Validate Binary Search Tree Problemi

Busenur Çil
Halide Ceyda Sarıçelik

Problem Tanımı:

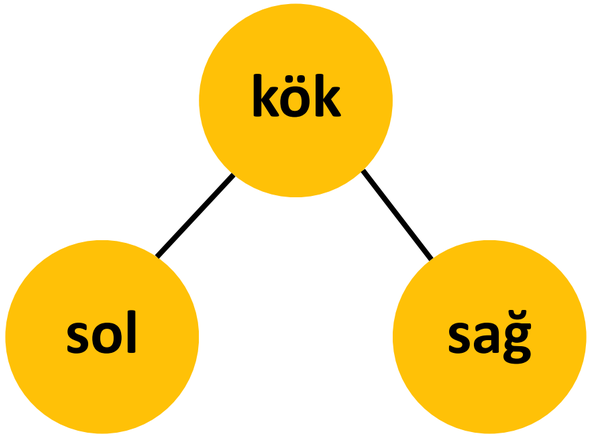
- Verilen bir ikili ağaçta her düğümün solundaki düğümler daha küçük, sağındakiler daha büyükse ağaç bir Binary Search Tree (BST) kabul edilir.
- Amaç, ağacın bu kurala uyup uymadığını kontrol etmektir.

Girdi ve Çıktı Türü:

- **Girdi:** TreeNode türünde bir kök (root) içeren ikili ağaç.
- **Çıktı:** Ağaç geçerli bir BST ise true, değilse false döndürülür.

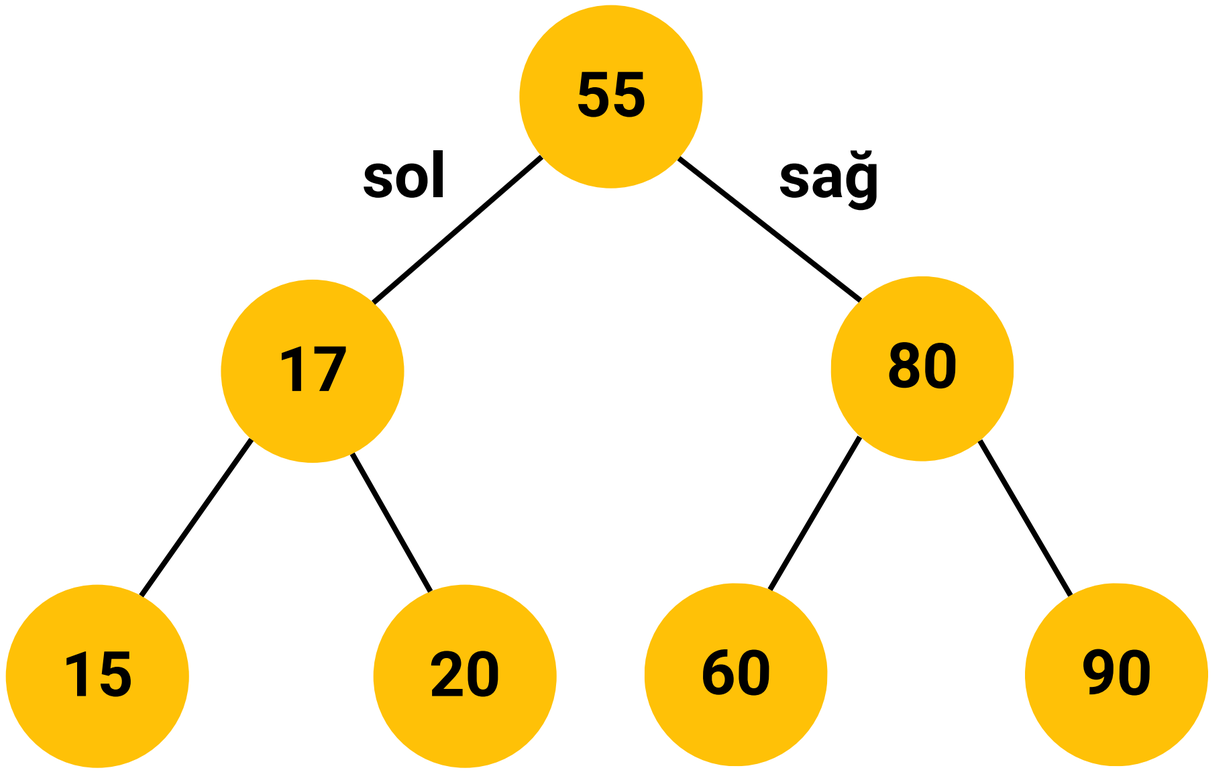
Binary Search Tree(BST) Nedir?

Binary Search Tree (İkili Arama Ağacı), özel kurallara sahip bir ikili ağaç (binary tree) yapısıdır. Bu kurallar, verilerin hızlı ve verimli şekilde eklenmesini, silinmesini ve aranmasını sağlar.



sol < kök < sağ

- Her düğümün solundaki değerler daha küçük
- Sağındaki değerler daha büyük olmalıdır.
- Bu kural her düğüm ve alt ağaç için geçerlidir.

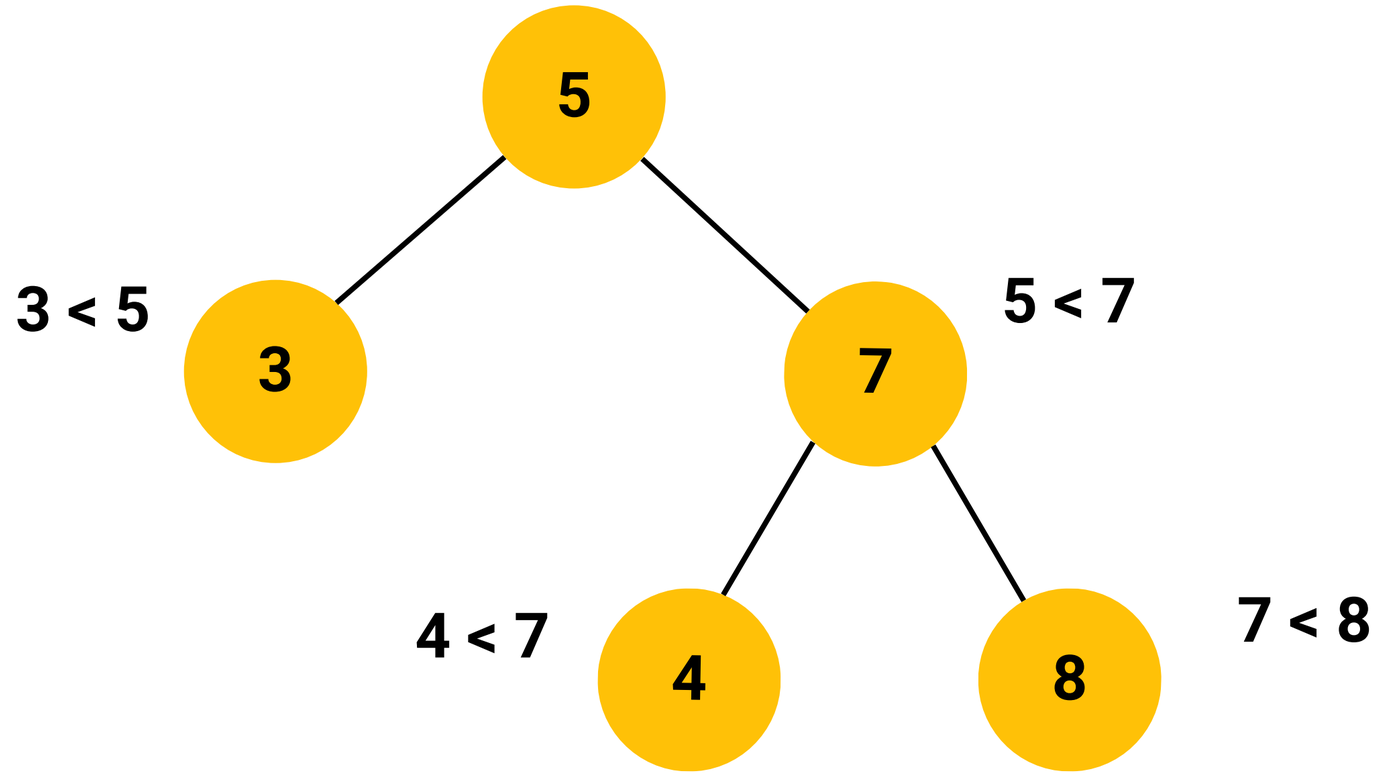


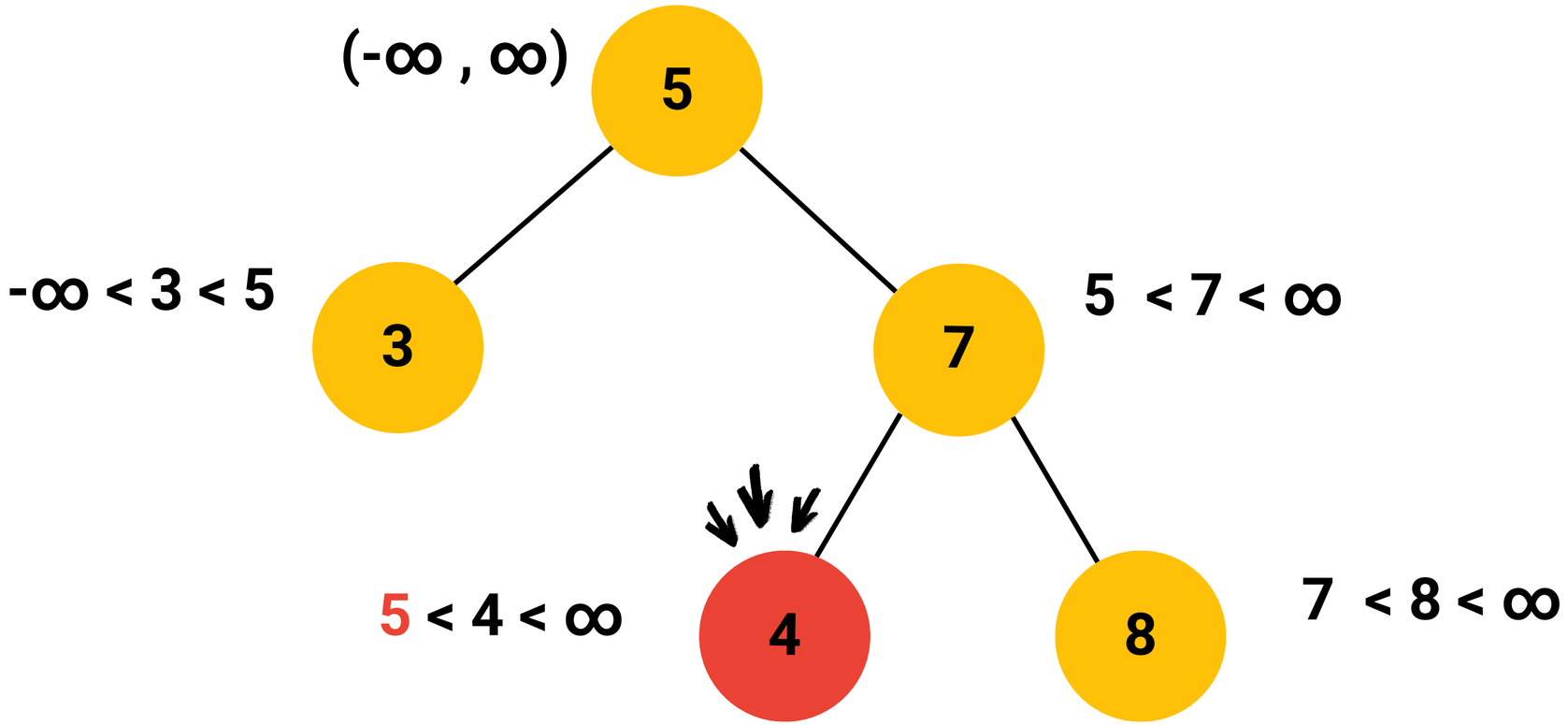
17 < 55 < 80 ✓

15 < 17 < 20 ✓

60 < 80 < 90 ✓

Binary search tree

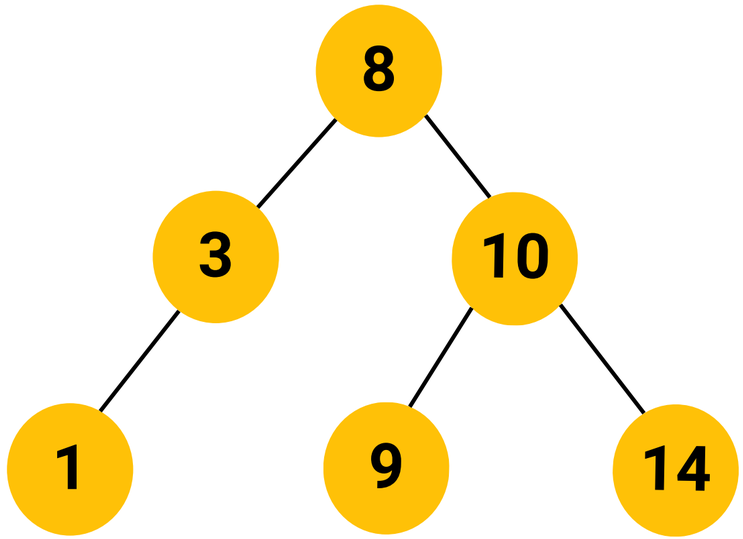

5
3 < 5
3
7
5 < 7
4 < 7
4
8
7 < 8



Binary search tree değil ✗

Algoritma - Pseudo Kod

```
Fonksiyon IsValidBST(düğüm, min, max):

    Eğer düğüm null ise:
        return true

    Eğer düğüm.değer ≤ min VEYA düğüm.değer ≥ max ise:
        return false

    solGeçerli = IsValidBST(düğüm.sol, min, düğüm.değer)

    sağGeçerli = IsValidBST(düğüm.sağ, düğüm.değer, max)

    return solGeçerli VE sağGeçerli
```



Çözüm ve Sınırlamalar

Problemi üç alternatif yolla ele alabiliriz. Bunlar:

- Recursive (Min-Max) Aralık Kontrolü
- Inorder Traversal
- Iterative Inorder Traversal (Stack kullanarak)

1. Yöntem: Min–Max Aralık Takibi (Recursive)

Mantık:

Her düğüm için, değerinin belli bir aralıkta olup olmadığını kontrol ederiz.
Bu aralık, ağaçta yukarıdan aşağıya doğru daraltılır.

Avantajları:

- Stack veya özel yapı gerekmez.
- Derinliği sınırlı ağaçlarda oldukça etkili

Dezavantajları:

- Derinlik çok artarsa, stack overflow (call stack) riski olur.
- Tüm ağaç gezilir, kontrol sürekli yapılır.

2. Yöntem: Inorder Traversal (Sıralama Mantığı)

Mantık:

BST’nin inorder traversal'ı her zaman sıralı (artan) olarak ilerler.

Bu yüzden, inorder sırasında düğümleri gezerken bir önceki düğümden küçük olup olmadığını kontrol ederiz.

Avantajları:

- BST’nin sıralı yapısını direkt kullanır.

Dezavantajları:

- Tüm düğümleri dolaşmak gerekir.
- prev karşılaştırması gerektiği için global/bellekte iz tutmak gerekir.

3. Yöntem: Inorder Traversal + Stack (Iteratif)

Mantık:

2. yöntemin aynısıdır ama recursive değil, stack ile elle traversal yapılır. Yani current.left, stack.Push(), current = stack.Pop() şeklinde ilerler.

Avantajları:

- Recursive kullanılmaz, call stack yerine kendi stack yapısı var.
- Daha kontrollü çalışır, derinlik limiti yoktur.

Dezavantajları:

- Kod yapısı biraz daha karmaşıktır (stack, current, prev gibi değişkenler gerekir).

Kısıtlar ve Beklentiler

Girdi Büyüklüğü

- Girdi büyüklüğü, probleme verilen verinin ölçüsüdür. Bu problemde, girdi olarak bir ikili ağaç veriliyor.
- Girdi büyüklüğü = n (ağaçtaki toplam düğüm sayısı)

Zaman Karmaşıklığı

- BST’deki her düğüm yalnızca bir kez ziyaret edilir.
- Her düğüm için sabit işlemler yapılır (karşılaştırma, stack işlemi vs.).
- Bu yüzden toplam işlem sayısı, düğüm sayısı kadar olur. O(n)

Kısıtlar ve Beklentiler

Alan Karmaşıklığı

Alan karmaşıklığı (space complexity) bir algoritmanın çalışırken ne kadar ek bellek (RAM) kullandığını gösterir.

- Değişkenler,
- Fonksiyon çağrıları (özellikle recursive olanlar),
- Veri yapıları (stack, queue vs.) gibi şeyleri kapsar.

Eğer bir algoritma sadece birkaç değişken kullanıyorsa:

→ O(1) alan (sabit alan).

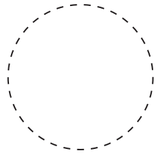
Ama eğer n elemanlı bir dizi veya n derinlikte recursive çağrılar kullanıyorsa:

→ O(n) alan.

Kısıtlar ve Beklentiler

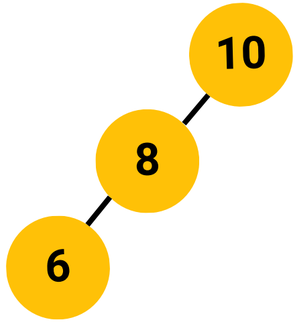
Yöntem	Zaman Karmaşıklığı	Dengeli Ağaçlarda Alan Karmaşıklığı	Dengesiz Ağaçlarda Alan Karmaşıklığı
Min-Max Recursive	O(n)	O(log n)	O(h)
Inorder Recursive	O(n)	O(log n)	O(h)
Inorder Iterative	O(n)	O(log n)	O(h)

Edge Caseler

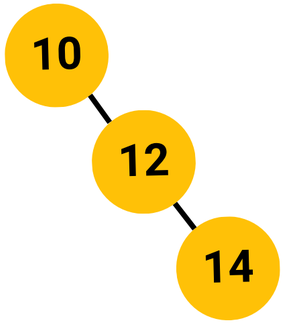


root == null (boş ağaç)

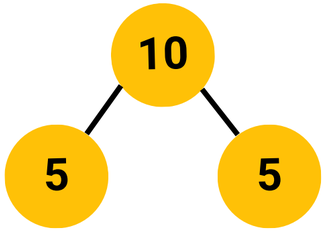
Tek düğümlü ağaç



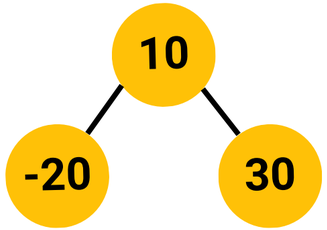
Tüm düğümler sol altta



Tüm düğümler sağ altta



Aynı değeri içeren iki düğüm



Negatif ve pozitif karışık değerler

Kullanılan Veri Yapıları

- **TreeNode**: Düğümler (val, left, right)
- **Stack**: İteratif çözüm için
- **List<int>** (bazı yöntemlerde tüm düğüm değerlerini saklamak için)

Gerçek Hayat Kullanım Senaryosu

- **Veri Tabanı Kayıtlarında Benzersizlik ve Sıralama:**

Müşteri, ürün ID veya kullanıcı adları BST ile sıralı ve benzersiz tutulur. Validate BST, yapının doğru olduğunu kontrol eder.

- **Log Dosyası Analizi ve Zaman Sıralaması:**

Loglar zaman damgasına göre BST'de sıralanır. Doğrulama, sıralamanın bozulmadığını garanti eder.

- **Benzersiz Müşteri/Ürün ID Kontrolü:**

Yeni ID eklenirken BST'de varlığı hızlıca kontrol edilir. Validate BST, yapının kurallara uygun olduğunu onaylar.

Sıkça Sorulan Sorular (SSS)

- Aynı değer iki kez olabilir mi?
- Null ağaç geçerli midir?
- Değer aralığı neden long?
- Ağaçta negatif değerler ve sıfır da olabilir mi?

Faydalandığımız Kaynaklar

- https://youtu.be/s6ATEkipzow?si=BoEQgK-_xsDbEO8B
- https://youtu.be/gGsEVFP0aQo?si=zTtrfr8CQXyPULV_
- https://www.youtube.com/watch?v=8HgTKh-ik30&list=PLK37qYAhi0EfUz9ztgca3sJYn68FIxWxk

Kaynak Kodlar

Kaynak kodlar için aşağıdaki bağlantıya tıklayabilir ya da QR okutabilirsiniz.

https://github.com/Bbusenur/ValidateBstProblem

Teşekkürler

Halide Ceyda Sarıçelik
Busenur Çil