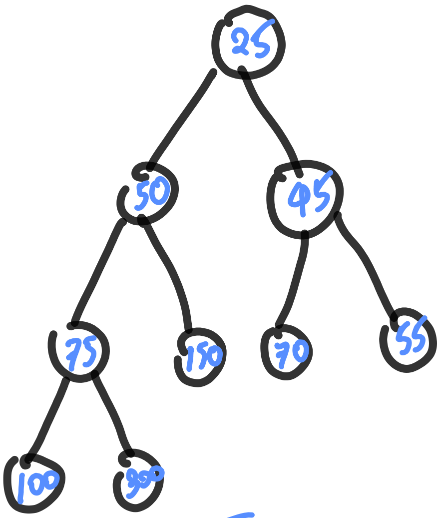
Data Structures and Algorithms

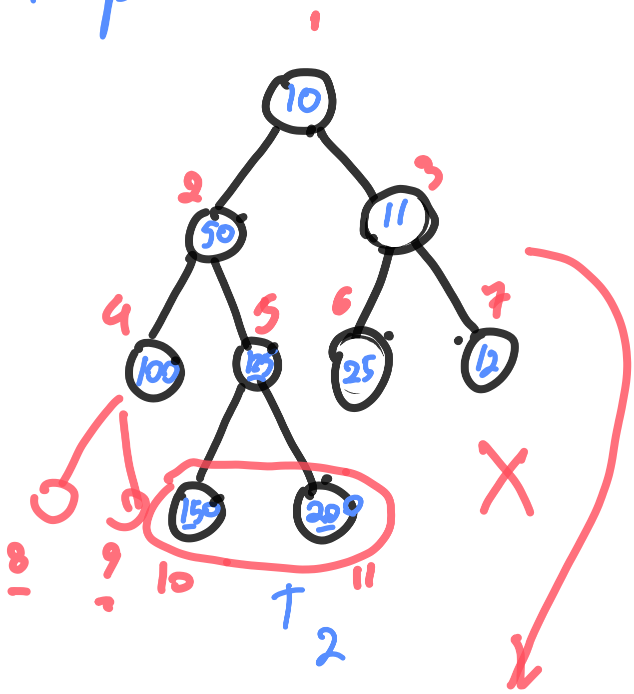
Lecture 23: Heaps

Nopadon Juneam
Department of Computer Science
Kasetsart university

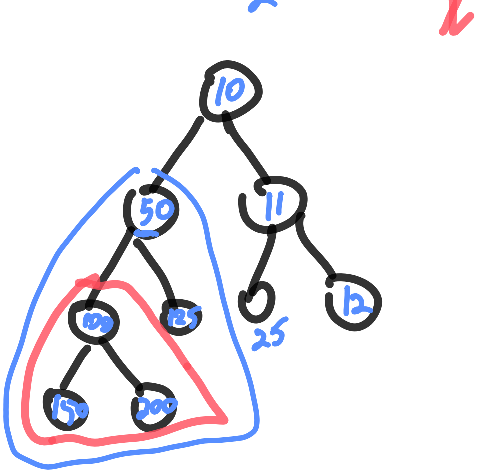
Min-Heap



T_1



T_2



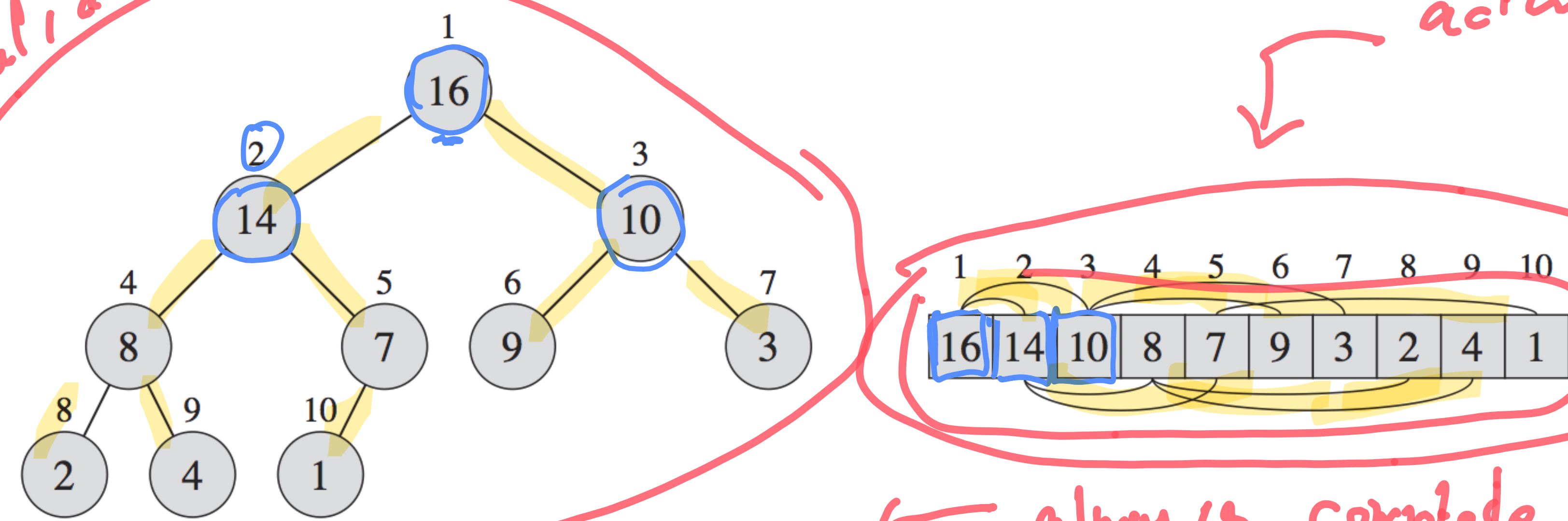
Outlines

- Basics of heaps
  - Heap’s property
- Basic Operations on Heaps
- Applications of Heaps

Heaps (Binary Heaps)

visualization

actual data structure



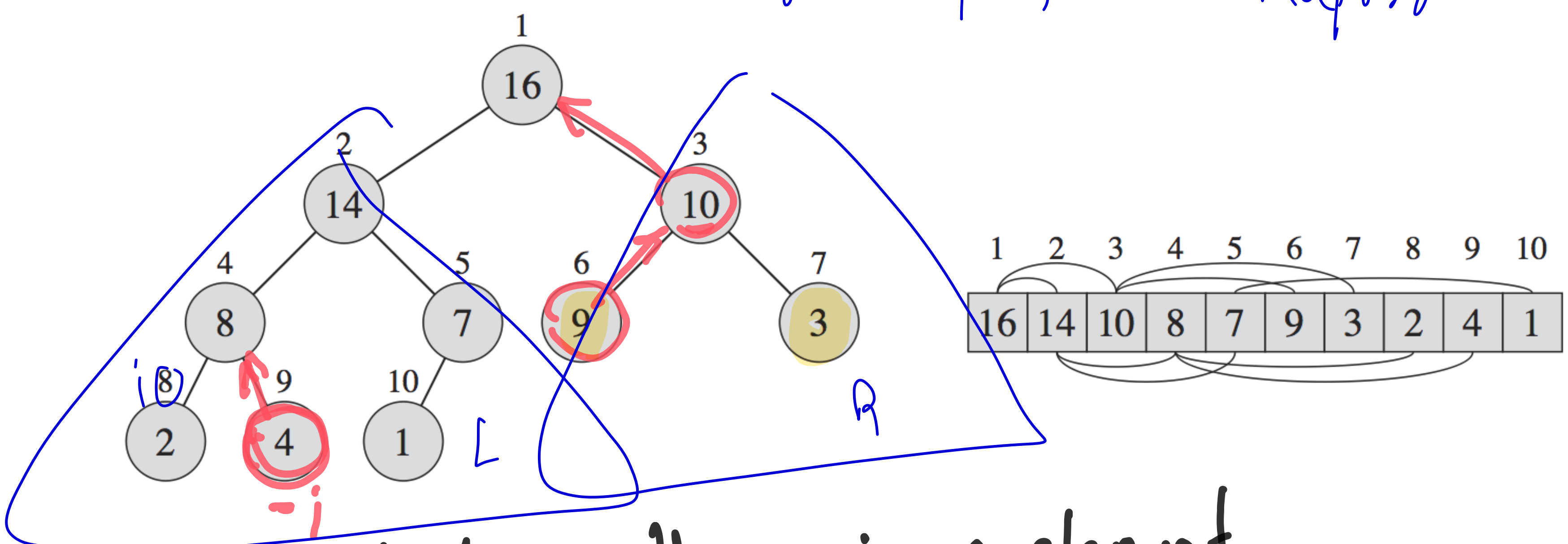
always complete, no skip numbering

- A **heap** is an array visualized as a **complete binary tree**
  - The root of the tree is the element at position i=1 (root is A[1])
  - The parent of an element at position i is the element at element i/2 (parent of A[i] is A[i/2])
  - The left child of an element at position i is the element at position 2i (left child of A[i] is A[2i])
  - The right child of an element at position i is the element at position 2i+1 (right child of A[i] is A[2i+1])
- In other words, a heap is **a complete binary tree based on array-based representation**

Heap = complete binary tree,

Heap's Property

T ต้นไม้ใหญ่เป็น heap L,R ต้องเป็น heap ด้วย



→ root keeps the maximum element

ค่าของ Parent(i) ต้องมากกว่า ค่าของ i

- In **max-heaps**, for all nodes i, excluding the root, A[Parent(i)] ≥ A[i]

  ทุก node i — ยกเว้น root

  → หา Min ที่ root เลย

- In **min-heaps,** for all nodes i, excluding the root, A[Parent(i)] ≤ A[i]. — root ของ subtree เก็บค่า Min ของ subtree นั้นๆ

  → root keeps the minimum element

By induction and transitivity of ≤, the heap's property guarantees that a largest element in a max-heap can be founded at the root. Symmetrically, a minimum element is at the root for a min-heap.

Basic Operations on Heaps

- A heap of n elements is based on a complete binary tree of n nodes. Therefore, its height is always $\log_2(n+1)-1$. All the following basic operations on heaps have complexity proportional to the height of the tree and thus take O(log n) time.

- Basic operations on heaps:

  - Max-Heapify(A, i, n): Maintain the heap's property

  - Build-Max-Heap(A, n): Convert an unordered array A into a max-heap

    Delete root

  - Max-Heap-Delete(A, n): Delete the maximum element from the max-heap

    Insert ตัวใหม่ แต่ยังไม่เป็น heap อยู่

  - Max-Heap-Insert(A, k, n): Insert the new element k into the max-heap

ทำให้เป็น heap

Operation: Max-Heapify(1)

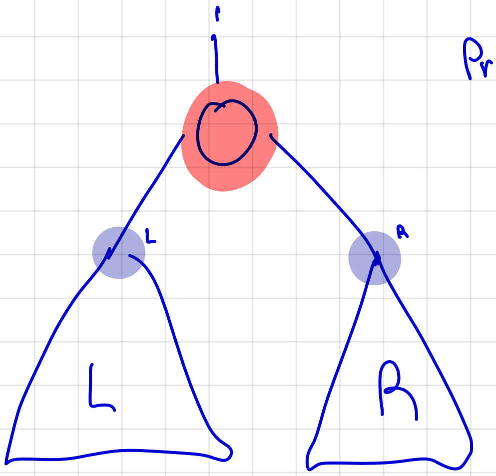
- *Precondition:* assume that the left and right subtrees of *i* are max-heaps, but A[i] might be smaller than its children (the heap's property violates at *i*).

สิ่งที่ได้กลับมา

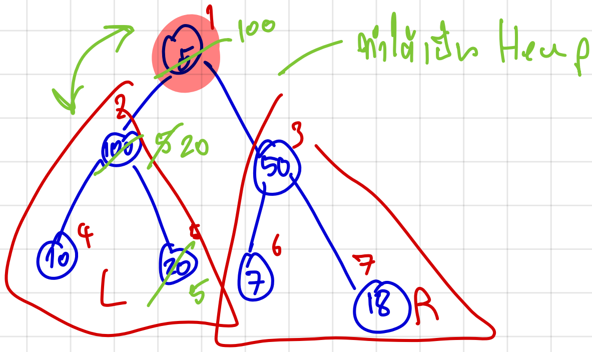
- *Postcondition:* after applying Max-Heapify(*A, i, n*), the subtree rooted at *i* becomes a heap.

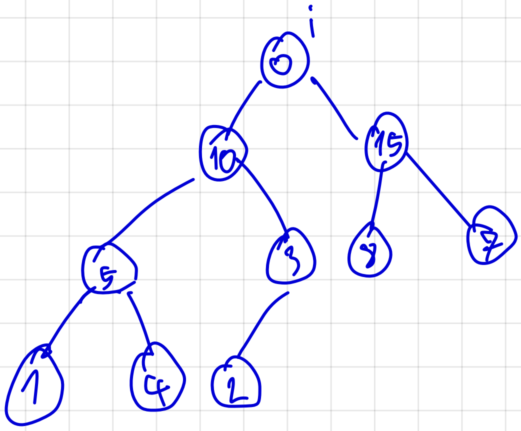
- Max-Heapify(*A*, *i, n*): Maintain the heap's property
  - Compare A[i], A[Left(i)], and A[Right(i)]
  - If necessary, swap A[i] with the larger of the two children to preserve heap property
  - After the swap, the heap's property might still violate. So, we repeat the process of comparing and swapping down the heap, until the subtree rooted at *i* becomes max-heap ทำลง

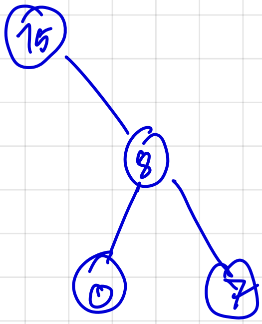
จน subtree ลูกที่ i กลาย
เป็น max-heap



Pre condition : L, R are heaps but the whole tree is not.







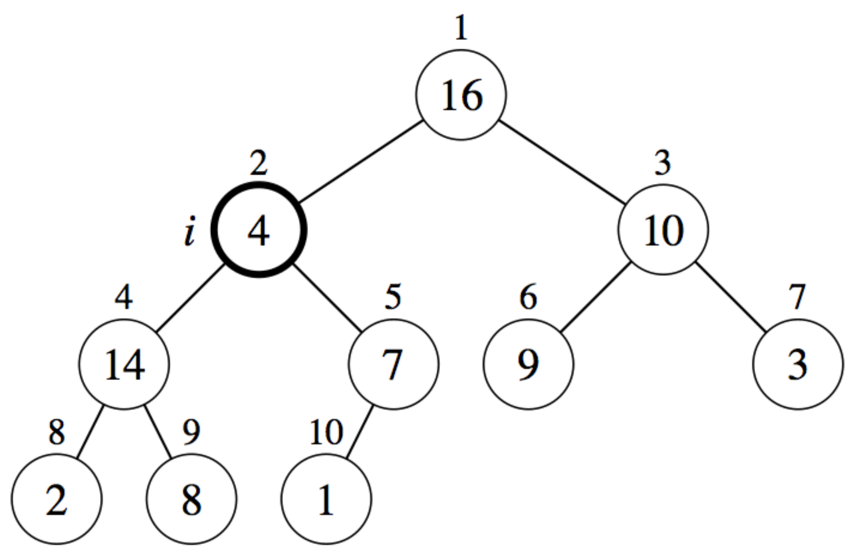
Operation: Max-Heapify(2)

- Max-Heapify(*A*, *i, n*): Maintain the heap's property
  - Compare A[i], A[Left(i)], and A[Right(i)]
  - If necessary, swap A[i] with the larger of the two children to preserve heap property
  - After the swap, the heap's property might still violate. So, we repeat the process of comparing and swapping down the heap, until the subtree rooted at *i* becomes max-heap

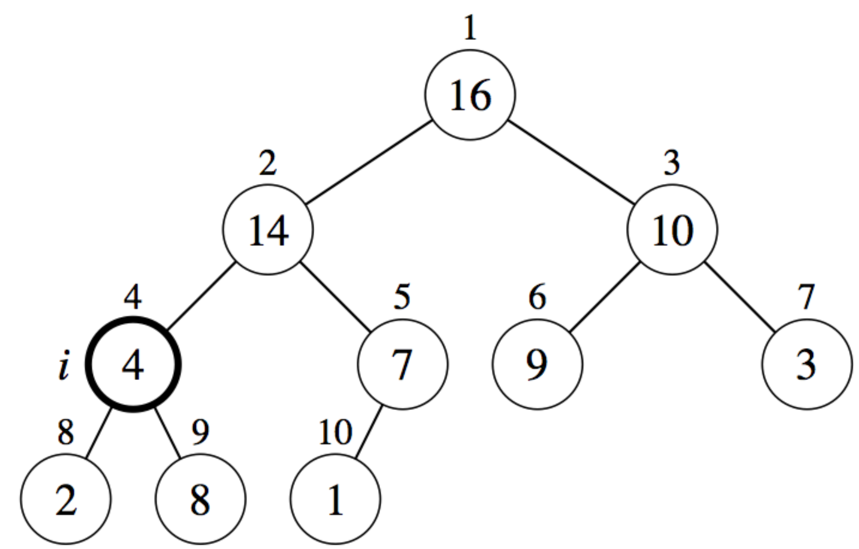
```
Max-Heapify(A, i, n):
    l = Left(i)  //2i
    r = right(i)  //2i+1
    swap_position = i
    if l <= n and A[l] > A[i]:
        swap_position = l
    else if r<= n and A[r] > A[swap_position]:
        swap_position =r
    if swap_position != i:
        swap(A[i], A[swap_position)
        Max-Heapify(A, swap_position, n)
```

Worst: O(log N)

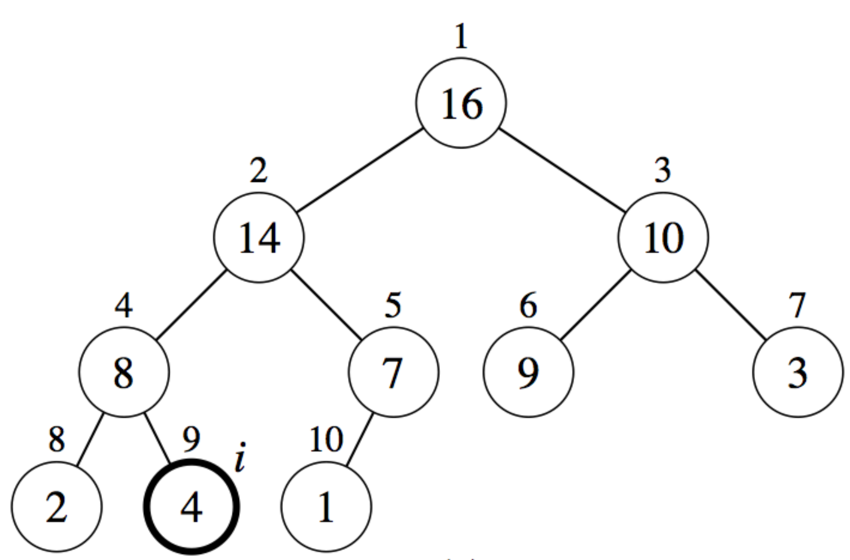
Max-Heapify in Actions



(a)



(b)



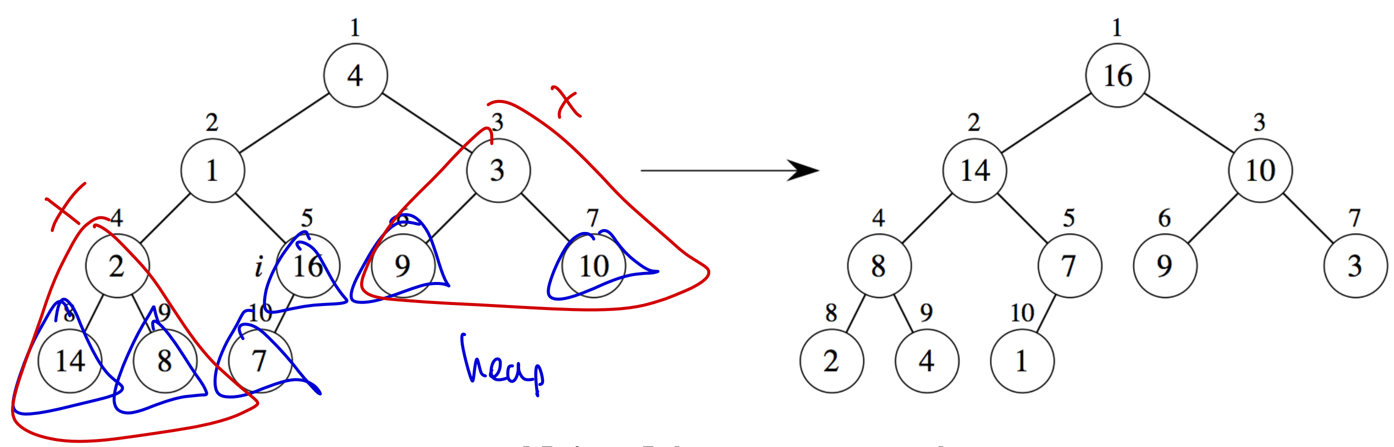
(c)

```
Max-Heapify(A, i, n):
        l = Left(i)
        r = right(i)
        swap_position = i
        if l <= n and A[l] > A[i]:
                swap_position = l
        else if r<= n and A[r] > A[swap_position]:
                swap_position =r
        if swap_position != r:
                swap(A[i], A[swap_position)
                Max-Heapify(A, swap_position, n)
```

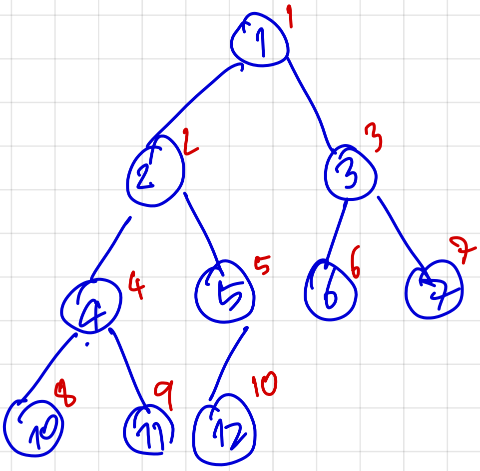
Operation: Build-Max-Heap (1)

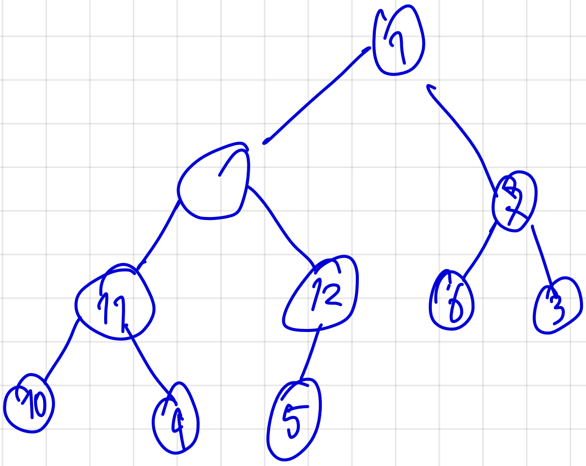
- *Precondition:* an unordered array $A[1..n]$ (A is not a max-heap)
- *Postcondition:* after applying Build-Max-Heap(A, n), the array A[1..n] is a max-heap

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

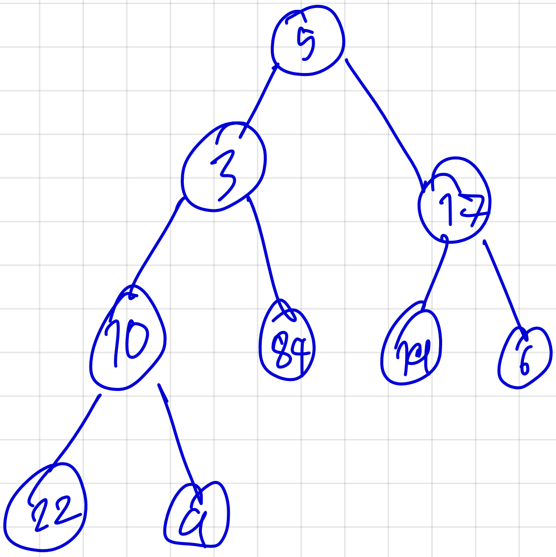


- Build-Max-Heap(A, n): converts array $A[1..n]$ into a max-heap
    - Iteratively apply Max-Heapify on the subtrees starting from the bottom to the top (from the smallest subtrees to the largest ones)

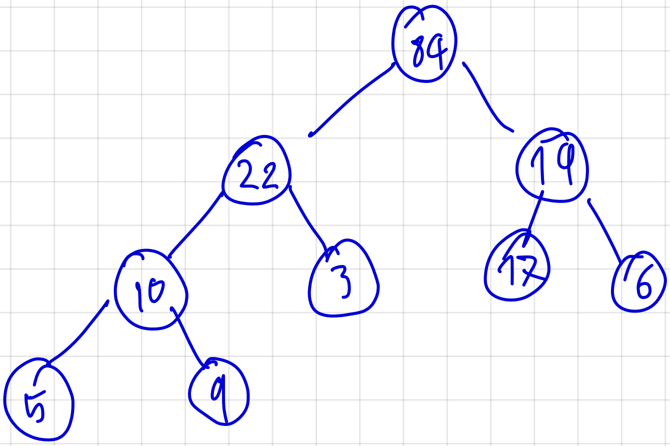

1
2
3
4
5
6
7
10
11
12
8
9
10


1
11
12
6
3
10
4
5

Quiz VI : ให้วาดภาพ complete binary tree ของ heap ที่ได้จากการ convert array A ด้วย Build-Max-Heap

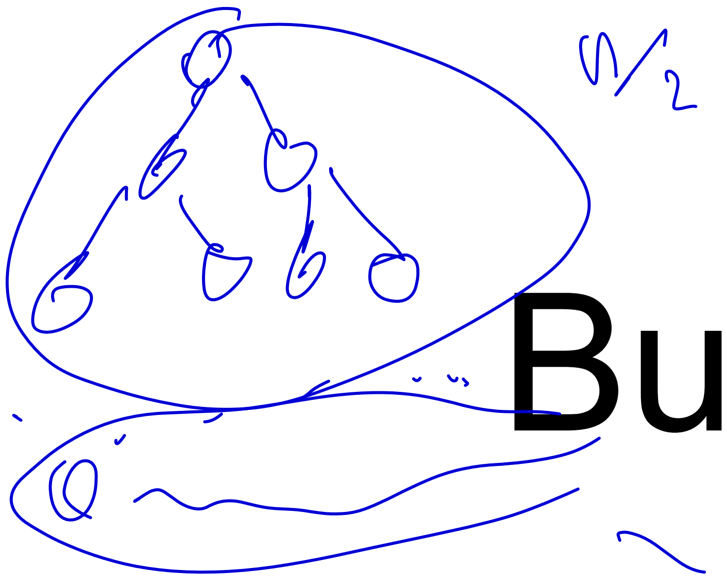


Build-max-heap



กิตติคุณ บุญต่ออุทัย

6510405334



Operation: Build-Max-Heap (2)

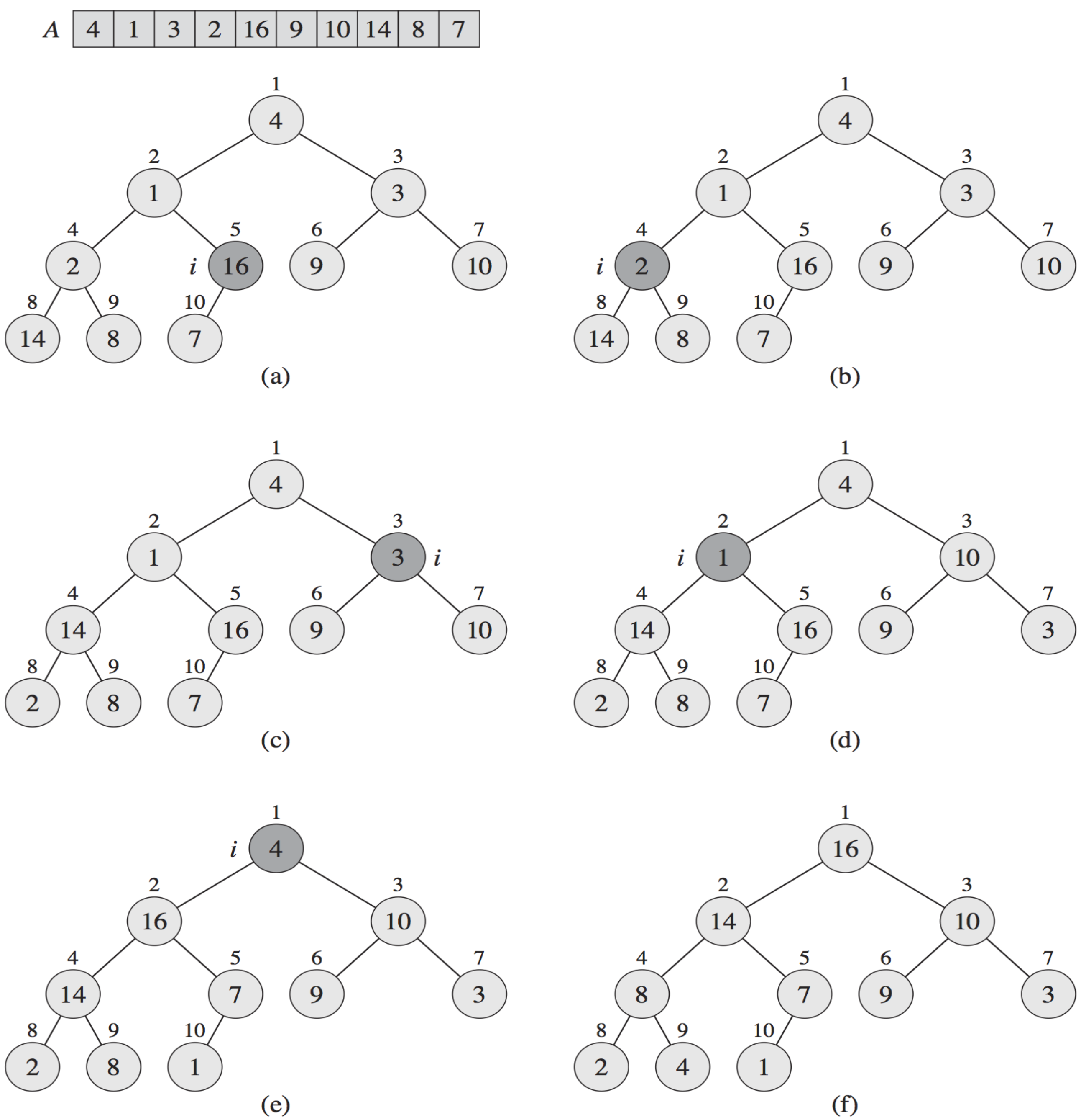
- Build-Max-Heap(*A*, *n*): converts array *A*[*1*..*n*] into a max-heap
  - Iteratively apply Max-Heapify on the subtrees starting from the bottom to the top (from the smallest subtrees to the largest ones)

```
Build-Max-Heap(A, n)
    for i = floor(n/2) down to 1:
        Max-Heapify(A, i, n)
```

start at $i = n/2$
จะเป็น internal nodes
ต้องมี index ระหว่าง
1 ถึง $n/2$

โหนดที่ตำแหน่ง $j = \frac{n}{2} + 1$ จะเป็น
external เสมอ
เพราะว่า ลูกซ้าย $left(i) = 2(\frac{n}{2} + 1) = n + 2$

Build-Max-Heap in Actions




```
Build-Max-Heap(A, n)
        for i = floor(n/2) down to 1:
                Max-Heapify(A, i, n)
```

$O(\log n) \cdot \frac{n}{2}$

$= O(n \log n)$

Operation: Max-Heap-Delete (1)

- *Precondition:* An array A[1..n] is a max-heap with a maximum element *r* found at the root.
- *Postcondition:* The element *r* is removed from *A*. The array *A*[1..*n*-1] is a max heap after deletion
- Max-Heap-Delete(*A, n*): Delete a maximum element *r* from the max-heap
  - Swap the root (maximum element *r*) with the last element (*A*[*n*])
  - Delete the last element *A*[*n*]
  - After the swap, the heap's property might violate at the new root *r'*. To fix this, we apply Max-Heapify on the tree rooted at *r'*

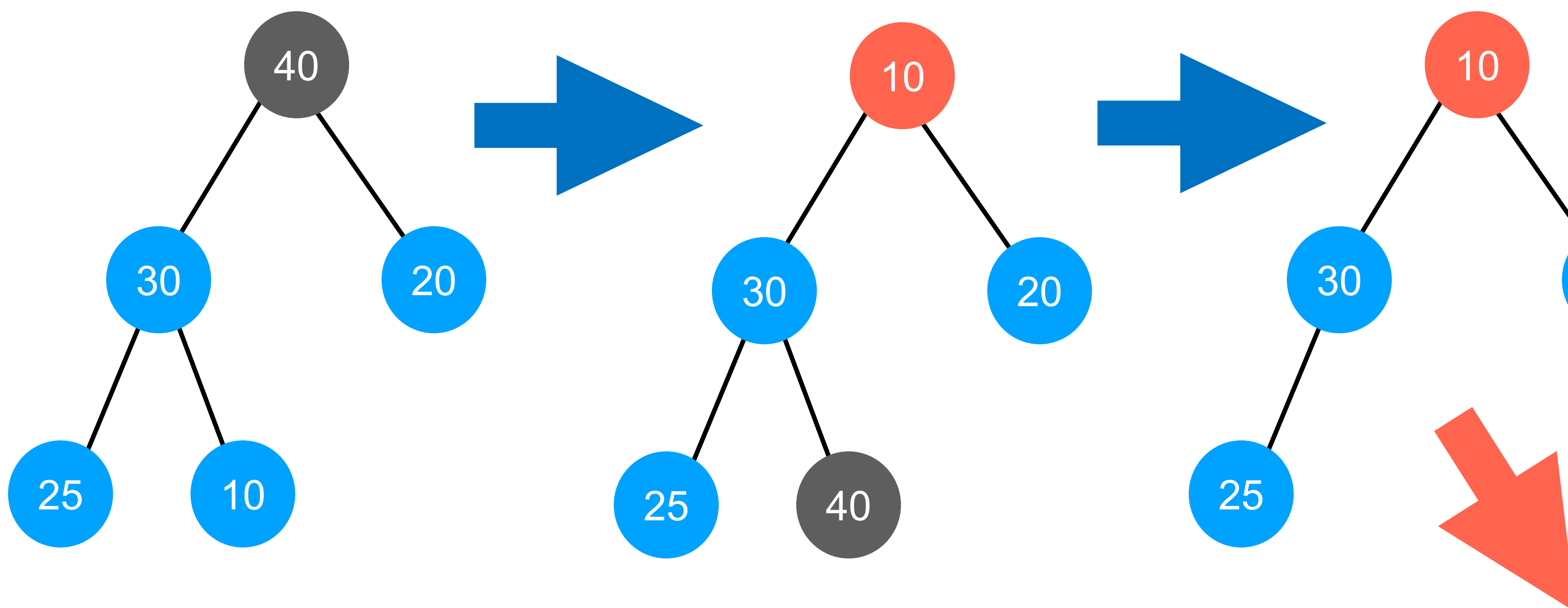
Operation: Max-Heap-Delete (2)

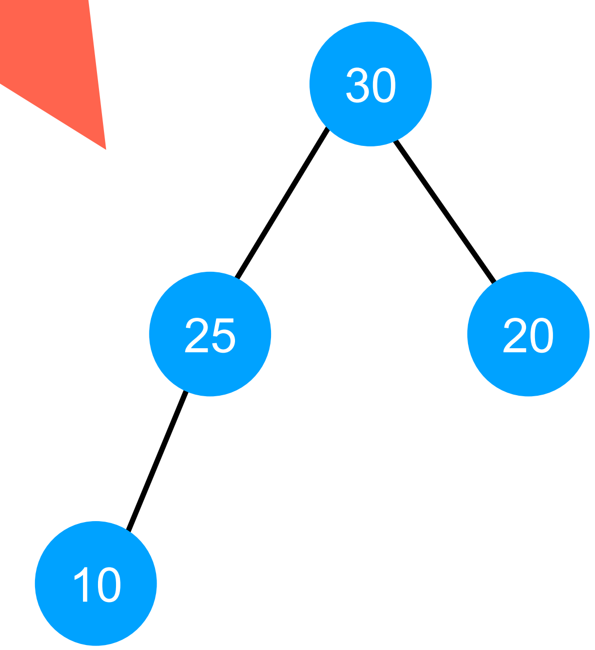
- Max-Heap-Delete(*A*, *n*): Delete a maximum element *r* from the max-heap
  - Swap the root (maximum element *r*) with the last element (*A*[*n*])
  - Delete the last element *A*[*n*]
  - After the swap, the heap's property might violate at the new root *r'*. To fix this, we apply Max-Heapify on the tree rooted at *r'*

```
Max-Heap-Delete(A, n):
    swap(A[1], A[n])        เปลี่ยน Root
    deleteLastElementFromArray(A, n)
    // call the above operation if you ever need to actually delete r
    Max-Heapify(A, 1, --n)
```

O(log n)

Max-Heap-Delete in Actions






```
Max-Heap-Delete(A, n):
        swap(A[1], A[n])
        deleteLastElementFromArray(A, n)
        // call the above operation if you ever need to actually delete r
        Max-Heapify(A, 1, --n)
```

Operation: Max-Heap-Insert (1)

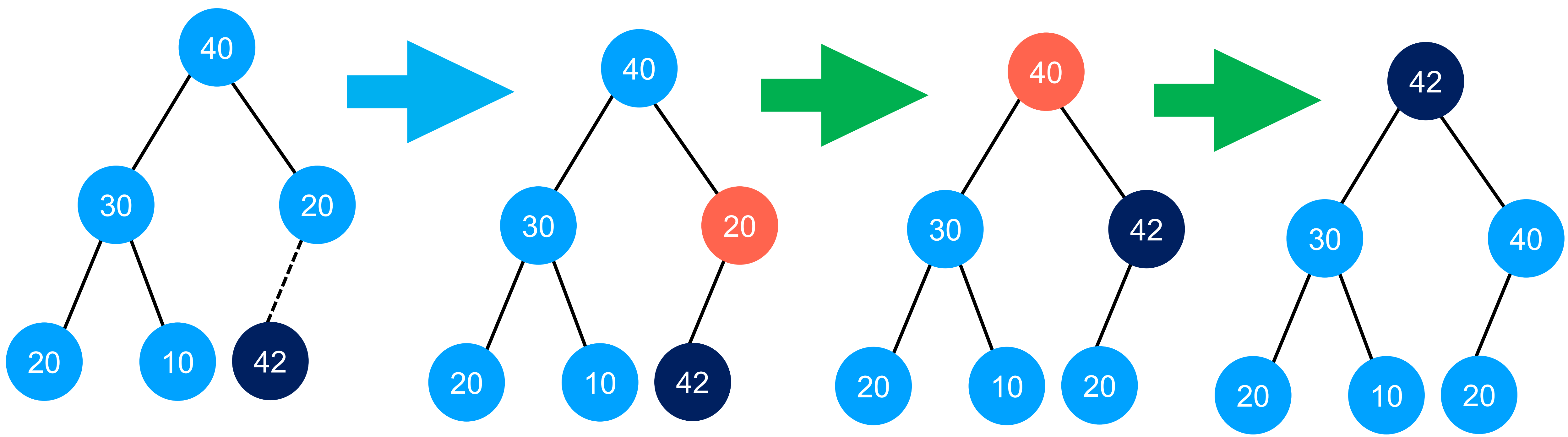
- *Precondition:* An array A[1..n] is a max-heap.
- *Postcondition:* The new element *k* is inserted into *A*. After insertion, the array *A*[1..*n*+1] is a max heap
- Max-Heap-Insert(*A*, *k*, *n*): Insert the new element k into the max heap
  - Increase the heap size by one
  - Insert the new element *k* at the end of array *A*
  - Heapify the new element *k* following a bottom-up approach (that is, recursively swapping *k* with its parent upward the tree as necessary to restore the heap's property)

Operation: Max-Heap-Insert (2)

- Max-Heap-Insert(*A*, *k*, *n*): Insert the new element k into the max heap
    - Increase the array size by one, an insert the new element *k* at the end of array *A*
    - Heapify the new element *k* following a bottom-up approach (that is, iteratively swapping *k* with its parent upward the tree as necessary to restore the heap's property)

Max-Heap-Insert(A, k, n)
    InsertToArray(A, k)
    i = n+1
    while i>1 and A[i] ~~<~~ > A[i/2]:
        swap(A[i], A[i/2])
        i = i/2

Max-Heap-Insert in Actions




```
Max-Heap-Insert(A, k, n)
        InsertToArray(A, k)
        i = n+1
        while i>1 and A[i] < A[i/2]:
                swap(A[i], A[i/2])
                i = i/2
```

Complexity of Operations on Heaps

Operations	Complexity
Max-Heapify	$O(\log n)$
Build-Max-Heap	$O(n \log n)$, but O(n) is the tighter bound
Max-Heap-Insert	$O(\log n)$
Max-Heap-Delete	$O(\log n)$
	Remark: All these operations have complexity that is proportional to the height of the complete binary tree, which is always $\log_2(n+1)-1$

Basic Applications of Heaps

- **Heapsort**: a sorting algorithm using heap which runs in $O(n \log n)$ time.
- **Priority queue:** an implementation of “priority queue” using heap offers $O(\log n)$ time complexity for most operations.