



Bölüm 1: Giriş

Veri Yapıları



Dijital Veri (Digital Data)

- Sayı, metin, görüntü ve ses gibi bilgilerin elektronik olarak temsil edilmesi için kullanılan bir formattır.
- **Örnekler:** Metin belgeleri, resim dosyaları, veritabanı, ses kayıtları.





Dijital Verinin Özellikleri

- **Sayısal Olması:** Veriler 0 ve 1 (ikili) olarak temsil edilir.
- **Kolay Saklama:** Elektronik ortamda saklanabilir ve kolayca paylaşılabilir.
- **İşlemeye Uygun:** Bilgisayarlar üzerinde hızlı bir şekilde işlenebilir.





Dijital Verinin Temsil Biçimleri

- **Metin:** ASCII, Unicode gibi karakter kodlamaları.
- **Sayısal:** Tam sayılar, kayan nokta sayıları.
- **Görüntü:** Piksellerin renk değerleri.
- **Ses:** Ses dalgalarının dijital örneklemei.



Dijital Veri Türleri

- **Yapısal:** Belirli bir düzen içinde saklanan veri (XML, JSON).
- **Metin:** İnsanlar tarafından okunabilen metinler.
- **Resim ve Grafik:** Görsel veriler (JPG, PNG).
- **Ses ve Müzik:** Ses kayıtları (MP3, WAV).



Veri Yapıları ve Algoritmalar

- **Algoritma,**
 - Belirli bir görevi gerçekleştirmek için adım adım hazırlanan bir plan veya talimatlar bütünüdür.
 - Bilgisayar biliminde temel bir yapı taşıdır.
- **Veri Yapıları,**
 - Verilerin organize edilmesini sağlayan ve depolanma biçimlerini tanımlayan yapılardır.
 - Verilerin etkili yönetimi ve işlenmesi için temel altyapıyı sağlar.
- Etkili algoritmalar ve veri yapıları, yazılımın verimli çalışmasını sağlar.
- Problemlere sistemli ve etkili çözümler sunar.

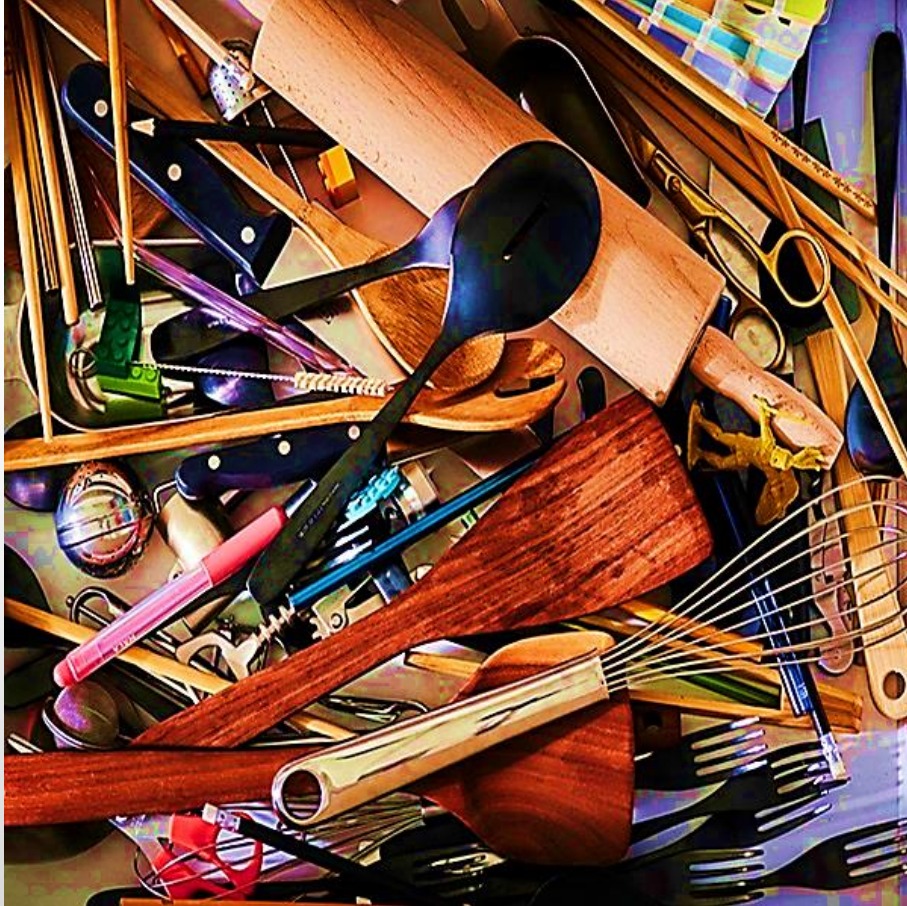


Veri Yapıları

- Verileri bilgisayarda düzenlemenin ve saklamanın bir yoludur.
- Temel amacı, verilere hızlı ve etkili bir şekilde erişmeyi sağlamaktır.
- Veri yapılarını anlamak, yazılım geliştirme sürecinde çok önemlidir.
- Doğru veri yapısı seçimi, programın performans ve verimliliğini artırır.



Veri Yapıları





Temel Yapı Taşları

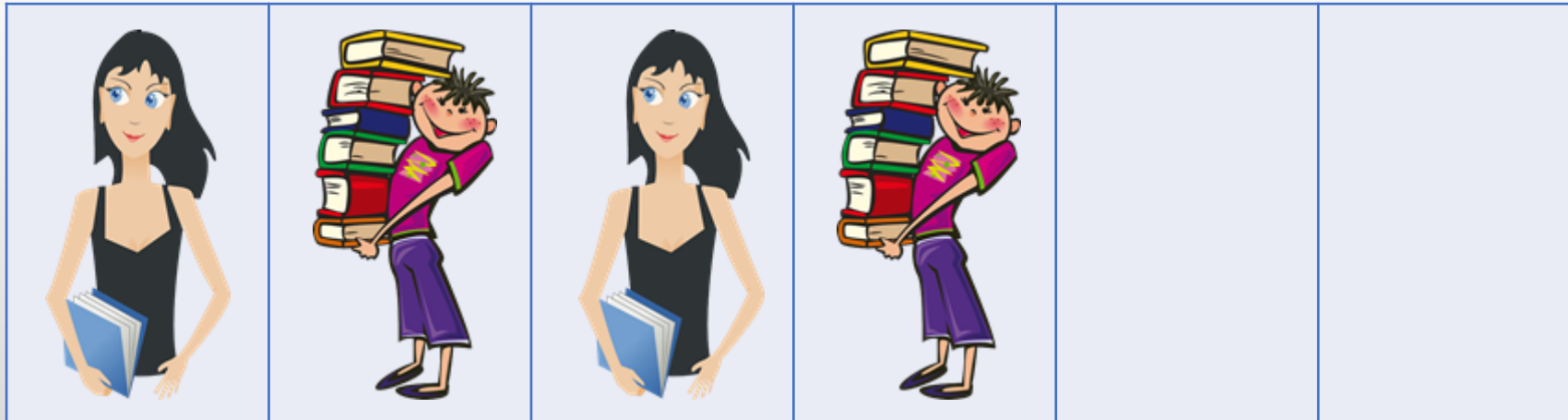
- Nesnelere: Özellikleri, isimlendirilmiş alanlara sahip tek bir birimde gruplar.
- Diziler: Varlıkları, bellekte ardışık yuvalarda toplu olarak saklar.
- Bağlantılar: Varlıklar arasında isteğe bağlı yönlü ilişkilere olanak tanır.





Temel Yapı Taşları

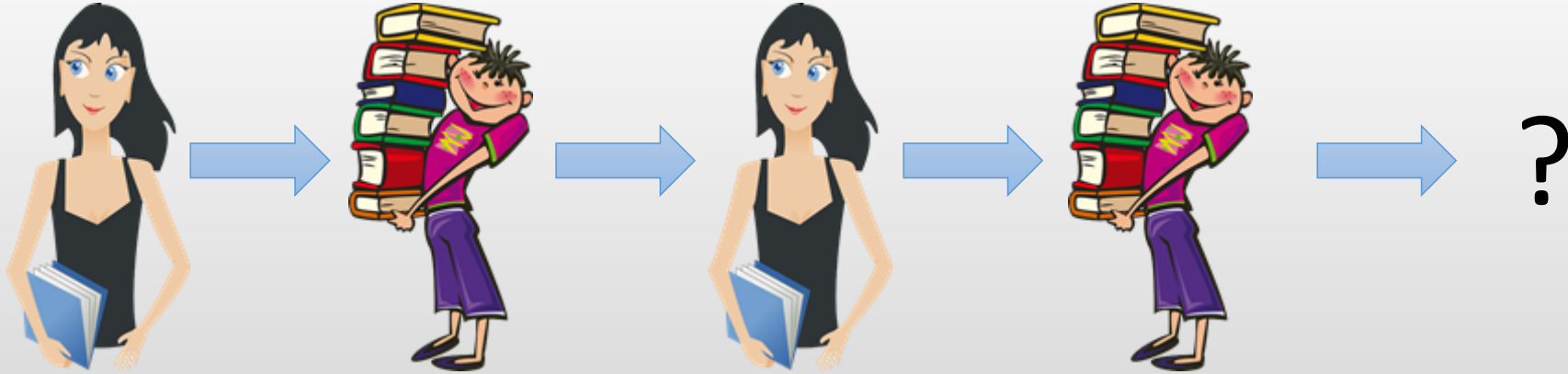
- Nesnelere: Özellikleri, isimlendirilmiş alanlara sahip tek bir birimde gruplar.
- Diziler: Varlıkları, bellekte ardışık yuvalarda toplu olarak saklar.
- Bağlantılar: Varlıklar arasında isteğe bağlı yönlü ilişkilere olanak tanır.





Temel Yapı Taşları

- Nesnelar: Özellikleri, isimlendirilmiş alanlara sahip tek bir birimde gruplar.
- Diziler: Varlıkları, bellekte ardışık yuvalarda toplu olarak saklar.
- Bağlantılar: Varlıklar arasında isteğe bağlı yönlü ilişkilere olanak tanır.





İlkel (Primitive) Veri Tipleri

- Temel veri tipleri:
 - Tam Sayılar (int, short, long, byte)
 - Ondalıklı Sayılar (float, double)
 - Mantıksal Değerler (boolean)
 - Karakterler (char)
- Basit ve temel veri tipleridir.
- Bellekte sabit bir boyuta sahiptirler.
- Doğrudan işleme yetenekleri vardır.



Veri Tipleri

- **Veri Tipi:** Bir deęer setini ve davranışlarını tanımlayan kümeyi ifade eder.
- **Deęer Seti:** Bir veri türündeki geçerli deęerlerin kümesi.
- **Davranışlar:** Deęerler üzerinde gerçekleştirilebilecek işlemler.
- Her işlem her türde geçerli değildir.
- **Tam Sayılar:** Toplama, çıkarma, çarpma, bölme gibi işlemler.
- **Metin Dizileri:** Karşılaştırma, birleştirme, alt dize alma gibi işlemler.
- **Tip Güvenlięi:** Veri türleri, hataları en aza indirmek için kullanılır.
- Kısıtlamalar, programcılara neyin mümkün, mümkün olmadığını söyler.



İlkel Olmayan (Non-Primitive) Veri Yapıları

- İlkel olmayan veri yapıları, daha karmaşık veri yapılarıdır. Örneğin:
 - **Sıralı:** Diziler, listeler
 - **Hiyerarşik:** Ağaçlar, çizgeler
 - **Bağlantılı:** Bağlı listeler
 - **İndekslenmiş:** Hash tabloları
- Karmaşıktır ve birden fazla ilkel veri tipini içerebilir.
- Dinamik boyuta sahiptirler, yani veri miktarı değişebilir.
- Özelleştirilmiş işlemleri gerçekleştirmek için kullanılırlar.



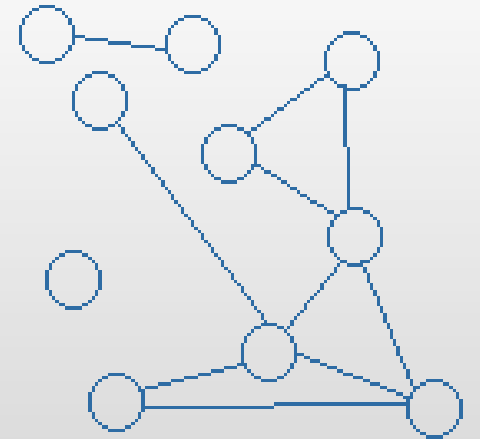
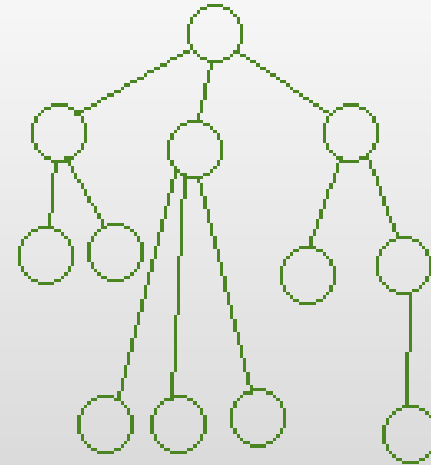
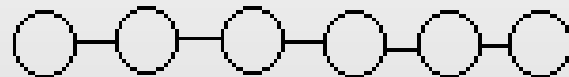
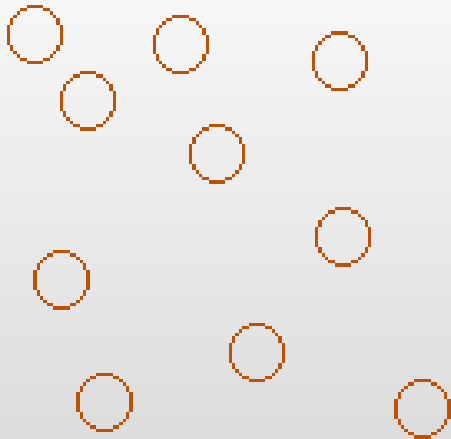
Farklı Veri Yapıları

- Her birinin kendine özgü güçlü ve zayıf yönleri bulunmaktadır.
- **Diziler (Arrays):** Verileri düzenlemek ve erişmek için kullanılır. Bellek kullanımı sabit boyutludur.
- **Bağlı Listeler (Linked Lists):** Verileri bağlantılı bir şekilde saklamak için kullanılır. Esnek boyutludur.
- **Yığınlar (Stacks):** Last in, first out (LIFO) mantığına dayalı olarak çalışır. Örneğin, geri alma işlemi bu yapıyla uygulanabilir.
- **Kuyruklar (Queues):** İlk giren, ilk çıkar (FIFO) mantığına dayalı olarak çalışır. İşlem sıralaması için kullanılır.
- **Ağaçlar (Trees):** Hiyerarşik verileri temsil etmek için kullanılır. Örneğin, ağaç yapıları veritabanı indekslerinde kullanılabilir.



Verilerin Düzenlenmesi

- Konumsal olmayan: ilişkisiz (sınıftaki öğrenciler)
- Sıralı: bire-bir (sıra bekleyen öğrenciler)
- Ağaç: bire-birçok (aile ağacı)
- Çizge: birçoğa-birçok (yol haritası)





Veri Yapısı Seçimi

- Veri yapısı seçiminde aşağıdaki sorular kritik bir rol oynar.
 - Veri yapısı başlangıçta tamamen doldurulabilir mi?
 - Veri yapısından öğeler silinebilir mi?
 - Ekleme sırasının önemi var mı?
 - Veri yapısındaki öğeler sıralanabilir mi?
 - Öğeler belirli bir sıra ile mi işlenir, yoksa rastgele erişim de gerekli mi?
- Veri yapısının başlangıçtaki durumu değişmeyecekse, daha statik yapılar tercih edilebilir.
- Silme ve ekleme işlemleri sıkça gerçekleşiyorsa, dinamik ve esnek veri yapıları düşünülmelidir.



Algoritmalar

- Bir algoritma, bir problemi çözmek için adım adım bir prosedürdür.
- Aynı problemi çözmek için farklı algoritmalar bulunabilir ve bazı algoritmalar diğerlerine göre daha verimli çalışabilir.
- Algoritmalar, bilgisayar bilimlerinin temel bir parçasıdır.
- Doğru algoritma seçimi, bir problemi verimli bir şekilde çözmek için kritik öneme sahiptir.
- İyi bir algoritma, bir işlemi daha az kaynak kullanarak gerçekleştirebilir.



Farklı Algoritmalar

- Bazı yaygın algoritmalar:
- **Sıralama Algoritmaları:** Verileri belirli bir sıraya göre düzenlemek için kullanılır. Örneğin, kabarcık sıralama veya birleştirme sıralaması.
- **Arama Algoritmaları:** Belirli bir öğeyi veri kümesinde bulmak için kullanılır. Örneğin, ikili arama veya lineer arama.
- **Çizge Algoritmaları:** Çizge teorisi problemlerini çözmek için kullanılır. Örneğin, en kısa yol bulma veya ağ akışı problemleri.



İyi Bir Algoritma

- Bir problemi etkili bir şekilde çözen algoritmadır.
- **Zaman Karmaşıklığı (Time Complexity):** Algoritmanın çalışma süresi veya işlem sayısı gibi faktörler, zaman karmaşıklığını belirler.
- **Alan Karmaşıklığı (Space Complexity):** Algoritma tarafından kullanılan bellek miktarı, alan karmaşıklığını belirler.
- **Doğruluk (Accuracy):** Algoritmanın istenen sonuçları doğru bir şekilde üretmesi önemlidir. Yanlış sonuçlar veren bir algoritma kullanışsızdır.
- **Sağlamlık (Robustness):** Algoritmanın çeşitli durumlar ve girdilerle başa çıkabilme yeteneği, sağlamlığını belirler. İyi bir algoritma, farklı senaryolara uyum sağlayabilmelidir.



Zaman Karmaşıklığı

- Algoritmanın çalışma süresinin ne kadar sürdüğünü ölçen bir ölçüdür.
- Zaman karmaşıklığı, Büyük O (Big O) gösterimi kullanılarak ifade edilir.
- Algoritma çalışma süresinin, girdi boyutuna bağlı olarak değişimini gösterir.
- Algoritmanın girdi boyuna bağlı olarak ne kadar hızlı veya yavaş çalışacağını anlamamıza yardımcı olur.



Büyük O (Big O) Gösterimi

- Algoritmanın en kötü durumda çalışma süresini ifade eder.
- Algoritmanın çalışma süresinin girdi boyutuna göre nasıl büyüdüğünü belirtir.
- Örneğin,
 - **O(1)** sabit zaman karmaşıklığına sahip bir algoritma, girdinin boyutundan bağımsız olarak aynı sürede çalışırken,
 - **O(n)** karmaşıklığına sahip bir algoritma, girdi boyutu arttıkça doğrusal olarak daha fazla sürede çalışır.



Büyük (O) Gösterimi Örnekleri:

- $O(1)$: Sabit zaman karmaşıklığı.
- $O(\log n)$: Logaritmik zaman karmaşıklığı.
- $O(n)$: Doğrusal zaman karmaşıklığı.
- $O(n \log n)$: Doğrusal logaritmik zaman karmaşıklığı.
- $O(n^2)$: Kare zaman karmaşıklığı.
- $O(2^n)$: Üstel zaman karmaşıklığı.



Büyük (O) Gösterimi Örnekleri:

- **$O(1)$** : Sabit Zaman Karmaşıklığı
 - **Örnek**: Bir dizinin ilk elemanına erişme.
 - Dizinin boyutu ne olursa olsun, erişim süresi sabittir.
- **$O(\log n)$** : Logaritmik Zaman Karmaşıklığı
 - **Örnek**: Sıralı bir listede ikili arama yapma.
 - Listenin boyutu arttıkça, arama süresi logaritmik olarak artar.
- **$O(n)$** : Doğrusal Zaman Karmaşıklığı
 - **Örnek**: Bir diziyi baştan sona tarama.
 - Listenin boyutu ile doğru orantılı olarak artan bir süreye sahiptir.



Büyük (O) Gösterimi Örnekleri:

- **$O(n \log n)$:** Doğrusal Logaritmik Zaman Karmaşıklığı
 - **Örnek:** Hızlı sıralama (Quick Sort) algoritması.
 - Genellikle hızlı sıralama gibi verileri bölüp sıralamak için kullanılan algoritmaların karmaşıklığıdır.
- **$O(n^2)$:** Kare Zaman Karmaşıklığı
 - **Örnek:** İç içe döngülerle bir matrisi tarama.
 - İç içe iki döngü kullanılarak, her elemanın diğer tüm elemanlarla karşılaştırıldığı durumda elde edilen karmaşıklıktaır.
- **$O(2^n)$:** Üssel Zaman Karmaşıklığı
 - **Örnek:** Tüm alt kümeleri bulma.
 - Kümelerin alt kümelerini bulmak gibi her bir adımda iki kat artan bir karmaşıklığa sahiptir.



Alan Karmaşıklığı

- Algoritmanın kullandığı bellek miktarını ölçen bir ölçüdür.
- Genellikle Büyük O (Big O) gösterimi kullanılarak ifade edilir.
- Algoritmanın bellek kullanımının, girdi boyutuna bağlı artışını gösterir.
- Sınırlı bellek kaynaklarına sahip sistemlerde bellek verimliliği önemlidir.



Büyük O Gösterimi Örnekleri

- **$O(1)$** : Sabit Alan Karmaşıklığı
 - **Örnek**: Bir değişken oluşturma.
 - Bellek kullanımı sabittir, girdi boyutuyla değişmez.
- **$O(n)$** : Doğrusal Alan Karmaşıklığı
 - **Örnek**: Bir dizinin tüm elemanlarını saklama.
 - Bellek kullanımı, girdi boyutu ile doğru orantılı olarak artar.
- **$O(n^2)$** : Kare Alan Karmaşıklığı
 - **Örnek**: İki boyutlu bir matrisi saklama.
 - Bellek kullanımı, girdi boyutunun karesi ile orantılı olarak artar.



Doğruluk

- Doğruluk, bir algoritmanın başarısını belirleyen önemli bir faktördür.
- Algoritma çıktısının doğru cevaba yakınlığını ölçen bir ölçüdür.
- Çıktı ile gerçek cevap arasındaki benzerlik derecesini ifade eder.
- Doğru sonuçlar üreten algoritmalar, güvenilir ve güvenli yazılım geliştirme, veri analizi, yapay zeka ve diğer birçok alanda kritik bir rol oynar.



Doğruluğun Ölçülmesi

- Algoritmanın ürettiği sonuçlar, gerçek verilere veya bilinen doğru sonuçlar ile karşılaştırılır. Örnek metrikler:
- **Hata Oranı (Error Rate):** Yanlış sonuçlarının oranı.
- **Doğruluk (Accuracy):** Doğru sonuçların oranı.
- **Hassasiyet (Precision):** Pozitif olarak tahmin edilen sonuçların, gerçek pozitif sonuçlara oranı.
- **Duyarlılık (Recall):** Gerçekten pozitif olan sonuçların, pozitif olarak tahmin edilen sonuçlara oranı.



Sağlamlık

- Algoritmanın beklenmeyen durumlara başa çıkma yeteneğini ölçer.
- Sağlam bir algoritma, normal aralığının dışındaki girdileri bile çökmeden veya yanlış sonuçlar üretmeden işleyebilir.
- Gerçek dünyada karşılaşılabilecek her türlü durumu ele alabilir.
- Beklenmedik hatalar, eksik veya bozuk veriler, aşırı yüklenmeler ve diğer olası sorunlar, bir algoritmanın sağlamlığını test eder.



Sağlamlığın Ölçülmesi

- Algoritmanın beklenmeyen girdilere nasıl tepki verdiğini anlamayı içerir:
- **Geçersiz Girdiler (Invalid Inputs):** Uygun olmayan veya beklenmeyen girdilerle başa çıkma yeteneği.
- **Hatalı Veriler (Corrupted Data):** Bozuk veya hatalı verilerle başa çıkma yeteneği.
- **Büyük Veri Kümesi (Large Data Sets):** Büyük veri kümesini işleme yeteneği.
- **Aşırı Yüklenme (Overload):** Algoritmanın aşırı yüklendiğinde nasıl davrandığı.



Diziler (Arrays)

- Verileri ardışık olarak bir bellek bloğunda saklar.
- Verilere kolay erişim sağlar.
- Veriler, dizi içerisinde eklenme sırasıyla depolanır.
- Her bir veri öğesine bir indeksle erişilebilir.
- Diziler, verileri düzenli bir şekilde saklamak ve hızlı erişim sağlamak için kullanışlı bir veri yapısıdır.



Diziler (Arrays)

0	1	2	3	4
7	2	9	5	1

- Üst sıra dizinin indislerini temsil eder (0, 1, 2, 3, 4).
- Alt sıra bu indislerdeki dizi öğelerinin değerlerini temsil eder (7, 2, 9, 5, 1).



Diziler (Arrays)

- Avantajları
 - Doğrudan indeksleme kullanarak verilere hızlı erişim sağlar.
 - Verileri düzenli bir şekilde saklar.
 - Kolay bir kullanım sunar.
- Dezavantajları
 - Sabit boyuta sahiptir, veri boyutu dinamik olarak değiştirilemez.
 - Eşit olarak dağılmayan veriler, dizinin sonunda veya başında sıklaşabilir.

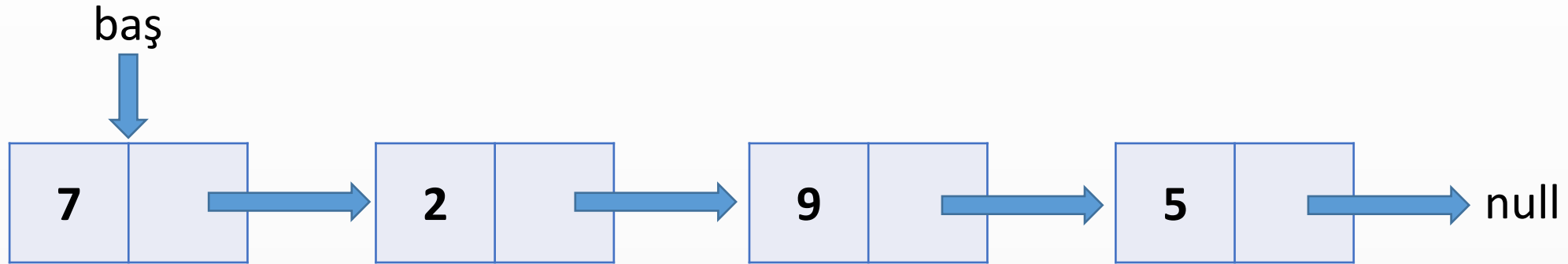


Bağlı Listeler (Linked Lists)

- Verileri düğümler adı verilen bağlı bir liste içinde saklar.
- Her düğüm, veriyi ve bir sonraki düğümün referansını içerir.
- Bu referanslar, verilerin bellekte ardışık saklanmadığı anlamına gelir.
- Bağlı listeler, verilerin dinamik olarak büyüdüğü veya dağılımı dengesiz olduğu durumlarda kullanışlı bir veri yapısıdır.
- Ancak, erişim süreleri dizilere göre daha yavaştır ve daha fazla bellek kullanır.



Bağlı Listeler (Linked Lists)



- "Baş" bağlı listenin başlangıç noktasını gösterir.
- Her düğüm, bir değeri (örneğin, 7, 2, 9, 5) içeren kutu olarak temsil edilir.
- Her düğüm, bir sonraki düğüme doğru bir ok ile bağlanır.



Bağlı Listeler (Linked Lists)

- Avantajları
 - Veri boyutu dinamik olarak değişebilir.
 - Düğümler listeye eklenebilir, listeden çıkarılabilir.
 - Düğümler rastgele yerleştirilebildiği için veri dağılımı sorunlarına daha dayanıklıdır.
- Dezavantajları
 - Verilere erişim, bağlı düğümler arasında gezinme gerektirdiği için dizilere göre daha yavaştır.
 - Her düğüm, veri ve bir sonraki düğümün referansını içerdiğinden, daha fazla bellek alanı kullanılır.



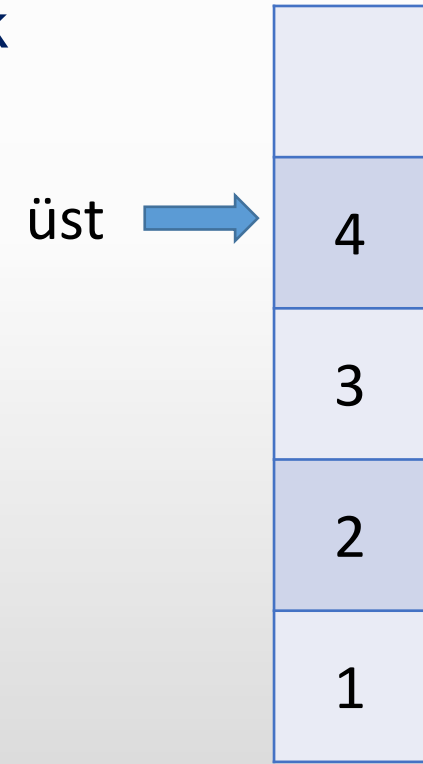
Yığınlar (Stacks)

- Verileri son giren, ilk çıkar (LIFO - Last-In, First-Out) mantığına göre saklar.
- Yeni öge ekleneceğinde yığının en üstüne eklenir.
- Bir öge çıkarılacağında en son eklenen öge çıkarılır.
- Özyinelemeli fonksiyonlar, geri izleme ve işlem geçmişi yönetimi gibi çeşitli alanlarda kullanılırlar.



Yığınlar (Stacks)

- "Üst" yığının üstünü veya en son eklenen öğeyi gösterir.
- Her öğe bir değeri (örneğin, 1, 2, 3, 4) içeren bir kutu olarak temsil edilir.





Yığınların Kullanım Alanları

- **Özyinelemeli Fonksiyonlar (Recursion):** Fonksiyonlar kendi kendini çağırdığında, her çağrı bir yığın çerçevesi olarak saklanır. Bu sayede, fonksiyon tamamlandığında çağrıldığı yere geri dönebilir.
- **Geri İzleme (Backtracking):** Problemi çözmek için olasılıklar denendikten sonra geri adım atılması gerektiğinde yığınlar kullanılır.
- **İşlem Geçmişi (Undo/Redo):** Bir uygulamada kullanıcı işlemlerini geri almak (Undo) veya geri getirmek (Redo) için yığınlar kullanılabilir.

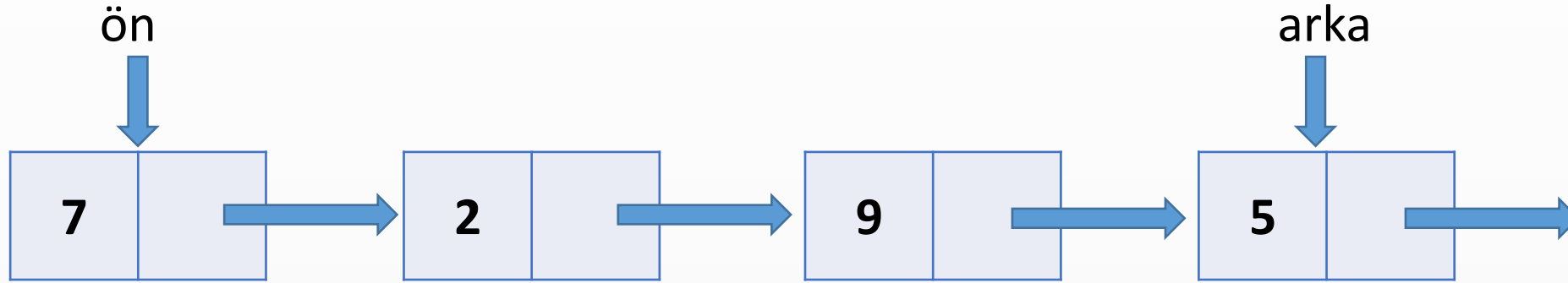


Kuyruklar (Queues)

- Kuyruk (Queue), verileri ilk giren, ilk çıkar (FIFO - First-In, First-Out) mantığına göre saklar.
- Kuyruğa en önce eklenen öge en önce çıkarılır.
- Görev çizelgeleme, istek işleme gibi algoritmaların uygulanmasında kullanılırlar.



Kuyruklar (Queues)



- "Ön" kuyruğun önünü veya kuyruğa en önce eklenen öğeyi gösterir.
- "Arka" kuyruğun arka kısmını veya en son eklenen öğeyi gösterir.
- Her öge, bir değeri (örneğin, 1, 2, 3) içeren bir kutu olarak temsil edilir.
- Her öge, bir sonraki öğeye doğru ok ile bağlanır.



Kuyrukların Kullanım Alanları

- **Görev çizelgeleme (Task Scheduling):** İşlemcinin görevleri sırayla çalıştırmasına yardımcı olur. Kuyruğa ilk giren görev ilk olarak çalıştırılır.
- **İstek İşleme (Request Processing):** Sunucular, gelen istekleri kuyrukta sıraya alır ve sırayla işler. Talep yükünü yönetmek için kullanışlıdır.
- **Veri Yapıları (Data Structures):** Bazı algoritmaların uygulanmasında verileri işlemek için kullanılır. Örneğin, genişlik öncelikli arama (Breadth-First Search) algoritması bir kuyruk kullanır.



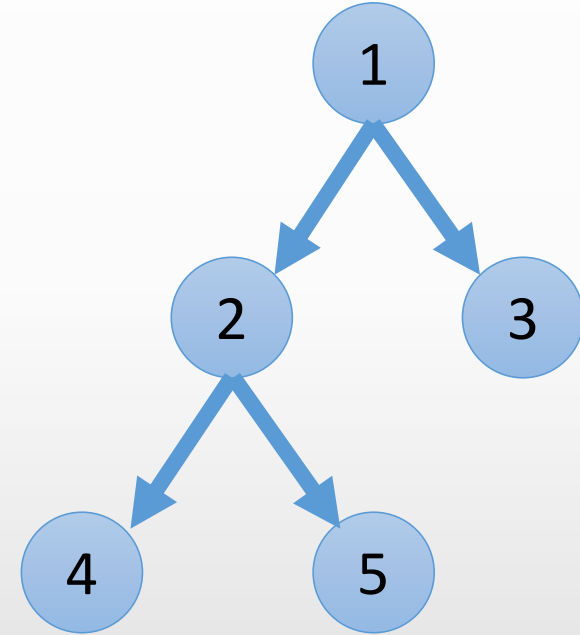
Ağaçlar (Trees)

- Verileri hiyerarşik bir yapı içinde saklar.
- Her ağaç, bir kök düğüm (root node) ve bu kök düğümden dallanmış alt düğümlere sahiptir.
- Her düğüm, kendisine bağlı alt düğümlere sahiptir ve bu şekilde bir hiyerarşi oluşturulur.
- Dosya sistemleri, biyolojik taksonomi ve veritabanı indekslemesi gibi birçok alanda yaygın olarak kullanılırlar.
- Verilerin doğal hiyerarşisini yansıtmak için mükemmel bir yapıdır.



Ağaçlar (Trees)

- Düğüm 1, ağacın kökünü temsil eder.
- Kökün iki çocuğu vardır, 2 ve 3.
- 2 düğümünün ise iki çocuğu vardır, 4 ve 5.





Ağaçların Kullanım Alanları

- **Dosya Sistemi (File System):** Kök düğüm ana dizini temsil ederken, alt düğümler alt klasörleri veya dosyaları temsil eder.
- **Biyolojik Taksonomi (Taxonomy):** Canlı organizmaların sınıflandırılması için biyologlar tarafından kullanılır. Her takson (tür, familya, cins vb.) ağaç yapısı içinde temsil edilir.
- **Veritabanı İndeksleri (Database Indexing):** Hızlı veri erişimi sağlamak için kullanılır. İndeks ağaçları, verileri sıralar ve erişim hızını artırır.



Harita Veri Yapısı (Maps)

- Anahtar-değer çiftlerini saklar.
- Her anahtar, bir değerle eşleştirilir.
- Anahtarlar benzersiz, tekildir ve her biri yalnızca bir değeri temsil eder.
- Anahtar kullanarak değerlere hızlı erişim sağlar.
- Büyük veri koleksiyonlarını etkili bir şekilde işlemek için kullanışlıdır.



Harita Veri Yapısı (Maps)

Anahtar	Değer
Ad	Ali
Soyadı	Veli
Yaş	30
Şehir	İstanbul

- "Anahtar" bölümü, her bir anahtarı içerir ("Ad", "Soyadı", "Yaş" gibi).
- "Değer" bölümü, anahtara karşılık gelen değeri içerir ("Ali," "Veli," 30 gibi).



Harita Kullanım Alanları

- **Veritabanı İşlemleri:** Veritabanı sistemlerinde, anahtarlarla değerlere erişmek için kullanılır. Özellikle NoSQL veritabanlarında yaygın.
- **Önbellek Yönetimi:** Sık kullanılan verilere hızlı erişim için kullanılır. Örneğin, web sayfası içeriği önbelleğe alınır ve URL'lere göre erişilebilir.
- **Yapay Zeka ve Veri Analizi:** Veri madenciliği ve yapay zeka uygulamalarında kullanılır. Veri indekslemesi ve öznitelik eşleştirmesi için kullanışlıdır.
- **Yazılım Geliştirme:** Yapılandırma dosyalarını yönetmek, dil çevirilerini tutmak ve çoklu ortamlara erişmek için harita veri yapısı kullanılabilir.

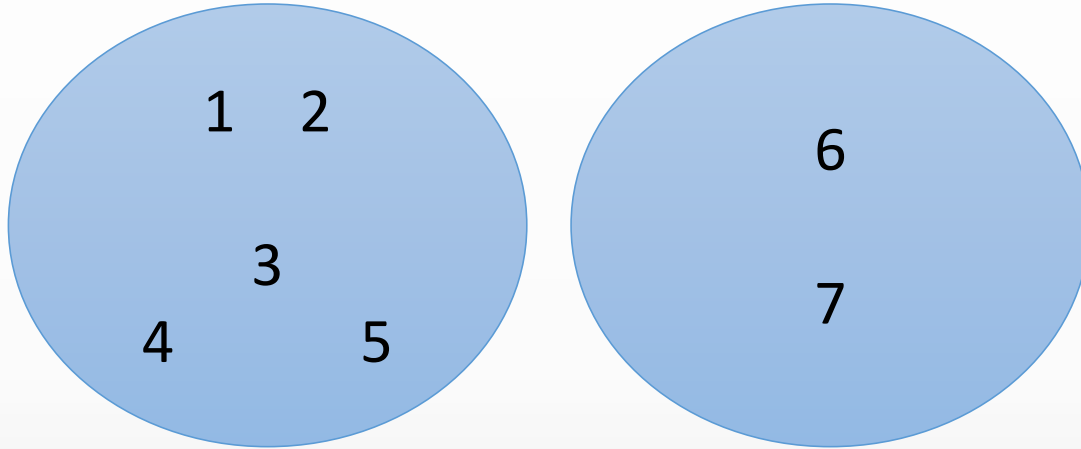


Küme Veri Yapısı (Sets)

- Benzersiz, tekil öğeler içerir.
- Öğelerin sırasız bir şekilde saklanır.
- Küme içerisinde her öğe yalnızca bir kez bulunabilir.
- Bir öğenin küme içinde olup olmadığı hızlı bir şekilde kontrol edilebilir.
- Veri analizi, mantıksal işlemler ve veritabanı gibi farklı alanda kullanılır.



Küme Veri Yapısı (Sets)



Küme: {1, 2, 3, 4, 5}

- "Küme:" kelimesi, bir küme veri yapısını temsil eder.
- Süslü parantezler { } veri yapısının başlangıcını ve sonunu gösterir.
- Rakamlar (1, 2, 3, 4, 5), kümenin içerisinde bulunan öğeleri temsil eder.

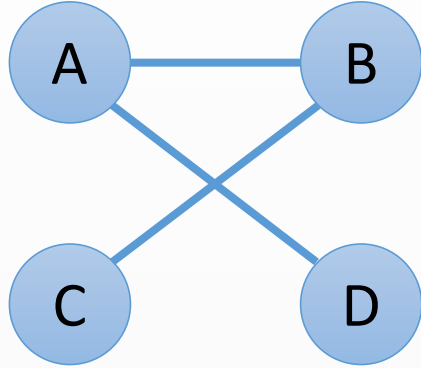


Çizge Veri Yapısı (Graphs)

- Bağlantılı nesnelere (düğümler) ve bu nesnelere birbirine bağlayan ilişkileri (kenarlar) içerir.
- Nesnelere arasındaki karmaşık ilişkileri modellemek ve analiz etmek için kullanılır.
- Sosyal ağlar, yolculuk planlaması, çevresel tasarım ve algoritmalar gibi birçok alanda yaygın olarak kullanılır.



Çizge Veri Yapısı (Graphs)



- "A," "B," "C," ve "D" harfleri düğümleri (nodes) temsil eder.
- Çizgiler, düğümler arasındaki bağlantıları (edges) gösterir.
- Örneğin, "A" ile "B" arasında bir kenar varsa, "A" ile "B" arasında bir bağlantı olduğunu belirtir.



Çizge Veri Yapısının Özellikleri

- **Düğümmler (Nodes):** Çizgenin temel yapı taşı ve verileri temsil eder.
- **Kenarlar (Edges):** Düğümleri birbirine bağlar ve ilişkileri gösterir.
- Kenarlar yönlendirilmiş veya yönlendirilmemiş oklar ile gösterilebilir.
- Kenarlara ağırlık (weight) verilerek, önem derecesi ifade edilebilir.
- Bir düğümün kendisine bir kenarla bağlanması sonucu oluşan döngülere izin verilebilir veya verilmez.



Çizge Kullanım Alanları

- **Sosyal Ağ Analizi:** Sosyal medya platformlarındaki ilişkileri ve etkileşimleri modellemek ve analiz etmek için kullanılır.
- **Harita Yolculukları:** Navigasyon uygulamaları, haritaları çizge veri yapısıyla oluşturur. En kısa yol veya rota bulmak için çizge algoritmalarını kullanır.
- **Çevre Tasarımı:** Şehir planlaması ve çevresel tasarımda yolları, yeşil alanları ve su yollarını modellemek için kullanılır.
- Birçok önemli algoritma, çizge veri yapısını temel alır. Örneğin, derinlik öncelikli arama (DFS) ve genişlik öncelikli arama (BFS) gibi.

Örnek Problemler





Tekrar Eden Sayıları Bulma

- **Soru:** Verilen bir dizide tekrar eden sayıları bulma
- tekrarEdenler adında boş bir dizi oluştur.
- İki iç içe döngü kullanılarak her iki sayı karşılaştır.
- Eğer iki sayı birbirine eşitse ve tekrarEdenler dizisi içinde bulunmuyorsa, sayıyı tekrarEdenler dizisine ekle.
- tekrarEdenler dizisi içinde bulunan tekrar eden sayılar döndür.



Tekrar Eden Sayıları Bulma

Fonksiyon TekrarEdenleriBul(dizi):

```
tekrarEdenler = BosDizi()
```

```
for i = 0 to dizi.length - 1:
```

```
    for j = i + 1 to dizi.length - 1:
```

```
        eğer dizi[i] == dizi[j] ise:
```

```
            eğer tekrarEdenler içermez dizi[i] ise:
```

```
                tekrarEdenler'e dizi[i]'yi ekle
```

```
return tekrarEdenler
```



Matris Çarpımı

- **Soru:** İki matrisin çarpımını hesaplama
- Verilen matrislerin boyutlarını kontrol et
- Sonuç matrisi oluştur
- Üç iç içe döngü kullanarak çarpım işlemini gerçekleştir.



Matris Çarpımı

Fonksiyon `MatrisCarp(matrisA, matrisB):`

```
satirA = matrisA.satirSayisi
```

```
sutunA = matrisA.sutunSayisi
```

```
satirB = matrisB.satirSayisi
```

```
sutunB = matrisB.sutunSayisi
```

eğer `sutunA` \neq `satirB` ise:

Hata: Matrisler çarpılamaz

Döndür Null



Matris Çarpımı

```
yeniMatris = BoşMatris(satirA, sutunB)
```

```
for i = 0 to satirA - 1:
```

```
    for j = 0 to sutunB - 1:
```

```
        toplam = 0
```

```
        for k = 0 to sutunA - 1:
```

```
            toplam += matrisA[i][k] * matrisB[k][j]
```

```
        yeniMatris[i][j] = toplam
```

Döndür yeniMatris



İşlem Geçmişini Kaydetme

- **Soru:** İşlem geçmişini kaydetmek için bağlı liste kullanma
- Yeni işlem listenin başına eklenir.
- En son işlem en üstte görünür.



İşlem Geçmişi Kaydetme

BağlıListe İşlemGeçmişi:

```
baş = Null # İşlem geçmişinin başı
```

Fonksiyon İşlemEkle(işlem):

```
yeniDüğüm = YeniDüğüm(işlem) # Yeni bir düğüm oluştur
```

```
yeniDüğüm.sonraki = baş # Yeni düğümü başa ekle
```

```
baş = yeniDüğüm # Baş güncelle
```



İşlem Geçmişi Kaydetme

Fonksiyon İşlemGeçmişiGöster():

```
düğüm = baş
```

```
EkранаYazdır("İşlem Geçmişi:")
```

```
while düğüm ≠ Null:
```

```
    EkранаYazdır(düğüm.işlem) # İçeriği ekrana yazdır
```

```
    düğüm = düğüm.sonraki # Bir sonraki düğüme geç
```




Parantez Uyumsuzluğu Kontrolü

- **Soru:** Bir ifade içindeki parantezlerin (yuvarlak, köşeli ve süslü) uyumsuzluğunu kontrol etmek için yığın (stack) kullanımı
- Açma parantezi geldiğinde yığına ekle
- Kapa parantez geldiğinde, yığındaki açma parantezi ile uyumlu olup olmadığını kontrol et.
- Tüm parantezler uyumluysa, True dön; aksi halde False dön.



Parantez Uyumsuzluğu Kontrolü

Fonksiyon ParantezUyumsuzluğuKontrolü(ifade):

```
yığın = BosYığın() # Boş bir yığın oluştur
```

```
for her karakter in ifade:
```

```
    eğer karakter bir açma parantezi ('(', '[', veya '{') ise:
```

```
        yığına karakteri ekle
```

```
    eğer karakter bir kapanış parantezi (')', ']', veya '}') ise:
```

```
        eğer yığın boş ise:
```

```
            Döndür False # Uyumsuz bir kapanış parantezi bulundu
```

```
            üstParantez = yığınınÜstündekiParantez()
```

```
            eğer karakter ve üstParantez uyumlu değilse:
```

```
                Döndür False # Uyumsuz kapanış parantezi bulundu
```

```
                yığından üstParantezi çıkar
```

```
    eğer yığın boş değilse:
```

```
        Döndür False # Uyumsuz açma parantezi bulundu
```

```
Döndür True # Tüm parantezler uyumludur
```



Banka Sıra İşlemleri

- **Soru:** Bir bankada müşterilerin sırayla işlem yapmasını kuyruk (queue) veri yapısı ile modelleme
- Müşterileri kuyruğa ekle
- Sırayla işlemleri başlat.
- İşlem başlatıldığında kuyrukta önde bulunan müşteriyi sıradan çıkart.



Banka Sıra İşlemleri

Kuyruk BankaSırası:

```
ön = Null # Kuyruğun önu  
arka = Null # Kuyruğun arkası
```

Fonksiyon MüşteriGirişi(müşteri):

```
yeniMüşteri = YeniMüşteri(müşteri) # Yeni bir müşteri oluştur
```

eğer arka boş ise:

```
ön = yeniMüşteri  
arka = yeniMüşteri
```

başka:

```
arka.sonraki = yeniMüşteri  
arka = yeniMüşteri
```



Banka Sıra İşlemleri

Fonksiyon İşlemBaşlat():

eğer ön boş ise:

```
EkranaYazdır("Sıra boş. Bir müşteri bekleniyor.")
```

Döndür

```
müşteri = ön.müşteri
```

```
ön = ön.sonraki
```

```
EkranaYazdır(müşteri + " işlemi başlatıldı.")
```

Eğer ön boş ise:

```
arka = Null # Son müşteri işlemi tamamladı
```

Döndür müşteri



İnternet Tarayıcı Geçmişi

- **Soru:** İnternet tarayıcının gezinme geçmişini ağaç veri yapısı kullanarak tutmak
- Her bir internet adresini, ağaç düğümleri olarak temsil et
- URL'leri alfabetik sıraya göre ekle.



İnternet Tarayıcı Geçmişi

Ağaç Düğüm:

URL

SolÇocuk

SağÇocuk

Fonksiyon YeniDüğüm(url):

düğüm = Yeni Düğüm()

düğüm.URL = url

düğüm.SolÇocuk = Null

düğüm.SağÇocuk = Null

Döndür düğüm



İnternet Tarayıcı Geçmişi

Fonksiyon GeçmişEkle(kök, url):

eğer kök boş ise:

kök = YeniDüğüm(url)

başka:

eğer url kök.URL'den küçükse:

kök.SolÇocuk = GeçmişEkle(kök.SolÇocuk, url)

başka:

kök.SağÇocuk = GeçmişEkle(kök.SağÇocuk, url)

Döndür kök



Kelime Sayma Uygulaması

- **Soru:** Verilen metinde kelimelerin kaç kez geçtiğini harita (map) veri yapısı kullanarak sayma
- Metni kelimelere ayır
- Her bir kelimeyi temizle (noktalama işaretleri ve büyük/küçük harf farkı gözetilmez).
- Her kelimeyi haritada sakla
- Sayımı güncelle.



Kelime Sayma Uygulaması

Harita KelimeSayacı:

```
kelime -> sayı
```

Fonksiyon KelimeleriSay(metin):

```
kelimeSayacı = BoşHarita() # Boş bir kelime sayacı haritası oluştur
```

```
kelimeler = metin.split(" ") # Metni kelimelere ayır
```

```
for kelime in kelimeler:
```

```
    kelime = Temizle(kelime) # Noktalama işaretlerini temizle ve küçük harfe dönüştür
```

```
    eğer kelime boş değilse: # Boş kelimeleri atla
```

```
        eğer kelime kelimeSayacı içermezse:
```

```
            kelimeSayacı[kelime] = 1 # Yeni kelimeyi ekleyin
```

```
        başka:
```

```
            kelimeSayacı[kelime] += 1 # Kelime zaten varsa sayacı artır
```

```
Döndür kelimeSayacı
```



Benzersiz Ürün İsimleri

- **Soru:** Bir çevrimiçi mağazada, kullanıcıların sepetine eklediği ürünlerin benzersiz olduğunu küme (set) veri yapısı kullanarak kontrol etme
- Ürün eklemeyen önce sepette olup olmadığına bak
- Eğer ürün sepette yoksa, ürünü sepete ekle.
- Aksi halde, bir uyarı mesajı göster.



Benzersiz Ürün İsimleri

Küme Sepet:
ürünler

Fonksiyon `ÜrünEkle(sepet, ürünAdı):`

`eğer ürünAdı sepette değilse:`

`sepete ürünAdı ekle`

`EkranaYazdır(ürünAdı + " sepete eklendi.")`

`başka:`

`EkranaYazdır(ürünAdı + " zaten sepette.")`



Sosyal Medya Arkadaşlık İlişkileri

- **Soru:** Sosyal medya platformunda kullanıcıların arkadaşlık ilişkilerini çizge (graph) veri yapısı kullanarak temsil etme
- Her kullanıcıyı bir düğüm olarak temsil et
- Arkadaşlıkları kenarlar ile bağla
- Kullanıcı eklerken, arkadaş listesini boş bir liste olarak başlat
- Arkadaş eklemelerini yap.



Sosyal Medya Arkadaşlık İlişkileri

Graf SosyalMedyaArkadaslik:

düğümler

kenarlar

Fonksiyon YeniKullanıcıEkle(graf, kullanıcıAdı):

eğer kullanıcıAdı graf.düğümder içinde değilse:

graf.düğümder'e kullanıcıAdı ekle

graf.kenarlar[kullanıcıAdı] = BoşListe()

EkранаYazdır(kullanıcıAdı + " kullanıcısı eklendi.")

başka:

EkранаYazdır(kullanıcıAdı + " zaten var.")



Sosyal Medya Arkadaşlık İlişkileri Grafiği

Fonksiyon ArkadasEkle(graf, kullanıcı1, kullanıcı2):

eğer kullanıcı1 ve kullanıcı2 graf.düğömler içinde ise:

```
graf.kenarlar[kullanıcı1].ekle(kullanıcı2)
```

```
graf.kenarlar[kullanıcı2].ekle(kullanıcı1)
```

```
EkranaYazdır(kullanıcı1 + " ve " + kullanıcı2 + " arkadaş oldular.")
```

başka:

```
EkranaYazdır("Kullanıcılar bulunamadı.")
```

Fonksiyon ArkadaslariGoster(graf, kullanıcıAdı):

eğer kullanıcıAdı graf.düğömler içinde ise:

```
EkranaYazdır(kullanıcıAdı + " kullanıcısının arkadaşları:")
```

```
her arkadas in graf.kenarlar[kullanıcıAdı]:
```

```
    EkranaYazdır(arkadas)
```

başka:

```
EkranaYazdır("Kullanıcı bulunamadı.")
```



SON