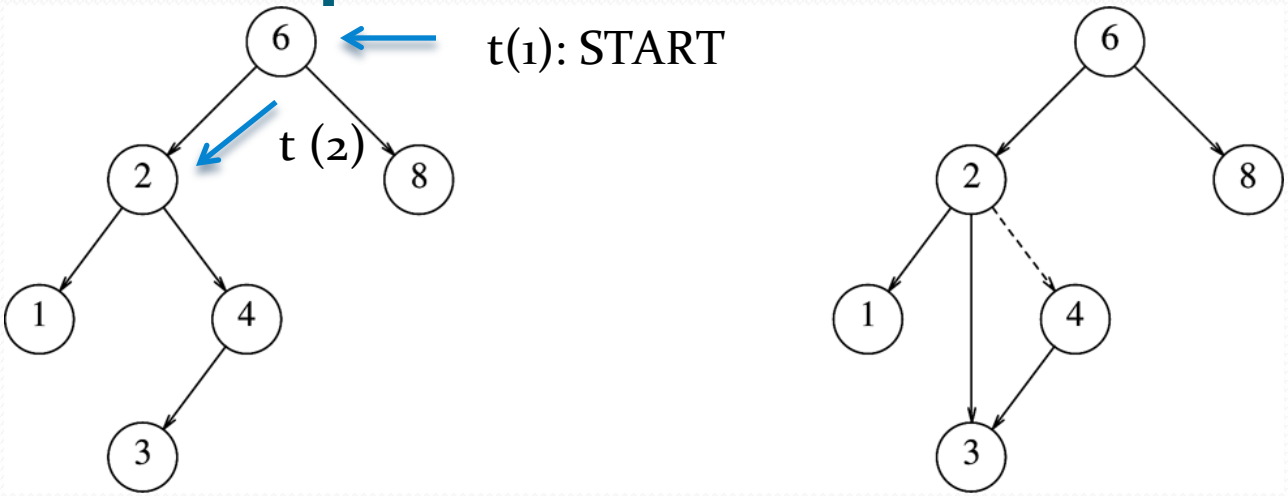
Remove 4



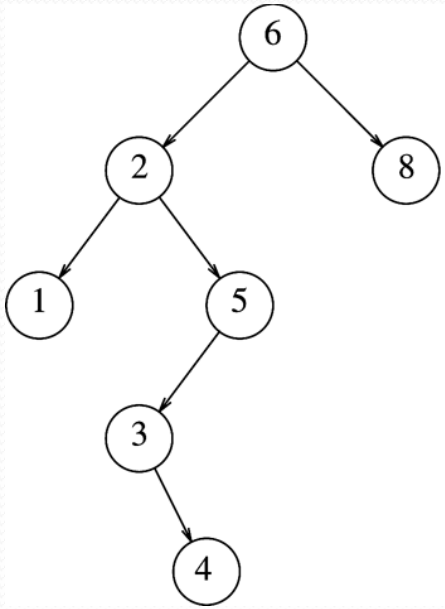


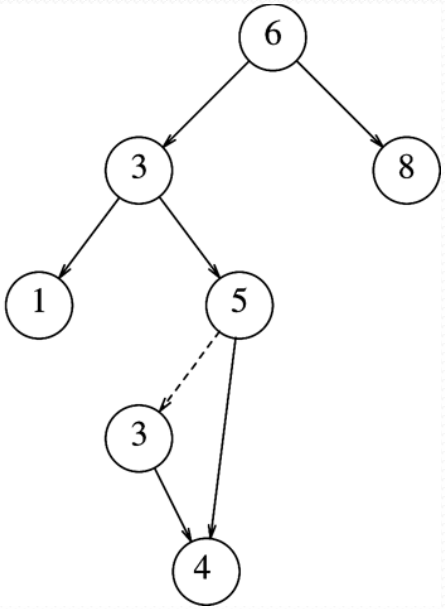
Remove 2

Examples of Deletion



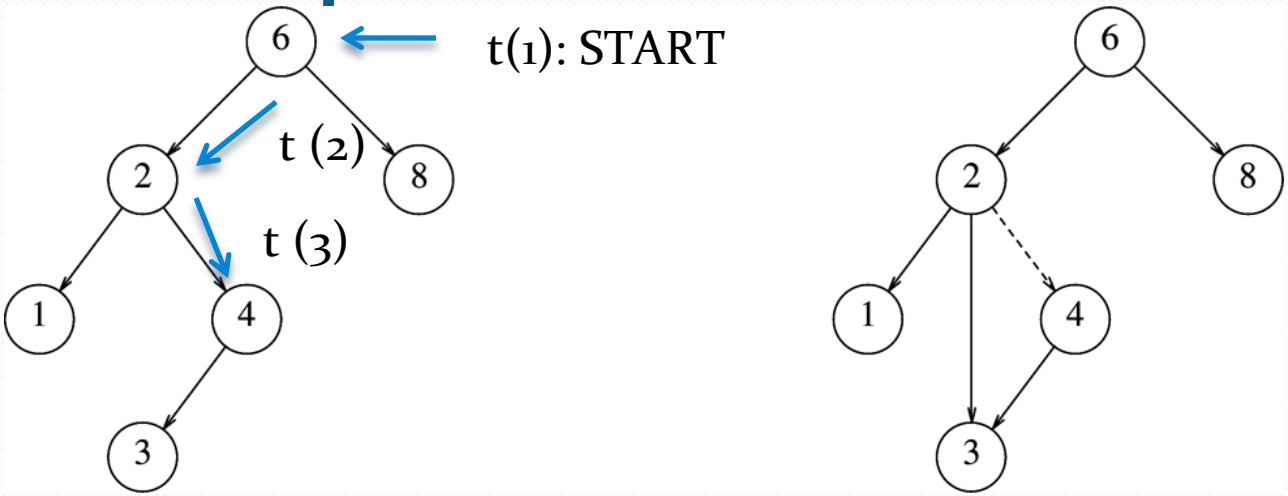
Remove 4



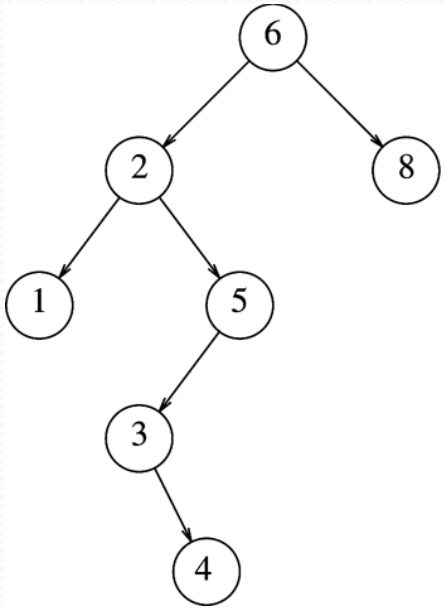


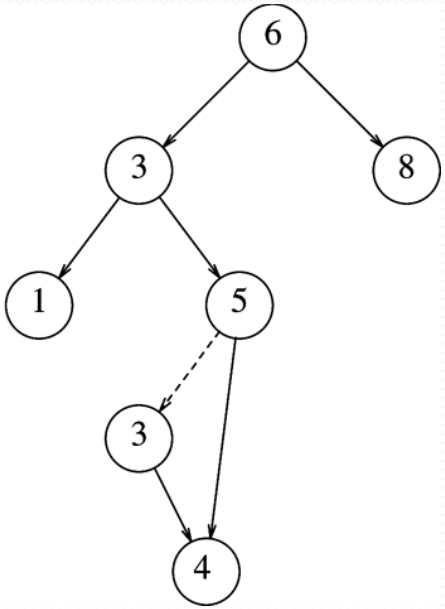
Remove 2

Examples of Deletion



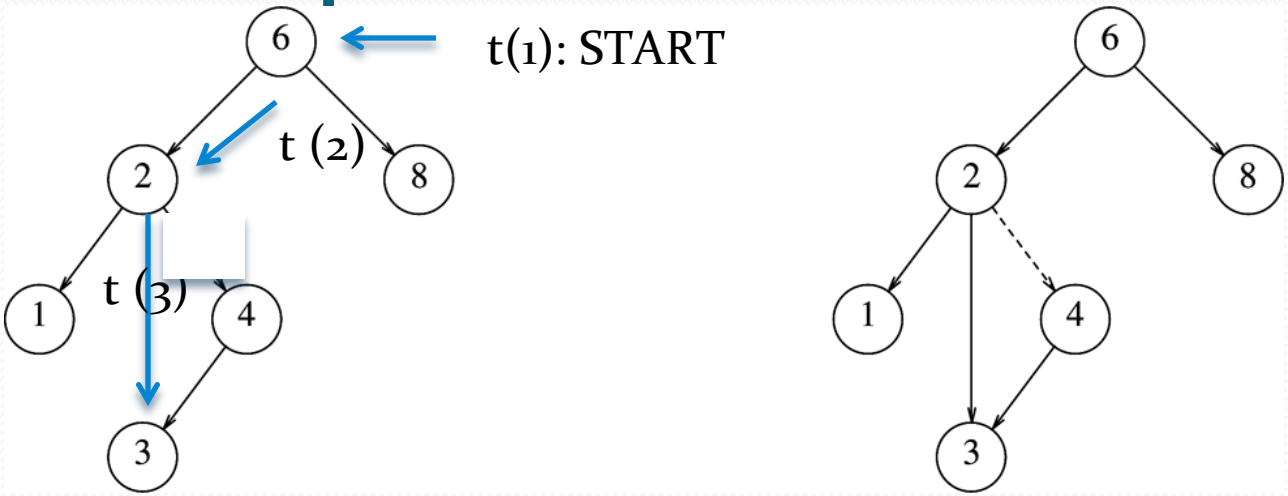
Remove 4



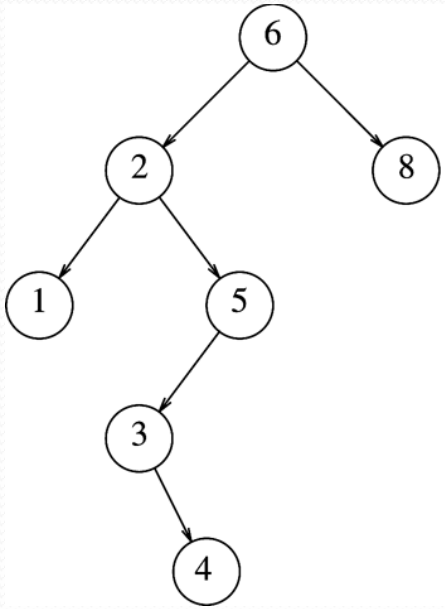


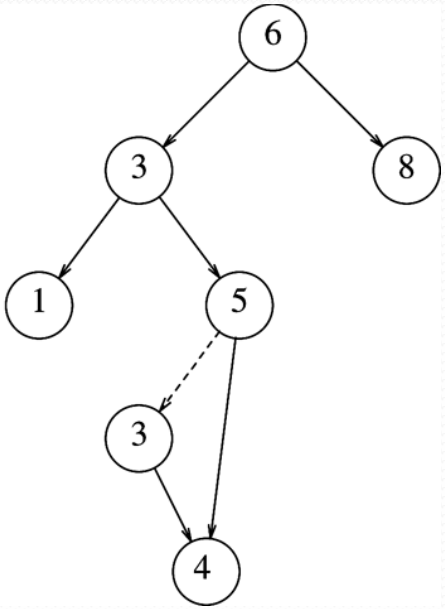
Remove 2

Examples of Deletion



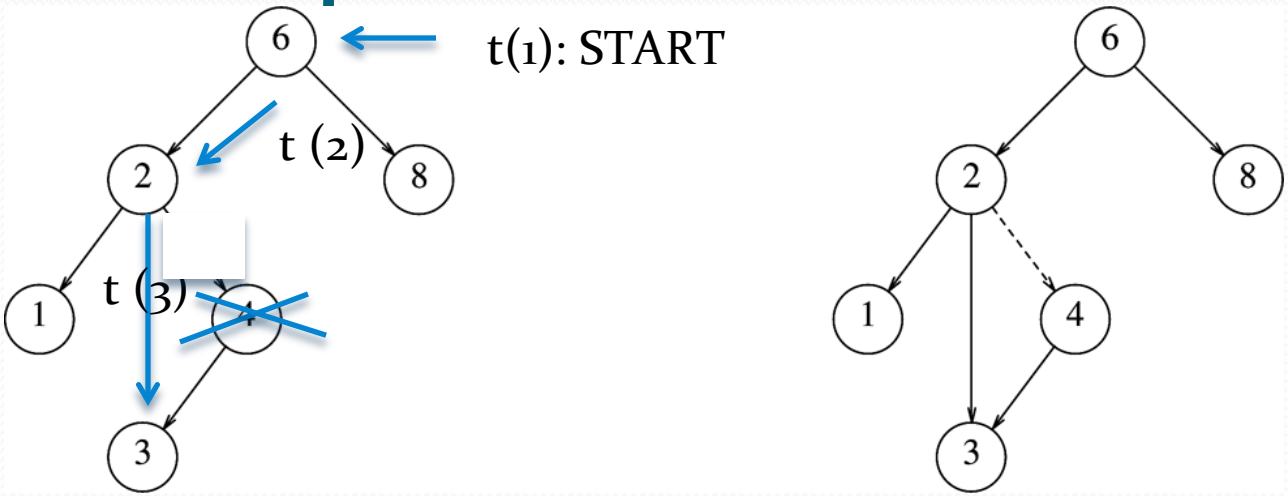
Remove 4



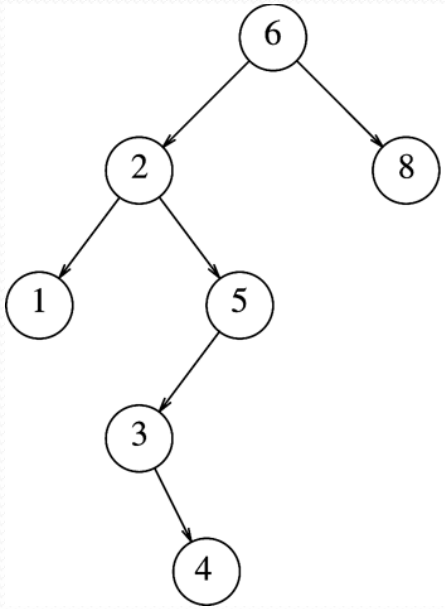


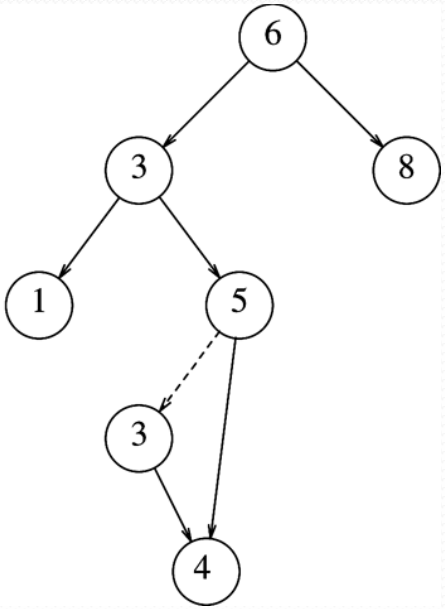
Remove 2

Examples of Deletion



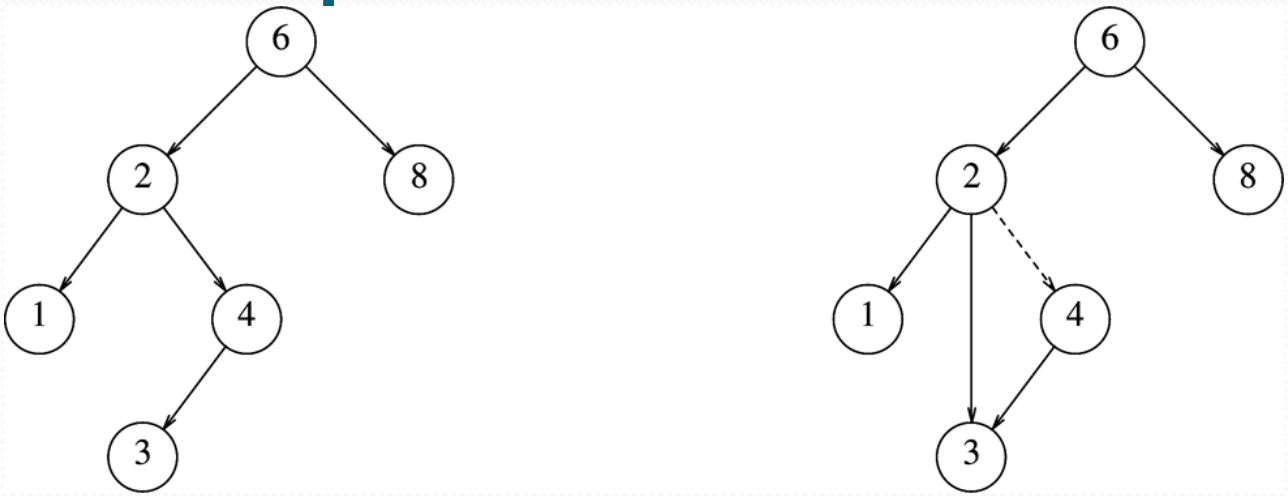
Remove 4



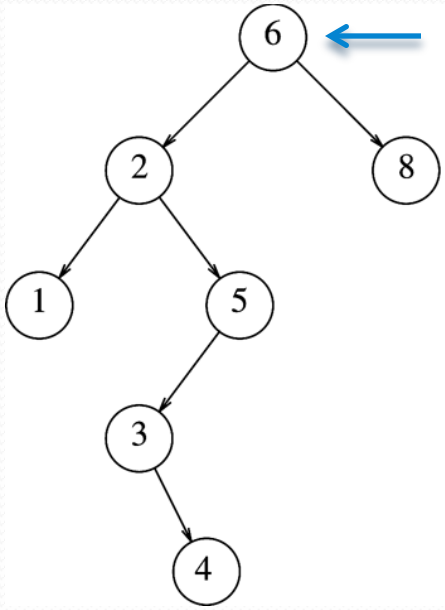


Remove 2

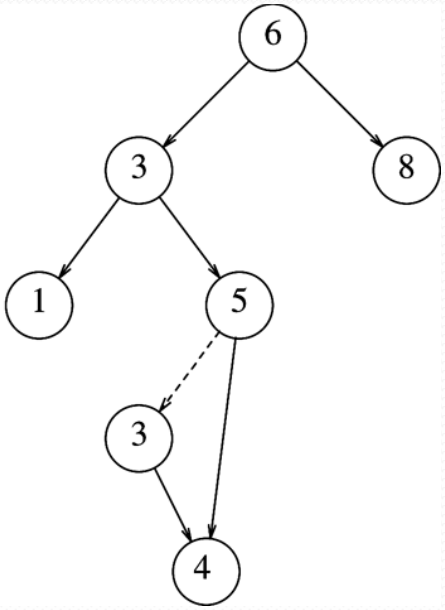
Examples of Deletion



Remove 4

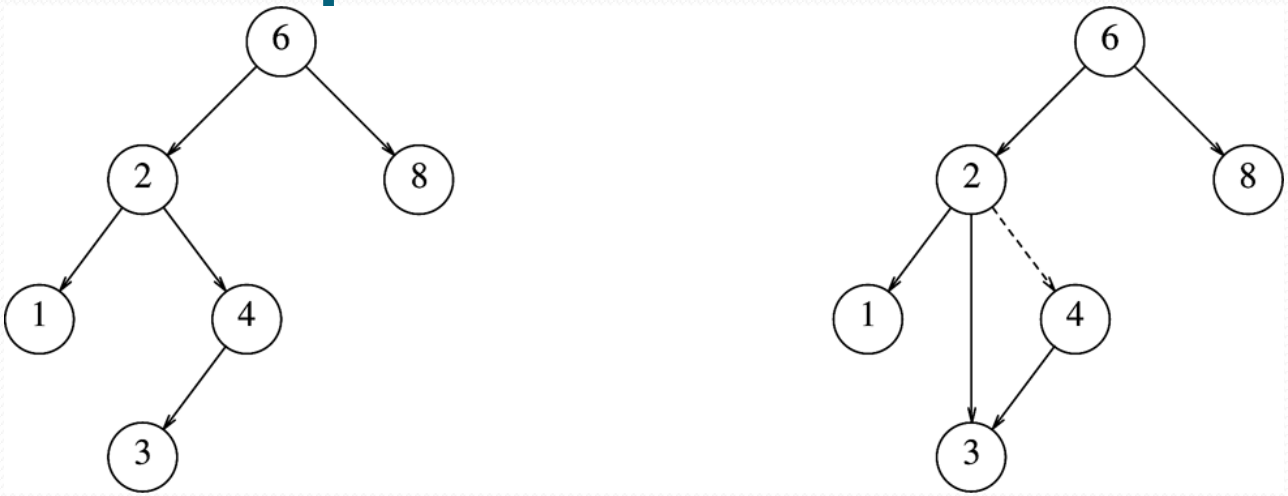


2, t(1): START

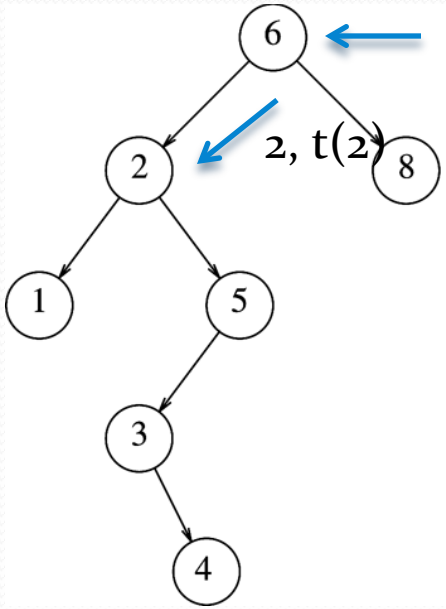


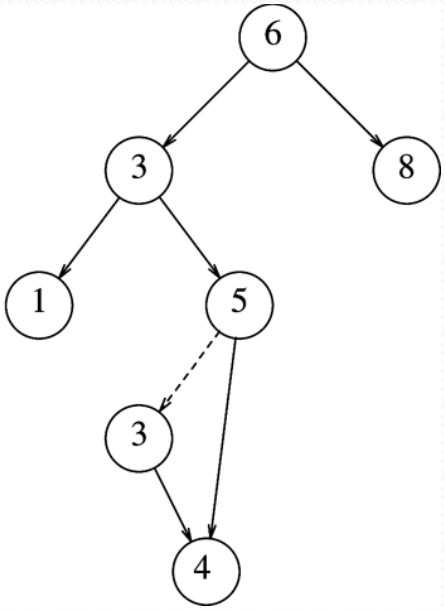
Remove 2

Examples of Deletion



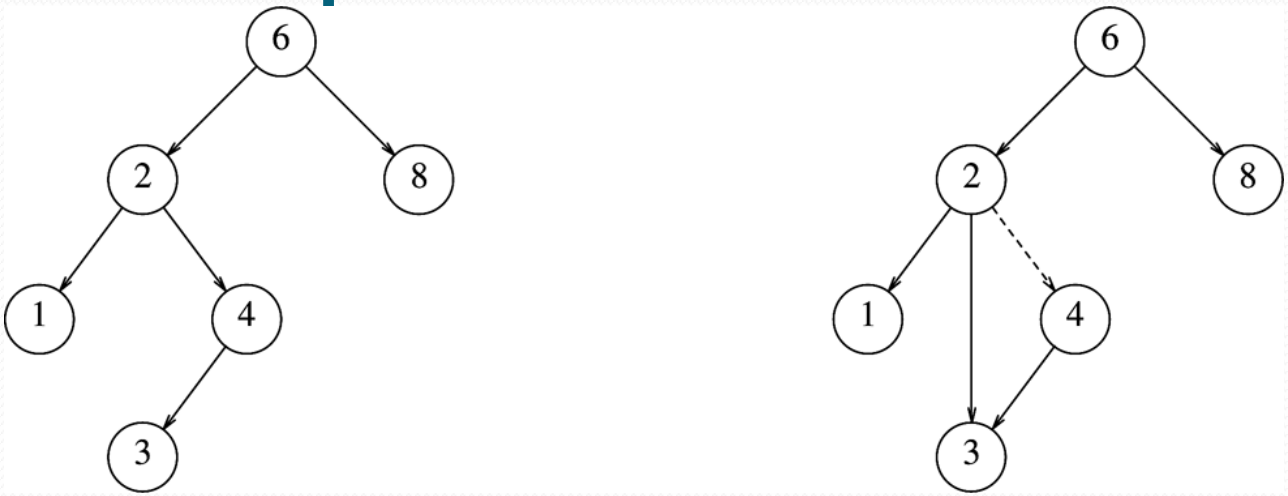
Remove 4



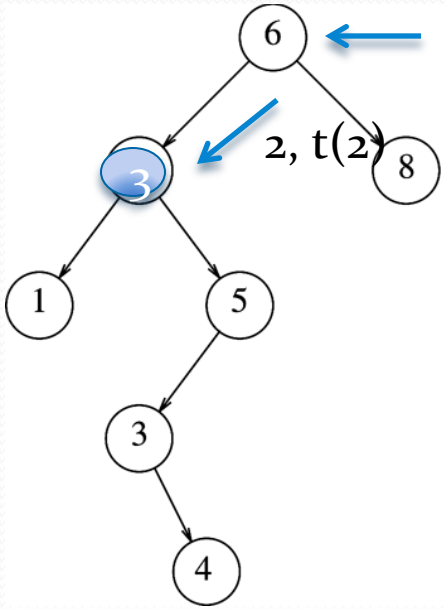


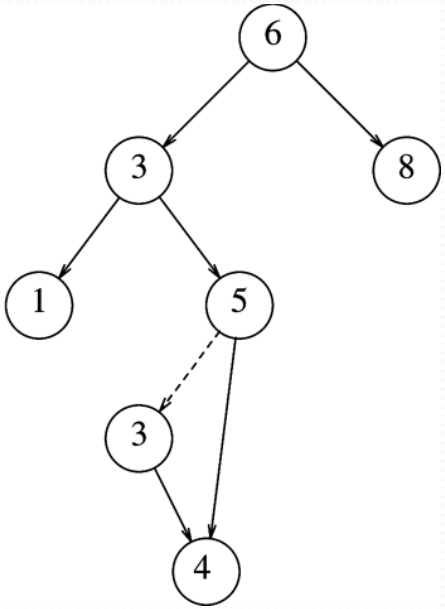
Remove 2

Examples of Deletion



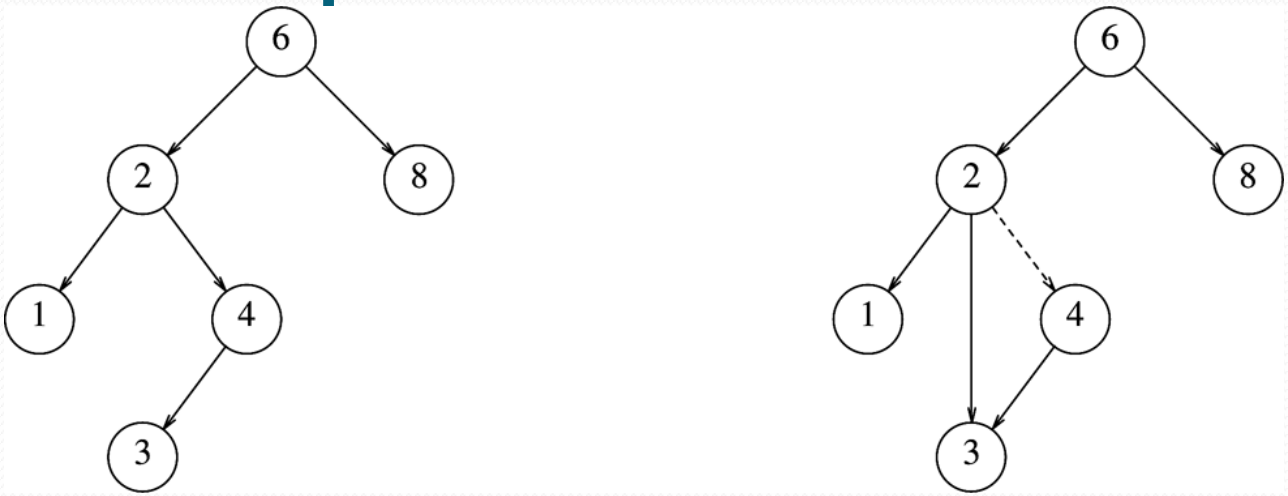
Remove 4



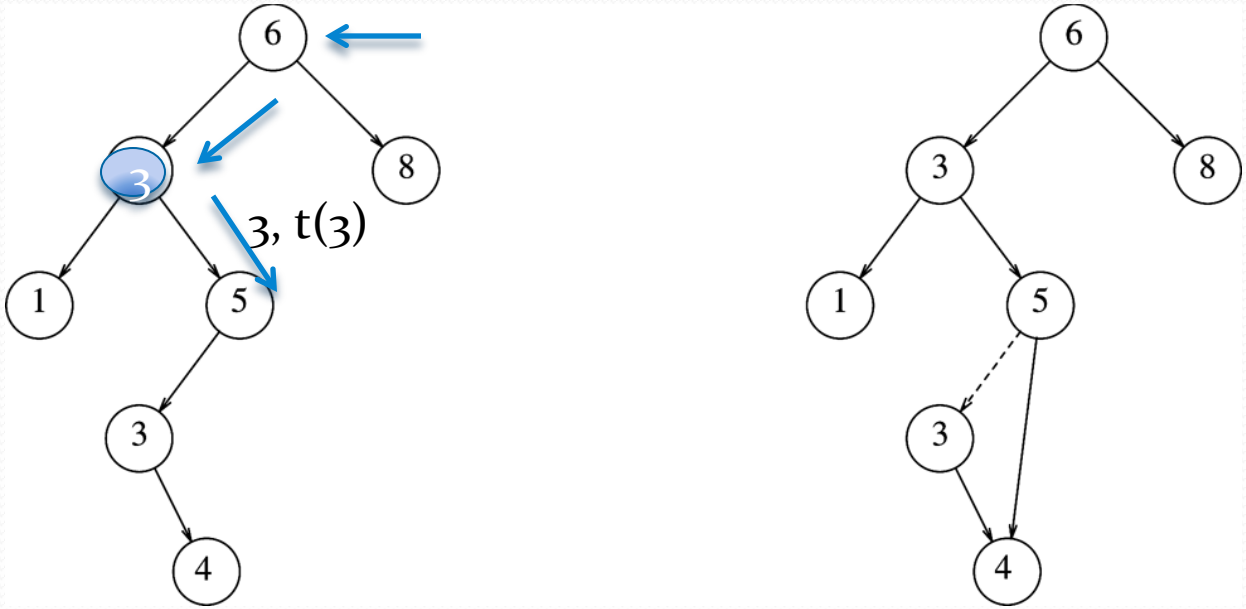


Remove 2

Examples of Deletion

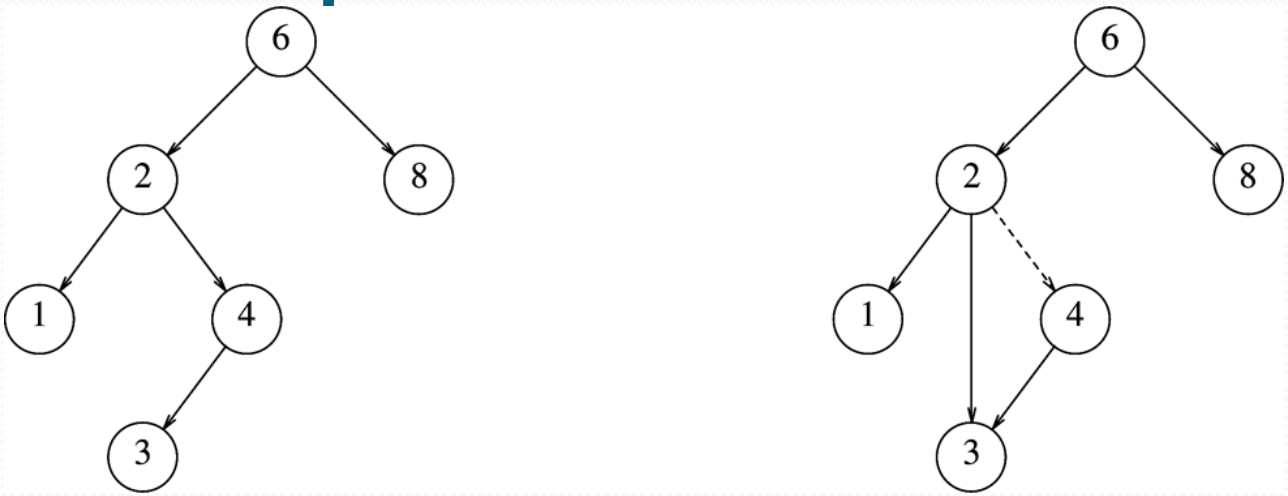


Remove 4

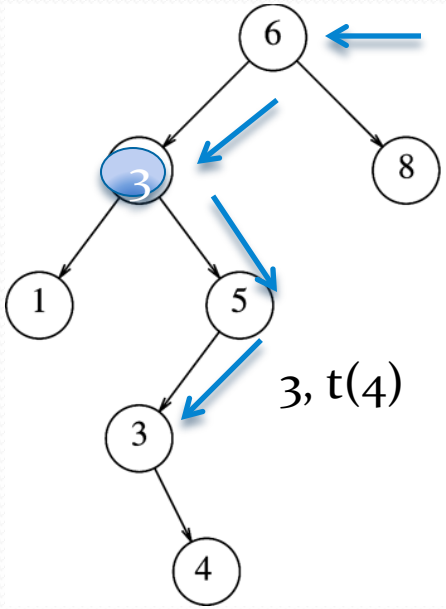


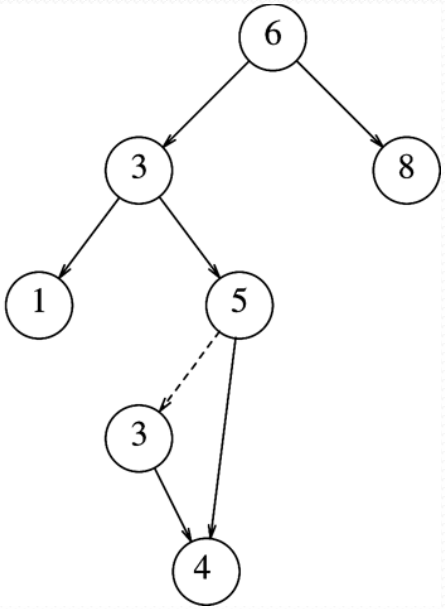
Remove 2

Examples of Deletion



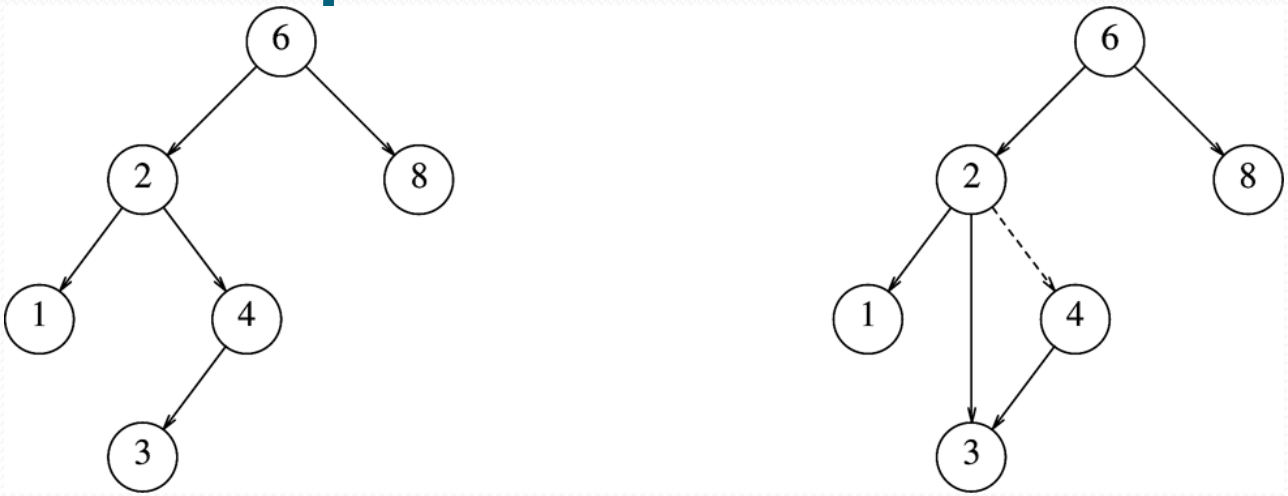
Remove 4



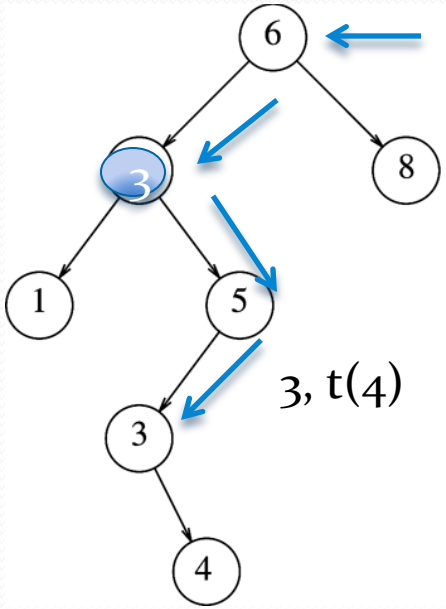


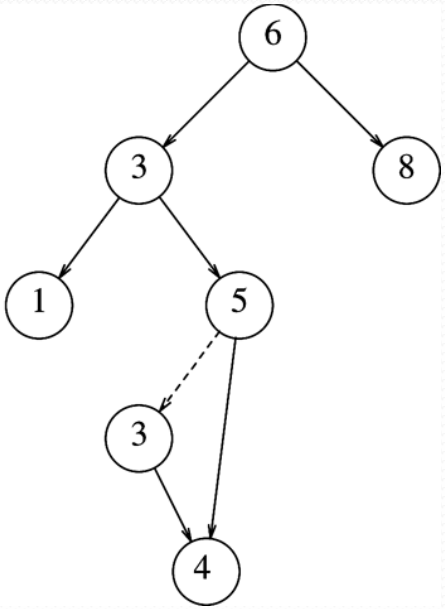
Remove 2

Examples of Deletion



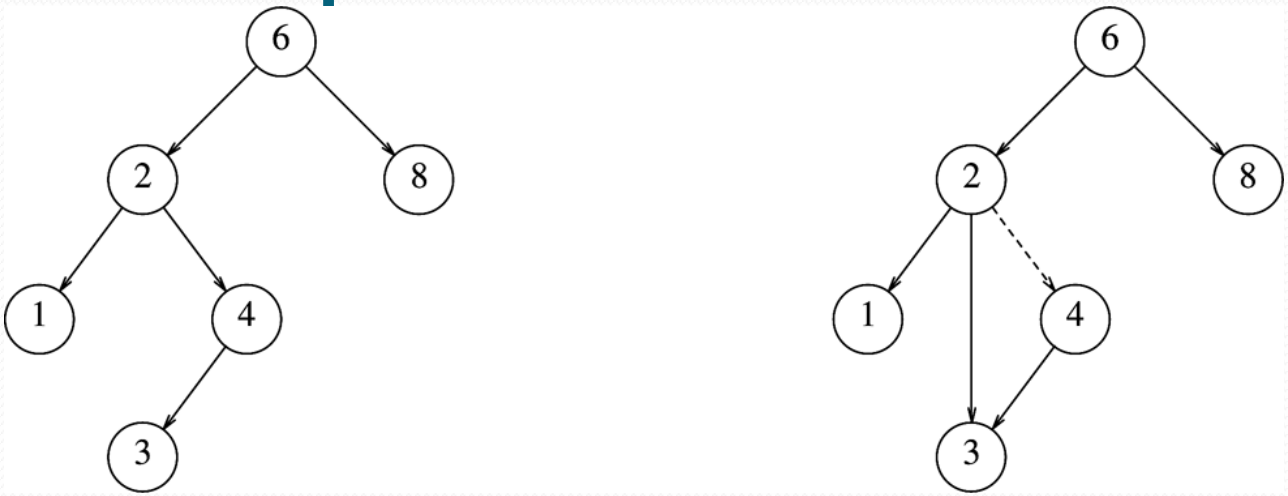
Remove 4



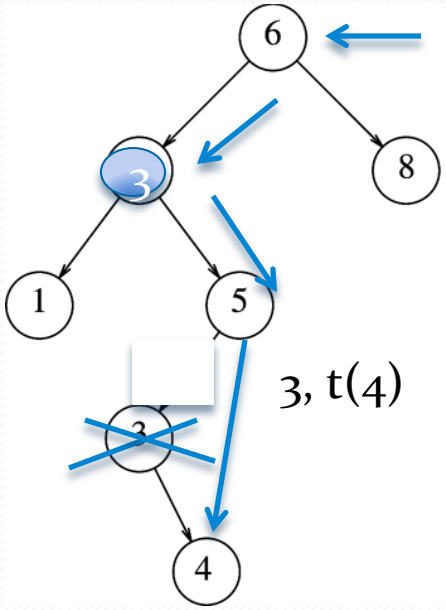


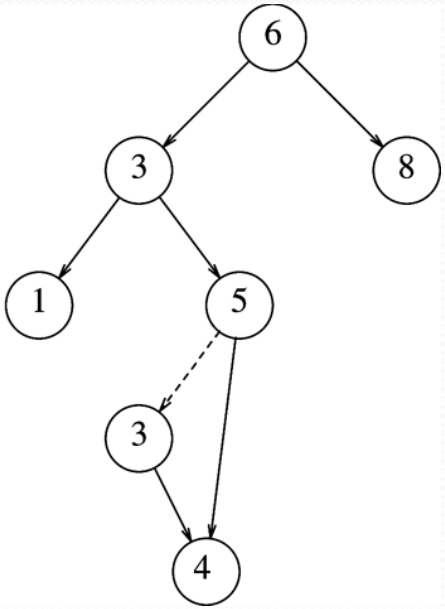
Remove 2

Examples of Deletion



Remove 4





Remove 2

Big Five

```
BinarySearchTree() : root_{nullptr} { } // 0-parameter constructor.


// Copy Constructor.
BinarySearchTree(const BinarySearchTree & rhs) : root_{nullptr} {
     root_ = Clone(rhs.root_);
}

// Move Constructor.
BinarySearchTree(BinarySearchTree &&rhs) : root_{rhs.root_} {
        rhs.root_ = nullptr;
}

~BinarySearchTree() {  // Destructor.
      MakeEmpty();
}
```

Big Five (cont..)

```
// Copy Assignment.
BinarySearchTree &operator=(const BinarySearchTree & rhs) {
      BinarySearchTree copy = rhs;
      std::swap(*this, copy);
      return *this;
}

// Move Assignment.
BinarySearchTree &operator=(BinarySearchTree &&rhs) {
       std::swap(root_, rhs.root_);
       return *this;
}
```

Big Five

```
// Deallocates memory of subtree with root t.
void MakeEmpty(BinaryNode *&t) {
  if (t == nullptr) return;
  MakeEmpty(t->left_);
  MakeEmpty(t->right_);
  delete t;
  t = nullptr;
}

// Clones the subtree with root t, and returns the root of the
// cloned tree.
BinaryNode *Clone(BinaryNode *t) const {
 return t == nullptr ? nullptr:
      BinaryNode{t->element_, Clone(t->left_), Clone(t->right_) };
}
```

Big Five

```
// Deallocates memory of subtree with root t.
void MakeEmpty(BinaryNode *&t) { // -> TRAVERSAL ??
  if (t == nullptr) return;
  MakeEmpty(t->left_);
  MakeEmpty(t->right_);
  delete t;
  t = nullptr;
}

// Clones the subtree with root t, and returns the root of the
// cloned tree.
BinaryNode *Clone(BinaryNode *t) const {
 return t == nullptr ? nullptr:
      BinaryNode{t->element_, Clone(t->left_), Clone(t->right_) };
}
```

Lazy Deletion

- Pros/Cons?

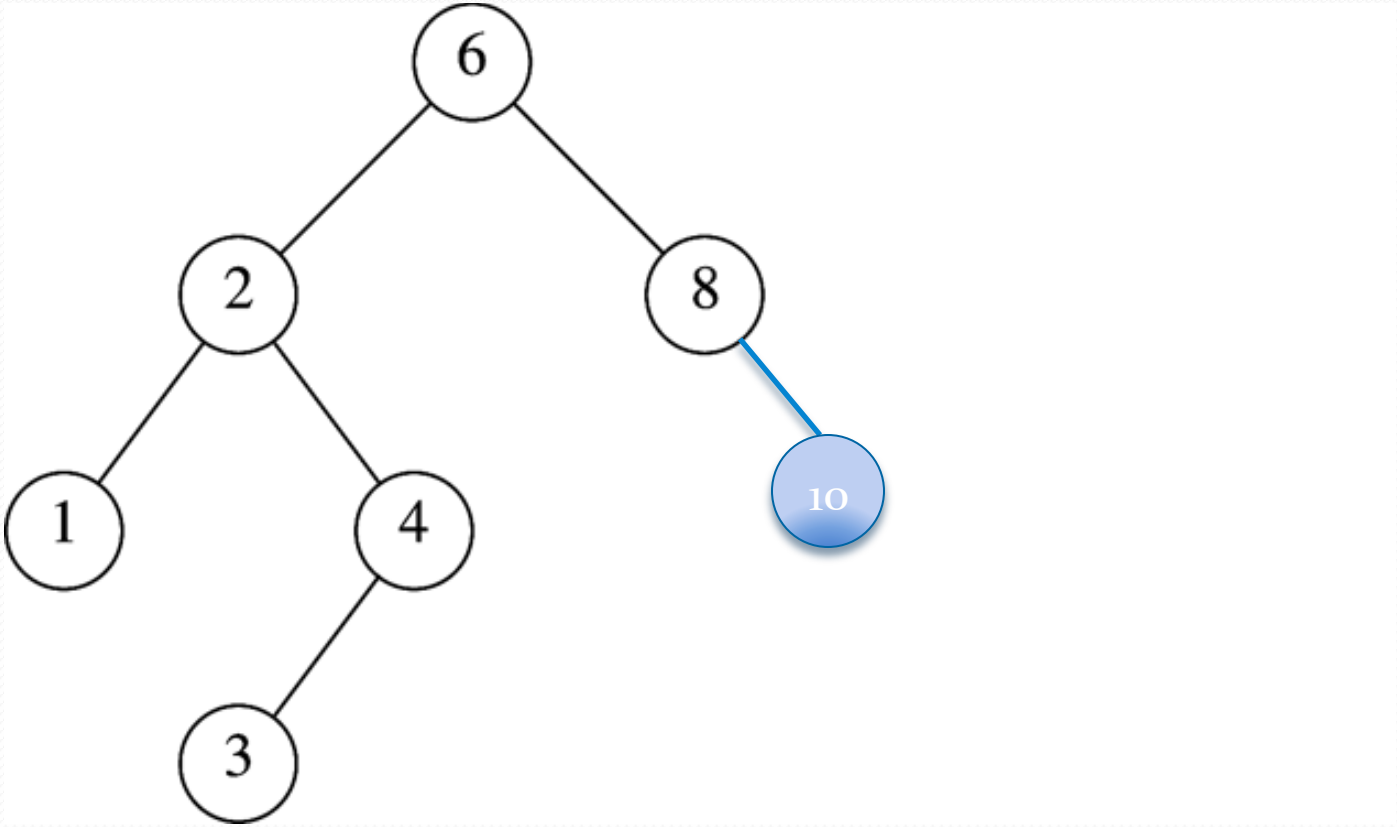
Lazy Deletion

- Small Pros
  - Easy to handle if a deleted item is reinserted, do not need to allocate a new node.
  - Do not need to handle finding a replacement node.
- Con
  - The depth of the tree will increase. However this increase is usually a small amount relative to the size of the tree.

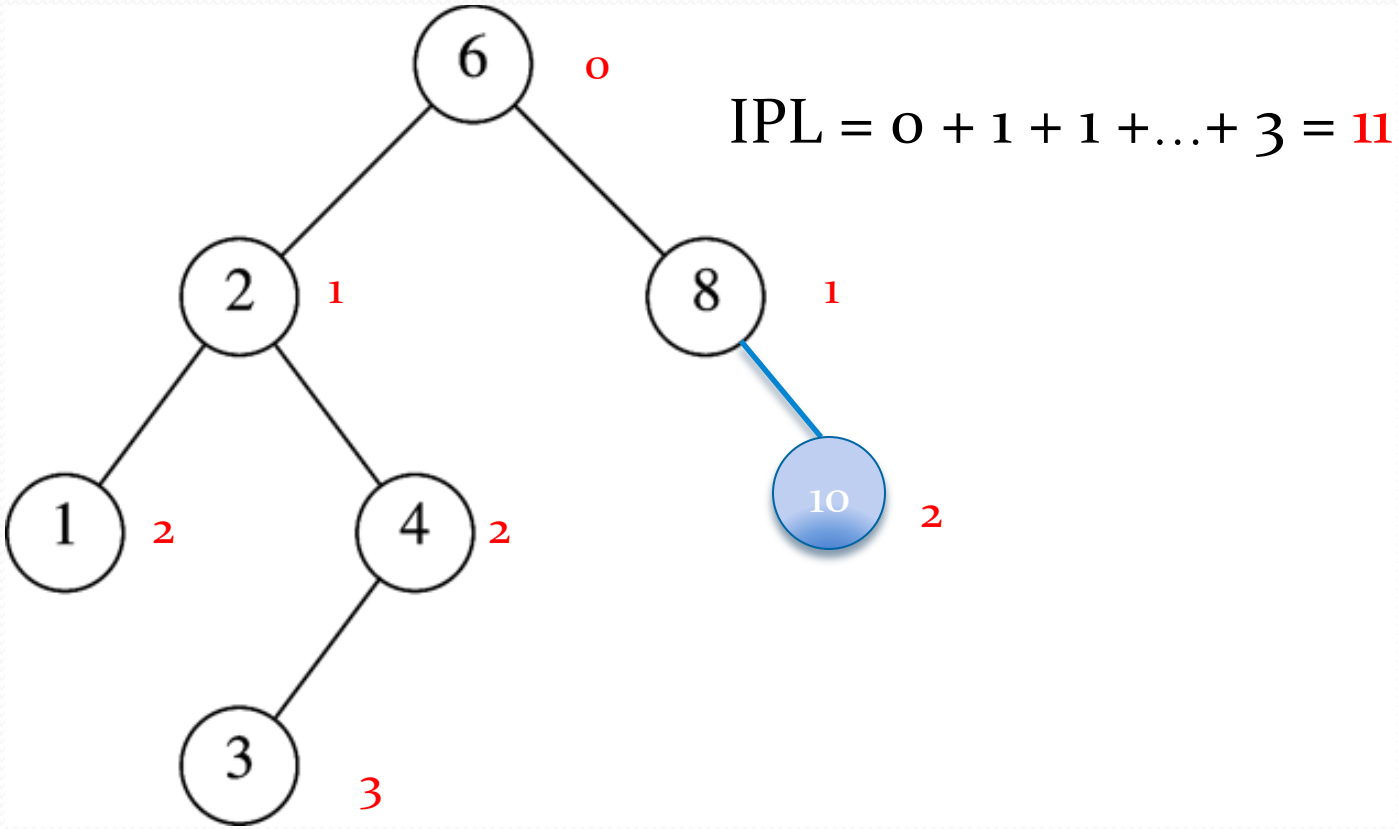
Average-Case Analysis

- **Internal path length**
  - Sum of the depths of all nodes in the tree

Internal Path Length



Internal Path Length

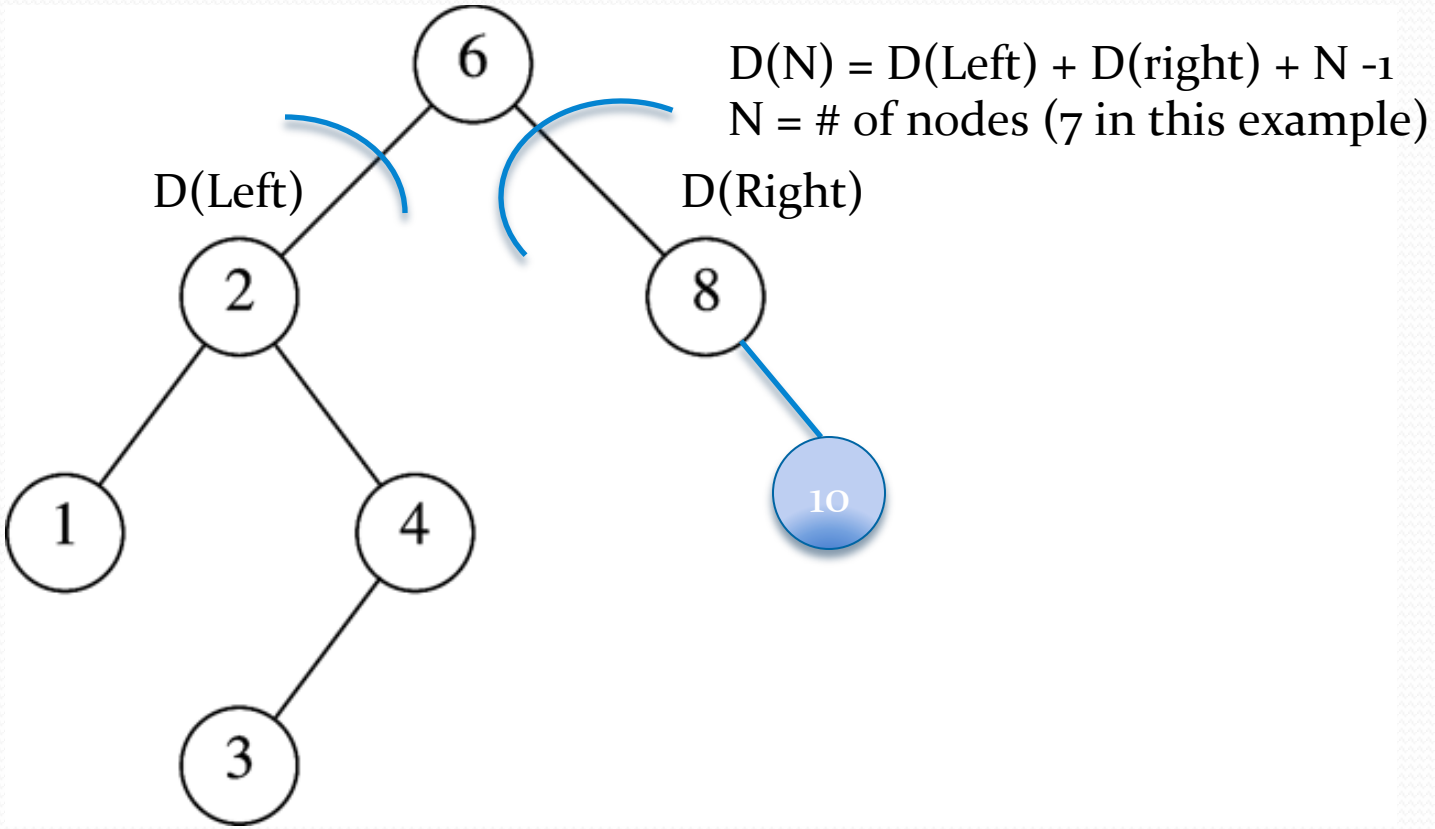


Average-Case Analysis

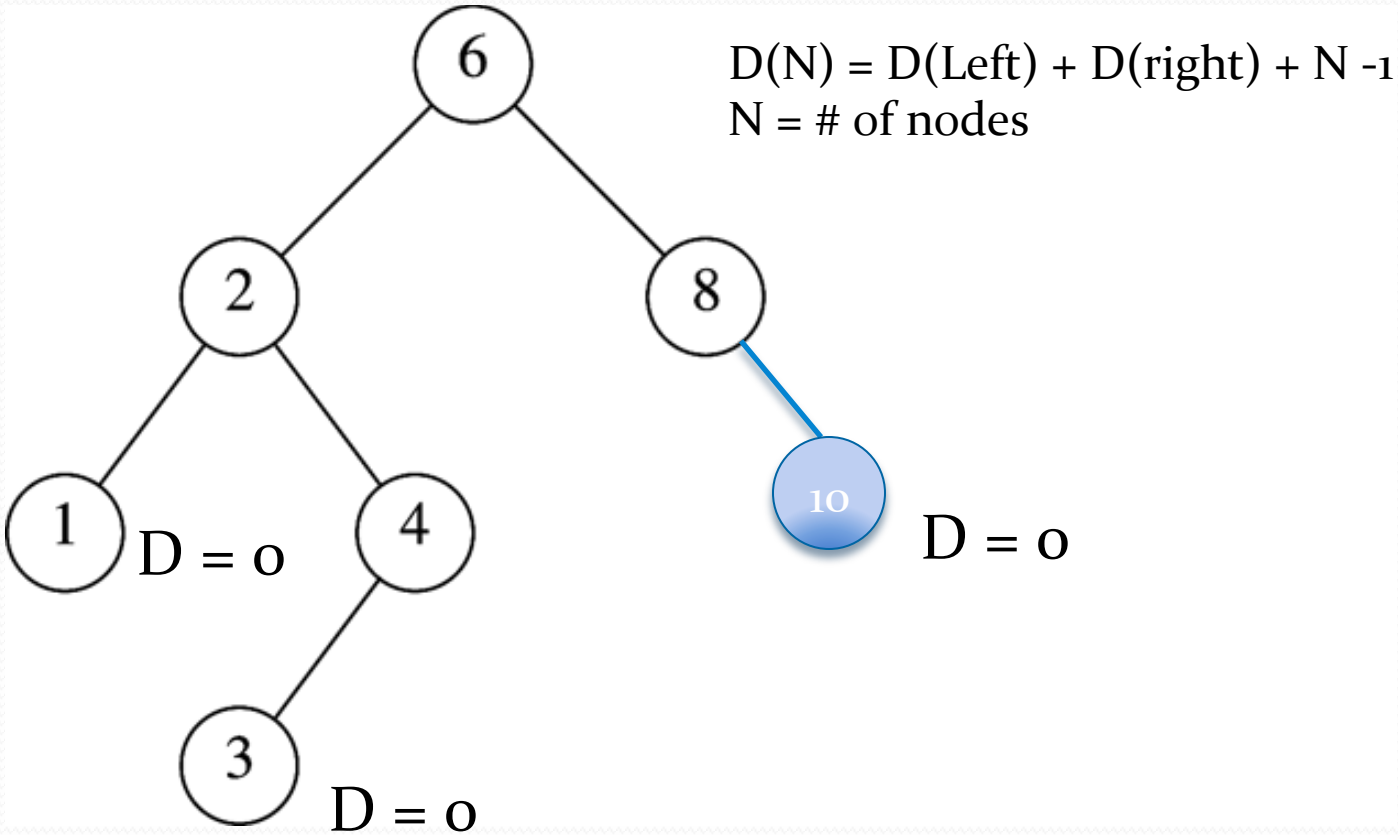
- $D(N)$: Internal Path Length of tree of N nodes
  - $D(1) = 0$
  - $D(N) = D(i) + D(N - i - 1) + (N - 1)$
  - Why ?

- In a BST the sizes of the left and right subtree depend on the first element inserted. Why?

Internal Path Length (Recursive)

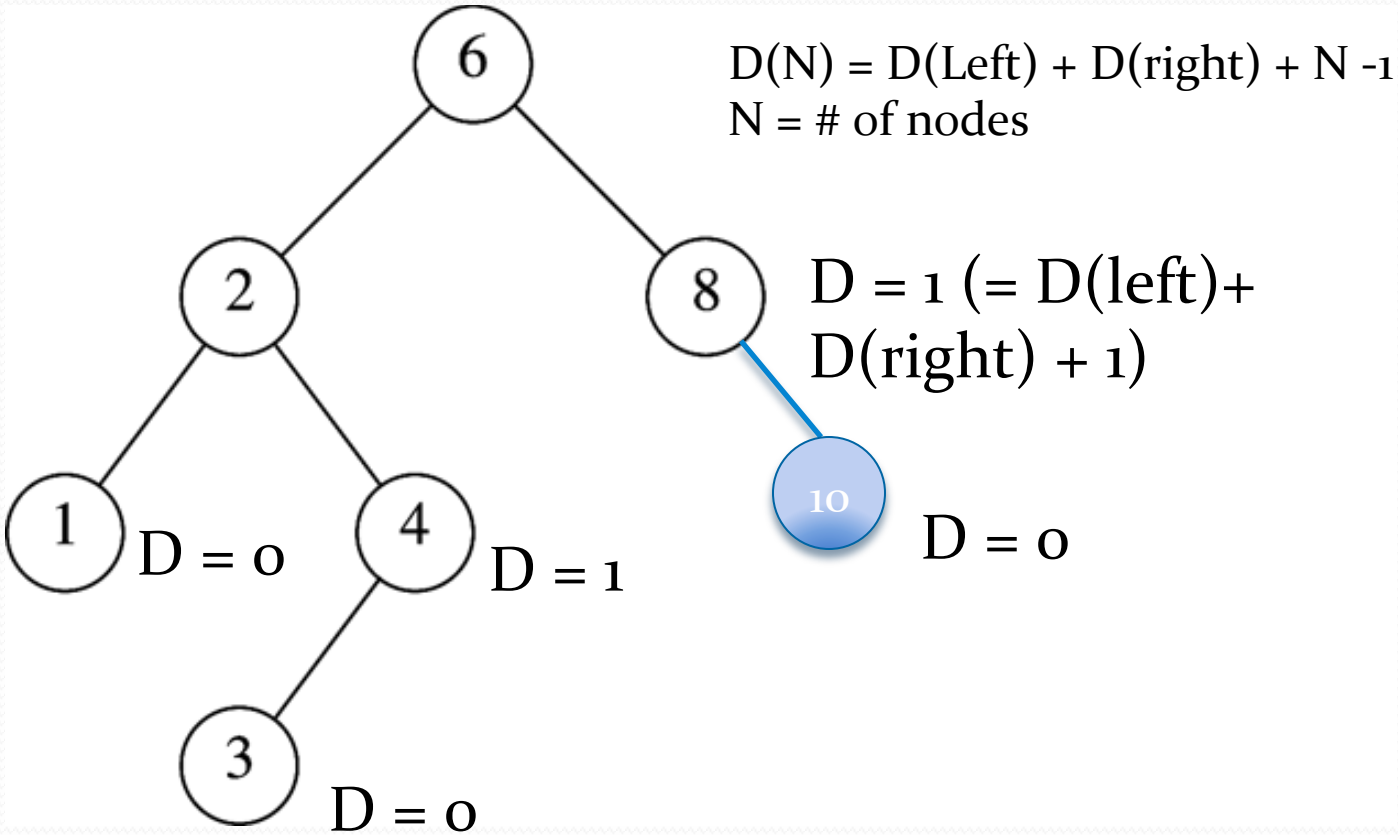


Internal Path Length (Recursive)

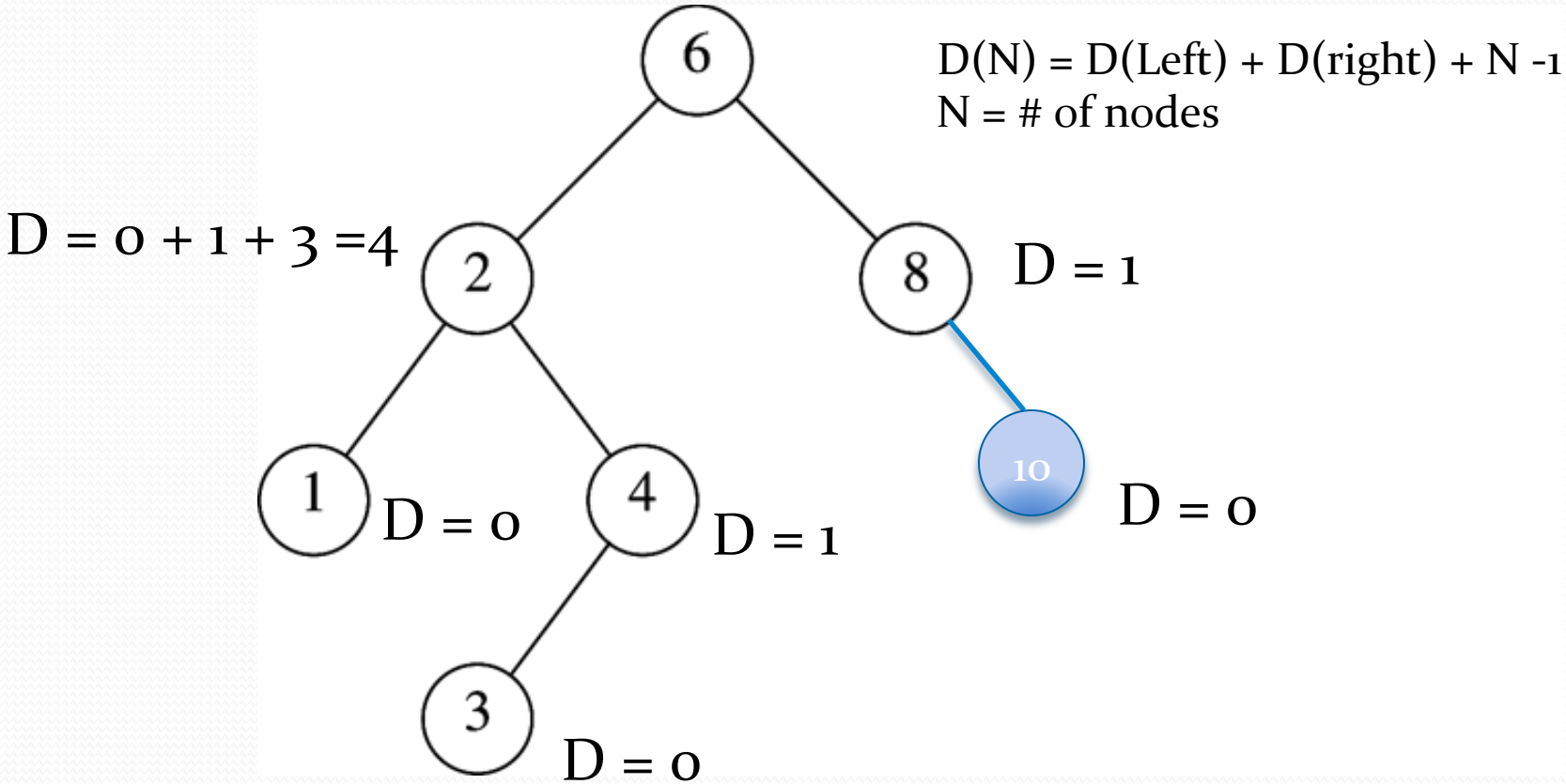


D(N) = D(Left) + D(right) + N -1
N = # of nodes

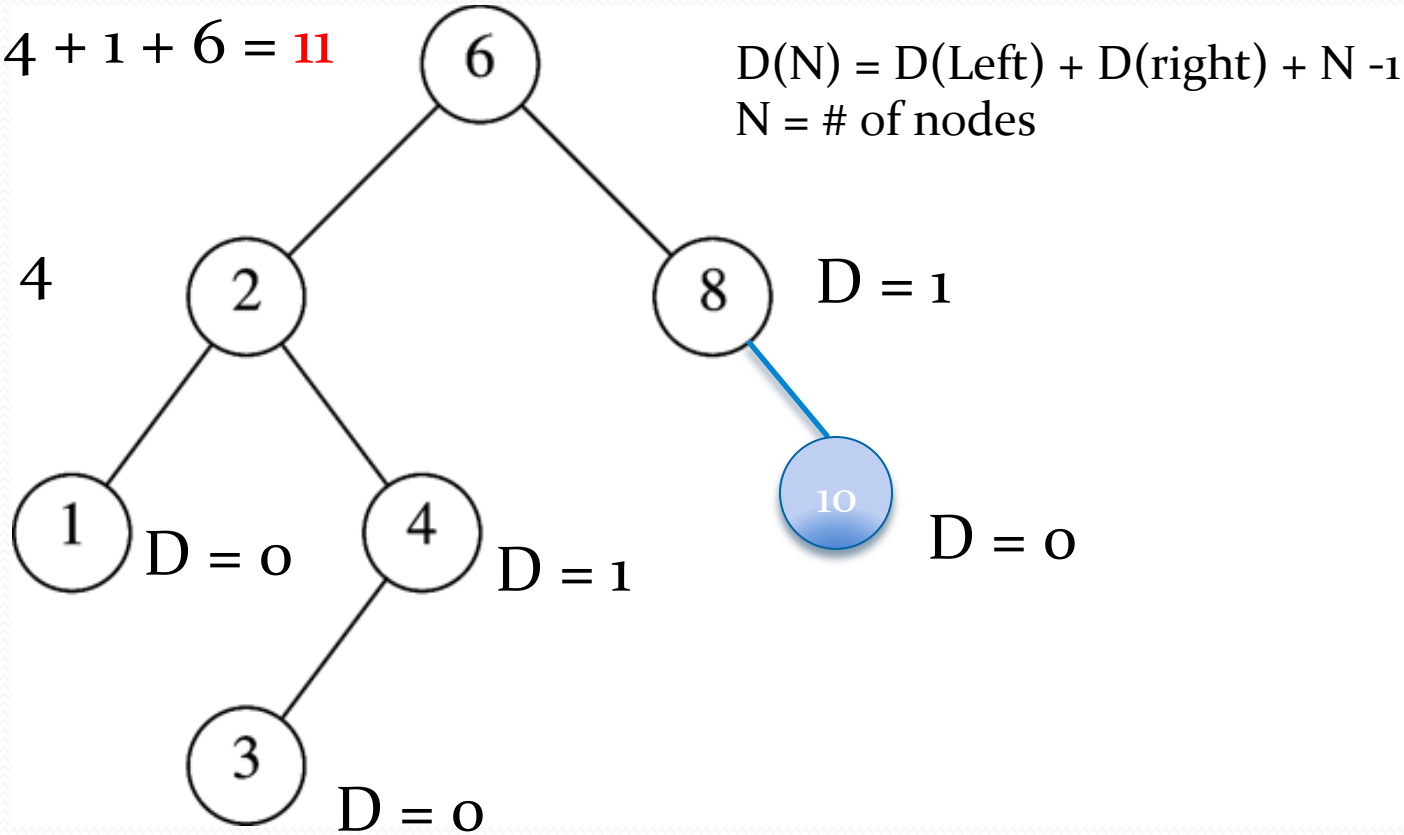
Internal Path Length (Recursive)



Internal Path Length (Recursive)



Internal Path Length (Recursive)



Average-Case Analysis

- In BST all subtree sizes are equally likely so,

$$D(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} D(j)\right] + N - 1$$

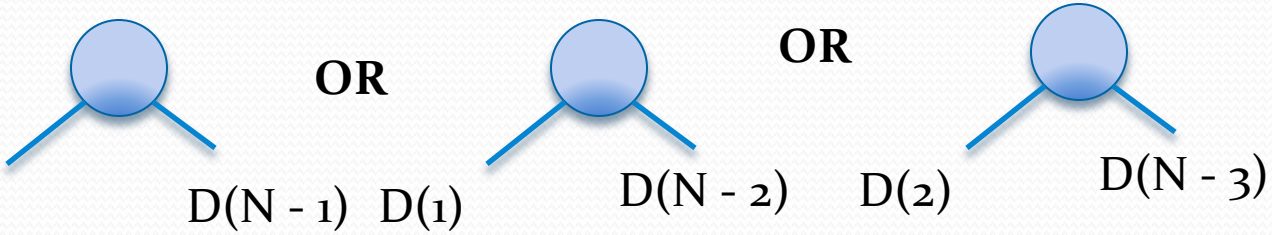
- Solving that recurrence we can show that

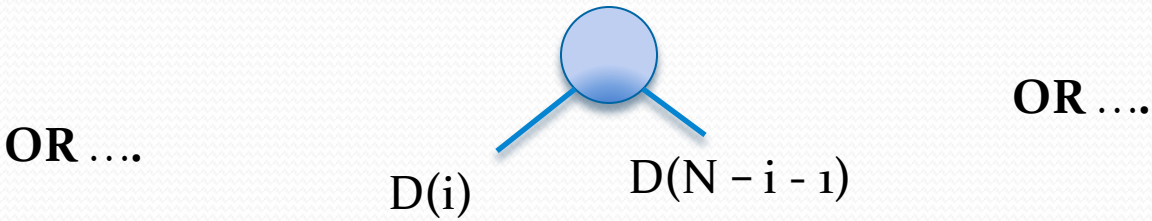
$$D(N) = O(N \log N)$$

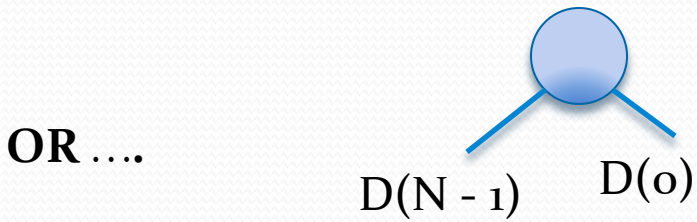
- Thus expected depth of any node is $O(\log N)$
- Example: 500 random nodes, expected depth 9.98
- However not all inputs are equally likely!

Average Case Analysis: Possible Binary Search Trees

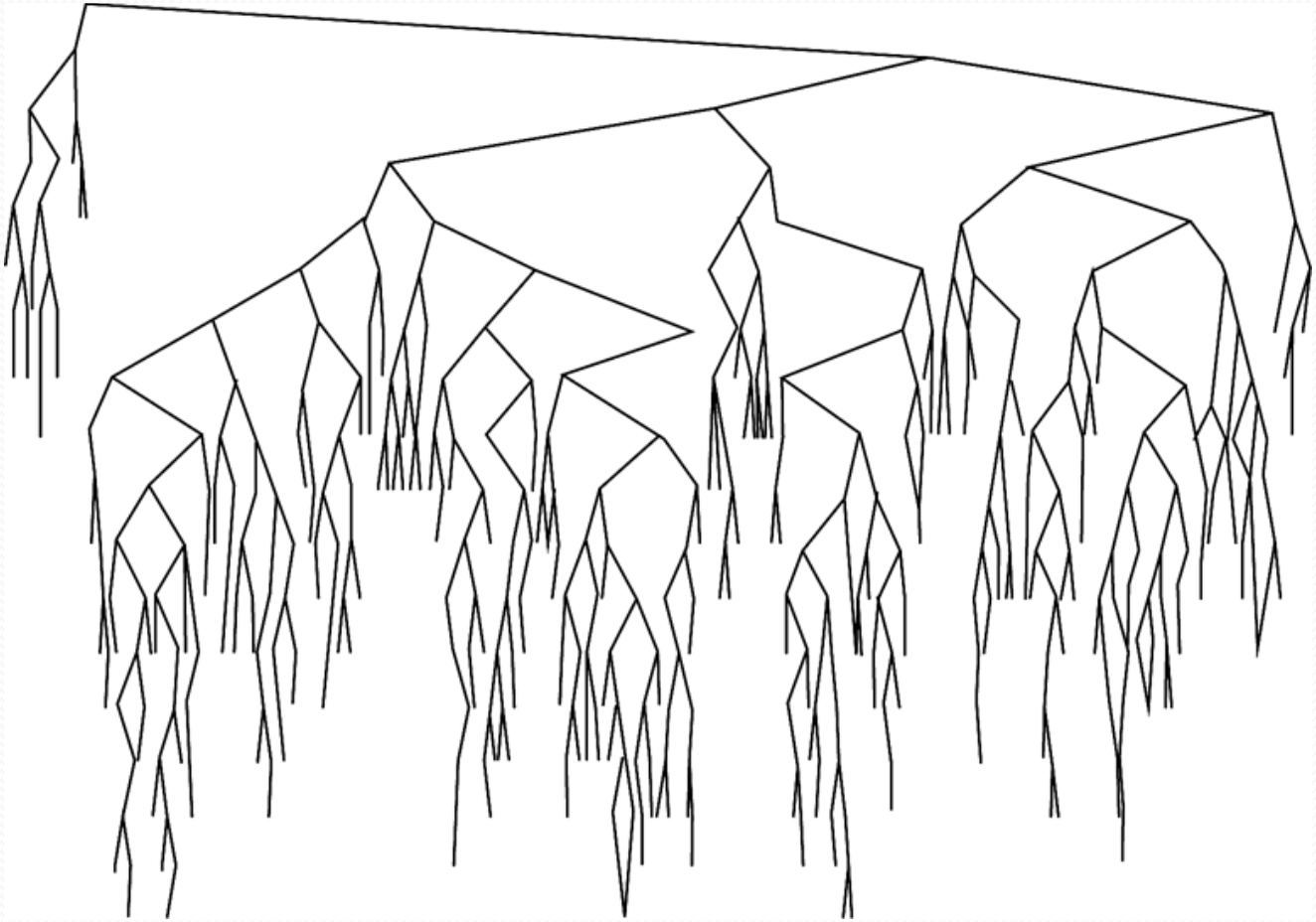
D(N):







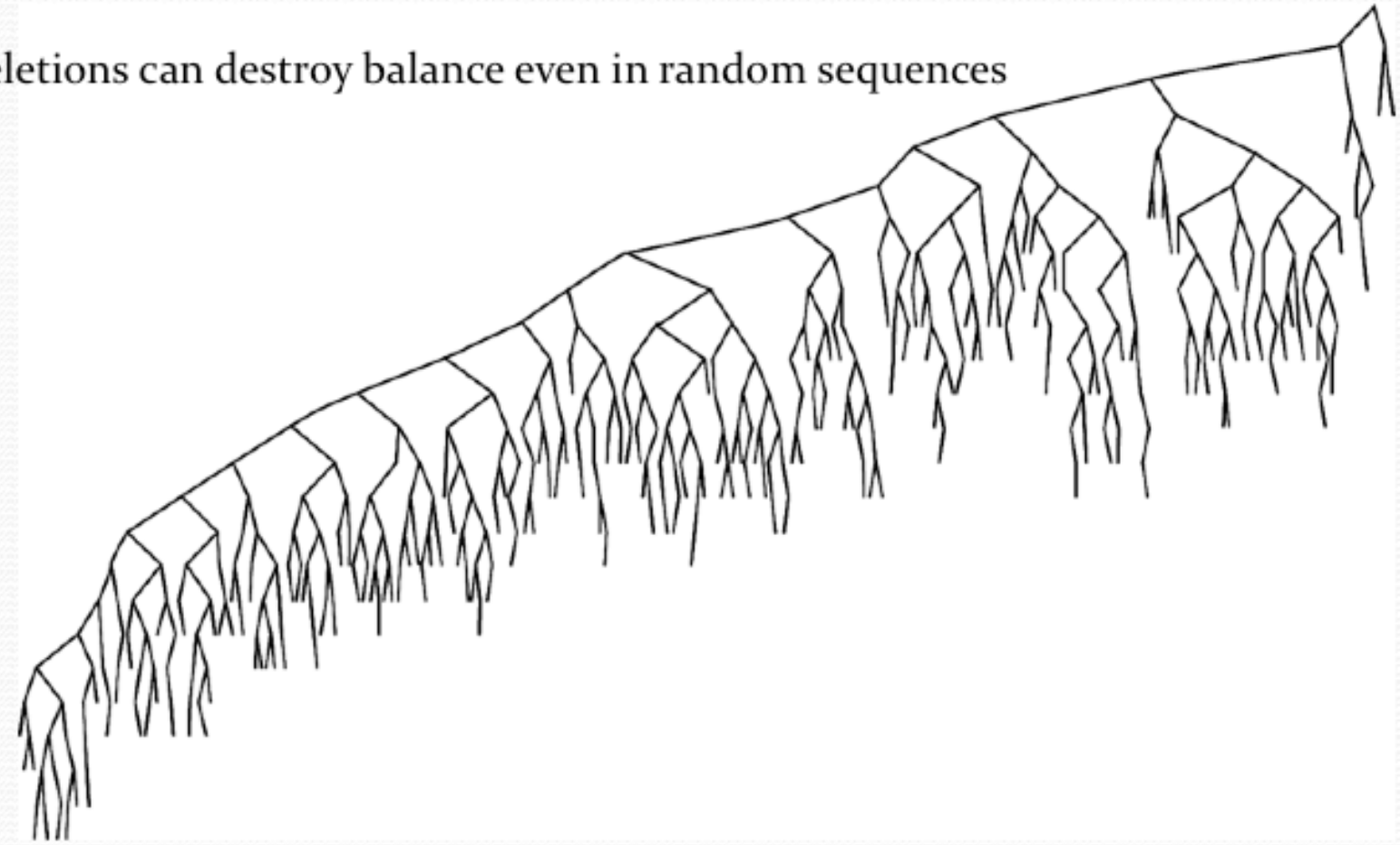
Average case analysis



Randomly generated BST

Average case analysis

Deletions can destroy balance even in random sequences



After $\Theta(N^2)$ insert/remove pairs, expected depth is $\Theta(\sqrt{N})$

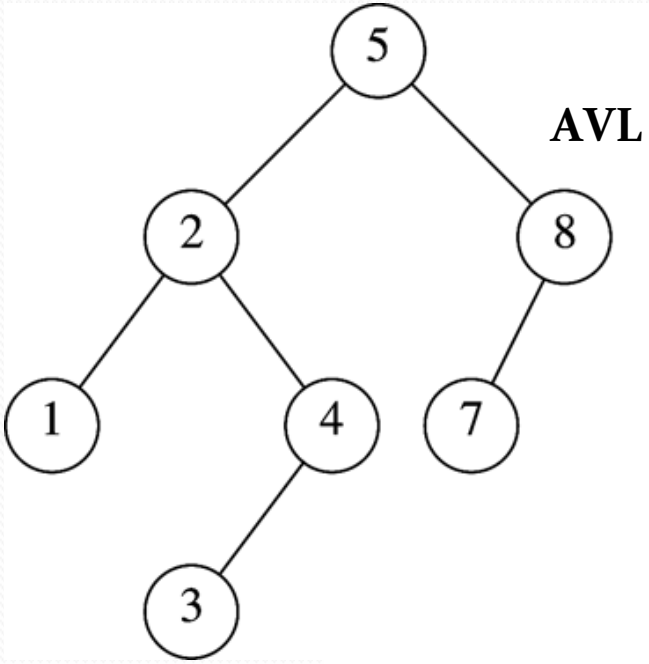
Average case analysis

- Deletion algorithm favors making left subtrees deeper than right subtrees, because we always replace a deleted node from the right subtree.
  - Can use a different replacement strategy with less bias but there is no proof this would be better.
- What is the worst-case input for a BST?

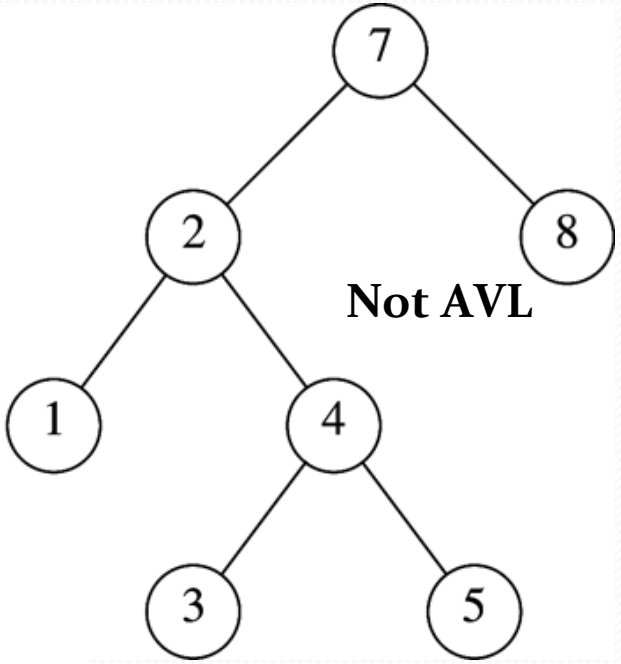
Balanced Trees

- Try to balance after tree operations and make operations logarithmic.
  - AVL tree (Balance is always preserved)
  - Splay tree (Self-adjusts towards balance)

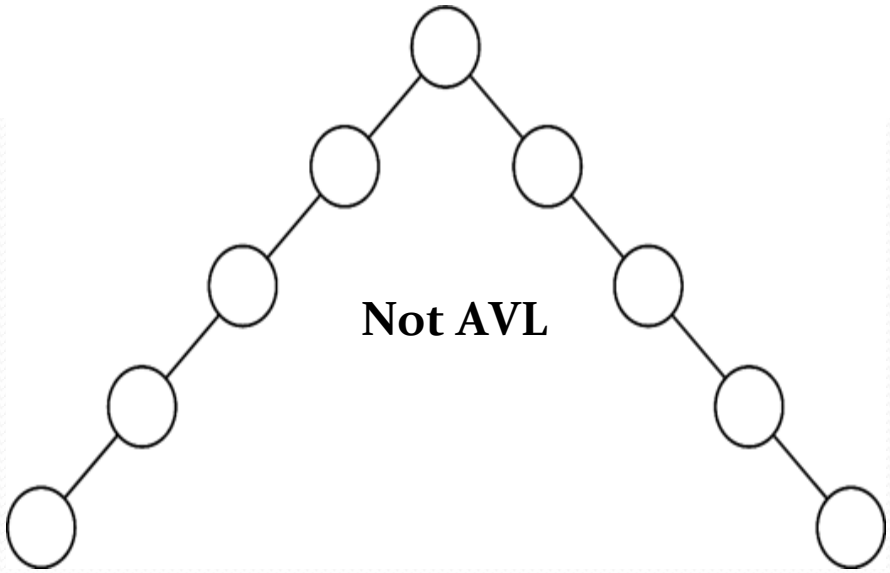
AVL Trees



AVL



Not AVL

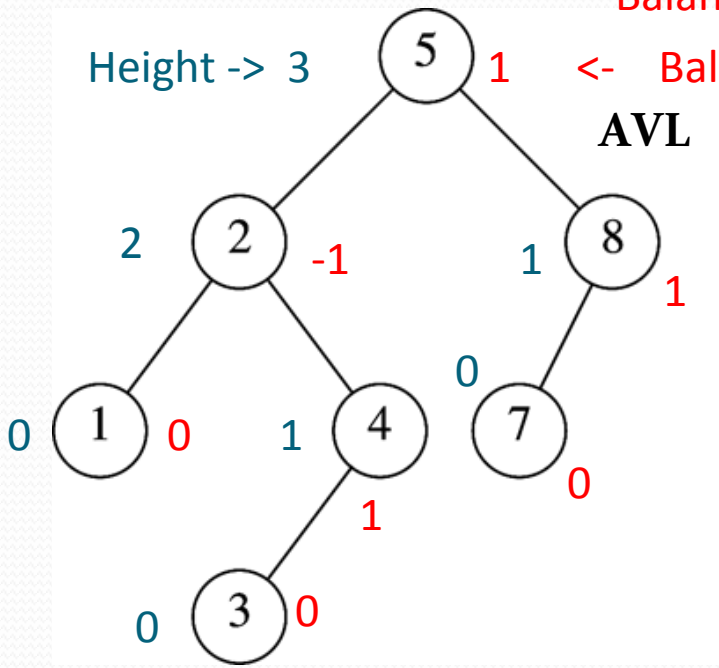


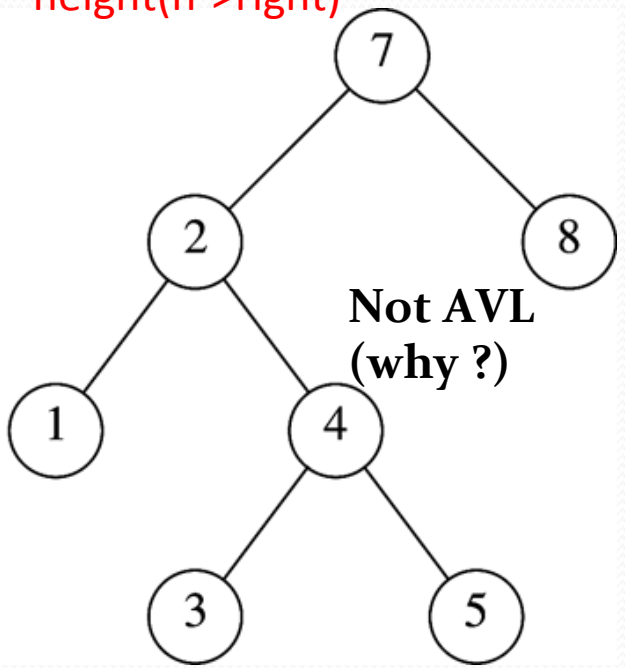
Not AVL

AVL Trees

n: pointer to a node

Balance(n) = height(n->left) – height(n->right)





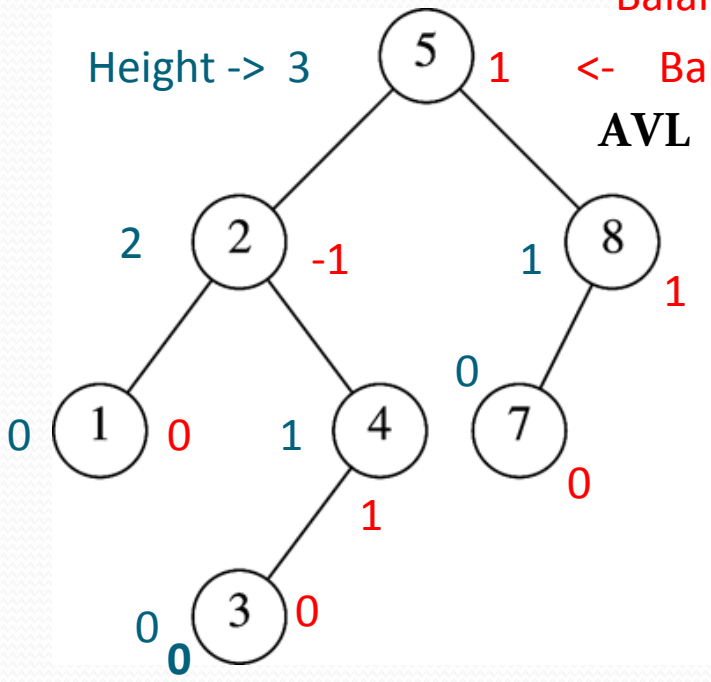
height(n) = 1 + max{height(n->left), height(n->right)}

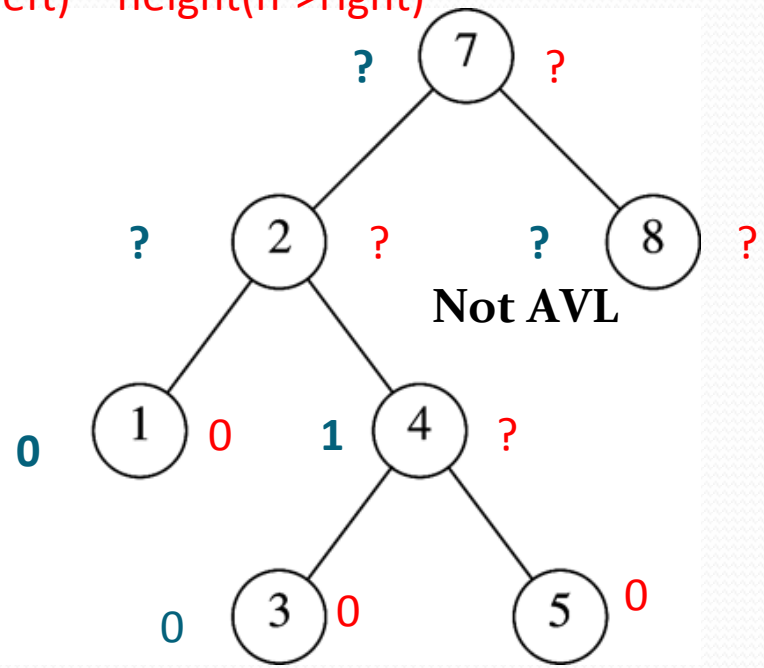
height(nullptr) = -1

AVL Trees

n: pointer to a node

Balance(n) = height(n->left) – height(n->right)

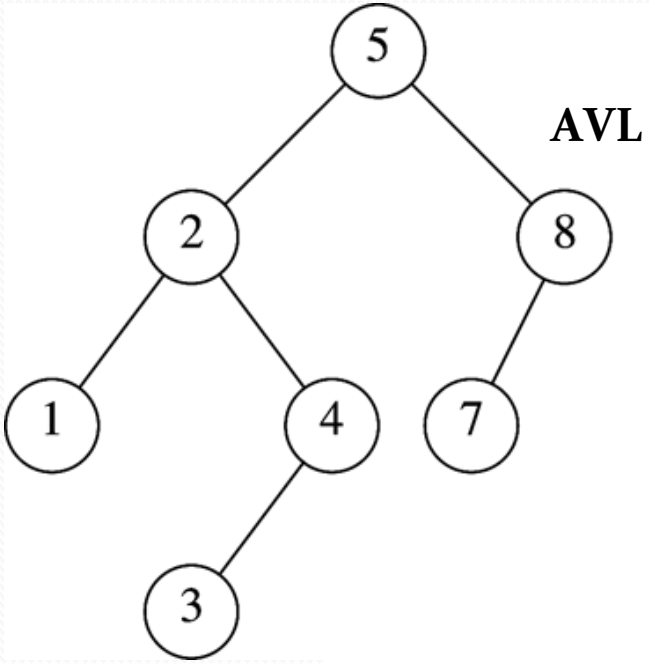




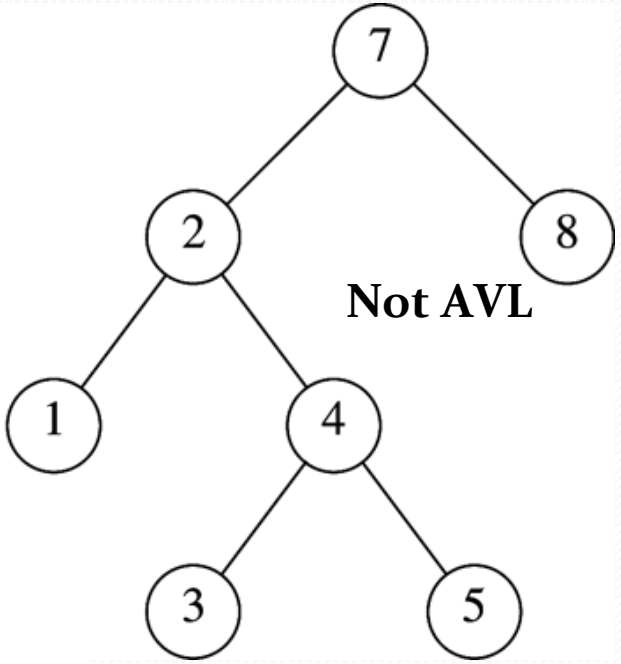
height(n) = 1 + max{height(n->left), height(n->right)}

height(nullptr) = -1

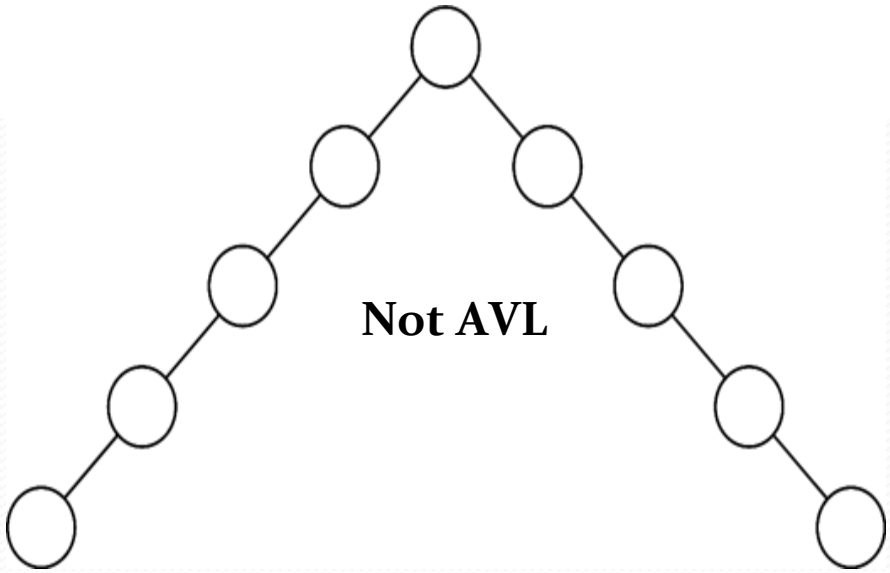
AVL Trees



AVL



Not AVL



Not AVL

AVL trees

- Adelson-Velskii and Landis
- Binary trees
- Balance condition ensures depth is $O(\log N)$
- For all nodes in the tree, the height of the left subtree minus the height of the right subtree must be either $0, 1,$ or -1.
- Assume that an empty subtree has height -1.

AVL trees [height on min. number of nodes]

- Minimum number (**N**) of nodes of AVL tree of height h:

$S(h) = S(h-1) + S(h-2) + 1$

$S(0)=1,\ S(1) = 2$

<Why ?>

- S(h) closely related to Fibonacci numbers!

AVL trees

- S(h) := Minimum number of nodes of AVL tree of height h:

S(0) = ?

S(1) = ?

S(2) = ?

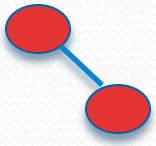
S(3) = ?

AVL trees

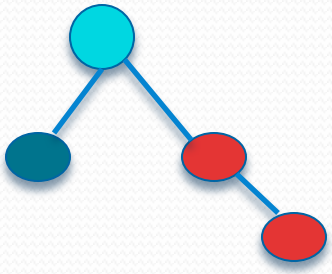
- S(h) := Minimum number of nodes of AVL tree of height h:

S(0) = 1

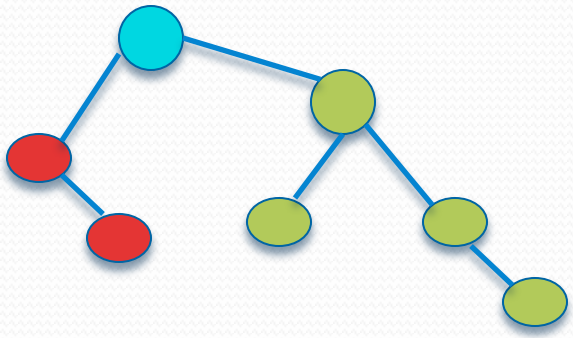
S(1) = 2



S(2) = 1 + 1 + 2 = 4



S(3) = 1 + 2 + 4 = 7



AVL trees [height on min. number of nodes]

- Minimum number (**N**) of nodes of AVL tree of height h:

$S(h) = S(h-1) + S(h-2) + 1$

$S(0)=1,\ S(1) = 2$

<Why ?>

- S(h) closely related to Fibonacci numbers!

Number of nodes grows exponentially as height increases =>

- We know for Fib. Sequence that

$$1.5^n < F(n) < (1.66)^n$$

Therefore: $S(h) = \Theta(a^h)$ [a is a small constant]

$\Rightarrow \mathbf{h} = \Theta(\log(S(h)) = \Theta(\log \mathbf{N})$

AVL trees [height on min. number of nodes]

- Minimum number (**N**) of nodes of AVL tree of height h:

  $S(h) = S(h - 1) + S(h - 2) + 1$

  $S(0)=1, S(1) = 2$

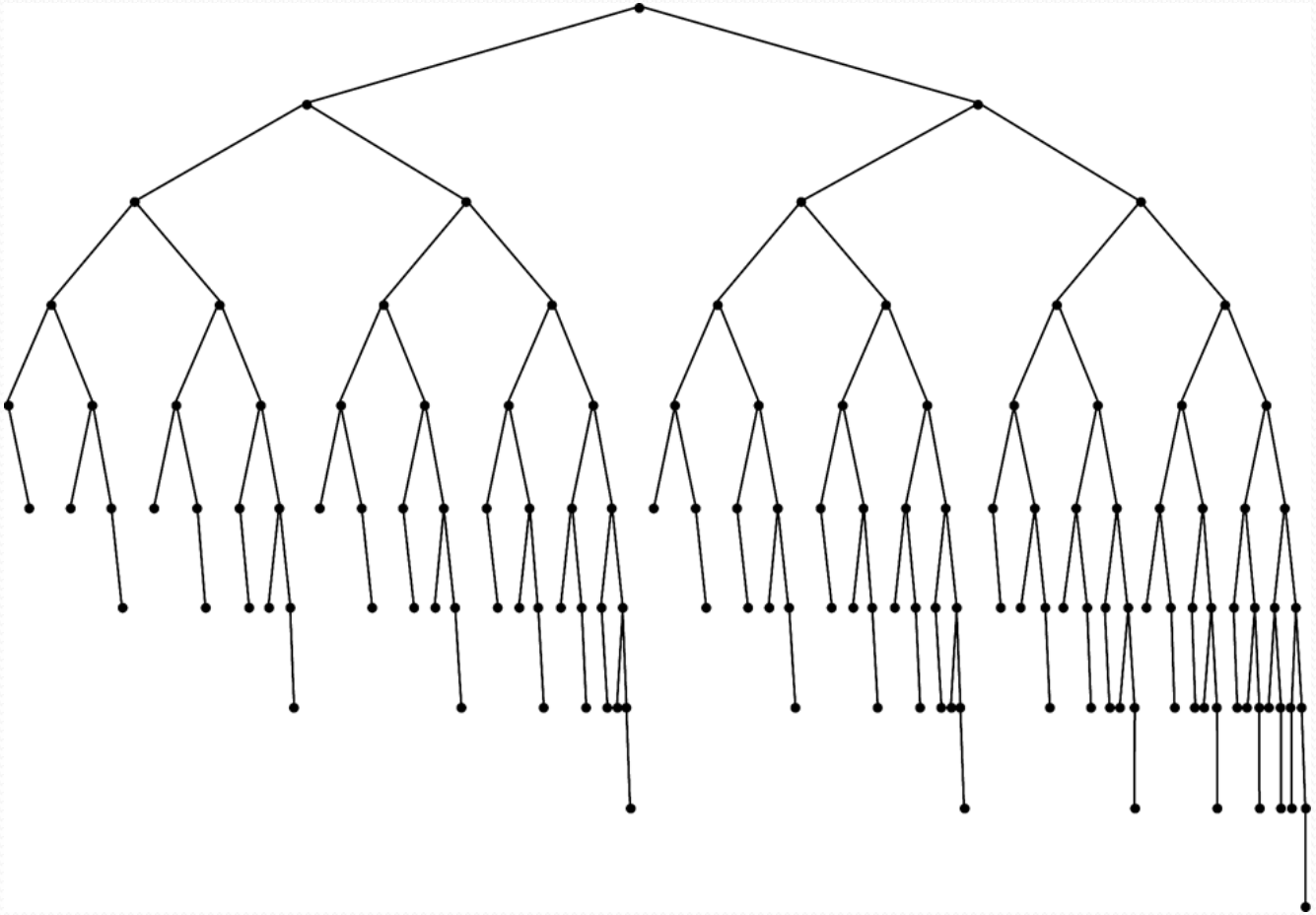
  <Why ?>
- S(h) closely related to Fibonacci numbers!

Number of nodes grows exponentially as height increases =>

⇒**h** roughly $1.44 \log(N + 2) - 1.328$

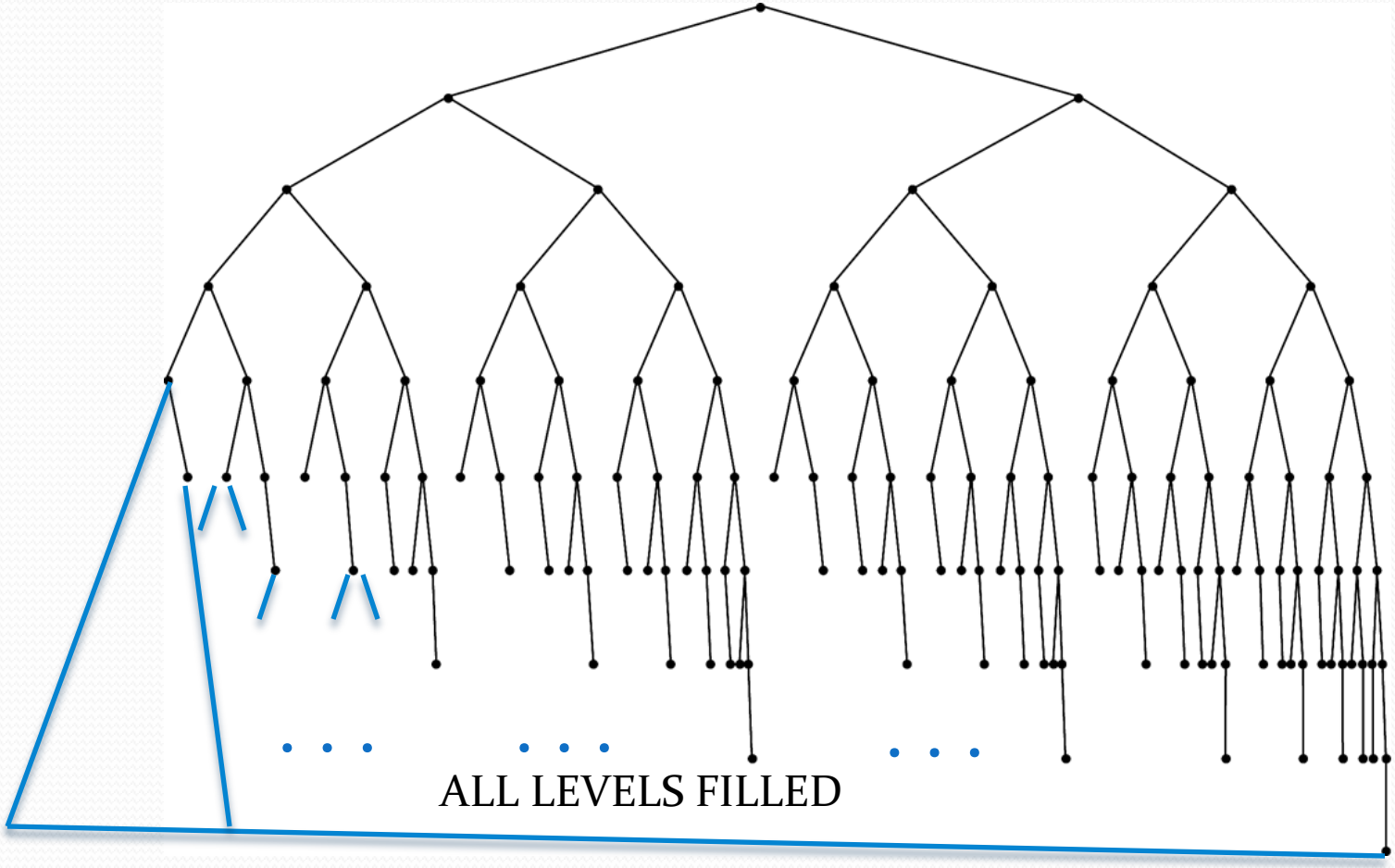
⇒A little more than logN

AVL trees: smallest number of nodes for tree of height 9)



AVL trees: largest number of nodes for tree of height 9

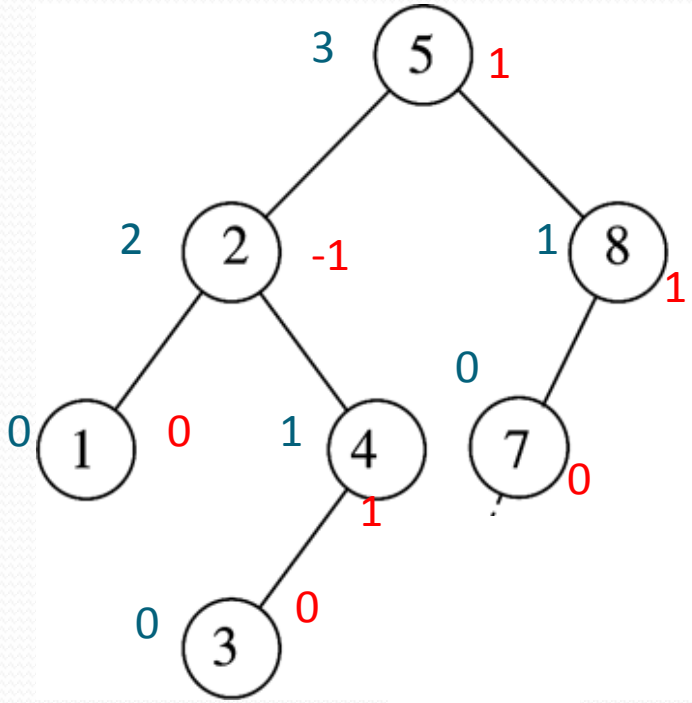
$N = 2^{h+1} - 1$: number of nodes for full tree of height h



Insertion cases

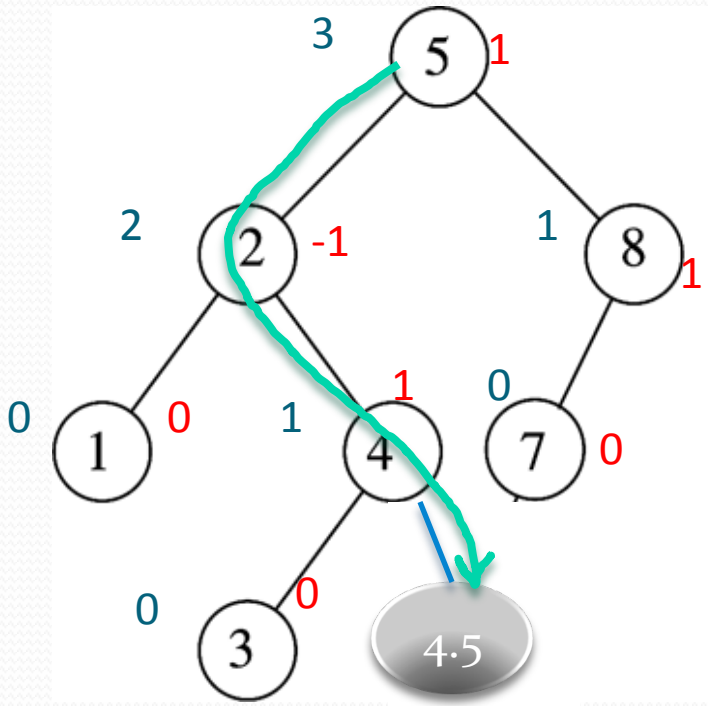
- After insertion only nodes on the path between insertion point and the root may have their balance altered.
- Suppose that α is the first node on the path that needs to be rebalanced.
- Four cases of violation. Insertion into:
  1. Left subtree of left child of α
  2. Right subtree of left child of α
  3. Left subtree of right child of α
  4. Right subtree of right child of α
- Rotation operation at α will balance the tree.

Example: insert 4.5 in this AVL tree



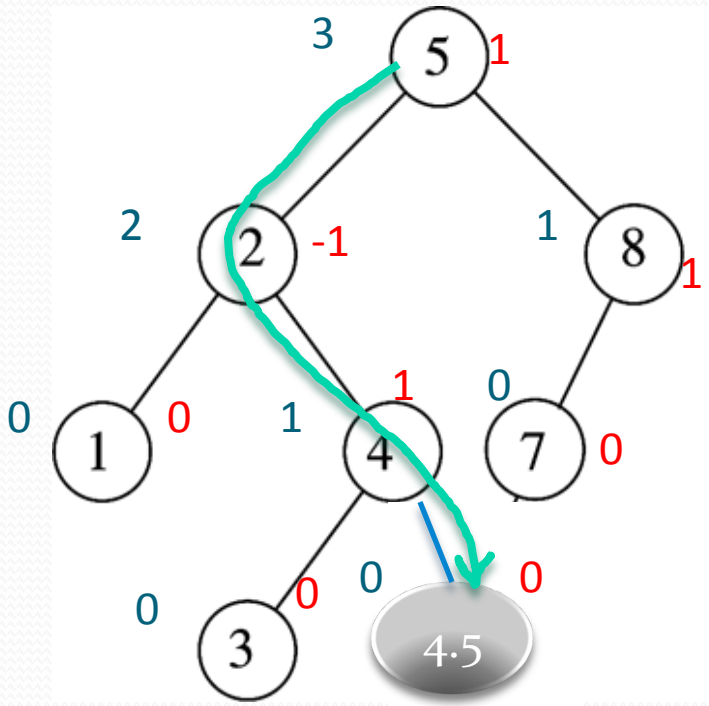
Example: insert 4.5 in this AVL tree

Nodes whose height will be affected by insertion



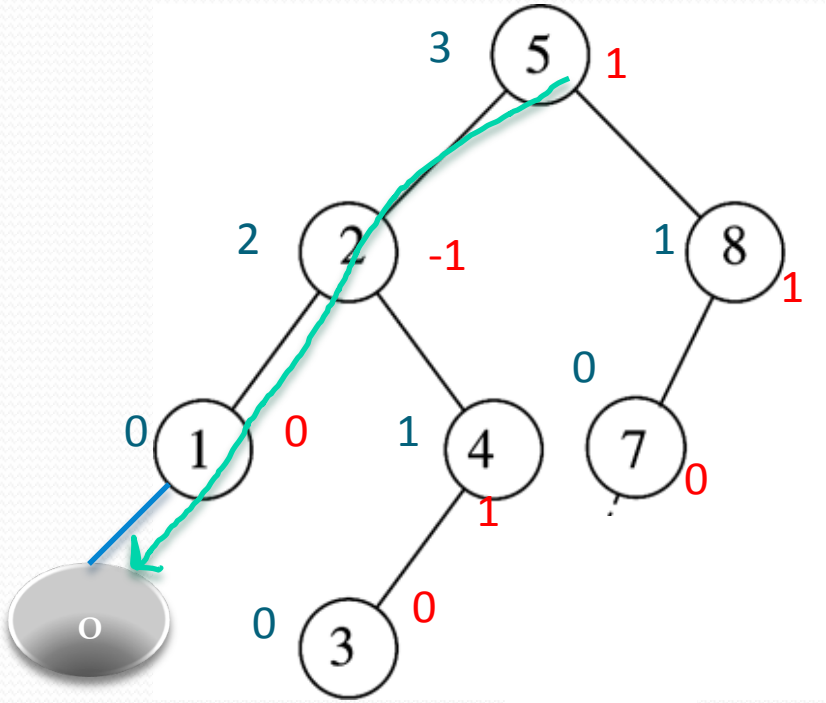
Example: insert 4.5 in this AVL tree

Nodes whose height maybe affected by insertion



Example: insert 0 in this AVL tree

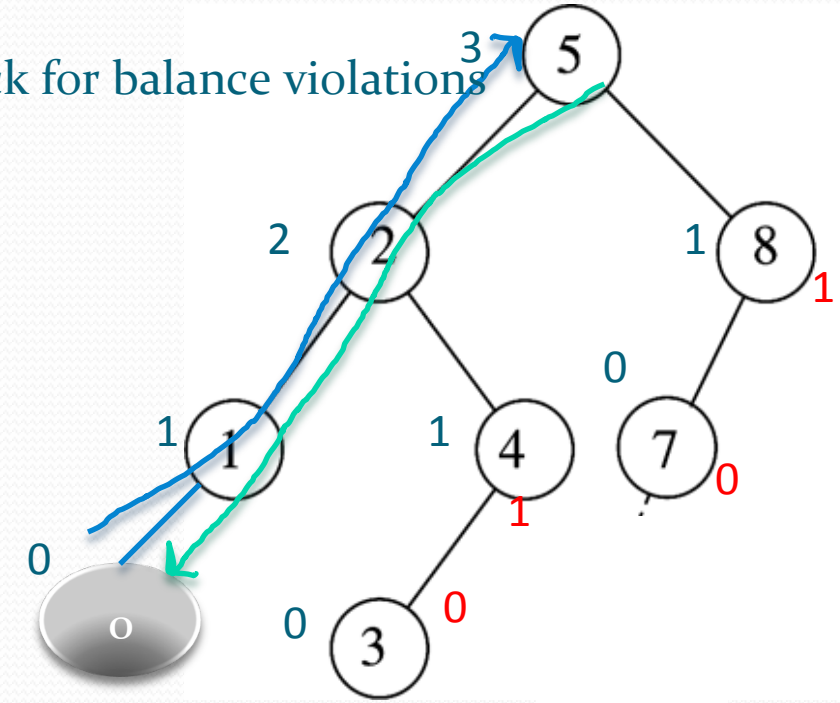
Nodes whose height maybe affected by insertion



Example: insert 0 in this AVL tree

Nodes whose height maybe affected by insertion

Check for balance violations

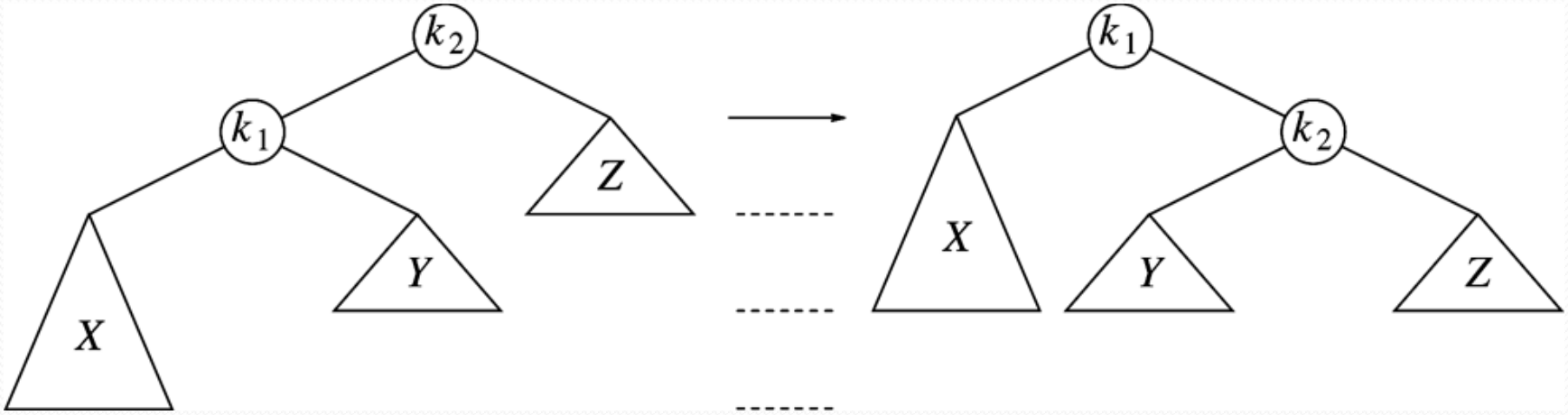


Insertion cases

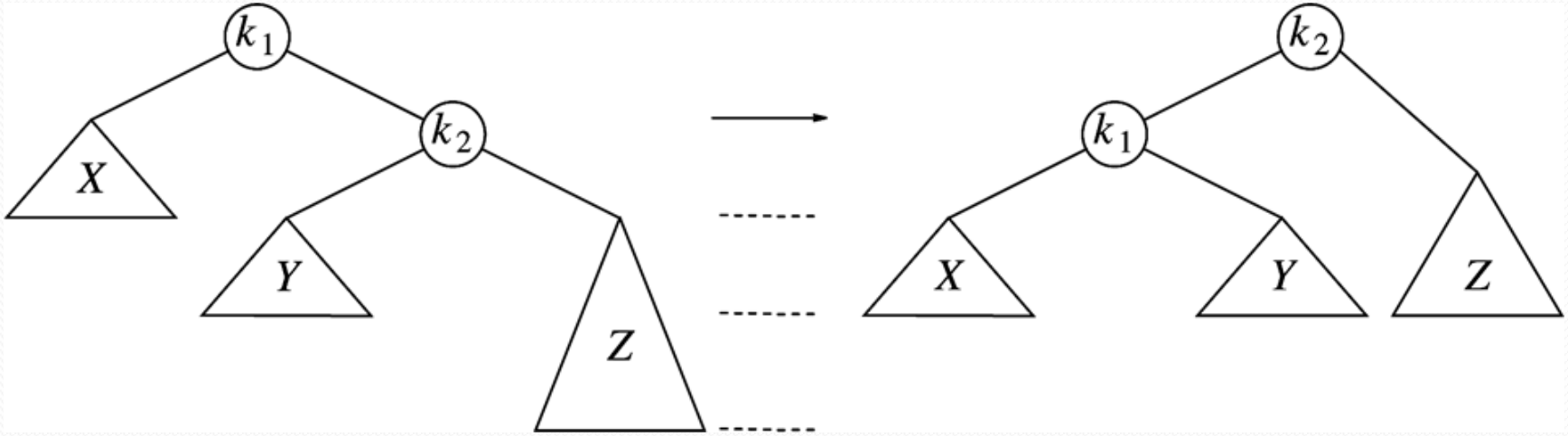
- 1,4 (left-left or right-right) : Single rotation
- 2,3 (right-left or left-right) : Double rotation

- Implementation of double rotation simply involves two calls to the single rotation function.
- Conceptually it may be easier to think about it as a different operation.

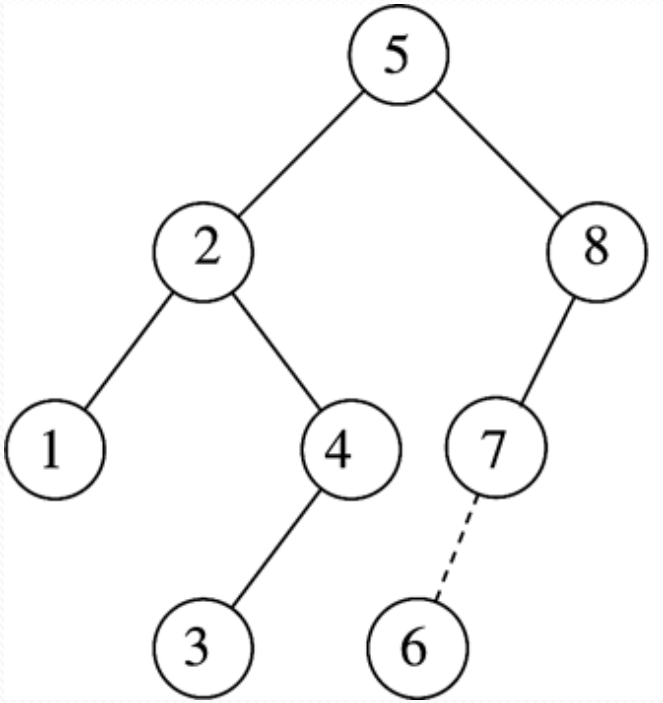
Single rotation (case 1)

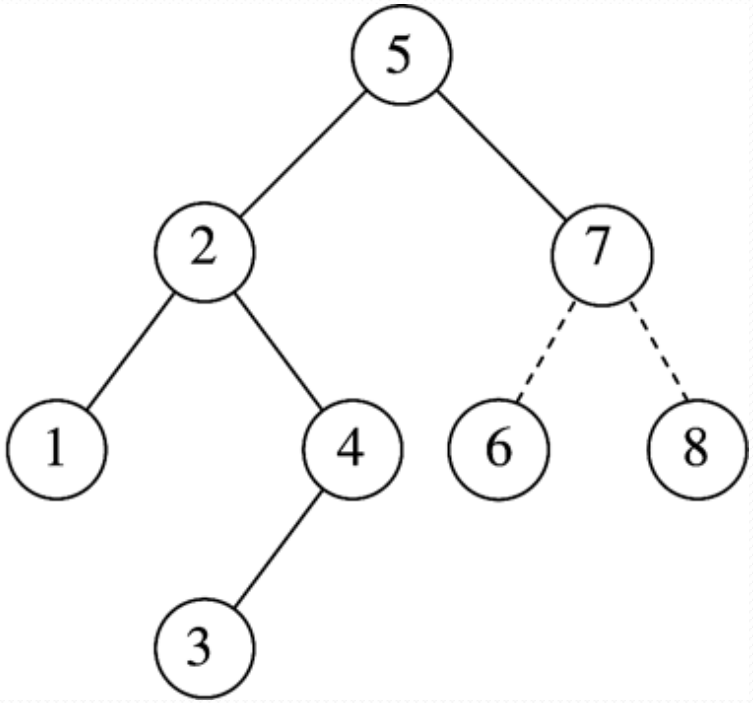


Single rotation (case 4)

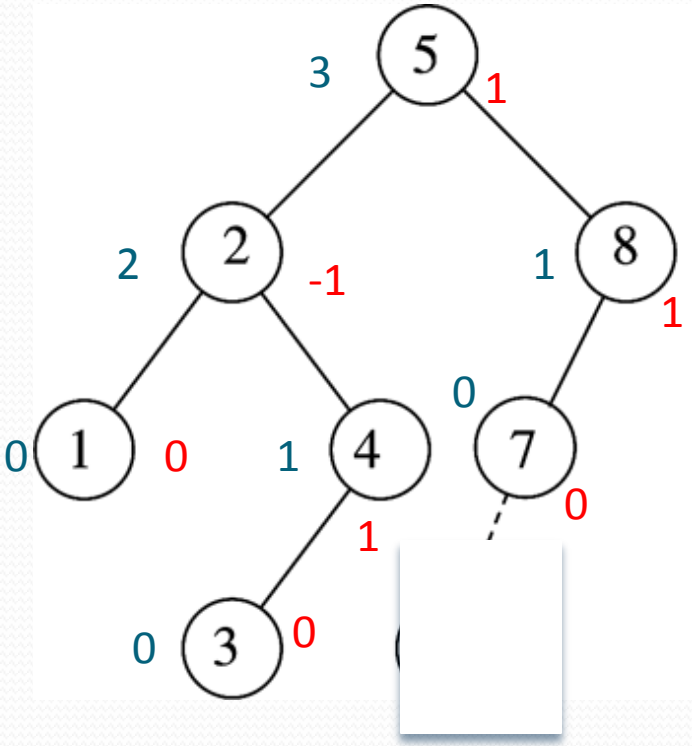


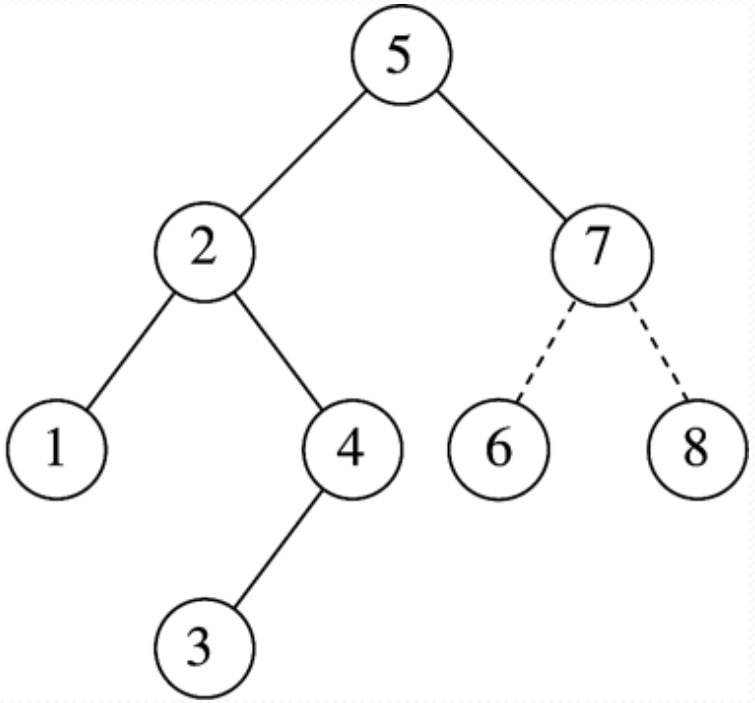
Example, case 1





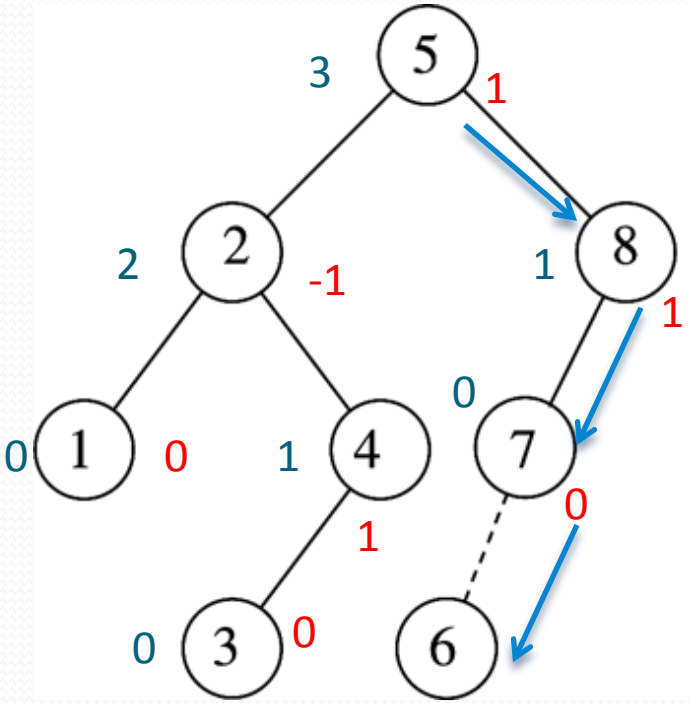
Example, case 1 (insert 6)

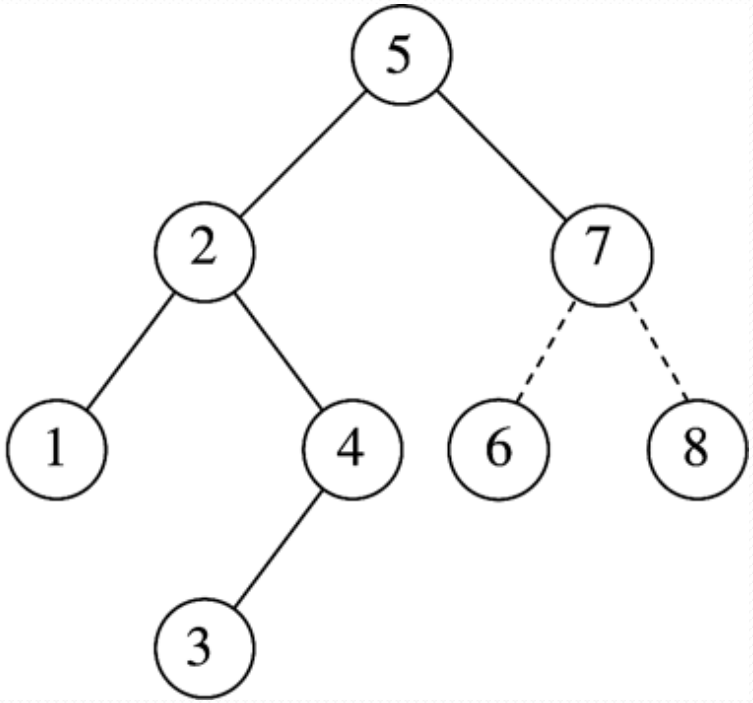




Example, case 1 (insert 6)

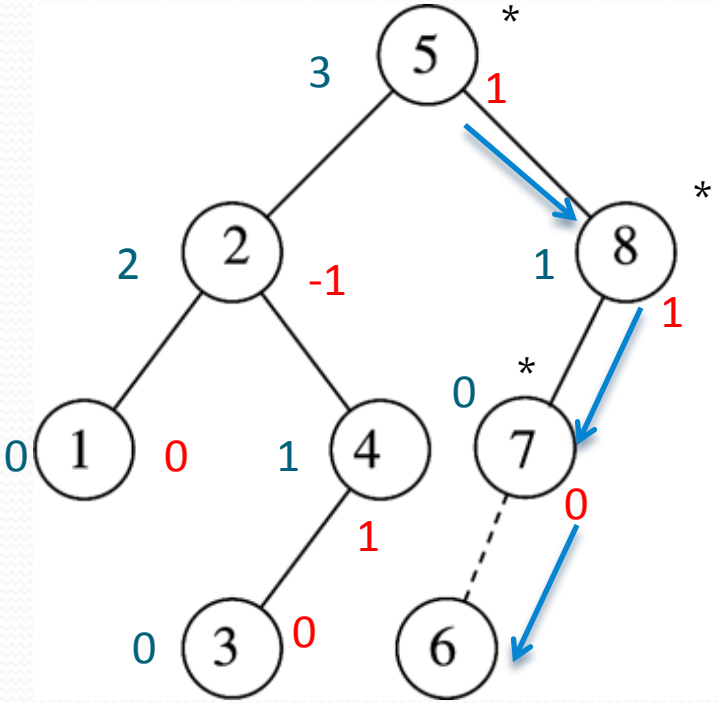
VIOLATIONS OF BALANCE POSSIBLE ONLY AT PATH OF INSERTION

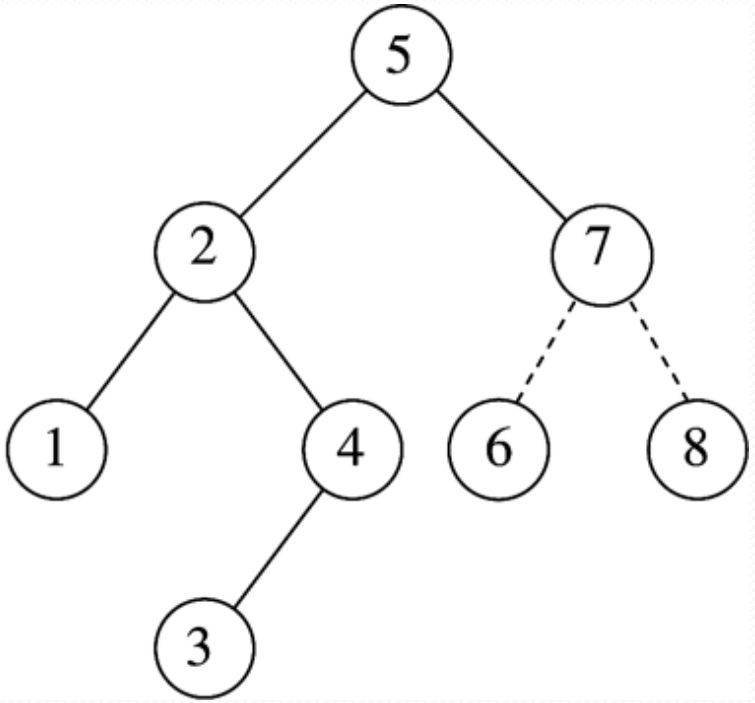




Example, case 1 (insert 6)

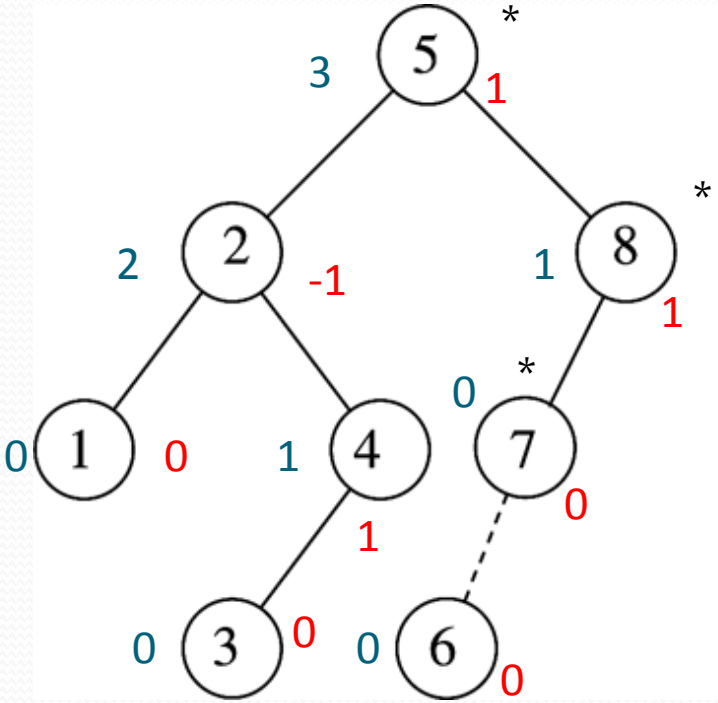
VIOLATIONS OF BALANCE POSSIBLE ONLY AT PATH OF INSERTION

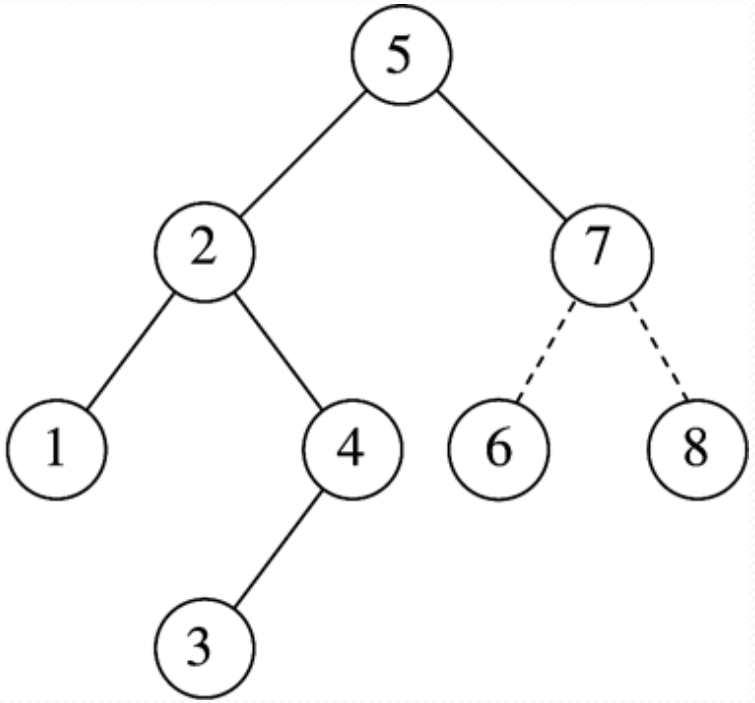




Example, case 1 (insert 6)

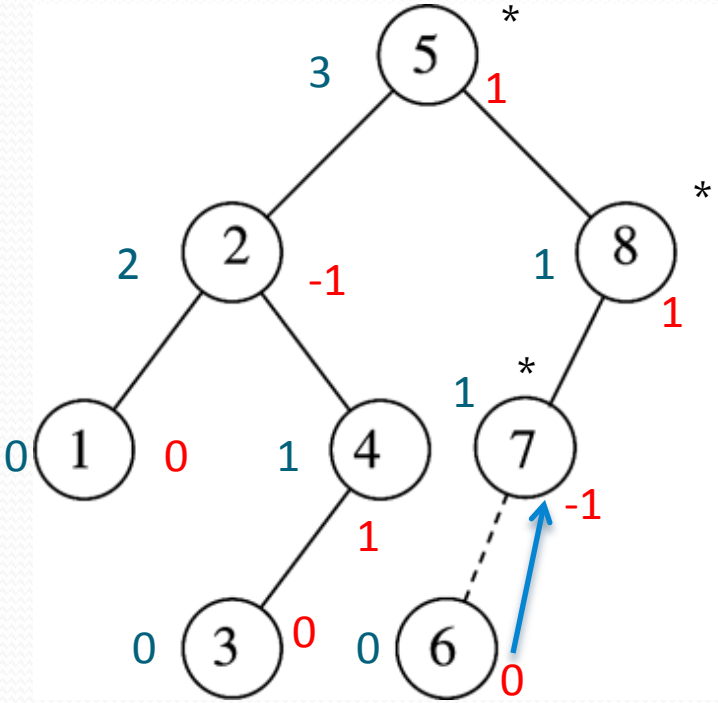
WALK IN OPPOSITE DIRECTION – UPDATE HEIGHTS AND CHECK FOR VIOLATIONS

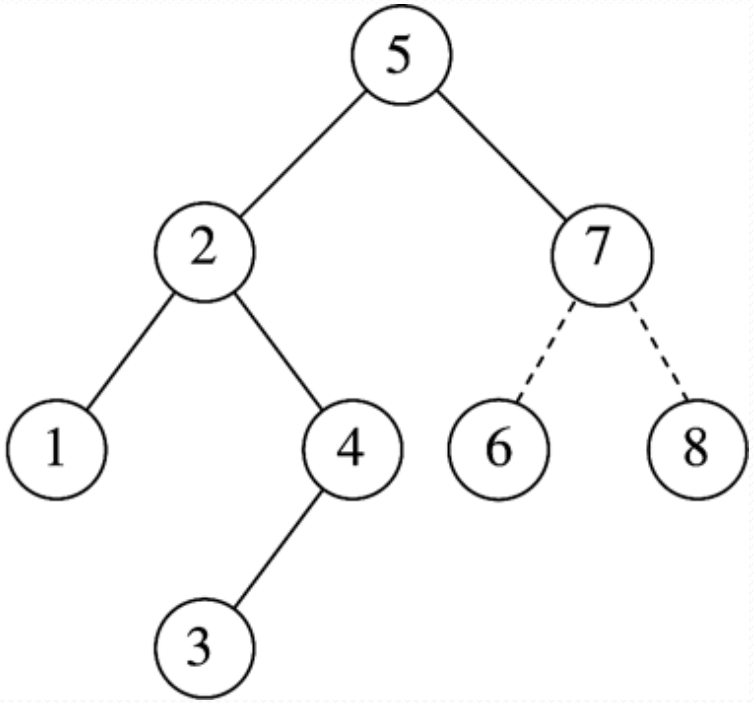




Example, case 1 (insert 6)

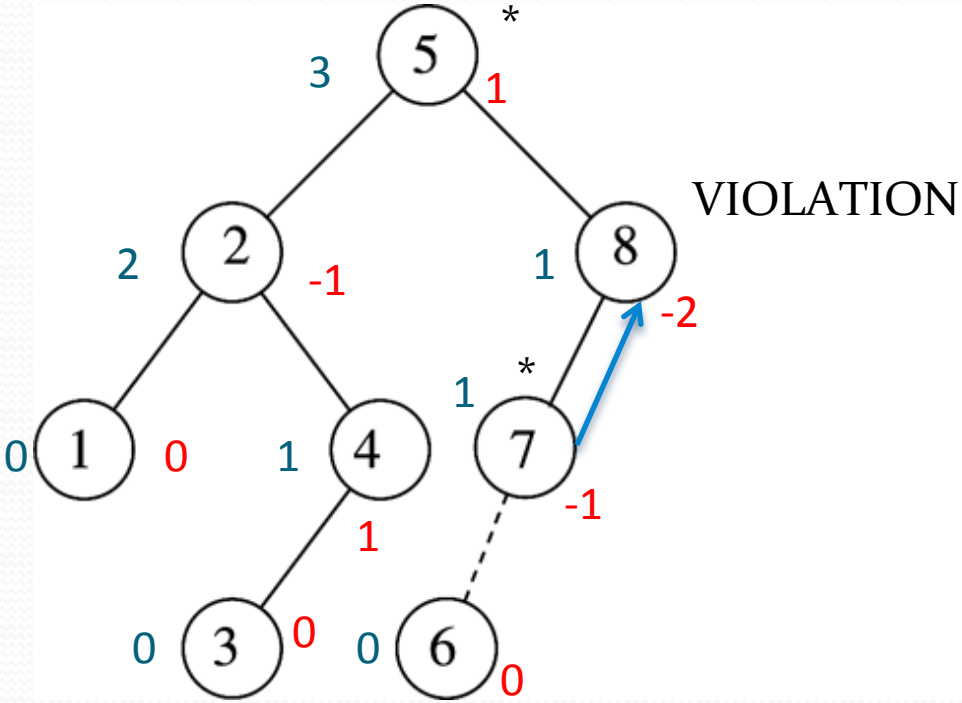
WALK IN OPPOSITE DIRECTION – UPDATE HEIGHTS AND CHECK FOR VIOLATIONS



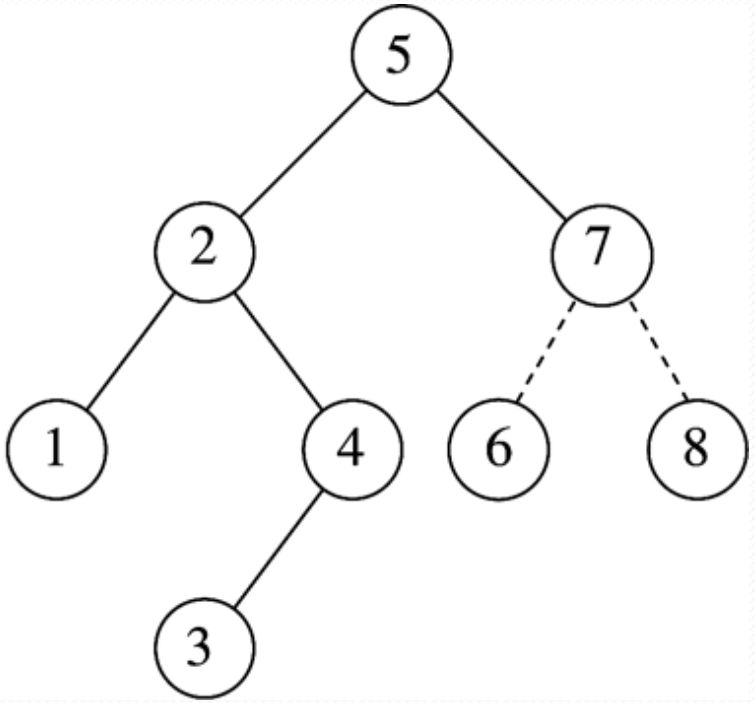


Example, case 1 (insert 6)

WALK IN OPPOSITE DIRECTION – UPDATE HEIGHTS AND CHECK FOR VIOLATIONS

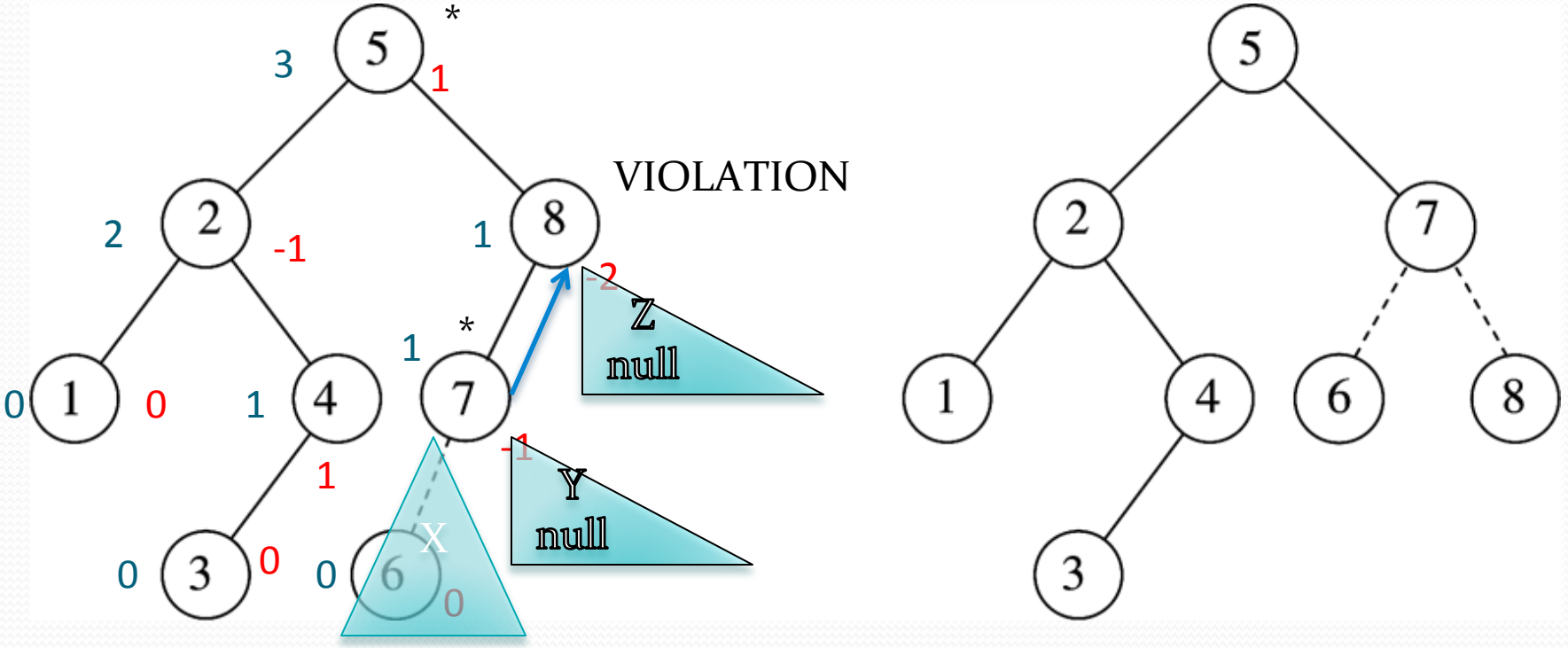


VIOLATION



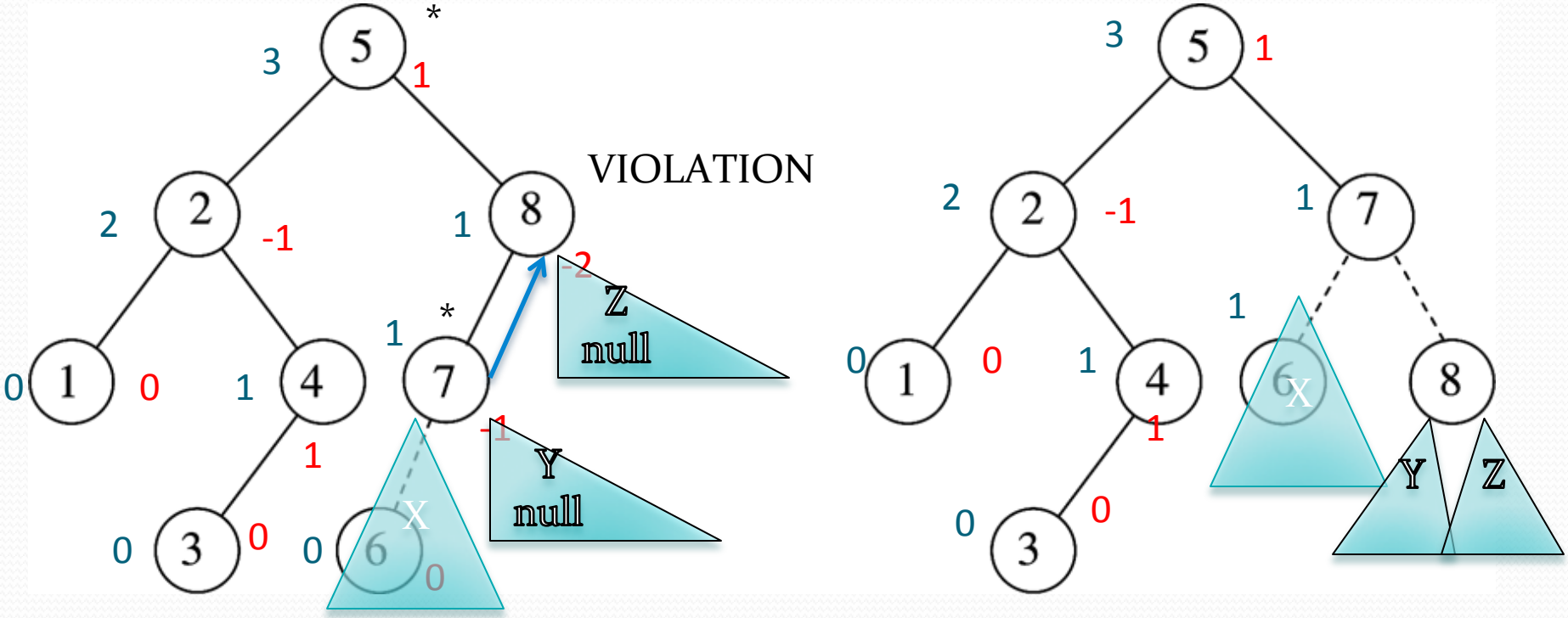
Example, case 1 (insert 6)

WALK IN OPPOSITE DIRECTION – UPDATE HEIGHTS AND CHECK FOR VIOLATIONS



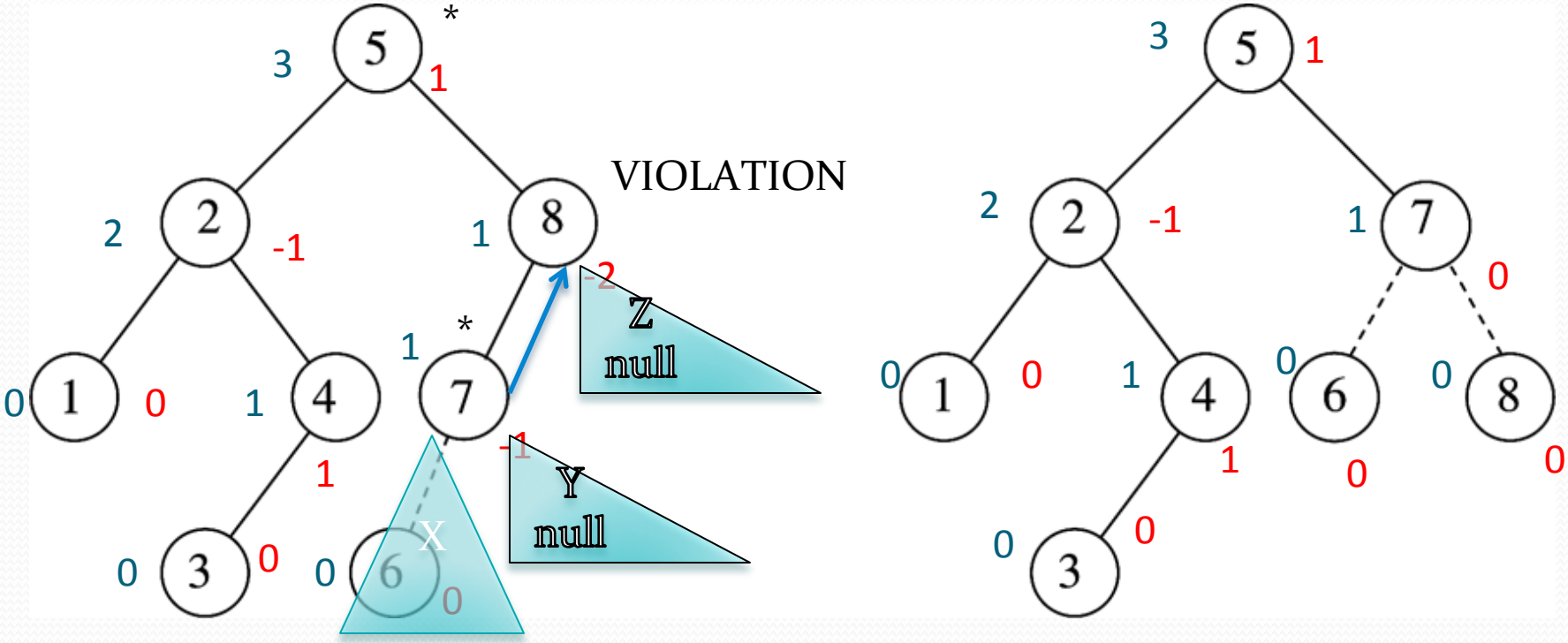
Example, case 1 -> single rotation

WALK IN OPPOSITE DIRECTION – UPDATE HEIGHTS AND CHECK FOR VIOLATIONS



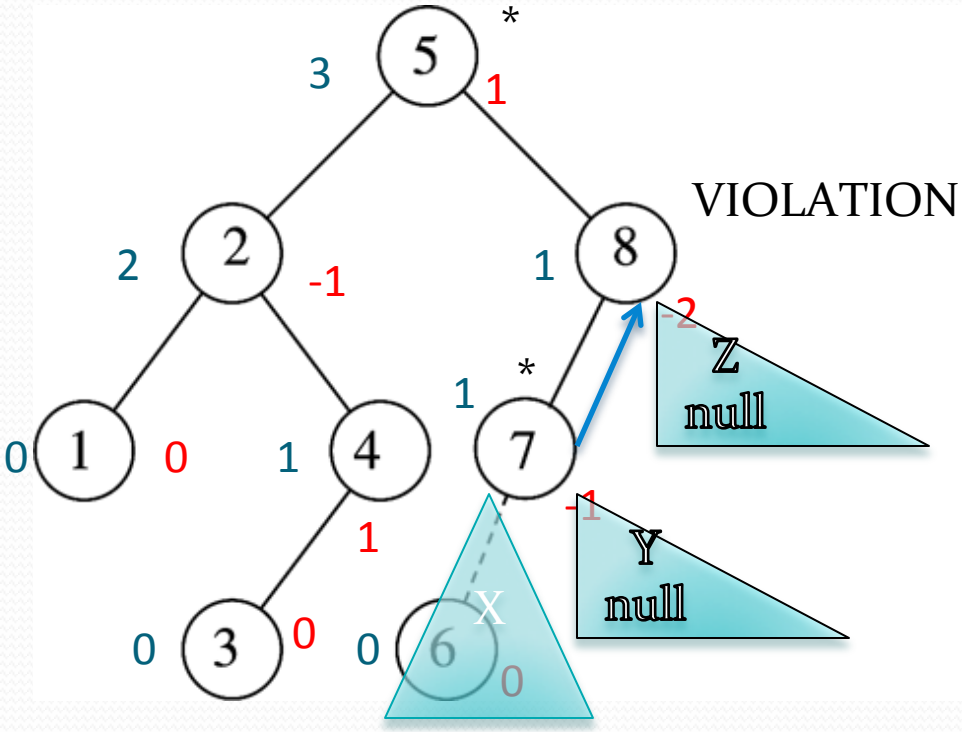
Example, case 1 -> single rotation

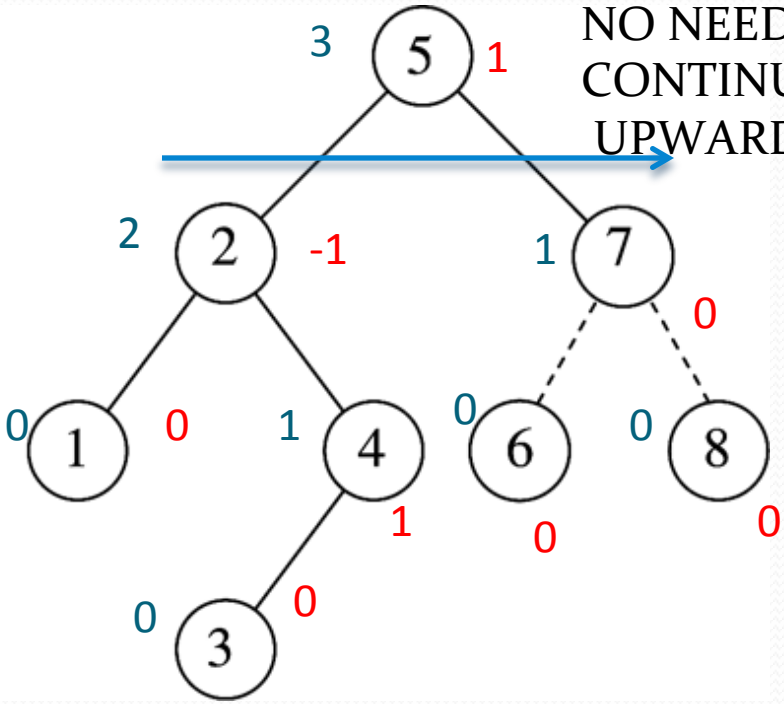
WALK IN OPPOSITE DIRECTION – UPDATE HEIGHTS AND CHECK FOR VIOLATIONS



Example, case 1 -> single rotation

WALK IN OPPOSITE DIRECTION – UPDATE HEIGHTS AND CHECK FOR VIOLATIONS



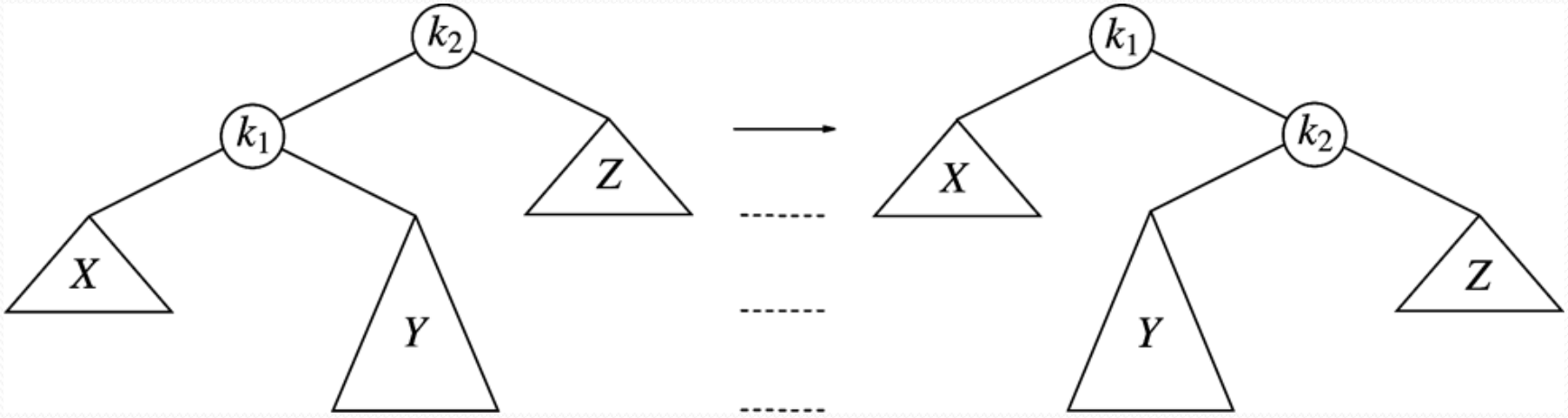


Single rotation example

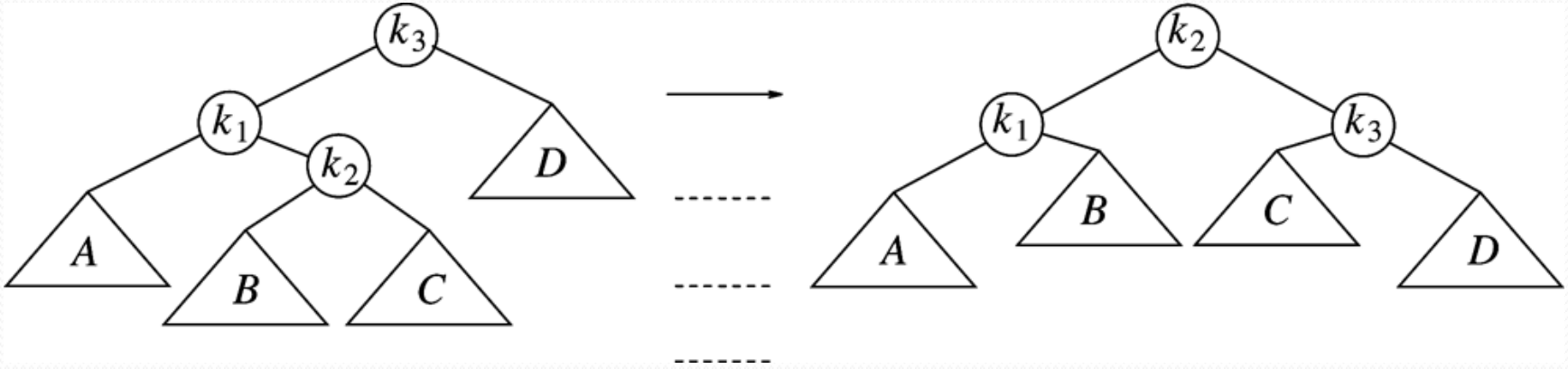
- Insert: 3,2,1
- Insert: 4,5,6,7
- AVL applet (just Google “avl applet”):

`https://www.cs.usfca.edu/~galles/visualization/AVLtree.html`

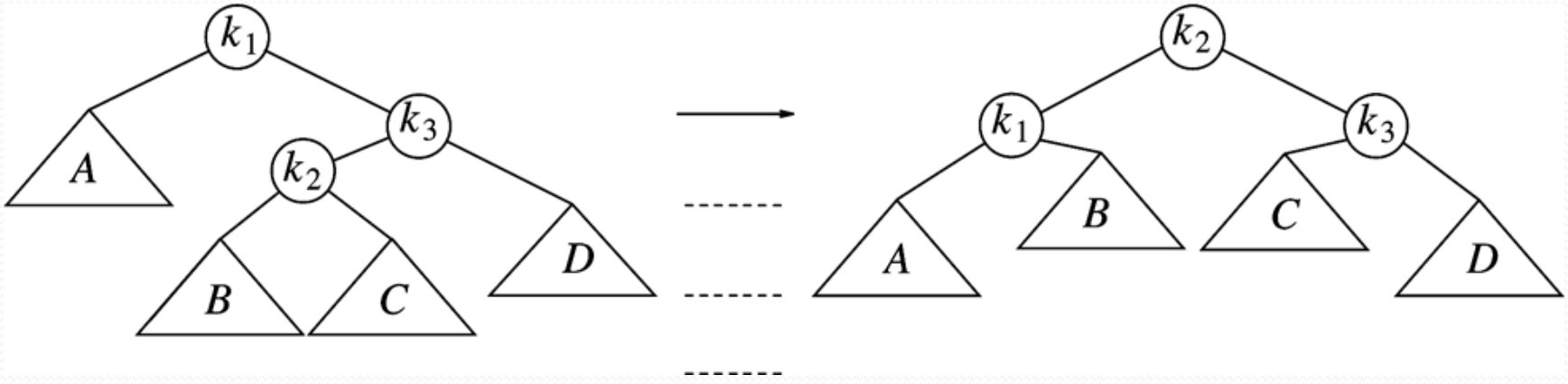
Single rotation fails in this case

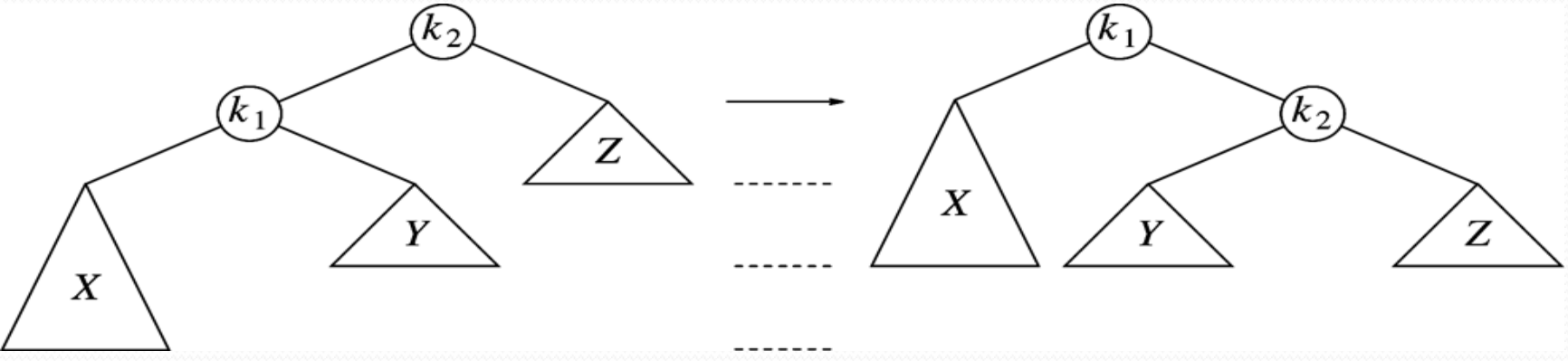


Double rotation (left-right) (case 2)



Double rotation (right-left) (case 3)

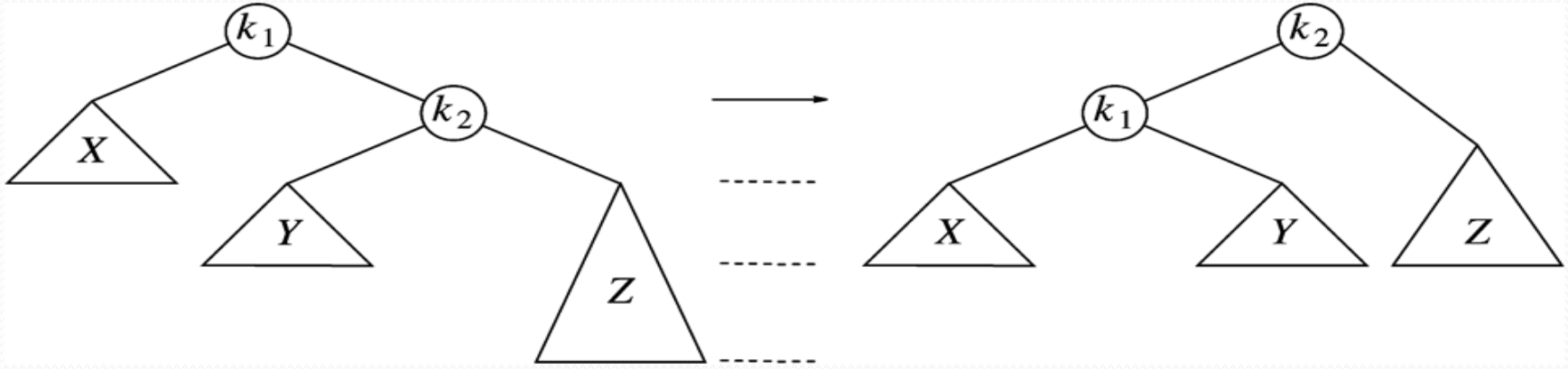


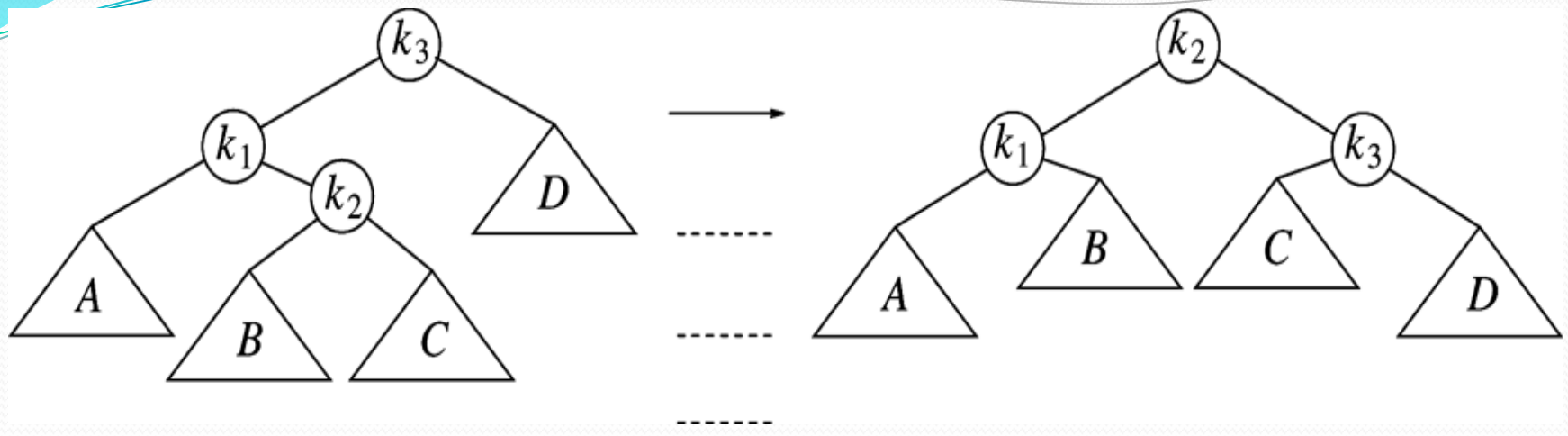


Case 1 (single)

k1 < k2

Case 4 (single)

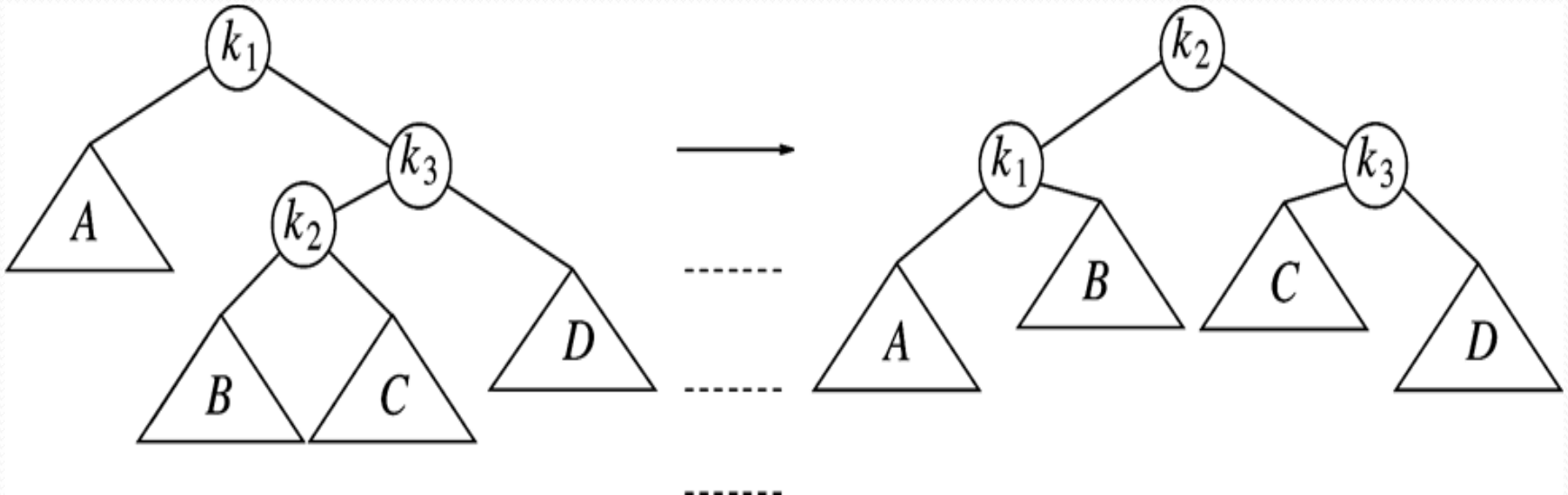


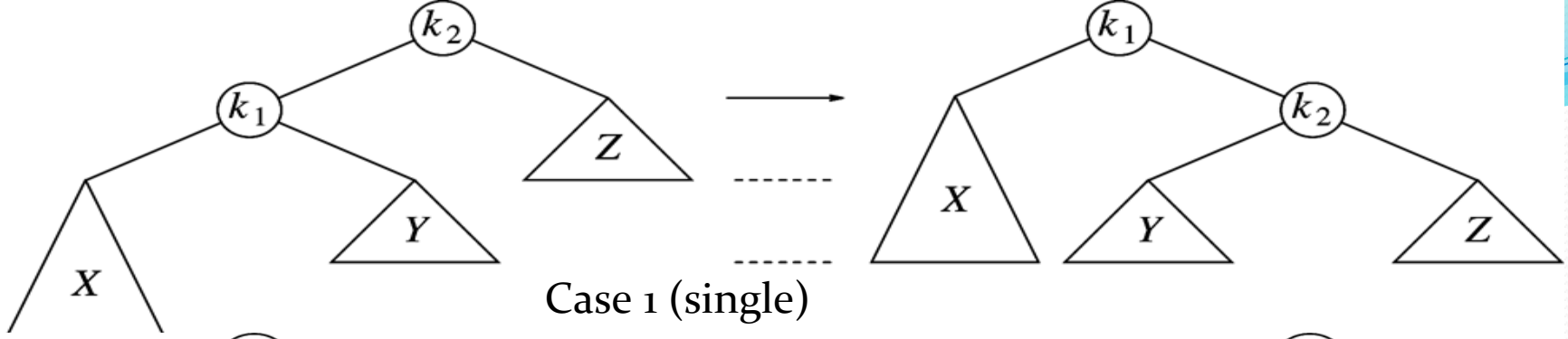


Case 2 (double)

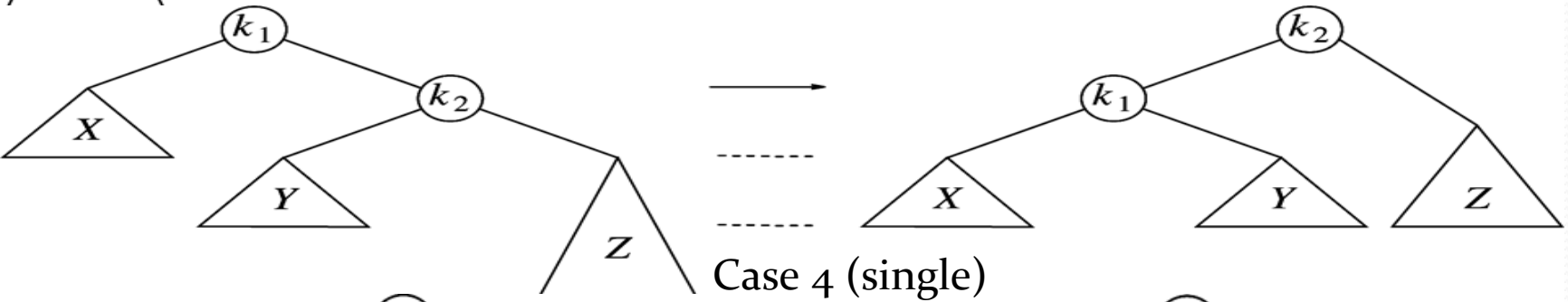
k1 < k2 < k3

Case 3 (double)

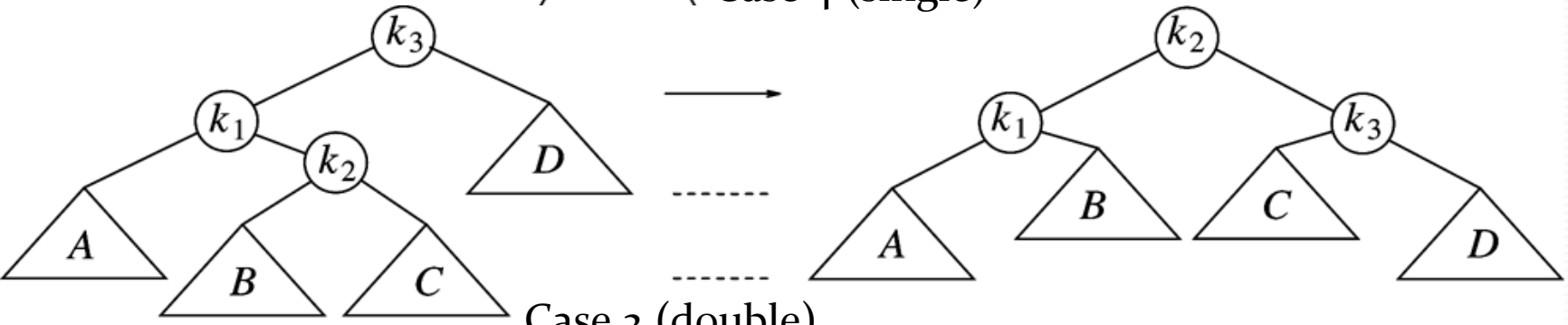




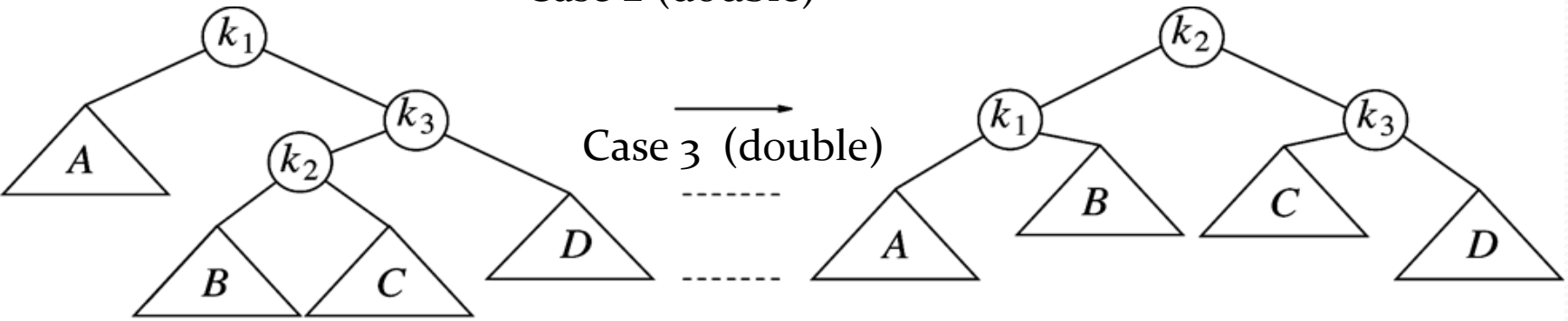
Case 1 (single)



Case 4 (single)



Case 2 (double)



Case 3 (double)

Example

- Continue by inserting: 16, 15, 14, 13, 12, 11, 10
- Insert 8,9.

AVL Node

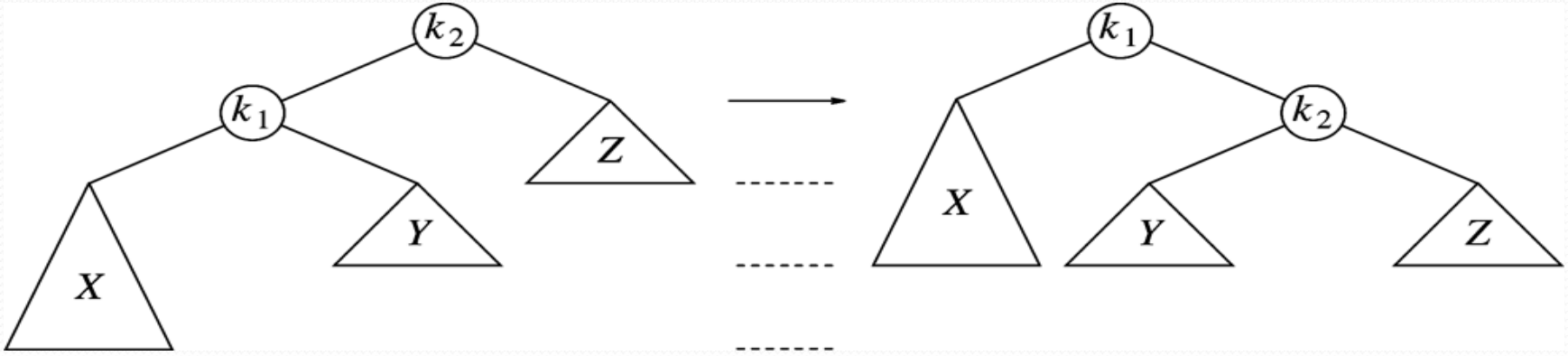
```
// AVL Tree implementation
// Usage: AvlTree<int> a_tree;
//        a_tree.Insert(10);

template <typename Comparable>
Class AvlTree {
 public: // … Big five.
 private:
   struct AvlNode {
       Comparable element_;
       AvlNode *left_;
       AvlNode *right_;
       int height_;
       AvlNode(const Comparable &the_element, AvlNode *lt, AvlNode *rt, int h = 0):
             element_{the_element}, left_{lt}, right_{rt}, height_{h} { }
       AvlNode(Comparable &&the_element, AvlNode *lt, AvlNode *rt, int h = 0):
             element_{std::move(the_element)}, left_{lt}, right_{rt}, height_{h} { }
    };
    AvlNode *root_;

    // Returns height of subtree with root t.
    int height(const AvlNode *t) const {
      return t == nullptr ? -1: t->height_;
    }
};
```

```
// Internal method to insert into an AVL subtree.
// x is the item to insert.
// t is the node that roots the subtree.
// t's height maybe updated. If there is a violation at t, a rotation (single or double)
// will be performed.
void Insert(const Comparable &x, AvlNode * &t) {
     if (t == nullptr) {
          t = new AvlNode{x, nullptr, nullptr, 0};
     } else if (x < t->element) {
          Insert(x, t->left);
          if (height(t->left_) - height(t->right_) == 2) {
            if (x < t->left_->element_)
               RotateWithLeftChild(t);
            else
               DoubleWithLeftChild(t);
           }
      } else if (t->element_ < x) {
           insert(x, t->right_);
           if (height(t->right_) - height(t->left_) == 2) {
             if (t->right->element_ < x)
                RotateWithRightChild(t);
              else
                DoubleWithRightChild(t);
           }
      } else { }  // Duplicate; do nothing
      t->height_ = Max(height(t->left_), height(t->right_)) + 1;
}
```

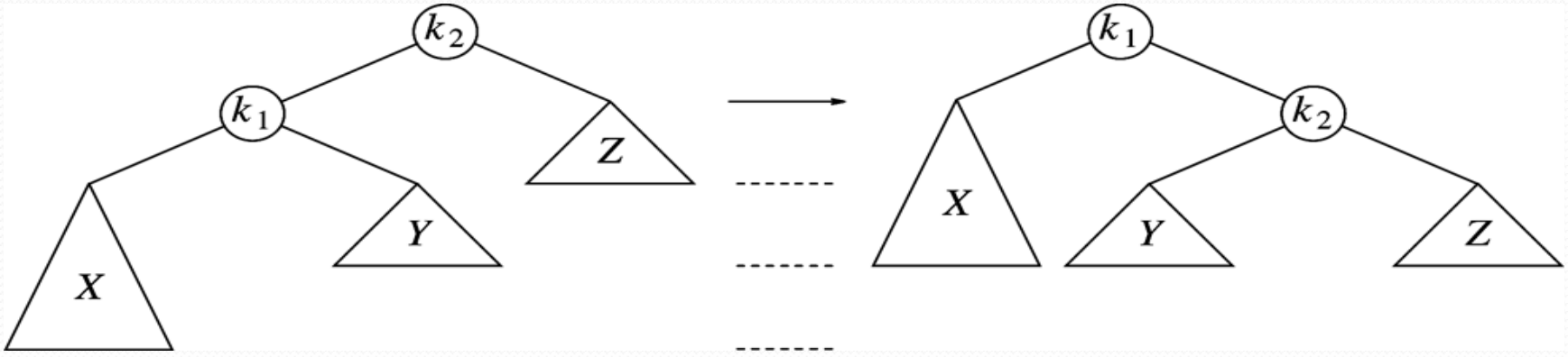
```
// ALTERNATIVE insert.
Void Insert(const Comparable &x, AvlNode * & t) {
    if (t == nullptr)
       t = new AvlNode{x, nullptr, nullptr, 0};
    else if (x < t->element_)
       Insert(x, t->left_);
    else if (t->element_ < x)
       Insert(x, t->right_);

    balance(t);
}
```

```
// Assume t is balanced or within one of being balanced. If node t is not balanced, AVL
// rotations will restore balance.
void balance(AvlNode *&t) {
    if (t == nullptr)
        return;
    if (height(t->left_) - height(t->right_) > 1) {
        if (height(t->left_->left_) >= height(t->left_->right_))
            RotateWithLeftChild(t);
        else
            DoubleWithLeftChild(t);
    } else if (height(t->right_) - height(t->left_) > 1) {
       if (height(t->right_->right_) >= height(t->right_->left_))
            rotateWithRightChild(t);
        else
            doubleWithRightChild(t);
    }

    t->height_ = Max(height(t->left_), height(t->right_)) + 1;
}
```

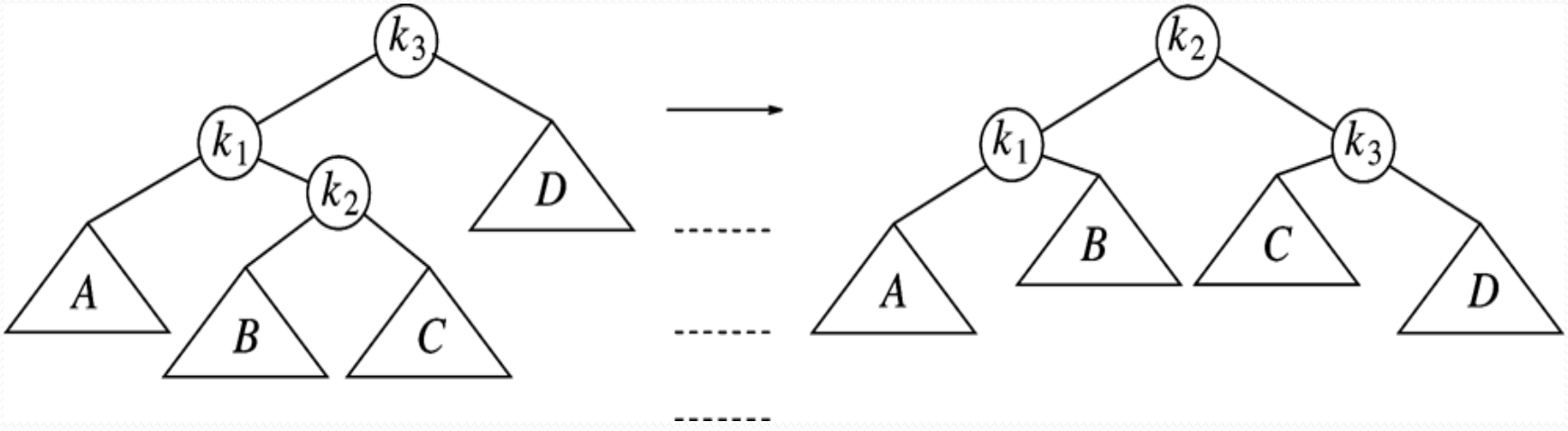
```
// Rotate binary tree node with left child.
// For AVL trees, this is a single rotation for case 1.
// Update heights, then set new root.
void RotateWithLeftChild(AvlNode * &k2) {



}
```



```
// Rotate binary tree node with left child.
// For AVL trees, this is a single rotation for case 1.
// Update heights, then set new root.
Void RotateWithLeftChild(AvlNode * &k2) {
        AvlNode *k1 = k2->left_;
        k2->left_ = k1->right_;
        k1->right_ = k2;
        k2->height_ = Max(height(k2->left_), height(k2->right_)) + 1;
        k1->height_ = Max(height(k1->left_), k2->height_) + 1;
        k2 = k1;
}
```



```
 // Double rotate binary tree node: first left child.
 // with its right child; then node k3 with new left child.
 // For AVL trees, this is a double rotation for case 2.
 // Update heights, then set new root.
void DoubleWithLeftChild(AvlNode * &k3) {
        RotateWithRightChild(k3->left_);
        RotateWithLeftChild(k3);
}
```



AVL Tree Deletion

- What could happen to the tree balance when we perform a remove?
- Can we simply add balance(t) to the BST implementation of remove?

AVL summary

- What is the cost of
  - Search
  - Insertion
  - Deletion

  in an AVL tree of N nodes?

AVL summary

- What is the cost of
  - Search
  - Insertion
  - Deletion

  in an AVL tree of N nodes?

$$\Theta(\log N)$$

AVL summary

- What is the cost of
  - Search
  - Insertion
  - Deletion

  Why? BECAUSE height h is **Θ(logN)**

  Search / Insert / Delete just go up / down a path from root to leaves, i.e. go up / down height steps.

Θ(logN)