기술면접 – 이진탐색트리(Binary Search Tree)

이진트리

- 이진트리(Binary Tree)는 각 노드가 **최대 두 개의 자식 노드를 가지는 트리 구조**를 말합니다.
- 이진트리는 데이터를 구조화하거나 표현하기 위해 사용됩니다.
- 이진트리에서는 **모든 노드가 최대 두 개의 자식 노드를 가지기** 때문에 각 노드는 왼쪽 서브트리와 오른쪽 서브트리를 가질 수 있습니다.

이진탐색

- **이진 탐색(이분 탐색) 알고리즘은 정렬되어 있는 리스트에서 탐색 범위를 절반씩 좁혀가며 데이터를 탐색하는 방법**이다.
- 이진 탐색은 배열 내부의 **데이터가 정렬되어 있어야만 사용할 수 있는 알고리즘**이다.
- 변수 3개(start, end, mid)를 사용하여 탐색한다. 찾으려는 데이터와 중간점 위치에 있는 데이터를 반복적으로 비교해서 원하는 데이터를 찾는 것이 이진 탐색의 과정이다.
- 이진 탐색은 **정렬된 리스트에만 사용**할 수 있다는 **단점**이 있지만, 검색이 반복될 때마다 검색 범위가 절반으로 줄기 때문에 **속도가 빠르다는 장점**이 있습니다.

- 이진 탐색의 동작 방식은 다음과 같습니다.

1. 배열의 중간 값을 가져오고, 중간 값과 검색 값을 비교합니다. 중간 값이 검색 값과 같다면 종료합니다. (mid = key)
2. 중간 값보다 검색 값이 크다면 중간값 기준 배열의 오른쪽 구간을 대상으로 탐색합니다. (mid < key)
3. 중간 값보다 검색 값이 작다면 중간값 기준 배열의 왼쪽 구간을 대상으로 탐색합니다. (mid > key)
4. 값을 찾거나 간격이 비어있을 때까지 반복합니다.

이진탐색트리

- 이진탐색트리(Binary Search Tree)는 이진트리의 일종으로
- **왼쪽 서브트리의 모든 노드는 해당 노드보다 작은 값을 갖고, 오른쪽 서브트리의 모든 노드는 해당 노드보다 큰 값을 갖습니다.**
- 따라서 이진탐색트리는 데이터를 저장하고 검색하기 위한 자료구조로 사용됩니다.
- 이진탐색트리는 말 그대로 이진 탐색에 트리의 개념이 더해진 것으로 **특정 값을 탐색하는 것에 빠른 성능**을 보여줍니다. **특정 데이터를 아주 빠른 시간 평균(O(logN)에 최악의 경우 O(N)(편향된 그래프)**에 찾을 수 있습니다.
- 트리나 그래프같은 **비선형 자료구조**이다.

이진탐색 vs 이진탐색트리

- **이진탐색은 정렬된 배열**을 사용하는 것이고
- **이진탐색트리는** 그래프나, 트리(그래프의 특수한 형태 중 하나로, 트리구조로 배열된 일종의 계층적 데이터의 집합)같은 자료구조가 일렬로 나열하지 않고, **자료 순서나 관계가 복잡한 비선형 구조이다.**

- 그러면, **이진 탐색트리는 이진 탐색을 사용하는 배열 자료구조보다 나은 점**이 무엇일까?
- **배열은 조회에 강점이 있는 자료구조 입니다. 그렇다 보니, 검색만 한다고 가정할 때는 트리 구조가 굳이 필요하지 않습니다.**
- 하지만, 배열의 원소가 **삽입/삭제** 될 수 있다고 가정하면 배열 자료구조는, 삽입/삭제 될 Index 를 찾은 후, 그 Index 뒤에 위치한 원소들을 한 칸 씩 전부 뒤로 이동해야 하는 수고가 존재합니다. 또한, 배열공간이 모자라, 배열 크기를 증가시키거나 혹은 공간이 너무 남아, 축소시키는 작업은 덤으로 발생할 수 있습니다.
- 그에 반면에, **트리 구조는 참조(포인터)만 변경해주면 되므로, 원소의 대량 이동이 필요 없고, 공간 복잡도를 신경안써도 되는 장점이 있습니다. 그렇다보니, 삽입/삭제 연산이 한번이라도 존재한다면, 이는 삽입이 연속적으로 N번 발생할 수도 있다는 의미이므로 굳이 배열 자료구조를 사용할 필요가 없습니다.**
- 사실 조회 자체도 이진 탐색 구조상 트리도 특정 노드의 Index 검색이 아닌, 특정 노드로부터 순차대로 검색하는 구조이므로 배열이랑 크게 차이가 없습니다.
- 예를 들어, 연결 리스트는 특정 Index 노드 검색 시간복잡도는 O(n)이지만, 전체 순회를 할 때는 각 노드의 참조 값을 순차대로 탐색하므로 각 노드의 검색 시간이 O(1) 입니다.
- 결론적으론, 이진 탐색트리가 더 보편적으로 사용될 수 있다고 말할 수 있겠다.

이진트리 vs 이진탐색트리

- 따라서 이진트리와 이진탐색트리의 가장 큰 차이점은,
- **이진탐색트리에서는 모든 노드가 왼쪽 서브트리의 값보다 크고 오른쪽 서브트리의 값보다 작다는 규칙을 따른다는 것입니다.**

이진탐색트리 조건

- **부모 노드의 왼쪽 노드는 부모 노드보다 작아야 한다.**
- **부모 노드의 오른쪽 노드는 부모 노드보다 커야 한다.**

이진탐색트리 삽입

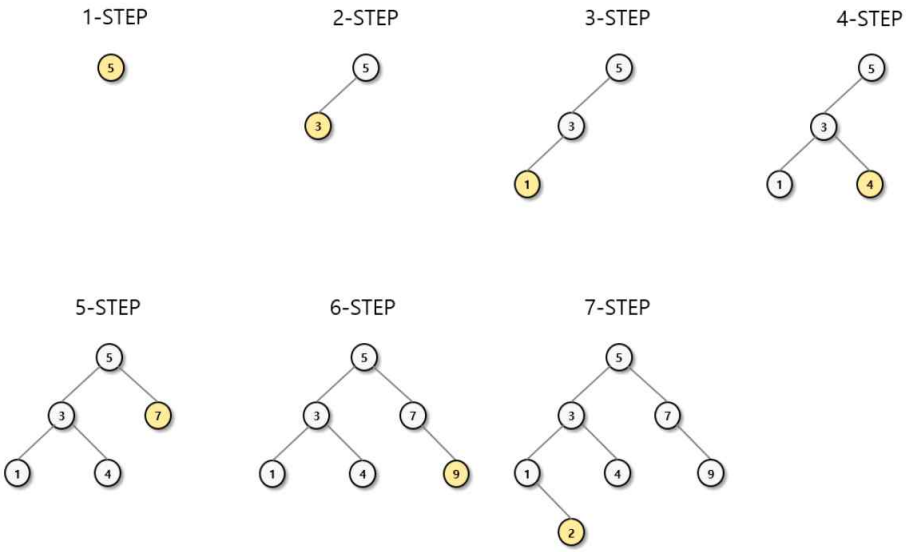
- 루트 노드가 비었다면, 삽입 노드를 루트 노드로 채웁니다.
- 루트 노드가 존재한다고 가정할 때, 삽입 노드의 값과 비교하여 삽입 노드의 값이 작으면 루트 노드의 오른쪽, 크면 오른쪽으로 이동합니다.
- 이동시, 노드가 비었다면 삽입 노드로 채웁니다.
- 만약 노드가 존재한다면, 2번~3번 과정을 반복합니다.

이진 탐색트리 에 노드를 넣는 순서는 다음과 같다고 가정합니다.

5, 3, 1, 4, 7, 9, 2

위에서 설명한 규칙대로 삽입과정을 거칩니다.

검색과 마찬가지로 이동하며, 자리를 찾아 노드를 저장합니다.

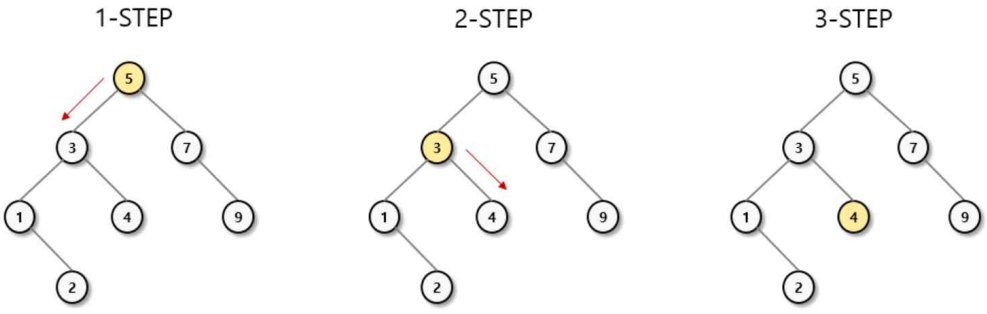


이진탐색트리 검색

- 루트 노드가 비어있으면 실행하지 않는다.
- 루트 노드부터 검색 노드의 위치를 찾아간다. 현재 노드에서 작으면 왼쪽, 크면 오른쪽으로 이동한다
- 검색노드와 일치하는 노드를 만나면 리턴한다.

예시의 이진 탐색트리 에서 4 노드를 검색한다고 가정해봅시다.

첫번째로 만나는 노드는 5 입니다. 검색노드가 5 보다 작으므로, 왼쪽으로 이동합니다.
두번째로 만나는 노드는 3 입니다. 검색노드가 3 보다 크므로, 오른쪽으로 이동합니다.
세번째로 만나는 노드는 4 입니다. 검색노드와 일치 하므로 해당 노드를 리턴 합니다.



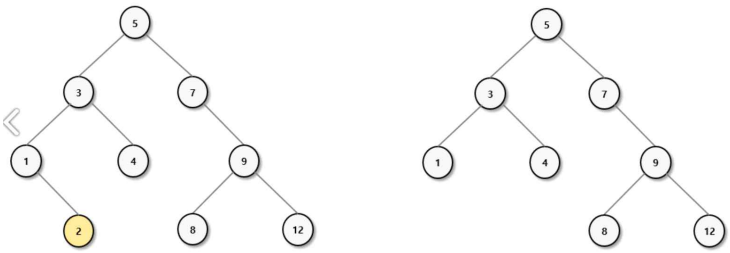
이진탐색트리 삭제

- 이진 탐색트리의 검색/삽입 연산과 달리 삭제는 조금 복잡합니다. 고려해야 할 케이스를 3개로 나눌 수 있습니다.

- 삭제할 노드가 말단 노드인 경우(자식노드가 없는 경우)
- 삭제할 노드의 자식 노드가 1개인 경우
- 삭제할 노드의 자식 노드가 2개인 경우

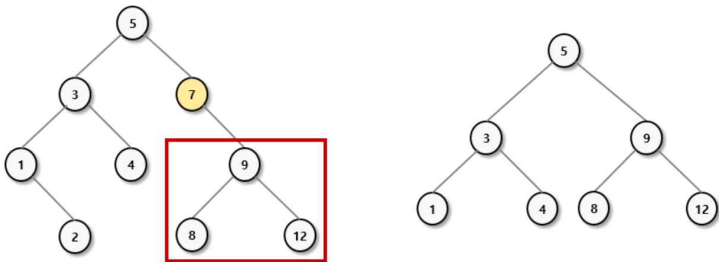
1. 삭제할 노드가 말단 노드인 경우
삭제할 노드만 찾아 삭제해주면 됩니다.

아래의 예시는 2 노드를 지웁니다.



2. 삭제할 노드의 자식 노드 1개인 경우
삭제할 노드의 자식들의 트리 구조 규칙은 삭제할 노드의 조상으로 부모를 바꾸더라도 규칙이 깨지지 않기 때문에 단순히 지우고 노드 연결구조만 바꾸면 된다.

이래의 예시는 7 노드를 지웁니다.

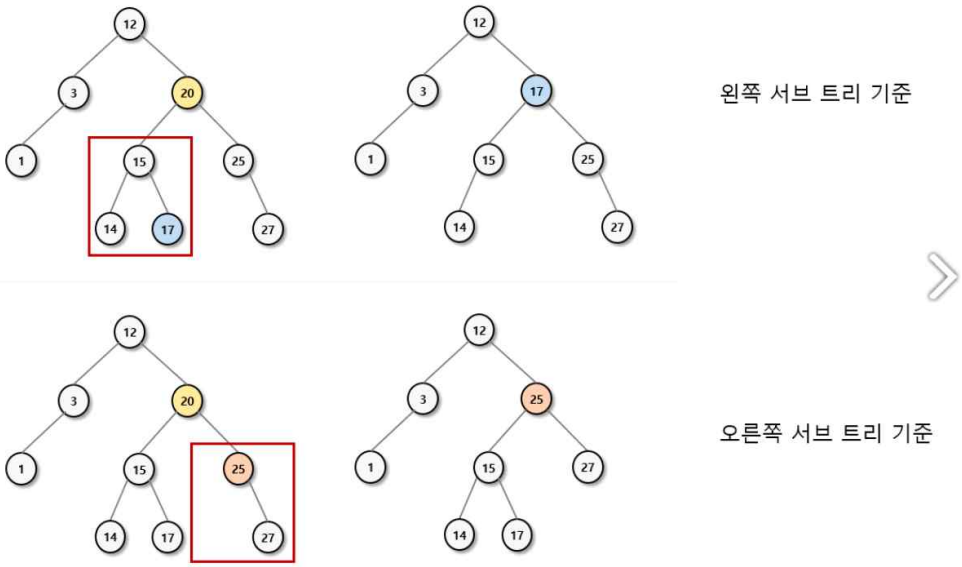


3. 삭제할 노드의 자식 노드 2개인 경우
삭제할 노드의 자리에 대신 채울 노드를 찾아야 하는 과정이 필요합니다.
대신 채울 노드를 찾는 방법은 2가지 존재합니다.

- 삭제할 노드의 왼쪽 서브 트리에서 최대 값 노드
- 삭제할 노드의 오른쪽 서브 트리에서 최소 값 노드

삭제할 노드의 왼쪽 서브트리에서 최대값을 찾는 방법은, 삭제할 노드 왼쪽 노드에서 오른쪽 자식이 없을 때까지 밑으로 찾아내려 가면 된다. 오른쪽 서브트리에서 최소값 찾는 방법도 이방법과 동일하다.

아래의 예시는 20 노드를 삭제할 때, 서브 트리에서 위 방법으로, 노드를 찾아 대체합니다.



이진탐색트리 시간복잡도

1. 검색/삽입 시간복잡도

- 위에서 검색/삽입 연산을 살펴보시면 알겠지만, 모든 원소를 탐색하지 않는 것이 특징입니다. 최악의 경우는 트리의 높이만큼 탐색하는 경우라 볼 수 있겠다.
- 따라서 높이가 H 인경우의 검색/삽입의 시간 복잡도는 O(H)입니다. 하지만, 시간복잡도는 원소의 개수로 표기할 때, 조금 더 좋은 표현이라 할 수 있습니다. 이진트리는 대개, 원소의 개수가 N 이면 logN으로 치환 가능합니다.
- 따라서 **검색/삽입 연산의 시간복잡도는 O(logN)**이다.(단, 이진 검색트리 구조가 완전이진트리라 가정할 최악의 경우로 한정적이긴 하다. 그래서 최악의 경우인 O(N)로 규정짓는 빅오 표기법으로는 애매하다.)

2. 삭제 시간복잡도

- 검색 시간복잡도는 O(H) 로 삭제할 대상을 검색하는 것도 동일하게 시간이 소모됩니다.
- 이때, 삭제할 노드가 단말 노드, 삭제할 노드의 자식 노드가 1개 인 경우는 지우거나 참조값만 붙여주면 되므로, O(1) 의 시간이 소모된다고 볼 수 있습니다. 따라서 총 시간 복잡도는 O(H + 1) → O(H) 라 할 수 있을 것이다.
- 삭제할 노드의 자식 노드가 2개 인 경우는 검색 후, 대체할 노드만 찾는다면 대체 노드의 참조값만 바꿔주면 되므로 O(1)입니다. 그말은, 대체할 노드를 찾는게 얼마나 걸리는지에 따라 시간 복잡도가 규정 된다는 점입니다.

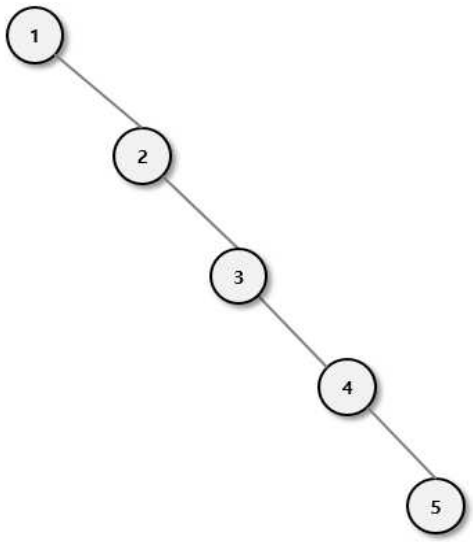
- 생각해보자면, 대체할 노드를 찾는것도 결국 탐색이 필요합니다. 다만 Full-Tree 가 아닌 Sub-Tree 에서의 검색이지요. 시간복잡도를 간단하게 정의해본다면 O(H-K) 정도가 될것입니다. K 는 탐색을 시작할 높이 값입니다.
- 시간복잡도는 최악의 경우를 생각해야 하므로, 최악의 경우의 K는 0 이 되는 경우가 있을 것이다. 이 경우는 루트를 지운다는 의미입니다.
- 결국 서브트리를 탐색하는 시간복잡도도 O(H) 라 생각할 수 있겠습니다. 정리해보자면, 아래의 작업이 필요합니다.

1. 삭제 노드를 탐색 O(H)
2. 대체 노드를 찾기 위한 서브 트리를 탐색 O(H)
3. 참조값 변경 O(1)

- 따라서 삭제 연산의 시간 복잡도는 O(H + H + 1) → O(2H) → O(H) 라 볼 수 있습니다. 위에서 N으로 치환한 것과 마찬가지로 **O(logN)**으로 표기가 가능하다
- 결론적으로 **탐색/삽입/삭제 시간복잡도는 전부 동일하게 O(logN)**이라고 정의할 수 있다.

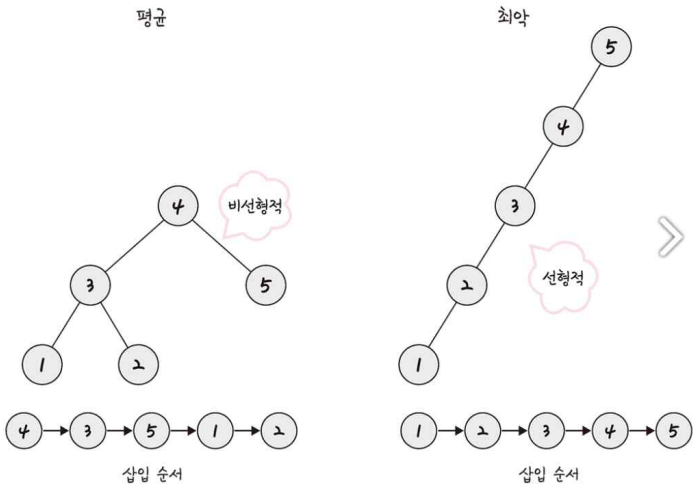
한쪽으로 쏠린 트리(편향된 트리)

- 이진탐색트리의 **시간복잡도는 O(logN)이지만, 사실 최악의 경우 O(N)**이 나오게 된다
- **최악의 경우는 선형적인 그래프 형태**이다.



해당 구조는 트리 라기보다, 연결 리스트 의 구조로 잡혀있습니다.

높이가 낮을수록 성능에 유리한 이진 검색트리가 이렇게 개수만큼 높이가 쌓이게 되면, 검색/삽입/삭제는 선형 연산이 돼버립니다. 선형 연산이기 때문에 시간 복잡 도는 O(N) 이 됩니다. 5를 찾으려면 5번 탐색 이 필요할 것입니다.



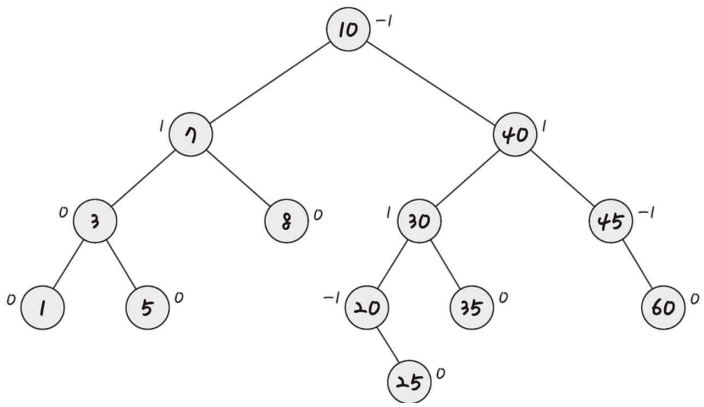
균형 이진 탐색트리

이진 탐색트리는 완전 이진트리 구조로 쌓여야, 요소 개수 N개의 대비 최소 높이가 됩니다. 그렇다면, **요소를 어떤 순서로 삽입 하더라도, 강제적으로 완전 이진트리 구조로 되게끔 규칙을 만들면 될 것이다.**

그러한 알고리즘을 추가한 이진 탐색트리를 **균형이진 탐색트리(자가균형트리) 라 부르며, 대표적은 구현체는 AVL 트리 , 레드 블랙 트리 등이 있습니다.**

AVL 트리(Adelson-Velsky and Landis Tree)

- AVL트리는 이진탐색트리의 최악의 경우 선형적인 트리가 되는 것을 방지하고
- 스스로 균형을 잡는 이진 탐색트리이다.
- 두 자식 서브 트리 높이는 항상 최대 1만큼 차이가 난다는 특징이있다.
- 높이 차이가 1보다 크면 균형을 맞추기 위해 트리 회전 알고리즘을 이용한다.



두 자식 서브트리의 높이는 항상 최대 1만큼 차이난다. 만약 어떤 시점에서 높이 차이가 1보다 커지면 이 속성을 유지하기 위해서 스스로 균형을 잡는다. 검색, 삽입, 삭제는 모두 평균과 최악의 경우 O(log n)의 시간복잡도가 걸린다. 삽입과 삭제는 한 번 이상의 트리 회전을 통해 균형을 잡을 수 있다.

AVL 트리의 시간복잡도		
	평균	최악의 경우
공간	$O(n)$	$O(n)$
검색	$O(\log n)$	$O(\log n)$
삽입	$O(\log n)$	$O(\log n)$
삭제	$O(\log n)$	$O(\log n)$

레드-블랙 트리(Red-Blaxk Tree)

- BST를 기반으로 하는 트리 형식의 자료구조이다.
- Search, Insert, Delete에 O(logN)의 시간 복잡도가 소요된다. 동일한 노드의 개수일 때, depth를 최소화하여 시간 복잡도를 줄이는 것이 핵심 아이디어이다.
- 이진탐색트리의 최악의 경우 선형적인 트리가 되는 것을 방지하고 스스로 균형을 잡는 이진 탐색트리이다.
- 루트부터 leaf까지의 모든 경로 중 최소 경로와 최대 경로의 크기 비율은 2보다 크지 않다. 이러한 상태를 balanced라고 한다.
- 노드의 child가 없을 경우 child를 가리키는 포인터는 nil 값을 저장한다. 이러한 nil들을 leaf node로 간주한다.

레드-블랙 트리 알고리즘, 구현 매커니즘

- 각 노드는 red, black이라는 색깔을 갖는다. root node는 black이다. 각 leaf node는 black이다.
- 어떤 노드의 색깔이 red라면 두 개의 children의 색깔은 모두 black이다.
- 모든 leaf node는 같은 black depth를 가진다.
- black-height를 가진다. 노드 x로부터 노드 x를 포함하지 않은 leaf node까지의 simple path상에 있는 black node들의 개수가 같다.

black depth : root에서 current node까지의 black edge의 수
black edge: black인 자식에서 parent로 가는 edge

삽입
BST의 특성을 유지하면서 노드를 삽입한다. 삽입된 노드의 색깔은 RED로 지정한다. 만약 Red의 자식은 black이어야한다는 규칙을 위배하는 double red가 발생한다면 두 가지 해결 방법을 통해 해결한다.

restructuring : uncle node가 black일 때 일렬 묶음에서 중간 값을 root로 재배치 후 색깔을 black이었던 노드와 바꿔준다.
Recoloring: uncle node가 red일 때 grand parent를 red (root라면 black), parent, uncle은 black으로 바꿔준다. 다시 double red가 발생할 수 있다.

따라서 수행은 아래와 같다

1. insertion 된 z 찾기
2. 추가하고 red 칠하기

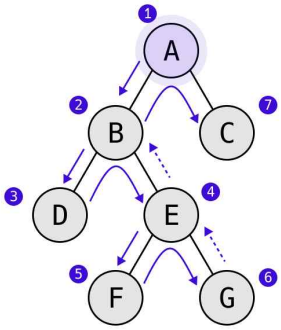
삭제
삽입과 마찬가지로 BST의 특성을 유지하면서 해당 노드를 삭제한다. 삭제될 노드의 child 개수에 따라 rotation 방법이 달라지게 된다. 만약 지워진 노드의 색깔이 black이라면 black-height가 1 감소한 경로에 black node가 1개 추가되도록 rotation하고 노드의 색깔을 조정한다. 지워진 노드의 색깔이 red라면 Violation이 발생하지 않으므로 RBT가 그대로 유지된다.

순회방법과, 이진탐색트리의 순회방법, 순서

- 순회 방식에는 세 가지가 있다. 전위, 중위, 후위
- **1. 전위 순회(Preorder Traversal)** : 전위순회는 루트 노드를 먼저 탐색하고, 자식 노드를 탐색하는 방식이다.
- 현재 노드(부모)를 출력(처리), 왼쪽 (자식)노드 방문, 오른쪽 (자식)노드 방문
  - **루트(부모노드) → 왼쪽 자식 노드 → 오른쪽 자식 노드**
- **2. 중위 순회(Inorder Traversal)** : 중위순회는 왼쪽 자식 노드를 탐색하고, 루트 노드를 탐색하고, 오른쪽 자식 노드를 탐색하는 방식이다.
- 왼쪽(자식) 노드를 방문(왼쪽 노드가 없을 때 까지 진행), 현재(부모) 노드를 출력(처리), 오른쪽(자식) 노드 방문
  - **왼쪽 자식 노드 → 루트(부모노드) → 오른쪽 자식 노드**
- **3. 후위 순회(Postorder Traversal)** : 후위순회는 왼쪽 자식 노드를 탐색하고, 오른쪽 자식 노드를 탐색하고, 루트 노드를 탐색하는 방식이다.
- 왼쪽(자식) 노드를 방문(왼쪽 노드가 없을 때 까지 진행), 오른쪽(자식) 노드를 방문, 현재(부모) 노드를 출력(처리)
  - **왼쪽 자식 노드 → 오른쪽 자식 노드 → 루트(부모노드)**

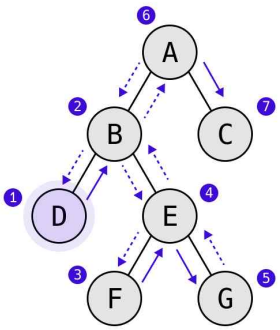
- **4. 추가로 레벨순회(Levelorder Traversal)** 또는 BFS(Breadth-Frist Search : 너비 우선 탐색)라는 것이 있다.
- **레벨순회(BFS)를 제외한 나머지 세가지 순회방식은 DFS(Depth-First Search : 깊이 우선 탐색)으로 분류할 수 있다.**

전위순회(Preorder Traversal) : 루트(부모노드) → 왼쪽 자식 노드 → 오른쪽 자식 노드



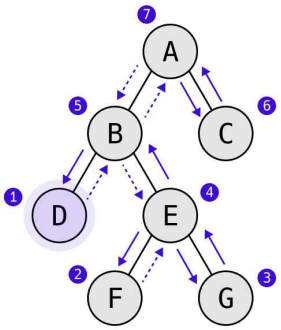
Preorder Traversal **A B D E F G C**

중위순회(Inorder Traversal) : 왼쪽 자식 노드 → 루트(부모노드) → 오른쪽 자식 노드



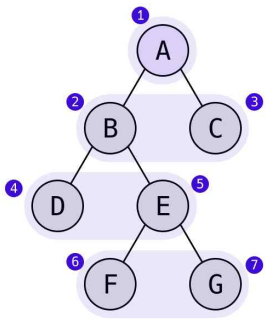
Inorder Traversal **D B F E G A C**

후위순회(Postorder Traversal) : 왼쪽 자식 노드 → 오른쪽 자식 노드 → 루트(부모노드)



postorder Traversal **D F G E B C A**

레벨순회(Levelorder Traversal : BFS) : 루트 노드를 먼저 탐색하고, 그 다음 레벨의 노드를 탐색하는 방식이다. queue를 하나 선언해두고, 루트 노드를 넣어준 다음, queue에 넣어준 노드를 하나씩 탐색하면서 자식 노드를 queue에 넣는다. queue가 비어있을 때까지 반복한다.



levelorder Traversal **A B C D E F G**