

Data Structures

Arrays

Birden çok bilgiyi bir arada tutmak isteyebiliriz. Bu mantık bir arada olunca arrays ve linked list ile olur. Arraylarda bu kutucuklar yanyana olur. Veri yapısı kaç byte dan oluşuyorsa o kadar kutucuk yanyana tutulur. En başta oluşturduğumuz array da yeni bir eleman eklendiğinde baştan eleman sayısı kadar kutucuk oluşur ve oraya eklenirler. Bu hafızada yer israfına sebebiyet vermektedir. Tuttuğum yerden az kişi gelirse hafıza israfı olur. Fazla kişi gelecekse yine aynı işlemi yapıp başka yere gider. Tek artısı herkesin yeri belli.

Linked List

Yan yana zorunluluğu olmadan verileri tutmamıza yarar. Hafızada dağılmış bir şekilde tutulur. Yeni gelen eleman için hafıza da yeni bir alan açmamıza gerek kalmaz. Yeni gelen kişi son sıradaki kişiye gideceği yeri bildirir ve istediği yere oturur. Herkesin lokasyon bilgisi vardır ama yanyana değildirler.

Array ve linked list karşılaştırma

Arrayin istediğimiz elemanına sabit sürede erişebiliyoruz.

Linked listte eleman eklemek ve silmek daha kolay.

Arrayde sadece elemanı tuttuğumuz için daha az yer kaplıyor.

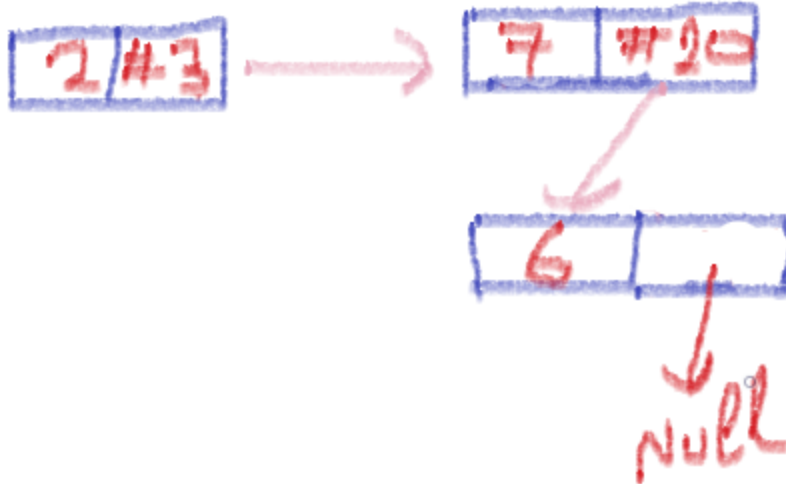
Linked listte elemanların bir blok olarak tutulması gerekmiyor, hafızada blok olarak yer yoksa da kullanılabilir.

Linked listte bir elemana ulaşmak için daha çok efor harcarız. Arraylerde random access mantığı ile yerini bilip gideriz.

Array da memory locality vardır. Yanyana tutulan şeyler yapılan işlemlerde hızlilik sağlar.

Linked list eleman ekleme - çıkarma

2,7,6,

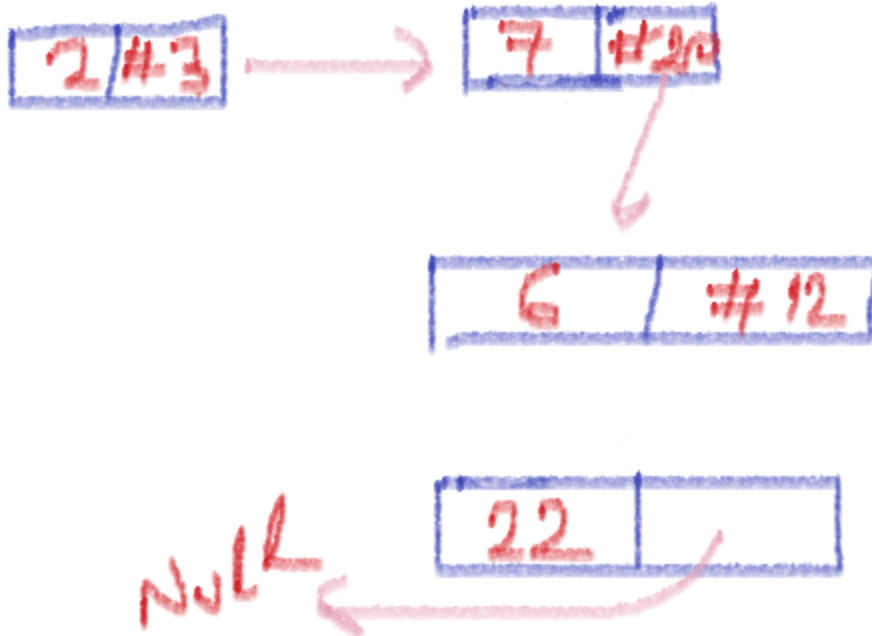


Gelin 3 elemanlı bir hücre oluşturalım.

Eleman Ekleme

- Adresi #12 olan 22 sayısını listeye eklemek istiyoruz. Yapmamız gereken 6 hücreğine 22 sayısının adresini yazmak.

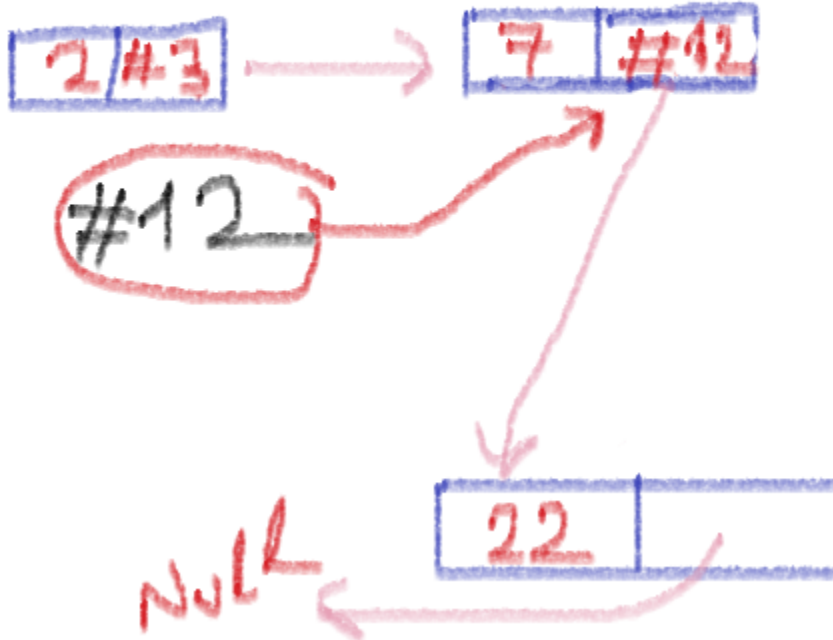
2, 7, 6, 22 → #12



Eleman Çıkarma

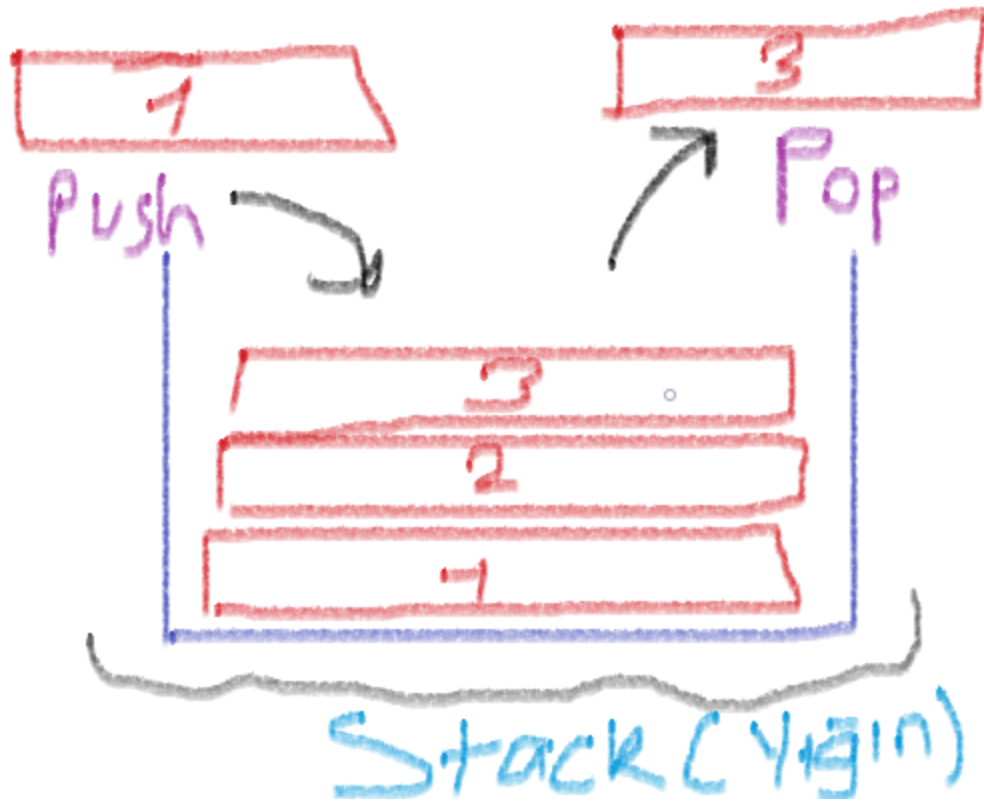
- Adresi #20 olan 6 numaralı hücreni çıkarmak istiyoruz. Linked-List'de bir önceki eleman adresini tutuyordu. Yani 7 numaralı hücrede bulunan 6'nın hücre adresini siliyoruz. Yerine 22 numaralı hücrenin adresini yazıyoruz.

2, 7, ~~16~~, 22 → #12



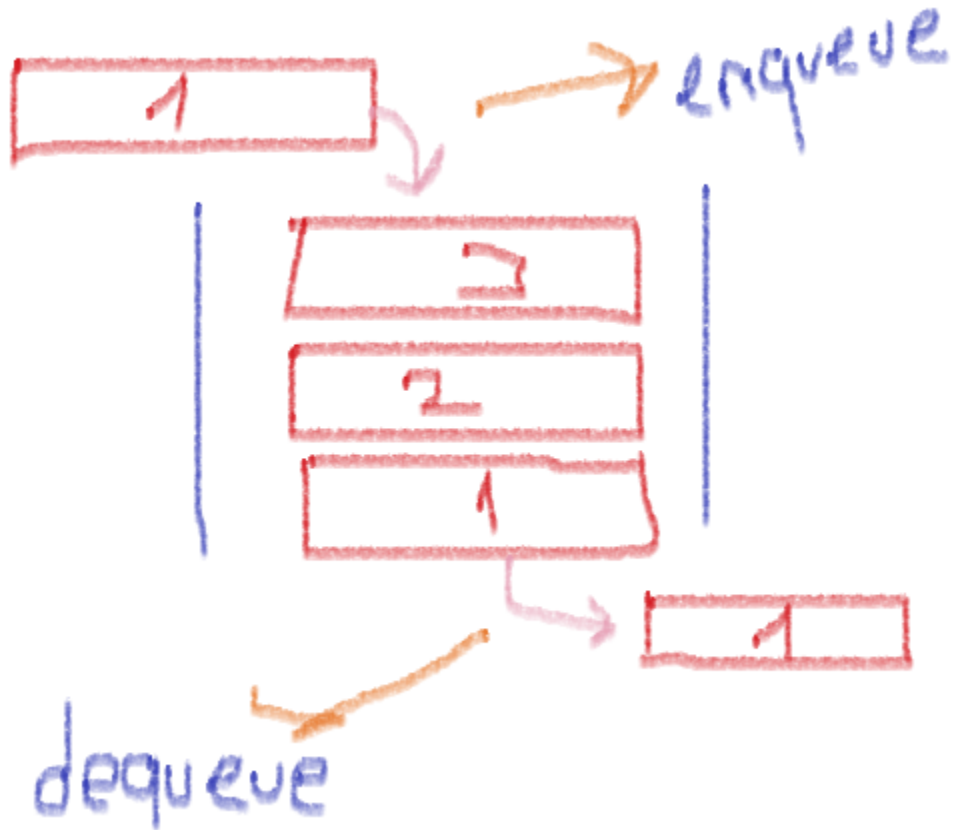
Stack

- Stack, LIFO (Last in First out) (En son giren en önce çıkar) mantığına dayanan, elemanlar topluluğundan oluşan bir yapıdır. Gelin hemen örneğimize geçelim. Taşınırken topladığınız koli kutusu düşünün. İçerisinde kitaplar var ve en, boy olarak koliye tam olarak koyuluyor. Mantıken kolinin altı kapalı ve üst üste koymanız gerekmektedir. Yeni taşındığınız yerde çıkartırken en üstekinden başlarsınız. İşte stack (Yığın) da aynı mantıkta çalışıyor.
- Yığılara eleman eklerken veya çıkartırken bazı methodlar uygulanır. Bunlardan biri push, diğeri ise pop. Push, yığının üzerine eleman eklemek için kullanılır (Koliye kitap koymak). Pop ise, yığından eleman çıkarmak için kullanılır.



QUEUE

- Queue (Kuyruk), FIFO (First in First out) (İlk giren ilk çıkar) prensibine dayanan, girişlerde ve çıkışlarda belirli bir kurala göre çalışan yapıdır. Stack de verdiğimiz örneği kuyruğa göre uyarlayalım. Biz örnekte altı kapalı bir koli kutusunu düşünmüştük. Şimdi o koli kutusunun altı yırtılmış. Sonuç olarak ne oluyor? İlk giren ilk çıkmış oluyor.
- Queue (Kuyruk)'da eleman eklemesi yaparken enqueue methodunu kullanıyoruz. Eleman silerken ise dequeue methodunu kullanıyoruz.



Hash Function/ Hash Table

Indexleme

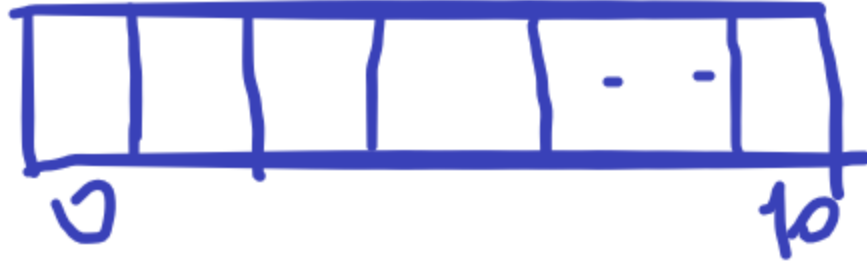
Arraylerde 0 bazlı bir indexleme vardır. Bazı programlama dillerin 1 bazlı indexlemeler olsa da genel olarak 0 bazlı indexleme kullanılır.


$C = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$
4 elemanlı
 $C[0] = 1. \text{ eleman}$
 $C[1] = 2. \text{ eleman}$
 $C[3] = 4. \text{ eleman}$
Indexleme

Hash Function/ Hash Table

Hash Table, key value prensibine dayanan bir array kümesidir. Key olarak çağırdığınız elemanın değerini (value) yansıtır.

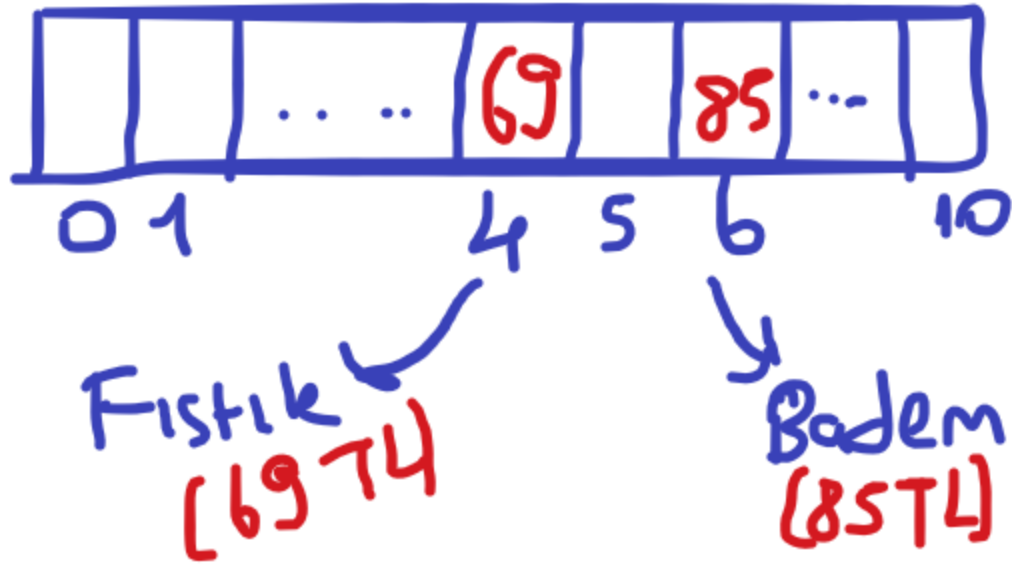
Hash Table yerine dizileri kullanabilirdik. Fakat her ürünü ve fiyatını tek tek aramak istemediğimiz için hash table kullanıyoruz. Peki bu süreç nasıl işliyor? Hemen bir örnek yapalım. Örneğimiz bir kuru yemiş dükkanından gelecek.



"Fıstık" →  → 4

"Badem" →  → 6

Bu kısımda ilk olarak bulunan ürün sayımız kadar değeri olan bir Array oluşturduk.
Daha sonra hash fonksiyonundan ürünleri geçirerek index değerlerine ulaştık.



Şifrelendiği için artık her badem keyi gönderildiğinde 85TL, fıstık keyi gönderildiğinde ise 69 sonucu verecektir.

Özetle, elimizde var olan verileri bir fonksiyondan geçirip indexliyoruz. Bu fonksiyona hash function, bu fonksiyon ile birleştığımız dizi yapısına ise Hash Table diyoruz.

Hash Function

Hash Function (Karma Fonksiyonu), karma fonksiyonu olabilmesi için bazı temel şartlar vardır. Bunlar;

1. Gönderdiğimiz anahtarlar (keys) farklı olmasına rağmen bize aynı sonuçları veriyorsa bu bir hash function değildir.
2. Fonksiyona gönderilen anahtarlar aynı fakat sonuç farklı ise hash function değildir.
3. Hash Table için kullanılan dizinin boyutu verilen sonuçların sayısı kadar olmalı.

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 olarak araçları bir hash function dan geçirelim. Bu fonksiyonumuz son harflerine 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.
- Çarpışma sayısı arttıkça aradığımız şeyi bulma hızı azalır.

Algoritma Analizi

- Algoritma analizi, var olan kaynaklara göre en uygun algoritmayı seçmek için uygulanır. Peki algoritma analizi en iyi nasıl yapılır? Kulağa karmaşık geliyor ama çok basit. Programlama dillerinden ve donanımlardan bağımsız bir şekilde Algoritma analizi yapılmalıdır. Aksi taktirde en uygun sonuç alınamayabilir.
- Donanımlar veya programlama dilleri farklı cihazlarda aynı performansı vermeyebilir. Örnek verecek olursak, cep telefonları için uygulama tasarladığımızı varsayalım. Bu uygulamanın performansı Apple telefonlar için farklı, Android telefonlar için farklı, arasında donanım farklı olanlar için ayrı olacaktır. Donanım ve diller ile algoritma analizi pek sağlıklı değildir.
- Algoritma analizi, bir algoritmanın çalışabilmesi için gerekli koşulların sağlanıp sağlanamadığını gösteren bir parametredir.

Ram Modeli

Bir algoritmayı farklı cihazlarda denemek bize pek fazla bir sonuç çıkarmıyordu. Çünkü kaynaklar değişebiliyordu. Bu probleme genel bir çözüm getirebilmek için hayalî bir cihaz düşünelim. Bu cihaz üzerinde bütün algoritmaları çalıştırdıktan sonra bize bir sonuç veriyor.

Bu hayalî cihaza RAM (Random Access Machine) diyoruz. Ram, algoritmalar arasındaki farkları belirlemek için kullanacağımız bir araç olacak.

Her işlemin birim zamanı var. Döngüler, kaç defa işlem yapıyorsa, (işlem sayısı * kaç kere tekrar edeceği) kadar birim zaman alır. Toplama, Çıkarma, and, or gibi aritmetik işlemler, 1 birim zaman alır.

Time Complexity

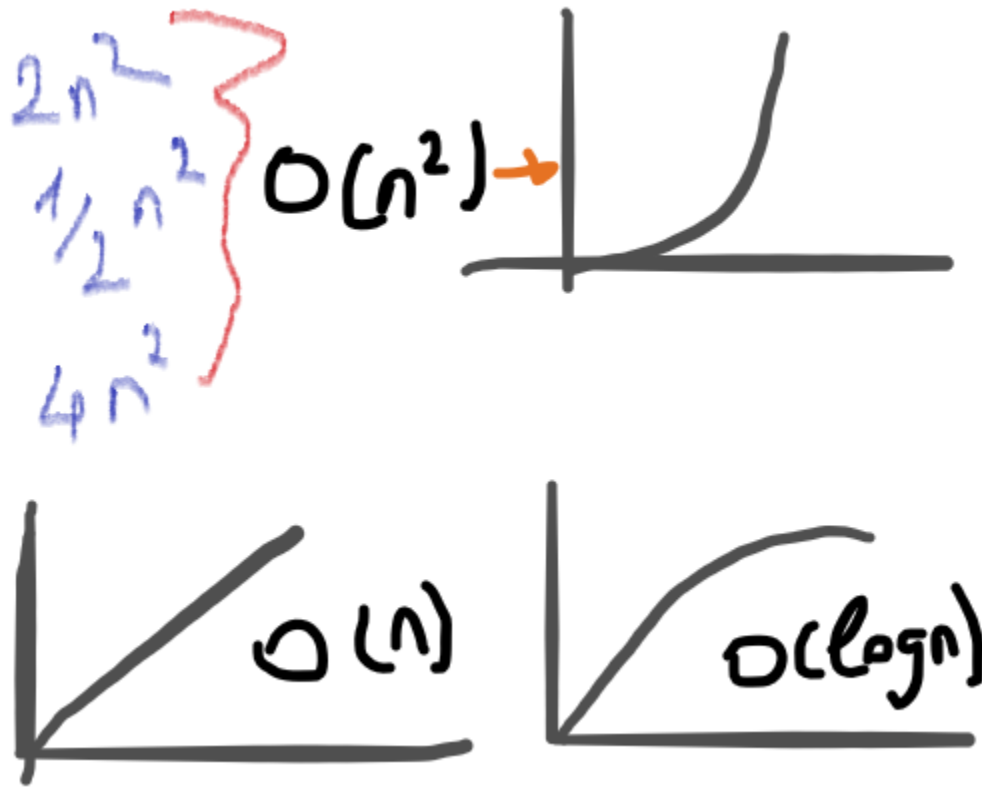
Algoritmanın verimli olması için belli kurallar vardır.

Örnekten bıktık diyenleri duyar gibi oluyorum. Hepsi sizin iyiliğiniz için :).

- Örnek: Raflara kitap yerleştirmek.
- Kitapları, gelişigüzel raflara dağıtırsak aradığımız kitabı daha fazla zamanda bulabiliriz. Aslında bu bir worst case'dir. Kitapları filtrelememiz gerekir. Kalın olanları bir rafa, ince olanları bir rafa, küçük boyutta olanları bir rafa koyduğumuz zaman aradığımız şeyi daha rahat bulabiliriz. Algoritma, en kötü senaryoya ne kadar hazırsa, bizi o kadar memnun edebilir.
- Algoritmalar için genellikle sık kullanılan average case'dir. Kitapların bölümüne göre kaç tane olduğunu biliyorsak average case kullanabiliriz. En büyük rafı miktarı fazla olana ayırabiliriz. Input yoksa average zordur!!!!
- Bir diğer senaryomuz ise best case'dir. Beklediğimiz en iyi durum. Kitap örneğine devam edecek olursak, bütün kitapların ayrı raflarda olması, alfabeye göre sıralanması best case olarak ifade edilebilir. Çünkü aradığımızı rahatlıkla bulabiliyoruz.

Big-O Notation

Big-O Notation grafikleri :

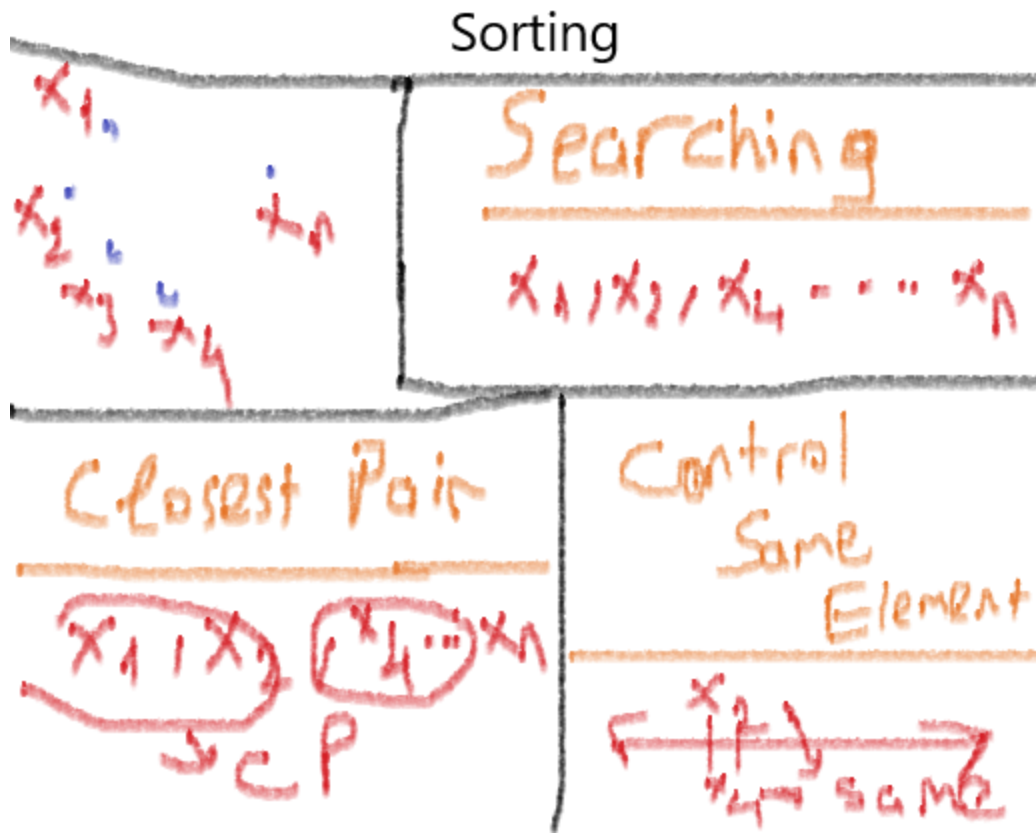


- İki farklı arama yöntemimiz var. Bunlardan A algoritması sayfa sayfa, B algoritması yarıya bölüp tarıyor. Sizce hangisi daha hızlı çalışır? Tabii ki B algoritması. Peki neden? Sürekli tarayacağı alan azalıyor. A algoritması daha işlemini bile yarılammışken, B algoritması sonuca ulaşıyor.
- N tane işlem üzerinden big-o gösterimi yapalım. A algoritması input olarak kaç sayfa varsa o kadar işlem yapıyor. B algoritması ise sayfa sayısını azaltmak için alfabetik sıraya göre sağ ve sol olarak yarıya indiriyor.

	n tane sayfa
A	$O(n)$
B	$O(\log n)$ \downarrow $2^x = n$

Sorting

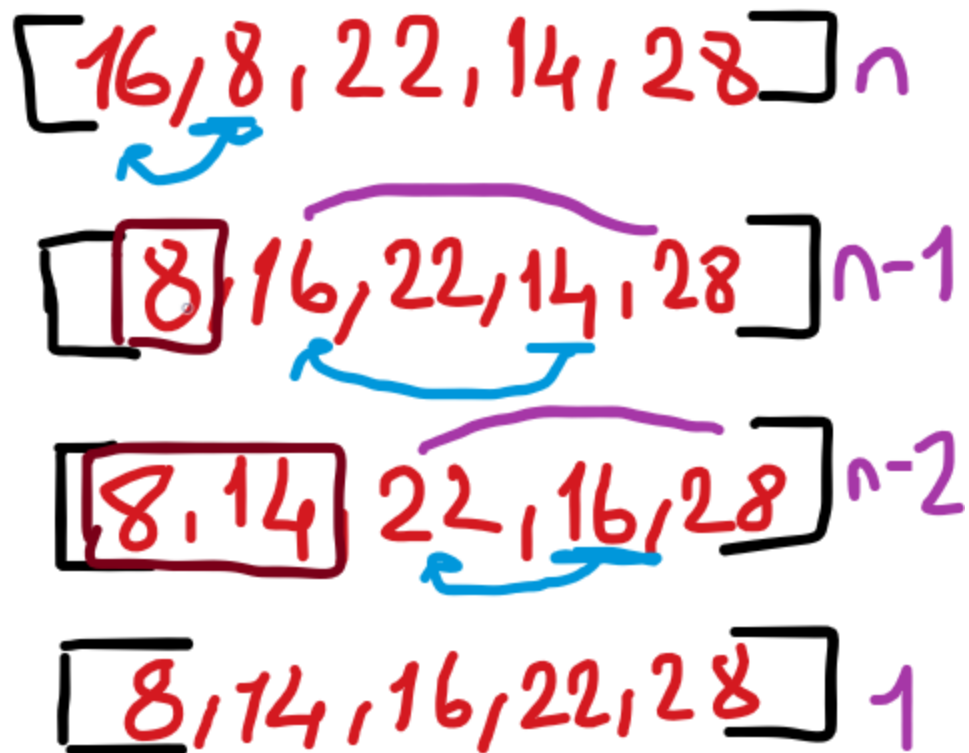
Sorting, kendinden sıralama algoritmaları olarak bahsetmektedir. Sorting, bir eleman dizisini, belirli sıralama kurallarına göre sıralama yapar.



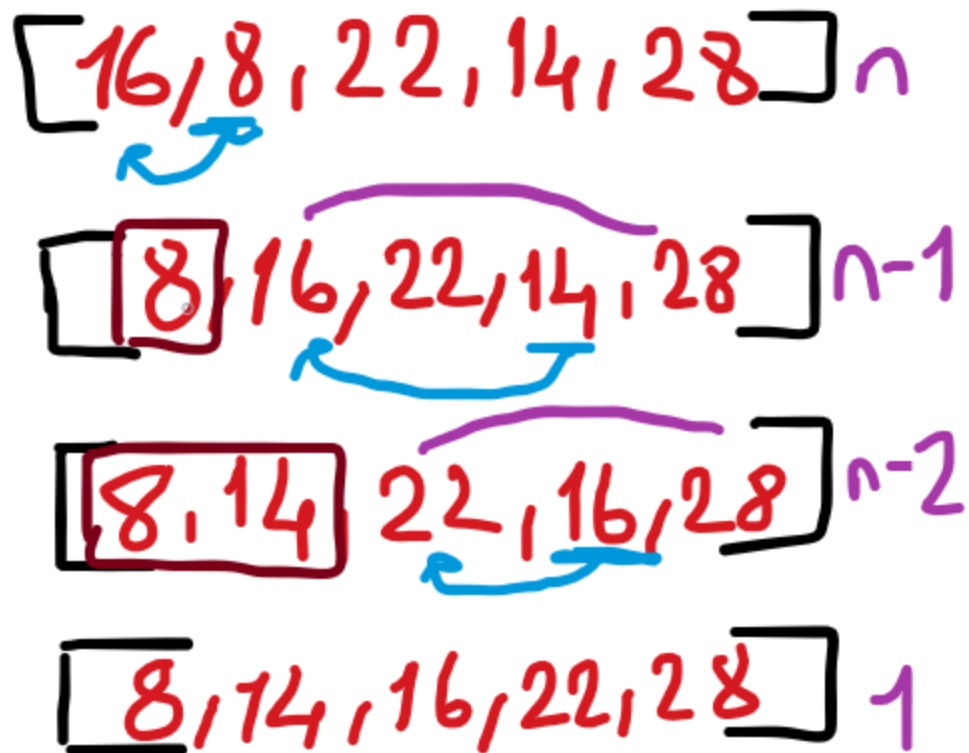
1. **Searching** yöntemini kullanarak elemanlarımızı sıraladık. Bunun sebebi, eleman ararken işimizin kolaylaşmasını istiyoruz.
2. **Closest Pair** yöntemini kullanarak birbirine yakın sayıları gruplandırdık ki arama yaparken zamanımızı efektif bir şekilde kullanalım.
3. **Aynı eleman kontrolü**: birbiriyle aynı olan sayıları örüntü içerisinde kaç tane aynı eleman varsa sayısını öğrenebilirim.
4. **Mode bulma**: eleman dizisini search ettikten sonra elemanların yan yana olanları sayarsam daha hızlı mode bulabilirim.

Selection Sort

En basit sorting algoritmalarından biridir.



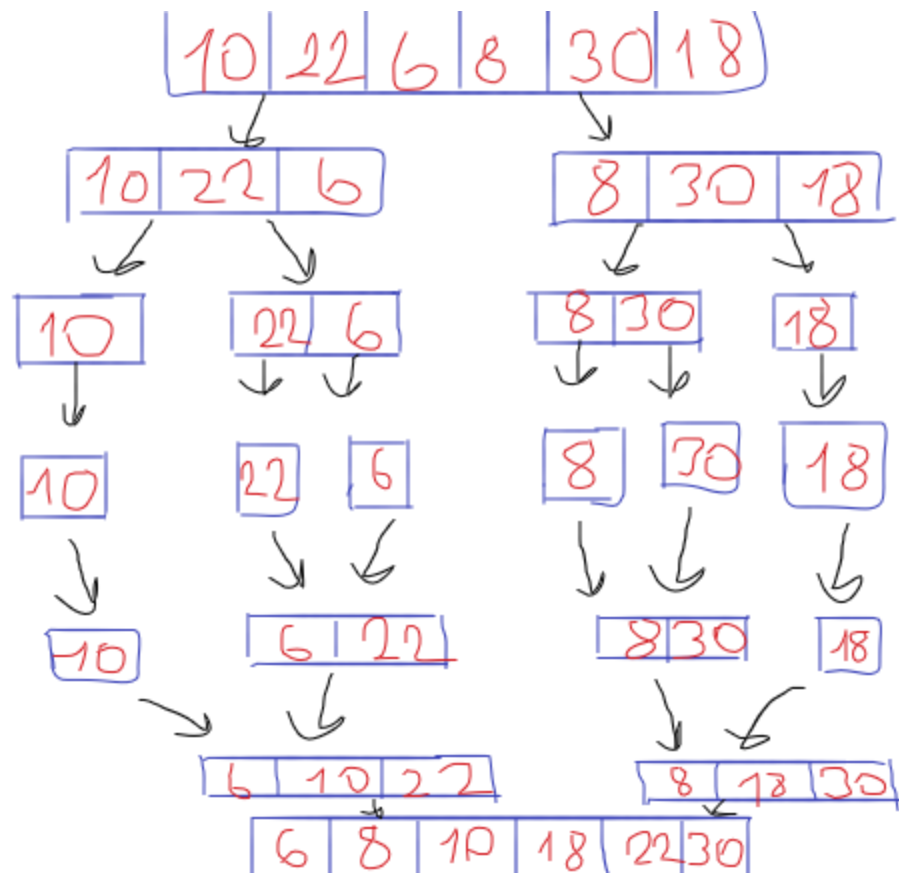
- Verilen örüntüye ait en küçük elemanı buluyor ve en baştaki sayı ile yer değiştiriyor. Peki ya devamı? İkinci en küçük elemanı buluyor ve 2. sıra ile değiştiriyor. Baktın ki 2.sıradaki eleman en küçük **hiç dokunma!!!**. Hemen 3. sıraya geç. 4, 5 derken dizi bitti. İşte insertion sort'un temel çalışma prensibini öğrendin.

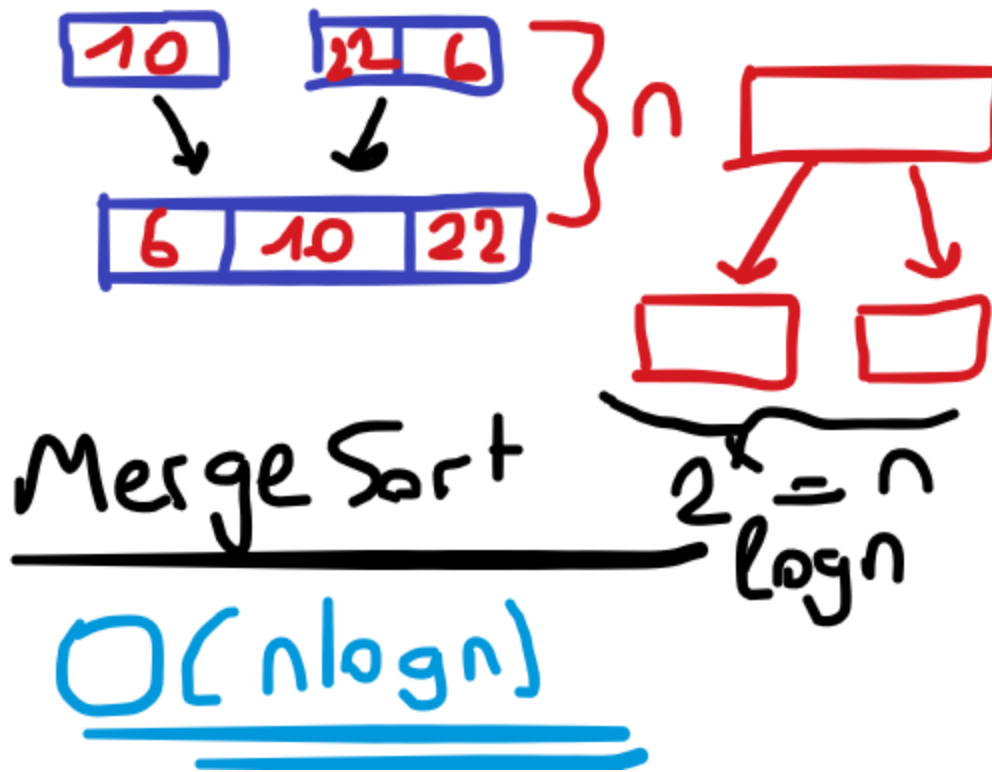


Merge Sort

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).

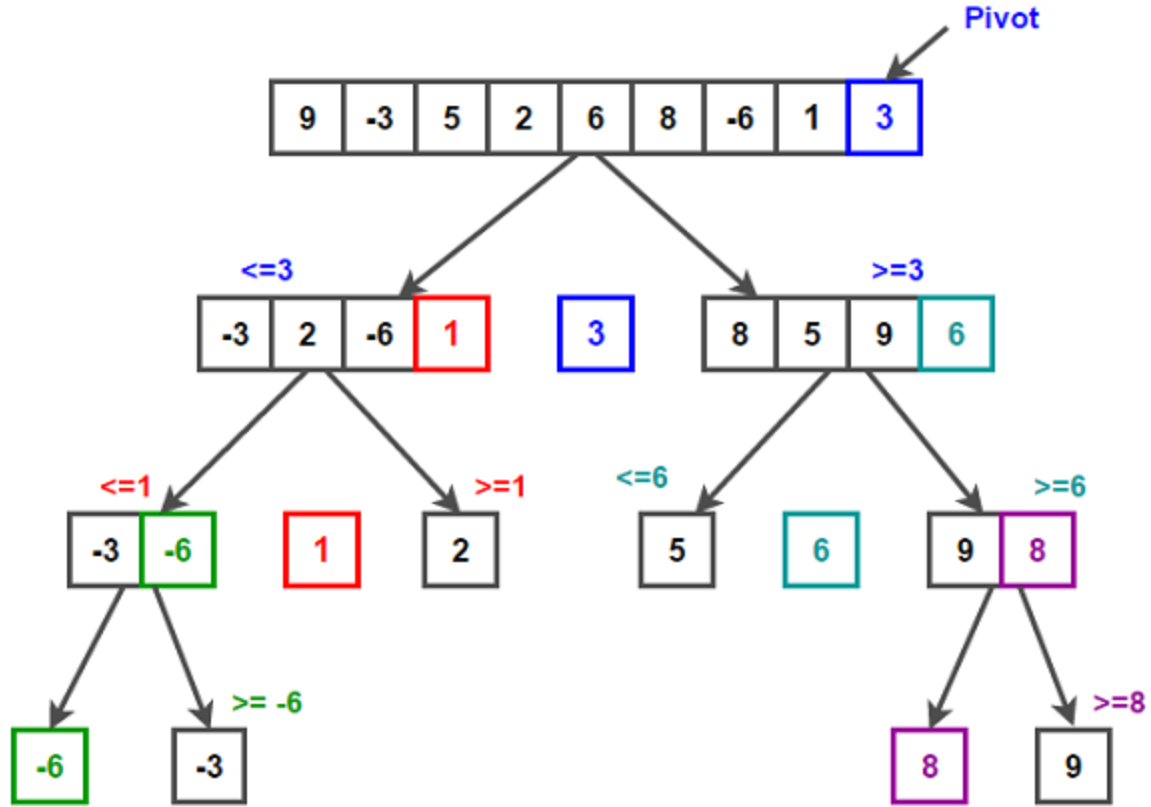




Insertion sort'da, time complexity n^2 olduğundan ötürü çalışma zamanımız artıyordu. Merge sort'da ise $n \log n$ olduğu için açık ara performans olarak daha iyi diyebiliriz.

Quick Sort

Hızlı sıralama günümüzde çok yaygın olarak kullanılan bir sıralama algoritmasıdır. N tane sayıyı average case e göre big-o $n \log n$, worst case e göre big-o 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.

Searching

- 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

Linear search, tek tek elemanları dolandıktan sonra istediğim elemanın olup olmadığına bakmaktır.

- 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.

- Big-o ya göre incelediğimizde bizim worst case'imiz neydi? Elemanın dizinin sonunda bulunmasıydı. Bu sebepten ötürü n elemanımız varsa big-o notasyonumuz otomatik olarak n oluyor.

Binary Search

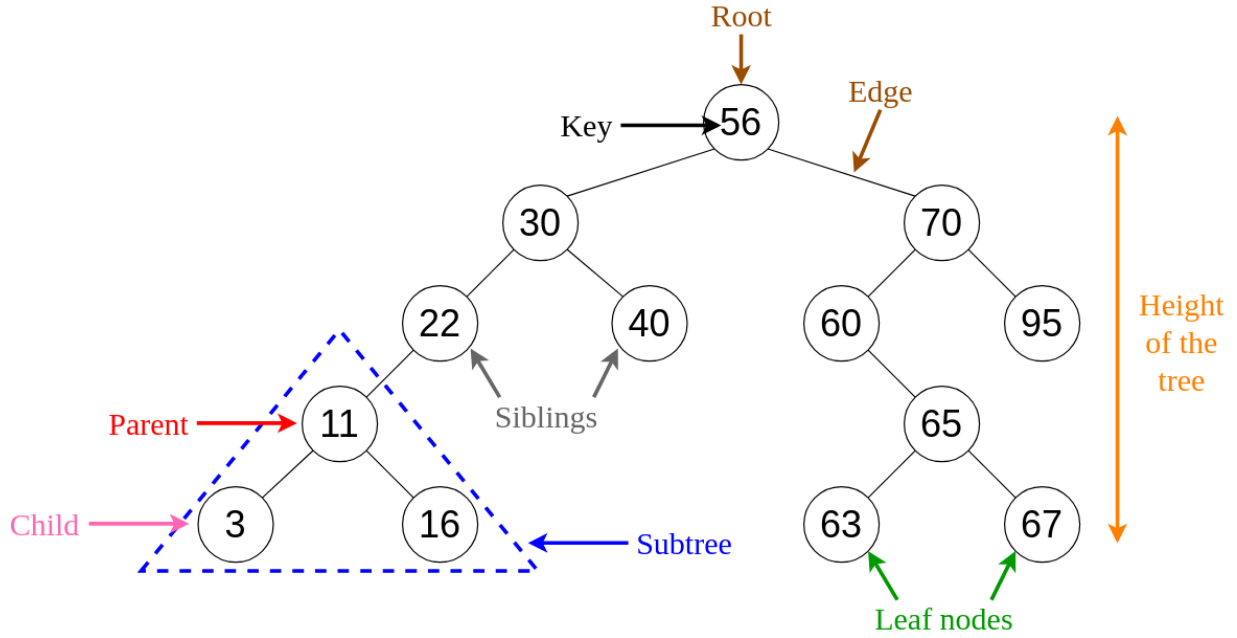
İkili arama algoritması, elimizde bulunan veri dizisini sıralı olduğunu varsayıyor, bu durumu değiştirerek sonuca varmak istiyor.

- İkili arama algoritması, diziyi her seferinde ikiye bölerek ikili arama yapar. Sıralı bir listem var ise benim Big-o $\log n$ olarak karşımıza çıkıyor.
- 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.