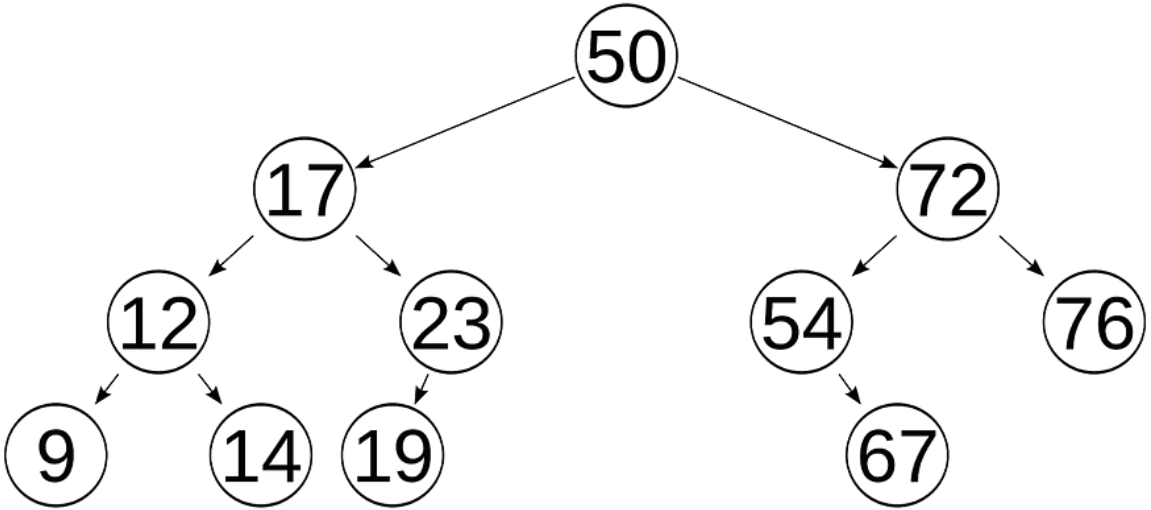
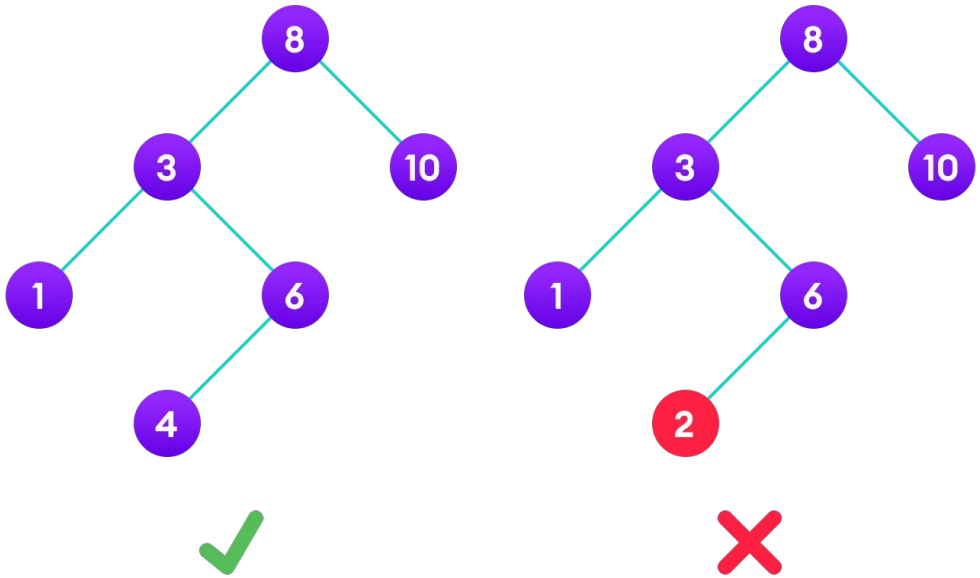
İkili Arama Ağaçları

Muhammet Baran Kuzu, 2111502013
Dostonbek Ismonov, 2111502234
Faruk Emre Şen, 2211502001
Serdar Kemal İncaman, 2211502009
Meriç Yanık, 2211502064

İkili arama ağacı, verileri sıralı bir şekilde düzenlemek ve depolamak için kullanılan bir veri yapısıdır. İkili arama ağacındaki her bir düğüm en fazla iki çocuğa sahip olabilir, sol çocuk ve sağ çocuk. Sol çocuk, ana düğümden daha küçük değerleri içerirken sağ çocuk, ana düğümden daha büyük değerleri içerir. Bu hiyerarşik yapı, ağaçta saklanan veriler üzerinde verimli bir şekilde arama, ekleme ve silme işlemlerinin gerçekleştirilmesini sağlar.



Sağ taraftaki ikili ağaç, bir ikili arama ağacı değildir. Çünkü, 3 değerinin bulunduğu düğümün sağ alt-ağacı kendisinden daha büyük değerleri içermelidir.



Kullanım Alanları

İkili Arama Ağaçlarının Uygulamaları

- **Öncelik kuyrukları:** İkili arama ağaçları, en yüksek önceliğe sahip öğenin ana düğümde bulunduğu ve alt-ağaçlarda daha düşük önceliğe sahip öğerin bulunduğu öncelik kuyruklarında kullanılabilir.
- **Arama motorları:** İkili arama ağaçları veri dizinlerini düzenli tutmak için kullanılır. Bu özellik arama motorlarının verileri hızlı bir şekilde bulmasına olanak tanır.
- **Veri depolama ve geri alma:** İkili arama ağaçları, belirli bir kaydın logaritmik sürede aranabildiği veritabanlarında olduğu gibi hızlı bir şekilde veri depolama ve veriyi çekmede kullanılabilir.
- **Oyun programlama:** İkili arama ağaçları, özellikle strateji ve rol yapma oyunlarında, yapay zeka karar mekanizmalarında, oyun içi varlıkları yönetmek ve hızlı karar alabilmek için kullanılabilir.

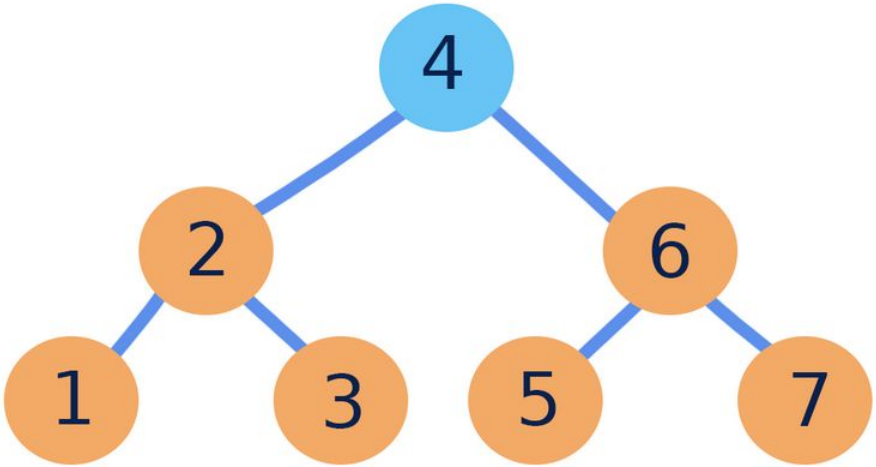
Zaman Karmaşıklığı

Zaman Karmaşıklığı

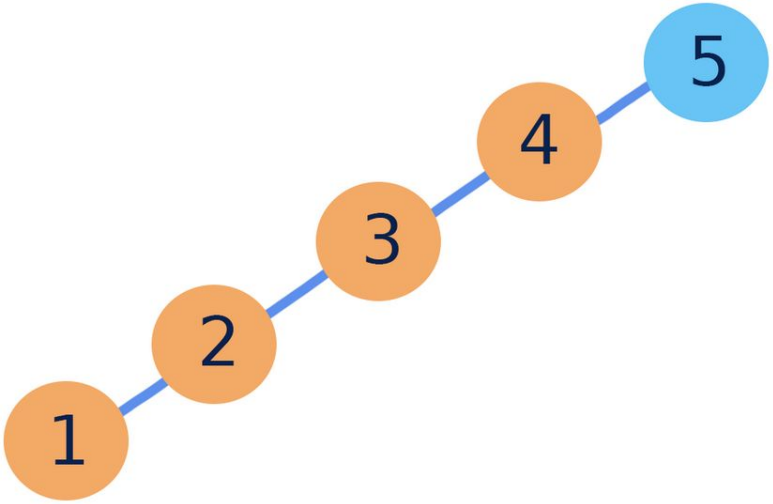
Eğer ağaç dengeli bir ağaçsa, yani sol ve sağ alt-ağaçlarının yükseklikleri arasındaki fark en fazla 1 ise, bu en iyi durumudur ve bu durumda zaman karmaşıklığı O(log n) (n, ağaçtaki düğüm sayısı) olur.

Eğer değerler artan veya azalan sıraya göre eklenirse, ağaç dengesiz hale gelecektir ve bu durumda ağaç bağlı listeye benzeyecektir. Dengesiz ağaç durumu en kötü durumdur ve zaman karmaşıklığı O(n) (n, ağacın yüksekliği) olur.

Sonuç olarak, ikili arama ağaçlarındaki işlemlerin optimize edilmesi için yükseklik minimize edilmeli, bunun için de ağaç dengeli bir ağaç olmalı.



Dengeli Bir İkili Arama Ağacı Örneği
Zaman Karmaşıklığı O(log n)



Dengesiz Bir İkili Arama Ağacı Örneği
Zaman Karmaşıklığı O(n)

Avantaj ve Dezavantajları

Avantajlar

- **Hızlı arama:** İkili arama ağacındaki belirli bir değeri aramada zaman karmaşıklığı ortalama O(log n) olduğundan bir dizide veya bağlı listede aramadan çok daha hızlıdır, bağlı listelerin zaman karmaşıklığı O(n)'dir.
- **Sıralı dolaşma:** İkili arama ağaçlarında sıralı olarak dolaşılabilir. Önce sol alt-ağaç, sonra kök ve sonra sağ alt-ağaç ziyaret edilir. Bir veri kümesi sıralamada bu kullanılabilir.
- **Alan verimliliği:** Diziler ve bağlı listelerin aksine, gereğinden fazla bilgi saklamadıklarından hafıza konusunda verimlidirler.

Dezavantajlar

- **Dengesiz ağaçlar:** Eğer ağaç dengesiz bir ağaca dönüşürse, zaman karmaşıklığı O(n) olacağından, ağacın kullanımı verimsiz hale gelecektir.
- **Ek süre gerekliliği:** Ağacı dengede tutmak için kullanılabilecek algoritmalar ve veri yapıları, fazladan zaman ve bellek harcayabilir.
- **Verimlilik:** İkili arama ağaçları, çok sayıda kopyaya sahip veri kümelerinde, yer israfına sebep olduklarından verimli olmayacaktır.
- **Doğrudan erişim:** Ağacın hiyerarşik doğasından dolayı, dizilerin aksine, belirli bir indeksteki elemente doğrudan bir erişim sağlanamaz.

Temel İşlemler

Arama İşlemi

İkili arama ağaçlarından “arama ağacı” olarak bahsedilmesinin nedeni, belirli bir değeri aramayı sırasız bir ağaca göre daha verimli hale getirmesidir. İdeal bir ikili arama ağacında, aranan değere ulaşmak için tüm düğümleri dolaşmamıza gerek kalmaz. Aşamalar şu şekildedir.

1. Aranan değer ile ana düğüm değeri karşılaştırılır.
    - Eğer eşitse düğümün yeri döndürülür.
    - Aranan değer, ana düğüm değerinden daha küçükse sol alt-ağaca gidilir.
    - Aranan değer, ana düğüm değerinden daha büyükse sağ alt-ağaca gidilir.
2. Yukarıdaki prosedür, eşleşme bulunana kadar yinelemeli olarak tekrar edilir.
3. Eğer element bulunamazsa veya ağaçta yoksa NULL döndürülür.

Ekleme İşlemi

Yeni düğümler her zaman yaprak olarak eklenir. Yeni düğümün yerini bulmak için bir arama işlemi gerçekleştirilir. Aşamalar şu şekildedir;

1. Eklenecek değer ile ana düğüm değeri karşılaştırılır.
    - Eklenecek değer, ana düğüm değerinden daha küçükse sol alt-ağaca gidilir.
    - Eklenecek değer, ana düğüm değerinden daha büyükse sağ alt-ağaca gidilir.
2. Yukarıdaki işlemler uygun bir boş yaprak yeri bulunana kadar devam ettirilir ve yeni düğüm uygun yere eklenir.

Silme İşlemi

Belirli bir düğümü silerken, ağacın geri kalanının doğru bir şekilde sıralı kalmış olmasına dikkat edilir. 3 farklı senaryo karşımıza çıkabilir:

1. Düğüm yaprak olabilir: Bu durumda düğüm direkt olarak silinir.
2. Düğüm bir adet çocuğa sahip olabilir: Bu durumda hedef düğüm ile çocuk düğüm yer değiştirilir ve hedef düğüm silinir.
3. Düğüm iki çocuğa sahip olabilir: Bu durumda öncelikle hedef düğümün sol alt-ağacındaki en büyük düğüm veya sağ alt-ağacındaki en küçük düğüm bulunur, bu düğüm her zaman bir yaprak düğümdür. Bu düğüm ile hedef düğüm yer değiştirilir ve hedef düğüm silinir.

Pseudo Ve JAVA Kodu

```java
import java.util.Scanner;
class Node {
    public int data;
    public Node left, right;

    public Node(int newData) {
        data = newData;
        left = null;
        right = null;
    }
}
public class BinarySearchTree {
    static Node root;

    public BinarySearchTree( ) {
        root = null;
    }

    private boolean search(Node root, int data) {
        if(root == null)
            return false;
        else if(root.data == data)
            return true;

        if(data < root.data)
            return search(root.left, data);
        else
            return search(root.right, data);
    }
```

Bir düğümü ifade eden sınıf. Düğümde tutulan değerin değişkeni ile sağ ve sol child düğüm referanslarından oluşur. Yapıcı metot; parametreden gelen değeri data değişkenine atar, sağ ve sol child'lara null değerini atar.

Ağacı ifade eden sınıf. Node sınıfından bir kök düğüm referansına sahiptir. Yapıcı metot ile bu köke null değeri atanır.

Arama metodu. Parametreleriyle başlangıç düğümünü ve aranacak veriyi alır. Eğer metodun bulunduğu düğüm boş ise false döndürür. Boş değilse ve bu düğümün değeri aranan değere eşitse true döndürür. Eğer aranan değer bu düğümün değerinden küçükse, aramaya düğümün sol alt-düğümünden devam etmek üzere metot yinelemeli olarak çağırılır. Eğer aranan değer bu düğümün değerinden büyükse, aramaya düğümün sağ alt-düğümünden devam etmek üzere metot yinelemeli olarak çağırılır.

```java
private Node insert(Node node, int data) {
    if(node == null)
        node = new Node(data);

    else {
        if(data < node.data)
            node.left = insert(node.left, data);
        else
            node.right = insert(node.right, data);
    }
    return node;
}
```

Ekleme metodu. Parametreleri, eklenecek ağacın başlangıç düğümü ve eklenecek değerden oluşur. Eğer metodun bulunduğu düğüm null ise, yeni değer buraya eklenir. Null değilse ve eklenecek değer bu düğümün değerinden küçükse sol alt-düğümde yer aramak üzere metot yinelemeli olarak çağırılır. Değer düğümün değerinden büyükse de sağ alt-düğümden devam edilir. Başarılı bir ekleme işleminden sonra güncellenen alt-ağaç döndürülür.

```java
public Node minValueNode(Node node) {
    while(node.left != null)
        node = node.left;
    return node;
}
```

Düğümün halefini (sağ alt-ağaçtaki en küçük düğüm) bulan metot. Parametresinden gelen başlangıç düğümünden başlayarak sol çocuğu olmayan bir düğüme gelene kadar sol-alt ağaç üzerinde dolaşır. Bulduğunda bu düğümü döndürür.

```java
private Node delete(Node root, int data) {
    if(root == null)
        return root;

    if(data < root.data)
        root.left = delete(root.left, data);
    else if (data > root.data)
        root.right = delete(root.right, data);
    else {
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        Node temp = minValueNode(root.right);
        root.data = temp.data;
        root.right = delete(root.right, temp.data);
    }
    return root;
}
```

Silme metodu. Bir düğümü silerken karşımıza üç farklı senaryo çıkar: Düğümün hiç çocuğu olmaması, bir çocuğu olması ve iki çocuğu olması.

Metot parametre olarak başlangıç düğümünü ve silinecek veriyi alır. Eğer metodun bulunduğu düğüm null ise silinecek veri bulunamamıştır, null döner. Eğer aranılan değer bulunulan düğümün değerinden küçükse düğümü bulmak üzere sol alt-ağaca, büyükse sağ alt-ağaca bakılır. Silinecek düğüm bulunduğunda:

1- Çocuğu olmaması veya bir çocuğu olması durumu: Null olmayan child (veya null) döndürülür. Child, silinen düğüm ile değiştirilmiş ve hedef düğüm ağaçtan silinmiş olur.

2- İki çocuğu olması durumu: Bir geçici değişken tanımlanır, bu değişkene minValueNode metoduyla, sağ alt-ağaçtaki en küçük düğüm (halefi) bulunarak atanır. Silinecek düğümün değeri bu geçici değişkendeki değerle değiştirilir. Daha sonra halefi, metot yinelemeli olarak çağırılarak silinir.

Başarılı bir silme işleminden sonra güncellenmiş alt-ağaç döndürülür.

```
    void inOrderTraversal(Node root) {
        if (root != null) {
            inOrderTraversal(root.left);
            System.out.print(root.data + " ");
            inOrderTraversal(root.right);
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        BinarySearchTree tree = new BinarySearchTree();

        root = tree.insert(root, 17);
        root = tree.insert(root, 3);
        root = tree.insert(root, 45);
        root = tree.insert(root, 54);
        root = tree.insert(root, 78);
        root = tree.insert(root, 91);
        root = tree.insert(root, 23);
        root = tree.insert(root, 5);
        root = tree.insert(root, 30);
        root = tree.insert(root, 69);
        root = tree.insert(root, 42);

        System.out.print("Ağaçta sıralı dolaşma: ");
        tree.inOrderTraversal(root);
        System.out.println();

        System.out.print("\nAğaçta aramak üzere bir sayı girin: ");
        int element = sc.nextInt();
        System.out.println("Aradığınız değer " + ((tree.search(root, element) == true) ?
"bulundu." : "bulunamadı."));

        tree.delete(root, 23);
        System.out.print("\n23 sayısı silindikten sonra ağaçta sıralı dolaşma: ");
        tree.inOrderTraversal(root);
        tree.delete(root, 69);
        System.out.print("\n69 sayısı silindikten sonra ağaçta sıralı dolaşma: ");
        tree.inOrderTraversal(root);

        System.out.println();
    }
}
```

Sıralı dolaşma metodu. Parametresi ile bir kök düğüm alır. Null bir düğüme gelene kadar çalışır. Önce sol alt-ağaç yinelemeli olarak dolaşılır, sonra bulunulan düğüm yazdırılır ve daha sonra sağ alt-ağaç yinelemeli olarak dolaşılır.

```
Ağaçta sıralı dolaşma: 3 5 17 23 30 42 45 54 69 78 91

Ağaçta aramak üzere bir sayı girin: 23
Aradığınız değer bulundu.

23 sayısı silindikten sonra ağaçta sıralı dolaşma: 3 5 17 30 42 45 54 69 78 91
69 sayısı silindikten sonra ağaçta sıralı dolaşma: 3 5 17 30 42 45 54 78 91
```