BEYKOZ ÜNİVERSİTESİ

MÜHENDİSLİK VE MİMARLIK FAKÜLTESİ

YAZILIM MÜHENDİSLİĞİ

BAHAR DÖNEMİ 2021-2022

# ALGORİTMA ANALİZİ

## HEAP SORT ALGORİTMASI ANALİZ

**PROJE RAPORU**

**HAZIRLAYAN:**

BEDİRHAN BÜYÜKÖZ

YAZILIM MÜHENDİSLİĞİ III.SINIF

1904040009

**HEAP SORT ALGORİTMASI**

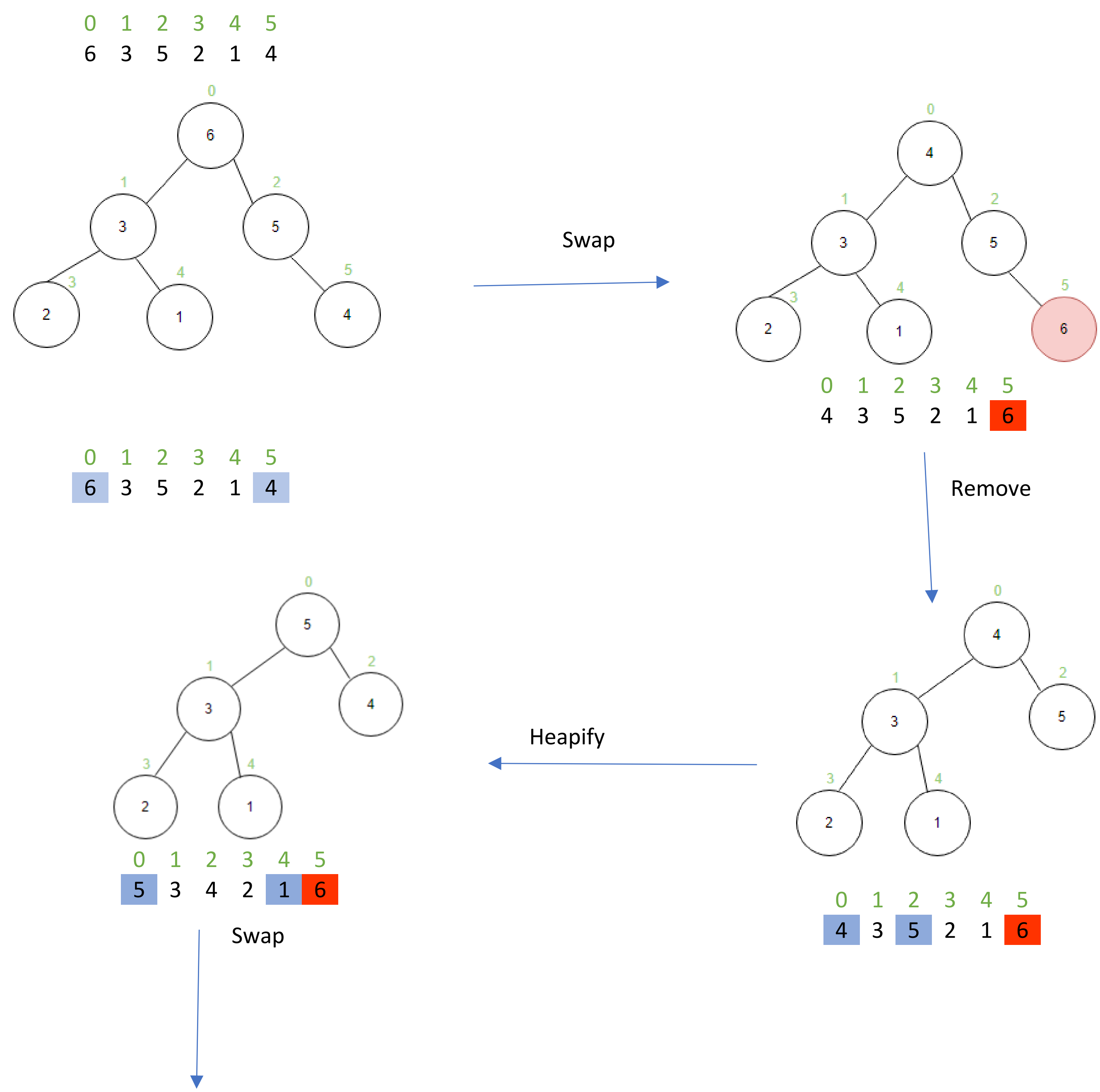
Verileri bellekte böl ve fethet yöntemi kullanarak sıralı tutmak için geliştirilmiş algoritmalardan biridir. Yığılmış sıralama, arka planda bir yığın oluşturur ve ağacın tepesindeki sayıyı alarak sıralar.

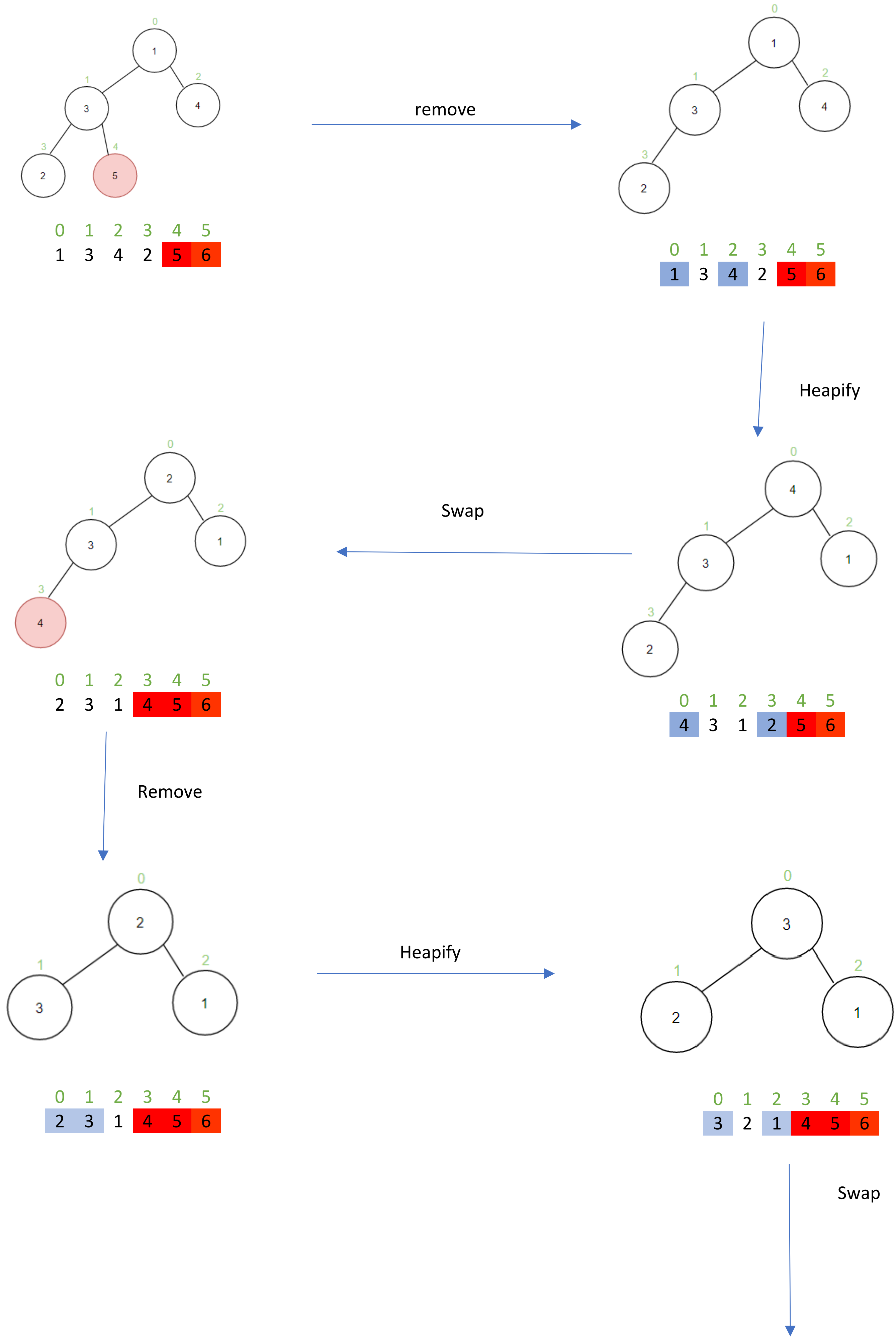
Linux çekirdeği gibi güvenlik ve gömülü sistemler, Heapsort'un çalışma zamanı üst sınırı ve ikincil depolamada sabit bir üst sınır nedeniyle heapsort kullanır.

Tam bir ikili ağaçtan başlayarak, yığının yaprak olmayan tüm öğelerinde heapify adlı bir işlevi çalıştırarak onu Max-Heap olacak şekilde değiştirir. (swap-remove-heapify)

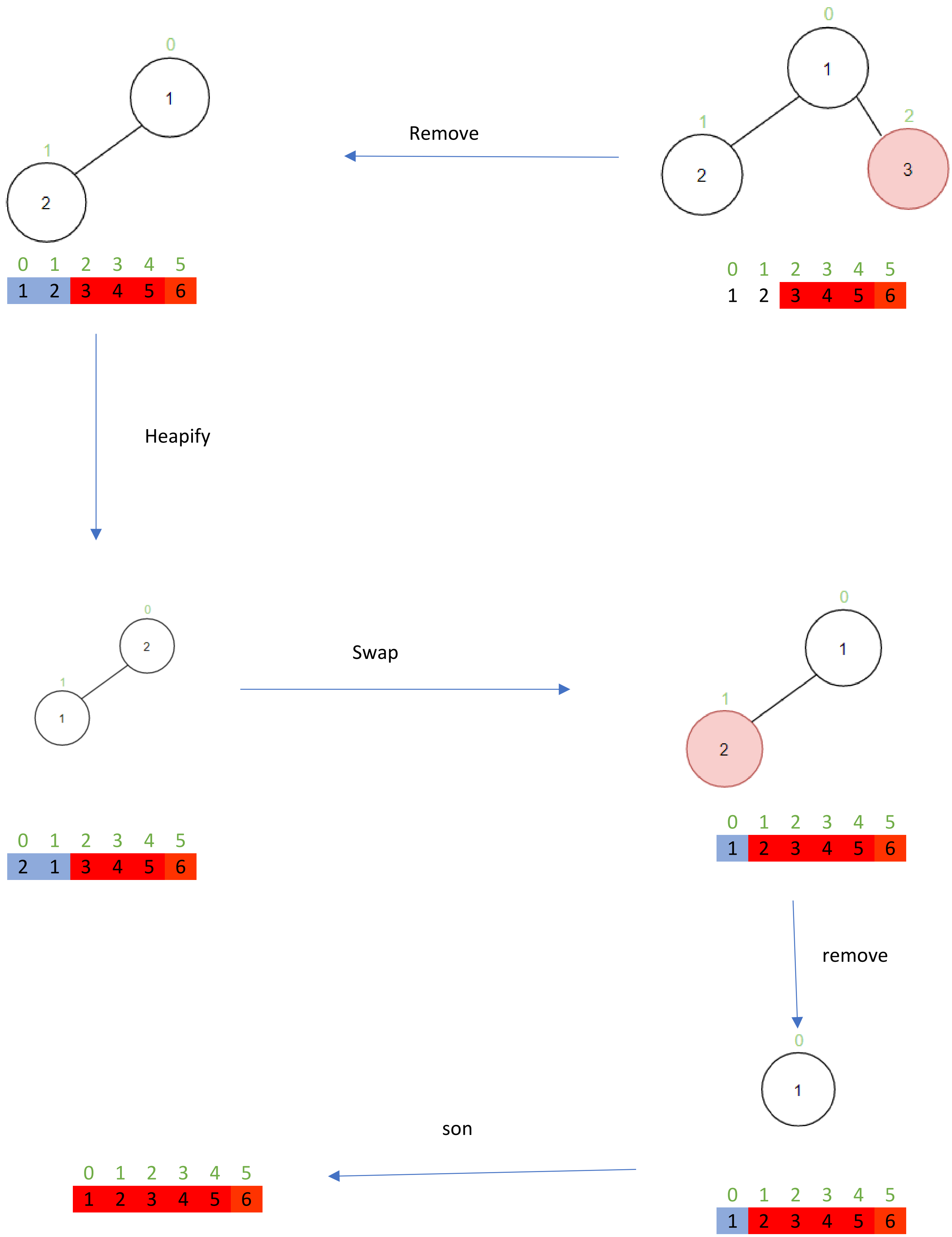
**Örnek Problem:**

Aşağıda sıralanmamış bir dizi verilmiştir. Heap sort algoritması kullanılarak sıralanıyınız.

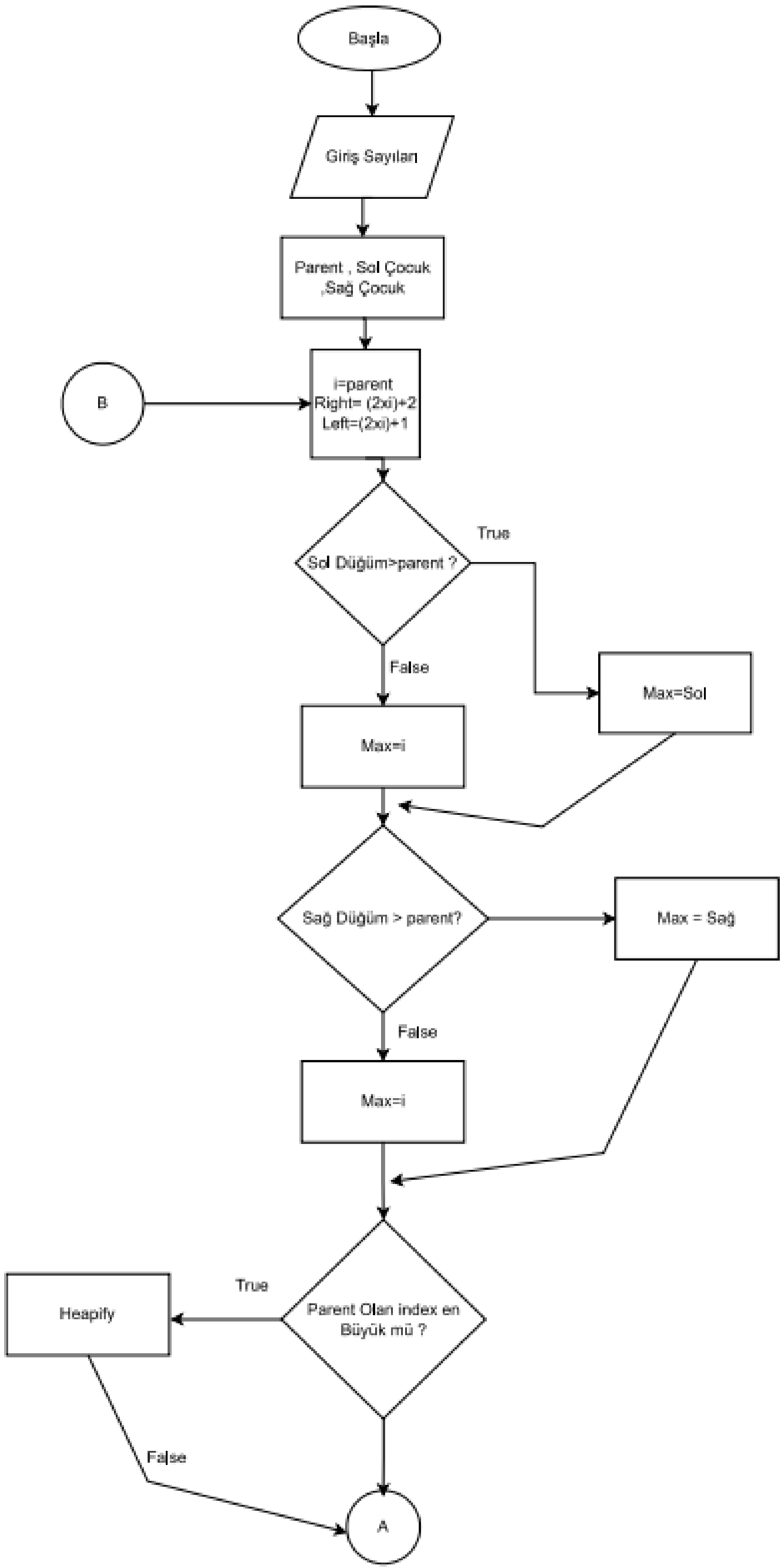




0 1 2 3 4 5
1 3 4 2 5 6
remove
0 1 2 3 4 5
1 3 4 2 5 6
Heapify
0 1 2 3 4 5
4 3 1 2 5 6
Swap
0 1 2 3 4 5
2 3 1 4 5 6
Remove
0 1 2 3 4 5
2 3 1 4 5 6
Heapify
0 1 2 3 4 5
3 2 1 4 5 6
Swap


Remove
Heapify
Swap
remove
son
0 1 2 3 4 5
1 2 3 4 5 6
0 1 2 3 4 5
1 2 3 4 5 6
0 1 2 3 4 5
2 1 3 4 5 6
0 1 2 3 4 5
1 2 3 4 5 6
0 1 2 3 4 5
1 2 3 4 5 6
0 1 2 3 4 5
1 2 3 4 5 6

**Çözümün Akış Diyagramı:**



**Heap Sort C++ Kod:**

```
// Heap Sort
#include <iostream>
using namespace std;

void heapify(int arr[], int n, int i) {
//Kök, sol alt ve sağ alt öğe arasında en büyüğünü bulun
int largest = i;
int left = 2 * i + 1;
int right = 2 * i + 2;

if (left < n && arr[left] > arr[largest])
largest = left;

if (right < n && arr[right] > arr[largest])
largest = right;

//Kök en büyük değilse takas edin ve yığınlamaya devam edin
if (largest != i) {
swap(arr[i], arr[largest]);
heapify(arr, n, largest);
}
}


// yığın sıralama yapmak için fonksiyon
void heapSort(int arr[], int n) {
// Max heap oluştur
for (int i = n / 2 - 1; i >= 0; i--)
heapify(arr, n, i);

// Heap sort
for (int i = n - 1; i >= 0; i--) {
swap(arr[0], arr[i]);

// En yüksek öğeyi kökte tekrar elde etmek için kök öğeyi yığınla
heapify(arr, i, 0);
}
}


//diziyi yazdır
void printArray(int arr[], int n) {
for (int i = 0; i < n; ++i)
cout << arr[i] << " ";
cout << "\n";
}

//Sürücü fonk.
int main() {
int arr[] = {6, 3, 5, 2, 1, 4};
int n = sizeof(arr) / sizeof(arr[0]);
heapSort(arr, n);

cout << "Sorted array is \n";
printArray(arr, n);
}
```

## Heap Sort Algoritma Analizi:

İkili bir ağaç kullandığımız için, yığının alt kısmı maksimum sayıda düğüm içerir. Bir seviye yukarı çıktıkça, düğüm sayısı yarı yarıya azalır. 'N' sayıda düğüm olduğu göz önüne alındığında, en alt seviyeden başlayan düğüm sayısı :

n/2
n/4
n/8
.
.
.
Devam eder

1 = N / 2^x
2^x = N
log(2^x) = log2 N
x * log2(2) = log2 N
x * 1 = log2 N

**Yeni bir düğüm eklemenin karmaşıklığı**
Bu nedenle yığını yaparken yığına yeni bir değer eklediğimizde, atmamız gereken maksimum adım sayısı O(log(n)) olarak çıkıyor.

İkili bir ağaç kullandığımız için, bu yapının maksimum yüksekliğinin her zaman O(log(n)) olur. Yığına yeni bir değer eklediğimizde, max heap özelliğini korumak için onu daha büyük bir değerle değiştiririz. Bu tür takasların sayısı O(log(n)) olacaktır. Bu nedenle, maksimum yığın oluştururken yeni bir değer eklemek O(log(n)) olur.

**Yığından maksimum değerli düğümü çıkarmanın karmaşıklığı**
Benzer şekilde, listenin sonuna eklemek için maksimum değerli düğümü yığından çıkardığımızda, gereken maksimum adım sayısı da O(log(n)) olacaktır. Maksimum değerli düğümü, en alt düzeye inene kadar değiştirdiğimiz için, atmamız gereken maksimum adım sayısı, yeni bir düğüm eklerken olduğu gibi, O(log(n)) ile aynıdır.

Bu nedenle, fonksiyonun toplam zaman karmaşıklığı heapify O(log(n)) olur.(void heapify)

**Bir yığın oluşturmanın karmaşıklığı:**

(n/2 * 0) + (n/4 * 1) + (n/8 * 2) + (n/16 * 3) + ... =n/2
Bu yüzden bunun karmaşıklığı O(n) olur.

**Toplam zaman karmaşıklığı:**

Son işlevinde, bir yığın oluşturmak için bir kez çalışan ve çalışma süresi O(n) olan işlevini heapsort kullanırız . //max heap oluştur, Sonra bir for-loop kullanarak, heapify yığına bir düğüm eklediğimizde veya çıkardığımızda max-heap özelliğini korumak için for her düğümü çağırırız. 'n' sayıda düğüm olduğundan, algoritmanın toplam çalışma süresi O(n(log(n)) olur ve heapify işlevi her düğüm için kullanırız.

Matematiksel olarak:

- Bir düğümün ilk kaldırılması log(n) zaman alır
- İkinci kaldırma log(n-1) zaman alır
- Üçüncü kaldırma log(n-2) zamanını alır
- ve böylece log(1) zamanını alacak olan son düğüme kadar

log(n) + log(n-1) + log(n-2) + ....log(1) log(x) + log(y)

= log(x * y olarak)

=log( n∗(n−1)∗(n−2)∗…∗2∗1)

=log(n!)

Daha fazla sadeleştirme ile,(Stirling Yöntemi ile Faktöryel değerlerin tahmini)

log(n!)

=n∗log(n ) olur. )−n+O(log(n))

En yüksek sıralı terim dikkate alındığında, toplam çalışma süresi O(n(log(n)) olur.

**Yığın Sıralamanın En Kötü Durum Zaman Karmaşıklığı**

Yığın sıralama için en kötü durum, listedeki tüm öğeler farklı olduğunda ortaya çıkabilir. Bu nedenle, max-heapifybir öğeyi her kaldırdığımızda aramamız gerekir. Böyle bir durumda, 'n' sayıda düğüm olduğu düşünülürse

- Her öğeyi kaldırmak için takas sayısı log(n) olacaktır, çünkü bu, yığının maksimum yüksekliğidir.
- Bunu her düğüm için yaptığımızı düşünürsek, toplam hamle sayısı n * (log(n)) olur.

Bu nedenle, en kötü durumda çalışma zamanı O(n(log(n)) olacaktır.

**Yığın Sıralamanın En İyi Vaka Zaman Karmaşıklığı**

Yığın sıralama için en iyi durum, sıralanacak listedeki tüm öğeler aynı olduğunda ortaya çıkar. Böyle bir durumda, 'n' düğüm sayısı için:

- Her bir düğümü öbekten kaldırmak, yalnızca sabit bir çalışma zamanı (O(1)) alacaktır. Tüm öğeler aynı olduğu için herhangi bir düğümü indirmeye veya maksimum değerli düğümü yukarı getirmeye gerek kalmayacaktır.
- Bunu her düğümde yaptığımız için toplam hamle sayısı n * O(1) olacaktır.

Bu nedenle, en iyi durumda çalışma zamanı O(n) olacaktır.

**Yığın Sıralamanın Uzay Karmaşıklığı**

Yığın sıralama yerinde tasarlanmış bir sıralama algoritması olduğundan, alan gereksinimi sabittir ve bu nedenle O(1). Bunun nedeni, herhangi bir giriş durumunda

- Bir yığın yapısı kullanarak tüm liste öğelerini yerinde düzenliyoruz
- Max düğümünü max-heap'ten çıkardıktan sonra kaldırılan öğeyi aynı listenin sonuna koyuyoruz.

Bu nedenle, bu algoritmayı uygularken fazladan boşluk kullanmıyoruz. Bu, algoritmaya O(1) uzay karmaşıklığını verir.

Özet olarak, heapsort şunları içerir:

- Böl ve fethet yöntemini kullanır
- En iyi durum zaman karmaşıklığı O(n) [tüm öğeler sıralanmışsa]
- O(n(log(n))'nin en kötü durum zaman karmaşıklığı [listedeki öğeler sıralanmamışsa]
- O(1) uzay karmaşıklığı