



Symbol Table

Symbol Table이 무엇이고, 어떻게 구현하며, 어떤 경우에 어떻게 활용하는지 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. 2-3 Tree 활용한 Symbol Table 구현 (기본 BST보다 더 효율적인 Symbol Table 구현 알아보기)
03. BST 활용한 2-3 Tree 구현 (LLRB, Left-Leaning Red Black BST)
04. 1-D Range Search (symbol table 기능 추가)
05. Line-segment Intersection (symbol table & priority queue 활용 예)
06. 실습: Symbol Table 기능 추가 & 활용

Symbol Table (Dictionary, Map): (key, value) 쌍 저장, 다양한 기능 수행

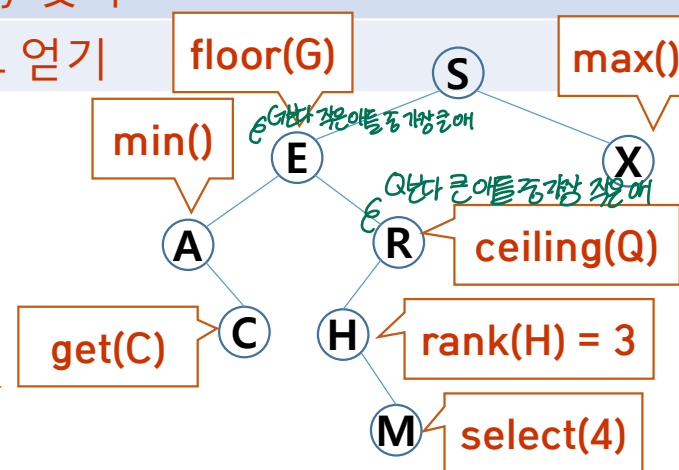
작업	기능
put(key, value)	새로운 (key, value) 쌍 저장 or 기존에 저장된 value 수정
delete(key)	기존에 저장된 (key, value) 쌍 삭제
get(key)	Key를 검색어로 주고, 저장된 value 찾기
min(), max()	최소, 최대 key 찾기
floor(key), ceiling(key)	key 바로 전 or 바로 후의 값 찾기
rank(key)	key보다 작은 원소 수 찾기 (정렬했을 때 key의 위치 찾기)
select(idx)	(정렬했을 때) idx 번째 key 찾기
ordered iteration	모든 key를 정렬된 순서로 얻기

추가/삭제

저장된 값에 대한
다양한 **탐색** 기능
(DB로 볼 수 있음)

대부분 정렬된 순서와
관련 있으므로 **ordered
operations**라 함

오늘은 이 외에도 좌표 계산에 유
용한 추가 작업 볼 것임 (range
search, nearest neighbor 등)



Copyright © by Sihyung Lee - All rights reserved.

정렬시.

A	C	E	H	M	R	S	X
0	1	2	3	4	5	6	7



```
CREATE TABLE 'STUDENT' ('StudentID' VARCHAR(20) NOT NULL,  
                           'Name' VARCHAR(50) NOT NULL,  
                           'PhoneNumber' VARCHAR(10) NOT NULL,  
                           'Address' VARCHAR(100) NOT NULL,  
                           PRIMARY KEY('StudentID'));
```



Field	Type	Key?
StudentID	VARCHAR(20)	Yes
Name	VARCHAR(50)	No
PhoneNumber	VARCHAR(10)	No
Address	VARCHAR(100)	No

key

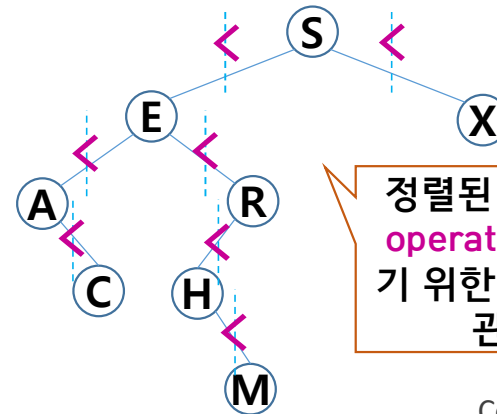
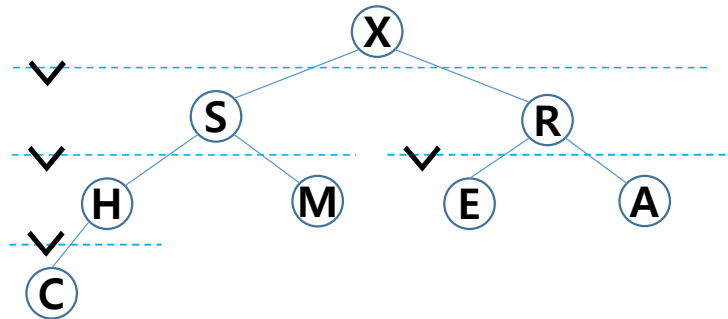
: ID → 유니크해야 함.

value

: 찾아야 할 애.



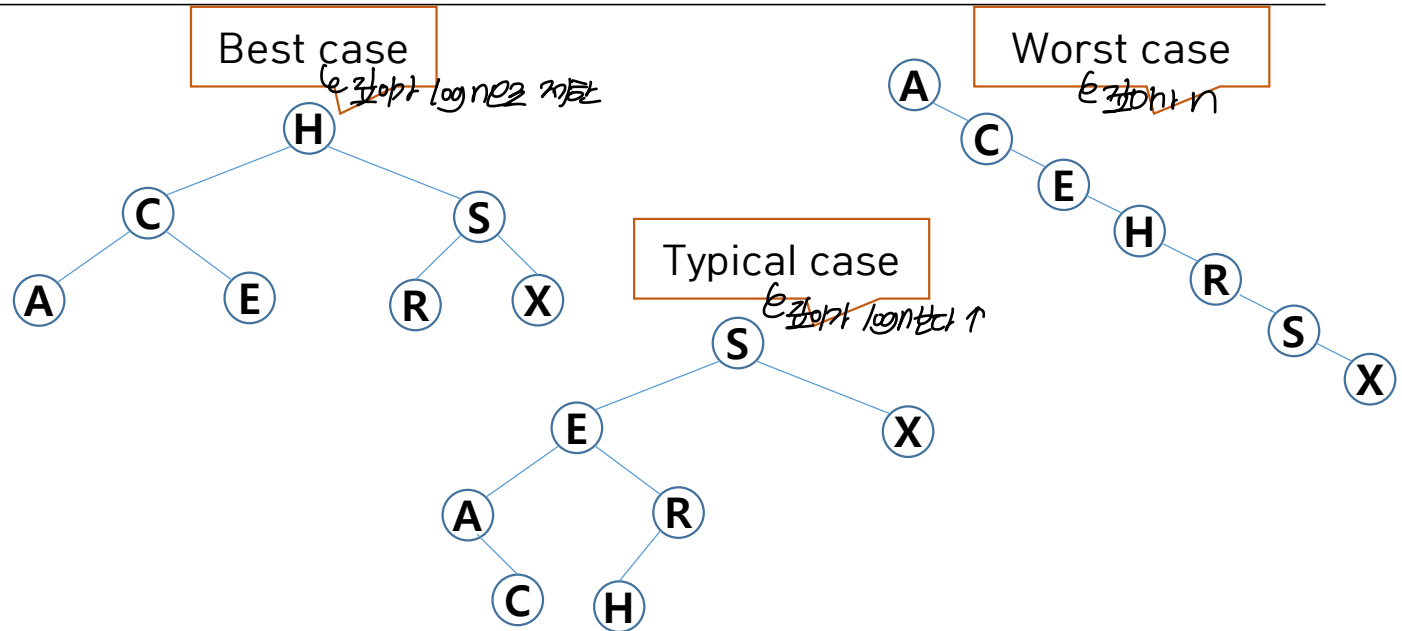
ID	분류	(Max) Heap	BST(Binary Search Tree)	탐색이(search) 주요 기능임 의미
1	목표	효율적인 Priority Queue 구현	효율적인 Symbol Table 구현	
2	기본	이진 트리 (0~2개 자식 가능)		
3	조건1	Complete 트리: 가장 아래 레벨을 제외하고는 빈 곳 없이 가득 차 있으며, 가장 아래 레벨은 왼쪽부터 차례로 채움	왼쪽, 오른쪽 중 어느 쪽이라도 먼저 자식이 추가될 수 있음	
4	구현	(Complete tree이므로) 배열로 보다 간편하게 구현 가능	배열로 구현 어려우며, 자식/부모와 링크 포함해 실제 트리 구조 구현 필요	더 다양한 작업 요구하므로, 배열로 간단하게 구현할 수 있는 방법은 아직 찾지 못함
5	조건2	(Max) Heap order: 자식 key \leq 부모 key 따라서 PQ에 필요한 최댓값 탐색/추출 편리	Symmetric order: 왼쪽 key < 부모 key < 오른쪽 key 따라서 symbol table에 필요한 임의의 key 탐색에 효율적	



정렬된 순서와 관련된 **ordered operations** 효율적으로 수행하기 위한 구조이므로 이러한 대소 관계 따라 원소 배치



작업	BST 성능
get(k)	$\sim \text{depth}$
put(k,v)	$\sim \text{depth}$
delete(k)	$\sim \text{depth}$
min(), max()	$\sim \text{depth}$
floor(k), ceiling(k)	$\sim \text{depth}$
rank(k)	$\sim \text{depth}$
select(i)	$\sim \text{depth}$
ordered iteration	$\sim N$



- BST의 **성능**은 트리의 **깊이에 비례 ($\sim \text{depth}$)**
- 원소 추가/삭제 순서에 따라 깊이는 $\log N$ 에서 N 까지 갈 수 있으며
- 많은 경우 원소 추가/삭제 반복하다 보면 **\sqrt{N} 에 가까워져 성능 떨어짐** 확인됨



Symbol Table의 다른 구현 방식: Hash Table

get('a') → $h('a') = 97 \% 19 = 2$

get('u') → $h('u') = 117 \% 19 = 3$

Index	저장된 (key, value)
0	('9', "911")
1	('M', "Marvel")
2	('a', "astounding")
3	('u', "uniform")
4	('c', "crown")
5	('Q', "Quebec")
...	...

hash 함수 $h(\text{key})$ 계산한 값을 index로 사용해
hash table의 저장할 위치에 바로 접근
따라서 get(key), put(key, value) 모두 상수 시간(~ 1)에 수행 가능



Hash Table: key와 정확히 같은 값 탐색에 유리 key에 근접한 값 탐색이나 원소 순서 파악에는 불리

Hash Table: 유사한 원소끼리 항상 근처에 있지는 않음

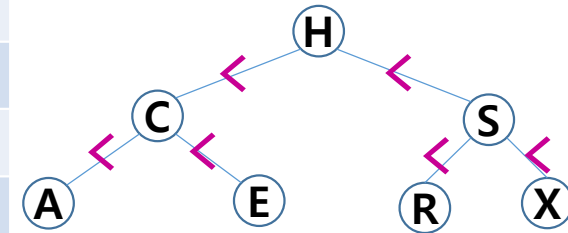
Index	저장된 (key, value)
0	('9', "911")
1	('M', "Marvel")
2	('a', "astounding")
3	('u', "uniform")
4	('c', "crown")
5	('Q', "Quebec")
...	...

작업	Hash Table	BST
get(k)	~ 1	$\sim \text{depth}$
put(k,v)	~ 1	$\sim \text{depth}$
delete(k)	~ 1	$\sim \text{depth}$
min(), max()	$\sim N$	$\sim \text{depth}$
floor(k), ceiling(k)	$\sim N$	$\sim \text{depth}$
rank(k)	$\sim N$	$\sim \text{depth}$
select(i)	$\sim N$	$\sim \text{depth}$
ordered iteration	$\sim N \log N$	$\sim N$
rangeSearch(a,b)	$\sim N$	$\sim \text{depth} + R$
rangeCount(a,b)	$\sim N$	$\sim \text{depth}$

a~b 사이 원소 모두 찾기

R: range 속한 원소 수

BST: 유사한 원소끼리 근처에 있음



근접한 값 탐색은 자주 활용됨

- range search 사용하는 SQL 구문 예:
- `select name from employee where salary >= 1000 and salary <= 2000`
- `select count(*) from employee where salary >= 1000 and salary <= 2000`

- 근사값 찾는 SQL 구문 예:
- `select floor(25.75) from score`
- `select ceiling(25.75) from score`



언어별 Symbol Table 구현 방식

- Java: TreeMap (BST 기반), HashMap (Hash Table 기반)
 - C++: `std::map` (BST 기반), `std::unordered_map` (Hash Table 기반)
 - Python: dictionary (Hash Table 기반), BST 기반 symbol table은 3-rd party library에 있음
 - ...
-
- BST 기반은 range search 기능 지원하나, Hash 기반은 그렇지 않으므로
 - 필요에 따라 적절한 symbol table 사용 필요



Symbol Table

Symbol Table이 무엇이고, 어떻게 구현하며, 어떤 경우에 어떻게 활용하는지 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. 2-3 Tree 활용한 Symbol Table 구현 (기본 BST보다 더 효율적인 Symbol Table 구현 알아보기)
03. BST 활용한 2-3 Tree 구현 (LLRB, Left-Leaning Red Black BST)
04. 1-D Range Search (symbol table 기능 추가)
05. Line-segment Intersection (symbol table & priority queue 활용 예)
06. 실습: Symbol Table 기능 추가 & 활용


트리 깊이 제한 위해 union find 같은 trick 사용하기는 어려움. 새 key가 트리 어디에 부착되지는 key 값에 따라 정해지므로. 그래서 다른 방식으로 깊이 제어 필요

2-3 Tree: 각 노드에 2개까지의 키 저장 & 3개까지의 자식 허용 → 트리 깊이 최소화

- ^{2-3 tree}
 ■ **BST** 사용해 구현 가능 (**BST의 변형 or 확장**)
 - x 왼쪽 자식 $key < \text{노드 } x \text{의 } key < x$ 오른쪽 자식 key 조건 만족
 - BST에서 수행 가능한 여러 탐색 작업 마찬가지로 수행 가능

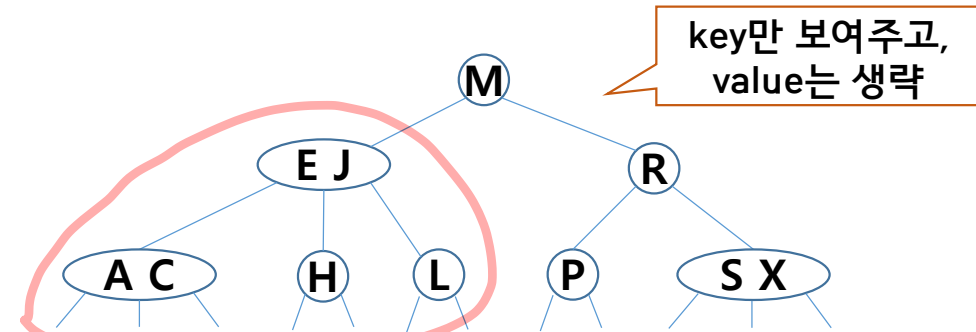
① root ~ leaf 까지 모든 경로가 같은 길이 가짐 → **균형 잡힌 트리**(트리 깊이 최소화)

② 이를 위해 각 node에 1~2개의 key를 저장

- 2-node: 1개 key, 2개의 자식 가진 노드
 
- 3-node: 2개의 key, 3개의 자식 가진 노드
 왼쪽 자식 < k_1 < 가운데 자식 < k_2 < 오른쪽 자식



^{key 2개 (3개 순으로 많음)}
^{자식 3개 有 (")}

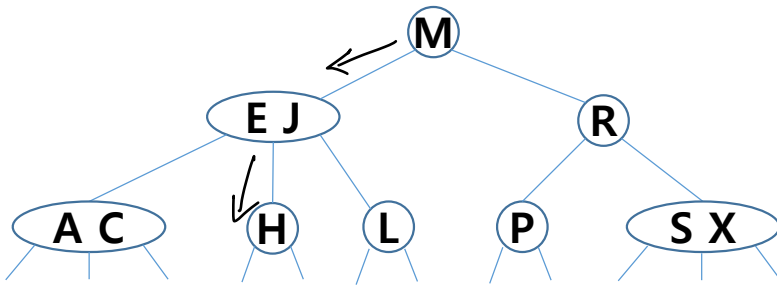


[Q] 위 2-3 Tree가 조건 ①~②를 모두 만족함을 확인 하시오.

2-3 Tree에서의 ^{탐색} **get(key): 3-node에서 3 방향 중 하나로 이동**. 2-node는 BST와 같음

■ **2-node** with k : BST와 같음

- ① $key < k$ 면 왼쪽 자식으로 이동
- ② $key == k$ 면 현재 노드 value 반환
- ③ $k < key$ 면 오른쪽 자식으로 이동

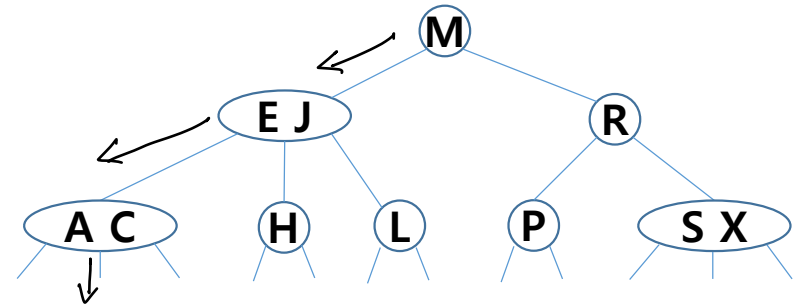
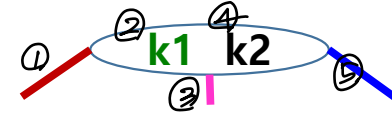


[Q] 위 규칙에 따르면 get(**H**) 하는 과정에서 어떤 노드 따라 내려가는가?

■ **3-node** with $k_1 < k_2$

경의 두가
대각으로 놓음

- ① $key < k_1$ 면 왼쪽 자식으로 이동
- ② $key == k_1$ 면 k_1 의 value 반환
- ③ $k_1 < key < k_2$ 면 가운데 자식으로 이동
- ④ $key == k_2$ 면 k_2 의 value 반환
- ⑤ $k_2 < key$ 면 오른쪽 자식으로 이동



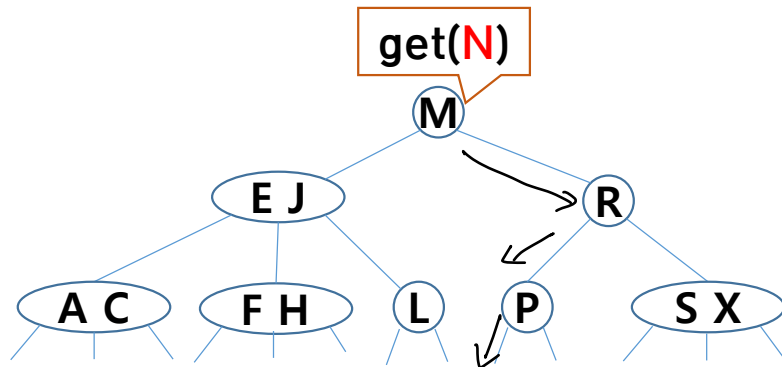
[Q] 위 규칙에 따르면 get(**B**) 하는 과정에서 어떤 노드를 따라가는가?

저장되지 않아대기

2-3 Tree에서의 **get(key): 3-node에서 3 방향 중 하나로 이동**. 2-node는 BST와 같음

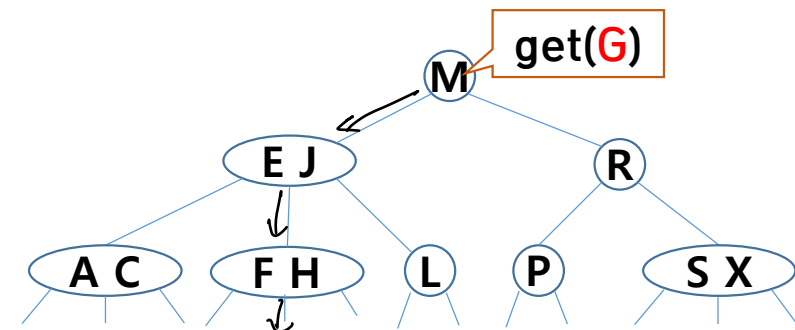
■ 2-node with k: BST와 같음

- ① $key < k$ 면 왼쪽 자식으로 이동
- ② $key == k$ 면 현재 노드 value 반환
- ③ $k < key$ 면 오른쪽 자식으로 이동



■ 3-node with $k_1 < k_2$

- ① $key < k_1$ 면 왼쪽 자식으로 이동
- ② $key == k_1$ 면 k_1 의 value 반환
- ③ $k_1 < key < k_2$ 면 가운데 자식으로 이동
- ④ $key == k_2$ 면 k_2 의 value 반환
- ⑤ $k_2 < key$ 면 오른쪽 자식으로 이동



[Q] get(N) 하는 과정과 get(G) 하는 과정을 비교해 보자.
어느 쪽이 더 대소 비교가 많이 필요한가? 그 차이는 상수 차이인가?

Sihyung Lee - All rights reserved.

탐색 시간 $\log n$ 으로, 비슷하다.

탐색 과정을 통해 추가할 위치를 찾고, 추가

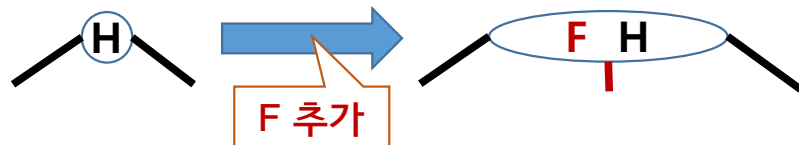
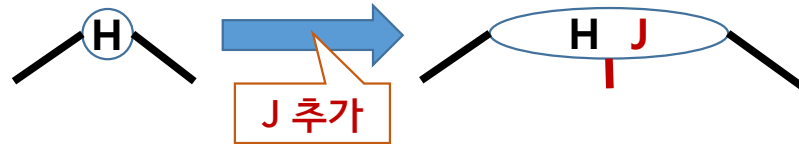
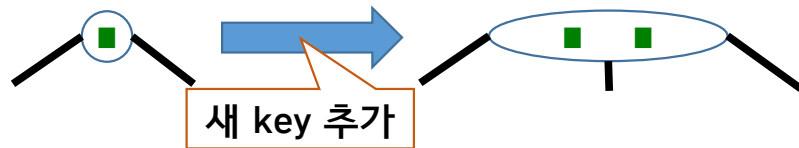
2-3 Tree에서의 **put(key, val)**: get(key)처럼 추가 위치 찾고 **k-node** → **(k+1)-node** 만들기

14

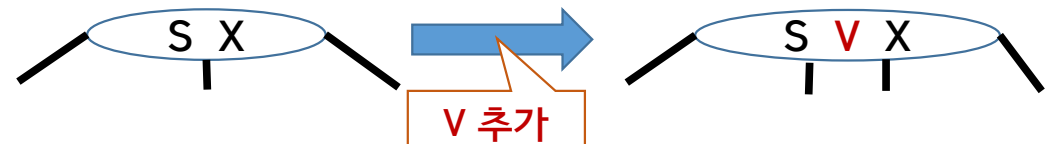
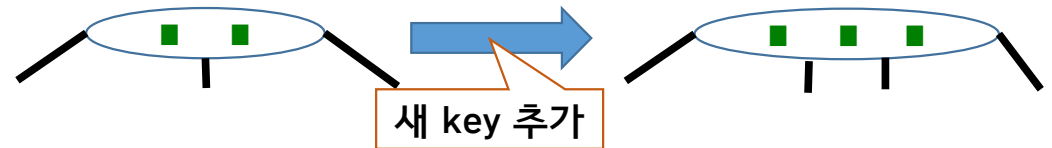
- get(key)와 같은 방식으로 key 있을 leaf 위치까지 이동
- 같은 key 이미 존재하면 value만 update
- 그렇지 않다면 **노드 타입에 따라** 아래와 같이 **(key, value) 추가**

Insert 방법이 트리를 늘 균형 잡히게 유지하는 핵심임

2-node → 3-node ε 분기 x (카운트는 st)

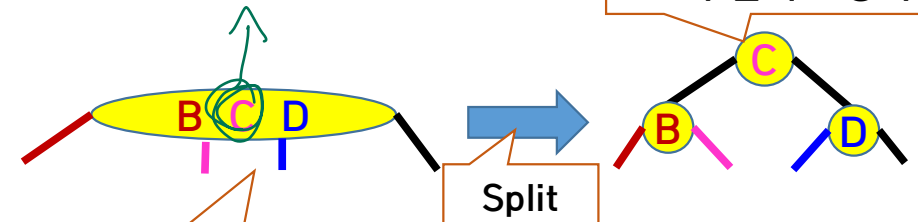
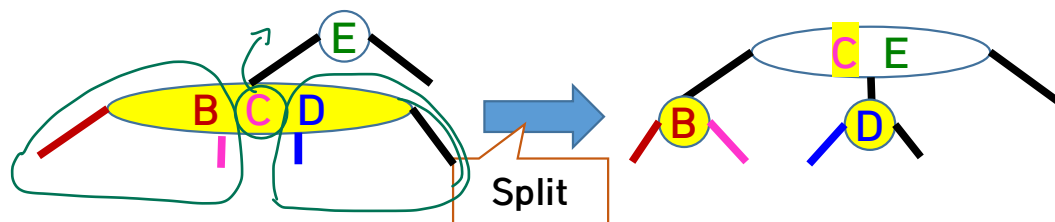
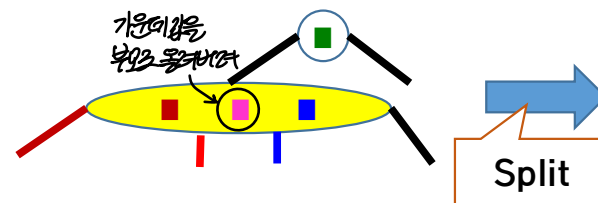


3-node → (임시로) 4-node ε 매개변수...



2-3 Tree에서의 put(key, val): 4-node는 2-node 두개로 split

- 4-node **a b c**는 가운데 key **b**를 부모 노드에 추가하며
- 2개의 2-node **a**, **c**로 분할
- 이렇게 만든 부모 노드도 4-node가 된다면 4-node 없을 때까지 (root 방향으로) split 반복
- root가 4-node라면, 가운데 key **b**를 새로운 root로 만들면서 **a**, **c**를 **b**의 자식으로 분할 (깊이 1 증가)



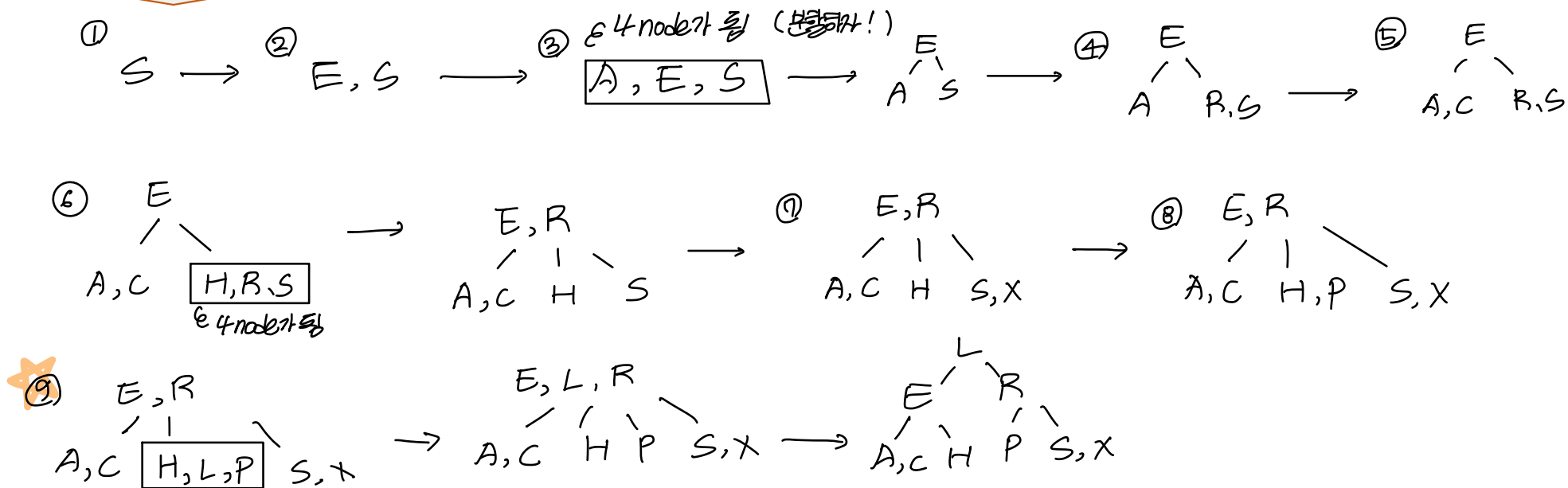
[Q] Split한 노드 아래의 링크는 분할 전 순서 그대로 분할된 두 노드 아래로 연결됨을 확인하시오.

4-node가 root인 경우

새로운 root 생기며 트리 깊이 1 증가

- get(key)와 같은 방식으로 key 있을 leaf 위치까지 이동 *탐색 → 리프까지 → 리프에 추가*
- 같은 key 이미 존재하면 value만 update
- 그렇지 않다면 2-node → 3-node, 3-node → 4-node 로 만들며 (key, value) 추가
- (root 방향으로 가며) 4-node가 있다면 2-node 2개로 split 반복

[Q] 비어 있는 트리에 Key를 S, E, A, R, C, H, X, P, L 순서로 put() 하는 과정을 그려 보시오.



[Q] 앞 문제의 전 과정에서 root ~ leaf 까지 모든 경로의 깊이가 같음을 확인 하시오.

3node가 ~~형성~~ *형성*되면서 깊이가 깊어지는 걸 막기함.

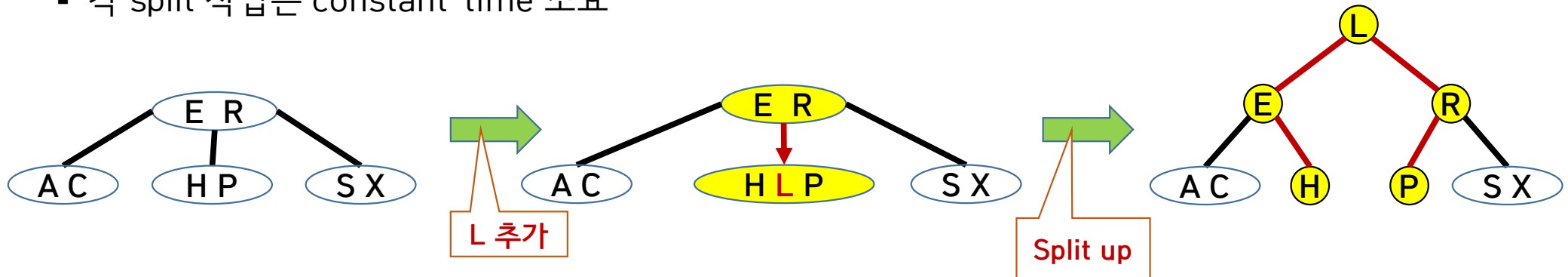
→ 무거워지면 무게를 조금 밑으로 나눠서 푸는 것임

2-3 Tree에서의 **put(key, val): 내려가며 추가위치 찾고, 올라오며 split**

- get(key)와 같은 방식으로 key 있을 leaf 위치까지 이동
- 같은 key 이미 존재하면 value만 update
- 그렇지 않다면 2-node \rightarrow 3-node, 3-node \rightarrow 4-node 로 만들며 (key, value) 추가
- (root 방향으로 가며) 4-node가 있다면 2-node 2개로 split 반복

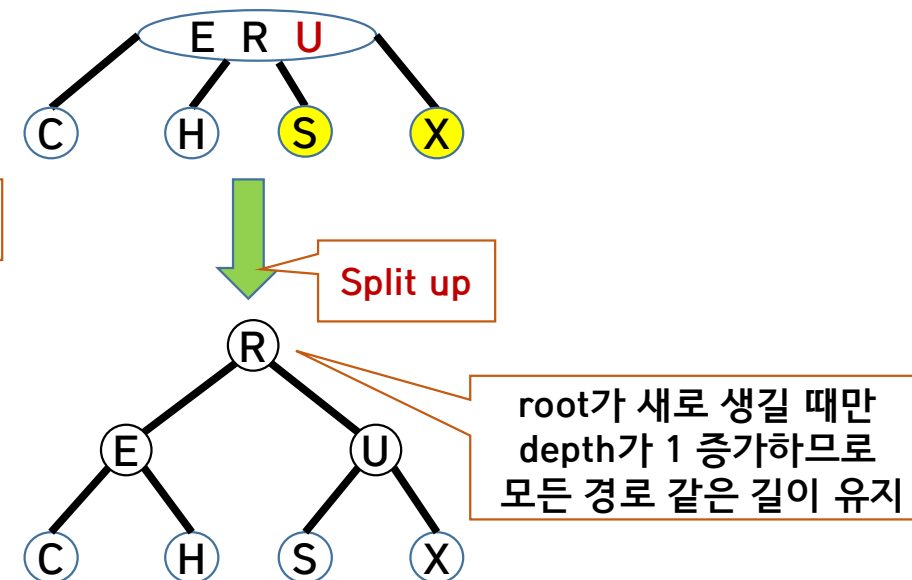
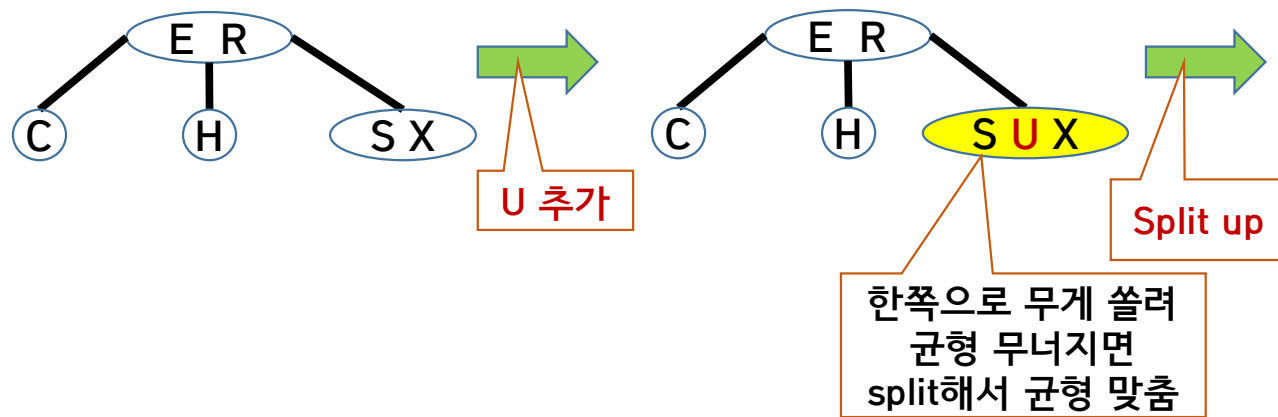
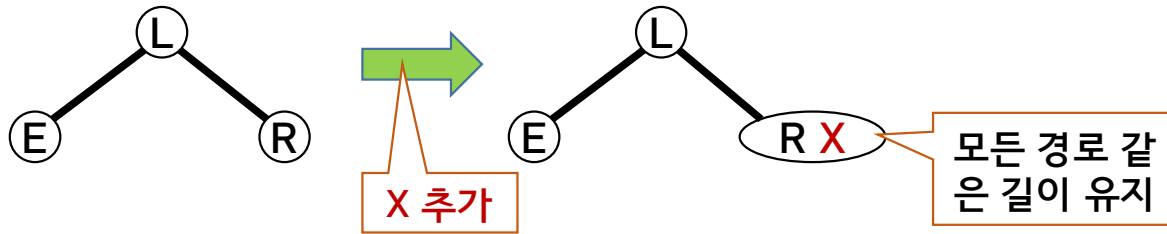
$\therefore \sim \log n$

- 트리 깊이까지 내려가며 새 (key, value) 추가할 leaf 찾음: $\sim \text{depth}$
- 4-node인 경우 root로 거슬러 올라가며 (4-node 없을때까지) split 반복: $\sim \text{depth}$
 - 각 split 작업은 constant-time 소요



2-3 Tree에서의 put(key, val):

(1) root~leaf까지 모든 경로 같은 길이 유지 & (2) 왼쪽/오른쪽 균형 유지





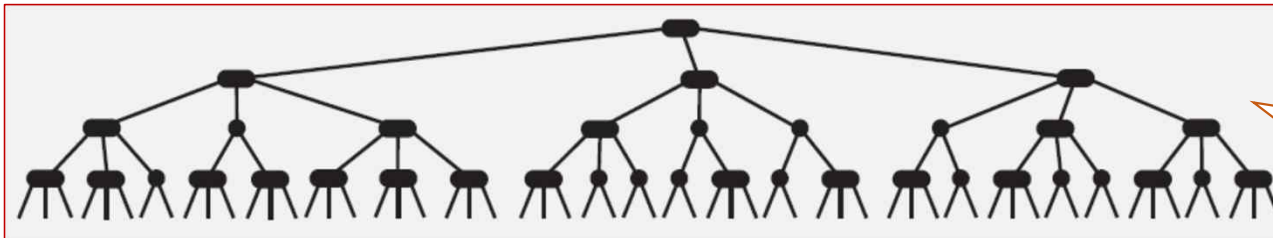
Insert and Delete 1..20





2-3 tree 성능: worst case $\sim \log N$

- root ~ leaf 까지 모든 경로가 같은 길이 \rightarrow 균형 잡힌 트리(트리 깊이 최소화)



기본 BST보다 훨씬 양쪽이
균형 잡힌 형태 유지

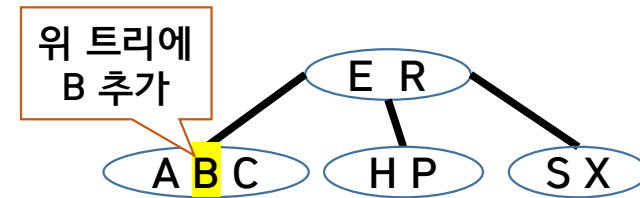
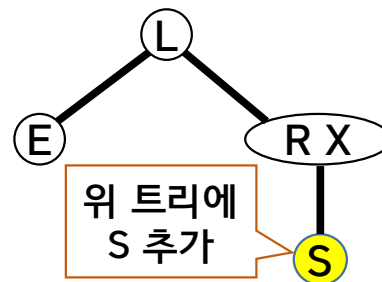
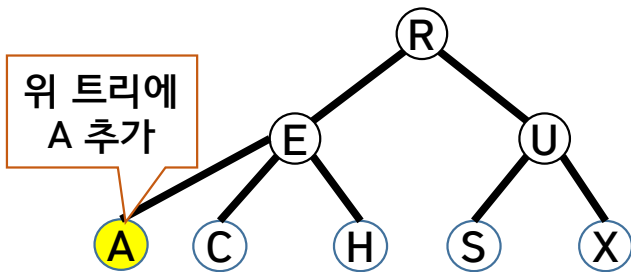
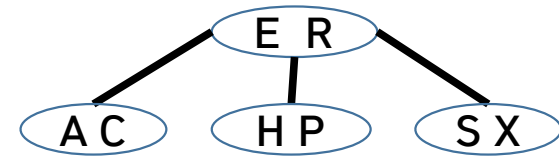
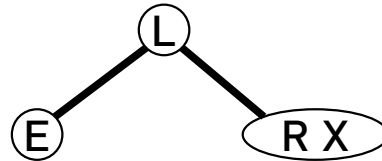
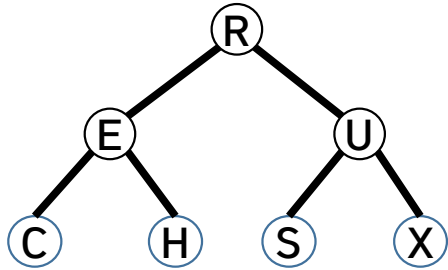
- Tree 깊이
 - worst case: $\log_2 N$ (모두 2-node인 경우)
 - best case: $\log_3 N$ (모두 3-node인 경우)
 - $N=1,000,000$ 인 경우 12~20
 - $N=1,000,000,000$ 인 경우 18~30
- 따라서 $\sim \log N$ 성능 보장



2-3 tree의 구현

- 처음부터 코드 작성하는 것은 아래와 같은 이유로 복잡하고 오류 발생 가능성 높음
 - 3가지 node type (2-node, 3-node, 4-node) 구분 처리
 - 3-node 따라 탐색해 내려갈 때는 대소 비교 여러 회 필요
 - Leaf에 새 값 추가 후 root로 거슬러 올라가며 4-node split 필요
 - 4-node split할 때 여러 링크를 정확한 위치에 바꾸어 연결해 주어야 함
 - ...
- 2-3 tree는 BST와 유사하므로, 이미 검증된 BST 코드 활용해 구현하면 약간의 변형으로 대부분의 코드 재활용 가능! 이를 **Red**-Black BST라 함

[Q] ① 다음 중 2-3 tree가 아닌 것을 고르고, ② 이들을 2-3 tree의 규칙에 맞게 수정하시오.
(2-3 tree의 규칙에 따라 값을 추가했을 때 나올 수 없는 형태를 모두 고르시오.)





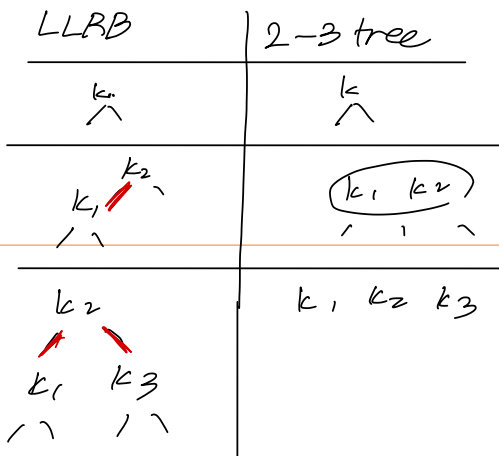
Symbol Table

Symbol Table이 무엇이고, 어떻게 구현하며, 어떤 경우에 어떻게 활용하는지 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. 2-3 Tree 활용한 Symbol Table 구현 (기본 BST보다 더 효율적인 Symbol Table 구현 알아보기)
03. BST 활용한 2-3 Tree 구현 (LLRB, Left-Leaning Red Black BST)
04. 1-D Range Search (symbol table 기능 추가)
05. Line-segment Intersection (symbol table & priority queue 활용 예)
06. 실습: Symbol Table 기능 추가 & 활용

BST 코드 변경 최소화하며
2-3 tree 아이디어 구현

↳ "색깔" 추가.



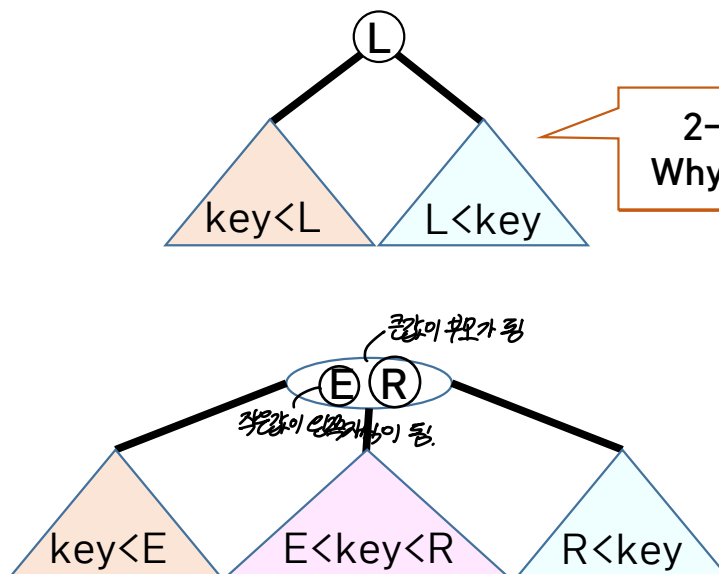


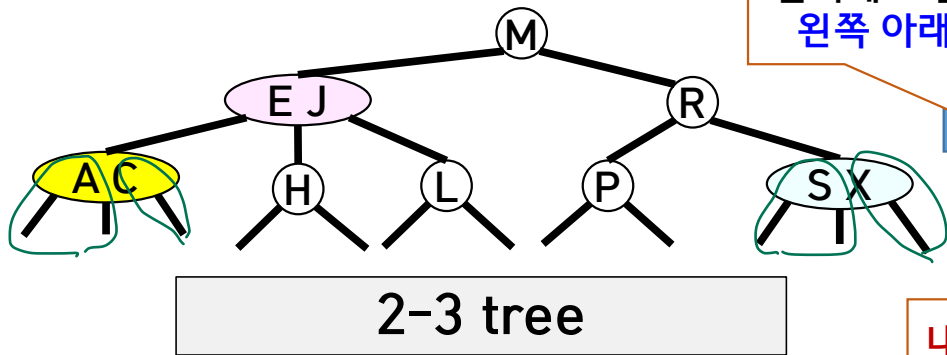
LLRB(Left-Leaning Red-Black) BST: BST로 표현한 2-3 tree

- ① 내부 연결(Red), 외부 연결(Black) 구분해 표현
- ② 내부 연결은 왼쪽으로 기울어지도록(Left-Leaning) 표현:
 - 원래 BST의 특성에 맞게 작은 값을 큰 값 왼쪽에 두는데, 특히 큰 값이 부모가 되도록 표현

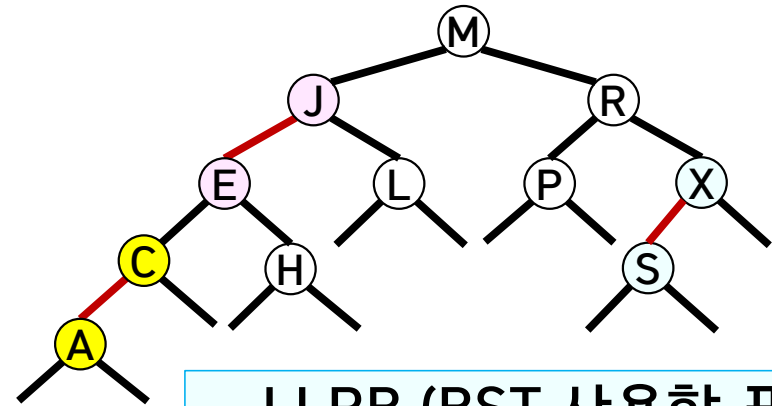
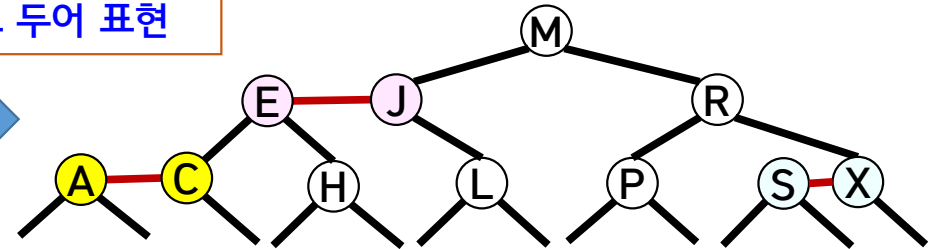
2-3 tree

LLRB (BST 사용한 표현)





3-node의 내부 연결을 Red로 구분되게 표현하고 작은 값을 큰 값 왼쪽 아래 자식으로 두어 표현



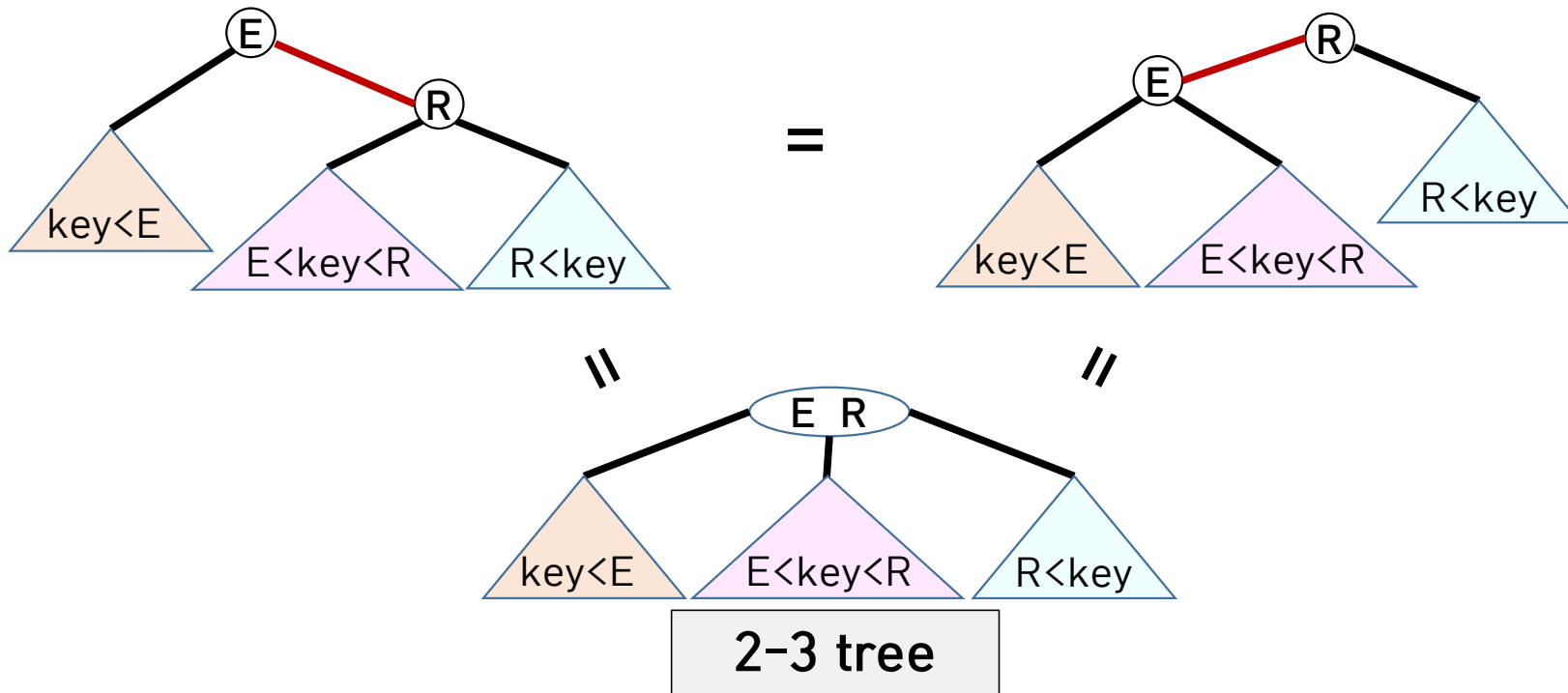


- 내부 연결에서
- 작은 값을 부모로
- 큰 값을 오른쪽 자식으로 함

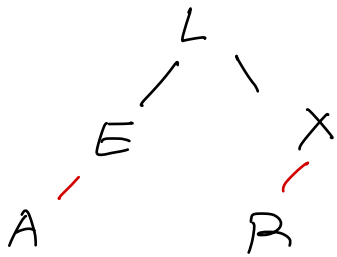
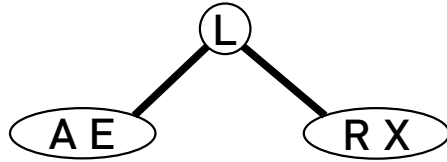
RLRB(Right-Leaning RB) BST

- 내부 연결에서
- **큰 값을 부모로**
- **작은 값을 왼쪽 자식으로 함**

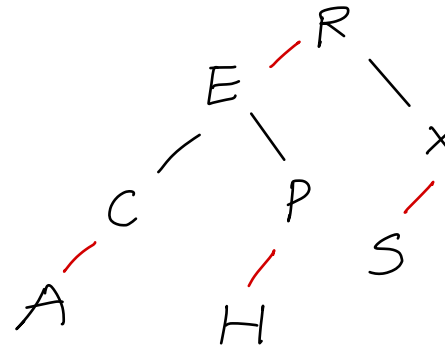
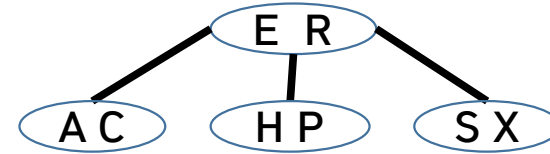
LLRB(Left-Leaning) BST



[Q] 아래 2-3 tree를 LLRB 형식으로 표현해 보시오.



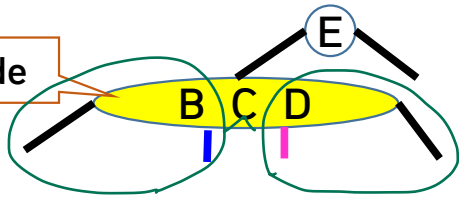
[Q] 아래 2-3 tree를 LLRB 형식으로 표현해 보시오.



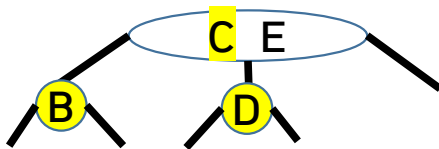
4-node: LL/RL 함께 사용해 표현하면, flip color 작업만으로 split 가능

2-3 tree

임시 4-node



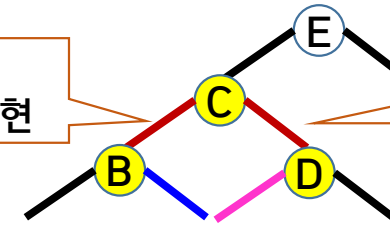
Split



LLRB (BST 사용한 표현)

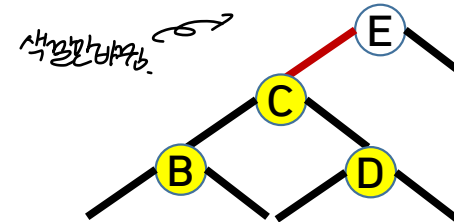
왼쪽은 Left-leaning하게 표현

오른쪽은 Right-leaning하게 표현

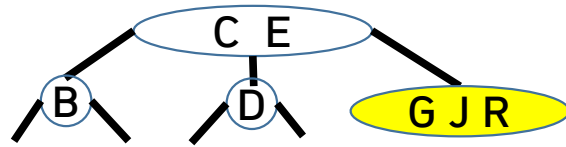


Split:

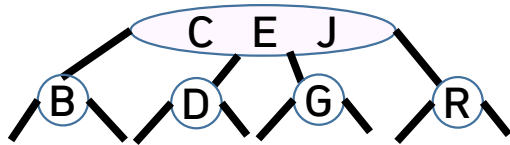
양쪽 자식과 link가 모두 RED면,
① 이들을 BLACK으로 만들며
② 부모와의 링크를 RED로 flip



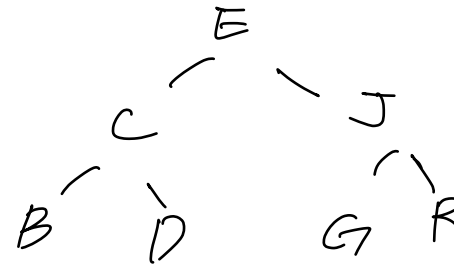
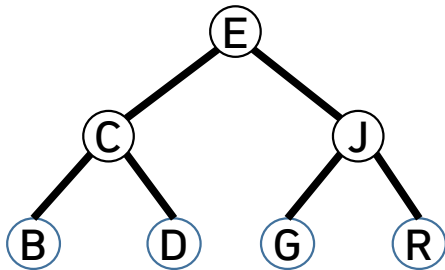
[Q] 아래는 왼쪽은 2-3 tree의 split 과정을 보여준다. 오른쪽에 이를 LLRB 형식으로 표현해 보시오.



Split (G, J, R)



Split (C, E, J)



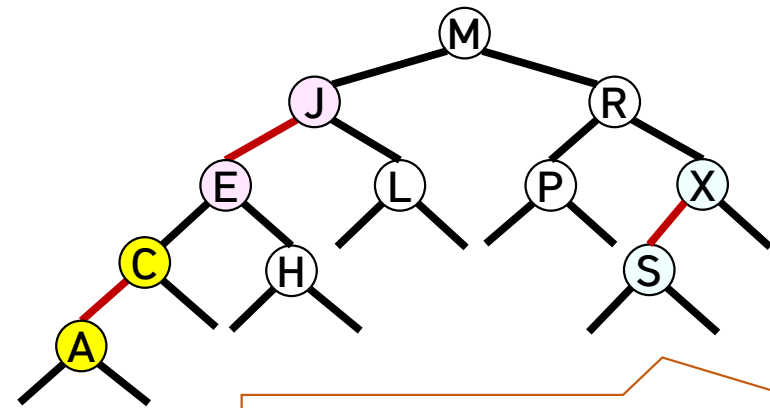


get(k) 포함 많은 탐색 작업을 BST 코드 그대로 사용해 수행 가능
(예: floor(k), select(i), inorder() 등)

- get(k): 링크 색깔 고려하지 않고 BST의 get(k) 그대로 사용해 탐색 가능

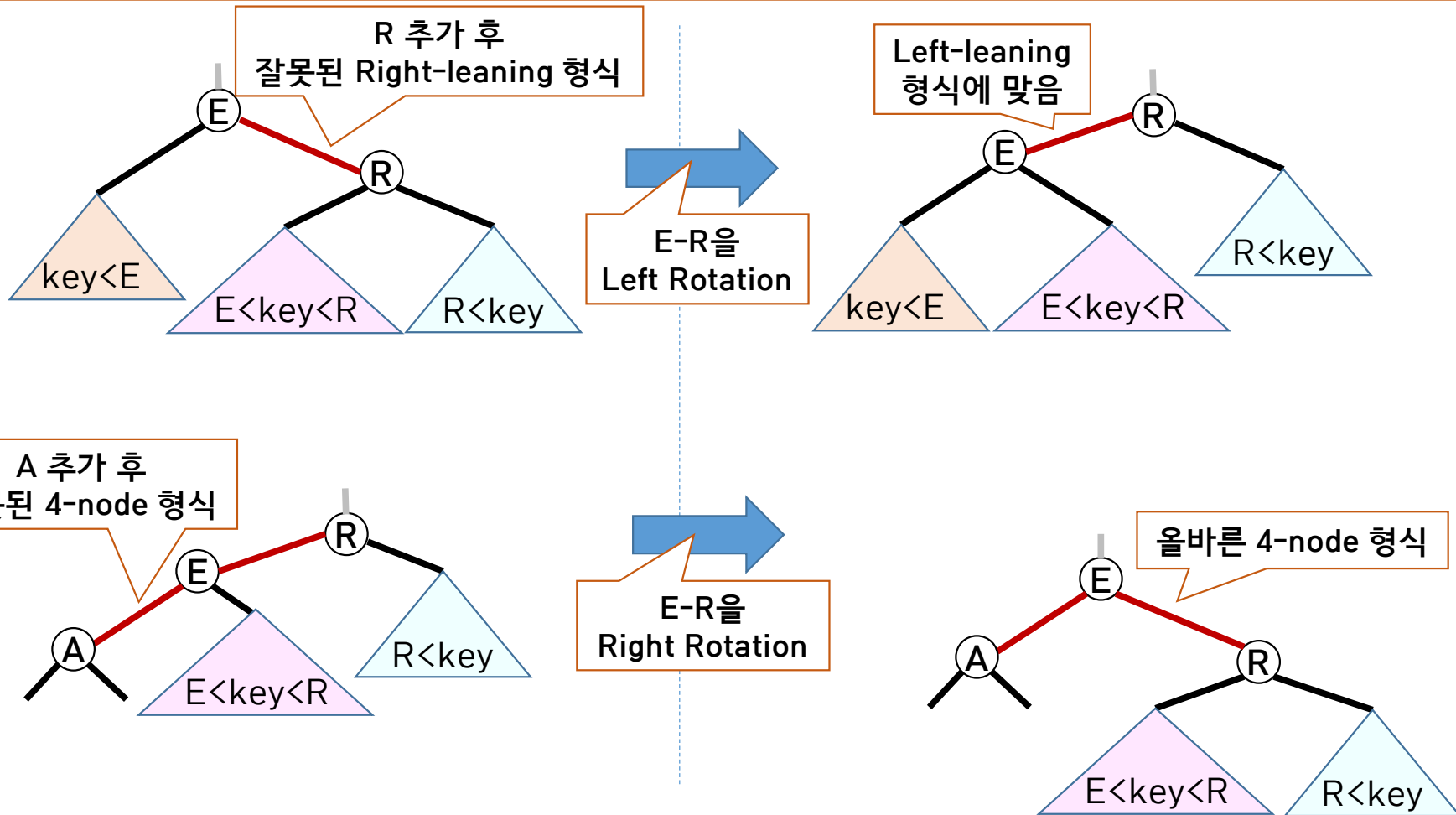
왼쪽 자식 < 부모 < 오른쪽 자식 조건
그대로 유지하도록 만들었으므로

```
def get(self, key):  
    x = self.root  
    while x != None:  
        if key < x.key: x = x.left  
        elif key > x.key: x = x.right  
        else: return x.val # key == x.key  
    return None # The key was NOT found
```



기본 BST보다 더 균형 잡힌 형태 유지
하므로 탐색 속도는 더 빠름

추가 작업 put(key, val):
 기본 BST 코드 그대로 사용 → LLRB 형식에 어긋나면 형식에 맞게 조정 (rotate)





LLRB 정리: 기본 BST의 2-3 tree 규칙 따른 변형

- 지금까지 본 것처럼 2-3 tree 규칙에 맞게 BST 운영하면 상당히 균형 잘 잡음
- Worst-case에도 대부분 작업에 대해 $\sim \log N$ 성능 유지
- 2-3 tree는 다양한 프로그래밍 언어에서 symbol table 구현에 사용
 - (Java) TreeMap, TreeSet, (C++) map, multimap, multiset, ...
- 2-3 tree의 확장형인 B-tree는 다양한 file system, database 구현에 사용
 - (Windows) NTFS, (Mac) HFS, HFS+, (Linux) JFX, XFS, ...
 - Database: ORACLE, PostgreSQL, ...



Symbol Table

Symbol Table이 무엇이고, 어떻게 구현하며, 어떤 경우에 어떻게 활용하는지 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. 2-3 Tree 활용한 Symbol Table 구현 (기본 BST보다 더 효율적인 Symbol Table 구현 알아보기)
03. BST 활용한 2-3 Tree 구현 (LLRB, Left-Leaning Red Black BST)
04. 1-D Range Search (symbol table 기능 추가)
05. Line-segment Intersection (symbol table & priority queue 활용 예)
06. 실습: Symbol Table 기능 추가 & 활용

1-D Range Search(lo, hi): [lo, hi] 범위의 key 찾기

작업	의미
get(k)	Key k에 대한 value 얻기
put(k,v)	새 원소 (k,v) 추가
delete(k)	Key가 k인 원소 삭제
min(), max()	Key의 최솟값/최댓값
floor(k) ceiling(k)	$\leq k$ 인 가장 큰 key $\geq k$ 인 가장 작은 key
rank(k)	k보다 작은 key의 개수
select(i)	(크기 순) i번째 key 얻기
ordered iteration	모든 key를 정렬된 순서로 얻기
rangeSearch(lo,hi)	$lo \leq key \leq hi$ 인 모든 key 찾기
rangeCount(lo,hi)	$lo \leq key \leq hi$ 인 key의 개수

- range search 사용하는 SQL 구문 예:
- `select name from employee where salary >= 1000 and salary <= 2000`
- `select count(*) from employee where salary >= 1000 and salary <= 2000`

한 원소 찾는 get(k)보다 더 일반적으로
범위에 속한 여러 원소를 찾는 기능

1-D Range Search(lo, hi): [lo, hi] 범위의 key 찾기

작업	의미	작업	BST에 저장된 원소 (정렬순) or 반환된 결과
get(k)	Key k에 대한 value 얻기	put(B,1)	B
put(k,v)	새 원소 (k,v) 추가	put(D,2)	B D
delete(k)	Key가 k인 원소 삭제	put(A,3)	A B D
min(), max()	Key의 최솟값/최댓값	put(I,4)	A B D I
floor(k) ceiling(k)	$\leq k$ 인 가장 큰 key $\geq k$ 인 가장 작은 key	put(H,5)	A B D H I
rank(k)	k보다 작은 key의 개수	put(F,6)	A B D F H I
select(i)	(크기 순) i번째 key 얻기	put(P,7)	A B D F H I P
ordered iteration	모든 key를 정렬된 순서로 얻기	rangeSelect(G,K)	H I
rangeSearch(lo,hi)	$lo \leq key \leq hi$ 인 모든 key 찾기	rangeCount(G,K)	2
rangeCount(lo,hi)	$lo \leq key \leq hi$ 인 key의 개수		

범위에 속하는 원소를
정렬된 순으로 돌려주기

1-D Range Search(lo, hi): [lo, hi] 범위의 key 찾기

put(F,6)	A B D F H I
put(P,7)	A B D F H I P
rangeSelect(G,K)	H I
rangeCount(G,K)	2

Key	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
존재 여부	•	•		•		•		•	•							•										

rangeSelect(G,K)

- Key들이 **1차원 직선** 상 점들이고
- **1차원의 범위**(range, interval)를 검색어로 주어
- 답 찾는 과정으로 볼 수 있으므로
- **1-D (1-Dimension) range search**라 함
- 여러 key 사용하는 것을 multi-dimension range search라 함

1-D Range Search(lo, hi): 성능 목표 $\sim(R + \log N)$

작업	unordered list (순차적 탐색)	ordered array (이진 탐색)	LLRB BST (2-3 tree)
get(k)	$\sim N$	$\sim \log N$	$\sim \log N$
put(k,v)	$\sim N$	$\sim N$	$\sim \log N$
delete(k)	$\sim N$	$\sim N$	$\sim \log N$
min(), max()	$\sim N$	~ 1	$\sim \log N$
floor(k), ceiling(k)	$\sim N$	$\sim \log N$	$\sim \log N$
rank(k)	$\sim N$	$\sim \log N$	$\sim \log N$
select(i)	$\sim N$	~ 1	$\sim \log N$
ordered iteration	$\sim N \log N$	$\sim N$	$\sim N$
rangeSearch(lo,hi)	$\sim N$	$\sim (R + \log N)$	$\sim (R + \log N)$
rangeCount(lo,hi)	$\sim N$	$\sim \log N$	$\sim \log N$

정렬되지 않았으므로,
모든 원소 뒤져야 함

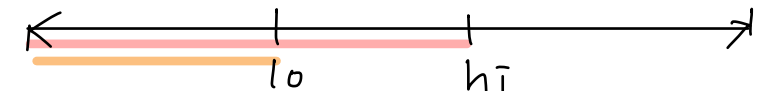
lo, hi 각각에 대해 이진 탐색하면
range 경계 찾을 수 있음

A B D F H I P T U V X Z

찾아내는 원소들의
개수.

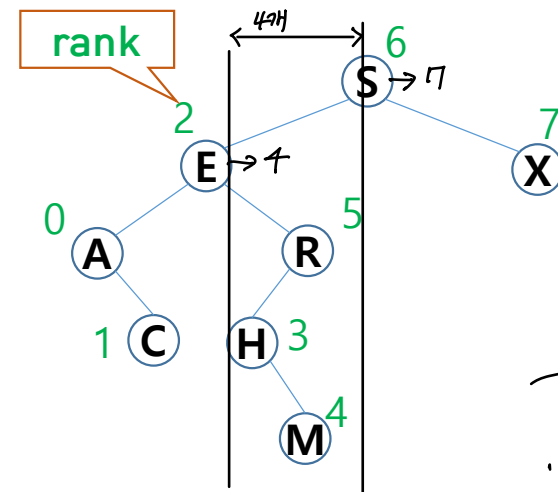
N: Symbol Table에 저장된 모든 key의 수
R: lo~hi Range에 속하는 key의 수

이항 탐색을 할 수 있음.



1-D Range Count(lo, hi): $lo \leq key \leq hi$ 범위의 key 개수 찾기

작업	의미
get(k)	Key k에 대한 value 얻기
...	...
rank(k)	key < k인 key의 개수 혹은 오름차순으로 볼 때 k의 순서 (0부터 시작)
...	...
rangeSearch(lo,hi)	$lo \leq key \leq hi$ 인 모든 key 찾기
rangeCount(lo,hi)	$lo \leq key \leq hi$ 인 key의 개수



```
def rangeCount(self, lo, hi):
    if self.contains(hi): return self.rank(hi) - self.rank(lo) + 1
    else: return self.rank(hi) - self.rank(lo) : High~Low
```

[Q] 위 코드에 따르면, rangeCount(F,T)는 무엇인가?

$$T \text{ rank} - F \text{ rank} = 4$$

$$(1) \quad (3)$$

[Q] 위 코드에 따르면, rangeCount(F,S)는 무엇인가?

$$6 - 3 + 1 = 4$$

[Q] 위 코드에 따르면, rangeCount(lo,hi)의 성능은 무엇에 비례하는가?

rank(k): key < k 인 key 개수

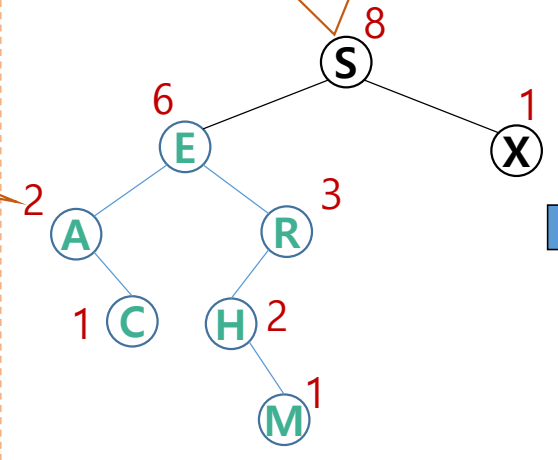
```

90 def rank(self, key): # How many keys < key?
91     def rankOnNode(x, key): # rank(key) on the subtree whose root is x
92         if x == None: return 0
93         if key < x.key: return rankOnNode(x.left, key)
94         elif key > x.key: return self.sizeOnNode(x.left) + 1 + rankOnNode(x.right, key)
95         else: return self.sizeOnNode(x.left) # key == x.key
96     return rankOnNode(self.root, key)

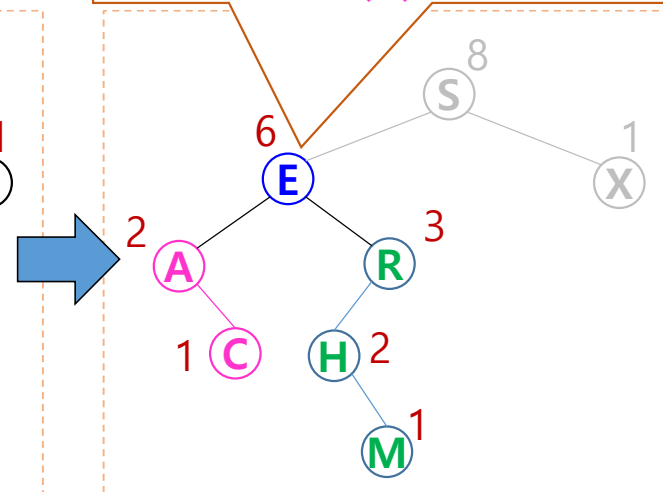
```

case 2: $R < S$, 따라서
 $\text{rankOnNode}(S, R) =$
 $\text{rankOnNode}(E, R)$

size: 자신 포함 아래 노드 개수

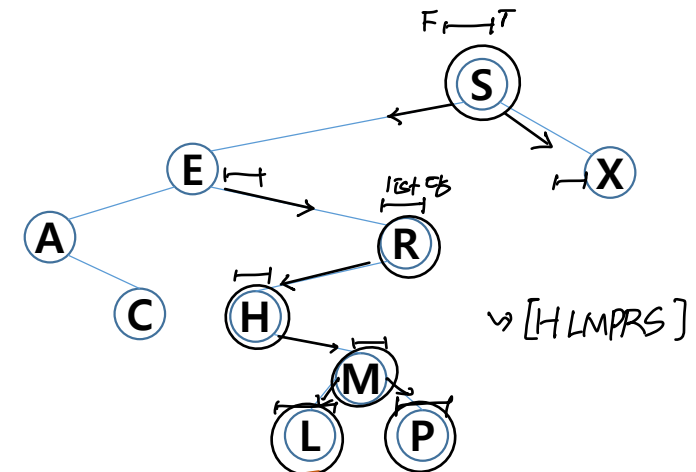


case 3: $R > E$, 따라서 $\text{rankOnNode}(E, R) =$
 $\text{sizeOnNode}(A) + 1 + \text{rankOnNode}(R, R)$



1-D Range Search(lo, hi): $lo \leq key \leq hi$ 범위의 모든 key (정렬된 순서로) 찾기

작업	의미
get(k)	Key k에 대한 value 얻기
...	...
ordered iteration	모든 key를 정렬된 순서로 얻기
rangeSearch(lo,hi)	$lo \leq key \leq hi$ 인 모든 key 찾기
rangeCount(lo,hi)	$lo \leq key \leq hi$ 인 key의 개수



- 노드 x 아래 모든 key에 대해 $lo \sim hi$ 범위의 key를 정렬된 순서로 얻으려면
- Root에서 시작해 아래를 재귀적으로 수행 (노드 x에 있다고 가정)

- ① (범위에 속하는 key 있을 수 있다면) x 왼쪽 subtree 탐색해 범위에 속한 모든 key 찾아 큐에 넣고
 - ② $lo \leq x.key \leq hi$ 라면 x.key를 큐에 넣고
 - ③ (범위에 속하는 key 있을 수 있다면) x 오른쪽 subtree 탐색해 범위에 속한 모든 key 찾아 큐에 넣고
- BST의 key가 그러한 대소 관계에 따라 저장되어 있기 때문

[Q] 아래 규칙 ①~③에 따라 rangeSearch(F,T) 수행해보기

ordered iteration: BST의 모든 key
정렬된 순서로 반환

```
def inorder(self):
    def inorderOnNode(x, q):
        if x == None: return
        inorderOnNode(x.left, q)
        q.append(x.key)
        inorderOnNode(x.right, q)
    q = []
    inorderOnNode(self.root, q)
    return q
```

```
for k in bst.inorder():
    print(k)
```

실행 결과

A
C
E
H
M
R
S
X

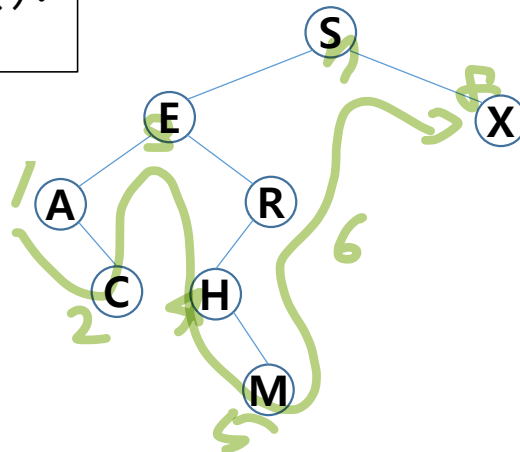
1-D Range Search(lo, hi): $lo \leq key \leq hi$ 범위 모든
key 정렬된 순서로 반환

```
def rangeSearch(self, lo, hi):
    def rangeSearchOnNode(x, lo, hi, q):
        if x == None: return
        if lo < x.key: rangeSearchOnNode(x.left, lo, hi, q)
        if lo <= x.key and x.key <= hi: q.append(x.key)
        if x.key < hi: rangeSearchOnNode(x.right, lo, hi, q)
    q = []
    rangeSearchOnNode(self.root, lo, hi, q)
    return q
```

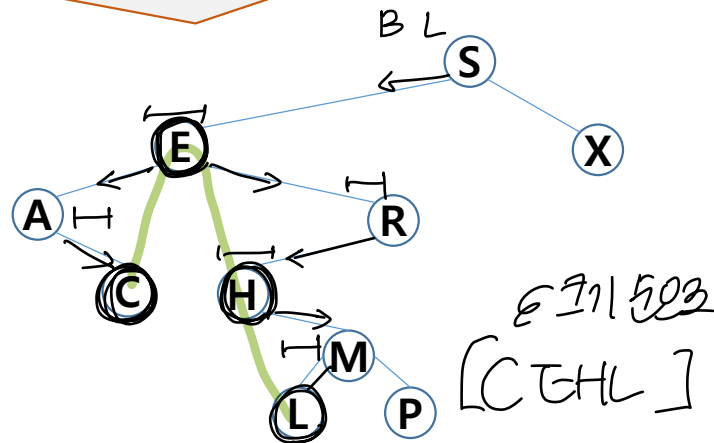
```
for k in bst.rangeSearch('F', 'T'):
    print(k)
```

실행 결과

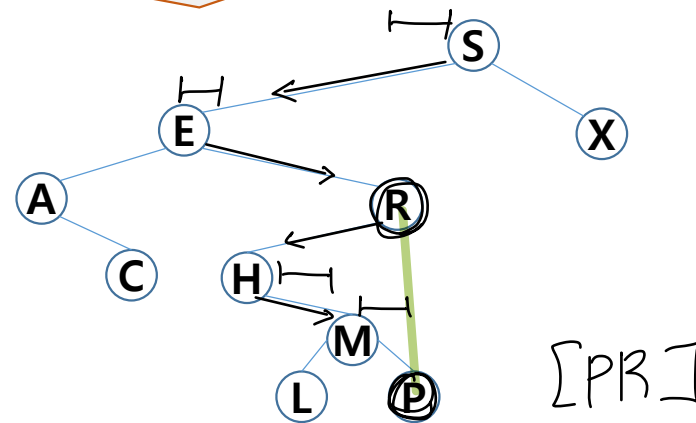
H
M
R
S



[Q] 아래 규칙 ①~③에 따라 rangeSearch(B,L) 수행하는 과정을 그려 보시오. 4개



[Q] 아래 규칙 ①~③에 따라 rangeSearch(P,R) 수행하는 과정을 그려 보시오. 2개



탐색 과정에서 거쳐 가지만 포함하지 않는 원소도 있으니 유의하시오.

[Q] 앞 두 문제를 보며 알고리즘의 수행시간이 $\sim(R+\log N)$ 임을 생각해 보시오. R은 질의한 range에 속하는 key의 수이다.

- Root에서 시작해 아래를 재귀적으로 수행 (노드 x에 있다고 가정)
- ① (범위에 속하는 key 있을 수 있다면) x 왼쪽 subtree 탐색해 범위에 속한 모든 key 찾아 큐에 넣고
- ② $lo \leq x.key \leq hi$ 라면 x.key를 큐에 넣고
- ③ (범위에 속하는 key 있을 수 있다면) x 오른쪽 subtree 탐색해 범위에 속한 모든 key 찾아 큐에 넣기



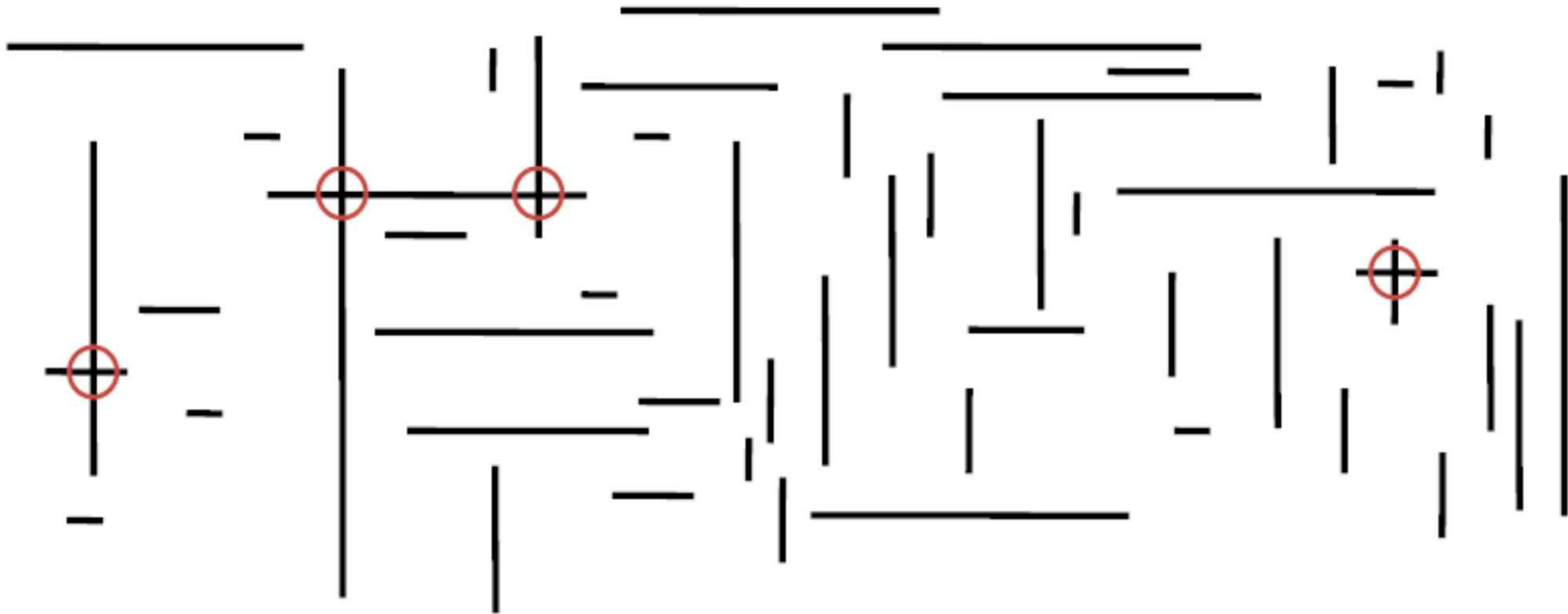
Symbol Table

Symbol Table이 무엇이고, 어떻게 구현하며, 어떤 경우에 어떻게 활용하는지 이해

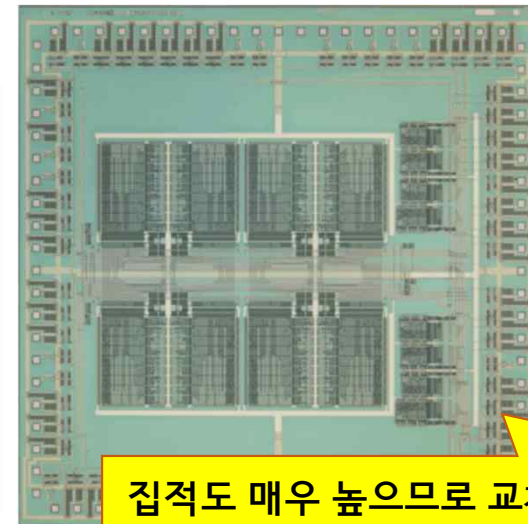
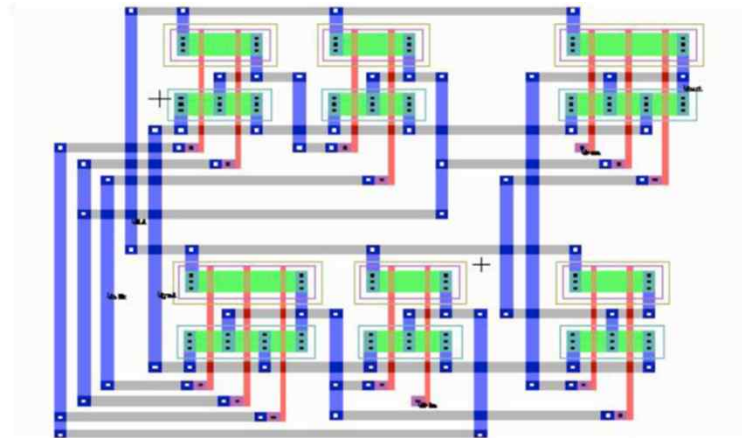
01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. 2-3 Tree 활용한 Symbol Table 구현 (기본 BST보다 더 효율적인 Symbol Table 구현 알아보기)
03. BST 활용한 2-3 Tree 구현 (LLRB, Left-Leaning Red Black BST)
04. 1-D Range Search (symbol table 기능 추가)
05. Line-segment Intersection (symbol table & priority queue 활용 예)
06. 실습: Symbol Table 기능 추가 & 활용

BST이기에 효율적인 range search 가능함 기억
(대소 관계에 따라 저장하므로)

Orthogonal line segment intersection: N개 수직/수평선 있을 때, 서로 교차하는 쌍 모두 찾기

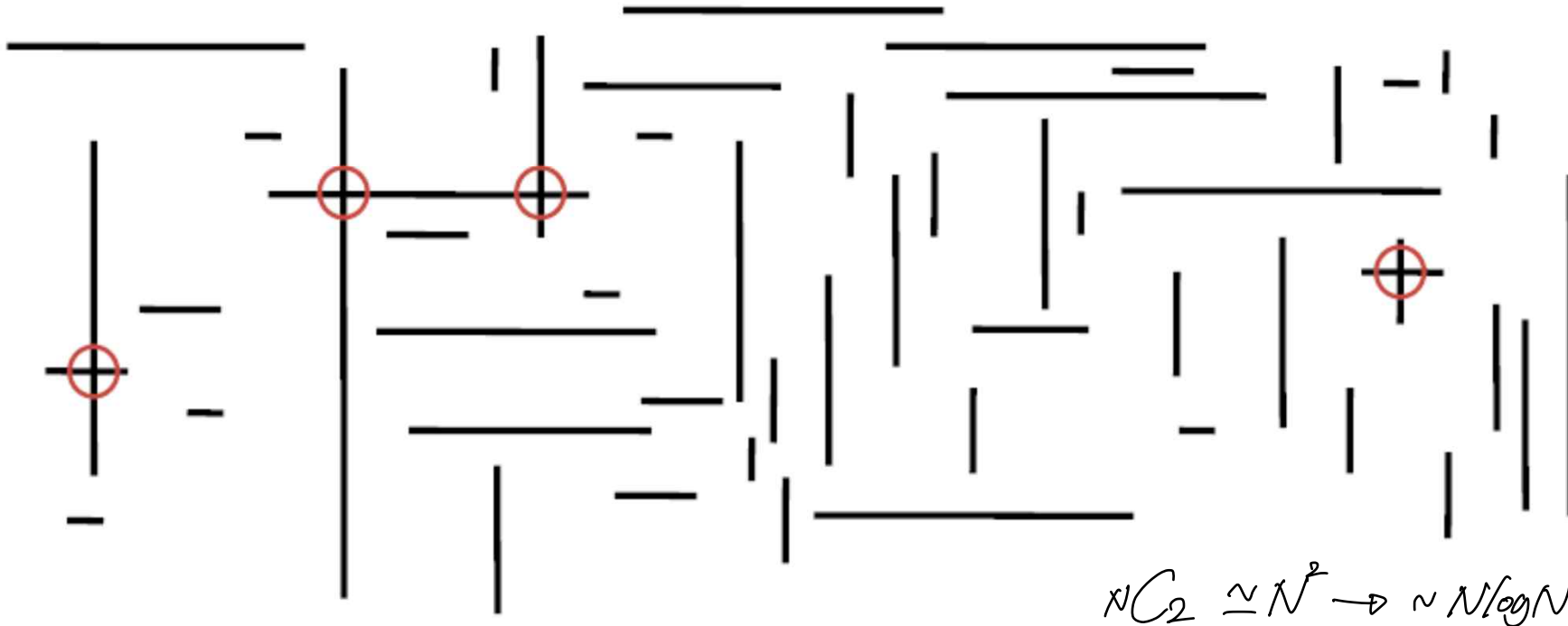


- 복잡한 반도체 소자 생산하며 (VLSI, Very Large Scale Integration)
- 소자 디자인과 검증에 컴퓨터 활용 시작 (CAD, Computer Aided Design)
- 컴퓨터로 설계, 검증 끝난 도면은 실제 HW로 생산
- 설계한 회로의 wire와 chip의 위치 결정 시 아래와 같은 규칙 검증 필요
 - 특정 wire들 간 혹은 wire와 칩 간에는 **서로 교차해서는 안 되며**
 - 지정된 거리 이상 떨어져 배치되어야 함
 - ...



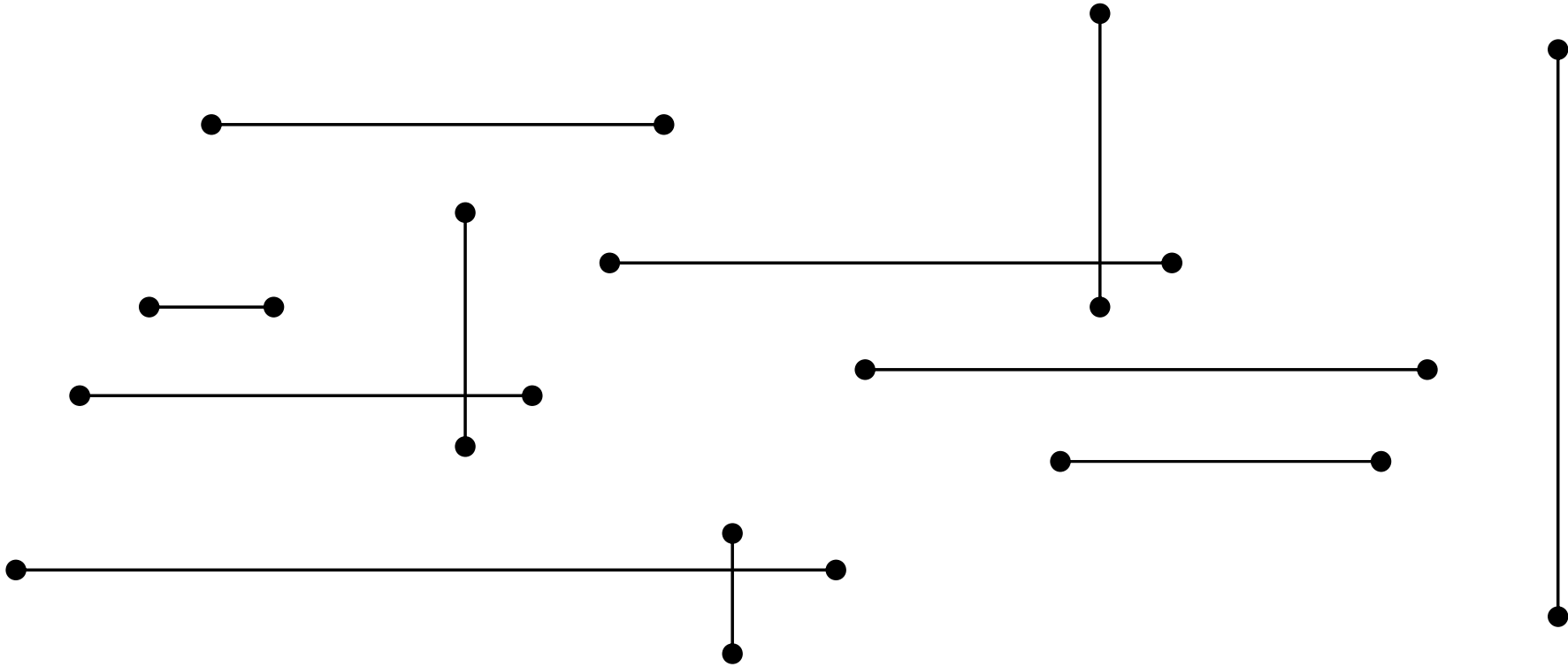
집적도 매우 높으므로 교차여부 확인하는
효율적인 알고리즘 필요!

Orthogonal line segment intersection: N개 수직/수평선 있을 때, 서로 교차하는 쌍 모두 찾기



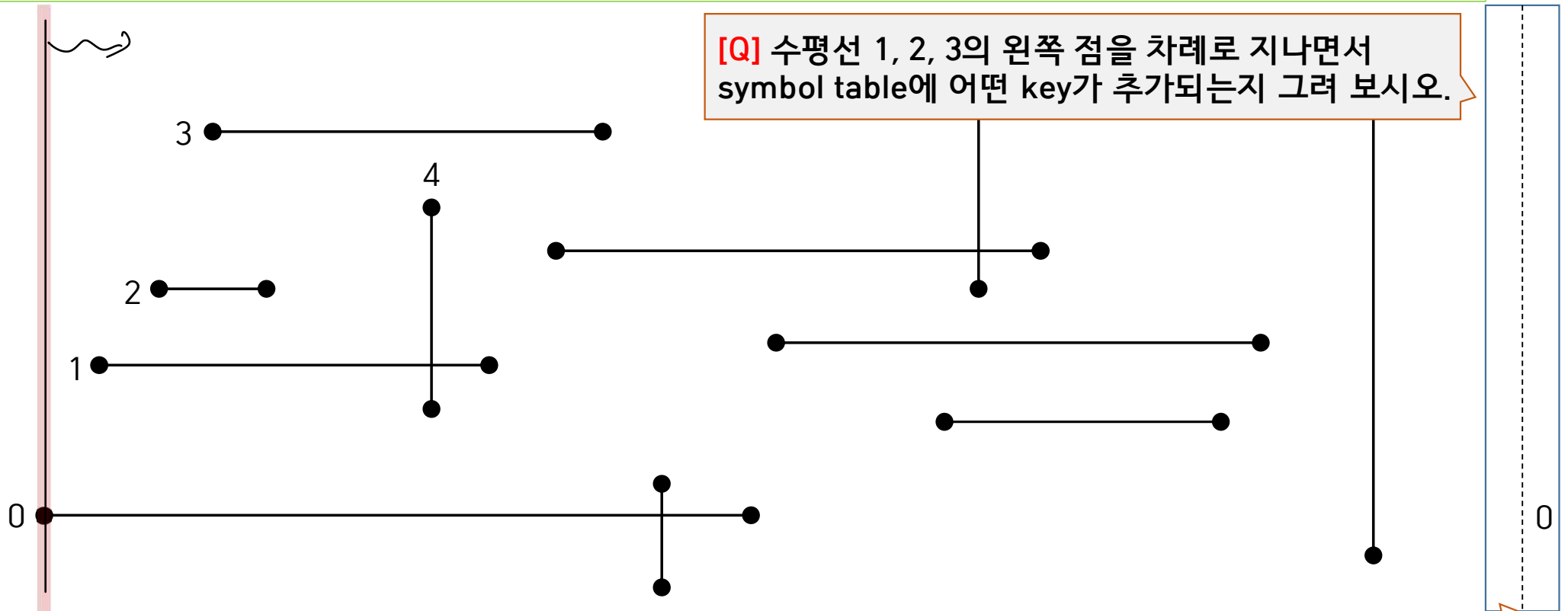
- Naïve (brute-force) 알고리즘: 모든 segment 쌍에 대해 교차 여부 확인 ($\sim N^2$)
- 더 효율적인 방법 필요

Sweep-line 알고리즘: 왼쪽→오른쪽 방향, 수직선 형태로 scan하며 교차 확인



- x좌표를 event로 보고 처리 (작은 좌표 → 큰 좌표 순서)
- 각 x좌표에서 지정된 작업 수행

Sweep-line 알고리즘: 왼쪽 → 오른쪽 방향, 수직선 형태로 scan하며 교차 확인



- x좌표를 event로 보고 처리 (작은 좌표 → 큰 좌표 순서)

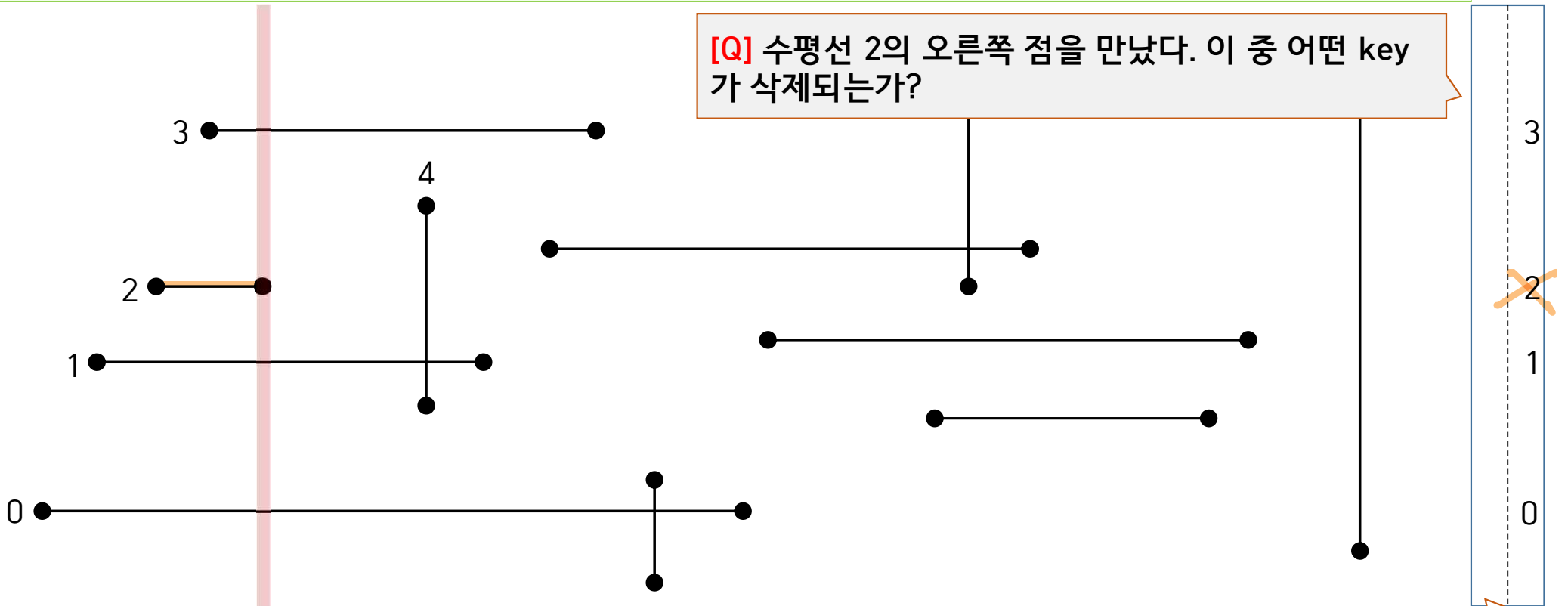
① 수평선 왼쪽 점 만나면: y 좌표를 table에 추가

y좌표 저장하는 table

붉은 선 옆에서 보는 단면으로 생각해도 됨

Sweep-line 알고리즘: 왼쪽→오른쪽 방향, 수직선 형태로 scan하며 교차 확인

[Q] 수평선 2의 오른쪽 점을 만났다. 이 중 어떤 key가 삭제되는가?



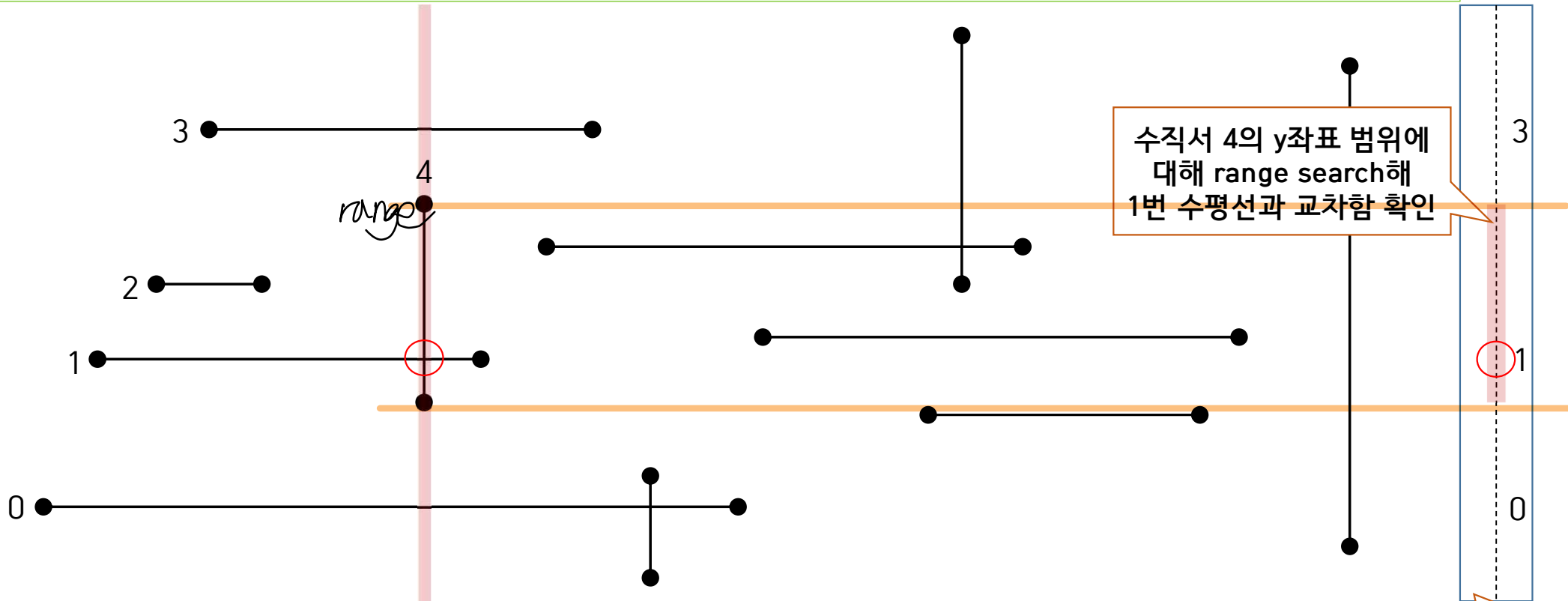
▪ x좌표를 event로 보고 처리 (작은 좌표 → 큰 좌표 순서)

- ① 수평선 왼쪽 점 만나면: y 좌표를 table에 추가
- ② 수평선 오른쪽 점 만나면: y 좌표를 table에서 제거

y좌표 저장하는
table

붉은 선 옆에서 보는
단면으로 생각해도 됨

Sweep-line 알고리즘: 왼쪽 → 오른쪽 방향, 수직선 형태로 scan하며 교차 확인

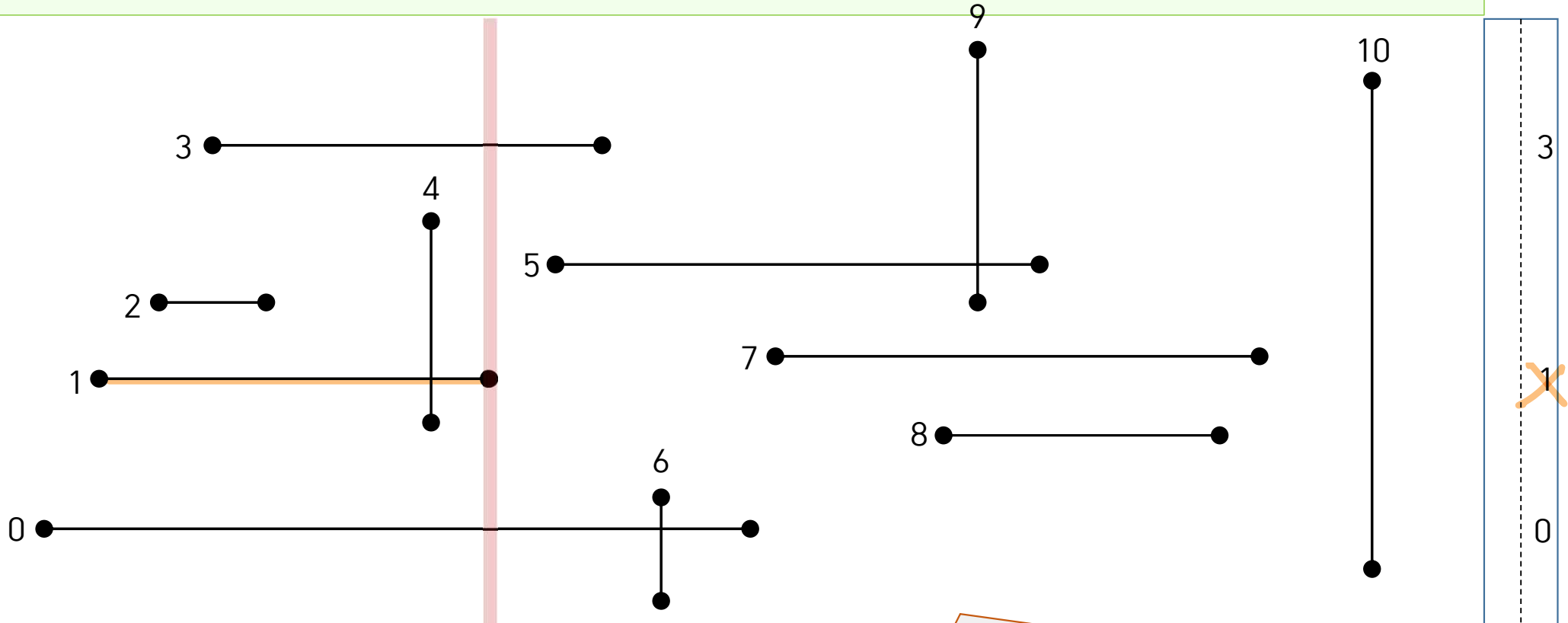


▪ x좌표를 event로 보고 처리 (작은 좌표 → 큰 좌표 순서)

- ① 수평선 왼쪽 점 만나면: y 좌표를 table에 추가
- ② 수평선 오른쪽 점 만나면: y 좌표를 table에서 제거
- ③ 수직선 만나면: 수직선의 y 범위에 대해 range search해 교차점 확인

y좌표 저장하는
table

Sweep-line 알고리즘: 왼쪽→오른쪽 방향, 수직선 형태로 scan하며 교차 확인

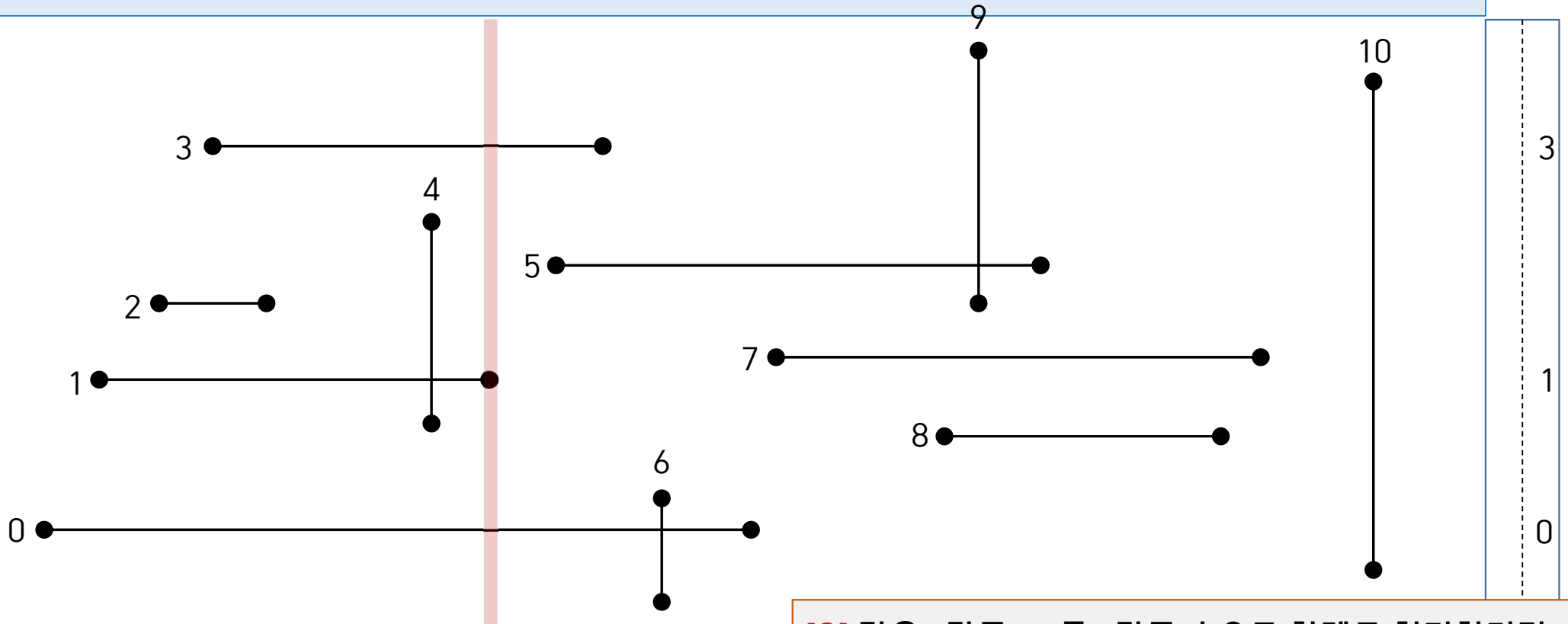


▪ x좌표를 event로 보고 처리 (작은 좌표 → 큰 좌표 순서)

- ① 수평선 왼쪽 점 만나면: y 좌표를 table에 추가
- ② 수평선 오른쪽 점 만나면: y 좌표를 table에서 제거
- ③ 수직선 만나면: 수직선의 y 범위에 대해 range search해 교차점 확인

[Q] 이후에는 table에 어떤 값 추가/삭제되고 어떤 교차점을 확인하는지 차례로 써 보시오.

Sweep-line 알고리즘의 구현



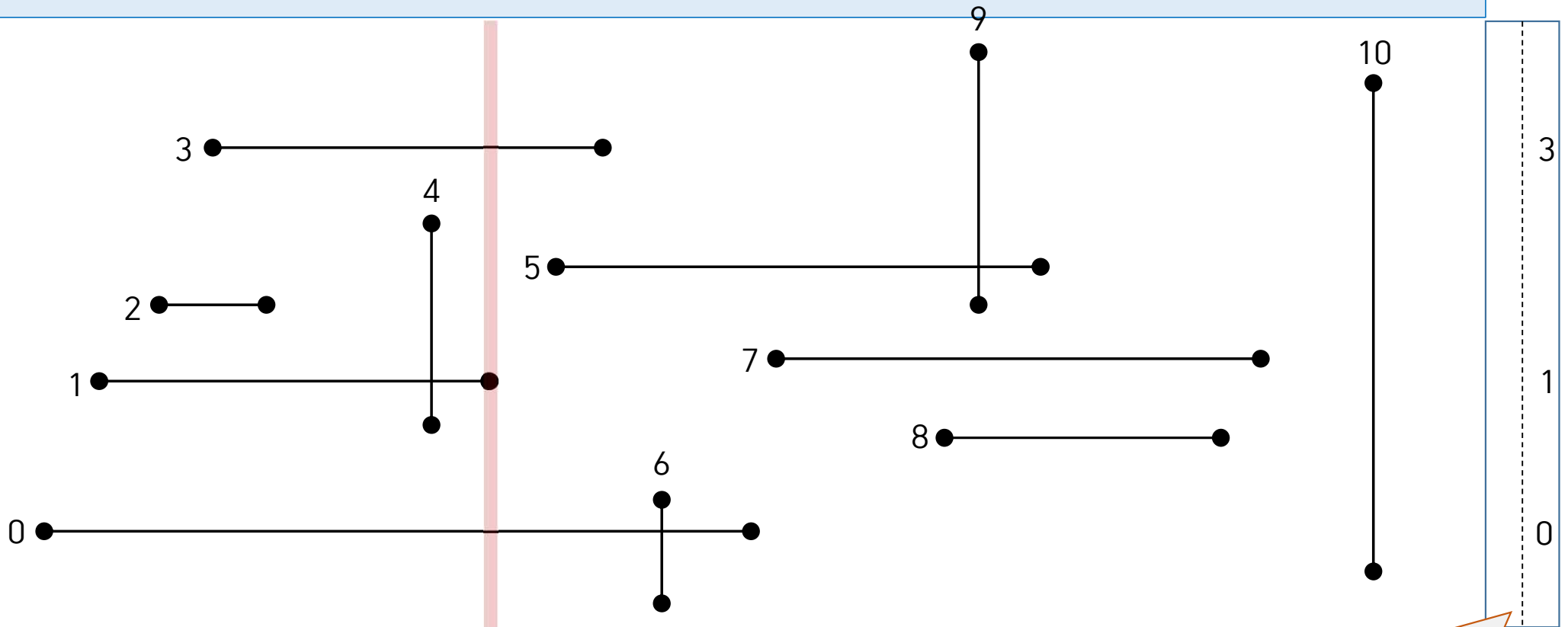
■ x좌표를 event로 보고 처리 (작은 좌표 → 큰 좌표 순서)

- ① 수평선 왼쪽 점 만나면: y 좌표를 table에 추가
- ② 수평선 오른쪽 점 만나면: y 좌표를 table에서 제거
- ③ 수직선 만나면: 수직선의 y 범위에 대해 range search해 교차점 확인

[Q] 작은 x좌표 → 큰 x좌표 순으로 차례로 처리하려면, 지금까지 배운 어떤 방법(자료구조) 사용하면 적절한가?

Sweep-line 알고리즘의 구현

55

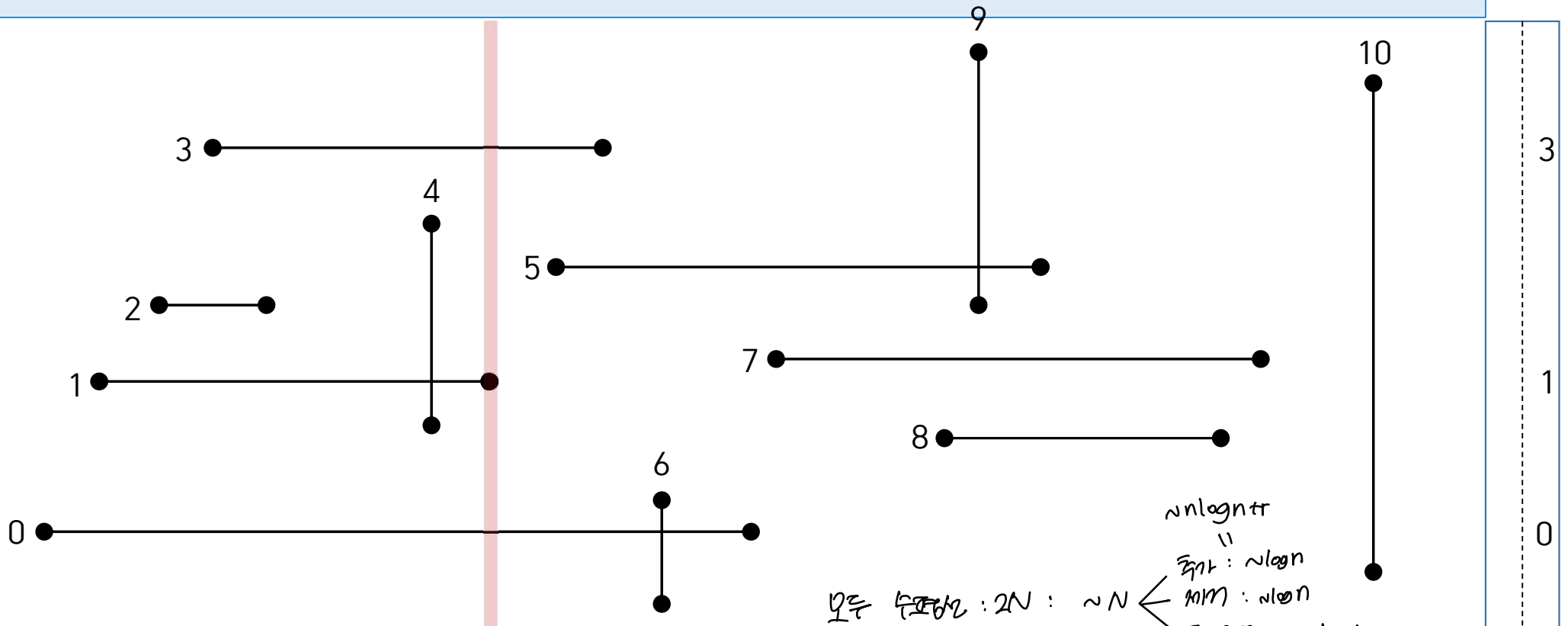


▪ x좌표를 event로 보고 처리 (작은 좌표 → 큰 좌표 순서)

- ① 수평선 왼쪽 점 만나면: y 좌표를 table에 추가
- ② 수평선 오른쪽 점 만나면: y 좌표를 table에서 제거
- ③ 수직선 만나면: 수직선의 y 범위에 대해 range search해 교차점 확인

[Q] 추가, 제거, range search 효율적으로 할 수 있는 방법(자료구조)는?

Sweep-line 알고리즘의 성능



▪ x좌표를 event로 보고 처리 (작은 좌표 \rightarrow 큰 좌표 순서)

- ① 수평선 왼쪽 점 만나면: y 좌표를 table에 추가
- ② 수평선 오른쪽 점 만나면: y 좌표를 table에서 제거
- ③ 수직선 만나면: 수직선의 y 범위에 대해 range search해 교차점 확인

[Q] 앞에서 선정한 방법 사용했을 때 성능은?
Brute-force 방식보다 더 빠른가?



정리: Sweep-line algorithm

- N개 수직/수평선 중 교차하는 쌍 모두 찾기
- 각 직선에 대해
 - 다른 모든 라인과 비교하는 대신 ($\sim N$)
 - 교차 가능한 값만 BST에서 range search 하도록 해서 ($\sim \log N$)
 - 더 빠르게 찾음



Symbol Table

Symbol Table이 무엇이고, 어떻게 구현하며, 어떤 경우에 어떻게 활용하는지 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
02. 2-3 Tree 활용한 Symbol Table 구현 (기본 BST보다 더 효율적인 Symbol Table 구현 알아보기)
03. BST 활용한 2-3 Tree 구현 (LLRB, Left-Leaning Red Black BST)
04. 1-D Range Search (symbol table 기능 추가)
05. Line-segment Intersection (symbol table & priority queue 활용 예)
06. 실습: Symbol Table 기능 활용 (Sweep-line 알고리즘 구현)



실습 목표: Sweep-line 알고리즘 구현

- Symbol Table과 Priority Queue를 적재적소에 활용해 보기

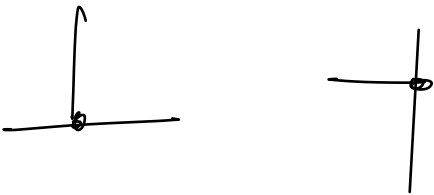
프로그램 구현 조건

- sweep line algorithm 수행하는 함수 구현
 - def sweepLine(**segments**):
- 입력 **segments**: 수평선, 수직선의 리스트로 각 선은 **Segment 클래스 객체**
- 반환 값
 - 모든 교차점의 리스트 반환
 - 리스트의 각 교차점: 교차점 만드는 (수평선, 수직선)의 tuple로 표현 (즉 **두 Segment 객체의 tuple**)
 - Sweep-line 알고리즘이 찾는 순서대로 리스트에 담음
- 이번 시간에 제공한 코드 SweepLine.py에 위 함수 추가해 제출
 - 위 코드에 포함된 Segment 클래스는 반드시 사용해야 함
 - 작은 x좌표 순으로 처리하기 위해 위 파일에 이미 import된 PriorityQueue 클래스 사용 (PQ 복습 위해 정렬 사용하지 말고 반드시 PQ 사용할 것)
 - 또한 Symbol Table로는 위 파일에 이미 import된 LLRB 클래스 사용

결과를 print하지 말고
반환하세요.

프로그램 구현 조건

- 최종 결과물로 **SweepLine.py** 파일 하나만 제출하며
- 이 파일과 수업에서 제공한 RedBlackBST.py 만으로 코드가 동작해야 함
- import는 원래 SweepLine.py 파일에서 import하던 2개 패키지 외에는 추가로 할 수 없음 (queue.PriorityQueue, RedBlackBST.LLRB)
- 각자 테스트에 사용하는 모든 코드는 반드시 `if __name__ == "__main__":` 아래에 넣어
- 제출한 파일을 import 했을 때는 실행되지 않도록 할 것



이런경우 2차X

구현된 API 정리 Segment Class - 한 line 나타내는 클래스 (수직선 or 수평선)

이미 구현된 클래스로 각 함수의 의미 이해하고 사용하기

```
class Segment:
```

```
    def __init__(self, x1, y1, x2, y2): # Segment 생성자. (x1,y1)과 (x2,y2)를 잇는 직선 생성
        # 이들은 멤버 변수 self.x1, self.y1, self.x2, self.y2에 저장됨
        # 수직선인 경우(x1==x2): y1, y2 중 더 작은 값이 self.y1에, 더 큰 값이 self.y2에 저장됨
        # 수평선인 경우(y1==y2): x1, x2 중 더 작은 값이 self.x1에, 더 큰 값이 self.x2에 저장됨
```

```
    def isHorizontal(self): # Segment 객체가 수평선이면 True를, 그렇지 않으면 False 반환
```

```
    def isVertical(self): # Segment 객체가 수직선이면 True를, 그렇지 않으면 False 반환
```

```
    def __str__(self): # Segment 객체에 저장된 내용을 string 형태로 반환. 출력, debugging에 사용
        # 반환하는 스트링: "(x1,y1)--(x2,y2)"
```

```
    def __eq__(self): # == 사용해 두 Segment 객체가 같음을 확인할 때 호출됨. Grading, debugging에 사용
```

구현된 API 정리: LLRB Class - Left-Leaning Red-Black BST 나타내는 클래스

```
class LLRB:
    # 이미 구현된 함수
    def __init__(self): # LLRB 생성자

    def put(self, key, value): # (key, value) 쌍을 BST에 추가. key가 이미 존재한다면 value만 update

    def get(self, key): # key에 대응되는 value 찾아 반환
    def contains(self, key): # key가 저장되어 있다면 True를, 그렇지 않다면 False 반환

    def delete(self, key): # key 및 대응되는 value를 BST에서 제거

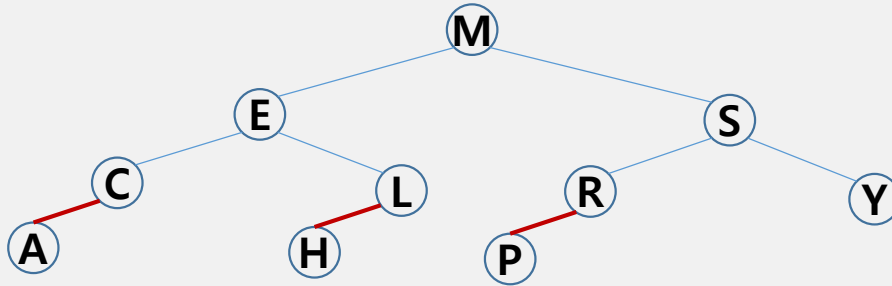
    def rank(self, key): # key보다 작은 key의 개수 반환

    def inorder(self): # BST에 저장된 key를 정렬된 순서로 리스트에 저장해 반환
    def levelorder(self): # BST에 저장된 key를 Top→Bottom 순서로(깊이(level) 순으로) 리스트에 저장해 반환
    ... # 그 외 함수는 위 함수들의 구현에 내부적으로 사용되었거나, 본 과제와 직접적으로 연관 없음

    def rangeCount(self, lo, hi): # lo~hi 범위에 속한 key의 개수 반환
    def rangeSearch(self, lo, hi): # lo~hi 범위에 속한 key를 정렬된 순서로 리스트에 저장해 반환
```

LLRB 클래스의 활용 예

```
bst2 = LLRB()
bst2.put("S",1)
bst2.put("E",2)
bst2.put("Y",3)
bst2.put("A",4)
bst2.put("R",5)
bst2.put("C",6)
bst2.put("H",7)
bst2.put("M",8)
bst2.put("L",9)
bst2.put("P",10)
print(bst2.rank("H"))
print(bst2.select(4))
print("level order", bst2.levelorder())
print("inorder",bst2.inorder())
print("range count", bst2.rangeCount("F", "T"))
print("range search", bst2.rangeSearch("F", "T"))
print("range count", bst2.rangeCount("B", "I"))
print("range search", bst2.rangeSearch("B", "I"))
print("range count", bst2.rangeCount("C", "H"))
print("range search", bst2.rangeSearch("C", "H"))
print("range count", bst2.rangeCount("J", "R"))
print("range search", bst2.rangeSearch("J", "R"))
```



실행 결과

3

L

level order ['M', 'E', 'S', 'C', 'L', 'R', 'Y', 'A', 'H', 'P']

inorder ['A', 'C', 'E', 'H', 'L', 'M', 'P', 'R', 'S', 'Y']

range count 6

range search ['H', 'L', 'M', 'P', 'R', 'S']

range count 3

range search ['C', 'E', 'H']

range count 3

range search ['C', 'E', 'H']

range count 4

range search ['L', 'M', 'P', 'R']

구현할 API 정리: sweepLine()

```
# 구현해야 할 함수. Segment 객체의 리스트를 입력으로 받아, 교차하는 Segment 쌍(2-tuple)의 리스트를 반환
def sweepLine(segments)
    # queue.PriorityQueue 클래스 사용해 minPQ 객체 생성
    #
    # minPQ에 담은 원소는 segments 리스트에 담긴 Segment 객체이며,
    # 특히 x축을 기준으로 정렬되도록 아래와 같은 2-tuple 형태로 담을 것
    # (Segment 객체의 x 좌표, Segment 객체)
    #
    # 그 후에는 while loop을 사용해 다음을 반복
    #     minPQ에서 가장 작은 원소를 get(). 이 원소는 2-tuple일 것이며 e라 하겠음
    #     If e가 수평선이라면: e의 y 좌표를 LLRB에 추가하거나 혹은 제거
    #     If e가 수직선이라면: e와 교차하는 수평선이 LLRB에 있다면 교차점을 결과 리스트에 추가
    #
    # 결과 리스트 반환
```


Python 기본 제공 PQ 클래스:

minPQ이나, **tuple**로 **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

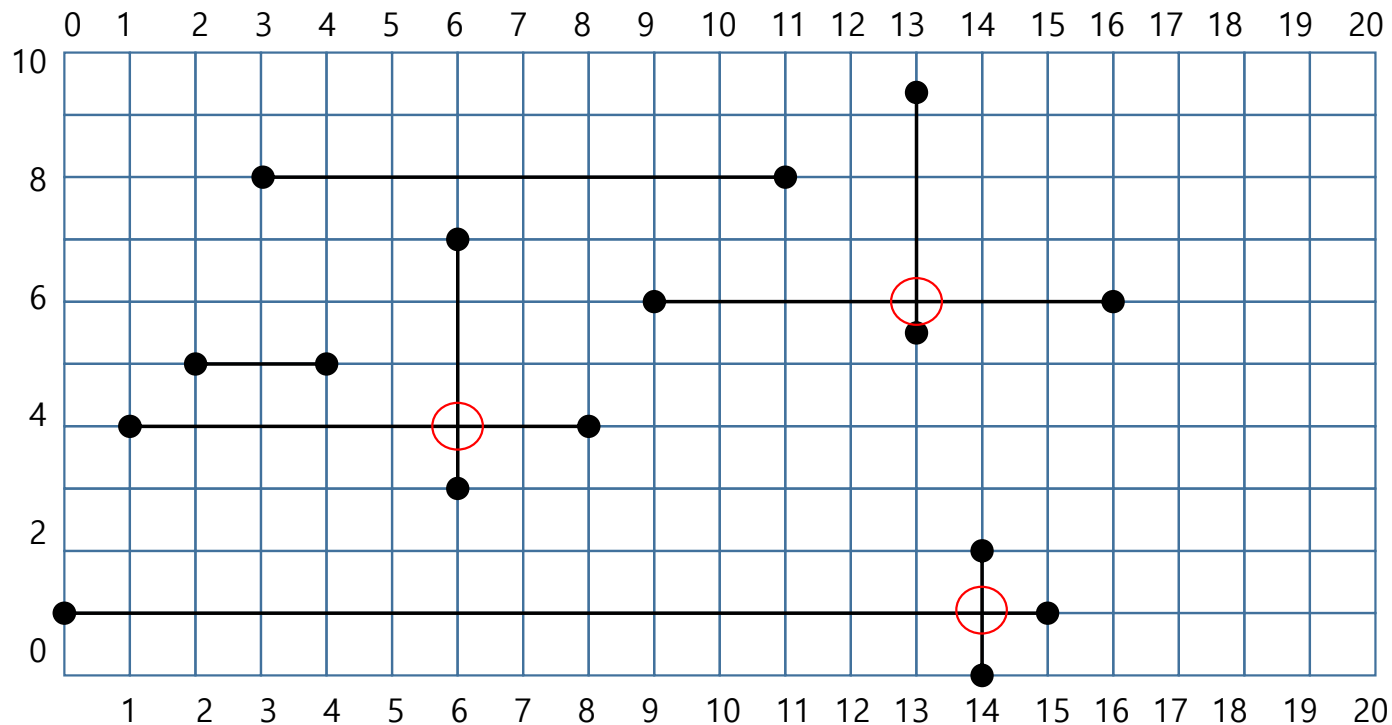
Tuple 추가할 때는 tuple
을 괄호()로 싸는 것 유의

프로그램 입출력 예

```
intersections = sweepLine([Segment(0,1,15,1), Segment(14,0,14,2), Segment(1,4,8,4), Segment(6,3,6,7),\
    Segment(2,5,4,5), Segment(3,8,11,8), Segment(9,6,16,6), Segment(13,5.5,13,9.5)])
print(intersections)
```

실행 결과: 3개 교차점

[((1,4)--(8,4), (6,3)--(6,7)), ((9,6)--(16,6), (13,5.5)--(13,9.5)), ((0,1)--(15,1), (14,0)--(14,2))]

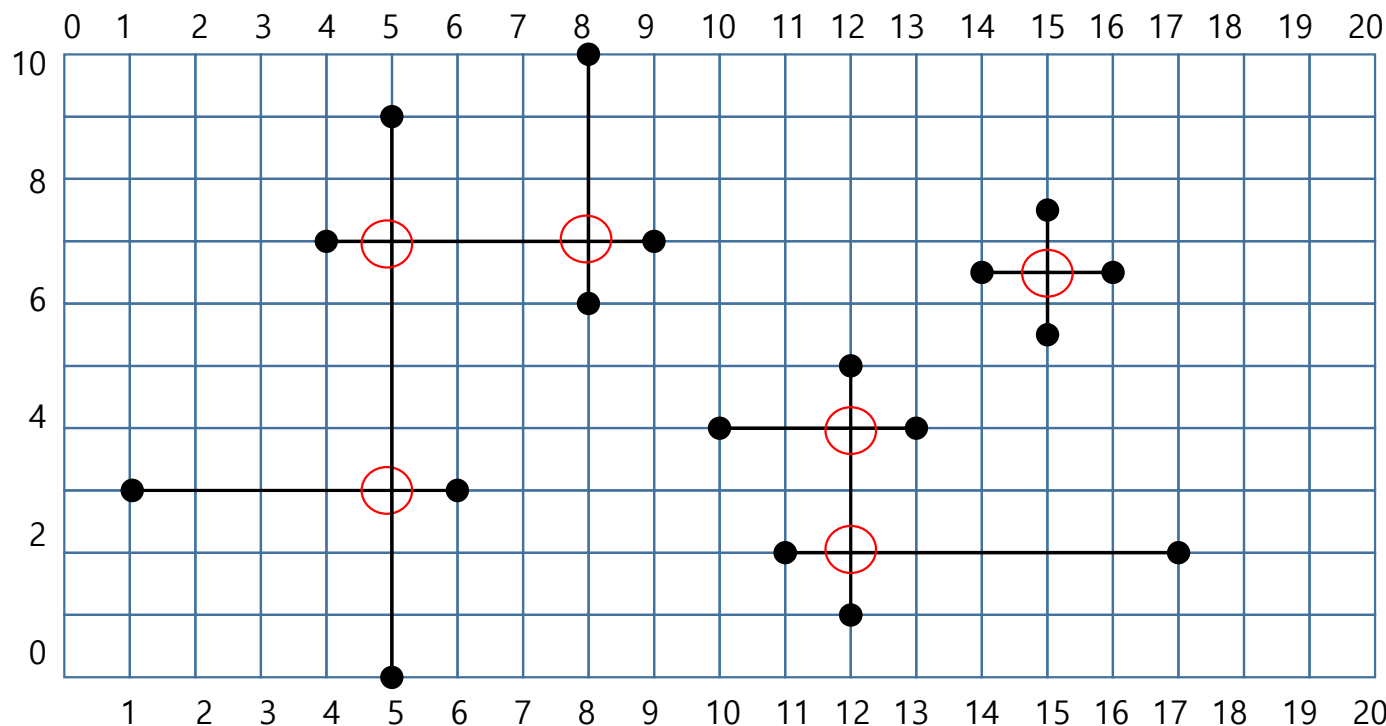


프로그램 입출력 예

```
intersections = sweepLine([Segment(1,3,6,3), Segment(5,0,5,9), Segment(4,7,9,7), Segment(8,6,8,10),\
    Segment(10,4,13,4), Segment(11,2,17,2), Segment(12,1,12,5), Segment(15,5.5,15,7.5), Segment(14,6.5,16,6.5)])
print(intersections)
```

실행 결과: 6개 교차점

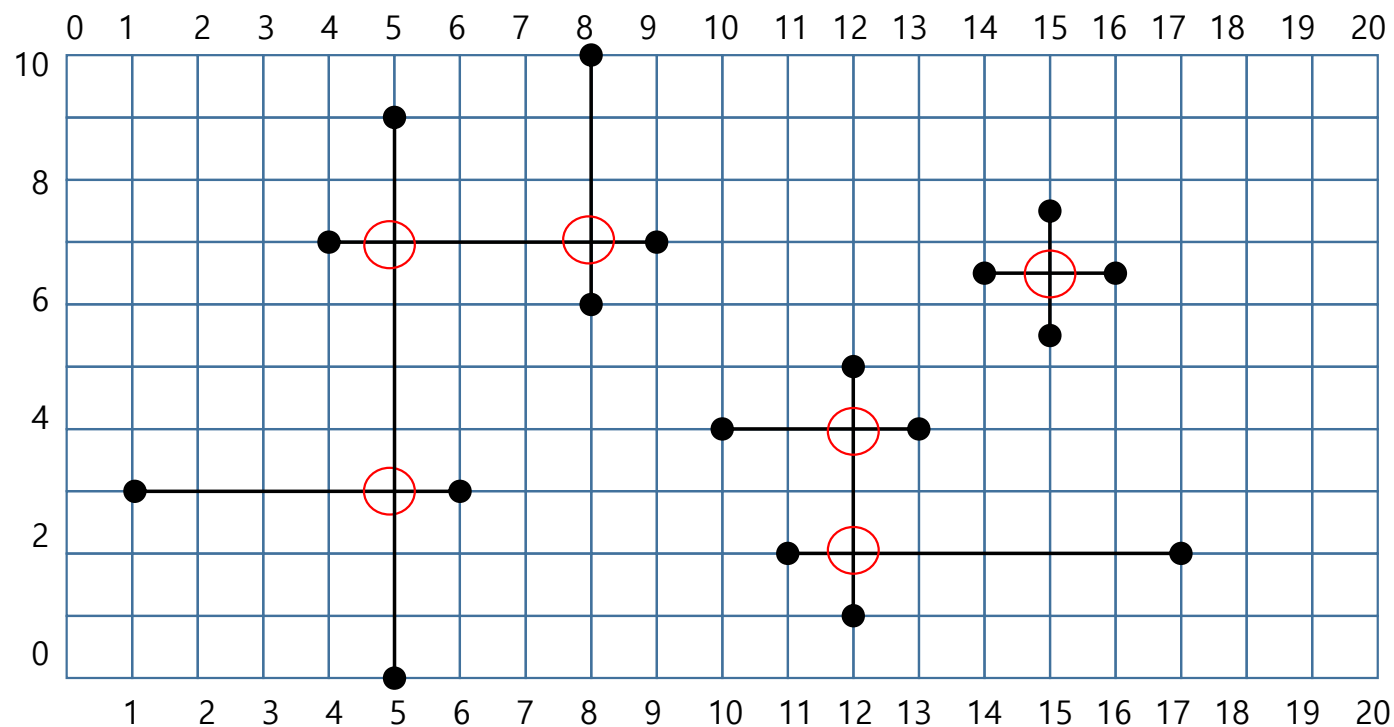
```
[((1,3)--(6,3), (5,0)--(5,9)), ((4,7)--(9,7), (5,0)--(5,9)), ((4,7)--(9,7), (8,6)--(8,10)), ((11,2)--(17,2), (12,1)--(12,5)), ((10,4)--(13,4), (12,1)--(12,5)), ((14,6.5)--(16,6.5), (15,5.5)--(15,7.5))]
```



그 외 유의사항: 프로그램 입력 조건

- x, y좌표는 임의의 실수로, 모두 서로 다름 (같은 경우는 입력으로 들어오지 않음)

```
intersections = sweepLine([Segment(1,3,6,3), Segment(5,0,5,9), Segment(4,7,9,7), Segment(8,6,8,10),\
    Segment(10,4,13,4), Segment(11,2,17,2), Segment(12,1,12,5), Segment(15,5.5,15,7.5), Segment(14,6.5,16,6.5)])
```

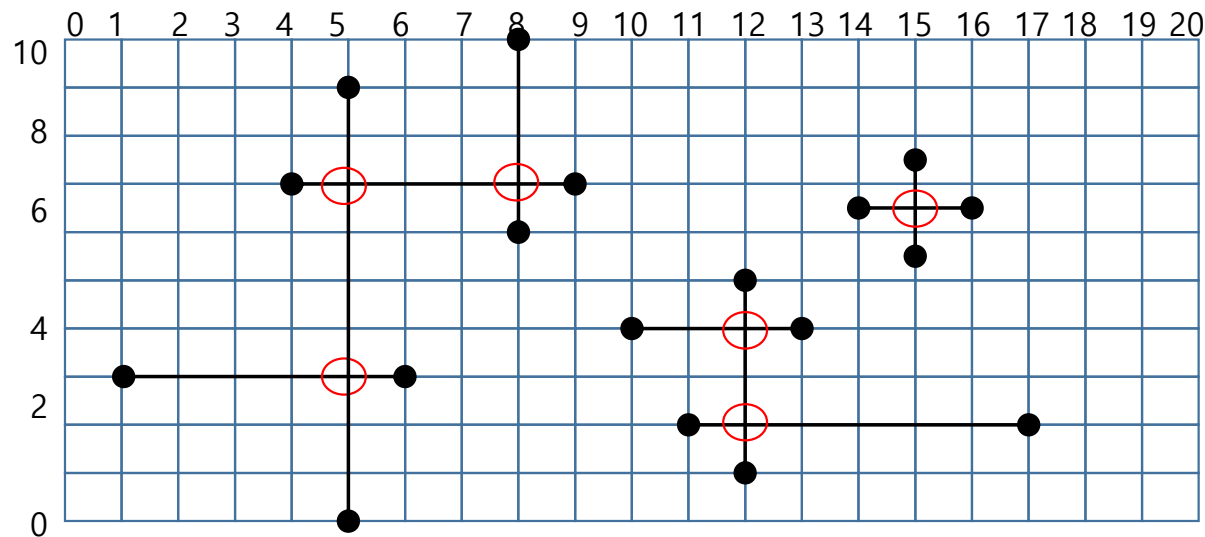


그 외 유의사항: sweepLine()이 반환하는 교차하는 Segment 쌍(2-tuple)의 리스트

- 각 Segment 쌍(2-tuple)은 (수평선 객체, 수직선 객체) 순으로 담기
- 교점의 x좌표가 작은 경우부터 넣기
 - sweep-line 알고리즘을 x좌표 작은 순서로(왼쪽→오른쪽 방향으로) 적용하면 자연스럽게 이렇게 됨
- x좌표 같은 교점 여럿이면, y좌표 작은 것부터 넣기
 - range search가 정렬된 순서로 결과 반환하므로, 그 순서대로 사용하면 자연스럽게 이렇게 됨

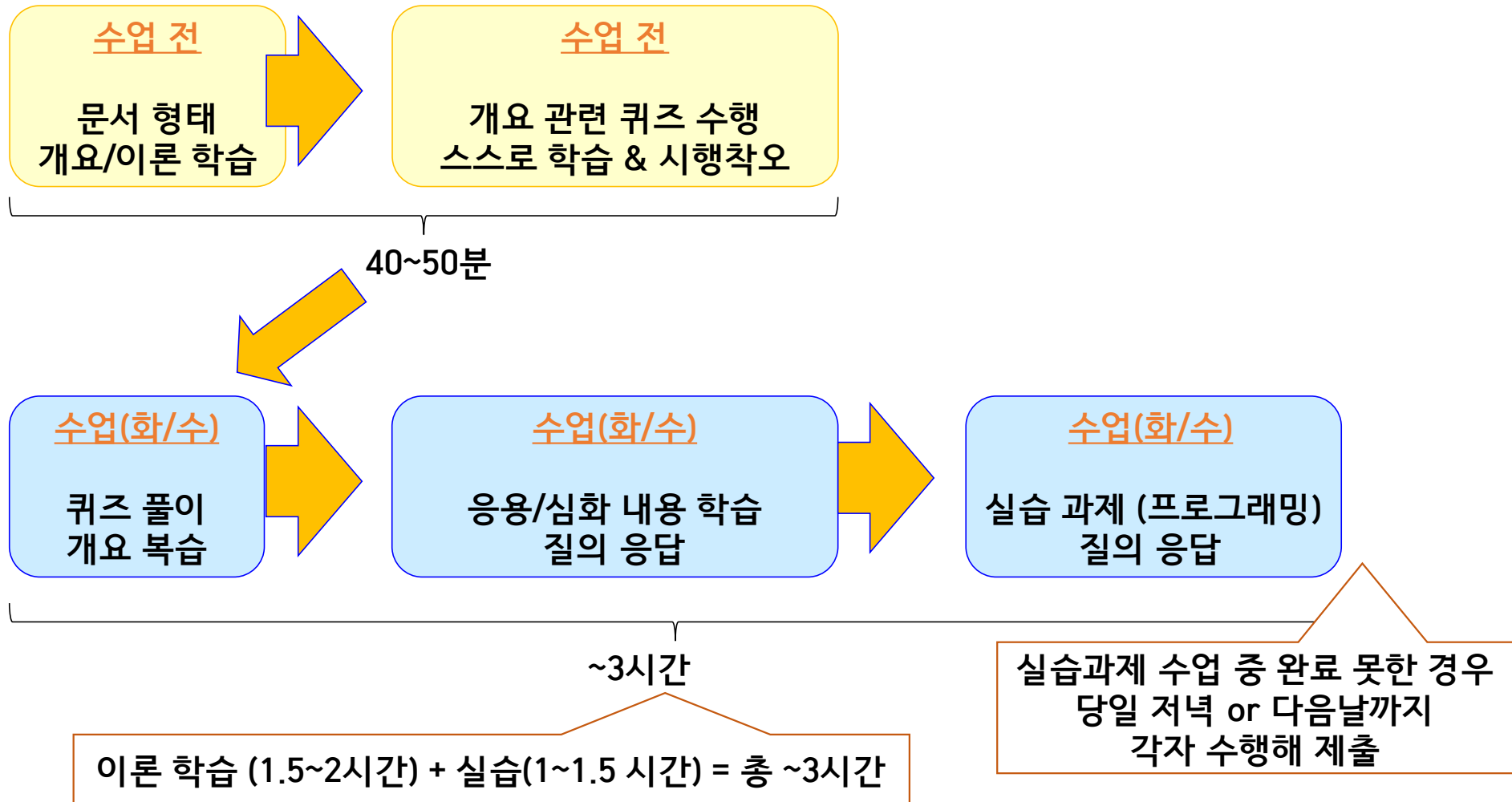
실행 결과: 6개 교차점

[((1,3)--(6,3), (5,0)--(5,9)), ((4,7)--(9,7), (5,0)--(5,9)), ((4,7)--(9,7), (8,6)--(8,10)), ((11,2)--(17,2), (12,1)--(12,5)), ((10,4)--(13,4), (12,1)--(12,5)), ((14,6.5)--(16,6.5), (15,5.5)--(15,7.5))]





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: Symbol Table 무엇이고, 어떻게 활용하며, 어떤 경우 활용하는지 이해

- Symbol Table: (key, value) 쌍 저장하며, key를 검색어로 value 찾는 것이 주 기능
- put(), get() 및 get()을 확장한 여러 작업을 $\sim \log N$ 시간에 효율적으로 구현하기 위해 **Binary Search Tree 구조** 사용 (왼쪽 자식 < 노드 < 오른쪽 자식)
 - Tree 구조 + 왼쪽<가운데<오른쪽 조건 → Key 값에 따른 효율적인 추가, 탐색 가능하게 함
- Symbol Table에는 put(), get() 뿐 아니라, **get() 확장한 다양한 탐색 작업을 효율적으로 수행 가능**
- Range count, range search, nearest neighbor, floor, ceiling, rank, select, min, max, ...