VERİ YAPILARI

VE

ALGORİTMALAR

İçindekiler

1. Temel Kavramlar

Algoritma: Bir problemin çözümü için gerekli olan adımların bütününe verilen isim.

Binary: İkili sayı sistemi. (0 veya 1)

Bytecode: Java derleyicisinin java ile yazılmış kodların makine dili yerine kendine has oluşturduğu binary dosyası.

JVM: Java bytecode formatında derlenmiş programların çalışmasını sağlayan bir sistemdir.

Binary to decimal: 2 üssü.

Decimal to binary: 2'ye böl. Tersten yaz.

Sayısal olmayan verilen tutulması: Bilgisayardaki her şey 1 ve 0'dan oluşur. Sayısal olmayan bir ifadenin sembol ile gösterimi yapılabilir.

ÖRNEK: 100101010. Bu, bir sembol olabilir ve karşılık geldiği değer "V" olabilir.

Bit: Verilerin en küçük yapı taşıdır.

1 bit'ten 2 adet (2^1) sembol,

2 bit'ten 4 adet (2^2) sembol,

1 bit'ten 8 adet (2^3) sembol oluşturulabilir. Alan sayılarıdır. 8 adet sembol gibi.

8 bit = 1 byte

Recursion: Bir problemin alt problemlere bölünüp hesaplanmasına, nerede son bulacağını belirttiğimiz ifadelere "recursion (özyineleme)" denir.

Recursive Fonksiyon: Bir fonksiyon kendi içerisinde kendisini, problemin daha küçüğünde çağırıyorsa ona "recursive fonksiyon" denir.

ÖRNEK: $5! = 5 \times 4! = 5 \times 4 \times 3!$

ÖRNEK: Fibonacci dizisi. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

2. Veri Yapıları

Arrays (Diziler): Anlam ifade etmesi için birden fazla nesneye ihtiyaç duyabilir.

ÖRNEK:

4	2	1	5	6	9

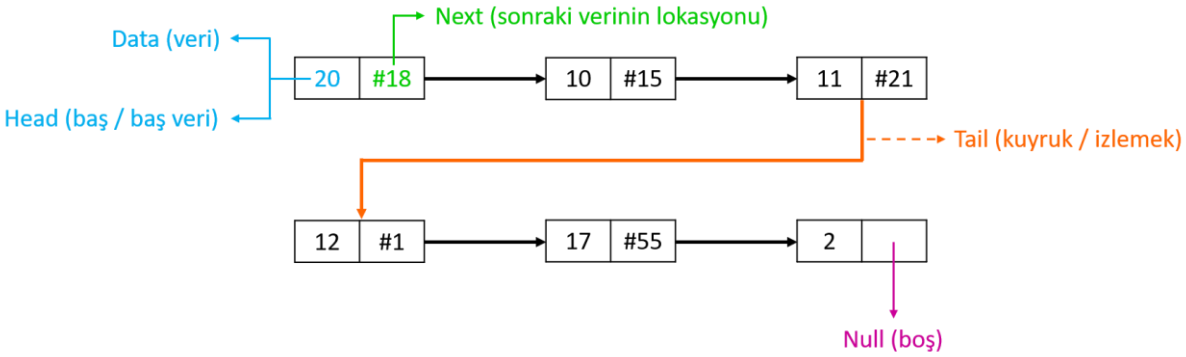
Array dezavantaj: Hafıza problemi. Verileri bir yerden başka bir yere taşırken zaman ve güç kaybederiz.

Array avantaj: Array'lerin birbirine bağlı olması ulaşabilirliği kolaylaştırır. 5.indexteki array'ede, 10000.indexteki array'ede aynı anda ulaşılabilir.

Dynamic Arrays (Dinamik Diziler): Yeni bir eleman için boşta yer tutmasından ötürü esnektir. Yeni elemanlar için yer tutulur.

Dynamic arrays dezavantaj: Hafızada fazladan yer kaplar. Gerçekleşecek olan bir diğer olayı engelleyebilir.

Linked – List (Bağlı Listeler): Yan yana zorunluluğu olmadan veri tutmamızı sağlayan yapılardır. Yeni gelen eleman için hafızada yeni bir alan açmamız gerekmez. Elemanlar hafıza içerisinde dağınıktır fakat son gelen eleman kendinden bir önceki elemana adresini bildirmek zorundadır. Her bir düğüm bir sonrakinin adresini tutar, her bir önceki eleman bir sonraki eleman ile bağlıdır.

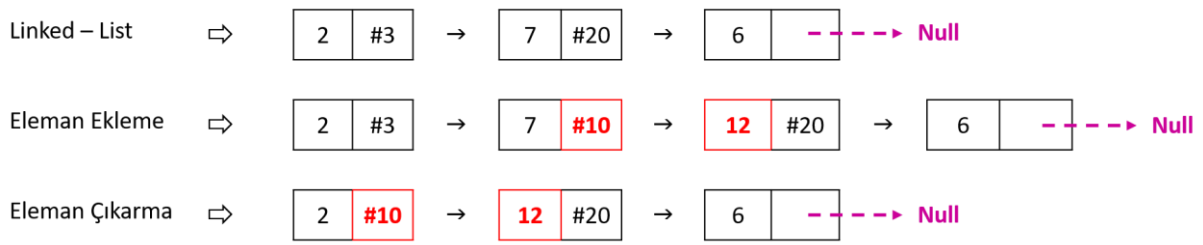


ARRAY	LINKED – LIST
☾ Array'in istediğimiz elemanına sabit sürede erişebiliyoruz. ☾ Sadece elemanı tuttuğumuz için daha az yer kaplıyor. ☾ Memory locality için iyi.	☾ Eleman eklemek ve silmek arraylere göre daha kolay. ☾ Elemanların bir blok olarak tutulması gerekmiyor, hafızada blok olarak yer yoksa da kullanılabilir.

Random Access (Rastgele Erişim): Arraylerde istediğimiz elemana sabit sürede erişebilme durumudur.

Pointer (Gösterici): Bir hafıza alanını işaret etmeye yarayan değişken tipidir. Bir değişkenin adresini taşıyan başka bir değişkendir. Bu adres "a" değişkeninin bellekteki (memory) yerinin adresidir.

ÖRNEK: Null'ı gösteren referans (pointer / gösterici).

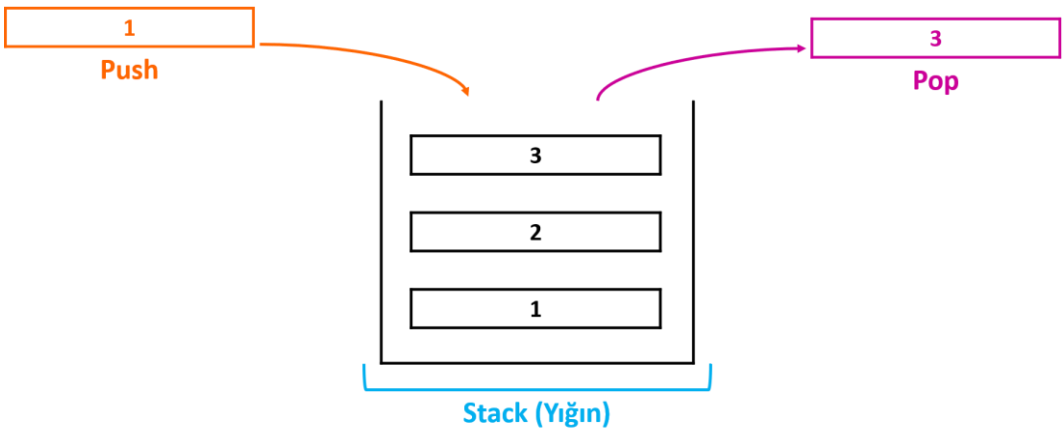


Stack (Yığın / Stack Veri Yapısı): LIFO (Last In First Out / En son giren en önce çıkar) mantığına dayanan, elemanlar topluluğundan oluşan bir yapıdır.

Push: Stack'in üzerine eleman eklemek için kullanılır.

Pop: Stack'ten eleman çıkarmak için kullanılır.

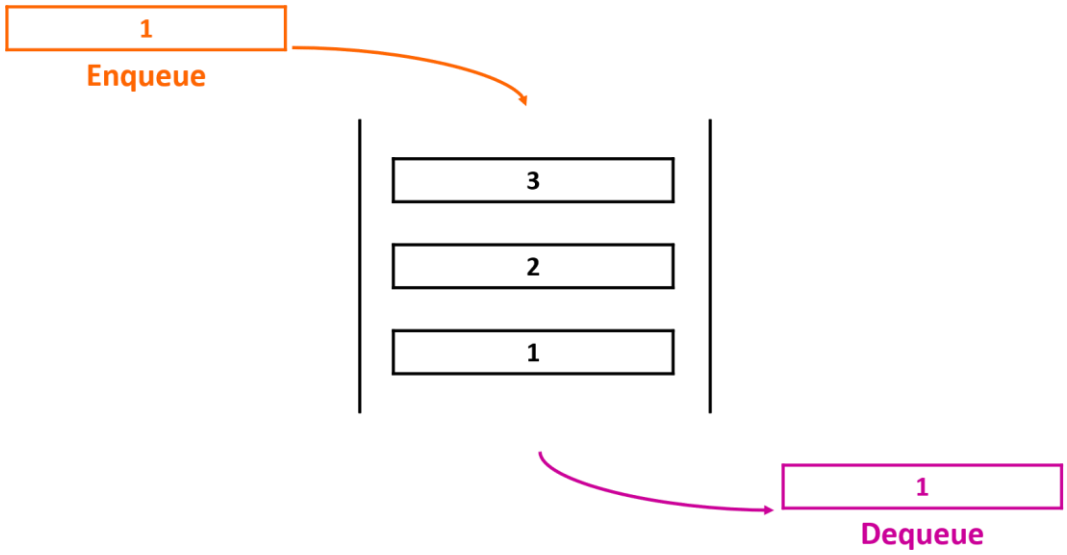
STACK – NOT: Linked – List gibi aradan eleman çıkarma veya araya eleman ekleme yoktur. Ya en üste eleman eklenir ya da en üstten eleman çıkartılır.



Queue (Kuyruk / Queue Veri Yapısı): FIFO (First In First Out / İlk giren ilk çıkar) prensibine dayanan, girişlerde ve çıkışlarda belirli bir kurala göre çalışan yapıdır.

Enqueue: Yeni elemanın queue'ya eklenmesi (yeni birinin sıraya girmesi).

Dequeue: Elemanın queue'dan çıkarılması (sırası gelenin sıradan çıkması).



QUEUE – NOT: Stack'teki gibi Queue'da da ortaya eleman ekleme veya çıkarma yoktur.

Indexleme (Gösterge / İşaret): Arraylerde 0 (sıfır) bazlı indexleme vardır. Bazı programlama dillerinde 1 bazlı olsa da genellikle 0 bazlı indexleme kullanılır.

NOT: Index'i lokasyon gibi düşün.

0 bazlı indexleme :	A =	1	2	3	A [0] = 1	A [1] = 2	A [2] = 3
1 bazlı indexleme :	B =	4	5	6	B [1] = 4	B [2] = 5	B [3] = 6

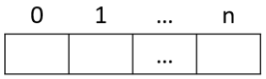
Hash Table: Key ve value prensibine dayanan bir array kümesidir. Key olarak çağırdığımız elemanın değerini (value) yansıtır. Hash Table yerine dizileri kullanabilirdik. Key'in valuesini tek tek aramak istemediğimiz için hash table kullanıyoruz.

ÖRNEK:

- Hash Table, ürünlerin fiyatını ezbere bilen çalışan gibidir.
- Ürünlerin isimlerini ve fiyatlarını array'in elemanları olarak tutup bu sorunu çözebilirdik ama ürünleri array'de tek tek aramak istemiyorum, anında aradığım ürünün fiyatını bulmak istiyorum.

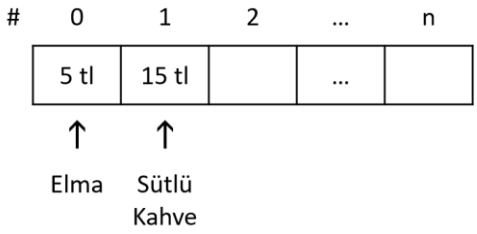
BUNU NASIL ÇÖZEBİLİRİZ? (" HASH FUNCTION ")

- İlk olarak; eleman sayısı ürün sayısına eşit bir array oluşturacağız:



- Ürünlerin isimlerini bir fonksiyona sokup çıktılar alacağız:

- Fonksiyonun çıktılarını oluşturduğumuz aray'in index'i olarak kullanıp, ürünlerin fiyatlarını bu indexlerde tutacağız:



- Array'ler bize kaçıncı eleman olursa olsun sabit sürede istenen lokasyondaki elemanı verebiliyordu.
- Bu sabit sürede erişmeyi lokasyon bazlı değil, tanım bazlı kullanmak, istiyoruz. Bana 3.elemanı getir değil, bana elmaya karşılık gelen elemanı (fiyatı) getir demek istiyoruz.

☾ Biri bize bir ürünün fiyatını sorduğunda bu ürünü oluşturduğumuz fonksiyona besleyip arraydeki index'i neredeymiş onu bulacağız.

☾ Bu fonksiyona **HASH FUNCTION**,
Hash function + Array yapısına **HASH TABLE** denir.

HASH FUNCTION (KARMA FONKSİYON) ŞARTLARI

1) Aynı key'e aynı value verilmeli

2) Farklı key'lere farklı value'ler olmalı

3) Array'in boyutu value'lerin sınırlarında (range) olmalı

✦ Maalesef her seferinde aynı girdiye (keys) farklı sonuç (value) veremiyor Hash function, bu duruma **" COLLISION "** deniyor.

Collision (Çarpışma / Hash Collision): Hash function, farklı iki değerden aynı sayı üretilirse bu duruma "collision (çarpışma)" denir. Bu olay istediğimiz bir durum değildir.

☾ Hash function'lar bazen farklı durumlar için farklı sonuçlar üretemeyebilir. Örnek: araçları bir hash function'dan geçirelim. Bu fonksiyonumuz son harflerine göre göre bir değer atıyor. Örneğin, motor ve tır için aynı değerleri ataması collision'a neden oluyor.

☾ Collision sorunuyla az karşılaşabilmek için kaliteli bir hash function olmalı. Bu sayede verimli bir Hash Table elde etmiş oluyoruz.

☾ Collision sayısı arttıkça aradığımız şeyi bulma hızı azalır.

3. Algoritma Analizi

Algoritma analizi: Var olan kaynaklara göre en uygun algoritmayı seçmek için uygulanır.

ALGORİTMA ANALİZİ EN İYİ NASIL YAPILIR?

- **Execution Time (Çalışma Süresi):** Çalışma süresi, programlama dilinden ve kullanılan bilgisayarın özelliklerinden etkilenir, o yüzden genelleme yapılamaz bu konuda.
- **İfade sayısı:** Programda kaç tane ifadenin çalıştığıdır. Programlama diline göre aynı işlem için çalışan ifade sayısı değişebilir. Bu yüzden genellenebilir değildir.
- **Rate of Growth (Büyüme Hızı):** Programa verdiğimiz input (girdi) boyutu ile çalışma zamanını fonksiyonel olarak birbirine bağlarsak bilgisayarlara ve programlama dillerine bağlı olmayan bir yapı oluşturmuş oluruz.

Donanım ve diller ile algoritma analizi (ilk 2 madde) pek sağlıklı değildir.

Algoritma analizi, bir algoritmanın çalışabilmesi için gerekli koşulların sağlanıp sağlanmadığını gösteren bir parametredir.

RAM Modeli

- Genellenebilir bir analiz yapmak için, her algoritmayı ayı bilgisayar ile test ediyor gibi yapacağız.
- Bu hayali makineye **RAM (Random Access Machine)** diyeceğiz. Bizim bilgisayarımızdaki RAM ile karıştırma, bu model sadece.

VARSAYDIĞIMIZ RAM'İN ÖZELLİKLERİ

1. Her basit işlem (+, -, and, or gibi) 1 birim zaman alır.
2. Döngüler 1 birim zaman değil, içerisinde;
   *kaç defa işlem oluyorsa iterasyon (tekrarlama) sayısı * işlem sayısı* kadar birim zaman alır.
3. Hafızadan her okuma işlemi 1 birim zaman alır.

TIME COMPLEXITY

- Kullanacağımız algoritmayı analiz etmek istiyoruz. Problem aynı olsa bile farklı inputlar için algoritmamız farklı performans seneryoları üretebilmektedir. Input, algoritmanın yapacağı işlem sayısını etkileyebilir.
- Bu yüzden algoritma analizimizi 3 ana başlıkta yapabiliriz:

TIME – COMPLEXITY CASE'LERİ

1. **Worst Case:** Vereceğimiz inputun algoritmamızı en yavaş (en fala işlem yapacak) şekilde çalıştırdığı durumdur. Her algoritmaya göre worst case farklıdır. Örnek: sözlükte "z" harfi ile bir kelime arıyoruz. Sondan başlarsak sözlüğe bakmaya bu, worst case olmaz. Baştan bakmaya başlarsak worst case olur. Burada algoritma, sayfalara tek tek bakmaktır. Yapılan işlem, sayfa çevirmedir.

2. **Average Case:** Genel olarak beklediğim durum.

3. **Best Case:** Vereceğimiz inputun algoritmamızı en hızlı şekilde çalıştırdığı durumdur.

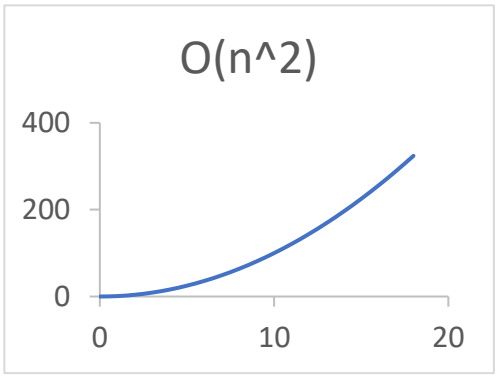
☾ Algoritmamızın çalışmasını en iyi yansıtan average case, ama bu durumu analiz etmek diğerlerine göre çok daha zor. Inputların geldiği dağılımı bilip, ona göre analiz etmek gerekiyor.

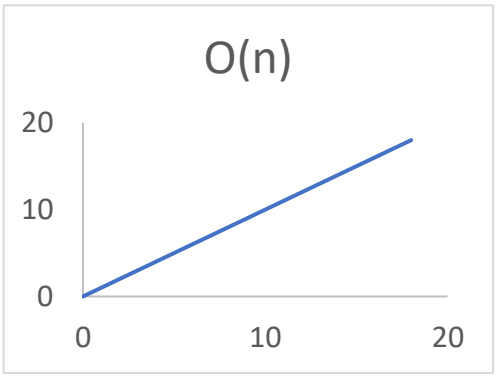
☾ Wost case'e göre analiz yaparsak performansımız için üst sınır çizmiş oluruz. Böylece worst case için bizi tatmin eden bir algoritmamız varsa, average case zaten bundan bu daha iyi (veya aynı) performans vereceği için o da bizi tatmin edecektir.

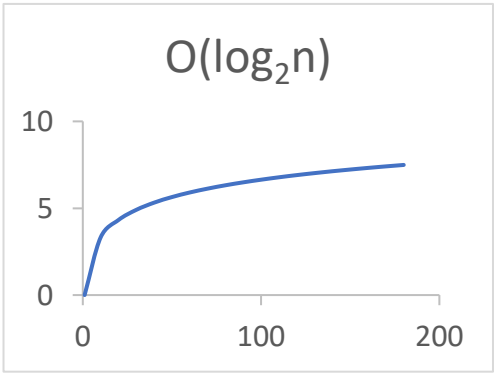
Big – O Notation

- ☾ İki farklı arama yöntemimiz var.
- ☾ Bunlardan A algoritması tek tek sayfalara bakıyor.
- ☾ B algoritması, sözlüğün alfabetik sıralanmış olduğundan yarralanarak en başta en ortadaki sayfayı açıyor. Eğer bu sayfadaki harfler aradığım kelimeden alfabetik olarak daha ilerideyse sol tarafa aynısını yoksa sağ tarafa aynısını yapıyor. Böylece problem her seferinde yarı boyutuna inmiş oluyor.
- ☾ Birkaç durum üzerinden konuşalım. Diyelim ki 1000 sayfalık bir sözlüğüm var.
  - A algoritmasını en kötü durumda (aradığım en son sayfadaysa) kaç işlem yapacak? 1000 işlem.
  - B algoritması ise en kötü durumda kaç işlem yapar? $2^n = 1000$ işlem. ($n \cong 10$)
- ☾ Sizce hangisi daha hızlı çalışır? Tabii ki B algoritması. Peki neden? Sürekli tarayacağı alan azalıyor. A algoritması daha işlemini bile yarılamamışken, B algoritması sonuca ulaşıyor.
- ☾ Bu örneğe bakarak B algoritmasının A algoritmasından daha hızlı olduğunu görebiliyoruz. 1000 / 10'dan 100 katı hızında diyebilir miyiz? Bu genellenebilir bir şey midir? Hayır.

- Diyemeyiz. Şöyle düşünelim, sözlüğüm 10.000 elemanlı olsa, A algoritması en kötü durumda 10.000, B algoritması n=13 (2^n=10.000) işlem yapar. 10.000 / 13 yaklaşıl 770 katı hızında gözüküyor.
- Bu yüzden algoritmaların sadece ,nput boyutuna göre karşılaştırmaların bakıp karar veremeyiz. Genel yapısını bize gösterecek bir analize ihtiyacımız var, işte burada Big O Notation devreye giriyor.
- Big O Notation, algoritmanın ne kadar sürede çalışacağını söylemeyecek. Bize algoritmamızın çalışma zamanının input boyutu ile nasıl değişeceğini söyleyecek.
- Mesela sözlük örneğimizde input size'ımıza n dersek, algoritmamızın en kötü durumda n işlem yaptığını söyleyebiliriz. Inputum n boyutunda olunca çalışma süremin de en kötü durumda n olmasını O(n) diye göstereceğim. Aynı şekilde B algoritması içinde $O(\log_2 n)$ olur.
- Big O Notation'da yapılacak toplam işlem sayısının (n) input size ile nasıl scale (ölçek) olacağına bakıyoruz. Benim için önemli olan fonksiyonun yapısı. (n, n^2, $1/n^2$, $\log_2 n$, …)
- İşlem sayısı (n) input size ile linear mi artıyor, karesi ile mi orantılı artıyor, logaritmik mi?
- Karakteristiği önemsediğimiz için 2n işlem yapan algoritmaya da n işlem yapan algoritmaya da O(n) diyoruz, ikisi de linear bir şekilde artıyor.
- Big O Notation bakarken katsayılar önemli değil.
- Analizimin sonucu $2n^2+3n+2$ gibi bir şey çıktı diyelim. n büyüdükçe, 3n+2 'nin etkisi $2n^2$ 'nin yanında önemsiz kalacak. O yüzden dominant olanı Big O Notation olarak yazabiliriz, $O(n^2)$.
- Big – O Notation Grafikeri:





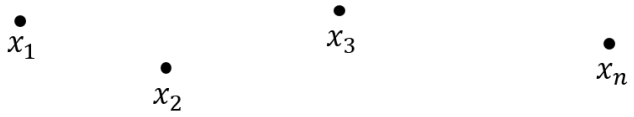


4. Sorting (Sırlama) Algoritmaları

Sorting (Sıralama Algoritmaları): Elimizdeki eleman dizisini belirli bir sıralama ölçütüne göre sıralamaya denmektedir. Sorting, kendinden sıralama algoritmaları olarak bahsetmektedir. Sorting, bir eleman dizisini, belirli sıralama kurallarına göre sıralama yapar.

Kullanımına göre Sorting'ler

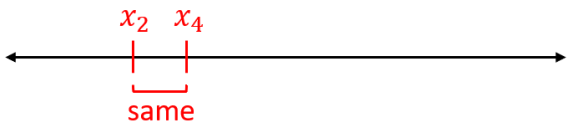
- **Searching** yöntemini kullanarak elemanlarımızı sıraladık. Bunun sebebi, eleman ararken işimizin kolaylaşmasını istiyoruz.
- **Closest Pair** yöntemini kullanarak birbirine yakın sayıları gruplandırdık ki arama yaparken zamanımızı efektif bir şekilde kullanalım. Mesela rastgele noktalar olduğunu düşünelim n tane. N tane noktanın içerisinden birbirine en yakın olan iki noktayı bulmak istiyoruz. Closest pair ile bunları sıralayabilir ve iki nokta bulunabilir.
- **Aynı eleman kontrolü:** birbiriyle aynı olan sayıları örüntü içerisinde kaç tane aynı eleman varsa sayısını öğrenebilirim.
- **Mode bulma:** eleman dizisini search ettikten sonra elemanların yan yana olanları sayarsam daha hızlı mode bulabilirim.

Noktalar ⇨ 

Seraching ⇨ $x_1 , x_2 , x_3 , \dots , x_n$

Closest pair ⇨ $\boxed{x_1 , x_5}$, $x_6 , \dots , x_n$

closest pair

Control same element ⇨ 

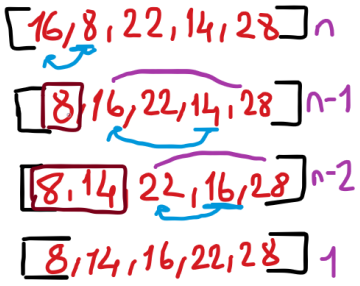
☾ **Selection (Insertion) Sort:** En basit sorting algoritmalarından biridir. Verilen örüntüye ait en küçük elemanı buluyor ve en baştaki sayı ile yer değiştiriyor. Peki ya devamı? İkinci en küçük elemanı buluyor ve 2. sıra ile değiştiriyor. Baktın ki 2.sıradaki eleman en küçük hiç dokunma! Hemen 3. sıraya geç. 4, 5 derken dizi bitti. İşte insertion sort'un temel çalışma prensibini öğrendin.

$$Yapılan\ işlem\ sayısı = n + (n-1) + (n-2) + \cdots + 1$$

$$= \frac{n \times (n+1)}{2} = \frac{n^2 + n}{2}$$

Insertion Sort'un Big – O Notation'ı = $O(n^2)$ olmaktadır.

n : eleman sayısı



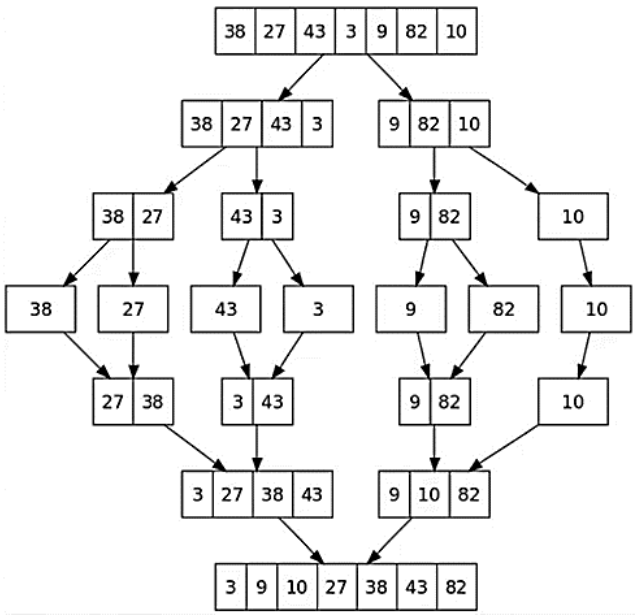
☾ **Merge Sort:** Insertion Sort'da, Big-O gösteriminden dolayı input'um arttığında n^2 olduğunda dolayı çalışma zamanı artıyor. Peki daha hızlı bir şekilde sıralama yapılabilir mi? Evet, Merge Sort burada yardımımıza koşuyor. Bir listeyi her adımda parçaya ayırıp tek eleman kalıncaya kadar bölüyor. Böldükten sonra sıralı bir şekilde bize sunuyor (Performans). Insertion sort'da, time complexity n^2 olduğundan ötürü çalışma zamanımız artıyordu. Merge sort'da ise $n\log_2 n$ olduğu için açık ara performans olarak daha iyi diyebiliriz.

$$Yapılan\ işlem\ sayısı > 2^x = n \qquad x = \log n$$

n : eleman sayısı

Time Complexity = $n.logn$ olur.

Merge Sort'un Big – O Notation'ı = $O(n \log n)$ olmaktadır.



☾ **Quick Sort:** Hızlı sıralama günümüzde çok yaygın olarak kullanılan bir sıralama algoritmasıdır. N tane sayıyı average case'e göre big-o nlogn, worst case'e göre Big–O nation'ı n^2 karmaşıklığı ile sıralanır. İlk olarak bir pivot belirler bu pivota göre pivottan küçük ve eşitler sol kısmına, pivottan büyük ve eşitler sağ kısmına yazılır. Parçalanmış kısımlar yeni bir pivot belirlenerek parça pinçik edilir.

$Yapılan\ işlem\ sayısı > 2^x = n \qquad x = \log_2 n$ n : eleman sayısı

$Worst\ Case > O(n^2)$

Worst Case'in Time Complexity > $n.O(n) = O(n^2)$ *olur. (toplam sorgulama sayısı)*

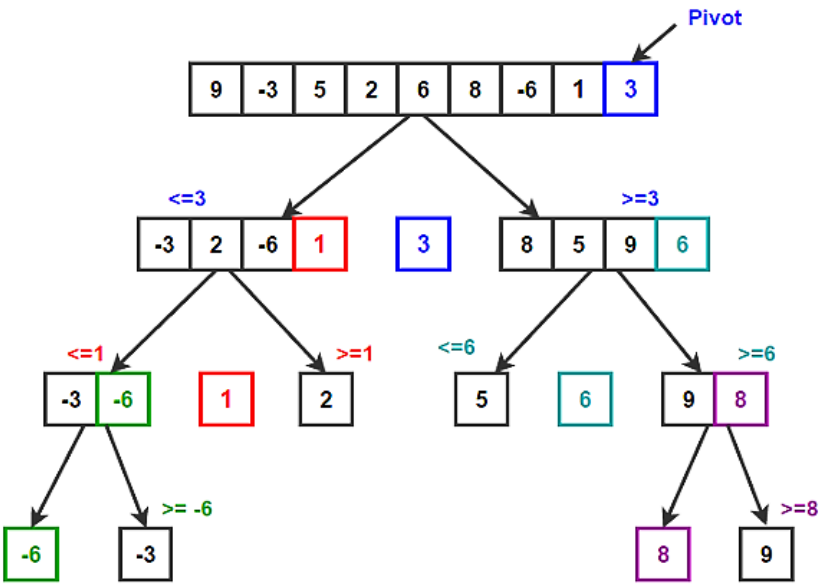
n elemanlı olsun, 1.pivot seçildiğinde worst case'i hesapladığımız için solda veya sağda 1 eleman diğer tarafta n-1 eleman kalır. n-1 elemanda da kötü bir 2.pivot seçilir ve bir tarafta 1 eleman diğer tarafta n-2 eleman kalır. Bu durum her iki tarafında 1 olduğu durumuna kadar devam eder. Her satırda seçilen pivot, en kötü durumda seçilir yani ya en büyük ya da en küçük değer olarak seçilir. Böylelikle n tane O(n) işlem yapılmış olunur toplamda. Bu da $O(n^2)$ olarak worst case'imizi verir.

$Avarage\ Case > O(nlogn)$

1 satırdaki Time Complexity = $O(n)$ *olur. (1 satırdaki sorgulama sayısı)*

Toplam Time Complexity = $O(nlogn)$ *olur. (toplam sorgulama sayısı / avarage case)*

Katsayısı merge sort'tan az olduğu için daha kısa olabilmekte.



5. Searching (Arama) Algoritmaları

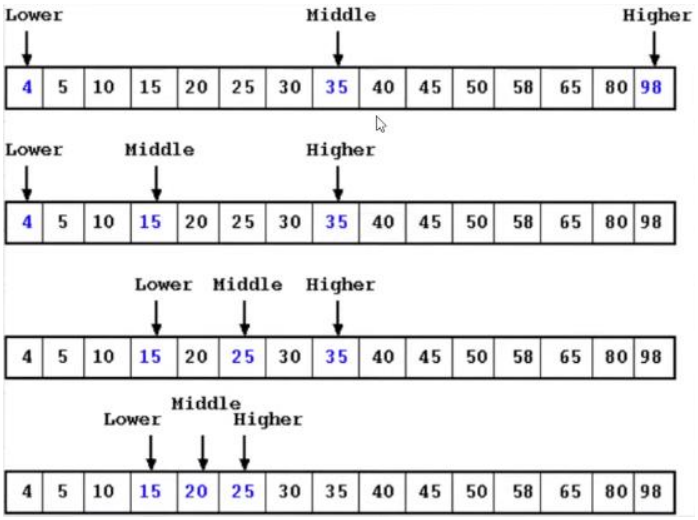
Günümüzde veriler gitgide artan bir hal alıyor. Her insanın bir bilgisayarı ve telefonu olduğunu düşünürsek, terabaytlarca veri ediyor. Arama algoritmaları ise istediğim özellikteki verinin elimdeki veri setlerinde aranıp, bulunup getirilmesi demek. Bunun hızlı olmasına önem gösterilir.

☾ **Linear Search:** Tek tek elemanları dolandıktan sonra istediğim elemanın olup olmadığına bakmaktır.

- Ekstra hafıza kullanımı yoktur.
- Örneğin, [20,25,46,48] veri setini ele alalım. Benim aradığım eleman 25. İlk elemana gidiyorum ve değeri 20 sen değilsin diyorum. İkinci elemana gidiyorum ve değeri 25 evet sensin diyorum. Linear search algoritmam burada bitmiş oluyor.
- Worst case için;
  - ✓ Worst case durumu istediğimiz eleman dizinin sonundaysa gerçekleşir.
  - ✓ Time complexity'nin Big – O durumu > O(n) (n: eleman sayısı)
  - ✓ Worst case Big – O durumu > O(n)

☾ **Binary Search:** İkili arama algoritması, elimizde bulunan veri dizisini sıralı olduğunu varsayıyor, bu durumu değiştirerek sonuca varmak istiyor.

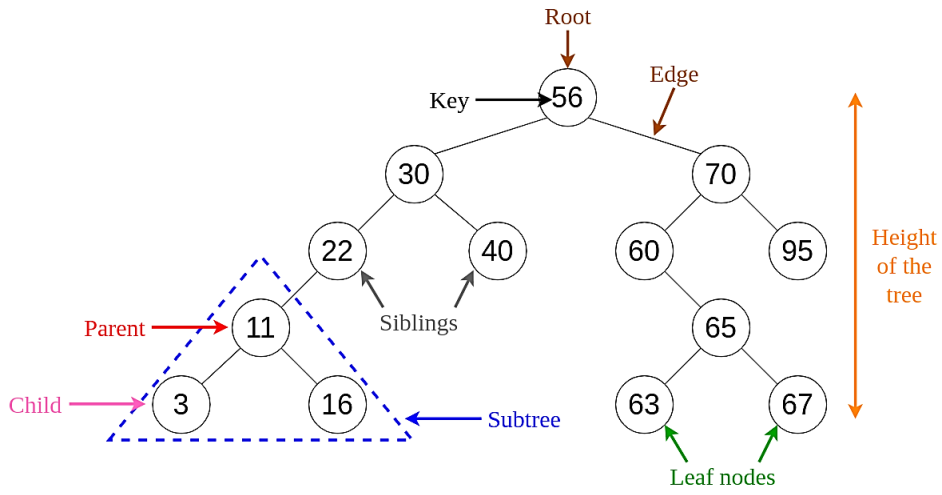
- Avarage case için;
  - ✓ İkili arama algoritması, diziyi her seferinde ikiye bölerek ikili arama yapar.
  - ✓ Sıralı bir listem var ise benim Big-O notation = O(logn) olarak karşımıza çıkıyor.
  - ✓ Aşağıdaki örnekte 20 sayısını dizide aradığımızı varsayarak işlemlere bakalım.



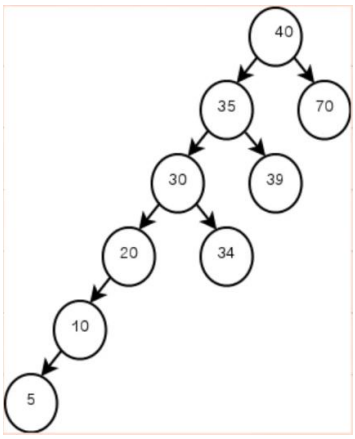
- Aradığım sayı 15 ve benim değer kümem [10,15,20,16,22,36,23] diyelim. Binary Search bu diziyi manipüle ederek şu ifadeye dönüştürüyor. [10,15,16,20,22,23,36]. 36 sayısını en yüksek sayı, 10 sayısını en düşük sayı ilan ediyor. Benim aradığım sayı ile ortada kalan sayıyı kıyaslıyor eğer benim sayım büyükse kendinden küçük bütün sayıları siliyor. Ve kendine yeni bir ortanca belirliyor. Böylelikle gereksiz arama yapmaktan kurtarıyor.

☾ **Binary Search Tree:** Bir düğüm her iki tarafa da referans verebiliyor. Sağ ve sol olarak. Sağ tarafından kendinden büyük elemanlar, sol tarafında ise kendinden küçük elemanlar bulunacak.

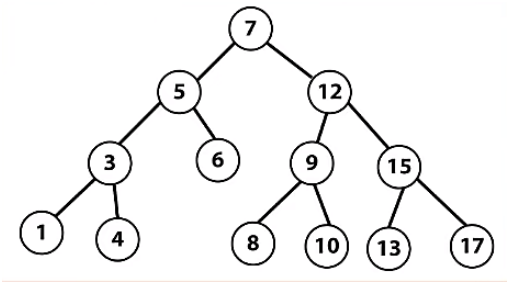
- Tree'ye eleman eklemek istediğimde root'dan başlıyorum. Örnek olarak ben 26 sayısını ağaç yapısına eklemek istiyorum. Root'a soruyorum senin değerin ne? 56. Baştaki açıklamamızı hatırlayalım. Sağ tarafında kendinden büyük, sol tarafında kendinden küçük elemanlar var. O yüzden sırasıyla 56 ve 30'a kadar ilerliyorum. 30 bana benim sol tarafıma geçmelisin çünkü sen benden küçüksün diyor. Karşıma 22 değerinde olan düğüm çıkıyor ve 22'den büyük olduğum için sağ tarafına bir köşe çekiyorum ve 26 sayısını bağlıyorum.



- Her bir sayının bulunduğu yere node (düğüm) denir.
- Worst case için;
  - ✓ Yanda yer alan binary search tree'ye 4 sayısını yerleştirmek istersek neredeyse bütün eleman sayısı (n) kadar sorgulama yapılmış olunur.
  - ✓ Worst case'in big-o > O(n)
  - ✓ Bu durumda bir eleman eklemek veya aramak n zamanda gerçekleşir.



- Average case için;
  - ✓ Tree'nin dengeli bir şekilde olma durumudur.
  - ✓ Average case Big – O durumu > $2^x = n \quad x = log\, n \quad O(\log n)$
  - ✓ Sorgulama sayısı az olmaktadır.
  - ✓ Örnek olarak yandaki binary search tree'ye 18 sayısını eklemek istersek sırasıyla 7, 12, 15 ve 17 sayısına sorgulama (toplam 4 sorgulama) yapmış olacağız.
  - ✓ Bu durumda bir eleman eklemek veya aramak logn zamanda gerçekleşir.



- Array'lerin aksine random access yapamaz. (5. Elemanı getir vs.)