Priority Queue

PQ의 내부 구조 (구현 방법) 이해 + 이에 따라 어떤 경우에 어떻게 활용하는지 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. Binary Heap 사용한 PQ의 구현
03. Priority Queue 사용 어플리케이션의 공통적인 특성
04. Slider Puzzle and A* Search with PQ
05. 실습: PQ를 사용한 Slider Puzzle Solver 구현

공지사항: lms에 과제물을 올릴 때 기존에 올린 파일과 같은 이름의 파일 올리면 -1, -2 등이 붙는데, 이때문에 감점되지는 않습니다. 또한 수업 자료도 변경되면 마찬가지로 숫자가 붙어 있으니 변경 여부 확인해 보세요 (첫 버전에서 약간씩 수정이 있습니다. 특히 실습 과제 요건 변경 있는지 꼭 확인).

Priority Queue(우선순위 큐): 가장 우선순위 높은 원소 먼저 삭제

데이터 구조	어떤 원소를 먼저 삭제하는가?
Stack	가장 늦게 추가한 원소
Queue	가장 먼저 추가한 원소
Randomized Queue	임의의 원소
Priority Queue	**가장 우선순위 높은 원소**

ID	작업	인자	반환 값
1	insert	P	
2	insert	Q	
3	insert	E	
4	del min		E
5	insert	X	
6	insert	A	
7	insert	M	
8	del min		A
9	insert	P	
10	insert	L	
11	insert	E	
12	del min		E

minPQ의 예

min, max 및 그 외 total order 지키는 어떤 key로도 우선순위 지정 가능

대소 관계 정의 시 지켜야 할 규칙 (**Total Order**)

- Anti-symmetry: if a≤b and b≤a, then a == b
- Transitivity: if a≤b and b≤c, then a≤c
- Totality: either a≤b or b≤a or a == b

"정렬"된 순으로 원소 사용할 때 정렬 대신 사용 가능

작업	1회 비용		횟수		총 비용
insert() 모든 원소 PQ에 넣기	~log N	x	N	=	~N log N
delMin() (정렬된 순서로) 한 원소씩 PQ에서 꺼내 쓰기	~log N	x	N	=	~N log N

- **원소 사용 전 입력 데이터 전체가 다 주어진 경우**
- **~N log N 비용의 정렬** 사용하는 것과 비용 유사
- Heap Sort 사용하는 것과도 유사

삭제하면서 추가도 같이 일어나는 경우 → PriorityQ가 더 유리

delMin() 하며 동시에 insert() 계속 되는 경우 PQ가 정렬보다 유리

(∵ 삭제, 추가 단계에 상한함이 logN의 성능을 보장하기 때문)

PQ 사용시 비용

작업	1회 비용	횟수	총 비용
insert()	~log N	N	**~N log N**
delMin()	~log N	N	**~N log N**

- 오늘 실습 과제도 이처럼 PQ가 정렬보다 유리한 경우에 해당

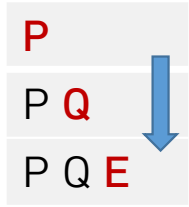
정렬 사용시 비용

작업	1회 비용	횟수	총 비용
insert()	**~N**	N	**~N^2**
delMin()	~1	N	**~N**

- 정렬된 배열에 insert()하려면 이진 탐색으로 추가할 위치 찾고 (~logN) 추가한 원소 이후 원소를 한 칸씩 옮겨 담아야 함 (~N)
- delMin()은 첫 원소 제거하고 시작점 위치를 다음 위치로 기억 (~1)

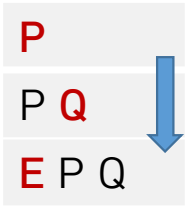
Priority Queue에 대한 기본 구현 방법

정렬 상태 유지 안 함
맨 뒤에 (혹은 맨 앞에) 새 원소 추가하므로 insert() 비용 작음
Min은 탐색해야 하므로 delMin() 비용 큼



구현 방식	insert() 비용	delMin() 비용
Unordered list	1	N
Ordered array	N	1
목표 (heap)	logN	logN

정렬 상태 유지함
정렬된 위치에 새 원소 추가하므로 insert() 비용 큼
Min은 항상 맨 앞에 있으므로 delMin() 비용 작음



Priority Queue

PQ가 무엇이고, 어떻게 구현하며, 어떤 경우에 어떻게 활용하는지 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Binary Heap 사용한 PQ의 구현
03. Priority Queue 사용 어플리케이션의 공통적인 특성
04. Slider Puzzle and A* Search with PQ
05. 실습: PQ를 사용한 Slider Puzzle Solver 구현

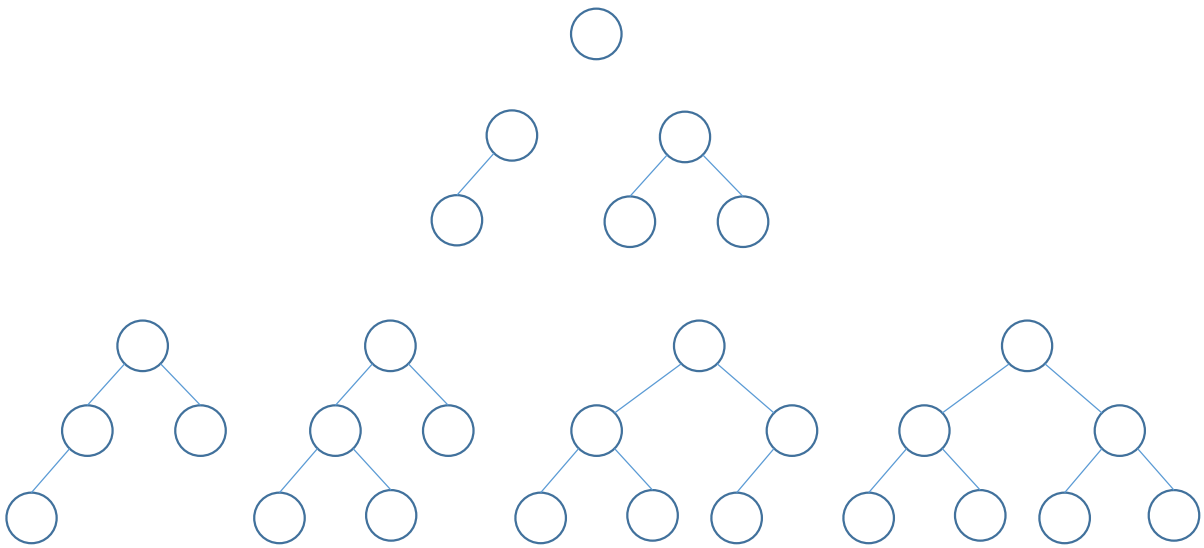
많은 프로그래밍 언어에서 Binary Heap 사용해 Priority Queue 구현

프로그래밍 언어	클래스
Python	`queue.PriorityQueue, heapq`
Java	`java.util.PriorityQueue`
C++	`std::priority_queue`
Ruby	`Containers::PriorityQueue`
…	…

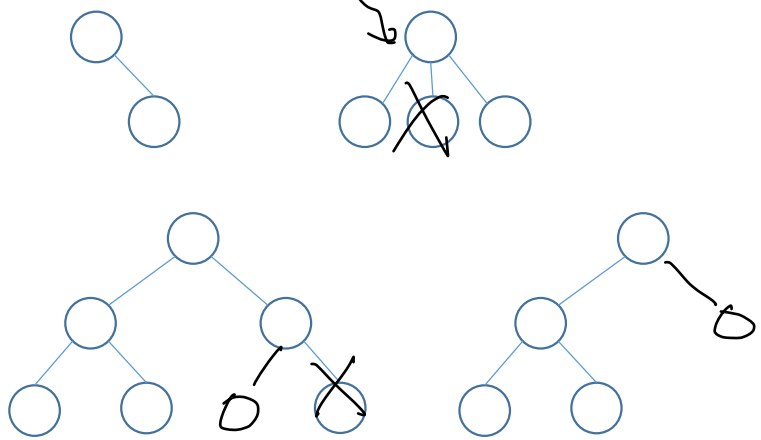
Binary Heap: Complete(Binary)Tree

- Binary tree: 모든 노드의 자식이 2개 (왼쪽, 오른쪽)
- Complete tree:
  - (가장 아래 level 제외한) 모든 level이 빈 곳 없이 가득 차 있으며
  - 가장 아래 레벨은 왼쪽부터 빈 곳 없이 차례로 채워진 트리

Complete Binary Tree 예



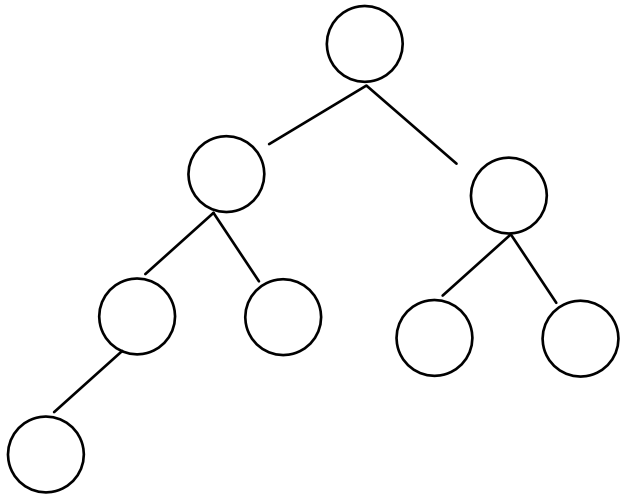
Complete Binary Tree 아닌 예



Binary Heap: Complete Binary Tree

- **Binary** tree: 모든 노드의 **자식이 2개** (왼쪽, 오른쪽)
- **Complete** tree:
  - (가장 아래 level 제외한) 모든 level이 빈 곳 없이 가득 차 있으며
  - 가장 아래 레벨은 **왼쪽부터 빈 곳 없이 차례로 채워진** 트리

[Q] 노드가 8개인 complete binary tree를 그리시오.

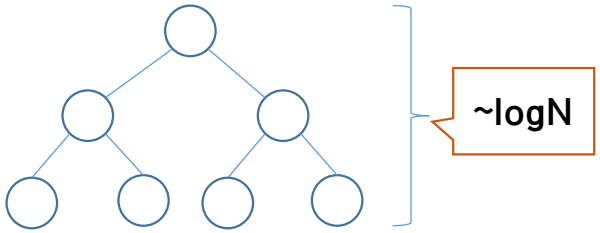


[Q] 노드 수가 k개인 complete binary tree는 단 한 가지 모양인가? 아니면 여러 다른 모양이 나올 수 있는가? 1개

https://alchetron.com/cdn/hyphaene-compressa-84ceade1-7c1e-45e9-82b0-5d2aa4e2163-resize-750.jpeg

왜 tree 사용하나? 접근 속도 ~logN 이도록 하기 위한 시도

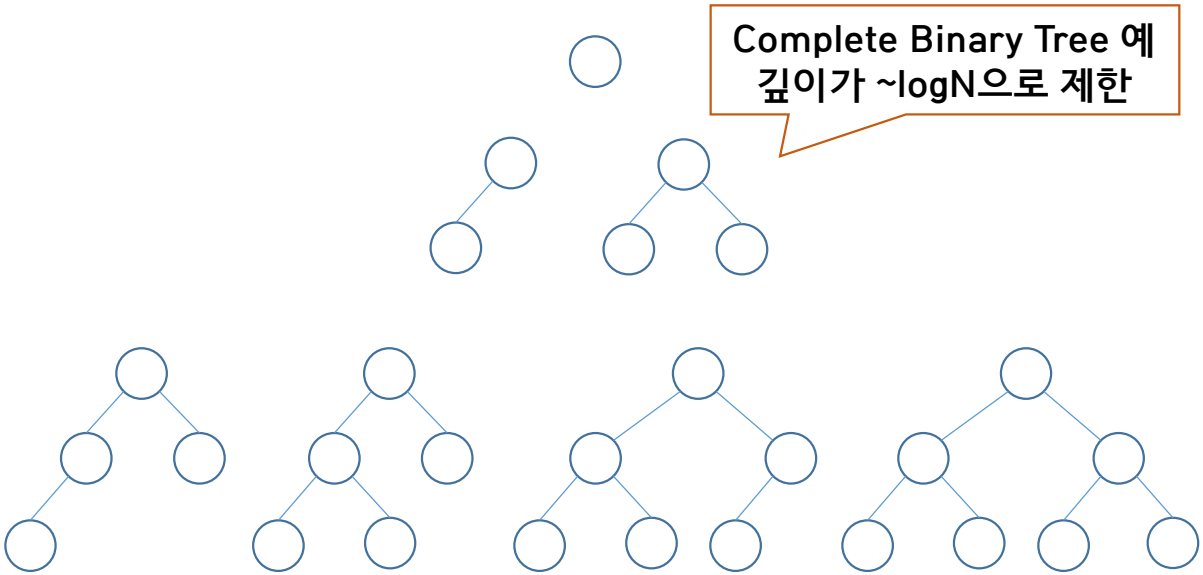
- ~logN 시간에 수행되는 많은 방법이 트리에 기반함 (예: Quick Union for Union Find)
- 트리 깊이(높이) ~logN 이므로
- insert(), delete() 시 트리 깊이에 비례한 만큼의 작업 수행하도록 하면
- ~logN 시간에 수행하도록 할 수 있음

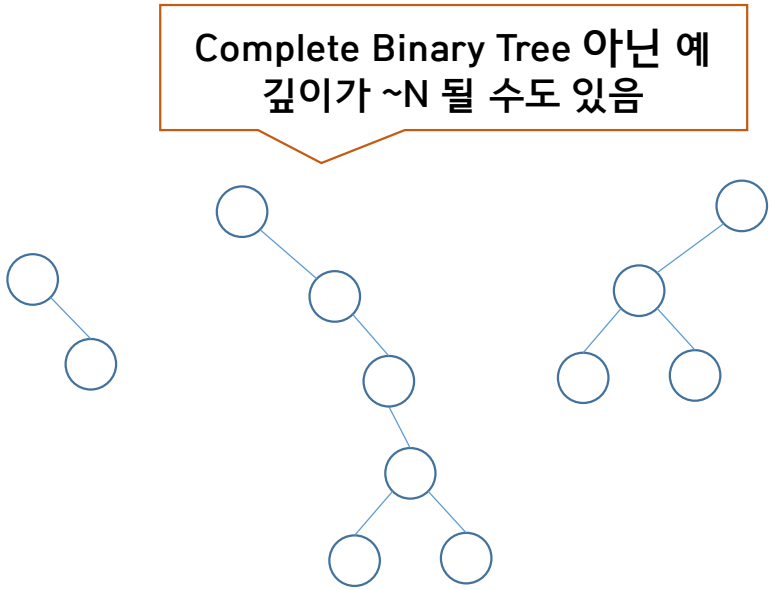


왜 complete binary tree 사용하나? (1) 깊이 제한되는 균형 잡힌 트리

logN

- 왼쪽부터 차곡차곡 빠짐없이 담는 구조이므로
- 트리가 한쪽으로만 길어지거나 하지 않아
- 깊이를 logN에 비례하게 유지 가능 (N은 정점 개수)
- 따라서 깊이에 비례한 횟수의 작업 하도록 할 때 (insert, delete 등) 비용을 logN으로 제한 가능





[Q] 정점 N개로 만든 이진 트리가
complete 이진 트리보다 더 얕아질 수 있는가?

NO..

(구현이 편리)

왜 complete binary tree 사용하나? (2) Linked list 아닌 배열에 담을 수 있어 편리

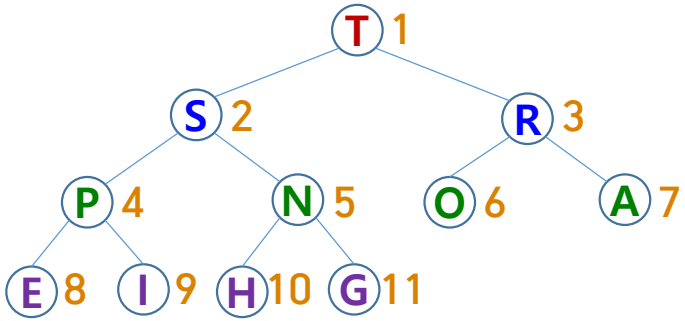
- **왼쪽부터 차곡차곡 빠짐없이** 담는 구조의 트리이므로
- (트리에 담는 순서 그대로) 배열에 왼쪽부터 차곡차곡 빠짐없이 담을 수 있음
- 배열에 담으면 **부모-자식 간 링크 만들지 않아도 index 만으로** 찾아갈 수 있어 메모리 절약 & 편리

실제 메모리에 저장하는 방식

k번째 노드는 정확히 a[k]에 저장됨

i	0	1	2	3	4	5	6	7	8	9	10	11
a[i]	-	T	S	R	P	N	O	A	E	I	H	G

우리가 생각하는 형태



- Node i의
  - 부모: i // 2
  - 왼쪽 자식: i × 2
  - 오른쪽 자식: i × 2 + 1

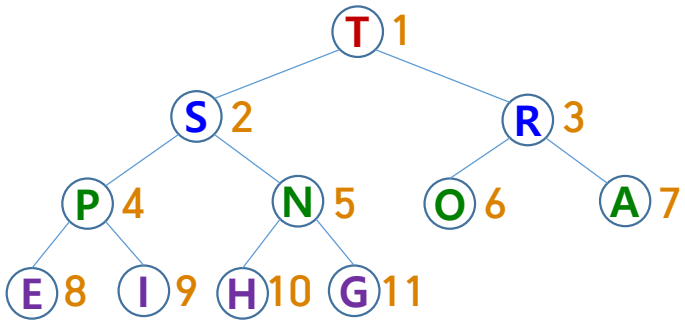
[Q] 이 식들이 맞는지 검증해 보시오.

Complete binary tree 사용 이유
① 값이 제한 가능
② 배열에 저장가능

왜 index 0 비워 두고 index 1 부터 담나? 부모-자식 접근 식 더 간단

Index 1부터 담기

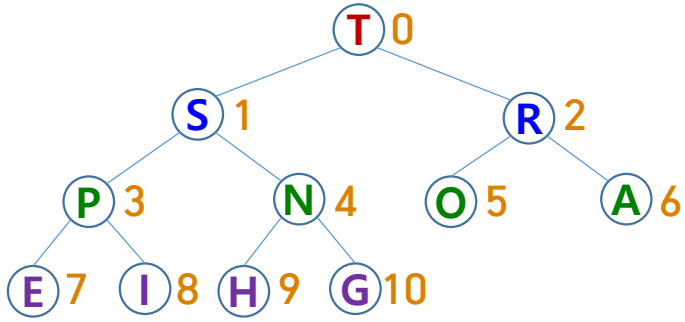
i	0	1	2	3	4	5	6	7	8	9	10	11
a[i]	-	T	S	R	P	N	O	A	E	I	H	G



- Node i의
  - 부모: i//2
  - 왼쪽 자식: i×2
  - 오른쪽 자식: i×2+1

Index 0부터 담기 (식이 더 복잡)

i	0	1	2	3	4	5	6	7	8	9	10	11
a[i]	T	S	R	P	N	O	A	E	I	H	G	-

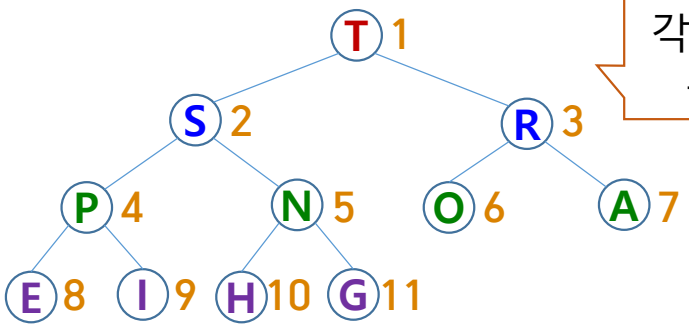


- Node i의
  - 부모: (i-1)//2
  - 왼쪽 자식: i×2+1
  - 오른쪽 자식: i×2+2

Binary heap(**complete binary tree**) 구조에서
Priority queue에 필요한 **insert(), delMax() 효율적 수행 위해** (maxPQ 가정)
Heap-order 조건 항상 만족하며 원소가 저장되도록 함

- (Max) **heap-order 조건: 부모 key ≥ 자식 key**

Key: 여러 값으로 구성된 tuple 저장되어 있을 때,
tuple 간 대소 비교에 사용하는 값



각 노드에는 key외에 다른 값도 함께 저장되어 있을 수 있으나, 이 그림에서는 **key(알파벳)**만 보여줌

[Q] 왼쪽 예제 원소들은 모두 heap-order에 맞게 저장되어 있는지 검증 하시오.

i	0	1	2	3	4	5	6	7	8	9	10	11
a[i]	-	T	S	R	P	N	O	A	E	I	H	G

[Q] Heap-order 만족하도록 저장하면 insert(), delMax()를 모두 ~logN 시간에 효율적으로 수행 가능하다. 예를 들어 delMax() 하려면 heap의 어디에 있는 값을 꺼내면 되나?

[Q] 다음 배열 중 max heap-order에 따라 저장된 binary heap이 아닌 것은?

① 루트

i	0	1	2	3	4	5	6	7	8	9	10
a[i]	-	19	18	17	16	15	14	13	12	11	10

부모가 자기자신보다 더크므로 OK

②

i	0	1	2	3	4	5	6	7	8	9	10
a[i]	-	10	10	10	10	10	10	10	10	10	10

③

i	0	1	2	3	4	5	6	7	8	9	10
a[i]	-	34	30	29	27	25	17	16	19	22	24

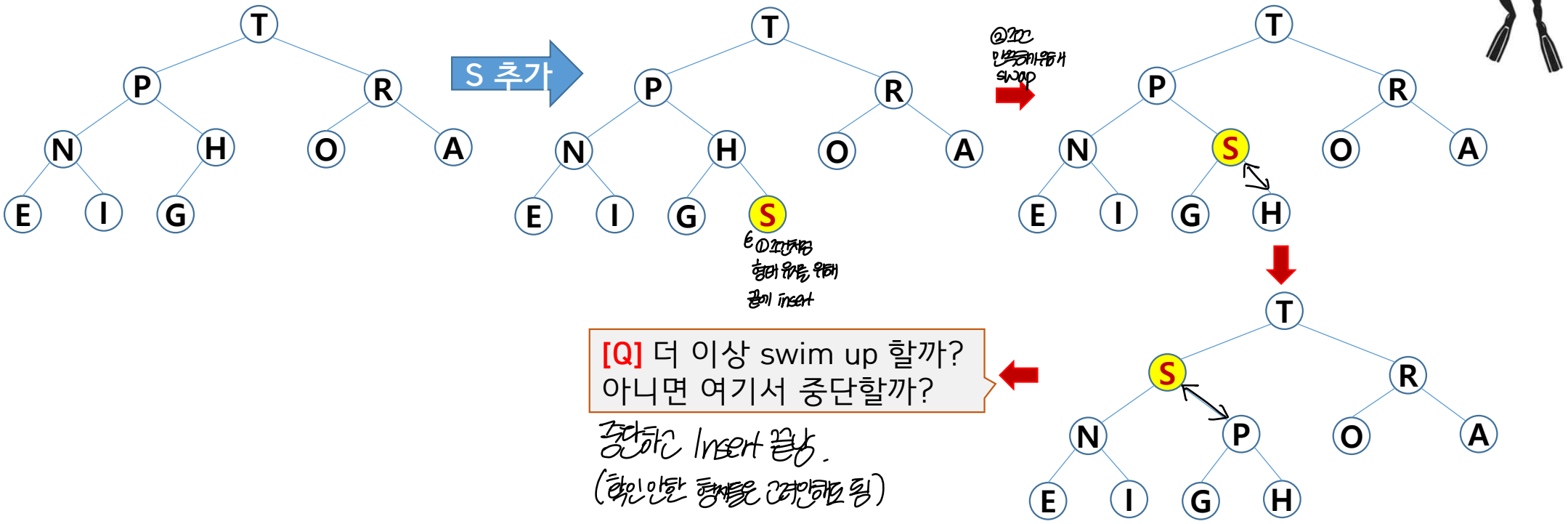
④

i	0	1	2	3	4	5	6	7	8	9	10
a[i]	-	30	27	23	~~17~~	16	15	13	14	~~18~~	11

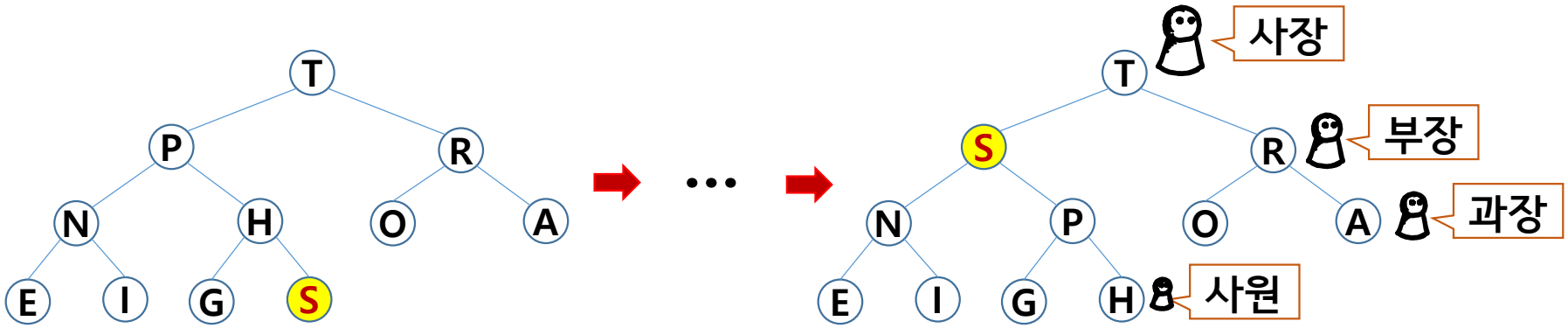
insert(): ① complete binary tree 형태 유지하며 ②heap order 유지하며 ③~logN 시간에 새 원소 추가

- insert(k) 하는 새 값을 k라 할 때
- k를 tree 가장 마지막에 (배열 끝에) 추가
- k를 부모 노드와 비교해 heap-order (부모≥자식) 어긴다면 swap하는 것 반복 (swim up)

  (즉, heap-order 만족할 때까지 부모 노드 따라 계속 올라감)



insert(): heap order 만족할 때까지 swim up (=swap up) ≒ 능력에 맞는 위치까지 진급

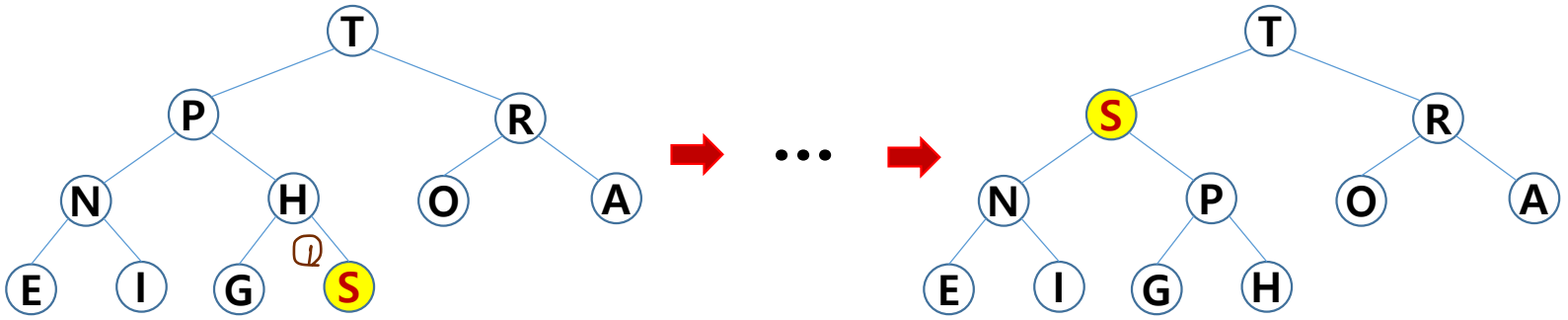


- max heap order(부모≥자식)를 능력에 따른 직장에서의 직급에 비유하면
- insert()시 swim up 하는 것은 능력치에 맞는 위치까지 진급하는 것으로 볼 수 있음
- 이상적인 구조 ☺

insert(): 끝에 추가 후 heap order (부모≥자식) 만족할 때까지 swim up

```
class MaxHeap:
    def __init__(self): # Constructor
        self.pq = ['']  # Empty key at pq[0]

    def insert(self, key):
      ① self.pq.append(key)  # 새 원소 key를 배열 맨 끝에 추가
        idx = len(self.pq)-1 # idx = 새 원소가 추가된 index
        while idx>1 and self.pq[idx//2] < key:  # 부모≥자식 조건 만족할 때까지 반복
            self.pq[idx], self.pq[idx//2] = self.pq[idx//2], self.pq[idx] # key와 부모 swap
            idx = idx//2     # key의 index를 부모가 있던 위치로 변경
```

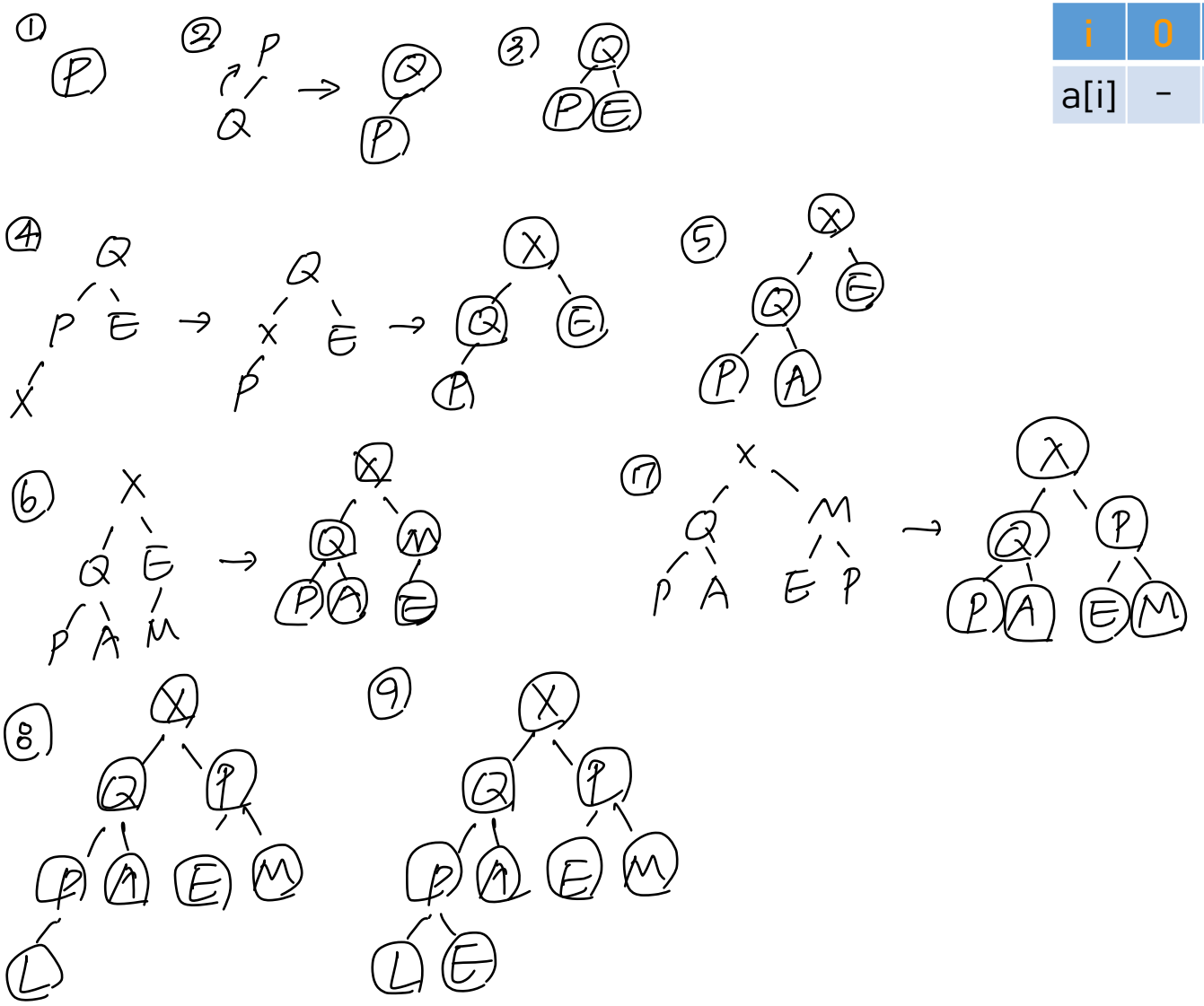
i	0
a[i]	-



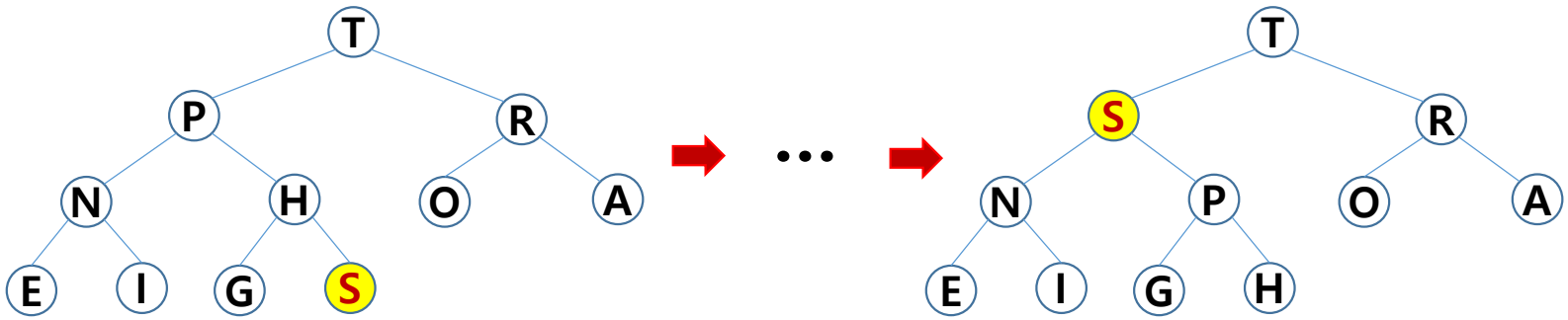
[Q] 비어 있는 max heap에 다음 순서로 원소를 추가했다. 마지막 원소까지 추가했을 때의 tree를 그려 보시오. 또한 이 tree의 배열 표현에는 어떤 값이 있을지도 써 보시오.

~~P, Q, E, X, A, M, P~~, L, E

i	0	1	2	3	4	5	6	7	8	9
a[i]	-	X	Q	P	P	A	E	M	L	E



insert(): 새 원소 마지막에 추가하고 heap order (부모≥자식) 만족할 때까지 swim up



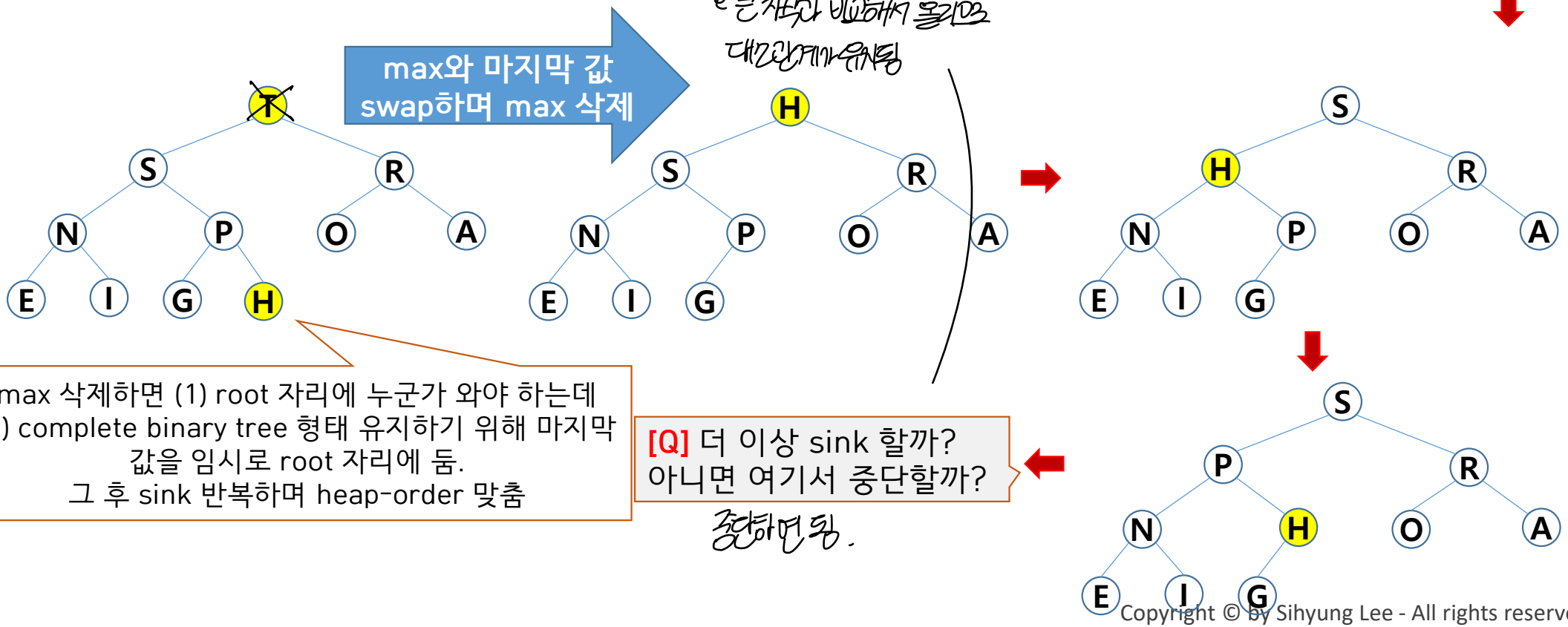
[Q] insert() 하면 ①heap 형태 (complete binary tree) 유지되나? yes

[Q] insert() 하면 tree 전체 ② heap order 유지되는가? 즉, **새 노드 ~ root 까지의 경로 따라서만 swim up (swap)** 해주는데, tree 다른 부분과의 heap order도 유지되나? yes

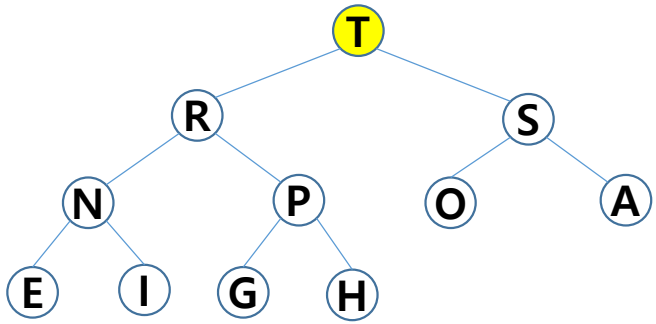
[Q] insert() 비용은 ③ ~logN으로 제한되는가? yes
insert()가 수행하는 작업은 대소비교 & swap

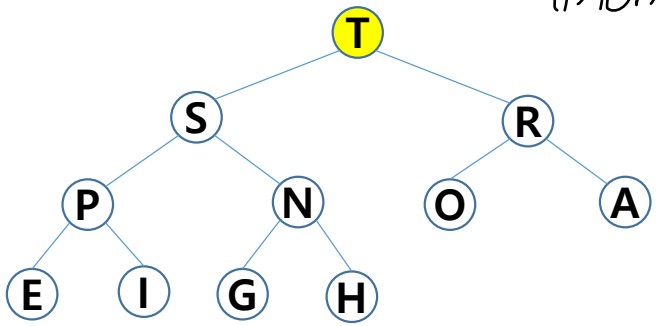
delMax(): ① complete binary tree 형태 유지하며 ②heap order 유지하며 ③~logN 시간에 max 삭제 (≒ sink)

- root 값을 tree 가장 마지막 값 k와 swap하며 배열에서 삭제
- k를 자식 노드 중 큰 쪽과 비교해 heap-order (부모≥자식) 어긴다면 swap하는 것 반복 (sink)
  (즉, heap-order 만족할 때까지 자식 노드 중 큰 쪽 따라 계속 내려감) : ~logN



[Q] delMax()할 때 root 삭제한 후, 자식 중 더 큰 쪽을 swim up 시키는 것 반복하면 안되나? 왜 마지막 원소를 root에 두고 sink 시키나? complete한 모양을 유지하기 위함.



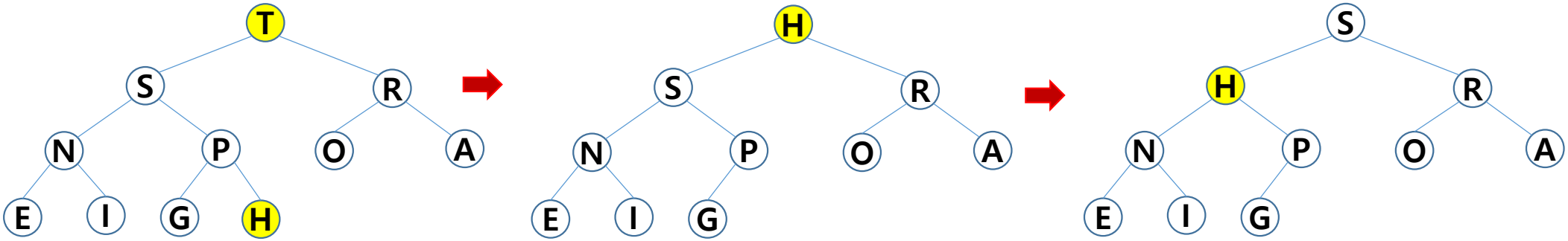


dexMax(): 마지막 원소 root에 두고 heap order (부모≥자식) 만족할 때까지 sink

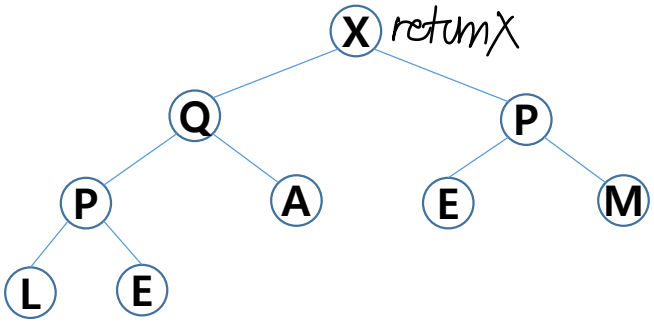

```
def delMax(self):
  # root(max)와 마지막 원소 k swap
  self.pq[1], self.pq[len(self.pq)-1] = self.pq[len(self.pq)-1], self.pq[1]
  max = self.pq.pop() # 마지막에 있는 max 값 제거
  idx = 1 # 마지막 원소 k가 있는 root에서 sink 시작
  while 2*idx <= len(self.pq)-1: # 아래에 자식이 있다면 계속 sink 시도
    idxChild = 2*idx
    if idxChild<len(self.pq)-1 and self.pq[idxChild] < self.pq[idxChild+1]:
      idxChild = idxChild+1  # 두 자식 중 더 큰 자식 찾기
    if self.pq[idx] >= self.pq[idxChild]: break   # heap order 만족하면 sink 중단
    # k와 더 큰 자식 swap함으로써 sink
    self.pq[idx], self.pq[idxChild] = self.pq[idxChild], self.pq[idx]
    idx = idxChild # k의 index를 자식이 있던 위치로 변경
  return max
```

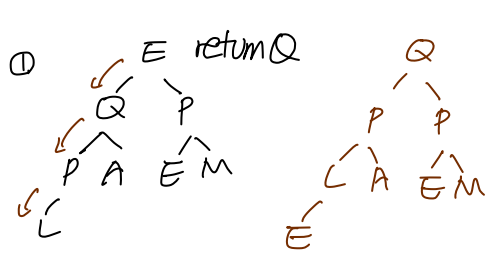
더 큰게 있는 부분 제외하고 insert 바꿔줌.

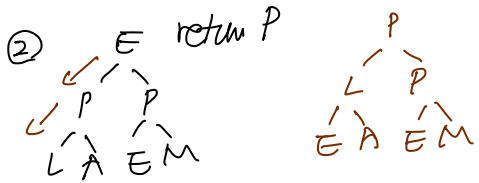
[Q] 왜 더 큰 자식 찾나?



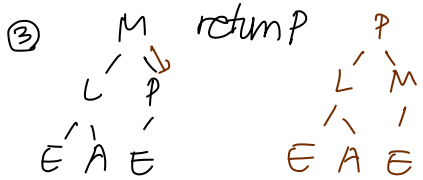
[Q] 아래와 같이 9개의 원소를 가진 max heap에 delMax()를 9번 수행하였다. delMax()를 수행할 때마다 (1) return되는 값을 표시하고 (2) tree가 변해가는 과정을 그려 보시오.

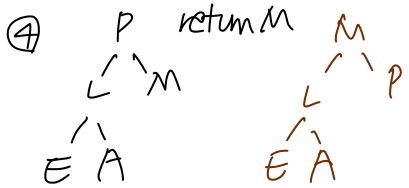






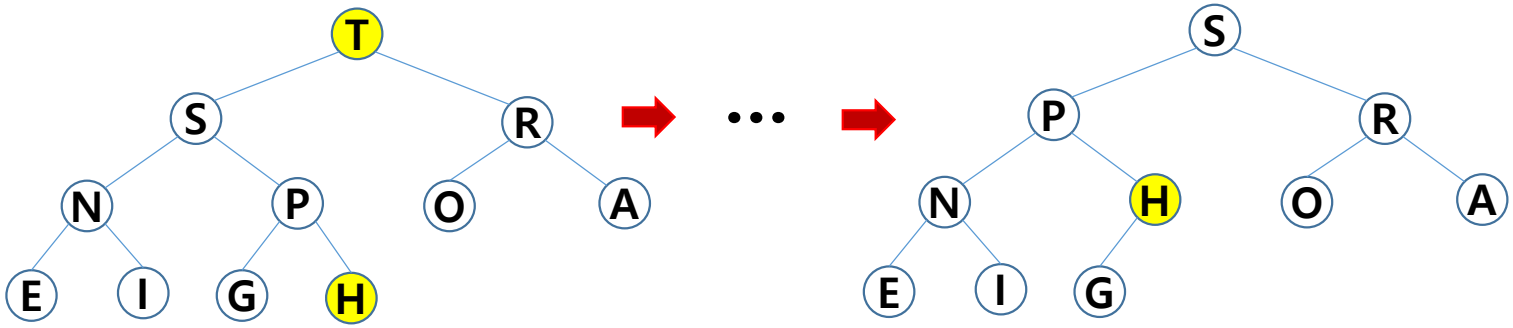
"큰것끼리 비교"





지금 수행한 과정이 Heap Sort와 유사.
N개 원소로 Heap 만든 후 → max (or min)을 제거한 순서대로 놓으면 정렬됨

delMax(): 마지막 원소를 root에 옮기고 heap order (부모≥자식) 만족할 때까지 sink



[Q] delMax() 하면 ①heap 형태 (complete binary tree) 유지되나? Yes

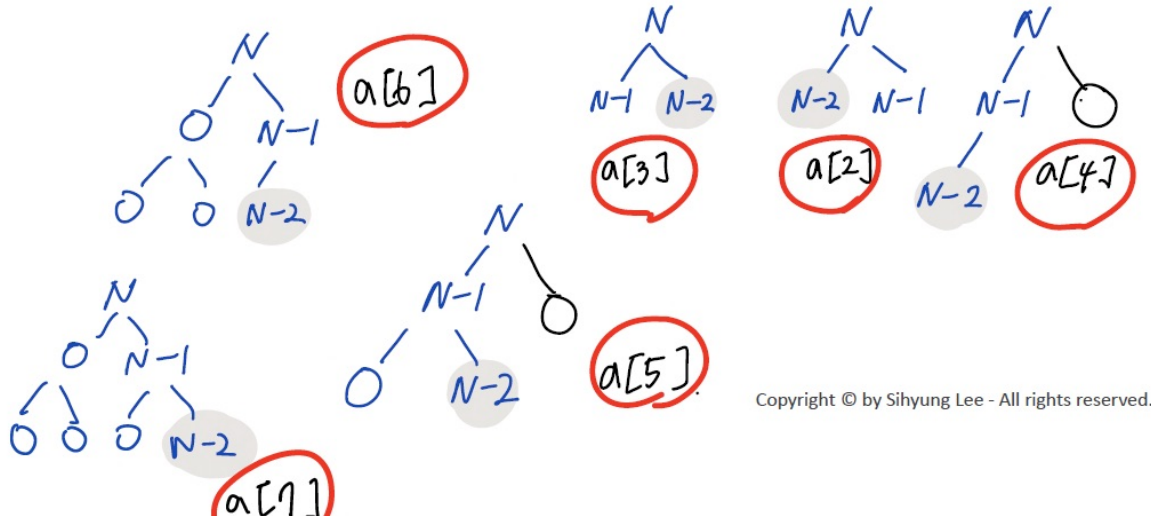
[Q] delMax() 하면 tree 전체의 ② heap order 유지되는가? 즉, root ~ 더 큰 자식을 따른 경로에서만 sink (swap) 해주는데, tree 다른 부분과의 heap order도 유지되나? Yes, Why

[Q] delMax() 비용은 ③ ~logN으로 제한되는가? Yes
delMax()가 수행하는 작업은 대소비교 & swap

[Q] 배열 a[]은 N개의 원소 1, 2, … , N(≥7)을 max heap 형태로 저장하고 있다. (이들이 어떤 순서로 heap에 추가되었는지는 모름) 이 중 가장 큰 원소인 N은 a[1]에 있을 것이며, 다음으로 큰 원소인 N-1은 a[2]나 a[3]에 있을 것이다. 그렇다면 3번째로 큰 원소인 N-2는 다음 중 어느 위치에 있을 수 있는가?

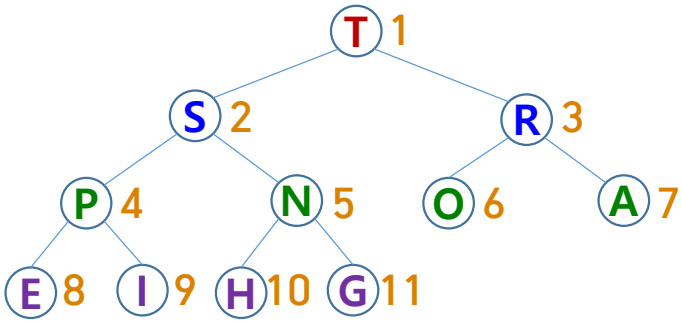
① a[2] 또는 a[3]
② a[4], a[5], a[6] 또는 a[7]
③ a[2], a[3], a[4], a[5], a[6], 또는 a[7]
④ a[1], a[2], a[3], a[4], a[5], a[6], 또는 a[7]

i	0	1	2	3	4	5	6	7	8	9
a[i]	-	N								



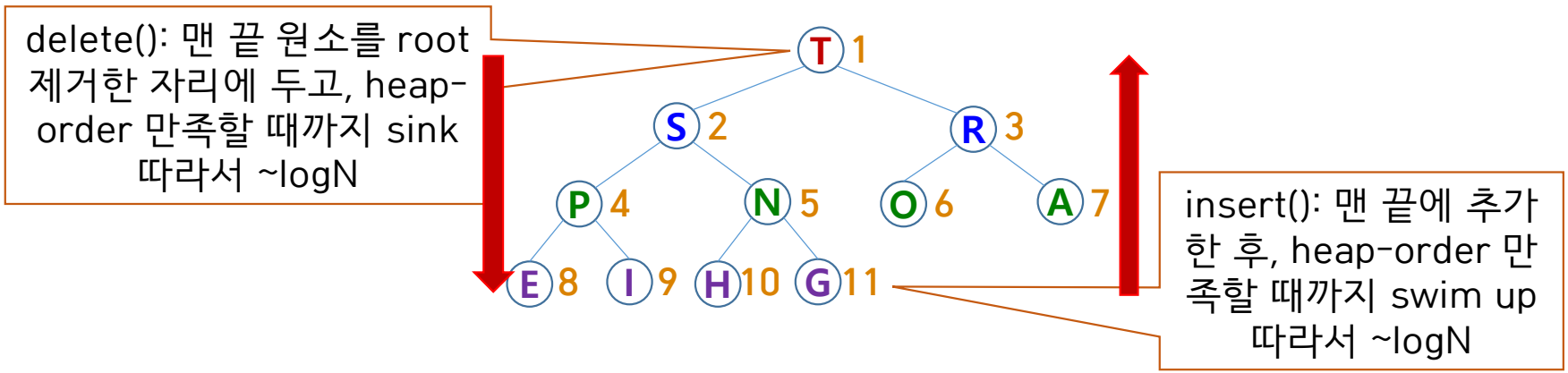
Heap 정리: heap order 따라 원소 배치한 complete binary tree

- Priority Queue의 효율적인 구현 방법
- (Max) heap order: 부모 key ≥ 자식 key
  - 따라서 어떤 노드라도 그 아래로 뻗어 나간 모든 노드 중 max. Delete(max 찾기) 쉽게 함
- Complete binary tree: 왼쪽부터 차례로 빈칸 없이 채워간 이진 트리
  - 실제 트리 만들지 않고 배열 사용해 편리하게 indexing 가능



Heap 정리: heap order 따라 원소 배치한 complete binary tree

- Priority Queue의 효율적인 구현 방법
- (Max) heap order: 부모 key ≥ 자식 key
  - 따라서 어떤 노드라도 그 아래로 뻗어 나간 모든 노드 중 max. Delete(max 찾기) 쉽게 함
- Complete binary tree: 왼쪽부터 차례로 빈칸 없이 채워간 이진 트리
  - 실제 트리 만들지 않고 배열 사용해 편리하게 indexing 가능
  - 트리 깊이 ~logN 으로 제한해 insert, delete 비용도 ~logN으로 제한



Priority Queue에 대한 구현 방법 비교 정리

구현 방식	insert() 비용	delMin() 비용
Unordered list	~1	~N
Ordered array	~N	~1
Binary Heap	**~$\log_2 N$**	**~$\log_2 N$**
d-ary Heap	**~$\log_d N$**	**~d × $\log_d N$**
불가능	**~1**	**~1**

자식 d개인 heap으로 binary heap의 일반화

sink할 때는 d개 자식 중 가장 큰 자식 찾아야 함

swim up은 부모와만 비교

Unordered list delMin이 ~1될 수 없음

Ordered array insert()가 ~1될 수 없음

Priority Queue

PQ가 무엇이고, 어떻게 구현하며, 어떤 경우에 어떻게 활용하는지 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Binary Heap 사용한 PQ의 구현
03. Priority Queue 사용 어플리케이션의 공통적인 특성
04. Slider Puzzle and A* Search with PQ
05. 실습: PQ를 사용한 Slider Puzzle Solver 구현

Priority Queue 활용(에 적합한) 어플리케이션의 공통점

- 가장 우선순위 높은 원소 계속 사용하며 진행 (delMin(), delMax())

- 처음부터 모든 원소 다 알 수 없으며, 작업 수행 중 새 원소 계속 추가됨 (insert())
  - 처음부터 모든 원소 다 얻을 수 있다면 정렬 사용해 해결 가능
  - iteration 진행하며 계속 새로운 원소 추가되므로, 효율적인 insert() 방법 필요

- PQ 사용 어플리케이션 예: Top K 찾기

- 입력이 실시간으로 계속 들어오며, 입력을 모두 받을 때까지는 원소 개수 N이 무엇인지 미리 알지 못함
- 어느 순간에도 그때까지 받은 원소 중 Top K 확인 가능해야 함 & 모든 원소를 다 저장하기에는 메모리 부족 → 저장한 입력이 K개에 도달하면, 새 입력 들어올 때마다 가장 작은 원소 삭제

Priority Queue 활용(에 적합한) 어플리케이션의 공통점

- 가장 우선순위 높은 원소 계속 사용하며 진행 (delMin(), delMax())

- 처음부터 모든 원소 다 알 수 없으며, 작업 수행 중 새 원소 계속 추가됨 (insert())
  - **처음부터 모든 원소 다 얻을 수 있다면 정렬 사용해 해결 가능**
  - iteration 진행하며 계속 새로운 원소 추가되므로, 효율적인 insert() 방법 필요

[Q] N개 입력 모두를 한 번에 받을 수 없고 실시간으로 계속 들어온다면, PQ 대신 정렬을 사용하는 것 더 효율적인가?

- PQ 사용 어플리케이션 예: Top K 찾기

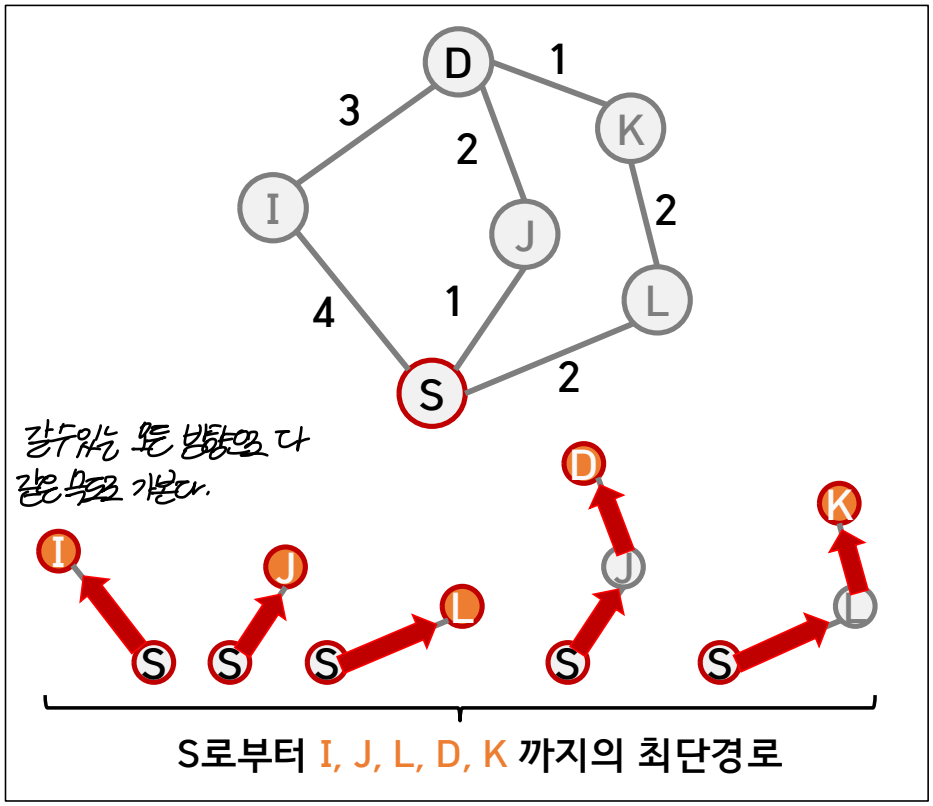
- 입력이 실시간으로 계속 들어오며, 입력을 모두 받을 때까지는 원소 개수 N이 무엇인지 미리 알지 못함
- 어느 순간에도 그때까지 받은 원소 중 Top K 확인 가능해야 함 &모든 원소를 다 저장하기에는 메모리 부족 → 저장한 입력이 K개에 도달하면, 새 입력 들어올 때마다 가장 작은 원소 삭제

[Q] N개 입력 모두를 한 번에 받아 저장할 수 있다면, 정렬로 ~NlogN 시간에 해결 가능한가?

반대로 정렬 사용해도 되는 상황이라면 PQ 사용해도 큰 문제는 없음

- 오름차순 (혹은 내림차순)으로 원소 하나씩 꺼내 사용하기 위해
- 정렬하는데 ~N log N 시간 걸린다면
- PQ 사용해도
- ~logN 작업을 ~N회 수행하므로

Dijkstra의 최단 경로 알고리즘:
하나의 출발지 S로부터 여러 다른 목적지에 대한 최단경로 한 번에 찾는 방법



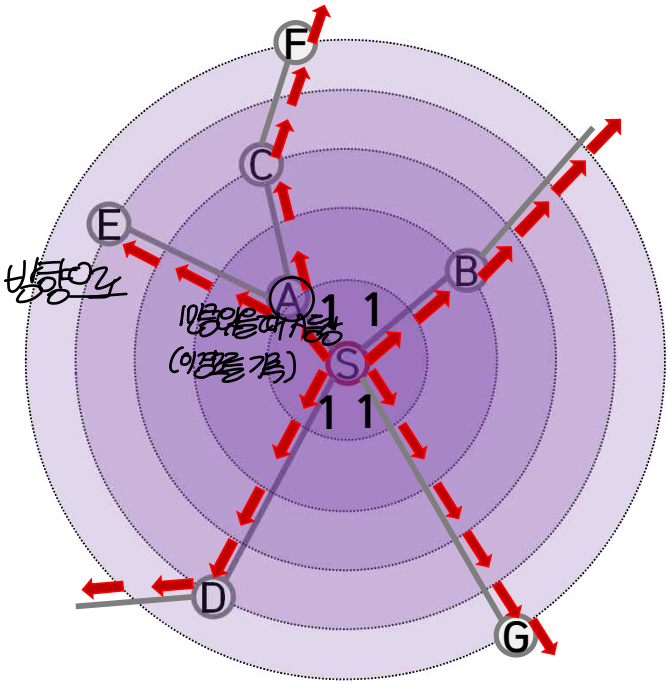
- [Q] 왜 모든 목적지에 대한 경로를 계산할까? [A] 한 목적지에 대한 경로만 갖고 있는 것보다 더 유용하기 때문

라우터의 라우팅 테이블	
목적지 IP	전송 링크 #
1.*.*.*	1
2.*.*.*	2
3.*.*.*	3
4.*.*.*	4
5.*.*.*	5
…	…

- 가정: link cost ≥ 0

① 모든 방향으로 ② 같은 속도로 탐색

- 탐색 중 기존에 못 본 새로운 목적지 α 발견하면, 그때까지 거쳐 온 경로를 α 까지 최단경로로 기록

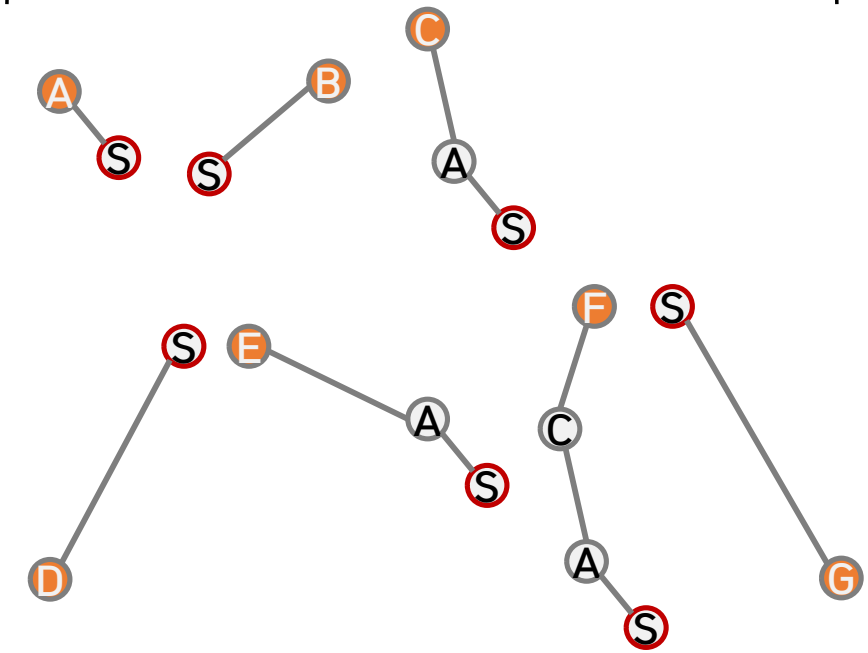


출발지에서 모든 방향으로 같은 속도로 감

1일때 처음 방문 (이경로 기록)

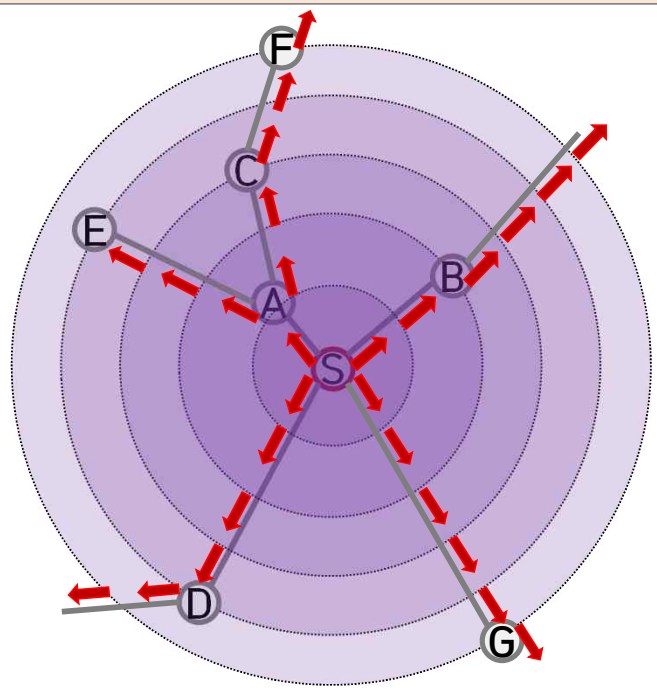
다음 길이의 모든 경로 탐색: 1 2 3 4 5

지금까지 발견한 최단 경로



① 모든 방향으로 ② 같은 속도로 카펫을 펼치는 것(혹은 구슬 굴리는 것)과 유사

- 점진적으로 S로부터 더 멀리 있는 (더 긴) 최단 경로 찾게 됨 (예: 길이 1→2→3→4→…)

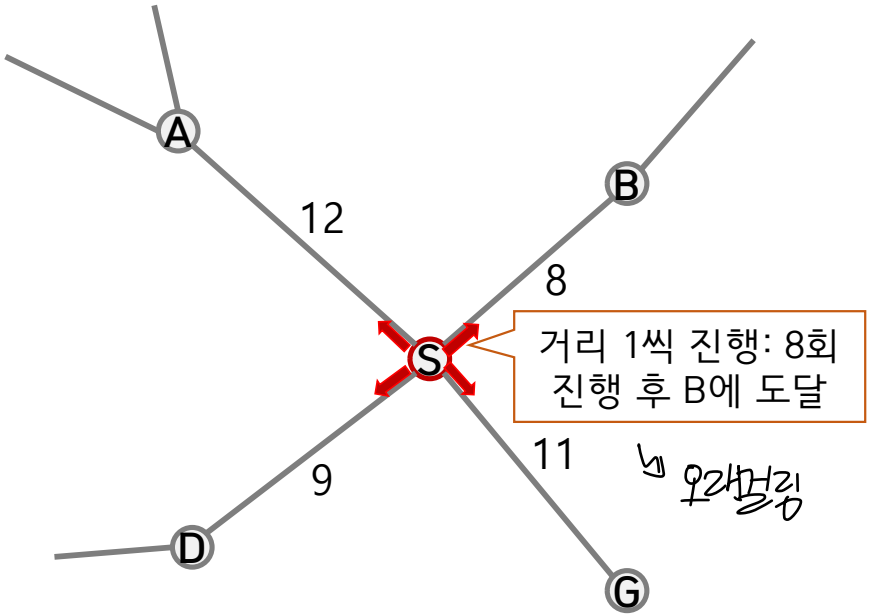


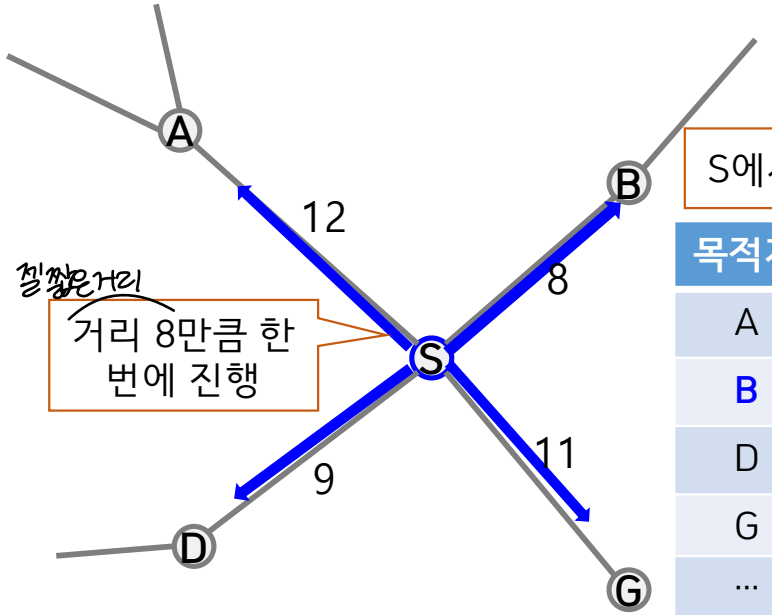
다음 길이의 모든 경로 탐색: 1 2 3 4 5

개선

거리 1씩 혹은 작은 값 △단위로 진행:
다음 도달 가능한 목적지까지 거리 먼 경우
(>> △) 시간 오래 걸림

지금까지 알게 된 경로 중
가장 먼저 도달할 수 있는 곳 선정해
그 거리만큼 한 번에 진행





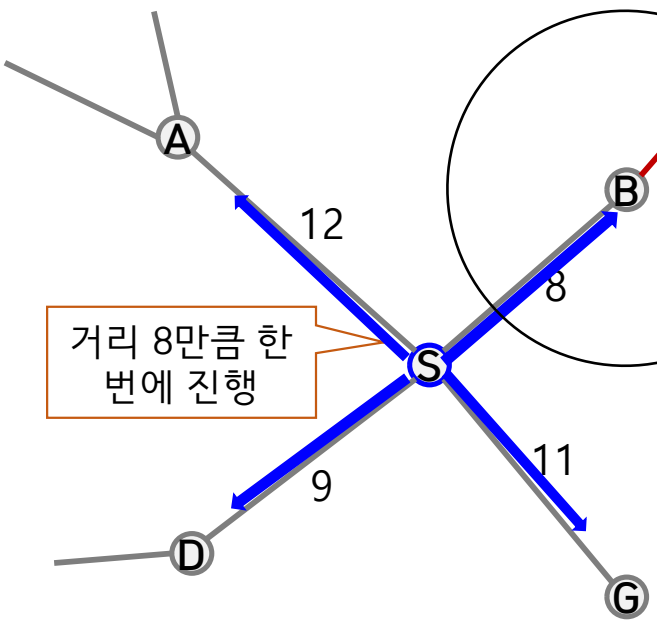
S에서 알고 있는 경로

목적지	S로부터 거리
A	12
B	8 (1st)
D	9
G	11
…	…

minimum PQ 사용

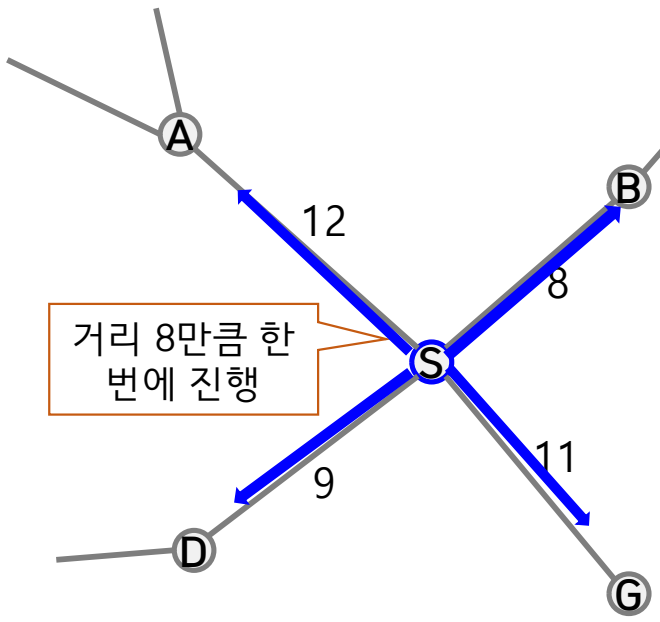
지금까지 알게 된 경로 중
가장 먼저 도달할 수 있는 곳 선정해
그 거리만큼 한 번에 진행

방금 도달한 곳에서 새롭게
알게 된 경로 표에 추가



지금까지 알게 된 경로

목적지	S로부터 거리
A	12
D	9
G	11
T	13 (=8+5) (2nd)
…	…



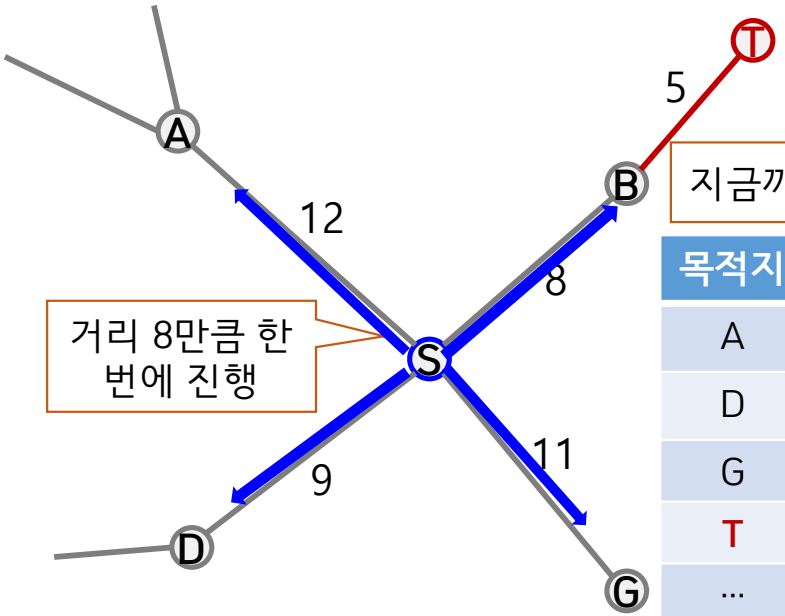
S에서 알고 있는 경로

목적지	S로부터 거리
A	12
B	8 (1st)
D	9
G	11
…	…

위 두 단계를
모든 목적지 다 도달할 때까지 반복

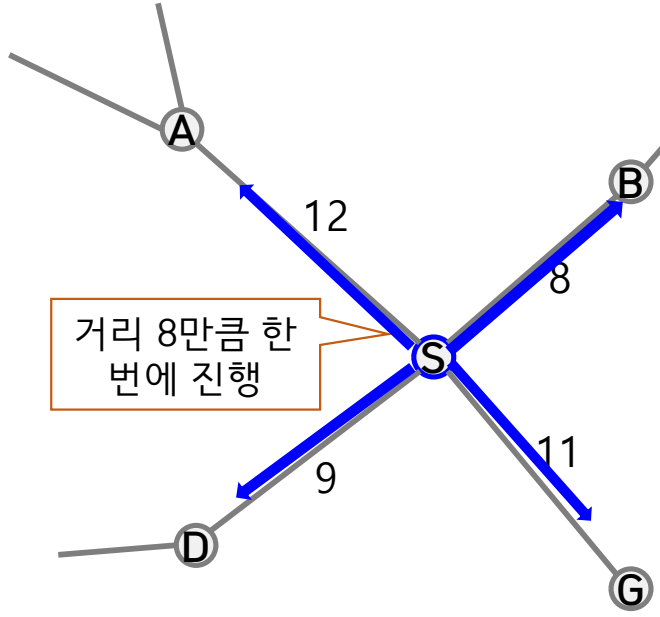
방금 도달한 곳에서 **새롭게 알게 된 경로 표에 추가**

지금까지 알게 된 경로 중 가장 먼저 도달할 수 있는 곳 선정해 그 거리만큼 한 번에 진행



지금까지 알게 된 경로

목적지	S로부터 거리
A	12
D	9
G	11
T	13 (=8+5)
…	…



S에서 알고 있는 경로

목적지	S로부터 거리
A	12
B	**8**
D	9
G	11
…	…

- 모든 목적지에 대한 최단경로 찾아가는 중 **알게 되는 새로운 경로를 계속 표에 추가하며 진행**하며, 처음부터 모든 경로를 미리 다 갖고 시작하지 않음

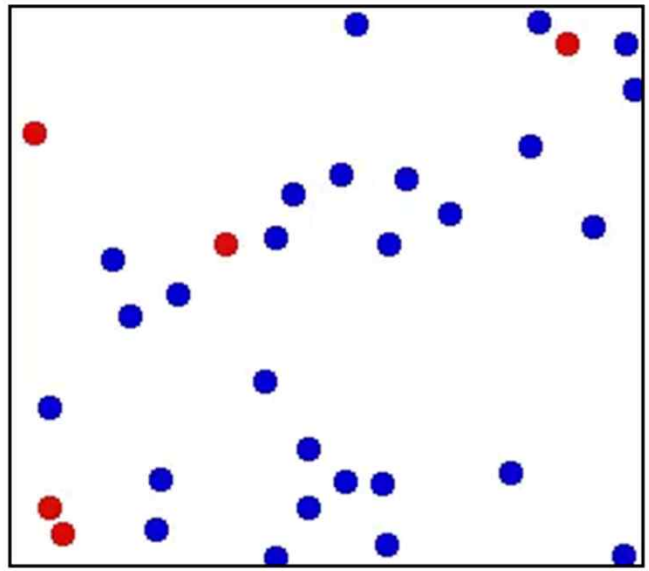
- 현재까지 알고 있는 경로 중 **가장 우선순위 높은 원소 (S로부터 거리 가장 작은 원소) 계속 사용하며 진행**

Priority Queue 사용하기 적절한 상황:
그때그때 새로 알게 된 경로를 Priority Queue에 추가하며 진행하면 됨

닫힌 공간에 있는 N개의 이동&충돌하는 입자의 시뮬레이션

- N개 입자의 초기 위치와 속도가 주어졌을 때
- 이들의 위치와 속도 변화 관찰

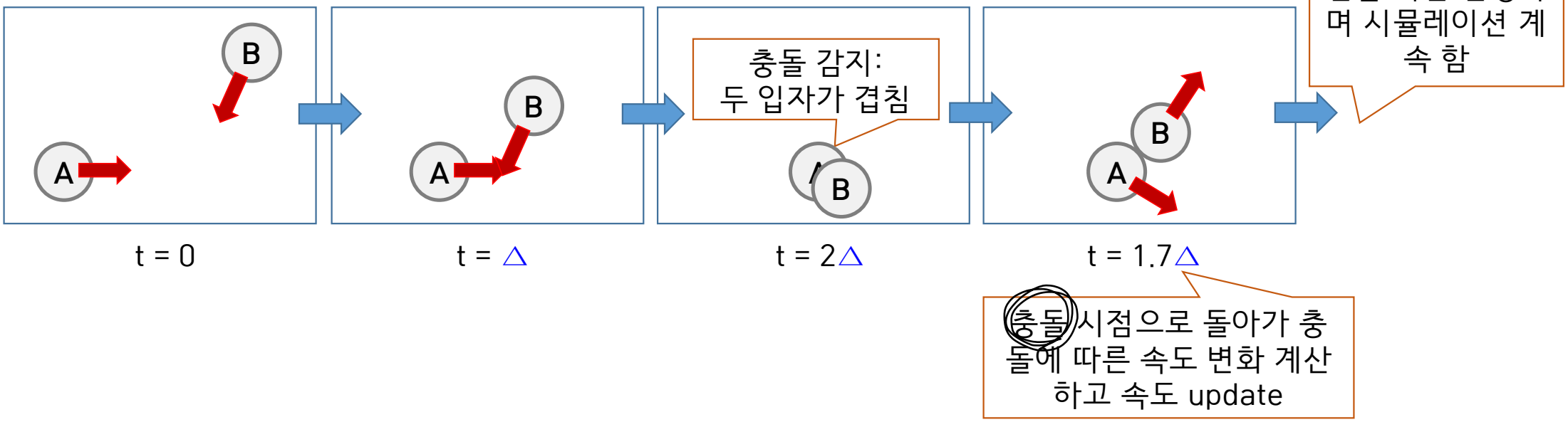
- 다양한 물리 실험에 활용
- 예: 분자 수, 무게, 닫힌 공간의 크기, 초기 온도 등이 주어졌을 때, 시간에 따른 분자의 이동 속도 및 온도 변화 관찰
- …



https://www.toppr.com/ask/content/story/amp/molecular-motion-122180/

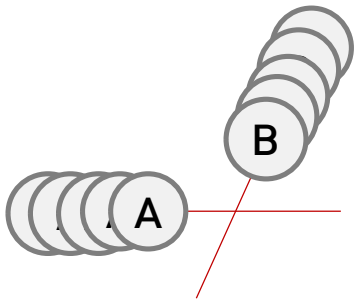
Time-driven Simulation

- 작은 **시간 값** △단위로 진행
- 매 iteration 마다 △만큼 시간 진행했다고 보고 각 입자의 새 위치 계산
  - 만약 두 입자가 겹친다면(overlap) 충돌한 것으로 보고
  - 충돌 시점으로 돌아가 (roll back the clock) 두 입자의 속도 update

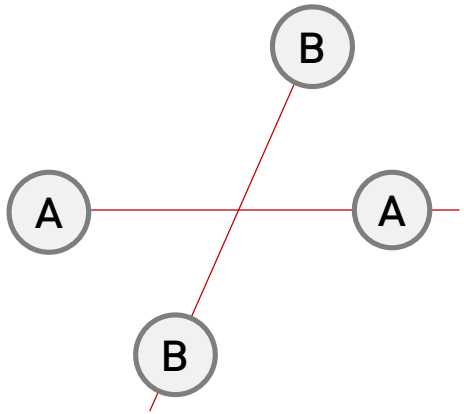


Time-driven Simulation 문제:
△작으면 시간 오래 걸림, △크면 충돌 감지 못하는 경우 많아짐

- △ 단위로 각 입자의 위치 계산 & 충돌 여부 확인 하므로
- △ **작을수록 계산 많아져** 시뮬레이션 시간 오래 걸림 (느려짐)
- 단 이벤트(충돌)는 잘 감지



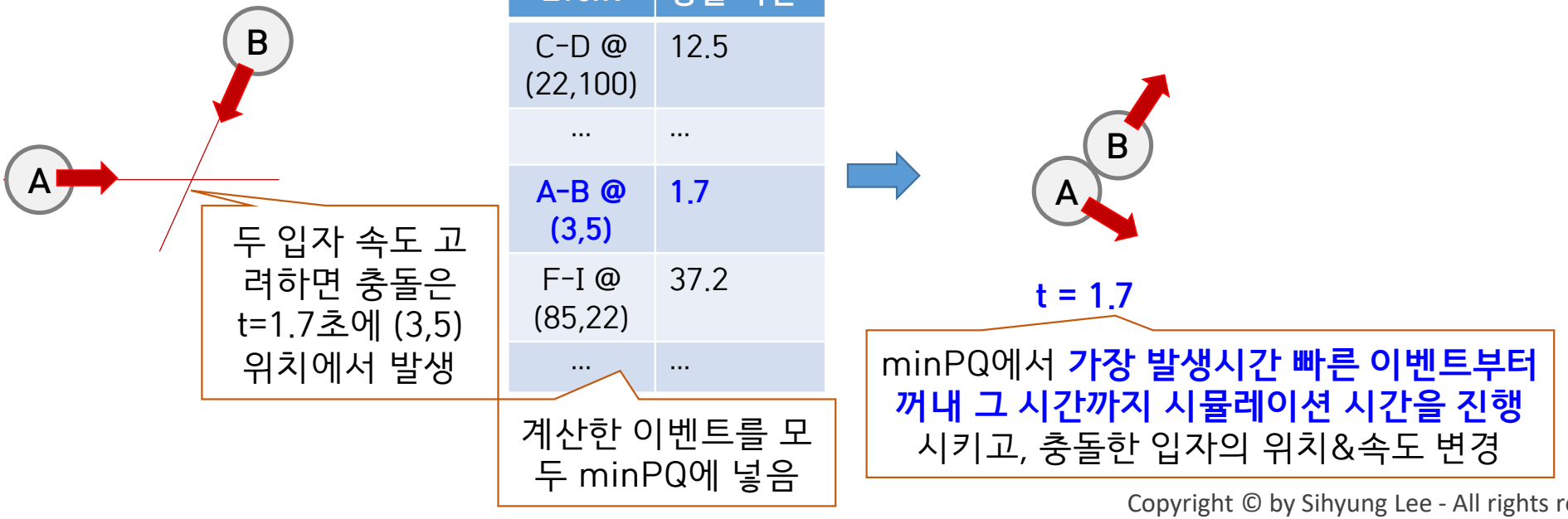
- 시뮬레이션 속도 높이기 위해 **△를 큰 값**으로 설정하면
- **감지 못하는 이벤트(충돌) 많아짐**



Event-driven Simulation

- 시간 단위로 진행하지 않고 **Event(충돌) 단위로 진행**
  - 입자들의 속도 고려하면 충돌하는 시간 계산할 수 있음
  - 두 입자 쌍 간의 **충돌 시간**을 **minPQ**에 넣음 (key는 충돌 시간)
  - **충돌 시간 빠른 event부터 minPQ에서 꺼내 처리**(충돌하는 입자의 속도 update)
  - 속도 update한 입자가 이후 다른 입자와 충돌하는 시간 계산해 minPQ에 추가

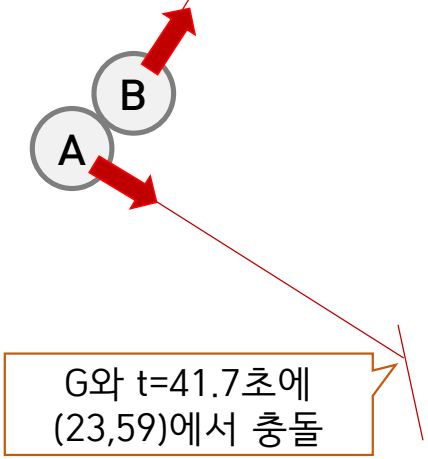
가장 먼저 일어나는 충돌부터 찾아서 PQ에



방금 충돌한 입자가 속도 변경 후 다른 입자와 충돌할 시간 계산해 PQ에 추가

지금까지 알고 있는 충돌 event 중 가장 먼저 발생하는 event 선정해 그 시간까지 한 번에 진행

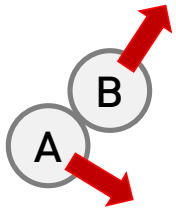
위쪽 벽과 t=23.6초에 (40,0)에서 충돌



G와 t=41.7초에 (23,59)에서 충돌

Event	충돌 시간
C-D @ (22,100)	12.5
B-wall @ (40,0)	23.6
A-G @ (23,59)	41.7
F-I @ (85,22)	37.2
…	…

t=1.7까지 바로 진행하고 충돌하는 입자의 위치& 속도 update



t = 1.7

지금까지 알고 있는 충돌 event

Event	충돌 시간
C-D @ (22,100)	12.5
…	…
A-B @ (3,5)	1.7
F-I @ (85,22)	37.2
…	…

- 시뮬레이션 중 **알게 되는 새로운 이벤트를 계속 PQ에 추가하며 진행**하며, 처음부터 모든 이벤트를 다 갖고 시작하지 않음

- 현재까지 알고 있는 원소 중 **가장 우선순위 높은 원소 (발생시간 가장 빠른 event) 계속 사용하며 진행**

Priority Queue 사용하기 적절한 상황: 그때그때 새로 알게 된 경로를 Priority Queue에 추가하며 진행하면 됨

PQ 활용하는 경우의 공통점 정리

- 지금까지 알게 된 지식의 집합 Knowledge{}에 기반해
- 가장 우선순위 높은 상태 p로 진행하고 (혹은 우선순위 높은 action 수행)
- p로 진행함으로써 알게 된 새로운 지식도 Knowledge{}에 추가해
- 목표 달성할 때까지 위 단계 반복

- 앞으로 수업에서도 PQ 사용한다면
- 이에 부합하는 경우임을 다시 떠올려 보자.

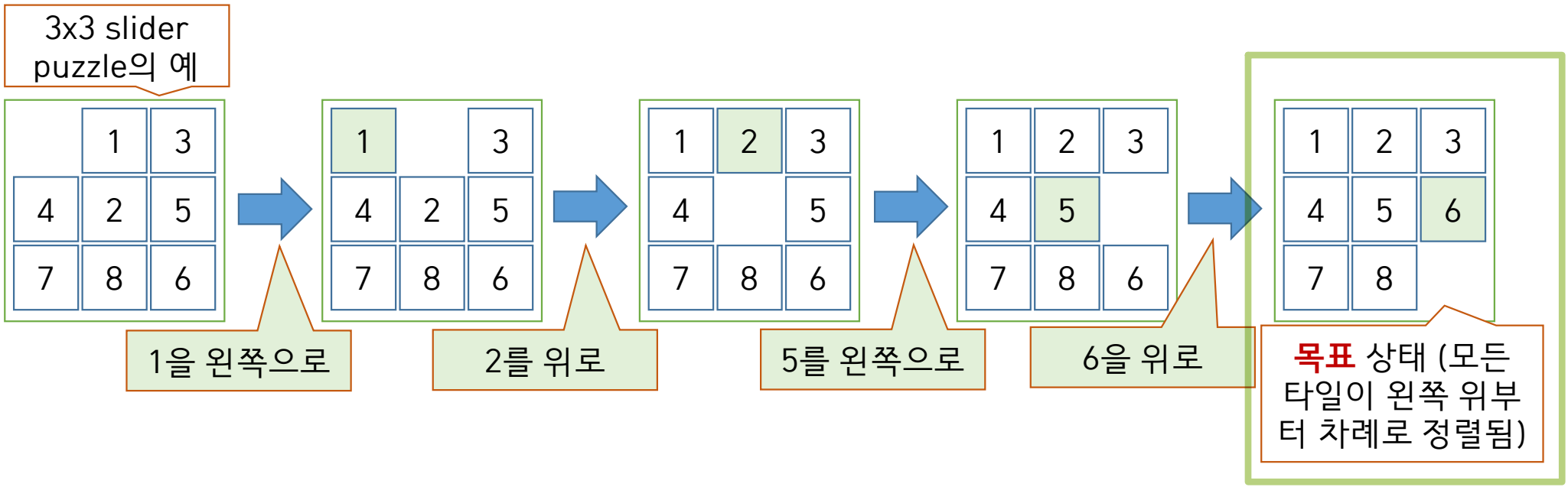
Priority Queue

PQ가 무엇이고, 어떻게 구현하며, 어떤 경우에 어떻게 활용하는지 이해

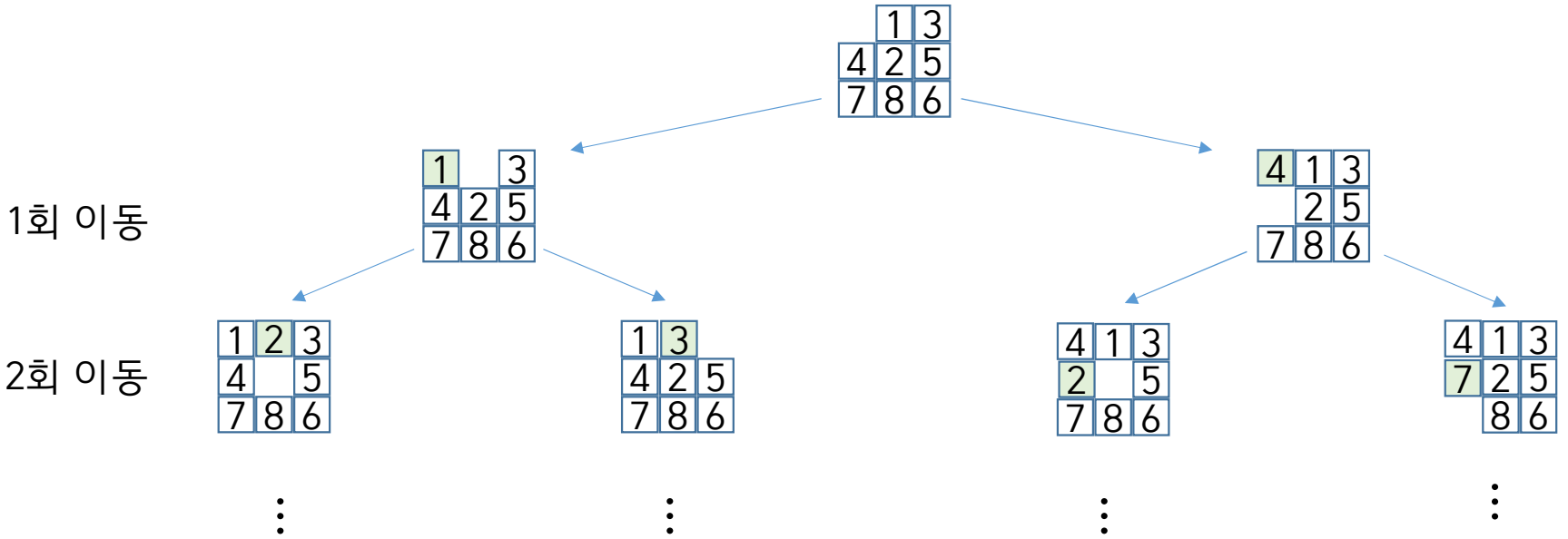
01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Binary Heap 사용한 PQ의 구현
03. Priority Queue 사용 어플리케이션의 공통적인 특성
04. Slider Puzzle and A* Search with PQ
05. 실습: PQ를 사용한 Slider Puzzle Solver 구현

실습: Slider Puzzle Solver 구현

- NxN slider puzzle:
- 1 ~ N^2-1 까지 타일과 하나의 빈 칸 있는 상태에서 시작
- (최소 횟수로) 모든 타일이 정렬된 상태로 이동하는 것이 목표
- 한 번에 한 타일을 빈 칸으로 slide해 위치 바꾸는 방식으로 이동



목표까지 가는 해 찾는 방법 (Brute Force): 모든 가능성을 목표 도달 시까지 트리 형태로 탐색 (게임 트리)



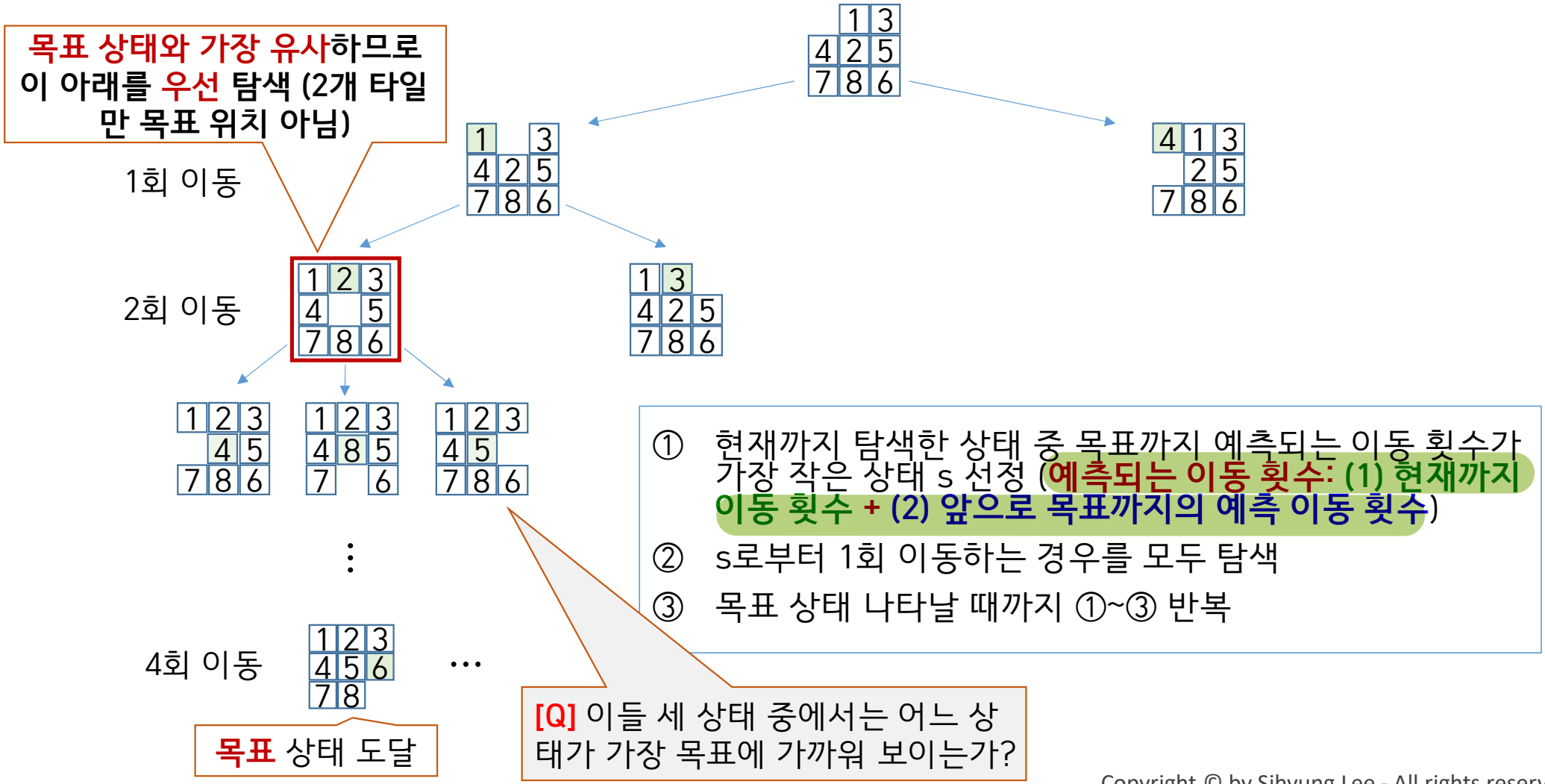
4회 이동 …

목표 상태 도달

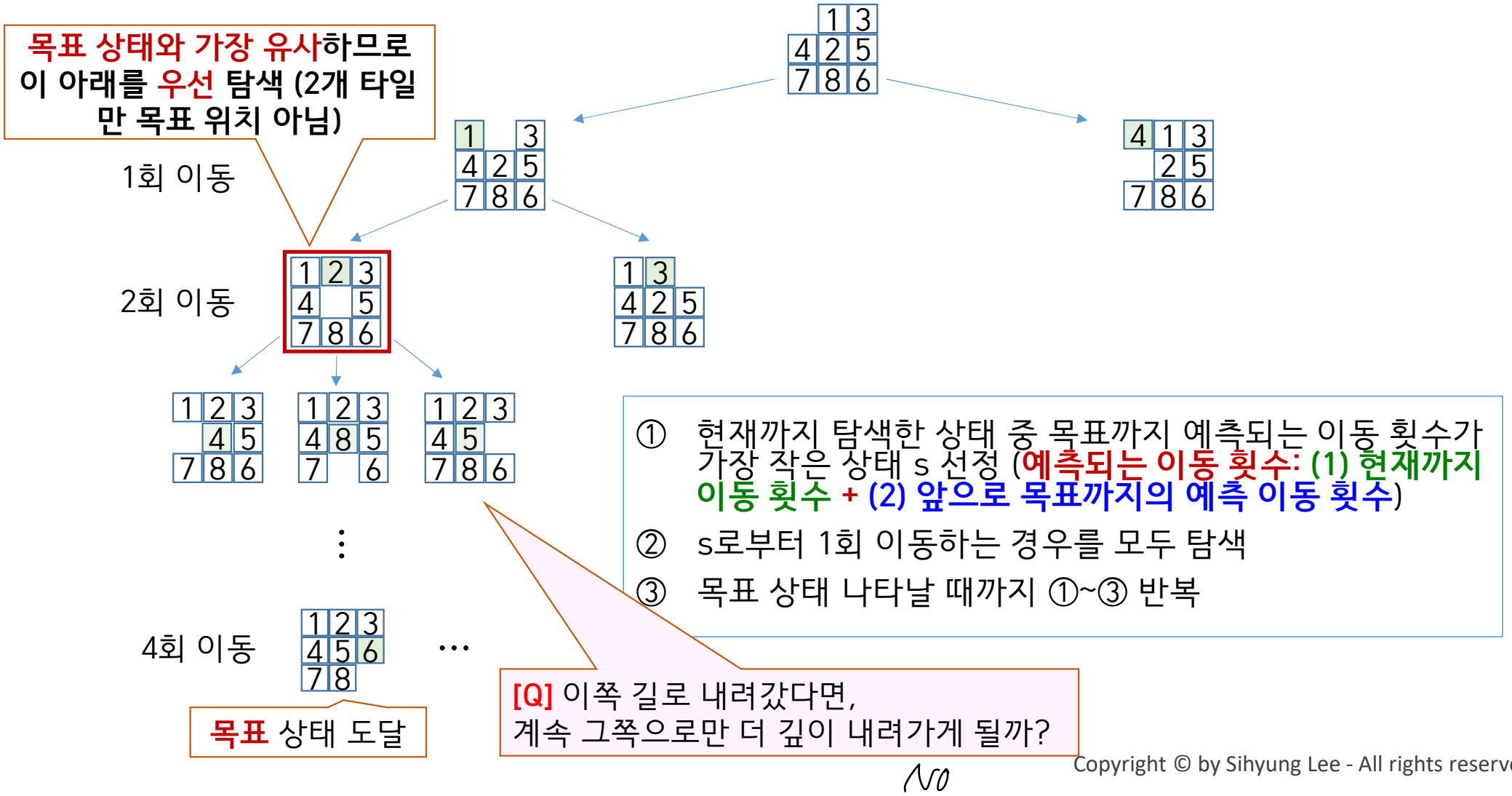
- 모든 가능한 경우를 이동 횟수 차례로 탐색 (1회 이동으로 가능한 경우들 → 2회 이동으로 가능한 경우들 → …)
- 바로 직전 상태로 돌아가는 이동은 하지 않음 (critical optimization)
- 탐색하는 경우 그려 보면 트리 형태 되며, 이를 '게임 트리'라 함

[Q] 각 depth마다 N가지의 가능한 경우 있다면,
depth=k까지 탐색해야 하는 상태 수는 무엇에 비례하는가?

목표까지 가는 해 좀더 빠르게 찾는 방법 (A* search, A-star로 발음):
목표까지 **예측되는 이동 횟수 가장 작은 경우부터 탐색**



목표까지 가는 해 좀더 빠르게 찾는 방법 (A* search, A-star로 발음):
목표까지 **예측되는 이동 횟수 가장 작은 경우부터** 탐색



A* Search의 활용 예: 다양한 어플리케이션에서 최단 경로 탐색

- 그래프에서 **시작점-목표지점까지 최단 경로** 찾기
  - 지도, 미로, …
- 게임에서 시작 상태-목표상태까지 최소 이동 방법 찾기
  - 바둑, 오목, 체스, …
- 합성하고자 하는 프로그램을 최소한의 명령어로 작성할 수 있는 방법 찾기
- …

Priority Queue 활용 어플리케이션의 공통점

- 가장 우선순위 높은 원소 계속 사용하며 진행 (delMin(), delMax())

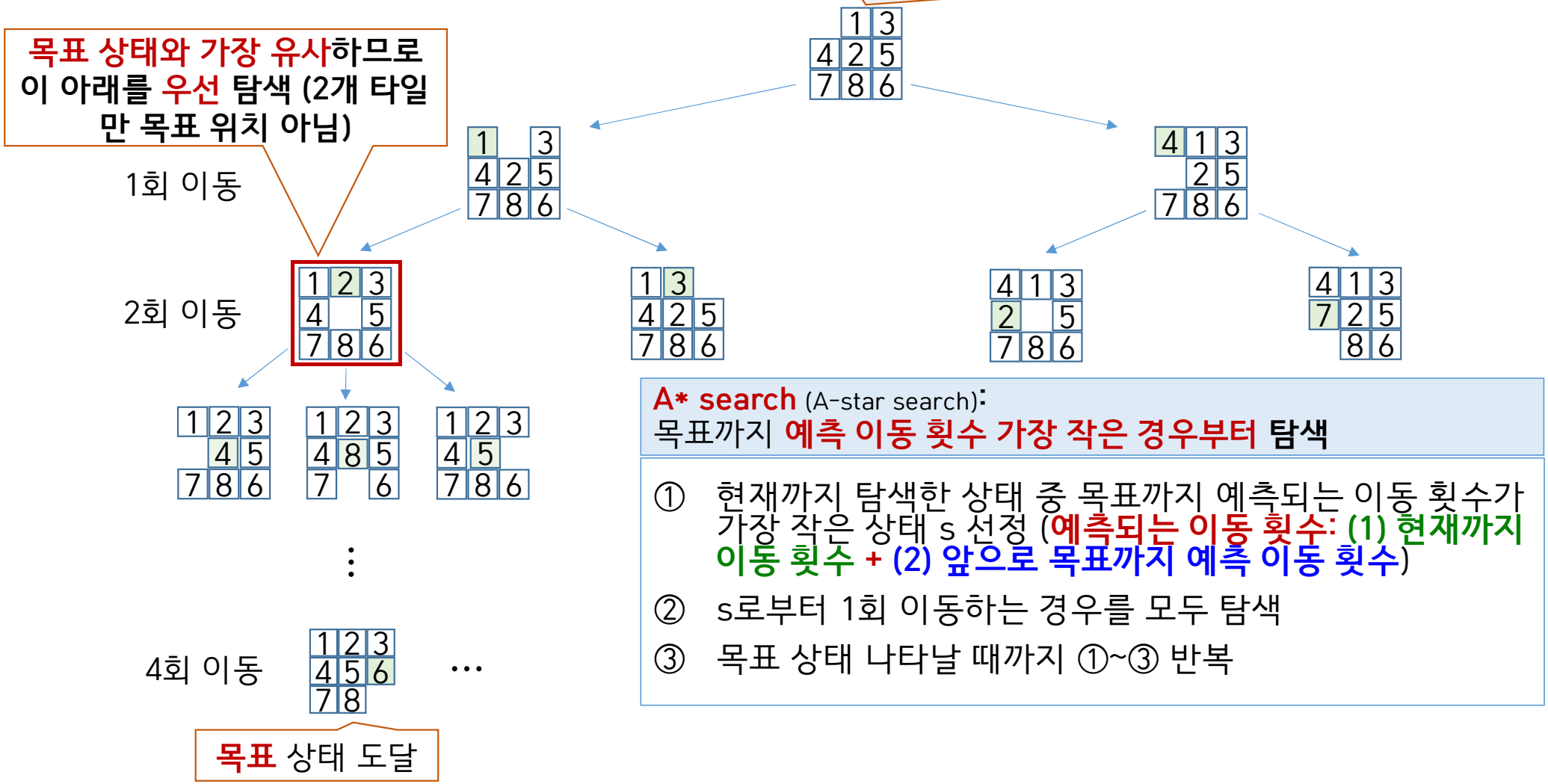
- 처음부터 모든 원소 다 알 수 없으며, 작업 수행 중 새 원소 계속 추가됨 (insert())
  - 처음부터 모든 원소 다 얻을 수 있다면 정렬 사용해 해결 가능
  - iteration 진행하며 계속 새로운 원소 추가되므로, 효율적인 insert() 방법 필요

- A* Search

- 시작점 - 목적지까지의 모든 가능한 경로를 미리 알지 못하며, **탐색을 진행함에 따라 조금씩 새로운 경로를 알게 됨**

- 현재까지 탐색한 경로 중 **목적지와 가장 유사할 것으로 (가까울 것으로) 예측되는 경로를 선정**해 더 자세히 탐색하는 것을 반복

[Q] 예측되는 총 이동 횟수 중 **(1) 현재까지 이동 횟수**는 root로부터의 거리로부터 알 수 있다.
(2) 앞으로 목표까지 이동 횟수는 어떻게 예측할까? (수치 등으로 나타내) 비교할 수 있는 방법이어야 함



A* search (A-star search):
목표까지 **예측 이동 횟수 가장 작은 경우부터 탐색**

① 현재까지 탐색한 상태 중 목표까지 예측되는 이동 횟수가 가장 작은 상태 s 선정 (**예측되는 이동 횟수: (1) 현재까지 이동 횟수 + (2) 앞으로 목표까지 예측 이동 횟수**)

② s로부터 1회 이동하는 경우를 모두 탐색

③ 목표 상태 나타날 때까지 ①~③ 반복

Slider Puzzle에서 (2) 목표 상태까지 이동 횟수 예측에 자주 사용하는 함수: Hamming & Manhattan Distance

- Hamming Distance
- 목표 위치에 **있지 않은** 타일의 **갯수**

- Manhattan Distance
- 각 타일의 **목표 위치까지 거리**를 모두 **합산**한 값
- 목표 위치까지 거리: 가로 거리 + 세로 거리

현재 상태
(이 상태로부터 목표까지 거리 예측)

8	1	3
4		2
7	6	5

타일	목표 위치에 있는지 여부
1	X
2	X
3	O
4	O
5	X
6	X
7	O
8	X

X가 몇개인가

Hamming Distance = **5**

타일	목표 위치까지 거리
1	1
2	2
3	0
4	0
5	2
6	2
7	0
8	3

장애물이 없을 때 몇번만에 갈수있는가.

Manhattan Distance = **10**

1	2	3
4	5	6
7	8	

목표 상태

[Q] 왜 'Manhattan Distance'라 이름 지었나?

https://i.pinimg.com/736x/1c/bf/d4/1cbfd40434708eb9919398510aef40d9--map-of-nyc-ny-map.jpg

Slider Puzzle에서 (2) 목표 상태까지 이동 횟수 예측에 자주 사용하는 함수: Hamming & Manhattan

- Hamming Distance
- 목표 위치에 **있지 않은** 타일의 **갯수**

- Manhattan Distance
- 각 타일의 **목표 위치까지 거리**를 모두 **합산**한 값
- 목표 위치까지 거리: 가로 거리 + 세로 거리

현재 상태
(이 상태로부터 목표까지 거리 예측)

3	2	1
6	5	4
	7	8

목표 상태

1	2	3
4	5	6
7	8	

[Q] Hamming Distance = 6

타일	목표 위치에 있는지 여부
1	X
2	O
3	X
4	X
5	O
6	X
7	X
8	X

타일	목표 위치까지 거리
1	2+0=2
2	0
3	2+0=2
4	2+0=2
5	0
6	2+0=2
7	1+0=1
8	1+0=1

[Q] Manhattan Distance = 10

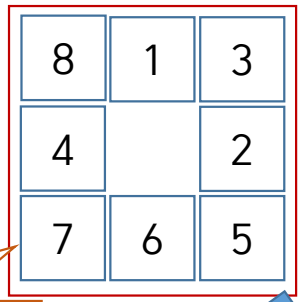
[Q] 둘 중 어느 예측함수가 실제 이동 횟수에 더 근접할 것 같은가 (즉 어느 쪽이 더 정확하게 예측할까)? 이유는?
더 정확하게 예측할수록 더 빠르게 답을 찾는데 도움이 된다.

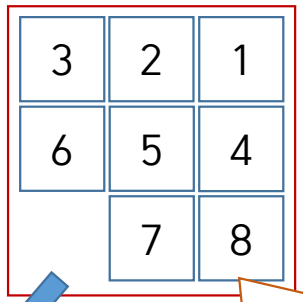
Manhatten

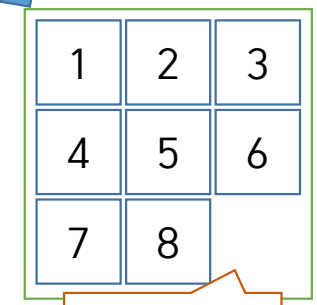
A* search에서 (목표 상태까지 이동 횟수) **예측 함수가 반드시 만족할 조건: 예측 횟수 ≤ 실제 이동횟수** (즉 예측 값이 실제 값의 lower bound)

- Hamming Distance
- 목표 위치에 있지 않은 타일의 갯수

- Manhattan Distance
- 각 타일의 목표 위치까지 거리를 모두 합산한 값
- 목표 위치까지 거리: 가로 거리 + 세로 거리







목표 상태

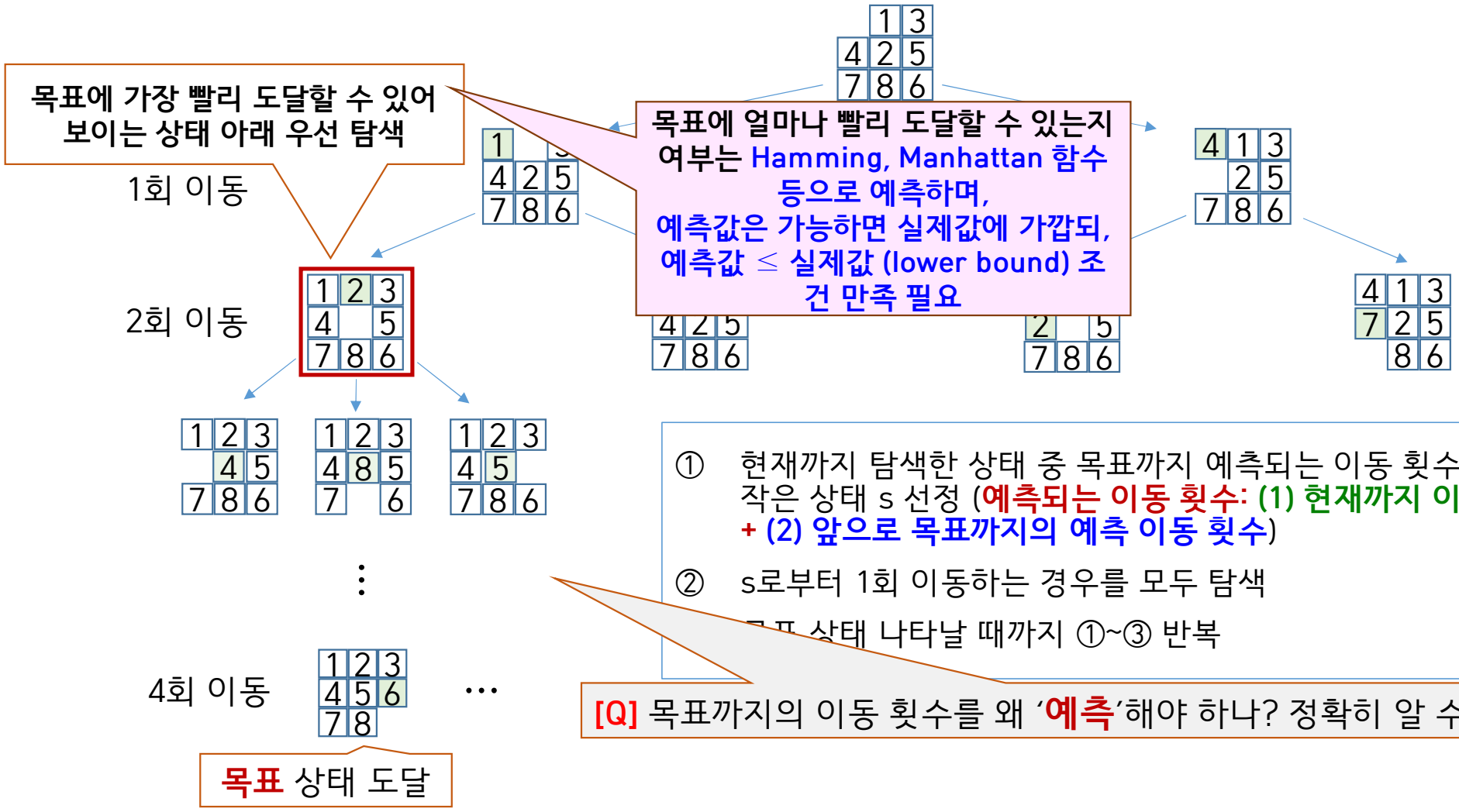
Hamming Distance = **5**
Manhattan Distance = **10**
≤
실제 최소 이동횟수 = **14**

이 조건 만족해야만 **A* search로 찾은 해가** 근사치가 아닌 **최소 이동횟수 해**가 됨

Hamming Distance = **6**
Manhattan Distance = **10**
≤
실제 최소 이동횟수 = **24**

[Q] 두 예측함수가 위 조건 만족하는 이유는?

A* search 정리: 목표 상태까지 가는 해 빠르게 찾기 위해 목표까지 이동 횟수 예측하고, 예측 횟수 가장 작은 경우부터 탐색



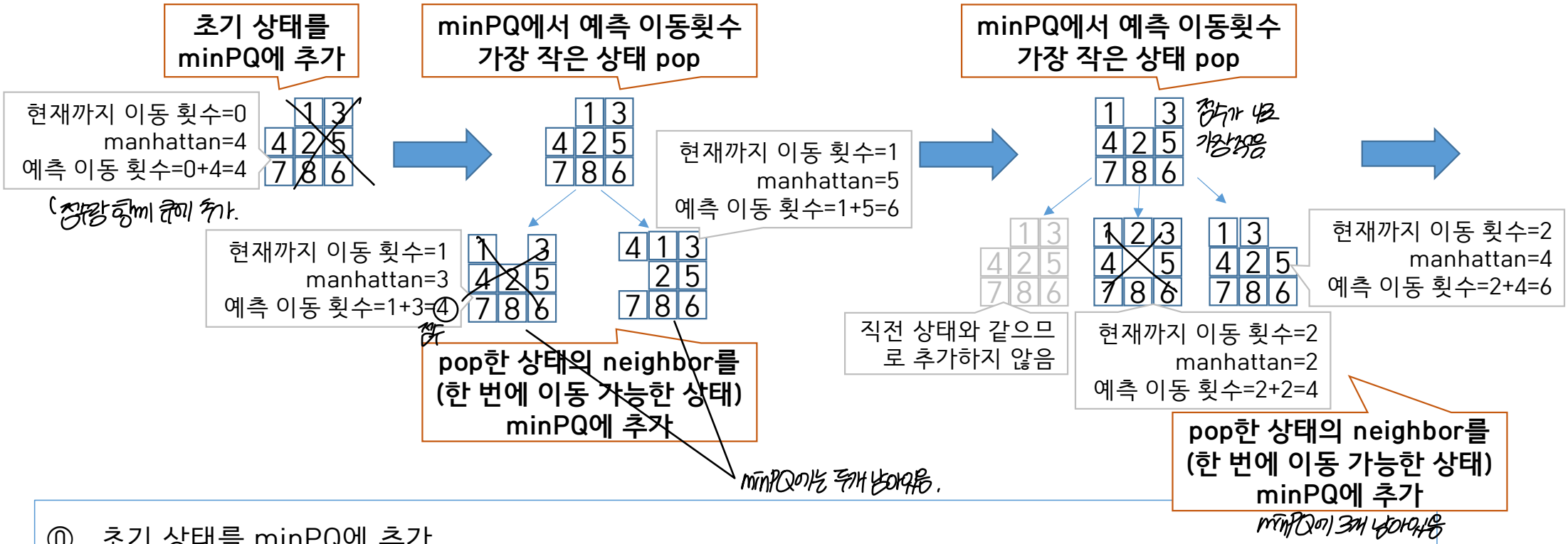
① 현재까지 탐색한 상태 중 목표까지 예측되는 이동 횟수가 가장 작은 상태 s 선정 (**예측되는 이동 횟수: (1) 현재까지 이동 횟수 + (2) 앞으로 목표까지의 예측 이동 횟수**)

② s로부터 1회 이동하는 경우를 모두 탐색

목표 상태 나타날 때까지 ①~③ 반복

[Q] 목표까지의 이동 횟수를 왜 '**예측**'해야 하나? 정확히 알 수는 없나?

A* search 구현 방법: 예측 이동 횟수 가장 작은 경우부터 우선 탐색하기 위해 Min Priority Queue 사용



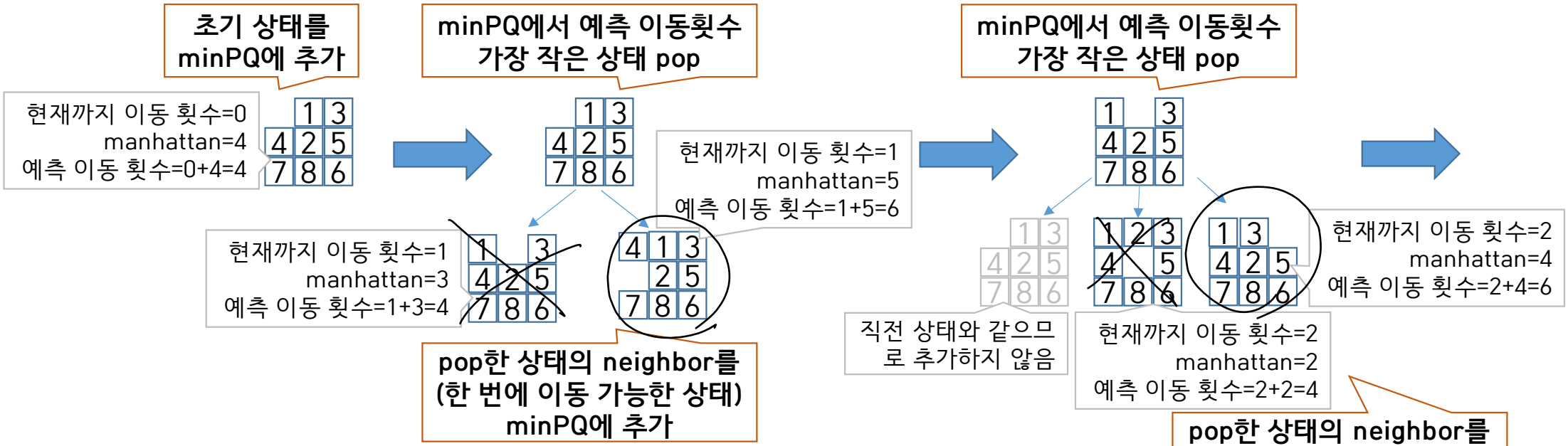
⓪ 초기 상태를 minPQ에 추가

① minPQ에서 총 예측 이동 횟수 (현재까지 이동 횟수 + 앞으로 목표까지 예측 이동 횟수) 최소인 상태 s를 pop

② s로부터 1회 이동하는 경우(**neighbor**)를 모두 minPQ에 추가 (바로 직전 상태로 다시 돌아가는 경우는 제외)

③ s == 목표 상태일 때까지 ①~② 반복

A* search 구현 방법: 예측 이동 횟수 가장 작은 경우부터 우선 탐색하기 위해 Min Priority Queue 사용

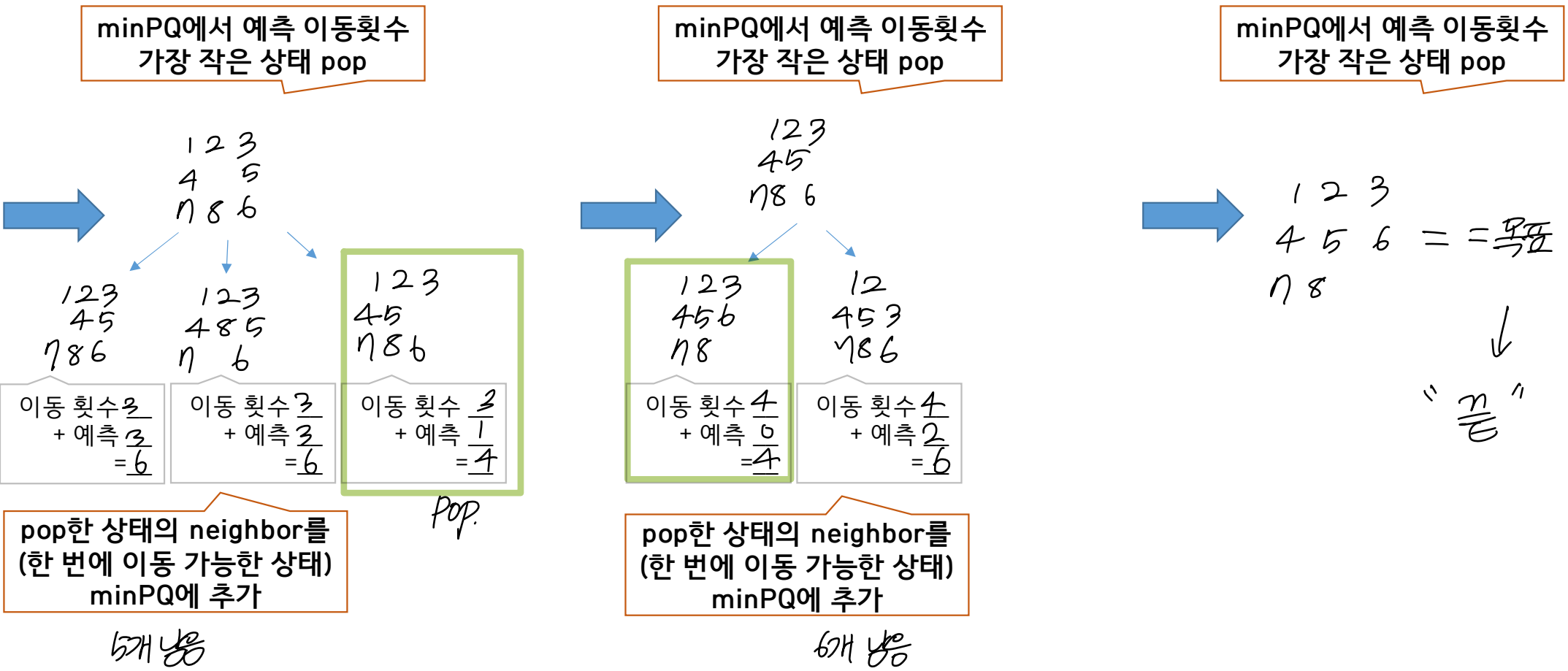


[Q] 이 후에 minPQ에 남아 있는 상태는 몇 개인가? 3개

[Q] 이중 다음에 pop되는 상태는 무엇인가?

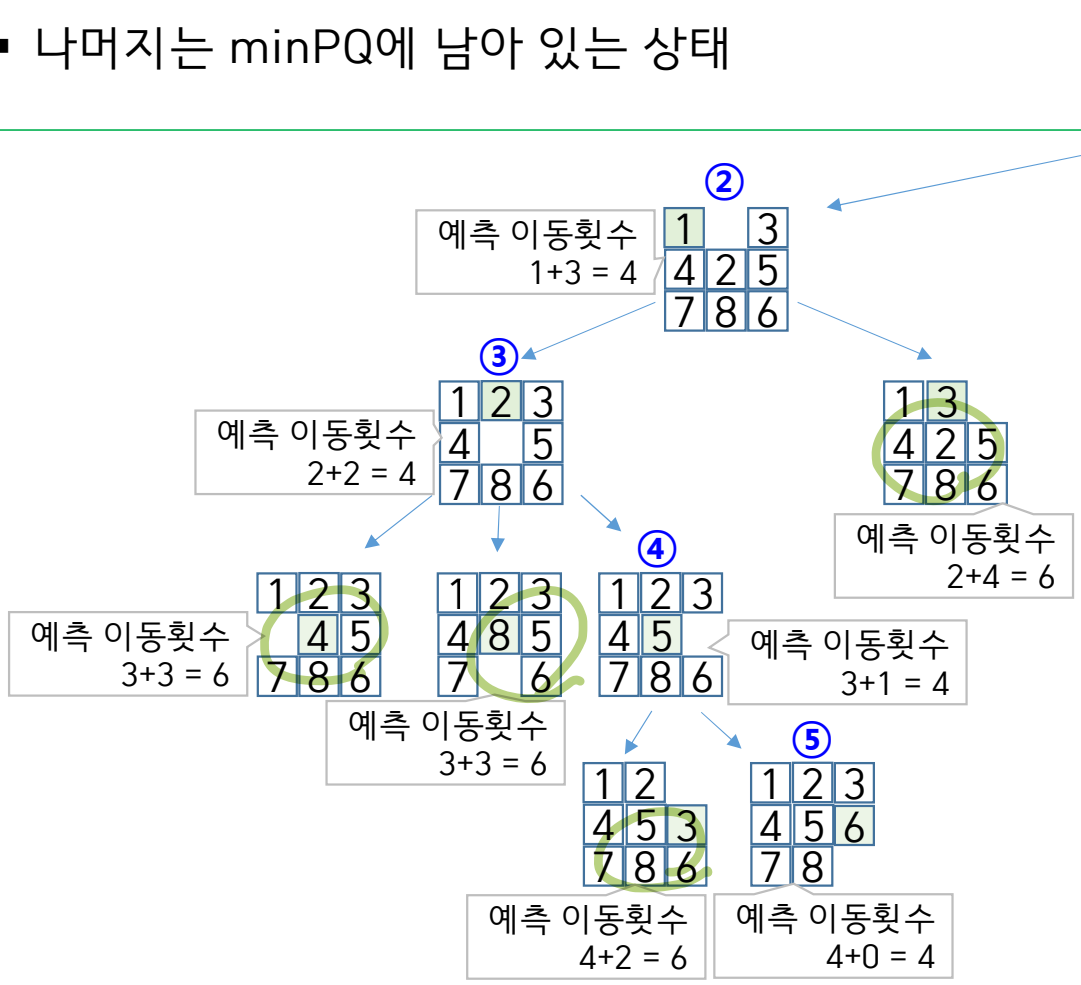
1 2 3
4 5
7 8 6

[Q] 앞 페이지 후에 어떤 상태가 pop 되고, 어떤 neighbor가 추가되는지를 종료 시까지 (pop한 상태 == 목표 상태) 그려 보시오. 직전 상태와 같아 추가하지 않는 상태는 그리지 마시오.



지금까지 A* search로 탐색해온 상태를 Game Tree 형태로 보기

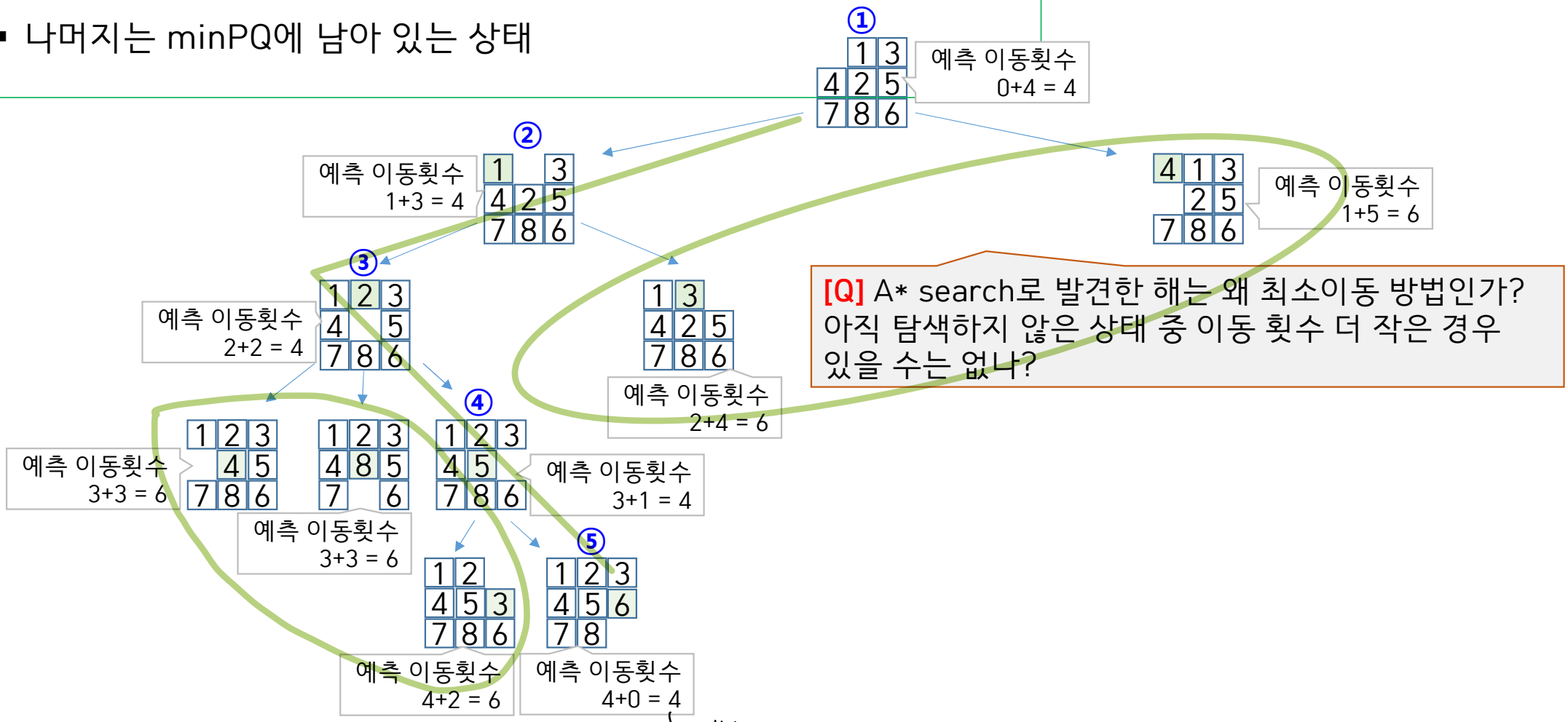
- ①~⑤ 순서로 minPQ에서 pop 되며 neighbor를 minPQ에 추가해감
- 나머지는 minPQ에 남아 있는 상태



[Q] 목표 상태에 도달했을 때 minPQ에 여전히 남아있는 상태를 모두 표시해 보시오. 5개

지금까지 A* search로 탐색해온 상태를 Game Tree 형태로 보기

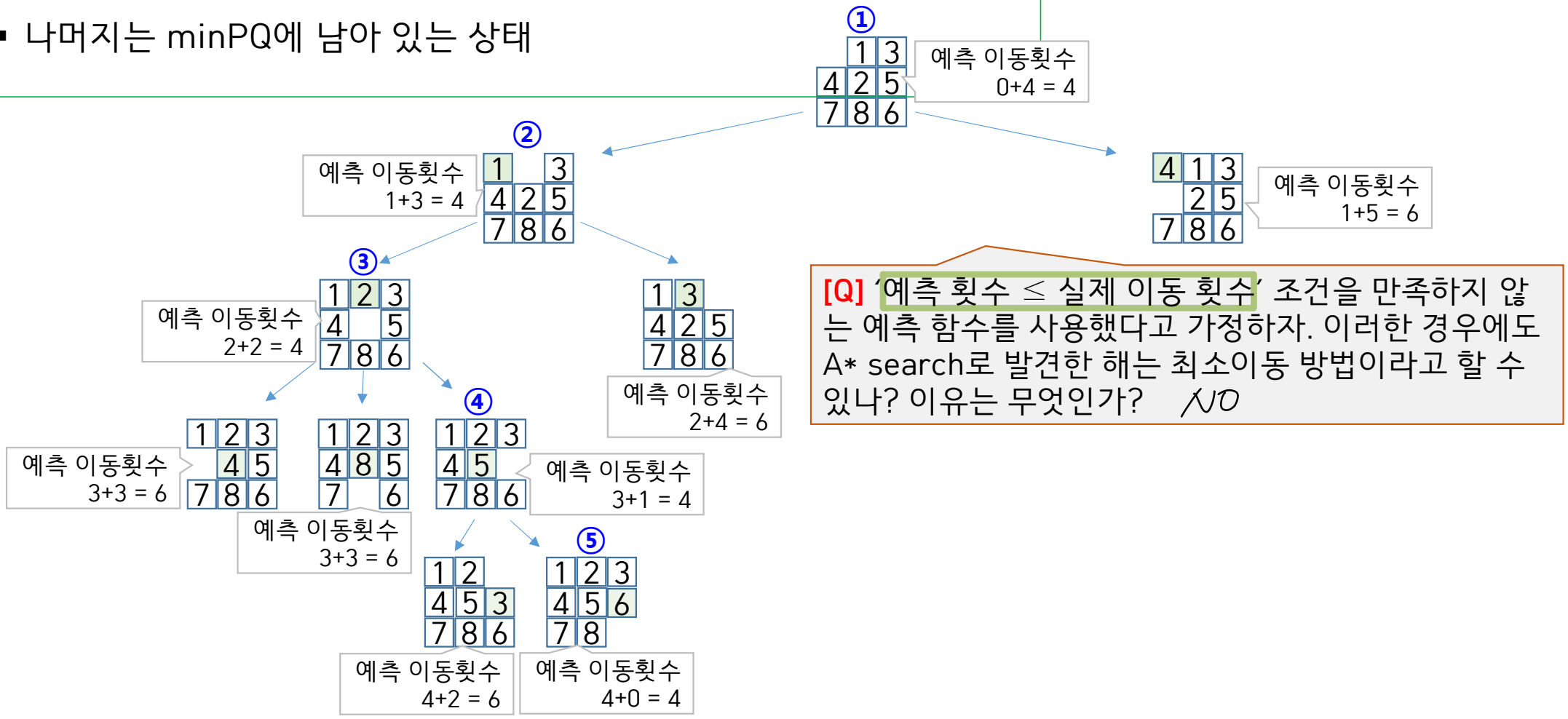
- ①~⑤ 순서로 minPQ에서 pop 되며 neighbor를 minPQ에 추가해감
- 나머지는 minPQ에 남아 있는 상태



[Q] A* search로 발견한 해는 왜 최소이동 방법인가? 아직 탐색하지 않은 상태 중 이동 횟수 더 작은 경우 있을 수는 없나?

지금까지 A* search로 탐색해온 상태를 Game Tree 형태로 보기

- ①~⑤ 순서로 minPQ에서 pop 되며 neighbor를 minPQ에 추가해감
- 나머지는 minPQ에 남아 있는 상태



[Q] '예측 횟수 ≤ 실제 이동 횟수' 조건을 만족하지 않는 예측 함수를 사용했다고 가정하자. 이러한 경우에도 A* search로 발견한 해는 최소이동 방법이라고 할 수 있나? 이유는 무엇인가? NO

조건-① 예측 횟수가 실제 이동 횟수에 가까울수록 (예측이 정확할수록) 불필요한 탐색을 줄일 수 있어 해를 더 빨리 찾을 수 있다. (예: hamming vs. manhattan)

혹은 대소 관계가 실제와 최대한 같게 유지되도록 예측하면 불필요한 탐색 적음

조건-② 예측 횟수 ≤ 실제 이동 횟수 조건을 만족해야만 A* search로 찾은 해가 최소 이동횟수를 보장한다.

- 두 조건이 서로 상반되는 부분 있으므로
- 어플리케이션에 따라 **두 조건을 모두 만족하는 함수 찾기 어려울 수도 있음**

- 그러한 경우
- 반드시 최적의 해(optimal solution) 필요하다면, 예측이 조금 부정확해서 탐색 시간 오래 걸리더라도 **조건-②** 항상 만족하는 예측 함수 사용
- 최적에 가까운 해(sub-optimal solution)라도 괜찮다면, 때론 조건-② 만족하지 않더라도 가능하면 실제 이동 횟수에 더 가까운(**조건-①**) 예측 함수 사용하면 해를 더 빨리 찾을 수 있음

Priority Queue

PQ가 무엇이고, 어떻게 구현하며, 어떤 경우에 어떻게 활용하는지 이해

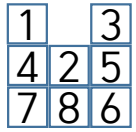
01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Binary Heap 사용한 PQ의 구현
03. Priority Queue 사용 어플리케이션의 공통적인 특성
04. Slider Puzzle and A* Search with PQ
05. 실습: PQ를 사용한 Slider Puzzle Solver 구현

프로그램 구현 조건

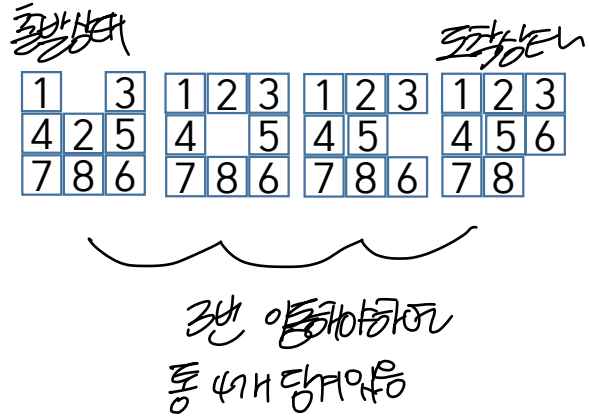
- A* Search with Manhattan distance 수행하는 함수 구현

  def solve**M**anhattan(**initialBoard**):

- 입력 **initialBoard**: Board 객체로 게임의 초기 상태 나타냄



- 반환 값
  - 초기 상태에서 목표 상태까지 거쳐가는 각 상태를 차례로 저장한 목록
  - 초기 상태와 목표 상태도 포함해야 함
  - 각 상태는 Board 객체로 표현

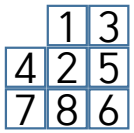
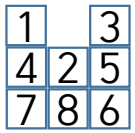
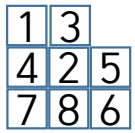


- 이번 시간에 제공한 코드 SliderPuzzle.py에 위 함수 추가해 제출
  - 위 코드에 포함된 Board 클래스는 반드시 사용해야 함
  - Priority Queue로는 위 파일에 이미 import된 PriorityQueue 클래스 사용

프로그램 구현 조건

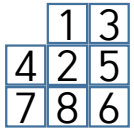
- 최종 결과물로 SliderPuzzle.py 파일 하나만 제출하며, 이 파일만으로 코드가 동작해야 함
- import는 원래 SliderPuzzle.py 파일에서 import하던 3개 패키지 외에는 추가로 할 수 없음 (copy, random, queue.PriorityQueue)
- 각자 테스트에 사용하는 모든 코드는 반드시 if __name__ == "__main__": 아래에 넣어
- 제출한 파일을 import 했을 때는 실행되지 않도록 할 것

구현된 API 정리: Board Class - NxN 보드에서 타일의 배치 나타내는 클래스

```
class Board:
    def __init__(self, tiles): # 보드 객체 생성자. tiles는 list of list로, tiles[row][col]이 (row,col)의 타일 값 나타냄
    def __str__(self): # 보드 객체를 사람이 읽을 수 있는 문자열 형태로 반환. 반환한 문자열은 출력에 사용
    def __eq__(self, other): # self 객체와 other 객체 비교하여 같으면 True 반환하고, 다르면 False 반환
    def __lt__(self, other): # self 객체와 other 객체 비교하여 self 객체가 더 작으면(<) True 반환하고, 아니면 False 반환
    def hamming(self): # self 객체의 hamming distance 반환
    def manhattan(self): # self 객체의 manhattan distance 반환
    def dimension(self): # self 객체가 NxN 보드를 나타낸다면, N을 반환
    def isGoal(self): # self 객체가 목표 상태와 같다면 True 반환하고, 다르면 False 반환
    def neighbors(self): # self 객체의 모든 neighbor 보드를 list에 담아 반환
    def twin(self): # self 객체에서 임의의 두 타일을 교환한 보드를 반환
```

```
b10 = Board([[0,1,3],[4,2,5],[7,8,6]])
print(b10.manhattan())
```



구현된 API 정리: Board Class - NxN 보드에서 타일의 배치 나타내는 클래스

```
class Board:
    def __init__(self, tiles): # 보드 객체 생성자. tiles는 list of list로, tiles[row][col]이 (row,col)의 타일 값 나타냄
        # a = Board([[1,2,3],[4,5,6],[7,0,8]])과 같이 Board 클래스 객체를 생성할 때 호출됨
        # tiles는 NxN 보드를 나타내며, 0 ~ N²-1 사이의 정수 N²개 포함. 0은 빈 타일을 나타냄
        # 2 ≤ N ≤ 128 (N이 큰 경우 답을 빨리 찾을 필요는 없고, minPQ 사용해 요구 조건에 맞게 탐색하면 됨)

    def __str__(self): # 보드 객체를 사람이 읽을 수 있는 문자열 형태로 반환. 반환한 문자열은 출력에 사용
        # a가 보드 객체라면 print(a) 했을 때, print(a.__str__())로 변환되어 실행됨
        # NxN 보드라면 한 줄에 N개의 원소씩 N줄을 포함하는 문자열 반환
        # 아래 왼쪽 보드를 예로 들면 반환하는 문자열은 아래 오른쪽과 같음. 각 숫자는 총 6칸을 차지하도록 함
                  1 3          0    1    3
              4 2 5            4    2    5
              7 8 6            7    8    6

    def __eq__(self, other): # self 객체와 other 객체 비교하여 같으면 True 반환하고, 다르면 False 반환
        # '=='을 사용해 두 보드 객체를 비교할 때 호출되는 함수
        # 'a == b'라 썼다면 a.__eq__(b)로 변환되어 실행되며, a가 self가 되고 b가 other가 됨
        # Slider Puzzle 어플리케이션에서는 현재 검사하는 보드 a가 목표 보드 b와 같은지 확인할 때 사용
        #
        # 아래와 같은 3개 라인으로 기본 입력 검사 후,
        # NxN개 값이 모두 같은 위치에 있으면 True를, 하나라도 다른 위치에 있으면 False 반환
        if other == None: return False
        if not isinstance(other, Board): return False
        if self.n != other.n: return False

    …
```

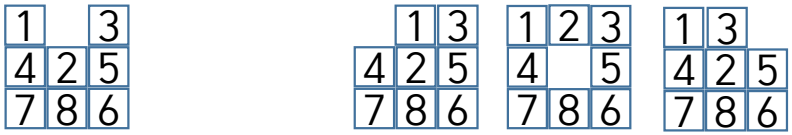
__로 시작해 __로 끝나는 함수는 모든 클래스에서 공통적으로 유사한 역할을 하는 함수로 magic function이라 함
(예: __init__, __str__, __eq__, __lt__ 등)

구현된 API 정리: Board Class - NxN 보드에서 타일의 배치 나타내는 클래스

```
class Board:
    …

    def __lt__(self, other): # self 객체와 other 객체 비교하여 self 객체가 더 작으면(<) True 반환하고, 아니면 False 반환
        # ‘<’를 사용해 두 보드 객체를 비교할 때 호출되는 함수. ‘less than’을 의미
        # ‘a < b’라 썼다면 a.__lt__(b)로 변환되어 실행되며, a가 self가 되고 b가 other가 됨
        # Slider Puzzle 어플리케이션에서는 Priority Queue에 담긴 두 보드의 대소 비교 시 사용
        #
        # 아래와 같이 구현하되, 다른 방식으로 대소 비교해도 어플리케이션 동작에는 큰 차이 없음
        if self.n < other.n: return True
        else:
            for rowId, row in enumerate(self.tiles):
                for colId, t in enumerate(row):
                    if t < other.tiles[rowId][colId]: return True
            return False

    def hamming(self): # self 객체의 hamming distance 반환
        # hamming distance는 여러 차례 필요할 수 있으므로
        # 생성자에서 미리 이 값을 계산해 클래스 변수에 저장해둔 후 (예: self.hammingDistance)
        # 이 함수에서는 저장해둔 값을 반환만 함

    def manhattan(self): # self 객체의 manhattan distance 반환
        # manhattan distance는 여러 차례 필요할 수 있으므로
        # 생성자에서 미리 이 값을 계산해 클래스 변수에 저장해둔 후 (예: self.manhattanDistance)
        # 이 함수에서는 저장해둔 값을 반환만 함

    …
```

구현된 API 정리: Board Class - NxN 보드에서 타일의 배치 나타내는 클래스

```
class Board:
    …

    def dimension(self): # self 객체가 NxN 보드를 나타낸다면, N을 반환

    def isGoal(self): # self 객체가 목표 상태와 같다면 True 반환하고, 다르면 False 반환

    def neighbors(self): # self 객체의 모든 neighbor 보드를 list에 담아 반환
        # 현재 상태에서 다음으로 이동 가능한 상태를 찾기 위해 호출하는 함수
        # neighbor는 한 타일을 빈 칸으로 이동(1회 이동)해서 갈 수 있는 상태를 의미
        # 빈 타일의 위치에 따라 2~4개의 neighbor가 있을 수 있음. 아래 왼쪽 예제의 경우 오른쪽과 같은 3개의 neighbor 있음
```



```
        #      self                    neighbors

    def twin(self): # self 객체에서 임의의 두 타일을 교환한 보드를 반환
        # 해가 없는 퍼즐도 있으며, 이를 탐지하기 위해 twin(임의의 두 타일을 교환한 보드)이 필요함
        # self와 twin 중 항상 하나만 해를 찾을 수 있고, 다른 하나는 해를 찾을 수 없음이 증명됨
        # 따라서 self와 twin을 병렬로 함께 풀어가면서 self의 해를 찾았다면 그 해를 반환하면 되고
        # twin의 해를 찾았다면 self의 해는 없음(None)을 반환하면 됨

        # 오른쪽은 self의 1과 3 타일을 교환해 twin을 만든 예임
        # 어떤 두 타일을 교환해 twin을 만들어도 상관없으나
        # twin()을 여러 차례 호출했을 때 항상 같은 보드가 반환되면 편리
        #
```

twin() 함수는 이번 과제에는 필요하지 않음. Solve 가능한 입력만 주어짐

self(해 있음)　　twin(해 없음)

구현된 API 정리: Board Class에 속하지 않은 함수 - 결과 출력에 사용

```
def solveNprint(initialBoard):    # solveFunction() 함수 사용해 initialBoard 해 찾고 이를 출력
    # initialBoard는 Board 클래스 객체로 퍼즐의 초기 상태를 나타냄
    # 아래와 같이 구현
    solution = solveManhattan(initialBoard)
        if solution != None:
            print(f"Solvable in {len(solution)-1} moves")
            for board in solution:
                print(board)
        else: print("Unsolvable")
```

구현할 API 정리: solveManhattan()

```
def solveManhattan(initialBoard):   # Manhattan distance를 예측 함수로 사용해 initialBoard의 해 찾아 반환
    # queue.PriorityQueue 클래스 사용해 minPQ 객체 생성
    #
    # minPQ에 담는 원소는 게임 트리에서 하나의 상태를 나타내며, Board 객체에 추가 정보를 담은 아래와 같은 4-tuple 임
    # (현재까지 이동 횟수 + Manhattan 거리, Board 객체, 현재까지 이동 횟수, 직전 상태 나타내는 4-tuple)
    #
    # 맨 처음에는 minPQ에 아래와 같은 초기 상태 담을 것
    # (0 + initialBoard.manhattan(), initialBoard, 0, None)
    #
    # 그 후에는 while loop을 사용해 다음을 반복
    #     minPQ에서 가장 작은 원소를 get(). 이 원소는 4-tuple일 것이며, minNode라 하겠음
    #     If minNode가 목표 상태와 같다면:
    #         초기 상태부터 목표 상태까지 거쳐가는 모든 Board 객체를 리스트에 담아 반환
    #     else:
    #         minNode의 각 neighbor를 아래와 같이 minPQ에 추가 (직전 상태와 같은 neighbor는 제외)
    #         (minNode의 이동횟수 + 1 + neighbor. manhattan(), neighbor, minNode의 이동횟수 + 1, minNode)
```

Python 기본 제공 PQ 클래스: minPQ이나, tuple로 key 지정해 다른 우선순위 지정 가능

MinPQ 제공 함수	설명
PriorityQueue()	Constructor(생성자). MinPQ 객체 생성
put(k)	원소 k를 PQ에 추가
get()	가장 작은 원소를 제거하며 반환
qsize()	PQ에 저장된 원소의 수를 반환

```
from queue import PriorityQueue
pq1 = PriorityQueue()
pq1.put(('A',100))
pq1.put(('B',20))
pq1.put(('C',150))
while pq.qsize() > 0:
    print(pq.get())
================================
('A',100)
('B',20)
('C',150)
```

알파벳을 key로 minPQ

```
from queue import PriorityQueue
pq1 = PriorityQueue()
pq1.put((100,'A'))
pq1.put((20,'B'))
pq1.put((150,'C'))
while pq.qsize() > 0:
    print(pq.get())
================================
(20,'B')
(100,'A')
(150,'C')
```

숫자를 key로 minPQ

```
from queue import PriorityQueue
pq1 = PriorityQueue()
pq1.put((-100,'A'))
pq1.put((-20,'B'))
pq1.put((-150,'C'))
while pq.qsize() > 0:
    a = pq.get()
    print((a[1], -a[0]))
================================
('C',150)
('A',100)
('B',20)
```

숫자를 key로 maxPQ

프로그램 입출력 예:
그 외 테스트 케이스는 SliderPuzzle.py “__main__” 아래에 있음

```
>>> solveNprint(Board([[0,1,3],[4,2,5],[7,8,6]]))
Solvable in 4 moves
     0     1     3
     4     2     5
     7     8     6

     1     0     3
     4     2     5
     7     8     6

     1     2     3
     4     0     5
     7     8     6

     1     2     3
     4     5     0
     7     8     6

     1     2     3
     4     5     6
     7     8     0
```

프로그램 입출력 예:
그 외 테스트 케이스는 SliderPuzzle.py “__main__” 아래에 있음

```
>>> solveNprint(Board([[8,1,3],[4,0,2],[7,6,5]]))
Solvable in 14 moves
     8     1     3
     4     0     2
     7     6     5

     8     1     3
     4     2     0
     7     6     5

     8     1     3
     4     2     5
     7     6     0

     8     1     3
     4     2     5
     7     0     6

     8     1     3
     4     2     5
     0     7     6

     8     1     3
     0     2     5
     4     7     6
```

```
     0     1     3
     8     2     5
     4     7     6

     1     0     3
     8     2     5
     4     7     6

     1     2     3
     8     0     5
     4     7     6

     1     2     3
     0     8     5
     4     7     6

     1     2     3
     4     8     5
     0     7     6

     1     2     3
     4     8     5
     7     0     6
```

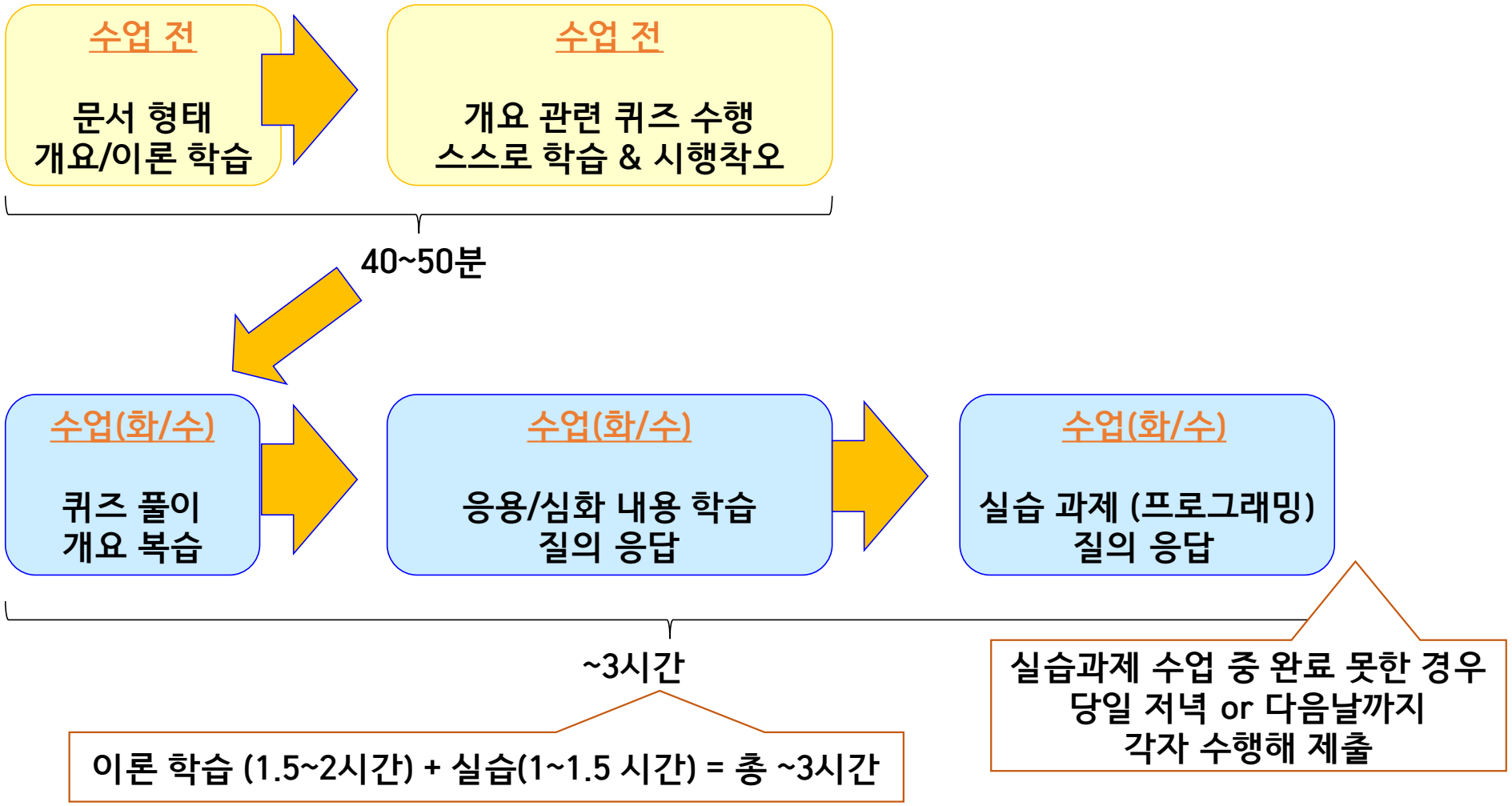
```
     1     2     3
     4     0     5
     7     8     6

     1     2     3
     4     5     0
     7     8     6

     1     2     3
     4     5     6
     7     8     0
```

그 외 동작 검증 방법

- 코드를 잘 작성했다면
- 작성한 코드에서 manhattan()을 호출하는 부분을 hamming()으로 변경해 실행해도
- 최종 결과는 같을 것임

- 하지만 실행 속도는
- (특히 더 많은 move가 필요한 경우일수록) Manhattan distance를 사용하는 편이 더 빠름을 볼 수 있음

스마트 출결



12:00까지 실습 & 질의응답

- 작성한 코드는 lms > 강의 콘텐츠 > 오늘 수업 > 실습 과제 제출함에 제출

- 시간 내 제출 못한 경우 내일 11:59pm까지 제출 마감
- 마감 시간 후에는 제출 불가하므로 그때까지 작성한 코드 꼭 제출하세요.

정리: Priority Queue 무엇이고, 어떻게 활용하며, 어떤 경우 활용하는지 이해

- Priority Queue: 우선순위가 높은 원소부터 꺼내 사용하는 Queue
- insert(), delete() 모두 ~logN 시간에 효율적으로 구현하기 위해 Binary Heap 구조 사용 (**Heap-order** 따라 원소 저장하는 **Complete Binary Tree**)
  - Tree 구조 ➔ 효율적인 insert(), delete() 가능하게 함
  - (Max) heap order: 부모 key ≥ 자식 key ➔ 효율적인 delete() 가능하게 함
  - Complete binary tree: 왼쪽 노드부터 빠짐없이 원소를 채운 이진 트리 ➔ 배열에 저장해 편리하게 indexing 가능하게 함

- Priority Queue 사용 어플리케이션의 특성:
- 처음부터 모든 정보 안다면 **정렬** 사용해도 해결 가능한 문제이나
- **문제 해결 중 새로운 정보 계속 추가**되며
- 그 중 **우선순위 가장 높은 것 선정해 처리**하는 방식으로 진행
- 예: TopK 찾기, Dijkstra의 최단경로 찾기, Event-driven Simulation, A* Search, …