Introduction to Algorithms

Meng-Tsung Tsai

11/14/2019

Announcements

Programming Quiz 1 will be held in EC 315/316/324 this Saturday (Nov 16) 13:30 - 17:30.

Details can be found in Slides 14.

Don't be late. You can only copy files from your USB flash drive to the computer assigned to you before 13:40. After which, we will disable the service of USB ports on "all" computers.

Reference

Quake Heaps: A Simple Alternative to Fibonacci Heaps,

Timothy M. Chan (2013).

Quake Heaps

What is a Quake heap?

A data structure better than binary heaps in that the supported operations run "faster".

Operations	Binary Heap (worst-case)	Quake Heap (amortized)
Make-Heap	O(1)	O(1)
Insert	O(log n)	O(1)
Extract-Min	O(log n)	O(log n)
Decrease-Key	O(log n)	O(1)

What is a Quake heap?

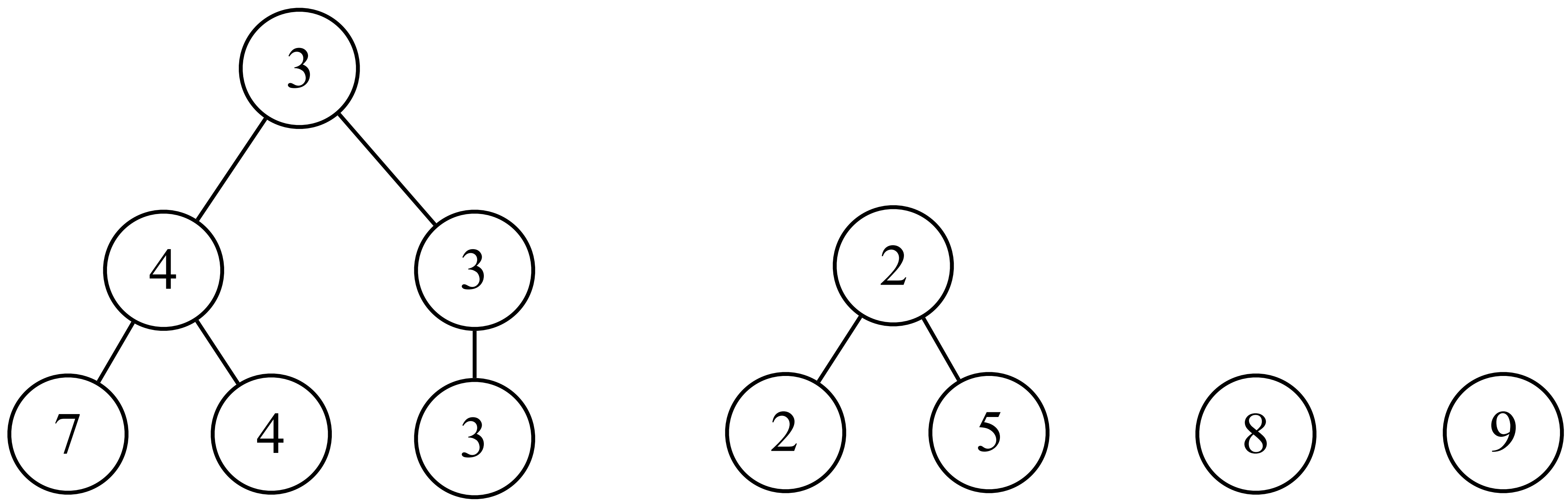
Quake heaps is a simplification of Fibonacci heaps.

You may pick up Fibonacci heaps from I2A (pp. 505 -- 530).

Basics

A Quake heap is a collection of tournament trees, where the value at parent node is the minimum of values at child nodes.

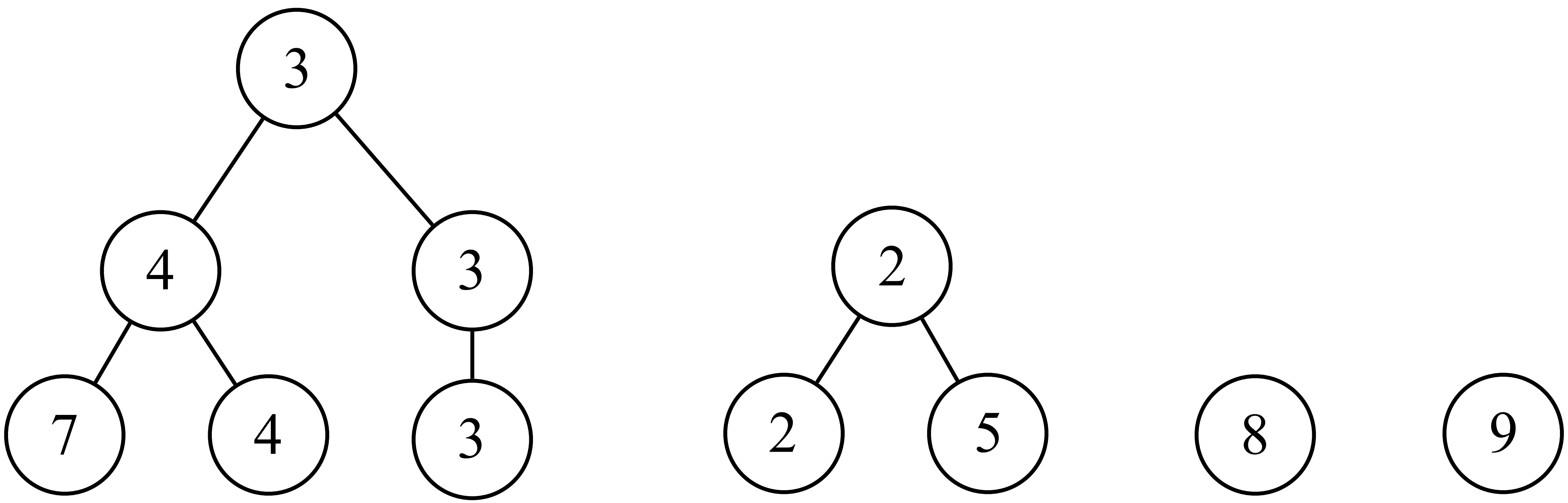
Every key in the heap has a copy at exactly one leaf node.



Basics

Each internal node has degree 1 or 2, where degree is defined as the number of children.

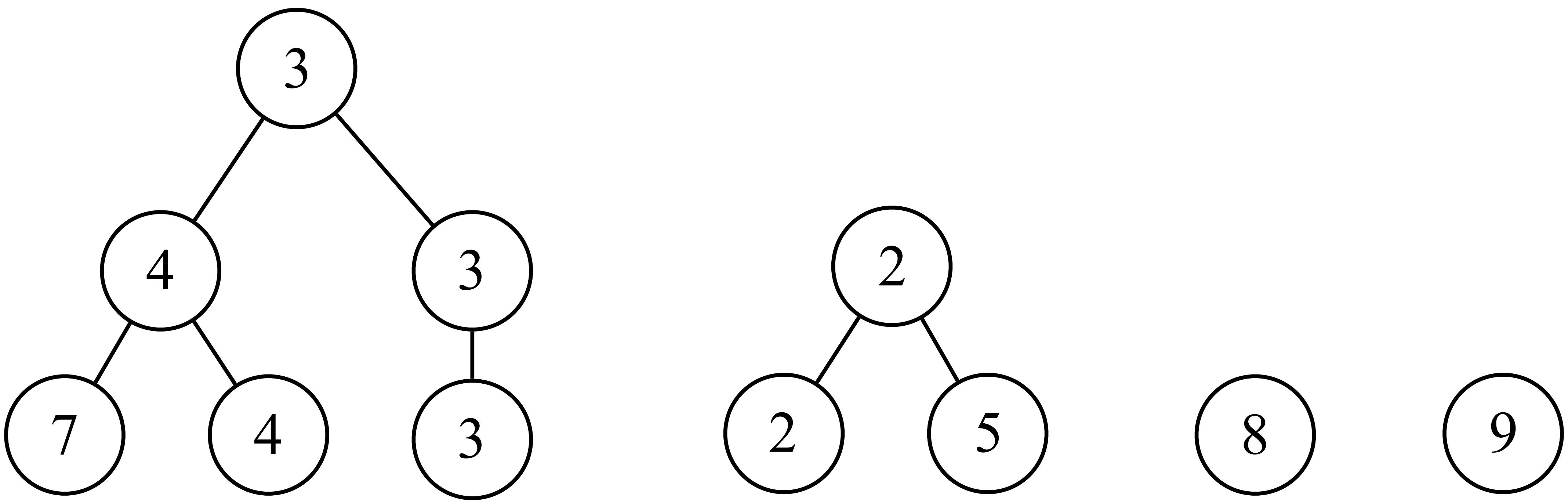
For every node x, for any two leaf descendants y, z of x, the distance dis(x, y) = dis(x, z).



Basics

For each node x, define height h(x) to be the distance between x and any of its descendant leaf node.

Define n_i to be the number of nodes of height i. Thus, n_0 denotes the number of keys, and $N = n_0 + n_1 + \ldots$

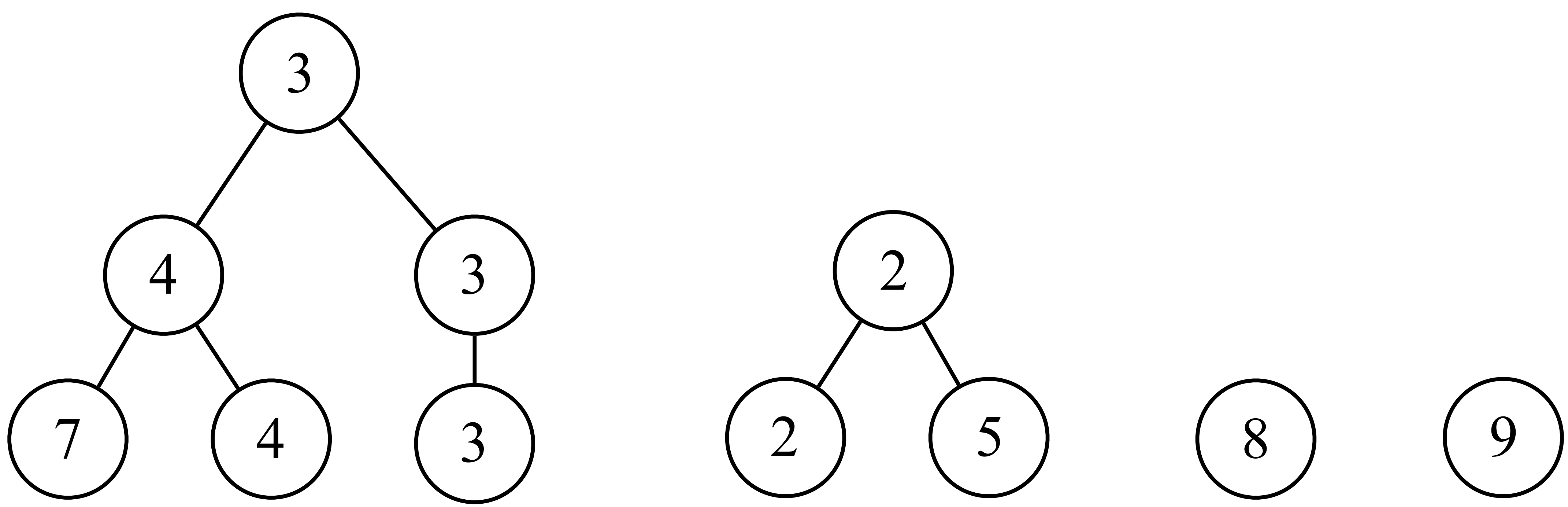


Basics

Pick an arbitrary constant α in $(1/2, 1)$.

It requires that $n_{i+1} \leq \alpha\, n_i$ initially and after every operation is done.

Thus, every tree has height $O(\log_{1/\alpha} n)$.



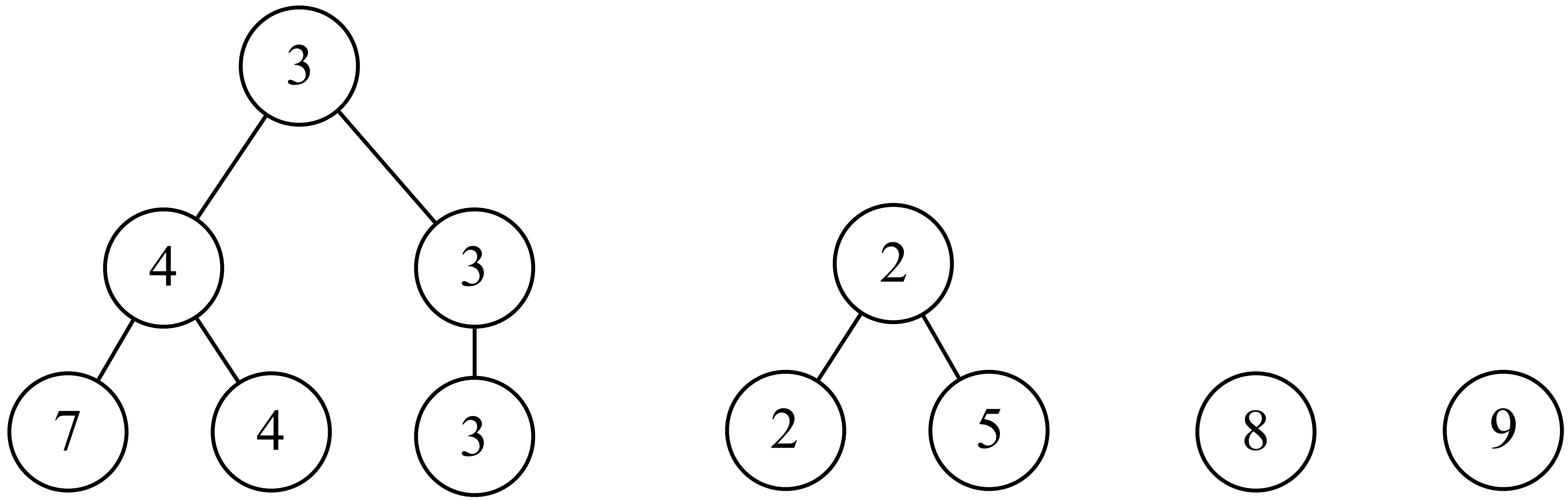
Potential

The potential function of a quake heap is defined as

$$N + T + B/(2\alpha - 1),$$

where N is the number nodes (not keys), T is the number of trees, and B is the number of degree-1 nodes.

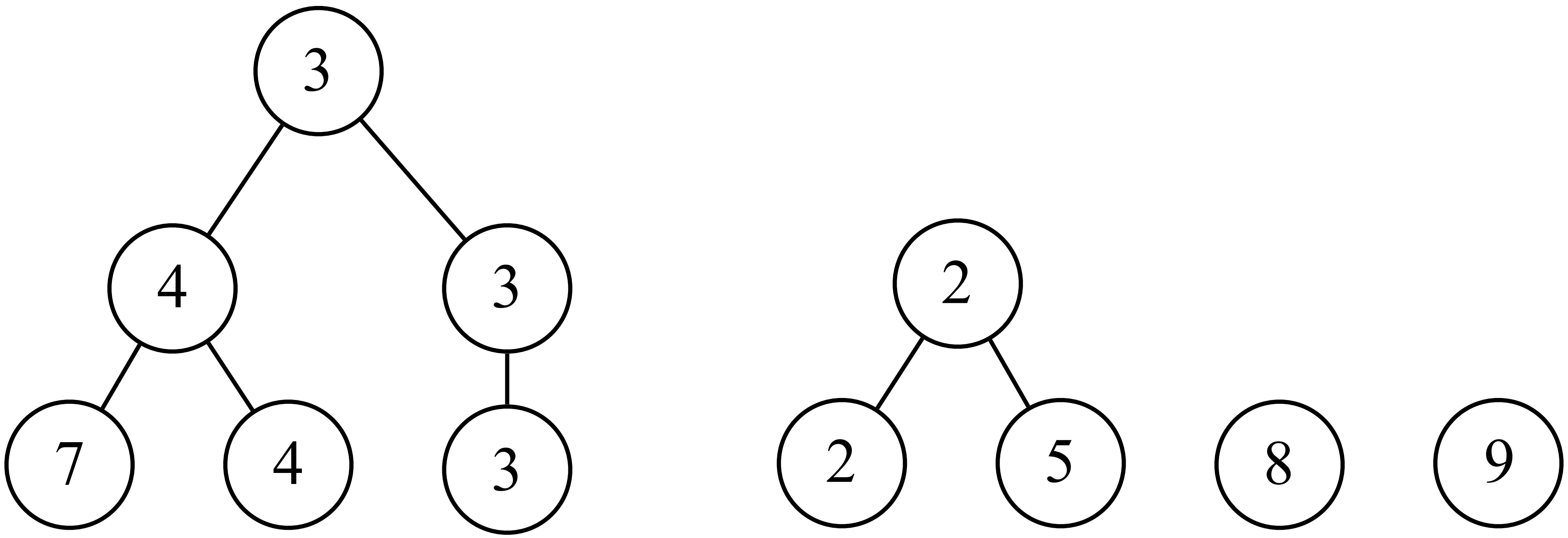
This example has N = 11, T = 4, and B = 1, so the potential is 16.



O(1)-time Insertion

Insert(x){// Insert a key x.
  Add a singleton tree so that the root has value x;
}

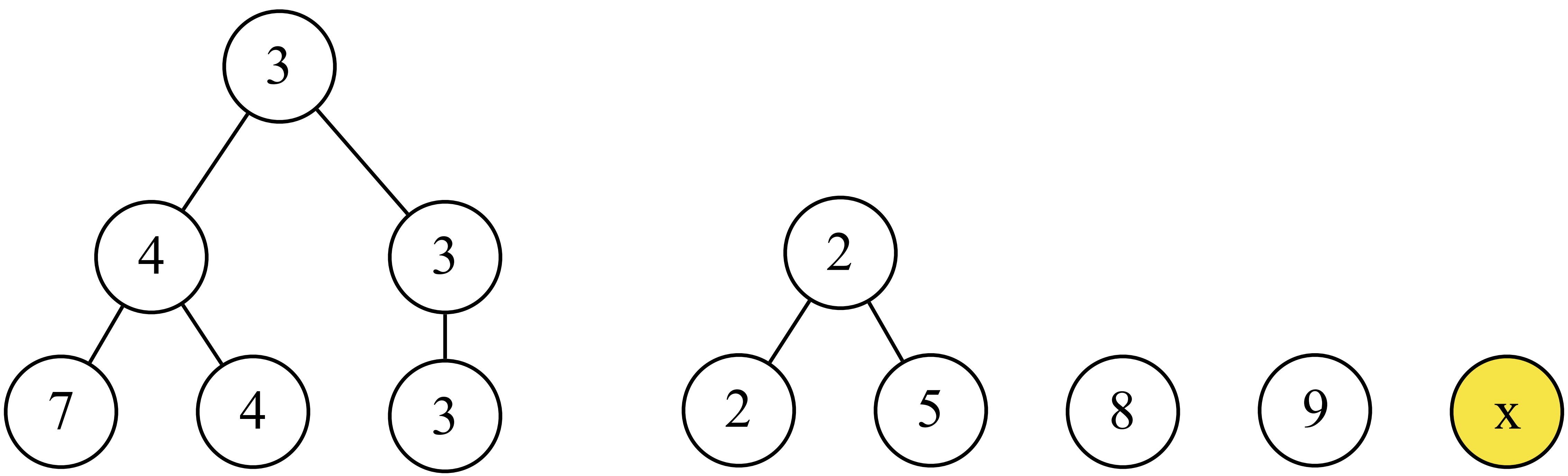
Actual cost is O(1), and potential change is $1 + 1 + 0/(2\alpha-1) = O(1)$.
So the amortized cost is O(1).



O(1)-time Insertion

Insert(x){// Insert a key x.
  Add a singleton tree so that the root has value x;
}

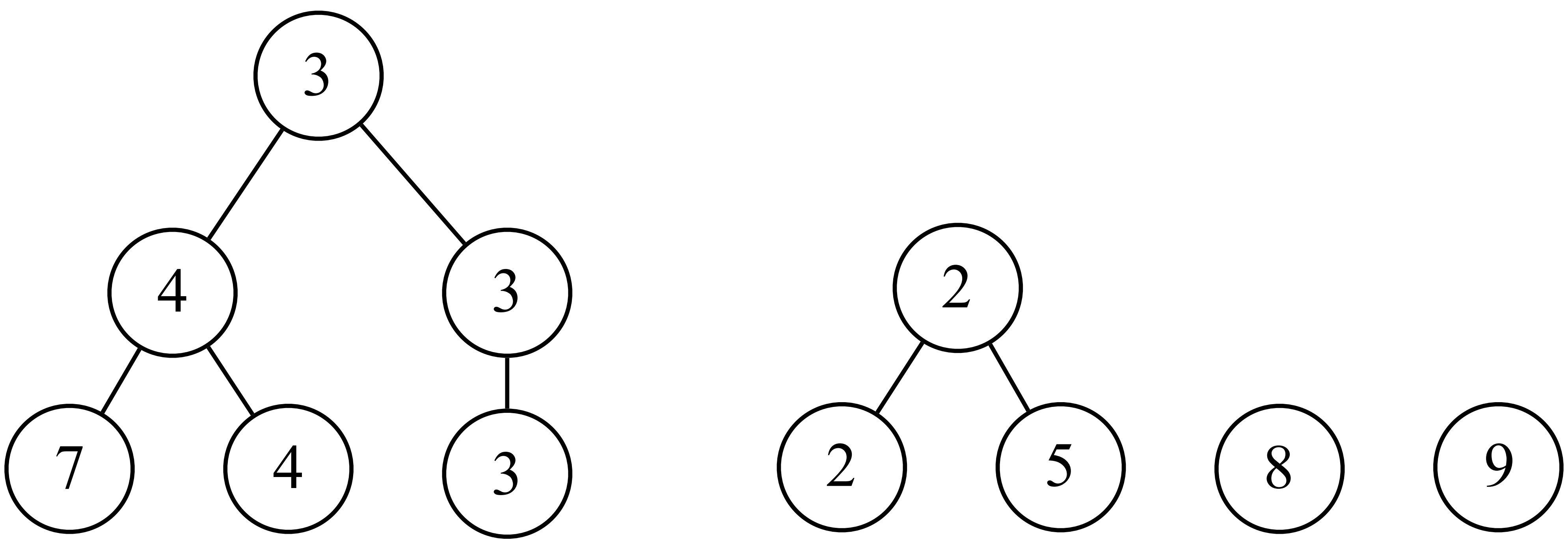
Actual cost is O(1), and potential change is $1 + 1 + 0/(2\alpha-1) = O(1)$.
So the amortized cost is O(1).



O(1)-time Decrease-Key

Decrease-Key(x, k){// give a pointer to key x, decrease its value to k.
    Cut the edge from the highest node with value x to its parent;
    Reduce the value of x's to k;
}

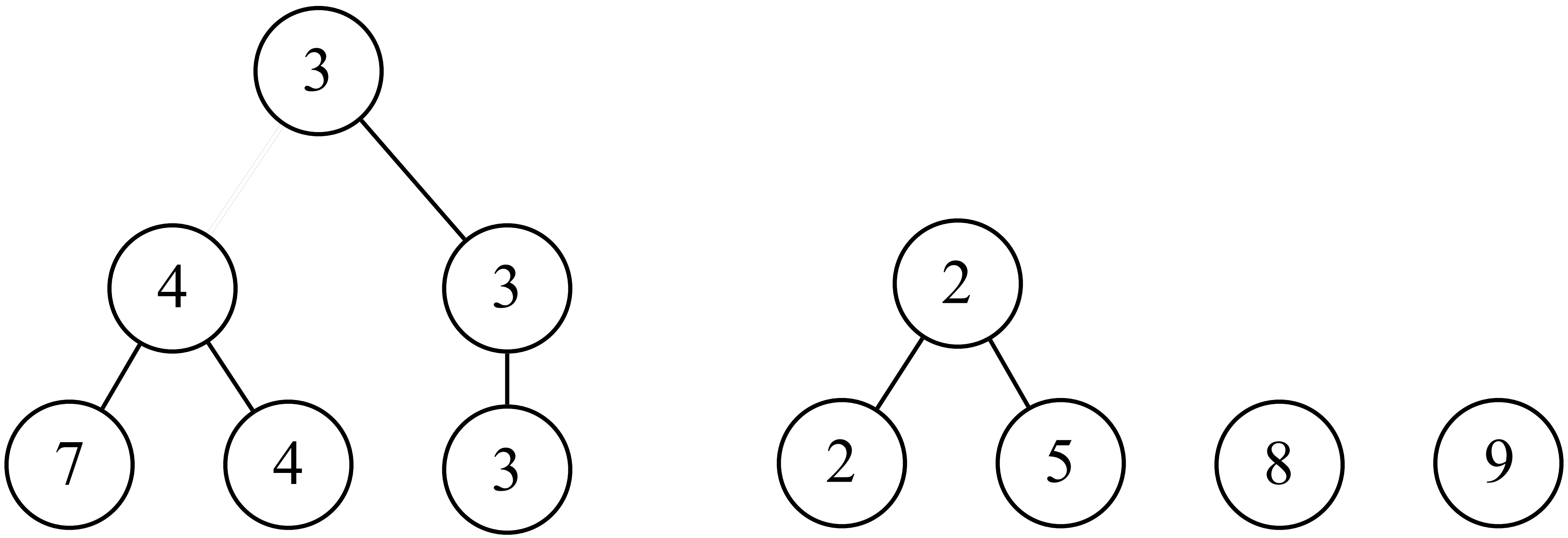
Actual cost is O(1), and potential change is $0 + 1 + 1/(2\alpha-1) = O(1)$.
So the amortized cost is O(1). For example, Decrease-Key(4, 2).



O(1)-time Decrease-Key

Decrease-Key(x, k){// give a pointer to key x, decrease its value to k.
    Cut the edge from the highest node with value x to its parent;
    Reduce the value of x's to k;
}

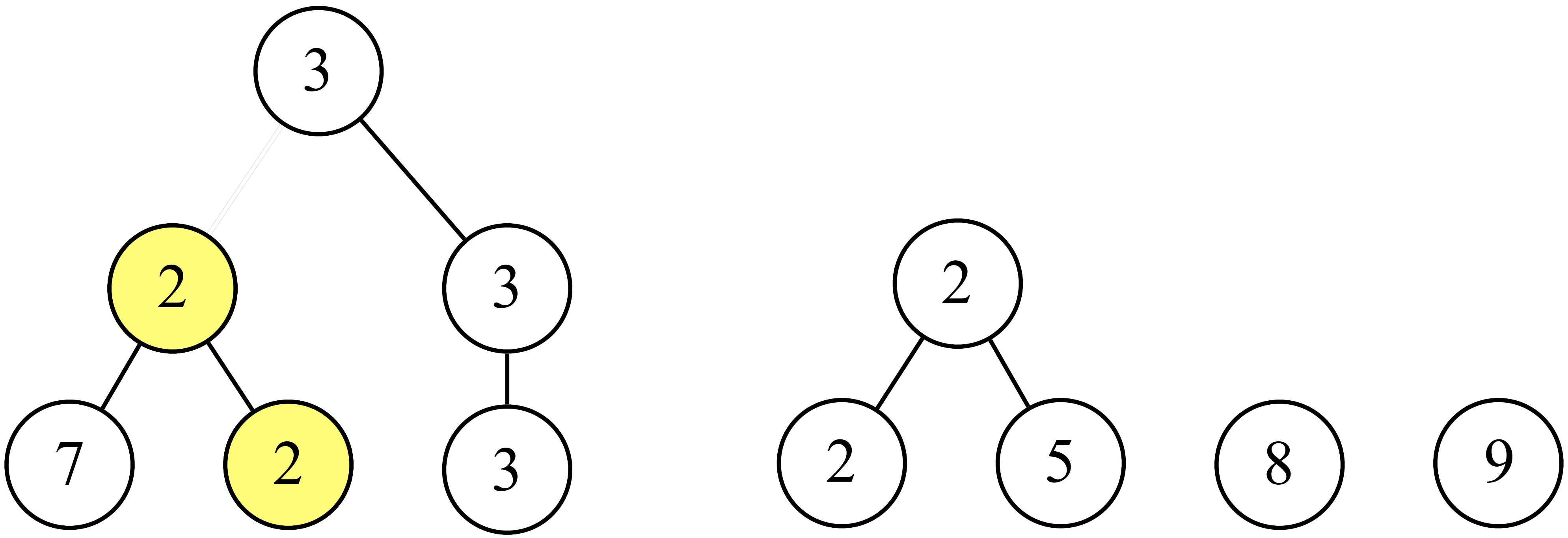
Actual cost is O(1), and potential change is $0 + 1 + 1/(2\alpha\text{-}1) = O(1)$.
So the amortized cost is O(1). For example, Decrease-Key(4, 2).



O(1)-time Decrease-Key

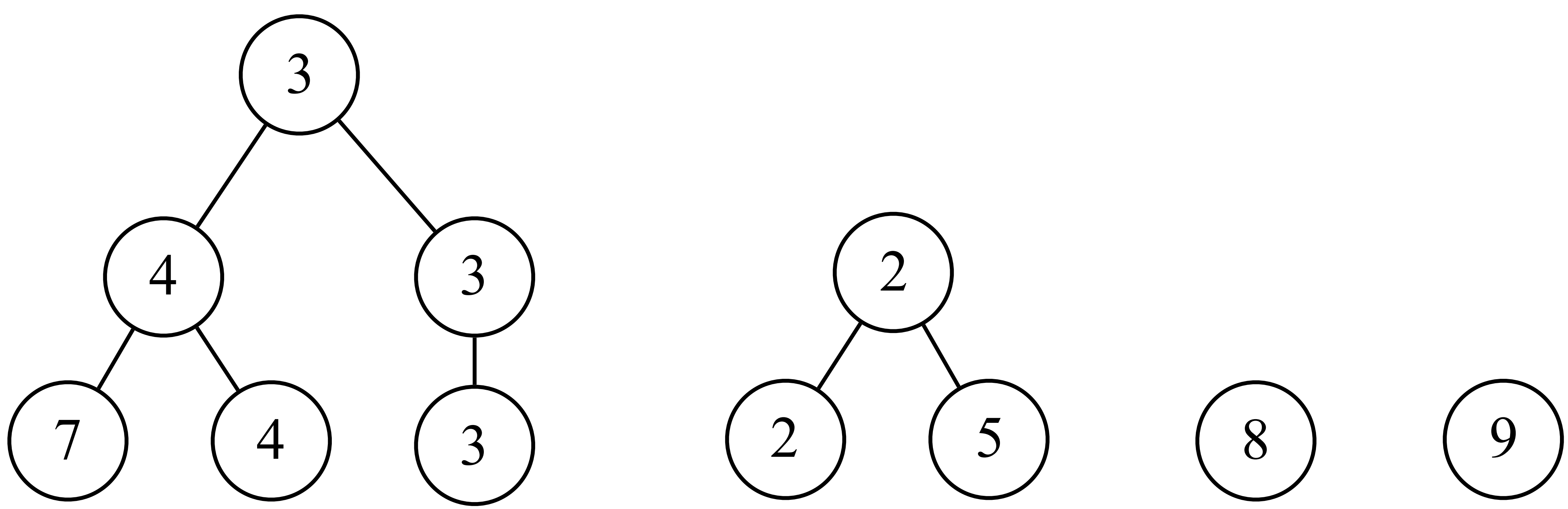
Decrease-Key(x, k){// give a pointer to key x, decrease its value to k.
Cut the edge from the highest node with value x to its parent;
Reduce the value of x's to k;
}

Actual cost is O(1), and potential change is $0 + 1 + 1/(2\alpha-1) = O(1)$.
So the amortized cost is O(1). For example, Decrease-Key(4, 2).



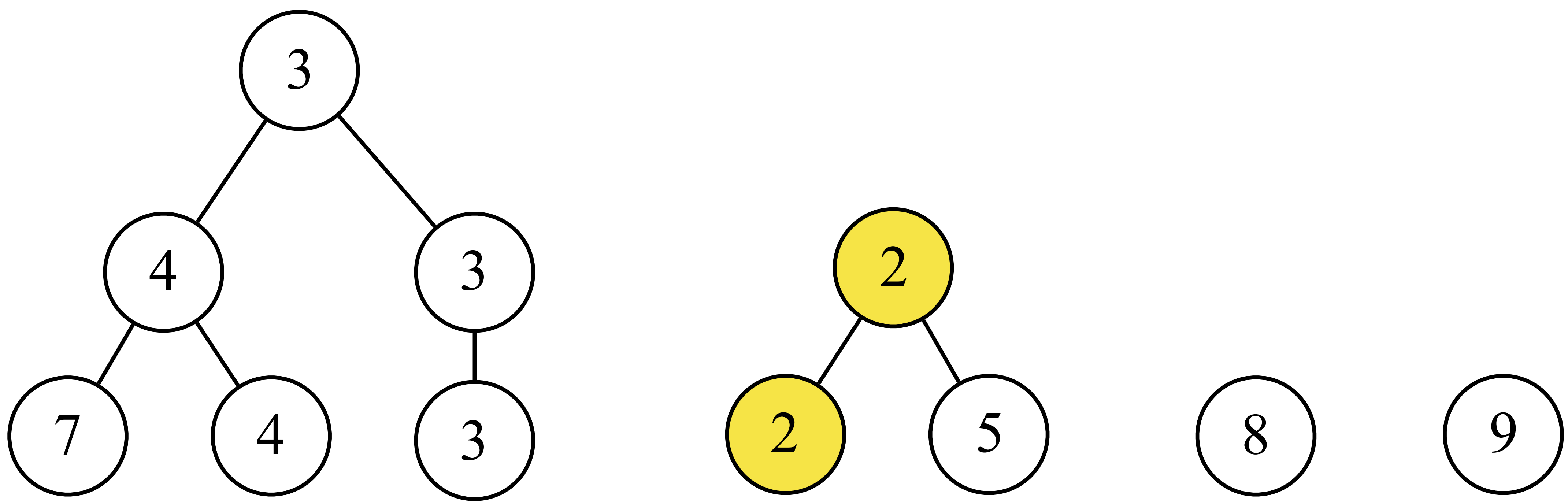
O(log n)-time Extract-Min

Extract-Min(){ // remove the min key and return it.
    Remove the entire root-to-leaf path that contains a min key;
    While(there exist two trees of equal height)
        Merge the two trees;
    If($n_{i+1} > \alpha n_i$)
        Remove all nodes of height > i;
}
For example, Extract-Min() twice.



O(log n)-time Extract-Min

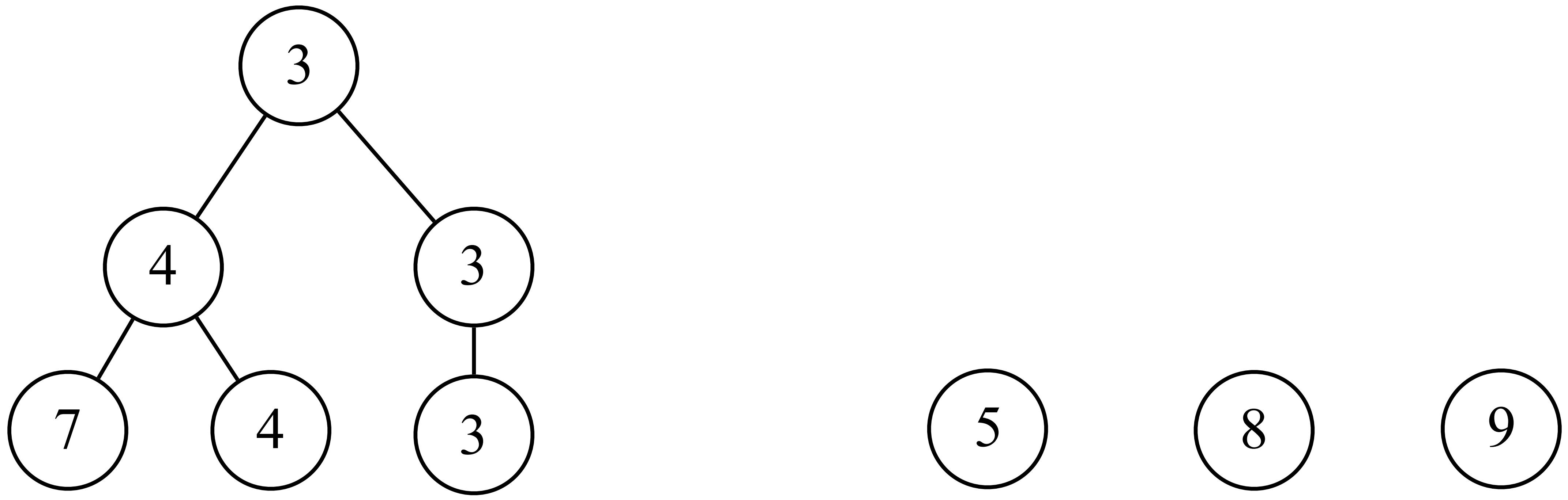
Extract-Min(){ // remove the min key and return it.
    Remove the entire root-to-leaf path that contains a min key;
    While(there exist two trees of equal height)
        Merge the two trees;
    If($n_{i+1} > \alpha n_i$)
        Remove all nodes of height > i;
}
For example, Extract-Min() twice.



O(log n)-time Extract-Min

Extract-Min(){ // remove the min key and return it.
    Remove the entire root-to-leaf path that contains a min key;
    While(there exist two trees of equal height)
        Merge the two trees;
    If($n_{i+1} > \alpha n_i$)
        Remove all nodes of height > i;
}

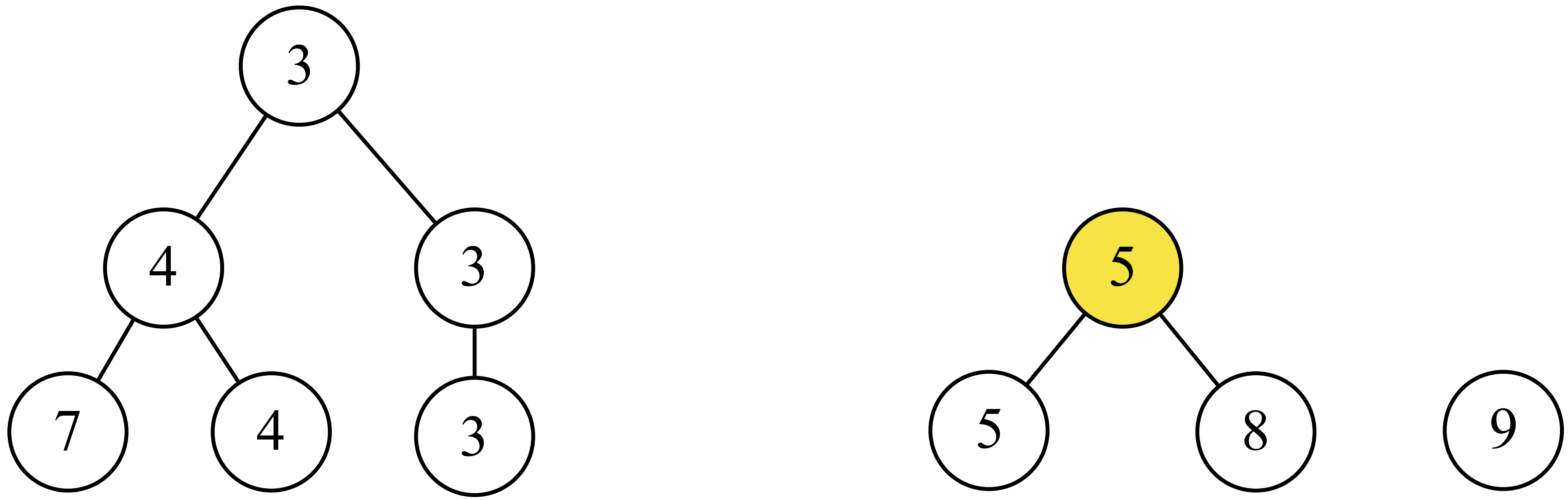
For example, Extract-Min() twice. Return 2.



O(log n)-time Extract-Min

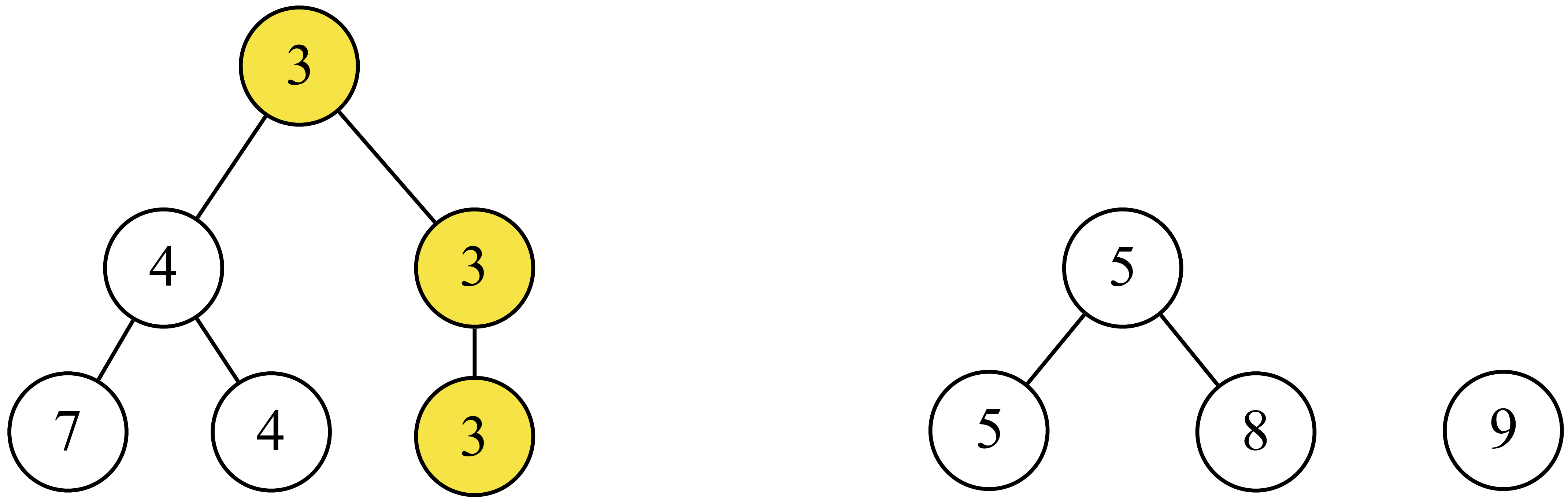
Extract-Min(){ // remove the min key and return it.
    Remove the entire root-to-leaf path that contains a min key;
    While(there exist two trees of equal height)
        Merge the two trees;
    If($n_{i+1} > \alpha n_i$)
        Remove all nodes of height > i;
}

For example, Extract-Min() twice. Return 2.



O(log n)-time Extract-Min

Extract-Min(){ // remove the min key and return it.
    Remove the entire root-to-leaf path that contains a min key;
    While(there exist two trees of equal height)
        Merge the two trees;
    If($n_{i+1} > \alpha n_i$)
        Remove all nodes of height > i;
}
For example, Extract-Min() twice. Return 2.



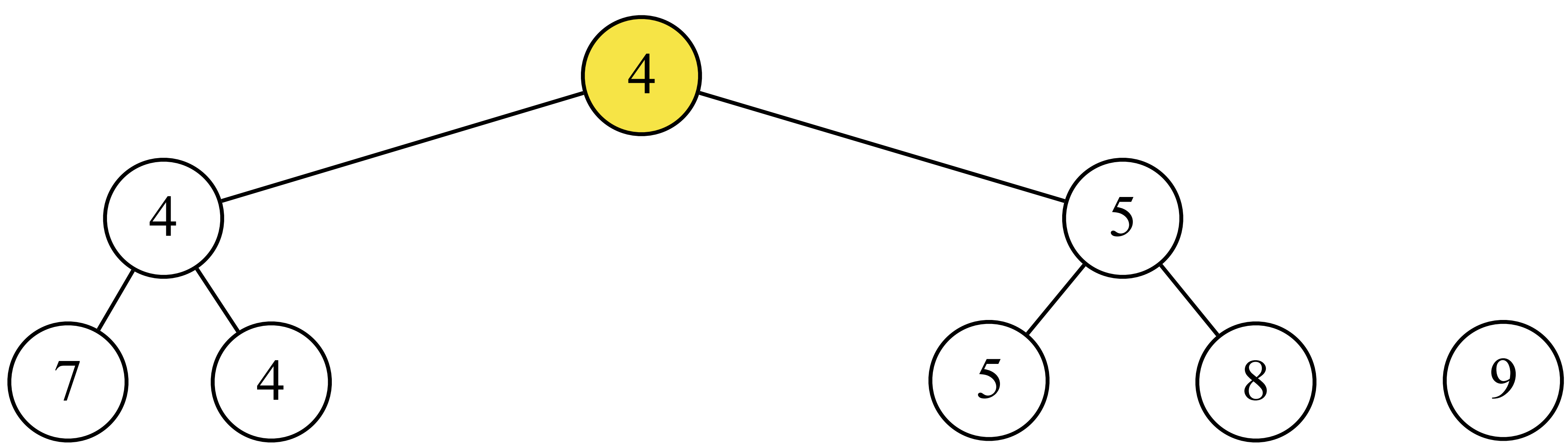
O(log n)-time Extract-Min

Extract-Min(){ // remove the min key and return it.
    Remove the entire root-to-leaf path that contains a min key;
    While(there exist two trees of equal height)
        Merge the two trees;
    If($n_{i+1} > \alpha n_i$)
        Remove all nodes of height > i;
}
For example, Extract-Min() twice. Return 2. Return 3.

O(log n)-time Extract-Min

Extract-Min(){ // remove the min key and return it.
    Remove the entire root-to-leaf path that contains a min key;
    While(there exist two trees of equal height)
        Merge the two trees;
    If($n_{i+1} > \alpha n_i$)
        Remove all nodes of height > i;
}

For example, Extract-Min() twice. Return 2. Return 3.



O(log n)-time Extract-Min

Extract-Min(){ // remove the min key and return it.

  Remove the entire root-to-leaf path that contains a min key;

  While(there exist two trees of equal height)

    Merge the two trees;

  If($n_{i+1} > \alpha n_i$)

    Remove all nodes of height $> i$;

}

	Before the Extract-Min()	Before Step r+1	After the Extract-Min()
# of nodes at height i	$n_i^{(0)}$	$n_i^{(r)}$	n_i
# of trees	$T^{(0)}$	$T^{(r)}$	T
# of degree-1 nodes	$b_i^{(0)}$	$b_i^{(r)}$	b_i

O(log n)-time Extract-Min

```
Extract-Min(){ // remove the min key and return it.
1:    Remove the entire root-to-leaf path that contains a min key;
2:    While(there exist two trees of equal height)
3:        Merge the two trees;
4:    If(n_{i+1} > αn_i)
5:        Remove all nodes of height > i;
}
```

Steps 1-3:

Actual cost is $T^{(0)} + O(\log_{1/\alpha} n)$, and pontential change is $\leq O(\log_{1/\alpha} n) - T^{(0)}$.

So amortized cost is $O(\log_{1/\alpha} n)$.

O(log n)-time Extract-Min

Extract-Min(){ // remove the min key and return it.
1: Remove the entire root-to-leaf path that contains a min key;
2: While(there exist two trees of equal height)
3: Merge the two trees;
4: If($n_{i+1} > \alpha n_i$)
5: Remove all nodes of height > i;
}

Steps 4-5:

Actual cost is $\sum_{j > i} n_j^{(0)}$, and potential change is
$\leq -\sum_{j > i} n_j^{(0)} + n_i^{(3)} - b_{i+1}^{(3)}/(2\alpha-1)$
$\leq -\sum_{j > i} n_j^{(0)} + n_i^{(3)} + (n_i^{(3)}-2n_{i+1}^{(3)})/(2\alpha-1)$ (because $n_i^{(3)} \geq 2n_{i+1}^{(3)} - b_{i+1}^{(3)}$)
$\leq -\sum_{j > i} n_j^{(0)} + n_i^{(3)} + (n_i^{(3)}-2\alpha n_i^{(3)})/(2\alpha-1)$ (because $n_{i+1}^{(3)} > \alpha n_i^{(3)}$)
$\leq -\sum_{j > i} n_j^{(0)}$.
Thus, the amortized cost is O(1).