BLM267

Bölüm 2: Veri Yapıları ve Algoritmalara Giriş

C Kullanarak Veri Yapıları, İkinci Baskı
Reema Thareja

- TEMEL TERMİNOLOJİ
- VERİ YAPILARININ SINIFLANDIRILMASI
- VERİ YAPILARI ÜZERİNDEKİ İŞLEMLER
- ÖZET VERİ TÜRÜ
- ALGORİTMALAR
- ALGORİTMA TASARLAMAYA FARKLI YAKLAŞIMLAR
- ALGORİTMALARDA KULLANILAN KONTROL YAPILARI
- ZAMAN VE MEKÂN KARMAŞIKLIĞI

C Kullanarak Veri Yapıları, İkinci Baskı
Reema Thareja

TEMEL TERMİNOLOJİ

- C dilinde basit programların nasıl yazılacağını, hata ayıklanacağını ve çalıştırılacağını biliyoruz.
- Amacımız iyi programlar tasarlamak olmuştur. İyi program, iyi bir programı, iyi bir program olarak tanımlamaktadır.
  - doğru şekilde çalışır
  - okunması ve anlaşılması kolaydır
  - hata ayıklaması kolaydır
  - değiştirilmesi kolaydır

TEMEL TERMİNOLOJİ

- Bir programın doğru sonuçlar vermesi gerekir, ancak bunun yanında verimli bir şekilde çalışması da gerekir.
- Bir programın en az zamanda ve en az bellek alanıyla çalışması verimli olarak adlandırılır.
- Verimli programlar yazabilmek için belirli veri yönetimi kavramlarını uygulamamız gerekir.
- Veri yapısı, veri yönetiminin önemli bir parçasıdır.
- Veri yapısı, temel olarak bir isim altında bir araya getirilen veri öğeleri grubudur ve bir bilgisayarda verilerin verimli bir şekilde kullanılabilmesi için belirli bir şekilde depolanması ve düzenlenmesini tanımlar.

TEMEL TERMİNOLOJİ

- Veri yapıları hemen hemen her programda veya yazılım sisteminde kullanılır.
- Veri yapılarının bazı yaygın örnekleri diziler, bağlı listeler, kuyruklar, yığınlar, ikili ağaçlar ve karma tablolardır.
- Bilgiyi temsil etmek bilgisayar biliminin temelini oluşturur.
- Bir program veya yazılımın temel amacı hesaplamalar veya işlemler yapmak değil, bilgileri mümkün olduğunca hızlı bir şekilde depolamak ve geri çağırmaktır.

TEMEL TERMİNOLOJİ

- Uygun bir veri yapısının uygulanması en verimli çözümü sağlar.
- Bir çözümün, verileri depolamak için kullanılabilen toplam alan ve her alt görevi gerçekleştirmek için gereken zaman gibi gerekli kaynak kısıtlamaları içerisinde sorunu çözüyorsa verimli olduğu söylenir.
- Ve en iyi çözüm, bilinen alternatiflerden daha az kaynak gerektiren çözümdür.
- Ayrıca bir çözümün maliyeti tükettiği kaynak miktarıdır.
- Bir çözümün maliyeti temel olarak zaman gibi bir temel kaynak açısından ölçülür ve çözümün diğer kaynak kısıtlamalarını da karşıladığı varsayılır.

TEMEL TERMİNOLOJİ

- Günümüzde bilgisayar programcıları yalnızca bir problemi çözmek için değil, aynı zamanda verimli bir program yazmak için program yazmaktadırlar.
- Bunun için öncelikle problemi analiz edip ulaşılması gereken performans hedeflerini belirlerler ve ardından bu iş için en uygun veri yapısını düşünürler.
- Ancak veri yapısı kavramları konusunda yetersiz bilgiye sahip program tasarımcıları bu analiz adımını göz ardı edip, kendilerinin rahatça çalışabileceği bir veri yapısını uygularlar.
- Uygulanan veri yapısı, ele alınan problem için uygun olmayabilir ve bu nedenle düşük performans (örneğin, işlem hızının yavaş olması) ile sonuçlanabilir.

TEMEL TERMİNOLOJİ

- Eğer bir program performans hedeflerini kullanımı kolay bir veri yapısıyla karşılıyorsa, sadece programcının becerisini sergilemek adına başka bir karmaşık veri yapısı uygulamanın bir anlamı yoktur.
- Bir problemi çözmek için veri yapısı seçilirken aşağıdaki adımların gerçekleştirilmesi gerekir.
  - Desteklenmesi gereken temel işlemleri belirlemek için sorunun analizi. Örneğin, temel işlem veri yapısından bir veri öğesini ekleme/silme/arama içerebilir.
  - Her operasyon için kaynak kısıtlamalarını niceliksel olarak belirleyin.
  - Bu gereksinimleri en iyi karşılayan veri yapısını seçin.
- Ele alınan sorun için uygun veri yapısının seçilmesine yönelik bu üç adımlı yaklaşım, tasarım sürecinin veri merkezli bir görünümünü destekler.

TEMEL TERMİNOLOJİ

- Yaklaşımda ilk kaygı, veri ve bu veri üzerinde gerçekleştirilecek işlemlerdir.
- İkinci kaygı verinin temsilidir ve son kaygı da bu temsilin uygulanmasıdır.
- C dilinin desteklediği farklı tipte veri yapıları vardır.
- Bir veri yapısı türü yeni veri öğelerinin yalnızca başlangıçta eklenmesine izin verirken, diğeri herhangi bir pozisyonda eklenmesine izin verebilir.
- Bir veri yapısı veri öğelerine sıralı olarak erişime izin verirken, diğeri verilere rastgele erişime izin verebilir.
- Bu nedenle, probleme uygun veri yapısının seçimi kritik bir karardır ve programın performansı üzerinde büyük bir etkiye sahip olabilir.

TEMEL TERMİNOLOJİ

Temel Veri Yapısı Organizasyonu

- Veri yapıları bir programın yapı taşlarıdır.
- Uygun olmayan veri yapıları kullanılarak oluşturulan bir program beklendiği gibi çalışmayabilir.
- Bu nedenle bir programcı olarak, programımız için en uygun veri yapılarını seçmek zorunludur.
- Veri terimi bir değer veya değerler kümesini ifade eder.
- Bir değişkenin ya da sabitin değerini belirtir (örneğin; öğrencilerin notları, bir çalışanın adı, bir müşterinin adresi, pi sayısının değeri, vb.).

TEMEL TERMİNOLOJİ

Temel Veri Yapısı Organizasyonu

- Alt veri öğeleri olmayan veri öğeleri temel öğe olarak sınıflandırılırken, bir veya daha fazla alt veri öğesinden oluşan veri öğelerine grup öğesi adı verilir.
- Örneğin, bir öğrencinin adı üç alt maddeye ayrılabilir: ad, ikinci ad ve soyadı; ancak sicil numarası normalde tek bir madde olarak ele alınır.
- Bir kayıt, veri öğelerinin bir koleksiyonudur. Örneğin, elde edilen ad, adres, ders ve notlar bireysel veri öğeleridir.
- Ancak tüm bu veri öğeleri bir kayıt oluşturmak üzere bir araya getirilebilir.

TEMEL TERMİNOLOJİ

Temel Veri Yapısı Organizasyonu

- Dosya, birbiriyle ilişkili kayıtların bir koleksiyonudur.
- Örneğin bir sınıfta 60 öğrenci varsa, o zaman öğrencilere ait 60 kayıt vardır.
- Tüm bu ilgili kayıtlar bir dosyada saklanır.
- Benzer şekilde, bir organizasyonda çalışan tüm çalışanların bir dosyasını, bir şirketin tüm müşterilerinin bir dosyasını, tüm tedarikçilerin bir dosyasını vb. bulundurabiliriz.
- Ayrıca, bir dosyadaki her kayıt birden fazla veri öğesinden oluşabilir ancak belirli bir veri öğesinin değeri dosyadaki kaydı benzersiz şekilde tanımlar. Böyle bir veri öğesi K'ye birincil anahtar denir ve bu alandaki K1, K2 ... değerlerine anahtarlar veya anahtar değerleri denir.

TEMEL TERMİNOLOJİ

Temel Veri Yapısı Organizasyonu

- Örneğin, öğrencinin kayıt numarası, adı, adresi, dersi ve aldığı notları içeren kayıtta, alan kayıt numarası birincil anahtardır.
- Diğer alanlar (isim, adres, ders ve notlar) birincil anahtar olarak kullanılamaz, çünkü iki veya daha fazla öğrencinin adı veya adresi aynı olabilir (aynı yerde kalıyor olabilirler), aynı derse kayıtlı olabilirler veya aynı notları almış olabilirler.
- Verilerin bu organizasyonu ve hiyerarşisi, Bölüm 2.2'de tartışılan daha karmaşık veri yapıları türlerini oluşturmak için daha da ileri götürülür.

VERİ YAPILARININ SINIFLANDIRILMASI

- Veri yapıları genel olarak iki sınıfa ayrılır: ilkel ve ilkel olmayan veri yapıları.
- İlkel veri yapıları, bir programlama dilinin desteklediği temel veri tipleridir.
- Bazı temel veri tipleri tam sayı, gerçek sayı, karakter ve boolean'dır.
- ‘Veri türü’, ‘temel veri türü’ ve ‘ilkel veri türü’ terimleri sıklıkla birbirinin yerine kullanılır.
- İlkel olmayan veri yapıları, ilkel veri yapıları kullanılarak oluşturulan veri yapılarıdır.
- Bu tür veri yapılarına örnek olarak bağlı listeler, yığınlar, ağaçlar ve grafikler verilebilir.

VERİ YAPILARININ SINIFLANDIRILMASI

Doğrusal ve Doğrusal Olmayan Yapılar

- İlkel olmayan veri yapıları da iki kategoriye ayrılabilir: doğrusal ve doğrusal olmayan veri yapıları.
- Bir veri yapısının elemanları doğrusal veya sıralı bir düzende saklanıyorsa bu durumda doğrusal veri yapısıdır.
- Örnek olarak diziler, bağlı listeler, yığınlar ve kuyruklar verilebilir.
- Doğrusal veri yapıları bellekte iki farklı şekilde temsil edilebilir.
- Bir yol, ardışık bellek konumları aracılığıyla öğeler arasında doğrusal bir ilişki kurmaktır.
- Diğer yol ise elemanlar arasında bağlantılar vasıtasıyla doğrusal bir ilişki kurmaktır.

VERİ YAPILARININ SINIFLANDIRILMASI

Doğrusal ve Doğrusal Olmayan Yapılar

- Ancak bir veri yapısının elemanları sıralı bir düzende saklanmıyorsa bu durumda doğrusal olmayan bir veri yapısıdır.
- Doğrusal olmayan bir veri yapısının elemanları arasında bitişiklik ilişkisi korunmaz.
- Örnek olarak ağaçlar ve grafikler verilebilir. C çeşitli veri yapılarını destekler.
- Şimdi tüm bu veri yapılarını tanıtacağız ve bunlar sonraki bölümlerde detaylı olarak ele alınacaktır.

VERİ YAPILARININ SINIFLANDIRILMASI

Diziler

- Dizi, benzer veri öğelerinin bir koleksiyonudur.
- Bu veri elemanlarının veri tipleri aynıdır.
- Dizinin elemanları ardışık bellek konumlarında saklanır ve bir dizinle (ayrıca dizin olarak da bilinir) referans alınır.
- C'de diziler şu sözdizimini kullanarak bildirilir: tür adı[boyut];
- Örneğin, int marks[10]; Yukarıdaki ifade, 10 öğe içeren marks dizisini bildirir.
- C'de dizi indeksi sıfırdan başlar.
- Bu, işaret dizisinin toplam 10 eleman içereceği anlamına gelir.
- İlk öğe marks[0]'da, ikinci öğe marks[1]'de, vb. saklanacaktır.
- Bu nedenle son eleman, yani 10. eleman marks[9]'da saklanacaktır. Bellekte dizi, Şekil 2.1'de gösterildiği gibi saklanacaktır.

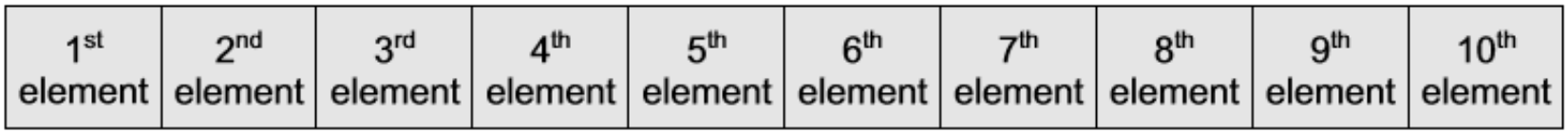

Figure 2.1 Memory representation of an array of 10 elements

VERİ YAPILARININ SINIFLANDIRILMASI

Diziler

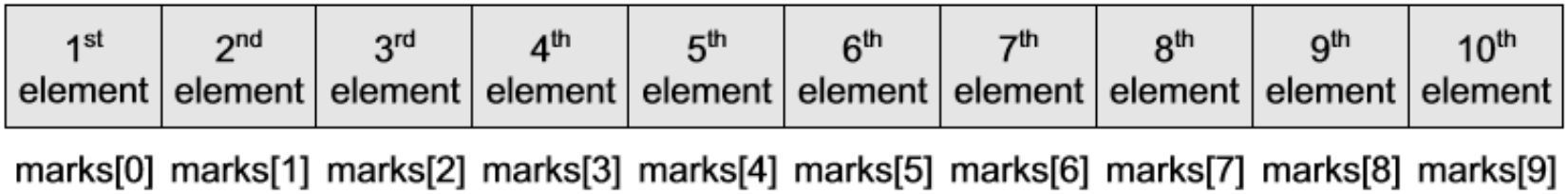

Figure 2.1 Memory representation of an array of 10 elements

- **Diziler genellikle büyük miktarda benzer türde veriyi depolamak istediğimizde kullanılır.**
- **Ancak bunların şu sınırlamaları vardır:**
  - **Diziler sabit boyuttadır.**
  - **Veri öğeleri, her zaman erişilebilir olmayabilen bitişik bellek konumlarında saklanır.**
  - **Elemanların yerlerinden kayması nedeniyle eleman ekleme ve çıkarma işlemleri sorunlu olabilir.**
- **Ancak bu sınırlamalar bağlı listeler kullanılarak aşılabilir.**
- **Diziler hakkında daha detaylı bilgiyi 3. Bölümde ele alacağız.**

Bağlantılı Listeler

- Bağlantılı liste, elemanların (düğüm adı verilir) ardışık bir liste oluşturduğu çok esnek, dinamik bir veri yapısıdır.
- Statik dizilerin aksine, bir programcının bağlı listede kaç eleman saklanacağı konusunda endişelenmesine gerek yoktur.
- Bu özellik programcıların daha az bakım gerektiren sağlam programlar yazmasına olanak tanır.
- Bağlantılı listede, her düğüm listeye eklendikçe ona alan tahsis edilir.

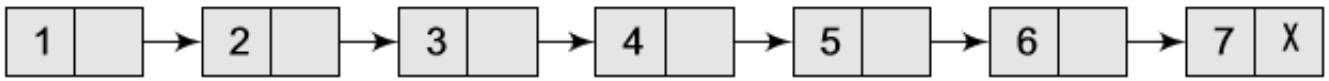

Figure 2.2 Simple linked list

Bağlantılı Listeler

- Listedeki her düğüm, listedeki bir sonraki düğüme işaret eder.
  - Bu nedenle, bağlantılı bir listede her düğüm aşağıdaki iki tür veriyi içerir: Düğümün değeri veya o düğüme karşılık gelen herhangi bir diğer veri
  - Listedeki bir sonraki düğüme bir işaretçi veya bağlantı
- Listedeki son düğüm, listenin sonu veya sonu olduğunu belirtmek için bir NULL işaretçisi içerir.
- Bir düğümün belleği, listeye eklendiğinde dinamik olarak tahsis edildiğinden, listeye eklenebilecek toplam düğüm sayı yalnızca kullanılabilir bellek miktarıyla sınırlıdır.
- Ş liste g

Figure 2.2 Simple linked list

VERİ YAPILARININ SINIFLANDIRILMASI

Yığınlar

- Yığın, elemanların eklenmesi ve çıkarılmasının yalnızca yığının en üst kısmı olarak bilinen bir uçtan yapıldığı doğrusal bir veri yapısıdır.
- Yığına son eklenen eleman, yığından silinen ilk eleman olduğu için, yığına son giren ilk çıkar (LIFO) yapısı denir.
- Bilgisayarın belleğinde yığınlar diziler veya bağlı listeler kullanılarak uygulanabilir.
- Şekil 2.3 bir yığının dizi uygulamasını göstermektedir.
- Her yığının kendisine ilişkilendirilmiş bir top değişkeni vardır. top, yığının en üst elemanının adresini depolamak için kullanılır.
- Elemanın ekleneceği veya silineceği konum burasıdır.
- Yığının depolayabileceği maksimum eleman sayısını depolamak için kullanılan bir de MAX değişkeni vardır.
- Eğer top = NULL ise, bu yığının boş olduğunu, eğer top = MAX–1 ise yığının dolu olduğunu gösterir.

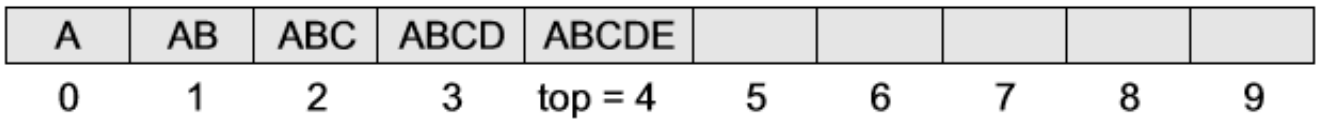

Figure 2.3 Array representation of a stack

VERİ YAPILARININ SINIFLANDIRILMASI

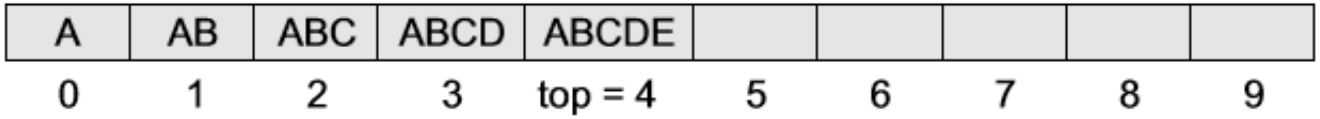

Figure 2.3 Array representation of a stack

Yığınlar

- Şekil 2.3'te top = 4 olduğundan ekleme ve çıkarmalar bu pozisyondan yapılacaktır.
- Burada yığın, endeksleri 0-9 arasında değişen en fazla 10 öğeyi depolayabilir.
- Yukarıdaki yığında beş eleman daha saklanabilir.
- Bir yığın üç temel işlemi destekler: itme, çıkarma ve gözetleme.
- İtme işlemi, bir elemanı yığının en üstüne ekler.
- Pop işlemi öğeyi yığının en üstünden kaldırır.
- Ve peep işlemi yığının en üst elemanının değerini (silinmeden) döndürür.
- Ancak yığına bir eleman eklemeden önce taşma koşullarının olup olmadığını kontrol etmemiz gerekir.
- Taşma, zaten dolu olan bir yığına bir eleman eklemeye çalıştığımızda meydana gelir.
- Benzer şekilde, yığından bir elemanı silmeden önce, alt taşma koşullarını kontrol etmeliyiz.
- Boş olan bir yığından bir öğeyi silmeye çalıştığımızda taşma durumu oluşur.

VERİ YAPILARININ SINIFLANDIRILMASI

Kuyruklar

- Kuyruk, ilk giren ilk çıkar (FIFO) veri yapısıdır; bu yapıda, ilk eklenen öğe ilk çıkarılır.
- Bir kuyruktaki elemanlar arka denilen bir uçtan eklenir ve ön denilen diğer uçtan çıkarılır.
- Yığınlar gibi kuyruklar da diziler veya bağlı listeler kullanılarak uygulanabilir.
- Her kuyruğun, sırasıyla silme ve ekleme işlemlerinin yapılabileceği konumu işaret eden ön ve arka değişkenleri vardır.
- Şekil 2.4'te görülen kuyruğu ele alalım.

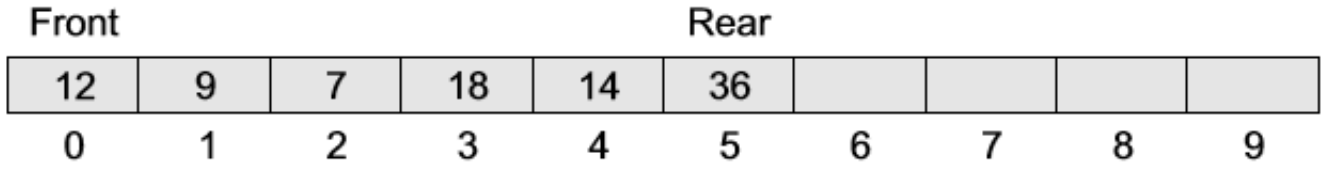

Figure 2.4 Array representation of a queue

VERİ YAPILARININ SINIFLANDIRILMASI

Kuyruklar

- Burada ön = 0 ve arka = 5.
- Listeye bir değer daha eklemek istediğimizde, diyelim ki 45 değerine sahip bir eleman daha eklemek istediğimizde, rear değeri 1 arttırılır ve değer, rear'ın işaret ettiği konumda saklanır.
- Ekleme sonrasında kuyruk Şekil 2.5'teki gibi olacaktır.
- Burada ön = 0 ve arka = 6.
- Her yeni bir eleman ekleneceği zaman aynı işlemi tekrarlayacağız.

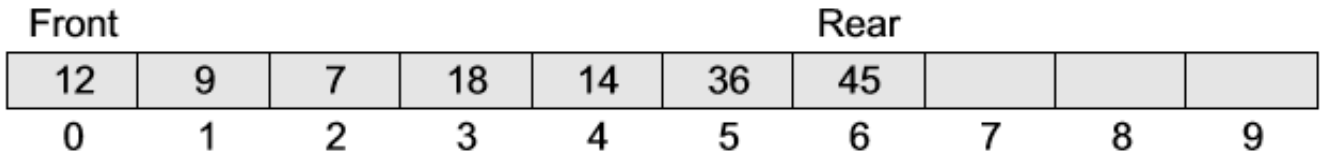

Figure 2.5 Queue after insertion of a new element

VERİ YAPILARININ SINIFLANDIRILMASI

Kuyruklar

- Şimdi eğer kuyruktan bir eleman silmek istersek front'un değeri artırılacaktır.
- Silmeler yalnızca kuyruğun bu ucundan yapılır.
- Silinme işleminden sonraki kuyruk Şekil 2.6'daki gibi olacaktır.

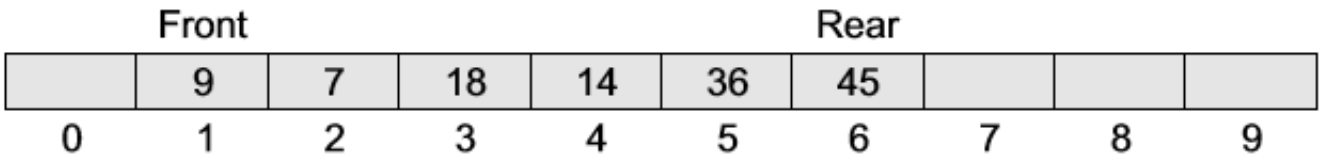

Figure 2.6 Queue after deletion of an element

VERİ YAPILARININ SINIFLANDIRILMASI

- Ancak kuyruğa bir eleman eklemeden önce taşma koşullarını kontrol etmemiz gerekir.
- Taşma, zaten dolu olan bir kuyruğa bir öğe eklemeye çalıştığımızda oluşur.
- Bir kuyruk, rear = MAX – 1 olduğunda doludur; burada MAX kuyruğun boyutudur, yani MAX kuyruktaki maksimum eleman sayısını belirtir.
- Dikkat ederseniz MAX – 1 yazdık çünkü indeks 0'dan başlıyor.
- Benzer şekilde, kuyruktan bir öğeyi silmeden önce, alt taşma koşullarını kontrol etmeliyiz.
- Bir kuyruktan boş olan bir öğeyi silmeye çalıştığımızda alt taşma durumu oluşur.
- Eğer front = NULL ve rear = NULL ise kuyrukta eleman yoktur.

VERİ YAPILARININ SINIFLANDIRILMASI

Ağaçlar

- Ağaç, hiyerarşik bir düzende düzenlenmiş düğümlerin bir araya gelmesiyle oluşan doğrusal olmayan bir veri yapısıdır.
- Düğümlerden biri kök düğüm olarak belirlenir ve kalan düğümler, her kümenin kökün bir alt ağacı olacağı şekilde ayrık kümelere bölünebilir.
- Ağacın en basit hali ikili ağaçtır.
- İkili ağaç, bir kök düğümden ve sol ve sağ alt ağaçlardan oluşur; her iki alt ağaç da ikili ağaçtır.

VERİ YAPILARININ SINIFLANDIRILMASI

Ağaçlar

- Her düğüm bir veri elemanı, sol alt ağacı işaret eden sol işaretçi ve sağ alt ağacı işaret eden sağ işaretçi içerir.
- Kök eleman, bir ‘kök’ işaretçisi tarafından işaret edilen en üst düğümdür.
- Eğer kök = NULL ise ağaç boştur.
- Şekil 2.7, R'nin kök düğüm ve T1 ve T2'nin R'nin sol ve sağ alt ağaçları olduğu ikili bir ağacı göstermektedir.
- Eğer T1 boş değilse, o zaman T1'in R'nin sol ardılı olduğu söylenir.
- Benzer şekilde eğer T2 boş değilse, o zaman R'nin sağ ardılı olarak adlandırılır.

VERİ YAPILARININ SINIFLANDIRILMASI

Ağaçlar

- Şekil 2.7'de, düğüm 2 kök düğüm 1'in sol çocuğu, düğüm 3 ise sağ çocuğudur.
- Kök düğümün sol alt ağacının 2, 4, 5, 8 ve 9 düğümlerinden oluştuğunu unutmayın.
- Benzer şekilde kök düğümün sağ alt ağacı 3, 6, 7, 10, 11 ve 12 düğümlerinden oluşur.

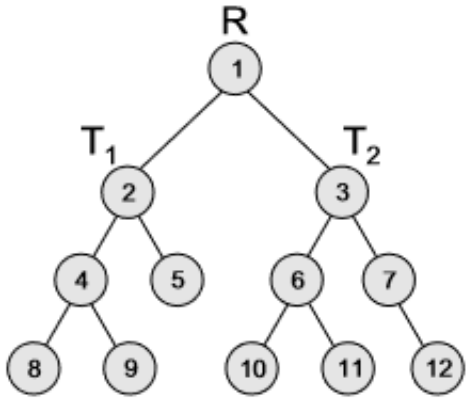

Figure 2.7 Binary tree

VERİ YAPILARININ SINIFLANDIRILMASI

Grafikler

- Bir grafik, köşelerin (node'lar olarak da bilinir) ve bu köşeleri birbirine bağlayan kenarların bir araya gelmesiyle oluşan doğrusal olmayan bir veri yapısıdır.
- Bir grafik genellikle ağaç yapısının genelleştirilmiş hali olarak görülür; burada ağaç düğümleri arasında salt ebeveyn-çocuk ilişkisi yerine, düğümler arasında her türlü karmaşık ilişki bulunabilir.

VERİ YAPILARININ SINIFLANDIRILMASI

Grafikler

- Bir ağaç yapısında, düğümlerin herhangi sayıda çocuğu olabilir ancak yalnızca bir ebeveyni olabilir; öte yandan bir grafik bu tür tüm kısıtlamaları gevşetir.
- Şekil 2.8 beş düğümlü bir grafiği göstermektedir.
- Grafikteki bir düğüm bir şehri, düğümleri birbirine bağlayan kenarlar ise yolları temsil edebilir.
- Bir grafik, düğümlerin iş istasyonları ve kenarların ağ bağlantıları olduğu bir bilgisayar ağını temsil etmek için de kullanılabilir.

VERİ YAPILARININ SINIFLANDIRILMASI

Grafikler

- Grafiklerin bilgisayar bilimi ve matematikte o kadar çok uygulaması vardır ki, grafiği arama ve grafiğin düğümleri arasındaki en kısa yolu bulma gibi standart grafik işlemlerini gerçekleştirmek için birçok algoritma yazılmıştır.
- Ağaçların aksine grafiklerin herhangi bir kök düğümü olmadığını unutmayın.
- Aksine, grafikteki her düğüm, grafikteki diğer her düğüme bağlanabilir.
- İki düğüm bir kenar aracılığıyla bağlandığında, bu iki düğüme komşu denir.
- Örneğin, Şekil 2.8'de, A düğümünün iki komşusu vardır: B ve D.

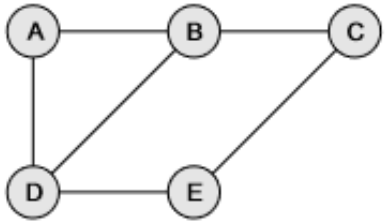

Figure 2.8 Graph

VERİ YAPILARI ÜZERİNDEKİ İŞLEMLER

- **Bu bölümde daha önce bahsedilen çeşitli veri yapıları üzerinde gerçekleştirilebilecek farklı işlemler ele alınmaktadır.**
- ***Gezinme: Her veri öğesine tam olarak bir kez erişerek işlenebilmesi anlamına gelir. Örneğin, bir sınıftaki tüm öğrencilerin adlarını yazdırmak.***
- ***Arama: Verilen kısıtlamayı karşılayan bir veya daha fazla veri öğesinin yerini bulmak için kullanılır. Böyle bir veri öğesi verilen veri öğeleri koleksiyonunda mevcut olabilir veya olmayabilir. Örneğin, matematikte 100 puan alan tüm öğrencilerin adlarını bulmak için.***
- ***Ekleme: Verilen veri öğeleri listesine yeni veri öğeleri eklemek için kullanılır. Örneğin, kursa yeni katılan bir öğrencinin ayrıntılarını eklemek için.***

VERİ YAPILARI ÜZERİNDEKİ İŞLEMLER

- ***Silme: Belirli bir veri öğesini verilen veri öğeleri koleksiyonundan kaldırmak (silmek) anlamına gelir. Örneğin, dersten ayrılan bir öğrencinin adını silmek.***
- ***Sıralama: Veri öğeleri, uygulama türüne bağlı olarak artan veya azalan düzende düzenlenebilir. Örneğin, bir sınıftaki öğrencilerin adlarını alfabetik sıraya göre düzenlemek veya katılımcıların puanlarını azalan sıraya göre düzenleyerek ilk üç kazananı hesaplamak ve ardından ilk üçü çıkarmak.***
- ***Birleştirme: İki sıralanmış veri öğesinin listesi birleştirilerek tek bir sıralanmış veri öğesi listesi oluşturulabilir.***
- **Birçok kez, belirli bir durumda iki veya daha fazla işlem aynı anda uygulanır. Örneğin, adı X olan bir öğrencinin ayrıntılarını silmek istiyorsak, önce X'in kaydının mevcut olup olmadığını ve mevcutsa hangi konumda olduğunu bulmak için öğrenci listesini aramamız gerekir, böylece ayrıntılar o belirli konumdan silinebilir.**

ÖZET VERİ TÜRÜ

- Soyut veri türü (ADT), bir veri yapısına bakış şeklimizdir; ne yaptığına odaklanırız ve işini nasıl yaptığını göz ardı ederiz.
- Örneğin, yığınlar ve kuyruklar ADT'nin mükemmel örnekleridir.
- Bu iki ADT'yi bir dizi veya bağlı liste kullanarak uygulayabiliriz.
- Bu, yığınların ve kuyrukların ‘soyut’ doğasını göstermektedir.
- Soyut veri türünün anlamını daha iyi anlamak için terimi ‘veri türü’ ve ‘soyut’ olarak ikiye ayıracağız ve ardından anlamlarını tartışacağız.
- Veri türü Bir değişkenin veri türü, değişkenin alabileceği değerler kümesidir.

ÖZET VERİ TÜRÜ

- C'deki temel veri tiplerini daha önce okumuştuk: int, char, float ve double.
- İlkel tipten (dahili veri tipi) bahsettiğimizde aslında iki şeyi düşünürüz: Belirli özelliklere sahip bir veri öğesi ve bu veri üzerinde yapılabilecek izin verilen işlemler.
- Örneğin, bir int değişkeni -32768 ile 32767 arasındaki herhangi bir tam sayı değerini içerebilir ve +, –, * ve / operatörleriyle çalıştırılabilir.
- Başka bir deyişle, bir veri türü üzerinde gerçekleştirilebilecek işlemler, o veri türünün kimliğinin ayrılmaz bir parçasıdır.
- Bu nedenle, soyut bir veri tipinde (örneğin, yığın veya kuyruk) bir değişken bildirdiğimizde, üzerinde gerçekleştirilebilecek işlemleri de belirtmemiz gerekir.

ÖZET VERİ TÜRÜ

- Özet Veri yapıları bağlamında ‘soyut’ kelimesi, ayrıntılı özelliklerden veya uygulamadan ayrı olarak düşünülen anlamına gelir.
- C'de soyut bir veri türü, uygulamasına bakılmaksızın ele alınan bir yapı olabilir.
- Yapıdaki verilerin bir ‘tanımı’ olarak düşünülebilir ve bu yapı içindeki veriler üzerinde gerçekleştirilebilecek işlemlerin bir listesi olarak düşünülebilir.
- Son kullanıcı, yöntemlerin görevlerini nasıl yerine getirdiğinin ayrıntılarıyla ilgilenmez.
- Sadece kendilerine sunulan yöntemlerin farkındadırlar ve sadece bu yöntemleri çağırıp sonuçlarını almakla ilgilenirler.

ÖZET VERİ TÜRÜ

- Nasıl çalıştıkları konusunda endişeleri yok.
- Örneğin bir yığın veya kuyruk kullandığımızda kullanıcı sadece verinin türü ve üzerinde yapılabilecek işlemlerle ilgilenir.
- Dolayısıyla verinin nasıl saklanacağına dair temel bilgiler kullanıcı tarafından görülemez olmalıdır.
- Yöntemlerin nasıl çalıştığı veya verilerin depolanmasında hangi yapıların kullanıldığıyla ilgilenmemeliler.
- Sadece yığınlarla çalışmak için push() ve pop() fonksiyonlarının mevcut olduğunu bilmeleri gerekir.
- Bu fonksiyonları kullanarak yığında saklanan veriler üzerinde değişiklik (ekleme veya silme) yapabilirler.

ÖZET VERİ TÜRÜ

- ADT'leri kullanmanın avantajı Gerçek dünyada, programlar yeni gereksinimler veya kısıtlamalar sonucunda gelişir, bu nedenle bir programda yapılan bir değişiklik genellikle veri yapılarından birinde veya daha fazlasında değişiklik yapılmasını gerektirir.
- Örneğin, her öğrenci hakkında daha fazla bilgi tutmak için öğrencinin kaydına yeni bir alan eklemek istiyorsanız, programın verimliliğini artırmak için diziyi bağlantılı bir yapı ile değiştirmek daha iyi olacaktır.
- Böyle bir senaryoda, değiştirilen yapıyı kullanan her prosedürün yeniden yazılması istenmez.
- Bu nedenle, daha iyi bir alternatif, veri yapısının kullanımını, onun uygulanmasının ayrıntılarından ayırmaktır.
- Soyut veri tiplerinin kullanımının altında yatan prensip budur.

ALGORİTMALAR

- Algoritmanın tipik tanımı ‘bazı hesaplamaları gerçekleştirmek için resmen tanımlanmış bir prosedür’dür.
- Bir prosedür resmi olarak tanımlanmışsa, o zaman resmi bir dil kullanılarak uygulanabilir ve böyle bir dile programlama dili denir.
- Genel anlamda bir algoritma, belirli bir problemi çözmek için bir program yazmak için bir plan sunar.
- Sonlu sayıda adımda bir problemi çözmek için etkili bir yöntem olduğu düşünülmektedir.
- Yani iyi tanımlanmış bir algoritma her zaman bir cevap verir ve sonlanması garantilidir.

ALGORİTMALAR

- Algoritmalar esas olarak yazılımın yeniden kullanılabilirliğini sağlamak amacıyla kullanılır.
- Bir çözüm fikrimiz veya taslağımız olduğunda bunu C, C++ veya Java gibi herhangi bir üst düzey dilde uygulayabiliriz.
- Algoritma, temel olarak bir problemi çözmeye yönelik bir dizi talimattır.
- Aynı problemi çözmek için birden fazla algoritma kullanılması olağandışı değildir, ancak belirli bir algoritmanın seçimi, algoritmanın zaman ve mekan karmaşıklığına bağlı olmalıdır.

ALGORİTMA TASARLAMAYA FARKLI YAKLAŞIMLAR

- Algoritmalar, veri yapılarında yer alan verileri işlemek için kullanılır.
- Veri yapılarıyla çalışılırken, saklanan veriler üzerinde işlemler gerçekleştirmek için algoritmalar kullanılır.
- Karmaşık bir algoritma genellikle modül adı verilen daha küçük birimlere bölünür.
- Bir algoritmayı modüllere ayırma işlemine modülerleştirme denir.

ALGORİTMA TASARLAMAYA FARKLI YAKLAŞIMLAR

- **Modülerleştirmenin temel avantajları şunlardır:**
  - **Karmaşık algoritmaların tasarlanmasını ve uygulanmasını daha basit hale getirir.**
  - **Her modül bağımsız olarak tasarlanabilir.**
  - **Bir modülü tasarlarken diğer modüllerin ayrıntıları göz ardı edilebilir, böylece tasarımda netlik artar ve bu da genel algoritmanın uygulanmasını, hata ayıklamasını, test edilmesini, belgelenmesini ve bakımını basitleştirir.**

ALGORİTMA TASARLAMAYA FARKLI YAKLAŞIMLAR

- **Bir algoritma tasarlamak için iki temel yaklaşım vardır: Şekil 2.9'da gösterildiği gibi yukarıdan aşağıya yaklaşım ve aşağıdan yukarıya yaklaşım.**

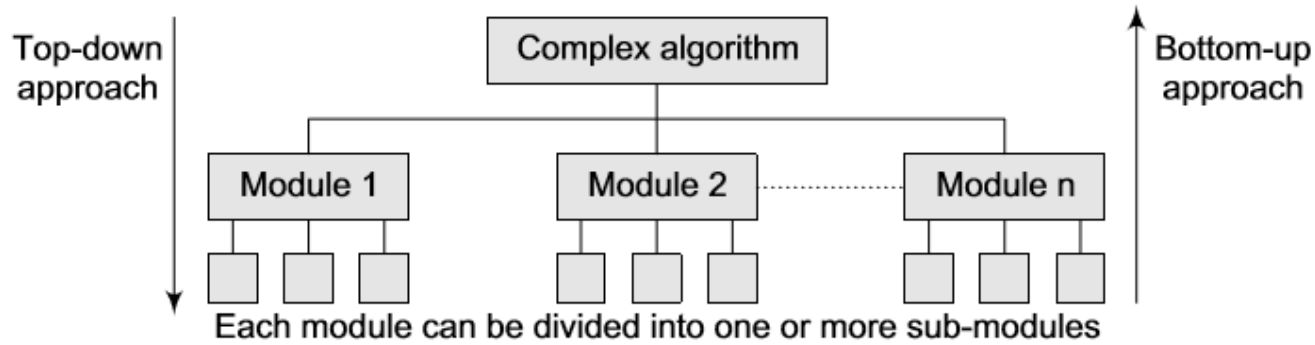

Figure 2.9 Different approaches of designing an algorithm

Yukarıdan aşağıya yaklaşım

- Yukarıdan aşağıya tasarım yaklaşımı, karmaşık algoritmayı bir veya daha fazla modüle bölerek başlar.
- Bu modüller daha sonra bir veya daha fazla alt modüle ayrıştırılabilir ve bu ayrıştırma sür istenen modül karmaşıklığı düzeyine ulaşılana kadar yinelenir.
- Yukarıdan aşağıya tasarım yöntemi, en üstteki modülden başlayıp, onun çağırdığı modüll kademeli olarak eklediğimiz bir tür aşamalı iyileştirmedir.
- Dolayısıyla yukarıdan aşağıya yaklaşımda, soyut bir tasarımdan başlıyoruz ve daha sonra adımda bu tasarım daha somut seviyelere doğru rafine ediliyor ve daha fazla rafine etme gerektirmeyen bir seviyeye ulaşılıyor.

ALGORİTMA TASARLAMAYA FARKLI YAKLAŞIMLAR

- **Bir algoritma tasarlamak için iki temel yaklaşım vardır: Şekil 2.9'da gösterildiği gibi yukarıdan aşağıya yaklaşım ve aşağıdan yukarıya yaklaşım.**

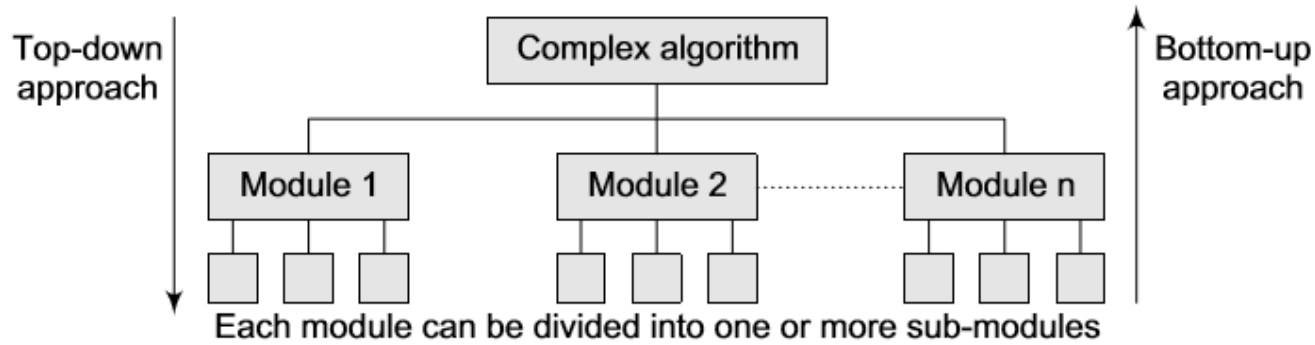

Figure 2.9 Different approaches of designing an algorithm

Aşağıdan yukarıya yaklaşım

- Aşağıdan yukarıya yaklaşım, yukarıdan aşağıya yaklaşımın tam tersidir. Aşağıdan yukarı tasarımda, en temel veya somut modülleri tasarlamakla başlarız ve ardından daha yükse seviyeli modülleri tasarlamak için ilerleriz.
- Daha yüksek seviyeli modüller, daha düşük seviyeli modüllerin gerçekleştirdiği işlemleri kullanarak uygulanır.
- Bu yaklaşımda alt modüller bir araya getirilerek daha üst seviye bir modül oluşturulur.
- Tüm üst seviye modüller bir araya getirilerek daha da üst seviye modüller oluşturulur.
- Bu işlem algoritmanın tam tasarımı elde edilinceye kadar tekrarlanır.

Yukarıdan aşağıya ve aşağıdan yukarıya yaklaşım

- Yukarıdan aşağıya mı yoksa aşağıdan yukarıya mı strateji izlenmesi gerektiği, eldeki uygulamaya bağlı olarak cevaplanabilecek bir sorudur.
- Yukarıdan aşağıya yaklaşım algoritmayı yönetilebilir modüllere ayırarak adım adım bir iyileştirmeyi takip ederken, aşağıdan yukarıya yaklaşım ise bir modülü tanımlar ve daha sonra birkaç modülü bir araya getirerek yeni, daha üst seviye bir modül oluşturur.

ALGORİTMA TASARLAMAYA FARKLI YAKLAŞIMLAR

Yukarıdan aşağıya ve aşağıdan yukarıya yaklaşım

- Yukarıdan aşağıya yaklaşım, modüllerin belgelendirilmesi, test vakalarının oluşturulması, kodun uygulanması ve hata ayıklamada kolaylık sağlaması nedeniyle oldukça beğenilmektedir.
- Ancak alt modüllerin, diğer modüllerle iletişimine veya bileşenlerin yeniden kullanılabilirliğine yoğunlaşılmadan izole bir şekilde analiz edilmesi ve verilere çok az dikkat edilmesi, böylece bilgi gizleme kavramının göz ardı edilmesi nedeniyle de eleştirilmektedir.

ALGORİTMA TASARLAMAYA FARKLI YAKLAŞIMLAR

Yukarıdan aşağıya ve aşağıdan yukarıya yaklaşım

- Aşağıdan yukarıya yaklaşım, öncelikle bir modülün içinde neyin kapsüllenmesi gerektiğini tanımlayıp daha sonra istemcilerden görüldüğü gibi modülün sınırlarını tanımlayan soyut bir arayüz sağladığı için bilgi gizlemeye izin verir.
- Ancak tüm bunların katı bir aşağıdan yukarıya stratejiyle yapılması zordur.
- Bunun için yukarıdan aşağıya doğru bazı faaliyetlerin yapılması gerekiyor.
- Sonuç olarak, karmaşık algoritmaların tasarımı sabit bir kalıba göre ilerlemekle sınırlandırılmamalı, yukarıdan aşağıya ve aşağıdan yukarıya yaklaşımların bir karışımı olmalıdır.

ALGORİTMALARDA KULLANILAN KONTROL YAPILARI

- Bir algoritmanın sonlu sayıda adımı vardır.
- Bazı adımlar karar vermeyi ve tekrarı gerektirebilir.
- Geniş anlamda, bir algoritma aşağıdaki kontrol yapılarından birini kullanabilir: (a) dizi, (b) karar ve (c) tekrar.
- Sıra: Sıra ile, bir algoritmanın her adımının belirli bir sırayla yürütülmesi kastedilmektedir.
- İki sayıyı toplayan bir algoritma yazalım.
- Bu algoritma, Şekil 2.10'da gösterildiği gibi adımları tamamen ardışık bir düzende gerçekleştirir.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: SET SUM = A+B
Step 4: PRINT SUM
Step 5: END
```

Figure 2.10 Algorithm to add two numbers

ALGORİTMALARDA KULLANILAN KONTROL YAPILARI

Karar:

- Karar ifadeleri, bir sürecin yürütülmesinin bir koşulun sonucuna bağlı olduğu durumlarda kullanılır.
- Örneğin, eğer x = y ise, o zaman EŞİT yazdır. Dolayısıyla, IF yapısının genel biçimi şu şekilde verilebilir: IF koşulu O zaman işlem Bu bağlamda bir koşul, doğru veya yanlış bir değere değerlendirilebilecek herhangi bir ifadedir.
- Yukarıdaki örnekte, bir x değişkeni y'ye eşit veya y'ye eşit olmayabilir. Ancak, hem doğru hem de yanlış olamaz.
- Koşul doğru ise işlem yürütülür.

Karar:

- Bir karar ifadesi aşağıdaki şekilde de ifade edilebilir: Eğer koşul O zaman süreç1 DEĞİLSE süreç2 Bu form, halk arasında IF-ELSE yapısı olarak bilinir.
- Burada koşul doğru ise process1, doğru değilse process2 çalıştırılır.
- Şekil 2.11 iki sayının eşit olup olmadığını kontrol eden bir algoritmayı göstermektedir.

```
Step 1: Input first number as A
Step 2: Input second number as B
Step 3: IF A = B
              PRINT "EQUAL"
        ELSE
              PRINT "NOT EQUAL"
        [END OF IF]
Step 4: END
```

Figure 2.11 Algorithm to test for equality of two numbers

ALGORİTMALARDA KULLANILAN KONTROL YAPILARI

Tekrarlama:

- Bir veya daha fazla adımı belirli sayıda kez yürütmeyi içeren tekrarlama, while, do–while ve for döngüleri gibi yapılar kullanılarak uygulanabilir.
- Bu döngüler, bir koşul doğru olana kadar bir veya daha fazla adımı yürütür.
- Şekil 2.12 ilk 10 doğal sayıyı yazdıran bir algoritmayı göstermektedir.

```
Step 1: [INITIALIZE] SET I = 1, N = 10
Step 2: Repeat Steps 3 and 4 while I<=N
Step 3: PRINT I
Step 4: SET I = I+1
        [END OF LOOP]
Step 5: END
```

Figure 2.12 Algorithm to print the first 10 natural of

ZAMAN VE MEKÂN KARMAŞIKLIĞI

- Bir algoritmayı analiz etmek, onu çalıştırmak için gereken kaynak miktarını (zaman ve bellek gibi) belirlemek anlamına gelir.
- Algoritmalar genellikle keyfi sayıda girdiyle çalışmak üzere tasarlanır, bu nedenle bir algoritmanın verimliliği veya karmaşıklığı zaman ve mekan karmaşıklığı açısından ifade edilir.
- Bir algoritmanın zaman karmaşıklığı, temel olarak bir programın çalışma süresinin girdi büyüklüğüne bağlı bir fonksiyonudur.
- Benzer şekilde, bir algoritmanın uzay karmaşıklığı, programın yürütülmesi sırasında giriş boyutunun bir fonksiyonu olarak ihtiyaç duyulan bilgisayar belleği miktarıdır.
- Başka bir deyişle, bir programın yürüttüğü makine talimatlarının sayısına o programın zaman karmaşıklığı denir.
- Bu sayı öncelikle programın girdisinin büyüklüğüne ve

ZAMAN VE MEKÂN KARMAŞIKLIĞI

- Bir programın ihtiyaç duyduğu alan genellikle şu iki kısma bağlıdır:
  - Sabit parça: Problemden probleme değişir. Talimatları, sabitleri, değişkenleri ve yapılandırılmış değişkenleri (diziler ve yapılar gibi) depolamak için gereken alanı içerir.
  - Değişken kısım: Programdan programa değişir. Özyineleme yığını için gereken alanı ve bir programın çalışma zamanı sırasında dinamik olarak alan tahsis edilen yapılandırılmış değişkenler için gereken alanı içerir.
- Ancak çalışma zamanı gereksinimleri bellek gereksinimlerinden daha kritiktir.
- Bu nedenle bu bölümde algoritmaların çalışma zamanı verimliliğine yoğunlaşacağız.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

En Kötü Durum, Ortalama Durum, En İyi Durum ve Amortize Edilmiş Zaman Karmaşıklığı

- **En kötü durum çalışma süresi Bu, bir algoritmanın giriş örneğinin olası en kötü durumuna göre davranışını belirtir.**
- **Bir algoritmanın en kötü çalışma süresi, herhangi bir girdi için çalışma süresinin üst sınırıdır.**
- **Dolayısıyla, en kötü durum çalışma zamanının bilgisine sahip olmak, algoritmanın hiçbir zaman bu zaman sınırını aşmayacağı konusunda bize güvence verir.**
- **Ortalama durum çalışma süresi Bir algoritmanın ortalama durum çalışma süresi, 'ortalama' bir girdi için çalışma süresinin tahminidir.**
- **Girişin belirli bir dağılımdan rastgele çekilmesi durumunda algoritmanın beklenen davranışını belirtir.**

ZAMAN VE MEKÂN KARMAŞIKLIĞI

En Kötü Durum, Ortalama Durum, En İyi Durum ve Amortize Edilmiş Zaman Karmaşıklığı

- Ortalama durum çalışma süresi, belirli bir boyuttaki tüm girdilerin eşit olasılıkla gerçekleştiğini varsayar.
- En iyi durum çalışma süresi 'En iyi durum performansı' terimi, bir algoritmayı en iyi koşullar altında analiz etmek için kullanılır. Örneğin, bir dizide basit bir doğrusal arama için en iyi durum, istenen öğenin listede ilk sırada olması durumunda ortaya çıkar.
- Ancak bir problemi çözmek için bir algoritma geliştirirken ve seçerken kararımızı en iyi durum performansına dayandırmayız.
- Bir algoritmanın ortalama performansını ve en kötü durum performansını iyileştirmek her zaman önerilir.
- Amortize edilmiş çalışma süresi, gerçekleştirilen tüm işlemler üzerinden ortalaması alınan bir dizi (ilgili) işlemi gerçekleştirmek için gereken süreyi ifade eder.
- Amortize analiz, her bir operasyonun en kötü durumdaki ortalama performansını garanti eder.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

Zaman-Uzay Dengesi

- Belirli bir problemi çözmek için en iyi algoritma, hiç şüphesiz daha az bellek alanı gerektiren ve yürütülmesi daha az zaman alan algoritmadır.
- Ama pratikte böylesine ideal bir algoritmayı tasarlamak hiç de kolay bir iş değil.
- Belirli bir problemi çözmek için birden fazla algoritma olabilir.
- Biri daha az bellek alanı gerektirirken, diğeri yürütülürken daha az CPU süresi gerektirebilir.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

Zaman-Uzay Dengesi

- Dolayısıyla bir şeyi diğeri uğruna feda etmek pek de olağandışı bir durum değildir.
- Dolayısıyla algoritmalar arasında bir zaman-mekan alışverişi vardır.
- Yani, eğer alan büyük bir kısıtlama ise, o zaman daha fazla CPU zamanı pahasına daha az yer kaplayan bir program seçilebilir.
- Tam tersine, eğer zaman önemli bir kısıtlama ise, o zaman daha fazla alan kaplayarak yürütülmesi en az zamanı alan bir program seçilebilir.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

Zaman ve Mekan Karmaşıklığını İfade Etmek

- Zaman ve mekan karmaşıklığı, n'nin çözülen problemin belirli bir örneği için girdi boyutu olduğu f(n) fonksiyonu kullanılarak ifade edilebilir.
- Karmaşıklığın ifade edilmesi gerektiğinde
  - Problemin girdi boyutu arttıkça karmaşıklığın büyüme oranını tahmin etmek istiyoruz.
  - Belirli bir probleme çözüm bulan birden fazla algoritma vardır ve en verimli olan algoritmayı bulmamız gerekir.
- Bu fonksiyonu f(n) ifade etmek için en yaygın kullanılan gösterim Büyük O gösterimidir. Karmaşıklık için üst sınırı sağlar.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

Algoritma Verimliliği

- Eğer bir fonksiyon doğrusal ise (herhangi bir döngü veya yineleme içermiyorsa), o algoritmanın verimliliği veya çalışma süresi, içerdiği talimat sayısı olarak verilebilir.
- Ancak bir algoritma döngüler içeriyorsa, o algoritmanın verimliliği, döngü sayısına ve algoritmadaki her bir döngünün çalışma süresine bağlı olarak değişebilir.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

Algoritma Verimliliği

- Döngülerin bir algoritmanın verimliliğini belirlediği farklı durumları ele alalım.
- *Doğrusal Döngüler: Tek döngüsü olan bir algoritmanın verimliliğini hesaplamak için öncelikle döngüdeki ifadelerin kaç kez yürütüleceğini belirlememiz gerekir.*
- Çünkü yineleme sayısı döngü faktörüyle doğru orantılıdır.
- Döngü faktörü ne kadar büyükse, yineleme sayısı da o kadar fazla olur.
- Örneğin, aşağıda verilen döngüyü ele alalım:
  for(i=0;i<100;i++) statement block;
- Burada 100 döngü faktörüdür.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

Algoritma Verimliliği

- Verimliliğin yineleme sayısıyla doğru orantılı olduğunu daha önce söylemiştik.
- Bu nedenle, doğrusal döngüler durumunda genel formül f(n) = n olarak verilebilir
- Ancak verimliliği hesaplamak yukarıdaki örnekte gösterildiği kadar basit değildir.
- Aşağıda verilen döngüyü ele alalım: for(i=0;i<100;i+=2) ifade bloğu;
- Burada, yineleme sayısı döngü faktörünün sayısının yarısıdır. Bu nedenle, burada verimlilik f(n) = n/2 olarak verilebilir

ZAMAN VE MEKÂN KARMAŞIKLIĞI

Logaritmik Döngüler

- Doğrusal döngülerde döngü güncelleme ifadesinin döngüyü kontrol eden değişkeni eklediğini veya çıkardığını gördük.
- Ancak logaritmik döngülerde, döngüyü kontrol eden değişken, döngünün her yinelemesinde çarpılır veya bölünür.
- Örneğin, aşağıda verilen döngülere bir bakalım: for(i=1;i<1000;i*=2) for(i=1000;i>=1;i/=2) ifade bloğu; ifade bloğu;
- Döngüyü kontrol eden değişken i'nin 2 ile çarpıldığı ilk for döngüsünü ele alalım.
- Döngü 1000 kez değil sadece 10 kez çalıştırılacak çünkü her yinelemede i değeri iki katına çıkıyor.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

Logaritmik Döngüler

- Şimdi, döngüyü kontrol eden değişken i'nin 2'ye bölündüğü ikinci döngüyü ele alalım.
- Bu durumda da döngü 10 kere çalıştırılacaktır.
- Bu nedenle, yineleme sayısı, döngüyü kontrol eden değişkenin bölündüğü veya çarpıldığı sayının bir fonksiyonudur.
- Ele alınan örneklerde bu sayı 2'dir.
- Yani n = 1000 olduğunda yineleme sayısı log 1000 olarak verilebilir ki bu da yaklaşık olarak 10'a eşittir.
- Bu nedenle, bu analizi genel terimlerle ifade edersek, yinelemelerin döngüyü kontrol eden değişkenleri böldüğü veya çarptığı döngülerin verimliliğinin f(n) = log n olarak verilebileceği sonucuna varabiliriz.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

İç içe döngüler

- Döngüler içeren döngülere iç içe döngüler denir.
- İç içe döngüleri analiz edebilmek için her döngünün kaç yineleme tamamladığını belirlememiz gerekir.
- Toplam daha sonra iç döngüdeki yineleme sayısı ile dış döngüdeki yineleme sayısının çarpımı olarak elde edilir. Bu durumda, algoritmanın verimliliğini doğrusal logaritmik, ikinci dereceden veya bağımlı ikinci dereceden iç içe geçmiş döngü olup olmadığına göre analiz ederiz.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

Doğrusal logaritmik döngü

- İç döngünün döngüyü kontrol eden değişkeninin her yinelemeden sonra çarpıldığı aşağıdaki kodu ele alalım.
- İç döngüdeki yineleme sayısı log 10'dur.
- Bu iç döngü, 10 kez yinelenen bir dış döngü tarafından kontrol edilir.
- Dolayısıyla formüle göre bu kod için yineleme sayısı 10 log 10 olarak verilebilir.

```
i=0;i<10;i++) için
    j=1; j<10;j*=2) ifadesi bloğu;
```

- Daha genel terimlerle, bu tür döngülerin verimliliği f(n) = n log n olarak verilebilir.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

İkinci dereceden döngü

- İkinci dereceden bir döngüde, iç döngüdeki yineleme sayısı, dış döngüdeki yineleme sayısına eşittir.
- Dış döngünün 10 kez yürütüldüğü ve dış döngünün her yinelemesinde iç döngünün de 10 kez yürütüldüğü aşağıdaki kodu ele alalım.
- Dolayısıyla buradaki verim 100'dür.

```
    i=0;i<10;i++) için
        for(j=0; j<10;j++) ifadesi bloğu;
```

- İkinci dereceden döngünün genelleştirilmiş formülü f(n) = n2 olarak verilebilir.
- Bağımlı ikinci dereceden döngü Bağımlı ikinci dereceden döngüde, iç döngüdeki yineleme sayısı dış döngüye bağlıdır.

ZAMAN VE MEKÂN KARMAŞIKLIĞI

İkinci dereceden döngü

- Aşağıda verilen kodu ele alalım:

```
i=0;i<10;i++) için
    for(j=0; j<=i;j++) ifadesi bloğu;
```

- Bu kodda, iç döngü ilk yinelemede yalnızca bir kez, ikinci yinelemede iki kez, üçüncü yinelemede üç kez, vb. yürütülecektir.
- Bu şekilde yineleme sayısı 1 + 2 + 3 + ... + 9 + 10 = 55 olarak hesaplanabilir.
- Bu döngünün ortalamasını (55/10 = 5,5) hesaplarsak, bunun dış döngüdeki yineleme sayısı (10) artı 1 bölü 2'ye eşit olduğunu göreceğiz.
- Genel olarak iç döngü (n + 1)/2 kez yinelenir.
- Bu nedenle, böyle bir kodun verimliliği f(n) = n (n + 1)/2 olarak verilebilir