**이진 탐색 트리[Binary Search Trees]**

이진 탐색 트리는 데이터를 효율적으로 찾기 위해 사용하는 특별한 형태의 트리 구조입니다. 마치 책의 목차처럼, 특정 정보를 빠르게 찾을 수 있도록 도와줍니다.

일반적인 배열이나 리스트에서 원하는 정보를 찾으려면, 처음부터 하나씩 다 확인해야 할 수도 있습니다. 하지만 이진 탐색 트리를 사용하면, 마치 책의 목차에서 원하는 챕터를 바로 찾듯이, 훨씬 빠르게 정보를 찾을 수 있습니다.

**검색 키[Search keys]**

"검색 키"라는 것은 우리가 어떤 데이터를 찾을 때 사용하는 기준, 즉 '검색어'와 같은 것입니다. 마치 책에서 원하는 내용을 찾기 위해 제목, 저자, 또는 색인어를 사용하는 것과 같습니다.

ex.

**학번:** 학생들을 구분하기 위해 사용하는 고유한 번호입니다. 학번을 '검색 키'로 사용하면 특정 학생의 정보를 빠르게 찾을 수 있습니다.

**단어:** 사전에서 특정 단어의 뜻을 찾을 때, 그 단어가 '검색 키'가 됩니다. 사전에 단어들이 알파벳 순서대로 정렬되어 있기 때문에, 원하는 단어를 쉽게 찾을 수 있습니다.

**전화번호부:** 전화번호부에서 사람의 이름을 '검색 키'로 사용하여 전화번호를 찾습니다.

* 검색 키'는 데이터를 찾기 위한 기준(검색어)이다.
* '검색 키'를 사용하여 데이터를 정렬하면 검색 속도를 향상시킬 수 있다.
* '검색 키'는 숫자, 단어 등 비교 가능한 어떤 것이든 될 수 있다.
* 실제 데이터는 '검색 키' 외에도 더 많은 정보를 포함할 수 있지만, 검색 알고리즘에서는 '검색 키'에 집중한다. (예: 사전에서 '단어'가 검색 키이고, '정의'는 추가 정보이다.)

따라서, '검색 키'라는 용어를 접했을 때, "아, 데이터를 찾기 위해 사용하는 기준이구나!"라고 생각하면 된다.

**이진 탐색 트리**

이진 탐색 트리는 데이터를 효율적으로 검색하기 위해 사용하는 특별한 형태의 이진 트리입니다.

정의. 이진 탐색 트리는 비어(empty) 있거나 다음 조건을 만족하는 이진 트리입니다.

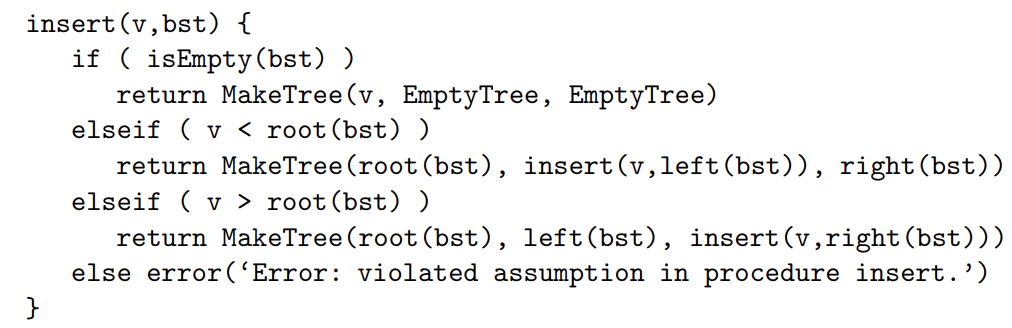
* 왼쪽 하위 트리에서 발생하는 모든 값은 루트의 값보다 작습니다.
* 오른쪽 하위 트리에서 발생하는 모든 값은 루트의 값보다 큽니다.
* 왼쪽 및 오른쪽 하위 트리 자체도 이진 탐색 트리입니다.

검색 키인 노드 값을 가진 특정 유형의 이진 트리일 뿐입니다. 즉, 일반적인 이진 트리에 대해 정의한 많은 연산자와 알고리즘을 상속할 수 있습니다. 추가적인 노드 값 순서만 유지하면 됩니다.

**이진 탐색 트리 구축[Building binary search trees]**

이진 탐색 트리를 구축할 때, 자연스럽게 루트에서 시작하여 필요에 따라 새로운 노드를 추가합니다. 따라서 새로운 값 v를 삽입하려면 다음과 같은 경우가 발생합니다.

* 주어진 트리가 비어 있으면 새 값 v를 루트에 할당하고 왼쪽 및 오른쪽 하위 트리를 비워 둡니다.
* 주어진 트리가 비어 있지 않으면 다음과 같이 값 v가 있는 노드를 삽입합니다.
  + v가 루트 값보다 작으면 v를 왼쪽 하위 트리에 삽입합니다.
  + v가 루트 값보다 크면 v를 오른쪽 하위 트리에 삽입합니다.
  + v가 루트 값과 같으면 가정 위반을 보고합니다.

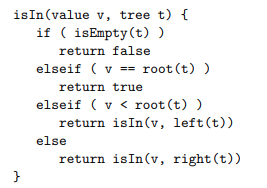


insert 함수는"새로운 트리를 생성하는 방식"으로 구현되어 있습니다. 실제 대용량 데이터를 다루는 시스템에서는 메모리 효율성과 성능을 위해 "기존 트리를 수정하는 방식" (포인터를 사용하는 방식)을 사용하는 것이 일반적입니다.

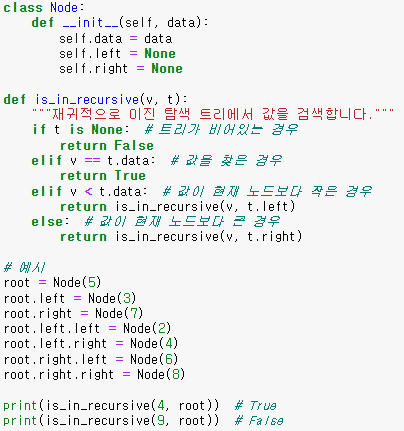
**이진 탐색 트리 검색[Searching a binary search tree]**

이진 탐색 트리는 **부모 노드보다 작은 값은 왼쪽 자식 노드에, 큰 값은 오른쪽 자식 노드에 저장**하는 규칙을 가지고 있습니다. 이 규칙 덕분에 매우 효율적으로 검색할 수 있습니다.

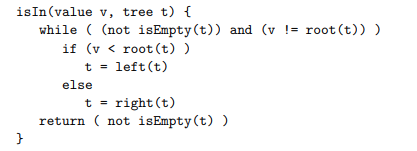
일반적으로 다음과 같은 재귀적 절차와 같은 의사 코드로 알고리즘을 작성합니다.



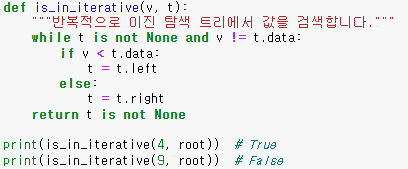
각 재귀는 적절하게 검색을 왼쪽 또는 오른쪽 하위 트리로 제한하여 검색 트리 높이를 1씩 줄이므로 알고리즘은 결국 종료됩니다.



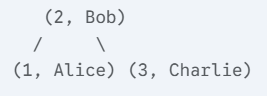
재귀는 while 루프로 쉽게 변환할 수 있습니다.



루프를 빠져나가는 유일한 방법은 필요한 값을 찾았거나 빈 트리만 남은 경우이므로 절차는 최종 트리가 비어 있는지 여부만 반환하면 됩니다.



실제로 단순한 true/false보다 더 많은 것을 반환하려는 경우가 많습니다. 예를 들어 학번을 검색하는 경우, 단순히 존재 여부를 확인하는 것이 아니라 해당 학생의 전체 기록에 대한 포인터를 원하는 경우가 보통입니다.

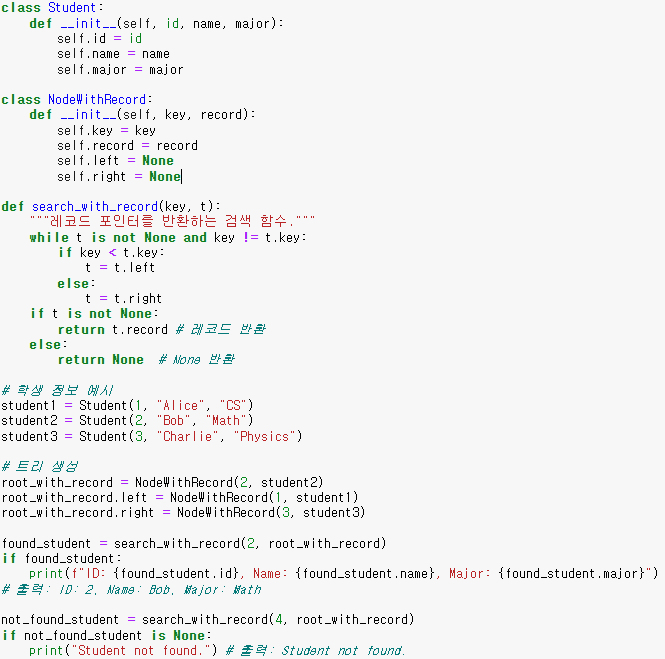
 <Visual Representation>

학번 2 (키: 2) 검색:

* 루트 노드 (학번 2)와 비교 - 동일 (검색 성공)
* 학생 정보 (Bob, Math) 반환

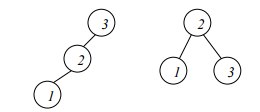
학번 4 (키: 4) 검색:

* 루트 노드 (학번 2)와 비교 - 4 > 2 (오른쪽 자식 노드로 이동)
* 오른쪽 자식 노드 (학번 3)과 비교 - 4 > 3 (트리에 학번 4가 없음)
* 왼쪽 자식 노드도 없으므로 검색 실패 (None 반환)
* "학생을 찾을 수 없습니다." 출력



**삽입 및 검색의 시간 복잡도[Time complexity of insertion and search]**

이진 탐색 트리에서 삽입과 검색은 트리의 높이에 매우 큰 영향을 받습니다.



최악의 경우: 트리가 한쪽으로 치우쳐진 경우 (예: 모든 노드가 왼쪽 자식만 갖거나 오른쪽 자식만 갖는 경우), 높이는 노드 수와 같아집니다. 이 경우 삽입과 검색 모두 O(n)의 시간 복잡도를 갖습니다. 마치 연결 리스트를 순차적으로 탐색하는 것과 같습니다.

**최선의 경우 및 평균적인 경우:** 트리가 균형 잡힌 경우 (왼쪽과 오른쪽 하위 트리의 높이가 거의 같은 경우), 높이는 log₂ n에 비례합니다. 이 경우 삽입과 검색 모두 O(log₂ n)의 시간 복잡도를 갖습니다. 이는 매우 효율적이며, 정렬된 배열에서 이진 탐색을 하는 것과 같은 효율성을 보입니다.

트리가 얼마나 균형 잡혀 있는지에 따라 성능이 크게 달라집니다. 완벽하게 균형 잡힌 트리는 최소 높이를 가지므로 가장 효율적입니다. 하지만 완벽하게 균형 잡히지 않더라도 어느 정도 균형을 유지하면 최악의 경우인 O(n)보다는 훨씬 나은 성능을 낼 수 있습니다.

**이진 탐색 트리 vs. 일반적인 이진 트리**

**이진 탐색 트리(Binary *Search* Tree):** 모든 노드에 대해 "왼쪽 하위 트리의 모든 값 < 현재 노드 값 < 오른쪽 하위 트리의 모든 값"이라는 **정렬된 속성**을 만족하는 트리입니다. 이 속성 때문에 효율적인 검색이 가능합니다.

**일반적인 이진 트리(General Binary Tree):** 단순히 각 노드가 최대 두 개의 자식을 가질 수 있는 트리일 뿐, 값의 대소 관계에 대한 제약이 없습니다. 즉, 정렬되어 있지 않을 수 있습니다.

일반적인 이진 트리는 정렬 조건이 없기 때문에, 가능한 모든 트리를 고려했을 때 평균 높이가 O(√n)입니다.

n개의 노드로 만들 수 있는 이진 트리의 개수는 n이 증가함에 따라 매우 빠르게 증가합니다. 즉, 대부분의 트리가 상당히 불균형한 형태를 가지게 됩니다. 이러한 높은 트리들이 평균 높이를 끌어올리는 효과를 내기 때문에, 평균 높이는 O(log n)보다는 훨씬 큰 O(√n)에 가까워지는 것입니다.

**이진 탐색 트리에서 노드 삭제 방법[Deleting nodes from a binary search tree]**

어떤 이유로 이진 탐색 트리에서 항목을 삭제해야 할 경우, 트리를 새로 구축하는 것은 매우 비효율적입니다. 노드를 하나씩 삭제하고 다시 추가하는 방식은 O(nlogn)의 복잡도를 가지므로, 보다 효율적인 방법이 필요합니다. 이 문제를 해결하기 위해 다음 알고리즘을 사용할 수 있습니다.

1 **노드가 리프 노드(자식이 없는 노드)인 경우:**

* 해당 노드를 단순히 삭제하면 됩니다.



노드 1을 삭제

2 **노드가 하나의 서브트리만 가지고 있는 경우:**

* 남아있는 서브트리를 부모 노드와 연결합니다. 즉, 남아있는 서브트리를 "위로 이동"시키는 방식입니다.



노드 3을 삭제

3 **노드가 두 개의 서브트리를 가지고 있는 경우:**

* 오른쪽 서브트리에서 가장 작은 값을 가진 노드(오른쪽 서브트리의 "최소 노드")를 찾습니다.
* 이 최소 노드의 값을 삭제할 노드에 덮어씌웁니다.
* 그런 다음, 오른쪽 서브트리에서 해당 최소 노드를 제거합니다.



노드 5를 삭제

오른쪽 서브트리의 최소 노드 6을 찾아서 5를 대체

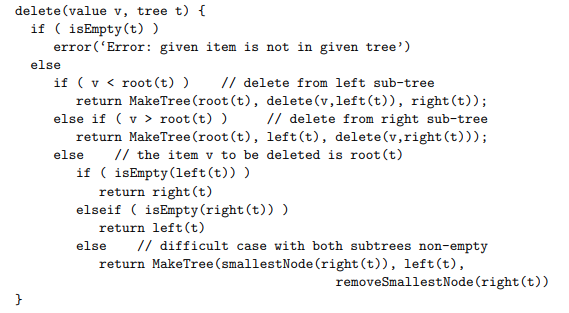
노드 7를 삭제

7의 오른쪽 서브트리의 최소값인 8이 7을 대체

8이 루트로 올라오면서 6은 그대로 왼쪽 자식으로 유지

\*\* 모든 **왼쪽 자식**은 부모보다 작아야 하고, 모든 **오른쪽 자식**은 부모보다 커야 합니다.

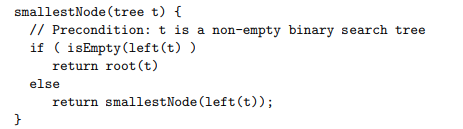
Delete:



이진 탐색 트리에서 지정된 값을 가진 노드를 삭제합니다.  
세 가지 경우를 처리:

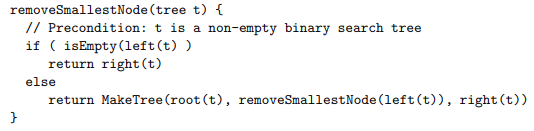
* 노드가 자식이 없는 경우 (리프 노드): 단순히 삭제.
* 노드가 하나의 자식만 가진 경우: 자식을 부모와 연결.
* 노드가 두 자식을 가진 경우: 오른쪽 서브트리에서 최소값을 찾아 삭제할 노드와 교체.

smallestNode:



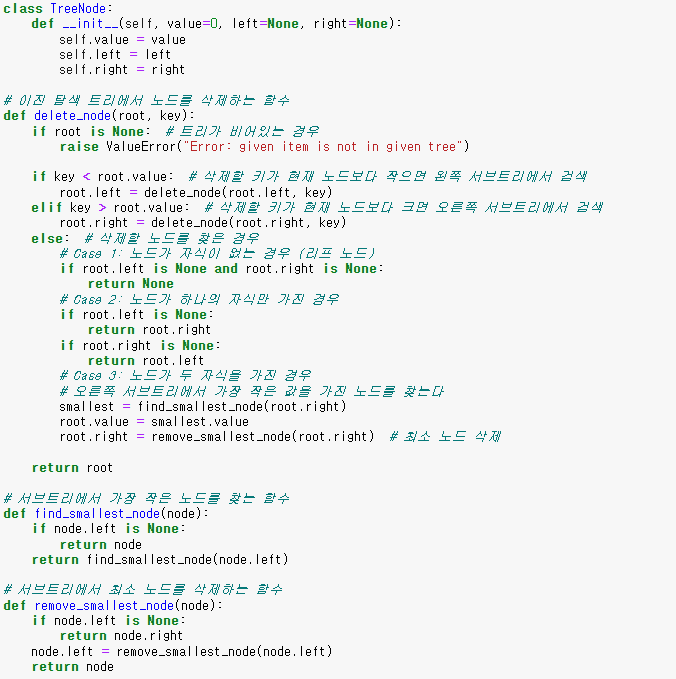
서브트리에서 가장 작은 값을 가진 노드를 반환합니다.  
이는 서브트리의 가장 왼쪽 노드입니다.

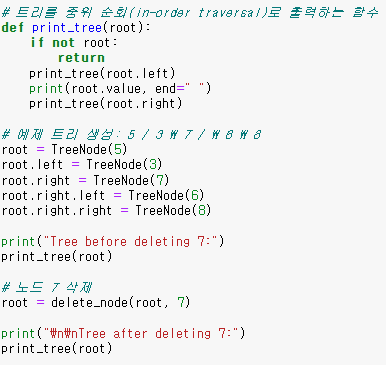
removeSmallestNode:



서브트리에서 최소값 노드를 삭제하고 나머지 트리를 반환합니다.

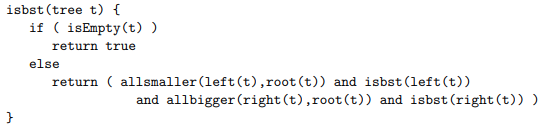
삭제 연산의 시간 복잡도는 검색, 삽입과 마찬가지로 트리의 높이에 따라 결정됩니다. 균형 잡힌 트리의 경우 O(log n), 최악의 경우(한쪽으로 치우쳐진 트리) O(n)입니다.

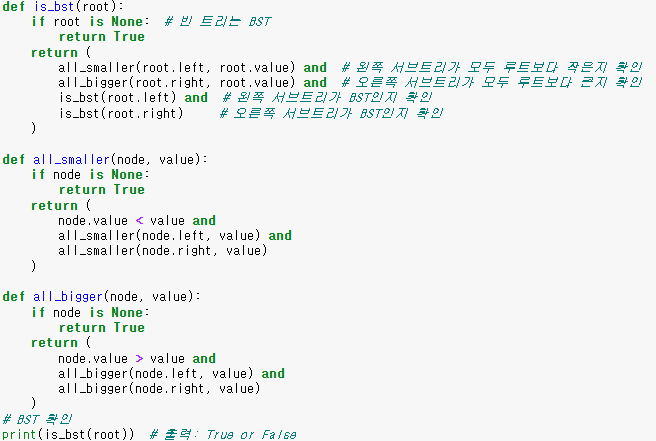
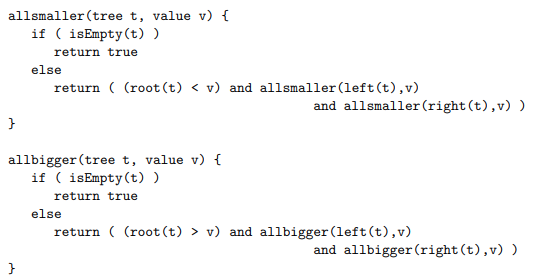




**이진 트리가 이진 탐색 트리(BST)인지 확인하기[Checking whether a binary tree is a BST]**

빈 트리는 이진 탐색 트리이며, 왼쪽 하위 트리의 모든 노드는 루트보다 작아야 하고 그 자체로 이진 탐색 트리를 이루어야 하며, 오른쪽 하위 트리의 모든 노드는 루트보다 커야 하고 그 자체로 이진 탐색 트리를 이루어야 한다는 것을 알고 있습니다.





가장 간단하거나 명백한 알고리즘이 항상 가장 효율적인 것은 아닙니다.

**이진 탐색 트리를 이용한 정렬[이진 탐색 트리를 이용한 정렬]**

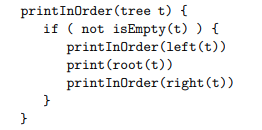
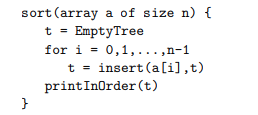
정렬은 주어진 데이터를 오름차순 또는 내림차순으로 재배열하는 과정입니다. 이진 탐색 트리를 활용하면 데이터를 정렬할 수 있습니다.

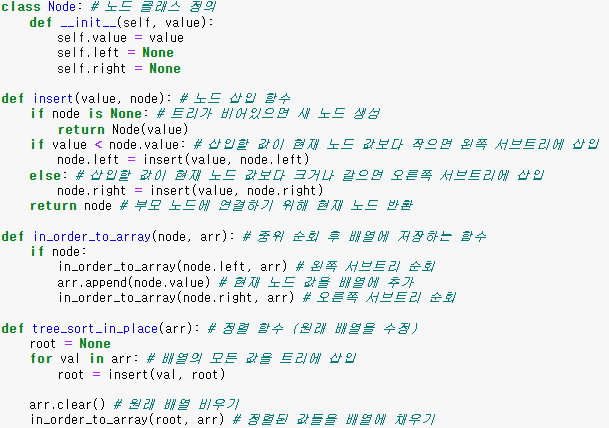
**이진 탐색 트리의 중위 순회(In-order Traversal)** #중위 순회는 이진 트리를 탐색하는 방법 중 하나

이진 탐색 트리의 특성상:

* 왼쪽 서브트리를 중위 순회한 결과 → **작은 값들**
* 루트 노드 → **중간 값**
* 오른쪽 서브트리를 중위 순회한 결과 → **큰 값들**

이 순서대로 값을 방문하면 항상 **오름차순으로 정렬된 값**을 얻을 수 있습니다.





이 숫자들로 만들어진 이진 탐색 트리는 다음과 같은 형태가 될 수 있습니다.

이렇게 만들어진 트리를 중위 순회하면 [1, 2, 4, 5, 8, 9]가 됩니다. 이 값들을 원래 배열에 다시 저장하면 정렬이 완료됩니다.  
  
  
