# Growth of Functions

# Asymptotic notation

| Function | small-ω | big-Ω | Θ | big-O | small-o |
|---|---|---|---|---|---|
| Real | $>$ | $\geq$ | $=$ | $\leq$ | $<$ |

# Θ-Notation

- Definition: $\Theta(f(n))=\{g$: there are constants $c_1$, $c_2$, $n_0>0$ such that $c_1 f(n) \leq g(n) \leq c_2 f(n)$ for every $n \geq n_0\}$
- $g(n)=\Theta(f(n))$ (precisely, $g(n) \in \Theta(f(n))$) means $f(n)$ is an asymptotically tight bound for $g(n)$.

# Example

- Problem: Show that $9n^3-6n^2+2n=\Theta(n^3)$.
- Goal: find out constant $c_1,c_2,n_0>0$ s. t. $c_1n^3\leq 9n^3-6n^2+2n\leq c_2n^3$ for $n\geq n_0$.
- Pick $c_1=8$ and solve $8n^3\leq 9n^3-6n^2+2n$
  - $0\leq n^2-6n+2$: setting $n_0\geq 10$ is sufficient.
- Pick $c_2=9$ and solve $9n^3-6n^2+2n\leq 9n^3$
  - $0\leq 6n-2$: setting $n_0\geq 10$ is sufficient.

# O-Notation

- Definition: $O(f(n))=\{g$: there are constants $c$, $n_0>0$ such that $g(n)\leq cf(n)$ for every $n\geq n_0\}$
- $g(n)=O(f(n))$ (precisely, $g(n)\in O(f(n))$) means $f(n)$ is an asymptotically upper bound for $g(n)$.
- $\Theta(f(n))\subseteq O(f(n))$

# $\Omega$-Notation

- Definition: $\Omega(f(n))=\{g$: there are constants $c$, $n_0>0$ such that $g(n)\geq cf(n)$ for every $n\geq n_0\}$
- $g(n)=\Omega(f(n))$ (precisely, $g(n)\in\Omega(f(n))$) means $f(n)$ is an asymptotically lower bound for $g(n)$.
- $\Theta(f(n))\subseteq\Omega(f(n))$
- $g(n)=\Theta(f(n))$
  $\Rightarrow g(n)=O(f(n))$ and $g(n)=\Omega(f(n))$

# Best and Worst

- The running time of an algorithm is O(f(n)) if the worst-case running time is O(f(n)).
- The running time of an algorithm is Ω(f(n)) if the best-case running time is Ω(f(n)).

iff: if and only if

# o-Notation

- Definition: $o(f(n))=\{g$: for every constant $c>0$, there exists a constant $n_0>0$ such that $0\leq g(n)<cf(n)$ for every $n\geq n_0\}$
- $g(n)=o(f(n))$ iff $\lim_{n\to\infty} g(n)/f(n)=0$
- $f(n)\neq o(f(n))$ (precisely, $f(n)\notin o(f(n))$)
- $o(f(n))\subseteq O(f(n))$

iff: if and only if

# $\omega$-Notation

- Definition: $\omega(f(n))=\{g$: for every constant $c>0$, there exists a constant $n_0>0$ such that $0\leq cf(n)<g(n)$ for every $n\geq n_0\}$
- $g(n)=\omega(f(n))$ iff $\lim_{n\to\infty} f(n)/g(n)=0$
- $f(n)\neq\omega(f(n))$ (precisely, $f(n)\notin\omega(f(n))$)
- $\omega(f(n))\subseteq\Omega(f(n))$

# Pitfall

- Algorithm A is in $O(n^2)$-time and algorithm B is in $O(n\log n)$-time.
- Is B faster?

# Pitfall

- Algorithm A is in $O(n^2)$-time and algorithm B is in $O(n\log n)$-time.
- Is B faster?
  - We cannot conclude $y \le x$ when $x \le 100$ and $y \le 70$.

# Pitfall

- Algorithm A is in $O(n^2)$-time and algorithm B is in $O(n\log n)$-time.
- Is B faster?
  - We cannot conclude $y \leq x$ when $x \leq 100$ and $y \leq 70$.
- A is insertion sort, and B is merge sort. If the input sequence is already sorted, then insertion sort is faster!

# Pitfall

- Algorithm A is in $\Theta(n^2)$-time and algorithm B is in $\Theta(n^{\log 3/\log 2})$-time.
- Is B faster?

# Pitfall

- Algorithm A is in $\Theta(n^2)$-time and algorithm B is in $\Theta(n^{\log 3/\log 2})$-time.
- Is B faster?
  - Yes if n is sufficiently large.
- Is 20000 sufficiently large?

# Pitfall

- Algorithm A is in $\Theta(n^2)$-time and algorithm B is in $\Theta(n^{\log 3/\log 2})$-time.
- Is B faster?
  - Yes if n is sufficiently large.
- Is 20000 sufficiently large?
  - It depends on the constants!
- Try to use Karatsuba multiplication algorithm in HW1-1.

# Some Useful Facts

$$\sum_{k=1}^{n} k^p = \Theta(n^{p+1})$$

$$\sum_{k=1}^{n} \frac{1}{k(k+1)} = 1 - \frac{1}{n+1}$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

$$\sum_{k=1}^{\infty} kx^k = \frac{x}{(1-x)^2}, |x| < 1$$

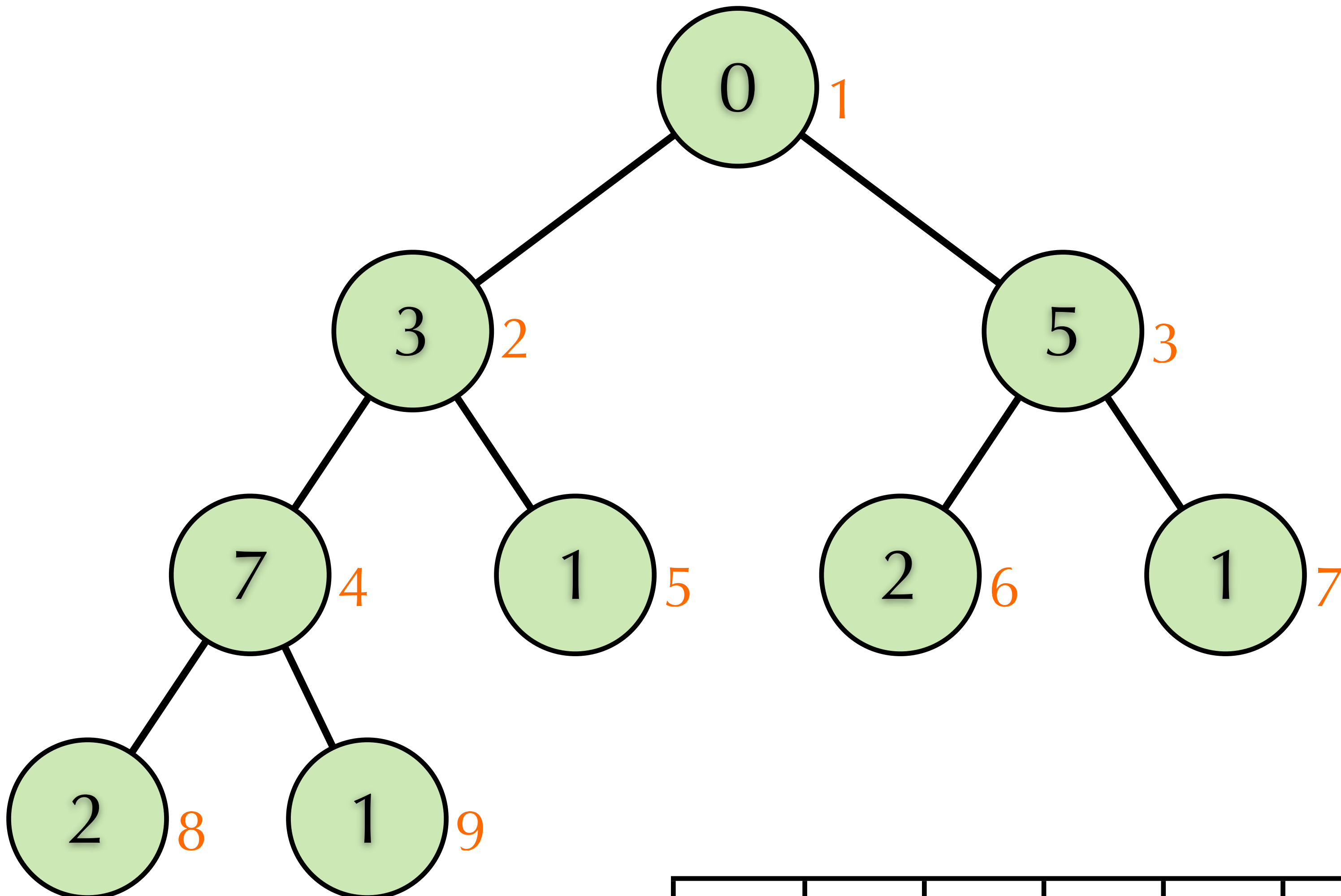
$$\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$$

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

# Practice: Build Heap

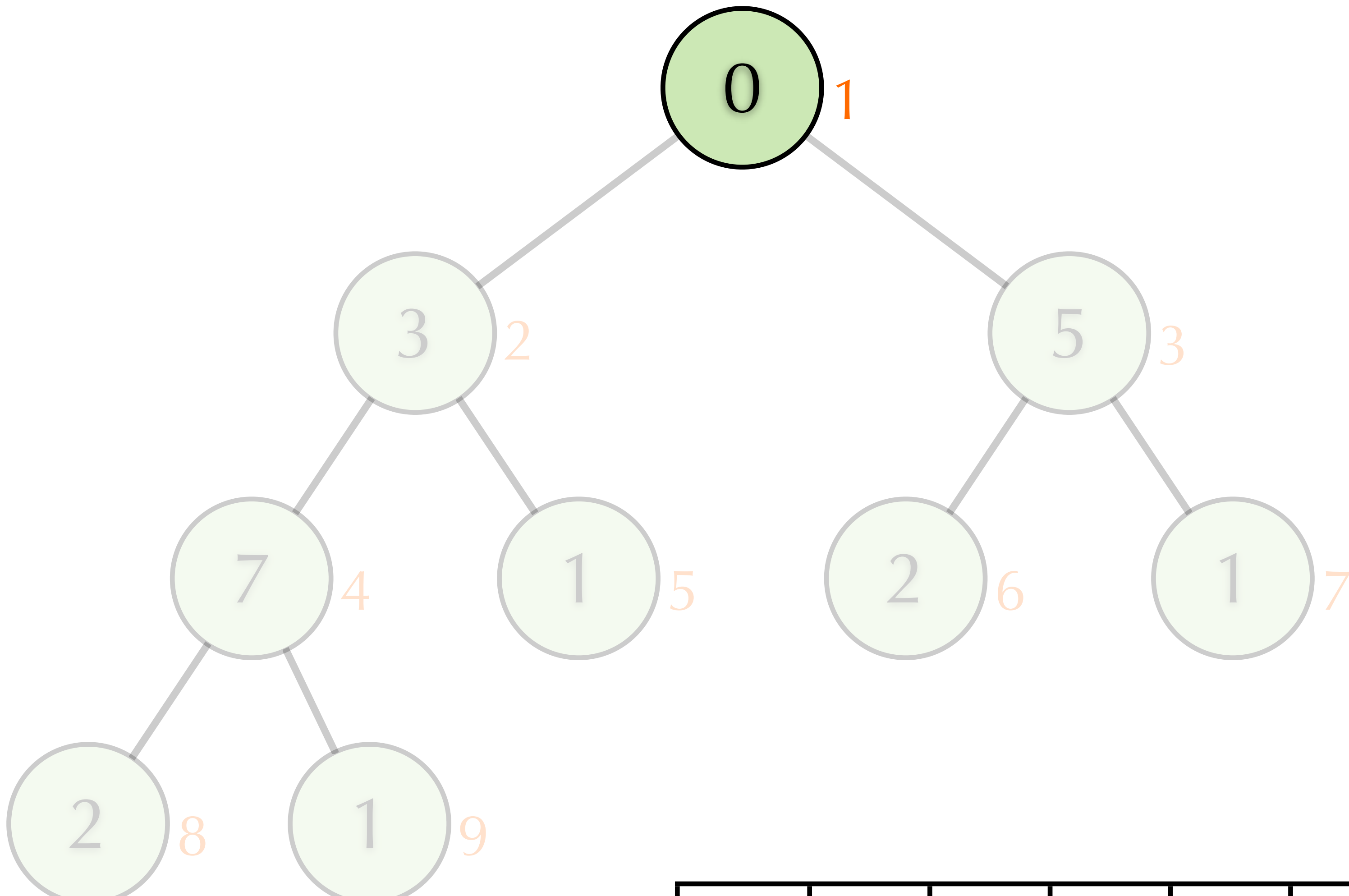
- Building a min binary heap in an array.
- Suppose the sequence are a[1],…,a[n].
- Method 1: Repeated insertion
  For i = 1 to n do
      Insert a[i] in to the heap
- Method 2: Bottom-Up build
  For i= $\lfloor n/2 \rfloor$ downto 1 do
      Heapify the subtree rooted a[i]

# Input



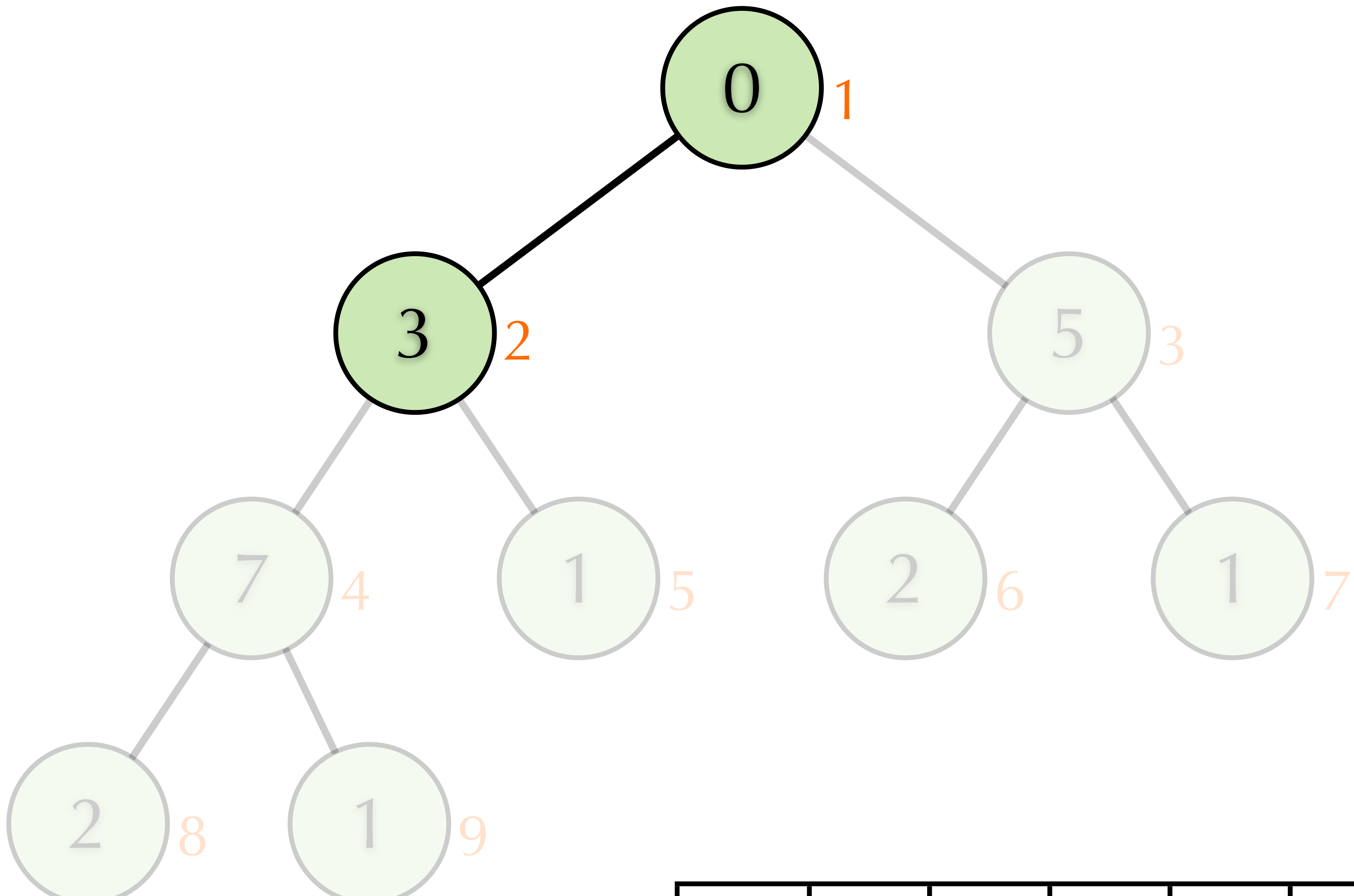

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 3 | 5 | 7 | 1 | 2 | 1 | 2 | 1 |

# Method 1



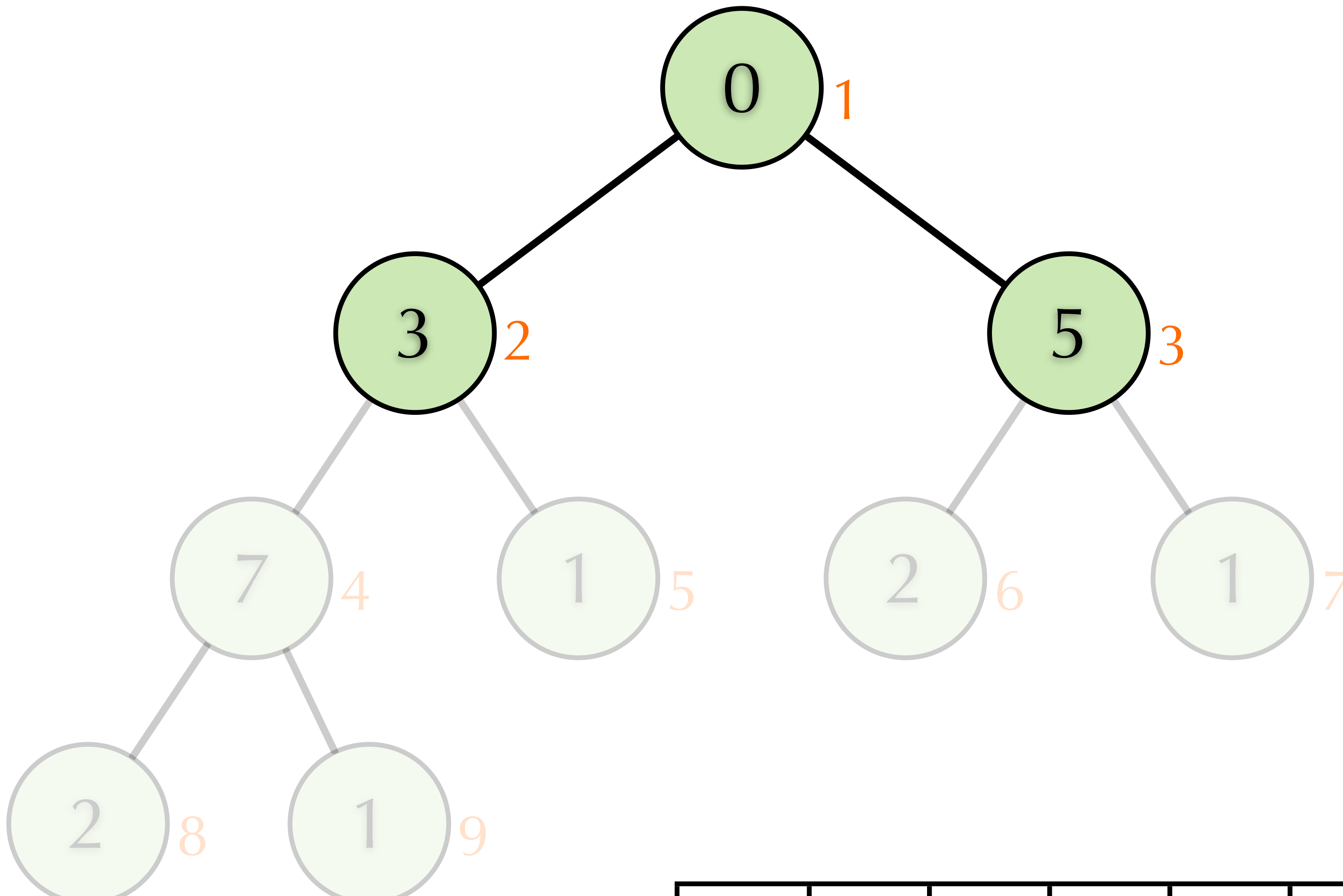

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 3 | 5 | 7 | 1 | 2 | 1 | 2 | 1 |

# Method 1



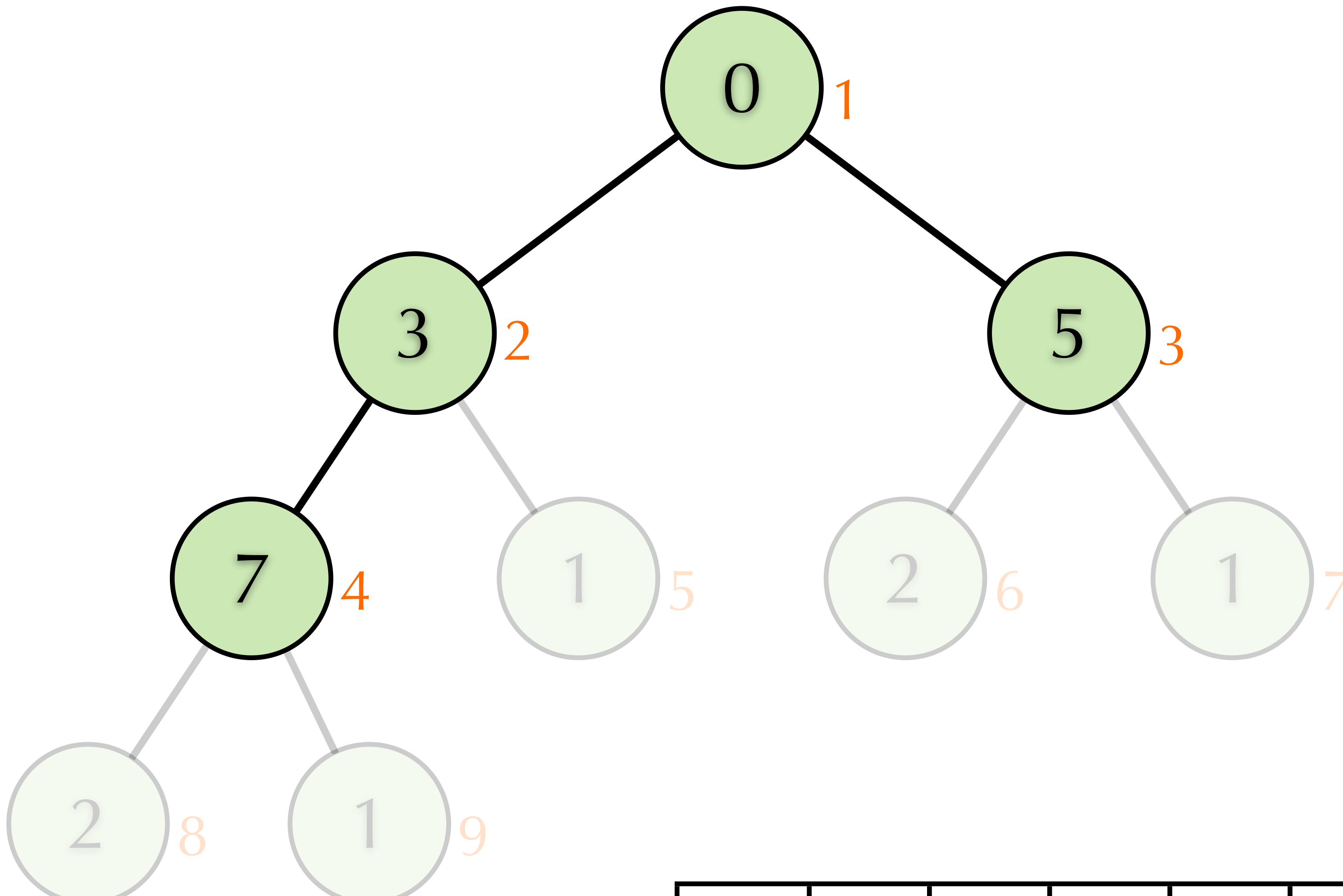

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 3 | 5 | 7 | 1 | 2 | 1 | 2 | 1 |

# Method 1



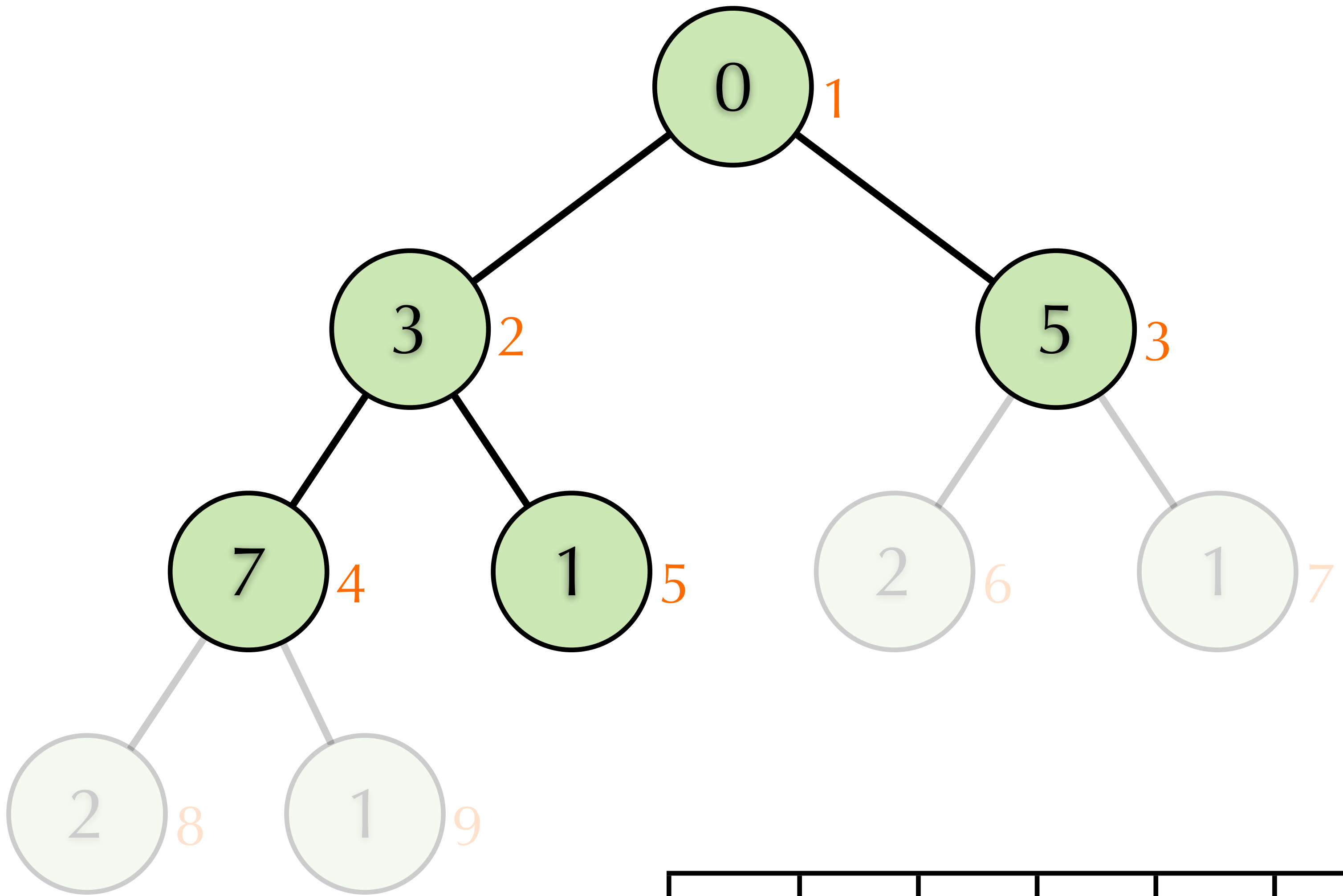

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 3 | 5 | 7 | 1 | 2 | 1 | 2 | 1 |

# Method 1



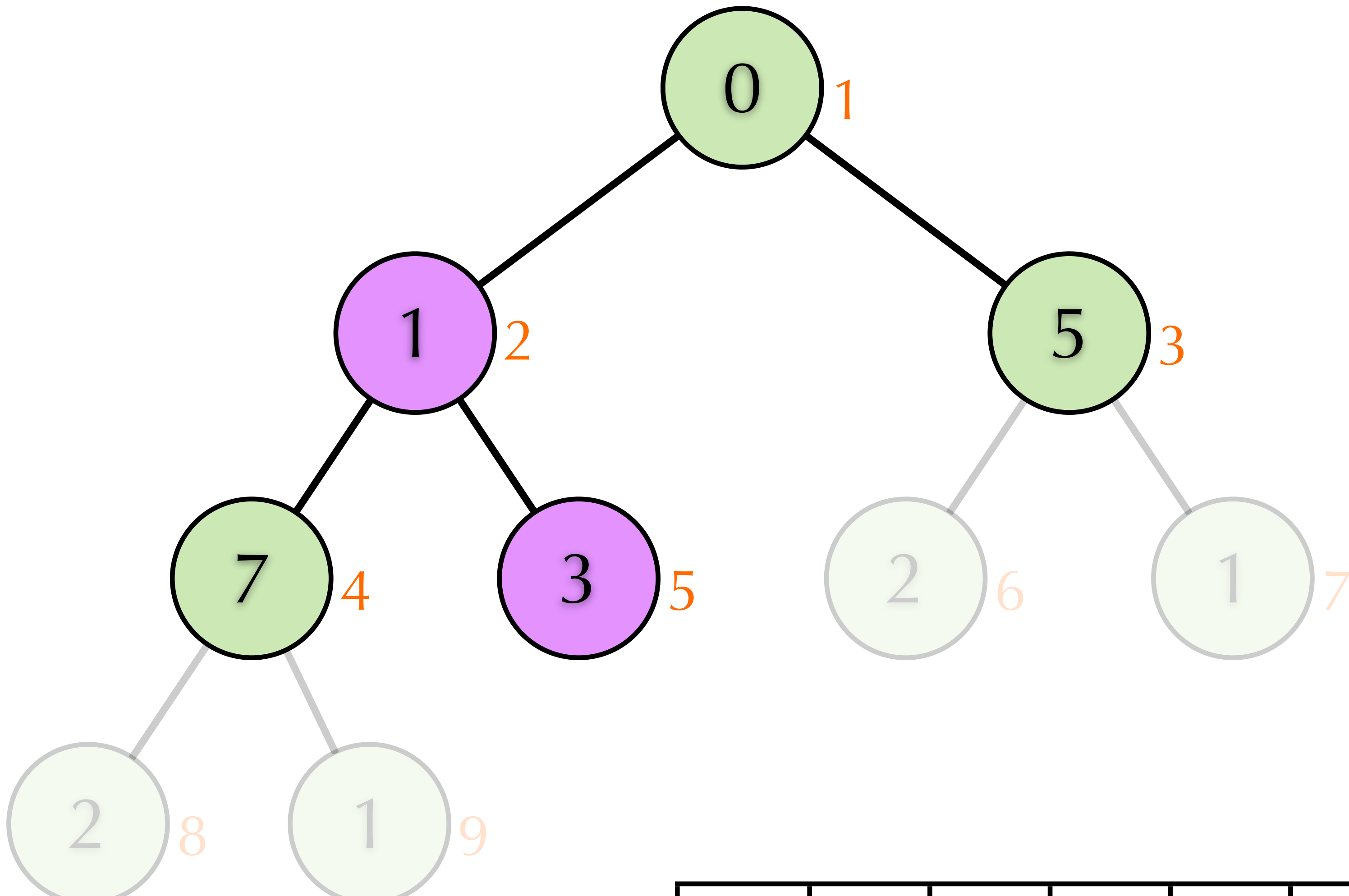

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 3 | 5 | 7 | 1 | 2 | 1 | 2 | 1 |

# Method 1



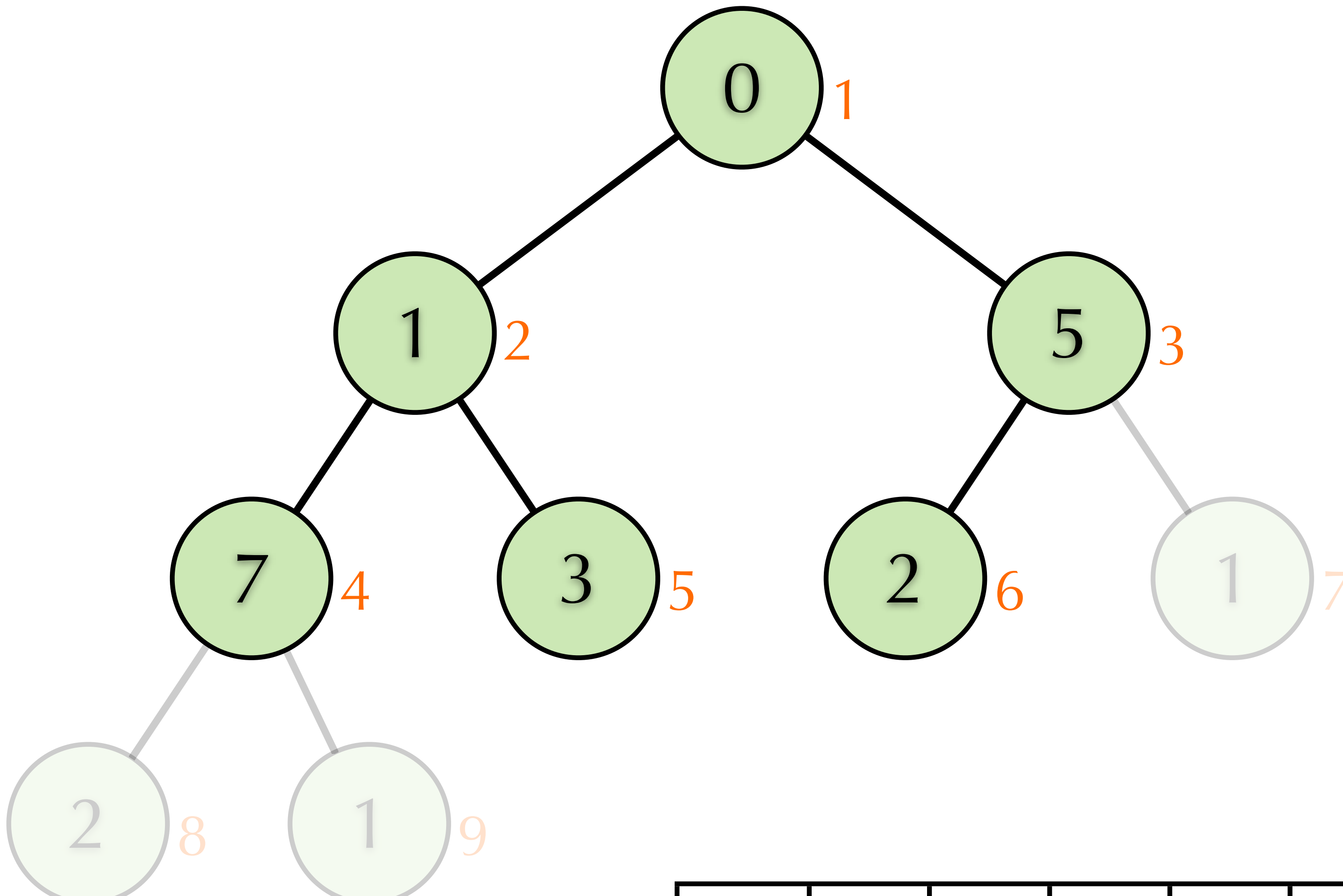

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 3 | 5 | 7 | 1 | 2 | 1 | 2 | 1 |

# Method 1



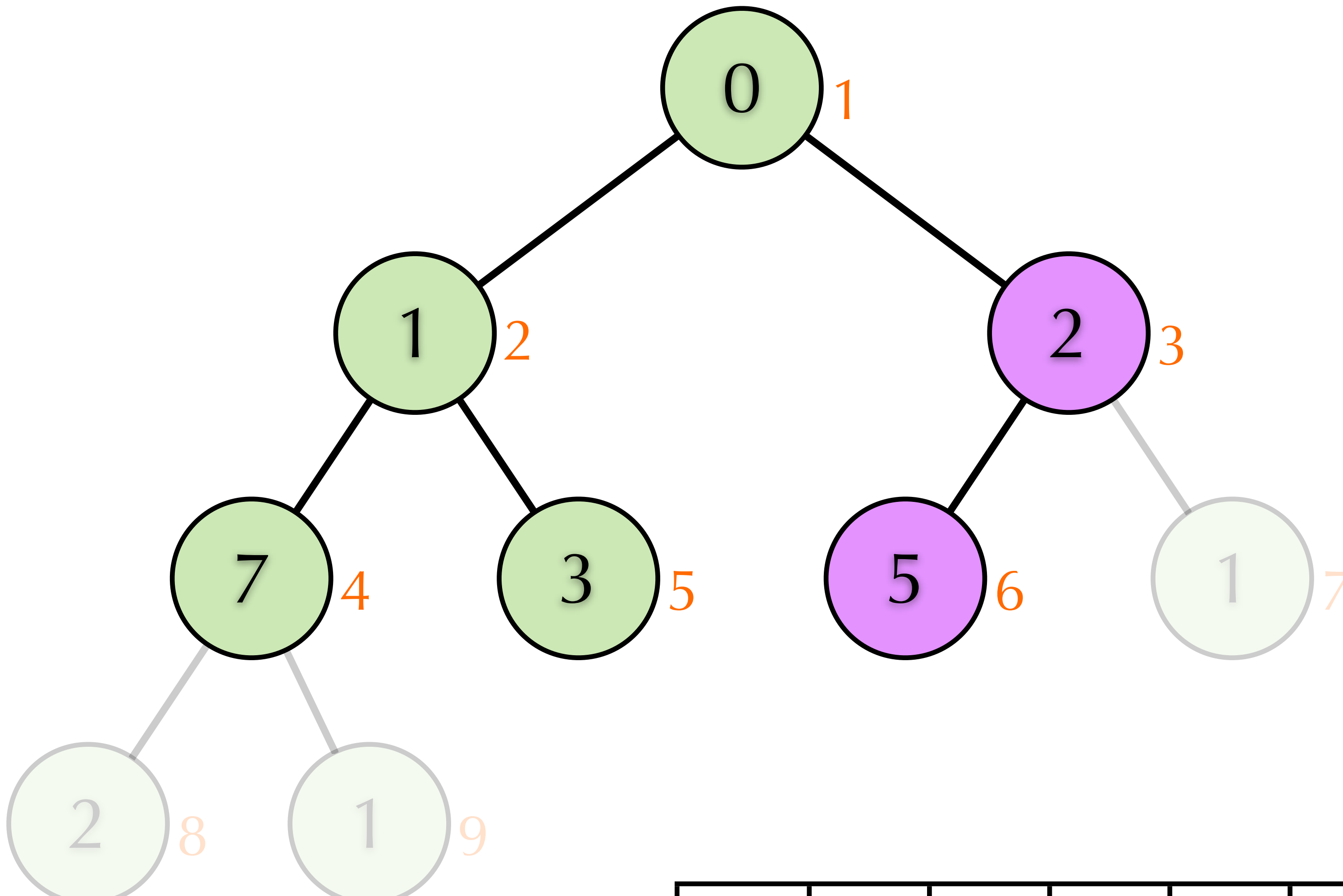

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 1 | 5 | 7 | 3 | 2 | 1 | 2 | 1 |

# Method 1



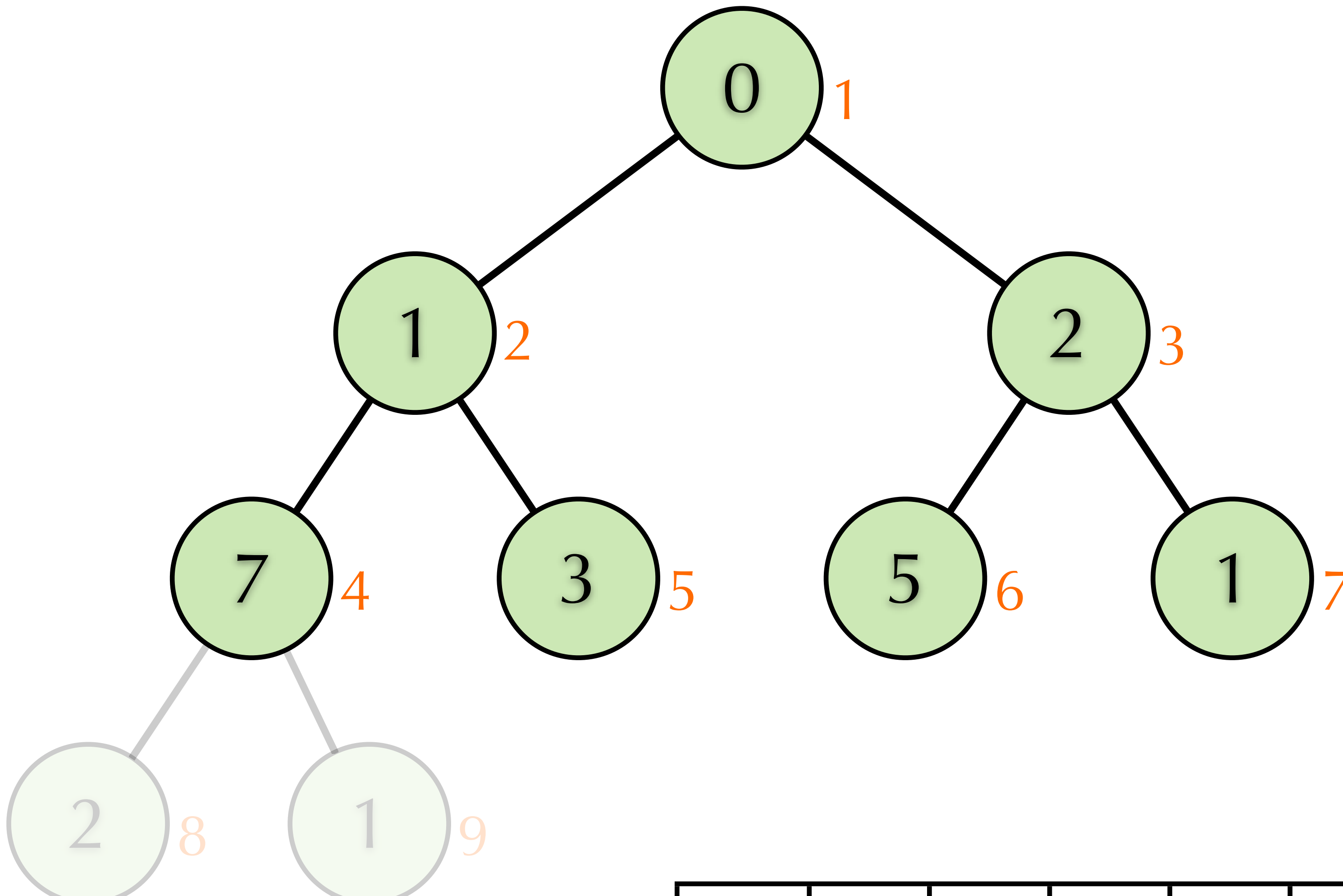

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 1 | 5 | 7 | 3 | 2 | 1 | 2 | 1 |

# Method 1



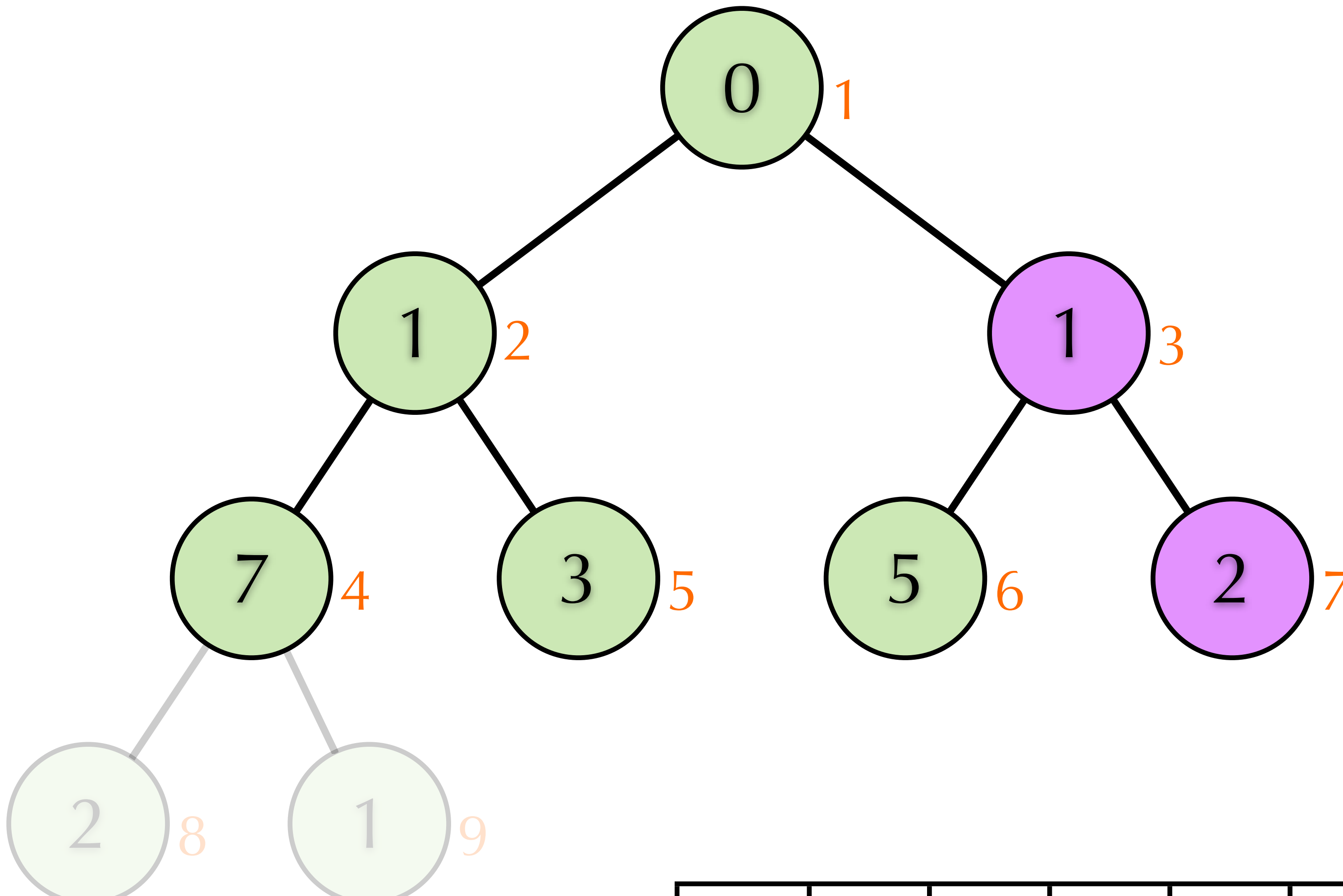

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 1 | 2 | 7 | 3 | 5 | 1 | 2 | 1 |

# Method 1



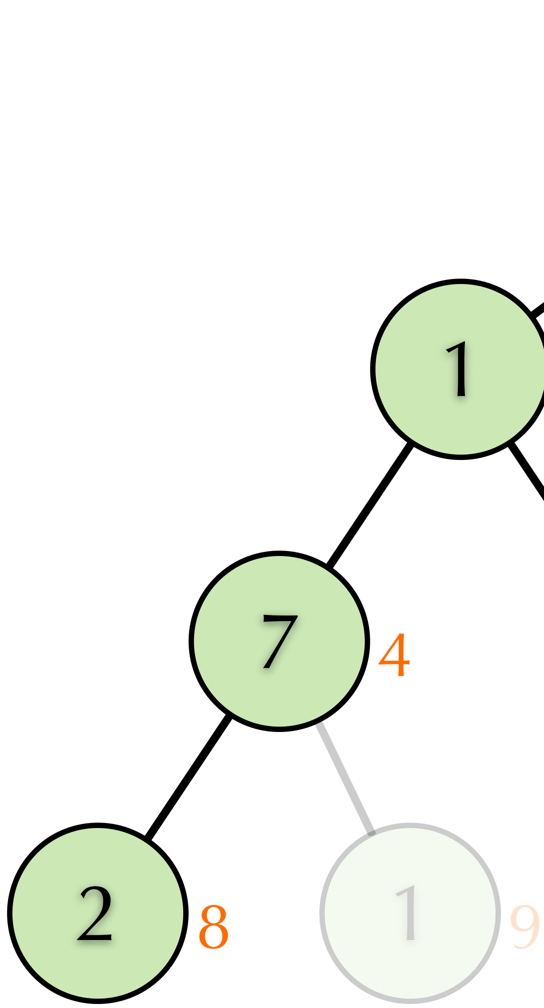

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 1 | 2 | 7 | 3 | 5 | 1 | 2 | 1 |

# Method 1



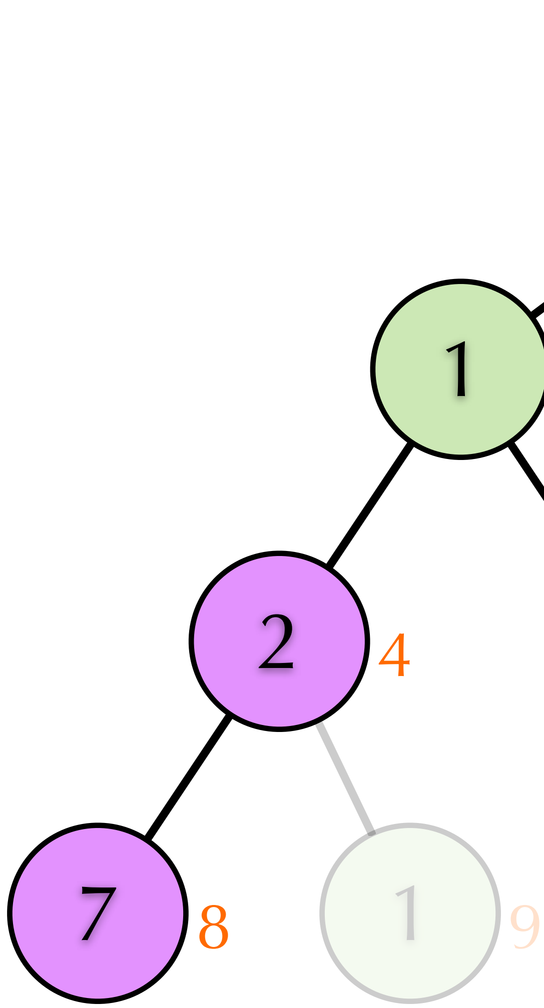

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 1 | 1 | 7 | 3 | 5 | 2 | 2 | 1 |

# Method 1



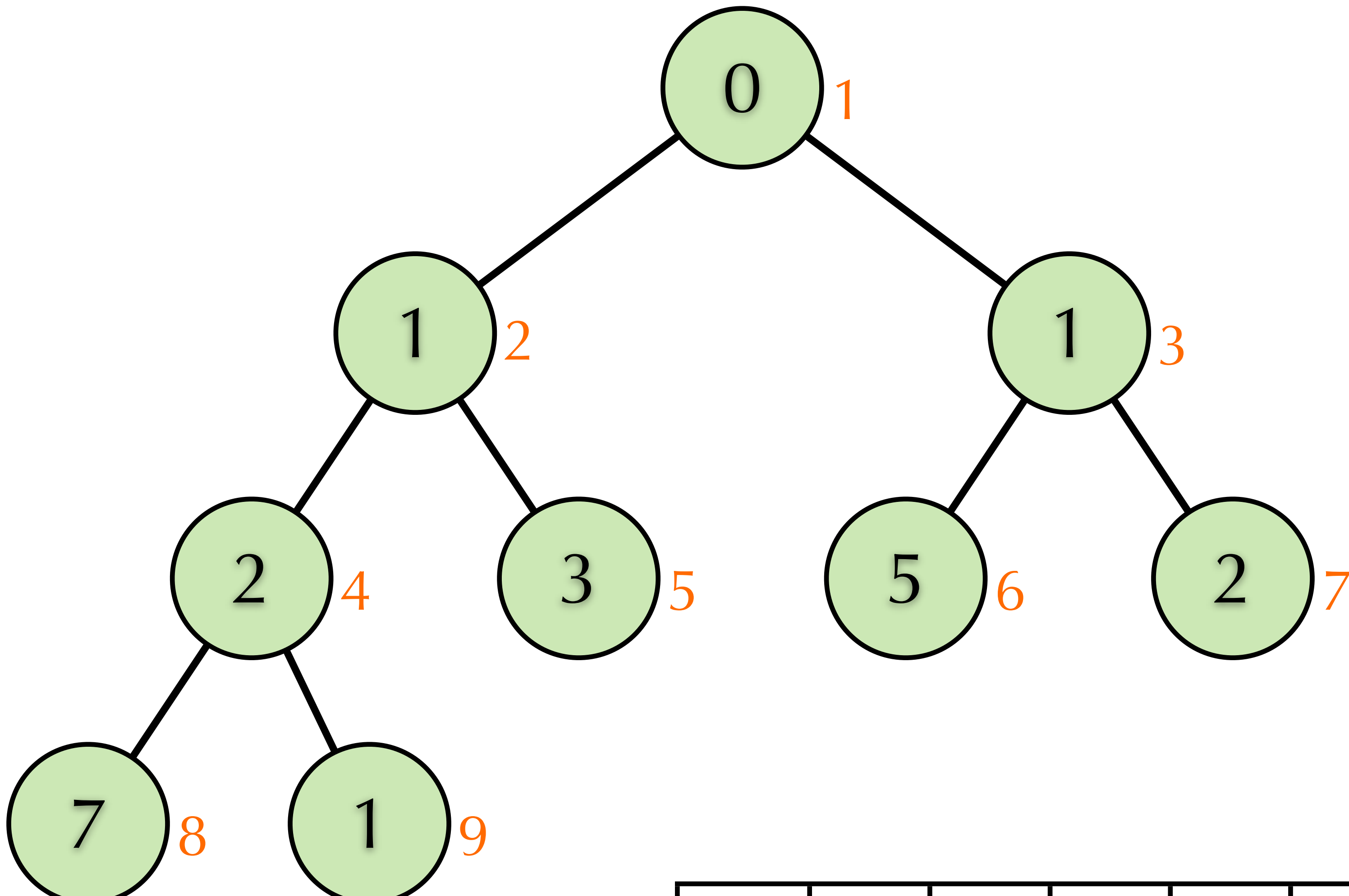

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 1 | 1 | 7 | 3 | 5 | 2 | 2 | 1 |

# Method 1



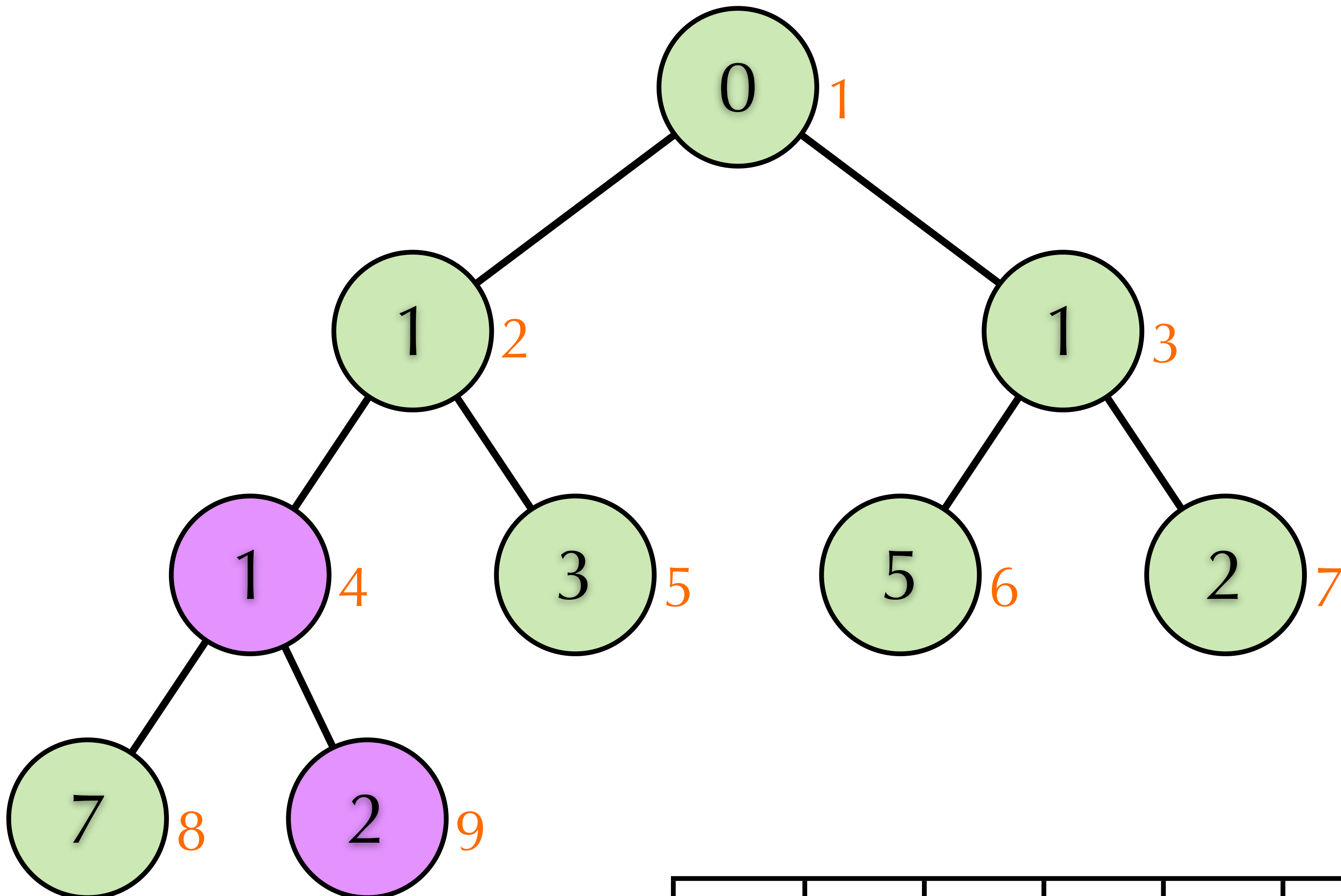

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 1 | 1 | 2 | 3 | 5 | 2 | 7 | 1 |

# Method 1



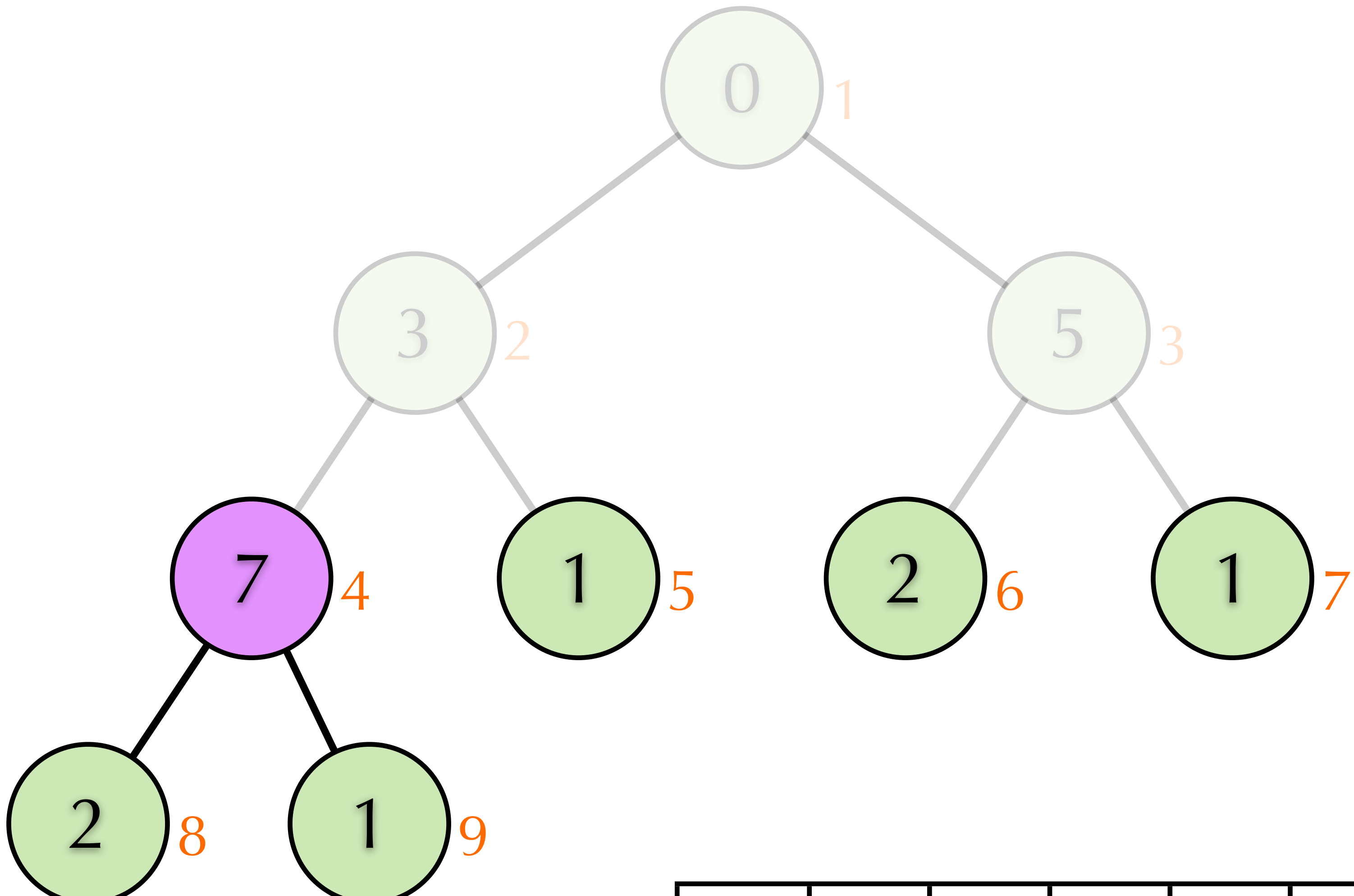

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 1 | 1 | 2 | 3 | 5 | 2 | 7 | 1 |

# Method 1



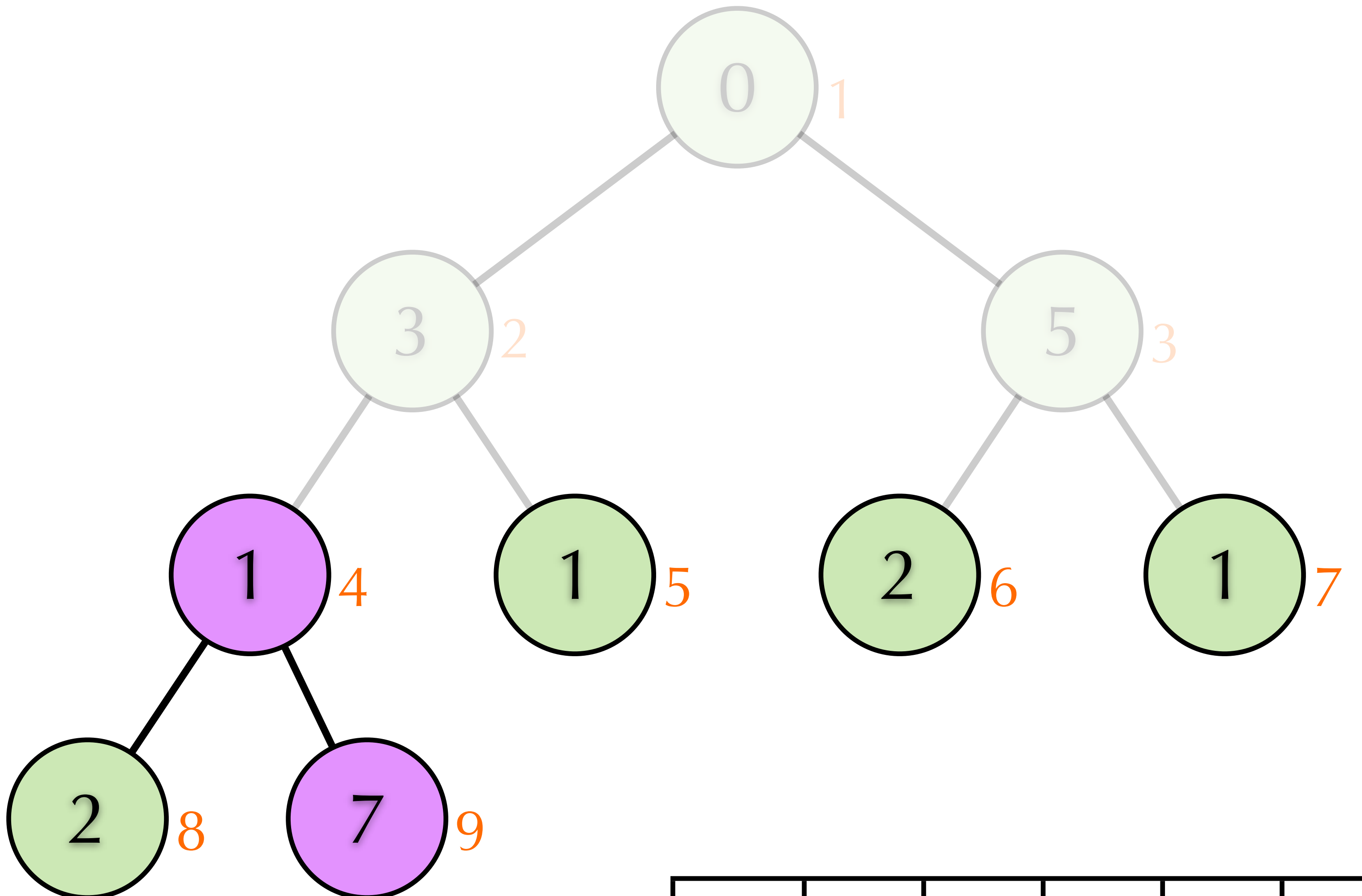

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 1 | 1 | 1 | 3 | 5 | 2 | 7 | 2 |

# Method 2



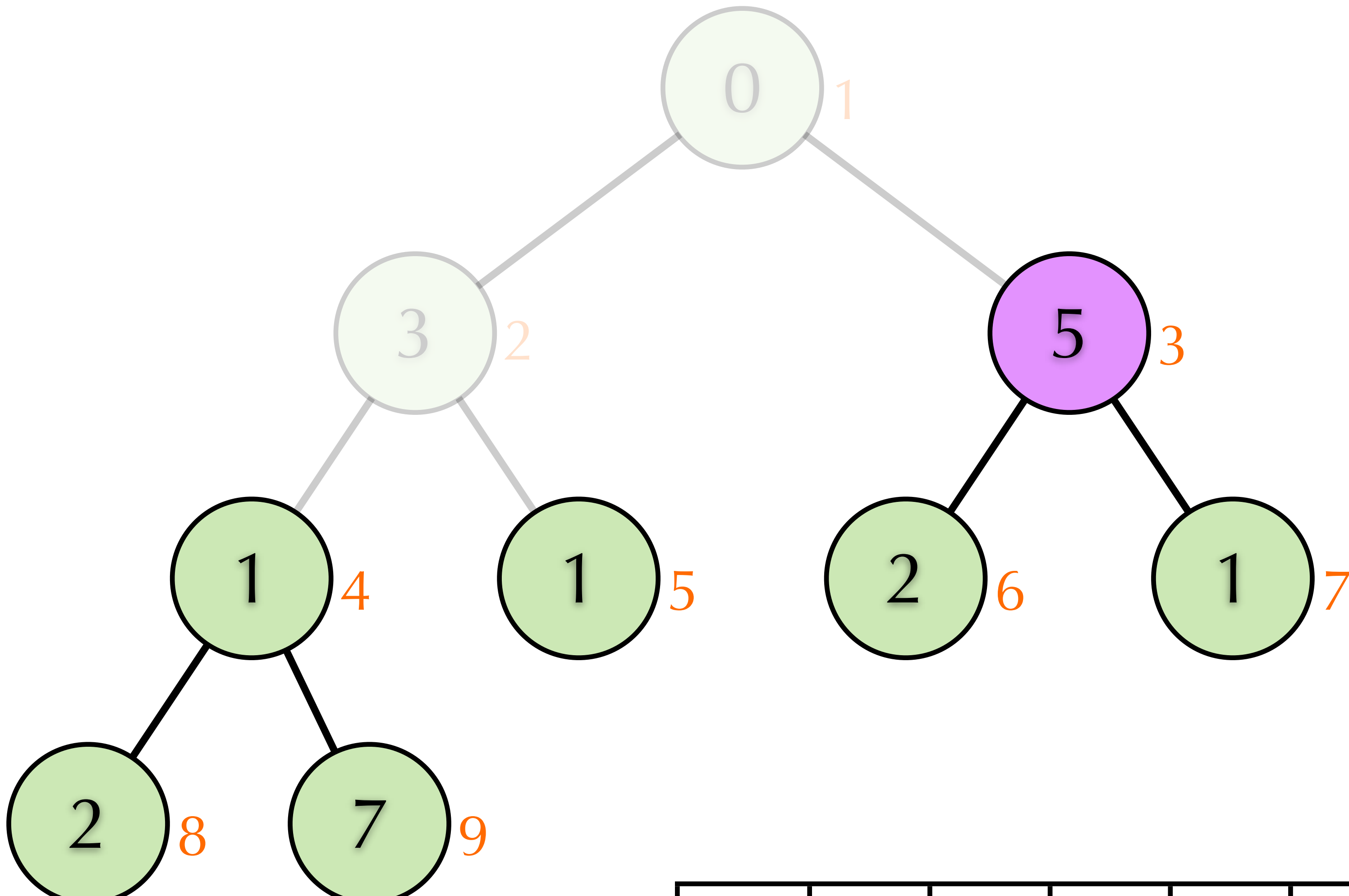

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 3 | 5 | 7 | 1 | 2 | 1 | 2 | 1 |

# Method 2



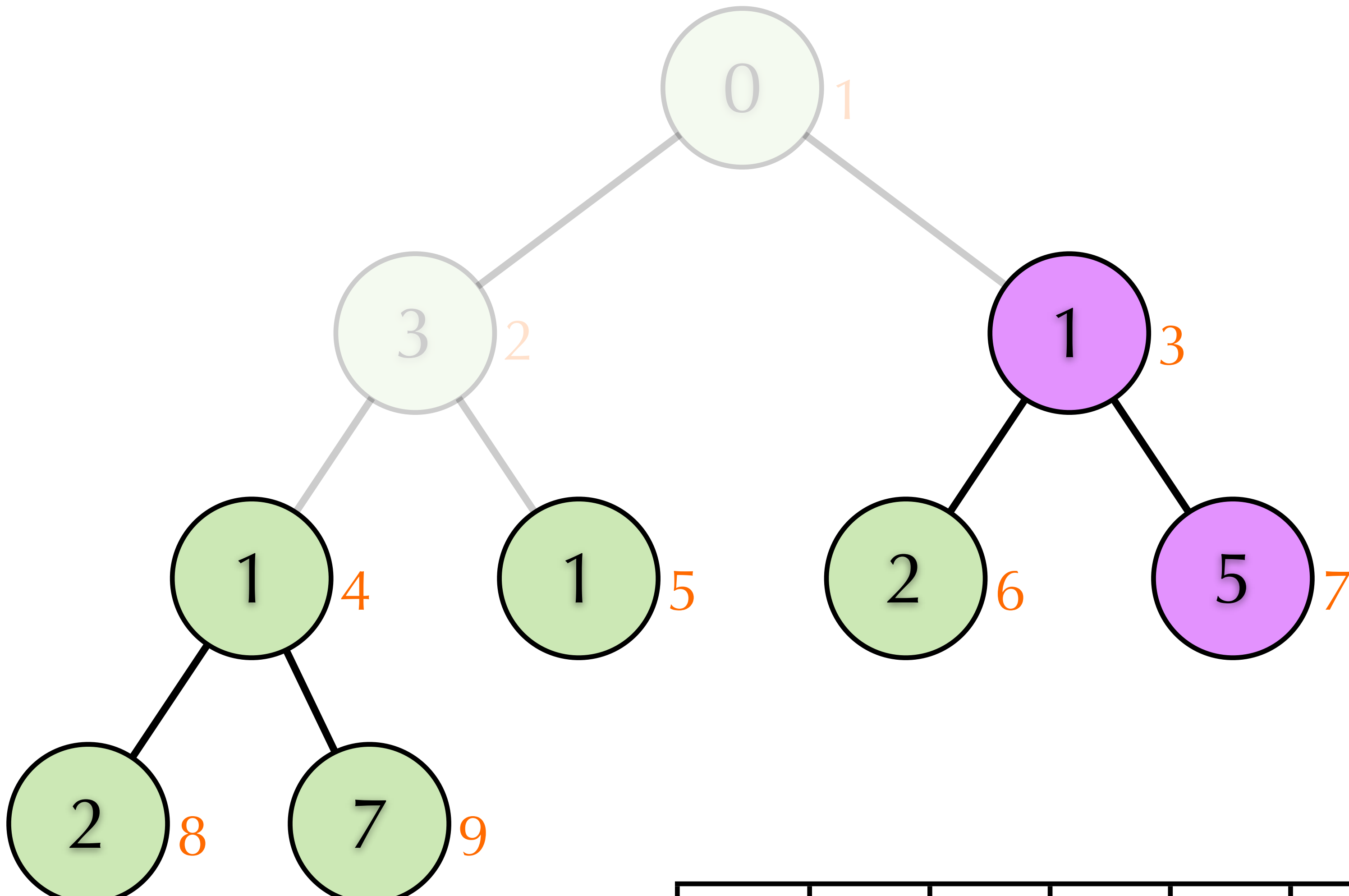

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 3 | 5 | 1 | 1 | 2 | 1 | 2 | 7 |

# Method 2



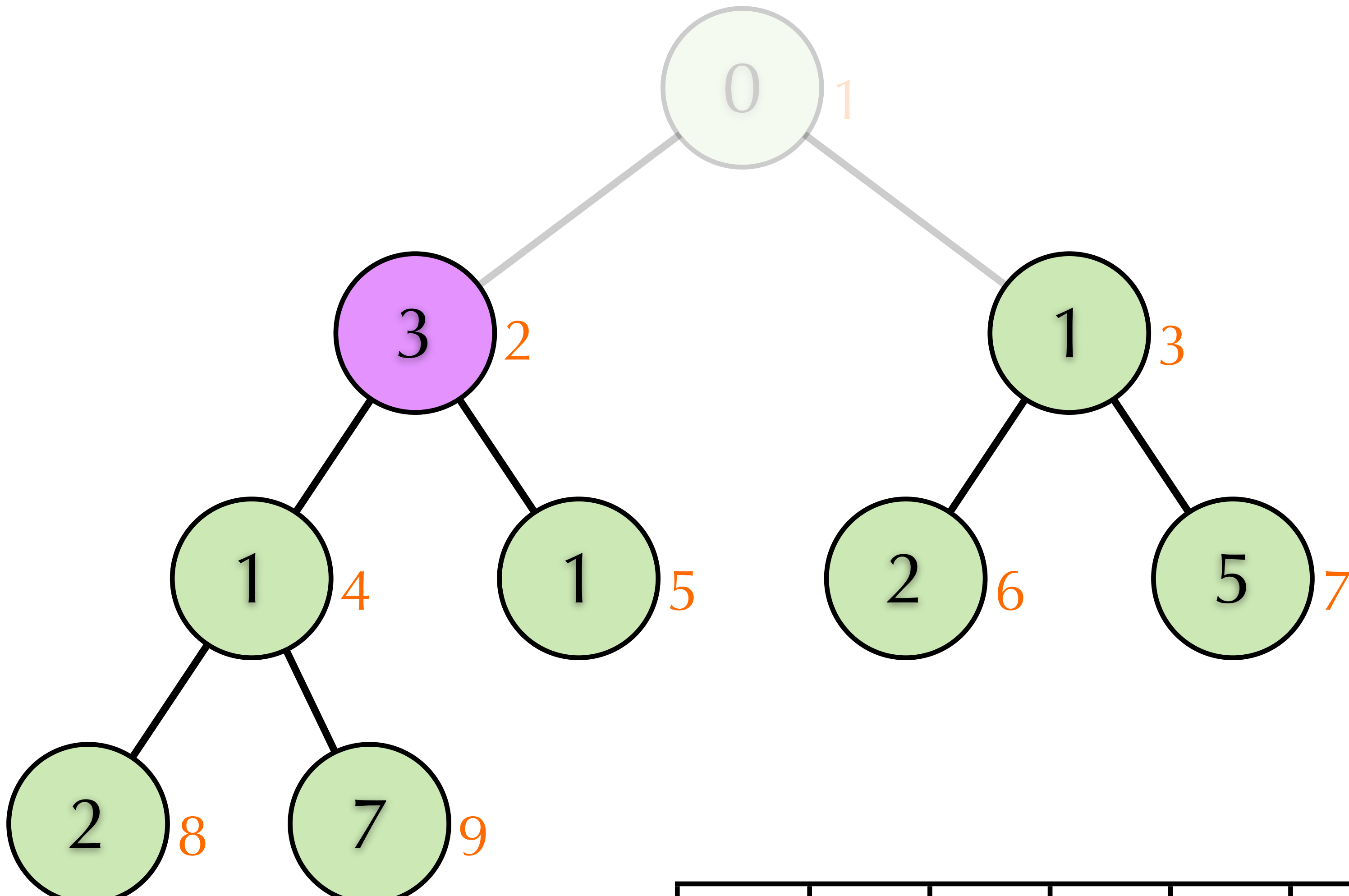

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 3 | 5 | 1 | 1 | 2 | 1 | 2 | 7 |

# Method 2



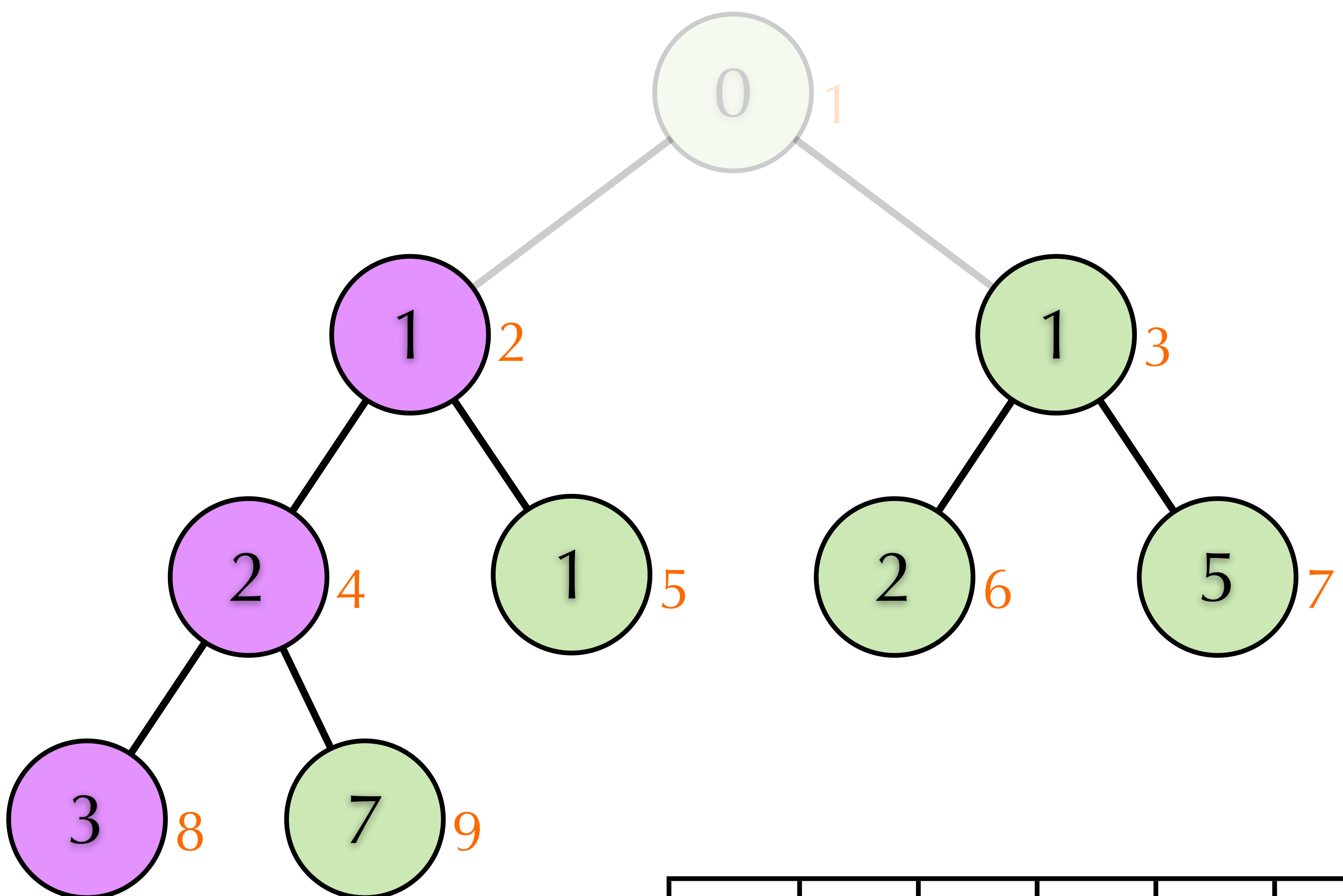

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 3 | 1 | 1 | 1 | 2 | 5 | 2 | 7 |

# Method 2



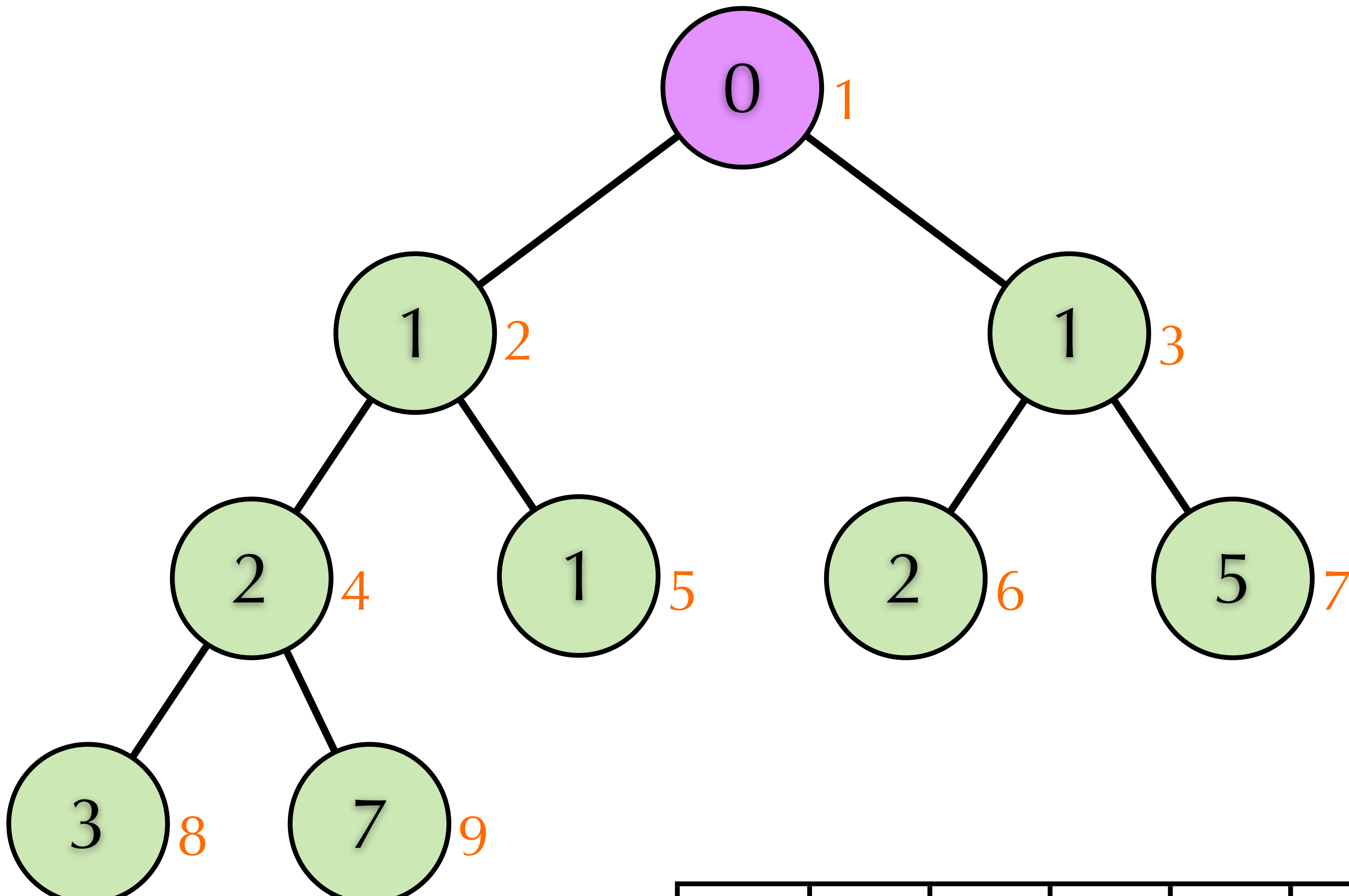

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 3 | 1 | 1 | 1 | 2 | 5 | 2 | 7 |

# Method 2




| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 1 | 1 | 2 | 1 | 2 | 5 | 3 | 7 |

# Method 2




| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[i] | 0 | 1 | 1 | 2 | 1 | 2 | 5 | 3 | 7 |

# Which is Faster?

- Method 1: Repeated insertion
  - 8 key-comparisons
  - 5 key-exchanges
- Method 2:
  - 13 key-comparisons
  - 4 key-exchanges
- What if n is sufficiently large?
- Best/average/worst cases?

# Complexity

Method 1: $\displaystyle\sum_{k=2}^{n} O(\log n) = O(n \log n)$

Method 2: $\displaystyle\sum_{k=1}^{\lfloor n/2 \rfloor} O(\log n - \log k) = O(n)$

Homework 1

# Function Iteration

- Note: in this course, we rarely use this.
- $f^{(i)}(n)=n$ if $i=0$
- $f^{(i)}(n)=f(f^{(i-1)}(n))$ if $i>0$
- Example: $\log^{(2)}n=\log\log n$
- For monotonically increasing function f, the iterated function:
  - $f_c^*(n)=\min\{i\geq 0: f^{(i)}(n)\leq c\}$.

# Iterated Logarithm

- $\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$
- $\lg^* 2 = 1$
- $\lg^* 4 = \lg^* 2^2 = 2$
- $\lg^* 16 = \lg^* 2^4 = 3$
- $\lg^* 65536 = \lg^* 2^{16} = 4$
- $\lg^* 2^{65536} = 5$

# Homework

- 2. Prove that $\max(f(n),g(n))=\Theta(f(n)+g(n))$ if $f(n)\geq 1$ and $g(n)\geq 1$ for $n\geq 1$.
- 3. Is $2^{n+1}=O(2^n)$? Is $3^n=O(2^n)$?
- 4. Is $\lceil \log_2 n \rceil ! = O(n^p)$ for some constant p?
- 5. Is $n!=O(n^n)$? Is $n!=\Omega(n^n)$? Is $\log(n!)=\Theta(\log(n^n))$?
- 6. Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?