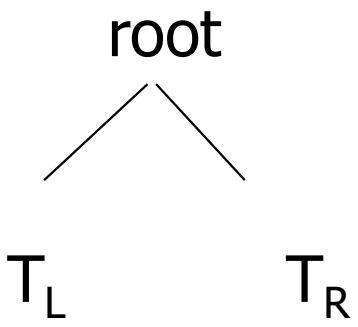
AVL Trees

Height of Binary Tree

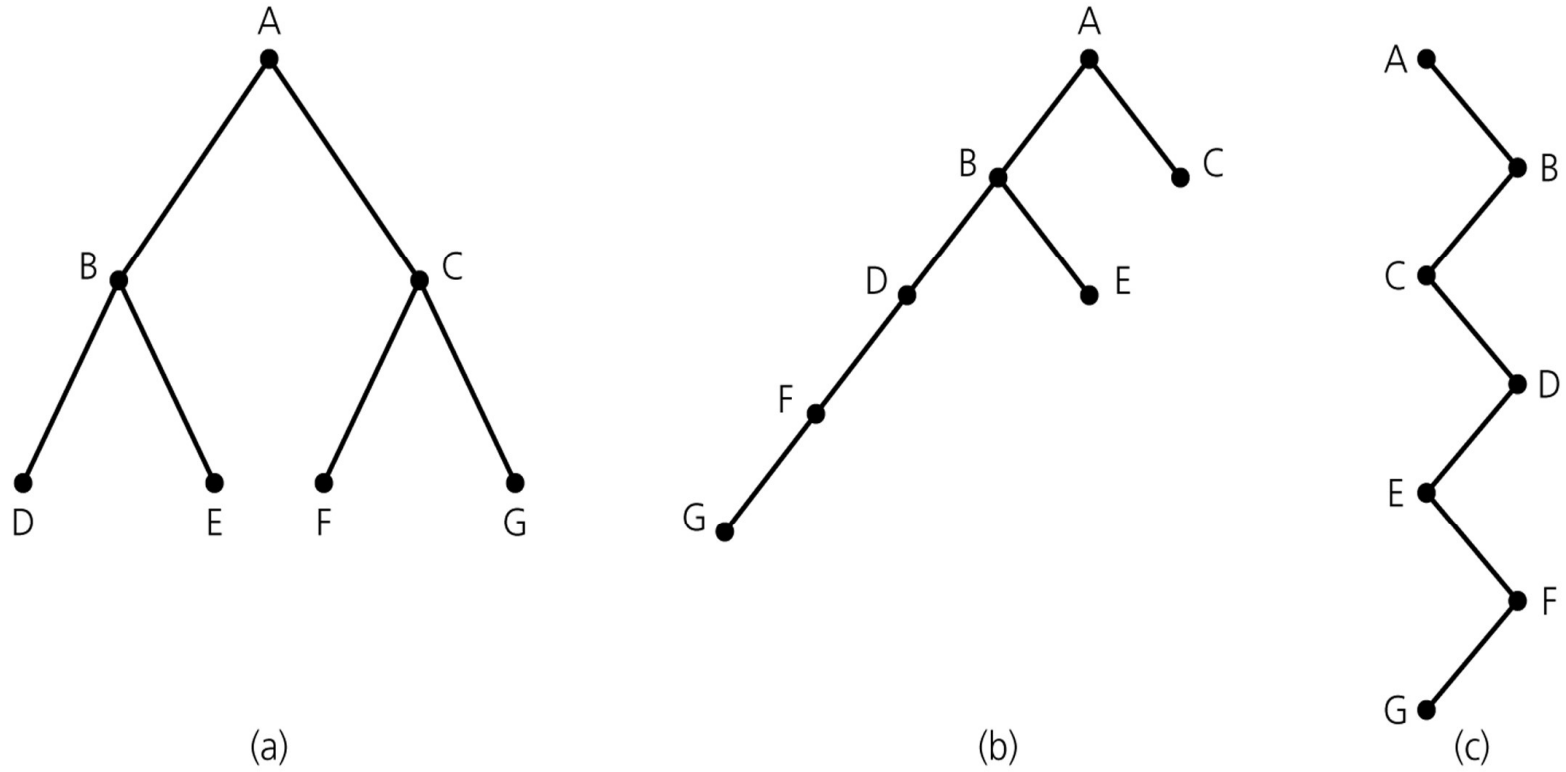
- The height of a binary tree T can be defined *recursively* as:
  - If T is empty, its height is -1.
  - If T is non-empty tree, then since T is of the form



the height of T is 1 greater than the height of its root's taller subtree; i.e.

$$\textbf{\textit{height}}(\textbf{\textit{T}}) = \textbf{\textit{1 + max}}\{\textit{height}(T_L), \textit{height}(T_R)\}$$

Height of Binary Tree (cont.)

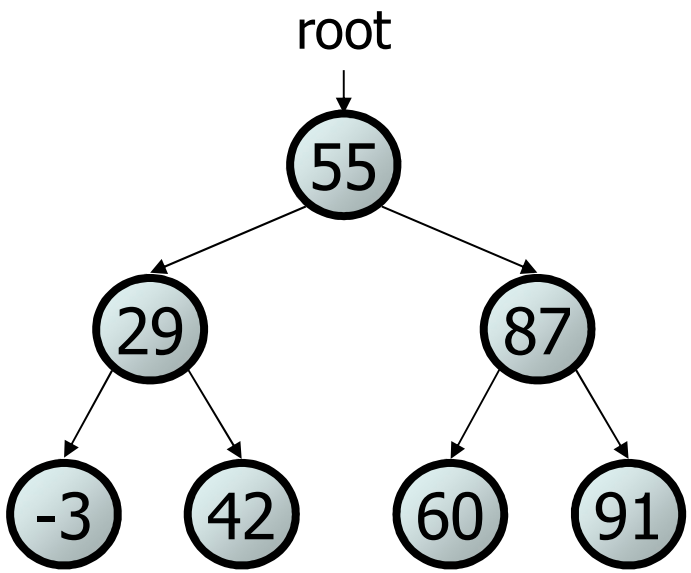


Binary trees with the same nodes but different heights

Binary Search Trees

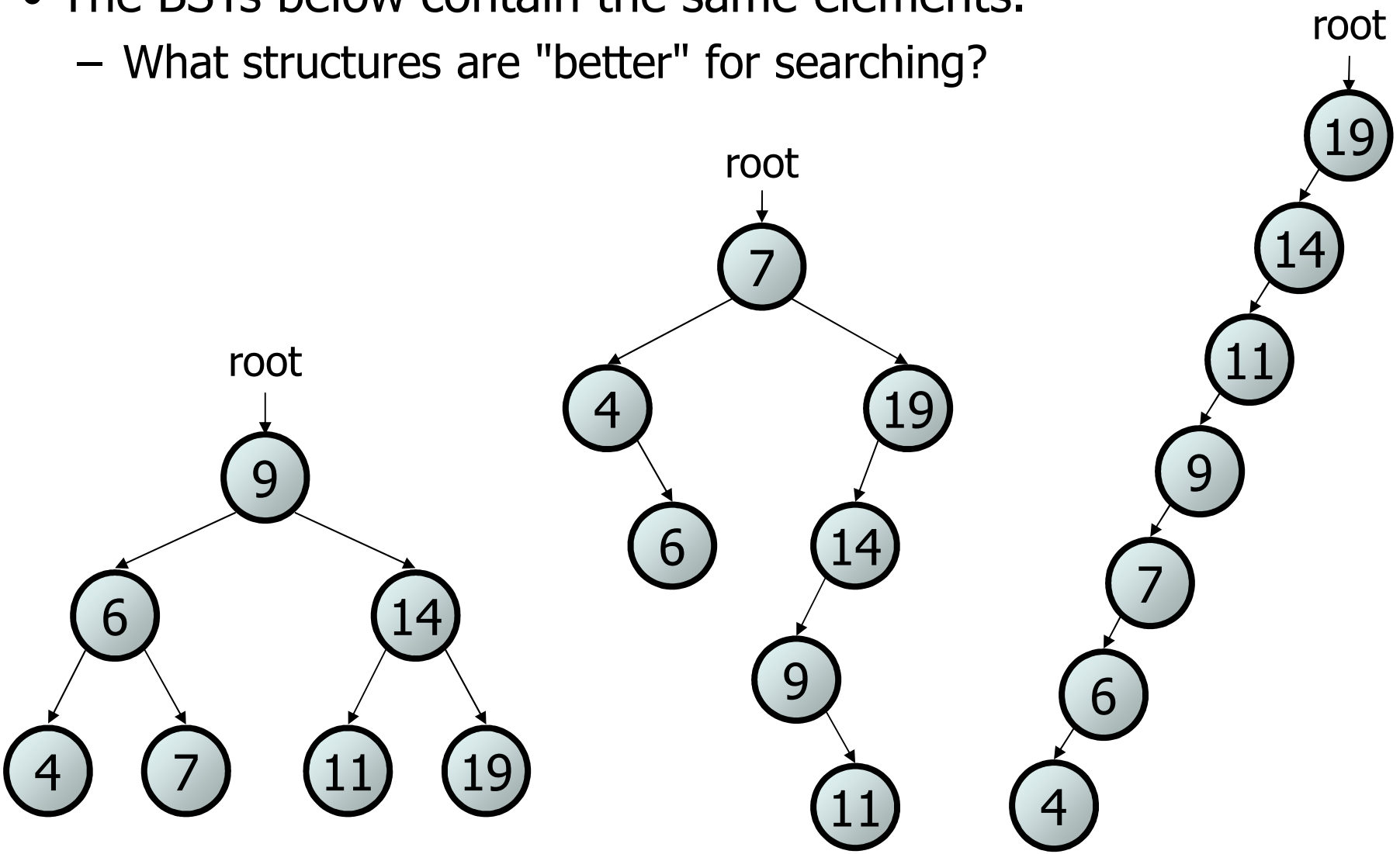
- **binary search tree** ("BST") is a binary tree that is either:
  - empty (`null`), or
  - a root node R such that:
    - every element of R's left subtree contains data "less than" R's data,
    - every element of R's right subtree contains data "greater than" R's,
    - R's left and right subtrees are also binary search trees.

- BSTs store their elements in sorted order, which is helpful for searching/sorting tasks.



Searching BSTs

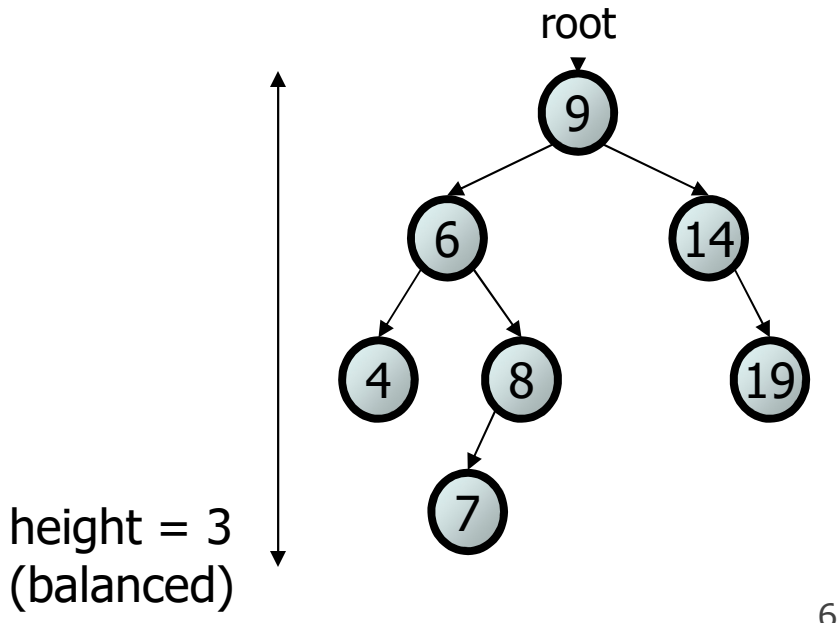
- The BSTs below contain the same elements.
  - What structures are "better" for searching?



Trees and balance

- **balanced tree**: One whose subtrees differ in height by at most 1 and are themselves balanced.
  - A balanced tree of N nodes has a height of ~ $\log_2$ N.
  - A very unbalanced tree can have a height close to N.

  - The runtime of adding to / searching a BST is closely related to height.



Analysis of tree operations

Operation	Average case	Worst case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)

The runtime of adding/deleting searching a BST is closely related to height.

AVL Trees

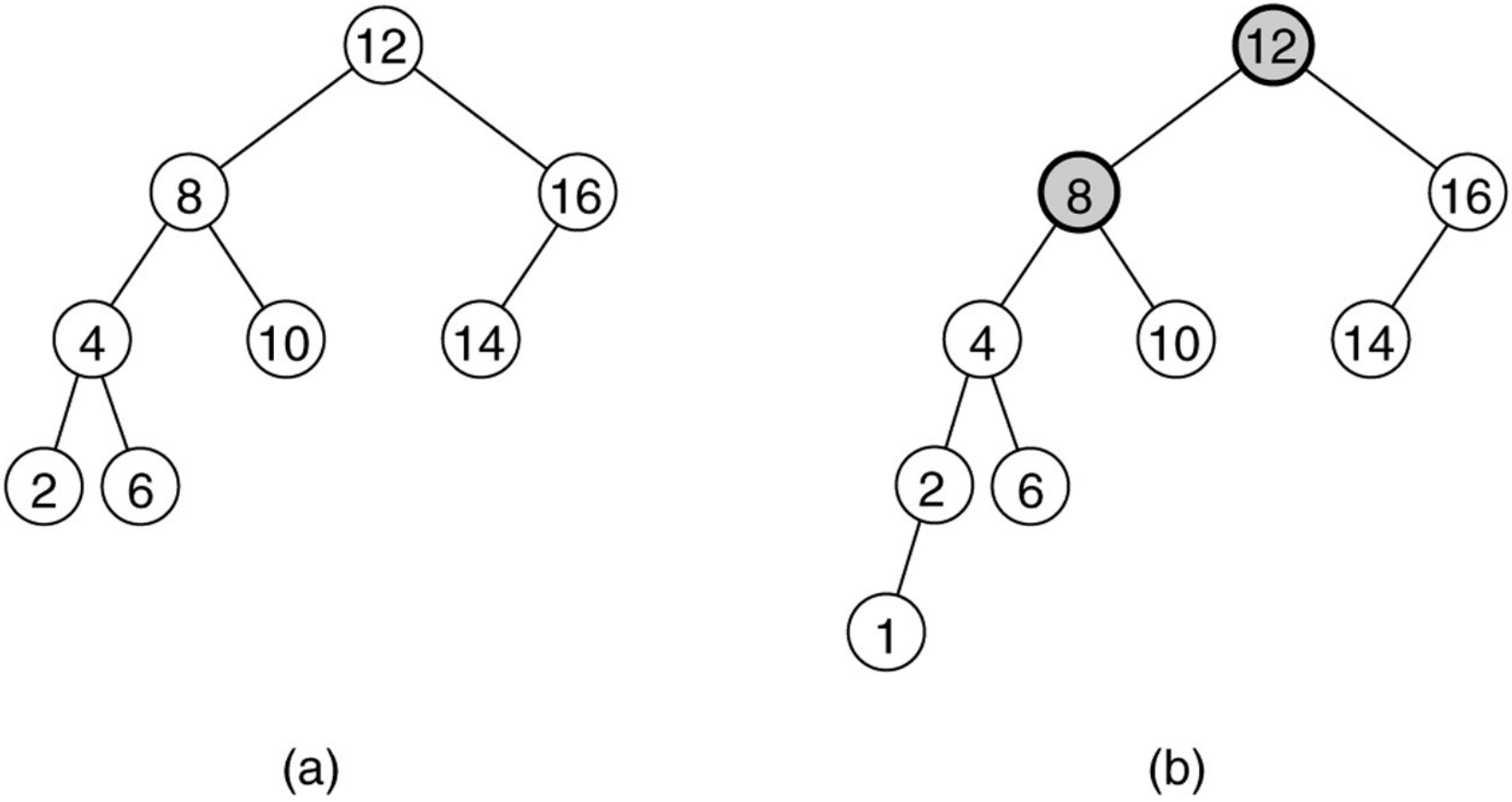
- An AVL tree is a binary search tree with a *balance* condition.
- AVL is named for its inventors: **A**del'son-**V**el'skii and **L**andis
- AVL tree *approximates* the ideal tree (completely balanced tree).
- AVL Tree maintains a height close to the minimum.

Definition:

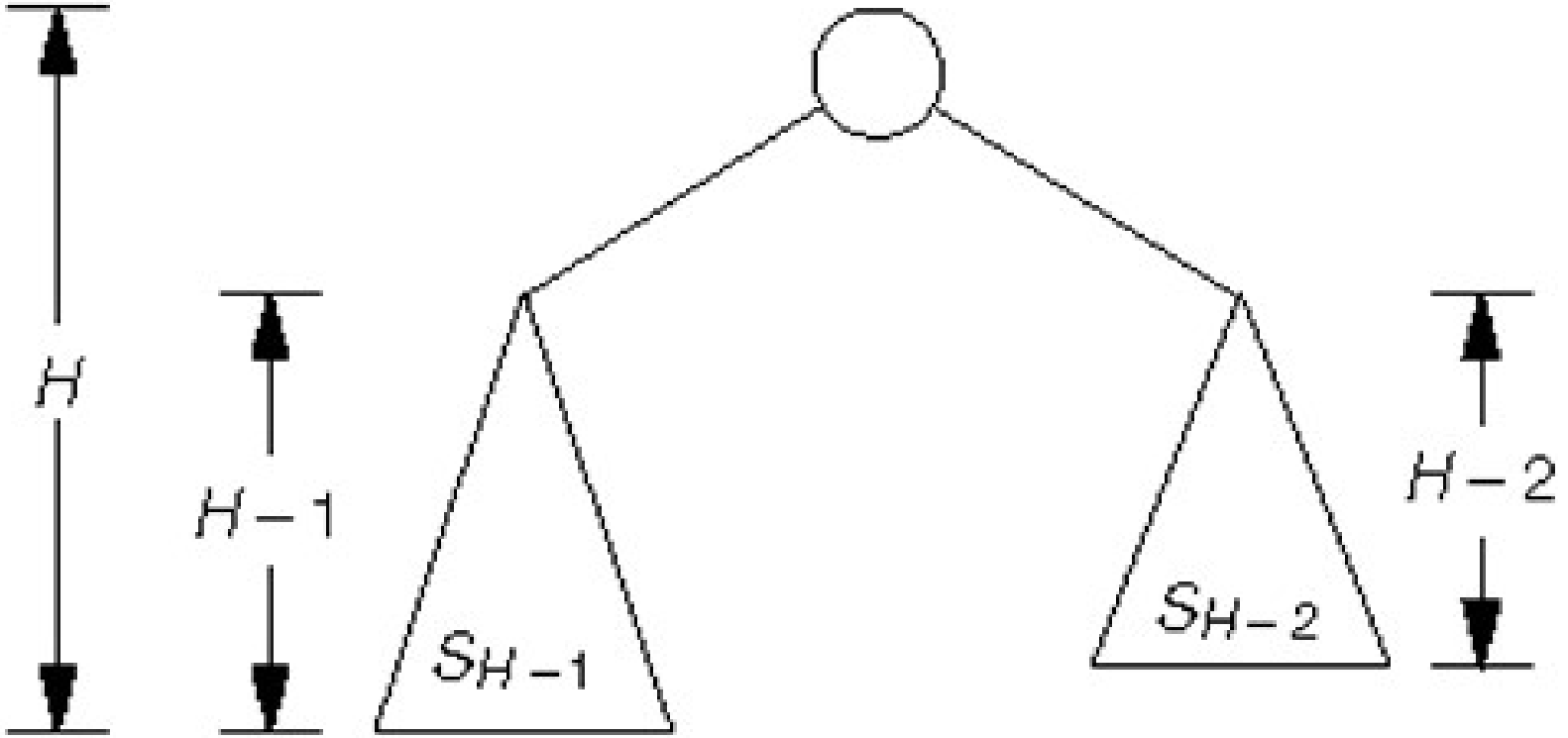
An AVL tree is a binary search tree such that for any node in the tree, the height of the left and right subtrees can differ by at most 1.

Example

Two binary search trees: (a) an AVL tree; (b) not an AVL tree (unbalanced nodes are darkened)



Minimum tree of height *H*



Properties

- The depth of a typical node in an AVL tree is very close to the optimal $log_2 N$.
  - Consequently, all searching operations in an AVL tree have logarithmic worst-case bounds.

- An update (insert or remove) in an AVL tree could destroy the balance. It must then be rebalanced before the operation can be considered complete.

- Key Property: After an insertion, only nodes that are on the path from the insertion point to the root can have their balances altered.

Rebalancing

Suppose the node to be rebalanced is X. There are 4 cases that we might have to fix (two are the mirror images of the other two):

1. An insertion in the left subtree of the left child of X,
2. An insertion in the right subtree of the left child of X,
3. An insertion in the left subtree of the right child of X, or
4. An insertion in the right subtree of the right child of X.

Balance is restored by tree ***rotations***.

Balancing Operations: Rotations

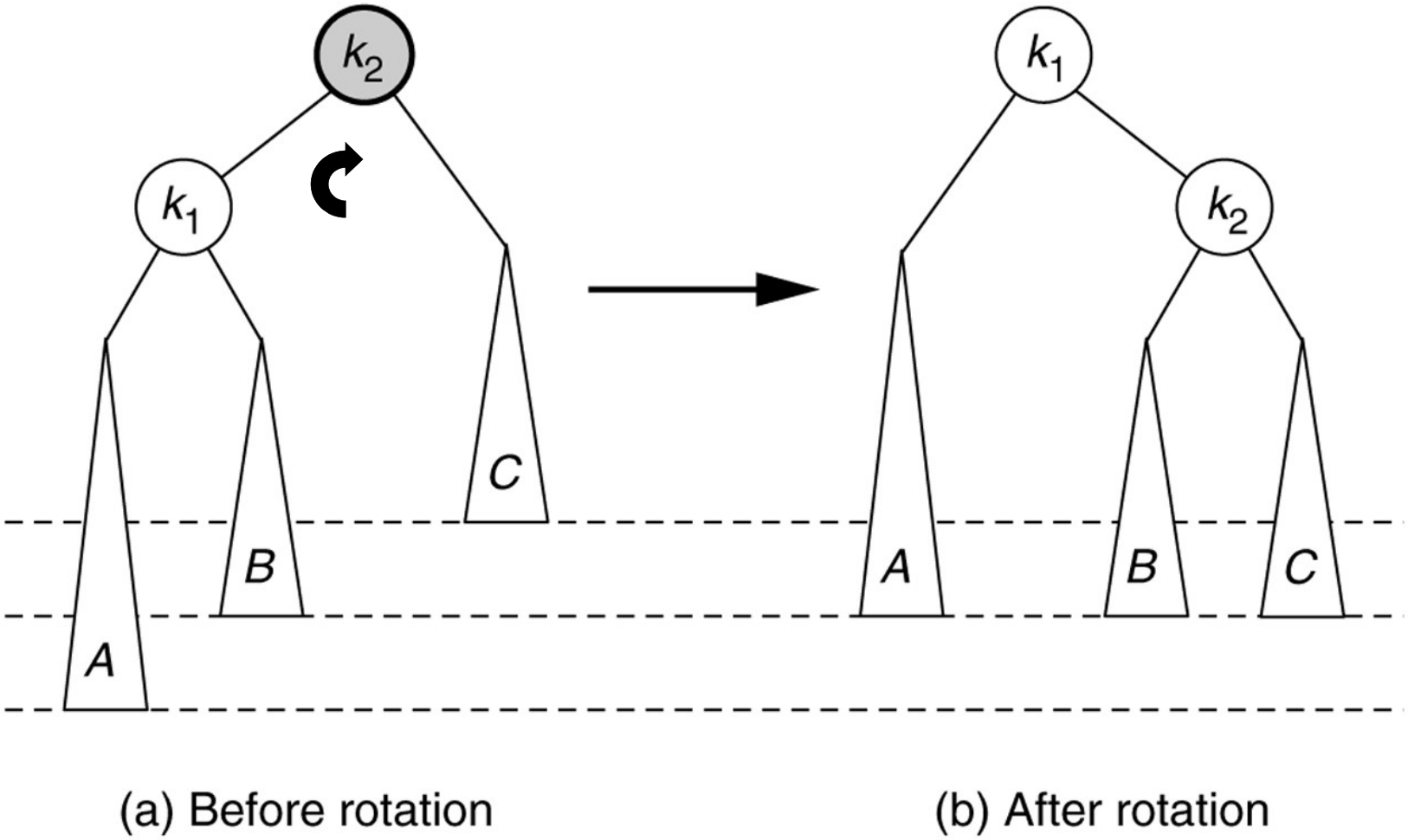
- Case 1 and case 4 are symmetric and requires the same operation for balance.
  - Cases 1,4 are handled by *single rotation*.

- Case 2 and case 3 are symmetric and requires the same operation for balance.
  - Cases 2,3 are handled by *double rotation*.

Single Rotation

- A *single rotation* switches the roles of the parent and child while maintaining the search order.
- We rotate between a node and its child.
  - Child becomes parent. Parent becomes right child in case 1, left child in case 4.
- The result is a binary search tree that satisfies the AVL property.

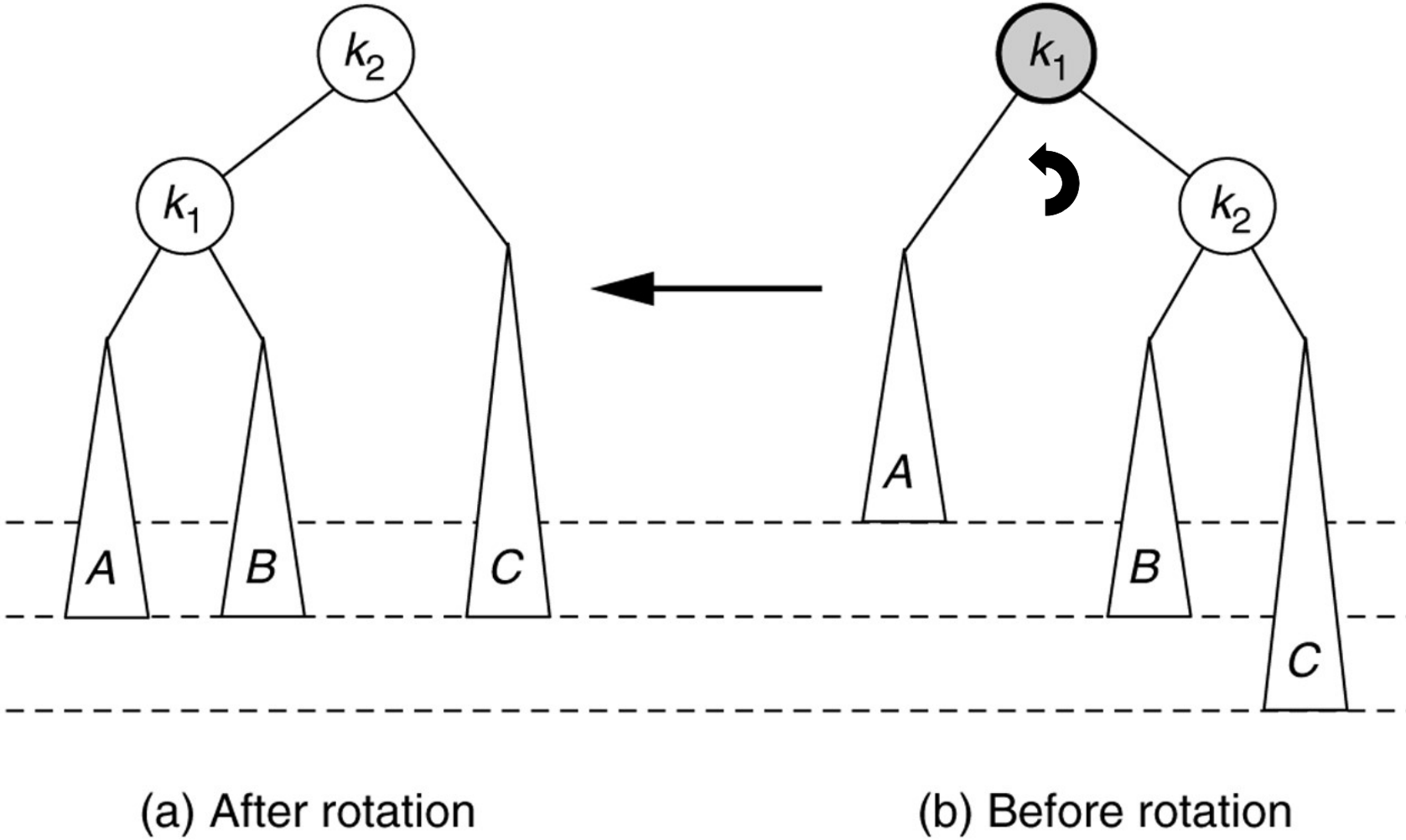
Single rotation to fix case 1: Rotate right



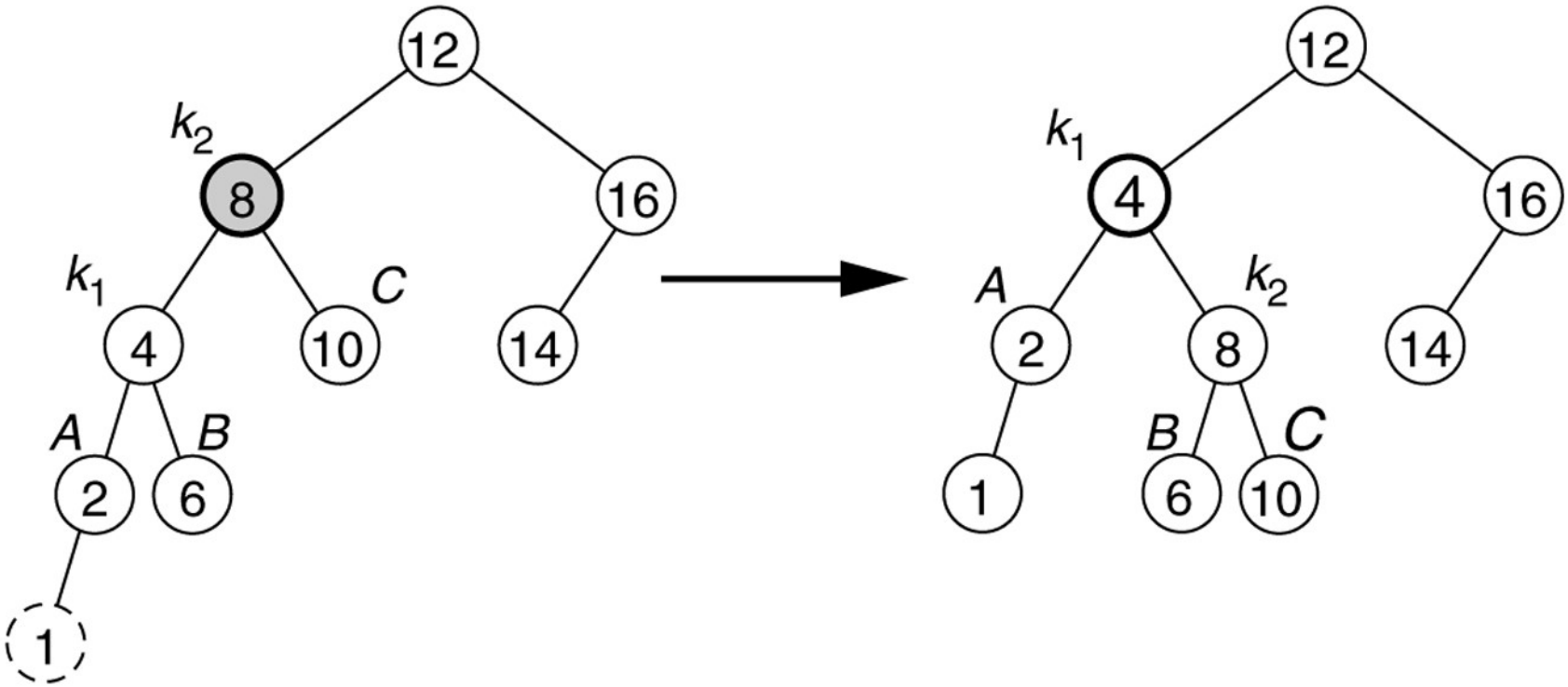
(a) Before rotation

(b) After rotation

Symmetric single rotation to fix case 4 : Rotate left



Single rotation fixes an AVL tree after insertion of 1.



(a) Before rotation

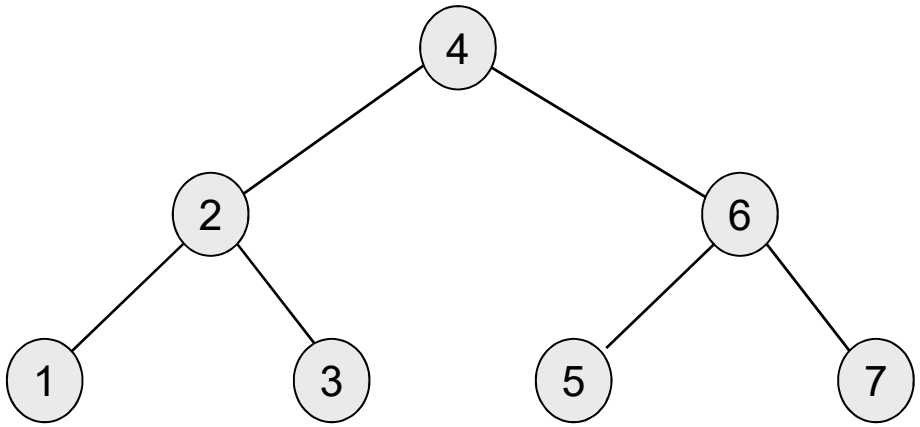
(b) After rotation

Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.
- Answer:

Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.
- Answer:



Analysis

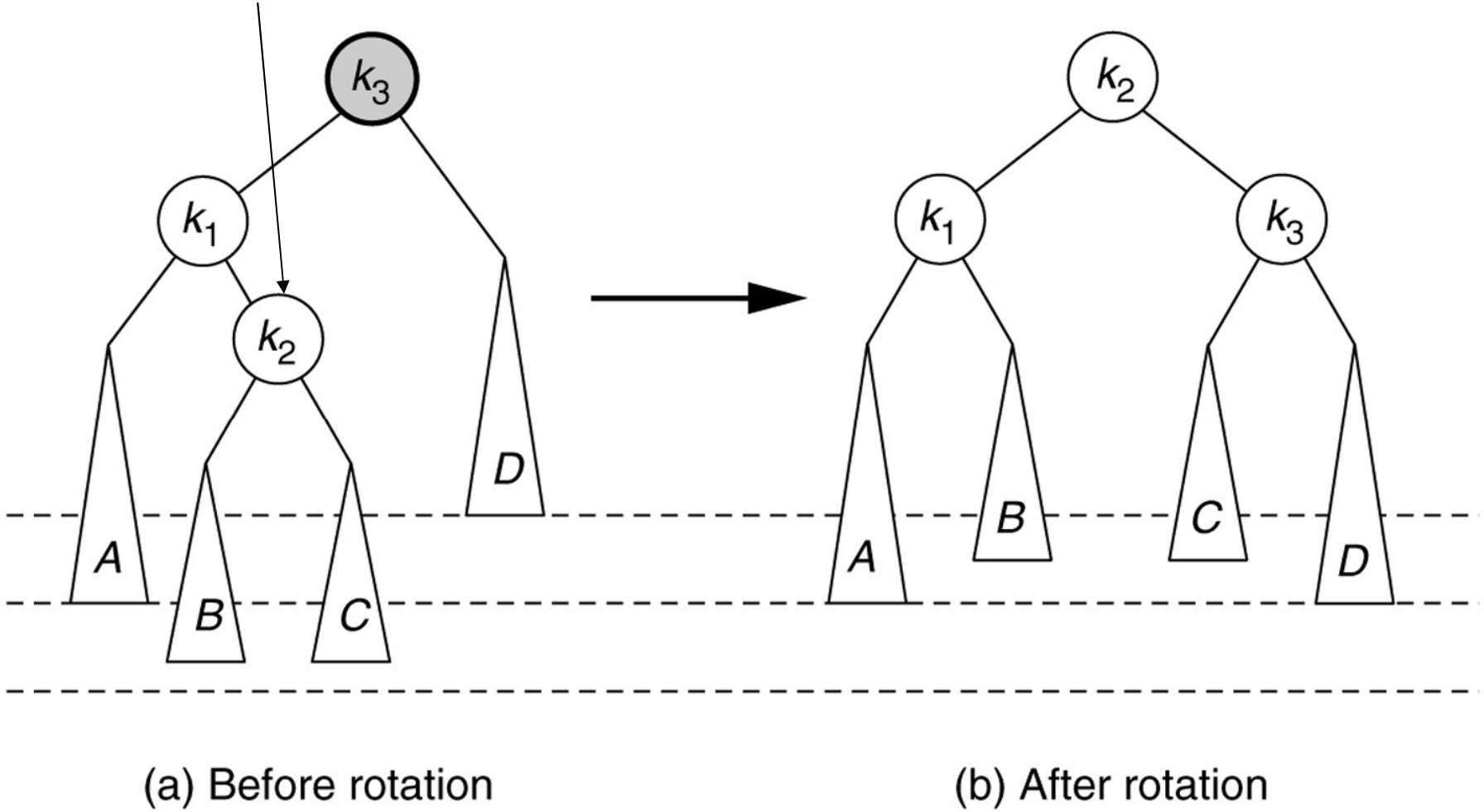
- One rotation suffices to fix cases 1 and 4.
- Single rotation preserves the original height:
  - The new height of the entire subtree is exactly the same as the height of the original subtree before the insertion.
- Therefore it is enough to do rotation only at the first node, where imbalance exists, on the path from inserted node to root.
- Thus the rotation takes O(1) time.
- Hence insertion is O(logN)

Double Rotation

- Single rotation does not fix the inside cases (2 and 3).
- These cases require a *double rotation*, involving three nodes and four subtrees.

Left–right double rotation to fix case 2

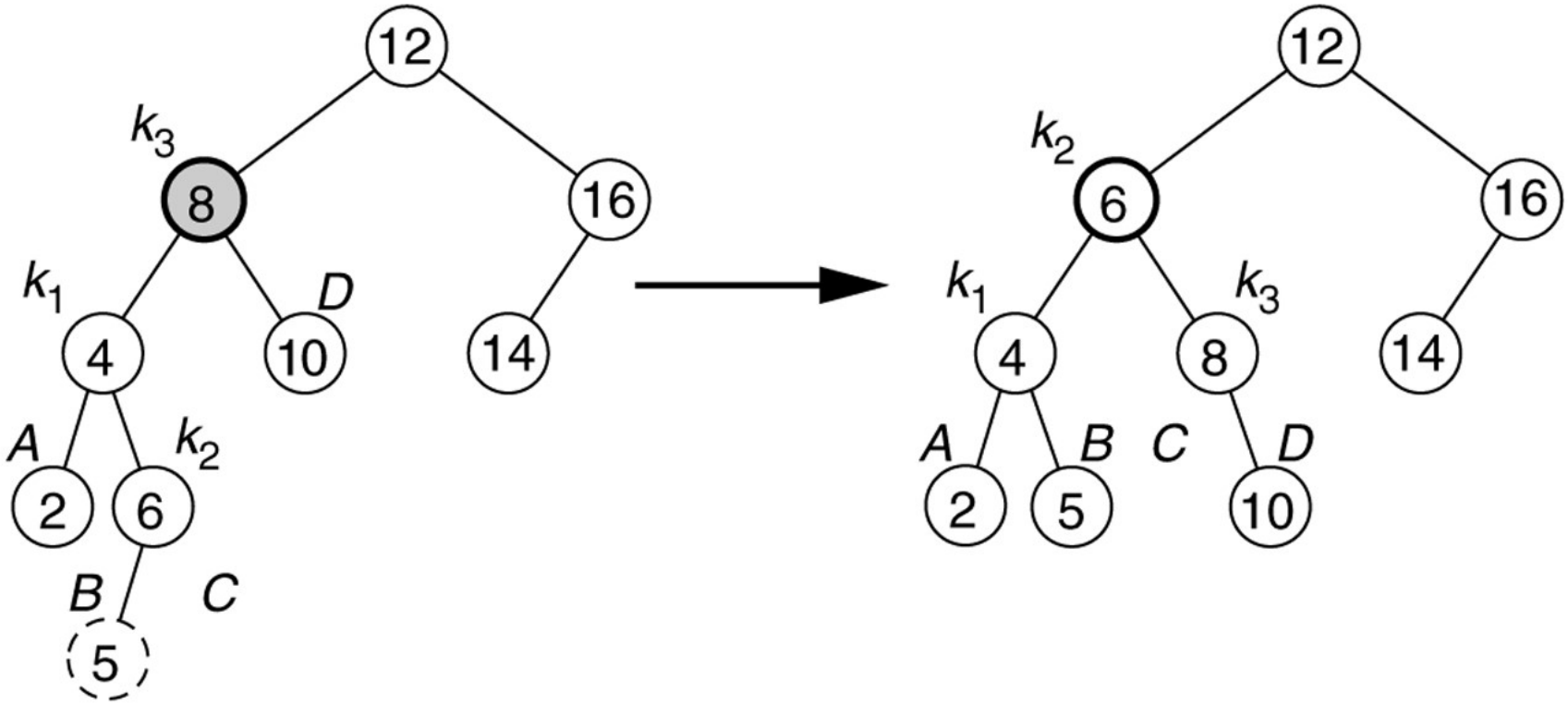
Lift this up:
first rotate left between (k_1,k_2),
then rotate right betwen (k_3,k_2)



Left-Right Double Rotation

- A left-right double rotation is equivalent to a sequence of two single rotations:
  - 1st rotation on the original tree:
    a *left* rotation between X's left-child and grandchild
  - 2nd rotation on the new tree:
    a *right* rotation between X and its new left child.

Double rotation fixes AVL tree after the insertion of 5

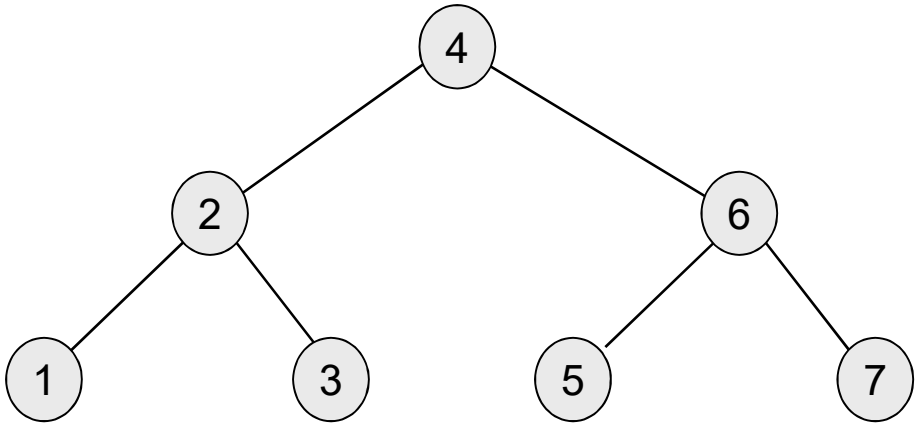


(a) Before rotation

(b) After rotation

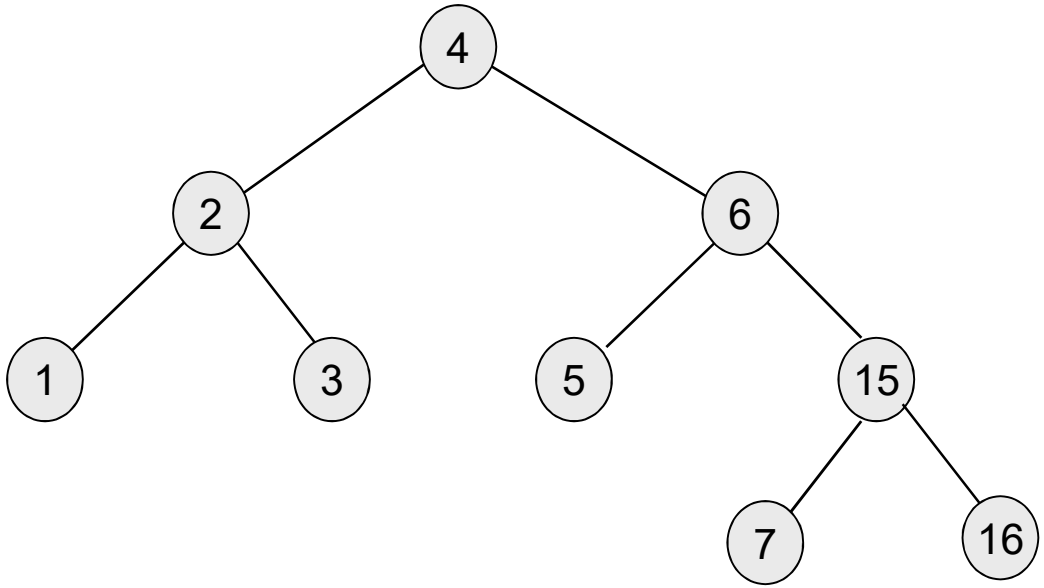
Example

- Insert 16, 15, 14, 13, 12, 11, 10, and 8, and 9 to the tree obtained in the previous example.



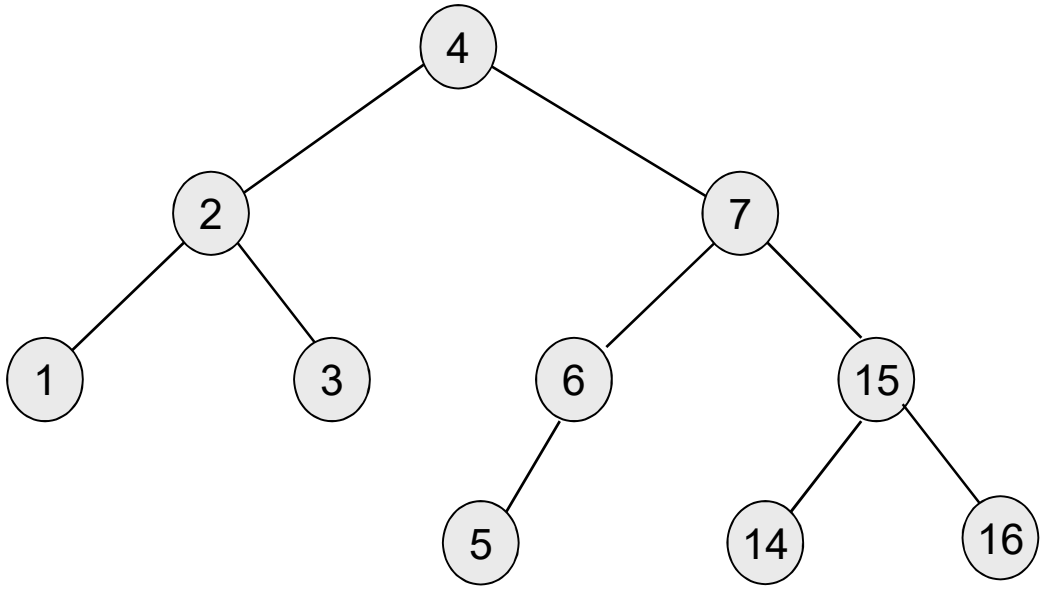
Example

- Insert 16, 15, 14, 13, 12, 11, 10, and 8, and 9 to the tree obtained in the previous example.



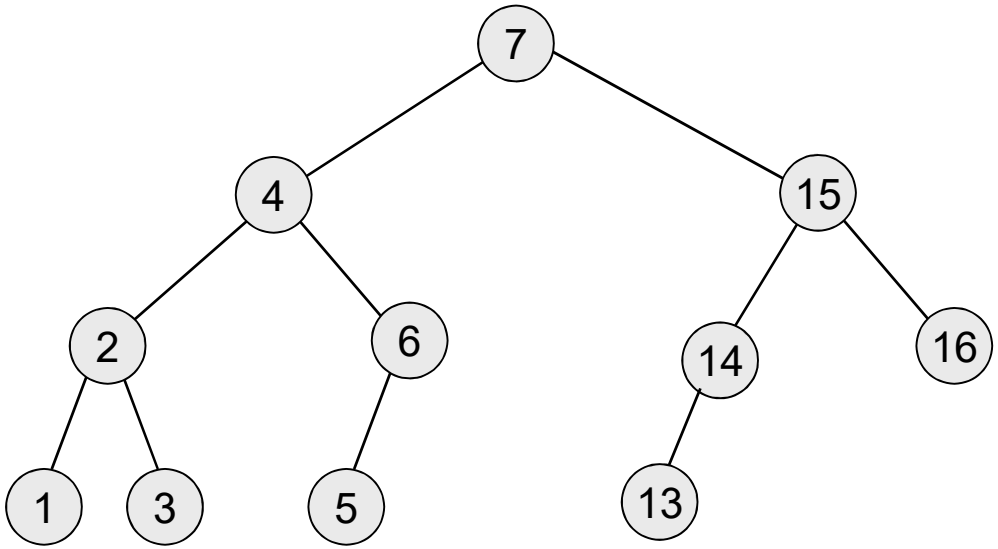
Example

- Insert 16, 15, 14, 13, 12, 11, 10, and 8, and 9 to the tree obtained in the previous example.

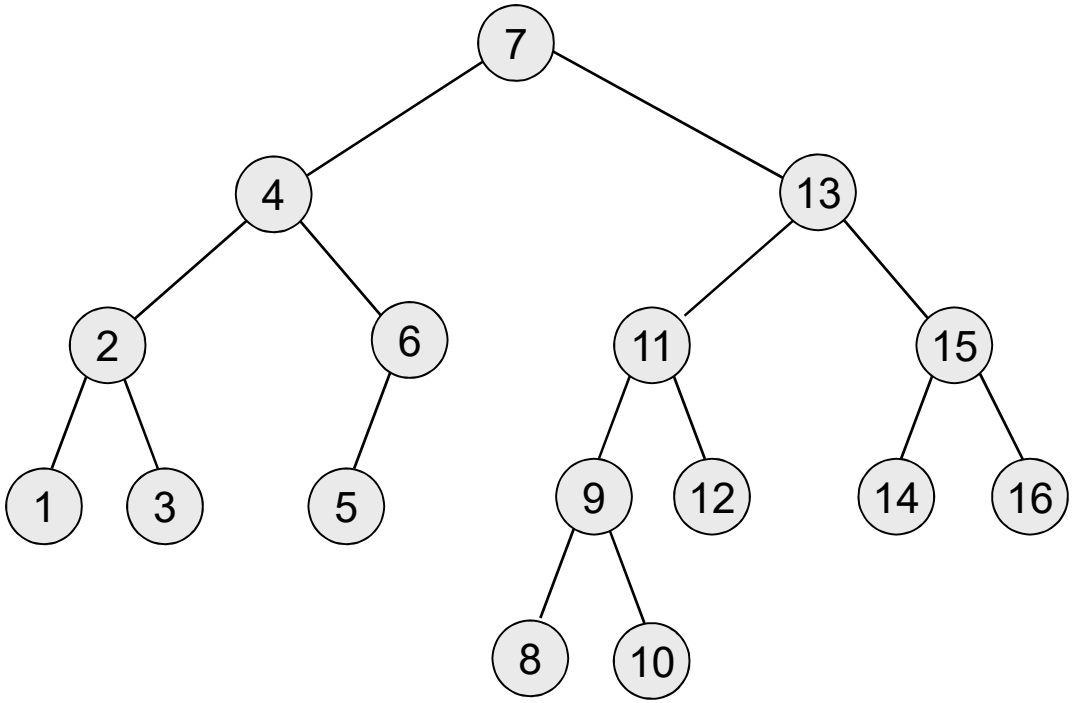


Example

- Insert 16, 15, 14, 13, 12, 11, 10, and 8, and 9 to the tree obtained in the previous example.



Solution



Node declaration for AVL trees

```
template <class T>
class AvlNode
{
   T element;
   AvlNode   *left;
   AvlNode   *right;
   int       height;

   AvlNode( const T & theElement, AvlNode *lt = NULL,
            AvlNode *rt = NULL, int h = 0 )
     : element( theElement ), left( lt ), right( rt ),
               height( h ) { }
};
```

Height

```
template class <T>
int height( const AvlNode<T> *t)
{
  return t == NULL ? -1 : t->height;
}
```

Single right rotation (Case 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.
 */
template <class T>
void rotateWithLeftChild( AvlNode<T> *& k2 )
{
    AvlNode<T> *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;
}
```

Single right rotation (Case 1)

```
template <class T>
void rotateWithLeftChild( AvlNode<T> *& k2 )
{
    AvlNode<T> *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;
}
```

Single left rotation (Case 4)

```
/**
 * Rotate binary tree node with right child.
 * For AVL trees, this is a single rotation for case 4.
 * Update heights, then set new root.
 */
template <class T>
void rotateWithRightChild( AvlNode<T> *& k2 )
{
    AvlNode<T> *k1 = k2->right;
    k2->right = k1->left;
    k1->left = k2;
    k2->height = max( height( k2->left ), height( k2->right))+1;
    k1->height = max( height( k1->right ), k2->height ) + 1;
    k2 = k1;
}
```

Double Rotation (Case 2)

```
/**
 * 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.
 */
template <class T>
void doubleWithLeftChild( AvlNode<T> *& k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```

Double Rotation (Case 2)

```
template <class T>
void doubleWithLeftChild( AvlNode<T> *& k3 )
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```

Double Rotation (Case 3)

```
/**
 * Double rotate binary tree node:
 * first right child with its left child; then node k3
 * with new right child.
 * For AVL trees, this is a double rotation for case 3.
 * Update heights, then set new root.
 */
template <class T>
void doubleWithRightChild( AvlNode<T> *& k3 )
{
    rotateWithLeftChild( k3->right );
    rotateWithRightChild( k3 );
}
```

```
/* Internal method to insert into a subtree.
 * x is the item to insert; t is the node that roots the tree.
 */
template <class T>
void insert( const T& x, AvlNode<T> *& t )
{
   if( t == NULL )
     t = new AvlNode<T>(x);
   else if( x < t->element )
   {
     insert( x, t->left );
     
   }
   else if( t->element < x )
   {
       insert( x, t->right );
     
    }
    else
       ;  // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

```
/* Internal method to insert into a subtree.
 * x is the item to insert; t is the node that roots the tree.
 */
template <class T>
void insert( const T& x, AvlNode<T> *& t )
{
    if( t == NULL )
      t = new AvlNode<T>(x);
    else if( x < t->element )
    {
      insert( x, t->left );
      if( height( t->left ) - height( t->right ) == 2 )
        if( x < t->left->element )
            rotateWithLeftChild( t );  // case 1
        else
            doubleWithLeftChild( t );  // case 2
    }
    else if( t->element < x )
    {
        insert( x, t->right );
     }
     else
        ;  // Duplicate; do nothing
     t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

```
/* Internal method to insert into a subtree.
 * x is the item to insert; t is the node that roots the tree.
 */
template <class T>
void insert( const T& x, AvlNode<T> *& t )
{
   if( t == NULL )
     t = new AvlNode<T>(x);
   else if( x < t->element )
   {
     insert( x, t->left );
     if( height( t->left ) - height( t->right ) == 2 )
       if( x < t->left->element )
           rotateWithLeftChild( t );  // case 1
       else
           doubleWithLeftChild( t );  // case 2
   }
   else if( t->element < x )
   {
       insert( x, t->right );
     if( height( t->right ) - height( t->left ) == 2 )
        if( t->right->element < x )
           rotateWithRightChild( t );  // case 4
        else
           doubleWithRightChild( t );  // case 3
    }
    else
       ;  // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

AVL Tree -- Deletion

- Deletion of a node x from an AVL tree requires the same basic ideas, including single and double rotations, that are used for insertion.
  - However, deletion is more complicated.

- We may need more than one rebalance on the path from deleted node to root.

- Deletion is O(logN)

Additional Data

- With each node of the AVL tree is associated a ***balance factor*** that is left high, equal or right high according, respectively, as the left subtree has height greater than, equal to, or less than that of the right subtree.

- We use a Boolean variable **shorter** to show if the height of a subtree has been shortened.

Method

Reduce the problem to the case when the node ***x*** to be deleted has at most one child (similar to regular BST deletion).

- If ***x*** has two children replace it with its immediate successor ***y*** under inorder traversal (the immediate predecessor would be just as good)
- Delete ***y*** from its original position, by proceeding as follows, using ***y*** in place of ***x*** in each of the following steps.

Method (cont.)

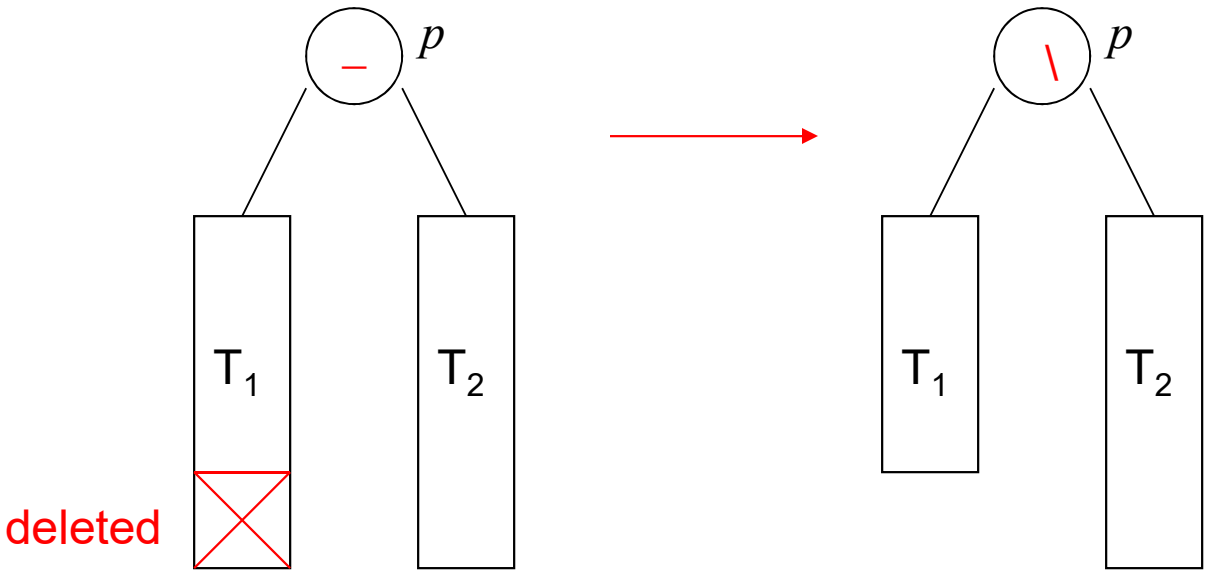
- Delete the node ***x*** from the tree.
  - We'll trace the effects of this change on height through all the nodes on the path from *x* back to the root.
  - The action to be taken at each node depends on
    - the value of **shorter**
    - **balance factor** of the node
    - sometimes **the balance factor of a child** of the node.

- **shorter** is initially `true`.
- The following steps are to be done for each node ***p*** on the path from the parent of ***x*** to the root, provided shorter remains `true`.
- When shorter becomes `false`, the algorithm terminates.

Case 1

Case 1: The current node *p* has balance factor equal (-).

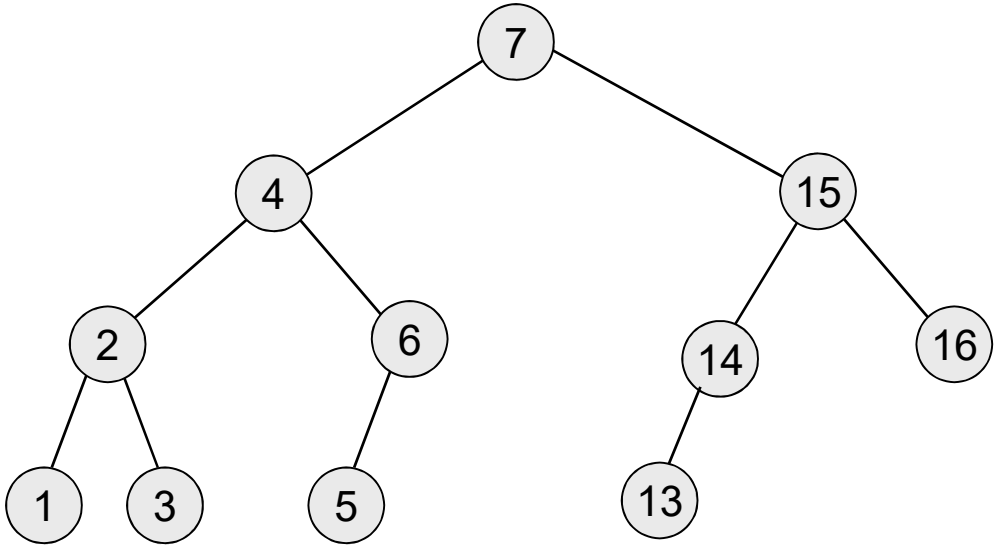
- Change the balance factor of *p.*
- shorter becomes false



- No rotations
- Height unchanged

Example – Case 1

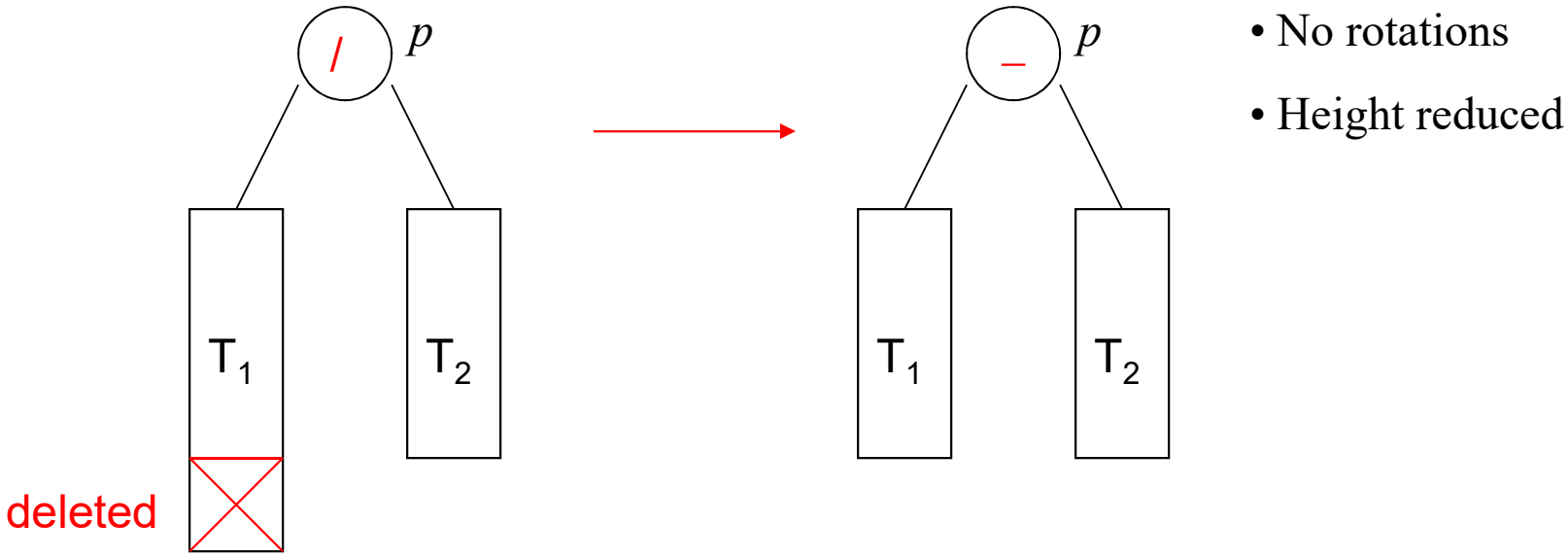
- Delete 3:



Case 2

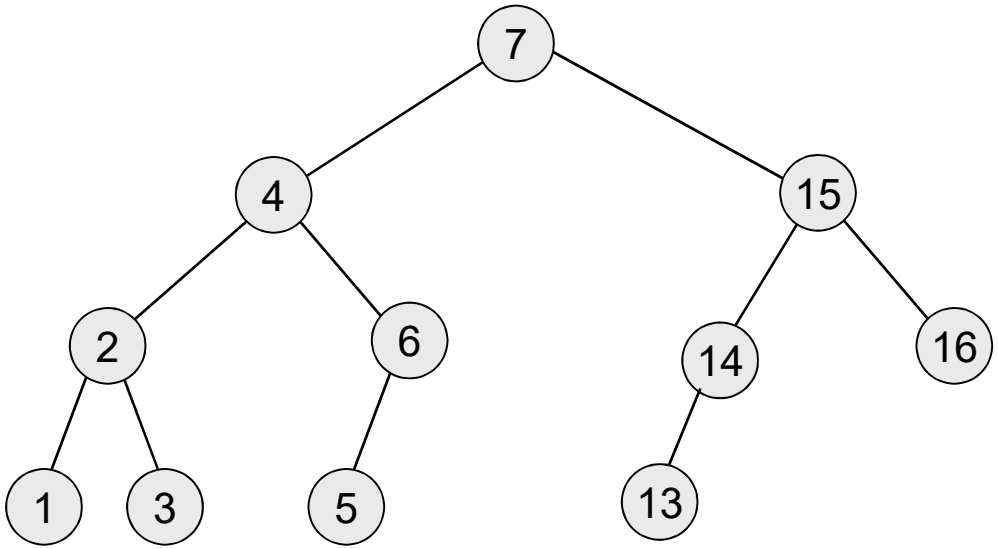
Case 2: The balance factor of *p* is not equal and the taller subtree was shortened.

- Change the balance factor of *p* to equal
- Leave shorter true.



Example – Case 2

- Delete 13:



Case 3

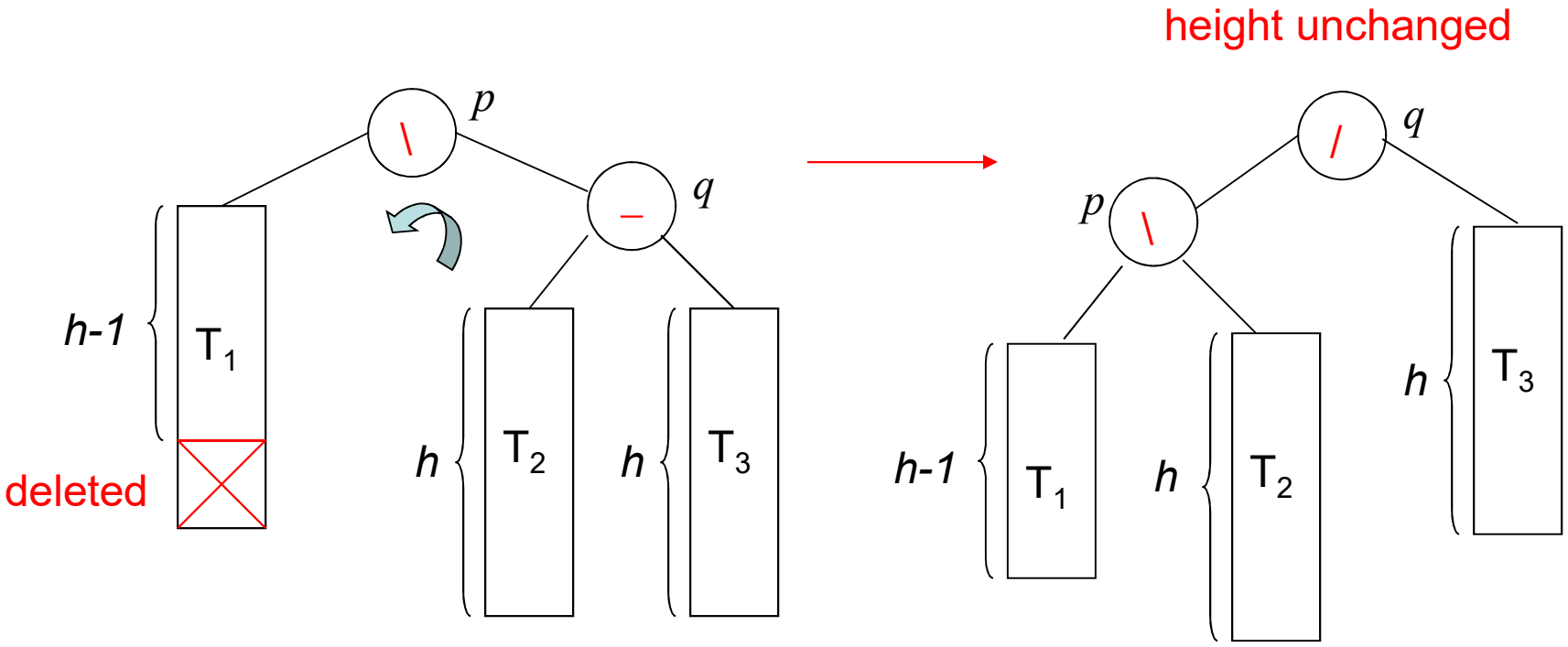
Case 3: The balance factor of *p* is not equal, and the shorter subtree was shortened.

- Rotation is needed.
- Let *q* be the root of the taller subtree of *p*. We have three cases according to the balance factor of *q*.

Case 3a

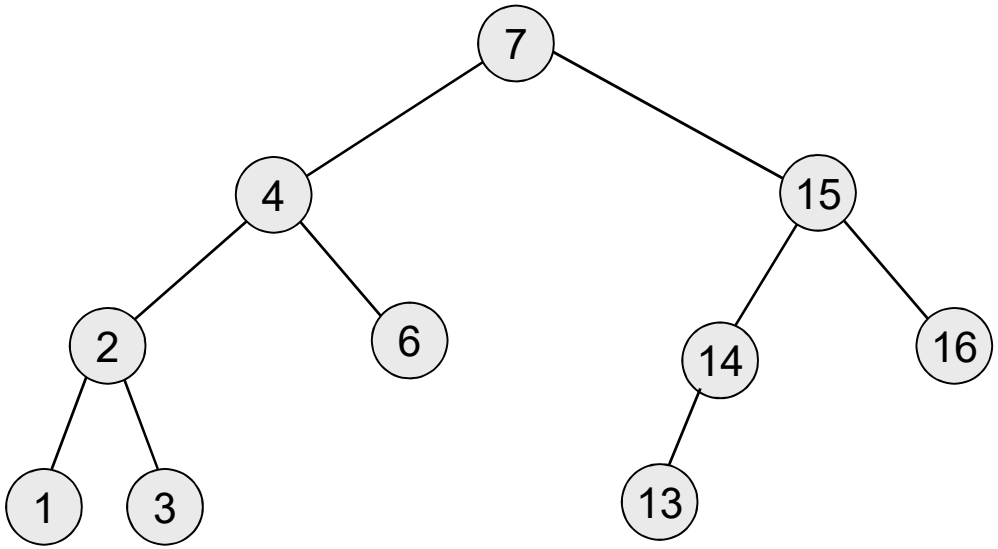
Case 3a: The balance factor of *q* is equal.

- Apply a single rotation
- shorter becomes false.



Example – Case 3a

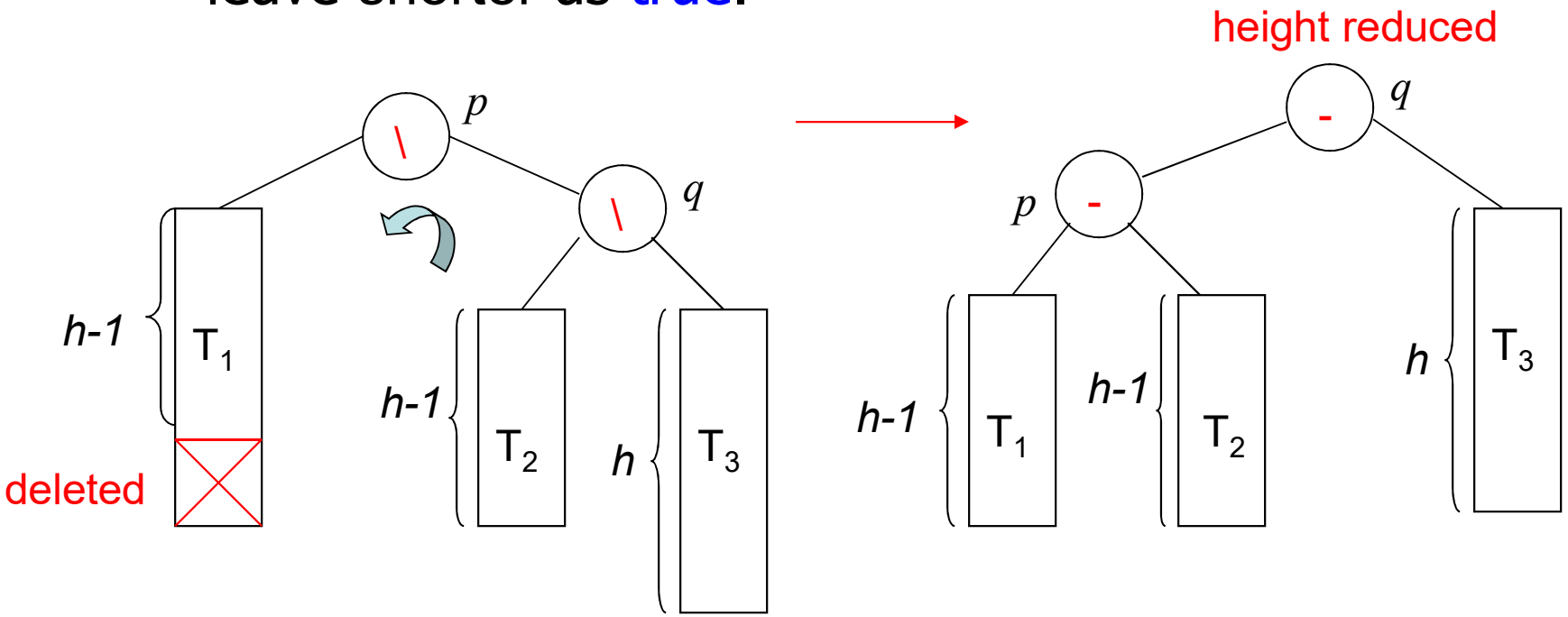
- Delete 6:



Case 3b

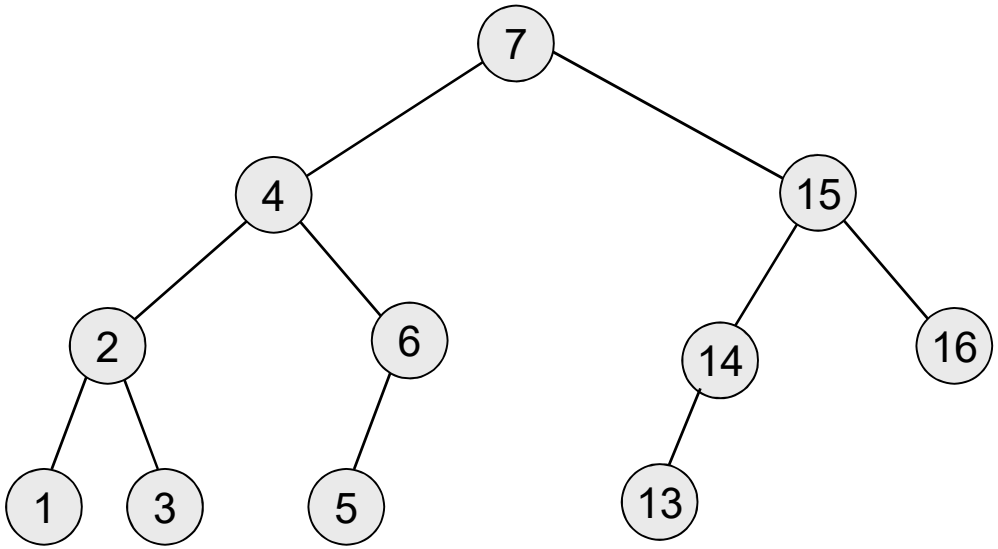
Case 3b: The balance factor of *q* is the same as that of p.

- Apply a single rotation
- Set the balance factors of *p* and *q* to equal
- leave shorter as true.



Example – Case 3b

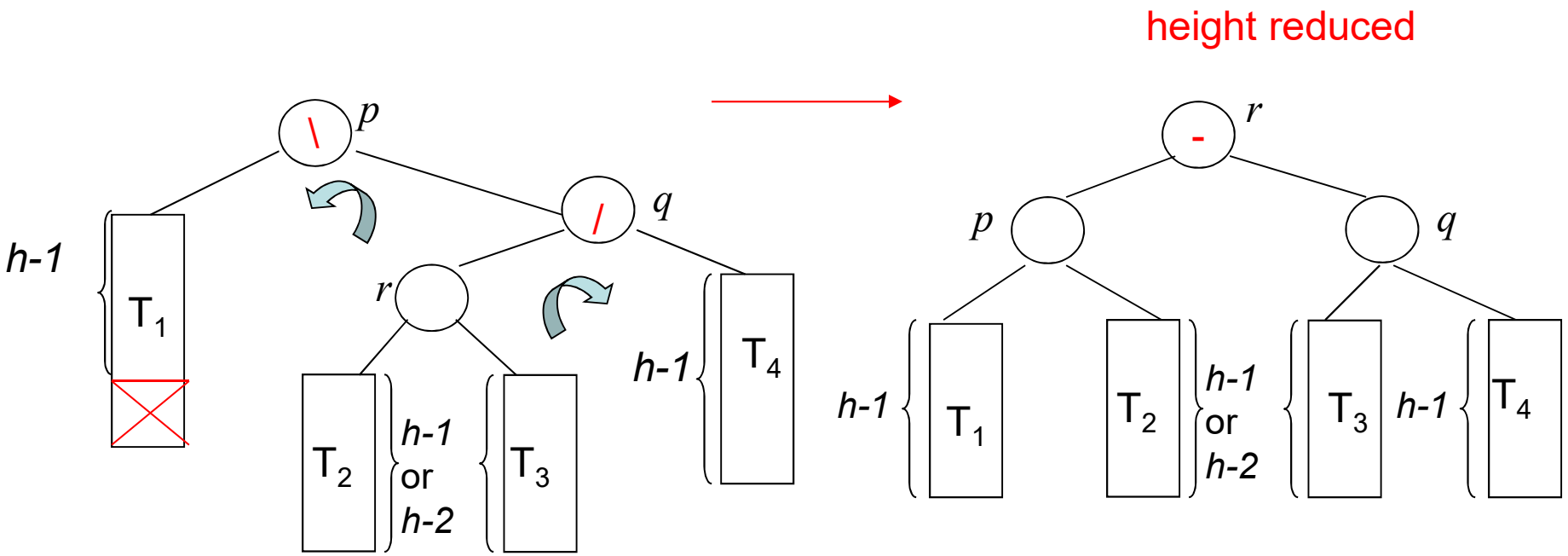
- Delete 16:



Case 3c

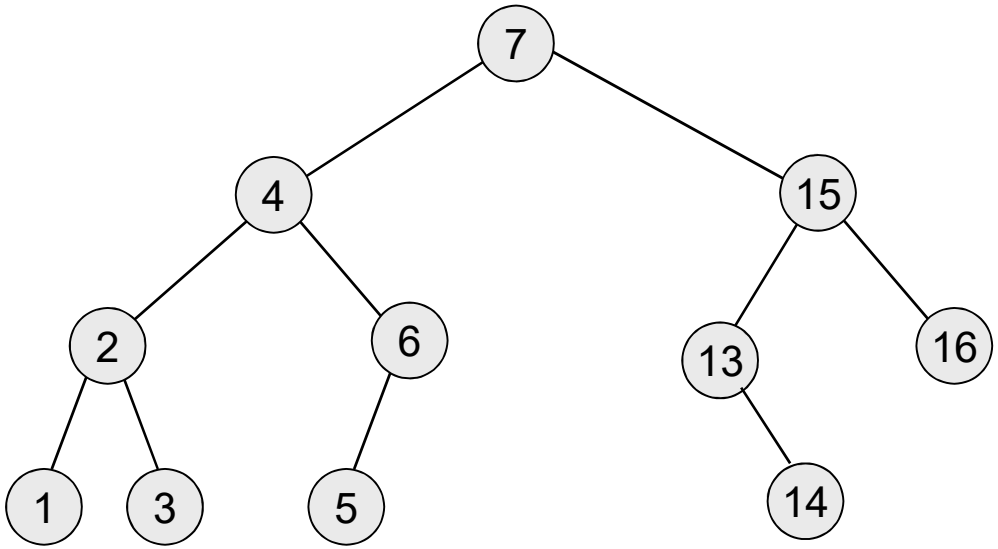
Case 3c: The balance factors of *p* and *q* are opposite.

- Apply a *double rotation*
- set the balance factors of the new root to equal
- leave shorter as true.



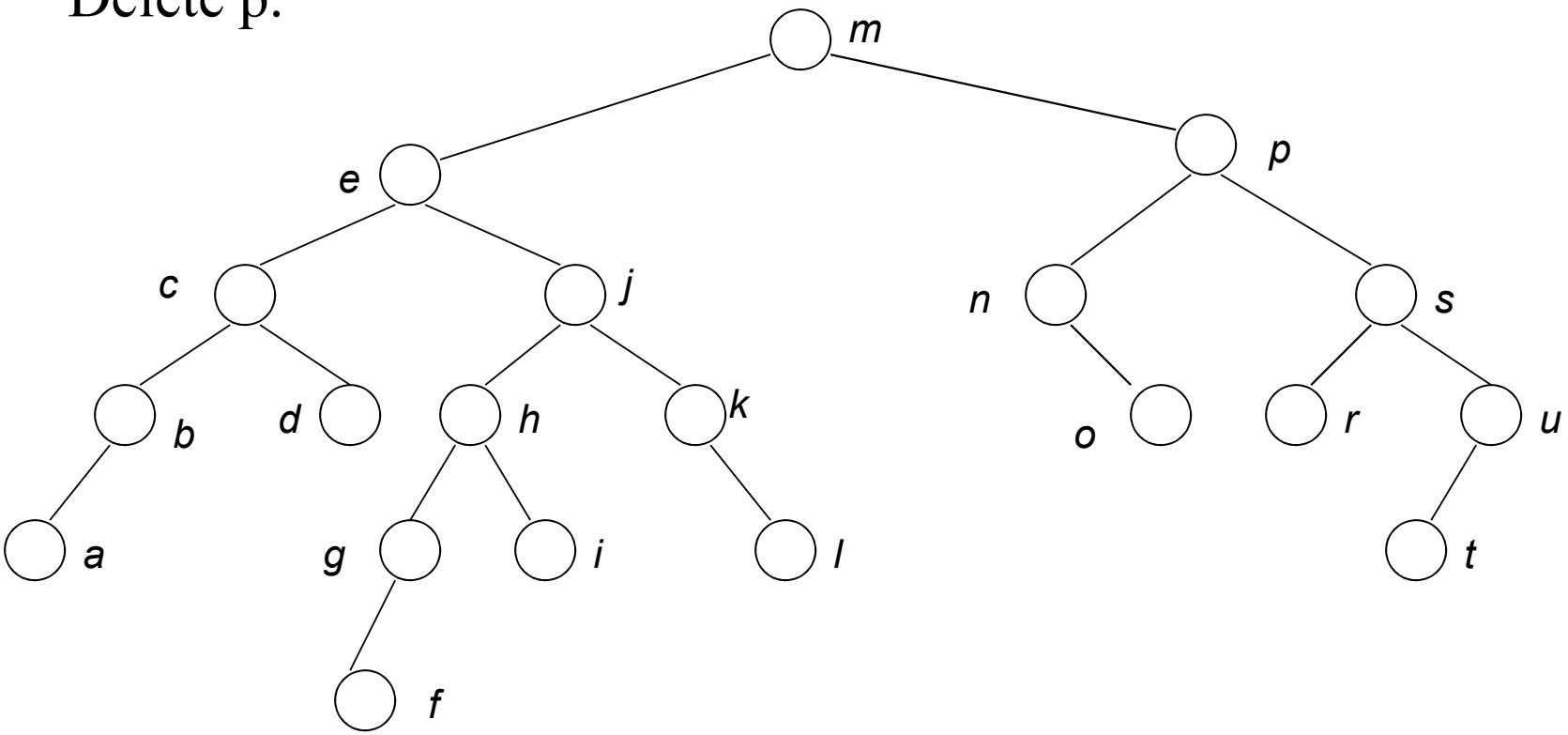
Example – Case 3c

- Delete 16:



Example

Delete p.

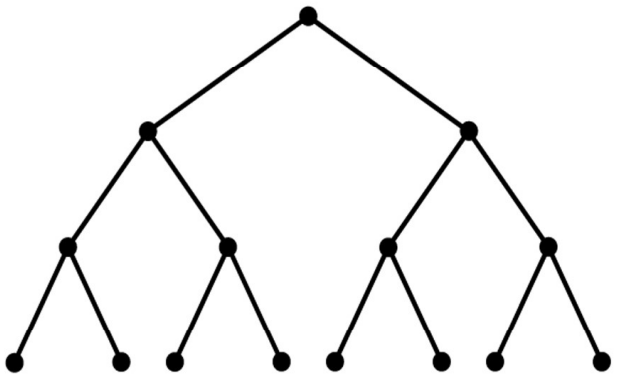


Some Height Theorems

Theorem 1: A binary tree with *h levels* can have a maximum of $2^h - 1$ nodes.

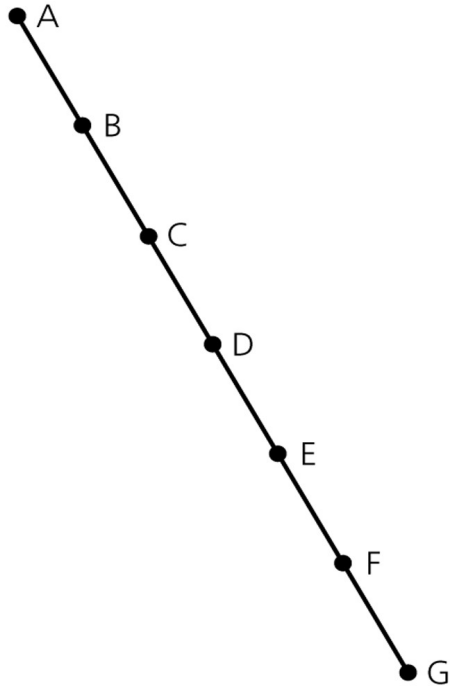
➔i.e. We cannot insert a new node into the binary tree without increasing its height.

Counting the nodes in a full binary tree

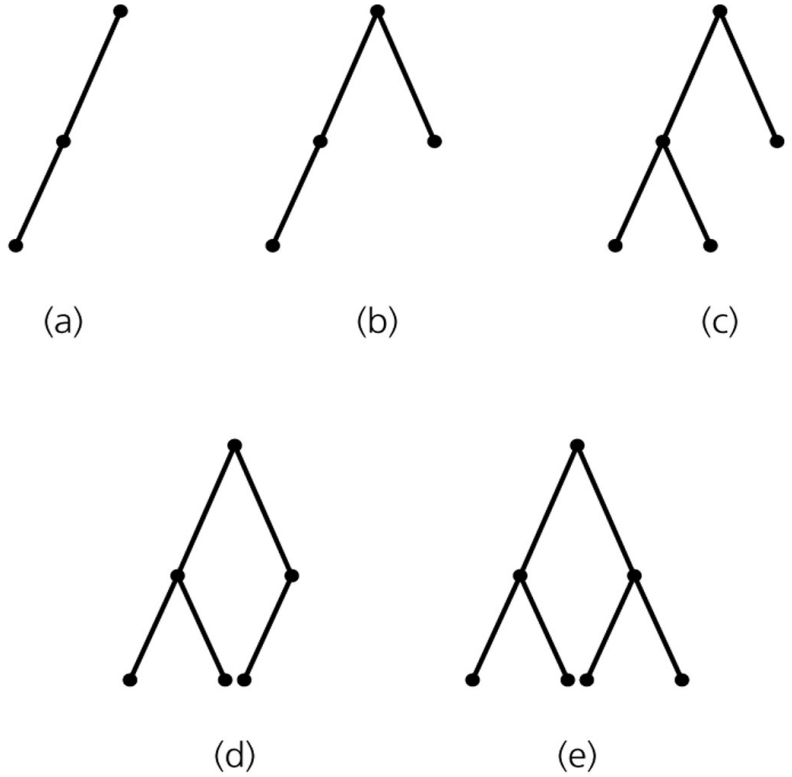


Level	Number of nodes at this level	Number of nodes at this and previous levels
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
.	.	.
.	.	.
.	.	.
h	2^{h-1}	$2^h - 1$

Maximum and Minimum Heights of a Binary Tree



A maximum-height binary tree with seven nodes



Some binary trees of height 2

Minimum Height

Theorem 2: The minimum height of a binary tree with n nodes is $\lceil \log_2(n+1)-1 \rceil$.

- Complete trees have minimum height.
- Insertion in search-key order produces a maximum-height *binary search tree*.
- Insertion in random order produces a near-minimum-height binary *search* tree.
- That is, the height of an n-node binary search tree
  - *Best Case* – $\lceil \log_2(n+1)-1 \rceil$ ➔ $O(\log_2 n)$
  - *Worst Case* – n-1 ➔ $O(n)$
  - *Average Case* – close to $\lceil \log_2(n+1)-1 \rceil$ ➔ $O(\log_2 n)$
    - In fact, $1.39\log_2 n$

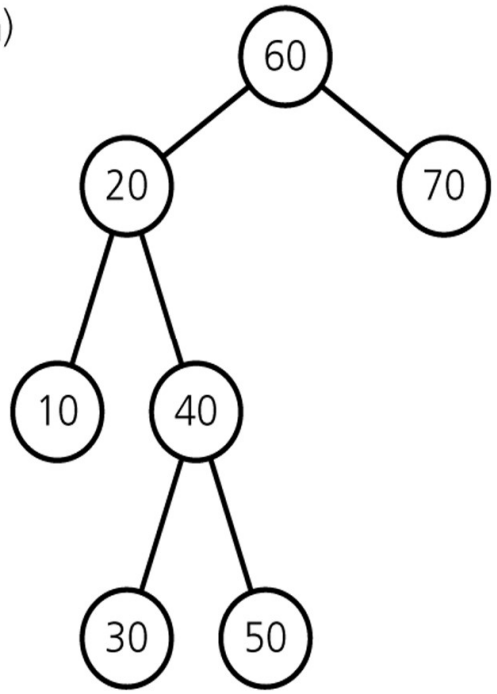
Saving a BST into a file, and restoring it to its original shape

- Save:
  - Use a preorder traversal to save the nodes of the BST into a file.

- Restore:
  - Start with an empty BST.
  - Read the nodes from the file one by one, and insert them into the BST.

Saving a BST into a file, and restoring it to its original shape

(a)



(b)

```
bst.searchTreeInsert(60);
bst.searchTreeInsert(20);
bst.searchTreeInsert(10);
bst.searchTreeInsert(40);
bst.searchTreeInsert(30);
bst.searchTreeInsert(50);
bst.searchTreeInsert(70);
```

Preorder: 60 20 10 40 30 50 70

Saving a BST into a file, and restoring it to a minimum-height BST

- Save:
  - Use an inorder traversal to save the nodes of the BST into a file. The saved nodes will be in ascending order.
  - Save the number of nodes (n) in somewhere.

- Restore:
  - Read the number of nodes (n).
  - Start with an empty BST.
  - Read the nodes from the file one by one to create a minimum-height binary search tree.

Building a minimum-height BST

```
readTree(out treePtr:TreeNodePtr, in n:integer)
// Builds a minimum-height binary search tree for n sorted
// values in a file. treePtr will point to the tree's root.

if (n>0) {
    // construct the left subtree
    treePtr = pointer to new node with NULL child pointers
    readTree(treePtr->leftChildPtr, n/2)

    // get the root
    Read item from file into treePtr->item

    // construct the right subtree
    readTree(treePtr->rightChildPtr, (n-1)/2)
  }
```

A full tree saved in a file by using inorder traversal