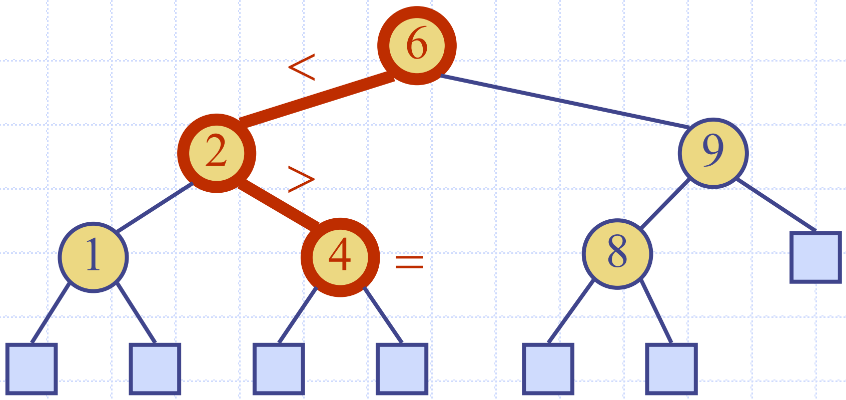
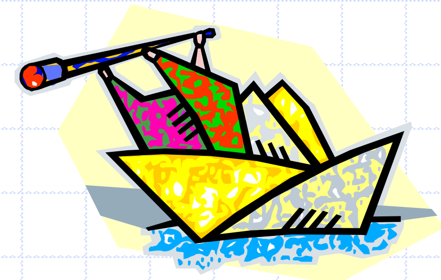
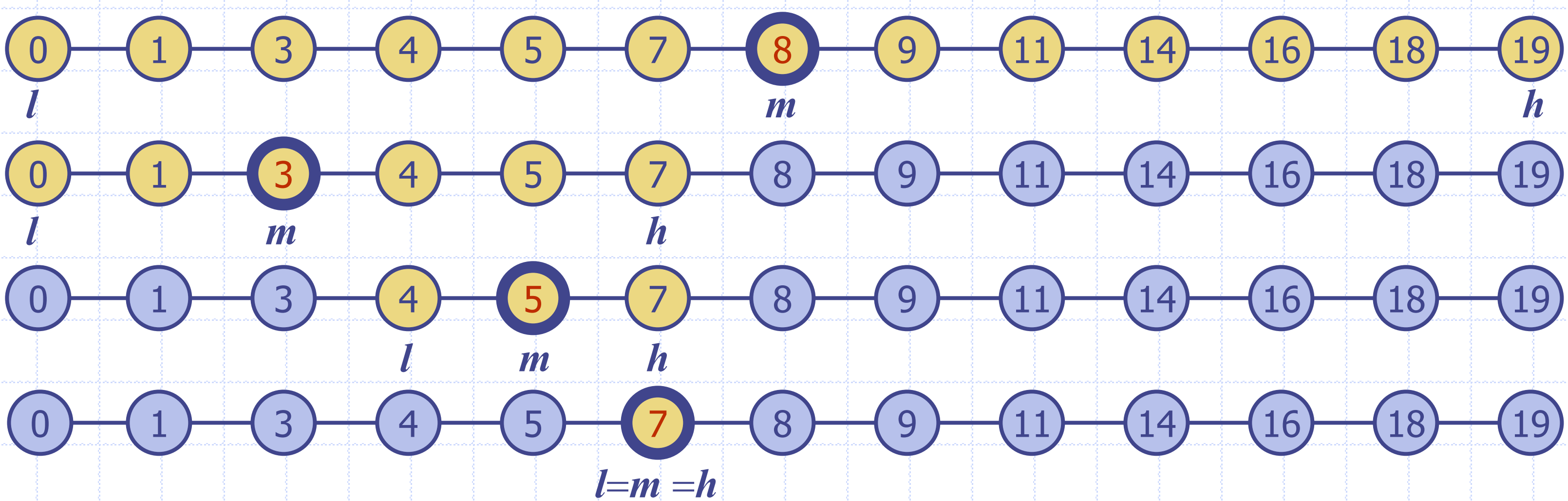
Binary Search Trees



Binary Search

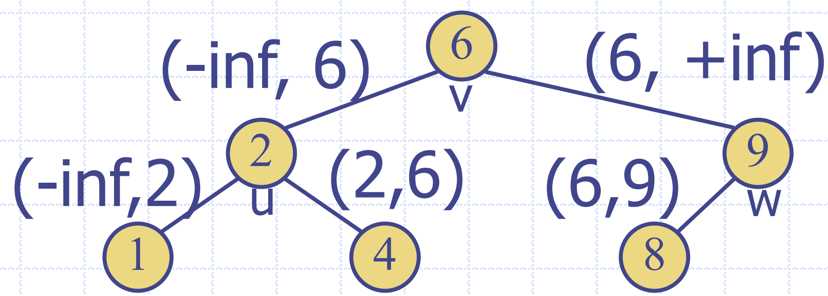


- Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
  - similar to the high-low children's game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: find(7)

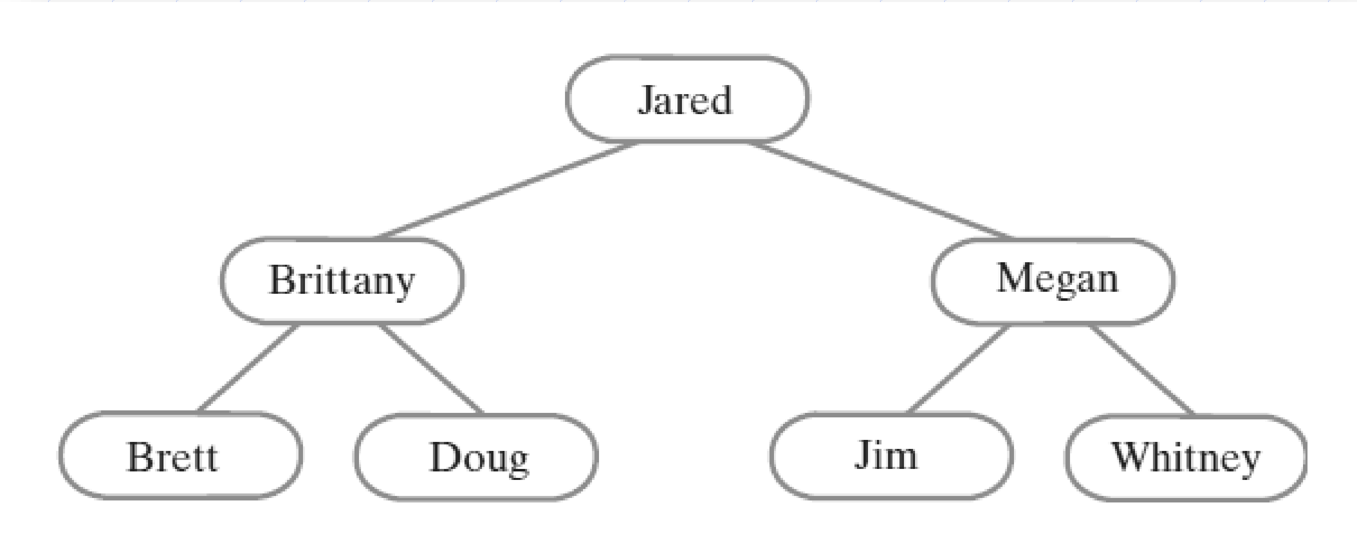


Binary Search Trees

- A binary search tree is a binary tree storing keys (or key-value items) at its nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) < key(v) < key(w)$
- An inorder traversal of a binary search trees visits the keys in increasing order

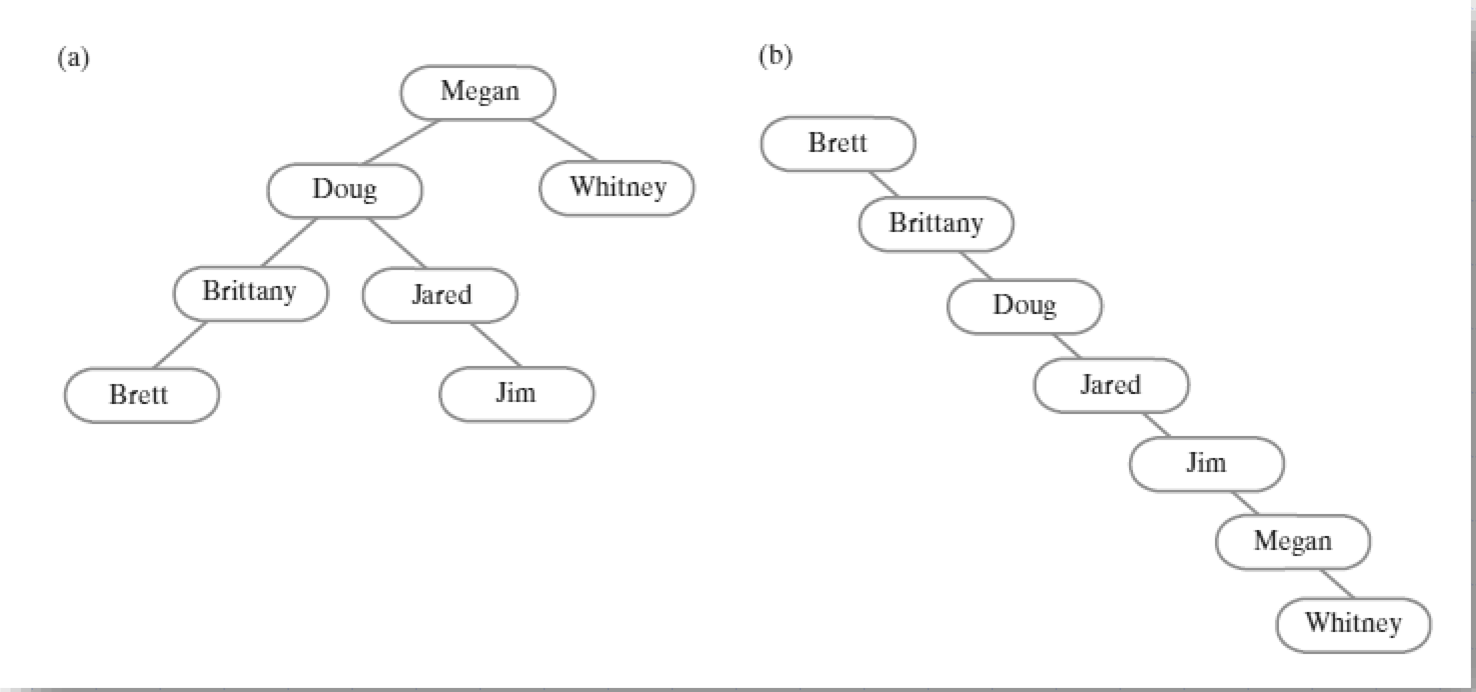


Binary Search Tree



A binary search tree of names

Binary Search Tree

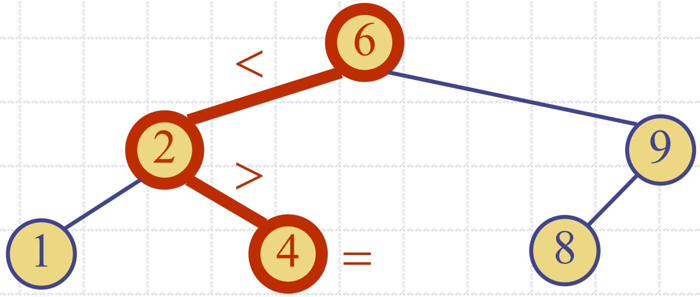


Two BSTs containing the same data as the previous BST

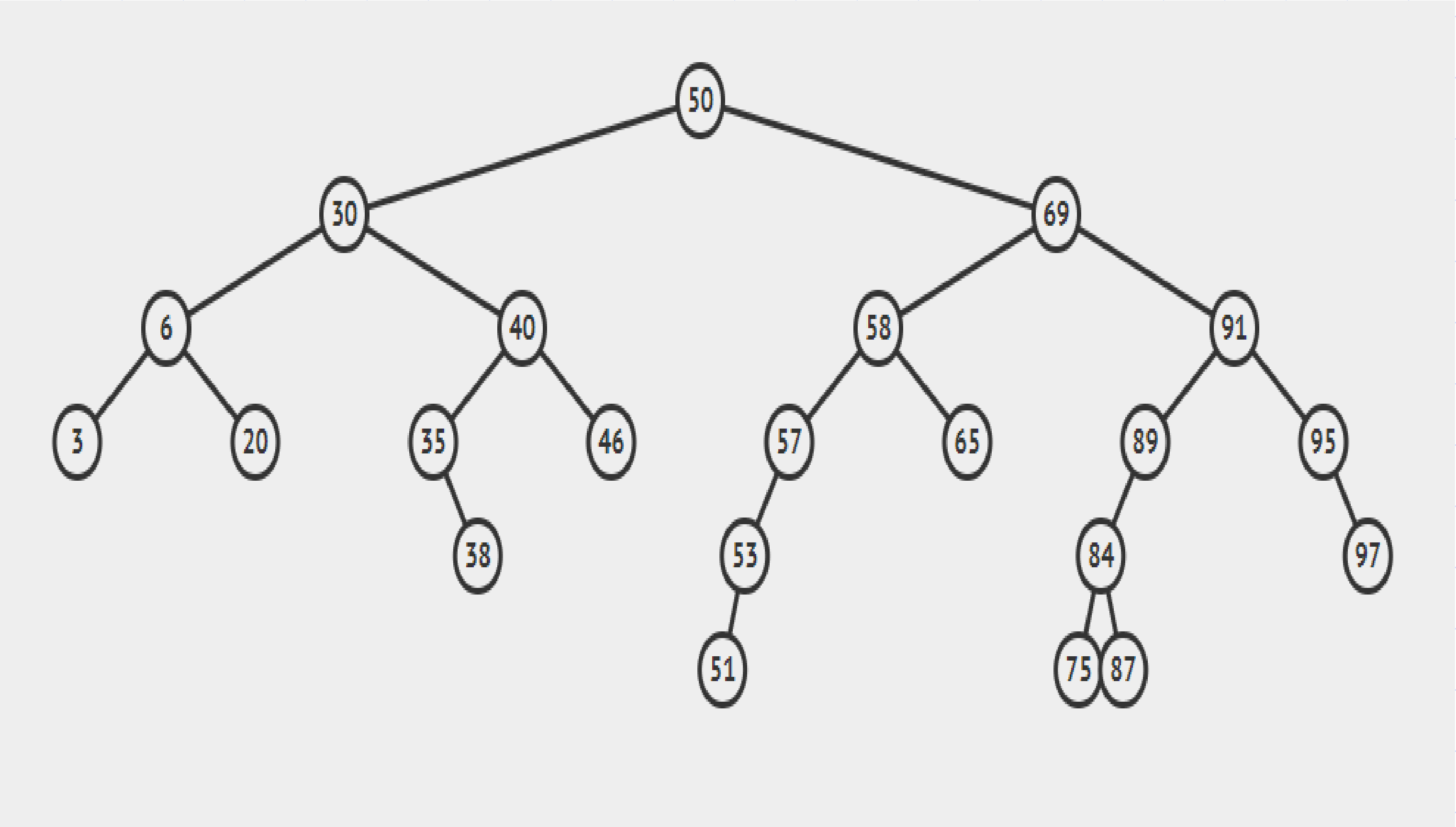
Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach a leaf (square box), the key is not found
- Example: find(4):
  - Call TreeSearch(T, T.root(), 4)
- The algorithms for nearest neighbor queries are similar

```
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}
```

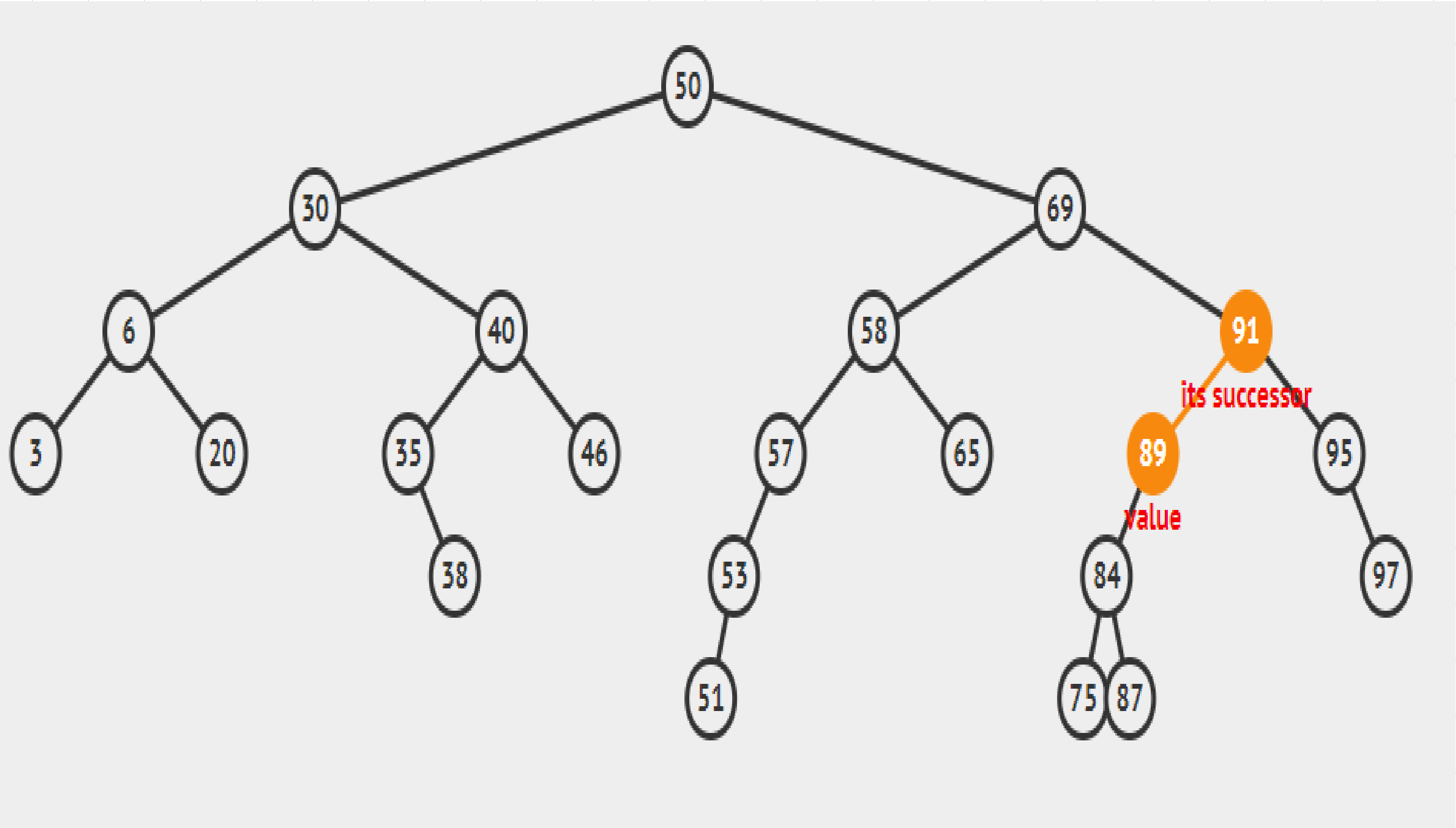


Successor of a node after(p)

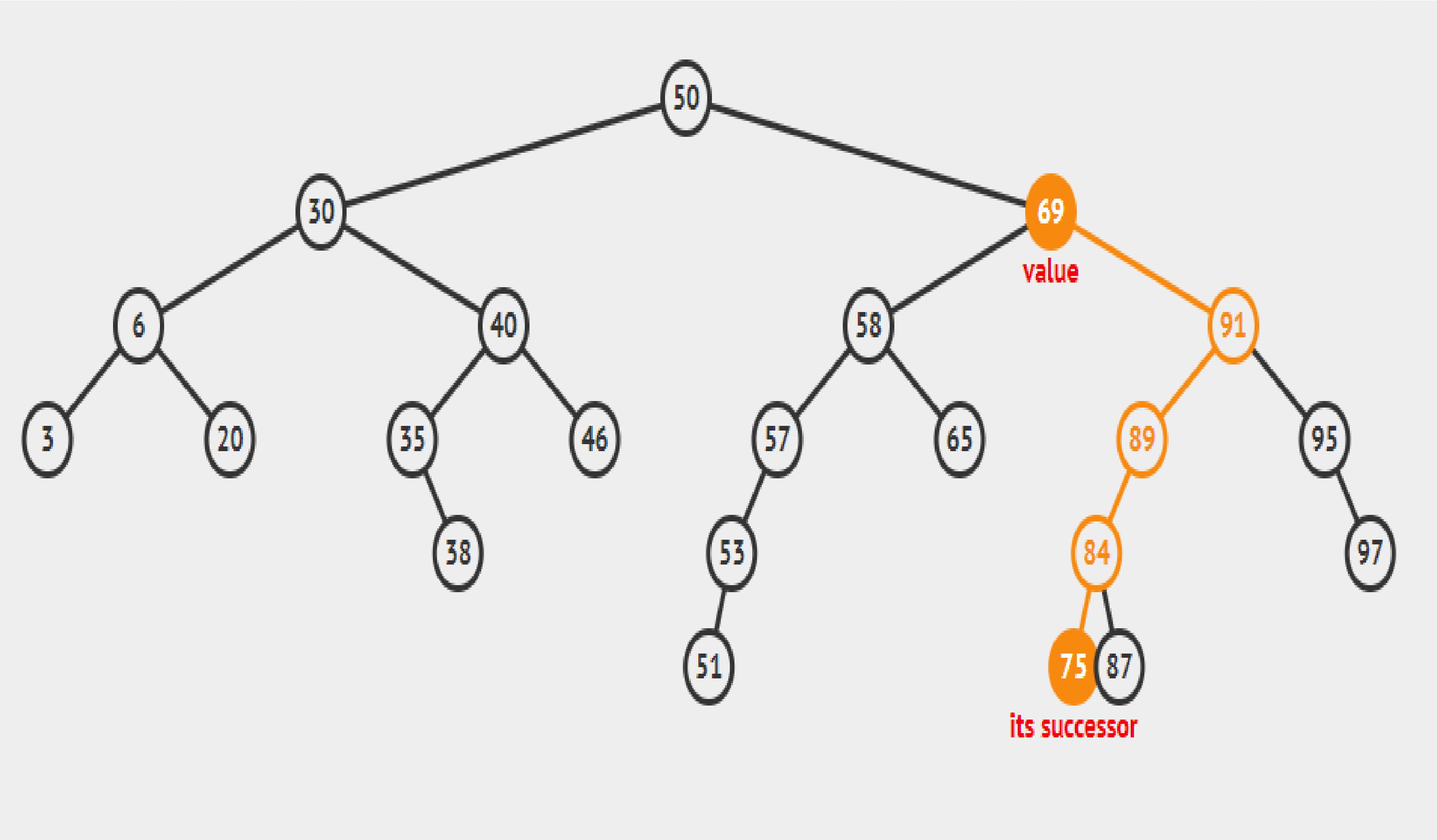


In-order traversal of BST results in a sorted sequence

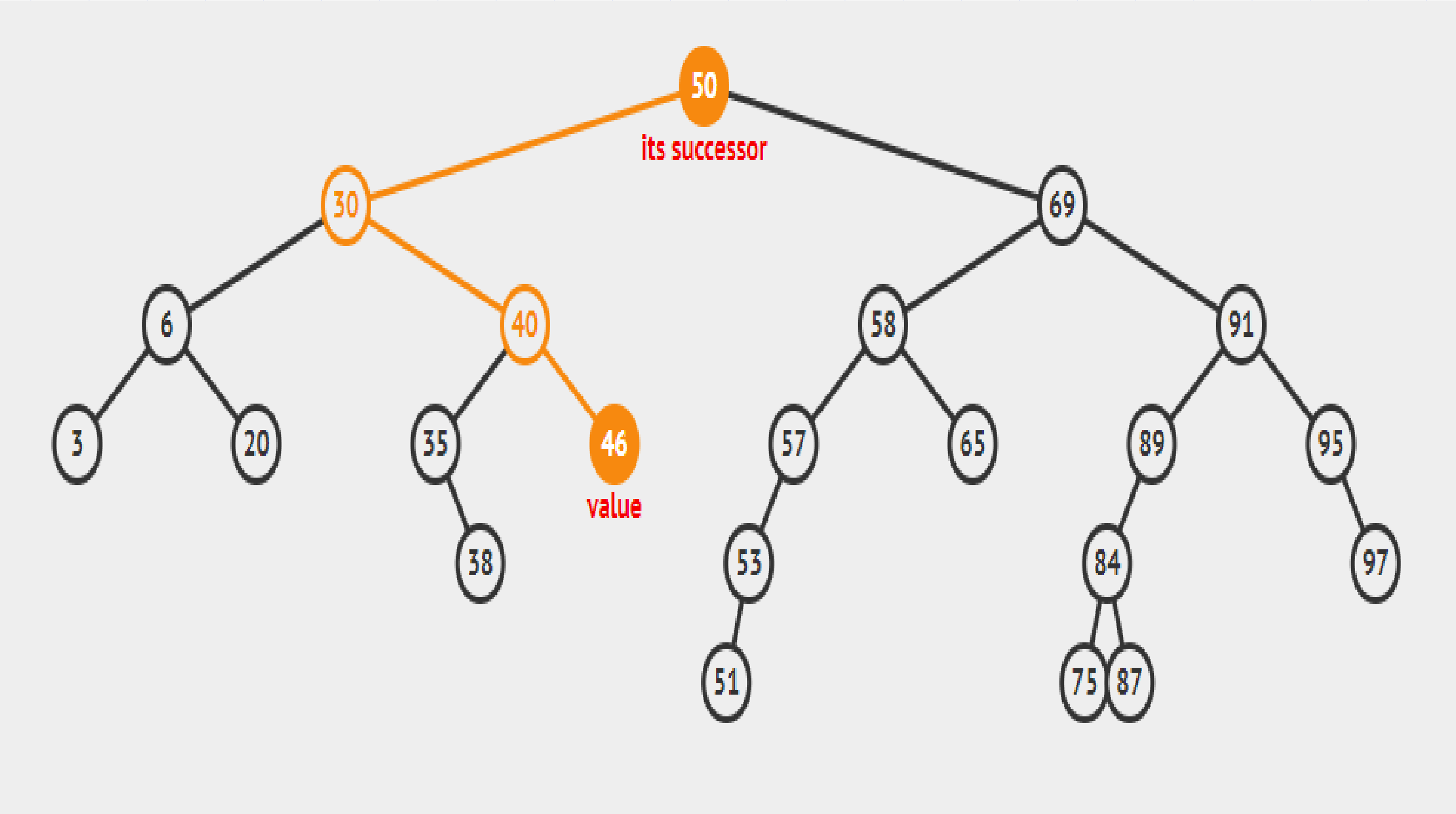
Successor of 89 is 91
after(89) = 91



Successor of 69 is 75
after(69) = 75



Successor of 46 is 50
after(46) = 50



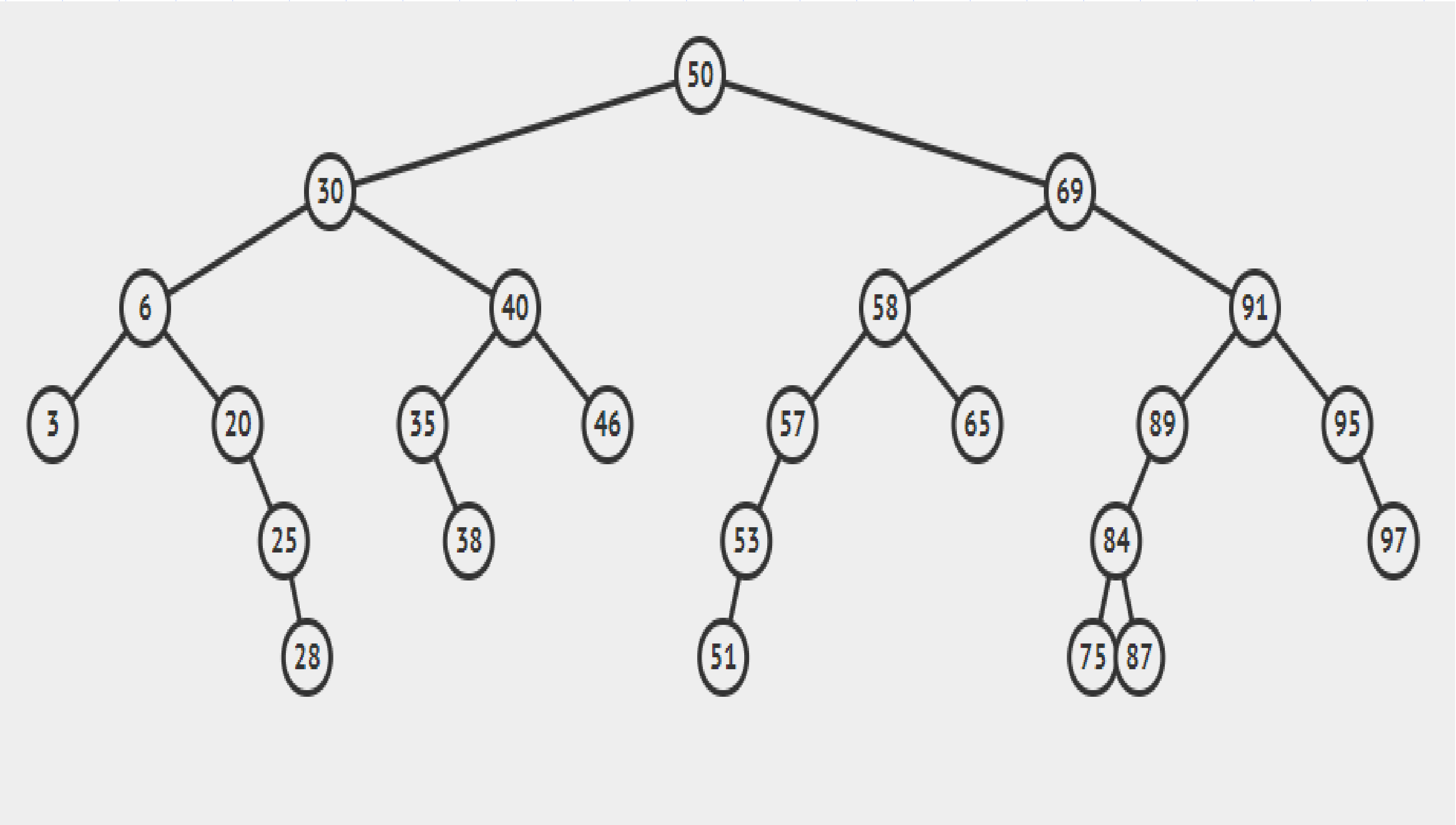
Algorithm for after(p)

```
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
```

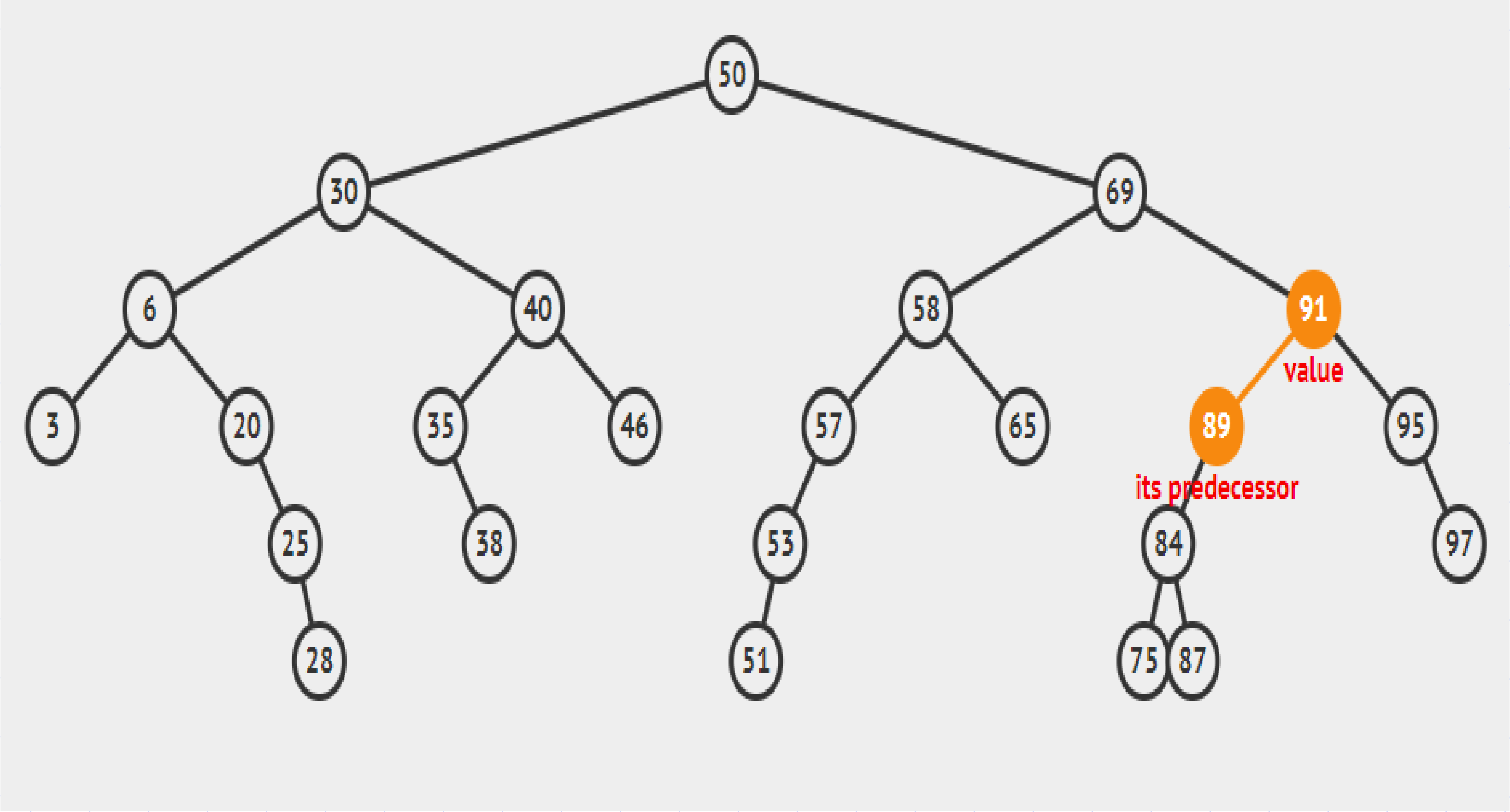
Find the smallest node in left subtree

Find the ancestor that has a bigger value

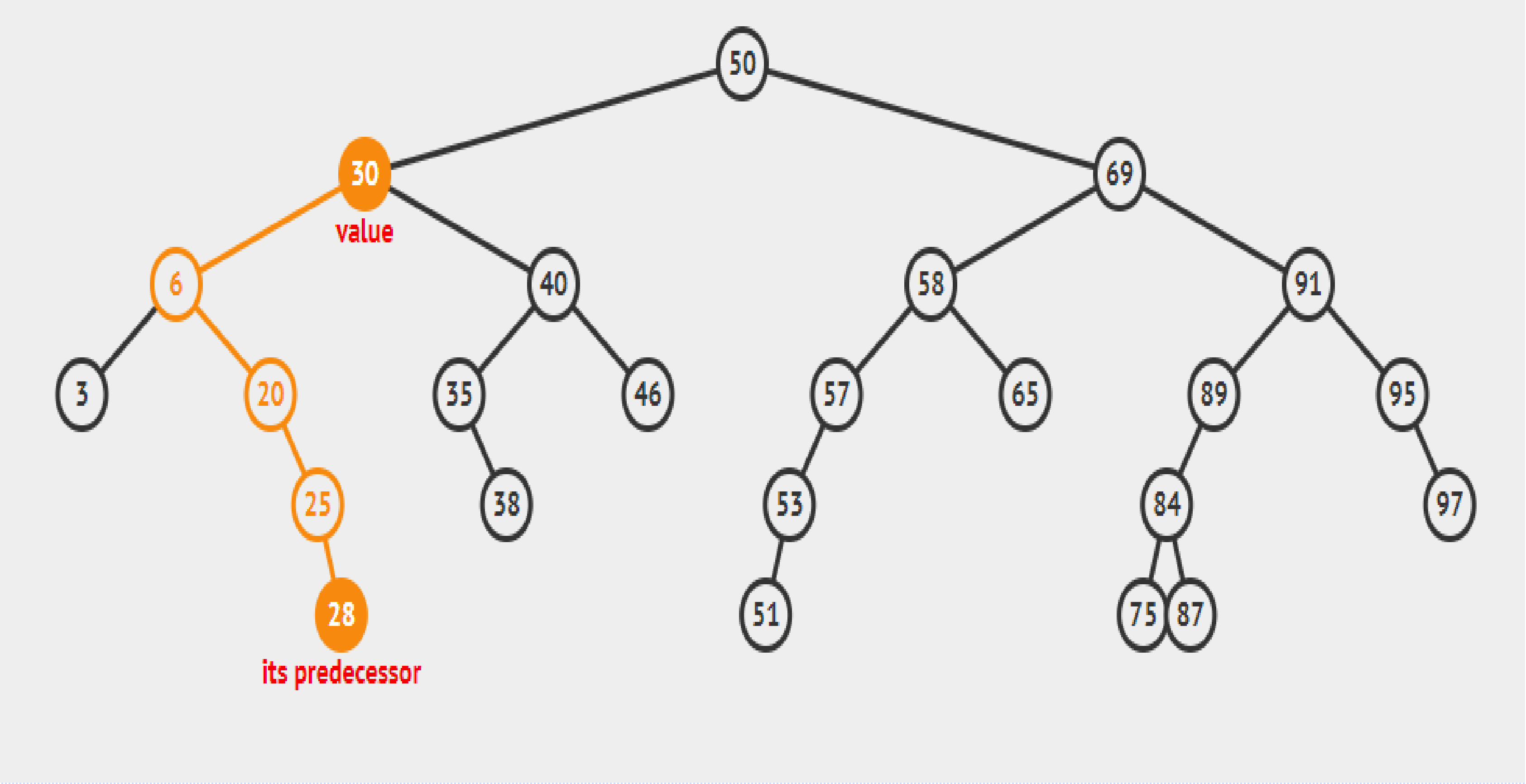
Predecessor of a node before(p)



before(91) = 89

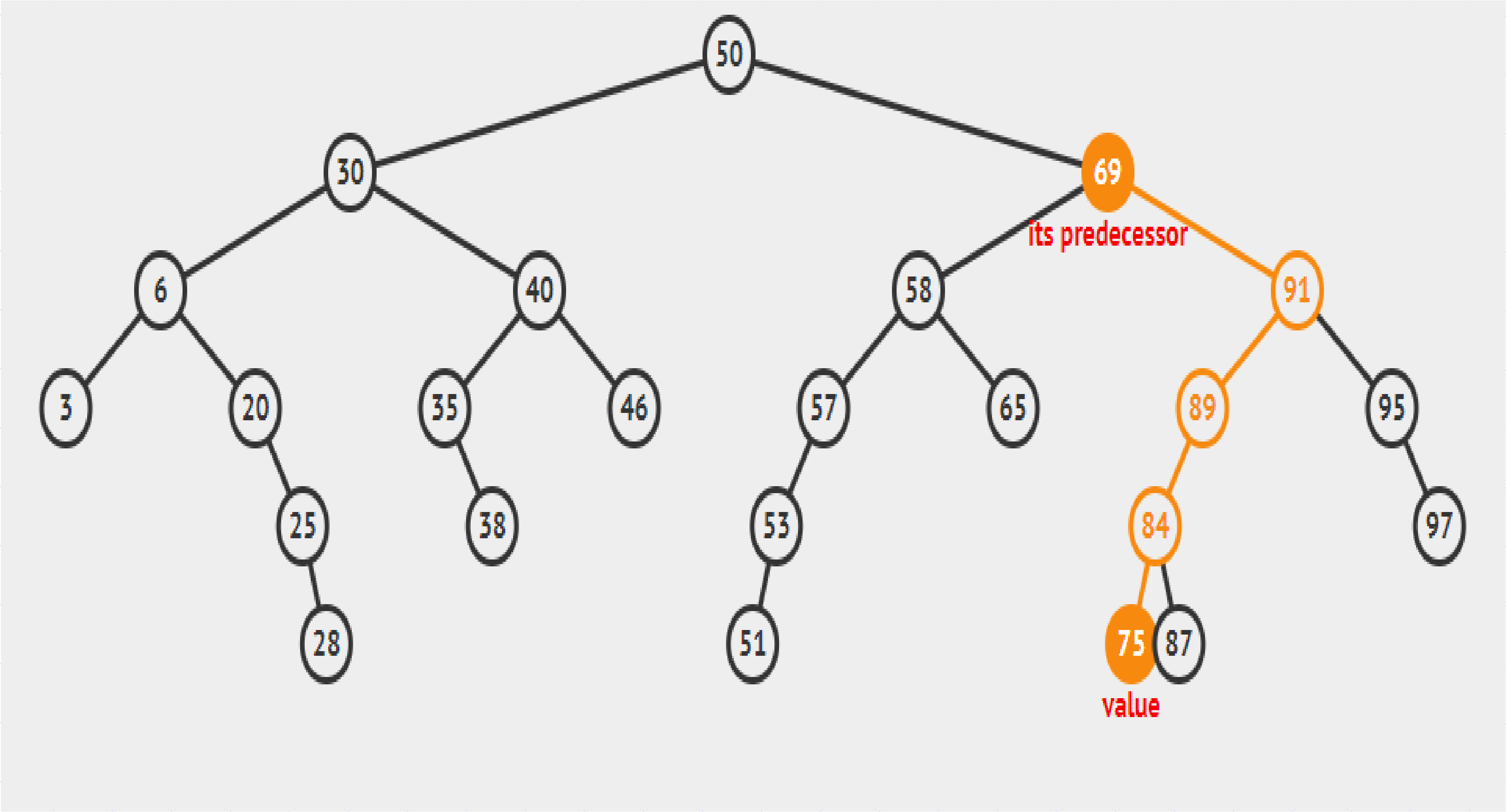

50
30
69
6
40
58
91
value
3
20
35
46
57
65
89
95
its predecessor
25
38
53
84
97
28
51
75
87

before(30) = 28



50
30
value
69
6
40
58
91
3
20
35
46
57
65
89
95
25
38
53
84
97
28
its predecessor
51
75
87

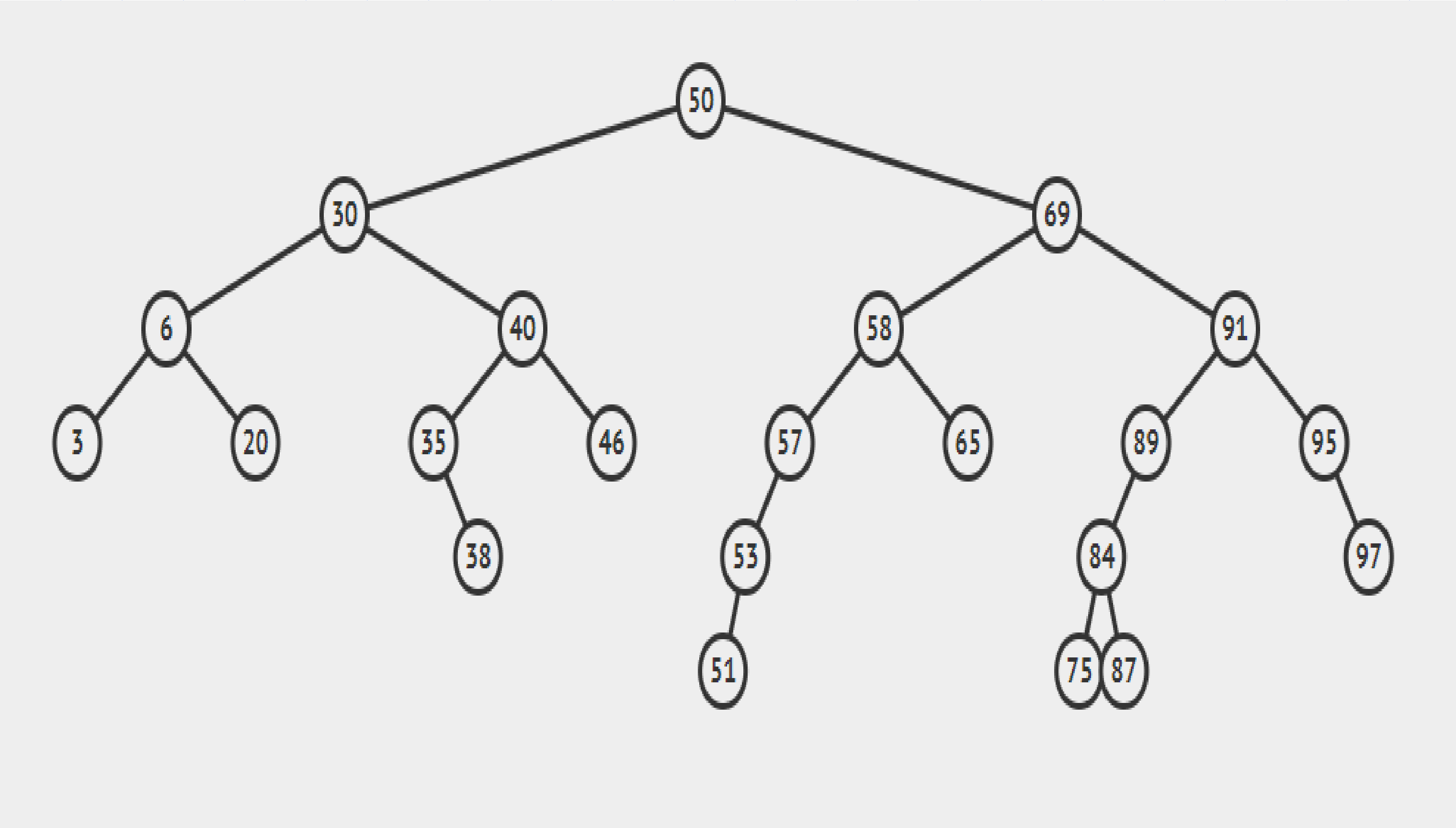
before(75) = 69


50
30
69
its predecessor
6
40
58
91
3
20
35
46
57
65
89
95
25
38
53
84
97
28
51
75
87
value

First and Last

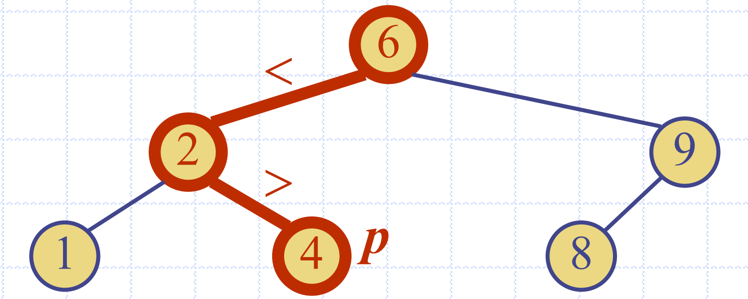
- first( ): Return the position containing the least key, or None if the tree is empty.
- last( ): Return the position containing the greatest key, or None if empty tree.

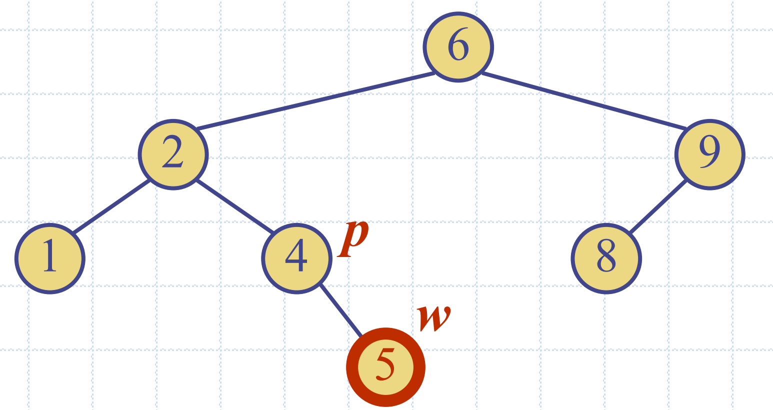
What is first() and last() position for this binary tree?



Insertion

- To perform operation put(k, v), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let p be the the node reached by the search
- We insert a new child node for p accordingly
- Example: insert 5

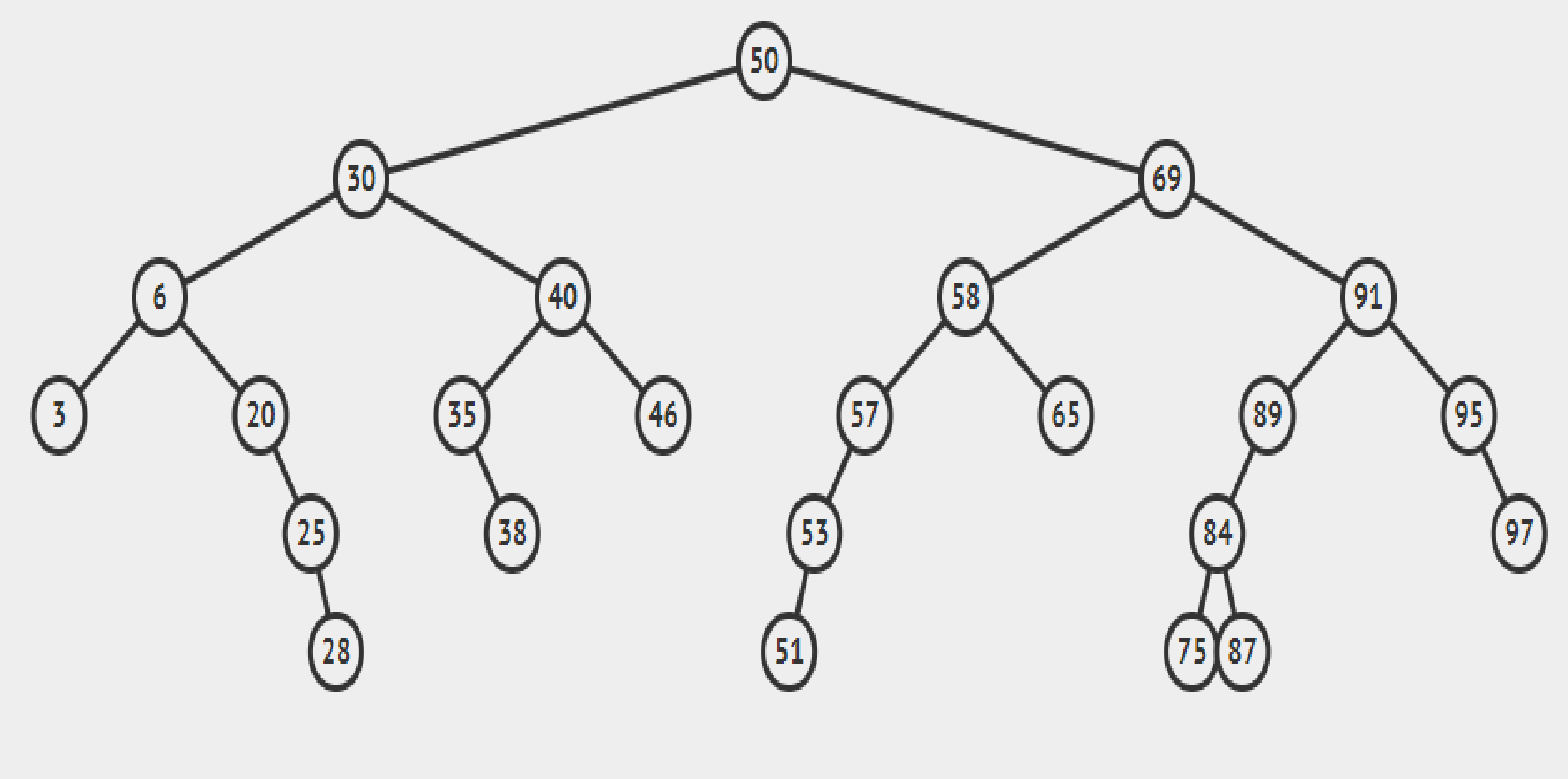




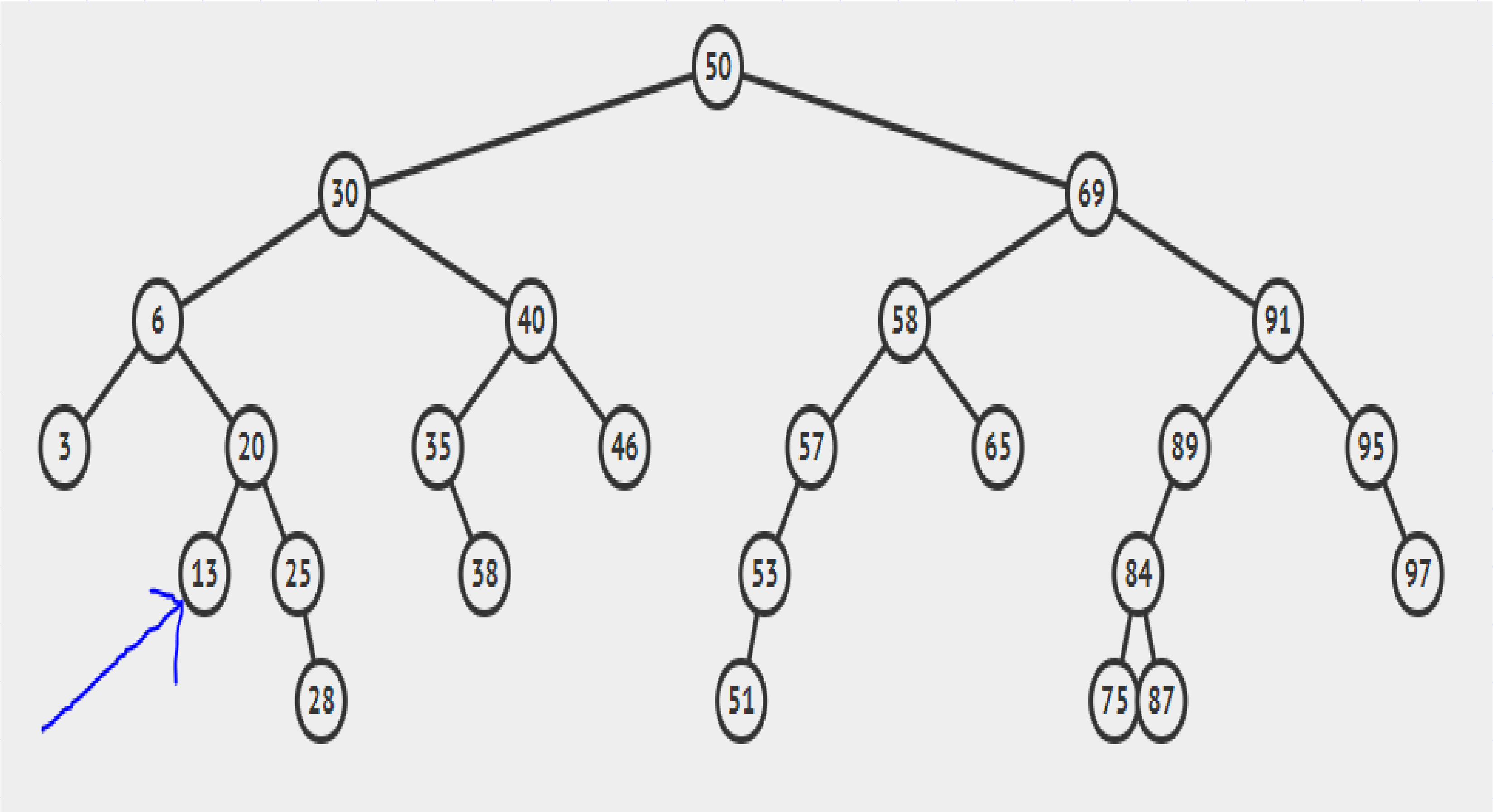
Insertion Pseudo-code

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

Insert 13 in this tree

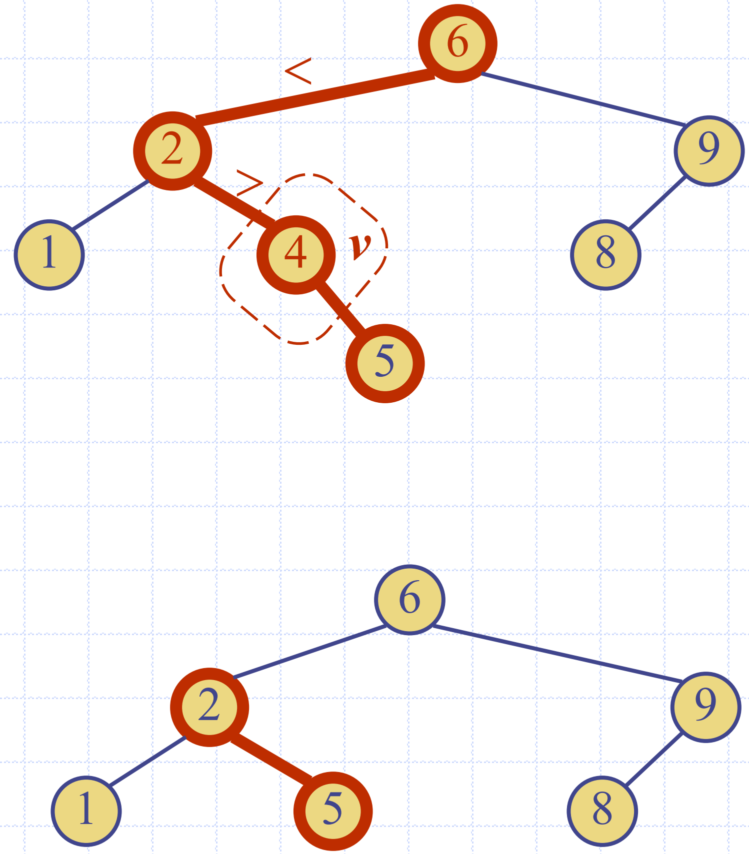


After inserting 13



Deletion

- To perform operation remove(k), we search for key k
- Assume key k is in the tree, and let v be the node storing k
- If node v has at most 1 child, connect v.parent with v.child
- Example: remove 4



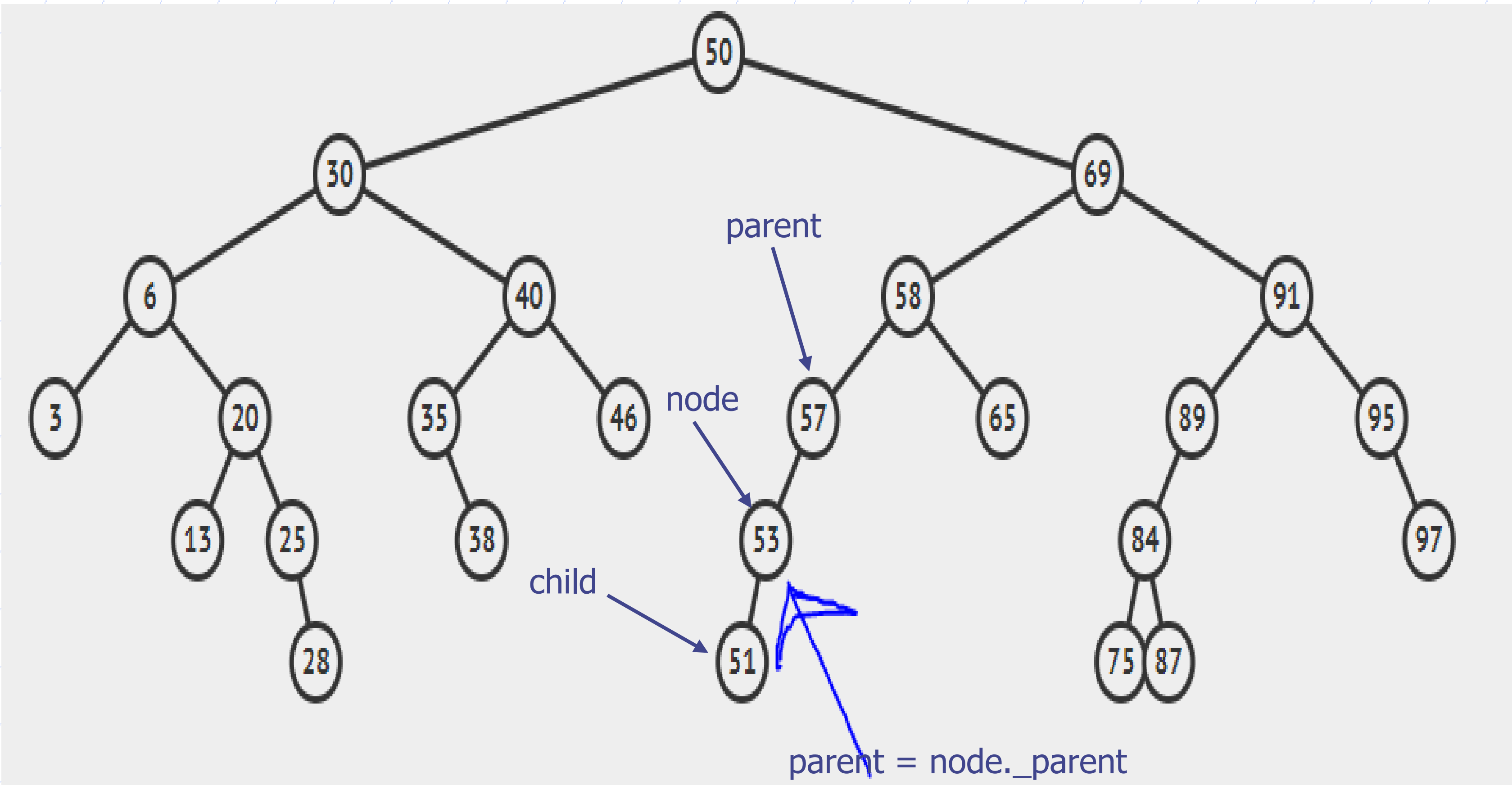
_delete method from LinkedBinaryTree Class

```
def _delete(self, p):
  """Delete the node at Position p, and replace it with its child, if any.

  Return the element that had been stored at Position p.
  Raise ValueError if Position p is invalid or p has two children.
  """
  node = self._validate(p)
  if self.num_children(p) == 2: raise ValueError('p has two children')
  child = node._left if node._left else node._right        # might be None
  if child is not None:
    child._parent = node._parent        # child's grandparent becomes parent
  if node is self._root:
    self._root = child                  # child becomes root
  else:
    parent = node._parent
    if node is parent._left:
      parent._left = child
    else:
      parent._right = child
  self._size -= 1
  node._parent = node                   # convention for deprecated node
  return node._element
```

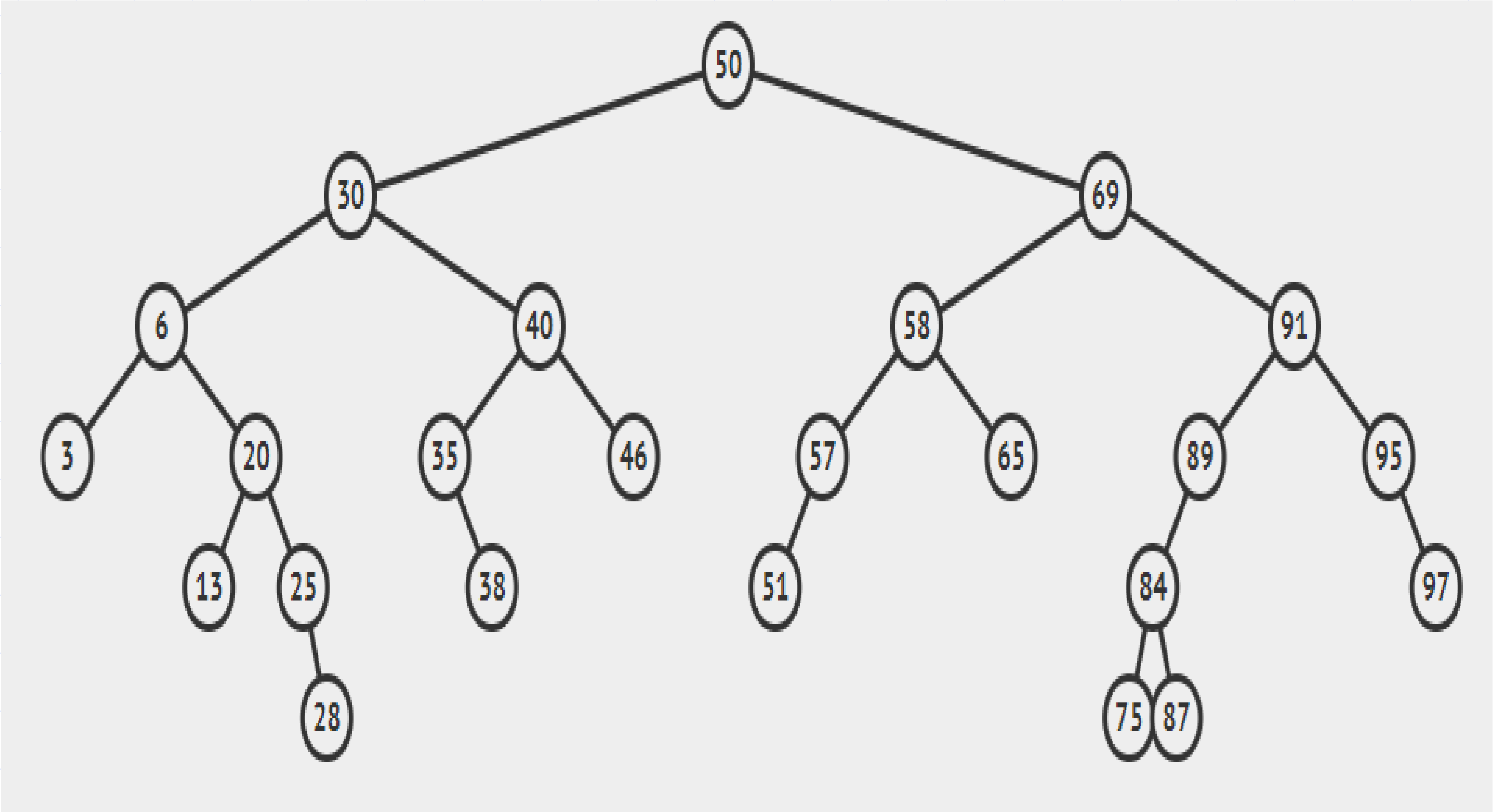
Check if node is a left or right child of its parent for correct connectivity

delete(53)
53 has just one child.



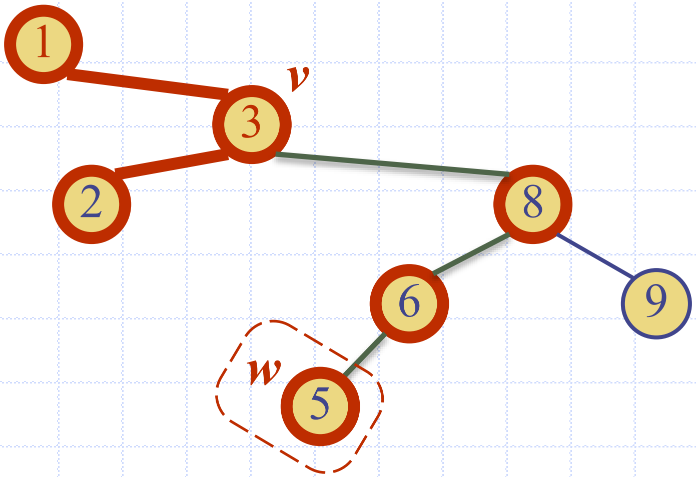
parent = node._parent
child = node._left if node._left else node._right
child._parent = parent
parent._left = child

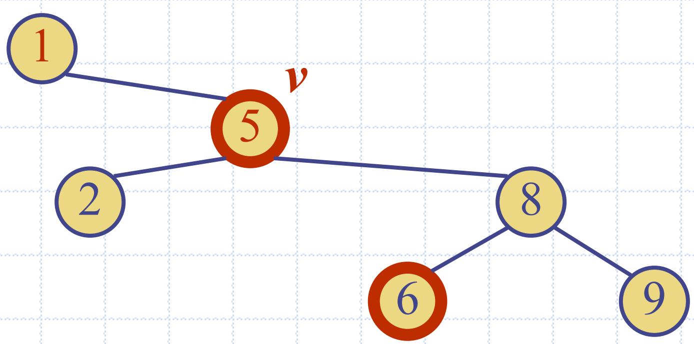
After deleting 53



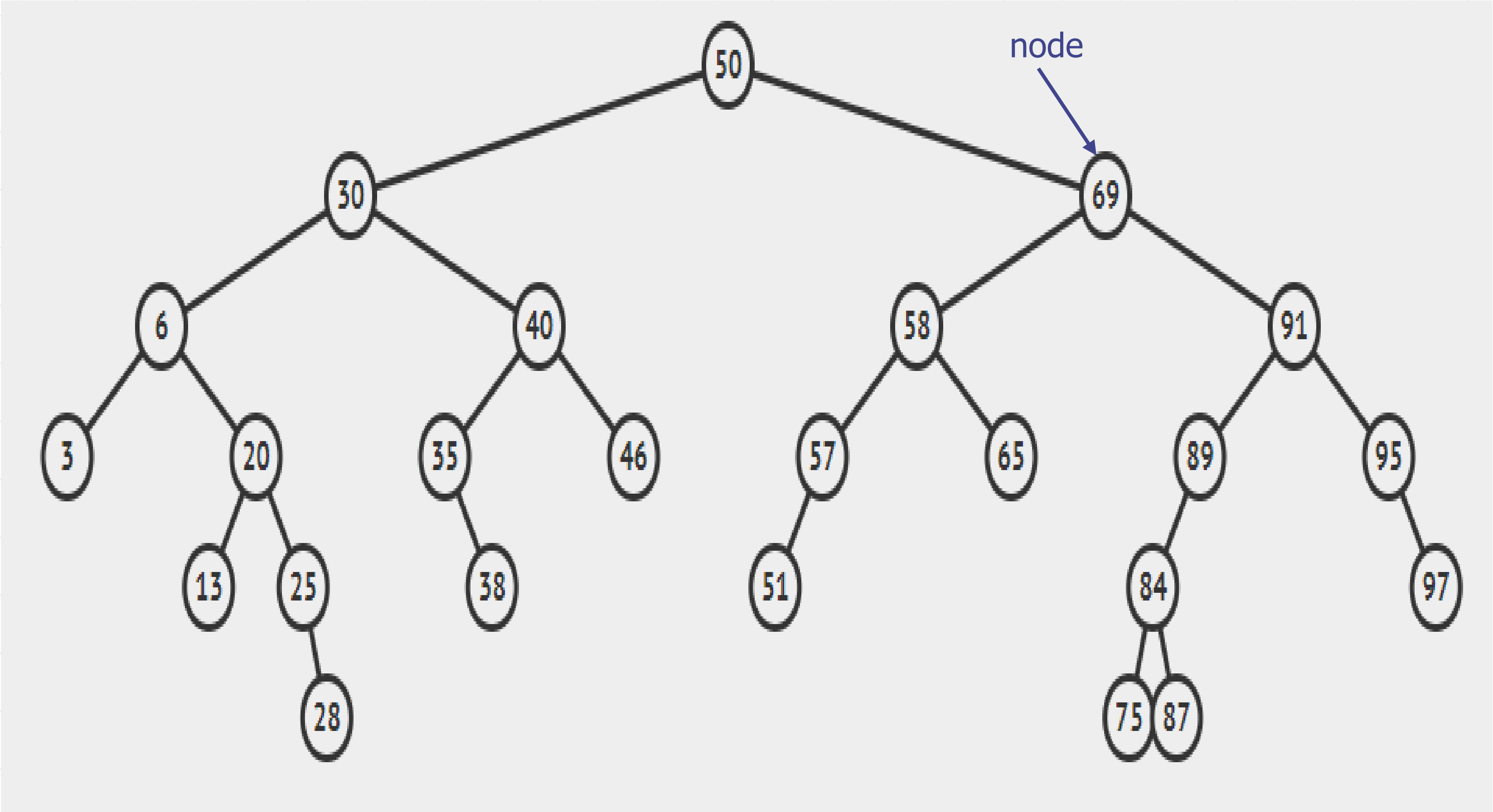
Deletion (cont.)

- We consider the case where the key k to be removed is stored at node v with 2 children
  - we find another node w that follows v in an inorder traversal (Find a substitution)
  - Leftmost node in subtree v.right
  - we copy $key(w)$ into node v
  - we remove node w
- Example: remove 3

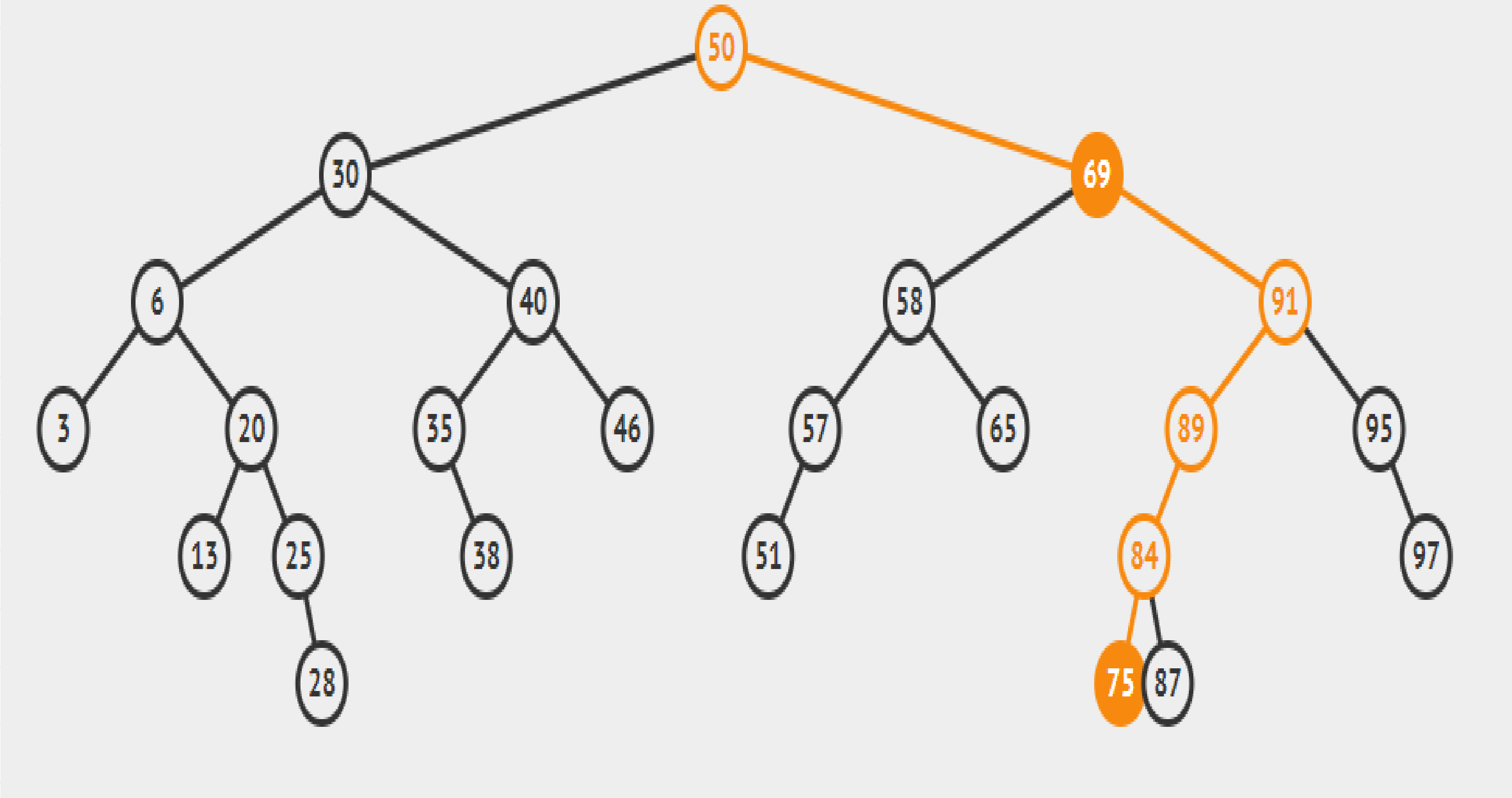




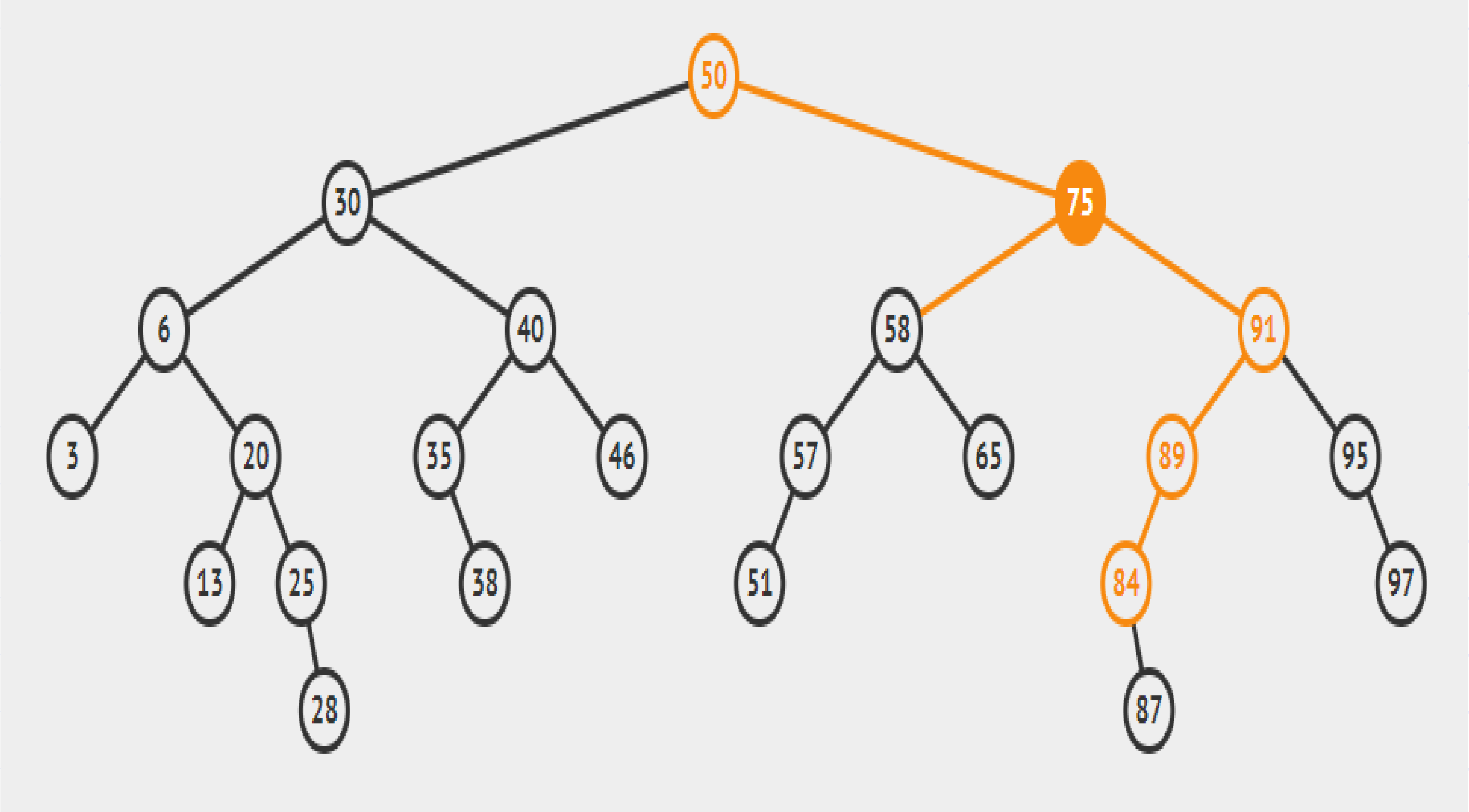
delete(69)
69 has two children.



Let's find successor of 69
after(69) = 75, replace 69 with 75 and then delete 69.

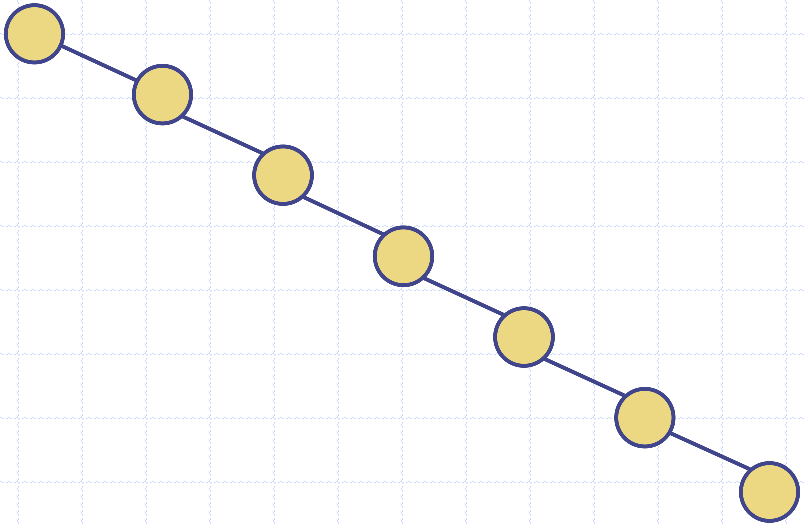


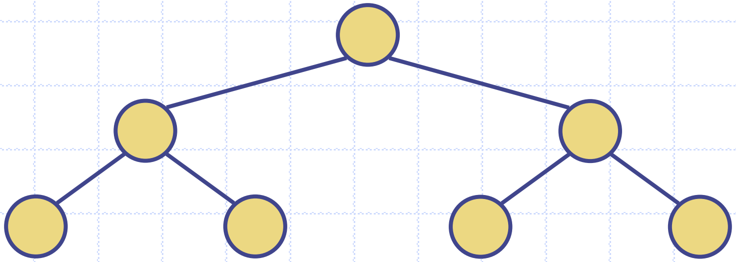
After removing 69



Performance

- Consider an ordered map with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - Search and update methods take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case





In-class exercise

- Download BinarySearchTree_without_position_student.py from Brightspace
- Implement several functions in class BinarySearchTree
  - Please refer to TODO in the code

Performance of BST

Operation	Running Time
k in T	$O(h)$
T[k], T[k] = v	$O(h)$
T.delete(p), del T[k]	$O(h)$
T.find_position(k)	$O(h)$
T.first(), T.last(), T.find_min(), T.find_max()	$O(h)$
T.before(p), T.after(p)	$O(h)$
T.find_lt(k), T.find_le(k), T.find_gt(k), T.find_ge(k)	$O(h)$
T.find_range(start, stop)	$O(s+h)$
iter(T), reversed(T)	$O(n)$

- Space usage is $O(n)$, where n is the number of items stored in the map.

Animation for BST

- https://visualgo.net/bn/bst