Bölüm 1: Giriş

Veri Yapıları

Veri Yapıları Nedir?

- Veri yapıları, verileri bilgisayarınızda düzenlemenin ve saklamanın bir yoludur.
- Temel amacı, verilere hızlı ve etkili bir şekilde erişmenizi sağlamaktır.

- Veri yapılarını anlamak, yazılım geliştirme sürecinizde çok önemlidir. İyi bir veri yapısı seçimi, programınızın performansını ve verimliliğini büyük ölçüde etkileyebilir.

Ders İçeriği

- Diziler (Arrays):
- Bağlı Liste (Linked Lists):
- Yığın (Stacks):
- Kuyruk (Queues):
- Ağaç (Trees):
- Çizge (Graphs):

Verimli Kod Nasıl Yazılır?

- **Doğru Veri Yapılarını Kullanın**: Göreviniz için uygun veri yapılarını kullanmak çok önemlidir. Verilerinizi doğru şekilde düzenlemek, işlemlerinizi daha hızlı yapmanıza yardımcı olabilir.
- **Doğru Algoritmaları Seçin**: Göreviniz için uygun algoritmaları seçmek, işlemlerinizi daha verimli hale getirebilir. Her problem için en uygun algoritmayı bulun.
- **Gereksiz Döngü ve Özyinelemeden Kaçının**: Gereksiz döngüler ve özyineleme, kodunuzu yavaşlatabilir. İhtiyacınız olmayan tekrarları önlemeye çalışın.
- **Uygun Veri Türlerini Kullanın**: Veri türlerini doğru bir şekilde seçmek, kodunuzun bellek kullanımını optimize etmenize yardımcı olabilir.
- **Kodunuzu Hedef Platforma Göre Optimize Edin**: Kodunuzu hedef platforma uygun şekilde optimize edin. Farklı platformlar farklı performans özelliklerine sahip olabilir.

Farklı Veri Yapıları

- Her birinin kendine özgü güçlü ve zayıf yönleri bulunmaktadır. İşte bazı yaygın veri yapıları:
- **Diziler (Arrays):** Verileri düzenlemek ve erişmek için kullanılır. Hafıza kullanımı sabit boyutludur.
- **Bağlı Listeler (Linked Lists):** Verileri bağlantılı bir şekilde saklamak için kullanılır. Esnek boyutlara sahiptir.
- **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.

Algoritmalar Nedir?

- Bir algoritma, bir problemi çözmek için adım adım bir prosedürdür.
- Aynı problemi çözmek için birçok farklı algoritma bulunabilir ve bazı algoritmalar diğerlerine göre daha verimli olabilir.
- 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 hızlı ve daha az kaynak kullanarak gerçekleştirebilir.

Farklı Algoritmalar

- Çeşitli algoritmalar vardır ve her biri belirli bir problemi çözmek için geliştirilmiştir. İşte 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.

Kodunuzu Neden Optimize Etmelisiniz?

- Kodunuzu optimize etmek, yazılımınızın daha hızlı çalışmasını ve daha az kaynak tüketmesini sağlayabilir. Bu, kullanıcı deneyimini iyileştirir ve maliyetleri düşürebilir.
- Kodunuzu optimize etmek, yazılım geliştirme sürecinin ayrılmaz bir parçasıdır. Doğru optimizasyon tekniklerini kullanmak, yazılımınızın performansını artırabilir ve daha iyi bir kullanıcı deneyimi sağlayabilir.

Kod Optimizasyonu İçin Yaygın Teknikler

- **Gereksiz Döngü ve Özyinelemeyi Kaldırma**: Kodunuzda gereksiz döngüler ve özyinelemeler bulunuyorsa, bunları kaldırmak veya azaltmak, kodunuzu hızlandırabilir.
- **Uygun Veri Türlerini Kullanma**: Doğru veri türlerini kullanmak, kodunuzu daha verimli hale getirebilir. Örneğin, büyük veri koleksiyonları için veri yapılarını kullanmak.
- **Kodunuzu Hedef Platforma Göre Optimize Etme**: Kodunuzu hedeflediğiniz platforma uygun şekilde optimize etmek önemlidir. Farklı platformlar farklı performans karakteristiklerine sahiptir.

Optimizasyonun Faydaları

- **Daha Hızlı Çalışma**: Optimize edilmiş kod, işlemleri daha hızlı gerçekleştirir.
- **Daha Az Kaynak Kullanımı**: Verimli kod, daha az bellek ve işlemci kaynağı kullanır.
- **Daha İyi Kullanıcı Deneyimi**: Hızlı ve düşük kaynak kullanımı, kullanıcıların yazılımınızı daha memnuniyetle kullanmasını sağlar.
- **Daha Düşük Maliyetler**: Optimize edilmiş kod, daha az sunucu kaynağı gerektirir, bu da maliyetleri düşürür.

İyi Bir Algoritma Nedir?

- İyi bir algoritma, bir problemi etkili bir şekilde çözen algoritmadır. Bir algoritmanın verimli olmasına katkıda bulunan birçok faktör bulunmaktadır ve bunlar şunları içerebilir:
- **Algoritmanın Zaman Karmaşıklığı (Time Complexity):** Algoritmanın çalışma süresi veya işlem sayısı gibi faktörler, zaman karmaşıklığını belirler. İyi bir algoritma, işlem süresini minimize eder.
- **Algoritmanın Alan Karmaşıklığı (Space Complexity):** Algoritma tarafından kullanılan bellek miktarı, alan karmaşıklığını belirler. İyi bir algoritma, bellek kullanımını optimize eder.
- **Algoritmanın Doğruluğu (Accuracy):** Algoritmanın istenen sonuçları doğru bir şekilde üretmesi önemlidir. Yanlış sonuçlar veren bir algoritma kullanışsızdır.
- **Algoritmanın Sağlamlığı (Robustness):** Algoritmanın çeşitli durumlar ve girdilerle başa çıkabilme yeteneği, sağlamlığını belirler. İyi bir algoritma, farklı senaryolara uyum sağlayabilir.

İyi Algoritmaların Faydaları

- **Daha Hızlı İşlem**: İyi bir algoritma, işlemleri hızlandırabilir.
- **Daha Az Bellek Kullanımı**: Alan karmaşıklığı düşük algoritmalar, daha az bellek kullanır.
- **Güvenilir Sonuçlar**: İyi algoritmalar, doğru ve güvenilir sonuçlar üretir.
- **Çeşitli Koşullara Uyum Sağlama**: İyi algoritmalar, farklı senaryolara uyum sağlar ve hata toleransı gösterir.

Zaman Karmaşıklığı Nedir?

- Bir algoritmanın zaman karmaşıklığı, algoritmanın çalışma süresinin ne kadar sürdüğünü ölçen bir ölçüdür. Zaman karmaşıklığı genellikle Büyük O (Big O) notasyonu kullanılarak ifade edilir. Büyük O notasyonu, algoritmanın çalışma süresinin girdi boyutunun artışına nasıl bağlı olduğunu gösterir.
- Zaman karmaşıklığı, bir algoritmanın ne kadar hızlı veya yavaş çalıştığını anlamamıza yardımcı olur. Büyük O notasyonu, algoritmaların performansını analiz etmek ve karşılaştırmak için güçlü bir araçtır.

Büyük O (Big O) Notasyonu Nedir?

- Büyük O notasyonu, bir 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 süre alır.

Büyük O Notasyonu Örnekleri:

- O(1): Sabit zaman karmaşıklığı.
- O(log n): Logaritmik zaman karmaşıklığı.
- O(n): Doğrusal zaman karmaşıklığı.
- O(n log n): Lineerithmic zaman karmaşıklığı.
- O(n^2): Kare zaman karmaşıklığı.
- O(2^n): Üssel zaman karmaşıklığı.

Büyük O Notasyonu Karmaşıklık Örnekleri

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

Büyük O Notasyonu Karmaşıklık Örnekleri

- O(n log n): Lineerithmic Zaman Karmaşıklığı
  - Örnek: Hızlı sıralama (Quick Sort) algoritması.
  - Açıklama: Genellikle hızlı sıralama gibi verileri bölüp sıralamak için kullanılan algoritmaların karmaşıklığıdır.
- O(n^2): Kare Zaman Karmaşıklığı
  - Örnek: İç içe döngülerle bir matrisi tarama.
  - Açıklama: İki döngü kullanıldığında, her elemanın diğer tüm elemanlarla karşılaştırıldığı bir karmaşıklıktır.
- O(2^n): Üssel Zaman Karmaşıklığı
  - Örnek: Tüm alt kümeleri bulma.
  - Açıklama: Kümelerin alt kümelerini bulmak gibi her bir adımda iki kat artan bir karmaşıklığa sahiptir.

Alan Karmaşıklığı Nedir?

- Bir algoritmanın alan karmaşıklığı, algoritmanın kullanıdığı bellek miktarını ölçen bir ölçüdür. Alan karmaşıklığı genellikle Büyük O (Big O) notasyonu kullanılarak ifade edilir. Büyük O notasyonu, algoritmanın bellek kullanımının girdi boyutunun artışına nasıl bağlı olduğunu gösterir.
- Alan karmaşıklığı, algoritma tasarımında önemlidir çünkü sınırlı bellek kaynaklarına sahip sistemlerde çalışan yazılımların bellek kullanımını etkiler. İyi bir alan karmaşıklığına sahip algoritmalar, bellek verimliliği açısından daha avantajlıdır.

Büyük O Notasyonu Örnekleri

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

Doğruluk Nedir?

- Bir algoritmanın doğruluğu, algoritmanın ürettiği çıktının doğru cevaba ne kadar yakın olduğunu ölçen bir ölçüdür. Başka bir deyişle, algoritmanın çıktısı 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. Yanlış sonuçlar, hatalı kararlar alınmasına ve güvenilir olmayan yazılımların oluşturulmasına neden olabilir.
- Doğruluk, bir algoritmanın başarısını belirleyen önemli bir faktördür. İyi bir algoritma, yüksek doğruluk seviyelerine ulaşırken, yanlış sonuçları minimumda tutar.

Doğruluğun Ölçülmesi

- Algoritmanın doğruluğunu ölçmek için genellikle çeşitli değerlendirme metrikleri kullanılır. Bu metrikler, algoritmanın ürettiği sonuçları gerçek verilere veya bilinen doğru sonuçlara karşı karşılaştırır. Örnek metrikler şunlar olabilir:

- **Hata Oranı (Error Rate):** Algoritmanın yanlış sonuçlarının oranı.
- **Doğruluk (Accuracy):** Doğru sonuçların oranı.
- **Hassasiyet (Precision):** Pozitif olarak tahmin edilen sonuçların ne kadarının gerçekte pozitif olduğunu ölçer.
- **Duyarlılık (Recall):** Gerçekten pozitif olanların ne kadarının pozitif olarak tahmin edildiğini ölçer.

Sağlamlık Nedir?

- Bir algoritmanın sağlamlığı, algoritmanın beklenmeyen girdileri nasıl işlediğini ölçen bir ölçüdür. Sağlam bir algoritma, normal aralığının dışındaki girdileri bile çökmeksizin veya yanlış sonuçlar üretmeden işleyebilir.
- Sağlam bir algoritma, 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. Sağlam algoritmalar, güvenilir yazılım geliştirme, güvenlik ve dayanıklılık için kritik öneme sahiptir.
- Sağlamlık, bir algoritmanın beklenmeyen durumlarla başa çıkma yeteneğini ölçen önemli bir faktördür. Sağlam algoritmalar, güvenilir ve dayanıklı yazılımların temelini oluşturur.

Sağlamlığın Ölçülmesi

- Sağlamlığın ölçülmesi, algoritmanın çeşitli beklenmeyen girdilere nasıl tepki verdiğini anlamayı içerir. Sağlamlık testleri, aşağıdaki gibi senaryoları içerebilir:
- **Geçersiz Girdiler (Invalid Inputs):** Algoritmanın 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):** Algoritmanın büyük veri kümesini işleme yeteneği.
- **Aşırı Yüklenme (Overload):** Algoritmanın aşırı yüklendiğinde nasıl davrandığı.

Sağlamlığın Faydaları

- **Kararlılık**: Sağlam algoritmalar, beklenmeyen durumlarla başa çıkabilir ve çökme riskini en aza indirir.
- **Güvenilirlik**: Güvenilir yazılım ve hizmetler oluşturmak için temel bir unsurdur.
- **Müşteri Memnuniyeti**: Sağlam yazılım, kullanıcıların güvenini kazanır ve memnuniyetini artırır.

İyi Bir Veri Yapısı Nedir?

- İyi bir veri yapısı, verileri etkili bir şekilde saklamak, erişmek ve işlemek için tasarlanmış bir yapıdır. Verilerin organizasyonu, bir veri yapısının kalitesini belirler. İyi bir veri yapısı, verileri düzenli bir şekilde yönetir ve istenilen sonuçları hızlıca üretir.
- Her veri yapısının kendine özgü avantajları ve sınırlamaları vardır, bu nedenle doğru seçim yapmak önemlidir.
- İyi bir veri yapısı, verilerin etkili bir şekilde saklandığı ve yönetildiği temel bir yapıdır. Veri yapısı seçiminde dikkatli olmak, yazılım geliştirme sürecini optimize etmek için kritiktir.

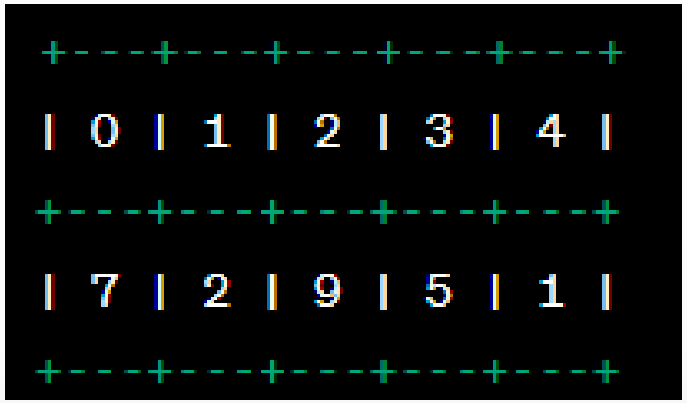
Farklı Veri Yapıları

- **Diziler (Arrays):** Verileri düzenlemek ve erişmek için kullanılır. Hafıza kullanımı sabit boyutludur.
- **Bağlı Listeler (Linked Lists):** Verileri bağlantılı bir şekilde saklamak için kullanılır. Esnek boyutlara sahiptir.
- **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.

Diziler (Arrays)

- Dizi, verileri ardışık bir bellek bloğunda saklayan bir veri yapısıdır. Diziler, verilere kolay erişim ve manipülasyon sağlar. Veriler, dizi içinde sırayla depolanır ve her veri öğesine bir indeksle erişilebilir.
- Diziler, verileri düzenli bir şekilde saklamak ve hızlı erişim sağlamak için kullanışlı bir veri yapısıdır. Ancak, verilerin eşit olarak dağılmadığı durumlarda veri erişimi ve işleme verimliliği azalabilir. Bu nedenle, dizi kullanmadan önce veri yapısının gereksinimlerini dikkatlice düşünmek önemlidir.

Diziler (Arrays)



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

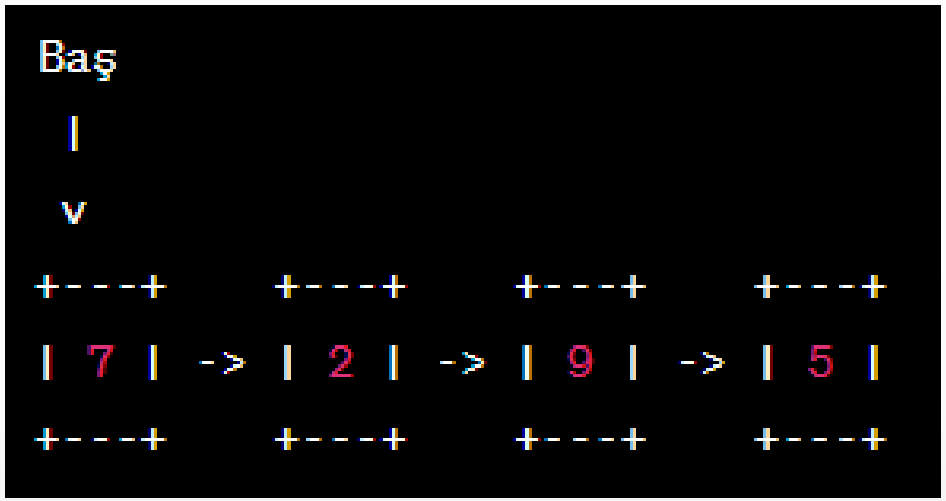
Diziler (Arrays)

- Dizilerin Avantajları
  - Hızlı Erişim: Diziler, doğrudan indeksleme kullanarak verilere hızlı erişim sağlar.
  - Basit Kullanım: Diziler, verileri düzenli bir şekilde saklamak için basit ve kolay bir yol sunar.
- Dizilerin Dezavantajları
  - Sabit Boyut: Diziler genellikle sabit bir boyuta sahiptir, bu nedenle veri boyutu dinamik olarak değiştirilemez.
  - Veri Dağılımı Sorunları: Veriler eşit olarak dağılmadığında, bazı durumlarda veri erişimi ve arama verimliliği azalabilir.
    - Eşit olarak dağılmayan veriler, dizinin sonunda veya başında sıkışabilir, bu da erişim sürelerini uzatabilir.
    - Dizi boyutu önceden belirlendiği için, veri boyutu büyüdüğünde veya küçüldüğünde uygun bir boyutu korumak zor olabilir.

Bağlı Listeler (Linked Lists)

- Bağlı liste, verileri düğümler adı verilen bağlı bir liste içinde saklayan bir veri yapısıdır. Her düğüm, veriyi ve bir sonraki düğümün referansını içerir. Bu referanslar, verilerin ardışık bellek yerine bağlı bir şekilde saklandığı anlamına gelir.
- Bağlı listeler, verilerin dinamik olarak büyüdüğü veya dağılımı dengesiz olduğu durumlarda kullanışlı bir veri yapısıdır. Ancak, erişim süreleri dizilere göre daha yavaş olabilir ve daha fazla bellek kullanabilir.

Bağlı Listeler (Linked Lists)



- "Baş" bağlı listenin başlangıç noktasını veya başını gösterir.
- Her düğüm, bir değeri (örneğin, 7, 2, 9, 5) içeren bir kutu olarak temsil edilir ve bir sonraki düğüme doğru bir ok ile bağlanır.
- "->" işareti düğümler arasındaki bağlantıyı gösterir ve bir düğümden diğerine geçişi temsil eder.

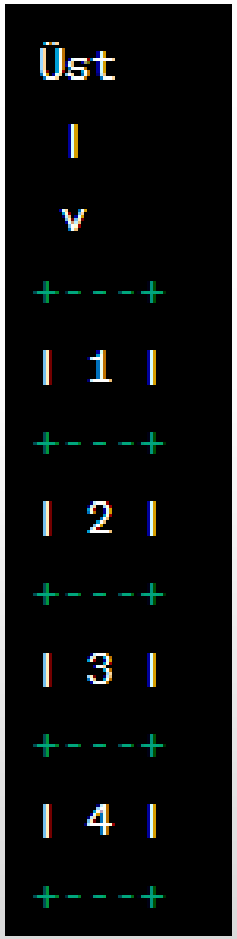
Bağlı Listeler (Linked Lists)

- Bağlı Listelerin Avantajları
  - Dinamik Boyut: Bağlı listeler, veri boyutunu dinamik olarak değiştirmenizi sağlar. Düğümler eklenip çıkarılabilir.
  - Veri Dağılımı Sorunlarına Dayanıklılık: Veriler eşit olarak dağılmadığında, bağlı listeler daha etkilidir. Düğümler rastgele yerleştirilebilir.
    - Bağlı listeler, eşit olarak dağılmayan verilere daha iyi uyum sağlar. Düğümler rastgele yerleştirilebildiği için veri dağılımı sorunlarına daha dayanıklıdır.
- Bağlı Listelerin Dezavantajları
  - Yavaş Erişim: Verilere erişim, bağlı düğümler arasında gezinme gerektirdiği için dizilere göre daha yavaş olabilir.
  - Daha Fazla Bellek Kullanımı: Her düğüm, veri ve bir sonraki düğümün referansını içerdiğinden, daha fazla bellek kullanır.

Yığınlar (Stacks)

- Yığın (Stack), verileri son giren, ilk çıkar (LIFO - Last-In, First-Out) mantığına göre saklayan bir veri yapısıdır. Yığınlar, verilerin eklenip çıkarılmasını sırayla gerçekleştirir ve her yeni öğe yığının en üstüne eklenir.
- Yığınlar, son eklenen öğenin ilk çıkarıldığı LIFO mantığına dayalı bir veri yapısıdır. Rekürsif fonksiyonlar, geri izleme ve işlem geçmişi yönetimi gibi çeşitli alanlarda kullanılırlar. Yığınları anlamak, bu tür problemleri daha etkili bir şekilde çözmeye yardımcı olabilir.

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 ve altındaki öğeye doğru bir ok ile bağlanır.

Yığınların Kullanım Alanları

- **Rekürsif Fonksiyonlar (Recursion):** Fonksiyonlar kendi kendini çağırdığında, her çağrı bir yığın çerçevesi olarak saklanır. Bu, fonksiyonların geri döndüğünde en son çağrının sonuçlarına geri dönebilmesini sağlar.
- **Geri İzleme (Backtracking):** Problemi çözmek için olasılıkları denediğinizde ve geri adım atmanız gerektiğinde yığınlar kullanılır. Özellikle problem çözme algoritmalarında yaygın bir kullanım alanıdır.
- **İşlem Geçmişi (Undo/Redo):** Bir uygulamada kullanıcı işlemlerini geri almak (Undo) veya geri getirmek (Redo) için yığınlar kullanılabilir.

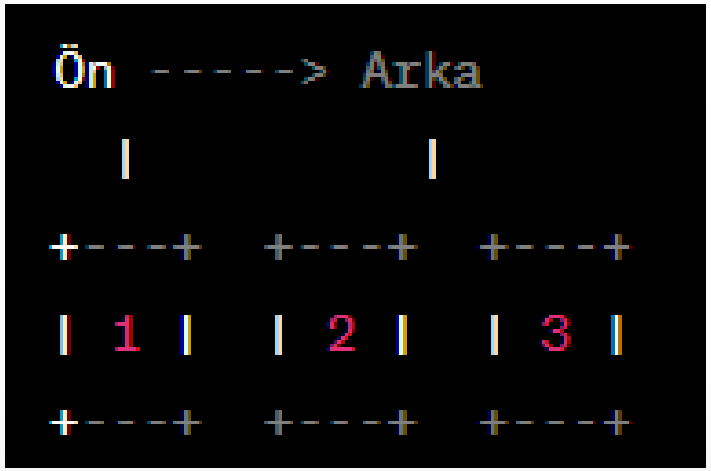
Yığınların Avantajları

- **LIFO Mantığı**: Yığınlar, son eklenen öğeyi ilk çıkararak LIFO mantığını takip eder. Bu, bazı algoritmalar için çok uygun olabilir.
- **Hafıza Yönetimi**: Yığınlar, belleği etkili bir şekilde kullanır ve gereksiz bellek sızıntılarından kaçınmaya yardımcı olur.

Kuyruklar (Queues)

- Kuyruk (Queue), verileri ilk giren, ilk çıkar (FIFO - First-In, First-Out) mantığına göre saklayan bir veri yapısıdır. Kuyruklar, verilere eklenme ve çıkarılma işlemlerini sırayla gerçekleştirir ve en önce eklenen öğe en önce çıkarılır.
- Kuyruklar, verileri FIFO mantığına göre sırayla işlemek için kullanılan bir veri yapısıdır. Görev sıralama, istek işleme ve bazı algoritmaların uygulanmasında kullanılırlar. Kuyrukları anlamak, iş sıralaması ve işlem sırasını yönetme konularında yardımcı olabilir.

Kuyruklar (Queues)



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

Kuyrukların Kullanım Alanları

- **Görev Sıralama (Task Scheduling):** İşlemcilerin görevleri sırayla çalıştırmasına yardımcı olur. İlk giren görev ilk olarak çalıştırılır.
- **İstek İşleme (Request Processing):** Sunucular, gelen istekleri kuyrukta sıraya alır ve sırayla işler. Bu, 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.

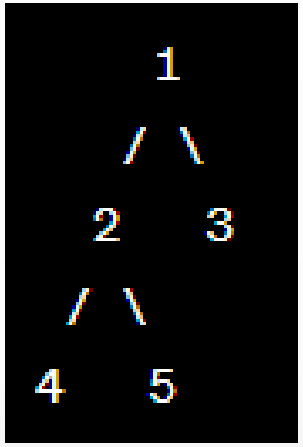
Kuyrukların Avantajları

- **FIFO Mantığı:** Kuyruklar, ilk giren öğenin ilk çıkarıldığı FIFO mantığını takip eder. İş sıralaması veya istek işleme gibi senaryolarda işlerin sırayla çalıştırılmasını sağlar.
- **İş Yükü Dengelemesi:** Kuyruklar, yüksek talep dönemlerinde iş yükünü dengelemeye yardımcı olabilir.

Ağaçlar (Trees)

- Ağaç (Tree), verileri hiyerarşik bir yapı içinde saklayan bir veri yapısıdır. Her ağaç, bir kök düğüm (root node) ile başlar ve bu kök düğümden dallanarak alt düğümlere sahiptir. Her düğüm, kendisine bağlı alt düğümleri yönlendirir ve bu şekilde bir hiyerarşi oluşturulur.
- Ağaçlar, verileri hiyerarşik bir yapıda saklamak ve temsil etmek için kullanılan önemli bir veri yapısıdır. Dosya sistemleri, biyolojik taksonomi ve veritabanı indekslemesi gibi birçok alanda yaygın olarak kullanılırlar. Ağaçları anlamak, verilerin karmaşıklığını düzenleme ve hızlı erişim sağlama konularında yardımcı olabilir.

Ağaçlar (Trees)



- Sayı 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):** Bilgisayar dosyalarını ve klasörlerini hiyerarşik bir yapıda saklamak için ağaçlar kullanılır. 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):** Veritabanlarında hızlı veri erişimi sağlamak için kullanılır. İndeks ağaçları, verileri sıralar ve erişim hızını artırır.

Ağaçların Avantajları

- **Hiyerarşik Temsil:** Verilerin doğal hiyerarşisini yansıtmak için mükemmel bir yapıdır.
- **Hızlı Arama:** Verilerin hızlıca bulunmasını sağlar. İndeksleme için idealdir.
- **Veri Yapısı Olarak Esneklik:** Ağaçlar, veri yapısı olarak esnek bir şekilde kullanılabilir ve veri türlerine uyum sağlayabilir.

Harita Veri Yapısı (Maps)

- Harita (Map), anahtar-değer çiftlerini depolayan bir veri yapısıdır. Her anahtar, bir değerle eşleştirilir ve bu eşleştirme bir harita içinde saklanır. Anahtarlar genellikle benzersizdir ve her biri yalnızca bir değeri temsil eder.
- Harita veri yapısı, anahtar-değer çiftlerini etkili bir şekilde saklamak ve erişmek için kullanılan önemli bir veri yapısıdır. İşlem hızını artırırken, veri organizasyonunu basitleştirmeye yardımcı olur.

Harita Veri Yapısı (Maps)

Anahtarlar:	Değerler:
"Ad"	"Ali"
"Soyad"	"Veli"
"Yaş"	30
"Şehir"	"İstanbul"

- "Anahtarlar" bölümü, her bir anahtarın adını içerir (örneğin, "Ad," "Soyad," "Yaş," "Şehir").
- "Değerler" bölümü, her bir anahtarın karşılık gelen değerini içerir (örneğin, "Ali," "Veli," 30, "İstanbul").

Harita Veri Yapısının Özellikleri

- **Anahtar-Değer İlişkisi:** Her anahtar, bir değerle ilişkilidir. Bu, her anahtarın eşsiz bir değeri temsil etmesini sağlar.
- **Hızlı Erişim:** Haritalar, anahtar kullanarak değerlere hızlı erişim sağlar. Bu, büyük veri koleksiyonlarını etkili bir şekilde işlemek için kullanışlıdır.
- **Benzersiz Anahtarlar:** Haritalar genellikle benzersiz anahtarlar gerektirir, çünkü aynı anahtar birden fazla değeri temsil edemez.

Harita Kullanım Alanları

- **Veritabanı İşlemleri**: Veritabanı sistemlerinde, anahtarlarla değerlere erişmek için kullanılır. Özellikle NoSQL veritabanlarında yaygın olarak kullanılır.
- **Önbellek Yönetimi**: Önbelleklerde, sık kullanılan verileri hızlıca erişmek için kullanılır. Örneğin, web sayfası içeriği önbelleğe alınabilir ve URL'lere göre erişilebilir.
- **Yapay Zeka ve Veri Analizi**: Haritalar, veri madenciliği ve yapay zeka uygulamalarında sıklıkla kullanılır. Özellikle veri indekslemesi ve öznitelik eşleştirmesi için kullanışlıdır.
- **Yazılım Geliştirme**: Programcılar, yapılandırma dosyalarını yönetmek, dil çevirilerini saklamak ve çoklu ortamlara erişmek için harita veri yapısını kullanabilirler.

Küme Veri Yapısı (Sets)

- Küme, benzersiz öğeleri içeren ve öğelerin sırasız bir şekilde saklandığı bir veri yapısıdır. Her öğe yalnızca bir kez bulunabilir.
- Küme veri yapısı, benzersiz öğeleri içeren ve hızlı erişim sağlayan bir veri yapısıdır. Veri analizi, mantıksal operasyonlar ve veritabanları gibi birçok farklı alanda kullanılır.

Küme Veri Yapısı (Sets)

```
Küme: {1, 2, 3, 4, 5}
```

- "Küme:" kelimesi, bir küme veri yapısını temsil ettiğini belirtir.
- Küme içindeki süslü parantezler {} veri yapısının başlangıcını ve sonunu gösterir.
- Küme içindeki rakamlar (örneğin, 1, 2, 3, 4, 5), kümenin içinde bulunan öğeleri temsil eder.

Küme Veri Yapısının Özellikleri

- **Benzersiz Öğeler**: Küme veri yapısı, yalnızca benzersiz öğeleri içerir. Aynı öğe birden fazla kez eklenemez.
- **Hızlı Erişim**: Küme, belirli bir öğenin küme içinde olup olmadığını hızlı bir şekilde kontrol etmek için kullanışlıdır.

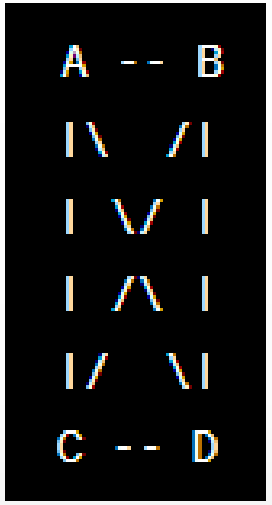
Küme Kullanım Alanları

- **Veri Analizi**: İstatistiksel veri analizi ve veri madenciliği uygulamalarında benzersiz öğeleri filtrelemek ve saymak için kullanılır.
- **Mantıksal Operasyonlar**: Küme işlemleri, matematiksel ve mantıksal işlemlerde kullanılır. Birleşim, kesişim ve fark işlemleri gibi.
- **Veritabanları**: Veritabanlarında benzersiz anahtarları ve değerleri saklamak için kullanılır. İlişkisel veritabanlarda "Birincil Anahtar" alanları gibi.
- **Web Geliştirme**: Web uygulamalarında, örneğin kullanıcıların tercihlerini saklamak için kullanılır.

Çizge Veri Yapısı (Graphs)

- Çizge, bağlantılı nesneleri (düğümler) ve bu nesneleri birbirine bağlayan ilişkileri (kenarlar) içeren bir veri yapısıdır. Bir çizge, karmaşık bağlantıları temsil etmek ve analiz etmek için kullanılır.
- Çizge veri yapısı, nesneler arasındaki karmaşık ilişkileri modellemek ve analiz etmek için kullanılan önemli bir veri yapısıdır. 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

- **Düğümler (Nodes):** Çizgenin temel yapı taşlarıdır ve verileri temsil ederler.
- **Kenarlar (Edges):** Düğümleri birbirine bağlar ve ilişkileri gösterir.
- **Yönlendirilmiş veya Yönlendirilmemiş:** Kenarlar yönlendirilmiş (oklarla gösterilir) veya yönlendirilmemiş (ok olmadan) olabilir.
- **Ağırlıklar (Weights):** Kenarlara ağırlık (weight) eklenerek, kenarların önem derecesi ifade edilebilir.
- **Döngüler (Loops):** Bir düğümün kendisine bir kenarla bağlanması sonucu oluşan döngülere izin verilebilir veya verilemez.

Çizge Kullanım Alanları

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

Tekrar Eden Sayıları Bulma

- **Soru**: Verilen bir 1D dizide tekrar eden sayıları nasıl bulabilirsiniz?

- İlk olarak, boş bir dizi oluşturulur ve ardından iki iç içe döngü kullanılarak her iki sayının karşılaştırılması sağlanır. Eğer iki sayı birbirine eşitse ve tekrarEdenler dizisi içinde bulunmuyorsa, bu sayılar tekrarEdenler dizisine eklenir. Sonunda, tekrarEdenler dizisi içinde bulunan tekrar eden sayılar döndürülür.

Tekrar Eden Sayıları Bulma

```
Fonksiyon TekrarEdenleriBul(dizi):
    tekrarEdenler = BosDizi()  # Tekrar eden sayıları saklamak için boş bir dizi oluştur

    for i = 0 to dizi.length - 1:
        for j = i + 1 to dizi.length - 1:
            eğer dizi[i] == dizi[j] ise:
                eğer tekrarEdenler içermezse dizi[i] ise:   # Aynı sayıyı yalnızca bir kez ekleyin
                    tekrarEdenler'e dizi[i]'yi ekle
                dizi[j]'yi tekrarEdenler'e ekle

    return tekrarEdenler
```

Matris Çarpımı

- **Soru**: İki matrisin çarpımını hesaplamak için nasıl bir algoritma oluşturabilirsiniz?

- İlk olarak, giriş matrislerinin boyutları kontrol edilir ve matrislerin çarpılabilirliği belirlenir. Ardından, sonuç matrisi oluşturulur ve üç iç içe döngü kullanılarak çarpım işlemi gerçekleştirilir.

Matris Çarpımı

```
Fonksiyon MatrisCarp(matrisA, matrisB):
    satirA = matrisA.satirSayisi
    sutunA = matrisA.sutunSayisi
    satirB = matrisB.satirSayisi
    sutunB = matrisB.sutunSayisi

    eğer sutunA ≠ satirB ise:
        Hata: Matrisler çarpılamaz
        Döndür Null
```

Matris Çarpımı

```
    yeniMatris = BoşMatris(satirA, sutunB)  # Sonuç matrisini
oluştur

    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 Geçmişi Kaydetme

- **Soru**: Bir işlem geçmişini kaydetmek için bağlı liste veri yapısını nasıl kullanabilirsiniz?

- İşlem geçmişi, yeni işlemler ekledikçe başa doğru büyür ve en yeni işlem en üstte görünür.

İşlem Geçmişi Kaydetme

```
BağlıListe İşlemGeçmişi:
    baş = Null  # İşlem geçmişinin başı

Fonksiyon İşlemEkle(işlem):
    yeniDüğüm = YeniDüğüm(işlem)   # Yeni bir bağlı liste düğümü oluştur
    yeniDüğüm.sonraki = baş   # Yeni düğümü işlem geçmişinin başına ekle
    baş = yeniDüğüm  # Baş güncelle
```

İşlem Geçmişi Kaydetme

```
Fonksiyon İşlemGeçmişiGöster():
    düğüm = baş
    EkranaYazdır("İşlem Geçmişi:")
    while düğüm ≠ Null:
        EkranaYazdır(düğüm.işlem)  # Düğümün içeriğini 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 bir yığın (stack) nasıl kullanılır?

- Açma parantezleri yığına eklenir ve kapanış parantezleriyle karşılaşıldığında, yığındaki üstteki açma parantezi ile uyumlu olup olmadığı kontrol edilir. Eğer tüm parantezler uyumluysa, işlem True döner; aksi halde False döner.

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 nasıl simüle edebilirsiniz? Bir sıra işlemlerini yönetmek için bir kuyruk (queue) veri yapısını nasıl kullanabilirsiniz?

- Müşteriler kuyruğa eklenir ve sırayla işlemleri başlatılır. Kuyruk, işlem başlatıldığında önündeki müşteriyi işlem sırasından çıkarır.

Banka Sıra İşlemleri

```
Kuyruk BankaSırası:
    ön = Null  # Kuyruğun önü
    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**: Bir internet tarayıcının gezinme geçmişini nasıl temsil edebilir ve bu geçmişi ağaç veri yapısı kullanarak nasıl simüle edebilirsiniz?

- Her internet adresi, ağaç düğümleri olarak temsil edilir ve URL'lerin alfabetik sıraya göre eklenmesi sağlanır.

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**: Bir metin belgesindeki kelimelerin kaç kez geçtiğini sayan bir uygulama nasıl oluşturulabilir? Bu işlemi harita (map) veri yapısı kullanarak nasıl yapabilirsiniz?

- İlk olarak, metin kelimelere ayrılır ve her kelime temizlenir (noktalama işaretleri ve büyük/küçük harf farkı gözetilmez). Ardından, her kelime haritada saklanır ve sayımı güncellenir.

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 nasıl kontrol edebilirsiniz? Bu işlemi bir küme (set) veri yapısı kullanarak nasıl yapabilirsiniz?

- Her ürün eklemesi önce sepette olup olmadığına bakar ve eğer ürün sepette yoksa, ürünü sepete ekler. Aksi halde, ürün zaten sepette bulunuyorsa bir uyarı mesajı gösterir.

Benzersiz Ürün İsimleri

```
Küme Sepet:
    ürünler

Fonksiyon ÜrünEkle(sepet, ürünAdı):
    eğer ürünAdı sepette değilse:
        sepette ürünAdı ekle
        EkranaYazdır(ürünAdı + " sepete eklendi.")
    başka:
        EkranaYazdır(ürünAdı + " zaten sepette.")
```

Sosyal Medya Arkadaşlık İlişkileri Grafiği

- **Soru**: Sosyal medya platformunda kullanıcıların arkadaşlık ilişkilerini nasıl temsil edebilir ve bu ilişkileri bir graf (graph) veri yapısı kullanarak nasıl modelleyebilirsiniz?

- Her kullanıcı bir düğüm olarak temsil edilir ve arkadaşlıklar kenarlarla bağlanır. Kullanıcılar eklenirken, yeni bir kullanıcının arkadaş listesi boş bir liste olarak başlatılır ve arkadaş eklemeleri yapılır.

Sosyal Medya Arkadaşlık İlişkileri Grafiği

```
Graf SosyalMedyaArkadaslik:
    düğümler
    kenarlar

Fonksiyon YeniKullanıcıEkle(graf, kullanıcıAdı):
    eğer kullanıcıAdı graf.düğümler içinde değilse:
        graf.düğümler'e kullanıcıAdı ekle
        graf.kenarlar[kullanıcıAdı]   =   BoşListe()      #   Yeni
kullanıcının arkadaş listesini oluştur
        EkranaYazdır(kullanıcıAdı + " kullanıcısı eklendi.")
    başka:
        EkranaYazdır(kullanıcıAdı + " zaten var.")
```

Sosyal Medya Arkadaşlık İlişkileri Grafiği

```
Fonksiyon ArkadasEkle(graf, kullanıcı1, kullanıcı2):
    eğer kullanıcı1 ve kullanıcı2 graf.düğümler 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.düğümler içinde ise:
        EkranaYazdır(kullanıcıAdı + " kullanıcısının arkadaşları:")
        her arkadas in graf.kenarlar[kullanıcıAdı]:
            EkranaYazdır(arkadas)
    başka:
        EkranaYazdır("Kullanıcı bulunamadı.")
```

Soru

Bir yığın (stack), hangi mantığa göre çalışır?

A) FIFO (First-In, First-Out)
B) LIFO (Last-In, First-Out)
C) Priority (Öncelik)
D) Rastgele

Soru

Bir çizge (graph) veri yapısında düğümler arasındaki ilişkileri temsil eden unsura ne denir?

A) Çizgi

B) Kenar (Edge)

C) Dal (Branch)

D) Kök (Root)

Soru

Hangi veri yapısı rastgele erişime en uygun olanıdır?

A) Kuyruk
B) Yığın
C) Bağlı Liste
D) Dizi

Soru

Bir harita (map) veri yapısının temel işlevi nedir?

A) Elemanları sıralı olarak saklamak
B) Elemanları benzersiz bir şekilde saklamak
C) Elemanları LIFO düzenine göre saklamak
D) Elemanları FIFO düzenine göre saklamak

Soru

Bağlı liste (linked list) veri yapısında, en kötü durumda bir elemanı aramak için hangi zaman karmaşıklığı kullanılır?

A) O(1)

B) O(log N)

C) O(N)

D) O(N^2)

Soru

Hangi veri yapısı, hiyerarşik ilişkileri (örneğin, dosya sistemi veya organizasyon hiyerarşisi) modellemek için uygun bir seçenektir?

A) Ağaç

B) Graf

C) Yığıt

D) Küme

Soru

Bir küme (set) veri yapısının elemanlarının temel özelliği nedir?

A) Sıralı olmaları
B) Sıralanmış olmaları
C) Benzersiz olmaları
D) Bağlı liste içinde bulunmaları

Soru

Bir yığın (stack) veri yapısının temel özelliği nedir?

A) İlk giren, ilk çıkar (FIFO) mantığı
B) Son giren, ilk çıkar (LIFO) mantığı
C) Rastgele erişim
D) Elemanların sıralı olması

Soru

Ağaç (Tree) veri yapısındaki bir düğümün kaç çocuğu olabilir?

A) 0
B) 1
C) 2
D) 3

Soru

Bir harita (map) veri yapısı ne tür bir ilişkilendirmeyi sağlar?

A) Anahtar-değer çiftlerini
B) İndeksli elemanları
C) Sıralı elemanları
D) Rastgele erişimi

Soru

Bir kuyruk (queue) veri yapısı hangi mantığa göre çalışır?

A) FIFO (First-In, First-Out)
B) LIFO (Last-In, First-Out)
C) Priority (Öncelik)
D) Rastgele

Soru

Bir yığıt (stack) veri yapısı üzerinde yapılan "pop" işlemi neyi geri döndürür?

A) En üstteki elemanı
B) En alttaki elemanı
C) Rasgele bir elemanı
D) En büyük elemanı

Soru

Hangi veri yapısı elemanları bir sıraya göre saklar ve elemanların çıkartılma sırası önemlidir?

A) Küme (Set)

B) Harita (Map)

C) Ağaç (Tree)

D) Yığın (Stack)

SON