COMP 182 Algorithmic Thinking

Weighted Graphs and Greedy Algorithms

Luay Nakhleh
Computer Science
Rice University

Reading Material

- Chapter 10, Section 6
- Chapter 11, Sections 4, 5

Weighted Graphs

- In many real-world applications, graphs have weights associated with their nodes, edges, or both.
- In this set of slides we will focus on weighted graphs where the weights are associated with edges.

Why Weighted Graphs

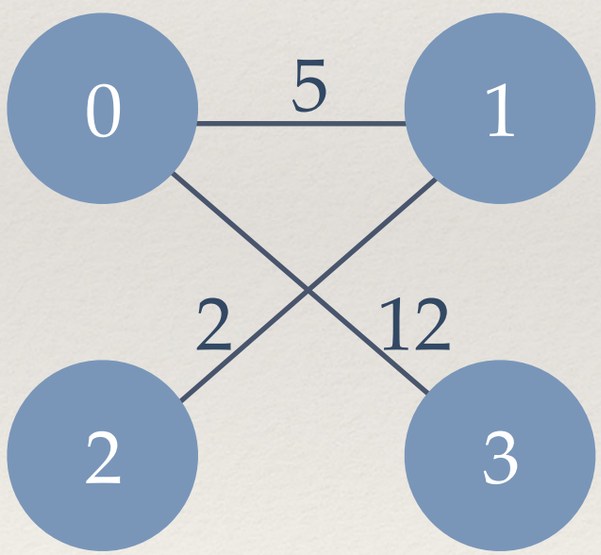
- Internet graph: Edge weights could correspond to the amount of data flowing along them or their bandwidth
- Roadmap graph: Edge weights could correspond to lengths, numbers of gas stations, …
- Food web graph: Edge weights could correspond to the total energy flow between prey and predator
- Social network graph: Edge weights could correspond to the frequency of contact

Weighted Graphs

- A weighted graph is a 3-tuple $g=(V,E,w)$, where V is the set of nodes, E is the set of edges, and $w:E\rightarrow R$ (R is the set of reals) is a function that assigns a weight to each edge.
- The definition applies to both directed and undirected graphs.

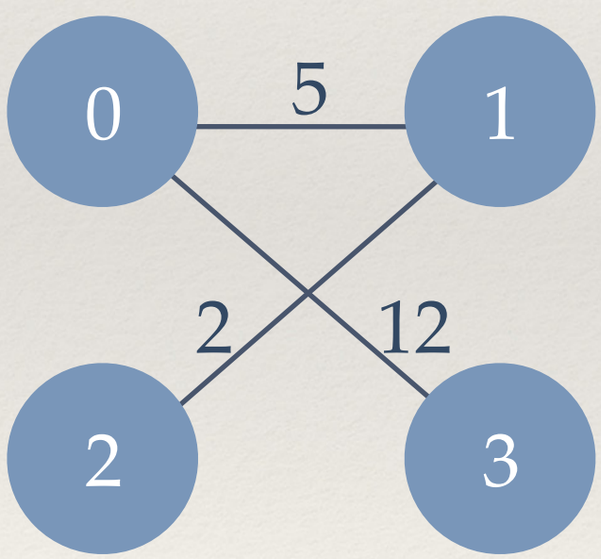
Representing Weighted Graphs

Weighted graph



Representing Weighted Graphs

Weighted graph

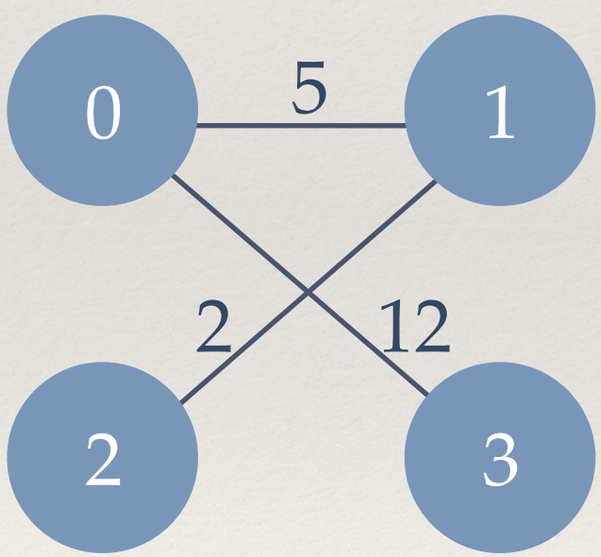


Adjacency matrix

	0	**1**	**2**	**3**
0	∞	5	∞	12
1	5	∞	2	∞
2	∞	2	∞	∞
3	12	∞	∞	∞

Representing Weighted Graphs

Weighted graph



Adjacency matrix

	0	**1**	**2**	**3**
0	∞	5	∞	12
1	5	∞	2	∞
2	∞	2	∞	∞
3	12	∞	∞	∞

Adjacency list

0	→	(1,5)	(3,12)
1	→	(0,5)	(2,2)
2	→	(1,2)	
3	→	(0,12)	

Trees, Spanning Trees, and Minimum Spanning Trees

- A <u>tree</u> is a connected, acyclic graph.
- A <u>spanning tree</u> of a graph $g=(V,E)$ is a connected, acyclic subgraph of g that contains all the nodes in V.
- The weight of a spanning tree of a weighted graph $g=(V,E,w)$ is the sum of the weights of the edges in the tree.
- A <u>minimum spanning tree (MST)</u> of a weighted graph $g=(V,E,w)$ is a spanning tree whose weight is minimum among all possible spanning trees of g.

Paths and Distances

- Paths and connectivity are defined as in the case of unweighted graphs.
- The weight of a path is the sum of the weights of the edges on the path.
- If the weight of an edge corresponds to length, then
  - the <u>length</u> of a path is the sum of the lengths of the edges on the path
  - the <u>distance</u> between two nodes is the length of a shortest path between them

Two Central Problems on Weighted Graphs

- The <u>minimum-spanning tree problem</u>:
  - **Input:** A weighted graph $g=(V,E,w)$.
  - **Output:** The set of edges E_T of an MST of g.
- The <u>single source shortest-paths problem</u>:
  - **Input:** A weighted graph $g=(V,E,w)$ and node $i \in V$.
  - **Output:** The distance d_j from node i for every node $j \in V$, and a shortest path from i to j.

Two Central Problems on Weighted Graphs

- It is easy to come up with brute-force algorithms for these two problems.
- However, these algorithms would be grossly inefficient.
- Efficient algorithms for both problems exist and they use an algorithmic strategy called <u>greedy algorithms</u>.

Greedy Algorithms

- A greedy algorithm constructs a solution to an optimization problem through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.
- In each step, the choice must be
  - feasible: it has to satisfy the problem's constraints
  - locally optimal: it has to be the best local choice for adding the next piece to the solution
  - irrevocable: once made, it is not changed in subsequence steps of the algorithm.

The Minimum Spanning Tree Problem

- **Input:** A weighted graph $g=(V,E,w)$.
- **Output:** The set of edges E_T of an MST of g.

The Minimum Spanning Tree Problem

- Think greedy:
  - feasibility: at every point, the sub-solution must be…
  - local optimality: the algorithm chooses the next edge that …
  - irrevocable: once an edge has been added to E_T, then …

The Minimum Spanning Tree Problem: Prim's Algorithm

Algorithm 2: Prim.

Input: Undirected, weighted graph $g = (V, E, w)$.
Output: E_T, the edges of an MST of g.

```
V_T ← {x};                    // Initialize the nodes of the MST to a randomly chosen nodes x
E_T ← ∅;
for i ← 1 to |V| − 1 do
    Let e = {u, v} be a minimum-weight edge among all edges with one endpoint in V_T and the other in V \ V_T;
    V_T ← V_T ∪ {u, v};
    E_T ← E_T ∪ {{u, v}};
return E_T;
```

The Minimum Spanning Tree Problem: Prim's Algorithm

Algorithm 2: Prim.

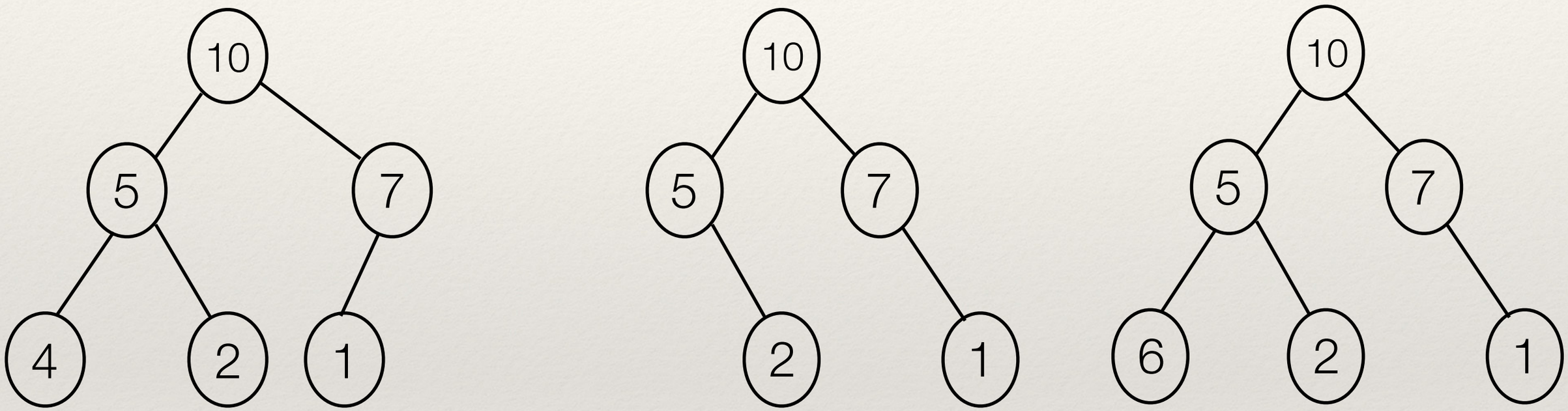
Input: Undirected, weighted graph $g = (V, E, w)$.
Output: E_T, the edges of an MST of g.

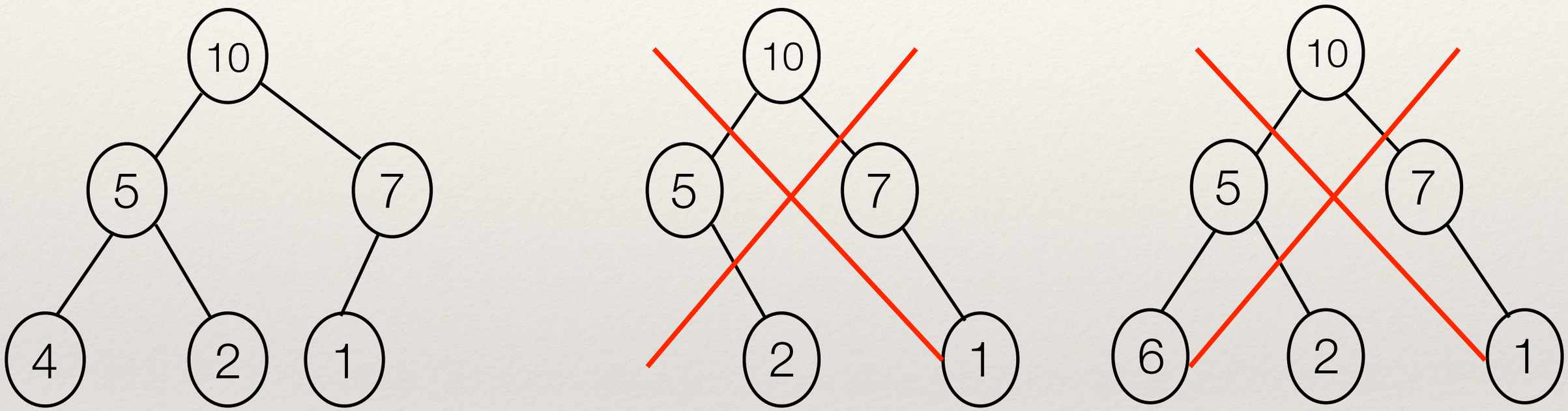
```
V_T ← {x};                    // Initialize the nodes of the MST to a randomly chosen nodes x
E_T ← ∅;
for i ← 1 to |V| − 1 do
    Let e = {u, v} be a minimum-weight edge among all edges with one endpoint in V_T and the other in V \ V_T;
    V_T ← V_T ∪ {u, v};
    E_T ← E_T ∪ {{u, v}};
return E_T;
```

What is the running time of Prim's algorithm?

The Heap Data Structure

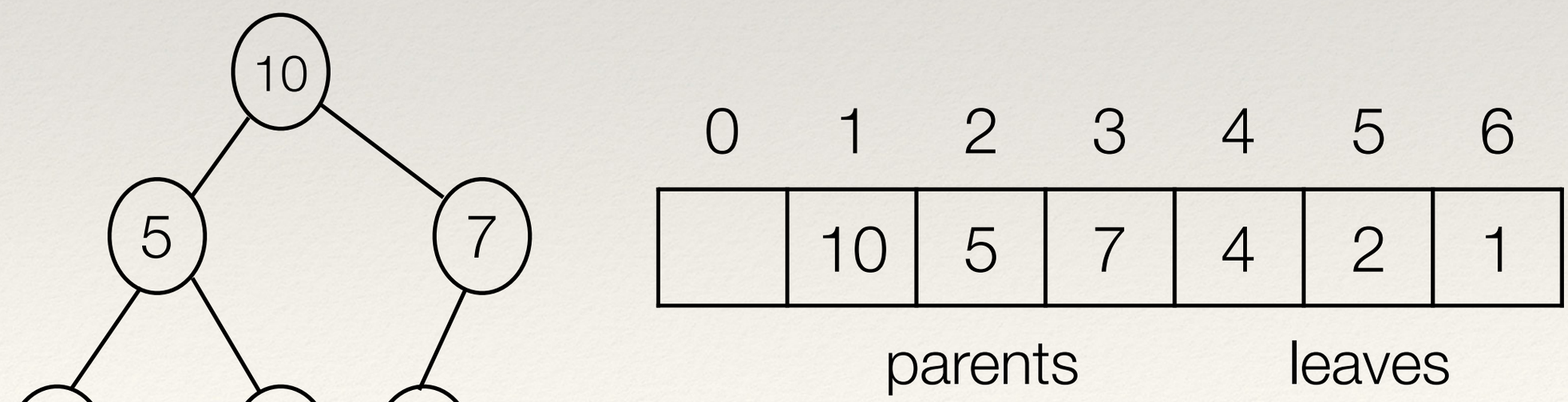
- A <u>heap</u> is a binary tree (every node has 0, 1, or 2 children) with keys assigned to its nodes (one key per node) provided the following two conditions hold:
  - The <u>tree shape requirement</u>: the binary tree is complete, that is, all its levels are full except possibly for the last level, where only some rightmost leaves may be missing.
  - The <u>parental dominance requirement</u>: The key at each node is greater than or equal to the keys of its children.
- A min-heap is one where the parental dominance requirement means the key of a node is smaller than the keys of its children.


10
5
7
4
2
1
10
5
7
2
1
10
5
7
6
2
1


10
5
7
4
2
1
10
5
7
2
1
10
5
7
6
2
1

- The root of a heap always contains its largest element.
- In a heap, a node along with all its descendants is also a heap.
- A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions *1* through *n*, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every key in the heap.

- In the array representation of a heap,
  - the parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions.
  - The children of a key in position i will be in positions $2i$ and $2i+1$

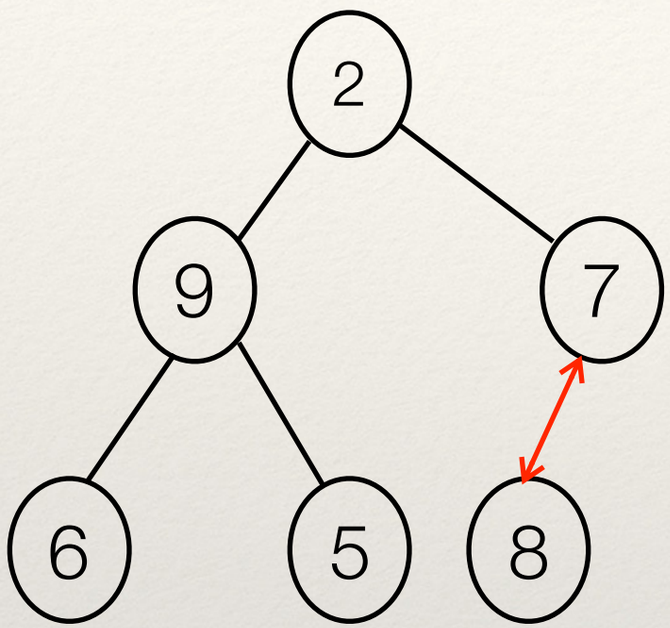


Bottom-up Heap Construction

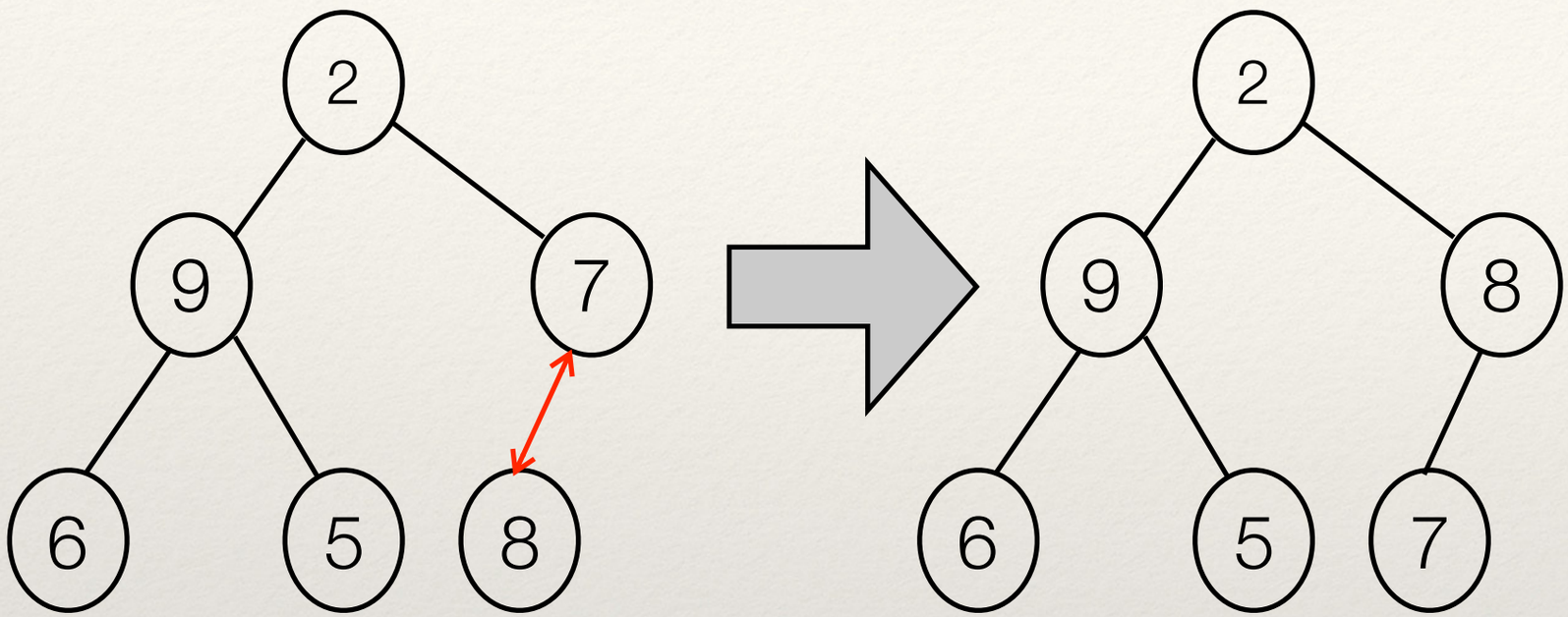
- To construct a heap from a given list of keys, we first initialize the complete binary tree with n nodes by placing keys in the order given, and then "heapify" the tree.

- <u>Heapification:</u> Starting with the last parental node, the "heapify" algorithm checks whether the parental dominance holds for the key at this node. If it does not, the algorithm exchanges the node's key *K* with the larger key of its children and checks whether the parental dominance holds for *K* in its new positions. This process continues until the parental dominance requirement for *K* is satisfied. After completing the heapification of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor. The algorithm stops after this is done for the tree's root.

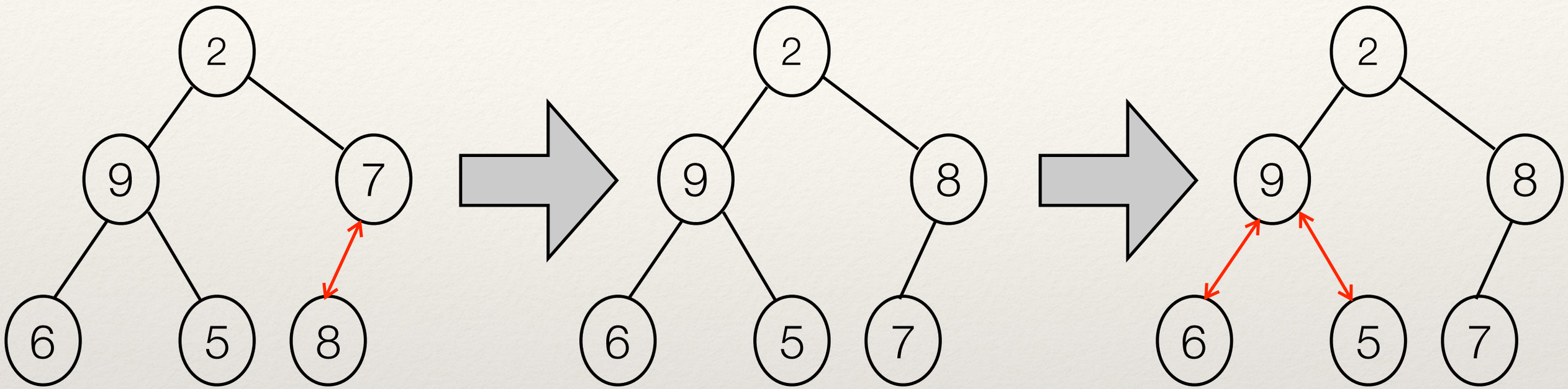
Bottom-up Heap Construction



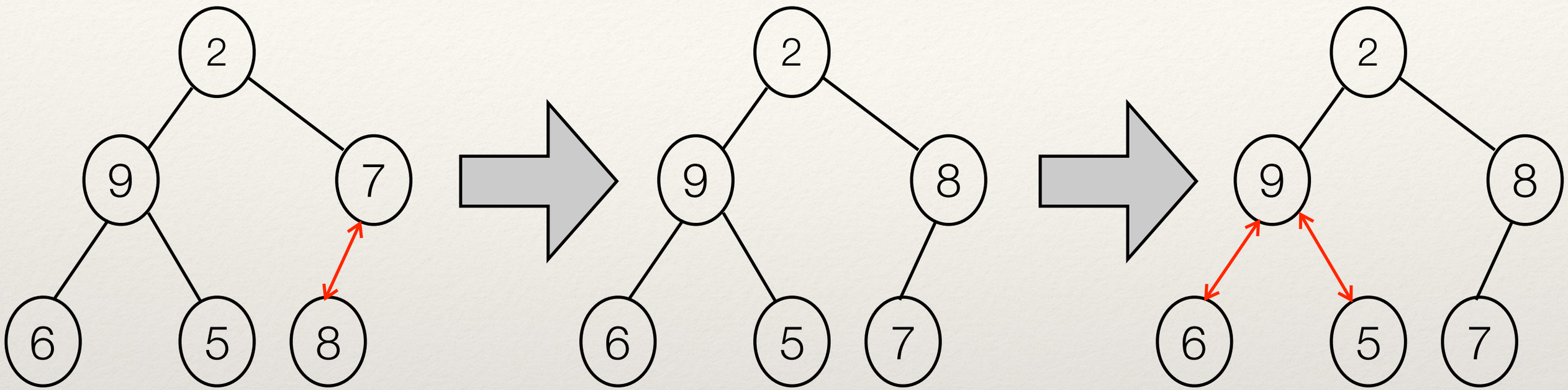
Bottom-up Heap Construction

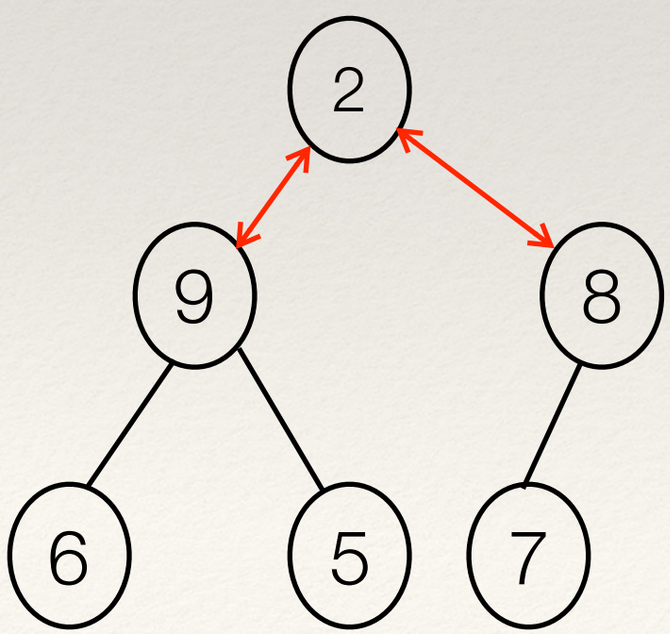


Bottom-up Heap Construction

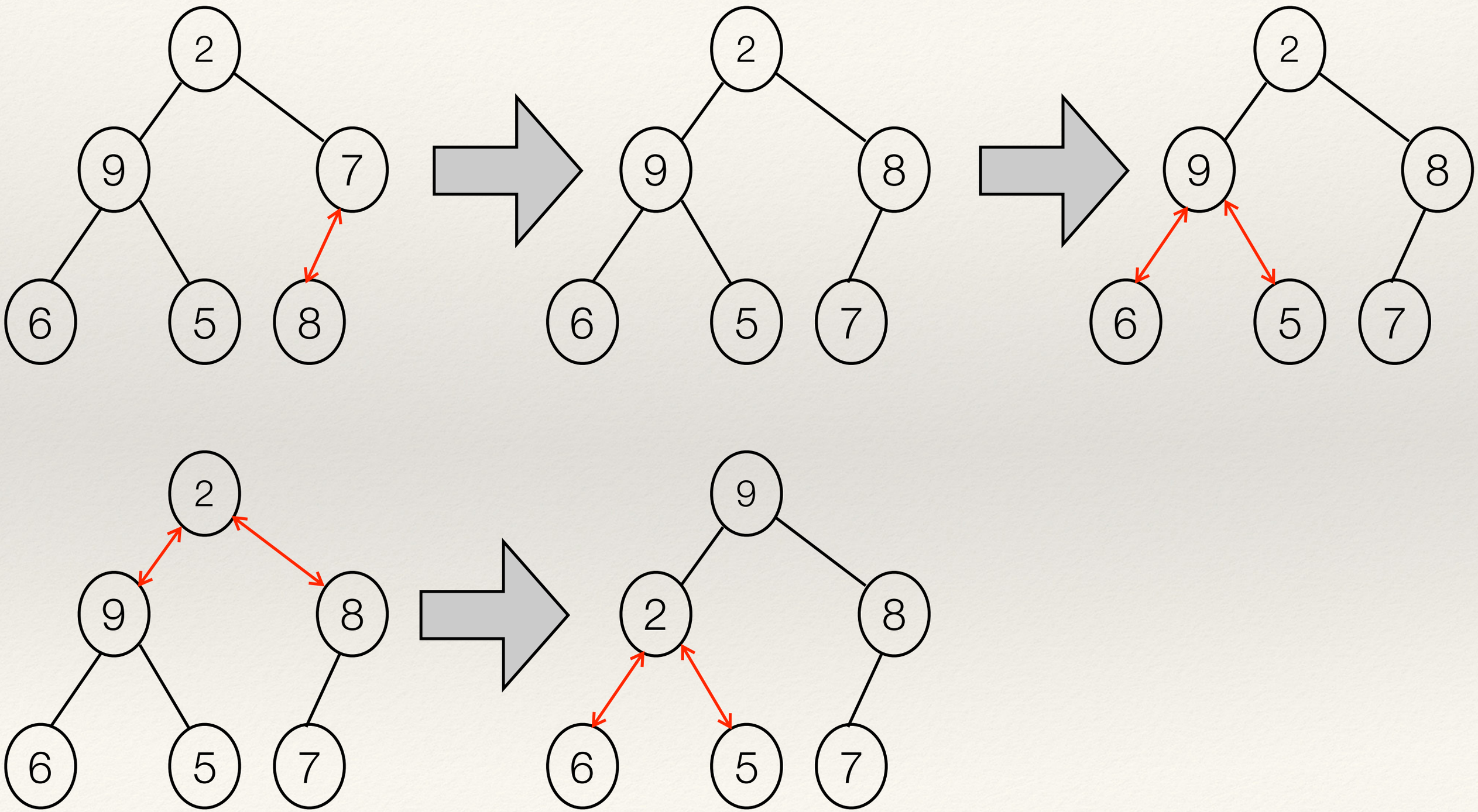


Bottom-up Heap Construction

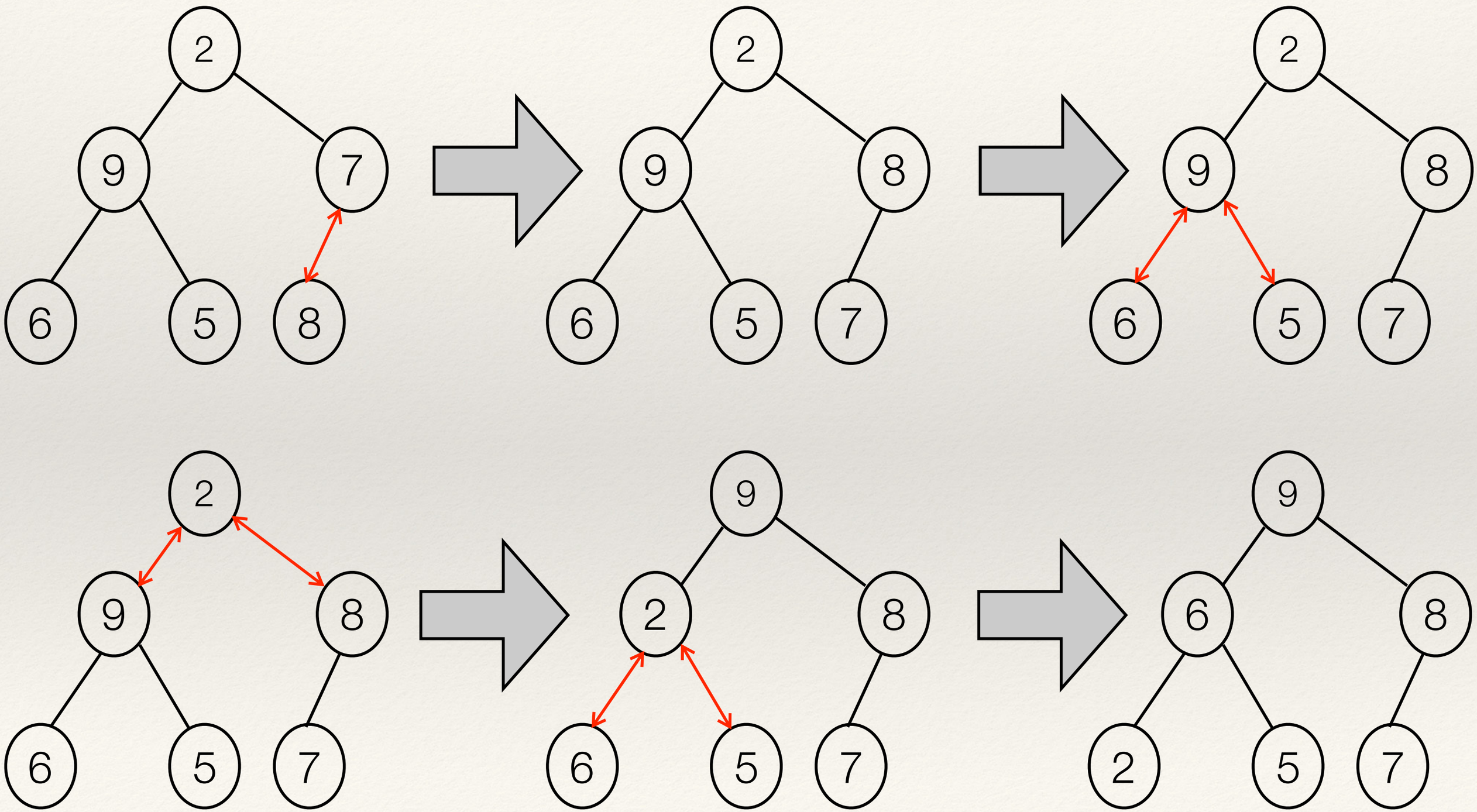




Bottom-up Heap Construction



Bottom-up Heap Construction



Bottom-up Heap Construction: The Pseudo-code

HeapBottomUp
Input: An array H[1..n] of orderable items
Modifies: Array H[1..n] is arranged in-place to reflect the heap properties
Output: None

```
For i ← ⌊n/2⌋ to 1
    k←i; v←H[k]; heap←false;
    While ( (not heap) and 2*k≤n)
        j←2*k;
        If j<n
            If H[j]<H[j+1]
                j←j+1;
        If v≥H[j]
            heap←true;
        Else
            H[k]←H[j]; k←j;
    H[k]←v
```

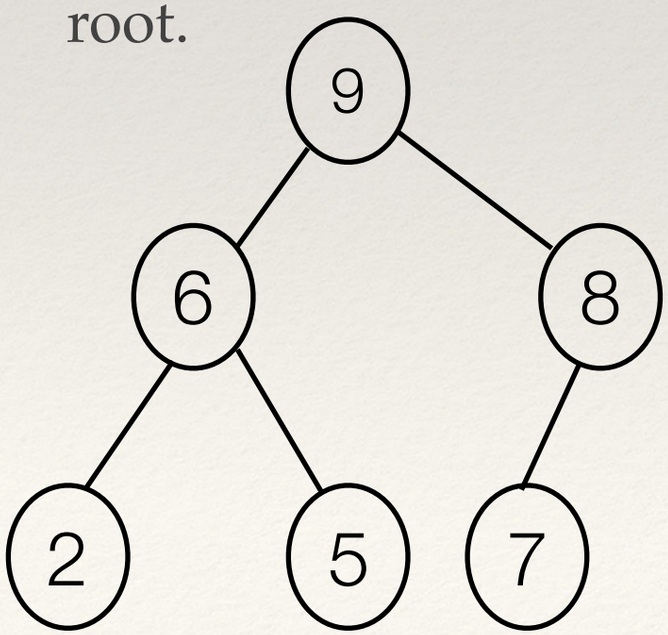
Bottom-up Heap Construction: Computational Complexity

- Assume, for simplicity, that $n=2^k-1$ (for some natural number k).
- Let h be the height of the tree; then, $h = \lfloor log_2\, n \rfloor$.
- In the worst case, each key on level i of the tree will travel to the leaf level h.
- Since moving to the next level down requires two comparisons, the total number of key comparisons involving a key on level i will be $2(h-i)$.
- Therefore, the total number of key comparisons in the worst case will be

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1))$$

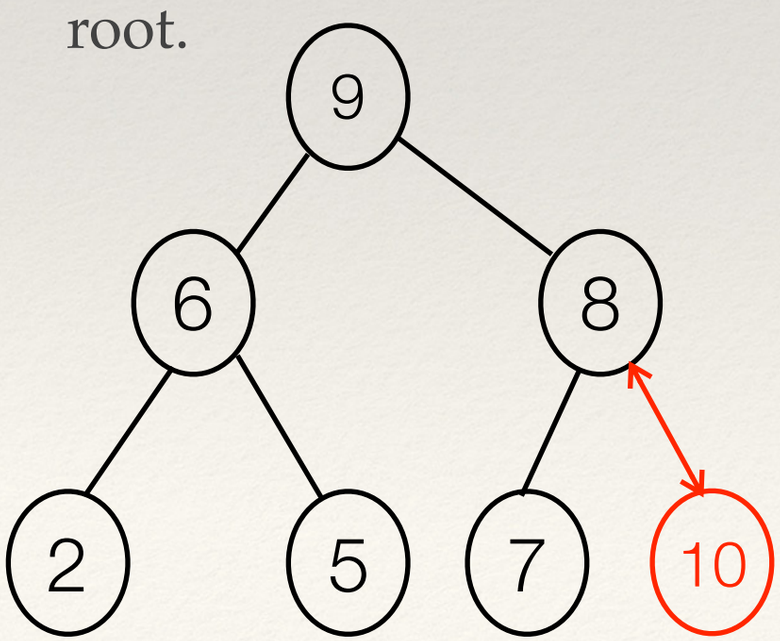
Inserting a Key Into a Heap

- First, attach a new node with key *K* in it after the last leaf of the existing heap.
- Then, sift *K* up to its appropriate place in the new heap as follows.
- Compare *K* with its parent's key: if the parent's key is greater than or equal to *K*, stop; otherwise, swap these two keys and compare *K* with its new parent.
- This swapping continues until *K* is not greater than its last parent or it reaches the root.



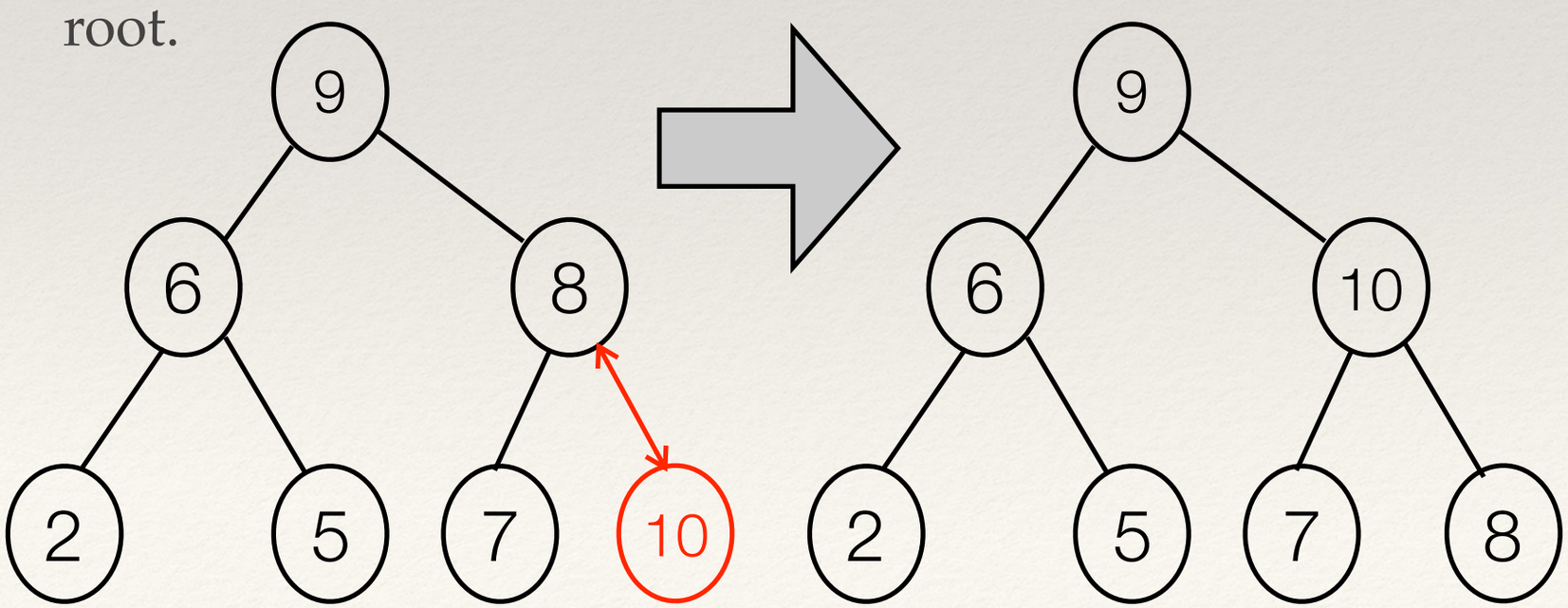
Inserting a Key Into a Heap

- First, attach a new node with key *K* in it after the last leaf of the existing heap.
- Then, sift *K* up to its appropriate place in the new heap as follows.
- Compare *K* with its parent's key: if the parent's key is greater than or equal to *K*, stop; otherwise, swap these two keys and compare *K* with its new parent.
- This swapping continues until *K* is not greater than its last parent or it reaches the root.



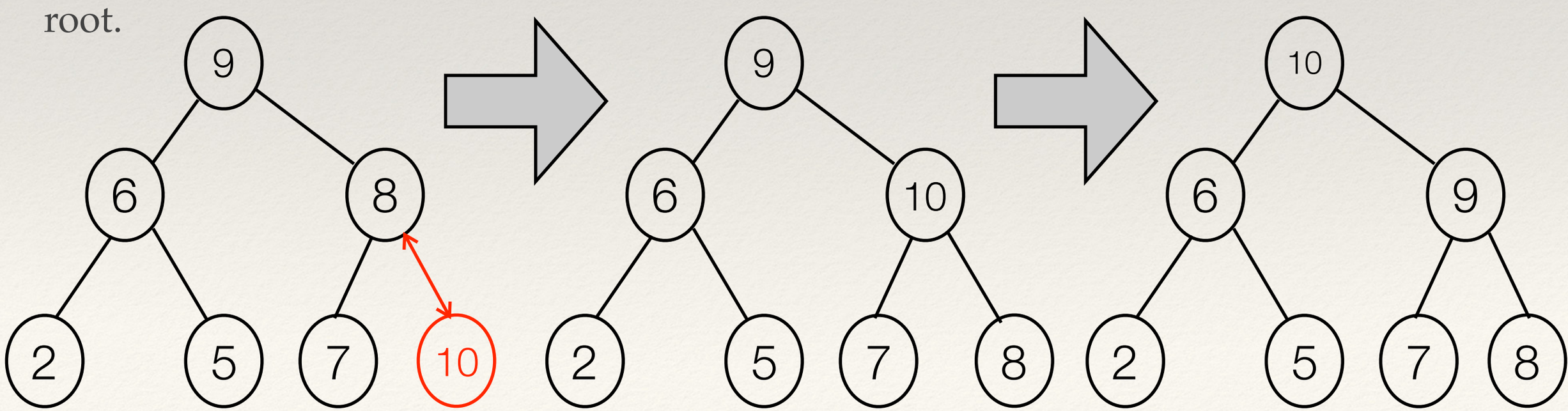
Inserting a Key Into a Heap

- First, attach a new node with key *K* in it after the last leaf of the existing heap.
- Then, sift *K* up to its appropriate place in the new heap as follows.
- Compare *K* with its parent's key: if the parent's key is greater than or equal to *K*, stop; otherwise, swap these two keys and compare *K* with its new parent.
- This swapping continues until *K* is not greater than its last parent or it reaches the root.



Inserting a Key Into a Heap

- First, attach a new node with key *K* in it after the last leaf of the existing heap.
- Then, sift *K* up to its appropriate place in the new heap as follows.
- Compare *K* with its parent's key: if the parent's key is greater than or equal to *K*, stop; otherwise, swap these two keys and compare *K* with its new parent.
- This swapping continues until *K* is not greater than its last parent or it reaches the root.



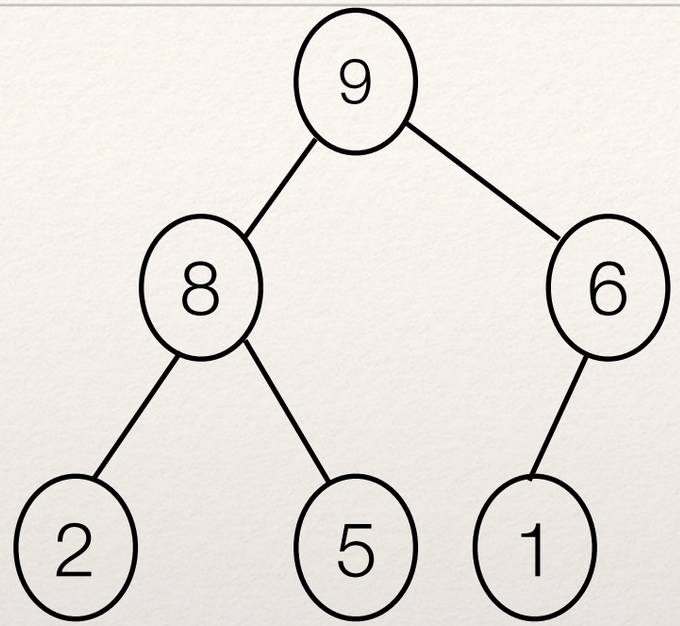
Inserting a Key Into a Heap

- What is the running time?

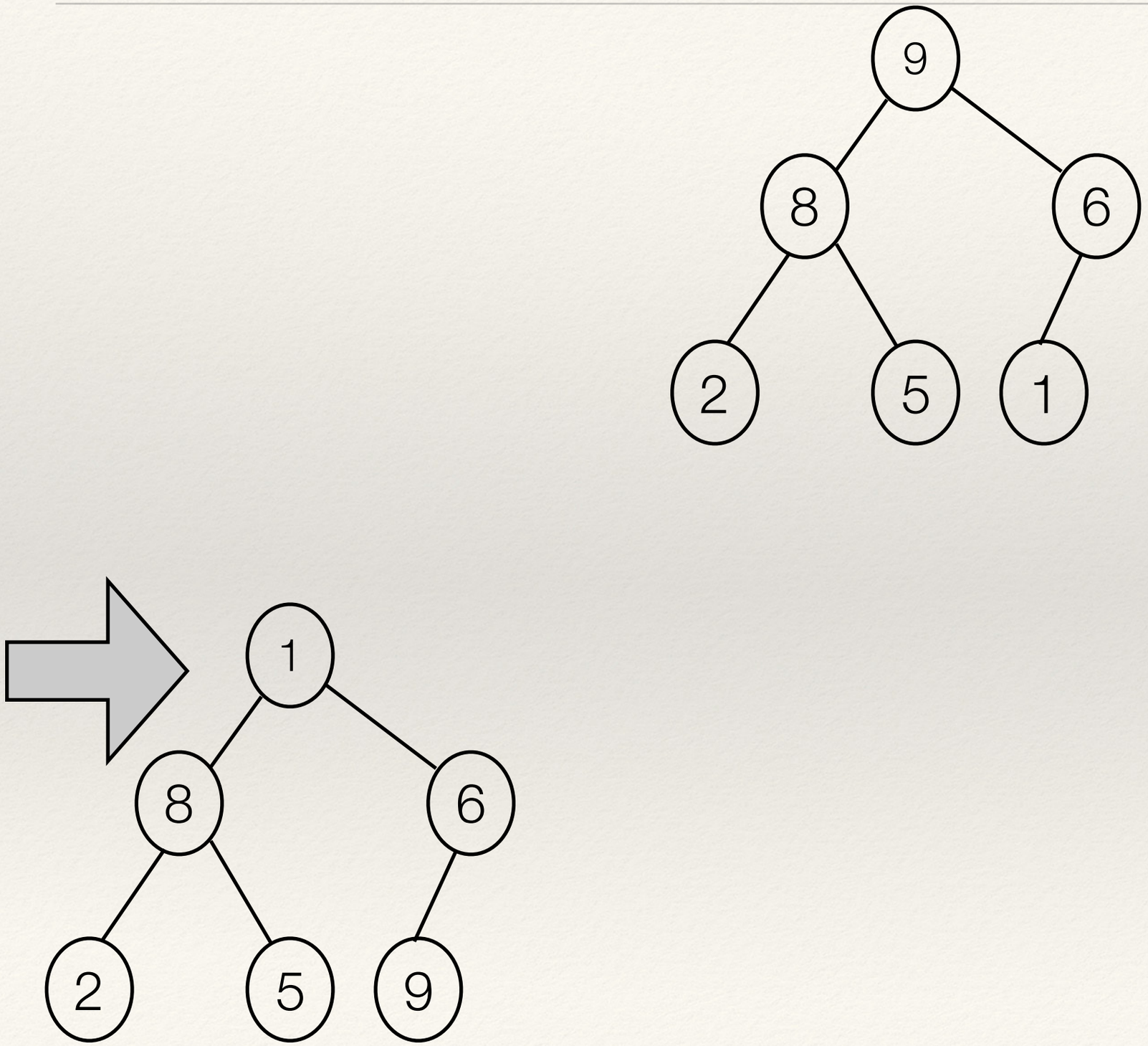
Deleting the Maximum Element From a Heap

- Exchange the root's key with the last key *K* of the heap.
- Decrease the heap's size by *1*.
- "Heapify" the smaller tree by sifting *K* down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for *K*: if it holds, we are done; if not, swap *K* with the larger of its children and repeat this operation until the parental dominance condition holds for *K* in its new position.

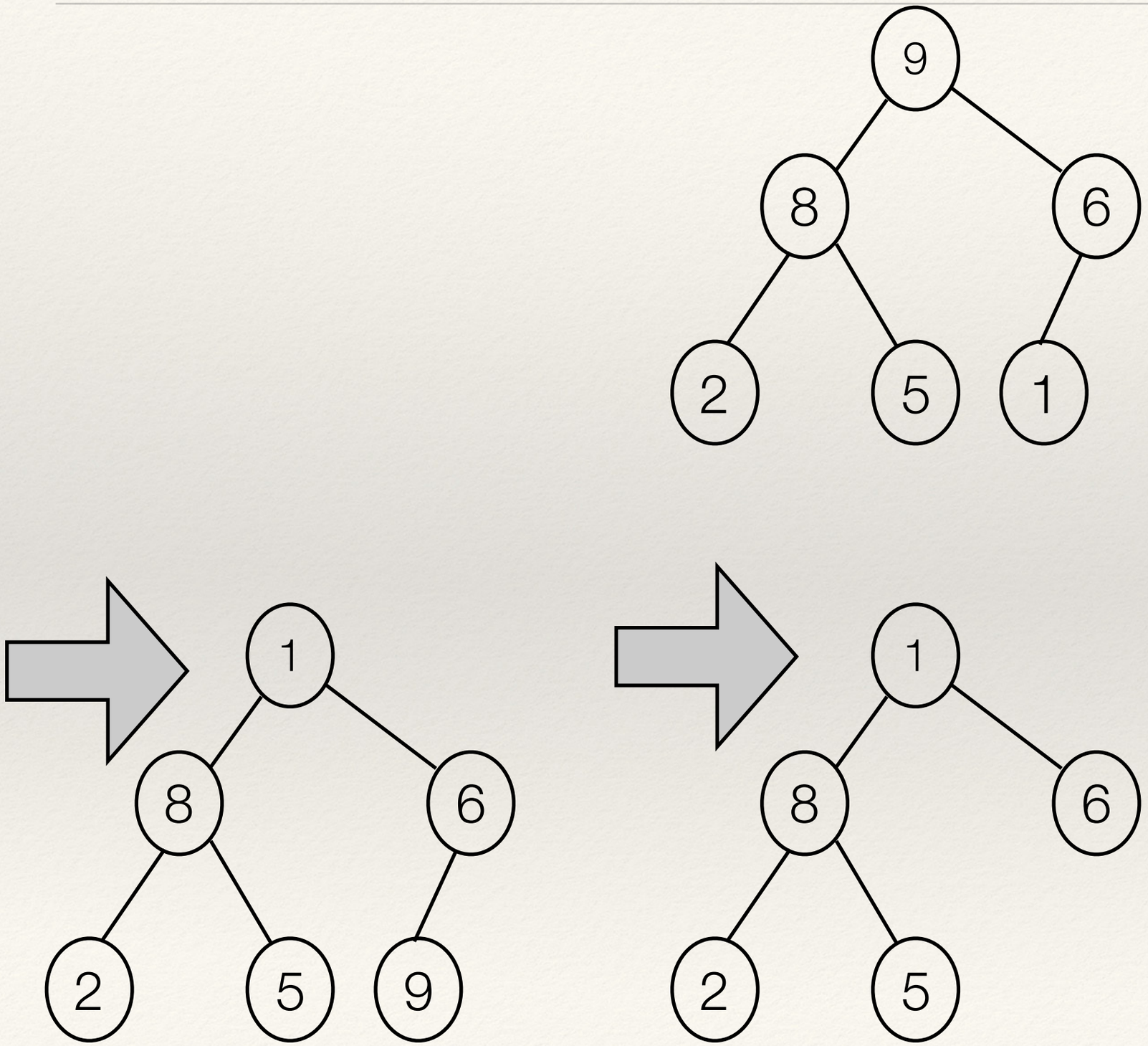
Deleting the Maximum Element From a Heap



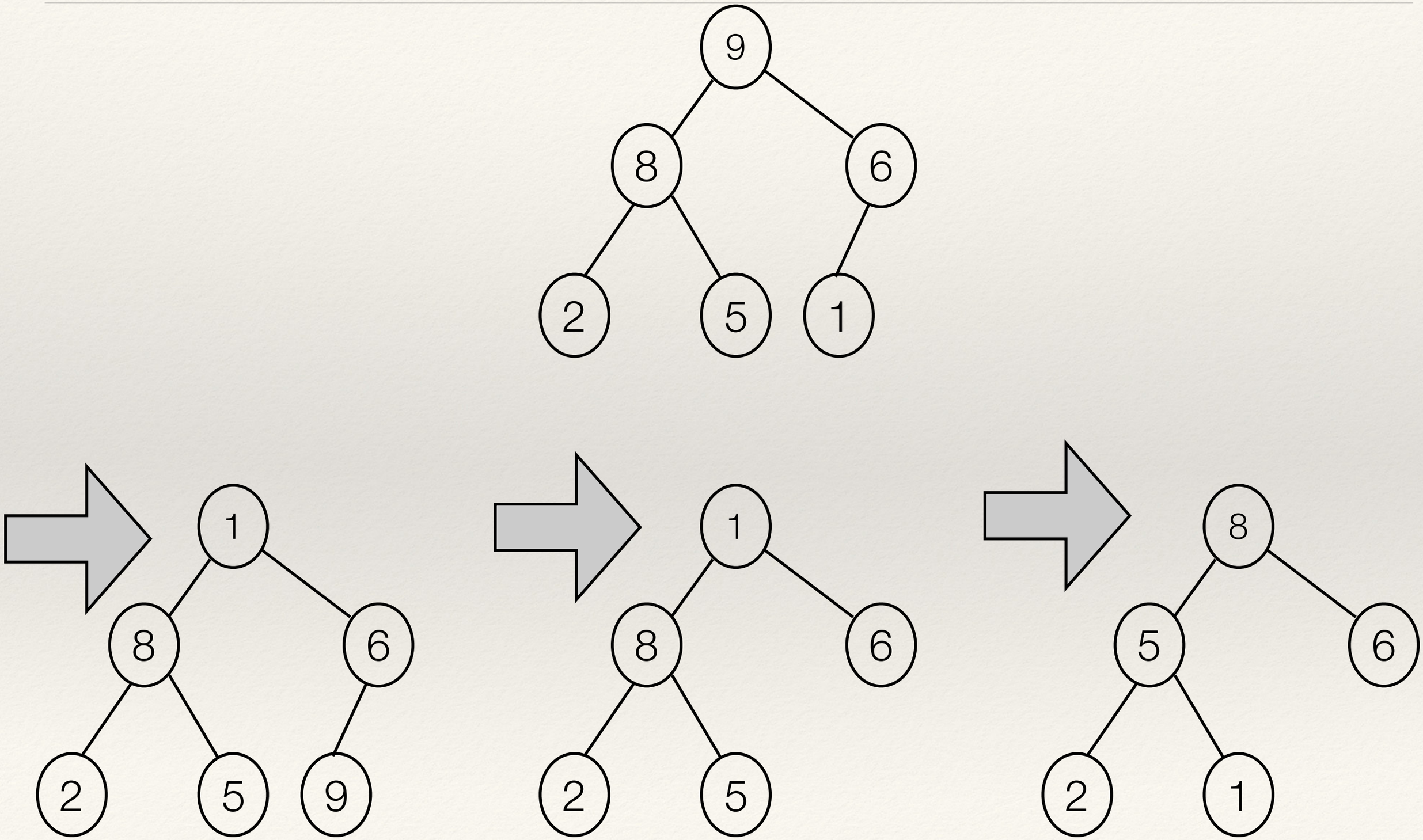
Deleting the Maximum Element From a Heap



Deleting the Maximum Element From a Heap



Deleting the Maximum Element From a Heap



Deleting the Maximum Element From a Heap

- What is the running time?

Back to Weighted Graph Problems...

The Minimum Spanning Tree Problem: Prim's Algorithm

Algorithm 2: Prim.

Input: Undirected, weighted graph $g = (V, E, w)$.
Output: E_T, the edges of an MST of g.

```
V_T ← {x};                    // Initialize the nodes of the MST to a randomly chosen nodes x
E_T ← ∅;
for i ← 1 to |V| − 1 do
    Let e = {u, v} be a minimum-weight edge among all edges with one endpoint in V_T and the other in V \ V_T;
    V_T ← V_T ∪ {u, v};
    E_T ← E_T ∪ {{u, v}};
return E_T;
```

The Minimum Spanning Tree Problem: Prim's Algorithm

Algorithm 2: Prim.

Input: Undirected, weighted graph $g = (V, E, w)$.
Output: E_T, the edges of an MST of g.

```
V_T ← {x};                    // Initialize the nodes of the MST to a randomly chosen nodes x
E_T ← ∅;
for i ← 1 to |V| − 1 do
    Let e = {u, v} be a minimum-weight edge among all edges with one endpoint in V_T and the other in V \ V_T;
    V_T ← V_T ∪ {u, v};
    E_T ← E_T ∪ {{u, v}};
return E_T;
```

What is the running time of Prim's algorithm?

The Minimum Spanning Tree Problem: Kruskal's Algorithm

Algorithm 3: Kruskal.

Input: Undirected, weighted graph $g = (V, E, w)$.
Output: E_T, the edges of an MST of g.
Sort E in non-decreasing order of the weights $w(e_{i_1}) \leq w(e_{i_2}) \leq \cdots \leq w(e_{i_{|E|}})$;
$E_T \leftarrow \emptyset$;
$numedges \leftarrow 0$; `// The number of edges in the growing MST`
$k \leftarrow 0$;
while $numedges < |V| - 1$ **do**
  $k \leftarrow k + 1$;
  if $E_T \cup \{e_{i_k}\}$ *is acyclic* **then**
    $E_T \leftarrow E_T \cup \{e_{i_k}\}$;
    $numedges \leftarrow numedges + 1$;
return E_T;

The Minimum Spanning Tree Problem: Kruskal's Algorithm

Algorithm 3: Kruskal.

Input: Undirected, weighted graph $g = (V, E, w)$.
Output: E_T, the edges of an MST of g.
Sort E in non-decreasing order of the weights $w(e_{i_1}) \leq w(e_{i_2}) \leq \cdots \leq w(e_{i_{|E|}})$;
$E_T \leftarrow \emptyset$;
$numedges \leftarrow 0$; `// The number of edges in the growing MST`
$k \leftarrow 0$;
while $numedges < |V| - 1$ **do**
  $k \leftarrow k + 1$;
  if $E_T \cup \{e_{i_k}\}$ *is acyclic* **then**
    $E_T \leftarrow E_T \cup \{e_{i_k}\}$;
    $numedges \leftarrow numedges + 1$;
return E_T;

What is the running time of Kruskal's algorithm?

The Single-Source Shortest Paths Problem

- **Input:** A weighted graph $g=(V,E,w)$ and node $i \in V$.
- **Output:** The distance d_j from node i for every node $j \in V$, and a shortest path from i to j.

The Single-Source Shortest Paths Problem

- Think greedy:
  - feasibility: at every point, the sub-solution must be…
  - local optimality: the algorithm chooses the next node that …
  - irrevocable: once the distance and shortest path to a node have been set, then …

The Single-Source Shortest Paths Problem:Dijkstra's Algorithm

Algorithm 1: Dijkstra.

Input: Undirected, weighted graph $g = (V, E, w)$; source node i.
Output: Shortest paths, as well as their lengths, from i to every other node in g.

```
X ← ∅;
foreach j ∈ V do
    d_j ← ∞ ;                    // d_j is the geodesic distance between i and j
    p_j ← null ;                 // p_j carries the node label of j's parent
    X ← X ∪ {j} ;                // Set X contains all nodes not visited yet
d_i ← 0;
while X ≠ ∅ do
    Let k be a node with the minimum value of d_k in the set X;
    if d_k = ∞ then
        break;
    X ← X \ {k};
    foreach neighbor h of k in the set X do
        if d_k + w((k, h)) < d_h then
            d_h ← d_k + w((k, h));
            p_h ← k;
return p, d;
```

The Single-Source Shortest Paths Problem:Dijkstra's Algorithm

Algorithm 1: Dijkstra.

Input: Undirected, weighted graph $g = (V, E, w)$; source node i.
Output: Shortest paths, as well as their lengths, from i to every other node in g.

```
X ← ∅;
foreach j ∈ V do
    d_j ← ∞ ;                    // d_j is the geodesic distance between i and j
    p_j ← null ;                 // p_j carries the node label of j's parent
    X ← X ∪ {j} ;                // Set X contains all nodes not visited yet
d_i ← 0;
while X ≠ ∅ do
    Let k be a node with the minimum value of d_k in the set X;
    if d_k = ∞ then
        break;
    X ← X \ {k};
    foreach neighbor h of k in the set X do
        if d_k + w((k, h)) < d_h then
            d_h ← d_k + w((k, h));
            p_h ← k;
return p, d;
```

What is the running time of Dijkstra's algorithm?

Questions?