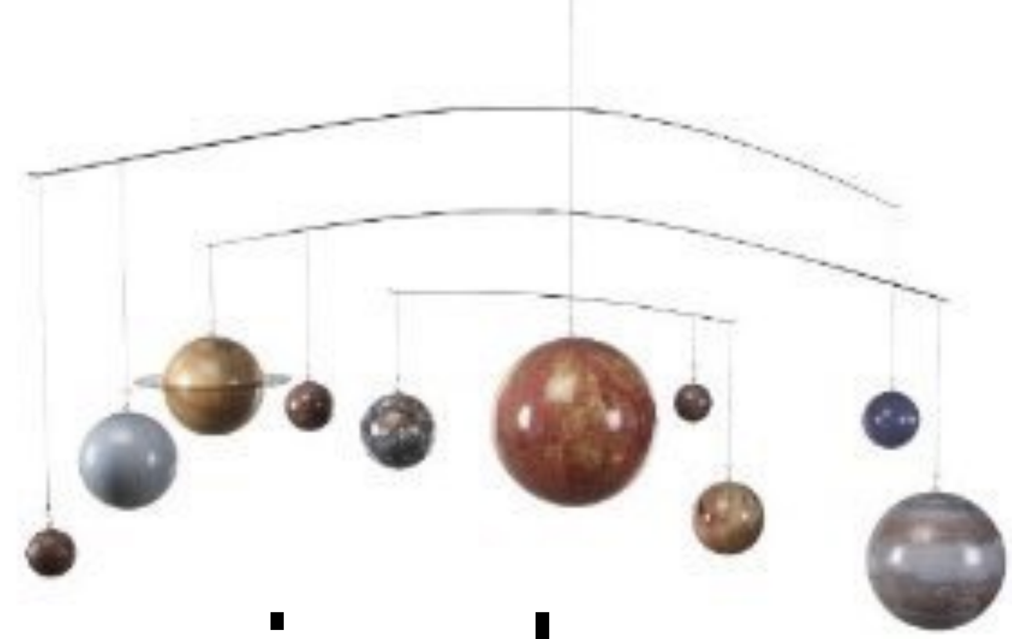


Data Structures in Java

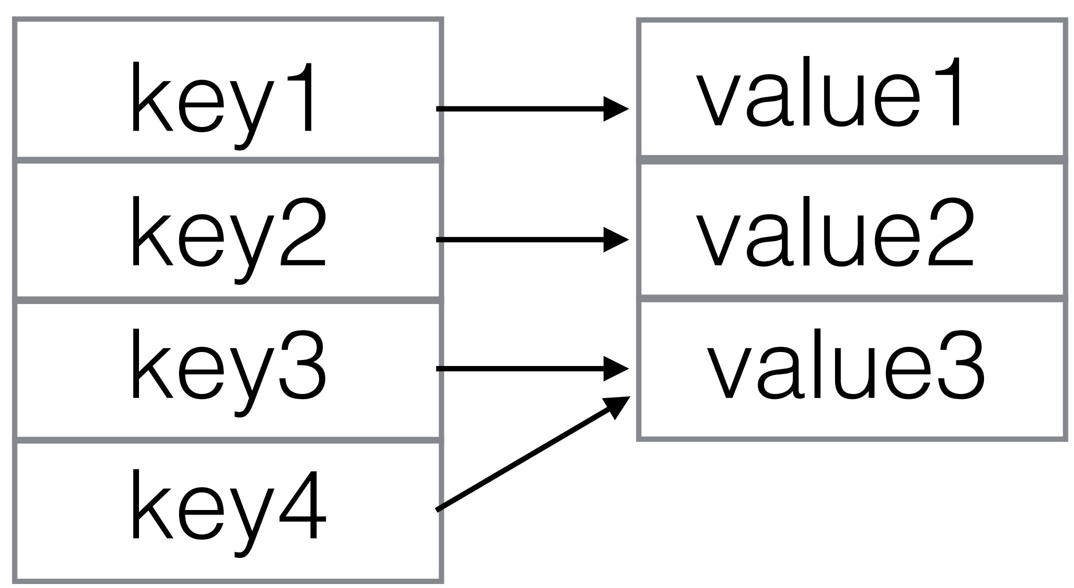
Lecture 9:
Self-Balancing Search Trees: AVL Trees

10/25/21

Daniel Bauer

Map ADT

- A *map* is collection of *(key, value)* pairs.
- Keys are unique, values need not be.
- Two operations:
  - `get(key)` returns the value associated with this key
  - `put(key, value)` (overwrites existing keys)



How do we implement map operations efficiently?

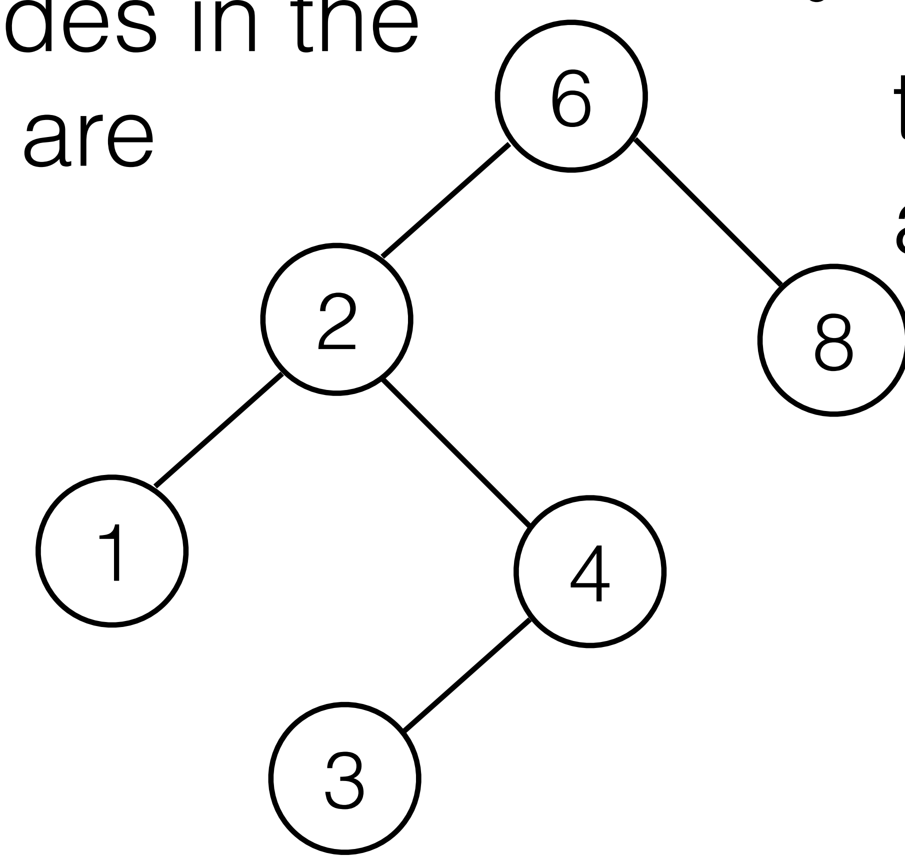
Binary Search Tree Property

- Goal: Reduce finding an item to O(log N)

- For every node n with key x

- the key of all nodes in the left subtree of n are smaller than x.

- The key of all nodes in the right subtree of n are larger than x.



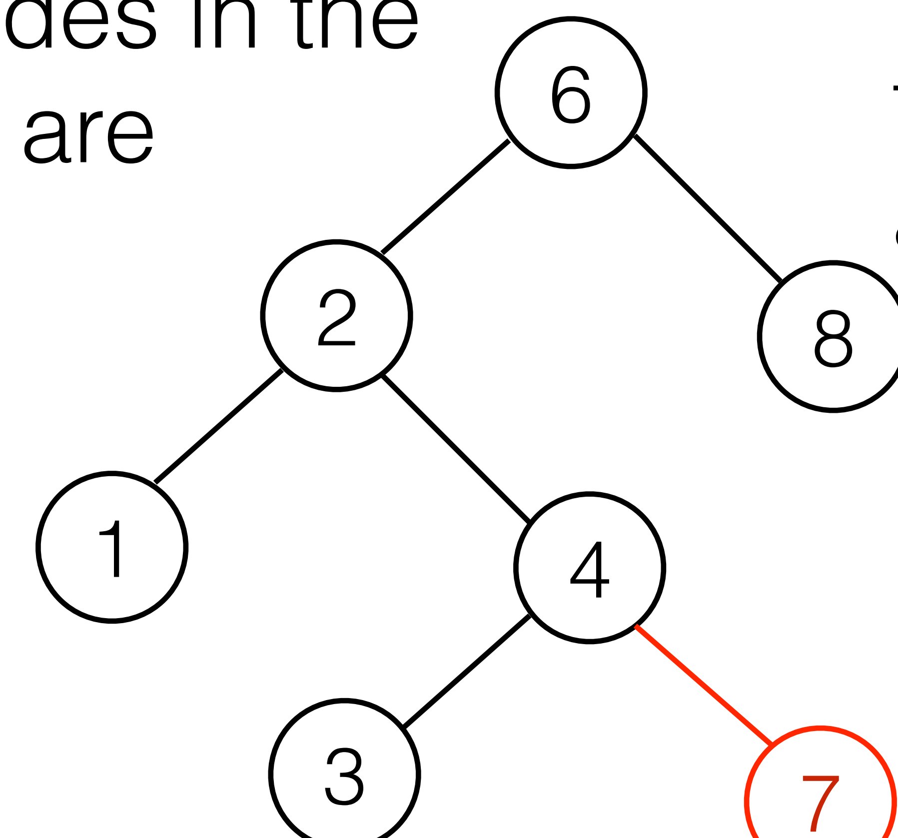
Binary Search Tree Property

- Goal: Reduce finding an item to O(log N)

- For every node n with key x

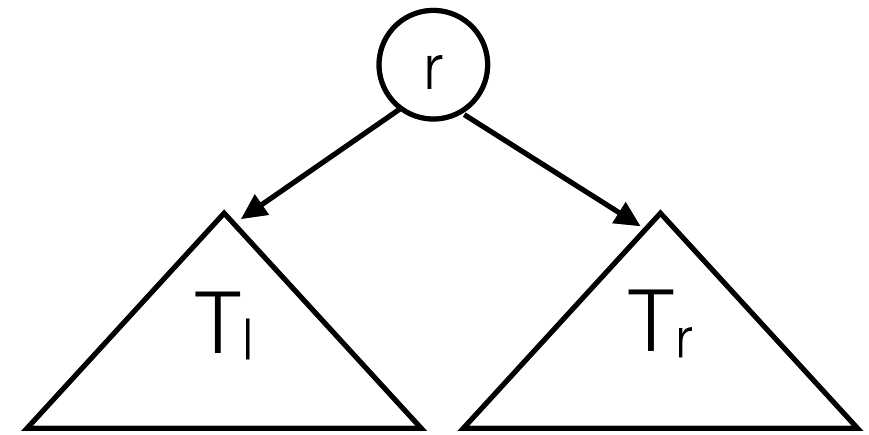
- the key of all nodes in the left subtree of n are smaller than x.

- The key of all nodes in the right subtree of n are larger than x.



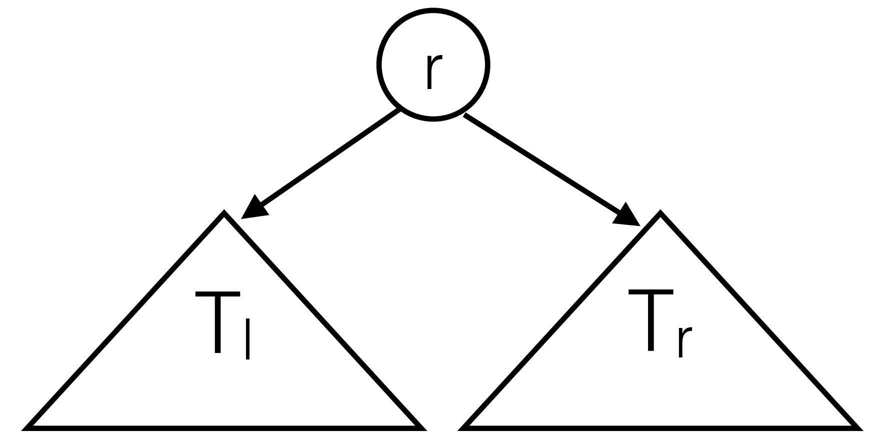
This is not a search tree

Binary Search Tree (BST) ADT



- A *Binary Search Tree T* consists of
    - A root node *r* with key r_{item}
    - *A*t most two non-empty subtrees T_l and T_r, connected by a directed edge from r.

Binary Search Tree (BST) ADT



- A *Binary Search Tree T* consists of
  - A root node *r* with key r_{item}
  - *A*t most two non-empty subtrees T_l and T_r, connected by a directed edge from r.

- T_l and T_r satisfy the BST property:
  - For all nodes *s* in T_l, $s_{item} < r_{item}$.
  - *F*or all nodes *t* in T_r, $t_{item} > r_{item}$.

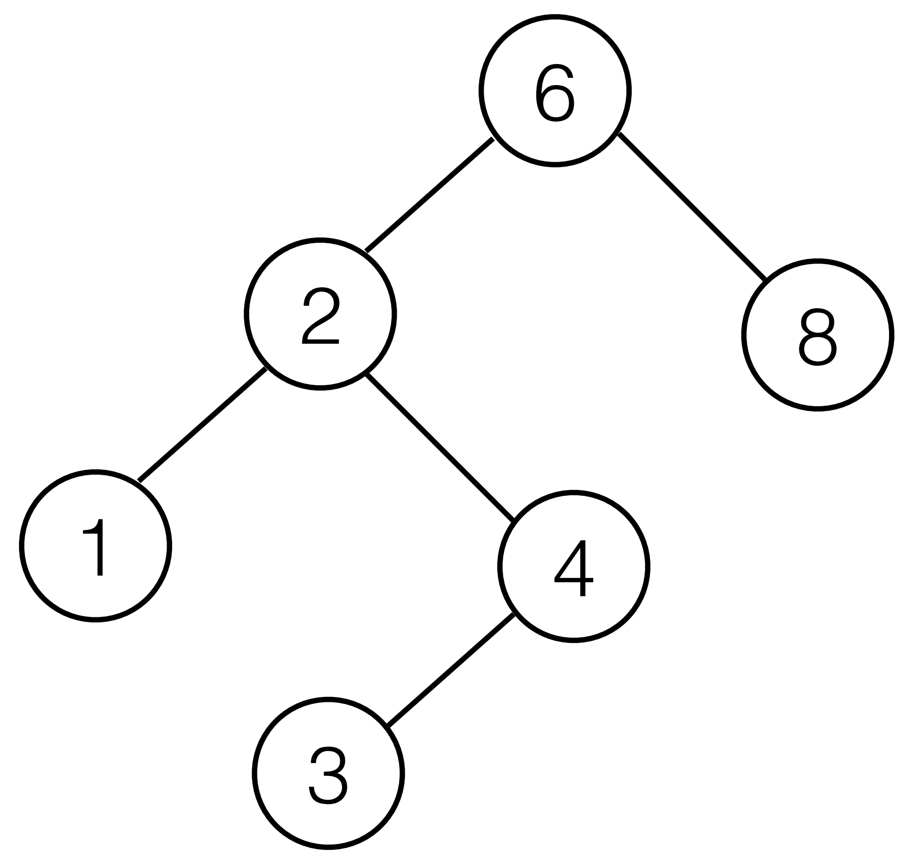
- No key appears more than once in the BST.

BST operations

- `insert(x)` - add key x to T.
- `contains(x)` - check if key x is in T.
- `findMin()` - find smallest key in T.
- `findMax()` - find largest key in T.
- `remove(x)` - remove a key from T.

BST operations: `contains`

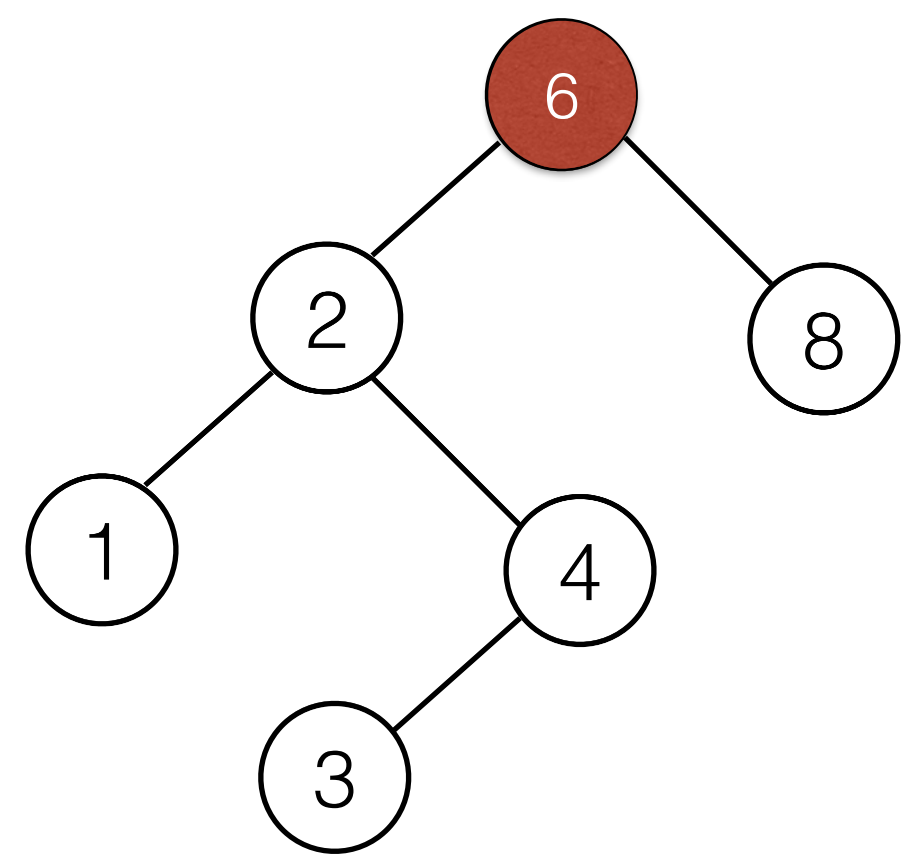
```
private boolean contains( Integer x, BinaryNode t ) {
     if( t == null )
       return false;

    if( x < t.data )
        return contains( x, t.left );
    else if( t.data < x )
        return contains( x, t.right );
    else
        return true;    // Match
}
```



BST operations: `contains`

```
private boolean contains( Integer x, BinaryNode t ) {
     if( t == null )
       return false;

    if( x < t.data )
        return contains( x, t.left );
    else if( t.data < x )
        return contains( x, t.right );
    else
        return true;    // Match
}
```

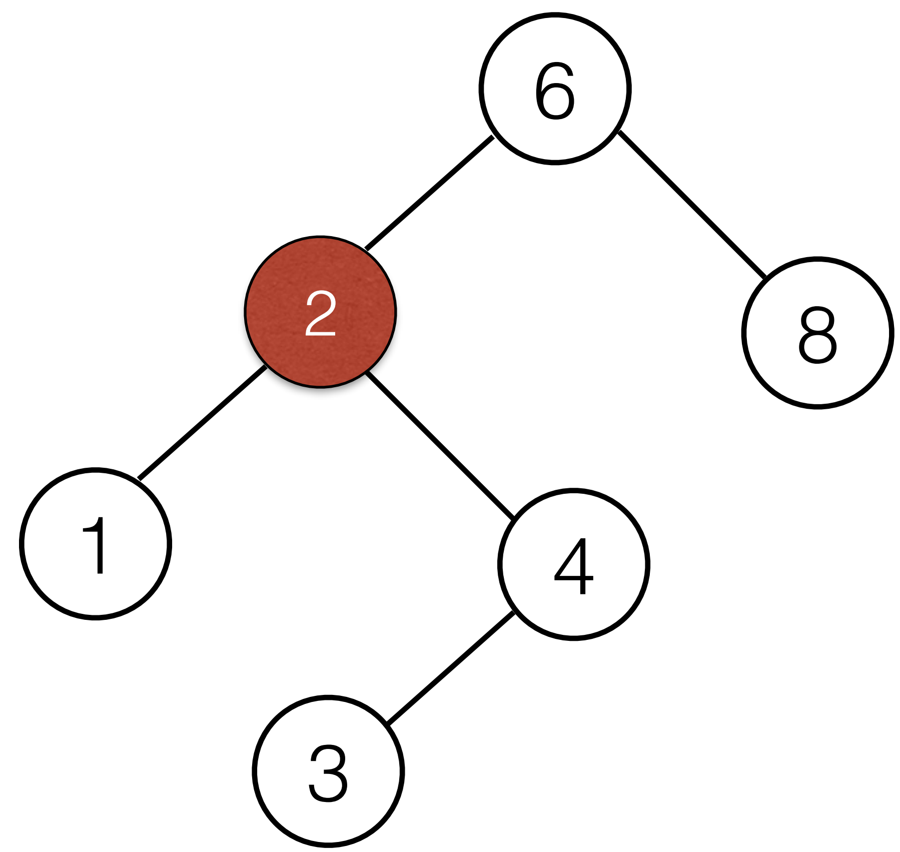
`contains(3)`



BST operations: `contains`

```
private boolean contains( Integer x, BinaryNode t ) {
    if( t == null )
      return false;

   if( x < t.data )
      return contains( x, t.left );
   else if( t.data < x )
      return contains( x, t.right );
   else
      return true;    // Match
}
```

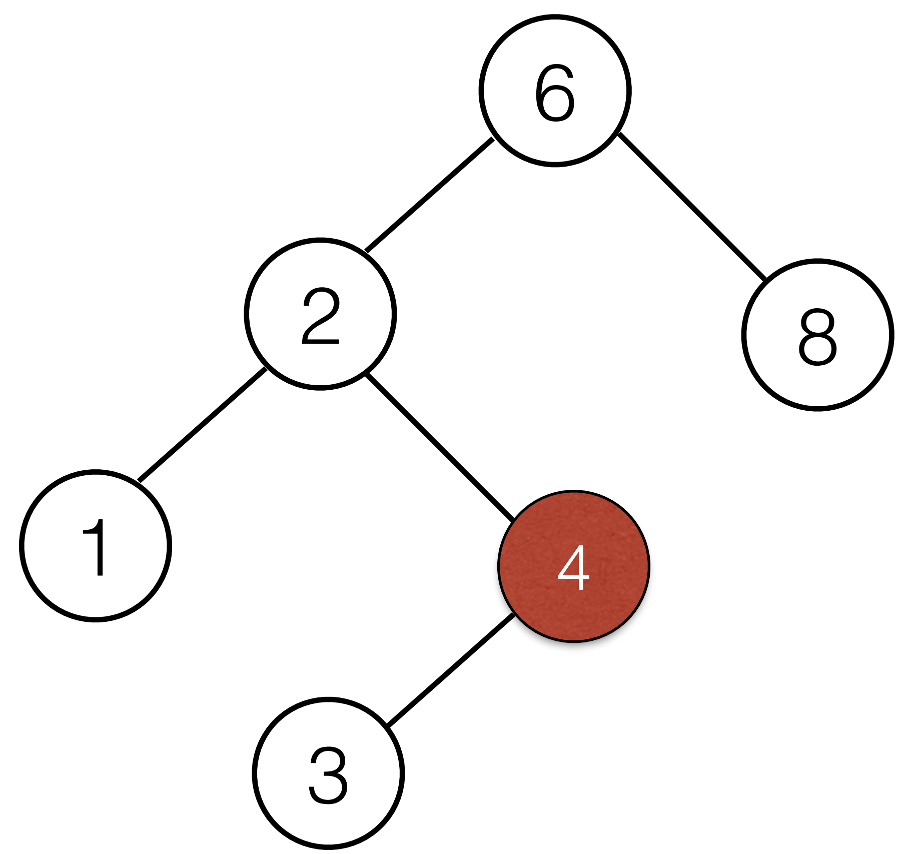
`contains(3)`



BST operations: `contains`

```
private boolean contains( Integer x, BinaryNode t ) {
    if( t == null )
      return false;

    if( x < t.data )
        return contains( x, t.left );
    else if( t.data < x )
        return contains( x, t.right );
    else
        return true;    // Match
}
```

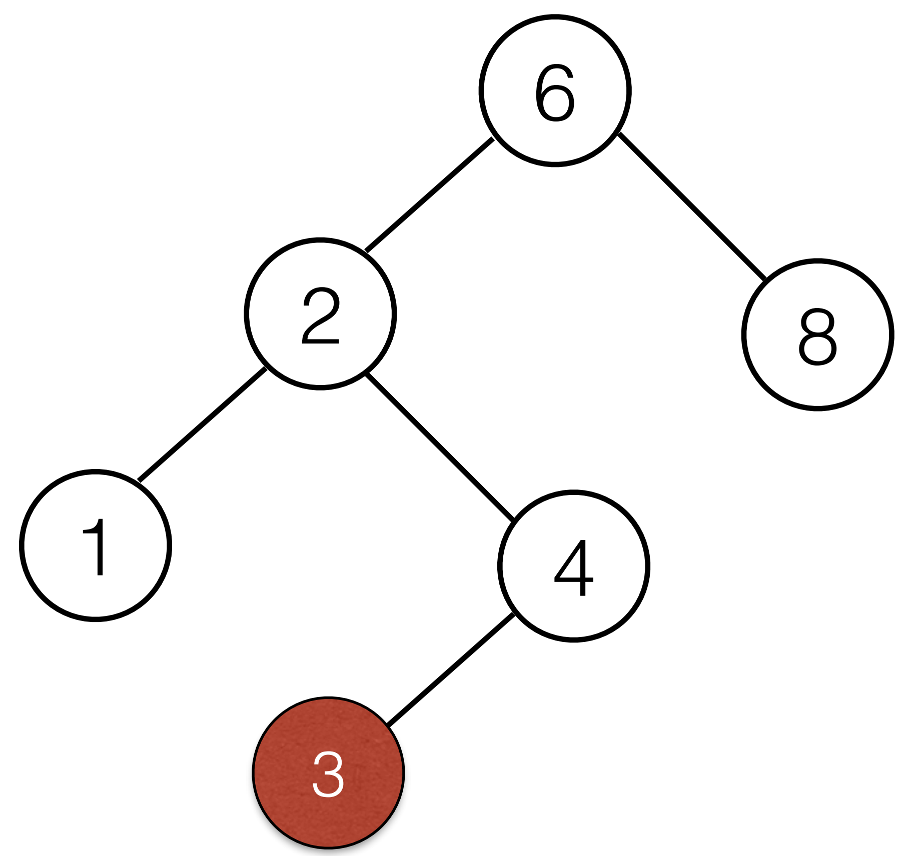
`contains(3)`



BST operations: `contains`

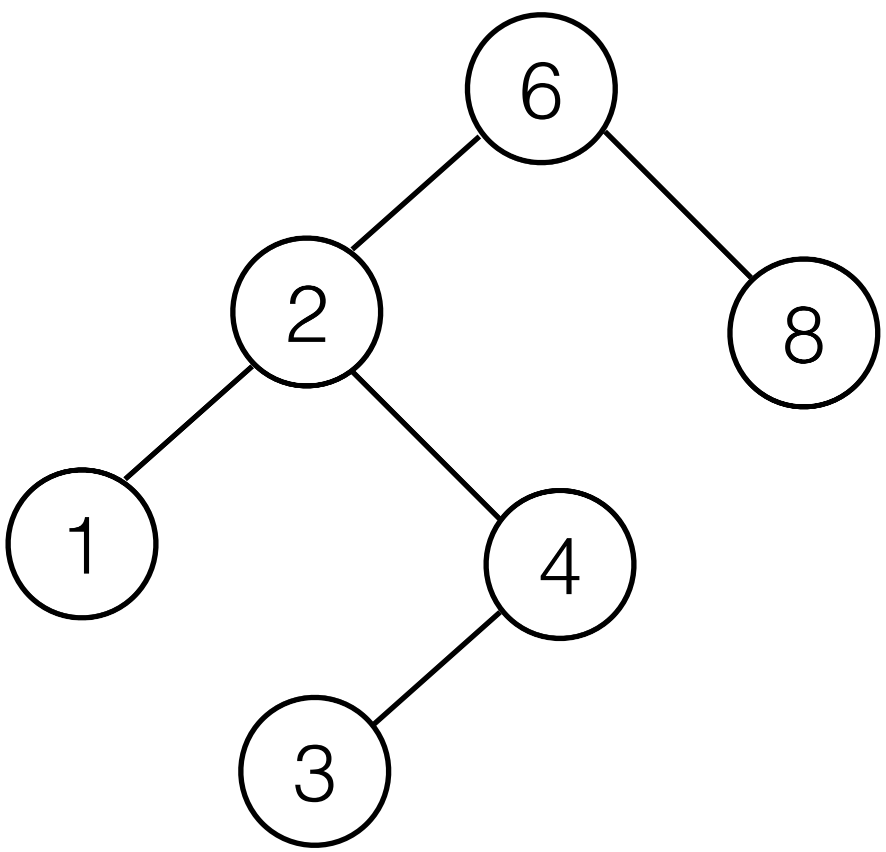
```
private boolean contains( Integer x, BinaryNode t ) {
    if( t == null )
      return false;

   if( x < t.data )
      return contains( x, t.left );
   else if( t.data < x )
      return contains( x, t.right );
   else
      return true;    // Match
}
```

`contains(3)`



BST operations: `findMin`

```
private BinaryNode findMin( BinaryNode t ) {
    if( t == null )
        return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}
```

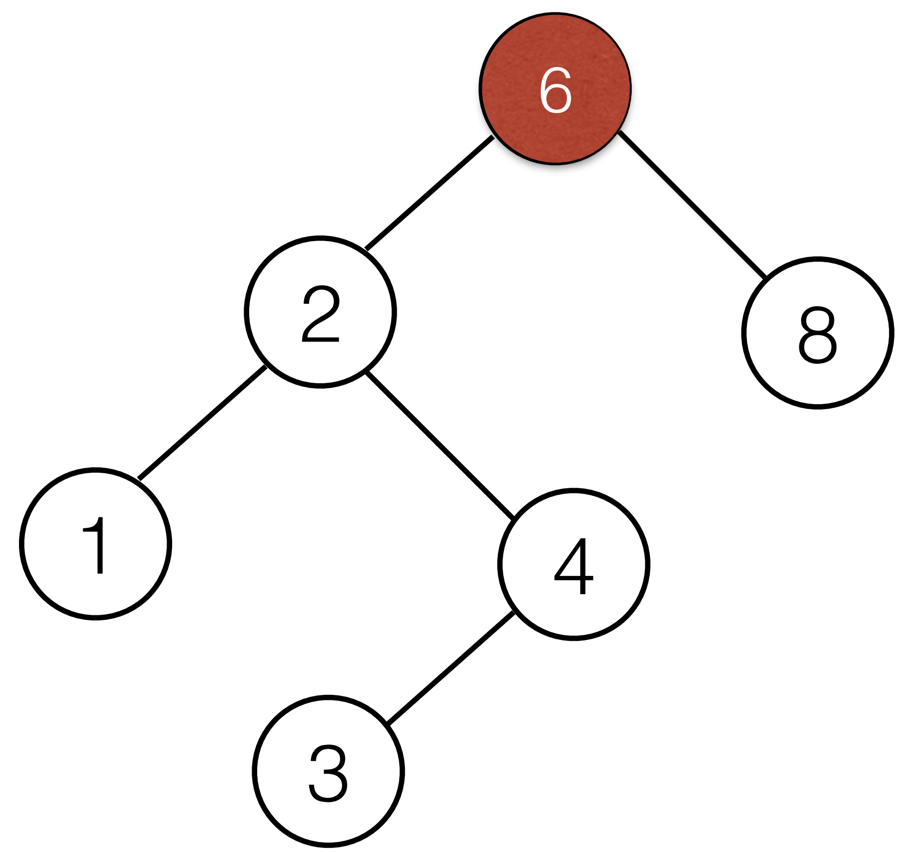


findMax is equivalent.

BST operations: `findMin`

```
private BinaryNode findMin( BinaryNode t ) {
    if( t == null )
        return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}
```

`findMin()`

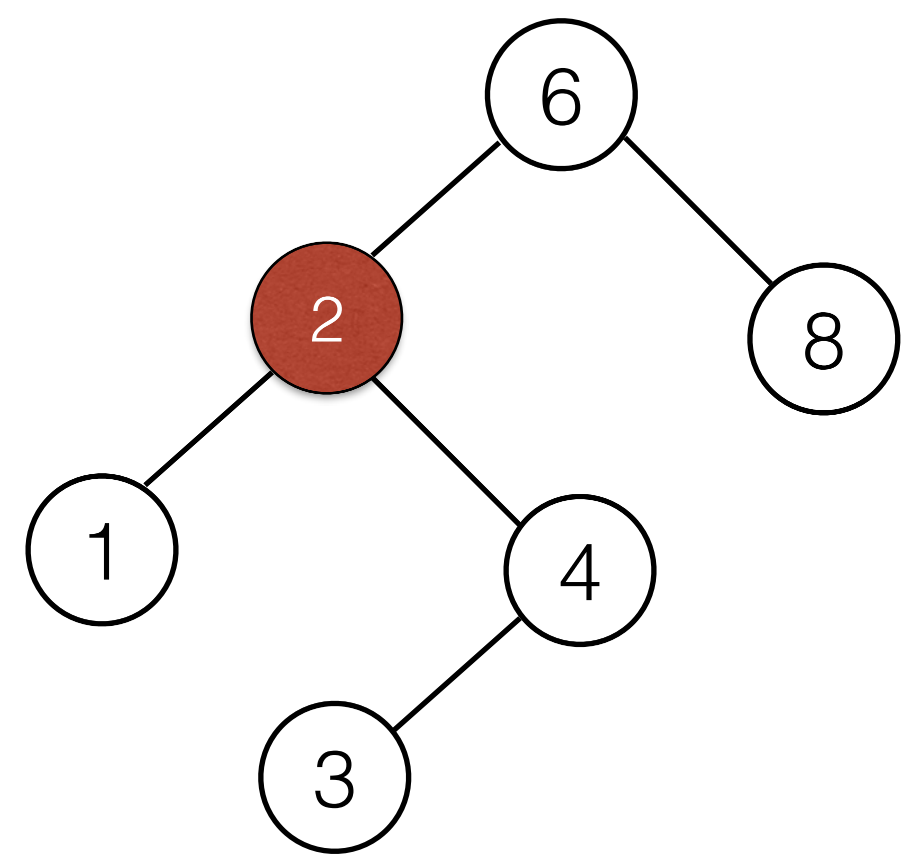


findMax is equivalent.

BST operations: `findMin`

```
private BinaryNode findMin( BinaryNode t ) {
    if( t == null )
        return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}
```

`findMin()`

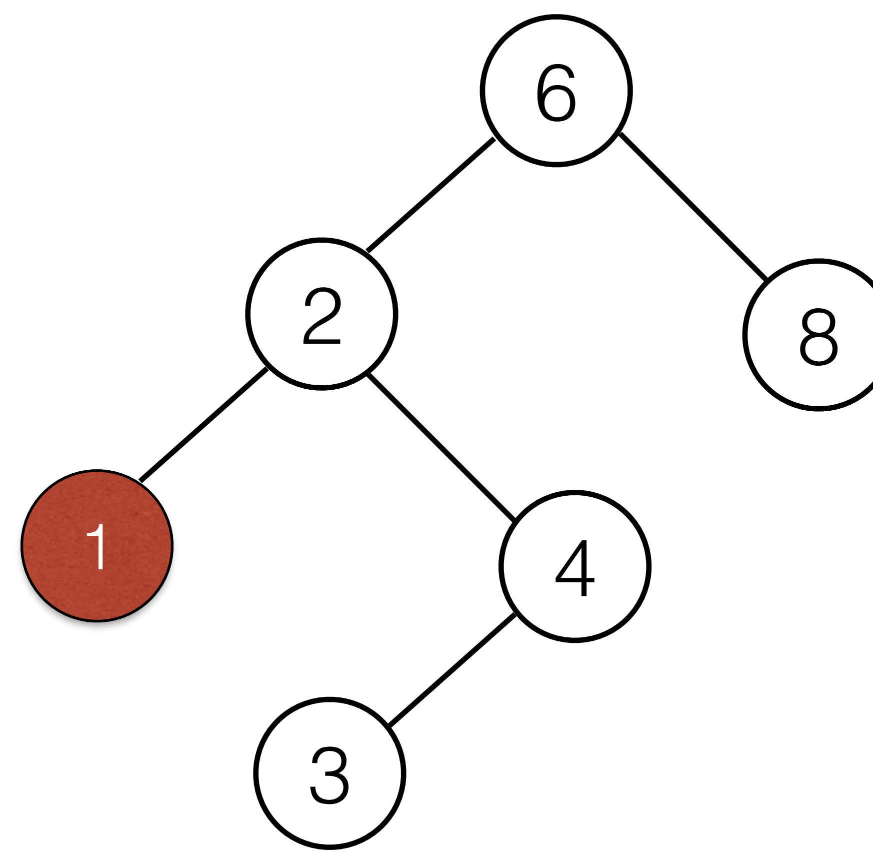


findMax is equivalent.

BST operations: `findMin`

```
private BinaryNode findMin( BinaryNode t ) {
    if( t == null )
        return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}
```

`findMin()`

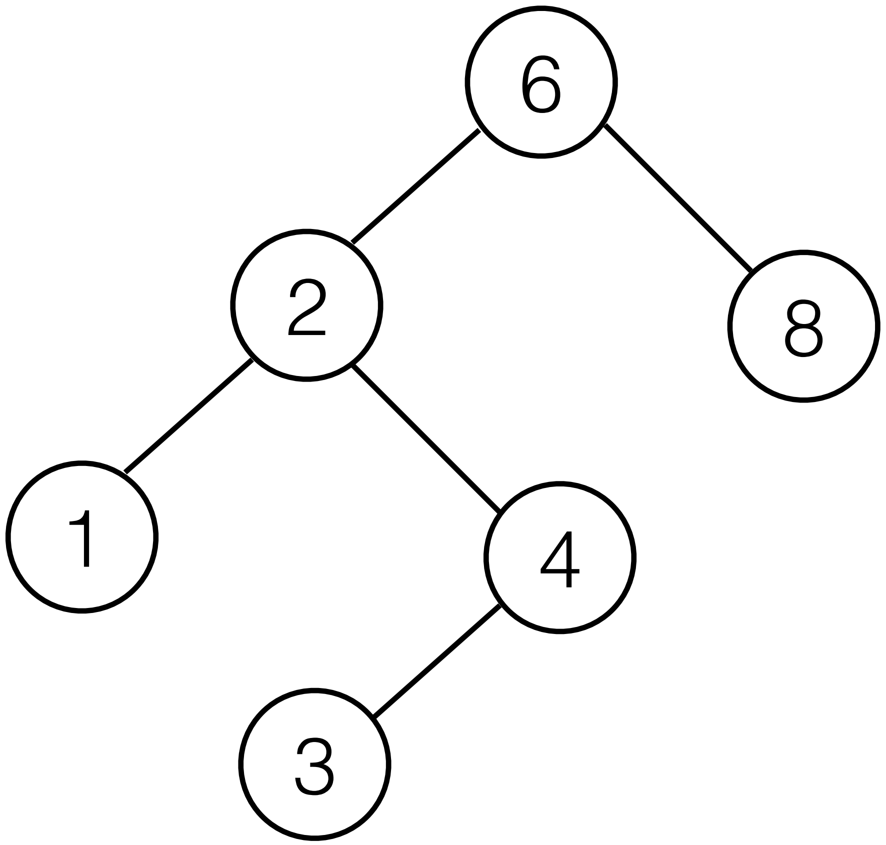


findMax is equivalent.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, *contains* stopped at leaf node n.
  Insert a new node for X as a left or right child of n.

```
private BinaryNode insert( Integer x,
                           BinaryNode t ){
  if( t == null )
      return new BinaryNode( x, null, null );

  if( x < t.data   )
      t.left = insert( x, t.left );
  else if( t.data < x )
      t.right = insert( x, t.right );

  return t;
}
```



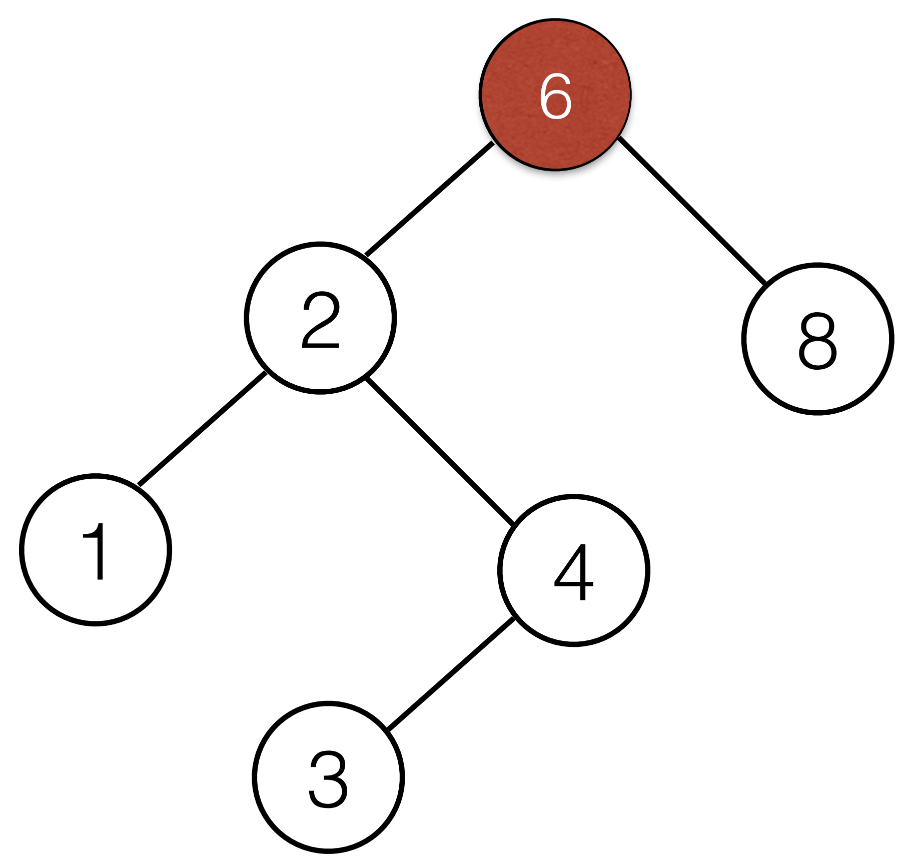
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, *contains* stopped at leaf node n.
  Insert a new node for X as a left or right child of n.

```
private BinaryNode insert( Integer x,
                           BinaryNode t ){
  if( t == null )
      return new BinaryNode( x, null, null );

  if( x < t.data   )
      t.left = insert( x, t.left );
  else if( t.data < x )
      t.right = insert( x, t.right );

  return t;
}
```

`insert(5)`



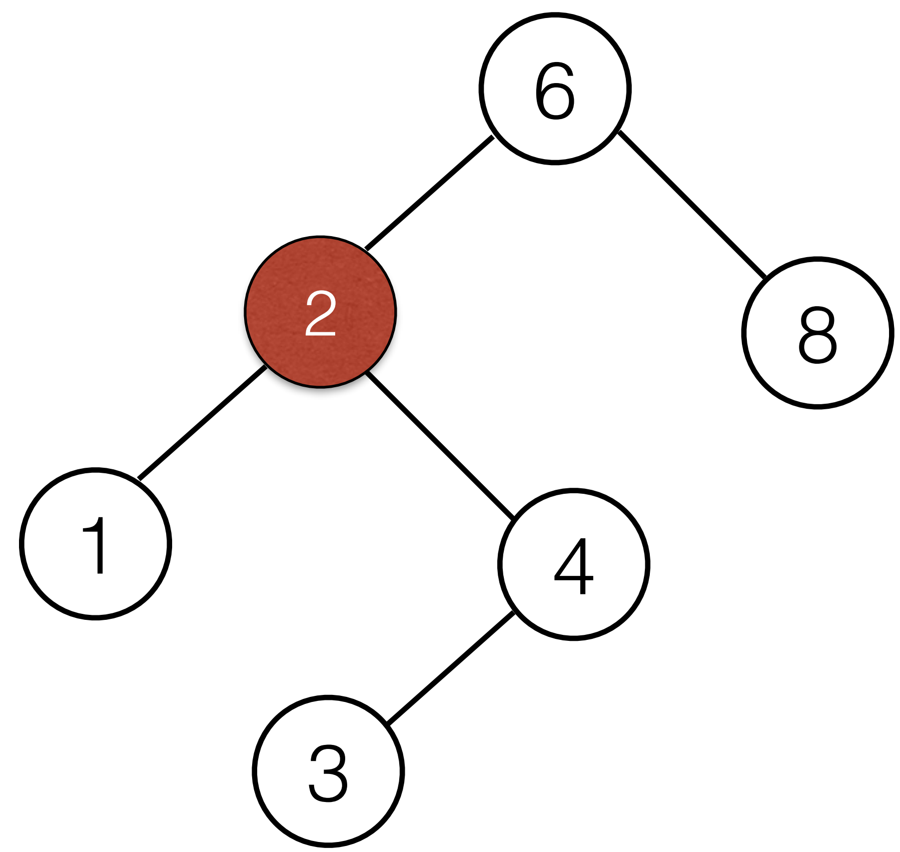
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, *contains* stopped at leaf node n.
  Insert a new node for X as a left or right child of n.

```
private BinaryNode insert( Integer x,
                           BinaryNode t ){
  if( t == null )
      return new BinaryNode( x, null, null );

  if( x < t.data   )
      t.left = insert( x, t.left );
  else if( t.data < x )
      t.right = insert( x, t.right );

  return t;
}
```

`insert(5)`



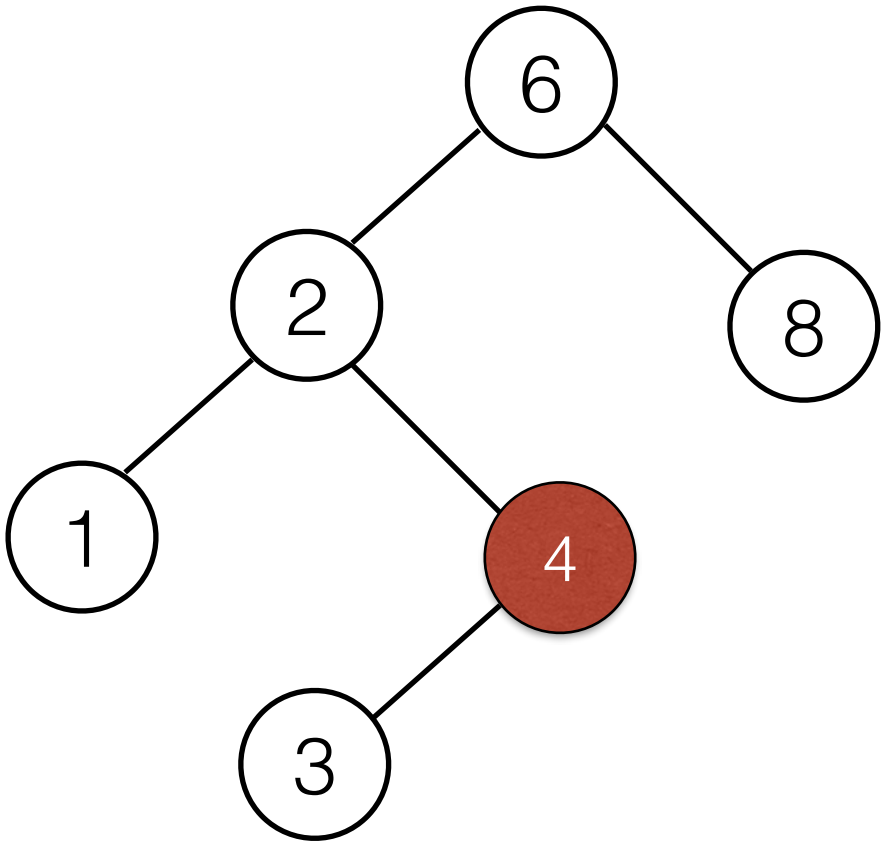
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, *contains* stopped at leaf node n.
  Insert a new node for X as a left or right child of n.

```
private BinaryNode insert( Integer x,
                           BinaryNode t ){
  if( t == null )
      return new BinaryNode( x, null, null );

  if( x < t.data   )
      t.left = insert( x, t.left );
  else if( t.data < x )
      t.right = insert( x, t.right );

  return t;
}
```

`insert(5)`



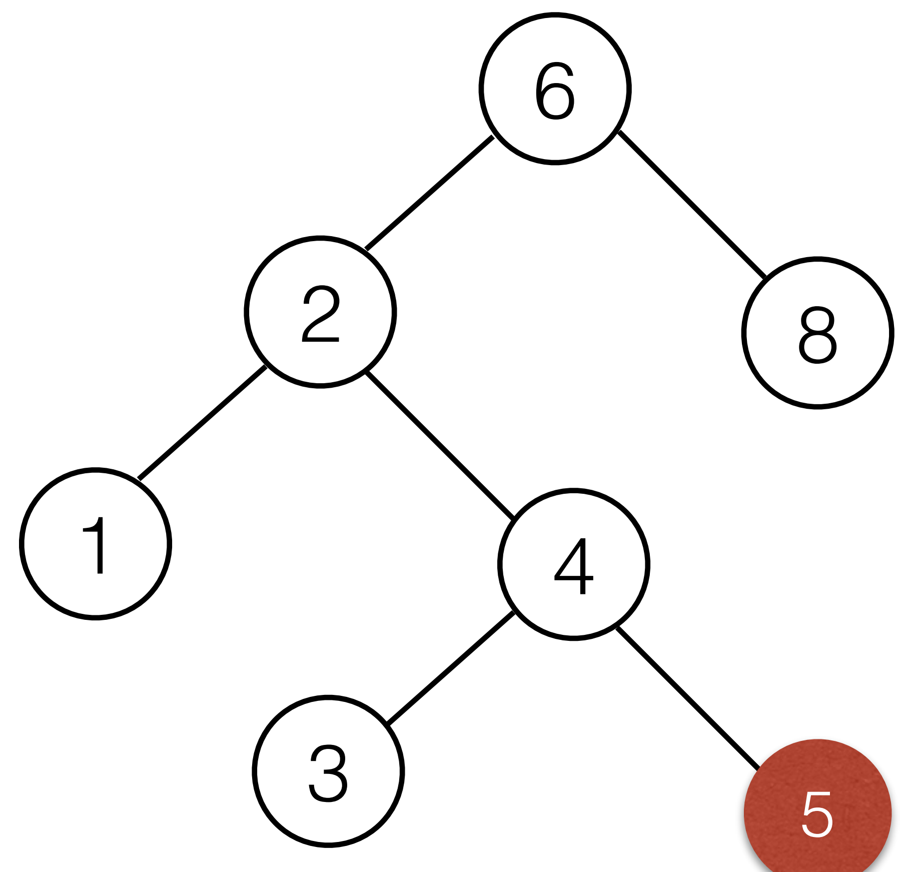
Maintains the BST property.

BST operations: `insert`

- Follow same steps as `contains(X)`
- if X is found, do nothing.
- Otherwise, *contains* stopped at leaf node n.
  Insert a new node for X as a left or right child of n.

```
private BinaryNode insert( Integer x,
                           BinaryNode t ){
  if( t == null )
      return new BinaryNode( x, null, null );

  if( x < t.data   )
      t.left = insert( x, t.left );
  else if( t.data < x )
      t.right = insert( x, t.right );

  return t;
}
```

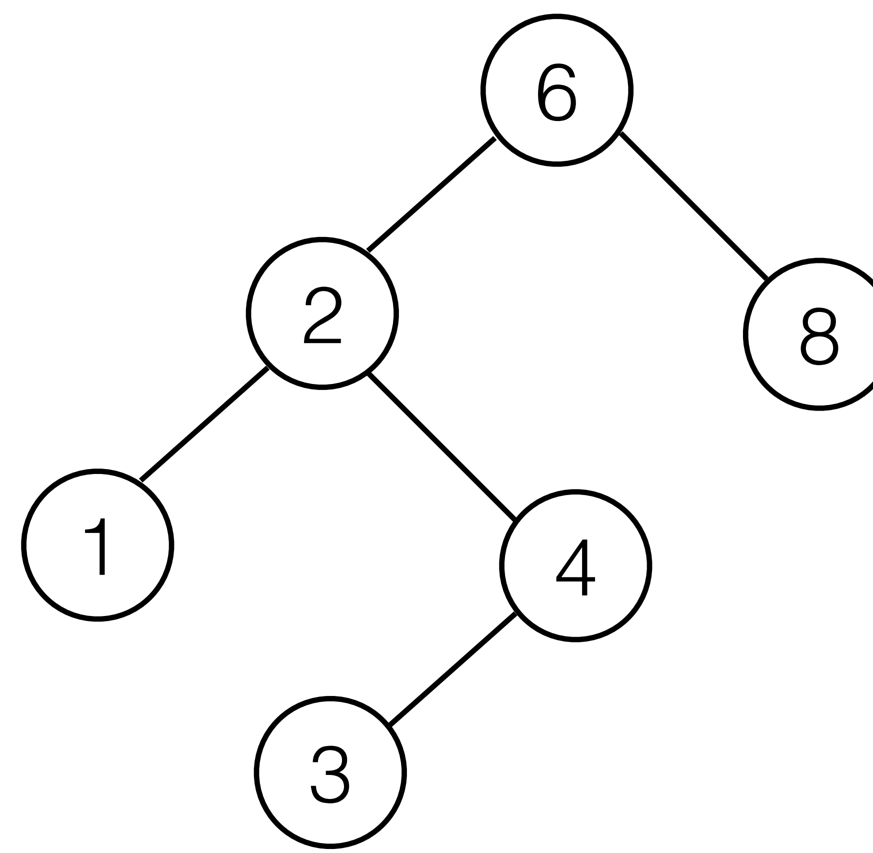
`insert(5)`



Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
  - if s is a leaf, just remove it.
  - if s has a single child t, attach t to the parent of s, in place of s.

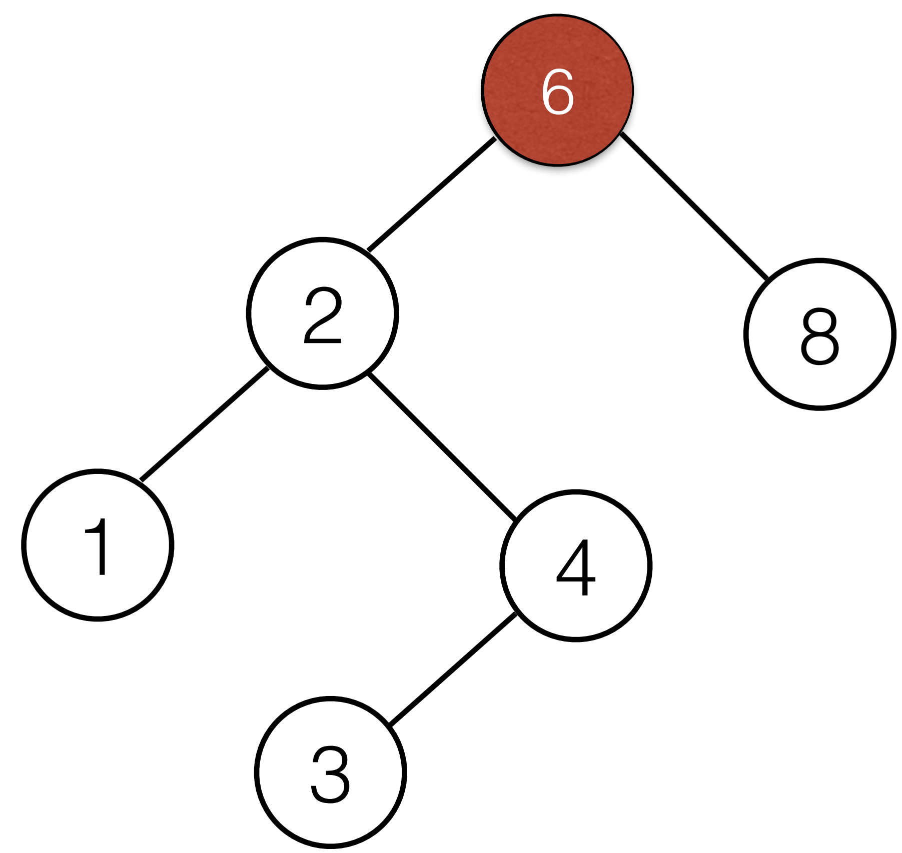


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
  - if s is a leaf, just remove it.
  - if s has a single child t, attach t to the parent of s, in place of s.

`remove(8)`

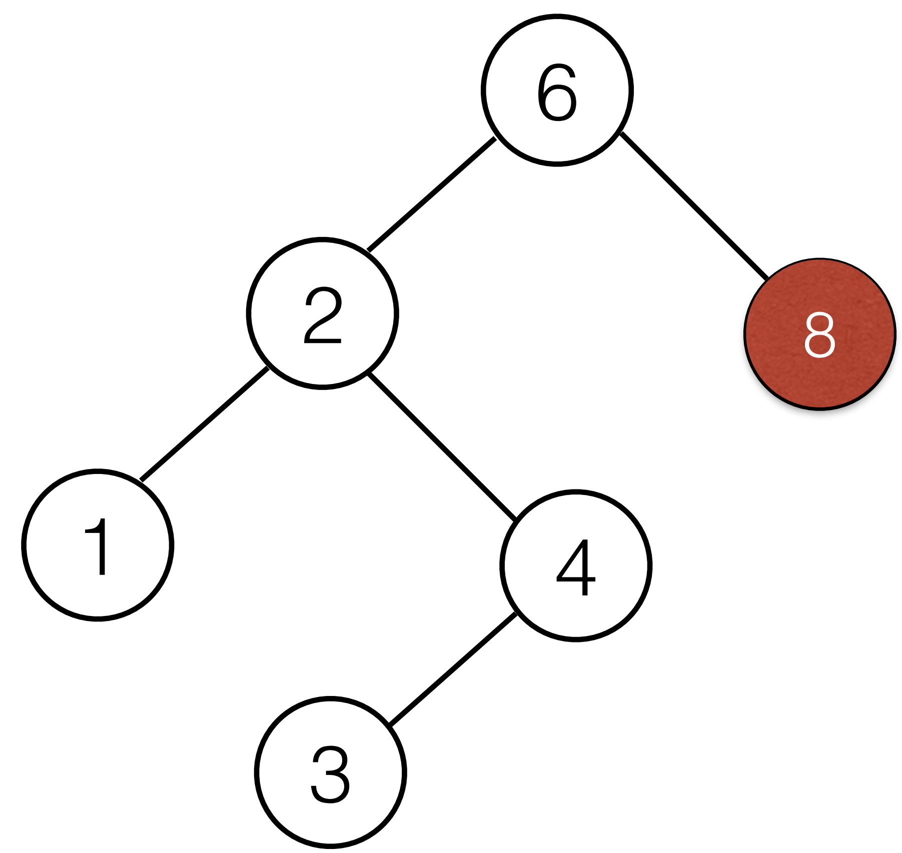


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
  - if s is a leaf, just remove it.
  - if s has a single child t, attach t to the parent of s, in place of s.

`remove(8)`

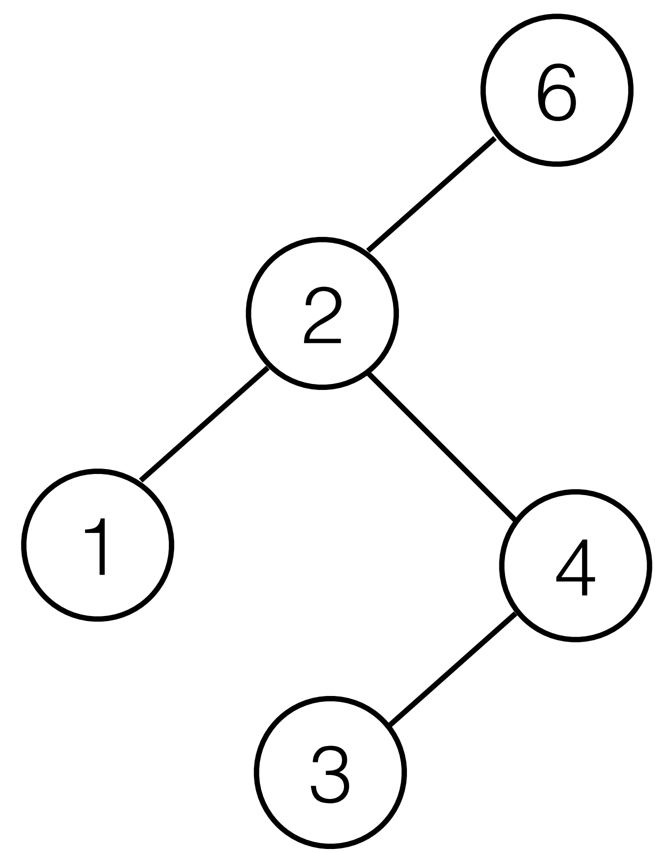


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
  - if s is a leaf, just remove it.
  - if s has a single child t, attach t to the parent of s, in place of s.

`remove(8)`

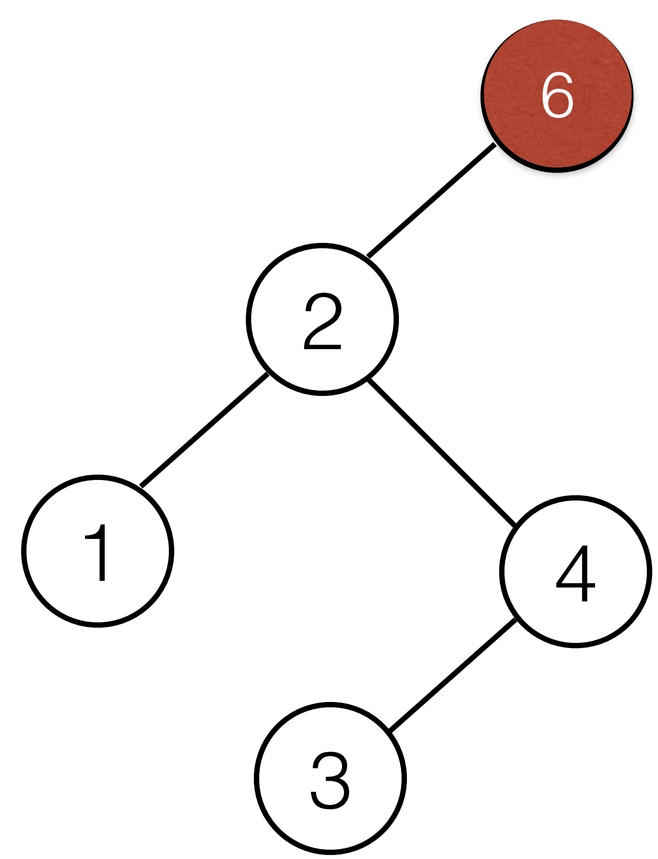


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
    - if s is a leaf, just remove it.
    - if s has a single child t, attach t to the parent of s, in place of s.

`remove(4)`

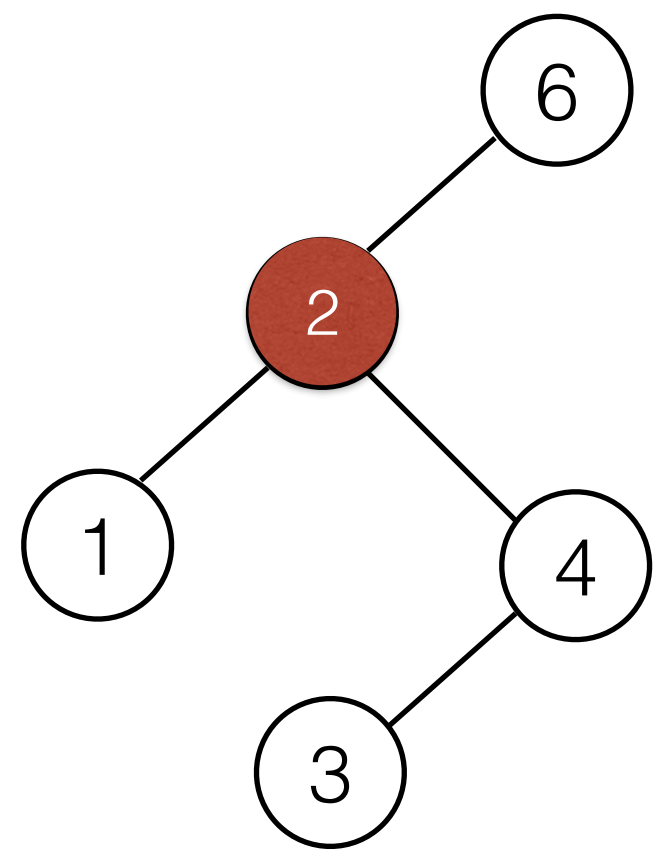


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
    - if s is a leaf, just remove it.
    - if s has a single child t, attach t to the parent of s, in place of s.

`remove(4)`

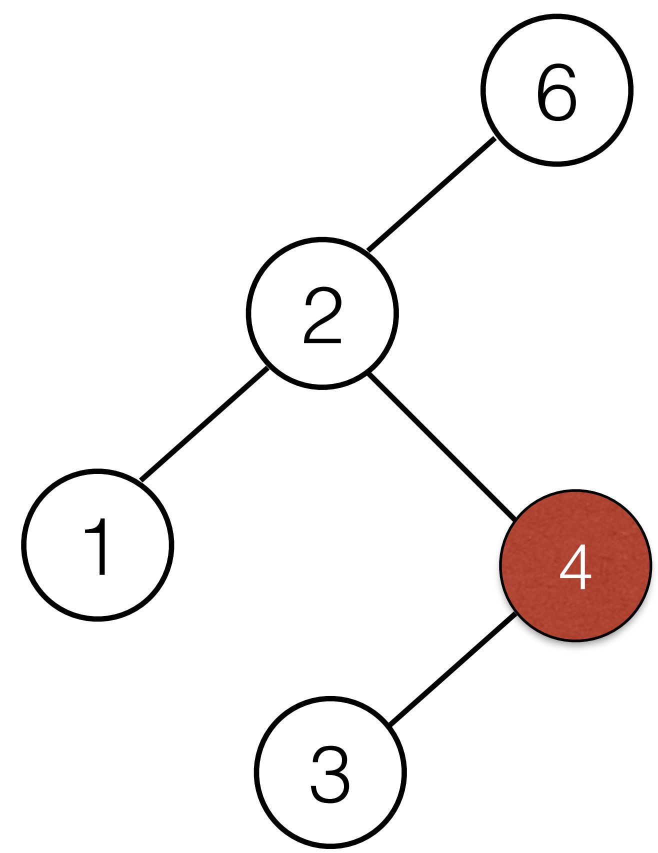


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
    - if s is a leaf, just remove it.
    - if s has a single child t, attach t to the parent of s, in place of s.

`remove(4)`

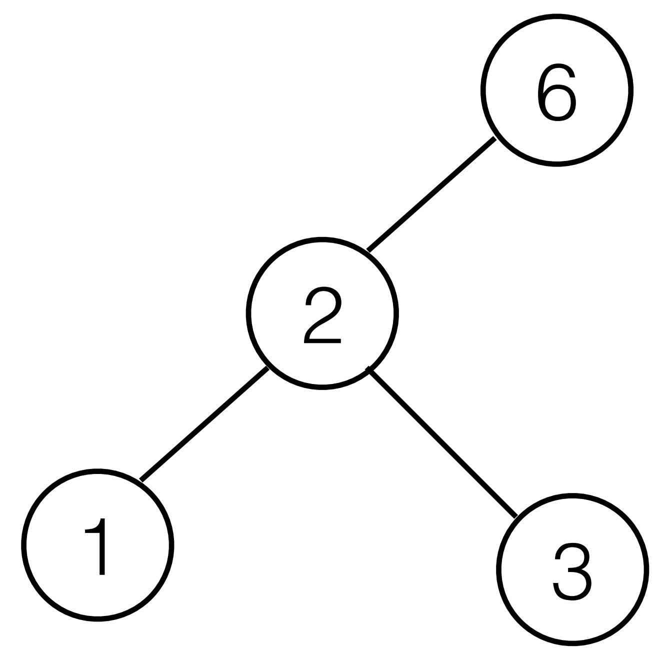


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
    - if s is a leaf, just remove it.
    - if s has a single child t, attach t to the parent of s, in place of s.

`remove(4)`

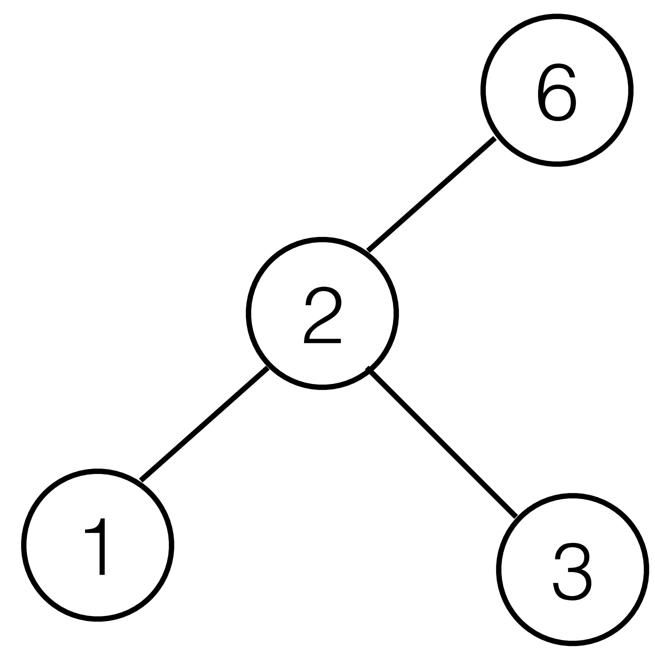


Maintains the BST property.

BST operations: `remove`

- First find x following the same steps as `contains(X)`.
- If x is found in a node s:
    - if s is a leaf, just remove it.
    - if s has a single child t, attach t to the parent of s, in place of s.
    - what if s has two children?

`remove(4)`

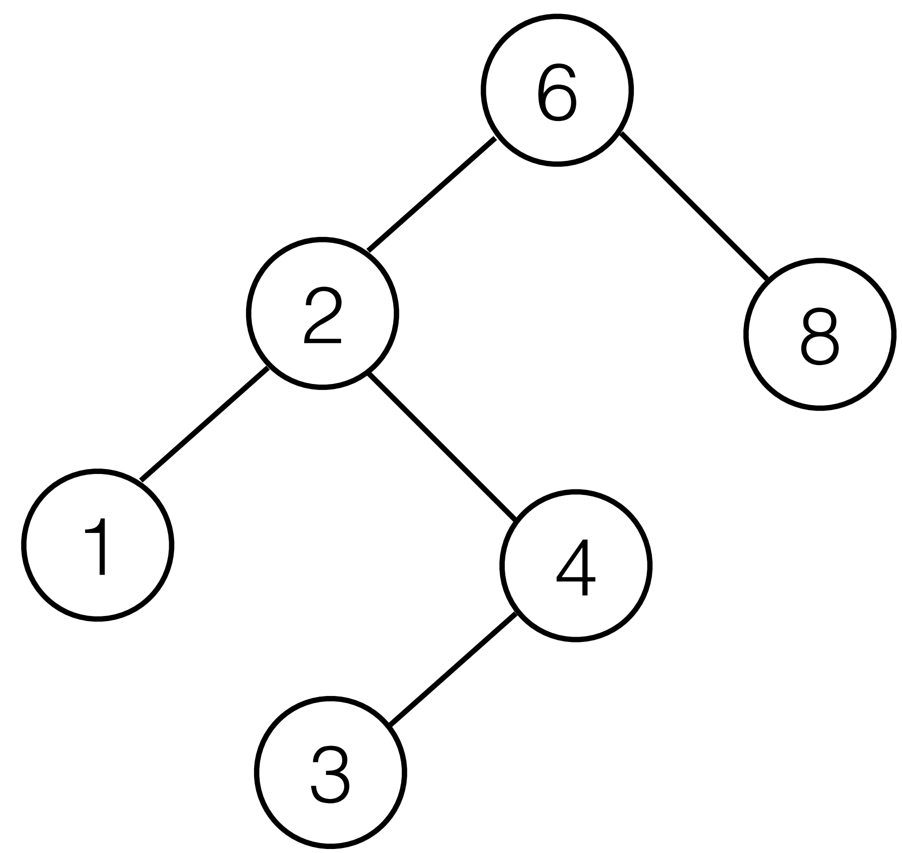


Maintains the BST property.

BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
  - Find the smallest node *u* in the subtree rooted in t_{right}.
  - replace value of s with value of u.
  - recursively remove u.

To maintain the BST property, the node that replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

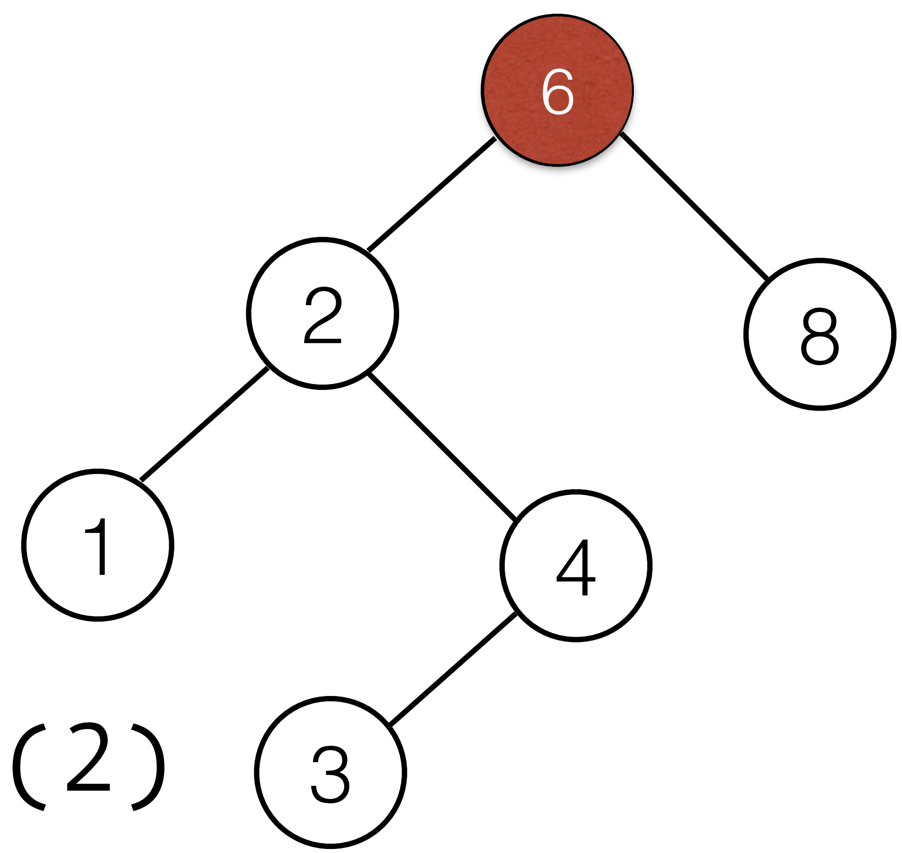


BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
  - Find the smallest node *u* in the subtree rooted in t_{right}.
  - replace value of s with value of u.
  - recursively remove u.

To maintain the BST property, the node that replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

`remove(2)`

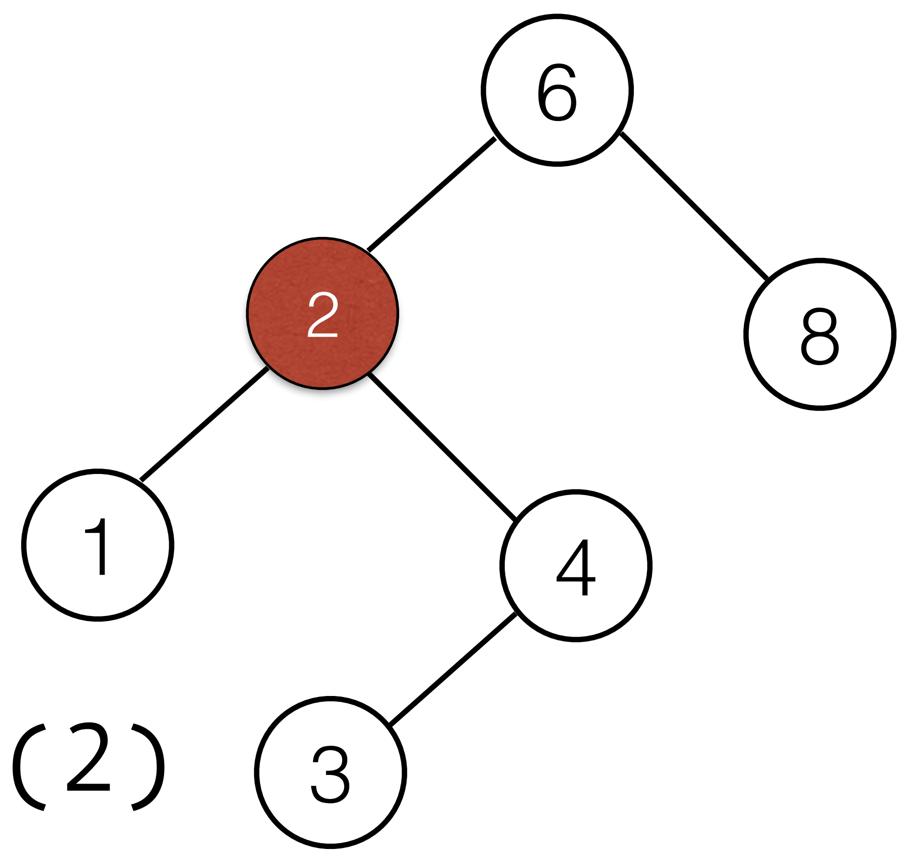


BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
  - Find the smallest node *u* in the subtree rooted in t_{right}.
  - replace value of s with value of u.
  - recursively remove u.

To maintain the BST property, the node that replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

`remove(2)`

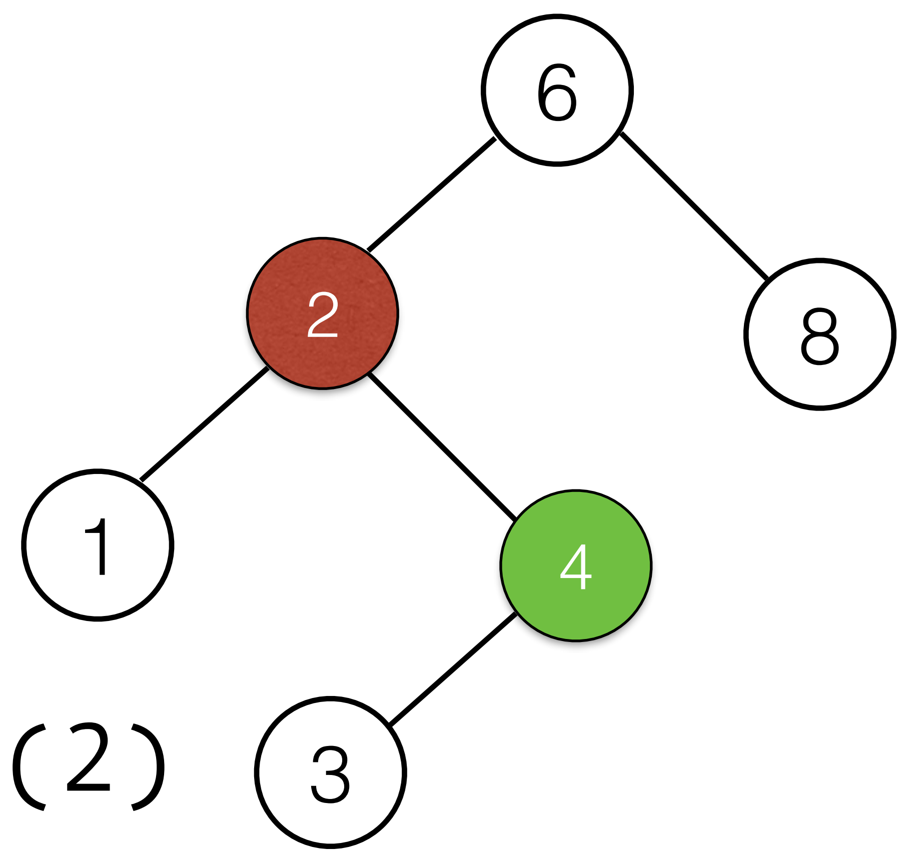


BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
  - Find the smallest node *u* in the subtree rooted in t_{right}.
  - replace value of s with value of u.
  - recursively remove u.

To maintain the BST property, the node that replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

`remove(2)`

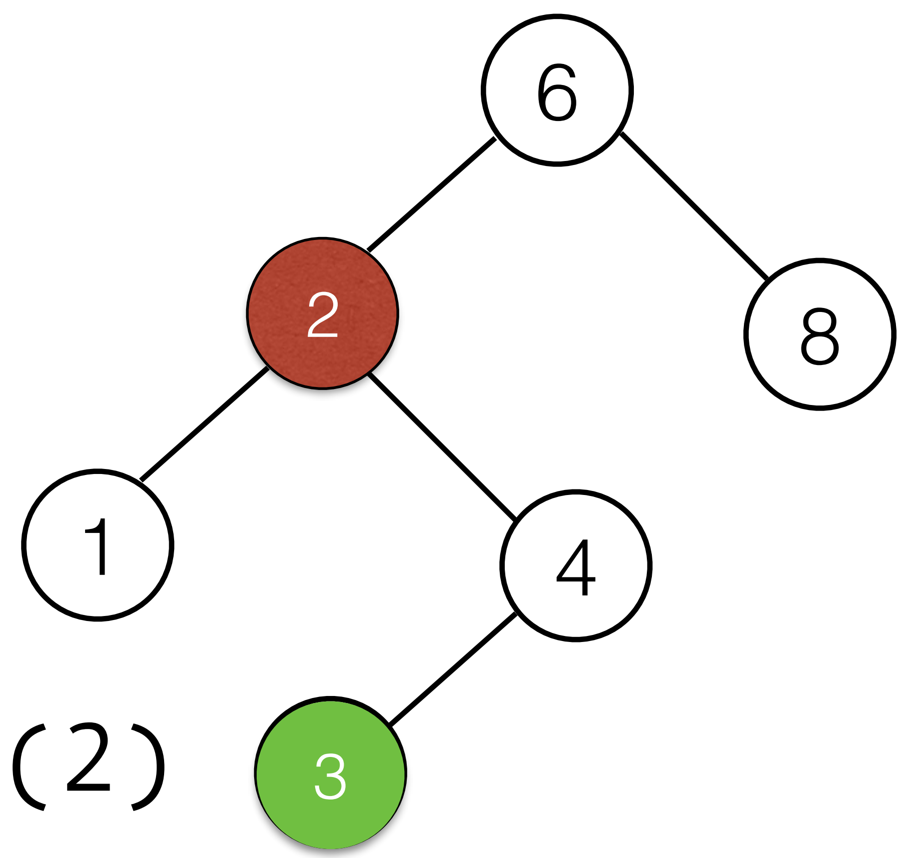


BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
  - Find the smallest node *u* in the subtree rooted in t_{right}.
  - replace value of s with value of u.
  - recursively remove u.

To maintain the BST property, the node that replace s needs to be
- larger than any node in the left subtree
- but smaller than any node in the right subtree.

`remove(2)`

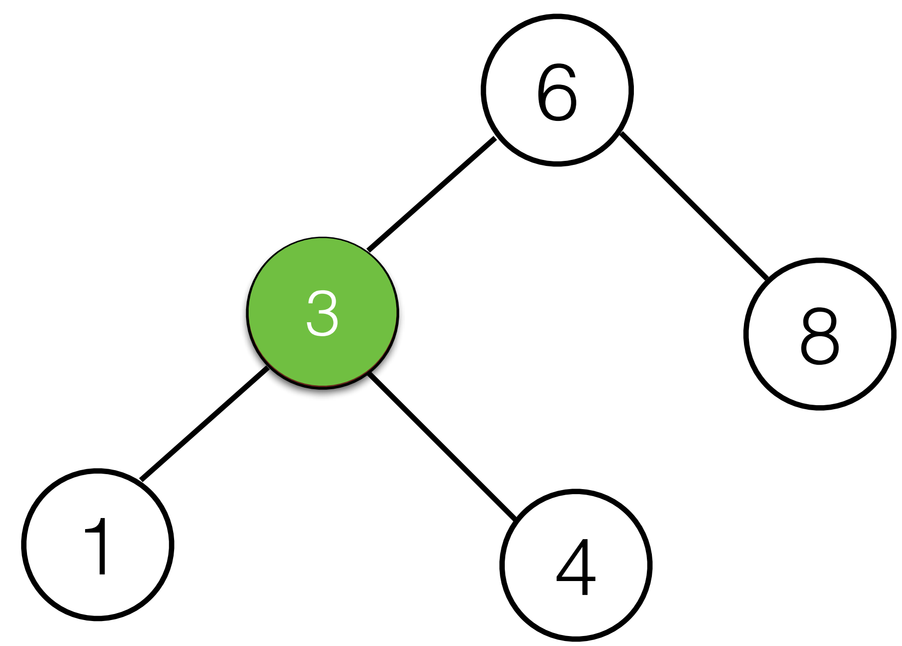


BST operations: `remove`

- If x is found in a node s that has two children t_{left} and t_{right}:
  - Find the smallest node *u* in the subtree rooted in t_{right}.
  - replace value of s with value of u.
  - recursively remove u.

To maintain the BST property, the node that replace s needs to be

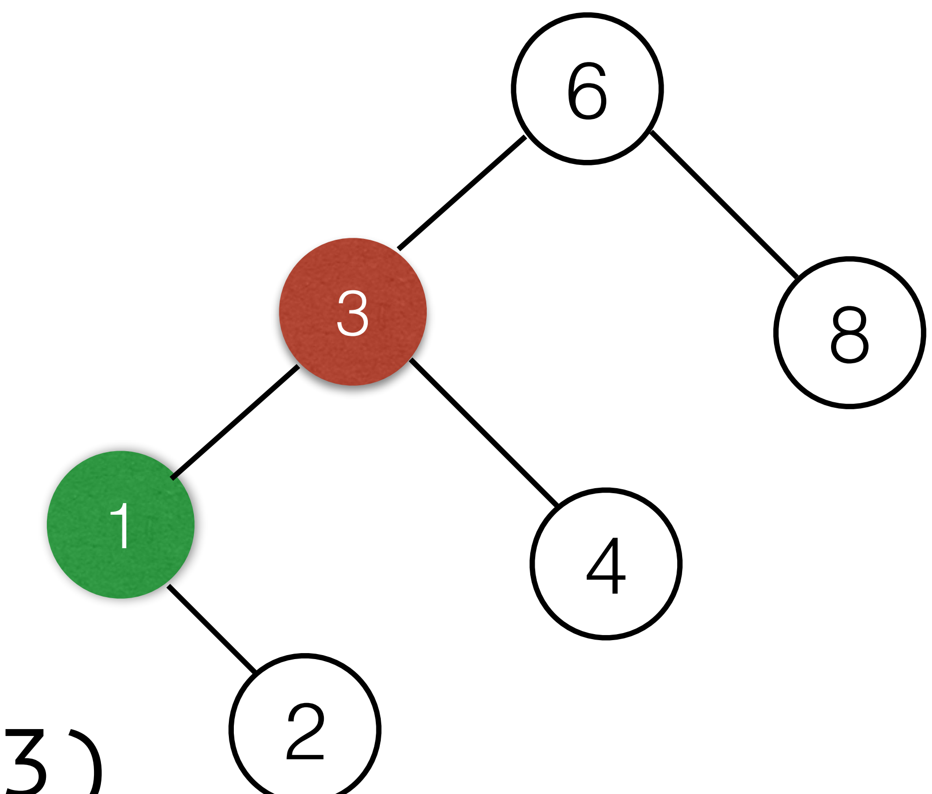
- larger than any node in the left subtree
- but smaller than any node in the right subtree.



`remove(2)`

BST operations: `remove`

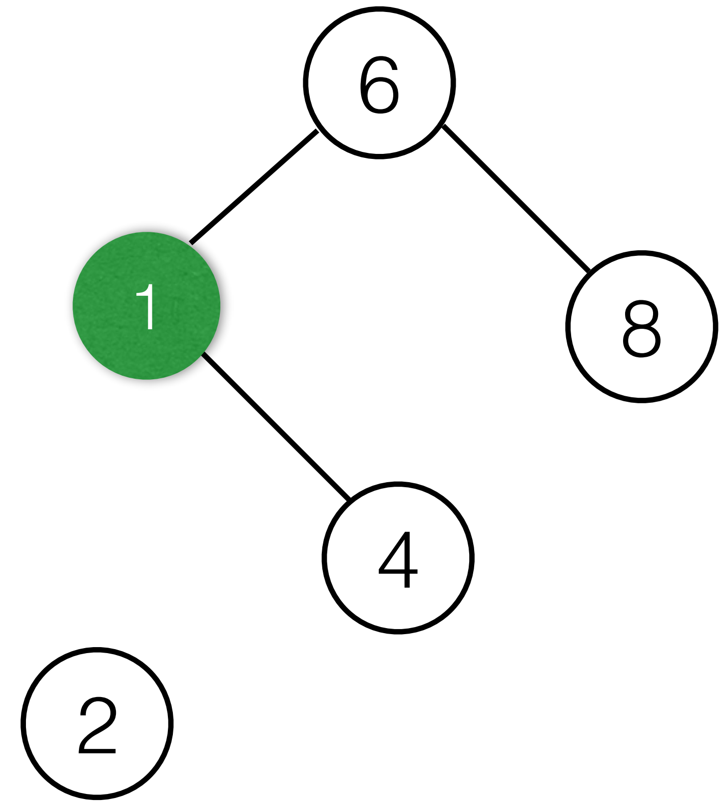
- Why not just replace s with the root of t_{left}?



`remove(3)`

BST operations: `remove`

- Why not just replace s with the root of t_{left}?



`remove(3)`

Implementing `remove`

```
private BinaryNode remove( Integer x, BinaryNode t ){
  if( t == null )
    return t; // Item not found; do nothing

  if (x < t.data )
    t.left = remove( x, t.left );
  else if(t.data < x )
    t.right = remove( x, t.right );

  else //found x
    if( t.left != null && t.right != null ) { // 2 children
      t.element = findMin( t.right ).element;
      t.right = remove( t.element, t.right );
    } else
      if (t.left != null) // 1 or 0 children.
        return t.left;
      else
        return t.right;
}
```

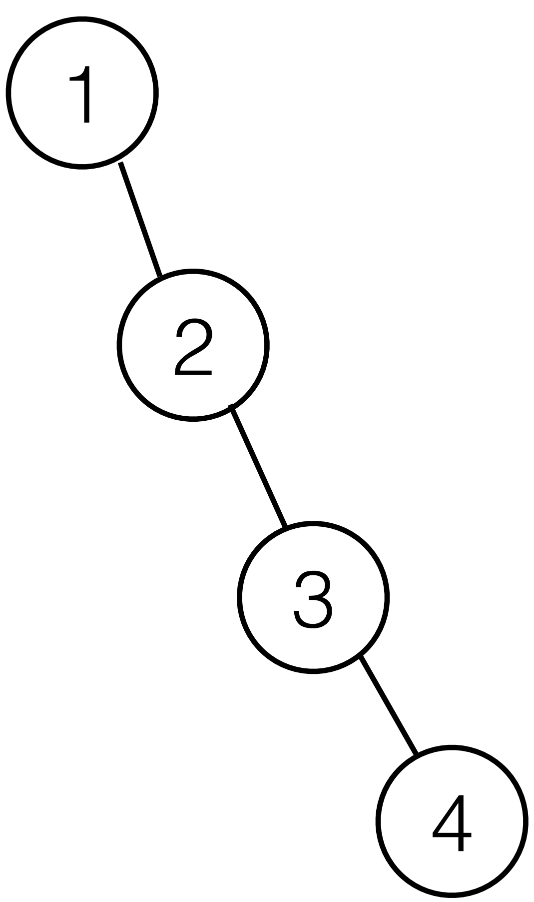
BST Running Time Analysis

- How long do the BST operations take?
- Given a BST *T*, we need a single pass down the tree to access some node *s* in *depth(s)* steps.
- What is the best/expected/worst-case depth of a node in any BST?

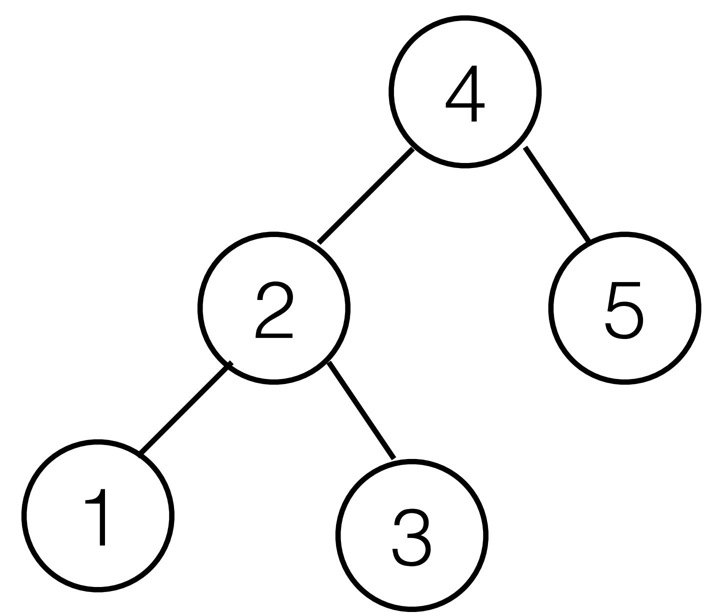
Worst and Best Case Height of a Binary Search Tree

- Assume we have a BST with N nodes.

- Worst case: T does not branch *height(T)=N*



- Best case: *height(T)= O(log N)*

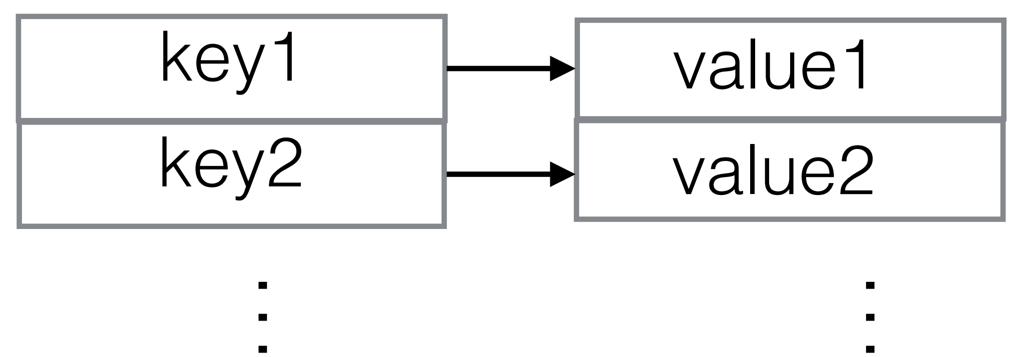


complete binary tree

Comparing Complex Items

- So far, our BSTs contained `Integers`.

- One Goal of BSTs: Implement efficient lookup for Map keys and sorted Sets.



- We can implement generic BSTs that can contain any kind of element, including (key,value) pairs.

- But we must be able to *sort* the elements, i.e. compare them using <, >, and =. The (key, value) pair class should implement Comparable.

Example (key/value) Pair Implementation

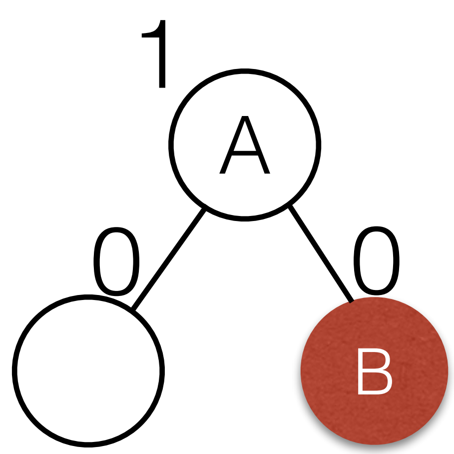
```
private class Pair<K extends Comparable<K>,V>
                    implements Comparable<Pair<K,?>>{
   public K key;
   public V value;

   public Pair(K theKey, V theValue) {
      key = theKey; value = theValue;
   }

   @Override
   public int compareTo(Pair<K,?> other) {
      return key.compareTo(other.key);
   }
}
```

Balanced BSTs

- Balance condition: Guarantee that the BST is always close to a *complete* binary tree.
  - Then the height of the tree will be O(*log N)* and all BST operations will run in *O(log N)*.
- There are different implementations of self-balancing BSTs:
  - AVL trees, Red-Black trees, B+-trees, ...

Height of an “Empty Subtree”

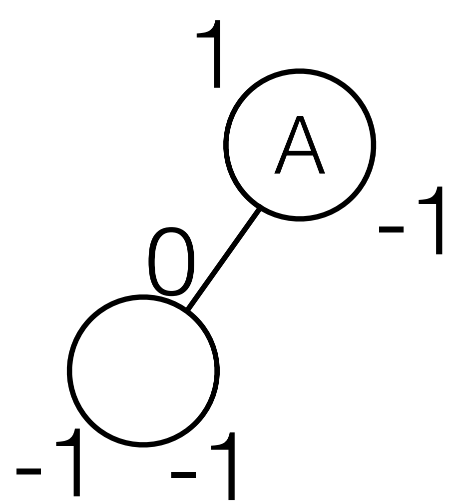
Recall that the hight of an empty subtree is -1.



height of the right subtree of A: 0

Height of an “Empty Subtree”

Recall that the hight of an empty subtree is -1.

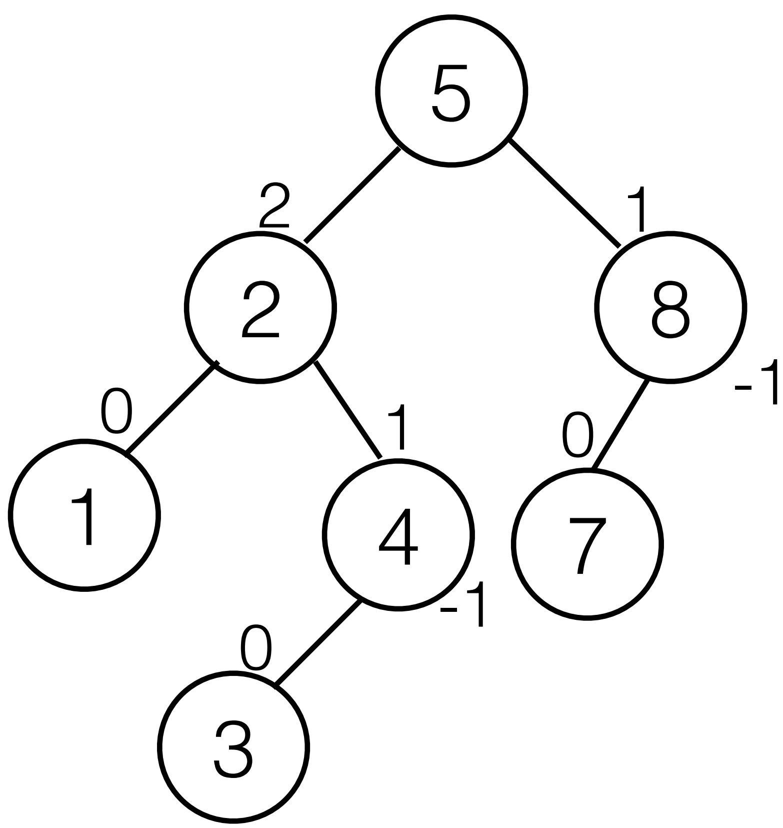


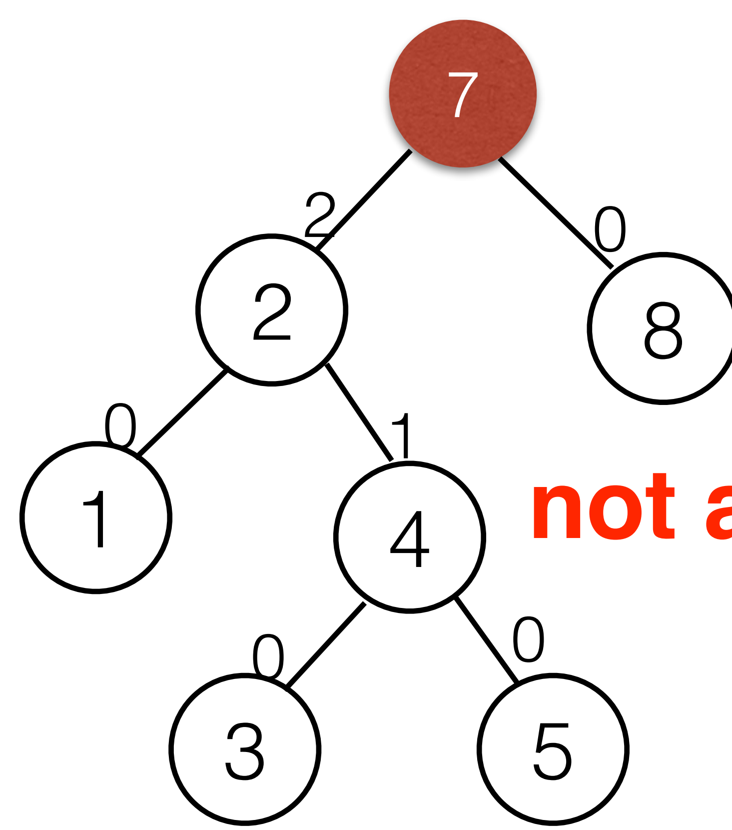
height of the right subtree of A: -1

AVL Tree Condition

- An AVL Tree is a Binary Search Tree in which the following **balance condition** holds after each operation:

  - For every node, the height of the left and right subtree differs by at most 1.





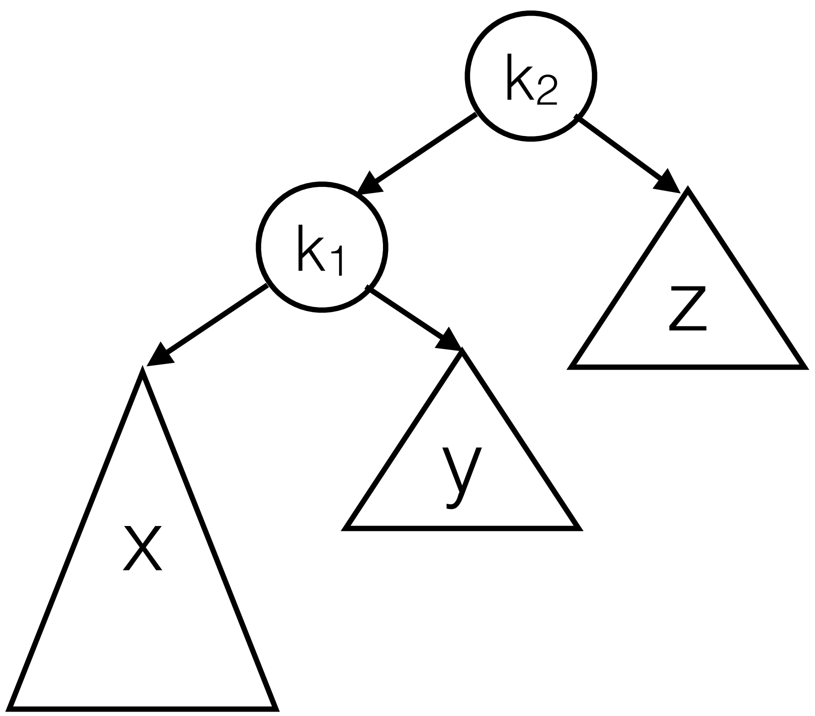
not an AVL tree

AVL Trees

- Height of an AVL tree is at most
  $\sim 1.44 \log(N+2) - 1.328 = O(\log N)$
- How to maintain the balance condition?
  - Rebalance the tree after each modification (insertion or deletion).
  - Rebalancing must be cheap.

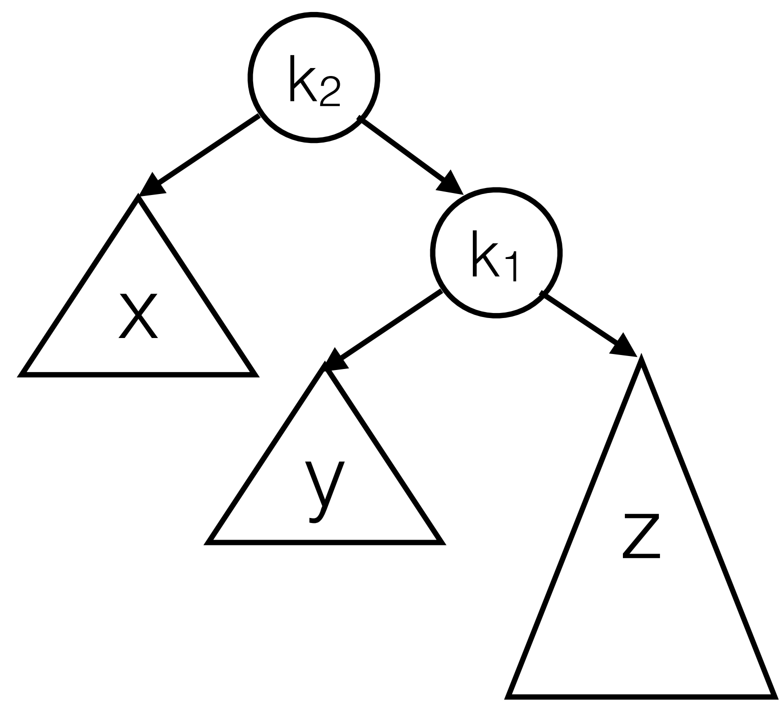
"Outside" Imbalance

node k_2 violates the balance condition



left subtree of **left** child too high

LL Outside



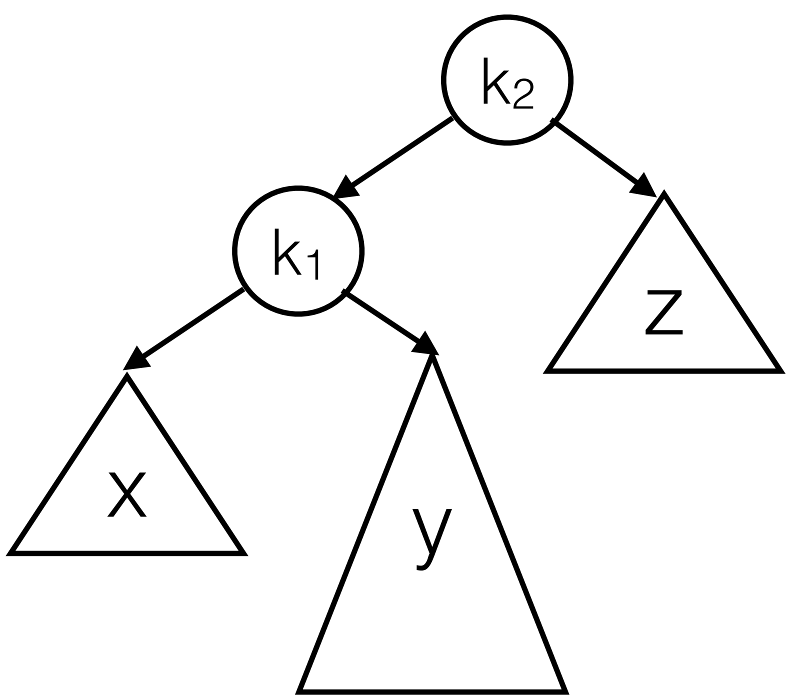
right subtree of **right** child too high

RR Outside

- Solution: Single rotation

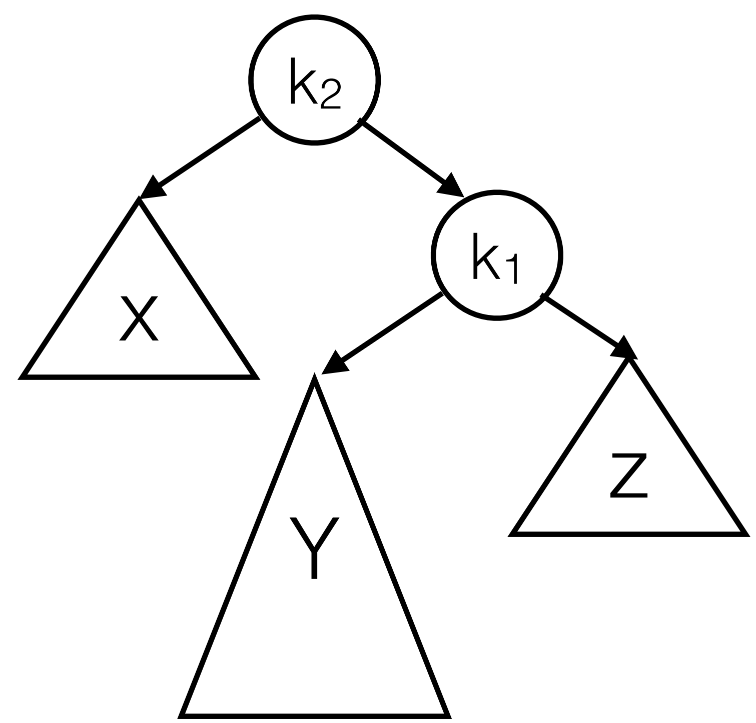
“Inside” Imbalance

node k_2 violates the balance condition



right subtree of **left** child too high

LR Inside

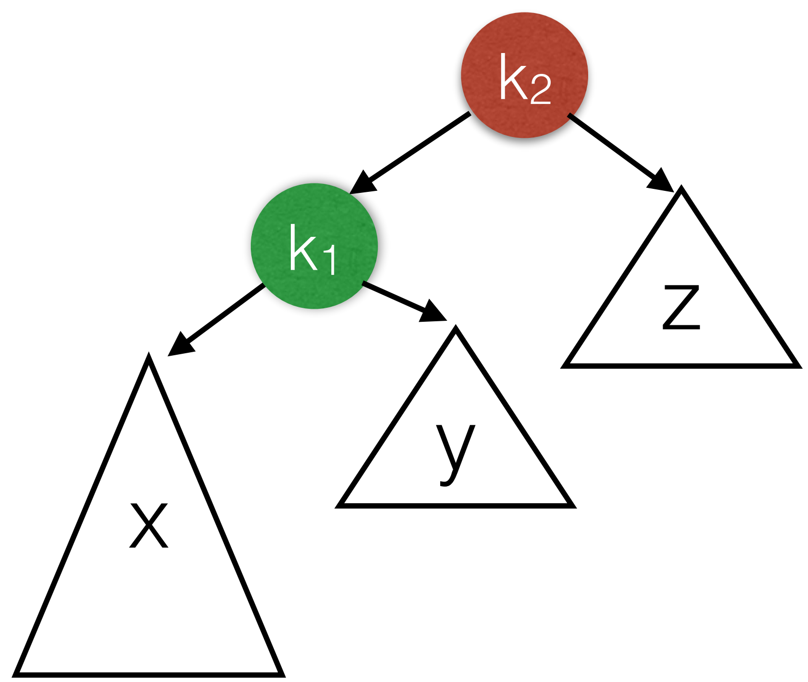


left subtree of **right** child too high

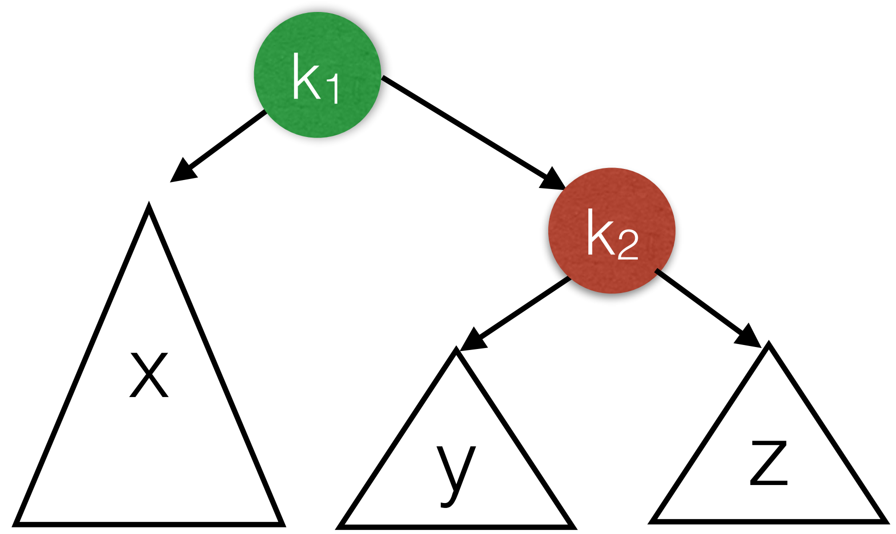
RL Inside

- Solution: Double rotation

Single Rotation (1)



Single Rotation (1)



Single Rotation (2)

Single Rotation maintains BST property:

- x is still left subtree of k_1.
- z is still right subtree of k_2.
- For all keys v in y: $k_1 < v < k_2$
  (y becomes new left subtree of k_2)

Single Rotation (2)

Changed references:

- k1.right = k2
- k2.left = root(y)

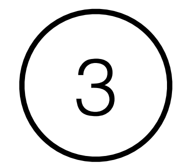
- parent(k2).left = k1 OR parent(k2).right = k1

Maintaining Balance in an AVL Tree

- After each insertion/deletion, find **the lowest node *k*** that violates the balance condition (if any), starting at the insertion site.
- Perform rotation to re-balance the tree.
- Rotation maintains original height of subtree under *k* before the insertion. No further rotations are needed.
- Invariant:
  After each modification, the tree maintains both the BST property and the AVL condition.

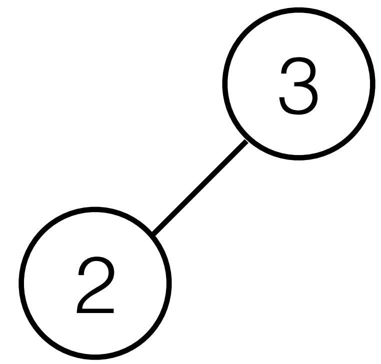
Single Rotation Example

insert(3)



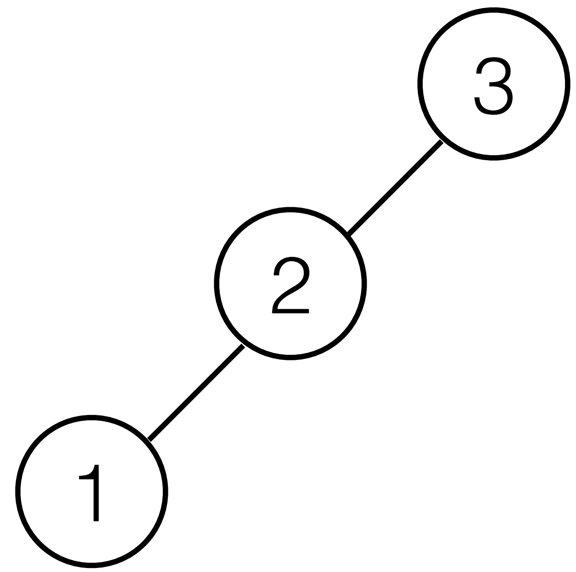
Single Rotation Example

insert(3)
insert(2)



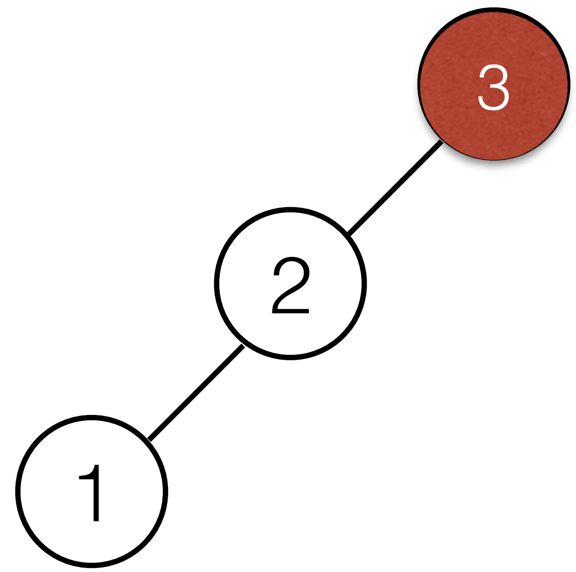
Single Rotation Example

insert(3)
insert(2)
insert(1)



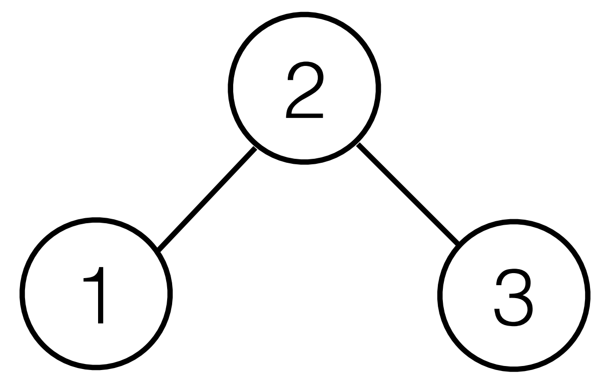
Single Rotation Example

insert(3)
insert(2)
insert(1) rotate_right(3)



Single Rotation Example

insert(3)
insert(2)
insert(1) rotate_right(3)



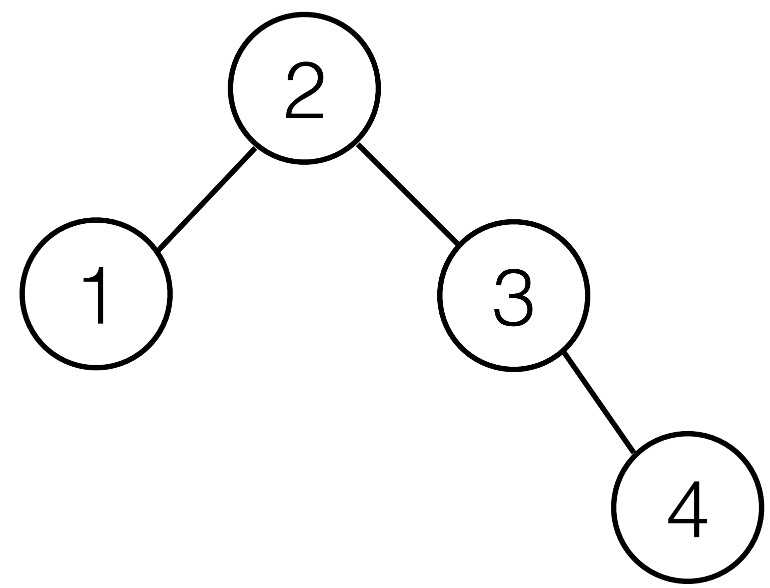
Single Rotation Example

insert(3)

insert(2)

insert(1) rotate_right(3)

insert(4)



Single Rotation Example

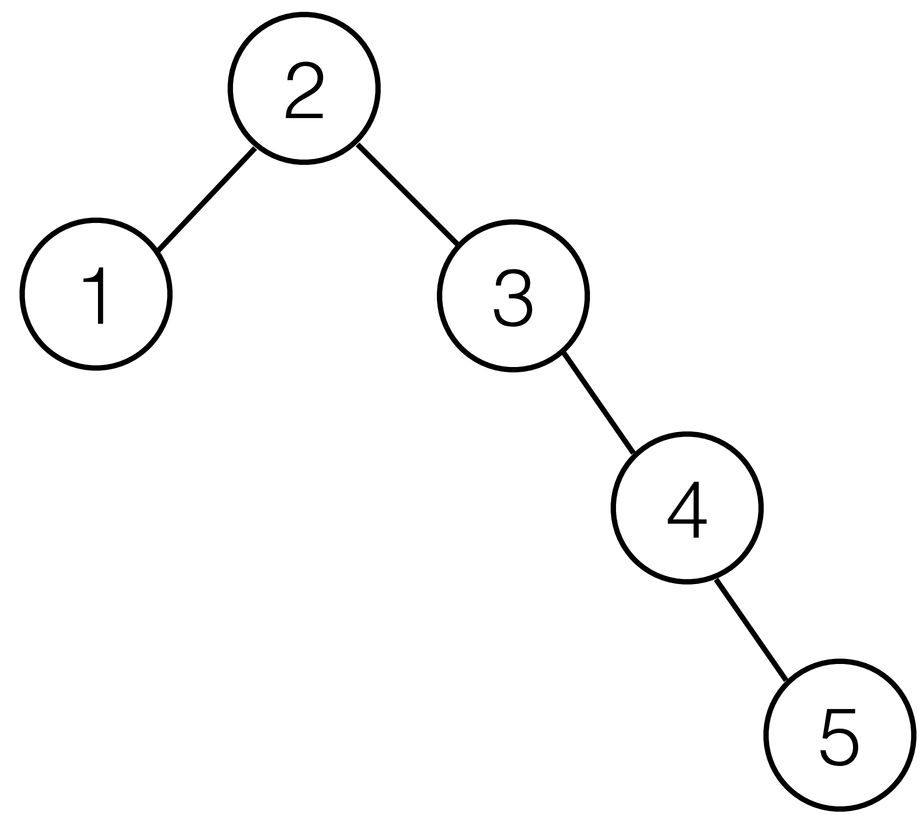
insert(3)

insert(2)

insert(1) rotate_right(3)

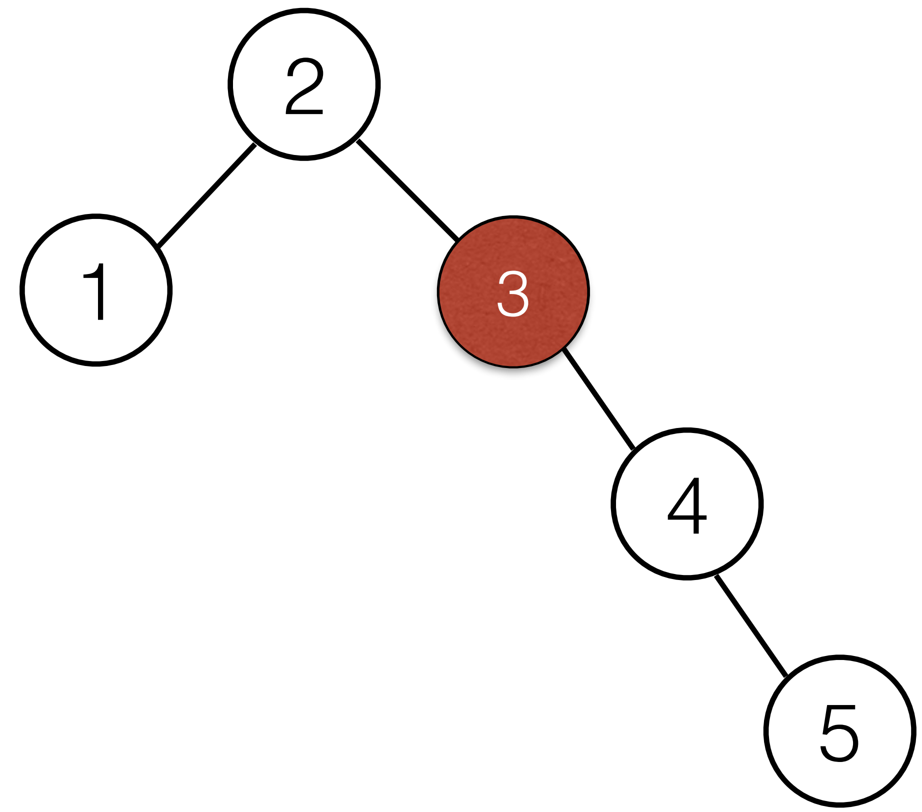
insert(4)

insert(5)



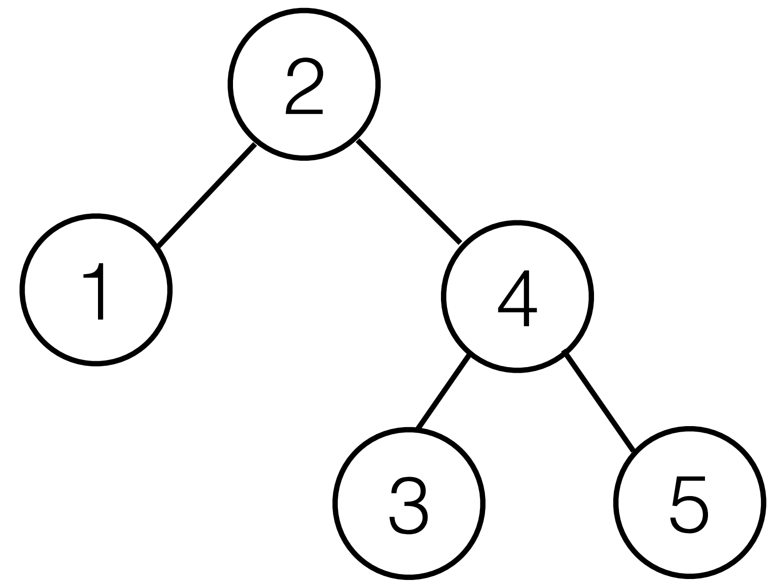
Single Rotation Example

insert(3)
insert(2)
insert(1) rotate_right(3)
insert(4)
insert(5) rotate_left(3)



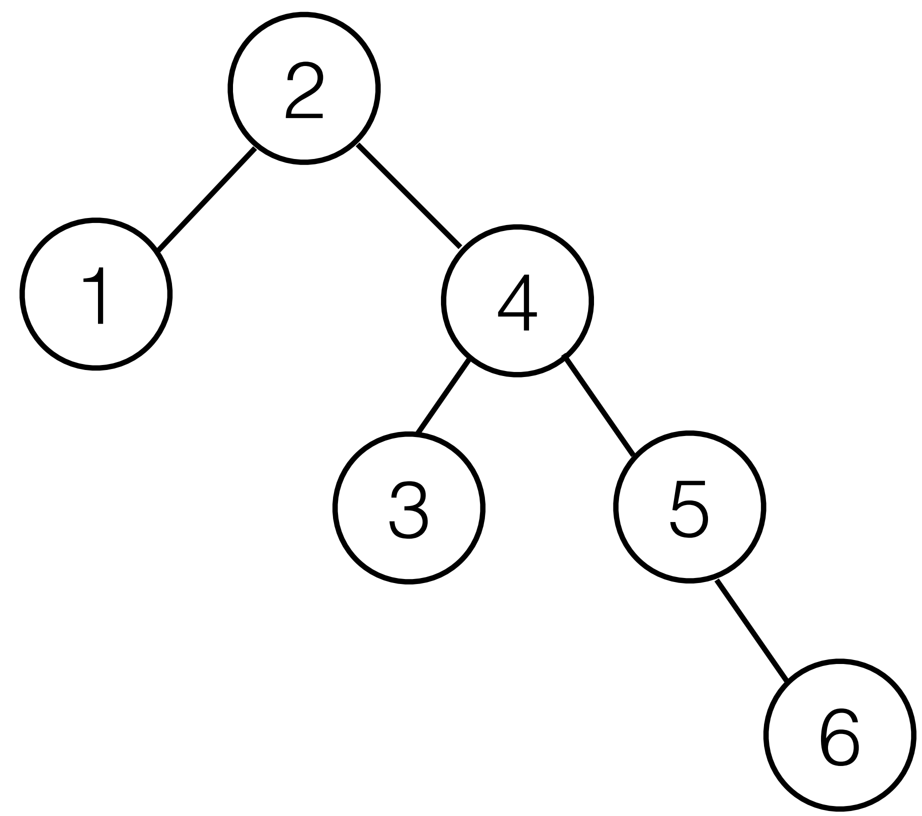
Single Rotation Example

insert(3)
insert(2)
insert(1) rotate_right(3)
insert(4)
insert(5) rotate_left(3)



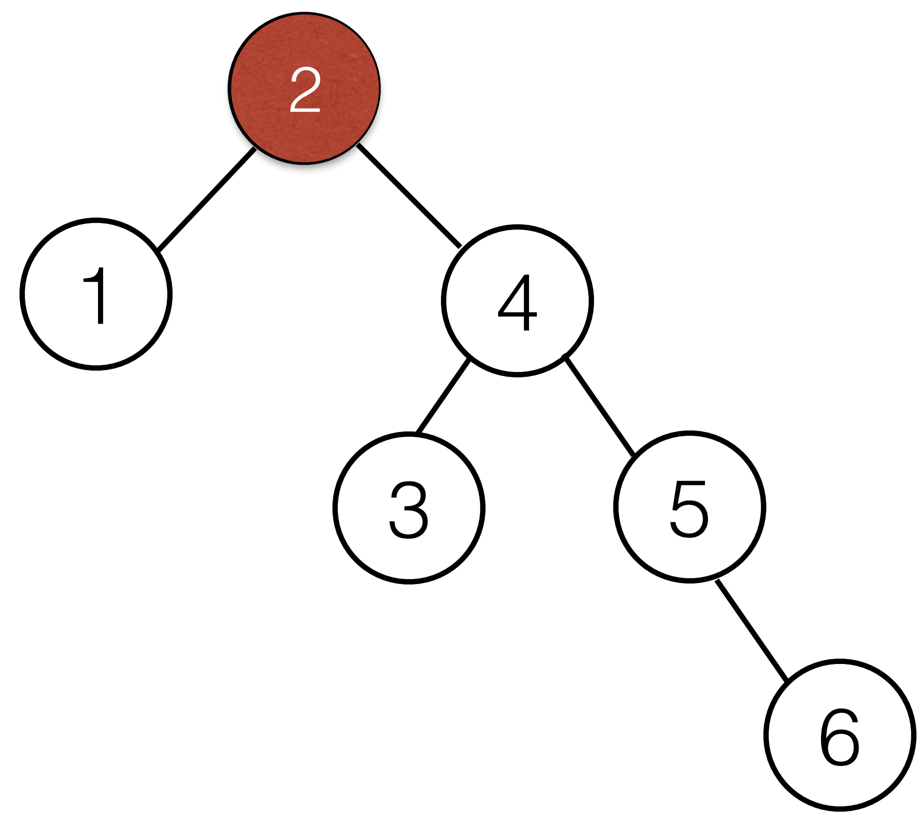
Single Rotation Example

insert(3)
insert(2)
insert(1) rotate_right(3)
insert(4)
insert(5) rotate_left(3)
insert(6)



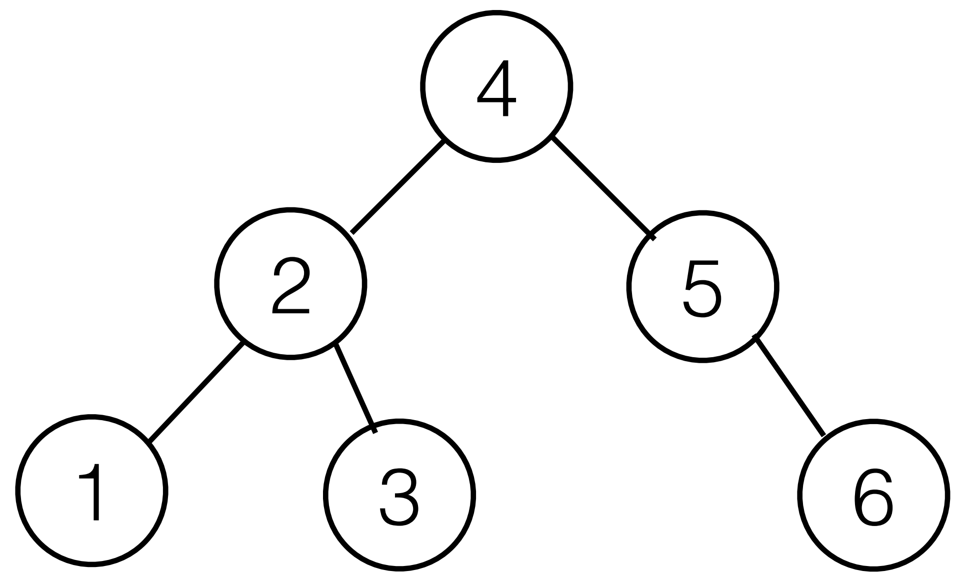
Single Rotation Example

insert(3)
insert(2)
insert(1) rotate_right(3)
insert(4)
insert(5) rotate_left(3)
insert(6) rotate_left(2)



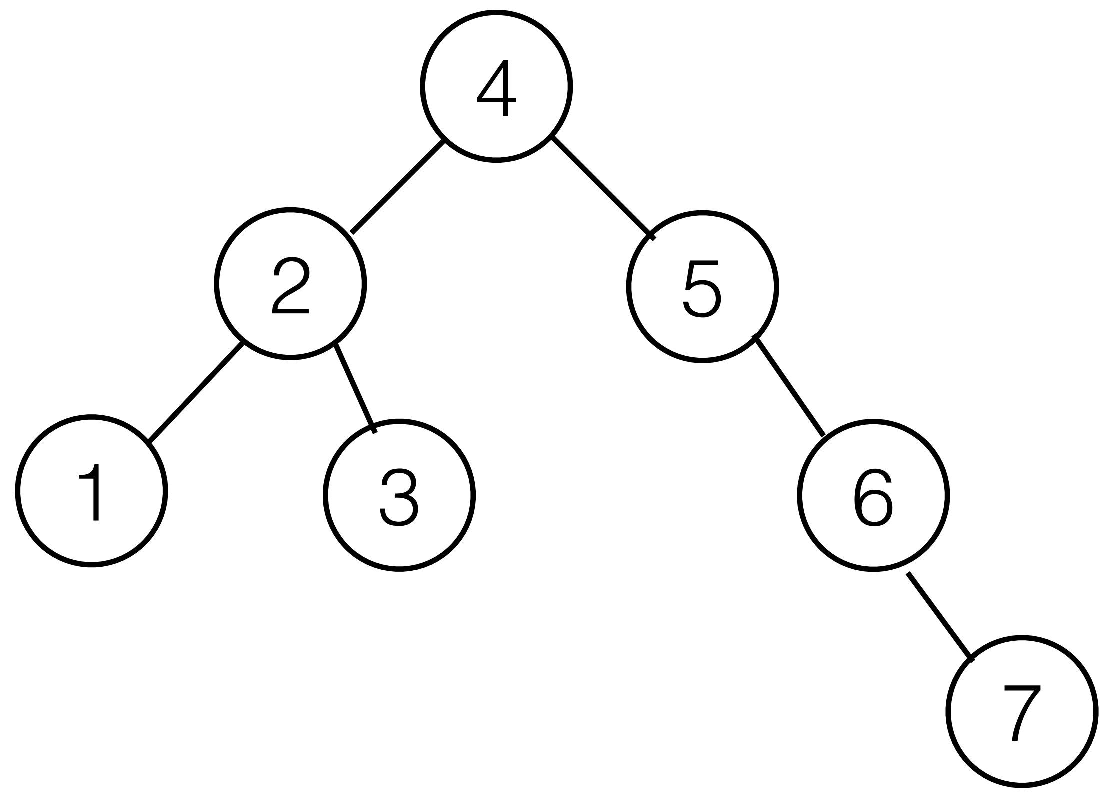
Single Rotation Example

insert(3)
insert(2)
insert(1) rotate_right(3)
insert(4)
insert(5) rotate_left(3)
insert(6) rotate_left(2)



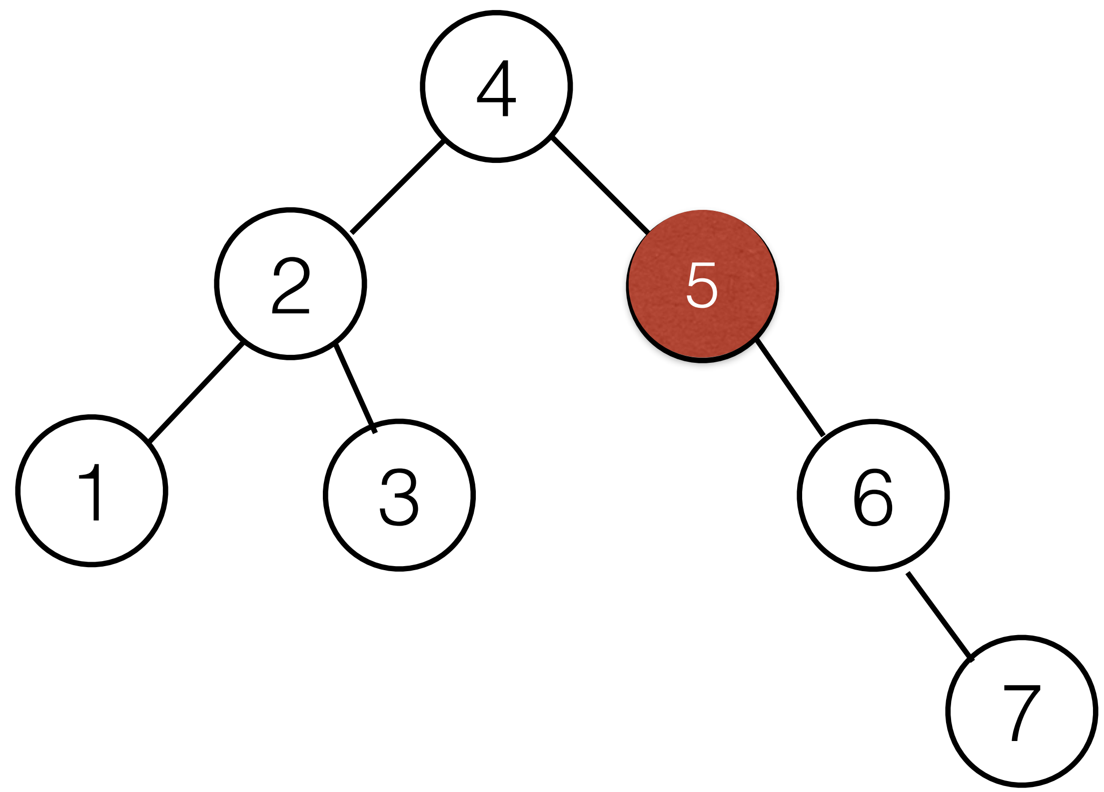
Single Rotation Example

insert(3)
insert(2)
insert(1) rotate_right(3)
insert(4)
insert(5) rotate_left(3)
insert(6) rotate_left(2)
insert(7)



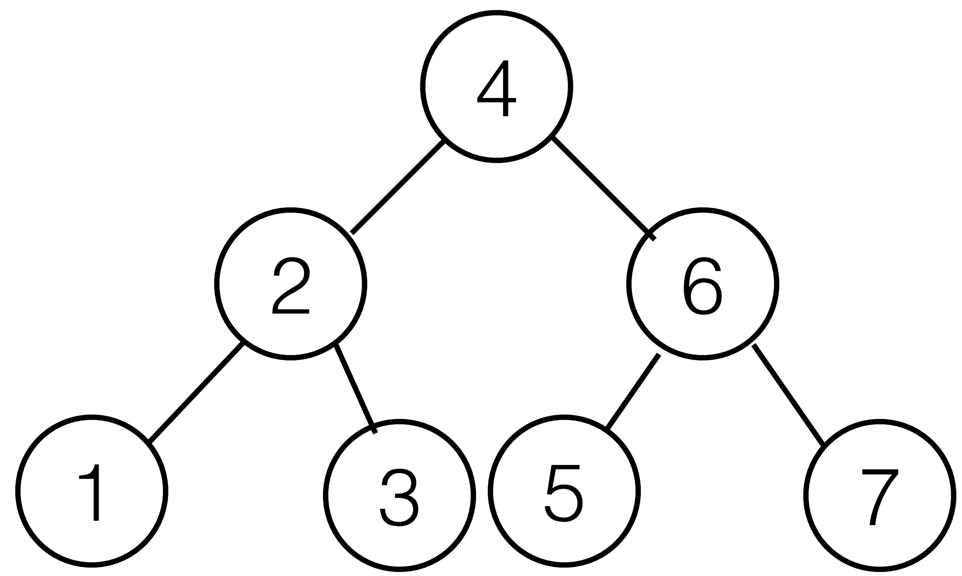
Single Rotation Example

insert(3)
insert(2)
insert(1) rotate_right(3)
insert(4)
insert(5) rotate_left(3)
insert(6) rotate_left(2)
insert(7) rotate_left(5)

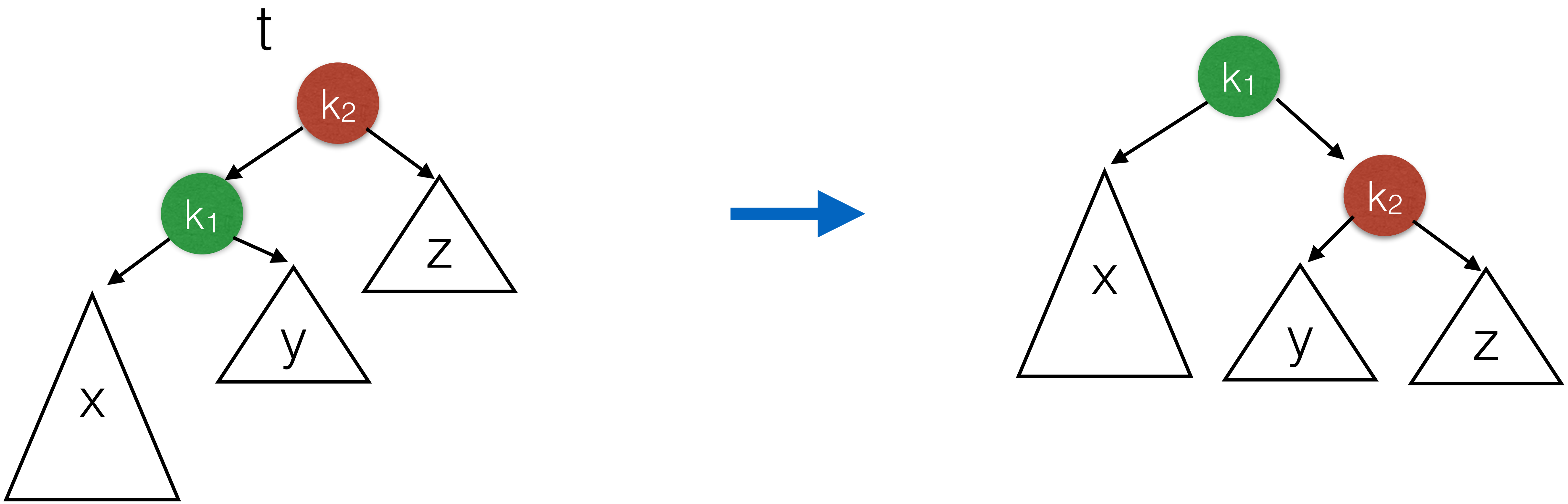


Single Rotation Example

insert(3)
insert(2)
insert(1) rotate_right(3)
insert(4)
insert(5) rotate_left(3)
insert(6) rotate_left(2)
insert(7) rotate_left(5)



Single Rotation (3)



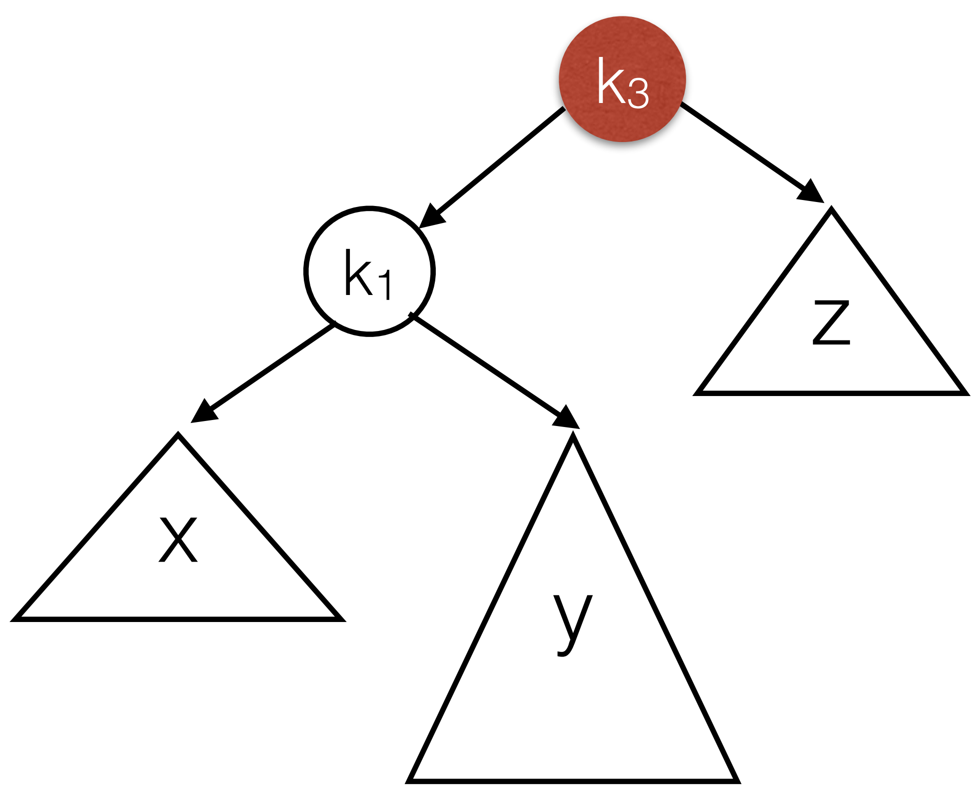
Which references do we need to modify?

- k_2.left = k_1.right
- k_1.right = k_2
- either parent(k_2).left = k_1 or parent(k_2).right = k_1

(see code)

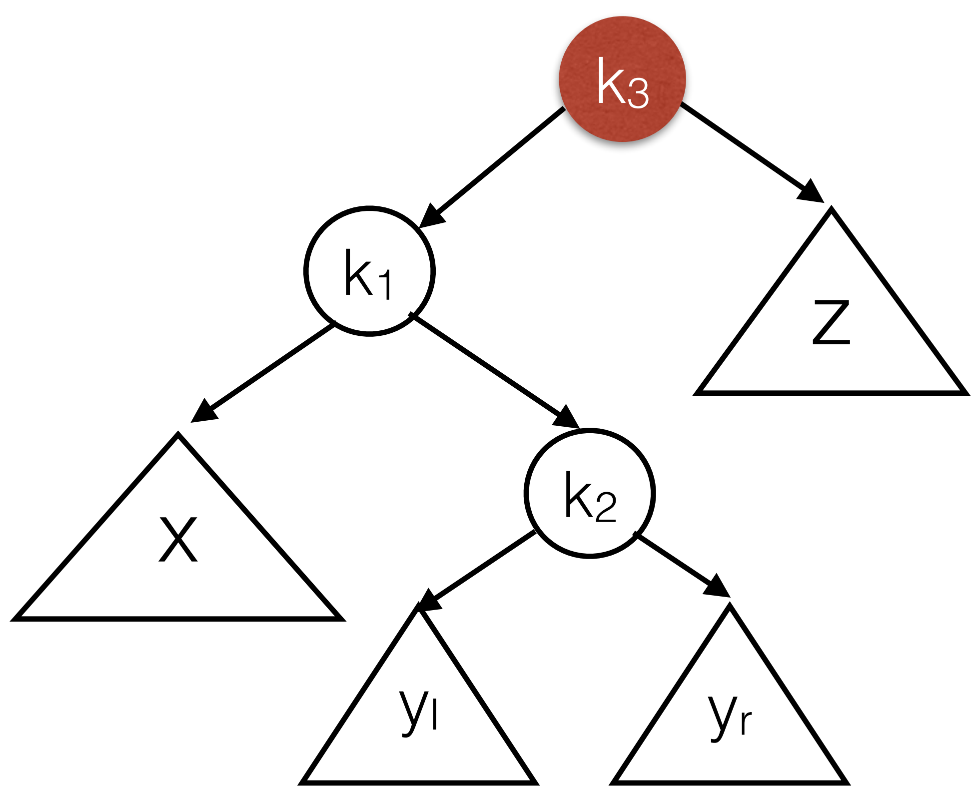
Double Rotation (1)

- y is non-empty (imbalance due to insertion into y or deletion from z)
- so we can view y as a root and two sub-trees.



Double Rotation (1)

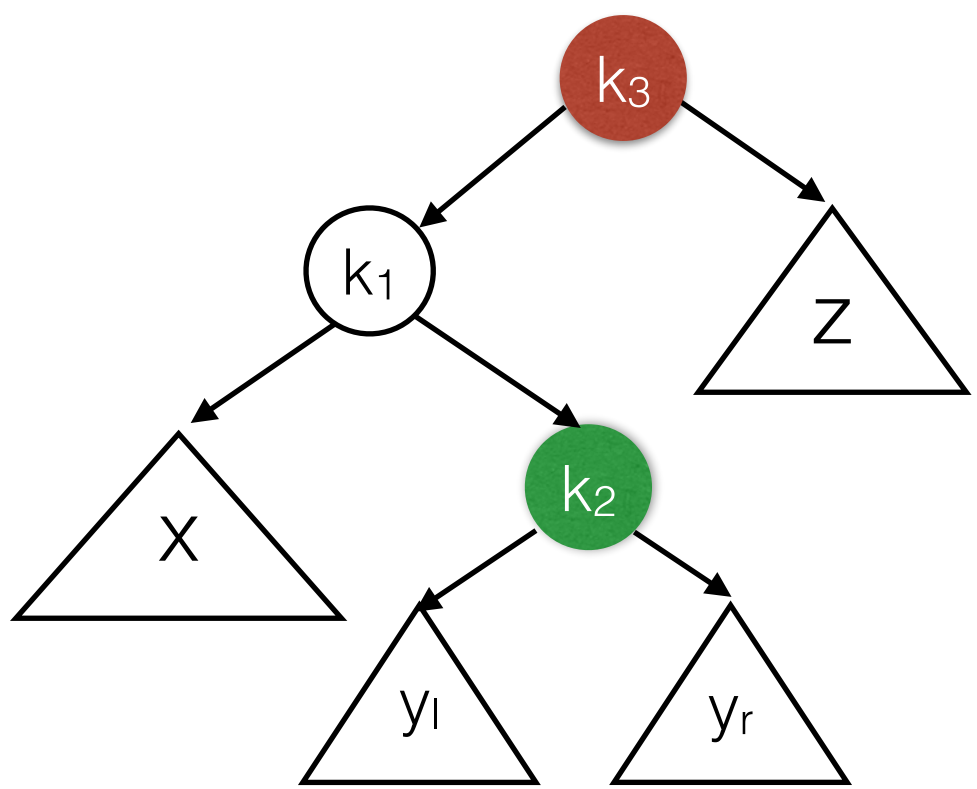
- y is non-empty (imbalance due to insertion into y or deletion from z)
- so we can view y as a root and two sub-trees.



- either y_l or y_r is two level higher than z

Double Rotation (1)

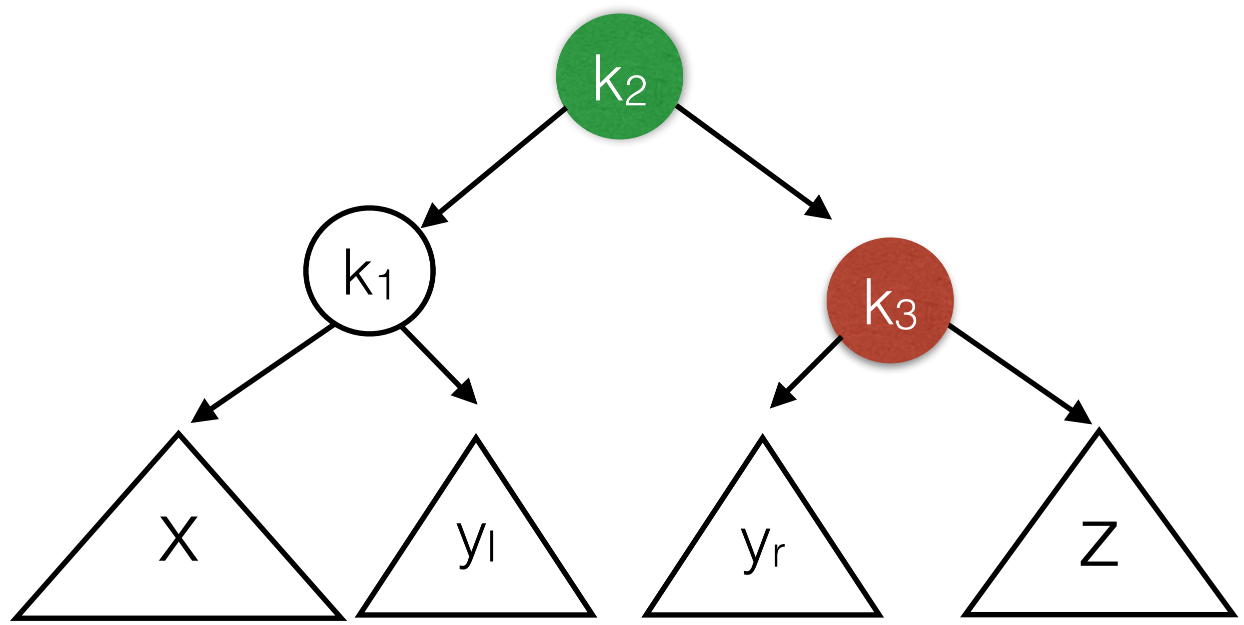
- y is non-empty (imbalance due to insertion into y or deletion from z)
- so we can view y as a root and two sub-trees.



- either y_l or y_r is two level higher than z

Double Rotation (1)

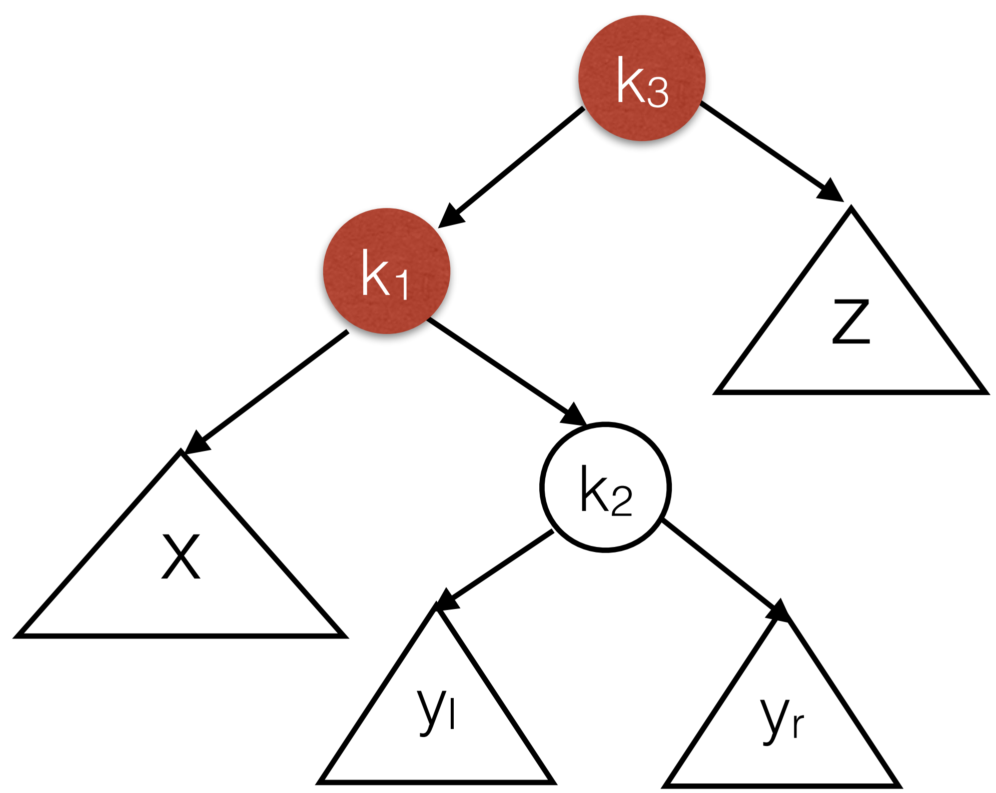
- y is non-empty (imbalance due to insertion into y or deletion from z)
- so we can view y as a root and two sub-trees.



- either y_l or y_r is two level higher than z

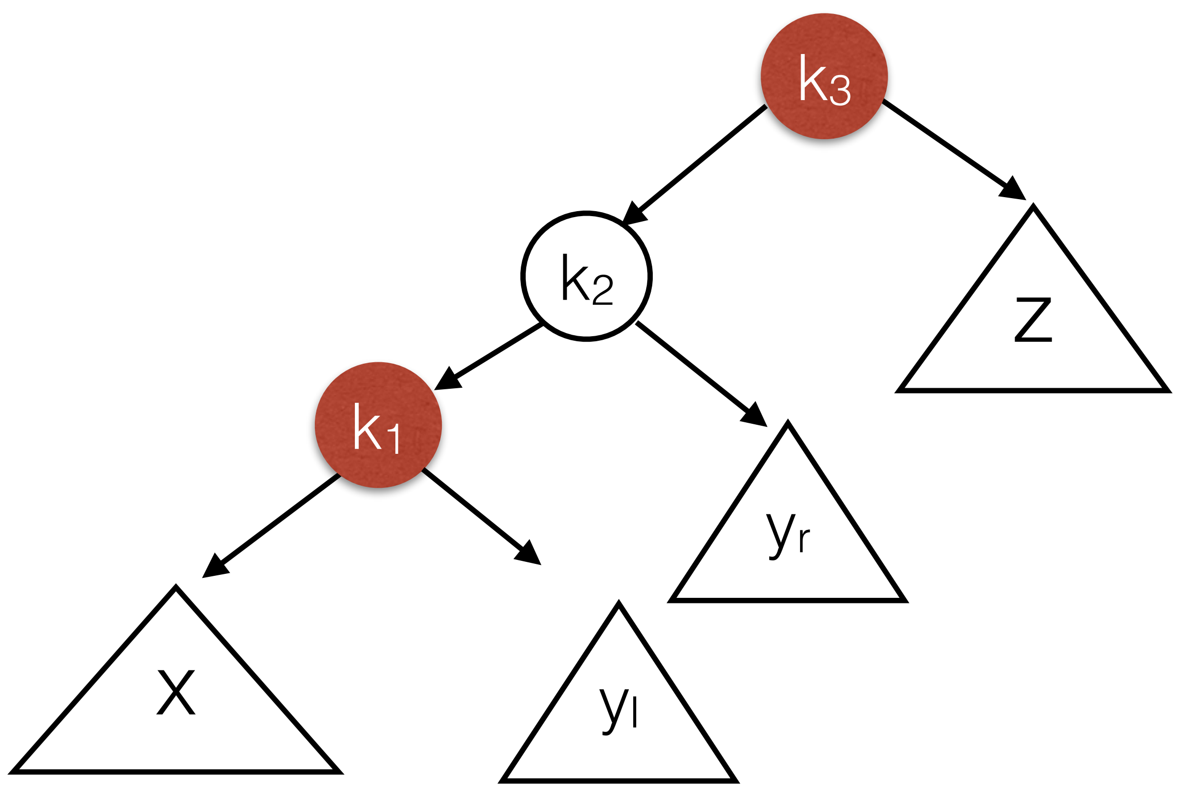
Double Rotation (2)

Can also think of this as two single rotations:
First at k_1, then at k_3.



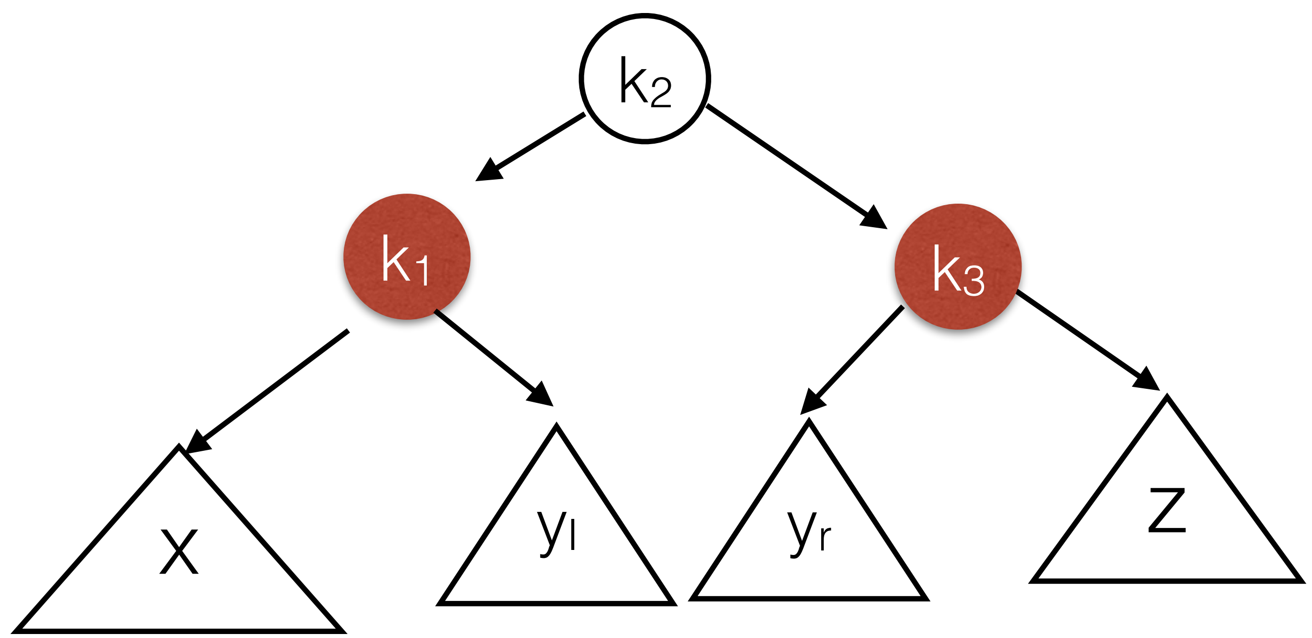
Double Rotation (2)

Can also think of this as two single rotations:
First at k_1, then at k_3.

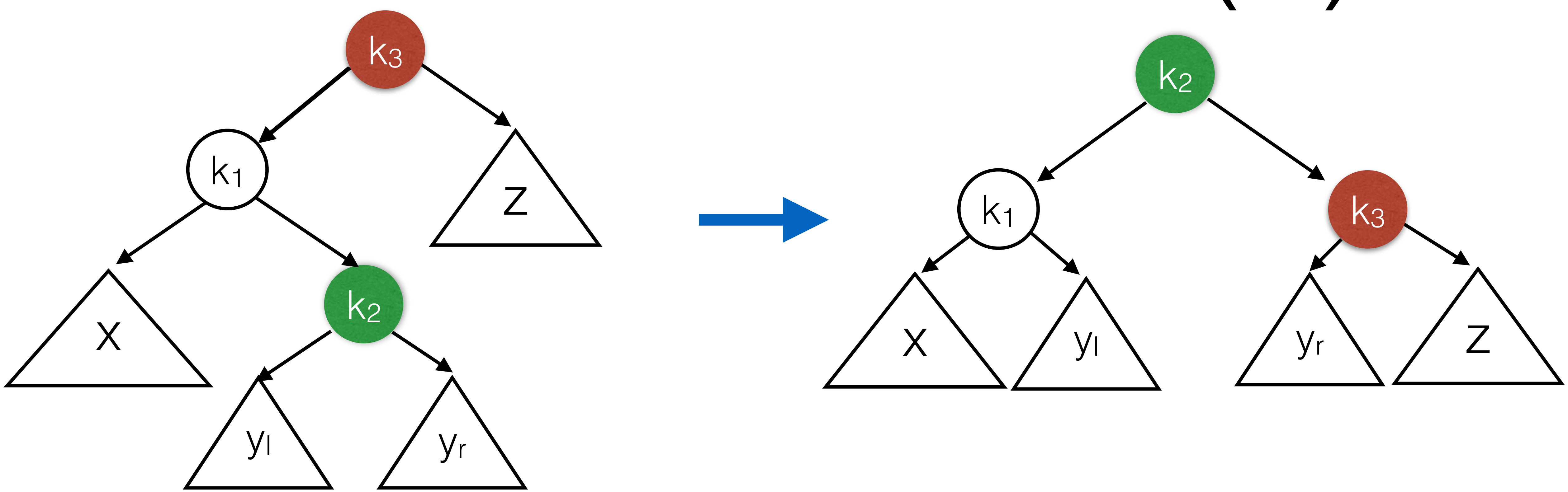


Double Rotation (2)

Can also think of this as two single rotations:
First at k_1, then at k_3.



Double Rotation (3)



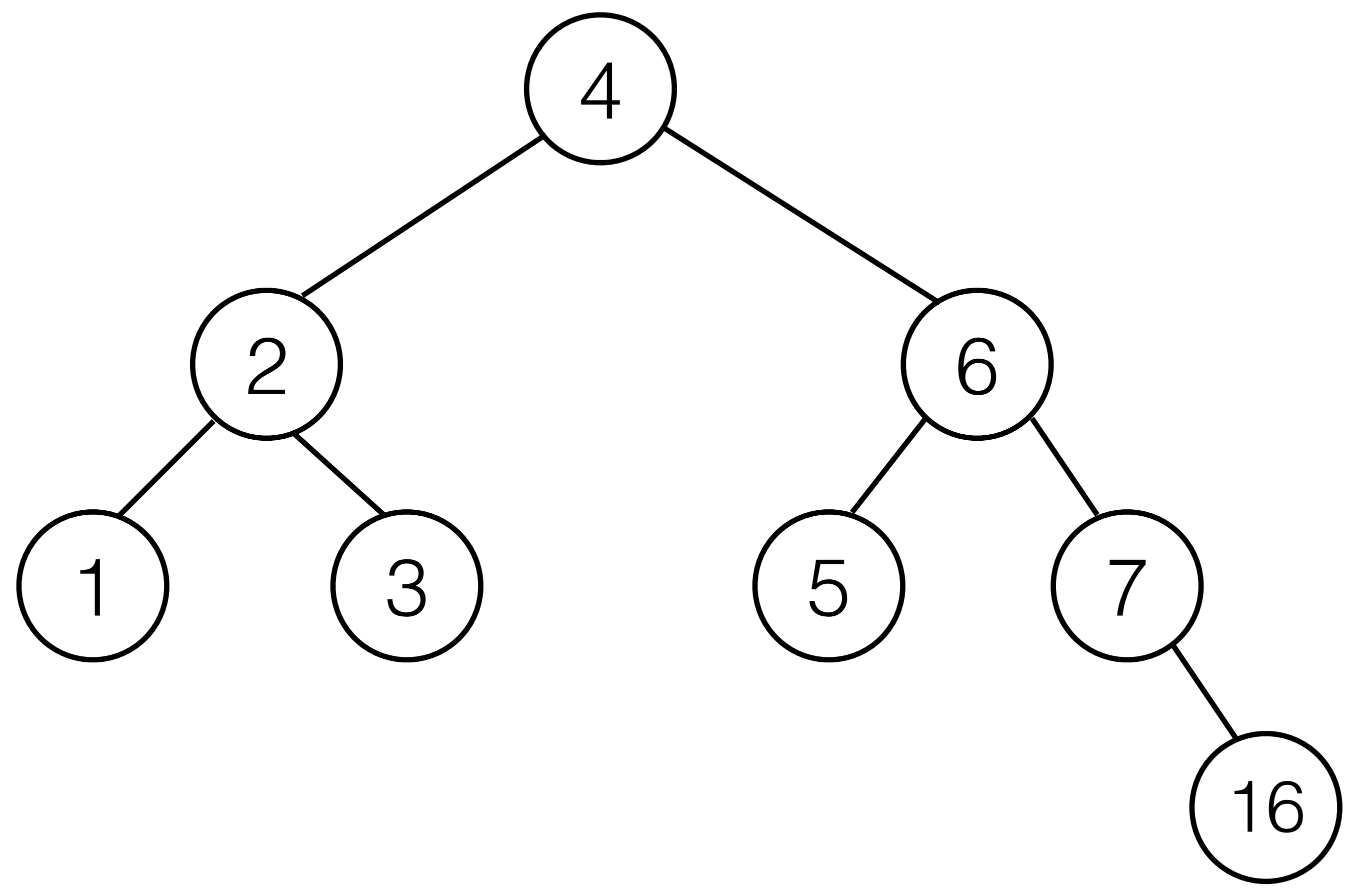
Which references do we need to modify?

- k_2.left = k_1
- k_2.right = k_3
- k_1.right = root(y_l)
- k_3.left = root(y_r)
- parent(k_3).left = k_2 or parent(k_3).right = k_2

(see code)

Double Rotation Example

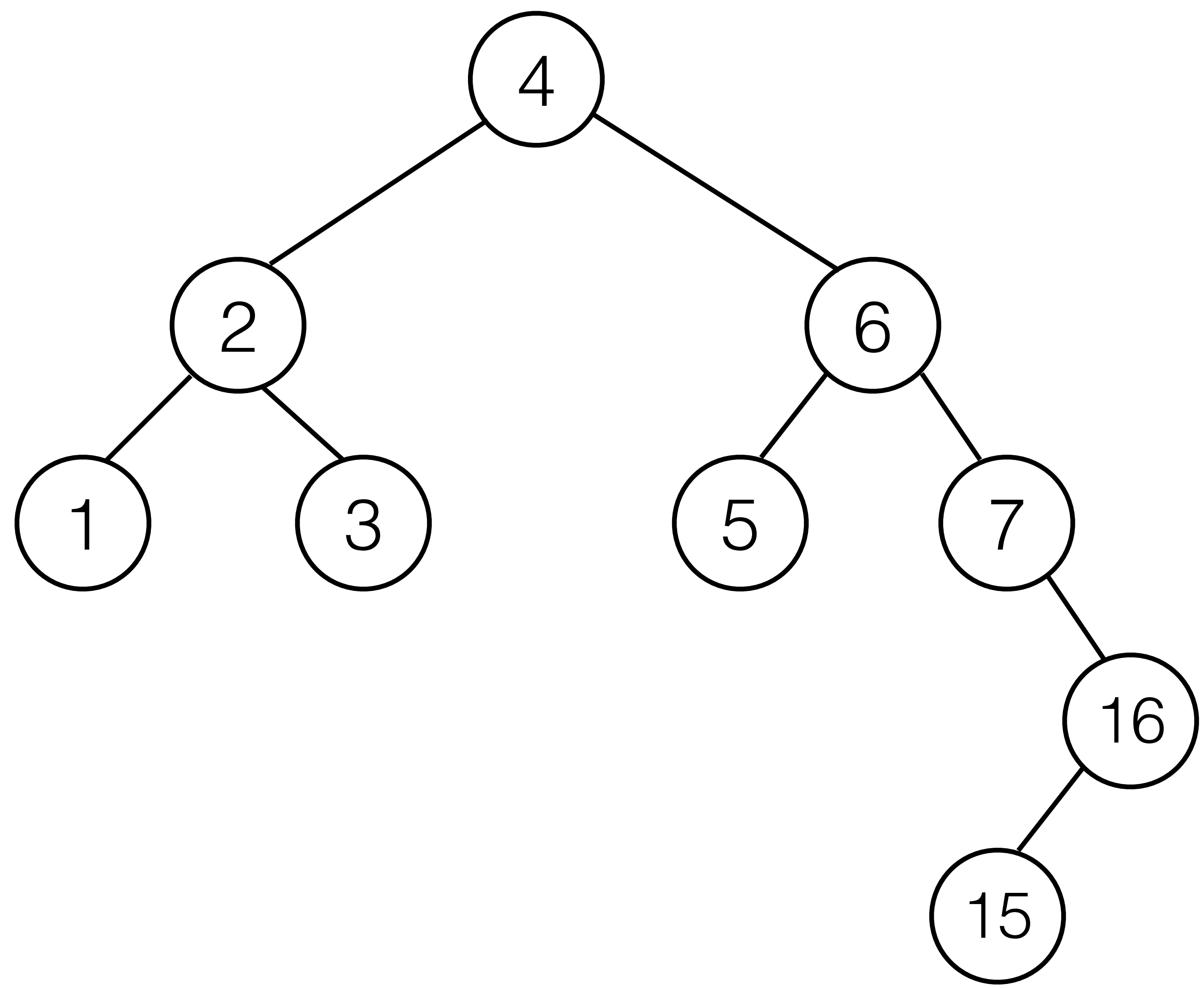
insert(16)



Double Rotation Example

insert(16)

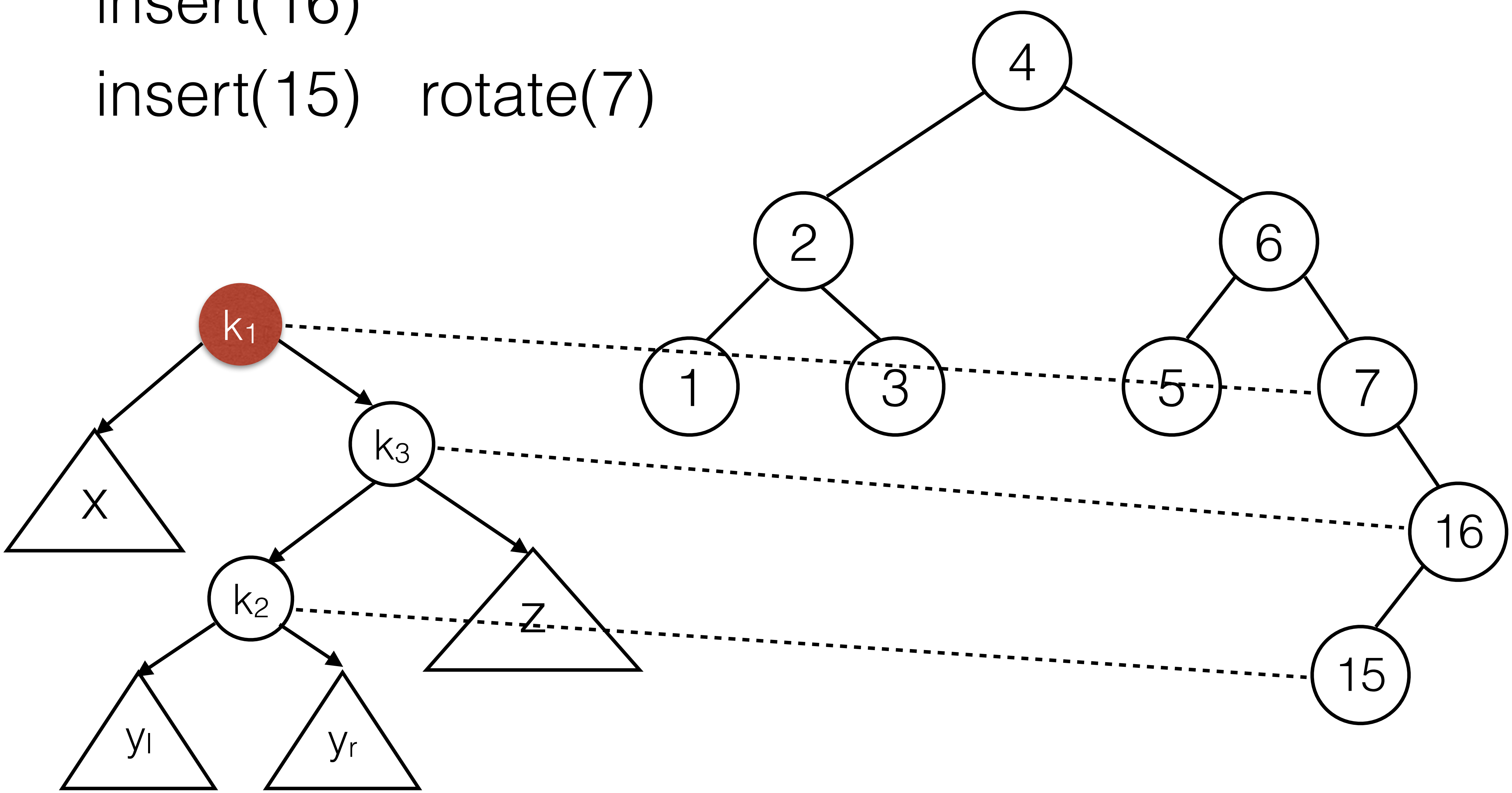
insert(15)



Double Rotation Example

insert(16)

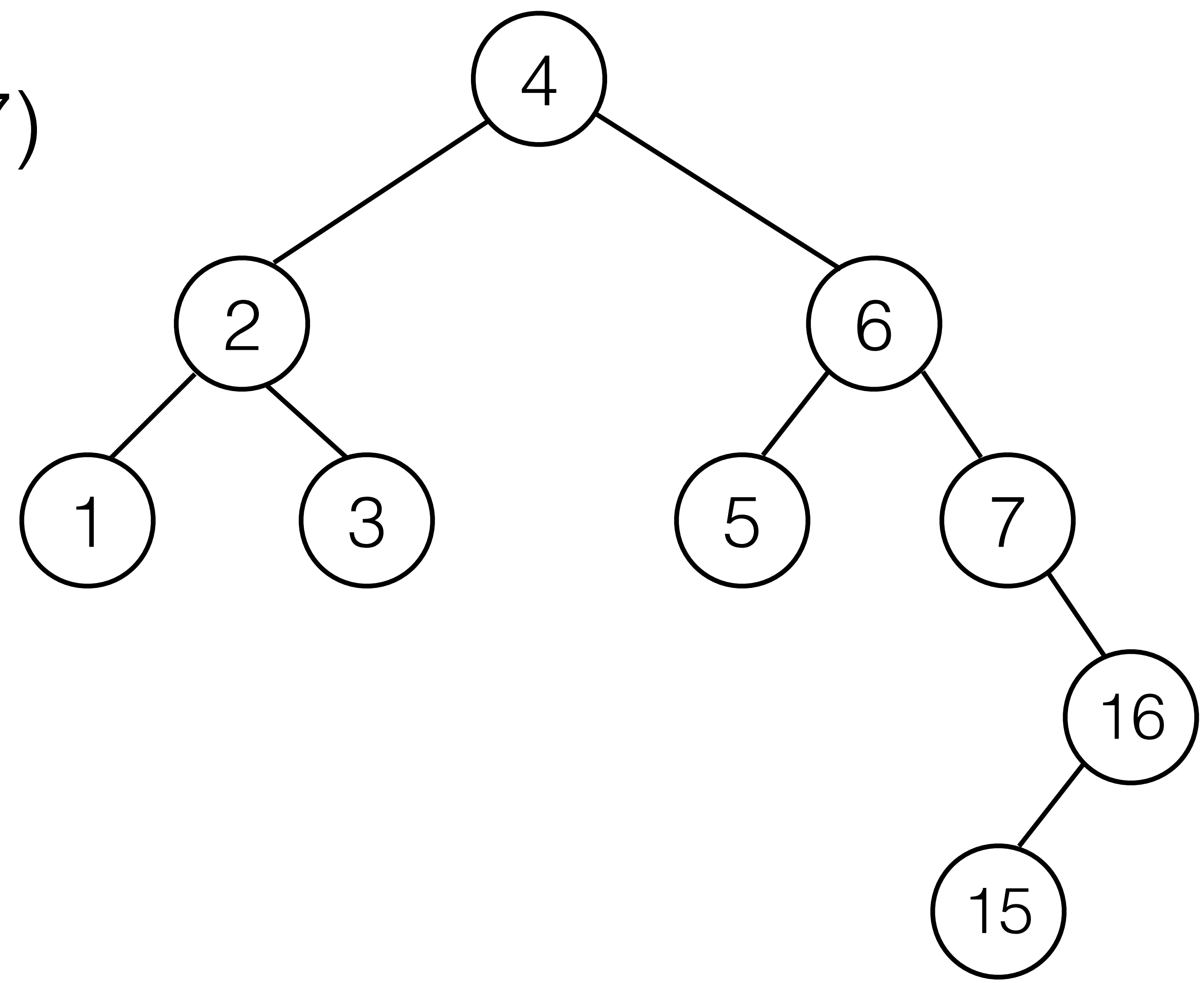
insert(15)   rotate(7)



Double Rotation Example

insert(16)

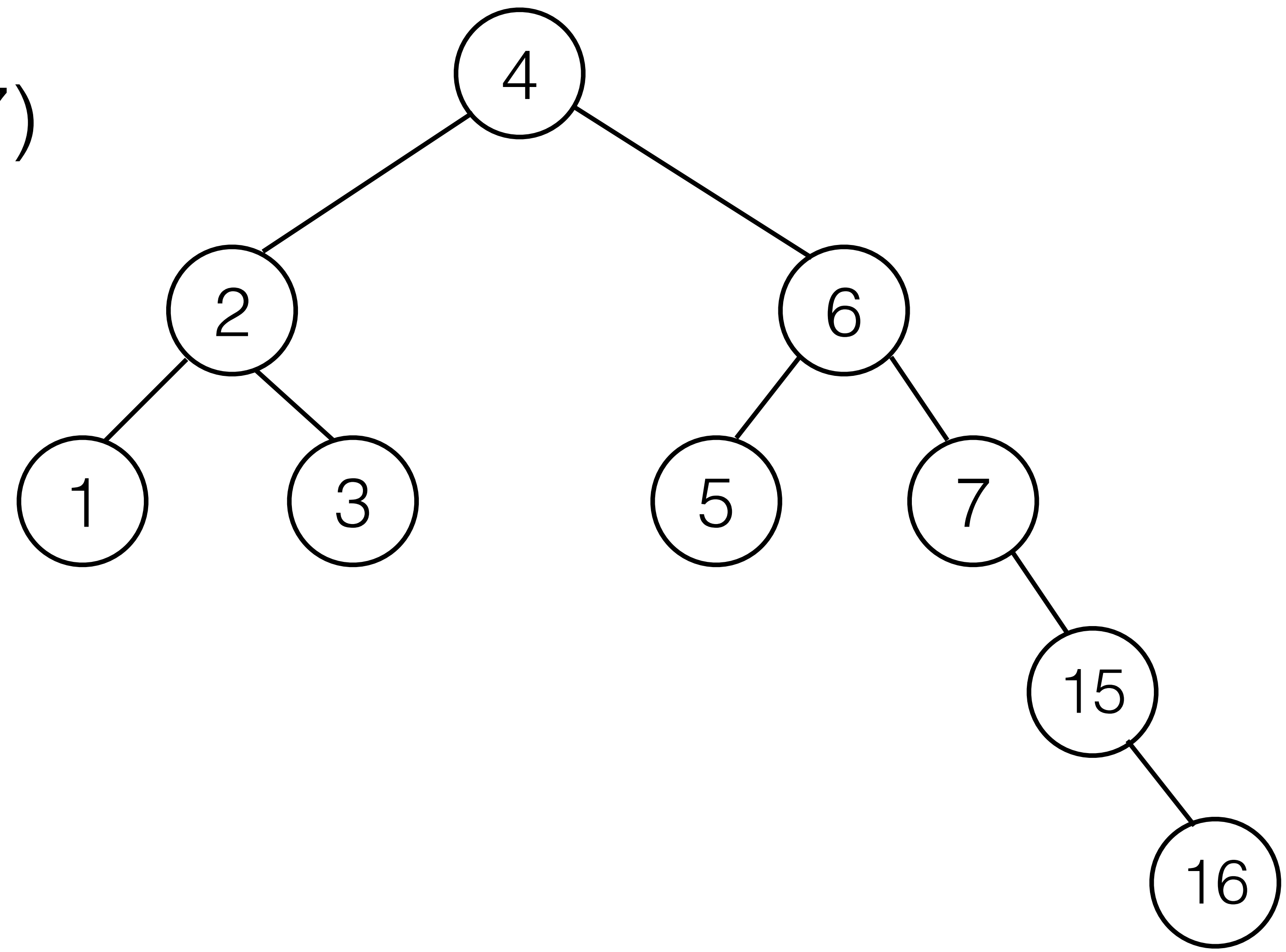
insert(15)   rotate(7)



Double Rotation Example

insert(16)

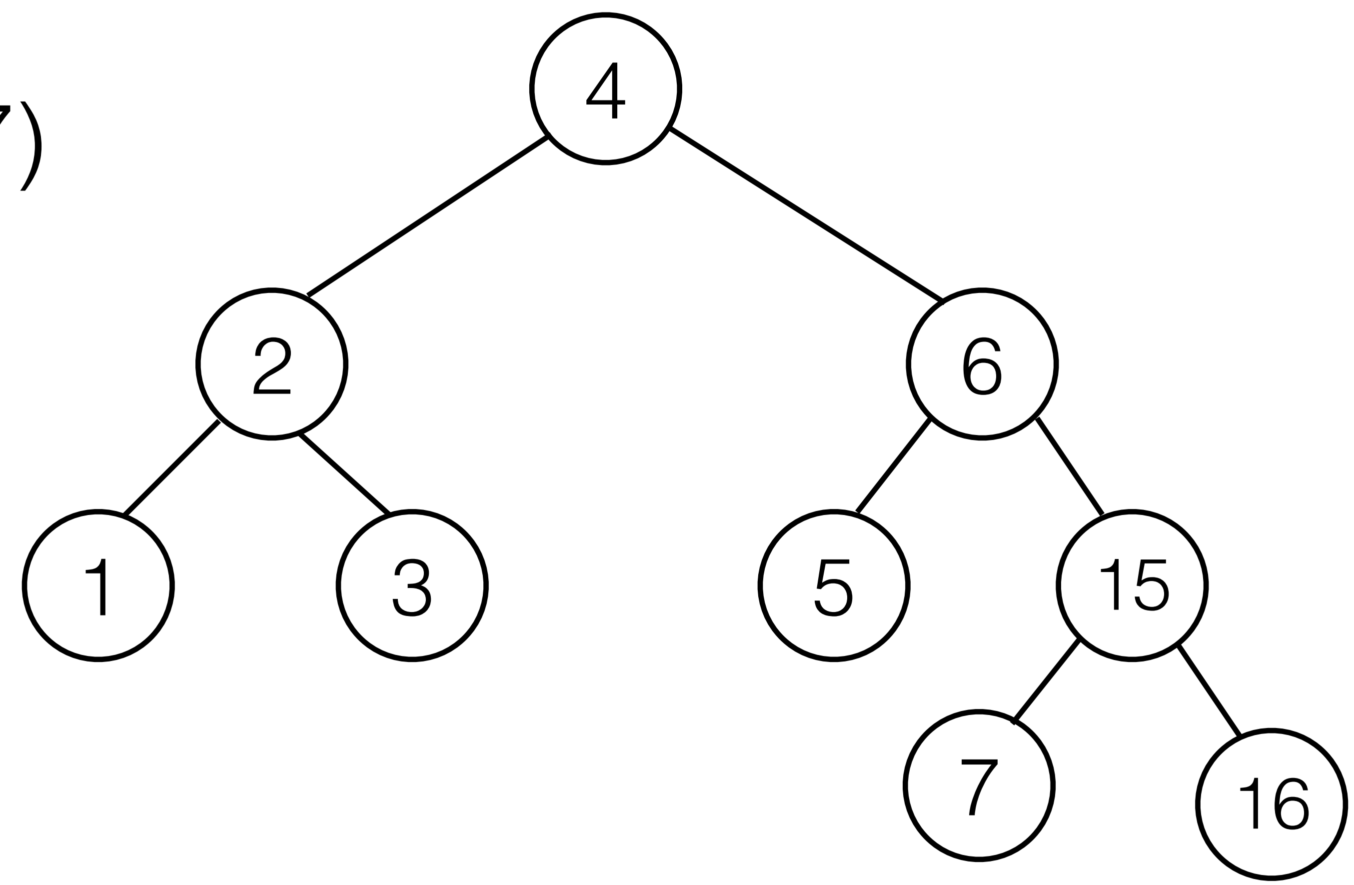
insert(15)   rotate(7)



Double Rotation Example

insert(16)

insert(15)   rotate(7)

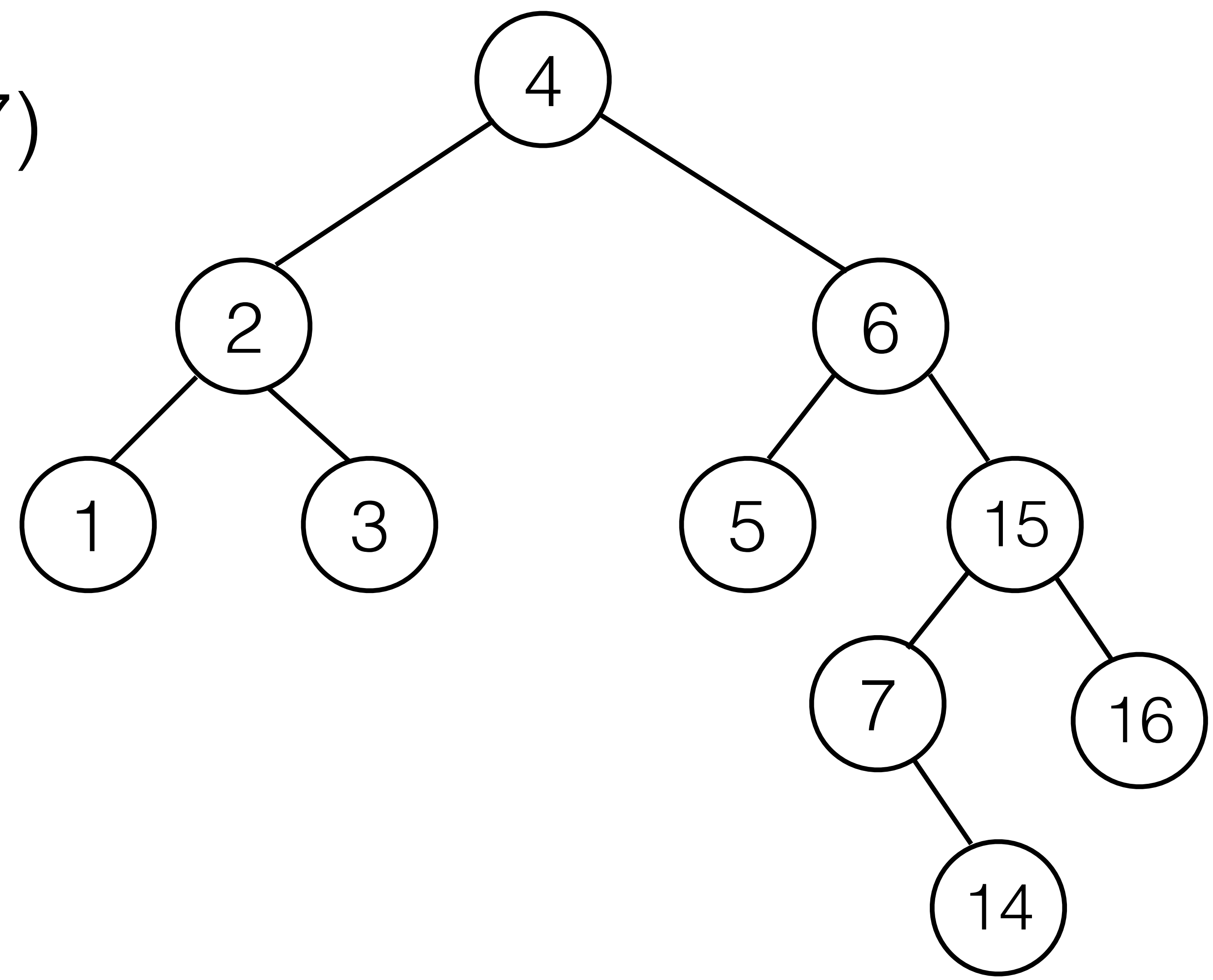


Double Rotation Example

insert(16)

insert(15) rotate(7)

insert(14)

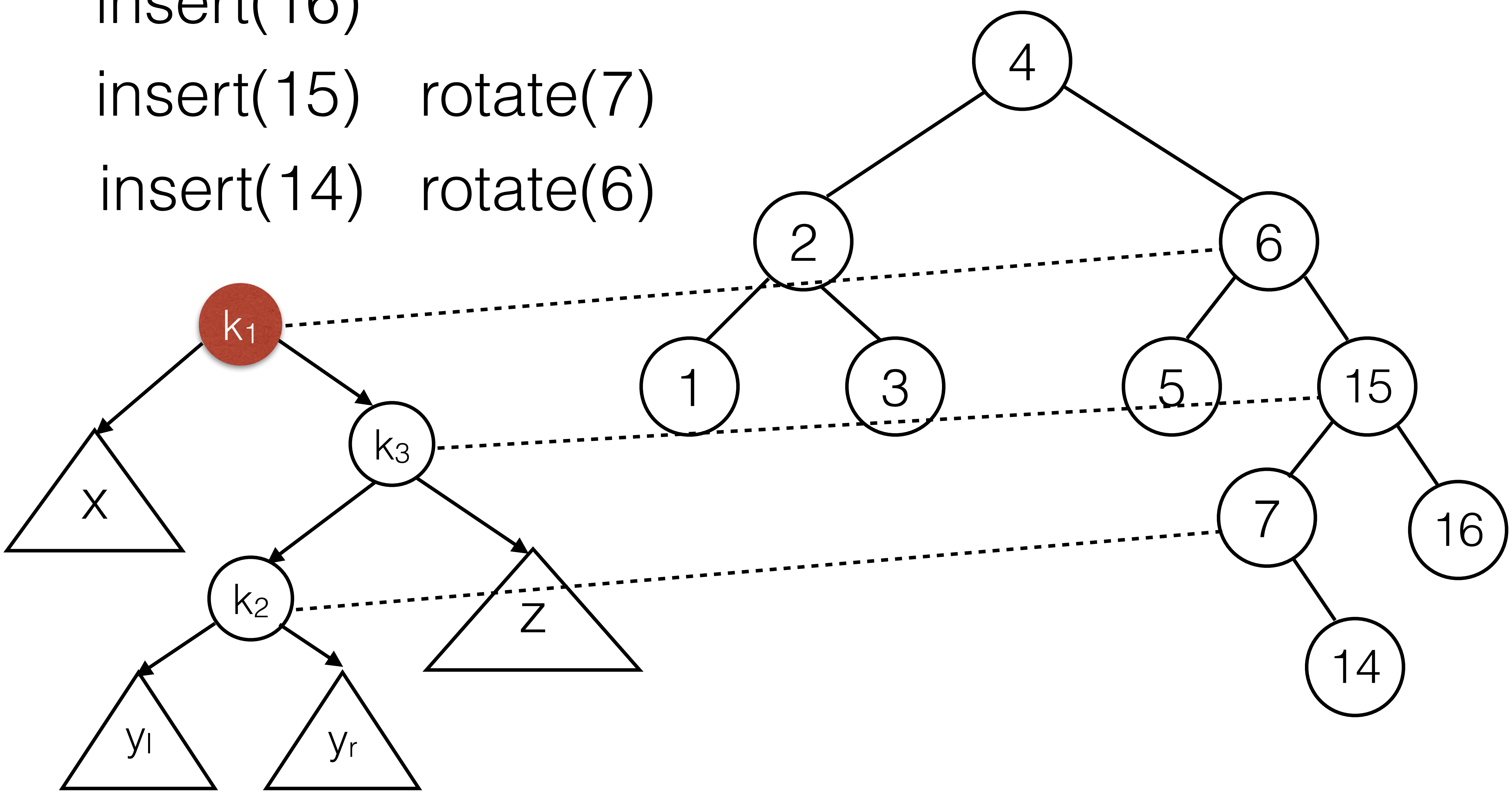


Double Rotation Example

insert(16)

insert(15) rotate(7)

insert(14) rotate(6)

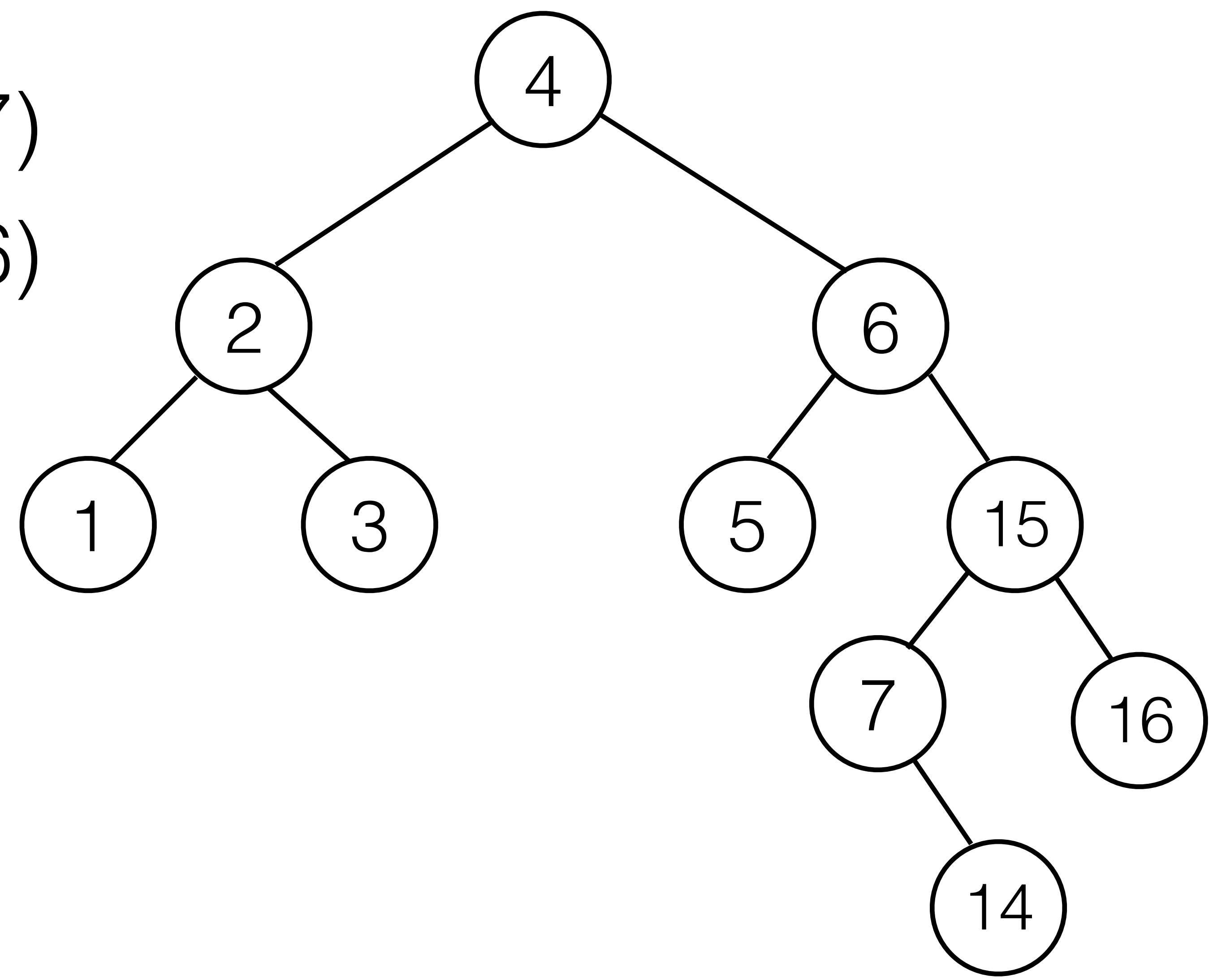


Double Rotation Example

insert(16)

insert(15)   rotate(7)

insert(14)   rotate(6)

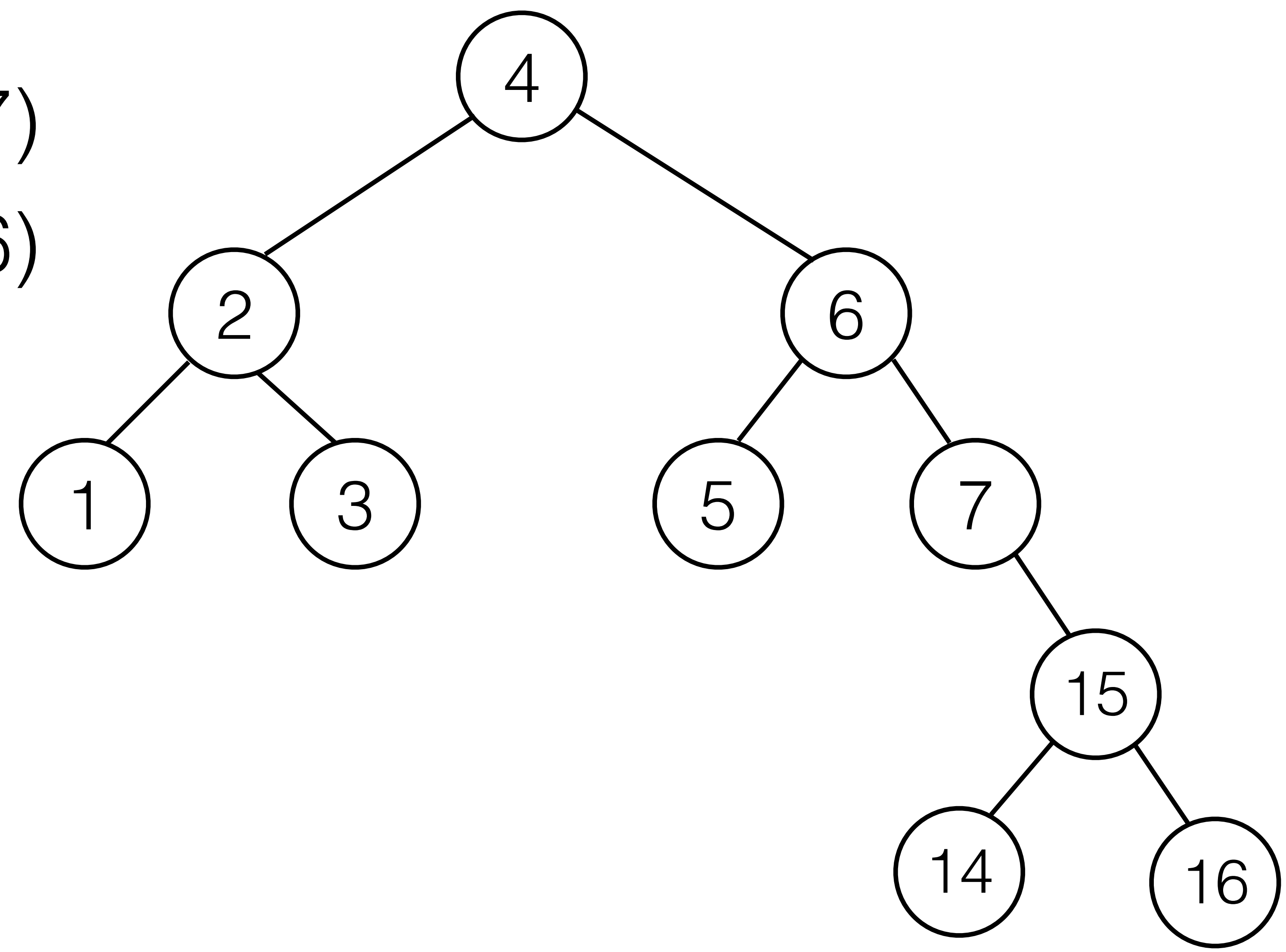


Double Rotation Example

insert(16)

insert(15) rotate(7)

insert(14) rotate(6)



Double Rotation Example

insert(16)

insert(15)   rotate(7)

insert(14)   rotate(6)