Lecture 7: AVL Trees

BST Review

- BST maintains a sorted order on a dynamic set

Data Structure	`insert(k)`	`delete(k)`	`find_min()`	`find(k)`	`find_next(k)`
Array	1^*	n	n	n	n
Sorted Array	n	n	1	$\log n$	$\log n$
Binary Heap	$\log n$	n	1	n	n
BST	h	h	h	h	h
Today!	$\log n$	$\log n$	1	$\log n$	$\log n$

- Worst case $O(h)$ per operation, where h is height, can be $h = O(n)$... :(
- Today:
  - How to make height $O(\log n)$?
  - Data structure augmentation
- Many ways to achieve 'logarithmic' height (red-black, B-trees, splay trees, skip lists, etc.)
- AVL trees was first (1962), and still perhaps one of the simplest

---

AVL Property

- **A**del'son-**V**el'skiĭ & **L**andis
- **AVL Property**: Sub-tree heights of a node's left and right children differ by at most one.
  - **node height**: #edges in longest path **down**, a.k.a., height of node's subtree. (Leaves have height 0.)
    - Don't confuse this with **node depth**: #edges in longest path **up**. (Root has depth 0.)
  - **node skew**: $\text{height}(\text{node.right}) - \text{height}(\text{node.left})$ (missing child contributes height -1)
- AVL property restated: skew $\in \{-1, 0, 1\}$
- **Claim**: Every node satisfies AVL Property $\implies$ height is $O(\log n)$
  - Want largest height for given number of nodes (i.e. fewest nodes for fixed height)

  - Let $S(h)$ be the minimum number of nodes in height h AVL tree
  - Worst case, when every node is skewed, with fewest nodes for height
  - $S(h) = S(h-1) + S(h-2) + 1 \geq 2S(h-2)$
  - $S'(h) = 2S'(h-2)$ has solution $S'(h) = 2^{h/2}$
  - $S(h) \geq S'(h) = 2^{h/2} \implies h \leq 2 \log S(h) \leq 2 \log n = O(\log n)$
  - Exercise: What is $S(h)$ exactly? (It's related to the Fibonacci numbers!)

Augmentation

- To speed up computation, store height (and skew) at each node
- How to maintain these?
- Sub-tree properties may change for any node effected by the change
- i.e. any ancestor of node inserted or removed
- Update sub-tree properties of ancestors going up the tree
- Other common augmentations computable from children, such as min, max, height, sum, count

```python
# Assumes node's descendants are updated
def update(node):
    left_height  = node.left.height  if node.left  else -1
    right_height = node.right.height if node.right else -1
    node.height = 1 + max(left_height, right_height)
    node.skew = right_height - left_height

# Assumes node and its ancestors are the
# only nodes that need updating
def maintain(node):
    update(node)
    if node.parent:
        maintain(node.parent)
```

- **Example:** Add keys $[6, 3, 9, 2, 3, 7, 4]$ and maintain height, skew

```
        ____________6_______
     ___3___           ___9
     2     3___        7
              4___
                (5)

h:   0  3  2  1  0  4  0  1
s:   0  2  2  1  0 -2  0 -1
```

- Insert key 5. Uh oh! Now some nodes do not satisfy AVL Property!
- How to fix skew while maintaining BST Property?

Rotations

- **Invariant**: tree is AVL before and after dynamic operation
- Consider lowest node X violating AVL after insert or delete
- Skew magnitude is two: without loss of generality assume positive
- Either heavier sub-tree has non-opposite (same or equal) or opposite skew

```
          Case 1   _____X_____            _____X_____    Case 2
                   A     ____Y____        A     ____Y____
                  / \    B       C       / \    B       C
        h ......./___\../ \...../ \...../___\../ \...../ \...... h
        h + 1 ........./___\.../   \........../   \.../___\..... h + 1
        h + 2 ......../_____\./_____\......../_____\............ h + 2
```

- If heavier sub-tree same or equal, rotation!
- A rotation rearranges attached sub-trees to maintain the BST property (confirm)

```
                ____y____    right_rotate(y)     ____x____
             ___x___    c           =>           a    ___y___
             a     b   / \                      / \   b     c
            / \   / \ /___\         <=         /___\ / \   / \
           /___\ /___\        left_rotate(x)        /___\ /___\
```

- For case 1, rotate left at X. A and X move down, B stays same, C and Y moves up
- Restores balance!

```
  Case 1   _____X_____        left_rotate(X)      ______Y_____
          A     ____Y____          =>        ____X___       C
         / \    B       C                    A      B      / \
h ....../___\../ \...../ \................. / \..../ \..../   \..... h
h + 1 ......../___\.../   \................/___\../___\../_____\.... h + 1
h + 2 ......./_____\./_____\...................../_____\............ h + 2
```

- For case 2, rotate right at Y. C moves down, B moves up or stays same
- Now looks like case 1! A left rotation at X restores balance!

```
  Case 1  _____X________     right_rotate(Y)   _____X_____
          A        ____Y_____      =>          A     ____B_____
         / \    ___B___     C                 / \    D     ___Y___
h ....../___\...D.....E..../ \.............../___\../_\....E...  C  .. h
h + 1 ........./_\.../_\../___\................../___\../_\.../ \... h + 1
h + 2 ......../___\./___\................................/___\./___\.. h + 2
```

- Call update on X and Y after rotation as sub-trees have changed
- Continue walking up the tree to repeatedly find and fix lowest violation in $O(\log n)$ time
- Note: might need to rebalance multiple times on the way up!

```
# Assumes all nodes other than node and its ancestors
# are updated and AVL
def maintain(node):
    update(node)
    balance(node)
    if node.parent:
        maintain(node.parent)

# Assumes node and descendants are updated,
# and descendants satisfy AVL property
def balance(node):
    if node.skew == 2:
        if node.right.skew == -1:
            right_rotate(node.right)
        left_rotate(node)
    if node.skew == -2:
        if node.left.skew == 1:
            left_rotate(node.left)
        right_rotate(node)
```

- **Example:** Balance insertion of key 5

```
                  ________6____          ________6____
                __3__       __9        __3____     __9
                2  (3)_     7      =>  2   __4__   7
                     (4)_                  3   5
                       (5)
```

- **Example:** Balance deletion of key 9

```
              _________6__                     ____(6)_               _____4_____
            _(3)____      7    =>         ____(4)_   (7)    =>     __3__     __6__
           (2)    _(4)_                 _(3)_   (5)                2    3    5    7
                 (3)  (5)               2    3
```

AVL Sort

```
def AVL_sort(A):
    T = AVL()
    for a in A:                    # n times
        T.insert(a)                # O(log n)
    return in_order_traversal(T)   # O(n)
```

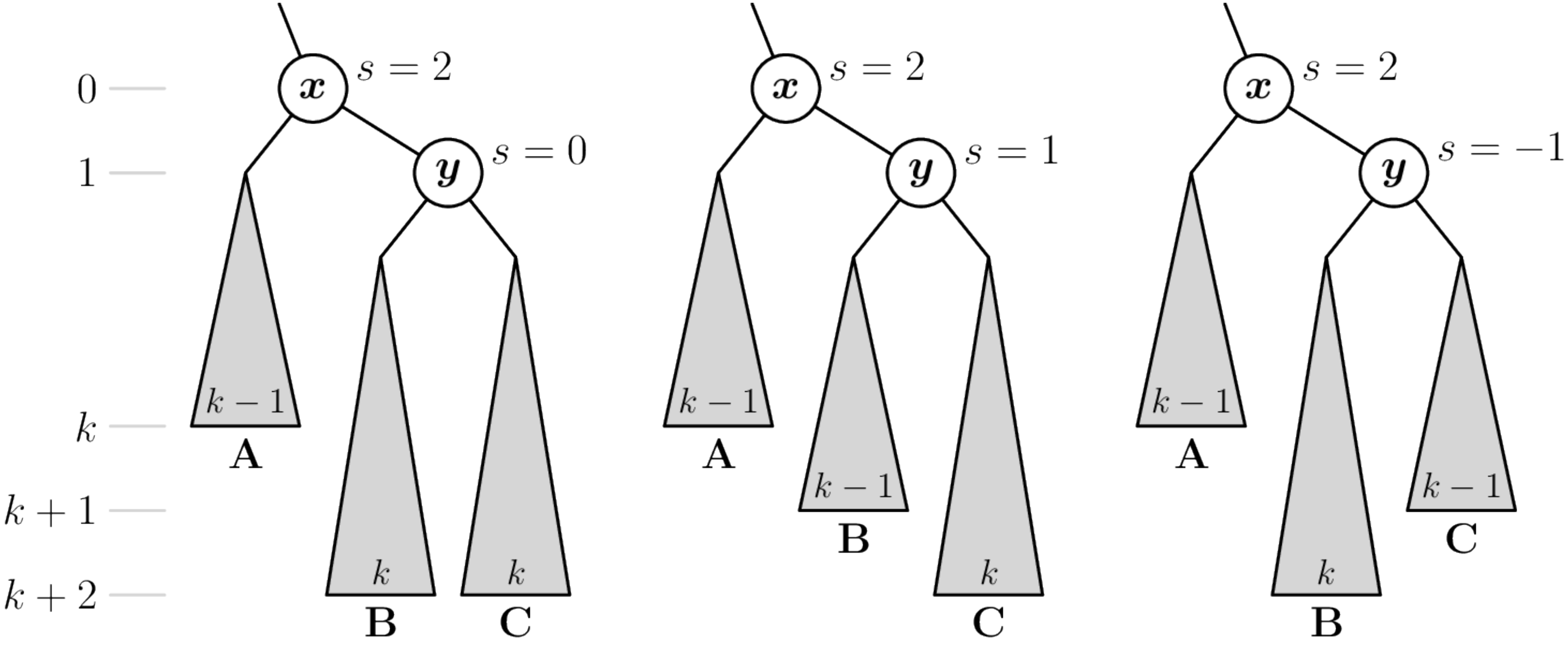
- Maintaining that height is $O(\log n)$ ensures insertion is also $O(\log n)$
- AVL sort runs in $O(n \log n)$ time.
- Can we do better? Next time!

Data Structure	Static Set		Dynamic Set		D.O.S.	Ordered Set			Space $\sim$
	find-key(k)	iter()	insert(x)	delete-key(k)	delete-min/max()	find-next/prev(k)	find-min/max()	order-iter()	
static array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \lg n)$	$1 \cdot n$
linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \lg n)$	$3 \cdot n$
dynam. array	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$ a.	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \lg n)$	$[n, 4n]$
sorted array	$\Theta(\lg n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(n)$	$1 \cdot n$
binary heap	$\Theta(n)$	$\Theta(n)$	$\Theta(\lg n)$ a.	$\Theta(n)$	one in $\Theta(\lg n)$	$\Theta(n)$	one in $\Theta(1)$	$\Theta(n \lg n)$	$1 \cdot n$
AVL tree	$\Theta(\lg n)$	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(n)$	$5 \cdot n$

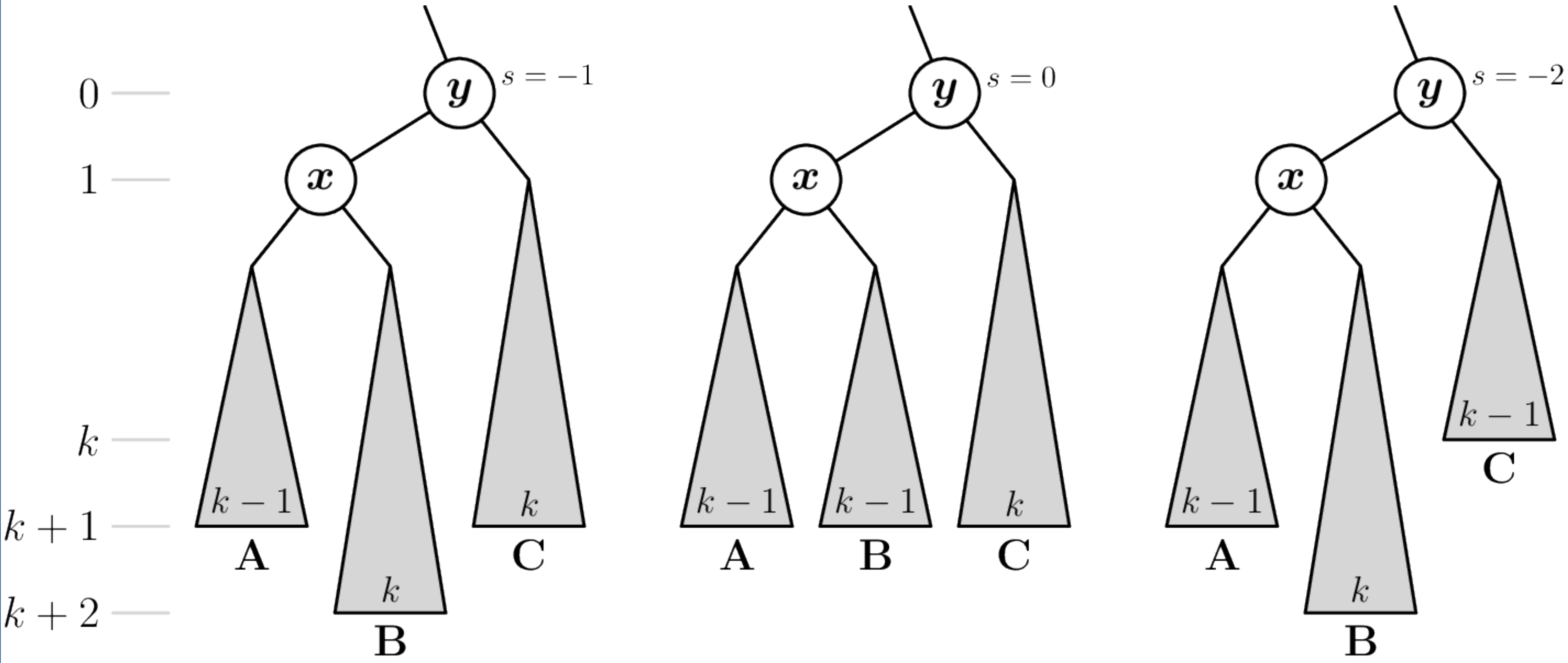
Data Structure	Static Set		Dynamic Set		D.O.S.	Ordered Set			Space
	find-key(k)	iter()	insert(x)	delete-key(k)	delete-min/max()	find-next/prev(k)	find-min/max()	order-iter()	$\sim$
static array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \lg n)$	$1 \cdot n$
linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \lg n)$	$3 \cdot n$
dynam. array	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$ a.	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \lg n)$	$[n, 4n]$
sorted array	$\Theta(\lg n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(n)$	$1 \cdot n$
binary heap	$\Theta(n)$	$\Theta(n)$	$\Theta(\lg n)$ a.	$\Theta(n)$	one in $\Theta(\lg n)$	$\Theta(n)$	one in $\Theta(1)$	$\Theta(n \lg n)$	$1 \cdot n$
AVL tree	$\Theta(\lg n)$	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(n)$	$5 \cdot n$

```python
# Assumes node's children are updated
def update(node):
   lh = node.left.height if node.left else -1
   rh = node.left.height if node.left else -1
   node.height = 1 + max(lh, rh)
   node.skew = rh - lh

# Assumes only node and ancestors need updating
def maintain(node):
   update(node)
   if node.parent:
      maintain(node.parent)
```

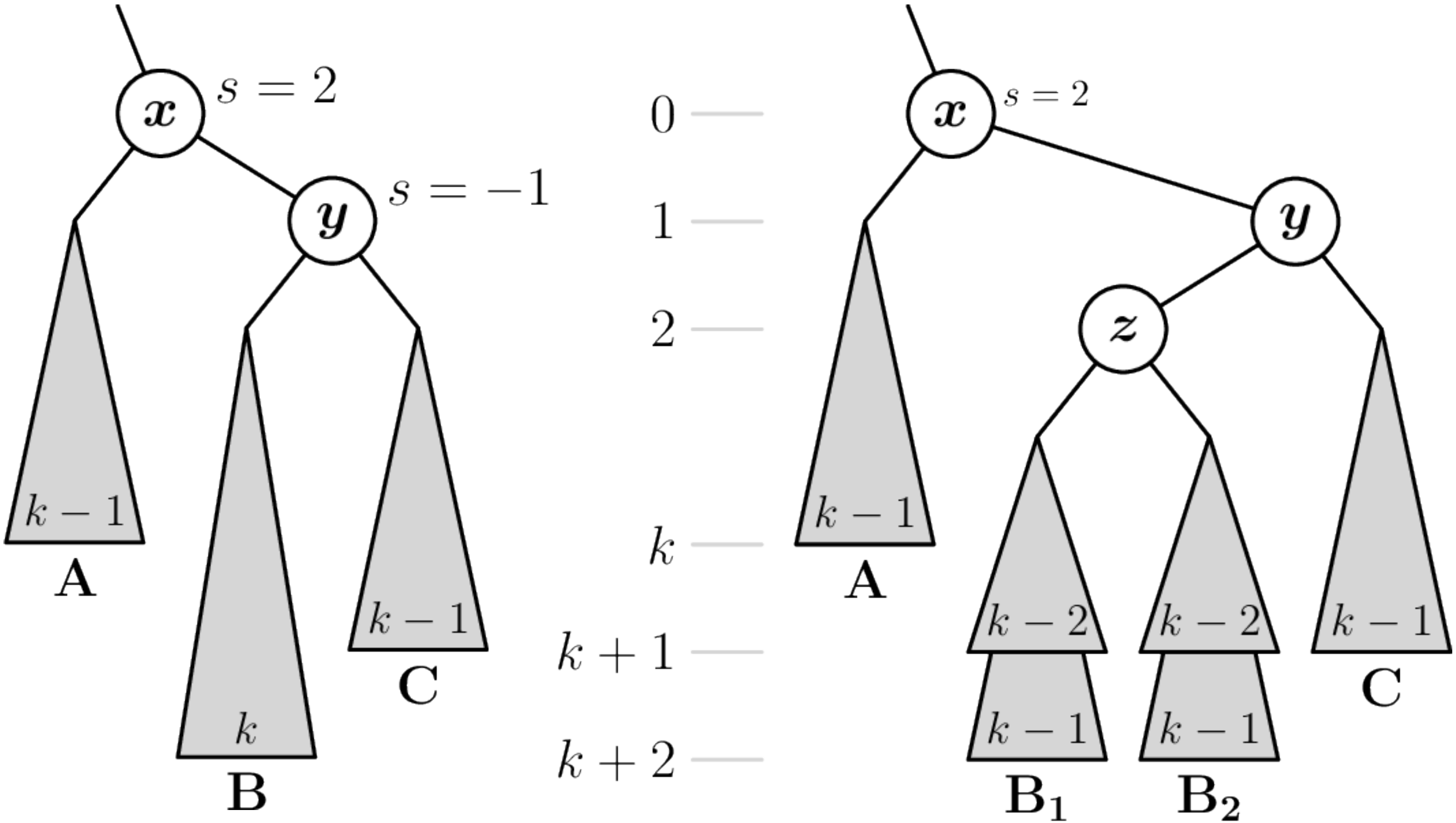
All 3 ways for x to have skew 2



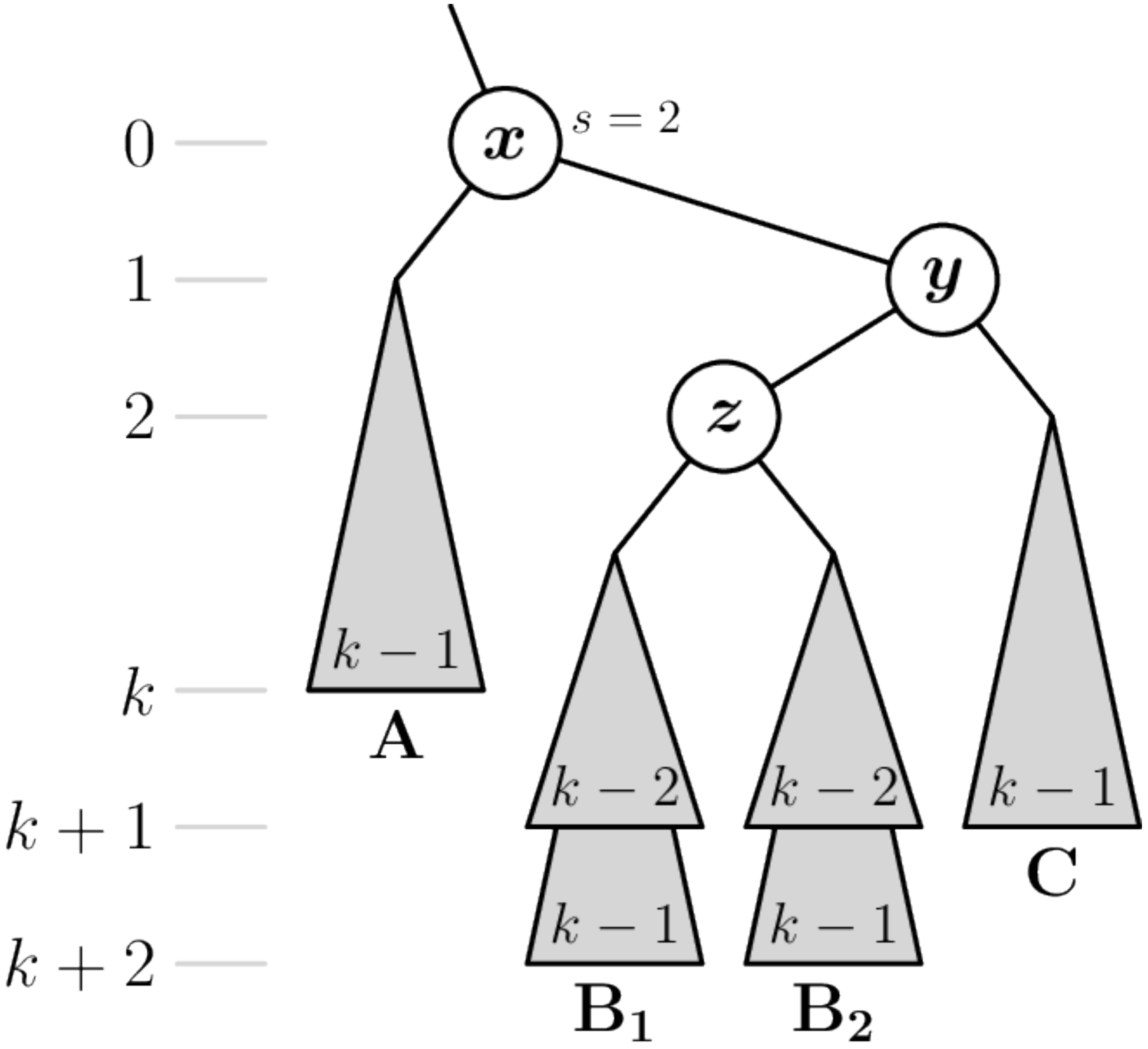
After rotating x left



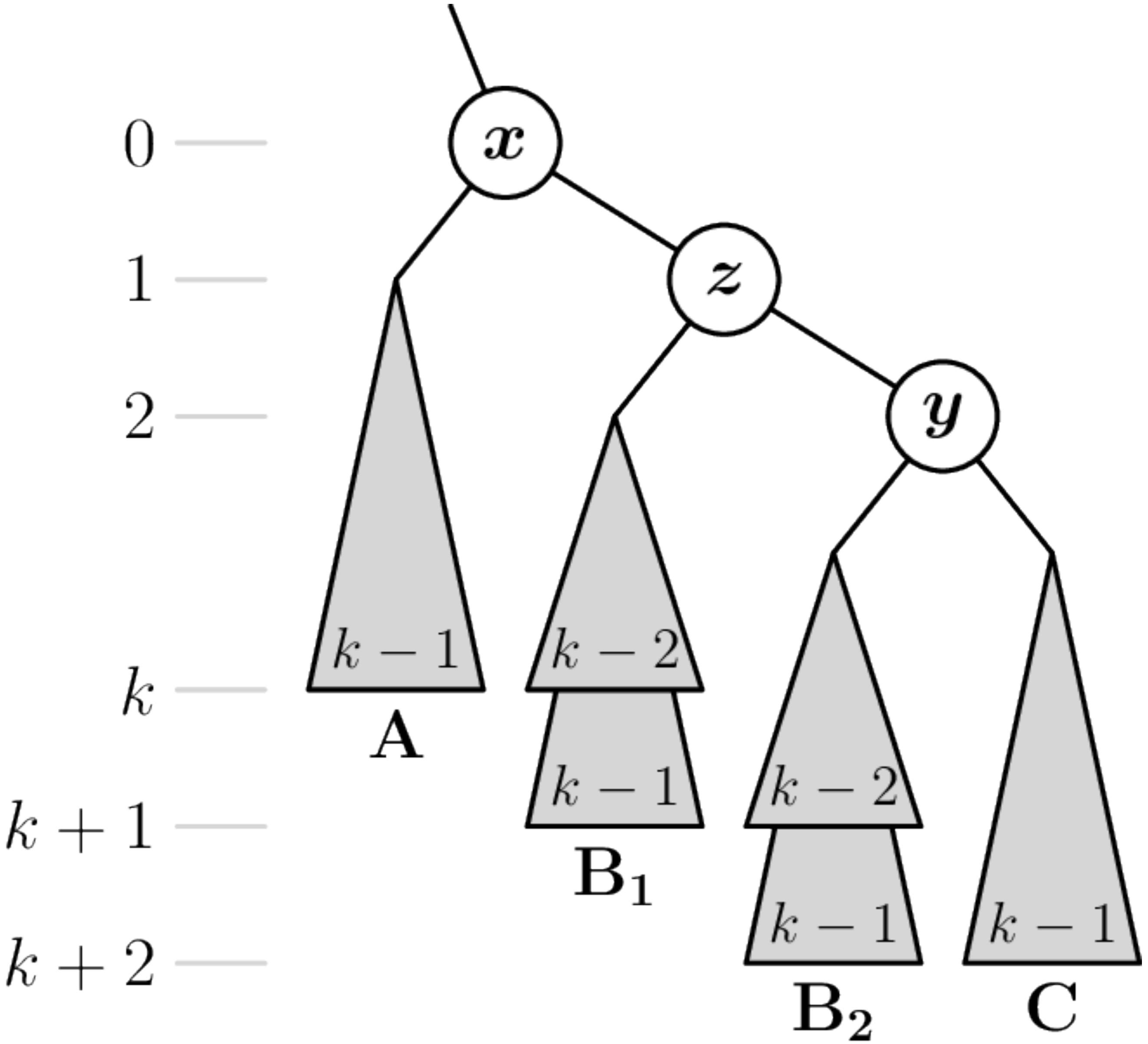
The harder case, dissected



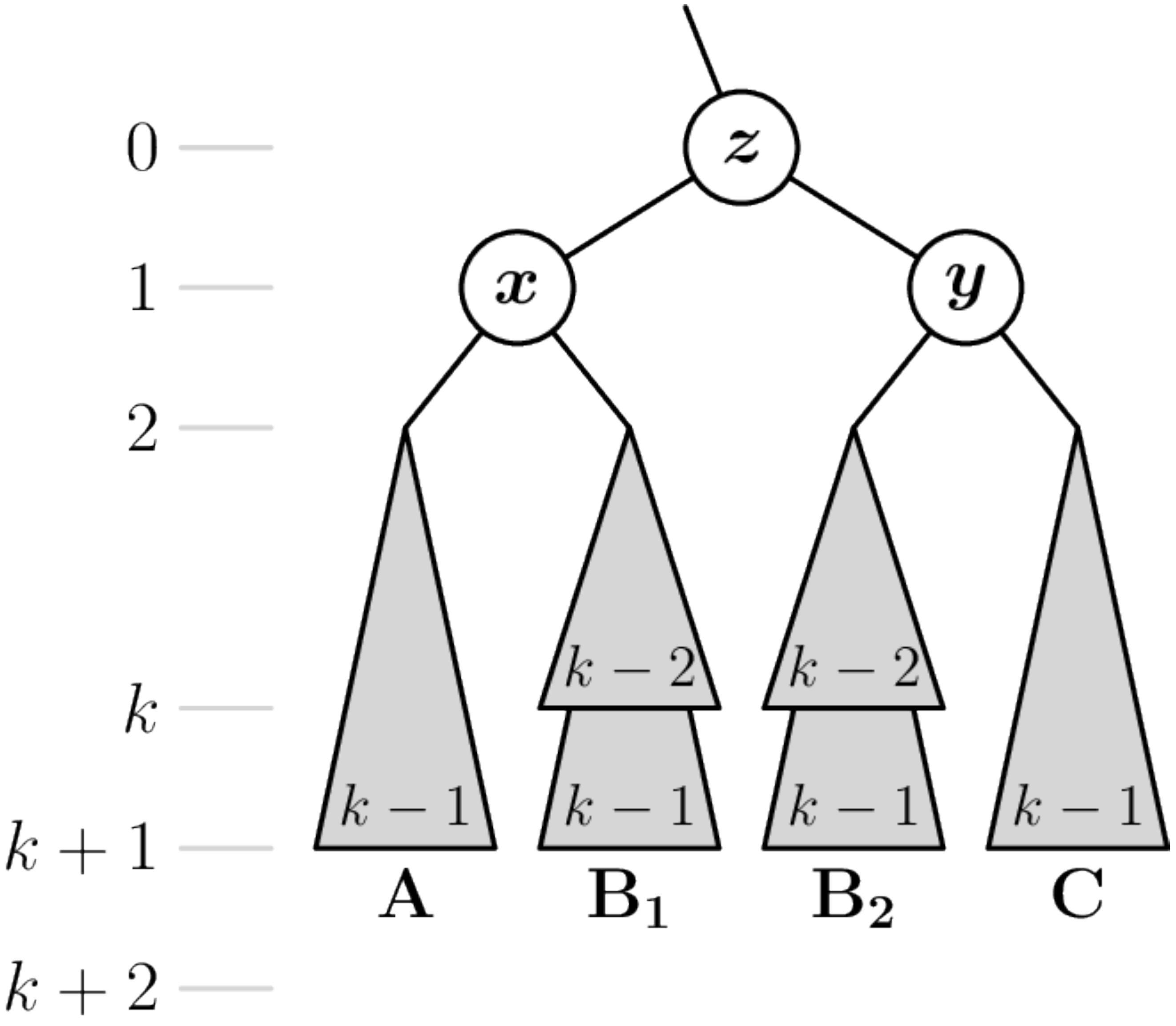
The harder case



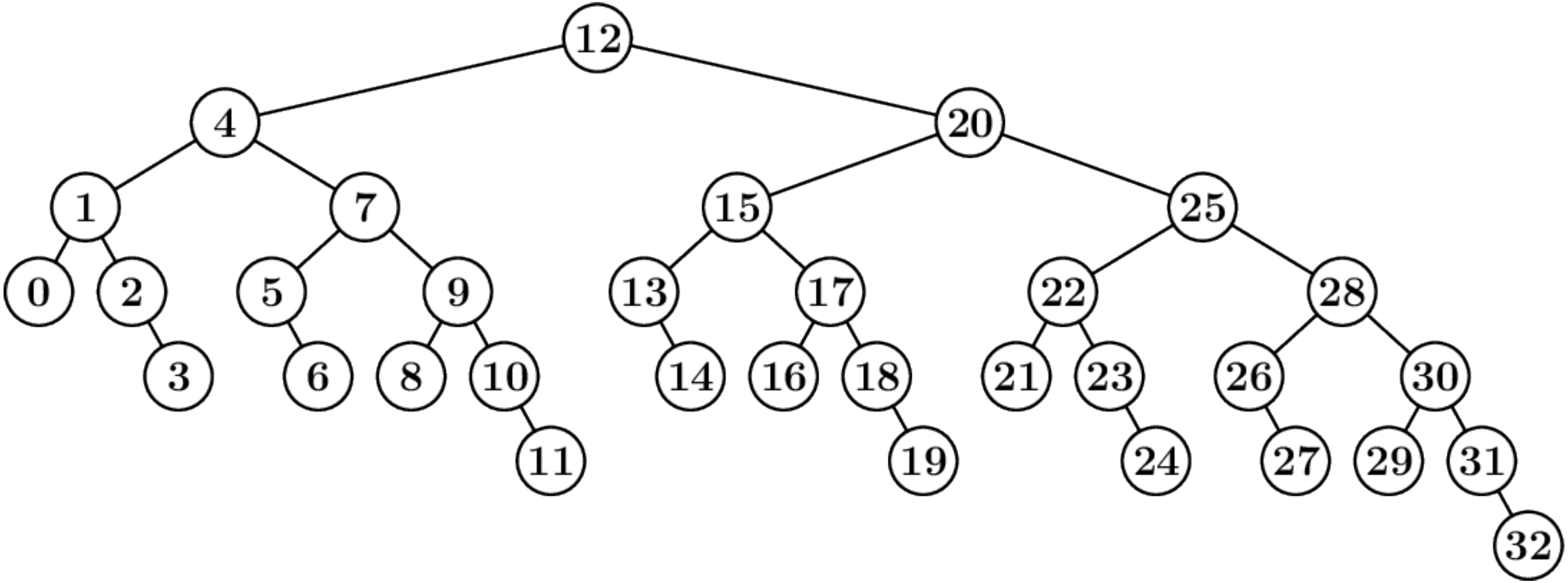
After rotating y right

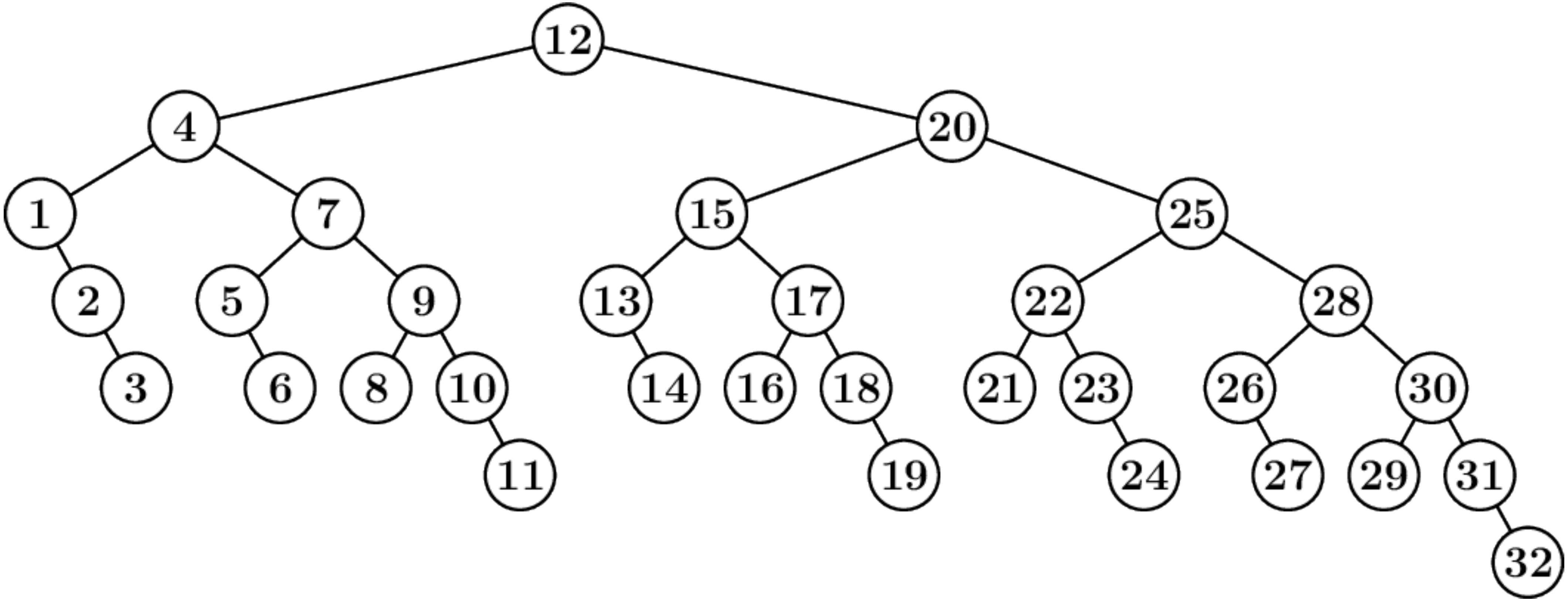


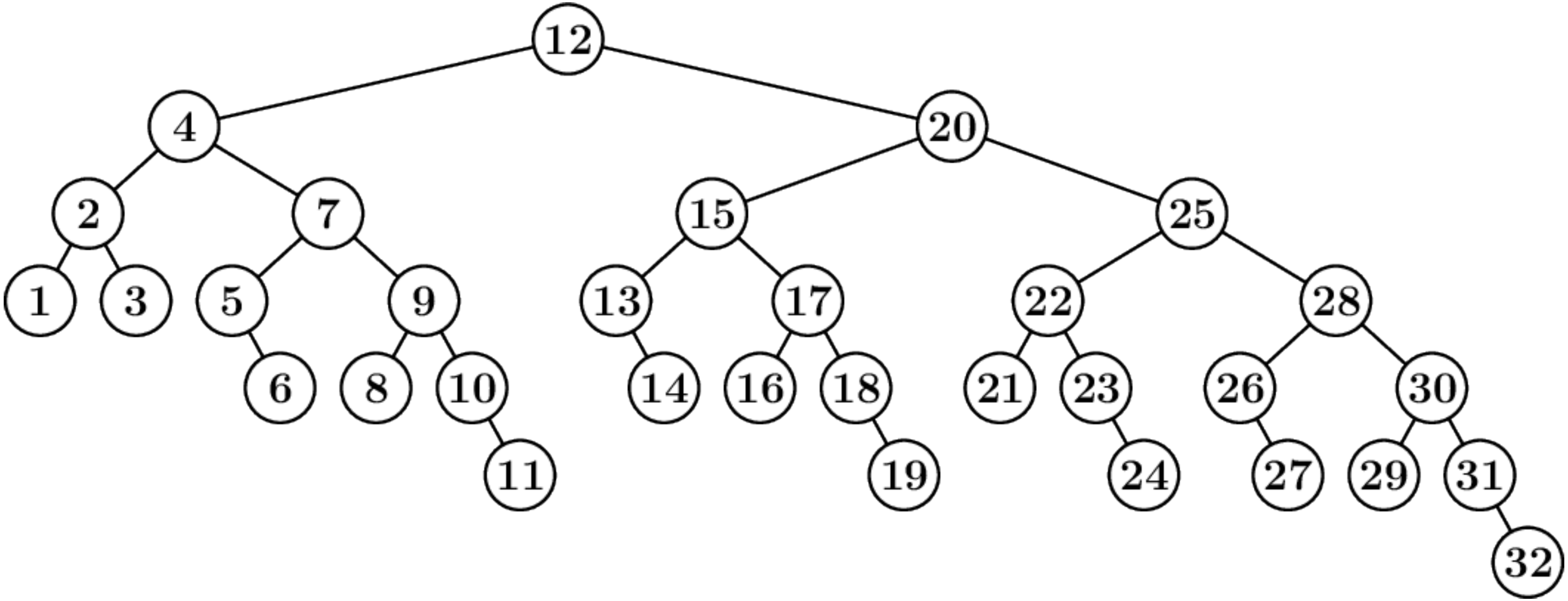
After rotating x left

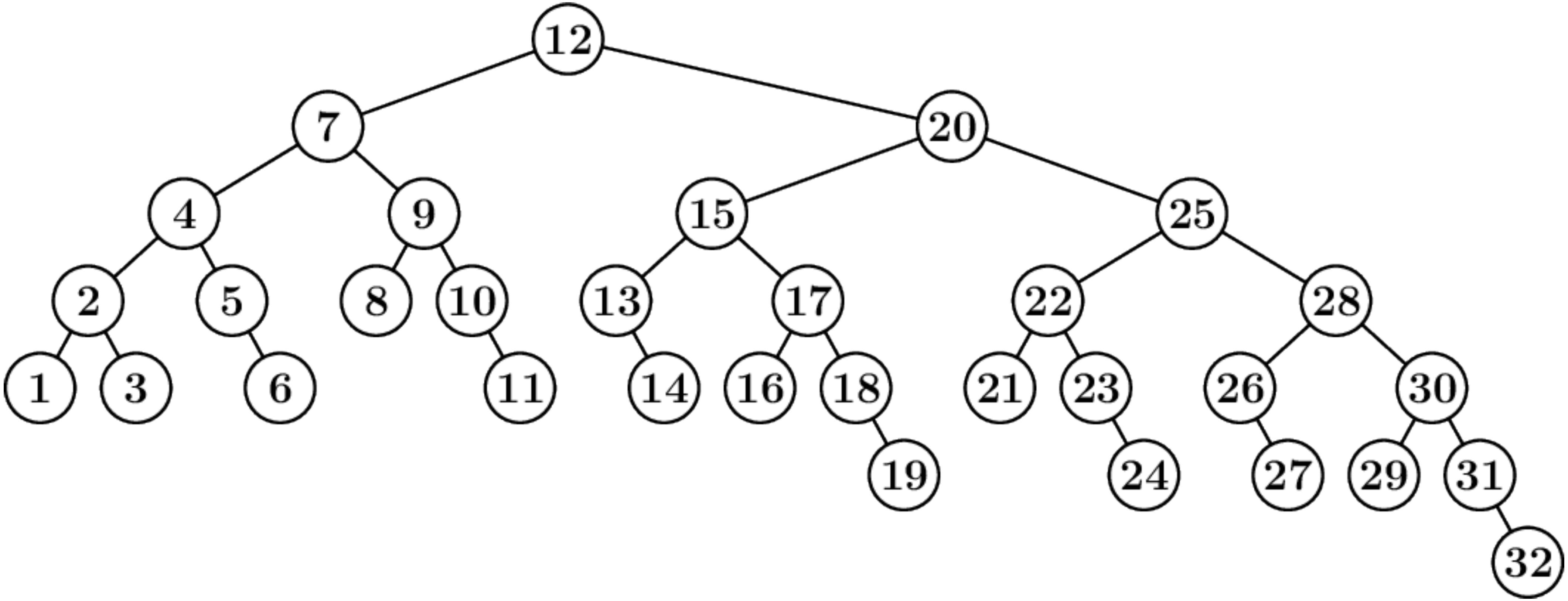


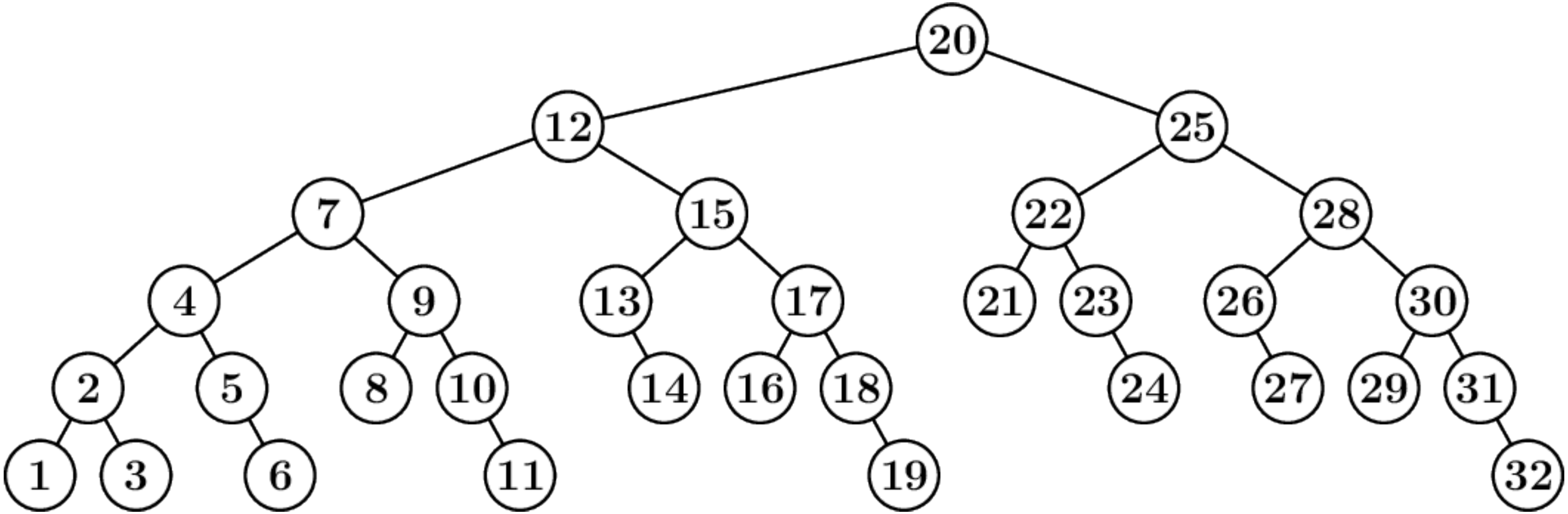
```
def balance(node):          # node and descendants
  if node.skew == 2:        # are updated, and node's
    if node.right.skew == -1:    # descendants
      rotate_right(node.right)   # are AVL
    rotate_left(node)
  elif node.skew == -2:
    <symmetrical>

def maintain(node):       # Assumes node's descendants
  update(node)            # are updated and AVL
  balance(node)
  if node.parent:
    maintain(node.parent)
```


12
4
20
1
7
15
25
0
2
5
9
13
17
22
28
3
6
8
10
14
16
18
21
23
26
30
11
19
24
27
29
31
32


12
4
20
1
7
15
25
2
5
9
13
17
22
28
3
6
8
10
14
16
18
21
23
26
30
11
19
24
27
29
31
32


12
4
20
2
7
15
25
1
3
5
9
13
17
22
28
6
8
10
14
16
18
21
23
26
30
11
19
24
27
29
31
32


12
7
20
4
9
15
25
2
5
8
10
13
17
22
28
1
3
6
11
14
16
18
21
23
26
30
19
24
27
29
31
32


20
12
25
7
15
22
28
4
9
13
17
21
23
26
30
2
5
8
10
14
16
18
24
27
29
31
1
3
6
11
19
32