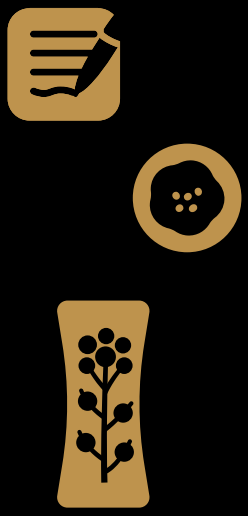


Theory of Data Structure

By Lee Jeong Yun

ADT

Concrete ⟷ Abstract

Instance | class | abstract class | interface | UML Model | ADT

Abstract Data Type

- all have in common.
- specify
  - description of target object
  - data that is stored.
  - operation on data
    - parameter
    - pre-condition
    - post-condition
    - return

Unified Model Language

graphical language used for designing & documenting OOP software.

UML diagram

class name
data ...
...
actions / methods

Interface

abstract class implement interface

class extends abstract class

Instance new class ;

Overriding	Same name of method in a parent class and its child class
overloading	Same name of method in one class with different parameter
polymorphism	associate many meanings to one method name.
inner class	nested class → belong together … readable & maintainable
Static variable	belong to the class as a whole, not just to one object. default = 0, Static method can access static variable, not instance variable.
Static method	{ can be used without calling object invoked by using class name belong to the class

ArrayList

```
ArrayList<Integer> integers = new ArrayList<>();
```

- .add ⇒ 뒤에 추가 or 인덱스 추가.
  .add(x) .add(i, x)
- .set ⇒ 특정 인덱스의 값 변경
  .set(i, x)
- .remove ⇒ 특정 인덱스의 값 삭제
  .remove(i)
- .size ⇒ arrayList 크기
  .size()
- .get ⇒ 특정 인덱스의 값 리턴
  .get(i)
- .contains ⇒ 값 존재 유무
  .contain(x)
- .indexOf ⇒ 값 위치
  .indexOf(x)

Generic

parameterized class

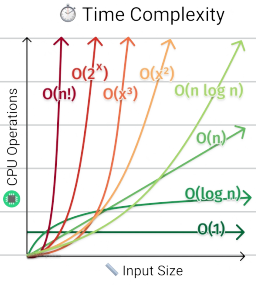
Performance

time complexity — time requirement

space complexity — memory space requirment

Factors determine running time
- problem size
- basic algorithm / actual processing
- memory access speed
- CPU/processor speed
- compile/linker optimization

Terminology
- Analysis of algorithms → the process of measuring the complexity of algorithms
- problem size → number of items that an algorithm processes
- Basic operation → the operation contributing the most to the total running time of an algorithm
- directly proportional → the time requirment increases by some factor
- growth-rate function (T(n)) → The number of basic operation for n.



Dominant term!

fast ↑	$\log_2 n$	logarithmic
	n	linear
	$n \log_2 n$	
	n^2	quadratic
	n^3	
slow ↓	2^n	exponential

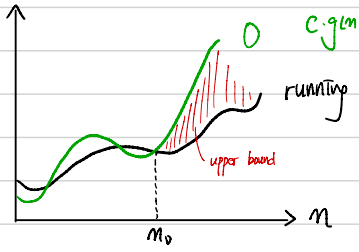
Asymptotic Notation

O ... big O notation → upper bound

Ω ... big Omega notation → lower bound

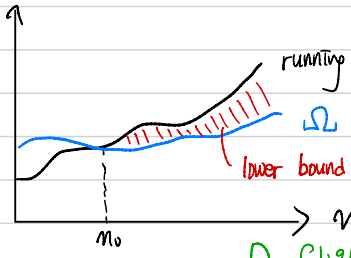
Θ ... big theta notation → upper & lower bound

O Notation



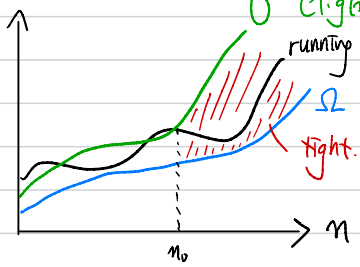
$0 \le f(n) \le C \cdot g(n), \quad n \ge n_0$

Ω Notation



$0 \le C \cdot g(n) \le f(n), \quad n \ge n_0$

Θ Notation



$0 \le C_2 \cdot g(n) \le f(n) \le C_1 \cdot g(n), \quad n \ge n_0$

$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

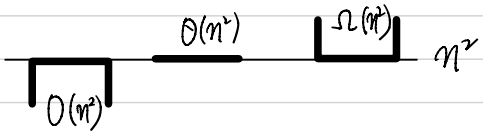
Notation cautions.

e.g., $T(n) \in O(n^2)$ ... (O)

$T(n) = O(n^2)$ ... (O)

$O(n^2) = T(n)$ ... (X)

Relation of Notation



List

ADT List

```
boolean   add(T)
boolean   add(i, T)
boolean   remove(T)
boolean   remove(i)
T         get(i)
integer   indexOf(T)
void      clear()
integer   size()
```

Linked List

data & next
getter & setter method

time complexity

	Array List	Linked List
add	O(n)	O(n)
remove	O(n)	O(n)
indexOf	O(n)	O(n)
clear	O(1)	O(1)
size & toArray	O(n)	O(n)

Sorted List

ascending order

Doubly Linked List

data & previous & next

Circular Linked List

Singly or doubly linked list. last node → first node
link

Stack

LIFO

push, pop, peek, isEmpty, clear, size

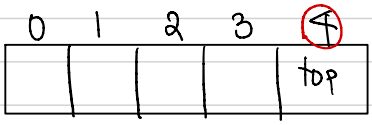
return null if stack is empty.

Linked Stack → size & top

Node

↳ data & next

Array Stack



Queue FIFO

ADT of a Queue

enqueue = put, insert
dequeue = get, remove
getFront = peek
isEmpty
clear

Linked Queue

-> size, first, last

Array Queue

linear queue vs circular queue
size vs unused element

Empty : (last - first + q.size) % q.size = q.size - 1
Full : (last - first + q.size) % q.size = q.size - 1

Empty : (last - first + q.size) % q.size = q.size - 1
Full : (last - first + q.size) % q.size = q.size - 2

Deque 덱

Double-ended Queue
add, remove from either head or tail

interface
- addToFront
- removeFront
- getFront
- addToBack
- removeBack
- getBack

Doubly Linked Deque

-> size, first, last

Node { data, next, prev }

Recursion

base case
inductive step
→ prevent stack overflow

factorial recursion

- $t(n) = \begin{cases} 1 & (n=1) \\ 1 + t(n-1) & (n>1) \end{cases}$ | $t(1)=1,\ t(2)=1+1,\ t(3)=1+2,\ \cdots,\ t(n)=1+n-1$

→ time complexity : $O(n)$

x^n recursion

$$x^n = \begin{cases} (x^{\frac{n}{2}})^2 & n=2k \\ x \cdot (x^{\frac{n-1}{2}})^2 & n=2k+1 \\ x^0 & n=0 \end{cases}, \qquad t(n) = \begin{cases} 1 + t(\frac{n}{2}) & n \geq 2 \\ 1 & n=1 \\ 1 & n=0 \end{cases}$$

$t(0)=1,\ t(1)=1,\ t(2)=2\quad t(4)=3\quad t(8)=4$

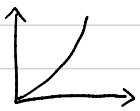
$t(n) = 1 + \log_2 n$

→ time complexity : $O(\log n)$

fibonacci

$$F_n = \begin{cases} 1 & n=0,\ n=1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

$$t(n) = \begin{cases} 1 & n=0,\ n=1 \\ 1 + t(n-1) + t(n-2) & n \geq 2 \end{cases} \Rightarrow \Omega(a^n),\ a>1$$



other approach { tail recursion, iteration, using stack, memorize } ⇒ constant time complexity

Hanoi

moving n disks. $m(n) \geq 2(n-1)+1 \geq 2m(n-1)+1 = m(n)$

binary search

$$t(n) = \begin{cases} 1 & n=0 \\ 1 + t(\frac{n}{2}) & n>0 \end{cases},$$

→ time complexity : $O(\log_2 n)$

triangular function

$\sin(x) = \sin(\frac{x}{2})\cos(\frac{x}{2})$

$\cos(x) = \sin(\frac{x}{2})^2 + \cos(\frac{x}{2})^2$

Tree = Node + Edge

node & link
empty tree → tree

S = set of trees
do not share nodes.
T(r, S) → tree
r = root.

Node : element of tree
Edge : link between two nodes, 간선.
Root : single node at top
Level : set of node with same hierarchy, 루트부터의 간선의 수
Parent & Children & Sibling
descendant, ancestor
subtree
Degree : # of children. degree of tree : the largest degree
Leaf : degree 0, no child
Interior node : not leaf
path, root path, root-to-leaf path. 노드 수 = length of path.
height : from leaf,
depth : from root,
Full tree : all same level, same degree.

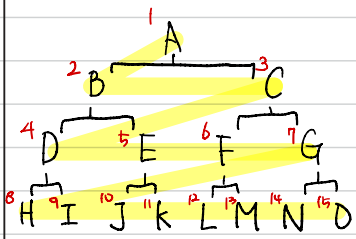
depth
height

Binary tree : degree 2
Ordered tree
Full binary tree : 0 or 2 children. total $(2^{h+1}-1)$ nodes. h=height
Complete binary tree : Full except leaf. leaf → left oriented.
Skewed Binary tree : every node has left (right) subtree

$$h+1 \leq \#\text{ of node} \leq 2^{h+1}-1$$

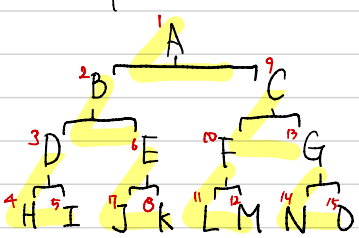
Breadth First Search.

Queue!
{add root, remove x, visit x, add its children}

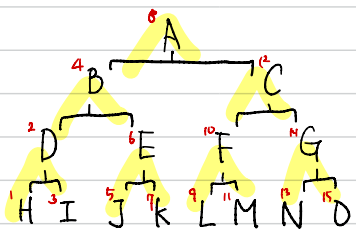


Depth First Search

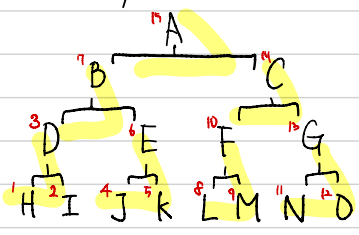
전위순회
pre order



중위순회
in order



후위순회
post order

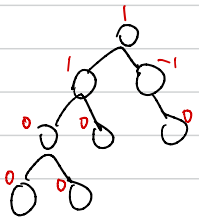


Binary Search Tree

up to 2 children, each node has key, keys are unique. left → less, right → more

in-order BST → sorted list

operation time complexity : $O(h)$, h for height



AVL tree

self-balancing by balance factor

left subtree height - right subtree height

empty → (-1)

rotation.

Heap

priority heap.
→ highest priority first out

Heap

complete binary tree.
Min-heap | parent ≤ children
Max-heap | parent ≥ children
find children [parent index × 2
" × 2 + 1
find parent - children index / 2
operation : insert, remove, find

construction : } Bottom-up $O(n)$
Top-down $O(n \log n)$

Sorting Algorithms.

pairwise vs distribute

internal sort vs external sort

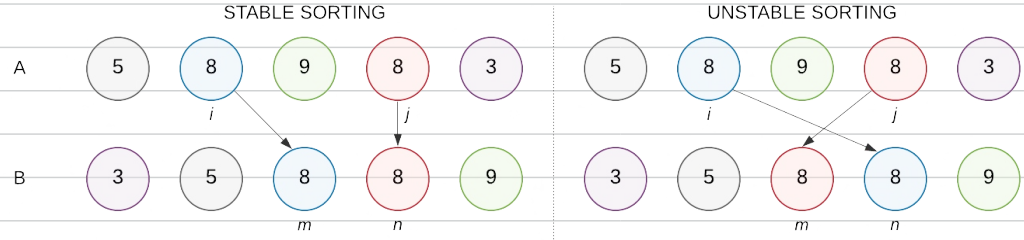
- bubble sort
- selection sort
- insertion sort
- shell sort
- heap sort
- merge sort
- quick sort

- bucket sort → 버킷 나눠서 버킷 내부 삽입정렬. O(n)
- counting sort → 각 key 개수세서 정렬. O(n)
- radix sort → LSD, MSD, with counting sort each digit

* 모든 정렬은 오름차순 기준 설명.

정렬의 안정성

key : value 쌍을 가진 객체 사이에서 같은 키를 갖는 객체 간 순서 유지될 때 → 안정성 Stability.

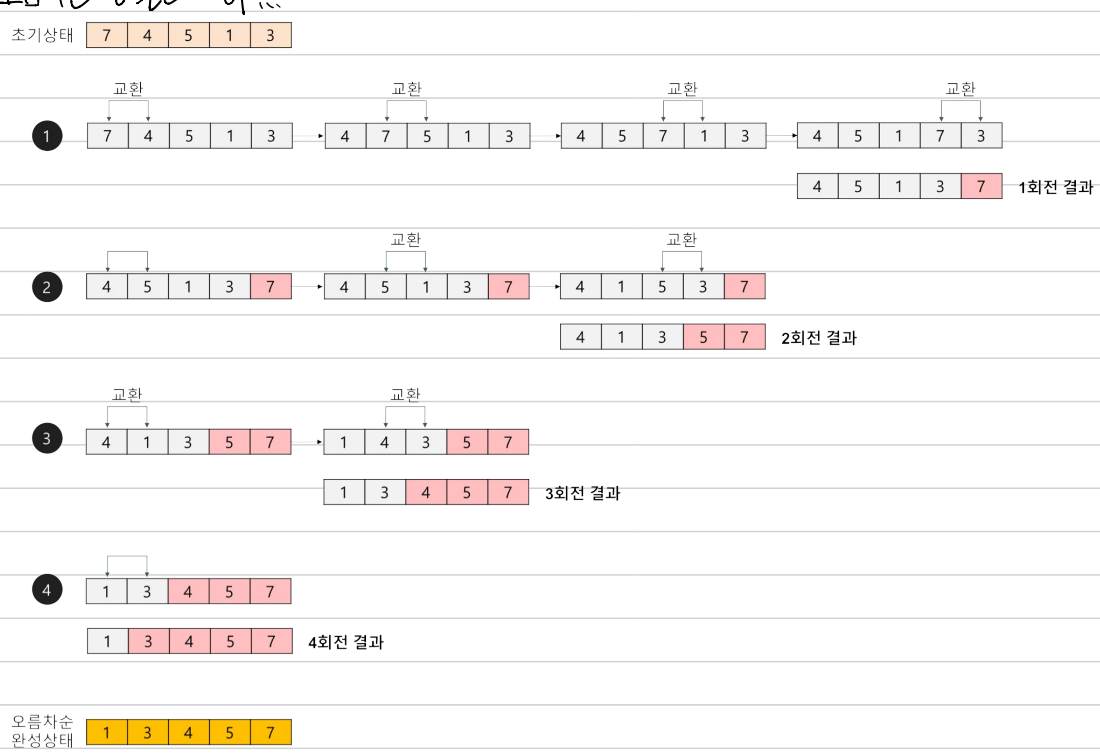


- 안정정렬
  bubble sort, insertion sort, merge sort
- 불안정 정렬.
  selection sort, quick sort, heap sort, shell sort

Bubble Sort

인접한 두 element를 비교, swap 하는 방식의 정렬 알고리즘

오름차순 정렬인 경우...



시간 복잡도 $T(n) = O(n^2)$

비교 횟수 : $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

Swap 횟수 [최대 : $3 \cdot \frac{n(n-1)}{2}$ ($\because$ swap → 3번의 이동필요.)
최소 : 0

```
swap (a,b)
    tmp = a
    a = b
    b = tmp
```

Name	Best	Avg	Worst	Run-time(정수 60,000개) 단위: sec
삽입정렬	n	n^2	n^2	7.438
선택정렬	n^2	n^2	n^2	10.842
버블정렬	n^2	n^2	n^2	22.894
셸 정렬	n	$n^{1.5}$	n^2	0.056
퀵 정렬	$n\log_2 n$	$n\log_2 n$	n^2	0.014
힙 정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.034
병합정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.026

Selection Sort

in-place sorting algorithm. → 입력받은 배열 외 추가적인 메모리 요구 X

각 순서의 위치는 정해져 있고, 이를 위해서 넣을지 selection 하는 algorithm

* 자료 이동 횟수가 미리 정해짐.

* 안정성 확보 X

시간 복잡도. $T(n) = \sum_{1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$

비교횟수 : 이중 반복문 ··· 외부 loop | $(n-1)$번.
내부 loop | $(n-1), (n-2), \cdots, 1$ 번

swap 횟수 : $(n-1)$번 교환. 상수시간작업.
→ $3(n-1)$회 이동.

Name	Best	Avg	Worst	Run-time(정수 60,000개) 단위: sec
삽입정렬	n	n^2	n^2	7.438
선택정렬	n^2	n^2	n^2	10.842
버블정렬	n^2	n^2	n^2	22.894
셸 정렬	n	$n^{1.5}$	n^2	0.056
퀵 정렬	$n\log_2 n$	$n\log_2 n$	n^2	0.014
힙 정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.034
병합정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.026

Insertion Sort

자료 배열의 모든 요소를 앞에서부터 차례대로 이미 정렬된 부분과 비교, 자신의 위치를 찾아 삽입해 정렬.
매 순서마다 해당 원소를 삽입할 수 있는 위치를 찾아 삽입.
=> index 1 (2nd element) 부터 시작!

* 안정 정렬.
* 이미 정렬된 경우 + 작은 데이터셋일 경우. ⇒ 유리.
* 레코드 이동 수. 레코드 크기 및 양 많은 경우 ⇒ 불리.

시간 복잡도.

- 최선. { 비교 (n-1)번 / 이동 X }

⇒ $T(n) = O(n)$

- 최악. (역순의 경우) { 비교 | 안쪽 loop ⇒ 1번 (n-1, n-2, ···, 2, 1) ··· $O(n^2)$ / swap | 안쪽 loop 각 단계별 (i+2)번 ··· $O(n^2)$ }

⇒ $T(n) = O(n^2)$

Name	Best	Avg	Worst	Run-time(정수 60,000개) 단위: sec
삽입정렬	n	n^2	n^2	7.438
선택정렬	n^2	n^2	n^2	10.842
버블정렬	n^2	n^2	n^2	22.894
셸 정렬	n	$n^{1.5}$	n^2	0.056
퀵 정렬	$n\log_2 n$	$n\log_2 n$	n^2	0.014
힙 정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.034
병합정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.026

Shell Sort

삽입정렬 보완

전체를 한번에 정렬하지 않는다.

derived from 빠른 정렬속도 of 삽입정렬 or 비슷하게 정렬되어 배열.

⇒ 전체를 비연속적인 부분으로 쪼개서 부분적 삽입 정렬 시행.

부분 리스트의 개수 줄여가며 삽입정렬. until 부분리스트가 1개, 즉 전체가 될 때까지!!

Skip number or gap = (정렬할 값의 수) / 2.

홀수가 좋다!

초기상태

10	8	6	20	4	3	22	1	0	15	16

정렬할 값의 수: 10
간격(gap) k의 초깃값: 10/2 = 5

① **간격 k=5** 일 때의 부분 리스트들

10					3					16
	8					22				
		6					1			
			20					0		
				4					15	

← 하나의 부분 리스트

간격 k=5 일 때의 부분 리스트를 각각 삽입 정렬로 정렬

3					10					16
	8					22				
		1					6			
			0					20		
				4					15	

1회전 결과

3	8	1	0	4	10	22	6	20	15	16

다음 k의 값: (5/2)+1 = 3

② **간격 k=3** 일 때의 부분 리스트들

3			0			22			15	
	8			4			6			16
		1			10			20		

간격 k=3 일 때의 부분 리스트를 각각 삽입 정렬로 정렬

0			3			15			22	
	4			6			8			16
		1			10			20		

2회전 결과

0	4	1	3	6	10	15	8	20	22	16

다음 k의 값: 3/2 = 1

③ **간격 k=1** 일 때의 부분 리스트들

0	4	1	3	6	10	15	8	20	22	16

간격 k=1 일 때의 부분 리스트를 각각 삽입 정렬로 정렬

0	1	3	4	6	8	10	15	16	20	22

3회전 결과

0	1	3	4	6	8	10	15	16	20	22

오름차순 완성상태

0	1	3	4	6	8	10	15	16	20	22

* 비연속적 부분리스트에서의 이동 => 더 긴 거리 이동 ... 최종 위치와 근접할 가능성 ↑
* 삽입정렬보다 빠르다.
* 간단한 구조.

시간 복잡도

Name	Best	Avg	Worst	Run-time(정수 60,000개) 단위: sec
삽입정렬	n	n^2	n^2	7.438
선택정렬	n^2	n^2	n^2	10.842
버블정렬	n^2	n^2	n^2	22.894
셸 정렬	n	$n^{1.5}$	n^2	0.056
퀵 정렬	$n\log_2 n$	$n\log_2 n$	n^2	0.014
힙 정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.034
병합정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.026

heap sort

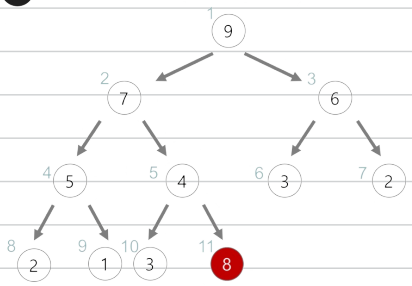
* 내림차순 기준.

(max heap) or (min heap) 구성 ⇒ 정렬!

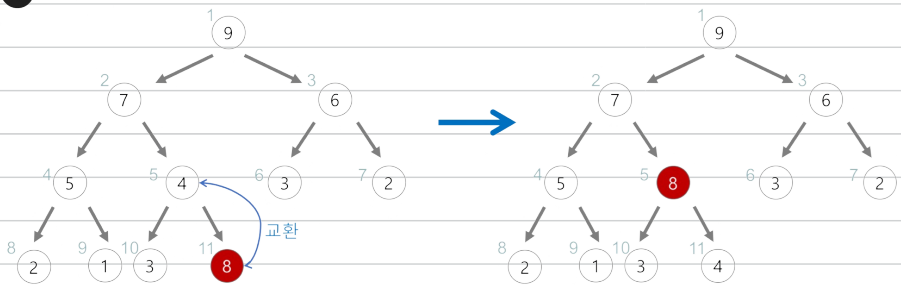
내림차순 / 오름차순.

heap에 추가

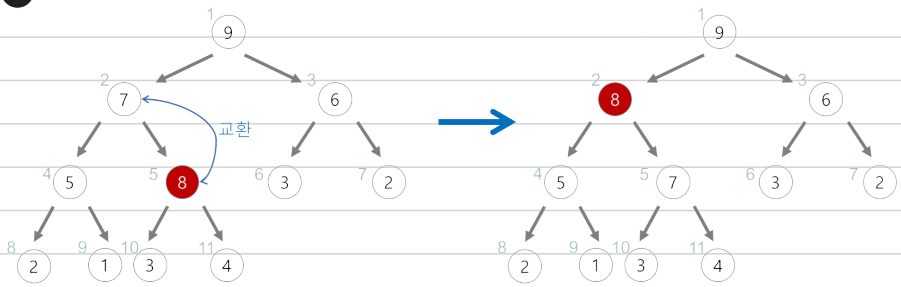
1. 인덱스순으로 가장 마지막 위치에 이어서 새로운 요소 8을 삽입



2. 부모 노드 4 < 삽입 노드 8 이므로 서로 교환



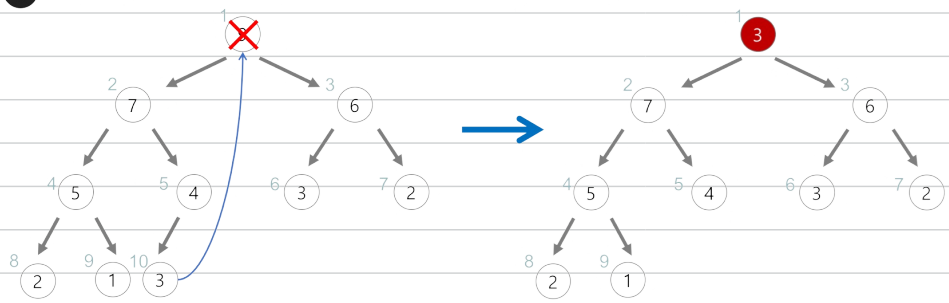
3. 부모 노드 7 < 삽입 노드 8 이므로 서로 교환



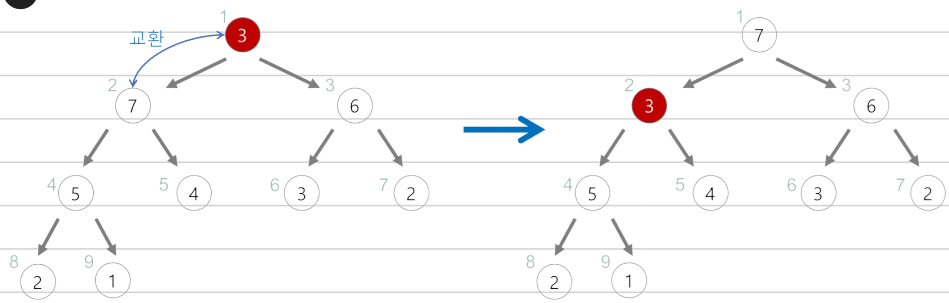
4. 부모 노드 9 > 삽입 노드 8 이므로 더 이상 교환하지 않는다.

heap에서 삭제

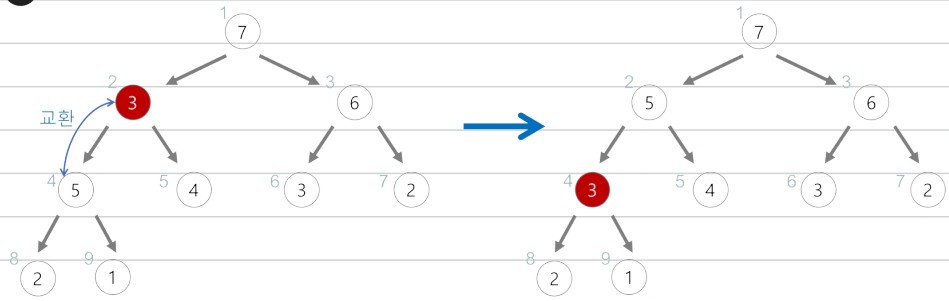
1 최댓값인 루트 노드 9를 삭제. (빈자리에는 최대 힙의 마지막 노드를 가져온다.)



2 삽입 노드와 자식 노드를 비교. 자식 노드 중 더 큰 값과 교환. (자식 노드 7 > 삽입 노드 3 이므로 서로 교환)



3 삽입 노드와 더 큰 값의 자식 노드를 비교. 자식 노드 5 > 삽입 노드 3 이므로 서로 교환



4 자식 노드 1, 2 < 삽입 노드 3 이므로 더 이상 교환하지 않는다.

시간복잡도 : $T(n) = O(n \log n)$

Name	Best	Avg	Worst	Run-time(정수 60,000개) 단위: sec
삽입정렬	n	n^2	n^2	7.438
선택정렬	n^2	n^2	n^2	10.842
버블정렬	n^2	n^2	n^2	22.894
셸 정렬	n	$n^{1.5}$	n^2	0.056
퀵 정렬	$n\log_2 n$	$n\log_2 n$	n^2	0.014
힙 정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.034
병합정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.026

Merge Sort

Divide & Conquer Algorithm:

→ 문제를 작은 두 문제로 나눠서 각각을 해결. 그 결과를 종합해 원래 문제 해결

divide → conquer → combine

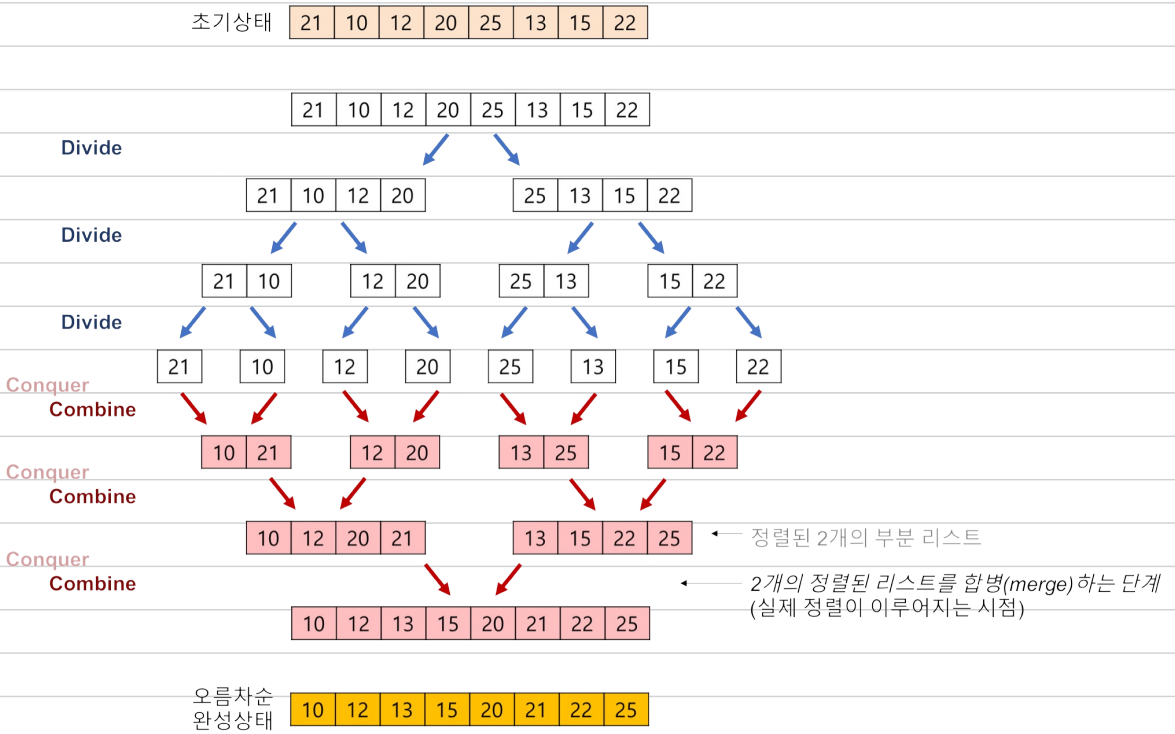
recursively call d & c until size of partial array is enough small.

divide : 입력 배열을 같은 크기의 부분배열 2개로 분할.

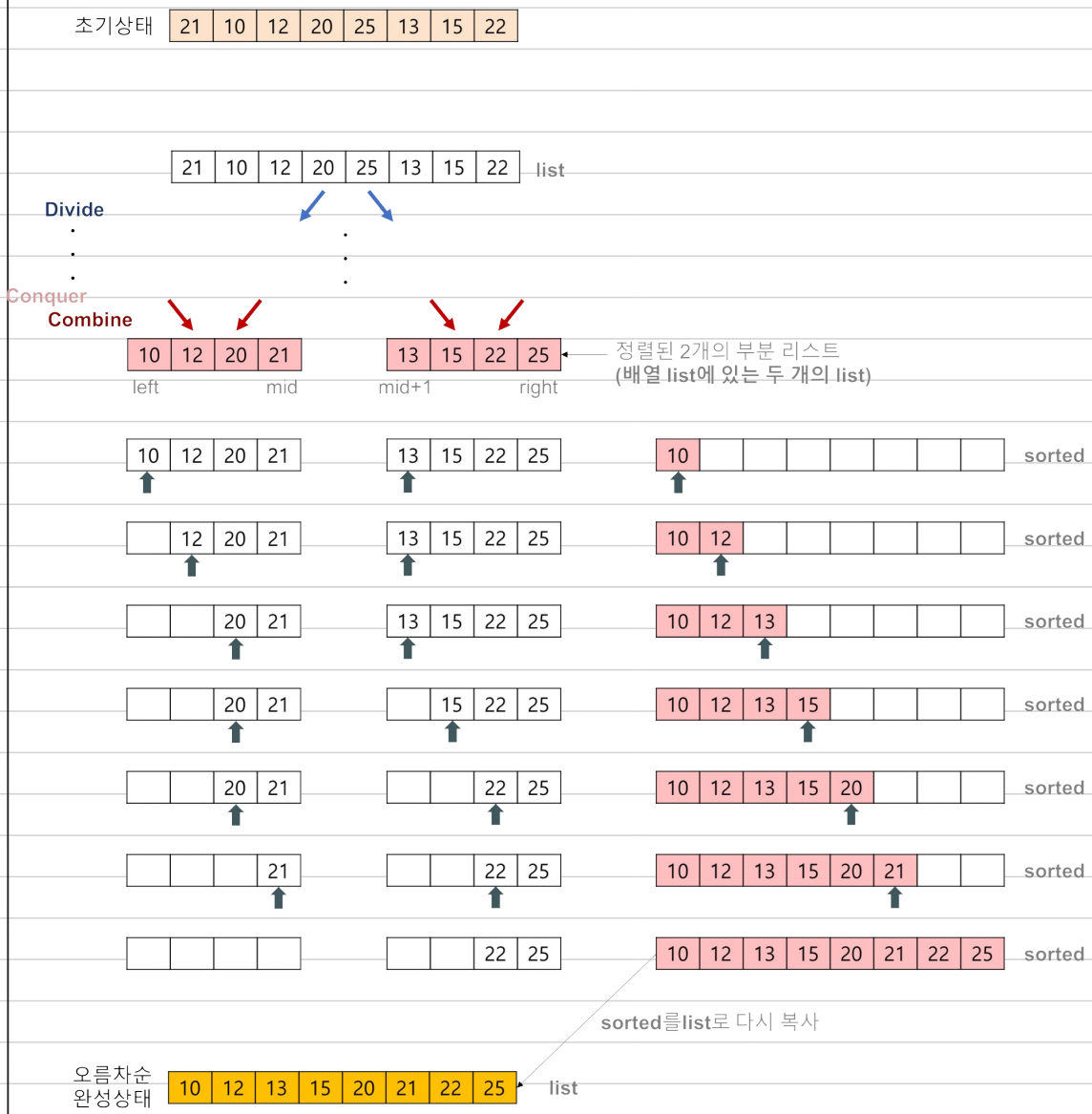
conquer : 부분 배열 정렬. 부분배열이 충분히 작지 않다면 순환호출(재귀)

→ 다시 분할 정복 적용.

combine : 정렬된 부분배열을 하나의 배열에 합병



- 2개의 정렬된 리스트 합병
  - 두 리스트 값을 하나씩 비교해 더 작은값을 새로운 리스트에 옮김
  - 둘중 하나 끝날때까지! 남은건 새 리스트 뒤에!
  - 새 리스트를 원래 리스트에 옮기기.



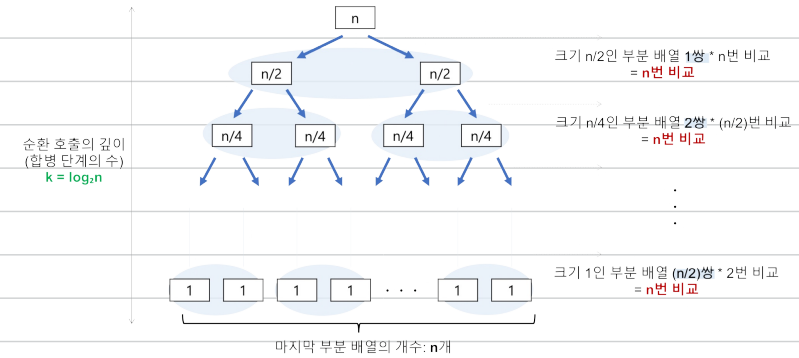
* 배열이라면 → 임시 배열 필요.
  이동 횟수↑ 시간↑

* 안정정렬. 데이터 분포에 영향 안 받는다.
  연결리스트 → 데이터 이동 ↓, 제자리 정렬 (in-place sorting) 가능
  큰 레코드 vs 연결리스트 → 가장 빠름!

시간복잡도.

divide ⋯ 비교. 이동 X

combine ⋯ 비교.



재귀 깊이 : $n = 2^k$ 개의 레코드, 깊이 $= k = \log_2 n$

하나의 합병단계 : 최대 n번의 비교

⇒ 총 $(n\log_2 n)$번

⋯ 이동

재귀 깊이 : $n = 2^k$개의 레코드, 깊이 $= k = \log_2 n$

각 합병의 이동 : $2n$번. 원래 배열 ↔ 임시 배열 왕복

⇒ 총 $2n\log_2 n$

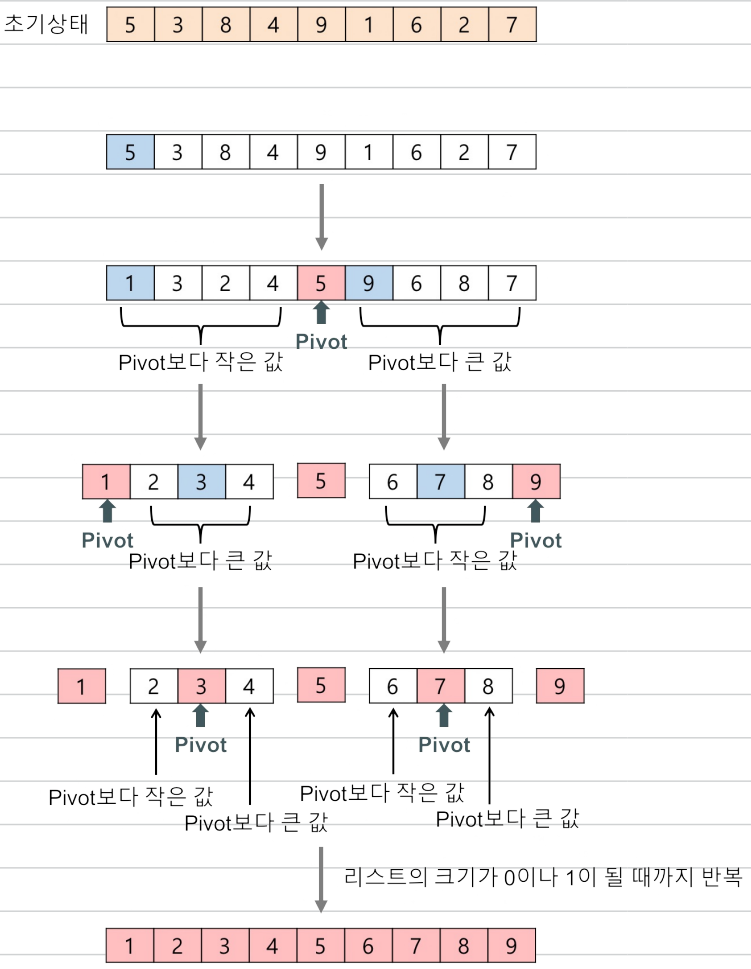
$T(n) = O(n\log n)$

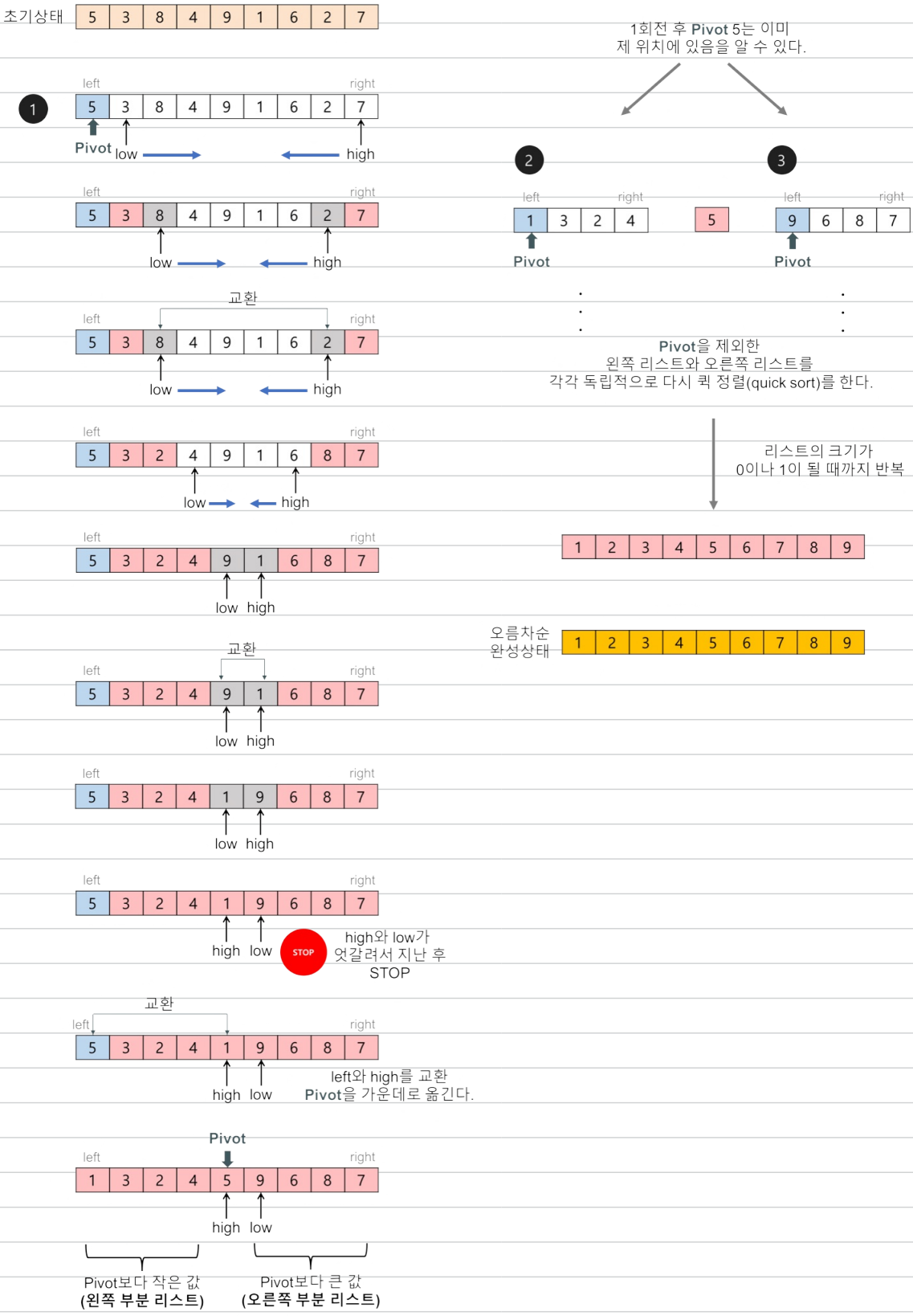
Name	Best	Avg	Worst	Run-time(정수 60,000개) 단위: sec
삽입정렬	n	n^2	n^2	7.438
선택정렬	n^2	n^2	n^2	10.842
버블정렬	n^2	n^2	n^2	22.894
셸 정렬	n	$n^{1.5}$	n^2	0.056
퀵 정렬	$n\log_2 n$	$n\log_2 n$	n^2	0.014
힙 정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.034
병합정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.026

Quick Sort

Divide & Conquer Algorithm
분할정복!

- 리스트 안 한 요소 ⇒ pivot으로 지정.
  - ? < pivot ⇒ pivot 왼쪽
  - ? > pivot ⇒ pivot 오른쪽.
- pivot 왼쪽 리스트 & pivot 오른쪽 리스트 재정렬.
  - by 순환호출(재귀) until 더 이상 분할 불가
    i.e., 리스트 크기 = 0 or 1





* 빠른 속도.
추가 메모리 필요 X, just O(log n) 만큼의 메모리!

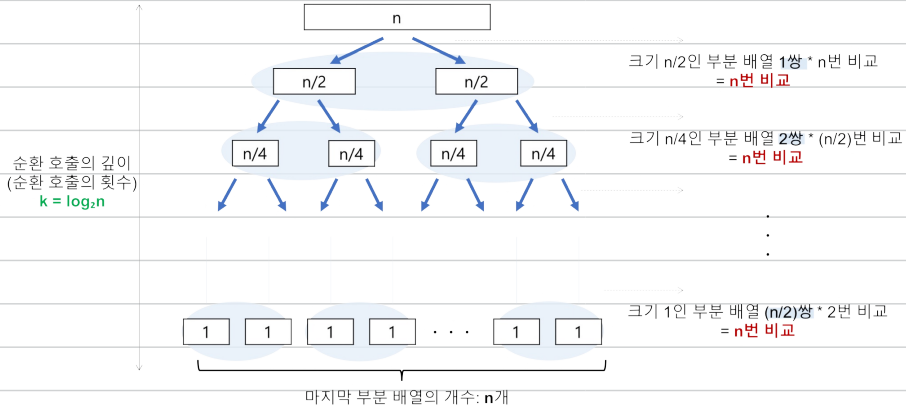
* 정렬된 배열의 경우 느림. (∵ 불균형 분할)

* 불균형 분할 방지 위해 균등 분할 가능한 데이터를 pivot으로 선정.

시간복잡도

퀵정렬

• 비교



순환호출 깊이 = $\log_2 n = k$, $n = 2^k$ = 데이터 개수

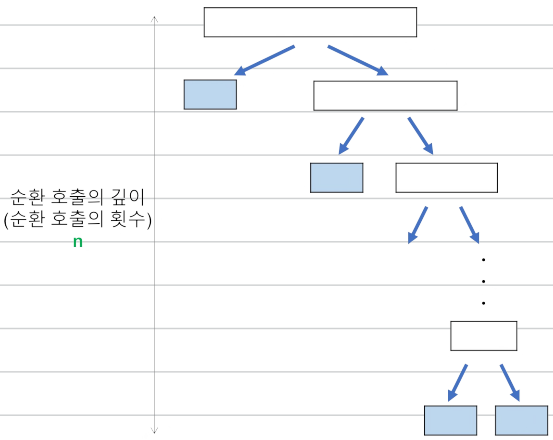
각 단계 비교 = n 회

=> $n\log_2 n$ 회 비교

• 이동 횟수 ... 적어서 무시 가능

$T(n) = O(n\log n)$

최악 ... 계속 불균형 분할되는 경우



• 비교

순환호출 깊이 = n

각 단계 비교 = n

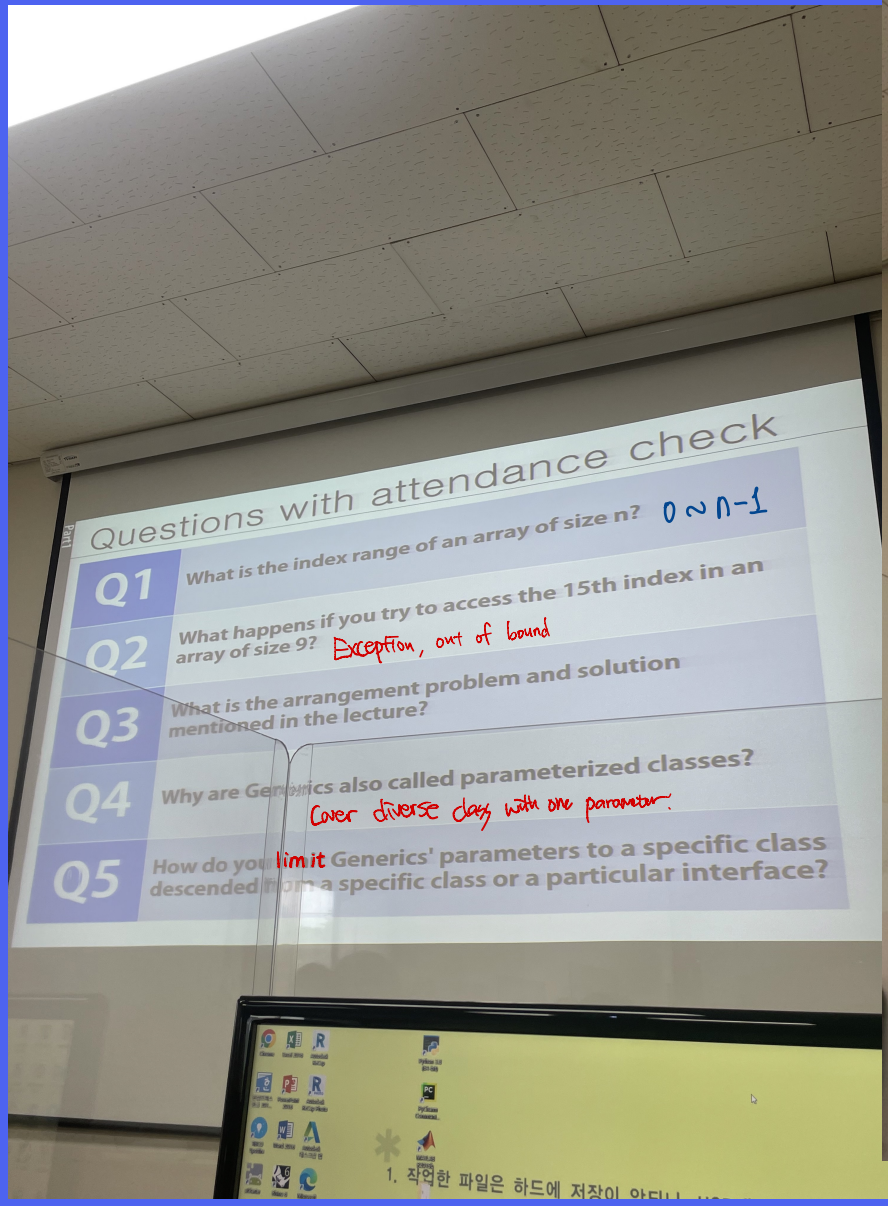
=> n^2

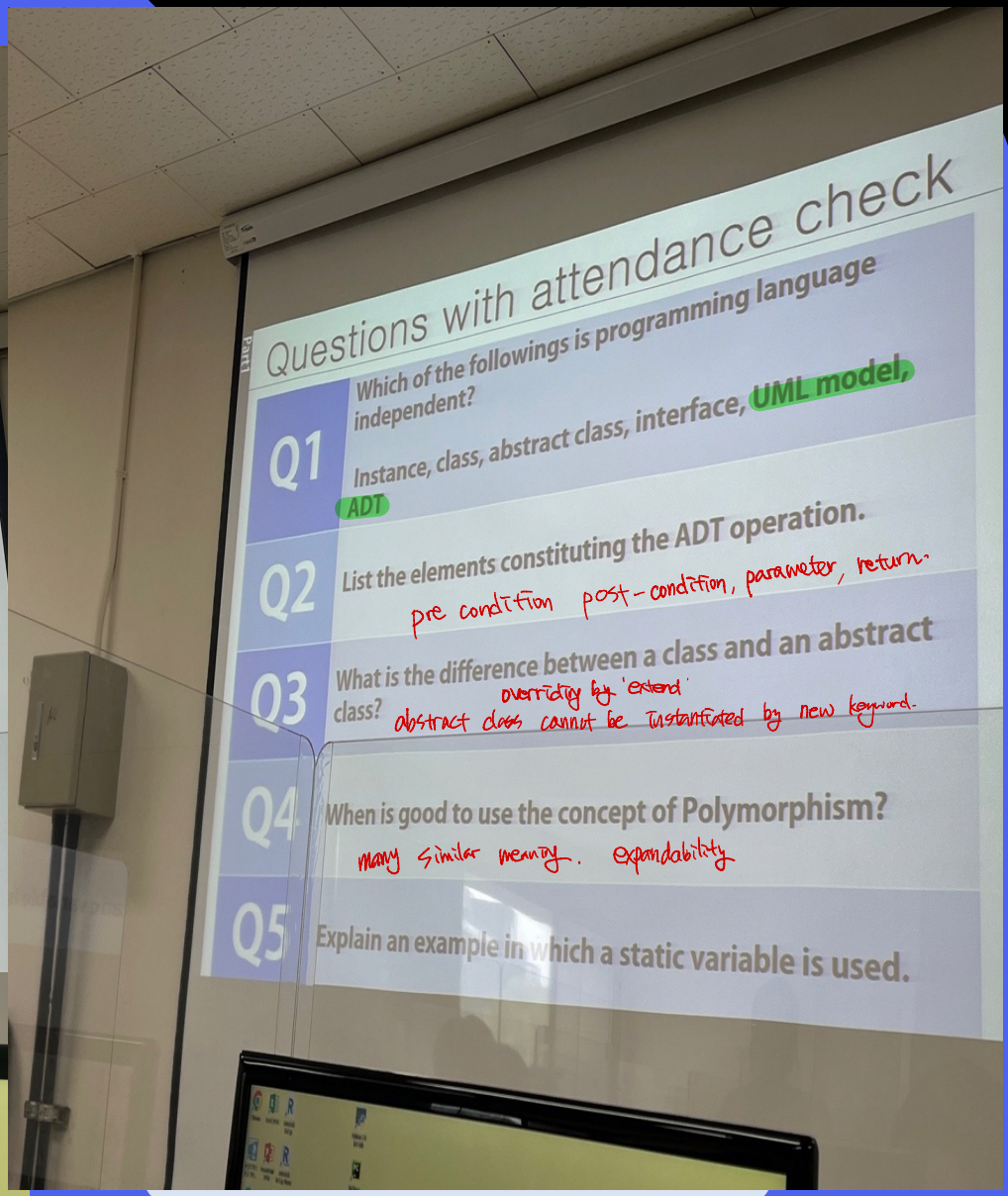
• 이동 횟수 ... 무시 가능!

$T(n) = O(n^2)$

Name	Best	Avg	Worst	Run-time(정수 60,000개) 단위: sec
삽입정렬	n	n^2	n^2	7.438
선택정렬	n^2	n^2	n^2	10.842
버블정렬	n^2	n^2	n^2	22.894
셸 정렬	n	$n^{1.5}$	n^2	0.056
퀵정렬	$n\log_2 n$	$n\log_2 n$	n^2	0.014
힙 정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.034
병합정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.026

Contents


Questions with attendance check
Q1 What is the index range of an array of size n? 0 ~ n-1
Q2 What happens if you try to access the 15th index in an array of size 9? Exception, out of bound
Q3 What is the arrangement problem and solution mentioned in the lecture?
Q4 Why are Generics also called parameterized classes? Cover diverse class with one parameter.
Q5 How do you limit Generics' parameters to a specific class descended from a specific class or a particular interface?


Questions with attendance check
Q1 Which of the followings is programming language independent?
Instance, class, abstract class, interface, UML model, ADT
Q2 List the elements constituting the ADT operation.
pre condition post-condition, parameter, return.
Q3 What is the difference between a class and an abstract class? overriding by 'extend'
abstract class cannot be instantiated by new keyword.
Q4 When is good to use the concept of Polymorphism?
many similar meaning. expandability
Q5 Explain an example in which a static variable is used.

Hash Table

Dictionary : key & value

performance

	BST	Balanced BST	Sorted array list	Sorted linked list
look up	O(log n)	O(log n)	O(log n)	O(n)
add	O(n)		O(n)	
delete				

Sequencial access → linear time | linked list
direct(random) access → constant time | array

Direct access + no advance info → hash table.
hash function : key % size, ...
operation put(K,V) : V remove(k) : V containskey(K) : Boolean
get(k) : V

Iterator for List Set Map Queue — method: hasNext, next, remove

λ expression — parameter → expression

load factor = λ — # of key / size of hashtable. desired : 0.75 ~ 0.8.
load factor > 1 → collision! size up : 2m+1

collision resolving

- closed addressing → separate chaning, array of buckets.
  i → A → B

- open addressing

→ linear probing ⇒ 옆칸 인덱스에 삽입. primary clustering issue
$h_i(k) = (h(k) + i) \% size$

→ quadratic probing ⇒ 제곱 간격으로.
$h_i(k) = (h(k) + i^2) \% size$
secondary clustering

→ double hashing ⇒ 그냥 hash 또 안됐다 double hashing 값 사용.
$h(i,k) = (h_1(k) + i \cdot h_2(k)) \% size$

no collision → look up table → waste space
perfect hash function ⇒ open addressing
constant time

100% load factor + look up table ⇒ Minimum perfect hash function

[3*{5+(15-7)/2-9}+1]^2

3 5 15 7 - 2 / + 9 - * 1 + 2 ^