Heap Uygulamaları

MaxHeap oluşturmak için ilk olarak elemanların tutulacağı Heap isimli bir dizi, dizinin boyutunu belirten size değişkeni ve heap için maksimum boyutu belirten maxsize değişkeni tanımlanmış ve boş bir heap oluşturacak şekilde yapıcı metot oluşturulmuştur.

```
public class MaxHeap {
    private int[] Heap;
    private int size;
    private int maxsize;

public MaxHeap(int maxsize) {
        this.maxsize = maxsize;
        this.size = 0;
        Heap = new int[this.maxsize];
}
```

Verilen bir sıra numarasındaki düğümün ebeveyni, sol çocuğu ve sağ çocuğunun sıra numaralarını döndürmek için metotlar yazılmıştır.

```
public int parent(int pos) {
        return (pos-1) / 2;
}

public int leftChild(int pos) {
        return (2 * pos) + 1;
}

public int rightChild(int pos) {
        return (2 * pos) + 2;
}
```

MaxHeap ekleme metodunda ilk olarak maksimum boyuta ulaşılıp ulaşılmadığı kontrol edilmiştir. Ulaşılmamış ise dizinin sonuna yeni eleman eklenmiştir. Son elemanın sıra numarası current değişkenine alınmıştır. Döngü içerisinde eklenen düğümün içeriği ile ebeveyninin içeriği karşılaştırılmış ve büyük ise MaxHeap özelliğinin sağlanması için swap metodu ile bu düğümler yer değiştirilmiştir. Yaprak düğümler kök düğümden küçük olana kadar yer değiştirmeler yapılmış ve son olarak dizi boyutu bir arttırılmıştır.

```
public void insert(int element) {
        if (size >= maxsize)
        return;

        Heap[size] = element;
        int current = size;
        while (Heap[current] > Heap[parent(current)]) {
                swap(current, parent(current));
                current = parent(current);
        }
            size++;
}
```

Yer değiştirme yapılan swap metodunda verilen sıra numaralarındaki düğümlerin içerikleri değiştirilmiştir.

```
public void swap(int fpos, int spos) {
        int tmp;
        tmp = Heap[fpos];
        Heap[fpos] = Heap[spos];
        Heap[spos] = tmp;
}
```

MaxHeap için en büyük değer kök düğümündedir. En büyük değeri silmek için ağaçtan kök düğümü çıkarmak gerekir. Kök düğüm dizinin 0 sıra numarasındadır. Bu değer bir değişkene alınarak geri döndürülür. Yeni kök değeri dizinin en sonundaki eleman yapılır. Ancak bu şekilde MaxHeap özelliği bozulmuş olur ve kök düğümde daha küçük değer yer alır. Bu yüzden maxHeapify metodu ile heap üzerinde gerekli değişiklikler yapılır.

```
public int extractMax() {
        int e = Heap[0];
        Heap[0] = Heap[size--];
        maxHeapify(0);
        return e;
}
```

Sıra numarası verilen düğüm yaprak değilse düzenleme yapılır. Eğer ilgili sıra numarasındaki düğümün sağ ve sol çocukları kendisinden büyük ise bu çocuklardan daha büyük olanı ile yer değiştirilir. Bu işlem sonucunda MaxHeap özelliği korunmuş olur.

```
public void maxHeapify(int pos) {
        if (isLeaf(pos))
        return;

        if (Heap[pos] < Heap[leftChild(pos)] ||
                Heap[pos] < Heap[rightChild(pos)]) {

                if (Heap[leftChild(pos)] > Heap[rightChild(pos)]) {
                        swap(pos, leftChild(pos));
                        maxHeapify(leftChild(pos));
                }
                else {
                        swap(pos, rightChild(pos));
                        maxHeapify(rightChild(pos));
                }
        }
```

Yaprak düğümler dizinin son yarısında yer alır. Verilen sıra numarasındaki düğümün sıra numarası dizinin ikinci yarısında yer alıyorsa yaprak düğüm olarak ifade edilmiştir.

```
public boolean isLeaf(int pos) {
        if (pos >= ((size-1) / 2) && pos <= (size-1) ){
        return true;
        }
        return false;
        }
```

MaxHeap te k. sıradaki maksimum elemanı elde etmek için k-1 kere kök eleman çıkarılmıştır. Daha sonra çıkacak olan eleman k. maksimum elemandır.

```
public int kMax(int k){
        for (int i = 1; i < k; i++) {
            extractMax();
        }
        return extractMax();
}
```

Kök düğümler yaprak düğümlerden büyük olduğundan minimum eleman yaprak düğümlerde aranmıştır. İlk olarak dizinin ortasındaki eleman minimum olarak belirlenmiş daha sonra dizinin ikinci yarısındaki elemanlar ile karşılaştırılmıştır. Daha küçük bir eleman bulunuyorsa minimum olarak değiştirilmiş ve bu eleman geri döndürülmüştür.

```
public int findMinimumElement(){
        int minimumElement = Heap[size/2];
        for (int i = (size/2)+1; i < size; ++i)
            if(Heap[i]<minimumElement){
                minimumElement = Heap[i];
            }
         return minimumElement;
    }
```

MaxHeap için yazılan tanımlamalar ve metotlar aynı şekilde MinHeap için de oluşturulmuştur. Ekleme metodunda döngü içerisinde eklenen düğümün içeriği ile ebeveyninin içeriği karşılaştırılmış ve küçük ise MinHeap özelliğinin sağlanması için swap metodu ile bu düğümler yer değiştirilmiştir. Yaprak düğümler kök düğümden büyük olana kadar yer değiştirmeler yapılmış ve son olarak dizi boyutu bir arttırılmıştır.

```
public void insert(int element) {
    if (size >= maxsize) {
        return;
        }
        Heap[size] = element;
        int current = size;
        while (Heap[current] < Heap[parent(current)]) {
                swap(current, parent(current));
                current = parent(current);
        }
            size++;
}
```

MinHeap için en küçük değer kök düğümündedir. En küçük değeri silmek için ağaçtan kök düğümü çıkarmak gerekir. Kök düğüm dizinin 0 sıra numarasındadır. Bu değer bir değişkene alınarak geri döndürülür. Yeni kök değeri dizinin en sonundaki eleman yapılır. Ancak bu şekilde MinHeap özelliği bozulmuş olur ve kök düğümde daha büyük değer yer alır. Bu yüzden minHeapify metodu ile heap üzerinde gerekli değişiklikler yapılır.

```
public int extractMin(){
        int e = Heap[0];
        Heap[0] = Heap[size--];
        minHeapify(0);
        return e;
}
```

Sıra numarası verilen düğüm yaprak değilse düzenleme yapılır. Eğer ilgili sıra numarasındaki düğümün sağ ve sol çocukları kendisinden küçük ise bu çocuklardan daha küçük olanı ile yer değiştirilir. Bu işlem sonucunda MinHeap özelliği korunmuş olur.

```
public void minHeapify(int pos) {
    if (!isLeaf(pos)) {
        if (Heap[pos] > Heap[leftChild(pos)]
                || Heap[pos] > Heap[rightChild(pos)]) {

            if (Heap[leftChild(pos)] < Heap[rightChild(pos)]) {
                swap(pos, leftChild(pos));
                minHeapify(leftChild(pos));
            }
            else {
                swap(pos, rightChild(pos));
                minHeapify(rightChild(pos));
            }
        }
```

Girilen bir değerden küçük olan elemanları elde etmek için yazılan metot x değeri ve kök düğümünün sıra numarasını parametre olarak almıştır. Değer ilk olarak kök ile karşılaştırılmış ve kök büyük ise çıkılmıştır. Çünkü MinHeap için kökte en küçük değer yer almaktadır ve yapraklar daha büyük değerler olacağı için bakmaya gerek yoktur. Eğer kök küçük ise yazdırılmıştır. Daha sonra sol ve sağ çocukları ile karşılaştırılmaya devam edilmiş ve küçük olan değerler yazdırılmıştır.

```
public void printSmallerThan(int x, int pos) {
    if (pos > size)
        return;

    if (Heap[pos] >= x)
        return;

        System.out.print(Heap[pos] + " ");

        printSmallerThan(x, leftChild(pos));
        printSmallerThan(x, rightChild(pos));
    }
```

Oluşturulan MinHeap i MaxHeap e dönüştürmek için yaprak düğümler yukarı doğru taşınmıştır. Bir düğümün sol ve sağ çocuklarından hangisi daha büyük ise onunla yer değiştirilerek büyük değerlerin köklere doğru taşınması sağlanmıştır.

```
public void convertMaxHeap(int arr[], int n) {

    for (int i = (n-2)/2; i >= 0; --i)
        MaxHeapify(arr, i, n);
    }

public void MaxHeapify(int arr[], int i, int n) {
        int l = leftChild(i);
        int r = rightChild(i);
        int largest = i;
        if (l < n && arr[l] > arr[i])
                largest = l;
        if (r < n && arr[r] > arr[largest])
                largest = r;
        if (largest != i) {
                swap(i, largest);
                MaxHeapify(arr, largest, n);
        }
}
```