SORTING (2)

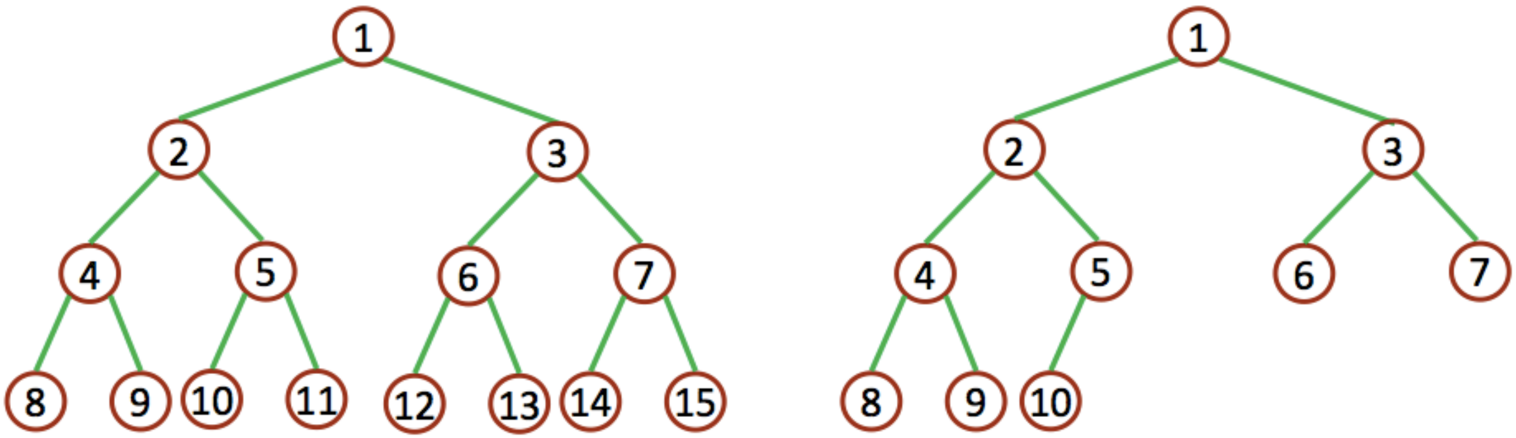
KULIAH ANALISIS ALGORITMA DAN KOMPLEKSITAS

HEAPSORT

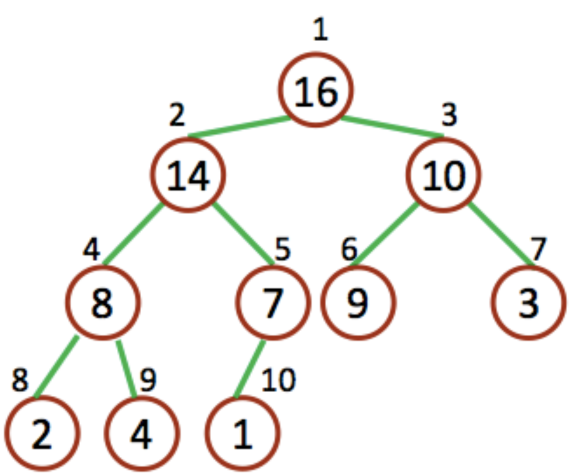
- **Running time: O(n lg n) → seperti merge sort.**
- **Hanya memerlukan 1 larik → seperti insertion sort.**
- **Menggunakan heap untuk mengelola informasi → tidak hanya berguna untuk sorting, tetapi juga untuk dapat dipakai untuk priority queue.**

HEAP

- (Binary) heap adalah objek larik yang dapat dipandang sebagai pohon biner yang hampir lengkap.
- Setiap node pada pohon berkorespondensi dengan elemen larik.
- Node pada pohon lengkap di semua level, kecuali mungkin level paling bawah.



- **Larik A yang merepresentasikan heap memiliki 2 atribut:**
  - A.length → jumlah elemen pada heap
  - A.heap-size → jumlah elemen pada heap yang disimpan pada larik A, dengan 0 ≤ A.heapsize ≤ A.length.
- **Akar dari pohon menjadi A[1].**
- **Heap direpresentasikan dari level ke level, dari kiri ke kanan.**



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

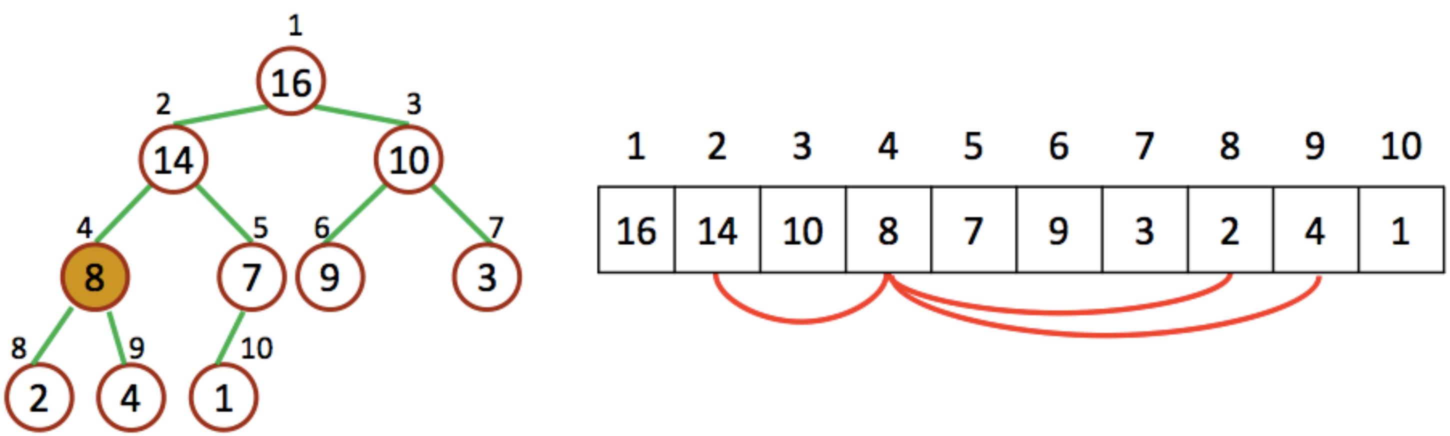
length[A]=heapsize[A]=10

- **Untuk suatu node dengan indeks i, dapat dicari parent-nya, anak kiri, dan anak kanan.**
  - Parent(i)

    return $\lfloor i/2 \rfloor$ → jika $i \neq 1$
  - Left(i)

    return 2i
  - Right(i)

    return 2i+1

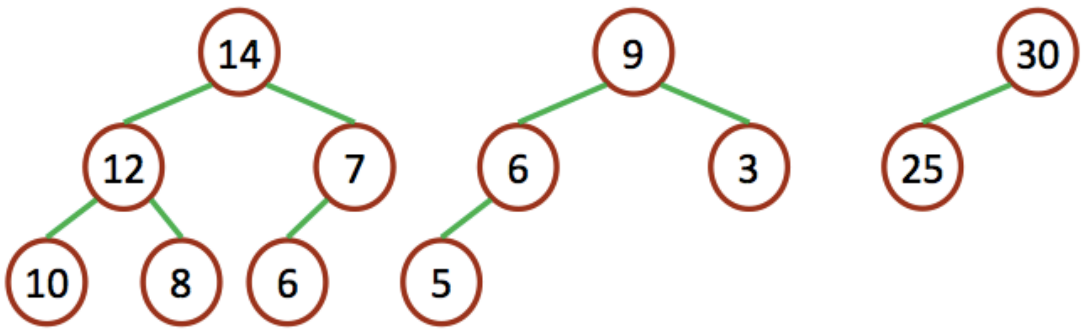


- **Perhitungan prosedur untuk bit:**
  - Menghitung Left(i): shift left 1 bit.
  - Menghitung Right(i): shift left 1 bit, kemudian menambahkan 1 sebagai bit paling kanan.
  - Menghitung Parent(i): shift right 1 posisi.

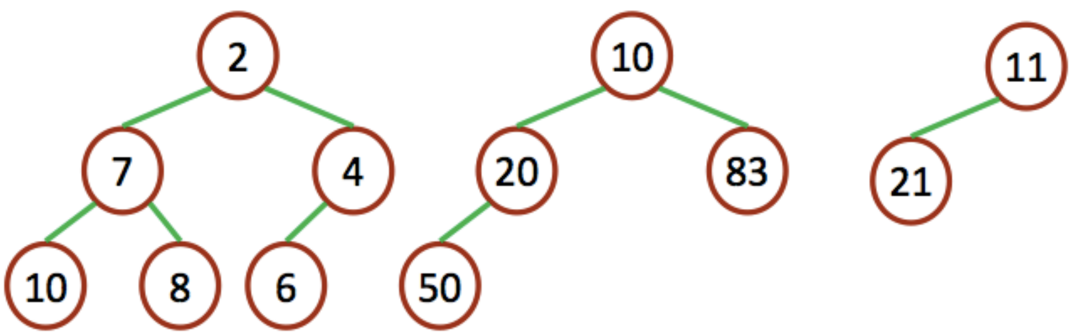
PROPERTI BINARY HEAPS

- **Terdapat 2 macam binary heaps:**
  - Max-heap
  - Min-heap
- **Properti max-heap: A[parent(*i*)] ≥ A[*i*] → akan dipakai pada sorting**
  - Nilai paling tinggi ada di root node
  - Sub-tree yang berakar pada suatu node memuat nilai-nilai yang tidak lebih besar dari nilai pada node tsb.
- **Properti min-heap: A[parent(*i*)] ≤ A[*i*]**
  - Nilai paling rendah ada di root node
  - Sub-tree yang berakar pada suatu node memuat nilai-nilai yang tidak lebih kecil dari nilai pada node tsb.

MAX HEAP DAN MIN HEAP



Max Heaps



Min Heaps

TINGGI HEAP

- **Height/tinggi node pada heap: jumlah sisi terpanjang pada pohon dari suatu node ke leaf.**
- **Kapankah suatu heap memiliki jumlah elemen maksimum?**
  - Jumlah maksimum elemen pada heap dengan tinggi h: $2^{h+1}-1$
- **Kapankah suatu heap memiliki jumlah elemen minimum?**
  - Jumlah minimum elemen pada heap dengan tinggi h: 2^{h}

TINGGI HEAP DENGAN N ELEMEN

- **Tinggi pohon dengan *n* node adalah $\lfloor lg\ n \rfloor$, yang merupakan Θ(lg *n*).**
- **Berarti, heap dengan n-element memiliki tinggi $\lfloor lg\ n \rfloor$**
  - $2^h \leq n \leq 2^{h+1}$
  - Ambil logaritma basis 2 untuk semua sisi:
  - $h \leq \lg n \leq h + 1 \rightarrow h = \lfloor lg\ n \rfloor$

OPERASI-OPERASI PADA HEAP

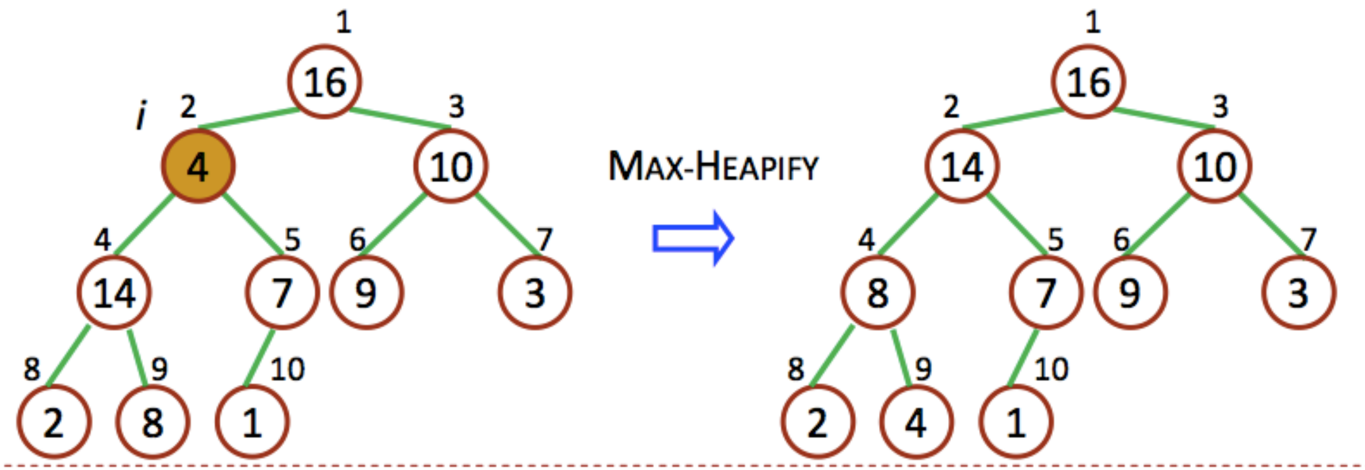
- **Prosedur MAX-HEAPIFY:**
  - Kompleksitas $O(\lg n)$
  - Digunakan untuk menjaga properti max-heap.
- **Prosedur BUILD-MAX-HEAP:**
  - Kompleksitas $O(n)$
  - Menghasilkan max-heap dari array yang tidak terurut.
- **Prosedur HEAPSORT**
  - Kompleksitas: $O(n \lg n)$
  - Mengurutkan array.
- **Prosedur MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, dan HEAP-MAXIMUM**
  - Kompleksitas $O(\lg n)$
  - Memungkinkan struktur data heap digunakan sebagai priority queue.

MENJAGA PROPERTI HEAP

- **Diberikan larik A dan indeks i pada larik.**
  - Subtree yang berakar pada anak dari A[*i*] merupakan heap tetapi node A[*i*] sendiri belum tentu memenuhi properti heap.
  - Dengan kata lain, bisa jadi A[*i*] < A[2*i*] atau A[*i*] < A[2*i* +1].
- **Prosedur 'Max-Heapify' digunakan untuk memanipulasi pohon tersebut sehingga menjadi heap.**

PROSEDUR MAX-HEAPIFY

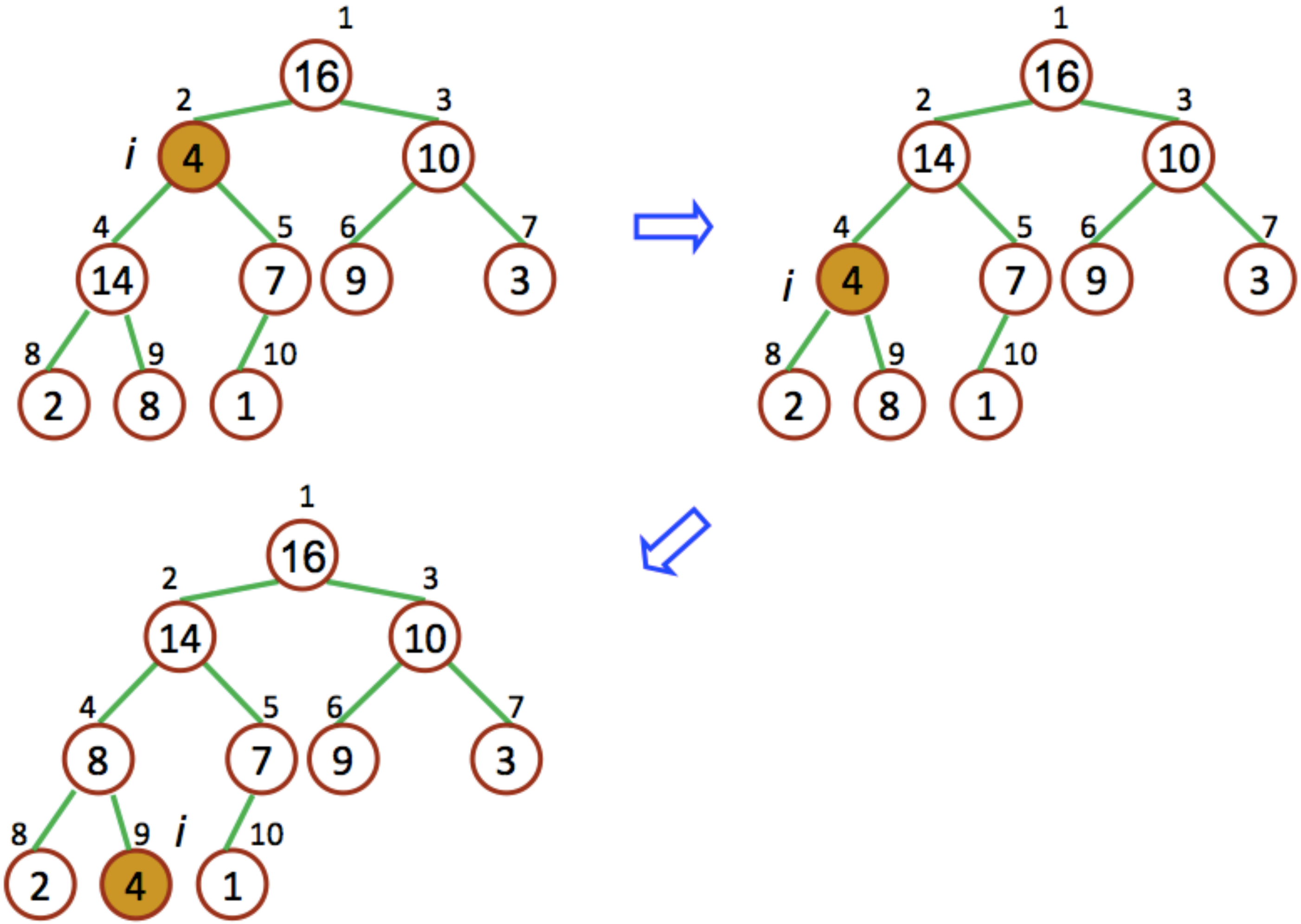
- **Merupakan prosedur penting untuk memanipulasi heap.**
  - Input: array A dan indeks *i*
  - Output: subtree yang berakar pada indeks i menjadi max-heap
  - Asumsi: pohon biner yang berakar pada LEFT(*i*) dan RIGHT(*i*) merupakan max-heap, tetapi A[i] dapat lebih kecil dari anak-anaknya
- **Heapify memilih child terbesar, kemudian dibandingkan dengan parent. Jika parent lebih besar, maka keluar, jika tidak tukar parent dengan child tersebut.**
- **Jika swap merusak properti heap, maka prosedur tersebut memanggil dirinya sendiri dengan child terbesar sebagai root.**



- **MAX-HEAPIFY(A, *i*)**

```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
        then largest←l
        else largest ← i
if r ≤ heap-size[A] and a[r] > A[largest]
        then largest ← r
if largest ≠ i
        then exchange A[i] ↔ A[largest]
                MAX-HEAPIFY (A, largest)
```

CONTOH


1
16
2 3
i 4 10
4 5 6 7
14 7 9 3
8 9 10
2 8 1
1
16
2 3
14 10
4 5 6 7
i 4 7 9 3
8 9 10
2 8 1
1
16
2 3
14 10
4 5 6 7
8 7 9 3
8 9 i 10
2 4 1

ANALISIS HEAPIFY

- **Jika nilai pada root selalu lebih kecil dari nilai di kiri dan kanan, Heapify akan dipanggil secara rekursif sampai mencapai leaf.**
- **Untuk mendapatkan longest path, pilih nilai yang membuat Heapify rekursif pada left child.**
- **Dengan demikian, Heapify akan dipanggil h kali, dengan h adalah tinggi heap.**
- **Running time θ (h) (karena setiap pemanggilan memerlukan 1 swap.**
- **Maka worst-case- nya adalah Ω(lgn).**

MEMBUAT HEAP

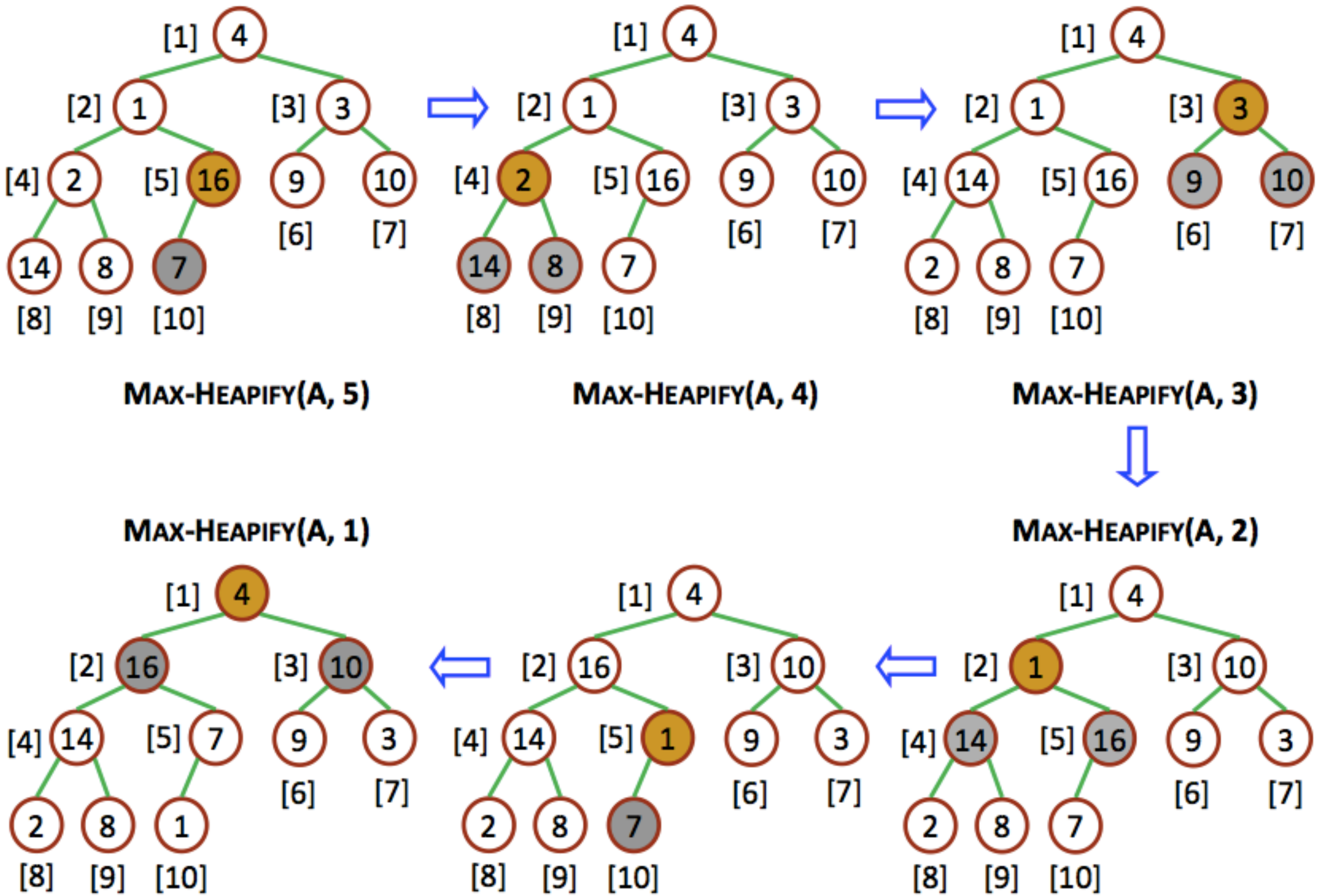
- **Dapat menggunakan prosedur 'Heapify' dari bawah ke atas (bottom-up) untuk mengkonversi larik A[1 . . *n*] menjadi heap.**
- **Karena elemen pada sub-array A[$\lfloor n/2 \rfloor$+1 . . *n*] adalah leaves, prosedur BuildHeap berjalan ke semua sisa node pada tree dan memanggil 'Heapify' pada setiap node.**
- **Pemrosesan secara bottom-up menjamin bahwa subtree yang berakar pada semua child node merupakan heap sebelum Heapify dijalankan pada parent-nya.**

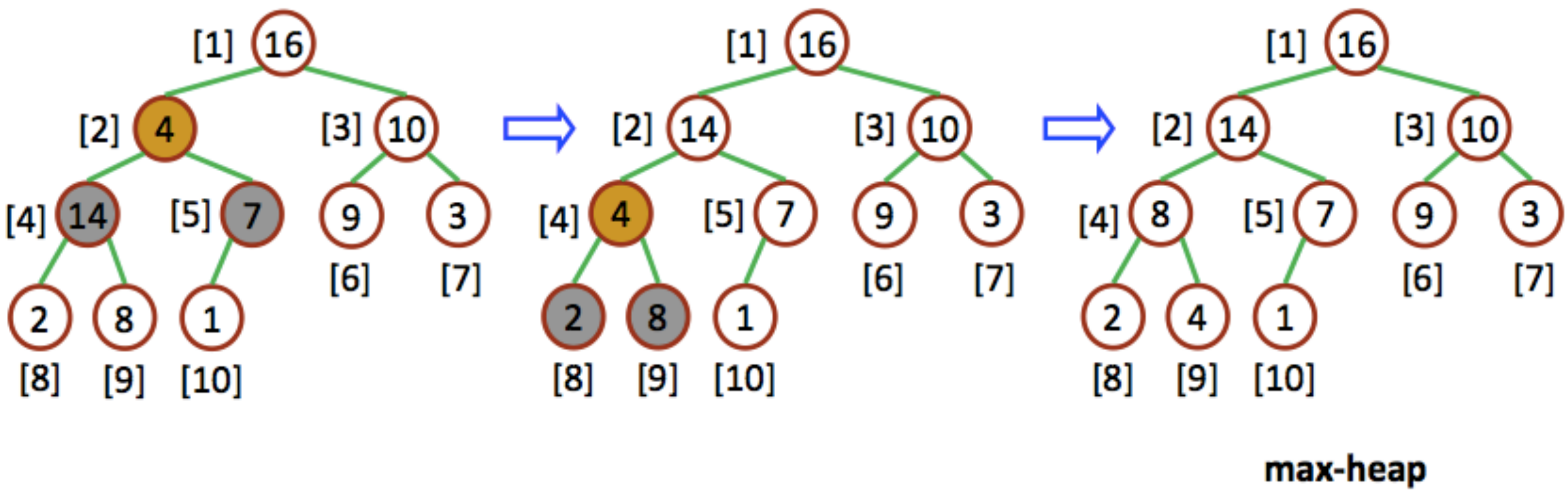
- **Prosedur:**

BUILD-MAX-HEAP(A)

1. *heap-size*[A] ← *length*[A]
2. for *i* ← ⌊*length*[A]/2⌋ downto 1
3. do MAX-HEAPIFY(A,*i*)

CONTOH


MAX-HEAPIFY(A, 5)
MAX-HEAPIFY(A, 4)
MAX-HEAPIFY(A, 3)
MAX-HEAPIFY(A, 2)
MAX-HEAPIFY(A, 1)


[1]
16
[2]
4
[3]
10
[4]
14
[5]
7
9
3
[6]
[7]
2
8
1
[8]
[9]
[10]
[1]
16
[2]
14
[3]
10
[4]
4
[5]
7
9
3
[6]
[7]
2
8
1
[8]
[9]
[10]
[1]
16
[2]
14
[3]
10
[4]
8
[5]
7
9
3
[6]
[7]
2
4
1
[8]
[9]
[10]
max-heap

CORRECTNESS

- **Untuk menunjukkan BUILD-MAX-HEAP benar, digunakan loop invariant:**
  - Pada setiap awal dari iterasi for loop pada baris 2-3, setiap node $i+1$, $i+2$, ..., n merupakan akar dari suatu max-heap.
  - BUILD-MAX-HEAP(A)
    1. *heap-size*[A] ← *length*[A]
    2. for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1
    3. do MAX-HEAPIFY(A,*i*)
- **Perlu ditunjukkan bahwa:**
  - Invariant ini benar sebelum iterasi pertama
  - Setiap iterasi pada loop menjaga invariant ini
  - Invariant mempunyai properti yang berguna untuk menunjukkan correctness ketika loop berhenti.

- **Initialization: sebelum iterasi pertama, $i = \lfloor n/2 \rfloor$.**
  - $\lfloor n/2 \rfloor+1, \ldots n$ merupakan daun, jadi pasti merupakan max-heap.
- **Maintenance: Pada saat memasuki loop, children dari node *i* merupakan max-heaps. Ini persis seperti kondisi yang diperlukan dalam pemanggilan MAX-HEAPIFY(*A*, *i*) untuk membuat node *i* menjadi akar dari max-heap. Selain itu, pemanggilan terhadap MAX-HEAPIFY menjaga properti bahwa node $i + 1, i + 2, \ldots, n$ semuanya merupakan akar dari max-heaps.**
- **Termination: pada akhir iterasi, i=0, setiap node 1, 2, ..., *n* merupakan akar dari max-heap. Secara khusus, node 1 merupakan akar dari heap.**

KOMPLEKSITAS

- **Analisis1:**
  - Setiap pemanggilan pada MAX-HEAPIFY berbiaya $O(\lg n)$, dan terdapat $O(n)$ pemanggilan.
  - Maka, running time adalah $O(n \lg n)$. Batas atas ini benar, tetapi tidak asymptotically tight.
- **Analisis 2:**
  - Untuk heap dengan n-element, tingginya adalah $\lfloor \lg n \rfloor$ dan maksimum terdapat $\lceil n / 2^{h+1} \rceil$ node pada sembarang tinggi h.
  - Waktu yang diperlukan oleh MAX-HEAPIFY pada saat dipanggil pada node dengan tinggi h adalah $O(h)$.
  - Total waktu:

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

- **Summation menghasilkan:**

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

- **Maka, running time dalam membangun BUILD-MAX-HEAP dapat dituliskan dengan**

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

- **Kita dapat membangun max-heap dari array tak terurut dalam waktu linear.**

ALGORITMA HEAPSORT

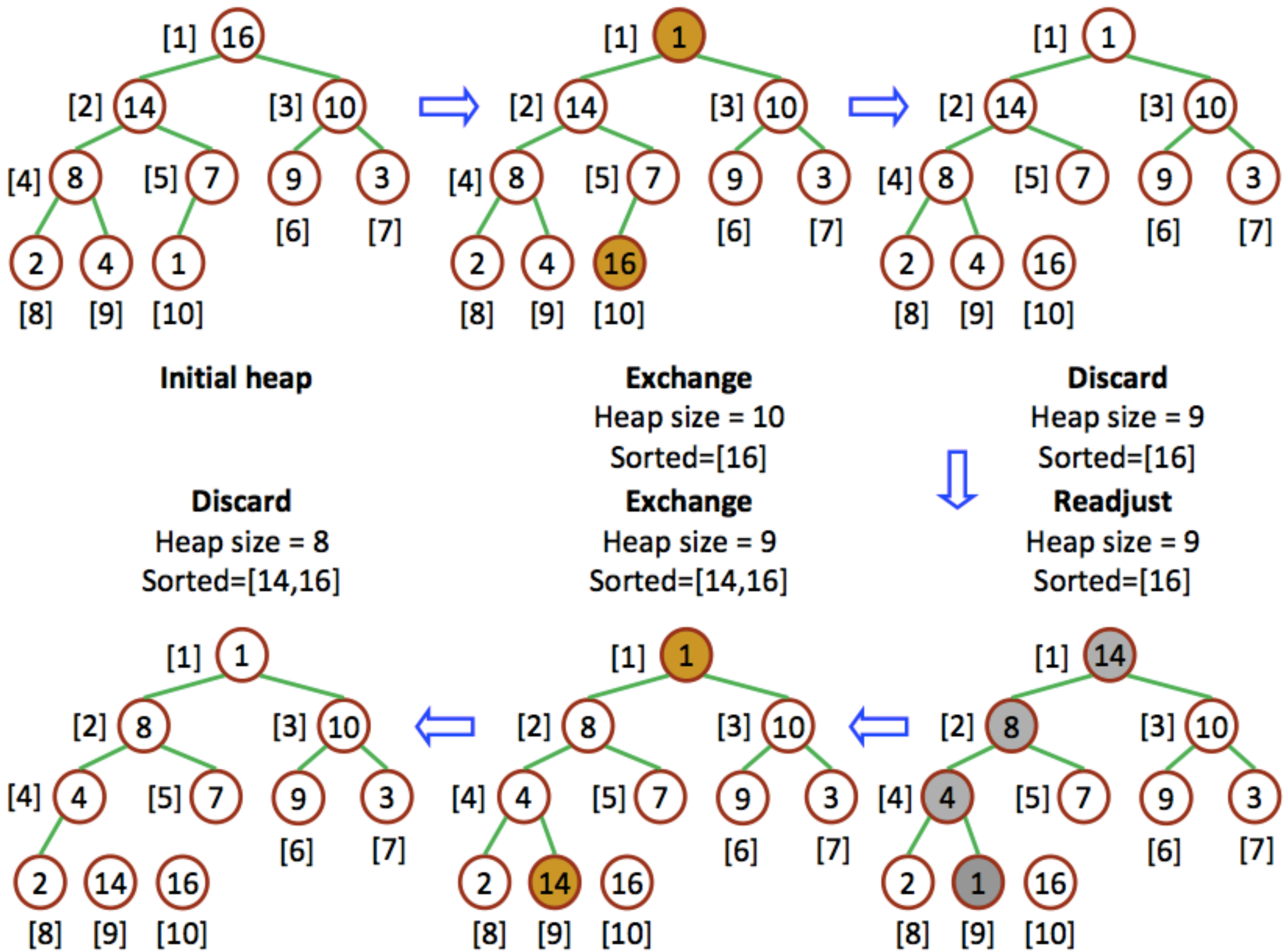
- **Dimulai dengan membuat heap untuk larik A[1 . . *n*].**
- **Karena nilai maksimum elemen disimpan pada A[1], maka elemen tersebut ditukar dengan elemen terakhir *A*[*n*].**
- **Node[n] dihapus dari heap.**
- **Dilakukan pengecekan terhadap properti heap dengan prosedur *M*ax-Heapify.**

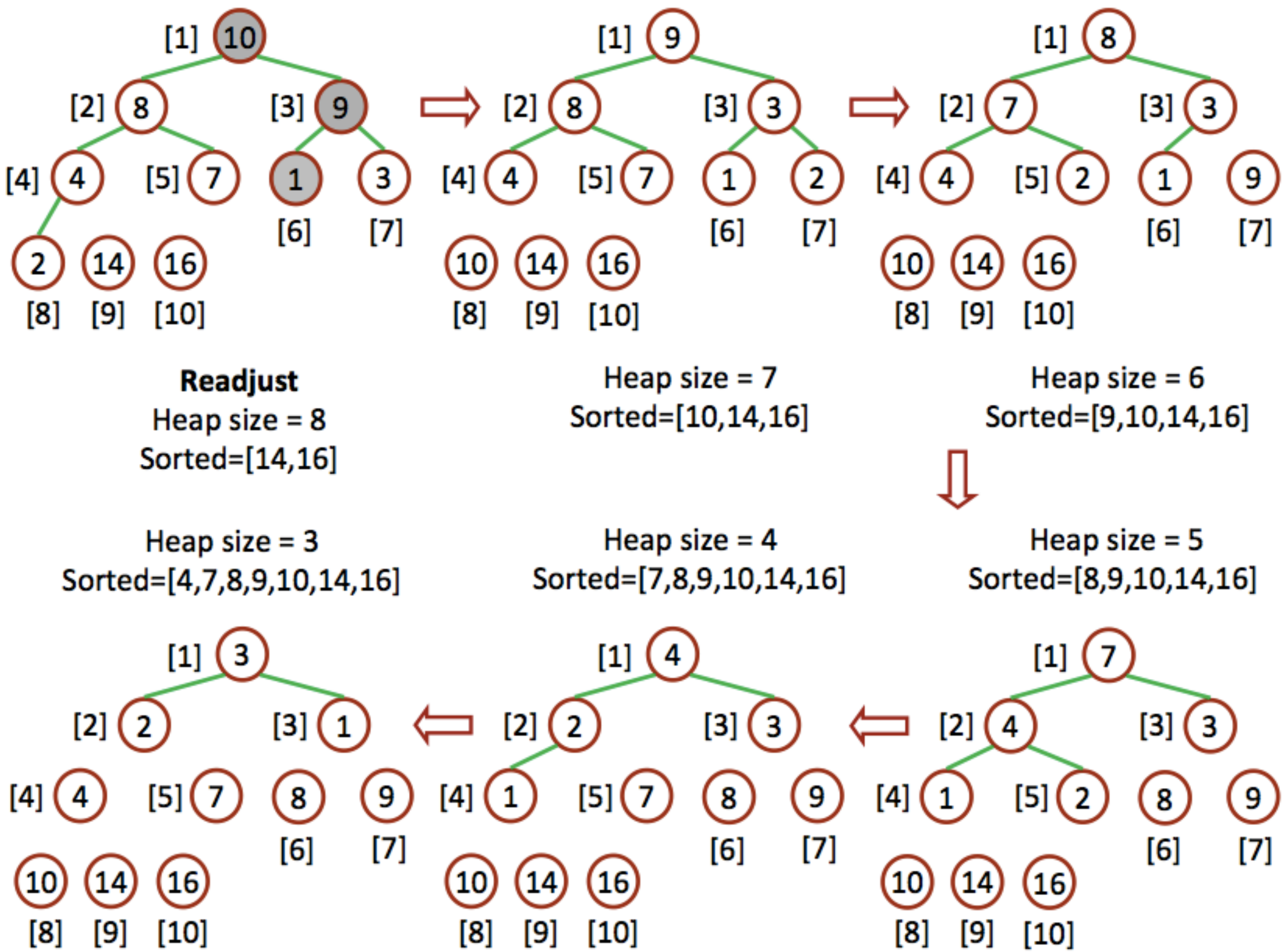
- **Algoritma:**

HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*)
2. for *i* ← *length*[A] downto 2
3. do exchange A[1]↔A[*i*]
4. *heap-size*[A] ← *heap-size*[A] -1
5. MAX-HEAPIFY(*A*, 1)

CONTOH


Initial heap
Exchange
Heap size = 10
Sorted=[16]
Discard
Heap size = 9
Sorted=[16]
Readjust
Heap size = 9
Sorted=[16]
Exchange
Heap size = 9
Sorted=[14,16]
Discard
Heap size = 8
Sorted=[14,16]


Readjust
Heap size = 8
Sorted=[14,16]
Heap size = 7
Sorted=[10,14,16]
Heap size = 6
Sorted=[9,10,14,16]
Heap size = 5
Sorted=[8,9,10,14,16]
Heap size = 4
Sorted=[7,8,9,10,14,16]
Heap size = 3
Sorted=[4,7,8,9,10,14,16]

- **Running time:**
  - Prosedur BuildHeap memerlukan waktu $O(n)$
  - Setiap n -1 panggilan ke Max-Heapify memerlukan waktu O(lg n).
  - Maka total running time O(n lg n)

LATIHAN

- **Apakah larik dengan nilai [23, 17, 14, 6, 13, 10, 1, 5, 4, 12] merupakan heap?**
- **Lakukan heapsort untuk larik: [8, 4, 1, 6, 20, 9, 14, 17]**