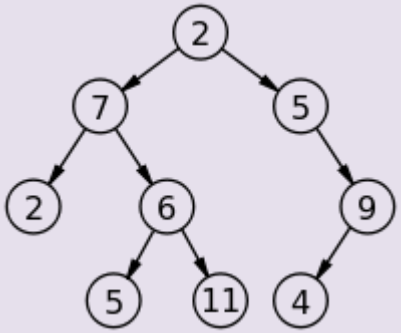
4. Data Structure #2 - Tree

2017010698
수학과 오서영

트리 (Tree)

: 선형으로 표현하기 어려운 형태의 자료 -> 계층구조
ex) 회사 조직도, 상품 분류기준
-> 계층적 구조를 갖는 자료를 표현하기 위한 자료구조 : **트리**



위 : 상위개념, 아래 : 하위개념

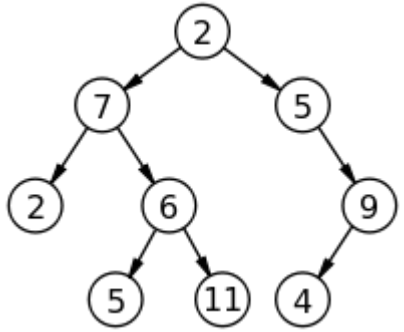
-> 어떤 개념이 다른 개념을 포함하면
두 개념을 **상위-하위로** 연결한다

탐색형 자료구조로 유용하게 쓰임
-> 특정한 조건을 지키도록 구성된 트리를 이용하면,
배열이나 리스트를 사용하는 것 보다 더 빠르게 작업가능

트리 구성요소

트리 : 자료가 저장된 **노드**(Node)들이
간선(Edge)으로 서로 연결
되어있는 자료구조

노드

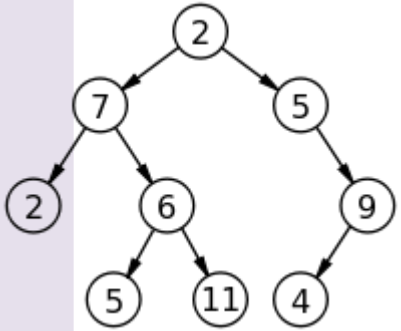


노드간에는 상, 하위 관계가 존재
- 상위노드 : **부모**(parent) 노드
- 하위노드 : **자식**(child) 노드
- 부모 노드가 같은 두 노드 : **형제**(sibling) 노드

트리에서 한 노드는 여러 개의 자식을 가질 수 있어도,
부모는 하나만 가질 수 있다
-> 다른 모든 노드들을 자손으로 갖는 단 하나의 노드 : **루트**(root)
-> 자손이 하나도 없는 노드 : **리프**(leaf)

트리 노드의 속성

깊이(depth) : 루트에서 어떤 노드에 도달하기 위해 거쳐야 하는 간선의 수
높이(height) : 트리에서 가장 깊숙히 있는 노드의 깊이



트리의 재귀적 성질

트리에서 한 노드와 그의 자손들을 모두 모으면
그들도 하나의 트리가 됨 -> **서브트리**

트리의 표현

각 노드를 하나의 구조체/객체로 표현하고, 이들을 서로 포인터로 연결

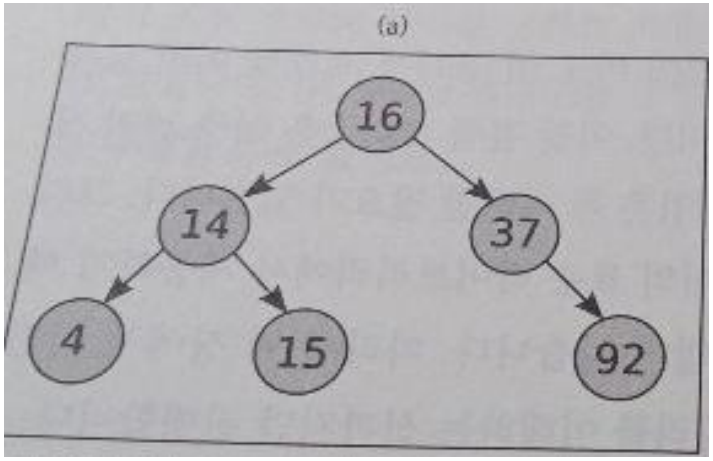
```
Struct TreeNode {
    string label;  // 저장할 자료(꼭 문자열x)
    TreeNode* parent; // 부모노드를 가리키는 포인터
    vector<TreeNode*> children; // 자손노드들을 가리키는 포인터
};
```

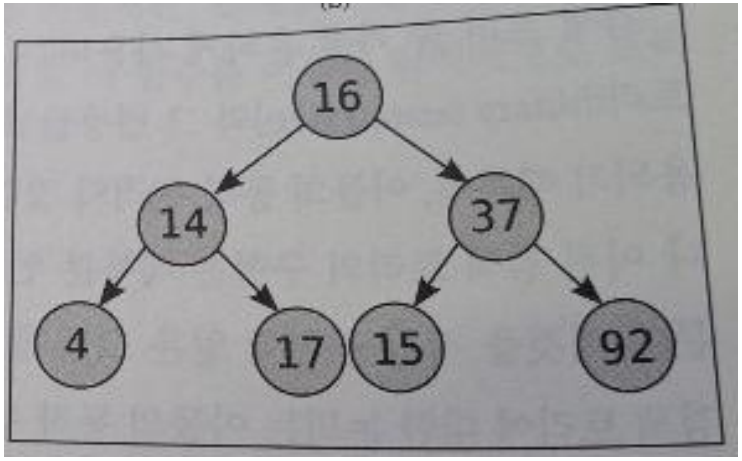
이진 검색 트리(Binary search tree)

검색트리 : 자료를 일정한 순서에 따라 정렬한 상태로 저장해둠
(리스트, 큐 : 주어진 순서에 따라 자료배치)

이진트리 : 각 노드가 왼쪽과 오른쪽,
최대 두개의 자식노드만을 가질 수 있는 트리
-> 자식 노드들의 배열 대신 두개의 포인터 left, right를 담는 객체로 구현됨

이진 검색트리 : 각 노드의 왼쪽 서브트리에는 해당 노드의 원소보다 작은 원소를 가진 노드들이, 오른쪽 서브트리에는 큰 원소를 가진 노드들이 들어감





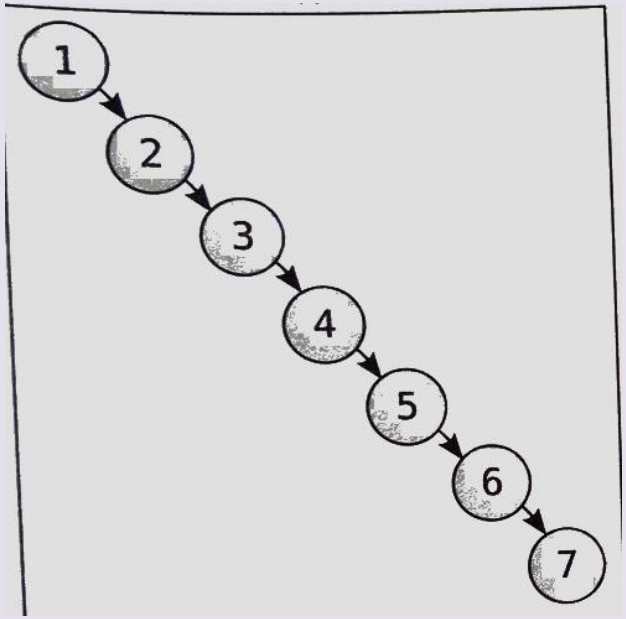
이진 검색 트리(Binary search tree)

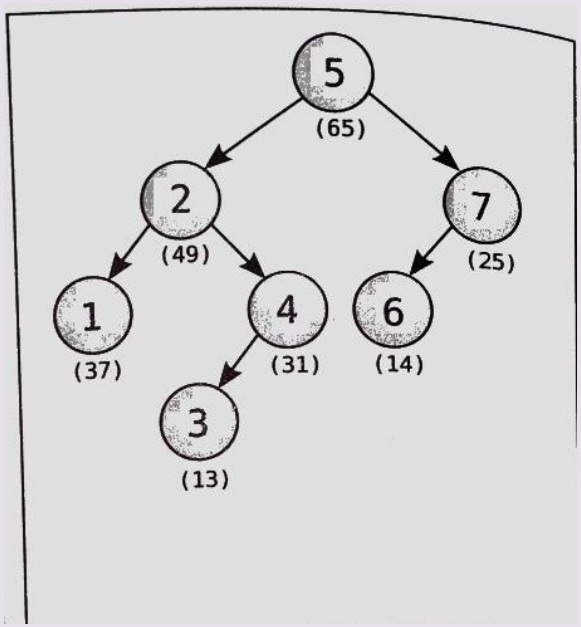
1. 새 원소를 삽입하기 위해서, 삽입할 위치를 우선 찾고 그 이후 원소들을 모두 한 칸씩 뒤로 옮길 필요 X
-> 이진 검색 트리는 선형적인 구조의 제약이 없기 때문에, 새 원소가 들어갈 위치를 찾고 거기에 노드를 추가하기만 하면됨

2. 이진 검색 트리에 대한 모든 연산은, 모두 루트에서부터 한 단계씩 트리를 내려 가며 재귀호출을 통해 수행되므로, 최대 재귀 호출의 횟수는 트리의 높이 h 와 같다. -> 모든 연산의 시간복잡도 : O(h)

트립(Treap)

: 입력이 특정 순서로 주어질 때 그 성능이 떨어진다는 이진 검색 트리의 단점을 해결 -> **랜덤화**된 이진 검색 트리
-> 트리의 형태가 원소들의 추가 순서에 의해 결정되지 않고 **난수**에 의해 결정됨
-> 원소들이 순서대로 추가되고 삭제되더라도 트리 높이의 기대치는 일정
-> 트립은 새 노드가 추가 될때마다 해당 노드에 우선순위를 부여(난수)





O(N)

O(logN)

힙(Heap)

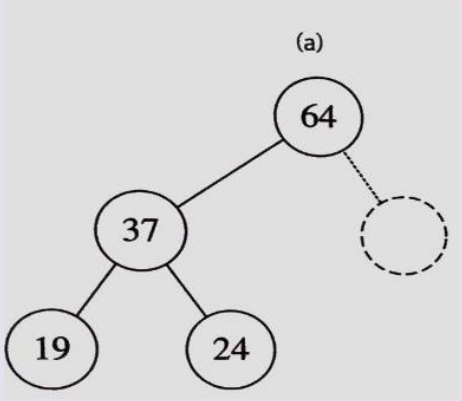
: 특정한 규칙을 만족하도록 구성된 이진 트리, 단순히 최대 원소를 가능한 한 빠르게 찾을 수 있는 방법으로 설계됨 -> 단순, 효율

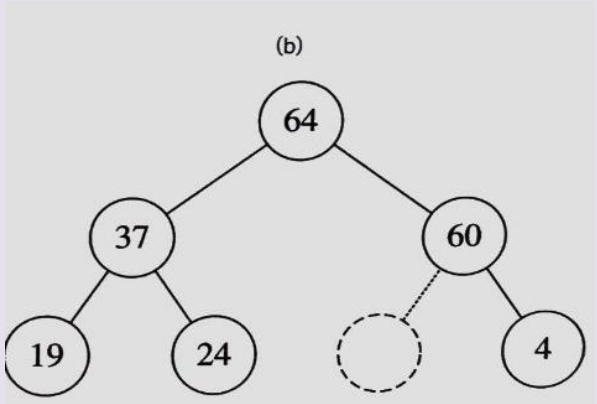
- 힙의 대소관계 규칙

: 부모 노드가 가진 원소는 항상 자식 노드가 가진 원소 이상이다
-> 트리에서 가장 큰 원소는 항상 트리의 루트에 들어가므로,
최대 원소를 빨리 찾을 수 있음

- 힙의 모양 규칙

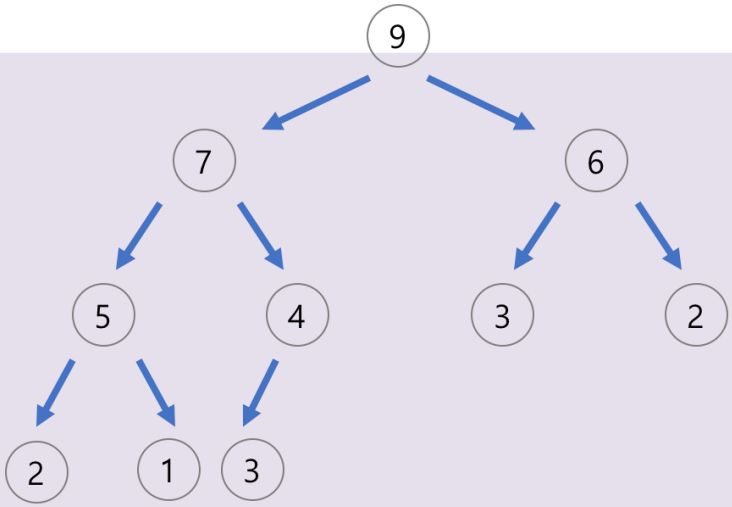
1. 마지막 레벨을 제외한 모든 레벨에 노드가 꽉 차있어야 한다
2. 마지막 레벨에 노드가 있을 때는
항상 가장 왼쪽부터 순서대로 채워져있어야 한다



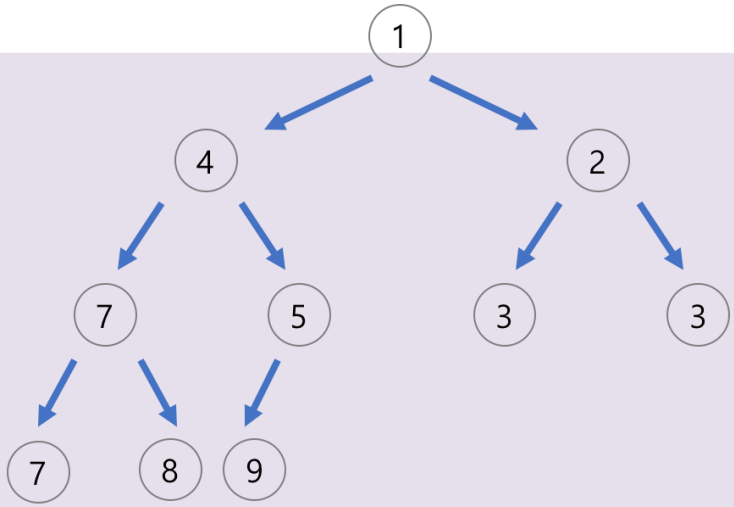


힙(heap)의 종류

1. 최대 힙(max heap)
부모 노드의 키 값이 자식 노드의 키 값보다 크거나 같은 완전 이진 트리
key(부모 노드) >= key(자식 노드)
2. 최소 힙(min heap)
부모 노드의 키 값이 자식 노드의 키 값보다 작거나 같은 완전 이진 트리
key(부모 노드) <= key(자식 노드)



-최대 힙(max heap)-



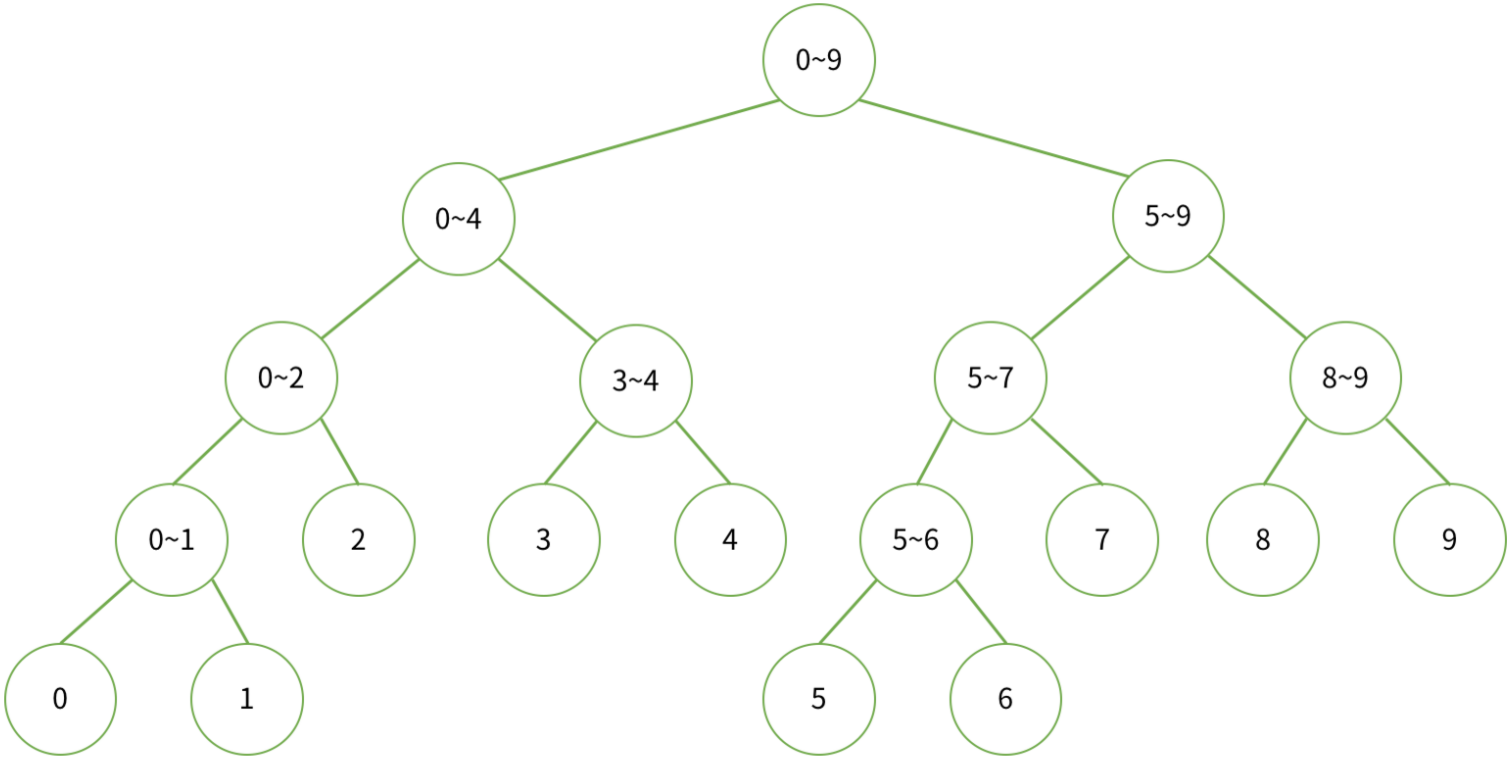
-최소 힙(min heap)-

세그먼트 트리(Segment Tree)

: 배열에 부분 합을 구할 때 사용하는 개념
이 때 문제는 배열의 값이 지속적으로 바뀔 수 있기 때문에 매 순간 배열의 부분 길이 만큼, 즉 O(N) 만큼의 시간이 걸리기 때문에
이를 트리로 구현하여 O(logN) 의 시간으로 해결하는 방법

주어진 배열을 이진 트리 구조로 만들어야 함
1. 부모노드의 값은 양 쪽 자식 노드 값의 **합**
2. 배열의 요소들은 리프 노드에 위치
-> N = 10 일 때의 세그먼트 트리

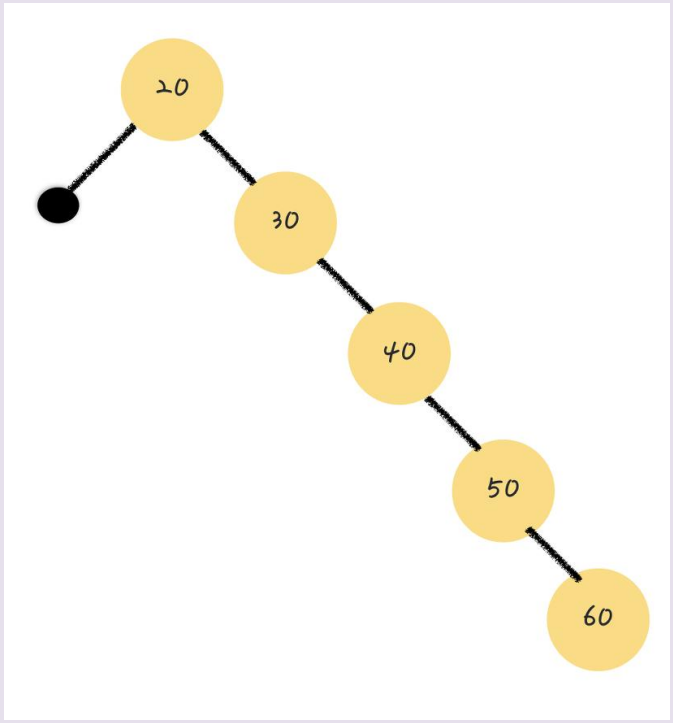
N = 10인 경우에 세그먼트 트리는 다음과 같습니다.



기존 데이터 배열의 크기를 N 이라 하면,
리프 노드의 개수가 N 이 되고, 트리의 높이 H 는 [logN] 이 되고,
배열의 크기는 2^(H+1) 이 됩니다.

Red Black Tree

아래와 같은 모양이 나오지 않도록 조건이 걸림
-> 균형잡힌 트리 -> **O(logn)의 시간복잡도**

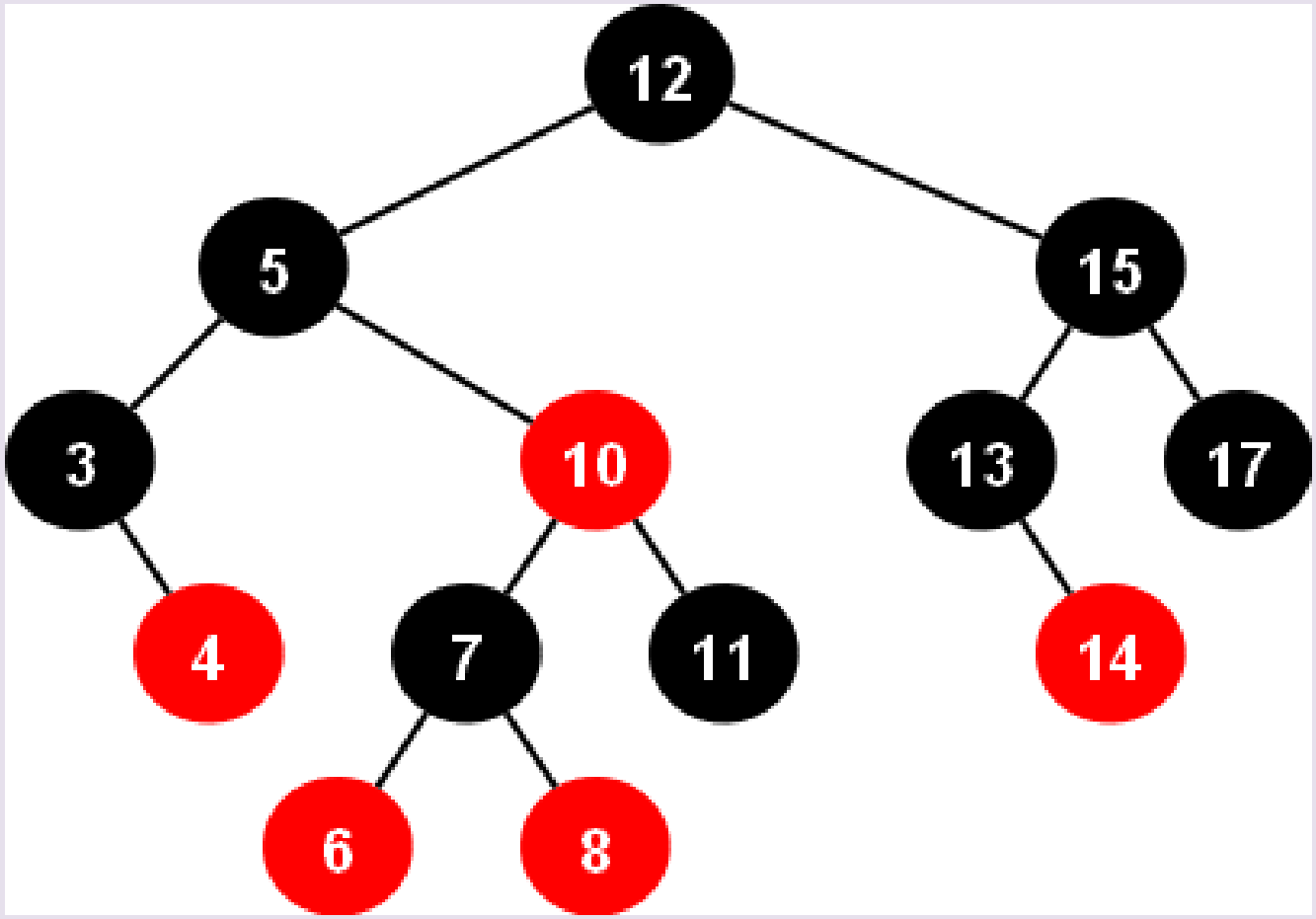


이진 탐색 트리
최악의 경우 -> O(n)

Red Black Tree

1. **Root Property** : 루트노드의 색깔은 **검정(Black)**이다.
2. **External Property** : 모든 **external node들은 검정(Black)**이다.
3. **Internal Property** : 빨강(Red)노드의 자식은 **검정(Black)**이다.
   -> No Double Red(빨간색 노드가 연속으로 나올 수 없다.)
4. **Depth Property** : 모든 리프노드에서 **Black Depth는 같다.**
   -> 리프노드에서 루트노드까지 가는 경로에서 만나는
   블랙노드의 개수는 같다.
   (그냥 노드의 수는 다를 수 있음)

-> 이진탐색트리인 **Red Black** 트리의 높이를 logn에 바운드되도록 해줌
-> 루트 노드로 부터 가장 먼 경로 까지의 거리가, 가장 짧은 경로 까지의 거리의 두배보다 항상 작아짐


12
5
15
3
10
13
17
4
7
11
14
6
8

Reference

[1] 구종만,『 프로그래밍 대회에서 배우는 알고리즘 문제 해결 전략 』, 인사이트(2012)
[2] [위키백과] 트리구조
https://ko.wikipedia.org/wiki/%ED%8A%B8%EB%A6%AC_%EA%B5%AC%EC%A1%B0
[3] [백준] 세그먼트 트리
https://www.acmicpc.net/blog/view/9
[4] 알고리즘) Red-Black Tree
https://zeddios.tistory.com/237