VERİ YAPILARI DERS NOTLARI

BAĞLI LİSTELER:

Tek Yönlü Bağlı Listeler:

-Bağlı Listelerde Düğüm Yapısı:

Bağlı Listelerde Düğüm sınıflarımızda o düğüm de tutacağımız Düğüm verisi ve bir sonraki düğüm adresini gösteren bir pointerımız olmalıdır.

-Bağlı Listede Arama Silme Ekleme işlemleri:

Bağlı listede Arama silme ve ekleme işlemlerini yapan fonksiyonlar içinde düğüm cinsinden geçici bir nesne oluştururuz ve istediğimiz düğüme ulaşmak için bu nesneyi kullanırız.

Ekleme fonksiyonu: Genel olarak bu fonksiyona parametre olarak eklemek istediğimiz düğüm verisi göndermemiz yeterlidir.

Oluşturacağımız geçici değişken ile son düğüme ulaşma işlemi

```
while (geciciDugum->getSonrakiDugum() != NULL)
{
    geciciDugum = geciciDugum->getSonrakiDugum();
}
```

Şeklinde düğümlerde ilerleme sağlıyabiliriz. Oluşturduğumuz yeni düğümüde

```
geciciDugum->setSonrakiDugum(yeniDugum);
```

Silme fonksiyonunda geçici nesneden dışında bir nesne daha oluştuturuz ve bu nesne geçici nesnenin bir önceki adresini bulundurur.

```
do
{
    if (geciciDugum->getDugumVerisi() == silinecekDugumVerisi)
    {
        break;
    }
    oncekiDugum = geciciDugum;
    geciciDugum = geciciDugum->getSonrakiDugum();
} while (geciciDugum != NULL);
```

Şeklindeki işlemle aradığımız düğümü bulma işlemi tamamlanır. Daha sonra

```
oncekiDugum->setSonrakiDugum(geciciDugum->getSonrakiDugum());
delete geciciDugum;
```

Şeklindeki işlemle bulunan düğümü silme işlemi yapılır.

Arama fonksiyonumuzuda yukarıdaki döngü ile halledebiliriz.

Çift Yönlü Bağlı Liste:

Çift yönlü bağlı listelerde düğüm yapısı: Tutacağımız düğüm verisi sonraki düğüm ve önceki düğüm adresleri olabilir.

Genel olarak tek yönlü bağlı liste ile arasında pek fark yoktur.

Arama fonksiyonda aynı döngü ile son düğüme gidip

```
geciciDugum->setSonrakiDugum(yeniDugum);
yeniDugum->setOncekiDugum(geciciDugum);
sonDugumAdresi=yeniDugum;
```

İşlemleriyle düğüm bağlantılarını kurabiliriz.

Silme fonksiyonunda herhangi bir değişiklik yapmamıza gerek yoktur.

STACK

Static Dizi İle Stack Yapısı:

Stack yapımızda elemansayımızı ve verileri depolayacağımız bir dizi olmalıdır.

Push fonksiyonumuz `depolamaBirimi[elemanSayisi++] = deger;`

Pop fonksiyomuz `--elemanSayisi;`

Şeklinde olabilir.

Dinamik Dizi ile Stack Yapısı:

Static diziden tek farkı diziyi heap bölgesinde oluşturmamızdır. İşlemler aynıdır.

Düğüm ile Stack Yapısı:

Bunun için oluşturduğumuz düğüm sınıfını kullanabiliriz. Tek ihtiyacımız olan basaEkle ve bastanSil fonksiyonlarını eklememiz olacaktır.

Stack sınıfımızda depolamaBirimi mizin türü Bağıl liste türünde olmalıdır ve ekleme silme işlemlerinde eleman sayısında oynama yaparak basaEkle ve bastanSil fonskiyonlarıyla işimizi hallederiz.

```
elemanSayisi++;
depolamaBirimi->basaEkle(deger);

--elemanSayisi;
depolamaBirimi->bastanSil();
```

İlki ekleme ikinci silme işlemi içindir.

İKİLİ ARAMA AĞACI

İkili Arama Ağaçlarının genel yapısı şu şekildedir;

Ağacımızı bir düzene göre oluştururuz. İkili arama ağaçlarında eklenecek düğüm verisi karşılaştırma yaptığımız düğüm verisinden küçükse düğümün soluna büyükse düğümün sağına yeni düğüm olarak ekleriz.

İkili arama ağaçlarında düğüm yapımız o düğümde tutacağımız veri ve sağ ve sol düğümleri gösterecek birer pointerımız olması yeterlidir.

```
string veri;
Dugum *sol;
Dugum *sag;
```

İkili arama ağacımızda ise değişken olarak başdüğüm adresini tutmamız yeterlidir.

```
Dugum *root;
```

Diğer işlemlerimizi yapabileceğimiz fonksiyonlarda bu sınıfta yer almalıdır.

Fonksiyonlarmızın genel yapısı recursive dir. Bu fonksiyonlarda recursive olmasının mantığı ise bağlı listedeki temel while döngüsü mantığıyla aynıdır. Yani istediğimiz düğüme ulaşana kadar düğümlerde gezmemiz bu işlemiz tek işlemle halletmemiz zordur bunun için bir döngü ya da recursive bir işlem kullanabiliriz.

Ekleme Fonksiyonu:

Bu fonskiyonun mantığı genel olarak şöyledir;

O an üzerinde durduğumuz düğümün verisi ile ekleyeceğimiz düğüm verisi karşılaştırılır. Eğer ekleyeceğimiz düğüm verisi daha küçükse üzerinde durduğumuz düğümü sol düğümü olarak ağaca eklenir eğer büyükse sağ düğüm olarak ağaca eklenir.

```
if(alt_root == NULL) alt_root = new Dugum<Nesne>(yeni); // recursive bitirme işlemi
//Eğer o an üzerinde durduğumuz düğüm düğüm boşsa artık buraya ekleme yapabiriz.
else if(yeni <= alt_root->veri) // eğer ekleyeceğimiz düğüm verisi daha küçükse o düğümün soluna gideriz.
    AraveEkle(alt_root->sol,yeni);//recursive işlem
// fonskiyona parametre olarak alt_root'un sol düğüm adresi gönderilmiştir yani artık fonkiyon arama ve ekleme
//  işlemlerini alt_root da değil alt_root'un sol düğümünde gerçekleştirecektir.
//Bu şekilde düğümler arasında hareket edebiliriz.
else if(yeni > alt_root->veri)//eğer ekleyeceğimiz düğüm verisi daha büyükse o düğümün sağına gideriz.
    AraveEkle(alt_root->sag,yeni);//recursive işlem
// Yukarıdaki açıklama burada da geçerlidir.
```

Arama ve Silme Fonksiyonu:

Bu fonskiyonun genel mantığı şu şekildedir;

Yine bi üsteki gibi o an üzerinde durduğumuz düğüm verisiyle sileceğimiz düğüm verisin karşılaştırırız ve o düğümün sağına mı soluna mı gideceğimizi belirleriz. Düğümlerde sağ veya sola gitme işlemlerini yine aynı recursive yapı ile halledebiliriz ve sileceğimiz düğüm verisini bulduğumuzda silme fonskiyonunu çağırırız.

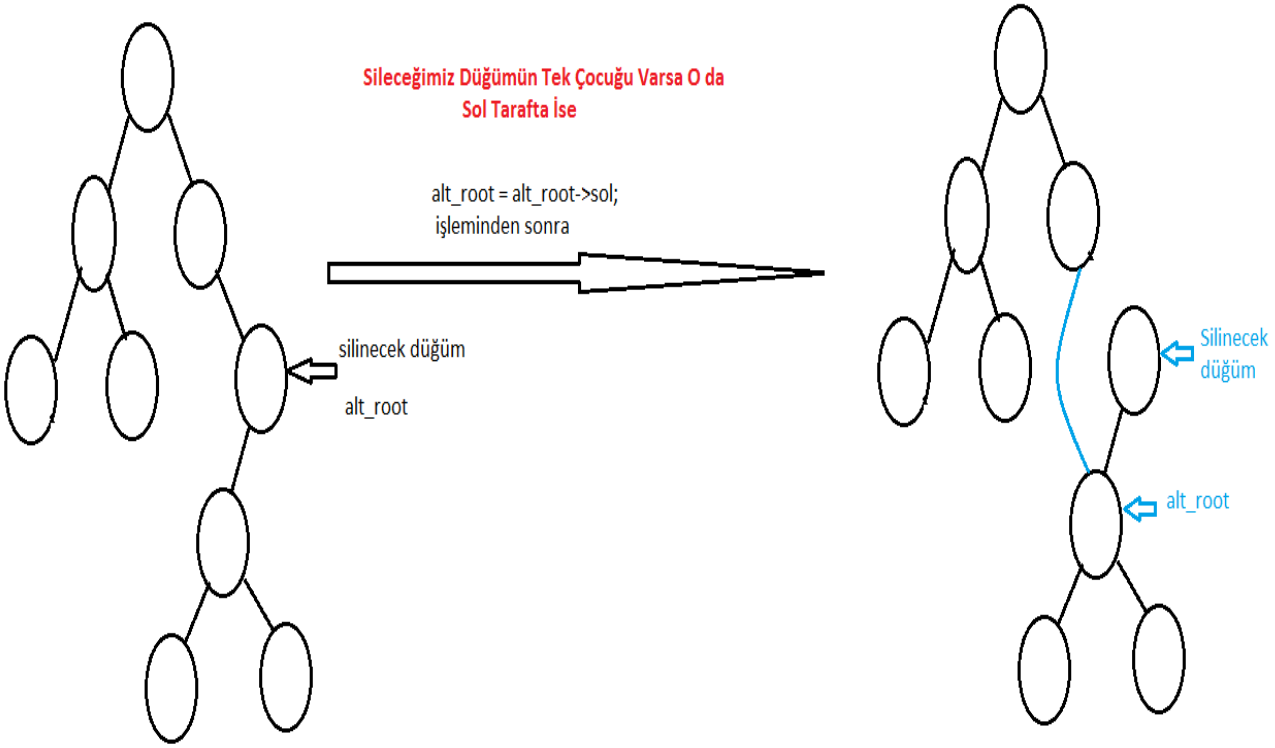
```
if(alt_root == NULL || alt_root->veri == yeni)
    return DugumSil(alt_root);//Aradığımız düğüm verisi bulunduğunda Dugum Silme fonksiyonu çağrılır.
else if(yeni < alt_root->veri)
    return AraveSil(alt_root->sol,yeni);
else
    return AraveSil(alt_root->sag,yeni);
```

Düğüm silme Fonksiyonumuzun mantığı

Düğüm silerken üzerinde durmamız gereken temel üç konu vardır.

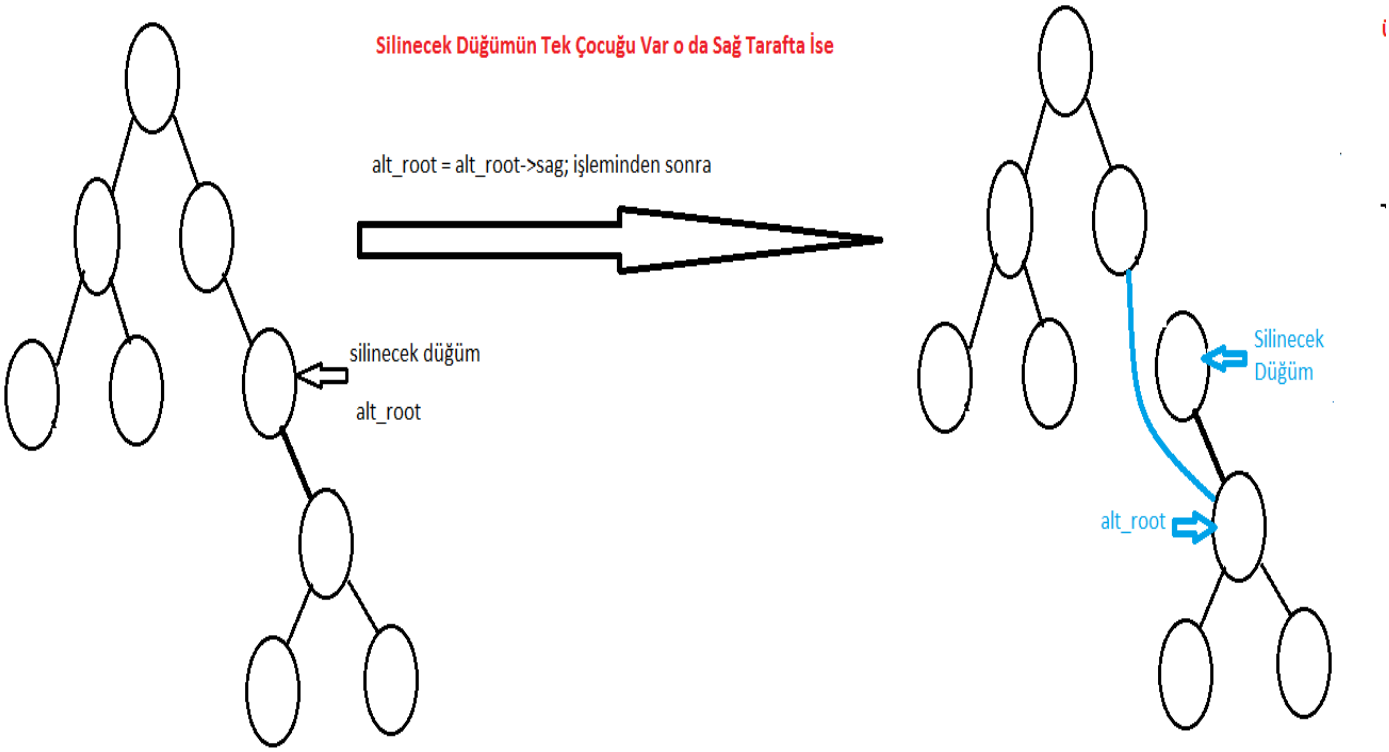
1. Silinecek düğümün sadece sol çocuğu mu var? Yapmamız gereken işlem.

Şekil 1



2. Silinecek düğümün sadece sağ çocuğu mu var? Yapmamız gereken işlem.

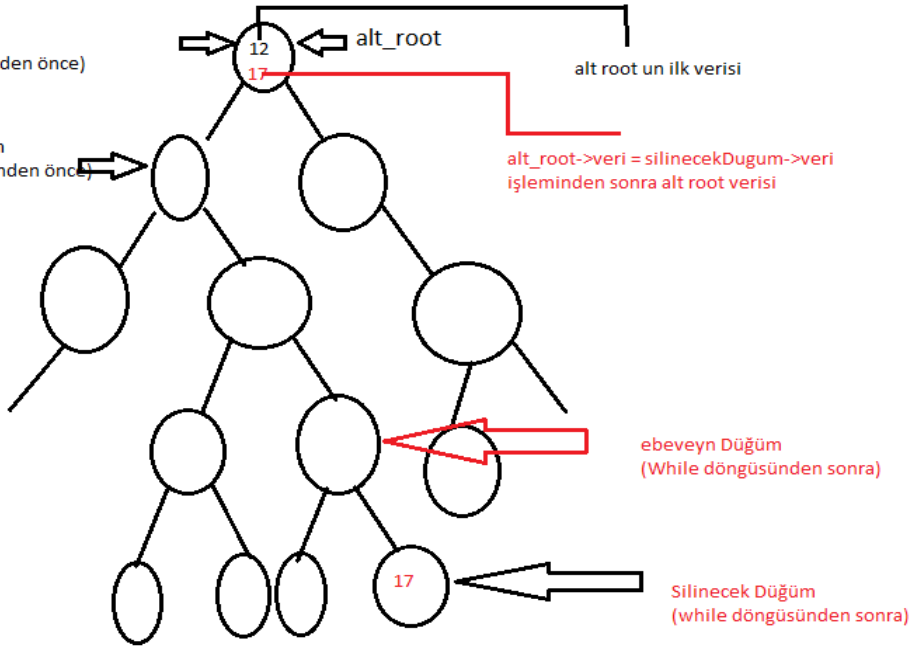
Şekil 2

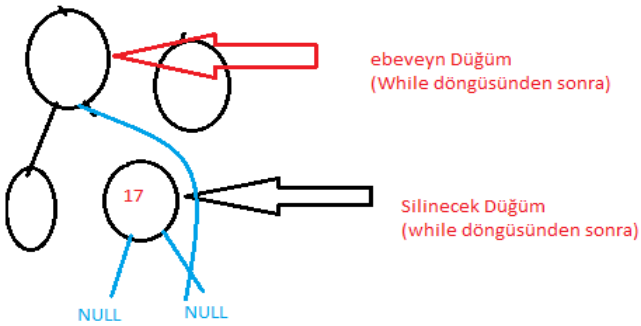


3. Silinecek Düğümün 2 çocuğu da var mı? Yapmamız gereken işlem.

Şekil 3

Burada iki çocuğu olan alt_root düğümünü silelim.
Yapmamız gereken şey alt_root un sağındaki düğümün en solunda ki düğüme gitmek(çünkü orada alt_root yerine yerleştirilecek en uygun elemanı bulucaz.)
Kırmızı renkli silinecek düğüm aradığımız yeri işaret eder. O düğümdeki veriyi alıp alt_root'a eşitledik. (alt_root'un yeni verisi kırmızı renkli olan 17 oldu.)
Aslında bu işlemle alt_root düğümünü silmedik ağaçda ki en uygun düğümü alt_root'un yerine yazdık ve silmemiz gereken düğüm ise alt_root'a kopyaladığımız düğüm olmalıdır. Yani kırmızı renkli silinecek düğümün işaret ettiği düğümü silmeliyiz.
Hemen altta yapılan işlem budur.
Ebeveyn düğümün sağ düğümü silinecek düğümü işaret ederken yapılan işlemler silinecek Düğümün solunu yani NULL 'ı işaret eder böylece aradaki bağı koparmış oluruz.
ebeveynDugum->sag = silinecekDugum->sol





İşlemlerin kodu

```
if(alt_root == NULL) return false; //Parametre olarak gelen adres boşsa silme yapamayız

Dugum<Nesne> *silinecekDugum = alt_root;// üzerinde işlem yapacağımız geçici nesne

if(alt_root->sag == NULL) //eğer silinecek düğümün tek çocuğu varsa o da sol tarafında ise
alt_root = alt_root->sol; //işlem için şekil 1 e bak.
else if(alt_root->sol == NULL)//eğer silinecek düğümün tek çocuğu varsa o da sağ tarafta ise
alt_root = alt_root->sag;//şekil 2
else{   // eğer silinecek düğümün sağ ve solunda düğüm varsa şekil 3
    silinecekDugum = alt_root->sol;
    Dugum<Nesne> *ebeveynDugum = alt_root;
    while(silinecekDugum->sag != NULL){//alt_root'un solundaki en sağ düğüme ulaşmak için
        ebeveynDugum=silinecekDugum; // ebeveyn düğüm her zaman silinecek düğümün bir üst düğümü
        silinecekDugum=silinecekDugum->sag;
    }
    alt_root->veri = silinecekDugum->veri;// alt_root'a alt_root'un solundaki en sağ düğümü kopyalıyoruz.
    if(ebeveynDugum == alt_root) alt_root->sol = silinecekDugum->sol;// while döngüsüne girmezse .
    //Silinecek düğümle aradaki bağı kopartıyoruz
    else ebeveynDugum->sag = silinecekDugum->sol;// silinecek düğümle aradaki bağı kopartıyoruz
}
delete silinecekDugum; // gerekli düğümü siliyoruz
```

Yazdırma Metodları:

3 tane yazdırma metodumuz vardır:

İnorder;

```
if(alt_root != NULL){
    Inorder(alt_root->sol);
    cout<<alt_root->veri<<" ";//Bayrak ortada
    Inorder(alt_root->sag);
}
```

Preorder;

```
if(alt_root != NULL){
    cout<<alt_root->veri<<" ";//Bayrak en başta(sol tarafta)
    Preorder(alt_root->sol);
    Preorder(alt_root->sag);
}
```

Postorder;

```
if(alt_root != NULL){
    Postorder(alt_root->sol);
    Postorder(alt_root->sag);
    cout<<alt_root->veri<<" ";//Bayrak en sonra (sag tarafta)
}
```

HEAP AĞACI

Heap ağaçları iki türlüdür.

Minimum heap: elemanları yukardan aşağı doğru gittikçe daima büyür. Yani en küçük eleman en baştadır ve elemanlar değerleri alt düğümlere indikçe artar.

Maximum heap: bu heap ağacı ise minumum heap ' ın tam tersidir.

Heap Ağaçlarında düğümlerden ziyade dizilerle uğraşırız. Veriler arasındaki bağlantıları dizilerin indeksleri üzerinden sağlarız. Örn; bir verinin dizideki yerine i indeksi dersek sağ çocuk indeksi 2*i+2 , sol çocuk indeksi 2*i+1 , ebeveyn indeksi (i-1)/2 olur.

Heap Ağacı sınıfımızda;

Değişken olarak dizideki elemanların yerlerini kontrol edip ekleme çıkarma yapmak için eleman sayisi ve verileri depolayacağımız bir dizi değişkeni oluşturmamız yeterlidir.

Bunun haricinde yardımcı fonksiyon olarak;

```
int Heap::SolCocukGetir(int i)
{
    return 2*i+1;
}
int Heap::SagCocukGetir(int i)
{
    return 2*i+2;
}
int Heap::EbeveynGetir(int i)
{
    return (i-1)/2;
}
```

fonksiyonlarını kullanıcağız.

Ekleme Fonksiyonu:

Ekleme fonksiyonunda yapmamız gereken veriyi dizinin sonuna direk olarak eklemek ve yukarı heap fonksiyonun çağırıp ağacı düzenlemek olacaktır. Çünkü Minimum heap de aşağı doğru indikçe veriler daima artıyordu ekliyeceğimiz veri küçük değere sahip olabilir bu yüzden ağacımızın yapısı bozulabilir. Ağacımızın yapısını yeniden düzenlemek için de yukarıya heap fonksiyonunu çağırırız.

```
if(ES==MAX)
    return false;
ES++; // eleman sayisi 1 artti (stack deki gibi)
Veri[ES-1] = e; // Eleman diziye eklendi
HeapifyUp(ES-1); // ağacı yeniden düzenlemek için
return true;
```

HeapifyUp fonksiyonu recursive bir fonksiyondur. Yaptığı işlem gönderilen parametreki veri ile ebeyninin verisini karşılaştırır eğer ebeveyn daha büyükse yer değiştitirler(Minimum heap a göre). Bu işlemler bitirme koşulları sağlanana kadar devam eder.

```
{ // Bu fonskiyon parametre olarak gelen indeksde ki veriyle ebeveyn deki veriyi karşılaştırır
// Eğer eklenecek veri daha küçükse ebeveynle eklediğimiz veri yer değiştirir.
    if(i==0) // fonksiyonu bitirme koşulu
        return;

    int eb = EbeveynGetir(i);

    if(Veri[eb]>Veri[i])// ebeveyn eklenecek veriden daha küçükse
    {
        int temp    = Veri[i];
        Veri[i]     = Veri[eb];
        Veri[eb]    = temp;
        HeapifyUp(eb); // fonksiyon yeniden çağrılır.
    }
}
```

Çıkarma Fonksiyonu:

Bu fonskiyonda yapmamız gereken şey çıkarıcağımız eleman her zaman dizinin başındaki elemandır. Burada bir sorun yok fakat agacımızın kökünü çıkardık böyle bir durum olmamalı. Bunun için geçici olarak dizinin son elemanını dizinin baş elemanı yapıyoruz. Fakat alt tarakdaki düğüm verileri üst taraftaki düğüm verilerinden daha büyük olduğu için bu sefer ağaç yapımız bozuldu. Ağacımızı yeniden düzenlememiz gerekli bunun için Aşağı doğru heap fonksiyonunu kullanırız.

```
if(ES==0)
    return false;
e = Veri[0];
Veri[0] = Veri[ES-1]; // en sondaki elemanı en başa alırız
ES--; // eleman sayisini azaltırız
HeapifyDown(0); // Ağacı Aşağı doğru heapleriz.
return true;
```

HeapifyDown Fonksiyonu:

Bu fonskiyona yine parametre olarak bir dizi indeksi göndeririz.

```
int Sol = SolCocukGetir(i); // sol çocuk indeksi
int Sag = SagCocukGetir(i); // sağ çocuk indeksi
int min;

if(Sag>=ES) // Sağ Çocuk var mı yok mu kontrol ediyoruz
{
    if(Sol>=ES)// Sağ Çocuk olmaması sol cocuk olmayacağı anlamına gelmez
    // Bu yüzden sol çocuğuda kontrol ediyoruz
        return;
    else
        min = Sol; // eğer sadece sol çocuk varsa min değerizi sol çocuk olur
}
else // Eğer iki çocukda varsa
{
    if(Veri[Sol]<Veri[Sag])// sağ ve soldaki verileri karşılaştırırız
    //Hangisi daha küçükse min değerimiz o olur
        min = Sol;
    else
        min = Sag;
}
ıkaridaki işlemlerde tamamen min değerinin bulmak için uğraştık şimdi yer değiştirelim
if(Veri[min]<Veri[i])//eğer üzerinde bulunduğumuz veri daha büyükse değiştirme işlemi olacak
{
    int temp    = Veri[i];
    Veri[i]     = Veri[min];
    Veri[min]   = temp;
    HeapifyDown(min);// Bu fonksiyon bir düğümün sağ ve sol çocuklarının olmaması durumuna kadar kendini yeniler.
}
```