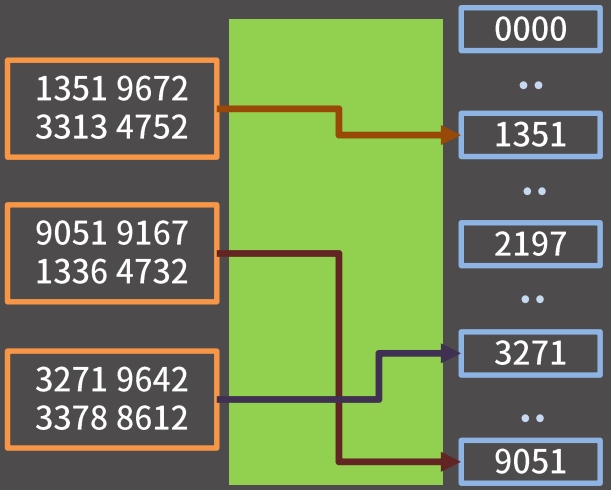
실전 알고리즘 0x0D강
해쉬, 이진 검색 트리, 힙

BaaaaaaaaaaaaaaaarkingDog

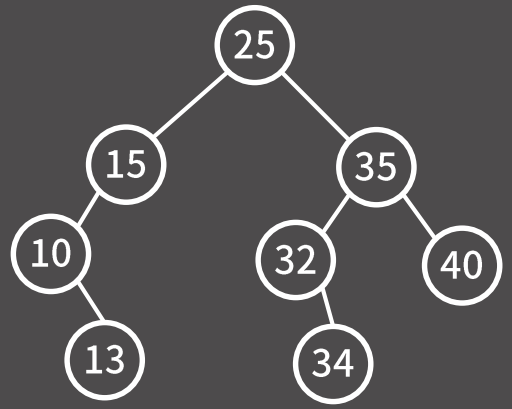
목차

0x00 프롤로그

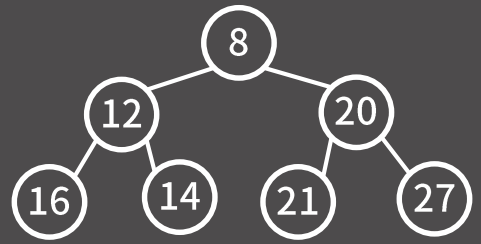
- 학부 자료구조에서의 3대장..



해쉬(Hash)



이진 검색 트리
(Binary Search Tree)



힙(Heap)

0x00 해쉬(Hash) – 정의

N명의 사람에 대해 각 사람이 가진 16자리의 카드 번호 데이터베이스가 주어져 있다. 이 때 카드 번호에 대응되는 사람을 어떻게 빠르게 찾을 것인가?

- 배열에 (사람, 카드 번호)를 넣었을 때 카드 번호에 대응되는 사람을 찾는데 필요한 시간복잡도는 얼마일까?

0	1	2	3	4	5	6	7	8	9
1351 9672 3313 4752 Kim	9811 5626 9133 4492 Lee	6166 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

0x00 해쉬(Hash) – 정의

N명의 사람에 대해 각 사람이 가진 16자리의 카드 번호 데이터베이스가 주어져 있다. 이 때 카드 번호에 대응되는 사람을 어떻게 빠르게 찾을 것인가?

- 카드번호가 “4941 9004 3383 4894”인 사람은 누구일까?

X ↓

0	1	2	3	4	5	6	7	8	9
1351 9672 3313 4752 Kim	9811 5626 9133 4492 Lee	6166 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

0x00 해쉬(Hash) – 정의

N명의 사람에 대해 각 사람이 가진 16자리의 카드 번호 데이터베이스가 주어져 있다. 이 때 카드 번호에 대응되는 사람을 어떻게 빠르게 찾을 것인가?

- 카드번호가 “4941 9004 3383 4894”인 사람은 누구일까?

X

0	1	2	3	4	5	6	7	8	9
1351 9672 3313 4752 Kim	9811 5626 9133 4492 Lee	6166 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

0x00 해쉬(Hash) – 정의

N명의 사람에 대해 각 사람이 가진 16자리의 카드 번호 데이터베이스가 주어져 있다. 이 때 카드 번호에 대응되는 사람을 어떻게 빠르게 찾을 것인가?

- 카드번호가 “4941 9004 3383 4894”인 사람은 누구일까? – 시간복잡도 O(N)

0	1	2	3	4	5	6	O ↓ 7	8	9
1351 9672 3313 4752 Kim	9811 5626 9133 4492 Lee	6166 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

0x00 해쉬(Hash) – 정의

- 만약 카드번호가 4자리였다면 더 효율적인 방법이 있지 않았을까?

0	1	2	3	4	5	6	7	8	9
1351 Kim	9811 Lee	6166 Choi	4611 Son	9748 Park	1237 Ma	9571 Bae	4941 Wang	4411 Cheon	0098 Ko

↓

0000 0001 ..... 0098 0099 ..... 1237 1238 1239 ..... 1351 ..... 4941 .....(후략)

x	x	x	x	Ko	x	x	Ma	x	x	x	Kim	x	x	x	Wang	.	.	.	.

- 카드 번호를 인덱스로 하는 테이블을 만들면 O(1)에 탐색이 가능하다.

0x00 해쉬(Hash) – 정의

- 하지만 카드번호가 16자리인 이상, 16자리의 테이블을 만드는 것은 현실적으로 불가능하다. (10^{16}개의 int 배열 = 40PB = 4×10^6 GB)

0	1	2	3	4	5	6	7	8	9
1351 9672 3313 4752 Kim	9811 5626 9133 4492 Lee	6166 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0001 0002 0003 0004 0005 0006 0007 0008

(공간의 제약으로 인해 불가능)

. . . (후략)

0x00 해쉬(Hash) – 정의

- 그렇다면 그냥 16자리를 전부 인덱스로 사용하는 대신 앞의 4자리를 가지고 만들어도 되지 않을까?

0	1	2	3	4	5	6	7	8	9
1351 9672 3313 4752 Kim	9811 5626 9133 4492 Lee	6166 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

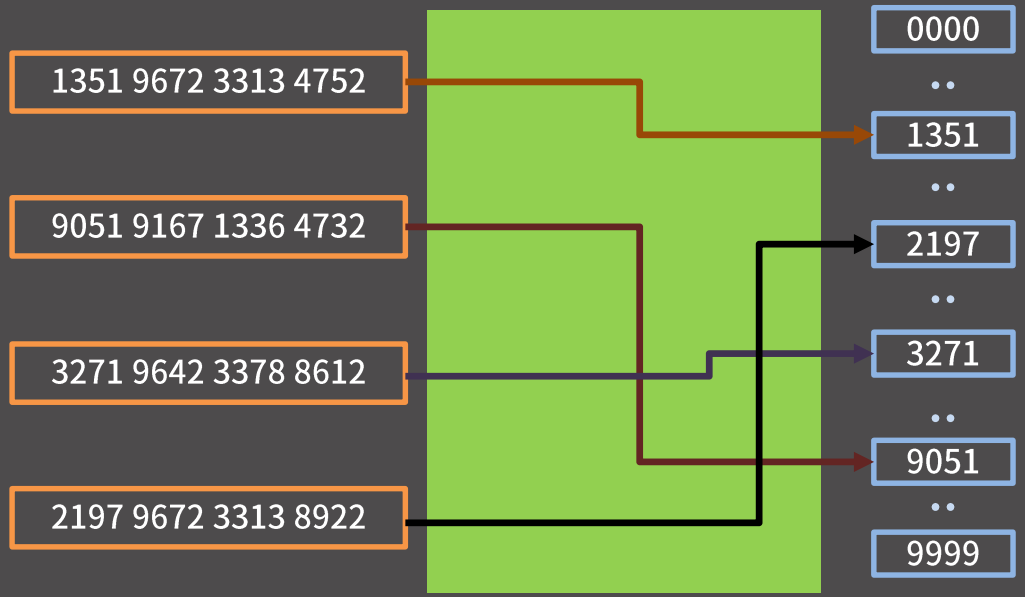
↓

0000 0001 ..... 0098 0099 ..... 1237 1238 1239 ..... 1351 ..... 4941 .....(후략)

x	x	x	x	Ko	x	x	Ma	x	x	x	Kim	x	x	x	Wang	.	.	.	.

0x00 해쉬(Hash) – 정의

- 해쉬함수 : 임의의 길이의 데이터를 고정된 길이의 데이터로 매핑하는 함수



0x00 해쉬(Hash) – 정의

- 앞의 예시에서 해쉬 함수는 16자리를 입력받아 앞 4자리를 반환하는 함수이고, 해쉬 함수를 이용해 만든 테이블을 **해쉬 테이블**이라고 한다.

0	1	2	3	4	5	6	7	8	9
1351 9672 3313 4752 Kim	9811 5626 9133 4492 Lee	6166 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

↓

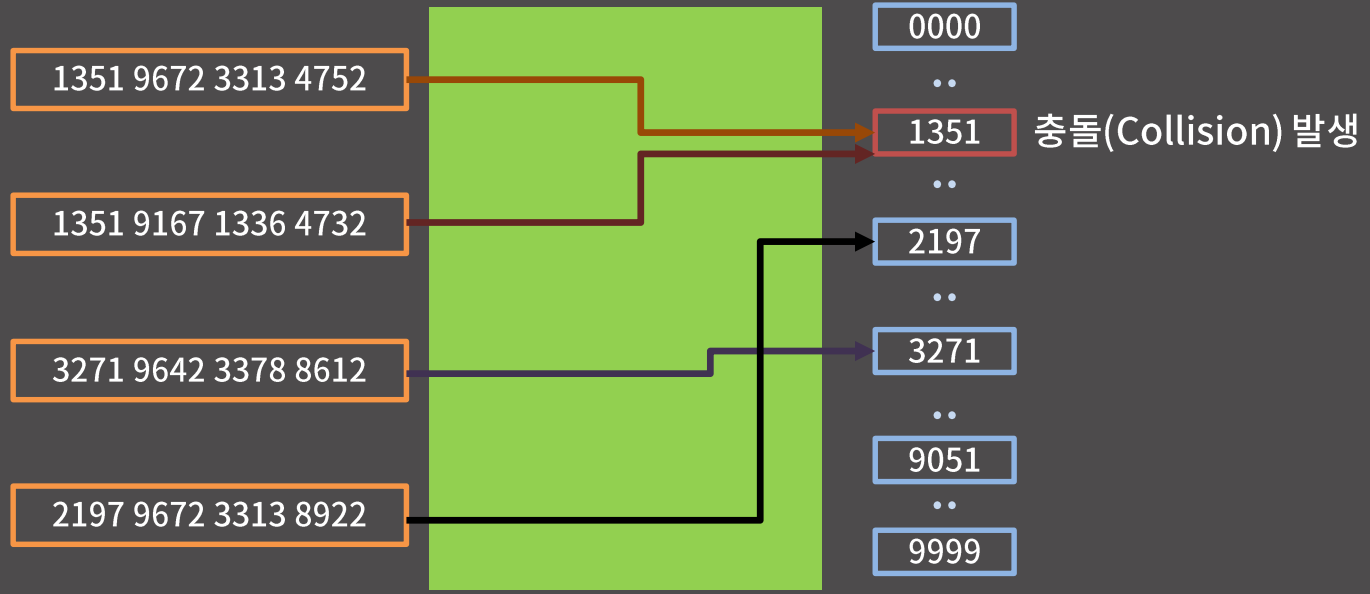
0000 0001 ..... 0098 0099 ..... 1237 1238 1239 ..... 1351 ..... 4941 .....(후략)

x	x	x	x	Ko	x	x	Ma	x	x	x	Kim	x	x	x	Wang	.	.	.	.

해쉬 테이블

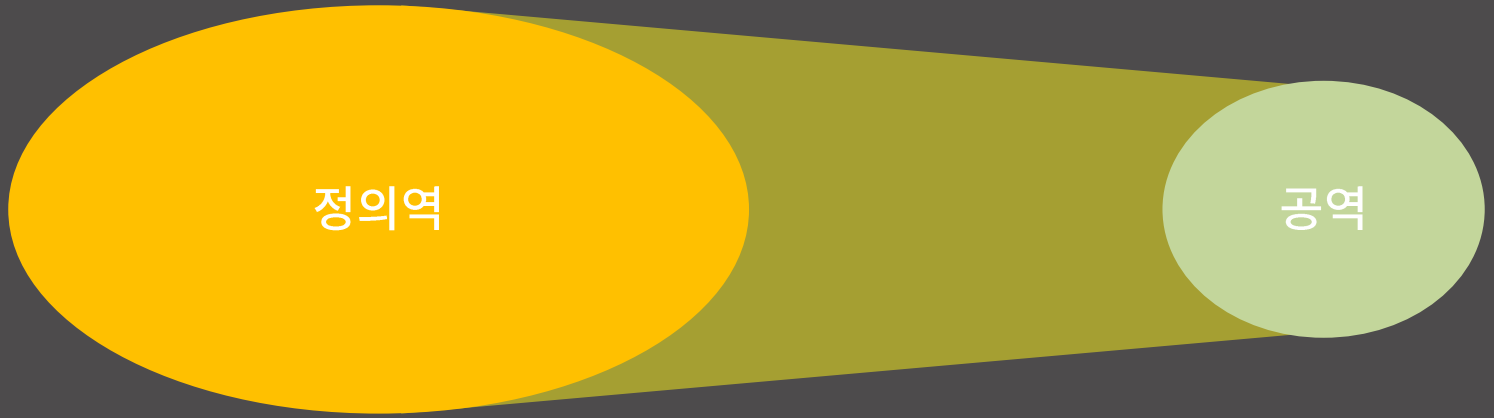
0x00 해쉬(Hash) – 충돌(Collision)

- 해쉬함수에는 두 입력값에 대해 출력값이 동일한, 충돌(Collision)이라는 치명적인 문제가 있다.



0x00 해쉬(Hash) – 충돌(Collision)

- 해쉬 함수의 특성상 충돌은 발생할 수 밖에 없다.



0x00 해쉬(Hash) – 충돌(Collision)

- 충돌을 회피하기 위한 방법(Collision Resolution)에는 Open Addressing, Chaining 등이 있다. 먼저 Open Addressing을 알아보자.

0	1	2	3	4	5	6	7	8	9
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

↓

0000 0001 ..... 0098 0099 ..... 1237 1238 1239 ..... 1351 ..... 4941 .....(후략)

x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

0x00 해쉬(Hash) – 충돌(Collision)

- Open Addressing은 충돌이 발생할 시 원소를 저장하는 인덱스를 바꾸는 충돌 회피 방법이다. 우선 Kim을 테이블에 기록하자.

0	1	2	3	4	5	6	7	8	9
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

↓

0000	0001	.....	0098	0099	.....	1237	1238	1239	.....	1351	.....	4941	.....(후략)

x	x	x	x	x	x	x	1237 9672 3313 4752 Kim	x	x	x	x	x	x	x	x	x	x	x	x

0x00 해쉬(Hash) – 충돌(Collision)

- Lee를 테이블에 기록하고 싶은데 1237번지에는 이미 Kim이 들어있다.

0	1	2	3	4	5	6	7	8	9
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

↓

0000	0001	.....	0098	0099	.....	1237	1238	1239	.....	1351	.....	4941	.....(후략)

x	x	x	x	x	x	x	1237 9672 3313 4752 Kim	x	x	x	x	x	x	x	x	x	x	x	x

??

0x00 해쉬(Hash) – 충돌(Collision)

- 1237번지에 기록하는 대신 한 칸 오른쪽에 기록한다.

0	1	2	3	4	5	6	7	8	9
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

↓

0000	0001	.....	0098	0099	.....	1237	1238	1239	.....	1351	.....	4941	.....(후략)

x	x	x	x	x	x	x	1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	x	x	x	x	x	x	x	x	x	x	x

0x00 해쉬(Hash) – 충돌(Collision)

- Choi를 테이블에 기록하고 싶은데 1237번지에는 이미 Kim이 들어있다.

0	1	2	3	4	5	6	7	8	9
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

↓

0000	0001	.....	0098	0099	.....	1237	1238	1239	.....	1351	.....	4941	.....(후략)

x	x	x	x	x	x	x	1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	x	x	x	x	x	x	x	x	x	x	x

0x00 해쉬(Hash) – 충돌(Collision)

- 한 칸 오른쪽으로 이동해 1238번지를 봐도 여전히 들어있다.

0	1	2	3	4	5	6	7	8	9
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

↓

0000	0001	.....	0098	0099	.....	1237	1238	1239	.....	1351	.....	4941	.....(후략)

x	x	x	x	x	x	x	1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	x	x	x	x	x	x	x	x	x	x	x

??

0x00 해쉬(Hash) – 충돌(Collision)

- 한 칸 더 오른쪽으로 이동해 1239번지로 가면 그 곳은 비어있으니 1239번지를 사용한다.(이후 과정은 후략)

0	1	2	3	4	5	6	7	8	9
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma	9571 9004 7483 5194 Bae	4941 9004 3383 4894 Wang	4411 9004 3523 4121 Cheon	0098 7890 3753 4752 Ko

↓

0000	0001	.....	0098	0099	.....	1237	1238	1239	.....	1351	.....	4941	.....(후략)

x	x	x	x	x	x	x	1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	x	x	x	x	x	x	x	x	x	x

0x00 해쉬(Hash) – 충돌(Collision)

- Open Addressing에서 왜 해쉬 테이블에 이름만 써놓는 것이 아니라 카드번호를 같이 써놓는걸까?

0000	0001	.....	0098	0099	.....	1237	1238	1239	.....	1351	.....	4941	.....(후략)
x	x	x	x	x	x	x	1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	x	x	x	x	x	x	x	x	x	x

VS

0000	0001	.....	0098	0099	.....	1237	1238	1239	.....	1351	.....	4941	.....(후략)
x	x	x	x	x	x	x	Kim	Lee	Choi	x	x	x	x	x	x	x	x	x	x

0x00 해쉬(Hash) – 충돌(Collision)

- Open Addressing에서 해당 칸이 이미 차있을 경우 지금처럼 1칸씩 뛰는 것을 Linear probing이라고 한다.
- 이외에도 어떤 방식으로 뛰냐에 따라 Quadratic probing, Double hashing 등의 방식이 있다.

0x00 해쉬(Hash) – 충돌(Collision)

- 충돌을 회피하기 위한 방법(Collision Resolution)중 두 번째인 Chaining을 알아보자. Chaining은 해쉬 테이블에서 각 인덱스가 원소 1개만을 담는 것이 아니라, Linked List 구조로 여러 원소를 담고 있는 방식을 의미한다.

0x00 해쉬(Hash) – 충돌(Collision)

0	1	2	3	4	5
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma

0000 0001 ..... 0098 0099 ..... 1237 1238 1239 ..... 1351 .....4611.....9748....(후략)

x	x	x	x	x	x	x	.	x	x	x	x	x	x	x	x	x	x	x	x

1237
9672
3313
4752
Kim

0x00 해쉬(Hash) – 충돌(Collision)

0	1	2	3	4	5
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma

0000 0001 ..... 0098 0099 ..... 1237 1238 1239 ..... 1351 .....4611.....9748....(후략)

x	x	x	x	x	x	x	.	x	x	x	x	x	x	x	x	x	x	x	x

1237 5626 9133 4492 Lee	→	1237 9672 3313 4752 Kim

0x00 해쉬(Hash) – 충돌(Collision)

0	1	2	3	4	5
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma

0000 0001 ..... 0098 0099 ..... 1237 1238 1239 ..... 1351 .....4611.....9748....(후략)

x	x	x	x	x	x	x	.	x	x	x	x	x	x	x	x	x	x	x	x

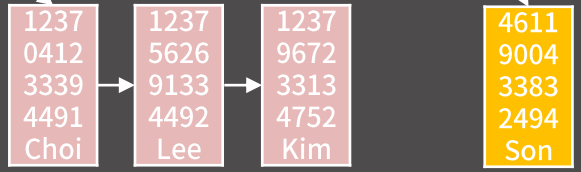
1237 0412 3339 4491 Choi	→	1237 5626 9133 4492 Lee	→	1237 9672 3313 4752 Kim

0x00 해쉬(Hash) – 충돌(Collision)

0	1	2	3	4	5
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma

0000 0001 ..... 0098 0099 ..... 1237 1238 1239 ..... 1351 .....4611.....9748....(후략)

x	x	x	x	x	x	x	.	x	x	x	x	x	.	x	x	x	x	x	x



0x00 해쉬(Hash) – 충돌(Collision)

0	1	2	3	4	5
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma

0000 0001 ..... 0098 0099 ..... 1237 1238 1239 ..... 1351 .....4611.....9748....(후략)

x	x	x	x	x	x	x	.	x	x	x	x	x	.	x	x	x	x	.	x

1237
0412
3339
4491
Choi → 1237
5626
9133
4492
Lee → 1237
9672
3313
4752
Kim

4611
9004
3383
2494
Son

9748
1584
9813
5678
Park

0x00 해쉬(Hash) – 충돌(Collision)

0	1	2	3	4	5
1237 9672 3313 4752 Kim	1237 5626 9133 4492 Lee	1237 0412 3339 4491 Choi	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park	1237 9223 3893 8186 Ma

0000 0001 ..... 0098 0099 ..... 1237 1238 1239 ..... 1351 .....4611.....9748....(후략)

x	x	x	x	x	x	x	.	x	x	x	x	x	.	x	x	x	x	.	x

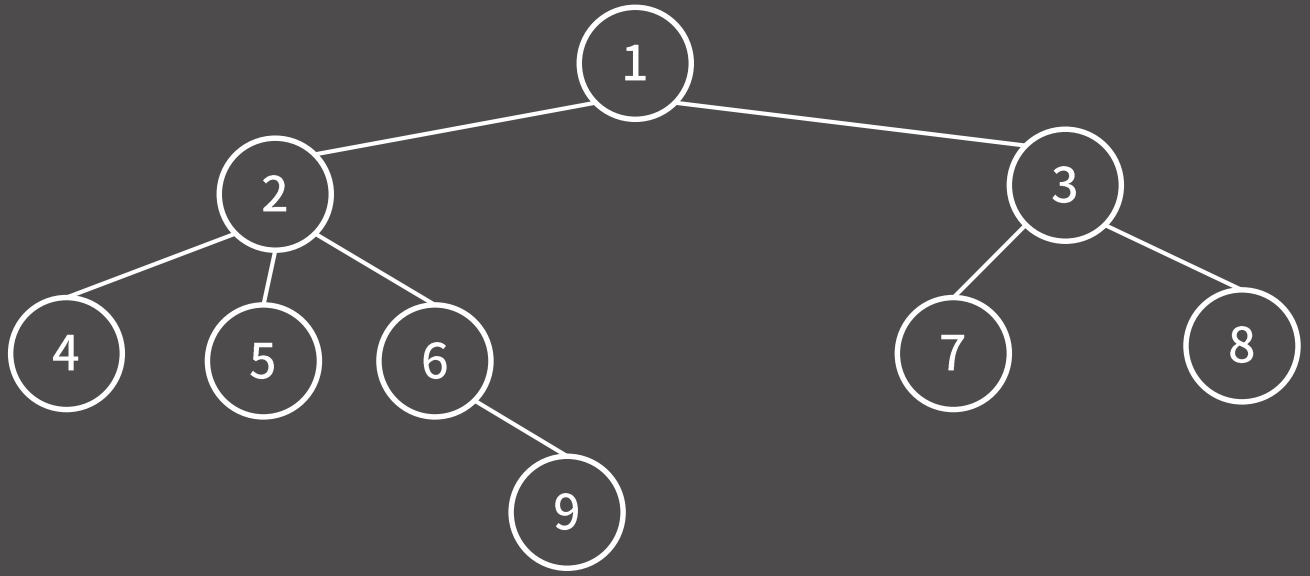
1237 9223 3893 8186 Ma	1237 0412 3339 4491 Choi	1237 5626 9133 4492 Lee	1237 9672 3313 4752 Kim	4611 9004 3383 2494 Son	9748 1584 9813 5678 Park

0x00 해쉬(Hash) – 충돌(Collision)

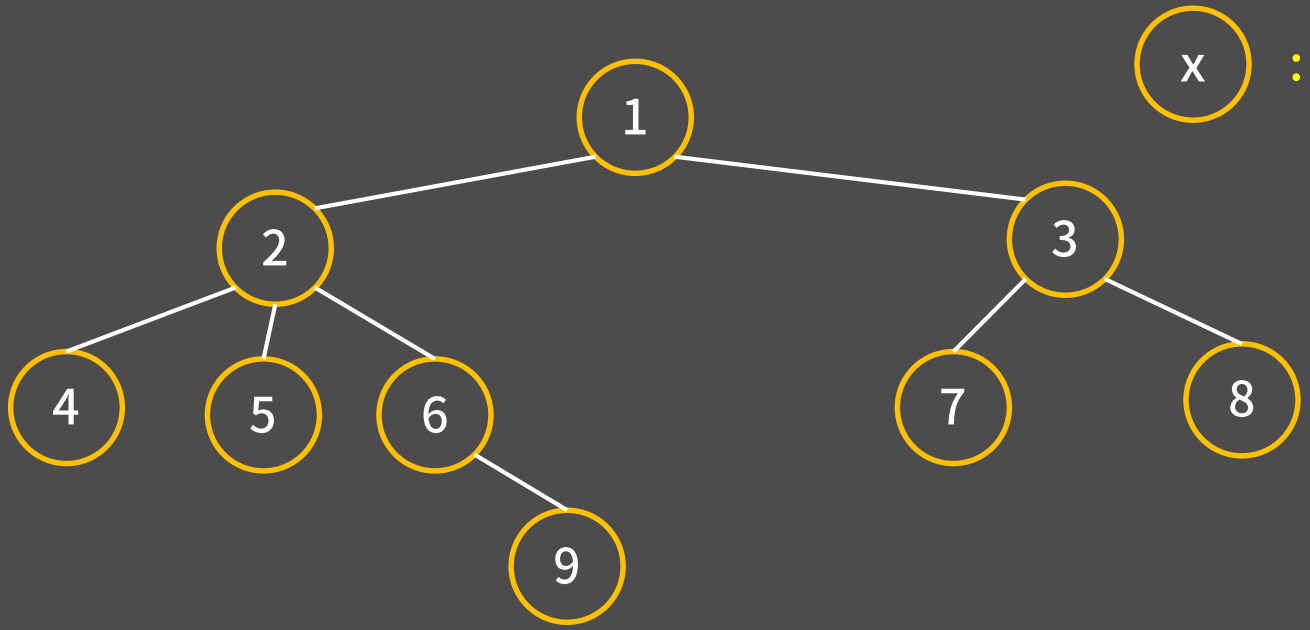
- 해쉬에서 삽입, 삭제, 검색은 모두 O(1)이지만 충돌이 빈번히 발생할수록 실제 시간 복잡도는 나빠진다.
- 코딩테스트에서 해쉬 테이블을 구현해야 할 일은 거의 없다.

0x01 이진 검색 트리(Binary Search Tree) - 정의

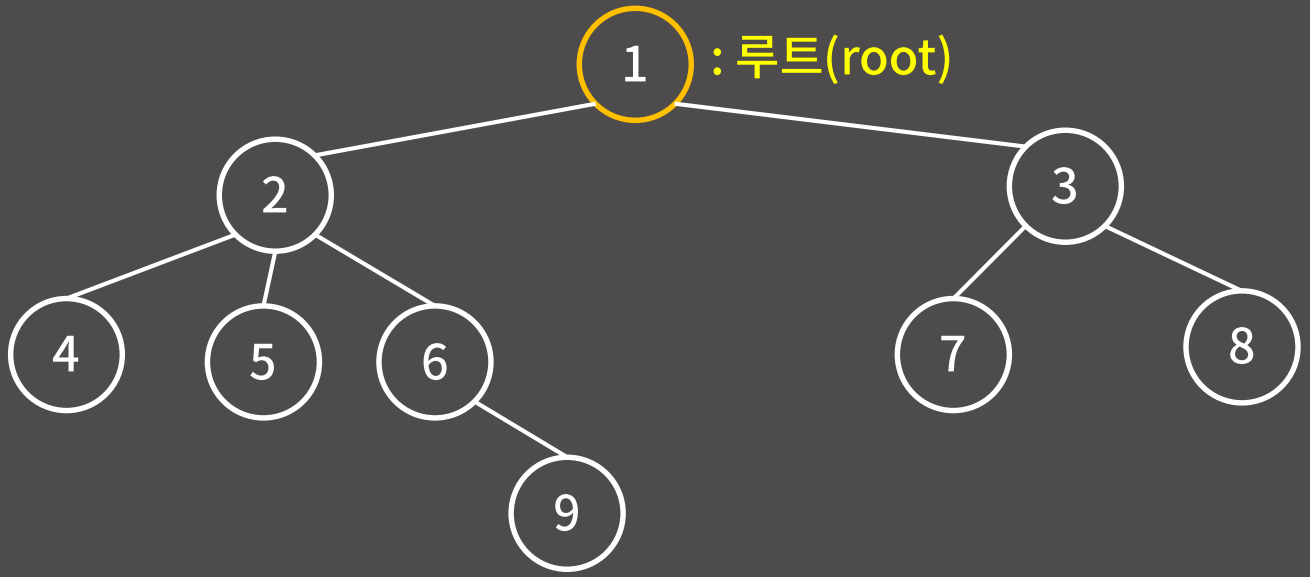
- 이진 검색 트리 : 특별한 성질을 만족하는 트리. 트리란 무엇인가?



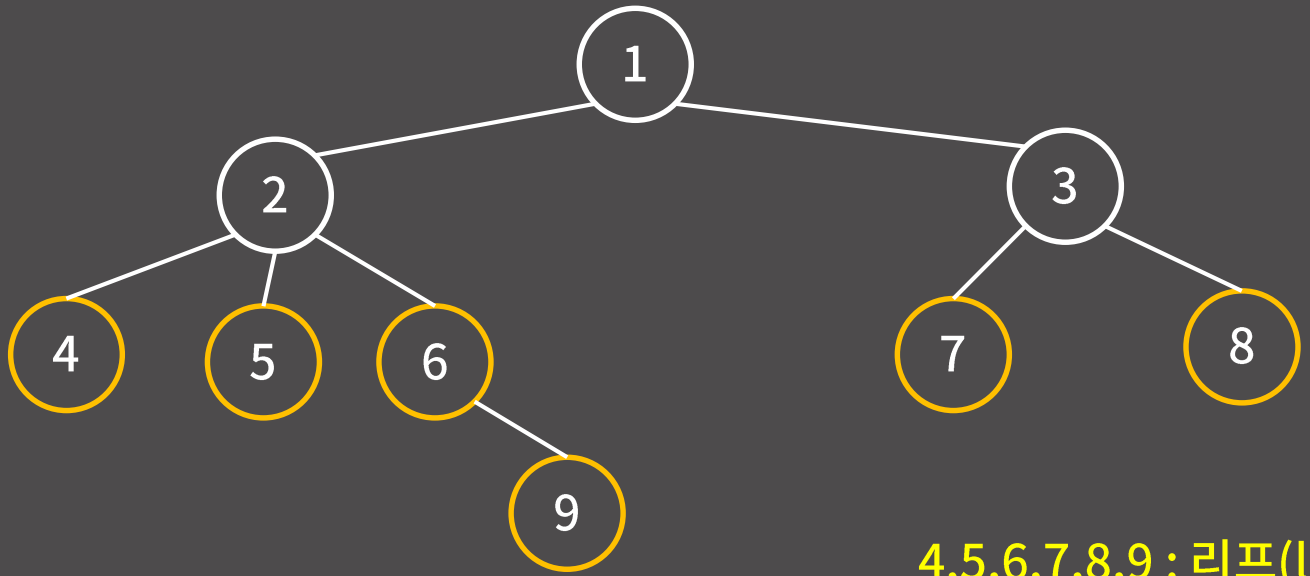
0x01 이진 검색 트리(Binary Search Tree) - 정의



0x01 이진 검색 트리(Binary Search Tree) - 정의

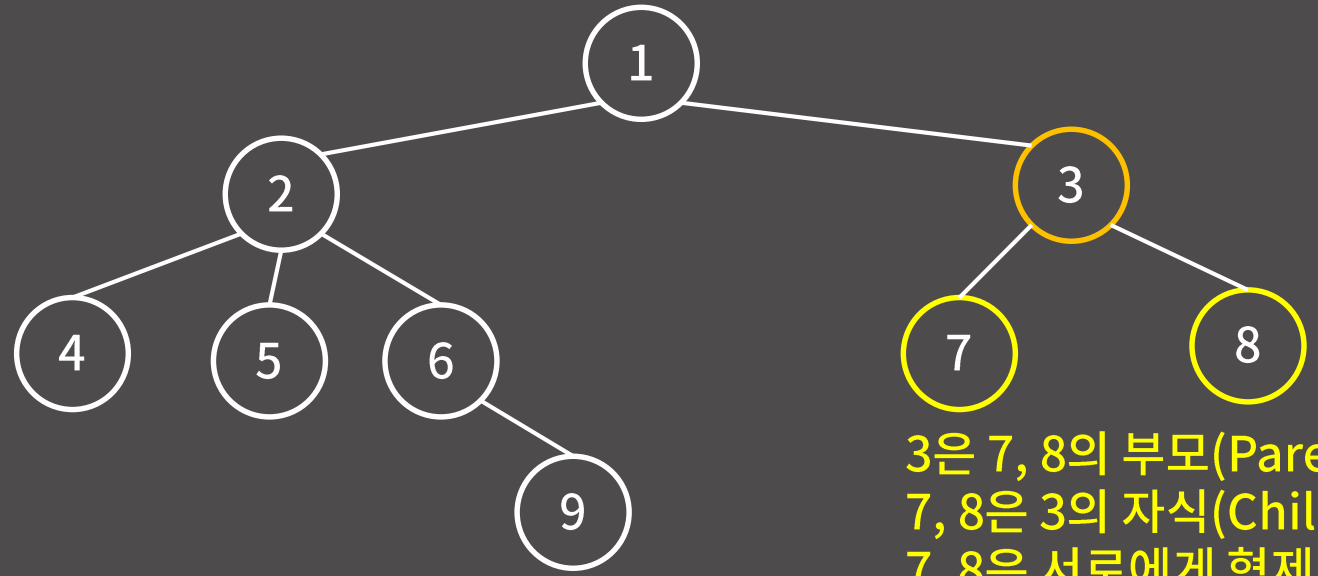


0x01 이진 검색 트리(Binary Search Tree) - 정의



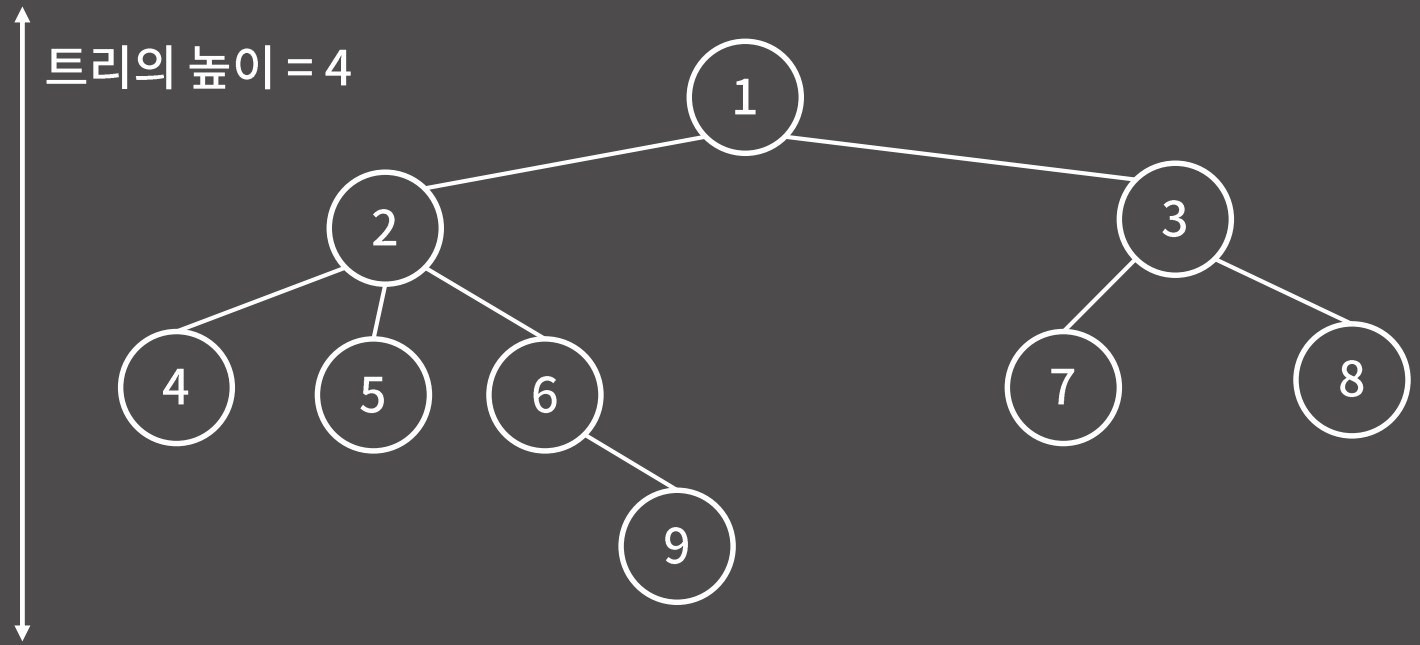
4,5,6,7,8,9 : 리프(leaf)

0x01 이진 검색 트리(Binary Search Tree) - 정의



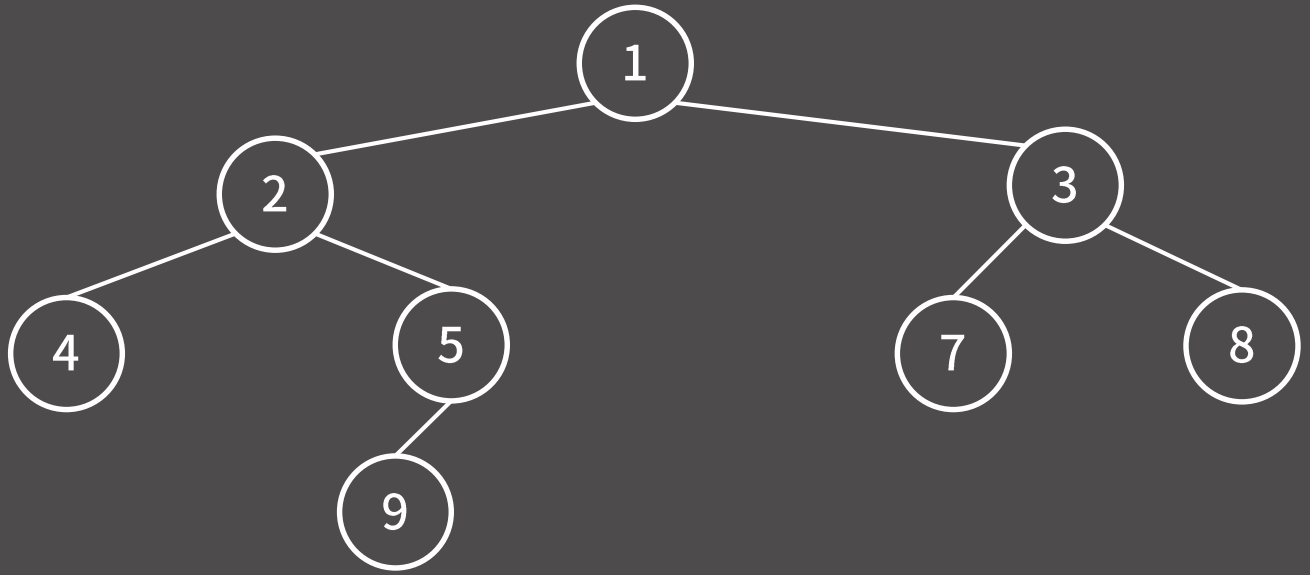
3은 7, 8의 부모(Parent)
7, 8은 3의 자식(Child)
7, 8은 서로에게 형제(Sibling)

0x01 이진 검색 트리(Binary Search Tree) - 정의



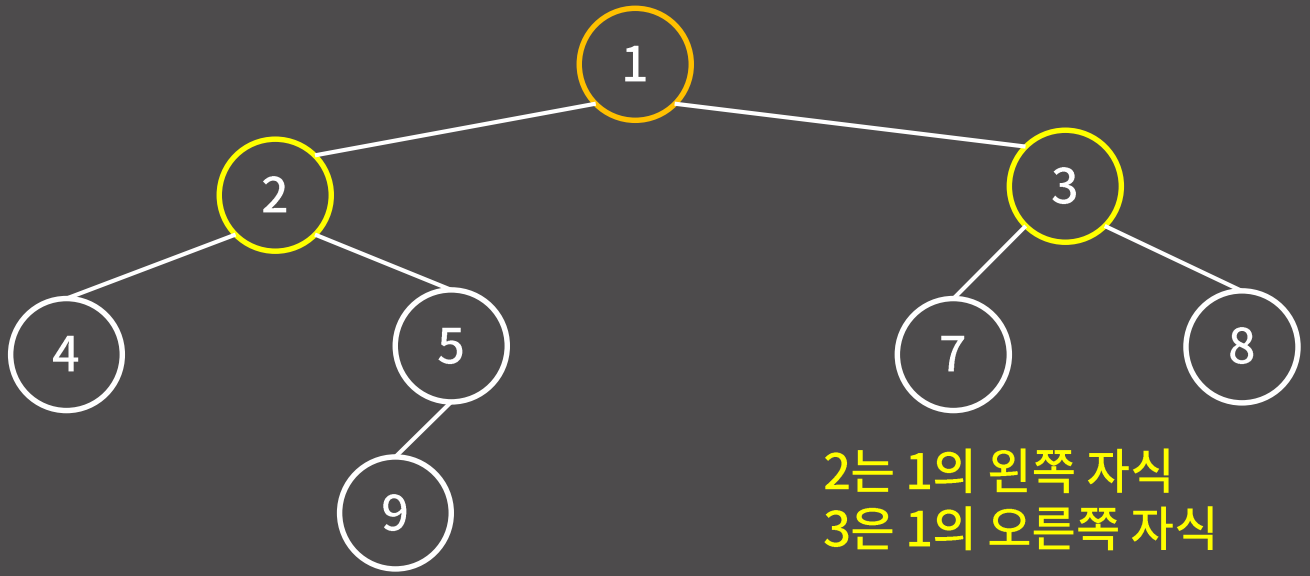
0x01 이진 검색 트리(Binary Search Tree) - 정의

- 이진 트리(Binary Tree) : 각 노드의 자식이 2개 이하인 트리



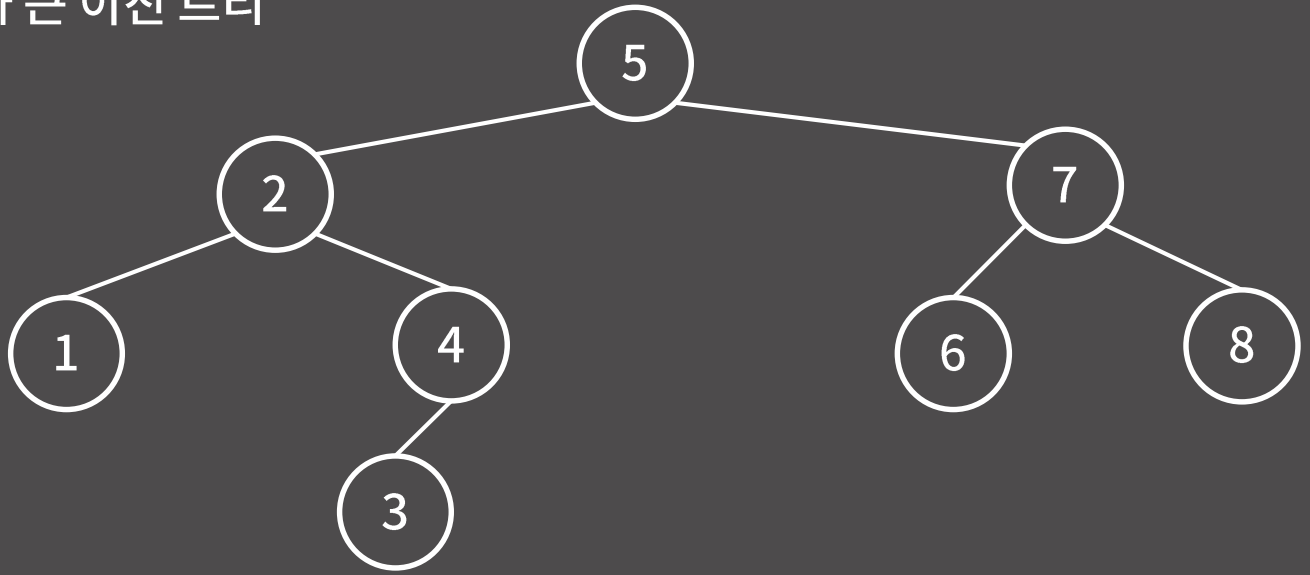
0x01 이진 검색 트리(Binary Search Tree) - 정의

- 자식이 2개 이하이기 때문에 자식을 왼쪽 자식과 오른쪽 자식으로 구분할 수 있다.



0x01 이진 검색 트리(Binary Search Tree) - 정의

- 이진 검색 트리(Binary Search Tree) : 왼쪽 자식은 부모보다 작고 오른쪽 자식은 부모보다 큰 이진 트리



0x01 이진 검색 트리(Binary Search Tree) - 삽입

- 주어진 데이터를 실제로 삽입해보며 이진 검색 트리를 만들어보자.

0x01 이진 검색 트리(Binary Search Tree) - 삽입

- 45 삽입

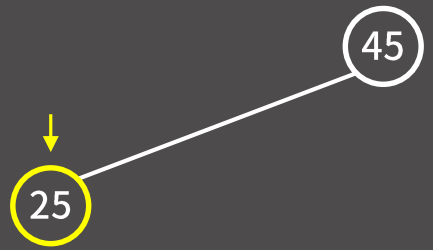
45 (루트)

0x01 이진 검색 트리(Binary Search Tree) - 삽입

- 25 삽입

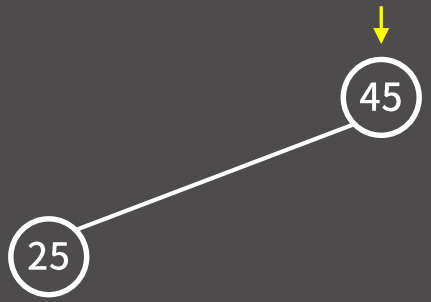
0x01 이진 검색 트리(Binary Search Tree) - 삽입

- 25 삽입



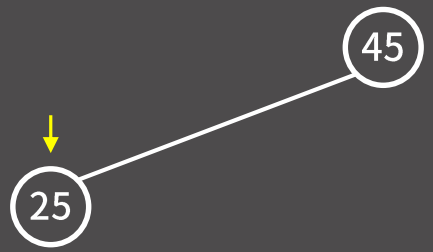
0x01 이진 검색 트리(Binary Search Tree) - 삽입

- 35 삽입



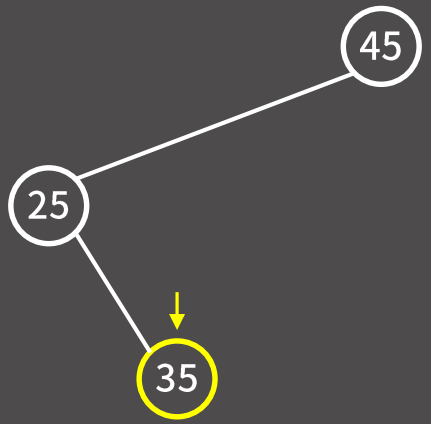
0x01 이진 검색 트리(Binary Search Tree) - 삽입

- 35 삽입



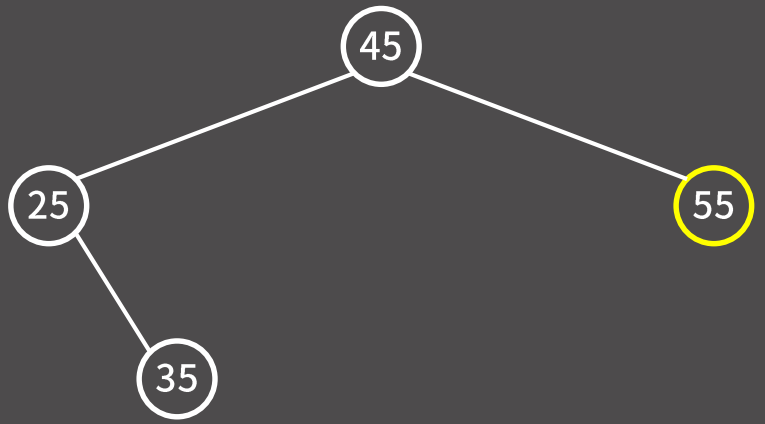
0x01 이진 검색 트리(Binary Search Tree) - 삽입

- 35 삽입



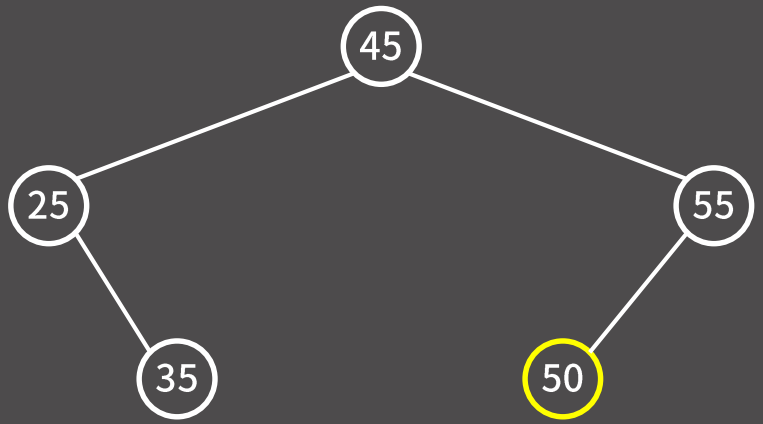
0x01 이진 검색 트리(Binary Search Tree) - 삽입

- 55 삽입



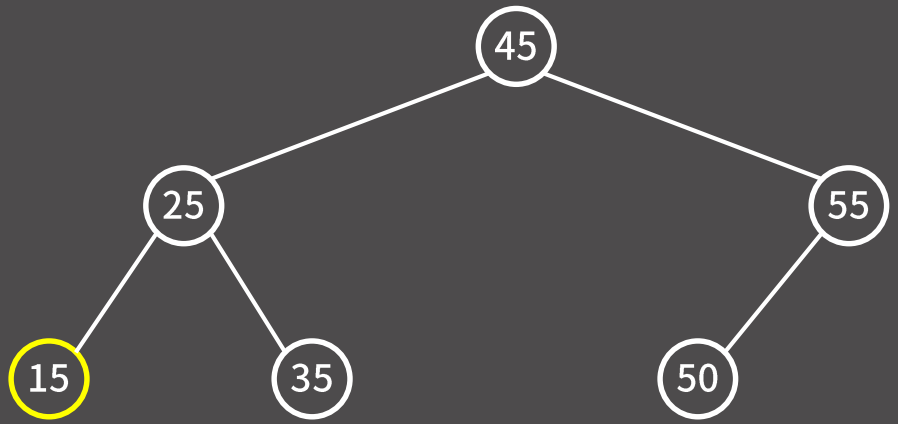
0x01 이진 검색 트리(Binary Search Tree) - 삽입

- 50 삽입



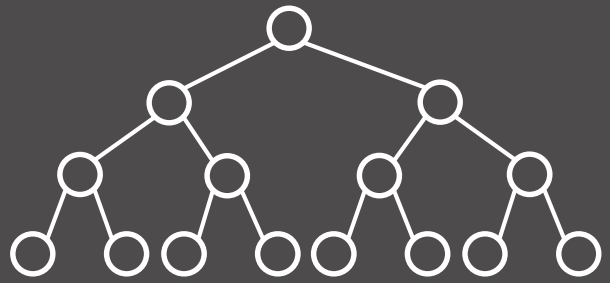
0x01 이진 검색 트리(Binary Search Tree) - 삽입

- 15 삽입

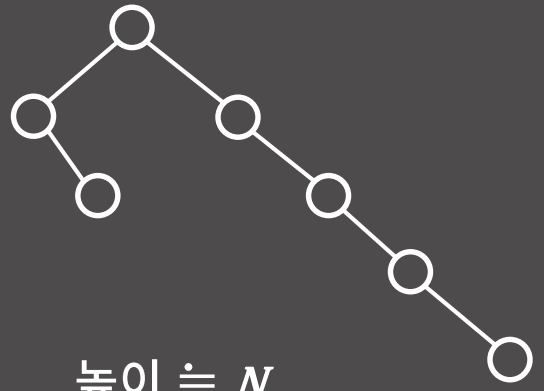


0x01 이진 검색 트리(Binary Search Tree) - 삽입

- 특정 노드를 삽입할 때 필요한 시간 : 해당 노드가 자기 자리를 찾아가기 위해 비교를 몇 번 해야 하는지(=해당 노드의 높이가 얼마인지)에 비례



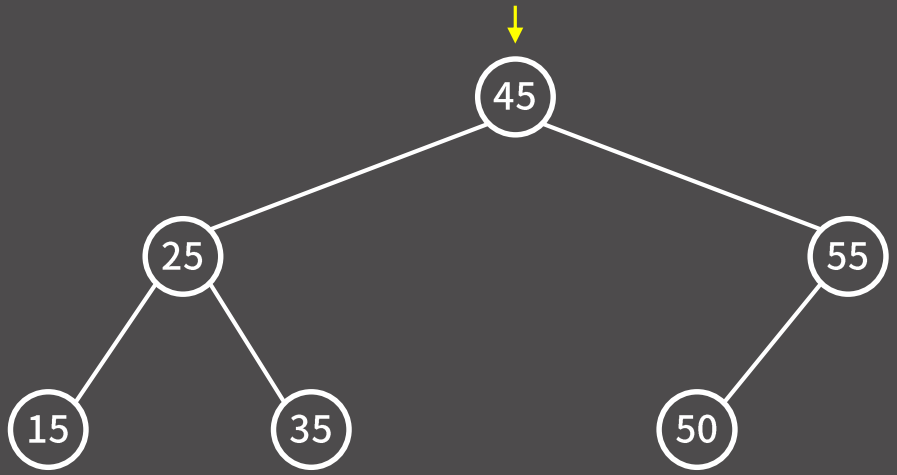
높이 $\fallingdotseq \lg N$



높이 $\fallingdotseq N$

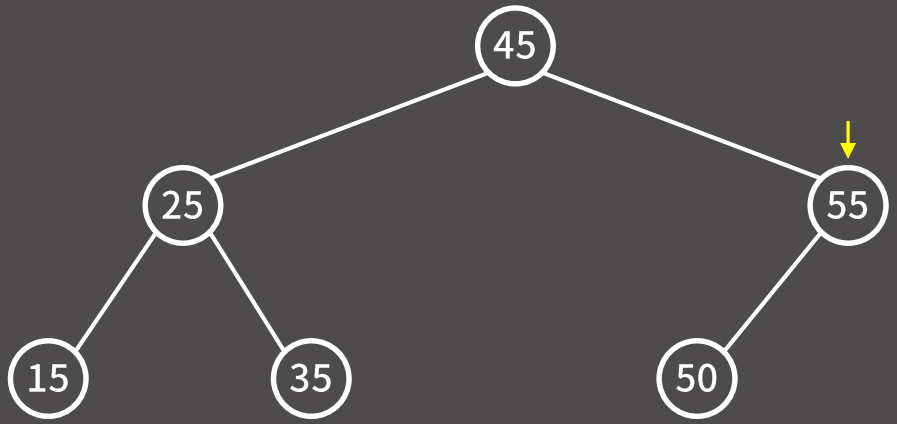
0x01 이진 검색 트리(Binary Search Tree) - 검색

- 삽입과 비슷하게 루트에서 출발해 좌우로 움직이면 됨. ex : 50을 찾고 싶다.



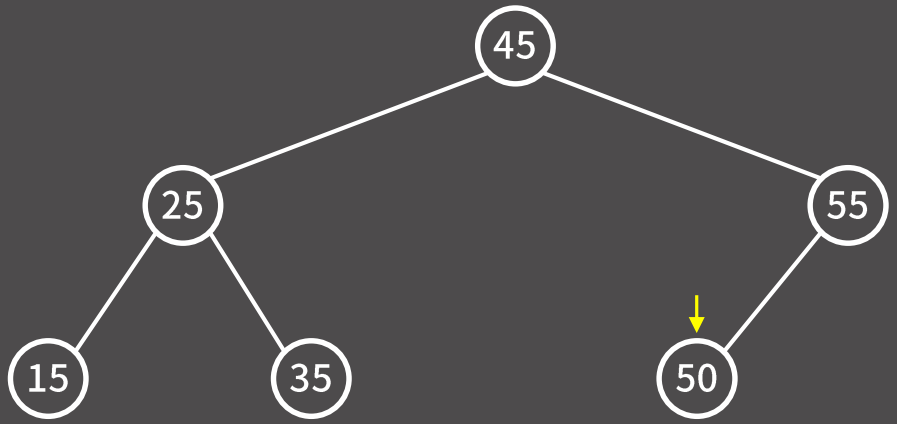
0x01 이진 검색 트리(Binary Search Tree) - 검색

- 삽입과 비슷하게 루트에서 출발해 좌우로 움직이면 됨. ex : 50을 찾고 싶다.



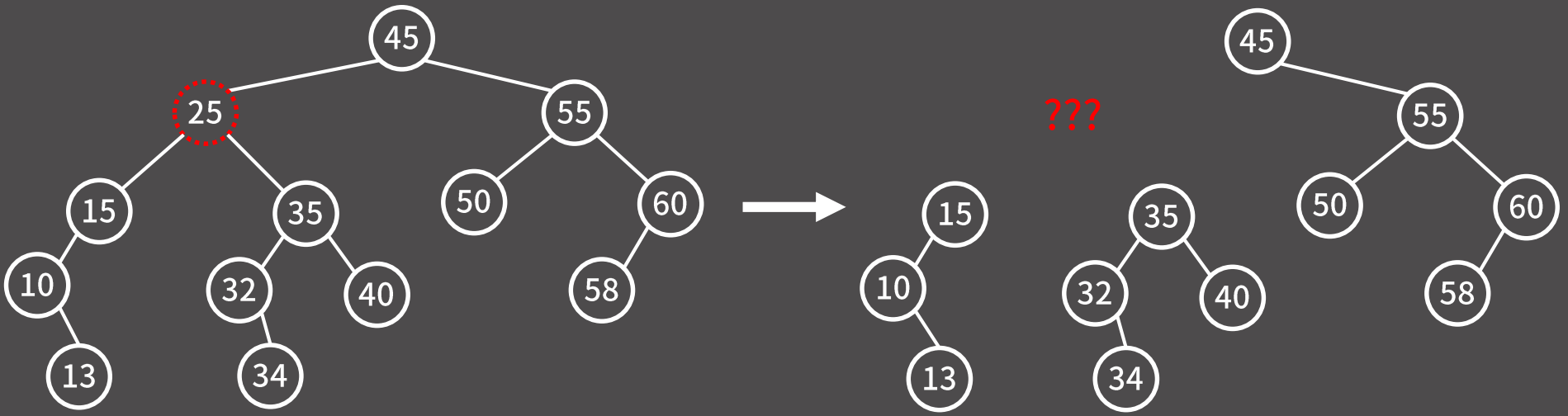
0x01 이진 검색 트리(Binary Search Tree) - 검색

- 삽입과 비슷하게 루트에서 출발해 좌우로 움직이면 됨. ex : 50을 찾고 싶다.
- 삽입과 마찬가지로 시간복잡도는 높이에 비례.



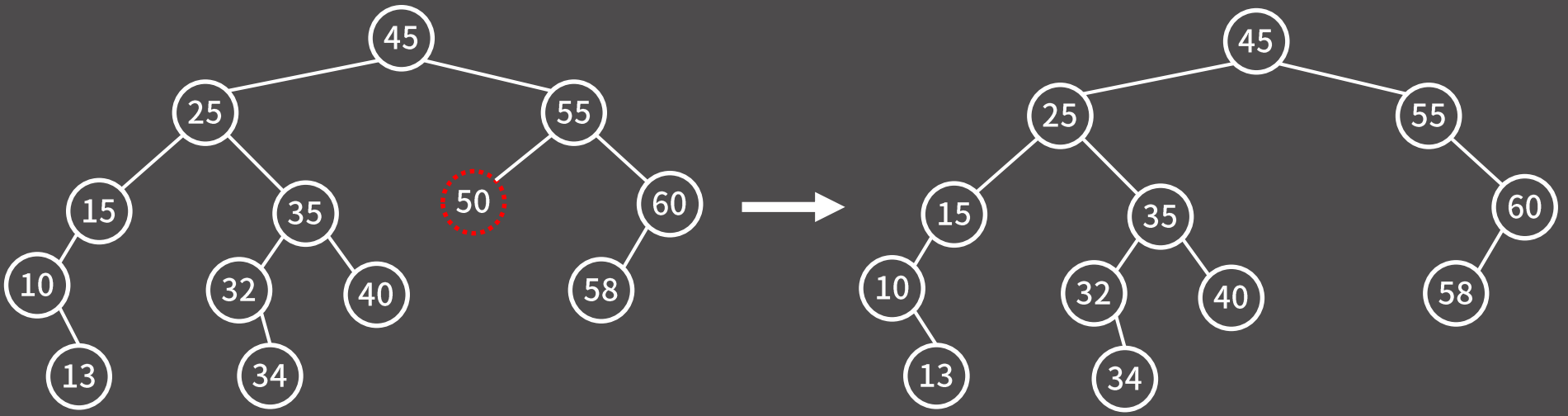
0x01 이진 검색 트리(Binary Search Tree) - 삭제

- 특정 노드를 삭제하고 싶을 때, 단순히 그 노드를 지운다고 끝이 아니다.



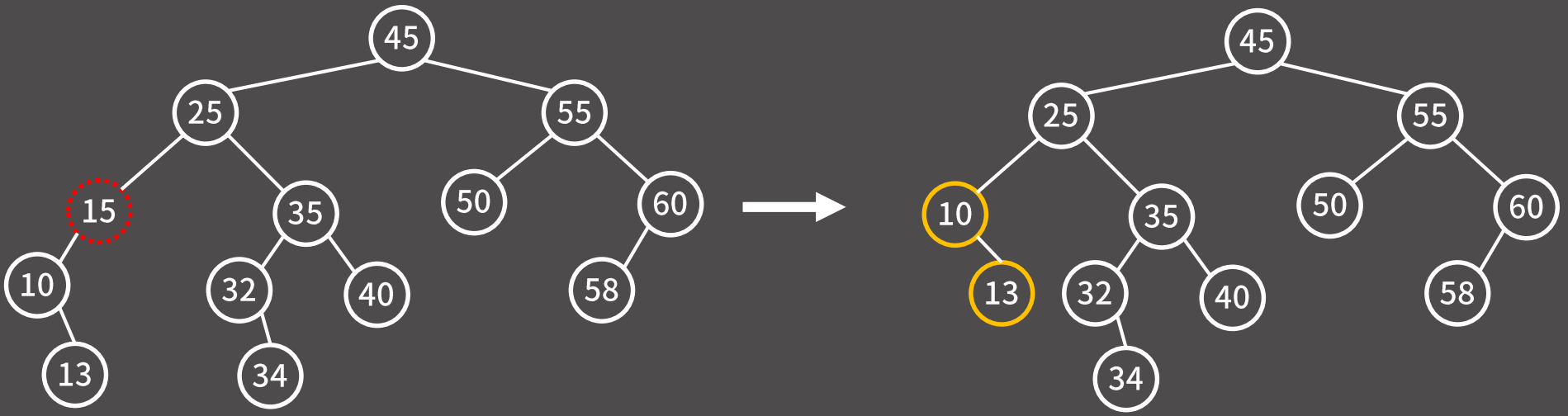
0x01 이진 검색 트리(Binary Search Tree) - 삭제

- Case I. 자식이 없는 노드를 지울 때 : 그냥 지우면 됨



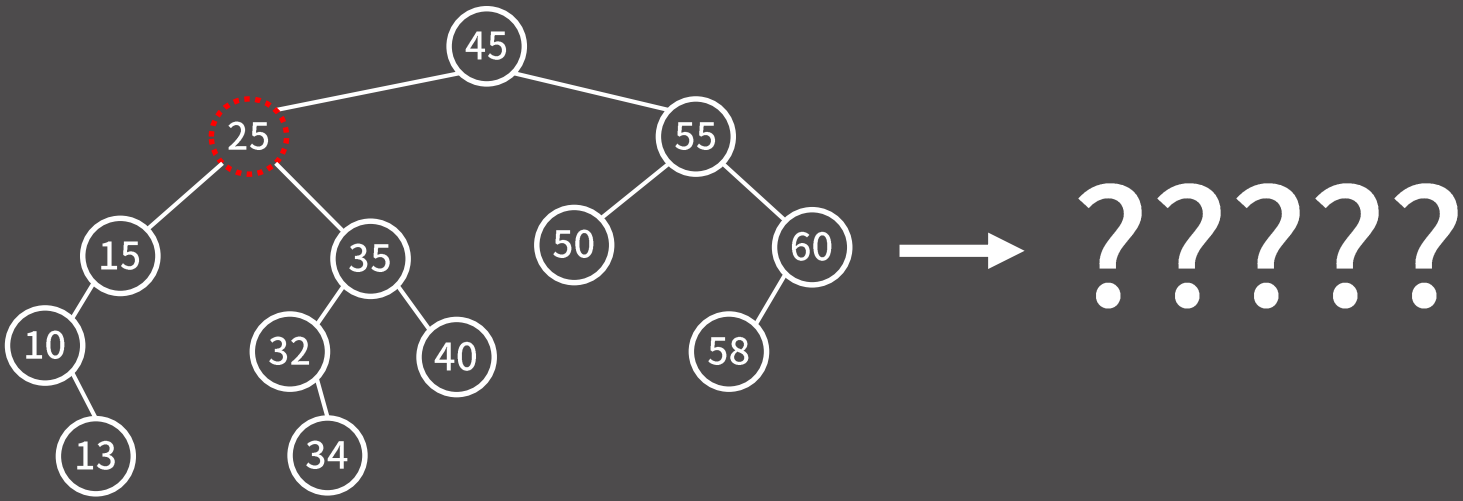
0x01 이진 검색 트리(Binary Search Tree) - 삭제

- Case II. 자식이 1개인 노드를 지울 때 : 자식을 지워진 노드의 자리에 올리면 된다.



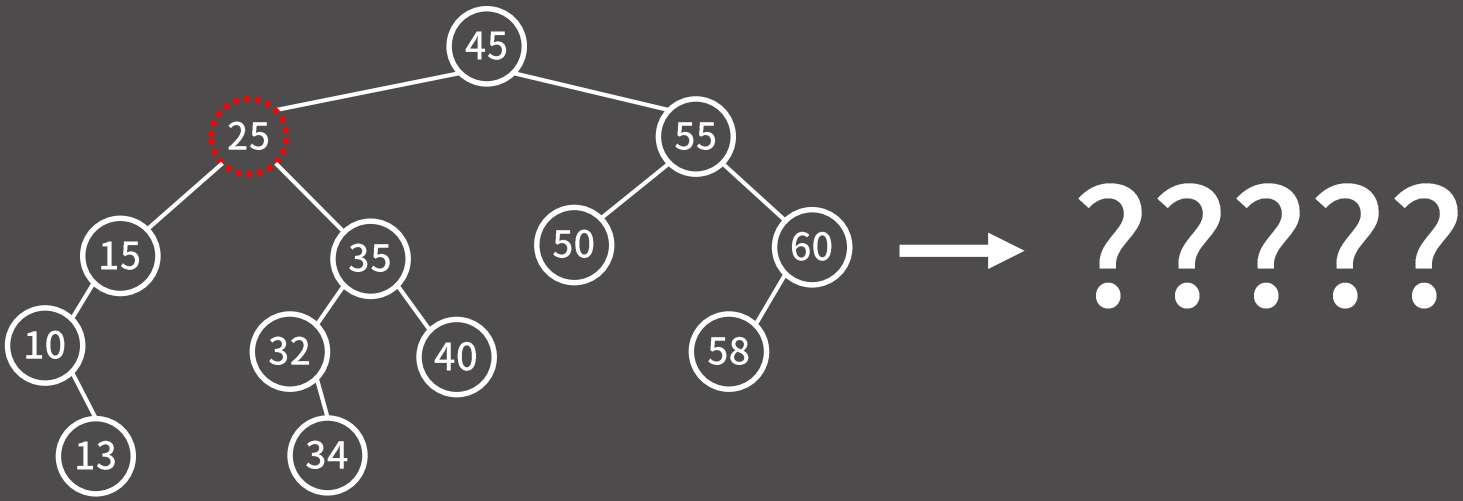
0x01 이진 검색 트리(Binary Search Tree) - 삭제

- Case III : 자식이 2개인 노드를 지울 때



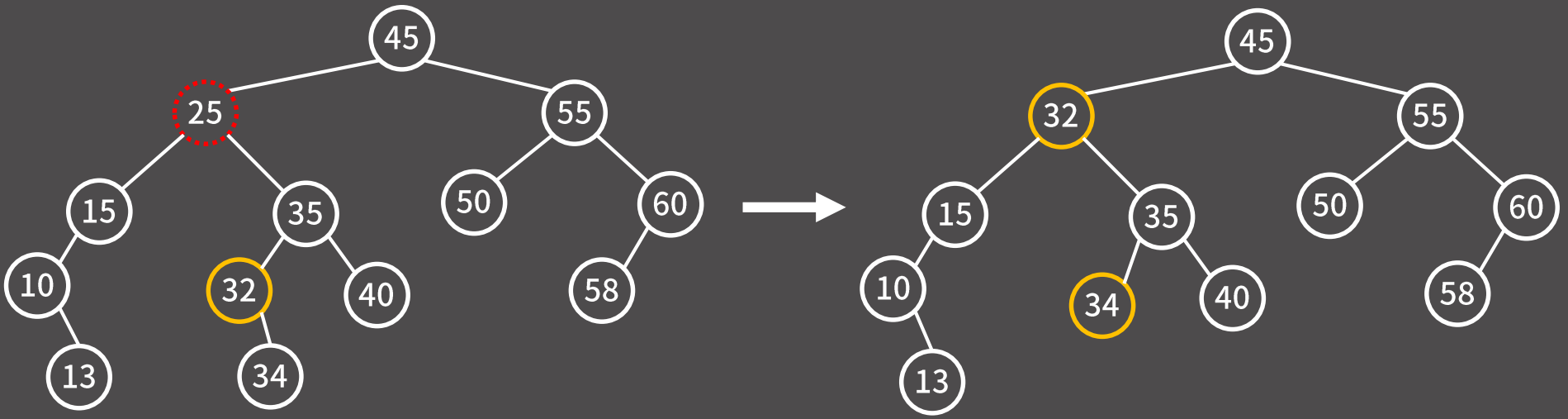
0x01 이진 검색 트리(Binary Search Tree) - 삭제

- 최대한 연산을 덜 하는 방향으로 삭제를 구현하고 싶다. 어떤 “적절한 노드”를 25가 있는 위치로 옮기고 싶다.



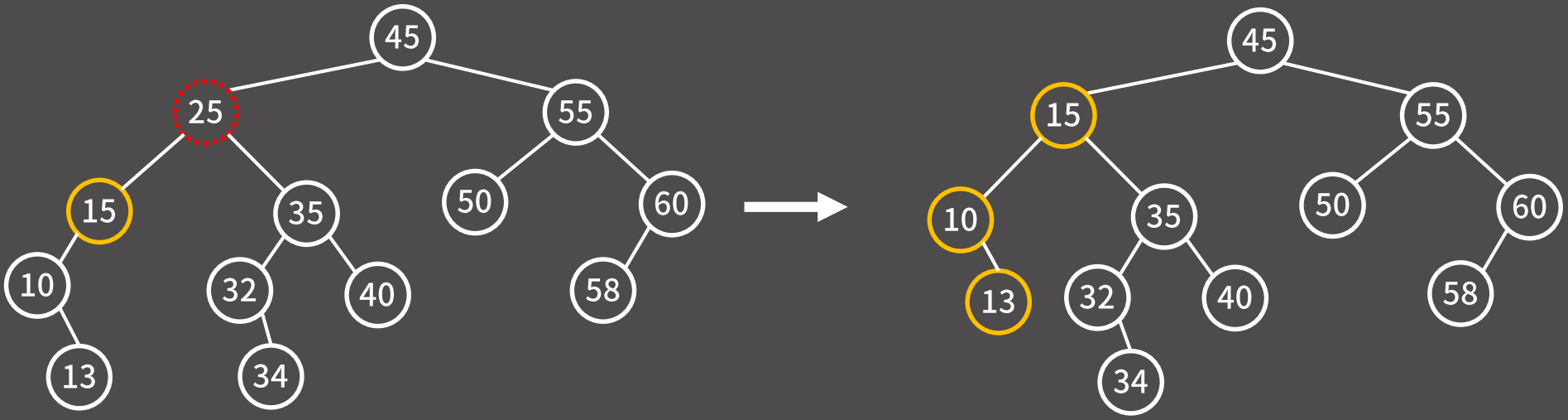
0x01 이진 검색 트리(Binary Search Tree) - 삭제

- 32를 지우고 25의 자리로 보냈을 때 트리의 구조가 깨지지 않음을 확인할 수 있다.



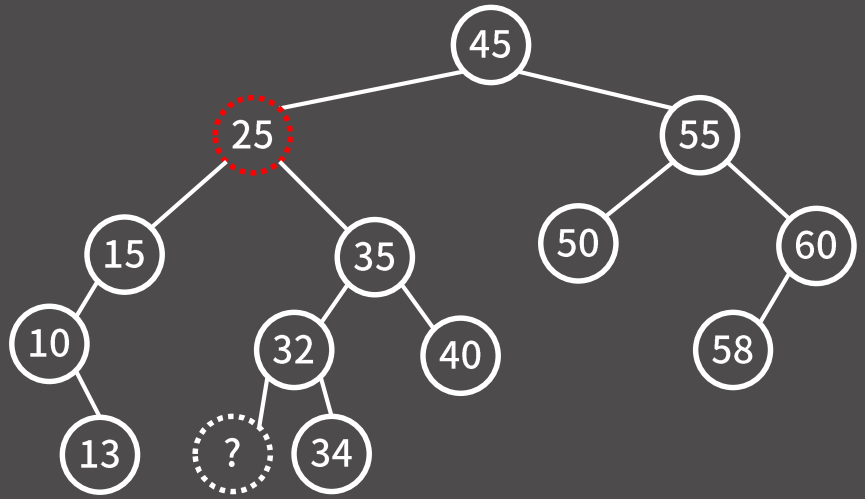
0x01 이진 검색 트리(Binary Search Tree) - 삭제

- 32는 25보다 크면서 가장 작은 원소이기 때문이다. 마찬가지 이유에서 32를 옮기는 대신 25보다 작으면서 가장 큰 15를 25의 자리로 옮겨도 문제가 없다.



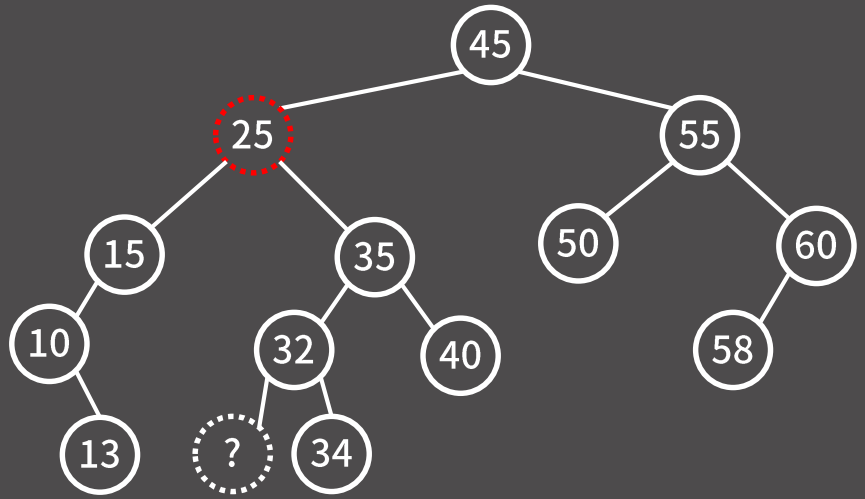
0x01 이진 검색 트리(Binary Search Tree) - 삭제

- Q. 만약 지금은 32에 자식이 1개였으니까 지울 수 있었는데 32에 자식이 2개였으면 어떡하나요?



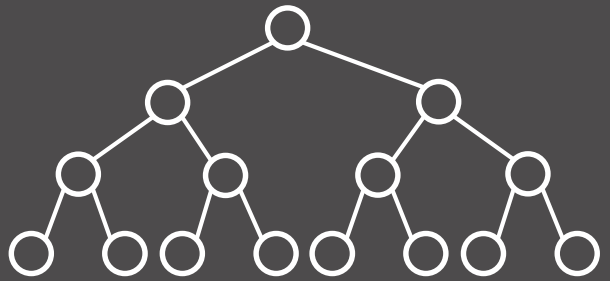
0x01 이진 검색 트리(Binary Search Tree) - 삭제

- A. 32에 왼쪽 자식이 있었다면 그 원소는 25보다 크고 32보다 작습니다. 그렇기에 왼쪽 자식이 있다면 32가 25보다 크면서 가장 작은 원소일 수 없습니다.



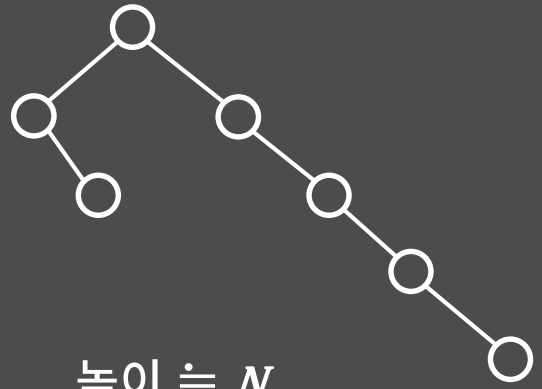
0x01 이진 검색 트리(Binary Search Tree) - 자가 균형 트리

- 이진 검색 트리에서 삽입, 검색, 삭제에 필요한 시간 : 해당 노드가 자기 자리를 찾아가기 위해 비교를 몇 번 해야 하는지(=해당 노드의 높이가 얼마인지)에 비례



높이 $\fallingdotseq \lg N$

Balanced Tree(균형 트리)

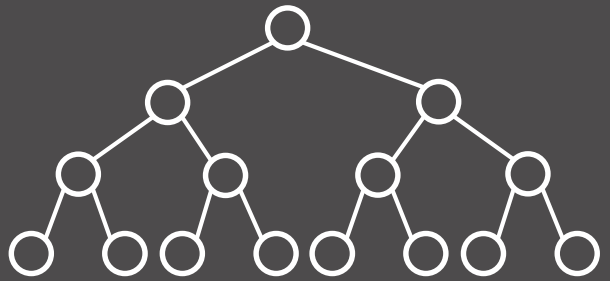


높이 $\fallingdotseq N$

Degenerated Tree(편향된 트리)

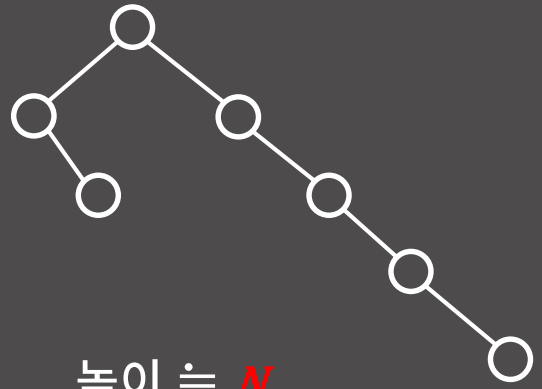
0x01 이진 검색 트리(Binary Search Tree) - 자가 균형 트리

- 트리가 편향되어 있을 경우 사실상 트리를 쓰는 이유가 없어진다.



높이 ≒ $\lg N$

Balanced Tree(균형 트리)

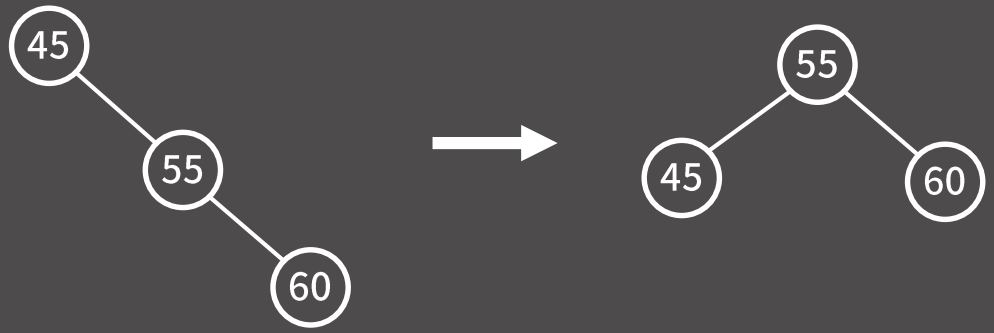


높이 ≒ N

Degenerated Tree(편향된 트리)

0x01 이진 검색 트리(Binary Search Tree) - 자가 균형 트리

- 이진 검색 트리가 편향되지 않게 바로잡도록 삽입, 삭제를 개선한 트리가 존재한다.
- 주로 AVL 트리와 Red Black 트리를 많이 다루나 이번 강의에서는 생략한다.



AVL 트리의 예시

0x01 이진 검색 트리(Binary Search Tree) - STL

- 이진 검색 트리는 삽입, 삭제, 검색을 $\lg N$에 수행할 수 있는 자료구조이나 코딩테스트에서 직접 구현해야 하는 일은 없다.
- STL set : 값들을 이진 검색 트리로 저장하는 자료구조, 단 중복 원소는 허용하지 않음
- STL map : (key, value)를 key에 대한 이진 검색 트리로 저장하는 자료구조, 단 중복 key는 허용하지 않음
- STL multiset : 값들을 이진 검색 트리로 저장하는 자료구조, 중복 원소가 허용됨
- STL multimap : (key, value)를 key에 대한 이진 검색 트리로 저장하는 자료구조, 중복 key가 허용됨

0x01 이진 검색 트리(Binary Search Tree) - STL

- STL set, multiset 예제 (set, multiset 헤더 안에 존재)

```cpp
#include <bits/stdc++.h>
//#include <set>
//#include <multiset>
using namespace std;
int main(){
  set<int> S;
  S.insert(3); // S = {3}
  S.insert({5,1,7}); // S = {1,3,5,7}
  if(S.find(6) == S.end())
    cout << "6 not in S\n"; // 이것이 출력됨
  else
    cout << "6 in S\n";
  cout << S.size() << '\n'; // 4
  S.erase(2); // 아무일도 일어나지 않음
  S.erase(1); // S = {3,5,7}
  S.insert(3); // set은 중복원소가 허용되지 않으므로 S = {3,5,7} 그대로.
  for(auto e : S) cout << e << ' '; // 3 5 7
  cout << '\n';
  S.clear();
  cout << S.size() << '\n'; // 0
  multiset<int> MS;
  MS.insert(1);
  MS.insert(1);
  MS.insert(1);
  cout << MS.count(1); // 3
}
```

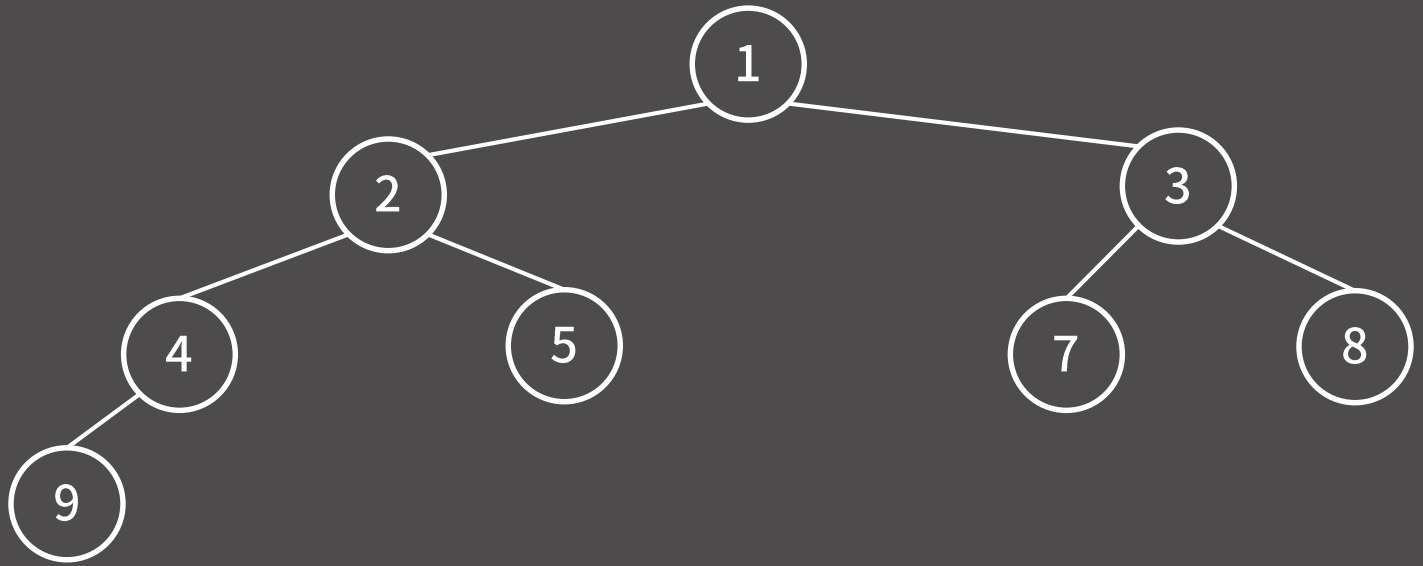
0x01 이진 검색 트리(Binary Search Tree) - STL

- STL map, multimap 예제 (map, multimap 헤더 안에 존재)

```cpp
#include <bits/stdc++.h>
//#include <map>
//#include <multimap>
using namespace std;
int main(){
  map<string, int> M; // key : string, value : int
  M["kim"] = 2621672;
  M["qwerty"] = 4271234;
  M["apple"] = -42234;
  for(auto e : M)
    cout << e.first << " : " << e.second << '\n';
  /*
  apple : -42234
  qwerty : 4271234
  kim : 2621672
  */
  cout << M.size() << '\n'; // 2
  if(M.find("apple") == M.end())
    cout << "key apple not in M\n";
  else
    cout << "key apple in M\n"; // 이것이 출력됨
  M["apple"] = 1234;
  cout << M["apple"] << '\n'; // 1234
  M.erase("qwerty");
  M.clear();
  multimap<int,string> MS;
  // Multimap은 동일한 키가 여러 개일 수 있으니 [] 연산자로 원소에 접근할 수 없다.
  MS.insert({1,"hello"});
  MS.insert({1,"hi"});
  MS.insert({-10,"zcv"});
  for(auto e : MS)
    cout << e.first << " : " << e.second << '\n';
  /*
  -10 : zcv
  1 : hello
  1 : hi
  */
}
```

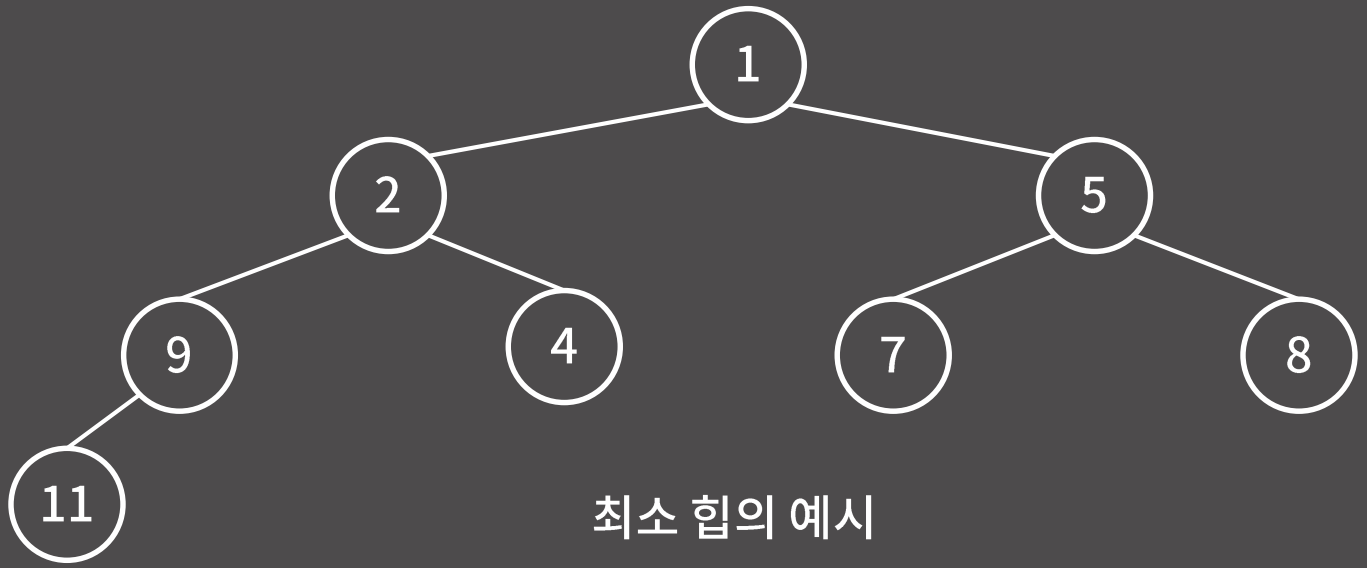
0x02 힙(Heap) – 정의

- 힙 : 최댓값 혹은 최솟값을 빠르게 찾아내기 위한 이진 트리. 이진 검색 트리와 헷갈리지 말 것.



0x02 힙(Heap) – 정의

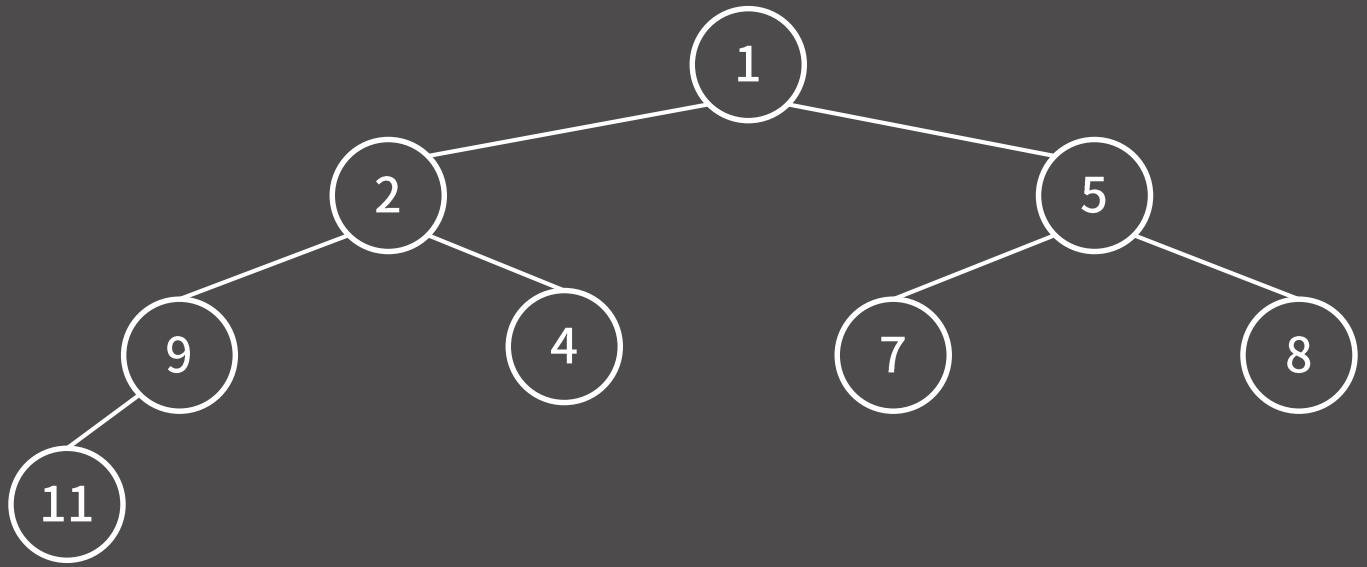
- 최소 힙의 경우 부모는 자식보다 작아야 하고, 최대 힙의 경우 부모는 자식보다 커야 한다. 이 때 루트가 최솟값 혹은 최댓값이 된다.



최소 힙의 예시

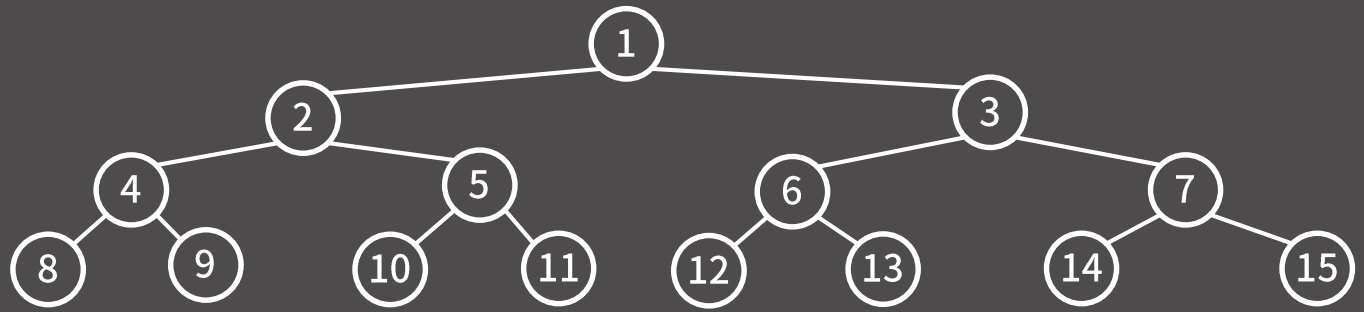
0x02 힙(Heap) – 정의

- 최소 힙에서는 원소의 삽입, 최솟값 확인, 최솟값 삭제의 기능을 제공하고, 최대 힙에서는 원소의 삽입, 최댓값 확인, 최댓값 삭제의 기능을 제공한다.



0x02 힙(Heap) – 삽입

- 힙에서의 삽입은 일단 원소를 트리 상의 다음 공간에 추가하고, 힙의 성질을 만족하게끔 서로 자리를 바꾸는 방식으로 구현된다. 삽입하는 순서가 아래와 같은 방식이기 때문에 균형 트리임이 보장된다.



삽입하는 순서

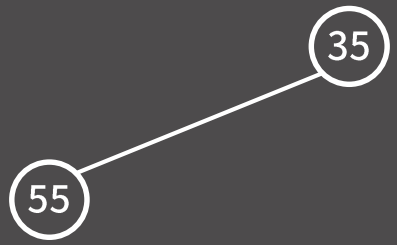
0x02 힙(Heap) – 삽입

- 최소 힙을 만들어보자. 35 삽입

35 (루트)

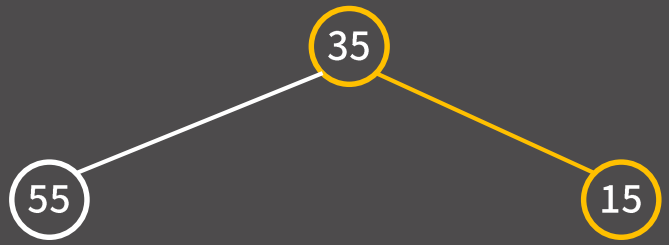
0x02 힙(Heap) – 삽입

- 55 삽입



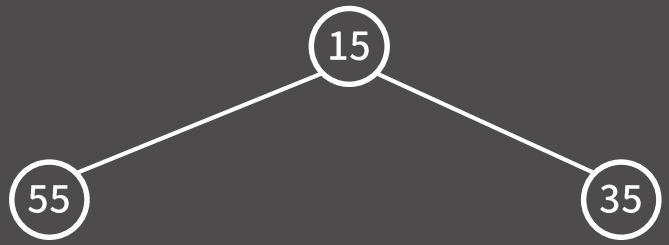
0x02 힙(Heap) – 삽입

- 15 삽입



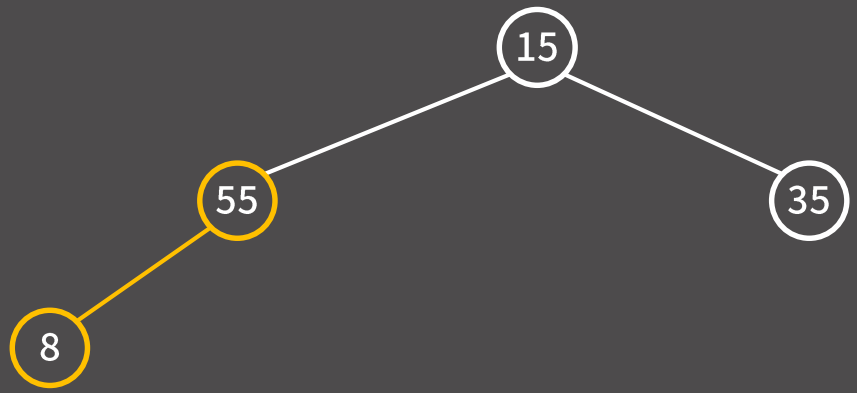
0x02 힙(Heap) – 삽입

- 15 삽입



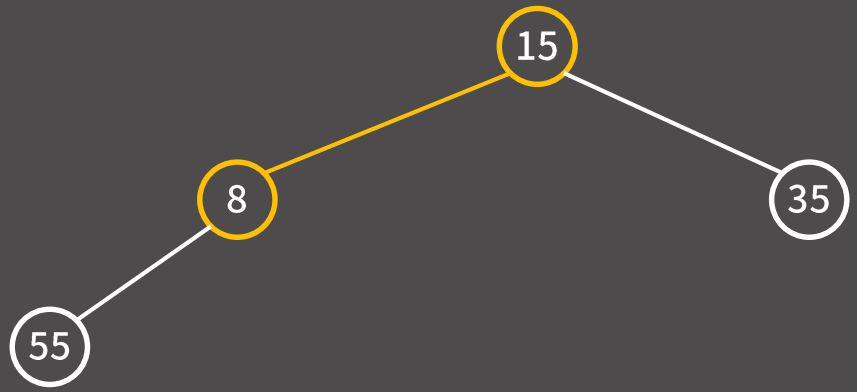
0x02 힙(Heap) – 삽입

- 8 삽입



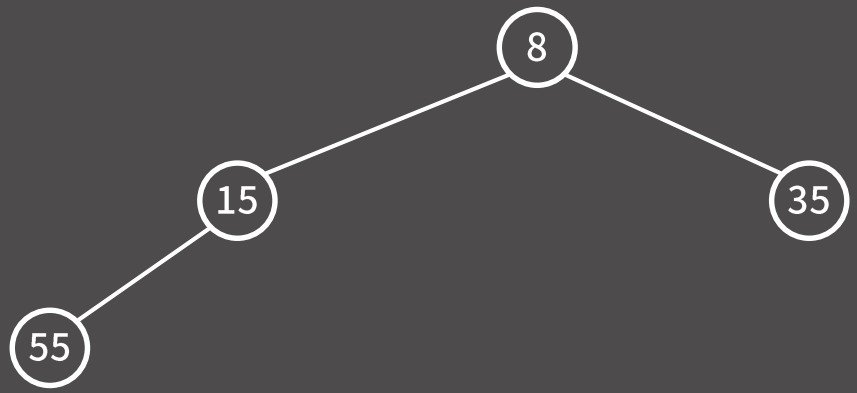
0x02 힙(Heap) – 삽입

- 8 삽입



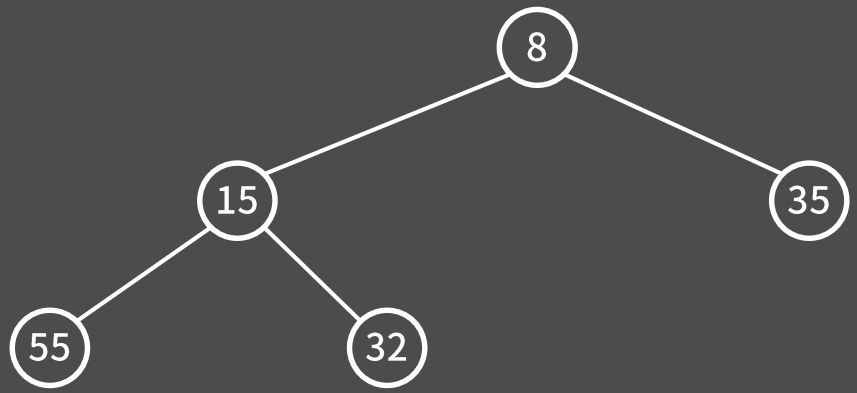
0x02 힙(Heap) – 삽입

- 8 삽입



0x02 힙(Heap) – 삽입

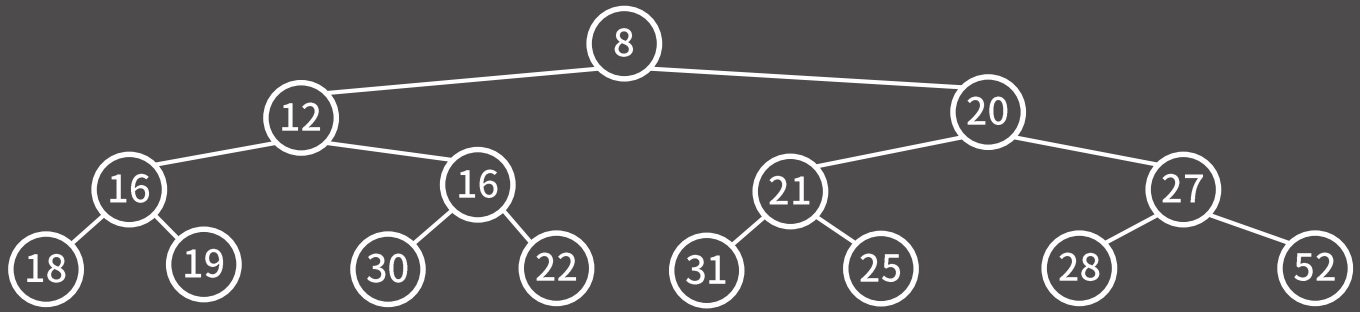
- 32 삽입



- 최대 높이 만큼만 올라가면 삽입이 가능하고, 균형 트리이기 때문에 $lg\,N$임이 보장된다.

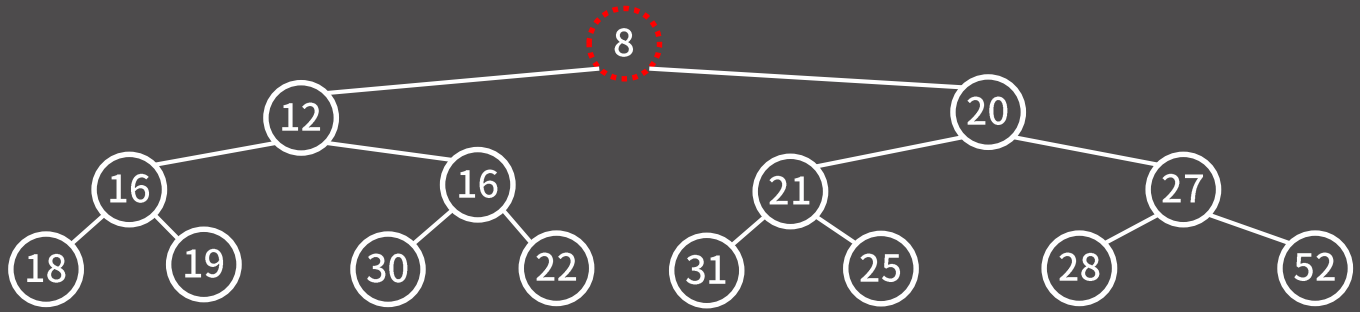
0x02 힙(Heap) – 최솟값 확인

- 루트에 적힌 원소가 최솟값이다.



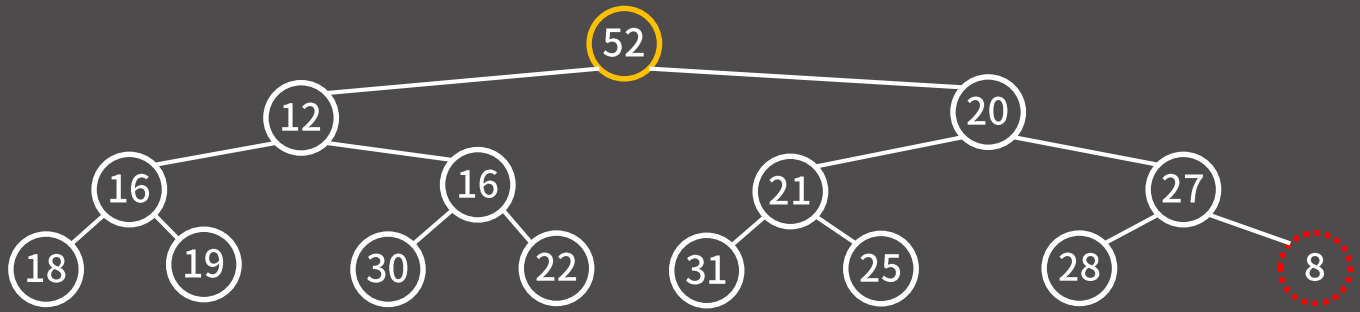
0x02 힙(Heap) – 최솟값 삭제

- 8을 제거하고 싶지만, 무턱대고 8을 지우면 트리 구조가 깨진다.



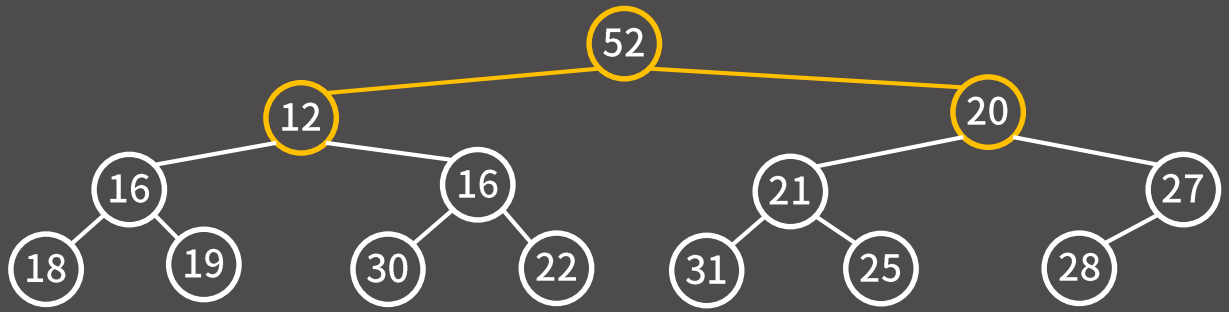
0x02 힙(Heap) – 최솟값 삭제

- 일단 8과 52의 위치를 바꾸고 8을 제거하자.



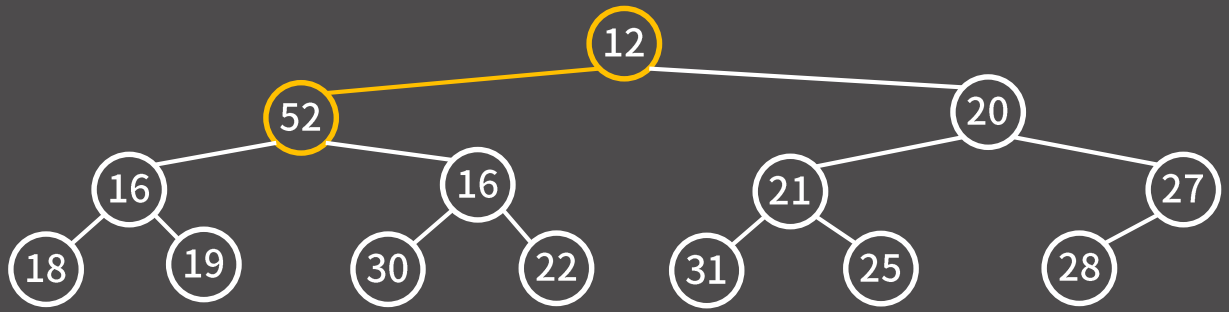
0x02 힙(Heap) – 최솟값 삭제

- 52를 루트로 올림으로 인해 8은 문제없이 지울 수 있지만 부모가 자식보다 작다는 조건이 깨졌다. 이를 52, 12, 20 사이에서 위치를 바꿈으로서 해결하고 싶다.



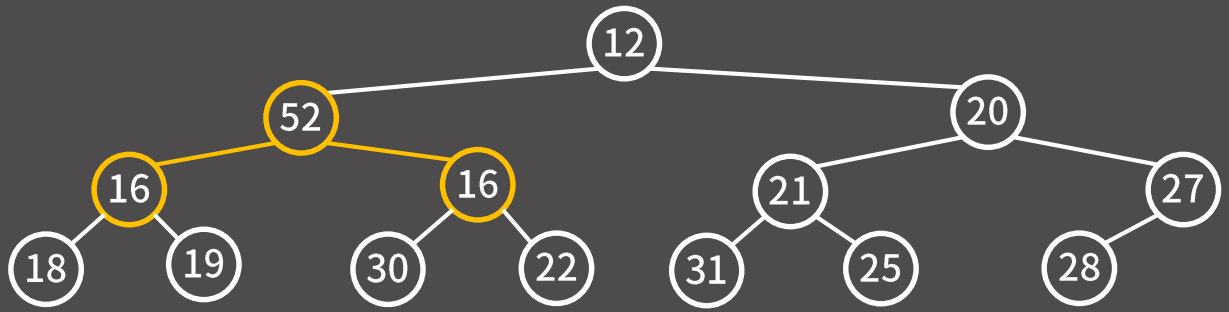
0x02 힙(Heap) – 최솟값 삭제

- 52, 12, 20 중에서 가장 작은 12를 부모로 올리면 해결이 된다.



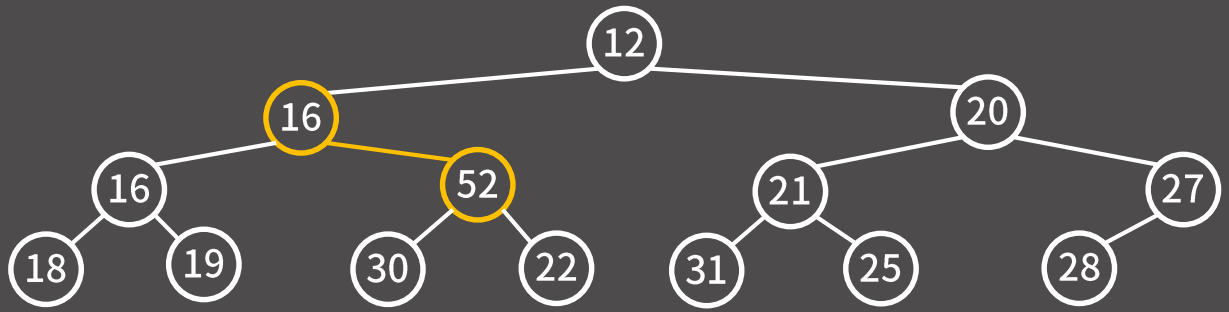
0x02 힙(Heap) – 최솟값 삭제

- 52, 16, 16 에서 또 최소 힙의 성질이 위배된다.



0x02 힙(Heap) – 최솟값 삭제

- 52, 16, 16 중에서 가장 작은 16을 부모로 올리면 해결이 된다.



0x02 힙(Heap) – 최솟값 삭제

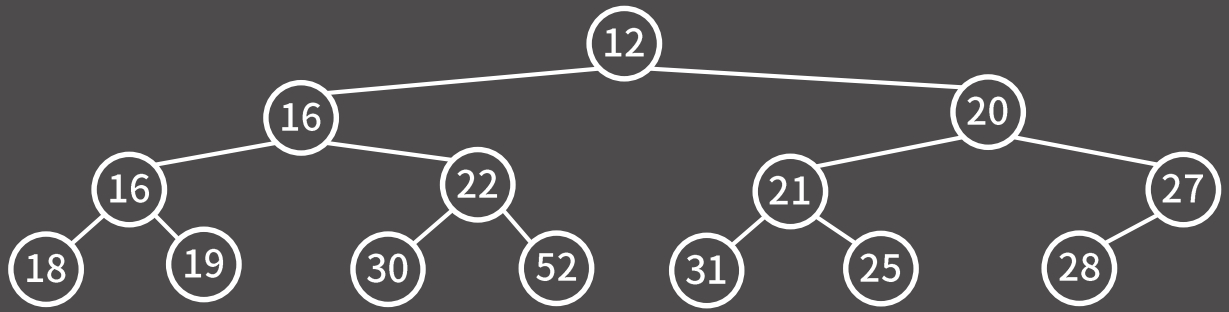
- 52, 30, 22 에서 또 최소 힙의 성질이 위배된다.

0x02 힙(Heap) – 최솟값 삭제

- 52, 30, 22 중에서 가장 작은 22를 부모로 올리면 해결이 된다.

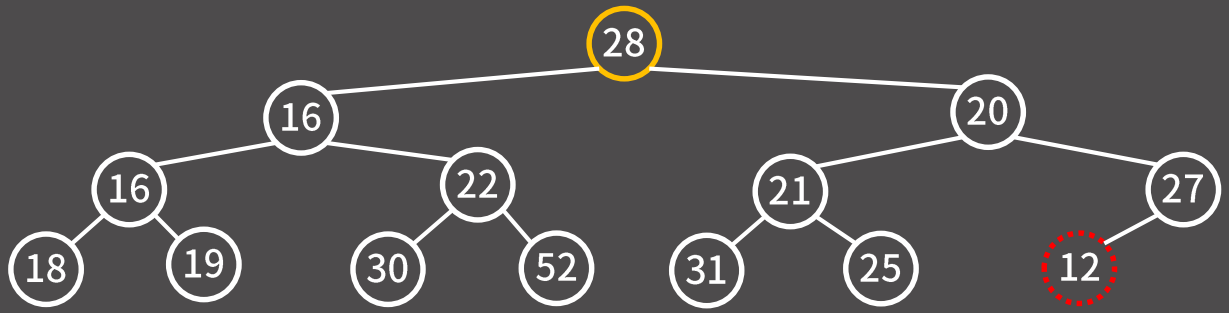
0x02 힙(Heap) – 최솟값 삭제

- 삭제가 완료되었다.



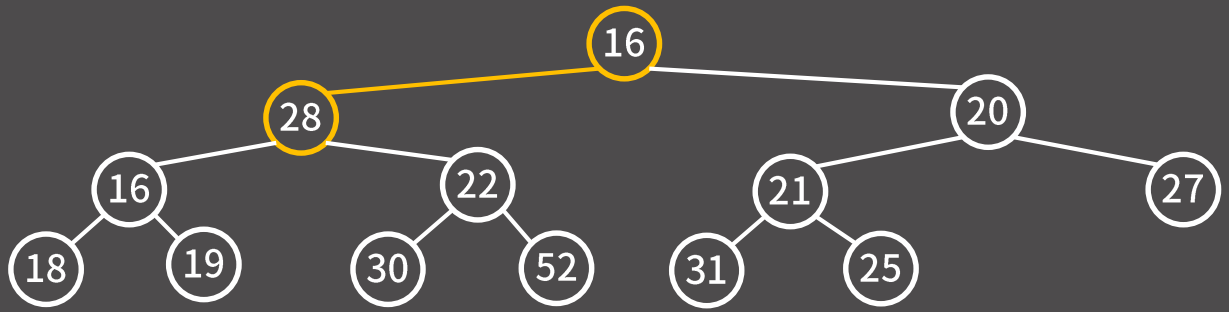
0x02 힙(Heap) – 최솟값 삭제

- 한번 더 삭제해보자.



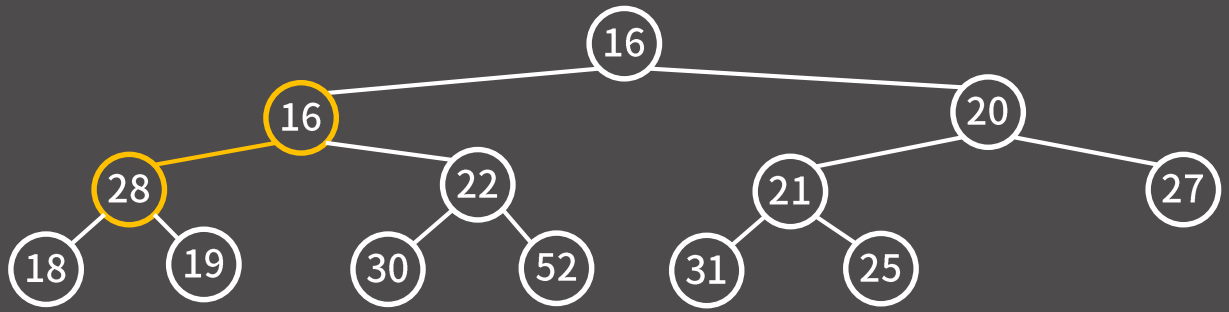
0x02 힙(Heap) – 최솟값 삭제

- 한번 더 삭제해보자.



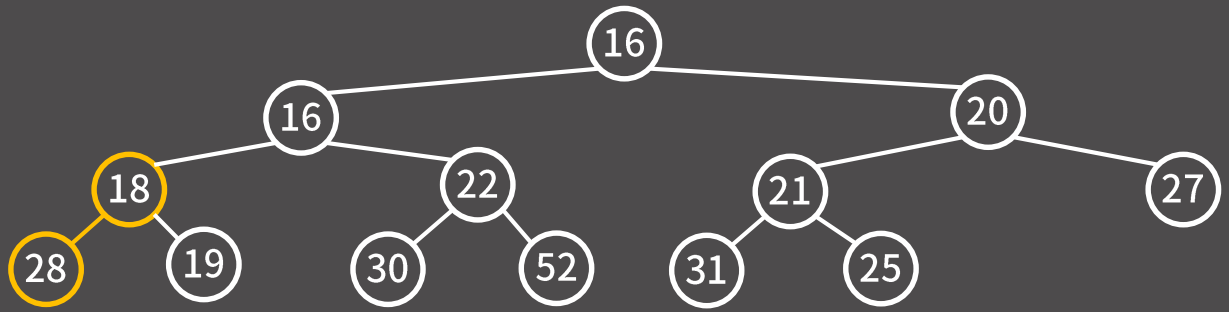
0x02 힙(Heap) – 최솟값 삭제

- 한번 더 삭제해보자.



0x02 힙(Heap) – 최솟값 삭제

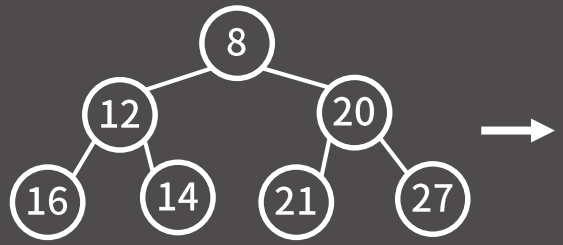
- 한번 더 삭제해보자.



- 마찬가지로 최대 높이 만큼만 내려가면 삭제가 가능하고, 균형 트리이기 때문에 $lg\,N$ 임이 보장된다.

0x02 힙(Heap) – 구현

- 트리의 각 원소를 배열에 대응시키면 구현이 간편하다.



0	1	2	3	4	5	6	7	8	9
	8	12	20	16	14	21	27		

x의 왼쪽, 오른쪽 자식 : 2x, 2x+1
x의 부모 : x/2

- BOJ 1927번 : 최소 힙 문제를 해결하는 코드 :
  http://boj.kr/8dba692181b540d28bf64d34f76cb0d1

0x02 힙(Heap) – STL

- STL에 힙도 있다. 이름은 priority_queue이다. 이 힙은 기본적으로 최대힙이다.
- STL을 이용해 BOJ 1927번 : 최소 힙 문제를 해결하는 코드 : http://boj.kr/e9f05230bfb140029dbbec4545c89625

- Q. 힙에서 할 수 있는건 어차피 균형 이진 트리에서도 할 수 있지 않나요? 그러면 균형 이진 트리가 더 제공해주는 기능이 많은데 힙을 쓸 이유가 있나요?
- A. 힙에서 할 수 있는건 균형 이진 트리에서도 할 수 있고, 시간복잡도도 $O(\lg N)$으로 동일합니다. 그런데 힙이 균형 이진 트리보다 수행 속도가 빠르고, 구현도 쉽고, 공간도 적게 차지합니다.

강의 정리

- 해쉬 – 삽입, 삭제, 검색 모두 이론적으로 O(1), 그러나 충돌이 많이 발생함에 따라 실제 시간복잡도는 더 나쁠 수 있다.
- 이진 검색 트리 – 삽입, 삭제, 검색 모두 $O(\lg N)$, 그러나 트리가 편향됨에 따라 실제 시간복잡도는 최악의 경우 $O(N)$이 될 수도 있고, 이를 해결하기 위해 자가 균형 트리가 존재한다.
- 힙 – 삽입, 최솟(혹은 최댓)값 삭제는 $O(\lg N)$, 최솟(혹은 최댓)값 확인은 O(1). 해쉬, 이진 검색 트리와 달리 구조적으로 시간복잡도가 보장된다.
- STL set, map, priority_queue는 꼭 익혀두자.