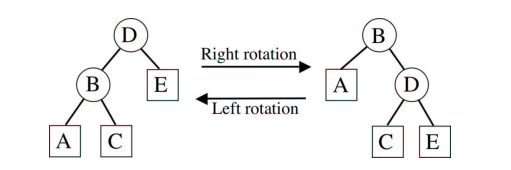
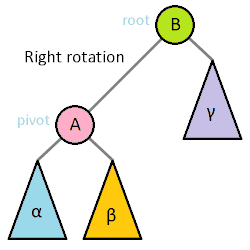
균형 이진 탐색 트리[Balancing binary search trees]

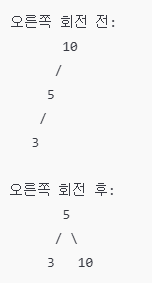
이진 탐색 트리에 항목이 무작위 순서로 추가되면 트리는 비교적 균형을 이루는 경향이 있으며 높이는 log₂ n보다 훨씬 크지 않습니다. 그러나 항목이 무작위 순서가 아닌 상황이 많이 있습니다. 새 항목이 오름차순으로 추가되는 극단적인 경우(ex. 1부터 10까지의 숫자를 순서대로 이진 탐색 트리에 넣으면, 모든 숫자가 오른쪽으로만 연결된 형태가 됩니다.)에는 트리가 오른쪽으로 하나의 긴 가지가 되어 높이가 n > log₂ n이 됩니다.

불균형 이진 탐색 트리를 피하는 또 다른 방법은 트리 회전을 사용하여 때때로 트리의 균형을 다시 맞추는 것입니다. 이 방법은 이진 탐색 트리의 속성을 유지하면서 트리의 높이를 줄이고 균형을 맞추는 데 효과적입니다.

트리 회전은 **왼쪽 회전(left rotation)**과 **오른쪽 회전(right rotation)**이라는 두 가지 주요 작업으로 구성됩니다.



**오른쪽 회전(right rotation)**

오른쪽 회전은 특정 서브트리의 루트를 아래로 이동시키고, 그 왼쪽 자식을 새로운 루트로 올리는 작업입니다.

**왼쪽 서브트리가 너무 커져서 트리가 왼쪽으로 치우칠 때** 사용합니다.

트리의 불균형이 **왼쪽 노드들로 인해 발생**하는 경우 높이를 줄이기 위해 수행됩니다.

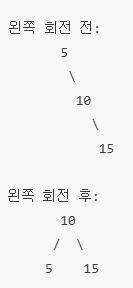
오른쪽 회전은 트리의 **왼쪽 자식**을 새로운 루트로 승격시키고, 기존 루트를 **새로운 루트의 오른쪽 자식**으로 이동시킵니다.

왼쪽 자식의 **오른쪽 서브트리**는 기존 루트의 **왼쪽 서브트리**로 이동합니다.

<- 10의 왼쪽 서브트리가 너무 크므로 5를 루트로 승격시켜 균형을 맞춥니다.

**왼쪽 회전(left rotation)**

왼쪽 회전은 특정 서브트리의 루트를 아래로 이동시키고, 그 오른쪽 자식을 새로운 루트로 올리는 작업입니다.

**오른쪽 서브트리가 너무 커져서 트리가 오른쪽으로 치우칠 때** 사용합니다.

트리의 불균형이 **오른쪽 노드들로 인해 발생**하는 경우 높이를 줄이기 위해 수행됩니다.

왼쪽 회전은 트리의 **오른쪽 자식**을 새로운 루트로 승격시키고, 기존 루트를 **새로운 루트의 왼쪽 자식**으로 이동시킵니다.

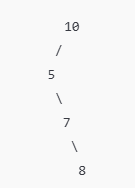
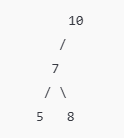
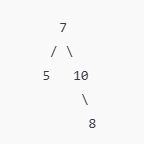
오른쪽 자식의 **왼쪽 서브트리**는 기존 루트의 **오른쪽 서브트리**로 이동합니다.

<- 5의 오른쪽 서브트리가 너무 크므로 10을 루트로 승격시켜 균형을 맞춥니다.

복잡한 회전

**좌-우 회전 (Left-Right): 왼쪽 자식의 오른쪽 서브트리가 불균형을 일으킬 때 사용합니다.**

1. **왼쪽 자식**에 대해 **왼쪽 회전**.
2. 전체 트리에 대해 **오른쪽 회전**.

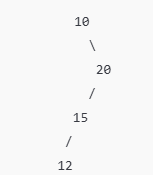
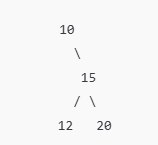
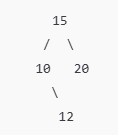
  

1.왼쪽 자식(5)에 대해 왼쪽 회전  
노드 7이 새로운 서브트리의 루트가 되고, 5는 7의 왼쪽 자식이 됩니다. 7의 오른쪽 자식8은 변하지 않습니다.

2.전체 트리(10)에 대해 오른쪽 회전  
노드 7이 전체 트리의 새로운 루트가 됩니다. 기존 루트 10은 7의 오른쪽 자식이 되고, 기존의 8은 그대로 유지됩니다.

**우-좌 회전 (Right-Left):**

1. **오른쪽 자식**에 대해 **오른쪽 회전**.
2. 전체 트리에 대해 **왼쪽 회전**.

1.오른쪽 자식(20)에 대해 오른쪽 회전  
노드 15가 새로운 서브트리의 루트가 되고, 20은 15의 오른쪽 자식이 됩니다. 12는 그대로 유지되며, 20의 오른쪽 자식은 비어 있습니다.

2.전체 트리(10)에 대해 왼쪽 회전  
노드 15가 전체 트리의 새로운 루트가 됩니다. 기존 루트 10은 15의 왼쪽 자식이 되고, 기존의 12와 20은 그대로 유지됩니다.

트리에서 **오른쪽 회전**과 **왼쪽 회전**은 **트리의 불균형 문제를 해결하고 높이를 최소화**하기 위해 사용됩니다. 이 회전들은 이진 탐색 트리(BST) 또는 AVL 트리에서 트리의 균형을 유지하여 탐색, 삽입, 삭제 작업의 효율성을 보장합니다.

**자가 균형 AVL 트리[Self-balancing AVL trees]**

자가 균형 이진 탐색 트리는 삽입 과정 전반에 걸쳐 트리를 자동으로 재균형화하여 각 단계에서 높이를 log₂ n에 가깝게 유지함으로써 불균형 트리의 문제를 피합니다.

편향 트리의 문제를 해결하기 위해 고안된 "자가 균형 이진 탐색 트리"입니다. 핵심 아이디어는 트리의 균형을 자동으로 유지하여 항상 효율적인 검색 성능을 보장하는 것입니다.

가장 초기의 자가 균형 이진 탐색 트리 유형은 **AVL 트리**(발명가인 G.M. Adelson-Velskii와 E.M. Landis의 이름을 따서 명명됨)였습니다. 이 트리는 **모든 노드의 두 하위 트리 높이 차이를 최대 1**로 유지합니다. 이를 위해서는 위에서 논의한 대로 하나 이상의 트리 회전을 수행하여 트리를 주기적으로 재균형화해야 하지만, 삽입, 삭제 및 검색의 복잡도는 여전히 O(log₂ n)입니다.

**AVL 트리의 장점:**

* 항상 균형을 유지하므로 삽입, 삭제, 검색의 시간 복잡도가 O(log n)으로 보장됩니다.
* 데이터 검색 속도가 매우 빠릅니다.

**AVL 트리의 단점:**

* 균형을 유지하기 위한 회전 연산으로 인해 삽입 및 삭제 속도가 상대적으로 느릴 수 있습니다.
* 구현이 비교적 복잡합니다.

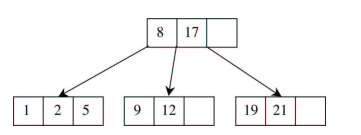
**B-트리[B-trees]**

B-트리는 자가 균형 이진 탐색 트리(self-balancing binary search tree)의 일반화된 구조로, 각 노드가 두 개 이상의 검색 키(search key)를 가질 수 있고, 두 개 이상의 자식을 가질 수 있습니다. 이러한 구조는 자가 균형을 더 효율적으로 유지하도록 설계되었으며, 특히 노드 데이터를 디스크 드라이브와 같은 외부 저장소에 저장해야 하는 경우에 유리합니다.

**B-트리 정의 (Knuth의 정의)**

B-트리의 차수 [각 노드가 가질 수 있는 최대 자식 수] (order)가 m인 경우 다음 조건을 만족합니다:

1. **모든 노드는 최대 m개의 자식을 가질 수 있습니다.**
2. **모든 비단말 노드(루트 노드를 제외)는 최소 ⌈m/2]개의 자식을 가져야 합니다.**
3. **루트 노드가 비단말 노드라면, 최소 두 개의 자식을 가져야 합니다.**
4. **비단말 노드가 c개의 자식을 가지면, c−1개의 검색 키를 포함합니다.**  
   이 검색 키는 각 서브트리를 구분하는 역할을 합니다.
5. **모든 단말 노드(leaf)는 동일한 레벨에 위치하며 정보를 저장합니다.**

****

**Ex.**

차수 m=4, 최대4개의 자식을 가질 수 있고, 최대 3개의 검색 키를 가질 수 있습니다.

모든 비단말 노드는 최소 ⌈4/2⌉=2개의 자식을 가져야 합니다.

10, 20, 5, 6, 12, 30, 7, 17

* 1. [10]
  2. [10,20]
  3. [5,10,20]
  4. [5,6,10,20] m−1=3개의 검색 키를 초과하므로 **노드 분할**