Heapsort

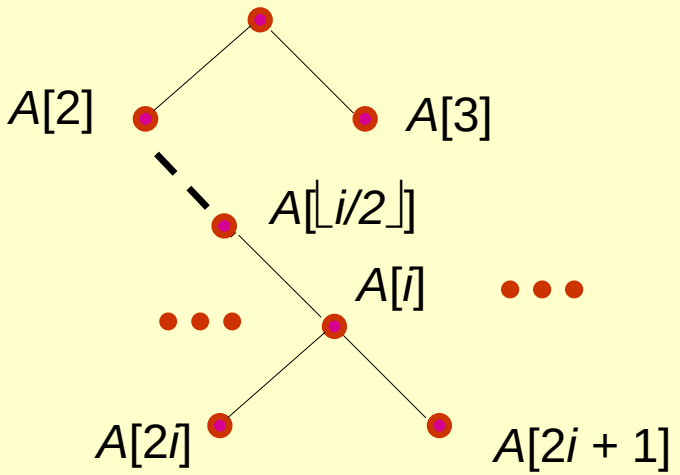
- What is a heap? Max-heap? Min-heap?
- Maintenance of Max-heaps
  - *MaxHeapify*
  - *BuildMaxHeap*
- Heapsort
  - Heapsort
  - Analysis
- Priority queues
  - Maintenance of priority queues

Heapsort

- Combines the better attributes of merge sort and quick sort.
  - » Like merge sort, but unlike quick sort, running time is $O(n \lg n)$.
  - » Like quick sort, but unlike merge sort, sorts in place.
- Introduces an algorithm design technique
  - » Create data structure (*heap*) to manage information during the execution of an algorithm.
- The *heap* has other applications beside sorting.
  - » Priority Queues

Data Structure Binary Heap

- Array viewed as a nearly complete binary tree.
  - Physically – linear array.
  - Logically – binary tree, filled on all levels (except lowest.)
- Map from array elements to tree nodes and vice versa
  - Root – $A[1]$, Left[Root] – $A[2]$, Right[Root] – $A[3]$
  - Left[i] – $A[2i]$
  - Right[i] – $A[2i+1]$
  - Parent[i] – $A[\lfloor i/2 \rfloor]$

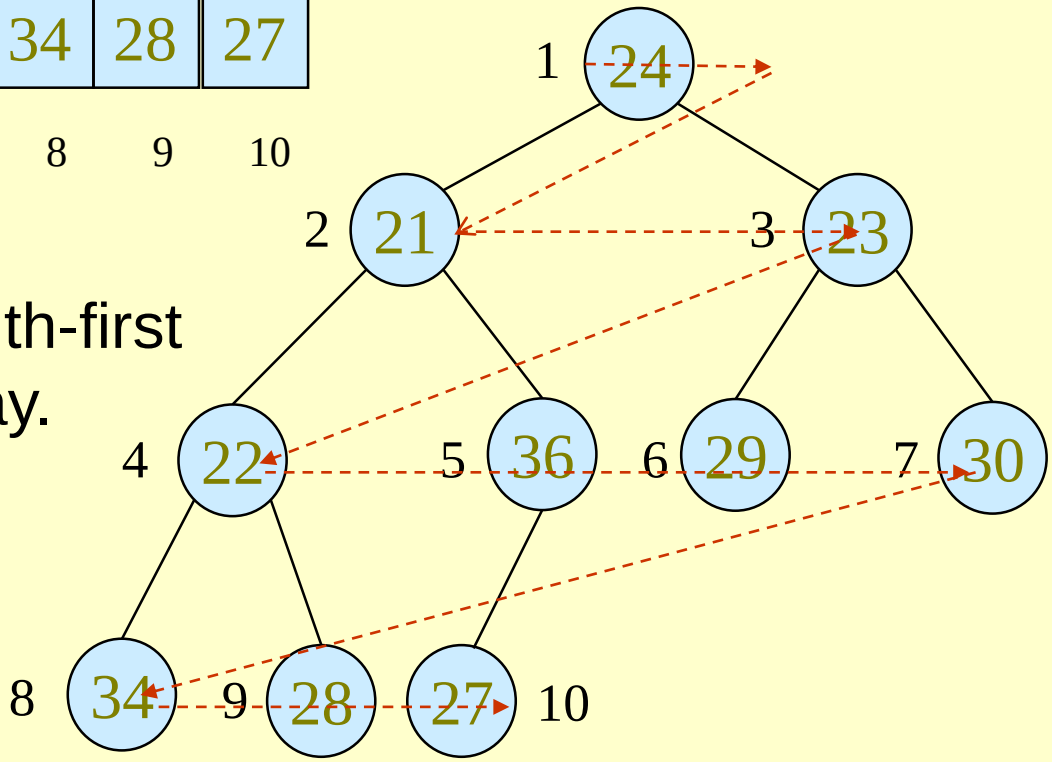


Data Structure Binary Heap

- length[*A*] – number of elements in array *A*.
- heap-size[*A*] – number of elements in heap stored in *A*.
  - » heap-size[*A*] ≤ length[*A*]

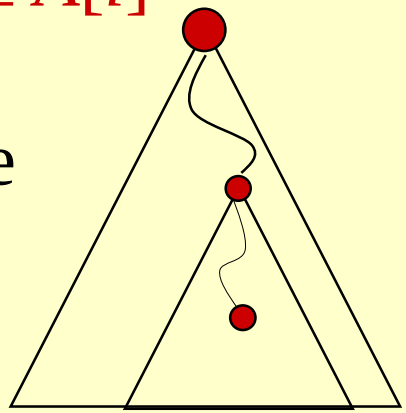
24	21	23	22	36	29	30	34	28	27
1	2	3	4	5	6	7	8	9	10

Searching the tree in breadth-first fashion, we will get the array.



Heap Property (Max and Min)

- Max-Heap
  - » For every node excluding the root, the value stored in that node is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree's root.



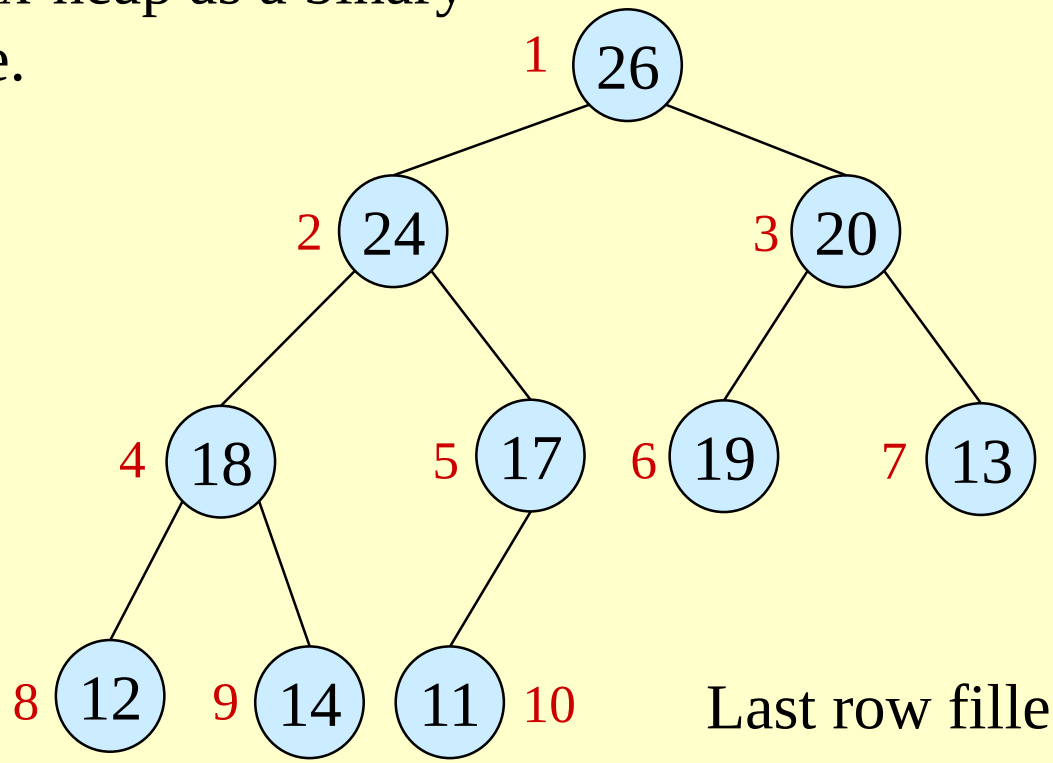
- Min-Heap
  - » For every node excluding the root, the value stored in that node is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at subtree's root

Heaps – Example

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10

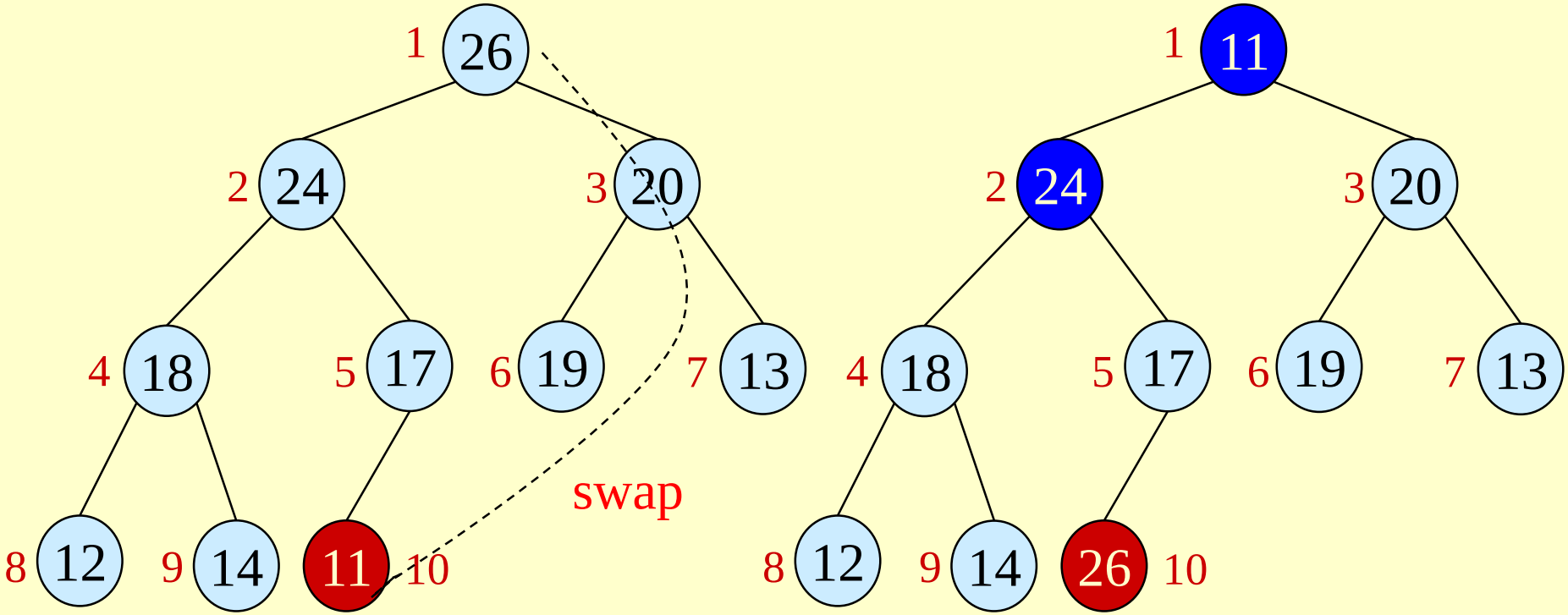
Max-heap as an array.

Max-heap as a binary tree.



Last row filled from left to right.

No longer a Max-heap:



Heaps in Sorting

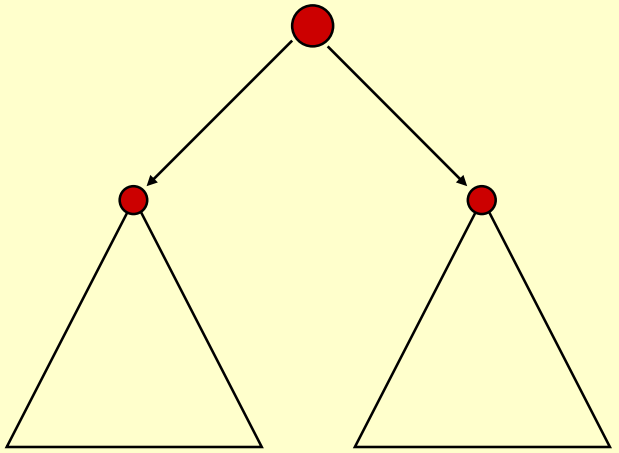
- Use max-heaps for sorting.
- The array representation of a max-heap is not sorted.
- Steps in sorting

  i) Convert a given array of size n to a max-heap (*BuildMaxHeap*)

  ii) Swap the first and last elements of the array.
    - Now, the largest element is in the last position – where it belongs.
    - That leaves $n - 1$ elements to be placed in their appropriate locations.
    - However, the array of first $n - 1$ elements is no longer a max-heap.
    - Float the element at the root down one of its subtrees so that the array remains a max-heap (*MaxHeapify*)
    - Repeat step (ii) until the array is sorted.

Maintaining the heap property

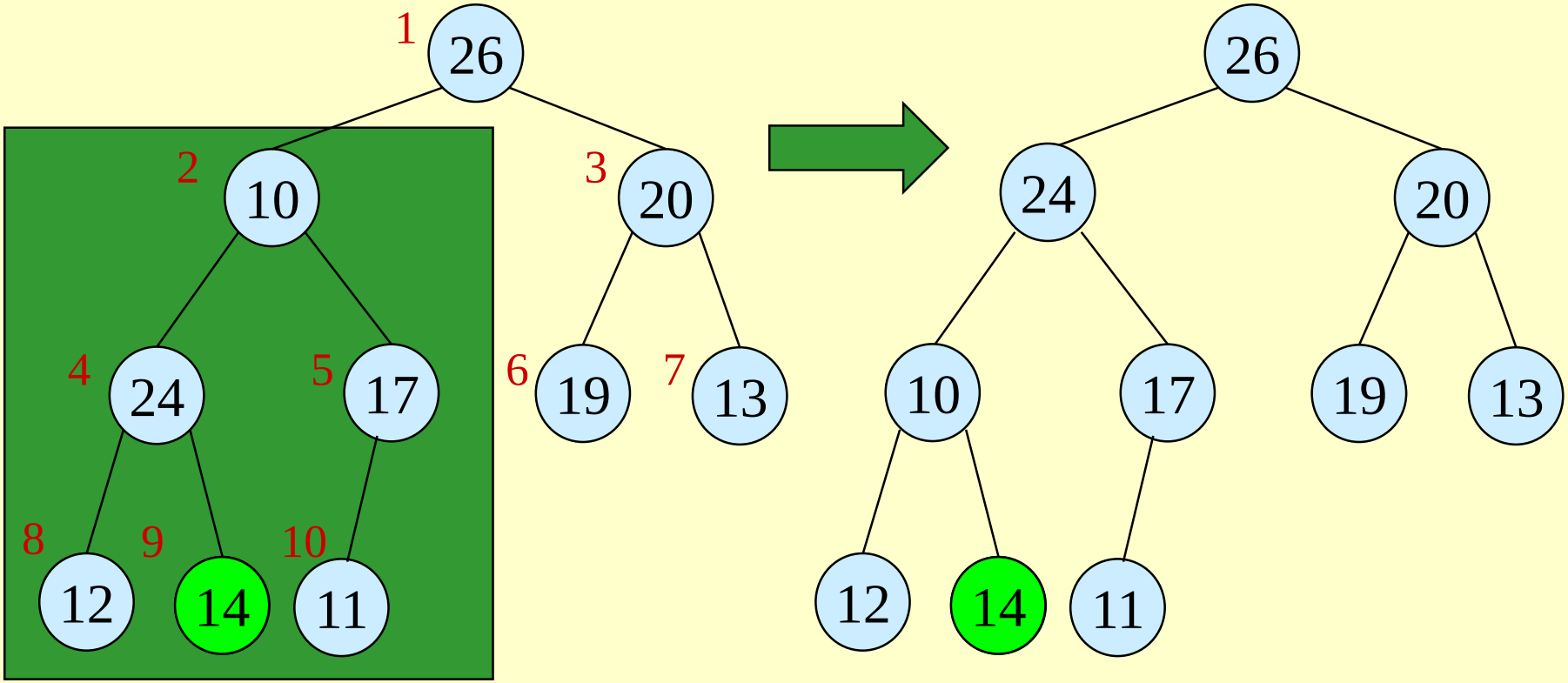
- Suppose two subtrees are max-heaps, but the root violates the max-heap property.

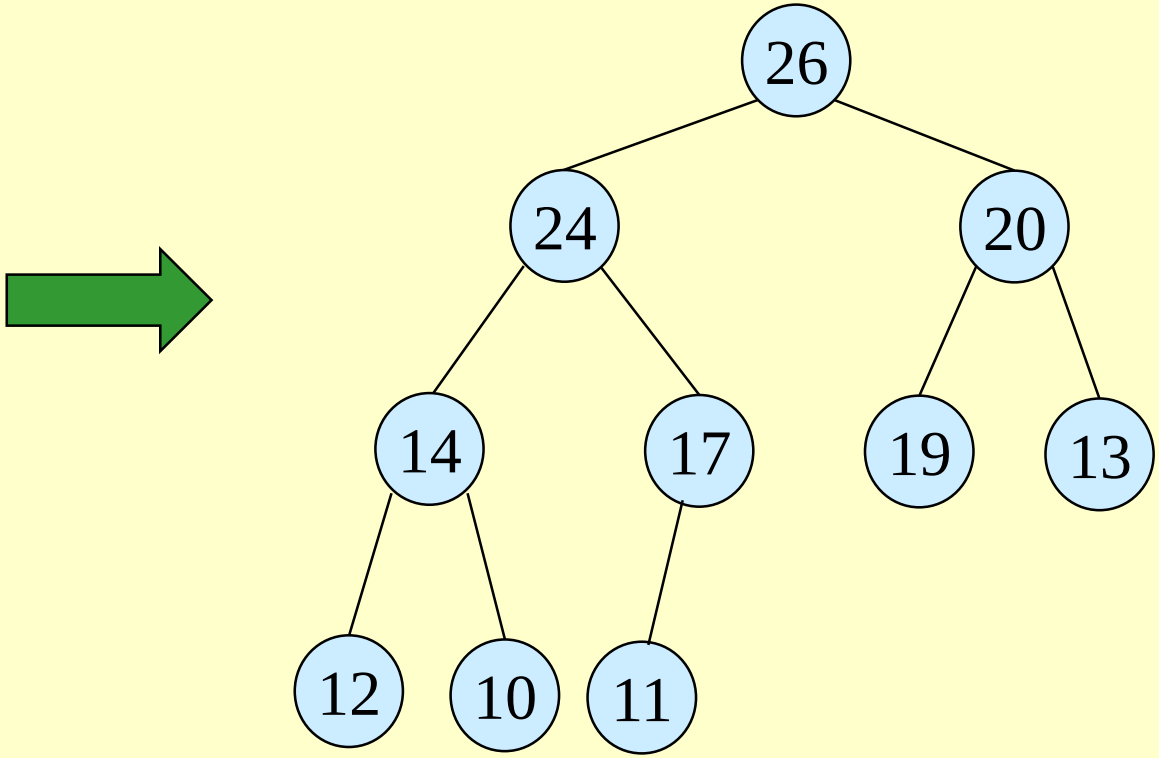


- Fix the offending node by exchanging the value at the node with the larger of the values at its children.
  - » May lead to the subtree at the child not being a max heap.
- Recursively fix the children until all of them satisfy the max-heap property.

MaxHeapify – Example

MaxHeapify(*A*, 2)



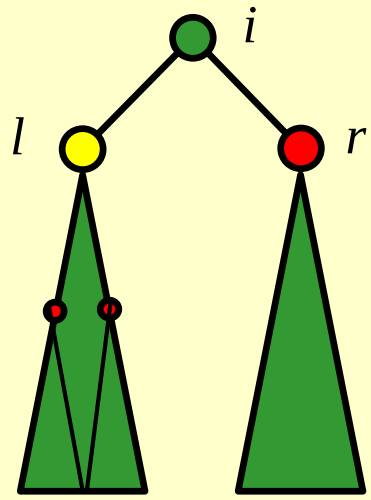

26
24
20
14
17
19
13
12
10
11

Procedure MaxHeapify

MaxHeapify(*A*, *i*)

1. $l \leftarrow$ left(i) (* $A[l]$ is the left child of $A[i]$.*)
2. $r \leftarrow$ right(i)
3. **if** $l \leq$ *heap-size*[A] and $A[l] > A[i]$
4. **then** *largest* $\leftarrow l$
5. **else** *largest* $\leftarrow i$
6. **if** $r \leq$ *heap-size*[A] **and** $A[r] > A[$*largest*$]$
7. **then** *largest* $\leftarrow r$
8. **if** *largest* $\neq i$
9. **then** exchange $A[i] \leftrightarrow A[$*largest*$]$
10. *MaxHeapify*(*A*, *largest*)

Assumption:
Left(i) and Right(i) are max-heaps.



A[*largest*] must be the largest among *A*[*i*], *A*[*l*] and *A*[*r*].

Running Time for MaxHeapify

MaxHeapify(*A*, *i*)

1. $l \leftarrow$ left(i)
2. $r \leftarrow$ right(i)
3. **if** $l \leq$ *heap-size*[A] and $A[l] > A[i]$
4. **then** *largest* $\leftarrow l$
5. **else** *largest* $\leftarrow i$
6. **if** $r \leq$ *heap-size*[A] **and** $A[r] > A[$*largest*$]$
7. **then** *largest* $\leftarrow r$
8. **if** *largest* $\neq i$
9. **then** exchange $A[i] \leftrightarrow A[$*largest*$]$
10. *MaxHeapify*(*A*, *largest*)

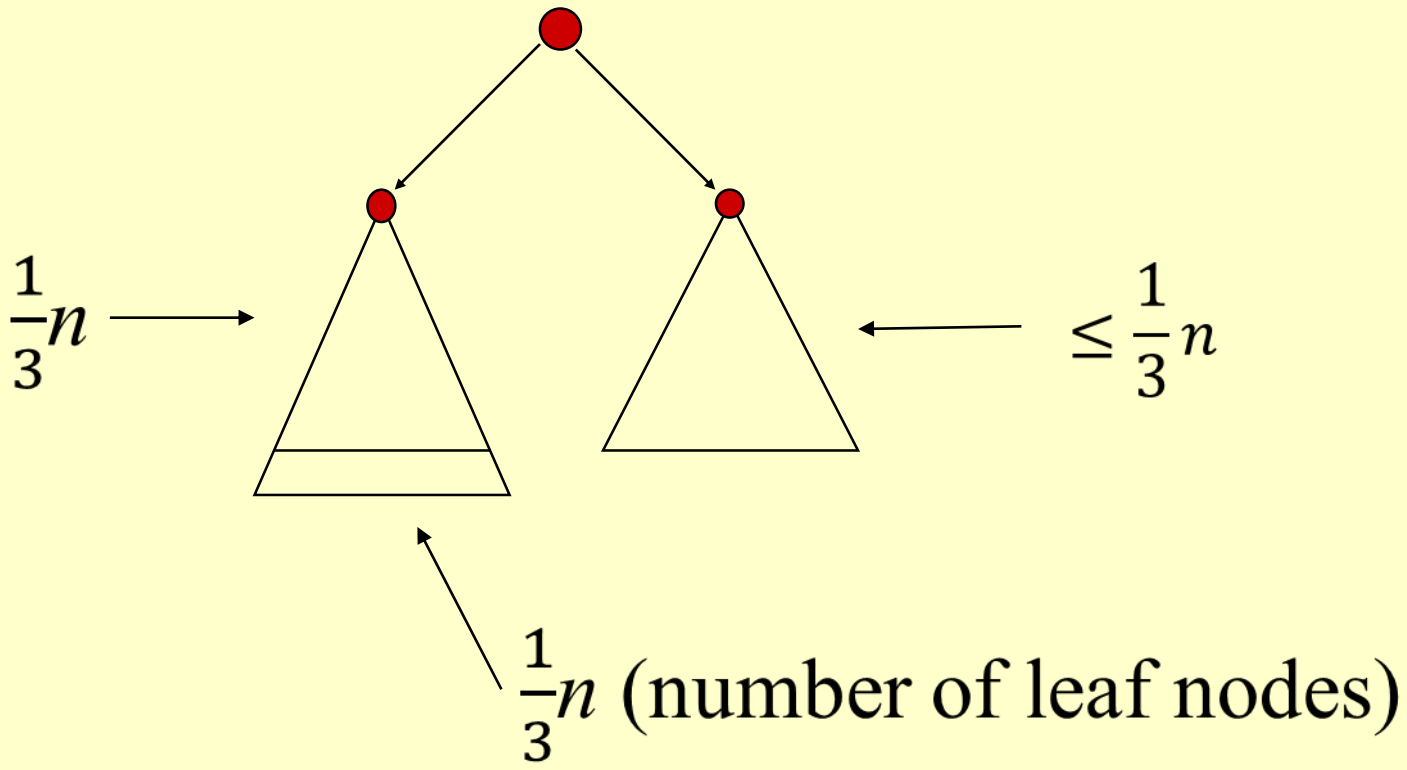
Time to fix node *i* and its children = **Θ(1)**

PLUS

Time to fix the subtree rooted at one of *i*'s children = *T*(size of subree at *largest*)

Running Time for MaxHeapify(A, n)

- $T(n) = T(\text{size of subree at } \textit{largest}) + \Theta(1)$
- size of subree at *largest* $\leq 2n/3$ (worst case occurs when the last row of tree is exactly half full)
- $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\log_{1.5} n)$
- Alternately, MaxHeapify takes $O(h)$ where h is the height of the node where MaxHeapify is applied


$\frac{1}{3}n$
$\leq \frac{1}{3}n$
$\frac{1}{3}n$ (number of leaf nodes)

Building a heap

- Use *MaxHeapify* to convert an array *A* into a max-heap.
- How?
- Call MaxHeapify on each element in a bottom-up manner.

BuildMaxHeap(*A*)

1. *heap-size*[*A*] ← *length*[*A*]
2. **for** *i* ← $\lfloor$*length*[*A*]/2$\rfloor$ **downto** 1 (**A*[$\lfloor$*length*[*A*]/2$\rfloor$ +1],
3. **do** *MaxHeapify*(*A*, *i*) *A*[$\lfloor$*length*[*A*]/2$\rfloor$ +2],

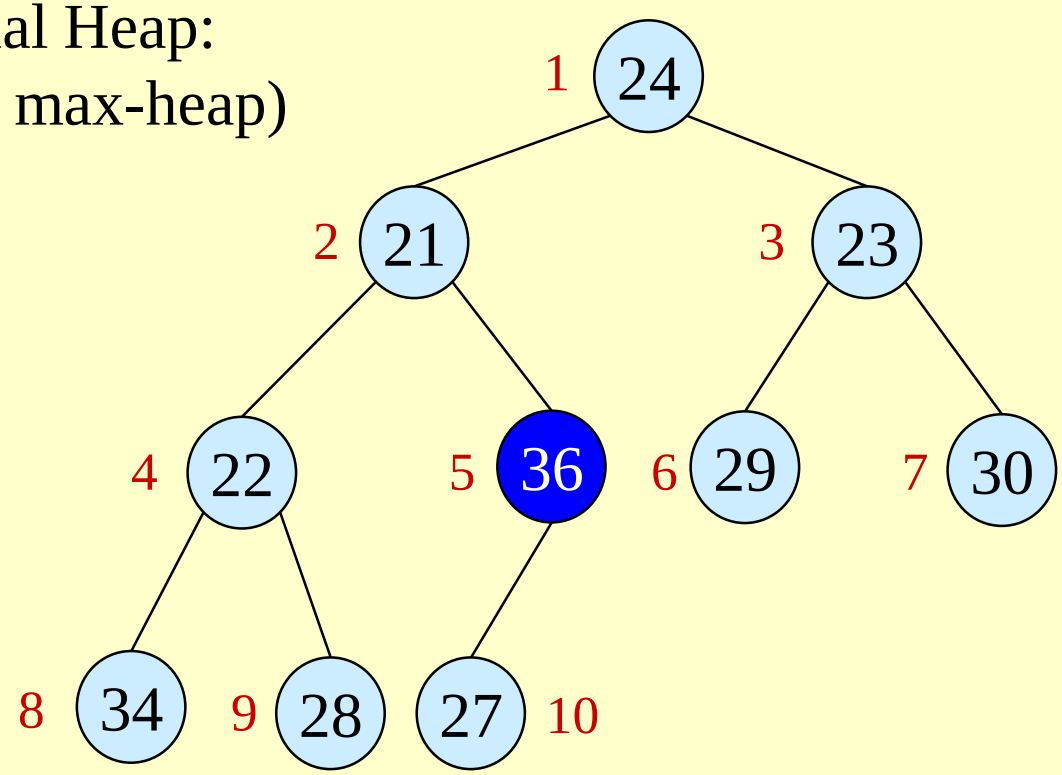
… are leaf nodes.*)

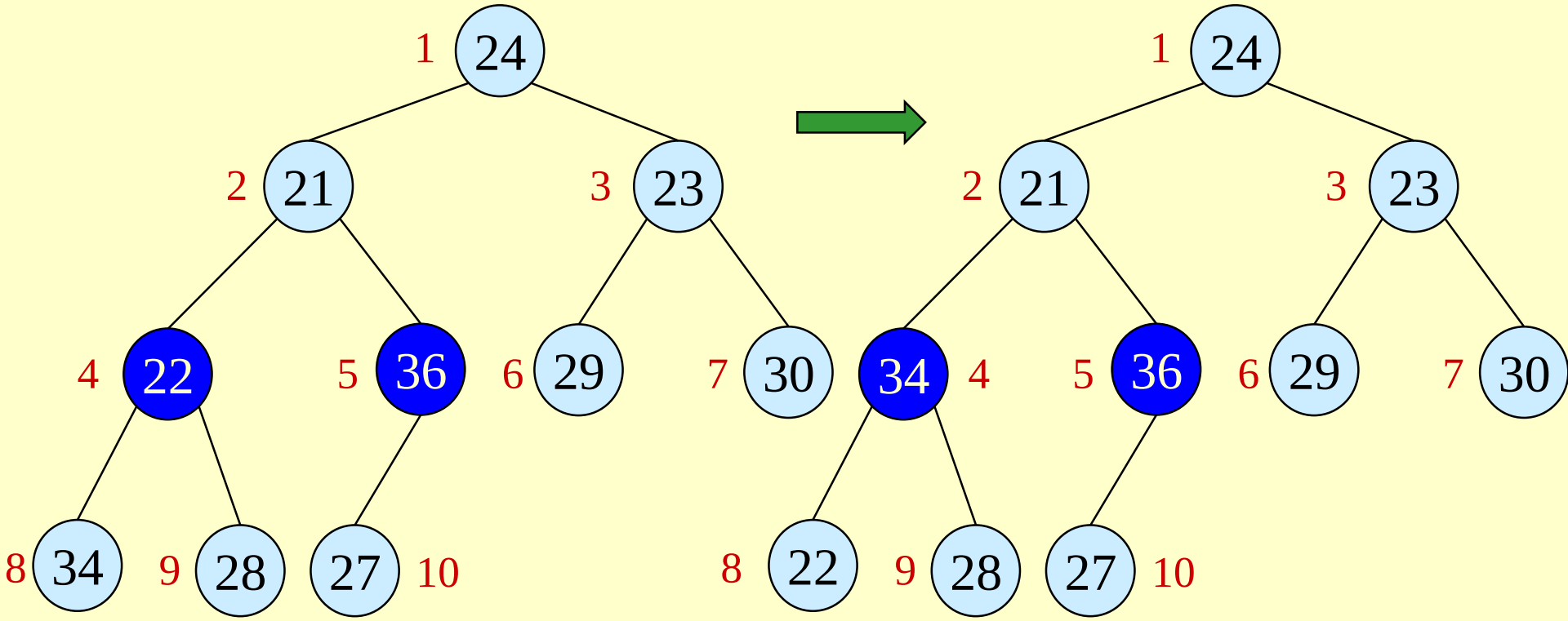
BuildMaxHeap – Example

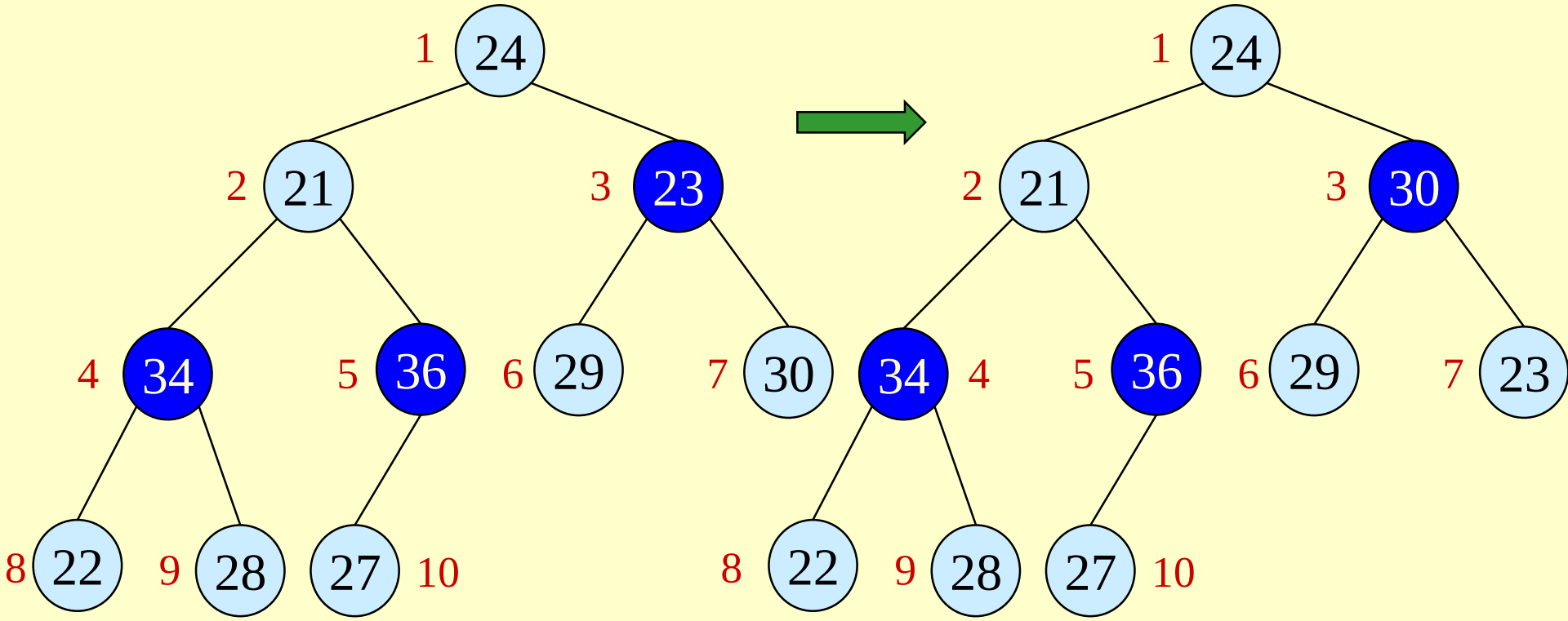
Input Array:

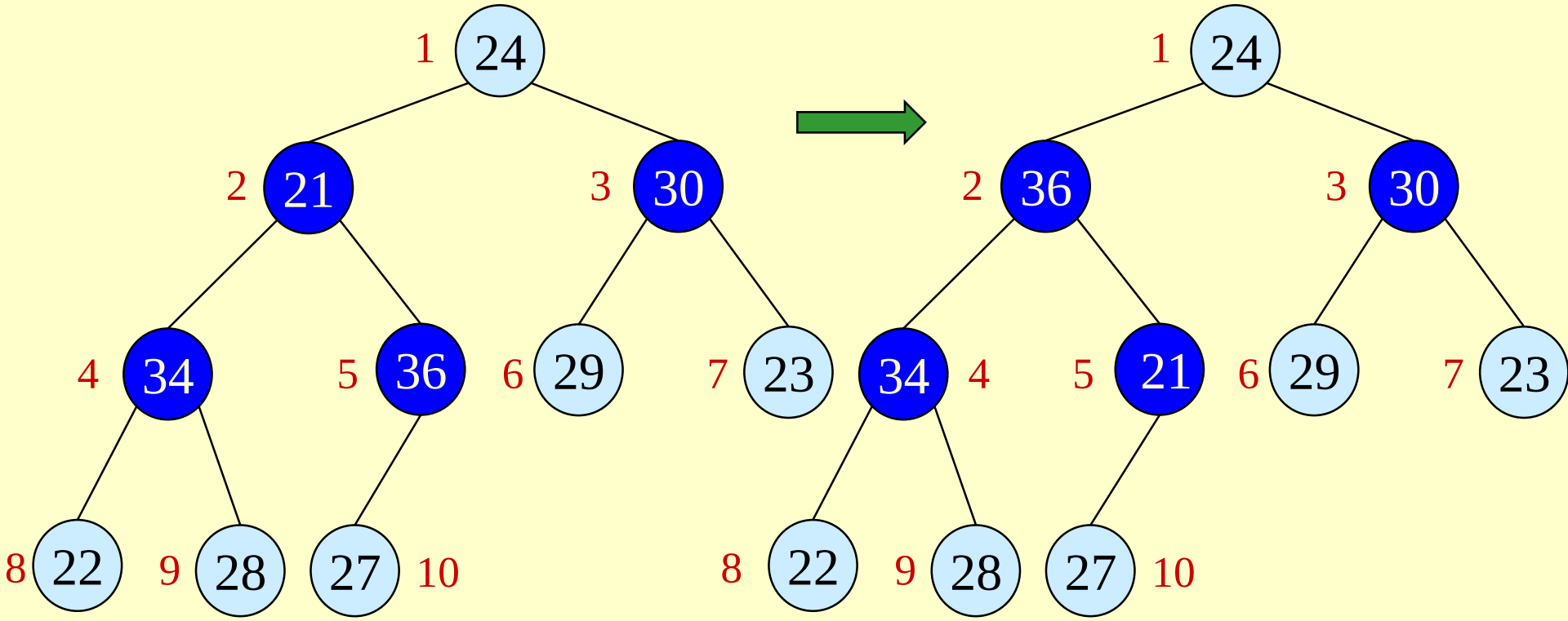
24	21	23	22	36	29	30	34	28	27

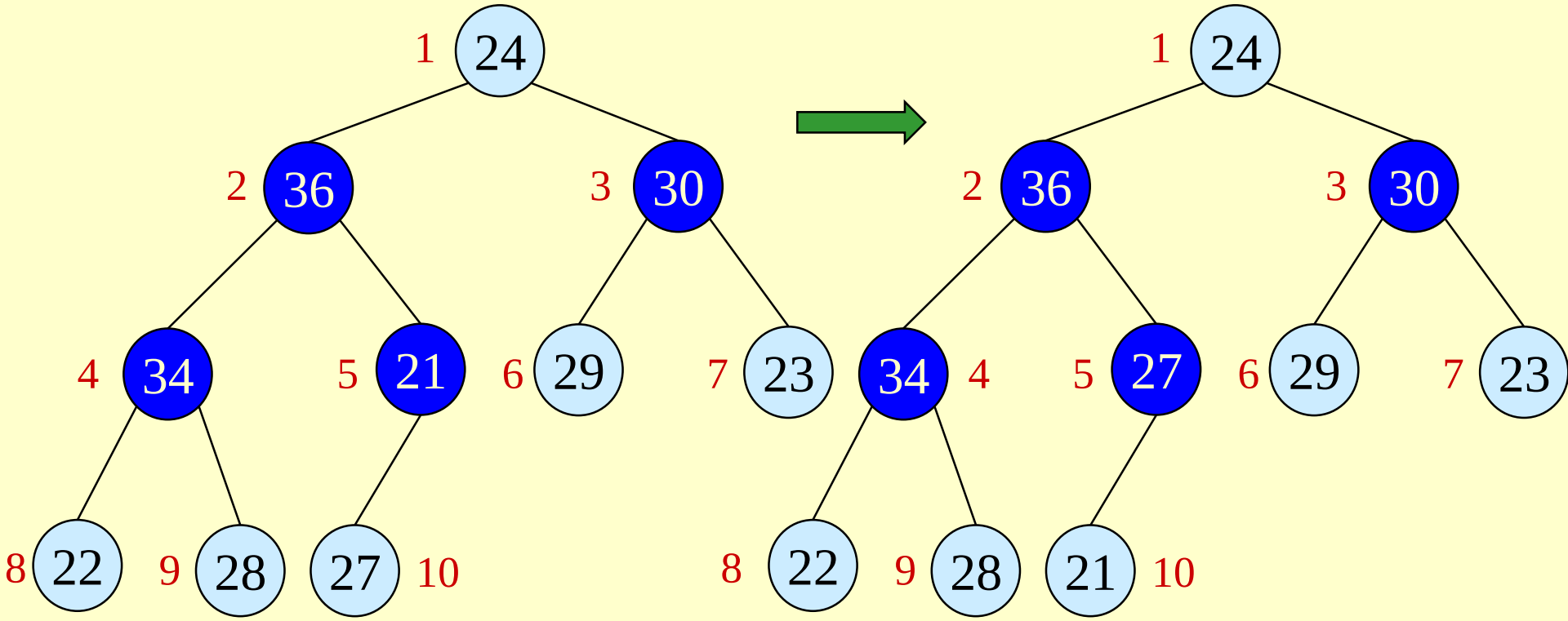
Initial Heap:
(not max-heap)

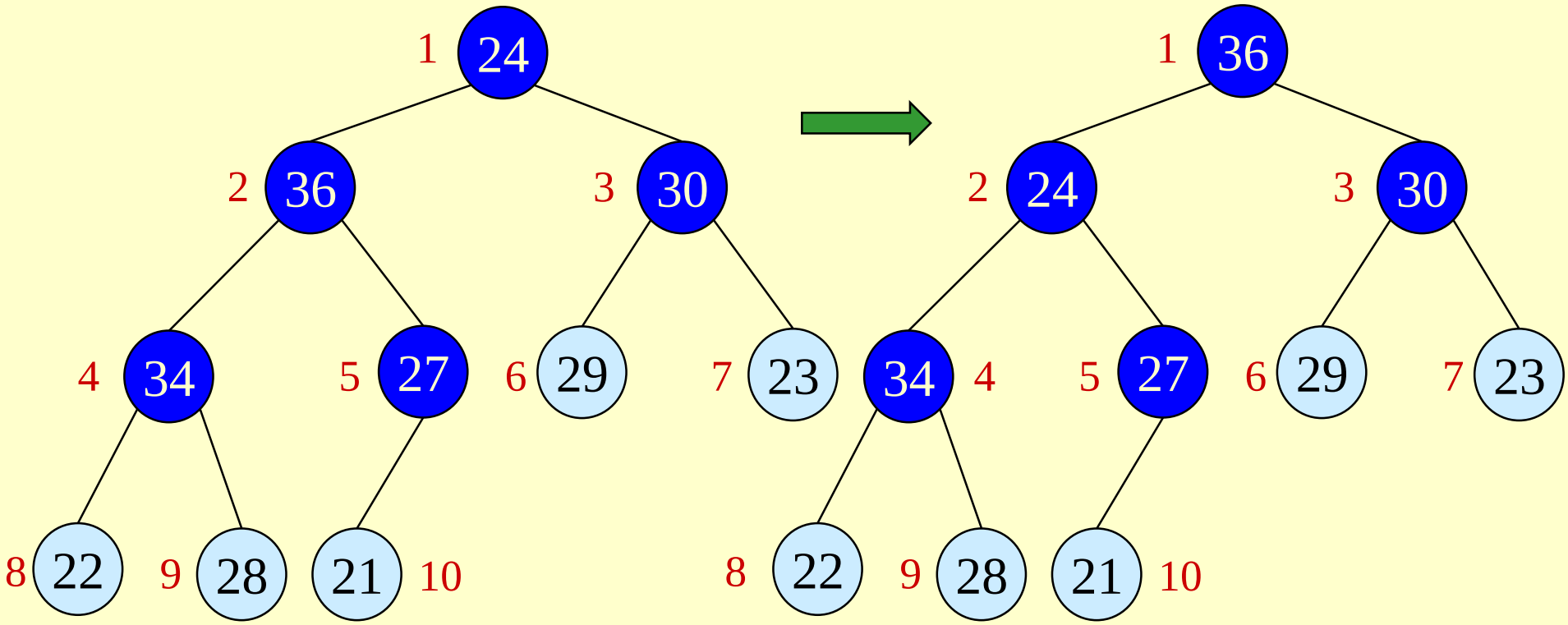



1
24
2
21
3
23
4
22
5
36
6
29
7
30
8
34
9
28
27
10
1
24
2
21
3
23
34
4
5
36
6
29
7
30
8
22
9
28
27
10


1
24
2
21
3
23
4
34
5
36
6
29
7
30
8
22
9
28
27
10
1
24
2
21
3
30
34
4
5
36
6
29
7
23
8
22
9
28
27
10


1
24
2
21
3
30
4
34
5
36
6
29
7
23
8
22
9
28
27
10
1
24
2
36
3
30
34
4
5
21
6
29
7
23
8
22
9
28
27
10


1
24
2
36
3
30
4
34
5
21
6
29
7
23
8
22
9
28
27
10
1
24
2
36
3
30
34
4
5
27
6
29
7
23
8
22
9
28
21
10


1
24
2
36
3
30
4
34
5
27
6
29
7
23
8
22
9
28
21
10
1
36
2
24
3
30
34
4
5
27
6
29
7
23
8
22
9
28
21
10

BuildMaxHeap – Example

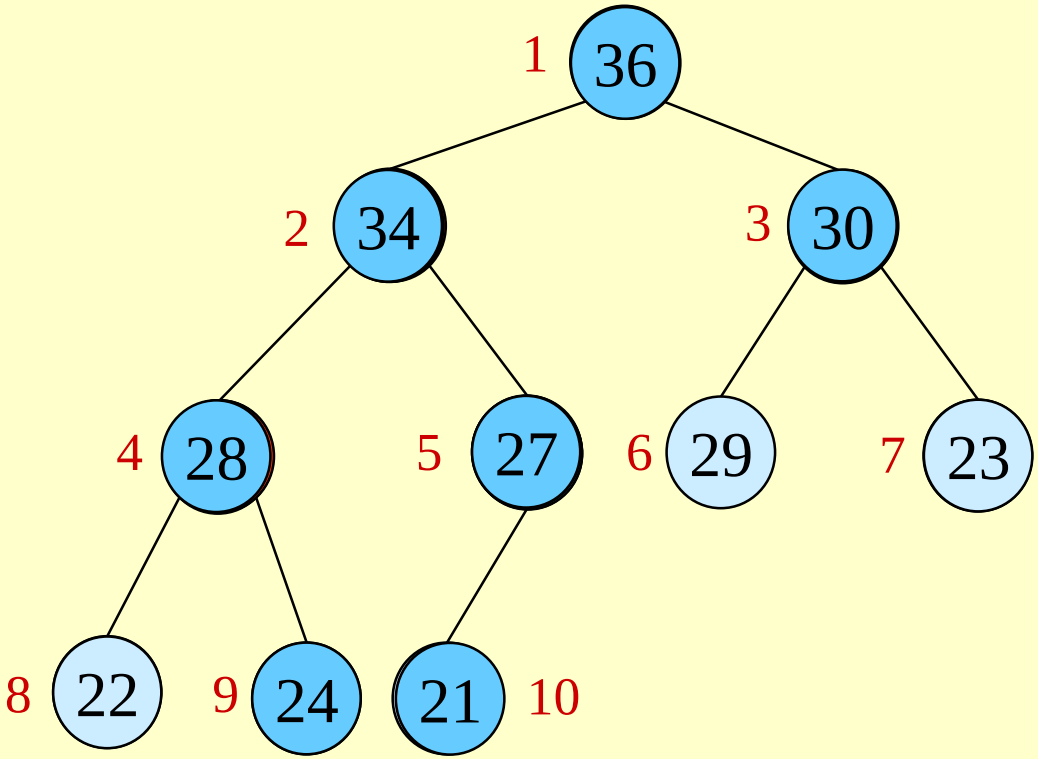
MaxHeapify(⌊10/2⌋ = 5)

MaxHeapify(4)

MaxHeapify(3)

MaxHeapify(2)

MaxHeapify(1)

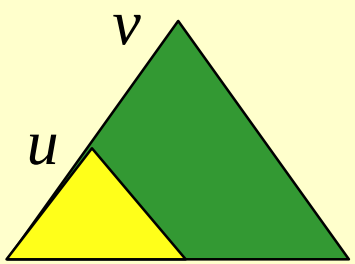


Correctness of *BuildMaxHeap*

- Loop Invariant: At the start of each iteration of the **for** loop, each node $i + 1$, $i + 2$, …, n is the root of a max-heap.
- Initialization:
  - » Before first iteration $i = \lfloor n/2 \rfloor$
  - » Nodes $\lfloor n/2 \rfloor+1, \lfloor n/2 \rfloor+2$, …, n are leaves and hence roots of max-heaps.
- Maintenance:
  - » By LI, subtrees at children of node i are max heaps.
  - » Hence, MaxHeapify(i) renders node i a max heap root (while preserving the max heap root property of higher-numbered nodes).
  - » Decrementing i reestablishes the loop invariant for the next iteration.
- Termination:
  - » On the termination, the root will be maxheapified. Thus, the whole tree is a Max-heap.

Running Time of *BuildMaxHeap*

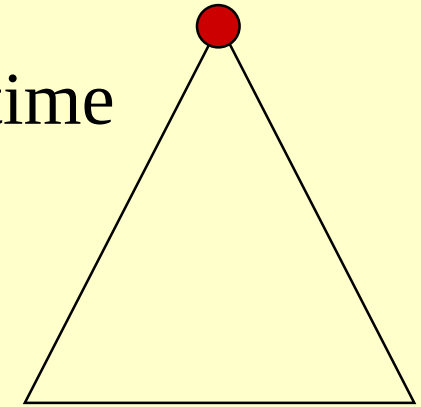
- Loose upper bound:
  - » Cost of a *MaxHeapify* call × No. of calls to *MaxHeapify*
  - » $O(\lg n) \times O(n) = O(n \lg n)$
- Tighter bound:
  - » Cost of a call to *MaxHeapify* at a node depends on the height, h, of the node – $O(h)$.
  - » Height of most nodes smaller than $\lfloor \lg n \rfloor$.
  - » Height of nodes h ranges from 0 to $\lfloor \lg n \rfloor$.
  - » No. of nodes of height h is at most $\lceil n/2^{h+1} \rceil$?



$$\text{Cost}_{maxheapify}(v) > \text{Cost}_{maxheapify}(u)$$

Height

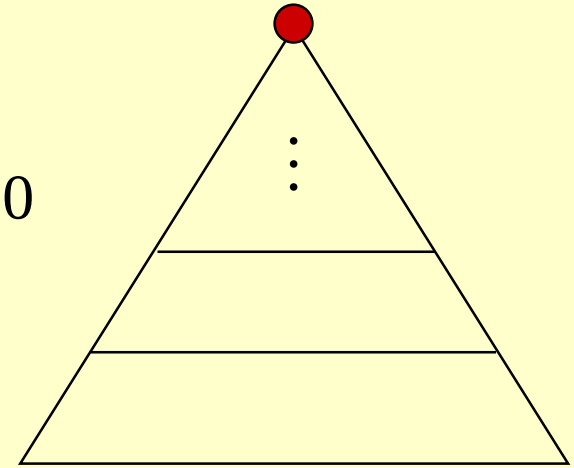
- *Height of a node in a tree*: the number of edges on the longest simple downward path from the node to a leaf.
- *Height of a tree*: the height of the root.
- Height of a heap: $\lfloor \lg n \rfloor$
  - » Basic operations on a heap run in $O(\lg n)$ time



Heap Characteristics

- *Height* $= h$
- No. of *leaves* $\leq \lceil n/2 \rceil$
- No. of nodes of height $h \leq \lceil n/2^{h+1} \rceil$?

height(*a leaf*) = 0



$\leq \lceil n/2^{2+1} \rceil$

$\leq \lceil n/2^{1+1} \rceil$

$\leq \lceil n/2^{0+1} \rceil$

Running Time of *BuildMaxHeap*

Tighter Bound for *T*(*BuildMaxHeap*)

T(*BuildMaxHeap*)

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

$$= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}$$

$$\le \sum_{h=0}^{\infty} \frac{h}{2^h}$$

$$= \frac{1/2}{(1-1/2)^2}$$

$$= 2$$

$x = 1$ in (A.8)

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= O(n)$$

Can build a heap from an unordered array in linear time.

Heapsort

- Sort by maintaining the as yet unsorted elements as a max-heap.
- Start by building a max-heap on all elements in A.
  - » Maximum element is in the root, $A[1]$.
- Move the maximum element to its correct final position.
  - » Exchange $A[1]$ with $A[n]$.
- Discard $A[n]$ – it is now sorted.
  - » Decrement heap-size[A] by one.
- Restore the max-heap property on $A[1..n–1]$.
  - » Call *MaxHeapify*(A, 1).
- Repeat until heap-size[A] is reduced to 2.

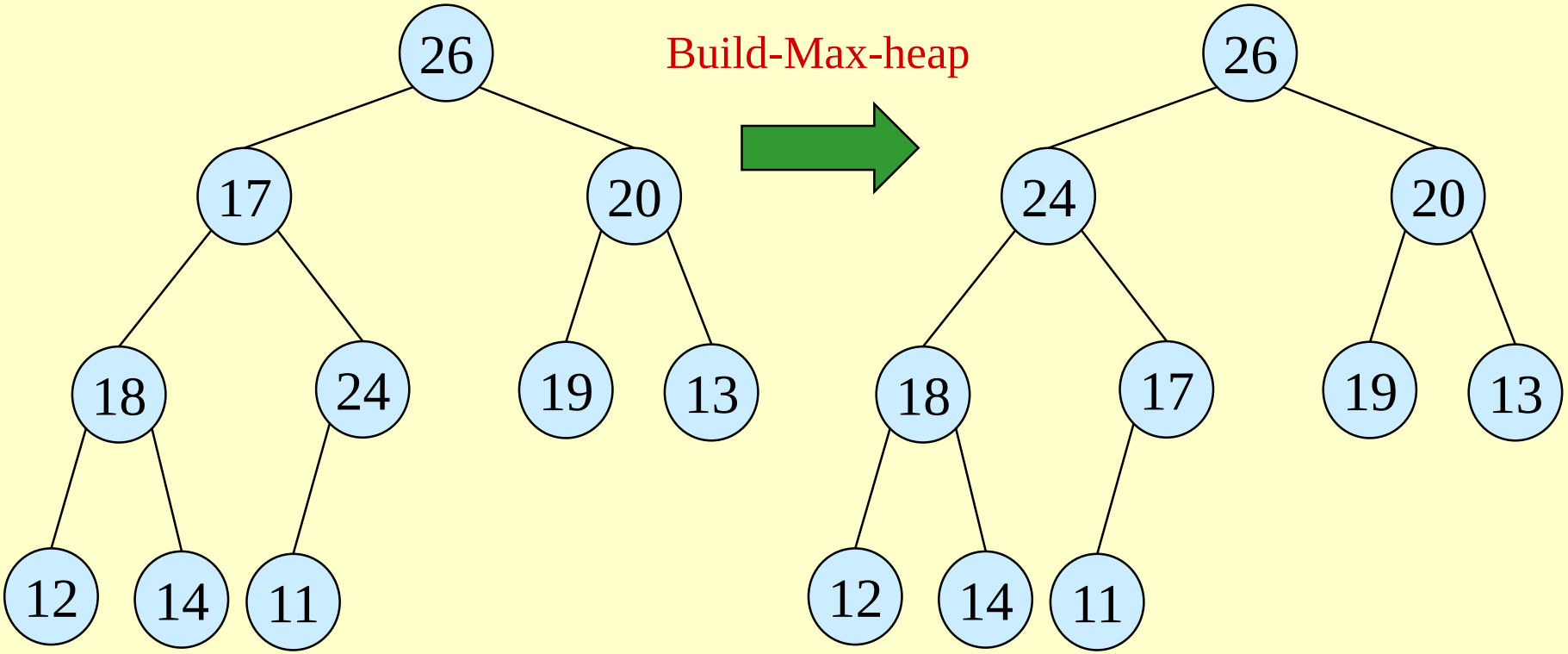
Heapsort(*A*)

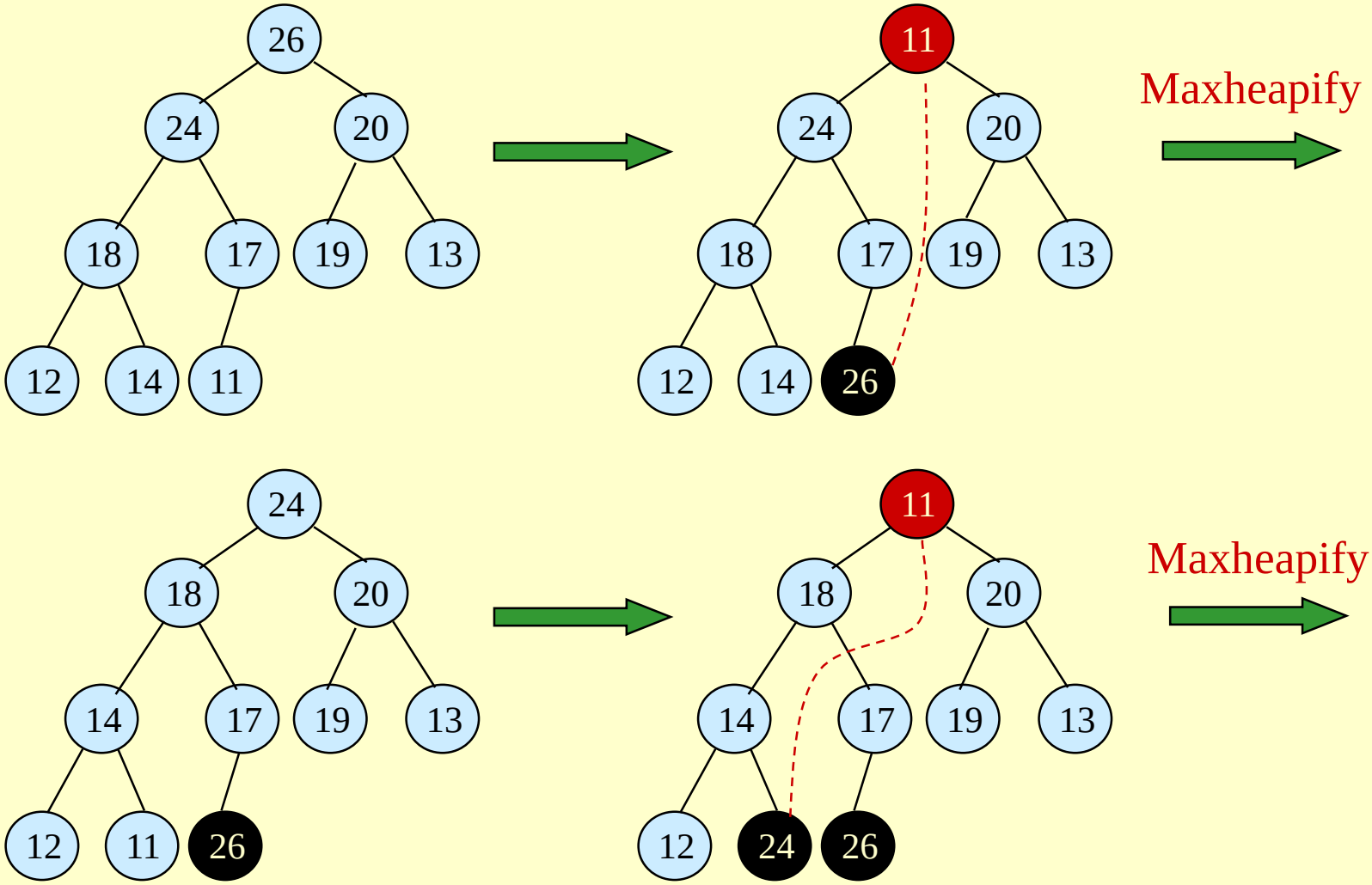
HeapSort(*A*)

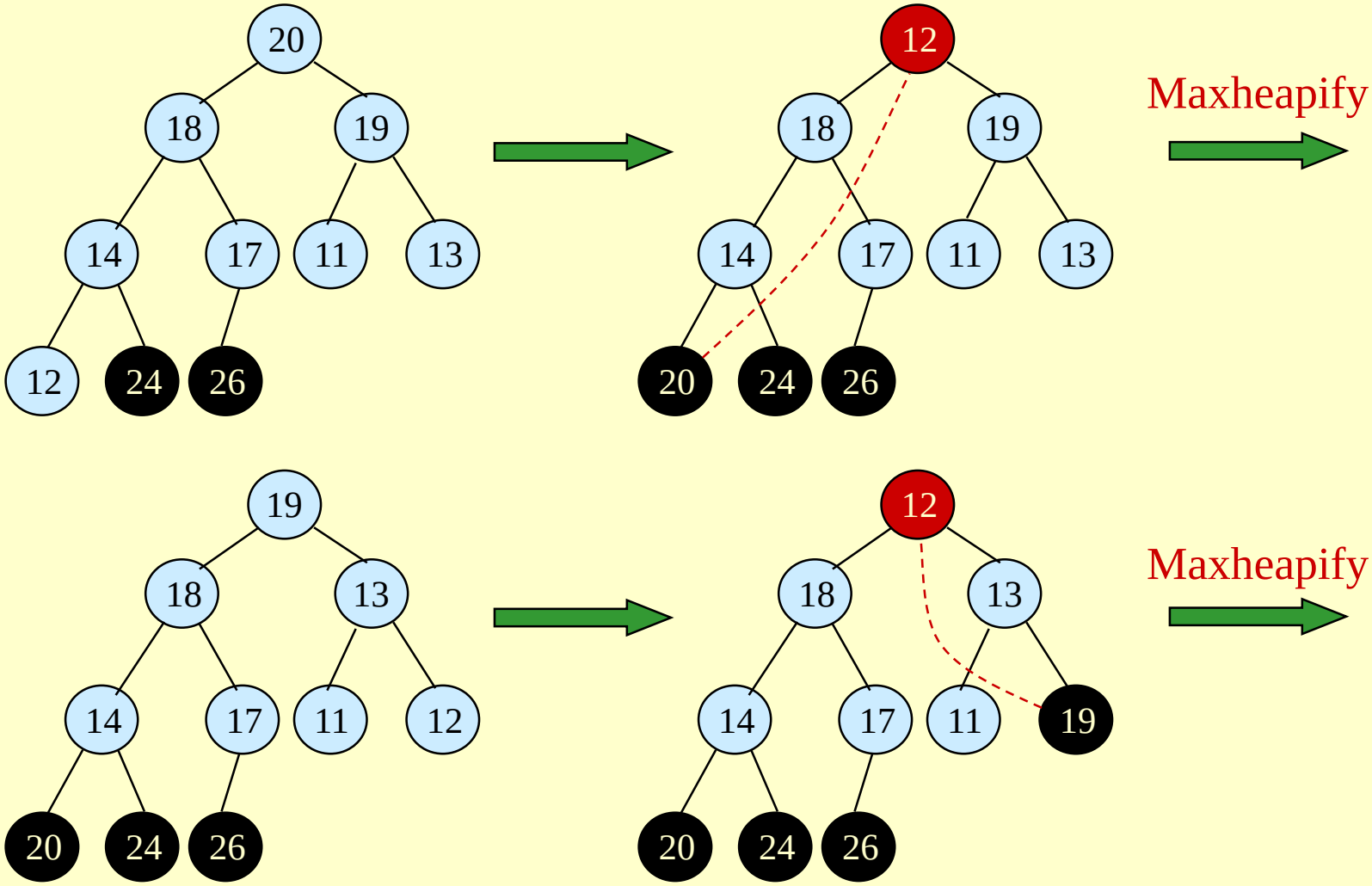
1. Build-Max-Heap(*A*)
2. **for** $i \leftarrow$ *length*[*A*] **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. *heap-size*[*A*] $\leftarrow$ *heap-size*[*A*] – 1
5. *MaxHeapify*(*A*, 1)

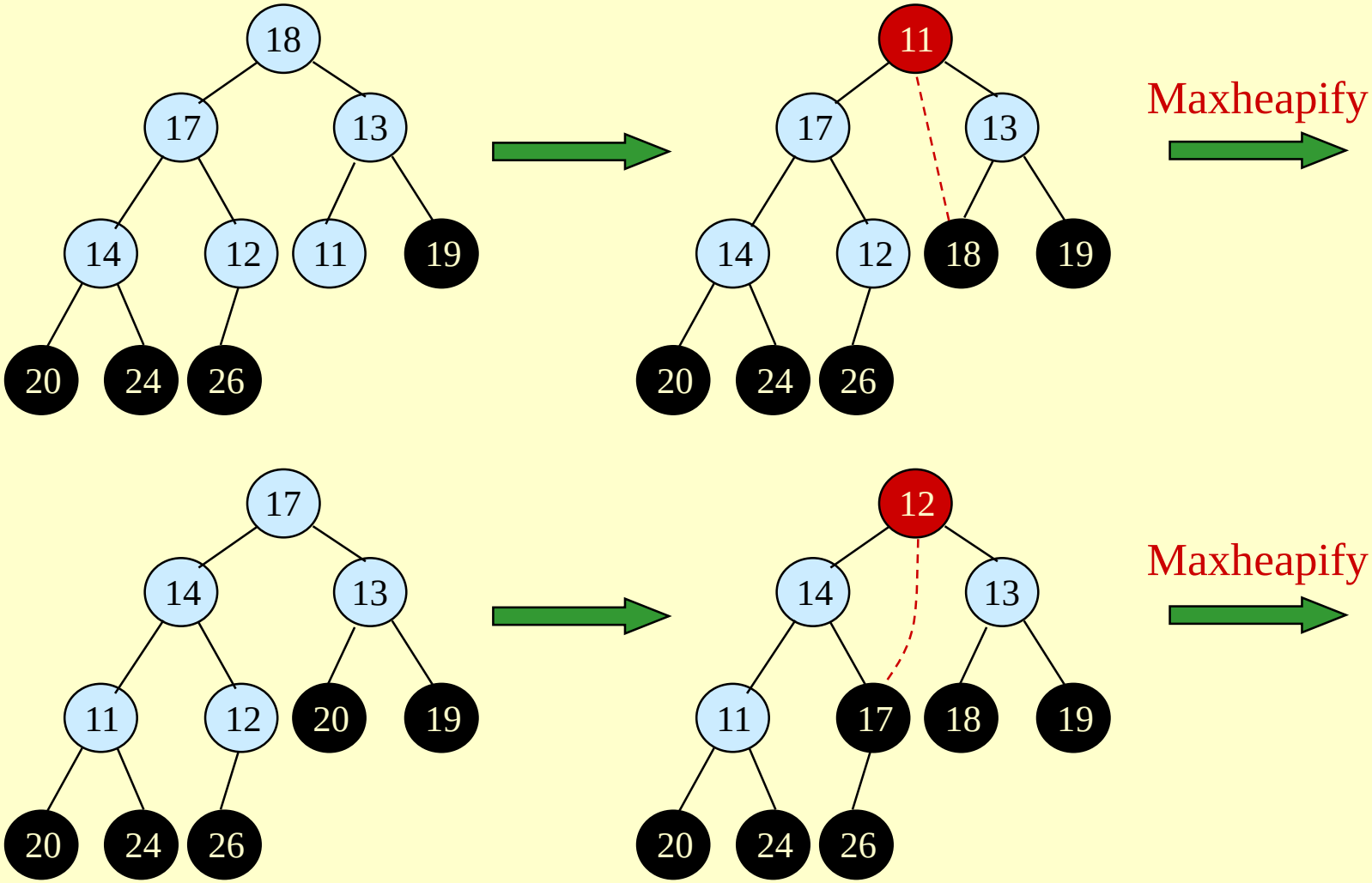
Heapsort – Example

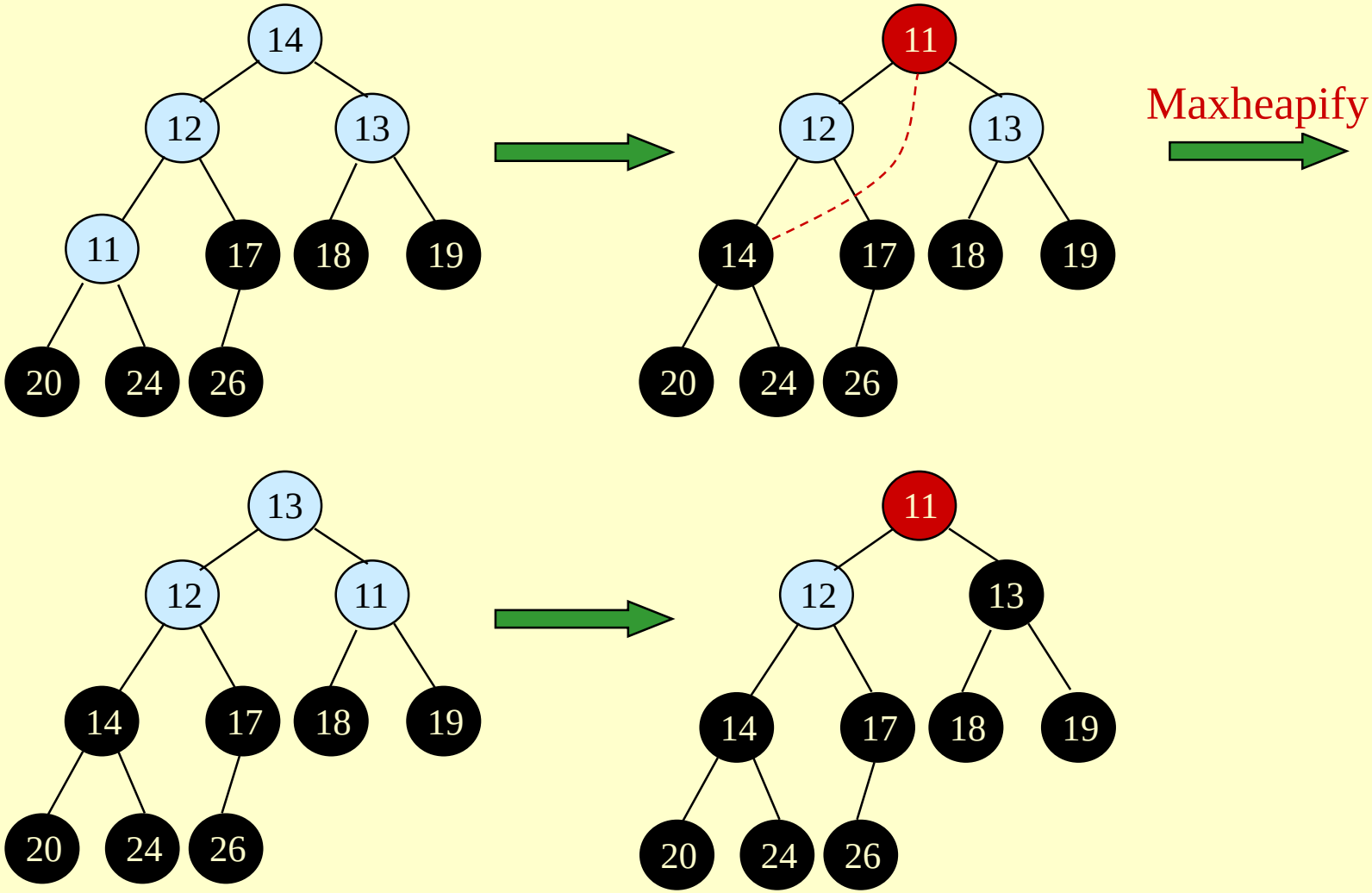
26	17	20	18	24	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10

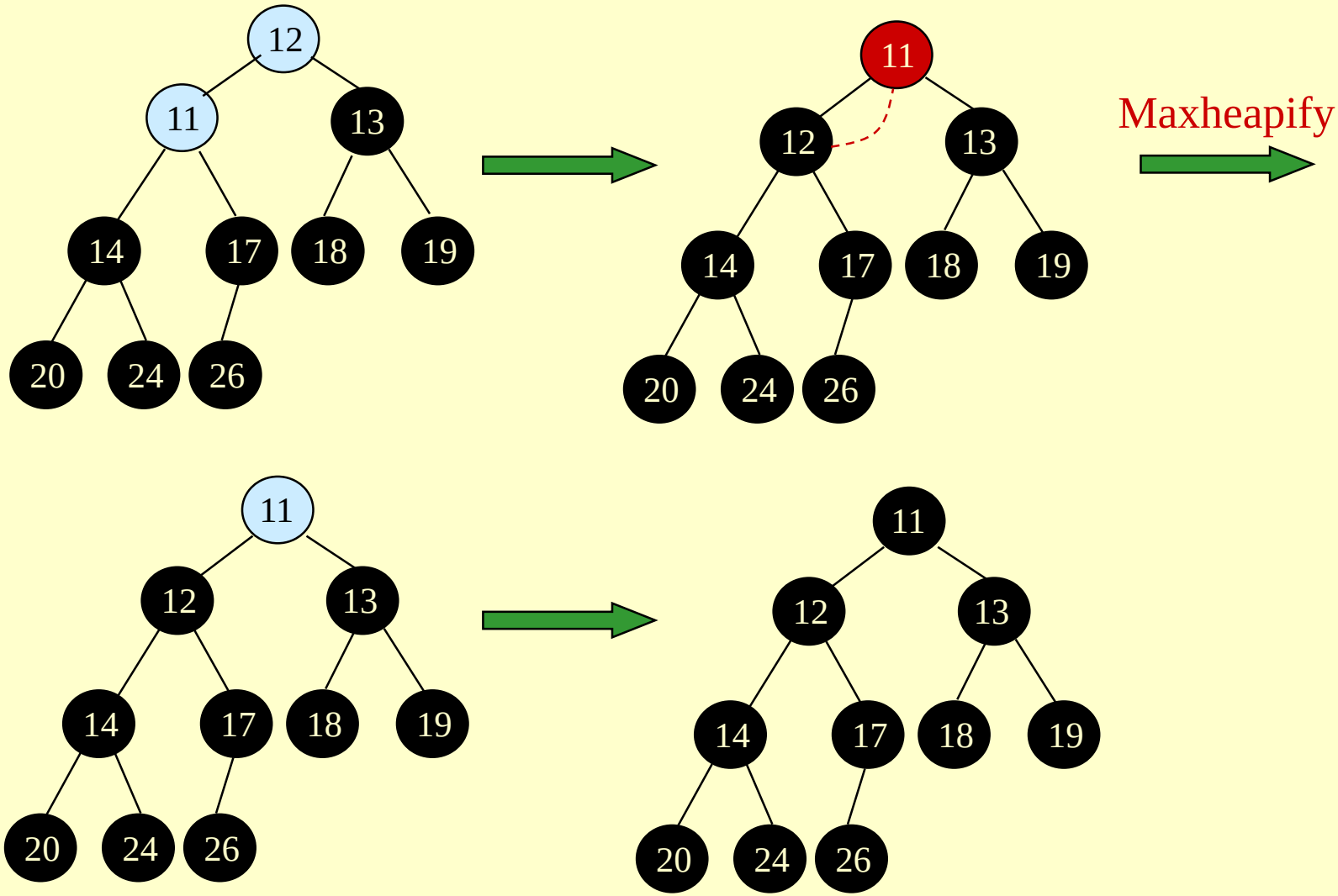



26
24
20
18
17
19
13
12
14
11
11
24
20
18
17
19
13
12
14
26
Maxheapify
24
18
20
14
17
19
13
12
11
26
11
18
20
14
17
19
13
12
24
26
Maxheapify


20
18
19
14
17
11
13
12
24
26
12
18
19
14
17
11
13
20
24
26
Maxheapify
19
18
13
14
17
11
12
20
24
26
12
18
13
14
17
11
19
20
24
26
Maxheapify


18
17
13
14
12
11
19
20
24
26
11
17
13
14
12
18
19
20
24
26
Maxheapify
17
14
13
11
12
20
19
20
24
26
12
14
13
11
17
18
19
20
24
26
Maxheapify


14
12
13
11
17
18
19
20
24
26
11
12
13
14
17
18
19
20
24
26
Maxheapify
13
12
11
14
17
18
19
20
24
26
11
12
13
14
17
18
19
20
24
26


12
11
13
14
17
18
19
20
24
26
11
12
13
14
17
18
19
20
24
26
Maxheapify
11
12
13
14
17
18
19
20
24
26
11
12
13
14
17
18
19
20
24
26

Algorithm Analysis

HeapSort(*A*)

1. Build-Max-Heap(*A*)
2. **for** $i \leftarrow$ *length*[*A*] **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. *heap-size*[*A*] $\leftarrow$ *heap-size*[*A*] – 1
5. *MaxHeapify*(*A*, 1)

- In-place
- Build-Max-Heap takes $O(n)$ and each of the n-1 calls to Max-Heapify takes time $O(\lg n)$.
- Therefore, $T(n) = O(n \lg n)$

Heap Procedures for Sorting

- MaxHeapify $O(\lg n)$
- BuildMaxHeap $O(n \lg n)$, $O(n)$?
- HeapSort $O(n \lg n)$

Priority Queue

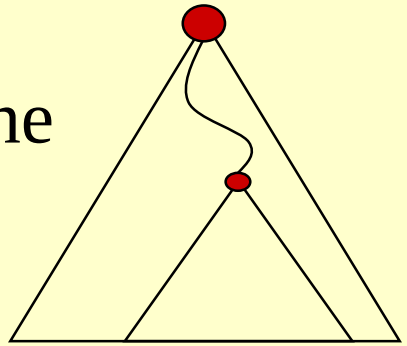
- Popular & important application of heaps.
- Max and min priority queues.
- Maintains a *dynamic* set *S* of elements.
- Each set element has a *key* – an associated value.
- Goal is to support insertion and extraction efficiently.
- Applications:
  - » Ready list of processes in operating systems by their priorities – the list is highly dynamic
  - » In event-driven simulators to maintain the list of events to be simulated in order of their time of occurrence.

Basic Operations

- Operations on a max-priority queue:
  - » Insert(S, x) - inserts the element x into the queue S
    - $S \leftarrow S \cup \{x\}$.
  - » Maximum(S) - returns the element of S with the largest key.
  - » Extract-Max(S) - removes and returns the element of S with the largest key.
  - » Increase-Key(S, x, k) – increases the value of element x's key to the new value k.
- Min-priority queue supports Insert, Minimum, Extract-Min, and Decrease-Key.
- Heap gives a good compromise between fast insertion but slow extraction and vice versa.

Heap Property (Max and Min)

- Max-Heap
  - » For every node excluding the root, the value stored in that node is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree root.



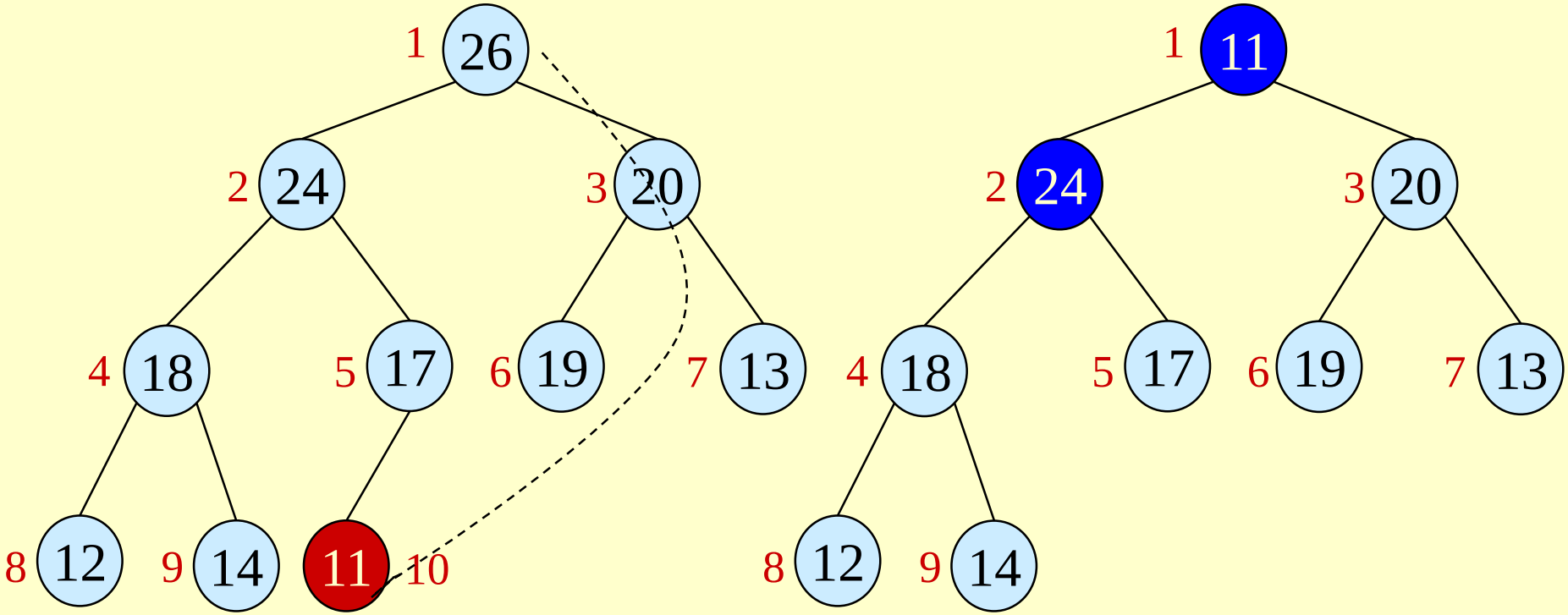
- Min-Heap
  - » For every node excluding the root, the value stored in that node is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at subtree root

Heap-Extract-Max(A)

Implements the Extract-Max operation.

Heap-Extract-Max(A)
1. if *heap-size*[A] < 1
2. then error "heap underflow"
3. $max \leftarrow A[1]$
4. $A[1] \leftarrow A[$*heap-size*$[A]]$
5. *heap-size*$[A] \leftarrow$ *heap-size*$[A] - 1$
6. MaxHeapify(A, 1)
7. return *max*

Running time : Dominated by the running time of MaxHeapify
= $O(\lg n)$


1
26
2
24
3
20
4
18
5
17
6
19
7
13
8
12
9
14
11
10
1
11
2
24
3
20
4
18
5
17
6
19
7
13
8
12
9
14

Heap-Insert(*A*, *key*)

Heap-Insert(*A*, *key*)

1. *heap-size*[*A*] ← *heap-size*[*A*] + 1
2. *i* ← *heap-size*[*A*]
4. **while** *i* > 1 **and** *A*[Parent(*i*)] < *key*
5. **do** *A*[*i*] ← *A*[Parent(*i*)]
6. *i* ← Parent(*i*)
7. *A*[*i*] ← *key*

Running time is $O(\lg n)$

The path traced from the new leaf to the root has length $O(\lg n)$

Examples

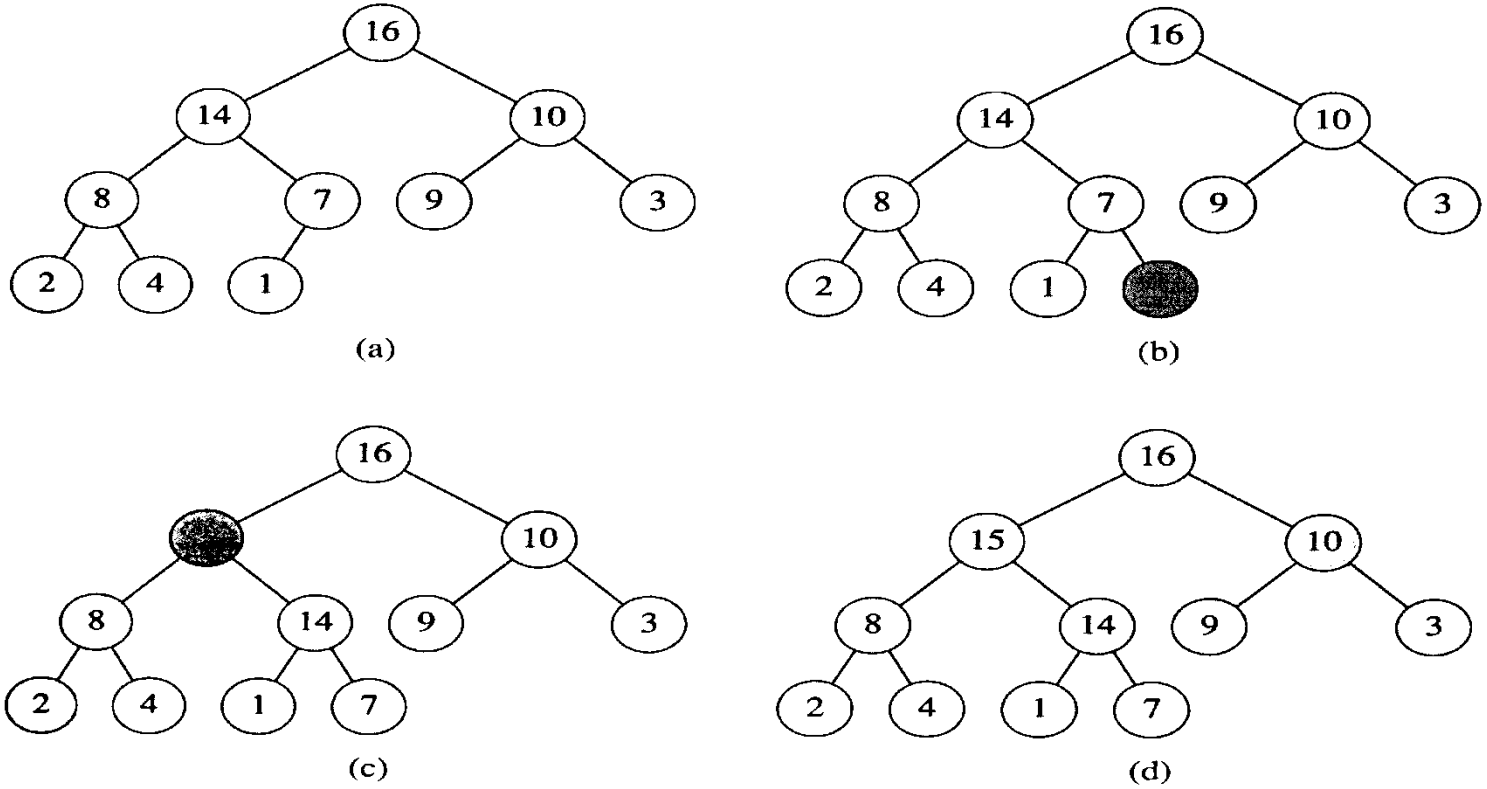


Figure 7.5 The operation of HEAP-INSERT. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.

Heap-Increase-Key(*A*, *i*, *key*)

Heap-Increase-Key(*A*, *i*, *key*)

```
1    If key < A[i]
2       then error "new key is smaller than the current key"
3    A[i] ← key
4    while i > 1 and  A[Parent[i]] < A[i]
5          do exchange A[i] ↔ A[Parent[i]]
6             i ← Parent[i]
```

Heap-Insert(*A*, *key*)

```
1    heap-size[A] ← heap-size[A] + 1
2    A[heap-size[A]] ← – ∞
3    Heap-Increase-Key(A, heap-size[A], key)
```