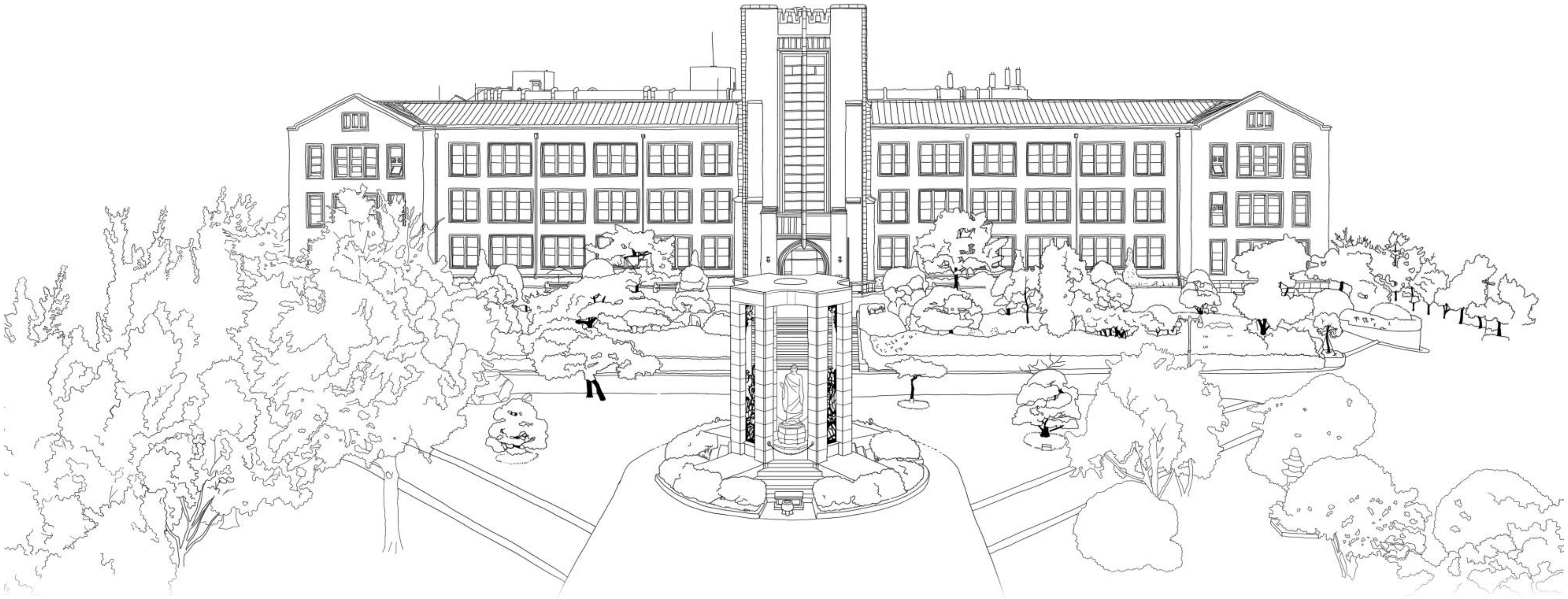


Introduction to Algorithms

DongYeon Kim
Department of Multimedia Engineering
Dongguk University

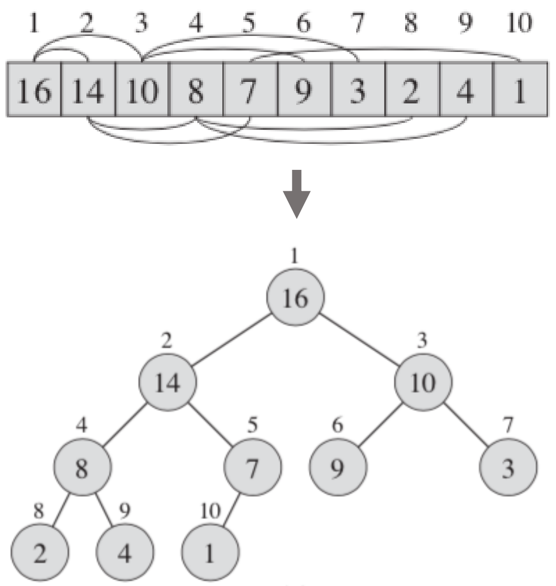
Contents

❖ Chapter.2 Sorting and Order Statistics

❖ Heap

- Introduction
  - ➢ Running time of heapsort: $O(n \lg n)$
  - ➢ In-place algorithm : only a constant number of array elements are stored outside the input array
  - ➢ New algorithm design technique : using a data structure (heap)

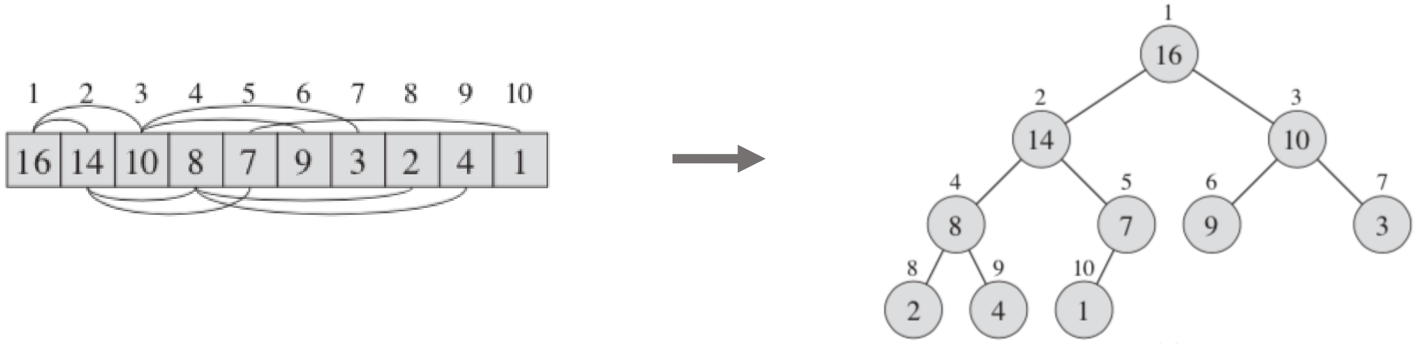
- Definition



  - ➢ Array object that we can view as a nearly complete binary tree
  - ➢ Filled from the left up to a point
  - ➢ Attributes
    - ✓ $A.length$: gives the number of elements in the array (10)
    - ✓ $Heap - size$: represents how many elements in the heap (10)
    - ✓ $0 \leq A.heap - size \leq A.length$

❖ Heap

- **Definition**
  - ➢ Index function
    - ✓ i : index of the nodes
    - ✓ $A[1]$: root of the tree
    - ✓ $PARENT(i) = \lfloor i/2 \rfloor$
    - ✓ $LEFT(i) = 2i$
    - ✓ $RIGHT(i) = 2i + 1$



- **Kinds of binary heaps**
  - ➢ $Max - heap : A[PARENT(i)] \geq A[i]$
    - ✓ Largest value is stored in root node
    - ✓ Heapsort uses max-heap
  - ➢ $Min - heap : A[PARENT(i)] \leq A[i]$
    - ✓ Smallest value is stored in root node

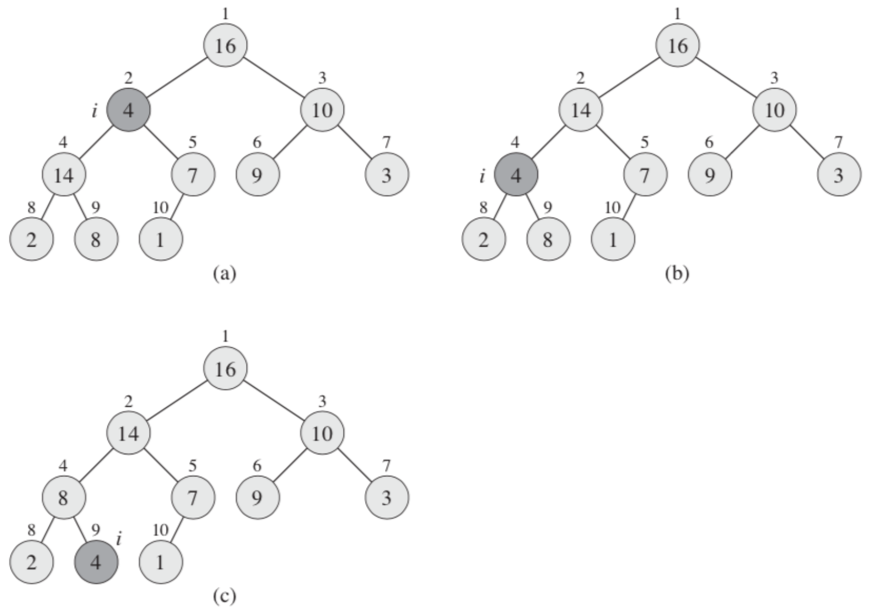
Heap

- **Operation of heap**
  - Height of node : the number of edges on the longest simple downward path from node to a leaf
  - Height of heap : height of root node
  - Height of n elements–heap : $\lfloor \lg n \rfloor$ (complete binary tree)

  - The basic operations on heaps run in time proportional to the height of the tree : $O(\lg n)$
    - ✓ $MAX-HEAPIFY$: maintain the max–heap property – $O(\lg n)$
    - ✓ $BUILD-MAX-HEAP$: produce a max–heap from unordered input array – linear time
    - ✓ $HEAPSORT$: sorts an unordered array in place – $O(n\lg n)$
    - ✓ $MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY,\ HEAP-MAXIMUM$: to implement a priority queue – $O(\lg n)$

❖ Maintaining the Heap-property

- Definition
  - ➢ To maintain the max-heap property, we call $MAX - HEAPIFY$
  - ➢ Assume that the binary trees rooted at $LEFT(i)$ and $RIGHT(i)$ are max-heaps, but $A[i]$ might be smaller than its children



The process of $MAX - HEAPIFY$

- ➢ (a) : $A[2]$ violates the max-heap property

- ➢ (b-1) : find the larger children between $A[2]$ and $A[4]$
- ➢ (b-2) : exchange $A[2]$ with $A[4]$
- ➢ (b-3) : $A[4]$ violates the max-heap property

- ➢ (c-1) : find the larger children between $A[8]$ and $A[9]$
- ➢ (c-2) : exchange $A[8]$ with $A[9]$
- ➢ (c-3) : $A[9]$ is fixed up

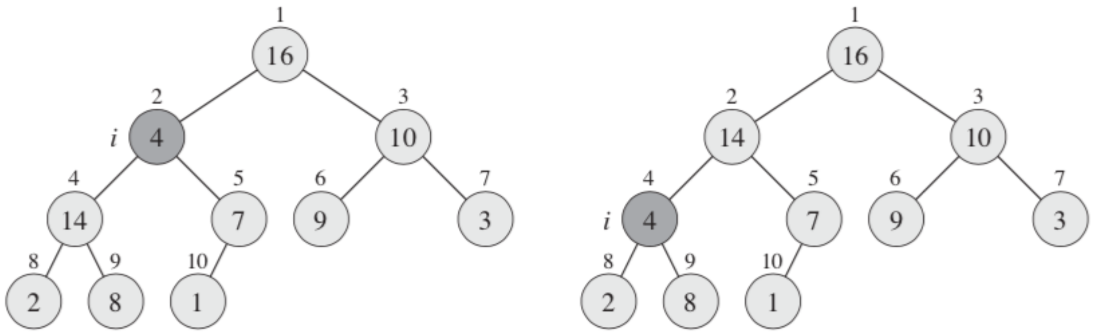
❖ Maintaining the Heap-property

- Pseudocode
  - ➢ Input : array A, index i

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

The pseudocode for $MAX - HEAPIFY$

- ➢ 1~7 : find the largest values in $A[i], A[LEFT(i)], A[RIGHT(i)]$
- ➢ 8~10 : check heap property is violated
  - ✓ If $A[i]$ is the largest → terminates
  - ✓ Otherwise $A[i]$ violates the heap property → swap and recursively call $MAX - HEAPIFY$

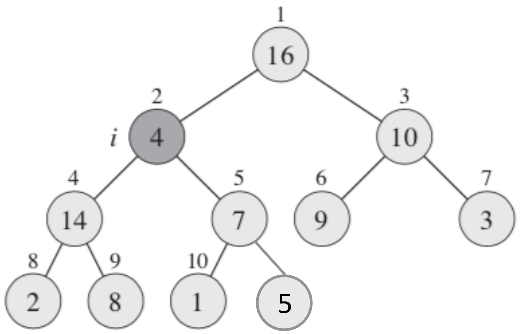


(a) (b)

❖ Maintaining the Heap-property

- The running time of $\boldsymbol{MAX-HEAPIFY}$
  - ➢ Subtree of size n
  - ➢ Exchange $A[i]$ with $A[LEFT(i)]$ or $A[RIGHT(i)]$: $\theta(1)$
  - ➢ Run $MAX-HEAPIFY$ on a subtree : $T(2n/3)$
    - ✓ Children's subtrees each have size at most 2n/3 – the worst case occurs
    - ✓ Worst case : exactly half full
  - ➢ Running time of MAX-HEAPIFY

$$T(n) \leq T\left(\frac{2n}{3}\right) + \theta(1)$$



(a)

❖ Maintaining the Heap-property

- The running time of $\boldsymbol{MAX - HEAPIFY}$
  - Running time of MAX-HEAPIFY

$$T(n) \leq T\left(\frac{2n}{3}\right) + \theta(1)$$

  - Solve this recurrence by using case 2 of the master theorem
    - ✓ In the form of equation : $T(n) = aT(n/b) + f(n)$

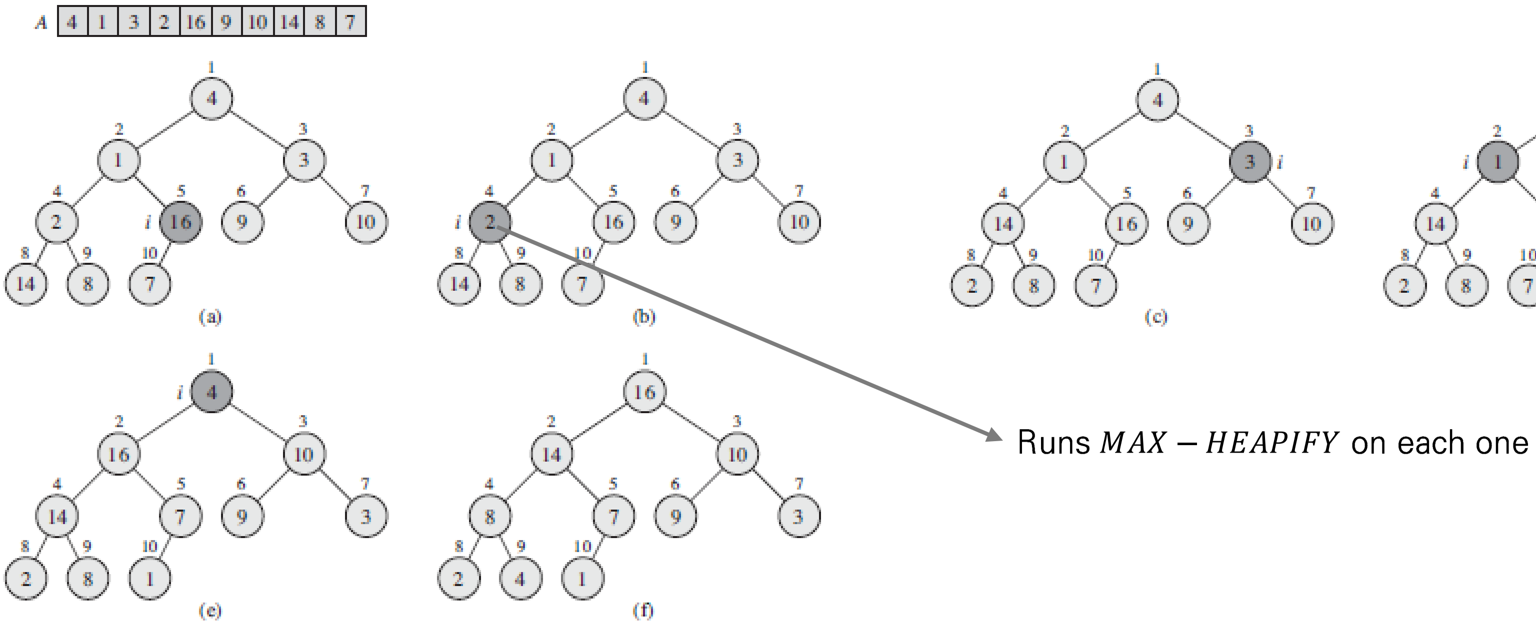
1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

    - ✓ In this case, $a = 1$ $b = 3/2$ $f(n) = \theta(1) = \theta\left(n^{\log_{3/2} 1}\right) = \theta(1)$
    - ✓ $T(n) = \theta\left(n^{\log_{3/2} 1} \lg n\right) = \theta(\lg n)$

  - Height of n elements-heap(h) : $\lg n$
    - ✓ $\theta(\lg n) = \theta(h)$

❖ Building a heap

- Definition
  - ➢ Using $MAX\_HEAPIFY$ in a bottom–up manner to convert an array $A[1..n]$ into a $\max - heap$
  - ➢ The elements in the subarray $A[(\lfloor n/2 \rfloor + 1)..n]$ are all leaves of the tree
    - ✓ In this case $N = 10$, $A[6, \dots, 10]$ are all leaves – already satisfy max–heap
  - ➢ The procedure $BUILD - MAX - HEAP$ goes through the remaining nodes of the tree



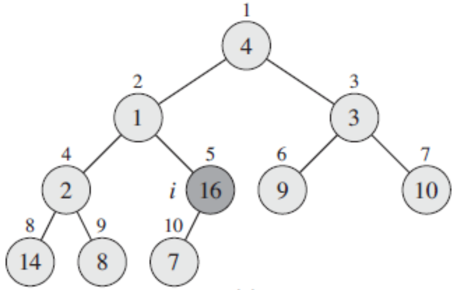
❖ Building a heap

- **Pseudocode**
  - ➢ Input : array A

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

  - ➢ 2~3 : compute $MAX - HEAPIFY$ in a bottom-up manner, starting from remaining nodes (not leaves node)

- **Loop invariant of $\boldsymbol{BUILD - MAX - HEAP}$**
  - ➢ At the start of each iteration of the for loop of lines 2–3, each node $i + 1, i + 2, \dots, n$ is the root of a max-heap
  - ➢ **Initialization**
    - ✓ Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$, each node $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2, \dots, n$ is a leaf and is thus the root of a trivial max-heap

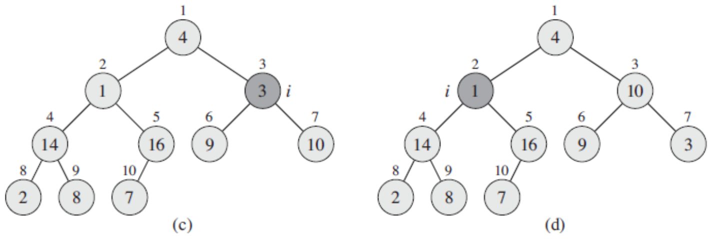


(a)

Initial state

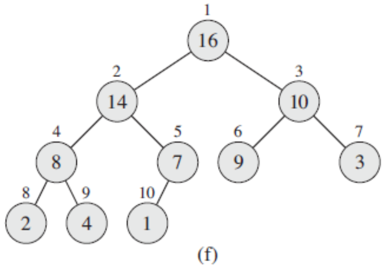
❖ Building a heap

- Loop invariant of $\boldsymbol{BUILD-MAX-HEAP}$
  - ➢ At the start of each iteration of the for loop of lines 2–3, each node $i+1, i+2, \dots, n$ is the root of a max-heap
  - ➢ Maintenance
    - ✓ Children of node i are numbered higher than i
    - ✓ By the loop invariants, They are both roots of max-heaps
    - ✓ $MAX-HEAPIFY(A, i)$ preserves the property at node i



Maintenance state

  - ➢ Termination
    - ✓ At termination $i = 0$
    - ✓ By the loop invariant, each node $1, 2, \dots, n$ is the root of a max-heap



Termination state

❖ Building a heap

- Upper bound running time of $\boldsymbol{BUILD-MAX-HEAP}$
  - ➢ As we computed, $MAX-HEAPIFY$ cost $O(\lg n)$
  - ➢ $BUILD-MAX-HEAP$ makes $O(n)$
    - ✓ Line 2~3 iterate $n/2$ terms
  - ➢ The upper bound running time : $O(n\lg n)$

BUILD-MAX-HEAP(A)

```
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

- Tighter bound running time of $\boldsymbol{BUILD-MAX-HEAP}$
  - ➢ n elements heap has height $\lfloor \log n \rfloor$
  - ➢ At height h, there are $\lceil n/2^{h+1} \rceil$ nodes

The number of node at level h

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

Iterates whole levels

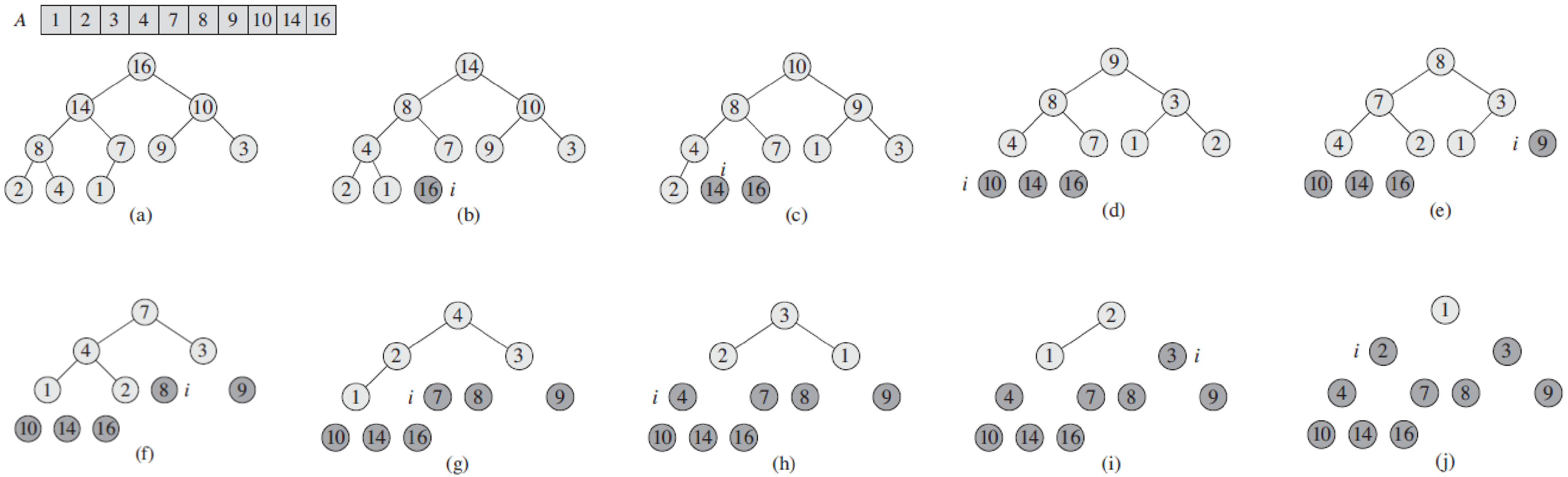
Running time of $MAX-HEAPIFY$

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

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

❖ The heapsort algorithm

- Definition
  - Start by using $BUILD-MAX-HEAP$ to build a max-heap on the unsorted input array $A[1..n]$
  - The maximum element of the array is stored at the root $A[1]$
  - Put the maximum element into final position by exchanging with $A[n]$
  - Discard node n → (heap_size – 1) → check new root element satisfy max-heap $property$ ($MAX-HEAPIFY$)



Heapsort

❖ The heapsort algorithm

- Pseudo code
  - Input : unordered array A

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

  - 1 : build a max−heap using $BUILD - MAX - HEAP$
  - $3\sim4$: exchange root($A[1]$) with $A[n]$ → resize the heap
  - 5 : rebuild the max−heap

- Running time of $\boldsymbol{HEAPSORT}$
  - $BUILD - MAX - HEAP$: $O(n)$
  - For loop iterate n−1 terms
    - ✓ $Call\ MAX - HEAPIFY$: $O(\lg n)$
  - Total running time : $(n-1)\ O(\lg n) + O(n) = O(n \lg n)$

❖ Priority queues

- Introduction
  - ➢ Heap data structure itself has many uses
  - ➢ Focus on the most popular applications of a heap : **priority queue**

- Definition of a priority queue
  - ➢ Data structure for maintaining a set S of elements, each with an associated value called a **key**
  - ➢ A max-priority queue operations
    - ✓ $\boldsymbol{INSERT(S,x)}$: insert the element x into the set S
    - ✓ $\boldsymbol{MAXIMUM(S)}$: returns the element of S with the largest key
    - ✓ $\boldsymbol{EXTRACT-MAX(S)}$: removes and return the element of S with the largest key
    - ✓ $\boldsymbol{INCREASE-KEY(S,x,k)}$: increases the value of element x's key to the new value k
  - ➢ Now, implement these operations using heap data structure

❖ Priority queues

- ***MAXIMUM*(*S*)**
  - The procedure $HEAP - MAXIMUM$ implements the $MAXIMUM$ operation
  - Takes $\theta(1)$ time

```
HEAP-MAXIMUM(A)
1  return A[1]
```

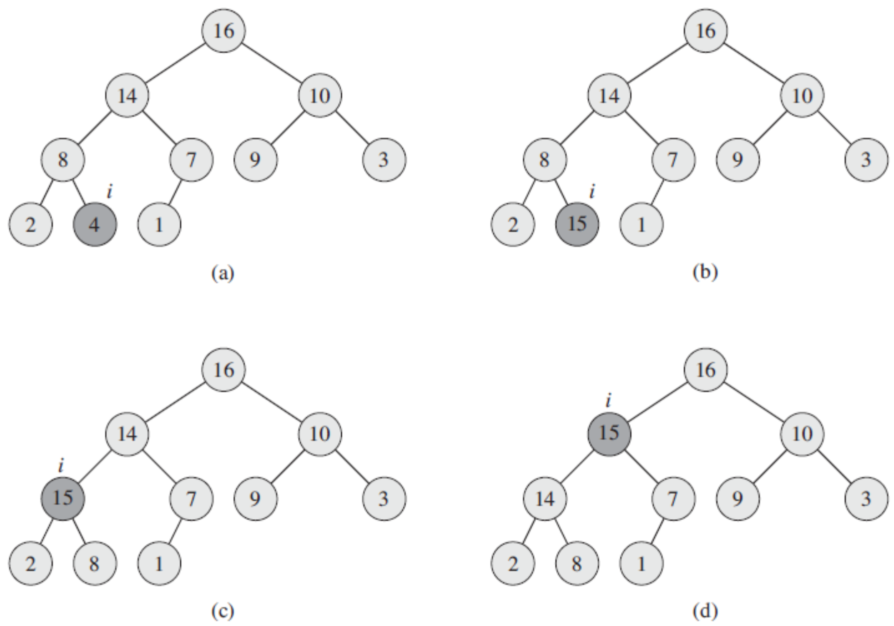
- ***EXTRACT − MAX*(S)**
  - The procedure $HEAP - EXTRACT - MAX$ implements the $EXTRACT - MAX$ operation
  - Similar to $HEAPSORT$ loop procedure
  - Takes $O(\lg n)$ time for $MAX - HEAPIFY$

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2      error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size - 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

- ✓ 3~4 : save the largest element ($A[1]$) & swapping with $A[n]$
- ✓ 5~6 : rebuild the max−heap
- ✓ 7 : Return the largest value

❖ Priority queues

- $\boldsymbol{INCREASE - KEY(S, x, k)}$
  - ➢ The procedure $HEAP - INCREASE - KEY$ implements the $INCREASE - KEY$ operation
  - ➢ Index i node want to increase key to a new value → violates max-heap property



- ✓ (a) : update the i node's key to a new value
- ✓ (b) : new key violates the max-heap property
- ✓ (c) : exchange with parents
- ✓ (d) : iterate (c) process toward the root until finding a proper place for newly increased key

Heapsort

❖ Priority queues

- $\boldsymbol{INCREASE - KEY(S, x, k)}$

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

✓ 3 : change key value of node i

✓ 4~6 : exchange with its parent until find proper place

➢ The running time of $HEAP - INCREASE - KEY$: $O(\lg n)$

✓ Find proper place of newly increased key → Traverse from node i to root

✓ The height of the tree : $\lg n$

❖ Priority queues

- $\boldsymbol{INSERT - KEY(S, x, k)}$
  - ➢ The procedure $MAX - HEAP - INSERT$ implements the $INSERT$ operation

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = -∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

✓ 1 : expand the heap-size

✓ 2 : add to the tree a new leaf whose key is $-\infty$

✓ 3 : find proper place of new node using $HEAP - INCREASE - KEY$

  - ➢ The running time of $MAX - HEAP - INSERT$: $O(\lg n)$
    - ✓ $HEAP - INCREASE - KEY$

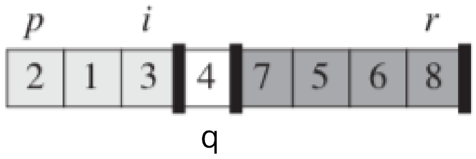
- Conclusion
  - ➢ A heap can support any priority-queue's operation in $O(\lg n)$ time

Contents

❖ Chapter.2 Sorting and Order Statistics

❖ Description of Quicksort

- Introduction
  - ➢ Worst-case running time : $\theta(n^2)$
  - ➢ Average-case running time : $\theta(n\lg n)$
  - ➢ In-place sorting algorithm
- Divide-and-Conquer
  - ➢ Quicksort applies the divide-and-conquer paradigm
  - ➢ Divide
    - ✓ Partition the array $A[p..r]$ into two subarray $A[p..q-1]$ and $A[q+1..r]$
    - ✓ Each element of $A[p,,q-1]$ is less than or equal to $A[q]$
    - ✓ Compute the index q as part of partitioning procedure
  - ➢ Conquer
    - ✓ Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort
  - ➢ Combine
    - ✓ Because the Subarrays are already sorted, no work is needed to combine



Quicksort

❖ Description of Quicksort

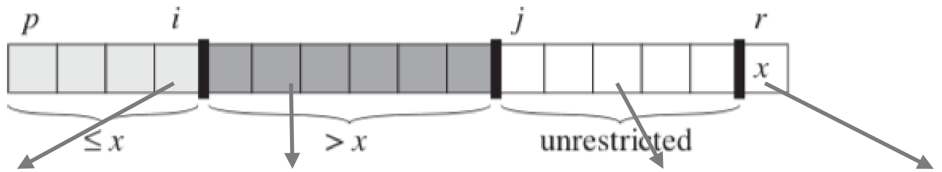
- Procedure of quicksort

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

✓ 2 : divide the unsorted array A using $PARTITION$

✓ 3~4 : conquer the subarray by recursive call

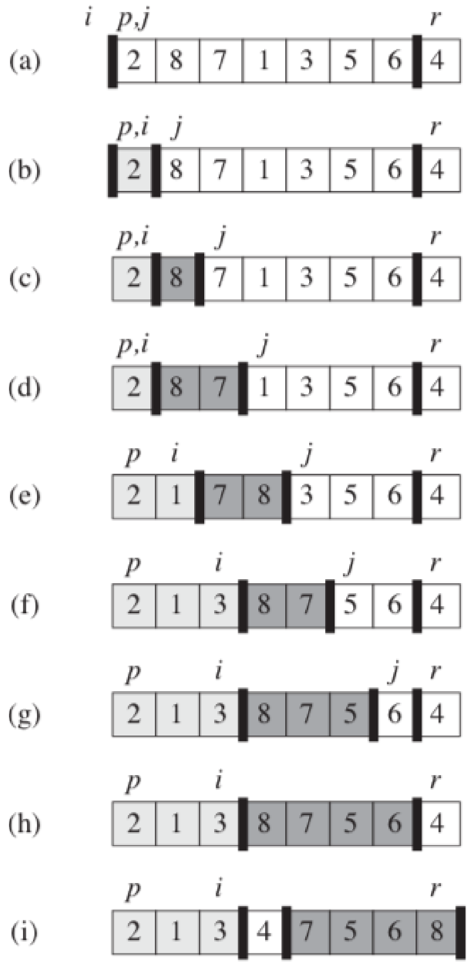
- Partitioning the array
  - ➢ The key procedure of the quicksort algorithm
  - ➢ Rearrange the subarray $A[p..r]$
  - ➢ Select an element $x = A[r]$ as a pivot element which to partition the subarray
    - ✓ Partition the array into four regions
    - ✓



Lower than x higher than x Not arranged Pivot element

❖ Description of Quicksort

- Partitioning the array



- ➢ (a) : initial state, pivot element is $A[r]$
- ➢ (b) : the value 2 is "swapped with itself" and put in the partition of smaller values
- ➢ (c) (d) : increase index j, because value 8, 7 is larger than pivot, added to the partition of larger values
- ➢ (e) : swap $A[i]$ and $A[j]$, value 1 and 8
- ➢ (f) : swap value 3 and 7
- ➢ (g) (h) : the larger partition grows to include 5 and 6 and loop terminates
- ➢ (i) : the pivot element is swapped

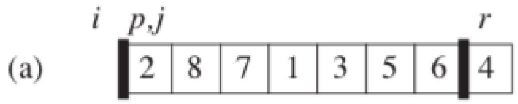
Quicksort

❖ Description of Quicksort

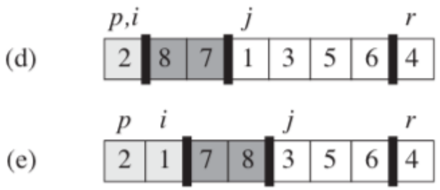
- Pseudocode for Partitioning

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

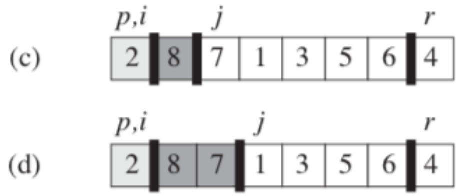
- 1 : initialize the pivot x and index i
- 3~6 : compare $A[j]$ with pivot x, increasing index j
  - ✓ If $A[j] \le pivot$: increase i and exchange $A[i]$ with $A[j]$
  - ✓ Otherwise : only increase index j
- 7~8 : exchange pivot with $A[i+1]$



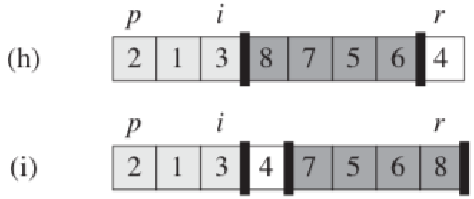
Line 1



Line 3~6
First case



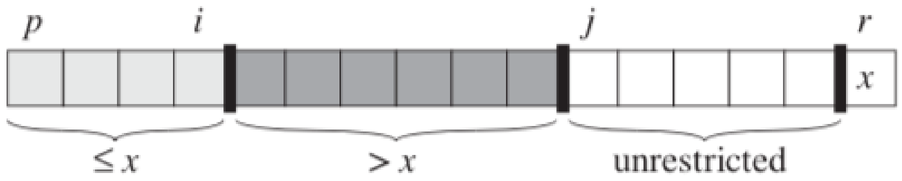
Line 3~6
Second case



Line 7~8

❖ Description of Quicksort

- Loop invariant of $\boldsymbol{PARTITION}$




```
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
```

➢ For any array index k, always satisfy these three property

1. If $p \le k \le i$, then $A[k] \le x$. → all elements of $A[p..i]$ are always smaller than x
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$. → all elements of $A[i+1..j-1]$ are always larger than x
3. If $k = r$, then $A[k] = x$. → $A[r]$ is a pivot element

➢ Initialization

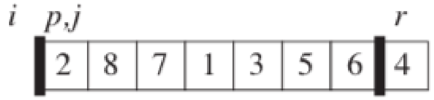
✓ Prior to the first iteration of the loop, $i = p - 1$ and $j = p$

✓ No value lie between p and i and between $i + 1$ and $j - 1$

✓ Loop invariant property 1,2 was satisfied

✓ Loop invariant property 3 is always true, because of line 1

(a)



BigDataLab
Dept. of Multimedia Engineering at Dongguk University

```
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
```

❖ Description of Quicksort

- Loop invariant of $\boldsymbol{PARTITION}$

1. If $p \le k \le i$, then $A[k] \le x$. ⟶ all elements of $A[p..i]$ are always smaller than x
2. If $i+1 \le k \le j-1$, then $A[k] > x$. ⟶ all elements of $A[i+1..j-1]$ are always larger than x
3. If $k = r$, then $A[k] = x$. ⟶ $A[r]$ is a pivot element

➢ Maintenance

✓ Consider two cases, depending on outcome of the test in line 4

✓ First case (a) : $A[j] > x$

i. Only action in the loop is to increment j

ii. After j incremented, condition 2 holds for $A[j-1]$ and all other entries remain unchanged

✓ Second case (b) : $A[j] \le x$

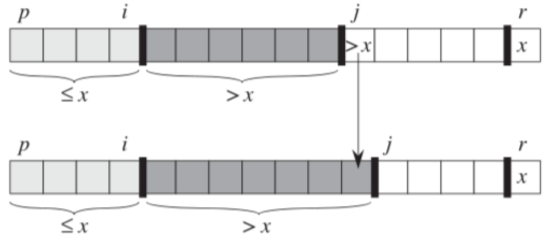
i. The loop increments index i

ii. Swap $A[i]$ and $A[j]$, and then increments j

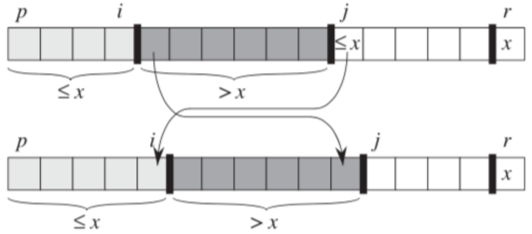
iii. $A[i] \le x$ and $A[j-1] \ge x$

iv. $A[i] \le x$ Condition 1,2 holds

(a)



(b)

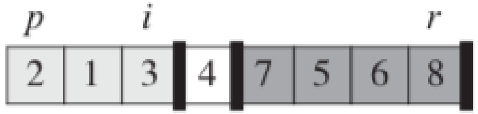


❖ Description of Quicksort

- Loop invariant of $\boldsymbol{PARTITION}$

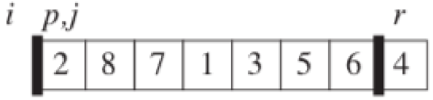
1. If $p \le k \le i$, then $A[k] \le x$. → all elements of $A[p..i]$ are always smaller than x
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$. → all elements of $A[i+1..j-1]$ are always larger than x
3. If $k = r$, then $A[k] = x$. → $A[r]$ is a pivot element

  - ➢ Termination
    - ✓ At termination $j = r$, therefore every entry in the array is in one of the three sets described by the invariant



- The running time of $\boldsymbol{PARTITION}$
  - ➢ $\theta(n)$
  - ➢ Index j iterate n−1 terms

```
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

(a) 

❖ Performance of Quicksort

- **Worst-case Partitioning**
  - ➢ Worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n-1$ elements and one with 0 elements
  - ➢ Assume that this unbalanced partitioning arises in each recursive call

4	6	8	9	7	10	5

pivot arranged

4	5	8	9	7	10	6

pivot arranged

  - ➢ The running time of $PARTITION$: $\theta(n)$
  - ➢ The running time of $PARTITION$ in array size 0 (just return) : $\theta(1)$
  - ➢ The running time of next iteration $PARTITION$: $T(n-1)$
  - ➢

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned} \longrightarrow \begin{aligned} \sum_{k=1}^{n} k &= \frac{1}{2}n(n+1) \\ &= \Theta(n^2) . \end{aligned}$$

Quicksort

❖ Performance of Quicksort

- Best-case Partitioning
  - ➢ Best-case behavior for quicksort occurs when the partitioning routine produces each size $\lfloor n/2 \rfloor$ and $\lceil n/2 - 1 \rceil$
  - ➢ Assume that this balanced partitioning arises in each recursive call

2	**3**	**1**	**4**	**7**	**10**	**5**

arranged　　　　pivot　　　　arranged

1	**2**	**3**	**4**	**5**	**7**	**10**

pivot　　　　pivot

  - ➢ The running time of $PARTITION$: $\theta(n)$
  - ➢ The running time of next iteration $PARTITION$: $2T(n/2)$
  - ➢ $T(n) = 2T(n/2) + \Theta(n)$
  - ➢ Using master theorem, $T(n) = \theta(n \lg n)$

❖ Performance of Quicksort

- average-case Partitioning
  - Expect that some of the splits will be reasonably well balanced and that some will be fairly unbalanced
  - $PARTITION$ produces a mix of “good” and “bad” splits

(a) (b)

  - (a) : mix of “bad” and “good” split = takes partitioning cost $\theta(n) + \theta(n-1) = \theta(n)$
  - (b) : only “good” split = takes partitioning cost $\theta(n)$
  - The cost of (a) and (b) are same!!

  - In average case the running time for partitioning is similar with best-case, still have $O(n \lg n)$ time

Thank You!