Algorithm

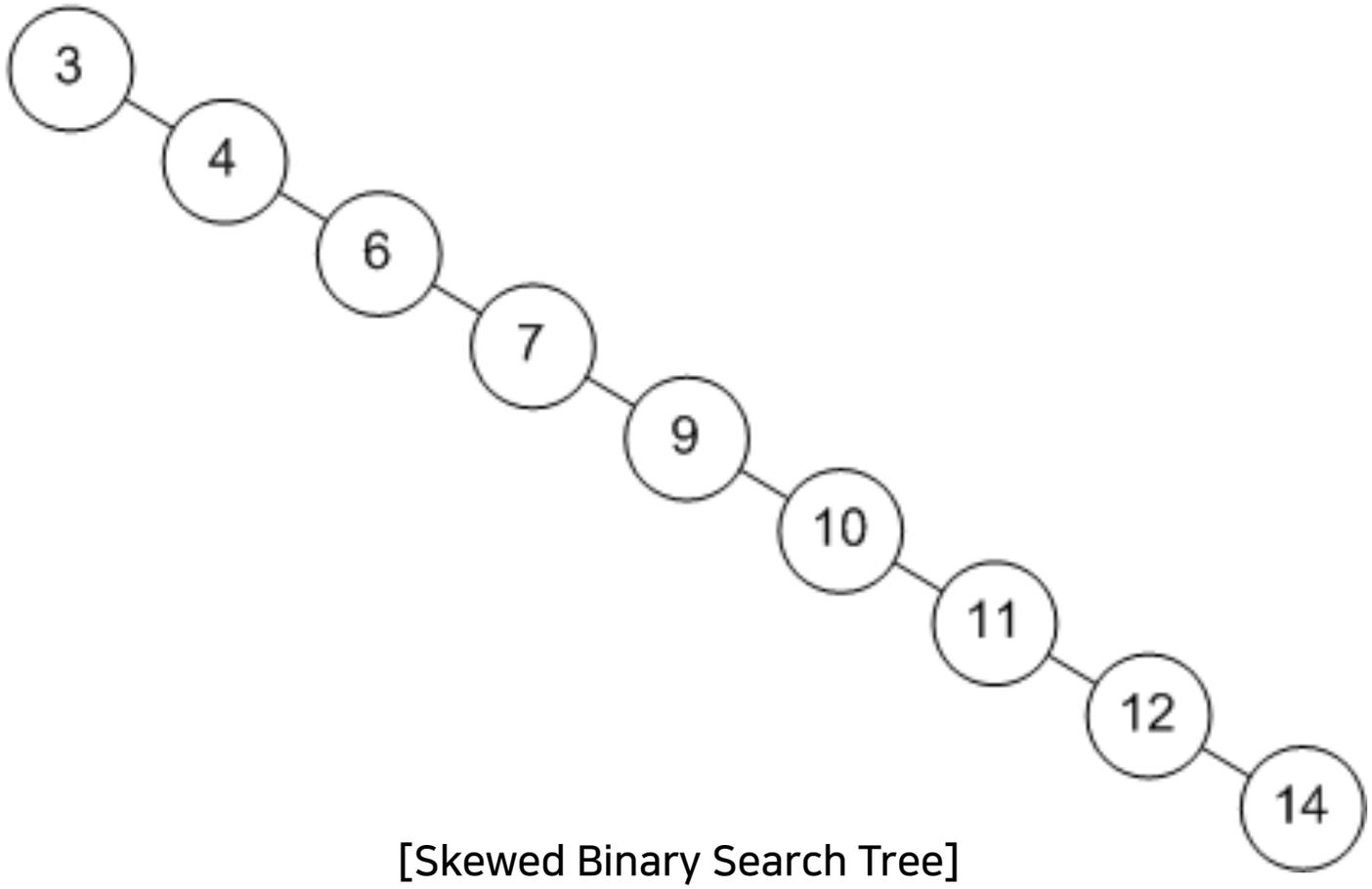
3주차: AVL Tree / Hash Table

3주차: AVL Tree, Hash Table

INDEX

1. 개요
2. AVL Tree
3. Hash Table
4. 과제
5. References

Binary Search Tree의 한계

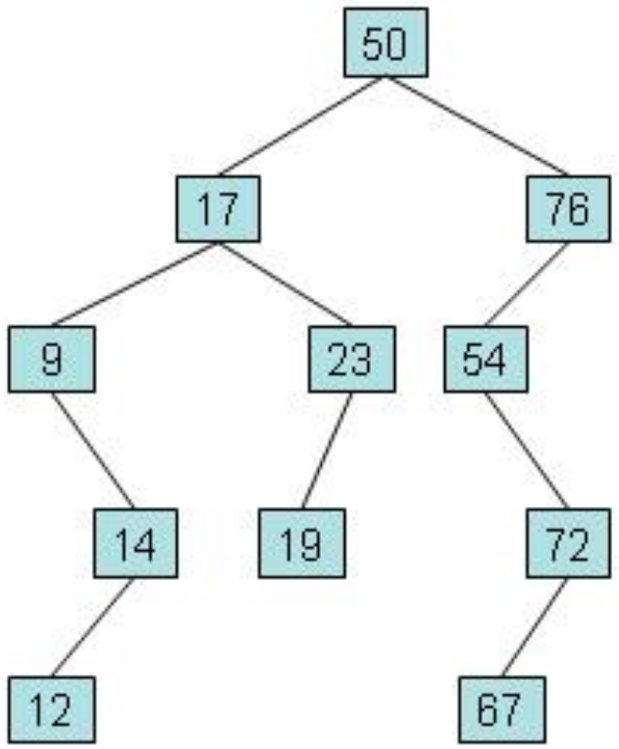


[Skewed Binary Search Tree]

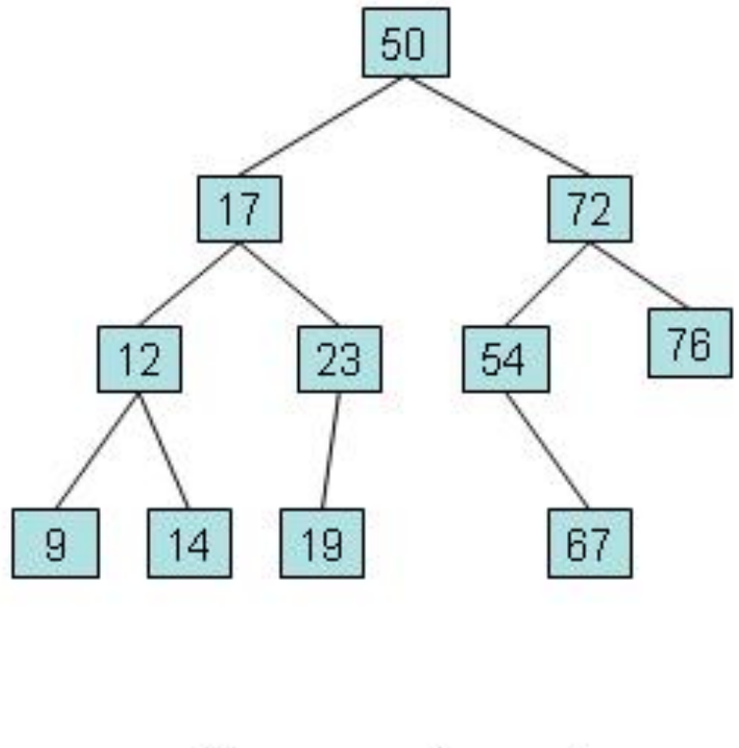
- BST 의 탐색 시간복잡도는 O(h).
- 만약 한쪽으로 치우쳐 있다면 O(h) = O(n).
- 이런 경우를 위해 Balanced BST가 연구됨.

1. AVL Tree

AVL Tree



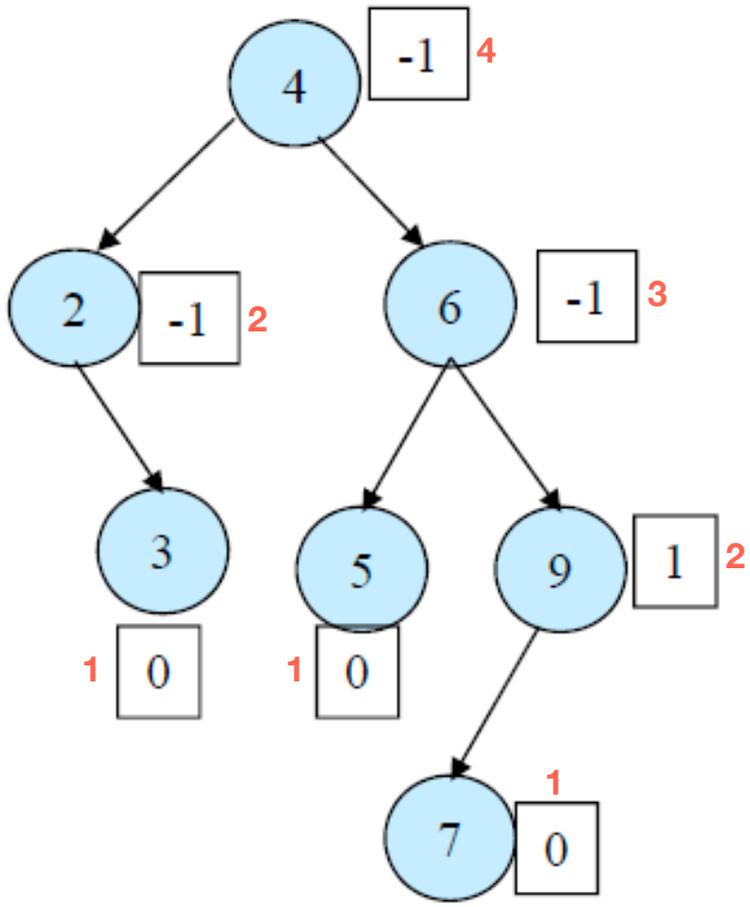
[Unbalanced Binary Search Tree]



[Balanced Binary Search Tree]

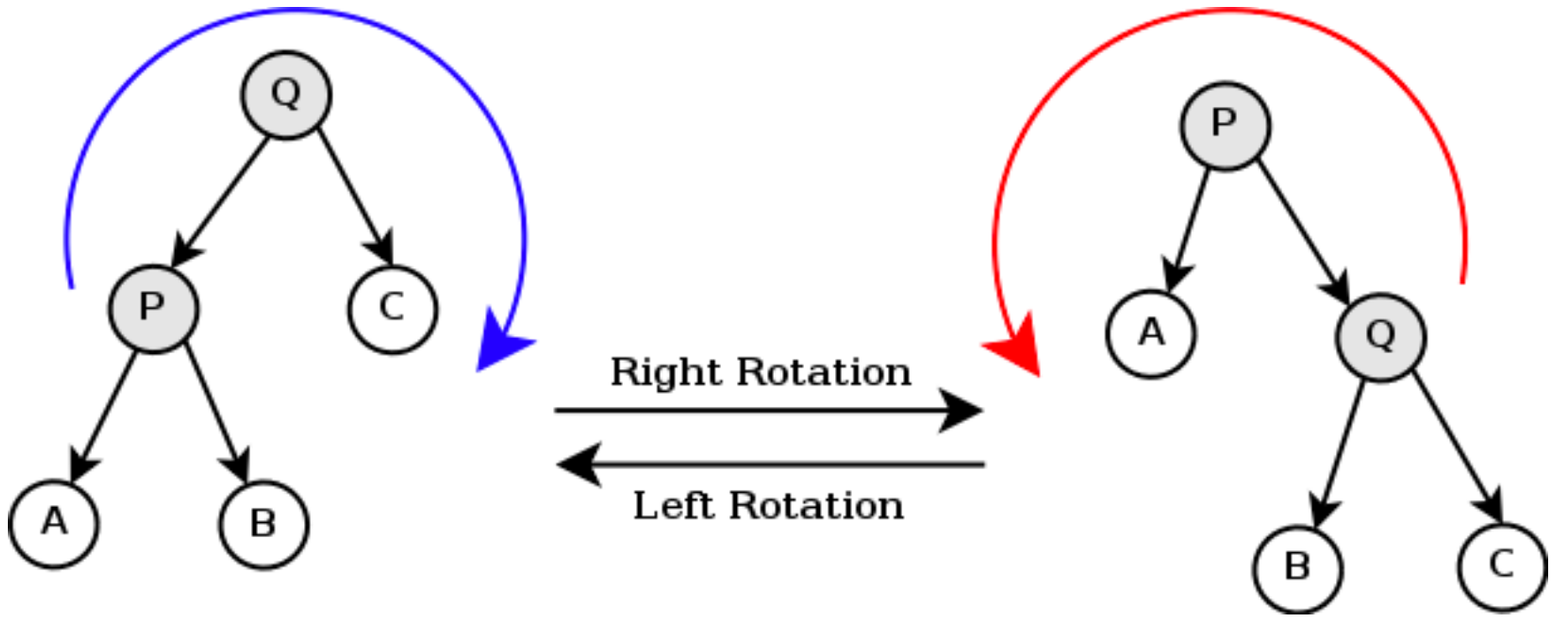
- Balanced BST의 한 종류.
- Rotation을 통해 balancing을 한다.

Balance Factor



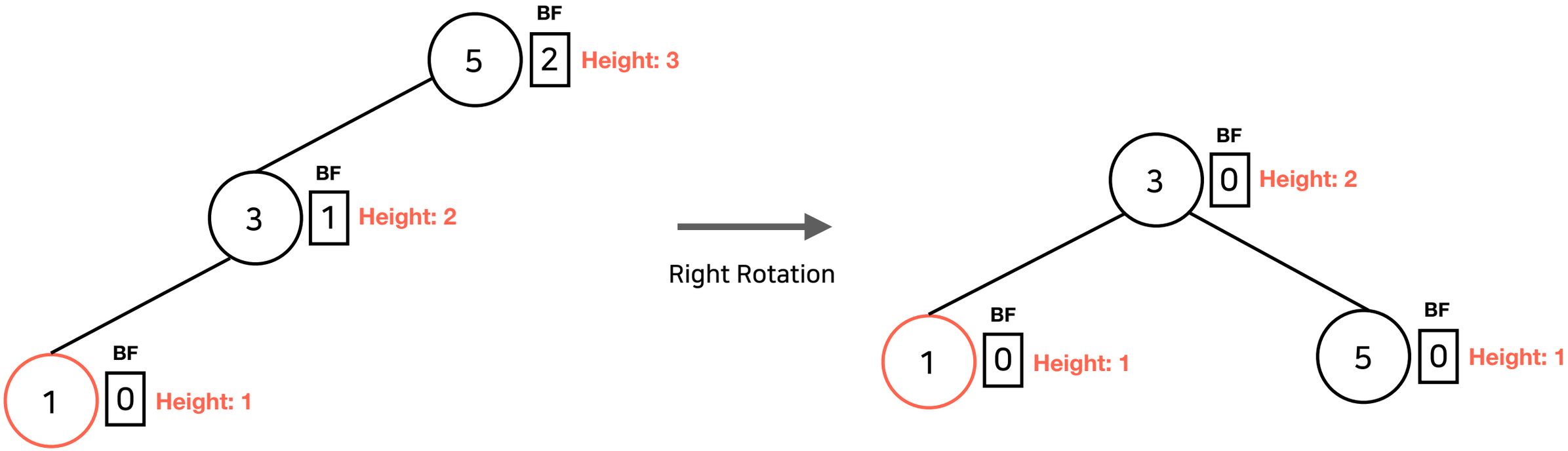
- Balance Factor = 왼쪽 자식 Node의 height - 오른쪽 자식 Node의 height
- AVL Tree는 모든 Node의 Balance Factor의 절대값을 2 미만으로 유지.

Rotation



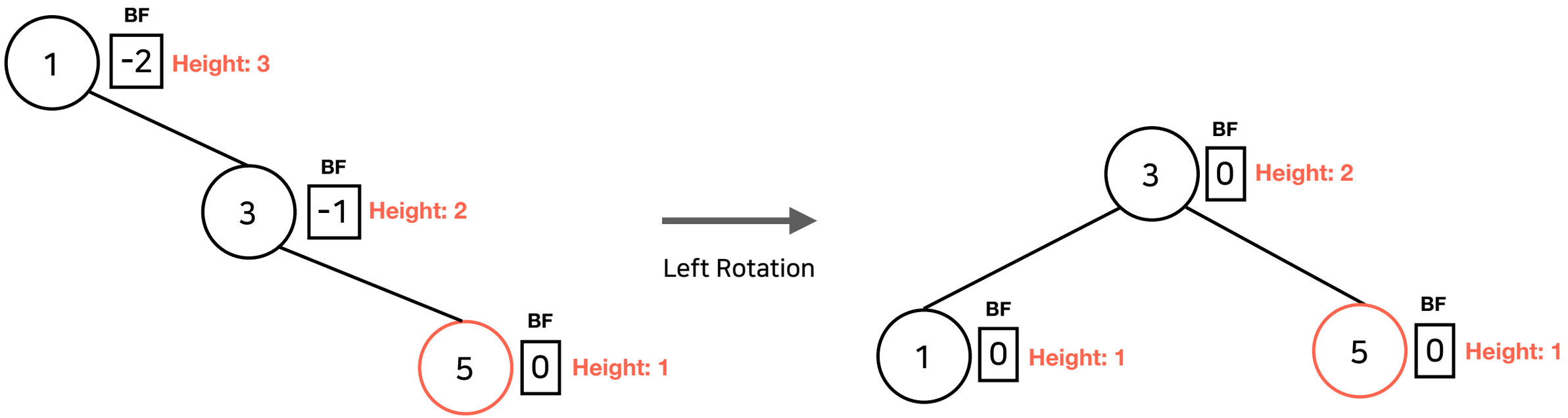
- Root Node를 잡아 당겨 Child Node 하나를 새로운 Root Node로 만든다는 느낌.
- AVL Tree의 Balancing할 때 가장 기본적인 동작이 Rotation.

Rebalance Scenario - LL



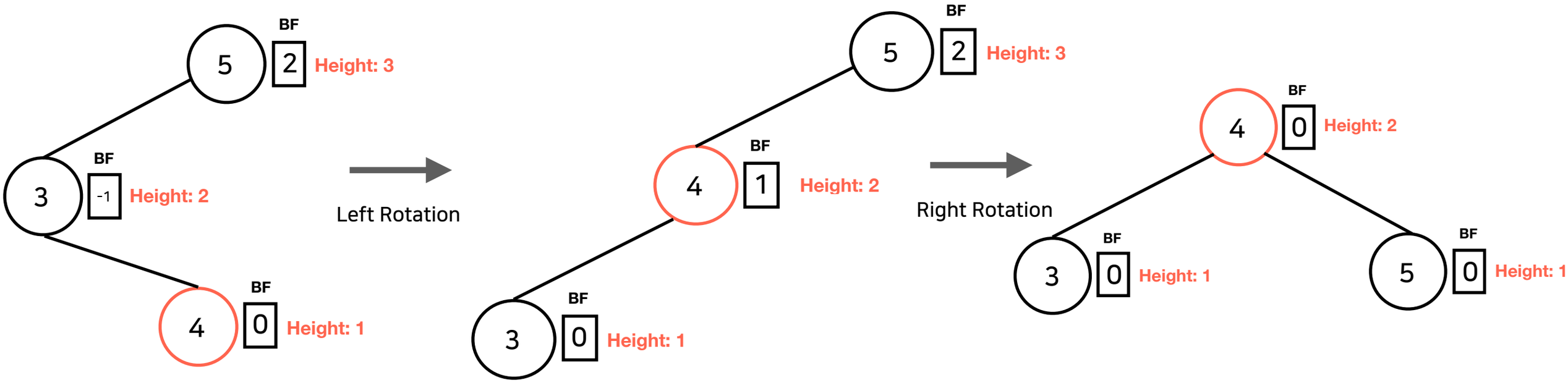
- 1번 Node을 삽입 할 때, 5번 Node의 Balance Factor가 2로 바뀌므로 Balance가 깨짐.
- Right Rotation을 1번 수행하면 모든 Node의 Balance Factor가 0으로 맞춰짐.

Rebalance Scenario - RR



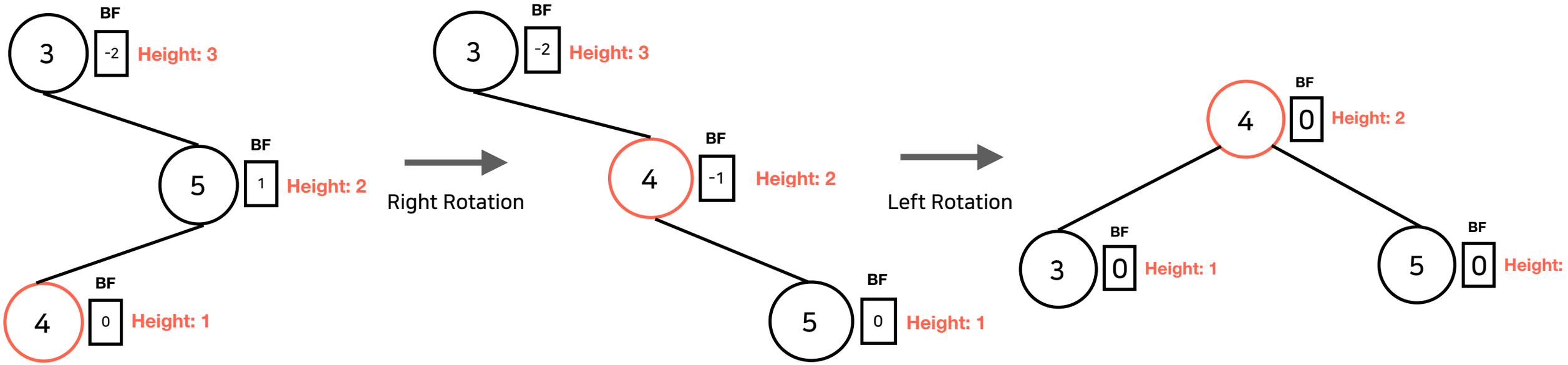
- 5번 Node을 삽입 할 때, 1번 Node의 Balance Factor가 2로 바뀌므로 Balance가 깨짐.
- Left Rotation을 1번 수행하면 모든 Node의 Balance Factor가 0으로 맞춰짐.

Rebalance Scenario - LR



- 1번 Node을 삽입 할 때, 5번 Node의 Balance Factor가 2로 바뀌므로 Balance가 깨짐.
- Right Rotation을 1번 수행하면 모든 Node의 Balance Factor가 0으로 맞춰짐.

Rebalance Scenario - RL



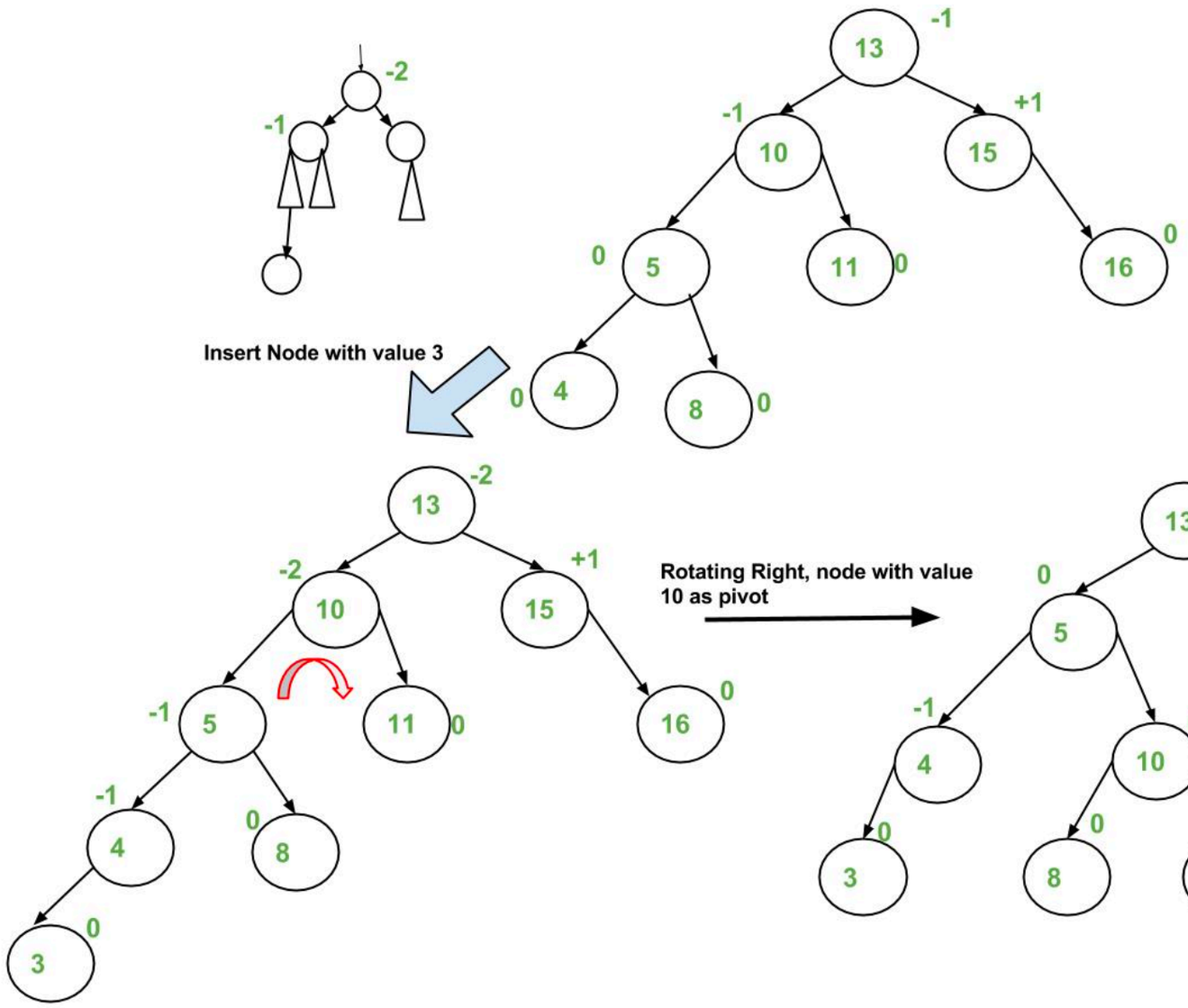
- 5번 Node을 삽입 할 때, 3번 Node의 Balance Factor가 2로 바뀌므로 Balance가 깨짐.
- Right Rotation을 1번 수행하면 모든 Node의 Balance Factor가 0으로 맞춰짐.

Rebalance

```
for (;node is not root; node = node->parent)
{
  if (abs(node->balance_factor) >= 2) // AVL Tree is unbalanced
  {
    if (node->balance_factor > 1) // Left subtree's height is greater than right subtree (LL + LR)
    {
      if (node->left_child->balance_factor < 0) rotate_left(node->left_child); // Only when LR
      rotate_right(node);
    }
    else if (node->balance_factor < -1) // Right subtree's height is greater than left subtree (RL + RR)
    {
      if (node->right_child->balance_factor > 0) rotate_right(node->right_child); // Only when RL
      rotate_left(node);
    }
  }
}
```

1. AVL Tree

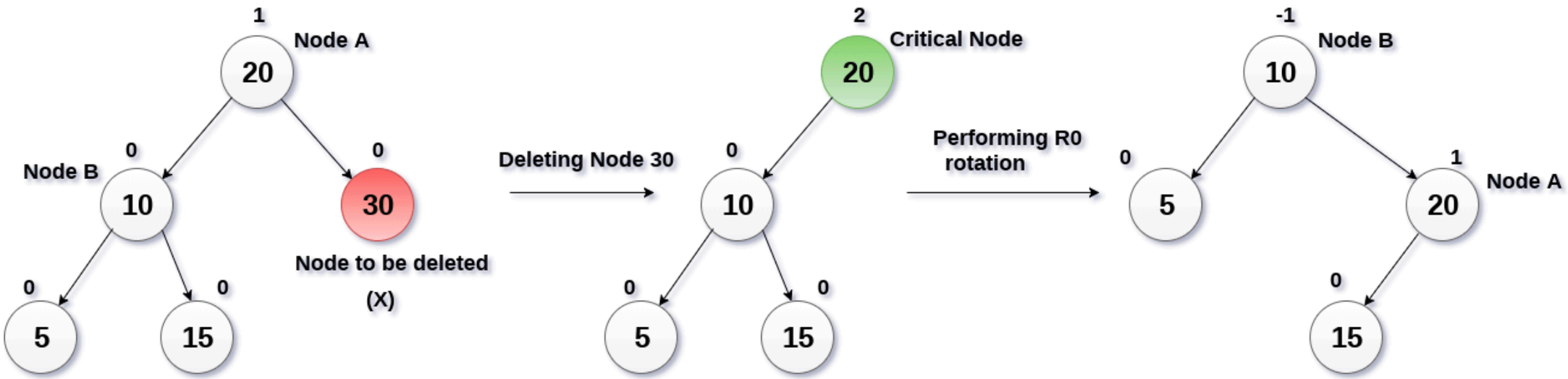
Insert



1. 기본 BST Insert 연산 수행.
2. 삽입한 위치부터 Rebalance 수행.

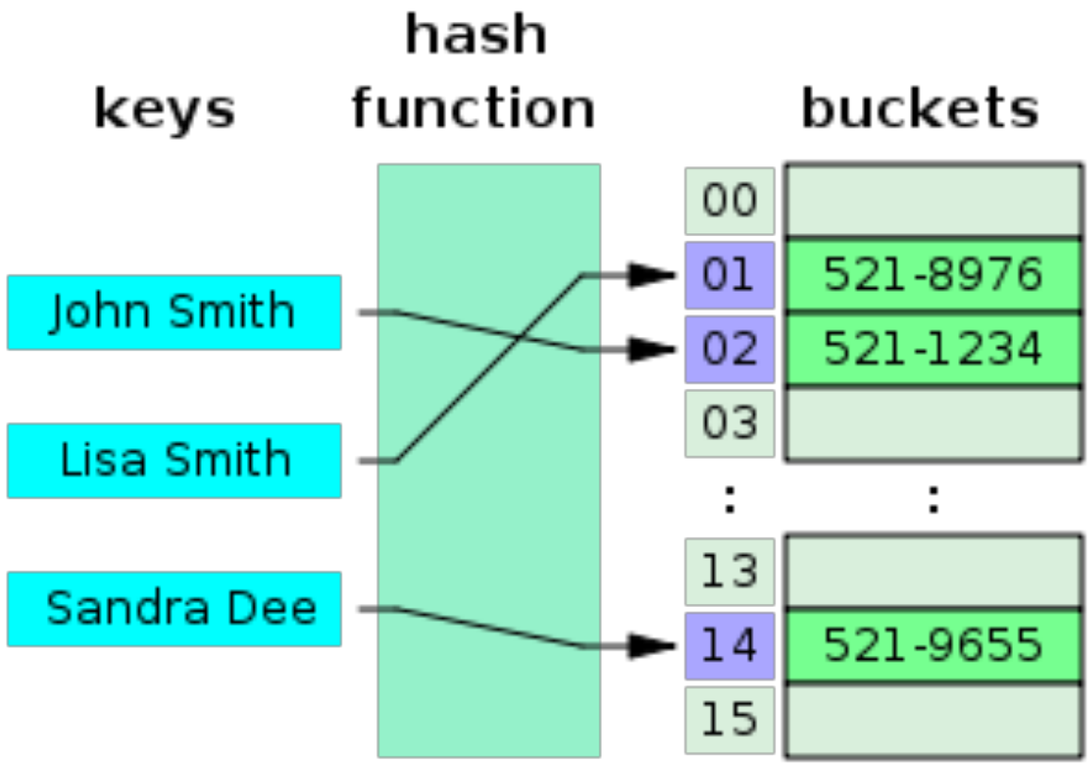
1. AVL Tree

Delete



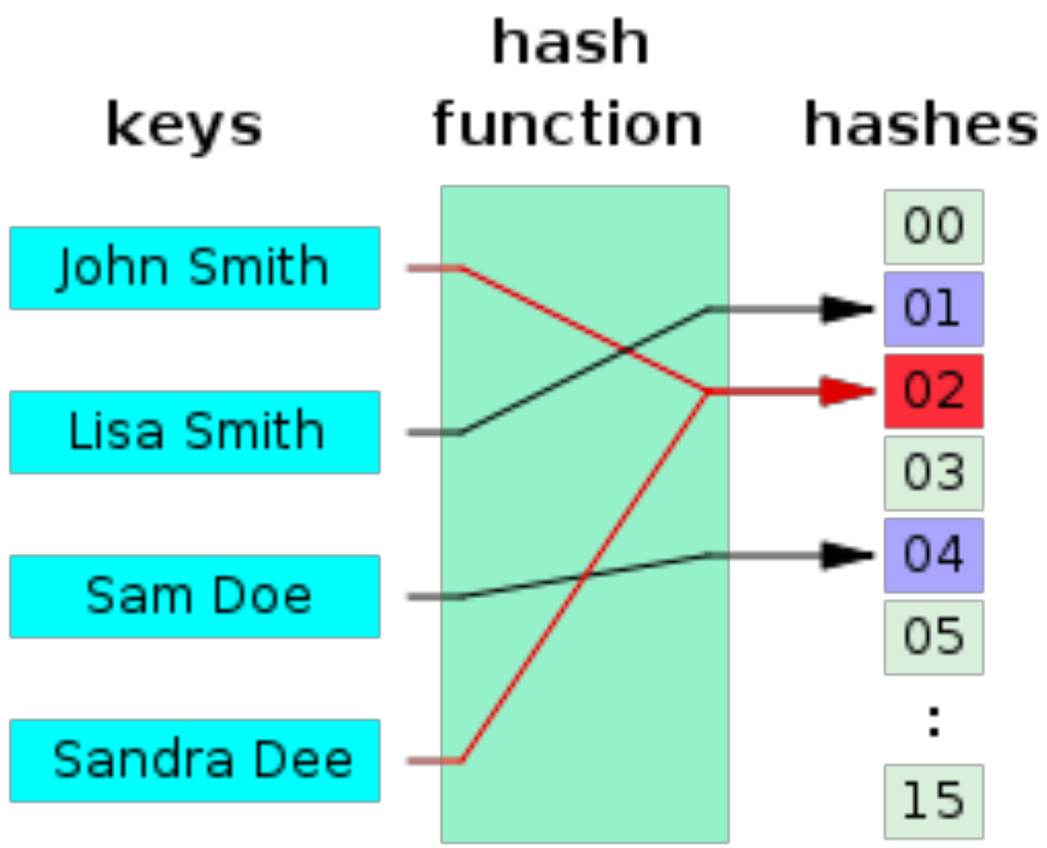
1. 기본 BST Delete 연산 수행.
2. 삭제한 Node의 위치를 찾아, 그 위치부터 Rebalance 수행.

Hash Table



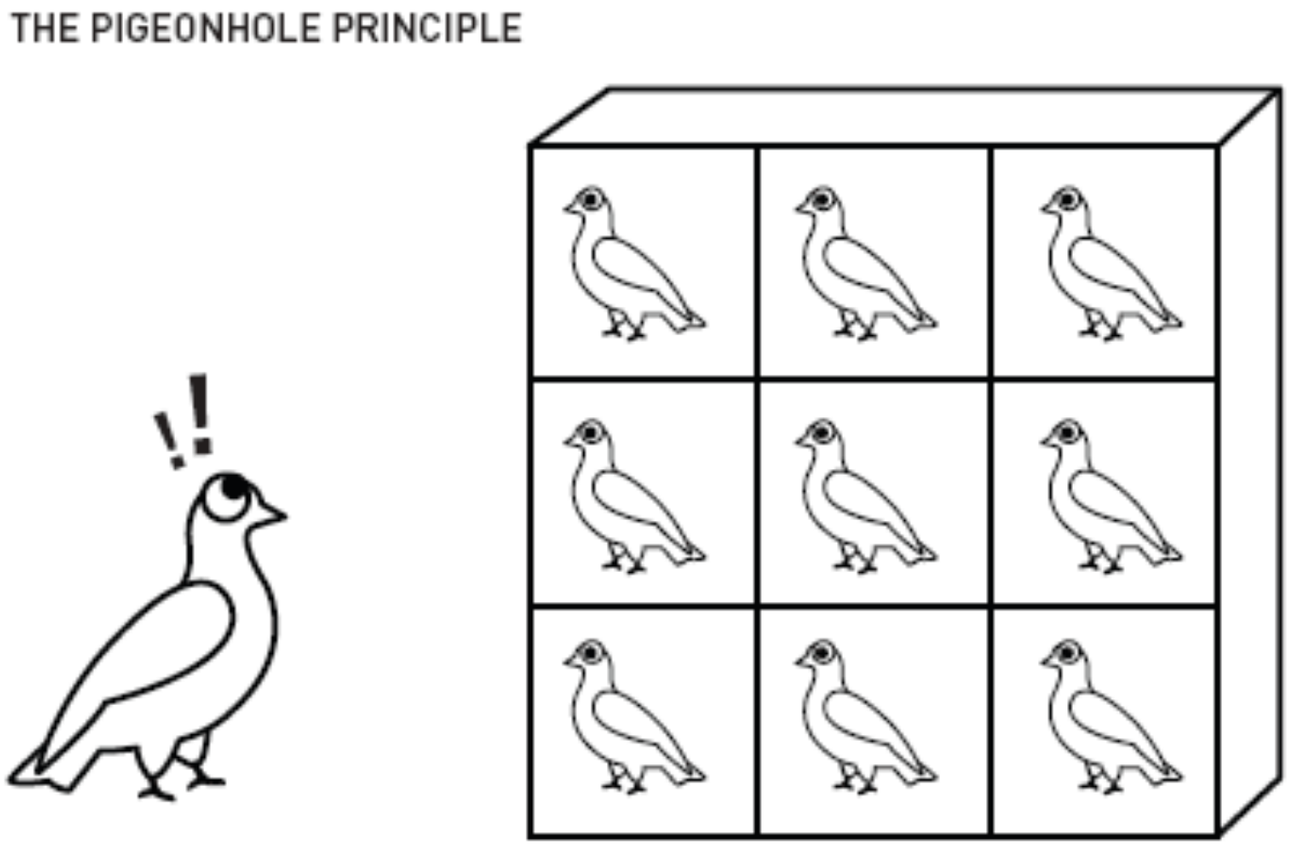
- Hash 함수 - 임의의 길이의 데이터에 대해 고정된 길이의 데이터로 매핑하는 함수.
- Hashing은 간단한 알고리즘으로 O(1)을 지향.
- 같은 입력값에 대해서는 같은 Hash 값을 보장.
- Hash 값을 가능한 한 고른 범위에 분포하도록 설계.
- Hash Table - Hash값을 index로 해서 데이터를 저장하는 자료구조.
- 버킷 - 데이터가 저장되는 부분.
- Load Factor - 입력된 데이터의 수 / Hash Table의 크기.

Hash Collision



- 서로 다른 데이터를 Hashing한 결과가 같을 경우.
- 무한한 데이터를 유한한 Hash값으로 변환하기 때문에, 충돌이 무조건 생김.

Hash Collision



[비둘기 집의 원리]

N개의 물품을 M개의 상자에 집어넣을 때 만약 n > m 이라면,
적어도 하나의 상자에는 [N / M] + 1 이상의 물품을 넣어야 한다.

↓

Hash Collision은 무조건 생긴다.

Hash Collision

사람이 N명 모였을 때, 생일이 같은 두명이 존재할 확률?

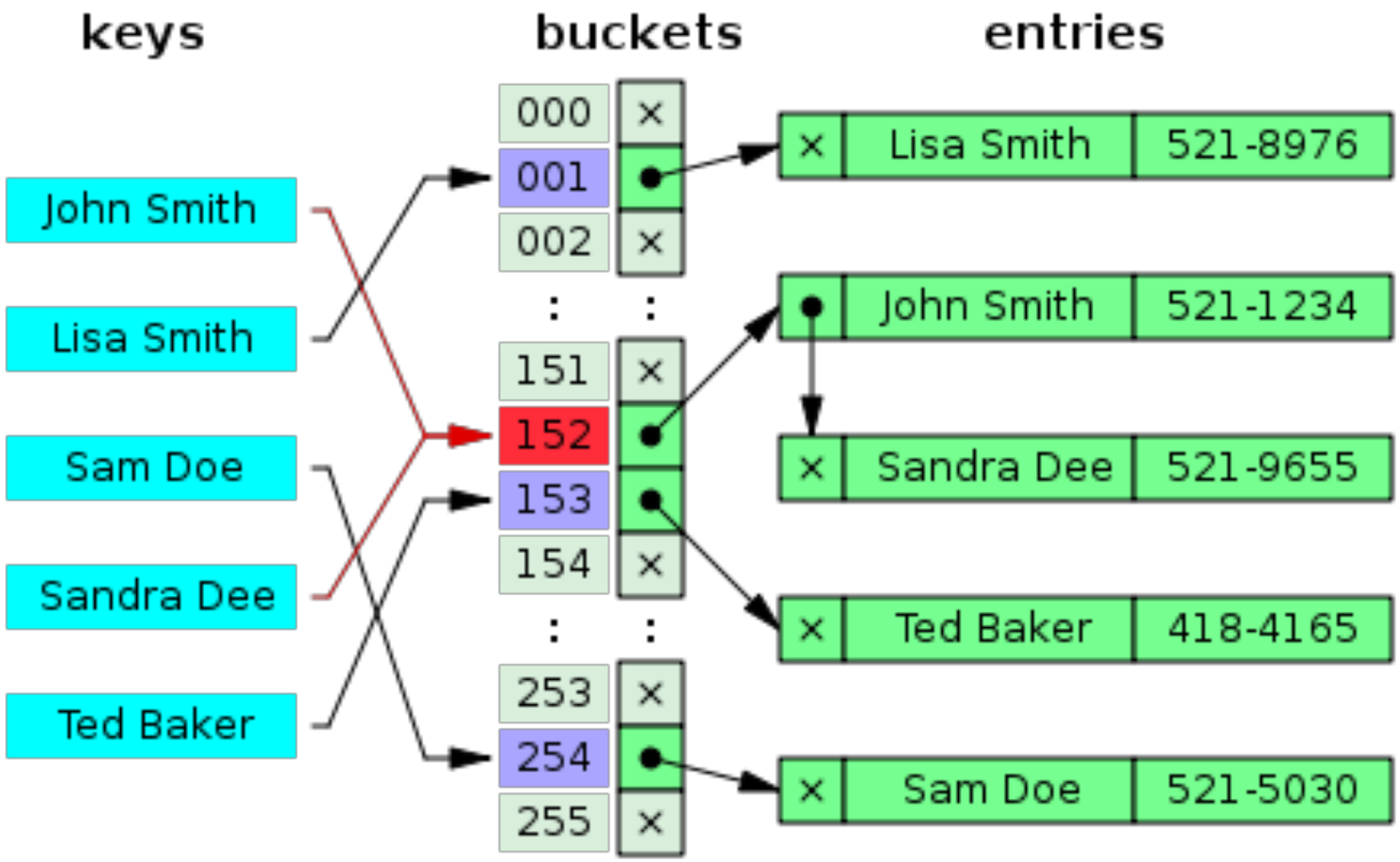
$$p(n) = 1 - \frac{365!}{365^n (365-n)!}$$

↓

입력하는 데이터가 많지 않아도 Hash Collision이 생길 확률이 높다.

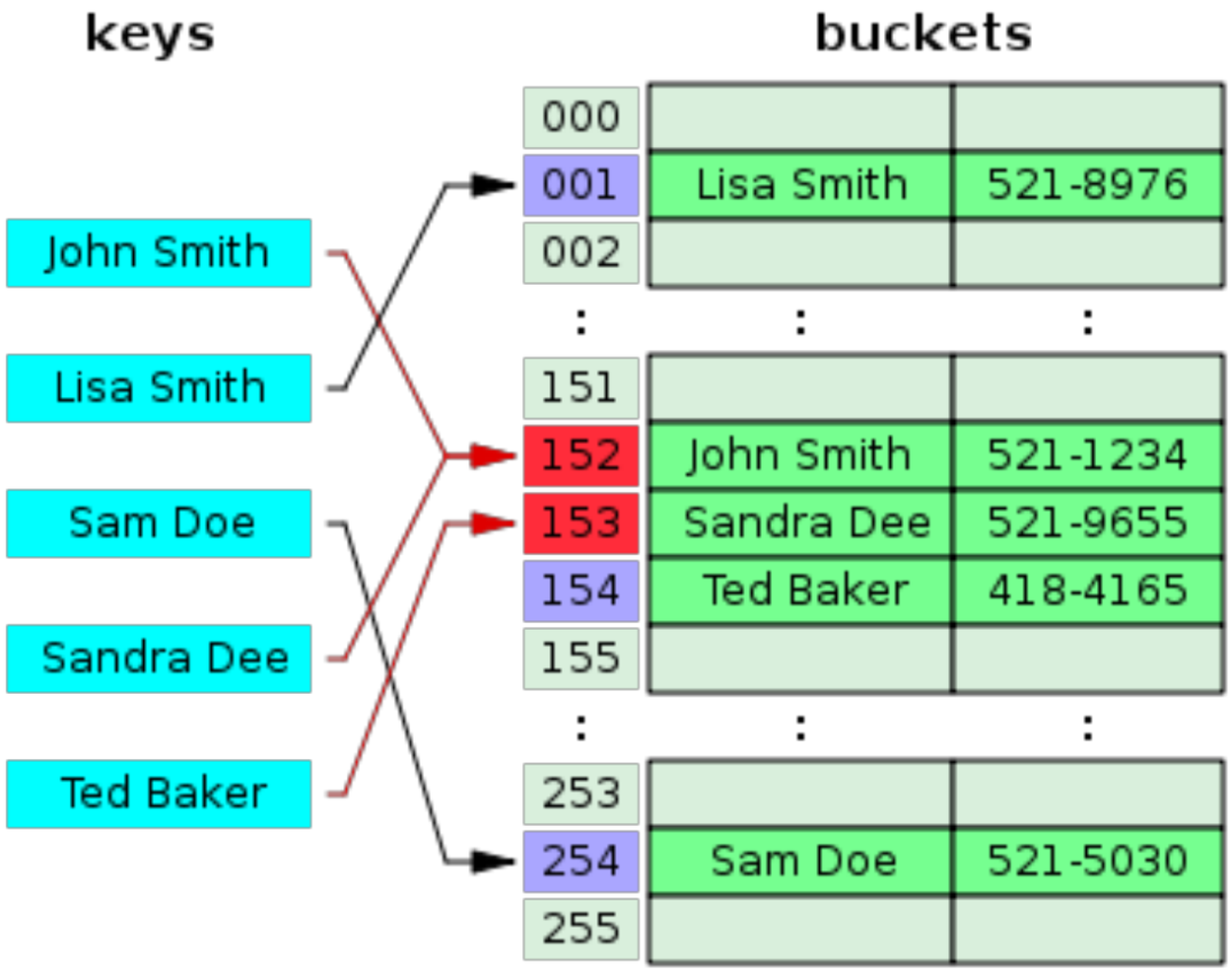
n	*p(n)*
1	0.0%
5	2.7%
10	11.7%
20	41.1%
30	70.6%
40	89.1%
50	97.0%
70	99.9%
100	99.99997%
366	100%

Separate Chaining



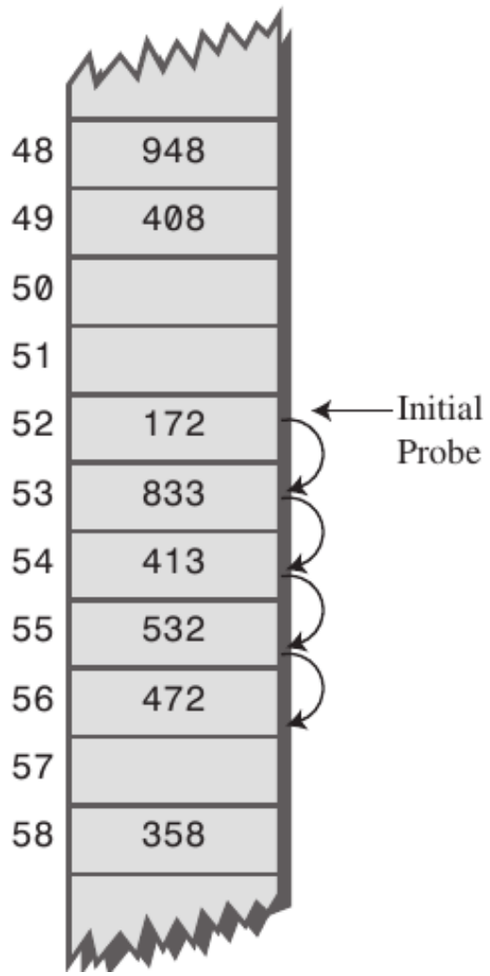
- 버킷에 들어갈 수 있는 엔트리의 수에 제한을 두지 않음.
- 추가하려는 버킷에 이미 데이터가 있다면 노드를 추가하여 다음 노드를 가르키는 방식.
- 대부분 연결리스트로 구현하나 Balancing Tree로 구현하기도 함.
- 삽입은 O(1), 탐색과 삭제는 O(Load Factor + 1)

Open Addressing



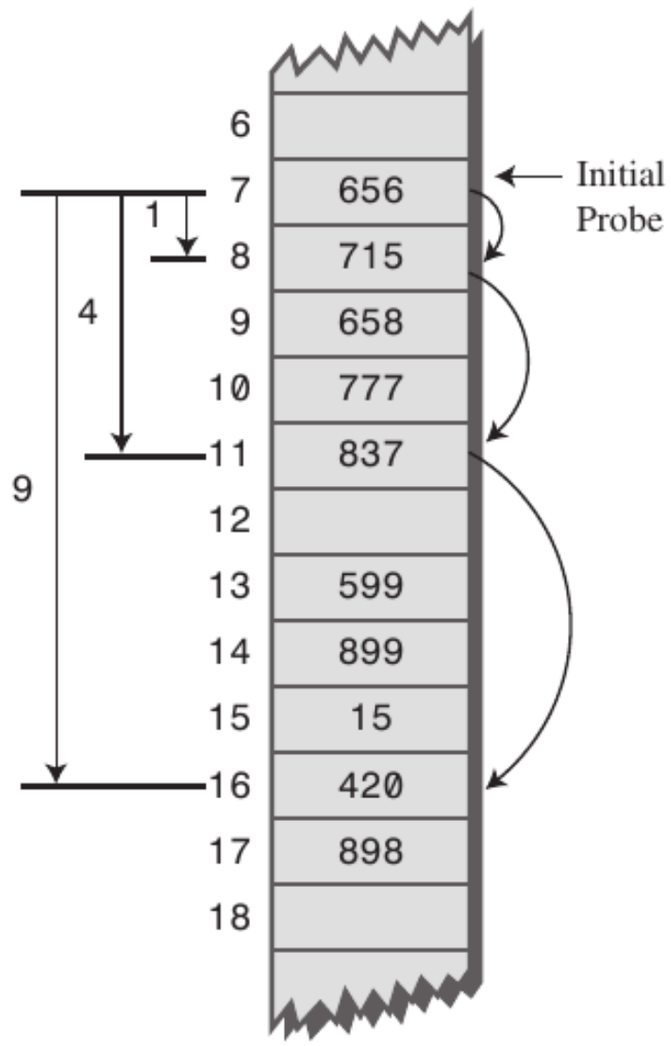
- 버킷에 들어갈 수 있는 엔트리의 수는 한개로 고정.
- Hash값을 index로 하는 버킷이 아닌 다른 위치에 저장하는 방식.
- 새로운 index를 정하는 일련의 과정을 probing이라고 함.

Linear Probing



- 최초 Hash값에 이미 다른 데이터가 저장돼있다면 이동.
- 이동하는 폭이 고정값 (위의 예시는 1)
- 빈 버킷을 만날 때 까지 반복.
- 특정한 Hash값 주변 버킷이 모두 채워져 있을 경우 취약.

Quadric Probing



- 최초 Hash값에 이미 다른 데이터가 저장돼있다면 이동.
- 이동하는 폭이 제곱수로 늘어남.
- 빈 버킷을 만날 때 까지 반복.
- 서로 다른 키들의 Hash값들의 초기값이 같은 경우 취약.

Double Hashing

- 탐사할 Hash값의 규칙성을 없애서 Linear, Quadric Probing의 취약성 해결.
- 초기 Hash값을 얻는 함수와, Probing할 때 이동폭을 구하는 Hash 함수를 준비.
- 최초 Hash값이 같더라도 이동폭이 달라지고, 이동폭이 같더라도 최초 Hash값이 달라짐.

- Example
- Hash 함수 : 3으로 나눈 나머지.
- 이동폭 결정 함수: 5로 나눈 나머지.
- 3과 6은 Hash값이 모두 0이지만, 이동폭은 3과 1로 달라짐.
- 6과 11을 이동폭이 모두 1이지만, Hash값은 0와 2로 달라짐.
- 단 나누는 수가 서로소이여야 효과를 냄.

AVL Tree 구현 과제

- AVL Tree를 구현한다.
- 프로그램은 5가지 Operation을 지원한다.
  1. Insert [i] - 정수를 하나 입력받아 AVL Tree에 삽입한다.
  2. Search [s] - 정수를 하나 입력받아 AVL Tree에 검색한다. 존재하지 않으면 X 출력.
  3. Delete [d] - 정수를 하나 입력받아 AVL Tree에서 해당 값을 삭제한다. 존재하지 않으면 X 출력.
  4. Print [p] - AVL Tree를 PreOrder 방식으로 순회하며 출력한다.
  5. Quit [q] - 프로그램을 종료한다.

입력예시 1

```
i 5
i 3
i 1
p
s 5
s 7
q
```

출력예시 1

```
□3 1 5
5
X
```

입력예시 2

```
i 7
i 3
i 5
i 9
d 3
p
q
```

출력예시 2

```
□7 5 9
```

3. 과제

AVL Tree 구현 과제

- AVL Tree 구현 과제를 한다.
- 문제를 푼 뒤, 코드를 메신저로 제출해서 문제의 정답여부를 먼저 확인한다.
- 문제를 맞췄을 시, 정답 코드를 Github Repository에 Pull Request를 보낸다.
- 기한은 2019년 7월 29일 17시 59분까지.
- 샘플코드는 기한 마감 18시간전 공개한다.
- 기한내 제출 못할 시 벌금있음.

4. References

https://en.wikipedia.org/wiki/Hash_table

https://ratsgo.github.io/data%20structure&algorithm/2017/10/25/hash/

https://en.wikipedia.org/wiki/Hash_function