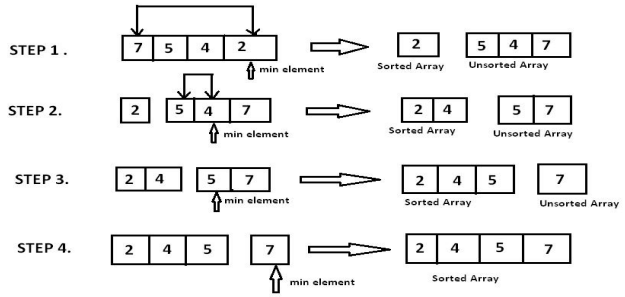
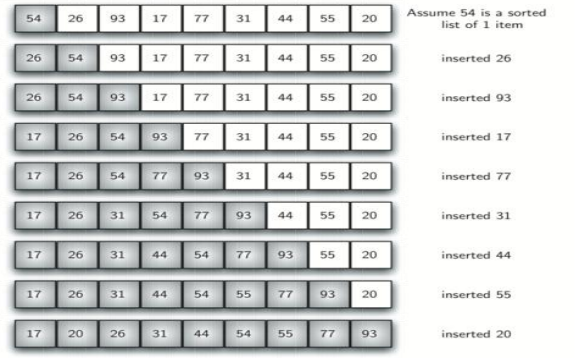
: A L G O R İ T A M A L A R :

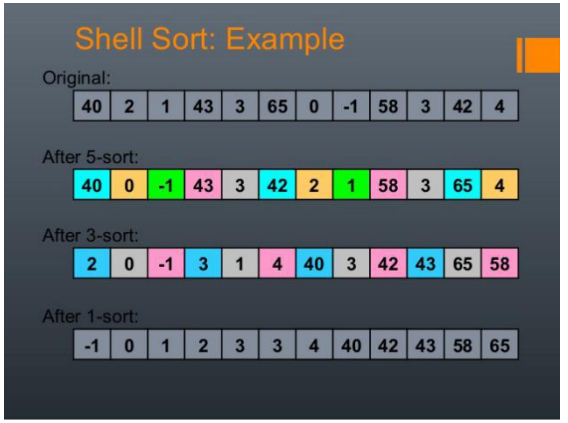
Ders01 - Sort Algoritmaları (https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html)



Selection Sort: Verinin hafızada sıralı tutulması için geliştirilen sıralama algoritmalarından (sorting algorithms) bir tanesidir. Basitçe her adımda dizideki en küçük sayının nerede olduğu bulunur. Bu sayı ile dizinin başındaki sayı yer değiştirilerek en küçük sayılar seçilerek başa atılmış olur.



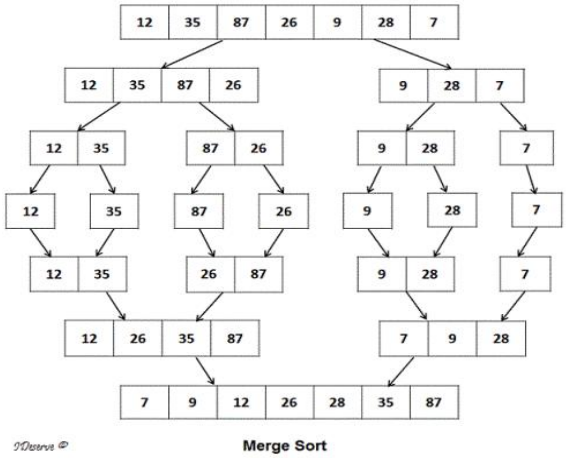
Insertion Sort: Programlaması oldukça basit ancak performansı bölme sıralaması merge sort, quick sort gibi sıralamalara göre nispeten yavaş bir sıralama algoritmasıdır.



Shell Sort: Bilgisayar bilimlerinde kullanılan sıralama algoritmalarından birisi de shell sorttur. İsmi Türkçeye kabuk sıralaması olarak çevrilsede aslında Donald Shell isimli, algoritmayı ilk bulan kişinin isminden gelmektedir. Çalışması aşağıdaki örnek üzerinde anlatılmıştır:
Sıralama işlemi için öncelikle bir atlama miktarı belirlenir. Atlama miktarının belirlenmesi için çok çeşitli yollar bulunmasına karşılık en basit yöntem elimizdeki sayıların yarısından başlamaktır.

Merge Sort:

Verinin hafızada sıralı tutulması için geliştirilen sıralama algoritmalarından bir tanesidir. Basitçe sıralanacak olan diziyi ikişer elemanı kalan parçalara inene kadar sürekli olarak ikiye böler. Sonra bu parçaları kendi içlerinde sıralayarak birleştirir. Sonuçta elde edilen dizi sıralı dizinin kendisidir. Bu açıdan bir parçala fethet (divide and conquere) yaklaşımıdır.

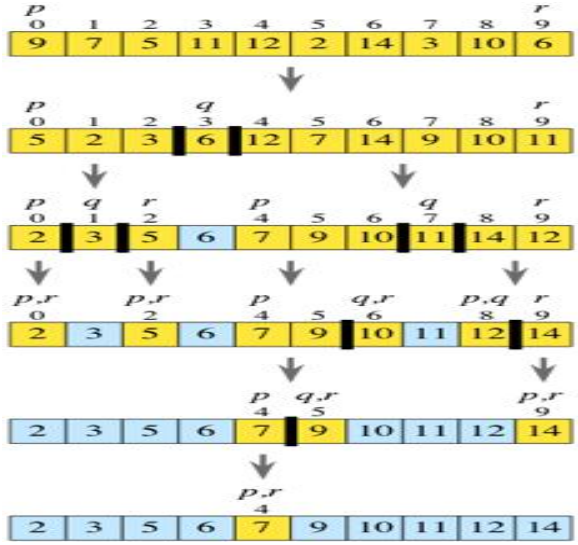


Merge Sort

Quick Sort:

Quiksort günümüzde yaygın olarak kullanılan bir sıralama algoritmasıdır. Quicksort algoritması, sıralanacak bir diziyi daha küçük iki parçaya ayırıp oluşan bu küçük parçaların kendi içinde sıralanması mantığıyla çalışır.

1. Diziden herhangi bir elemanı pivot (kilit) eleman olarak seçer.
2. Diziyi, pivot elemandan küçük olan bütün elemanlar pivot elemanın önüne, pivot elemandan büyük olan bütün elemanlar pivot elemanın arkasına gelecek biçimde düzenler. Pivot elemana eşit olan sayılar sıralamanın küçükten büyüğe ya da büyükten küçüğe olmasına bağlı olarak pivot elemanın her iki tarafına da geçebilir.
3. Quicksort algoritması özyineli (recursive) çağrılarak, oluşan küçük diziler tekrar sıralanır.
4. Algoritma eleman sayısı sıfır olan bir alt diziye ulaşana kadar bu işlem devam eder.
5. Eleman sayısı sıfır olan bir alt diziye ulaşıldığında algoritma bu dizinin sıralanmış olduğunu varsayar ve sıralama işlemi tamamlanmış olur.



Stability (Kararlılık):

Sıralama algoritmalarında bazen kararlılık ile ilgili bir takım bilgiler görürüz.Kısaca açıklamak gerekirse, sırasız bir dizide aynı değerlere sahip elemanların dizilişi, dizi sıralandığında da korunuyorsa algoritma kararlıdır.

Kararlı özelliğine sahip algoritmalar; Insertion sort, Merge sort, Bubble sort.
Kararsız özelliğine sahip algoritmalar; Selection sort, Shell sort, Heap sort, Quick sort.

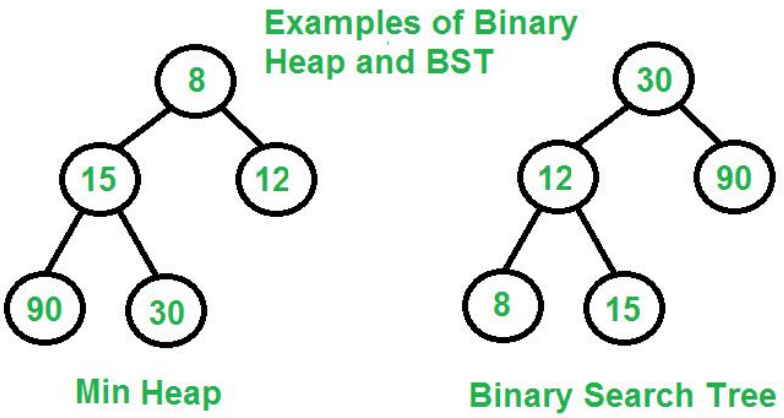
Ancak verilen kararlı olmayan aloritmalar kolayca kararlı hale getirelebilirler.

Dijkstra 3-Way Partitioning:

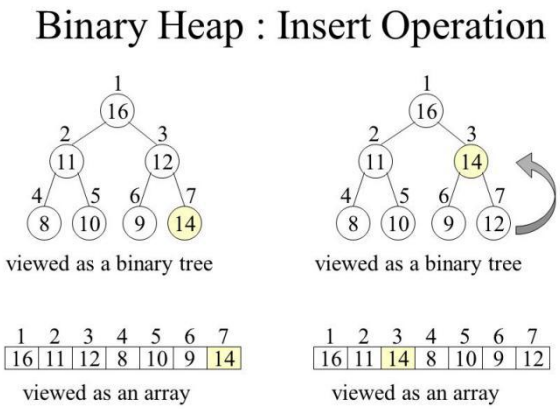
https://www.cs.princeton.edu/courses/archive/fall12/cos226/demo/23DemoPartitioningDijkstra.pdf

Heap Tree (Yığın Ağacı):

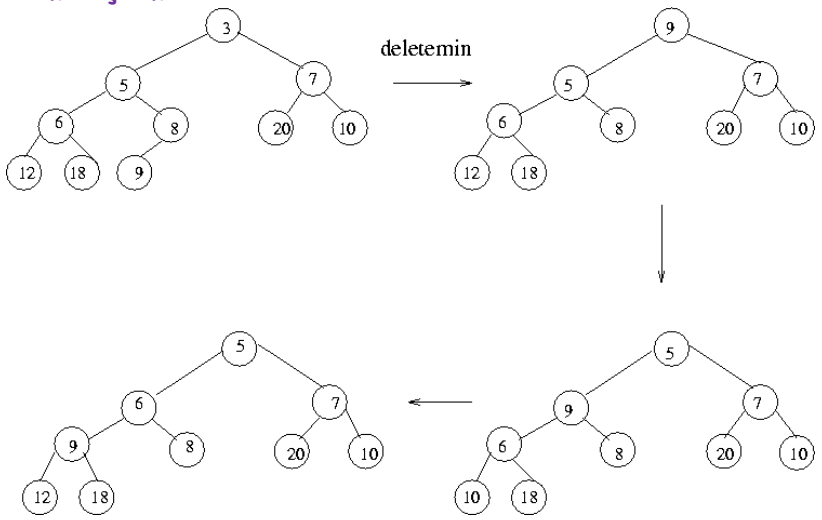
Yığın ağacı bilgisayar bilimlerinde özellikle sıralama amacıyla çokca kullanılan bir veri yapısıdır. Bu veri yapısı üst düğümün (atasının) alt düğümlerden (çocuklarından) her zaman büyük olduğu bir ikili ağaç (binary tree) şeklinde düşünülebilir.



Ekleme İşlemi:



Silme İşlemi:



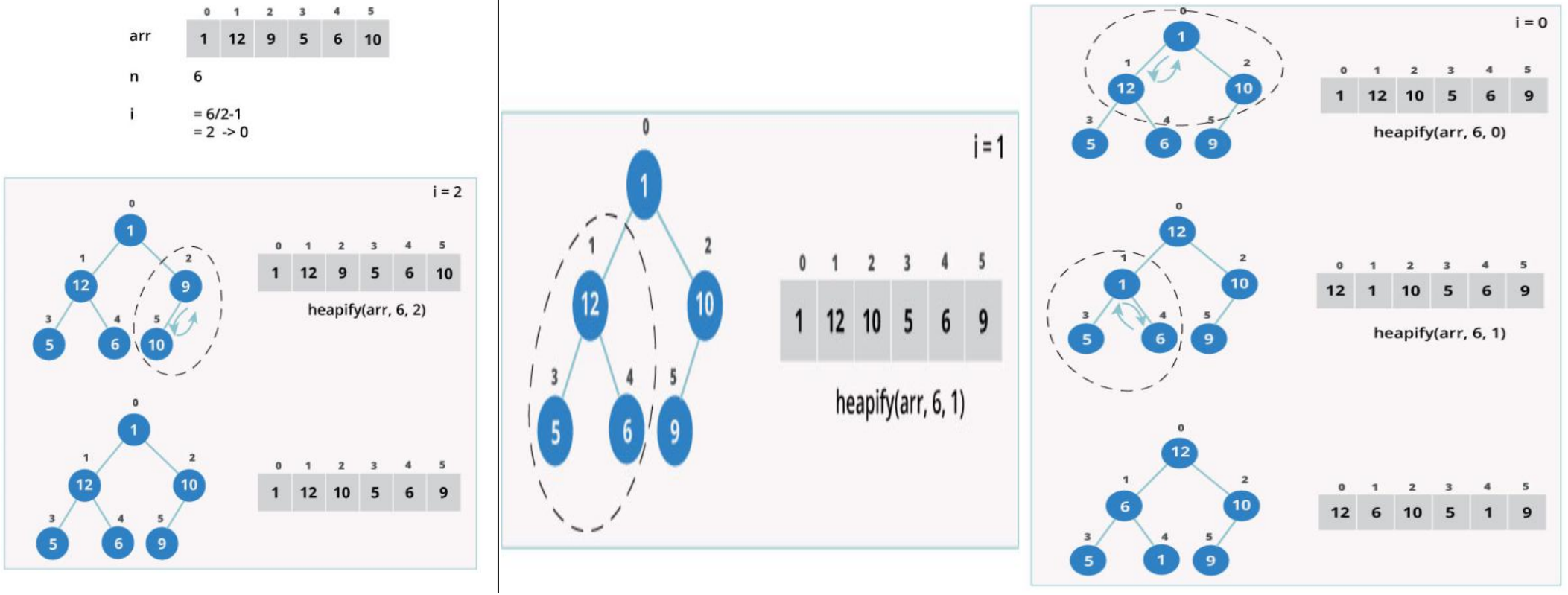
Priority Queue:

order-of-growth of running time for priority queue with N items

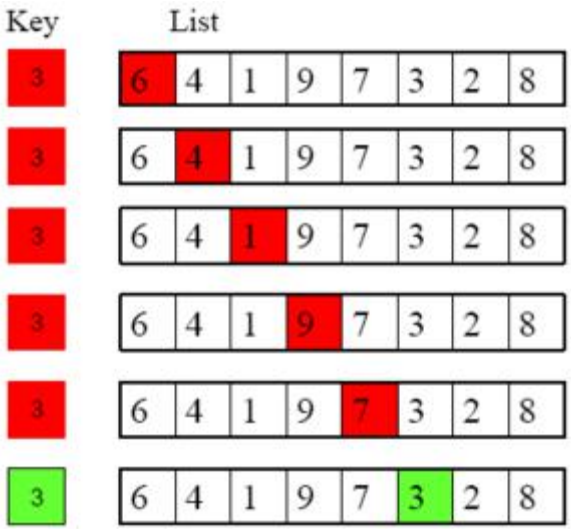
implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
goal	log N	log N	log N

Heap Sort: (https://www.cs.usfca.edu/~galles/visualization/HeapSort.html **)**

Verinin hafızada sıralı tutulması için geliştirilen sıralama algoritmalarından (sorting algorithms) bir tanesidir. Yığınlama sıralaması, arka planda bir yığın ağacı (heap) oluşturur ve bu ağacın en üstündeki sayıyı alarak sıralama işlemi yapar. (Lütfen yığın ağacındaki en büyük sayının her zaman en üstte duracağını hatırlayınız.)



Ders02 - Search Algoritmaları



Sequential Search:

Arama yöntemleri içinde en basit yöntem olarak karşımıza çıkan sırayla (sequential) arama yöntemi bir dizi veya bağlantılı liste (linked list) içinde yer alan elemanların içinde belirli bir verinin, elemanların sırayla kontrol edilerek aranması olarak tanımlanabilir.

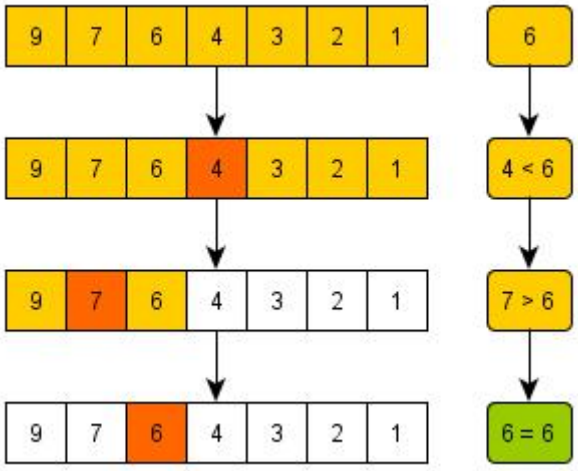
https://www.cs.usfca.edu/~galles/visualization/Search.html

Binary Search:

Bilgisayar bilimlerinde bir bilgi kaynağı veya veri yapısı üzerinde problemi her adımda iki parçaya bölerek yapılan arama algoritmasının ismidir. Bu anlamda bazı kaynaklarda bölerek arama olarak da geçmektedir.

Arama algoritması, yapı olarak parçala fethet (divide and conquere) yaklaşımının bir uygulamasıdır.

Bu yazı kapsamında diziler üzerinde ikili arama işleminin nasıl yapıldığı anlatılacaktır. Ancak algoritma, diziler dışında çok farklı veri yapıları ve veri kaynakları için de kullanılabilir. Algoritmanın her durumda çalışması aşağıdaki şekildedir.



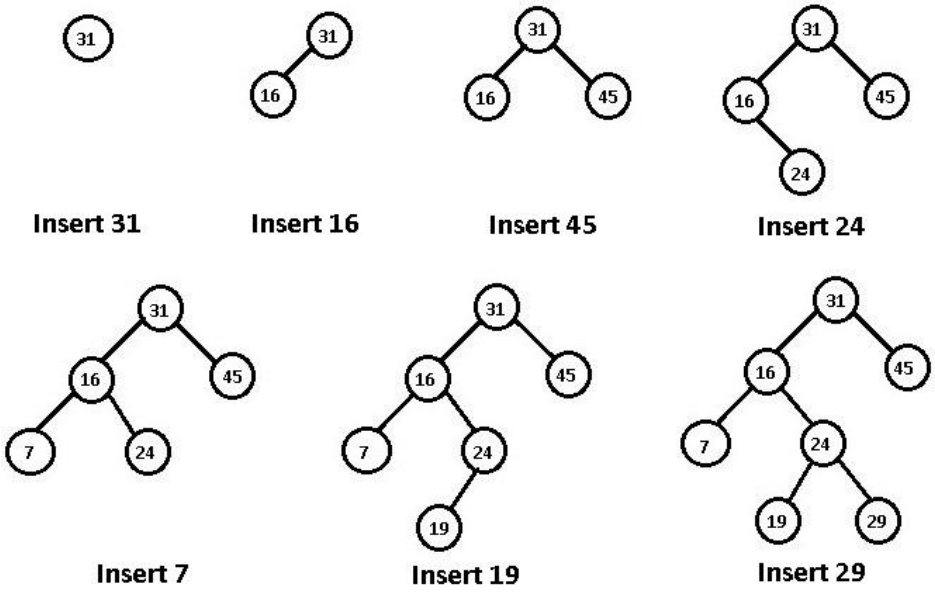
1) Problemde aranacak uzayın tam orta noktasına bak
2) Şayet aranan değer bulunduysa bit
3) Şayet bakılan değer aranan değerden büyükse arama işlemini problem uzayının küçük elamanlarında devam ettir.
4) Şayet bakılan değer arana değerden küçükse arama işlemini problem uzayının büyük elemanlarında devam ettir.
5) Şayet bakılan aralık 1 veya daha küçükse arana değer bulunamadı olarak bitir.

Binary Search Tree: (https://www.cs.usfca.edu/~galles/visualization/BST.html)

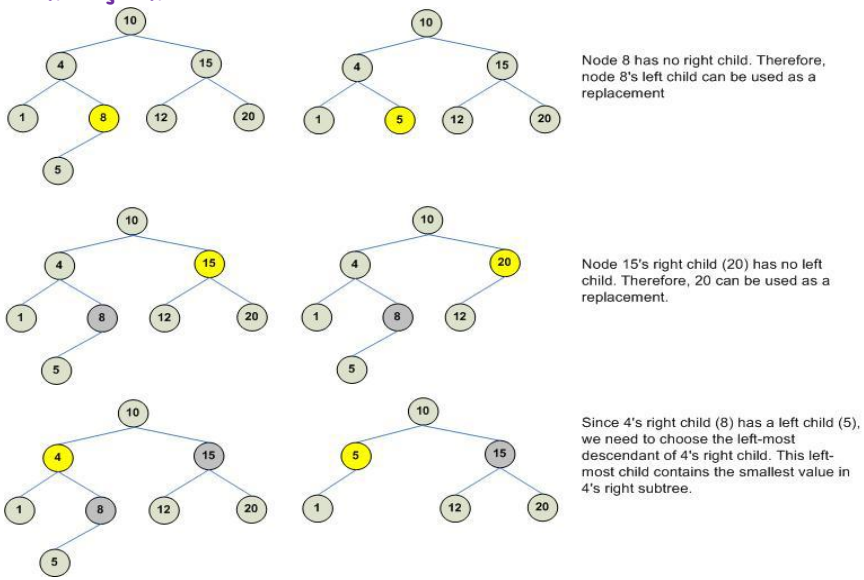
İkili ağaçların (Binary Tree) özel bir hali olan ikili arama ağaçlarında, düğümlerde duran bilgilerin birbirine göre küçüklük-büyüklük ilişkisi bulunmalıdır. Örneğin tam sayılardan (integer) oluşan veriler tutulacaksa bu verilerin aralarında küçük-büyük ilişkisi bulunmalıdır.

İkili arama ağacı, her düğümün solundaki koldan ulaşılabilecek bütün verilerin düğümün değerinden küçük, sağ kolundan ulaşılabilecek verilerin değerinin o düğümün değerinden büyük olmasını şart koşar.

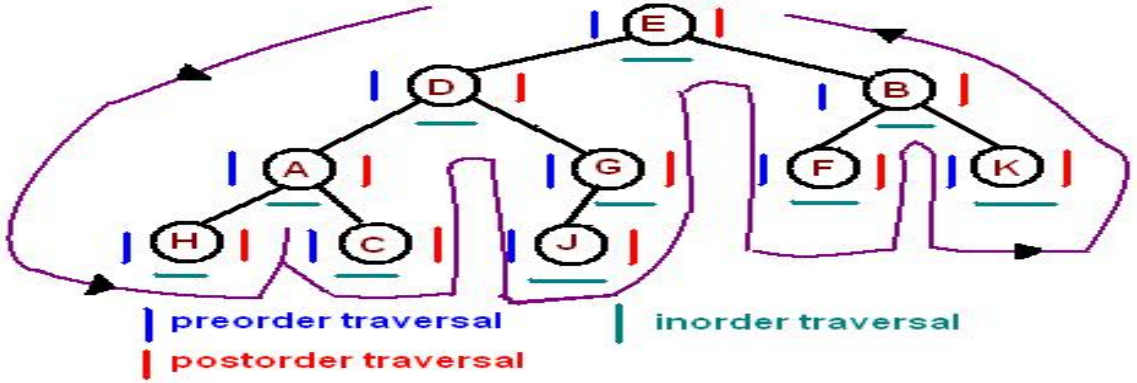
Ekleme İşlemi:



Silme İşlemi:



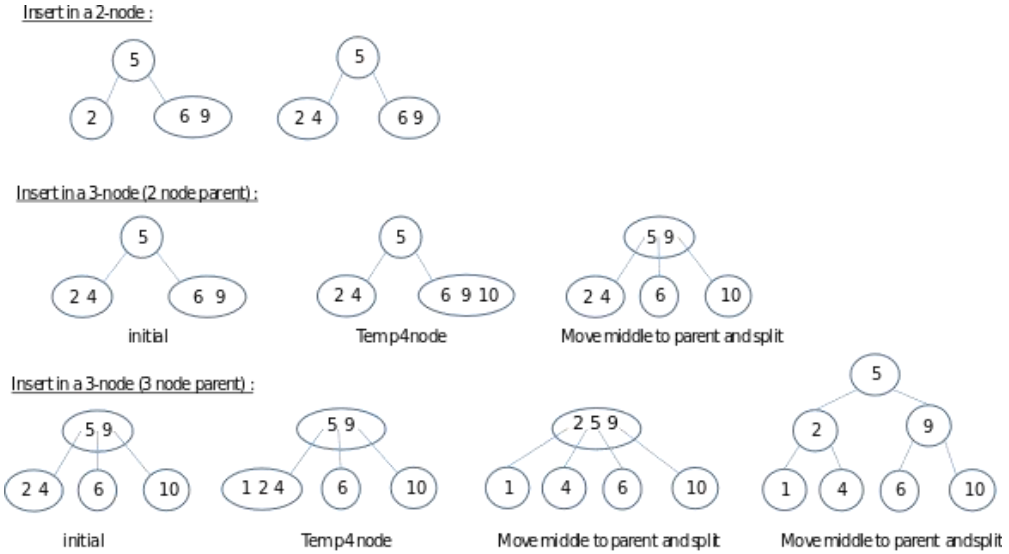
Traversals:



2-3 Search Tree:

Bilgisayar bilimlerinde kullanılan bir veri yapısıdır (data structures). Özel bir ağaç yapısıdır ve amaç, ağacı sürekli olarak dengeli (balanced) tutmaktır.

Ağaçtaki düğümlere (nodes) isim olarak 2 veya 3 ismi verilebilir. Her düğüm aldığı isme göre farklı işleme tabi tutulur. Düğümlerin isimlendirilmesi aşağıdaki şekilde yapılır.



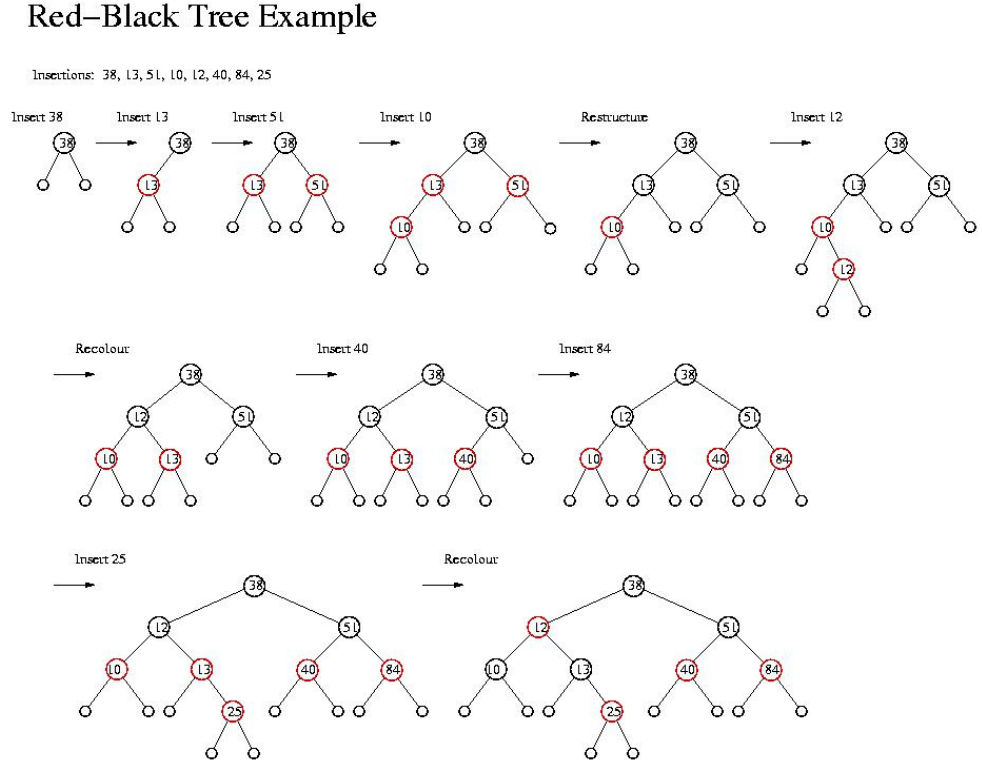
Red Black Tree: (https://www.cs.usfca.edu/~galles/visualization/RedBlack.html)

Bilgisayar bilimlerinde, veriyi ağaçta (tree) tutarken, ağacın dengeli (balanced) olmasını sağlayan bir algoritmadır. Algoritma, veriyi tutuş şekli sayesinde, arama, ekleme veya silme gibi temel işlemlerin en kötü durum analizi (worst case analysis) O(logn)'dir, yani algoritma n elman için bu işlemleri en kötü O(logn) zamanda yapmaktadır.

Kırmızı-siyah ağaçlar (red-black trees) tanım itibariyle ikili arama ağaçlarıdır (binary search tree) ve bu anlamda, herhangi bir düğümün solunda kendisinden küçük ve sağında ise büyük verilerin durması beklenir. Ağaçta ayrıca her düğüm için bir renk özelliği tutulur. Yani bir düğüm kırmızı veya siyah renk özelliği taşıyabilir. Ağaçtaki düğümlerin taşıması gereken bu özellikler aşağıdaki şekilde sıralanabilir:

i. Kök düğüm (root node) her zaman için siyahtır.
ii. Bütün yaprak düğümler (leaf nodes) siyahtır.
iii. Herhangi bir kırmızı düğümün bütün çocukları siyahtır.
iv. Herhangi bir düğümden, yaprak düğüme kadar gidilen bütün yollarda eşit sayıda siyah düğüm bulunur.
v. Yukarıdaki bu kurallar ışığında, herhangi bir düğümden, yapraklara kadar olan yolun, gidilebilecek en kısa yolun iki mislinden kısa olduğu garanti edilebilir. Diğer bir deyişle, ağacın aynı seviyedeki düğümleri aynı renktir. Ayrıca --->>

ağaçtaki renklendirme kökten başlayarak, siyah – kırmızı – siyah – kırmızı sıralamasıyla değişmektedir.



Hash Function:

Özetleme fonksiyonlarının çalışma şekli, uzun bir girdiyi alarak daha kısa bir alanda göstermektir. Amaç girende bir değişiklik olduğunda bunun çıkışa da yansımasıdır.

Buna göre özetleme fonksiyonları ya veri güvenliğinde, verinin farklı olup olmadığını kontrol etmeye yarar ya da verileri sınıflandırmak için kullanılır.

Anlaşılması en basit özetleme fonksiyonu modülo işlemidir. Buna göre örneğin mod 10 işlemini ele alalım, aşağıdaki sayıların mod 10 sonuçları listelenmiş ve gruplanmıştır:

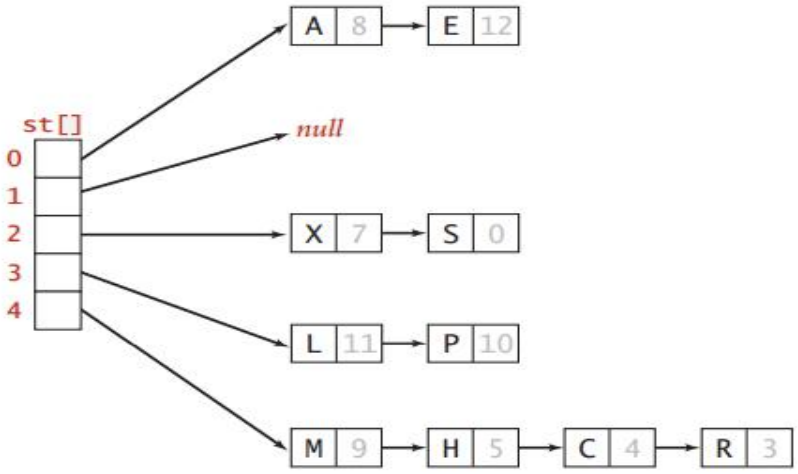
Sayılar: 8,3 ,4,12,432,34,95,344,549,389,2339,349,54,81,17,62,94,67,44,9

Demet (Buket, Bucket)	Sayılar					
0						
1	81					
2	12	432	62			
3	3					
4	4	34	344	54	94	44
5	95					
6						
7	17	67				
8	8					
9	389	2339	349	9		

Kısaca yukarıdaki sayıların hepsi 1 haneli bir sayıya özetlenmiştir. Örneğin 81 -> 1, 344 -> 4 gibi. Elbette aynı sayıya özetlenen birden fazla sayı bulunmaktadır. Bu duruma çakışma (collusion) adı verilmektedir.

Özetleme fonksiyonlarının ingilizcesi olan Hash kelimesinin kökü arapçadan girmiş olan haşhaş kelimesi ile aynıdır. Ve insan üzerinde yapmış olduğu deformasyondan esinlenerek hash function'a giren bilgilere yapmış olduğu deformasyondan dolayı bu ismi almıştır.

Seperate Chanining



Linear Probing

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
separate chaining	N *	N *	N *	3-5 *	3-5 *	3-5 *	no	`equals()`
linear probing	N *	N *	N *	3-5 *	3-5 *	3-5 *	no	`equals()`

* under uniform hashing assumption

** BST SİLME İŞLEMİ: KÖK(N)

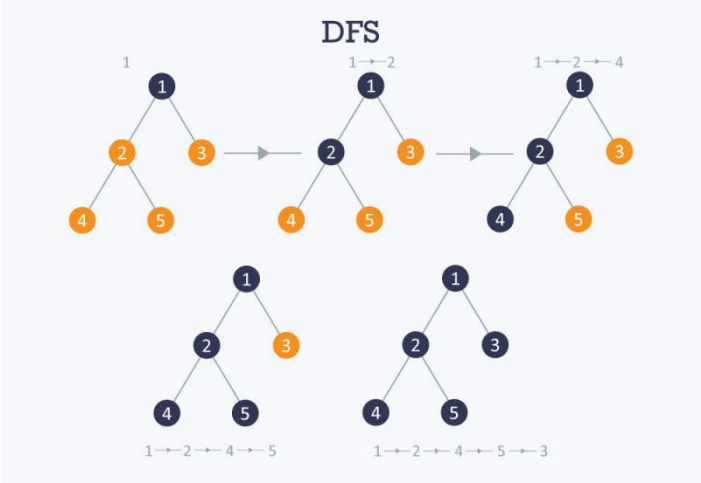
Hash Search Uygulamaları YOK!!!

Ders03 - Undirected Graph (https://www.cs.usfca.edu/~galles/visualization/BFS.html)

Depth First Search (DFS):

DFS gidebildiği yere kadar gitmeyi, daha sonra geri dönerek grafı dolaşmayı hedefler. Yani DFS sürekli olarak komşunun komşusuna gitmeyi amaçlamaktadır. DFS algoritması Stack Veri Yapısı kullanmaktadır.

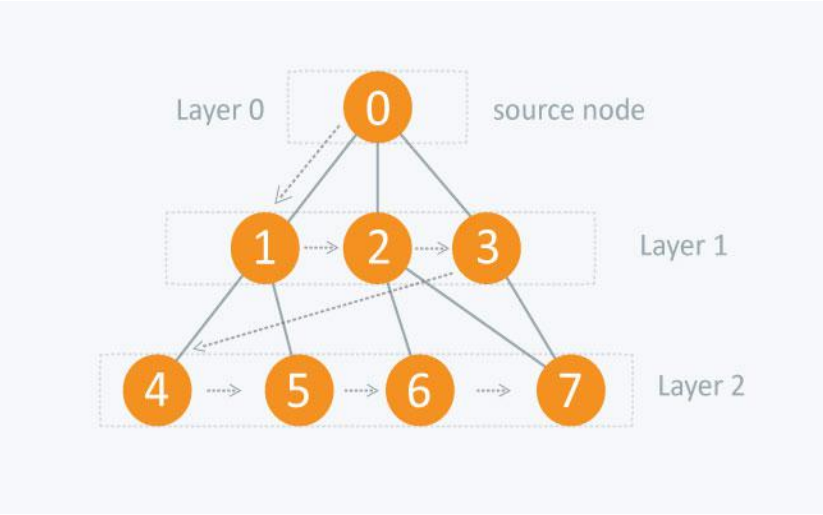
DFS Algoritmasında ziyaret edilen düğümler yığına eklenir. Gidilecek komşu bulunamadığında pop ile yığından eleman atılır ve ziyaret edilebilecek komşu aranır. Aşağıda örnek üzerinde gösterelim.



Breath First Search (BFS):

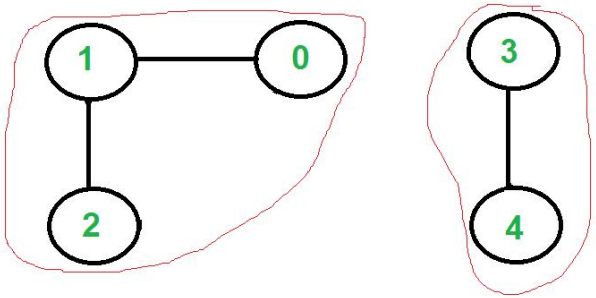
BFS Algoritması temel prensip olarak en yakın düğümleri ziyaret etmeyi hedefler. En yakındaki düğümler taranarak uzaktaki düğümlere gidilir. Yakın olan düğümler öncelikli olarak ziyaret edileceği için bizim bir veri yapısı kullanmamızı gerektirmektedir. BFS Algoritmasında Kuyruk Veri Yapısı Kullanılır.

Ziyaret edilen düğümün komşuları kuyruğa atılır. Her adımda kuyruktan bir eleman çıkartılır ve bu mantıkla ziyaret edilmemiş düğümler kuyruğa eklenir. Bu şekilde tüm düğümler dolaşılır. Eğer bir düğümü arıyorsanız sadece bir if satementı ile işinizi halledebilirsiniz. Tabii ki bu algoritmayı adım adım örnekle anlatacağız.



Connected Component:

Bir graph'ta bulunan bir düğümün diğer tüm düğümlerle bağlantısının olması connected component olduğuna işarettir.



There are two connected components in above undirected graph
0 1 2
3 4

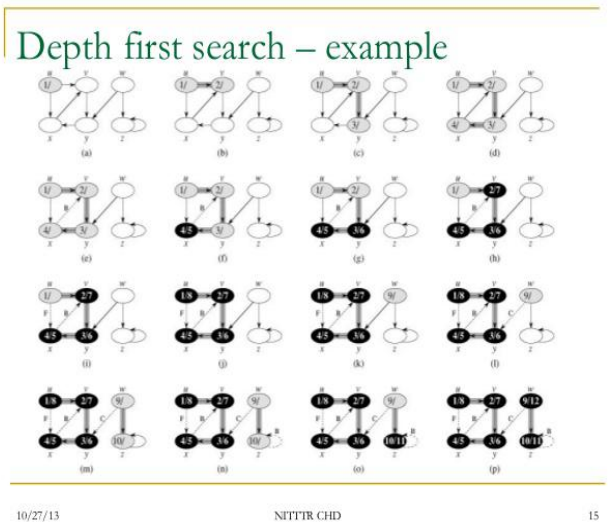
**** Undirected Graph Challenges YOK!!!**

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	E + V	1	degree(v)	degree(v)

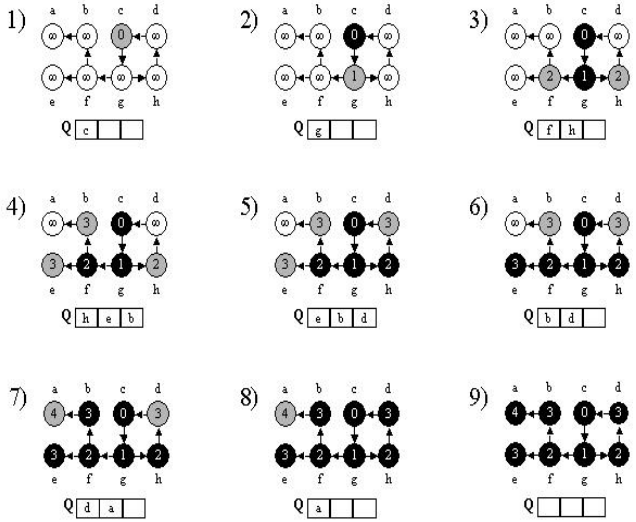
* disallows parallel edges

Ders04 - Directed Graph (https://www.cs.usfca.edu/~galles/visualization/BFS.html, https://www.cs.usfca.edu/~galles/visualization/TopoSortDFS.html**)**

Depth First Search:

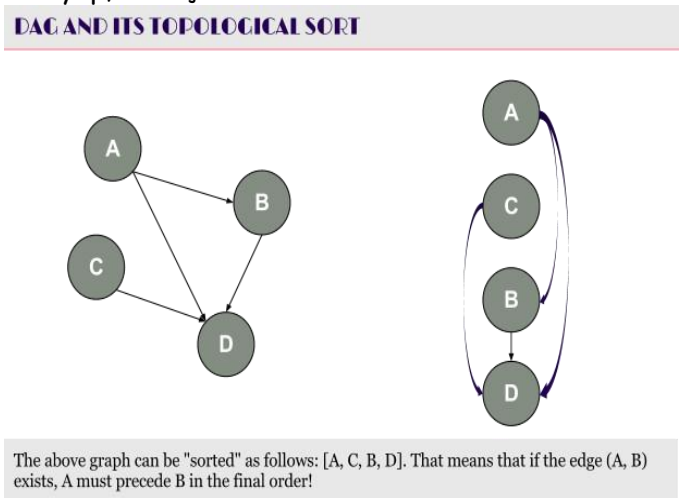


Breath First Search:



Topological Sort:

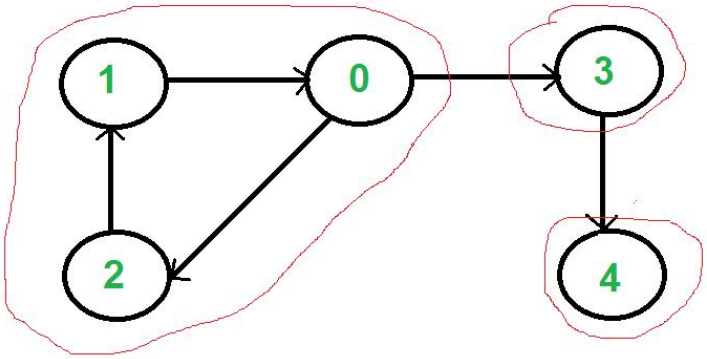
Dfs yap, ters çevir.



Strongly Connected Components:

https://www.cs.usfca.edu/~galles/visualization/ConnectedComponent.html

Bir grafta bulunan bütün düğümleri diğer bütün düğümlere bağlayan birer kenar bulunuyorsa bu grafa güçlü bağlı graf adı verilir.



Kosaraju Algoritması:

Strongly Connected Components doğruluğunu sağlamak için kullanılır.

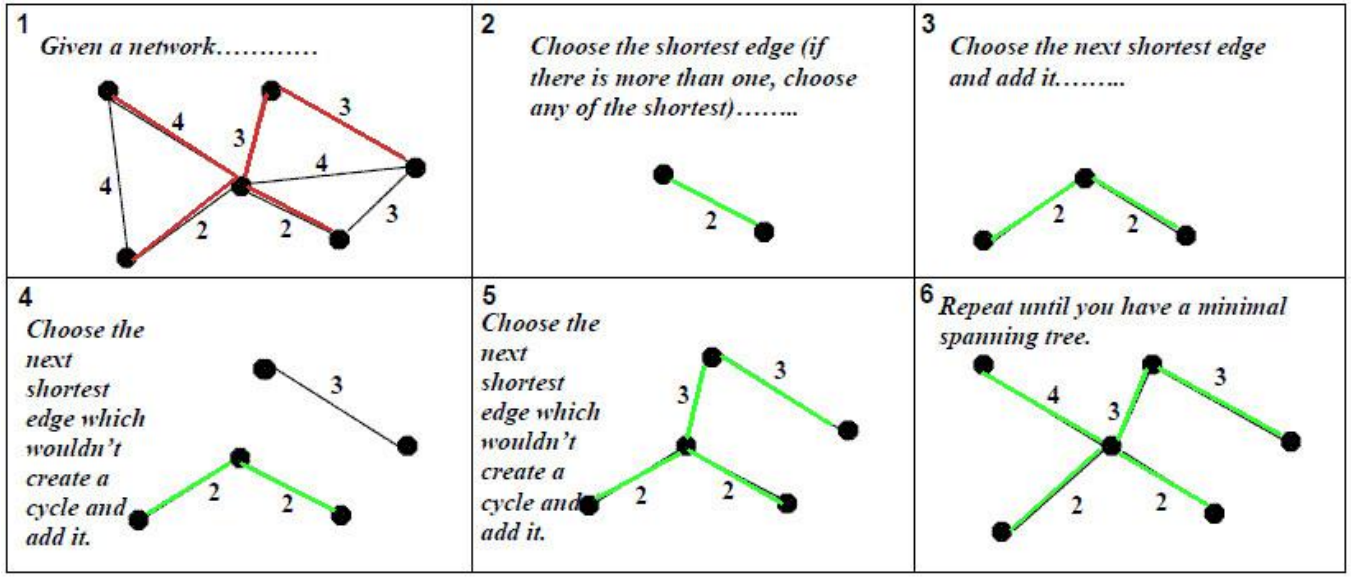
I. Graph'ı ters çevir.
II. DFS Yap.
III. Sonucu tersten yaz.
IV. Normal graph üstünde tekrardan DFS yap.
V. Doğruluğunu kontrol et.

Ders05 - Minimum Spanning Trees (MST)

Kruskal Algoritması:

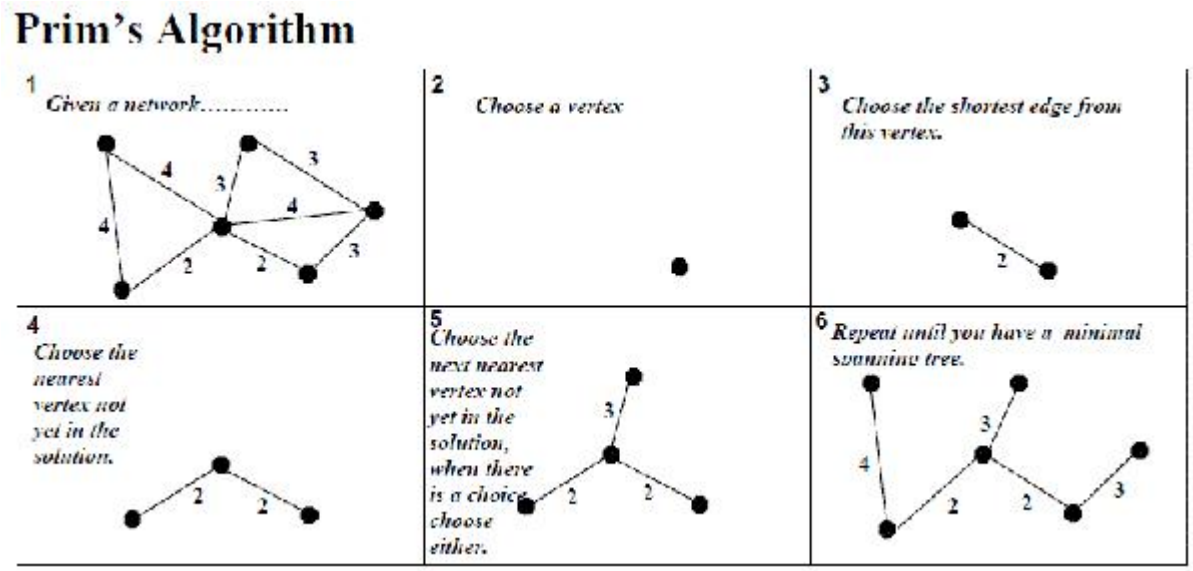
(https://www.cs.usfca.edu/~galles/visualization/Kruskal.html)

Bir asgari tarama ağacı (minimum spanning tree) algoritması olan Kruskal algoritması, küçükten büyüğe doğru sıralanmış düğümleri bünyesine katarak ilerler. Buna göre aşağıdaki grafiğin asgari tarama ağacını çıkaralım:



Prim's Algoritması:
(https://www.cs.usfca.edu/~galles/visualization/Prim.html)

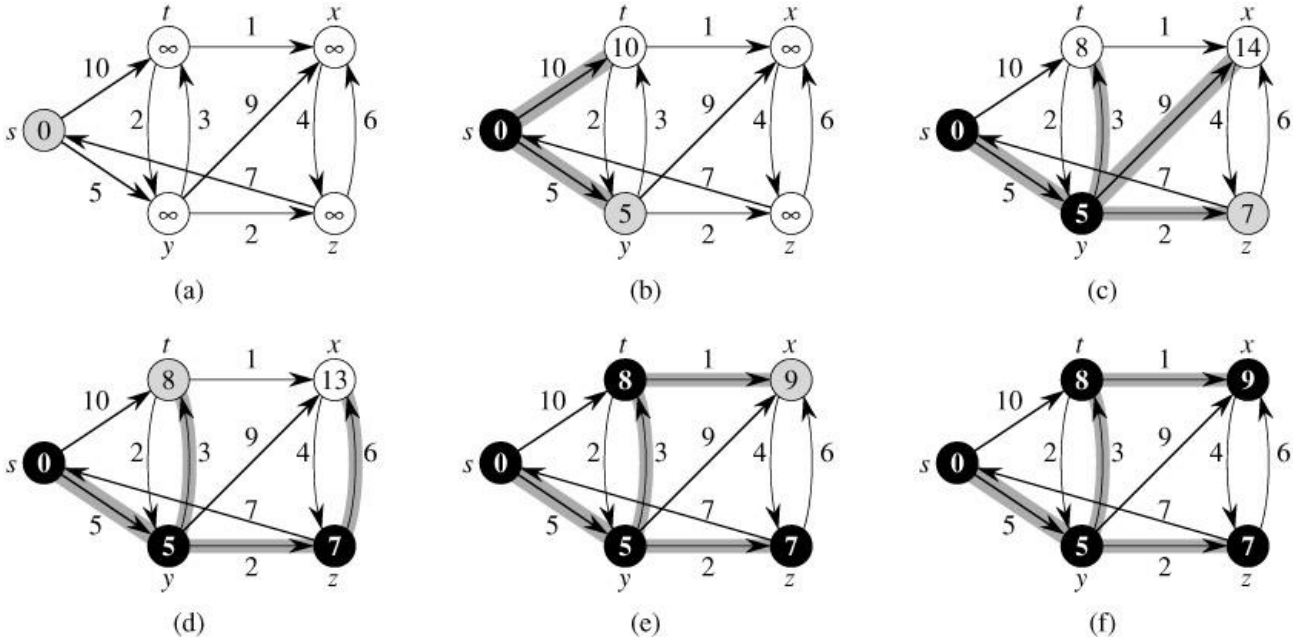
Bir asgari tarama ağacı (minimum spanning tree) algoritması olan Prim algoritması, işaretlemiş olduğu komşuluklara en yakın düğümü bünyesine katarak ilerler. Buna göre aşağıdaki grafiğin asgari tarama ağacını çıkaralım:



Ders06 - Shortest Path

Dijkstra's Algoritması:
(https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html)

a. Başlangıç noktasını belirle.
b. Başlangıç noktasından diğer noktalara olan maliyeti belirle ve düşük maliyetli noktayı işaretle.
c. İkinci adımda işaretlenen noktadan gidilebilen diğer noktalar arasında da aynı işlemi tekrarla.



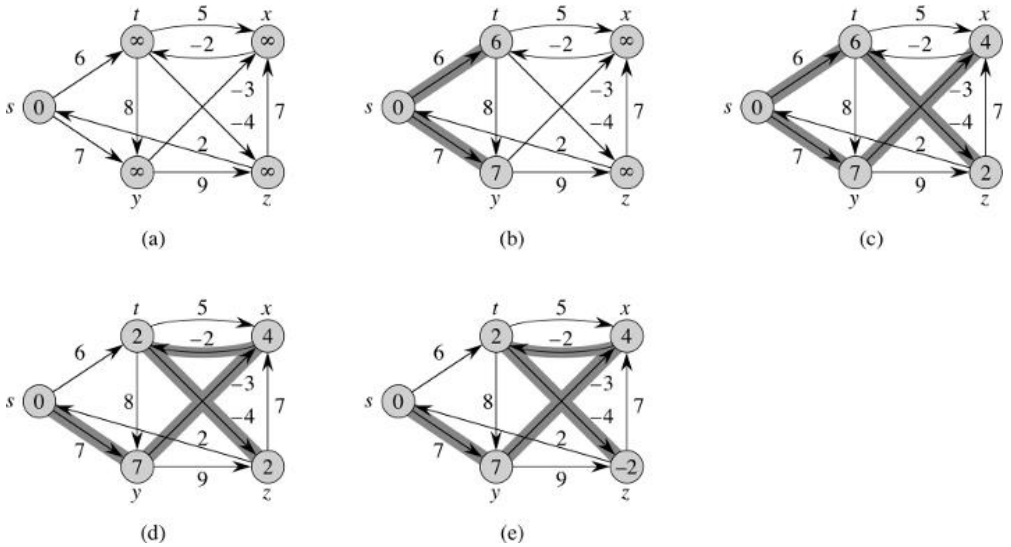
**** Shortest Path Topological Sort Algoritması YOK!!! (https://www.cs.usfca.edu/~galles/visualization/TopoSortDFS.html)**

Bellman - Ford Algoritması:
(http://bilgisayarkavramlari.sadievrenseker.com/2010/08/05/bellman-ford-algoritmasi-2/)

Bu algoritmanın amacı, bir şekil (graph) üzerindeki, bir kaynaktan (source) bir hedefe (target veya sink) giden en kısa yolu bulmaktır. Bu anlamda, literatürde en kısa yol bulma algoritması (shortest path algorithm) olarak sınıflandırılabilir. Algoritma ağırlıklı şekiller (weighted graph) üzerinde çalışır. Kabaca, bütün düğümler için bütün kenarları dolaşır. Dolayısıyla performansı düğüm sayı ile kenar sayısının çarpımı olarak düşünülebilir. O(V*E)

!!! Dijkstra algoritmasında olan ve negatif değerli kenarlardan kaynaklanan döngüye girilme ihtimali, Bellman-Ford algoritmasında bulunmaz.

Algoritma Dijkstra algoritmasında olduğu gibi en küçük değere sahip olan kenardan gitmek yerine bütün graf üzerindeki kenarları test eder.



algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	E + V	E + V	V
Dijkstra (binary heap)	no negative weights	E log V	E log V	V
Bellman-Ford	no negative cycles	E V	E V	V
Bellman-Ford (queue-based)		E + V	E V	V

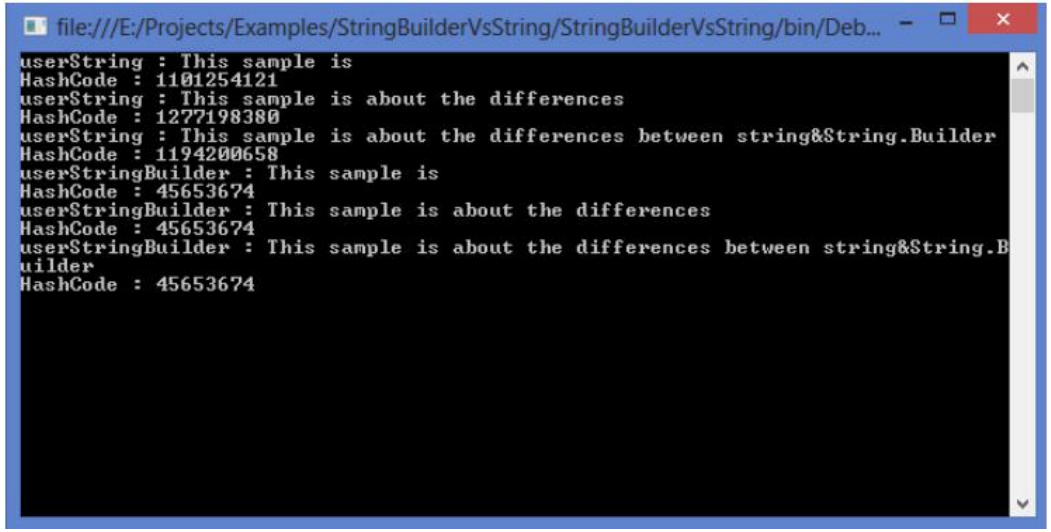
Ders07 - String Sort

String vs String Builder:

Stringler değişmez (immutable) objelerdir. Öyle ki,tanımladığınız bir string'in içeriğini değiştirdiğinizde, o objenin içeriği değişmez. Her ne kadar değişiyor gibi gözükse de, arka planda bellek üzerinde bir kopyası daha oluşturulur. Bu da string'e her değer atamanızda yeni bir instance oluşması anlamına gelmektedir.

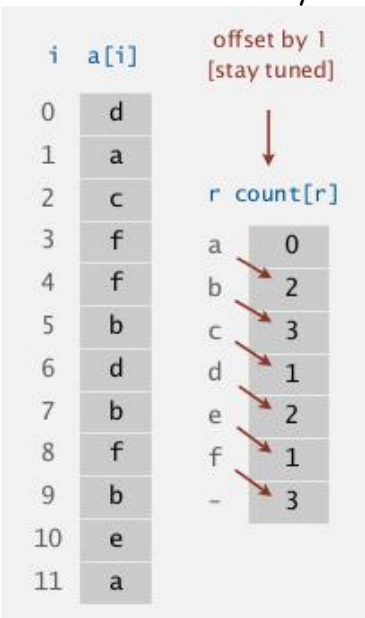
Örneğin uygulamanız içinde bir log dosyası oluşturup, uygulamanın çalışması süresince her adımı log dosyasına yazacağınızı düşünün.Tanımladığınız string'in içeriğini sürekli değiştirerek log dosyasını yazmanız gerekiyor. Böyle bir durumda string kullanmak performans açısından kötü bir seçim olacaktır. Bunun yerine StringBuilder kullanmanız gerekir. StringBuilder sınıfından bir nesne örneği oluşturup, değişiklikleri bu örnek üzerinde yaptığınızda bellek üzerinde kopya bir instance oluşmayacak ve mevcut nesne örneği kullanılacaktır. Aşağıdaki örnekte main methodu içinde string tipinde userString ve StringBuilder tipinde userStringBuilder objeleri tanımlanmıştır. Her iki obje içindeki metinsel ifade değiştirilmiş ve hashCode'ları kontrol edilmiştir.

Programı derlediğinizde userString objesi için her defasında farklı hashCode'lar üretilirken, userStringBuilder objesine ait hashCode'un sabit kaldığını fakat içeriğini değiştiğini göreceksiniz.

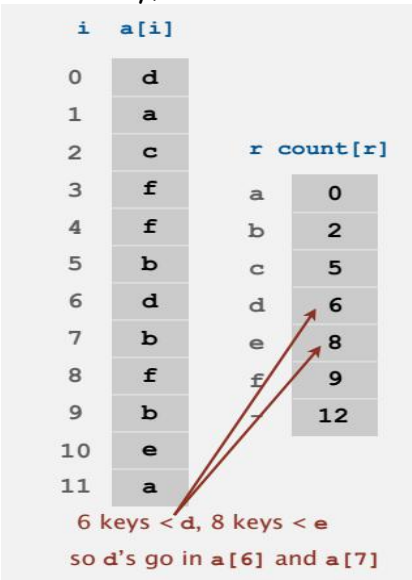


Key - Indexed Sort:

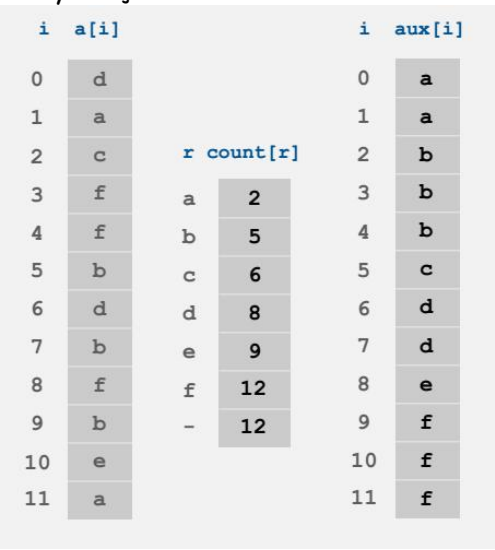
İlk önce bir count array'i oluştur.



Sonra count[r] = count[r]+count[r-1] formülü ile count array, sıralanacak halini almış olur.

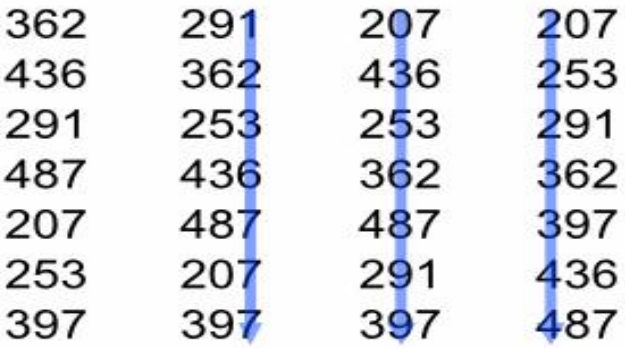


Sonra başlangıç array'indeki sırayla başka bir array'a count sırasına bağlı olarak harfler yerleştirilir. Ve her yerleştirilen harfin count'u 1 arttırılır.



LSD Radix Sort:

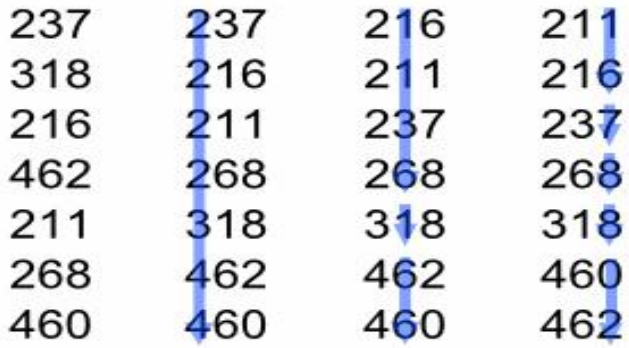
En düşük basamaktan başlayarak sıralama işlemi.



LSD Radix Sorting:
Sort by the last digit, then
by the middle and the first one

MSD Radix Sort:

En yüksek basamaktan başlayarak sıralama işlemi.



MSD Radix Sorting:
Sort by the first digit, then sort
each of the groups by the next digit

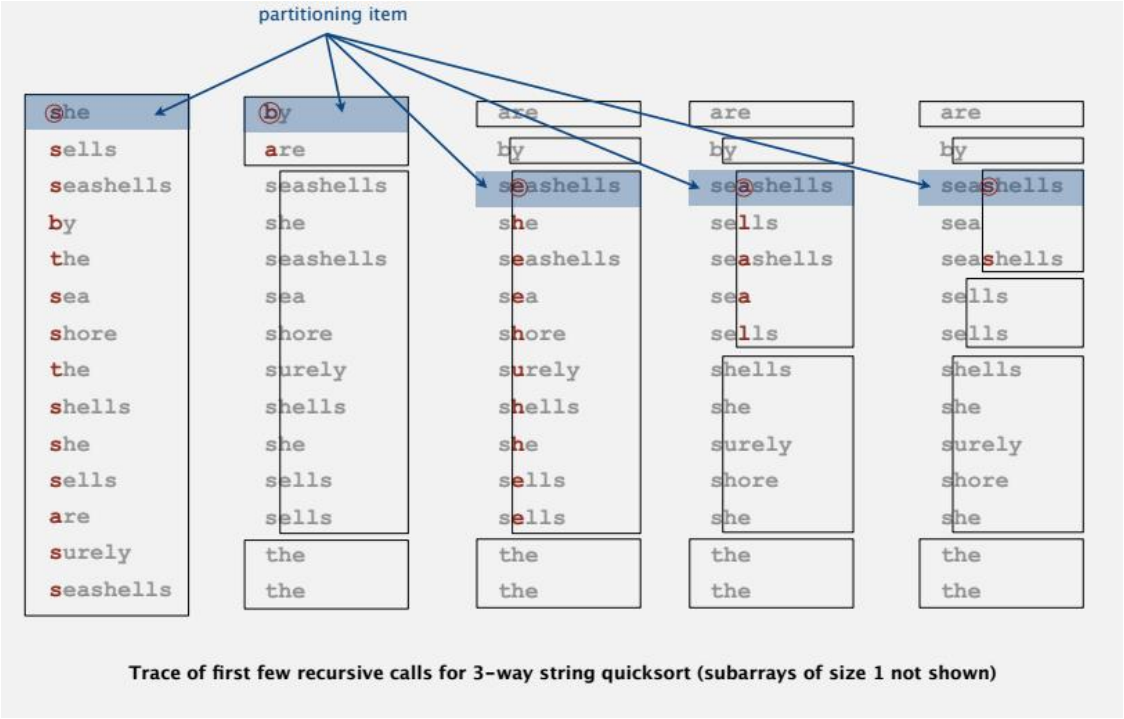
Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	`compareTo()`
mergesort	N lg N	N lg N	N	yes	`compareTo()`
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	`compareTo()`
heapsort	2 N lg N	2 N lg N	1	no	`compareTo()`
LSD †	2 N W	2 N W	N + R	yes	`charAt()`
MSD ‡	2 N W	N $\log_R$ N	N + D R	yes	`charAt()`
3-way string quicksort	1.39 W N lg N *	1.39 N lg N	log N + W	no	`charAt()`

* probabilistic
† fixed-length W keys
‡ average-length W keys

3 Way Radix Quick Sort:

a) İlk string'in ilk elemanına göre sıralama yap.
b) Sonrasında ilk string sıralanana kadar aynı işlemi tekrarla.
c) Sıralandıktan sonra ikinci string'e geç.
d) Recursive olarak bu işlemi tekrarla.



Trace of first few recursive calls for 3-way string quicksort (subarrays of size 1 not shown)

Suffix Sort:

1. Kelimenin başından bir harf eksilterek count array'e yaz.
2. Bu yazılan kelimelere 3 way radix quicksort uygula.
3. Bu işlemi uygularken countları değiştirmeyi unutma!
4. Sıralanma sonucu oluşan listenin count'u sana kelimenin harf sırasını verir ve sıralanma işlemi biter.

Let the given string be "banana".

```
0 banana                    5 a
1 anana     Sort the Suffixes   3 ana
2 nana      ---------------->   1 anana
3 ana        alphabetically     0 banana
4 na                        4 na
5 a                         2 nana
```

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}

time to suffix sort (seconds)

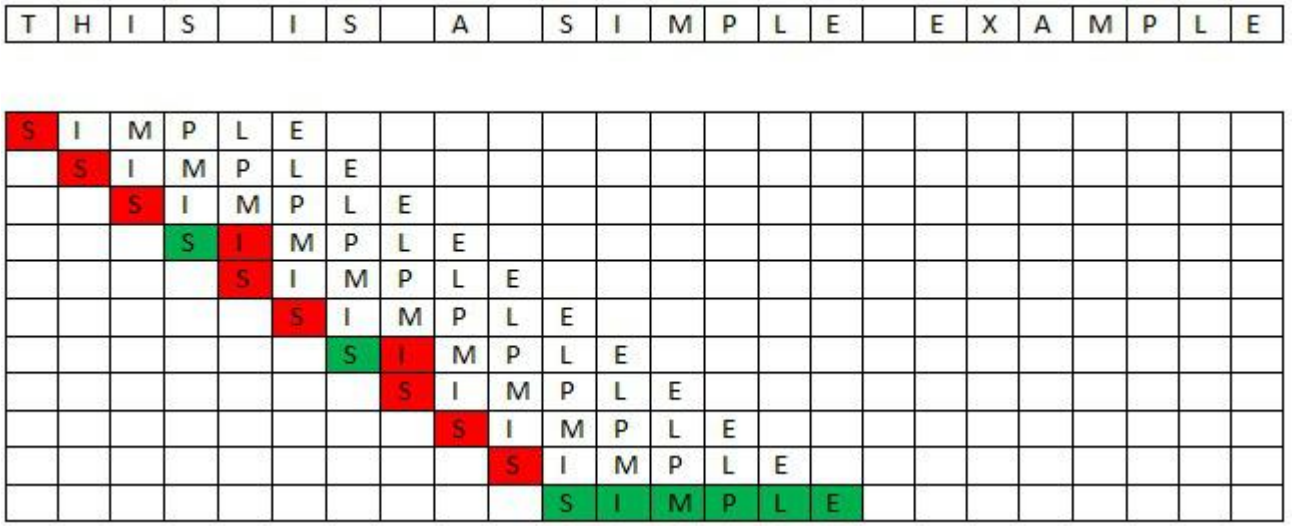
algorithm	mobydick.txt	aesopaesop.txt
brute-force	36.000 †	4000 †
quicksort	9,5	167
LSD	not fixed length	not fixed length
MSD	395	out of memory
MSD with cutoff	6,8	162
3-way string quicksort	2,8	400
Manber MSD	17	8,5

† estimated

Ders08 - Substring Search

Brute-Force Search:

Bilgisayar bilimlerinde bir metnin içerisinde başka bir metnin aranması için kullanılan en ilkel ve dolayısıyla en düşük performanslı arama algoritmasıdır (search algorithm). Algoritma hedef metinde, aranan metni harf harf bulmaya çalışır. Bu yapısından dolayı diziler üzerinde kullanılan doğrusal arama (linear search) algoritmasına oldukça benzer ve literatürde doğrusal metin araması (linear text search) ismi de verilmektedir.



Prefix function

Knoth-Morris-Prat Search:

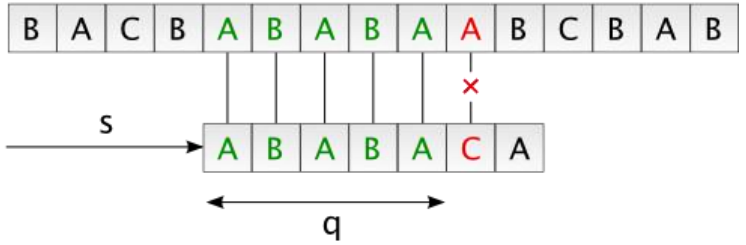
(https://people.ok.ubc.ca/ylucet/DS/KnuthMorrisPratt.html)

Knuth-Morris-Prat algoritması bir kelimenin (yada bir metin parçasının) bir metin içerisinde aranmasını sağlayan algoritmadır. Basitçe bu algoritmada bir kelimenin aranan metinde bakılması ve bakıldığı yerde bulunamaması durumunda nerede olabileceği ile ilgili bir bilginin elde edilmesi hedeflenir.

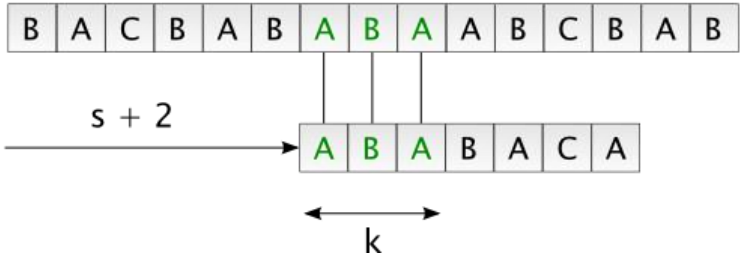
Algoritma aranan kelimenin, aranan metinde bulunmaması durumunda, kelimenin içerisindeki harflerden yola çıkarak birden fazla ihtimali elemektedir.

Klasik bir metin arama işleminde aranan kelime metindeki bütün ihtimallerde denenir. (örneğin doğrusal arama (linear search) bu şekilde çalışır). KMP algoritmasında ise aranan metindeki bütün ihtimaller denenmez. Bu durumu anlamak için bir örnek üzerinden algoritmayı inceleyelim:

1)



2)



Algoritmanın performansı olarak O(n) değeri bulunabilir (n metnin boyutu olarak düşünülürse). Doğrusal arama ile aynı sonucun alındığı algoritma performansı için en kötü durum analizi yapıldığı ve en kötü durumda doğrusal arama ile aynı olacağı unutulmamalıdır.

**** DFA YOKTUR!!!**

Boyer-Moore Search:

(https://people.ok.ubc.ca/ylucet/DS/BoyerMoore.html)

Bir metin veya hedef dizgi (string) içerisinde bir başka dizginin (string) aranması sırasında kullanılan algoritmalardan birisidir. KMP (Knuth Morris Prat) algoritması ile birlikte en çok kullanılan arama algoritmalarındandır.

Bu algoritmadaki amaç bütün harfleri teker teker kontrol eden doğrusal aramadan (linear search) daha iyi bir sonuç elde etmektir.

BM algoritması basitçe aranan metni hedef metin ile eşleştirir. Bulduğu sonuca göre de atlama gerçekleştirir. Ayrıca aranan metne göre de bir atlama tablosu (jump table) tutarak işlemi hızlandırır.

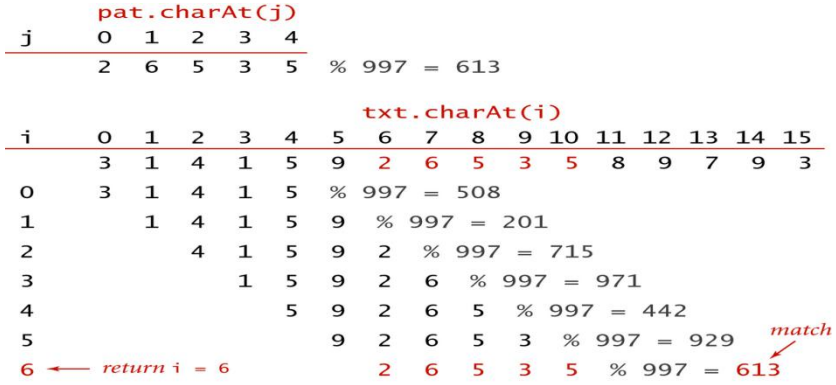
Bu atlama işlemini bir örnek üzerinden anlamak daha kolay olacaktır.

Algoritma performansı doğrusal aramada aranan kelime uzunluğu (a) ile hedef kelime uzunluğu (h) çarpımıdır: ha

Ancak BM algoritaması burada devreye girerek aranan kelimenin bütün harflerinin kontrolünü her seferinde engellemektedir. Bu yüzden algoritma başarısı h olarak indirgenebilir. dolayısıyla doğrusal hıza sahip olunur ve O(n) ile ifade edilebilir.

Rabin-Karp Search:

Hash fonksiyonunun uygulandığı bir arama yöntemi.



Basis for Rabin-Karp substring search

Örnekte; 997'e göre mod alınarak uygulanmıştır.

Cost of searching for an M-character pattern in an N-character text.

algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1\,N$	*yes*	*yes*	1
Knuth-Morris-Pratt	*full DFA (Algorithm 5.6)*	$2N$	$1.1\,N$	*no*	*yes*	MR
	mismatch transitions only	$3N$	$1.1\,N$	*no*	*yes*	M
Boyer-Moore	*full algorithm*	$3N$	N / M	*yes*	*yes*	R
	mismatched char heuristic only (Algorithm 5.7)	MN	N / M	*yes*	*yes*	R
Rabin-Karp†	*Monte Carlo (Algorithm 5.8)*	$7\,N$	$7\,N$	*no*	*yes†*	1
	Las Vegas	$7\,N$ †	$7\,N$	*yes*	*yes*	1

† probabilisitic guarantee, with uniform hash function

Ders09 - Data Compression

Run-Length Encoding (RLE):

Run Length Encoding ya da burada kullanacağımız kısa adıyla RLE, oldukça sık başvurulan, en basit veri sıkıştırma yöntemlerinden birisidir.

Kayıpsız veri sıkıştırma tekniğine dayanan bu yöntemde, birbirini tekrarlayan uzun sembol dizileri, bu sembolün bir örneği ve sembolün kaç kez tekrarlandığı yan yana yazılarak sıkıştırma yapılması amaçlanmıştır. Şöyle ki; bir sembol dizisi, 'L' tane 'S' sembolünün tekrarlanmasından oluşuyor ise, bu diziyi kısaca 'LS' şeklinde yazmak yeterli olacaktır... (Devam)

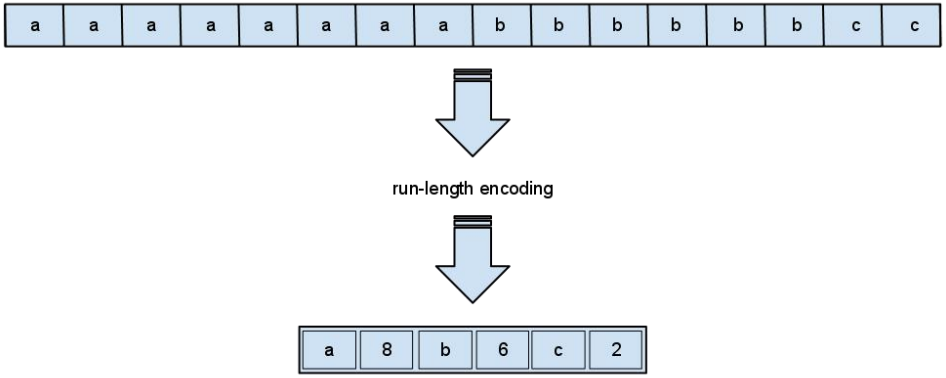
Örnek vermek gerekirse;
AAAAAAAABBBBBCCC
karakter dizisini şöyle gösterebiliriz:
8A5B3C

Bu şekilde bir gösterimle, toplam 16 karakterden oluşan bir diziyi, hiçbir kayıp yaşamadan 6 karakterden oluşan bir alana sıkıştırabiliyoruz. Bu karakterlerin yüzlerce kez tekrar ettiği bir diziyi ele aldığımızda, sıkıştırma oranının ne kadar yüksek olacağını öngörebilirsiniz.

Diğer taraftan, RLE tekniğinin her zaman bu kadar etkili olmayabileceğini de tahmin etmek zor değil.

Karakterlerin tekrar etmediği bir diziyi ele alırsak;
ABCDE
şeklindeki 5 karakterden oluşan dizi,
1A1B1C1D1E

şeklinde 10 karakterden oluşan bir dizi ile gösterilmelidir. Yani dizinin boyutunu %100 oranında artıran bir sıkıştırma uygulamış oluyoruz. Bu ve buna benzer durumlarla karşılaşan kişiler, temeli yukarıda anlattığımız gibi olan RLE tekniğine yeni özellikler katmışlar, yeni öneriler sunmuşlardır. İşte bu yüzden, temeli aynı olan farklı yöntemler geliştirilmiştir.



Hoffman Compression: (http://lti.cs.vt.edu/LTI_ruby/AV/Development/huffmanCustomBuildAV.html)

Bilgisayar bilimlerinde veri sıkıştırmak için kullanılan bir kodlama yöntemidir. Kayıpsız (lossless) olarak veriyi sıkıştırıp tekrar açmak için kullanılır. Huffman kodlamasının en büyük avantajlarından birisi kullanılan karakterlerin frekanslarına göre bir kodlama yapması ve bu sayede sık kullanılan karakterlerin daha az, nadir kullanılan karakterlerin ise daha fazla yer kaplamasını sağlamasıdır.

Şayet bütün karakterlerin dağılımı eşitse yani aynı oranda tekrarlanıyorsa, bu durumda Huffman kodlaması aslında blok sıkıştırma algoritması (örneğin ASCII kodlama) ile aynı başarıya sahiptir. Ancak bu teorik durumun gerçekleşmesi imkansız olduğu için her zaman daha başarılı sonuçlar verir.

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)