09
Search Trees
Chapter 11

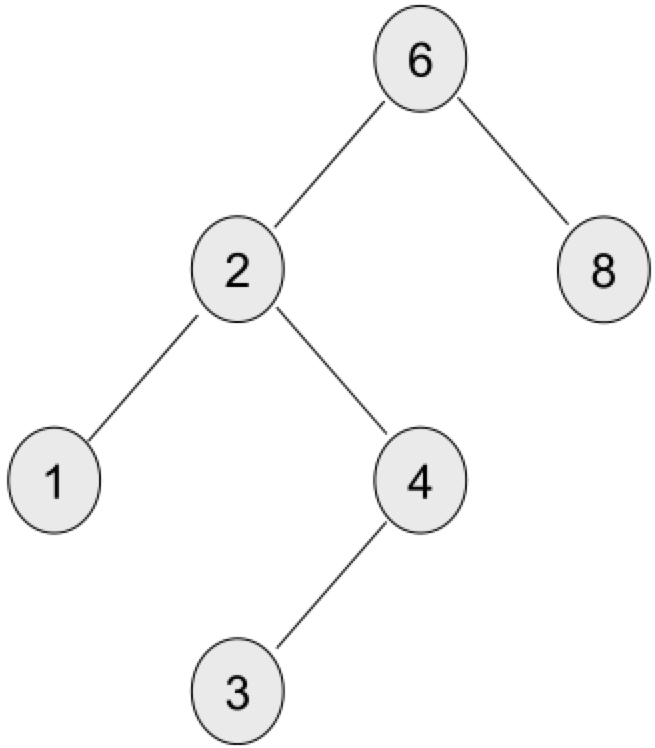
Binary Search Trees

An important application of binary trees is their use in searching.

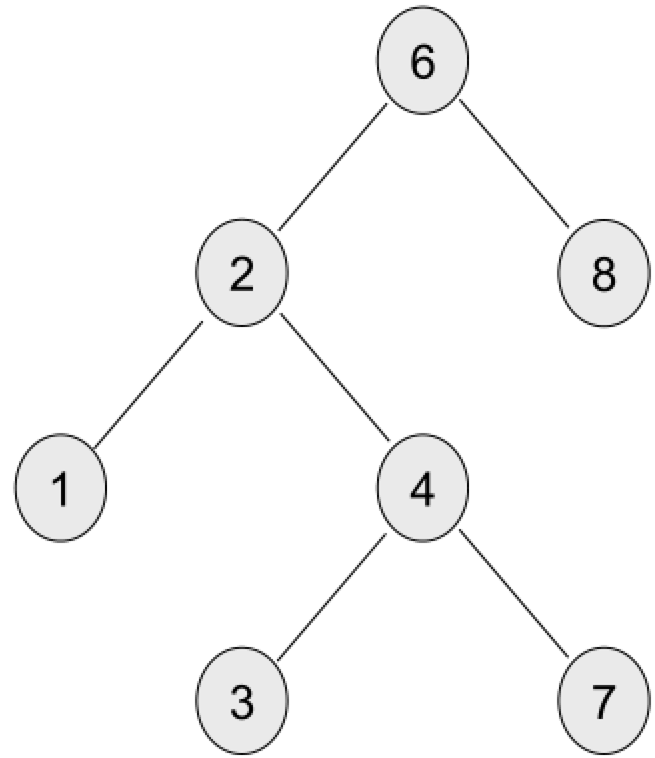
Binary search tree is a binary tree in which every node X contains a data value satisfying the following:

- All data values in its left subtree are smaller than the data value in X
- The data value in X is smaller than all the values in its right subtree.
- The left and right subtrees are also binary search tees.

Binary Search Trees



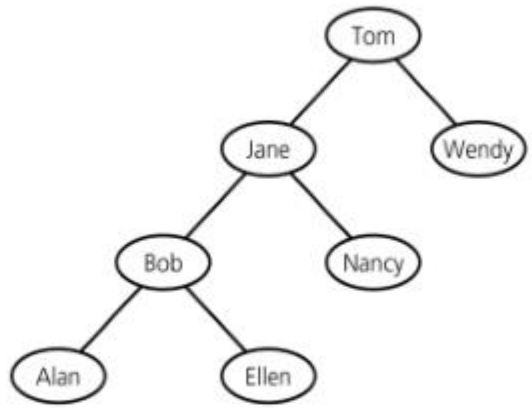
A *binary search tree*



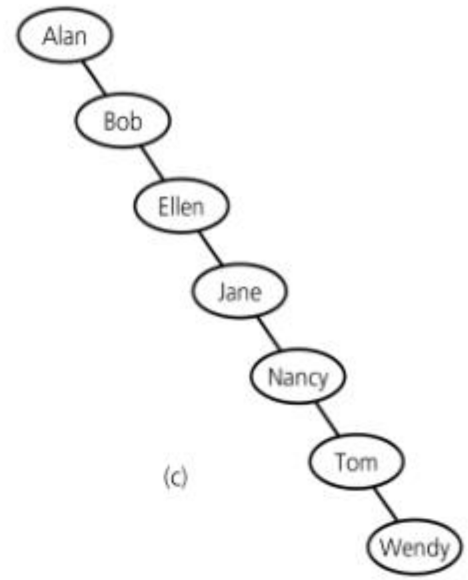
Not a *binary search tree*, but a *binary tree*

Binary Search Trees

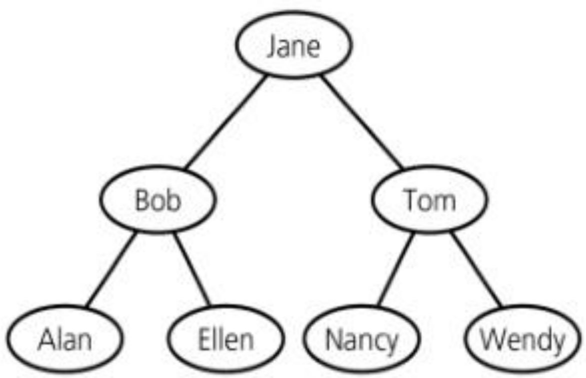
Various binary search trees having the same data

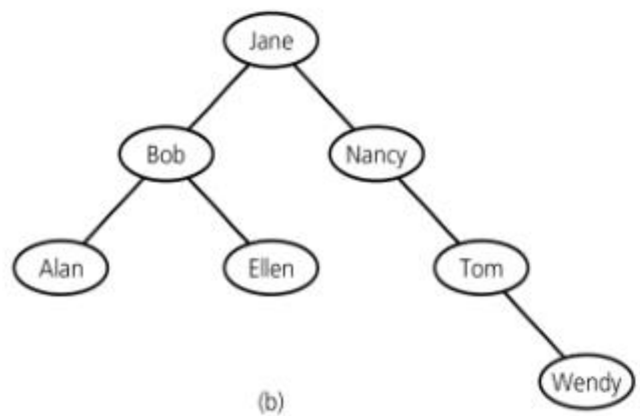


(a)



(c)

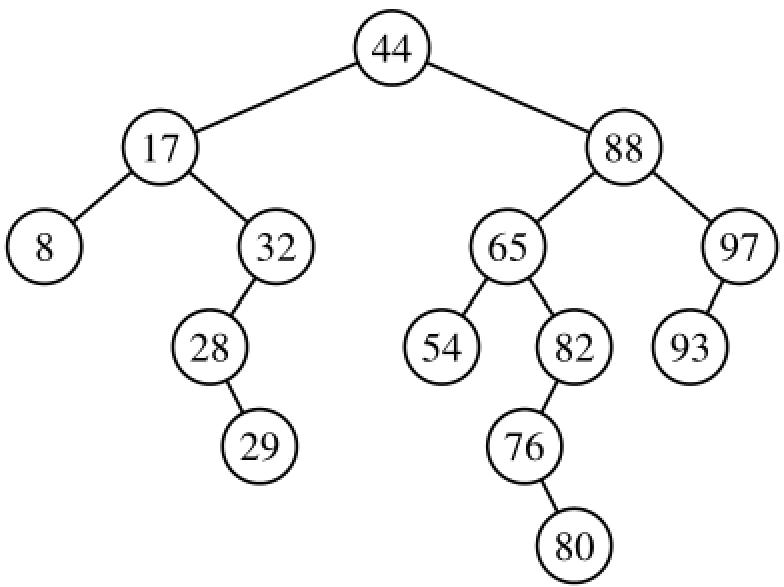




(b)

Binary Search Trees

An in-order traversal of a binary search tree visits positions in increasing order of their keys.



Algorithm inorder(p):

  if p has a left child lc **then**

    inorder(lc)

  perform the "visit" action for position p

  if p has a right child rc **then**

    inorder(rc)

Binary Search Trees

Binary Search Tree Operations

In addition to basic binary tree operations such as parent, left, child, right, etc., binary search trees also supports the following operations:

first()	Returns the position with the lowest key, None if tree is empty.
last()	Returns the position with the highest key, None if tree is empty.
before(p)	Returns the position having the highest key that is smaller than that of p, return None if p is the first.
after(p)	Returns the position having the lowest key that is larger than that of p, return None if p is the last.

Binary Search Trees

Binary Search Tree Operations

first()

```
def first(T):
    if T is None: return None
    p = T.root()
    while p.left() is not None:
        p = p.left()
    return p
```

Binary Search Trees

Binary Search Tree Operations

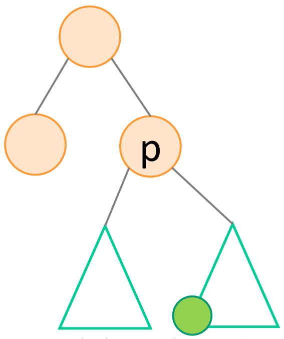
last()

```
def last(T):
    if T is None: return None
    p = T.root()
    while p.right() is not None:
        p = p.right()
    return p
```

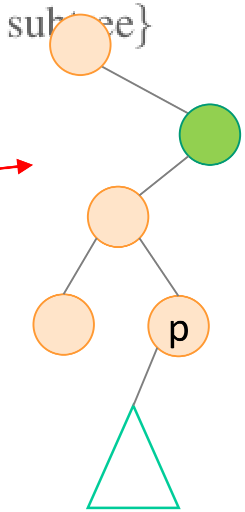
Binary Search Trees

Binary Search Tree Operations

after()




```
Algorithm after(p):
  if right(p) is not None then {successor is leftmost position in p's right subtree}
    walk = right(p)
    while left(walk) is not None do
      walk = left(walk)
    return walk
  else {successor is nearest ancestor having p in its left subtree}
    walk = p
    ancestor = parent(walk)
    while ancestor is not None and walk == right(ancestor) do
      walk = ancestor
      ancestor = parent(walk)
    return ancestor
```

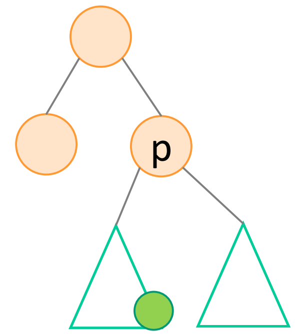


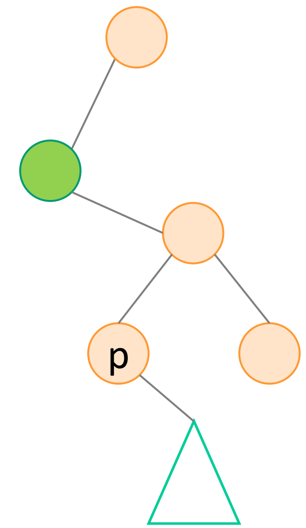
Binary Search Trees

Binary Search Tree Operations

before()

Take the writing the pseudo code as a HW!





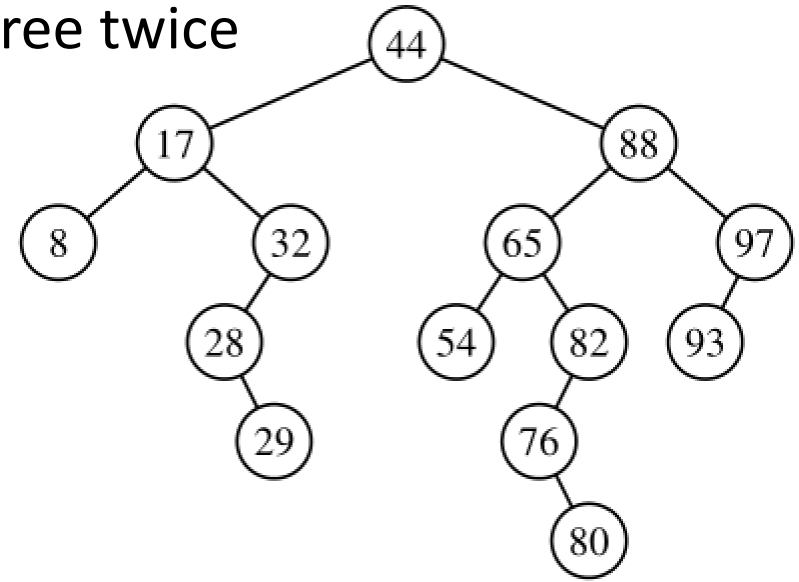
Binary Search Trees

Binary Search Tree Operations

Worst-case complexity of **after()** and **before()** is O(h).

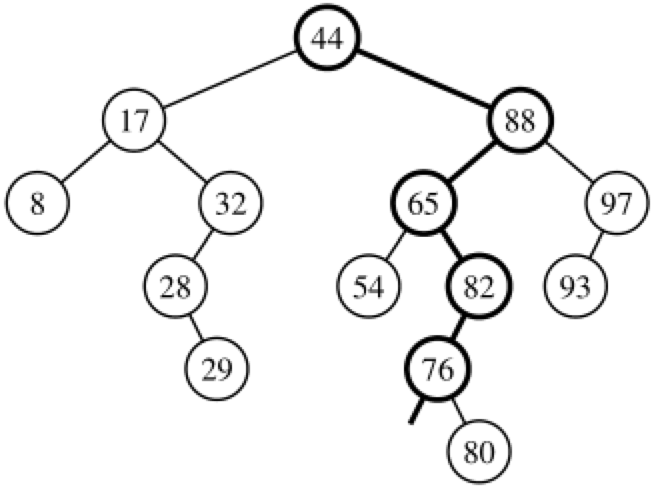
However, they run in O(1*) in the long run.

- Make after() calls starting from the first()
- It will be like traversing the tree twice



Binary Search Trees

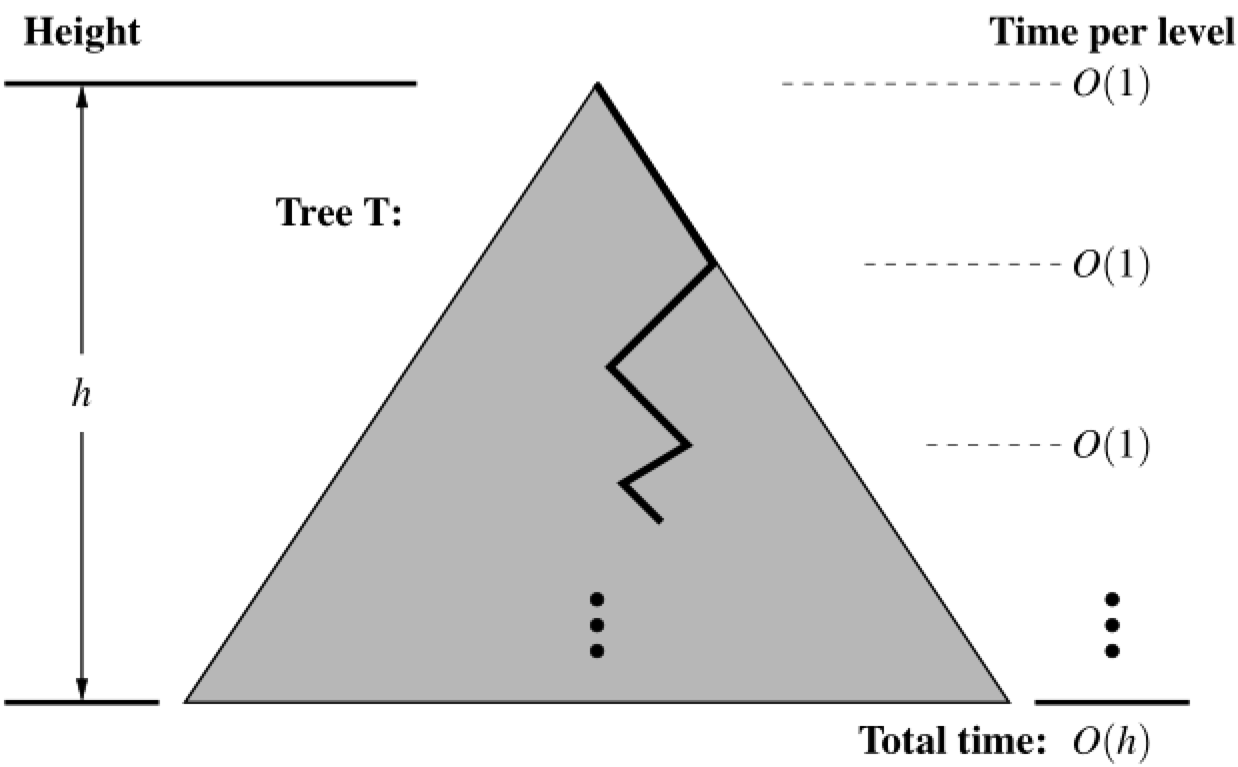
Search



Algorithm TreeSearch(T, p, k):
    if k == p.key() **then**
        return *p*   {successful search}
    else if k < p.key() and T.left(p) is not None **then**
        return TreeSearch(T, T.left(p), k)   {recur on left subtree}
    else if k > p.key() and T.right(p) is not None **then**
        return TreeSearch(T, T.right(p), k)   {recur on right subtree}
    return p   {unsuccessful search}

Binary Search Trees

Search



Binary Search Trees

TreeInsert(T, k, v)

Search for the node whose key is k

If search is successful, replace value of node k with value v

If not (let p be the node where search ends),

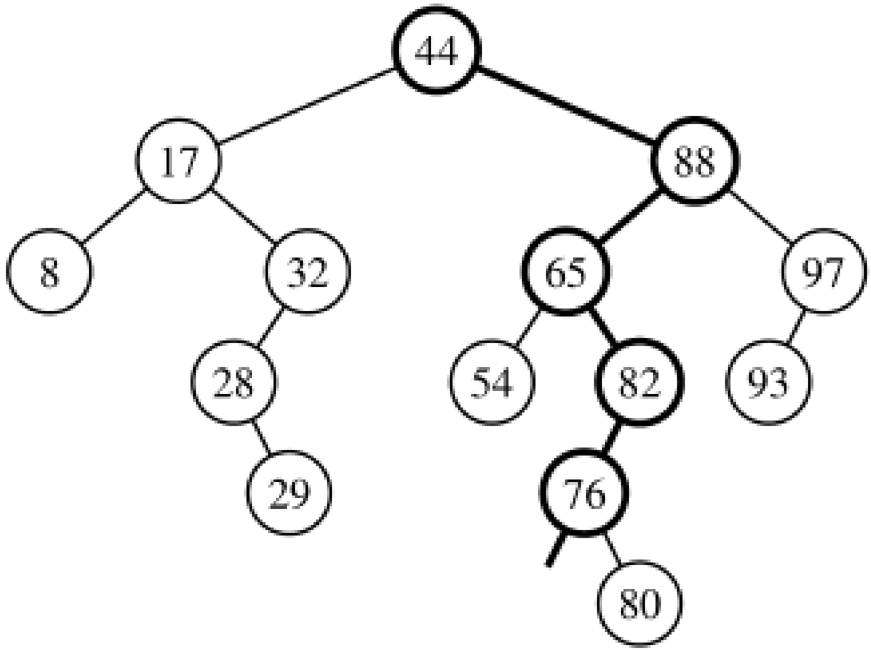
- If k < p.key() then add new node (k,v) as the left child of p
- Otherwise, add new node (k,v) as the right child of p

Algorithm TreeInsert(T, k, v):
  Input: A search key k to be associated with value v
  p = TreeSearch(T, T.root(), k)
  if k == p.key() **then**
    Set p's value to v
  else if k < p.key() **then**
    add node with item (k,v) as left child of p
  else
    add node with item (k,v) as right child of p

Binary Search Trees

TreeInsert(T, k, v)

Let's try to add value 68.

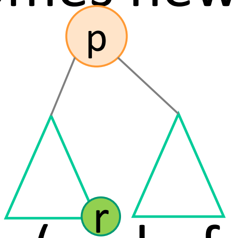


TreeDelete(T, k)

The node to be deleted, p, if found with TreeSearch(T, T.root(), k)

If p does not have a child, it is simply deleted.

Else if p has one child r, p is deleted and r becomes new child of parent of p.



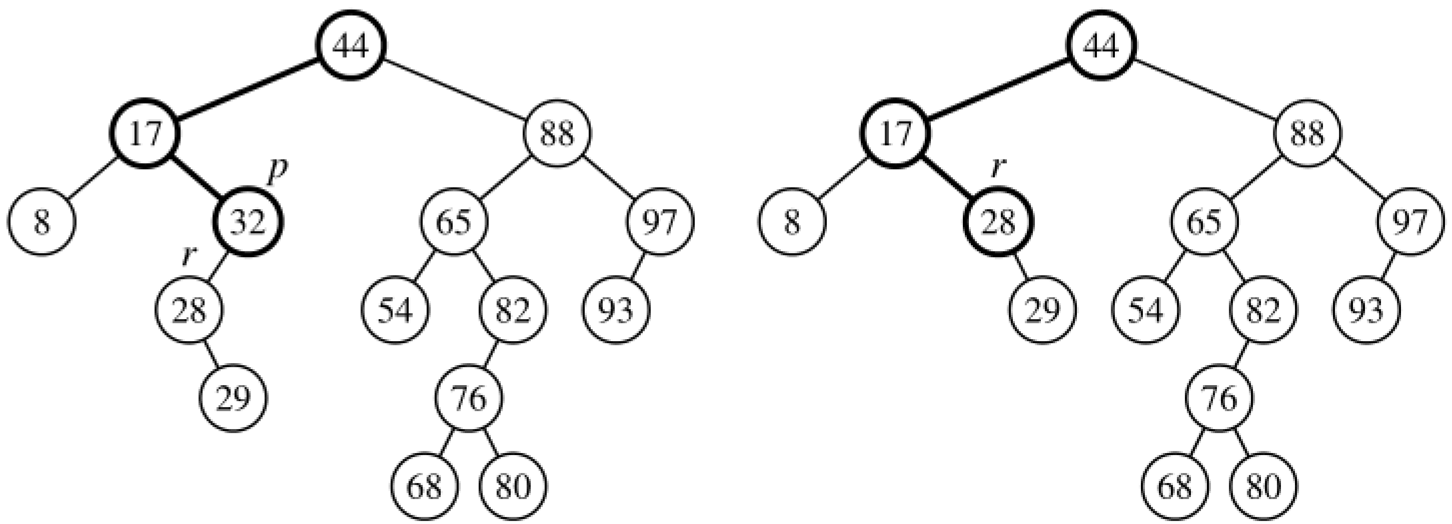
Else,

- We find the greatest key of left subtree of p. (r = before(p))
- Replace p with r.
- TreeDelete(T,r) (r will not have a right child, so deleting r is simple)

Binary Search Trees

TreeDelete(T, k)

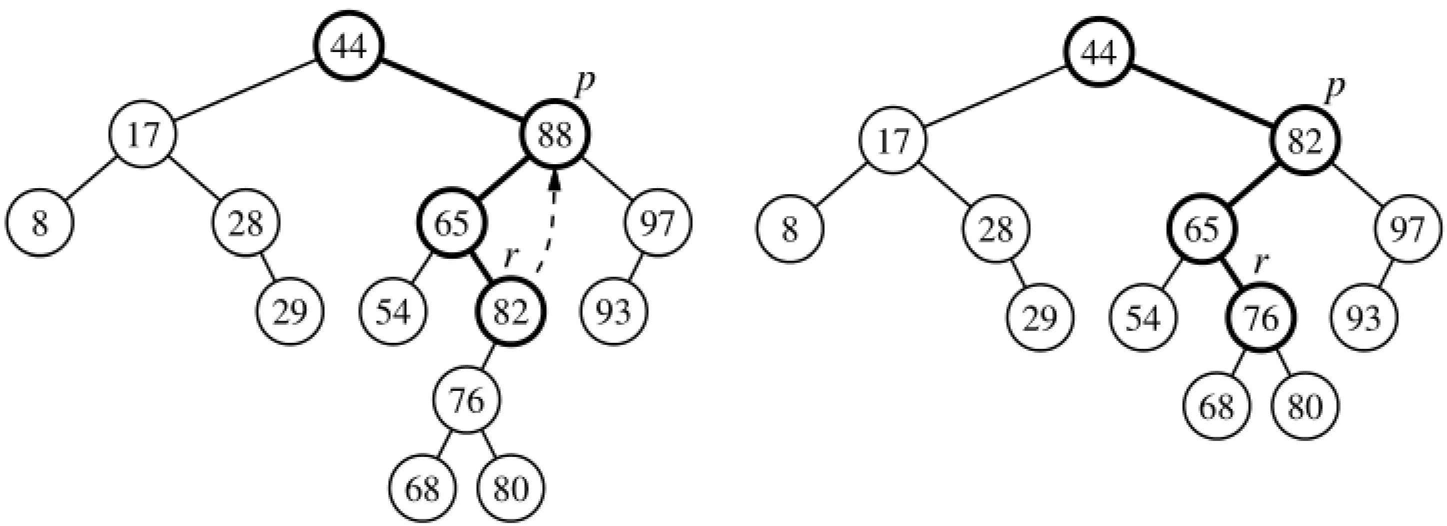
Simple Case



Binary Search Trees

TreeDelete(T, k)

Not-so-simple Case



Binary Search Trees

Complexities

first **O(h)**

last **O(h)**

before **O(h)**

after **O(h)**

search **O(h)**

insert **O(h)**

delete **O(h)**

Binary Search Trees

Complexities

first **O(h)**

last **O(h)**

before **O(h)**

after **O(h)**

search **O(h)**

insert **O(h)**

delete **O(h)**

Seems great.
In the best case h = log(n+1)-1.

What if h = n-1, which is the worst-case?

So we need to find a way to make height O(logn) in the worst-case.

Balancing Trees

In order to make height O(logn), we can use more advanced structures such as AVL trees, splay trees, red-black trees, and multiway trees.

In this course, we will cover

- AVL trees and
- Multiway trees.

AVL Trees

AVL: Adel'son-Vel'skii and Landis

Definition:

A binary search tree is said be an AVL tree if it satisfies heigh-balance property.

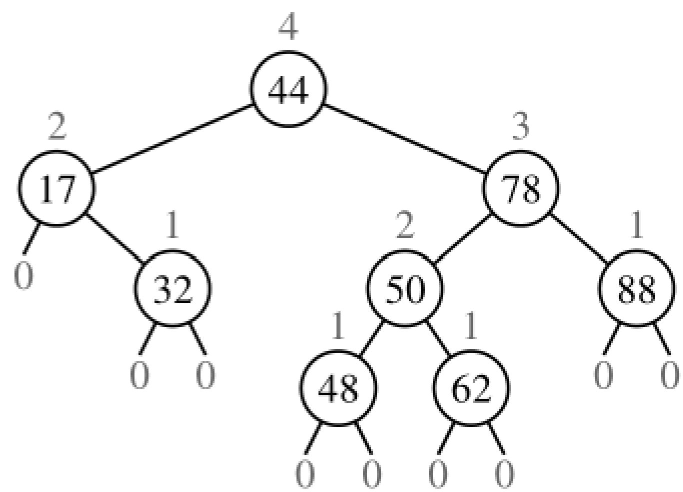
Height-balance Property:

For every position p of T, the heights of the children of p differ by at most 1.

AVL Trees

Height Definition (in the AVL context):

Maximum number of nodes from a node p to leaf node.



AVL Trees

A subtree of an AVL tree is also an AVL tree.

The height of an AVL tree storing n entries is O(logn).

Let f(h) be the function of number of nodes changing with h.

$f(1) = 1$

$2 <= f(2) <= 3$

Height of left and right subtrees of root has to minimally be h-1 and h-2. So the total number of nodes

$f(h) = f(h-1) + f(h-2) + 1$ (left subtree+right subtree+root node)

$f(h) > 2*f(h-2) = 2*f(2*f(h-4)) = 2^i * f(h-2i)$

AVL Trees

The height of an AVL tree storing n entries is O(logn).

$f(h) > 2^i * f(h-2i)$

$f(1) = 1$

When i becomes (h-1)/2,

$f(h) > 2^{(h-1)/2} * f(1)$

$\log(f(h)) > (h-1)/2$

$2*\log(f(h))+1 > h$

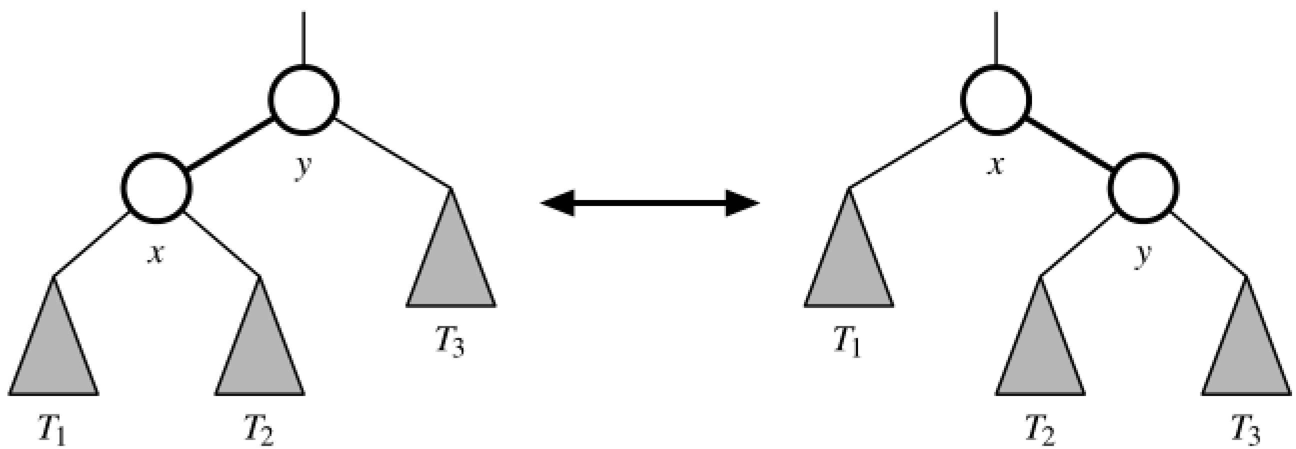
Now it is easy to find k and n_0 to show that h is O(logn).

Rotation

After insertion or deletion, a tree might become unbalanced.

Rotation: One of the primary operations to rebalance a binary search tree.

"We rotate a child to be above its parent."



Rotation

There are four cases that we need to consider when making rebalancing.

Suppose the node to be rebalanced is X. Possible cases:

Case 1: An insertion in the left subtree of the left child of X.

Case 2: An insertion in the right subtree of the left child of X.

Case 3: An insertion in the left subtree of the right child of X.

Case 4: An insertion in the right subtree of the right child of X.

Case 1 and 4 requires single rotation.

Case 2 and 3 requires double rotation.

Rotation

How do we determine which node rebalance (X)?

Suppose that an AVL tree has become unbalanced after adding a new as a child of node p. The node to rebalance (X) is the nearest ancestor of p that becomes unbalanced.

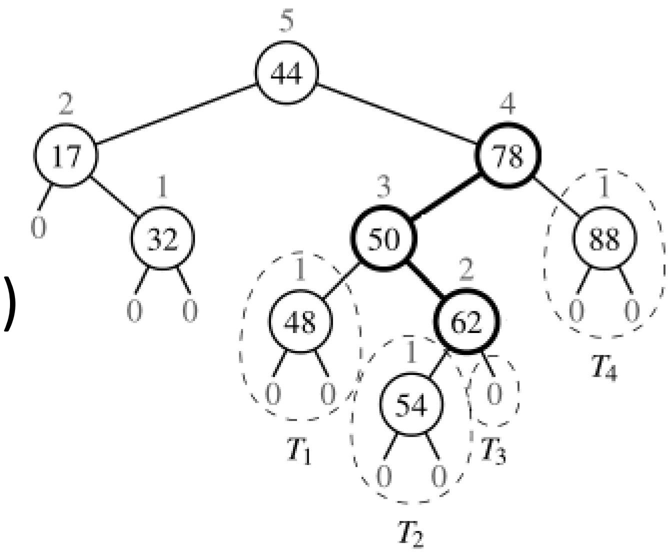
<u>Example:</u>

Node added: 54

p: 62

X: 78 (nearest unbalanced ancestor of p)

(Note that 50 is balanced.)



Single Rotation

A single rotation switches the roles of the parent and child while maintaining the search order.

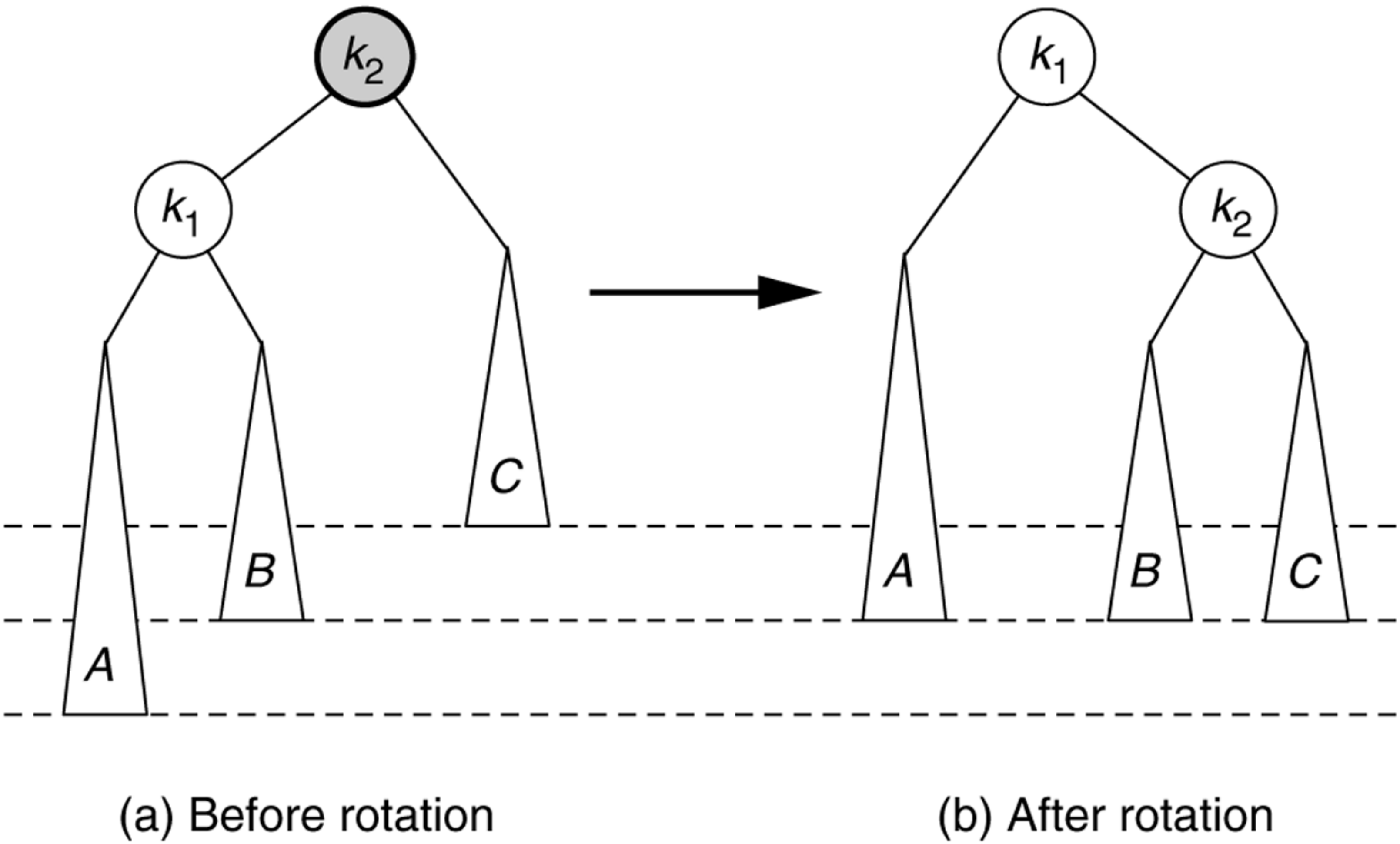
Single rotation handles the outside cases (Case 1 and 4).

We rotate between a node and its child.

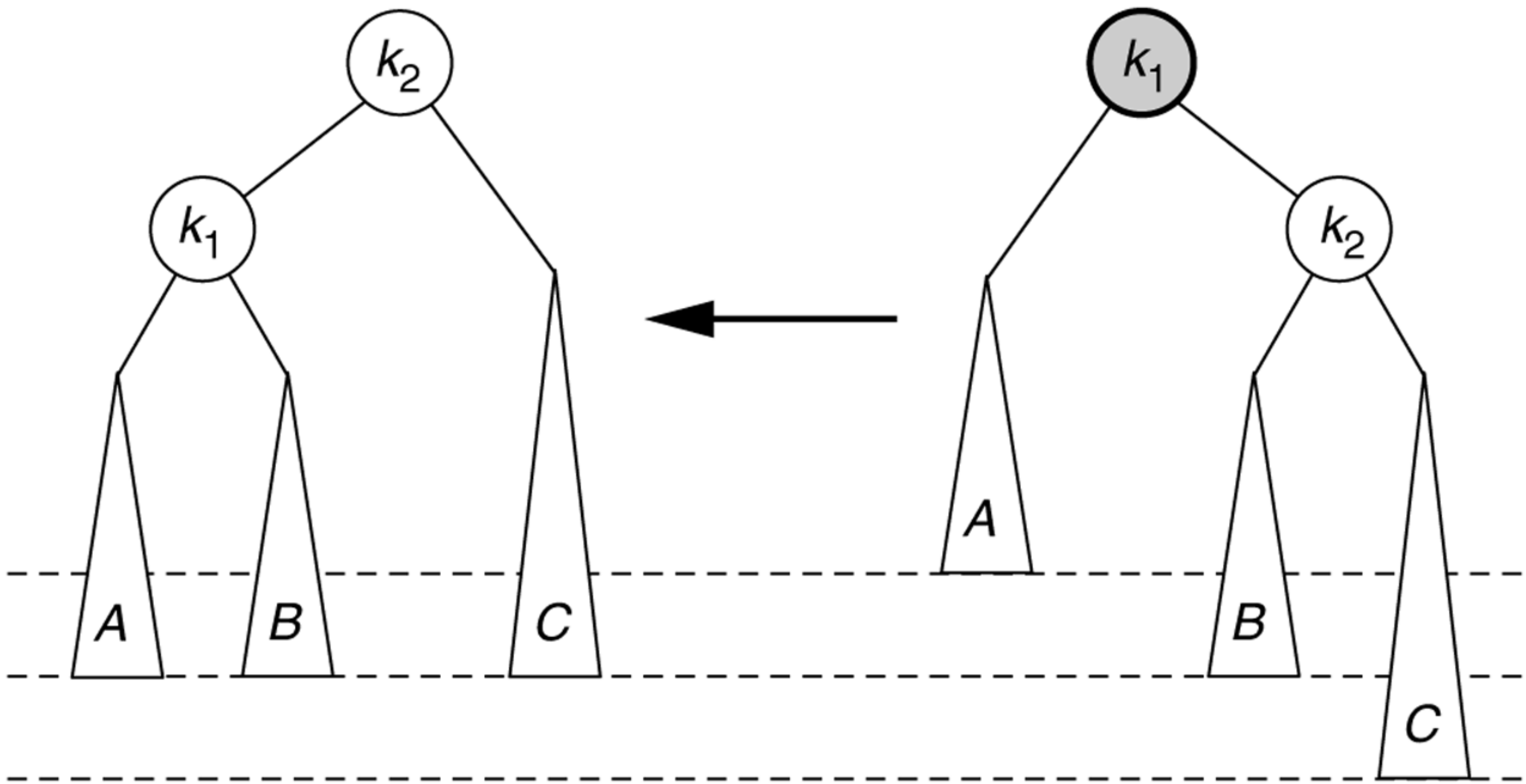
Child becomes parent. Parent becomes right child in case 1, left child in case 4.

The result is a binary search tree that satisfies the AVL property.

Single Rotation (Case 1: Rotate Right)

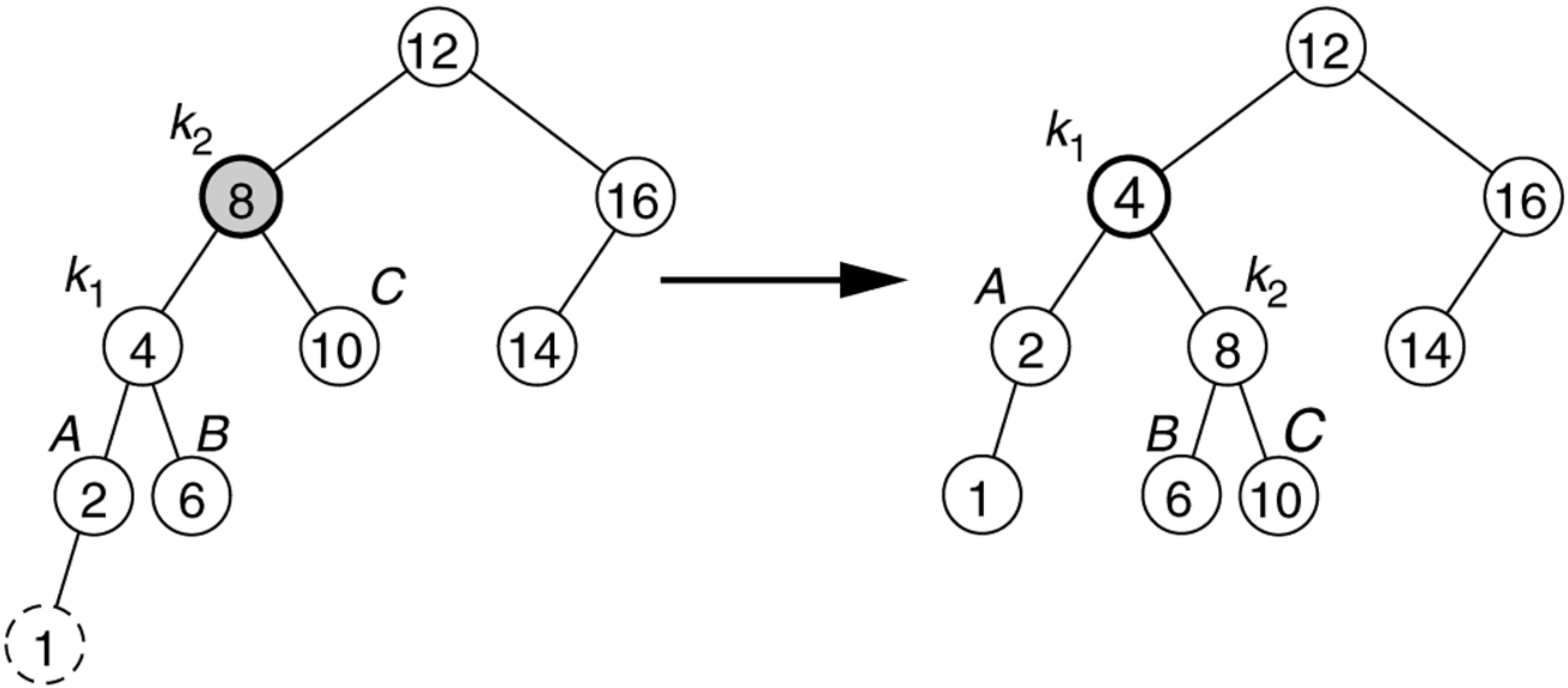


Single Rotation (Case 4: Rotate Left)



(a) After rotation (b) Before rotation

Single Rotation (Case 1 Example)



(a) Before rotation (b) After rotation

Double Rotation

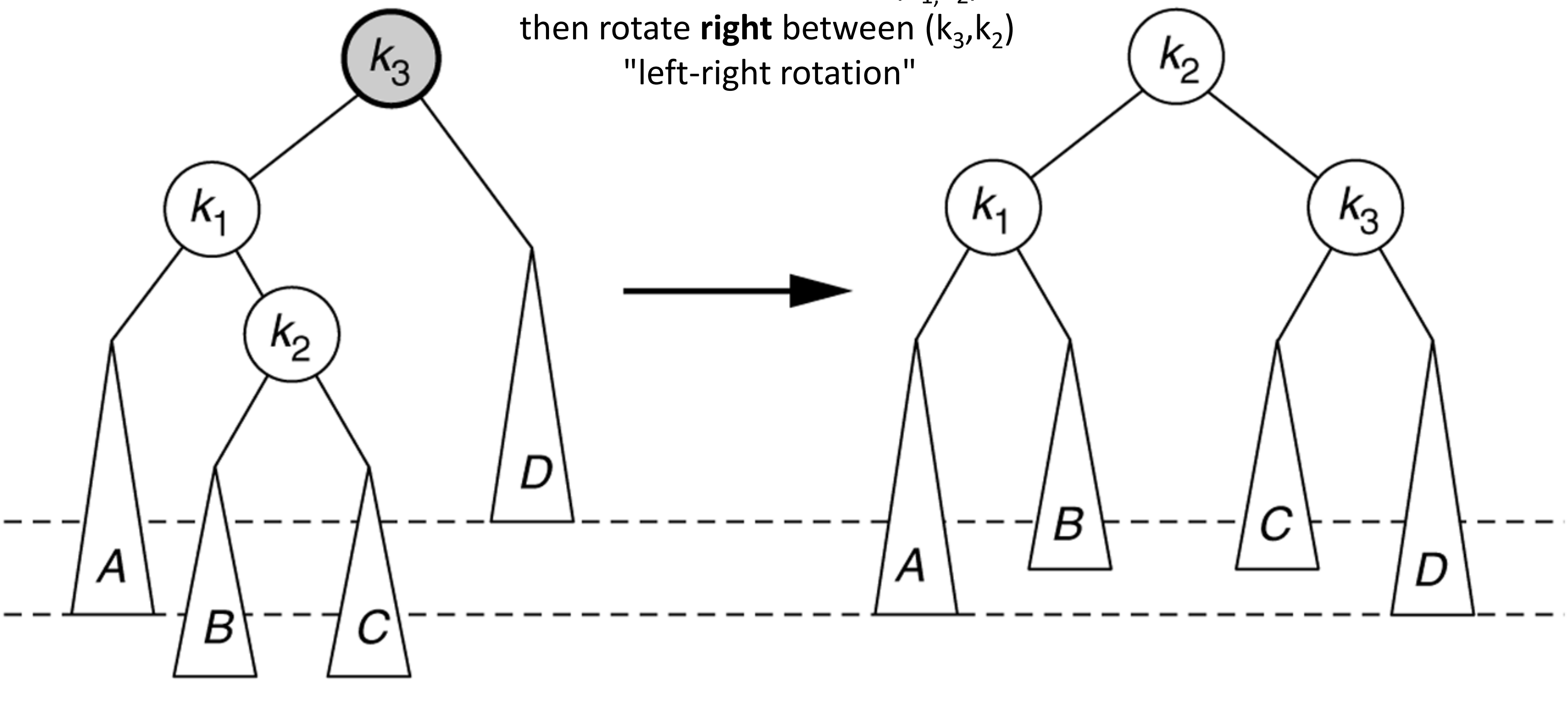
In order to keep the tree balanced, rotation may be applied multiple times.

Case 2 and 3 can be solved by applying double rotation.

Double rotation involves three nodes and four subtrees.

Double Rotation (Case 2)

first rotate **left** between (k_1,k_2),
then rotate **right** between (k_3,k_2)
"left-right rotation"



(a) Before rotation (b) After rotation

Double Rotation (Case 2)

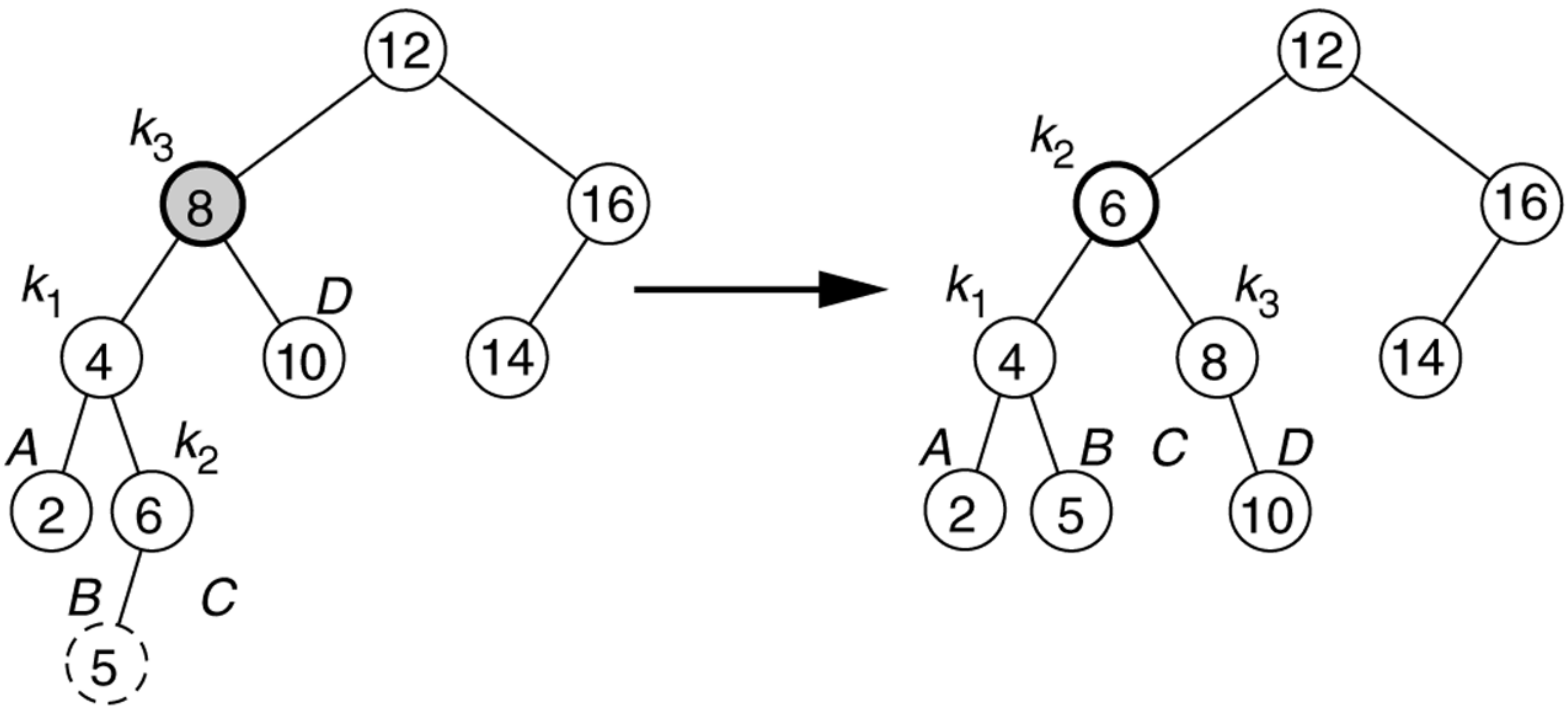
Left-right Rotation

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 (Case 2)

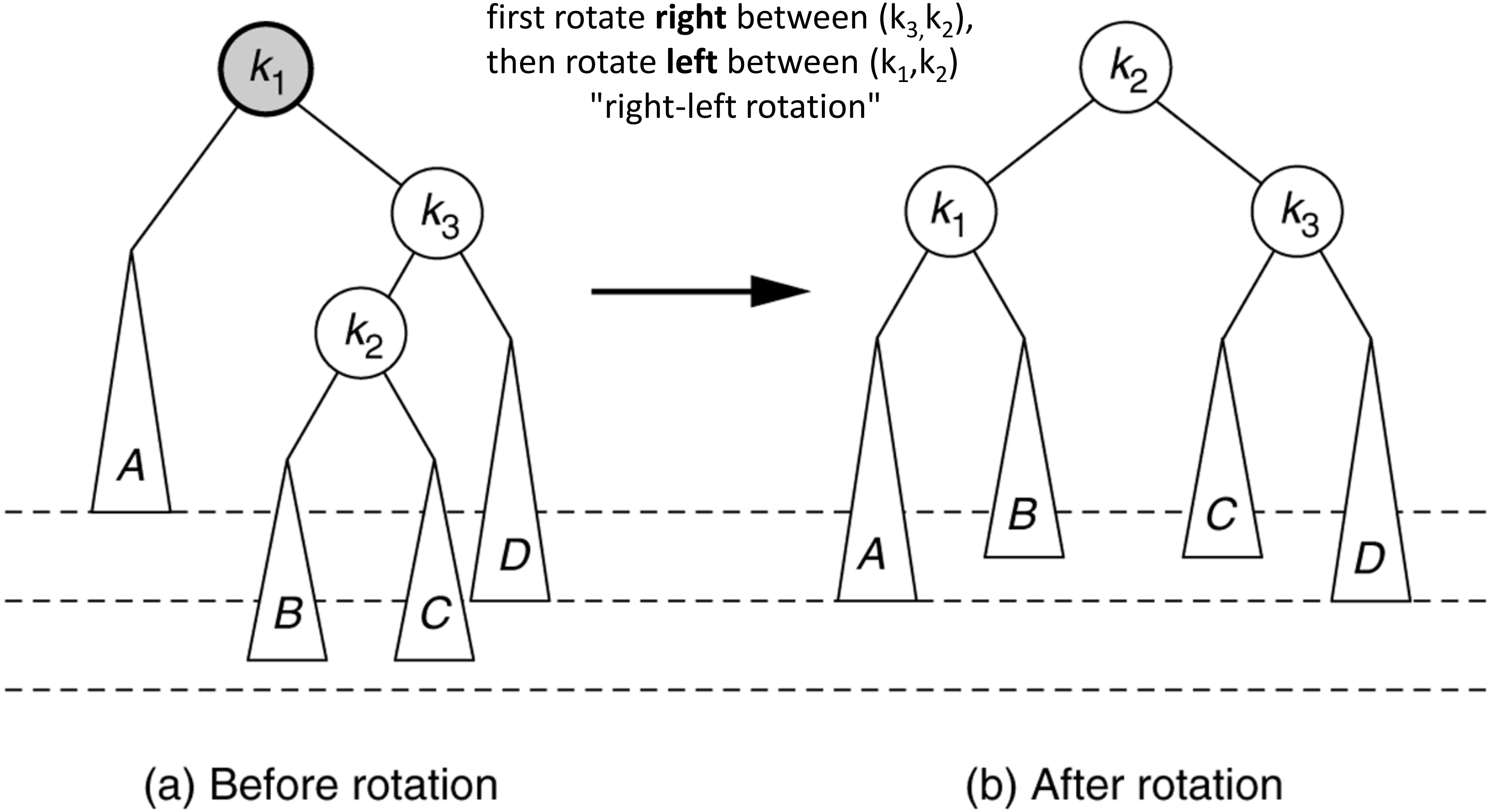


(a) Before rotation

(b) After rotation

Double Rotation (Case 3)

first rotate **right** between (k_3,k_2),
then rotate **left** between (k_1,k_2)
"right-left rotation"



Double Rotation (Case 3)

Right-left Rotation

A right-left double rotation is equivalent to a sequence of two single rotations:

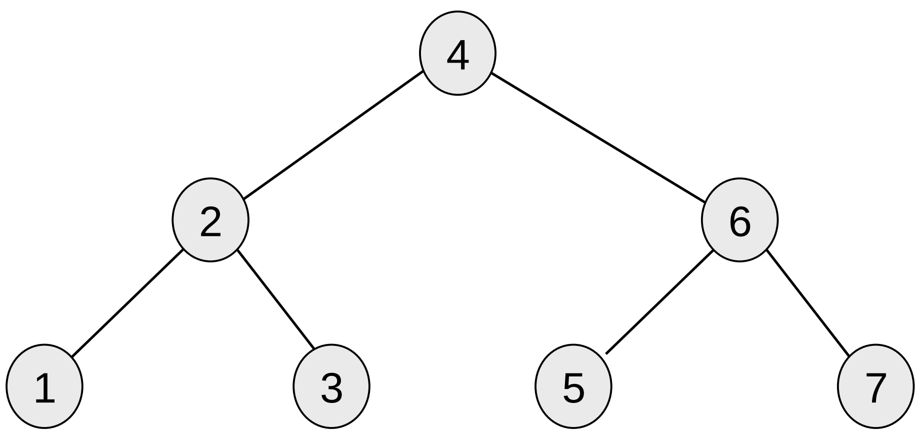
1st rotation on the original tree:
a *right* rotation between X's right-child and grandchild

2nd rotation on the new tree:
a *left* rotation between X and its new right child.

Single Rotation (HW Example 1)

Start with an empty AVL tree and insert the items 3, 2, 1, 4, 5, 6 and 7 in sequential order.

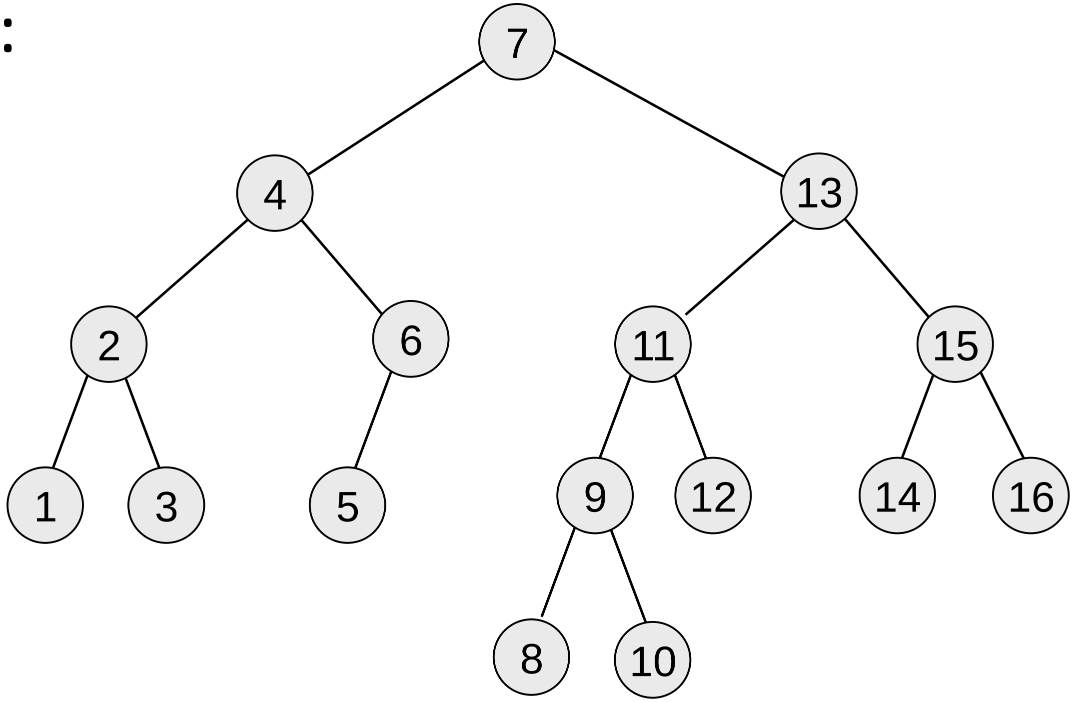
Answer (Try to work out each addition):



Double Rotation (HW Example 2)

Continue with the AVL tree of the previous example and insert the items 16, 15, 14, 13, 12, 11, 10, 8, and 9 in sequential order.

Answer:



Node Deletion

Node is deleted from the BST as we have seen before. Then based on the following rules, (if tree is unbalanced) we do rotation:

Let p be the node to be deleted.

Let X be the first unbalanced ancestor node of p.

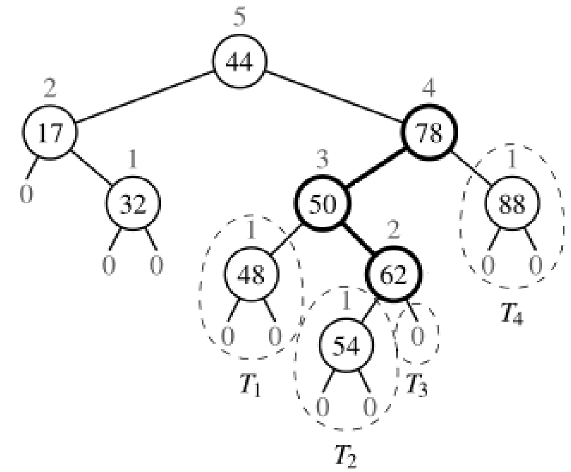
Let q be the taller child of p.

Let r be the taller child of q. If children of q have the same height, then r should be at the same side with q (if q is a right child, then r is the right, vice versa.)

<u>Example:</u>

Node to be deleted is 32.

p: 32, X: 44, q: 78, r: 50

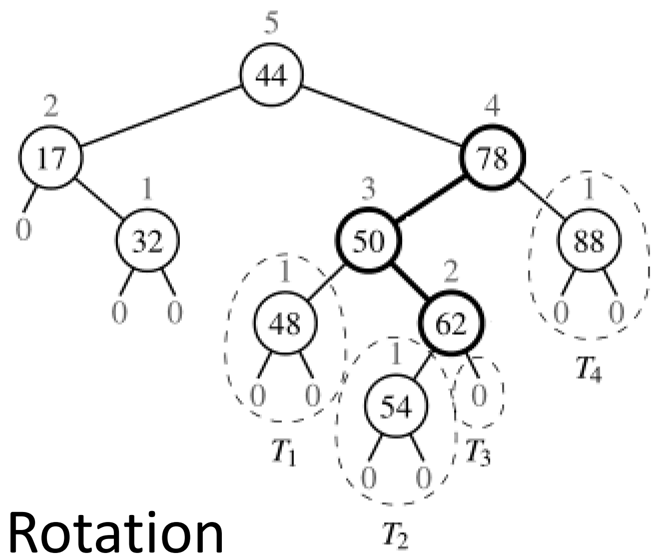


Node Deletion

Example:

Node to be deleted is 32.

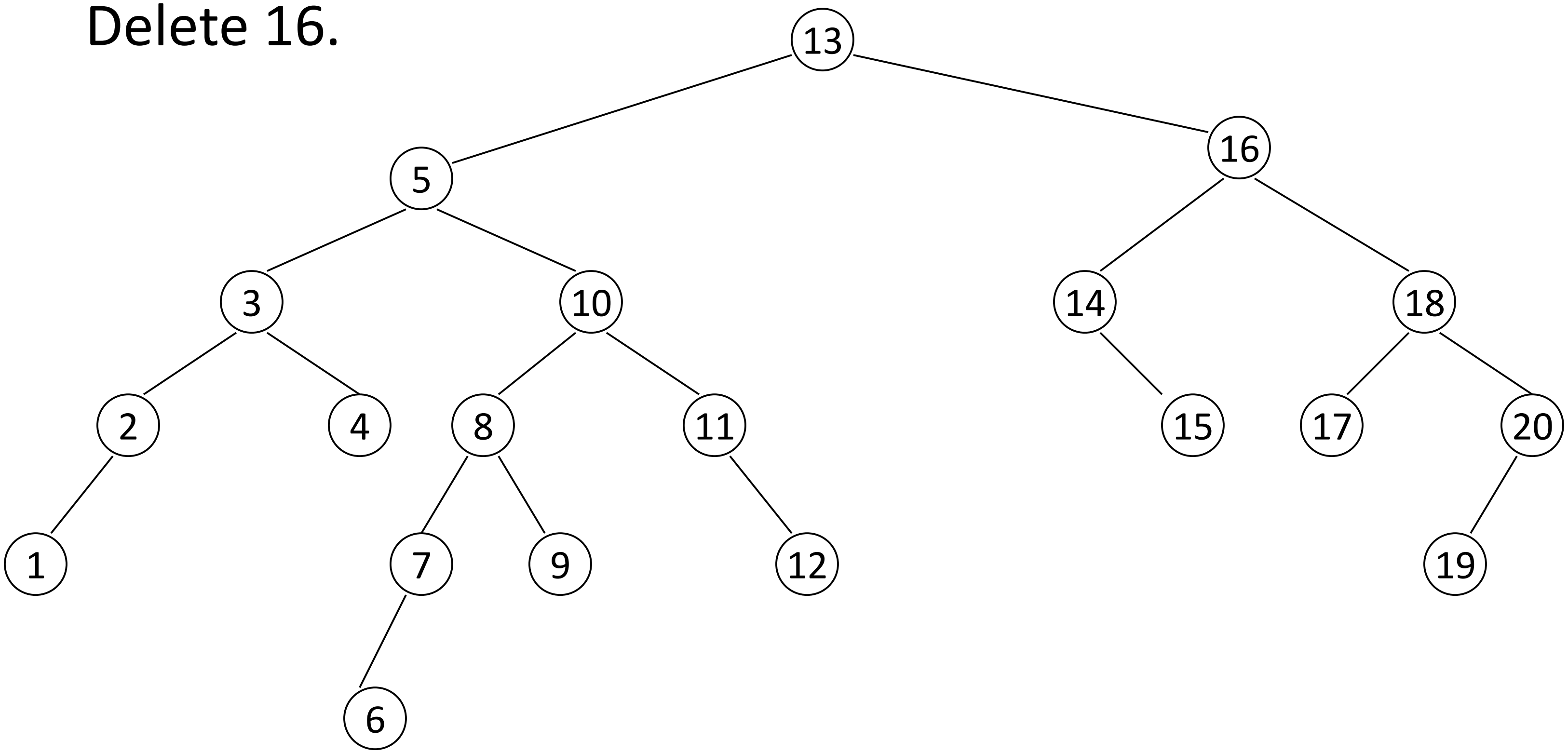
p: 32, X: 44, q: 78, r: 50



r and q both right children	Single Rotation
r and q both left children	Single Rotation
r is left, q is right child	Double Rotation
r is right, q is left child	Double Rotation

Node Deletion

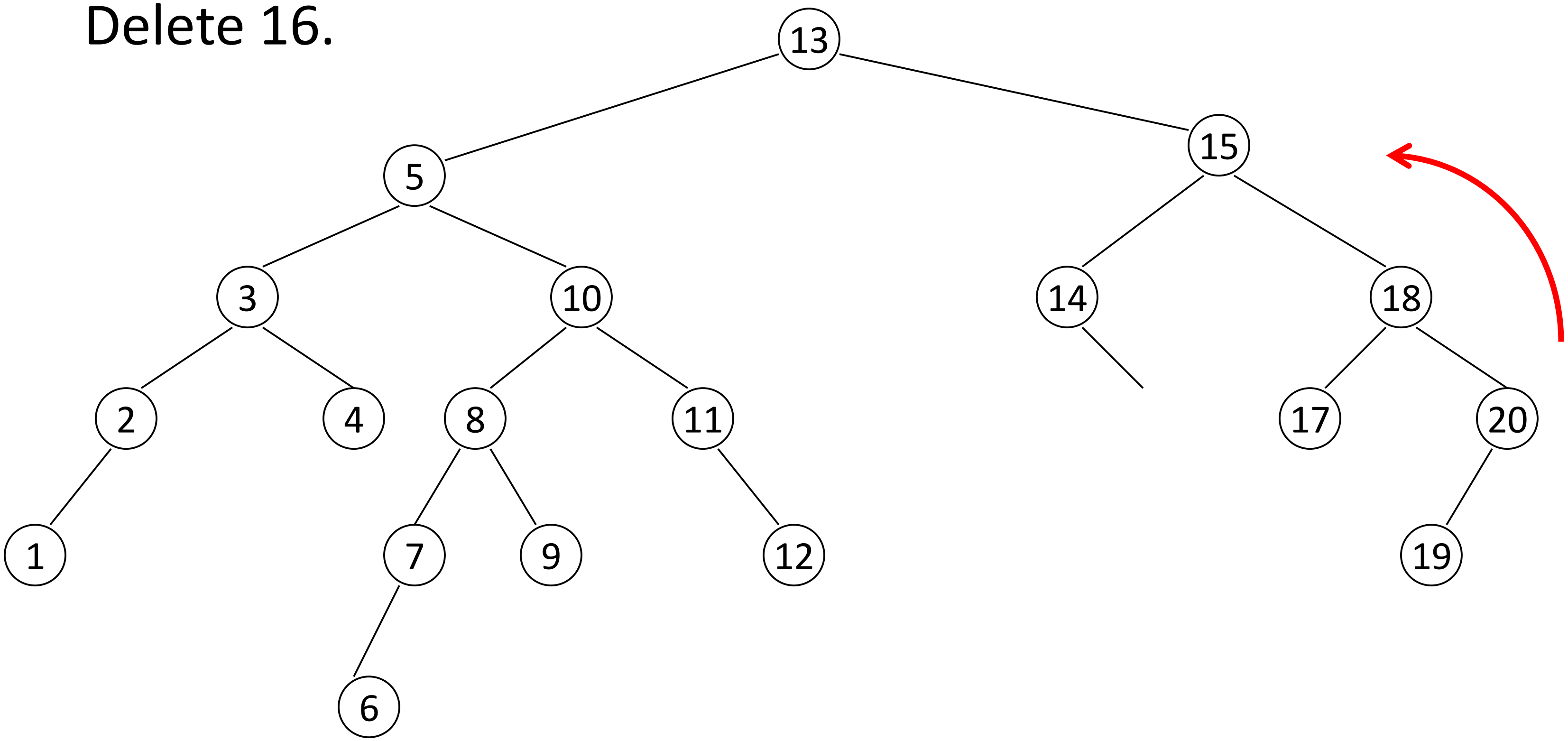
Delete 16.



p: 16, X: 16, q: 18, r: 20

Node Deletion

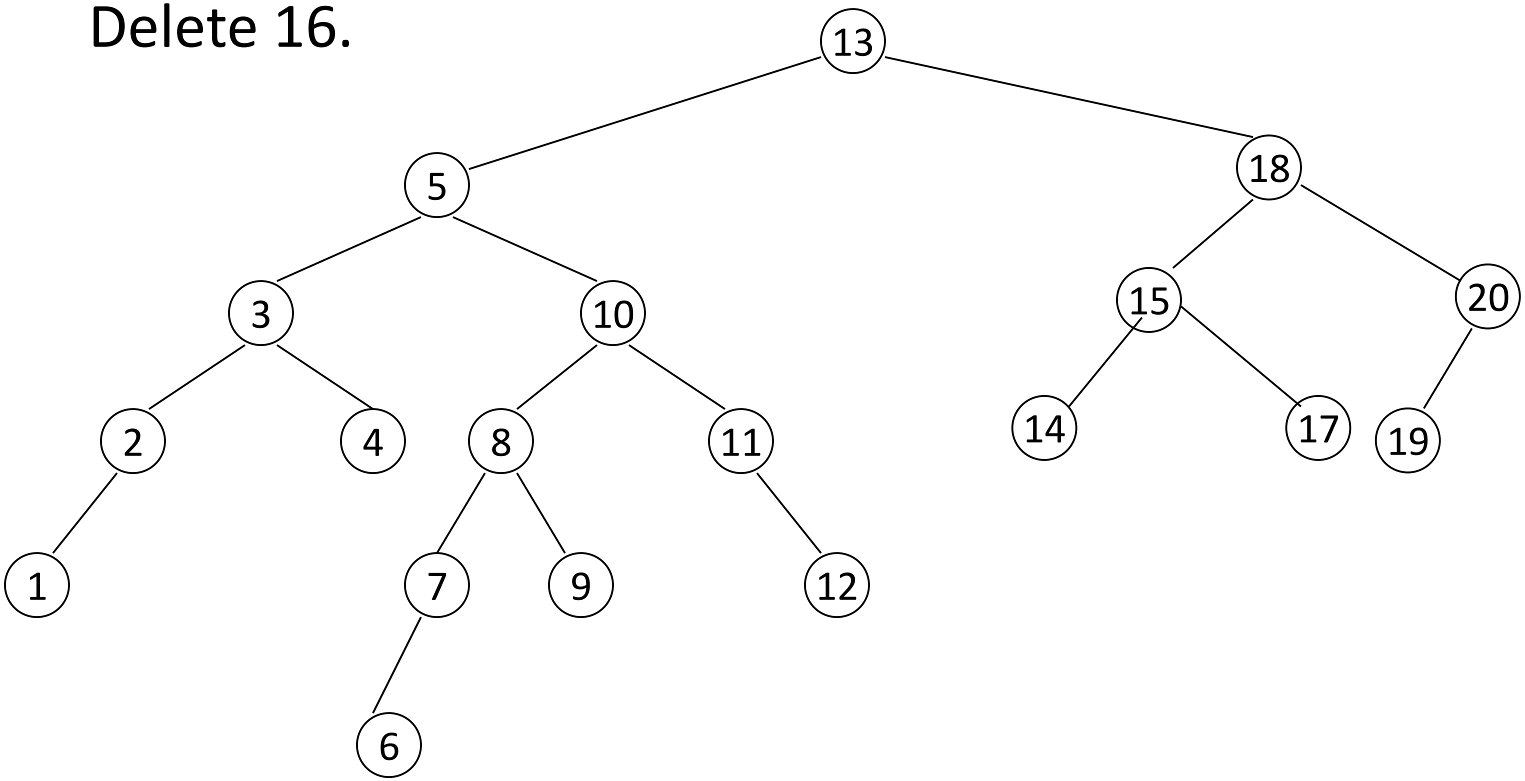
Delete 16.



p: 16, X: 16, q: 18, r: 20

Node Deletion

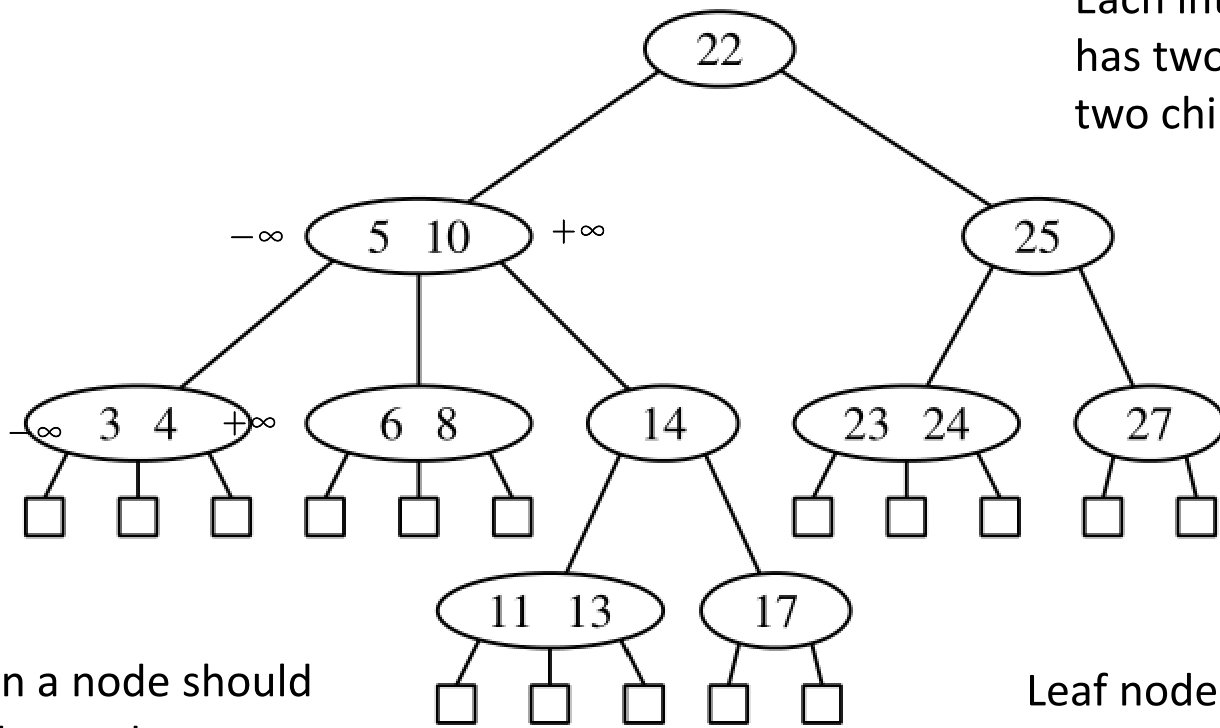
Delete 16.



p: 16, X: 16, q: 18, r: 20

Multiway Trees

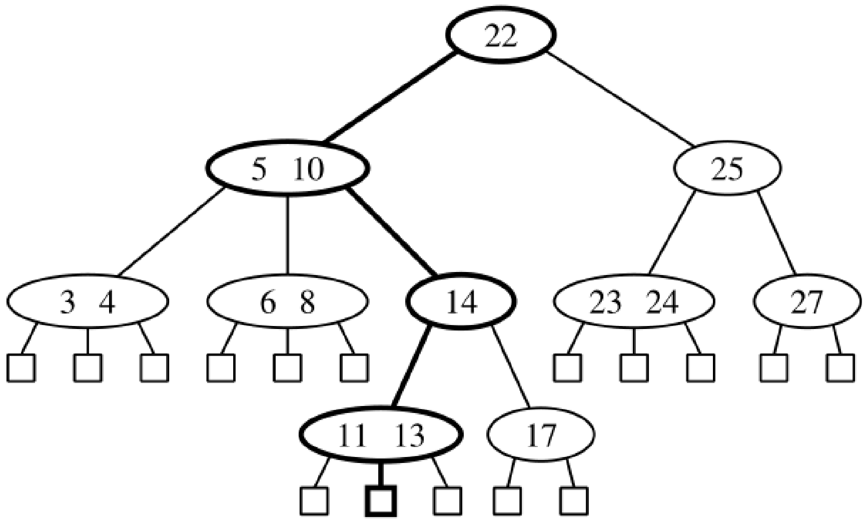
Each internal node has two have at least two children.



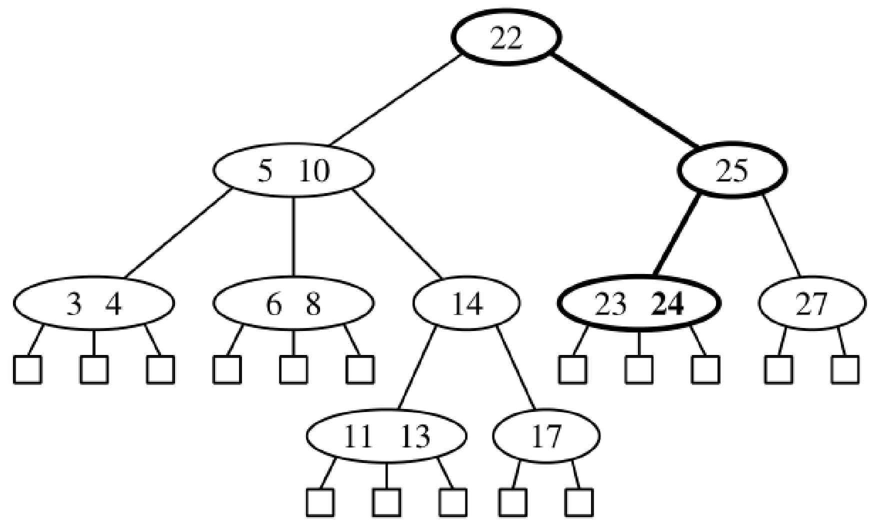
All keys in a node should be in-between two consecutive key pairs (do not forget the fictitious -/+ infinity keys.).

Leaf nodes do not have any key.

Multiway Trees



Unsuccessful Search

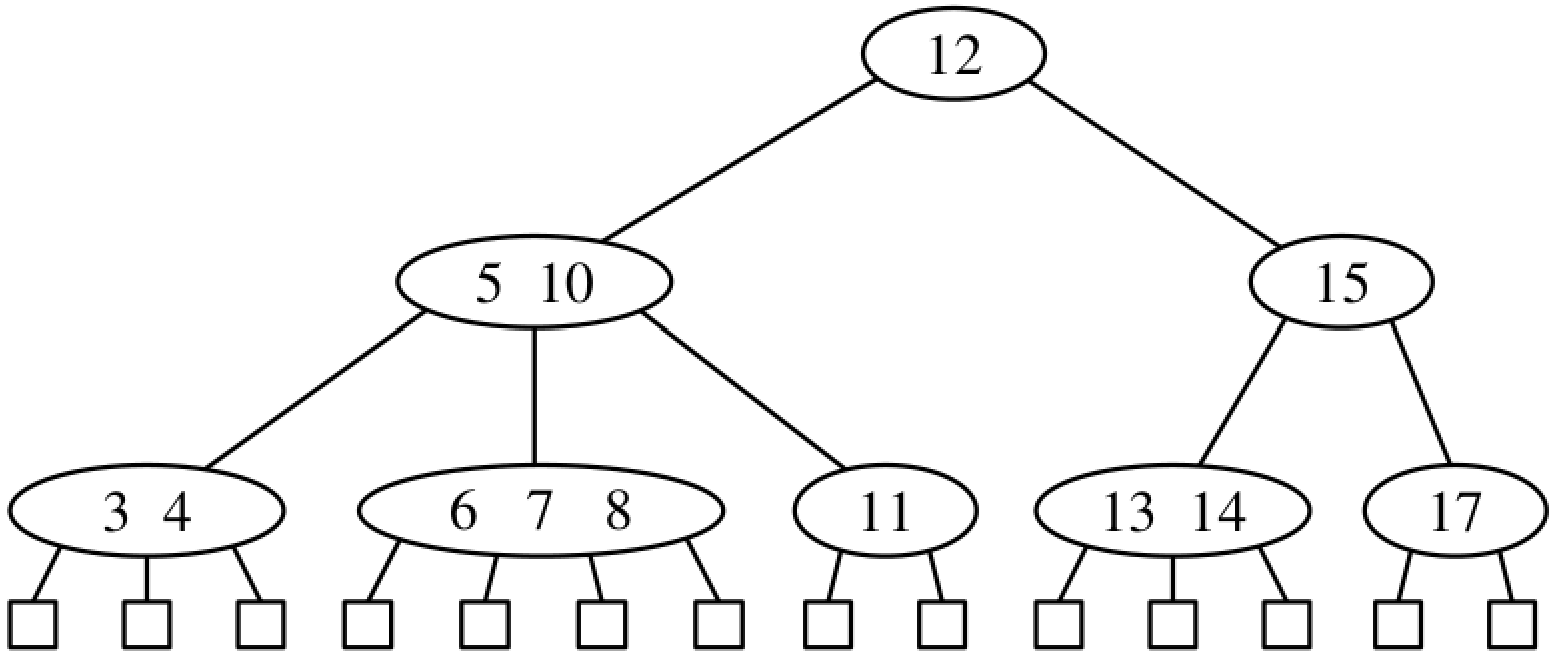


Successful Search

(2,4) Trees

A special type of multiway trees.

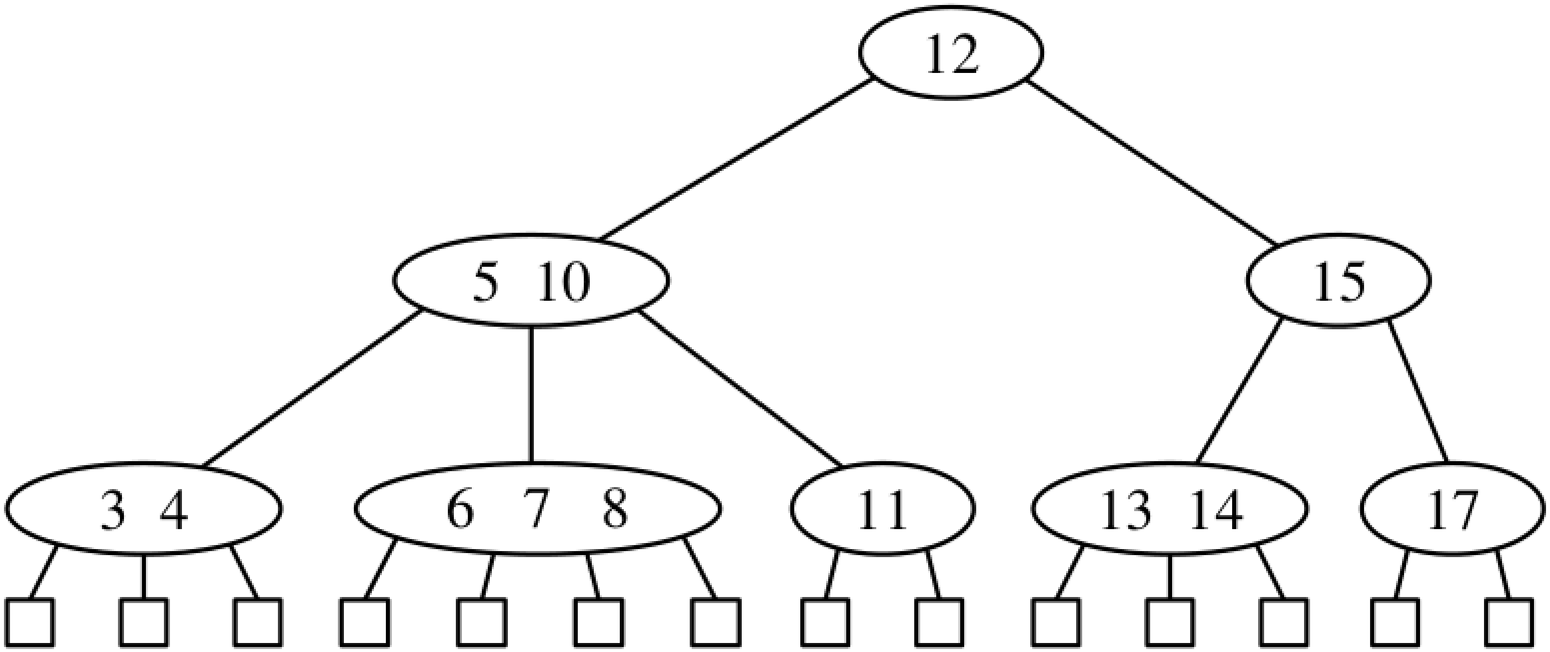
Sometimes called 2-4 tree, 2-3-4 tree.



(2,4) Trees

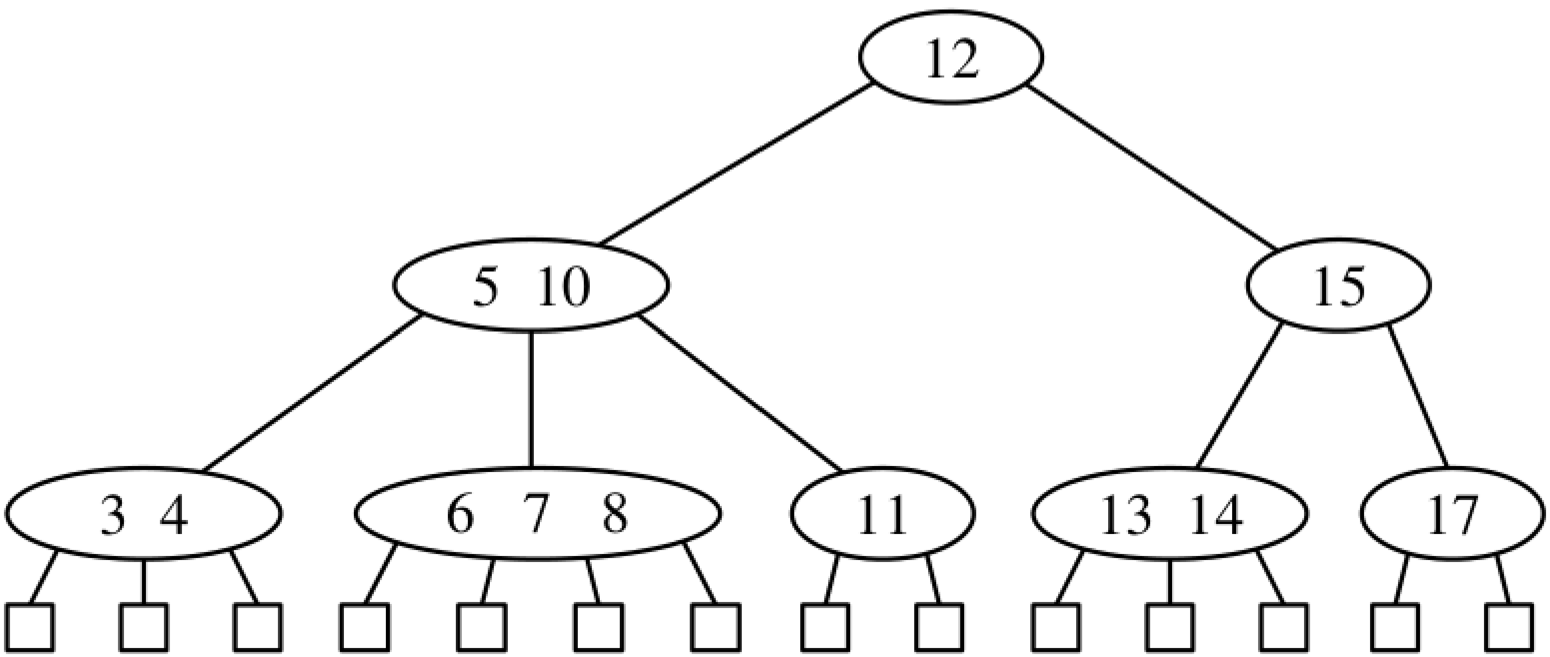
Size Property: Every external node has at most 4 children.

Depth Property: All the external nodes have the same depth.



(2,4) Trees

The height of a (2,4) tree storing n items is O(logn).



(2,4) Trees

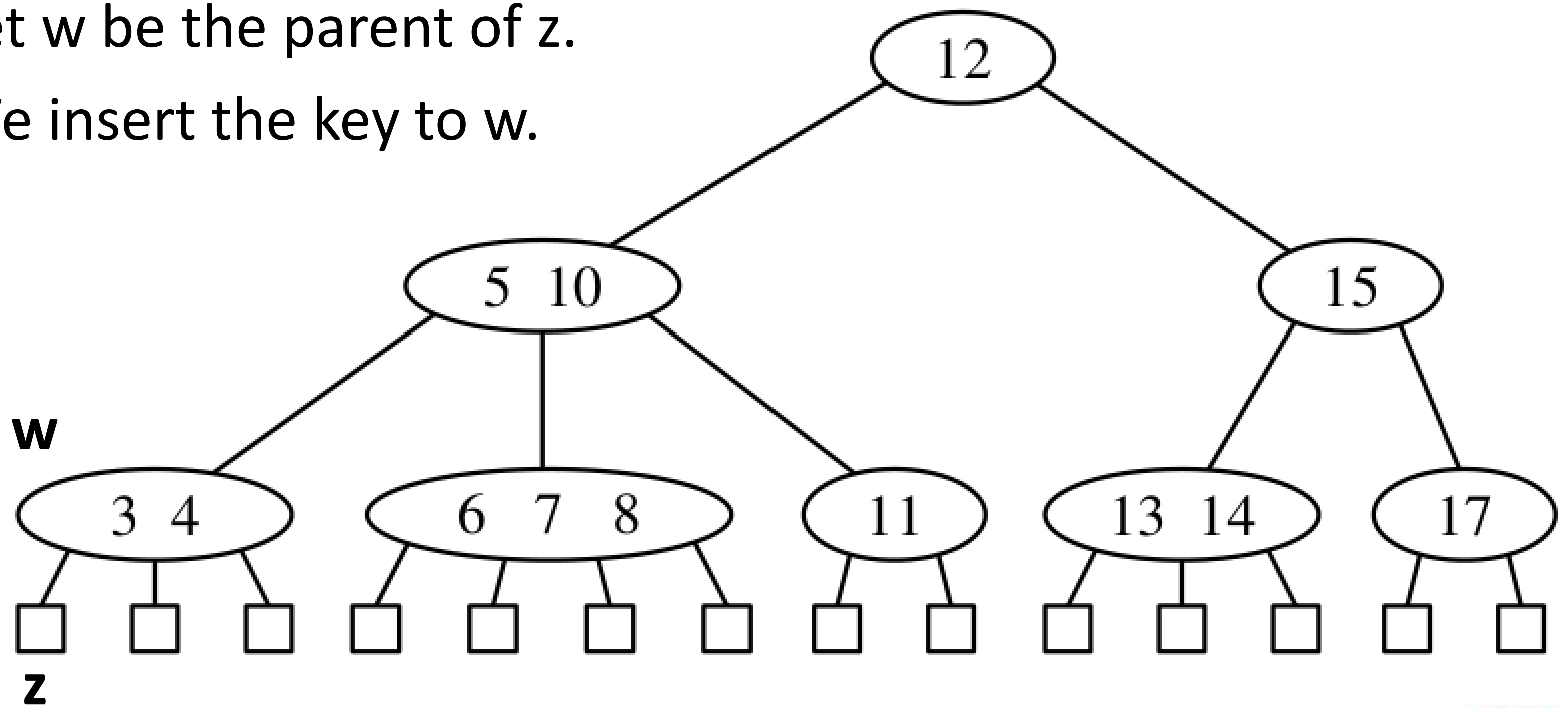
Insert item k.

Search the item k in the tree.

Let the unsuccessful search end at node z.

Let w be the parent of z.

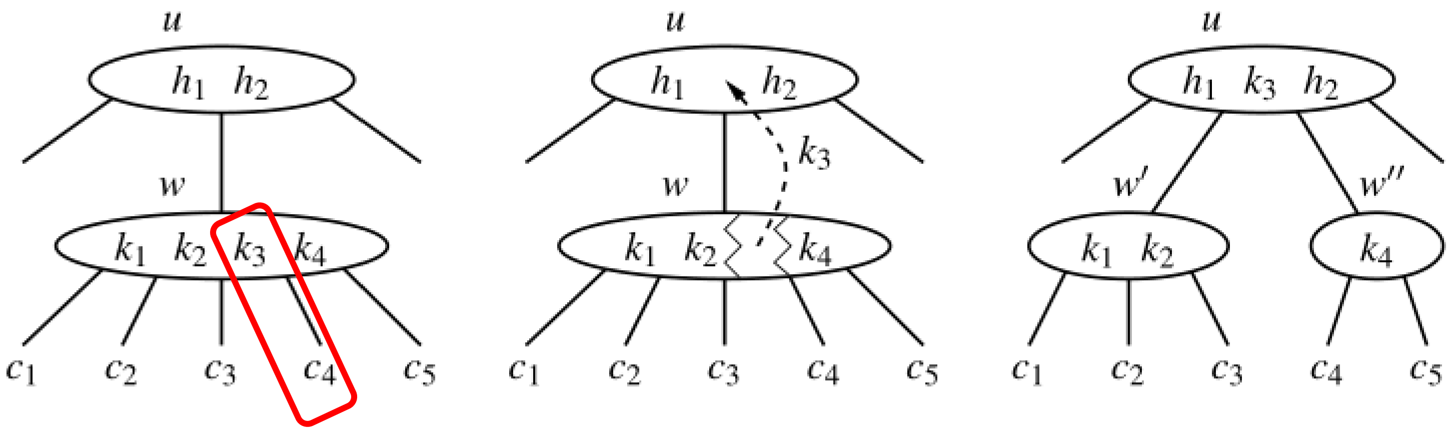
We insert the key to w.



(2,4) Trees

Insert item k.

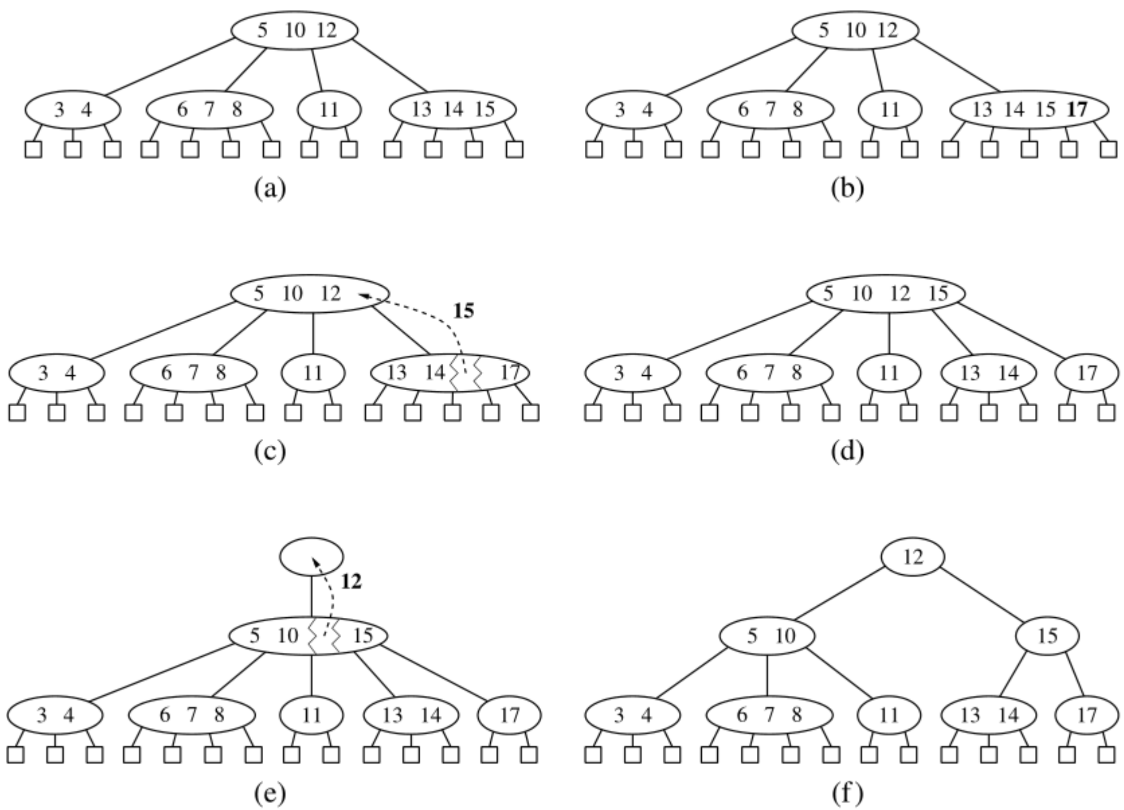
If, after insert, size property is violated, then node split should occur.



(2,4) Trees

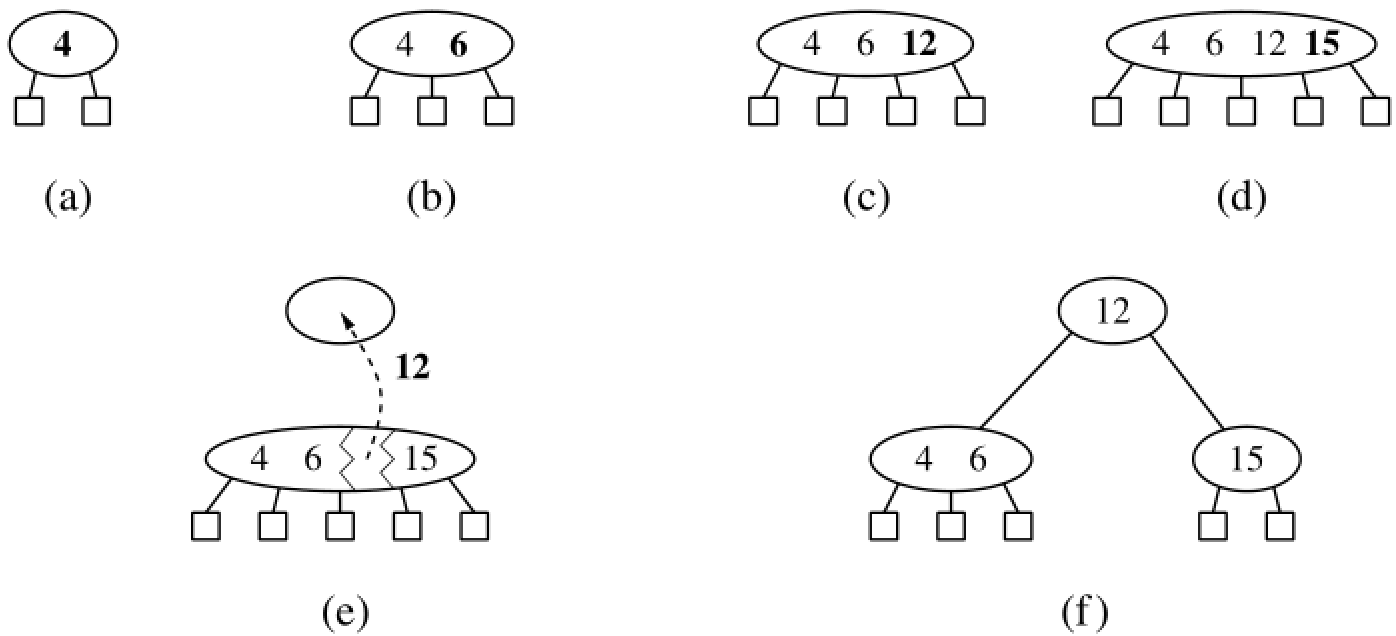
Insert item k.

Split operation may continue all the way up to the root.



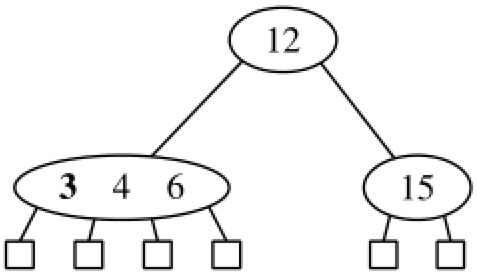
(2,4) Trees

Insert Example

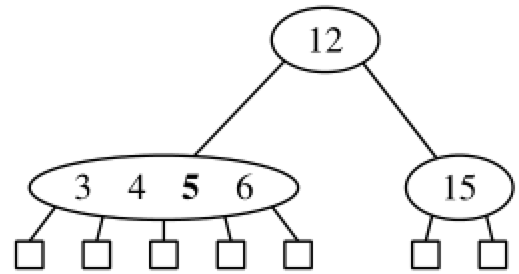


(2,4) Trees

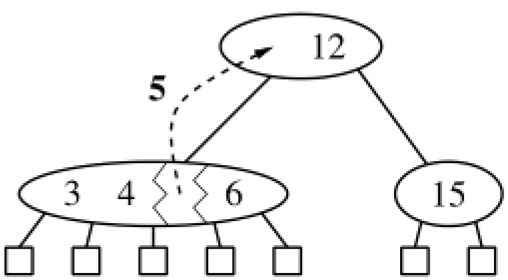
Insert Example



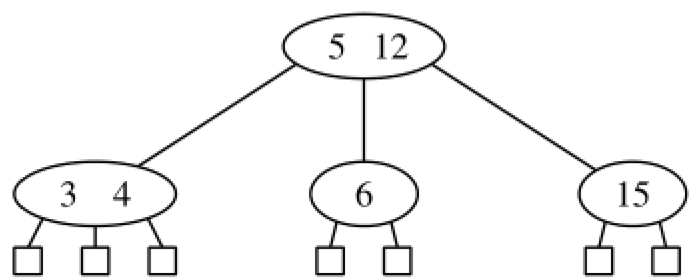
(g)



(h)



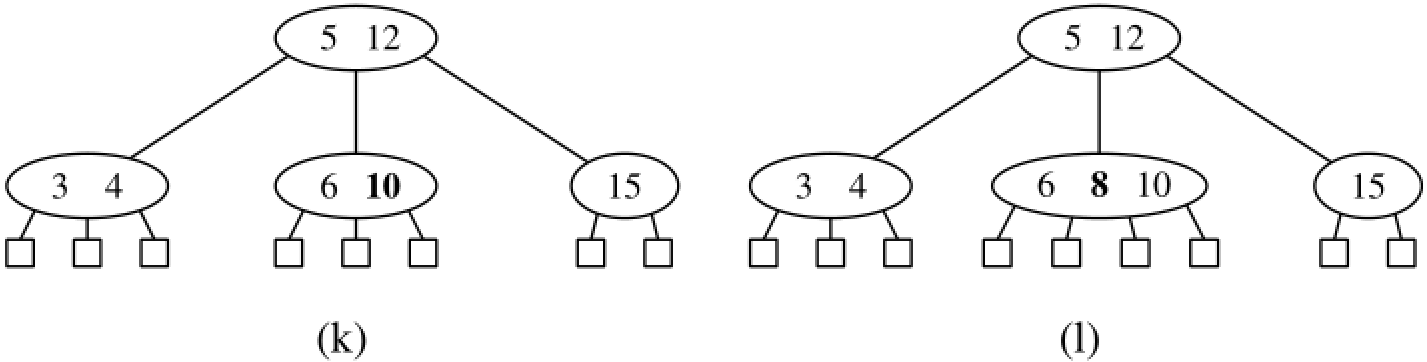
(i)



(j)

(2,4) Trees

Insert Example



Insertion is O(logn).

- Searching item is O(logn).
- Inserting key to the node is O(1).
- Splitting can elevate at most only up to the root O(logn).