Binary Search Tree

Binary Search Tree düğüm(node) tabanlı bir veri yapısı olup şu özelliklere sahiptir:

1. Bir düğümün sol alt ağacındaki düğümlerin hepsi, bu düğümden daha küçük değerlere sahiptir.
2. Bir düğümün sağ alt ağacındaki düğümlerin hepsi, bu düğümden daha büyük değerlere sahiptir.
3. Hem sol hem sağ alt ağaç da aynı zamanda Binary Search Tree olmalı.
4. Aynı düğümden birden fazla olmamalı.

Binary Search Tree ile şu işlemleri yapacağız:

1. Arama
2. Eleman ekleme
3. Min değer
4. Max değer

Ama işlemlerden önce düğümün bulunduğu node class'ını inceleyelim:

```
package BST;
public class Node {
    protected Node left;
    protected Node right;
    protected int value;
    public Node(int value){
        this.value=value;
    }
}
```

Arama

Arama recursive veya iterative bir işlem olabilir.

Önce gelen değeri root düğümün değeri ile karşılaştırıyoruz ve eğer değer root düğümünün değerine eşitse aramamız başarılı oldu. Değilse, değerin root düğümünden büyük mü küçük mü olduğuna bakıyoruz. Eğer küçükse sol altağaca gidiyoruz. Ve önceki işlemleri tekrarlıyoruz. Eğer büyükse sağ altağaca gidiyoruz ve yukarıdaki işlemleri tekrarlıyoruz. Arama işlemi O(logN) zaman istiyor.

Pseudo Kodu

Belirli bir arama.

1. Root düğümünden current(mevcut) düğüm olarak başla.
2. Eğer aranan değer current düğümün değeriyle uyuşuyorsa bulduk demektir.
3. Eğer aranan değer current düğümün değerinden büyükse
    a. Eğer current düğümü sağ çocuğa sahipse sağ çocuğu ara
    b. Aksi takdirde aranan değer ağaçta yoktur.
4. Eğer aranan değer current düğümün değerinden küçükse
    a. Eğer current düğümü sol çocuğa sahipse sol çocuğu ara
    b. Aksi takdirde aranan değer ağaçta yoktur.

Java implementasyonu

```
public void search(Node n, int value){
    if(n.value == value || n==null){
```

```
            System.out.println("\nFound Value: " + n.value);
        }else if(value<n.value){
            search(n.left,value);
        }else {
            search(n.right,value);
        }
}
```

Eleman ekleme

Eleman ekleme işlemi de arama işlemi gibi davranır. İlk olarak, eklenecek değerin root ile aynı olup olmadığını karşılaştırır, değilse sağ veya sol altağaçlardan birini gelen değerin roottan büyük veya küçük olmasına bağlı olarak verilen sırasıyla seçeriz.
Sonunda düğümün değerine göre soluna veya sağına eklenecek bir düğüme ulaşırız.
Bu da bir recursive işlemdir. Roottan başlarız ve elemanı ekleyeceğimiz doğru yeri buluncaya kadar dallanırız. Bu işlem ağacın yüksekliğine bağlı olarak O(logN) zaman alır.

Pseudo Kodu

Her zaman yeni düğümü yaprak düğümü olarak ekle.

1. Roottan current düğüm olarak başla
2. Eğer yeni düğümün değeri<current'ın değeri
    a. Eğer current sol çocuğa sahipse solu ara
    b. Aksi takdirde sol çocuk olarak yeni elemanı ekle
3. Eğer yeni düğümün değeri>current'ın değeri
    a. Eğer current sağ çocuğa sahipse sağı ara
    b. Aksi takdirde sağ çocuk olarak yeni elemanı ekle

Java implementasyonu

```
public void insert(Node n, int value){
    if(value < n.value){
        if(n.left != null){
            insert(n.left,value);
        }else{
            n.left=new Node(value);
        }
    }
    if(value > n.value){
        if(n.right != null){
            insert(n.right,value);
        }else{
            n.right=new Node(value);
        }
   }
}
```

Min Max değer

Bu işlemde ağaçtaki minimum ve maksimum değerleri buluyoruz. Bu işlem de O(logN) zaman tutar.

Java implementasyonu

```
public int minValue(Node n){
    while(n.left!=null){
        n=n.left;
    }
    return n.value;
}
public int maxValue(Node n){
    while(n.right!=null){
        n=n.right;
    }
    return n.value;
}
```